Arm® Architecture Reference Manual

for A-profile architecture

Document number	ARM DDI 0487
Document quality	EAC
Document version	M.b
Document confidentiality	Non-Confidential

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.

Release information

The following releases of this document have been made.

Date	Version	Changes
06/Apr/2026	M.b	• Updated Non-Confidential Armv9.6 EAC release, incorporating ASL1 and withdrawing FEAT_SEBEP
17/Dec/2025	M.a.a	• Non-Confidential Armv9.6 EAC release with event links fix. No other content changes.
01/Dec/2025	M.a	• Initial Non-Confidential Armv9.6 EAC release
30/Apr/2025	L.b	• Updated Non-Confidential Armv9.5 EAC release with the MPAM PE content rewritten as rules
30/Nov/2024	L.a	• Initial Non-Confidential Armv9.5 EAC release and separation of the RAS System Architecture into Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100)
20/Mar/2024	K.a	• Initial Non-Confidential Armv8.9 and Armv9.4 EAC release, incorporating MPAM and SME
21/Apr/2023	J.a	• Updated Non-Confidential Armv8.8 and Armv9.3 EAC release incorporating MEC, RAS, and RME
19/Aug/2022	I.a	• Updated Non-Confidential Armv8.8 and Armv9.3 EAC release incorporating BRBE, ETE, and TRBE
04/Feb/2022	H.a	• Initial Non-Confidential Armv8.8 and Armv9.3 EAC release, incorporating SVE
22/Jul/2021	G.b	• Updated Non-Confidential Armv8.7 EAC release
22/Jan/2021	G.a	• Initial Non-Confidential Armv8.7 EAC release
17/Jul/2020	F.c	• Initial Non-Confidential Armv8.6 EAC release
31/Mar/2020	F.b	• Non-Confidential Armv8.5 EAC, initial Armv8.6 Beta release
20/Feb/2020	F.a	• Initial Non-Confidential Armv8.6 Beta release
05/Jul/2019	E.a	• Initial Non-Confidential Armv8.5 EAC release
29/Apr/2019	D.b	• Updated Non-Confidential Armv8.4 EAC release incorporating accessibility changes
31/Oct/2018	D.a	• Initial Non-Confidential Armv8.4 EAC release
20/Dec/2017	C.a	• Initial Non-Confidential Armv8.3 EAC release
26/Sep/2017	B.b	• Initial Non-Confidential Armv8.2 EAC release, incorporating SPE
31/Mar/2017	B.a	• Initial Non-Confidential Armv8.1 EAC, v8.2 Beta release
30/Sep/2016	A.k	• Updated Non-Confidential EAC release
03/Jun/2016	A.j	• Non-Confidential EAC release
28/Jan/2016	A.i	• Ninth Non-Confidential Beta release
30/Sep/2015	A.h	• Eighth Non-Confidential Beta release
10/Jul/2015	A.g	• Seventh Non-Confidential Beta release
25/Mar/2015	A.f	• Sixth Non-Confidential Beta release
17/Dec/2014	A.e	• Fifth Non-Confidential Beta release
09/Oct/2014	A.d	• Fourth Non-Confidential Beta release
18/Jul/2014	A.c	• Third Non-Confidential Beta release
24/Dec/2013	A.b	• Second Non-Confidential Beta release
04/Sep/2013	A.a	• Non-Confidential Beta release
12/Jun/2013	A.a-2	• Second confidential Beta draft of first issue, limited circulation
30/Apr/2013	A.a-1	• Confidential Beta draft of first issue, limited circulation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-20349

8 March 2024

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

The information in this manual is at EAC quality, which means that all features of the specification are described in the manual.

Web Address

https://www.arm.com

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Arm® Architecture Reference Manual for A-profile architecture

Preface

About this Manual 23

Using this Manual 27

Conventions 34

Additional reading 37

Feedback 38

Part A — **Arm Architecture Introduction and Overview**

Chapter A1 — **Introduction to the Arm Architecture**

A1.1 About the Arm architecture 41

A1.2 Architecture profiles 43

A1.3 Arm architectural concepts 44

A1.4 Supported data types 47

A1.5 Floating-point support 63

A1.6 The Arm memory model 81

A1.7 Reliability, Availability, and Serviceability 82

Chapter A2 — **A-profile Architecture Extensions**

A2.1 About the A-profile architecture extensions 86

A2.2 Armv8-A architecture extensions 87

A2.3 Armv9-A architecture extensions 163

Part B — **The AArch64 Application Level Architecture**

Chapter B1 — **The AArch64 Application Level Programmers' Model**

B1.1 About the Application level programmers' model 203

B1.2 Registers in AArch64 Execution state 204

B1.3 Process state, PSTATE 208

B1.4 The Scalable Vector and Scalable Matrix Extensions (SVE & SME) 211

B1.5 Software control features and EL0 240

Chapter B2 — **The AArch64 Application Level Memory Model**

B2.1 About the Arm memory model 246

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2 Atomicity in the Arm architecture 248
B2.3 Ordering requirements defined by the formal concurrency model 253
B2.4 Additional ordering requirements outside of the scope of the formal concurrency model 285
B2.5 Restrictions on the effects of speculation 290
B2.6 Memory barriers 297
B2.7 Caches and memory hierarchy 305
B2.8 Alignment support 310
B2.9 Endian support 313
B2.10 Memory types and attributes 317
B2.11 Mismatched memory attributes 329
B2.12 Synchronization and semaphores 332

Part C The AArch64 Instruction Set

Chapter C1 The A64 Instruction Set

C1.1 About the A64 instruction set 345
C1.2 Structure of the A64 assembler language 346
C1.3 Address generation 352
C1.4 Instruction aliases 355

Chapter C2 About the A64 Instruction Descriptions

C2.1 Understanding the A64 instruction descriptions 357
C2.2 General information about the A64 instruction descriptions 360

Chapter C3 A64 Instruction Set Overview

C3.1 Branches, Exception generating, and System instructions 364
C3.2 Loads and stores 377
C3.3 Loads and stores - SVE, SVE2 409
C3.4 Loads and stores - SME, SME2, SVE2p1 419
C3.5 Data processing - immediate 424
C3.6 Data Processing - MTE 430
C3.7 Data processing - register 431
C3.8 Data processing - SIMD and floating-point 440
C3.9 Data processing - SVE 469
C3.10 Data processing - SVE2 494
C3.11 Data processing - SME, SME2 522

Chapter C4 A64 Instruction Set Encoding

C4.1 A64 instruction set encoding 544

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C5 **The A64 System Instruction Class**

C5.1 The System instruction class encoding space 915

C5.2 Special-purpose registers 936

C5.3 A64 System instructions for cache maintenance 1068

C5.4 A64 System instructions for address translation 1155

C5.5 A64 System instructions for TLB maintenance 1187

C5.6 A64 System instructions for prediction restriction 1740

C5.7 A64 System instructions for the Branch Record Buffer Extension 1757

C5.8 A64 System instructions for the Trace Extension 1762

C5.9 A64 System instructions for the Guarded Control Stack 1764

C5.10 A64 System instructions introduced by FEAT_RME_GPC3 1778

C5.11 A64 IMPLEMENTATION DEFINED System instructions 1781

Chapter C6 **A64 Base Instruction Descriptions**

C6.1 About the A64 base instructions 1786

C6.2 Alphabetical list of A64 base instructions 1789

Chapter C7 **A64 Advanced SIMD and Floating-point Instruction Descriptions**

C7.1 About the A64 Advanced SIMD and floating-point instructions 2918

C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions 2921

Chapter C8 **SVE Instruction Descriptions**

C8.1 About the SVE instructions 4097

C8.2 Alphabetical list of SVE instructions 4098

Chapter C9 **SME Instruction Descriptions**

C9.1 About the SME instructions 6231

C9.2 Alphabetical list of SME instructions 6232

Part D **The AArch64 System Level Architecture**

Chapter D1 **The AArch64 System Level Programmers' Model**

D1.1 Exception levels 7192

D1.2 Registers for instruction processing and exception handling 7197

D1.3 System register locking 7199

D1.4 Exceptions 7201

D1.5 Process state, PSTATE 7246

D1.6 Resets and power domains 7249

D1.7 Mechanisms for entering a low-power state 7252

D1.8 Self-hosted debug 7257

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.9 Event monitors 7258
D1.10 Interprocessing 7259
D1.11 Check Feature 7265

Chapter D2 AArch64 Self-hosted Debug

D2.1 About self-hosted debug 7267
D2.2 Routing debug exceptions 7271
D2.3 The debug exception enable controls 7273
D2.4 The effect of powerdown on debug exceptions 7275
D2.5 Summary of the routing and enabling of debug exceptions 7276
D2.6 Pseudocode description of debug exceptions 7278
D2.7 Breakpoint Instruction exceptions 7279
D2.8 Breakpoint exceptions 7281
D2.9 Watchpoint exceptions 7304
D2.10 Vector Catch exceptions 7322
D2.11 Software Step exceptions 7323
D2.12 Synchronization and debug exceptions 7338

Chapter D3 AArch64 Self-hosted Trace

D3.1 About self-hosted trace 7340
D3.2 Prohibited regions in self-hosted trace 7341
D3.3 Self-hosted trace timestamps 7343
D3.4 Synchronization in self-hosted trace 7344

Chapter D4 The Embedded Trace Extension

D4.1 About the Embedded Trace Extension 7346
D4.2 Programmers' model 7353
D4.3 Trace elements 7357
D4.4 Instruction and exception classification 7372
D4.5 About the ETE trace unit 7381
D4.6 Resource operation 7421

Chapter D5 ETE Protocol Descriptions

D5.1 About the ETE protocol 7451
D5.2 Summary list of ETE packets 7453
D5.3 Alphabetical list of ETE packets 7456

Chapter D6 The Trace Buffer Extension

D6.1 About the Trace Buffer Extension 7578
D6.2 The trace buffer 7581

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.3 Trace buffer Self-hosted mode 7586
D6.4 Trace buffer External mode 7594
D6.5 Trace buffer management 7599
D6.6 Synchronization and the Trace Buffer Unit 7614
D6.7 Trace synchronization and memory barriers 7618
D6.8 Trace of Speculative execution 7619
D6.9 Trace in Debug state 7620
D6.10 Synchronization litmus tests 7621
D6.11 UNPREDICTABLE behavior 7625

Chapter D7 The AArch64 System Level Memory Model

D7.1 About the memory system architecture 7628
D7.2 Address space 7629
D7.3 Mixed-endian support in AArch64 7630
D7.4 Memory Encryption Contexts 7631
D7.5 Cache support 7633
D7.6 External aborts 7663
D7.7 Memory barrier instructions 7665
D7.8 Pseudocode description of general memory System instructions 7666

Chapter D8 The AArch64 Virtual Memory System Architecture

D8.1 Address translation 7670
D8.2 Translation process 7686
D8.3 Translation table descriptor formats 7724
D8.4 Memory access control 7744
D8.5 Hardware updates to the translation tables 7770
D8.6 Memory region attributes 7785
D8.7 Other descriptor fields 7794
D8.8 Address tagging 7799
D8.9 Logical Address Tagging 7801
D8.10 Pointer authentication 7803
D8.11 Checked Pointer Arithmetic 7809
D8.12 Memory Encryption Contexts extension 7811
D8.13 Virtualization Host Extensions 7814
D8.14 Nested virtualization 7821
D8.15 Memory aborts 7829
D8.16 Translation Lookaside Buffers 7842
D8.17 TLB maintenance 7847

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.18 Caches 7866

D8.19 Pseudocode description of VMSA address translation 7868

Chapter D9 The Granule Protection Check Mechanism

D9.1 GPC behavior overview 7871

D9.2 GPC bypass windows 7872

D9.3 Granular Data Isolation 7873

D9.4 GPC faults 7874

D9.5 GPT caching and invalidation 7876

D9.6 GPT formats 7878

D9.7 GPT lookup process 7883

D9.8 Explicit PAS control 7885

Chapter D10 The Memory Tagging Extension

D10.1 Introduction 7887

D10.2 Allocation Tags 7888

D10.3 Memory region tagging types 7889

D10.4 Tag checking 7891

D10.5 Allocation Tag Access controls 7893

D10.6 Physical Tag locations 7894

D10.7 Tag Check Faults 7896

Chapter D11 The Guarded Control Stack

D11.1 Introduction 7899

D11.2 The Guarded Control Stack 7900

D11.3 Procedure returns 7902

D11.4 Exception returns 7904

D11.5 Stage 1 permission model 7908

D11.6 Stage 2 Permission model 7910

D11.7 Guarded Control Stack switching 7911

D11.8 GCS record for software use 7914

D11.9 Guarded Control Stack exceptions 7915

D11.10 Guarded Control Stack data accesses 7916

D11.11 Detecting when FEAT_GCS is enabled 7922

Chapter D12 The Generic Timer in AArch64 state

D12.1 About the Generic Timer 7924

D12.2 The AArch64 view of the Generic Timer 7929

Chapter D13 The Performance Monitors Extension

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.1 About the Performance Monitors 7938
D13.2 Accuracy of the Performance Monitors 7942
D13.3 Behavior on overflow 7944
D13.4 Attributability 7949
D13.5 Controlling the PMU counters 7951
D13.6 Multithreaded implementations 7956
D13.7 Event filtering 7958
D13.8 Event threshold and edge counting 7960
D13.9 PMU snapshots 7965
D13.10 Performance Monitors and Debug state 7969
D13.11 Counter access 7970
D13.12 Performance Monitors Extension registers 7972

Chapter D14 PMU Event Descriptions

D14.1 Definitions 7974
D14.2 The PMU event number space and common events 7986
D14.3 Common event numbers 8040
D14.4 Cycle event counting 8244
D14.5 Meaningful combinations of common events 8246
D14.6 Required events 8251
D14.7 IMPLEMENTATION DEFINED event numbers 8253

Chapter D15 The System Performance Monitors Extension

D15.1 About the System Performance Monitors 8255
D15.2 System PMU configuration 8256
D15.3 Accessing System PMUs 8258
D15.4 Generating System PMU overflow interrupt requests 8260

Chapter D16 The Activity Monitors Extension

D16.1 About the Activity Monitors Extension 8262
D16.2 Properties and behavior of the activity monitors 8263
D16.3 AMU events and event numbers 8265

Chapter D17 The Statistical Profiling Extension

D17.1 About the Statistical Profiling Extension 8267
D17.2 Defining the sample population 8269
D17.3 Controlling when an operation is sampled 8270
D17.4 Enabling profiling 8273
D17.5 Filtering sample records 8275
D17.6 The profiling data 8278

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.7 The Profiling Buffer 8292
D17.8 Profiling Buffer management 8299
D17.9 Synchronization and Statistical Profiling 8310

Chapter D18 Statistical Profiling Extension Sample Record Specification

D18.1 About the Statistical Profiling Extension sample records 8313
D18.2 Alphabetical list of Statistical Profiling Extension packets 8316

Chapter D19 The Branch Record Buffer Extension

D19.1 About the Branch Record Buffer Extension 8360
D19.2 Branch record filtering 8365
D19.3 Branch record buffer operation 8370
D19.4 Branch record buffer 8373
D19.5 Programmers' model 8375

Chapter D20 RAS PE Architecture

D20.1 About the RAS Extension 8378
D20.2 PE error handling 8380
D20.3 Generating error exceptions 8385
D20.4 Taking error exceptions 8386
D20.5 Error synchronization event 8395
D20.6 Virtual SError exceptions and delegated SError exceptions 8402
D20.7 Error records in the PE 8403

Chapter D21 MPAM PE Architecture

D21.1 About MPAM 8408
D21.2 PARTID spaces 8412
D21.3 Partition Identifiers (PARTIDs) 8416
D21.4 Performance Monitoring Groups (PMGs) 8419
D21.5 Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on PE Secure state execution 8420
D21.6 Effect of access type on MPAM information 8422
D21.7 Virtualization support 8424
D21.8 PE-side Maximum bandwidth limit injection control 8427
D21.9 MPAM errors 8428
D21.10 Synchronization of MPAM System register changes 8429

Chapter D22 The Scalable Matrix Extension

D22.1 Overview 8431
D22.2 SME traps and exceptions 8432
D22.3 Validity of SME and SVE state 8433

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.4 Streaming execution priority 8434
D22.5 Floating-point behaviors in Streaming SVE mode 8436
D22.6 Floating-point behaviors for instructions that target the SME ZA array 8437
D22.7 Security and power considerations 8438

Chapter D23 AArch64 System Register Encoding

D23.1 The System register encoding space 8440
D23.2 Moves to and from debug and trace System registers 8441
D23.3 Moves to and from non-debug System registers, Special-purpose registers 8446

Chapter D24 AArch64 System Register Descriptions

D24.1 About the AArch64 System registers 8464
D24.2 General system control registers 8474
D24.3 Debug System registers 10166
D24.4 Trace registers 10334
D24.5 Performance Monitors registers 10676
D24.6 Activity Monitors registers 10925
D24.7 Statistical Profiling Extension registers 10971
D24.8 Branch Record Buffer Extension registers 11122
D24.9 RAS registers 11175
D24.10 Generic Timer registers 11235
D24.11 Guarded Control Stack registers 11368
D24.12 MPAM registers 11400

Part E The AArch32 Application Level Architecture

Chapter E1 The AArch32 Application Level Programmers' Model

E1.1 About the Application level programmers' model 11504
E1.2 The Application level programmers' model in AArch32 state 11505
E1.3 Advanced SIMD and floating-point instructions 11518
E1.4 About the AArch32 System register interface 11536
E1.5 Exceptions 11537

Chapter E2 The AArch32 Application Level Memory Model

E2.1 About the Arm memory model 11539
E2.2 Atomicity in the Arm architecture 11541
E2.3 Definition of the memory model 11545
E2.4 Ordering of translation table walks 11555
E2.5 Caches and memory hierarchy 11556
E2.6 Alignment support 11561

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.7 Endian support 11563
E2.8 Memory types and attributes 11567
E2.9 Mismatched memory attributes 11577
E2.10 Synchronization and semaphores 11580

Part F The AArch32 Instruction Sets

Chapter F1 About the T32 and A32 Instruction Descriptions

F1.1 Format of instruction descriptions 11592
F1.2 Standard assembler syntax fields 11597
F1.3 Conditional execution 11598
F1.4 Shifts applied to a register 11600
F1.5 Memory accesses 11602
F1.6 Encoding of lists of general-purpose registers and the PC 11603
F1.7 General information about the T32 and A32 instruction descriptions 11604
F1.8 Additional pseudocode support for instruction descriptions 11618
F1.9 Additional information about Advanced SIMD and floating-point instructions 11619

Chapter F2 The AArch32 Instruction Sets Overview

F2.1 Support for instructions in different versions of the Arm architecture 11626
F2.2 Unified Assembler Language 11627
F2.3 Branch instructions 11629
F2.4 Data-processing instructions 11630
F2.5 PSTATE and banked register access instructions 11638
F2.6 Load/store instructions 11639
F2.7 Load/store multiple instructions 11642
F2.8 Miscellaneous instructions 11643
F2.9 Exception-generating and exception-handling instructions 11644
F2.10 System register access instructions 11646
F2.11 Advanced SIMD and floating-point load/store instructions 11647
F2.12 Advanced SIMD and floating-point register transfer instructions 11649
F2.13 Advanced SIMD data-processing instructions 11650
F2.14 Floating-point data-processing instructions 11661

Chapter F3 T32 Instruction Set Encoding

F3.1 T32 instruction set encoding 11664
F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding 11742

Chapter F4 A32 Instruction Set Encoding

F4.1 A32 instruction set encoding 11744

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding 11812

Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions

F5.1 Alphabetical list of T32 and A32 base instruction set instructions 11814

F5.2 Encoding and use of banked register transfer instructions 12593

Chapter F6 T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

F6.1 Alphabetical list of Advanced SIMD and floating-point instructions 12598

Part G The AArch32 System Level Architecture

Chapter G1 The AArch32 System Level Programmers' Model

G1.1 About the AArch32 System level programmers' model 13457

G1.2 Exception levels 13458

G1.3 Exception terminology 13459

G1.4 Execution state 13461

G1.5 Instruction Set state 13463

G1.6 Security state 13464

G1.7 Security state, Exception levels, and AArch32 execution privilege 13467

G1.8 Virtualization 13469

G1.9 AArch32 state PE modes 13471

G1.10 AArch32 general-purpose registers, the PC, and the Special-purpose registers 13477

G1.11 Process state, PSTATE 13480

G1.12 Instruction set states 13486

G1.13 Handling exceptions that are taken to an Exception level using AArch32 13488

G1.14 Routing of aborts taken to AArch32 state 13506

G1.15 Exception return to an Exception level using AArch32 13508

G1.16 Asynchronous exception behavior for exceptions taken from AArch32 state 13513

G1.17 AArch32 state exception descriptions 13521

G1.18 Reset into AArch32 state 13543

G1.19 Mechanisms for entering a low-power state 13545

G1.20 The AArch32 System register interface 13550

G1.21 Advanced SIMD and floating-point support 13553

G1.22 Configurable instruction controls 13557

Chapter G2 AArch32 Self-hosted Debug

G2.1 About self-hosted debug 13563

G2.2 Routing debug exceptions 13567

G2.3 The debug exception enable controls 13569

G2.4 The effect of powerdown on debug exceptions 13571

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.5 Summary of permitted routing and enabling of debug exceptions 13572
G2.6 Pseudocode description of debug exceptions 13574
G2.7 Breakpoint Instruction exceptions 13575
G2.8 Breakpoint exceptions 13578
G2.9 Watchpoint exceptions 13602
G2.10 Vector Catch exceptions 13615
G2.11 Synchronization and debug exceptions 13623

Chapter G3 AArch32 Self-hosted Trace

G3.1 About self-hosted trace 13626
G3.2 Prohibited regions in self-hosted trace 13628
G3.3 Self-hosted trace timestamps 13629
G3.4 Synchronization in self-hosted trace 13630

Chapter G4 The AArch32 System Level Memory Model

G4.1 About the memory system architecture 13632
G4.2 Address space 13633
G4.3 Mixed-endian support in AArch32 13634
G4.4 AArch32 cache and branch predictor support 13635
G4.5 System register support for IMPLEMENTATION DEFINED memory features 13659
G4.6 External aborts 13660
G4.7 Memory barrier instructions 13662
G4.8 Pseudocode description of general memory System instructions 13663

Chapter G5 The AArch32 Virtual Memory System Architecture

G5.1 About VMSAv8-32 13667
G5.2 The effects of disabling address translation stages on VMSAv8-32 behavior 13675
G5.3 Translation tables 13679
G5.4 The VMSAv8-32 Short-descriptor translation table format 13684
G5.5 The VMSAv8-32 Long-descriptor translation table format 13693
G5.6 Memory access control 13712
G5.7 Memory region attributes 13723
G5.8 Translation Lookaside Buffers 13735
G5.9 TLB maintenance requirements 13739
G5.10 Caches in VMSAv8-32 13753
G5.11 VMSAv8-32 memory aborts 13756
G5.12 Exception reporting in a VMSAv8-32 implementation 13768
G5.13 Address translation instructions 13786
G5.14 Pseudocode description of VMSAv8-32 memory system operations 13792

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.15 About the System registers for VMSAv8-32 13795
G5.16 Functional grouping of VMSAv8-32 System registers 13800

Chapter G6 The Generic Timer in AArch32 state

G6.1 About the Generic Timer in AArch32 state 13802
G6.2 The AArch32 view of the Generic Timer 13807

Chapter G7 AArch32 System Register Encoding

G7.1 The AArch32 System register encoding space 13816
G7.2 Organization of registers in the (`coproc` == `0b1110`) encoding space 13817
G7.3 Organization of registers in the (`coproc` == `0b1111`) encoding space 13820

Chapter G8 AArch32 System Register Descriptions

G8.1 About the AArch32 System registers 13834
G8.2 General system control registers 13849
G8.3 Debug System registers 14460
G8.4 Performance Monitors registers 14622
G8.5 Activity Monitors registers 14738
G8.6 RAS registers 14785
G8.7 Generic Timer registers 14862

Part H External Debug

Chapter H1 About External Debug

H1.1 Introduction to external debug 14965
H1.2 External debug 14967
H1.3 Required debug authentication 14968

Chapter H2 Debug State

H2.1 About Debug state 14971
H2.2 Halting the PE on debug events 14972
H2.3 Entering Debug state 14979
H2.4 Behavior in Debug state 14982
H2.5 Exiting Debug state 15016

Chapter H3 Halting Debug Events

H3.1 Introduction to Halting debug events 15019
H3.2 Halting Step debug events 15021
H3.3 Halt Instruction debug event 15031
H3.4 Exception Catch debug event 15032
H3.5 External Debug Request debug event 15036

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.6 OS Unlock Catch debug event 15039
H3.7 Reset Catch debug events 15040
H3.8 Software Access debug event 15041
H3.9 Synchronization and Halting debug events 15042

Chapter H4 The Debug Communication Channel and Instruction Transfer Register

H4.1 Introduction 15045
H4.2 DCC and ITR registers 15046
H4.3 DCC and ITR access modes 15050
H4.4 Flow control of the DCC and ITR registers 15054
H4.5 Synchronization of DCC and ITR accesses 15058
H4.6 Interrupt-driven use of the DCC 15063
H4.7 Pseudocode description of the operation of the DCC and ITR registers 15064

Chapter H5 The Embedded Cross-Trigger Interface

H5.1 About the Embedded Cross-Trigger 15066
H5.2 Basic operation on the ECT 15068
H5.3 Cross-triggers on a PE in an Arm A-profile implementation 15072
H5.4 Description and allocation of CTI triggers 15073
H5.5 CTI registers programmers' model 15077
H5.6 Examples 15078

Chapter H6 Debug Reset and Powerdown Support

H6.1 About Debug over powerdown 15082
H6.2 Power domains and debug 15083
H6.3 Core power domain power states 15084
H6.4 Powerup request mechanism 15086
H6.5 Emulating low-power states 15087
H6.6 Debug OS Save and Restore sequences 15089
H6.7 Reset and debug 15095

Chapter H7 The PC Sample-based Profiling Extension

H7.1 About the PC Sample-based Profiling Extension 15098

Chapter H8 About the External Debug Registers

H8.1 Relationship between external debug and System registers 15103
H8.2 Endianness and supported access sizes 15104
H8.3 Synchronization of changes to the external debug registers 15105
H8.4 Memory-mapped accesses to the external debug interface 15109
H8.5 External debug interface register access permissions 15111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.6 External debug interface registers 15115
H8.7 Cross-trigger interface registers 15121
H8.8 External debug register resets 15123

Chapter H9 External Debug Register Descriptions

H9.1 About the external debug registers 15127
H9.2 External debug registers 15128
H9.3 External trace registers 15289
H9.4 External Trace Buffer registers 15483
H9.5 Cross-Trigger Interface registers 15552

Part I Memory-mapped Components of the Arm Architecture

Chapter I1 Requirements for Memory-mapped Components

I1.1 Supported access sizes 15618
I1.2 Synchronization of memory-mapped registers 15620
I1.3 Access requirements for reserved and unallocated registers 15622

Chapter I2 System Level Implementation of the Generic Timer

I2.1 About the Generic Timer specification 15624
I2.2 Memory-mapped counter module 15626
I2.3 Memory-mapped timer components 15630

Chapter I3 Recommended External Interface to the Performance Monitors

I3.1 About the external interface to the Performance Monitors registers 15635

Chapter I4 Recommended External Interface to the Activity Monitors

I4.1 About the external interface to the Activity Monitors Extension registers 15643

Chapter I5 External System Control Register Descriptions

I5.1 About the external system control register descriptions 15646
I5.2 External Performance Monitors registers summary 15648
I5.3 Performance Monitors external register descriptions 15653
I5.4 External Activity Monitors Extension registers summary 15828
I5.5 Activity Monitors external register descriptions 15831
I5.6 Generic Timer memory-mapped registers overview 15901
I5.7 Generic Timer memory-mapped register descriptions 15902

Part J Architectural Pseudocode

Chapter J1 A-profile Architecture Pseudocode

J1.1 Standard library 15955

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2 Pseudocode for AArch64 operation 15957
J1.3 Pseudocode for AArch32 operation 16350
J1.4 Shared pseudocode 16468

Part K Appendixes

Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors 16801
K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors 16822

Appendix K2 Recommendations for Reporting Memory Attributes on an Interconnect

K2.1 Arm recommendations for reporting memory attributes on an interconnect 16843

Appendix K3 ETE Recommended Configurations

K3.1 Configurations 16845

Appendix K4 Stages of Execution

K4.1 Stages of execution 16847

Appendix K5 Recommended External Debug Interface

K5.1 About the recommended external debug interface 16849
K5.2 PMUEVENT bus 16853
K5.3 Recommended authentication interface 16854
K5.4 Management registers and CoreSight compliance 16856

Appendix K6 Additional Information for Implementations of the Generic Timer

K6.1 Providing a complete set of features in a system level implementation 16867
K6.2 Gray count scheme for timer distribution scheme 16869

Appendix K7 Legacy Instruction Syntax for AArch32 Instruction Sets

K7.1 Legacy Instruction Syntax 16871

Appendix K8 Address Translation Examples

K8.1 AArch64 Address translation examples 16878
K8.2 AArch32 Address translation examples 16892

Appendix K9 Recommended Upload and Download Processes for External Debug

K9.1 Using Memory access mode in AArch64 state 16902

Appendix K10 Software Usage Examples

K10.1 Use of the Advanced SIMD complex number instructions 16907
K10.2 Use of the Armv8.2 extensions to the Cryptographic Extension 16909
K10.3 SVE and SVE2 Programming Examples 16916

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.4 Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) hint instructions 16917
K10.5 Use of the Embedded Trace and Trace Buffer Extensions 16919
K10.6 Use of the GCS Extension 16981
K10.7 Use of the RAS Extension 16991
K10.8 Example OS Save and Restore sequences 16992
K10.9 RME example software flows 16995
K10.10 MEC example software flows 17000

Appendix K11 Barrier Litmus Tests

K11.1 Introduction 17006
K11.2 Load-Acquire, Store-Release and barriers 17009
K11.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers 17012
K11.4 Using a mailbox to send an interrupt 17017
K11.5 Cache and TLB maintenance instructions and barriers 17018

Appendix K12 Random Number Generation

K12.1 Properties of the generated random number 17031

Appendix K13 Arm Pseudocode Definition

K13.1 About the Arm pseudocode 17033
K13.2 Pseudocode for instruction descriptions 17034

Appendix K14 Registers Index

K14.1 Introduction and register disambiguation 17036
K14.2 Alphabetical index of AArch64 registers and System instructions 17047
K14.3 Functional index of AArch64 registers and System instructions 17067
K14.4 Alphabetical index of AArch32 registers and System instructions 17087
K14.5 Functional index of AArch32 registers and System instructions 17096
K14.6 Alphabetical index of memory-mapped registers 17106
K14.7 Functional index of memory-mapped registers 17119

Glossary

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm Architecture Reference Manual, for A-profile architecture*. It contains the following sections:

- About this Manual.
- Using this Manual.
- Conventions.
- Additional reading.
- Feedback.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this Manual

This Manual describes the Arm® A-profile architecture versions Armv8-A and Armv9-A.

This Manual is also available in HTML on the Arm Developer website, at https://developer.arm.com/documentation/ddi0487/mb. The HTML version is currently an early-access preview, provided for convenience only. In the event of any difference between this PDF and the HTML version, this PDF is the authoritative reference.

Download this PDF from https://developer.arm.com/documentation/ddi0487/mb.

The Architecture Speaks chatbot, available at https://developer.arm.com/architecture/the-architecture-speaks, aims to answer questions about the *Arm® Architecture Reference Manual for A-profile architecture*, and point users to sources within it that justify the answers.

Known limitations of this release

Validation of this release has identified the following issues that Arm will address in future issues:

- The A64 Instruction Set Encoding is not yet included in the HTML version. It is currently only available in the PDF version.
- The pseudocode in Architectural Pseudocode conforms to the definitions of ASL1, which is described in *Arm® Architecture Specification Language (ASL) Reference* (ARM DDI 0626). Other content in this Manual will be updated to align with ASL1 in future releases.
- Some diagrams in the register descriptions chapter have long field names that split over several lines.

For a monthly updated list of known issues in this release, see the Arm® Architecture Reference Manual for A-profile architecture, Known Issues.

For release notes corresponding to the most recent Register and Instruction XML release, see https://developer.arm.com/Architectures/A-Profile%20Architecture#Downloads.

Corresponding Collateral

The following packages provide additional information, including machine-readable data, that is aligned to this release:

Name	Architecture version	Architecture quality	Package version	Package format	Description & Links
Machine-Readable Descriptions for Features, Registers, and Instructions.	up to 9.6	EAC	2026-03	XML & HTML	The following packages provide machine-readable descriptions for Armv8.9-A and Armv9.6-A content: • Features • Registers • Instructions (A64) • Instructions (A32/T32)
Architecture Specification Language (ASL) Definition	N/A	N/A	00eac0	PDF	*Architecture Specification Language* (ASL) is used to specify system registers and instruction descriptions in the *Arm® Architecture Reference Manual for A-profile architecture* and in the other collateral listed in this table. The following PDFs provide information on ASL: • ASL Reference (DDI 0626) version 00eac0, the formal language definition for ASL. • ASL Reader's Guide (111069) version A.a, an informal guide to reading ASL code. See Architecture Specification Language.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Architecture version	Architecture quality	Package version	Package format	Description & Links
Architecture Specification Language (ASL) Ecosystem	N/A	N/A	00eac0	Source files & tools	*ASLRef* is the reference definition ecosystem for ASL, and is available in a publicly accessible repository that contains all of: • Source code for Arm's reference implementation of ASL. • Tests that illustrate the ASL language, including those that also appear in ASL Reference (DDI 0626), the formal language definition for ASL. • The source files used to produce ASL Reference (DDI 0626). The ASLRef version corresponding to this release is *ASLRef EAC0 release*.
cat Language Definition	N/A	N/A	Latest	PDF	*cat* is a domain specific language used to specify the Arm memory model. The following PDF provides information on cat: • Syntax and semantics of the weak consistency model specification language cat.
cat file (Machine-Readable, Executable and Formal Concurrency Model Description).	N/A	EAC	Latest	Source files	*cat file* (Machine-Readable, Executable, and Formal Concurrency Model Description). • aarch64.cat contains the definitions and ordering rules that correspond to the section Ordering requirements defined by the formal concurrency model. The herdtools7 version corresponding to this release is *ArmARM-M.b*.
Concurrency Model Ecosystem	N/A	EAC	Latest	Source files & tools	*herdtools7* is a tool suite used to develop, simulate, and test the Arm Concurrency Model, among others. It is available in a publicly accessible repository that contains all of: • The source code for herd7, the concurrency model simulation tool. • The source code for litmus7, a tool that generates hardware-executable binaries from litmus tests. • The source code for diy7/diyone7, tools that generate litmus tests. • A collection of litmus tests illustrating the ordering requirements defined in the section Ordering requirements defined by the formal concurrency model. The herdtools7 version corresponding to this release is *ArmARM-M.b*. See also: • herd7 Tool.

Architecture description

The Arm® A-profile architecture describes the operation of an Armv8-A and an Armv9-A *Processing element* (PE). This Manual includes descriptions of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The two Execution states, AArch64 and AArch32.
- The instruction sets:
  - In AArch32 state, the T32 and A32 instruction sets, which are compatible with earlier versions of the Arm architecture.
  - In AArch64 state, the A64 instruction set.
- The states that determine how a PE operates, including the current Exception level and Security state, and in AArch32 state the PE mode.
- The Exception model.
- The interprocessing model, that supports transitioning between AArch64 state and AArch32 state.
- The memory model, that defines memory ordering and memory management. This Manual covers the Arm A architecture profile, both Armv8-A and Armv9-A, that defines a *Virtual Memory System Architecture* (VMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, which provide high-performance:
  - Single-precision, half-precision, and double-precision floating-point operations.
  - Conversions between double-precision, single-precision, and half-precision floating-point values.
  - Integer, single-precision floating-point, and half-precision floating-point vector operations in all instruction sets.
  - Double-precision floating-point vector operations in the A64 instruction set.
- The security model, which provides up to four Security states to support Secure applications and confidential computing.
- The virtualization model.
- The Debug architecture, which provides software access to debug features.

This Manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this Manual is not a tutorial for Arm assembler language, nor does it describe Arm assembler language, except at a basic level. To make effective use of Arm assembler language, read the documentation supplied with the assembler being used.

This Manual is organized into parts:

Part A Provides an introduction to the Arm architecture, and an overview of the AArch64 and AArch32 Execution states.

Part B Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers' model and the memory model.

Part C Describes the A64 instruction set, which is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as *unprivileged* execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Part D Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers' model and the memory model. This part includes the description of self-hosted debug.

Part E Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers' model and the memory model.

Note

In AArch32 state, execution at EL0 is execution in User mode.

Part F Describes the T32 and A32 instruction sets, which are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the Arm architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

User mode is the only mode where software execution is unprivileged.

Part G Describes the system level view of the AArch32 Execution state, which is generally compatible with earlier versions of the Arm architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the instruction interface to those registers. It also describes the system level view of the programmers' model and the memory model.

Part H Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Part I Describes additional features of the architecture that are not closely coupled to a PE, and therefore are accessed through memory-mapped interfaces. Some of these features are OPTIONAL.

Part J Provides pseudocode that describes various features of the Arm architecture.

Part K, Appendixes

Provide additional information. Some appendixes give information that is not part of the A-profile architectural requirements. The cover page of each appendix indicates its status.

Glossary Defines terms used in this Manual that have a specialized meaning.

Note

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Using this Manual

The information in this Manual is organized into parts, as described in this section.

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-A and Armv9-A architecture profiles, including its relationship to the other Arm PE architectures. It introduces the terminology used to describe the architecture, and gives an overview of the Executions states, AArch64 and AArch32.

It contains the following chapters:

Introduction to the Arm Architecture

Read this for an introduction to the Arm architecture.

A-profile Architecture Extensions

Read this for an introduction to the A-profile architecture extensions and features.

Part B, The AArch64 Application Level Architecture

Part B describes the AArch64 state application level view of the architecture. It contains the following chapters:

The AArch64 Application Level Programmers' Model

Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

The AArch64 Application Level Memory Model

Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The A64 Instruction Set

Part C describes the A64 instruction set, which is used in AArch64 state. It contains the following chapters:

The A64 Instruction Set

Read this for a description of the A64 instruction set and common instruction operation details.

About the A64 Instruction Descriptions

Read this to understand the format of the A64 instruction descriptions.

A64 Instruction Set Overview

Read this for an overview of the A64 instructions.

A64 Instruction Set Encoding

Read this for a description of the A64 instruction set encoding.

The A64 System Instruction Class

Read this for a description of the AArch64 System instructions and register descriptions, and the System instruction class encoding space.

A64 Base Instruction Descriptions

Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A64 Advanced SIMD and Floating-point Instruction Descriptions

Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

SVE Instruction Descriptions

Read this for information on key aspects of the SVE instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

SME Instruction Descriptions

Read this for information on key aspects of the SME instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Part D, The AArch64 System Level Architecture

Part D describes the AArch64 state system level view of the architecture. It contains the following chapters:

The AArch64 System Level Programmers' Model

Read this for a description of the AArch64 state system level view of the programmers' model.

AArch64 Self-hosted Debug

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

AArch64 Self-hosted Trace

Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

The Embedded Trace Extension

Read this for a description of the Embedded Trace Extension.

ETE Protocol Descriptions

Read this for a description of the ETE packets, which are listed in alphabetical order.

The Trace Buffer Extension

Read this for a description of the Trace Buffer Extension.

The AArch64 System Level Memory Model

Read this for a description of the AArch64 state system level view of the general features of the memory system.

The AArch64 Virtual Memory System Architecture

Read this for a system level view of the AArch64 *Virtual Memory System Architecture* (VMSA), the memory system architecture of an A-profile architecture implementation executing in AArch64 state.

The Granule Protection Check Mechanism

Read this for a description of the granule protection check mechanism, a feature that is added as part of the OPTIONAL Realm Management Extension.

The Memory Tagging Extension

Read this for a description of the Memory Tagging Extension.

The Guarded Control Stack

Read this for a description of the Guarded Control Stack extension.

The Generic Timer in AArch64 state

Read this for a description of the AArch64 view of the Arm Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Performance Monitors Extension

Read this for a description of the Arm Performance Monitors, an optional non-invasive debug component.

The System Performance Monitors Extension

Read this for a description of the System Performance Monitors, an optional non-invasive debug component.

The Activity Monitors Extension

Read this for a description of the Arm Activity Monitors.

The Statistical Profiling Extension

Read this for a description of the Statistical Profiling Extension, an optional AArch64 state non-invasive debug component.

Statistical Profiling Extension Sample Record Specification

Read this for a description of the sample records generated by the Statistical Profiling Extension.

The Branch Record Buffer Extension

Read this for a description of the Branch Record Buffer Extension.

RAS PE Architecture

Read this for a description of the RAS Extension PE architecture.

MPAM PE Architecture

Read this for a description of the MPAM Extension PE architecture.

The Scalable Matrix Extension

Read this for a description of the Scalable Matrix Extension.

AArch64 System Register Encoding

Read this for a description of the encoding of the AArch64 System registers, and the other uses of the AArch64 System registers encoding space.

AArch64 System Register Descriptions

Read this for an introduction to, and description of, each of the AArch64 System registers.

Part E, The AArch32 Application Level Architecture

Part E describes the AArch32 state application level view of the architecture. It contains the following chapters:

The AArch32 Application Level Programmers' Model

Read this for an application level description of the programmers' model for software executing in AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

The AArch32 Application Level Memory Model

Read this for an application level description of the memory model for software executing in AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32 state. It includes information about Arm memory types, attributes, and memory access controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part F, The AArch32 Instruction Sets

Part F describes the T32 and A32 instruction sets, which are used in AArch32 state. It contains the following chapters:

About the T32 and A32 Instruction Descriptions

Read this to understand the format of the T32 and A32 instruction descriptions.

The AArch32 Instruction Sets Overview

Read this for an overview of the T32 and A32 instruction sets.

T32 Instruction Set Encoding

Read this for a description of the T32 instruction set encoding. This includes the T32 encoding of the Advanced SIMD and floating-point instructions.

A32 Instruction Set Encoding

Read this for a description of the A32 instruction set encoding. This includes the A32 encoding of the Advanced SIMD and floating-point instructions.

T32 and A32 Base Instruction Set Instruction Descriptions

Read this for a description of each of the T32 and A32 base instructions.

T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

Read this for a description of each of the T32 and A32 Advanced SIMD and floating-point instructions.

Part G, The AArch32 System Level Architecture

Part G describes the AArch32 state system level view of the architecture. It contains the following chapters:

The AArch32 System Level Programmers' Model

Read this for a description of the AArch32 state system level view of the programmers' model for execution in an Exception level that is using AArch32.

AArch32 Self-hosted Debug

Read this for an introduction to, and a description of, self-hosted debug in AArch32 state.

AArch32 Self-hosted Trace

Read this for an introduction to, and a description of, self-hosted trace in AArch32 state.

The AArch32 System Level Memory Model

Read this for a system level view of the general features of the memory system.

The AArch32 Virtual Memory System Architecture

Read this for a description of the AArch32 Virtual Memory System Architecture (VMSA).

The Generic Timer in AArch32 state

Read this for a description of the AArch32 view of an implementation of the Arm Generic Timer.

AArch32 System Register Encoding

Read this for a description of the encoding of the AArch32 System registers, including the System instructions that are part of the AArch32 System registers encoding space.

AArch32 System Register Descriptions

Read this for a description of each of the AArch32 System registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part H, External Debug

Part H describes the architecture for external debug. It contains the following chapters:

About External Debug

Read this for an introduction to external debug, and a definition of the scope of this part of the Manual.

Debug State

Read this for a description of Debug state, which the PE might enter as the result of a Halting debug event.

Halting Debug Events

Read this for a description of the external debug events referred to as Halting debug events.

The Debug Communication Channel and Instruction Transfer Register

Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

The Embedded Cross-Trigger Interface

Read this for a description of the Embedded Cross-Trigger Interface.

Debug Reset and Powerdown Support

Read this for a description of reset and powerdown support in the Debug architecture.

The PC Sample-based Profiling Extension

Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension to an Armv8 or Armv9 implementation.

About the External Debug Registers

Read this for some additional information about the external debug registers.

External Debug Register Descriptions

Read this for a description of each external debug register.

Part I, Memory-mapped Components of the Arm architecture

Part I describes the memory-mapped components in the architecture. It contains the following chapters:

Requirements for Memory-mapped Components

Read this for descriptions of some general requirements for memory-mapped components within a system that complies with the Arm architecture.

System Level Implementation of the Generic Timer

Read this for a definition of a system level implementation of the Generic Timer.

Recommended External Interface to the Performance Monitors

Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

Recommended External Interface to the Activity Monitors

Read this for a description of the recommended memory-mapped interface to the Activity Monitors.

External System Control Register Descriptions

Read this for a description of each memory-mapped system control register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part J, Architectural Pseudocode

Part J contains pseudocode that describes various features of the Arm architecture. It contains the following chapter:

A-profile Architecture Pseudocode

Read this for the pseudocode definitions that describe various features of the Arm architecture, for operation in AArch64 state and in AArch32 state.

Part K, Appendixes

Note

Some of the descriptions in the following appendixes are not part of the Arm architecture specification. They are included here as supplementary information, for the convenience of developers and users who might require this information.

This Manual contains the following appendixes:

Architectural Constraints on UNPREDICTABLE Behaviors

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Arm architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

Recommendations for Reporting Memory Attributes on an Interconnect

Read this for the Arm recommendations about how the architectural memory attributes are reported on an interconnect.

ETE Recommended Configurations

Read this for a description of the ETE recommended configurations.

Stages of Execution

Read this for a description of the stages of execution.

Recommended External Debug Interface

Read this for a description of the recommended external debug interface.

Additional Information for Implementations of the Generic Timer

Read this for additional information about implementations of the Arm Generic Timer. This information does not form part of the architectural definition of the Generic Timer.

Legacy Instruction Syntax for AArch32 Instruction Sets

Read this for information about the pre-UAL syntax of the AArch32 instruction sets, which are valid for the A32 instruction set.

Address Translation Examples

Read this for examples of translation table lookups using the translation regimes described in The AArch64 Virtual Memory System Architecture and The AArch32 Virtual Memory System Architecture.

Recommended Upload and Download Processes for External Debug

Read this for information about implementing and using the Arm architecture.

Software Usage Examples

Read this for software examples that help understanding of some aspects of the Arm architecture.

Barrier Litmus Tests

Read this for examples of the use of barrier instructions provided by the Arm architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Random Number Generation

Read this for information on the generation of random numbers using FEAT_RNG.

Arm Pseudocode Definition

Read this for definitions of the Arm pseudocode.

Registers Index

Read this for an alphabetic and functional index of AArch32 and AArch64 registers, and memory-mapped registers.

Glossary

Defines terms used in this Manual that have a specialized meaning.

Note

Terms that are well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Conventions

The following sections describe conventions that this book can use:

- Typographic conventions.
- Rules-based writing.
- Signals.
- Numbers.
- Pseudocode descriptions.
- Assembler syntax descriptions.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

bold Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the Glossary.

Colored text Indicates a link. This can be:

- A URL, for example https://developer.arm.com.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the Manual that defines the colored term, for example Simple sequential execution or SCTLR.

{ and } Braces, { and }, have two distinct uses:

Optional items

In syntax descriptions braces enclose optional items. In the following example they indicate that the `<shift>`parameter is optional:

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Similarly, they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

Sets of items

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE.

Notes Notes are formatted as:

Note

This is a Note.

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note might repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rules-based writing

Some sections of this Manual use rules-based writing. Rules-based writing consists of a set of individual content items. A content item is classified as one of the following:

- Rule.
- Information.
- Software usage.
- Declaration.

Rules are normative statements. An implementation that is compliant with this specification must conform to all Rules in this Manual that apply to that implementation.

Rules must not be read in isolation. Where a particular feature is specified by multiple Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read all chapters and sections of this Manual to ensure that an implementation is compliant.

Content items other than Rules are informative statements. These are provided as an aid to understanding this Manual.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this Manual. After content reaches beta status, a given content item has the same identifier across subsequent versions of this Manual.

Content item rendering

In this Manual, a content item is rendered with a token of the following format in the left margin: L_{iiiii}. L is a label that indicates the content class of the content item. iiiii is the identifier of the content item.

Content item classes

Each of the content item classes has a different function in this Manual.

Rule

A Rule is a statement that describes the behavior of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label R.

Information

An Information statement provides information and guidance as an aid to understanding the Manual.

An Information statement is rendered with the label I.

Software usage

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

A Software usage statement is rendered with the label S.

Declaration

A Declaration statement introduces concepts or terminology.

A Declaration does not describe behavior.

A Declaration is rendered with the label D.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF_0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This Manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in `monospace` font, and is described in Arm Pseudocode Definition.

Assembler syntax descriptions

This Manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in Structure of the A64 assembler language, and Arm Pseudocode Definition.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional reading

This section lists relevant publications from Arm and third parties.

See Arm Developer, https://developer.arm.com, for access to Arm documentation.

Arm publications

- *AMBA® 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1* (ARM IHI 0032).
- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* (ARM IHI 0099).
- *Arm® Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile* (ARM DDI 0600).
- *ARM® Architecture Reference Manual Supplement - Armv8, for the ARMv8-R AArch32 architecture profile* (ARM DDI 0568).
- *Arm® Architecture Specification Language Reference* (ARM DDI 0626).
- *Arm® Architecture Specification Language Readers' Guide* (ARM 111069)
- *Arm® Debug Interface Architecture Specification, ADIv6.0* (ARM IHI 0074).
- *Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *ARM® Procedure Call Standard for the ARM 64-bit Architecture* (ARM IHI 0055).
- *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).
- *Arm® Server Base System Architecture* (ARM DEN 0029).
- *Arm® Base System Architecture* (ARM DEN 0094).

Other publications

The following publications are referred to in this Manual, or provide more information:

- *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001.
- IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
- IEEE Std 754-1985, *IEEE Standard for Floating-point Arithmetic*, March 1985.
- *Secure Hash Standard (SHA)*, Federal Information Processing Standards Publication 180-2, August 2002.
- *The Galois/Counter Mode of Operation*, McGraw, D. and Viega, J., Submission to NIST Modes of Operation Process, January 2004.
- *Memory Consistency Models for Shared Memory-Multiprocessors*, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685.
- *Standard Manufacturer's Identification Code, JEP106*, JEDEC Solid State Technology Association.
- *SM3 Cryptographic Hash Algorithm*, China Internet Network Information Center (CNNIC).
- *SM4 Block Cipher Algorithm*, China Internet Network Information Center (CNNIC).
- *The QARMA Block Cipher Family*, Roberto Avanzi, Qualcomm Product Security Initiative. Available from https://eprint.iacr.org/2016/444.
- *OCP 8-bit Floating Point Specification (OFP8)* Paulius Micikevicius, Stuart Oberman, Pradeep Dubey, Marius Cornea, Andres Rodriguez, Ian Bratt, Richard Grisenthwaite, et al., June 2023. Available from https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Manual

If you have any comments or queries about this Manual, create a ticket at https://support.developer.arm.com.

As part of the ticket, include:

- The title, Arm® Architecture Reference Manual for A-profile architecture.
- The number, ARM DDI 0487 M.b.
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader. Search performance may be significantly improved by increasing the Fast Find Maximum Cache Size under Search in Preferences.
If using Google Chrome™ or Microsoft Edge™, load time is significantly improved since the April 2025 release of these browsers.
Users may find, on some systems, that PDF viewers such as Okular or Foxit™ give good performance.

Inclusive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included language that can be offensive. We have replaced this language. To report offensive language in this document, email `terms@arm.com`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part A
Arm Architecture Introduction and Overview

Chapter A1
Introduction to the Arm Architecture

This chapter introduces the Arm architecture. It contains the following sections:

- About the Arm architecture.
- Architecture profiles.
- Arm architectural concepts.
- Supported data types.
- Floating-point support.
- The Arm memory model.
- Reliability, Availability, and Serviceability.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.1 About the Arm architecture

The Arm architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a *processing element*, often abbreviated to *PE*. Implementations compliant with the Arm architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the Arm architecture must be the same as a simple sequential execution of the program on the processing element. The programmer-visible behavior does not include a specified execution time for any instruction but requires each instruction to execute in finite time, as long as another observer is not continually modifying any Location associated with a Memory Effect, other than an Explicit Memory Effect, that is architecturally required for execution of the instruction. In cases where another observer is continually modifying a Location associated with a Memory Effect, other than an Explicit Memory Effect, at least one of the observers must make forward progress in finite time. Unless otherwise specified, the architecturally defined effects of each instruction also complete in finite time.

The Arm Architecture Reference Manual also describes rules for software to use the PE.

The Arm architecture includes definitions of:

- An associated debug architecture, see:
  - AArch64 Self-hosted Debug.
  - AArch32 Self-hosted Debug.
  - External Debug.
- Associated trace architectures that define trace units that implementers can implement with the associated processor hardware. For more information, see:
  - The *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
  - AArch64 Self-hosted Trace.
  - The Embedded Trace Extension.
  - The Trace Buffer Extension.
  - AArch32 Self-hosted Trace.

Arm also provides system architecture specifications for systems aimed at particular product categories, such as servers, or personal computers, *Arm® Server Base System Architecture* (ARM DEN 0029) and *Arm® Base System Architecture* (ARM DEN 0094). These specifications build on the architecture specification in this Manual, restricting, as defined in those specifications, some of the optional and IMPLEMENTATION DEFINED features defined in this specification, as well as specifying additional features applicable at the system level.

The Arm architecture is a *Reduced Instruction Set Computer* (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.

This document defines versions 8 and 9 of the A-profile architecture.

See Architecture profiles for more information.

The Arm architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the Arm architecture.

An important feature of the Arm architecture is backward compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases.

The Arm architecture supports:

- A 64-bit Execution state, AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A 32-bit Execution state, AArch32, that is compatible with previous versions of the Arm architecture.

Features that are optional are explicitly defined as such in this Manual.

Note

The presence of an ID register field for a feature does not imply that the feature is optional.

Both Execution states support floating-point instructions:

- AArch32 state provides:
  - SIMD instructions in the base instruction sets that operate on the 32-bit general-purpose registers.
  - Advanced SIMD instructions that operate on registers in the *SIMD and floating-point register* (SIMD&FP register) file.
  - Scalar floating-point instructions that operate on registers in the SIMD&FP register file.
- AArch64 state provides:
  - Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  - Scalar floating-point instructions that operate on registers in the SIMD&FP register file.

    Note

    The A64 instruction set does not include SIMD instructions that operate on the general-purpose registers, therefore, some AArch64 instructions descriptions use SIMD as a synonym for Advanced SIMD.

  - SVE instructions that operate on registers in the SVE register files.
  - SME instructions that operate on registers in the SVE register files and ZA storage.

Note

See Conventions for information about conventions used in this Manual, including the use of SMALL CAPITALS for particular terms that have Arm-specific meanings that are defined in the Glossary.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Nine major versions of the architecture have been defined to date, denoted by the version numbers 1 to 9. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

AArch64 Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

Note

The *Base instruction set* comprises the supported instructions other than the floating-point instructions.

See sections Execution state and The instruction sets for more information.

Arm defines three architecture profiles:

A Application profile, described in this Manual:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

  Note

  An Armv8-A implementation can be called an AArchv8-A implementation and an Armv9-A implementation can be called an AArchv9-A implementation.

- Supports the A64, A32, and T32 instruction sets.

R Real-time profile:

- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).
- Supports an optional VMSA based on an MMU.
- Supports the A64, or the A32 and T32 instruction sets.

M Microcontroller profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Supports a PMSA based on an MPU.
- Supports a variant of the T32 instruction set.

This Manual describes only Armv8-A and Armv9-A. For information about the R and M architecture profiles, and earlier Arm architecture versions, see:

- The *Arm® Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile*.
- The *ARM® Architecture Reference Manual Supplement - ARMv8, for the ARMv8-R AArch32 architecture profile*.
- The *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition*.
- The *Armv8-M Architecture Reference Manual*.
- The *Armv7-M Architecture Reference Manual*.
- The *ARMv6-M Architecture Reference Manual*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.3 Arm architectural concepts

This section introduces architectural concepts and the associated terminology.

The following subsections describe key architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:

- Execution state.
- The instruction sets.
- System registers.
- Arm Debug.

A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The Exception model.
  - The *Virtual Memory System Architecture* (VMSA).
  - The programmers' model.

The Execution states are:

AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *Program Counter* (PC), *stack pointers* (SPs), and *Exception Link Registers* (ELRs).
- Provides 32 128-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see The instruction sets.
- Defines the Armv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an *execution privilege* hierarchy, see Exception levels.
- Provides support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see The AArch64 Virtual Memory System Architecture.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

AArch32 The 32-bit Execution state. This Execution state:

- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and *Link Register* (LR). The LR is used as both an ELR and a procedure link register.
  Some of these registers have multiple *banked* instances for use in different PE *modes*.
- Provides a single ELR, for exception returns from Hyp mode.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, T32 and A32. For more information, see The instruction sets.
- Supports the Armv7-A Exception model, based on *PE modes*, and maps this onto the Armv8 Exception model, that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The T32 and A32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the *Application Program Status Register* (APSR) or the *Current Program Status Register* (CPSR).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Later subsections give more information about the different properties of the Execution states.

Transferring control between the AArch64 and AArch32 Execution states is known as *interprocessing*. The PE can move between Execution states only on a change of Exception level, and subject to the rules given in Interprocessing. This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

A1.3.2 The instruction sets

The possible instruction sets depend on the Execution state:

AArch64 AArch64 state supports a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.

For information on the A64 instruction set, see A64 Instruction Set Overview.

If FEAT_SVE is implemented, the A64 instruction set supports scalable vector instructions. See About the SVE instructions.

If FEAT_SME is implemented, the A64 instruction set supports scalable matrix instructions. See About the SME instructions.

AArch32 AArch32 state supports the following instruction sets:

A32 This is a fixed-length instruction set that uses 32-bit instruction encodings.

T32 This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

In previous documentation, these instruction sets were called the ARM and Thumb® instruction sets. Armv8 and Armv9 extend each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.

For information on the T32 and A32 instruction sets, see The AArch32 Instruction Sets Overview.

The instruction sets support SIMD and scalar floating-point instructions. See Floating-point support.

A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:

- <register_name>_ELx, where x is 0, 1, 2, or 3.

For information about Exception levels, see Exception levels.

The System registers comprise:

- The following registers that are described in this Manual:
  - General system control registers.
  - Debug registers.
  - RAS registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.
  - Optionally, the Activity Monitors registers.
  - Optionally, the Scalable Vector Extension registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Optionally, in Armv9, Trace System registers.
- Optionally, one or more of the following groups of registers that are defined in other Arm architecture specifications:
  - Trace System registers, as defined in the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
  - *Generic Interrupt Controller* (GIC) System registers, see The Arm Generic Interrupt Controller System registers.

For information about the AArch64 System registers, see AArch64 System Register Descriptions.

For information about the AArch32 System registers, see AArch32 System Register Descriptions.

A1.3.3.1 The Arm Generic Interrupt Controller System registers

From version 3 of the Arm Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this Manual include these registers, see:

- About the GIC System registers, for more information about the AArch64 GIC System registers.
- About the GIC System registers, for more information about the AArch32 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

Note

The programmers' model for earlier versions of the GIC architecture is wholly memory-mapped.

A1.3.4 Arm Debug

Armv8 and later architectures support the following:

Self-hosted debug

In this model, the PE generates *debug exceptions*. Debug exceptions are part of the Armv8 Exception model.

External debug

In this model, *debug events* cause the PE to enter *Debug state*. In Debug state, the PE is controlled by an external debugger.

All Armv8 and later implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:

- In AArch64 state, see AArch64 Self-hosted Debug.
- In AArch32 state, see AArch32 Self-hosted Debug.

For more information about external debug, see External Debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4 Supported data types

The Arm architecture supports the following integer data types:

Byte 8 bits.

Halfword 16 bits.

Word 32 bits.

Doubleword 64 bits.

Quadword 128 bits.

The architecture also supports the following floating-point data types:

- Half-precision. See Half-precision floating-point formats.
- Single-precision. See Single-precision floating-point format.
- Double-precision. See Double-precision floating-point format.
- BFloat16. See BFloat16 floating-point format.
- 8-bit floating-point. See FP8 floating-point formats.

It also supports vectors, where a register holds a vector of multiple elements, each the same size and data type. See Advanced SIMD vector formats and SVE vector formats.

The architecture provides the following:

- A general-purpose register file.
- A SIMD&FP register file.
- If FEAT_SVE or FEAT_SME is implemented, an SVE scalable vector register file and an SVE scalable predicate register file.
- If FEAT_SME is implemented, the scalable ZA storage.
- If FEAT_SME2 is implemented, the SME2 ZT0 register.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- The general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers, or as 32-bit registers using only the bottom 32 bits.
- The SIMD&FP register file contains 128-bit registers:
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see Instruction Mnemonics.
- The SVE scalable vector register file contains registers of an IMPLEMENTATION DEFINED width:
  - An SVE scalable vector register has an IMPLEMENTATION DEFINED width that is a power of two, from a minimum of 128 bits up to a maximum of 2048 bits.
  - All SVE scalable vector registers in an implementation are the same width.
- The SVE scalable predicate register file contains registers of an IMPLEMENTATION DEFINED width:
  - An SVE predicate register has an IMPLEMENTATION DEFINED width that is a power of two, from a minimum of 16 bits up to a maximum of 256 bits.
- The SME scalable ZA storage is a two-dimensional array of bytes. See ZA storage.
- The SME2 ZT0 register is 512 bits. See SME2 ZT0 register.

For more information on the register files in AArch64 state, see Registers in AArch64 Execution state.

In AArch32 state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The general-purpose register file contains 32-bit registers:
  - Two 32-bit registers can support a doubleword.
  - Vector formatting is supported, see Figure A1-4.
- The SIMD&FP register file contains 64-bit registers:
  - AArch32 state does not support quadword integer or floating-point data types.

Note

Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32 state, see The general-purpose registers, and the PC, in AArch32 state.

A1.4.1 Advanced SIMD vector formats

In an implementation that includes the Advanced SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. In AArch32 state, the combination of a register and a data type describes a vector of elements, where the number of elements in the vector is implied by the size of the data type and the size of the register. In AArch64 state, the explicit combination of a register, number of elements, and element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction.

Vector indices are in the range 0 to (number of elements - 1). An index of 0 refers to the least significant bits of the vector.

For more information on the Advanced SIMD and floating-point registers in AArch32 state, see The SIMD and floating-point register file.

A1.4.1.1 Advanced SIMD vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as V*n*, where *n* is a value from 0 to 31.

The SIMD&FP registers support the following data formats for loads, stores, and data-processing operations:

- A single, scalar, element in the least significant bits of the register. See Registers in AArch64 Execution state.
- A 128-bit vector of byte, halfword, word, or doubleword elements. See Figure A1-1.
- A 64-bit vector of byte, halfword, word, or doubleword elements. See Figure A1-1.

For vectors, the element sizes are defined in Table A1-1, with the vector format described as:

- For a 128-bit vector: Vn.{2D, 4S, 8H, 16B}.
- For a 64-bit vector: Vn.{1D, 2S, 4H, 8B}.

Table A1-1 SIMD elements in AArch64 state

Mnemonic	Size
B	8 bits
H	16 bits
S	32 bits
D	64 bits

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

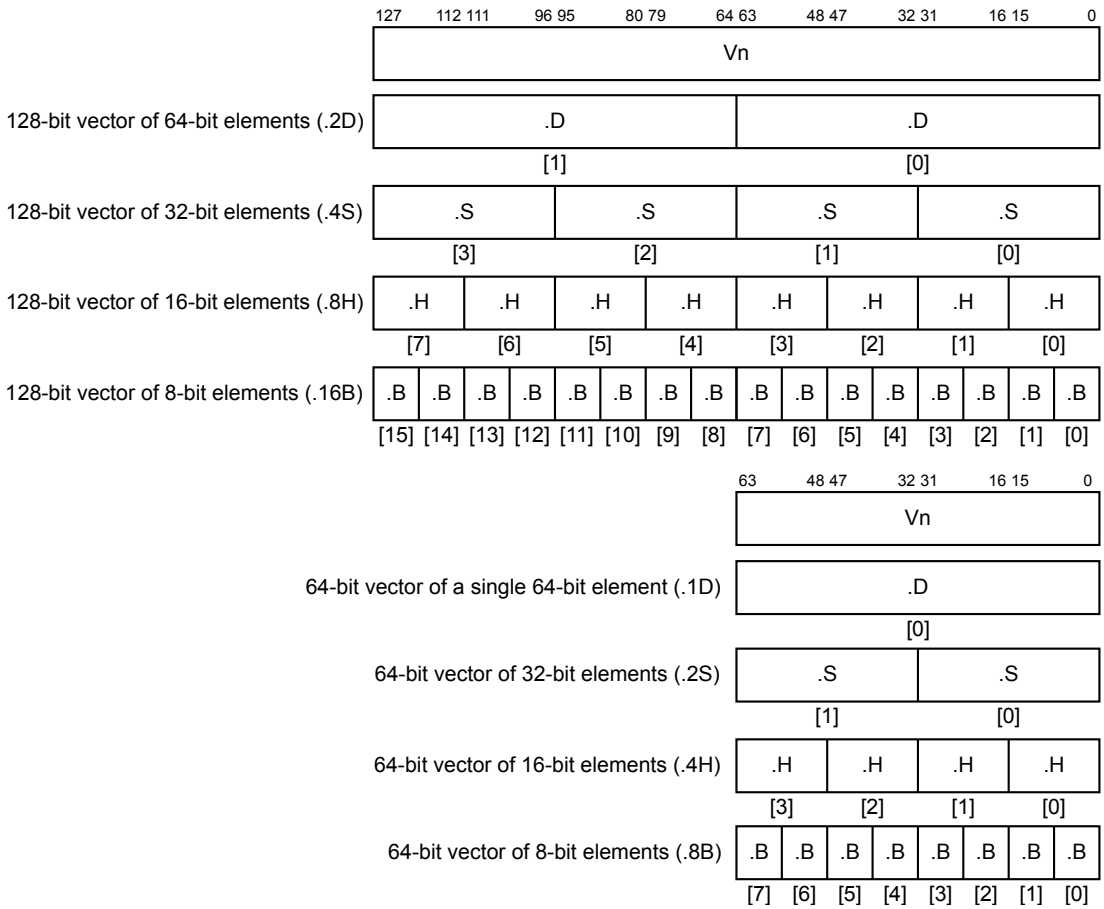

Figure A1-1 SIMD vectors in AArch64 state

A1.4.1.2 Advanced SIMD vector formats in AArch32 state

Table A1-2 shows the available formats. Each instruction description specifies the data types that the instruction supports.

Table A1-2 Advanced SIMD data types in AArch32 state

Data type specifier	Meaning
`.<size>`	Any element of `<size>` bits
`.F<size>`	Floating-point number of `<size>` bits
`.I<size>`	Signed or unsigned integer of `<size>` bits
`.P<size>`	Polynomial over {0, 1} of degree less than `<size>`
`.S<size>`	Signed integer of `<size>` bits
`.U<size>`	Unsigned integer of `<size>` bits

Polynomial arithmetic over {0, 1} describes the polynomial data type.

The `.F16` data type is the half-precision data type selected by the FPSCR.AHP bit, see Half-precision floating-point formats.

The `.F32` data type is the Arm standard single-precision floating-point data type, see Single-precision floating-point format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-2 shows the hierarchy of the Advanced SIMD data types.

<table>
<tr><td rowspan="4">.8</td><td rowspan="2">.I8</td><td>.S8</td></tr>
<tr><td>.U8</td></tr>
<tr><td colspan="2">.P8</td></tr>
<tr><td colspan="2">-</td></tr>
<tr><td rowspan="5">.16</td><td rowspan="2">.I16</td><td>.S16</td></tr>
<tr><td>.U16</td></tr>
<tr><td colspan="2">.P16 †</td></tr>
<tr><td colspan="2">.F16</td></tr>
<tr><td colspan="2">.BF16</td></tr>
<tr><td rowspan="3">.32</td><td rowspan="2">.I32</td><td>.S32</td></tr>
<tr><td>.U32</td></tr>
<tr><td colspan="2">-</td></tr>
<tr><td rowspan="5">.64</td><td colspan="2">.F32</td></tr>
<tr><td rowspan="2">.I64</td><td>.S64</td></tr>
<tr><td>.U64</td></tr>
<tr><td colspan="2">.P64 ‡</td></tr>
<tr><td colspan="2">-</td></tr>
</table>

† Output format only. See `VMULL` instruction description.

‡ Available only if the Cryptographic Extension is implemented. See `VMULL` instruction description.

Figure A1-2 Advanced SIMD data type hierarchy in AArch32 state

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish between signed and unsigned inputs.

Figure A1-3 shows the Advanced SIMD vectors in AArch32 state.

Note

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

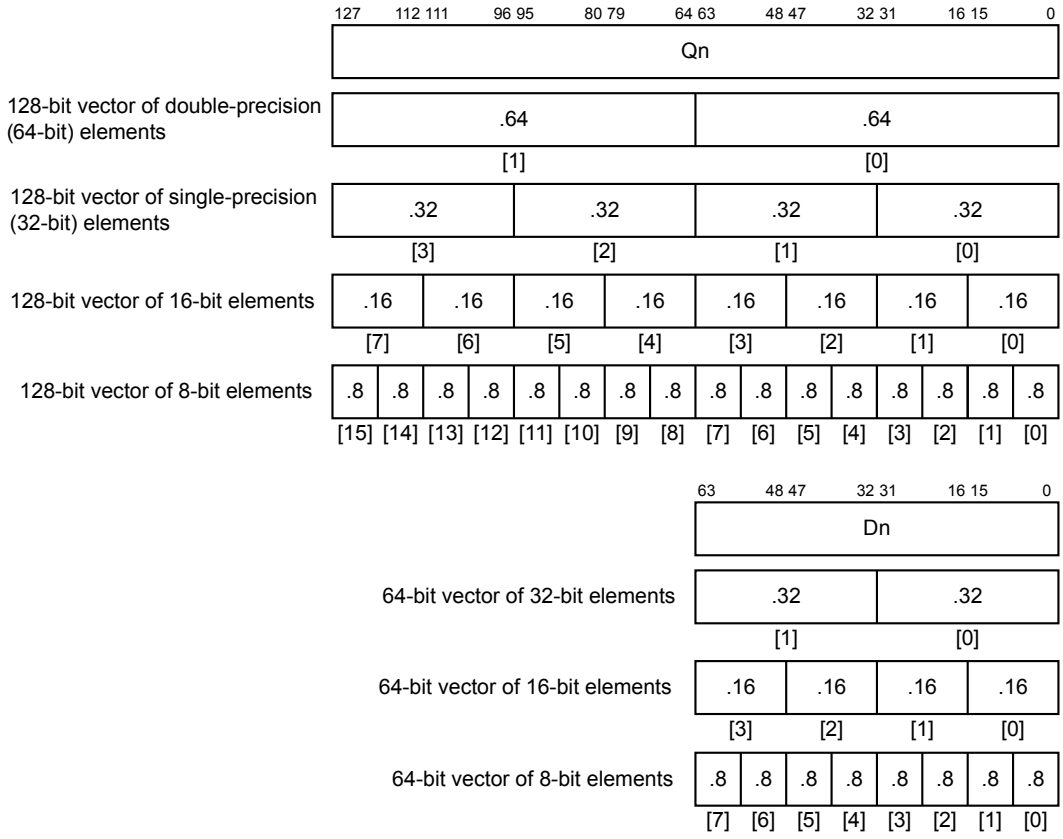

Figure A1-3 Advanced SIMD vectors in AArch32 state

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-4 shows these formats, that means that a general-purpose register can be treated as either 2 halfwords or 4 bytes.

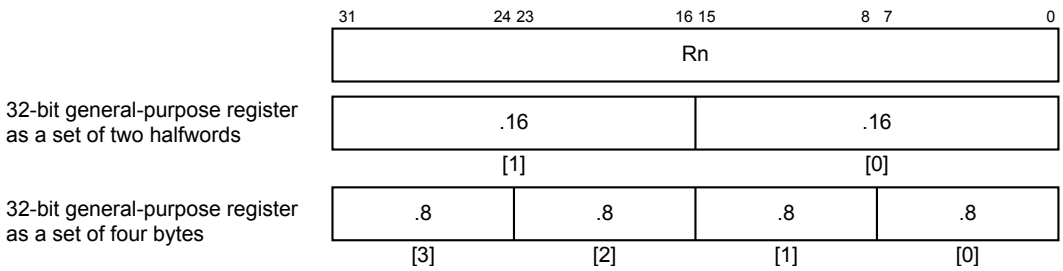

Figure A1-4 Vector formatting in AArch32 state

A1.4.2 SVE vector formats

In an implementation that includes the AArch64 SVE instructions, an SVE register can hold one or more packed or unpacked elements, all of the same size and type. The combination of a register and an element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the *Effective SVE vector length* of the register.

Vector element indexes are in the range 0 to (number of elements - 1). An index of 0 refers to bits 0 to (element size -1) of the vector.

For the definition of Effective SVE vector length, see Configurable SVE vector lengths.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4.2.1 Scalable vector formats in AArch64 state

In AArch64 state, the SVE scalable vector registers can be referred to as Z*n*, where *n* is a value from 0 to 31.

The SVE scalable vector registers support the following data formats for loads, stores, and data-processing operations:

- Advanced SIMD data formats. Bits[127:0] of each Z*n* register hold the correspondingly numbered V*n* SIMD&FP register.
- A configurable-length vector of byte, halfword, word, doubleword, or quadword elements. See Figure A1-5. Also see Z0-Z31 in The AArch64 Application Level Programmers' Model.

The element sizes are defined in Table A1-3 with the vector format described as:

- Zn.{Q, D, S, H, B}.

Table A1-3 SVE elements in AArch64 state

Mnemonic	Element size
`B`	8 bits
`H`	16 bits
`S`	32 bits
`D`	64 bits
`Q`	128 bits

Figure A1-5 shows the vector formats of an SVE scalable vector register containing:

- A configurable-length vector, where the *Effective SVE vector length* is 256 bits.
- A fixed-length 128-bit Advanced SIMD vector.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

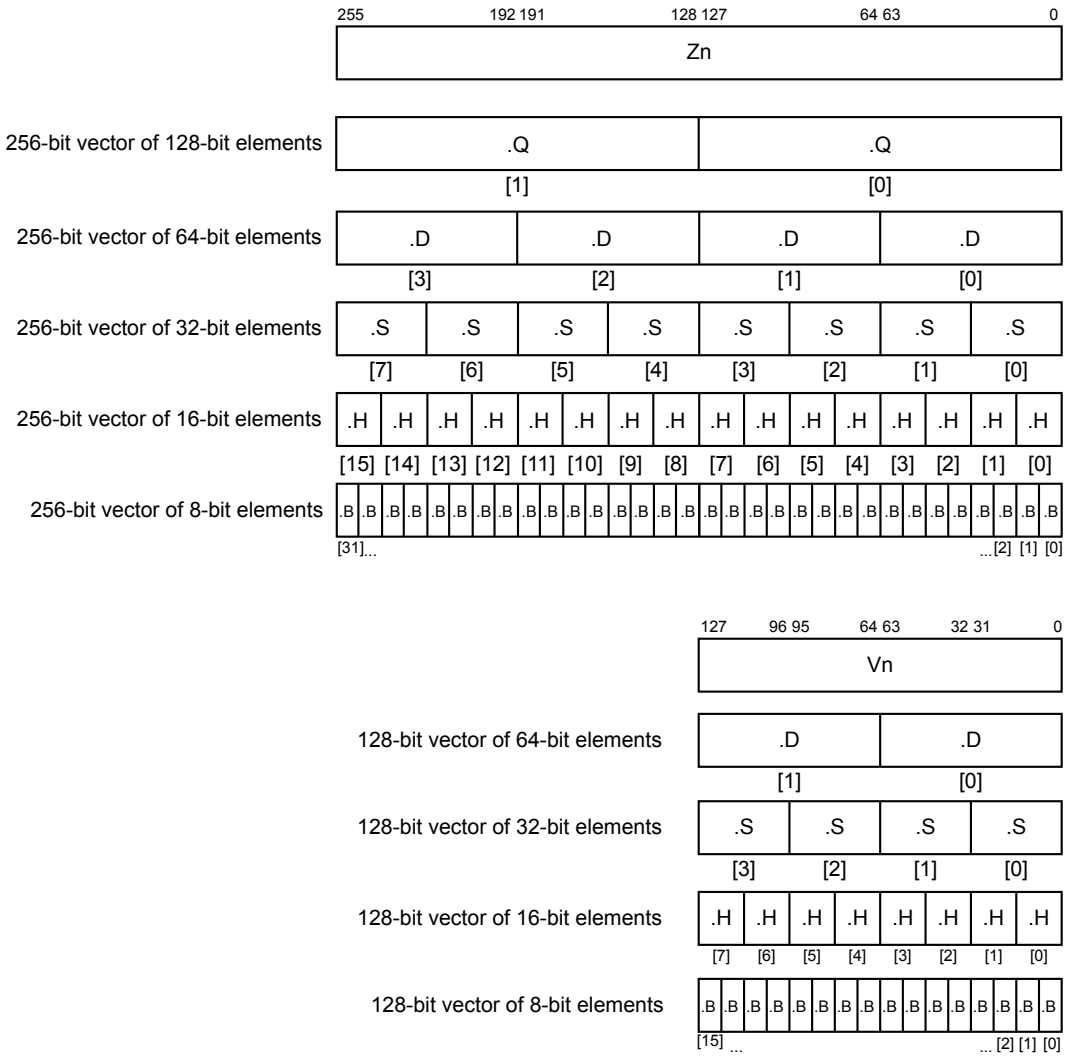

Figure A1-5 SVE vectors, and Advanced SIMD vectors, in an SVE scalable vector register

A1.4.3 Half-precision floating-point formats

The Arm architecture supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.
- Arm *alternative half-precision* format.

Note

BFloat16 is not a half-precision floating-point format, see BFloat16 floating-point format.

Both formats can be used for conversions to and from other floating-point formats. FPCR.AHP controls the format in AArch64 state and FPSCR.AHP controls the format in AArch32 state. FEAT_FP16 adds half-precision data-processing instructions, which always use the IEEE format. These instructions ignore the value of the relevant AHP field, and behave as if it has an Effective value of 0. All SVE half-precision data-processing instructions, including conversions, ignore the value of FPCR.AHP, and behave as if it has an Effective value of 0.

The description of IEEE half-precision includes Arm-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:

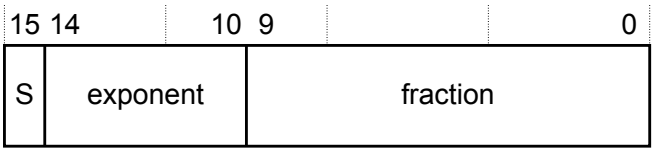

The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}, or approximately 6.104×10^{-5}.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.fraction)$

The minimum positive denormalized number is 2^{-24}, or approximately 5.960×10^{-8}.

Half-precision denormalized numbers are not flushed to zero by default. When FEAT_FP16 is implemented, the FPCR.FZ16 bit controls whether flushing denormalized numbers to zero is enabled for half-precision data-processing instructions. For details, see Flushing denormalized numbers to zero.

exponent == `0x1F`

The value depends on which half-precision format is being used:

IEEE half-precision

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[9]:

bit[9] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

Alternative half-precision

The value is a normalized number and is equal to:

$-1^{S} \times 2^{16} \times (1.\text{fraction})$

The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

A1.4.4 Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes Arm-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:

31	30 … 23	22 … 0
S	exponent	fraction

The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^{S} \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 When S==0.

-0 When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and -0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero.

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN*, see Default NaN.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.5 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both SIMD and floating-point instructions in AArch64 state, and only by floating-point instructions in AArch32 state.

This description includes IMPLEMENTATION SPECIFIC details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

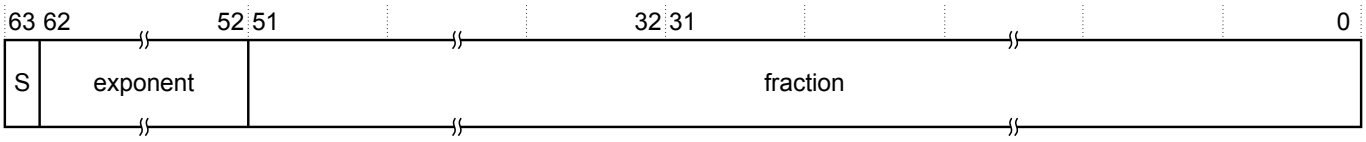

Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < `0x7FF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-1023)} \times (1.fraction)$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.fraction)$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see Flushing denormalized numbers to zero.

exponent == `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. As for single-precision, there are two infinities:

+infinity When S==0.

-infinity When S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

bit[51] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN*, see Default NaN.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.6 BFloat16 floating-point format

BFloat16 is a 16-bit floating-point storage format that inherits many of its properties and behaviors from the IEEE 754 single-precision format described in Single-precision floating-point format.

The BFloat16 format is:

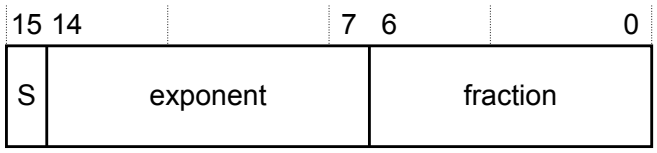

0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-127)} \times (1.fraction)$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-7}) \times 2^{127}$, or approximately 3.390×10^{38}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0.

-0 when S==1.

These usually behave identically. However, they yield different results in some circumstances. For example, the sign of the result produced as the result of multiplying by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer bitwise comparison of the two halfwords.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.fraction)$

The minimum positive denormalized number is 2^{-133}, or approximately 9.184×10^{-41}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

exponent == `0xFF`

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[6]:

bit[6] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[6] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

BFloat16 values are 16-bit halfwords that software can convert to single-precision format, by appending 16 zero bits, so that single-precision arithmetic instructions can be used. A single-precision value can be converted to a BFloat16 value, either by:

- Truncating, by removing the least significant 16 bits.
- Using the BFloat16 conversion instructions, see Table C3-113.

See also:

- Summary of BFloat16 instruction behaviors.

A1.4.7 FP8 floating-point formats

The Arm architecture supports two 8-bit floating-point (FP8) formats:

- OFP8 E4M3 format.
- OFP8 E5M2 format.

These formats are used by FP8 instructions, which are instructions, where the input or output elements are in FP8 floating-point format. FPMR.F8S1 controls the format for inputs named *First FP8 input data stream* and FPMR.F8S2 controls the format for inputs named *Second FP8 input data stream*. The format of an FP8 output from an FP8 instruction is determined by FPMR.F8D field.

For more information, see *OCP 8-bit Floating Point Specification (OFP8)*.

A1.4.7.1 OFP8 E4M3 format

The OFP8 E4M3 format is:

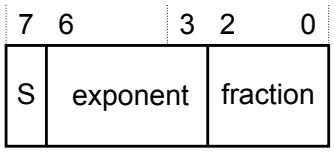

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

(0 < exponent < `0xF`**) || (exponent ==** `0xF` **&& fraction <** `0x7`**)**

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-7)} \times (1.fraction)$

The minimum positive normalized number is 2^{-6}.

The maximum positive normalized number is $(2 - 2^{-2}) \times 2^8$.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-6} \times (0.fraction)$

The minimum positive denormalized number is 2^{-9}.

FP8 denormalized numbers are never flushed to zero.

exponent == `0xF` **&& fraction ==** `0x7`

The value is a *signaling NaN*.

An infinity is not supported.

A1.4.7.2 OFP8 E5M2 format

The OFP8 E5M2 format is:

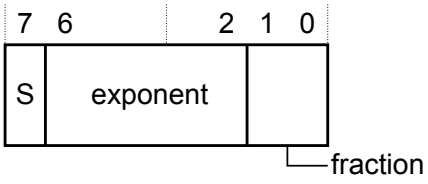

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}.

The maximum positive normalized number is $(2 - 2^{-2}) \times 2^{15}$.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-16}.

FP8 denormalized numbers are never flushed to zero.

exponent == `0x1F`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[1]:

bit[1] == 0 The NaN is a signaling NaN. The sign bit can take any value, and bit[0] is 1.

bit[1] == 1 The NaN is a quiet NaN. The sign bit and bit[0] can take any value.

A1.4.8 Conversion between floating-point and fixed-point values

The Arm architecture supports the conversion of a scalar floating-point value to or from a signed or unsigned fixed-point value in a general-purpose register. Conversion instructions take an argument, #fbits, that specifies the number of fraction bits in the fixed-point number. That is, #fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register:

- For a 64-bit register X_d:
  - The integer part is X_d[63:#fbits].
  - The fractional part is X_d[(#fbits-1):0].
- For a 32-bit register W_d or R_d:
  - The integer part is W_d[31:#fbits] or R_d[31:#fbits].
  - The fractional part is W_d[(#fbits-1):0] or R_d[(#fbits-1):0].

These instructions can cause the following floating-point exceptions:

Invalid Operation When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.

Inexact When the numeric result differs from the input value.

Input Denormal When flushing denormalized numbers to zero is enabled and the denormal input is replaced by a zero, see Flushing denormalized numbers to zero and Input Denormal exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

An out of range fixed-point result is saturated to the destination size.

For more information, see Floating-point exceptions and exception traps.

A1.4.9 Polynomial arithmetic over {0, 1}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over {0, 1}, see Supported data types. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit[k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- $0 + 0 = 1 + 1 = 0$.
- $0 + 1 = 1 + 0 = 1$.
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$.
- $1 \times 1 = 1$.

That is:

- Adding two polynomials over {0, 1} is the same as a bitwise exclusive-OR.
- Multiplying two polynomials over {0, 1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

A64, A32, and T32 provide instructions for performing polynomial multiplication of 8-bit values.

- For AArch32, see VMUL (integer and polynomial) and VMULL (integer and polynomial).
- For AArch64, see PMUL and PMULL, PMULL2.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See PMULL, PMULL2.

A1.4.9.1 Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function defined in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5 Floating-point support

The architecture includes the following types of floating-point instructions:

- Scalar floating-point instructions that operate on the lowest numbered element of the SIMD&FP registers.
- Advanced SIMD floating-point instructions that operate on multiple elements of the SIMD&FP registers.
- If FEAT_SVE is implemented, AArch64 SVE instructions that operate on multiple elements of the SVE scalable vector registers, in which the SIMD&FP registers occupy the least significant 128 bits.
- If FEAT_SME is implemented, AArch64 SME instructions that operate on multiple elements of the SVE scalable vector registers and the ZA storage.

The architecture can provide the following levels of floating-point support:

- No floating-point support. This option is licensed only for implementations targeting specialized markets.
- Advanced SIMD and floating-point support.
- SVE, plus Advanced SIMD and floating-point support.
- SME and SVE, plus Advanced SIMD and floating-point support.

Note

All Armv8-A systems that support standard operating systems with rich application environments provide hardware support for Advanced SIMD and floating-point instructions. All Armv9-A systems that support standard operating systems with rich application environments also provide hardware support for SVE2 instructions. It is a requirement of the ARM Procedure Call Standard for AArch64, see *Procedure Call Standard for the Arm 64-bit Architecture*.

The architecture supports the following floating-point formats:

Table A1-4 Floating-point formats

	AArch32 state	AArch64 state
IEEE 754 half-precision	Conversion instructions to and from IEEE 754 half-precision.	
	Supported for data processing if FEAT_FP16 is implemented.	
Arm *alternative half-precision*	Conversion instructions to and from non-IEEE 754 Arm alternative half-precision.	
IEEE 754 single-precision	Supported for scalar data processing.	Supported for scalar and vector data processing.
	Supported for Advanced SIMD, with restrictions. See Arm standard floating-point input and output values.	
IEEE 754 double-precision	Supported for scalar data processing.	Supported for scalar and vector data processing.
	Not supported for Advanced SIMD.	
BFloat16	Supported if FEAT_AA32BF16 is implemented.	Supported if FEAT_BF16 is implemented.
OFP8 E4M3	Not supported.	Supported if FEAT_FP8 is implemented.
OFP8 E5M2	Not supported.	Supported if FEAT_FP8 is implemented.

For many floating-point instructions, there are configurable behaviors, such as:

- Configurable rounding modes. See Rounding.
- Configurable trapping of floating-point exceptions. See Floating-point exceptions and exception traps.
- Configurable non-IEEE 754 *Default NaN* behavior. See NaN handling and the Default NaN.
- Configurable non-IEEE 754 flushing to zero of denormalized numbers. See Flushing denormalized numbers to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Floating-point calculation using AArch32 Advanced SIMD instructions remains unchanged from Armv7. T32 and A32 Advanced SIMD floating-point instructions always use Arm standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see Flushing denormalized numbers to zero.
- Only default NaNs are supported, see Default NaN.
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

If floating-point exception trapping is supported, then unless stated otherwise, when a floating-point exception is not trapped, that exception causes a cumulative status register bit to be set to 1 and the operation produces a default result. For more information, see Floating-point exceptions and exception traps.

In:

- AArch64 state, the Floating-point Control Register, FPCR, controls floating-point operation, and the Floating-point Status Register, FPSR, returns floating-point status information.
- AArch32 state, there is a single Floating-Point Status and Control Register, FPSCR, combining the FPCR and FPSR fields.

In AArch64 state, the PSTATE.{N, Z, C, V} condition flags are updated by floating-point comparison operations.

In AArch32 state, the FPSCR.{N, Z, C, V} condition flags are updated by floating-point comparison operations.

If FEAT_FlagM2 is implemented, the AArch64 instructions `AXFLAG` and `XAFLAG` convert between the Arm floating-point condition flags in PSTATE and an alternative format shown in Table C6-1.

Note

For additional details of AArch32 state floating-point support, see Advanced SIMD and floating-point support.

The remainder of this section contains:

- Instruction support.
- Floating-point standards, and terminology.
- Arm standard floating-point input and output values.
- Summary of BFloat16 instruction behaviors.
- Summary of FP8 instruction behaviors.
- Flushing denormalized numbers to zero.
- NaN handling and the Default NaN
- Rounding
- Floating-point exceptions and exception traps
- Alternate BFloat16 behaviors
- BFloat16 behaviors for instructions that calculate sum-of-products

A1.5.1 Instruction support

The floating-point instructions support:

- Load and store for single elements and vectors of multiple elements.

  Note

  Single elements are also referred to as scalar elements.

- Data processing on single and multiple elements.
- Complex number arithmetic.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Floating-point conversion between different levels of precision.
- Conversion between floating-point and integer types.
- Floating-point rounding.
- Atomic floating-point in-memory operations.

For more information on floating-point instructions in AArch64 state, see A64 Instruction Set Overview.

For more information on floating-point instructions in AArch32 state, see The AArch32 Instruction Sets Overview.

A1.5.2 Floating-point standards, and terminology

The Arm architecture includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this Manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- Arm floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-5 shows how the terminology in this Manual differs from that used in IEEE 754-2008.

Table A1-5 Floating-point terminology

This manual	IEEE 754-2008
Normalized[a]	Normal
Denormal, or denormalized	Subnormal
Round towards Minus Infinity (RM)	roundTowardsNegative
Round towards Plus Infinity (RP)	roundTowardsPositive
Round towards Zero (RZ)	roundTowardZero
Round to Nearest (RN)	roundTiesToEven
Round to Nearest with Ties to Away	roundTiesToAway
Rounding mode	Rounding-direction attribute

a. *Normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this Manual.

A1.5.3 Arm standard floating-point input and output values

The Arm architecture provides full IEEE 754 floating-point arithmetic support. In AArch32 state, floating-point operations performed using Advanced SIMD instructions are limited to *Arm standard floating-point operation*, regardless of the selected rounding mode in the FPSCR.

Arm standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see Flushing denormalized numbers to zero.
- NaNs.
- Infinities.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

Arm standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results with an absolute value less than the minimum normalized number are flushed to zero, see Flushing denormalized numbers to zero.
- NaNs produced in floating-point operations are always the default NaN, see Default NaN.
- Infinities.

Note

AArch64 Advanced SIMD floating-point arithmetic is performed using the rounding mode selected by the FPCR.

A1.5.4 Summary of BFloat16 instruction behaviors

BFloat16 instructions follow the floating-point behaviors shown in Table A1-6 and Table A1-7:

Table A1-6 Advanced SIMD and floating-point, and SVE, BFloat16 instruction behaviors

		Behaviors when the PE is in Streaming SVE mode	Behaviors when the PE is not in Streaming SVE mode
Advanced SIMD & FP	`BFCVT`	Behaves as: Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.
		This instruction also supports the Alternate BFloat16 behaviors.	
	`BFCVTN`, `BFCVTN2` `BFMLALB`, `BFMLALT`	If *legal* when executed in Streaming SVE mode[a], behave as Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.
		These instructions also support the Alternate BFloat16 behaviors.	
	`BFDOT`, `BFMMLA`	If legal when executed in Streaming SVE mode[a], behave as BFloat16 behaviors for instructions that calculate sum-of-products.	Behave as BFloat16 behaviors for instructions that calculate sum-of-products.

SVE	`BFADD`, `BFSUB`
	`BFMLA`, `BFMLS`,
	`BFMUL`
		Behave as Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.

	`BFCLAMP`
	`BFMAX`, `BFMAXNM`
	`BFMIN`, `BFMINNM`
		Behave as both:
		• Floating-point behaviors in Streaming SVE mode.
		• Additional behaviors described in the individual instruction descriptions.
			Follows both:
			• Single-precision behaviors.
			• Additional behaviors described in the individual instruction descriptions.

	`BFCVT`, `BFCVTNT`
	`BFMLALB`, `BFMLALT`
	`BFMLSLB`, `BFMLSLT`
		Behave as Floating-point behaviors in Streaming SVE mode.	Follows single-precision behaviors.

		These instructions also support the Alternate BFloat16 behaviors.	
	`BFDOT`	Behave as BFloat16 behaviors for instructions that calculate sum-of-products.	
	`BFMMLA`	If legal when executed in Streaming SVE mode[a], behaves as BFloat16 behaviors for instructions that calculate sum-of-products.	Behaves as BFloat16 behaviors for instructions that calculate sum-of-products.

a. These instructions generate an *SME illegal instruction exception* in *Streaming SVE mode* if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level. See Streaming SVE mode and R_{ZFGJP}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table A1-7 SME BFloat16 instruction behaviors

SME	Behaviors in Streaming SVE mode		Behaviors when the PE is not in Streaming SVE mode
	ZA storage enabled	ZA storage disabled	
BFADD, BFMLA, BFMLS, BFSUB BFMOPA (non-widening) BFMOPS (non-widening)	Behave as Floating-point behaviors for instructions that target the SME ZA array.	Illegal[a]	Illegal[a]
BFCLAMP BFMAX, BFMAXNM BFMIN, BFMINNM	Behave as both: • Floating-point behaviors in Streaming SVE mode. • Additional behaviors described in the instruction descriptions.		Illegal[a]
BFCVT (single-precision to BFloat16), BFCVTN	Behave as Floating-point behaviors in Streaming SVE mode. These instructions also support the Alternate BFloat16 behaviors.		Illegal[a]
BFDOT, BFVDOT BFMOPA (widening) BFMOPS (widening)	Behave as BFloat16 behaviors for instructions that calculate sum-of-products.	Illegal[a]	Illegal[a]
BFMLAL BFMLSL	Behave as Floating-point behaviors for instructions that target the SME ZA array.	Illegal[a]	Illegal[a]

a. The instructions generate an *SME illegal instruction exception*. See Streaming SVE mode and R_{ZFGJP}.

Unless otherwise specified, BFloat16 instructions follow the behaviors defined in all of:

- Flushing denormalized numbers to zero.
- Rounding.
- Floating-point exceptions and exception traps

See also:

- Streaming SVE mode.
- ZA storage.

A1.5.5 Summary of FP8 instruction behaviors

All FP8 instructions:

- Use the Round to Nearest (RN) mode, ignoring FPCR.RMode.
- Do not flush denormalized inputs and outputs to zero, as if FPCR.{FZ, FIZ, FZ16} are all 0.
- Generate Default NaN outputs, as if FPCR.DN is 1, unless otherwise specified.
- Honor FPCR.AH, if FEAT_AFP is implemented.
- Treat an unsupported FP8 input format, as defined in FPMR.F8S1 and FPMR.F8S2, as a CONSTRAINED UNPREDICTABLE choice of:
  - A signaling NaN.
  - Any of the supported FP8 formats.
- Handle an unsupported FP8 output format, as defined in FPMR.F8D, by a CONSTRAINED UNPREDICTABLE choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - Setting the result to `0xff` and causing an Invalid Operation floating-point exception.
    - Generating the expected result of any of the supported FP8 formats.
- Do not cause an Input Denormal floating-point exception or update FPSR.IDC.

FP8 convert instructions:

- Produce the expected default result and update FPSR.{IOC, IXC, UFC} when they detect Invalid Operation, Inexact, or Underflow exceptional floating-point conditions.
- When the rounded result with unbounded exponent has an absolute value that is greater than the maximum normalized number for the output precision, or when an Infinity is converted from other floating-point formats to an FP8 format:
    - Generate one of the following outputs with the sign of the result:
        - When FPMR.OSC is 1, the maximum normalized number.
        - When FPMR.OSC is 0 and the output is in the OFP8 E4M3 format, a *signaling NaN*.
        - Otherwise, an Infinity.
    - Regardless of the value of FPMR.OSC, cause an Overflow and Inexact floating-point exception.
- When the PE is in Streaming SVE mode, disable trapped floating-point exceptions as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all zero.

FP8 multiplication instructions:

- Produce the expected default result and do not update FPSR.{IOC, IXC, UFC} when they detect Invalid Operation, Inexact, or Underflow exceptional floating-point conditions.
- When the rounded result with unbounded exponent has an absolute value that is greater than the maximum normalized number for the output precision:
    - Generate one of the following outputs with the sign of the result:
        - When FPMR.OSM is 1, the maximum normalized number.
        - When FPMR.OSM is 0, an Infinity.
    - Do not update FPSR.OFC.
- Disable trapped floating-point exceptions, as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all zero.

A1.5.6 Flushing denormalized numbers to zero

For this section if FEAT_AFP is not implemented, the behavior is the same as if FPCR.AH == 0, FPCR.FIZ == 0, and FPCR.NEP == 0.

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization.

If a number `value` satisfies the condition `0 < Abs(value) < MinNorm`, it is treated as a denormalized number.

`MinNorm` is defined as follows:

- For OFP8 E4M3 numbers, `MinNorm` is 2^{-6}.
- For half-precision and OFP8 E5M2 numbers, `MinNorm` is 2^{-14}.
- For single-precision and BFloat16 numbers, `MinNorm` is 2^{-126}.
- For double-precision numbers, `MinNorm` is 2^{-1022}.

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

A1.5.6.1 Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has a denormalized input that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For floating-point instructions, if the instruction processes half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If FPCR.FZ16 == 0, denormalized half-precision inputs are not flushed to zero.
- If FPCR.FZ16 == 1:
  - Denormalized half-precision inputs to data processing instructions are flushed to zero.
  - Denormalized half-precision inputs to convert instructions are not flushed to zero.

If FPCR.FIZ == 1, or both FPCR.AH == 0 and FPCR.FZ == 1, then for floating-point instructions other than `FABS` and `FNEG`, for all inputs other than half-precision, denormalized inputs are flushed to zero.

If FPCR.FZ == 0, for all inputs other than half-precision, FPCR.FZ does not cause denormalized inputs to be flushed to zero, although other factors might cause denormalized inputs to be flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FIZ, all of the following instructions flush all input denormal numbers to zero:

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

A1.5.6.2 Flushing denormalized outputs to zero

If a denormalized output is flushed to zero, the output is returned as zero with the same sign bit as the denormalized output value.

If FPCR.AH == 0, the test for a denormalized number for the purpose of flushing the output to zero occurs before rounding.

If FPCR.AH == 1, the test for a denormalized number for the purpose of flushing the output to zero occurs after rounding using an unbounded exponent.

If FPCR.FZ == 0, then unless otherwise specified, for all outputs other than half-precision, FPCR.FZ does not cause denormalized outputs to be flushed to zero, although other factors might cause denormalized outputs to be flushed to zero.

If FPCR.FZ == 1, for all outputs other than half-precision, for floating-point instructions other than `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, denormalized outputs are flushed to zero.

If FPCR.EBF == 0, `BFDOT`, `BFMMLA`, BFVDOT, BFMOPA (widening), and BFMOPS (widening) instructions unconditionally flush denormalized outputs to zero, as defined in BFloat16 behaviors for instructions that calculate sum-of-products.

If FPCR.FZ16 == 0, denormalized half-precision outputs are not flushed to zero.

If FPCR.FZ16 == 1, for floating-point instructions other than `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, denormalized half-precision outputs are flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.{FZ, FZ16}, denormalized outputs of `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, are not flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FZ, all of the following instructions flush all output denormal numbers to zero:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

If FPCR.FIZ == 1, for the atomic floating-point memory instructions defined in Atomic floating-point memory operations, denormalized BFloat16, single-precision, and double-precision outputs are flushed to zero.

A1.5.6.3 Flushing denormalized BFloat16 intermediate results to zero

BFloat16 arithmetic instructions `BFDOT`, `BFMMLA`, `BFVDOT`, `BFMOPA (widening)`, and `BFMOPS (widening)` in AArch64 state, and VDOT (by element), VDOT (vector), VMMLA in AArch32 state when working with BFloat16 inputs, convert BFloat16 input values to IEEE single-precision format, and calculate N-way dot-products, accumulating the products in single-precision accumulators. If one of these instructions processes an intermediate result that is a single-precision denormalized number:

- If the instruction is one of AArch64 `BFDOT`, `BFMMLA`, `BFVDOT`, `BFMOPA (widening)`, and `BFMOPS (widening)`, the intermediate result is unconditionally flushed to zero if FEAT_EBF16 is not implemented or when FPCR.EBF is 0. See BFloat16 behaviors for instructions that calculate sum-of-products.
- If the instruction is one of AArch32 VDOT (by element), VDOT (vector), VMMLA, the intermediate result is unconditionally flushed to zero.

A1.5.7 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the mantissa, and specifies that the sign bit of a NaN has no significance.

The Arm architecture additionally defines the following in Table A1-8:

- A *Default NaN*, compliant with the IEEE 754 standard.
- A BFloat16 NaN, that follows the encoding in BFloat16 floating-point format.
- OFP8 NaN, that follow the encodings in FP8 floating-point formats.

For a quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a speculative result of a floating-point calculation not involving NaN inputs.

A1.5.7.1 Default NaN

A Default NaN is encoded as described in Table A1-8.

Table A1-8 Default NaN encodings

	Half-precision, IEEE Format	Single-precision	Double-precision	BFloat16	OFP8 E4M3	OFP8 E5M2
Sign bit if FPCR.AH == 0	0	0	0	0	0	0
Sign bit if FPCR.AH == 1	1	1	1	1	1	1
Exponent	`0x1F`	`0xFF`	`0x7FF`	`0xFF`	`0xF`	`0x1F`
Fraction	Bit[9] == ‘1’, bits[8:0] == 0	Bit[22] == ‘1’, bits[21:0] == 0	Bit[51] == ‘1’, bits[50:0] == 0	Bit[6] == ‘1’, bits[5:0] == 0	Bits[2:0] == ‘111’	Bit[1] == ‘1’, bit[0] == ‘0’

When FPCR.DN == 1, if any input to a floating-point operation is a NaN, the output is a Default NaN, unless the instruction is one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMAX`, and `BFMIN`. For these, see the individual instruction descriptions for their Default NaN behavior.

The atomic floating-point memory instructions defined in Atomic floating-point memory operations generate only the Default NaN value, as if FPCR.DN == 1.

Floating-point instructions that detect an Invalid Operation condition for a reason other than one of its inputs being a NaN, produce a Default NaN as the result.

A1.5.7.2 NaN handling

The IEEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived quiet NaN output might have both a different sign and different fraction bits from its source. See individual instruction descriptions for details.

A1.5.7.3 NaN propagation

For NaN propagation rules for:

- BFDOT and BFMMLA, see BFloat16 behaviors for instructions that calculate sum-of-products.
- FP8 instructions, see Summary of FP8 instruction behaviors.

Otherwise, behavior is as described in this section.

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the instruction and the number of operands.

If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is derived from the input NaN.
- If the input NaN is a signaling NaN, the output NaN is derived as follows:
  - If the handling of a signaling NaN by the instruction detects an Invalid Operation condition, the output NaN is derived from a quietened version of the input NaN.
  - If the handling of a signaling NaN by the instruction does not detect an Invalid Operation condition, the output NaN is derived from the input NaN.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, the term “first operand” and “second operand” relate to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If FPCR.DN == 0, for instructions that perform a floating-point operation, other than `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, NaN outputs that derive from NaN inputs are derived as follows:

- If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - FPCR.AH == 0.
  - At least one operand is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, an instruction outputs a quiet NaN derived from the first NaN operand:
  - FPCR.AH == 0.
  - At least one operand is a NaN, but none of the operands is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, the output is a quiet NaN derived from the NaN operand:
  - FPCR.AH == 1.
  - The operation has two floating-point inputs.
  - The operation has only one NaN operand.
- If all of the following apply, the output is a NaN derived from the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>`, or `<Dn>` register, except for SVE FSUBR (vectors) and SVE FDIVR, for which the output is a NaN derived from the `<Zm>` register:
  - FPCR.AH == 1.
  - The operation has two floating-point inputs.
  - The operation has two NaN operands.
- If all of the following apply, the output is a NaN derived from the NaN held in the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>`, or `<Dn>` registers:
  - FPCR.AH == 1
  - The instruction is one of those shown in Table A1-9.
  - One of the following applies:
    - The operation has three NaN operands.
    - The operation has two NaN operands and the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>` or `<Dn>` register holds a NaN.
- If all of the following apply, the output is a NaN derived from the NaN held in the `<Zm>`, `<Vm>`, `<Hm>`, `<Sm>`, or `<Dm>` registers:
  - FPCR.AH == 1
  - The instruction is one of those shown in Table A1-9.
  - The operation has two NaN operands and the `<Zn>`, `<Vn>`, `<Hn>`, `<Sn>` or `<Dn>` register does not hold a NaN.

Table A1-9

Advanced SIMD&FP	`BFMLALB`, `BFMLALT`, `FCMLA`, `FMADD`, `FMLA`, `FMLAL`, `FMLAL2`, `FMLS`, `FMLSL`, `FMLSL2`, `FMSUB`, `FNMADD`, and `FNMSUB`.
SVE	`BFMLA`, `BFMLALB`, `BFMLALT`, `BFMLS`, `BFMLSLB`, `BFMLSLT`, `FCMLA`, `FMAD`, `FMLA`, `FMLALB`, `FMLALT`, `FMLS`, `FMLSLB`, `FMLSLT`, `FMSB`, `FNMAD`, `FNMLA`, `FNMLS`, `FNMSB`.

- When the operation has four or more floating-point inputs:
  - If at least one argument to `FPDot()` is a signaling NaN, a function outputs a quiet NaN derived from the first signaling NaN argument.
  - If at least one argument to `FPDot()` is a NaN, but none of the arguments is a signaling NaN, a function outputs a quiet NaN derived from the first NaN argument.

If FPCR.AH == 0, and an output NaN is derived from an input NaN, the pseudocode functions `FPAbs()`, `FPNeg()`, `BFNeg()`, `FPTrigMAdd()`, and `FPTrigSSel()` can change the sign of the NaN.

If FPCR.AH == 1, and an output NaN is derived from an input NaN, for all cases, the sign bit of the NaN is unchanged.

For `FMAX*`, `FMIN*`, `BFMAX*`, `BFMIN*`, `FCLAMP`, and `BFCLAMP`, the NaN handling is described in the instruction description.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.8 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPCR.RMode or by the instruction.

If FPCR.AH == 1, for any value of FPCR.RMode, the following instructions use Round to Nearest on outputs:

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision and double-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.
- Half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

FPCR.RMode can select one of:

- Round to Nearest (RN) mode.
- Round towards Plus Infinity (RP) mode.
- Round towards Minus Infinity (RM) mode.
- Round towards Zero (RZ) mode.

The following additional rounding behaviors are not selected by FPCR.RMode, but are used by some instructions:

- Round to Odd.
- Round to Nearest with Ties to Away.

A1.5.8.1 Round to Nearest (RN) mode

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - — If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - — If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

A1.5.8.2 Round towards Plus Infinity (RP) mode

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

A1.5.8.3 Round towards Minus Infinity (RM) mode

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

A1.5.8.4 Round towards Zero (RZ) mode

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.8.5 Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is calculated as follows:

- If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with the largest absolute value.
- If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round to Nearest with Ties to Away rounding is used by FCVTAS (scalar), FCVTAS (vector), FCVTAU (scalar), FCVTAU (vector), FRINTA (scalar), and FRINTA (vector).

A1.5.8.6 Round to Odd

Round to Odd is not defined by IEEE 754.

Round to Odd rounds the exact result of a floating-point operation to a value that is representable in the destination format. If the result of the rounding is inexact, the least significant bit of the mantissa is forced to 1.

The following instructions use Round to Odd:

- BFloat16 instructions defined in BFloat16 behaviors for instructions that calculate sum-of-products.
- FCVTXN, FCVTXN2, FCVTX, and FCVTXNT, for which Round to Odd rounding can avoid double rounding errors when a floating-point value is converted to a lower precision destination format through an intermediate precision format.

Example A1-1 Converting 64-bit floating-point format to 16-bit floating-point format

A 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value using the following steps:

1. Use an `FCVTXN` instruction to produce a 32-bit value.
2. Use another instruction with the required rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

A1.5.9 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD, SVE, or SME instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*.

Predicated SVE floating-point instructions only generate floating-point exceptions in response to floating-point operations performed on *Active elements*.

Note

In AArch64 state, an Advanced SIMD, SVE, or SME instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an *operation*, rather than on an *instruction*.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

Synchronous exception generation by floating-point exceptions is enabled using the FPCR as follows:

- For each floating-point exception that supports synchronous exception generation, the relevant control bit chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} is used to enable synchronous exception generation.
- For each floating-point exception that does not support synchronous exception generation, the relevant bit chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} is RAZ/WI.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

A1.5.9.1 Operations that do not generate floating-point exceptions

The following instructions do not generate any floating-point exceptions:

- The SVE exponential accelerator instruction, FEXPA.
- The SVE trigonometric select coefficient instruction, FTSSEL.
- The BFloat16 instructions defined in BFloat16 behaviors for instructions that calculate sum-of-products.
- The SME instructions defined in Floating-point behaviors for instructions that target the SME ZA array.
- The instructions defined in Atomic floating-point memory operations.

If FPCR.AH is 1, all of the following instructions do not generate any floating-point exceptions regardless of their input values:

- The BFloat16 instructions defined in Alternate BFloat16 behaviors.
- Single-precision, double-precision and half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

All the following apply to `FPAbs()`, `FPNeg()`, and `BFNeg()`:

- They cannot generate floating-point exceptions.
- The floating-point behavior described in the Flushing denormalized numbers to zero does not apply to them.
- The floating-point behavior described in the section NaN handling and the Default NaN does not apply to them.
- When FPCR.AH is 1, the sign of a NaN input is not changed. When FPCR.AH is 0, the sign of a NaN input can be changed.

A1.5.9.2 Input Denormal exceptions

The cumulative floating-point exception bit FPSR.IDC, and the trap enable bit FPCR.IDE both relate to Input Denormal exceptions.

Input Denormal exceptions are not generated for denormalized half-precision inputs. For BFloat16 denormalized inputs and FP8 denormalized inputs, see Summary of BFloat16 instruction behaviors and Summary of FP8 instruction behaviors respectively. For single-precision and double-precision inputs:

- When FPCR.{AH, FZ, FIZ} is {0, 0, x}, Input Denormal exceptions are not generated.
- When FPCR.{AH, FZ, FIZ} is {0, 1, x}, an Input Denormal exception is generated when an input is flushed to zero.
- When FPCR.{AH, FZ, FIZ} is {1, x, 1}, Input Denormal exceptions are not generated.
- When FPCR.{AH, FZ, FIZ} is {1, x, 0}, an operation with a denormalized input generates an Input Denormal exception unless any of the following are true:
  - Another operand to the operation is a NaN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The operation generates a Divide-by-Zero exception.
- The operation generates an Invalid Operation exception.
- The instruction that generated the operation was one of:
  - Advanced SIMD&FP convert from single-precision to BFloat16: BFCVT, BFCVTN, BFCVTN2.
  - Advanced SIMD&FP convert to integer: `FCVTAS`, `FCVTAU`, `FCVTMS`, `FCVTMU`, `FCVTNS`, `FCVTNU`, `FCVTPS`, `FCVTPU`, `FCVTZS`, `FCVTZU`.
  - Advanced SIMD&FP round to integral: `FRINT32X`, `FRINT32Z`, `FRINT64X`, `FRINT64Z`, `FRINTA`, `FRINTI`, `FRINTM`, `FRINTN`, `FRINTP`, `FRINTX`, `FRINTZ`.
  - Advanced SIMD&FP JavaScript convert to signed fixed-point: FJCVTZS.
  - SVE convert from single-precision to BFloat16: BFCVT, BFCVTNT.
  - SVE convert to integer: FCVTZS, FCVTZU.
  - SVE round to integral: FRINT<r>.
  - SME convert from single-precision to BFloat16: BFCVT, BFCVTN.
  - SME convert to integer: FCVTZS, FCVTZU.
  - SME round to integral: FRINTA, FRINTM, FRINTN, FRINTP.

When a denormalized input is flushed to zero, the occurrence of floating-point exceptions other than Input Denormal is determined by treating the input value that is flushed to zero as zero.

A1.5.9.3 Inexact exceptions

The cumulative floating-point exception bit FPSR.IXC and the trap enable bit FPCR.IXE both relate to Inexact exceptions.

If a denormalized output is flushed to zero, all of the following apply:

- If FPCR.AH is 1, an Inexact exception is generated.
- If FPCR.AH is 0, an Inexact exception is not generated.

If a result is not flushed to zero, and the result does not equal the result calculated with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

A1.5.9.4 Underflow exceptions

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPCR.UFE both relate to Underflow exceptions.

If FPCR.AH is 0, for the purpose of Underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

If FPCR.AH is 1, for the purpose of Underflow floating-point exception generation, a denormalized number is detected after rounding with an unbounded exponent.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then:

- If the result is exact, an Underflow floating-point exception is generated only if FPCR.UFE is 1.
- If the result is inexact, an Underflow floating-point exception is always generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is generated, however this floating-point exception is not trapped regardless of the value of FPCR.UFE.

A1.5.9.5 Overflow exceptions

The cumulative floating-point exception bit FPSR.OFC, and the trap enable bit FPCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an Overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the selected rounding behavior and the sign of the result before rounding, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding, except for FP8 instructions, for which, see Summary of FP8 instruction behaviors.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision's largest finite number with the sign of the result before rounding.
- Round to Nearest Ties to Away carries all overflows to infinity with the sign of the result before rounding.
- Round to Odd, when used by convert instructions, carries all overflows to the output precision's largest finite number with the sign of the result before rounding.
- Round to Odd, when used by some BFloat16 instructions, carries all overflows to infinity with the sign of the result before rounding.

A1.5.9.6 Divide by Zero exceptions

The cumulative floating-point exception bit FPSR.DZC, and the trap enable bit FPCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite nonzero number by zero, a Divide by Zero exception is generated.

If a floating-point operation divides a finite nonzero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

A1.5.9.7 Invalid Operation exceptions

The cumulative floating-point exception bit FPSR.IOC, and the trap enable bit FPCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN.
- Subtracting two equal-magnitude infinities that are of the same sign.
- When FPCR.AH is 0, multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Taking the square root of an operand that is less than zero.

When FPCR.AH is 1, a floating-point instruction multiplying a zero by an infinity:

- Generates an Invalid Operation exception if the instruction is one of the following fused multiply-add instructions and the addend is not a quiet NaN:
  - Advanced SIMD `FMLA`, `FMLAL`, `FMLAL2`, `FMLS`, `FMLSL`, `FMLSL2`.
  - SVE `FMLA`, `FMLALB`, `FMLALT`, `FMLS`, `FMLSLB`, `FMLSLT`, `BFMLA`, `BFMLS`.
- Generates an Invalid Operation exception otherwise.

If the input is one of: a quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation exception.

For the signaling compare instructions FCMPE and FCCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

If FPCR.AH is 1, for FMAX (vector), FMAX (scalar), FMAXP (scalar), FMAXP (vector), FMAXV, FMIN (vector), FMIN (scalar), FMINP (scalar), FMINP (vector), FMINV, BFMAX, and BFMIN, if either input is any type of NaN, then an Invalid Operation floating-point exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.9.8 Handling floating-point exceptions

If an implementation supports synchronous exception generation for floating-point exceptions, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception and that is not at a lower Exception level than where the exception was generated.

If an implementation supports synchronous exception generation for floating-point exceptions in AArch64 state, all of the following apply:

- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception, except that an implementation is permitted to not restore the cumulative floating-point exception bits in the event of such an exception.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported.

The `AArch64_FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.

A1.5.9.9 Combinations of floating-point exceptions

More than one floating-point exception might occur on the same operation. The architecture permits only the following combinations of floating-point exceptions:

- Overflow with Inexact.
- Underflow with Inexact.
- If FPCR.AH is 0, Input Denormal with one or more of the other floating-point exception types.
- If FPCR.AH is 1, Input Denormal with Inexact, Underflow, or Overflow.

If two floating-point exceptions occur on the same operation, the Input Denormal exception is treated as highest priority and the Inexact exception is treated as lowest priority.

Some floating-point instructions specify more than one floating-point operation, this is indicated by the pseudocode descriptions of the instruction. In these cases, it is possible for one instruction to generate multiple exceptions. Multiple exceptions from one instruction are prioritized as follows:

- If an exception generating operation outputs a result that is used by a second exception generating operation, the exception generated by the operation that outputs the result is treated as higher priority than the exception generated by the second operation that uses the result.
- If exception generating operations do not use the outputs of other exception generating operations, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority. The exception prioritized might differ between different instances of the same two floating-point exceptions being generated on the same operation during execution of the instruction.
- A trapped Underflow exception has priority over a trapped Inexact exception.

If none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative exception flag bit in the FPSR to be set to 1.

When FPCR.AH is 1, if the result of a floating-point operation is a denormalized number that is flushed to zero, both an untrapped Underflow floating-point exception and an Inexact floating-point exception are generated, which always sets the cumulative Underflow exception flag FPSR.UFC to 1 regardless of whether the Inexact floating-point exception is trapped. Otherwise, when a trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value it had immediately before the instruction that generated the trapped floating-point exception. If the IMPLEMENTATION DEFINED choice is that the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which trapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative exception flag bits being set to 1.

When a trapped floating-point exception is taken, then in the ESR_ELx, all of the following apply:

- The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
- If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

The architectural requirements for floating-point exception prioritization apply only to multiple floating-point exceptions generated on the same element of an Advanced SIMD, SVE or SME operation. For trapped floating-point exceptions from Advanced SIMD, SVE or SME instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.

A1.5.10 Alternate BFloat16 behaviors

The behaviors in this section apply to the following instructions:

- Advanced SIMD&FP: `BFCVT`, `BFCVTN`, `BFCVTN2`, `BFMLALB`, `BFMLALT`.
- SVE: `BFCVT`, `BFCVTNT`, `BFMLALB`, `BFMLALT`, `BFMLSLB`, `BFMLSLT`.
- SME: `BFCVT` (single-precision to BFloat16), `BFCVTN`.

Table A1-6 and Table A1-7 describe when these instructions are legal.

When FPCR.AH is 1, the instructions:

- Produce the expected IEEE 754 default result but do not update the FPSR cumulative exception flag bits.
- Disable trapped floating-point exceptions, as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all 0.
- Use Round to Nearest Even, ignoring FPCR.RMode.
- Flush denormalized inputs and outputs to zero, as if FPCR.{FZ, FIZ} is {1, 1}.

A1.5.11 BFloat16 behaviors for instructions that calculate sum-of-products

The behaviors in this section apply to the following instructions:

- Advanced SIMD: `BFDOT` and `BFMMLA`.
- SVE: `BFDOT` and `BFMMLA`.
- SME: `BFMOPA` (widening) and `BFMOPS` (widening), and SME2 `BFDOT` and `BFVDOT`.

Table A1-6 and Table A1-7 describe when these instructions are legal.

When FPCR.EBF is 0, the instructions:

- Perform unfused two-way sum-of-products for each pair of adjacent BFloat16 elements in the source vectors, with rounding of all intermediate products and sums.
- Ignore FPCR.RMode and use non-IEEE 754 Round to Odd behavior. See Round to Odd.
- Flush denormalized inputs and outputs to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Disable alternative floating-point behaviors, as if FPCR.AH is 0.

When FPCR.EBF is 1, the instructions:

- Perform a fused two-way sum-of-products for each pair of adjacent BFloat16 elements in the source vectors, without rounding of the intermediate products, but rounding the single-precision sum before addition to the single-precision accumulator element.
- Honor FPCR.RMode, supporting all four IEEE 754 rounding modes.
- Honor FPCR.{FZ, FIZ}.
- Honor FPCR.AH.

Regardless of the value of FPCR.EBF, the instructions:

- Produce the expected IEEE 754 single-precision default result but do not update the FPSR cumulative exception flag bits.
- Disable trapped floating-point exceptions, as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Generate an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.
- Generate an intermediate sum-of-products that is a Default NaN when any of the following are true:
  - Any multiplier input is a NaN.
  - Any product is infinity x 0.0.
  - Both products are infinity of differing signs.
- Generate a Default NaN, as if FPCR.DN is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.6 The Arm memory model

The Arm memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating *virtual addresses* (VAs) provided by executing instructions to *physical addresses* (PAs).
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.
- Barriers that control and prevent speculative access to memory.

VA support depends on the Execution state, as follows:

AArch64 state

Supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

AArch32 state

Supports 32-bit virtual addressing, with the Translation Control Register determining the supported VA range. For execution at EL1 and EL0, system software can split the VA range into two subranges, each with its own translation controls.

The supported PA space is IMPLEMENTATION DEFINED, and can be discovered by system software.

Regardless of the Execution state, the *Virtual Memory System Architecture* (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported PA space.

For more information, see:

For execution in AArch64 state

- The AArch64 Application Level Memory Model.
- The AArch64 System Level Memory Model.
- The AArch64 Virtual Memory System Architecture.

For execution in AArch32 state

- The AArch32 Application Level Memory Model.
- The AArch32 System Level Memory Model.
- The AArch32 Virtual Memory System Architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.7 Reliability, Availability, and Serviceability

A1.7.1 Introduction

I_{LMHPD} RAS are three aspects of the *dependability* of a system:

- Reliability, that is, the continuity of correct service.
- Availability, that is, the readiness for correct service.
- Serviceability, that is, the ability to undergo modifications and repairs.

I_{HWHJM} RAS techniques reduce unplanned outages because:

- Transient errors can be detected and corrected before they cause application or system failure.
- Failing components can be identified and replaced.
- Failure can be predicted ahead-of-time to allow replacement during planned maintenance.

A1.7.2 Faults, errors, and failures

R_{NVNNC} Correct service is delivered when the service implements the system function.

I_{QWSVK} Correct service might include:

- Producing correct results.
- Producing results within the time allotted to the task.
- Not divulging secret or secure information.

R_{KRTSC} For the purpose of describing the RAS Extension and RAS System Architecture, deviation from correct service is defined using the following terms:

- A failure is the event of deviation from correct service. This includes data corruption, data loss, and service loss.
- An error is the deviation from correct service. An incorrect value that has an error is corrupt.
- A fault is the cause of the error.

R_{JNBDX} Errors that are present but not detected are latent errors or undetected errors.

I_{TNQPK} In a system with no error detection, all errors are latent errors and are silently propagated by components until they are either masked or cause failure.

I_{GRYKV} The severity of a failure can range from minor to catastrophic:

- The harmful consequences of a minor failure are of a similar cost to the benefits provided by correct service delivery.
- The harmful consequences of a catastrophic failure are orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

I_{NMGPQ} There are many sources of faults in a system, including both software and hardware faults:

- *Hardware faults* originate in, or affect, hardware.
- *Software faults* affect software, that is programs or data.

The RAS Extension and RAS System Architecture primarily address errors produced from hardware faults. These fall into two main areas:

- Transient faults.
- Non-transient or *persistent faults*.

A1.7.3 General taxonomy of errors

A1.7.3.1 Error detection

R_{FHXWP} When a component accesses memory or other state, an error might be detected in that memory or state.

I_{WKPVR} The error might be corrected or deferred by the component, or signaled to another component as either a Deferred error or a Detected error.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.7.3.2 Error propagation

R_{LRZDN} A transaction occurs when a producer of the transaction passes a value or other signal to a consumer of the transaction.

I_{VYCCX} Transactions are part of the service provided by the producer for the consumer.

I_{RHGDL} In many protocols and service interface definitions, a high-level transaction consists of a sequence of operations, for instance between a *Requester* and a *Completer*.

For the purposes of this manual, the most basic form of a unidirectional transfer between a *producer* and *consumer* is considered as a transaction.

That is, each one of the sequence of operations is considered a separate transaction. For some operations, such as a request, the Requester is producer and the Completer is the consumer. For other operations, such as a response, the Completer is producer and the Requester is the consumer.

R_{SKZZG} An error is propagated by the producer of a transaction when the service interface is incorrect because of the error. The error is propagated to the consumer.

R_{CHCCV} An error is propagated by deviations from correct service, including when any of the following occurs that would not have been permitted to occur had the fault not been activated:

- A corrupt value is passed from producer to consumer.
- A transaction or other operation occurs that should not have occurred.
- A transaction or other operation that should have occurred does not occur.
- A loss of uniprocessor semantics or any other loss of coherency in a multiprocessor coherent system is observed.
- Changing the timing and/or order of transactions or other operations such that the timing and/or order of those transactions or operations is incorrect. In this case, the service interface defines acceptable timings and/or orders for transactions and other operations.

R_{YFBRV} The service interface for a transaction might include means to signal that the transaction is propagating either of the following:

- A Detected error.
- A Deferred error.

R_{BHWVX} An error is silently propagated by the producer of a transaction if the consumer of the transaction cannot detect the error and consumes an undetected error because of the transaction. This might be because of one of the following:

- The error is present on the transaction, but was not detected by the producer. The error is silently propagated by the producer.
- The error is present on the transaction, but was not signaled to the consumer as an error. For example, a corrupt value was passed in the transaction with no indication that it was corrupt. The error is silently propagated by the producer.

R_{FPBYS} A latent, possibly detectable, error is silently propagated by the consumer of an otherwise correct transaction if the transaction causes the error to become undetectable.

Example A1-2

A partial write to a protection granule removes poison, leaving the unchanged portion of the location corrupt. To implement a partial write, the consumer logically reads the current value of the location, modifies the value, and then writes the modified value back. These are internal transactions in the consumer that silently propagate the error. In this example there was no error at the producer nor on the transaction.

I_{QXRLB} Errors might be propagated by components in a system until one of the following occurs:

- They are masked and do not affect the outcome of the system.
  The error might be masked because a corrupt value is discarded or overwritten, or the error is detected and removed.
- They affect the service interface of the system and possibly cause failure. If the error has been silently propagated to the service interface, then:
  - This is a Silent Data Corruption, SDC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The rate of such failures, measured as the number of failures per billion device-hours of operation, is called the Silent Data Corruption Failure-in-Time rate, SDC FIT rate.

Alternatively, the error might have been detected, causing the system to invoke error handling and recovery. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

A1.7.3.3 Infected and poisoned

R_{KNHWB} The state of a component becomes infected when the component consumes an Uncorrected error that updates the state.

R_{TZBSW} A value is poisoned in the state of a component if it is marked as being in error, such that a subsequent access of the state will detect the value is so marked and is treated as a Detected error.

I_{YBMFK} Poison is used to defer an error.

A1.7.3.4 Containable and uncontainable

R_{DXQRD} An undetected error is uncontained at the component that failed to detect it.

R_{RJYRQ} A silently propagated error is uncontained at the component that silently propagated it.

R_{GJQNR} A Detected Uncorrected error is uncontainable at the component if it might be uncontained at the component. A Detected Uncorrected error is containable at the component if it is not uncontainable at the component. If the component cannot determine whether a Detected Uncorrected error is uncontainable at the component or containable at the component, then the component treats the Detected Uncorrected error as uncontainable at the component.

I_{MRDMR} An error that is uncontainable at the component might be containable at the system level.

I_{NWZGB} Reporting an error as containable allows software to contain the error. This does not mean that hardware has contained the error.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter A2
A-profile Architecture Extensions

This chapter introduces the Arm A-profile architecture extensions. It contains the following sections:

- About the A-profile architecture extensions.
- Armv8-A architecture extensions.
- Armv9-A architecture extensions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.1 About the A-profile architecture extensions

An architecture extension defines a set of features that, when implemented, may make an implementation compliant with the extension.

Each feature description includes:

- A feature name.
- A brief description of the feature.
- The Execution state the feature is supported in.
- Whether the implementation of the feature is mandatory or OPTIONAL.
- Dependencies on the implementation of other features if any.
- The register field that identifies the presence of the feature.

A2.1.1 Permitted implementation of subsets of the A-profile architectural features

Arm regularly introduces new features to the architecture. When a feature is introduced, the architecture specifies when the feature can be implemented. Typically, a feature introduced as part of Armv8.*x* or Armv9.*x* is permitted to be built in Armv8.(*x*-1) or Armv9.(*x*-1), subject only to those constraints that require that certain features be implemented together.

See the individual feature descriptions for details of when each feature is permitted or required to be built.

A2.1.2 Withdrawn architecture features

This section lists features previously described in the architecture but subsequently withdrawn. Features listed in this section are not permitted to be included in any version of the architecture:

FEAT_TME, Transactional Memory Extension

FEAT_TME introduced a set of instructions to support hardware transaction memory, which means a group of instructions can appear to be collectively executed as a single atomic operation.

FEAT_SEBEP, Synchronous Exception-based Event Profiling

FEAT_SEBEP introduced support for synchronous, precise Performance Monitors Unit (PMU) profiling by permitting profiling events to be reported through synchronous exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2 Armv8-A architecture extensions

A2.2.1 The Armv8.0 architecture extension

The original Armv8-A architecture is called Armv8.0. It contains mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.0 compliant if it includes all of the Armv8.0 architectural features that are mandatory.

An Armv8.0 compliant implementation can additionally include:

- Armv8.0 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.1, subject only to those constraints that require that certain features be implemented together.

FEAT_AA32, PE supports AArch32.

FEAT_AA32 is OPTIONAL.

If FEAT_AA32 is implemented, then FEAT_AA32EL0 is implemented.

FEAT_AA32EL0, Support for AArch32 at EL0

FEAT_AA32EL0 is OPTIONAL from Armv8.0.

If FEAT_AA32EL0 is implemented, then FEAT_EL0 is implemented.

If FEAT_AA32EL0 is implemented, then FEAT_AA32 is implemented.

FEAT_AA32EL1, Support for AArch32 at EL1

FEAT_AA32EL1 is OPTIONAL from Armv8.0.

If Armv9.0 is implemented, then FEAT_AA32EL1 is not implemented.

If FEAT_AA32EL1 is implemented, then FEAT_AA32EL0 is implemented.

If FEAT_AA32EL1 is implemented, then FEAT_EL1 is implemented.

FEAT_AA32EL2, Support for AArch32 at EL2

FEAT_AA32EL2 is OPTIONAL from Armv8.0.

If FEAT_AA32EL2 is implemented, then FEAT_AA32EL1 is implemented.

If FEAT_AA32EL2 is implemented, then FEAT_EL2 is implemented.

FEAT_AA32EL3, Support for AArch32 at EL3

FEAT_AA32EL3 is OPTIONAL from Armv8.0.

If FEAT_AA32EL3 is implemented, then FEAT_AA32EL1 is implemented.

When FEAT_AA32EL3 and FEAT_EL2 are implemented, FEAT_AA32EL2 is implemented.

If FEAT_AA32EL3 is implemented, then FEAT_EL3 is implemented.

FEAT_AA64, PE uses AArch64 after last reboot

FEAT_AA64 is OPTIONAL from Armv8.0.

If FEAT_AA64 is implemented, then FEAT_AA64EL0, FEAT_AA64EL1, FEAT_AA64EL2, or FEAT_AA64EL3 is implemented.

When FEAT_AA64EL0, FEAT_AA64EL1, FEAT_AA64EL2, or FEAT_AA64EL3 is implemented, FEAT_AA64 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AA64EL0, Support for AArch64 at EL0

FEAT_AA64EL0 is OPTIONAL from Armv8.0.

FEAT_AA64EL0 is mandatory from Armv9.0.

If FEAT_AA64EL0 is implemented, then FEAT_AA64EL1 is implemented.

If FEAT_AA64EL0 is implemented, then FEAT_EL0 is implemented.

FEAT_AA64EL1, Support for AArch64 at EL1

FEAT_AA64EL1 is OPTIONAL from Armv8.0.

FEAT_AA64EL1 is mandatory from Armv9.0.

If FEAT_AA64EL1 is implemented, then FEAT_AA64EL0 is implemented.

When FEAT_AA64EL1 and FEAT_EL2 are implemented, FEAT_AA64EL2 is implemented.

When FEAT_AA64EL1 and FEAT_EL3 are implemented, FEAT_AA64EL3 is implemented.

If FEAT_AA64EL1 is implemented, then FEAT_EL1 is implemented.

FEAT_AA64EL2, Support for AArch64 at EL2

FEAT_AA64EL2 is OPTIONAL from Armv8.0.

If FEAT_AA64EL2 is implemented, then FEAT_AA64EL1 is implemented.

When FEAT_AA64EL2 and FEAT_EL3 are implemented, FEAT_AA64EL3 is implemented.

If FEAT_AA64EL2 is implemented, then FEAT_EL2 is implemented.

FEAT_AA64EL3, Support for AArch64 at EL3

FEAT_AA64EL3 is OPTIONAL from Armv8.0.

If FEAT_AA64EL3 is implemented, then FEAT_AA64EL1 is implemented.

When FEAT_AA64EL3 and FEAT_EL2 are implemented, FEAT_AA64EL2 is implemented.

If FEAT_AA64EL3 is implemented, then FEAT_EL3 is implemented.

FEAT_AES, Advanced SIMD AES instructions

FEAT_AES provides the AES* instructions to support AES encryption and decryption, AESD, AESE, AESMC, and AESIMC.

This feature is supported in both AArch64 and AArch32 states.

FEAT_AES is OPTIONAL from Armv8.0.

If FEAT_AES is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_AES:

- ID_AA64ISAR0_EL1.AES.

FEAT_ASID16, 16 bit ASID

FEAT_ASID16 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_ASID16:

- ID_AA64MMFR0_EL1.ASIDBits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AdvSIMD, Advanced SIMD Extension

FEAT_AdvSIMD includes support for the SISD and SIMD operations. All Armv8-A systems that support standard operating systems with rich application environments also provide hardware support for Advanced SIMD instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_AdvSIMD is OPTIONAL from Armv8.0.

If FEAT_AdvSIMD is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_AdvSIMD:

- ID_AA64PFR0_EL1.AdvSIMD.
- EDPFR.AdvSIMD.

For more information, see:

- Supported data types.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.
- Advanced SIMD and floating-point instructions.
- T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions.

FEAT_BigEnd, Support for big-endian at EL1 and above

FEAT_BigEnd is OPTIONAL from Armv8.0.

If FEAT_BigEnd is implemented, then FEAT_BigEndEL0 is implemented.

If FEAT_MixedEnd is implemented, then FEAT_BigEnd is implemented.

FEAT_BigEndEL0, Support for big-endian at EL0

FEAT_BigEndEL0 is OPTIONAL from Armv8.0.

If FEAT_MixedEndEL0 is implemented, then FEAT_BigEndEL0 is implemented.

FEAT_CRC32, CRC32 instructions

FEAT_CRC32 introduces the support for the CRC32* instructions that perform cyclic redundancy check calculations.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CRC32 is OPTIONAL from Armv8.0.

FEAT_CRC32 is mandatory from Armv8.1.

The following fields identify the presence of FEAT_CRC32:

- ID_AA64ISAR0_EL1.CRC32.
- ID_ISAR5_EL1.CRC32.
- ID_ISAR5.CRC32.

FEAT_CSV2, Cache Speculation Variant 2

FEAT_CSV2 introduces a mechanism to identify if hardware cannot disclose information about whether any prediction resource, such as branch target, data-value, or cache prefetch, trained in one hardware described context can control speculative execution in a different hardware described context.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CSV2 is OPTIONAL from Armv8.0.

FEAT_CSV2 is mandatory from Armv8.5.

The following fields identify the presence of FEAT_CSV2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64PFR0_EL1.CSV2.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

For more information, see:

- For AArch64, Restrictions on the effects of speculation.
- AArch32 restrictions on the effects of speculation.

FEAT_CSV2_1p1, Cache Speculation Variant 2

For FEAT_CSV2_1p1, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p1 does not support the SCXTNUM_ELx registers.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CSV2_1p1 is OPTIONAL from Armv8.0.

If FEAT_CSV2_1p1 is implemented, then FEAT_CSV2 is implemented.

The following fields identify the presence of FEAT_CSV2_1p1:

- ID_AA64PFR1_EL1.CSV2_frac.
- ID_AA64PFR0_EL1.CSV2.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

For more information, see:

- For AArch64, Restrictions on the effects of speculation.
- AArch32 restrictions on the effects of speculation.

FEAT_CSV2_1p2, Cache Speculation Variant 2 version 1.2

For FEAT_CSV2_1p2, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p2 introduces the SCXTNUM_ELx registers, but the contexts do not include the SCXTNUM_ELx register contexts.

This feature is supported in AArch64 state only.

FEAT_CSV2_1p2 is OPTIONAL from Armv8.0.

If FEAT_CSV2_1p2 is implemented, then FEAT_CSV2_1p1 is implemented.

The following fields identify the presence of FEAT_CSV2_1p2:

- ID_AA64PFR1_EL1.CSV2_frac.
- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, Restrictions on the effects of speculation.
- AArch32 restrictions on the effects of speculation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_CSV2_2, Cache Speculation Variant 2 version 2

FEAT_CSV2_2 introduces the SCXTNUM_ELx registers, which provide a number that can be used to separate out different context numbers within their respective Exception levels for the purpose of protecting against speculative attacks through prediction resources.

This feature is supported in AArch64 state only.

FEAT_CSV2_2 is OPTIONAL from Armv8.0.

If FEAT_CSV2_2 is implemented, then FEAT_CSV2 is implemented.

If FEAT_CSV2_2 is implemented, then FEAT_CSV2_1p1 is not implemented.

The following field identifies the presence of FEAT_CSV2_2:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, Restrictions on the effects of speculation.
- AArch32 restrictions on the effects of speculation.

FEAT_CSV2_3, Cache Speculation Variant 2 version 3

FEAT_CSV2_3 introduces a mechanism to identify if hardware cannot disclose information about whether prediction resources based on branch history trained in one hardware described context can control speculative execution in a different hardware described context.

This feature is supported in AArch64 state only.

FEAT_CSV2_3 is OPTIONAL from Armv8.0.

If FEAT_CSV2_3 is implemented, then FEAT_CSV2_2 is implemented.

The following field identifies the presence of FEAT_CSV2_3:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, Restrictions on the effects of speculation.
- AArch32 restrictions on the effects of speculation.

FEAT_Crypto, Cryptographic Extension

The Arm Cryptographic Extension provides instructions for the acceleration of encryption and decryption. The presence of the Cryptographic Extension in an implementation is subject to export license controls.

The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file. It also provides multiply instructions that operate on long polynomials.

In an implementation that supports both AArch64 state and AArch32 state, FEAT_AES, FEAT_PMULL, FEAT_SHA1 and FEAT_SHA256 provide the same functionality in both states.

This feature is supported in both AArch64 and AArch32 states.

FEAT_Crypto is OPTIONAL.

If FEAT_Crypto is implemented, then FEAT_AES or FEAT_SHA1 is implemented.

In an Armv8.2 implementation, if FEAT_Crypto is implemented, FEAT_PMULL, FEAT_SHA256, FEAT_SM3, or FEAT_SM4 is implemented.

For more information, see:

- The Cryptographic Extension in AArch64 state.
- The Cryptographic Extension in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_DoubleLock, Double Lock

FEAT_DoubleLock is the mnemonic used for the OS Double Lock.

FEAT_DoubleLock is OPTIONAL from Armv8.0.

If Armv9.0 is implemented, then FEAT_DoubleLock is not implemented.

The following fields identify the presence of FEAT_DoubleLock:

- ID_AA64DFR0_EL1.DoubleLock.
- DBGDEVID.DoubleLock.

FEAT_EL0, Support for execution at EL0

FEAT_EL0 is mandatory from Armv8.0.

If FEAT_EL0 is implemented, then FEAT_AA32EL0 or FEAT_AA64EL0 is implemented.

FEAT_EL1, Support for execution at EL1

FEAT_EL1 is mandatory from Armv8.0.

If FEAT_EL1 is implemented, then FEAT_AA32EL1 or FEAT_AA64EL1 is implemented.

FEAT_EL2, Support for execution at EL2

FEAT_EL2 is OPTIONAL from Armv8.0.

If FEAT_EL2 is implemented, then FEAT_AA32EL2 or FEAT_AA64EL2 is implemented.

FEAT_EL3, Support for EL3

FEAT_EL3 is OPTIONAL from Armv8.0.

If FEAT_EL3 is implemented, then FEAT_AA32EL3 or FEAT_AA64EL3 is implemented.

FEAT_ETMv4, Embedded Trace Macrocell version 4

FEAT_ETMv4 indicates support for the Embedded Trace Macrocell architecture ETMv4.

FEAT_ETMv4 is OPTIONAL from Armv8.0.

If Armv9.0 is implemented, then FEAT_ETMv4 is not implemented.

If FEAT_ETMv4 is implemented, then FEAT_TRC_SR or FEAT_TRC_EXT is implemented.

For more information, see the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4(ARM IHI 0064)*.

FEAT_ETS2, Enhanced Translation Synchronization

FEAT_ETS2 introduces support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ETS2 is OPTIONAL from Armv8.0.

FEAT_ETS2 is mandatory from Armv8.8.

The following fields identify the presence of FEAT_ETS2:

- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
- ID_MMFR5.ETS.

For more information, see:

- Ordering requirements defined by the formal concurrency model.
- Ordering of memory accesses from translation table walks.
- Ordering of translation table walks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FP, Floating Point extensions

FEAT_FP includes support for single-precision and double-precision floating-point types. All Armv8-A systems that support standard operating systems with rich application environments also provide hardware support for floating-point instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_FP is OPTIONAL from Armv8.0.

If FEAT_FP is implemented, then FEAT_AdvSIMD is implemented.

The following fields identify the presence of FEAT_FP:

- ID_AA64PFR0_EL1.FP.
- EDPFR.FP.

For more information, see:

- Supported data types.
- Floating-point support.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.
- Advanced SIMD and floating-point instructions.
- T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions.

FEAT_IVIPT, The IVIPT Extension

FEAT_IVIPT describes any permitted instruction cache implementation. This includes *Virtual Index, Physical tag* (VIPT) cache policy and *Physical Index, Physical Tag* (PIPT) cache policy.

FEAT_IVIPT is mandatory from Armv8.0.

For more information, see:

- Instruction caches.
- Instruction caches.

FEAT_LittleEnd, Support for little-endian at EL1 and above

FEAT_LittleEnd is OPTIONAL from Armv8.0.

If FEAT_LittleEnd is implemented, then FEAT_LittleEndEL0 is implemented.

If FEAT_MixedEnd is implemented, then FEAT_LittleEnd is implemented.

FEAT_LittleEndEL0, Support for little-endian at EL0

FEAT_LittleEndEL0 is OPTIONAL from Armv8.0.

If FEAT_MixedEndEL0 is implemented, then FEAT_LittleEndEL0 is implemented.

FEAT_MixedEnd, Mixed-endian support

FEAT_MixedEnd provides support for mixed-endian configuration.

FEAT_MixedEnd is OPTIONAL from Armv8.0.

If FEAT_MixedEnd is implemented, then FEAT_MixedEndEL0 is implemented.

The following field identifies the presence of FEAT_MixedEnd:

- ID_AA64MMFR0_EL1.BigEnd.

For more information, see:

- Mixed-endian support in AArch64.
- Mixed-endian support in AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MixedEndEL0, Mixed-endian support at EL0

FEAT_MixedEndEL0 provides support for mixed-endian at EL0.

FEAT_MixedEndEL0 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_MixedEndEL0:

- ID_AA64MMFR0_EL1.BigEndEL0.
- ID_AA64MMFR0_EL1.BigEnd.

For more information, see:

- Mixed-endian support in AArch64.
- Mixed-endian support in AArch32.

FEAT_PCSRv8, PC Sample-based Profiling extension

FEAT_PCSRv8 introduces support for PC Sample-based Profiling Extension that provides coarse-grained, non-invasive profiling by an external debugger.

FEAT_PCSRv8 is OPTIONAL from Armv8.0.

If FEAT_PCSRv8 is implemented, then FEAT_PCSRv8p2 is not implemented.

The following fields identify the presence of FEAT_PCSRv8:

- DBGDEVID.PCSample.
- DBGDEVID1.PCSROffset.
- EDDEVID1.PCSROffset.
- EDDEVID.PCSample.

For more information, see About the PC Sample-based Profiling Extension.

FEAT_PMULL, Advanced SIMD PMULL instructions

FEAT_PMULL provides the AES* instructions that support multiplication of 64-bit polynomials, PMULL, PMULL2.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMULL is OPTIONAL from Armv8.0.

If FEAT_PMULL is implemented, then FEAT_AES is implemented.

The following field identifies the presence of FEAT_PMULL:

- ID_AA64ISAR0_EL1.AES.

FEAT_PMUv3, PMU extension version 3

The Performance Monitors Extension, FEAT_PMUv3, is an optional non-invasive debug component.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_PMUv3:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see The Performance Monitors Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3_EXT, External interface to the Performance Monitors

FEAT_PMUv3_EXT indicates support for external access to the FEAT_PMUv3 and/or FEAT_PCSRv8p2 registers.

FEAT_PMUv3_EXT is OPTIONAL from Armv8.0.

If FEAT_PMUv3_EXT is implemented, then FEAT_PMUv3 or FEAT_PCSRv8p2 is implemented.

If FEAT_PMUv3_EXT is implemented, then FEAT_PMUv3_EXT32 or FEAT_PMUv3_EXT64 is implemented.

FEAT_PMUv3_EXT32, 32-bit external interface to the Performance Monitors

FEAT_PMUv3_EXT32 indicates the external Performance Monitors and CoreSight registers are implemented as mostly 32-bit registers.

FEAT_PMUv3_EXT32 is OPTIONAL from Armv8.0.

If FEAT_PMUv3_EXT32 is implemented, then FEAT_PMUv3_EXT is implemented.

If FEAT_PMUv3_EXT32 is implemented, then FEAT_PMUv3_EXT64 is not implemented.

The following field identifies the presence of FEAT_PMUv3_EXT32:

- PMDEVARCH.ARCHPART.

For more information, see Recommended External Interface to the Performance Monitors.

FEAT_SHA1, Advanced SIMD SHA1 instructions

FEAT_SHA1 implements the SHA1* instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SHA1 is OPTIONAL from Armv8.0.

If FEAT_SHA1 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SHA1:

- ID_AA64ISAR0_EL1.SHA1.

FEAT_SHA256, Advanced SIMD SHA256 instructions

FEAT_SHA256 implements the SHA256* instructions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SHA256 is OPTIONAL from Armv8.0.

If FEAT_SHA256 is implemented, then FEAT_SHA1 is implemented.

The following field identifies the presence of FEAT_SHA256:

- ID_AA64ISAR0_EL1.SHA2.

FEAT_Secure, Support for Secure state

FEAT_Secure adds support for Secure state.

This feature is not supported if EL2 is using AArch32.

FEAT_Secure is OPTIONAL from Armv8.0.

If FEAT_RME is not implemented and FEAT_EL3 is implemented, then FEAT_Secure is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SpecSEI, SError interrupt exceptions from speculative reads of memory

Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches.

FEAT_SpecSEI is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_SpecSEI:

- ID_AA64MMFR1_EL1.SpecSEI.
- ID_MMFR4_EL1.SpecSEI.
- ID_MMFR4.SpecSEI.

FEAT_TGran16K, Support for 16KB memory translation granule size at stage 1

FEAT_TGran16K is OPTIONAL from Armv8.0.

In an Armv8.5 implementation, if FEAT_AA64EL2 and FEAT_TGran16K are implemented, FEAT_S2TGran16K is implemented.

The following field identifies the presence of FEAT_TGran16K:

- ID_AA64MMFR0_EL1.TGran16.

FEAT_TGran4K, Support for 4KB memory translation granule size at stage 1

FEAT_TGran4K is OPTIONAL from Armv8.0.

In an Armv8.5 implementation, if FEAT_AA64EL2 and FEAT_TGran4K are implemented, FEAT_S2TGran4K is implemented.

The following field identifies the presence of FEAT_TGran4K:

- ID_AA64MMFR0_EL1.TGran4.

FEAT_TGran64K, Support for 64KB memory translation granule size at stage 1

FEAT_TGran64K is OPTIONAL from Armv8.0.

In an Armv8.5 implementation, if FEAT_AA64EL2 and FEAT_TGran64K are implemented, FEAT_S2TGran64K is implemented.

The following field identifies the presence of FEAT_TGran64K:

- ID_AA64MMFR0_EL1.TGran64.

FEAT_TRC_EXT, Trace external registers

FEAT_TRC_EXT indicates support for external access to the ETMv4 or ETE registers.

FEAT_TRC_EXT is OPTIONAL from Armv8.0.

If FEAT_TRC_EXT is implemented, then FEAT_ETMv4 or FEAT_ETE is implemented.

FEAT_TRC_SR, Trace System registers

FEAT_TRC_SR indicates support for System registers for ETMv4 or ETE.

This feature is supported in both AArch64 and AArch32, but only when either of these are supported at EL1.

FEAT_TRC_SR is OPTIONAL from Armv8.0.

If FEAT_TRC_SR is implemented, then FEAT_ETMv4 or FEAT_ETE is implemented.

The following fields identify the presence of FEAT_TRC_SR:

- ID_AA64DFR0_EL1.TraceVer.
- EDDFR.TraceVer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_DFR0.CopTrc.
- ID_DFR0_EL1.CopTrc.

FEAT_nTLBPA, Intermediate caching of translation table walks

FEAT_nTLBPA introduces a mechanism to identify if the intermediate caching of translation table walks does not include non-coherent caches of previous valid translation table entries since the last completed TLBI applicable to the PE.

This feature is supported in both AArch64 and AArch32 states.

FEAT_nTLBPA is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_nTLBPA:

- ID_AA64MMFR1_EL1.nTLBPA.
- ID_MMFR5_EL1.nTLBPA.
- ID_MMFR5.nTLBPA.

For more information, see:

- TLB maintenance.
- General TLB maintenance requirements.

A2.2.1.1 Features added to the Armv8.0 extension in later releases

- FEAT_CHK.
- FEAT_CLRBHB.
- FEAT_CP15SDISABLE2.
- FEAT_CSV3.
- FEAT_DGH.
- FEAT_ECBHB.
- FEAT_ETS3.
- FEAT_GTG.
- FEAT_PRFMSLC.
- FEAT_RAS.
- FEAT_RPRFM.
- FEAT_SB.
- FEAT_SCTLR2.
- FEAT_SPECRES2.
- FEAT_SPECRES.
- FEAT_SSBS2.
- FEAT_SSBS.
- FEAT_TCR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2.2 The Armv8.1 architecture extension

The Armv8.1 architecture extension is an extension to Armv8.0. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.1 compliant if all of the following apply:

- It is Armv8.0 compliant.
- It includes all of the Armv8.1 architectural features that are mandatory.

An Armv8.1 compliant implementation can additionally include:

- Armv8.1 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.2, subject only to those constraints that require that certain features be implemented together.

FEAT_Debugv8p1, Debug v8.1

FEAT_Debugv8p1 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_Debugv8p1:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

FEAT_E2H0, Programming of HCR_EL2.E2H

When FEAT_VHE is implemented, FEAT_E2H0 indicates that HCR_EL2.E2H can be programmed to the value 0. The absence of FEAT_E2H0 results in a relaxed behavior where HCR_EL2.E2H is RES1, with EL2 host mode always enabled.

This feature is supported in AArch64 state only.

FEAT_E2H0 is OPTIONAL from Armv8.0.

If FEAT_E2H0 is implemented, then FEAT_VHE is implemented.

If FEAT_E2H0 is implemented, then FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_E2H0:

- ID_AA64MMFR4_EL1.E2H0.

FEAT_HAF, Hardware management of the Access flag.

Hardware can perform updates to the translation tables to manage the Access flag.

This feature is supported in AArch64 state only.

FEAT_HAF is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_HAF:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see Hardware management of the Access flag.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_HAFDBS, Hardware management of dirty state

Hardware can perform updates to dirty state in the translation tables.

This feature is supported in AArch64 state only.

FEAT_HAFDBS is OPTIONAL from Armv8.0.

If FEAT_HAFDBS is implemented, then FEAT_HAF is implemented.

The following field identifies the presence of FEAT_HAFDBS:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see Hardware management of the dirty state.

FEAT_HPDS, Hierarchical permission disables in translations tables

FEAT_HPDS introduces the facility to disable the hierarchical permissions, APTable, PXNTable, and UXNTable, in the translation tables. This disable has no effect on the NSTable bit.

FEAT_HPDS is OPTIONAL from Armv8.0.

FEAT_HPDS is mandatory from Armv8.1.

The following field identifies the presence of FEAT_HPDS:

- ID_AA64MMFR1_EL1.HPDS.

This feature is added only to the VMSAv8-64 translation regimes. Armv8.2 extends this to the AArch32 translation regimes, see FEAT_AA32HPD.

FEAT_LOR, Limited ordering regions

Limited ordering regions allow large systems to perform special Load-Acquire and Store-Release instructions that provide order between the memory accesses to a region of the PA map as observed by a limited set of observers.

This feature is supported in AArch64 state only.

FEAT_LOR is OPTIONAL from Armv8.0.

FEAT_LOR is mandatory from Armv8.1.

The following field identifies the presence of FEAT_LOR:

- ID_AA64MMFR1_EL1.LO.

For more information, see Limited ordering regions.

FEAT_LSE, Large System Extensions

FEAT_LSE introduces a set of atomic instructions:

- Compare and Swap instructions, `CAS` and `CASP`.
- Atomic memory operation instructions, LD<OP> and ST<OP>, where <OP> is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, and UMIN.
- Swap instruction, SWP.

This feature is supported in AArch64 state only.

FEAT_LSE is OPTIONAL from Armv8.0.

FEAT_LSE is mandatory from Armv8.1.

The following field identifies the presence of FEAT_LSE:

- ID_AA64ISAR0_EL1.Atomic.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Atomic integer memory operations.
- Swap operations.
- Compare and Swap operations.

FEAT_PAN, Privileged access never

FEAT_PAN introduces a bit to PSTATE. When the value of this PAN state bit is 1, any privileged data access from EL1, or EL2 when the Effective value of HCR_EL2.E2H is 1, to a virtual memory address that is accessible to data accesses at EL0, generates a Permission fault.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PAN is OPTIONAL from Armv8.0.

FEAT_PAN is mandatory from Armv8.1.

The following fields identify the presence of FEAT_PAN:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

For more information, see:

- PSTATE.PAN.
- About the PAN bit.

FEAT_PMUv3p1, Armv8.1 PMU extensions

FEAT_PMUv3p1 introduces the following:

- The event number space is extended to 16 bits.
- The HPMD bit is added to MDCR_EL2. This bit disables event counting at EL2.
- The STALL_FRONTEND and STALL_BACKEND events are required to be implemented.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p1 is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p1 is implemented.

If FEAT_PMUv3p1 is implemented, then FEAT_PMUv3 is implemented.

The following fields identify the presence of FEAT_PMUv3p1:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see Required events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RDM, Advanced SIMD rounding double multiply-accumulate instructions

FEAT_RDM introduces Rounding Double Multiply Add/Subtract Advanced SIMD instructions. For more information, see:

For the A64 instruction set

- SQRDMLAH (by element).
- SQRDMLAH (vectors).
- SQRDMLSH (by element).
- SQRDMLSH (vector).

For the T32 and A32 instruction sets

- VQRDMLAH.
- VQRDMLSH.

This feature is supported in both AArch64 and AArch32 states.

FEAT_RDM is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_AdvSIMD is implemented, FEAT_RDM is implemented.

The following fields identify the presence of FEAT_RDM:

- ID_AA64ISAR0_EL1.RDM.
- ID_ISAR5_EL1.RDM.
- ID_ISAR5.RDM.

FEAT_VHE, Virtualization Host Extensions

Armv8.1 introduces the *Virtualization Host Extensions* (VHE) that provide enhanced support for Type 2 hypervisors in Non-secure state.

FEAT_VHE is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_AA64EL2 is implemented, FEAT_VHE is implemented.

If FEAT_VHE is implemented, then FEAT_LSE, FEAT_Debugv8p1, and FEAT_AA64EL2 are implemented.

The following field identifies the presence of FEAT_VHE:

- ID_AA64MMFR1_EL1.VH.

For more information, see Virtualization Host Extensions

FEAT_VMID16, 16-bit VMID

In an Armv8.1 implementation, when EL2 is using AArch64, the *virtual machine identifier* (VMID) size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

When implemented, this feature is supported only when EL2 is using AArch64.

FEAT_VMID16 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_VMID16:

- ID_AA64MMFR1_EL1.VMIDBits.

For more information, see VMID size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2.2.1 Features added to the Armv8.1 extension in later releases

- FEAT_DPB2.
- FEAT_DotProd.
- FEAT_FHM.
- FEAT_FlagM.
- FEAT_PAN3.

A2.2.3 The Armv8.2 architecture extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.2 compliant if all of the following apply:

- It is Armv8.1 compliant.
- It includes all of the Armv8.2 architectural features that are mandatory.

An Armv8.2 compliant implementation can additionally include:

- Armv8.2 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.3, subject only to those constraints that require that certain features be implemented together.

FEAT_AA32HPD, AArch32 Hierarchical permission disables

FEAT_HPDS introduced the ability to disable the hierarchical permissions, APTable, PXNTable, and UXNTable, in the VMSAv8-64 translation regimes. FEAT_AA32HPD extends this functionality to the VMSAv8-32 translation regimes when those regimes are using the Long descriptor Translation Table format.

This feature is supported in AArch32 state only.

FEAT_AA32HPD is OPTIONAL from Armv8.1.

If FEAT_AA32HPD is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_AA32HPD:

- ID_MMFR4_EL1.HPDS.
- ID_MMFR4.HPDS.

For more information, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors

FEAT_AA32I8MM, AArch32 Int8 matrix multiply-accumulate

FEAT_AA32I8MM introduces integer matrix multiply-accumulate instructions and dot product instructions.

This feature is supported in AArch32 state only.

FEAT_AA32I8MM is OPTIONAL from Armv8.1.

If FEAT_AA32I8MM is implemented, then FEAT_I8MM is implemented.

If FEAT_AA32I8MM is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_AA32I8MM:

- ID_ISAR6_EL1.I8MM.
- ID_ISAR6.I8MM.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Advanced SIMD dot product instructions.
- Advanced SIMD matrix multiply-accumulate instructions.

FEAT_ASMv8p2, Armv8.2 changes to the A64 ISA

FEAT_ASMv8p2 introduces the BFC instruction to the A64 instruction set as an alias of BFM. It also requires that the BFC instruction and the A64 pseudo-instruction REV64 are implemented by assemblers.

In Armv8.0 and Armv8.1, the A64 pseudo-instruction REV64 is OPTIONAL.

FEAT_ASMv8p2 is OPTIONAL from Armv8.1.

FEAT_ASMv8p2 is mandatory from Armv8.2.

For more information, see:

- BFC.
- REV64.

FEAT_DPB, DC CVAP instruction

FEAT_DPB introduces a mechanism to identify and manage persistent memory locations in a shared memory hierarchy, including adding the DC CVAP instruction.

This feature is supported in AArch64 state only.

FEAT_DPB is OPTIONAL from Armv8.1.

FEAT_DPB is mandatory from Armv8.2.

The following field identifies the presence of FEAT_DPB:

- ID_AA64ISAR1_EL1.DPB.

For more information, see Memory hierarchy.

FEAT_Debugv8p2, Debug v8.2

FEAT_Debugv8p2 provides the following changes:

- If the Core power domain is powered up and DoubleLockStatus() == TRUE, EDPRSR.{DLK,SPD,PU} is only permitted to read {UNKNOWN, 0, 0}.
- The definition of Exception Catch debug events is extended to include reset entry.
- All CONSTRAINED UNPREDICTABLE cases that generate Exception Catch debug events are removed.
- Controls are added to EDECCR to control Exception Catch debug event generation on exception return.
- All IMPLEMENTATION DEFINED control of external debug accesses to OSLAR_EL1 is removed.
- ExternalSecureNoninvasiveDebugEnabled() cannot override software controls of counting attributable events in Secure state.

FEAT_Debugv8p2 is OPTIONAL from Armv8.1.

FEAT_Debugv8p2 is mandatory from Armv8.2.

If FEAT_Debugv8p2 is implemented, then FEAT_Debugv8p1 is implemented.

The following fields identify the presence of FEAT_Debugv8p2:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- DBGDIDR.Version.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Exception Catch debug event.
- EDPRSR.{DLK, SPD, PU} and the Core power domain.
- Interaction with EL3.
- External access disabled.

FEAT_F32MM, Single-precision floating-point matrix multiply-accumulate

FEAT_F32MM introduces support for the SVE single-precision floating-point matrix multiply-accumulate variant of the FMMLA instruction.

This feature is supported in AArch64 state only.

FEAT_F32MM is OPTIONAL from Armv8.2.

If FEAT_F32MM is implemented, then FEAT_SVE is implemented.

The following field identifies the presence of FEAT_F32MM:

- ID_AA64ZFR0_EL1.F32MM.

FEAT_F64MM, Double-precision floating-point matrix multiply-accumulate

FEAT_F64MM introduces support for the following SVE instructions:

- FMMLA (double-precision floating-point matrix multiply-accumulate variant).
- LD1ROB (scalar plus immediate).
- LD1ROB (scalar plus scalar).
- LD1ROD (scalar plus immediate).
- LD1ROD (scalar plus scalar).
- LD1ROH (scalar plus immediate).
- LD1ROH (scalar plus scalar).
- LD1ROW (scalar plus immediate).
- LD1ROW (scalar plus scalar).
- TRN1, TRN2 (vectors) (128-bit variant).
- UZP1, UZP2 (vectors) (128-bit variant).
- ZIP1, ZIP2 (vectors) (128-bit variant).

This feature is supported in AArch64 state only.

FEAT_F64MM is OPTIONAL from Armv8.2.

If FEAT_F64MM is implemented, then FEAT_SVE is implemented.

The following field identifies the presence of FEAT_F64MM:

- ID_AA64ZFR0_EL1.F64MM.

FEAT_FP16, Half-precision floating-point data processing

FEAT_FP16 supports:

- Half-precision data-processing instructions for Advanced SIMD and floating-point.

The following fields enable flushing of denormalized numbers to zero for half-precision data-processing instructions:

- FPCR.FZ16.
- FPSCR.FZ16.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in both AArch64 and AArch32 states.

FEAT_FP16 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_FHM is not implemented, then FEAT_FP16 is not implemented.

In an Armv8.2 implementation, if FEAT_SVE or FEAT_FHM is implemented, FEAT_FP16 is implemented.

The following fields identify the presence of FEAT_FP16:

- ID_AA64PFR0_EL1.FP.
- ID_AA64PFR0_EL1.AdvSIMD.
- MVFR1_EL1.FPHP.
- MVFR1_EL1.SIMDHP.
- MVFR1.FPHP.
- MVFR1.SIMDHP.

For more information, see:

- Half-precision floating-point formats.
- Flushing denormalized numbers to zero.
- Modified immediate constants in A64 floating-point instructions.

FEAT_HPDS2, Hierarchical permission disables

FEAT_HPDS2 provides a mechanism to allow operating systems or hypervisors to make up to four bits of Translation Table final-level descriptors available for IMPLEMENTATION DEFINED hardware use.

This feature is supported in both AArch64 and AArch32 states.

FEAT_HPDS2 is OPTIONAL from Armv8.1.

When FEAT_AA32EL1 and FEAT_HPDS2 are implemented, FEAT_AA32HPD is implemented.

If FEAT_HPDS2 is implemented, then FEAT_HPDS is implemented.

The following fields identify the presence of FEAT_HPDS2:

- ID_AA64MMFR1_EL1.HPDS.
- ID_MMFR4_EL1.HPDS.
- ID_MMFR4.HPDS.

For more information, see:

- Page Based Hardware attributes.
- Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors.

FEAT_I8MM, AArch64 Int8 matrix multiply-accumulate

FEAT_I8MM introduces Advanced SIMD and SVE integer matrix multiply-accumulate instructions and dot product instructions.

This feature is supported in AArch64 state only.

FEAT_I8MM is OPTIONAL from Armv8.1.

FEAT_I8MM is mandatory from Armv8.6.

The following fields identify the presence of FEAT_I8MM:

- ID_AA64ISAR1_EL1.I8MM.
- ID_AA64ZFR0_EL1.I8MM.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SIMD integer dot product.
- SIMD integer matrix multiply-accumulate.
- SVE Integer dot product.
- SVE Integer matrix multiply operations.

FEAT_IESB, Implicit Error Synchronization event

FEAT_IESB introduces control for an implicit error synchronization event at exception entry and return.

The implicit error synchronization events affect the same synchronizable asynchronous events that are synchronized by the ESB instruction. This means that if an implementation has either no sources of SError exceptions, or all SError exceptions are not synchronizable errors, then the controls added by this feature have no effect.

This feature is supported in AArch64 state only.

FEAT_IESB is OPTIONAL from Armv8.1.

If FEAT_IESB is implemented, then FEAT_RAS is implemented.

The following field identifies the presence of FEAT_IESB:

- ID_AA64MMFR2_EL1.IESB.

For more information, see Error synchronization event.

FEAT_LPA, Large PA and IPA support

FEAT_LPA:

- Allows a larger *physical address* (PA) and *intermediate physical address* (IPA) space of up to 52 bits when using the 64KB translation granule.
- Allows a level 1 block size where the block covers a 4TB address range for the 64KB translation granule if the implementation support 52 bits of PA.

This feature is supported in AArch64 state only.

FEAT_LPA is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_LPA:

- ID_AA64MMFR0_EL1.PARange.

For more information about FEAT_LPA, see:

- Implemented physical address size.
- Output address size configuration.
- Intermediate physical address size configuration
- Translation using the 64KB granule
- Table descriptor format
- Block descriptor and Page descriptor formats

FEAT_LSMAOC, AArch32 Load/Store Multiple instruction atomicity and ordering controls

FEAT_LSMAOC introduces controls that disable legacy behavior of AArch32 load multiple and store multiple instructions, and provide a trap of one aspect of this legacy behavior.

This feature is supported in both AArch64 and AArch32 states.

FEAT_LSMAOC is OPTIONAL from Armv8.1.

If FEAT_LSMAOC is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_LSMAOC:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR2_EL1.LSM.
- ID_MMFR4_EL1.LSM.
- ID_MMFR4.LSM.

For more information, see the register field descriptions and:

- Generation of Alignment faults by load/store multiple accesses to Device memory.
- Multi-register loads and stores that access Device memory.
- Taking an interrupt or other exception during a multiple-register load or store.

FEAT_LVA, Large VA support

FEAT_LVA supports a *virtual address* (VA) space for each translation table base register of up to 52 bits when using any of the following:

- VMSAv9-128 translation format.
- VMSAv8-64 translation format with the 64KB translation granule.

This feature is supported in AArch64 state only.

FEAT_LVA is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_LVA:

- ID_AA64MMFR2_EL1.VARange.

For more information about FEAT_LVA, see:

- Supported virtual address ranges.
- Input address size configuration.
- Translation using the 64KB granule.

FEAT_PAN2, AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN

FEAT_PAN2 introduces variants of the AArch64 AT S1E1R and AT S1E1W instructions and the AArch32 ATS1CPR and ATS1CPW instructions. These instructions factor in the PSTATE.PAN bit when determining whether or not the location will generate a Permission fault for a privileged access, as is reported in the PAR. For more information, see:

For the AArch64 System instructions:

- AT S1E1RP, Address Translate Stage 1 EL1 Read PAN.
- AT S1E1RP, Address Translate Stage 1 EL1 Write PA.

For the AArch32 System instructions:

- ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN.
- ATS1CPRP, Address Translate Stage 1 Current state PL1 write PAN.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PAN2 is OPTIONAL from Armv8.1.

FEAT_PAN2 is mandatory from Armv8.2.

If FEAT_PAN2 is implemented, then FEAT_PAN is implemented.

The following fields identify the presence of FEAT_PAN2:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Address translation instructions.
- Address translation instructions.
- ATS1C**, Address translation stage 1, current security state.
- Encoding and availability of the address translation instructions.

FEAT_PCSRv8p2, PC Sample-based Profiling Extension

FEAT_PCSRv8p2 moves the control and implementation of the OPTIONAL PC Sample-based Profiling Extension from the ED*SR Debug registers to PM*SR registers in the Performance Monitors address space.

FEAT_PCSRv8p2 is OPTIONAL from Armv8.1.

If FEAT_PCSRv8p2 is implemented, then FEAT_PCSRv8 is not implemented.

The following field identifies the presence of FEAT_PCSRv8p2:

- PMDEVID.PCSample.

For more information, see The PC Sample-based Profiling Extension.

FEAT_RAS, Reliability, Availability and Serviceability (RAS) Extension

FEAT_RAS introduces the following:

- The *Error Synchronization Barrier* (ESB) instruction for the A32, T32, and A64 instruction sets.
- Error records.
- The Error synchronization event.
- Additional syndrome information on External Aborts.

FEAT_RAS is OPTIONAL from Armv8.0.

FEAT_RAS is mandatory from Armv8.2.

The following fields identify the presence of FEAT_RAS:

- ID_AA64PFR0_EL1.RAS.
- ID_PFR0_EL1.RAS.
- ID_PFR0.RAS.

For more information, see:

- Reliability, Availability, and Serviceability.
- RAS PE Architecture.
- *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_SHA3, Advanced SIMD SHA3 instructions

FEAT_SHA3 introduces Advanced SIMD instructions that support SHA3 functionality.

These instructions are added to the A64 instruction set only.

FEAT_SHA3 is OPTIONAL from Armv8.1.

If FEAT_SHA3 is implemented, then FEAT_Crypto is implemented.

If FEAT_SHA3 is implemented, then FEAT_SHA256 and FEAT_SHA1 are implemented.

The following field identifies the presence of FEAT_SHA3:

- ID_AA64ISAR0_EL1.SHA3.

For more information, see FEAT_SHA3, SHA3 functionality.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SHA512, Advanced SIMD SHA512 instructions

FEAT_SHA512 introduces Advanced SIMD instructions that support SHA2-512 functionality.

These instructions are added to the A64 instruction set only.

FEAT_SHA512 is OPTIONAL from Armv8.1.

If FEAT_SHA512 is implemented, then FEAT_Crypto is implemented.

If FEAT_SHA512 is implemented, then FEAT_SHA256 and FEAT_SHA1 are implemented.

The following field identifies the presence of FEAT_SHA512:

- ID_AA64ISAR0_EL1.SHA2.

For more information, see FEAT_SHA512, SHA2-512 functionality.

FEAT_SM3, Advanced SIMD SM3 instructions

FEAT_SM3 introduces Advanced SIMD instructions that support the Chinese cryptography algorithm SM3.

These instructions are added to the A64 instruction set only.

FEAT_SM3 is OPTIONAL from Armv8.1.

If FEAT_SM3 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SM3:

- ID_AA64ISAR0_EL1.SM3.

For more information, see FEAT_SM3, SM3 functionality.

FEAT_SM4, Advanced SIMD SM4 instructions

FEAT_SM4 introduces Advanced SIMD instructions that support the Chinese cryptography algorithm SM4.

These instructions are added to the A64 instruction set only.

FEAT_SM4 is OPTIONAL from Armv8.1.

If FEAT_SM4 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SM4:

- ID_AA64ISAR0_EL1.SM4.

For more information, see FEAT_SM4, SM4 functionality.

FEAT_SPE, Statistical Profiling Extension

FEAT_SPE provides a non-invasive method of sampling software and hardware using randomized sampling of either architectural instructions, as defined by the instruction set architecture, or by microarchitectural operations.

This feature is supported in AArch64 state only.

FEAT_SPE is OPTIONAL from Armv8.1.

When FEAT_SPE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following field identifies the presence of FEAT_SPE:

- ID_AA64DFR0_EL1.PMSVer.

For more information, see The Statistical Profiling Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPE_ArchInst, Statistical Profiling architectural instruction sampling

FEAT_SPE_ArchInst provides sampling of architectural instructions by FEAT_SPE. If FEAT_SPE is implemented and FEAT_SPE_ArchInst is not implemented, then the PE samples microarchitectural operations.

This feature is supported in AArch64 state only.

FEAT_SPE_ArchInst is OPTIONAL.

If FEAT_SPE_ArchInst is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_ArchInst:

- PMSIDR_EL1.ArchInst.

For more information, see The Statistical Profiling Extension.

FEAT_SPE_ERnd, Statistical Profiling random sampling at end of period

FEAT_SPE_ERnd provides a secondary counter, PMSICR_EL1.ECOUNT, that is used after the primary sample counter, PMSICR_EL1.COUNT, reaches zero when PMSIRR_EL1.RND is 1. If FEAT_SPE_ERnd is not implemented, then bits [7:0] of the primary counter are used to provide a random offset.

This feature is supported in AArch64 state only.

FEAT_SPE_ERnd is OPTIONAL.

If FEAT_SPE_ERnd is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_ERnd:

- PMSIDR_EL1.ERnd.

For more information, see The Statistical Profiling Extension.

FEAT_SPE_LDS, Statistical Profiling data source packet generation

FEAT_SPE_LDS indicates that the Statistical Profiling Unit (SPU) can generate Data Source packets.

This feature is supported in AArch64 state only.

FEAT_SPE_LDS is OPTIONAL.

If FEAT_SPE_LDS is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_LDS:

- PMSIDR_EL1.LDS.

FEAT_SVE, Scalable Vector Extension

The Scalable Vector Extension includes the following functionality:

- Configurable vector length with scalable vector lengths from 128 bits up to 2048 bits.
- Predication using scalable predicate registers from 16 bits up to 256 bits.
- Instructions that operate on scalable size vectors and predicates.
- Gather-load and scatter-store.
- Software-managed speculative vectorization.
- System registers and fields to configure the Effective SVE vector length and traps.

The Scalable Vector Extension complements the AArch64 Advanced SIMD and floating-point functionality. SVE does not replace the AArch64 Advanced SIMD and floating-point functionality.

This feature is supported in AArch64 state only.

FEAT_SVE is OPTIONAL from Armv8.2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an Armv8.3 implementation, if FEAT_SVE is implemented, FEAT_FCMA and FEAT_FP16 are implemented.

When FEAT_SVE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following field identifies the presence of FEAT_SVE:

- ID_AA64PFR0_EL1.SVE.

FEAT_TTCNP, Translation table Common not private translations

FEAT_TTCNP permits multiple PEs in the same Inner Shareable domain to use the same translation tables for a given stage of address translation.

This facility is available for all VMSAv8-64 translation regimes and for VMSAv8-32 translation stages that use the Long descriptor Translation Table format.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TTCNP is OPTIONAL from Armv8.1.

FEAT_TTCNP is mandatory from Armv8.2.

The following fields identify the presence of FEAT_TTCNP:

- ID_AA64MMFR2_EL1.CnP.
- ID_MMFR4_EL1.CnP.
- ID_MMFR4.CnP.

For more information, see:

- Common not private translations.
- Common not private translations in VMSAv8-32.

FEAT_UAO, Unprivileged Access Override control

FEAT_UAO introduces the UAO bit to PSTATE. When the Effective value of UAO is 1, if memory accesses that are described as unprivileged are made by the unprivileged memory access instructions, they are privileged.

This feature is supported in AArch64 state only.

FEAT_UAO is OPTIONAL from Armv8.1.

FEAT_UAO is mandatory from Armv8.2.

The following field identifies the presence of FEAT_UAO:

- ID_AA64MMFR2_EL1.UAO.

For more information, see:

- PSTATE.UAO.
- Load/store unprivileged.
- UAO.

FEAT_XNX, Translation table stage 2 Unprivileged Execute-never

FEAT_XNX extends the stage 2 translation table access permissions to provide control of whether memory is executable at EL0 independent of whether it is executable at EL1.

This facility is available for stage 2 translation stages in VMSAv8-64 and VMSAv8-32.

This feature is supported in both AArch64 and AArch32 states.

FEAT_XNX is OPTIONAL from Armv8.1.

In an Armv8.2 implementation, if FEAT_EL2 is implemented, FEAT_XNX is implemented.

The following fields identify the presence of FEAT_XNX:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR1_EL1.XNX.
- ID_MMFR4_EL1.XNX.
- ID_MMFR4.XNX.

For more information, see:

- Stage 2 instruction execution using Direct permissions.
- Access permissions for instruction execution.

A2.2.3.1 Features added to the Armv8.2 extension in later releases

- FEAT_AA32BF16.
- FEAT_BF16.
- FEAT_EBF16.
- FEAT_EVT2.
- FEAT_EVT.
- FEAT_LRCPC2.
- FEAT_LRCPC3.
- FEAT_LSE2.
- FEAT_MPAM.
- FEAT_MPAMv1p0.
- FEAT_RASv1p1.

A2.2.4 The Armv8.3 architecture extension

The Armv8.3 architecture extension is an extension to Armv8.2. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.3 compliant if all of the following apply:

- It is Armv8.2 compliant.
- It includes all of the Armv8.3 architectural features that are mandatory.

An Armv8.3 compliant implementation can additionally include:

- Armv8.3 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.4, subject only to those constraints that require that certain features be implemented together.

FEAT_CCIDX, Extended cache index

FEAT_CCIDX introduces the following registers to allow caches to be described with greater numbers of sets and greater associativity:

- A 64-bit format of CCSIDR_EL1.
- CCSIDR2_EL1.
- CCSIDR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in both AArch64 and AArch32 states.

FEAT_CCIDX is OPTIONAL from Armv8.2.

The following fields identify the presence of FEAT_CCIDX:

- ID_AA64MMFR2_EL1.CCIDX.
- ID_MMFR4_EL1.CCIDX.
- ID_MMFR4.CCIDX.

For more information, see:

- Possible formats of the Cache Size Identification Register, CCSIDR_EL1.
- Possible formats of the Cache Size Identification Register, CCSIDR and CCSIDR2.

FEAT_CONSTPACFIELD, PAC algorithm enhancement

FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field when adding a PAC to the virtual address, even when the top byte is not being ignored.

This feature is supported in AArch64 state only.

FEAT_CONSTPACFIELD is OPTIONAL from Armv8.2.

If FEAT_CONSTPACFIELD is implemented, then FEAT_PAuth2 is implemented.

The following field identifies the presence of FEAT_CONSTPACFIELD:

- ID_AA64ISAR2_EL1.PAC_frac.

For more information, see PAC field.

FEAT_DoPD, Debug over Powerdown

FEAT_DoPD provides a debug programmers' model where all debug and PMU registers are in the Core power domain and all CTI registers are in the Debug power domain. Power control is provided by an OPTIONAL external powerup request mechanism.

When FEAT_DoPD is implemented:

- The optional Software Lock is not implemented by the architecturally defined debug components in the PE Core power domain.
- If an ETMv4 trace unit is implemented, the ETM must implement:
  - ETMv4.2 or later.
  - The Unified Power Domain Model.
- If FEAT_ETE is implemented, the trace unit always implements a single power domain.

FEAT_DoPD is OPTIONAL from Armv8.2.

If FEAT_DoPD is implemented, then FEAT_DoubleLock is not implemented.

If FEAT_DoPD is implemented, then FEAT_Debugv8p2 is implemented.

The following field identifies the presence of FEAT_DoPD:

- EDDEVID.DebugPower.

For more information, see Debug Reset and Powerdown Support.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_EPAC, Enhanced pointer authentication

FEAT_EPAC introduces functionality that permits setting the *Pointer Authentication Code* (PAC) field to 0 on performing a PAC operation on a non-canonical address.

Arm deprecates implementation of this feature.

This feature is supported in AArch64 state only.

FEAT_EPAC is OPTIONAL from Armv8.2.

If FEAT_EPAC is implemented, then FEAT_PAuth is implemented.

If FEAT_EPAC is implemented, then FEAT_PAuth2 is not implemented.

The following fields identify the presence of FEAT_EPAC:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_FCMA, Floating-point complex number instructions

FEAT_FCMA introduces instructions for floating-point multiplication and addition of complex numbers.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_FCMA is OPTIONAL from Armv8.2.

In an Armv8.3 implementation, if FEAT_FP is implemented, FEAT_FCMA is implemented.

If FEAT_FCMA is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_FCMA:

- ID_AA64ISAR1_EL1.FCMA.
- ID_ISAR5_EL1.VCMA.
- ID_ISAR5.VCMA.

For more information, see:

- AArch64 Advanced SIMD complex number arithmetic instructions.
- AArch32 Advanced SIMD complex number arithmetic instructions.

FEAT_FPAC, Faulting on AUT* instructions

FEAT_FPAC introduces faulting on an AUT* instruction.

FEAT_FPAC is added as a further extension to FEAT_PAuth2.

This feature is supported in AArch64 state only.

FEAT_FPAC is OPTIONAL from Armv8.2.

If FEAT_FPAC is implemented, then FEAT_PAuth2 is implemented.

The following fields identify the presence of FEAT_FPAC:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FPACCOMBINE, Faulting on combined pointer authentication instructions

FEAT_FPACCOMBINE introduces faulting on the combined instructions that perform pointer authentication.

FEAT_FPACCOMBINE is added as a further extension to FEAT_FPAC.

FEAT_FPACCOMBINE is OPTIONAL from Armv8.2.

If FEAT_FPACCOMBINE is implemented, then FEAT_FPAC is implemented.

The following fields identify the presence of FEAT_FPACCOMBINE:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication.

FEAT_FPACC_SPEC, Speculative behavior of pointer authentication instructions

FEAT_FPACC_SPEC introduces consistent impact of speculation for instructions that perform authentication.

FEAT_FPACC_SPEC is added as a further extension to FEAT_FPACCOMBINE.

This feature is supported in AArch64 state only.

FEAT_FPACC_SPEC is OPTIONAL from Armv8.2.

If FEAT_FPACC_SPEC is implemented, then FEAT_FPACCOMBINE is implemented.

The following field identifies the presence of FEAT_FPACC_SPEC:

- ID_AA64MMFR3_EL1.Spec_FPACC.

For more information, see Faulting on pointer authentication.

FEAT_JSCVT, JavaScript conversion instructions

FEAT_JSCVT introduces JavaScript convert instructions that truncate a double-precision value to a 32-bit signed integer, setting the condition flags to indicate whether the converted value was in range.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_JSCVT is OPTIONAL from Armv8.2.

In an Armv8.3 implementation, if FEAT_FP is implemented, FEAT_JSCVT is implemented.

If FEAT_JSCVT is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_JSCVT:

- ID_AA64ISAR1_EL1.JSCVT.
- ID_ISAR6_EL1.JSCVT.
- ID_ISAR6.JSCVT.

For more information, see:

- Floating-point conversion.
- About the A64 Advanced SIMD and floating-point instructions.
- Advanced SIMD and floating-point instructions.
- Floating-point data-processing instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LRCPC, Load-Acquire RCpc instructions

FEAT_LRCPC introduces instructions that support the weaker *Release Consistency processor consistent* (RCpc) model that enables the reordering of a Store-Release followed by a Load-Acquire to a different address.

These instructions are added to the A64 instruction set only.

FEAT_LRCPC is OPTIONAL from Armv8.2.

FEAT_LRCPC is mandatory from Armv8.3.

The following field identifies the presence of FEAT_LRCPC:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Load-Acquire, Load-AcquirePC, and Store-Release.
- Load-Acquire/Store-Release.

FEAT_NV, Nested Virtualization

FEAT_NV provides support for a Guest Hypervisor to run in EL1 and ensures that the Guest Hypervisor is unaware that it is running at that Exception level. A Guest Hypervisor is supported regardless of the Effective value of HCR_EL2.E2H.

This feature is supported in AArch64 state only.

FEAT_NV is OPTIONAL from Armv8.2.

If FEAT_NV is implemented, then FEAT_EL2 is implemented.

The following fields identify the presence of FEAT_NV:

- ID_AA64MMFR4_EL1.NV_frac.
- ID_AA64MMFR2_EL1.NV.

For more information, see Nested virtualization.

FEAT_PACIMP, Pointer authentication - IMPLEMENTATION DEFINED algorithm

FEAT_PACIMP permits an IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

This feature is supported in AArch64 state only.

FEAT_PACIMP is OPTIONAL from Armv8.2.

If FEAT_PACIMP is implemented, then FEAT_PAuth is implemented.

If FEAT_PACIMP is implemented, then FEAT_PACQARMA5 is not implemented.

If FEAT_PACIMP is implemented, then FEAT_PACQARMA3 is not implemented.

The following fields identify the presence of FEAT_PACIMP:

- ID_AA64ISAR1_EL1.GPI.
- ID_AA64ISAR1_EL1.API.

For more information, see Pointer authentication.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PACQARMA3, Pointer authentication - QARMA3 algorithm

FEAT_PACQARMA3 introduces the QARMA3 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

FEAT_PACQARMA3 is OPTIONAL from Armv8.2.

If FEAT_PACQARMA3 is implemented, then FEAT_PAuth is implemented.

If FEAT_PACQARMA3 is implemented, then FEAT_PACQARMA5 is not implemented.

If FEAT_PACQARMA3 is implemented, then FEAT_PACIMP is not implemented.

The following fields identify the presence of FEAT_PACQARMA3:

- ID_AA64ISAR3_EL1.PACM.
- ID_AA64ISAR2_EL1.GPA3.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_PACQARMA5, Pointer authentication - QARMA5 algorithm

FEAT_PACQARMA5 introduces the QARMA5 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

FEAT_PACQARMA5 is OPTIONAL from Armv8.2.

If FEAT_PACQARMA5 is implemented, then FEAT_PAuth is implemented.

If FEAT_PACQARMA5 is implemented, then FEAT_PACQARMA3 is not implemented.

If FEAT_PACQARMA5 is implemented, then FEAT_PACIMP is not implemented.

The following fields identify the presence of FEAT_PACQARMA5:

- ID_AA64ISAR3_EL1.PACM.
- ID_AA64ISAR1_EL1.GPA.
- ID_AA64ISAR1_EL1.APA.

For more information, see Pointer authentication.

FEAT_PAuth, Pointer authentication

FEAT_PAuth introduces functionality that supports address authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load.

When FEAT_PAuth is implemented, all of the following must be true:

- Exactly one of the PAC algorithms is implemented.
- The PACGA instruction and other Pointer authentication instructions use the same algorithm.

The PAC algorithm features are:

- FEAT_PACQARMA5.
- FEAT_PACIMP.
- FEAT_PACQARMA3.

This feature is supported in AArch64 state only.

FEAT_PAuth is OPTIONAL from Armv8.2.

FEAT_PAuth is mandatory from Armv8.3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_PAuth is implemented, then FEAT_PACQARMA5, FEAT_PACIMP, or FEAT_PACQARMA3 is implemented.

For more information, see Pointer authentication.

FEAT_PAuth2, Enhancements to pointer authentication

FEAT_PAuth2 introduces enhanced pointer authentication functionality that changes the mechanism by which a PAC is added to the pointer.

This feature is supported in AArch64 state only.

FEAT_PAuth2 is OPTIONAL from Armv8.2.

FEAT_PAuth2 is mandatory from Armv8.6.

If FEAT_PAuth2 is implemented, then FEAT_PAuth is implemented.

If FEAT_PAuth2 is implemented, then FEAT_EPAC is not implemented.

The following fields identify the presence of FEAT_PAuth2:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_SPEv1p1, Statistical Profiling Extension version 1.1

FEAT_SPEv1p1 introduces the following to the Statistical Profiling Extension:

- Support for an Alignment Flag in the Events packet and filtering on this event using PMSEVFR_EL1.
- Support for the profiling of Scalable Vector Extension operations.

This feature is supported in AArch64 state only.

FEAT_SPEv1p1 is OPTIONAL from Armv8.2.

In an Armv8.5 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p1 is implemented.

If FEAT_SPEv1p1 is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPEv1p1:

- ID_AA64DFR0_EL1.PMSVer.

An implementation that includes FEAT_SVE and the Statistical Profiling Extension is strongly recommended to implement FEAT_SPEv1p1 whenever possible.

For more information, see:

- The Statistical Profiling Extension.
- Statistical Profiling Extension Sample Record Specification.

A2.2.5 The Armv8.4 architecture extension

The Armv8.4 architecture extension is an extension to Armv8.3. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.4 compliant if all of the following apply:

- It is Armv8.3 compliant.
- It includes all of the Armv8.4 architectural features that are mandatory.

An Armv8.4 compliant implementation can additionally include:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Armv8.4 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.5, subject only to those constraints that require that certain features be implemented together.

FEAT_AMU_EXT, External Activity Monitors

FEAT_AMU_EXT indicates support for external access to the Activity Monitors.

FEAT_AMU_EXT is OPTIONAL.

If FEAT_AMU_EXT is implemented, then FEAT_AMU_EXT32 or FEAT_AMU_EXT64 is implemented.

If FEAT_AMU_EXT is implemented, then FEAT_AMUv1 is implemented.

FEAT_AMU_EXT32, 32-bit External Activity Monitors

FEAT_AMU_EXT32 indicates the external AMU registers are implemented as mostly 32-bit registers.

FEAT_AMU_EXT32 is OPTIONAL.

If FEAT_AMU_EXT32 is implemented, then FEAT_AMU_EXT is implemented.

If FEAT_AMU_EXT32 is implemented, then FEAT_AMU_EXT64 is not implemented.

The following field identifies the presence of FEAT_AMU_EXT32:

- AMDEVARCH.ARCHID.

For more information, see Recommended External Interface to the Activity Monitors.

FEAT_AMUv1, Activity Monitors Extension version 1

FEAT_AMUv1 provides a function similar to a subset of the existing Performance Monitors Extension functionality, intended for system management use rather than debugging and profiling.

This feature is supported in both AArch64 and AArch32 states.

FEAT_AMUv1 is OPTIONAL from Armv8.3.

The following fields identify the presence of FEAT_AMUv1:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see The Activity Monitors Extension.

FEAT_BBML1, Translation table break-before-make level 1

FEAT_BBML1 provides support for reducing the requirements for following a break-before-make sequence when changing between table or block sizes for a translation.

This feature is supported in AArch64 state only.

FEAT_BBML1 is OPTIONAL from Armv8.3.

The following field identifies the presence of FEAT_BBML1:

- ID_AA64MMFR2_EL1.BBM.

For more information, see:

- Block descriptor and Page descriptor formats.
- Block translation entry.
- Support levels for changing block size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_BBML2, Translation table break-before-make level 2

FEAT_BBML2 provides support for further reducing the requirements for following a break-before-make sequences when changing between table or block sizes for a translation.

This feature is supported in AArch64 state only.

FEAT_BBML2 is OPTIONAL from Armv8.3.

If FEAT_BBML2 is implemented, then FEAT_BBML1 is implemented.

The following field identifies the presence of FEAT_BBML2:

- ID_AA64MMFR2_EL1.BBM.

For more information, see FEAT_BBML1.

FEAT_CNTSC, Generic Counter Scaling

FEAT_CNTSC introduces a scaling register to the memory-mapped counter module that allows the frequency of the counter that is generated to be scaled from the basic frequency reported in the counter ID mechanisms.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CNTSC is OPTIONAL from Armv8.3.

The following field identifies the presence of FEAT_CNTSC:

- CNTID.CNTSC.

For more information, see CNTCR.

FEAT_DIT, Data Independent Timing instructions

FEAT_DIT provides a mechanism for software to indicate that a sequence of code is intended to be independent from data values from certain instructions within the sequence.

This feature is supported in both AArch64 and AArch32 states.

FEAT_DIT is OPTIONAL from Armv8.3.

FEAT_DIT is mandatory from Armv8.4.

The following fields identify the presence of FEAT_DIT:

- ID_AA64PFR0_EL1.DIT.
- ID_PFR0_EL1.DIT.
- ID_PFR0.DIT.

For more information, see:

- About PSTATE.DIT.
- About the DIT bit.

FEAT_Debugv8p4, Debug v8.4

FEAT_Debugv8p4 provides the following mandatory changes:

- The fields MDCR_EL3.{EPMAD, EDAD} control Non-secure access to the debug and PMU registers. The bus Requester is responsible for other debug authentication.
- The Software Lock is obsolete.
- Non-invasive Debug controls are relaxed.
- Secure and Non-secure views of the debug registers are enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_Debugv8p4 is OPTIONAL from Armv8.3.

FEAT_Debugv8p4 is mandatory from Armv8.4.

If FEAT_Debugv8p4 is implemented, then FEAT_Debugv8p2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_Debugv8p4 is implemented.

The following fields identify the presence of FEAT_Debugv8p4:

- ID_DFR0_EL1.CopDbg.
- ID_AA64DFR0_EL1.DebugVer.
- DBGDIDR.Version.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

For more information, see:

- Definition and constraints of a debugger in the context of external debug.
- External debug interface register access permissions.

FEAT_DotProd, Advanced SIMD dot product instructions

FEAT_DotProd provides instructions to perform a four-way vector dot product of 8-bit integers, accumulating each sum of four products into a 32-bit integer. Each 8-bit input can be treated as a signed or unsigned value.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_DotProd is OPTIONAL from Armv8.1.

In an Armv8.4 implementation, if FEAT_AdvSIMD is implemented, FEAT_DotProd is implemented.

The following fields identify the presence of FEAT_DotProd:

- ID_AA64ISAR0_EL1.DP.
- ID_ISAR6_EL1.DP.
- ID_ISAR6.DP.

For more information, see:

- SIMD integer dot product.
- Advanced SIMD dot product instructions.

FEAT_DoubleFault, Double Fault Extension

FEAT_DoubleFault provides controls for routing and masking error exceptions.

This feature is supported in AArch64 state only.

FEAT_DoubleFault is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_AA64EL3 is implemented, FEAT_DoubleFault is implemented.

If FEAT_DoubleFault is implemented, then FEAT_DoubleFault2 or FEAT_AA64EL3 is implemented.

The following field identifies the presence of FEAT_DoubleFault:

- ID_AA64PFR0_EL1.RAS.

For more information, see:

- Taking error exceptions.
- Error synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FHM, Floating-point half-precision to single-precision multiply-add instructions

FEAT_FHM introduces half-precision to single-precision fused multiply-add instructions.

These instructions are added to the A64 and A32/T32 instruction sets.

FEAT_FHM is OPTIONAL from Armv8.1.

In an Armv8.4 implementation, if FEAT_FP16 is implemented, FEAT_FHM is implemented.

In an Armv8.2 implementation, if FEAT_FHM is implemented, FEAT_FP16 is implemented.

The following fields identify the presence of FEAT_FHM:

- ID_AA64ISAR0_EL1.FHM.
- ID_ISAR6_EL1.FHM.
- ID_ISAR6.FHM.

For more information, see:

- SIMD arithmetic.
- SIMD by element arithmetic.
- Advanced SIMD multiply instructions.

FEAT_FlagM, Condition flag manipulation instructions

FEAT_FlagM provides instructions that manipulate the PSTATE.{N,Z,C,V} flags.

These instructions are added to the A64 instruction set only.

FEAT_FlagM is OPTIONAL from Armv8.1.

FEAT_FlagM is mandatory from Armv8.4.

The following field identifies the presence of FEAT_FlagM:

- ID_AA64ISAR0_EL1.TS.

For more information, see Flag manipulation instructions.

FEAT_IDST, ID space trap handling

FEAT_IDST causes all AArch64 read accesses to the feature ID space when exceptions are generated to be reported in ESR_ELx using the EC code `0x18`.

This feature is supported in AArch64 state only.

FEAT_IDST is OPTIONAL from Armv8.3.

FEAT_IDST is mandatory from Armv8.4.

The following field identifies the presence of FEAT_IDST:

- ID_AA64MMFR2_EL1.IDS.

FEAT_LRCPC2, Load-Acquire RCpc instructions version 2

FEAT_LRCPC2 provides versions of `LDAPR` and `STLR` with a 9-bit unscaled signed immediate offset.

These instructions are added to the A64 instruction set only.

FEAT_LRCPC2 is OPTIONAL from Armv8.2.

FEAT_LRCPC2 is mandatory from Armv8.4.

If FEAT_LRCPC2 is implemented, then FEAT_LRCPC is implemented.

The following field identifies the presence of FEAT_LRCPC2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Changes to single-copy atomicity in Armv8.4.
- Load-Acquire/Store-Release.
- A64 instructions that are changed in Debug state.

FEAT_LSE2, Large System Extensions version 2

FEAT_LSE2 introduces changes to single-copy atomicity requirements for loads and stores, and changes to alignment requirements for loads and stores.

This feature is supported in AArch64 state only.

FEAT_LSE2 is OPTIONAL from Armv8.2.

FEAT_LSE2 is mandatory from Armv8.4.

The following field identifies the presence of FEAT_LSE2:

- ID_AA64MMFR2_EL1.AT.

For more information, see:

- Requirements for single-copy atomicity.
- Alignment of data accesses.

FEAT_MPAM, Memory System Resource Partitioning and Monitoring Extension

The MPAM Extension provides a framework for memory-system component controls that partition one or more of the performance resources of the component. There are three versions of the MPAM extension: v1p0, v1p1, and v0p1.

This feature is supported in AArch64 state only.

FEAT_MPAM is OPTIONAL from Armv8.2.

If FEAT_MPAM is implemented, then FEAT_MPAMv1p0 or FEAT_MPAMv0p1 is implemented.

For more information, see MPAM PE Architecture.

FEAT_MPAMv1p0, Memory Partitioning and Monitoring Extension version 1.0

FEAT_MPAMv1p0 introduces support for version 1.0 of the MPAM extension.

This feature is supported in AArch64 state only.

FEAT_MPAMv1p0 is OPTIONAL from Armv8.2.

If FEAT_MPAMv1p0 is implemented, then FEAT_MPAM is implemented.

If FEAT_MPAMv1p0 is implemented, then FEAT_MPAMv0p1 is not implemented.

The following field identifies the presence of FEAT_MPAMv1p0:

- ID_AA64PFR0_EL1.MPAM.

For more information, see MPAM PE Architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_NV2, Enhanced nested virtualization support

FEAT_NV2 supports nested virtualization by redirecting register accesses that would be trapped to EL1 and EL2 to access memory instead. The address of the memory access depends on information held in VNCR_EL2.

This feature is supported in AArch64 state only.

FEAT_NV2 is OPTIONAL from Armv8.3.

If FEAT_NV2 is implemented, then FEAT_NV is implemented.

The following fields identify the presence of FEAT_NV2:

- ID_AA64MMFR4_EL1.NV_frac.
- ID_AA64MMFR2_EL1.NV.

For more information, see Enhanced support for nested virtualization.

FEAT_PMUv3p4, Armv8.4 PMU extensions

FEAT_PMUv3p4 introduces:

- PMMIR_EL1
- PMMIR registers.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p4 is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p4 is implemented.

If FEAT_PMUv3p4 is implemented, then FEAT_PMUv3p1 is implemented.

The following fields identify the presence of FEAT_PMUv3p4:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see PMU Event Descriptions.

FEAT_RASv1p1, RAS extension v1.1

FEAT_RASv1p1 introduces support for System register access to the following RAS System Architecture v1.1 features:

- Additional ERR<n>MISC<m> registers.
- The optional RAS Common Fault Injection Model Extension.

This feature is supported in both AArch64 and AArch32 states.

FEAT_RASv1p1 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_RAS is implemented, FEAT_RASv1p1 is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RAS is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RASSAv1p1 is implemented.

The following fields identify the presence of FEAT_RASv1p1:

- ID_AA64PFR0_EL1.RAS.
- ID_AA64PFR1_EL1.RAS_frac.
- ID_PFR0_EL1.RAS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_PFR2_EL1.RAS_frac.
- ID_PFR0.RAS.
- ID_PFR2.RAS_frac.

For more information, see *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_S2FWB, Stage 2 forced Write-Back

FEAT_S2FWB reduces the requirement of additional cache maintenance instructions in systems where the data Cacheability attributes used by the Guest operating system are different from those expected by the Hypervisor. If this feature is implemented, there is no meaningful distinction between the Inner and Outer Shareability domains for accesses to Normal Cacheable memory.

This feature is supported in AArch64 state only.

FEAT_S2FWB is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_AA64EL2 is implemented, FEAT_S2FWB is implemented.

If FEAT_S2FWB is implemented, then FEAT_AA64EL2 is implemented.

The following fields identify the presence of FEAT_S2FWB:

- ID_AA64MMFR2_EL1.FWB.
- CLIDR_EL1.LoUU.
- CLIDR_EL1.LoUIS.

For more information, see:

- Block descriptor and Page descriptor formats.
- Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled.

FEAT_SEL2, Secure EL2

FEAT_SEL2 permits EL2 to be implemented in Secure state. When Secure EL2 is enabled, a translation regime is introduced that follows the same format as the other Secure translation regimes.

This feature is not supported if EL2 is using AArch32.

FEAT_SEL2 is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_AA64EL2 and FEAT_Secure are implemented, FEAT_SEL2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_TTST is implemented.

If FEAT_SEL2 is implemented, then FEAT_PCSRv8 is not implemented.

If FEAT_SEL2 is implemented, then FEAT_EL2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_Secure is implemented.

The following field identifies the presence of FEAT_SEL2:

- ID_AA64PFR0_EL1.SEL2.

For more information, see:

- Security states
- Translation regimes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TLBIOS, TLB invalidate instructions in Outer Shareable domain

FEAT_TLBIOS provides TLBI maintenance instructions that extend to the Outer Shareable domain.

This feature is supported in AArch64 state only.

FEAT_TLBIOS is OPTIONAL from Armv8.3.

FEAT_TLBIOS is mandatory from Armv8.4.

The following field identifies the presence of FEAT_TLBIOS:

- ID_AA64ISAR0_EL1.TLB.

For more information, see TLB maintenance instructions.

FEAT_TLBIRANGE, TLB invalidate range instructions

FEAT_TLBIRANGE provides TLBI maintenance instructions that apply to a range of input addresses.

This feature is supported in AArch64 state only.

FEAT_TLBIRANGE is OPTIONAL from Armv8.3.

FEAT_TLBIRANGE is mandatory from Armv8.4.

If FEAT_TLBIRANGE is implemented, then FEAT_TLBIOS is implemented.

The following field identifies the presence of FEAT_TLBIRANGE:

- ID_AA64ISAR0_EL1.TLB.

For more information, see:

- TLB maintenance instructions.
- TLB maintenance instructions that do not apply to a range of addresses.

FEAT_TRF, Self-hosted Trace extensions

FEAT_TRF introduces control of trace in a self-hosted system through System registers.

The feature provides:

- Control of Exception levels and Security states where trace generation is prohibited.
- Control of whether an offset is used for the timestamp recorded with trace information.
- A context synchronization instruction, TSB CSYNC, that can be used to prevent reordering of trace operation accesses with respect to other accesses of the same System registers.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TRF is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_ETMv4 is implemented, FEAT_TRF is implemented.

If FEAT_TRF is implemented, then FEAT_TRC_SR is implemented.

The following fields identify the presence of FEAT_TRF:

- ID_DFR0_EL1.TraceFilt.
- ID_AA64DFR0_EL1.TraceFilt.
- ID_DFR0.TraceFilt.
- EDDFR.TraceFilt.

For more information on FEAT_TRF, see:

- AArch64 Self-hosted Trace.
- AArch32 Self-hosted Trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TTL, Translation Table Level

FEAT_TTL provides the TTL field to indicate the level of translation table walk holding the leaf entry for the address that is being invalidated. This field is provided in all TLB maintenance instructions that take a VA or an IPA argument.

This feature is supported in AArch64 state only.

FEAT_TTL is OPTIONAL from Armv8.3.

FEAT_TTL is mandatory from Armv8.4.

The following field identifies the presence of FEAT_TTL:

- ID_AA64MMFR2_EL1.TTL.

For more information, see:

- TLB maintenance instructions.
- TLB maintenance instructions that do not apply to a range of addresses.

FEAT_TTST, Small translation tables

FEAT_TTST relaxes the lower limit on the size of translation tables, by increasing the maximum permitted value of the T1SZ and T0SZ fields in TCR_EL1, TCR_EL2, TCR_EL3, VTCR_EL2 and VSTCR_EL2.

This feature is supported in AArch64 state only.

FEAT_TTST is OPTIONAL from Armv8.3.

If FEAT_SEL2 is implemented, then FEAT_TTST is implemented.

The following field identifies the presence of FEAT_TTST:

- ID_AA64MMFR2_EL1.ST.

For more information, see:

- Input address size configuration.
- Translation using the 4KB granule.
- Translation using the 16KB granule.
- Translation using the 64KB granule.

A2.2.5.1 Features added to the Armv8.4 extension in later releases

- FEAT_AMU_EXTACR.
- FEAT_NV2p1.

A2.2.6 The Armv8.5 architecture extension

The Armv8.5 architecture extension is an extension to Armv8.4. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.5 compliant if all of the following apply:

- It is Armv8.4 compliant.
- It includes all of the Armv8.5 architectural features that are mandatory.

An Armv8.5 compliant implementation can additionally include:

- Armv8.5 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.6, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_BTI, Branch Target Identification

FEAT_BTI allows memory pages to be guarded against the execution of instructions that are not the intended target of a branch. To do this, it introduces:

- The GP field, which denotes the blocks and pages in stage 1 translation tables that are guarded pages.
- The PSTATE.BTYPE field, which is used to determine whether an access to a guarded memory region will generate a Branch Target exception.
- The BTI instruction, which is used to guard against the execution of instructions that are not the intended target of a branch.

This feature is supported in AArch64 state only.

FEAT_BTI is OPTIONAL from Armv8.4.

FEAT_BTI is mandatory from Armv8.5.

The following field identifies the presence of FEAT_BTI:

- ID_AA64PFR1_EL1.BT.

For more information, see:

- Table descriptor format.
- PSTATE.BTYPE.
- Effect of entering Debug state on PSTATE.

FEAT_CSV3, Cache Speculation Variant 3

FEAT_CSV3 introduces a mechanism to identify whether data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, can be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CSV3 is OPTIONAL from Armv8.0.

FEAT_CSV3 is mandatory from Armv8.5.

The following field identifies the presence of FEAT_CSV3:

- ID_AA64PFR0_EL1.CSV3.

FEAT_DPB2, DC CVADP instruction

FEAT_DPB2 allows two levels of cache clean to the Point of Persistence by:

- Redefining Point of Persistence, which changes the scope of DC CVAP.
- Defining a Point of Deep Persistence.
- Adding the DC CVADP System instruction.

This feature is supported in AArch64 state only.

FEAT_DPB2 is OPTIONAL from Armv8.1.

FEAT_DPB2 is mandatory from Armv8.5.

If FEAT_DPB2 is implemented, then FEAT_DPB is implemented.

The following field identifies the presence of FEAT_DPB2:

- ID_AA64ISAR1_EL1.DPB.

For more information, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_E0PD, Preventing EL0 access to halves of address maps

FEAT_E0PD prevents access at EL0 to half of the addresses in the memory map.

This feature is supported in AArch64 state only. When EL1 is using AArch64 state, this feature affects access to EL0, in either Execution state.

FEAT_E0PD is OPTIONAL from Armv8.4.

FEAT_E0PD is mandatory from Armv8.5.

If FEAT_E0PD is implemented, then FEAT_CSV3 is implemented.

The following field identifies the presence of FEAT_E0PD:

- ID_AA64MMFR2_EL1.E0PD.

For more information, see Preventing EL0 access to halves of the address map.

FEAT_EVT, Enhanced Virtualization Traps

FEAT_EVT introduces additional traps for EL1 and EL0 Cache controls in:

- HCR_EL2
- HCR2.

These traps are independent of existing controls.

This feature is supported in both AArch64 and AArch32 states.

FEAT_EVT is OPTIONAL from Armv8.2.

In an Armv8.5 implementation, if FEAT_EL2 is implemented, FEAT_EVT is implemented.

The following fields identify the presence of FEAT_EVT:

- ID_AA64MMFR2_EL1.EVT.
- ID_MMFR4_EL1.EVT.
- ID_MMFR4.EVT.

FEAT_EVT2, Enhanced Virtualization Traps 2

FEAT_EVT2 extends FEAT_EVT to introduce additional traps for further EL1 and EL0 system control operations in:

- HCR_EL2.
- HCR2.

This feature is supported in both AArch64 and AArch32 states.

FEAT_EVT2 is OPTIONAL from Armv8.2.

In an Armv8.5 implementation, if FEAT_EL2 is implemented, FEAT_EVT2 is implemented.

If FEAT_EVT2 is implemented, then FEAT_EVT is implemented.

The following fields identify the presence of FEAT_EVT2:

- ID_AA64MMFR2_EL1.EVT.
- ID_MMFR4_EL1.EVT.
- ID_MMFR4.EVT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ExS, Context synchronization and exception handling

FEAT_ExS provides a mechanism to control whether exception entry and exception return are Context Synchronization events.

Fields in the SCTLR_ELx registers enable and disable context synchronization at exception entry and return at an Exception level.

This feature is supported in AArch64 state only.

FEAT_ExS is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_ExS:

- ID_AA64MMFR0_EL1.ExS.

For more information, see Context Synchronization event.

FEAT_FRINTTS, Floating-point to integer instructions

FEAT_FRINTTS provides instructions that round a floating-point number to an integral valued floating-point number that fits in a 32-bit or 64-bit integer number range.

These instructions are added to the A64 instruction set only.

FEAT_FRINTTS is OPTIONAL from Armv8.4.

In an Armv8.5 implementation, if FEAT_FP is implemented, FEAT_FRINTTS is implemented.

If FEAT_FRINTTS is implemented, then FEAT_FP and FEAT_AdvSIMD are implemented.

The following field identifies the presence of FEAT_FRINTTS:

- ID_AA64ISAR1_EL1.FRINTTS.

For more information, see Floating-point round to integral value.

FEAT_FlagM2, Enhancements to flag manipulation instructions

FEAT_FlagM2 provides instructions that convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format described in Condition flags and related instructions.

These instructions are added to the A64 instruction set only.

FEAT_FlagM2 is OPTIONAL from Armv8.4.

If FEAT_FlagM2 is implemented, then FEAT_FlagM is implemented.

The following field identifies the presence of FEAT_FlagM2:

- ID_AA64ISAR0_EL1.TS.

For more information, see Flag manipulation instructions.

FEAT_GTG, Guest translation granule size

FEAT_GTG allows a hypervisor to support different granule sizes for stage 2 and stage 1 translation, and allows a nested hypervisor to determine what stage 2 granule sizes are available.

This feature is supported in AArch64 state only.

FEAT_GTG is OPTIONAL from Armv8.0.

In an Armv8.5 implementation, if FEAT_AA64EL2 is implemented, FEAT_GTG is implemented.

The following fields identify the presence of FEAT_GTG:

- ID_AA64MMFR0_EL1.TGran4_2.
- ID_AA64MMFR0_EL1.TGran16_2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR0_EL1.TGran64_2.

For more information, see Translation granules.

FEAT_MTE, Memory Tagging Extension

FEAT_MTE provides architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

This feature is supported in AArch64 state only.

FEAT_MTE is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_MTE:

- ID_AA64PFR1_EL1.MTE.

For more information, see:

- The Memory Tagging Extension.
- The AArch64 Application Level Memory Model.
- PMU Event Descriptions.
- The Statistical Profiling Extension.
- Debug State.

FEAT_MTE2, Memory Tagging Extension

FEAT_MTE2 provides architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

This feature is supported in AArch64 state only.

FEAT_MTE2 is OPTIONAL from Armv8.4.

If FEAT_MTE2 is implemented, then FEAT_MTE is implemented.

The following field identifies the presence of FEAT_MTE2:

- ID_AA64PFR1_EL1.MTE.

For more information, see:

- The Memory Tagging Extension.
- The AArch64 Application Level Memory Model.
- PMU Event Descriptions.
- The Statistical Profiling Extension.
- Debug State.

FEAT_PMUv3p5, Armv8.5 PMU extensions

FEAT_PMUv3p5 extends event counters to 64-bit event counters, and introduces mechanisms to disable the cycle counter in Secure state, in EL3, and in EL2.

FEAT_PMUv3p5 relaxes the behavior of PMCR.{IMP, IDCODE}, and deprecates use of these fields.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p5 is OPTIONAL from Armv8.4.

In an Armv8.5 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p5 is implemented.

If FEAT_PMUv3p5 is implemented, then FEAT_PMUv3p4 is implemented.

The following fields identify the presence of FEAT_PMUv3p5:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- Behavior on overflow.
- Controlling the PMU counters.
- PMU Event Descriptions.

FEAT_RNG, Random number generator

FEAT_RNG introduces the RNDR and RNDRRS registers. Reads to these registers return a 64-bit random number. A read to RNDRRS will cause a reseeding of the random number before the generation of the random number that is returned.

This feature is supported in AArch64 state only.

FEAT_RNG is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_RNG:

- ID_AA64ISAR0_EL1.RNDR.

For more information, see:

- Effect of random number generation instructions on Condition flags
- Appendix K14 Random Number Generation

FEAT_RNG_TRAP, Trapping support for RNDR/RNDRRS

FEAT_RNG_TRAP introduces support for EL3 trapping of reads of the RNDR and RNDRRS registers.

This feature is supported in AArch64 state only.

FEAT_RNG_TRAP is OPTIONAL from Armv8.4.

The following field identifies the presence of FEAT_RNG_TRAP:

- ID_AA64PFR1_EL1.RNDR_trap.

FEAT_S2TGran16K, Support for 16KB memory translation granule size at stage 2

FEAT_S2TGran16K is OPTIONAL.

If FEAT_S2TGran16K is implemented, then FEAT_AA64EL2 and FEAT_TGran16K are implemented.

The following field identifies the presence of FEAT_S2TGran16K:

- ID_AA64MMFR0_EL1.TGran16_2.

FEAT_S2TGran4K, Support for 4KB memory translation granule size at stage 2

FEAT_S2TGran4K is OPTIONAL.

If FEAT_S2TGran4K is implemented, then FEAT_AA64EL2 and FEAT_TGran4K are implemented.

The following field identifies the presence of FEAT_S2TGran4K:

- ID_AA64MMFR0_EL1.TGran4_2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_S2TGran64K, Support for 64KB memory translation granule size at stage 2

FEAT_S2TGran64K is OPTIONAL.

If FEAT_S2TGran64K is implemented, then FEAT_AA64EL2 and FEAT_TGran64K are implemented.

The following field identifies the presence of FEAT_S2TGran64K:

- ID_AA64MMFR0_EL1.TGran64_2.

FEAT_SB, Speculation Barrier

Speculation Barrier FEAT_SB introduces a barrier to control speculation.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SB is OPTIONAL from Armv8.0.

FEAT_SB is mandatory from Armv8.5.

The following fields identify the presence of FEAT_SB:

- ID_AA64ISAR1_EL1.SB.
- ID_ISAR6_EL1.SB.
- ID_ISAR6.SB.

For more information, see:

- Speculation Barrier (SB).
- Barriers and CLREX instructions.
- Speculation Barrier (SB).
- Miscellaneous instructions.

FEAT_SPECRES, Speculation restriction instructions

FEAT_SPECRES introduces System instructions that prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPECRES is OPTIONAL from Armv8.0.

FEAT_SPECRES is mandatory from Armv8.5.

The following fields identify the presence of FEAT_SPECRES:

- ID_AA64ISAR1_EL1.SPECRES.
- ID_ISAR6_EL1.SPECRES.
- ID_ISAR6.SPECRES.

For more information, see:

- Prediction restriction instructions.
- Execution, data prediction and prefetching restriction System instructions.
- Execution and data prediction restriction System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SSBS, Speculative Store Bypass Safe

FEAT_SSBS allows software to indicate whether hardware is permitted to load or store speculatively in a manner that could give rise to a cache timing side channel, which in turn could be used to derive an address from values loaded to a register from memory.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SSBS is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_SSBS:

- ID_AA64PFR1_EL1.SSBS.
- ID_PFR2_EL1.SSBS.
- ID_PFR2.SSBS.

For more information, see:

- AArch64 Speculative Store Bypass Safe.
- AArch32 Speculative Store Bypass Safe.

FEAT_SSBS2, MRS and MSR instructions for SSBS version 2

FEAT_SSBS2 provides controls for the MSR and MRS instructions to read and write the PSTATE.SSBS field.

This feature is supported in AArch64 state only.

FEAT_SSBS2 is OPTIONAL from Armv8.0.

If FEAT_SSBS2 is implemented, then FEAT_SSBS is implemented.

The following field identifies the presence of FEAT_SSBS2:

- ID_AA64PFR1_EL1.SSBS.

For more information, see:

- AArch64 Speculative Store Bypass Safe.
- AArch32 Speculative Store Bypass Safe.

A2.2.6.1 Features added to the Armv8.5 extension in later releases

- FEAT_MTE3.
- FEAT_MTE_ASYNC.

A2.2.7 The Armv8.6 architecture extension

The Armv8.6 architecture extension is an extension to Armv8.5. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.6 compliant if all of the following apply:

- It is Armv8.5 compliant.
- It includes all of the Armv8.6 architectural features that are mandatory.

An Armv8.6 compliant implementation can additionally include:

- Armv8.6 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.7, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AA32BF16, AArch32 BFloat16 instructions

FEAT_AA32BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch32 state. This format supports:

- Arithmetic instructions to multiply BFloat16 values and accumulate into single-precision results.
- Arithmetic instructions to accelerate dot product and matrix multiply-accumulate of BFloat16 values into single-precision results.
- Instructions to convert single-precision floating-point values to BFloat16 format.

This feature is supported in AArch32 state only.

FEAT_AA32BF16 is OPTIONAL from Armv8.2.

If FEAT_AA32BF16 is implemented, then FEAT_BF16 is implemented.

If FEAT_AA32BF16 is implemented, then FEAT_AA32 is implemented.

The following fields identify the presence of FEAT_AA32BF16:

- ID_ISAR6_EL1.BF16.
- ID_ISAR6.BF16.

For more information, see:

- BFloat16 floating-point format.
- Advanced SIMD BFloat16 instructions.
- Floating-point data-processing.

FEAT_AMUv1p1, Activity Monitors Extension version 1.1

FEAT_AMUv1p1 introduces support for virtualization of Activity Monitors event counters, and introduces controls to disable access to auxiliary event counters below the highest Exception level.

This feature is supported in AArch32 state and AArch64 state, if the hypervisor is using AArch64.

FEAT_AMUv1p1 is OPTIONAL from Armv8.5.

If FEAT_AMUv1p1 is implemented, then FEAT_AMUv1 is implemented.

The following fields identify the presence of FEAT_AMUv1p1:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see The Activity Monitors Extension.

FEAT_BF16, AArch64 BFloat16 instructions

FEAT_BF16 supports the BFloat16 floating-point format in AArch64 state. This format supports:

- Arithmetic instructions to multiply BFloat16 values and accumulate into single-precision results.
- Arithmetic instructions to accelerate dot product and matrix multiply-accumulate of BFloat16 values into single-precision results.
- Instructions to convert single-precision floating-point values to BFloat16 format.

This feature is supported in AArch64 state only.

FEAT_BF16 is OPTIONAL from Armv8.2.

In an Armv8.6 implementation, if FEAT_FP is implemented, FEAT_BF16 is implemented.

The following fields identify the presence of FEAT_BF16:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64ISAR1_EL1.BF16.
- ID_AA64ZFR0_EL1.BF16.

For more information, see:

- BFloat16 floating-point format.
- Summary of BFloat16 instruction behaviors.

FEAT_CP15SDISABLE2, CP15SDISABLE2

FEAT_CP15SDISABLE2 provides an implementation-defined mechanism, the **CP15SDISABLE2** signal, which when asserted HIGH prevents writes to a set of Secure CP15 registers. This signal is analogous to the existing **CP15SDISABLE** signal.

This feature is supported only when EL3 is executing in AArch32 state.

FEAT_CP15SDISABLE2 is OPTIONAL from Armv8.0.

If FEAT_CP15SDISABLE2 is implemented, then FEAT_AA32EL3 is implemented.

For more information, see The CP15SDISABLE and CP15SDISABLE2 input signals.

FEAT_DGH, Data Gathering Hint

FEAT_DGH introduces the Data Gathering Hint instruction to the hint space.

These instructions are added to the A64 instruction set only.

FEAT_DGH is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_DGH:

- ID_AA64ISAR1_EL1.DGH.

For more information, see Hint instructions.

FEAT_ECV, Enhanced Counter Virtualization

FEAT_ECV enhances the Generic Timer architecture. FEAT_ECV provides:

- Self-synchronizing views of the virtual and physical timers in AArch64 state and AArch32 state.
- The ability to scale the generation of the event stream when executing in AArch64 state or AArch32 state.
- When EL2 is using AArch64 state, traps to EL0 and EL1 access to the virtual counter or timer registers, and the physical timer registers when accessed using an EL02 mnemonic. The traps are configured in CNTHCTL_EL2, and apply to EL1 and EL0 accesses, whether EL1 and EL0 are in AArch64 state or AArch32 state.

For more information on the offset to views of physical time, see FEAT_ECV_POFF.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ECV is OPTIONAL from Armv8.5.

FEAT_ECV is mandatory from Armv8.6.

The following field identifies the presence of FEAT_ECV:

- ID_AA64MMFR0_EL1.ECV.

For more information, see:

- Self-hosted trace timestamps.
- The profiling data.
- The AArch64 view of the Generic Timer.
- The AArch32 view of the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ECV_POFF, Enhanced Counter Virtualization Physical Offset

FEAT_ECV_POFF provides an offset between the EL1 or EL0 view of physical time, and the EL2 or EL3 view of physical time.

The offset to views of physical time at EL1 and EL0 apply in AArch64 state and AArch32 state.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ECV_POFF is OPTIONAL from Armv8.5.

If FEAT_ECV_POFF is implemented, then FEAT_ECV is implemented.

When FEAT_AA64 and FEAT_ECV_POFF are implemented, FEAT_AA64EL2 is implemented.

If FEAT_RME is implemented, then FEAT_ECV_POFF is implemented.

The following field identifies the presence of FEAT_ECV_POFF:

- ID_AA64MMFR0_EL1.ECV.

FEAT_FGT, Fine Grain Traps

FEAT_FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or small groups of System registers and instructions, and traps to EL3 and EL2 of the Debug Communications Channel registers. The traps are independent of existing controls.

This feature is supported in AArch64, and when EL1 is using AArch64, EL0 accesses using AArch32 are also trapped.

FEAT_FGT is OPTIONAL from Armv8.5.

In an Armv8.6 implementation, if FEAT_AA64EL2 or FEAT_AA64EL3 is implemented, FEAT_FGT is implemented.

The following field identifies the presence of FEAT_FGT:

- ID_AA64MMFR0_EL1.FGT.

For more information, see Configurable instruction controls.

FEAT_HPMN0, Setting of MDCR_EL2.HPMN to zero

FEAT_HPMN0 permits a hypervisor to provide zero PMU event counters for a guest operating system by setting MDCR_EL2.HPMN to zero.

This feature is supported in both AArch64 and AArch32 states.

FEAT_HPMN0 is OPTIONAL from Armv8.5.

In an Armv8.8 implementation, if FEAT_PMUv3 and FEAT_EL2 are implemented, FEAT_HPMN0 is implemented.

If FEAT_HPMN0 is implemented, then FEAT_EL2 is implemented.

If FEAT_HPMN0 is implemented, then FEAT_PMUv3 and FEAT_FGT are implemented.

The following fields identify the presence of FEAT_HPMN0:

- ID_AA64DFR0_EL1.HPMN0.
- ID_DFR1_EL1.HPMN0.
- ID_DFR1.HPMN0.

For more information, see:

- Interaction with EL2.
- Controlling the PMU counters.
- The Performance Monitors Extension.
- The Performance Monitors Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MPAMv0p1, Memory System Resource Partitioning and Monitoring extension version 0.1

FEAT_MPAMv0p1 introduces support for version 0.1 of the MPAM extension.

This feature is supported in AArch64 state only.

FEAT_MPAMv0p1 is OPTIONAL from Armv8.5.

If FEAT_MPAMv0p1 is implemented, then FEAT_MPAM is implemented.

The following fields identify the presence of FEAT_MPAMv0p1:

- ID_AA64PFR0_EL1.MPAM.
- ID_AA64PFR1_EL1.MPAM_frac.

For more information, see MPAM PE Architecture.

FEAT_MPAMv1p1, Memory Partitioning and Monitoring extension version 1.1

FEAT_MPAMv1p1 introduces support for version 1.1 of the MPAM extension.

This feature is supported in AArch64 state only.

FEAT_MPAMv1p1 is OPTIONAL from Armv8.5.

If FEAT_MPAMv1p1 is implemented, then FEAT_MPAM is implemented.

If FEAT_MPAMv1p1 is implemented, then FEAT_MPAMv1p0 is implemented.

If FEAT_MPAMv1p1 is implemented, then FEAT_MPAMv0p1 is not implemented.

The following fields identify the presence of FEAT_MPAMv1p1:

- ID_AA64PFR0_EL1.MPAM.
- ID_AA64PFR1_EL1.MPAM_frac.

For more information, see MPAM PE Architecture.

FEAT_MTPMU, Multi-threaded PMU extensions

FEAT_MTPMU introduces controls to disable PMEVTYPER<n>_EL0.MT.

From Armv8.6, when FEAT_PMUv3 is implemented, multithreaded event counting is only supported in multithreaded implementations that also include FEAT_MTPMU.

This feature is supported in both AArch64 and AArch32 states.

FEAT_MTPMU is OPTIONAL from Armv8.5.

If FEAT_MTPMU is implemented, then FEAT_PMUv3 is implemented.

If FEAT_MTPMU is implemented, then FEAT_EL2 or FEAT_EL3 is implemented.

The following fields identify the presence of FEAT_MTPMU:

- ID_AA64DFR0_EL1.MTPMU.
- ID_DFR1.MTPMU.
- ID_DFR1_EL1.MTPMU.

For more information, see:

- Multithreaded implementations.
- MDCR_EL3.MTPME
- SDCR.MTPME
- MDCR_EL2.MTPME
- HDCR.MTPME.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TWED, Delayed Trapping of WFE

FEAT_TWED introduces support for configurable delayed trapping of the WFE and WFET instructions.

FEAT_TWED is OPTIONAL from Armv8.5.

The following field identifies the presence of FEAT_TWED:

- ID_AA64MMFR1_EL1.TWED.

For more information, see The Wait for Event and Wait for Event with Timeout instructions.

A2.2.7.1 Features added to the Armv8.6 extension in later releases

- FEAT_SPE_DPFZS.

A2.2.8 The Armv8.7 architecture extension

The Armv8.7 architecture extension is an extension to Armv8.6. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.7 compliant if all of the following apply:

- It is Armv8.6 compliant.
- It includes all of the Armv8.7 architectural features that are mandatory.

An Armv8.7 compliant implementation can additionally include:

- Armv8.7 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.8, subject only to those constraints that require that certain features be implemented together.

FEAT_AFP, Alternate floating-point behavior

FEAT_AFP allows alternate behavior for specified floating-point instructions including:

- Flushing of denormalized numbers to zero can be controlled separately on inputs and outputs.
- Alternate NaN propagation rules and Default NaN values can apply.
- Certain scalar SIMD and floating-point instructions can be configured to preserve higher numbered SIMD vector elements.
- Changes to floating-point exception generation.

This feature is supported in AArch64 state only.

FEAT_AFP is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_FP is implemented, FEAT_AFP is implemented.

If FEAT_AFP is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_AFP:

- ID_AA64MMFR1_EL1.AFP.

For more information, see:

- Flushing denormalized numbers to zero.
- NaN handling and the Default NaN.
- Rounding.
- Floating-point exceptions and exception traps.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_EBF16, AArch64 Extended BFloat16 behaviors

FEAT_EBF16 supports the Extended BFloat16 behaviors.

This feature is supported in AArch64 state only.

FEAT_EBF16 is OPTIONAL from Armv8.2.

If FEAT_EBF16 is implemented, then FEAT_BF16 is implemented.

If FEAT_EBF16 is implemented, then FEAT_AdvSIMD, FEAT_FP, FEAT_SVE, or FEAT_SME is implemented.

The following fields identify the presence of FEAT_EBF16:

- ID_AA64ISAR1_EL1.BF16.
- ID_AA64ZFR0_EL1.BF16.

For more information, see:

- BFloat16 floating-point format.
- Floating-point behaviors for instructions that calculate sum-of-products.

FEAT_HCX, Support for the HCRX_EL2 register

FEAT_HCX introduces the Extended Hypervisor Configuration Register, HCRX_EL2, that provides configuration controls for virtualization in addition to those provided by HCR_EL2, including defining whether various operations are trapped to EL2.

This feature is supported in AArch64 state only.

FEAT_HCX is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_AA64EL2 is implemented, FEAT_HCX is implemented.

When FEAT_AA64 and FEAT_HCX are implemented, FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_HCX:

- ID_AA64MMFR1_EL1.HCX.

For more information, see Configurable instruction controls.

FEAT_LPA2, Larger physical address for 4KB and 16KB translation granules

FEAT_LPA2:

- Allows a larger VA space for each translation table base register of up to 52 bits when using the 4KB or 16KB translation granules.
- Allows a larger intermediate physical address (IPA) and PA space of up to 52 bits when using the 4KB or 16KB translation granules.
- Allows a level 0 block size where the block covers a 512GB address range for the 4KB translation granule if the implementation supports 52 bits of PA.
- Allows a level 1 block size where the block covers a 64GB address range for the 16KB translation granule if the implementation supports 52 bits of PA.

This feature is supported in AArch64 state only.

FEAT_LPA2 is OPTIONAL from Armv8.6.

If FEAT_LPA2 is implemented, then FEAT_LVA is implemented.

The following fields identify the presence of FEAT_LPA2:

- ID_AA64MMFR0_EL1.TGran4_2.
- ID_AA64MMFR0_EL1.TGran16_2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR0_EL1.TGran4.
- ID_AA64MMFR0_EL1.TGran16.

For more information, see:

- Implemented physical address size.
- Output address size configuration.
- Supported virtual address ranges.
- Input address size configuration.
- Intermediate physical address size configuration.
- Translation using the 4KB granule.
- Translation using the 16KB granule.
- Table descriptor format.
- Block descriptor and Page descriptor formats.

FEAT_LS64, Support for 64-byte loads and stores without status

FEAT_LS64 introduces support for single-copy atomic 64-byte loads and stores without status result. For more information, see:

- LD64B.
- ST64B.

This feature is supported in AArch64 state only.

FEAT_LS64 is OPTIONAL from Armv8.6.

The following field identifies the presence of FEAT_LS64:

- ID_AA64ISAR1_EL1.LS64.

For more information, see Single-copy atomic 64-byte load/store.

FEAT_LS64_ACCDATA, Support for 64-byte EL0 stores with status

FEAT_LS64_ACCDATA introduces support for single-copy atomic 64-byte EL0 stores with status result. For more information, see:

- ST64BV0.
- ACCDATA_EL1.

Note

The meaning of any status being returned by the ST64BV0 instruction is defined by the peripheral providing the response.

This feature is supported in AArch64 state only.

FEAT_LS64_ACCDATA is OPTIONAL from Armv8.6.

If FEAT_LS64_ACCDATA is implemented, then FEAT_LS64_V is implemented.

The following field identifies the presence of FEAT_LS64_ACCDATA:

- ID_AA64ISAR1_EL1.LS64.

For more information, see Single-copy atomic 64-byte load/store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LS64_V, Support for 64-byte stores with status

FEAT_LS64_V introduces support for single-copy atomic 64-byte stores with status result. For more information, see:

- ST64BV.

Note

The meaning of any status being returned by the ST64BV instruction is defined by the peripheral providing the response.

This feature is supported in AArch64 state only.

FEAT_LS64_V is OPTIONAL from Armv8.6.

If FEAT_LS64_V is implemented, then FEAT_LS64 is implemented.

The following field identifies the presence of FEAT_LS64_V:

- ID_AA64ISAR1_EL1.LS64.

For more information, see Single-copy atomic 64-byte load/store.

FEAT_MTE3, MTE Asymmetric Fault Handling

FEAT_MTE3 introduces support for asymmetric Tag Check Fault handling.

This feature is supported in AArch64 state only.

FEAT_MTE3 is OPTIONAL from Armv8.5.

In an Armv8.7 implementation, if FEAT_MTE_ASYNC is implemented, FEAT_MTE3 is implemented.

If FEAT_MTE3 is implemented, then FEAT_MTE2 is implemented.

The following field identifies the presence of FEAT_MTE3:

- ID_AA64PFR1_EL1.MTE.

For more information, see The Memory Tagging Extension.

FEAT_MTE_ASYM_FAULT, Memory tagging asymmetric faults

FEAT_MTE_ASYM_FAULT introduces support for asymmetric MTE Tag Check fault handling.

This feature is supported in AArch64 state only.

FEAT_MTE_ASYM_FAULT is OPTIONAL.

If FEAT_MTE3 is implemented, then FEAT_MTE_ASYM_FAULT is implemented.

If FEAT_MTE_ASYM_FAULT is implemented, then FEAT_MTE3 is implemented.

In an Armv8.9 implementation, if FEAT_MTE_ASYM_FAULT is implemented, FEAT_MTE_ASYNC is implemented.

The following field identifies the presence of FEAT_MTE_ASYM_FAULT:

- ID_AA64PFR1_EL1.MTE.

FEAT_PAN3, Support for SCTLR_ELx.EPAN

FEAT_PAN3 introduces a bit to SCTLR_EL1 and SCTLR_EL2, EPAN, to support using Privileged Access Never with instruction accesses for stage 1 translation regimes.

This feature is supported in AArch64 state only.

FEAT_PAN3 is OPTIONAL from Armv8.1.

FEAT_PAN3 is mandatory from Armv8.7.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_PAN3 is implemented, then FEAT_PAN2 is implemented.

The following field identifies the presence of FEAT_PAN3:

- ID_AA64MMFR1_EL1.PAN.

For more information, see PSTATE PAN.

FEAT_PMUv3p7, Armv8.7 PMU extensions

FEAT_PMUv3p7 introduces the following to the Performance Monitors Extension:

- PMU counters can be frozen when an event counter has an unsigned overflow.
- Event counters can be prohibited from counting events at EL3 without affecting other Exception levels in Secure state.
- The cycle counter can be prohibited from counting cycles at EL3 without affecting other Exception levels in Secure state.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p7 is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p7 is implemented.

If FEAT_PMUv3p7 is implemented, then FEAT_PMUv3p5 is implemented.

The following fields identify the presence of FEAT_PMUv3p7:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.
- PMCFGR.FZO.

For more information, see:

- Controlling the PMU counters.
- Freezing event counters.

FEAT_RPRES, Increased precision of FRECPE and FRSQRTE

FEAT_RPRES allows an increase in the precision of the single-precision floating-point reciprocal estimate and reciprocal square root estimate from an 8-bit mantissa to a 12-bit mantissa.

This feature is supported in AArch64 state only.

FEAT_RPRES is OPTIONAL from Armv8.6.

If FEAT_RPRES is implemented, then FEAT_AFP is implemented.

The following field identifies the presence of FEAT_RPRES:

- ID_AA64ISAR2_EL1.RPRES.

For more information, see `RecipEstimate()` and `RecipSqrtEstimate()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPE_FnE, Statistical Profiling inverse event filter

FEAT_SPE_FnE provides the capability to filter sample records by the inverse value of bits in the sampled Events packet.

This feature is supported in AArch64 state only.

FEAT_SPE_FnE is OPTIONAL.

If FEAT_SPE_FnE is implemented, then FEAT_SPEv1p2 is implemented.

If FEAT_SPEv1p2 is implemented, then FEAT_SPE_FnE is implemented.

The following field identifies the presence of FEAT_SPE_FnE:

- PMSIDR_EL1.FnE.

For more information, see Filtering sample records.

FEAT_SPE_PBT, Statistical Profiling previous branch target

FEAT_SPE_PBT provides support for generating a packet that provides the target address for the previous taken branch.

This feature is supported in AArch64 state only.

FEAT_SPE_PBT is OPTIONAL.

If FEAT_SPE_PBT is implemented, then FEAT_SPEv1p2 is implemented.

The following field identifies the presence of FEAT_SPE_PBT:

- PMSIDR_EL1.PBT.

For more information, see:

- Previous branch target.
- Address packet.

FEAT_SPEv1p2, Statistical Profiling Extensions version 1.2

FEAT_SPEv1p2 introduces the following to the Statistical Profiling Extension:

- Controls to freeze the PMU event counters after an SPE buffer management event occurs.
- A discard mode that allows all SPE data to be discarded rather than written to memory.

This feature is supported in AArch64 state only.

FEAT_SPEv1p2 is OPTIONAL from Armv8.6.

In an Armv8.7 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p2 is implemented.

If FEAT_SPEv1p2 is implemented, then FEAT_SPEv1p1 is implemented.

The following field identifies the presence of FEAT_SPEv1p2:

- ID_AA64DFR0_EL1.PMSVer.

For more information, see:

- Freezing event counters.
- Discard mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_WFxT, WFE and WFI instructions with timeout

FEAT_WFxT introduces WFET and WFIT. These instructions support the generation of a local timeout event to act as a wake-up event for the PE when the virtual count in CNTVCT_EL0 equals or exceeds the value supplied by the instruction for the first time.

These instructions are added to the A64 instruction set only.

FEAT_WFxT is OPTIONAL from Armv8.6.

FEAT_WFxT is mandatory from Armv8.7.

The following field identifies the presence of FEAT_WFxT:

- ID_AA64ISAR2_EL1.WFxT.

For more information, see:

- Wait for Event.
- Wait for Interrupt mechanism.

FEAT_XS, XS attribute

FEAT_XS introduces the XS attribute for memory to indicate that an access could take a long time to complete. This feature provides variants of DSB instructions and TLB maintenance instructions, the completion of which does not depend on the completion of memory accesses with the XS attribute.

FEAT_XS adds:

- A mechanism to define the XS attribute for memory.
- An optional nXS variant to the AArch64 DSB instruction and OPTIONAL nXS qualifier to each AArch64 TLBI instruction to handle memory accesses with the XS attribute.
- The HCRX_EL2.FGTnXS bit to determine the behavior of fine-grained traps in HFGITR_EL2 for TLB maintenance instructions with the nXS qualifier.
- The HCRX_EL2.FnXS bit to determine the behavior of pre-existing TLB maintenance instructions in relation to the XS attribute.

This feature is supported in AArch64 state only, but the XS attribute also impacts AArch32 state execution.

FEAT_XS is OPTIONAL from Armv8.6.

FEAT_XS is mandatory from Armv8.7.

The following field identifies the presence of FEAT_XS:

- ID_AA64ISAR1_EL1.XS.

For more information, see:

- Data Synchronization Barrier (DSB).
- Behavior when stage 1 address translation is disabled.
- Block descriptor and Page descriptor formats.
- Stage 1 memory type and Cacheability attributes.
- XS attribute modifier.
- Overview of memory region attributes for stage 1 translations.
- Ordering and completion of TLB maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2.8.1 Features added to the Armv8.7 extension in later releases

- FEAT_CSSC.
- FEAT_HAFT.
- FEAT_MTE4.
- FEAT_MTE_PERM.

A2.2.9 The Armv8.8 architecture extension

The Armv8.8 architecture extension is an extension to Armv8.7. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.8 compliant if all of the following apply:

- It is Armv8.7 compliant.
- It includes all of the Armv8.8 architectural features that are mandatory.

An Armv8.8 compliant implementation can additionally include:

- Armv8.8 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.9, subject only to those constraints that require that certain features be implemented together.

FEAT_CMOW, Control for cache maintenance permission

FEAT_CMOW introduces support for controlling the required permissions for some cache maintenance instructions that operate by VA and perform translation such that:

- Stage 1 can be configured to generate a Permission fault if write permission is not present for cache maintenance instructions executed at EL0.
- Stage 2 can be configured to generate a Permission fault if write permission is not present for cache maintenance instructions executed at EL1 or EL0.

This feature is supported in AArch64 state only, but also impacts AArch32 instructions.

FEAT_CMOW is OPTIONAL from Armv8.7.

FEAT_CMOW is mandatory from Armv8.8.

The following field identifies the presence of FEAT_CMOW:

- ID_AA64MMFR1_EL1.CMOW.

For more information, see:

- A64 Cache maintenance instructions.
- Permission fault.

FEAT_Debugv8p8, Debug v8.8

FEAT_Debugv8p8 introduces support to allow another exception to be taken after an exception which generates an Exception Catch debug event, before the PE halts.

This feature is supported in both AArch64 and AArch32 states.

FEAT_Debugv8p8 is OPTIONAL from Armv8.7.

FEAT_Debugv8p8 is mandatory from Armv8.8.

If FEAT_Debugv8p8 is implemented, then FEAT_Debugv8p4 is implemented.

The following fields identify the presence of FEAT_Debugv8p8:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- DBGDIDR.Version.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

For more information, see Exception Catch debug event.

FEAT_HBC, Hinted conditional branches

FEAT_HBC provides the BC.cond instruction to give a conditional branch to a label at a PC-relative offset, with a hint that this branch is very unlikely to change direction.

This feature is supported in AArch64 state only.

FEAT_HBC is OPTIONAL from Armv8.7.

FEAT_HBC is mandatory from Armv8.8.

The following field identifies the presence of FEAT_HBC:

- ID_AA64ISAR2_EL1.BC.

For more information, see Conditional branch.

FEAT_MOPS, Standardization of memory operations

FEAT_MOPS provides instructions that perform a memory copy or memory set, and introduces Memory Copy and Memory Set exceptions.

FEAT_MOPS also introduces the HCRX_EL2.{MSCEn, MCE2}, SCTLR_EL1.MSCEn, and SCTLR_EL2.MSCEn control bits.

This feature is supported in AArch64 state only.

FEAT_MOPS is OPTIONAL from Armv8.7.

FEAT_MOPS is mandatory from Armv8.8.

The following field identifies the presence of FEAT_MOPS:

- ID_AA64ISAR2_EL1.MOPS.

For more information, see:

- Memory Copy and Memory Set instructions.
- Memory Copy and Memory Set exceptions.

FEAT_NMI, Non-maskable Interrupts

FEAT_NMI provides a mechanism to support *non-maskable interrupts* (NMI) and *less-masked interrupts* (LMI). In addition to legacy behavior, the feature includes the following:

- A mode for supporting an LMI interrupt mask that is distinct from PSTATE.{I, F}.
- A mode for supporting a limited NMI, where the value when PSTATE.SP is 1 is taken as an interrupt mask for all interrupts targeting that Exception level, and where the LMI interrupt mask can also be used

FEAT_NMI adds:

- The AllIntMask variable.
- An Optional Superpriority attribute to denote virtual and physical IRQ and FIQ interrupts as non-maskable.
- The SCTLR_ELx.{NMI, SPINTMASK} control bits.
- The PSTATE.ALLINT bit and associated instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The HCRX_EL2.TALLINT bit to enable trapping of ALLINT instructions at EL1.

This feature is supported in AArch64 state only.

FEAT_NMI is OPTIONAL from Armv8.7.

FEAT_NMI is mandatory from Armv8.8.

The following field identifies the presence of FEAT_NMI:

- ID_AA64PFR1_EL1.NMI.

For more information, see:

- Asynchronous exception types.
- Virtual interrupts.
- PSTATE fields that are meaningful in AArch64 state.
- WFE wake-up events.

FEAT_PMUv3_EXT64, 64-bit external interface to the Performance Monitors

FEAT_PMUv3_EXT64 indicates the external Performance Monitors registers are implemented as 64-bit registers. The 32-bit CoreSight management registers remain 32-bit registers.

FEAT_PMUv3_EXT64 is OPTIONAL from Armv8.8.

If FEAT_PMUv3_EXT64 is implemented, then FEAT_PMUv3_EXT is implemented.

If FEAT_PMUv3_EXT64 is implemented, then FEAT_PMUv3_EXT32 is not implemented.

The following field identifies the presence of FEAT_PMUv3_EXT64:

- PMDEVARCH.ARCHPART.

For more information, see Recommended External Interface to the Performance Monitors.

FEAT_PMUv3_TH, Event counting threshold

FEAT_PMUv3_TH introduces threshold condition controls to each PMEVTYPER<n>_EL0 register. This feature permits the counter to count only when PMEVTYPER<n>.{MT, evtCount} describe an event whose count meets a specified threshold condition.

This feature is supported in both AArch64 and AArch32 states. The threshold condition controls are accessible only in AArch64 state. However, threshold conditions still apply in AArch32 state.

FEAT_PMUv3_TH is OPTIONAL from Armv8.7.

If FEAT_PMUv3_TH is implemented, then FEAT_PMUv3 is implemented.

The following fields identify the presence of FEAT_PMUv3_TH:

- PMMIR_EL1.THWIDTH.
- PMMIR.THWIDTH.
- PMMIR.THWIDTH.

For more information, see Event threshold and edge counting.

FEAT_PMUv3p8, Armv8.8 PMU extensions

FEAT_PMUv3p8 introduces the following to the Performance Monitors Extension:

- The Common event number space is extended to include the ranges `0x0040-0x00BF` and `0x4040-0x40BF`.
- For an event counter n, if any reserved or unimplemented PMU event number is written to PMEVTYPER<n>.evtCount, then event counter n does not count, and a read of PMEVTYPER<n>.evtCount returns the value written.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p8 is OPTIONAL from Armv8.7.

In an Armv8.8 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p8 is implemented.

If FEAT_PMUv3p8 is implemented, then FEAT_PMUv3p7 is implemented.

The following fields identify the presence of FEAT_PMUv3p8:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see PMU Event Descriptions.

FEAT_SCTLR2, Extension to SCTLR_ELx

FEAT_SCTLR2 introduces the SCTLR2_ELx registers, which provide top-level control of the system, including its memory system. These registers are extensions of the corresponding SCTLR_ELx registers.

This feature is supported in AArch64 state only.

FEAT_SCTLR2 is OPTIONAL from Armv8.0.

FEAT_SCTLR2 is mandatory from Armv8.9.

When FEAT_SCTLR2 and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following field identifies the presence of FEAT_SCTLR2:

- ID_AA64MMFR3_EL1.SCTLRX.

FEAT_SPEv1p3, Statistical Profiling Extensions version 1.3

FEAT_SPEv1p3 introduces the following to the Statistical Profiling Extension:

- Support for sampling Tag operations.
- Support for sampling Memory Copy and Set operations.

This feature is supported in AArch64 state only.

FEAT_SPEv1p3 is OPTIONAL from Armv8.7.

In an Armv8.8 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p3 is implemented.

If FEAT_SPEv1p3 is implemented, then FEAT_SPEv1p2 is implemented.

The following field identifies the presence of FEAT_SPEv1p3:

- ID_AA64DFR0_EL1.PMSVer.

For more information, see:

- Additional information for each profiled memory access operation.
- About the Statistical Profiling Extension sample records.
- Address packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TCR2, Support for TCR2_ELx

FEAT_TCR2 introduces the TCR2_ELx registers which provide top-level control of the EL1&0 and EL2&0 translation regimes respectively. These registers are extensions of the corresponding TCR_ELx registers.

This feature is supported in AArch64 state only.

FEAT_TCR2 is OPTIONAL from Armv8.0.

FEAT_TCR2 is mandatory from Armv8.9.

When FEAT_TCR2 and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following field identifies the presence of FEAT_TCR2:

- ID_AA64MMFR3_EL1.TCRX.

FEAT_TIDCP1, EL0 use of IMPLEMENTATION DEFINED functionality

FEAT_TIDCP1 introduces a control at EL1 and EL2 to enable trapping of EL0 accesses to registers that might control IMPLEMENTATION DEFINED functions.

This feature introduces controls only in AArch64 state, and controls IMPLEMENTATION DEFINED execution at EL0 in both AArch32 and AArch64 states.

FEAT_TIDCP1 is OPTIONAL from Armv8.7.

FEAT_TIDCP1 is mandatory from Armv8.8.

The following field identifies the presence of FEAT_TIDCP1:

- ID_AA64MMFR1_EL1.TIDCP1.

For more information, see Prioritization of Synchronous exceptions taken to AArch64 state

A2.2.10 The Armv8.9 architecture extension

The Armv8.9 architecture extension is an extension to Armv8.8. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.9 compliant if all of the following apply:

- It is Armv8.8 compliant.
- It includes all of the Armv8.9 architectural features that are mandatory.

An Armv8.9 compliant implementation can additionally include:

- Armv8.9 features that are optional.

FEAT_ADERR, Asynchronous Device Error Exceptions

FEAT_ADERR introduces controls for whether an error signaled on a load from Device memory is handled precisely and synchronously.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ADERR is OPTIONAL from Armv8.8.

If FEAT_ADERR is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_ADERR and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following fields identify the presence of FEAT_ADERR:

- ID_AA64MMFR3_EL1.ADERR.
- ID_AA64MMFR3_EL1.SDERR.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- About the RAS Extension.
- Taking error exceptions.

FEAT_AIE, Memory Attribute Index Enhancement

FEAT_AIE increases the stage 1 descriptor attribute index bit width from 3 to 4, allowing use of up to 16 memory attributes.

This feature is supported in AArch64 state only.

FEAT_AIE is OPTIONAL from Armv8.8.

If FEAT_AIE is implemented, then FEAT_TCR2 is implemented.

If FEAT_AIE is implemented, then FEAT_HPDS is implemented.

The following field identifies the presence of FEAT_AIE:

- ID_AA64MMFR3_EL1.AIE.

For more information, see Stage 1 memory type and Cacheability attributes.

FEAT_AMU_EXT64, 64-bit external Activity Monitors extension

FEAT_AMU_EXT64 indicates the external AMU registers are implemented as 64-bit registers.

FEAT_AMU_EXT64 is OPTIONAL.

If FEAT_AMU_EXT64 is implemented, then FEAT_AMU_EXT is implemented.

If FEAT_AMU_EXT64 is implemented, then FEAT_AMU_EXT32 is not implemented.

The following field identifies the presence of FEAT_AMU_EXT64:

- AMDEVARCH.ARCHID.

For more information, see Recommended External Interface to the Activity Monitors.

FEAT_ANERR, Asynchronous Normal Error Exceptions

FEAT_ANERR introduces controls for whether an error signaled on a load from Normal memory is handled precisely and synchronously.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ANERR is OPTIONAL from Armv8.8.

If FEAT_ANERR is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_ANERR and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following fields identify the presence of FEAT_ANERR:

- ID_AA64MMFR3_EL1.ANERR.
- ID_AA64MMFR3_EL1.SNERR.

For more information, see:

- About the RAS Extension.
- Taking error exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ATS1A, Address Translation operations that ignore stage 1 permissions

FEAT_ATS1A introduces instructions that provide the output address and attributes of a valid translation without checking for stage 1 permissions.

These instructions are added to the A64 instruction set only.

FEAT_ATS1A is OPTIONAL from Armv8.8.

The following field identifies the presence of FEAT_ATS1A:

- ID_AA64ISAR2_EL1.ATS1A.

FEAT_CLRBHB, Support for Clear Branch History instruction

FEAT_CLRBHB provides a CLRBHB instruction, which can be used to clear the branch history.

This feature is supported in both AArch64 and AArch32 states.

FEAT_CLRBHB is OPTIONAL from Armv8.0.

FEAT_CLRBHB is mandatory from Armv8.9.

The following fields identify the presence of FEAT_CLRBHB:

- ID_AA64ISAR2_EL1.CLRBHB.
- ID_ISAR6_EL1.CLRBHB.
- ID_ISAR6.CLRBHB.

For more information, see:

- Branch prediction.
- AArch32 cache and branch predictor maintenance instructions.

FEAT_CSSC, Common Short Sequence Compression instructions

FEAT_CSSC introduces a set of instructions for optimization of short instruction sequences using general-purpose registers.

This feature is supported in AArch64 state only.

FEAT_CSSC is OPTIONAL from Armv8.7.

FEAT_CSSC is mandatory from Armv8.9.

The following field identifies the presence of FEAT_CSSC:

- ID_AA64ISAR2_EL1.CSSC.

For more information, see:

- Integer minimum and maximum (immediate).
- Integer maximum and minimum (register).
- Absolute value.
- Bit operation.

FEAT_Debugv8p9, Debug v8.9

FEAT_Debugv8p9 introduces all of the following:

- The ability to implement more than 16 breakpoints.
- The ability to implement more than 16 watchpoints.
- DBGBCR<n>_EL1 and DBGWCR<n>_EL1 are 64-bit registers in the external debug interface.
- DSPSR2 is added to extend DSPSR for holding the saved process state for Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in both AArch64 and AArch32 states.

FEAT_Debugv8p9 is OPTIONAL from Armv8.8.

FEAT_Debugv8p9 is mandatory from Armv8.9.

If FEAT_Debugv8p9 is implemented, then FEAT_Debugv8p8 is implemented.

When FEAT_Debugv8p9 and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_Debugv8p9:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- ID_DFR0.CopDbg.
- DBGDIDR.Version.
- EDDEVARCH.ARCHVER.
- EDDEVID1.HSR.

For more information, see:

- About Breakpoint exceptions.
- About Watchpoint exceptions.

FEAT_DoubleFault2, Double Fault Extension v2

FEAT_DoubleFault2 provides additional controls for routing and masking error exceptions.

This feature is supported in AArch64 state only.

FEAT_DoubleFault2 is OPTIONAL from Armv8.8.

If FEAT_DoubleFault2 is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_AA64 and FEAT_DoubleFault2 are implemented, FEAT_AA64EL1 is implemented.

When FEAT_DoubleFault2 and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

If FEAT_DoubleFault2 is implemented, then FEAT_DoubleFault is implemented.

The following field identifies the presence of FEAT_DoubleFault2:

- ID_AA64PFR1_EL1.DF2.

For more information, see:

- Synchronous exception types.
- Asynchronous exception types.
- Error synchronization event.

FEAT_ECBHB, Exploitative control using branch history information

FEAT_ECBHB imposes restrictions on branch history speculation around exceptions.

This feature is supported in AArch64 state only.

FEAT_ECBHB is OPTIONAL from Armv8.0.

FEAT_ECBHB is mandatory from Armv8.9.

The following field identifies the presence of FEAT_ECBHB:

- ID_AA64MMFR1_EL1.ECBHB.

For more information, see Branch prediction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_EDHSR, Support for EDHSR

FEAT_EDHSR introduces the EDHSR, which holds syndrome information of a Debug event.

FEAT_EDHSR is OPTIONAL.

If FEAT_Debugv8p9 is implemented, then FEAT_EDHSR is implemented.

If FEAT_EDHSR is implemented, then FEAT_Debugv8p2 is implemented.

The following field identifies the presence of FEAT_EDHSR:

- EDDEVID1.HSR.

FEAT_FGT2, Fine-grained traps 2

FEAT_FGT2 introduces the hypervisor registers HFGITR2_EL2, HFGRTR2_EL2, HFGWTR_EL2, HDFGRTR2_EL2, and HDFGWTR2_EL2. These registers are extensions of the corresponding FGT registers.

This feature is supported in AArch64, and when EL1 is using AArch64, EL0 accesses using AArch32 are also trapped.

FEAT_FGT2 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_AA64EL2 is implemented, FEAT_FGT2 is implemented.

If FEAT_FGT2 is implemented, then FEAT_FGT is implemented.

The following field identifies the presence of FEAT_FGT2:

- ID_AA64MMFR0_EL1.FGT.

FEAT_HAFT, Hardware managed Access Flag for Table descriptors

FEAT_HAFT introduces the support for hardware management of the Table descriptor Access flag.

This feature is supported in AArch64 state only.

FEAT_HAFT is OPTIONAL from Armv8.7.

If FEAT_HAFT is implemented, then FEAT_HAFDBS is implemented.

If FEAT_HAFT is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_HAFT:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see Hardware management of the Table descriptor Access Flag.

FEAT_LRCPC3, Load-Acquire RCpc instructions version 3

FEAT_LRCPC3 introduces variants of load/store pair and load/store single register instructions, with release consistency, to optimize additional use cases where ordering is required. FEAT_LRCPC3 also introduces a set of additional load/store instructions with release consistency ordering in the Advanced SIMD and floating-point instruction set.

This feature is supported in AArch64 state only.

FEAT_LRCPC3 is OPTIONAL from Armv8.2.

If FEAT_LRCPC3 is implemented, then FEAT_LRCPC2 is implemented.

The following field identifies the presence of FEAT_LRCPC3:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Changes to single-copy atomicity in Armv8.4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Load-Acquire/Store-Release.
- A64 instructions that are changed in Debug state.

FEAT_MTE4, Enhanced Memory Tagging Extension

FEAT_MTE4 introduces support for the following sub-features:

- Canonical tag checking, identified as FEAT_MTE_CANONICAL_TAGS.
- Reporting of all non-address bits on a fault, identified as FEAT_MTE_TAGGED_FAR.
- Store-only Tag checking, identified as FEAT_MTE_STORE_ONLY.
- Memory tagging with Address tagging disabled, identified as FEAT_MTE_NO_ADDRESS_TAGS.

This feature is supported in AArch64 state only.

FEAT_MTE4 is OPTIONAL from Armv8.7.

In an Armv8.9 implementation, if FEAT_MTE2 is implemented, FEAT_MTE4 is implemented.

If FEAT_MTE4 is implemented, then FEAT_MTE2 is implemented.

If FEAT_MTE4 is implemented, then FEAT_MTE_PERM is implemented.

If FEAT_MTE4 is implemented, then FEAT_MTE_CANONICAL_TAGS, FEAT_MTE_NO_ADDRESS_TAGS, FEAT_MTE_TAGGED_FAR, and FEAT_MTE_STORE_ONLY are implemented.

FEAT_MTE_ASYNC, Asynchronous reporting of Tag Check Fault

FEAT_MTE_ASYNC provides support for asynchronously accumulating Tag Check Faults into the TFSRE0_EL1 or TFSR_ELx registers. A PE that is compliant with FEAT_MTE2 is compliant with the behavior defined for this feature.

This feature is supported in AArch64 state only.

FEAT_MTE_ASYNC is OPTIONAL from Armv8.5.

If FEAT_MTE_ASYNC is implemented, then FEAT_MTE2 is implemented.

The following field identifies the presence of FEAT_MTE_ASYNC:

- ID_AA64PFR1_EL1.MTE_frac.

For more information, see:

- The Memory Tagging Extension.
- The AArch64 Application Level Memory Model.
- PMU Event Descriptions.
- The Statistical Profiling Extension.
- Debug State.

FEAT_MTE_CANONICAL_TAGS, Canonical Tag checking for Untagged memory

FEAT_MTE_CANONICAL_TAGS introduces support for MTE canonical tag checking.

This feature is supported in AArch64 state only.

FEAT_MTE_CANONICAL_TAGS is OPTIONAL.

If FEAT_MTE_CANONICAL_TAGS is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_CANONICAL_TAGS:

- ID_AA64PFR1_EL1.MTEX.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_MTE_NO_ADDRESS_TAGS, Memory tagging with Address tagging disabled

FEAT_MTE_NO_ADDRESS_TAG introduces support for MTE tagging with Address tagging disabled.

This feature is supported in AArch64 state only.

FEAT_MTE_NO_ADDRESS_TAGS is OPTIONAL.

If FEAT_MTE_NO_ADDRESS_TAGS is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_NO_ADDRESS_TAGS:

- ID_AA64PFR1_EL1.MTEX.

FEAT_MTE_PERM, Allocation tag access permission

FEAT_MTE_PERM introduces support for the Stage 2 NoTagAccess memory attribute.

This feature is supported in AArch64 state only.

FEAT_MTE_PERM is OPTIONAL from Armv8.7.

In an Armv8.9 implementation, if FEAT_MTE2 is implemented, FEAT_MTE_PERM is implemented.

If FEAT_MTE_PERM is implemented, then FEAT_MTE2 is implemented.

The following field identifies the presence of FEAT_MTE_PERM:

- ID_AA64PFR2_EL1.MTEPERM.

FEAT_MTE_STORE_ONLY, Store-only Tag Checking

This feature is supported in AArch64 state only.

FEAT_MTE_STORE_ONLY is OPTIONAL.

If FEAT_MTE_STORE_ONLY is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_STORE_ONLY:

- ID_AA64PFR2_EL1.MTESTOREONLY.

FEAT_MTE_TAGGED_FAR, FAR_ELx on a Tag Check Fault

FEAT_MTE_TAGGED_FAR introduces support for reporting all non-address bits on a synchronous MTE tag check fault.

This feature is supported in AArch64 state only.

FEAT_MTE_TAGGED_FAR is OPTIONAL.

If FEAT_MTE_TAGGED_FAR is implemented, then FEAT_MTE4 is implemented.

The following field identifies the presence of FEAT_MTE_TAGGED_FAR:

- ID_AA64PFR2_EL1.MTEFAR.

FEAT_PCSRv8p9, Armv8.9 PC Sample-based Profiling Extension

FEAT_PCSRv8p9 introduces a mechanism to suspend PC Sample-based Profiling.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PCSRv8p9 is OPTIONAL from Armv8.8.

If FEAT_PCSRv8p9 is implemented, then FEAT_PCSRv8p2 is implemented.

The following field identifies the presence of FEAT_PCSRv8p9:

- PMDEVID.PCSample.

For more information, see Suspending and activating PC Sample-based Profiling.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PFAR, Physical Fault Address Register Extension

FEAT_PFAR introduces the Physical Fault Address Registers, PFAR_ELx, that record the faulting physical address for a synchronous External abort or SError exception.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PFAR is OPTIONAL from Armv8.8.

When FEAT_AA64 and FEAT_PFAR are implemented, FEAT_AA64EL1 is implemented.

When FEAT_PFAR and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_PFAR:

- ID_AA64PFR1_EL1.PFAR.

FEAT_PMUv3_EDGE, PMU event edge detection

FEAT_PMUv3_EDGE introduces edge-detection logic to support counting threshold crossing events.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3_EDGE is OPTIONAL from Armv8.8.

If FEAT_PMUv3_EDGE is implemented, then FEAT_PMUv3_TH is implemented.

The following fields identify the presence of FEAT_PMUv3_EDGE:

- PMMIR_EL1.EDGE.
- PMMIR.EDGE.
- PMMIR.EDGE.

For more information, see Event threshold and edge counting.

FEAT_PMUv3_ICNTR, Fixed-function instruction counter

FEAT_PMUv3_ICNTR introduces a fixed-function instruction counter to the PMU.

This feature is supported in both AArch64 and AArch32 states. The counter is not accessible from AArch32 state.

FEAT_PMUv3_ICNTR is OPTIONAL from Armv8.8.

If FEAT_PMUv3_ICNTR is implemented, then FEAT_PMUv3p9 is implemented.

When FEAT_PMUv3_ICNTR and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_PMUv3_ICNTR:

- ID_AA64DFR1_EL1.PMICNTR.
- PMCFGR.NCG.

For more information, see About the Performance Monitors.

FEAT_PMUv3_SS, PMU Snapshot extension

FEAT_PMUv3_SS defines an IMPLEMENTATION DEFINED Snapshot Extension, compatible with the CoreSight PMU Snapshot Extension.

This feature is supported in both AArch64 and AArch32 states. The PMU snapshot registers are not accessible from AArch32 state.

FEAT_PMUv3_SS is OPTIONAL from Armv8.8.

If FEAT_PMUv3_SS is implemented, then FEAT_PMUv3p9 is implemented.

When FEAT_PMUv3_SS and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

If FEAT_PMUv3_SS is implemented, then FEAT_AA32EL1 is not implemented.

The following fields identify the presence of FEAT_PMUv3_SS:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64DFR0_EL1.PMSS.
- PMDEVID.PMSS.

For more information, see PMU snapshots.

FEAT_PMUv3p9, Armv8.9 PMU extensions

FEAT_PMUv3p9 introduces the following to the Performance Monitors Extension:

- Provides finer-grained control over allocation of PMU event counters to an EL0 process.
- Allows an arbitrary combination of event counters and fixed-function counters to be zeroed.
- Provides controls to configure the PMU to directly request the PE enters Debug state without using the CTI.
- Updates PMU event definitions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3p9 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p9 is implemented.

If FEAT_PMUv3p9 is implemented, then FEAT_PMUv3p8 is implemented.

When FEAT_PMUv3p9 and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_PMUv3p9:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- PMU Overflow external debug request.
- EL0 access controls.
- PMU Event Descriptions.
- Resetting counters.

FEAT_PRFMSLC, SLC target support for PRFM and PRFUM instructions

FEAT_PRFMSLC introduces a system level cache option for the PRFM and PRFUM instructions.

This feature is supported in AArch64 state only.

FEAT_PRFMSLC is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_PRFMSLC:

- ID_AA64ISAR2_EL1.PRFMSLC.

FEAT_RASv2, RAS Extension v2

FEAT_RASv2 introduces the following to the Reliability, Availability, and Serviceability Extension:

- The features defined by FEAT_RASSAv2 in the System register error records.
- The error group status register, ERXGSR_EL1.
- A control to trap writes to RAS error record System registers to EL3.
- An additional syndrome to ESR_ELx for error exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in both AArch64 and AArch32 states.

FEAT_RASv2 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_RAS is implemented, FEAT_RASv2 is implemented.

When FEAT_RASv2 and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

If FEAT_RASv2 is implemented, then FEAT_RASv1p1 is implemented.

If FEAT_RASv2 is implemented, then FEAT_RASSAv2 is implemented.

The following fields identify the presence of FEAT_RASv2:

- ID_AA64PFR0_EL1.RAS.
- ID_PFR0_EL1.RAS.
- ID_PFR0.RAS.

For more information, see:

- RAS PE architecture.
- *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_RPRFM, Support for Range Prefetch Memory instruction

FEAT_RPRFM introduces the Range Prefetch Memory hint instruction, RPRFM which specifies the range of addresses that are likely to be accessed in the near future and their expected reuse.

This feature is supported in AArch64 state only.

FEAT_RPRFM is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_RPRFM:

- ID_AA64ISAR2_EL1.RPRFM.

FEAT_S1PIE, Stage 1 permission indirections

FEAT_S1PIE introduces a way to set stage 1 permissions that allows more efficient use of the permission bits in translation table descriptors and provides the ability to introduce additional permission types.

This feature is supported in AArch64 state only.

FEAT_S1PIE is OPTIONAL from Armv8.8.

If FEAT_S1PIE is implemented, then FEAT_ATS1A is implemented.

If FEAT_S1PIE is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_S1PIE:

- ID_AA64MMFR3_EL1.S1PIE.

For more information, see Stage 1 Indirect permissions.

FEAT_S1POE, Stage 1 permission overlays

FEAT_S1POE allows stage 1 permissions to be progressively restricted without requiring TLB maintenance. The feature is available at all Exception levels, including EL0, and can reduce the number of calls to privileged software.

This feature is supported in AArch64 state only.

FEAT_S1POE is OPTIONAL from Armv8.8.

If FEAT_S1POE is implemented, then FEAT_TCR2 is implemented.

If FEAT_S1POE is implemented, then FEAT_ATS1A is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_S1POE is implemented, then FEAT_HPDS is implemented.

The following field identifies the presence of FEAT_S1POE:

- ID_AA64MMFR3_EL1.S1POE.

For more information, see Stage 1 Overlay permissions.

FEAT_S2PIE, Stage 2 permission indirections

FEAT_S2PIE introduces all of the following:

- A method to set stage 2 permissions that allows more efficient use of the permission bits in translation table descriptors and provides the ability to introduce additional permission types.
- The Mostly Read Only (MRO) permission in stage 2 translations.

This feature is supported in AArch64 state only.

FEAT_S2PIE is OPTIONAL from Armv8.8.

If FEAT_S2PIE is implemented, then FEAT_EL2 is implemented.

The following field identifies the presence of FEAT_S2PIE:

- ID_AA64MMFR3_EL1.S2PIE.

For more information, see Stage 2 Indirect permissions.

FEAT_S2POE, Stage 2 permission overlays

FEAT_S2POE provides a mechanism for stage 2 permissions to be progressively restricted, such that different EL1&0 contexts that are using the same stage 2 translation tables can be provided with different sets of permissions.

This feature is supported in AArch64 state only.

FEAT_S2POE is OPTIONAL from Armv8.8.

If FEAT_S2POE is implemented, then FEAT_EL2 is implemented.

If FEAT_S2POE is implemented, then FEAT_S2PIE is implemented.

The following field identifies the presence of FEAT_S2POE:

- ID_AA64MMFR3_EL1.S2POE.

For more information, see Stage 2 Overlay permissions.

FEAT_SPECRES2, Enhanced speculation restriction instructions

FEAT_SPECRES2 introduces a speculation restriction instruction, Clear Other Speculative Prediction Restriction by Context (COSP), to the instructions that are part of FEAT_SPECRES.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPECRES2 is OPTIONAL from Armv8.0.

FEAT_SPECRES2 is mandatory from Armv8.9.

The following fields identify the presence of FEAT_SPECRES2:

- ID_AA64ISAR1_EL1.SPECRES.
- ID_ISAR6_EL1.SPECRES.
- ID_ISAR6.SPECRES.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Prediction restriction instructions.
- Execution, data prediction and prefetching restriction System instructions.
- Execution and data prediction restriction System instructions.

FEAT_SPE_CRR, Statistical Profiling call return branch records

FEAT_SPE_CRR extends the Operation Type packet to provide more information whether the branch is a procedure call or a procedure return.

This feature is supported in AArch64 state only.

FEAT_SPE_CRR is OPTIONAL.

In an Armv8.9 implementation, if FEAT_SPE is implemented, FEAT_SPE_CRR is implemented.

If FEAT_SPE_CRR is implemented, then FEAT_SPEv1p4 is implemented.

If FEAT_SPE and FEAT_GCS are implemented, then FEAT_SPE_CRR is implemented.

The following field identifies the presence of FEAT_SPE_CRR:

- PMSIDR_EL1.CRR.

FEAT_SPE_DPFZS, Disable Cycle Counter on SPE Freeze

FEAT_SPE_DPFZS introduces controls to disable cycle counting when event counting is frozen on a Statistical Profiling Buffer Management event.

This feature is supported in AArch64 state only.

FEAT_SPE_DPFZS is OPTIONAL from Armv8.6.

If FEAT_PMUv3p9 and FEAT_SPEv1p4 are implemented, then FEAT_SPE_DPFZS is implemented.

If FEAT_SPE_DPFZS is implemented, then FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented.

The following field identifies the presence of FEAT_SPE_DPFZS:

- ID_AA64DFR1_EL1.DPFZS.

FEAT_SPE_FDS, Statistical Profiling data source filtering

FEAT_SPE_FDS provides the capability to filter sample records by all or part of the sampled Data Source packet.

This feature is supported in AArch64 state only.

FEAT_SPE_FDS is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_SPE_LDS is implemented, FEAT_SPE_FDS is implemented.

If FEAT_SPE_FDS is implemented, then FEAT_SPE_LDS is implemented.

If FEAT_SPE_FDS is implemented, then FEAT_SPEv1p4 is implemented.

When FEAT_SPE_FDS and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_SPE_FDS:

- PMSIDR_EL1.FDS.

For more information, see Filtering sample records.

FEAT_SPEv1p4, Statistical Profiling Extension version 1.4

FEAT_SPEv1p4 introduces the following to the Statistical Profiling Extension:

- Additions to the Events packet to provide more information about the data source.
- Additional event-based record filtering control bits in PMSEVFR_EL1 and PMSNEVFR_EL1 that add the ability to filter by additional existing bits in the Events packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_SPEv1p4 is OPTIONAL from Armv8.8.

In an Armv8.9 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p4 is implemented.

If FEAT_SPEv1p4 is implemented, then FEAT_SPEv1p3 is implemented.

The following field identifies the presence of FEAT_SPEv1p4:

- ID_AA64DFR0_EL1.PMSVer.

FEAT_SPMU, System Performance Monitors Extension

FEAT_SPMU provides a framework of architectural System registers and behaviors for System PMUs, which are PMUs other than the Performance Monitors Extension PMU, that are accessible by the PE.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPMU is OPTIONAL from Armv8.8.

When FEAT_SPMU and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_SPMU:

- ID_AA64DFR1_EL1.SPMU.

For more information, see System Performance Monitors Extension.

FEAT_THE, Translation Hardening Extension

FEAT_THE provides a mechanism to prevent modification of an arbitrary subset of translation table entries from within the Exception level that owns the translation tables, and introduces the following:

- The stage 1 Assured Translation property and the stage 2 AssuredOnly property.
- Stage 2 TopLevel checks.
- Read-Check-Write instructions, RCW* and RCWS*.
- Read-Check-Write mask registers, RCWMASK_EL1 and RCWSMASK_EL1.
- The Protected attribute in stage 1 descriptors.

This feature is supported in AArch64 state only.

FEAT_THE is OPTIONAL from Armv8.8.

When FEAT_THE and FEAT_AA64EL2 are implemented, FEAT_S2PIE is implemented.

When FEAT_THE and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

If FEAT_THE is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_THE:

- ID_AA64PFR1_EL1.THE.

For more information, see:

- Translation Hardening Extension.
- Read-Check-Write.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3 Armv9-A architecture extensions

The AArch32 state might optionally be implemented at EL0. The AArch32 state is not implemented at EL1, EL2, and EL3.

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

An implementation of the Armv9-A architecture cannot include an ETM.

A2.3.1 The Armv9.0 architecture extension

The Armv9.0 architecture extension adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.0 compliant if all of the following apply:

- It is Armv8.5 compliant.
- It includes all of the Armv9.0 architectural features that are mandatory.

An Armv9.0 compliant implementation can additionally include:

- Armv9.0 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.1, subject only to those constraints that require that certain features be implemented together.

In an Armv9.0 implementation, the AArch32 state might optionally be implemented at EL0. The AArch32 state is not implemented at EL1, EL2, and EL3.

An implementation of Armv9.0 cannot implement FEAT_DoubleLock.

An implementation of the Armv9.0 architecture cannot include an ETM.

In an Armv9.0 implementation, if FEAT_FP and FEAT_AdvSIMD are implemented, the following features are implemented:

- FEAT_RDM.
- FEAT_FP16.
- FEAT_DotProd.
- FEAT_FHM.
- FEAT_FCMA.
- FEAT_JSCVT.
- FEAT_FlagM2.
- FEAT_FRINTTS.

FEAT_Armv9_Crypto, Armv9 Cryptographic Extension

The Armv9 Cryptographic Extension provides instructions for the acceleration of encryption and decryption. The presence of the Cryptographic Extension in an implementation is subject to export license controls.

This feature is supported in AArch64 state only.

FEAT_Armv9_Crypto is OPTIONAL.

If FEAT_Armv9_Crypto is implemented, then FEAT_Crypto is implemented.

If FEAT_Armv9_Crypto is implemented, then FEAT_PMULL, FEAT_AES, FEAT_SHA1, FEAT_SHA3, FEAT_SHA256, and FEAT_SHA512 are implemented.

When FEAT_Armv9_Crypto, FEAT_SVE and FEAT_SVE_SHA3 are implemented, FEAT_SVE_AES, and FEAT_SVE_PMULL128 are implemented.

When FEAT_Armv9_Crypto, FEAT_SVE and FEAT_SVE_AES are implemented, FEAT_SVE_PMULL128, and FEAT_SVE_SHA3 are implemented.

When FEAT_Armv9_Crypto, FEAT_SVE and FEAT_SVE_PMULL128 are implemented, FEAT_SVE_AES, and FEAT_SVE_SHA3 are implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- The Cryptographic Extension in AArch64 state.
- The Cryptographic Extension in AArch32 state.

FEAT_ETE, Embedded Trace Extension

FEAT_ETE provides a trace unit that records details about software control flow running on a PE, which can be used to aid debugging or optimizing. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETE is OPTIONAL from Armv9.0.

If FEAT_ETE is implemented, then FEAT_TRBE is implemented.

If FEAT_ETE is implemented, then FEAT_TRF is implemented.

If FEAT_ETE is implemented, then FEAT_TRC_SR is implemented.

If FEAT_ETE is implemented, then FEAT_ETMv4 is not implemented.

When FEAT_ETE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

For more information, see The Embedded Trace Extension.

FEAT_SVE2, Scalable Vector Extension version 2

The *Scalable Vector Extension version 2* (SVE2) is a superset of SVE that incorporates functionality similar to Advanced SIMD, and other enhancements. In this Manual, unless stated otherwise, when SVE is used, the behavior also applies to SVE2. All Armv9-A systems that support standard operating systems with rich application environments also provide hardware support for SVE2 instructions.

This feature is supported in AArch64 state only.

FEAT_SVE2 is OPTIONAL from Armv9.0.

If FEAT_SVE2 is implemented, then FEAT_SVE is implemented.

The following field identifies the presence of FEAT_SVE2:

- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- FEAT_SVE.
- Data processing - SVE2.

FEAT_SVE_AES, Scalable Vector AES instructions

FEAT_SVE_AES provides the following scalable vector AES cryptographic instructions:

- AESD.
- AESE.
- AESIMC.
- AESMC.
- PMULLB.
- PMULLT.

This feature is supported in AArch64 state only.

FEAT_SVE_AES is OPTIONAL from Armv9.0.

If FEAT_SVE_AES is implemented, then FEAT_SVE2 or FEAT_SSVE_AES is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SVE_AES is implemented, then FEAT_SVE_PMULL128 is implemented.

If FEAT_SVE_PMULL128 is implemented, then FEAT_SVE_AES is implemented.

If FEAT_SVE_AES is implemented, then FEAT_Armv9_Crypto is implemented.

The following field identifies the presence of FEAT_SVE_AES:

- ID_AA64ZFR0_EL1.AES.

FEAT_SVE_BitPerm, Scalable Vector Bit Permutes instructions

FEAT_SVE_BitPerm provides the following scalable vector bit permute instructions:

- BEXT.
- BDEP.
- BGRP.

This feature is supported in AArch64 state only.

FEAT_SVE_BitPerm is OPTIONAL from Armv9.0.

If FEAT_SVE_BitPerm is implemented, then FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_SVE_BitPerm:

- ID_AA64ZFR0_EL1.BitPerm.

For more information, see:

- Bit permutation.

FEAT_SVE_PMULL128, SVE single-vector Advanced Encryption Standard and 128-bit polynomial multiply long instructions

FEAT_SVE_PMULL128 implements the SVE single-vector Advanced Encryption Standard and 128-bit destination element variants of PMULLB and PMULLT instructions, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SVE_PMULL128 is OPTIONAL from Armv9.0.

If FEAT_SVE_PMULL128 is implemented, then FEAT_SVE2 or FEAT_SSVE_AES is implemented.

If FEAT_SVE_PMULL128 is implemented, then FEAT_SVE_AES is implemented.

The following field identifies the presence of FEAT_SVE_PMULL128:

- ID_AA64ZFR0_EL1.AES.

FEAT_SVE_SHA3, Scalable Vector SHA3 instructions

FEAT_SVE_SHA3 provides the following scalable vector SHA3 instruction:

- RAX1.

This feature is supported in AArch64 state only.

FEAT_SVE_SHA3 is OPTIONAL from Armv9.0.

If FEAT_SVE_SHA3 is implemented, then FEAT_SVE2 or FEAT_SME2p1 is implemented.

If FEAT_SVE_SHA3 is implemented, then FEAT_Armv9_Crypto is implemented.

The following field identifies the presence of FEAT_SVE_SHA3:

- ID_AA64ZFR0_EL1.SHA3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SVE_SM4, Scalable Vector SM4 instructions

FEAT_SVE_SM4 provides the following scalable vector SM4 instructions:

- SM4E.
- SM4EKEY.

This feature is supported in AArch64 state only.

FEAT_SVE_SM4 is OPTIONAL from Armv9.0.

If FEAT_SVE_SM4 is implemented, then FEAT_SVE2 is implemented.

If FEAT_SVE_SM4 is implemented, then FEAT_SM4 is implemented.

The following field identifies the presence of FEAT_SVE_SM4:

- ID_AA64ZFR0_EL1.SM4.

FEAT_TRBE, Trace Buffer Extension

FEAT_TRBE enables support for a Trace Buffer Unit within a PE. When the Trace Buffer Unit is enabled, program-flow trace generated by a trace unit is written directly to memory by the Trace Buffer Unit, rather than routing trace data to a trace sink.

This feature is supported in AArch64 state, and collects trace in both AArch64 and AArch32 states.

FEAT_TRBE is OPTIONAL from Armv9.0.

If FEAT_TRBE is implemented, then FEAT_TRF is implemented.

If FEAT_TRBE is implemented, then FEAT_ETE is implemented.

When FEAT_TRBE and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following field identifies the presence of FEAT_TRBE:

- ID_AA64DFR0_EL1.TraceBuffer.

For more information, see The Trace Buffer Extension.

A2.3.1.1 Features added to the Armv9.0 extension in later releases

- FEAT_IDTE3.
- FEAT_PCDPHINT.
- FEAT_UINJ.

A2.3.2 The Armv9.1 architecture extension

The Armv9.1 architecture extension is an extension to Armv9.0. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.1 compliant if all of the following apply:

- It is Armv8.6 compliant.
- It is Armv9.0 compliant.
- It includes all of the Armv9.1 architectural features that are mandatory.

An Armv9.1 compliant implementation can additionally include:

- Armv9.1 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.2, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ETEv1p1, Embedded Trace Extension

FEAT_ETEv1p1 extends FEAT_ETE to provide more flexibility for tracing Timestamp values.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETEv1p1 is OPTIONAL from Armv9.0.

If FEAT_ETEv1p1 is implemented, then FEAT_ETE is implemented.

The following fields identify the presence of FEAT_ETEv1p1:

- TRCDEVARCH.REVISION.
- TRCDEVARCH.REVISION.

For more information, see The Embedded Trace Extension..

A2.3.3 The Armv9.2 architecture extension

The Armv9.2 architecture extension is an extension to Armv9.1. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.2 compliant if all of the following apply:

- It is Armv8.7 compliant.
- It is Armv9.1 compliant.
- It includes all of the Armv9.2 architectural features that are mandatory.

An Armv9.2 compliant implementation can additionally include:

- Armv9.2 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.3, subject only to those constraints that require that certain features be implemented together.

FEAT_AMU_EXTACR, Activity Monitors External Control Register

FEAT_AMU_EXTACR implements a control register to enable access to the Activity Monitors External registers.

- If FEAT_RME is implemented, the register AMROOTCR is implemented to control Root, Realm, Secure and Non-secure accesses to the external AMU registers.
- If FEAT_RME is not implemented, the register AMSCR is implemented to control Secure and Non-secure accesses to the external AMU registers.

FEAT_AMU_EXTACR is OPTIONAL from Armv8.4.

If FEAT_AMU_EXTACR is implemented, then FEAT_AMU_EXT is implemented.

For more information, see The Activity Monitors Extension.

FEAT_BRBE, Branch Record Buffer Extension

FEAT_BRBE provides a Branch record buffer for capturing control path history.

This feature is supported in AArch64 state only.

FEAT_BRBE is OPTIONAL from Armv9.1.

The following field identifies the presence of FEAT_BRBE:

- ID_AA64DFR0_EL1.BRBE.

For more information, see The Branch Record Buffer Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ETEv1p2, Embedded Trace Extension

FEAT_ETEv1p2 extends FEAT_ETE to support FEAT_RME.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETEv1p2 is OPTIONAL from Armv9.1.

If FEAT_ETEv1p2 is implemented, then FEAT_ETEv1p1 is implemented.

If FEAT_ETE and FEAT_RME are implemented, then FEAT_ETEv1p2 is implemented.

The following fields identify the presence of FEAT_ETEv1p2:

- TRCDEVARCH.REVISION.
- TRCDEVARCH.REVISION.

For more information, see The Embedded Trace Extension..

FEAT_RME, Realm Management Extension

The Realm Management Extension (RME) is an extension to the Armv9 A-profile architecture.

RME introduces all of the following features:

- Two additional Security states, Root and Realm.
- Two additional physical address spaces, Root and Realm.
- The ability to dynamically transition memory granules between physical address spaces.
- Granule Protection Check mechanism.

FEAT_RME is one component of the Arm Confidential Compute Architecture (Arm CCA). Together with the other components of the Arm CCA, RME enables support for dynamic, attestable, and trusted execution environments (Realms) to be run on an Arm PE.

This feature is supported in AArch64 state only.

FEAT_RME is OPTIONAL from Armv9.1.

If FEAT_RME is implemented, then FEAT_AA64EL3, FEAT_AA64EL2, and FEAT_RNG or FEAT_RNG_TRAP are implemented.

When FEAT_RME and FEAT_AES or FEAT_SHA1 are implemented, FEAT_PMULL, FEAT_AES, FEAT_SHA3, FEAT_SHA256, and FEAT_SHA512 are implemented.

When FEAT_RME, FEAT_SVE and FEAT_AES or FEAT_SHA1 are implemented, FEAT_SVE_PMULL128, and FEAT_SVE_SHA3 are implemented.

When FEAT_ETE and FEAT_RME are implemented, FEAT_ETEv1p2 is implemented.

When FEAT_PMUv3 and FEAT_RME are implemented, FEAT_PMUv3p7 is implemented.

When FEAT_SPE and FEAT_RME are implemented, FEAT_SPEv1p2 is implemented.

When FEAT_MPAM and FEAT_RME are implemented, FEAT_MPAMv1p1 is implemented.

If FEAT_RME is implemented, then FEAT_PCSRv8 is not implemented.

The following field identifies the presence of FEAT_RME:

- ID_AA64PFR0_EL1.RME.

If the Activity Monitors Extension is implemented, then Arm strongly recommends that a PE that implements FEAT_RME also implements FEAT_AMUv1p1.

Arm recommends that a PE that implements FEAT_RME also implements all of the following:

- FEAT_VMID16.
- FEAT_HAFDBS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- The AArch64 System Level Programmers´ Model.
- The AArch64 Virtual Memory System Architecture.
- The Granule Protection Check Mechanism.

FEAT_SME, Scalable Matrix Extension

FEAT_SME introduces two AArch64 execution modes that can be enabled and disabled by application software:

- In ZA storage enabled mode, scalable, two-dimensional, architectural ZA tile storage becomes available and instructions are defined to load, store, extract, insert, and clear rows and columns of the ZA tiles.
- In Streaming SVE mode, the Effective SVE vector length changes to match the Effective ZA tile width, support for a substantial subset of the SVE2 instruction set is available, and, when ZA mode is also enabled, instructions are defined that accumulate the matrix outer product of two SVE vectors into a ZA tile.

This feature is supported in AArch64 state only.

FEAT_SME is OPTIONAL from Armv9.2.

If FEAT_SME is implemented, then FEAT_FCMA, FEAT_FP16, FEAT_BF16, and FEAT_FHM are implemented.

When FEAT_SME and FEAT_EL2 are implemented, FEAT_FGT and FEAT_HCX are implemented.

When FEAT_SME and FEAT_PMUv3 are implemented, FEAT_PMUv3p1 is implemented.

The following fields identify the presence of FEAT_SME:

- ID_AA64PFR1_EL1.SME.
- ID_AA64SMFR0_EL1.F16F32.
- ID_AA64SMFR0_EL1.B16F32.
- ID_AA64SMFR0_EL1.F32F32.
- ID_AA64SMFR0_EL1.I8I32.

If FEAT_SME is implemented, this does not imply that FEAT_SVE and FEAT_SVE2 are implemented when the PE is not in Streaming SVE mode.

For more information, see The Scalable Matrix Extension.

FEAT_SME_F64F64, Double-precision floating-point instructions

FEAT_SME_F64F64 indicates SME support for instructions that accumulate into double-precision floating-point elements in the ZA array.

This feature is supported in AArch64 state only.

FEAT_SME_F64F64 is OPTIONAL from Armv9.2.

If FEAT_SME_F64F64 is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_SME_F64F64:

- ID_AA64SMFR0_EL1.F64F64.

For more information, see The Scalable Matrix Extension.

FEAT_SME_FA64, Full A64 instruction set support in Streaming SVE mode

FEAT_SME_FA64 indicates support for execution of the full A64 instruction set in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SME_FA64 is OPTIONAL from Armv9.2.

If FEAT_SME_FA64 is implemented, then FEAT_SME is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SME_FA64 is implemented, then FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_SME_FA64:

- ID_AA64SMFR0_EL1.FA64.

For more information, see The Scalable Matrix Extension.

FEAT_SME_I16I64, 16-bit to 64-bit integer widening instructions

FEAT_SME_I16I64 indicates SME support for instructions that accumulate into 64-bit integer elements in the ZA array.

This feature is supported in AArch64 state only.

FEAT_SME_I16I64 is OPTIONAL from Armv9.2.

If FEAT_SME_I16I64 is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_SME_I16I64:

- ID_AA64SMFR0_EL1.I16I64.

For more information, see The Scalable Matrix Extension.

A2.3.3.1 Features added to the Armv9.2 extension in later releases

- FEAT_F8F16MM.
- FEAT_F8F32MM.
- FEAT_FAMINMAX.
- FEAT_FP8DOT2.
- FEAT_FP8DOT4.
- FEAT_FP8FMA.
- FEAT_FP8.
- FEAT_FPMR.
- FEAT_LS64WB.
- FEAT_LUT.
- FEAT_SME2p1.
- FEAT_SME_B16B16.
- FEAT_SME_F16F16.
- FEAT_SME_F8F16.
- FEAT_SME_F8F32.
- FEAT_SME_LUTv2.
- FEAT_SPE_SME.
- FEAT_SSVE_FP8DOT2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_SSVE_FP8DOT4.
- FEAT_SSVE_FP8FMA.
- FEAT_SVE2p1.
- FEAT_SVE_B16B16.
- FEAT_SVE_BFSCALE.
- FEAT_SVE_F16F32MM.

A2.3.4 The Armv9.3 architecture extension

The Armv9.3 architecture extension is an extension to Armv9.2. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.3 compliant if all of the following apply:

- It is Armv8.8 compliant.
- It is Armv9.2 compliant.
- It includes all of the Armv9.3 architectural features that are mandatory.

An Armv9.3 compliant implementation can additionally include:

- Armv9.3 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.4, subject only to those constraints that require that certain features be implemented together.

FEAT_BRBEv1p1, Branch Record Buffer Extension version 1.1

FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

This feature is supported in AArch64 state only.

FEAT_BRBEv1p1 is OPTIONAL from Armv9.2.

In an Armv9.3 implementation, if FEAT_BRBE is implemented, FEAT_BRBEv1p1 is implemented.

If FEAT_BRBEv1p1 is implemented, then FEAT_BRBE is implemented.

The following field identifies the presence of FEAT_BRBEv1p1:

- ID_AA64DFR0_EL1.BRBE.

For more information, see The Branch Record Buffer Extension.

FEAT_MEC, Memory Encryption Contexts

Memory Encryption Contexts (MEC) is an extension to the RME.

An existing RME enabled system uses a combination of isolation and external memory protection to guarantee privacy of the Realm Security state. Isolation between Realms is enforced by the Realm stage 2 translation tables.

FEAT_MEC introduces all of the following features:

- Memory encryption contexts are provided to all physical address spaces.
- Multiple memory encryption contexts are provided to the Realm physical address space for assignment to Realm virtual machines, with policy controlled by Realm EL2.
- The Non-secure, Secure, and Root, physical address spaces each have one encryption context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_MEC is OPTIONAL from Armv9.2.

If FEAT_MEC is implemented, then FEAT_RME is implemented.

If FEAT_MEC is implemented, then FEAT_SCTLR2 is implemented.

If FEAT_MEC is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_MEC:

- ID_AA64MMFR3_EL1.MEC.

FEAT_SME2, Scalable Matrix Extensions version 2

FEAT_SME2 is a superset of FEAT_SME that introduces the following:

- The ability to treat the SME ZA array as containing addressable groups of one-dimensional ZA array vectors, instead of two-dimensional ZA tiles.
- Multi-vector instructions that operate on groups of Z vector registers and ZA array vectors.
- A multi-vector predication mechanism for multi-vector load and store.
- A dedicated 512-bit lookup table register, ZT0, for data decompression.

This feature is supported in AArch64 state only.

FEAT_SME2 is OPTIONAL from Armv9.2.

If FEAT_SME2 is implemented, then FEAT_SME is implemented.

The following fields identify the presence of FEAT_SME2:

- ID_AA64SMFR0_EL1.SMEver.
- ID_AA64PFR1_EL1.SME.
- ID_AA64SMFR0_EL1.BI32I32.
- ID_AA64SMFR0_EL1.I16I32.

For more information, see The Scalable Matrix Extension.

A2.3.4.1 Features added to the Armv9.3 extension in later releases

- FEAT_LSFE.
- FEAT_MPAM_PE_BW_CTRL.
- FEAT_OCCMO.

A2.3.5 The Armv9.4 architecture extension

The Armv9.4 architecture extension is an extension to Armv9.3. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.4 compliant if all of the following apply:

- It is Armv8.9 compliant.
- It is Armv9.3 compliant.
- It includes all of the Armv9.4 architectural features that are mandatory.

An Armv9.4 compliant implementation can additionally include:

- Armv9.4 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.5, subject only to those constraints that require that certain features be implemented together.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ABLE, Address Breakpoint Linking Extension

FEAT_ABLE introduces the capability to link a watchpoint to an address matching breakpoint.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ABLE is OPTIONAL from Armv9.3.

If FEAT_ABLE is implemented, then FEAT_BWE is implemented.

If FEAT_ABLE is implemented, then FEAT_Debugv8p9 is implemented.

The following fields identify the presence of FEAT_ABLE:

- ID_AA64DFR1_EL1.ABLE.
- EDDFR1.ABLE.

For more information, see About Breakpoint exceptions.

FEAT_BWE, Breakpoint and watchpoint enhancements

FEAT_BWE introduces the capability to define an included-range-based breakpoint and an excluded-range-based breakpoint.

This feature is supported in AArch64 state only.

FEAT_BWE is OPTIONAL from Armv9.3.

The following fields identify the presence of FEAT_BWE:

- ID_AA64DFR2_EL1.BWE.
- ID_AA64DFR1_EL1.ABLE.
- EDDFR1.ABLE.
- EDDFR2.BWE.

For more information, see Other usage constraints for Address breakpoints.

FEAT_CHK, Check Feature Status

FEAT_CHK introduces the CHKFEAT instruction, which allows software to detect when certain features are enabled.

A PE that is compliant with architectures from Armv8.0 to Armv9.3 is compliant with the behavior defined for this feature.

This feature is supported in AArch64 state only.

FEAT_CHK is OPTIONAL from Armv8.0.

FEAT_CHK is mandatory from Armv9.4.

For more information, see Detecting when FEAT_GCS is enabled.

FEAT_D128, 128-bit Translation Tables, 56 bit PA

FEAT_D128 introduces support for the VMSAv9-128 translation system, comprising the following:

- 128-bit translation table descriptors.
- Support for encoding up to 56-bit physical addresses in translation table descriptors.
- If FEAT_LVA or FEAT_LVA3 are implemented, support for translating up to 56-bit virtual addresses.
- TLBIP VA*, TLBIP RVA*, TLBIP IPA*, TLBIP RIPA* instructions that can take 128-bit inputs.

This feature is supported in AArch64 state only.

FEAT_D128 is OPTIONAL from Armv9.3.

If FEAT_D128 is implemented, then FEAT_SYSREG128 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_D128 is implemented, then FEAT_SYSINSTR128 is implemented.

If FEAT_D128 is implemented, then FEAT_LSE128 is implemented.

If FEAT_D128 is implemented, then FEAT_S1PIE is implemented.

When FEAT_D128 and FEAT_EL2 are implemented, FEAT_S2PIE is implemented.

If FEAT_D128 is implemented, then FEAT_AIE is implemented.

If FEAT_D128 is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_D128:

- ID_AA64MMFR3_EL1.D128.

For more information, see The AArch64 Virtual Memory System Architecture.

FEAT_EBEP, Exception-based Event Profiling

FEAT_EBEP provides support for reporting PMU counter overflows as PMU Profiling exceptions. This allows generation of higher quality profiles by eliminating the interrupt latency and jitter incurred outside the PE.

This feature is supported in both AArch64 and AArch32 states.

FEAT_EBEP is OPTIONAL from Armv9.3.

In an Armv9.3 implementation, if FEAT_PMUv3p9 is implemented, FEAT_EBEP is implemented.

If FEAT_EBEP is implemented, then FEAT_PMUv3p8 is implemented.

When FEAT_EBEP and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

When FEAT_EBEP and FEAT_AA32EL0 are implemented, FEAT_Debugv8p9 is implemented.

The following field identifies the presence of FEAT_EBEP:

- ID_AA64DFR1_EL1.EBEP.

For more information, see Exception-based event profiling.

FEAT_ETEv1p3, Embedded Trace Extension version 1.3

FEAT_ETEv1p3 extends FEAT_ETE to support all of the following:

- The ETE External Debug Request, when FEAT_Debugv8p9 is implemented.
- The ETE Trace Output Enable, which is mandatory for FEAT_ETEv1p3 and OPTIONAL for FEAT_ETE.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

FEAT_ETEv1p3 is OPTIONAL from Armv9.3.

If FEAT_ETEv1p3 is implemented, then FEAT_ETEv1p2 is implemented.

The following fields identify the presence of FEAT_ETEv1p3:

- TRCDEVARCH.REVISION.
- TRCDEVARCH.REVISION.

For more information, see The Embedded Trace Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_GCS, Guarded Control Stack Extension

FEAT_GCS introduces support for a Guarded Control Stack, an area of memory in which procedure return addresses and exception return addresses are stored and protected from modification.

This feature is supported in AArch64 state only.

FEAT_GCS is OPTIONAL from Armv9.3.

If FEAT_GCS is implemented, then FEAT_CHK is implemented.

If FEAT_GCS is implemented, then FEAT_S1PIE is implemented.

The following field identifies the presence of FEAT_GCS:

- ID_AA64PFR1_EL1.GCS.

For more information, see Guarded Control Stack.

FEAT_ITE, Instrumentation Trace Extension

FEAT_ITE provides all of the following to allow software to inject instrumentation information into the ETE trace stream:

- The TRCIT instruction, that injects the value of a general purpose register into the ETE trace stream.
- Instrumentation Packet that contains the value written by the TRCIT instruction.
- Controls that define the behavior of the TRCIT instruction.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ITE is OPTIONAL from Armv9.3.

If FEAT_ITE is implemented, then FEAT_TRF is implemented.

If FEAT_ITE is implemented, then FEAT_ETE is implemented.

If FEAT_ITE is implemented, then FEAT_TRBE is implemented.

When FEAT_ITE and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_ITE:

- ID_AA64DFR1_EL1.ITE.
- TRCIDR0.ITE.
- TRCIDR0.ITE.

For more information, see:

- Instrumentation extension.
- Instrumentation element.

FEAT_LSE128, 128-bit Atomics

FEAT_LSE128 introduces support for 128-bit atomic instructions.

This feature is supported in AArch64 state only.

FEAT_LSE128 is OPTIONAL from Armv9.3.

If FEAT_LSE128 is implemented, then FEAT_LSE is implemented.

The following field identifies the presence of FEAT_LSE128:

- ID_AA64ISAR0_EL1.Atomic.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Atomic integer memory operations.
- Swap operations.

FEAT_LVA3, 56-bit VA

FEAT_LVA3 introduces support for 56-bit virtual addresses.

This feature is supported in AArch64 state only.

FEAT_LVA3 is OPTIONAL from Armv9.3.

If FEAT_LVA3 is implemented, then FEAT_D128 is implemented.

If FEAT_LVA3 is implemented, then FEAT_LVA is implemented.

The following field identifies the presence of FEAT_LVA3:

- ID_AA64MMFR2_EL1.VARange.

For more information, see Supported virtual address ranges.

FEAT_SME2p1, Scalable Matrix Extension version 2.1

The Scalable Matrix Extension version 2.1 (SME2.1) is a superset of SME2 that adds:

- Support for SVE RAX1 instructions in Streaming SVE mode if FEAT_SVE_SHA3 is implemented.
- SME MOVAZ instructions that move and zero ZA array vectors or ZA tile slices.
- SME ZERO instructions that zero ZA array vectors.
- SME LUTI2 and LUTI4 lookup table instructions to Z vectors with strided numbering.

In this Manual, unless stated otherwise, when SME is used, the behavior also applies to SME2.1.

This feature is supported in AArch64 state only.

FEAT_SME2p1 is OPTIONAL from Armv9.2.

In an Armv9.4 implementation, if FEAT_SME2 is implemented, FEAT_SME2p1 is implemented.

If FEAT_SME2p1 is implemented, then FEAT_SME2 is implemented.

If FEAT_SME and FEAT_SVE2p1 are implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SME2p1:

- ID_AA64SMFR0_EL1.SMEver.

For more information, see:

- Data processing - SME, SME2.
- The Scalable Matrix Extension.

FEAT_SME_B16B16, Non-widening BFloat16 to BFloat16 SME ZA-targeting arithmetic

FEAT_SME_B16B16 introduces SME ZA-targeting non-widening BFloat16 floating-point instructions to SME.

This feature is supported in AArch64 state only.

FEAT_SME_B16B16 is OPTIONAL from Armv9.2.

If FEAT_SME_B16B16 is implemented, then FEAT_SME2 is implemented.

If FEAT_SME_B16B16 is implemented, then FEAT_SVE_B16B16 is implemented.

The following field identifies the presence of FEAT_SME_B16B16:

- ID_AA64SMFR0_EL1.B16B16.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SME_F16F16, Non-widening half-precision FP16 to FP16 arithmetic for SME2

FEAT_SME_F16F16 introduces the SME2 half-precision to single-precision convert instructions and non-widening half-precision floating-point instructions.

This feature is supported in AArch64 state only.

FEAT_SME_F16F16 is OPTIONAL from Armv9.2.

If FEAT_SME_F16F16 is implemented, then FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_SME_F16F16:

- ID_AA64SMFR0_EL1.F16F16.

For more information, see:

- Data processing - SME, SME2.
- The Scalable Matrix Extension.

FEAT_SVE2p1, Scalable Vector Extensions version 2.1

The Scalable Vector Extension version 2.1 (SVE2.1) is a superset of SVE2 that adds:

- SVE instructions in Non-streaming SVE mode, which were previously added by SME in Streaming SVE mode. These are:
  - Contiguous multi-vector load and store instructions.
  - Predicate-as-counter instructions.
  - General-purpose SVE instructions.
- Other relaxations and enhancements.

In this Manual, unless stated otherwise, when SVE is used, the behavior also applies to SVE2.1.

This feature is supported in AArch64 state only.

FEAT_SVE2p1 is OPTIONAL from Armv9.2.

In an Armv9.4 implementation, if FEAT_SVE2 is implemented, FEAT_SVE2p1 is implemented.

If FEAT_SVE2p1 is implemented, then FEAT_SVE2 is implemented.

If FEAT_SVE2 and FEAT_SME2p1 are implemented, then FEAT_SVE2p1 is implemented.

The following field identifies the presence of FEAT_SVE2p1:

- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- FEAT_SVE.
- Loads and stores - SME, SME2, SVE2p1.
- Data processing - SVE2.

FEAT_SVE_B16B16, Non-widening BFloat16 to BFloat16 arithmetic for SVE2 and SME2

FEAT_SVE_B16B16 introduces the following:

- If FEAT_SVE2 is implemented, the SVE non-widening BFloat16 instructions when the PE is not in Streaming SVE mode.
- If FEAT_SME2 is implemented, the SVE non-widening BFloat16 instructions when the PE is in Streaming SVE mode, and the SME Z-targeting multi-vector non-widening BFloat16 instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This feature is supported in AArch64 state only.

FEAT_SVE_B16B16 is OPTIONAL from Armv9.2.

If FEAT_SVE_B16B16 is implemented, then FEAT_SME2 or FEAT_SVE2 is implemented.

In an Armv9.6 implementation, if FEAT_SVE_B16B16 is implemented, FEAT_SVE_BFSCALE is implemented.

The following field identifies the presence of FEAT_SVE_B16B16:

- ID_AA64ZFR0_EL1.B16B16.

For more information, see:

- BFloat16 arithmetic.
- BFloat16 minimum/maximum.
- Clamp to minimum/maximum.

FEAT_SYSINSTR128, 128-bit System instructions

FEAT_SYSINSTR128 introduces support for IMPLEMENTATION DEFINED System instructions that can take 128-bit inputs.

This feature is supported in AArch64 state only.

FEAT_SYSINSTR128 is OPTIONAL from Armv9.3.

If FEAT_SYSINSTR128 is implemented, then FEAT_SCTLR2 is implemented.

If FEAT_SYSINSTR128 is implemented, then FEAT_D128 is implemented.

The following field identifies the presence of FEAT_SYSINSTR128:

- ID_AA64ISAR2_EL1.SYSINSTR_128.

For more information, see The A64 System Instruction Class.

FEAT_SYSREG128, 128-bit System registers

FEAT_SYSREG128 introduces the following support for 128-bit System registers:

- The MRRS instruction to move a 128-bit System register into a pair of 64-bit general-purpose registers.
- The MSRR instruction to move a pair of 64-bit general-purpose registers to a 128-bit System register.
- 128-bit formats of the following system registers:
- The Physical Address Register, PAR_EL1.
- The Read-Check-Write mask registers, RCWMASK_EL1 and RCWSMASK_EL1.
- The following translation table base address registers, TTBR0_EL1, TTBR0_EL2, TTBR1_EL1, TTBR1_EL2, VTTBR_EL2.

This feature is supported in AArch64 state only.

FEAT_SYSREG128 is OPTIONAL from Armv9.3.

If FEAT_SYSREG128 is implemented, then FEAT_SCTLR2 is implemented.

If FEAT_SYSREG128 is implemented, then FEAT_D128 is implemented.

The following field identifies the presence of FEAT_SYSREG128:

- ID_AA64ISAR2_EL1.SYSREG_128.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TRBE_EXT, Trace Buffer external mode

FEAT_TRBE_EXT allows an external debugger, as well as a self-hosted debugger, to use the Trace Buffer Unit. All of the following registers are introduced to determine the parameters of the implementation:

- TRBDEVARCH.
- TRBDEVID.
- TRBDEVID1.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TRBE_EXT is OPTIONAL from Armv9.3.

If FEAT_TRBE_EXT is implemented, then FEAT_TRBE is implemented.

The following fields identify the presence of FEAT_TRBE_EXT:

- ID_AA64DFR0_EL1.ExtTrcBuff.
- EDDFR.ExtTrcBuff.
- EDDFR.TraceBuffer.

For more information, see Trace buffer External mode.

FEAT_TRBE_MPAM, Trace Buffer MPAM extensions

FEAT_TRBE_MPAM allows software to program the MPAM PARTID and PMG to use different MPAM values for trace data. TRBDEVID1.{PMG_MAX, PARTID_MAX} are used to determine the parameters of the MPAM implementation.

This feature is supported in both AArch64 and AArch32 states.

FEAT_TRBE_MPAM is OPTIONAL from Armv9.3.

If FEAT_TRBE_MPAM is implemented, then FEAT_TRBE_EXT is implemented.

If FEAT_TRBE_MPAM is implemented, then FEAT_MPAM is implemented.

When FEAT_TRBE_MPAM and FEAT_AA64EL2 are implemented, FEAT_FGT2 is implemented.

The following fields identify the presence of FEAT_TRBE_MPAM:

- TRBIDR_EL1.MPAM.
- TRBIDR_EL1.MPAM.

For more information, see External mode and FEAT_MPAM.

A2.3.5.1 Features added to the Armv9.4 extension in later releases

- FEAT_RME_GDI.
- FEAT_SME_MOP4.
- FEAT_SME_TMOP.
- FEAT_SSVE_BitPerm.
- FEAT_SSVE_FEXPA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3.6 The Armv9.5 architecture extension

The Armv9.5 architecture extension is an extension to Armv9.4. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.5 compliant if all of the following apply:

- It is Armv9.4 compliant.
- It includes all of the Armv9.5 architectural features that are mandatory.

An Armv9.5 compliant implementation can additionally include:

- Armv9.5 features that are optional.
- Any arbitrary subset of the architectural features of Armv9.6, subject only to those constraints that require that certain features be implemented together.

FEAT_ASID2, Support for concurrent use of two ASIDs

FEAT_ASID2 introduces support for use of one ASID for each TTBR_ELx.

This feature is supported in AArch64 state only.

FEAT_ASID2 is OPTIONAL from Armv9.4.

FEAT_ASID2 is mandatory from Armv9.5.

If FEAT_ASID2 is implemented, then FEAT_TCR2 is implemented.

The following field identifies the presence of FEAT_ASID2:

- ID_AA64MMFR4_EL1.ASID2.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

FEAT_BWE2, Breakpoint and watchpoint enhancements 2

FEAT_BWE2 provides the capability to generate a Watchpoint exception or debug event when software accesses an address outside of one or more user-supplied address regions.

This feature is supported in both AArch64 and AArch32 states.

FEAT_BWE2 is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_BWE is implemented, FEAT_BWE2 is implemented.

If FEAT_BWE2 is implemented, then FEAT_BWE is implemented.

The following fields identify the presence of FEAT_BWE2:

- ID_AA64DFR2_EL1.BWE.
- EDDFR2.BWE.

For more information, see:

- About Watchpoint exceptions.
- Exception syndrome information, fault address information, and preferred return address.
- Watchpoint data address comparisons.

FEAT_CPA, Instruction-only Checked Pointer Arithmetic

FEAT_CPA introduces support for Checked Pointer Arithmetic instructions.

This feature is supported in AArch64 state only.

FEAT_CPA is OPTIONAL from Armv9.4.

FEAT_CPA is mandatory from Armv9.5.

The following field identifies the presence of FEAT_CPA:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64ISAR3_EL1.CPA.

For more information, see Checked Pointer Arithmetic.

FEAT_CPA2, Checked Pointer Arithmetic

FEAT_CPA2 introduces support for enabling Checked Pointer Arithmetic. Checked Pointer Arithmetic detects and reports corruption of the following:

- The top 8 bits of a pointer.
- Multiplication overflows that lead to invalid array indices being calculated.

This feature is supported in AArch64 state only.

FEAT_CPA2 is OPTIONAL from Armv9.4.

If FEAT_CPA2 is implemented, then FEAT_CPA is implemented.

If FEAT_CPA2 is implemented, then FEAT_SCTLR2 is implemented.

The following field identifies the presence of FEAT_CPA2:

- ID_AA64ISAR3_EL1.CPA.

For more information, see Checked Pointer Arithmetic.

FEAT_E3DSE, Delegated SError exception injection

FEAT_E3DSE provides an SError injection mechanism for EL3 called delegated SErrors.

This feature is supported in AArch64 state only.

FEAT_E3DSE is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_AA64EL3 is implemented, FEAT_E3DSE is implemented.

The following field identifies the presence of FEAT_E3DSE:

- ID_AA64MMFR4_EL1.E3DSE.

For more information, see Asynchronous exception types.

FEAT_ETS3, Enhanced Translation Synchronization

FEAT_ETS3 introduces support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in both AArch64 and AArch32 states.

FEAT_ETS3 is OPTIONAL from Armv8.0.

FEAT_ETS3 is mandatory from Armv9.5.

If FEAT_ETS3 is implemented, then FEAT_ETS2 is implemented.

The following fields identify the presence of FEAT_ETS3:

- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
- ID_MMFR5.ETS.

For more information, see:

- Ordering requirements defined by the formal concurrency model.
- Ordering of memory accesses from translation table walks.
- Ordering of translation table walks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FAMINMAX, Floating-point maximum and minimum absolute value instructions

FEAT_FAMINMAX introduces the Advanced SIMD, SVE and SME `FAMAX` and `FAMIN` instructions that calculate floating-point maximum and minimum absolute value.

FEAT_FAMINMAX is OPTIONAL from Armv9.2.

In an Armv9.5 implementation, if FEAT_FP is implemented, FEAT_FAMINMAX is implemented.

If FEAT_FAMINMAX is implemented, then FEAT_AdvSIMD, FEAT_SVE2, or FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_FAMINMAX:

- ID_AA64ISAR3_EL1.FAMINMAX.

For more information, see:

- SIMD arithmetic.
- Floating-point minimum/maximum absolute value.
- Multi-vector minimum/maximum.

FEAT_FGWTE3, Fine-Grained Write Trap EL3

FEAT_FGWTE3 introduces traps for write accesses at EL3 to individual *_EL3 System registers. The traps are independent of existing controls. The bits in the control register FGWTE3_EL3 are sticky, and writes of 0 are ignored.

This feature is supported in AArch64 state only.

FEAT_FGWTE3 is OPTIONAL from Armv9.4.

If FEAT_FGWTE3 is implemented, then FEAT_EL3 is implemented.

The following field identifies the presence of FEAT_FGWTE3:

- ID_AA64MMFR4_EL1.FGWTE3.

For more information, see EL3 configurable instruction controls.

FEAT_FP8, FP8 convert instructions

FEAT_FP8 introduces the following:

- OFP8 formats E5M2 and E4M3 for FP8 instructions.
- Advanced SIMD: FP8 convert instructions and a floating point scaling `FSCALE` instruction.
- SME: FP8 convert instructions and floating point scaling `FSCALE` instructions.
- SVE: FP8 convert instructions.

FEAT_FP8 is OPTIONAL from Armv9.2.

If FEAT_FP8 is implemented, then FEAT_FPMR is implemented.

If FEAT_FP8 is implemented, then FEAT_FAMINMAX is implemented.

If FEAT_FP8 is implemented, then FEAT_LUT is implemented.

If FEAT_FP8 is implemented, then FEAT_BF16 is implemented.

If FEAT_FP8 is implemented, then FEAT_AdvSIMD, FEAT_SVE2, or FEAT_SME2 is implemented.

The following fields identify the presence of FEAT_FP8:

- ID_AA64FPFR0_EL1.F8CVT.
- ID_AA64FPFR0_EL1.F8E4M3.
- ID_AA64FPFR0_EL1.F8E5M2.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FP8 floating-point formats.
- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point conversion.
- For SVE, FP8 floating-point conversions.
- For SME, FP8 floating-point conversions.
- The *OCP 8-bit Floating Point Specification (OFP8)*.

FEAT_FP8DOT2, FP8 2-way dot product to half-precision instructions

FEAT_FP8DOT2 introduces the Advanced SIMD FP8 to half-precision 2-way dot product instructions.

If FEAT_SVE2 is implemented, FEAT_FP8DOT2 also implements the SVE versions of these instructions.

Note

This feature does not enable execution of the SVE instructions in Streaming SVE mode.

FEAT_FP8DOT2 is OPTIONAL from Armv9.2.

If FEAT_FP8DOT2 is implemented, then FEAT_FP8DOT4 is implemented.

If FEAT_FP8DOT2 is implemented, then FEAT_AdvSIMD or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_FP8DOT2:

- ID_AA64FPFR0_EL1.F8DP2.

For more information, see:

- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point dot product.
- For SVE, FP8 floating-point dot product.

FEAT_FP8DOT4, FP8 4-way dot product to single-precision instructions

FEAT_FP8DOT4 introduces the following:

- Advanced SIMD: FP8 to single-precision 4-way dot product instructions.
- SVE: SVE FP8 to single-precision 4-way dot product instructions.

Note

This feature does not enable execution of the SVE instructions in Streaming SVE mode.

FEAT_FP8DOT4 is OPTIONAL from Armv9.2.

If FEAT_FP8DOT4 is implemented, then FEAT_FP8FMA is implemented.

If FEAT_FP8DOT4 is implemented, then FEAT_AdvSIMD or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_FP8DOT4:

- ID_AA64FPFR0_EL1.F8DP4.

For more information, see:

- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point dot product.
- For SVE, FP8 floating-point dot product.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FP8FMA, FP8 multiply-accumulate to half-precision and single-precision instructions

FEAT_FP8FMA introduces the following:

- Advanced SIMD: FP8 to half-precision and single-precision multiply-accumulate instructions.
- SVE: FP8 to half-precision and single-precision multiply-accumulate instructions.

Note

This feature does not enable execution of the SVE instructions in Streaming SVE mode.

FEAT_FP8FMA is OPTIONAL from Armv9.2.

If FEAT_FP8FMA is implemented, then FEAT_FP8 is implemented.

If FEAT_FP8FMA is implemented, then FEAT_AdvSIMD or FEAT_SVE2 is implemented.

The following field identifies the presence of FEAT_FP8FMA:

- ID_AA64FPFR0_EL1.F8FMA.

For more information, see:

- Summary of FP8 instruction behaviors.
- For Advanced SIMD, FP8 floating-point widening multiply-accumulate.
- For SVE, FP8 floating-point widening multiply-accumulate.

FEAT_FPMR, Floating-point Mode Register

FEAT_FPMR introduces the Special-purpose register FPMR.

FEAT_FPMR is OPTIONAL from Armv9.2.

If FEAT_FPMR is implemented, then FEAT_AdvSIMD is implemented.

When FEAT_FPMR and FEAT_EL2 are implemented, FEAT_FGT is implemented.

When FEAT_FPMR and FEAT_EL2 are implemented, FEAT_HCX is implemented.

The following field identifies the presence of FEAT_FPMR:

- ID_AA64PFR2_EL1.FPMR.

For more information, see Summary of FP8 instruction behaviors.

FEAT_HACDBS, Hardware accelerator for cleaning Dirty state

FEAT_HACDBS introduces the Hardware accelerator for cleaning Dirty state, which enhances the process of updating stage 2 descriptors from writable-dirty to writable-clean.

This feature is supported in AArch64 state only.

FEAT_HACDBS is OPTIONAL from Armv9.4.

If FEAT_HACDBS is implemented, then FEAT_HDBSS is implemented.

The following field identifies the presence of FEAT_HACDBS:

- ID_AA64MMFR4_EL1.HACDBS.

For more information, see Hardware accelerator for cleaning dirty state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_HDBSS, Hardware Dirty state tracking structure

FEAT_HDBSS introduces the Hardware Dirty state tracking structure, which enhances tracking the dirty state of stage 2 translation table descriptors.

This feature is supported in AArch64 state only.

FEAT_HDBSS is OPTIONAL from Armv9.4.

If FEAT_HDBSS is implemented, then FEAT_HAFDBS is implemented.

If FEAT_HDBSS is implemented, then FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_HDBSS:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see Hardware dirty state tracking structure.

FEAT_LUT, Lookup table instructions with 2-bit and 4-bit indices

FEAT_LUT introduces the following lookup table instructions with 2-bit and 4-bit indices:

- The Advanced SIMD `LUTI2` and `LUTI4` instructions.
- If FEAT_SVE2 is implemented, the SVE `LUTI2` and `LUTI4` instructions when the PE is not in Streaming SVE mode.
- If FEAT_SME2 is implemented, the SVE `LUTI2` and `LUTI4` instructions when the PE is in Streaming SVE mode.

FEAT_LUT is OPTIONAL from Armv9.2.

In an Armv9.5 implementation, if FEAT_AdvSIMD is implemented, FEAT_LUT is implemented.

If FEAT_LUT is implemented, then FEAT_AdvSIMD, FEAT_SVE2, or FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_LUT:

- ID_AA64ISAR2_EL1.LUT.

For more information, see:

- For Advanced SIMD, Lookup table using 2-bit or 4-bit indices.
- For SVE, Lookup table using 2-bit or 4-bit indices.

FEAT_PAuth_LR, Pointer authentication instructions that allow signing of LR using SP and PC as diversifiers

FEAT_PAuth_LR introduces instructions that allow signing of the 64-bit value in LR against the value of SP, and a PC-relative instruction address.

This feature is supported in AArch64 state only.

FEAT_PAuth_LR is OPTIONAL from Armv9.4.

If FEAT_PAuth_LR is implemented, then FEAT_FPACCOMBINE is implemented.

If FEAT_PAuth_LR is implemented, then FEAT_SCTLR2 is implemented.

When FEAT_PAuth_LR and FEAT_AA64EL2 are implemented, FEAT_HCX is implemented.

The following fields identify the presence of FEAT_PAuth_LR:

- ID_AA64ISAR3_EL1.PACM.
- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3_SME, Performance Monitors extensions for SME

FEAT_PMUv3_SME supports filtering to control counting events in Streaming and Non-streaming SVE modes.

This feature is supported in AArch64 state only.

FEAT_PMUv3_SME is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_PMUv3 and FEAT_SME are implemented, FEAT_PMUv3_SME is implemented.

If FEAT_PMUv3_SME is implemented, then FEAT_PMUv3 is implemented.

If FEAT_PMUv3_SME is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_PMUv3_SME:

- PMMIR_EL1.SME.

For more information, see Filtering by SVE Streaming mode.

FEAT_PMUv3_TH2, Performance Monitors event counter linking extension

FEAT_PMUv3_TH2 extends the Performance Monitors Extension to add the ability to program an event counter to count the logical combination of two events.

This feature is supported in both AArch64 and AArch32 states. The threshold condition controls are accessible only in AArch64 state. However, threshold conditions still apply in AArch32 state.

FEAT_PMUv3_TH2 is OPTIONAL from Armv9.4.

If FEAT_PMUv3_TH2 is implemented, then FEAT_PMUv3_TH is implemented.

If FEAT_PMUv3_TH2 is implemented, then FEAT_PMUv3_EDGE is implemented.

The following fields identify the presence of FEAT_PMUv3_TH2:

- PMMIR_EL1.EDGE.
- PMMIR.EDGE.

For more information, see Event threshold and edge counting.

FEAT_RME_GPC2, RME Granule Protection Check 2 Extension

FEAT_RME_GPC2 introduces two mechanisms to the GPC that enable memory isolation models:

- Non-secure Only GPI encoding.
- Physical Address Space Disable.

This feature is supported in AArch64 state only.

FEAT_RME_GPC2 is OPTIONAL from Armv9.4.

If FEAT_RME_GPC2 is implemented, then FEAT_RME is implemented.

The following field identifies the presence of FEAT_RME_GPC2:

- ID_AA64PFR0_EL1.RME.

For more information, see:

- GPC faults.
- GPI field encoding in GPT descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SME_F8F16, SME2 ZA-targeting FP8 multiply-accumulate, dot product, and outer product to half-precision instructions

FEAT_SME_F8F16 introduces the following SME instructions:

- ZA-targeting FP8 to half-precision multiply-accumulate, dot product, and outer product instructions.
- ZA-targeting multi-vector non-widening half-precision FADD and FSUB instructions.

FEAT_SME_F8F16 is OPTIONAL from Armv9.2.

If FEAT_SME_F8F16 is implemented, then FEAT_SME_F8F32 is implemented.

The following field identifies the presence of FEAT_SME_F8F16:

- ID_AA64SMFR0_EL1.F8F16.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point widening multiply-accumulate.
- FP8 floating-point dot product.
- FP8 floating-point outer product.

FEAT_SME_F8F32, SME2 ZA-targeting FP8 multiply-accumulate, dot product, and outer product to single-precision instructions

FEAT_SME_F8F32 introduces the SME ZA-targeting FP8 to single-precision multiply-accumulate, dot product, and outer product instructions.

FEAT_SME_F8F32 is OPTIONAL from Armv9.2.

If FEAT_SME_F8F32 is implemented, then FEAT_SME2 is implemented.

If FEAT_SME_F8F32 is implemented, then FEAT_FP8 is implemented.

The following field identifies the presence of FEAT_SME_F8F32:

- ID_AA64SMFR0_EL1.F8F32.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point widening multiply-accumulate.
- FP8 floating-point dot product.
- FP8 floating-point outer product.

FEAT_SME_LUTv2, Lookup table instructions with 4-bit indices and 8-bit elements

FEAT_SME_LUTv2 introduces the following:

- The LUTI4 (four registers, 8-bit) instruction.
- The MOVT (vector to table) instruction.

FEAT_SME_LUTv2 is OPTIONAL from Armv9.2.

If FEAT_SME_LUTv2 is implemented, then FEAT_SME2 is implemented.

The following field identifies the presence of FEAT_SME_LUTv2:

- ID_AA64SMFR0_EL1.LUTv2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPE_ALTCLK, Statistical Profiling alternate clock domain extension

FEAT_SPE_ALTCLK supports collection of timing data using the Statistical Profiling Extension when profiling a PE that includes asynchronous accelerators, such as a Streaming Mode Compute Unit (SMCU).

This feature is supported in AArch64 state only.

FEAT_SPE_ALTCLK is OPTIONAL from Armv9.4.

If FEAT_SPE_ALTCLK is implemented, then FEAT_SPE is implemented.

The following field identifies the presence of FEAT_SPE_ALTCLK:

- PMSIDR_EL1.ALTCLK.

For more information, see Asynchronous operation.

FEAT_SPE_EFT, Statistical Profiling extended filtering by type

FEAT_SPE_EFT extends SPE filtering to support targeted profiling of floating-point and SIMD operations.

This feature is supported in AArch64 state only.

FEAT_SPE_EFT is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_SPE is implemented, FEAT_SPE_EFT is implemented.

If FEAT_SPE_EFT is implemented, then FEAT_SPE is implemented.

If FEAT_SPE_EFT is implemented, then FEAT_SPE_FPF is implemented.

The following field identifies the presence of FEAT_SPE_EFT:

- PMSIDR_EL1.EFT.

For more information, see Filtering by operation type.

FEAT_SPE_FPF, Statistical Profiling floating-point and SIMD flag extension

FEAT_SPE_FPF extends the profiling information recorded by the Statistical Profiling Extension to identify sampled scalar floating-point and Advanced SIMD instructions.

This feature is supported in AArch64 state only.

FEAT_SPE_FPF is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_SPE is implemented, FEAT_SPE_FPF is implemented.

If FEAT_SPE_FPF is implemented, then FEAT_SPE is implemented.

If FEAT_SPE_FPF is implemented, then FEAT_SPE_EFT is implemented.

The following field identifies the presence of FEAT_SPE_FPF:

- PMSIDR_EL1.FPF.

For more information, see Additional information for other operations.

FEAT_SPE_SME, Statistical Profiling extensions for SME

FEAT_SPE_SME provides support for profiling of software that uses SME instructions using the Statistical Profiling Extension.

This feature is supported in AArch64 state only.

FEAT_SPE_SME is OPTIONAL from Armv9.2.

In an Armv9.5 implementation, if FEAT_SPEv1p1 and FEAT_SME are implemented, FEAT_SPE_SME is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SPE_SME is implemented, then FEAT_SPEv1p1 is implemented.

If FEAT_SPE_SME is implemented, then FEAT_SME is implemented.

The following field identifies the presence of FEAT_SPE_SME:

- PMSIDR_EL1.SME.

For more information, see Additional information for each profiled SVE and SME operation.

FEAT_SPMU2, System Performance Monitors Extension version 2

FEAT_SPMU2 provides the functionality to set multiple event counters in a System PMU to zero in a single operation.

This feature is supported in both AArch64 and AArch32 states.

FEAT_SPMU2 is OPTIONAL from Armv9.4.

In an Armv9.5 implementation, if FEAT_SPMU is implemented, FEAT_SPMU2 is implemented.

If FEAT_SPMU2 is implemented, then FEAT_SPMU is implemented.

The following field identifies the presence of FEAT_SPMU2:

- ID_AA64DFR1_EL1.SPMU.

For more information, see Zeroing System PMU counters.

FEAT_SSVE_FP8DOT2, SVE FP8 2-way dot product to half-precision instructions in Streaming SVE mode

FEAT_SSVE_FP8DOT2 enables execution of SVE FP8 to half-precision 2-way dot product instructions in Streaming SVE mode.

FEAT_SSVE_FP8DOT2 is OPTIONAL from Armv9.2.

If FEAT_SSVE_FP8DOT2 is implemented, then FEAT_SSVE_FP8DOT4 is implemented.

If FEAT_SME2 and FEAT_FP8DOT2 are implemented, then FEAT_SSVE_FP8DOT2 is implemented.

The following field identifies the presence of FEAT_SSVE_FP8DOT2:

- ID_AA64SMFR0_EL1.SF8DP2.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point dot product.

FEAT_SSVE_FP8DOT4, SVE2 FP8 4-way dot product to single-precision instructions in Streaming SVE mode

FEAT_SSVE_FP8DOT4 enables execution of SVE FP8 to single-precision 4-way dot product instructions in Streaming SVE mode.

FEAT_SSVE_FP8DOT4 is OPTIONAL from Armv9.2.

If FEAT_SSVE_FP8DOT4 is implemented, then FEAT_SSVE_FP8FMA is implemented.

If FEAT_SME2 and FEAT_FP8DOT4 are implemented, then FEAT_SSVE_FP8DOT4 is implemented.

The following field identifies the presence of FEAT_SSVE_FP8DOT4:

- ID_AA64SMFR0_EL1.SF8DP4.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point dot product.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SSVE_FP8FMA, SVE2 FP8 multiply-accumulate to half-precision and single-precision instructions in Streaming SVE mode

FEAT_SSVE_FP8FMA enables execution of SVE FP8 to half-precision and single-precision multiply-accumulate instructions in Streaming SVE mode.

FEAT_SSVE_FP8FMA is OPTIONAL from Armv9.2.

If FEAT_SSVE_FP8FMA is implemented, then FEAT_FP8 is implemented.

If FEAT_SSVE_FP8FMA is implemented, then FEAT_SME2 is implemented.

If FEAT_SME2 and FEAT_FP8FMA are implemented, then FEAT_SSVE_FP8FMA is implemented.

The following field identifies the presence of FEAT_SSVE_FP8FMA:

- ID_AA64SMFR0_EL1.SF8FMA.

For more information, see:

- Summary of FP8 instruction behaviors.
- FP8 floating-point widening multiply-accumulate.

FEAT_STEP2, Enhanced Software Step Extension

FEAT_STEP2 introduces support for a debugger to control the instruction executed by the PE when single-stepping, without modifying the instruction in memory.

This feature is supported in both AArch64 and AArch32 states.

FEAT_STEP2 is OPTIONAL from Armv9.4.

FEAT_STEP2 is mandatory from Armv9.5.

When FEAT_STEP2 and FEAT_EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_STEP2:

- ID_AA64DFR2_EL1.STEP.

For more information, see Behavior in the active-not-pending state.

FEAT_TLBIW, TLBI VMALL for Dirty state

FEAT_TLBIW provides the following TLBI instructions that remove the stage 2 dirty state from TLB entries:

- TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS.
- TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS.
- TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS.

This feature is supported in AArch64 state only.

FEAT_TLBIW is OPTIONAL from Armv9.4.

If FEAT_TLBIW is implemented, then FEAT_AA64EL2 is implemented.

The following field identifies the presence of FEAT_TLBIW:

- ID_AA64ISAR3_EL1.TLBIW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.3.7 The Armv9.6 architecture extension

The Armv9.6 architecture extension is an extension to Armv9.5. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv9.6 compliant if all of the following apply:

- It is Armv9.5 compliant.
- It includes all of the Armv9.6 architectural features that are mandatory.

An Armv9.6 compliant implementation can additionally include:

- Armv9.6 features that are optional.

FEAT_CMPBR, Compare and Branch instructions

FEAT_CMPBR introduces the A64 base compare and branch instructions.

This feature is supported in AArch64 state only.

FEAT_CMPBR is OPTIONAL from Armv9.5.

FEAT_CMPBR is mandatory from Armv9.6.

The following field identifies the presence of FEAT_CMPBR:

- ID_AA64ISAR2_EL1.CSSC.

For more information, see:

- Conditional branch.
- Concurrent modification and execution of instructions.

FEAT_F8F16MM, 8-bit floating-point matrix multiply-accumulate to half-precision

FEAT_F8F16MM introduces the Advanced SIMD 8-bit floating-point matrix multiply-accumulate to half-precision instruction.

If FEAT_SVE2 is implemented, FEAT_F8F16MM implements the SVE 8-bit floating-point matrix multiply-accumulate to half-precision instruction, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_F8F16MM is OPTIONAL from Armv9.2.

If FEAT_F8F16MM is implemented, then FEAT_F8F32MM is implemented.

If FEAT_F8F16MM is implemented, then FEAT_FP8DOT2 is implemented.

The following field identifies the presence of FEAT_F8F16MM:

- ID_AA64FPFR0_EL1.F8MM4.

For more information, see:

- For Advanced SIMD: FP8 matrix multiply-accumulate.
- For SVE: FP8 floating-point matrix multiply-accumulate.

FEAT_F8F32MM, 8-bit floating-point matrix multiply-accumulate to single-precision

FEAT_F8F32MM introduces the Advanced SIMD 8-bit floating-point matrix multiply-accumulate to single-precision instruction.

If FEAT_SVE2 is implemented, FEAT_F8F32MM implements the SVE 8-bit floating-point matrix multiply-accumulate to single-precision instruction, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_F8F32MM is OPTIONAL from Armv9.2.

If FEAT_F8F32MM is implemented, then FEAT_FP8DOT4 is implemented.

The following field identifies the presence of FEAT_F8F32MM:

- ID_AA64FPFR0_EL1.F8MM8.

For more information, see:

- For Advanced SIMD: FP8 matrix multiply-accumulate.
- For SVE: FP8 floating-point matrix multiply-accumulate.

FEAT_FPRCVT, Floating-Point to/from Integer in Scalar FP register

FEAT_FPRCVT introduces A64 base floating-point to integer and integer to floating-point convert instructions with only scalar SIMD&FP register operands and results, and with different input and output register sizes.

FEAT_FPRCVT implements the Advanced SIMD scalar floating-point to integer and integer to floating-point convert instructions with only scalar SIMD&FP register operands and results, and with different and same input and output register sizes, when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_FPRCVT is OPTIONAL from Armv9.5.

If FEAT_FPRCVT is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_FPRCVT:

- ID_AA64ISAR3_EL1.FPRCVT.

For more information, see:

- Floating-point conversion.

FEAT_IDTE3, Trapping ID register accesses to EL3

FEAT_IDTE3 introduces support for trapping ID register accesses to EL3.

This feature is supported in AArch64 state only.

In an Armv9.6 implementation, if FEAT_EL3 is implemented, FEAT_IDTE3 is implemented.

FEAT_IDTE3 is OPTIONAL from Armv9.0.

The following field identifies the presence of FEAT_IDTE3:

- ID_AA64MMFR2_EL1.IDS.

FEAT_LS64WB, LS64 for Write-back cacheable memory

FEAT_LS64WB provides support for large single-copy atomic load and store instructions for accesses to Conventional memory to enable scalability of atomic message passing without relying on separate flag and data accesses.

This feature is supported in AArch64 state only.

FEAT_LS64WB is OPTIONAL from Armv9.2.

If FEAT_LS64WB is implemented, then FEAT_LS64_ACCDATA is implemented.

The following field identifies the presence of FEAT_LS64WB:

- ID_AA64ISAR1_EL1.LS64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LSFE, Large System Float Extension

FEAT_LSFE implements A64 base Atomic floating-point in-memory instructions.

This feature is supported in AArch64 state only.

FEAT_LSFE is OPTIONAL from Armv9.3.

If FEAT_LSFE is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_LSFE:

- ID_AA64ISAR3_EL1.LSFE.

For more information, see:

- Atomic floating-point memory operations.

FEAT_LSUI, Unprivileged Load Store

FEAT_LSUI introduces unprivileged variants of load and store instructions so that clearing PSTATE.PAN is never required in privileged software.

This feature is supported in AArch64 state only.

FEAT_LSUI is OPTIONAL from Armv9.5.

FEAT_LSUI is mandatory from Armv9.6.

The following field identifies the presence of FEAT_LSUI:

- ID_AA64ISAR3_EL1.LSUI.

FEAT_MPAM_PE_BW_CTRL, MPAM PE-side Bandwidth Controls

FEAT_MPAM_PE_BW_CTRL introduces support for PE-side bandwidth controls.

This feature is supported in AArch64 state only.

FEAT_MPAM_PE_BW_CTRL is OPTIONAL from Armv9.3.

If FEAT_MPAM_PE_BW_CTRL is implemented, then FEAT_MPAMv1p1 or FEAT_MPAMv0p1 is implemented.

The following field identifies the presence of FEAT_MPAM_PE_BW_CTRL:

- MPAMIDR_EL1.HAS_BW_CTRL.

FEAT_NV2p1, Enhanced nested virtualization support

FEAT_NV2p1 further supports nested virtualization by ensuring that control bits in EL1 registers are stateful if the corresponding bits in the corresponding EL2 registers are stateful. This is to prevent loss of the Guest hypervisor's state, and to permit the Host hypervisor to correctly emulate the EL2 environment.

This feature is supported in AArch64 state only.

FEAT_NV2p1 is OPTIONAL from Armv8.4.

In an Armv9.6 implementation, if FEAT_NV is implemented, FEAT_NV2p1 is implemented.

If FEAT_NV2p1 is implemented, then FEAT_NV2 is implemented.

The following field identifies the presence of FEAT_NV2p1:

- ID_AA64MMFR4_EL1.NV_frac.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_OCCMO, Outer Cacheable Cache Maintenance Operation

FEAT_OCCMO introduces the DC CIVAOC system instruction that provides software with a mechanism to publish writes to the Outer cache level.

This feature is supported in AArch64 state only.

FEAT_OCCMO is OPTIONAL from Armv9.3.

FEAT_OCCMO is mandatory from Armv9.6.

The following field identifies the presence of FEAT_OCCMO:

- ID_AA64ISAR3_EL1.OCCMO.

FEAT_PCDPHINT, Producer-Consumer Data Placement Hints

FEAT_PCDPHINT provides hint instructions to indicate each of the following:

- A store in the current execution thread is generating data at a specific location, which a thread of execution on one or more other observers is waiting on.
- The thread of execution on the current PE will read a location that may not yet have been written with the value to be consumed.

This feature is supported in AArch64 state only.

FEAT_PCDPHINT is OPTIONAL from Armv9.0.

The following field identifies the presence of FEAT_PCDPHINT:

- ID_AA64ISAR2_EL1.PCDPHINT.

FEAT_PMUv3_EXTPMN, Reserving PMU event counters for external agents

FEAT_PMUv3_EXTPMN provides external agents with a mechanism to reserve event counters for external use.

This feature is supported in both AArch64 and AArch32 states.

FEAT_PMUv3_EXTPMN is OPTIONAL from Armv9.5.

If FEAT_PMUv3_EXTPMN is implemented, then FEAT_PMUv3 is implemented.

If FEAT_PMUv3_EXTPMN is implemented, then FEAT_PMUv3_EXT is implemented.

If FEAT_PMUv3_EXTPMN is implemented, then FEAT_FGT is implemented.

When FEAT_PMUv3_EXTPMN and FEAT_EL2 are implemented, FEAT_HPMN0 is implemented.

The following field identifies the presence of FEAT_PMUv3_EXTPMN:

- PMDEVID.EXTPMN.

FEAT_PoPS, Point of Physical Storage

FEAT_PoPS defines the Point of Physical Storage (PoPS) and introduces System instructions to clean and invalidate to the PoPS.

This feature is supported in AArch64 state only.

FEAT_PoPS is OPTIONAL from Armv9.5.

The following field identifies the presence of FEAT_PoPS:

- ID_AA64MMFR4_EL1.PoPS.

For more information, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RME_GDI, RME Granular Data Isolation extension

FEAT_RME_GDI introduces additional GPT GPI encodings to enable memory isolation of non-PE data flows from the PEs, within an RME system.

This feature is supported in AArch64 state only.

FEAT_RME_GDI is OPTIONAL from Armv9.4.

If FEAT_RME_GDI is implemented, then FEAT_RME is implemented.

If FEAT_RME_GDI is implemented, then FEAT_RME_GPC2 is implemented.

The following field identifies the presence of FEAT_RME_GDI:

- ID_AA64MMFR4_EL1.RMEGDI.

FEAT_RME_GPC3, RME Granule Protection Check 3 Extension

FEAT_RME_GPC3 introduces a method for defining a set of windows in the memory map for which Granule Protection Table lookups are skipped, and instead applies a set of default settings associated with the window.

FEAT_RME_GPC3 introduces the APAS system instruction, which provides an architected mechanism for directly programming the PA space association for regions of memory that support memory-side dynamic PA space association.

This feature is supported in AArch64 state only.

FEAT_RME_GPC3 is OPTIONAL from Armv9.5.

If FEAT_RME_GPC3 is implemented, then FEAT_RME is implemented.

If FEAT_RME_GPC3 is implemented, then FEAT_RME_GPC2 is implemented.

The following field identifies the presence of FEAT_RME_GPC3:

- ID_AA64PFR0_EL1.RME.

FEAT_SME2p2, Scalable Matrix Extension version 2.2

The Scalable Matrix Extension version 2.2 (SME2.2) is a superset of SME2.1 that adds:

- SME multi-vector Z-targeting non-widening floating-point FMUL instruction.
- Quarter-tile outer product instructions.
- Structured sparsity outer product instructions.
- Support for SVE2.2 instructions in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SME2p2 is OPTIONAL from Armv9.5.

If FEAT_SME2p2 is implemented, then FEAT_SME2p1 is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SME_MOP4 is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SME_TMOP is implemented.

If FEAT_SME2p2 is implemented, then FEAT_SSVE_FEXPA is implemented.

If FEAT_SME and FEAT_SVE2p2 are implemented, then FEAT_SME2p2 is implemented.

The following field identifies the presence of FEAT_SME2p2:

- ID_AA64SMFR0_EL1.SMEver.

For more information, see:

- The Scalable Matrix Extension.
- Multi-vector multiply.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Quarter-tile outer product.
- Structured sparsity outer product.

FEAT_SME_MOP4, Quarter-tile outer product instructions

FEAT_SME_MOP4 introduces the SME Quarter-tile outer product instructions

FEAT_SME_MOP4 is OPTIONAL from Armv9.4.

If FEAT_SME_MOP4 is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SME_MOP4:

- ID_AA64SMFR0_EL1.SMOP4.

For more information, see:

- Quarter-tile outer product.

FEAT_SME_TMOP, Structured sparsity outer product instructions

FEAT_SME_TMOP introduces the SME Structured sparsity outer product instructions.

FEAT_SME_TMOP is OPTIONAL from Armv9.4.

If FEAT_SME_TMOP is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SME_TMOP:

- ID_AA64SMFR0_EL1.STMOP.

For more information, see:

- Structured sparsity outer product.

FEAT_SPE_EXC, SPE Profiling exception extension

FEAT_SPE_EXC provides support for reporting Profiling Buffer management events as SPE Profiling exceptions.

This feature is supported in AArch64 state only.

FEAT_SPE_EXC is OPTIONAL from Armv9.5.

If FEAT_SPE_EXC is implemented, then FEAT_SPEv1p5 is implemented.

The following field identifies the presence of FEAT_SPE_EXC:

- ID_AA64DFR2_EL1.SPE_EXC.

FEAT_SPE_nVM, Statistical Profiling physical addressing mode extension

FEAT_SPE_nVM provides support for defining the Profiling Buffer using physical addresses or intermediate physical addresses.

This feature is supported in AArch64 state only.

FEAT_SPE_nVM is OPTIONAL from Armv9.5.

If FEAT_SPE_nVM is implemented, then FEAT_SPE is implemented.

When FEAT_SPE_nVM and FEAT_EL2 are implemented, FEAT_FGT2 is implemented.

The following field identifies the presence of FEAT_SPE_nVM:

- ID_AA64DFR2_EL1.SPE_nVM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPEv1p5, Statistical Profiling Extension version 1.5

FEAT_SPEv1p5 introduces the following to the Statistical Profiling Extension:

- If EL2 and FEAT_FGT are implemented, a fine-grained trap on the `PSB CSYNC` instruction.
- The SPE Profiling exception extension, FEAT_SPE_EXC.
- The Statistical Profiling physical addressing mode extension, FEAT_SPE_nVM.

This feature is supported in AArch64 state only.

FEAT_SPEv1p5 is OPTIONAL from Armv9.5.

In an Armv9.6 implementation, if FEAT_SPE is implemented, FEAT_SPEv1p5 is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPEv1p4 is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPE_CRR is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPE_EXC is implemented.

If FEAT_SPEv1p5 is implemented, then FEAT_SPE_nVM is implemented.

The following field identifies the presence of FEAT_SPEv1p5:

- ID_AA64DFR0_EL1.PMSVer.

FEAT_SRMASK, Bitwise System Register Write Masks

FEAT_SRMASK introduces aliases and bitwise write masks for EL1 control registers to reduce trapping of EL1 System register accesses to EL2. The equivalent bitwise write masks are also present for EL2.

This feature is supported in AArch64 state only.

FEAT_SRMASK is OPTIONAL from Armv9.5.

FEAT_SRMASK is mandatory from Armv9.6.

If FEAT_SRMASK is implemented, then FEAT_E2H0 is not implemented.

The following field identifies the presence of FEAT_SRMASK:

- ID_AA64MMFR4_EL1.SRMASK.

FEAT_SSVE_AES, Streaming SVE Mode Advanced Encryption Standard and 128-bit polynomial multiply long instructions

FEAT_SSVE_AES implements the SVE AES and 128-bit polynomial multiply long instructions, identified as implemented by ID_AA64ZFR0_EL1.AES field, when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SSVE_AES is OPTIONAL from Armv9.5.

If FEAT_SSVE_AES is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SSVE_AES:

- ID_AA64SMFR0_EL1.AES.

For more information, see:

- AES-128 instructions.
- Polynomial arithmetic.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SSVE_BitPerm, Streaming Scalable Vector Bit Permutes instructions

FEAT_SSVE_BitPerm implements the SVE2 bit permute instructions, identified as implemented by ID_AA64ZFR0_EL1.BitPerm, when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SSVE_BitPerm is OPTIONAL from Armv9.4.

If FEAT_SSVE_BitPerm is implemented, then FEAT_SME2p1 is implemented.

When FEAT_SSVE_BitPerm and FEAT_SVE2 is implemented, FEAT_SVE_BitPerm is implemented.

The following field identifies the presence of FEAT_SSVE_BitPerm:

- ID_AA64SMFR0_EL1.SBitPerm.

For more information, see:

- Bit permutation.

FEAT_SSVE_FEXPA, Streaming FEXPA instruction

FEAT_SSVE_FEXPA implements the SVE FEXPA instruction when the PE is in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SSVE_FEXPA is OPTIONAL from Armv9.4.

If FEAT_SSVE_FEXPA is implemented, then FEAT_SME2p1 is implemented.

The following field identifies the presence of FEAT_SSVE_FEXPA:

- ID_AA64SMFR0_EL1.SFEXPA.

FEAT_SVE2p2, Scalable Vector Extensions version 2.2

The Scalable Vector Extension version 2.2 (SVE2.2) is a superset of SVE2.1 that adds:

- Floating-point round to integral floating-point value instructions.
- Scalar index of first/last true predicate element.
- Zeroing-predication variant to all SVE unary instructions.
- Additional variants of `COMPACT` and `EXPAND` instructions.

This feature is supported in AArch64 state only.

FEAT_SVE2p2 is OPTIONAL from Armv9.5.

If FEAT_SVE2p2 is implemented, then FEAT_SVE2p1 is implemented.

If FEAT_SVE2 and FEAT_SME2p2 are implemented, then FEAT_SVE2p2 is implemented.

The following field identifies the presence of FEAT_SVE2p2:

- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- Predicate element index.
- Floating-point round.
- Vector permutation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SVE_AES2, SVE multi-vector Advanced Encryption Standard and 128-bit polynomial multiply long instructions

FEAT_SVE_AES2 implements the SVE multi-vector Advanced Encryption Standard and 128-bit destination element polynomial multiply long instructions, when the PE is not in Streaming SVE mode.

This feature is supported in AArch64 state only.

FEAT_SVE_AES2 is OPTIONAL from Armv9.5.

If FEAT_SVE_AES2 is implemented, then FEAT_SVE_PMULL128 is implemented.

If FEAT_SVE_AES2 is implemented, then FEAT_SVE2p1 or FEAT_SSVE_AES is implemented.

The following field identifies the presence of FEAT_SVE_AES2:

- ID_AA64ZFR0_EL1.AES.

For more information, see:

- AES-128 instructions.
- Polynomial arithmetic.

FEAT_SVE_BFSCALE, BFloat16 Floating-Point Adjust Exponent

FEAT_SVE_BFSCALE introduces the following:

- If FEAT_SVE2 is implemented, the SVE `BFSCALE` instruction when the PE is not in Streaming SVE mode.
- If FEAT_SME2 is implemented, the SVE `BFSCALE` instruction when the PE is in Streaming SVE mode, and the SME multi-vector Z-targeting BFloat16 scaling instructions, `BFSCALE` and `BFMUL`.

This feature is supported in AArch64 state only.

FEAT_SVE_BFSCALE is OPTIONAL from Armv9.2.

If FEAT_SVE_BFSCALE is implemented, then FEAT_SVE_B16B16 is implemented.

The following field identifies the presence of FEAT_SVE_BFSCALE:

- ID_AA64ZFR0_EL1.B16B16.

For more information, see:

- For SVE: Floating-point adjust exponent.
- For SME: Floating-point adjust exponent.

FEAT_SVE_F16F32MM, SVE Half-precision floating-point matrix multiply-accumulate to single-precision

FEAT_SVE_F16F32MM introduces the SVE half-precision floating-point matrix multiply-accumulate to single-precision instruction.

This feature is supported in AArch64 state only.

FEAT_SVE_F16F32MM is OPTIONAL from Armv9.2.

If FEAT_SVE_F16F32MM is implemented, then FEAT_SVE2p1 is implemented.

The following field identifies the presence of FEAT_SVE_F16F32MM:

- ID_AA64ZFR0_EL1.F16MM.

For more information, see:

- Floating-point matrix multiply-accumulate.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TRBE_EXC, Trace Buffer Profiling exception extension

FEAT_TRBE_EXC provides support for reporting trace buffer management events as TRBE Profiling exceptions.

This feature is supported in AArch64 state only.

FEAT_TRBE_EXC is OPTIONAL from Armv9.5.

If FEAT_TRBE_EXC is implemented, then FEAT_TRBEv1p1 is implemented.

The following field identifies the presence of FEAT_TRBE_EXC:

- ID_AA64DFR2_EL1.TRBE_EXC.

FEAT_TRBEv1p1, Trace Buffer Extension version 1.1

FEAT_TRBEv1p1 introduces the following to the Trace Buffer Extension:

- If EL2 and FEAT_FGT are implemented, a fine-grained trap on the `TSB CSYNC` instruction.
- If EL2 is implemented, an EL2 control to override TRBLIMITR_EL1.nVM.
- Support for the TRBE Profiling exception extension, FEAT_TRBE_EXC.

This feature is supported in AArch64 state only.

FEAT_TRBEv1p1 is OPTIONAL from Armv9.5.

In an Armv9.6 implementation, if FEAT_TRBE is implemented, FEAT_TRBEv1p1 is implemented.

If FEAT_TRBEv1p1 is implemented, then FEAT_TRBE is implemented.

If FEAT_TRBEv1p1 is implemented, then FEAT_TRBE_EXC is implemented.

The following fields identify the presence of FEAT_TRBEv1p1:

- ID_AA64DFR0_EL1.TraceBuffer.
- TRBDEVARCH.REVISION.

FEAT_UINJ, Injection of Undefined Instruction exceptions

FEAT_UINJ introduces support for software injection of Undefined Instruction exceptions.

This feature is supported in AArch64 state only.

FEAT_UINJ is OPTIONAL from Armv9.0.

FEAT_UINJ is mandatory from Armv9.6.

The following field identifies the presence of FEAT_UINJ:

- ID_AA64PFR2_EL1.UINJ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part B
The AArch64 Application Level Architecture

Chapter B1
The AArch64 Application Level Programmers' Model

This chapter describes the AArch64 application level programmers' model. It contains the following sections:

- About the Application level programmers' model.
- Registers in AArch64 Execution state.
- Process state, PSTATE.
- The Scalable Vector and Scalable Matrix Extensions (SVE & SME).
- Software control features and EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see Exception levels.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to The AArch64 System Level Programmers' Model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.2 Registers in AArch64 Execution state

The following registers are visible at EL0 using AArch64:

R0-R30 31 general-purpose registers, R0 to R30. Each can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

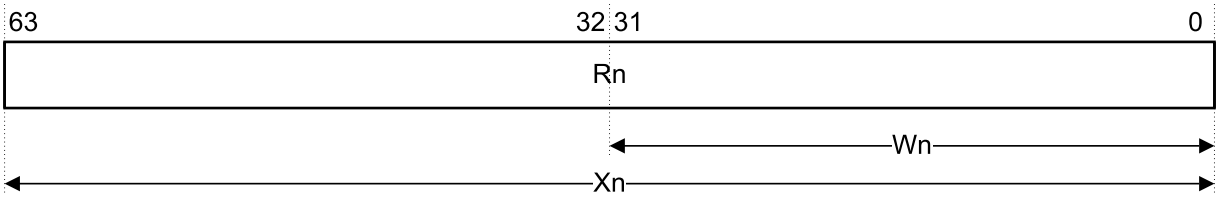

Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

Note

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see PC alignment checking.

V0-V31 32 SIMD&FP registers, V0 to V31. Each can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.
- A 128-bit vector of elements. See Figure A1-1.
- A 64-bit vector of elements. See Figure A1-1.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

For more information about data types and vector formats, see Supported data types.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

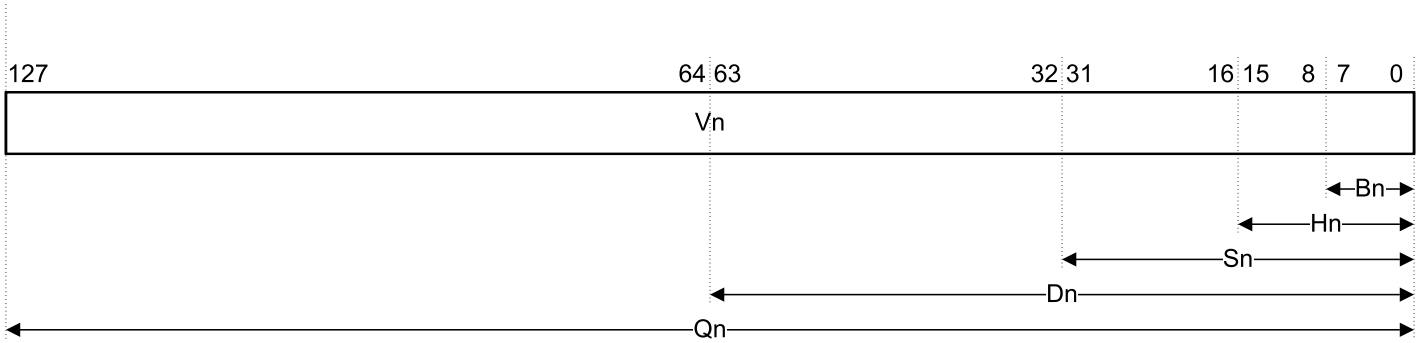

Figure B1-2 SIMD and floating-point register naming

FPCR, FPSR The FPCR is the floating-point control register. The FPSR is the floating-point status register.

Z0-Z31 32 SVE scalable vector registers, Z0 to Z31, of equal length. Each register can be accessed as:

- A configurable-length vector of elements. The length, VL, is a power of two, from a minimum of 128 bits to an IMPLEMENTATION DEFINED maximum no greater than 2048 bits. See Figure B1-3, Figure A1-5, and Configurable SVE vector lengths.
- A SIMD&FP register, as described in V0-V31. Bits[127:0] of each Zn register hold the correspondingly numbered V0-V31 SIMD&FP register, as Figure B1-3 shows:

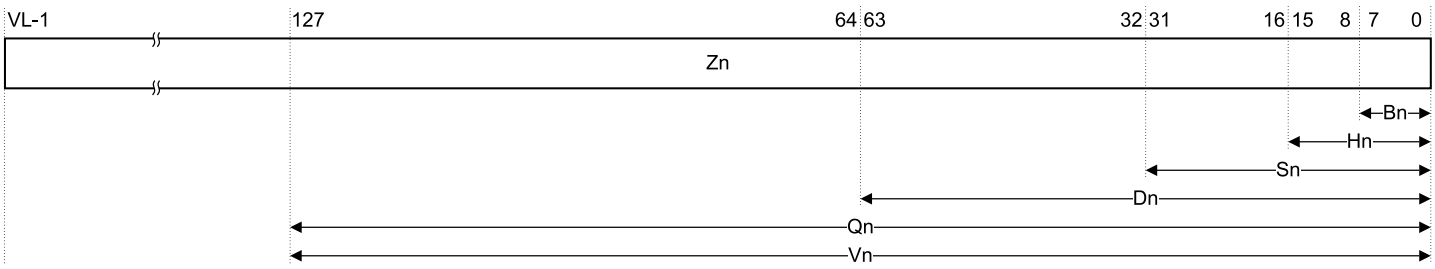

Figure B1-3 SVE scalable vector register naming

See also:

- Maximum implemented SVE vector lengths.
- Configurable SVE vector lengths.
- Treatment of SVE scalable vector registers.
- SVE writes of scalar values to registers.

P0-P15 16 SVE predicate registers, named P0 to P15. Each SVE predicate register holds one bit for each byte of an SVE scalar vector register.

Note

The *Maximum implemented SVE predicate length* is the *Maximum implemented SVE vector length* divided by 8. See Maximum implemented SVE vector lengths.

Also see Vector predication.

FFR The dedicated SVE First Fault Register that has the same size and format as the SVE predicate registers, P0-P15. See FFR, First Fault Register.

ZA Architectural state capable of holding a two-dimensional array of bytes. See ZA storage.

ZT0 A 512-bit SME2 lookup table register. See SME2 ZT0 register.

See also:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- System registers.
- Pseudocode description of registers in AArch64 state.
- Registers for instruction processing and exception handling.

B1.2.1 FFR, First Fault Register

R_{TRLWH} SVE has a dedicated First Fault Register named FFR.

I_{XPLQW} The FFR captures the cumulative fault status of a sequence of SVE First-fault and Non-fault vector load instructions.

R_{CPQQN} The FFR and the predicate registers have the same size and format.

I_{PBWPM} The FFR is a Special-purpose register.

R_{CGHCK} All bits in the FFR that are accessible at the current Exception level are initialized to 1 by using the `SETFFR` instruction.

R_{WZJVT} Bits in the FFR are indirectly set to 0 as a result of a suppressed access or suppressed fault corresponding to an *Active element* of an SVE First-fault or Non-fault vector load.

R_{BZLJG} Bits in the FFR are never set to 1 as a result of a vector load instruction.

I_{XLZQY} After a sequence of one or more SVE First-fault or Non-fault loads that follow a `SETFFR` instruction, without any intervening `WRFFR` instruction, the FFR contains a monotonic predicate value.

D_{SRDGX} A monotonic predicate value is one that consists of, starting from predicate element 0, a sequence of zero or more TRUE elements, followed only by zero or more FALSE elements.

I_{TQMTV} The TRUE elements in the FFR indicate the shortest sequence of consecutive elements that could contain valid data loaded from memory.

R_{GHFRQ} The only instructions that directly read the FFR are:

- `RDFFR`.
- `RDFFRS`.

R_{LHBRN} The only instructions that directly write the FFR are:

- `WRFFR`.
- `SETFFR`.

I_{DTHXN} The `WRFFR` instruction requires that its source predicate contains a monotonic predicate value. If the source is not a monotonic predicate value, then the resulting value in the FFR is UNKNOWN.

R_{XXMMP} All direct and indirect reads and writes to the FFR occur in program order relative to other instructions, without explicit synchronization.

B1.2.2 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register without access permission causes the instruction to be trapped. The registers that can be accessed from EL0 are:

Debug registers A Debug Communications Channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, DLR_EL0 and DSPSR_EL0 registers.

General system control registers

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.
- The SCXTNUM_EL0 register provides a number that can be used to separate out different context numbers.
- The POR_EL0 Permission Overlay register determines the unprivileged stage 1 overlay permissions for the EL1&0 and EL2&0 translation regimes.
- The TPIDR_EL0, TPIDR2_EL0, and TPIDRRO_EL0 registers provide locations where software can store thread identifying information.

Generic timer registers

- Read access to the system counter effective frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0, CNTVCT_EL0, CNTPCTSS_EL0 and CNTVCTSS_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

Performance Monitors registers

The Performance Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure some of these registers to be accessible at EL0.

For more details, see The Performance Monitors Extension.

Activity Monitors registers

The Activity Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure these registers to be accessible at EL0.

For more details, see The Activity Monitors Extension.

B1.2.3 Pseudocode description of registers in AArch64 state

The AArch64 pseudocode uses getter and setter accessor functions to read and write register files that are visible at EL0:

- The setter or assignment accessor, for example "`X{width}(num) = result`", is used for register writes.
- The getter or non-assignment accessor, "`operand = X{width}(num)`", is used for register reads.
- The *num* parameter specifies the register number to be accessed within the named register file, and *width* specifies the bit width of the register access.
- For the general-purpose register `X{width}()` accessor, register number 31 accesses the zero register, ZR, which reads as zero and ignores writes.
- For other register accessors, such as the SIMD&FP `V{width}()` accessors, there is no special interpretation of the register number.

Other special AArch64 accessors include:

- The `SP{width}()` accessors have no parameters and are used to read or write the currently selected 64-bit stack pointer at the current Exception level.
- The `PC64{width}()` accessor has no argument list and is used to read the 64-bit Program Counter.
- The `Vpart{width}()` accessors are used to perform a partial read or write of an Advanced SIMD vector register, using an additional *part* parameter to select which half of the vector register to access.
- The `Z{width}()`, `V{width}()`, and `Vpart{width}()` accessors read and write the same underlying vector register file.

Other register accessors are defined as required by other architectural features. See A-profile Architecture Pseudocode, sections aarch64/functions/registers/.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.3 Process state, PSTATE

Process state, or PSTATE, is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

For the system level view of PSTATE, see Process state, PSTATE in The AArch64 System Level Programmers' Model.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see Condition flags and related instructions.

The exception masking bits

D Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

A SError mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

SME enable controls

SM *Streaming SVE mode* enable:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0 Streaming SVE mode not enabled.

1 Streaming SVE mode enabled.

See Streaming SVE mode.

ZA *ZA storage* and *ZT0 register* enable:

0 ZA storage and the ZT0 register are not enabled.

1 ZA storage and the ZT0 register are enabled.

See About PSTATE.ZA, ZA storage and SME2 ZT0 register.

B1.3.1 Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using AArch64. All other PSTATE fields do not have direct read and write access at EL0.

Table B1-1 Accessing PSTATE fields at EL0 using MRS and MSR (register)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
SVCR	SM, ZA
SSBS	SSBS
DIT	DIT
TCO	TCO

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F, SM, ZA, SSBS, DIT, TCO}. Table B1-2 shows the MSR (immediate) operands that can directly write to PSTATE.{D, A, I, F, SM, ZA, SSBS, DIT, TCO} when the PE is at EL0 using AArch64 state.

Table B1-2 Accessing PSTATE.{D, A, I, F, SM, ZA, SSBS, DIT, TCO} at EL0 using MSR (immediate)

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Directly sets any of the PSTATE.{D, A, I, F} bits to 1
DAIFClr	D, A, I, F	Directly clears any of the PSTATE.{D, A, I, F} bits to 0
SVCRSM	SM	Directly sets the PSTATE.SM bit to CRm<0>.
SVCRZA	ZA	Directly sets the PSTATE.ZA bit to CRm<0>.
SVCRSMZA	SM, ZA	Directly sets the PSTATE.{SM, ZA} bits to CRm<0>.
SSBS	SSBS	Directly sets the PSTATE.SSBS bit to CRm<0>
DIT	DIT	Directly sets the PSTATE.DIT bit to CRm<0>
TCO	TCO	Directly sets the PSTATE.TCO bit to CRm<0>

Access to the PSTATE.{D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Software can more efficiently change PSTATE.{SM, ZA} by using SMSTART and SMSTOP instructions.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

B1.3.2 SVE use of PSTATE N, Z, C, and V Condition flags

I_{WYXLS} This section describes the SVE-specific use of PSTATE.

I_{YZYCQ} PSTATE N, Z, C and V condition flags can be updated by any of the following:

- An SVE instruction that generates a predicate result and updates the PSTATE N, Z, C and V Condition flags based on the value of the result.
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the value in its predicate source register or FFR:
  - PTEST.
  - RDFFRS (predicated).
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the values in its general-purpose source registers:
  - CTERMEQ.
  - CTERMNE.

R_{TPXTF} When setting the PSTATE N, Z, C and V Condition flags for SVE predicated flag-setting instructions, the instruction's *Governing predicate* determines which predicate elements are considered Active.

R_{QJBRW} When setting the PSTATE N, Z, C and V Condition flags for SVE unpredicated flag-setting instructions, all predicate elements are considered Active.

R_{ZMRXC} Unless otherwise specified in an instruction description, the SVE flag-setting instructions update the PSTATE N, Z, C and V Condition flags as follows:

Flag	SVE Name	SVE interpretation
N	First	Set to 1 if the *First active element* was TRUE, otherwise cleared to 0.
Z	None	Cleared to 0 if any *Active element* was TRUE, otherwise set to 1.
C	Not last	Cleared to 0 if the *Last active element* was TRUE, otherwise set to 1.
V	-	Cleared to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4 The Scalable Vector and Scalable Matrix Extensions (SVE & SME)

This section contains:

- Maximum implemented SVE vector lengths.
- Configurable SVE vector lengths.
- Treatment of SVE scalable vector registers.
- SVE writes of scalar values to registers.
- Vector predication.
- Streaming SVE mode.
- About PSTATE.ZA.
- ZA storage.
- ZA array vector access.
- ZA tile access.
- ZA storage layout.
- SME2 Multi-vector operands.
- SME2 ZT0 register.

B1.4.1 Maximum implemented SVE vector lengths

R_{QSVCQ} There are the following IMPLEMENTATION DEFINED vector lengths for the SVE scalable vector registers, Z0-Z31:

- A *Maximum implemented Streaming SVE vector length*.
- A *Maximum implemented Non-streaming SVE vector length*.

For each IMPLEMENTATION DEFINED vector length, all the following apply:

- The smallest architecturally defined maximum length is 128 bits.
- The largest architecturally defined maximum length is 2048 bits.
- The length is a power of two.

I_{GZRPL} There is no requirement for the Maximum implemented Streaming SVE vector length to be greater than or equal to the Maximum Non-streaming implemented SVE vector length.

I_{SGCVT} Where this architecture specification uses the term *Maximum implemented SVE vector length*, it means:

- The Maximum implemented Streaming SVE vector length when the PE is in *Streaming SVE mode*.
- The Maximum implemented Non-streaming SVE vector length otherwise.

See also:

- Z0-Z31.
- Configurable SVE vector lengths.

B1.4.2 Configurable SVE vector lengths

R_{KFTHG} There are the following configurable vector lengths for the SVE scalable vector registers, Z0-Z31:

- *Effective Streaming SVE vector length (SVL)*.
- *Effective Non-streaming SVE vector length (NSVL)*.

I_{QQRNR} SMCR_ELx.LEN requests an SVL. The architecturally defined SVL set is all powers of two from 128 to 2048 bits inclusive. Implementing any subset is permitted, up to and including the *Maximum implemented Streaming SVE vector length*. The subset is not required to be contiguous and is not required to start at 128 bits. Implementing a single SVL is permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{NWYBP} ZCR_ELx.LEN requests an NSVL. The architecturally defined NSVL set is all powers of two from 128 to 2048 bits inclusive. An implementation is required to implement all architecturally defined values up to and including the *Maximum implemented Non-streaming SVE vector length*.

I_{WDKGR} An implementation is permitted to implement an SVL set that does not overlap with the implemented NSVL set.

R_{GWVHP} When SMCR_ELx.LEN requests an SVL, the PE selects an SVL according to the steps described in the SMCR_ELx.LEN field descriptions.

R_{MMCTJ} When ZCR_ELx.LEN requests an NSVL, the PE selects an NSVL according to the steps described in the ZCR_ELx.LEN field descriptions.

I_{VCQBB} Where this architecture specification uses the term *Effective SVE vector length (VL)*, it means:

- SVL when the PE is in *Streaming SVE mode*.
- NSVL otherwise.

R_{PXZTM} When executing at an Exception level where VL is less than the *Maximum implemented SVE vector length*, bits VL and higher in the SVE scalable vector registers, and bits VL ÷ 8 and higher in the SVE predicate registers and FFR, are inaccessible.

R_{YRPDH} When VL is increased, it is CONSTRAINED UNPREDICTABLE whether the previously inaccessible bits that become accessible have:

- A value of zero.
- The value they had before executing at the more constrained length.

I_{HGYPV} VL might increase because of an explicit action such as writing to SMCR_ELx.LEN or ZCR_ELx.LEN, or an implicit action such as taking an exception to an Exception level with a less constrained VL.

I_{PDLWX} When changing VL, the contents in the bits that consistently remain accessible remain the same.

R_{KXKNK} VL at Exception level ELx is 128 bits if all the following are true:

- SVE instructions are disabled or trapped at ELx, or are not available because ELx is using AArch32.
- Floating-point instructions are not trapped at ELx.

R_{DMBPN} When taking an exception to a target Exception level, if SVE, SME, and floating-point instructions are disabled, trapped, or not available at all Exception levels below the target Exception level, for the current Security state, the SVE register state at the target Exception level is preserved.

I_{FMWZZ} When VL is increased, steps must be taken to ensure the newly accessible area does not expose values unrelated to another body of software in a different trust or security scope. This might be achieved by, for example, ensuring that the previously inaccessible bits in the SVE Z, P, and FFR registers, and the SME ZA storage and ZT0 register, are reset to zero before they are next used.

I_{BRMMV} System software provides a maximum VL to lower-privileged software, which might further constrain the VL. However, system software must initialize and context switch values consistent with the maximum VL it provides and should not make assumptions about any smaller length that lower-privileged software is using. For example, if a hypervisor exposes a VL of 512 to a VM, that VM might choose to constrain VL to 256. The hypervisor must still save and restore 512-bit vectors to prevent leakage of values between VMs, because the VM might later raise its SVL to 512 and must not be able to observe values created by other software in the newly visible upper portion of the registers.

See also:

- Maximum implemented SVE vector lengths.
- Streaming SVE mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4.3 Treatment of SVE scalable vector registers

I_{GKWYJ} Unless stated otherwise in an instruction description, SVE instructions treat an SVE scalable vector register as containing one or more *vector elements* of equal size.

I_{CDKJQ} SVE instructions can process vector elements in parallel, unless an instruction description states otherwise.

R_{CJZLM} If the order of operations performed by an SVE instruction on vector or predicate elements has observable significance, elements are processed in increasing element number order.

R_{KHDBN} When an SVE instruction treats an SVE scalable vector register as containing multiple vector elements, the element size is encoded in the opcode of the instruction. The element size is 8, 16, 32, 64, or 128 bits.

R_{WKYLB} When the *Effective SVE vector length* (VL) at the current Exception level is greater than 128 bits, any AArch64 instruction that writes to an 128-bit SIMD&FP register, V0-V31, sets bits [VL-1:128] to zero.

B1.4.4 SVE writes of scalar values to registers

I_{ZDLGD} Certain SVE instructions generate a scalar result that is written to an AArch64 general-purpose register or to element[0] of a vector register.

R_{HNVTM} When an SVE instruction generates a scalar result of width N bits, the instruction places the result in bits [N-1:0] of the destination register.

R_{QCLSH} When an SVE instruction generates a scalar result of width N bits, and N is less than the maximum accessible destination register width RW, the instruction sets bits [RW-1:N] of the destination register to zero.

B1.4.5 Vector predication

R_{JFJZX} There are the following vector predication concepts:

- Predicate-as-mask, if FEAT_SVE or FEAT_SME is implemented.
- Predicate-as-counter, if FEAT_SME2 or FEAT_SVE2p1 is implemented.

I_{HWRFM} The assembler syntax for a predicate register is of the form:

- *Pg* for predicate-as-mask.
- *PNg* for predicate-as-counter.

Pg and PNg refer to the same SVE predicate register.

The remainder of this section contains:

- Predicate-as-mask.
- Predicate-as-counter.
- SVE predicated instructions.

B1.4.5.1 Predicate-as-mask

I_{JHCTW} Most SVE instructions interpret SVE predicate registers as a predicate-as-mask encoding.

R_{XMPLM} A predicate-as-mask encoding is divided into 1-bit, 2-bit, 4-bit, or 8-bit *predicate elements*. Each predicate element corresponds to a vector element in an SVE scalable vector register.

R_{XVRKX} Unless stated otherwise in an instruction description, SVE instructions treat an SVE predicate register as containing one or more predicate elements of equal size.

R_{XCZQR} When an SVE instruction treats an SVE predicate register as containing multiple predicate elements, the element size is encoded in the opcode of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HRPMD} If the lowest-numbered bit of a predicate element is 0, the value of the predicate element is FALSE.

R_{DNMFH} If the lowest-numbered bit of a predicate element is 1, the value of the predicate element is TRUE.

R_{LTGQC} For SVE instructions that generate predicate-as-mask encodings, if all the following are true, all bits except the lowest-numbered bit of each destination predicate element are set to zero:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

R_{HBMLS} For SVE instructions that consume predicate-as-mask encodings, if all the following are true, all bits except the lowest-numbered bit of each source predicate element are ignored on reads:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

B1.4.5.2 Predicate-as-counter

R_{DSFKR} SME2 and SVE2.1 multi-vector instructions interpret bits[15:0] of SVE predicate registers as a *predicate-as-counter encoding*:

Table B1-4 Predicate-as-counter encoding

Bit[15] Invert	Bits[14:(LSZ+1)][a] Element count[b]	Bits[LSZ:0] Element size	Meaning
`0bX`	`0bXXXXXXXXXXXXXXX`	`0b1`	Byte-size elements
`0bX`	`0bXXXXXXXXXXXXXX`	`0b10`	Halfword-size elements
`0bX`	`0bXXXXXXXXXXXXX`	`0b100`	Word-size elements
`0bX`	`0bXXXXXXXXXXXX`	`0b1000`	Doubleword-size elements
`0bX`	`0bXXXXXXXXXXXX`	`0b0000`	Any-size elements. All FALSE predicate.

a. LSZ = the number of trailing zeroes in the element size field.
b. The 4 most significant bits are reserved.

Element size, bits[LSZ:0]

For values shown in Table B1-4 other than `0b0000`:

- Number of trailing zeroes = $\log_2$(element size in bytes).
  For example, 2 trailing zeroes = $\log_2$(4 bytes). This means word-size elements.

Element count, bits[14:(LSZ+1)]

An unsigned integer value:

- The number of consecutive elements, starting from element 0, that are TRUE or FALSE according to bit[15].

Invert, bit[15]

`0b0` [14:(LSZ+1)] is holding the number of TRUE elements. All other elements are FALSE.

`0b1` [14:(LSZ+1)] is holding the number of FALSE elements. All other elements are TRUE.

I_{ZPLKP} The maximum number of elements that the predicate-as-counter encoding can count is four vectors of byte elements, where the length of each vector is the largest architecturally defined length, 2048 bits:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table B1-5

Bit[15] Invert	Bits[14:(LSZ+1)] Element count	Bits[LSZ:0] Element size	Total number of bits
1 bit	4 reserved bits. 10 count bits. 10 count bits can count 1024 elements, which is the number of byte elements in four 2048-bit long vectors.	1 bit	16 bits

D_{BJMYH} The canonical all-FALSE predicate-as-counter encoding has an element count of zero, with the invert bit set to 0 and an element size field set to `0b0000`.

D_{JGSYR} The canonical all-TRUE predicate-as-counter encoding has an element count of zero, with the invert bit set to 1 and a nonzero element size field determined by the generating instruction.

R_{YCRMW} If the *Effective SVE vector length* (VL) is greater than 128 bits, then an instruction generating a predicate-as-counter encoding sets bits 16 and higher to zero.

R_{SGVTC} If VL is greater than 128 bits, then an instruction consuming a predicate-as-counter encoding ignores bits 16 and higher.

R_{YSTVC} An instruction consuming a predicate-as-counter encoding uses the least significant bits of the element count field required to count the number of bytes in VL times four. The instruction ignores the more significant bits in the element count field. Table B1-6 gives an example for VL = 512 bits and byte-size elements:

Table B1-6

Bit[15]Invert	Bits[14:(LSZ+1)] Element count	Bits[LSZ:0] Element size
1 bit	4 reserved bits, bits[14:11]. The instruction ignores these. 2 bits, bits[10:9]. The instruction ignores these. 8 count bits, bits[8:1]. The instruction uses these. 8 count bits can count 256 elements, which is the number of byte elements in four 512-bit long vectors.	1 bit

See also:

- P0-P15.
- Configurable SVE vector lengths.

B1.4.5.3 SVE predicated instructions

I_{VKHDR} If an instruction supports predication, it is known as a *predicated instruction*.

I_{FNFRN} The predicate operand that is used to determine the *Active elements* of a predicated instruction is known as the *Governing predicate*.

I_{SVYXB} An instruction that does not have a Governing predicate operand and implicitly treats all other vector and predicate elements as Active is known as an *unpredicated instruction*.

I_{KNKBN} Many predicated instructions can only use P0-P7 as the Governing predicate.

R_{LZVFJ} When a Governing predicate element is TRUE, the corresponding element in other vector or predicate operands is an Active element.

R_{CNFLG} When a Governing predicate element is FALSE, the corresponding element in other vector or predicate operands is an *Inactive element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CBYJH} Predicated instructions process Active elements.

R_{LDXSF} Predicated instructions do not process Inactive elements.

R_{GJLPZ} Unpredicated instructions process all elements in their vector or predicate operands.

R_{WLQBD} When a predicated instruction writes to a vector destination register or a predicate destination register, one of the following happens:

- The Inactive elements in the destination register are set to zero.
- The Inactive elements in the destination register retain their previous value.

I_{QBHRN} Zeroing predication is performed when the Inactive elements in the destination register are set to zero.

I_{YPYRF} Merging predication is performed when Inactive elements in the destination register retain their previous value.

B1.4.6 Streaming SVE mode

R_{XDPXS} A PE with FEAT_SME has *Streaming SVE mode*.

I_{ZTTNW} In Streaming SVE mode:

- SME and SME2 instructions access *Streaming SVE register state*, which comprises:
  - SVE scalable vector registers, Z0 to Z31.
  - SVE predicate registers, P0 to P15.
  - SVE First Fault register, FFR, if FEAT_SME_FA64 is implemented and enabled at the current Exception level.
- PSTATE.ZA can be toggled, to enable and disable the *ZA storage* and *ZT0 register* as required.

R_{RSWFQ} When the Effective value of PSTATE.SM is changed by any method from 0 to 1, an entry to Streaming SVE mode is performed, and all implemented bits of Streaming SVE register state are set to zero.

R_{KFRQZ} When the Effective value of PSTATE.SM is changed by any method from 1 to 0, an exit from Streaming SVE mode is performed, and in the newly-entered mode, all implemented bits of the SVE scalable vector registers, SVE predicate registers, and FFR, are set to zero.

I_{NHNFF} A *legal* instruction is an instruction that the architecture permits execution of when PSTATE.{SM, ZA} are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

I_{HZFSG} An *illegal* instruction is an instruction whose attempted execution by a PE when PSTATE.SM and PSTATE.ZA are not in the required state causes an *SME illegal instruction exception* to be taken, unless its execution at the current Exception level is prevented by a higher-priority configurable trap or enable.

R_{VJZBC} In Streaming SVE mode:

- Streaming SVE register state is valid, and SME and SME2 instructions that access it are legal.
- When PSTATE.ZA is 1, SME and SME2 instructions that access ZA storage or the ZT0 register are legal.
- SME and SME2 instructions that do not access ZA storage or the ZT0 register are legal.
- Legal instructions that access the SVE scalar vector registers or their correspondingly numbered SIMD&FP registers access Streaming SVE register state.
- The following are illegal if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:
  - Some SVE and SVE2 instructions. The individual instruction descriptions in SVE Instruction Descriptions describe which are illegal.
  - All those instructions that call the AArch64.CheckFPAdvSIMDEnabled() function.

R_{XBBFD} When the PE is not in Streaming SVE mode:

- Streaming SVE register state is not valid, and SME and SME2 instructions that access it are illegal.
- When PSTATE.ZA is 1:
  - SME `LDR` (array vector), `STR` (array vector), and `ZERO` (tiles) instructions that access ZA storage are legal. All other instructions that access ZA storage are illegal.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— SME2 `LDR` (table), `STR` (table), and `ZERO` (table) instructions that access the ZT0 register are legal. All other instructions that access the ZT0 register are illegal.

I_{TDSPN} In Streaming SVE mode, the following instructions might be significantly delayed if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

- Instructions which are dependent on results generated from vector or SIMD&FP register sources written to a general-purpose destination register, a predicate destination register, or the NZCV condition flags.

The Operational information sections of instruction descriptions in A64 Advanced SIMD and Floating-point Instruction Descriptions and SVE Instruction Descriptions describe which instructions are affected.

I_{MHTLZ} When the Effective value of PSTATE.SM is changed by any method from 0 to 1 or from 1 to 0, the FPSR is set to the value `0x0000_0000_0800_009f`, setting all the cumulative status bits to 1.

I_{YTZVD} Statements which refer to the value of the SVE scalable vector registers, Z0-Z31, implicitly also refer to the lower bits of those registers accessed by the SIMD&FP register names V0-V31, Q0-Q31, D0-D31, S0-S31, H0-H31, and B0-B31.

See also:

- Registers in AArch64 Execution state.
- About PSTATE.ZA.
- Accessing PSTATE fields at EL0.
- Floating-point behaviors in Streaming SVE mode.

B1.4.7 About PSTATE.ZA

I_{HVKTL} PSTATE.ZA enables ZA storage and, if FEAT_SME2 is implemented, the ZT0 register.

R_{JHMYL} When PSTATE.ZA is 0, the contents of ZA storage and the ZT0 register are not valid, and SME and SME2 instructions that access them are *illegal*.

R_{SFWMY} When PSTATE.ZA is 1, the contents of ZA storage and the ZT0 register are valid and are retained by hardware irrespective of whether the PE is in *Streaming SVE mode*, and:

- SME `LDR` (array vector), `STR` (array vector), and `ZERO` (tiles) instructions that access ZA storage are legal.
- SME2 `LDR` (table), `STR` (table), and `ZERO` (table) instructions that access the ZT0 register are legal.
- When the PE is in Streaming SVE mode, all other SME and SME2 instructions that access ZA storage or the ZT0 register are legal.

R_{YRZRM} When PSTATE.ZA is changed by any method from 0 to 1, all implemented bits of ZA storage and the ZT0 register are set to zero.

I_{LRDZR} When PSTATE.ZA is changed from 1 to 0, there is no architecturally defined effect on ZA storage and the ZT0 register, because the contents of ZA storage and the ZT0 register cannot be observed when PSTATE.ZA is 0.

I_{QWCJS} When PSTATE.ZA is changed from 0 to 1, or from 1 to 0, there is no effect on the SVE scalable vector registers, SVE predicate registers, and FPSR if PSTATE.SM is not changed.

See also:

- Streaming SVE mode.
- ZA storage.
- SME2 ZT0 register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4.8 ZA storage

R_{FFWLL} A PE with FEAT_SME has *ZA storage*.

D_{JBVYJ} There are the following terms for the number of elements in a vector of SVL bits:

SVL_B The number of byte elements, SVL ÷ 8.

SVL_H The number of halfword elements, SVL ÷ 16.

SVL_S The number of word elements, SVL ÷ 32.

SVL_D The number of doubleword elements, SVL ÷ 64.

SVL_Q The number of quadword elements, SVL ÷ 128.

R_{SSXPL} ZA storage is architectural register state consisting of a two-dimensional ZA array of [$SVL_B \times SVL_B$] bytes.

See also:

- About PSTATE.ZA.
- Accessing PSTATE fields at EL0.
- Streaming SVE mode.

B1.4.9 ZA array vector access

R_{FFWNB} The *ZA* array can be accessed as vectors of SVL bits.

D_{PPPCM} An untyped vector access to the *ZA* array is represented by ZA[*N*], where *N* is in the range 0 to SVL_B-1 inclusive.

D_{DTVZN} In SME `LDR` (array vector) and `STR` (array vector) instructions, an untyped *ZA array vector* is selected by the sum of a 32-bit general-purpose vector select register W*v* and an immediate vector select offset *offs*, modulo SVL_B.

D_{YXHFR} The preferred disassembly for an untyped *ZA* array vector is ZA[W*v*, *offs*], where *offs* is an immediate in the range 0-15 inclusive.

D_{CRJPC} The *ZA* array can be accessed as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.

D_{WMVZT} An elementwise vector access to the *ZA* array is indicated by appending a vector index "[*N*]" to the *ZA* array name and element size qualifier, where *N* is in the range 0 to SVL_B-1 inclusive, as follows:

- An 8-bit element vector access to the *ZA* array is represented by ZA.B[*N*].
- A 16-bit element vector access to the *ZA* array is represented by ZA.H[*N*].
- A 32-bit element vector access to the *ZA* array is represented by ZA.S[*N*].
- A 64-bit element vector access to the *ZA* array is represented by ZA.D[*N*].
- A 128-bit element vector access to the *ZA* array is represented by ZA.Q[*N*].

B1.4.10 ZA tile access

D_{VSVMX} A *ZA tile* is a square, two-dimensional sub-array of elements within the *ZA* array.

I_{WLRTV} Depending on the element size with which it is accessed, the ZA array is treated as containing one or more *ZA* tiles, as described in the following sections.

D_{DWMYT} A *ZA* tile is indicated by appending the tile number to the *ZA* name.

D_{ZGBHT} A *ZA tile slice* is a one-dimensional set of horizontally or vertically contiguous elements within a *ZA* tile.

R_{PZNWB} A vector access to a tile reads or writes a *ZA* tile slice.

I_{NFXHH} A *ZA* tile can be accessed as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{YZDBS} A *ZA* tile can be accessed as horizontal slices of SVL bits.

R_{GPVSZ} A *ZA* tile is accessed as horizontal slices if the V field in the accessing instruction opcode is 0.

D_{TRHTX} An access to horizontal tile slices is indicated by an "H" suffix on the *ZA* tile name.

I_{HBYTT} A *ZA* tile can be accessed as vertical slices of SVL bits.

R_{GPPPK} A *ZA* tile is accessed as vertical slices if the V field in the accessing instruction opcode is 1.

D_{WSBVG} An access to vertical tile slices is indicated by a suffix on the *ZA* tile name.

R_{TWWTL} In SME instructions, the tile slice is selected by the sum of a 32-bit general-purpose slice index register W*s* and an immediate slice index offset *offs*, modulo the number of slices in the named tile.

B1.4.10.1 Accessing an 8-bit element ZA tile

D_{HMSNH} An 8-bit element *ZA* tile is indicated by a ".B" qualifier following the tile name.

D_{NLCNH} There is a single tile named ZA0.B which consists of $[SVL_B \times SVL_B]$ 8-bit elements and occupies all of the *ZA* storage.

R_{NBSMJ} An access to a horizontal or vertical 8-bit element ZA tile slice reads or writes SVL_B 8-bit elements.

D_{NMHLM} An access to a horizontal or vertical 8-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *N* is in the range 0 to SVL_B-1 inclusive:

- ZA0H.B[*N*] indicates a horizontal 8-bit element *ZA* tile slice selection.
- ZA0V.B[*N*] indicates a vertical 8-bit element *ZA* tile slice selection.

I_{JVTNY} Horizontal and vertical ZA0.B slice accesses are illustrated in the following diagram for SVL of 256 bits:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

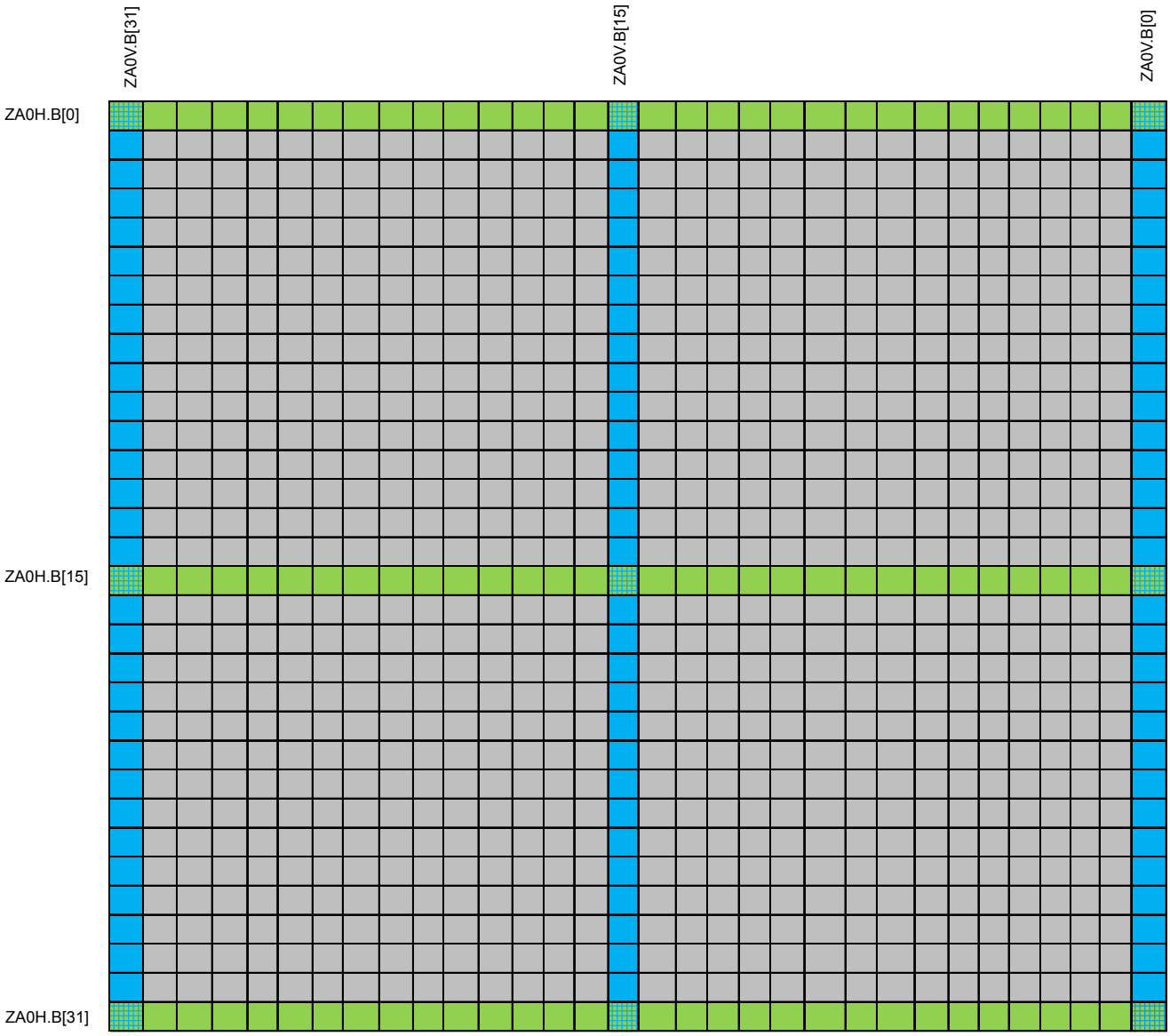

Figure B1-4 Horizontal and vertical ZA0.B slice for SVL of 256

R_{DCSDX} An access to the horizontal slice ZA0H.B[*N*] reads or writes the SVL_B bytes in *ZA* array vector ZA.B[*N*].

R_{FHYSQ} An access to the vertical slice ZA0V.B[*N*] reads or writes the 8-bit element [*N*] within each horizontal slice of ZA0.B.

D_{CDDVV} The preferred disassembly is:

- ZA0H.B[W*s*, *offs*], for a horizontal 8-bit element *ZA* tile slice selection.
- ZA0V.B[W*s*, *offs*], for a vertical 8-bit element *ZA* tile slice selection.

Where *offs* is an immediate in the range 0-15 inclusive.

B1.4.10.2 Accessing a 16-bit element ZA tile

D_{LNXPD} A 16-bit element *ZA* tile is indicated by a ".H" qualifier following the tile name.

D_{GWZDM} There are two tiles named ZA0.H and ZA1.H. Each tile consists of [$SVL_H \times SVL_H$] 16-bit elements, and occupies half of the *ZA* storage.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NMGXG} An access to a horizontal or vertical 16-bit element ZA tile slice reads or writes SVL_H 16-bit elements.

D_{DHKMC} An access to a horizontal or vertical 16-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is 0 or 1, and *N* is in the range 0 to SVL_H-1 inclusive:

- ZA*t*H.H[*N*] indicates a horizontal 16-bit element *ZA* tile slice selection.
- ZA*t*V.H[*N*] indicates a vertical 16-bit element *ZA* tile slice selection.

I_{ZSWJW} Horizontal and vertical ZA*t*.H slice accesses, where *t* is 0 or 1, are illustrated in the following diagram for SVL of 256 bits:

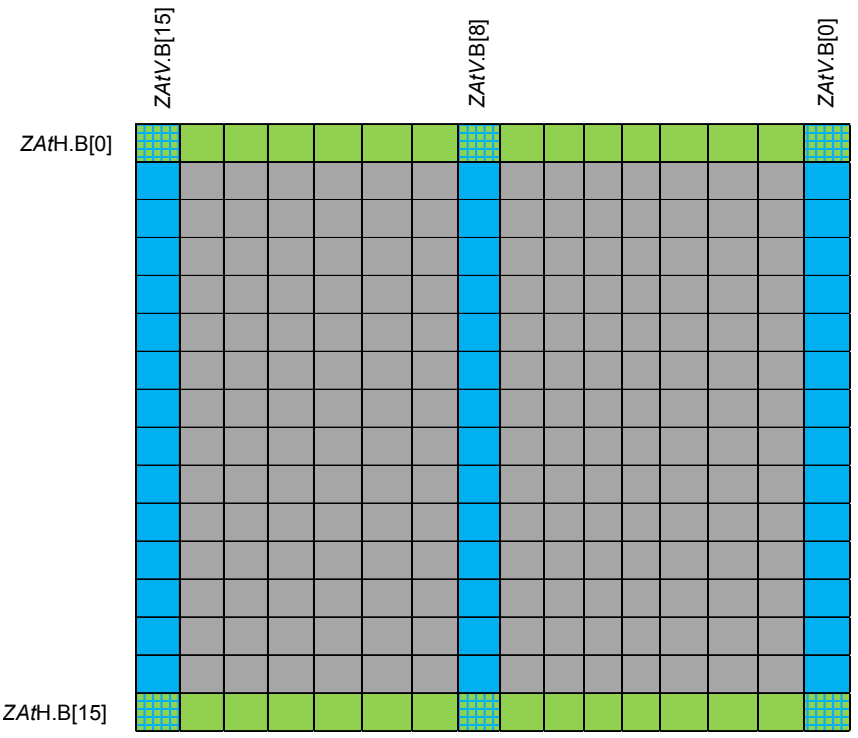

Figure B1-5 Horizontal and vertical ZAt.H slice for SVL of 256

R_{BTLQC} An access to the horizontal slice ZA*t*H.H[*N*] reads or writes the SVL_H 16-bit elements in *ZA* array vector ZA.H[*t* + 2 * *N*].

R_{NGJBJ} An access to the vertical slice ZA*t*V.H[*N*] reads or writes the 16-bit element [*N*] within each horizontal slice of ZA*t*.H.

D_{RHQJT} The preferred disassembly is as follows:

- ZA*t*H.H[W*s*, *offs*], for a horizontal 16-bit element *ZA* tile slice selection.
- ZA*t*V.H[W*s*, *offs*], for a vertical 16-bit element *ZA* tile slice selection.

Where *t* is 0 or 1, and *offs* is an immediate in the range 0-7 inclusive.

B1.4.10.3 Accessing a 32-bit element ZA tile

D_{HBKZV} A 32-bit element *ZA* tile is indicated by a ".S" qualifier following the tile name.

D_{RDRRT} There are four tiles named ZA0.S, ZA1.S, ZA2.S, and ZA3.S. Each tile consists of [$SVL_S \times SVL_S$] 32-bit elements, and occupies a quarter of the *ZA* storage.

R_{XFPPL} An access to a horizontal or vertical 32-bit element ZA tile slice reads or writes SVL_S 32-bit elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D_{JFPSJ} An access to a horizontal or vertical 32-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is 0, 1, 2, or 3, and *N* is in the range 0 to SVL_S-1 inclusive:

- ZA*t*H.S[*N*] indicates a horizontal 32-bit element *ZA* tile slice selection.
- ZA*t*V.S[*N*] indicates a vertical 32-bit element *ZA* tile slice selection.

I_{SZXZR} Horizontal and vertical ZA*t*.S slice accesses, where *t* is 0, 1, 2, or 3, are illustrated in the following diagram for SVL of 256 bits:

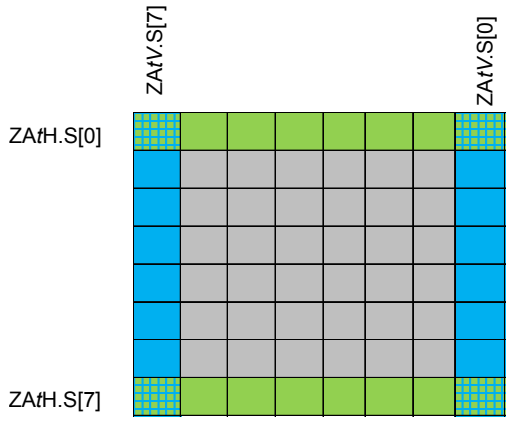

Figure B1-6 Horizontal and vertical ZAt.S slice for SVL of 256

R_{JBJZY} An access to the horizontal slice ZA*t*H.S[*N*] reads or writes the SVL_S 32-bit elements in *ZA* array vector ZA.S[*t* + 4 * *N*].

R_{GBYSJ} An access to the vertical slice ZA*t*V.S[*N*] reads or writes the 32-bit element [*N*] within each horizontal slice of ZA*t*.S.

D_{LQLJH} The preferred disassembly is:

- ZA*t*H.S[W*s*, *offs*], for a horizontal 32-bit element *ZA* tile slice selection.
- ZA*t*V.S[W*s*, *offs*], for a vertical 32-bit element *ZA* tile slice selection.

Where *t* is 0, 1, 2, or 3, and *offs* is 0, 1, 2, or 3.

B1.4.10.4 Accessing a 64-bit element ZA tile

D_{TWMMM} A 64-bit element *ZA* tile is indicated by a ".D" qualifier following the tile name.

D_{THPSD} There are eight tiles named ZA0.D, ZA1.D, ZA2.D, ZA3.D, ZA4.D, ZA5.D, ZA6.D, and ZA7.D. Each tile consists of [$SVL_D \times SVL_D$] 64-bit elements, and occupies an eighth of the *ZA* storage.

R_{ZXYBQ} An access to a horizontal or vertical 64-bit element ZA tile slice reads or writes SVL_D 64-bit elements.

D_{DCXSX} An access to a horizontal or vertical 64-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is in the range 0-7 inclusive, and *N* is in the range 0 to SVL_D-1 inclusive:

- ZA*t*H.D[*N*] indicates a horizontal 64-bit element *ZA* tile slice selection.
- ZA*t*V.D[*N*] indicates a vertical 64-bit element *ZA* tile slice selection.

I_{LGJZC} Horizontal and vertical ZA*t*.D slice accesses, where *t* is in the range 0-7 inclusive, are illustrated in the following diagram for SVL of 256 bits:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

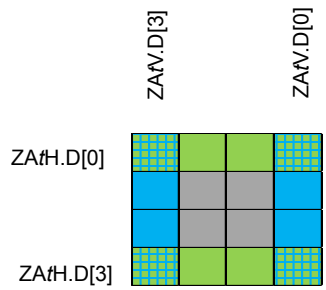

Figure B1-7 Horizontal and vertical ZAt.D slice for SVL of 256

R_{CVVJK} An access to the horizontal slice ZA*t*H.D[*N*] reads or writes the SVL_D 64-bit elements in *ZA* array vector ZA.D[*t* + 8 * *N*].

R_{JYQKK} An access to the vertical slice ZA*t*V.D[*N*] reads or writes the 64-bit element [*N*] within each horizontal slice of ZA*t*.D.

D_{MQQPX} The preferred disassembly is:

- ZA*t*H.D[W*s*, *offs*], for a horizontal 64-bit element *ZA* tile slice selection.
- ZA*t*V.D[W*s*, *offs*], for a vertical 64-bit element *ZA* tile slice selection.

Where *t* is in the range 0-7 inclusive, and *offs* is 0 or 1.

B1.4.10.5 Accessing a 128-bit element ZA tile

D_{GZDSH} A 128-bit element *ZA* tile is indicated by a ".Q" qualifier following the tile name.

D_{RPMJL} There are 16 tiles named ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, and ZA15.Q. Each tile consists of [$SVL_Q \times SVL_Q$] 128-bit elements, and occupies 1/16 of the *ZA* storage.

R_{QGHPF} An access to a horizontal or vertical 128-bit element ZA tile slice reads or writes SVL_Q 128-bit elements.

D_{RLQKW} An access to a horizontal or vertical 128-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is in the range 0-15 inclusive, and *N* is in the range 0 to SVL_Q-1 inclusive:

- ZA*t*H.Q[*N*] indicates a horizontal 128-bit element *ZA* tile slice selection.
- ZA*t*V.Q[*N*] indicates a vertical 128-bit element *ZA* tile slice selection.

I_{YQPWS} Horizontal and vertical ZA*t*.Q slice accesses, where *t* is in the range 0-15 inclusive, are illustrated in the following diagram for SVL of 256 bits:

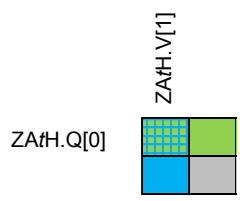

Figure B1-8 Horizontal and vertical ZAt.Q slice for SVL of 256

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{PJTQJ} An access to the horizontal slice ZA*t*H.Q[*N*] reads or writes the SVL_Q 128-bit elements in *ZA* array vector ZA.Q[*t* + 16 * *N*].

R_{TRJFZ} An access to the vertical slice ZA*t*V.Q[*N*] reads or writes the 128-bit element [*N*] within each horizontal slice of ZA*t*.Q.

D_{VCLJP} The preferred disassembly is:

- ZA*t*H.Q[W*s*, 0], for a horizontal 128-bit element *ZA* tile slice selection.
- ZA*t*V.Q[W*s*, 0], for a vertical 128-bit element *ZA* tile slice selection.

Where *t* is in the range 0-15 inclusive, and the slice index offset is always zero.

B1.4.11 ZA storage layout

B1.4.11.1 ZA array vector and tile slice mappings

I_{PYTLW} Each horizontal tile slice corresponds to one *ZA* array vector.
The horizontal slice mappings for all tile sizes are illustrated by this table:

ZA Array Vector	8-bit element Tile Horizontal Slice	16-bit element Tile Horizontal Slice	32-bit element Tile Horizontal Slice	64-bit element Tile Horizontal Slice	128-bit element Tile Horizontal Slice
ZA[0]	ZA0H.B[0]	ZA0H.H[0]	ZA0H.S[0]	ZA0H.D[0]	ZA0H.Q[0]
ZA[1]	ZA0H.B[1]	ZA1H.H[0]	ZA1H.S[0]	ZA1H.D[0]	ZA1H.Q[0]
ZA[2]	ZA0H.B[2]	ZA0H.H[1]	ZA2H.S[0]	ZA2H.D[0]	ZA2H.Q[0]
ZA[3]	ZA0H.B[3]	ZA1H.H[1]	ZA3H.S[0]	ZA3H.D[0]	ZA3H.Q[0]
ZA[4]	ZA0H.B[4]	ZA0H.H[2]	ZA0H.S[1]	ZA4H.D[0]	ZA4H.Q[0]
ZA[5]	ZA0H.B[5]	ZA1H.H[2]	ZA1H.S[1]	ZA5H.D[0]	ZA5H.Q[0]
ZA[6]	ZA0H.B[6]	ZA0H.H[3]	ZA2H.S[1]	ZA6H.D[0]	ZA6H.Q[0]
ZA[7]	ZA0H.B[7]	ZA1H.H[3]	ZA3H.S[1]	ZA7H.D[0]	ZA7H.Q[0]
ZA[8]	ZA0H.B[8]	ZA0H.H[4]	ZA0H.S[2]	ZA0H.D[1]	ZA8H.Q[0]
ZA[9]	ZA0H.B[9]	ZA1H.H[4]	ZA1H.S[2]	ZA1H.D[1]	ZA9H.Q[0]
ZA[10]	ZA0H.B[10]	ZA0H.H[5]	ZA2H.S[2]	ZA2H.D[1]	ZA10H.Q[0]
ZA[11]	ZA0H.B[11]	ZA1H.H[5]	ZA3H.S[2]	ZA3H.D[1]	ZA11H.Q[0]
ZA[12]	ZA0H.B[12]	ZA0H.H[6]	ZA0H.S[3]	ZA4H.D[1]	ZA12H.Q[0]
ZA[13]	ZA0H.B[13]	ZA1H.H[6]	ZA1H.S[3]	ZA5H.D[1]	ZA13H.Q[0]
ZA[14]	ZA0H.B[14]	ZA0H.H[7]	ZA2H.S[3]	ZA6H.D[1]	ZA14H.Q[0]
ZA[15]	ZA0H.B[15]	ZA1H.H[7]	ZA3H.S[3]	ZA7H.D[1]	ZA15H.Q[0]
if applicable ZA[16] to ZA[SVL_B-1]	. . .	. . .	. . .	. . .	. . .

B1.4.11.2 Tile mappings

I_{YVYJP} The smallest *ZA* tile granule is the 128-bit element tile. When the *ZA* storage is viewed as an array of tiles, the larger 64-bit, 32-bit, 16-bit, and 8-bit element tiles overlap multiple 128-bit element tiles as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Tile	Overlaps
ZA0.B	ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, ZA15.Q
ZA0.H	ZA0.Q, ZA2.Q, ZA4.Q, ZA6.Q, ZA8.Q, ZA10.Q, ZA12.Q, ZA14.Q
ZA1.H	ZA1.Q, ZA3.Q, ZA5.Q, ZA7.Q, ZA9.Q, ZA11.Q, ZA13.Q, ZA15.Q
ZA0.S	ZA0.Q, ZA4.Q, ZA8.Q, ZA12.Q
ZA1.S	ZA1.Q, ZA5.Q, ZA9.Q, ZA13.Q
ZA2.S	ZA2.Q, ZA6.Q, ZA10.Q, ZA14.Q
ZA3.S	ZA3.Q, ZA7.Q, ZA11.Q, ZA15.Q
ZA0.D	ZA0.Q, ZA8.Q
ZA1.D	ZA1.Q, ZA9.Q
ZA2.D	ZA2.Q, ZA10.Q
ZA3.D	ZA3.Q, ZA11.Q
ZA4.D	ZA4.Q, ZA12.Q
ZA5.D	ZA5.Q, ZA13.Q
ZA6.D	ZA6.Q, ZA14.Q
ZA7.D	ZA7.Q, ZA15.Q

I_{WGZBT} The architecture permits concurrent use of different element size tiles.

B1.4.11.3 Horizontal tile slice mappings

I_{NJJXW} The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for a 32-bit element and 64-bit element horizontal tile slice.

Each small numbered square represents 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

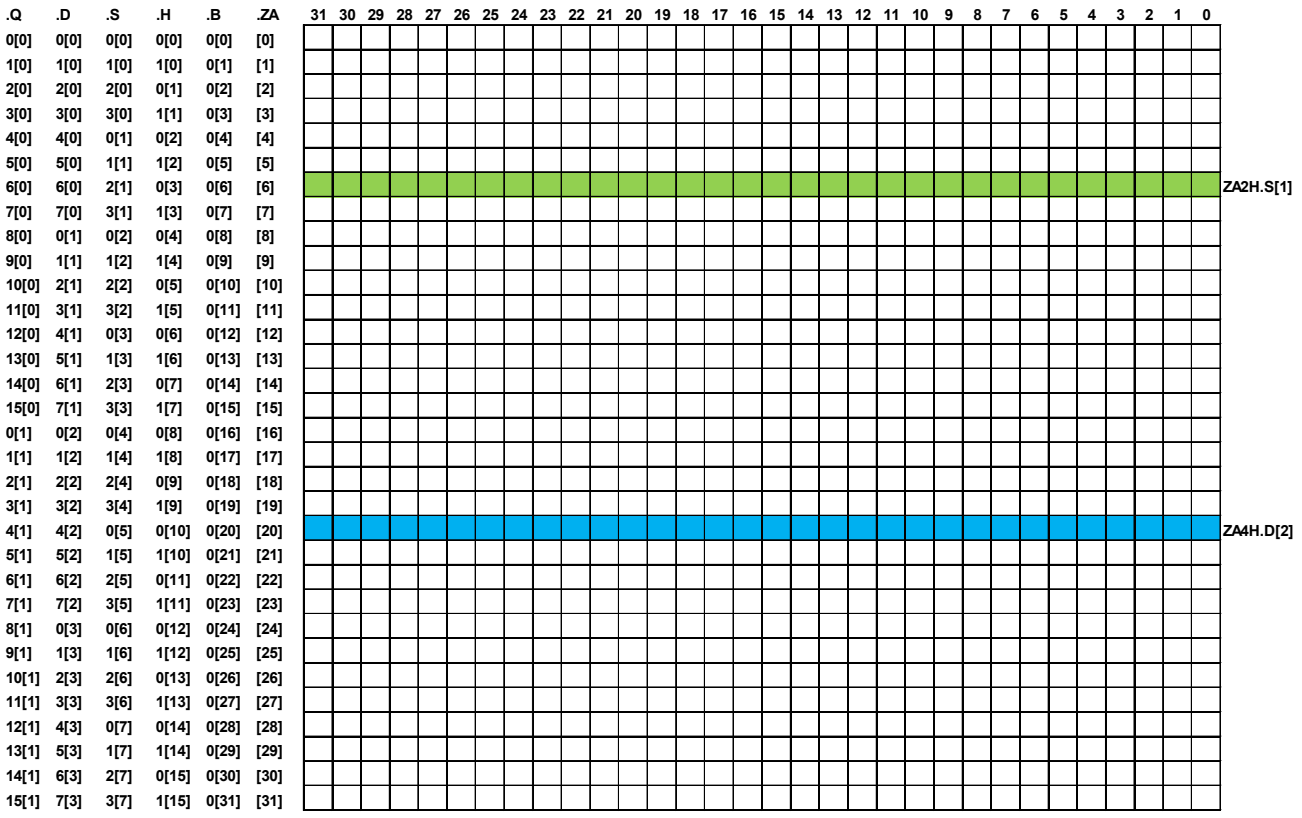

Figure B1-9 ZA storage mapping for SVL of 256 bits: 32-bit and 64-bit element horizontal tile slice

An SME vector load, store, or move instruction that accesses horizontal tile slices ZA2H.S[1] or ZA4H.D[2] treats the slices as vectors with the following layout:

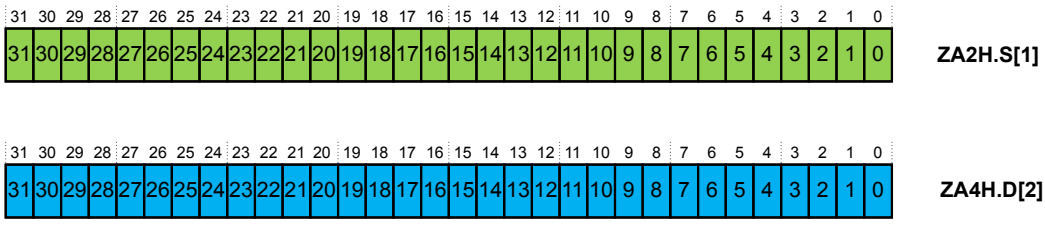

Figure B1-10 ZA2H.S[1] and ZA4H.D[2] vector layout

B1.4.11.4 Vertical tile slice mappings

I_{TNCCV}

The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for a 32-bit element and 64-bit element vertical tile slice.

Each small numbered square represents 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

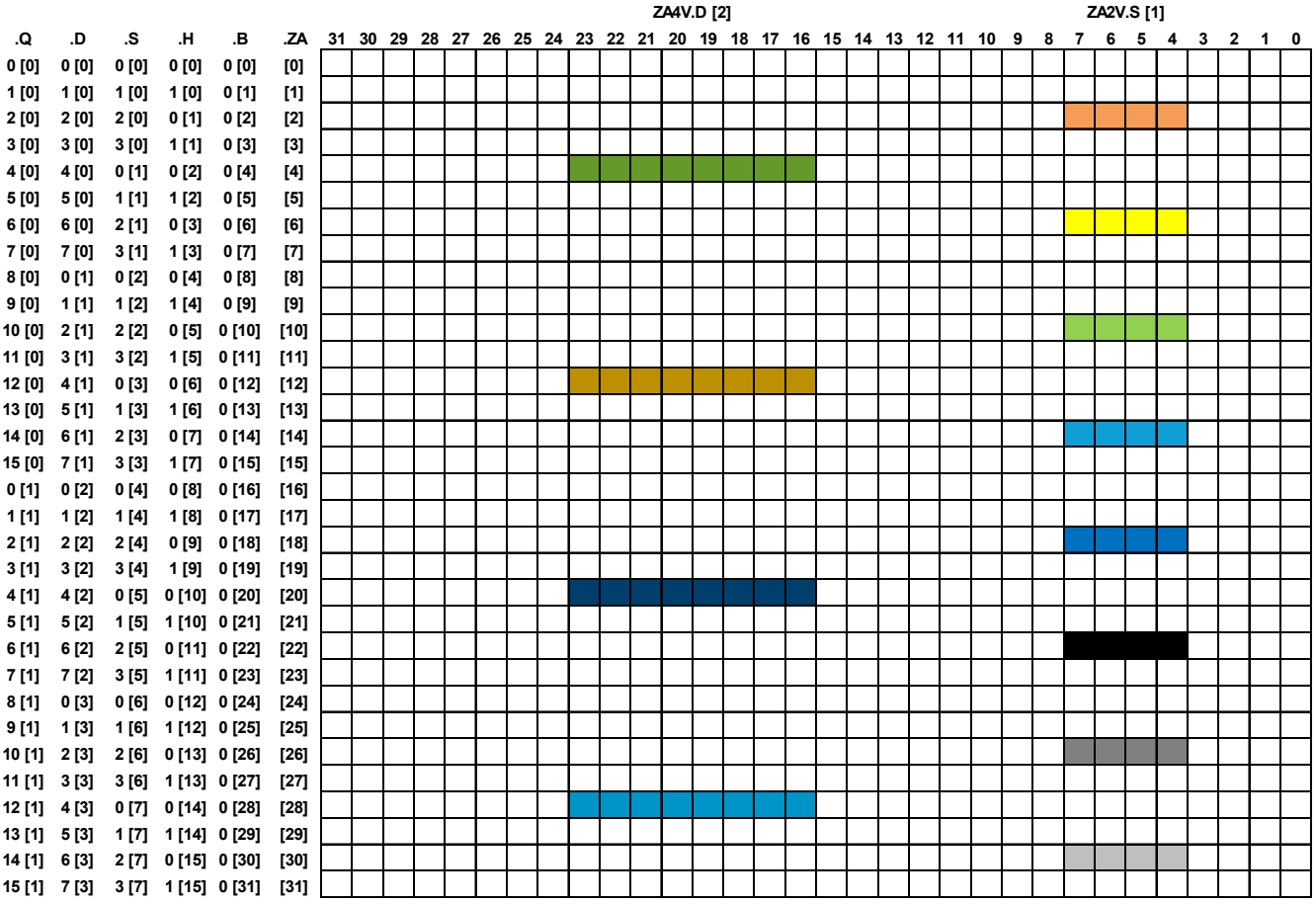

Figure B1-11 ZA storage mapping for SVL of 256 bits: 32-bit and 64-bit element vertical tile slices

An SME vector load, store, or move instruction which accesses vertical tile slices ZA2V.S[1] or ZA4V.D[2] treats the slices as vectors with the following layout:

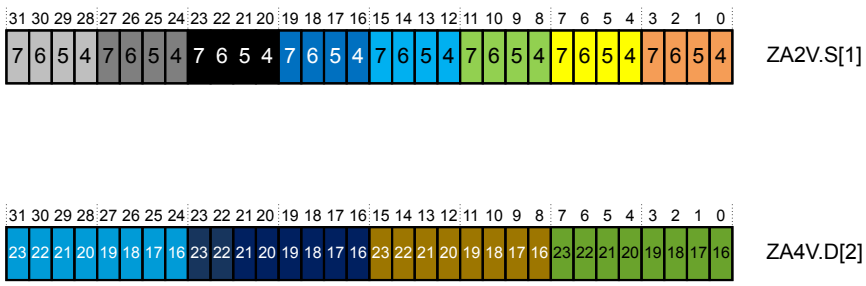

Figure B1-12 ZA2V.S[1] and ZA4V.D[2] vector layout

B1.4.11.5 Mixed horizontal and vertical tile slice mappings

ICGXPJ

The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for various element size tiles, horizontal tile slices, and vertical tile slices.

Each small square represents 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

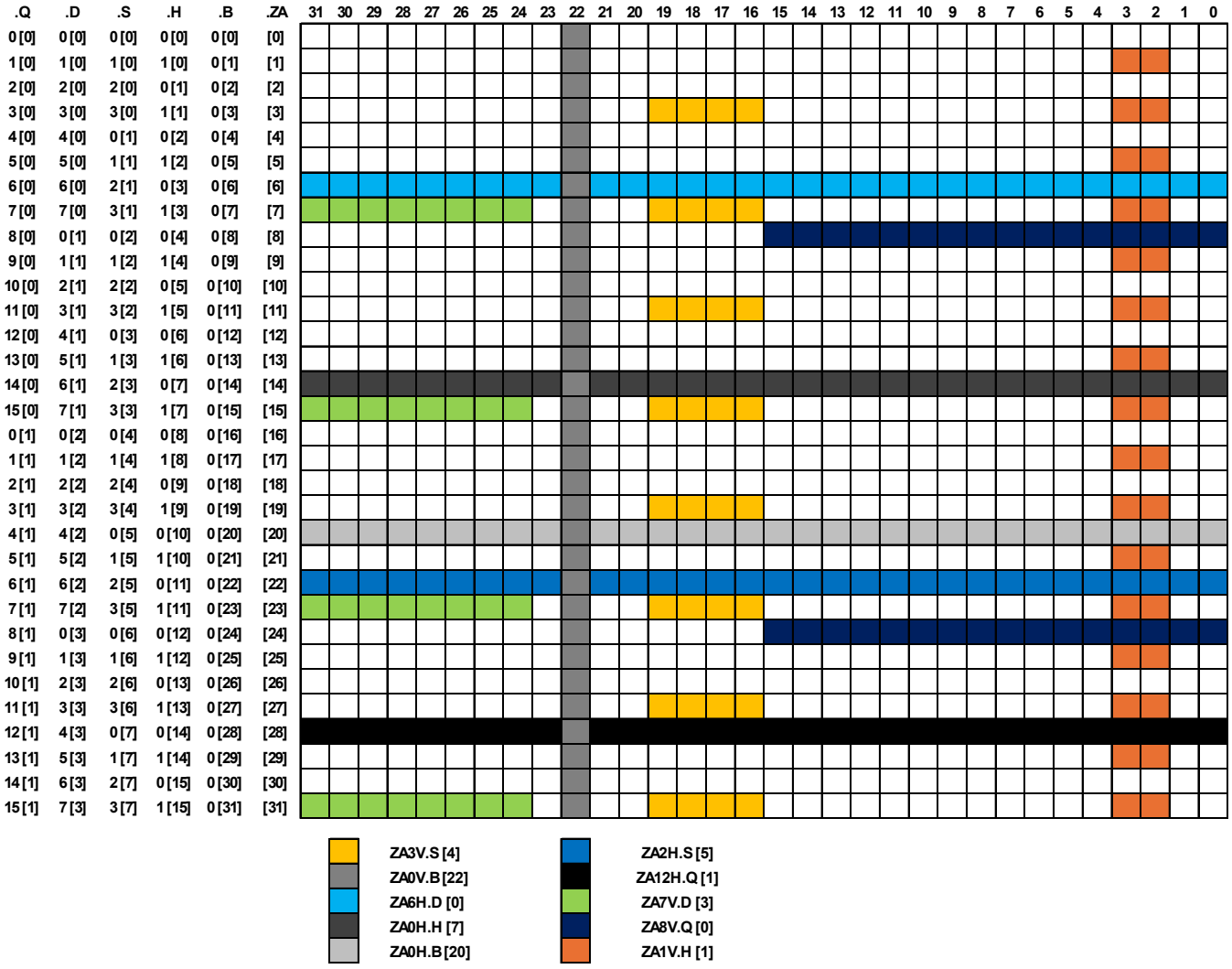

Figure B1-13 ZA storage mapping for SVL of 256 bits: various vertical and tile slices

I_{HVFMB} It is possible to simultaneously use non-overlapping *ZA* array vectors within tiles of differing element sizes. For example, tiles ZA1.H, ZA0.S, and ZA2.D have no *ZA* array vectors in common, as illustrated in the following diagram for SVL of 256 bits:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

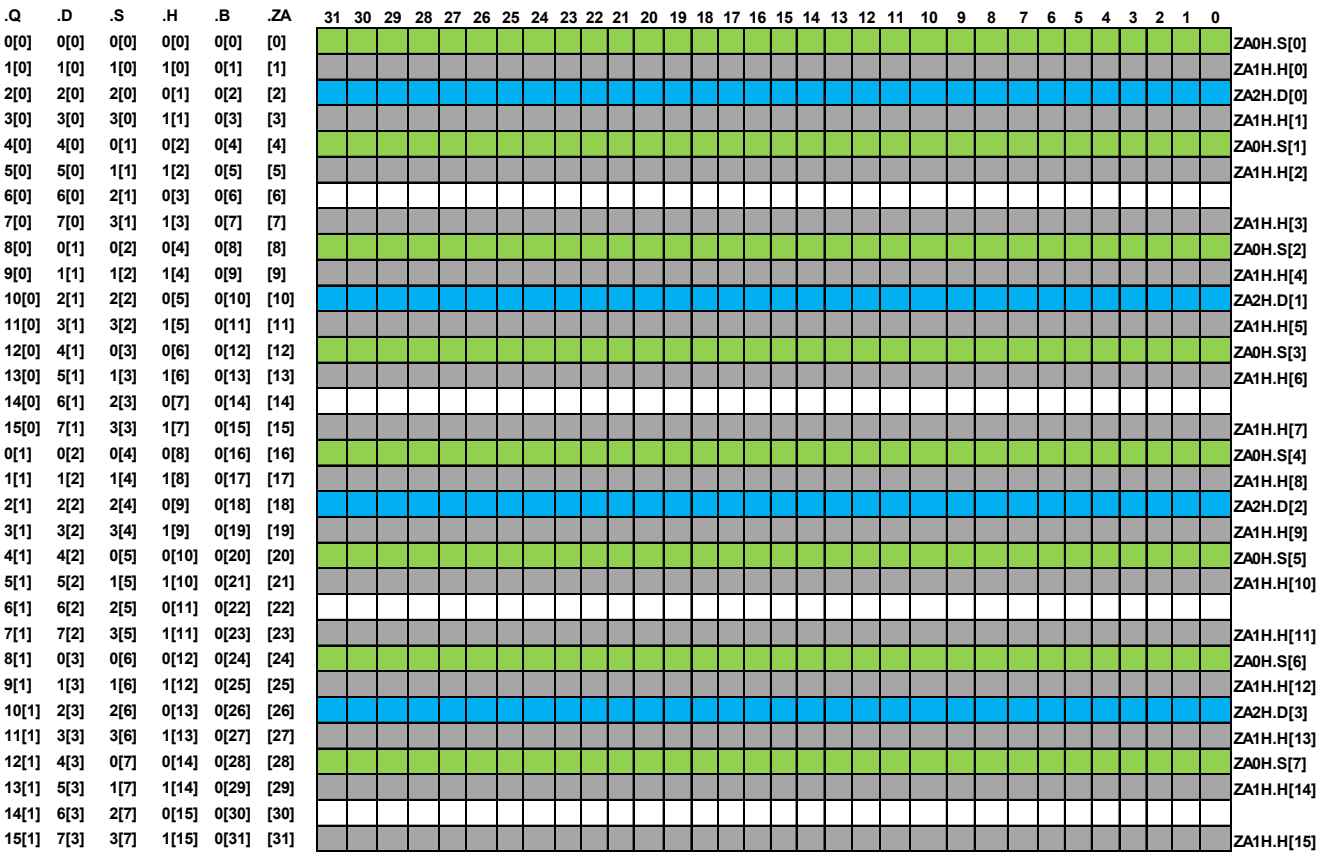

Figure B1-14 Using various non-overlapping ZA array vectors within tiles of different element sizes

IWDMCK It is possible to access overlapping *ZA* array vectors within tiles of differing element sizes. For example, tiles ZA0.H, ZA2.S, and ZA6.D have common *ZA* array vectors.

B1.4.12 SME2 Multi-vector operands

RKLRKJ Multi-vector operands allow certain SME2 instructions to access as source and destination operands:

- A group of two or four SVE *Z* vector registers.
- A group of two or four *ZA* tile slices.
- A group of two, four, eight, or sixteen *ZA* array vectors.

B1.4.12.1 Z multi-vector operands

DPSTFY A multi-vector operand consisting of two or four SVE *Z* vector registers is called *Z* multi-vector operand.

RVCXBQ A *Z* multi-vector operand can occupy:

- Consecutively numbered *Z* registers.
- *Z* registers with strided numbering.

DNYNRZ A *Z* multi-vector operand occupying two consecutively numbered *Z* vectors consists of Z*n+0* and Z*n+1*, where *n+x* modulo 32 is a register number in the range 0 to 31 inclusive.

DDZDBM A *Z* multi-vector operand occupying four consecutively numbered *Z* vectors consists of Z*n+0* to Z*n+3*, where *n+x* modulo 32 is a register number in the range 0 to 31 inclusive.

DVYKCM The preferred disassembly for a *Z* multi-vector operand of consecutively numbered *Z* vectors is a dash-separated register range, for example { Z0.S-Z1.S } or { Z30.B-Z1.B }. Toolchains must also support assembler source code which uses

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the alternative comma-separated list notation, for example { Z0.S, Z1.S } or { Z30.B, Z31.B, Z0.B, Z1.B}, and disassemblers may provide an option to select between the dash-separated range and comma-separated list notations.

D_{PCYZS} A *Z* multi-vector operand occupying two *Z* vectors with strided register numbering consists of a first register in the range Z0-Z7 or Z16-Z23, followed by a second register with a number that is 8 higher than the first.

D_{RZTTV} A *Z* multi-vector operand occupying four *Z* vectors with strided register numbering consists of a first register in the range Z0-Z3 or Z16-Z19, followed by three registers with a number that is each 4 higher than the last.

D_{DMTSL} The preferred disassembly for a *Z* multi-vector operand of *Z* vectors with strided register numbering is a comma-separated register list, for example { Z0.D, Z8.D } or { Z0.H, Z4.H, Z8.H, Z12.H }.

B1.4.12.2 ZA multi-slice operands

D_{JMCTK} A multi-vector operand consisting of two or four *ZA* tile slices is called *ZA* multi-slice operand.

R_{SCHNH} A *ZA* multi-slice operand can occupy:

- Consecutively numbered horizontal *ZA* tile slices.
- Consecutively numbered vertical *ZA* tile slices.

D_{VXMGC} In a *ZA* multi-slice operand that consists of:

- Two *ZA* tile slices, the lowest numbered slice is a multiple of 2.
- Four *ZA* tile slices, the lowest numbered slice is a multiple of 4.

D_{JFDSB} In instructions operating on *ZA* multi-slice operands, the lowest-numbered slice is selected by rounding down the sum of a 32-bit general-purpose register (slice index register W*s*) and an offset (slice index offset *offs*) to the nearest lower multiple of the number of the slices in the ZA multi-slice operand, modulo the number of slices in the named tile.

R_{XMMKZ} Instructions operating on the following *ZA* multi-slice operands are treated as UNDEFINED:

- The four-slice operand in a 64-bit element tile when SVL is 128 bits.
- The two-slice operand in a 128-bit element tile when SVL is 128 bits.
- The four-slice operand in a 128-bit element tile when SVL is 128 bits or 256 bits.

D_{GJTMX} The preferred disassembly for a *ZA* multi-slice operand is as follows:

- ZA*t*H.*T*[W*s*, *offs1*:*offs2*], for horizontal *ZA* two-slice operands, where *offs2* = *offs1* + 1.
- ZA*t*H.*T*[W*s*, *offs1*:*offs4*], for horizontal *ZA* four-slice operands, where *offs4* = *offs1* + 3.
- ZA*t*V.*T*[W*s*, *offs1*:*offs2*], for vertical *ZA* two-slice operands, where *offs2* = *offs1* + 1.
- ZA*t*V.*T*[W*s*, *offs1*:*offs4*], for vertical *ZA* four-slice operands, where *offs4* = *offs1* + 3.

B1.4.12.3 ZA multi-vector operands

D_{RGXBK} A multi-vector operand consisting of two, four, eight, or sixteen *ZA* array vectors is called *ZA* multi-vector operand.

D_{TGDRF} One *ZA* array vector is called a *ZA* single-vector group.

D_{FCYGL} Two consecutively-numbered vectors in the *ZA* array are called a *ZA* double-vector group.

D_{GCTYB} Four consecutively-numbered vectors in the *ZA* array are called a *ZA* quad-vector group.

I_{PMQRQ} The *ZA* multi-vector operand consists of one, two, or four vector groups, where a vector group is one of the following:

- *ZA* single-vector group.
- *ZA* double-vector group.
- *ZA* quad-vector group.

I_{KLBYZ} The SME2 architecture includes multi-vector instructions that access a *ZA* multi-vector operand consisting of the same number of vector groups as there are vectors in each *Z* multi-vector operand.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{HPKZM} The preferred disassembly for a *ZA* multi-vector operand consisting of two or four vector groups, defined in declarations D_{KQZYZ}, D_{JWRSN}, and D_{TTNGH}, includes the symbol `VGx2` or `VGx4`, respectively. The symbol `VGx2` or `VGx4` can optionally be omitted in assembler source code if it can be inferred from the other operands.

D_{CLJBX} In instructions that access a *ZA* multi-vector operand, the lowest-numbered vector is selected by the sum of a 32-bit general-purpose register (vector select register W*v*) and an offset (vector select offset *offs*), modulo one of the following values:

- SVL_B when the operand consists of one *ZA* vector group.
- $SVL_B/2$ when the operand consists of two *ZA* vector groups.
- $SVL_B/4$ when the operand consists of four *ZA* vector groups.

B1.4.12.3.1 ZA multi-vector operands of single-vector groups

D_{QFPHH} In instructions where the *ZA* multi-vector operand consists of two single-vector groups, each vector group is held in a separate half of the *ZA* array: ZA[*n+0*] and ZA[$SVL_B/2$ + *n+0*], where *n* is in the range 0 to ($SVL_B/2$ - 1) inclusive.

D_{TTHGQ} In instructions where the *ZA* multi-vector operand consists of four single-vector groups, each vector group is held in a separate quarter of the *ZA* array: ZA[*n+0*], ZA[$SVL_B/4$ + *n+0*], ZA[$SVL_B/2$ + *n+0*], and ZA[SVL_B*3/4 + *n+0*], where *n* is in the range 0 to ($SVL_B/4$ - 1) inclusive.

D_{KQZYZ} The preferred disassembly for a *ZA* multi-vector operand of single-vector groups is:

- ZA.*T*[W*v*, *offs*, VGx2], when the operand consists of two single-vector groups.
- ZA.*T*[W*v*, *offs*, VGx4], when the operand consists of four single-vector groups.

Where *offs* is in the range 0 to 7 inclusive, and *T* is one of B, H, S, or D.

I_{BYBQL} The mapping between *ZA* multi-vector operands of single-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

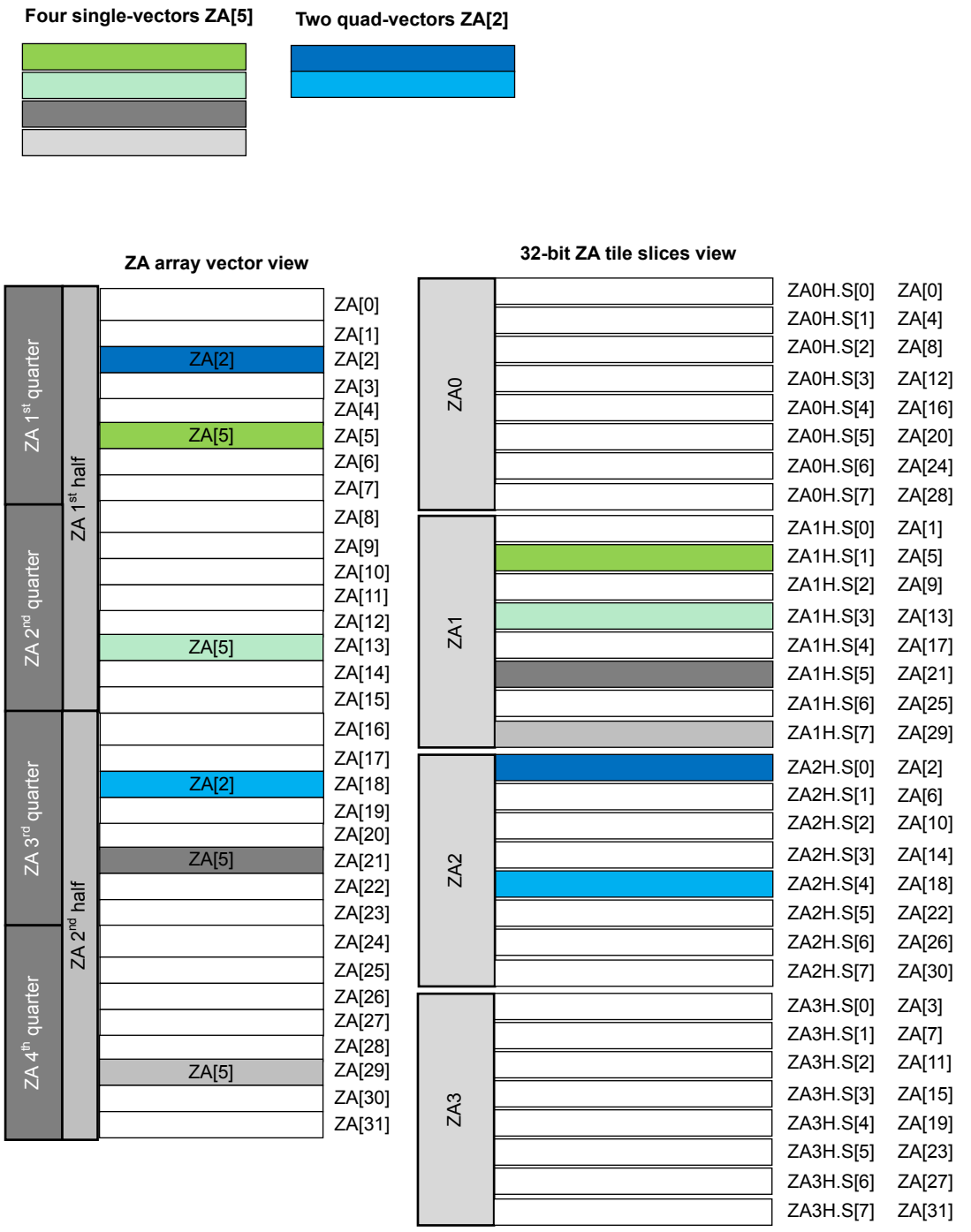

Figure B1-15 ZA array vector view and 32-bit element ZA tile slice view for single-vector groups

IMLNNG The mapping between *ZA* multi-vector operands of single-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

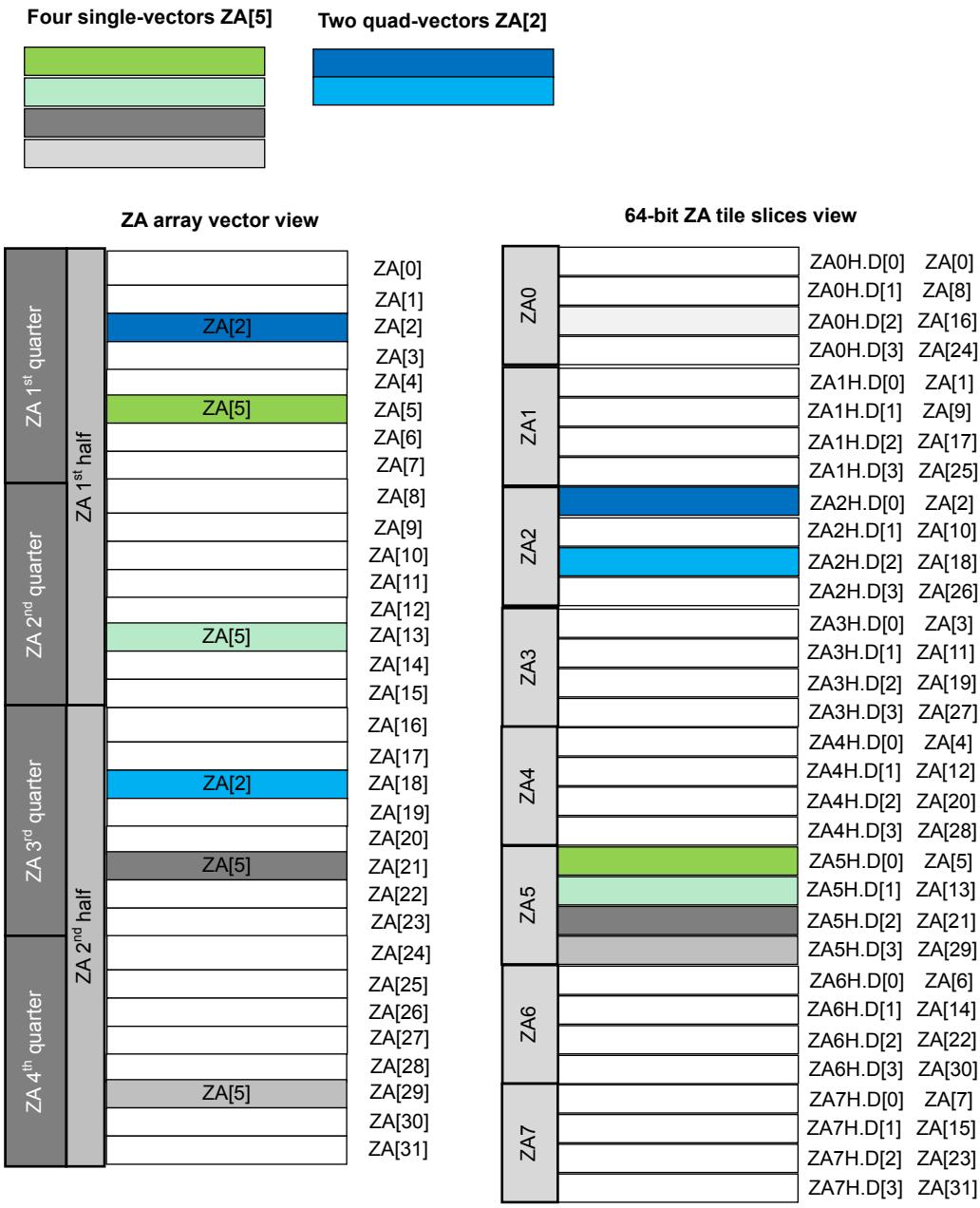

Figure B1-16 ZA array vector view and 64-bit element ZA tile slice view for single-vector groups

B1.4.12.3.2 ZA multi-vector operands of double-vector groups

D_{RBSQJ} In instructions where the *ZA* multi-vector operand consists of one double-vector group, the vector group is held in *ZA* array vectors: ZA[*n+0*] to ZA[*n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B - 2) inclusive.

D_{KKVVG} In instructions where the *ZA* multi-vector operand consists of two double-vector groups, each vector group is held in a separate half of the *ZA* array: ZA[*n+0*] to ZA[*n+1*], and ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B/2 - 2) inclusive.

D_{VMYGN} In instructions where the *ZA* multi-vector operand consists of four double-vector groups, each vector group is held in a separate quarter of the *ZA* array: ZA[*n+0*] to ZA[*n+1*], ZA[SVL_B/4 + *n+0*] to ZA[SVL_B/4 + *n+1*], ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+1*], and ZA[SVL_B*3/4 + *n+0*] to ZA[SVL_B*3/4 + *n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B/4 - 2) inclusive.

D_{JWRSN} The preferred disassembly for a *ZA* multi-vector operand of double-vector groups is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ZA.*T*[W*v*, *offs1*:*offs2*], where: *offs1* is a multiple of 2 in the range 0 to 14 inclusive, when the operand consists of one double-vector group.
- ZA.*T*[W*v*, *offs1*:*offs2*, VGx2], where *offs1* is a multiple of 2 in the range 0 to 6 inclusive, when the operand consists of two double-vector groups.
- ZA.*T*[W*v*, *offs1*:*offs2*, VGx4], where *offs1* is a multiple of 2 in the range 0 to 6 inclusive, when the operand consists of four double-vector groups.

Where *offs2* = *offs1* + 1, and *T* is one of B, H, S, or D.

ILZRTK The mapping between *ZA* multi-vector operands of double-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

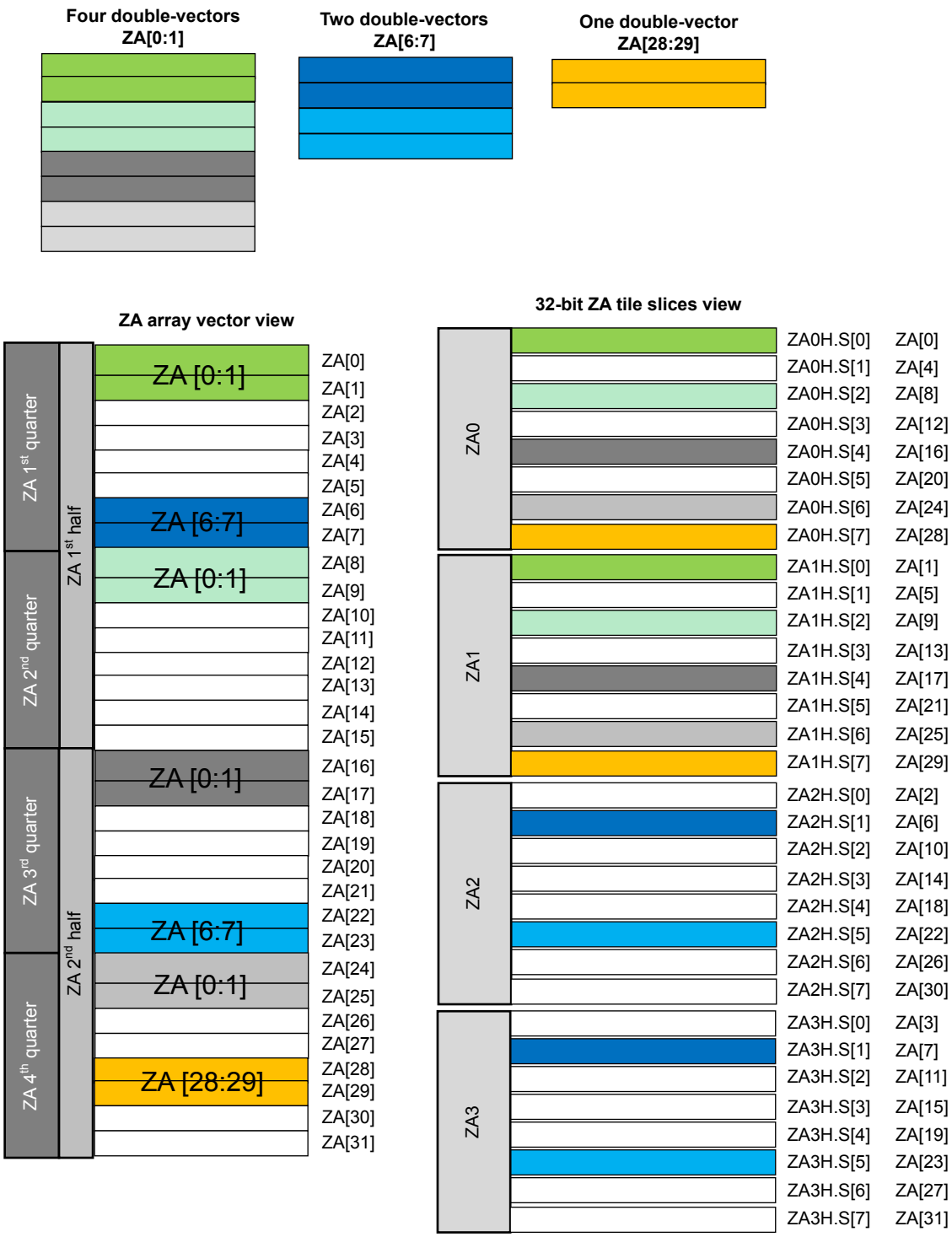

Figure B1-17 ZA array vector view and 32-bit element ZA tile slice view for double-vector groups

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IJYQTB The mapping between *ZA* multi-vector operands of double-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

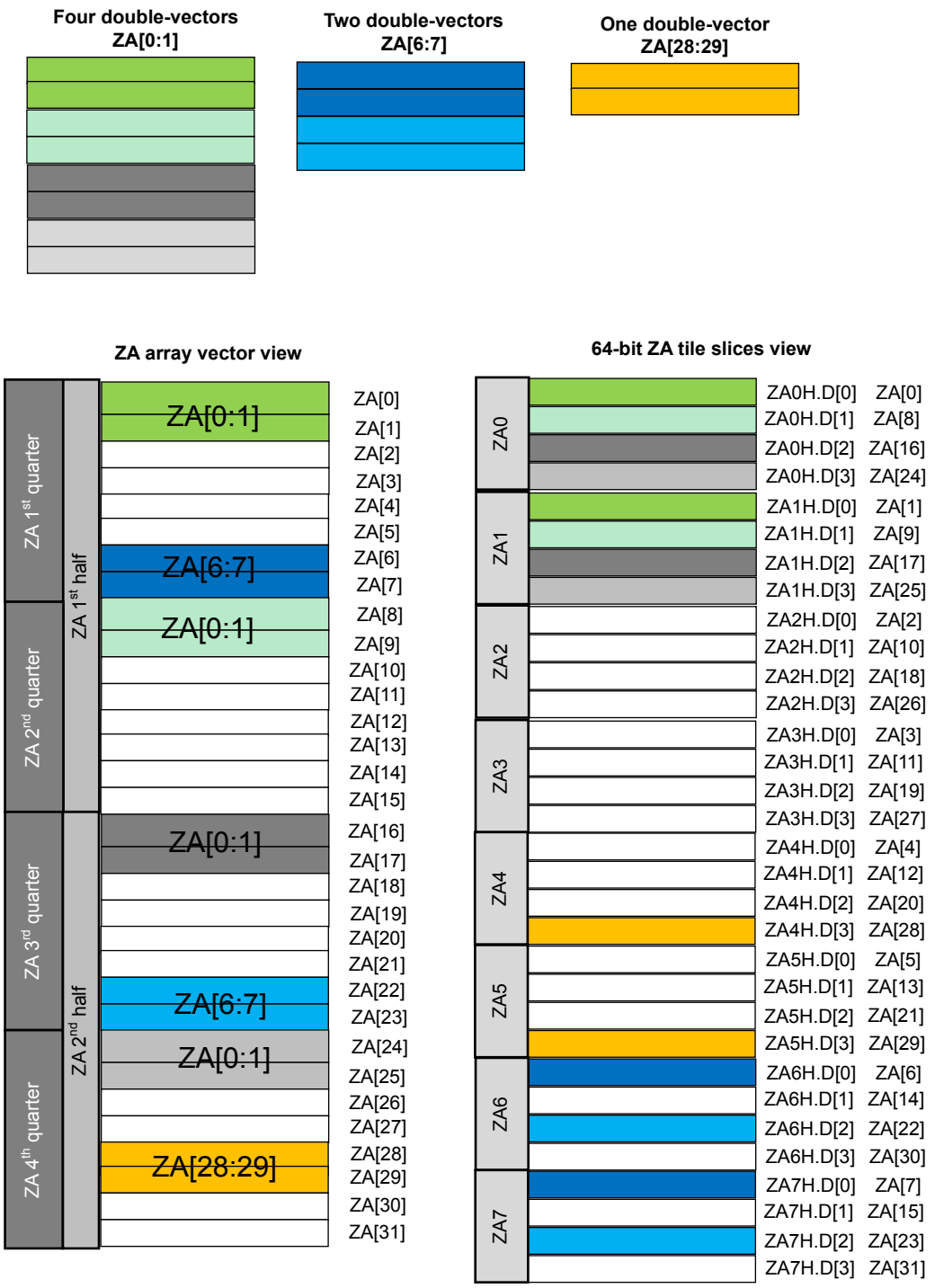

Figure B1-18 ZA array vector view and 64-bit element ZA tile slice view for double-vector groups

B1.4.12.3.3 ZA multi-vector operands of quad-vector groups

DWSTWB In instructions where the *ZA* multi-vector operand consists of one quad-vector group, the vector group is held in *ZA* array vectors: ZA[*n*+*0*] to ZA[*n*+*3*], where *n* is a multiple of 4 in the range 0 to (SVL_B - 4) inclusive.

DQJXHS In instructions where the *ZA* multi-vector operand consists of two quad-vector groups, each vector group is held in a separate half of the *ZA* array: ZA[*n*+*0*] to ZA[*n*+*3*], and ZA[SVL_B/2 + *n*+*0*] to ZA[SVL_B/2 + *n*+*3*], where *n* is a

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

multiple of 4 in the range 0 to (SVL_B/2 - 4) inclusive.

D_{BQWJD} In instructions where the *ZA* multi-vector operand consists of four quad-vector groups, each vector group is held in a separate quarter of the *ZA* array: ZA[*n+0*] to ZA[*n+3*], ZA[SVL_B/4 + *n+0*] to ZA[SVL_B/4 + *n+3*], ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+3*], and ZA[SVL_B*3/4 + *n+0*] to ZA[SVL_B*3/4 + *n+3*], where *n* is a multiple of 4 in the range 0 to (SVL_B/4 - 4) inclusive.

D_{TTNGH} The preferred disassembly for a *ZA* multi-vector operand of quad-vector groups is:

- ZA.*T*[W*v*, *offs1*:*offs4*], where: *offs1* is a multiple of 4 in the range 0 to 12 inclusive, when the operand consists of one quad-vector group.
- ZA.*T*[W*v*, *offs1*:*offs4*, VGx2], where *offs1* is 0 or 4, when the operand consists of two quad-vector groups.
- ZA.*T*[W*v*, *offs1*:*offs4*, VGx4], where *offs1* is 0 or 4, when the operand consists of four quad-vector groups.

Where *offs4* = *offs1* + 3, and *T* is one of B, H, S, or D.

I_{ZNSGW} The mapping between *ZA* multi-vector operands of quad-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

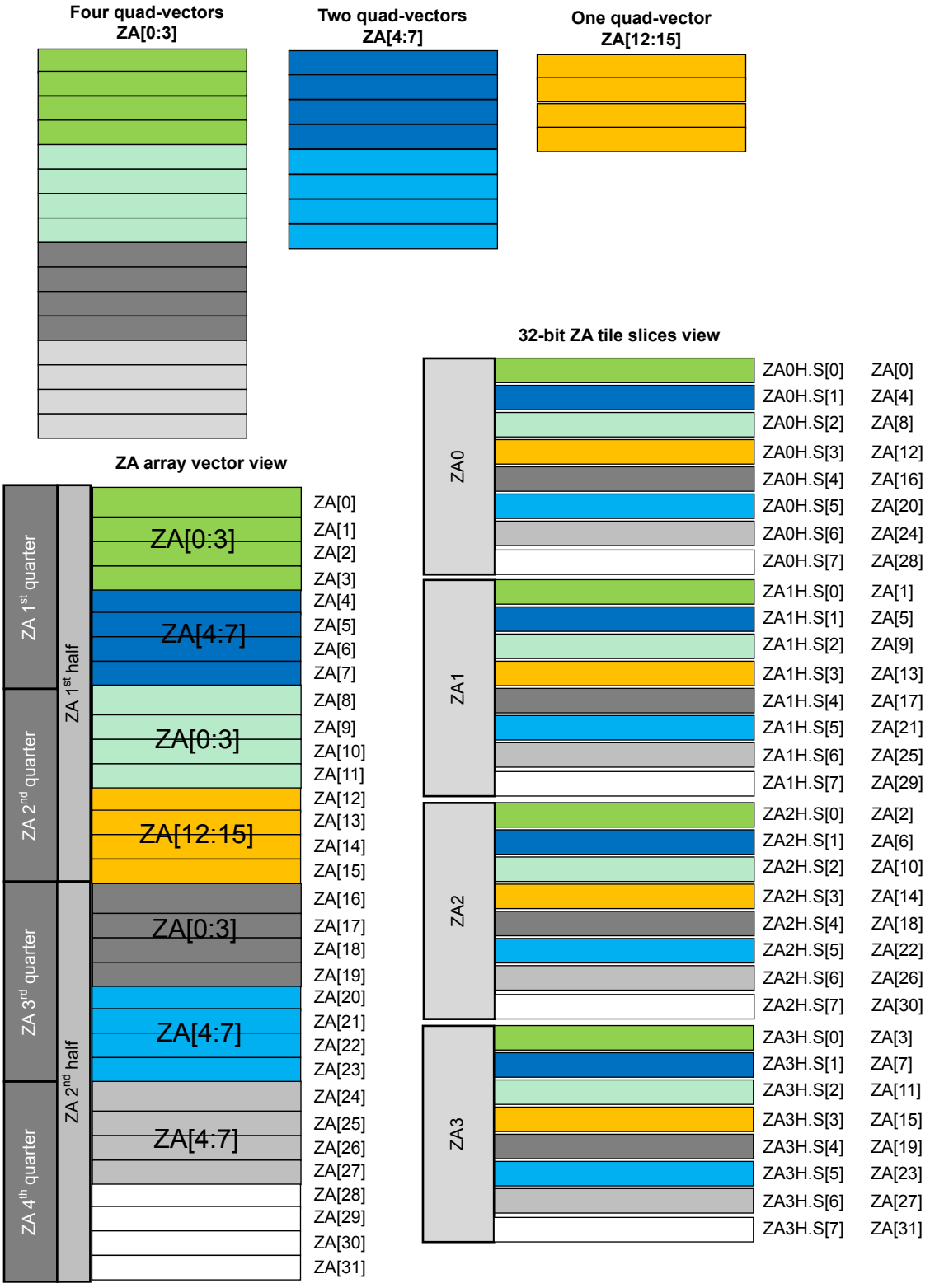

Figure B1-19 ZA array vector view and 32-bit element ZA tile slice view for quad-vector groups

IKBMLX The mapping between *ZA* multi-vector operands of quad-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

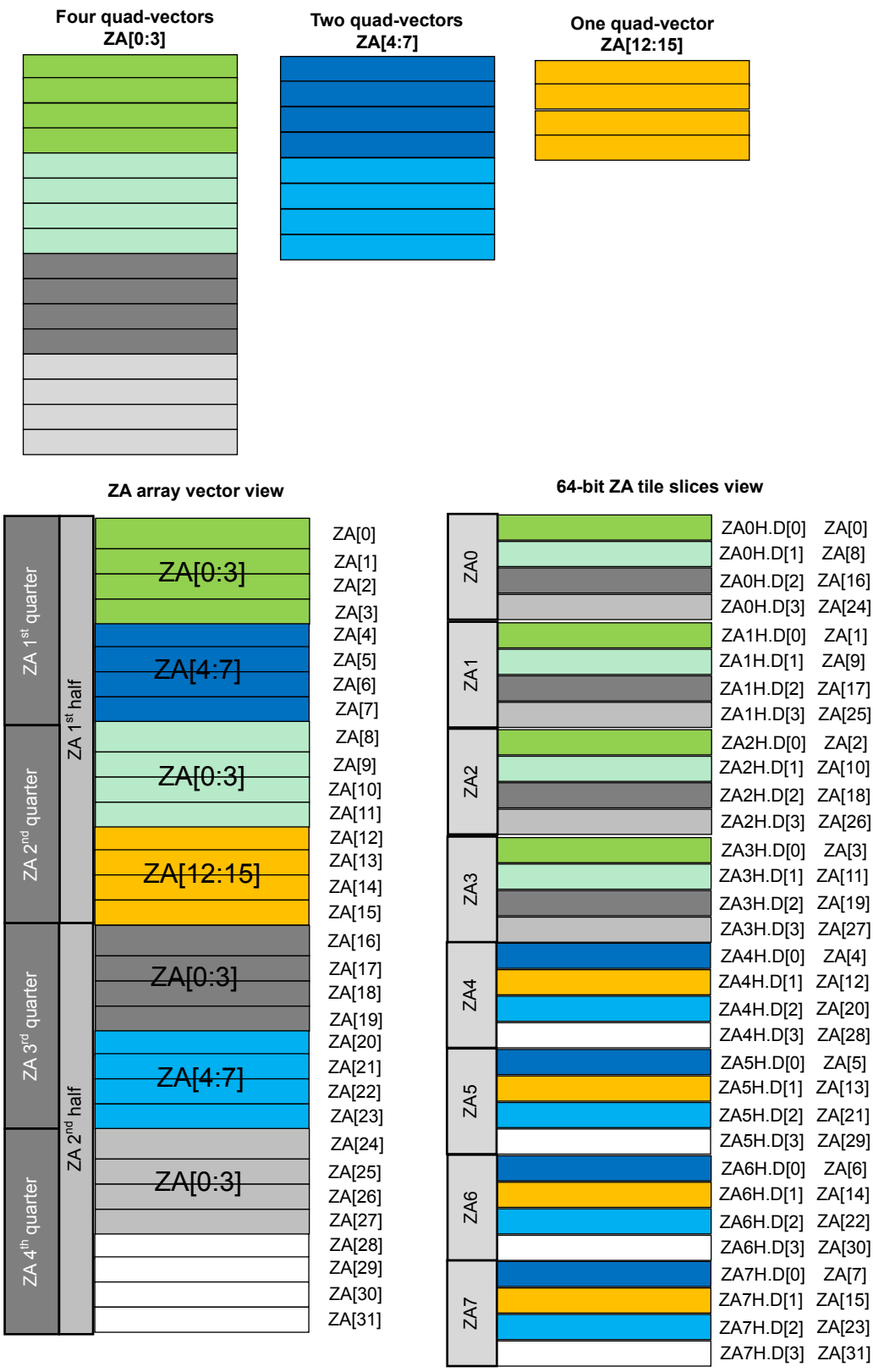

Figure B1-20 ZA array vector view and 64-bit element ZA tile slice view for quad-vector groups

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4.13 SME2 ZT0 register

R_{KHZMV} A PE with FEAT_SME2 has a 512-bit register, *ZT0*, to store a lookup table.

R_{WDJBC} The ZT0 register holds 8-bit, 16-bit, or 32-bit lookup table elements that are stored in the least significant bits of 32-bit table entries. The lowest numbered 32 bits in the register hold table entry 0.

R_{JQXLS} The ZT0 register lookup table can be accessed using fully packed 2-bit or 4-bit indices from a numbered portion of one source Z*n* SVE scalable vector register.

I_{BRRGG} When the ZT0 register lookup table is addressed by 2-bit indices, four different table elements (0-3) of a given element size can be accessed. When the lookup table is addressed by 4-bit indices, 16 different table elements (0-15) of a given element size can be accessed.

R_{JKYRB} The indexed 8-bit, 16-bit, or 32-bit table elements are read from the ZT0 register and packed into consecutive or strided elements of an SVE Z vector or Z multi-vector operand.

See also:

- About PSTATE.ZA.
- Streaming SVE mode.
- Accessing PSTATE fields at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.5 Software control features and EL0

The following sections describe the EL0 view of the software control features:

- Exception handling.
- Wait for Interrupt and Wait for Event.
- The YIELD instruction.
- Application level cache management.
- Instructions relating to Debug.
- About PSTATE.DIT.

B1.5.1 Exception handling

In the Arm architecture, an *exception* causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:

- Interrupts.
- Memory system aborts.
- Exceptions generated by attempting to execute an instruction that is UNDEFINED.
- System calls.
- Secure monitor or Hypervisor traps.
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in The AArch64 System Level Programmers' Model.

The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

The `BRK` instruction generates a Breakpoint Instruction exception. This provides a mechanism for debugging software using debugger executing on the same PE, see Breakpoint Instruction exceptions.

Note

The `BRK` instruction is supported only in the A64 instruction set. The equivalent instruction in the T32 and A32 instruction sets is `BKPT`.

B1.5.2 Wait for Interrupt and Wait for Event

Issuing a `WFI` instruction indicates that no further execution is required until a WFI wakeup event occurs, see Wait for Interrupt mechanism. This permits entry to a low-power state.

Issuing a `WFE` instruction indicates that no further execution is required until a WFE wakeup event occurs, see Wait for Event. This permits entry to a low-power state.

B1.5.3 The YIELD instruction

The `YIELD` instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see YIELD. This mechanism can be used to improve overall performance in a *Symmetric Multithreading* (SMT) or *Symmetric Multiprocessing* (SMP) system.

Examples of when the `YIELD` instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The `YIELD` instruction permits binary compatibility between SMT and SMP systems.

The `YIELD` instruction is a `NOP` hint instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The YIELD instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use YIELD in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

B1.5.4 Application level cache management

A small number of cache maintenance instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation without access permission causes the instruction to be trapped.

About the available operations, see Application level access to functionality related to caches.

B1.5.5 Instructions relating to Debug

Exception handling refers to the BRK instruction, which generates a Breakpoint Instruction exception. In addition, in AArch64 state and AArch32 state, the HLT instruction causes the PE to halt execution and enter Debug state. This provides a mechanism for debugging software using a debugger that is external to the PE, see About External Debug.

Note

In AArch32 state, previous versions of the architecture defined the DBG instruction, which could provide a hint to the debug system. This instruction executes as a NOP. Arm deprecates the use of the DBG instruction.

B1.5.6 About PSTATE.DIT

A *data-independent-time sequence of code* is the sequence of instructions executed from the first instruction executed after a change of PSTATE.DIT from 0 to 1 through the last instruction executed before a change of PSTATE.DIT from 1 to 0, inclusive. Instructions speculatively executed between the change of PSTATE.DIT from 0 to 1 and the change of PSTATE.DIT from 1 to 0 are considered part of the data-independent-time sequence of code.

A *data-independent-time resource* is any of the following:

- A general-purpose register other than the zero register (ZR).
- A Stack Pointer register.
- A SIMD&FP register.
- If FEAT_SVE or FEAT_SME is implemented, SVE scalable vector register.
- If FEAT_SVE or FEAT_SME is implemented, SVE scalable predicate register.
- If FEAT_SME is implemented, ZA register.
- If FEAT_SME2 is implemented, ZT0 register.
- SPSR_ELx.{N, Z, C, V}.
- PSTATE.{N, Z, C, V}.

Note

The collections of PSTATE.{N, Z, C, V} and SPSR_ELx.{N, Z, C, V} are each considered a single data-independent-time resource.

A *data-independent-time instruction* is an instruction that is required to honor the data-independent timing behaviors described in this section.

A *data-independent-time* value is any of:

- Any given value (V) read from memory by the Explicit Memory Read Effect of a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C), where V is not consumed by any other instruction (I2) for which any of the following apply:
  - I2 is not a data-independent-time instruction.
  - I2 is not executed in C.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
  - If I2 is an SVE instruction (regardless of whether streaming mode is enabled), V is used as the Governing predicate operand for I2.
  - If I2 is an SME instruction, V is used as the Governing predicate operand, slice index register, or vector select register for I2.

- Any given value (V) written to a data-independent-time resource by a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C), where V is not consumed by any other instruction (I2) for which any of the following apply:
  - I2 is not a data-independent-time instruction.
  - I2 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
  - If I2 is an SVE instruction (regardless of whether streaming mode is enabled), V is used as the Governing predicate operand for I2.
  - If I2 is an SME instruction, V is used as the Governing predicate operand, slice index register, or vector select register for I2.

- Any given value (V) read from a data-independent-time resource by a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C) and none of the following apply:
  - V was produced by another instruction (I0) and any of the following apply:
    - I0 is not a data-independent-time instruction.
    - I0 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect produced by I1.
  - V is not consumed by any other instruction (I2) for which any of the following apply:
    - I2 is not a data-independent-time instruction.
    - I2 is not executed in C.
    - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
    - If I2 is an SVE instruction (regardless of whether streaming mode is enabled), V is used as the Governing predicate operand for I2.
    - If I2 is an SME instruction, V is used as the Governing predicate operand, slice index register, or vector select register for I2.

- Any given value (V) written to memory by the Explicit Memory Write Effect of a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C) and none of the following apply:
  - V was produced by another instruction (I0) and any of the following apply:
    - I0 is not a data-independent-time instruction.
    - I0 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect produced by I1.
  - V was sourced from the zero register.
  - V is not consumed by any other instruction (I2) for which any of the following apply:
    - I2 is not a data-independent-time instruction.
    - I2 is not executed in C.
    - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
    - If I2 is an SVE instruction (regardless of whether streaming mode is enabled), V is used as the Governing predicate operand for I2.
    - If I2 is an SME instruction, V is used as the Governing predicate operand, slice index register, or vector select register for I2.

- Any given value (V) speculatively read from memory at a Location (L) as the result of executing a data-independent-time sequence of code, where L is not explicitly read by the data-independent-time sequence of code. Examples of this include:
  - Reading additional bytes from memory while reading the bytes required by an Explicit Memory Read Effect.
  - Hardware-generated prefetches.
  - Prefetch memory instructions.

The execution time of a given data-independent-time sequence of code must be independent of all data-independent-time values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The execution time of a data-independent-time instruction (I1) executed within a data-independent-time sequence of code must be independent of all data-independent-time values consumed or produced by I1.

The time to respond to an asynchronous exception taken from within a data-independent-time sequence of code must be independent of all data-independent-time values.

The execution of a data-independent-time sequence of code (C1) must not cause an irreversible change in microarchitectural state of the PE that would permit a sequence of code (C2) that is executed after C1 to determine any data-independent-time value accessed during the execution of C1. For the purposes of this requirement, a data-independent-time value written by an Explicit Memory Write Effect that is held in caches is not considered an irreversible change in microarchitectural state.

The execution of a data-independent-time sequence of code (C1) must not alter the state of PMU counters in a way that allows a sequence of code (C2) that is executed after C1 to determine any data-independent-time value accessed during C1.

Note

In cases where an instruction is architecturally required to only read or write a subset of a data-independent-time resource, the entire data-independent-time resource is considered to have been read or written for the sake of these requirements.

Note

The architecture does not guarantee that the execution time of a data-independent-time sequence of code is independent of data-independent-time values if those values are read from or written to a peripheral.

Note

The classification of a value being a data-independent-time value due to having been read from a memory location (L) by an Explicit Memory Read Effect or written to L by an Explicit Memory Write Effect is based on the particular Effect generated by a data-independent-time instruction executed from within a data-independent-time sequence of code. The value held in L is not itself considered a data-independent-time value. As a result, it is possible for the execution time of a data-independent-time sequence of code to vary based on the value held in L if L is accessed via any of the following:

- Outside the data-independent-time sequence of code, either architecturally or speculatively.
- Within a data-independent-time sequence of code but not as a data-independent-time value.

Note

- The *Operational information* section of an instruction description indicates whether or not that instruction is a data-independent-time instruction. If the *Operational information* section of an instruction description does not mention data-independent-time or if the section does not exist, then the instruction is not a data-independent-time instruction.
- The use of value prediction for data-independent-time values, is not compatible with the requirement that the timing is insensitive to the data value being loaded.
- Arm recommends that the FEAT_PAuth instructions do not have their timing dependent on the key value used in the pointer authentication, regardless of whether the instruction is executed in a data-independent-time sequence of code.
- The architecture makes no statement about the timing properties of any instructions executed outside of a data-independent-time sequence of code. However, it is likely that these instructions have timing that is invariant of the data in many situations.
- For SVE and SME predicated instructions, it is the programmer's responsibility to use a *Governing predicate* that does not reflect the values of the data being operated on.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

On an exception that is taken from AArch64 state to AArch64 state, PSTATE.DIT is copied to SPSR_ELx.DIT.

On an exception that is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

On an exception return from AArch64 state:

- SPSR_ELx.DIT is copied to PSTATE.DIT, when the target Exception level is in AArch64 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SPSR_ELx.DIT is copied to CPSR.DIT, when the target Exception level is in AArch32 state.

PSTATE.DIT can be written and read at all Exception levels.

Note

- PSTATE.DIT is unchanged on entry into Debug state.
- PSTATE.DIT is not guaranteed to have any effect in Debug state.
- Software is advised to ensure that no indirect branch is executed while secret data is held in registers.
- Software is advised to ensure that no direct branches target non-constant-time code while secret data is held in registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B2
The AArch64 Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:

- About the Arm memory model.
- Atomicity in the Arm architecture.
- Ordering requirements defined by the formal concurrency model.
- Additional ordering requirements outside of the scope of the formal concurrency model
- Restrictions on the effects of speculation.
- Memory barriers.
- Caches and memory hierarchy.
- Alignment support.
- Endian support.
- Memory types and attributes.
- Mismatched memory attributes.
- Synchronization and semaphores.

Note

In this chapter, System register names usually link to the description of the register in AArch64 System Register Descriptions, for example. SCTLR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model.

This section contains:

- Address space.
- Memory type overview.
- SVE memory model.
- SME memory model.

B2.1.1 Address space

Address calculations are performed using 64-bit registers.

However, supervisory software can configure the top eight address bits for use as a tag, as described in Address tagging. If this is done, address bits[63:56]:

- Are not considered when determining whether the address is valid.
- Are never propagated to the Program Counter.

Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

Simple sequential execution of instructions might overflow the valid address range. For more information, see Virtual address space overflow.

Memory accesses use the `Mem()` function. This function makes an access of the required type. If supervisory software configures the top eight address bits for use as a tag, the top eight address bits are ignored.

The `AccessType` defines the different access types and attributes.

Note

- The AArch64 System Level Memory Model and The AArch64 Virtual Memory System Architecture include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also include pseudocode descriptions of these operations.
- For information on the pseudocode that relates to memory accesses, see Basic memory access, Unaligned memory access, and Aligned memory access.

B2.1.2 Memory type overview

The Arm architecture provides the following mutually-exclusive memory types:

Normal This is generally used for the majority of memory operations, both read/write and read-only operations.

Device The Arm architecture forbids Speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive Locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See Gathering.
- They preserve the access order and synchronization requirements for accesses to a single peripheral. See Reordering.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- They indicate whether a write can be acknowledged other than at the end point. See Early Write Acknowledgement.

For more information on Normal memory and Device memory, see Memory types and attributes.

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A *Note* in Device memory describes how these memory types map onto the Armv8 memory types.

B2.1.3 SVE memory model

I_{CZFSY} SVE predicated memory operations have a vector element size and a memory element access size. The vector element size specifies the data that is read from and written to the vector. The memory element access size specifies the amount of data that is read from and written to the memory.

I_{TJQJF} The vector element size and the memory element access size do not need to have the same value.

I_{LGGHH} For each memory element, there is an associated element address.

SVE also affects behavior in the following areas:

- Requirements for single-copy atomicity.
- SVE memory ordering relaxations.
- Load or Store of Single or Multiple registers.
- Endianness in SVE operations.
- SVE loads and stores that access Device memory.

B2.1.4 SME memory model

R_{BQSCG} SME load/store memory accesses are subject to the same rules that govern SVE load/store memory accesses.

I_{WWVYJ} When the PE is in *Streaming SVE mode*, there are relaxations for Advanced SIMD&FP instructions in the following areas:

- Streaming SVE mode memory ordering relaxations.
- Streaming SVE mode loads and stores that access Device memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. In the Arm architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- Requirements for single-copy atomicity.
- Properties of single-copy atomic accesses.
- Multi-copy atomicity.
- Requirements for multi-copy atomicity.
- Concurrent modification and execution of instructions.
- Possible implementation restrictions on using atomic instructions.

For more information about the memory types, see Memory type overview.

B2.2.1 Requirements for single-copy atomicity

For explicit memory effects generated from an Exception level the following rules apply:

- A read that is generated by a load instruction that loads a single general-purpose register and is aligned to the size of the read in the instruction is single-copy atomic.
- A write that is generated by a store instruction that stores a single general-purpose register and is aligned to the size of the write in the instruction is single-copy atomic.
- Reads that are generated by a Load Pair instruction that loads two general-purpose registers and are aligned to the size of the load to each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes that are generated by a Store pair instruction that stores two general-purpose registers and are aligned to the size of the store of each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-Exclusive Pair instructions of two 32-bit quantities and Store-Exclusive Pair instructions of 32-bit quantities are single-copy atomic.
- When the Store-Exclusive of a Load-Exclusive/Store-Exclusive pair instruction using two 64-bit quantities succeeds, it causes a single-copy atomic update of the entire memory location being updated.

  Note

  To atomically load two 64-bit quantities, perform a Load-Exclusive pair/Store-Exclusive pair sequence of reading and writing the same value for which the Store-Exclusive pair succeeds, and use the read values from the Load-Exclusive pair.

- Where translation table walks generate a read of a translation table entry, this read is single-copy atomic.
- For the atomicity of instruction fetches, see Concurrent modification and execution of instructions.
- Reads to SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being stored, have each element treated as a single-copy atomic store.
- Reads to SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit reads.
- Writes from SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit writes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_LS64WB is implemented, accesses to two 128-bit SIMD and floating-point registers by an LDP (SIMD&FP) or STP (SIMD&FP) instruction are treated as one single-copy atomic 256-bit access when both the following are true:
  - The access is 32-byte aligned in memory.
  - The access is to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory.
- SVE predicated load and store instructions have the same single-copy atomicity guarantees as the SIMD load and store instructions for 64-bit or smaller elements.
- An SVE predicated load or store of a 128-bit element that is 64-bit aligned is treated as a pair of 64-bit single-copy atomic accesses.
- SVE unpredicated load and store instructions are performed as a sequence of byte accesses.
- SVE unpredicated load and store instructions do not guarantee that any access larger than a byte will be performed as a single-copy atomic access.
- When FEAT_LS64 is implemented, a single-copy atomic load of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte read from the target address.
- When FEAT_LS64 is implemented, a single-copy atomic store of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte write to the target address.
- For unaligned memory accesses, the single-copy atomicity is described in Alignment of data accesses.
- The read and write generated by an Atomic instruction is single-copy atomic at the overall size of the data access.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

Note

In AArch64 state, no memory accesses from a `DC ZVA` have single-copy atomicity of any quantity greater than individual bytes.

If, according to these rules, an instruction is executed as a sequence of accesses, exceptions, including interrupts, can be taken during that sequence, regardless of the memory type being accessed. If any of these exceptions are returned from using their preferred return address, the instruction that generated the sequence of accesses is re-executed, and so any access performed before the exception was taken is repeated. See also Taking an asynchronous exception during a multi-access load or store.

Note

The exception behavior for these multiple access instructions means that they are not suitable for use for writes to memory for the purpose of software synchronization.

B2.2.1.1 Changes to single-copy atomicity in Armv8.4

In addition to the single-copy atomicity requirements listed above:

Instructions that are introduced in FEAT_LRCPC are single-copy atomic when all of the following conditions are true:

- All bytes being accessed are within the same 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, all loads and stores are single-copy atomic when all of the following conditions are true:

- Accesses are unaligned to their data size but all bytes being accessed are within a 16-byte quantity that is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, `LDP`, `LDNP`, and `STP` instructions that load or store two 64-bit registers are single-copy atomic when all of the following conditions are true:

- The overall memory access is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_LSE2 is implemented, `LDP`, `LDNP`, and `STP` instructions that access fewer than 16 bytes are single-copy atomic when all of the following conditions are true:

- All bytes being accessed are within a 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

B2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2, if one of the overlapping reads generated by L_1 Reads-from one of the overlapping writes generated by S_2, then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1.

For more information, see Ordering requirements defined by the formal concurrency model.

B2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

B2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The Arm memory model is Other-multi-copy atomic. For more information, see External visibility requirement.

B2.2.5 Concurrent modification and execution of instructions (CMODX)

The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction behaving like any possible sequence of instructions executable at the same Exception level. However, the behavior is more restricted if both the instruction before and the instruction after modification are one of `B`, `B.cond`, `BL`, `BRK`, `CB<cc>`, `CBB<cc>`, `CBH<cc>`, `CBNZ`, `CBZ`, `HVC`, `ISB`, `NOP`, `SMC`, `SVC`, `TBNZ`, `TBZ`, `TRCIT`, or `UDF`.

The more restricted behavior is that the architecture guarantees that the behavior of the instruction that is being concurrently modified and executed is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

When the cases of the more restricted behavior do not apply, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. No PE must be executing an instruction when another PE is modifying that instruction.
2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn                 ; Clean data cache by VA to point of unification (PoU)
        DSB ISH                     ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn                 ; Invalidate instruction cache by VA to PoU
        DSB ISH
```

Note

- The `DC CVAU` operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- In a multiprocessor system, the DC CVAU and IC IVAU are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence.

3. When the modified instructions are observable, each PE that is executing the modified instructions must execute an `ISB` or perform a context synchronization event to ensure execution of the modified instructions:

```
        ISB                             ; Synchronize fetched instruction stream
```

Note

For the purposes of CMODX behavior, it is not considered a modification of an instruction at a Location L if a Memory Write Effect E_2 writes the same value to L that was written by the latest Memory Write Effect E_1 earlier in the coherence order for L than E_2.

For more information about the required synchronization operation, see Synchronization and coherency issues between data and instruction accesses.

For information about memory accesses caused by instruction fetches, see Ordering relations.

B2.2.6 Possible implementation restrictions on using atomic instructions

In some implementations, and for some memory types, the properties of atomicity can be met only by functionality outside the PE. Some system implementations might not support atomic instructions for all regions of the memory. In particular, this can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Device, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such implementations, it is defined by the system:

- Whether the atomic instructions are atomic in regard to other agents that access memory.
- If the atomic instructions are atomic in regard to other agents that access memory, which address ranges or memory types this applies to.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the atomic instructions will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

The architecture only requires that Conventional memory that is mapped in this way supports this functionality.

If the atomic instructions are not atomic in regard to other agents that access memory, then performing an atomic instruction to such a location can have one or more of the following effects:

- The instruction generates a synchronous External abort.
- The instruction generates a System Error interrupt.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  For the EL1&0 translation regime, if the atomic instruction is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.
- The instruction is treated as a `NOP`.
- The instructions are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a System Error interrupt.

A system implementation might support atomic floating-point memory operation instructions on different regions of Device and Non-cacheable memory from those supported by other atomic instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3 Ordering requirements defined by the formal concurrency model

The material presented in the current section corresponds to what is described in the formal concurrency model. The definitions of relations, including but not limited to Dependency-ordered-before, Barrier-ordered-before, or Atomic-ordered-before, are automatically transliterated from the formal concurrency model.

The Arm memory model introduces a number of relations which are as follows:

- *Intrinsic relations*; For example, Intrinsic Data Dependencies, Intrinsic Control Dependencies, Intrinsic Order Dependencies and Translation-intrinsically-before are hardware requirements which stem from the instruction semantics.
- *After relations*; For example, Coherence-after and TLBI-after are relations which happen to be oriented that way in a specific execution, but could be oriented the other way in a different execution.
- *Observation relations*; For example, Explicit-Observed-by and TLBI-Observed-by build on *After relations* to describe an execution.
- *Ordered relations*; For example, Ordered-before and TLBI-ordered-before are architectural requirements which must be respected by hardware in all executions.

This section describes observation and ordering in the Arm memory model. It contains the following subsections:

- Basic definitions.
- Intrinsic Dependency relations.
- Tag-check-intrinsically-before.
- Translation-intrinsically-before.
- Fetch-intrinsically-before.
- Dependency relations.
- Ordering relations.
- Observation relations.
- Definition of an Architecturally Allowed Execution.
- Alternative formulations of the External visibility requirement for Explicit Memory Effects.

For more information about endpoint ordering of memory accesses, see Reordering.

In the Arm memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers. See Memory types and attributes.

The Arm architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- The AArch64 System Level Memory Model.
- The AArch64 Virtual Memory System Architecture.

See also Mismatched memory attributes.

The formal memory model described in this section covers much, but not all, of the memory system functionality. In particular the following are in scope:

- Ordering requirements for instructions that explicitly access memory (for example: `LDR`, `STR`, `SWP`, etc.).
  - This includes the ordering requirements between any two Explicit Memory Effects where:
    - The Explicit Memory Effects are to the same Location either from the same or different PEs.
    - One or both of the Explicit Memory Effects are generated by an atomic instruction.
    - A barrier instruction is in program order between the Explicit Memory Effects.
    - There is a dependency between the Explicit Memory Effects.
  - And excludes cases where:
    - At least one of the Explicit Memory Effects is generated by SVE instructions.
    - At least one of the Explicit Memory Effects is generated by SME instructions.
    - At least one of the Explicit Memory Effects is generated by instructions added by FEAT_MOPS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Ordering requirements relating to the Virtual Memory System Architecture:
  - This includes:
    - Implicit TTD Memory Read Effects.
    - Implicit TTD Memory Write Effects when FEAT_HAF or FEAT_HAFDBS is implemented.
    - MMU Fault Effects that cause Data Abort exceptions.
  - This excludes:
    - Any Effects relating to address translation instructions.
    - Any Effects relating to address translation for instruction fetch.
    - The ordering requirements between Implicit TTD Memory Effects that belong to different levels or stages of translation.
    - The ordering requirements relating to FEAT_HDBSS.
    - The ordering requirements relating to FEAT_HACDBS.
- Ordering requirements relating to tag checks when FEAT_MTE2 is implemented. This includes:
  - Implicit Tag Memory Read Effects generated by tag checked instructions.
  - Explicit Tag Memory Effects generated by tag load, tag store, tag and data store, and bulk allocation tag access instructions.
  - TagCheck Fault Effects generated by tag checked instructions.
- Ordering requirements relating to instruction fetch. This includes:
  - Implicit Instruction Memory Read Effects generated as a result of instruction fetch.
  - DC CVAU Effects.
  - IC Effects.
  - Undefined Instruction Fault Effects.

Note

Arm is in the process of formalizing many features of the memory system architecture, but this work is not complete at this time. The following items are not in scope:

- Ordering requirements relating to the Event Register.
- Ordering requirements relating to the Exclusives monitor.
- Endpoint arrival of memory accesses.
- Interactions between memory accesses and peripherals.
- Ordering requirements relating to asynchronous exceptions and error exceptions.
- Ordering requirements relating to the generic interrupt controller.
- Ordering requirements for direct and indirect system register reads and writes.
- Ordering requirements relating to FEAT_GCS.
- Ordering requirements relating to FEAT_SPE.
- Ordering requirements relating to FEAT_TRBE.
- Ordering requirements relating to FEAT_RME.
- Ordering requirements relating to RAS PE architecture.
- Ordering requirements of DC Effects generated by `DC` instructions other than the `DC CVAU, Xt` instruction.

These will be addressed progressively in future versions of the Arm architecture, but until then, the required behaviors are described less formally in the prose and pseudocode of the Arm ARM.

For the purposes of the memory model, all Effects generated due to execution of an instruction are considered to be generated by the PE on which the instruction is executed. The list of all Effects of an instruction in the scope of the Arm Memory Model is further presented in Effects.

In this section Ordering requirements defined by the formal concurrency model, all of the following are assumed:

- One of the following applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - All PEs are part of the same Inner Shareable domain and all Memory Effects are to Inner Shareable or Outer Shareable memory as long as no TLB invalidations nor their corresponding `DSB` specify the Non-shareable domain.
  - All PEs are part of the same Outer Shareable domain and all Memory Effects are to Outer Shareable memory as long as no TLB invalidations nor their corresponding `DSB` specify the Non-shareable domain or the Inner Shareable domain.
  - All PEs are part of the Full system domain and all Memory Effects are to Normal, Inner Non-Cacheable, Outer Non-Cacheable memory or Device memory as long as there are no TLB invalidations.

- All Memory Effects are to Normal, Inner Write-Back, Outer Write-Back memory. Behaviors in Ordering requirements defined by the formal concurrency model also apply to Memory Effects to memory with attributes other than Normal, Inner Write-Back, Outer Write-Back (for example, Device or Inner Write-Through, Outer Write-Through) as long as the behavior of the instruction which generated the Memory Effect is guaranteed for the memory type. For example, atomic instructions are only guaranteed to be atomic for Normal, Inner Write-Back, Outer Write-Back, Shareable memory. Memory Effects to memory with attributes other than Normal, Inner Write-Back, Outer Write-Back may require additional ordering behaviors than specified in Ordering requirements defined by the formal concurrency model. Such additional behaviors are described in Memory types and attributes.
- All Memory Effects to a given Location are not using mismatched memory attributes as described in Mismatched memory attributes.

B2.3.1 Basic definitions

The Arm memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory.

Location

A *Location* is a Memory Location or a Tag Location. A *Memory Location* is a byte of data that is associated with an address in the physical address space. A *Tag Location* is a 4-bit MTE Allocation Tag that is associated with an address in the physical address space.

Effects

The *Effects* of an instruction can be:

- Register Effects.
- Memory Effects.
- Explicit Memory Effects.
- Implicit Translation Table Descriptor (TTD) Memory Effects.
- Hardware Update Effects.
- Tag Memory Effects.
- Implicit Instruction Memory Read Effects.
- Barrier Effects.
- Instruction Fetch Barrier Effects.
- Conditional Branching, and Intrinsic Branching Effects.
- Fault Effects, Exception Entry, and Exception Return Effects.
- TLBI Effects, Completed TLBI Effects and Invalidation Scopes.
- TLBUncacheable Effects.
- DC CVAU Effects.
- IC Effects, Completed IC Effects.
- Empty Effects.

Program order

An Effect E_1 of an instruction I_1 appears in program order before an Effect E_2 of an instruction I_2 if and only if I_1 occurs before I_2 in the order specified by the program. Each Effect generated by an instruction has a unique Effect identifier that characterizes it among the Effects generated by the same instruction.

Register Effects

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The *Register Effects* of an instruction are register reads or register writes of that instruction. Register Effects only pertain to:

- General purpose registers from R0 to R30, excluding the zero register.
- SVE registers.
- SIMD/FP registers.
- PSTATE.NZCV.
- Direct reads from System registers that are readable.
- Direct writes to System registers that are writable.
- Direct reads from Special-purpose registers that are readable.
- Direct writes to Special-purpose registers that are writable.

For an instruction that accesses registers, a *Register Read Effect* is generated for each register read by the instruction and a *Register Write Effect* is generated for each register written by the instruction. An instruction may generate both Register Read and Write Effects.

Memory Effects

The *Memory Effects* of an instruction are the Read or Write Effects from memory of that instruction. For an instruction that accesses memory, a *Memory Read Effect* is generated for each Location read from by the instruction and a *Memory Write Effect* is generated for each Location written to by the instruction. An instruction can generate both Memory Read and Write Effects.

Explicit Memory Effects

An *Explicit Memory Effect* is a Memory Read or Memory Write Effect that is generated by one of the following:

- A load instruction, store instruction, atomic instruction, Tag and Data store instruction, `DC ZVA`, `DC GZVA`, or `STZGM`; to access the Memory Location(s) addressed by any of the following from the instruction:
  - The `Xn|SP` operand, plus any applicable offset.
  - The `PC` plus the offset specified by the `<label>` argument.
- A Tag load instruction, Tag store instruction, Tag and Data store instruction, Bulk Allocation tag access instruction, `DC GZVA`, or `DC GVA`; to access the Tag Location(s) addressed by the `Xn|SP` operand, plus any applicable offset, of the instruction.
- A Memory Copy or Memory Set instruction; to access the Memory Location(s) addressed by the `Xs` or `Xd` operand of the instruction, including all of the bytes within the range specified by the `Xn` operand of the instruction.
- A Memory Set with tag setting instruction; to access the Tag Location(s) addressed by the `Xs` or `Xd` operand of the instruction, including all of the Allocation tags within the range specified by the `Xn` operand of the instruction.
- An `MRS` or `MSR` instruction that is transformed into a Memory Effect due to FEAT_NV2; to access the Memory Locations addressed by VNCR_EL2 plus the offset for the specified register.

Note

Prefetch memory instructions do not generate any Explicit Memory Effects.

Implicit Translation Table Descriptor Memory Effects

Implicit Translation Table Descriptor Memory Effects (Implicit TTD Memory Effects) are Memory Read or Write Effects to translation table descriptors caused by address translation, including the hardware updates for the Access Flag or the Dirty State of a TTD.

Hardware Update Effects

Hardware Update Effects are Implicit TTD Memory Write Effects that correspond to writes modifying the Access Flag or the Dirty State of a TTD.

For more information on Hardware Updates of the Access Flag or Dirty State, see Hardware management of the Access flag and Hardware management of the dirty state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A Hardware Update Effect for the Access Flag may occur even in the absence of corresponding architecturally executed instructions, in which case the Hardware Update Effect is considered autonomous. Autonomous Hardware Update Effects are not constrained by the ordering relations that apply to the Hardware Update Effects of architecturally executed instructions.

Tag Memory Effects

Tag Memory Effects are a subset of Memory Effects. A Tag Memory Effect refers to a Memory Read or Write Effect to a Tag Location. For example:

- `LDG` and `LDGM` instructions generate an Explicit Tag Memory Read Effect for their read from a Tag Location.
- `STG`, `STGM`, `STGP`, `DC GZVA`, and `DC GVA` instructions generate an Explicit Tag Memory Write Effect for their write to a Tag Location.
- A Tag Checked memory access generates an Implicit Tag Memory Read Effect for its read of a Tag Location, as well as an Explicit Data Memory Effect.

Implicit Instruction Memory Read Effects

An *Implicit Instruction Memory Read Effect* refers to a Memory Read Effect generated as a result of an instruction fetch.

Barrier Effects

Barrier instructions generate *Barrier Effects*. In this chapter, the resulting Effect is named after the instruction. Therefore, an ISB instruction generates an ISB Effect, a DMB instruction generates a DMB Effect, and a DSB instruction generates a DSB Effect.

The following conventions are used when referring to DMB Effects:

- A DMB FULL Effect is generated by a DMB instruction with neither a LD nor ST qualifier.
- A DMB LD Effect is generated by a DMB instruction with the LD qualifier.
- A DMB ST Effect is generated by a DMB instruction with the ST qualifier.

The following conventions are used when referring to DSB Effects:

- A DSB FULL Effect is generated by a DSB instruction with neither a LD nor ST qualifier.
- A DSB LD Effect is generated by a DSB instruction with the LD qualifier.
- A DSB ST Effect is generated by a DSB instruction with the ST qualifier.

Instruction Fetch Barrier Effects

An *Instruction Fetch Barrier Effect* (IFBE) is generated by any of the following:

- A Context Synchronization event.
- An exception entry to ELx, regardless of the value of SCTLR_ELx.EIS, due to an exception generated for any reason other than any of the following:
  - `SVC` instruction execution that is not trapped.
  - `HVC` instruction execution that is not disabled.
  - `SMC` instruction execution that is not trapped or disabled.
  - `BKPT` instruction execution.
  - `BRK` instruction execution.

Note

- If FEAT_ExS is not implemented, or if FEAT_ExS is implemented and the SCTLR_ELx.EOS field is 1, an Exception Return effect is an IFBE due to it being a Context Synchronization event.
- If FEAT_ExS is not implemented, or if FEAT_ExS is implemented and the SCTLR_ELx.EIS field is 1, the Exception entry effects due to the exceptions excluded above are IFBEs due to being Context Synchronization events.

Conditional Branching, and Intrinsic Branching Effects

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Branching Effects are generated when a choice is made between two possible paths in the execution flow. Branching Effects represent such decision points.

Conditional Branching Effects are generated by Conditional branch instructions, when as a consequence of evaluating a condition a choice potentially affecting the execution flow of other subsequent instructions is made.

Intrinsic Branching Effects due to a translation occur when, as a consequence of reading a TTD, a choice is made between raising a fault (for example, if the descriptor is invalid) or performing a physical access. *Intrinsic Branching Effects* represent such decision points.

Intrinsic Branching Effects due to tag checking occur when, as a consequence of reading a tag, a choice is made between raising a fault (that is, due to a failed tag check) or performing a physical access. *Intrinsic Branching Effects* represent such decision points.

Intrinsic Branching Effects due to instruction fetch occur when, as a consequence of reading and decoding an instruction, a choice is made between raising a fault (for example, as a result of fetching an illegal instruction) or proceeding to execute the valid instruction. *Intrinsic Branching Effects* represent such decision points.

Note

Conditional Selection instructions and Compare-and-Swap instructions generate *Intrinsic Branching Effects*.

Fault Effects, Exception Entry, and Exception Return Effects

An instruction that encounters a fault generates a Fault Effect. An instruction that encounters a fault leading to a synchronous exception generates a Fault Effect that is also an *Exception Entry Effect*, for example as a result of reading an invalid TTD or fetching an illegal instruction.

An ERET instruction generates an *Exception Return Effect.*

A Fault Effect generated by the MMU is called an *MMU Fault Effect*. See MMU fault types for a full list of faults generated by the MMU.

A Fault Effect generated by a failed Tag Check is called a *TagCheck Fault Effect*.

Note

- An instruction that encounters an asynchronous Tag Check Fault generates a TagCheck Fault Effect that is not an Exception Entry Effect.
- An instruction that encounters a synchronous Tag Check Fault generates a TagCheck Fault Effect that is also an Exception Entry Effect.

TLBI Effects, Completed TLBI Effects and Invalidation Scopes

TLBI Effects are generated by TLBI instructions.

A TLBI Effect E_1 is a *Completed TLBI Effect* if E_1 appears in program order before a DSB FULL Effect.

The *Invalidation scope* of a TLBI Effect is a set Memory Effects. For a Virtual Address *x*, a Memory Read or Write Effect to the TTD of x is in the Invalidation Scope of a TLBI Effect if *x* is in the scope of the TLBI instruction. For the full description of the scope of TLBI instructions, see TLB maintenance instruction scope.

TLBUncacheable Effects

An MMU Fault Effect is TLBUncacheable if it is one of:

- A Translation Fault Effect.
- An Access Flag Fault Effect.
- An Address Size Fault Effect.
- A synchronous External Abort on translation table walk, such that it cannot be determined whether the TTD would otherwise generate a Translation Fault, Access Flag Fault, or Address Size Fault effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A Permission Fault Effect is not TLBUncacheable.

DC CVAU Effects

DC CVAU Effects are generated by DC CVAU instructions.

IC Effects, Completed IC Effects

IC Effects are generated by IC IALLUIS, IC IALLU or IC IVAU instructions. An IC Effect E_1 is a *Completed IC Effect* if E_1 appears in program order before a DSB FULL Effect.

Empty Effects

Empty Effects are generated by instructions where a decision point, represented by a Branching Effect, leads to no further action resulting from that decision. For example, when FEAT_MTE_ASYNC is implemented and Tag Check Faults are reported asynchronously, a successful tag comparison generates an Empty Effect instead of a TagCheck Fault Effect.

Low-order Bits

The Low-order Bits of an address are the bits that remain unchanged in the translation process of an input address to an output address. The number of low-order bits that remain unchanged depends on the translation granule size.

Same Low-order Bits

Two Effects E_1 and E_2 have the Same Low-order Bits if the Low-order Bits of the address of E_1 and the Low-order Bits of the address of E_2 match.

Note

The address associated with an MMU Fault Effect is the faulting address as indicated in FAR_ELx.

Effects with a valid Physical Address

An Effect E_1 is an Effect with a valid Physical Address if one of the following applies:

- E_1 is a Memory Effect.
- E_1 is a DC Effect.
- E_1 is an IC Effect.

Note

In situations where accessing a Physical Address generates an MMU fault, the PE does not generate the corresponding Memory Effect or DC Effect or IC Effect.

Same Location

An Effect E_1 and an Effect E_2 are to the *Same Location* if one of the following applies:

- All of the following apply:
  - E_1 is an Effect with a valid Physical Address.
  - It is not the case that E_1 is a Tag Memory Effect.
  - E_1 and E_2 are to the same Physical Address.
  - E_2 is an Effect with a valid Physical Address.
  - It is not the case that E_2 is a Tag Memory Effect.
- All of the following apply:
  - E_1 is a Tag Memory Effect.
  - E_1 and E_2 are to the same Physical Address.
  - E_2 is a Tag Memory Effect.

Same Cache Line

An Effect E_1 and an Effect E_2 are to the *Same Cache Line* if:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_1 and E_2 are Memory Effects or DC CVAU Effects or IC Effects.
- E_1 and E_2 are to PAs that are on the Same Cache Line as follows from section Cache identification.

Successful Read-Modify-Write pair

Two Effects E_1 and E_2 form a *Successful Read-Modify-Write* pair if all of the following apply:

- E_1 is a Memory Read Effect.
- E_2 is a Memory Write Effect.
- E_1 and E_2 are to the Same Location.
- One of the following applies:
  - — E_1 and E_2 are generated by the same atomic instruction.
  - — E_1 and E_2 are generated by the same successful Load-Exclusive/Store-Exclusive instruction pair.

Reads-from-memory

The *Reads-from-memory* relation couples Memory Read and Write Effects to the same Location so that each Memory Read Effect is paired with exactly one Memory Write Effect in the execution of a program. A Memory Read Effect E_2 Reads-from-memory a Memory Write Effect E_1, if and only if E_1 and E_2 are to the same location and E_2 takes its data from E_1.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_1 Reads-from-memory E_2.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 and E_2 form a successful Read-Modify-Write pair.

then it is not the case that E_1 Reads-from-memory E_2.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Implicit Tag Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_1 Reads-from-memory E_2.

Coherence order

There is a per-location *Coherence order* relation that provides a total order over all Memory Write Effects to that Location, starting with a notional Memory Write Effect of the initial value. The Coherence order of a Location represents the order in which Memory Write Effects to the Location arrive at memory.

Local register read successor

An Effect E_2 is a *Local register read successor* of an Effect E_1 if all of the following apply:

- E_1 is a Register Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the same Register.
- It is not the case that all of the following apply:
  - — E_1 appears in program order before E_3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - E_1 and E_3 are to the same Register.
    - E_3 is a Register Write Effect.
    - E_3 appears in program order before E_2.
    - E_3 and E_2 are to the same Register.
- E_2 is a Register Read Effect.

Local memory read successor

An Effect E_2 is a *Local memory read successor* of an Effect E_1 if all of the following apply:

- E_1 is a Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.
- E_2 is a Memory Read Effect.
- There is no Memory Write Effect E_3 to the Same Location as E_1 such that E_1 appears in program order before E_3 and E_3 appears in program order before E_2.

Local memory write successor

An Effect E_2 is a *Local memory write successor* of an Effect E_1 if one of the following applies:

- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Effect.
        - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 appears in program order before E_2.
    - E_1 and E_2 are to the Same Location.
    - E_2 is an Explicit Memory Write Effect.
- All of the following apply:
    - E_1 is an Implicit TTD Memory Read Effect.
    - E_1 appears in program order before E_2.
    - E_1 and E_2 are to the Same Location.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.

Coherence-before, Coherence-after

A Memory Write Effect E_1 is *Coherence-before* a Memory Write Effect E_2 to the same Location if E_1 is sequenced before E_2 in the Coherence order for the Location.

A Memory Read Effect E_1 is *Coherence-before* a Memory Write Effect E_2 to the same Location if E_1 Reads-from-memory a Memory Write Effect E_3 and E_3 is Coherence-before E_2.

An Effect E_2 is *Coherence-after* an Effect E_1 if E_1 is *Coherence-before* E_2.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1.

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Explicit Memory Read Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Implicit Tag Memory Read Effect.
- E_1 appears in program order before E_2.
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1.

TLBI-before, TLBI-after

For two Effects E_1 and E_2, if and only if all of the following applies:

- E_1 is a Completed TLBI Effect.
- E_2 is an Implicit TTD Memory Read Effect.
- E_2 is in the Invalidation Scope of E_1.

then one and only one of the following applies:

- E_1 is *TLBI-after* E_2 (equivalently E_2 is *TLBI-before* E_1), or
- E_2 is *TLBI-after* E_1 (equivalently E_1 is *TLBI-before* E_2).

Note

The TLBI-after relation enumerates all possible pairs (E_1,E_2) where E_1 is a TLBI Effect and E_2 is an Implicit TTD Memory Read Effect such that E_2 is in the Invalidation Scope of E_1, and the TLBI-after relation is asymmetric.

DC-before, DC-after

For two Effects E_1 and E_2, if all of the following applies:

- E_1 is a DC CVAU Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 and E_2 are to the Same Cache Line.

then one of the following applies:

- E_1 is *DC-after* E_2 (equivalently E_2 is *DC-before* E_1), or
- E_2 is *DC-after* E_1 (equivalently E_1 is *DC-before* E_2).

Note

The DC-after relation enumerates all possible pairs (E_1, E_2) where E_1 is a DC CVAU Effect and E_2 is an Explicit Memory Write Effect such that E_1 and E_2 are to the Same Cache Line, and the relation DC-after is asymmetric.

IC-before, IC-after

For two Effects E_1 and E_2, if and only if all of the following applies:

- E_1 is a completed IC Effect.
- E_2 is an Implicit Instruction Memory Read Effect.
- E_1 and E_2 are to the Same Cache Line.

then one and only one of the following applies:

- E_1 is *IC-after* E_2 (equivalently E_2 is *IC-before* E_1), or
- E_2 is *IC-after* E_1 (equivalently E_1 is *IC-before* E_2).

Note

The IC-after relation enumerates all possible pairs (E_1,E_2) where E_1 is a completed IC Effect and E_2 is an Implicit Instruction Memory Read Effect such that E_1 and E_2 are to the Same Cache Line, and the relation IC-after is asymmetric.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-copy-atomic grouping

A *single-copy-atomic grouping* gathers Explicit Memory Effects generated by the same instruction when the architecture requires the Explicit Memory Effects to be single-copy atomic as defined in Requirements for single-copy atomicity. For example, an aligned STRH instruction generates two Explicit Memory Write Effects that are in the same single-copy-atomic grouping.

Atomicity of Successful Read-Modify-Write pair

For two Effects E_1 and E_2 if all of the following apply:

- E_1 and E_2 form a successful Read-Modify-Write pair.
- There is an Explicit Memory Write Effect E_3.
- It is not the case that all of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 and E_3 are from the same Processing Element.
  - E_2 is an Explicit Memory Effect.
  - E_2 and E_3 are from the same Processing Element.

then it is not the case that E_1 is Coherence-before E_3 and E_3 is Coherence-before E_2.

B2.3.2 Intrinsic Dependency relations

Intrinsic Dependencies are as follows:

Intrinsic Data Dependency

There is an *Intrinsic Data Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- It is not the case that E_1 is an Intrinsic Branching Effect.
- E_1 and E_2 are generated by the same instruction `I`.
- The instruction semantics for that instruction `I` indicate that the value produced by E_1 is consumed by E_2.

Note

This applies even in cases where the Intrinsic Data Dependency might be perceived to be a false dependency.

Intrinsic Control Dependency

There is an *Intrinsic Control Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- E_1 is an Intrinsic Branching Effect.
- E_1 and E_2 are generated by the same instruction `I`.
- The instruction semantics for that instruction `I` indicate that the generation of E_2 is conditional and based on the determination of E_1.

Intrinsic Order Dependency

There is an *Intrinsic Order Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- E_1 is a Memory Effect.
- E_1 and E_2 are generated by the same instruction `I`.
- The instruction semantics for that instruction `I` indicate that E_1 is generated before E_2.
- It is not the case that there is an Intrinsic Data Dependency from E_1 to E_2.
- It is not the case that there is an Intrinsic Control Dependency from E_1 to E_2.
- E_2 is a Memory Effect.

In the following, instructions whose Intrinsic dependencies are special cases that arose during the formalization of the memory model are listed. This does not preclude the existence of other special cases in the future.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.2.1 Conditional Selection instructions

The following applies to the Effects generated by `CSEL` X_d,X_n,X_m, cond

When the condition `cond` is true all of the following apply:

- Dt_1: There is an Intrinsic Data Dependency from the Register Read Effect of `PSTATE.NZCV` to the Intrinsic Branching Effect corresponding to the condition `cond`.
- Ct_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks the condition `cond` to the Register Read Effect of X_n.
- Dt_2: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Register Write Effect of X_d.

When the condition `cond` is false all of the following apply:

- Df_1: There is an Intrinsic Data Dependency from the Register Read Effect of `PSTATE.NZCV` to the Intrinsic Branching Effect corresponding to the condition `cond`.
- Cf_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect corresponding to the condition `cond` to the Register Read Effect of X_m.
- Df_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_m to the Register Write Effect of X_d.

B2.3.2.2 Compare-and-Swap instructions

The following apply to the Effects generated by `CAS` X_s,X_t,[X_n]:

When the value of the Memory Location x addressed by X_n is not equal to the value of X_s one of the following applies:

- D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
- D_2: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
- D_3: There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
- Df_1: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_s.

When the value of the Memory Location x addressed by X_n is equal to the value of X_s one of the following applies:

- All of the following apply:
  - D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
  - D_2: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
  - D_3: There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.
  - Ds_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Ds_2: There is an Intrinsic Data Dependency from Register Read Effect X_t to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Cs_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
  - Ds_{31}: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_s.
- All of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.

— D_2: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.

— D_3: There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s.

— Ds_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.

— Ds_2: There is an Intrinsic Data Dependency from the Register Read Effect of X_t to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.

— Cs_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.

— Ds_3: There is an Intrinsic Data Dependency from Register Read Effect X_s to the Register Write Effect of X_s.

— Cs_2: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Register Write Effect of X_s.

B2.3.2.3 Swap instructions

All of the following apply to the Effects generated by the `SWP` X_s, X_t,[X_n] instruction:

- D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
- D_2: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
- O_1: There is an Intrinsic Order Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.
- D_3: There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_t.
- D_4: There is an Intrinsic Data Dependency from the Register Read Effect of X_s, to the Explicit Memory Write Effect of the Memory Location x addressed by X_n.

B2.3.2.4 Branch with Link to Register instructions

All of the following apply to the Effects generated by the `BLR` X_n instruction:

- D_1: There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Branching Effect of the instruction.

B2.3.2.5 MTE Checked instructions

If FEAT_MTE2 is implemented, a Checked `LDR` X_t,[X_n] instruction has the following additional dependencies:

When Tag Check Faults are configured to raise a Synchronous Exception one of the following applies:

- If the Logical Address Tag is equal to the Allocation tag, all of the following apply:
  — Dst_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  — Cst_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n.
- If the Logical Address Tag is not equal to the Allocation tag, all of the following apply:
  — Dsf_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Csf_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the TagCheck Fault Effect.

When Tag Check Faults are configured to not raise a Synchronous Exception one of the following applies:

- If the Logical Address Tag is equal to the Allocation tag, all of the following apply:
  — Dat_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  — Cat_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the empty Effect.
- If the Logical Address Tag is not equal to the Allocation tag, all of the following apply:
  — Daf_1: There is an Intrinsic Data Dependency from the Implicit Tag Memory Read Effect of the Tag Location x of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n.
  — Caf_1: There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the Allocation Tag of x is equal to the Logical Address Tag in X_n to the TagCheck Fault Effect.

B2.3.3 Tag-check-intrinsically-before

Tag-check-before

An Effect E_1 is *Tag-check-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Tag Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3.
- E_3 is a Branching Effect.
- There is an Intrinsic Control Dependency from E_3 to E_2.
- One of the following applies:
  — E_2 is an Explicit Memory Effect.
  — E_2 is a TagCheck Fault Effect.

Tag-check-intrinsically-before

An Effect E_1 is *Tag-check-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is Tag-check-before E_2.
- It is not the case that E_2 is an Explicit Memory Read Effect.

B2.3.4 Translation-intrinsically-before

An Effect E_1 is *Translation-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3.
- E_3 is a Branching Effect.
- There is an Intrinsic Control Dependency from E_3 to E_2.
- One of the following applies:
  — E_2 is an Explicit Memory Effect.
  — E_2 is an MMU Fault Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.3.5 Fetch-intrinsically-before

An Effect E_1 is *Fetch-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3.
- E_3 is a Branching Effect.
- One of the following applies:
  - — There is an Intrinsic Control Dependency from E_3 to E_2.
  - — All of the following apply:
    - — There is an Intrinsic Control Dependency from E_3 to E_4.
    - — There is an Intrinsic Data Dependency from E_4 to E_2.

B2.3.6 Dependency relations

Dependency through registers and memory

There is a *Dependency through registers and memory* from an Effect E_1 to an Effect E_2 if one of the following applies:

- All of the following apply:
  - — It is not the case that E_1 is generated by a Store-Exclusive instruction as part of a successful Load-Exclusive/Store-Exclusive pair.
  - — E_2 is a Local register read successor of E_1.
- E_2 is a Local memory read successor of E_1.
- There is an Intrinsic Data Dependency from E_1 to E_2.
- All of the following apply:
  - — There is a Dependency through registers and memory from E_1 to E_3.
  - — There is a Dependency through registers and memory from E_3 to E_2.

Data dependency

There is a *Data dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the data value written by E_2.
- E_2 is an Explicit Memory Write Effect.

Address dependency

There is an *Address dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the address of the Location of E_2.
- One of the following applies:
  - — E_2 is an Explicit Memory Effect.
  - — E_2 is an Implicit Tag Memory Read Effect.
  - — E_2 is an Implicit TTD Memory Read Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - E_2 is a Hardware Update Effect.
  - E_2 is a TLBI Effect.
  - E_2 is a DC CVAU Effect.
  - E_2 is an IC IVAU Effect.

Control dependency

There is a *Control dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_4.
- E_4 is a Conditional Branching Effect.
- E_4 appears in program order before E_2.

Pick dependency through registers and memory

There is a *Pick dependency through registers and memory* from an Effect E_1 to an Effect E_2 if one of the following applies:

- There is a Dependency through registers and memory from E_1 to E_2.
- There is an Intrinsic Control Dependency from E_1 to E_2.
- All of the following apply:
  - There is a Pick dependency through registers and memory from E_1 to E_3.
  - There is a Pick dependency through registers and memory from E_3 to E_2.

Pick Basic dependency

There is a *Pick Basic dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_2.

Pick Data dependency

There is a *Pick Data dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- One of the following applies:
  - There is an Intrinsic Data Dependency from E_3 to E_2.
  - All of the following apply:
    - There is an Intrinsic Data Dependency from E_3 to E_4.
    - There is an Intrinsic Control Dependency from E_4 to E_2.
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the data value written by E_2.
- E_2 is an Explicit Memory Write Effect.

Pick Address dependency

There is a *Pick Address dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the address of the Location of E_2.
- One of the following applies:
  - E_2 is an Explicit Memory Effect.
  - E_2 is an Implicit Tag Memory Read Effect.
  - E_2 is an Implicit TTD Memory Read Effect.
  - E_2 is a Hardware Update Effect.
  - E_2 is a TLBI Effect.
  - E_2 is a DC CVAU Effect.
  - E_2 is an IC IVAU Effect.

Pick Control dependency

There is a *Pick Control dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3.
- E_1 appears in program order before E_3.
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_4.
- E_4 is a Conditional Branching Effect.
- E_4 appears in program order before E_2.

Pick dependency

There is a *Pick dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- One of the following applies:
  - There is a Pick Basic dependency from E_1 to E_2.
  - There is a Pick Address dependency from E_1 to E_2.
  - There is a Pick Data dependency from E_1 to E_2.
  - There is a Pick Control dependency from E_1 to E_2.
- It is not the case that E_1 and E_2 are generated by the same instruction.

B2.3.7 Ordering relations

Explicit-hazard-ordered-before

An Effect E_1 is *Explicit-hazard-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_1 appears in program order before E_3.
- E_1 and E_3 are to the Same Location.
- E_3 is an Explicit Memory Read Effect.
- One of the following applies:
  - E_3 belongs to the same single-copy-atomic grouping as E_4.
  - E_3 and E_4 are the same Effect.
- E_4 is an Explicit Memory Read Effect.
- E_4 is Coherence-before E_5.
- E_4 and E_5 are from different Processing Elements.
- One of the following applies:
  - E_5 is an Explicit Memory Write Effect.
  - E_5 is a Hardware Update Effect.
- One of the following applies:
  - E_5 belongs to the same single-copy-atomic grouping as E_2.
  - E_5 and E_2 are the same Effect.
- One of the following applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— E_2 is an Explicit Memory Write Effect.
— E_2 is a Hardware Update Effect.

Note

The Explicit-hazard-ordered-before relation does not apply when either E_1 or E_2 or both are generated by SVE instructions.

TTD-read-ordered-before

An Effect E_1 is *TTD-read-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  — E_1 is TLBI-before E_3.
  — E_3 is a TLBI Effect.
  — E_3 is DSB-ordered-before E_2.
- All of the following apply:
  — E_1 is TLBI-before E_3.
  — E_3 is a TLBI Effect.
  — E_3 is Instruction-fetch-barrier-ordered-before E_2.

TLBI-ordered-before

An Effect E_1 is *TLBI-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is TTD-read-ordered-before E_2.
- All of the following apply:
  — E_3 is Translation-intrinsically-before E_1.
  — E_3 is TTD-read-ordered-before E_2.
  — E_3 and E_2 are from different Processing Elements.
- All of the following apply:
  — E_1 is an Implicit TTD Memory Read Effect.
  — E_1 is Translation-intrinsically-before E_4.
  — E_4 and E_5 have the Same Low-order Bits.
  — E_4 appears in program order before E_5.
  — E_3 is Translation-intrinsically-before E_5.
  — E_3 is an Implicit TTD Memory Read Effect.
  — E_1 and E_3 are generated in the same translation context.
  — E_3 is TTD-read-ordered-before E_2.
  — E_3 and E_2 are from different Processing Elements.

Note

The clause of TLBI-ordered-before which refers to “Same Low-order Bits” applies in cases where an instruction I_1 which generates E_1 and E_4 appears in program order before an instruction I_2 which generates E_3 and E_5, and I_1 and I_2 access overlapping bytes in memory.

Instruction-read-ordered-before

An Effect E_1 is *Instruction-read-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  — E_1 is IC-before E_3.
  — E_3 is an IC Effect.
  — E_3 is DSB-ordered-before E_2.
- All of the following apply:
  — E_1 is IC-before E_3.
  — E_3 is an IC Effect.
  — E_3 is Instruction-fetch-barrier-ordered-before E_2.
- All of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - DIC is implemented.
  - E_1 is Coherence-before E_2.

IC-ordered-before

An Effect E_1 is *IC-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- E_1 appears in program order before E_3.
- E_3 is an Implicit Instruction Memory Read Effect.
- E_3 is Instruction-read-ordered-before E_2.

Hazard-ordered-before

An Effect E_1 is *Hazard-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Explicit-hazard-ordered-before E_2.
- E_1 is TLBI-ordered-before E_2.
- E_1 is IC-ordered-before E_2.

ETS-ordered-before

An Effect E_1 is *ETS-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - FEAT_ETS2 is implemented.
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a TLBUncacheable Fault Effect.
  - E_2 is Translation-intrinsically-before E_3.
  - E_2 is an Implicit TTD Memory Read Effect.
- All of the following apply:
  - FEAT_ETS3 is implemented.
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is an MMU Fault Effect.
  - E_2 is Translation-intrinsically-before E_3.
  - E_2 is an Implicit TTD Memory Read Effect.
- All of the following apply:
  - FEAT_ETS3 is implemented.
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a TagCheck Fault Effect that generates a synchronous exception.
  - E_2 is Tag-check-intrinsically-before E_3.
  - E_2 is an Implicit Tag Memory Read Effect.

Note

The ETS-ordered-before relation does not apply when E_1 is generated by an Advanced SIMD, floating-point, SVE, or SME instruction. It also does not apply when E_2 is an Implicit TTD Memory Read Effect generated for an instruction fetch.

Fetch-ordered-before

An Effect E_1 is *Fetch-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- E_1 appears in program order before E_2.
- It is not the case that E_2 is an Implicit Instruction Memory Read Effect.

Same-Cache-Line-ordered-before

An Effect E_1 is *Same-Cache-Line-ordered-before* an Effect E_2 if one of the following applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 appears in program order before E_2.
  - E_1 and E_2 are to the Same Cache Line.
  - E_2 is a DC CVAU Effect.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 appears in program order before E_2.
  - E_1 and E_2 are to the Same Cache Line.
  - E_2 is an Explicit Memory Effect.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 appears in program order before E_2.
  - E_1 and E_2 are to the Same Cache Line.
  - E_2 is a DC CVAU Effect.

DSB-ordered-before

An Effect E_1 is *DSB-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - One of the following applies:
    - E_1 is a Memory Effect.
    - E_1 is a DC CVAU Effect.
    - E_1 is an IC Effect.
    - E_1 is a TLBI Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DSB FULL Effect.
  - E_3 appears in program order before E_2.
  - It is not the case that one of the following applies:
    - E_2 is an Implicit TTD Memory Effect.
    - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
  - FEAT_ETS2 is implemented.
  - One of the following applies:
    - E_1 is a Memory Effect.
    - E_1 is a DC CVAU Effect.
    - E_1 is an IC Effect.
    - E_1 is a TLBI Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DSB FULL Effect.
  - E_3 appears in program order before E_2.
  - E_2 is an Implicit TTD Memory Effect.
- All of the following apply:
  - One of the following applies:
    - All of the following apply:
      - E_1 is an Explicit Memory Read Effect.
      - It is not the case that E_1 is generated by an instruction whose destination register is WZR or XZR.
    - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DSB LD Effect.
  - E_3 appears in program order before E_2.
  - It is not the case that one of the following applies:
    - E_2 is an Implicit TTD Memory Effect.
    - E_2 is an Implicit Instruction Memory Read Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following apply:
  - FEAT_ETS2 is implemented.
  - E_1 is an Explicit Memory Read Effect.
  - It is not the case that E_1 is generated by an instruction whose destination register is WZR or XZR.
  - E_1 appears in program order before E_3.
  - E_3 is a DSB LD Effect.
  - E_3 appears in program order before E_2.
  - E_2 is an Implicit TTD Memory Effect.
- All of the following apply:
  - E_1 is an Explicit Memory Write Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DSB ST Effect.
  - E_3 appears in program order before E_2.
  - It is not the case that one of the following applies:
    - E_2 is an Implicit TTD Memory Effect.
    - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
  - FEAT_ETS2 is implemented.
  - E_1 is an Explicit Memory Write Effect.
  - E_1 appears in program order before E_3.
  - E_3 is a DSB ST Effect.
  - E_3 appears in program order before E_2.
  - E_2 is an Implicit TTD Memory Effect.

Note

Regardless of whether FEAT_ETS2 is implemented, the DSB-ordered-before relation does not apply when E_2 is an Implicit TTD Memory Read Effect generated for an instruction fetch.

Instruction-fetch-barrier-ordered-before

An Effect E_1 is *Instruction-fetch-barrier-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Explicit Memory Read Effect.
  - There is a Control dependency from E_1 to E_3.
  - E_3 is an Instruction Fetch Barrier Effect.
  - E_3 appears in program order before E_2.
- All of the following apply:
  - E_1 is an Explicit Memory Read Effect.
  - There is a Pick Control dependency from E_1 to E_3.
  - E_3 is an Instruction Fetch Barrier Effect.
  - E_3 appears in program order before E_2.
- All of the following apply:
  - E_1 is an Explicit Memory Read Effect.
  - There is an Address dependency from E_1 to E_3.
  - E_3 is an Explicit Memory Effect.
  - E_3 appears in program order before E_4.
  - E_4 is an Instruction Fetch Barrier Effect.
  - E_4 appears in program order before E_2.
- All of the following apply:
  - E_1 is an Explicit Memory Read Effect.
  - There is a Pick Address dependency from E_1 to E_3.
  - E_3 is an Explicit Memory Effect.
  - E_3 appears in program order before E_4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Explicit Memory Read Effect.
    - There is a Pick Address dependency from E_1 to E_3.
    - One of the following applies:
        - E_3 is Tag-check-intrinsically-before E_4.
        - E_3 is Translation-intrinsically-before E_4.
    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.
- All of the following apply:
    - E_1 is DSB-ordered-before E_3.
    - E_3 is an Instruction Fetch Barrier Effect.
    - E_3 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Implicit TTD Memory Read Effect.
    - E_1 is Translation-intrinsically-before E_3.
    - E_3 is an Instruction Fetch Barrier Effect.
    - E_3 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 is Tag-check-intrinsically-before E_3.
    - E_3 is an Instruction Fetch Barrier Effect.
    - E_3 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Implicit TTD Memory Read Effect.
    - E_1 is Translation-intrinsically-before E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_4.
    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.
- All of the following apply:
    - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 is Tag-check-before E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_4.
    - E_4 is an Instruction Fetch Barrier Effect.
    - E_4 appears in program order before E_2.

Dependency-ordered-before

An Effect E_1 is *Dependency-ordered-before* an Effect E_2 if one of the following applies:

- There is an Address dependency from E_1 to E_2.
- There is a Data dependency from E_1 to E_2.
- All of the following apply:
    - There is a Control dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
        - E_2 is a TLBI Effect.
        - E_2 is a DC CVAU Effect.
        - E_2 is an IC Effect.
- All of the following apply:
    - There is an Address dependency from E_1 to E_3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
- All of the following apply:
    - There is an Address dependency from E_1 to E_3.
    - E_3 is an Explicit Memory Effect.
    - E_2 is a Local memory read successor of E_3.
    - One of the following applies:
        - E_2 is an Explicit Memory Read Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
    - There is a Data dependency from E_1 to E_3.
    - E_3 is an Explicit Memory Effect.
    - E_2 is a Local memory read successor of E_3.
    - One of the following applies:
        - E_2 is an Explicit Memory Read Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
    - E_1 is an Implicit TTD Memory Read Effect.
    - E_1 is Translation-intrinsically-before E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
- All of the following apply:
    - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 is Tag-check-before E_3.
    - E_3 is an Explicit Memory Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.

Pick-ordered-before

An Effect E_1 is *Pick-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
    - There is a Pick Address dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
        - E_2 is a TLBI Effect.
        - E_2 is a DC CVAU Effect.
        - E_2 is an IC Effect.
- There is a Pick Data dependency from E_1 to E_2.
- All of the following apply:
    - There is a Pick Control dependency from E_1 to E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Write Effect.
        - E_2 is a Hardware Update Effect.
        - E_2 is a TLBI Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

        - E_2 is a DC CVAU Effect.
        - E_2 is an IC Effect.
    - All of the following apply:
        - There is a Pick Address dependency from E_1 to E_3.
        - E_3 is an Explicit Memory Effect.
        - E_3 appears in program order before E_2.
        - One of the following applies:
            - E_2 is an Explicit Memory Write Effect.
            - E_2 is a Hardware Update Effect.

Atomic-ordered-before

E_1 and E_2 form a successful Read-Modify-Write pair if and only if E_1 and E_2 are generated by the same atomic instruction or the same successful Load-Exclusive/Store-Exclusive instruction pair to the Same Location.

An Effect E_1 is *Atomic-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
    - E_1 is an Explicit Memory Effect.
    - E_1 and E_2 form a successful Read-Modify-Write pair.
    - E_2 is an Explicit Memory Effect.
- All of the following apply:
    - E_1 is an Explicit Memory Effect.
    - E_1 and E_3 form a successful Read-Modify-Write pair.
    - E_2 is a Local memory read successor of E_3.
    - One of the following applies:
        - E_2 is an Explicit Memory Read Effect with Acquire semantics.
        - E_2 is an Explicit Memory Read Effect with AcquirePC semantics.
- All of the following apply:
    - E_1 is an Implicit TTD Memory Read Effect.
    - E_1 and E_2 form a successful Read-Modify-Write pair.
    - E_2 is a Hardware Update Effect.

Barrier-ordered-before

An Effect E_1 is *Barrier-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
    - One of the following applies:
        - E_1 is an Explicit Memory Effect.
        - E_1 is an Implicit Tag Memory Read Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DMB FULL Effect.
    - E_3 appears in program order before E_2.
    - One of the following applies:
        - E_2 is an Explicit Memory Effect.
        - E_2 is an Implicit Tag Memory Read Effect.
        - E_2 is an MMU Fault Effect.
- All of the following apply:
    - E_1 is an Explicit Memory Effect.
    - E_1 appears in program order before E_3.
    - E_3 is a DMB FULL Effect.
    - E_3 appears in program order before E_2.
    - E_2 is a DC CVAU Effect.
- All of the following apply:
    - E_1 is a DC CVAU Effect.
    - E_1 appears in program order before E_3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- — E_3 is a DMB FULL Effect.
- — E_3 appears in program order before E_2.
- — E_2 is an Explicit Memory Effect.

- All of the following apply:
  - — E_1 is a DC CVAU Effect.
  - — E_1 appears in program order before E_3.
  - — E_3 is a DMB FULL Effect.
  - — E_3 appears in program order before E_2.
  - — E_2 is a DC CVAU Effect.
- All of the following apply:
  - — One of the following applies:
    - — All of the following apply:
      - · E_1 is an Explicit Effect.
      - · E_1 is a Memory Read Effect.
      - · It is not the case that E_1 is generated by an instruction whose destination register is WZR or XZR.
    - — E_1 is an Implicit Tag Memory Read Effect.
  - — E_1 appears in program order before E_3.
  - — E_3 is a DMB LD Effect.
  - — E_3 appears in program order before E_2.
  - — One of the following applies:
    - — E_2 is an Explicit Memory Effect.
    - — E_2 is an Implicit Tag Memory Read Effect.
    - — E_2 is an MMU Fault Effect.
- All of the following apply:
  - — E_1 is an Explicit Memory Write Effect.
  - — E_1 appears in program order before E_3.
  - — E_3 is a DMB ST Effect.
  - — E_3 appears in program order before E_2.
  - — One of the following applies:
    - — E_2 is an Explicit Memory Write Effect.
    - — E_2 is an MMU Fault Effect.
- All of the following apply:
  - — E_1 is an Explicit Memory Write Effect with Release semantics.
  - — E_3 and E_1 are generated by an atomic instruction.
  - — E_3 is an Explicit Memory Read Effect with Acquire semantics.
  - — E_1 appears in program order before E_2.
  - — One of the following applies:
    - — E_2 is an Explicit Memory Effect.
    - — E_2 is an Implicit Tag Memory Read Effect.
    - — E_2 is an MMU Fault Effect.
- All of the following apply:
  - — E_1 is an Explicit Memory Write Effect with Release semantics.
  - — E_1 appears in program order before E_2.
  - — E_2 is an Explicit Memory Read Effect with Acquire semantics.
- All of the following apply:
  - — One of the following applies:
    - — E_1 is an Explicit Memory Read Effect with Acquire semantics.
    - — E_1 is an Explicit Memory Read Effect with AcquirePC semantics.
  - — E_1 appears in program order before E_2.
  - — One of the following applies:
    - — E_2 is an Explicit Memory Effect.
    - — E_2 is an Implicit Tag Memory Read Effect.
    - — E_2 is an MMU Fault Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following apply:
  - E_1 is an Explicit Memory Read Effect with AcquirePC semantics.
  - There is an Intrinsic Order Dependency from E_1 to E_2.
  - E_2 is an Explicit Memory Read Effect with AcquirePC semantics.
- All of the following apply:
  - One of the following applies:
    - E_1 is an Explicit Memory Effect.
    - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 appears in program order before E_2.
  - E_2 is an Explicit Memory Write Effect with Release semantics.
- All of the following apply:
  - E_1 is an Explicit Memory Write Effect with Release semantics.
  - There is an Intrinsic Order Dependency from E_1 to E_2.
  - E_2 is an Explicit Memory Write Effect with Release semantics.

Locally-ordered-before

An Effect E_1 is *Locally-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Tag-check-intrinsically-before E_2.
- E_1 is Translation-intrinsically-before E_2.
- E_1 is Fetch-intrinsically-before E_2.
- E_1 is ETS-ordered-before E_2.
- E_1 is Fetch-ordered-before E_2.
- E_1 is Same-Cache-Line-ordered-before E_2.
- E_1 is DSB-ordered-before E_2.
- E_1 is Instruction-fetch-barrier-ordered-before E_2.
- E_2 is a Local memory write successor of E_1.
- All of the following apply:
  - E_3 is a Local memory write successor of E_1.
  - E_3 belongs to the same single-copy-atomic grouping as E_2.
- E_1 is Dependency-ordered-before E_2.
- E_1 is Pick-ordered-before E_2.
- E_1 is Atomic-ordered-before E_2.
- E_1 is Barrier-ordered-before E_2.
- All of the following apply:
  - E_1 is Locally-ordered-before E_3.
  - E_3 is Locally-ordered-before E_2.

Pick-locally-ordered-before

An Effect E_1 is *Pick-locally-ordered-before* an Effect E_2 if all of the following apply:

- There is a Pick dependency from E_1 to E_3.
- E_3 is Locally-ordered-before E_2.
- E_2 is an Explicit Memory Write Effect.

Locally-hardware-required-ordered-before

An Effect E_1 is *Locally-hardware-required-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Locally-ordered-before E_2.
- E_1 is Pick-locally-ordered-before E_2.
- All of the following apply:
  - E_1 is Locally-hardware-required-ordered-before E_3.
  - E_3 is Locally-hardware-required-ordered-before E_2.

Hardware-required-ordered-before

An Effect E_1 is *Hardware-required-ordered-before* an Effect E_2 if one of the following applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_1 is Locally-hardware-required-ordered-before E_2.
- E_1 is Hazard-ordered-before E_2.

B2.3.8 Observation relations

Explicit-Observed-by

An Effect E_1 is *Explicit-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_2 Reads-from-memory E_1.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Explicit Memory Effect.
- All of the following apply:
  - E_1 is an Explicit Memory Effect.
  - E_1 is Coherence-before E_2.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Explicit Memory Effect.

Tag-Observed-by

An Effect E_1 is *Tag-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Explicit Memory Write Effect.
  - E_2 Reads-from-memory E_1.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
  - E_1 is an Implicit Tag Memory Read Effect.
  - E_1 is Coherence-before E_2.
  - E_1 and E_2 are from different Processing Elements.
  - E_2 is an Explicit Memory Write Effect.

TLBUncacheable-coherence-before

An Effect E_1 is *TLBUncacheable-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- E_1 is Translation-intrinsically-before E_3.
- E_3 is a TLBUncacheable Fault Effect.
- E_1 is Coherence-before E_2.
- One of the following applies:
  - E_2 is an Explicit Memory Write Effect.
  - E_2 is a Hardware Update Effect.

Hardware-Update-coherence-before

An Effect E_1 is *Hardware-Update-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_1 is Coherence-before E_2.
- E_2 is a Hardware Update Effect.

TLBI-coherence-before

An Effect E_1 is *TLBI-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is a TLBI Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_1 is TLBI-before E_3.
- E_3 is an Implicit TTD Memory Read Effect.
- E_3 is Coherence-before E_2.
- E_2 is a Memory Write Effect.

TTD-Observed-by

An Effect E_1 is *TTD-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
  - E_1 is an Implicit TTD Memory Effect.
  - E_2 Reads-from-memory E_1.
- All of the following apply:
  - E_2 Reads-from-memory E_1.
  - E_2 is an Implicit TTD Memory Effect.
- E_1 is TLBUncacheable-coherence-before E_2.
- E_1 is Hardware-Update-coherence-before E_2.
- All of the following apply:
  - E_1 is a Hardware Update Effect.
  - E_1 is Coherence-before E_2.
  - E_2 is a Memory Write Effect.
- All of the following apply:
  - E_1 is a Memory Write Effect.
  - E_1 is Coherence-before E_2.
  - E_2 is a Hardware Update Effect.
- E_1 is TLBI-coherence-before E_2.

IC-Coherence-before

An Effect E_1 is *IC-coherence-before* an Effect E_2 if one of the following applies:

- All of the following apply:
  - it is not the case that DIC is implemented and it is not the case that IDC is implemented.
  - E_1 is an IC Effect.
  - E_1 is IC-before E_3.
  - E_3 is an Implicit Instruction Memory Read Effect.
  - E_3 is Coherence-before E_4.
  - E_4 is a Memory Write Effect.
  - E_4 is DC-before E_2.
  - E_2 is a DC CVAU Effect.
- All of the following apply:
  - it is not the case that DIC is implemented and IDC is implemented.
  - E_1 is an IC Effect.
  - E_1 is IC-before E_3.
  - E_3 is an Implicit Instruction Memory Read Effect.
  - E_3 is Coherence-before E_2.
  - E_2 is a Memory Write Effect.
- All of the following apply:
  - DIC is implemented and IDC is implemented.
  - E_1 is an Implicit Instruction Memory Read Effect.
  - E_1 is Coherence-before E_2.
  - E_2 is a Memory Write Effect.

Instruction-Observed-by

An Effect E_1 is *Instruction-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - E_2 Reads-from-memory E_1.
  - E_2 is an Implicit Instruction Memory Read Effect.
- E_1 is IC-before E_2.
- All of the following apply:
  - E_1 is a DC CVAU Effect.
  - E_1 is DC-before E_2.
  - E_2 is a Memory Write Effect.
- All of the following apply:
  - E_1 is a Memory Write Effect.
  - E_1 is DC-before E_2.
  - E_2 is a DC CVAU Effect.
- E_1 is IC-coherence-before E_2.

Observed-by

An Effect E_1 is *Observed-by* an Effect E_2 if one of the following applies:

- E_1 is Explicit-Observed-by either E_2 or an Effect which belongs to the same single-copy-atomic grouping as E_2.
- E_1 is Tag-Observed-by either E_2 or an Effect which belongs to the same single-copy-atomic grouping as E_2.
- E_1 is TTD-Observed-by E_2.
- E_1 is Instruction-Observed-by E_2.

B2.3.9 Definition of an Architecturally Allowed Execution

An Architecturally Allowed Execution must satisfy the *External visibility requirement*, which is formulated in terms of an ordering relation called *Ordered-before*. An execution that does not satisfy the external visibility requirement is an Architecturally Forbidden Execution.

Ordered-before

An Effect E_1 is *Ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Hardware-required-ordered-before E_2.
- E_1 is Observed-by E_2.
- All of the following apply:
  - E_1 is Ordered-before E_3.
  - E_3 is Ordered-before E_2.

External visibility requirement

The external visibility requirement is defined as follows: an Architecturally Allowed Execution must not exhibit a cycle in the Ordered-before relation.

In particular, for an Effect E_1 that is Ordered-before an Effect E_2, the External visibility requirement requires that E_2 is not Observed-by E_1.

B2.3.10 Alternative formulations of the External visibility requirement for Explicit Memory Effects

B2.3.10.1 Additional ordering relations

Single-copy-atomic-ordered-before

A Memory Read Effect E_1 is *single-copy-atomic-ordered-before* another Memory Read Effect E_2 if all the following apply:

- E_1 belongs to the same single-copy-atomic grouping as E_2.
- E_1 Reads-from-memory a Memory Write Effect E_3.
- E_1 and E_3 are from different Processing Elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- E_2 Reads-from-memory a Memory Write Effect E_4.
- E_2 and E_4 are from the same Processing Element.

B2.3.10.2 External completion requirement for Explicit Memory Effects

The *External completion requirement for Explicit Memory Effects* is an alternative way of formulating the External visibility requirement, in the context of Explicit Memory Effects only.

The Completes-before order is a total order that corresponds to the order in which Memory Effects complete within the system. Effects E_1 and E_2 constitute a single entry in the Completes-before order, if any of the following applies:

- All of the following apply:
  - E_1 is a Memory Write Effect.
  - E_2 is a Memory Write Effect.
  - E_1 belongs to the same single-copy-atomic grouping as E_2.
- All of the following apply:
  - E_1 is a Memory Read Effect.
  - E_2 is a Memory Read Effect.
  - E_1 belongs to the same single-copy-atomic grouping as E_2.
  - E_1 Reads-from-memory a Memory Write Effect E_3.
  - E_1 and E_3 are from different Processing Elements.
  - E_2 Reads-from-memory a Memory Write Effect E_4.
  - E_2 and E_4 are from different Processing Elements.
- All of the following apply:
  - E_1 is a Memory Read Effect.
  - E_2 is a Memory Read Effect.
  - E_1 belongs to the same single-copy-atomic grouping as E_2.
  - E_1 Reads-from-memory a Memory Write Effect E_3.
  - E_1 and E_3 are from the same Processing Element.
  - E_2 Reads-from-memory a Memory Write Effect E_4.
  - E_2 and E_4 are from the same Processing Element.
  - E_3 belongs to the same single-copy-atomic grouping as E_4.

All other Memory Read Effects constitute distinct entries in the Completes-before order.

Completes-before

A Memory Read or Write Effect E_1 completes-before a Memory Read or Write Effect E_2 if E_1 appears in the *Completes-before* order before E_2.

Deriving Reads-from-memory and Coherence order from the Completes-before order

The Reads-from-memory relation can be derived from the Completes-before order by specifying which Memory Write Effect every Memory Read Effect gets its value from. Specifically, for a Memory Read Effect E_1 one of the following must apply:

- If all of the following apply:
  - There is a Memory Write Effect E_2.
  - E_1 is a Local memory read successor of E_2.
  - E_1 Completes-before E_2.
  - There is no Memory Write Effect E_3 to the Same Location in the Completes-before order between E_2 and an Explicit Memory Read Effect E_4 to the Same Location that appears in program order before E_1.

  then it must be that E_1 Reads-from-memory E_2.
- If all of the following apply:
  - There is a Memory Write Effect E_2 (possibly representing the writing of the initial value to the Location).
  - E_2 and E_1 are to the Same Location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— E_2 Completes-before E_1.
— It is not the case that E_2 Completes-before a Memory Write Effect E_3 and E_1 is the Local memory read successor of E_3.
— There is no Memory Write Effect E_4 to the Same Location in the Completes-before order between E_2 and E_1.
— There is no Memory Write Effect E_5 to the Same Location in the Completes-before order between E_2 and a Memory Read Effect E_6 to the Same Location that appears in program order before E_1.

then it must be that E_1 Reads-from-memory E_2.

The Coherence order can be derived from the Completes-before order by letting the Coherence order of Memory Write Effects to a memory location be the order in which those Memory Write Effects appear in the Completes-before order. The final value of each memory location is therefore determined by the final Memory Write Effect to each Location in the Completes-before order. If no such Memory Write Effect exists for a given Location, the final value is the initial value of that Location.

External completion requirement for Explicit Memory Effects

The External completion requirement for Explicit Memory Effects requires that an Memory Effect E_1 Completes-before a Memory Effect E_2 if any of the following statements are true:

- E_1 is Locally-hardware-required-ordered-before E_2.
- E_1 is a Memory Read Effect, E_2 is a Memory Read Effect, and E_1 is single-copy-atomic-ordered-before E_2.

B2.3.10.3 External global completion requirement for Explicit Memory Effects

The *External global completion requirement for Explicit Memory Effects* is an alternative way of formulating the External visibility requirement, in the context of Explicit Memory Effects only.

The Globally-completes-before order is a total order that corresponds to the order in which Memory Effects complete within the system. Effects E_1 and E_2 constitute a single entry in the Globally-completes-before order, if any of the following applies:

- All of the following apply:
  — E_1 is a Memory Write Effect.
  — E_2 is a Memory Write Effect.
  — E_1 belongs to the same single-copy-atomic grouping as E_2.
- All of the following apply:
  — E_1 is a Memory Read Effect.
  — E_2 is a Memory Read Effect.
  — E_1 belongs to the same single-copy-atomic grouping as E_2.
  — E_1 Reads-from-memory a Memory Write Effect E_3.
  — E_1 and E_3 are from different Processing Elements.
  — E_2 Reads-from-memory a Memory Write Effect E_4.
  — E_2 and E_4 are from different Processing Elements.
- All of the following apply:
  — E_1 is a Memory Read Effect.
  — E_2 is a Memory Read Effect.
  — E_1 belongs to the same single-copy-atomic grouping as E_2.
  — E_1 Reads-from-memory a Memory Write Effect E_3.
  — E_1 and E_3 are from the same Processing Element.
  — E_2 Reads-from-memory a Memory Write Effect E_4.
  — E_2 and E_4 are from the same Processing Element.
  — E_3 belongs to the same single-copy-atomic grouping as E_4.

All other Memory Read Effects constitute distinct entries in the Globally-completes-before order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Globally-completes-before

A Memory Read or Write Effect E_1 *Globally-completes-before* a Memory Read or Write Effect E_2 if E_1 appears in the Globally-completes-before order before E_2.

Deriving Reads-from-memory and Coherence order from the Globally-completes-before order

The Reads-from-memory relation can be derived from the Globally-completes-before order by specifying which Memory Write Effect every Memory Read Effect gets its value from. Specifically, for a Memory Read Effect E_1 it must be that if all of the following applies:

- There is Memory Write Effect E_2 (possibly representing the writing of the initial value to the Location).
- E_2 Globally-completes-before E_1.
- E_1 and E_2 are to the Same Location.
- There is no Memory Write Effect E_3 to the Same Location in the Globally-completes-before order between E_2 and E_1.
- There is no Memory Write Effect E_4 to the Same Location in the Globally-completes-before order between E_2 and an Explicit Memory Read Effect E_5 to the Same Location that appears in program order before E_1.

then it must be that E_1 Reads-from-memory E_2.

The Coherence order can be derived from the Globally-completes-before order by letting the Coherence order of Memory Write Effects to a memory location be the order in which those Memory Write Effects appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final Memory Write Effect to each Location in the Globally-completes-before order. If no such Memory Write Effect exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

The External global completion requirement requires that a Memory Effect E_1 Globally-completes-before a Memory Effect E_2 if any of the following statements are true:

- E_1 is Locally-hardware-required-ordered-before E_2 and one of the following applies:
  - E_1 is a Memory Write Effect.
  - E_1 is a Memory Read Effect, and one of the following applies:
    - E_1 Reads-from-memory a Memory Write Effect E_3, and E_1 and E_3 are from different Processing Elements, or.
    - E_1 Reads-from-memory a Memory Write Effect E_3, E_1, and E_3 are from the same Processing Element, and E_3 is Locally-hardware-required-ordered-before E_2.
- E_1 is a Memory Read Effect, E_2 is a Memory Read Effect, and E_1 is single-copy-atomic-ordered before E_2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.4 Additional ordering requirements outside of the scope of the formal concurrency model

The following sections present ordering rules which have been architected but not yet formalized. As a consequence they do not appear in the formal concurrency model, nor in the corresponding transliteration in Ordering requirements defined by the formal concurrency model. They will be integrated into Ordering requirements defined by the formal concurrency model in due course.

B2.4.1 Completion and endpoint ordering

For all memory, the completion rules are defined as:

- A Memory Read effect E_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any Memory Write Effect to the same Location by an observer within the shareability domain will be Coherence-after E_1.
  - All Implicit TTD Memory Read Effects associated with E_1 are complete for that shareability domain.
- A Memory Write effect E_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any Memory Write Effect to the same Location by an observer within the shareability domain will be Coherence-after E_1.
  - Any Memory Read Effect to the same Location by an observer within the shareability domain will be such that either Reads-from E_1 or Reads-from a Memory Write effect that is Coherence-after E_1.
  - All Implicit TTD Memory Read Effects associated with E_1 are complete for that shareability domain.
- An Implicit TTD Memory Read Effect is complete for a shareability domain when the memory accesses associated with the Implicit TTD Memory Read Effect, including Hardware Update Effects, are complete for that shareability domain, and the TLB is updated.
- A cache maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and Implicit TTD Memory Read Effects that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side effects is complete only when the read or write both:

- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side effects, whether they affect other peripheral devices, PEs, or memory.

Interaction between observers in a system is not restricted to communication via shared variables in coherent memory. For example, an observer could configure an interrupt controller to raise an interrupt on another observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different observers. When these forms of interaction are used in conjunction with shared variables, a `DSB` instruction can be used to enforce ordering between them.

Note

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

Note

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

B2.4.1.1 Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and writes to a peripheral which is defined as the Peripheral arrival order:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Memory-mapped peripheral

A *Memory-mapped peripheral* occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

Peripheral arrival order

The *Peripheral arrival order* of a Memory-mapped peripheral is a total order on all Memory Read Effects and Memory Write Effects to that peripheral.

For a Memory Effect E_1 and a Memory Effect E_2 to the same peripheral if any of the following applies:

- All of the following apply:
  - Any of the following applies:
    - E_1 is to Device-nGnRE Memory.
    - E_1 is to Device-nGnRnE Memory.
  - Any of the following applies:
    - E_2 is to Device-nGnRE Memory.
    - E_2 is to Device-nGnRnE Memory.
  - Any of the following applies:
    - FEAT_XS is not implemented.
    - All of the following apply:
      - FEAT_XS is implemented.
      - E_1 has the XS attribute.
      - E_2 has the XS attribute.
    - All of the following apply:
      - FEAT_XS is implemented.
      - E_1 does not have the XS attribute.
      - E_2 does not have the XS attribute.
  - E_1 is in program-order before E_2.
- All of the following apply:
  - Any of the following applies:
    - E_1 is to Device-nGRE Memory.
    - E_1 is to Device-nGnRE Memory.
    - E_1 is to Device-nGnRnE Memory.
  - Any of the following applies:
    - E_2 is to Device-nGRE Memory.
    - E_2 is to Device-nGnRE Memory.
    - E_2 is to Device-nGnRnE Memory.
  - E_1 is Ordered-before E_2.
- All of the following apply:
  - Any of the following applies:
    - E_1 is to Device Memory.
    - E_1 is to Non-cacheable Normal Memory.
  - Any of the following applies:
    - E_2 is to Device Memory.
    - E_2 is to Non-cacheable Normal Memory.
  - E_1 is Barrier-ordered-before E_2.

Then E_1 is inserted in the Peripheral arrival order for the peripheral before E_2 and it is not permitted that E_1 and E_2 are gathered into a single transaction. Otherwise, exactly one of following applies:

- E_1 and E_2 are gathered into a single transaction and appear as one Effect in Peripheral arrival order for the peripheral.
- E_1 is inserted in the Peripheral arrival order for the peripheral before E_2.
- E_2 is inserted in the Peripheral arrival order for the peripheral before E_1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When FEAT_XS is implemented, if E_1 and E_2 are to Device-nGnRE Memory or to Device-nGnRnE Memory and E_1 and E_2 are within the same Memory-mapped peripheral, but E_1 and E_2 use different XS attribute, the order of arrival at the endpoint is not defined by the architecture.

Out-of-band-ordered-before

An Effect E_1 is *Out-of-band-ordered-before* an Effect E_2 if and only if all of the following apply:

- E_1 is a Memory Effect.
- Any of the following applies:
  - E_1 is DSB-ordered-before E_3.
  - E_1 is Instruction-fetch-barrier-ordered-before E_3.
- There is an IMPLEMENTATION DEFINED instruction sequence beginning with E_3 indirectly leading to the generation of E_2.
- E_2 is a Memory Effect.

If an Effect E_1 is Out-of-band-ordered-before an Effect E_2, then E_1 is Ordered-before E_2.

Note

Arm expects that, in most systems with early acknowledgments, those acknowledgments will come from a point at or after the point that establishes global visibility. This is expected in such systems to enable the acknowledgments to be used as part of the mechanisms to implement the ordering requirements of the Arm memory model.

B2.4.2 Metadata Dependencies

I_{YRYHC} The execution of the following instructions create ordering relations for the purposes of the Arm memory model:

- `IRG`.
- `PACDA`.
- `PACDB`.
- `AUTDA`.
- `AUTDB`

R_{TVHSL} If FEAT_MTE is implemented, the execution of an `IRG Xd,Xn,Xm` instruction creates:

- A Dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.
- If Allocation Tag Access is not prevented for the current Exception level, a Dependency through registers and memory from the Register Read Effect of Xm to Register Write Effect of Xd.

R_{MSWTM} If FEAT_PAuth is implemented, the execution of a `PACDA Xd,Xn` instruction creates:

- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.
- If authentication using the APDAKey_EL1 key is enabled, a Dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.

R_{JMNSC} If FEAT_FPAC is implemented, the execution of a `PACDB Xd,Xn` instruction creates:

- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.
- If authentication using the APDBKey_EL1 key is enabled, a Dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.

R_{XYRJT} If FEAT_FPAC is implemented, the execution of an `AUTDA Xd,Xn` instruction creates:

- If authentication using the APDAKey_EL1 key is enabled, a Pick dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.
- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SPBZX} If FEAT_FPAC is implemented, the execution of an `AUTDB Xd,Xn` instruction creates:

- If authentication using the APDBKey_EL1 key is enabled, a Pick dependency through registers and memory from the Register Read Effect of Xn to Register Write Effect of Xd.
- A Dependency through registers and memory from the Register Read Effect of Xd to Register Write Effect of Xd.

B2.4.3 Limited ordering regions

FEAT_LOR introduces *limited ordering regions* (LORegions), which allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the PA map as observed by a set of observers.

LORegions are defined in the Non-secure physical address space. LORegions cannot be defined in the Secure, Realm, or Root physical address spaces.

This feature is supported in AArch64 state only.

B2.4.3.1 Specification of the LORegions

The LORegions are defined in the Non-secure physical memory map using a set of LORegion descriptors. The number of LORegion descriptors is IMPLEMENTATION DEFINED, and can be discovered by reading the LORID_EL1 register.

Each LORegion descriptor consists of:

- A tuple of the following values:
  - A Start Address.
  - An End Address.
  - An LORegion Number.
- Valid bit which indicates whether that LORegion descriptor is valid.

A memory location lies within the LORegion identified by the LORegion Number if the PA lies between the Start Address and the End Address, inclusive. The Start Address must be defined to be aligned to 64KB and the End Address must be defined as the top byte of a 64KB block of memory.

It is permitted for multiple LORegion descriptors with non-overlapping address ranges to be configured with the same LORegion Number.

The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1, and LORC_EL1 registers in the System register space. These registers describe only memory addresses in the Non-secure memory map. These registers are UNDEFINED if accessed when SCR_EL3.NS == 0.

If a LoadLOAcquire or a StoreLORelease does not match with any LORegion, then:

- The LoadLOAcquire will behave as a Load-Acquire, and will be ordered in the same way with respect to all accesses, independent of their LORegions.
- The StoreLORelease will behave as a Store-Release, and will be ordered in the same way with respect to all accesses, independent of their LORegions.

Note

If no LORegions are implemented, then the LoadLOAcquire and StoreLORelease will therefore behave as a Load-Acquire and Store-Release.

B2.4.4 SVE memory ordering relaxations

I_{CTNGV} The Arm memory model is relaxed for reads and writes generated by SVE load and store instructions.

R_{QLJPC} When two reads generated by SVE vector load instructions have an address dependency, the dependency does not contribute to the dependency-ordered-before relation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{YMBMZ} When a pair of reads access the same location, and at least one of the reads is generated by an SVE load instruction, for a given observer, the hazard-ordered-before relation does not apply to the pair of reads.

R_{CJHWV} When a single SVE vector store instruction generates multiple writes to the same location, the instruction ensures that these writes appear in the coherence order for that location, in order of increasing vector element number. No other ordering restrictions apply to memory effects generated by the same SVE store instruction.

R_{LVXTJ} If a single SVE load instruction generates multiple reads, the order in which the reads for different elements and registers appear is not architecturally defined.

R_{VMDYZ} If an address dependency exists between two Read Memory and an SVE non-temporal vector load instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

I_{CCWGN} For any SVE load or store instruction that generates multiple single-copy atomic accesses to Normal or Device memory, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic memory element access sizes.

B2.4.5 Streaming SVE mode memory ordering relaxations

R_{HBBTV} If a pair of memory reads access the same location, and at least one of the reads is generated by an Advanced SIMD&FP load instruction, for a given observer, the hazard-ordered-before relation does not apply to the pair of reads when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

See also:

- Streaming SVE mode.

B2.4.6 Ordering rules for GCS

I_{QNKNL} If FEAT_GCS is implemented, see Guarded Control Stack data accesses for relevant ordering rules.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.5 Restrictions on the effects of speculation

A *permitted-speculative-next-PC* is one of:

- For any instruction that is neither an Unconditional branch (immediate) nor an exception generating instruction, the PC plus four.
- For any instruction, an exception vector.
- For a direct branch, the PC plus the offset contained in the immediate field of the opcode.
- For an indirect branch, any address.

A *permitted-speculatively-executed-instruction* is the speculative execution of an instruction at a permitted-speculative-next-PC where that PC was generated from one of the following:

- An architecturally executed instruction.
- A permitted-speculatively-executed-instruction.

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context Synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context Synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context Synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context Synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a Translation fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.
- A speculative Explicit Memory Read Effect generated by an instruction (I_2) running in a hardware-defined context ($context_2$) will not speculatively read data from a speculative Explicit Memory Write Effect generated by an instruction (I_1) running in a different hardware-defined context ($context_1$) if I_1 generates an MMU fault Effect instead of the Explicit Memory Write Effect.
- When data is loaded under speculation with a GPC fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence, and the execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.
- When stage 2 translation is enabled and a stage 1 translation table entry is loaded under speculation with a GPC fault, the output address or Next-level table address from the entry cannot be used to form an address to be used by other fetches in the translation table walk.
- Granule protection checks apply to speculative instruction fetch and speculative execution. Any instruction fetched under speculation with a GPC fault:
  - Cannot cause an update to any architectural or micro-architectural state as a result of speculative execution of the instruction, where the update of the state is dependent on the content of the instruction.
  - Cannot be stored in a cache that is not affected by `DC PAPA` operations.
- If GPCCR_EL3.GPCP == 0, data from a translation table walk for which the granule protection check for the address being accessed has not been architecturally resolved can not be used to form an address for a subsequent read access or for generating syndrome information, until the granule protection check has passed.

  Note

  Permitting read accesses to locations for which the granule protection check has not been architecturally resolved means the GPT does not protect non-idempotent locations from these speculative read operations.

- When data is loaded under speculation from a location that does not have a valid translation for the translation regime being speculated in, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When data is loaded as a result of speculative accesses made after `TLBI` + `DSB` + `ERET` using a translation that was invalidated by the TLBI, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the microarchitectural state.
- Changes to Special-purpose registers can occur speculatively.
- Accesses to Device memory are subject to many restrictions on the effects of Speculation. See Device memory.
- Execute-never controls apply to speculative instruction fetching. See Effects on instruction execution permissions and restrictions on instruction fetch.
- When writing new instructions to memory, there is no requirement for an `SB` instruction to prevent speculative execution of the old code. See Instruction cache maintenance instructions.
- Write speculation that is visible to other observers is prohibited for all memory types.
- A load will not speculatively read data from any store that appears in program order after the load itself.
- From Armv9.4, an instruction (I_1) that is not a permitted-speculatively-executed-instruction must not speculatively perform any function, other than being fetched, decoded or performing any side effects resulting from being fetched or decoded, that alters the microarchitectural state of the system in such a way that some other architecturally executed code (code2) can determine any architectural state, except for any architectural state accessed by I_1 as part of I_1 being fetched and decoded, that could not be determined had I_1 not speculatively executed.

When an architectural or speculative Explicit Memory Write Effect (E_1) to a Location (A_1) is properly synchronized such that a subsequent architectural or speculative instruction fetch (I_1) from A_1 is required to observe E_1, the permitted-speculative-next-PC generated from I_1 is permitted to be generated from the opcode before or after the modification of A_1 by E_1.

Note

The prohibition of using data loaded under speculation with faults to form addresses, condition codes or SVE predicate values does not prohibit the use of value predicted data from such locations for such purposes, so long as the training of the data value prediction was from the hardware defined context that is using the prediction. A consequence of this is that training of value prediction cannot be based on data loaded under speculation with a translation or Permission fault.

B2.5.1 AArch64 Speculative Store Bypass Safe

When FEAT_SSBS is implemented, PSTATE.SSBS is a control that can be set by software to indicate whether hardware is permitted to use, in a manner that is potentially speculatively exploitable, a speculative value in a register that has been loaded from memory using a load instruction that speculatively read the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

When the value of PSTATE.SSBS is 0, hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Setting PSTATE.SSBS to 0 is deprecated for performance reasons.

When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.
- Before the introduction of FEAT_SSBS, the control bit SPSR_ELx.SSBS was RES0, and therefore software written without awareness of FEAT_SSBS is expected to program it to 0. This means that PSTATE.SSBS will not be set, so the PE will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

B2.5.2 Definition of exploitative control of speculative execution

The execution of some code (code1) can exploitatively control speculative execution of some other code (code2) if all of the following apply:

- The actions of code1 can influence, in a manner that is not hard-to-determine, the prediction of multi-bit values that determine speculative execution of code2 to cause an irreversible change to the microarchitectural state of the PE that is indicative of some architectural state accessible to the execution context of code2.
- code1 has control in determining the choice of the architectural state that the irreversible change to the microarchitectural state is indicative of.
- The irreversible changes to the microarchitectural state of the PE can be measured by code executing in an execution context other than that of code2 to allow the retrieval of the architectural state in a manner that is not hard-to-determine.

Note

Eviction of entries from one context by another context is not considered exploitative control as long as eviction does not result in other forms of prediction that could be exploitatively controlled.

B2.5.3 Definition of predictive leakage to speculative execution

The execution of some code (code1) can predictively leak to some other code (code2) if all of the following apply:

- The execution of code1 influences, in a manner that is not hard-to-determine, the predictive microarchitectural structures of the implementation that predict multi-bit values, not binary choices, to behave in a way that is indicative of some architectural state accessible to the execution context of code1.
- The predictive microarchitectural structures of the implementation impact the timing of the speculative execution of code2 in a way that enables code2 to recover the architectural state in a manner that is not hard-to-determine.
- Code1 and code2 are not collaborating to communicate using the mechanisms in the previous two bullets.

Note

Mechanisms to prevent the influence and the state recovery being “not hard-to-determine” are left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.
Eviction of entries from predictive microarchitectural structures associated with one context (code2) caused by another context (code1) is considered predictive leakage if it enables code2 to recover some architectural state from code1.

B2.5.4 Restrictions on the effects of speculation from Armv8.5

Note

If SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated by software for:

- Secure EL0 for all ASID and VMID values.
- Secure EL1 for all VMID values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Further restrictions on speculation are introduced by some additional architectural features as described here.

FEAT_CSV3 introduces these restrictions:

- An attempt to access data under speculation, where the data is inaccessible given the current state of the PE cannot be used to alter the architectural or microarchitectural state in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed through means such as measuring the timing of code. This restriction applies to any of the following cases which make the data inaccessible:
  - reading from a Location with a Permission or Domain fault.
  - directly reading from a system register that is not architecturally readable.
  - reading from a SIMD&FP register, SVE register, or SME register that is not accessible due to traps configured by a more privileged Exception level.
  - indirectly reading from a system register containing a Pointer Authentication Key for which the corresponding Pointer Authentication instructions are disabled.

  Note

  The current state of the PE is defined as the EL, values of all special-purpose registers, and values of all system registers under which the hardware attempts to access the data, which may be on the speculative execution path. If on the speculative execution path, the hardware must respect any changes to the PE state that occur on the speculative path, respecting the same rules that would be required if executing on the non-speculative path. Note that the sequence of instructions on the speculative path may not be the same sequence of instructions that would occur on the non-speculative path as the hardware control flow predictions could cause the two sequences to differ.

  Note

  The following is a list of examples in which use of the inaccessible data could lead to recovery of the value of the inaccessible data:

  - to form an address, generate condition codes, or generate SVE predicate values to be used in the speculative sequence.
  - to form the register value used for the comparison of a compare-and-branch, test-and-branch, or compare-and-swap instruction in the speculative sequence. Note that this is to cover conditional instructions that do not use condition codes to determine their outcomes.
  - by prediction mechanisms such as Cache Prefetch or Data Value predictions.
  - as an input to an instruction in the speculative sequence where the execution timing of the instruction is dependent on the data value.

  Note

  It is permissible for a value of zero to be produced by the load or register read and consumed by an instruction newer than the load in the speculative sequence even if the value of zero results in different execution timing compared to non-zero values.

  Note

  As the effects of speculation are not architecturally visible, this restriction requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- The value of Allocation tags loaded under speculation as the result of a Tag checked access with a NoTagAccess fault or to Untagged memory cannot influence the execution of the instruction causing the Tag checked access or other instructions in the speculative sequence.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the microarchitectural state.

  Note

  Changes to Special-purpose registers can occur speculatively.

FEAT_CSV2, FEAT_CSV2_1p1, FEAT_CSV2_1p2, FEAT_CSV2_2 and FEAT_CSV2_3 introduce a range of additional restrictions.

If FEAT_CSV2 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2), as a result of the behavior of any of the following resources:
  - Branch target prediction based on the branch targets used in context1.
    - This applies to both direct and indirect branches, including return instructions, but excludes the prediction of the direction of a conditional branch.
  - Branch type prediction based on the branch types used in context1, when the hardware predicts a branch of a given type to be a different type of branch, and the type of the branch is used to exploitatively control target prediction of indirect branches within context2. *Branch type* is used to distinguish between:
    - Not a branch, direct branch, indirect branch, and branch and link (or call).
  - Data Value predictions based on data value from execution in context1.

    Note

    `PSTATE.{N, Z, C, V}` values from context1 are not considered a data value for this purpose.

  - Virtual address-based cache prefetch predictions generated as a result of execution in context1, based on, or causing dereference of, data values from memory.
  - Address predictions based on addresses used in context1.
    - This includes any form of memory disambiguation prediction or data forwarding based on partial address match.
  - Any other prediction mechanisms, other than Branch, Data Value, Cache Prefetch predictions, or Address predictions.

  In this definition, the hardware-defined context is determined by:
  - The Exception level.
  - The Security state.
  - When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
  - When executing at EL0 and using the EL1&0 translation regime, the *address space identifier* (ASID) and, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL2&0 translation regime, the ASID.

  If FEAT_CSV2_2 is implemented, then SCXTNUM_ELx is also part of the hardware-defined context.

  For unprivileged accesses generated by unprivileged memory access instructions, the hardware-defined context associated with the data addresses and data values of the unprivileged accesses is determined as if the access was generated by an instruction executed at EL0.

If either FEAT_CSV2_1p1 or FEAT_CSV2_3 is implemented, code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2) as a result of the behavior of any prediction mechanism based on the branch history used in context1.

If FEAT_CSV2_1p1 is implemented, branch or data values trained from one instruction address cannot exploitatively control, or predictively leak to, the speculative execution of code from a different address.

If FEAT_CSV2_1p2 is implemented, the SCXTNUM_ELx register is implemented, but is not part of the hardware-defined context.

B2.5.5 Restrictions on the effects of speculation from Armv9.5

If FEAT_BTI is implemented:

- Branches predicted or executed under speculation into a Guarded Page that land on instructions other than the intended BTI instructions are also guarded against speculative execution of those instructions.

B2.5.6 Branch history

If FEAT_CLRBHB is not implemented, then the architecture does not define any branch history maintenance instructions for AArch64 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_CLRBHB is implemented, the `CLRBHB` instruction is available. When the `CLRBHB` instruction is executed, the branch history is cleared for the current context to the extent that branch history information created before the `CLRBHB` instruction cannot be used by code before the `CLRBHB` instruction to exploitatively control the execution of any code in the current context appearing in program order after the instruction.

When FEAT_ECBHB is implemented, the branch history information created in a context before an exception entry to a higher Exception level using AArch64 cannot be used by code before that exception entry to exploitatively control the execution of any code in a different context after the exception entry.

Note

FEAT_CLRBHB and FEAT_ECBHB restrict the use of the branch history to prevent exploitative control. These restrictions apply only to prediction resources that are protected by FEAT_CSV2.

B2.5.7 Execution, data prediction and prefetching restriction System instructions

When FEAT_SPECRES is implemented alone or alongside FEAT_SPECRES2, the System instructions listed in A64 System instructions for prediction restriction prevent predictions based on information gathered from earlier execution within a particular execution context (CTX), from affecting the later speculative execution within that CTX, to the extent that the speculative execution is observable through side-channels.

The prediction restriction System instructions being used by a particular CTX apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

For these System instructions, the CTX is defined by:

- The Security state.
- The Exception level.
- When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
- When executing at EL0 when using the EL1&0 translation regime, the ASID and, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0 when using the EL2&0 translation regime, the ASID.

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 only applies to the current hardware defined context.
- Execution of the System instruction at EL1only applies to the current VMID and Security state, and does not apply to EL2 or EL3.
- Execution of the System instruction at EL2 can only apply to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to a combination of Security state and Exception level that is not implemented, or an Exception level which is higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a `NOP`.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.
- Prediction restriction System instructions are permitted to invalidate more prediction information than is defined by the supplied execution context.

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch64 state, EL0 access to the System instructions is controlled by:

- When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, SCTLR_EL1.EnRCTX.
- When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, SCTLR_EL2.EnRCTX.

Note

If SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated for:

- Secure EL0 for all ASID and VMID values.
- Secure EL1 for all VMID values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.6 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the Arm architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The following subsections describe the Arm memory barrier instructions:

- Instruction Synchronization Barrier.
- Data Memory Barrier.
- Speculation Barrier.
- Consumption of Speculative Data Barrier.
- Speculative Store Bypass Barrier.
- Profiling Synchronization Barrier.
- Physical Speculative Store Bypass Barrier.
- Trace Synchronization Barrier.
- Data Synchronization Barrier.
- Load-Acquire, Load-AcquirePC, and Store-Release.
- LoadLOAcquire, StoreLORelease.
- Guarded Control Stack Barrier (GCSB).

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

`DMB` and `DSB` instructions affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

B2.6.1 Instruction Synchronization Barrier

An `ISB` instruction ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed. Using an `ISB` ensures that the effects of context-changing operations executed before the `ISB` are visible to the instructions fetched after the `ISB` instruction. Examples of context-changing operations that require the insertion of an `ISB` instruction to ensure the effects of the operation are visible to instructions fetched after the `ISB` instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the `ISB` instruction take effect only after the `ISB` has been executed.

The pseudocode function for the operation of an `ISB` is `InstructionSynchronizationBarrier()`.

See also Memory barriers.

B2.6.2 Data Memory Barrier

The `DMB` instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The `DMB` instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The full definition of the DMB instruction is covered formally in the Ordering requirements defined by the formal concurrency model and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been observed by the PE before the DMB instruction is executed, are observed by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are observed by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The DMB instruction affects only memory accesses and the operation of data cache and unified cache maintenance instructions, see A64 Cache maintenance instructions. It has no effect on the ordering of any other instructions executing on the PE.

The pseudocode function for the operation of a DMB instruction is `DataMemoryBarrier()`.

B2.6.3 Speculation Barrier

An SB instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation. An example is speculative allocation into any caching structure where the allocation of that entry could indicate any data value present in memory or in the registers.

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

The speculative execution of an SB instruction cannot be as a result of any of the following:

- Control flow speculation.
- Data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when all of the following apply:

- It is known that it is not speculative, or it is speculative only as a result of either:
  - Speculating that an instruction that could generate an exception does not generate an exception.
  - Speculating past the point in the Execution stream where a precise asynchronous exception is taken.
- All the data values generated by instructions appearing in program order before the SB instruction are architecturally resolved, and so are not speculative.

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after the SB instruction.

B2.6.4 Consumption of Speculative Data Barrier

The CSDB instruction is a memory barrier instruction that controls speculative execution arising from data value prediction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Any instruction, other than a branch instruction, that appears in program order after the CSDB cannot be speculatively executed using the results of any of the following predictions if those predictions come from instructions that appear in program order before the CSDB and have not been architecturally resolved:

- Data value predictions of any instructions.
- `PSTATE.{N, Z, C, V}` predictions of any instructions other than conditional branch instructions appearing in program order before the `CSDB` that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

Note

For purposes of the definition of CSDB, `PSTATE.{N, Z, C, V}` and SVE predication state are not considered data values. This definition permits:

- Control flow speculation before and after the `CSDB` instruction.
- Speculative execution of conditional data processing instructions after the `CSDB` instruction, unless they use the results of data value, SVE predication state, or `PSTATE.{N, Z, C, V}` predictions of instructions appearing in program order before the `CSDB` instruction that have not been architecturally resolved.

B2.6.5 Speculative Store Bypass Barrier

The `SSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `SSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the `SSBB` instruction.

B2.6.6 Profiling Synchronization Barrier

The `PSB` instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a `NOP`.

B2.6.7 Physical Speculative Store Bypass Barrier

The `PSSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `PSSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the `PSSBB` instruction.

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.6.8 Trace Synchronization Barrier

The TSB instruction is a barrier instruction that preserves the relative order of accesses to System registers due to trace operations and other accesses to the same registers.

A trace operation is an operation of the trace unit generating trace for an instruction when FEAT_TRF is implemented and enabled.

A TSB instruction is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that TSB instruction is executed in the necessary order.

If trace is generated between a Context synchronization event and a TSB operation, these trace operations may be reordered with respect to the TSB operation, and therefore may not be synchronized.

The following situations are synchronized using a TSB operation:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a Context synchronization event C.
  - C appears in program order before a TSB operation T.
  - B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a Context synchronization event C1.
  - C1 appears in program order before TSB operation T.
  - T is executed in program order before a second Context synchronization event C2.
  - B is executed in program order after C2.

A TSB operation is not needed to ensure a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a Context synchronization event C.
- B is executed in program order before C.

If FEAT_TRBE is implemented, the requirements in this section are extended.

See Synchronization and the Trace Buffer Unit.

The pseudocode function for the operation of a TSB instruction is `TraceSynchronizationBarrier()`.

B2.6.9 Data Synchronization Barrier

A DSB instruction is a memory barrier that ensures that memory accesses that occur before the DSB instruction have completed before the completion of the DSB instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a DSB instruction:

- At EL2 ensures that any memory accesses caused by Speculative translation table walks from the EL1&0 translation regime have been observed.
- At EL3 ensures that any memory accesses caused by speculative translation table walks from the EL2, EL1&0 or EL2&0 translation regimes have been observed.

For more information, see Out-of-context translation regimes.

A DSB instruction executed by a PE completes when all of the following are complete for the set of observers in the required shareability domain:

- If the required access types of the DSB is reads, all of the following:
  - All Explicit Memory Read Effects generated by instructions in program order before the DSB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If FEAT_GCS is implemented, all GCS Memory Read Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.
  - If FEAT_MTE2 is implemented, all Implicit Tag Memory Read Effects generated by instructions in program order before the DSB.
- If the required access types of the DSB is writes, all of the following:
  - All Explicit Memory Write Effects generated by instructions in program order before the DSB.
  - If FEAT_GCS is implemented, all GCS Memory Write Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.
- If the required access types of the DSB is reads and writes, all of the following:
  - All Explicit Memory Effects generated by instructions in program order before the DSB.
  - All Implicit TTD Memory Effects generated by instructions in program order before the DSB.
  - All Implicit Instruction Memory Read Effects generated by instructions in program order before the DSB.
  - All cache maintenance instructions in program order before the DSB.
  - If FEAT_MTE2 is implemented, all Implicit Tag Memory Read Effects generated by instructions in program order before the DSB.
  - If FEAT_SPMU is implemented, all Direct System Register Write Effects to a System PMU register generated by instructions in program order before the DSB.
  - If FEAT_SPE is implemented, all PSB CSYNC instructions in program order before the DSB.
  - If FEAT_XS is implemented and the DSB has the nXS qualifier, all of the following:
    - All TLBInXS maintenance operations generated by AArch64 TLB maintenance instructions with the nXS qualifier in program order before the DSB.
    - If HCRX_EL2.FnXS is 1, all TLBInXS maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 in program order before the DSB.
  - If FEAT_XS is not implemented or the DSB does not have the nXS qualifier, all of the following:
    - All TLBI maintenance operations generated by TLB maintenance instructions in program order before the DSB.
  - If FEAT_HDBSS is implemented and the DSB is executed at EL2 or EL3, all HDBSS Memory Write Effects generated by instructions in program order before the DSB.
  - If FEAT_TRBE is implemented, all TSB CSYNC instructions in program order before the DSB.
  - All CFP RCTX, COSP RCTX, CPP RCTX, DVP RCTX instructions in program order before the DSB.
  - If FEAT_MTE_ASYNC is implemented, all Indirect System Register Write Effects to the Tag Fault Status Register accessible at ELx generated by instructions in program order before the DSB.
  - If FEAT_GCS is implemented, all GCS Memory Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes other than:

- Being fetched from memory and decoded, or any change to architectural or microarchitectural state resulting from being fetched or decoded.
- Indirectly or directly reading any of the following, so long as the reading of these resources does not cause a change in architectural state:
  - General-purpose registers
  - SIMD&FP registers
  - The Stack Pointer register
  - The Program Counter
  - If FEAT_SVE or FEAT_SME is implemented, SVE scalable vector registers or SVE predicate registers
  - If FEAT_SVE is implemented, the First Fault Register
  - If FEAT_SME is implemented, the ZA storage
  - If FEAT_SME2 is implemented, the ZT0 register
  - Special-purpose registers
  - System registers other than CNTPCTSS_EL0 and CNTVCTSS_EL0
- If FEAT_ETS2 is not implemented, translating any virtual addresses of instructions that generate Explicit Memory Effects.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_MTE_ASYNC is implemented, on completion of a `DSB` instruction with the LD qualifier, or neither the LD nor ST qualifier, all updates to TFSR_ELx.TFy or TFSRE0_EL1.TFy due to Tag Check Faults caused by accesses generated by instructions occurring in program order before the `DSB` will be complete. For more information on FEAT_MTE_ASYNC, see The Memory Tagging Extension.

When FEAT_XS is implemented and HCRX_EL2.FnXS is 1, an AArch64 `DSB` instruction executed at EL1 or EL0 behaves in the same way as the corresponding `DSB` instruction with the nXS qualifier executed at EL1 or EL0.

A `DSB` instruction ordered after a direct write to a System PMU register does not complete until all observers observe the direct write. A Context synchronization event is required to create the order between the direct write and the `DSB` instruction.

If FEAT_TRBE is implemented, the requirements in this section are extended. See Trace synchronization and memory barriers.

The pseudocode function for the operation of a `DSB` is `DataSynchronizationBarrier()`.

See also:

- Memory barriers.
- Ordering and completion of TLB maintenance instructions.

B2.6.9.1 Maintenance scope of the data synchronization barrier operations

A `DSB` instruction that applies to both reads and writes takes an argument that specifies its maintenance scope. The maintenance scope indicates the subset of TLB invalidate (TLBI) instructions, TLB invalidate pair (TLBIP) instructions, and Instruction cache maintenance (IC) instructions that are covered by this `DSB`, for one of:

- Outer Shareable domain.
- Inner Shareable domain.
- Non-shareable domain.

When a `DSB` instruction that applies to both reads and writes is used to ensure completion of TLB maintenance instructions or Instruction cache (IC) instructions, all of the following apply:

- A `DSB NSH` is sufficient to ensure completion of TLB maintenance instructions or IC instructions that apply to a single PE.
- A `DSB ISH` is sufficient to ensure completion of TLB maintenance instructions or IC instructions that apply to PEs in the same Inner Shareable domain.
- A `DSB OSH` is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Outer Shareable domain.

B2.6.10 Load-Acquire, Load-AcquirePC, and Store-Release

Arm provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the *Release Consistency sequentially consistent* (RCsc) model. In addition, FEAT_LRCPC provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release can be use to support the weaker *Release Consistency processor consistent* (RCpc) model.

The full definitions of the Load-Acquire and Load-AcquirePC instructions are covered formally in the Ordering requirements defined by the formal concurrency model. This introduction to the Load-Acquire and Load-AcquirePC instructions is not intended to contradict that section.

The basic principle of both Load-Acquire and Load-AcquirePC instructions is to introduce order between:

- The memory access generated by the Load-Acquire or Load-AcquirePC instruction.
- The memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction, such that the memory access generated by the Load-Acquire or Load-AcquirePC instruction is observed by each PE to the extent that the PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction are observed by that PE to the extent that the PE is required to observe the accesses coherently.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The use of a Load-Acquire or Load-AcquirePC instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the Ordering requirements defined by the formal concurrency model and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the following:

- A set of memory accesses, RWx, that are generated by the PE executing the Store-Release instruction and that appear in program order before the Store-Release instruction, together with those that originate from a different PE to the extent that the PE is required to observe them coherently, observed by the PE before executing the Store-release.
- The memory access generated by the Store-Release (Wrel), such that all of the memory accesses, RWx, are observed by each PE to the extent that the PE is required to observe those accesses coherently, before Wrel is observed by that PE to the extent that the PE is required to observe that access coherently.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Where a Load-Acquire appears in program order after a Store-Release, the memory access generated by the Store-Release instruction is observed by each PE to the extent that PE is required to observe the access coherently, before the memory access generated by the Load-Acquire instruction is observed by that PE, to the extent that the PE is required to observe the access coherently. In addition, the use of a Load-Acquire, Load-AcquirePC or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral arrival order.

Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit `DMB` instruction.

B2.6.11 LoadLOAcquire, StoreLORelease

For each PE, the Non-secure physical memory map is divided into a set of LORegions using a table that is held within the PE. Any PA in the Non-secure memory map can be a member of one LORegion. If a PA is assigned to more than one LORegion, then an implementation might treat it as if it has been assigned to fewer LORegions than that have been specified. If a PA is not in the Non-secure physical memory map, then that PA cannot be a member of any LORegion. For more information, see Limited ordering regions.

FEAT_LOR provides a set of instructions with Acquire semantics for loads, and Release semantics for stores that apply in relation to the defined LORegions. The new variants of the Load-Acquire and Store-Release instructions are LoadLOAcquire and StoreLORelease. See LoadLOAcquire/StoreLORelease.

For all memory types, these instructions have the following ordering requirements:

- LoadLOAcquire has the same semantics as Load-Acquire except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.
- StoreLORelease has the same semantics as Store-Release except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In addition, for accesses to Memory-mapped peripherals:

- LoadLOAcquire has the same semantics as Load-Acquire except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.
- StoreLORelease has the same semantics as Store-Release except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See Load-Acquire, Load-AcquirePC, and Store-Release.

Note

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit `DMB` instruction.

B2.6.12 Guarded Control Stack Barrier (GCSB)

The `GCSB` instruction is a barrier instruction that generates a GCSB event. The GCSB event applies ordering requirements between general load/store accesses and Guarded Control Stack data accesses. For more information, see Guarded Control Stack data access behaviors.

If FEAT_GCS is not implemented, this instruction executes as a `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.7 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The Arm architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- Introduction to caches.
- Memory hierarchy.
- Application level access to functionality related to caches
- Implication of caches for the application programmer.
- Prefetching into cache.

B2.7.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

Spatial locality

An access to one Location is likely to be followed by accesses to adjacent Locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The Arm architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B2.7.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B2-1 shows an example of such a system in an Armv8-A system that supports virtual addressing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

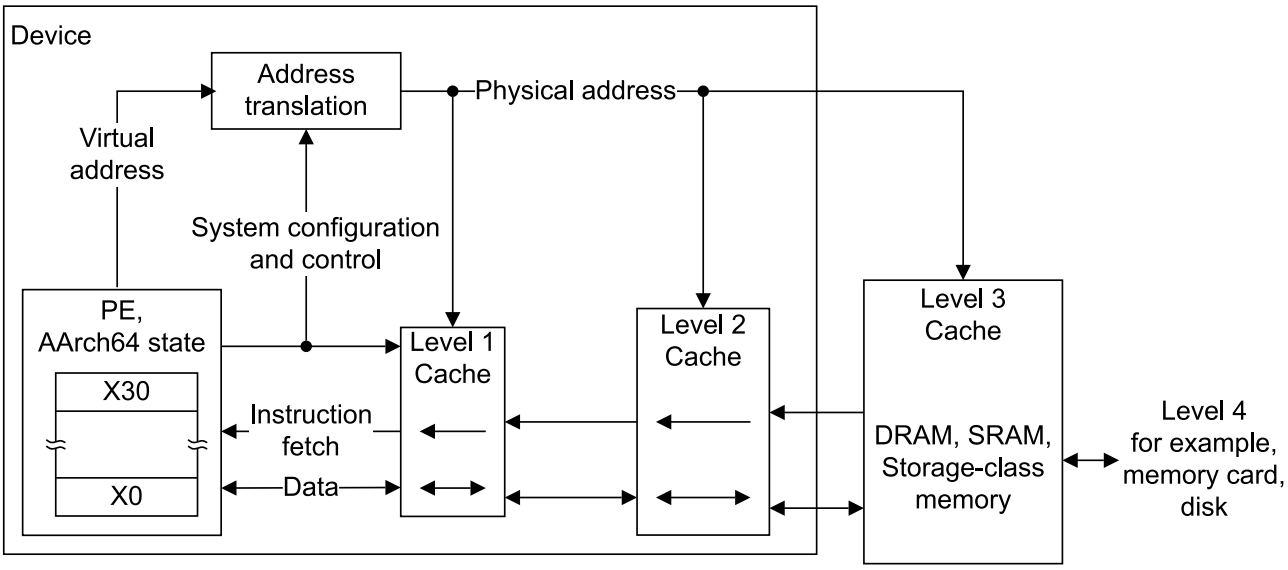

Figure B2-1 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU), Point of Coherency (PoC), Point of Persistence (PoP), and Point of Deep Persistence (PoDP).

For more information, including the definitions of PoU, PoC, PoP, and PoDP, see About cache maintenance in AArch64 state.

If FEAT_MTE2 is implemented, the behavior of cache maintenance instructions is modified. For more information, see Allocation Tags.

B2.7.2.1 The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about Cacheability and Shareability, see Memory types and attributes.

B2.7.3 Application level access to functionality related to caches

As indicated in About the Application level programmers' model, the application level corresponds to execution at EL0. The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable use of some of this functionality from EL0, as follows:

When the value of SCTLR_EL1.UCI is 1

Software executing at EL0 can access:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The data cache maintenance instructions, `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CVADP`, and `DC CIVAC`. See The data cache maintenance instruction (DC).
- The instruction cache maintenance instruction `IC IVAU`. See The instruction cache maintenance instruction (IC).

Attempted execution of these instructions might generate a Permission fault as described in Permission fault.

When the value of SCTLR_EL1.UCT is 1

Software executing at EL0 can access the cache type register. See CTR_EL0.

When the value of SCTLR_EL1.DZE is 1

Software executing at EL0 can access the data cache zero instruction `DC ZVA`. See Data cache zero instruction.

The SCTLR_EL1.{UCI, UCT, DZE} control fields are accessible only by software executing at EL1 or higher.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the controls {UCI, UCT and DZE} are found in SCTLR_EL2.

This functionality is UNDEFINED at EL0 when the value of the corresponding SCTLR_EL1 control field is 0.

B2.7.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

B2.7.4.1 Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using cache maintenance instructions to manage the coherency issues in software. See Application level access to functionality related to caches.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see Non-shareable Normal memory and Shareable, Inner Shareable, and Outer Shareable Normal memory.

Note

The performance of these hardware coherency mechanisms is highly IMPLEMENTATION SPECIFIC. In some implementations, the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

B2.7.4.2 Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last Context Synchronization event on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a Context Synchronization event, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture requires the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory. A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0} The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1} The write is complete for the shareability domain.

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When CTR_EL0.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the IC IVAU instructions of a different core.
  - When CTR_EL0.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

The full definition of the ordering implications of CTR_EL0.{DIC, IDC} is covered formally in the Ordering requirements defined by the formal concurrency model and the information in this section is not intended to contradict that section.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context Synchronization events and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.
- For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the Armv8 AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See Non-cacheable accesses and instruction caches.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions.
- The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.

B2.7.5 Prefetching into cache

The Arm architecture provides memory system hints `PRFM`, `PRFUM`, `RPRFM`, `LDNP`, and `STNP` that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Prefetch instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is taken using an SError exception. For more information, see ISS encoding for an exception from a Data Abort.

`PrefetchHint {}` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as prefetching the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PRFM`, `PRFUM`, `RPRFM`, and load/store instructions that provide hints to the memory system, see Prefetch memory and Load/store SIMD and floating-point non-temporal pair. For more information on `SVE PRF*` instructions that provide hints to the memory system, see Predicated non-contiguous element accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.8 Alignment support

This section describes alignment support. It contains the following subsections:

- Instruction alignment.
- Alignment of data accesses.

B2.8.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a PC alignment fault. See PC alignment checking.

B2.8.2 Alignment of data accesses

When instructions that load or store single or multiple registers are executed, if the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated if any of the following apply:

- For Load-Exclusive, Store-Exclusive, and Atomic instructions, the address that is accessed is not aligned to the overall access size.
- For all other instructions, the address that is accessed is not aligned to the size of the data element being accessed.

If FEAT_LSE2 is implemented, Load-Acquire/Store-Release instructions without exclusive or atomic behavior generate an Alignment fault if all of the following apply:

- Not all bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes.
- The value of SCTLR_ELx.nAA applicable to the current Exception level is 0.

For an unaligned access to any type of Device memory:

- If the memory location cannot support unaligned accesses then an Alignment fault is generated.
- If the memory location supports unaligned accesses, then it is IMPLEMENTATION DEFINED whether the access generates an Alignment fault if the location would not generate an Alignment fault if the same access were made to Normal memory.

B2.8.2.1 Unaligned accesses to Normal memory

The behavior of unaligned accesses to Normal memory is dependent on all of the following:

- The instruction causing the memory access.
- The memory attributes of the accessed memory.
- The value of SCTLR_ELx.{A, nAA}.
- Whether or not FEAT_LSE2 is implemented.

B2.8.2.1.1 Load or Store of Single or Multiple registers

For all instructions that load or store single or multiple registers, but not Load-Exclusive, Store-Exclusive, Load-Acquire/Store-Release, Atomic, and `SETG*` Memory Copy and Memory Set instructions, if the address that is accessed is not aligned to the size of the data element being accessed, then:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

- An unaligned access is performed.
- If FEAT_LSE2 is not implemented, the access is not guaranteed to be single-copy atomic except at the byte access level.
- If FEAT_LSE2 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, the memory access is single-copy atomic. For `LDNP`, `LDP`, or `STP` instructions, the entire memory access will be single-copy atomic.

— If all the bytes of the memory accessed do not lie within a 16-byte quantity aligned to 16 bytes, or the access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, the access is not guaranteed to be single-copy atomic except at the byte access level.

For these instructions, the definition of an unaligned access is based on the size of the accessed elements, not the overall size of the memory access. This affects SIMD and SVE element and structure loads and stores, and also load/store pair instructions.

For predicated SVE vector element and structure load or store instructions, alignment checks are based on the memory element access size, not on the vector element size.

For predicated SVE vector element and structure load or store instructions, *Inactive elements* cannot cause an Alignment fault.

For unpredicated SVE vector register load or store instructions, the base address is checked for 16-byte alignment.

For unpredicated SVE predicate register load or store instructions, the base address is checked for 2-byte alignment.

B2.8.2.1.2 Load-Exclusive/ Store-Exclusive and Atomic instructions

If FEAT_LSE2 is not implemented and the value of SCTLR_ELx.A applicable to the current Exception level is 0, Load-Exclusive/Store-Exclusive and Atomic instructions, including those with acquire or release semantics, generate an alignment fault when the address being accessed is not aligned to the overall size of the access.

If FEAT_LSE2 is implemented, then:

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, an unaligned access is performed.
- If all the bytes of the memory access do not lie within a 16-byte quantity aligned to 16-bytes, or the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  — An unaligned access is performed meeting all of the semantics of the instruction.
  — An Alignment fault is generated.

Where memory access is performed, then it is single-copy atomic.

For these instructions, the definition of an unaligned access is based on the overall access size.

If FEAT_LS64 is implemented, when a single-copy atomic 64-byte instruction accesses a memory location that is not aligned to 64 bytes, an Alignment fault always occurs, regardless of the value of SCTLR_ELx.A.

If FEAT_THE is implemented, Read-Check-Write instructions, `RCW`, and Read-Check-Write Software instructions, `RCWS`, generate an Alignment fault if the address being accessed is not aligned to the overall access size regardless of the value of SCTLR_ELx.A.

B2.8.2.1.3 Non-atomic Load-Acquire/Store-Release instructions

For Load-Acquire/Store-Release instructions that do not have exclusive or atomic behaviors, when the address being accessed is not aligned to the size of the data element being accessed and the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, then these instructions generate an Alignment fault.

If FEAT_LSE2 is implemented, then:

- If the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  — An unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.
  — An Alignment fault is generated.
- If all of the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes and SCTLR_ELx.nAA applicable to the current Exception level is 1, then an unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In this case, when an access is performed it meets all the semantics of the instruction, but the architecture does not define the order of the different transactions of the access defined by the single instructions relative to each other.

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, an unaligned access meeting all the semantics of the instruction is performed.

Note

- Unaligned accesses typically take additional cycles to complete compared to a naturally-aligned access.
- An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary.

B2.8.2.1.4 Memory Copy and Memory Set instructions

For `SETG*` instructions:

- There is an alignment check regardless of the value of SCTLR_ELx.A.
- If Xn is not a multiple of 16, an Alignment fault is generated.
- If Xd is not aligned to a multiple of 16, an Alignment fault is generated.

For more information, see the individual `SETG*` instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.9 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- Instruction endianness.
- Data endianness.
- Endianness of memory-mapped peripherals.

B2.9.1 General description of endianness in the Arm architecture

This section describes only memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The quadword at address A.
- The doubleword at address A and A+8.
- The words at addresses A, A+4, A+8, and A+12.
- The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B2-2 have the following definitions:

B_A Byte at address A.

HW_A Halfword at address A.

MSByte Most significant byte.

LSByte Least significant byte.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

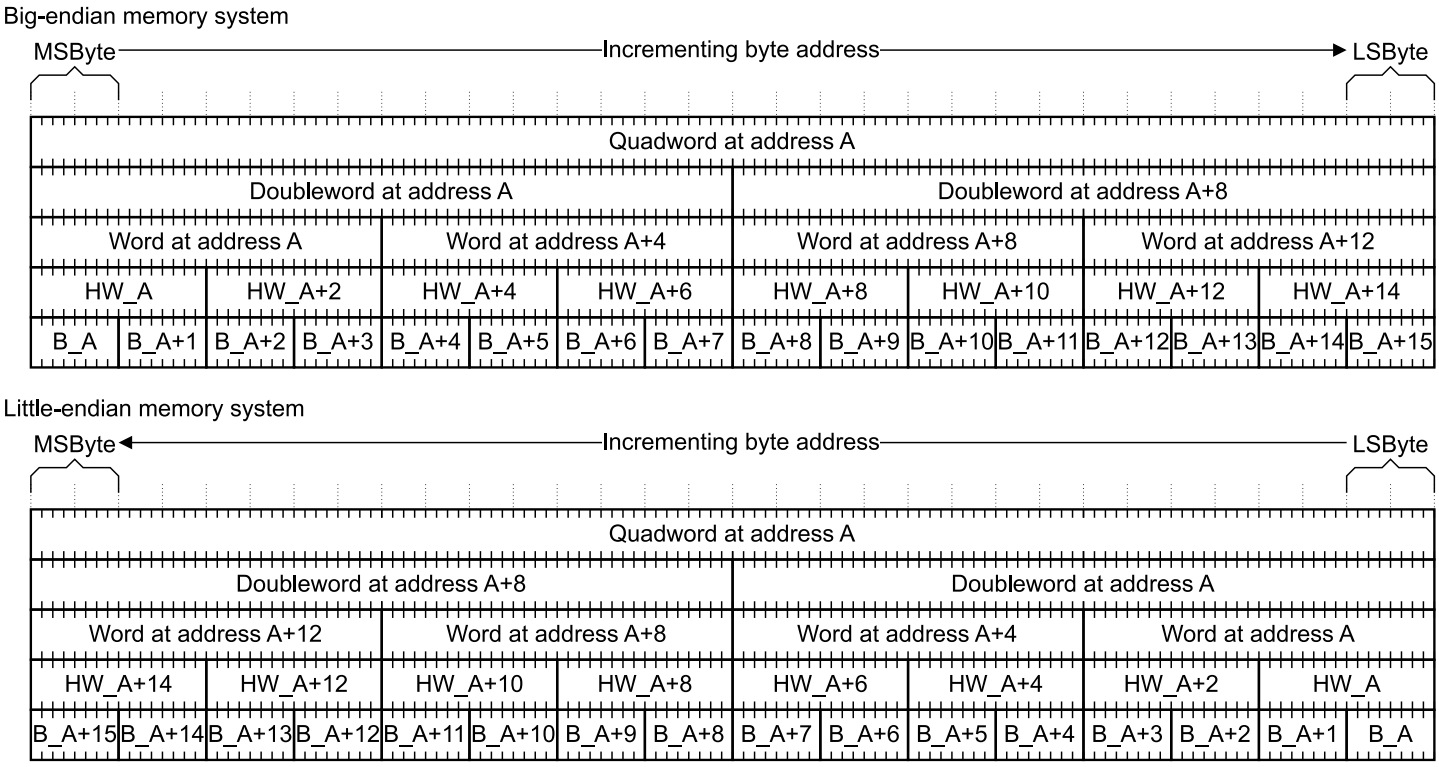

Figure B2-2 Endianness relationships

The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word, or halfword are interpreted. For example, a load of a word from address `0x1000` always results in an access to the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these 4 bytes.

B2.9.2 Instruction endianness

A64 instructions have a fixed length of 32 bits and are always little-endian.

B2.9.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the control is from SCTLR_EL2.E0E.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores. For more information, see Endianness in SIMD operations.

Note

This means the Armv8 architecture introduces a requirement for 128-bit endian conversions.

B2.9.3.1 Instructions to reverse bytes in a general-purpose register, a SIMD and floating-point register, or an SVE scalable vector register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table B2-1 shows the instructions that provide this functionality:

Table B2-1 Byte reversal instructions

Function	Instructions	Notes
Reverse bytes in 32-bit word or words[a]	REV32	For use with general-purpose registers
Reverse bytes in whole register	REV	For use with general-purpose registers
Reverse bytes in 16-bit halfwords	REV16	For use with general-purpose registers
Reverse elements in doublewords, vector	REV64	For use with SIMD and floating-point registers
Reverse elements in words, vector	REV32	For use with SIMD and floating-point registers
Reverse elements in halfwords, vector	REV16	For use with SIMD and floating-point registers
Reverse bytes/halfwords/words within elements, predicated	REVB, REVH, REVW	For use with SVE scalable vector registers

a. Can operate on multiple words.

B2.9.3.2 Endianness in SIMD operations

SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

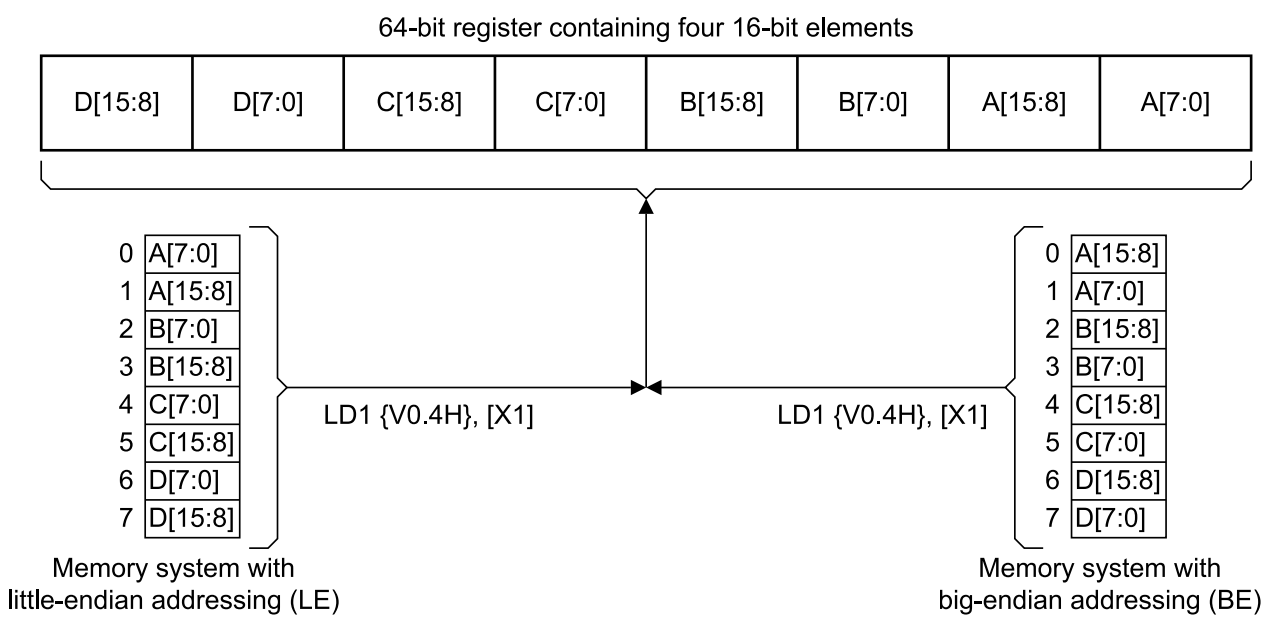

Figure B2-3 SIMD byte order example

The `BigEndian()` pseudocode function determines the current endianness of the data.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in A-profile Architecture Pseudocode.

B2.9.3.3 Endianness in SVE operations

R_{VDGQK} Rules on byte and element order of SIMD load and store instructions apply to SVE load and store instructions.

I_{RFQJP} Additional rules apply to the data endianness of memory accesses performed by SVE load and store instructions.

R_{CNKCL} For predicated SVE vector element and structure load and store instructions, an endianness conversion is performed using the memory element access size. The size of the vector element is not used in endianness conversion.

R_{QHXPL} For unpredicated SVE vector register load and store instructions, the vector byte elements are transferred in increasing element number order without any endianness conversion.

R_{RWLXY} For unpredicated SVE predicate register load and store instructions, each 8 bits from the predicate are transferred as a byte in increasing element number order without any endianness conversion.

R_{YGSBQ} When an SVE load instruction is executed, endianness conversion occurs before any sign-extension or zero-extension into a vector element.

R_{KYRQW} When an SVE store instruction is executed, endianness conversion occurs after any truncation from the vector element to the memory element access size.

B2.9.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a Cross Trigger Interface, see About the External Debug Registers.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see System Level Implementation of the Generic Timer.
- A memory-mapped register interface to the Performance Monitors, see Recommended External Interface to the Performance Monitors.
- A memory-mapped register interface to the Activity Monitors, see Recommended External Interface to the Activity Monitors.
- A memory-mapped register interface to the Memory System Resource Partitioning and Monitoring architecture (MPAM), see the *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification (ARM IHI 0099).*
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.*

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.10 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- Normal memory.
- Device memory.
- Memory access restrictions.

B2.10.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform Speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- An Explicit Memory Write Effect to a memory location with the Normal attribute completes in finite time.
- Explicit Memory Write Effects to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer Cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes. For the purposes of this requirement, the endpoint for a location in Conventional memory is the PoC.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Normal memory.
- Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction.

Note

Write speculation that is visible to other observers is prohibited for all memory types.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.

The following sections describe the other attributes for Normal memory:

- Shareable Normal memory.
- Non-shareable Normal memory.
- Cacheability attributes for Normal memory.

See also:

- Multi-register loads and stores that access Normal memory.
- Atomicity in the Arm architecture.
- Memory barriers.
- Concurrent modification and execution of instructions.

B2.10.1.1 Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

- Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
- Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see Synchronization and coherency issues between data and instruction accesses.

Note

- System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

B2.10.1.1.1 Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is a member of only a single Inner Shareability domain.
- Each observer is a member of only a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The details of the use of the shareability attributes are system-specific. Example B2-1 shows how they might be used.

Example B2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

B2.10.1.2 Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

B2.10.1.3 Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see Cacheability, cache allocation hints, and cache transient hints.

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint, this does not affect the behavior of accesses to that location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For other cases, see Mismatched memory attributes.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner Cacheability attributes can lie outside a cache that is controlled by the Outer Cacheability attributes.
- An implementation might not have any outer cache.

Example B2-2, Example B2-3, and Example B2-4 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see Memory hierarchy.
- When managing coherency, system designs must consider both the Inner and Outer Cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example B2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner Cacheability attribute applied to L1 and L2 cache.
- The Outer Cacheability attribute applied to L3 cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example B2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner Cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer Cacheability attribute.

Example B2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner Cacheability attribute applied to L1 cache.
- The Outer Cacheability attribute applied to L2 and L3 cache.

B2.10.1.4 Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more SVE or Advanced SIMD&FP registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B2.10.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering Identified as G or nG, see Gathering.

Reordering Identified as R or nR, see Reordering.

Early Write Acknowledgement

Identified as E or nE, see Early Write Acknowledgement.

The Armv8 Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early Write Acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that Speculative accesses to Device-GRE memory is forbidden.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  The following exceptions to this apply:

  — Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — For reads, including hardware speculation, that are performed by an SVE unpredicated load instruction, all of the following are true:

    — For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction, any byte in the window can be accessed by the instruction.

    — All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction.

  — For reads, including hardware speculation, that are performed by an SVE predicated load instruction that is not a non-temporal load, all of the following are true:

    — For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.

    — All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.

  — For reads, including hardware speculation, that are performed by an SVE predicated non-temporal load instruction from memory locations with the Gathering attributes, all of the following are true:

    — For any 128-byte window aligned to 128 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.

    — All bytes accessed by the instruction are in a 128-byte window aligned to 128 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.

  — The architecture permits a Memory Copy and Memory Set `CPY*` instruction to perform speculative reads of any memory location, even those marked as Device, within a 64-byte quantity, aligned to 64 bytes, of a location that is within the range [Xs] to [Xs+Xn-1].

  — For Device memory with the Gathering attribute, reads generated by the `LDNP` instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See Properties of single-copy atomic accesses.

  — An LDRAA or LDRAB instruction that fails the pointer authentication check and loads from a location in Device memory is permitted to cause one read access to that location if all of the other requirements for accessing that Device location are met.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

— An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception, the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

— Write speculation that is visible to other observers is prohibited for all memory types.

- An Explicit Memory Write Effect to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit Memory Write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Explicit Memory Write Effects to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two Explicit Memory Write Effects of Device memory type to the same location might be merged before they reach the endpoint, unless both Explicit Memory Write Effects have the non-Gathering attribute or there is an ordered-before relationship between the two Explicit Memory Write Effects.
- For accesses to any Device memory type, a `DMB` instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

Note

This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note

In the EL1&0 translation regime in systems where HCR_EL2.TGE == 1 and HCR_EL2.DC == 0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

See also Memory access restrictions.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

Note

In general, making a translation table walk to any Device memory type is the result of a programming error.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

B2.10.2.1 Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

Note

This also applies to writebacks from the cache, whether caused by a Natural eviction or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their single-copy atomic sizes, except that there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by multi-register load/store instructions that generate more than one single-copy atomic access. See Multi-register loads and stores that access Device memory.

Gathering between two accesses, where one access is barrier-ordered-before the other access, is not permitted.

Gathering between memory accesses generated by Memory Copy and Memory Set instructions is always permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.10.2.2 Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see Ordering relations.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

Note

Some interconnect fabrics, such as PCIe, perform very limited reordering, which is not important for the software usage. It is outside the scope of the Arm architecture to prohibit the use of a non-Reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.
- The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an IMPLEMENTATION DEFINED size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in Ordering relations as part of the formal definition of the memory model. It does have an effect on the Peripheral arrival order described in Peripheral arrival order.
- If the same memory location is mapped with different aliases, and different attribute values, these are a type of mismatched attribute. The different attributes could be:
  - A different Reordering attribute value.
  - A different Device memory attribute value.
  - When FEAT_XS is implemented, a different XS attribute value.

For information about the effects of accessing memory with mismatched attributes, see Mismatched memory attributes.

An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

The non-Reordering attribute has no effect on the memory accesses caused by Memory Copy and Memory Set instructions.

B2.10.2.3 Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgment of a write comes from the endpoint, it is guaranteed that:
  - Only the endpoint of the write access returns a write acknowledgment of the access.
  - No earlier point in the memory system returns a write acknowledgment.
- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgment of a write comes from the endpoint, the acknowledgment of a write is not required to come from the endpoint.

Note

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgment of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgment is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
- The areas of the physical memory map for which write acknowledgment from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgment from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgment from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgment from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgment from the endpoint.

B2.10.2.4 Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register, the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all instructions that load or store one or more Advanced SIMD&FP or SVE registers, and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions, even for access to any type of Device memory.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address, even if the resulting accesses are to any type of Device memory.

B2.10.2.5 SVE loads and stores that access Device memory

R_{XLLJZ} All rules applying to Device memory accesses by Advanced SIMD and floating-point load and store instructions apply to Device memory access by SVE load and store instructions.

I_{YHWJT} Additional rules apply to Device memory access by SVE load and store instructions.

R_{NYMWH} When an SVE vector prefetch instruction is executed, any resulting Read Memory is guaranteed not to access Device memory.

R_{SBHLD} When an SVE Non-fault vector load is executed, or when any element from a First-fault load except the *First active element* attempts to access Device memory, the resulting Read Memory will not access Device memory.

R_{TMVNR} When an SVE Non-fault vector load instruction is executed, an attempt by any *Active element* to access Device memory is suppressed and reported in the FFR.

R_{SFBKQ} When an SVE First-fault vector load instruction is executed, any Read Memory performed for the *First active element* can access Device memory.

R_{BHNQN} When an SVE First-fault vector load instruction is executed, an attempt by any *Active element* other than the *First active element* to access Device memory is suppressed and is reported in the FFR.

R_{QBLMZ} Any access to Device memory performed by an SVE load or store instruction is relaxed such that it might behave as if:

- The memory has the Gathering attribute, regardless of the configured value of the nG attribute.
- The memory has the Reordering attribute, regardless of the configured value of the nR attribute.
- The memory has the Early Acknowledgment attribute, regardless of the configured value of the nE attribute.

Whether attributes are classified as mismatched is determined strictly by the memory attributes derived from the translation table entry.

B2.10.2.6 Streaming SVE mode loads and stores that access Device memory

R_{XHFBX} When the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, any access to Device memory performed by an Advanced SIMD&FP load/store instruction is relaxed such that it might behave as if:

- The Gathering attribute is 1, regardless of the configured value of the nG attribute.
- The Reordering attribute is 1, regardless of the configured value of the nR attribute.
- The Early Acknowledgement attribute is 1, regardless of the configured value of the nE attribute.

Whether attributes are classified as mismatched is determined strictly by the memory attributes derived from the translation table entry.

See also:

- Streaming SVE mode.

B2.10.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit Memory Read effects to any two adjacent bytes in memory, p and $p+1$, generated by the same instruction, and for two explicit Memory Write effects to any two adjacent bytes in memory, p and $p+1$, that are generated by the same instruction:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The bytes *p* and *p+1* must have the same memory type and Shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LD1`, `ST1`, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.

— Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes *p* and *p+1*.

For the permitted CONSTRAINED UNPREDICTABLE behavior, see Crossing a page boundary with different memory types or Shareability attributes.

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule, then behavior is CONSTRAINED UNPREDICTABLE, and Crossing a peripheral boundary with a Device access describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

Note

— The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.

— This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

— Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.11 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see The AArch64 Virtual Memory System Architecture.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.
- When FEAT_XS is implemented, XS attribute.

Collectively these are referred to as memory attributes.

If FEAT_MTE2 is implemented, accesses to a location which use a common definition of the memory attributes but the Tagged attribute of that location differs do not cause a mismatched access to occur.

If FEAT_MTE_PERM is implemented, the NoTagAccess attribute does not factor into the determination of mismatched attributes.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of Speculative read accesses.
- Prohibition on Gathering.
- Prohibition on reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.
   - Inner Non-cacheable, Outer Non-cacheable Normal memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   - Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a DMB barrier instruction between any accesses to the same cacheable Location that use different attributes.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see Ordering and completion of data and instruction cache instructions.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
   - Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
   - A DMB barrier is placed between any accesses to the same memory Location that use different attributes.

   Note

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

As described in Non-cacheable accesses and instruction caches, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.12 Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table B2-2 shows the synchronization primitives and the associated `CLREX` instruction.

Table B2-2 Synchronization primitives and associated instruction, A64 instruction set

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	`LDXRB`	`STXRB`	-
	Load-Acquire/Store-Release	`LDAXRB`	`STLXRB`	-
Halfword	-	`LDXRH`	`STXRH`	-
	Load-Acquire/Store-Release	`LDAXRH`	`STLXRH`	-
Register[b]	-	`LDXR`	`STXR`	-
	Load-Acquire/Store-Release	`LDAXR`	`STLXR`	-
Pair[b]	-	`LDXP`	`STXP`	-
	Load-Acquire/Store-Release	`LDAXP`	`STLXP`	-
None	Clear-Exclusive	-	-	`CLREX`

a. Instruction in the A64 instruction set.

b. A register instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register A pair instruction operates on two doublewords if access X registers, or on two words if accessing W registers.

Except for the row showing the `CLREX` instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the Write Memory succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see Marking and the size of the marked memory block. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- Exclusive access instructions and Non-shareable memory locations.
- Exclusive access instructions and Shareable memory locations.
- Marking and the size of the marked memory block.
- Context switch support.
- Load-Exclusive and Store-Exclusive instruction usage restrictions.
- Use of WFE and SEV instructions by spin-locks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2.12.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks the address of the Explicit Memory Read effect generated by any Load-Exclusive instruction executed by the PE.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Explicit Memory Write effect of the Store-Exclusive instruction is the same as the address that has been marked in the monitor due to the execution of an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place, the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state, the monitor is *set.*

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Open Access state, the monitor is *clear*.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure B2-4 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

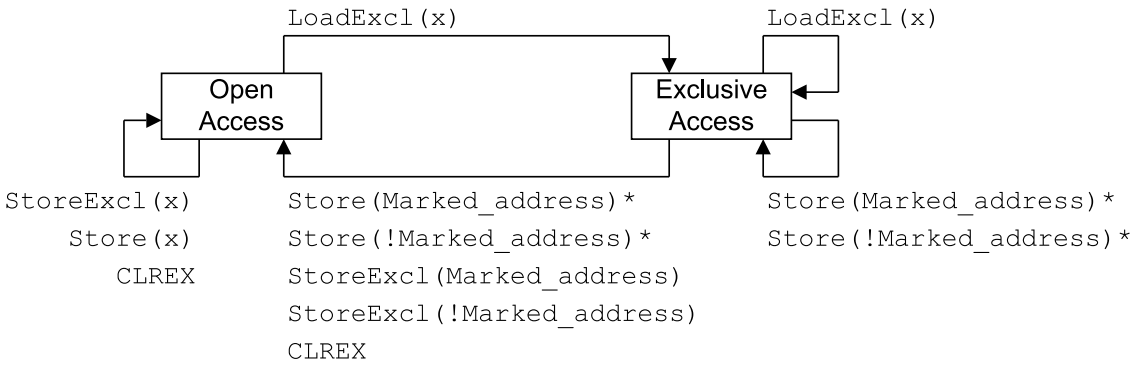

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address `x` used for the operation.

Figure B2-4 Local monitor state machine diagram

For more information about marking, see Marking and the size of the marked memory block.

Note

For the local monitor state machine, as shown in Figure B2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

B2.12.1.1 Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B2-4 must not indefinitely delay forward progress of execution.

B2.12.2 Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in Exclusive access instructions and Non-shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

memory locations, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:

— Updating memory.

— Returning a status value of 0.

The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:

  — Can hold at least one marked block.

  — Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, see Load-Exclusive and Store-Exclusive instruction usage restrictions.

Note

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

If FEAT_MTE2 is implemented, it is IMPLEMENTATION DEFINED whether explicit accesses to Allocation Tags are monitored by a global monitor. For more information, see The Memory Tagging Extension.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all Arm PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

The architecture only requires that Conventional memory mapped in this way supports this functionality.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the EL1&0 translation regime then:

  — If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.

  — Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by Arm PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

B2.12.2.1 Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor supports only a single outstanding exclusive access to shareable memory per PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:

  — A status value of 0 is returned to a register to acknowledge the successful store.

  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.

  — If the address accessed is marked for exclusive access in the global monitor state machine for any other PE, then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:

  — A status value of 1 is returned to a register to indicate that the store failed.

  — The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:

  — If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.

  — If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means that it responds to:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses generated by PE(n).
- Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

B2.12.2.1.1 Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see Mechanisms for entering a low-power state.

Figure B2-5 shows the state machine for PE(n) in a global monitor.

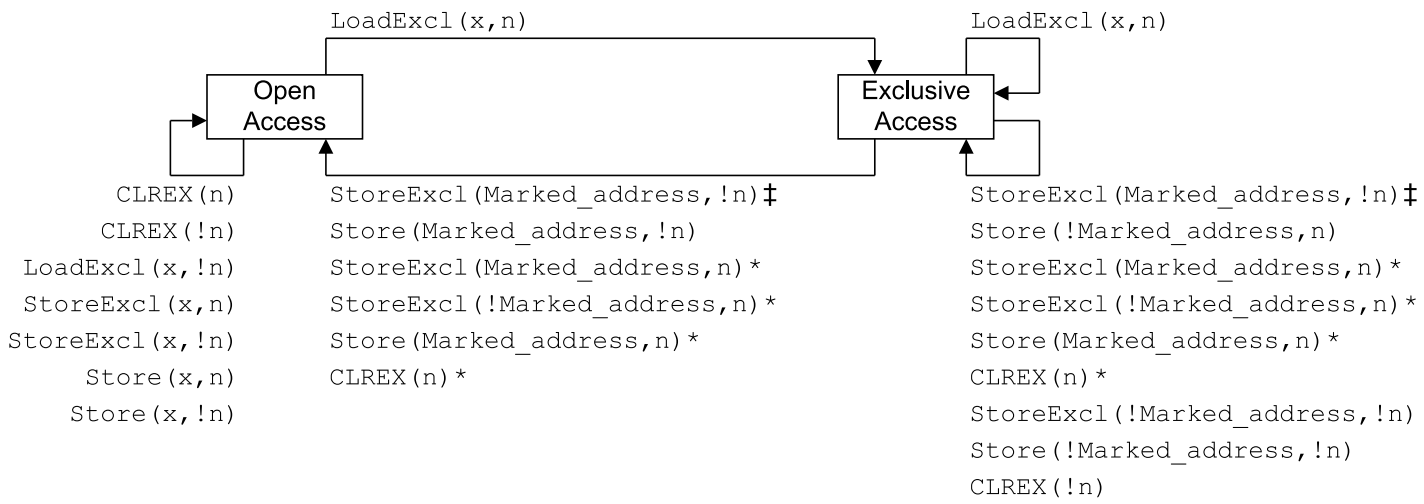

‡`StoreExcl`(Marked_address,!n) clears the monitor only if the `StoreExcl` updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see Marking and the size of the marked memory block.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For the global monitor state machine, as shown in Figure B2-5:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B2-5 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

B2.12.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4-512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where *a* is 4.
- 512 words in an implementation where *a* is 11.

For example, in an implementation where *a* is 4, a successful `LDXRB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B2.12.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

B2.12.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDXP`/`STXP` pair or a `LDXR`/`STXR` pair, executed as part of a loop that contains only a single Load-Exclusive and Store-Exclusive pair.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a `LoadExcl`/`StoreExcl` instruction pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B2-2. In this context, a `LoadExcl`/`StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses.

  - The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes.

  - The block of data of the size of the larger of the transaction sizes used by the `LoadExcl`/`StoreExcl` pair at the address accessed by the `LoadExcl`/`StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.
- An implementation of the `LoadExcl` and `StoreExcl` instructions can require that, in any thread of execution, the `StoreExcl` instruction accesses the same number of registers as the preceding `LoadExcl` instruction executed in that thread. If the `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they access the same number of registers. For more information, see CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the Tag Checked property of a memory access due to a `StoreExcl` instruction is the same as the Tag Checked property of a memory access by the preceding `LoadExcl` instruction, unless FEAT_MTE_STORE_ONLY is implemented and the preceding `LoadExcl` is not Tag Checked only due to Store-only Tag Checking being enabled. If the Tag Checked property of memory accesses due to a `LoadExcl`/`StoreExcl` pair in the same thread of execution is not compatible, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of such a `LoadExcl`/`StoreExcl` pair, and fail for other instances of such a `LoadExcl`/`StoreExcl` pair.

  - The data at the address accessed by the `LoadExcl`/`StoreExcl` pair is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the memory is accessed with a compatible Tag Checked property.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_LSE is not implemented, `LoadExcl`/`StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, the software meets all of the following conditions. If FEAT_LSE is implemented, software can maximize the likelihood that `LoadExcl`/`StoreExcl` loops make forward progress, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, by meeting all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, software prefetches, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, `ISB` barriers, indirect branches, or Branch with Link instructions.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:

  — There are no stores or `PRFM` or `RPRFM` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive. This includes VA aliases to those addresses.

  — There are no loads or prefetches to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.

  — There are no other Store-Exclusive instructions to any other address.

  — There are no direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, indirect branches, or Branch with Link instructions.

  — All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

  The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

  — Performing stores to a PA covered by the Exclusives monitor.

  — Prefetching with intent to write to a PA covered by the Exclusives monitor.

  — Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.

  — Executing instruction cache invalidate all instructions.

  — Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.

  — Executing TLB maintenance to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.

- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.

- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.

- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:

  — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

  Note

  This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.

  — The data at the address accessed by the `StoreExcl` is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in one Security state cannot cause a denial of service on a PE in another Security state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `StoreExcl` instruction, and the change is performed using a break-before-make sequence as described in Using break-before-make when updating translation table entries, if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

  Note

  Arm expects that, in many implementations, either:

  — The TLB invalidation will clear either the local or global monitor.

  — The PA will be checked between the `LoadExcl` and `StoreExcl`.

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PRFM PST*` or `RPRFM` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PRFM PST*` or `RPRFM` accesses to a memory location might impede the forward progress of another PE.
- If FEAT_MTE2 is implemented, and if a Tag Unchecked store exclusive instruction would not perform the store and return a status value of 1, it is CONSTRAINED UNPREDICTABLE whether:

  — The instruction is a Tag Checked access,

  — The instruction is an Tag Unchecked access.

  For more information, see The Memory Tagging Extension.

Note

In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

B2.12.5.1 CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers

As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED UNPREDICTABLE if, in a single thread of execution, either:

- An `LDXP` instruction of two 32-bit quantities is followed by an `STXR` instruction of one 64-bit quantity at the same address.
- An `LDXR` instruction of one 64-bit quantity is followed by an `STXP` instruction of two 32-bit quantities at the same address.

In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:

- The `STXP` or `STXR` instruction generates an external Data Abort.
- The `STXP` or `STXR` instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `0b110101`.
- The `STXP` or `STXR` instruction always fails, returning a status of 1.
- The `STXP` or `STXR` instruction always passes, returning a status of 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- This STXP or STXR instruction has the same pass or fail behavior that it would have had if the instruction had used the same size and number of registers as the preceding LDXR or LDXP instruction.

B2.12.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, WFE, SEV, and SEVL, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in Wait for Event. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an SEVL instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an SEV instruction after clearing a spinlock.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part C
The AArch64 Instruction Set

Chapter C1
The A64 Instruction Set

This chapter describes the A64 instruction set. It contains the following sections:

- About the A64 instruction set.
- Structure of the A64 assembler language.
- Address generation.
- Instruction aliases.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.1 About the A64 instruction set

The A64 instruction set is the instruction set supported in the AArch64 Execution state. All A64 instructions have a width of 32 bits.

The following chapters contain alphabetical lists of A64 instructions:

- A64 Base Instruction Descriptions lists all of:
  - Branch instructions, exception generating instructions, system instructions.
  - Load, store, and data-processing instructions associated with the general-purpose registers.
- A64 Advanced SIMD and Floating-point Instruction Descriptions lists the load, store, and data-processing instructions associated with Advanced SIMD and floating-point support.
- SVE Instruction Descriptions lists the load, store, and data-processing instructions associated with SVE support.
- SME Instruction Descriptions lists the load, store, and data-processing instructions associated with SME support.

When an instruction supports more than one syntax, each syntax is an *instruction variant*. Instruction variants can occur because of differences in:

- The size or format of the operands.
- The register file used for the operands.
- The addressing mode used for load/store memory operands.

Instruction variants might also arise as the result of other factors.

Instruction variants are described in the instruction description for the individual instructions.

A64 instruction set encoding describes the A64 encoding structure. A64 instructions have a regular bit encoding structure:

- 5-bit register operand fields at fixed positions within the instruction. For general-purpose register operands, the values 0-30 select one of 31 registers. The value 31 is used as a special case that can:
  - Indicate use of the current stack pointer, when identifying a load/store base register or in a limited set of data-processing instructions. See The stack pointer registers.
  - Indicate the value zero when used as a source register operand.
  - Indicate discarding the result when used as a destination register operand.
- For Advanced SIMD and floating-point register access and SVE scalable vector register access, a 5-bit register operand field selects one of 32 registers of the appropriate type.
- Certain Advanced SIMD, SVE, and SME instructions have a register operand field smaller than 5 bits when specifying an indexed vector element or multi-vector operand, which restricts the number of vector registers that can be accessed by those operands.
- For SME instructions that access ZA tiles, ZA tile slices, or ZA array vectors, see ZA storage.
- Immediate bits that provide constant data-processing values or address offsets are placed in contiguous bitfields. Some calculated values in instruction variants use one or more immediate bitfields together with the secondary encoding bitfields.

All encodings that are not fully defined are described as unallocated. An attempt to execute an unallocated instruction is UNDEFINED, unless the behavior is otherwise defined in this Manual.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2 Structure of the A64 assembler language

The following sections describe the A64 assembler syntax:

- General requirements.
- Common syntax terms.
- Instruction Mnemonics.
- Condition code.
- Register names.

C1.2.1 General requirements

The letter `W` denotes a general-purpose register holding a 32-bit word, and `X` denotes a general-purpose register holding a 64-bit doubleword.

An A64 assembler recognizes both uppercase and lowercase variants of the instruction mnemonics and register names, but not mixed case variants. An A64 disassembler can output either uppercase or lowercase mnemonics and register names. Program and data labels are case-sensitive.

The A64 assembly language does not require the # character to introduce constant immediate operands, but an assembler must allow immediate values introduced with or without the # character.

In Example C1-1, the sequence // is used as a comment leader and A64 assemblers are encouraged to accept this syntax.

C1.2.2 Common syntax terms

The following syntax terms are used frequently throughout the A64 instruction set description.

`UPPER` Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name `Xn` indicates that the `X` is required, followed by a variable register number, for example `X29`.

`< >` Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.

`{ }` Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.

`[ ]` Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.

`a|b` Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, `U(ADD|SUB)W` represents `UADDW` or `USUBW`.

`±` This indicates an optional + or - sign. If neither is used then + is assumed.

`uimmn` An *n*-bit unsigned, positive, immediate value.

`simmn` An *n*-bit two's complement, signed immediate value, where *n* includes the sign bit.

`SP` See Register names.

`Wn` See Register names.

`WSP` See Register names.

`WZR` See Register names.

`Xn` See Register names.

`XZR` See Register names

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.3 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following `ADD` instructions all have different opcodes. However, the programmer must remember only one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.

Example C1-1 ADD instructions with different opcodes

```
ADD W0, W1, W2          // add 32-bit register
ADD X0, X1, X2          // add 64-bit register
ADD X0, X1, W2, SXTW    // add 64-bit extended register
ADD X0, X1, #42         // add 64-bit immediate
```

C1.2.4 Condition code

The A64 ISA has some instructions that set Condition flags or test Condition codes or both. For information about instructions that set the Condition flags or use the condition mnemonics, see Condition flags and related instructions.

Table C1-1 shows the available Condition codes.

Table C1-1 Condition codes

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

a. Unordered means at least one NaN operand.

b. The Condition code `NV` exists only to provide a valid disassembly of the `0b1111` encoding, otherwise its behavior is identical to `AL`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.5 SVE Condition code aliases

The SVE and SME assembler syntax defines an alternative set of condition code aliases for use with AArch64 conditional instructions, as follows:

Table C1-2 shows the available SVE Condition code aliases.

Table C1-2 SVE Condition codes

cond	Mnemonic	SVE alias	Meaning	Condition flags
0000	EQ	NONE	All *Active elements* were FALSE or there were no *Active elements*.	Z == 1
0001	NE	ANY	An *Active element* was TRUE.	Z == 0
0010	CS or HS	NLAST	The *Last active element* was FALSE or there were no *Active elements*.	C == 1
0011	CC or LO	LAST	The *Last active element* was TRUE.	C == 0
0100	MI	FIRST	The *First active element* was TRUE.	N == 1
0101	PL	NFRST	The *First active element* was FALSE or there were no *Active elements*.	N == 0
0110	VS	-	CTERM comparison failed, but end of partition reached.	V == 1
0111	VC	-	CTERM comparison succeeded, or end of partition not reached.	V == 0
1000	HI		An *Active element* was TRUE, but the *Last active element* was FALSE.	C ==1 && Z == 0
1001	LS	PLAST	The *Last active element* was TRUE, or all *Active elements* were FALSE, or there were no *Active elements*.	C ==0 || Z ==1
1010	GE	TCONT	CTERM termination condition not detected.	N == V
1011	LT	TSTOP	CTERM termination condition detected.	N != V

C1.2.6 Register names

See:

- General-purpose register file and zero register and stack pointer.
- Advanced SIMD and floating-point register file.
- Advanced SIMD and floating-point scalar register names.
- SIMD vector register names.
- SIMD vector element names.
- For SVE register names, see Z0-Z31, P0-P15, and FFR, First Fault Register.
- For SME ZA storage, see ZA array vector access and ZA tile access.
- For SME2 ZT0 storage, see ZT0.

C1.2.6.1 General-purpose register file and zero register and stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. In a general-purpose register field the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-3 shows the qualified names for registers, where *n* is a register number 0-30.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C1-3 Naming of general-purpose registers, the zero register, and the stack pointer

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

This list gives more information about the instruction arguments shown in Table C1-3:

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the Arm 64-bit Architecture*.

C1.2.6.2 Advanced SIMD and floating-point register file

The 32 registers in the Advanced SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the Advanced SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See General-purpose register file and zero register and stack pointer.

Note

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials, or cryptographic hashes.

C1.2.6.3 Advanced SIMD and floating-point scalar register names

Advanced SIMD and floating-point instructions that operate on scalar data only access the lower bits of an Advanced SIMD and floating-point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Table C1-4 shows the qualified names for accessing scalar Advanced SIMD and floating-point registers. The letter *n* denotes a register number between 0 and 31.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C1-4 Advanced SIMD and floating-point scalar register names

Size	Name
8 bits	`Bn`
16 bits	`Hn`
32 bits	`Sn`
64 bits	`Dn`
128 bits	`Qn`

C1.2.6.4 SIMD vector register names

If a register holds multiple data elements on which arithmetic is performed in a parallel, SIMD manner, then a qualifier describes the vector shape. The vector shape is the element size and the number of elements or lanes. If the element size in bits multiplied by the number of lanes does not equal 128, then the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

Table C1-5 shows the SIMD vector register names. The letter *n* denotes a register number between 0 and 31.

Table C1-5 SIMD vector register names

Shape	Name
8 bits × 8 lanes	`Vn.8B`
8 bits × 16 lanes	`Vn.16B`
16 bits × 4 lanes	`Vn.4H`
16 bits × 8 lanes	`Vn.8H`
32 bits × 2 lanes	`Vn.2S`
32 bits × 4 lanes	`Vn.4S`
64 bits × 1 lane	`Vn.1D`
64 bits × 2 lanes	`Vn.2D`

C1.2.6.5 SIMD vector element names

Appending a constant, zero-based element index to the register name inside square brackets indicates that a single element from a SIMD and floating-point register is used as a scalar operand. The number of lanes is not represented, as it is not encoded in the instruction and can only be inferred from the index value.

Table C1-6 shows the vector register names and the element index. The letter *i* denotes the element index.

Table C1-6 Vector register names with element index

Size	Name
8 bits	`Vn.B[i]`
16 bits	`Vn.H[i]`
32 bits	`Vn.S[i]`
64 bits	`Vn.D[i]`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An assembler must accept a fully qualified SIMD register name if the number of lanes is greater than the index value. See SIMD vector register names. For example, an assembler must accept all of the following forms as the name for the 32-bit element in bits [63:32] of the SIMD and floating-point register V9:

```
V9.S[1]     //standard disassembly
V9.2S[1]    //optional number of lanes
V9.4S[1]    //optional number of lanes
```

Note

The SIMD and floating-point register element name `Vn.S[0]` is not equivalent to the scalar SIMD and floating-point register name `Sn`. Although they represent the same bits in the register, they select different instruction encoding forms, either the vector element or the scalar form.

C1.2.6.5.1 SIMD vector register list

Where an instruction operates on multiple SIMD&FP or SVE registers, for example vector load/store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register numbers are increasing by 1. The following examples are equivalent representations of a set of four registers `V4` to `V7`, each holding four lanes of 32-bit elements:

```
{ V4.4S - V7.4S }              //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S } //alternative representation
```

C1.2.6.5.2 SIMD vector element list

Registers in a list can also have a vector element form. For example, the `LD4` instruction can load one element into each of four registers, and in this case the index is appended to the list as follows:

```
{ V4.S - V7.S }[3]                //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S }[3] //alternative with optional number of lanes
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.3 Address generation

The A64 instruction set supports 64-bit virtual addresses (VAs). The valid VA range is determined by the following factors:

- The size of the implemented virtual address space.
- *Memory Management Unit* (MMU) configuration settings.
- *Memory Protection Unit* (MPU) configuration settings.

Limits on the VA size mean that the most significant bits of the virtual address do not hold valid address bits. These unused bits can hold:

- A tag, see Address tagging.
- If FEAT_PAuth is implemented, a Pointer authentication code (PAC), see Pointer authentication.

For more information on memory management and address translation, see The AArch64 Virtual Memory System Architecture.

C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative load/store operations, and address generation with a range of ± 4GB can be performed using two instructions.

C1.3.3 Load/store addressing modes

Load/store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, `SP`, with an optional immediate or register offset. Table C1-7 shows the assembler syntax for the complete set of load/store addressing modes.

Table C1-7 A64 Load/store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	`[base{, #0}]`	-	-
Base plus offset	`[base{, #imm}]`	`[base, Xm{, LSL #imm}]`	`[base, Wm, (S|U)XT(X|W){#imm}]`
Pre-indexed	`[base, #imm]!`	-	-
Post-indexed	`[base], #imm`	`[base], Xm`[a]	-
Literal (PC-relative)	`label`	-	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

a. The post-indexed by register offset mode can be used with the SIMD load/store structure instructions described in Load/store Advanced SIMD. Otherwise the post-indexed by register offset mode is not available.

Some types of load/store instruction support only a subset of the load/store addressing modes listed in Table C1-7. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can be used only for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of load/store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of load/store instruction and the transfer size. Table C1-8 shows the offset and the type of load/store instruction.

Table C1-8 Immediate offsets and the type of load/store instruction

Offset bits	Sign	Scaling	Write-Back	Load/store type
0	-	-	-	Exclusive/acquire/release
7	Signed	Scaled	Optional	Register pair
9	Signed	Unscaled	Optional	Single register
12	Unsigned	Scaled	No	Single register

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if `LSL #imm` is present and where `imm` must be equal to $\log_2$(transfer_size). The `SXTX` extend/shift option is functionally equivalent to `LSL`, but the `LSL` option is preferred in source code.
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by `#imm`, where imm must be equal to $\log_2$(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific load/store instruction requires this. SP cannot be used as a register offset.

C1.3.3.1 Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Table C1-9 shows the arithmetic instructions that can calculate addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C1-9 Arithmetic instructions to calculate addressing modes

Addressing Form	Offset		
	Immediate	Register	Extended Register
Base register (no offset)	`MOV Xd|SP, base`	-	-

Base plus offset	`ADD Xd|SP, base, #imm`
	or
	`SUB Xd|SP, base, #imm`
		`ADD <Xd|SP>, base, Xm{,LSL#imm}`	`ADD <Xd|SP>, base, Wm,(S|U)XT(W|H|B|X) {#imm}`

Pre-indexed	-	-	-
Post-indexed	-	-	-
Literal (PC-relative)	`ADR Xd, label`	-	-

Note

- For the 64-bit base plus register offset form, the `UXTX` mnemonic is an alias for the `LSL` shift option, but `LSL` is preferred for disassembly. Similarly the `SXTX` extend/shift option is functionally equivalent to the `LSL` option, but the `LSL` option is preferred in source code.
- To calculate a base plus immediate offset the `ADD` instructions defined in Arithmetic (immediate) accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single `ADD` instruction cannot support the full range of byte offsets available to a single register load/store with a scaled 12-bit immediate offset. For example, a quadword `LDR` effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two `ADD` instructions. The following example shows this:

```
ADD Xd, base, #(imm & 0xFFF)
ADD Xd, Xd, #(imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the `ADD` instructions defined in Arithmetic (extended register) provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```
ADD  Xd, base, Wm, SXTW #3   // Xd = base + (SignExtend(Wm) LSL 3)
ADD  Xd, base, Wm, UXTH #4   // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one load/store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then reuse it as a simple register offset. The extend and scale calculation can be performed using the `SBFIZ` and `UBFIZ` bitfield instructions defined in Bitfield move, for example:

```
SBFIZ Xd, Xm, #3, #32   //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16   //Xd = "Wm, UXTH #4"
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.4 Instruction aliases

Some instructions have an associated *architecture alias* that is used for disassembly of the encoding when the associated conditions are met. Architecture alias instructions are included in the alphabetic lists of instruction types and clearly presented as an alias form in descriptions for the individual instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C2
About the A64 Instruction Descriptions

This chapter describes the *instruction descriptions* contained in A64 Base Instruction Descriptions, A64 Advanced SIMD and Floating-point Instruction Descriptions, SVE Instruction Descriptions, and SME Instruction Descriptions.

It contains the following sections:

- Understanding the A64 instruction descriptions.
- General information about the A64 instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.1 Understanding the A64 instruction descriptions

Each instruction description in A64 Base Instruction Descriptions, A64 Advanced SIMD and Floating-point Instruction Descriptions, SVE Instruction Descriptions, and SME Instruction Descriptions has the following content:

1. A title.
2. An introduction to the instruction.
3. The instruction encoding or encodings.
4. Any alias conditions.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

The following sections describe each of these.

C2.1.1 The title

The title of an instruction description includes the base mnemonic for the instruction.

If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in A64 Base Instruction Descriptions there are the following titles for different forms of the `ADD` instruction:

- ADD (extended register).
- ADD (immediate).
- ADD (shifted register).

C2.1.2 An introduction to the instruction

This briefly describes the function of the instruction. The introduction is not a complete description of the instruction, and it is not definitive. If there is any conflict between it and the more detailed information that follows it, the more detailed information takes priority.

C2.1.3 The instruction encoding or encodings

This shows the instruction encoding diagram, or if the instruction has more than one encoding, shows all of the encoding diagrams. Each diagram has a subheading.

For example, for load and store instructions, the subheadings might be:

- Post-index.
- Pre-index.
- Unsigned offset.

Each diagram numbers the bits from 31 to 0. The diagram for an instruction at address *A* shows, from left to right, the bytes at addresses *A*+3, *A*+2, *A*+1, and *A*.

There might be variants of an encoding, if the *assembler syntax prototype* differs depending on the value in one or more of the encoding fields. In this case, each variant has a subheading that describes the variant and shows the distinguishing field value or values in parentheses. For example, in A64 Base Instruction Descriptions there are the following subheadings for variants of the `ADC` instruction encoding:

- 32-bit variant (sf = 0).
- 64-bit variant (sf = 1).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The assembler syntax prototype for an encoding or variant of an encoding shows how to form a complete assembler source code instruction that assembles to the encoding. Unless otherwise stated, the prototype is also the preferred syntax for a disassembler to disassemble the encoding to. Disassemblers are permitted to omit optional symbols that represent the default value of a field or set of fields, to produce more readable disassembled code, provided that the output re-assembles to the same encoding.

Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See Pseudocode describing how the instruction operates.

C2.1.4 Any alias conditions, if applicable

This is an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

C2.1.5 A list of the assembler symbols for the instruction

The *Assembler symbols* subsection of the instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use, if any.

In assembler syntax prototypes, the following conventions are used:

`< >` Angle brackets. Any symbol enclosed by these is a name or a value that the user supplies. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

`{ }` Brace brackets. Any symbols enclosed by these are optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

`#` This usually precedes a numeric constant. All uses of # are optional in A64 assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

`+/-` This indicates an optional + or – sign. If neither is coded, + is assumed.

Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, the characters shown are used as part of a meta-language to define the architectural assembler syntax for an instruction encoding or alias, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

The following symbol conventions are used:

`<Xn>` The 64-bit name of a general-purpose register (X0-X30) or the zero register (XZR).

`<Wn>` The 32-bit name of a general-purpose register (W0-W30) or the zero register (WZR).

`<Xn|SP>` The 64-bit name of a general-purpose register (X0-X30) or the current stack pointer (SP).

`<Wn|WSP>` The 32-bit name of a general-purpose register (W0-W30) or the current stack pointer (WSP).

`<Bn>, <Hn>, <Sn>, <Dn>, <Qn>`

The 8, 16, 32, 64, or 128-bit name of a SIMD and floating-point register in a scalar context, as described in Register names.

`<Vn>` The name of a SIMD and floating-point register in a vector context, as described in Register names.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<Zn>` The name of an SVE scalable vector register, as described in Treatment of SVE scalable vector registers.

`<Pn>` The name of an SVE scalable predicate register, as described in Vector predication.

If the description of a symbol specifies that the symbol is a register, the description might also specify that the range of permitted registers is extended or restricted. It also specifies any differences from the default rules for such fields.

For information about SME conventions, see ZA storage.

Note

Register names provides the A64 register names.

C2.1.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

Note

For a description of Arm pseudocode and the execution model for an instruction, see *Arm® Architecture Specification Language Reference* (ARM DDI 0626).

C2.1.7 Notes, if applicable

If applicable, other notes about the instruction appear under additional subheadings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.2 General information about the A64 instruction descriptions

This section provides general information about the A64 instruction descriptions. Some of this information also applies to System register descriptions, for example the terms defined in Fixed values in AArch64 instruction and System register descriptions apply to the AArch64 descriptions throughout this manual. The following subsections provide this information:

- Execution of instructions in Debug state.
- Fixed values in AArch64 instruction and System register descriptions.
- Modified immediate constants in A64 floating-point instructions.

C2.2.1 Execution of instructions in Debug state

In general, except for the instructions described in Debug state, the A64 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see Executing instructions in Debug state.

Note

For many instructions, execution is unchanged in Debug state. Executing instructions in Debug state identifies these instructions.

C2.2.2 Fixed values in AArch64 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See Read-As-Zero (RAZ).

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, Should-Be-Zero (SBZ) or RES0.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

The `(0)` and RES0 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RAO Read-As-One. See Read-As-One (RAO).

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, Should-Be-One (SBO) or RES1.

In instruction encoding diagrams, and sometimes in other descriptions, `(1)` indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(1)(1)(1)` or as `(111)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of *RES1* for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields,

The `(1)` and RES1 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

C2.2.3 Modified immediate constants in A64 floating-point instructions

Table C2-1 shows the immediate constants available in `FMOV` (scalar, immediate) and `FMOV` (vector, immediate) floating-point instructions.

Table C2-1 A64 Floating-point modified immediate constants

Data type	immediate	Constant[a]
`F16`	abcdefgh	`aBbbcdef gh000000`
`F32`	abcdefgh	`aBbbbbbc defgh000 00000000 00000000`
`F64`	abcdefgh	`aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000`

a. In this column, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

The immediate value shown in the table is either:

- The value of the `imm8` field for an `FMOV` (scalar, immediate) instruction, see FMOV (scalar, immediate).
- The value obtained by concatenating the `a:b:c:d:e:f:g:h` fields for an `FMOV` (vector, immediate) instruction, see FMOV (vector, immediate).

Table C2-2 shows the floating-point constant values encoded in the `b:c:d:e:f:g:h` fields of the `FMOV` (vector, immediate) instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C2-2 Floating-point constant values

efgh	bcd							
	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

C2.2.3.1 Operation of modified immediate constants, floating-point instructions

For an A64 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C3
A64 Instruction Set Overview

This chapter provides an overview of the A64 instruction set. It contains the following sections:

- Branches, Exception generating, and System instructions.
- Loads and stores.
- Loads and stores - SVE, SVE2.
- Loads and stores - SME, SME2, SVE2p1.
- Data processing - immediate.
- Data processing - MTE.
- Data processing - register.
- Data processing - SIMD and floating-point.
- Data processing - SVE.
- Data processing - SVE2.
- Data processing - SME, SME2.

For a structured breakdown of instruction groups by encoding, see A64 Instruction Set Encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and System instructions. It contains the following subsections:

- Conditional branch.
- Unconditional branch (immediate).
- Unconditional branch (register).
- Direct and indirect branch.
- Exception generation and return.
- System register instructions.
- System instructions.
- Hint instructions.
- Barriers and CLREX instructions.
- Pointer authentication instructions.

Note

Software must:

- Use only `BLR` or `BL` to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the `BLR` or `BL` instruction incremented by four.
- Use only `RET` to perform a subroutine return, when that subroutine is expected to have been entered by a `BL` or `BLR` instruction.
- Use only `B`, `BR`, or the instructions listed in Table C3-1 to perform a control transfer that is not a subroutine call or subroutine return described in this *Note*.

C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the following:

- The current state of the Condition flags.
- The result of comparison between 0 and a value or a bit in a general purpose register.
- The result of comparison between values in two general purpose registers.
- The result of comparison between an immediate value and a value in a general purpose register.

See Table C1-1 for a list of the Condition codes that can be used for `cond`.

Table C3-1 shows the Conditional branch instructions.

Table C3-1 Conditional branch instructions

Mnemonic	Instruction	Branch offset range from the PC	See
`B.cond`	Branch conditionally	±1MB	B.cond
`BC.cond`	Branch Consistent conditionally	±1MB	BC.cond
`CBEQ`	Compare equal to immediate and branch	±1KB	CB<cc> (immediate)
	Compare equal to register and branch		CB<cc> (register)
`CBGE`	Compare signed greater than or equal to immediate and branch	±1KB	CBGE (immediate)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	Branch offset range from the PC	See
	Compare signed greater than or equal to register and branch		CB<cc> (register)
CBGT	Compare signed greater than immediate and branch	±1KB	CB<cc> (immediate)
	Compare signed greater than register and branch		CB<cc> (register)
CBHI	Compare unsigned higher than immediate and branch	±1KB	CB<cc> (immediate)
	Compare unsigned higher than register and branch		CB<cc> (register)
CBHS	Compare unsigned higher than or same as immediate and branch	±1KB	CBHS (immediate)
	Compare unsigned higher than or same as register and branch		CB<cc> (register)
CBLE	Compare signed less than or equal to immediate and branch	±1KB	CBLE (immediate)
	Compare signed less than or equal to register and branch		CBLE (register)
CBLO	Compare unsigned lower than immediate and branch	±1KB	CB<cc> (immediate)
	Compare unsigned lower than register and branch		CBLO (register)
CBLS	Compare unsigned lower than or same as immediate and branch	±1KB	CBLS (immediate)
	Compare unsigned lower than or same as register and branch		CBLS (register)
CBLT	Compare signed less than immediate and branch	±1KB	CB<cc> (immediate)
	Compare signed less than register and branch		CBLT (register)
CBNE	Compare not equal to immediate and branch	±1KB	CB<cc> (immediate)
	Compare not equal to register and branch		CB<cc> (register)
CBBEQ	Compare equal to bytes and branch	±1KB	CBB<cc>
CBBGE	Compare signed greater than or equal to bytes and branch		
CBBGT	Compare signed greater than bytes and branch		
CBBHI	Compare unsigned higher than bytes and branch		
CBBHS	Compare unsigned higher than or same as bytes and branch		
CBBLE	Compare signed less than or equal to bytes and branch	±1KB	CBBLE
CBBLO	Compare unsigned lower than bytes and branch	±1KB	CBBLO
CBBLS	Compare unsigned lower than or same as bytes and branch	±1KB	CBBLS
CBBLT	Compare signed less than bytes and branch	±1KB	CBBLT
CBBNE	Compare not equal to bytes and branch	±1KB	CBB<cc>
CBHEQ	Compare equal to halfwords and branch	±1KB	CBH<cc>
CBHGE	Compare signed greater than or equal to halfwords and branch		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	Branch offset range from the PC	See
CBHGT	Compare signed greater than halfwords and branch		
CBHHI	Compare unsigned higher than halfwords and branch		
CBHHS	Compare unsigned higher than or same as halfwords and branch		
CBHLE	Compare signed less than or equal halfwords and branch	±1KB	CBHLE
CBHLO	Compare unsigned lower than halfwords and branch	±1KB	CBHLO
CBHLS	Compare unsigned lower than or same as halfwords and branch	±1KB	CBHLS
CBHLT	Compare signed less than halfwords and branch	±1KB	CBHLT
CBHNE	Compare not equal to halfwords and branch	±1KB	CBH<cc>
CBNZ	Compare and branch if nonzero	±1MB	CBNZ
CBZ	Compare and branch if zero	±1MB	CBZ
TBNZ	Test bit and branch if nonzero	±32KB	TBNZ
TBZ	Test bit and branch if zero	±32KB	TBZ

C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The BL instruction also writes the address of the sequentially following instruction to general-purpose register, X30.

Table C3-2 shows the Unconditional branch instructions with an immediate branch offset.

Table C3-2 Unconditional branch instructions (immediate)

Mnemonic	Instruction	Immediate branch offset range from the PC	See
B	Branch unconditionally	±128MB	B
BL	Branch with link	±128MB	BL

C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The BLR instruction also writes the address of the sequentially following instruction to general-purpose register X30. The RET instruction behaves identically to BR, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-3 Unconditional branch instructions (register)

Mnemonic	Instruction	See
BLR	Branch with link to register	BLR
BR	Branch to register	BR
RET	Return from subroutine	RET

C3.1.4 Direct and indirect branch

Table C3-4 and Table C3-5 shows the Direct branch and Indirect branch instructions.

Table C3-4 Direct branch instructions

Mnemonic	Instruction	See
B	Unconditional branch	B
B.cond	Conditional branch	B.cond
BC.cond	Branch consistent conditionally	BC.cond
BL	Branch with link	BL
CB<cc>	Compare register with immediate and branch	CB<cc> (immediate)
	Compare registers and branch	CB<cc> (register)
CBB<cc>	Compare bytes and branch	CBB<cc>
CBH<cc>	Compare halfwords and branch	CBH<cc>
CBZ	Compare and branch on zero	CBZ
CBNZ	Compare and branch on nonzero	CBNZ
TBZ	Test bit and branch if zero	TBZ
TBNZ	Test bit and branch if nonzero	TBZ

Table C3-5 Indirect branch instructions

Mnemonic	Instruction	See
BLR	Branch with link to register	BLR
BLRAA	Branch with link to register, with pointer authentication	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRAAZ		
BLRAB		
BLRABZ		
BR	Branch to register	BR
BRAA	Branch to register, with pointer authentication	BRAA, BRAAZ, BRAB, BRABZ
BRAAZ		
BRAB		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BRABZ		
ERET	Exception return	ERET
ERETAA	Exception return, with pointer authentication	ERETAA, ERETAB
ERETAB		
RET	Return from subroutine	RET
RETAA	Return from subroutine, with pointer authentication	RETAA, RETAB
RETAB		
RETAASPPC	Return from subroutine, with enhanced pointer authentication using an immediate offset	RETAASPPC, RETABSPPC
RETABSPPC		
RETAASPPCR	Return from subroutine, with enhanced pointer authentication using a register	RETAASPPCR, RETABSPPCR
RETABSPPCR		

C3.1.5 Exception generation and return

This section describes the following exceptions:

- Exception generating.
- Exception return.
- Debug state.

C3.1.5.1 Exception generating

Table C3-6 shows the Exception generating instructions.

Table C3-6 Exception generating instructions

Mnemonic	Instruction	See
BRK	Breakpoint Instruction	BRK
HLT	Halt Instruction	HLT
HVC	Generate exception targeting Exception level 2	HVC
SMC	Generate exception targeting Exception level 3	SMC
SVC	Generate exception targeting Exception level 1	SVC

C3.1.5.2 Exception return

Table C3-7 shows the Exception return instructions.

Table C3-7 Exception return instructions

Mnemonic	Instruction	See
ERET	Exception return using current ELR and SPSR	ERET

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
ERETAA, ERETAB	Exception return, with pointer authentication	ERETAA, ERETAB

C3.1.5.3 Debug state

Table C3-8 shows the Debug state instructions.

Table C3-8 Debug state instructions

Mnemonic	Instruction	See
DCPS1	Debug switch to Exception level 1	DCPS1
DCPS2	Debug switch to Exception level 2	DCPS2
DCPS3	Debug switch to Exception level 3	DCPS3
DRPS	Debug restore PE state	DRPS

C3.1.6 System register instructions

For detailed information about the System register instructions, see The A64 System Instruction Class. Table C3-9 shows the System register instructions.

If FEAT_SYSREG128 is implemented, the following instructions are added that allow the PE to move values between a 128-bit System register and two adjacent 64-bit general-purpose registers:

- MRRS.
- MSRR.

Table C3-9 System register instructions

Mnemonic	Instruction	See
MRS	Move System register to general-purpose register	MRS
MSR	Move general-purpose register to System register	MSR (register)
	Move immediate to PE state field	MSR (immediate)
MRRS	Move 128-bit System register to two adjacent 64-bit general-purpose registers	MRRS
MSRR	Move two adjacent 64-bit general-purpose registers to 128-bit System register	MSRR

C3.1.7 System instructions

For detailed information about the System instructions, see The A64 System Instruction Class.

If FEAT_SYSINSTR128 is implemented, the 128-bit System instruction, SYSP, is supported.

Table C3-10 shows the System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-10 System instructions

Mnemonic	Instruction	See
SYS	System instruction	SYS
SYSL	System instruction with result	SYSL
SYSP	128-bit System instruction	SYSP
IC	Instruction cache maintenance	IC and Table C5-1
DC	Data cache maintenance	DC and Table C5-1
AT	Address translation	AT and Table C5-3
TLBI	TLB Invalidate	TLBI and Table C5-4

C3.1.8 Hint instructions

Table C3-11 shows the Hint instructions.

Table C3-11 Hint instructions

Mnemonic	Instruction	See
NOP	No operation	NOP
YIELD	Yield hint	YIELD
WFE	Wait for event	WFE
WFET	Wait for event with timeout	WFET
WFI	Wait for interrupt	WFI
WFIT	Wait for interrupt with timeout	WFIT
SEV	Send event	SEV
SEVL	Send event local	SEVL
HINT	Unallocated hint	HINT
DGH	Data Gathering Hint	DGH
CLRBHB	Clear Branch History	CLRBHB
CHKFEAT	Check Feature	CHKFEAT
STSHH	Store Shared Hint	STSHH

C3.1.9 Barriers and CLREX instructions

Table C3-12 shows the barrier and CLREX instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-12 Barriers and CLREX instructions

Mnemonic	Instruction	See
CLREX	Clear Exclusives monitor	CLREX
DMB	Data memory barrier	DMB
DSB	Data synchronization barrier	DSB
ISB	Instruction synchronization barrier	ISB

For more information about DSB, DMB, and ISB, see Memory barriers.

Table C3-13 shows the speculation and synchronization barriers. If these instructions are not implemented, then these instructions execute as a NOP.

Table C3-13 Speculation and synchronization barriers

Mnemonic	Instruction	See
CSDB	Consumption of Speculative Data Barrier	CSDB
ESB	Error synchronization barrier	ESB
PSB	Profiling synchronization barrier	PSB
PSSBB	Physical Speculative Store Bypass Barrier	PSSBB
SB	Speculation Barrier	SB
SSBB	Speculative Store Bypass Barrier	SSBB
TSB	Trace Synchronization Barrier	TSB
GCSB	Guarded Control Stack Barrier	GCSB

For more information about:

- CSDB, PSSBB, SB, SSBB, TSB, see Memory barriers.
- ESB, see Error synchronization event.
- PSB, see The Statistical Profiling Extension.
- GCSB, see Guarded Control Stack data accesses.

C3.1.10 Pointer authentication instructions

FEAT_PAuth adds support for pointer authentication, see Pointer authentication. This functionality includes the A64 instructions described in this section. These instructions fall into two groups, see:

- Basic pointer authentication instructions.
- Combined instructions that include pointer authentication.

C3.1.10.1 Basic pointer authentication instructions

Each of these instructions only performs an operation that supports pointer authentication.

Table C3-14 shows the instructions that add a *Pointer Authentication Code* (PAC) to the address in a register:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-14 Instructions that add a PAC

Mnemonic	Instruction	See
PACIASP	Add PAC to instruction address using APIAKey_EL1 and SP	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACIAZ	Add PAC to instruction address using APIAKey_EL1 and zero	
PACIA1716	Add PAC to instruction address X17 using APIAKey_EL1 and X16	
PACIA171615	Add PAC to instruction address X17 using APIAKey_EL1, X16, X15	PACIA171615
PACIASPPC	Add PAC to instruction address LR using APIAKey_EL1, SP, PC	PACIASPPC
PACNBIASPPC	Add PAC to instruction address LR using APIAKey_EL1, SP, PC	PACNBIASPPC
PACIBSP	Add PAC to instruction address using APIBKey_EL1 and SP	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACIBZ	Add PAC to instruction address using APIBKey_EL1 and zero	
PACIB1716	Add PAC to instruction address X17 using APIBKey_EL1 and X16	
PACIB171615	Add PAC to instruction address X17 using APIBKey_EL1, X16, X15	PACIB171615
PACIBSPPC	Add PAC to instruction address LR using APIBKey_EL1, SP, PC	PACIBSPPC
PACNBIBSPPC	Add PAC to instruction address LR using APIBKey_EL1, SP, PC	PACNBIBSPPC
PACIA	Add PAC to instruction address using APIAKey_EL1, registers	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACDA	Add PAC to data address using APDAKey_EL1, registers	PACDA, PACDZA
PACIB	Add PAC to instruction address using APIBKey_EL1, registers	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACDB	Add PAC to data address using APDBKey_EL1, registers	PACDB, PACDZB
PACIZA	Add PAC to instruction address using APIAKey_EL1, register and zero	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACDZA	Add PAC to data address using APDAKey_EL1, register and zero	PACDA, PACDZA
PACIZB	Add PAC to instruction address using APIBKey_EL1, register and zero	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACDZB	Add PAC to data address using APDBKey_EL1, register and zero	PACDB, PACDZB
PACGA	Add generic PAC using APGAKey_EL1, registers	PACGA

Table C3-15 shows the instructions that authenticate a PAC in a register:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-15 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIA171615	Authenticate PAC for instruction address X17 using APIAKey_EL1, X16, and X15	AUTIA171615
AUTIASPPC	Authenticate PAC for instruction address LR using APIAKey_EL1, SP, and PC with an immediate offset	AUTIASPPC
AUTIASPPCR	Authenticate PAC for instruction address LR using APIAKey_EL1, SP, and a register	AUTIASPPCR
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIB171615	Authenticate PAC for instruction address X17 using APIBKey_EL1, X16, X15	AUTIB171615
AUTIBSPPC	Authenticate PAC for instruction address LR using APIBKey_EL1, SP, and PC with an immediate offset	AUTIBSPPC
AUTIBSPPCR	Authenticate PAC for instruction address LR using APIBKey_EL1, SP, and a register	AUTIBSPPCR
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	AUTDA, AUTDZA
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	AUTDB, AUTDZB
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	AUTDA, AUTDZA
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	AUTDB, AUTDZB

Table C3-15 shows the instructions that authenticate a PAC in a register:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-16 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	AUTDA, AUTDZA
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	AUTDB, AUTDZB
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	AUTDA, AUTDZA
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	AUTDB, AUTDZB

Table C3-17 shows the instructions that strip a PAC from a register, without performing any authentication:

Table C3-17 Instructions that strip a PAC

Mnemonic	Instruction	See
XPACLRI	Strip instruction address PAC from LR	XPACD, XPACI, XPACLRI
XPACI	Strip instruction address PAC, register	
XPACD	Strip data address PAC, register	

Table C3-18 shows the instructions that set PSTATE.PACM to 1:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-18 Instructions that set PSTATE.PACM to 1

Mnemonic	Instruction	See
PACM	Pointer authentication modifier	PACM

C3.1.10.2 Combined instructions that include pointer authentication

Each of these instructions combines a pointer authentication with another operation that uses the authenticated pointer. Table C3-19 shows these instructions:

Table C3-19 Combined pointer authentication instructions

Mnemonic	Instruction	See
RETAA	Authenticate PAC for LR using APIAKey_EL1 and SP, and return	RETAA, RETAB
RETAB	Authenticate PAC for LR using APIBKey_EL1 and SP, and return	
RETAASPPC	Authenticate PAC for LR using APIAKey_EL1, SP, and PC with an immediate offset, and return	RETAASPPC, RETABSPPC
RETABSPPC	Authenticate PAC for LR using APIBKey_EL1, SP, and PC with an immediate offset, and return	
RETAASPPCR	Authenticate PAC for LR using APIAKey_EL1, SP, and a register, and return	RETAASPPCR, RETABSPPCR
RETABSPPCR	Authenticate PAC for LR using APIBKey_EL1, SP, and a register, and return	
BRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch	BRAA, BRAAZ, BRAB, BRABZ
BRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch	
BLRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch with link	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch with link	
BRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch	BRAA, BRAAZ, BRAB, BRABZ
BRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch	
BLRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch with link	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch with link	
ERETAA	Authenticate PAC for ELR using APIAKey_EL1 and SP, and exception return	ERETAA, ERETAB
ERETAB	Authenticate PAC for ELR using APIBKey_EL1 and SP, and exception return	
LDRAA	Authenticate PAC for data address using APDAKey_EL1 (register and zero) and Load	LDRAA, LDRAB
LDRAB	Authenticate PAC for data address using APDBKey_EL1 (register and zero) and Load	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1.11 Checked Pointer Arithmetic instructions

FEAT_CPA adds support for Checked Pointer Arithmetic, see Checked Pointer Arithmetic.

If FEAT_CPA is implemented, the instructions shown in Table C3-20 are implemented.

Table C3-20 Checked Pointer Arithmetic instructions

Mnemonic	Instruction	See
ADDPT	Scalar add checked pointer	ADDPT
	SVE add checked pointer (predicated)	ADDPT
	SVE add checked pointer (unpredicated)	ADDPT
SUBPT	Scalar subtract checked pointer	SUBPT
	SVE subtract checked pointer (predicated)	SUBPT
	SVE subtract checked pointer (unpredicated)	SUBPT
MADDPT	Scalar multiply-add checked pointer	MADDPT
MADPT	SVE multiply-add checked pointer vectors, writing multiplicand	MADPT
MLAPT	SVE Multiply-add checked pointer vectors, writing addend	MLAPT
MSUBPT	Scalar multiply-subtract checked pointer	MSUBPT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2 Loads and stores

This section describes the load/store instructions. It contains the following subsections:

- Load/store register.
- Load/store register (unscaled offset).
- Load/store pair.
- Load/store pair unprivileged.
- Load/store non-temporal pair.
- Load/store non-temporal pair unprivileged.
- Load/store unprivileged.
- Load-Exclusive/Store-Exclusive.
- Load-Exclusive/Store-Exclusive unprivileged.
- Load-Acquire/Store-Release.
- Load-Acquire/Store-Release unprivileged.
- LoadLOAcquire/StoreLORelease.
- Load/store scalar SIMD and floating-point.
- Load/store Advanced SIMD.
- Prefetch memory.
- Atomic instructions.
- Memory Tagging instructions.
- Memory Copy and Memory Set instructions.

The requirements for the alignment of data memory accesses are strict. For more information, see Alignment of data accesses.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See SP alignment checking. Using a misaligned stack pointer generates an SP alignment fault exception.

Note

In some cases, load/store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See AArch64 CONSTRAINED UNPREDICTABLE behaviors.

C3.2.1 Load/store register

The load/store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also Load/store addressing modes.

If a Load instruction specifies writeback and the register being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store instruction performs a writeback and the register that is stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is UNKNOWN.

Table C3-21 shows the load/store register instructions.

Table C3-21 Load/store register instructions

Mnemonic	Instruction	See
LDR	Load register (register offset)	LDR (register)
	Load register (immediate offset)	LDR (immediate)
	Load register (PC-relative literal)	LDR (literal)
LDRB	Load byte (register offset)	LDRB (register)
	Load byte (immediate offset)	LDRB (immediate)
LDRSB	Load signed byte (register offset)	LDRSB (register)
	Load signed byte (immediate offset)	LDRSB (immediate)
LDRH	Load halfword (register offset)	LDRH (register)
	Load halfword (immediate offset)	LDRH (immediate)
LDRSH	Load signed halfword (register offset)	LDRSH (register)
	Load signed halfword (immediate offset)	LDRSH (immediate)
LDRSW	Load signed word (register offset)	LDRSW (register)
	Load signed word (immediate offset)	LDRSW (immediate)
	Load signed word (PC-relative literal)	LDRSW (literal)
STR	Store register (register offset)	STR (register)
	Store register (immediate offset)	STR (immediate)
STRB	Store byte (register offset)	STRB (register)
	Store byte (immediate offset)	STRB (immediate)
STRH	Store halfword (register offset)	STRH (register)
	Store halfword (immediate offset)	STRH (immediate)

C3.2.2 Load/store register (unscaled offset)

The load/store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See Load/store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The load/store register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:

- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a load/store instruction, for example `LDR`, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-22. Arm recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a load/store single register mnemonic, for example, `LDR`.

Table C3-22 shows the load/store register instructions with an unscaled offset.

Table C3-22 Load/store register (unscaled offset) instructions

Mnemonic	Instruction	See
`LDUR`	Load register (unscaled offset)	LDUR
`LDURB`	Load byte (unscaled offset)	LDURB
`LDURSB`	Load signed byte (unscaled offset)	LDURSB
`LDURH`	Load halfword (unscaled offset)	LDURH
`LDURSH`	Load signed halfword (unscaled offset)	LDURSH
`LDURSW`	Load signed word (unscaled offset)	LDURSW
`STUR`	Store register (unscaled offset)	STUR
`STURB`	Store byte (unscaled offset)	STURB
`STURH`	Store halfword (unscaled offset)	STURH

C3.2.3 Load/store pair

The load/store pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.

Table C3-23 shows the load/store pair instructions.

Table C3-23 Load/store pair instructions

Mnemonic	Instruction	See
LDP	Load Pair	LDP
LDPSW	Load Pair signed words	LDPSW
STP	Store Pair	STP

C3.2.4 Load/store pair unprivileged

The Load unprivileged pair and Store unprivileged pair instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store unprivileged pair instructions support the following address modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See Load/store addressing modes.

Table C3-24 Load/store unprivileged pair instructions

Mnemonic	Instruction	See
LDTP	Load unprivileged Pair	LDTP
STTP	Store unprivileged Pair	STTP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.5 Load/store non-temporal pair

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See Load/store addressing modes.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be prefetched and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-25 shows the load/store non-temporal pair instructions.

Table C3-25 Load/store non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load Non-temporal Pair	LDNP
STNP	Store Non-temporal Pair	STNP

C3.2.6 Load/store non-temporal pair unprivileged

The load/store non-temporal pair unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

Table C3-26 Load/store non-temporal pair instructions

Mnemonic	Instruction	See
LDTNP	Load Non-temporal Pair	LDTNP
STTNP	Store Non-temporal Pair	STTNP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.7 Load/store unprivileged

The load/store unprivileged instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See Load/store addressing modes.

The access permissions that apply to accesses made at EL0 apply to the memory accesses made by a load/store unprivileged instruction that is executed either:

- At EL1 when the Effective value of PSTATE.UAO is 0.
- At EL2 when both the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the Effective value of PSTATE.UAO is 0.

Otherwise, memory accesses made by a load/store unprivileged instruction are subject to the access permissions that apply to the Exception level at which the instruction is executed. These are the permissions that apply to the corresponding load/store register instruction, see Load/store register.

Note

This means that when the value of PSTATE.UAO is 1 the access permissions for a load/store unprivileged instruction are always the same as those for the corresponding load/store register instruction.

Table C3-27 shows the load/store unprivileged instructions.

Table C3-27 Load-Store unprivileged instructions

Mnemonic	Instruction	See
`LDTR`	Load unprivileged register	LDTR
`LDTRB`	Load unprivileged byte	LDTRB
`LDTRSB`	Load unprivileged signed byte	LDTRSB
`LDTRH`	Load unprivileged halfword	LDTRH
`LDTRSH`	Load unprivileged signed halfword	LDTRSH
`LDTRSW`	Load unprivileged signed word	LDTRSW
`STTR`	Store unprivileged register	STTR
`STTRB`	Store unprivileged byte	STTRB
`STTRH`	Store unprivileged halfword	STTRH

C3.2.8 Single-copy atomic 64-byte load/store

If FEAT_LS64 is implemented, the following instructions are implemented:

- LD64B.
- ST64B.

If FEAT_LS64_V is implemented, the following instructions are implemented:

- LD64B.
- ST64B.
- ST64BV.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_LS64_ACCDATA is implemented, the following instructions are implemented:

- LD64B.
- ST64B.
- ST64BV.
- ST64BV0.

LD64B and ST64B can only target Device or Normal Non-cacheable memory unless FEAT_LS64WB is implemented.

The single-copy atomic 64-byte load/store instructions support one addressing mode:

- Base register only.

See also Load/store addressing modes.

The memory location accessed by the instructions is required to be aligned on a 64-byte boundary, otherwise an Alignment fault occurs.

If an ST64BV, ST64BV0, or if FEAT_LS64WB is not implemented an LD64B or ST64B, attempts to access a memory type that is not one of the following, then a Data Abort with DFSC code `0b110101` is generated:

- Normal Inner Non-cacheable, Outer Non-cacheable.
- Device-GRE.
- Device-nGRE.
- Device-nGnRE.
- Device-nGnRnE.

It is IMPLEMENTATION DEFINED which of the following approaches is used to provide this check:

- The check is performed at each enabled stage of translation, and the fault is reported for the first stage of translation that provides an inappropriate memory type. In this case, the value of the HCR_EL2.DC bit does not cause accesses generated by the instructions to generate a stage 1 Data abort,
- The check is performed against the resulting memory type after all enabled stages of translation. In this case the fault is reported at the final enabled stage of translation.
- The check is performed against the resulting memory type after all enabled stages of translation. In this case the fault is reported for the first stage of translation that provided an inappropriate memory type. The value of the HCR_EL2.DC bit does not cause accesses generated by the instructions to generate a stage 1 Data abort.

If FEAT_LS64WB is implemented, then LD64B and ST64B are also supported and 64-byte single-copy atomic for Conventional memory that is Inner and Outer Write-Back Cacheable, Shareable. In this case, if an LD64B or ST64B attempts to access:

- A memory region that is Inner and Outer Write-Back Cacheable, Shareable, but the target memory location is not in Conventional memory, then it is IMPLEMENTATION DEFINED whether:
  - A stage 1 Data Abort with DFSC code `0b110101` is generated.
  - The access succeeds but is not guaranteed to be single-copy atomic above the byte level.
- A cacheable type of memory other than Inner and Outer Write-Back Cacheable, Shareable, then it is IMPLEMENTATION DEFINED whether:
  - A stage 1 Data Abort with DFSC code `0b110101` is generated.
  - The access succeeds as a 64-byte single-copy atomic operation.

For behavior when FEAT_LS64WB is implemented and an LD64B or ST64B targets Tagged memory that is Inner and Outer Write-Back Cacheable, Shareable, see Tag Check Faults.

If the target memory location does not support the LD64B or ST64B instructions, then one of the following behaviors occurs:

- A stage 1 Data Abort with DFSC code of `0b110101`, is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

If the target memory location does not support the ST64BV or ST64BV0 instructions, then the register specified by <Xs> is set to `0xFFFF_FFFF_FFFF_FFFF`.

Regardless of the memory type:

- The memory access generated by an ST64BV or ST64BV0 instruction is not merged with any accesses.
- The memory access generated by an ST64B instruction is not merged with any accesses generated by store instructions appearing in program order after the instruction.

Table C3-28 shows the single-copy atomic 64-byte load/store instructions.

Table C3-28 Single-copy atomic 64-byte load/store instructions

Mnemonic	Instruction	See
`LD64B`	Single-copy atomic 64-byte load	LD64B
`ST64B`	Single-copy atomic 64-byte store without status result	ST64B

Table C3-29 shows the single-copy atomic 64-byte store with status result.

Table C3-29 Single-copy atomic 64-byte store with status result instructions

Mnemonic	Instruction	See
`ST64BV`	Single-copy atomic 64-byte store with status result	ST64BV

Table C3-30 shows the single-copy atomic 64-byte EL0 store with status result.

Table C3-30 Single-copy atomic 64-byte EL0 store with status result instructions

Mnemonic	Instruction	See
`ST64BV0`	Single-copy atomic 64-byte EL0 store with status result	ST64BV0

C3.2.9 Load-Exclusive/Store-Exclusive

The Load-Exclusive/Store-Exclusive instructions support only one addressing mode:

- Base register with no offset.

See Load/store addressing modes.

The Load-Exclusive instructions mark the physical address being accessed as an exclusive access. This exclusive access mark is checked by the Store-Exclusive instruction, permitting the construction of atomic read-modify-write operations on shared memory variables, semaphores, mutexes, and spinlocks. See Synchronization and semaphores.

If FEAT_LSE2 is not implemented then:.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Load-Exclusive/Store-Exclusive instructions other than Load-Exclusive pair and Store-Exclusive pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Exclusive pair or Store-Exclusive pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

  For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

When a Store-Exclusive pair succeeds, it causes a single-copy atomic update of the entire memory location being stored to.

Table C3-31 shows the Load-Exclusive/Store-Exclusive instructions.

Table C3-31 Load-Exclusive/Store-Exclusive instructions

Mnemonic	Instruction	See
`LDXR`	Load Exclusive register	LDXR
`LDXRB`	Load Exclusive byte	LDXRB
`LDXRH`	Load Exclusive halfword	LDXRH
`LDXP`	Load Exclusive pair	LDXP
`STXR`	Store Exclusive register	STXR
`STXRB`	Store Exclusive byte	STXRB
`STXRH`	Store Exclusive halfword	STXRH
`STXP`	Store Exclusive pair	STXP

C3.2.10 Load-Exclusive/Store-Exclusive unprivileged

The Load-Exclusive unprivileged and Store-Exclusive unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The Load-Exclusive/Store-Exclusive unprivileged instructions support only one address mode:

- Base register with no offset.

See Load/store addressing modes.

Table C3-32 Load-exclusive/Store-exclusive unprivileged instructions

Mnemonic	Instruction	See
`LDTXR`	Load unprivileged exclusive register	LDTXR
`STTXR`	Store unprivileged exclusive register	STTXR

C3.2.11 Load-Acquire/Store-Release

The Load-Acquire, Load-AcquirePC and Store-Release instructions are added as part of the FEAT_LRCPC, FEAT_LRCPC2, and FEAT_LRCPC3 architecture features.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_LSE2 is not implemented, then:

- The Load-Acquire, Load-AcquirePC, and Store-Release instructions other than Load-Acquire pair and Store-Release pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Acquire pair or Store-Release pair instructions must be aligned to the size of the accessed element in the pair, otherwise the access generates an Alignment fault.

For more information on alignment requirements and behaviors, see Non-atomic Load-Acquire/Store-Release instructions.

The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of Load-Acquire, Load-AcquirePC, and Store-Release, see Load-Acquire, Load-AcquirePC, and Store-Release.

A Store-Release Exclusive instruction has the Release semantics only if the store is successful.

Armv8.1 adds more instructions with load-acquire and store-release mechanisms, see LoadLOAcquire/StoreLORelease.

Table C3-33 shows the Non-exclusive Load-Acquire/Store-Release instructions.

Table C3-33 Non-exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAPR	Load-Acquire RCpc Register	LDAPR
LDAPRB	Load-Acquire RCpc Register Byte	LDAPRB
LDAPRH	Load-Acquire RCpc Register Halfword	LDAPRH
LDAPUR	Load-Acquire RCpc Register (unscaled)	LDAPUR
LDAPURB	Load-Acquire RCpc Register Byte (unscaled)	LDAPURB
LDAPURH	Load-Acquire RCpc Register Halfword (unscaled)	LDAPURH
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 32-bit	LDAPURSB
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 64-bit	LDAPURSB
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 32-bit	LDAPURSH
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 64-bit	LDAPURSH
LDAPURSW	Load-Acquire RCpc Register Signed Word (unscaled)	LDAPURSW
LDAR	Load-Acquire Register	LDAR
LDARB	Load-Acquire Byte	LDARB
LDARH	Load-Acquire Halfword	LDARH
STLR	Store-Release Register	STLR
STLRB	Store-Release Byte	STLRB
STLRH	Store-Release Halfword	STLRH
STLUR	Store-Release Register (unscaled)	STLUR
STLURB	Store-Release Register Byte (unscaled)	STLURB
STLURH	Store-Release Register Halfword (unscaled)	STLURH

Table C3-34 shows the Exclusive Load-Acquire/Store-Release instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-34 Exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAXR	Load-Acquire Exclusive register	LDAXR
LDAXRB	Load-Acquire Exclusive byte	LDAXRB
LDAXRH	Load-Acquire Exclusive halfword	LDAXRH
LDAXP	Load-Acquire Exclusive pair	LDAXP
STLXR	Store-Release Exclusive register	STLXR
STLXRB	Store-Release Exclusive byte	STLXRB
STLXRH	Store-Release Exclusive halfword	STLXRH
STLXP	Store-Release Exclusive pair	STLXP

Table C3-35 Register Pair Ordered Load-AcquirePC and Store-Release instructions

Mnemonic	Instruction	See
LDIAPP	Load-Acquire RCpc Ordered Pair of Registers	LDIAPP
STILP	Store-Release Pair Ordered	STILP

C3.2.12 Load-Acquire/Store-Release unprivileged

The Load-Acquire unprivileged and Store-Release unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load-acquire/store-release unprivileged instructions support only one addressing mode:

- Base register with no offset.

See Load/store addressing modes.

Table C3-36 Load-Acquire/Store-Release unprivileged instructions

Mnemonic	Instruction	See
LDATXR	Load-acquire unprivileged exclusive register	LDATXR
STLTXR	Store-release unprivileged exclusive register	STLTXR

C3.2.13 LoadLOAcquire/StoreLORelease

The LoadLOAcquire/StoreLORelease instructions support only one addressing mode:

- Base register with no offset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See Load/store addressing modes.

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of LoadLOAcquire/StoreLORelease, see LoadLOAcquire, StoreLORelease.

If FEAT_LSE2 is not implemented, then the LoadLOAcquire/StoreLORelease instructions require natural alignment, and an unaligned address generates an Alignment fault.

Table C3-37 shows the LoadLOAcquire/StoreLORelease instructions.

Table C3-37 LoadLOAcquire and StoreLORelease instructions

Mnemonic	Instruction	See
`LDLARB`	LoadLOAcquire byte	LDLARB
`LDLARH`	LoadLOAcquire halfword	LDLARH
`LDLAR`	LoadLOAcquire register	LDLAR
`STLLRB`	StoreLORelease byte	STLLRB
`STLLRH`	StoreLORelease halfword	STLLRH
`STLLR`	StoreLORelease register	STLLR

C3.2.14 Load/store scalar SIMD and floating-point

The load/store scalar SIMD and floating-point instructions operate on scalar values in the SIMD and floating-point register file as described in Advanced SIMD and floating-point scalar register names. The memory addressing modes available, described in Load/store addressing modes, are identical to the general-purpose register load/store instructions, and like those instructions permit arbitrary address alignment unless strict alignment checking is enabled. However, unlike the load/store instructions that transfer general-purpose registers, load/store scalar SIMD and floating-point instructions make no guarantee of atomicity, even when the address is naturally aligned to the size of the data.

C3.2.14.1 Load/store scalar SIMD and floating-point register

The load/store scalar SIMD and floating-point register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus unscaled 9-bit signed immediate offset.
- Base plus 64-bit register offset, optionally scaled.
- Base plus 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

For more information on the addressing modes, see Load/store addressing modes.

Note

The unscaled 9-bit signed immediate offset address mode requires its own instruction form, see Load/store scalar SIMD and floating-point register (unscaled offset).

Table C3-38 shows the load/store instructions for a single SIMD and floating-point register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-38 Load/store single SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDR	Load scalar SIMD&FP register (register offset)	LDR (register, SIMD&FP)
	Load scalar SIMD&FP register (immediate offset)	LDR (immediate, SIMD&FP)
	Load scalar SIMD&FP register (PC-relative literal)	LDR (literal, SIMD&FP)
STR	Store scalar SIMD&FP register (register offset)	STR (register, SIMD&FP)
	Store scalar SIMD&FP register (immediate offset)	STR (immediate, SIMD&FP)

C3.2.14.2 Load/store scalar SIMD and floating-point register (unscaled offset)

The load /store scalar SIMD and floating-point register instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See also Load/store addressing modes.

The load/store scalar SIMD and floating-point register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store single SIMD and floating-point instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset. This is similar to the load/store register (unscaled offset) instructions, that disambiguate this instruction class from the load/store register instruction, see Load/store register (unscaled offset).

Table C3-39 shows the load/store SIMD and floating-point register instructions with an unscaled offset.

Table C3-39 Load/store SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDUR	Load scalar SIMD&FP register (unscaled offset)	LDUR (SIMD&FP)
STUR	Store scalar SIMD&FP register (unscaled offset)	STUR (SIMD&FP)

C3.2.14.3 Load/store SIMD and floating-point register pair

The load/store SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

If a Load pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register being loaded takes an UNKNOWN value.

For behavior when FEAT_LS64WB is implemented, see Requirements for single-copy atomicity.

Table C3-40 shows the load/store SIMD and floating-point register pair instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-40 Load/store SIMD and floating-point register pair instructions

Mnemonic	Instruction	See
LDP	Load pair of scalar SIMD&FP registers	LDP (SIMD&FP)
STP	Store pair of scalar SIMD&FP registers	STP (SIMD&FP)

C3.2.14.4 Load/store unprivileged SIMD and floating-point register pair

The load/store unprivileged SIMD and floating-point register pair instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store unprivileged SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

Table C3-41 Load/store unprivileged SIMD and floating-point register pair instructions

Mnemonic	Instruction	See
LDTP	Load unprivileged pair of scalar SIMD&FP registers	LDTP (SIMD&FP)
STTP	Store unprivileged pair of scalar SIMD&FP registers	STTP (SIMD&FP)

C3.2.14.5 Load/store SIMD and floating-point non-temporal pair

The load/store SIMD and floating-point non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be prefetched and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a load non-temporal pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a load non-temporal pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-42 shows the load/store SIMD and floating-point Non-temporal pair instructions.

Table C3-42 Load/store SIMD and floating-point non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load pair of scalar SIMD&FP registers	LDNP (SIMD&FP)
STNP	Store pair of scalar SIMD&FP registers	STNP (SIMD&FP)

C3.2.14.6 Load/store unprivileged SIMD and floating-point non-temporal pair

The load/store unprivileged SIMD and floating-point non-temporal pair instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The load/store SIMD and floating-point non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

Table C3-43 Load/store unprivileged SIMD and floating-point non-temporal pair instructions

Mnemonic	Instruction	See
LDTNP	Load Unprivileged Non-temporal Pair	LDTNP (SIMD&FP)
STTNP	Store Unprivileged Non-temporal Pair	STTNP (SIMD&FP)

C3.2.15 Load/store Advanced SIMD

The Advanced SIMD load/store structure instructions support the following addressing modes:

- Base register only.
- Post-indexed by a 64-bit register.
- Post-indexed by an immediate, equal to the number of bytes transferred.

Load/store vector instructions, like other load/store instructions, allow any address alignment, unless strict alignment checking is enabled. If strict alignment checking is enabled, then alignment checking to the size of the element is performed. However, unlike the load/store instructions that transfer general-purpose registers, the load/store vector instructions do not guarantee atomicity, even when the address is naturally aligned to the size of the element.

C3.2.15.1 Load/store structures

Table C3-44 shows the load/store structure instructions. A post-increment immediate offset, if present, must be 8, 16, 24, 32, 48, or 64, depending on the number of elements transferred.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-44 Load/store multiple structures instructions

Mnemonic	Instruction	See
LD1	Load single 1-element structure to one lane of one register	LD1 (single structure)
	Load multiple 1-element structures to one register or to two, three, or four consecutive registers	LD1 (multiple structures)
LD2	Load single 2-element structure to one lane of two consecutive registers	LD2 (single structure)
	Load multiple 2-element structures to two consecutive registers	LD2 (multiple structures)
LD3	Load single 3-element structure to one lane of three consecutive registers	LD3 (single structure)
	Load multiple 3-element structures to three consecutive registers	LD3 (multiple structures)
LD4	Load single 4-element structure to one lane of four consecutive registers	LD4 (single structure)
	Load multiple 4-element structures to four consecutive registers	LD4 (multiple structures)
ST1	Store single 1-element structure from one lane of one register	ST1 (single structure)
	Store multiple 1-element structures from one register, or from two, three, or four consecutive registers	ST1 (multiple structures)
ST2	Store single 2-element structure from one lane of two consecutive registers	ST2 (single structure)
	Store multiple 2-element structures from two consecutive registers	ST2 (multiple structures)
ST3	Store single 3-element structure from one lane of three consecutive registers	ST3 (single structure)
	Store multiple 3-element structures from three consecutive registers	ST3 (multiple structures)
ST4	Store single 4-element structure from one lane of four consecutive registers	ST4 (single structure)
	Store multiple 4-element structures from four consecutive registers	ST4 (multiple structures)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.15.2 Load single structure and replicate

Table C3-45 shows the Load single structure and replicate instructions. A post-increment immediate offset, if present, must be 1, 2, 3, 4, 6, 8, 12, 16, 24, or 32, depending on the number of elements transferred.

Table C3-45 Load single structure and replicate instructions

Mnemonic	Instruction	See
`LD1R`	Load single 1-element structure and replicate to all lanes of one register	LD1R
`LD2R`	Load single 2-element structure and replicate to all lanes of two registers	LD2R
`LD3R`	Load single 3-element structure and replicate to all lanes of three registers	LD3R
`LD4R`	Load single 4-element structure and replicate to all lanes of four registers	LD4R

Table C3-46 SIMD and floating-point Non-exclusive Load-AcquirePC and Store-Release instructions

Mnemonic	Instruction	See
`LDAP1 (SIMD&FP)`	Load-Acquire RCpc one single-element structure to one lane of one register	LDAP1 (SIMD&FP)
`LDAPUR (SIMD&FP)`	Load-Acquire RCpc SIMD&FP Register (unscaled offset)	LDAPUR (SIMD&FP)
`STL1 (SIMD&FP)`	Store-Release a single-element structure from one lane of one register	STL1 (SIMD&FP)
`STLUR (SIMD&FP)`	Store-Release SIMD&FP Register (unscaled offset)	STLUR (SIMD&FP)

C3.2.16 Prefetch memory

The Prefetch memory instructions support the following addressing modes:

- Base register with no offset.
- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset. This can be optionally scaled by 8-bits, for example `LSL`#3.
- Base plus a 32-bit extended register offset. This can be optionally scaled by 8-bits.
- PC-relative literal.

The Prefetch memory instructions signal to the memory system that memory accesses from a specified address or range of addresses are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when they do occur, such as prefetching the specified address or range of addresses into one or more caches. Because these signals are only hints, it is valid for the PE to treat any or all prefetch instructions as a `NOP`.

Because they are hints to the memory system, the operation of a prefetch memory instruction cannot cause a synchronous exception. However, a memory operation performed as a result of one of these memory system hints might in exceptional cases trigger an asynchronous event, and thereby influence the execution of the PE. An example of an asynchronous event that might be triggered is an SError exception.

A prefetch memory instruction can only have an effect on software visible structures as defined in the translation regime of the current Exception level, such as caches and translation lookaside buffers associated with memory locations that can be accessed by reads, writes, or execution.

A `PRFM`, `PRFUM`, or `RPRFM` instruction is guaranteed not to access Device memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A PRFM or PRFUM instruction using a PLI hint cannot result in any access that could not be performed by the PE speculatively fetching an instruction. Therefore, if all associated MMUs are disabled, a PLI hint can only access memory locations consistent with instruction fetches permitted by The effects of disabling an address translation stage.

Arm recommends that when a PRFM or a PRFUM instruction describes a particular cache level, then the memory location described at that cache level should be made available at that same cache level, even if the line had previously been at a level of the cache that is closer to the PE than the specified cache level.

Note

For a PRFM PST instruction, this might result in the demotion of a cache line from a level that was closer to the cache.

Table C3-47 shows the Prefetch memory instructions.

Table C3-47 Prefetch memory instructions

Mnemonic	Instruction	See
PRFM	Prefetch memory (register offset)	PRFM (register)
	Prefetch memory (immediate offset)	PRFM (immediate)
	Prefetch memory (PC-relative offset)	PRFM (literal)
PRFUM	Prefetch memory (unscaled offset)	PRFUM
RPRFM	Range prefetch memory	RPRFM

C3.2.17 Atomic instructions

The atomic instructions perform atomic read and write operations on a memory location such that the architecture guarantees that no modification of that memory location by another observer can occur between the read and the write defined by that instruction.

This section describes the following operations:

- Atomic integer memory operations.
- Atomic floating-point memory operations.
- Unprivileged atomic memory operations.
- Swap operations.
- Swap unprivileged.
- Compare and Swap operations.
- Compare and Swap unprivileged.
- Read-Check-Write.

The atomic instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

For the purpose of permission checking, and for watchpoints, all of the Atomic memory operation instructions are treated as performing both a load and a store.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity. These operations map to the acquire and release definitions, and are counted as Load-Acquire and Store-Release operations respectively.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.17.1 Atomic integer memory operations

The address accessed by the 128-bit atomic instructions must be aligned to 128-bits, otherwise the instructions generate an Alignment fault.

For the `LD<OP>` instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

The `ST<OP>` instructions, and `LD<OP>` instructions where the destination register is WZR or XZR, are not regarded as doing a read for the purpose of a `DMB LD` barrier.

For information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-48 Atomic integer memory operation instructions

Mnemonic	Instruction	See
LDADD	Atomic add	LDADD, LDADDA, LDADDAL, LDADDL
LDADDB	Atomic add on byte	LDADDB, LDADDAB, LDADDALB, LDADDLB
LDADDH	Atomic add on halfword	LDADDH, LDADDAH, LDADDALH, LDADDLH
LDCLR	Atomic bit clear	LDCLR, LDCLRA, LDCLRAL, LDCLRL
LDCLRB	Atomic bit clear on byte	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB
LDCLRH	Atomic bit clear on halfword	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH
LDCLRP	Atomic bit clear on quadword	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL
LDEOR	Atomic exclusive-OR	LDEOR, LDEORA, LDEORAL, LDEORL
LDEORB	Atomic exclusive-OR on byte	LDEORB, LDEORAB, LDEORALB, LDEORLB
LDEORH	Atomic exclusive-OR on halfword	LDEORH, LDEORAH, LDEORALH, LDEORLH
LDSET	Atomic bit set	LDSET, LDSETA, LDSETAL, LDSETL
LDSETB	Atomic bit set on byte	LDSETB, LDSETAB, LDSETALB, LDSETLB
LDSETH	Atomic bit set on halfword	LDSETH, LDSETAH, LDSETALH, LDSETLH
LDSETP	Atomic bit set on quadword	LDSETP, LDSETPA, LDSETPAL, LDSETPL
LDMAX	Atomic signed maximum	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL
LDMAXB	Atomic signed maximum on byte	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB
LDMAXH	Atomic signed maximum on halfword	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH
LDMIN	Atomic signed minimum	LDSMIN, LDSMINA, LDSMINAL, LDSMINL
LDMINB	Atomic signed minimum on byte	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB
LDMINH	Atomic signed minimum on halfword	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH
LDUMAX	Atomic unsigned maximum	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL
LDUMAXB	Atomic unsigned maximum on byte	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB
LDUMAXH	Atomic unsigned maximum on halfword	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH
LDUMIN	Atomic unsigned minimum	LDUMIN, LDUMINA, LDUMINAL, LDUMINL
LDUMINB	Atomic unsigned minimum on byte	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB
LDUMINH	Atomic unsigned minimum on halfword	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
STADD	Atomic add, without return	STADD, STADDL
STADDB	Atomic add on byte, without return	STADDB, STADDLB
STADDH	Atomic add on halfword, without return	STADDH, STADDLH
STCLR	Atomic bit clear, without return	STCLR, STCLRL
STCLRB	Atomic bit clear on byte, without return	STCLRB, STCLRLB
STCLRH	Atomic bit clear on halfword, without return	STCLRH, STCLRLH
STEOR	Atomic exclusive-OR, without return	STEOR, STEORL
STEORB	Atomic exclusive-OR on byte, without return	STEORB, STEORLB
STEORH	Atomic exclusive-OR on halfword, without return	STEORH, STEORLH
STSET	Atomic bit set, without return	STSET, STSETL
STSETB	Atomic bit set on byte, without return	STSETB, STSETLB
STSETH	Atomic bit set on halfword, without return	STSETH, STSETLH
STMAX	Atomic signed maximum, without return	STSMAX, STSMAXL
STMAXB	Atomic signed maximum on byte, without return	STSMAXB, STSMAXLB
STMAXH	Atomic signed maximum on halfword, without return	STSMAXH, STSMAXLH
STMIN	Atomic signed minimum, without return	STSMIN, STSMINL
STMINB	Atomic signed minimum on byte, without return	STSMINB, STSMINLB
STMINH	Atomic signed minimum on halfword, without return	STSMINH, STSMINLH
STUMAX	Atomic unsigned maximum, without return	STUMAX, STUMAXL
STUMAXB	Atomic unsigned maximum on byte, without return	STUMAXB, STUMAXLB
STUMAXH	Atomic unsigned maximum on halfword, without return	STUMAXH, STUMAXLH
STUMIN	Atomic unsigned minimum, without return	STUMIN, STUMINL
STUMINB	Atomic unsigned minimum on byte, without return	STUMINB, STUMINLB
STUMINH	Atomic unsigned minimum on halfword, without return	STUMINH, STUMINLH

C3.2.17.2 Atomic floating-point memory operations

The atomic floating-point memory operation instructions atomically load a value from memory, perform a floating-point arithmetic operation, and store the result back to memory.

For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-49 Atomic floating-point memory instructions

Mnemonic	Instruction	See
LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL	Atomic BFloat16 add	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL
LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL	Atomic BFloat16 maximum	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL
LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML	Atomic BFloat16 maximum number	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL	Atomic BFloat16 minimum	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL
LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML	Atomic BFloat16 minimum number	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML
LDFADD, LDFADDA, LDFADDAL, LDFADDL	Atomic floating-point add	LDFADD, LDFADDA, LDFADDAL, LDFADDL
LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL	Atomic floating-point maximum	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL
LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML	Atomic floating-point maximum number	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML
LDFMIN, LDFMINA, LDFMINAL, LDFMINL	Atomic floating-point minimum	LDFMIN, LDFMINA, LDFMINAL, LDFMINL
LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML	Atomic floating-point minimum number	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML
STBFADD, STBFADDL	Atomic BFloat16 add, without return	STBFADD, STBFADDL
STBFMAX, STBFMAXL	Atomic BFloat16 maximum, without return	STBFMAX, STBFMAXL
STBFMAXNM, STBFMAXNML	Atomic BFloat16 maximum number, without return	STBFMAXNM, STBFMAXNML
STBFMIN, STBFMINL	Atomic BFloat16 minimum, without return	STBFMIN, STBFMINL
STBFMINNM, STBFMINNML	Atomic BFloat16 minimum number, without return	STBFMINNM, STBFMINNML
STFADD, STFADDL	Atomic floating-point add, without return	STFADD, STFADDL
STFMAX, STFMAXL	Atomic floating-point maximum, without return	STFMAX, STFMAXL
STFMAXNM, STFMAXNML	Atomic floating-point maximum number, without return	STFMAXNM, STFMAXNML
STFMIN, STFMINL	Atomic floating-point minimum, without return	STFMIN, STFMINL
STFMINNM, STFMINNML	Atomic floating-point minimum number, without return	STFMINNM, STFMINNML

C3.2.17.3 Unprivileged atomic memory operations

The unprivileged atomic memory operation instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The atomic memory operation instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-50 Unprivileged atomic memory operation instructions

Mnemonic	Instruction	See
LDTADD	Atomic add unprivileged	LDTADD, LDTADDA, LDTADDAL, LDTADDL
STTADD	Atomic add unprivileged, without return	STTADD, STTADDL
LDTSET	Atomic bit set unprivileged	LDTSET, LDTSETA, LDTSETAL, LDTSETL
STTSET	Atomic bit set unprivileged, without return	STTSET, STTSETL
LDTCLR	Atomic bit clear unprivileged	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL
STTCLR	Atomic bit clear unprivileged, without return	STTCLR, STTCLRL

C3.2.17.4 Swap operations

For the SWP instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

If the destination register Rt is WZR or XZR, this is not regarded as doing a read for the purpose of a DMB LD barrier.

For information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-51 Swap instructions

Mnemonic	Instruction	See
SWP	Swap	SWP, SWPA, SWPAL, SWPL
SWPB	Swap byte	SWPB, SWPAB, SWPALB, SWPLB
SWPH	Swap halfword	SWPH, SWPAH, SWPALH, SWPLH
SWPP	Swap quadword	SWPP, SWPPA, SWPPAL, SWPPL

C3.2.17.4.1 Swap unprivileged

The swap unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The swap unprivileged instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

Table C3-52 Swap unprivileged instructions

Mnemonic	Instruction	See
SWPT	Swap	SWPT, SWPTA, SWPTAL, SWPTL

C3.2.17.5 Compare and Swap operations

If FEAT_LSE2 is not implemented then:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CAS instructions require natural alignment.
- The CASP instructions require alignment to the total size of the memory being accessed.

If a compare and swap instruction does not perform a store, then the instruction does not have release semantics, regardless of the instruction ordering options.

For the CAS and CASP instructions, the architecture permits that a data read clears any Exclusives monitors associated with that location, even if the compare subsequently fails. If these instructions generate a synchronous Data Abort, the registers which are compared and loaded are restored to the values held in the registers before the instruction was executed.

If the destination register Rs is WZR or XZR, this is not regarded as doing a read for the purpose of a DMB LD barrier.

For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-53 Compare and swap instructions

Mnemonic	Instruction	See
CAS	Compare and swap	CAS, CASA, CASAL, CASL
CASB	Compare and swap byte	CASB, CASAB, CASALB, CASLB
CASH	Compare and swap halfword	CASH, CASAH, CASALH, CASLH
CASP	Compare and swap pair	CASP, CASPA, CASPAL, CASPL

C3.2.17.5.1 Compare and Swap unprivileged

The Compare and Swap unprivileged instructions are added as part of the FEAT_LSUI architecture feature. These instructions belong to the class of unprivileged memory access instructions. The privileges of the Explicit Memory Effects generated by these instructions are controlled by PSTATE.UAO. For more information, see PSTATE.UAO.

The Compare and Swap unprivileged instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

Table C3-54 Compare and swap unprivileged instructions

Mnemonic	Instruction	See
CAST	Compare and swap unprivileged	CAST, CASAT, CASALT, CASLT
CASPT	Compare and swap pair unprivileged	CASPT, CASPAT, CASPALT, CASPLT

C3.2.17.6 Read-Check-Write

If FEAT_THE is implemented, then the Read-Check-Write instructions, RCW, and Read-Check-Write Software instructions, RCWS, are provided to conditionally update stage 1 descriptors.

RCW and RCWS instructions update PSTATE.{N, Z, C, V} to {0, 0, 1, 0}, with all of the following exceptions:

- For the compare and swap variants of the RCW and RCWS instructions, if the compare fails, then PSTATE.N is set to 1.
- Otherwise, all of the following apply:
  - — If RCW and RCWS instructions fail the RCW checks, then PSTATE.Z is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If RCWS instructions fail the RCWS checks, then PSTATE.C is set to 0.

If FEAT_D128 is implemented, then RCW and RCWS instructions that operate on 128 bits of data are provided.

If the address is not aligned to the data transfer size, then RCW and RCWS instructions generate an Alignment fault.

The following table shows the RCW and RCWS instructions that operate on 64 bits of data:

Table C3-55 64 bit Read-Check-Write instructions

Mnemonic	Instruction	See
RCWCAS	Read Check Write Compare and Swap doubleword	RCWCAS, RCWCASA, RCWCASL, RCWCASAL
RCWSCAS	Read Check Write Software Compare and Swap doubleword	RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL
RCWCLR	Read Check Write Atomic bit clear on doubleword	RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL
RCWSCLR	Read Check Write Software Atomic bit clear on doubleword	RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL
RCWSET	Read Check Write Atomic bit set on doubleword	RCWSET, RCWSETA, RCWSETL, RCWSETAL
RCWSWP	Read Check Write Swap doubleword	RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL
RCWSSWP	Read Check Write Software Swap doubleword	RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL

The following table shows the RCW and RCWS instructions that operate on 128 bits of data:

Table C3-56 128 bit Read-Check-Write instructions

Mnemonic	Instruction	See
RCWCASP	Read Check Write Compare and Swap quadword	RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL
RCWSCASP	Read Check Write Software Compare and Swap quadword	RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL
RCWCLRP	Read Check Write Atomic bit clear on quadword	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL
RCWSCLRP	Read Check Write Software Atomic bit clear on quadword	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL
RCWSETP	Read Check Write Atomic bit set on quadword	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL
RCWSWPP	Read Check Write Swap quadword	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL
RCWSSWPP	Read Check Write Software Swap quadword	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL

C3.2.18 Memory Tagging instructions

C3.2.18.1 Tag load

Table C3-57 shows the Memory Tagging Extension Tag load instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-57 Tag load instructions

Mnemonic	Instruction	See
LDG	Load Allocation Tag	LDG

C3.2.18.2 Tag store

Table C3-58 shows the Memory Tagging Extension Tag store instructions.

Also see DC GVA.

Table C3-58 Tag store instructions

Mnemonic	Instruction	See
STG	Store Allocation Tag	STG
ST2G	Store Allocation Tags	ST2G

C3.2.18.3 Tag and Data store

Table C3-59 shows the Memory Tagging Extension Tag and Data store instructions.

Also see DC GZVA and Table C3-63.

Table C3-59 Tag and Data store instructions

Mnemonic	Instruction	See
STZG	Store Allocation Tag, zeroing	STZG
STZ2G	Store Allocation Tags, zeroing	STZ2G
STGP	Store Allocation Tag and pair of registers	STGP

C3.2.18.4 Bulk Allocation tag access

Table C3-60 shows the Memory Tagging Extension Bulk Allocation tag access instructions.

Table C3-60 Bulk Allocation tag access instructions

Mnemonic	Instruction	See
LDGM	Load tag multiple	LDGM
STGM	Store Allocation Tag multiple	STGM
STZGM	Store Allocation Tag and zero multiple	STZGM

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.19 Memory Copy and Memory Set instructions

If FEAT_MOPS is implemented, the `CPY*`, `SETE*`, `SETM*`, and `SETP*` instructions described in this section are implemented. If FEAT_MOPS and FEAT_MTE are implemented, the `SETG*` instructions described in this section are also implemented. Collectively, the instructions described in this section are referred to as *Memory Copy and Memory Set* instructions.

To perform a memory copy or memory set, three instructions - a prologue, then a main, and then an epilogue - are expected to be run in succession, with no instruction appearing in the code between them:

- To perform a memory copy, forward-only: `CPYFP*`, then `CPYFM*`, and then `CPYFE*`.
- To perform a memory copy, forward or backward: `CPYP*`, then `CPYM*`, and then `CPYE*`.
- To perform a memory set with tag setting: `SETGP*`, then `SETGM*`, and then `SETGE*`.
- To perform a memory set without tag setting: `SETP*`, then `SETM*`, and then `SETE*`.

The variant of each instruction is also expected to be the same throughout one of these sequences. For example, `CPYFPWTWN`, `CPYFMWTWN`, `CPYFEWTWN`.

Note

The Memory Copy and Memory Set instructions are expected to be the preferred approach for compilation for performance for any of the following cases:

- The size or alignments of the copy or set operation are not known at compile time.
- The size or alignments of the copy or set operation are known at compile time, but not amenable to the operation being performed using load and store instructions without looping.

Arm recommends that hardware implementations optimise the performance of these cases.

Fetching of a Memory Copy and Memory Set instruction multiple times during its execution is permissible.

The handling of synchronous exceptions generated by Memory Copy and Memory Set instructions is described in ESR_ELx, FAR_ELx, Definition of a precise exception and imprecise exception, and Synchronous exception entry.

The handling of asynchronous exceptions generated by Memory Copy and Memory Set instructions is described in Definition of a precise exception and imprecise exception and Asynchronous exception entry.

Memory Copy and Memory Set exceptions can be generated by execution of Memory Copy and Memory Set instructions restarting on a physical hardware PE implementation that is different from the physical hardware PE implementation that an exception was taken from. For more information, see Memory Copy and Memory Set exceptions.

If an exception is taken during the execution of a Memory Copy and Memory Set instruction, that instruction has not been executed for the purposes of instruction counting, instruction tracing, statistical profiling, or single stepping.

For the purposes of single stepping and performance monitoring, each Memory Copy and Memory Set instruction is regarded as a single instruction that performs a store and, in the case of `CPY*` instructions, also performs a load.

For Memory Copy and Memory Set instructions, the following are not architecturally defined:

- The size of the memory transactions they create.
- The order between accesses to different addresses.

The `CPY*` instructions are guaranteed to make forward progress if none of the following leaf level translation table entries fault:

- The source leaf level translation table entry used to translate the address encoded in `Xs`.
- If the range of remaining source addresses, encoded in `Xs` and `Xn`, crosses a page boundary:
  - The next leaf level translation table entry, as determined by the copy direction, adjacent to the source leaf level translation table entry used to translate the address encoded in `Xs`.
- The destination leaf level translation table entry used to translate the address encoded in `Xd`.
- If the range of remaining destination addresses, encoded in `Xd` and `Xn`, crosses a page boundary:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The next leaf level translation table entry, as determined by the copy direction, adjacent to the destination leaf level translation table entry used to translate the address encoded in `Xd`.

The `SET*` instructions are guaranteed to make forward progress if none of the following leaf level translation table entries fault:

- The destination leaf level translation table entry used to translate the address encoded in `Xd`.
- If the range of remaining addresses, encoded in `Xd` and `Xn`, crosses a page boundary:
  — The next leaf level translation table entry adjacent to the destination leaf level translation table entry used to translate the address encoded in `Xd`.

Note

For a `CPY*` instruction, destination address, source address and copy size are encoded in `Xd`, `Xs` and `Xn` depending on the Option and Direction chosen by the implementation.
For a `SET*` instruction, destination address and set size are encoded in `Xd` and `Xn` depending on the Option and Direction chosen by the implementation.
A PE can ensure that the forward progress described in this section can be achieved by ensuring that, when an MMU fault is encountered, all bytes leading up to the fault have been operated on and the operand registers have been updated.
A PE should not report MMU faults other than for the leaf level translation table entries described in this section.

Table C3-61 shows the memory copy, forward-only instructions.

Table C3-61 Memory copy, forward-only instructions

Mnemonic	Instruction	See
`CPYFE`	Memory Copy Forward-only Epilogue	CPYFP, CPYFM, CPYFE Epilogue variant
`CPYFEN`	Memory Copy Forward-only Epilogue, reads and writes non-temporal	CPYFPN, CPYFMN, CPYFEN Epilogue variant
`CPYFERN`	Memory Copy Forward-only Epilogue, reads non-temporal	CPYFPRN, CPYFMRN, CPYFERN Epilogue variant
`CPYFERT`	Memory Copy Forward-only Epilogue, reads unprivileged	CPYFPRT, CPYFMRT, CPYFERT Epilogue variant
`CPYFERTN`	Memory Copy Forward-only Epilogue, reads unprivileged, reads and writes non-temporal	CPYFPRTN, CPYFMRTN, CPYFERTN Epilogue variant
`CPYFERTRN`	Memory Copy Forward-only Epilogue, reads unprivileged and non-temporal	CPYFPRTRN, CPYFMRTRN, CPYFERTRN Epilogue variant
`CPYFERTWN`	Memory Copy Forward-only Epilogue, reads unprivileged, writes non-temporal	CPYFPRTWN, CPYFMRTWN, CPYFERTWN Epilogue variant
`CPYFET`	Memory Copy Forward-only Epilogue, reads and writes unprivileged	CPYFPT, CPYFMT, CPYFET Epilogue variant
`CPYFETN`	Memory Copy Forward-only Epilogue, reads and writes unprivileged and non-temporal	CPYFPTN, CPYFMTN, CPYFETN Epilogue variant
`CPYFETRN`	Memory Copy Forward-only Epilogue, reads and writes unprivileged, reads non-temporal	CPYFPTRN, CPYFMTRN, CPYFETRN Epilogue variant
`CPYFETWN`	Memory Copy Forward-only Epilogue, reads and writes unprivileged, writes non-temporal	CPYFPTWN, CPYFMTWN, CPYFETWN Epilogue variant
`CPYFEWN`	Memory Copy Forward-only Epilogue, writes non-temporal	CPYFPWN, CPYFMWN, CPYFEWN Epilogue variant
`CPYFEWT`	Memory Copy Forward-only Epilogue, writes unprivileged	CPYFPWT, CPYFMWT, CPYFEWT Epilogue variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYFEWTN	Memory Copy Forward-only Epilogue, writes unprivileged, reads and writes non-temporal	CPYFPWTN, CPYFMWTN, CPYFEWTN Epilogue variant
CPYFEWTRN	Memory Copy Forward-only Epilogue, writes unprivileged, reads non-temporal	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Epilogue variant
CPYFMWTWN	Memory Copy Forward-only Epilogue, writes unprivileged and non-temporal	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Epilogue variant
CPYFM	Memory Copy Forward-only Main	CPYFP, CPYFM, CPYFE Main variant
CPYFMN	Memory Copy Forward-only Main, reads and writes non-temporal	CPYFPN, CPYFMN, CPYFEN Main variant
CPYFMRN	Memory Copy Forward-only Main, reads non-temporal	CPYFPRN, CPYFMRN, CPYFERN Main variant
CPYFMRT	Memory Copy Forward-only Main, reads unprivileged	CPYFPRT, CPYFMRT, CPYFERT Main variant
CPYFMRTN	Memory Copy Forward-only Main, reads unprivileged, reads and writes non-temporal	CPYFPRTN, CPYFMRTN, CPYFERTN Main variant
CPYFMRTRN	Memory Copy Forward-only Main, reads unprivileged and non-temporal	CPYFPRTRN, CPYFMRTRN, CPYFERTRN Main variant
CPYFMRTWN	Memory Copy Forward-only Main, reads unprivileged, writes non-temporal	CPYFPRTWN, CPYFMRTWN, CPYFERTWN Main variant
CPYFMT	Memory Copy Forward-only Main, reads and writes unprivileged	CPYFPT, CPYFMT, CPYFET Main variant
CPYFMTN	Memory Copy Forward-only Main, reads and writes unprivileged and non-temporal	CPYFPTN, CPYFMTN, CPYFETN Main variant
CPYFMTRN	Memory Copy Forward-only Main, reads and writes unprivileged, reads non-temporal	CPYFPTRN, CPYFMTRN, CPYFETRN Main variant
CPYFMTWN	Memory Copy Forward-only Main, reads and writes unprivileged, writes non-temporal	CPYFPTWN, CPYFMTWN, CPYFETWN Main variant
CPYFMWN	Memory Copy Forward-only Main, writes non-temporal	CPYFPWN, CPYFMWN, CPYFEWN Main variant
CPYFMWT	Memory Copy Forward-only Main, writes unprivileged	CPYFPWT, CPYFMWT, CPYFEWT Main variant
CPYFMWTN	Memory Copy Forward-only Main, writes unprivileged, reads and writes non-temporal	CPYFPWTN, CPYFMWTN, CPYFEWTN Main variant
CPYFMWTRN	Memory Copy Forward-only Main, writes unprivileged, reads non-temporal	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Main variant
CPYFMWTWN	Memory Copy Forward-only Main, writes unprivileged and non-temporal	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Main variant
CPYFP	Memory Copy Forward-only Prologue	CPYFP, CPYFM, CPYFE Prologue variant
CPYFPN	Memory Copy Forward-only Prologue, reads and writes non-temporal	CPYFPN, CPYFMN, CPYFEN Prologue variant
CPYFPRN	Memory Copy Forward-only Prologue, reads non-temporal	CPYFPRN, CPYFMRN, CPYFERN Prologue variant
CPYFPRT	Memory Copy Forward-only Prologue, reads unprivileged	CPYFPRT, CPYFMRT, CPYFERT Prologue variant
CPYFPRTN	Memory Copy Forward-only Prologue, reads unprivileged, reads and writes non-temporal	CPYFPRTN, CPYFMRTN, CPYFERTN Prologue variant
CPYFPRTRN	Memory Copy Forward-only Prologue, reads unprivileged and non-temporal	CPYFPRTRN, CPYFMRTRN, CPYFERTRN Prologue variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYFPRTWN	Memory Copy Forward-only Prologue, reads unprivileged, writes non-temporal	CPYFPRTWN, CPYFMRTWN, CPYFERTWN Prologue variant
CPYFPT	Memory Copy Forward-only Prologue, reads and writes unprivileged	CPYFPT, CPYFMT, CPYFET Prologue variant
CPYFPTN	Memory Copy Forward-only Prologue, reads and writes unprivileged and non-temporal	CPYFPTN, CPYFMTN, CPYFETN Prologue variant
CPYFPTRN	Memory Copy Forward-only Prologue, reads and writes unprivileged, reads non-temporal	CPYFPTRN, CPYFMTRN, CPYFETRN Prologue variant
CPYFPTWN	Memory Copy Forward-only Prologue, reads and writes unprivileged, writes non-temporal	CPYFPTWN, CPYFMTWN, CPYFETWN Prologue variant
CPYFPWN	Memory Copy Forward-only Prologue, writes non-temporal	CPYFPWN, CPYFMWN, CPYFEWN Prologue variant
CPYFPWT	Memory Copy Forward-only Prologue, writes unprivileged	CPYFPWT, CPYFMWT, CPYFEWT Prologue variant
CPYFPWTN	Memory Copy Forward-only Prologue, writes unprivileged, reads and writes non-temporal	CPYFPWTN, CPYFMWTN, CPYFEWTN Prologue variant
CPYFPWTRN	Memory Copy Forward-only Prologue, writes unprivileged, reads non-temporal	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Prologue variant
CPYFPWTWN	Memory Copy Forward-only Prologue, writes unprivileged and non-temporal	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Prologue variant

Table C3-62 shows the memory copy, forward or backward instructions.

Table C3-62 Memory copy, forward or backward instructions

Mnemonic	Instruction	See
CPYE	Memory Copy Epilogue	CPYP, CPYM, CPYE Epilogue variant
CPYEN	Memory Copy Epilogue, reads and writes non-temporal	CPYPN, CPYMN, CPYEN Epilogue variant
CPYERN	Memory Copy Epilogue, reads non-temporal	CPYPRN, CPYMRN, CPYERN Epilogue variant
CPYERT	Memory Copy Epilogue, reads unprivileged	CPYPRT, CPYMRT, CPYERT Epilogue variant
CPYERTN	Memory Copy Epilogue, reads unprivileged, reads and writes non-temporal	CPYPRTN, CPYMRTN, CPYERTN Epilogue variant
CPYERTRN	Memory Copy Epilogue, reads unprivileged and non-temporal	CPYPRTRN, CPYMRTRN, CPYERTRN Epilogue variant
CPYERTWN	Memory Copy Epilogue, reads unprivileged, writes non-temporal	CPYPRTWN, CPYMRTWN, CPYERTWN Epilogue variant
CPYET	Memory Copy Epilogue, reads and writes unprivileged	CPYPT, CPYMT, CPYET Epilogue variant
CPYETN	Memory Copy Epilogue, reads and writes unprivileged and non-temporal	CPYPTN, CPYMTN, CPYETN Epilogue variant
CPYETRN	Memory Copy Epilogue, reads and writes unprivileged, reads non-temporal	CPYPTRN, CPYMTRN, CPYETRN Epilogue variant
CPYETWN	Memory Copy Epilogue, reads and writes unprivileged, writes non-temporal	CPYPTWN, CPYMTWN, CPYETWN Epilogue variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYEWN	Memory Copy Epilogue, writes non-temporal	CPYPWN, CPYMWN, CPYEWN Epilogue variant
CPYEWT	Memory Copy Epilogue, writes unprivileged	CPYPWT, CPYMWT, CPYEWT Epilogue variant
CPYEWTN	Memory Copy Epilogue, writes unprivileged, reads and writes non-temporal	CPYPWTN, CPYMWTN, CPYEWTN Epilogue variant
CPYEWTRN	Memory Copy Epilogue, writes unprivileged, reads non-temporal	CPYPWTRN, CPYMWTRN, CPYEWTRN Epilogue variant
CPYMWTWN	Memory Copy Epilogue, writes unprivileged and non-temporal	CPYPWTWN, CPYMWTWN, CPYEWTWN Epilogue variant
CPYM	Memory Copy Main	CPYP, CPYM, CPYE Main variant
CPYMN	Memory Copy Main, reads and writes non-temporal	CPYPN, CPYMN, CPYEN Main variant
CPYMRN	Memory Copy Main, reads non-temporal	CPYPRN, CPYMRN, CPYERN Main variant
CPYMRT	Memory Copy Main, reads unprivileged	CPYPRT, CPYMRT, CPYERT Main variant
CPYMRTN	Memory Copy Main, reads unprivileged, reads and writes non-temporal	CPYPRTN, CPYMRTN, CPYERTN Main variant
CPYMRTRN	Memory Copy Main, reads unprivileged and non-temporal	CPYPRTRN, CPYMRTRN, CPYERTRN Main variant
CPYMRTWN	Memory Copy Main, reads unprivileged, writes non-temporal	CPYPRTWN, CPYMRTWN, CPYERTWN Main variant
CPYMT	Memory Copy Main, reads and writes unprivileged	CPYPT, CPYMT, CPYET Main variant
CPYMTN	Memory Copy Main, reads and writes unprivileged and non-temporal	CPYPTN, CPYMTN, CPYETN Main variant
CPYMTRN	Memory Copy Main, reads and writes unprivileged, reads non-temporal	CPYPTRN, CPYMTRN, CPYETRN Main variant
CPYMTWN	Memory Copy Main, reads and writes unprivileged, writes non-temporal	CPYPTWN, CPYMTWN, CPYETWN Main variant
CPYMWN	Memory Copy Main, writes non-temporal	CPYPWN, CPYMWN, CPYEWN Main variant
CPYMWT	Memory Copy Main, writes unprivileged	CPYPWT, CPYMWT, CPYEWT Main variant
CPYMWTN	Memory Copy Main, writes unprivileged, reads and writes non-temporal	CPYPWTN, CPYMWTN, CPYEWTN Main variant
CPYMWTRN	Memory Copy Main, writes unprivileged, reads non-temporal	CPYPWTRN, CPYMWTRN, CPYEWTRN Main variant
CPYMWTWN	Memory Copy Main, writes unprivileged and non-temporal	CPYPWTWN, CPYMWTWN, CPYEWTWN Main variant
CPYP	Memory Copy Prologue	CPYP, CPYM, CPYE Prologue variant
CPYPN	Memory Copy Prologue, reads and writes non-temporal	CPYPN, CPYMN, CPYEN Prologue variant
CPYPRN	Memory Copy Prologue, reads non-temporal	CPYPRN, CPYMRN, CPYERN Prologue variant
CPYPRT	Memory Copy Prologue, reads unprivileged	CPYPRT, CPYMRT, CPYERT Prologue variant
CPYPRTN	Memory Copy Prologue, reads unprivileged, reads and writes non-temporal	CPYPRTN, CPYMRTN, CPYERTN Prologue variant
CPYPRTRN	Memory Copy Prologue, reads unprivileged and non-temporal	CPYPRTRN, CPYMRTRN, CPYERTRN Prologue variant
CPYPRTWN	Memory Copy Prologue, reads unprivileged, writes non-temporal	CPYPRTWN, CPYMRTWN, CPYERTWN Prologue variant
CPYPT	Memory Copy Prologue, reads and writes unprivileged	CPYPT, CPYMT, CPYET Prologue variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CPYPTN	Memory Copy Prologue, reads and writes unprivileged and non-temporal	CPYPTN, CPYMTN, CPYETN Prologue variant
CPYPTRN	Memory Copy Prologue, reads and writes unprivileged, reads non-temporal	CPYPTRN, CPYMTRN, CPYETRN Prologue variant
CPYPTWN	Memory Copy Prologue, reads and writes unprivileged, writes non-temporal	CPYPTWN, CPYMTWN, CPYETWN Prologue variant
CPYPWN	Memory Copy Prologue, writes non-temporal	CPYPWN, CPYMWN, CPYEWN Prologue variant
CPYPWT	Memory Copy Prologue, writes unprivileged	CPYPWT, CPYMWT, CPYEWT Prologue variant
CPYPWTN	Memory Copy Prologue, writes unprivileged, reads and writes non-temporal	CPYPWTN, CPYMWTN, CPYEWTN Prologue variant
CPYPWTRN	Memory Copy Prologue, writes unprivileged, reads non-temporal	CPYPWTRN, CPYMWTRN, CPYEWTRN Prologue variant
CPYPWTWN	Memory Copy Prologue, writes unprivileged and non-temporal	CPYPWTWN, CPYMWTWN, CPYEWTWN Prologue variant

Table C3-63 shows the memory set with tag setting instructions.

Table C3-63 Memory set with tag setting instructions

Mnemonic	Instruction	See
SETGE	Memory Set with tag setting Epilogue	SETGP, SETGM, SETGE Epilogue variant
SETGEN	Memory Set with tag setting Epilogue, non-temporal	SETGPN, SETGMN, SETGEN Epilogue variant
SETGET	Memory Set with tag setting Epilogue, unprivileged	SETGPT, SETGMT, SETGET Epilogue variant
SETGETN	Memory Set with tag setting Epilogue, unprivileged and non-temporal	SETGPTN, SETGMTN, SETGETN Epilogue variant
SETGM	Memory Set with tag setting Main	SETGP, SETGM, SETGE Main variant
SETGMN	Memory Set with tag setting Main, non-temporal	SETGPN, SETGMN, SETGEN Main variant
SETGMT	Memory Set with tag setting Main, unprivileged	SETGPT, SETGMT, SETGET Main variant
SETGMTN	Memory Set with tag setting Main, unprivileged and non-temporal	SETGPTN, SETGMTN, SETGETN Main variant
SETGP	Memory Set with tag setting Prologue	SETGP, SETGM, SETGE Prologue variant
SETGPN	Memory Set with tag setting Prologue, non-temporal	SETGPN, SETGMN, SETGEN Prologue variant
SETGPT	Memory Set with tag setting Prologue, unprivileged	SETGPT, SETGMT, SETGET Prologue variant
SETGPTN	Memory Set with tag setting Prologue, unprivileged and non-temporal	SETGPTN, SETGMTN, SETGETN Prologue variant

Table C3-64 shows the memory set without tag setting instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-64 Memory set without tag setting instructions

Mnemonic	Instruction	See
SETE	Memory Set Epilogue	SETP, SETM, SETE Epilogue variant
SETEN	Memory Set Epilogue, non-temporal	SETPN, SETMN, SETEN Epilogue variant
SETET	Memory Set Epilogue, unprivileged	SETPT, SETMT, SETET Epilogue variant
SETETN	Memory Set Epilogue, unprivileged and non-temporal	SETPTN, SETMTN, SETETN Epilogue variant
SETM	Memory Set Main	SETP, SETM, SETE Main variant
SETMN	Memory Set Main, non-temporal	SETPN, SETMN, SETEN Main variant
SETMT	Memory Set Main, unprivileged	SETPT, SETMT, SETET Main variant
SETMTN	Memory Set Main, unprivileged and non-temporal	SETPTN, SETMTN, SETETN Main variant
SETP	Memory Set Prologue	SETP, SETM, SETE Prologue variant
SETPN	Memory Set Prologue, non-temporal	SETPN, SETMN, SETEN Prologue variant
SETPT	Memory Set Prologue, unprivileged	SETPT, SETMT, SETET Prologue variant
SETPTN	Memory Set Prologue, unprivileged and non-temporal	SETPTN, SETMTN, SETETN Prologue variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.3 Loads and stores - SVE, SVE2

SVE vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate transfer registers. SVE also includes vector prefetch instructions that provide read and write hints to the memory system. For SVE predicated load, store, and prefetch instructions, the memory element access size and type that is associated with each vector element is specified by a suffix to the instruction mnemonic, independently of the element size of the transfer registers. For example, LD1SH. The following table shows the supported instruction suffixes for SVE load, store, and prefetch instructions:

Instruction suffix	Memory element access size and type
B	Unsigned byte
H	Unsigned halfword or half-precision floating-point
W	Unsigned word or single-precision floating-point
D	Unsigned doubleword or double-precision floating-point
SB	Signed byte
SH	Signed halfword
SW	Signed word

The element size of the transfer registers is always greater than or equal to the memory element access size. When the element size of the transfer registers is strictly greater than the memory element access size, then these are referred to as unpacked data accesses. In the case of unpacked data accesses:

- For load instructions, each element access is sign-extended or zero-extended to fill the vector element, according to its memory element access size and type.
- For store instructions, each vector element is truncated to the memory element access size.

Where the vector element size and the memory element access size are the same, then these are referred to as packed data accesses. Signed access types are not supported for packed data accesses. Packed and unpacked access sizes and types relate to the vector element size of the transfer registers, as defined in the following table:

Vector element	Packed access suffix	Unpacked access suffixes
.B	B	-
.H	H	B, SB
.S	W	H, SH, B, SB
.D	D	W, SW, H, SH, B, SB

For gather-load and scatter-store instructions, the vector element size can only be .S or .D. This means that any non-contiguous memory element access of less than a word is unpacked. Non-contiguous memory element accesses of a word can be either packed or unpacked, depending on the vector element size.

Load, store, and prefetch instructions consist of the following:

- Predicated single vector contiguous element accesses.
- Predicated multiple vector contiguous structure load/store.
- Predicated non-contiguous element accesses.
- Predicated replicating element loads.
- Unpredicated vector register load/store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Unpredicated predicate register load/store.
- Non-temporal gather/scatter.

All predicated load instructions zero the Inactive elements of the destination vector, except for Non-fault loads and First-fault loads when the corresponding FFR element is FALSE. When this happens, the content of the corresponding destination vector element is CONSTRAINED UNPREDICTABLE, as described in rule R_{NGFTJ}.

Prefetch instructions provide hints to hardware and do not change architectural state. Therefore, a *Governing predicate* for a prefetch instruction provides an additional hint which indicates the memory locations to be prefetched. Prefetch instructions require a prefetch operation specifier. SVE prefetch instructions support all of the prefetch operations except for the PLI prefetch operand types.

Load, store, and prefetch instructions that multiply a scalar index register or an index vector element by the memory element access size specify a shift type, followed by a shift amount in bits. The shift type can be one of `LSL`, `SXTW`, or `UXTW`. The shift amount is always Log_2 of the memory element access size, in bytes. The shift amount defaults to zero when the memory element access size is a byte, and the shift size can be omitted. The shift type of `LSL` must be omitted if the shift amount is omitted.

When included as part of the assembler syntax for an instruction, `MUL VL` indicates that the specified immediate index value is multiplied by the size of the addressed vector or predicate in memory, measured in bytes, irrespective of predication. For a detailed description of the meaning of this assembler syntax for each instruction, see the appropriate subsection below. When used in pseudocode, the symbol `VL` represents the vector length, measured in bits.

SVE load, store, and prefetch instructions do not support pre-indexed or post-indexed addressing.

C3.3.1 Predicated single vector contiguous element accesses

Predicated contiguous load and store instructions access memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate index value that is in the range -8 to 7, inclusive. This defaults to zero if omitted.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate index.
- Scalar base plus scalar index.

Predicated contiguous prefetch instructions address memory locations in a similar manner, with the index being either:

- A scalar index register.
- An immediate index value that is in the range of -32 to 31, inclusive. This defaults to zero if omitted.

For predicated contiguous load and store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The scalar index register value is multiplied by the memory element access size in bytes. The index value is incremented by one after each element access, but the scalar index register is not updated by the instruction.
- When alignment checking is enabled for loads and stores, the value of the base address register must be aligned to the memory element access size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-67 Predicated single vector contiguous element accesses

Mnemonic	Instruction	See
LD1	Contiguous load unsigned bytes to vector (scalar plus immediate, scalar plus scalar)	LD1B (scalar plus immediate, single register) LD1B (scalar plus scalar, single register)
	Contiguous load doublewords to vector (scalar plus immediate, scalar plus scalar)	LD1D (scalar plus immediate, single register) LD1D (scalar plus scalar, single register)
	Contiguous load unsigned halfwords to vector (scalar plus immediate, scalar plus scalar)	LD1H (scalar plus immediate, single register) LD1H (scalar plus scalar, single register)
	Contiguous load signed bytes to vector (scalar plus immediate)	LD1SB (scalar plus immediate)
	Contiguous load signed halfwords to vector (scalar plus immediate)	LD1SH (scalar plus immediate)
	Contiguous load signed words to vector (scalar plus immediate)	LD1SW (scalar plus immediate)
	Contiguous load unsigned words to vector (scalar plus immediate)	LD1W (scalar plus immediate, single register)
ST1	Contiguous store bytes from vector (scalar plus immediate, scalar plus scalar)	ST1B (scalar plus immediate, single register) ST1B (scalar plus scalar, single register)
	Contiguous store doublewords from vector (scalar plus immediate, scalar plus scalar)	ST1D (scalar plus immediate, single register) ST1D (scalar plus scalar, single register)
	Contiguous store halfwords from vector (scalar plus immediate, scalar plus scalar)	ST1H (scalar plus immediate, single register) ST1H (scalar plus scalar, single register)
	Contiguous store words from vector (scalar plus immediate, scalar plus scalar)	ST1W (scalar plus immediate, single register) ST1W (scalar plus scalar, single register)
LDFF1	Contiguous load first-fault unsigned bytes to vector (scalar plus scalar)	LDFF1B (scalar plus scalar)
	Contiguous load first-fault doublewords to vector (scalar plus scalar)	LDFF1D (scalar plus scalar)
	Contiguous load first-fault unsigned halfwords to vector (scalar plus scalar)	LDFF1H (scalar plus scalar)
	Contiguous load first-fault signed bytes to vector (scalar plus scalar)	LDFF1SB (scalar plus scalar)
	Contiguous load first-fault signed halfwords to vector (scalar plus scalar)	LDFF1SH (scalar plus scalar)
	Contiguous load first-fault signed words to vector (scalar plus scalar)	LDFF1SW (scalar plus scalar)
	Contiguous load first-fault unsigned words to vector (scalar plus scalar)	LDFF1W (scalar plus scalar)
LDNF1	Contiguous load non-fault unsigned bytes to vector	LDNF1B
	Contiguous load non-fault doublewords to vector	LDNF1D
	Contiguous load non-fault unsigned halfwords to vector	LDNF1H
	Contiguous load non-fault signed bytes to vector	LDNF1SB
	Contiguous load non-fault signed halfwords to vector	LDNF1SH
	Contiguous load non-fault signed words to vector	LDNF1SW
	Contiguous load non-fault unsigned words to vector	LDNF1W
LDNT1	Contiguous load non-temporal bytes to vector (scalar plus immediate, scalar plus scalar)	LDNT1B (scalar plus immediate, single register) LDNT1B (scalar plus scalar, single register)
	Contiguous load non-temporal doublewords to vector (scalar plus immediate, scalar plus scalar)	LDNT1D (scalar plus immediate, single register) LDNT1D (scalar plus scalar, single register)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous load non-temporal halfwords to vector (scalar plus immediate, scalar plus scalar)	LDNT1H (scalar plus immediate, single register) LDNT1H (scalar plus scalar, single register)
	Contiguous load non-temporal words to vector (scalar plus immediate, scalar plus scalar)	LDNT1W (scalar plus immediate, single register) LDNT1W (scalar plus scalar, single register)
STNT1	Contiguous store non-temporal bytes from vector (scalar plus immediate, scalar plus scalar)	STNT1B (scalar plus immediate, single register) STNT1B (scalar plus scalar, single register)
	Contiguous store non-temporal doublewords from vector (scalar plus immediate, scalar plus scalar)	STNT1D (scalar plus immediate, single register) STNT1D (scalar plus scalar, single register)
	Contiguous store non-temporal halfwords from vector (scalar plus immediate, scalar plus scalar)	STNT1H (scalar plus immediate, single register) STNT1H (scalar plus scalar, single register)
	Contiguous store non-temporal words from vector (scalar plus immediate, scalar plus scalar)	STNT1W (scalar plus immediate, single register) STNT1W (scalar plus scalar, single register)
PRF	Contiguous prefetch bytes (scalar plus immediate, scalar plus scalar)	PRFB (scalar plus immediate) PRFB (scalar plus scalar)
	Contiguous prefetch doublewords (scalar plus immediate, scalar plus scalar)	PRFD (scalar plus immediate) PRFD (scalar plus scalar)
	Contiguous prefetch halfwords (scalar plus immediate, scalar plus scalar)	PRFH (scalar plus immediate) PRFH (scalar plus scalar)
	Contiguous prefetch words (scalar plus immediate, scalar plus scalar)	PRFW (scalar plus immediate) PRFW (scalar plus scalar)

C3.3.2 Predicated multiple vector contiguous structure load/store

Predicated multiple vector contiguous structure load/store instructions are defined by a scalar base register plus either:

- A scalar index register.
- An immediate index that is a multiple of N, in the range -8xN to 7xN, inclusive. This defaults to zero if omitted.

The predicated contiguous structure load and store instructions have two supporting addressing modes:

- Scalar base plus immediate index.
- Scalar base plus scalar index.

For the predicated multiple vector contiguous structure load/store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The scalar index register value is multiplied by the memory element access size in bytes. Following each element access, the index value is incremented by one but the instruction does not update the scalar index register.
- Each predicate element applies to a single structure in memory, or equivalently to the same element number within each of the two, three, or four transferred vector registers.
- These instructions support packed data accesses only.
- When alignment checking is enabled for loads and stores, the base address must be aligned to the element access size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-68 Predicated multiple vector contiguous structure load/store

Mnemonic	Instruction	See
LD2	Contiguous load 2-byte structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2B (scalar plus immediate) LD2B (scalar plus scalar)
	Contiguous load two-doubleword structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2D (scalar plus immediate) LD2D (scalar plus scalar)
	Contiguous load two-halfword structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2H (scalar plus immediate) LD2H (scalar plus scalar)
	Contiguous load two-word structures to two vectors (scalar plus immediate, scalar plus scalar)	LD2W (scalar plus immediate) LD2W (scalar plus scalar)
LD3	Contiguous load 3-byte structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3B (scalar plus immediate) LD3B (scalar plus scalar)
	Contiguous load three-doubleword structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3D (scalar plus immediate) LD3D (scalar plus scalar)
	Contiguous load three-halfword structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3H (scalar plus immediate) LD3H (scalar plus scalar)
	Contiguous load three-word structures to three vectors (scalar plus immediate, scalar plus scalar)	LD3W (scalar plus immediate) LD3W (scalar plus scalar)
LD4	Contiguous load 4-byte structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4B (scalar plus immediate) LD4B (scalar plus scalar)
	Contiguous load four-doubleword structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4D (scalar plus immediate) LD4D (scalar plus scalar)
	Contiguous load four-halfword structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4H (scalar plus immediate) LD4H (scalar plus scalar)
	Contiguous load four-word structures to four vectors (scalar plus immediate, scalar plus scalar)	LD4W (scalar plus immediate) LD4W (scalar plus scalar)
ST2	Contiguous store 2-byte structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2B (scalar plus immediate) ST2B (scalar plus scalar)
	Contiguous store two-doubleword structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2D (scalar plus immediate) ST2D (scalar plus scalar)
	Contiguous store two-halfword structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2H (scalar plus immediate) ST2H (scalar plus scalar)
	Contiguous store two-word structures from two vectors (scalar plus immediate, scalar plus scalar)	ST2W (scalar plus immediate) ST2W (scalar plus scalar)
ST3	Contiguous store 3-byte structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3B (scalar plus immediate) ST3B (scalar plus scalar)
	Contiguous store three-doubleword structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3D (scalar plus immediate) ST3D (scalar plus scalar)
	Contiguous store three-halfword structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3H (scalar plus immediate) ST3H (scalar plus scalar)
	Contiguous store three-word structures from three vectors (scalar plus immediate, scalar plus scalar)	ST3W (scalar plus immediate) ST3W (scalar plus scalar)
ST4	Contiguous store 4-byte structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4B (scalar plus immediate) ST4B (scalar plus scalar)
	Contiguous store four-doubleword structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4D (scalar plus immediate) ST4D (scalar plus scalar)
	Contiguous store four-halfword structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4H (scalar plus immediate) ST4H (scalar plus scalar)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous store four-word structures from four vectors (scalar plus immediate, scalar plus scalar)	ST4W (scalar plus immediate) ST4W (scalar plus scalar)

C3.3.3 Predicated non-contiguous element accesses

Predicated non-contiguous element accesses address non-contiguous memory locations that are specified by either:

- A scalar base register plus a vector of indices or offsets.
- A vector of base addresses plus an immediate byte offset. The immediate byte offset is a multiple of the memory element access size, in the range 0 to 31 times the memory element access size, inclusive, and defaults to zero if omitted.

The predicated non-contiguous element accesses have two supporting addressing modes:

- Scalar base plus 64-bit vector index.
- Scalar base plus 32-bit vector index.
- Vector base plus immediate offset.

For this group of SVE instructions:

- Vector registers used as part of the address must specify a vector element size of 32 bits or 64 bits, .S or .D. For load and store instructions, the transfer register must specify the same vector element size.
- If the index vector register contains 32-bit index values then the lowest 32 bits of each index vector element can either be zero-extended or sign-extended to 64 bits.
- For load and store instructions, the index vector elements are then optionally multiplied by the memory element access size, in bytes, if a shift amount is specified. For prefetch instructions, the index vector elements are always multiplied by the memory element access size, in bytes.
- When alignment checking is enabled for loads and stores, the calculated virtual address of each element must be aligned to the memory element access size.

Table C3-69 Predicated non-contiguous element accesses

Mnemonic	Instructions	See
`LD1`	Gather load unsigned bytes to vector (scalar plus vector, vector plus immediate)	LD1B (scalar plus vector) LD1B (vector plus immediate)
	Gather load doublewords to vector (scalar plus vector, vector plus immediate)	LD1D (scalar plus vector) LD1D (vector plus immediate)
	Gather load unsigned halfwords (scalar plus vector, vector plus immediate)	LD1H (scalar plus vector) LD1H (vector plus immediate)
	Gather load signed bytes to vector (scalar plus vector, vector plus immediate)	LD1SB (scalar plus vector) LD1SB (vector plus immediate)
	Gather load signed halfwords to vector (scalar plus vector, vector plus immediate)	LD1SH (scalar plus vector) LD1SH (vector plus immediate)
	Gather load unsigned words to vector (scalar plus vector, vector plus immediate)	LD1SW (scalar plus vector) LD1SW (vector plus immediate)
`ST1`	Scatter store bytes from a vector (scalar plus vector, vector plus immediate)	ST1B (scalar plus vector) ST1B (vector plus immediate)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instructions	See
	Scatter store doublewords from a vector (scalar plus vector, vector plus immediate)	ST1D (scalar plus vector) ST1D (vector plus immediate)
	Scatter store halfwords from a vector (scalar plus vector, vector plus immediate)	ST1H (scalar plus vector) ST1H (vector plus immediate)
	Scatter store words from a vector (scalar plus vector, vector plus immediate)	ST1W (scalar plus vector) ST1W (vector plus immediate)
`LDFF1`	Gather load first-fault unsigned bytes to vector (scalar plus vector, vector plus immediate)	LDFF1B (scalar plus vector) LDFF1B (vector plus immediate)
	Gather load first-fault doublewords to vector (scalar plus vector, vector plus immediate)	LDFF1D (scalar plus vector) LDFF1D (vector plus immediate)
	Gather load first-fault unsigned halfwords to vector (scalar plus vector, vector plus immediate)	LDFF1H (scalar plus vector) LDFF1SH (vector plus immediate)
	Gather load first-fault signed bytes to vector (scalar plus vector, vector plus immediate)	LDFF1SB (scalar plus vector) LDFF1SB (vector plus immediate)
	Gather load first-fault signed halfwords to vector (scalar plus vector, vector plus immediate)	LDFF1SH (scalar plus vector) LDFF1SH (vector plus immediate)
	Gather load first-fault signed words to vector (scalar plus vector, vector plus immediate)	LDFF1SW (scalar plus vector) LDFF1SW (vector plus immediate)
`LDFF1`	Gather load first-fault unsigned words to vector (scalar plus vector, vector plus immediate)	LDFF1W (scalar plus vector) LDFF1W (vector plus immediate)
`PRF`	Gather prefetch bytes (scalar plus vector, vector plus immediate)	PRFB (scalar plus vector) PRFB (vector plus immediate)
	Gather prefetch doublewords (scalar plus vector, vector plus immediate)	PRFD (scalar plus vector) PRFD (vector plus immediate)
	Gather prefetch halfwords (scalar plus vector, vector plus immediate)	PRFH (scalar plus vector) PRFH (vector plus immediate)
	Gather prefetch words (scalar plus vector, vector plus immediate)	PRFW (scalar plus vector) PRFW (vector plus immediate)

C3.3.4 Predicated replicating element loads

The load and replicate instructions read one or more contiguous memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate byte offset.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate offset.
- Scalar base plus scalar index.

This defaults to zero if omitted. For predicated replicating element load SVE instructions:

- When alignment checking is enabled, the base address must be aligned to the memory element access size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-70 Predicated replicating element loads

Mnemonic	Instructions	See
LD1RB	Load and broadcast unsigned byte to vector	LD1RB
LD1RD	Load and broadcast doubleword to vector	LD1RD
LD1RH	Load and broadcast unsigned halfword to vector	LD1RH
LD1ROB	Contiguous load and replicate 32-bytes (scalar plus immediate, scalar plus scalar)	LD1ROB (scalar plus immediate) LD1ROB (scalar plus scalar)
LD1ROD	Contiguous load and replicate four doublewords (scalar plus immediate, scalar plus scalar)	LD1ROD (scalar plus immediate) LD1ROD (scalar plus scalar)
LD1ROH	Contiguous load and replicate 16 halfwords (scalar plus immediate, scalar plus scalar)	LD1ROH (scalar plus immediate) LD1ROH (scalar plus scalar)
LD1ROW	Contiguous load and replicate eight words (scalar plus immediate, scalar plus scalar)	LD1ROW (scalar plus immediate) LD1ROW (scalar plus scalar)
LD1RQB	Contiguous load and replicate 16 bytes (scalar plus immediate, scalar plus scalar)	LD1RQB (scalar plus immediate) LD1RQB (scalar plus scalar)
LD1RQD	Contiguous load and replicate two doublewords (scalar plus immediate, scalar plus scalar)	LD1RQD (scalar plus immediate) LD1RQD (scalar plus scalar)
LD1RQH	Contiguous load and replicate eight halfwords (scalar plus immediate, scalar plus scalar)	LD1RQH (scalar plus immediate) LD1RQH (scalar plus scalar)
LD1RQW	Contiguous load and replicate four words (scalar plus immediate, scalar plus scalar)	LD1RQW (scalar plus immediate) LD1RQW (scalar plus scalar)
LD1RSB	Load and broadcast signed byte to vector	LD1RSB
LD1RSH	Load and broadcast signed halfword to vector	LD1RSH
LD1RSW	Load and broadcast signed word to vector	LD1RSW
LD1RW	Load and broadcast unsigned word to vector	LD1RW

C3.3.5 Unpredicated vector register load/store

The unpredicated vector register load, LDR, and store, STR, instructions transfer a single vector register from or to memory locations that are specified by a scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted.

The unpredicated vector register load/store instructions have one supporting addressing mode:

- Scalar base plus immediate index.

For the unpredicated vector register load/store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the current vector register length in bytes.
- The data transfer is performed as a contiguous stream of byte accesses in ascending element order, without endianness conversion.
- When alignment checking is enabled for loads and stores, the base address must be 16-byte aligned.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-71 Unpredicated vector register load/store

Mnemonic	Instruction	See
LDR	Load vector register (vector)	LDR (vector)
STR	Store vector register (vector)	STR (vector)

C3.3.6 Unpredicated predicate register load/store

The unpredicated predicate register load, LDR, and store, STR, instructions transfer a single predicate register from or to memory locations that are specified by a scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted.

The unpredicated predicate register load/store instructions have one supporting addressing mode:

- Scalar base plus immediate index.

For unpredicated predicate register load/store SVE instructions:

- The immediate index value is a predicate index, not an element index. The immediate index value is multiplied by the current predicate register length, in bytes.
- The data transfer is performed as a contiguous stream of byte accesses, each byte containing 8 consecutive predicate bits, in ascending bit and element order, without endian conversion.
- When alignment checking is enabled for loads and stores, the base address must be 2-byte aligned.

Table C3-72 Unpredicated predicate register load/store

Mnemonic	Instruction	See
LDR	Load predicate register (predicate)	LDR (predicate)
STR	Store predicate register (predicate)	STR (predicate)

C3.3.7 Non-temporal gather/scatter

The non-temporal gather load and scatter store instructions provide a hint to the memory system that the data structure being accessed has a low reuse frequency. The memory system can use the hint to avoid retaining the data or evicting more frequently-used data from the caches.

These instructions support a single addressing mode consisting of 64-bit or 32-bit vector base addresses plus an unscaled 64-bit scalar offset that defaults to the zero register, XZR. Other addressing modes can be constructed using extra instructions.

Table C3-73 Non-temporal gather load and scatter store instructions

Mnemonic	Instruction	See
LDNT1B	Gather load non-temporal unsigned bytes	LDNT1B (vector plus scalar)
LDNT1D	Gather load non-temporal unsigned doublewords	LDNT1D (vector plus scalar)
LDNT1H	Gather load non-temporal unsigned halfwords	LDNT1H (vector plus scalar)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
LDNT1SB	Gather load non-temporal signed bytes	LDNT1SB
LSNT1SH	Gather load non-temporal signed halfwords	LDNT1SH
LSNT1SW	Gather load non-temporal signed words	LDNT1SW
LDNT1W	Gather load non-temporal unsigned words	LDNT1W (vector plus scalar)
STNT1B	Scatter store non-temporal bytes	STNT1B (vector plus scalar)
STNT1D	Scatter store non-temporal doublewords	STNT1D (vector plus scalar)
STNT1H	Scatter store non-temporal halfwords	STNT1H (vector plus scalar)
STNT1W	Scatter store non-temporal words	STNT1W (vector plus scalar)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4 Loads and stores - SME, SME2, SVE2p1

The following subsections describe the SME, SME2 and SVE2p1 loads and stores:

- Array vector/table load and store.
- Consecutive multi-vector loads and stores.
- SVE2.1 quadword loads and stores.

C3.4.1 Array vector/table load and store

These instructions transfer data between the ZA array vector or the ZT0 registers and memory.

These instructions are performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte.

Table C3-74 SME array vector/table load and store

Mnemonic	Instruction	See
`LDR`	Load ZA array vector	LDR (array vector)
	Load ZT0 register	LDR (table)
`STR`	Store ZA array vector	STR (array vector)
	Store ZT0 register	STR (table)

C3.4.2 Contiguous multi-vector loads and stores

C3.4.2.1 Consecutive multi-vector loads and stores

Multi-vector load and store instructions that transfer data between memory and consecutive SVE vector registers.

Table C3-75 Consecutive multi-vector load and store instructions

Mnemonic	Instruction	See
`LD1`	Contiguous load of bytes to multiple consecutive vectors (immediate index, scalar index)	LD1B (scalar plus immediate, consecutive registers) LD1B (scalar plus scalar, consecutive registers)
	Contiguous load of doublewords to multiple consecutive vectors (immediate index, scalar index)	LD1D (scalar plus immediate, consecutive registers) LD1D (scalar plus scalar, consecutive registers)
	Contiguous load of halfwords to multiple consecutive vectors (immediate index, scalar index)	LD1H (scalar plus immediate, consecutive registers) LD1H (scalar plus scalar, consecutive registers)
	Contiguous load of words to multiple consecutive vectors (immediate index, scalar index)	LD1W (scalar plus immediate, consecutive registers) LD1W (scalar plus scalar, consecutive registers)
`LDNT1`	Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index, scalar index)	LDNT1B (scalar plus immediate, consecutive registers) LDNT1B (scalar plus scalar, consecutive registers)
	Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index, scalar index)	LDNT1D (scalar plus immediate, consecutive registers) LDNT1D (scalar plus scalar, consecutive registers)
	Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index, scalar index)	LDNT1H (scalar plus immediate, consecutive registers) LDNT1H (scalar plus scalar, consecutive registers)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous load non-temporal of words to multiple consecutive vectors (immediate index, scalar index)	LDNT1W (scalar plus immediate, consecutive registers) LDNT1W (scalar plus scalar, consecutive registers)
ST1	Contiguous store of bytes from multiple consecutive vectors (immediate index, scalar index)	ST1B (scalar plus immediate, consecutive registers) ST1B (scalar plus scalar, consecutive registers)
	Contiguous store of doublewords from multiple consecutive vectors (immediate index, scalar index)	ST1D (scalar plus immediate, consecutive registers) ST1D (scalar plus scalar, consecutive registers)
	Contiguous store of halfwords from multiple consecutive vectors (immediate index, scalar index)	ST1H (scalar plus immediate, consecutive registers) ST1H (scalar plus scalar, consecutive registers)
	Contiguous store of words from multiple consecutive vectors (immediate index, scalar index)	ST1W (scalar plus immediate, consecutive registers) ST1W (scalar plus scalar, consecutive registers)
STNT1	Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index, scalar index)	STNT1B (scalar plus immediate, consecutive registers) STNT1B (scalar plus scalar, consecutive registers)
	Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index, scalar index)	STNT1D (scalar plus immediate, consecutive registers) STNT1D (scalar plus scalar, consecutive registers)
	Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index, scalar index)	STNT1H (scalar plus immediate, consecutive registers) STNT1H (scalar plus scalar, consecutive registers)
	Contiguous store non-temporal of words from multiple consecutive vectors (immediate index, scalar index)	STNT1W (scalar plus immediate, consecutive registers) STNT1W (scalar plus scalar, consecutive registers)

C3.4.2.2 Strided multi-vector loads and stores

Multi-vector load and store instructions that transfer data between memory and strided numbered vector registers.

Table C3-76 Strided multi-vector load and store instructions

Mnemonic	Instruction	See
LD1	Contiguous load of bytes to multiple strided vectors (immediate index, scalar index)	LD1B (scalar plus immediate, strided registers) LD1B (scalar plus scalar, strided registers)
	Contiguous load of doublewords to multiple strided vectors (immediate index, scalar index)	LD1D (scalar plus immediate, strided registers) LD1D (scalar plus scalar, strided registers)
	Contiguous load of halfwords to multiple strided vectors (immediate index, scalar index)	LD1H (scalar plus immediate, strided registers) LD1H (scalar plus scalar, strided registers)
	Contiguous load of words to multiple strided vectors (immediate index, scalar index)	LD1W (scalar plus immediate, strided registers) LD1W (scalar plus scalar, strided registers)
LDNT1	Contiguous load non-temporal of bytes to multiple strided vectors (immediate index, scalar index)	LDNT1B (scalar plus immediate, strided registers) LDNT1B (scalar plus scalar, strided registers)
	Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index, scalar index)	LDNT1D (scalar plus immediate, strided registers) LDNT1D (scalar plus scalar, strided registers)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index, scalar index)	LDNT1H (scalar plus immediate, strided registers) LDNT1H (scalar plus scalar, strided registers)
	Contiguous load non-temporal of words to multiple strided vectors (immediate index, scalar index)	LDNT1W (scalar plus immediate, strided registers) LDNT1W (scalar plus scalar, strided registers)
`ST1`	Contiguous store of bytes from multiple strided vectors (immediate index, scalar index)	ST1B (scalar plus immediate, strided registers) ST1B (scalar plus scalar, strided registers)
	Contiguous store of doublewords from multiple strided vectors (immediate index, scalar index)	ST1D (scalar plus immediate, strided registers) ST1D (scalar plus scalar, strided registers)
	Contiguous store of halfwords from multiple strided vectors (immediate index, scalar index)	ST1H (scalar plus immediate, strided registers) ST1H (scalar plus scalar, strided registers)
	Contiguous store of words from multiple strided vectors (immediate index, scalar index)	ST1W (scalar plus immediate, strided registers) ST1W (scalar plus scalar, strided registers)
`STNT1`	Contiguous store non-temporal of bytes from multiple strided vectors (immediate index, scalar index)	STNT1B (scalar plus immediate, strided registers) STNT1B (scalar plus scalar, strided registers)
	Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index, scalar index)	STNT1D (scalar plus immediate, strided registers) STNT1D (scalar plus scalar, strided registers)
	Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index, scalar index)	STNT1H (scalar plus immediate, strided registers) STNT1H (scalar plus scalar, strided registers)
	Contiguous store non-temporal of words from multiple strided vectors (immediate index, scalar index)	STNT1W (scalar plus immediate, strided registers) STNT1W (scalar plus scalar, strided registers)

C3.4.2.3 Tile slice multi-vector loads and stores

Tile slice load and store instructions transfer bytes between a tile slice and memory.

Table C3-77 Tile slice multi-vector load and store instructions

Mnemonic	Instruction	See
`LD1`	Contiguous load of bytes to 8-bit element ZA tile slice	LD1B (scalar plus scalar, tile slice)
	Contiguous load of doublewords to 64-bit element ZA tile slice	LD1D (scalar plus scalar, tile slice)
	Contiguous load of halfwords to 16-bit element ZA tile slice	LD1H (scalar plus scalar, tile slice)
	Contiguous load of quadwords to 128-bit element ZA tile slice	LD1Q
	Contiguous load of words to 32-bit element ZA tile slice	LD1W (scalar plus scalar, tile slice)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
ST1	Contiguous store of bytes from 8-bit element ZA tile slice	ST1B (scalar plus scalar, tile slice)
	Contiguous store of doublewords from to 64-bit element ZA tile slice	ST1D (scalar plus scalar, tile slice)
	Contiguous store of halfwords from 16-bit element ZA tile slice	ST1H (scalar plus scalar, tile slice)
	Contiguous store of quadwords from 128-bit element ZA tile slice	ST1Q
	Contiguous store of words from 32-bit element ZA tile slice	ST1W (scalar plus scalar, tile slice)

C3.4.3 SVE2.1 quadword loads and stores

C3.4.3.1 Contiguous quadword loads and stores

Contiguous load or store to/from elements of a vector register from/to the memory address generated by a scalar base and an immediate index, which is multiplied and added to the base address, or a scalar index, which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Table C3-78 SME2/SVE2.1 Contiguous quadword load and store instructions

Mnemonic	Instruction	See
LD1D	Contiguous load unsigned doublewords to vector (immediate or scalar index)	LD1D (scalar plus immediate, single register) LD1D (scalar plus scalar, single register)
LD1W	Contiguous load unsigned words to vector (immediate or scalar index)	LD1W (scalar plus immediate, single register) LD1W (scalar plus scalar, single register)
ST1D	Contiguous store doublewords from vector (immediate or scalar index)	ST1D (scalar plus immediate, single register) ST1D (scalar plus scalar, single register)
ST1W	Contiguous store words from vector (immediate or scalar index)	ST1W (scalar plus immediate, single register) ST1W (scalar plus scalar, single register)

C3.4.3.2 Quadword gather and scatter

Gather load or scatter store of quadwords between Active elements of a vector register and memory addresses generated by a vector base plus an unscaled scalar register offset.

Table C3-79 SME2/SVE2.1 Quadword gather and scatter instructions

Mnemonic	Instruction	See
LD1Q	Gather load quadwords	LD1Q
ST1Q	Scatter store quadwords	ST1Q

C3.4.3.3 Quadword structure loads and stores

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-80 SVE2.1 Quadword structure load and store instructions

Mnemonic	Instruction	See
LD	Contiguous load two-quadword structures to two vectors (immediate index, scalar index)	LD2Q (scalar plus immediate) LD2Q (scalar plus scalar)
	Contiguous load three-quadword structures to three vectors (immediate index, scalar index)	LD3Q (scalar plus immediate) LD3Q (scalar plus scalar)
	Contiguous load four-quadword structures to four vectors (immediate index, scalar index)	LD4Q (scalar plus immediate) LD4Q (scalar plus scalar)
ST	Contiguous store two-quadword structures from two vectors (immediate index, scalar index)	ST2Q (scalar plus immediate) ST2Q (scalar plus scalar)
	Contiguous store three-quadword structures from three vectors (immediate index, scalar index)	ST3Q (scalar plus immediate) ST3Q (scalar plus scalar)
	Contiguous store four-quadword structures from four vectors (immediate index, scalar index)	ST4Q (scalar plus immediate) ST4Q (scalar plus scalar)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- Arithmetic (immediate).
- Integer minimum and maximum (immediate)
- Logical (immediate).
- Move (wide immediate).
- Move (immediate).
- PC-relative address calculation.
- Bitfield move.
- Bitfield insert and extract
- Extract register.
- Shift (immediate).
- Sign-extend and Zero-extend.

C3.5.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits.

The Arithmetic (immediate) instructions that do not set Condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-81 shows the Arithmetic instructions with an immediate offset.

Table C3-81 Arithmetic instructions with an immediate

Mnemonic	Instruction	See
`ADD`	Add	ADD (immediate)
`ADDS`	Add and set flags	ADDS (immediate)
`SUB`	Subtract	SUB (immediate)
`SUBS`	Subtract and set flags	SUBS (immediate)
`CMP`	Compare	CMP (immediate)
`CMN`	Compare negative	CMN (immediate)

C3.5.2 Integer minimum and maximum (immediate)

The Integer maximum and minimum (immediate) instructions determine the maximum/minimum of the source register value and immediate.

These instructions are only present when FEAT_CSSC is implemented.

Table C3-82 shows the Integer maximum and minimum (immediate) instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-82 Integer maximum and minimum (immediate) instructions

Mnemonic	Instruction	See
SMAX	Signed Maximum (immediate)	SMAX (immediate)
SMIN	Signed Minimum (immediate)	SMIN (immediate)
UMAX	Unsigned Maximum (immediate)	UMAX (immediate)
UMIN	Unsigned Minimum (immediate)	UMIN (immediate)

C3.5.3 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

Note

Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the Condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.

Note

Apart from ANDS and its TST alias, Logical (immediate) instructions do not set the Condition flags. However, the final results of a bitwise operation can be tested by a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-83 shows the Logical immediate instructions.

Table C3-83 Logical immediate instructions

Mnemonic	Instruction	See
AND	Bitwise AND	AND (immediate)
ANDS	Bitwise AND and set flags	ANDS (immediate)
EOR	Bitwise exclusive OR	EOR (immediate)
ORR	Bitwise inclusive OR	ORR (immediate)
TST	Test bits	TST (immediate)

C3.5.4 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-84 shows the Move (wide immediate) instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-84 Move (wide immediate) instructions

Mnemonic	Instruction	See
MOVZ	Move wide with zero	MOVZ
MOVN	Move wide with NOT	MOVN
MOVK	Move wide with keep	MOVK

C3.5.5 Move (immediate)

The Move (immediate) instructions are aliases for a single MOVZ, MOVN, or ORR (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly, it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the MOVZ, MOVN, and ORR instruction to encode the immediate, then an assembler must prefer MOVZ to MOVN, and MOVZ or MOVN to ORR, to ensure reversability. A disassembler must output ORR (immediate with zero register) MOVZ, and MOVN, as a MOV mnemonic except that the underlying instruction must be used when:

- ORR has an immediate that can be generated by a MOVZ or MOVN instruction.
- A MOVN instruction has an immediate that can be encoded by MOVZ.
- MOVZ #0 or MOVN #0 have a shift amount other than LSL #0.

Table C3-85 shows the Move (immediate) instructions.

Table C3-85 Move (immediate) instructions

Mnemonic	Instruction	See
MOV	Move (inverted wide immediate)	MOV (inverted wide immediate)
	Move (wide immediate)	MOV (wide immediate)
	Move (bitmask immediate)	MOV (bitmask immediate)

C3.5.6 PC-relative address calculation

The ADR instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The ADRP instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an ADD (immediate) instruction, or a load/store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

Note

The term *page* used in the ADRP description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-86 shows the instructions used for PC-relative address calculations are as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-86 PC-relative address calculation instructions

Mnemonic	Instruction	See
ADRP	Compute address of 4KB page at a PC-relative offset	ADRP
ADR	Compute address of label at a PC-relative offset.	ADR

C3.5.7 Bitfield move

The Bitfield move instructions copy a field of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For BFM, the remaining bits are unchanged.
- For UBFM the lower bits, if any, and upper bits, if any, are set to zero.
- For SBFM, the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied field.

Table C3-87 shows the Bitfield move instructions.

Table C3-87 Bitfield move instructions

Mnemonic	Instruction	See
BFM	Bitfield move	BFM
SBFM	Signed bitfield move	SBFM
UBFM	Unsigned bitfield move (32-bit)	UBFM

C3.5.8 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-88 shows the Bitfield insert and extract aliases.

Table C3-88 Bitfield insert and extract instructions

Mnemonic	Instruction	See
BFC	Bitfield clear	BFC
BFI	Bitfield insert	BFI
BFXIL	Bitfield extract and insert low	BFXIL
SBFIZ	Signed bitfield insert in zero	SBFIZ
SBFX	Signed bitfield extract	SBFX
UBFIZ	Unsigned bitfield insert in zero	UBFIZ
UBFX	Unsigned bitfield extract	UBFX

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See

C3.5.9 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-89 shows the Extract (immediate) instructions.

Table C3-89 Extract register instructions

Mnemonic	Instruction	See
`EXTR`	Extract register from pair	EXTR

C3.5.10 Shift (immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.

Table C3-90 shows the aliases that can be used as immediate shift and rotate instructions.

Table C3-90 Aliases for immediate shift and rotate instructions

Mnemonic	Instruction	See
`ASR`	Arithmetic shift right	ASR (immediate)
`LSL`	Logical shift left	LSL (immediate)
`LSR`	Logical shift right	LSR (immediate)
`ROR`	Rotate right	ROR (immediate)

C3.5.11 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-91 shows the aliases that can be used as zero-extend and sign-extend instructions.

Table C3-91 Zero-extend and sign-extend instructions

Mnemonic	Instruction	See
`SXTB`	Sign-extend byte	SXTB
`SXTH`	Sign-extend halfword	SXTH
`SXTW`	Sign-extend word	SXTW

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
`UXTB`	Unsigned extend byte	UXTB
`UXTH`	Unsigned extend halfword	UXTH

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.6 Data Processing - MTE

The following subsections describe the MTE processing instructions:

- Pointer arithmetic instructions.
- Tag generation instructions.

C3.6.1 Pointer arithmetic instructions

Table C3-92 shows the Memory Tagging Extension Pointer Arithmetic instruction.

Table C3-92 MTE Pointer Arithmetic instruction

Mnemonic	Instruction	See
`SUBP`	Subtract pointer	SUBP
`SUBPS`	Subtract pointer, setting flags	SUBPS

C3.6.2 Tag generation instructions

Table C3-93 shows the Memory Tagging Extension Tag generation instructions.

Table C3-93 Tag generation instructions

Mnemonic	Instruction	See
`ADDG`	Add with tag	ADDG
`GMI`	Tag mask insert	GMI
`IRG`	Insert random tag	IRG
`SUBG`	Subtract with tag	SUBG

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- Arithmetic (shifted register).
- Arithmetic (extended register).
- Arithmetic with carry.
- Integer maximum and minimum (register)
- Flag manipulation instructions.
- Logical (shifted register).
- Move (register).
- Absolute value
- Shift (register).
- Multiply and divide.
- CRC32.
- Bit operation.
- Conditional select.
- Conditional comparison.

C3.7.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR`, and `LSR` accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See Arithmetic (extended register) for arithmetic instructions that can operate on the current stack pointer.

Table C3-94 shows the Arithmetic (shifted register) instructions.

Table C3-94 Arithmetic (shifted register) instructions

Mnemonic	Instruction	See
`ADD`	Add	ADD (shifted register)
`ADDS`	Add and set flags	ADDS (shifted register)
`SUB`	Subtract	SUB (shifted register)
`SUBS`	Subtract and set flags	SUBS (shifted register)
`CMN`	Compare negative	CMN (shifted register)
`CMP`	Compare	CMP (shifted register)
`NEG`	Negate	NEG (shifted register)
`NEGS`	Negate and set flags	NEGS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The extended shift is described by the mandatory extend operator `SXTB`, `SXTH`, `SXTW`, `UXTB`, `UXTH`, or `UXTW` . This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms, the additional operators `UXTX` and `SXTX` use all 64 bits of the second source register with an optional shift. In that case, Arm recommends `UXTX` as the operator. If and only if at least one register is SP, Arm recommends use of the `LSL` operator name, rather than `UXTX`, and when the shift amount is also zero then both the operator and the shift amount can be omitted. `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms, the operators `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case, Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions, the final register operand is written as `Wm` for all except the `UXTX`/`LSL` and `SXTX` extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1              // SUB SP, SP, X1, UXTX #0
```

Table C3-95 shows the Arithmetic (extended register) instructions.

Table C3-95 Arithmetic (extended register) instructions

Mnemonic	Instruction	See
`ADD`	Add	ADD (extended register)
`ADDS`	Add and set flags	ADDS (extended register)
`SUB`	Subtract	SUB (extended register)
`SUBS`	Subtract and set flags	SUBS (extended register)
`CMN`	Compare negative	CMN (extended register)
`CMP`	Compare	CMP (extended register)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-96 shows the Arithmetic with carry instructions

Table C3-96 Arithmetic with carry instructions

Mnemonic	Instruction	See
ADC	Add with carry	ADC
ADCS	Add with carry and set flags	ADCS
SBC	Subtract with carry	SBC
SBCS	Subtract with carry and set flags	SBCS
NGC	Negate with carry	NGC
NGCS	Negate with carry and set flags	NGCS

C3.7.4 Integer maximum and minimum (register)

The Integer maximum and minimum (register) instructions determine the maximum/minimum of the two source register values.

These instructions are only present when FEAT_CSSC is implemented.

Table C3-97 shows the Integer maximum and minimum (register) instructions.

Table C3-97 Integer maximum and minimum (register) instructions

Mnemonic	Instruction	See
SMAX	Signed Maximum (register)	SMAX (register)
SMIN	Signed Minimum (register)	SMIN (register)
UMAX	Unsigned Maximum (register)	UMAX (register)
UMIN	Unsigned Minimum (register)	UMIN (register)

C3.7.5 Flag manipulation instructions

The Flag manipulation instructions set the value of the NZCV condition flags directly.

The instructions SETF8 and SETF16 accept one source register and set the NZV condition flags based on the value of the input register. The instruction RMIF accepts one source register and two immediate values, rotating the first source register using the first immediate value and setting the NZCV condition flags masked by the second immediate value.

The instructions XAFLAG and AXFLAG convert PSTATE condition flags between the FCMP instruction format and an alternative format. See Table C6-1 for more information.

Table C3-98 shows the Flag manipulation instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-98 Flag manipulation instructions

Mnemonic	Instruction	See
AXFLAG	Convert from FCMP comparison format to the alternative format	AXFLAG
CFINV	Invert value of the PSTATE.C bit	CFINV
RMIF	Rotate, mask insert flags	RMIF
SETF8	Evaluation of 8-bit flags	SETF8, SETF16
SETF16	Evaluation of 16-bit flags	SETF8, SETF16
XAFLAG	Convert from alternative format to FCMP comparison format	XAFLAG

C3.7.6 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, LSR, and ROR accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator and amount implies LSL #0, which means that there is no shift. A disassembler must not output LSL #0. However, a disassembler must output all other shifts by zero.

Note

Apart from ANDS, TST, and BICS, the logical instructions do not set the Condition flags, but the final result of a bit operation can usually directly control a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-99 shows the Logical (shifted register) instructions.

Table C3-99 Logical (shifted register) instructions

Mnemonic	Instruction	See
AND	Bitwise AND	AND (shifted register)
ANDS	Bitwise AND and set flags	ANDS (shifted register)
BIC	Bitwise bit clear	BIC (shifted register)
BICS	Bitwise bit clear and set flags	BICS (shifted register)
EON	Bitwise exclusive-OR NOT	EON (shifted register)
EOR	Bitwise exclusive-OR	EOR (shifted register)
ORR	Bitwise inclusive OR	ORR (shifted register)
MVN	Bitwise NOT	MVN
ORN	Bitwise inclusive OR NOT	ORN (shifted register)
TST	Test bits	TST (shifted register)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.7 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

Table C3-100 MOV register instructions

Mnemonic	Instruction	See
`MOV`	Move register	MOV (register)
	Move register to SP or move SP to register	MOV (to/from SP)

C3.7.8 Absolute value

The Absolute value instruction is only present when FEAT_CSSC is implemented.

Table C3-101 shows the Absolute value instruction.

Table C3-101 Absolute value instruction

Mnemonic	Instruction	See
`ABS`	Absolute value	ABS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.7.9 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].

Table C3-102 shows the Shift (register) instructions.

Table C3-102 Shift (register) instructions

Mnemonic	Instruction	See
`ASRV`	Arithmetic shift right variable	ASRV
`LSLV`	Logical shift left variable	LSLV
`LSRV`	Logical shift right variable	LSRV
`RORV`	Rotate right variable	RORV

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in Shift (immediate).

Table C3-103 shows the aliases for Shift (register) instructions.

Table C3-103 Aliases for Variable shift instructions

Mnemonic	Instruction	See
`ASR`	Arithmetic shift right	ASR (register)
`LSL`	Logical shift left	LSL (register)
`LSR`	Logical shift right	LSR (register)
`ROR`	Rotate right	ROR (register)

C3.7.10 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- Multiply.
- Divide.

C3.7.10.1 Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using `SMULH` or `UMULH` to generate the upper 64 bits. Table C3-104 shows the Multiply instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-104 Multiply integer instructions

Mnemonic	Instruction	See
MADD	Multiply-add	MADD
MSUB	Multiply-subtract	MSUB
MNEG	Multiply-negate	MNEG
MUL	Multiply	MUL
SMADDL	Signed multiply-add long	SMADDL
SMSUBL	Signed multiply-subtract long	SMSUBL
SMNEGL	Signed multiply-negate long	SMNEGL
SMULL	Signed multiply long	SMULL
SMULH	Signed multiply high	SMULH
UMADDL	Unsigned multiply-add long	UMADDL
UMSUBL	Unsigned multiply-subtract long	UMSUBL
UMNEGL	Unsigned multiply-negate long	UMNEGL
UMULL	Unsigned multiply long	UMULL
UMULH	Unsigned multiply high	UMULH

C3.7.10.2 Divide

The Divide instructions calculate the quotient of a division, rounded towards zero. The remainder can then be calculated as (numerator - (quotient × denominator)), using the `MSUB` instruction.

If a signed integer division (`INT_MIN / -1`) is performed where `INT_MIN` is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is `INT_MIN`.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-105 shows the Divide instructions.

Table C3-105 Divide instructions

Mnemonic	Instruction	See
SDIV	Signed divide	SDIV
UDIV	Unsigned divide	UDIV

C3.7.11 CRC32

The `CRC32` instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of `CRC` instructions, `CRC32`, and `CRC32C`, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to `0b0001`, the `CRC` instructions are implemented.

These instructions are optional in an Armv8.0 implementation.

All implementations of Armv8.1 architecture and later are required to implement the `CRC32` instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-106 shows the CRC instructions.

Table C3-106 CRC32 instructions

Mnemonic	Instruction	See
CRC32B	CRC-32 sum from byte	CRC32B, CRC32H, CRC32W, CRC32X
CRC32H	CRC-32 sum from halfword	CRC32B, CRC32H, CRC32W, CRC32X
CRC32W	CRC-32 sum from word	CRC32B, CRC32H, CRC32W, CRC32X
CRC32X	CRC-32 sum from doubleword	CRC32B, CRC32H, CRC32W, CRC32X
CRC32CB	CRC-32C sum from byte	CRC32CB, CRC32CH, CRC32CW, CRC32CX
CRC32CH	CRC-32C sum from halfword	CRC32CB, CRC32CH, CRC32CW, CRC32CX
CRC32CW	CRC-32C sum from word	CRC32CB, CRC32CH, CRC32CW, CRC32CX
CRC32CX	CRC-32C sum from doubleword	CRC32CB, CRC32CH, CRC32CW, CRC32CX

C3.7.12 Bit operation

The CNT and CTZ instructions are only present when FEAT_CSSC is implemented.

Table C3-107 shows the Bit operation instructions.

Table C3-107 Bit operation instructions

Mnemonic	Instruction	See
CLS	Count leading sign bits	CLS
CLZ	Count leading zero bits	CLZ
CNT	Count bits	CNT
CTZ	Count trailing zero bits	CTZ
RBIT	Reverse bit order	RBIT
REV	Reverse bytes in register	REV
REV16	Reverse bytes in halfwords	REV16
REV32	Reverse bytes in words	REV32
REV64	Reverse bytes in register	REV64

C3.7.13 Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the Condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.

Table C3-108 shows the Conditional select instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-108 Conditional select instructions

Mnemonic	Instruction	See
CSEL	Conditional select	CSEL
CSINC	Conditional select increment	CSINC
CSINV	Conditional select inversion	CSINV
CSNEG	Conditional select negation	CSNEG
CSET	Conditional set	CSET
CSETM	Conditional set mask	CSETM
CINC	Conditional increment	CINC
CINV	Conditional invert	CINV
CNEG	Conditional negate	CNEG

C3.7.14 Conditional comparison

The Conditional comparison instructions provide a conditional select for the NZCV Condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-109 shows the Conditional comparison instructions.

Table C3-109 Conditional comparison instructions

Mnemonic	Instruction	See
CCMN	Conditional compare negative (register)	CCMN (register)
CCMN	Conditional compare negative (immediate)	CCMN (immediate)
CCMP	Conditional compare (register)	CCMP (register)
CCMP	Conditional compare (immediate)	CCMP (immediate)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8 Data processing - SIMD and floating-point

This section describes the instruction groups for data processing with SIMD and floating-point register operands.

Common features of SIMD instructions gives general information about SIMD instructions.

The following subsections describe the scalar floating-point data processing instructions:

- Floating-point move (register).
- Floating-point move (immediate).
- Floating-point conversion.
- Floating-point round to integral value.
- Floating-point multiply-add.
- Floating-point arithmetic (one source).
- Floating-point arithmetic (two sources).
- Floating-point minimum and maximum.
- Floating-point comparison.
- Floating-point conditional select.

The following subsections describe the SIMD data processing instructions:

- SIMD move.
- SIMD arithmetic.
- SIMD compare.
- SIMD widening and narrowing arithmetic.
- SIMD unary arithmetic.
- SIMD by element arithmetic.
- SIMD permute.
- SIMD immediate.
- SIMD shift (immediate).
- SIMD conversions.
- SIMD reduce (across vector lanes).
- SIMD pairwise arithmetic.
- SIMD matrix multiply-accumulate.
- SIMD dot product.
- SIMD table lookup.
- SIMD complex number arithmetic instructions.
- SIMD BFloat16.
- SIMD multiply-accumulate.
- The Cryptographic Extension in AArch64 state.

For information about the floating-point exceptions, see Floating-point exceptions and exception traps.

C3.8.1 Common features of SIMD instructions

A number of SIMD instructions come in three forms:

Wide Indicated by the suffix *W*. The element width of the destination register and the first source operand is double that of the second source operand.

Long Indicated by the suffix *L*. The element width of the destination register is double that of both source operands.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Narrow Indicated by the suffix *N*. The element width of the destination register is half that of both source operands.

In addition, each vector form of the instruction is part of a pair, with a second and upper part suffix of *2*, to identify the variant of the instruction:

- Where a SIMD operation widens or lengthens a 64-bit vector to a 128-bit vector, the instruction provides a second part operation that can extract the source from the upper 64 bits of the source registers.
- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the instruction provides a second-part operation that can pack the result of a second operation into the upper part of the same destination register.

Note

This is referred to as a *lane set specifier*.

C3.8.2 Floating-point move (register)

The Floating-point move (register) instructions copy a scalar floating-point value from one register to another register without performing any conversion.

Some of the Floating-point move (register) instructions overlap with the functionality provided by the Advanced SIMD instructions `DUP`, `INS`, and `UMOV`. However, Arm recommends using the `FMOV` instructions when operating on scalar floating-point data to avoid the creation of scalar floating-point code that depends on the availability of the Advanced SIMD instruction set.

Table C3-110 shows the Floating-point move (register) instructions.

Table C3-110 Floating-point move (register) instructions

Mnemonic	Instruction	See
`FMOV`	Floating-point move register without conversion	FMOV (register)
	Floating-point move to or from general-purpose register without conversion	FMOV (general)

C3.8.3 Floating-point move (immediate)

The Floating-point move (immediate) instructions convert a small constant immediate floating-point value into a half-precision, single-precision, or double-precision scalar floating-point value in a SIMD and floating-point register.

The floating-point constant can be specified either in decimal notation, such as 12.0 or -1.2e1, or as a string beginning with `0x` followed by a hexadecimal representation of the IEEE 754 half-precision, single-precision, or double-precision encoding. Arm recommends that a disassembler uses the decimal notation, provided that this displays the value precisely.

Note

When FEAT_FP16 is not implemented, the only half-precision instructions that are supported are floating-point conversions between half-precision, single-precision, and double-precision.

The floating-point value must be expressible as ($\pm n/16 \times 2^r$), where n is an integer in the range $16 \leq n \leq 31$ and r is an integer in the range of $-3 \leq r \leq 4$, that is a normalized binary floating-point encoding with one sign bit, four bits of fraction, and a 3-bit exponent.

Table C3-111 shows the Floating-point move (immediate) instruction:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-111 Floating-point move (immediate) instruction

Mnemonic	Instruction	See
FMOV	Floating-point move immediate	FMOV (scalar, immediate)

C3.8.4 Floating-point conversion

The following subsections describe the conversion of floating-point values:

- Convert floating-point precision.
- Convert from floating-point single-precision to BFloat16.
- Convert from floating-point to integer or fixed-point.
- Convert from integer or fixed-point to floating-point.

C3.8.4.1 Convert floating-point precision

These instructions convert a floating-point scalar with one precision to a floating-point scalar with a different precision, using the current rounding mode as specified by FPCR.RMode.

Table C3-112 shows the Floating-point precision conversion instruction.

Table C3-112 Floating-point precision conversion instruction

Mnemonic	Instruction	See
FCVT	Floating-point convert precision (scalar)	FCVT

C3.8.4.2 Convert from floating-point single-precision to BFloat16

The BFCVT instruction is provided by FEAT_BF16. This instruction converts a single-precision floating-point input to BFloat16 format, honoring the FPCR rounding mode controls to give a more accurate conversion than simply removing the bottom 16 bits of the input.

Table C3-113 shows this instruction.

Table C3-113 Floating-point single-precision to BFloat16 conversion instruction

Mnemonic	Instruction	See
BFCVT	BFloat16 floating-point convert from single-precision to BFloat16 format (scalar)	BFCVT

C3.8.4.3 Convert from floating-point to integer or fixed-point

Floating-point to integer or fixed-point conversion is supported by the following instruction variants:

- Scalar instruction variant, which converts a floating-point value in a SIMD and floating-point source register to an integer or fixed-point value in a general-purpose destination register.
- Scalar SIMD&FP instruction variant, which converts a floating-point value in a SIMD and floating-point source register to an integer value in a SIMD and floating-point destination register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

Table C3-114 shows the floating-point to integer or fixed-point conversion instructions.

Table C3-114 Floating-point to integer or fixed-point conversion instructions

Mnemonic	Instruction	See
`FCVTAS`	Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar form)	FCVTAS (scalar)
	Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar SIMD&FP form)	FCVTAS (scalar SIMD&FP)
`FCVTAU`	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar form)	FCVTAU (scalar)
	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar SIMD&FP form)	FCVTAU (scalar SIMD&FP)
`FCVTMS`	Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar form)	FCVTMS (scalar)
	Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar SIMD&FP form)	FCVTMS (scalar SIMD&FP)
`FCVTMU`	Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar form)	FCVTMU (scalar)
	Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar SIMD&FP form)	FCVTMU (scalar SIMD&FP)
`FCVTNS`	Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar form)	FCVTNS (scalar)
	Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar SIMD&FP form)	FCVTNS (scalar SIMD&FP)
`FCVTNU`	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar form)	FCVTNU (scalar)
	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar SIMD&FP form)	FCVTNU (scalar SIMD&FP)
`FCVTPS`	Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar form)	FCVTPS (scalar)
	Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar SIMD&FP form)	FCVTPS (scalar SIMD&FP)
`FCVTPU`	Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar form)	FCVTPU (scalar)
	Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar SIMD&FP form)	FCVTPU (scalar SIMD&FP)
`FCVTZS`	Floating-point scalar convert to signed integer, rounding toward zero (scalar form)	FCVTZS (scalar, integer)
	Floating-point scalar convert to signed fixed-point, rounding toward zero (scalar form)	FCVTZS (scalar, fixed-point)
	Floating-point scalar convert to signed integer, rounding toward zero (scalar SIMD&FP form)	FCVTZS (scalar SIMD&FP)
`FCVTZU`	Floating-point scalar convert to unsigned integer, rounding toward zero (scalar form)	FCVTZU (scalar, integer)
	Floating-point scalar convert to unsigned fixed-point, rounding toward zero (scalar form)	FCVTZU (scalar, fixed-point)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Floating-point scalar convert to unsigned integer, rounding toward zero (scalar SIMD&FP form)	FCVTZU (scalar SIMD&FP)
FJCVTZS	Floating-point JavaScript convert to signed fixed-point, rounding toward zero (scalar form)	FJCVTZS

C3.8.4.4 Convert from integer or fixed-point to floating-point

The integer or fixed-point to floating-point conversion is supported by the following instruction variants:

- Scalar instruction variant, which converts an integer or fixed-point value in a general-purpose register to a floating-point value in a SIMD and floating-point destination register.
- Scalar SIMD&FP variant, which converts an integer value in a SIMD and floating-point source register to a floating-point value in a SIMD and floating-point destination register.

For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

Table C3-115 shows the integer or fixed-point to floating-point conversion instructions.

Table C3-115 Integer or fixed-point to floating-point conversion instructions

Mnemonic	Instruction	See
SCVTF	Signed integer scalar convert to floating-point, using the current rounding mode (scalar form)	SCVTF (scalar, integer)
	Signed fixed-point convert to floating-point, using the current rounding mode (scalar form)	SCVTF (scalar, fixed-point)
	Signed integer scalar convert to floating-point, using the current rounding mode (scalar SIMD&FP form)	SCVTF (scalar SIMD&FP)
UCVTF	Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar form)	UCVTF (scalar, integer)
	Unsigned fixed-point convert to floating-point, using the current rounding mode (scalar form)	UCVTF (scalar, fixed-point)
	Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar SIMD&FP form)	UCVTF (scalar SIMD&FP)

C3.8.5 Floating-point round to integral value

The following subsections describe instructions which round a floating-point number to an integral valued floating-point number in the same format:

- Floating-point round to an integer of the same size as the register
- Floating-point round to 32-bit or 64-bit integer

C3.8.5.1 Floating-point round to an integer of the same size as the register

The following instructions round a floating-point value to an integer floating-point value of the same size.

For these instructions:

- A zero input gives a zero result with the same sign.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An infinite input gives an infinite result with the same sign.
- A NaN is propagated as in normal floating-point arithmetic.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-116 shows the Floating-point round to integer instructions.

Table C3-116 Floating-point round to integer instructions

Mnemonic	Instruction	See
FRINTA	Floating-point round to integer, to nearest with ties to away	FRINTA (scalar)
FRINTI	Floating-point round to integer, using current rounding mode	FRINTI (scalar)
FRINTM	Floating-point round to integer, toward minus infinity	FRINTM (scalar)
FRINTN	Floating-point round to integer, to nearest with ties to even	FRINTN (scalar)
FRINTP	Floating-point round to integer, toward positive infinity	FRINTP (scalar)
FRINTX	Floating-point round to integer exact, using current rounding mode	FRINTX (scalar)
FRINTZ	Floating-point round to integer, toward zero	FRINTZ (scalar)

C3.8.5.2 Floating-point round to 32-bit or 64-bit integer

The following instructions are present if FEAT_FRINTTS is implemented, The instructions round to a value that fits in a 32-bit integer or a 64-bit integer size, and use either round towards zero or the ambient rounding model.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-117 shows the Floating-point round to 32-bit or 64-bit integer instructions.

Table C3-117 Floating-point round to integer instructions

Mnemonic	Instruction	See
FRINT32X	Floating-point round to 32-bit integer, using current rounding model	FRINT32X (scalar)
FRINT32Z	Floating-point round to 32-bit integer, toward zero	FRINT32Z (scalar)
FRINT64X	Floating point round to 64-bit integer, using current rounding model	FRINT64X (scalar)
FRINT64Z	Floating point round to 64-bit integer, toward zero	FRINT64Z (scalar)

C3.8.6 Floating-point multiply-add

Table C3-118 shows the Floating-point multiply-add instructions that require three source register operands.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-118 Floating-point multiply-add instructions

Mnemonic	Instruction	See
FMADD	Floating-point scalar fused multiply-add	FMADD
FMSUB	Floating-point scalar fused multiply-subtract	FMSUB
FNMADD	Floating-point scalar negated fused multiply-add	FNMADD
FNMSUB	Floating-point scalar negated fused multiply-subtract	FNMSUB

C3.8.7 Floating-point arithmetic (one source)

Table C3-119 shows the Floating-point arithmetic instructions that require a single source register operand.

Table C3-119 Floating-point arithmetic instructions with one source register

Mnemonic	Instructions	See
FABS	Floating-point scalar absolute value	FABS (scalar)
FNEG	Floating-point scalar negate	FNEG (scalar)
FSQRT	Floating-point scalar square root	FSQRT (scalar)

C3.8.8 Floating-point arithmetic (two sources)

Table C3-120 shows the Floating-point arithmetic instructions that require two source register operands.

Table C3-120 Floating-point arithmetic instructions with two source registers

Mnemonic	Instruction	See
FADD	Floating-point scalar add	FADD (scalar)
FDIV	Floating-point scalar divide	FDIV (scalar)
FMUL	Floating-point scalar multiply	FMUL (scalar)
FNMUL	Floating-point scalar multiply-negate	FNMUL (scalar)
FSUB	Floating-point scalar subtract	FSUB (scalar)

C3.8.9 Floating-point minimum and maximum

The `FPMin(x,y)` and `FPMax(x,y)` operations return a quiet NaN when either *x* or *y* is NaN.

As described in Flushing denormalized numbers to zero, if flushing denormalized inputs to zero is enabled, denormal operands are flushed to zero before comparison, and if the result of the comparison is the flushed value, then a zero value is returned. Where both *x* and *y* are zero, or denormal values flushed to zero, with different signs, then +0.0 is returned by `FPMax()` and -0.0 by `FPMin()`.

The `FPMinNum(x,y)` and `FPMaxNum(x,y)` operations follow the IEEE 754-2008 standard and return the numerical operand when one operand is numerical and the other a quiet NaN. Apart from this additional handling of a single quiet NaN, the result is then identical to `FPMin(x,y)` and `FPMax(x,y)`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-121 shows the Floating-point instructions that can perform floating-point minimum and maximum operations.

Table C3-121 Floating-point minimum and maximum instructions

Mnemonic	Instruction	See
FMAX	Floating-point scalar maximum	FMAX (scalar)
FMAXNM	Floating-point scalar maximum number	FMAXNM (scalar)
FMIN	Floating-point scalar minimum	FMIN (scalar)
FMINNM	Floating-point scalar minimum number	FMINNM (scalar)

C3.8.10 Floating-point comparison

These instructions set the NZCV Condition flags in PSTATE, based on the result of a comparison of two operands. If the floating-point comparisons are *unordered*, where one or both operands are a form of NaN, the C and V bits are set to 1 and the N and Z bits are cleared to 0.

Note

The NZCV flags in the FPSR are associated with AArch32 state. The A64 floating-point comparison instructions do not change the Condition flags in the FPSR.

For the conditional Floating-point comparison instructions, if the condition is TRUE, the flags are updated to the result of the comparison, otherwise the flags are updated to the immediate value that is defined in the instruction encoding.

The quiet compare instructions generate an Invalid Operation floating-point exception if either of the source operands is a signaling NaN. The signaling compare instructions generate an Invalid Operation floating-point exception if either of the source operands is any type of NaN.

Note

If FEAT_FlagM2 is implemented, instructions AXFLAG and XAFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See FEAT_FlagM for more information.

Table C3-122 shows the Floating-point comparison instructions.

Table C3-122 Floating-point comparison instructions

Mnemonic	Instruction	See
FCMP	Floating-point quiet compare	FCMP
FCMPE	Floating-point signaling compare	FCMPE
FCCMP	Floating-point conditional quiet compare	FCCMP
FCCMPE	Floating-point conditional signaling compare	FCCMPE

C3.8.11 Floating-point conditional select

Table C3-123 shows the Floating-point conditional select instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-123 Floating-point conditional select instructions

Mnemonic	Instruction	See
FCSEL	Floating-point scalar conditional select	FCSEL

C3.8.12 SIMD move

The functionality of some data movement instructions overlaps with that provided by the scalar floating-point FMOV instructions described in Floating-point move (register).

Table C3-124 shows the SIMD move instructions.

Table C3-124 SIMD move instructions

Mnemonic	Instruction	See
DUP	Duplicate vector element to vector or scalar	DUP (element)
DUP	Duplicate general-purpose register to vector	DUP (general)
INS[a]	Insert vector element from another vector element	INS (element)
	Insert vector element from general-purpose register	INS (general)
MOV	Move vector element to vector element	MOV (element)
	Move general-purpose register to vector element	MOV (from general)
	Move vector element to scalar	MOV (scalar)
	Move vector element to general-purpose register	MOV (to general)
UMOV	Unsigned move vector element to general-purpose register	UMOV
SMOV	Signed move vector element to general-purpose register	SMOV

a. Disassembles as MOV.

C3.8.13 SIMD arithmetic

Table C3-125 shows the SIMD arithmetic instructions.

Table C3-125 SIMD arithmetic instructions

Mnemonic	Instruction	See
ADD	Add (vector and scalar form)	ADD (vector)
AND	Bitwise AND (vector form)	AND (vector)
BIC	Bitwise bit clear (register) (vector form)	BIC (vector, register)
BIF	Bitwise insert if false (vector form)	BIF
BIT	Bitwise insert if true (vector form)	BIT
BSL	Bitwise select (vector form)	BSL
EOR	Bitwise exclusive-OR (vector form)	EOR (vector)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FABD	Floating-point absolute difference (vector and scalar form)	FABD
FADD	Floating-point add (vector form)	FADD (vector)
FAMAX	Floating-point absolute maximum	FAMAX
FAMIN	Floating-point absolute minimum	FAMIN
FDIV	Floating-point divide (vector form)	FDIV (vector)
FMAX	Floating-point maximum (vector form)	FMAX (vector)
FMAXNM	Floating-point maximum number (vector form)	FMAXNM (vector)
FMIN	Floating-point minimum (vector form)	FMIN (vector)
FMINNM	Floating-point minimum number (vector form)	FMINNM (vector)
FMUL	Floating-point multiply (vector form)	FMUL (vector)
FMULX	Floating-point multiply extended (vector and scalar form)	FMULX
FRECPS	Floating-point reciprocal step (vector and scalar form)	FRECPS
FRSQRTS	Floating-point reciprocal square root step (vector and scalar form)	FRSQRTS
FSCALE	Floating-point adjust exponent by vector	FSCALE
FSUB	Floating-point subtract (vector form)	FSUB (vector)
MLA	Multiply-add (vector form)	MLA (vector)
MLS	Multiply-subtract (vector form)	MLS (vector)
MUL	Multiply (vector form)	MUL (vector)
MOV	Move vector register (vector form)	MOV (vector)
ORN	Bitwise inclusive OR NOT (vector form)	ORN (vector)
ORR	Bitwise inclusive OR (register) (vector form)	ORR (vector, register)
PMUL	Polynomial multiply (vector form)	PMUL
SABA	Signed absolute difference and accumulate (vector form)	SABA
SABD	Signed absolute difference (vector form)	SABD
SHADD	Signed halving add (vector form)	SHADD
SHSUB	Signed halving subtract (vector form)	SHSUB
SMAX	Signed maximum (vector form)	SMAX
SMIN	Signed minimum (vector form)	SMIN
SQADD	Signed saturating add (vector and scalar form)	SQADD
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	SQDMULH (vector)
SQRSHL	Signed saturating rounding shift left (register) (vector and scalar form)	SQRSHL
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	SQRDMLAH (vector)
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	SQRDMLSH (vector)
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	SQRDMULH (vector)
SQSHL	Signed saturating shift left (register) (vector and scalar form)	SQSHL (register)
SQSUB	Signed saturating subtract (vector and scalar form)	SQSUB
SRHADD	Signed rounding halving add (vector form)	SRHADD
SRSHL	Signed rounding shift left (register) (vector and scalar form)	SRSHL
SSHL	Signed shift left (register) (vector and scalar form)	SSHL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SUB	Subtract (vector and scalar form)	SUB (vector)
UABA	Unsigned absolute difference and accumulate (vector form)	UABA
UABD	Unsigned absolute difference (vector form)	UABD
UHADD	Unsigned halving add (vector form)	UHADD
UHSUB	Unsigned halving subtract (vector form)	UHSUB
UMAX	Unsigned maximum (vector form)	UMAX
UMIN	Unsigned minimum (vector form)	UMIN
UQADD	Unsigned saturating add (vector and scalar form)	UQADD
UQRSHL	Unsigned saturating rounding shift left (register) (vector and scalar form)	UQRSHL
UQSHL	Unsigned saturating shift left (register) (vector and scalar form)	UQSHL (register)
UQSUB	Unsigned saturating subtract (vector and scalar form)	UQSUB
URHADD	Unsigned rounding halving add (vector form)	URHADD
URSHL	Unsigned rounding shift left (register) (vector and scalar form)	URSHL
USHL	Unsigned shift left (register) (vector and scalar form)	USHL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.14 SIMD compare

The SIMD compare instructions compare vector or scalar elements according to the specified condition and set the destination vector element to all ones if the condition holds, or to zero if the condition does not hold.

Note

Some of the comparisons, such as LS, LE, LO, and LT, can be made by reversing the operands and using the opposite comparison, HS, GE, HI, or GT.

Table C3-126 shows that SIMD compare instructions.

Table C3-126 SIMD compare instructions

Mnemonic	Instruction	See
`CMEQ`	Compare bitwise equal (vector and scalar form)	CMEQ (register)
	Compare bitwise equal to zero (vector and scalar form)	CMEQ (zero)
`CMHS`	Compare unsigned higher or same (vector and scalar form)	CMHS (register)
`CMGE`	Compare signed greater than or equal (vector and scalar form)	CMGE (register)
	Compare signed greater than or equal to zero (vector and scalar form)	CMGE (zero)
`CMHI`	Compare unsigned higher (vector and scalar form)	CMHI (register)
`CMGT`	Compare signed greater than (vector and scalar form)	CMGT (register)
	Compare signed greater than zero (vector and scalar form)	CMGT (zero)
`CMLE`	Compare signed less than or equal to zero (vector and scalar form)	CMLE (zero)
`CMLT`	Compare signed less than zero (vector and scalar form)	CMLT (zero)
`CMTST`	Compare bitwise test bits nonzero (vector and scalar form)	CMTST
`FCMEQ`	Floating-point compare equal (vector and scalar form)	FCMEQ (register)
	Floating-point compare equal to zero (vector and scalar form)	FCMEQ (zero)
`FCMGE`	Floating-point compare greater than or equal (vector and scalar form)	FCMGE (register)
	Floating-point compare greater than or equal to zero (vector and scalar form)	FCMGE (zero)
`FCMGT`	Floating-point compare greater than (vector and scalar form)	FCMGT (register)
	Floating-point compare greater than zero (vector and scalar form)	FCMGT (zero)
`FCMLE`	Floating-point compare less than or equal to zero (vector and scalar form)	FCMLE (zero)
`FCMLT`	Floating-point compare less than zero (vector and scalar form)	FCMLT (zero)
`FACGE`	Floating-point absolute compare greater than or equal (vector and scalar form)	FACGE
`FACGT`	Floating-point absolute compare greater than (vector and scalar form)	FACGT

C3.8.15 SIMD widening and narrowing arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-127 shows the SIMD widening and narrowing arithmetic instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-127 SIMD widening and narrowing arithmetic instructions

Mnemonic	Instruction	See
ADDHN, ADDHN2	Add returning high, narrow (vector form)	ADDHN, ADDHN2
PMULL, PMULL2	Polynomial multiply long (vector form)	PMULL, PMULL2
RADDHN, RADDHN2	Rounding add returning high, narrow (vector form)	RADDHN, RADDHN2
RSUBHN, RSUBHN2	Rounding subtract returning high, narrow (vector form)	RSUBHN, RSUBHN2
SABAL, SABAL2	Signed absolute difference and accumulate long (vector form)	SABAL, SABAL2
SABDL, SABDL2	Signed absolute difference long (vector form)	SABDL, SABDL2
SADDL, SADDL2	Signed add long (vector form)	SADDL, SADDL2
SADDW, SADDW2	Signed add wide (vector form)	SADDW, SADDW2
SMLAL, SMLAL2	Signed multiply-add long (vector form)	SMLAL, SMLAL2 (vector)
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	SMLSL, SMLSL2 (vector)
SMULL, SMULL2	Signed multiply long (vector form)	SMULL, SMULL2 (vector)
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	SQDMLAL, SQDMLAL2 (vector)
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector and scalar form)	SQDMLSL, SQDMLSL2 (vector)
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	SQDMULL, SQDMULL2 (vector)
SSUBL, SSUBL2	Signed subtract long (vector form)	SSUBL, SSUBL2
SSUBW, SSUBW2	Signed subtract wide (vector form)	SSUBW, SSUBW2
SUBHN, SUBHN2	Subtract returning high, narrow (vector form)	SUBHN, SUBHN2
UABAL, UABAL2	Unsigned absolute difference and accumulate long (vector form)	UABAL, UABAL2
UABDL, UABDL2	Unsigned absolute difference long (vector form)	UABDL, UABDL2
UADDL, UADDL2	Unsigned add long (vector form)	UADDL, UADDL2
UADDW, UADDW2	Unsigned add wide (vector form)	UADDW, UADDW2
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	UMLAL, UMLAL2 (vector)
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	UMLSL, UMLSL2 (vector)
UMULL, UMULL2	Unsigned multiply long (vector form)	UMULL, UMULL2 (vector)
USUBL, USUBL2	Unsigned subtract long (vector form)	USUBL, USUBL2
USUBW, USUBW2	Unsigned subtract wide (vector form)	USUBW, USUBW2

C3.8.16 SIMD unary arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-128 shows the SIMD unary arithmetic instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-128 SIMD unary arithmetic instructions

Mnemonic	Instruction	See
ABS	Absolute value (vector and scalar form)	ABS
CLS	Count leading sign bits (vector form)	CLS (vector)
CLZ	Count leading zero bits (vector form)	CLZ (vector)
CNT	Population count per byte (vector form)	CNT
FABS	Floating-point absolute (vector form)	FABS (vector)
FNEG	Floating-point negate (vector form)	FNEG (vector)
FRECPE	Floating-point reciprocal estimate (vector and scalar form)	FRECPE
FRECPX	Floating-point reciprocal exponent (scalar form)	FRECPX
FRINT32X	Floating-point round to 32-bit integer, using current rounding mode (vector form)	FRINT32X (vector)
FRINT32Z	Floating-point round to 32-bit integer, toward zero (vector form)	FRINT32Z (vector)
FRINT64X	Floating-point round to 64-bit integer, using current rounding mode (vector form)	FRINT64X (vector)
FRINT64Z	Floating-point round to 64-bit integer, toward zero (vector form)	FRINT64Z (vector)
FRINTA	Floating-point round to integer, to nearest with ties to away (vector form)	FRINTA (vector)
FRINTI	Floating-point round to integer, using current rounding mode (vector form)	FRINTI (vector)
FRINTM	Floating-point round to integer, toward minus infinity (vector form)	FRINTM (vector)
FRINTN	Floating-point round to integer, to nearest with ties to even (vector form)	FRINTN (vector)
FRINTP	Floating-point round to integer, toward positive infinity (vector form)	FRINTP (vector)
FRINTX	Floating-point round to integer exact, using current rounding mode (vector form)	FRINTX (vector)
FRINTZ	Floating-point round to integer, toward zero (vector form)	FRINTZ (vector)
FRSQRTE	Floating-point reciprocal square root estimate (vector and scalar form)	FRSQRTE
FSQRT	Floating-point square root (vector form)	FSQRT (vector)
MVN	Bitwise NOT (vector form)	MVN
NEG	Negate (vector and scalar form)	NEG (vector)
NOT	Bitwise NOT (vector form)	NOT
RBIT	Bitwise reverse (vector form)	RBIT (vector)
REV16	Reverse elements in 16-bit halfwords (vector form)	REV16 (vector)
REV32	Reverse elements in 32-bit words (vector form)	REV32 (vector)
REV64	Reverse elements in 64-bit doublewords (vector form)	REV64
SADALP	Signed add and accumulate long pairwise (vector form)	SADALP
SADDLP	Signed add long pairwise (vector form)	SADDLP
SQABS	Signed saturating absolute value (vector and scalar form)	SQABS
SQNEG	Signed saturating negate (vector and scalar form)	SQNEG
SQXTN, SQXTN2	Signed saturating extract narrow (vector form)	SQXTN, SQXTN2
SQXTUN, SQXTUN2	Signed saturating extract unsigned narrow (vector and scalar form)	SQXTUN, SQXTUN2
SUQADD	Signed saturating accumulate of unsigned value (vector and scalar form)	SUQADD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SXTL, SXTL2	Signed extend long	SXTL, SXTL2
UADALP	Unsigned add and accumulate long pairwise (vector form)	UADALP
UADDLP	Unsigned add long pairwise (vector form)	UADDLP
UQXTN, UQXTN2	Unsigned saturating extract narrow (vector form)	UQXTN, UQXTN2
URECPE	Unsigned reciprocal estimate (vector form)	URECPE
URSQRTE	Unsigned reciprocal square root estimate (vector form)	URSQRTE
USQADD	Unsigned saturating accumulate of signed value (vector and scalar form)	USQADD
UXTL, UXTL2	Unsigned extend long	UXTL, UXTL2
XTN, XTN2	Extract narrow (vector form)	XTN, XTN2

C3.8.17 SIMD by element arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-129 shows the SIMD by element arithmetic instructions.

Table C3-129 SIMD by element arithmetic instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add (vector and scalar form)	FMLA (by element)
FMLAL, FMLAL2	Floating-point fused multiply-add long (vector form)	FMLAL, FMLAL2 (by element)
FMLS	Floating-point fused multiply-subtract (vector and scalar form)	FMLS (by element).
FMLSL, FMLSL2	Floating-point fused multiply-subtract long (vector form)	FMLSL, FMLSL2 (by element)
FMUL	Floating-point multiply (vector and scalar form)	FMUL (by element)
FMULX	Floating-point multiply extended (vector and scalar form)	FMULX (by element)
MLA	Multiply-add (vector form)	MLA (by element)
MLS	Multiply-subtract (vector form)	MLS (by element)
MUL	Multiply (vector form)	MUL (by element)
SMLAL, SMLAL2	Signed multiply-add long (vector form)	SMLAL, SMLAL2 (by element)
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	SMLSL, SMLSL2 (by element)
SMULL, SMULL2	Signed multiply long (vector form)	SMULL, SMULL2 (by element)
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	SQDMLAL, SQDMLAL2 (by element)
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector form)	SQDMLSL, SQDMLSL2 (by element)
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	SQDMULH (by element)
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	SQDMULL, SQDMULL2 (by element)
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	SQRDMLSH (by element)
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	SQRDMLSH (vector)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	SQRDMULH (by element)
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	UMLAL, UMLAL2 (by element)
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	UMLSL, UMLSL2 (by element)
UMULL, UMULL2	Unsigned multiply long (vector form)	UMULL, UMULL2 (by element)

C3.8.18 SIMD permute

Table C3-130 shows the SIMD permute instructions.

Table C3-130 SIMD permute instructions

Mnemonic	Instruction	See
EXT	Extract vector from a pair of vectors	EXT
TRN1	Transpose vectors (primary)	TRN1
TRN2	Transpose vectors (secondary)	TRN2
UZP1	Unzip vectors (primary)	UZP1
UZP2	Unzip vectors (secondary)	UZP2
ZIP1	Zip vectors (primary)	ZIP1
ZIP2	Zip vectors (secondary)	ZIP2

C3.8.19 SIMD immediate

Table C3-131 shows the SIMD immediate instructions.

Table C3-131 SIMD immediate instructions

Mnemonic	Instruction	See
BIC	Bitwise bit clear immediate	BIC (vector, immediate)
FMOV	Floating-point move immediate	FMOV (vector, immediate)
MOVI	Move immediate	MOVI
MVNI	Move inverted immediate	MVNI
ORR	Bitwise inclusive OR immediate	ORR (vector, immediate)

C3.8.20 SIMD shift (immediate)

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-132 shows the SIMD shift immediate instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-132 SIMD shift (immediate) instructions

Mnemonic	Instruction	See
RSHRN, RSHRN2	Rounding shift right narrow immediate (vector form)	RSHRN, RSHRN2
SHL	Shift left immediate (vector and scalar form)	SHL
SHLL, SHLL2	Shift left long (by element size) (vector form)	SHLL, SHLL2
SHRN, SHRN2	Shift right narrow immediate (vector form)	SHRN, SHRN2
SLI	Shift left and insert immediate (vector and scalar form)	SLI
SQRSHRN, SQRSHRN2	Signed saturating rounded shift right narrow immediate (vector and scalar form)	SQRSHRN, SQRSHRN2
SQRSHRUN, SQRSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	SQRSHRUN, SQRSHRUN2
SQSHL	Signed saturating shift left immediate (vector and scalar form)	SQSHL (immediate)
SQSHLU	Signed saturating shift left unsigned immediate (vector and scalar form)	SQSHLU
SQSHRN, SQSHRN2	Signed saturating shift right narrow immediate (vector and scalar form)	SQSHRN, SQSHRN2
SQSHRUN, SQSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	SQSHRUN, SQSHRUN2
SRI	Shift right and insert immediate (vector and scalar form)	SRI
SRSHR	Signed rounding shift right immediate (vector and scalar form)	SRSHR
SRSRA	Signed rounding shift right and accumulate immediate (vector and scalar form)	SRSRA.
SSHLL, SSHLL2	Signed shift left long immediate (vector form)	SSHLL, SSHLL2
SSHR	Signed shift right immediate (vector and scalar form)	SSHR
SSRA	Signed integer shift right and accumulate immediate (vector and scalar form)	SSRA
SXTL, SXTL2	Signed integer extend (vector only)	SXTL, SXTL2
UQRSHRN, UQRSHRN2	Unsigned saturating rounded shift right narrow immediate (vector and scalar form)	UQRSHRN, UQRSHRN2
UQSHL	Unsigned saturating shift left immediate (vector and scalar form)	UQSHL (immediate)
UQSHRN, UQSHRN2	Unsigned saturating shift right narrow immediate (vector and scalar form)	UQSHRN, UQSHRN2
URSHR	Unsigned rounding shift right immediate (vector and scalar form)	URSHR
URSRA	Unsigned integer rounding shift right and accumulate immediate (vector and scalar form)	URSRA
USHLL, USHLL2	Unsigned shift left long immediate (vector form)	USHLL, USHLL2
USHR	Unsigned shift right immediate (vector and scalar form)	USHR
USRA	Unsigned shift right and accumulate immediate (vector and scalar form)	USRA
UXTL, UXTL2	Unsigned integer extend (vector only)	UXTL, UXTL2

C3.8.21 SIMD conversions

C3.8.21.1 Floating-point conversion

Table C3-133 shows the SIMD floating-point conversion instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-133 SIMD floating-point conversion instructions

Mnemonic	Instruction	See
FCVTAS	Floating-point convert to signed integer, rounding to nearest with ties to away (vector and scalar form)	FCVTAS (vector)
FCVTAU	Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector and scalar form)	FCVTAU (vector)
FCVTL, FCVTL2	Floating-point convert to higher precision long (vector form)	FCVTL, FCVTL2
FCVTMS	Floating-point convert to signed integer, rounding toward minus infinity (vector and scalar form)	FCVTMS (vector)
FCVTMU	Floating-point convert to unsigned integer, rounding toward minus infinity (vector and scalar form)	FCVTMU (vector)
FCVTN, FCVTN2	Floating-point convert to lower precision narrow (vector form)	FCVTN, FCVTN2 (double to single-precision, single to half-precision)
FCVTNS	Floating-point convert to signed integer, rounding to nearest with ties to even (vector and scalar form)	FCVTNS (vector)
FCVTNU	Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector and scalar form)	FCVTNU (vector)
FCVTPS	Floating-point convert to signed integer, rounding toward positive infinity (vector and scalar form)	FCVTPS (vector)
FCVTPU	Floating-point convert to unsigned integer, rounding toward positive infinity (vector and scalar form)	FCVTPU (vector)
FCVTXN, FCVTXN2	Floating-point convert to lower precision narrow, rounding to odd (vector and scalar form)	FCVTXN, FCVTXN2
FCVTZS	Floating-point convert to signed integer, rounding toward zero (vector and scalar form)	FCVTZS (vector, integer)
	Floating-point convert to signed fixed-point, rounding toward zero (vector and scalar form)	FCVTZS (vector, fixed-point)
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero (vector and scalar form)	FCVTZU (vector, integer)
	Floating-point convert to unsigned fixed-point, rounding toward zero, (vector and scalar form)	FCVTZU (vector, fixed-point)

C3.8.21.2 Integer conversion

Table C3-134 shows the SIMD integer conversion instructions.

Table C3-134 SIMD integer conversion instructions

Mnemonic	Instruction	See
SCVTF	Signed integer convert to floating-point (vector and scalar form)	SCVTF (vector, integer)
	Signed fixed-point convert to floating-point (vector and scalar form)	SCVTF (vector, fixed-point)
UCVTF	Unsigned integer convert to floating-point (vector and scalar form)	UCVTF (vector, integer)
	Unsigned fixed-point convert to floating-point (vector and scalar form)	UCVTF (vector, fixed-point)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.21.3 FP8 floating-point conversion

Convert each floating-point element of the source vector while scaling the value, and place the results in the corresponding elements of the destination vector.

For instructions that convert other floating-point formats to FP8 formats, each input element is scaled by adding the value specified in FPMR.NSCALE to its unbounded exponent during the conversion.

For instructions that convert FP8 formats to other floating-point formats, each input element is scaled by subtracting the value specified in FPMR.LSCALE or FPMR.LSCALE2 from its unbounded exponent during the conversion.

For more information, see the instruction descriptions.

Table C3-135 shows the SIMD FP8 floating-point conversion instructions.

Table C3-135 SIMD FP8 floating-point conversion instructions

Mnemonic	Instruction	See
`BF1CVTL`, `BF1CVTL2`, `BF2CVTL`, `BF2CVTL2`	8-bit floating-point convert to BFloat16 (vector)	BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2
`F1CVTL`, `F1CVTL2`, `F2CVTL`, `F2CVTL2`	8-bit floating-point convert to half-precision (vector)	F1CVTL, F1CVTL2, F2CVTL, F2CVTL2
`FCVTN`	Half-precision to 8-bit floating-point convert and narrow (vector)	FCVTN (half-precision to 8-bit floating-point)
`FCVTN`, `FCVTN2`	Single-precision to 8-bit floating-point convert and narrow (vector)	FCVTN, FCVTN2 (single-precision to 8-bit floating-point)

C3.8.22 SIMD reduce (across vector lanes)

The SIMD reduce (across vector lanes) instructions perform arithmetic operations horizontally, that is across all lanes of the input vector. They deliver a single scalar result.

Table C3-136 shows the SIMD reduce (across vector lanes) instructions.

Table C3-136 SIMD reduce (across vector lanes) instructions

Mnemonic	Instruction	See
`ADDV`	Add (across vector)	ADDV
`FMAXNMV`	Floating-point maximum number (across vector)	FMAXNMV
`FMAXV`	Floating-point maximum (across vector)	FMAXV
`FMINNMV`	Floating-point minimum number (across vector)	FMINNMV
`FMINV`	Floating-point minimum (across vector)	FMINV
`SADDLV`	Signed add long (across vector)	SADDLV
`SMAXV`	Signed maximum (across vector)	SMAXV
`SMINV`	Signed minimum (across vector)	SMINV

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
UADDLV	Unsigned add long (across vector)	UADDLV
UMAXV	Unsigned maximum (across vector)	UMAXV
UMINV	Unsigned minimum (across vector)	UMINV

C3.8.23 SIMD pairwise arithmetic

The SIMD pairwise arithmetic instructions perform operations on pairs of adjacent elements and deliver a vector result.

Table C3-137 shows the SIMD pairwise arithmetic instructions.

Table C3-137 SIMD pairwise arithmetic instructions

Mnemonic	Instruction	See
ADDP	Add pairwise (vector and scalar form)	ADDP (vector) ADDP (scalar)
FADDP	Floating-point add pairwise (vector and scalar form)	FADDP (vector) FADDP (scalar)
FMAXNMP	Floating-point maximum number pairwise (vector and scalar form)	FMAXNMP (vector) FMAXNMP (scalar)
FMAXP	Floating-point maximum pairwise (vector and scalar form)	FMAXP (vector) FMAXP (scalar)
FMINNMP	Floating-point minimum number pairwise (vector and scalar form)	FMINNMP (vector) FMINNMP (scalar)
FMINP	Floating-point minimum pairwise (vector and scalar form)	FMINP (vector) FMINP (scalar)
SMAXP	Signed maximum pairwise	SMAXP
SMINP	Signed minimum pairwise	SMINP
UMAXP	Unsigned maximum pairwise	UMAXP
UMINP	Unsigned minimum pairwise	UMINP

C3.8.24 SIMD dot product

C3.8.24.1 Integer dot product

FEAT_DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second vector.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note

That is, a single element from the second vector is used, and the dot product is calculated between each element of the first vector and this single element from the second vector.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-138 shows the SIMD integer dot product instructions.

Table C3-138 SIMD integer dot product instructions

Mnemonic	Instruction	See
SDOT	Signed dot product (vector form)	SDOT (vector)
UDOT	Unsigned dot product (vector form)	UDOT (vector)
SDOT	Signed dot product (indexed form)	SDOT (by element)
UDOT	Unsigned dot product (indexed form)	UDOT (by element)
USDOT	Mixed sign integer dot product (vector form)[a]	USDOT (vector)
	Mixed sign integer dot product by indexed quadruplet[a]	USDOT (by element)
SUDOT	Mixed sign integer dot product by indexed quadruplet[a]	SUDOT (by element)

a. This instruction is supported when FEAT_I8MM is implemented.

C3.8.24.2 FP8 floating-point dot product

The FP8 floating-point dot product instructions accumulate the dot product of two or four adjacent FP8 source elements into half-precision or single-precision destination elements. They generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-139 shows these instructions.

Table C3-139 SIMD FP8 floating-point dot product instructions

Mnemonic	Instruction	See
FDOT	8-bit floating-point dot product to half-precision (vector, by element)	FDOT (8-bit floating-point to half-precision, by element)
	8-bit floating-point dot product to half-precision (vector)	FDOT (8-bit floating-point to half-precision, vector)
	8-bit floating-point dot product to single-precision (vector, by element)	FDOT (8-bit floating-point to single-precision, by element)
	8-bit floating-point dot product to single-precision (vector)	FDOT (8-bit floating-point to single-precision, vector)

C3.8.25 SIMD matrix multiply-accumulate

C3.8.25.1 Integer matrix multiply-accumulate

The integer matrix multiply-accumulate instructions are implemented by FEAT_I8MM.

The source and destination vectors of this instruction are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit integers in row-by-row order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In the second source vector, an 8x2 matrix of 8-bit integers in column-by-column order.
- In the destination vector, a 2x2 matrix of 32-bit integers in row-by-row order.

One matrix multiplication is performed per vector and accumulated into the destination vector.

Table C3-140 shows these instructions.

Table C3-140 SIMD integer matrix multiply-accumulate instructions

Mnemonic	Instruction	See
`SMMLA`	Widening signed 8-bit to 32-bit integer matrix multiply-accumulate	SMMLA (vector)
`UMMLA`	Widening unsigned 8-bit to 32-bit integer matrix multiply-accumulate	UMMLA (vector)
`USMMLA`	Widening mixed sign 8-bit to 32-bit integer matrix multiply-accumulate	USMMLA (vector)

C3.8.25.2 FP8 matrix multiply-accumulate

The FP8 `FMMLA` (widening, 8-bit floating-point to single-precision) instruction is implemented by FEAT_F8F32MM. The source and destination vectors of this instruction are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, an 8x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per vector and accumulated into the destination vector.

The FP8 `FMMLA` (widening, 8-bit floating-point to half-precision) instruction is implemented by FEAT_F8F16MM. This instruction delimits each source and destination vector into 64-bit segments. The 64-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, a 4x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of half-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

These instructions generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the intermediate sum of products is downscaled by subtracting the value in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-141 SIMD FP8 matrix multiply-accumulate instructions

Mnemonic	Instruction	See
`FMMLA`	8-bit floating-point matrix multiply-accumulate into 2x2 single-precision matrix	FMMLA (widening, 8-bit floating-point to single-precision)
`FMMLA`	8-bit floating-point matrix multiply-accumulate into 2x2 half-precision matrix	FMMLA (widening, 8-bit floating-point to half-precision)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.26 SIMD table lookup

C3.8.26.1 Table vector lookup

Table C3-142 shows the SIMD table vector lookup instructions.

Table C3-142 SIMD table vector lookup instructions

Mnemonic	Instruction	See
TBL	Table vector lookup	TBL
TBX	Table vector lookup extension	TBX

C3.8.26.2 Lookup table using 2-bit or 4-bit indices

Copy indexed elements from the first source vector(s) to the destination vector, using packed indices from a portion of the second source vector.

Table C3-143 shows the SIMD lookup table instructions.

Table C3-143 SIMD lookup table using 2-bit or 4-bit indices instructions

Mnemonic	Instruction	See
LUTI2	Lookup table using 2-bit indices	LUTI2
LUTI4	Lookup table using 4-bit indices	LUTI4

C3.8.27 SIMD complex number arithmetic instructions

FEAT_FCMA provides SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide double-precision and single-precision versions. If FEAT_FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Table C3-144 shows the FEAT_FCMA SIMD instructions.

Table C3-144 SIMD complex number arithmetic instructions

Mnemonic	Instruction	See
FCADD	Floating-point complex add	FCADD
FCMLA	Floating-point complex multiply accumulate (vector form)	FCMLA
FCMLA	Floating-point complex multiply accumulate (indexed form)	FCMLA (by element)

A pair of FCMLA instructions can be used to perform a complex number multiplication. This is demonstrated in Complex multiplication.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.28 SIMD multiply-accumulate

C3.8.28.1 Floating-point fused multiply-accumulate

The SIMD floating-point fused multiply-accumulate instructions multiply the vector elements in the first source SIMD&FP register by the corresponding value in the second source SIMD&FP register, and accumulate or subtract the product in the vector elements of the destination SIMD&FP register.

Table C3-145 shows these instructions.

Table C3-145 SIMD floating-point fused multiply-accumulate instructions

Mnemonic	Instruction	See
`FMLA`	Floating-point fused multiply-add (vector form)	FMLA (vector)
`FMLAL`, `FMLAL2`	Floating-point fused multiply-add long (vector form)	FMLAL, FMLAL2 (vector)
`FMLS`	Floating-point fused multiply-subtract (vector form)	FMLS (vector)
`FMLSL`, `FMLSL2`	Floating-point fused multiply-subtract long (vector form)	FMLSL, FMLSL2 (vector)

C3.8.28.2 FP8 floating-point widening multiply-accumulate

The FP8 floating-point multiply-accumulate instructions widen the 8-bit floating-point elements in the source vectors to half-precision or single-precision format and multiply the corresponding elements.

They generate a fused, downscaled product and accumulate into the corresponding elements of the destination vector, without intermediate rounding.

For these instructions, the unbounded intermediate product is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-146 shows these instructions.

Table C3-146 SIMD FP8 floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
`FMLALB, FMLALT`	8-bit floating-point multiply-add long to half-precision (vector, by element)	FMLALB, FMLALT (by element)
	8-bit floating-point multiply-add long to half-precision (vector)	FMLALB, FMLALT (vector)
`FMLALLBB`, `FMLALLBT`, `FMLALLTB`, `FMLALLTT`	8-bit floating-point multiply-add long long to single-precision (vector, by element)	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element)
	8-bit floating-point multiply-add long long to single-precision (vector)	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector)

C3.8.29 SIMD BFloat16

If FEAT_BF16 is implemented, the instructions in this section are available.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.29.1 SIMD BFloat16 floating-point multiply-add

The BFloat16 floating-point multiply-add instructions perform an implicit conversion of the bottom (even-numbered) or top (odd-numbered) BFloat16 source elements to IEEE 754 single-precision floating-point format before performing a fused multiply-add without intermediate rounding to the overlapping single-precision destination element. These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Table C3-147 SIMD BFloat16 floating-point multiply-add instructions

Mnemonic	Instruction	See
`BFMLALB`	BFloat16 floating-point widening multiply-add long bottom (vector and indexed forms)	BFMLALB, BFMLALT (vector) BFMLALB, BFMLALT (by element)
`BFMLALT`	BFloat16 floating-point widening multiply-add long top (vector and index forms)	BFMLALB, BFMLALT (vector) BFMLALB, BFMLALT (by element)

C3.8.29.2 SIMD BFloat16 floating-point dot product

The BFloat16 floating-point dot product instruction, `BFDOT`, performs an implicit conversion of vectors of BFloat16 input values to IEEE 754 single-precision floating-point format.

The BFloat16 dot product instructions delimit their source vectors into pairs of BFloat16 elements.

Table C3-148 SIMD BFloat16 floating-point dot product instructions

Mnemonic	Instruction	See
`BFDOT`	BFloat16 floating-point dot product (vector and indexed forms)	BFDOT (vector) BFDOT (by element)

C3.8.29.3 SIMD BFloat16 floating-point matrix multiply-accumulate

The BFloat16 floating-point matrix multiply-accumulate instruction, `BFMMLA` is implemented by FEAT_BF16.

The source and destination vectors of this instruction are organized as follows:

- In the first source vector, a 2x4 matrix of BFloat16 values in row-by-row order.
- In the second source vector, a 4x2 matrix of BFloat16 values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per source vector and accumulated into the corresponding destination vector. This corresponds to accumulating two 2-way BFloat16 widening dot products into each single-precision destination element, following the numeric behaviors described for `BFDOT` instruction in SIMD BFloat16 floating-point dot product.

Table C3-149 SIMD BFloat16 floating-point matrix multiply instructions

Mnemonic	Instruction	See
`BFMMLA`	BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix	BFMMLA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.29.4 SIMD BFloat16 floating-point convert

The BFloat16 floating-point convert instructions perform accurately rounded down-conversion of single-precision source vector elements to BFloat16 format.

Table C3-150 SIMD BFloat16 floating-point convert instructions

Mnemonic	Instruction	See
BFCVTN, BFCVTN2	Floating-point convert from single-precision to BFloat16 format (vector form)	BFCVTN, BFCVTN2

C3.8.30 The Cryptographic Extension in AArch64 state

C3.8.30.1 The Armv8.0 Cryptographic Extension

The instructions provided by the OPTIONAL Armv8.0 Cryptographic Extension use the SIMD and floating-point register file. For more information about the functions they provide, see:

- *Advanced Encryption Standard (AES)*.
- *Secure Hash Algorithm (SHA)*.
- *The Galois/Counter Mode of Operation*.

Table C3-151 shows the Armv8.0 Cryptographic Extension instructions.

Table C3-151 Cryptographic Extension instructions

Mnemonic	Instruction	See
AESD	AES single round decryption	AESD
AESE	AES single round encryption	AESE
AESIMC	AES inverse mix columns	AESIMC
AESMC	AES mix columns	AESMC
PMULL	Polynomial multiply long	PMULL, PMULL2 [a]
SHA1C	SHA1 hash update (choose)	SHA1C
SHA1H	SHA1 fixed rotate	SHA1H
SHA1M	SHA1 hash update (majority)	SHA1M
SHA1P	SHA1 hash update (parity)	SHA1P
SHA1SU0	SHA1 schedule update 0	SHA1SU0
SHA1SU1	SHA1 schedule update 1	SHA1SU1
SHA256H	SHA256 hash update, part 1	SHA256H
SHA256H2	SHA256 hash update, part 2	SHA256H2
SHA256SU0	SHA256 schedule update 0	SHA256SU0
SHA256SU1	SHA256 schedule update 1	SHA256SU1

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.8.30.2 Armv8.2 extensions to the Cryptographic Extension

Armv8.2 supports the following OPTIONAL extensions to the Cryptographic Extension:

- FEAT_SHA512, SHA2-512 functionality.
- FEAT_SHA3, SHA3 functionality.
- FEAT_SM3, SM3 functionality.
- FEAT_SM4, SM4 functionality.

C3.8.30.2.1 FEAT_SHA512, SHA2-512 functionality

FEAT_SHA512 provides instructions to accelerate the SHA-2 hash algorithm using a digest that is larger than 256 bits. The relevant standards are SHA-384, SHA-512, SHA-512|224 and SHA-512|256. These are all based on the SHA-512 calculation, and therefore this set of instructions is described as the *SHA512 instructions*.

Implementation of FEAT_SHA512 requires the implementation of the SHA1 and SHA2-256 instructions from the Armv8.0 Cryptographic Extension.

Note

Implementation of FEAT_SHA512 does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When FEAT_SHA512 is implemented, the value of ID_AA64ISAR0_EL1.SHA2 is `0b0010`, indicating support for the SHA512 instructions.

Table C3-152 shows the FEAT_SHA512 instructions:

Table C3-152 FEAT_SHA512 instructions

Mnemonic	Instruction	See
`SHA512H`	SHA512 Hash update part 1	SHA512H
`SHA512H2`	SHA512 Hash update part 2	SHA512H2
`SHA512SU0`	SHA512 Schedule Update 0	SHA512SU0
`SHA512SU1`	SHA512 Schedule Update 1	SHA512SU1

Use of the SHA512 instructions shows an example of the use of these instructions to calculate a SHA512 hash iteration. This example code is not part of the architectural definition of these instructions.

C3.8.30.2.2 FEAT_SHA3, SHA3 functionality

FEAT_SHA3 provides instructions to accelerate the SHA-3 hash algorithm. This set of instructions is described as the *SHA3 instructions.*

Note

Implementation of FEAT_SHA3 does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When FEAT_SHA3 is implemented, the value of ID_AA64ISAR0_EL1.SHA3 is `0b0001`, indicating support for the SHA3 instructions.

Table C3-153 shows the FEAT_SHA3 instructions. The SHA-3 hash algorithm is based on a running digest of 1600 bytes, arranged as a five by five array of 64-bit registers. The Arm acceleration of these instructions is based on mapping the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is performed on these registers as part of a round of the SHA-3 hash calculation.

The SIMD nature of the vector registers means the acceleration can calculate two parallel SHA3 hash calculations, where one calculation is performed using the zeroth 64-bit element of each vector, and the other calculation is performed using the first 64-bit element of each vector.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To provide acceleration where the SIMD calculation is not required, the instructions provide variants that operate only on the zeroth 64-bit elements. These are provided as a power optimization.

Table C3-153 FEAT_SHA3 instructions

Mnemonic	Instruction	See
BCAX	Bit Clear and Exclusive OR	BCAX
EOR3	Three-way Exclusive OR	EOR3
RAX1	Rotate and Exclusive OR	RAX1
XAR	Exclusive OR and Rotate	XAR

Use of the SHA3 instructions shows an example of the use of these instructions to calculate the combined theta, phi, rho and chi operations of a SHA3 iteration. This example code is not part of the architectural definition of these instructions.

C3.8.30.2.3 FEAT_SM3, SM3 functionality

FEAT_SM3 provides instructions to accelerate the SM3 hash algorithm, the standard Chinese hash algorithm. These are described as the *SM3 instructions*.

FEAT_SM3 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA512.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM3 functionality.

When FEAT_SM3 is implemented, the value of ID_AA64ISAR0_EL1.SM3 is `0b0001`, indicating support for the SM3 instructions.

Table C3-154 shows the FEAT_SM3 instructions. The SM3 algorithm calculates a digest of 256 bits, which can be held in two vector registers. The SM3 instructions include instructions to accelerate the computation of the hash and the schedule update.

Note

The SM3 instruction names refer to intermediate variables defined as part of the *SM3 Cryptographic Hash Algorithm* specification.

Table C3-154 FEAT_SM3 instructions

Mnemonic	Instruction	See
SM3SS1	SM3 SS1 calculation	SM3SS1
SM3TT1A	SM3 TT1 calculation, part A	SM3TT1A
SM3TT1B	SM3 TT1 calculation, part B	SM3TT1B
SM3TT2A	SM3 TT2 calculation, part A	SM3TT2A
SM3TT2B	SM3 TT2 calculation, part B	SM3TT2B
SM3PARTW1	SM3 PARTW calculation, part 1	SM3PARTW1
SM3PARTW2	SM3 PARTW calculation, part 1	SM3PARTW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Use of the SM3 instructions shows an example of the use of these instructions to generate an SM3 hash. This example code is not part of the architectural definition of these instructions.

C3.8.30.2.4 FEAT_SM4, SM4 functionality

FEAT_SM4 provides instruction to accelerate the SM4 encryption algorithm, the standard Chinese encryption algorithm. This set of instructions is described as the *SM4 instructions*.

FEAT_SM4 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA3.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM4 functionality.

When FEAT_SM4 is implemented, the value of ID_AA64ISAR0_EL1.SM4 is `0b0001`, indicating support for the SM4 instructions.

Table C3-155 shows the FEAT_SM4 instructions. The SM4 algorithm is 128-bit wide block cipher. The `SM4E` instruction accelerates a single round of encryption or decryption, and the `SM4EKEY` instruction accelerates a single round of key generation:

Table C3-155 FEAT_SM4 instructions

Mnemonic	Instruction	See
`SM4E`	SM4 Encrypt	SM4E
`SM4EKEY`	SM4 Key	SM4EKEY

Use of the SM4 instructions shows an example of the use of these instructions to perform SM4 encryption and decryption. This example code is not part of the architectural definition of these instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.9 Data processing - SVE

The following subsections describe the SVE processing instructions:

- SVE Vector integer operations
- SVE Vector address calculation
- SVE Bitwise logical operations
- SVE Bitwise shift, reverse, and count
- SVE Vector floating-point operations
- Predicate operations
- Loop control
- SVE Move operations
- Index vector generation
- Move prefix
- Reduction operations

C3.9.1 SVE Vector integer operations

C3.9.1.1 SVE Integer arithmetic

For binary operations, the Integer arithmetic instructions perform arithmetic operations on a source vector containing integer element values, and a second source vector of either integer element values or an immediate value.

For ternary operations, these instructions perform arithmetic operations on a source vector containing integer element values, a second source vector of either integer element values or an immediate value, and a third source vector containing integer element values.

Table C3-156 Integer arithmetic instructions

Mnemonic	Instruction	See
ABS	Absolute value	ABS
ADD	Add vectors (predicated)	ADD (vectors, predicated)
	Add vectors (unpredicated)	ADD (vectors, unpredicated)
	Add immediate	ADD (immediate)
CNOT	Logically invert Boolean condition	CNOT
MAD	Multiply-add, writing to the multiplicand register	MAD
MLA	Multiply-add, writing to the addend register	MLA (indexed) MLA (vectors)
MLS	Multiply-subtract, writing to the addend register	MLS (indexed) MLS (vectors)
MSB	Multiply-subtract, writing to the multiplicand register	MSB
MUL	Multiply by immediate	MUL (immediate)
	Multiply vectors	MUL (vectors, predicated) MUL (vectors, unpredicated)
NEG	Negate	NEG
SABD	Signed absolute difference	SABD
SDIV	Signed divide	SDIV

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SDIVR	Signed reversed divide	SDIVR
SMAX	Signed maximum with immediate	SMAX (immediate)
	Signed maximum vectors	SMAX (vectors)
SMIN	Signed minimum with immediate	SMIN (immediate)
	Signed minimum vectors	SMIN (vectors)
SMULH	Signed multiply returning high half	SMULH (predicated) SMULH (unpredicated)
SQADD	Signed saturating add immediate	SQADD (immediate)
	Signed saturating add vectors	SQADD (vectors, predicated) SQADD (vectors, unpredicated)
SQSUB	Signed saturating subtract immediate	SQSUB (immediate)
	Signed saturating subtract vectors	SQSUB (vectors, predicated) SQSUB (vectors, unpredicated)
SUB	Subtract immediate	SUB (immediate)
	Subtract vectors (predicated)	SUB (vectors, predicated)
	Subtract vectors (unpredicated)	SUB (vectors, unpredicated)
SUBR	Reversed subtract from immediate	SUBR (immediate)
	Reversed subtract vectors	SUBR (vectors)
SXTB	Signed byte extend	SXTB, SXTH, SXTW
SXTH	Signed halfword extend	SXTB, SXTH, SXTW
SXTW	Signed word extend	SXTB, SXTH, SXTW
UABD	Unsigned absolute difference	UABD
UDIV	Unsigned divide	UDIV
UDIVR	Unsigned reversed divide	UDIVR
UMAX	Unsigned maximum with immediate	UMAX (immediate)
	Unsigned maximum vectors	UMAX (vectors)
UMIN	Unsigned minimum with immediate	UMIN (immediate)
	Unsigned minimum vectors	UMIN (vectors)
UMULH	Unsigned multiply returning high half	UMULH (predicated) UMULH (unpredicated)
UQADD	Unsigned saturating add immediate	UQADD (immediate)
	Unsigned saturating add vectors	UQADD (vectors, predicated) UQADD (vectors, unpredicated)
UQSUB	Unsigned saturating subtract immediate	UQSUB (immediate)
	Unsigned saturating subtract vectors	UQSUB (vectors, predicated) UQSUB (vectors, unpredicated)
UXTB	Unsigned byte extend	UXTB, UXTH, UXTW
UXTH	Unsigned halfword extend	UXTB, UXTH, UXTW
UXTW	Unsigned word extend	UXTB, UXTH, UXTW

C3.9.1.2 SVE Integer dot product

The Integer dot product instructions delimit the source vectors into groups of four 8-bit or 16-bit integer elements. Within each group of four elements, the elements in the first source vector are multiplied by the corresponding elements

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

in the second source vector. The resulting widened products are summed and added to the 32-bit or 64-bit element of the accumulator and destination vector that aligns with the group of four elements in the first source vector.

The indexed forms of these instructions specify a single, numbered, group of four elements within each 128-bit segment of the second source vector as the multiplier for all the groups of four elements within the corresponding 128-bit segment of the first source vector.

Table C3-157 Integer dot product instructions

Mnemonic	Instruction	See
SDOT	Signed integer indexed dot product	SDOT (2-way, indexed)
	Signed integer dot product	SDOT (2-way, vectors)
SDOT	Signed dot product by vector	SDOT (4-way, vectors)
	Signed dot product by indexed elements	SDOT (4-way, indexed)
SUDOT	Signed by unsigned integer dot product by indexed elements	SUDOT
UDOT	Unsigned integer indexed dot product	UDOT (2-way, indexed)
	Unsigned integer dot product	UDOT (2-way, vectors)
UDOT	Unsigned dot product by vector	UDOT (4-way, vectors)
	Unsigned dot product by indexed elements	UDOT (4-way, indexed)
USDOT	Unsigned by signed integer dot product	USDOT (vectors)
	Unsigned by signed integer dot product by indexed elements	USDOT (indexed)

C3.9.1.3 SVE Integer matrix multiply-accumulate

The Integer matrix multiply-accumulate instructions are implemented by FEAT_I8MM.

The matrix multiply-accumulate instructions delimit source and destination vectors into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit integers in row-by-row order.
- In the second source vector, an 8x2 matrix of 8-bit integers in column-by-column order.
- In the destination vector, a 2x2 matrix of 32-bit integers in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

Table C3-158 Integer matrix multiply operations

Mnemonic	Instruction	See
SMMLA	Widening signed 8-bit integer matrix multiply-accumulate into 2x2 matrix	SMMLA
UMMLA	Widening unsigned 8-bit integer matrix multiply-accumulate into 2x2 matrix	UMMLA
USMMLA	Widening mixed sign 8-bit integer matrix multiply-accumulate into 2x2 matrix	USMMLA

C3.9.1.4 SVE Integer comparison

The Integer comparison instructions compare *Active elements* in the first source vector with the corresponding elements in a second vector or with an immediate value. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. *Inactive elements* in the destination predicate register are set to FALSE. All integer comparisons set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The wide element variants of the compare instructions allow a packed vector of narrower elements to be compared with wider 64-bit elements. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and compare each narrow element of the first source vector with the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first source vector are compared with 64-bit element[0] of the second source vector, 8-bit element[8] to element[15] with 64-bit element[1], and so on. All 64 bits of the wide elements are significant for the comparison, with the narrow elements being sign-extended or zero-extended to 64 bits as appropriate for the type of comparison.

Table C3-159 Integer comparison instructions

Mnemonic	Instruction	See
CMPEQ	Compare signed equal to immediate	CMP<cc> (immediate) Equal
	Compare signed equal to wide elements	CMP<cc> (wide elements) Equal
	Compare signed equal to vector	CMP<cc> (vectors) Equal
CMPGE	Compare signed greater than or equal to immediate	CMP<cc> (immediate) Greater than or equal
	Compare signed greater than or equal to wide elements	CMP<cc> (wide elements) Greater than or equal
	Compare signed greater than or equal to vector	CMP<cc> (vectors) Greater than or equal
CMPGT	Compare signed greater than immediate	CMP<cc> (immediate) Greater than
	Compare signed greater than wide elements	CMP<cc> (wide elements) Greater than
	Compare signed greater than vector	CMP<cc> (vectors) Greater than
CMPHI	Compare unsigned higher than immediate	CMP<cc> (immediate) Higher
	Compare unsigned higher than wide elements	CMP<cc> (wide elements) Higher
	Compare unsigned higher than vector	CMP<cc> (vectors) Higher
CMPHS	Compare unsigned higher than or same as immediate	CMP<cc> (immediate) Higher or same
	Compare unsigned higher than or same as wide elements	CMP<cc> (wide elements) Higher or same
	Compare unsigned higher than or same as vector	CMP<cc> (vectors) Higher or same
CMPLE	Compare signed less than or equal to immediate	CMP<cc> (immediate) Less than or equal
	Compare signed less than or equal to wide elements	CMP<cc> (wide elements) Less than or equal
	Compare signed less than or equal to vector	CMPLE (vectors)
CMPLO	Compare unsigned lower than immediate	CMP<cc> (immediate) Lower
	Compare unsigned lower than 64-bit wide elements	CMP<cc> (wide elements) Lower
	Compare unsigned lower than vector	CMPLO (vectors)
CMPLS	Compare unsigned lower or same as immediate	CMP<cc> (immediate) Lower or same
	Compare unsigned lower or same as wide elements	CMP<cc> (wide elements) Lower or same
	Compare unsigned lower or same as vector	CMPLS (vectors)
CMPLT	Compare signed less than immediate	CMP<cc> (immediate) Less than
	Compare signed less than wide elements	CMP<cc> (wide elements) Less than
	Compare signed less than vector	CMPLT (vectors)
CMPNE	Compare not equal to immediate	CMP<cc> (immediate) Not equal
	Compare not equal to wide elements	CMP<cc> (wide elements) Not equal
	Compare not equal to vector	CMP<cc> (vectors) Not equal

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.9.2 SVE Vector address calculation

The Vector address calculation instructions calculate vectors of addresses and addresses of vectors. This includes instructions to add a multiple of the current vector length or predicate register length, in bytes, to a general-purpose register.

The `ADR` instruction is an integer arithmetic operation that is used to calculate a vector of 64-bit or 32-bit addresses.

The `ADR` destination vector elements are calculated by the addition of the corresponding elements in the source vectors, with an optional sign or zero extension and optional bitwise left shift of 1-3 bits applied to the final operands. This can be considered as the addition of a vector base and a scaled vector index.

The `ADR` instruction calculates a vector of 32-bit addresses by the addition of a 32-bit base and a scaled 32-bit unsigned index. The ADR instruction calculates a vector of 64-bit addresses by one of:

- Addition of a 64-bit base and a scaled 64-bit unsigned index.
- Addition of a 64-bit base and a scaled, zero-extended 32-bit index.
- Addition of a 64-bit base and a scaled, sign-extended 32-bit index.

Table C3-160 Vector address calculation instructions

Mnemonic	Instruction	See
`ADDPL`	Add multiple of predicate register length, in bytes, to scalar register	ADDPL
`ADDVL`	Add multiple of vector length, in bytes, to scalar register	ADDVL
`ADR`	Calculate vector of addresses	ADR
`RDVL`	Read multiple of vector register length, in bytes, to scalar register	RDVL

C3.9.3 SVE Bitwise logical operations

The Bitwise logical operations instructions perform bitwise logical operations on vectors. Where operations are unpredicated, the operations are independent of the element size.

Table C3-161 Bitwise logical operations

Mnemonic	Instruction	See
`AND`	Bitwise AND vectors (predicated)	AND (vectors, predicated)
	Bitwise AND vectors (unpredicated)	AND (vectors, unpredicated)
	Bitwise AND with immediate	AND (immediate)
`BIC`	Bitwise clear with vector (predicated)	BIC (vectors, predicated)
	Bitwise clear with vector (unpredicated)	BIC (vectors, unpredicated)
	Bitwise clear using immediate	BIC (immediate)
`DUPM`	Broadcast bitmask immediate to vector (unpredicated)	DUPM
`EON`	Bitwise exclusive-OR with inverted immediate	EON
`EOR`	Bitwise exclusive-OR vectors (predicated)	EOR (vectors, predicated)
	Bitwise exclusive-OR vectors (unpredicated)	EOR (vectors, unpredicated)
	Bitwise exclusive-OR with immediate	EOR (immediate)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
MOV	Move bitmask immediate to vector	MOV
	Move vector register	MOV (vector, unpredicated)
NOT	Bitwise invert vector	NOT (vector)
ORN	Bitwise OR with inverted immediate	ORN (immediate)
ORR	Bitwise OR vectors (predicated)	ORR (vectors, predicated)
	Bitwise OR vectors (unpredicated)	ORR (vectors, unpredicated)
	Bitwise OR with immediate	ORR (immediate)

C3.9.4 SVE Bitwise shift, reverse, and count

Bitwise shifts, reversals, and counts within vector elements.

Shift counts saturate at the number of bits per element, rather than being used modulo the element size. If modulo behavior is required, then the modulus must be calculated separately.

The wide element variants of the bitwise shift instructions allow a packed vector of narrower elements to be shifted by wider 64-bit shift amounts. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and shift each narrow element of the first source vector by the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first vector are shifted by 64-bit element[0] of the second source vector, 8-bit element [8] to element[15] by 64-bit element[1], and so on. All 64 bits of the wide shift amount are significant.

Table C3-162 Bitwise shift, reverse, and count instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right by immediate (predicated)	ASR (immediate, predicated)
	Arithmetic shift right by immediate (unpredicated)	ASR (immediate, unpredicated)
	Arithmetic shift right by wide elements (predicated)	ASR (wide elements, predicated)
	Arithmetic shift right by wide elements (unpredicated)	ASR (wide elements, unpredicated)
	Arithmetic shift right by vector	ASR (vectors)
ASRD	Arithmetic shift right for divide by immediate	ASRD
ASRR	Reversed arithmetic shift right by vector	ASRR
CLS	Count leading sign bits	CLS
CLZ	Count leading zero bits	CLZ
CNT	Count nonzero bits	CNT
LSL	Logical shift left by immediate (predicated)	LSL (immediate, predicated)
	Logical shift left by immediate (unpredicated)	LSL (immediate, unpredicated)
	Logical shift left by wide elements (predicated)	LSL (wide elements, predicated)
	Logical shift left by wide elements (unpredicated)	LSL (wide elements, unpredicated)
	Logical shift left by vector	LSL (vectors)
LSLR	Reversed logical shift left by vector	LSLR
LSR	Logical shift right by immediate (predicated)	LSR (immediate, predicated)
	Logical shift right by immediate (unpredicated)	LSR (immediate, unpredicated)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Logical shift right by wide elements (predicated)	LSR (wide elements, predicated)
	Logical shift right by wide elements (unpredicated)	LSR (wide elements, unpredicated)
	Logical shift right by vector	LSR (vectors)
LSRR	Reversed logical shift right by vector	LSRR
RBIT	Reverse bits	RBIT

C3.9.5 SVE Vector floating-point operations

The following instructions operate on floating-point data within a vector.

C3.9.5.1 SVE Floating-point arithmetic

The Floating-point arithmetic instructions perform arithmetic operations on vectors containing floating-point element values.

Table C3-163 Floating-point arithmetic instructions

Mnemonic	Instruction	See
FABD	Floating-point absolute difference	FABD
FABS	Floating-point absolute value	FABS
FADD	Floating-point add (immediate)	FADD (immediate)
	Floating-point add (predicated)	FADD (vectors, predicated)
	Floating-point add (unpredicated)	FADD (vectors, unpredicated)
FDIV	Floating-point divide	FDIV
FDIVR	Floating-point reversed divide	FDIVR
FMAX	Floating-point maximum with immediate	FMAX (immediate)
	Floating-point maximum vectors	FMAX (vectors)
FMAXNM	Floating-point maximum number with immediate	FMAXNM (immediate)
	Floating-point maximum number vectors	FMAXNM (vectors)
FMIN	Floating-point minimum with immediate	FMIN (immediate)
	Floating-point minimum vectors	FMIN (vectors)
FMINNM	Floating-point minimum number with immediate	FMINNM (immediate)
	Floating-point minimum number vectors	FMINNM (vectors)
FMUL	Floating-point multiply by immediate	FMUL (immediate)
	Floating-point multiply vectors (predicated)	FMUL (vectors, predicated)
	Floating-point multiply vectors (unpredicated)	FMUL (vectors, unpredicated)
FMULX	Floating-point multiply-extended	FMULX
FNEG	Floating-point negate	FNEG
FRECPE	Floating-point reciprocal estimate	FRECPE
FRECPS	Floating-point reciprocal step	FRECPS
FRECPX	Floating-point reciprocal exponent	FRECPX

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FRSQRTE	Floating-point reciprocal square root estimate	FRSQRTE
FRSQRTS	Floating-point reciprocal square root step	FRSQRTS
FSCALE	Floating-point adjust exponent by vector	FSCALE
FSQRT	Floating-point square root	FSQRT
FSUB	Floating-point subtract immediate	FSUB (immediate)
	Floating-point subtract vectors (predicated)	FSUB (vectors, predicated)
	Floating-point subtract vectors (unpredicated)	FSUB (vectors, unpredicated)
FSUBR	Floating-point reversed subtract from immediate	FSUBR (immediate)
	Floating-point reversed subtract vectors	FSUBR (vectors)

C3.9.5.2 SVE Floating-point multiply accumulate

The Floating-point multiply accumulate instructions perform floating-point fused multiply-add or multiply-subtract operations and their negated forms. There are two groups of these instructions, as follows:

- Instructions where the result of the operation is written to the addend register.
  - Supported instructions are: FMLA, FMLS, FNMLA, FNMLS.
- Instructions where the result of the operation is written to the multiplicand register.
  - Supported instructions are: FMAD, FMSB, FNMAD, FNMSB.

Table C3-164 Floating-point multiply accumulate instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add vectors, writing to the addend	FMLA (indexed) FMLA (vectors)
FMLS	Floating-point fused multiply-subtract vectors, writing to the addend	FMLS (indexed) FMLS (vectors)
FNMLA	Floating-point negated fused multiply-add vectors, writing to the addend	FNMLA
FNMLS	Floating-point negated fused multiply-subtract vectors, writing to the addend	FNMLS
FMAD	Floating-point fused multiply-add vectors, writing to the multiplicand	FMAD
FMSB	Floating-point fused multiply-subtract vectors, writing to the multiplicand	FMLSLB (indexed) FMLSLB (vectors)
FNMAD	Floating-point negated fused multiply-add vectors, writing to the multiplicand	FNMAD
FNMSB	Floating-point negated fused multiply-subtract vectors, writing to the multiplicand	FNMSB

C3.9.5.3 SVE Floating-point complex arithmetic

The Floating-point complex arithmetic instructions perform arithmetic on vectors containing floating-point complex numbers as interleaved pairs of elements, where the even-numbered elements contain the real components and the odd-numbered elements contain the imaginary components.

The FCADD instructions rotate the complex numbers in the second source vector by 90 degrees or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, before adding active pairs of elements to the corresponding elements of the first source vector in a destructive manner.

The FCMLA instructions perform a transformation of the operands to allow the creation of multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The transformations performed are as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The complex numbers in the second source vector, considered in polar form, are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector, considered in polar form, are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are then added to the corresponding components of the destination and addend vector, without intermediate rounding.

Two FCMLA instructions can be used as follows:

```
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #A
...
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #B
```

For example, some meaningful combinations of A and B are:

- A=0, B=90. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are added to the complex numbers in Zda.
- A=0, B=270. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are added to the complex numbers in Zda.
- A=180, B=270. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are subtracted from the complex numbers in Zda.
- A=180, B=90. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are subtracted from the complex numbers in Zda.

Note

The lack of intermediate rounding can give unexpected results in certain cases relative to a traditional sequence of independent multiply, add, and subtract instructions.
In addition, when using these instructions, the behavior of calculations such as (∞+∞i) multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex. The expectation is that these instructions are only used in situations where the effect of differences in the rounding and handling of infinities are not material to the calculation.

Table C3-165 Floating-point complex arithmetic instructions

Mnemonic	Instruction	See
FCADD	Floating-point complex add with rotate	FCADD
FCMLA	Floating-point complex multiply-add with rotate	FCMLA (vectors)

C3.9.5.4 SVE Floating-point rounding and conversion

The Floating-point rounding and conversion instructions change floating-point size and precision, round floating-point to integral floating-point with explicit rounding mode, and convert floating-point to or from integer format.

Table C3-166 Floating-point rounding and conversion instructions

Mnemonic	Instruction	See
BFCVT	Floating-point down convert to BFloat16 format	BFCVT
BFCVTNT	Floating-point down convert and narrow to BFloat16 format (top, predicated)	BFCVTNT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FCVT	Floating-point convert precision	FCVT
FCVTZS	Floating-point convert to signed integer, rounding toward zero	FCVTZS
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero	FCVTZU
FRINTA	Floating-point round to integral value, to nearest with ties away from zero	FRINT<r> Nearest with ties to away
FRINTI	Floating-point round to integral value, using the current rounding mode	FRINT<r> Current mode
FRINTM	Floating-point round to integral value, toward minus infinity	FRINT<r> Toward minus infinity
FRINTN	Floating-point round to integral value, to nearest with ties to even	FRINT<r> Nearest with ties to even
FRINTP	Floating-point round to integral value, toward plus infinity	FRINT<r> Toward plus infinity
FRINTX	Floating-point round to integral value exact, using the current rounding mode	FRINT<r> Current mode signalling inexact
FRINTZ	Floating-point round to integral value, toward zero	FRINT<r> Toward zero
SCVTF	Signed integer convert to floating-point	SCVTF
UCVTF	Unsigned integer convert to floating-point	UCVTF

C3.9.5.5 SVE Floating-point comparisons

The Floating-point comparison instructions compare active floating-point element values in the first source vector with corresponding elements in the second vector or with the immediate value +0.0. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. *Inactive elements* in the destination predicate register are set to FALSE. Floating-point vector comparisons do not set the condition flags.

Table C3-167 Floating-point comparison instructions

Mnemonic	Instruction	See
FACGE	Floating-point absolute compare greater than or equal	FAC<cc> - Greater than or equal
FACGT	Floating-point absolute compare greater than	FAC<cc> - Greater than
FACLE	Floating-point absolute compare less than or equal	FACLE
FACLT	Floating-point absolute compare less than	FACLT
FCMEQ	Floating-point compare equal to zero	FCM<cc> (zero) - Equal
	Floating-point compare equal to vector	FCM<cc> (vectors) - Equal
FCMGE	Floating-point compare greater than or equal to zero	FCM<cc> (zero) - Greater than or equal
	Floating-point compare greater than or equal to vector	FCM<cc> (vectors) - Greater than or equal
FCMGT	Floating-point compare greater than zero	FCM<cc> (zero) - Greater than
	Floating-point compare greater than vector	FCM<cc> (vectors) - Greater than
FCMLE	Floating-point compare less than or equal to zero	FCM<cc> (zero) - Less than or equal
	Floating-point compare less than or equal to vector	FCMLE (vectors)
FCMLT	Floating-point compare less than zero	FCM<cc> (zero) - Less than
	Floating-point compare less than vector	FCMLT (vectors)
FCMNE	Floating-point compare not equal to zero	FCM<cc> (zero) - Not equal
	Floating-point compare not equal to vector	FCM<cc> (vectors) - Not equal
FCMUO	Floating-point unordered vectors	FCM<cc> (vectors) - Unordered

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See

C3.9.5.6 SVE Floating-point transcendental acceleration

The Floating-point transcendental instructions accelerate calculations of sine, cosine, and exponential functions for vectors containing floating-point element values.

The trigonometric instructions accelerate the calculation of a polynomial series approximation for the sine and cosine functions. The exponential instruction accelerates the polynomial series calculation of the exponential function.

Table C3-168 Floating-point transcendental instructions

Mnemonic	Instruction	See
`FTMAD`	Floating-point trigonometric multiply-add coefficient	FTMAD
`FTSMUL`	Floating-point trigonometric starting value	FTSMUL
`FTSSEL`	Floating-point trigonometric select coefficient	FTSSEL
`FEXPA`	Floating-point exponential accelerator	FEXPA

C3.9.5.7 SVE Floating-point indexed multiples

The Floating-point indexed multiples instructions multiply all floating-point elements within each 128-bit segment of the first source vector by the single numbered element within the corresponding segment of the second source vector.

For the `FMLA` and `FMLS` instructions, the products are destructively added or subtracted from the corresponding elements of the addend and destination vector, without intermediate rounding.

Table C3-169 Floating-point indexed multiples instructions

Mnemonic	Instruction	See
`FMLA`	Floating-point fused multiply-add by indexed elements	FMLA (indexed)
`FMLS`	Floating-point fused multiply-subtract by indexed elements	FMLS (indexed)
`FMUL`	Floating-point multiply by indexed elements	FMUL (indexed)

C3.9.5.8 SVE Floating-point matrix multiply-accumulate and companion instructions

The floating-point matrix multiply-accumulate and companion instructions facilitate matrix multiplication calculation.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. The segments are organized as follows:

- In the first source vector, a 2x2 matrix in row-by-row order.
- In the second source vector, a 2x2 matrix in column-by-column order.
- In the destination vector, a 2x2 matrix in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment. For the double-precision matrix multiply-accumulate instructions, the vector segment length and minimum vector length is 256 bits. Double-precision matrix multiply-accumulate instructions are not supported when the vector length is 128 bits. For the single-precision matrix multiply-accumulate instruction, the vector segment length is 128 bits.

The floating-point matrix multiply-accumulate instructions strictly define the order of accumulations, and the multiplications and additions are not fused, so intermediate rounding is performed after every multiplication and every addition.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The companion instructions support data rearrangements in vector registers as required by the double-precision matrix multiply-accumulate instructions.

The following table shows the floating-point matrix multiplication instructions and companion instructions that are implemented by FEAT_F64MM:

Table C3-170 Floating-point matrix multiply instructions

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (double-precision)	FMMLA
LD1ROB	Contiguous load and replicate thirty-two bytes, scalar plus scalar	LD1ROB (scalar plus scalar)
	Contiguous load and replicate thirty-two bytes, scalar plus immediate	LD1ROB (scalar plus immediate)
LD1ROD	Contiguous load and replicate four doublewords, scalar plus scalar	LD1ROD (scalar plus scalar)
	Contiguous load and replicate four doublewords, scalar plus immediate	LD1ROD (scalar plus immediate)
LD1ROH	Contiguous load and replicate sixteen halfwords, scalar plus scalar	LD1ROH (scalar plus scalar)
	Contiguous load and replicate sixteen halfwords, scalar plus immediate	LD1ROH (scalar plus immediate)
LD1ROW	Contiguous load and replicate eight words, scalar plus scalar	LD1ROW (scalar plus scalar)
	Contiguous load and replicate eight words, scalar plus immediate	LD1ROW (scalar plus immediate)
TRN1, TRN2	Interleave even or odd 128-bit elements from two vectors	TRN1, TRN2 (vectors)
UZP1, UZP2	Concatenate even or odd 128-bit elements from two vectors	UZP1, UZP2 (vectors)
ZIP1, ZIP2	Interleave 128-bit elements from two half vectors	ZIP1, ZIP2 (vectors)

The following table shows the floating-point matrix multiplication instruction, which is implemented by FEAT_F32MM:

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (single-precision)	FMMLA

C3.9.5.9 SVE BFloat16 floating-point multiply-add

The BFloat16 floating-point multiply-add instructions perform an implicit conversion of the bottom (even-numbered) or top (odd-numbered) BFloat16 source elements to IEEE 754 single-precision floating-point format before performing a fused multiply-add without intermediate rounding to the overlapping single-precision destination element. These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Table C3-172 SVE BFloat16 floating-point multiply-add instructions

Mnemonic	Instruction	See
BFMLALB	BFloat16 floating-point widening multiply accumulate long bottom (vector and indexed forms)	BFMLALB (vectors) BFMLALB (indexed)
BFMLALT	BFloat16 floating-point widening multiply accumulate long top (vector and indexed forms)	BFMLALT (vectors) BFMLALT (indexed)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BFMLSLB	BFloat16 floating-point multiply-subtract long from single-precision (bottom, indexed)	BFMLSLB (indexed)
	BFloat16 floating-point multiply-subtract long from single-precision (bottom)	BFMLSLB (vectors)
BFMLSLT	BFloat16 floating-point multiply-subtract long from single-precision (top, indexed)	BFMLSLT (indexed)
	BFloat16 floating-point multiply-subtract long from single-precision (top)	BFMLSLT (vectors)

C3.9.5.10 SVE BFloat16 floating-point dot product

The BFloat16 floating-point dot product instruction, BFDOT, performs an implicit conversion of vectors of BFloat16 input values to IEEE 754 single-precision floating-point format.

The BFloat16 dot product instructions delimit their source vectors into pairs of BFloat16 elements.

Table C3-173 SVE BFloat16 floating-point dot product instructions

Mnemonic	Instruction	See
BFDOT	BFloat16 floating-point dot product (vector and indexed forms)	BFDOT (vectors) BFDOT (indexed)

C3.9.5.11 SVE BFloat16 floating-point matrix multiply-accumulate

The BFloat16 floating-point matrix multiply-accumulate instruction, BFMMLA, is implemented by FEAT_BF16.

This instruction delimits the source and destination vectors into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of BFloat16 values in row-by-row order.
- In the second source vector, a 4x2 matrix of BFloat16 values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per source vector segment and accumulated into the corresponding destination vector segment. This corresponds to accumulating two 2-way BFloat16 widening dot products into each single-precision destination element, following the numeric behaviors described for BFDOT instruction in SVE BFloat16 floating-point dot product.

Table C3-174 SVE BFloat16 floating-point matrix multiply instruction

Mnemonic	Instruction	See
BFMMLA	BFloat16 floating-point matrix multiply-accumulate	BFMMLA

C3.9.5.12 SVE BFloat16 floating-point convert

The BFloat16 floating-point convert instructions perform accurately rounded down-conversion of IEEE 754 single-precision source vector elements to BFloat16 format.

The BFCVT instruction places its BFloat16 results in the bottom or even-numbered 16-bit elements of the destination vector, and sets the top or odd-numbered elements to zero.

The BFCVTNT instruction places its BFloat16 results in the top or odd-numbered 16-bit elements of the destination vector, leaving the bottom or even-numbered elements unchanged.

These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-175 SVE BFloat16 floating-point convert instructions

Mnemonic	Instruction	See
BFCVT	Floating-point down convert to BFloat16 format (predicated)	BFCVT
BFCVTNT	Floating-point down convert and narrow to BFloat16 (top, predicated)	BFCVTNT

C3.9.6 Predicate operations

The Predicate instructions perform operations that manipulate the predicate registers.

Some of these instructions are insensitive to the predicate element size and specify an explicit byte element size qualifier, .B, but an assembler must accept any qualifier, or none.

C3.9.6.1 SVE Predicate initialization

The Predicate initialization instructions initialize predicate elements.Predicate elements can be initialized to be FALSE, or to be TRUE when their element number is less than:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, which is implicitly a multiple of two.

Unspecified or out of range constraint encodings generate a predicate with values that are all FALSE and do not cause an Undefined Instruction exception.

Table C3-176 Predicate initialization instructions

Mnemonic	Instruction	See
PFALSE	Set all predicate elements to FALSE	PFALSE
PTRUE	Initialize predicate elements from named constraint	PTRUE (predicate)
PTRUES	Initialize predicate elements from named constraint, setting the condition flags	PTRUES

C3.9.6.2 SVE Predicate move operations

The Predicate move instructions operate on all bits of the predicate registers, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Table C3-177 Predicate move instructions

Mnemonic	Instruction	See
SEL	Select predicate elements from two predicates	SEL (predicates)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
MOV	Move predicate elements (predicated, merging)	MOV (predicate, predicated, merging)
	Move predicate elements (predicated, zeroing)	MOV (predicate, predicated, zeroing)
	Move predicate elements (unpredicated)	MOV
MOVS	Move predicate elements, setting the condition flags (predicated)	MOVS (predicated)
	Move predicate elements, setting the condition flags (unpredicated)	MOVS (unpredicated)

C3.9.6.3 SVE Predicate logical operations

The Predicate logical operation instructions perform bitwise logical operations on predicate registers that operate on all bits of the register, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Inactive elements in the destination Predicate register are set to zero, except for PTEST which does not specify a destination register. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Table C3-178 Predicate logical operation instructions

Mnemonic	Instruction	See
AND	Bitwise AND predicates	AND (predicates)
ANDS	Bitwise AND predicates, setting the condition flags	ANDS
BIC	Bitwise clear predicates	BIC (predicates)
BICS	Bitwise clear predicates, setting the condition flags	BICS
EOR	Bitwise exclusive-OR predicates	EOR (predicates)
EORS	Bitwise exclusive-OR predicates, setting the condition flags	EORS
NAND	Bitwise NAND predicates	NAND
NANDS	Bitwise NAND predicates, setting the condition flags	NANDS
NOR	Bitwise NOR predicates	NOR
NORS	Bitwise NOR predicates, setting the condition flags	NORS
NOT	Bitwise invert predicate	NOT (predicate)
NOTS	Bitwise invert predicate, setting the condition flags	NOTS
ORN	Bitwise OR inverted predicate	ORN (predicates)
ORNS	Bitwise OR inverted predicate, setting the condition flags	ORNS
ORR	Bitwise OR predicates	ORR (predicates)
ORRS	Bitwise OR predicates, setting the condition flags	ORRS
PTEST	Test predicate value, setting the condition flags	PTEST

C3.9.6.4 FFR predicate handling

The FFR predicate handling instructions work with SVE First-fault and Non-fault loads using the FFR to determine which elements have been successfully loaded and which remain to be loaded on a subsequent iteration.

The RDFFRS instruction sets the N, Z, and C condition flags based on the predicate result, and sets the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the RDFFRS instruction should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-179 FFR predicate handling instructions

Mnemonic	Instruction	See
RDFFR	Return predicate of successfully loaded elements (unpredicated)	RDFFR (unpredicated)
	Return predicate of successfully loaded elements (predicated)	RDFFR (predicated)
RDFFRS	Return predicate of successfully loaded elements, setting the condition flags (predicated)	RDFFRS
SETFFR	Initialize the First-fault register to all TRUE	SETFFR
WRFFR	Write a predicate register to the First-fault register	WRFFR

C3.9.6.5 Predicate count

The Predicate count instructions count either the number of Active predicate elements that are set to TRUE, or the number of elements implied by a named predicate constraint. The count can be placed in a general-purpose register, or used to increment or decrement a vector or general-purpose register.

Signed or unsigned saturating variants handle cases where, for example, an increment might cause a vectorized scalar loop index to overflow and therefore never satisfy a loop termination condition that compares it with a limit that is close to the maximum integer value.

The named predicate constraint limits the number of elements to:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, implicitly a multiple of two.

Unspecified or out of range predicate constraint encodings generate a zero element count and do not cause an Undefined Instruction exception.

Table C3-180 Predicate count instructions

Mnemonic	Instruction	See
CNTB	Set scalar to multiple of 8-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTH	Set scalar to multiple of 16-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTW	Set scalar to multiple of 32-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTD	Set scalar to multiple of 64-bit predicate constraint element count	CNTB, CNTD, CNTH, CNTW
CNTP	Set scalar to the number of Active predicate elements that are TRUE	CNTP (predicate)
DECB	Decrement scalar by multiple of 8-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
DECH	Decrement scalar by multiple of 16-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
	Decrement vector by multiple of 16-bit predicate constraint element count	DECD, DECH, DECW (vector)
DECW	Decrement scalar by multiple of 32-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
	Decrement vector by multiple of 32-bit predicate constraint element count	DECD, DECH, DECW (vector)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
DECD	Decrement scalar by multiple of 64-bit predicate constraint element count	DECB, DECD, DECH, DECW (scalar)
	Decrement vector by multiple of 64-bit predicate constraint element count	DECD, DECH, DECW (vector)
DECP	Decrement scalar by the number of predicate elements that are TRUE	DECP (scalar)
	Decrement vector by the number of Active predicate elements that are TRUE	DECP (vector)
INCB	Increment scalar by multiple of 8-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
INCH	Increment scalar by multiple of 16-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
	Increment vector by multiple of 16-bit predicate constraint element count	INCD, INCH, INCW (vector)
INCW	Increment scalar by multiple of 32-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
	Increment vector by multiple of 32-bit predicate constraint element count	INCD, INCH, INCW (vector)
INCD	Increment scalar by multiple of 64-bit predicate constraint element count	INCB, INCD, INCH, INCW (scalar)
	Increment vector by multiple of 64-bit predicate constraint element count	INCD, INCH, INCW (vector)
INCP	Increment scalar by the number of predicate elements that are TRUE	INCP (scalar)
	Increment vector by the number of predicate elements that are TRUE	INCP (vector)
SQDECB	Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count	SQDECB
SQDECH	Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count	SQDECH (scalar)
	Signed saturating decrement vector by multiple of 16-bit predicate constraint element count	SQDECH (vector)
SQDECW	Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count	SQDECW (scalar)
	Signed saturating decrement vector by multiple of 32-bit predicate constraint element count	SQDECW (vector)
SQDECD	Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count	SQDECD (scalar)
	Signed saturating decrement vector by multiple of 64-bit predicate constraint element count	SQDECD (vector)
SQDECP	Signed saturating decrement scalar the number of predicate elements that are TRUE	SQDECP (scalar)
	Signed saturating decrement vector by the number of predicate elements that are TRUE	SQDECP (vector)
SQINCB	Signed saturating increment scalar by multiple of 8-bit predicate constraint element count	SQINCB
SQINCH	Signed saturating increment scalar by multiple of 16-bit predicate constraint element count	SQINCH (scalar)
	Signed saturating increment vector by multiple of 16-bit predicate constraint element count	SQINCH (vector)
SQINCW	Signed saturating increment scalar by multiple of 32-bit predicate constraint element count	SQINCW (scalar)
	Signed saturating increment vector by multiple of 32-bit predicate constraint element count	SQINCW (vector)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SQINCD	Signed saturating increment scalar by multiple of 64-bit predicate constraint element count	SQINCD (scalar)
	Signed saturating increment vector by multiple of 64-bit predicate constraint element count	SQINCD (vector)
SQINCP	Signed saturating increment scalar by the number of predicate elements that are TRUE	SQINCP (scalar)
	Signed saturating increment vector by the number of predicate elements that are TRUE	SQINCP (vector)
UQDECB	Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count	UQDECB
UQDECH	Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count	UQDECH (scalar)
	Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count	UQDECH (vector)
UQDECW	Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count	UQDECW (scalar)
	Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count	UQDECW (vector)
UQDECD	Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count	UQDECD (scalar)
	Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count	UQDECD (vector)
UQDECP	Unsigned saturating decrement scalar by the number of predicate elements that are TRUE	UQDECP (scalar)
	Unsigned saturating decrement vector by the number of predicate elements that are TRUE	UQDECP (vector)
UQINCB	Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count	UQINCB
UQINCH	Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count	UQINCH (scalar)
	Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count	UQINCH (vector)
UQINCW	Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count	UQINCW (scalar)
	Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count	UQINCW (vector)
UQINCD	Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count	UQINCD (scalar)
	Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count	UQINCD (vector)
UQINCP	Unsigned saturating increment scalar by the number of predicate elements that are TRUE	UQINCP (scalar)
	Unsigned saturating increment vector by the number of predicate elements that are TRUE	UQINCP (vector)

C3.9.7 Loop control

These instructions control counted vector loops and vector loops with data-dependent termination conditions. These instructions create a loop partition predicate with *Active elements* set to TRUE up to the point where the loop should

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

terminate, and FALSE thereafter. Two loop concepts are supported, simple loops and data-dependent loops.

C3.9.7.1 Simple loops

An up-counting `WHILE` instruction that increments the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the lowest numbered element are true while the comparison is true.
- The predicate elements thereafter, up to the highest numbered element, are false when the comparison becomes false.

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is incremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the maximum integer value of the selected size and type of comparison.

The N, Z, C, and V condition flags are unconditionally set to control a subsequent conditional branch.

Table C3-181 Single loop instructions

Mnemonic	Instruction	See
`WHILELE`	While incrementing signed scalar less than or equal to scalar	WHILELE (predicate)
`WHILELO`	While incrementing unsigned scalar lower than scalar	WHILELO (predicate)
`WHILELS`	While incrementing unsigned scalar lower than or the same as scalar	WHILELS (predicate)
`WHILELT`	While incrementing signed scalar less than scalar	WHILELT (predicate)

C3.9.7.2 Data-independent loops

For data-dependent termination conditions, it is necessary to convert the result of a vector comparison into a loop partition predicate. The new partition truncates the current vector partition immediately before or after the first active TRUE comparison. The N, Z, C, and V condition flags are optionally set to control a subsequent conditional branch.

The `BRKA` instructions set active destination predicate elements to TRUE up to and including the first active TRUE element in their source predicate register, setting subsequent elements to FALSE.

The `BRKB` instructions set active destination predicate elements to TRUE up to but excluding the first active TRUE element in their source predicate register, setting subsequent elements to FALSE.

The `BRKPA` and `BRKPB` instructions propagate the result of a previous `BRKB` or `BRKPB` instruction, by setting their destination predicate register to all FALSE if the *Last active element* of their first source predicate register is not TRUE, but otherwise generate the destination predicate from their second source predicate as described for the `BRKA` and `BRKB` instructions.

The BRKN instructions propagate the result of a previous `BRKB` or `BRKPB` instruction by setting the destination predicate register to all FALSE if the *Last active element* of their first source predicate register is not TRUE, but otherwise leave the destination predicate unchanged. The destination and second source predicate must have been created by another instruction, such as `RDFFR` or `WHILE`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-182 Data-independent loop instructions

Mnemonic	Instruction	See
BRKA	Break after the first true condition	BRKA
BRKAS	Break after the first true condition, setting the condition flags	BRKAS
BRKB	Break before the first true condition	BRKB
BRKBS	Break before the first true condition, setting the condition flags	BRKBS
BRKN	Propagate break to next partition	BRKN
BRKNS	Propagate break to next partition, setting the condition flags	BRKNS
BRKPA	Break after the first true condition, propagating from previous partition	BRKPA
BRKPAS	Break after the first true condition, propagating from previous partition, setting the condition flags	BRKPAS
BRKPB	Break before the first true condition, propagating from the previous partition	BRKPB
BRKPBS	Break before the first true condition, propagating from the previous partition, setting the condition flags	BRKPBS

C3.9.7.3 Serialized operations

The Serialized operation instructions permit *Active elements* within a vector to be processed sequentially without unpacking the vector. The condition flags are unconditionally set to control a subsequent conditional branch.

Table C3-183 Serialized operation instructions

Mnemonic	Instruction	See
PFIRST	Set the *First active element* to TRUE	PFIRST
PNEXT	Find next *Active element*	PNEXT
CTERMEQ	Compare and terminate loop when equal	CTERMEQ, CTERMNE
CTERMNE	Compare and terminate loop when not equal	CTERMEQ, CTERMNE

C3.9.8 SVE Move operations

C3.9.8.1 Element move and broadcast

The Element move and broadcast instructions copy data from scalar registers, immediate values, and other vectors to all vector elements or only to Active vector elements. The copied data might be in an integer or floating-point format.

Table C3-184 Element move and broadcast instructions

Mnemonic	Instruction	See
CPY	Copy signed integer immediate to Active vector elements	CPY (immediate, merging) CPY (immediate, zeroing)
	Copy general-purpose register to Active vector elements	CPY (scalar)
	Copy SIMD&FP scalar register to Active vector elements	CPY (SIMD&FP scalar)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
DUP	Broadcast signed immediate to all vector elements	DUP (immediate)
	Broadcast general-purpose register to all vector elements	DUP (scalar)
FCPY	Copy floating-point immediate to Active vector elements	FCPY
FDUP	Broadcast floating-point immediate to all vector elements	FDUP
FMOV	Move floating-point +0.0 to vector elements (unpredicated)	FMOV (zero, unpredicated)
	Move floating-point +0.0 to vector elements (predicated)	FMOV (zero, predicated)
	Move floating-point immediate to vector elements (unpredicated)	FMOV (immediate, unpredicated)
	Move floating-point immediate to vector element (predicated)	FMOV (immediate, predicated)
MOV	Move signed integer immediate to vector elements (unpredicated)	MOV (immediate, unpredicated)
	Move signed integer immediate to vector elements (predicated)	MOV (immediate, predicated, merging) MOV (immediate, predicated, zeroing)
	Move general-purpose register to vector elements (unpredicated)	MOV (scalar, unpredicated)
	Move general-purpose register to vector elements (predicated)	MOV (scalar, predicated)
	Move SIMD&FP scalar register to vector elements (unpredicated)	MOV (SIMD&FP scalar, unpredicated)
MOV	Move SIMD&FP scalar register to vector elements (predicated)	MOV (SIMD&FP scalar, predicated)
	Move vector register (unpredicated)	MOV (vector, unpredicated)
	Move vector register (predicated)	MOV (vector, predicated)
SEL	Select vector elements from two vectors	SEL (predicates) SEL (vectors)

C3.9.8.2 Element permute and shuffle

The Element permute and shuffle instructions move data between different vector elements, or between vector elements and scalar registers.

These instructions perform the following operations:

- Conditionally extract the *Last active element* of a vector or the following element.
  - The supported instructions are CLASTA and CLASTB.
- Unconditionally extract the *Last active element* of a vector or the following element.
  - The supported instructions are LASTA and LASTB.
- Variable permute instructions where the permutation is determined by the values in a predicate register or a table of element index values.
  - The supported instructions are COMPACT, SPLICE, and TBL.
- Fixed permute instructions where the form of the permutation is encoded in the instruction.
  - The supported instructions are: DUP, EXT, INSR, REV, REVB, REVH, REVW, SUNPKHI, SUNPKLO, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

Table C3-185 Element permute and shuffle instructions

Mnemonic	Instruction	See
CLASTA	Conditionally extract element after the *Last active element* to general-purpose register	CLASTA (scalar)
	Conditionally extract element after the *Last active element* to SIMD&FP scalar	CLASTA (SIMD&FP scalar)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Conditionally extract element after the *Last active element* to vector	CLASTA (vectors)
CLASTB	Conditionally extract *Last active element* to general-purpose register	CLASTB (scalar)
	Conditionally extract *Last active element* to SIMD&FP scalar	CLASTB (SIMD&FP scalar)
	Conditionally extract *Last active element* to vector	CLASTB (vectors)
LASTA	Extract element after the *Last active element* to general-purpose register	LASTA (scalar)
	Extract element after the *Last active element* to SIMD&FP scalar	LASTA (SIMD&FP scalar)
LASTB	Extract *Last active element* to general-purpose register	LASTB (scalar)
	Extract *Last active element* to SIMD&FP scalar	LASTA (SIMD&FP scalar)
COMPACT	Copy Active vector elements to lower-numbered elements	COMPACT
SPLICE	Splice two vectors under predicate control	SPLICE
TBL	Programmable table lookup using vector of element indexes	TBL
DUP	Broadcast indexed vector element	DUP (indexed)
EXT	Extract vector from pair of vectors	EXT
INSR	Insert general-purpose register into shifted vector	INSR (scalar)
	Insert SIMD&FP scalar register into shifted vector	INSR (SIMD&FP scalar)
MOV	Move indexed element or SIMD&FP scalar to vector (unpredicated)	MOV (SIMD&FP scalar, unpredicated)
	Move SIMD&FP scalar register to vector elements (predicated)	MOV (SIMD&FP scalar, predicated)
REV	Reverse all elements in vector	REV (vector)
REVB	Reverse 8-bit bytes in elements	REVB, REVH, REVW
REVH	Reverse 16-bit halfwords in elements	REVB, REVH, REVW
REVW	Reverse 32-bit words in elements	REVB, REVH, REVW
TRN1	Interleave even elements from two vectors	TRN1, TRN2 (vectors)
TRN2	Interleave odd elements from two vectors	TRN1, TRN2 (vectors)
UZP1	Concatenate even elements from two vectors	UZP1, UZP2 (vectors)
UZP2	Concatenate odd elements from two vectors	UZP1, UZP2 (vectors)
ZIP1	Interleave elements from low halves of two vectors	ZIP1, ZIP2 (vectors)
ZIP2	Interleave elements from high halves of two vectors	ZIP1, ZIP2 (vectors)

C3.9.8.3 Unpacking instructions

The Unpacking instructions unpack half of the elements from the source vector register or predicate register, widen the unpacked elements to twice the width, and place the result in the destination register.

Table C3-186 Unpacking instructions

Mnemonic	Instruction	See
SUNPKHI	Unpack and sign-extend elements from high half of vector	SUNPKHI, SUNPKLO
SUNPKLO	Unpack and sign-extend elements from low half of vector	SUNPKHI, SUNPKLO
UUNPKHI	Unpack and zero-extend elements from high half of vector	UUNPKHI, UUNPKLO
UUNPKLO	Unpack and zero-extend elements from low half of vector	UUNPKHI, UUNPKLO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
PUNPKHI	Unpack and widen elements from high half of predicate	PUNPKHI, PUNPKLO
PUNPKLO	Unpack and widen elements from low half of predicate	PUNPKHI, PUNPKLO

C3.9.8.4 Predicate permute

The Predicate instructions are used to move and permute predicate elements. These instructions generally mirror the fixed vector permutes to allow predicates to follow their data. The permutes move all of the bits in a predicate element, not just the canonical bits.

Table C3-187 Predicate instructions

Mnemonic	Instruction	See
REV	Reverse all elements in predicate	REV (predicate)
TRN1	Interleave even elements from two predicates	TRN1, TRN2 (predicates)
TRN2	Interleave odd elements from two predicates	TRN1, TRN2 (predicates)
UZP1	Select even elements from two predicates	UZP1, UZP2 (predicates)
UZP2	Select odd elements from two predicates	UZP1, UZP2 (predicates)
ZIP1	Interleave elements from low halves of two predicates	ZIP1, ZIP2 (predicates)
ZIP2	Interleave elements from high halves of two predicates	ZIP1, ZIP2 (predicates)

C3.9.9 Index vector generation

The INDEX instruction initializes a vector horizontally by setting its first element to an integer value, and then repeatedly incrementing it by a second integer value to generate the subsequent elements. Each integer value can be specified as a signed immediate or a general-purpose register.

Table C3-188 Index instructions

Mnemonic	Instruction	See
INDEX	Create index vector starting from and incremented by immediates	INDEX (immediates)
	Create index vector starting from immediate and incremented by general-purpose register	INDEX (scalars)
	Create index vector starting from general-purpose register and incremented by immediate	INDEX (scalar, immediate)
	Create index vector starting from immediate and incremented by general-purpose registers	INDEX (immediate, scalar)

C3.9.10 Move prefix

The MOVPRFX (predicated) instruction is a predicated vector move that can be combined with a predicated destructive instruction that immediately follows it, in program order, to create a single constructive operation, or to convert an instruction with merging predication to use zeroing predication.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The MOVPRFX (unpredicated) instruction is an unpredicated vector move that can be combined with a predicated or unpredicated destructive instruction that immediately follows it, in program order, to create a single constructive operation.

The *Operational information* section of an SVE instruction description indicates whether or not an instruction can be predictably prefixed by a MOVPRFX instruction. If the *Operational information* of an *SVE* instruction description does not mention MOVPRFX or if the section does not exist, then the instruction cannot be predictably prefixed by a MOVPRFX instruction.

The prefixed instruction that immediately follows a MOVPRFX instruction in program order must be an implemented SVE instruction that can be predictably prefixed by a MOVPRFX instruction, or an A64 HLT instruction, or an A64 BRK instruction. For an SVE instruction that can be predictably prefixed by a MOVPRFX instruction, all of the following apply:

- The destination register field implicitly specifies one of the source operands, which means that it is a destructive binary or ternary vector operation or unary operation with merging predication, excluding MOVPRFX.
- The destination register is the same as the MOVPRFX destination register.
- The prefixed instruction does not use the MOVPRFX destination register in any of its other source register fields, even if it has a different name but refers to the same architectural register state. For example, Z1, V1, and D1 all refer to the same architectural register.
- If the MOVPRFX instruction is predicated, then the prefixed instruction is predicated using the same *Governing predicate* register, and the maximum encoded element size is the same as the MOVPRFX element size, excluding the fixed-size 64-bit elements of the wide elements form of bitwise shift and integer compare operations.
- If the MOVPRFX instruction is unpredicated, then the prefixed instruction can use any *Governing predicate* register and element size, or it can be unpredicated. A predicated MOVPRFX cannot be used with an unpredicated instruction.

If the instruction that follows a MOVPRFX instruction is not an implemented SVE instruction that can be predictably prefixed by a MOVPRFX instruction, the two instructions behave in one of the following CONSTRAINED UNPREDICTABLE ways:

- Either or both instructions can execute with their individually described effects.
- Either instruction can generate an Undefined Instruction exception.
- Either or both instructions can execute as a NOP.
- The second instruction can execute with an UNKNOWN value for any of its source registers.
- Any register that is written by either or both instructions can be set to an unknown value.
- A control flow instruction that writes the PC can set the PC to an unknown value.

Table C3-189 Move prefix instructions

Mnemonic	Instruction	See
MOVPRFX	Move prefix (predicated)	MOVPRFX (predicated)
	Move prefix (unpredicated)	MOVPRFX (unpredicated)

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that, for performance reasons, software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

When a MOVPRFX instruction is executed, except for PMU events SVE_MOVPRFX_SPEC, SVE_MOVPRFX_Z_SPEC, SVE_MOVPRFX_M_SPEC, and SVE_MOVPRFX_U_SPEC, 0x807C-0x807F, it is IMPLEMENTATION DEFINED for each execution of the instruction whether or not any Performance Monitor counts the instruction. This can vary dynamically for each execution of the same instruction.

When a microarchitectural operation is executed because of a MOVPRFX instruction, except for PMU events SVE_MOVPRFX_SPEC, SVE_MOVPRFX_Z_SPEC, SVE_MOVPRFX_M_SPEC, and SVE_MOVPRFX_U_SPEC, 0x807C-0x807F, it is IMPLEMENTATION DEFINED for each execution of the operation whether or not the Performance Monitor counts the operation. This can vary dynamically for each execution of the same instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.9.10.1 MOVPRFX instruction behavior in self-hosted debug

A `MOVPRFX` instruction can legally prefix a `BRK` or `HLT` instruction.

If a hardware breakpoint is programmed with the address of a legal `MOVPRFX` instruction, when any of the following events occur, the hardware breakpoint generates a Breakpoint exception:

- The `MOVPRFX` instruction is committed for execution.
- The combined `MOVPRFX` and Prefixed instruction is committed for execution.

If a hardware breakpoint is programmed with the address of an illegal `MOVPRFX` instruction or a Prefixed instruction, when any of the `MOVPRFX` instruction and Prefixed instruction are committed for execution, it is CONSTRAINED UNPREDICTABLE whether or not the hardware breakpoint generates a Breakpoint exception.

If a single-step is performed for a `MOVPRFX` instruction, it is CONSTRAINED UNPREDICTABLE whether the PE steps over the pair of instructions or steps over only the `MOVPRFX` instruction.

C3.9.11 Reduction operations

C3.9.11.1 Horizontal reductions

The Horizontal reduction instructions perform arithmetic horizontally across *Active elements* of a single source vector and deliver a scalar result.

The floating-point horizontal accumulating sum instruction, `FADDA`, operates strictly in order of increasing element number across a vector, using the scalar destination register as a source for the initial value of the accumulator. This preserves the original program evaluation order where non-associativity is required.

The other floating-point reductions calculate their result using a recursive pair-wise algorithm that does not preserve the original program order, but permits increased parallelism for code that does not require strict order of evaluation.

Integer reductions are fully associative, and the order of evaluation is not specified by the architecture.

Table C3-190 Horizontal reduction instructions

Mnemonic	Instruction	See
`ANDV`	Bitwise AND reduction, treating *Inactive elements* as all ones	ANDV
`EORV`	Bitwise EOR reduction, treating *Inactive elements* as zero	EORV
`FADDA`	Floating-point add strictly-ordered reduction, accumulating in scalar, ignoring *Inactive elements*	FADDA
`FADDV`	Floating-point add recursive reduction, treating *Inactive elements* as +0.0	FADDV
`FMAXNMV`	Floating-point maximum number recursive reduction, treating *Inactive elements* as the default NaN	FMAXNMV
`FMAXV`	Floating-point maximum recursive reduction, treating *Inactive elements* as negative infinity	FMAXV
`FMINNMV`	Floating-point minimum number recursive reduction, treating *Inactive elements* as the default NaN	FMINNMV
`FMINV`	Floating-point minimum recursive reduction, treating *Inactive elements* as positive infinity	FMINV
`ORV`	Bitwise OR reduction, treating *Inactive elements* as zero	ORV
`SADDV`	Signed add reduction, treating *Inactive elements* as zero	SADDV
`SMAXV`	Signed maximum reduction, treating *Inactive elements* as the minimum signed integer	SMAXV
`SMINV`	Signed minimum reduction, treating *Inactive elements* the maximum signed integer	SMINV
`UADDV`	Unsigned add reduction, treating *Inactive elements* as zero	UADDV
`UMAXV`	Unsigned maximum reduction, treating *Inactive elements* as zero	UMAXV
`UMINV`	Unsigned minimum reduction, treating *Inactive elements* as the maximum unsigned integer	UMINV

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10 Data processing - SVE2

The following subsections describe the SVE2 processing instructions:

- Down-counting loops.
- Constructive multiply.
- Uniform DSP operations.
- Widening DSP operations.
- Narrowing DSP operation.
- Unary narrowing operations.
- Non-widening pairwise arithmetic.
- Widening pairwise arithmetic.
- Bitwise ternary logical instructions.
- Large integer arithmetic.
- Multiplication by indexed elements.
- Complex integer arithmetic.
- Conversions.
- Floating-point widening multiply-accumulate.
- Floating-point integer binary logarithm.
- Cross-lane match detect.
- Bit permutation.
- Polynomial arithmetic.
- Vector permutation.
- Table lookup.
- Cryptography support.
- BFloat16 arithmetic.
- BFloat16 minimum/maximum.
- Floating-point adjust exponent.
- Floating-point multiply-accumulate.
- Dot product.
- Floating-point matrix multiply-accumulate.
- Floating-point round.
- Quadword operations.
- Integer shift and convert.
- Predicate element index.
- Multi-vector predication.
- Permute vector.
- Predicate pair loop control.
- Predicate move.
- Predicate select.

C3.10.1 Down-counting loops

A down-counting `WHILE` instruction decrements the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the highest-numbered element are true while the comparison remains true.
- The predicate elements thereafter, down to the lowest-numbered element, are false when the comparison becomes false.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is decremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the minimum integer value of the selected size and type of comparison.

Table C3-191 Down counting loop instructions

Mnemonic	Instruction	See
`WHILEGE`	While decrementing signed 32-bit or 64-bit scalar greater than or equal to scalar	WHILEGE (predicate)
`WHILEGT`	While decrementing signed 32-bit or 64-bit scalar greater than scalar	WHILEGT (predicate)
`WHILEHI`	While decrementing unsigned 32-bit or 64-bit scalar higher than scalar	WHILEHI (predicate)
`WHILEHS`	While decrementing unsigned 32-bit or 64-bit scalar higher or same as scalar	WHILEHS (predicate)

C3.10.2 Constructive multiply

SVE2 includes the following constructive, three-operand versions of the integer multiply instructions:

Table C3-192 Constructive multiply instructions

Mnemonic	Instruction	See
`MUL`	Multiply vectors (unpredicated)	MUL (vectors, unpredicated)
`SMULH`	Signed multiply returning high half (unpredicated)	SMULH (unpredicated)
`UMULH`	Unsigned multiply returning high half (unpredicated)	UMULH (unpredicated)

C3.10.3 Uniform DSP operations

The uniform DSP instructions are based on AArch64 Advanced SIMD instructions with the same mnemonic. The instructions operate on integer operands and produce results with a uniform element size. The operation of an instruction might include one or more of rounding, halving, saturation, and accumulation.

Table C3-193 Uniform DSP instructions

Mnemonic	Instruction	See
`SABA`	Signed absolute difference and accumulate	SABA
`SHADD`	Signed halving addition	SHADD
`SHSUB`	Signed halving subtract	SHSUB
`SHSUBR`	Signed halving subtract reversed vectors	SHSUBR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SLI	Shift left and insert (immediate)	SLI
SQABS	Signed saturating absolute value	SQABS
SQADD	Signed saturating addition (predicated)	SQADD (vectors, predicated)
SQDMULH	Signed saturating doubling multiply high (unpredicated)	SQDMULH (vectors)
SQNEG	Signed saturating negate	SQNEG
SQRDMLAH	Signed saturating rounding doubling multiply-add high to accumulator (unpredicated)	SQRDMLAH (indexed)
SQRDMLSH	Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated)	SQRDMLSH (vectors)
SQRDMULH	Signed saturating rounding doubling multiply high (unpredicated)	SQRDMULH (vectors)
SQRSHL	Signed saturating rounding shift left by vector (predicated)	SQRSHL
SQRSHLR	Signed saturating rounding shift left reversed vectors (predicated)	SQRSHLR
SQSHL	Signed saturating shift left by immediate	SQSHL (immediate)
SQSHL	Signed saturating shift left by vector (predicated)	SQSHL (vectors)
SQSHLR	Signed saturating shift left reversed vectors (predicated)	SQSHLR
SQSHLU	Signed saturating shift left unsigned by immediate	SQSHLU
SQSUB	Signed saturating subtraction (predicated)	SQSUB (vectors, predicated)
SQSUBR	Signed saturating subtraction reversed vectors (predicated)	SQSUBR
SRHADD	Signed rounding halving addition	SRHADD
SRI	Shift right and insert (immediate)	SRI
SRSHL	Signed rounding shift left by vector (predicated)	SRSHL
SRSHLR	Signed rounding shift left reversed vectors (predicated)	SRSHLR
SRSHR	Signed rounding shift right by immediate	SRSHR
SRSRA	Signed rounding shift right and accumulate (immediate)	SRSRA
SSRA	Signed shift right and accumulate (immediate)	SSRA
SUQADD	Signed saturating addition of unsigned value	SUQADD
UABA	Unsigned absolute difference and accumulate.	UABA
UHADD	Unsigned halving addition.	UHADD
UHSUB	Unsigned halving subtract	UHSUB
UHSUBR	Unsigned halving subtract reversed vectors	UHSUBR
UQADD	Unsigned saturating addition (predicated)	UQADD (vectors, predicated)
UQRSHL	Unsigned saturating rounding shift left by vector (predicated)	UQRSHL
UQRSHLR	Unsigned saturating rounding shift left reversed vectors (predicated)	UQRSHLR
UQSHL	Unsigned saturating shift left by immediate	UQSHL (immediate)
UQSHL	Unsigned saturating shift left by vector (predicated)	UQSHL (vectors)
UQSHLR	Unsigned saturating shift left reversed vectors (predicated)	UQSHLR
UQSUB	Unsigned saturating subtraction (predicated)	UQSUB (vectors, predicated)
UQSUBR	Unsigned saturating subtraction reversed vectors (predicated)	UQSUBR
URECPE	Unsigned reciprocal estimate (predicated)	URECPE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
URHADD	Unsigned rounding halving addition.	URHADD
URSHL	Unsigned rounding shift left by vector (predicated)	URSHL
URSHLR	Unsigned rounding shift left reversed vectors (predicated)	URSHLR
URSHR	Unsigned rounding shift right by immediate	URSHR
URSQRTE	Unsigned reciprocal square root estimate (predicated)	URSQRTE
URSRA	Unsigned rounding shift right and accumulate (immediate)	URSRA
USQADD	Unsigned saturating addition of signed value	USQADD
USRA	Unsigned shift right and accumulate (immediate)	USRA

C3.10.4 Widening DSP operations

The widening DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on integer values and produce results that are twice the width of some or all of the inputs. The instructions read the narrow inputs from either the even-numbered (bottom) or odd-numbered (top) source elements and place each result in the double-width destination elements that overlap the narrow source elements.

Table C3-194 Widening DSP instructions

Mnemonic	Instruction	See
SABALB	Signed absolute difference and accumulate long (bottom)	SABALB
SABALT	Signed absolute difference and accumulate long (top)	SABALT
SABDLB	Signed absolute difference long (bottom)	SABDLB
SABDLT	Signed absolute difference long (top)	SABDLT
SADDLB	Signed add long (bottom)	SADDLB
SADDLT	Signed add long (top)	SADDLT
SADDWB	Signed add wide (bottom)	SADDWB
SADDWT	Signed add wide (top)	SADDWT
SMLALB	Signed multiply-add long to accumulator (bottom)	SMLALB (vectors)
SMLALT	Signed multiply-add long to accumulator (top)	SMLALT (vectors)
SMLSLB	Signed multiply-subtract long from accumulator (bottom)	SMLSLB (vectors)
SMLSLT	Signed multiply-subtract long from accumulator (top)	SMLSLT (vectors)
SMULLB	Signed multiply long (bottom)	SMULLB (vectors)
SMULLT	Signed multiply long (top)	SMULLT (vectors)
SQDMLALB	Signed saturating doubling multiply-add long to accumulator (bottom)	SQDMLALB (vectors)
SQDMLALT	Signed saturating doubling multiply-add long to accumulator (top)	SQDMLALT (vectors)
SQDMLSLB	Signed saturating doubling multiply-subtract long from accumulator (bottom)	SQDMLSLB (vectors)
SQDMLSLT	Signed saturating doubling multiply-subtract long from accumulator (top)	SQDMLSLT (vectors)
SQDMULLB	Signed saturating doubling multiply long (bottom)	SQDMULLB (vectors)
SQDMULLT	Signed saturating doubling multiply long (top)	SQDMULLT (vectors)
SSHLLB	Signed shift left long by immediate (bottom)	SSHLLB

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SSHLLT	Signed shift left long by immediate (top)	SSHLLT
SSUBLB	Signed subtract long (bottom)	SSUBLB
SSUBLT	Signed subtract long (top)	SSUBLT
SSUBWB	Signed subtract wide (bottom)	SSUBWB
SSUBWT	Signed subtract wide (top)	SSUBWT
UABALB	Unsigned absolute difference and accumulate long (bottom)	UABALB
UABALT	Unsigned absolute difference and accumulate long (top)	UABALT
UABDLB	Unsigned absolute difference long (bottom)	UABDLB
UABDLT	Unsigned absolute difference long (top)	UABDLT
UADDLB	Unsigned add long (bottom)	UADDLB
UADDLT	Unsigned add long (top)	UADDLT
UADDWB	Unsigned add wide (bottom)	UADDWB
UADDWT	Unsigned add wide (top)	UADDWT
UMLALB	Unsigned multiply-add long to accumulator (bottom)	UMLALB (vectors)
UMLALT	Unsigned multiply-add long to accumulator (top)	UMLALT (vectors)
UMLSLB	Unsigned multiply-subtract long from accumulator (bottom)	UMLSLB (vectors)
UMLSLT	Unsigned multiply-subtract long from accumulator (top)	UMLSLT (vectors)
UMULLB	Unsigned multiply long (bottom)	UMULLB (vectors)
UMULLT	Unsigned multiply long (top)	UMULLT (vectors)
USHLLB	Unsigned shift left long by immediate (bottom)	USHLLB
USHLLT	Unsigned shift left long by immediate (top)	USHLLT
USUBLB	Unsigned subtract long (bottom)	USUBLB
USUBLT	Unsigned subtract long (top)	USUBLT
USUBWB	Unsigned subtract wide (bottom)	USUBWB
USUBWT	Unsigned subtract wide (top)	USUBWT

C3.10.5 Narrowing DSP operation

The narrowing DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on integer values and produce results that are half the width of the inputs. The instructions read wide source elements and place each result in one of the following:

- The overlapped even-numbered (bottom) half-width destination elements. The odd-numbered destination elements are set to zero.
- The overlapped odd-numbered (top) half-width destination elements. The even-numbered destination elements are unchanged, which means that the instructions are implicitly merging operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-195 Narrowing DSP instructions

Mnemonic	Instruction	See
ADDHNB	Add narrow high part (bottom)	ADDHNB
ADDHNT	Add narrow high part (top)	ADDHNT
RADDHNB	Rounding add narrow high part (bottom)	RADDHNB
RADDHNT	Rounding add narrow high part (top)	RADDHNT
RSHRNB	Rounding shift right narrow by immediate (bottom)	RSHRNB
RSHRNT	Rounding shift right narrow by immediate (top)	RSHRNT
RSUBHNB	Rounding subtract narrow high part (bottom)	RSUBHNB
RSUBHNT	Rounding subtract narrow high part (top)	RSUBHNT
SHRNB	Shift right narrow by immediate (bottom)	SHRNB
SHRNT	Shift right narrow by immediate (top)	SHRNT
SQRSHRNB	Signed saturating rounding shift right narrow by immediate (bottom)	SQRSHRNB
SQRSHRNT	Signed saturating rounding shift right narrow by immediate (top)	SQRSHRNT
SQRSHRUNB	Signed saturating rounding shift right unsigned narrow by immediate (bottom)	SQRSHRUNB
SQRSHRUNT	Signed saturating rounding shift right unsigned narrow by immediate (top)	SQRSHRUNT
SQSHRNB	Signed saturating shift right narrow by immediate (bottom)	SQSHRNB
SQSHRNT	Signed saturating shift right narrow by immediate (top)	SQSHRNT
SQSHRUNB	Signed saturating shift right unsigned narrow by immediate (bottom)	SQSHRUNB
SQSHRUNT	Signed saturating shift right unsigned narrow by immediate (top)	SQSHRUNT
SUBHNB	Subtract narrow high part (bottom)	SUBHNB
SUBHNT	Subtract narrow high part (top)	SUBHNT
UQRSHRNB	Unsigned saturating rounding shift right narrow by immediate (bottom)	UQRSHRNB
UQRSHRNT	Unsigned saturating rounding shift right narrow by immediate (top)	UQRSHRNT
UQSHRNB	Unsigned saturating shift right narrow by immediate (bottom)	UQSHRNB
UQSHRNT	Unsigned saturating shift right narrow by immediate (top)	UQSHRNT

C3.10.6 Unary narrowing operations

The unary narrowing instructions are unpredicated and do not write to the source register. The instructions read elements from the source vector and saturate each value to the half-width destination element size. The instructions place the narrow results in one of the following:

- The overlapped even-numbered (bottom) half-width elements. The odd-numbered elements are set to zero.
- The overlapped odd-numbered (top) half-width elements. The even-numbered elements are unchanged.

Non-saturating (truncating) conversions can be performed using existing SVE instructions such as shifts, masks, and permutes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-196 Unary narrowing instructions

Mnemonic	Instruction	See
SQXTNB	Signed saturating extract narrow (bottom)	SQXTNB
SQXTNT	Signed saturating extract narrow (top)	SQXTNT
SQXTUNB	Signed saturating unsigned extract narrow (bottom)	SQXTUNB
SQXTUNT	Signed saturating unsigned extract narrow (top)	SQXTUNT
UQXTNB	Unsigned saturating extract narrow (bottom)	UQXTNB
UQXTNT	Unsigned saturating extract narrow (top)	UQXTNT

C3.10.7 Non-widening pairwise arithmetic

The non-widening pairwise arithmetic instructions operate on pairs of adjacent elements in each source vector and produce a result element that is the same size as a single input element. The results from the first and second source vectors are interleaved, so that the source and result elements overlap. The result is destructively placed in the first source vector. The AArch64 Advanced SIMD instructions do not interleave the results from the first and second source vectors. Predication applies to the destination vector. The even-numbered predicate elements enable an operation on a pair of elements in the first source vector. The odd-numbered predicate elements enable an operation on a pair of elements in the second source vector. *Inactive elements* in the destination vector register are not modified.

Table C3-197 Non-widening pairwise arithmetic instructions

Mnemonic	Instruction	See
ADDP	Add pairwise	ADDP
FADDP	Floating-point add pairwise	FADDP
FMAXNMP	Floating-point maximum number pairwise	FMAXNMP
FMAXP	Floating-point maximum pairwise	FMAXP
FMINNMP	Floating-point minimum number pairwise	FMINNMP
FMINP	Floating-point minimum pairwise	FMINP
SMAXP	Signed maximum pairwise	SMAXP
SMINP	Signed minimum pairwise	SMINP
UMAXP	Unsigned maximum pairwise	UMAXP
UMINP	Unsigned minimum pairwise	UMINP

C3.10.8 Widening pairwise arithmetic

The widening pairwise arithmetic instructions operate on pairs of adjacent elements in a single source vector and produce a double-width result element that is accumulated into the destination vector. *Inactive elements* in the destination vector register are not modified.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-198 Widening pairwise arithmetic instructions

Mnemonic	Instruction	See
SADALP	Signed add and accumulate long pairwise	SADALP
UADALP	Unsigned add and accumulate long pairwise	UADALP

C3.10.9 Bitwise ternary logical instructions

The bitwise ternary logical instructions enable complex bit processing codes to be accelerated using multiple bitwise logical operations in a shorter instruction sequence. All of the following operations are supported by the bitwise ternary logical instructions:

- The BCAX instruction combines a ternary bitwise clear with an exclusive-OR.
- The EOR3 instruction provides a ternary exclusive-OR.
- The XAR instruction combines an exclusive-OR with rotation by a constant amount.
- The bitwise select instructions, BSL, BSL1N, BSL2N, and NBSL, can be used with other bitwise logical instructions to generate all 256 possible bitwise combinations of three input bits using at most three instructions.

The bitwise ternary logical instructions are unpredicated.

Table C3-199 Bitwise ternary logical instructions

Mnemonic	Instruction	See
BCAX	Bitwise clear and exclusive-OR	BCAX
BSL	Bitwise select	BSL
BSL1N	Bitwise select with first input inverted	BSL1N
BSL2N	Bitwise select with second input inverted	BSL2N
EOR3	Bitwise exclusive-OR of three vectors	EOR3
NBSL	Bitwise inverted select	NBSL
XAR	Bitwise exclusive-OR and rotate right by immediate	XAR

C3.10.10 Large integer arithmetic

The large integer arithmetic instructions aid the processing of large integers in vector registers by maintaining multiple carry chains that are interleaved in accumulator vectors.A large integer arithmetic instruction takes as input all of the following:

- Either the even-numbered (bottom) or odd-numbered (top) elements of the first source vector.
- A 1-bit carry input from the least-significant bit of the odd-numbered elements of the second source vector.

The inputs to the instruction are added to or subtracted from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-200 Large integer arithmetic instructions

Mnemonic	Instruction	See
ADCLB	Add with carry long (bottom)	ADCLB
ADCLT	Add with carry long (top)	ADCLT
SBCLB	Subtract with carry long (bottom)	SBCLB
SBCLT	Subtract with carry long (top)	SBCLT

C3.10.11 Multiplication by indexed elements

The multiplication by indexed elements instructions take all integer elements in each 128-bit vector segment of the first source vector and multiplies them by the indexed element in the corresponding segment of the second source vector.

The products might be destructively added to or subtracted from the corresponding elements of an addend vector.

The second source vector elements are specified using an immediate index that selects the same element position in each 128-bit vector segment. The index range is 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the element size.

Table C3-201 Multiplication by indexed elements instructions

Mnemonic	Instruction	See
MLA	Multiply-add to accumulator (indexed)	MLA (indexed)
MLS	Multiply-subtract from accumulator (indexed)	MLS (indexed)
MUL	Multiply (indexed)	MUL (indexed)
SMLALB	Signed multiply-add long to accumulator (bottom, indexed)	SMLALB (indexed)
SMLALT	Signed multiply-add long to accumulator (top, indexed)	SMLALT (indexed)
SMLSLB	Signed multiply-subtract long from accumulator (bottom, indexed)	SMLSLB (indexed)
SMLSLT	Signed multiply-subtract long from accumulator (top, indexed)	SMLSLT (indexed)
SQDMLALT	Signed saturating doubling multiply-add long to accumulator (top, indexed)	SQDMLALT (indexed)
SQDMLSLB	Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed)	SQDMLSLB (indexed)
SQDMLSLT	Signed saturating doubling multiply-subtract long from accumulator (top, indexed)	SQDMLSLB (indexed)
SQDMULLT	Signed saturating doubling multiply long (top, indexed)	SQDMULLT (indexed)
SQRDMLAH	Signed saturating rounding doubling multiply-add high to accumulator (indexed)	SQRDMLAH (indexed)
SQRDMLSH	Signed saturating rounding doubling multiply-subtract high from accumulator (indexed)	SQRDMLSH (indexed)
SQRDMULH	Signed saturating rounding doubling multiply high (indexed)	SQRDMULH (indexed)
UMLALB	Unsigned multiply-add long to accumulator (bottom, indexed)	UMLALB (indexed)
UMLALT	Unsigned multiply-add long to accumulator (top, indexed)	UMLALT (indexed)
UMLSLB	Unsigned multiply-subtract long from accumulator (bottom, indexed)	UMLSLB (indexed)
UMLSLT	Unsigned multiply-subtract long from accumulator (top, indexed)	UMLSLT (indexed)
UMULLB	Unsigned multiply long (bottom, indexed)	UMULLB (indexed)
UMULLT	Unsigned multiply long (top, indexed)	UMULLT (indexed)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.12 Complex integer arithmetic

Complex integer arithmetic instructions operate on signed integer complex numbers in vectors containing the following interleaved element pairs:

- The even-numbered elements contain the real parts of the complex numbers.
- The odd-numbered elements contain the imaginary parts of the complex numbers.

C3.10.12.1 Uniform complex integer arithmetic

The uniform complex integer arithmetic instructions operate on vectors containing integral complex numbers. The instructions operate on complex numbers that are in polar form.

The `CADD` instructions rotate the complex numbers in the second source vector before adding element pairs to the corresponding elements of the first source vector, in a destructive manner. The rotation direction is 90 degrees or 270 degrees from the positive real axis towards the positive imaginary axis.

The `CMLA` instructions transform the operands to enable multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The following transformations are done:

- The complex numbers in the second source vector are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are added to the corresponding components of the destination and addend vector.

Two `CMLA` instructions can be used, as follows:

- `CMLA Zda.S, Zn.S, Zm.S, #A`
- `CMLA Zda.S, Zn.S, Zm.S, #B`

Some meaningful combinations of A and B are:

- A=0, B=90. The complex number vectors, Zn and Zm, are multiplied and the products are added to the complex numbers in Zda.
- A=0, B=270. The complex number conjugates in Zn are multiplied by the complex numbers in Zm and the products are added to the complex numbers in Zda.
- A=180, B=270. The two complex number vectors, Zn and Zm, are multiplied and the products are subtracted from the complex numbers in Zda.
- A=180, B=90. The complex number conjugates in Zn are multiplied by the complex numbers in Zm and the products are subtracted from the complex numbers in Zda.

The `CMLA` indexed form uses a single complex number in each 128-bit segment of the second source vector as the multiplier for all complex numbers in the corresponding first source vector segment. The complex numbers in the second source vector are specified using an immediate index that selects the same complex number position in each 128-bit vector segment. The index range is 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the complex number size.

Table C3-202 Uniform complex integer arithmetic instructions

Mnemonic	Instruction	See
`CADD`	Complex integer add with rotate	CADD
`CMLA`	Complex integer multiply-add with rotate	CMLA (vectors)
	Complex integer multiply-add with rotate (indexed)	CMLA (indexed)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SQCADD	Saturating complex integer add with rotate	SQCADD
SQRDCMLAH	Saturating rounding doubling complex integer multiply-add high with rotate	SQRDCMLAH (vectors)
SQRDCMLAH	Saturating rounding doubling complex integer multiply-add high with rotate (indexed)	SQRDCMLAH (indexed)

C3.10.12.2 Widening complex integer arithmetic

The widening complex integer instructions deinterleave the real and imaginary components of integral complex numbers, and generate complex result components that have a higher numeric precision than the input values. The instructions differ from other complex instructions that process the real and imaginary components of complex numbers and write the complex result components to the destination.

The following instructions are useful when generating the widened components of the result of a complex multiply-add:

- SQDMLALBT, the imaginary results.
- SQDMLSLT, the real results.
- SQDMLALB, the conjugate real results.
- SQDMLSLBT, the conjugate imaginary results.

The following instructions are useful when generating the widened components of the result of a complex addition (X + jY) or (X - jY), given complex numbers X and Y:

- SADDLBT, the imaginary results when calculating (X + jY) or real values when calculating (X - jY).
- SSUBLBT, the real results when calculating (X + jY) or imaginary values when calculating (X - jY).

Table C3-203 Widening complex integer arithmetic instructions

Mnemonic	Instruction	See
SADDLBT	Signed add long (bottom + top)	SADDLBT
SQDMLALBT	Signed saturating doubling multiply-add long to accumulator (bottom × top)	SQDMLALBT
SQDMLSLBT	Signed saturating doubling multiply-subtract long from accumulator (bottom × top)	SQDMLSLBT
SSUBLBT	Signed subtract long (bottom - top)	SSUBLBT
SSUBLTB	Signed subtract long (top - bottom)	SSUBLTB

C3.10.12.3 Complex integer dot product

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. The complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector. The wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element that overlaps all four of the elements that form a pair of complex number values in the first source vector.

Each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates four wide real sums or four wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180, or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, considered in polar form, by applying the following transformations before the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector accumulates the imaginary parts of a complex dot product.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the rotation is #180, there is no transformation. The destination vector accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of the instruction selects a single complex number pair within each 128-bit segment of the second source vector to multiply with all complex number pairs within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even and odd pair of elements. The real part is the even-numbered element and the imaginary part is the odd-numbered element.

Table C3-204 Complex integer dot product instructions

Mnemonic	Instruction	See
`CDOT`	Complex integer dot product	CDOT (vectors)
	Complex integer dot product (indexed)	CDOT (indexed)

C3.10.13 Conversions

C3.10.13.1 Floating-point conversions

The floating-point extra conversion instructions convert to and from fully packed vectors of narrower floating-point elements.

The `FCVTLT` instruction converts the top or odd-numbered narrow floating-point vector elements to wider elements of the next higher precision. The conversion is similar to what is done by the widening integer instructions.

The `FCVTNT` and `FCVXNT` instructions convert wider floating-point vector elements to the top or odd-numbered narrower elements of the next lower precision. The conversion is similar to what is done by the narrowing integer instructions.

The `FCVTXNT` and `FCVTX` instructions convert from double-precision to fully packed half-precision in two narrowing steps, double-precision to single-precision and then single-precision to half-precision. The two-step conversion is done without an intermediate rounding error by using von Neumann rounding, which rounds an inexact mantissa to an odd value.

The existing SVE `FCVT` instructions implement the corresponding widening and narrowing conversions on the bottom or even-numbered half-width elements.

Table C3-205 Floating-point extra conversion instructions

Mnemonic	Instruction	See
`FCVTLT`	Floating-point up convert long (top, predicated)	FCVTLT
`FCVTNT`	Floating-point down convert narrow (top, predicated)	FCVTNT (predicated)
`FCVTX`	Floating-point down convert, rounding to odd (predicated)	FCVTX
`FCVTXNT`	Floating-point down convert, rounding to odd (top, predicated)	FCVTXNT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.13.2 FP8 floating-point conversions

Convert each floating-point element of the source vector while scaling the value, and place the results in the corresponding elements of the destination vector.

For instructions that convert other floating-point formats to FP8 formats, each input element is scaled by adding the value specified in FPMR.NSCALE to its unbounded exponent during the conversion.

For instructions that convert FP8 formats to other floating-point formats, each input element is scaled by subtracting the value specified in FPMR.LSCALE or FPMR.LSCALE2 from its unbounded exponent during the conversion.

For more information, see the instruction descriptions.

Table C3-206 FP8 floating-point convert instructions

Mnemonic	Instruction	See
`BF1CVT, BF2CVT`	8-bit floating-point convert to BFloat16	BF1CVT, BF2CVT
`BF1CVTLT, BF2CVTLT`	8-bit floating-point convert to BFloat16 (top)	BF1CVTLT, BF2CVTLT
`BFCVTN`	BFloat16 convert, narrow and interleave to 8-bit floating-point	BFCVTN
`F1CVT, F2CVT`	8-bit floating-point convert to half-precision	F1CVT, F2CVT
`F1CVTLT, F2CVTLT`	8-bit floating-point convert to half-precision (top)	F1CVTLT, F2CVTLT
`FCVTN`	Half-precision convert, narrow and interleave to 8-bit floating-point	FCVTN
`FCVTNB`	Single-precision convert, narrow and interleave to 8-bit floating-point (bottom)	FCVTNB
`FCVTNT`	Single-precision convert, narrow and interleave to 8-bit floating-point (top)	FCVTNT (unpredicated)

C3.10.14 Multiply-accumulate

C3.10.14.1 Floating-point widening multiply-accumulate

The floating-point widening multiply-accumulate instructions multiply the even-numbered or odd-numbered half-precision elements of the two source vectors and then destructively add or subtract the single-precision intermediate products. Intermediate rounding is not done. The result is placed into the overlapping single-precision elements of the addend vector.

The instructions implicitly convert the half-precision inputs to single-precision and can be used to mitigate the impact of round-off errors when accumulating half-precision floating-point values over many iterations.

The instructions are unpredicated and preserve the multiplier and multiplicand source vectors.

Table C3-207 Floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
`FMLALB`	Floating-point fused multiply-add long to accumulator (bottom)	FMLALB (vectors)
`FMLALB`	Floating-point fused multiply-add long to accumulator (bottom, indexed)	FMLALB (indexed)
`FMLALT`	Floating-point fused multiply-add long to accumulator (top)	FMLALT (vectors)
`FMLALT`	Floating-point fused multiply-add long to accumulator (top, indexed)	FMLALT (indexed)
`FMLSLB`	Floating-point fused multiply-subtract long from accumulator (bottom)	FMLSLB (vectors)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FMLSLB	Floating-point fused multiply-subtract long from accumulator (bottom, indexed)	FMLSLB (indexed)
FMLSLT	Floating-point fused multiply-subtract long from accumulator (top)	FMLSLT (vectors)
FMLSLT	Floating-point fused multiply-subtract long from accumulator (top, indexed)	FMLSLT (indexed)

C3.10.14.2 FP8 floating-point widening multiply-accumulate

The FP8 floating-point multiply-accumulate instructions widen the 8-bit floating-point elements in the source vectors to half-precision or single-precision format and multiply the corresponding elements. They generate a fused, downscaled product and accumulate into the corresponding elements of the destination vector, without intermediate rounding.

For these instructions, the unbounded intermediate product is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-208 FP8 floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
FMLALB	8-bit floating-point multiply-add long to half-precision (bottom, indexed)	FMLALB (indexed, FP8 to FP16)
	8-bit floating-point multiply-add long to half-precision (bottom)	FMLALB (vectors, FP8 to FP16)
FMLALLBB	8-bit floating-point multiply-add long long to single-precision (bottom bottom, indexed)	FMLALLBB (indexed)
	8-bit floating-point multiply-add long long to single-precision (bottom bottom)	FMLALLBB (vectors)
FMLALLBT	8-bit floating-point multiply-add long long to single-precision (bottom top, indexed)	FMLALLBT (indexed)
	8-bit floating-point multiply-add long long to single-precision (bottom top)	FMLALLBT (vectors)
FMLALLTB	8-bit floating-point multiply-add long long to single-precision (top bottom, indexed)	FMLALLTB (indexed)
	8-bit floating-point multiply-add long long to single-precision (top bottom)	FMLALLTB (vectors)
FMLALLTT	8-bit floating-point multiply-add long long to single-precision (top top, indexed)	FMLALLTT (indexed)
	8-bit floating-point multiply-add long long to single-precision (top top)	FMLALLTT (vectors)
FMLALT	8-bit floating-point multiply-add long to half-precision (top, indexed)	FMLALT (indexed, FP8 to FP16)
	8-bit floating-point multiply-add long to half-precision (top)	FMLALT (vectors, FP8 to FP16)

C3.10.15 Floating-point integer binary logarithm

The floating-point integer binary logarithm instruction returns the signed integer base 2 logarithm of each floating-point input element |x| after normalization.

The instruction produces the unbiased exponent of x used in the representation of the floating-point value. For positive x, $x = \text{significand} \times 2^{\text{exponent}}$.

The integer results are placed in elements of the destination vector, which have the same width as the floating-point input elements:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If x is normal, the result is the base 2 logarithm of x.
- If x is subnormal, the result corresponds to the normalized representation.
- If x is infinite, the result is $2^{(esize-1)}-1$.
- If x is ±0.0 or NaN, the result is $-2^{(esize-1)}$.

Inactive elements in the destination vector register are not modified.

Table C3-209 Floating-point integer binary logarithm instruction

Mnemonic	Instruction	See
`FLOGB`	Floating-point base 2 logarithm as integer	FLOGB

C3.10.16 Floating-point minimum/maximum absolute value

Table C3-210 Floating-point absolute minimum/maximum instructions

Mnemonic	Instruction	See
`FAMAX`	Floating-point absolute maximum (predicated)	FAMAX
`FAMIN`	Floating-point absolute minimum (predicated)	FAMIN

C3.10.17 Cross-lane match detect

This section includes instructions that detect or count matching elements within another vector, or within a 128-bit vector segment.

C3.10.17.1 Vector Histogram Count

The vector histogram count instructions create vector histograms.

- `HISTSEG` compares each 8-bit byte element in the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector. The instruction counts the matching elements and places the result in the corresponding destination vector element. The instruction is unpredicated.
- `HISTCNT` compares each active 32-bit or 64-bit element in the first source vector with all elements in the second source vector that have an element number less than or equal to the *Active element* in the first source vector. The number of matching elements is counted and the result is placed in the corresponding destination vector element. *Inactive elements* in the destination vector are set to zero. *Inactive elements* in the second source vector do not cause a match.

Table C3-211 Vector histogram count instructions

Mnemonic	Instruction	See
`HISTCNT`	Count matching elements in vector	HISTCNT
`HISTSEG`	Count matching elements in vector segments	HISTSEG

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.17.2 Character match

The character match instructions can be used to scan each 128-bit segment of the second source vector for an 8-bit or 16-bit character string from the first source vector.

The `MATCH` and `NMATCH` instructions compare each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. When the first source character matches any (`MATCH`) or does not match any (`NMATCH`) character in the second segment, a true value is placed in the corresponding destination predicate element, otherwise a false value is placed in the destination predicate element. *Inactive elements* in the destination predicate register are set to zero.

The instruction sets the NZC condition flags based on the predicate result, and sets the V flag to zero.

Table C3-212 Character match instructions

Mnemonic	Instruction	See
`MATCH`	Detect any matching elements, setting the condition flag	MATCH
`NMATCH`	Detect no matching elements, setting the condition flags	NMATCH

C3.10.17.3 Contiguous conflict detection

The contiguous conflict detection instructions check two addresses for a conflict or overlap between address ranges that could result in a loop-carried dependency through memory. The address range has the form [addr,addr+*VL*÷8], where *VL* is the Effective SVE vector length in bits. A conflict can occur when contiguous load and store instructions use these addresses within the same loop iteration.

The instructions generate a predicate with elements that are true when the addresses cannot conflict within the same iteration, and false thereafter. The instructions set the NZC condition flags based on the predicate result, and the V flag is set to zero.

Table C3-213 Contiguous conflict detection instructions

Mnemonic	Instruction	See
`WHILERW`	While free of read-after-write conflicts	WHILERW
`WHILEWR`	While free of write-after-read/write conflicts	WHILEWR

C3.10.18 Bit permutation

The bit permutation instructions are optional. The bit permutation instructions are configured by the ID_AA64ZFR0_EL1.BitPerm bit. The instructions can be used to scatter, gather, or separate a set of bits within each first source vector element under the control of a bit mask or sieve in the corresponding second source vector elements. The instructions are unpredicated.

The `BDEP` instruction scatters the lowest-numbered contiguous bits within each first source vector element to the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The order of the bits is preserved. The bits corresponding to a zero mask bit are set to zero.

The `BEXT` instruction gathers bits in each first source vector element from the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The bits are gathered to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order. The remaining higher-numbered bits are set to zero.

The `BGRP` instruction selects bits from each first source vector element and groups them into the corresponding destination element, using a corresponding mask element in the second source vector, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The nonzero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the lowest-numbered contiguous bits of the destination element.
- The zero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the highest-numbered contiguous bits of the destination element.

The bit order within each group is preserved.

Table C3-214 Bit permutation instructions

Mnemonic	Instruction	See
BDEP	Scatter lower bits into positions selected by bitmask	BDEP
BEXT	Gather lower bits from positions selected by bitmask	BEXT
BGRP	Group bits to right or left as selected by bitmask	BGRP

C3.10.19 Polynomial arithmetic

The polynomial arithmetic instructions support polynomial arithmetic over [0, 1], where exclusive-OR takes the place of addition. The instructions can be used in applications such as CRC calculations, AES-GCM, elliptic curve cryptography, Diffie-Hellman key exchange, and others.

The PMUL and widening PMLAL, PMULL, PMULLB, and PMULLT instructions perform polynomial multiplication over [0,1]. PMLAL additionally performs a bitwise exclusive-OR of the multiplication result on the element in the destination vector.

The PMULLB and PMULLT instructions read the source operands from either the even-numbered (bottom) or odd-numbered (top) elements, respectively. The widened results are placed in the double-width destination elements that overlap with the corresponding source elements.

The multi-vector PMLAL and PMULL instructions produce results in two destination vectors. The double-width results calculated from even-numbered source elements are placed in the first destination vector, while the double-width results calculated from odd-numbered source elements are placed in the second destination vector.

The EORBT and EORTB instructions operate on the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register. The result is either placed in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged, or placed in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged.

These instructions are unpredicated.

Table C3-215 Polynomial arithmetic instructions

Mnemonic	Instruction	See
EORBT	Interleaving exclusive-OR (bottom, top)	EORBT
EORTB	Interleaving exclusive-OR (top, bottom)	EORTB
PMLAL	Multi-vector polynomial multiply long and accumulate	PMLAL
PMUL	Polynomial multiply vectors (unpredicated)	PMUL
PMULL	Multi-vector polynomial multiply long	PMULL
PMULLB	Polynomial multiply long (bottom)	PMULLB
PMULLT	Polynomial multiply long (top)	PMULLT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.20 Vector permutation

The vector concatenation instructions `EXT` and `SPLICE` have constructive versions that are introduced in SVE2 that preserve both of the source operands. In the constructive versions of the instruction, only the first source vector register number is encoded, which requires the source vectors to be in consecutively numbered registers (modulo 32).

The `EXPAND` instruction copies the lowest numbered source vector elements to active elements of the destination vector, this complements the `COMPACT` instruction.

For more information see Table C3-185.

Table C3-216 Vector permutation instructions

Mnemonic	Instruction	See
`COMPACT`	Copy active vector elements to lower-numbered elements	COMPACT
`EXPAND`	Copy lower numbered vector elements to active elements	EXPAND
`EXT`	Extract vector from pair of vectors	EXT
`SPLICE`	Splice two vectors under predicate control	SPLICE

C3.10.21 Table lookup

C3.10.21.1 Extended table lookup permute

The SVE2 extended table lookup instructions, `TBL` and `TBX` enable the construction of table lookups or programmable vector permutes where the table consists of two or more vector registers.

Because the index values can select any element in a vector, the instructions are not naturally vector length agnostic.

Table C3-217 Extended table lookup instructions

Mnemonic	Instruction	See
`TBL`	Programmable table lookup in one or two vector table (zeroing)	TBL
`TBX`	Programmable table lookup in single vector table (zeroing)	TBX

C3.10.21.2 Lookup table using 2-bit or 4-bit indices

Copy indexed elements from the first source vector(s) to the destination vector, using packed indices from a portion of the second source vector.

Table C3-218 SVE2 lookup table using 2-bit or 4-bit indices instructions

Mnemonic	Instruction	See
`LUTI2`	Lookup table using 2-bit indices	LUTI2
`LUTI4`	Lookup table using 4-bit indices	LUTI4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.10.22 Cryptography support

C3.10.22.1 AES-128 instructions

AES-128 is a 128-bit block cipher that is calculated using a combination of linear EOR operations, the use of rotations by fixed values, and a set of 8-bit non-linear substitutions.

The `AESE`, `AESMC`, and `AESEMC` instructions accelerate a single encryption round, and the `AESD`, `AESIMC`, and `AESDIMC` instructions accelerate a single decryption round. These instructions read a 16-byte state from a 128-bit segment of the first source vectors, and a round key from a 128-bit segment of the second source vector.

The AES instructions are unpredicated.

Table C3-219 AES-128 instructions

Mnemonic	Instruction	See
`AESD`	Multi-vector AES single round decryption	AESD (indexed)
	AES single round decryption	AESD (vectors)
`AESDIMC`	Multi-vector AES single round decryption and inverse mix columns	AESDIMC
`AESE`	Multi-vector AES single round encryption	AESE (indexed)
	AES single round encryption	AESE (vectors)
`AESEMC`	Multi-vector AES single round encryption and mix columns	AESEMC
`AESIMC`	AES inverse mix columns	AESIMC
`AESMC`	AES mix columns	AESMC

C3.10.22.2 SHA-3 instructions

The SHA-3 instructions accelerate the SHA-3 hash algorithm.

The SHA-3 hash is based on a running digest of 1600 bits, arranged as a five by five array of 64-bit values. The instructions map the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is done on these registers during a round of the SHA-3 hash calculation.

Two or more parallel SHA-3 hash calculations are combined as a SIMD operation, where one calculation operates on the 0th 64-bit element of each vector, and the other calculation operates on the first 64-bit element of each vector. The SIMD operation is useful for the fast parallel hash algorithm recently introduced into the SHA-3 standard that allows a single input stream to be calculated using multiple SHA-3 hashes in parallel.

The SHA3 instructions are unpredicated.

Table C3-220 SHA-3 instructions

Mnemonic	Instruction	See
`RAX1`	Bitwise rotate left by 1 and exclusive-OR	RAX1

See also Bitwise ternary logical instructions.

C3.10.22.3 SM4 instructions

SM4 is the standard Chinese symmetric encryption algorithm which can be accelerated using a similar approach to that used for AES.

SM4 is a 128-bit wide block cipher that is calculated using a combination of linear EOR operations, the use of fixed-value rotations, and a set of 8-bit non-linear substitutions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `SM4E` instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, and four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each input data block is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector.
- The `SM4EKEY` instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, and four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector.

The SM4 instructions are unpredicated.

Table C3-221 SM4 instructions

Mnemonic	Instruction	See
`SM4E`	SM4 encryption and decryption	SM4E
`SM4EKEY`	SM4 key updates	SM4EKEY

C3.10.23 BFloat16 arithmetic

C3.10.23.1 Add and subtract

Add or subtract BFloat16 elements of the second source vector from the corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Table C3-222 SVE2 BFloat16 add and subtract instructions

Mnemonic	Instruction	See
`BFADD`	BFloat16 floating-point add vectors (predicated, unpredicated)	BFADD (predicated) BFADD (unpredicated)
`BFSUB`	BFloat16 floating-point subtract vectors (predicated, unpredicated)	BFSUB (predicated) BFSUB (unpredicated)

C3.10.23.2 Multiply and multiply-accumulate

Multiply all BFloat16 elements of the first source vector by the specified element in the corresponding second source vector and place the results in the corresponding elements of the destination vector. The accumulate instructions add or subtract the result without intermediate rounding from the corresponding elements of the destination vector.

`BFMLA` and `BFMLS` perform a fused multiply-add or multiply-subtract of BFloat16 values without intermediate rounding.

Table C3-223 SVE2 BFloat16 multiply and multiply-accumulate instructions

Mnemonic	Instruction	See
`BFMUL`	BFloat16 floating-point multiply vectors by indexed elements	BFMUL (indexed)
	BFloat16 floating-point multiply vectors (predicated, unpredicated)	BFMUL (vectors, predicated) BFMUL (vectors, unpredicated)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BFMLA	BFloat16 floating-point fused multiply-add vectors by indexed elements	BFMLA (indexed)
	BFloat16 floating-point fused multiply-add vectors	BFMLA (vectors)
BFMLS	BFloat16 floating-point fused multiply-subtract vectors by indexed elements	BFMLS (indexed)
	BFloat16 floating-point fused multiply-subtract vectors	BFMLS (vectors)

C3.10.24 BFloat16 minimum/maximum

Determine the maximum or minimum active BFloat16 elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Table C3-224 SVE2 BFloat16 minimum/maximum instructions

Mnemonic	Instruction	See
BFMAX	BFloat16 floating-point maximum (predicated)	BFMAX
BFMAXNM	BFloat16 floating-point maximum number (predicated)	BFMAXNM
BFMIN	BFloat16 floating-point minimum (predicated)	BFMIN
BFMINNM	BFloat16 floating-point minimum number (predicated)	BFMINNM

C3.10.25 Floating-point adjust exponent

The SVE2 BFSCALE instruction performs BFloat16 scaling by integer powers of two.

Table C3-225 SVE2 floating-point adjust exponent instructions

Mnemonic	Instruction	See
BFSCALE	Scales BFloat16 values by integer powers of two	BFSCALE

C3.10.26 Clamp to minimum/maximum

Clamp each element in the destination vectors to between the minimum value in the corresponding element of the first source vector and the maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vector.

C3.10.26.1 Floating-point clamp to minimum/maximum

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-226 SVE2.1 Floating-point clamp to minimum/maximum instructions

Mnemonic	Instruction	See
BFCLAMP	BFloat16 floating-point clamp to minimum/maximum number	BFCLAMP
FCLAMP	Floating-point clamp to minimum/maximum number	FCLAMP

C3.10.26.2 Integer clamp to minimum/maximum

Table C3-227 SVE2.1 Integer clamp to minimum/maximum instructions

Mnemonic	Instruction	See
SCLAMP	Signed clamp to minimum/maximum vector	SCLAMP
UCLAMP	Unsigned clamp to minimum/maximum vector	UCLAMP

C3.10.27 Dot product

C3.10.27.1 Floating-point dot product

Compute the dot product of a pair of values held in the corresponding elements of the first source vectors multiplied by the corresponding elements of the second source vector. The dot product is then destructively added to the corresponding element of the destination vector.

Table C3-228 SVE2.1 Floating-point dot product instructions

Mnemonic	Instruction	See
FDOT	Half-precision floating-point indexed dot product	FDOT (indexed)
	Half-precision floating-point dot product	FDOT (vectors)

C3.10.27.2 FP8 floating-point dot product

The FP8 floating-point dot product instructions accumulate the dot product of two or four adjacent FP8 source elements into half-precision or single-precision destination elements. They generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE. For more information, see the instruction descriptions.

Table C3-229 SVE2 FP8 floating-point dot product instructions

Mnemonic	Instruction	See
FDOT	8-bit floating-point indexed dot product to half-precision	FDOT (2-way, indexed, FP8 to FP16)
	8-bit floating-point dot product to half-precision	FDOT (2-way, vectors, FP8 to FP16)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	8-bit floating-point indexed dot product to single-precision	FDOT (4-way, indexed)
	8-bit floating-point dot product to single-precision	FDOT (4-way, vectors)

C3.10.28 Floating-point matrix multiply-accumulate

C3.10.28.1 Floating-point matrix multiply-accumulate

The `FMMLA` (widening, FP16 to FP32) instruction is implemented by FEAT_SVE_F16F32MM. This instruction delimits each source and destination vector into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of half-precision values in row-by-row order.
- In the second source vector, a 4x2 matrix of half-precision values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

The intermediate single-precision sums-of-products are rounded before they are summed, and the intermediate sum is rounded before addition to the single-precision destination elements.

Table C3-230 SVE2 Floating-point matrix multiply-accumulate instructions

Mnemonic	Instruction	See
`FMMLA`	Half-precision matrix multiply-accumulate to single-precision	FMMLA (widening, half-precision to single-precision)

C3.10.28.2 FP8 floating-point matrix multiply-accumulate

The FP8 `FMMLA` (widening, FP8 to FP32) instruction is implemented by FEAT_F8F32MM. This instruction delimits each source and destination vector into 128-bit segments. The 128-bit segments are organized as follows:

- In the first source vector, a 2x8 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, an 8x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of single-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

The FP8 `FMMLA` (widening, FP8 to FP16) instruction is implemented by FEAT_F8F16MM. This instruction delimits each source and destination vector into 64-bit segments. The 64-bit segments are organized as follows:

- In the first source vector, a 2x4 matrix of 8-bit floating-point values in row-by-row order.
- In the second source vector, a 4x2 matrix of 8-bit floating-point values in column-by-column order.
- In the destination vector, a 2x2 matrix of half-precision values in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment.

These instructions generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding. For these instructions, the intermediate sum of products is downscaled by subtracting the value in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-231 SVE2 FP8 Floating-point matrix multiply-accumulate instructions

Mnemonic	Instruction	See
FMMLA	8-bit floating-point matrix multiply-accumulate to single-precision	FMMLA (widening, FP8 to FP32)
	8-bit floating-point matrix multiply-accumulate to half-precision	FMMLA (widening, FP8 to FP16)

C3.10.29 Floating-point round to integral value

The following instructions round a floating-point value to an integral floating-point value of 32-bit or 64-bit integer size.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-232 SVE2 Floating-point round to integral value instructions

Mnemonic	Instruction	See
FRINT32X	Floating-point round to 32-bit integer, using current rounding mode (predicated)	FRINT32X
FRINT32Z	Floating-point round to 32-bit integer, towards zero (predicated)	FRINT32Z
FRINT64X	Floating-point round to 64-bit integer, using current rounding mode (predicated)	FRINT64X
FRINT64Z	Floating-point round to 64-bit integer, towards zero (predicated)	FRINT64Z

C3.10.30 Quadword operations

C3.10.30.1 Permute within quadwords

In these instructions the same permutation pattern is applied within each quadword.

Table C3-233 SVE2.1 Permute within quadwords instructions

Mnemonic	Instruction	See
DUPQ	Broadcast indexed element within each quadword vector segment (unpredicated)	DUPQ
EXTQ	Extract vector segment from each pair of quadword vector segments	EXTQ
TBLQ	Programmable table lookup within each quadword vector segment (zeroing)	TBLQ
TBXQ	Programmable table lookup within each quadword vector segment (merging)	TBXQ
UZPQ1	Concatenate even elements within each pair of quadword vector segments	UZPQ1
UZPQ2	Concatenate odd elements within each pair of quadword vector segments	UZPQ2
ZIPQ1	Interleave elements from low halves of each pair of quadword vector segments	ZIPQ1
ZIPQ2	Interleave elements from high halves of each pair of quadword vector segments	ZIPQ2

C3.10.30.2 Reduction to quadword

In the reductions to quadwords instructions a full SVE vector is reduced to a single 128-bit vector. The result of the operations is placed into the corresponding element of the 128-bit Advanced SIMD and floating-point destination register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-234 SVE2.1 Reduction to quadword instructions

Mnemonic	Instruction	See
ADDQV	Unsigned add reduction of quadword vector segments	ADDQV
ANDQV	Bitwise AND reduction of quadword vector segments	ANDQV
EORQV	Bitwise exclusive OR reduction of quadword vector segments	EORQV
FADDQV	Floating-point add recursive reduction of quadword vector segments	FADDQV
FMAXNMQV	Floating-point maximum number recursive reduction of quadword vector segments	FMAXNMQV
FMAXQV	Floating-point maximum reduction of quadword vector segments	FMAXQV
FMINNMQV	Floating-point minimum number recursive reduction of quadword vector segments	FMINNMQV
FMINQV	Floating-point minimum recursive reduction of quadword vector segments	FMINQV
ORQV	Bitwise inclusive OR reduction of quadword vector segments	ORQV
SMAXQV	Signed maximum reduction of quadword vector segments	SMAXQV
SMINQV	Signed minimum reduction of quadword vector segments	SMINQV
UMAXQV	Unsigned maximum reduction of quadword vector segments	UMAXQV
UMINQV	Unsigned minimum reduction of quadword vector segments	UMINQV

C3.10.31 Integer shift and convert

Convert or shift the value of each element of the source vectors and place the results in the corresponding destination vector elements.

These are narrowing, interleaving, saturating variants of the integer shift and convert instructions, to avoid write after write dependencies for merging narrowing top instructions.

Table C3-235 SVE2.1 Integer shift and convert instructions

Mnemonic	Instruction	See
SQCVTN	Signed saturating extract narrow and interleave	SQCVTN
SQCVTUN	Signed saturating unsigned extract narrow and interleave	SQCVTUN
SQRSHRN	Signed saturating rounding shift right narrow by immediate and interleave	SQRSHRN
SQRSHRUN	Signed saturating rounding shift right unsigned narrow by immediate and interleave	SQRSHRUN
UQCVTN	Unsigned saturating extract narrow and interleave	UQCVTN
UQRSHRN	Unsigned saturating rounding shift right narrow by immediate and interleave	UQRSHRN

C3.10.32 Predicate element index

Determine the index of the first or last active TRUE predicate element.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-236 SVE2 Scalar index instructions

Mnemonic	Instruction	See
FIRSTP	Scalar index of first TRUE predicate element (predicated)	FIRSTP
LASTP	Scalar index of last TRUE predicate element (predicated)	LASTP

C3.10.33 Multi-vector predication

Multi-vector predication is required when using the Contiguous multi-vector loads and stores, and also reduces the number of predicate registers and predicate calculations in a loop.

The WHILE instructions generate a predicate-as-counter encoding. These instructions have an operand that indicates the number of vectors (2 or 4) to be controlled by this predicate, which determines:

- The maximum value that can be stored in the count.
- The number of elements that are considered *Active* when calculating the *Any Active element* and *Last Active element* SVE condition flags.

CNTP has an operand that indicates the limit of the number of elements to be counted. The limit corresponds to the total number of elements in either 2 or 4 vectors.

For more information, see Predicate-as-counter.

Table C3-237 SVE2.1 Multi-vector predication instructions

Mnemonic	Instruction	See
CNTP	Set scalar to count from predicate-as-counter	CNTP (predicate as counter)
PEXT	Set pair of predicates from predicate-as-counter	PEXT (predicate pair)
	Set predicate from predicate-as-counter	PEXT (predicate)
PTRUE	Initialize predicate-as-counter to all active	PTRUE (predicate as counter)
WHILEGE	While decrementing signed scalar greater than or equal to scalar (predicate-as-counter)	WHILEGE (predicate as counter)
WHILEGT	While decrementing signed scalar greater than scalar (predicate-as-counter)	WHILEGT (predicate as counter)
WHILEHI	While decrementing unsigned scalar higher than scalar (predicate-as-counter)	WHILEHI (predicate as counter)
WHILEHS	While decrementing unsigned scalar higher or same as scalar (predicate-as-counter)	WHILEHS (predicate as counter)
WHILELE	While incrementing signed scalar less than or equal to scalar (predicate-as-counter)	WHILELE (predicate as counter)
WHILELO	While incrementing unsigned scalar lower than scalar (predicate-as-counter)	WHILELO (predicate as counter)
WHILELS	While incrementing unsigned scalar lower or same as scalar (predicate-as-counter)	WHILELS (predicate as counter)
WHILELT	While incrementing signed scalar less than scalar (predicate-as-counter)	WHILELT (predicate as counter)

C3.10.34 Permute vector

This instruction is a fixed permute instruction, which is a doubleword variant of the reverse within elements instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-238 SVE2.1 Permute vector instruction

Mnemonic	Instruction	See
REVD	Reverse 64-bit doublewords in elements (predicated)	REVD

C3.10.35 Predicate pair loop control

Generate a pair of predicate registers to simplify loop unrolling.

Table C3-239 SVE2.1 Predicate pair loop control instructions

Mnemonic	Instruction	See
WHILEGE	While decrementing signed scalar greater than or equal to scalar (pair of predicates)	WHILEGE (predicate pair)
WHILEGT	While decrementing signed scalar greater than scalar (pair of predicates)	WHILEGT (predicate pair)
WHILEHI	While decrementing unsigned scalar higher than scalar (pair of predicates)	WHILEHI (predicate pair)
WHILEHS	While decrementing unsigned scalar higher or same as scalar (pair of predicates)	WHILEHS (predicate pair)
WHILELE	While incrementing signed scalar less than or equal to scalar (pair of predicates)	WHILELE (predicate pair)
WHILELO	While incrementing unsigned scalar lower than scalar (pair of predicates)	WHILELO (predicate pair)
WHILELS	While incrementing unsigned scalar lower or same as scalar (pair of predicates)	WHILELS (predicate pair)
WHILELT	While incrementing signed scalar less than scalar (pair of predicates)	WHILELT (predicate pair)

C3.10.36 Predicate move

With the PMOV (to predicate) instruction, based on the specified element size and immediate index, the selected bits in a vector register are expanded into a predicate register ready to be used as an SVE governing predicate.

With the PMOV (to vector) instruction a portion of a bitarray is created and can be stored to memory or further processed in registers.

Table C3-240 SVE2.1 Predicate move instructions

Mnemonic	Instruction	See
PMOV	Move predicate from vector	PMOV (to predicate)
	Move predicate to vector	PMOV (to vector)

C3.10.37 Predicate select

PSEL places contents of the first source predicate register into a destination register or sets the destination to all-FALSE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-241 SVE2.1 Predicate select instruction

Mnemonic	Instruction	See
`PSEL`	Predicate select between predicate register or all-false	PSEL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11 Data processing - SME, SME2

The following subsections describe the SME, SME2, and SME2.1 processing instructions:

- Clamp to minimum/maximum.
- Dot product.
- Vertical dot product.
- Element concatenate and interleave.
- Floating-point conversions.
- Floating-point round.
- Floating-point adjust exponent.
- Lookup table.
- Move operations.
- Multi-vector arithmetic.
- Multi-vector minimum/maximum.
- Multi-vector multiply.
- Multi-vector multiply-accumulate.
- Multi-vector multiply high.
- Multi-vector select.
- Multi-vector shift and convert.
- Outer product.
- Quarter-tile outer product.
- Structured sparsity outer product.
- Stack frame operations.
- Unpack and extend.
- Zero operations.

C3.11.1 Clamp to minimum/maximum

Clamp each element in the destination vectors to between the minimum value in the corresponding element of the first source vector and the maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vectors.

C3.11.1.1 Floating-point clamp to minimum/maximum

Table C3-242 SME2 Floating-point clamp to minimum/maximum instructions

Mnemonic	Instruction	See
`BFCLAMP`	Multi-vector BFloat16 floating-point clamp to minimum/maximum number	BFCLAMP
`FCLAMP`	Multi-vector floating-point clamp to minimum/maximum number	FCLAMP

C3.11.1.2 Integer clamp to minimum/maximum

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-243 SME2 Integer clamp to minimum/maximum instructions

Mnemonic	Instruction	See
SCLAMP	Multi-vector signed clamp to minimum/maximum vector	SCLAMP
UCLAMP	Multi-vector unsigned clamp to minimum/maximum vector	UCLAMP

C3.11.2 Floating-point dot product

Compute the dot product of two or four values held in the corresponding elements of the first source vectors and in the corresponding element of the second source vector. The widened dot product is then added to the corresponding element of the ZA single-vector groups.

C3.11.2.1 BFloat16 floating-point dot product

Table C3-244 BFloat16 floating-point dot product instructions

Mnemonic	Instruction	See
BFDOT	Multi-vector BFloat16 floating-point dot-product by indexed element	BFDOT (multiple and indexed vector)
	Multi-vector BFloat16 floating-point dot-product by vector	BFDOT (multiple and single vector)
	Multi-vector BFloat16 floating-point dot-product	BFDOT (multiple vectors)

C3.11.2.2 Floating-point dot product

Table C3-245 SME2 Floating-point Dot product instructions

Mnemonic	Instruction	See
FDOT	Multi-vector half-precision floating-point dot-product by indexed element	FDOT (multiple and indexed vector)
	Multi-vector half-precision floating-point dot-product by vector	FDOT (multiple and single vector)
	Multi-vector half-precision floating-point dot-product	FDOT (multiple vectors)

C3.11.2.3 FP8 floating-point dot product

The FP8 floating-point dot product instructions accumulate the dot product of two or four FP8 source elements into half-precision or single-precision destination elements. They generate a fused, downscaled sum-of-products and accumulate into each destination element, without intermediate rounding.

For these instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-246 SME2 FP8 floating-point convert instructions

Mnemonic	Instruction	See
FDOT	Multi-vector 8-bit floating-point dot-product by indexed element to half-precision	FDOT (2-way, multiple and indexed vector, FP8 to FP16)
	Multi-vector 8-bit floating-point dot-product by vector to half-precision	FDOT (2-way, multiple and single vector, FP8 to FP16)
	Multi-vector 8-bit floating-point dot-product to half-precision	FDOT (2-way, multiple vectors, FP8 to FP16)
	Multi-vector 8-bit floating-point dot-product by indexed element to single-precision	FDOT (4-way, multiple and indexed vector)
	Multi-vector 8-bit floating-point dot-product by vector to single-precision	FDOT (4-way, multiple and single vector)
	Multi-vector 8-bit floating-point dot-product to single-precision	FDOT (4-way, multiple vectors)

C3.11.2.4 Integer dot product

Table C3-247 SME2 Integer Dot product instructions

Mnemonic	Instruction	See
SDOT	Multi-vector signed integer dot-product by indexed element	SDOT (2-way, multiple and indexed vector) SDOT (4-way, multiple and indexed vector)
	Multi-vector signed integer dot-product by vector	SDOT (2-way, multiple and single vector) SDOT (4-way, multiple and single vector)
	Multi-vector signed integer dot-product	SDOT (2-way, multiple vectors) SDOT (4-way, multiple vectors)
SUDOT	Multi-vector signed by unsigned integer dot-product by indexed elements	SUDOT (multiple and indexed vector)
	Multi-vector signed by unsigned integer dot-product by vector	SUDOT (multiple and single vector)
UDOT	Multi-vector unsigned integer dot-product by indexed element	UDOT (2-way, multiple and indexed vector) UDOT (4-way, multiple and indexed vector)
	Multi-vector unsigned integer dot-product by vector	UDOT (2-way, multiple and single vector) UDOT (4-way, multiple and single vector)
	Multi-vector unsigned integer dot-product	UDOT (2-way, multiple vectors) UDOT (4-way, multiple vectors)
USDOT	Multi-vector unsigned by signed integer dot-product by indexed element	USDOT (multiple and indexed vector)
	Multi-vector unsigned by signed integer dot-product by vector	USDOT (multiple and single vector)
	Multi-vector unsigned by signed integer dot-product	USDOT (multiple vectors)

C3.11.3 Vertical dot product

Compute the vertical dot product of the corresponding values held in the first source vectors and in the corresponding elements of second source vector. The widened dot product is then added to the corresponding element of the ZA single-vector groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.3.1 BFloat16 vertical dot product

Table C3-248 SME2 BFloat16 floating-point vertical dot product instructions

Mnemonic	Instruction	See
BFVDOT	Multi-vector BFloat16 floating-point vertical dot-product by indexed element	BFVDOT

C3.11.3.2 Floating-point vertical dot product

Table C3-249 SME2 Floating-point vertical dot product instructions

Mnemonic	Instruction	See
FVDOT	Multi-vector half-precision floating-point vertical dot-product by indexed element	FVDOT (FP16 to FP32)

C3.11.3.3 FP8 floating-point vertical dot product

For FP8 vertical dot product instructions, the unbounded intermediate sum-of-products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-250 SME2 FP8 floating-point vertical dot product instructions

Mnemonic	Instruction	See
FVDOT	Multi-vector 8-bit floating-point vertical dot-product by indexed element to half-precision	FVDOT (FP8 to FP16)
FVDOTB	Multi-vector 8-bit floating-point vertical dot-product by indexed element to single-precision (bottom)	FVDOTB
FVDOTT	Multi-vector 8-bit floating-point vertical dot-product by indexed element to single-precision (top)	FVDOTT

C3.11.3.4 Integer vertical dot product

Table C3-251 SME2 Integer vertical dot product instructions

Mnemonic	Instruction	See
SUVDOT	Multi-vector signed by unsigned integer vertical dot-product by indexed element	SUVDOT
SVDOT	Multi-vector signed integer vertical dot-product by indexed element	SVDOT (2-way) SVDOT (4-way)
USVDOT	Multi-vector unsigned by signed integer vertical dot-product by indexed element	USVDOT
UVDOT	Multi-vector unsigned integer vertical dot-product by indexed element	UVDOT (2-way) UVDOT (4-way)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.4 Element concatenate and interleave

Join elements from the source vectors and place them in the corresponding elements of the destination vectors.

Table C3-252 SME2 Element concatenate/interleave instructions

Mnemonic	Instruction	See
`UZP`	Concatenate elements from four/two vectors	UZP (four registers) UZP (two registers)
`ZIP`	Interleave elements from four/two vectors	ZIP (four registers) ZIP (two registers)

C3.11.5 Floating-point conversions

Convert each element of the source vectors and place the results in the corresponding elements of the destination vectors.

C3.11.5.1 Floating-point conversions

Table C3-253 SME2 Floating-point conversion instructions

Mnemonic	Instruction	See
`FCVT`	Multi-vector floating-point convert from single-precision to packed half-precision	FCVT (narrowing)
	Multi-vector floating-point convert from half-precision to single-precision (in-order)	FCVT (widening)
`FCVTL`	Multi-vector floating-point convert from half-precision to deinterleaved single-precision	FCVTL
`FCVTN`	Multi-vector floating-point convert from single-precision to interleaved half-precision	FCVTN
`FCVTZS`	Multi-vector floating-point convert to signed integer, rounding toward zero	FCVTZS
`FCVTZU`	Multi-vector floating-point convert to unsigned integer, rounding toward zero	FCVTZU

C3.11.5.2 BFloat16 floating-point conversions

Table C3-254 BFloat16 Floating-point conversion instructions

Mnemonic	Instruction	See
`BFCVT`	Multi-vector floating-point convert from single-precision to packed BFloat16 format	BFCVT
`BFCVTN`	Multi-vector floating-point convert from single-precision to interleaved BFloat16 format	BFCVTN

C3.11.5.3 FP8 floating-point conversions

Convert each floating-point element of the source vectors while scaling the value, and place the results in the corresponding elements of the destination vector.

For instructions that convert other floating-point formats to FP8 formats, each input element is scaled by adding the value specified in FPMR.NSCALE to its unbounded exponent during the conversion.

For instructions that convert FP8 formats to other floating-point formats, each input element is scaled by subtracting the value specified in FPMR.LSCALE or FPMR.LSCALE2 from its unbounded exponent during the conversion.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see the instruction descriptions.

Table C3-255 FP8 floating-point conversion instructions

Mnemonic	Instruction	See
`BF1CVT`, `BF2CVT`	Multi-vector floating-point convert from 8-bit floating-point to BFloat16 (in-order)	BF1CVT, BF2CVT
`BF1CVTL`, `BF2CVTL`	Multi-vector floating-point convert from 8-bit floating-point to deinterleaved BFloat16	BF1CVTL, BF2CVTL
`BFCVT`	Multi-vector floating-point convert from BFloat16 to packed 8-bit floating-point format	BFCVT
`F1CVT`, `F2CVT`	Multi-vector floating-point convert from 8-bit floating-point to half-precision (in-order)	F1CVT, F2CVT
`F1CVTL`, `F2CVTL`	Multi-vector floating-point convert from 8-bit floating-point to deinterleaved half-precision	F1CVTL, F2CVTL
`FCVT`	Multi-vector floating-point convert from half-precision to packed 8-bit floating-point format	FCVT (narrowing, FP16 to FP8)
	Multi-vector floating-point convert from single-precision to packed 8-bit floating-point format	FCVT (narrowing, FP32 to FP8)
`FCVTN`	Multi-vector floating-point convert from single-precision to interleaved 8-bit floating-point format	FCVTN (FP32 to FP8)

C3.11.6 Floating-point round

Round each element of the source vectors to an integral floating-point value and place the results in the corresponding elements of the destination vectors.

Table C3-256 SME2 Floating-point round to integral value instructions

Mnemonic	Instruction	See
`FRINTA`	Multi-vector floating-point round to integral value, to nearest with ties away from zero	FRINTA
`FRINTM`	Multi-vector floating-point round to integral value, toward minus Infinity	FRINTM
`FRINTN`	Multi-vector floating-point round to integral value, to nearest with ties to even	FRINTN
`FRINTP`	Multi-vector floating-point round to integral value, toward plus Infinity	FRINTP

C3.11.7 Floating-point adjust exponent

C3.11.7.1 Floating-point scaling

The `FSCALE` instructions perform floating-point scaling by integer powers of two.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-257 SME2 floating-point scaling instructions

Mnemonic	Instruction	See
FSCALE	Multi-vector floating-point adjust exponent	FSCALE (multiple and single vector)
	Multi-vector floating-point adjust exponent	FSCALE (multiple vectors)

C3.11.7.2 BFloat16 scaling

The BFSCALE instructions perform BFloat16 scaling by integer powers of two.

Table C3-258 SME2 BFloat16 scaling instructions

Mnemonic	Instruction	See
BFSCALE	Multi-vector BFloat16 adjust exponent	BFSCALE (multiple and single vector)
	Multi-vector BFloat16 adjust exponent	BFSCALE (multiple vectors)

C3.11.8 Lookup table

Copy elements from ZT0 to the destination vectors using packed indices from a portion of the source vector.

Table C3-259 SME2/SME2.1 Lookup table read instructions

Mnemonic	Instruction	See
LUTI2	Lookup table read using 2-bit indexes	LUTI2 (four registers)
		LUTI2 (single)
		LUTI2 (two registers)
LUTI4	Lookup table read using 4-bit indexes	LUTI4 (four registers, 16-bit & 32-bit)
		LUTI4 (single)
		LUTI4 (two registers)
		LUTI4 (four registers, 8-bit)

C3.11.9 Move operations

Move between ZA single-vector groups, vector registers, or individual horizontal or vertical ZA tile slices.

Table C3-260 SME/SME2 Move instructions

Mnemonic	Instruction	See
MOVA	Move four ZA single-vector groups to four vector registers	MOVA (array to vector, four registers)
	Move two ZA single-vector groups to two vector registers	MOVA (array to vector, two registers)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Move four ZA tile slices to four vector registers	MOVA (tile to vector, four registers)
	Move ZA tile slice to vector register	MOVA (tile to vector, single)
	Move two ZA tile slices to two vector registers	MOVA (tile to vector, two registers)
	Move four vector registers to four ZA single-vector groups	MOVA (vector to array, four registers)
	Move two vector registers to two ZA single-vector groups	MOVA (vector to array, two registers)
	Move four vector registers to four ZA tile slices	MOVA (vector to tile, four registers)
	Move vector register to ZA tile slice	MOVA (vector to tile, single)
	Move two vector registers to two ZA tile slices	MOVA (vector to tile, two registers)

C3.11.9.1 Lookup table move

Move 8 bytes between the ZT0 register and a general-purpose register at a specified byte offset.

`MOVT` (vector to table) copies the source vector register to ZT0 at a specified vector length offset.

Table C3-261 SME2 Lookup table move instructions

Mnemonic	Instruction	See
`MOVT`	Move 8 bytes from general-purpose register to ZT0	MOVT (scalar to table)
	Move 8 bytes from ZT0 to general-purpose register	MOVT (table to scalar)
	Move vector register to ZT0	MOVT (vector to table)

C3.11.9.2 Move and zero

Move contents of ZA single-vector groups or of consecutive horizontal or vertical slices within a ZA tile to the destination vectors and zero the source contents.

Table C3-262 SME2.1 Move and zero instructions

Mnemonic	Instruction	See
`MOVAZ`	Move and zero four ZA single-vector groups to vector registers	MOVAZ (array to vector, four registers)
	Move and zero two ZA single-vector groups to vector registers	MOVAZ (array to vector, two registers)
	Move and zero four ZA tile slices to vector registers	MOVAZ (tile to vector, four registers)
	Move and zero ZA tile slice to vector register	MOVAZ (tile to vector, single)
	Move and zero two ZA tile slices to vector registers	MOVAZ (tile to vector, two registers)

C3.11.10 Multi-vector arithmetic

C3.11.10.1 Add and subtract

Add or subtract vectors from ZA single-vector groups or destructively place results in corresponding elements of the source vectors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-263 SME2 Add and subtract instructions

Mnemonic	Instruction	See
ADD	Add multi-vector to ZA array vector accumulators	ADD (array accumulators)
	Add replicated single vector to multi-vector with ZA array vector results	ADD (array results, multiple and single vector)
	Add multi-vector to multi-vector with ZA array vector results	ADD (array results, multiple vectors)
	Add replicated single vector to multi-vector with multi-vector result	ADD (to vector)
SUB	Subtract multi-vector from ZA array vector accumulators	SUB (array accumulators)
	Subtract replicated single vector from multi-vector with ZA array vector results	SUB (array results, multiple and single vector)
	Subtract multi-vector from multi-vector with ZA array vector results	SUB (array results, multiple vectors)

C3.11.10.2 Add vector to array

Add each element of the source vector to the corresponding Active element of each vertical or horizontal slice of a ZA tile.

Table C3-264 SME Add vector to array instructions

Mnemonic	Instruction	See
ADDHA	Add horizontally vector elements to ZA tile	ADDHA
ADDVA	Add vertically vector elements to ZA tile	ADDVA

C3.11.10.3 Floating-point add and subtract

Destructively add or subtract all elements of the source vectors from the corresponding elements of the ZA single-vector groups.

Table C3-265 SME2 Floating-point add and subtract instructions

Mnemonic	Instruction	See
BFADD	BFloat16 floating-point add multi-vector to ZA array vector accumulators	BFADD
BFSUB	BFloat16 floating-point subtract multi-vector from ZA array vector accumulators	BFSUB
FADD	Floating-point add multi-vector to ZA array vector accumulators	FADD
FSUB	Floating-point subtract multi-vector from ZA array vector accumulators	FSUB

C3.11.11 Multi-vector minimum/maximum

Determine the minimum or maximum of elements of the second source vector and the corresponding elements of the first source vectors and destructively place the results in the corresponding elements of the first source vectors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.11.1 BFloat16 minimum/maximum

These instructions follow the SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors. If FEAT_SVE_B16B16 is implemented, these instructions are also implemented.

Table C3-266 SME2.1 BFloat16 minimum/maximum

Mnemonic	Instruction	See
BFMAX	Multi-vector BFloat16 floating-point maximum by vector	BFMAX (multiple and single vector)
	Multi-vector BFloat16 floating-point maximum	BFMAX (multiple vectors)
BFMAXNM	Multi-vector BFloat16 floating-point maximum number by vector	BFMAXNM (multiple and single vector)
	Multi-vector BFloat16 floating-point maximum number	BFMAXNM (multiple vectors)
BFMIN	Multi-vector BFloat16 floating-point minimum by vector	BFMIN (multiple and single vector)
	Multi-vector BFloat16 floating-point minimum	BFMIN (multiple vectors)
BFMINNM	Multi-vector BFloat16 floating-point minimum number by vector	BFMINNM (multiple and single vector)
	Multi-vector BFloat16 floating-point minimum number	BFMINNM (multiple vectors)

C3.11.11.2 Floating-point minimum/maximum

Table C3-267 SME2 Floating-point minimum/maximum

Mnemonic	Instruction	See
FMAX	Multi-vector floating-point maximum by vector	FMAX (multiple and single vector)
	Multi-vector floating-point maximum	FMAX (multiple vectors)
FMAXNM	Multi-vector floating-point maximum number by vector	FMAXNM (multiple and single vector)
	Multi-vector floating-point maximum number	FMAXNM (multiple vectors)
FMIN	Multi-vector floating-point minimum by vector	FMIN (multiple and single vector)
	Multi-vector floating-point minimum	FMIN (multiple vectors)
FMINNM	Multi-vector floating-point minimum number by vector	FMINNM (multiple and single vectors)
	Multi-vector floating-point minimum number	FMINNM (multiple vectors)

C3.11.11.3 Floating-point minimum/maximum absolute value

Table C3-268 SME2 Floating-point absolute minimum/maximum

Mnemonic	Instruction	See
FAMAX	Multi-vector floating-point absolute maximum	FAMAX
FAMIN	Multi-vector floating-point absolute minimum	FAMIN

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.11.4 Integer minimum/maximum

Table C3-269 SME2 Integer minimum/maximum

Mnemonic	Instruction	See
SMAX	Multi-vector signed maximum by vector	SMAX (multiple and single vector)
	Multi-vector signed maximum	SMAX (multiple vectors)
SMIN	Multi-vector signed minimum by vector	SMIN (multiple and single vector)
	Multi-vector signed minimum	SMIN (multiple vectors)
UMAX	Multi-vector unsigned maximum by vector	UMAX (multiple and single vector)
	Multi-vector unsigned maximum	UMAX (multiple vectors)
UMIN	Multi-vector unsigned minimum by vector	UMIN (multiple and single vector)
	Multi-vector unsigned minimum	UMIN (multiple vectors)

C3.11.12 Multi-vector multiply

Multiply the floating-point elements of the first source vectors with the corresponding elements of the second source vectors and place the results in the corresponding elements of the destination vectors.

C3.11.12.1 BFloat16 multi-vector multiply

Table C3-270 SME2.2 BFloat16 multi-vector multiply instructions

Mnemonic	Instruction	See
BFMUL	Multi-vector BFloat16 multiply by vector	BFMUL (multiple and single vector)
	Multi-vector BFloat16 multiply	BFMUL (multiple vectors)

C3.11.12.2 Floating-point multi-vector multiply

Table C3-271 SME2.2 Floating-point multi-vector multiply instructions

Mnemonic	Instruction	See
FMUL	Multi-vector floating-point multiply by vector	FMUL (multiple and single vector)
	Multi-vector floating-point multiply	FMUL (multiple vectors)

C3.11.13 Multi-vector multiply-accumulate

Multiply the corresponding elements of the first source vectors by the corresponding elements of the second source vector and destructively, without intermediate rounding, add to or subtract from the corresponding elements of the ZA single-vector groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.13.1 BFloat16 multiply-accumulate

Table C3-272 SME2 BFloat16 multiply-accumulate instructions

Mnemonic	Instruction	See
BFMLA	Multi-vector BFloat16 floating-point fused multiply-add by indexed element	BFMLA (multiple and indexed vector)
	Multi-vector BFloat16 floating-point fused multiply-add by vector	BFMLA (multiple and single vector)
	Multi-vector BFloat16 floating-point fused multiply-add	BFMLA (multiple vectors)
BFMLAL	Multi-vector BFloat16 floating-point multiply-add long by indexed element	BFMLAL (multiple and indexed vector)
	Multi-vector BFloat16 floating-point multiply-add long by vector	BFMLAL (multiple and single vector)
	Multi-vector BFloat16 floating-point multiply-add long	BFMLAL (multiple vectors)
BFMLS	Multi-vector BFloat16 floating-point fused multiply-subtract by indexed element	BFMLS (multiple and indexed vector)
	Multi-vector BFloat16 floating-point fused multiply-subtract by vector	BFMLS (multiple and single vector)
	Multi-vector BFloat16 floating-point fused multiply-subtract	BFMLS (multiple vectors)
BFMLSL	Multi-vector BFloat16 floating-point multiply-subtract long by indexed element	BFMLSL (multiple and indexed vector)
	Multi-vector BFloat16 floating-point multiply-subtract long by vector	BFMLSL (multiple and single vector)
	Multi-vector BFloat16 floating-point multiply-subtract long	BFMLSL (multiple vectors)

C3.11.13.2 Floating-point multiply-accumulate

Table C3-273 SME2 Floating-point multi-vector multiply-accumulate instructions

Mnemonic	Instruction	See
FMLA	Multi-vector floating-point fused multiply-add by indexed element	FMLA (multiple and indexed vector)
	Multi-vector floating-point fused multiply-add by vector	FMLA (multiple and single vector)
	Multi-vector floating-point fused multiply-add	FMLA (multiple vectors)
FMLAL	Multi-vector floating-point multiply-add long by indexed element	FMLAL (multiple and indexed vector)
	Multi-vector floating-point multiply-add long by vector	FMLAL (multiple and single vector)
	Multi-vector floating-point fused multiply-add long	FMLAL (multiple vectors)
FMLS	Multi-vector floating-point fused multiply-subtract by indexed element	FMLS (multiple and indexed vector)
	Multi-vector floating-point fused multiply-subtract by vector	FMLS (multiple and single vector)
	Multi-vector floating-point fused multiply-subtract	FMLS (multiple vectors)
FMLSL	Multi-vector floating-point multiply-subtract long by indexed element	FMLSL (multiple and indexed vector)
	Multi-vector floating-point multiply-subtract long by vector	FMLSL (multiple and single vector)
	Multi-vector floating-point multiply-subtract long	FMLSL (multiple vectors)

C3.11.13.3 FP8 floating-point widening multiply-accumulate

The FP8 floating-point multiply-accumulate instructions widen the 8-bit floating-point elements in the source vectors to half-precision or single-precision format and multiply the corresponding elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

They generate a fused, downscaled product and accumulate into the corresponding elements of the destination vector, without intermediate rounding.

For these instructions, the unbounded intermediate product is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Table C3-274 SME2 FP8 floating-point widening multiply-accumulate instructions

Mnemonic	Instruction	See
FMLAL	Multi-vector 8-bit floating-point multiply-add long by indexed element to half-precision	FMLAL (multiple and indexed vector, FP8 to FP16)
	Multi-vector 8-bit floating-point multiply-add long by vector to half-precision	FMLAL (multiple and single vector, FP8 to FP16)
	Multi-vector 8-bit floating-point multiply-add long to half-precision	FMLAL (multiple vectors, FP8 to FP16)
FMLALL	Multi-vector 8-bit floating-point multiply-add long long by indexed element to single-precision	FMLALL (multiple and indexed vector)
	Multi-vector 8-bit floating-point multiply-add long long by vector to single-precision	FMLALL (multiple and single vector)
	Multi-vector 8-bit floating-point multiply-add long long to single-precision	FMLALL (multiple vectors)

C3.11.13.4 Integer multiply-accumulate

Table C3-275 SME2 Integer multi-vector multiply-accumulate instructions

Mnemonic	Instruction	See
SMLAL	Multi-vector signed integer multiply-add long by indexed element	SMLAL (multiple and indexed vector)
	Multi-vector signed integer multiply-add long by vector	SMLAL (multiple and single vector)
	Multi-vector signed integer multiply-add long	SMLAL (multiple vectors)
SMLALL	Multi-vector signed integer multiply-add long long by indexed element	SMLALL (multiple and indexed vector)
	Multi-vector signed integer multiply-add long long by vector	SMLALL (multiple and single vector)
	Multi-vector signed integer multiply-add long long	SMLALL (multiple vectors)
SMLSL	Multi-vector signed integer multiply-subtract long by indexed element	SMLSL (multiple and indexed vector)
	Multi-vector signed integer multiply-subtract long by vector	SMLSL (multiple and single vector)
	Multi-vector signed integer multiply-subtract long	SMLSL (multiple vectors)
SMLSLL	Multi-vector signed integer multiply-subtract long long by indexed element	SMLSLL (multiple and indexed vector)
	Multi-vector signed integer multiply-subtract long long by vector	SMLSLL (multiple and single vector)
	Multi-vector signed integer multiply-subtract long long	SMLSLL (multiple vectors)
SUMLALL	Multi-vector signed by unsigned integer multiply-add long long by indexed element	SUMLALL (multiple and indexed vector)
	Multi-vector signed by unsigned integer multiply-add long long by vector	SUMLALL (multiple and single vector)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
UMLAL	Multi-vector unsigned integer multiply-add long by indexed element	UMLAL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-add long by vector	UMLAL (multiple and single vector)
	Multi-vector unsigned integer multiply-add long	UMLAL (multiple vectors)
UMLALL	Multi-vector unsigned integer multiply-add long long by indexed element	UMLALL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-add long long by vector	UMLALL (multiple and single vector)
	Multi-vector unsigned integer multiply-add long long	UMLALL (multiple vectors)
UMLSL	Multi-vector unsigned integer multiply-subtract long by indexed element	UMLSL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-subtract long by vector	UMLSL (multiple and single vector)
	Multi-vector unsigned integer multiply-subtract long by indexed element	UMLSL (multiple vectors)
UMLSLL	Multi-vector unsigned integer multiply-subtract long long by indexed element	UMLSLL (multiple and indexed vector)
	Multi-vector unsigned integer multiply-subtract long long by vector	UMLSLL (multiple and single vector)
	Multi-vector unsigned integer multiply-subtract long long	UMLSLL (multiple vectors)
USMLALL	Multi-vector unsigned by signed integer multiply-add long long by indexed element	USMLALL (multiple and indexed vector)
	Multi-vector unsigned by signed integer multiply-add long long by vector	USMLALL (multiple and single vector)
	Multi-vector unsigned by signed integer multiply-add long long	USMLALL (multiple vectors)

C3.11.14 Multi-vector multiply high

These instructions multiply then double the corresponding elements of the first and second source vectors and destructively place the most significant half of the result in the corresponding elements of the first and second source vectors.

Table C3-276 SME2 Multi-vector multiply high

Mnemonic	Instruction	See
SQDMULH	Multi-vector signed saturating doubling multiply high by vector	SQDMULH (multiple and single vector)
	Multi-vector signed saturating doubling multiply high	SQDMULH (multiple vectors)

C3.11.15 Multi-vector select

Copy into elements of the destination vectors the corresponding elements from the first source vectors where the corresponding predicate elements are TRUE, and from the second source vectors where the corresponding predicate elements are FALSE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-277 SME2 Multi-vector select

Mnemonic	Instruction	See
SEL	Multi-vector conditionally select elements from two vectors	SEL

C3.11.16 Multi-vector shift and convert

Convert or shift the value of each element of the source vectors and place the results in the corresponding destination vector elements.

Table C3-278 SME2 Multi-vector shift and convert instructions

Mnemonic	Instruction	See
SCVTF	Multi-vector signed integer convert to floating-point	SCVTF
SQCVT	Multi-vector signed saturating extract narrow	SQCVT (four registers) SQCVT (two registers)
SQCVTN	Multi-vector signed saturating extract narrow and interleave	SQCVTN
SQCVTU	Multi-vector signed saturating unsigned extract narrow	SQCVTU (four registers) SQCVTU (two registers)
SQCVTUN	Multi-vector signed saturating unsigned extract narrow and interleave	SQCVTUN
SQRSHR	Multi-vector signed saturating rounding shift right narrow by immediate	SQRSHR (four registers) SQRSHR (two registers)
SQRSHRN	Multi-vector signed saturating rounding shift right narrow by immediate and interleave	SQRSHRN
SQRSHRU	Multi-vector signed saturating rounding shift right unsigned narrow by immediate	SQRSHRU (four registers) SQRSHRU (two registers)
SQRSHRUN	Multi-vector signed saturating rounding shift right unsigned narrow by immediate and interleave	SQRSHRUN
SRSHL	Multi-vector signed rounding shift left by vector	SRSHL (multiple and single vector)
	Multi-vector signed rounding shift left	SRSHL (multiple vectors)
UCVTF	Multi-vector unsigned integer convert to floating-point	UCVTF
UQCVT	Multi-vector unsigned saturating extract narrow	UQCVT (four registers) UQCVT (two registers)
UQCVTN	Multi-vector unsigned saturating extract narrow and interleave	UQCVTN
UQRSHR	Multi-vector unsigned saturating rounding shift right narrow by immediate	UQRSHR (four registers) UQRSHR (two registers)
UQRSHRN	Multi-vector unsigned saturating rounding shift right narrow by immediate and interleave	UQRSHRN
URSHL	Multi-vector unsigned rounding shift left by vector	URSHL (multiple and single vector)
	Multi-vector unsigned rounding shift left	URSHL (multiple vectors)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.17 Outer product

Multiply, or widen and multiply, the sub-matrix in the first source vector by the sub-matrix in the second source vector and add or subtract from the destination tile.

C3.11.17.1 BFloat16 outer product

Table C3-279 SME BFloat16 outer product instructions

Mnemonic	Instruction	See
BFMOPA	BFloat16 floating-point outer product and accumulate	BFMOPA (non-widening)
	BFloat16 sum of outer products and accumulate	BFMOPA (widening)
BFMOPS	BFloat16 floating-point outer product and subtract	BFMOPS (non-widening)
	BFloat16 sum of outer products and subtract	BFMOPS (widening)

C3.11.17.2 Binary outer product

Table C3-280 SME2 Binary outer product instructions

Mnemonic	Instruction	See
BMOPA	Bitwise exclusive NOR population count outer product and accumulate	BMOPA
BMOPS	Bitwise exclusive NOR population count outer product and subtract	BMOPS

C3.11.17.3 Floating-point outer product

Table C3-281 SME Floating-point outer product instructions

Mnemonic	Instruction	See
FMOPA	Floating-point outer product and accumulate	FMOPA (non-widening)
	Half-precision floating-point sum of outer products and accumulate	FMOPA (widening)
FMOPS	Floating-point outer product and subtract	FMOPS (non-widening)
	Half-precision floating-point sum of outer products and subtract	FMOPS (widening)

C3.11.17.4 FP8 floating-point outer product

For FP8 outer product instructions, the unbounded intermediate sum of outer products is scaled by subtracting the value specified in FPMR.LSCALE from its unbounded exponent. For more information, see the instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-282 SME2 FP8 floating-point outer product instructions

Mnemonic	Instruction	See
FMOPA	8-bit floating-point sum of outer products and accumulate	FMOPA (widening, 2-way, FP8 to FP16)
		FMOPA (widening, 4-way)

C3.11.17.5 Integer outer product

Table C3-283 SME Integer outer product instructions

Mnemonic	Instruction	See
SMOPA	Signed integer sum of outer products and accumulate	SMOPA (2-way) SMOPA (4-way)
SMOPS	Signed integer sum of outer products and subtract	SMOPS (2-way) SMOPS (4-way)
SUMOPA	Signed by unsigned integer sum of outer products and accumulate	SUMOPA
SUMOPS	Signed by unsigned integer sum of outer products and subtract	SUMOPS
UMOPA	Unsigned integer sum of outer products and accumulate	UMOPA (2-way) UMOPA (4-way)
UMOPS	Unsigned integer sum of outer products and subtract	UMOPS (2-way) UMOPS (4-way)
USMOPA	Unsigned by signed integer sum of outer products and accumulate	USMOPA
USMOPS	Unsigned by signed integer sum of outer products and subtract	USMOPS

C3.11.18 Quarter-tile outer product

The quarter-tile outer product instructions generate four independent quarter-tile outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors. The resulting quarter-tile outer products are added to or subtracted from the destination ZA tile.

C3.11.18.1 BFloat16 quarter-tile outer product

Table C3-284 SME BFloat16 quarter-tile outer product instructions

Mnemonic	Instruction	See
BFMOP4A	BFloat16 quarter-tile outer products, accumulating	BFMOP4A (non-widening)
	BFloat16 quarter-tile sums of two outer products, accumulating	BFMOP4A (widening)
BFMOP4S	BFloat16 quarter-tile outer products, subtracting	BFMOP4S (non-widening)
	BFloat16 quarter-tile sums of two outer products, subtracting	BFMOP4S (widening)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.18.2 Floating-point quarter-tile outer product

Table C3-285 SME floating-point quarter-tile outer product instructions

Mnemonic	Instruction	See
FMOP4A	Floating-point quarter-tile outer products, accumulating	FMOP4A (non-widening)
	Half-precision quarter-tile sums of two outer products, accumulating	FMOP4A (widening, 2-way, FP16 to FP32)
FMOP4S	Floating-point quarter-tile outer products, subtracting	FMOP4S (non-widening)
	Half-precision quarter-tile sums of two outer products, subtracting	FMOP4S (widening)

C3.11.18.3 FP8 floating-point quarter-tile outer product

Table C3-286 SME2 FP8 floating-point quarter-tile outer product instructions

Mnemonic	Instruction	See
FMOP4A	8-bit floating-point quarter-tile sums of two outer products, accumulating	FMOP4A (widening, 2-way, FP8 to FP16)
	8-bit floating-point quarter-tile sums of four outer products, accumulating	FMOP4A (widening, 4-way)

C3.11.18.4 Integer quarter-tile outer product

Table C3-287 SME Integer quarter-tile outer product instructions

Mnemonic	Instruction	See
SMOP4A	Signed integer quarter-tile sums of two outer products, accumulating	SMOP4A (2-way)
	Signed integer quarter-tile sums of four outer products, accumulating	SMOP4A (4-way)
SMOP4S	Signed integer quarter-tile sums of two outer products, subtracting	SMOP4S (2-way)
	Signed integer quarter-tile sums of four outer products, subtracting	SMOP4S (4-way)
SUMOP4A	Signed by unsigned integer quarter-tile sums of four outer products, accumulating	SUMOP4A
SUMOP4S	Signed by unsigned integer quarter-tile sums of four outer products, subtracting	SUMOP4S
UMOP4A	Unsigned integer quarter-tile sums of two outer products, accumulating	UMOP4A (2-way)
	Unsigned integer quarter-tile sums of four outer products, accumulating	UMOP4A (4-way)
UMOP4S	Unsigned integer quarter-tile sums of two outer products, subtracting	UMOP4S (2-way)
	Unsigned integer quarter-tile sums of four outer products, subtracting	UMOP4S (4-way)
USMOP4A	Unsigned by signed integer quarter-tile sums of four outer products, accumulating	USMOP4A

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
USMOP4S	Unsigned by signed integer quarter-tile sums of four outer products, subtracting	USMOP4S

C3.11.19 Structured sparsity outer product

The SME structured sparsity outer product instructions accumulate the products or sum-of-products of all elements of the second source vector and selected elements of the first source vectors.

The SME widening structured sparsity outer product instructions widen and multiply a dense matrix by a 2:4 sparse matrix and accumulate to the destination tile. In a 2:4 structured sparsity pattern, two elements are selected per group of four elements, discarding the remaining two elements.

The SME non-widening structured sparsity outer product instructions multiply a dense matrix by a 1:2 sparse matrix and accumulate to the destination tile. In a 1:2 structured sparsity pattern, one element is selected out of a pair of elements, discarding the remaining element.

C3.11.19.1 BFloat16 sparsity outer product

Table C3-288 SME BFloat16 sparsity outer product instructions

Mnemonic	Instruction	See
BFTMOPA	BFloat16 sparse outer product, accumulating	BFTMOPA (non-widening)
	BFloat16 sparse sum of two outer products, accumulating	BFTMOPA (widening)

C3.11.19.2 Floating-point sparsity outer product

Table C3-289 SME Floating-point sparsity outer product instructions

Mnemonic	Instruction	See
FTMOPA	Floating-point sparse outer product, accumulating	FTMOPA (non-widening)
	Half-precision sparse sum of two outer products, accumulating	FTMOPA (widening, 2-way, FP16 to FP32)

C3.11.19.3 FP8 floating-point sparsity outer product

Table C3-290 SME2 FP8 floating-point sparsity outer product instructions

Mnemonic	Instruction	See
FTMOPA	8-bit floating-point sparse sum of two outer products, accumulating	FTMOPA (widening, 2-way, FP8 to FP16)
	8-bit floating-point sparse sum of four outer products, accumulating	FTMOPA (widening, 4-way)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.19.4 Integer sparsity outer product

Table C3-291 SME Integer sparsity outer product instructions

Mnemonic	Instruction	See
STMOPA	Signed integer sparse sum of two outer products, accumulating	STMOPA (2-way)
	Signed integer sparse sum of four outer products, accumulating	STMOPA (4-way)
SUTMOPA	Signed by unsigned integer sparse sum of four outer products, accumulating	SUTMOPA
USTMOPA	Unsigned by signed integer sparse sum of four outer products, accumulating	USTMOPA
UTMOPA	Unsigned integer sparse sum of two outer products, accumulating	UTMOPA (2-way)
	Unsigned integer sparse sum of four outer products, accumulating	UTMOPA (4-way)

C3.11.20 Stack frame operations

Multiply the Streaming SVE vector register size in bytes by an immediate and add the result to the destination general-purpose register or current stack pointer. RDSVL only places the result in the destination general-purpose register without addition.

Table C3-292 SME Stack frame instructions

Mnemonic	Instruction	See
ADDSPL	Add multiple of Streaming SVE predicate register size to scalar register	ADDSPL
ADDSVL	Add multiple of Streaming SVE vector register size to scalar register	ADDSVL
RDSVL	Read multiple of Streaming SVE vector register size to scalar register	RDSVL

C3.11.21 Unpack and extend

Unpack elements from the source vectors and extend them to place in elements of twice their size within the destination vectors.

Table C3-293 SME2 Unpack and extend instructions

Mnemonic	Instruction	See
SUNPK	Unpack and sign-extend multi-vector elements	SUNPK
UUNPK	Unpack and zero-extend multi-vector elements	UUNPK

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.11.22 Zero operations

Set all corresponding bytes or ZA vector groups to zero.

Table C3-294 SME/SME2 Zero instructions

Mnemonic	Instruction	See
ZERO	Zero ZT0	ZERO (table)
	Zero a list of 64-bit element ZA tiles	ZERO (tiles)

C3.11.22.1 Zero ZA array vectors

Table C3-295 SME2.1 Zero ZA array vectors instructions

Mnemonic	Instruction	See
ZERO	Zero ZA double-vector groups	ZERO (double-vector)
	Zero ZA quad-vector groups	ZERO (quad-vector)
	Zero ZA single-vector groups	ZERO (single-vector)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C4
A64 Instruction Set Encoding

This chapter describes the encoding of the A64 instruction set. It contains the following section:

- A64 instruction set encoding.

In this chapter:

- In the decode tables, an entry of `x` for a field value means the value of the field does not affect the decoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1 A64 instruction set encoding

31	30 29	28 25	24 0
op0		op1	

op0	op1	Instruction details
0	0000	Reserved
1	0000	SME encodings
x	0010	SVE encodings
x	00x1	UNALLOCATED
x	100x	Data Processing – Immediate
x	101x	Branches, Exception Generating and System instructions
x	x101	Data Processing – Register
x	x111	Data Processing – Scalar Floating-Point and Advanced SIMD
x	x1x0	Loads and Stores

Reserved

Reserved

The encodings in this section are decoded from A64 instruction set encoding.

31	30 29	28	27	26	25	24 16	15 0
0	op0	0	0	0	0	op1	

op0	op1	Instruction details
00	000000000	UDF
00	!= 000000000	UNALLOCATED
!= 00	xxxxxxxxx	UNALLOCATED

SME encodings

SME encodings

The encodings in this section are decoded from A64 instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30 29	28 25	24 10	9 6	5 0
1	op0	0 0 0 0	op1		op2

op0	op1	op2	Instruction details
00	x00xxxxx0x00000	0xx0xx	SME2 Quarter Tile Outer Product - 16-bit and 32-bit
00	x00xxxxx0x00000	1xx0xx	UNALLOCATED
00	x00xxxxx0x00001	xxx0xx	UNALLOCATED
00	x00xxxxx0x0001x	xxx0xx	UNALLOCATED
00	x00xxxxx0x001xx	xxx0xx	UNALLOCATED
00	x00xxxxx0x01xxx	xxx0xx	UNALLOCATED
00	x00xxxxx1x0xxxx	xxx0xx	UNALLOCATED
00	x01xxxxxxx0xxxx	xxx0xx	SME2 Sparse Outer Product
00	x0xxxxxxxx0xxxx	xxx1xx	UNALLOCATED
00	x0xxxxxxxx1xxxx	xxxxxx	UNALLOCATED
00	x10xxxxxxxxxxxx	xx00xx	SME FP Outer Product - 32 bit
00	x10xxxxxxxxxxxx	xx10xx	SME2 Outer Product - Misc
01	00xxxxxxxxxxxxx	xxxxxx	SME2 Multi-vector - Memory (Contiguous)
01	10xxxxxxxxxxxxx	xxxxxx	SME2 Multi-vector - Memory (Strided)
01	x10xxxxxxxxxxxx	xxx0xx	SME Integer Outer Product - 32 bit
0x	x10xxxxxxxxxxxx	xxx1xx	UNALLOCATED
0x	x11xxxxx0000000	0x1xxx	SME2 Quarter Tile Outer Product - 64-bit
0x	x11xxxxx0000000	1x1xxx	UNALLOCATED
0x	x11xxxxx0000001	xx1xxx	UNALLOCATED
0x	x11xxxxx000001x	xx1xxx	UNALLOCATED
0x	x11xxxxx00001xx	xx1xxx	UNALLOCATED
0x	x11xxxxx0001xxx	xx1xxx	UNALLOCATED
0x	x11xxxxx001xxxx	xx1xxx	UNALLOCATED
0x	x11xxxxx01xxxxx	xx1xxx	UNALLOCATED
0x	x11xxxxx1xxxxxx	xx1xxx	UNALLOCATED
0x	x11xxxxxxxxxxxx	xx0xxx	SME Outer Product - 64 bit
10	0000010xxxxxxxx	xxxxxx	SME zero array
10	0000011xxxxxxxx	xxxxxx	SME2 Multiple Zero
10	0010010xxxxxxxx	xxxxxx	SME2 zero lookup table
10	0010011xxxxxxxx	xxxxxx	SME2 Move Lookup Table
10	00x011xxxxxxxxx	xxxxxx	UNALLOCATED
10	010011xxxxxxxxx	xxxxxx	SME2 Expand Lookup Table (Non-contiguous)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
10	011011xxxxxxxxx	xxxxxx	UNALLOCATED
10	01x001xxxxxxxxx	xxxxxx	SME2 Expand Lookup Table (Contiguous)
10	0xx000x0xxxxxxx	x0xxxx	SME Move into Array
10	0xx000x0xxxxxxx	x1xxxx	UNALLOCATED
10	0xx000x1xxxxxxx	xxxxxx	SME Move from Array
10	0xx010xxxxxxxxx	xx0xxx	SME Add Vector to Array
10	0xx010xxxxxxxxx	xx1xxx	UNALLOCATED
10	0xx1xxxxxxxxxxx	xxxxxx	UNALLOCATED
10	10x10xxxx0xxxxx	xxxxxx	SME2 Multi-vector - Multiple and Single Array Vectors (Two registers)
10	10x11xxxx0xxxxx	xxxxxx	SME2 Multi-vector - Multiple and Single Array Vectors (Four registers)
10	11x1xxxx00xxxxx	xxxxxx	SME2 Multi-vector - Multiple Array Vectors (Two registers)
10	11x1xxxx10xxxxx	xxxxxx	SME2 Multi-vector - Multiple Array Vectors (Four registers)
10	1xx00xxxxxxxxxx	xxxxxx	SME2 Multi-vector - Indexed (One register)
10	1xx01xxxx0xxxxx	xxxxxx	SME2 Multi-vector - Indexed (Two registers)
10	1xx01xxxx1xxxxx	xxxxxx	SME2 Multi-vector - Indexed (Four registers)
10	1xx10xxxx10100x	xxxxxx	SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)
10	1xx10xxxx10101x	xxxx0x	SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)
10	1xx10xxxx10101x	xxxx1x	UNALLOCATED
10	1xx11xxxx1010xx	xxxxxx	UNALLOCATED
10	1xx1xxx00101110	xxxx0x	SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)
10	1xx1xxx00101111	xxxx0x	SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers)
10	1xx1xxx0010111x	xxxx1x	UNALLOCATED
10	1xx1xxx1010111x	xxxxxx	UNALLOCATED
10	1xx1xxxx0101100	xxxxxx	SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)
10	1xx1xxxx0101101	xxxxxx	SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers)
10	1xx1xxxx01111xx	xxxxxx	UNALLOCATED
10	1xx1xxxx11x11xx	xxxxxx	UNALLOCATED
10	1xx1xxxxx100xxx	xxxxxx	SME2 Multi-vector - SVE Select
10	1xx1xxxxx110xxx	xxxxxx	SME2 Multi-vector - SVE Constructive Binary
10	1xx1xxxxx111000	xxxxxx	SME2 Multi-vector - SVE Constructive Unary

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
10	1xx1xxxxx111001	0xxxx0	SME2 Multi-vector - FP Multiply
10	1xx1xxxxx111001	!= 0xxxx0	UNALLOCATED
10	1xx1xxxxx111010	0xxxx0	SME2 Multiple and Single Vector - FP multiply
10	1xx1xxxxx111010	!= 0xxxx0	UNALLOCATED
10	1xx1xxxxx111011	xxxxxx	UNALLOCATED
11	xxxxxxxxxxxxxxx	xxxxxx	SME Memory

SME2 Quarter Tile Outer Product - 16-bit and 32-bit

SME2 Quarter Tile Outer Product - 16-bit and 32-bit

The encodings in this section are decoded from SME encodings.

31							24	23	22	21	20	19			16	15	14	13		11	10	9	8	5	4	3	0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	1	0	0	0	Pg		0	Pd	

op0	op1	op2	op3	op4	Instruction details
0	0	0	x0	x	SME2 FP32 non-widening quarter tile outer product
0	0	0	x1	0	UNALLOCATED
0	1	0	00	x	SME2 FP8 to FP32 quarter tile outer product
0	1	0	01	0	SME2 FP8 to FP16 quarter tile outer product
0	1	0	10	1	UNALLOCATED
0	1	0	1x	0	UNALLOCATED
1	0	0	x0	x	SME2 BF16 to FP32 quarter tile outer product
1	0	0	x1	0	SME2 FP16 non-widening quarter tile outer product
1	1	0	x0	x	SME2 FP16 to FP32 quarter tile outer product
1	1	0	x1	0	SME2 BF16 non-widening quarter tile outer product
x	0	1	x1	x	SME2 Int16 to Int32 quarter tile outer product
x	1	1	x1	x	UNALLOCATED
x	x	0	x1	1	UNALLOCATED
x	x	1	x0	x	SME2 Int8 to Int32 quarter tile outer product

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 FP32 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	S	0	0	ZAda	

M	N	S	Instruction Details	Condition
0	0	0	FMOP4A (non-widening) — Single-precision, single vectors	FEAT_SME_MOP4
0	0	1	FMOP4S (non-widening) — Single-precision, single vectors	FEAT_SME_MOP4
0	1	0	FMOP4A (non-widening) — Single-precision, multiple and single vectors	FEAT_SME_MOP4
0	1	1	FMOP4S (non-widening) — Single-precision, multiple and single vectors	FEAT_SME_MOP4
1	0	0	FMOP4A (non-widening) — Single-precision, single and multiple vectors	FEAT_SME_MOP4
1	0	1	FMOP4S (non-widening) — Single-precision, single and multiple vectors	FEAT_SME_MOP4
1	1	0	FMOP4A (non-widening) — Single-precision, multiple vectors	FEAT_SME_MOP4
1	1	1	FMOP4S (non-widening) — Single-precision, multiple vectors	FEAT_SME_MOP4

SME2 FP8 to FP32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	0	0	0	ZAda	

M	N	Instruction Details	Condition
0	0	FMOP4A (widening, 4-way) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32
0	1	FMOP4A (widening, 4-way) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32
1	0	FMOP4A (widening, 4-way) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32
1	1	FMOP4A (widening, 4-way) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 FP8 to FP16 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	0	1	0	0	ZA

M	N	Instruction Details	Condition
0	0	FMOP4A (widening, 2-way, FP8 to FP16) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16
0	1	FMOP4A (widening, 2-way, FP8 to FP16) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16
1	0	FMOP4A (widening, 2-way, FP8 to FP16) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16
1	1	FMOP4A (widening, 2-way, FP8 to FP16) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F8F16

SME2 BF16 to FP32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	M	Zm			0	0	0	0	0	0	0	0	N	Zn			0	S	0	0	ZAda	

M	N	S	Instruction Details	Condition
0	0	0	BFMOP4A (widening) — Single vectors	FEAT_SME_MOP4
0	0	1	BFMOP4S (widening) — Single vectors	FEAT_SME_MOP4
0	1	0	BFMOP4A (widening) — Multiple and single vectors	FEAT_SME_MOP4
0	1	1	BFMOP4S (widening) — Multiple and single vectors	FEAT_SME_MOP4
1	0	0	BFMOP4A (widening) — Single and multiple vectors	FEAT_SME_MOP4
1	0	1	BFMOP4S (widening) — Single and multiple vectors	FEAT_SME_MOP4
1	1	0	BFMOP4A (widening) — Multiple vectors	FEAT_SME_MOP4
1	1	1	BFMOP4S (widening) — Multiple vectors	FEAT_SME_MOP4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 FP16 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19	17	16	15	14					10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	M	Zm		0	0	0	0	0	0	0	0	N	Zn		0	S	1	0	0	ZA

M	N	S	Instruction Details	Condition
0	0	0	FMOP4A (non-widening) — Half-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
0	0	1	FMOP4S (non-widening) — Half-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
0	1	0	FMOP4A (non-widening) — Half-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
0	1	1	FMOP4S (non-widening) — Half-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	0	0	FMOP4A (non-widening) — Half-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	0	1	FMOP4S (non-widening) — Half-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	1	0	FMOP4A (non-widening) — Half-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16
1	1	1	FMOP4S (non-widening) — Half-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F16F16

SME2 FP16 to FP32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19	17	16	15	14					10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	M	Zm		0	0	0	0	0	0	0	0	N	Zn		0	S	0	0	ZAda	

M	N	S	Instruction Details	Condition
0	0	0	FMOP4A (widening, 2-way, FP16 to FP32) — Single vectors	FEAT_SME_MOP4
0	0	1	FMOP4S (widening) — Single vectors	FEAT_SME_MOP4
0	1	0	FMOP4A (widening, 2-way, FP16 to FP32) — Multiple and single vectors	FEAT_SME_MOP4
0	1	1	FMOP4S (widening) — Multiple and single vectors	FEAT_SME_MOP4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	N	S	Instruction Details	Condition
1	0	0	FMOP4A (widening, 2-way, FP16 to FP32) — Single and multiple vectors	FEAT_SME_MOP4
1	0	1	FMOP4S (widening) — Single and multiple vectors	FEAT_SME_MOP4
1	1	0	FMOP4A (widening, 2-way, FP16 to FP32) — Multiple vectors	FEAT_SME_MOP4
1	1	1	FMOP4S (widening) — Multiple vectors	FEAT_SME_MOP4

SME2 BF16 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	M	Zm			0	0	0	0	0	0	0		N	Zn			0	S	1	0	0	ZA

M	N	S	Instruction Details	Condition
0	0	0	BFMOP4A (non-widening) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
0	0	1	BFMOP4S (non-widening) — Single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
0	1	0	BFMOP4A (non-widening) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
0	1	1	BFMOP4S (non-widening) — Multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	0	0	BFMOP4A (non-widening) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	0	1	BFMOP4S (non-widening) — Single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	1	0	BFMOP4A (non-widening) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16
1	1	1	BFMOP4S (non-widening) — Multiple vectors	FEAT_SME_MOP4 && FEAT_SME_B16B16

SME2 Int16 to Int32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	u0	0	0	0	M	Zm			0	1	0	0	0	0	0		N	Zn			0	S	1	0	ZAda	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	M	N	S	Instruction Details	Condition
0	0	0	0	SMOP4A (2-way) — Single vectors	FEAT_SME_MOP4
0	0	0	1	SMOP4S (2-way) — Single vectors	FEAT_SME_MOP4
0	0	1	0	SMOP4A (2-way) — Multiple and single vectors	FEAT_SME_MOP4
0	0	1	1	SMOP4S (2-way) — Multiple and single vectors	FEAT_SME_MOP4
0	1	0	0	SMOP4A (2-way) — Single and multiple vectors	FEAT_SME_MOP4
0	1	0	1	SMOP4S (2-way) — Single and multiple vectors	FEAT_SME_MOP4
0	1	1	0	SMOP4A (2-way) — Multiple vectors	FEAT_SME_MOP4
0	1	1	1	SMOP4S (2-way) — Multiple vectors	FEAT_SME_MOP4
1	0	0	0	UMOP4A (2-way) — Single vectors	FEAT_SME_MOP4
1	0	0	1	UMOP4S (2-way) — Single vectors	FEAT_SME_MOP4
1	0	1	0	UMOP4A (2-way) — Multiple and single vectors	FEAT_SME_MOP4
1	0	1	1	UMOP4S (2-way) — Multiple and single vectors	FEAT_SME_MOP4
1	1	0	0	UMOP4A (2-way) — Single and multiple vectors	FEAT_SME_MOP4
1	1	0	1	UMOP4S (2-way) — Single and multiple vectors	FEAT_SME_MOP4
1	1	1	0	UMOP4A (2-way) — Multiple vectors	FEAT_SME_MOP4
1	1	1	1	UMOP4S (2-way) — Multiple vectors	FEAT_SME_MOP4

SME2 Int8 to Int32 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 16-bit and 32-bit.

31						25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	u0	0	0	u1	M	Zm			0	1	0	0	0	0	0		N	Zn			0	S	0	0	ZAda	

u0	u1	M	N	S	Instruction Details	Condition
0	0	0	0	0	SMOP4A (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
0	0	0	0	1	SMOP4S (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
0	0	0	1	0	SMOP4A (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	0	0	1	1	SMOP4S (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	0	1	0	0	SMOP4A (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Condition
0	0	1	0	1	SMOP4S (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	0	1	1	0	SMOP4A (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4
0	0	1	1	1	SMOP4S (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4
0	1	0	0	0	SUMOP4A — 32-bit, single vectors	FEAT_SME_MOP4
0	1	0	0	1	SUMOP4S — 32-bit, single vectors	FEAT_SME_MOP4
0	1	0	1	0	SUMOP4A — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	1	0	1	1	SUMOP4S — 32-bit, multiple and single vectors	FEAT_SME_MOP4
0	1	1	0	0	SUMOP4A — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	1	1	0	1	SUMOP4S — 32-bit, single and multiple vectors	FEAT_SME_MOP4
0	1	1	1	0	SUMOP4A — 32-bit, multiple vectors	FEAT_SME_MOP4
0	1	1	1	1	SUMOP4S — 32-bit, multiple vectors	FEAT_SME_MOP4
1	0	0	0	0	USMOP4A — 32-bit, single vectors	FEAT_SME_MOP4
1	0	0	0	1	USMOP4S — 32-bit, single vectors	FEAT_SME_MOP4
1	0	0	1	0	USMOP4A — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	0	0	1	1	USMOP4S — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	0	1	0	0	USMOP4A — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	0	1	0	1	USMOP4S — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	0	1	1	0	USMOP4A — 32-bit, multiple vectors	FEAT_SME_MOP4
1	0	1	1	1	USMOP4S — 32-bit, multiple vectors	FEAT_SME_MOP4
1	1	0	0	0	UMOP4A (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
1	1	0	0	1	UMOP4S (4-way) — 32-bit, single vectors	FEAT_SME_MOP4
1	1	0	1	0	UMOP4A (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	1	0	1	1	UMOP4S (4-way) — 32-bit, multiple and single vectors	FEAT_SME_MOP4
1	1	1	0	0	UMOP4A (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	1	1	0	1	UMOP4S (4-way) — 32-bit, single and multiple vectors	FEAT_SME_MOP4
1	1	1	1	0	UMOP4A (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4
1	1	1	1	1	UMOP4S (4-way) — 32-bit, multiple vectors	FEAT_SME_MOP4

SME2 Sparse Outer Product

SME2 Sparse Outer Product

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encodings in this section are decoded from SME encodings.

31							24	23	22	21	20				16	15	14	13	12	11	10	9					4	3			0
0	0	1	0	0	1	0	1			1						0	0	op0		op1								op2			

op0	op1	op2	op3	op4	op5	Instruction details	Condition
0	0	0	0	0	x	FTMOPA (non-widening) — Single-precision	FEAT_SME_TMOP
0	0	0	0	1	0	UNALLOCATED	-
0	1	0	0	0	x	FTMOPA (widening, 4-way)	FEAT_SME_TMOP && FEAT_SME_F8F32
0	1	0	0	1	0	FTMOPA (widening, 2-way, FP8 to FP16)	FEAT_SME_TMOP && FEAT_SME_F8F16
1	0	0	0	0	x	BFTMOPA (widening)	FEAT_SME_TMOP
1	0	0	0	1	0	FTMOPA (non-widening) — Half-precision	FEAT_SME_TMOP && FEAT_SME_F16F16
1	1	0	0	0	x	FTMOPA (widening, 2-way, FP16 to FP32)	FEAT_SME_TMOP
1	1	0	0	1	0	BFTMOPA (non-widening)	FEAT_SME_TMOP && FEAT_SME_B16B16
x	0	1	0	1	x	SME2 Int16 to Int32 sparse outer product	-
x	1	1	0	1	x	UNALLOCATED	-
x	x	0	0	1	1	UNALLOCATED	-
x	x	1	0	0	x	SME Int8 to Int32 sparse outer product	-
x	x	x	1	x	x	UNALLOCATED	-

SME2 Int16 to Int32 sparse outer product

The encodings in this section are decoded from SME2 Sparse Outer Product.

31							24	23	22	21	20				16	15		13	12	11	10	9			5	4	3			0
0	0	1	0	0	1	0	1	size		1	Rm					0	0	0	sf	U	lt	Rn				eq	Pd			

u0	Instruction Details	Condition
0	STMOPA (2-way)	FEAT_SME_TMOP
1	UTMOPA (2-way)	FEAT_SME_TMOP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME Int8 to Int32 sparse outer product

The encodings in this section are decoded from SME2 Sparse Outer Product.

31							24	23	22	21	20				16	15	14	13	12	11	10	9				5	4	3			0
0	0	1	0	0	1	0	1	op	sz	1	Rm					0	0	1	0	0	0	Rn					ne	0	0	0	0

u0	u1	Instruction Details	Condition
0	0	STMOPA (4-way)	FEAT_SME_TMOP
0	1	SUTMOPA	FEAT_SME_TMOP
1	0	USTMOPA	FEAT_SME_TMOP
1	1	UTMOPA (4-way)	FEAT_SME_TMOP

SME FP Outer Product - 32 bit

SME FP Outer Product - 32 bit

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20																		5	4	3	2	1	0
1	0	0	0	0	0	0	op0	1	0	op1																				op2	0	0		

op0	op1	op2	Instruction details	Condition
0	0	x	SME FP32 outer product	-
0	1	0	FMOPA (widening, 4-way)	FEAT_SME_F8F32
0	1	1	UNALLOCATED	-
1	0	x	SME BF16 widening outer product	-
1	1	x	SME FP16 widening outer product	-

SME FP32 outer product

The encodings in this section are decoded from SME FP Outer Product - 32 bit.

31							24	23	22	21	20				16	15	14	13			11	10					5	4	3	2			0
0	0	1	0	0	1	0	1			1	op0					0	1	op1				op2						1	op3				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	Instruction Details	Condition
0	FMOPA (non-widening)	FEAT_SME
1	FMOPS (non-widening)	FEAT_SME

SME BF16 widening outer product

The encodings in this section are decoded from SME FP Outer Product - 32 bit.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		S	0	0	ZAda	

S	Instruction Details	Condition
0	BFMOPA (widening)	FEAT_SME
1	BFMOPS (widening)	FEAT_SME

SME FP16 widening outer product

The encodings in this section are decoded from SME FP Outer Product - 32 bit.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	1	Zm		Pm		Pn		Zn		S	0	0	ZAda	

S	Instruction Details	Condition
0	FMOPA (widening, 2-way, FP16 to FP32)	FEAT_SME
1	FMOPS (widening)	FEAT_SME

SME2 Outer Product - Misc

SME2 Outer Product - Misc

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20	5	4	3	2	1	0
1	0	0	0	0	0	0	op0	1	0	op1			op2	1	0	op3	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
0	0	x	x	SME2 32-bit binary outer product	-
0	1	0	0	FMOPA (widening, 2-way, FP8 to FP16)	FEAT_SME_F8F16
0	1	1	0	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	0	x	0	SME2 FP16 non-widening outer product	-
1	1	x	0	SME2 BF16 non-widening outer product	-
1	x	x	1	UNALLOCATED	-

SME2 32-bit binary outer product

The encodings in this section are decoded from SME2 Outer Product - Misc.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1 0 0 0 0 0 0		0	1	0	0	Zm		Pm		Pn		Zn		S	1	0	ZAda	

S	Instruction Details	Condition
0	BMOPA	FEAT_SME2
1	BMOPS	FEAT_SME2

SME2 FP16 non-widening outer product

The encodings in this section are decoded from SME2 Outer Product - Misc.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1 0 0 0 0 0 0		1	1	0	0	Zm		Pm		Pn		Zn		S	1	0	0	ZAda

S	Instruction Details	Condition
0	FMOPA (non-widening)	FEAT_SME_F16F16
1	FMOPS (non-widening)	FEAT_SME_F16F16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 BF16 non-widening outer product

The encodings in this section are decoded from SME2 Outer Product - Misc.

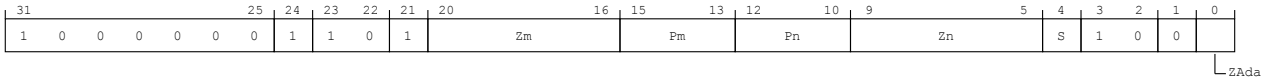

S	Instruction Details	Condition
0	BFMOPA (non-widening)	FEAT_SME_B16B16
1	BFMOPS (non-widening)	FEAT_SME_B16B16

SME2 Multi-vector - Memory (Contiguous)

SME2 Multi-vector - Memory (Contiguous)

The encodings in this section are decoded from SME encodings.

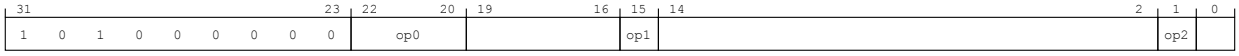

op0	op1	op2	Instruction details
00x	0	x	SME2 multi-vec contiguous load (scalar plus scalar, two registers)
00x	1	0	SME2 multi-vec contiguous load (scalar plus scalar, four registers)
01x	0	x	SME2 multi-vec contiguous store (scalar plus scalar, two registers)
01x	1	0	SME2 multi-vec contiguous store (scalar plus scalar, four registers)
100	0	x	SME2 multi-vec contiguous load (scalar plus immediate, two registers)
100	1	0	SME2 multi-vec contiguous load (scalar plus immediate, four registers)
110	0	x	SME2 multi-vec contiguous store (scalar plus immediate, two registers)
110	1	0	SME2 multi-vec contiguous store (scalar plus immediate, four registers)
1x1	x	x	UNALLOCATED
!= 1x1	1	1	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec contiguous load (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20			16	15	14	13	12			10	9				5	4				1	0
1	0	1	0	0	0	0	0	0	0	0	Rm				0	msz			PNg			Rn					Zt					N

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous load (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20			16	15	14	13	12			10	9				5	4			2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm				1	msz			PNg			Rn					Zt				0	N

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
10	0	LD1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm		0	msz		PNg		Rn		Zt		N

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31										21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm		1	msz		PNg		Rn		Zt		0	N

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus scalar, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous load (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31								23	22		20	19		16	15	14	13	12		10	9		5	4		1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4			0	msz		PNg			Rn			Zt			N

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec contiguous load (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31–23	22–20	19–16	15	14–13	12–10	9–5	4–2	1	0
1 0 1 0 0 0 0 0 0	1 0 0	imm4	1	msz	PNg	Rn	Zt	0	N

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	LDNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	LD1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	LDNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	LD1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	LDNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	LD1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	LDNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31–23	22–20	19–16	15	14–13	12–10	9–5	4–1	0
1 0 1 0 0 0 0 0 0	1 1 0	imm4	0	msz	PNg	Rn	Zt	N

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
01	0	ST1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 multi-vec contiguous store (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Contiguous).

31–23	22–20	19–16	15	14–13	12–10	9–5	4–2	1	0
1 0 1 0 0 0 0 0 0	1 1 0	imm4	1	msz	PNg	Rn	Zt	0	N

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
00	1	STNT1B (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	0	ST1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
01	1	STNT1H (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	0	ST1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
10	1	STNT1W (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	0	ST1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1
11	1	STNT1D (scalar plus immediate, consecutive registers)	FEAT_SME2 || FEAT_SVE2p1

SME2 Multi-vector - Memory (Strided)

SME2 Multi-vector - Memory (Strided)

The encodings in this section are decoded from SME encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–23	22–20	19–16	15	14–3	2	1–0
1 0 1 0 0 0 0 1 0	op0		op1		op2	

op0	op1	op2	Instruction details
00x	0	x	SME2 multi-vec non-contiguous load (scalar plus scalar, two registers)
00x	1	0	SME2 multi-vec non-contiguous load (scalar plus scalar, four registers)
01x	0	x	SME2 multi-vec non-contiguous store (scalar plus scalar, two registers)
01x	1	0	SME2 multi-vec non-contiguous store (scalar plus scalar, four registers)
100	0	x	SME2 multi-vec non-contiguous load (scalar plus immediate, two registers)
100	1	0	SME2 multi-vec non-contiguous load (scalar plus immediate, four registers)
110	0	x	SME2 multi-vec non-contiguous store (scalar plus immediate, two registers)
110	1	0	SME2 multi-vec non-contiguous store (scalar plus immediate, four registers)
1x1	x	x	UNALLOCATED
!= 1x1	1	1	UNALLOCATED

SME2 multi-vec non-contiguous load (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31–21	20–16	15	14–13	12–10	9–5	4	3	2–0
1 0 1 0 0 0 0 1 0 0 0	Rm	0	msz	PNg	Rn	T	N	Zt

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus scalar, strided registers)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
10	1	LDNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous load (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm		1	msz		PNg		Rn		T	N	0	Zt	

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	LDNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous store (scalar plus scalar, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm		0	msz		PNg		Rn		T	N	Zt	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous store (scalar plus scalar, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31										21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm		1	msz		PNg		Rn		T	N	0	Zt	

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus scalar, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus scalar, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus scalar, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus scalar, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus scalar, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus scalar, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus scalar, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus scalar, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous load (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–23	22–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 0 1 0 0 0 0 1 0	1 0 0	imm4	0	msz	PNg	Rn	T	N	Zt

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	LDNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous load (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31–23	22–20	19–16	15	14–13	12–10	9–5	4	3	2	1–0
1 0 1 0 0 0 0 1 0	1 0 0	imm4	1	msz	PNg	Rn	T	N	0	Zt

msz	N	Instruction Details	Condition
00	0	LD1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	LDNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	LD1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	LDNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	LD1W (scalar plus immediate, strided registers)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
10	1	LDNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	LD1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	LDNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous store (scalar plus immediate, two registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31								23	22		20	19			16	15	14		13	12			10	9			5	4	3	2			0
1	0	1	0	0	0	0	1	0	1	1	0	imm4				0	msz			PNg				Rn				T	N	Zt			

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME2 multi-vec non-contiguous store (scalar plus immediate, four registers)

The encodings in this section are decoded from SME2 Multi-vector - Memory (Strided).

31								23	22		20	19			16	15	14		13	12			10	9			5	4	3	2	1		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4				1	msz			PNg				Rn				T	N	0	Zt		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	N	Instruction Details	Condition
00	0	ST1B (scalar plus immediate, strided registers)	FEAT_SME2
00	1	STNT1B (scalar plus immediate, strided registers)	FEAT_SME2
01	0	ST1H (scalar plus immediate, strided registers)	FEAT_SME2
01	1	STNT1H (scalar plus immediate, strided registers)	FEAT_SME2
10	0	ST1W (scalar plus immediate, strided registers)	FEAT_SME2
10	1	STNT1W (scalar plus immediate, strided registers)	FEAT_SME2
11	0	ST1D (scalar plus immediate, strided registers)	FEAT_SME2
11	1	STNT1D (scalar plus immediate, strided registers)	FEAT_SME2

SME Integer Outer Product - 32 bit

SME Integer Outer Product - 32 bit

The encodings in this section are decoded from SME encodings.

31						25	24	23	22	21	20	4	3	2	1	0
1	0	1	0	0	0	0		1	0	op0			op1	0		

op0	op1	Instruction details
0	1	SME2 Int16 two-way outer product
1	1	UNALLOCATED
x	0	SME Int8 outer product

SME2 Int16 two-way outer product

The encodings in this section are decoded from SME Integer Outer Product - 32 bit.

31						25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	0	0	Zm		Pm		Pn		Zn		S	1	0	ZAda	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	S	Instruction Details	Condition
0	0	SMOPA (2-way)	FEAT_SME2
0	1	SMOPS (2-way)	FEAT_SME2
1	0	UMOPA (2-way)	FEAT_SME2
1	1	UMOPS (2-way)	FEAT_SME2

SME Int8 outer product

The encodings in this section are decoded from SME Integer Outer Product - 32 bit.

31						25	24	23	22	21	20			16	15		13	12		10	9			5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	0	u1	Zm				Pm			Pn			Zn				S	0	0	ZAda	

u0	u1	S	Instruction Details	Condition
0	0	0	SMOPA (4-way)	FEAT_SME
0	0	1	SMOPS (4-way)	FEAT_SME
0	1	0	SUMOPA (4-way)	FEAT_SME
0	1	1	SUMOPS	FEAT_SME
1	0	0	USMOPA (4-way)	FEAT_SME
1	0	1	USMOPS	FEAT_SME
1	1	0	UMOPA (4-way)	FEAT_SME
1	1	1	UMOPS (4-way)	FEAT_SME

SME2 Quarter Tile Outer Product - 64-bit

SME2 Quarter Tile Outer Product - 64-bit

The encodings in this section are decoded from SME encodings.

31	30	29	28			25	24	23	22	21	20			17	16						10	9			6	5	4	3	2		0
1	0	op0	0	0	0	0	op1	1	1	op2					0	0	0	0	0	0	0					0		1			

op0	op1	op2	Instruction details
0	0	0	SME2 FP64 non-widening quarter tile outer product
0	0	1	UNALLOCATED
0	1	x	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
1	x	x	SME2 Int16 to Int64 quarter tile outer product

SME2 FP64 non-widening quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 64-bit.

31	30	29	28			25	24	23	22	21	20	19	17	16						10	9	8	6	5	4	3	2	0
1	0	0	0	0	0	0	0	1	1	0	M	Zm		0	0	0	0	0	0	0	N	Zn		0	S	1	ZAda	

M	N	S	Instruction Details	Condition
0	0	0	FMOP4A (non-widening) — Double-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
0	0	1	FMOP4S (non-widening) — Double-precision, single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
0	1	0	FMOP4A (non-widening) — Double-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
0	1	1	FMOP4S (non-widening) — Double-precision, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	0	0	FMOP4A (non-widening) — Double-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	0	1	FMOP4S (non-widening) — Double-precision, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	1	0	FMOP4A (non-widening) — Double-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64
1	1	1	FMOP4S (non-widening) — Double-precision, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_F64F64

SME2 Int16 to Int64 quarter tile outer product

The encodings in this section are decoded from SME2 Quarter Tile Outer Product - 64-bit.

31	30	29	28			25	24	23	22	21	20	19	17	16						10	9	8	6	5	4	3	2	0
1	0	1	0	0	0	0	u0	1	1	u1	M	Zm		0	0	0	0	0	0	0	N	Zn		0	S	1	ZAda	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Condition
0	0	0	0	0	SMOP4A (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	0	0	1	SMOP4S (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	0	1	0	SMOP4A (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	0	1	1	SMOP4S (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	0	0	SMOP4A (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	0	1	SMOP4S (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	1	0	SMOP4A (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	0	1	1	1	SMOP4S (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	0	0	SUMOP4A — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	0	1	SUMOP4S — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	1	0	SUMOP4A — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	0	1	1	SUMOP4S — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	0	0	SUMOP4A — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	0	1	SUMOP4S — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	1	0	SUMOP4A — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
0	1	1	1	1	SUMOP4S — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	0	0	USMOP4A — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	0	1	USMOP4S — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	1	0	USMOP4A — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	0	1	1	USMOP4S — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	0	0	USMOP4A — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	0	1	USMOP4S — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	1	0	USMOP4A — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	0	1	1	1	USMOP4S — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	0	0	UMOP4A (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	0	1	UMOP4S (4-way) — 64-bit, single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	1	0	UMOP4A (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	0	1	1	UMOP4S (4-way) — 64-bit, multiple and single vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

u0	u1	M	N	S	Instruction Details	Condition
1	1	1	0	0	UMOP4A (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	0	1	UMOP4S (4-way) — 64-bit, single and multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	1	0	UMOP4A (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64
1	1	1	1	1	UMOP4S (4-way) — 64-bit, multiple vectors	FEAT_SME_MOP4 && FEAT_SME_I16I64

SME Outer Product - 64 bit

SME Outer Product - 64 bit

The encodings in this section are decoded from SME encodings.

31	30	29	28			25	24	23	22	21	20 … 4	3	2 … 0
1	0	op0	0	0	0	0	op1	1	1	op2		0	

op0	op1	op2	Instruction details
0	0	0	SME FP64 outer product
0	0	1	UNALLOCATED
0	1	x	UNALLOCATED
1	x	x	SME Int16 outer product

SME FP64 outer product

The encodings in this section are decoded from SME Outer Product - 64 bit.

31	30	29	28			25	24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4	3	2 … 0
1	0	0	0	0	0	0	0	1	1	0	Zm	Pm	Pn	Zn	S	0	ZAda

S	Instruction Details	Condition
0	FMOPA (non-widening)	FEAT_SME_F64F64
1	FMOPS (non-widening)	FEAT_SME_F64F64

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME Int16 outer product

The encodings in this section are decoded from SME Outer Product - 64 bit.

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	u0	1	1	u1	Zm		Pm		Pn		Zn		S	0	ZAda	

u0	u1	S	Instruction Details	Condition
0	0	0	SMOPA (4-way)	FEAT_SME_I16I64
0	0	1	SMOPS (4-way)	FEAT_SME_I16I64
0	1	0	SUMOPA (4-way)	FEAT_SME_I16I64
0	1	1	SUMOPS	FEAT_SME_I16I64
1	0	0	USMOPA (4-way)	FEAT_SME_I16I64
1	0	1	USMOPS	FEAT_SME_I16I64
1	1	0	UMOPA (4-way)	FEAT_SME_I16I64
1	1	1	UMOPS (4-way)	FEAT_SME_I16I64

SME zero array

SME zero array

The encodings in this section are decoded from SME encodings.

31												18	17	8	7	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	op0		

op0	Instruction details	Condition
0000000000	ZERO (tiles)	FEAT_SME
!= 0000000000	UNALLOCATED	-

SME2 Multiple Zero

SME2 Multiple Zero

The encodings in this section are decoded from SME encodings.

31												18	17	13	12	3	2	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1		op0			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0000000000	SME multiple vectors zero array
!= 0000000000	UNALLOCATED

SME multiple vectors zero array

The encodings in this section are decoded from SME2 Multiple Zero.

31													18	17		15	14	13	12									3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	opc			Rv		0	0	0	0	0	0	0	0	0	0	opc2	

opc	opc2	Instruction Details	Condition
x1x	1xx	UNALLOCATED	-
000	xxx	ZERO (single-vector) — Two ZA single-vectors	FEAT_SME2p1
001	xxx	ZERO (double-vector) — One ZA double-vector	FEAT_SME2p1
010	0xx	ZERO (double-vector) — Two ZA double-vectors	FEAT_SME2p1
011	0xx	ZERO (double-vector) — Four ZA double-vectors	FEAT_SME2p1
100	xxx	ZERO (single-vector) — Four ZA single-vectors	FEAT_SME2p1
101	0xx	ZERO (quad-vector) — One ZA quad-vector	FEAT_SME2p1
101	1xx	UNALLOCATED	-
11x	01x	UNALLOCATED	-
110	00x	ZERO (quad-vector) — Two ZA quad-vectors	FEAT_SME2p1
111	00x	ZERO (quad-vector) — Four ZA quad-vectors	FEAT_SME2p1

SME2 zero lookup table

The encodings in this section are decoded from SME encodings.

31													18	17		4	3		0
1	1	0	0	0	0	0	0	0	1	0	0	1	0	op0			opc		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	opc	Instruction Details	Condition
00000000000000	0001	ZERO (table)	FEAT_SME2
00000000000000	!= 0001	UNALLOCATED	-
!= 00000000000000	xxxx	UNALLOCATED	-

SME2 Move Lookup Table

SME2 Move Lookup Table

The encodings in this section are decoded from SME encodings.

31													18	17	16	15	14	13	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	op0	op1		op2		

op0	op1	op2	Instruction details
0	00	x	SME2 move from lookup table
0	10	x	UNALLOCATED
1	00	x	SME2 move into lookup table
1	10	0	SME2 move vector to lookup table
1	10	1	UNALLOCATED
x	x1	x	UNALLOCATED

SME2 move from lookup table

The encodings in this section are decoded from SME2 Move Lookup Table.

31													18	17	16	15	14	12	11	5	4	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	imm3		opc		Rt	

opc	Instruction Details	Condition
0011111	MOVT (table to scalar)	FEAT_SME2
!= 0011111	UNALLOCATED	-

SME2 move into lookup table

The encodings in this section are decoded from SME2 Move Lookup Table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31													18	17	16	15	14	12	11	5	4	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	imm3		opc		Rt	

opc	Instruction Details	Condition
0011111	MOVT (scalar to table)	FEAT_SME2
!= 0011111	UNALLOCATED	-

SME2 move vector to lookup table

The encodings in this section are decoded from SME2 Move Lookup Table.

31													18	17	16	15	14	13	12	11	5	4	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	0	off2		opc		Zt	

opc	Instruction Details	Condition
0011111	MOVT (vector to table)	FEAT_SME_LUTv2
!= 0011111	UNALLOCATED	-

SME2 Expand Lookup Table (Non-contiguous)

SME2 Expand Lookup Table (Non-contiguous)

The encodings in this section are decoded from SME encodings.

31												19	18	16	15	14	13	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	op0		op1				op2		op3			

op0	op1	op2	op3	Instruction details
011	00	0	00	SME2 lookup table two source registers expand to four non-contiguous destination registers
011	00	1	00	UNALLOCATED
xxx	10	x	00	SME2 lookup table expand four non-contiguous registers
xxx	x0	x	01	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details
xxx	x1	x	0x	SME2 lookup table expand two non-contiguous registers
xxx	xx	x	1x	UNALLOCATED
!=011	00	x	00	UNALLOCATED

SME2 lookup table two source registers expand to four non-contiguous destination registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Non-contiguous).

31												19	18		16	15	14	13	12	11	10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	size		opc		Zn			0	D	0	0	Zd	

opc	Instruction Details	Condition
00	LUTI4 (four registers, 8-bit)	FEAT_SME2p1 && FEAT_SME_LUTv2
!=00	UNALLOCATED	-

SME2 lookup table expand four non-contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Non-contiguous).

31												19	18		16	15	14	13	12	11	10	9		5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1		opc		1	0	size		opc2		Zn			D	0	0	Zd	

opc	opc2	Instruction Details	Condition
xxx	!=00	UNALLOCATED	-
00x	00	UNALLOCATED	-
01x	00	LUTI4 (four registers, 16-bit and 32-bit)	FEAT_SME2p1
1xx	00	LUTI2 (four registers)	FEAT_SME2p1

SME2 lookup table expand two non-contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Non-contiguous).

31												19	18		15	14	13	12	11	10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	0	0	1	1		opc		1		size		opc2		Zn		D	0		Zd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	Instruction Details	Condition
xxxx	!= 00	UNALLOCATED	-
00xx	00	UNALLOCATED	-
01xx	00	LUTI4 (two registers)	FEAT_SME2p1
1xxx	00	LUTI2 (two registers)	FEAT_SME2p1

SME2 Expand Lookup Table (Contiguous)

SME2 Expand Lookup Table (Contiguous)

The encodings in this section are decoded from SME encodings.

31	23	22	21	19	18	14	13	6	5	4	2	1	0
1 1 0 0 0 0 0 0 1		op0	0 0 1		op1				op2			op3	

op0	op1	op2	op3	Instruction details
0	00x00	x	00	UNALLOCATED
0	01000	x	00	UNALLOCATED
0	01100	0	00	SME2 lookup table two source registers expand to four contiguous destination registers
0	01100	1	00	UNALLOCATED
0	1xx00	x	00	UNALLOCATED
0	xxx10	x	00	SME2 lookup table expand four contiguous registers
0	xxxx0	x	10	UNALLOCATED
0	xxxx1	x	x0	SME2 lookup table expand two contiguous registers
0	xxxxx	x	x1	UNALLOCATED
1	xxxxx	x	xx	SME2 lookup table expand one register

SME2 lookup table two source registers expand to four contiguous destination registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31	23	22	21	19	18	14	13	12	11	10	9	6	5	4	2	1	0
1 1 0 0 0 0 0 0 1		0	0 0 1		0 1 1 0 0		size		opc		Zn		0	Zd		0	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
00	LUTI4 (four registers, 8-bit)	FEAT_SME_LUTv2
!= 00	UNALLOCATED	-

SME2 lookup table expand four contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31								23	22	21		19	18		16	15	14	13		12	11		10	9			5	4			2	1		0
1	1	0	0	0	0	0	0	1	0	0	0	1	opc			1	0	size			opc2			Zn				Zd				0		0

opc	opc2	Instruction Details	Condition
xxx	!= 00	UNALLOCATED	-
00x	00	UNALLOCATED	-
01x	00	LUTI4 (four registers, 16-bit and 32-bit)	FEAT_SME2
1xx	00	LUTI2 (four registers)	FEAT_SME2

SME2 lookup table expand two contiguous registers

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31								23	22	21		19	18		15	14	13		12	11		10	9			5	4			1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	opc			1	size			opc2			Zn				Zd				0

opc	opc2	Instruction Details	Condition
xxxx	!= 00	UNALLOCATED	-
00xx	00	UNALLOCATED	-
01xx	00	LUTI4 (two registers)	FEAT_SME2
1xxx	00	LUTI2 (two registers)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 lookup table expand one register

The encodings in this section are decoded from SME2 Expand Lookup Table (Contiguous).

31	23	22	21	19	18	14	13	12	11	10	9	5	4	0
1 1 0 0 0 0 0 0 1		1	0 0 1		opc		size		opc2		Zn		Zd	

opc	opc2	Instruction Details	Condition
xxxxx	!=00	UNALLOCATED	-
00xxx	00	UNALLOCATED	-
01xxx	00	LUTI4 (single)	FEAT_SME2
1xxxx	00	LUTI2 (single)	FEAT_SME2

SME Move into Array

SME Move into Array

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	19	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1 1 0 0 0 0 0	0	op0		0 0	0	op1	0	op2				op3				op4		0	op5		

op0	op1	op2	op3	op4	op5	Instruction details	Condition
00	1	00	010	x0	0	MOVA (vector to array, two registers)	FEAT_SME2
00	1	00	011	00	0	MOVA (vector to array, four registers)	FEAT_SME2
00	1	00	011	10	0	UNALLOCATED	-
00	1	01	01x	x0	0	UNALLOCATED	-
xx	0	xx	xxx	xx	x	SME move vector to array	-
xx	1	0x	000	x0	0	SME2 move vector to tile, two registers	-
xx	1	0x	001	00	0	SME2 move vector to tile, four registers	-
xx	1	0x	001	10	0	UNALLOCATED	-
xx	1	0x	0xx	x0	1	UNALLOCATED	-
xx	1	0x	0xx	x1	x	UNALLOCATED	-
xx	1	0x	1xx	xx	x	UNALLOCATED	-
xx	1	1x	xxx	xx	x	UNALLOCATED	-
!=00	1	0x	01x	x0	0	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME move vector to array

The encodings in this section are decoded from SME Move into Array.

31–24	23–22	21–19	18	17	16	15	14–13	12–10	9–5	4	3–0
1 1 0 0 0 0 0 0	size	0 0 0	0	0	Q	V	Rs	Pg	Zn	0	opc

size	Q	Instruction Details	Condition
00	0	MOVA (vector to tile, single) — 8-bit	FEAT_SME
01	0	MOVA (vector to tile, single) — 16-bit	FEAT_SME
10	0	MOVA (vector to tile, single) — 32-bit	FEAT_SME
11	0	MOVA (vector to tile, single) — 64-bit	FEAT_SME
11	1	MOVA (vector to tile, single) — 128-bit	FEAT_SME
!= 11	1	UNALLOCATED	-

SME2 move vector to tile, two registers

The encodings in this section are decoded from SME Move into Array.

31–24	23–22	21–19	18	17–16	15	14–13	12–10	9–6	5–4	3	2–0
1 1 0 0 0 0 0 0	size	0 0 0	1	0 0	V	Rs	0 0 0	Zn	0 0	0	opc

size	Instruction Details	Condition
00	MOVA (vector to tile, two registers) — 8-bit	FEAT_SME2
01	MOVA (vector to tile, two registers) — 16-bit	FEAT_SME2
10	MOVA (vector to tile, two registers) — 32-bit	FEAT_SME2
11	MOVA (vector to tile, two registers) — 64-bit	FEAT_SME2

SME2 move vector to tile, four registers

The encodings in this section are decoded from SME Move into Array.

31–24	23–22	21–19	18	17–16	15	14–13	12–10	9–7	6–5	4	3	2–0
1 1 0 0 0 0 0 0	size	0 0 0	1	0 0	V	Rs	0 0 1	Zn	0 0	0	0	opc

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Condition
00	0xx	MOVA (vector to tile, four registers) — 8-bit	FEAT_SME2
01	0xx	MOVA (vector to tile, four registers) — 16-bit	FEAT_SME2
10	0xx	MOVA (vector to tile, four registers) — 32-bit	FEAT_SME2
11	xxx	MOVA (vector to tile, four registers) — 64-bit	FEAT_SME2
!= 11	1xx	UNALLOCATED	-

SME Move from Array

SME Move from Array

The encodings in this section are decoded from SME encodings.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7					2	1	0
1	1	0	0	0	0	0	0	op0		0	0	0		op1	1	op2				op3				op4								op5	

op0	op1	op2	op3	op4	op5	Instruction details	Condition
00	1	00	010	00	x0	MOVA (array to vector, two registers)	FEAT_SME2
00	1	00	010	10	x0	MOVAZ (array to vector, two registers)	FEAT_SME2p1
00	1	00	011	00	00	MOVA (array to vector, four registers)	FEAT_SME2
00	1	00	011	10	00	MOVAZ (array to vector, four registers)	FEAT_SME2p1
00	1	00	011	x0	10	UNALLOCATED	-
00	1	01	01x	x0	x0	UNALLOCATED	-
xx	0	xx	000	1x	xx	SME zeroing move array to vector	-
xx	0	xx	xxx	0x	xx	SME move array to vector	-
xx	0	xx	!= 000	1x	xx	UNALLOCATED	-
xx	1	0x	000	00	x0	SME2 move tile to vector, two registers	-
xx	1	0x	000	10	x0	SME2 zeroing move tile to vector, two registers	-
xx	1	0x	001	00	00	SME2 move tile to vector, four registers	-
xx	1	0x	001	10	00	SME2 zeroing move tile to vector, four registers	-
xx	1	0x	001	x0	10	UNALLOCATED	-
xx	1	0x	0xx	x0	x1	UNALLOCATED	-
xx	1	0x	0xx	x1	xx	UNALLOCATED	-
xx	1	0x	1xx	xx	xx	UNALLOCATED	-
xx	1	1x	xxx	xx	xx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	op5	Instruction details	Condition
!=00	1	0x	01x	x0	x0	UNALLOCATED	-

SME zeroing move array to vector

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4				0
1	1	0	0	0	0	0	0	size		0	0	0		0	1	Q	V	Rs		0	0	0		1	opc				Zd				

size	Q	Instruction Details	Condition
00	0	MOVAZ (tile to vector, single) — 8-bit	FEAT_SME2p1
01	0	MOVAZ (tile to vector, single) — 16-bit	FEAT_SME2p1
10	0	MOVAZ (tile to vector, single) — 32-bit	FEAT_SME2p1
11	0	MOVAZ (tile to vector, single) — 64-bit	FEAT_SME2p1
11	1	MOVAZ (tile to vector, single) — 128-bit	FEAT_SME2p1
!=11	1	UNALLOCATED	-

SME move array to vector

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4				0
1	1	0	0	0	0	0	0	size		0	0	0		0	1	Q	V	Rs		Pg				0	opc				Zd				

size	Q	Instruction Details	Condition
00	0	MOVA (tile to vector, single) — 8-bit	FEAT_SME
01	0	MOVA (tile to vector, single) — 16-bit	FEAT_SME
10	0	MOVA (tile to vector, single) — 32-bit	FEAT_SME
11	0	MOVA (tile to vector, single) — 64-bit	FEAT_SME
11	1	MOVA (tile to vector, single) — 128-bit	FEAT_SME
!=11	1	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 move tile to vector, two registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	0	size		0	0	0		1	1	0	V	Rs		0	0	0		0	0	opc				Zd			0

size	Instruction Details	Condition
00	MOVA (tile to vector, two registers) — 8-bit	FEAT_SME2
01	MOVA (tile to vector, two registers) — 16-bit	FEAT_SME2
10	MOVA (tile to vector, two registers) — 32-bit	FEAT_SME2
11	MOVA (tile to vector, two registers) — 64-bit	FEAT_SME2

SME2 zeroing move tile to vector, two registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	0	size		0	0	0		1	1	0	V	Rs		0	0	0		1	0	opc				Zd			0

size	Instruction Details	Condition
00	MOVAZ (tile to vector, two registers) — 8-bit	FEAT_SME2p1
01	MOVAZ (tile to vector, two registers) — 16-bit	FEAT_SME2p1
10	MOVAZ (tile to vector, two registers) — 32-bit	FEAT_SME2p1
11	MOVAZ (tile to vector, two registers) — 64-bit	FEAT_SME2p1

SME2 move tile to vector, four registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	0	size		0	0	0		1	1	0	V	Rs		0	0	1		0	0	opc				Zd			0	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Condition
00	0xx	MOVA (tile to vector, four registers) — 8-bit	FEAT_SME2
01	0xx	MOVA (tile to vector, four registers) — 16-bit	FEAT_SME2
10	0xx	MOVA (tile to vector, four registers) — 32-bit	FEAT_SME2
11	xxx	MOVA (tile to vector, four registers) — 64-bit	FEAT_SME2
!= 11	1xx	UNALLOCATED	-

SME2 zeroing move tile to vector, four registers

The encodings in this section are decoded from SME Move from Array.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	0	size		0	0	0	1	1	0	V	Rs		0	0	1	1	0	opc			Zd			0	0

size	opc	Instruction Details	Condition
00	0xx	MOVAZ (tile to vector, four registers) — 8-bit	FEAT_SME2p1
01	0xx	MOVAZ (tile to vector, four registers) — 16-bit	FEAT_SME2p1
10	0xx	MOVAZ (tile to vector, four registers) — 32-bit	FEAT_SME2p1
11	xxx	MOVAZ (tile to vector, four registers) — 64-bit	FEAT_SME2p1
!= 11	1xx	UNALLOCATED	-

SME Add Vector to Array

SME Add Vector to Array

The encodings in this section are decoded from SME encodings.

31							24	23	22	21		19	18	17	16		5	4	3	2		0
1	1	0	0	0	0	0	0	op0		0	1	0	op1					op2	0			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0	xx	x	UNALLOCATED
1	00	0	SME add vector to array
1	00	1	UNALLOCATED
1	!=00	x	UNALLOCATED

SME add vector to array

The encodings in this section are decoded from SME Add Vector to Array.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	op	0	1	0	0	0	V	Pm			Pn			Zn			0	0	opc2		

op	V	opc2	Instruction Details	Condition
0	x	1xx	UNALLOCATED	-
0	0	0xx	ADDHA — 32-bit	FEAT_SME
0	1	0xx	ADDVA — 32-bit	FEAT_SME
1	0	xxx	ADDHA — 64-bit	FEAT_SME_I16I64
1	1	xxx	ADDVA — 64-bit	FEAT_SME_I16I64

SME2 Multi-vector - Multiple and Single Array Vectors (Two registers)

SME2 Multi-vector - Multiple and Single Array Vectors (Two registers)

The encodings in this section are decoded from SME encodings.

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4			2	1	0
1	1	0	0	0	0	0	1	0	op0	1	0					0			op1							op2				op3	

op0	op1	op2	op3	Instruction details	Condition
0	000	000	1	FMLALL (multiple and single vector) — Two ZA quad-vectors	FEAT_SME_F8F32
0	000	!=000	1	UNALLOCATED	-
0	010	xxx	x	SME2 single-multi long FMA two sources	-
0	011	xxx	x	SME2 multiple and single vector long FMA one source	-
0	100	xxx	x	SME2 single-multi FP dot product two registers	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
0	101	x1x	x	SME2 single-multi mixed dot product two registers	-
1	000	xxx	1	UNALLOCATED	-
1	010	xxx	x	SME2 single-multi long MLA two sources	-
1	011	xxx	x	SME2 multiple and single vector long MLA one source	-
1	100	xxx	x	UNALLOCATED	-
1	101	x1x	x	SME2 single-multi two-way dot product two registers	-
x	000	xxx	0	SME2 single-multi long long MLA two sources	-
x	001	xxx	x	SME2 multiple and single vector long long FMA one source	-
x	101	x0x	x	SME2 single-multi four-way dot product two registers	-
x	110	0xx	x	SME2 single-multi ternary FP two registers	-
x	110	1xx	x	SME2 single-multi ternary int two registers	-
x	111	0xx	x	SME2 single-multi ternary FP16 two registers	-
x	111	1xx	x	UNALLOCATED	-

SME2 single-multi long FMA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

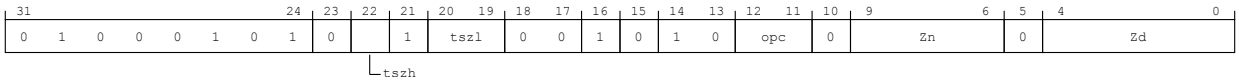

op	S	o2	Instruction Details	Condition
0	0	0	FMLAL (multiple and single vector, FP16 to FP32)	FEAT_SME2
0	0	1	FMLAL (multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
0	1	0	FMLSL (multiple and single vector)	FEAT_SME2
0	1	1	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	BFMLAL (multiple and single vector)	FEAT_SME2
1	1	0	BFMLSL (multiple and single vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple and single vector long FMA one source

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	1	1	Zn				op	S	off3			

op	S	Instruction Details	Condition
0	0	FMLAL (multiple and single vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and single vector)	FEAT_SME2
1	0	BFMLAL (multiple and single vector)	FEAT_SME2
1	1	BFMLSL (multiple and single vector)	FEAT_SME2

SME2 single-multi FP dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4		3	2			0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	0	0	Zn				opc			off3		

opc	Instruction Details	Condition
00	FDOT (2-way, multiple and single vector, FP16 to FP32)	FEAT_SME2
01	FDOT (2-way, multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
10	BFDOT (multiple and single vector)	FEAT_SME2
11	FDOT (4-way, multiple and single vector)	FEAT_SME_F8F32

SME2 single-multi mixed dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20				16	15		14	13	12	11	10	9			6	5	4				0
0	1	0	0	0	1	0	1	1	op0	1	opc					0		0	op1	U	R	0	Zn				0	Zd			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Condition
0	USDOT (4-way, multiple and single vector)	FEAT_SME2
1	SUDOT (4-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm				0	Rv		0	1	0	Zn				U	S	op	off2	

U	S	op	Instruction Details	Condition
x	x	1	UNALLOCATED	-
0	0	0	SMLAL (multiple and single vector)	FEAT_SME2
0	1	0	SMLSL (multiple and single vector)	FEAT_SME2
1	0	0	UMLAL (multiple and single vector)	FEAT_SME2
1	1	0	UMLSL (multiple and single vector)	FEAT_SME2

SME2 multiple and single vector long MLA one source

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	1	0	Zm				0	Rv		0	1	1	Zn				U	S	off3		

U	S	Instruction Details	Condition
0	0	SMLAL (multiple and single vector)	FEAT_SME2
0	1	SMLSL (multiple and single vector)	FEAT_SME2
1	0	UMLAL (multiple and single vector)	FEAT_SME2
1	1	UMLSL (multiple and single vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi two-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	1	1	0	Zm				0	Rv		1	0	1	Zn				U	1	off3			

U	Instruction Details	Condition
0	SDOT (2-way, multiple and single vector)	FEAT_SME2
1	UDOT (2-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		0	0	0	Zn				U	S	op	0	o1

sz	U	S	op	Instruction Details	Condition
x	0	0	0	SMLALL (multiple and single vector)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
x	0	1	0	SMLSLL (multiple and single vector)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
x	1	0	0	UMLALL (multiple and single vector)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
x	1	1	0	UMLSLL (multiple and single vector)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
0	x	1	1	UNALLOCATED	-
0	0	0	1	USMLALL (multiple and single vector)	FEAT_SME2
0	1	0	1	SUMLALL (multiple and single vector)	FEAT_SME2
1	x	x	1	UNALLOCATED	-

SME2 multiple and single vector long long FMA one source

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		0	0	1	Zn				U	S	op	off2	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Condition
x	0	0	0	SMLALL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	0	1	0	SMLSLL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	0	0	UMLALL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	1	0	UMLSLL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
0	0	0	1	USMLALL (multiple and single vector)	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	x	x	1	UNALLOCATED	-

SME2 single-multi four-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31							24	23 22	21	20		18	17	16	15				11	10	9				5	4	0
0	1	0	0	0	1	0	1	size	1	0	0	0	0	0	1	1	1	0	0	op	0	0	0	0	0	Zdn	

U	Instruction Details	Condition
0	SDOT (4-way, multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
1	UDOT (4-way, multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

SME2 single-multi ternary FP two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31								23	22	21	20	19			16	15	14	13	12		10	9				5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		1	1	0	Zn					0	S	off3		

S	Instruction Details	Condition
0	FMLA (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64)
1	FMLS (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi ternary int two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 1 0 0 0 0 0 1 0	sz	1 0	Zm	0	Rv	1 1 0	Zn	1	S	off3

S	Instruction Details	Condition
0	ADD (to array, multiple and single vector)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
1	SUB (to array, multiple and single vector)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)

SME2 single-multi ternary FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Two registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
1 1 0 0 0 0 0 1 0	sz	1 0	Zm	0	Rv	1 1 1	Zn	0	S	off3

sz	S	Instruction Details	Condition
0	0	FMLA (multiple and single vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and single vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and single vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and single vector)	FEAT_SME_B16B16

SME2 Multi-vector - Multiple and Single Array Vectors (Four registers)

SME2 Multi-vector - Multiple and Single Array Vectors (Four registers)

The encodings in this section are decoded from SME encodings.

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4–2	1	0
1 1 0 0 0 0 0 1 0	op0	1 1		0		op1		op2	op3	

op0	op1	op2	op3	Instruction details	Condition
0	000	000	1	FMLALL (multiple and single vector) — Four ZA quad-vectors	FEAT_SME_F8F32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
0	000	!=000	1	UNALLOCATED	-
0	001	000	x	FMLALL (multiple and single vector) — One ZA quad-vector	FEAT_SME_F8F32
0	001	001	x	UNALLOCATED	-
0	010	xxx	x	SME2 single-multi long FMA four sources	-
0	011	00x	x	FMLAL (multiple and single vector, FP8 to FP16) — One ZA double-vector	FEAT_SME_F8F16
0	0x1	!=00x	x	UNALLOCATED	-
0	100	xxx	x	SME2 single-multi FP dot product four registers	-
0	101	x1x	x	SME2 single-multi mixed dot product four registers	-
1	000	xxx	1	UNALLOCATED	-
1	010	xxx	x	SME2 single-multi long MLA four sources	-
1	0x1	xxx	x	UNALLOCATED	-
1	100	xxx	x	UNALLOCATED	-
1	101	x1x	x	SME2 single-multi two-way dot product four registers	-
x	000	xxx	0	SME2 single-multi long long MLA four sources	-
x	101	x0x	x	SME2 single-multi four-way dot product four registers	-
x	110	0xx	x	SME2 single-multi ternary FP four registers	-
x	110	1xx	x	SME2 single-multi ternary int four registers	-
x	111	0xx	x	SME2 single-multi ternary FP16 four registers	-
x	111	1xx	x	UNALLOCATED	-

SME2 single-multi long FMA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				op	S	o2	off2	

op	S	o2	Instruction Details	Condition
0	0	0	FMLAL (multiple and single vector, FP16 to FP32)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	o2	Instruction Details	Condition
0	0	1	FMLAL (multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
0	1	0	FMLSL (multiple and single vector)	FEAT_SME2
0	1	1	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	BFMLAL (multiple and single vector)	FEAT_SME2
1	1	0	BFMLSL (multiple and single vector)	FEAT_SME2

SME2 single-multi FP dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	0	Zn				opc		off3			

opc	Instruction Details	Condition
00	FDOT (2-way, multiple and single vector, FP16 to FP32)	FEAT_SME2
01	FDOT (2-way, multiple and single vector, FP8 to FP16)	FEAT_SME_F8F16
10	BFDOT (multiple and single vector)	FEAT_SME2
11	FDOT (4-way, multiple and single vector)	FEAT_SME_F8F32

SME2 single-multi mixed dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	1	Zn				U	1	off3			

U	Instruction Details	Condition
0	USDOT (4-way, multiple and single vector)	FEAT_SME2
1	SUDOT (4-way, multiple and single vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		0	1	0	Zn				U	S	op	off2	

U	S	op	Instruction Details	Condition
x	x	1	UNALLOCATED	-
0	0	0	SMLAL (multiple and single vector)	FEAT_SME2
0	1	0	SMLSL (multiple and single vector)	FEAT_SME2
1	0	0	UMLAL (multiple and single vector)	FEAT_SME2
1	1	0	UMLSL (multiple and single vector)	FEAT_SME2

SME2 single-multi two-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		1	0	1	Zn				U	1	off3		

U	Instruction Details	Condition
0	SDOT (2-way, multiple and single vector)	FEAT_SME2
1	UDOT (2-way, multiple and single vector)	FEAT_SME2

SME2 single-multi long long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		0	0	0	Zn				U	S	op	0	o1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Condition
x	0	0	0	SMLALL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	0	1	0	SMLSLL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	0	0	UMLALL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	1	0	UMLSLL (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
0	x	1	1	UNALLOCATED	-
0	0	0	1	USMLALL (multiple and single vector)	FEAT_SME2
0	1	0	1	SUMLALL (multiple and single vector)	FEAT_SME2
1	x	x	1	UNALLOCATED	-

SME2 single-multi four-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14		13	12			10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv			1	0		1	Zn				U	0	off3			

U	Instruction Details	Condition
0	SDOT (4-way, multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
1	UDOT (4-way, multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

SME2 single-multi ternary FP four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31								23	22	21	20	19			16	15	14		13	12			10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv			1	1		0	Zn				0	S	off3			

S	Instruction Details	Condition
0	FMLA (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64)
1	FMLS (multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi ternary int four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31–24	23	22–21	20–16	15–14	13	12	11–10	9–5	4–0
01100100	1	01	Zm	10	0	T	10	Zn	Zda

S	Instruction Details	Condition
0	ADD (to array, multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
1	SUB (to array, multiple and single vector)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

SME2 single-multi ternary FP16 four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single Array Vectors (Four registers).

31–23	22	21–20	19–16	15	14–13	12–10	9–5	4	3	2–0
110000001	sz	11	Zm	0	Rv	111	Zn	0	S	off3

sz	S	Instruction Details	Condition
0	0	FMLA (multiple and single vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and single vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and single vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and single vector)	FEAT_SME_B16B16

SME2 Multi-vector - Multiple Array Vectors (Two registers)

SME2 Multi-vector - Multiple Array Vectors (Two registers)

The encodings in this section are decoded from SME encodings.

31–23	22	21	20–19	18–17	16–15	14–13	12–10	9–6	5	4–2	1	0
110000011	op0	1	op1	op2	00		op3		op4	op5	op6	

op0	op1	op2	op3	op4	op5	op6	Instruction details	Condition
0	xx	xx	000	1	000	0	FMLALL (multiple vectors) — Two ZA quad-vectors	FEAT_SME_F8F32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	op5	op6	Instruction details	Condition
0	xx	xx	000	1	001	0	UNALLOCATED	-
0	xx	xx	000	1	10x	0	UNALLOCATED	-
0	xx	xx	000	1	x0x	1	UNALLOCATED	-
0	xx	xx	010	0	xx0	x	SME2 multiple vectors long FMA two sources	-
0	xx	xx	010	1	000	x	FMLAL (multiple vectors, FP8 to FP16) — Two ZA double-vectors	FEAT_SME_F8F16
0	xx	xx	010	1	100	x	UNALLOCATED	-
0	xx	xx	010	1	x01	x	UNALLOCATED	-
0	xx	xx	100	x	x0x	x	SME2 multiple vectors FP dot product two registers	-
0	xx	xx	101	0	01x	x	USDOT (4-way, multiple vectors) — Two ZA single-vectors	FEAT_SME2
0	xx	xx	101	0	11x	x	UNALLOCATED	-
0	xx	xx	110	1	x0x	x	UNALLOCATED	-
0	xx	xx	xx0	1	x1x	x	UNALLOCATED	-
0	xx	xx	!= xx0	1	xxx	x	UNALLOCATED	-
1	xx	xx	010	0	xx0	x	SME2 multiple vectors long MLA two sources	-
1	xx	xx	100	0	x0x	x	UNALLOCATED	-
1	xx	xx	101	0	x1x	x	SME2 multiple vectors two-way dot product two registers	-
1	xx	xx	xxx	1	xxx	x	UNALLOCATED	-
x	00	00	111	0	0xx	x	SME2 multiple vectors binary FP two registers	-
x	00	00	111	0	1xx	x	SME2 multiple vectors binary int two registers	-
x	00	10	111	0	0xx	x	SME2 multiple vectors binary FP16 two registers	-
x	00	10	111	0	1xx	x	UNALLOCATED	-
x	00	x1	111	0	xxx	x	UNALLOCATED	-
x	xx	xx	000	0	xxx	0	SME2 multiple vectors long long MLA two sources	-
x	xx	xx	000	0	xxx	1	UNALLOCATED	-
x	xx	xx	010	0	xx1	x	UNALLOCATED	-
x	xx	xx	0x1	0	xxx	x	UNALLOCATED	-
x	xx	xx	100	0	x1x	x	SME2 multiple vectors ternary FP16 two registers	-
x	xx	xx	101	0	x0x	x	SME2 multiple vectors four-way dot product two registers	-
x	xx	xx	110	0	0xx	x	SME2 multiple vectors ternary FP two registers	-
x	xx	xx	110	0	1xx	x	SME2 multiple vectors ternary int two registers	-
x	!= 00	xx	111	0	xxx	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors long FMA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2	1		0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv			0	1		0	Zn				0	op	S	0	off2		

op	S	Instruction Details	Condition
0	0	FMLAL (multiple vectors, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple vectors)	FEAT_SME2
1	0	BFMLAL (multiple vectors)	FEAT_SME2
1	1	BFMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors FP dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31							24	23		22	21			19	18			16	15	14		13	12			10	9			5	4			0
0	1	1	0	0	1	0	0	opc			0	1		1	0	1		0	1	opc2			Pg				Zn				Zd			

opc	Instruction Details	Condition
00	FDOT (2-way, multiple vectors, FP16 to FP32)	FEAT_SME2
01	BFDOT (multiple vectors)	FEAT_SME2
10	FDOT (2-way, multiple vectors, FP8 to FP16)	FEAT_SME_F8F16
11	FDOT (4-way, multiple vectors)	FEAT_SME_F8F32

SME2 multiple vectors long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2	1		0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv			0	1		0	Zn				0	U	S	0	off2		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Condition
0	0	SMLAL (multiple vectors)	FEAT_SME2
0	1	SMLSL (multiple vectors)	FEAT_SME2
1	0	UMLAL (multiple vectors)	FEAT_SME2
1	1	UMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors two-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv			1	0		1	Zn				0	U	1	off3			

U	Instruction Details	Condition
0	SDOT (2-way, multiple vectors)	FEAT_SME2
1	UDOT (2-way, multiple vectors)	FEAT_SME2

SME2 multiple vectors binary FP two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								24	23		22	21					17	16	15	14	13	12			10	9			5	4			0
0	1	1	0	0	1	0	0	opc				0	1	1	1	0		o2	1	o3	U	Pg				Zn				Zd			

S	Instruction Details	Condition
0	FADD	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)
1	FSUB	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors binary int two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31							23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv	1	1	1	Zm			0	1	S	off3		

S	Instruction Details	Condition
0	ADD (to array, array and multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
1	SUB (to array, array and multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

SME2 multiple vectors binary FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31							23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	1	0	0	0	Rv	1	1	1	Zm			0	0	S	off3		

sz	S	Instruction Details	Condition
0	0	FADD	FEAT_SME_F16F16 || FEAT_SME_F8F16
0	1	FSUB	FEAT_SME_F16F16 || FEAT_SME_F8F16
1	0	BFADD	FEAT_SME_B16B16
1	1	BFSUB	FEAT_SME_B16B16

SME2 multiple vectors long long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31							23	22	21	20			17	16	15	14	13	12		10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	0	Rv		0	0	0	Zn			0	U	S	op	0	o1

sz	U	S	op	Instruction Details	Condition
x	0	0	0	SMLALL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	0	1	0	SMLSLL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	U	S	op	Instruction Details	Condition
x	1	0	0	UMLALL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	1	0	UMLSLL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
0	0	0	1	USMLALL (multiple vectors)	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	x	x	1	UNALLOCATED	-

SME2 multiple vectors ternary FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv			1	0		0	Zn				0	S	1	off3			

sz	S	Instruction Details	Condition
0	0	FMLA (multiple vectors)	FEAT_SME_F16F16
0	1	FMLS (multiple vectors)	FEAT_SME_F16F16
1	0	BFMLA (multiple vectors)	FEAT_SME_B16B16
1	1	BFMLS (multiple vectors)	FEAT_SME_B16B16

SME2 multiple vectors four-way dot product two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14		13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv			1	0		1	Zn				0	U	0	off3			

U	Instruction Details	Condition
0	SDOT (4-way, multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
1	UDOT (4-way, multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors ternary FP two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	1	1		0			Zn		0	0	S		off3		

S	Instruction Details	Condition
0	FMLA (multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)
1	FMLS (multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)

SME2 multiple vectors ternary int two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Two registers).

31								23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	1	1		0			Zn		0	1	S		off3		

S	Instruction Details	Condition
0	ADD (to array, multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
1	SUB (to array, multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)

SME2 Multi-vector - Multiple Array Vectors (Four registers)

SME2 Multi-vector - Multiple Array Vectors (Four registers)

The encodings in this section are decoded from SME encodings.

31								23	22	21	20		19	18		17	16	15	14		13	12			10	9			7	6		5	4			2	1	0
1	1	0	0	0	0	0	1	1	op0	1	op1			op2			1	0				op3								op4			op5			op6	

op0	op1	op2	op3	op4	op5	op6	Instruction details	Condition
0	xx	x0	000	01	000	0	FMLALL (multiple vectors) — Four ZA quad-vectors	FEAT_SME_F8F32
0	xx	x0	000	01	001	0	UNALLOCATED	-
0	xx	x0	000	01	10x	0	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	op5	op6	Instruction details	Condition
0	xx	x0	000	01	x0x	1	UNALLOCATED	-
0	xx	x0	010	00	xx0	x	SME2 multiple vectors long FMA four sources	-
0	xx	x0	010	01	000	x	FMLAL (multiple vectors, FP8 to FP16) — Four ZA double-vectors	FEAT_SME_F8F16
0	xx	x0	010	01	100	x	UNALLOCATED	-
0	xx	x0	010	01	x01	x	UNALLOCATED	-
0	xx	x0	100	0x	x0x	x	SME2 multiple vectors FP dot product four registers	-
0	xx	x0	101	00	01x	x	USDOT (4-way, multiple vectors) — Four ZA single-vectors	FEAT_SME2
0	xx	x0	101	00	11x	x	UNALLOCATED	-
0	xx	x0	110	01	x0x	x	UNALLOCATED	-
0	xx	x0	xx0	01	x1x	x	UNALLOCATED	-
0	xx	x0	!= xx0	01	xxx	x	UNALLOCATED	-
1	xx	x0	010	00	xx0	x	SME2 multiple vectors long MLA four sources	-
1	xx	x0	100	00	x0x	x	UNALLOCATED	-
1	xx	x0	101	00	x1x	x	SME2 multiple vectors two-way dot product four registers	-
1	xx	x0	xxx	01	xxx	x	UNALLOCATED	-
x	00	00	111	00	0xx	x	SME2 multiple vectors binary FP four registers	-
x	00	00	111	00	1xx	x	SME2 multiple vectors binary int four registers	-
x	00	10	111	00	0xx	x	SME2 multiple vectors binary FP16 four registers	-
x	00	10	111	00	1xx	x	UNALLOCATED	-
x	xx	x0	000	00	xxx	0	SME2 multiple vectors long long MLA four sources	-
x	xx	x0	000	00	xxx	1	UNALLOCATED	-
x	xx	x0	010	00	xx1	x	UNALLOCATED	-
x	xx	x0	0x1	00	xxx	x	UNALLOCATED	-
x	xx	x0	100	00	x1x	x	SME2 multiple vectors ternary FP16 four registers	-
x	xx	x0	101	00	x0x	x	SME2 multiple vectors four-way dot product four registers	-
x	xx	x0	110	00	0xx	x	SME2 multiple vectors ternary FP four registers	-
x	xx	x0	110	00	1xx	x	SME2 multiple vectors ternary int four registers	-
x	xx	x0	xxx	1x	xxx	x	UNALLOCATED	-
x	xx	x1	xxx	xx	xxx	x	UNALLOCATED	-
x	!= 00	x0	111	00	xxx	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors long FMA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	op	S	0	off2	

op	S	Instruction Details	Condition
0	0	FMLAL (multiple vectors, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple vectors)	FEAT_SME2
1	0	BFMLAL (multiple vectors)	FEAT_SME2
1	1	BFMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors FP dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	opc		0	off3		

opc	Instruction Details	Condition
00	FDOT (2-way, multiple vectors, FP16 to FP32)	FEAT_SME2
01	BFDOT (multiple vectors)	FEAT_SME2
10	FDOT (2-way, multiple vectors, FP8 to FP16)	FEAT_SME_F8F16
11	FDOT (4-way, multiple vectors)	FEAT_SME_F8F32

SME2 multiple vectors long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	U	S	0	off2	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Condition
0	0	SMLAL (multiple vectors)	FEAT_SME2
0	1	SMLSL (multiple vectors)	FEAT_SME2
1	0	UMLAL (multiple vectors)	FEAT_SME2
1	1	UMLSL (multiple vectors)	FEAT_SME2

SME2 multiple vectors two-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	1	0	Rv		1	0	1	Zn			0	0	U	1	off3		

U	Instruction Details	Condition
0	SDOT (2-way, multiple vectors)	FEAT_SME2
1	UDOT (2-way, multiple vectors)	FEAT_SME2

SME2 multiple vectors binary FP four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20	19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv		1	1	1	Zm			0	0	0	S	off3		

S	Instruction Details	Condition
0	FADD	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)
1	FSUB	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)

SME2 multiple vectors binary int four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								24	23	22	21					17	16	15			12	11	10	9				5	4				0
0	1	1	0	0	1	0	1		0	0	0	0	1	0	0		L	0	0	1	1		opc		Zn					Zd		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	Instruction Details	Condition
0	ADD (to array, array and multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
1	SUB (to array, array and multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)

SME2 multiple vectors binary FP16 four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20	19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	1	0	1	0	Rv		1	1	1	Zm			0	0	0	S	off3		

sz	S	Instruction Details	Condition
0	0	FADD	FEAT_SME_F16F16 || FEAT_SME_F8F16
0	1	FSUB	FEAT_SME_F16F16 || FEAT_SME_F8F16
1	0	BFADD	FEAT_SME_B16B16
1	1	BFSUB	FEAT_SME_B16B16

SME2 multiple vectors long long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31								23	22	21	20		18	17		15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	1	0	Rv		0	0	0	Zn			0	0	U	S	op	0	o1

sz	U	S	op	Instruction Details	Condition
x	0	0	0	SMLALL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	0	1	0	SMLSLL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	0	0	UMLALL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
x	1	1	0	UMLSLL (multiple vectors)	FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64)
0	0	0	1	USMLALL (multiple vectors)	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1	x	1	UNALLOCATED	-
1	x	x	1	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors ternary FP16 four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31–24	23–22	21–17	16	15–12	11–10	9–5	4–0
0 1 1 0 0 1 0 1	size	0 0 1 1 1	op	0 0 1 1	0 0	Zn	Zd

sz	S	Instruction Details	Condition
0	0	FMLA (multiple vectors)	FEAT_SME_F16F16
0	1	FMLS (multiple vectors)	FEAT_SME_F16F16
1	0	BFMLA (multiple vectors)	FEAT_SME_B16B16
1	1	BFMLS (multiple vectors)	FEAT_SME_B16B16

SME2 multiple vectors four-way dot product four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31–23	22	21	20–18	17–15	14–13	12–10	9–7	6–5	4	3	2–0
1 1 0 0 0 0 0 1 1	sz	1	Zm	0 1 0	Rv	1 0 1	Zn	0 0	U	0	off3

U	Instruction Details	Condition
0	SDOT (4-way, multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
1	UDOT (4-way, multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)

SME2 multiple vectors ternary FP four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31–23	22	21	20–18	17–15	14–13	12–10	9–7	6–5	4	3	2–0
1 1 0 0 0 0 0 1 1	sz	1	Zm	0 1 0	Rv	1 1 0	Zn	0 0	0	S	off3

S	Instruction Details	Condition
0	FMLA (multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)
1	FMLS (multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_F64F64)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multiple vectors ternary int four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Array Vectors (Four registers).

31–23	22	21	20–18	17–15	14–13	12–10	9–7	6–5	4	3	2–0
11000001	sz	1	Zm	010	Rv	110	Zn	00	1	S	off3

S	Instruction Details	Condition
0	ADD (to array, multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)
1	SUB (to array, multiple vectors)	FEAT_SME2 && (sz == ’0’ || FEAT_SME_I16I64)

SME2 Multi-vector - Indexed (One register)

SME2 Multi-vector - Indexed (One register)

The encodings in this section are decoded from SME encodings.

31–24	23–22	21–19	18	17–16	15–13	12–0
01100101		011	op0		001	

op0	op1	op2	Instruction details	Condition
00	x	xxx	SME2 multi-vec indexed long long MLA one source 32-bit	-
01	x	000	FMLALL (multiple and indexed vector) — One ZA quad-vector	FEAT_SME_F8F32
01	x	!=000	UNALLOCATED	-
10	0	xx0	SME2 multi-vec indexed long long MLA one source 64-bit	-
10	0	xx1	UNALLOCATED	-
10	1	xxx	SME2 multi-vec indexed long FMA one source	-
11	0	0xx	FMLAL (multiple and indexed vector, FP8 to FP16) — One ZA double-vector	FEAT_SME_F8F16
11	0	1xx	UNALLOCATED	-
11	1	xxx	SME2 multi-vec indexed long MLA one source	-

SME2 multi-vec indexed long long MLA one source 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm		i4h	Rv		i4l		Zn		U	S	op	off2	

U	S	op	Instruction Details	Condition
x	1	1	UNALLOCATED	-
0	0	0	SMLALL (multiple and indexed vector)	FEAT_SME2
0	0	1	USMLALL (multiple and indexed vector)	FEAT_SME2
0	1	0	SMLSLL (multiple and indexed vector)	FEAT_SME2
1	0	0	UMLALL (multiple and indexed vector)	FEAT_SME2
1	0	1	SUMLALL (multiple and indexed vector)	FEAT_SME2
1	1	0	UMLSLL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long long MLA one source 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm		i3h	Rv		0	i3l		Zn		U	S	0	off2	

U	S	Instruction Details	Condition
0	0	SMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	0	UMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long FMA one source

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm		i3h	Rv		1	i3l		Zn		op	S	off3	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	Instruction Details	Condition
0	0	FMLAL (multiple and indexed vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and indexed vector)	FEAT_SME2
1	0	BFMLAL (multiple and indexed vector)	FEAT_SME2
1	1	BFMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long MLA one source

The encodings in this section are decoded from SME2 Multi-vector - Indexed (One register).

31	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1 1 0 0 0 0 0 1		1	1	0	0	Zm		i3h	Rv		1	i3l		Zn		U	S	off3	

U	S	Instruction Details	Condition
0	0	SMLAL (multiple and indexed vector)	FEAT_SME2
0	1	SMLSL (multiple and indexed vector)	FEAT_SME2
1	0	UMLAL (multiple and indexed vector)	FEAT_SME2
1	1	UMLSL (multiple and indexed vector)	FEAT_SME2

SME2 Multi-vector - Indexed (Two registers)

SME2 Multi-vector - Indexed (Two registers)

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	19	16	15	14	13	12	11	10	6	5	4	3	2	0
1 1 0 0 0 0 0 1		op0		0	1			0			op1				op2	op3			

op0	op1	op2	op3	Instruction details	Condition
00	0x	x	xx	SME2 multi-vec indexed long long MLA two sources 32-bit	-
00	1x	x	xx	SME2 multi-vec ternary indexed two registers 16-bit	-
01	xx	x	xx	SME2 multi-vec ternary indexed two registers 32-bit	-
10	00	0	xx	SME2 multi-vec indexed long long MLA two sources 64-bit	-
10	01	0	xx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
10	0x	1	00	FMLALL (multiple and indexed vector) — Two ZA quad-vectors	FEAT_SME_F8F32
10	0x	1	!=00	UNALLOCATED	-
10	1x	0	xx	SME2 multi-vec indexed long FMA two sources	-
10	1x	1	0x	UNALLOCATED	-
10	1x	1	1x	FMLAL (multiple and indexed vector, FP8 to FP16) — Two ZA double-vectors	FEAT_SME_F8F16
11	00	0	xx	SME2 multi-vec ternary indexed two registers 64-bit	-
11	01	0	xx	SME2 multi-vec indexed FP8 two-way vertical dot product to single-precision two registers	-
11	1x	0	xx	SME2 multi-vec indexed long MLA two sources	-
11	xx	1	0x	SME2 multi-vec indexed FP8 two-way dot product to FP16 two registers	-
11	xx	1	1x	UNALLOCATED	-

SME2 multi-vec indexed long long MLA two sources 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm		0	Rv		0	i4h		Zn		op	U	S	i41		o1

op	U	S	Instruction Details	Condition
0	0	0	SMLALL (multiple and indexed vector)	FEAT_SME2
0	0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2
0	1	0	UMLALL (multiple and indexed vector)	FEAT_SME2
0	1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2
1	x	1	UNALLOCATED	-
1	0	0	USMLALL (multiple and indexed vector)	FEAT_SME2
1	1	0	SUMLALL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed two registers 16-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				op	S	i31	off3		

op	S	Instruction Details	Condition
0	0	FMLA (multiple and indexed vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and indexed vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and indexed vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and indexed vector)	FEAT_SME_B16B16

SME2 multi-vec ternary indexed two registers 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5		3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		op	i2		Zn				opc2			off3		

op	opc2	Instruction Details	Condition
0	000	FMLA (multiple and indexed vector)	FEAT_SME2
0	001	FVDOT (FP16 to FP32)	FEAT_SME2
0	010	FMLS (multiple and indexed vector)	FEAT_SME2
0	011	BFVDOT	FEAT_SME2
0	100	SVDOT (2-way)	FEAT_SME2
0	101	UNALLOCATED	-
0	110	UVDOT (2-way)	FEAT_SME2
0	111	FDOT (4-way, multiple and indexed vector)	FEAT_SME_F8F32
1	000	SDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	001	FDOT (2-way, multiple and indexed vector, FP16 to FP32)	FEAT_SME2
1	010	UDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	011	BFDOT (multiple and indexed vector)	FEAT_SME2
1	100	SDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	101	USDOT (4-way, multiple and indexed vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	opc2	Instruction Details	Condition
1	110	UDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	111	SUDOT (4-way, multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed long long MLA two sources 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19			16	15	14		13	12		11	10	9				6	5	4	3	2		1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				0	Rv			0		0	i3h	Zn					0	U	S	i31			o1

U	S	Instruction Details	Condition
0	0	SMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	0	UMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long FMA two sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23		22	21	20			18	17		16	15			13	12			10	9				5	4					0
0	1	1	0	0	1	0	1	opc			0	0	1	0		opc2			1	0		1	Pg				Zn					Zd					

op	S	Instruction Details	Condition
0	0	FMLAL (multiple and indexed vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and indexed vector)	FEAT_SME2
1	0	BFMLAL (multiple and indexed vector)	FEAT_SME2
1	1	BFMLSL (multiple and indexed vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec ternary indexed two registers 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31	23	21	19	15	14	12	10	9	5	4	2
1 1 0 0 0 0 0 1	1 1	0 1	Zm	0	Rv	0 0	i1	Zn	0	opc	off3

opc	Instruction Details	Condition
00	FMLA (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
01	SDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
10	FMLS (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
11	UDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed FP8 two-way vertical dot product to single-precision two registers

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31	23	21	19	15	14	12	10	9	5	4	3	2
1 1 0 0 0 0 0 1	1 1	0 1	Zm	0	Rv	0 1	i2h	Zn	0	T	i2l	off3

T	Instruction Details	Condition
0	FVDOTB	FEAT_SME_F8F32
1	FVDOTT	FEAT_SME_F8F32

SME2 multi-vec indexed long MLA two sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31	23	21	18	16	15	12	9	4
0 1 1 0 0 1 0 1	opc	0 1 0	opc2	U	1 0 1	Pg	Zn	Zd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	S	Instruction Details	Condition
0	0	SMLAL (multiple and indexed vector)	FEAT_SME2
0	1	SMLSL (multiple and indexed vector)	FEAT_SME2
1	0	UMLAL (multiple and indexed vector)	FEAT_SME2
1	1	UMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec indexed FP8 two-way dot product to FP16 two registers

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Two registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm		0	Rv		op	i3h		Zn		1	0	i31	off3	

op	Instruction Details	Condition
0	FDOT (2-way, multiple and indexed vector, FP8 to FP16)	FEAT_SME_F8F16
1	FVDOT (FP8 to FP16)	FEAT_SME_F8F16

SME2 Multi-vector - Indexed (Four registers)

SME2 Multi-vector - Indexed (Four registers)

The encodings in this section are decoded from SME encodings.

31							24	23	22	21	20	19	16	15	14	13	12	11	10	7	6	4	3	2	0
1	1	0	0	0	0	0	1	op0		0	1			1			op1				op2		op3		

op0	op1	op2	op3	Instruction details	Condition
00	0x	0xx	x	SME2 multi-vec indexed long long MLA four sources 32-bit	-
00	0x	100	0	FMLALL (multiple and indexed vector) — Four ZA quad-vectors	FEAT_SME_F8F32
00	0x	100	1	UNALLOCATED	-
00	1x	0xx	x	SME2 multi-vec ternary indexed four registers 16-bit	-
00	1x	100	x	FDOT (2-way, multiple and indexed vector, FP8 to FP16) — Four ZA single-vectors	FEAT_SME_F8F16
00	xx	101	x	UNALLOCATED	-
00	xx	11x	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
01	xx	0xx	x	SME2 multi-vec ternary indexed four registers 32-bit	-
10	00	00x	x	SME2 multi-vec indexed long long MLA four sources 64-bit	-
10	01	00x	x	UNALLOCATED	-
10	0x	01x	x	UNALLOCATED	-
10	1x	00x	x	SME2 multi-vec indexed long FMA four sources	-
10	1x	010	x	FMLAL (multiple and indexed vector, FP8 to FP16) — Four ZA double-vectors	FEAT_SME_F8F16
10	1x	011	x	UNALLOCATED	-
11	0x	00x	x	SME2 multi-vec ternary indexed four registers 64-bit	-
11	1x	00x	x	SME2 multi-vec indexed long MLA four sources	-
11	xx	01x	x	UNALLOCATED	-
!= 00	xx	1xx	x	UNALLOCATED	-

SME2 multi-vec indexed long long MLA four sources 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm		1	Rv		0	i4h		Zn		0	op	U	S	i41		o1

op	U	S	Instruction Details	Condition
0	0	0	SMLALL (multiple and indexed vector)	FEAT_SME2
0	0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2
0	1	0	UMLALL (multiple and indexed vector)	FEAT_SME2
0	1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2
1	x	1	UNALLOCATED	-
1	0	0	USMLALL (multiple and indexed vector)	FEAT_SME2
1	1	0	SUMLALL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed four registers 16-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm		1	Rv		1	i3h		Zn		0	op	S	i31	off3	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	Instruction Details	Condition
0	0	FMLA (multiple and indexed vector)	FEAT_SME_F16F16
0	1	FMLS (multiple and indexed vector)	FEAT_SME_F16F16
1	0	BFMLA (multiple and indexed vector)	FEAT_SME_B16B16
1	1	BFMLS (multiple and indexed vector)	FEAT_SME_B16B16

SME2 multi-vec ternary indexed four registers 32-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	3	2	0
1 1 0 0 0 0 0 1		0	1	0	1	Zm		1	Rv		op	i2		Zn		0	opc2		off3	

op	opc2	Instruction Details	Condition
0	000	FMLA (multiple and indexed vector)	FEAT_SME2
0	001	FDOT (4-way, multiple and indexed vector)	FEAT_SME_F8F32
0	010	FMLS (multiple and indexed vector)	FEAT_SME2
0	011	UNALLOCATED	-
0	100	SVDOT (4-way)	FEAT_SME2
0	101	USVDOT	FEAT_SME2
0	110	UVDOT (4-way)	FEAT_SME2
0	111	SUVDOT	FEAT_SME2
1	000	SDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	001	FDOT (2-way, multiple and indexed vector, FP16 to FP32)	FEAT_SME2
1	010	UDOT (2-way, multiple and indexed vector)	FEAT_SME2
1	011	BFDOT (multiple and indexed vector)	FEAT_SME2
1	100	SDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	101	USDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	110	UDOT (4-way, multiple and indexed vector)	FEAT_SME2
1	111	SUDOT (4-way, multiple and indexed vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec indexed long long MLA four sources 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm		1	Rv		0	0	i3h	Zn		0	0	U	S	i3l		o1

U	S	Instruction Details	Condition
0	0	SMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	1	SMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	0	UMLALL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	1	UMLSLL (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long FMA four sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm		1	Rv		1	i3h		Zn		0	0	op	S	i3l	off2	

op	S	Instruction Details	Condition
0	0	FMLAL (multiple and indexed vector, FP16 to FP32)	FEAT_SME2
0	1	FMLSL (multiple and indexed vector)	FEAT_SME2
1	0	BFMLAL (multiple and indexed vector)	FEAT_SME2
1	1	BFMLSL (multiple and indexed vector)	FEAT_SME2

SME2 multi-vec ternary indexed four registers 64-bit

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31							24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm		1	Rv		0	op	i1	Zn		0	0	opc2		off3	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	opc2	Instruction Details	Condition
0	00	FMLA (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
0	01	SDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
0	10	FMLS (multiple and indexed vector)	FEAT_SME2 && FEAT_SME_F64F64
0	11	UDOT (4-way, multiple and indexed vector)	FEAT_SME2 && FEAT_SME_I16I64
1	x0	UNALLOCATED	-
1	01	SVDOT (4-way)	FEAT_SME2 && FEAT_SME_I16I64
1	11	UVDOT (4-way)	FEAT_SME2 && FEAT_SME_I16I64

SME2 multi-vec indexed long MLA four sources

The encodings in this section are decoded from SME2 Multi-vector - Indexed (Four registers).

31	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	0
1 0 0 0 0 1 0		0	0	xs	1	Zm		0	msz		Pg		Rn		0	prfop	

U	S	Instruction Details	Condition
0	0	SMLAL (multiple and indexed vector)	FEAT_SME2
0	1	SMLSL (multiple and indexed vector)	FEAT_SME2
1	0	UMLAL (multiple and indexed vector)	FEAT_SME2
1	1	UMLSL (multiple and indexed vector)	FEAT_SME2

SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)

SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	0
1 1 0 0 0 0 0	1			1	0			1 0 1 0 0		op0	op1		op2			

op0	op1	op2	Instruction details
0	00	00x	SME2 single-multi int min/max two registers
0	01	00x	SME2 single-multi FP min/max two registers

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0	01	100	SME2 multiple and single vector FSCALE two registers
0	10	xxx	SME2 single-multi shift two registers
0	11	000	SME2 single-multi add two registers
0	11	001	UNALLOCATED
0	11	10x	UNALLOCATED
0	!= 1x	101	UNALLOCATED
0	!= 10	x1x	UNALLOCATED
1	00	000	SME2 single-multi signed saturating doubling multiply high two registers
1	00	!= x00	UNALLOCATED
1	!= 00	xxx	UNALLOCATED
x	00	100	UNALLOCATED

SME2 single-multi int min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23 22	21 20	19 16	15				11	10	9 8	7 6	5	4 1	0
1	1	0	0	0	0	0	1	size	1 0	Zm	1	0	1	0	0	0	0 0	0 0	op	Zdn	U

op	U	Instruction Details	Condition
0	0	SMAX (multiple and single vector)	FEAT_SME2
0	1	UMAX (multiple and single vector)	FEAT_SME2
1	0	SMIN (multiple and single vector)	FEAT_SME2
1	1	UMIN (multiple and single vector)	FEAT_SME2

SME2 single-multi FP min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31						25	24	23	22	21	20 16	15	14	13	12 10	9 5	4 0
1	0	0	0	0	1	0	1	0	xs	1	Zm	0	U	ff	Pg	Rn	Zt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	op	o2	Instruction Details	Condition
00	0	0	BFMAX (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	0	1	BFMIN (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	0	BFMAXNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	1	BFMINNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	0	FMAX (multiple and single vector)	FEAT_SME2
!= 00	0	1	FMIN (multiple and single vector)	FEAT_SME2
!= 00	1	0	FMAXNM (multiple and single vector)	FEAT_SME2
!= 00	1	1	FMINNM (multiple and single vector)	FEAT_SME2

SME2 multiple and single vector FSCALE two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23	22	21	20	19				16	15				11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm					1	0	1	0	0	0	0	1	1	0	0		Zdn				op

size	op	Instruction Details	Condition
xx	1	UNALLOCATED	-
00	0	BFSCALE (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	0	FSCALE (multiple and single vector)	FEAT_SME2 && FEAT_FP8

SME2 single-multi shift two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31							24	23	22	21	20	19				16	15				11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm					1	0	1	0	0	0	1	0	opc				Zdn				U

opc	U	Instruction Details	Condition
001	0	SRSHL (multiple and single vector)	FEAT_SME2
001	1	URSHL (multiple and single vector)	FEAT_SME2
!= 001	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 single-multi add two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1 0 0 0 0 1 0	msz		0	0	Rm		1	0	U	Pg		Zn		Zt	

op	Instruction Details	Condition
0	ADD (to vector, multiple vectors)	FEAT_SME2
1	UNALLOCATED	-

SME2 single-multi signed saturating doubling multiply high two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers).

31	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	1	0
1 1 0 0 0 0 0	1	size		1	0	Zm		1 0 1 0 0		1	0	0	0 0 0		Zdn		op

op	Instruction Details	Condition
0	SQDMULH (multiple and single vector)	FEAT_SME2
1	UNALLOCATED	-

SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)

SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)

The encodings in this section are decoded from SME encodings.

31	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	2	1	0
1 1 0 0 0 0 0	1			1	0			1 0 1 0	1	op0	op1		op2				0	

op0	op1	op2	Instruction details
0	00	00x	SME2 single-multi int min/max four registers

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0	01	00x	SME2 single-multi FP min/max four registers
0	01	100	SME2 multiple and single vector FSCALE four registers
0	10	xxx	SME2 single-multi shift four registers
0	11	000	SME2 single-multi add four registers
0	11	001	UNALLOCATED
0	11	10x	UNALLOCATED
0	!= 1x	101	UNALLOCATED
0	!= 10	x1x	UNALLOCATED
1	00	000	SME2 single-multi signed saturating doubling multiply high four registers
1	00	!= x00	UNALLOCATED
1	!= 00	xxx	UNALLOCATED
x	00	100	UNALLOCATED

SME2 single-multi int min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	op	Zdn				0	U

op	U	Instruction Details	Condition
0	0	SMAX (multiple and single vector)	FEAT_SME2
0	1	UMAX (multiple and single vector)	FEAT_SME2
1	0	SMIN (multiple and single vector)	FEAT_SME2
1	1	UMIN (multiple and single vector)	FEAT_SME2

SME2 single-multi FP min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	1	0	0	op	Zdn				0	o2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	op	o2	Instruction Details	Condition
00	0	0	BFMAX (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	0	1	BFMIN (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	0	BFMAXNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
00	1	1	BFMINNM (multiple and single vector)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	0	FMAX (multiple and single vector)	FEAT_SME2
!= 00	0	1	FMIN (multiple and single vector)	FEAT_SME2
!= 00	1	0	FMAXNM (multiple and single vector)	FEAT_SME2
!= 00	1	1	FMINNM (multiple and single vector)	FEAT_SME2

SME2 multiple and single vector FSCALE four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23 22	21	20	19 16	15				11	10	9	8	7		5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm	1	0	1	0	1	0	0	1	1	0	0	Zdn	0	op

size	op	Instruction Details	Condition
xx	1	UNALLOCATED	-
00	0	BFSCALE (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	0	FSCALE (multiple and single vector)	FEAT_SME2 && FEAT_FP8

SME2 single-multi shift four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23 22	21	20	19 16	15				11	10	9	8	7		5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm	1	0	1	0	1	0	1	0	opc			Zdn	0	U

opc	U	Instruction Details	Condition
001	0	SRSHL (multiple and single vector)	FEAT_SME2
001	1	URSHL (multiple and single vector)	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	U	Instruction Details	Condition
!=001	x	UNALLOCATED	-

SME2 single-multi add four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19				16	15				11	10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm					1	0	1	0	1	0	1	1	0	0	0		Zdn				0	op

op	Instruction Details	Condition
0	ADD (to vector, multiple vectors)	FEAT_SME2
1	UNALLOCATED	-

SME2 single-multi signed saturating doubling multiply high four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers).

31							24	23	22	21	20	19				16	15				11	10	9	8	7			5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm					1	0	1	0	1	1	0	0	0	0	0		Zdn				0	op

op	Instruction Details	Condition
0	SQDMULH (multiple and single vector)	FEAT_SME2
1	UNALLOCATED	-

SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)

SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)

The encodings in this section are decoded from SME encodings.

31							25	24	23	22	21	20			18	17							10	9			7	6				2	1	0
1	1	0	0	0	0	0		1			1					0	0	1	0	1	1	1	0	op0									0	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
000	SME2 multiple vectors int min/max four registers
001	UNALLOCATED
010	SME2 multiple vectors FP min/max four registers
011	SME2 multiple vectors FSCALE four registers
10x	SME2 multiple vectors shift four registers
11x	UNALLOCATED

SME2 multiple vectors int min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23 22	21	20 18	17							10	9		7	6 5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	Zm	0	0	1	0	1	1	1	0	0	0	0	opc	Zdn	0	U

opc	U	Instruction Details	Condition
00	0	SMAX (multiple vectors)	FEAT_SME2
00	1	UMAX (multiple vectors)	FEAT_SME2
01	0	SMIN (multiple vectors)	FEAT_SME2
01	1	UMIN (multiple vectors)	FEAT_SME2
1x	x	UNALLOCATED	-

SME2 multiple vectors FP min/max four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23 22	21	20 18	17							10	9		7	6 5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	Zm	0	0	1	0	1	1	1	0	0	1	0	opc	Zdn	0	o2

size	opc	o2	Instruction Details	Condition
xx	10	0	FAMAX	FEAT_SME2 && FEAT_FAMINMAX
xx	10	1	FAMIN	FEAT_SME2 && FEAT_FAMINMAX
xx	11	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	o2	Instruction Details	Condition
00	00	0	BFMAX (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	00	1	BFMIN (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	0	BFMAXNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	1	BFMINNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	00	0	FMAX (multiple vectors)	FEAT_SME2
!= 00	00	1	FMIN (multiple vectors)	FEAT_SME2
!= 00	01	0	FMAXNM (multiple vectors)	FEAT_SME2
!= 00	01	1	FMINNM (multiple vectors)	FEAT_SME2

SME2 multiple vectors FSCALE four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23 22	21	20 18	17							10	9 7			6 5	4 2	1	0
1	1	0	0	0	0	0	1	size	1	Zm	0	0	1	0	1	1	1	0	0	1	1	opc	Zdn	0	o2

size	opc	o2	Instruction Details	Condition
xx	00	1	UNALLOCATED	-
xx	!= 00	x	UNALLOCATED	-
00	00	0	BFSCALE (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	00	0	FSCALE (multiple vectors)	FEAT_SME2 && FEAT_FP8

SME2 multiple vectors shift four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers).

31							24	23 22	21	20 18	17									8	7 5		4 2	1	0
1	1	0	0	0	0	0	1	size	1	Zm	0	0	1	0	1	1	1	0	1	0	opc		Zdn	0	U

opc	U	Instruction Details	Condition
001	0	SRSHL (multiple vectors)	FEAT_SME2
001	1	URSHL (multiple vectors)	FEAT_SME2
!= 001	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers)

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers)

The encodings in this section are decoded from SME encodings.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1 0 0 1 0	dtype	0 0	Rm	1 0 0	Pg	Rn	Zt

op0	Instruction details
00000	SME2 multi-vector signed saturating doubling multiply high four registers
!= 00000	UNALLOCATED

SME2 multi-vector signed saturating doubling multiply high four registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers).

31–24	23–22	21	20–18	17–10	9–5	4–2	1	0
1 1 0 0 0 0 0 1	size	1	Zm	0 0 1 0 1 1 1 1	0 0 0 0 0	Zdn	0	op

op	Instruction Details	Condition
0	SQDMULH (multiple vectors)	FEAT_SME2
1	UNALLOCATED	-

SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)

SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)

The encodings in this section are decoded from SME encodings.

31–25	24	23–22	21	20–17	16–10	9–7	6–0
1 1 0 0 0 0 0	1		1		0 1 0 1 1 0 0	op0	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
000	SME2 multiple vectors int min/max two registers
001	UNALLOCATED
010	SME2 multiple vectors FP min/max two registers
011	SME2 multiple vectors FSCALE two registers
10x	SME2 multiple vectors shift two registers
11x	UNALLOCATED

SME2 multiple vectors int min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23 22	21	20 17	16						10	9		7	6 5	4 1	0
1	1	0	0	0	0	0	1	size	1	Zm	0	1	0	1	1	0	0	0	0	0	opc	Zdn	U

opc	U	Instruction Details	Condition
00	0	SMAX (multiple vectors)	FEAT_SME2
00	1	UMAX (multiple vectors)	FEAT_SME2
01	0	SMIN (multiple vectors)	FEAT_SME2
01	1	UMIN (multiple vectors)	FEAT_SME2
1x	x	UNALLOCATED	-

SME2 multiple vectors FP min/max two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31						25	24 21	20	19 16	15		13	12 10	9 5	4 0
1	0	1	0	0	1	0	dtype	0	imm4	1	0	1	Pg	Rn	Zt

size	opc	o2	Instruction Details	Condition
xx	10	0	FAMAX	FEAT_SME2 && FEAT_FAMINMAX
xx	10	1	FAMIN	FEAT_SME2 && FEAT_FAMINMAX
xx	11	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	o2	Instruction Details	Condition
00	00	0	BFMAX (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	00	1	BFMIN (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	0	BFMAXNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
00	01	1	BFMINNM (multiple vectors)	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	00	0	FMAX (multiple vectors)	FEAT_SME2
!= 00	00	1	FMIN (multiple vectors)	FEAT_SME2
!= 00	01	0	FMAXNM (multiple vectors)	FEAT_SME2
!= 00	01	1	FMINNM (multiple vectors)	FEAT_SME2

SME2 multiple vectors FSCALE two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	1	1		opc		Zdn				o2

size	opc	o2	Instruction Details	Condition
xx	00	1	UNALLOCATED	-
xx	!= 00	x	UNALLOCATED	-
00	00	0	BFSCALE (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	00	0	FSCALE (multiple vectors)	FEAT_SME2 && FEAT_FP8

SME2 multiple vectors shift two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers).

31							24	23	22	21	20			17	16								8	7		5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	1	0		opc			Zdn			U

opc	U	Instruction Details	Condition
001	0	SRSHL (multiple vectors)	FEAT_SME2
001	1	URSHL (multiple vectors)	FEAT_SME2
!= 001	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers)

SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers)

The encodings in this section are decoded from SME encodings.

31–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1 0 0 1 0	dtype	Rm	0 1 0	Pg	Rn	Zt

op0	Instruction details
00000	SME2 multi-vector signed saturating doubling multiply high two registers
!=00000	UNALLOCATED

SME2 multi-vector signed saturating doubling multiply high two registers

The encodings in this section are decoded from SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers).

31–24	23–22	21	20–17	16–10	9–5	4–1	0
1 1 0 0 0 0 0 1	size	1	Zm	0 1 0 1 1 0 1	0 0 0 0 0	Zdn	op

op	Instruction Details	Condition
0	SQDMULH (multiple vectors)	FEAT_SME2
1	UNALLOCATED	-

SME2 Multi-vector - SVE Select

SME2 Multi-vector - SVE Select

The encodings in this section are decoded from SME encodings.

31–24	23–22	21	20–18	17–16	15–13	12–7	6–5	4–2	1–0
1 1 0 0 0 0 0 1		1		op0	1 0 0		op1		op2

op0	op1	op2	Instruction details	Condition
01	00	00	SEL — Four registers	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details	Condition
01	00	10	UNALLOCATED	-
01	10	x0	UNALLOCATED	-
11	x0	x0	UNALLOCATED	-
x0	x0	x0	SEL — Two registers	FEAT_SME2
xx	x0	x1	UNALLOCATED	-
xx	x1	xx	UNALLOCATED	-

SME2 Multi-vector - SVE Constructive Binary

SME2 Multi-vector - SVE Constructive Binary

The encodings in this section are decoded from SME encodings.

31	25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1 0 0 1 0		msz	!= 00	0	imm4	1 1 1	Pg	Rn	Zt
			opc						

op0	op1	op2	Instruction details
00	101	x	SME2 multi-vec quadwords ZIP two registers
01	101	x	UNALLOCATED
10	101	x	UNALLOCATED
11	101	x	SME2 multi-vec saturating shift right narrow two registers
xx	000	x	SME2 multi-vec FCLAMP two registers
xx	001	x	SME2 multi-vec CLAMP two registers
xx	010	0	SME2 multi-vec FCLAMP four registers
xx	011	0	SME2 multi-vec CLAMP four registers
xx	01x	1	UNALLOCATED
xx	100	x	SME2 multi-vec ZIP two registers
xx	11x	x	SME2 multi-vec saturating shift right narrow four registers

SME2 multi-vec quadwords ZIP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31 24	23 22	21	20 16	15 13	12 10	9 5	4 1	0
1 1 0 0 0 0 0 1	0 0	1	Zm	1 1 0	1 0 1	Zn	Zd	op

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Condition
0	ZIP (two registers)	FEAT_SME2
1	UZP (two registers)	FEAT_SME2

SME2 multi-vec saturating shift right narrow two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20	19			16	15			13	12			10	9			6	5	4				0
1	1	0	0	0	0	0	1	1	1	1	op	imm4				1	1	0		1	0	1		Zn				U	Zd				

op	U	Instruction Details	Condition
0	0	SQRSHR (two registers)	FEAT_SME2
0	1	UQRSHR (two registers)	FEAT_SME2
1	0	SQRSHRU (two registers)	FEAT_SME2
1	1	UNALLOCATED	-

SME2 multi-vec FCLAMP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31							24	23	22	21	20				16	15			13	12			10	9			5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm					1	1	0		0	0	0		Zn				Zd				op

size	op	Instruction Details	Condition
xx	1	UNALLOCATED	-
00	0	BFCLAMP	FEAT_SME2 && FEAT_SVE_B16B16
!= 00	0	FCLAMP	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec CLAMP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31	24	23 22	21	20 16	15 13	12 10	9 5	4 1	0
1 1 0 0 0 0 0 1		size	1	Zm	1 1 0	0 0 1	Zn	Zd	U

U	Instruction Details	Condition
0	SCLAMP	FEAT_SME2
1	UCLAMP	FEAT_SME2

SME2 multi-vec FCLAMP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31	25	24 23	22 21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0 0 0 1	0	0 0	1 1	Zm	1	msz	Pg	Rn	0	prfop

size	op	Instruction Details	Condition
xx	1	UNALLOCATED	-
00	0	BFCLAMP	FEAT_SME2 && FEAT_SVE_B16B16
!=00	0	FCLAMP	FEAT_SME2

SME2 multi-vec CLAMP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31	24	23 22	21	20 16	15 13	12 10	9 5	4 2	1	0
1 1 0 0 0 0 0 1		size	1	Zm	1 1 0	0 1 1	Zn	Zd	0	U

U	Instruction Details	Condition
0	SCLAMP	FEAT_SME2
1	UCLAMP	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec ZIP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31	24	23 22	21	20 16	15 13	12 10	9 5	4 1	0
1 1 0 0 0 0 0 1		size	1	Zm	1 1 0	1 0 0	Zn	Zd	op

op	Instruction Details	Condition
0	ZIP (two registers)	FEAT_SME2
1	UZP (two registers)	FEAT_SME2

SME2 multi-vec saturating shift right narrow four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Binary.

31	24	23 22	21	20 16	15 11	10	9 7	6	5	4 0
1 1 0 0 0 0 0 1		tsize	1	imm5	1 1 0 1 1	N	Zn	op	U	Zd

N	op	U	Instruction Details	Condition
x	1	1	UNALLOCATED	-
0	0	0	SQRSHR (four registers)	FEAT_SME2
0	0	1	UQRSHR (four registers)	FEAT_SME2
0	1	0	SQRSHRU (four registers)	FEAT_SME2
1	0	0	SQRSHRN	FEAT_SME2
1	0	1	UQRSHRN	FEAT_SME2
1	1	0	SQRSHRUN	FEAT_SME2

SME2 Multi-vector - SVE Constructive Unary

SME2 Multi-vector - SVE Constructive Unary

The encodings in this section are decoded from SME encodings.

31	24	23 22	21	20 18	17 16	15 10	9 7	6 5	4 2	1 0
1 1 0 0 0 0 0 1		op0	1	op1	op2	1 1 1 0 0 0		op3		op4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	Instruction details
00	000	01	xx	x0	SME2 multi-vec FP to int convert two registers
00	000	10	xx	x0	SME2 multi-vec int to FP two registers
00	001	11	xx	xx	UNALLOCATED
00	100	01	0x	00	SME2 multi-vec FP to int convert four registers
00	100	10	0x	00	SME2 multi-vec int to FP four registers
00	101	00	0x	xx	SME2 multi-vec FP8 down convert four registers
00	101	11	00	x0	SME2 multi-vec quadwords ZIP four registers
01	000	01	xx	x0	UNALLOCATED
01	001	11	xx	xx	UNALLOCATED
0x	000	00	xx	xx	SME2 multi-vec FP down convert two registers
0x	000	11	xx	xx	SME2 multi-vec int down convert two registers
0x	001	00	x0	xx	SME2 multi-vec FP8 down convert two registers
0x	001	00	x1	xx	UNALLOCATED
10	000	00	xx	xx	SME2 multi-vec convert two registers
10	001	00	1x	xx	UNALLOCATED
10	x01	00	0x	xx	UNALLOCATED
11	00x	00	xx	xx	UNALLOCATED
1x	000	11	xx	x1	UNALLOCATED
1x	000	x1	xx	x0	UNALLOCATED
1x	001	11	xx	xx	UNALLOCATED
x0	100	00	0x	0x	UNALLOCATED
xx	000	01	xx	x1	UNALLOCATED
xx	000	10	xx	x1	UNALLOCATED
xx	001	01	xx	xx	SME2 multi-vec unpack two registers
xx	001	10	xx	xx	SME2 multi-vec FP8 up convert two registers
xx	01x	xx	x0	x0	SME2 multi-vec FRINT two registers
xx	100	01	0x	01	UNALLOCATED
xx	100	01	1x	xx	UNALLOCATED
xx	100	10	0x	01	UNALLOCATED
xx	100	11	xx	xx	SME2 multi-vec int down convert four registers
xx	100	!=11	0x	1x	UNALLOCATED
xx	101	01	x0	0x	SME2 multi-vec unpack four registers
xx	101	01	x0	1x	UNALLOCATED
xx	101	01	x1	xx	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	Instruction details
xx	101	10	00	x0	SME2 multi-vec ZIP four registers
xx	101	11	1x	xx	UNALLOCATED
xx	101	1x	00	x1	UNALLOCATED
xx	101	1x	01	xx	UNALLOCATED
xx	10x	x0	1x	xx	UNALLOCATED
xx	11x	xx	00	00	SME2 multi-vec FRINT four registers
xx	11x	xx	00	10	UNALLOCATED
xx	11x	xx	10	x0	UNALLOCATED
xx	x1x	xx	x0	x1	UNALLOCATED
xx	x1x	xx	x1	xx	UNALLOCATED
!= x0	101	00	0x	1x	UNALLOCATED
!= x0	10x	00	0x	0x	UNALLOCATED
!= 00	000	10	xx	x0	UNALLOCATED
!= 00	100	01	0x	00	UNALLOCATED
!= 00	100	10	0x	00	UNALLOCATED
!= 00	101	11	00	x0	UNALLOCATED

SME2 multi-vec FP to int convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20			18	17	16	15						10	9			6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0		0	1	1	1	1	0	0	0		Zn				U	Zd				0

U	Instruction Details	Condition
0	FCVTZS	FEAT_SME2
1	FCVTZU	FEAT_SME2

SME2 multi-vec int to FP two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20			18	17	16	15						10	9			6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0		1	0	1	1	1	0	0	0		Zn				U	Zd				0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Condition
0	SCVTF	FEAT_SME2
1	UCVTF	FEAT_SME2

SME2 multi-vec FP to int convert four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15						10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	Zn				0	U	Zd			0	0

U	Instruction Details	Condition
0	FCVTZS	FEAT_SME2
1	FCVTZU	FEAT_SME2

SME2 multi-vec int to FP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15						10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	1	0	1	1	1	0	0	0	Zn				0	U	Zd			0	0

U	Instruction Details	Condition
0	SCVTF	FEAT_SME2
1	UCVTF	FEAT_SME2

SME2 multi-vec FP8 down convert four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15						10	9		7	6	5	4				0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	0	Zn				0	N	Zd				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

N	Instruction Details	Condition
0	FCVT (narrowing, FP32 to FP8)	FEAT_SME2 && FEAT_FP8
1	FCVTN (FP32 to FP8)	FEAT_SME2 && FEAT_FP8

SME2 multi-vec quadwords ZIP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn		0	0	Zd		op	0

op	Instruction Details	Condition
0	ZIP (four registers)	FEAT_SME2
1	UZP (four registers)	FEAT_SME2

SME2 multi-vec FP down convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31								23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	op	1	0	0	0	0	0	1	1	1	0	0	0	Zn		N	Zd	

op	N	Instruction Details	Condition
0	0	FCVT (narrowing, FP32 to FP16)	FEAT_SME2
0	1	FCVTN (FP32 to FP16)	FEAT_SME2
1	0	BFCVT (single-precision to BFloat16)	FEAT_SME2
1	1	BFCVTN	FEAT_SME2

SME2 multi-vec int down convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31								23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	op	1	0	0	0	1	1	1	1	1	0	0	0	Zn		U	Zd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	Instruction Details	Condition
0	0	SQCVT (two registers)	FEAT_SME2
0	1	UQCVT (two registers)	FEAT_SME2
1	0	SQCVTU (two registers)	FEAT_SME2
1	1	UNALLOCATED	-

SME2 multi-vec FP8 down convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31								23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	op	1	0	0	1	0	0	1	1	1	0	0	0	Zn		0	Zd	

op	Instruction Details	Condition
0	FCVT (narrowing, FP16 to FP8)	FEAT_SME2 && FEAT_FP8
1	BFCVT (BFloat16 to 8-bit floating-point)	FEAT_SME2 && FEAT_FP8

SME2 multi-vec convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	1	1	0	0	0	Zn		Zd		L

L	Instruction Details	Condition
0	FCVT (widening)	FEAT_SME_F16F16
1	FCVTL	FEAT_SME_F16F16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 multi-vec unpack two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31	24	23 22	21	20 18	17 16	15 10	9 5	4 1	0
1 1 0 0 0 0 0 1		size	1	0 0 1	0 1	1 1 1 0 0 0	Zn	Zd	U

U	Instruction Details	Condition
0	SUNPK	FEAT_SME2
1	UUNPK	FEAT_SME2

SME2 multi-vec FP8 up convert two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31	25	24 23	22	21	20 16	15 13	12 10	9 5	4 0
1 1 1 0 0 1	0	msz	1	0	Rm	0 0 1	Pg	Zn	Zt

opc	L	Instruction Details	Condition
00	0	F1CVT, F2CVT — F1CVT	FEAT_SME2 && FEAT_FP8
00	1	F1CVTL, F2CVTL — F1CVTL	FEAT_SME2 && FEAT_FP8
01	0	BF1CVT, BF2CVT — BF1CVT	FEAT_SME2 && FEAT_FP8
01	1	BF1CVTL, BF2CVTL — BF1CVTL	FEAT_SME2 && FEAT_FP8
10	0	F1CVT, F2CVT — F2CVT	FEAT_SME2 && FEAT_FP8
10	1	F1CVTL, F2CVTL — F2CVTL	FEAT_SME2 && FEAT_FP8
11	0	BF1CVT, BF2CVT — BF2CVT	FEAT_SME2 && FEAT_FP8
11	1	BF1CVTL, BF2CVTL — BF2CVTL	FEAT_SME2 && FEAT_FP8

SME2 multi-vec FRINT two registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31	24	23 22	21 19	18 16	15 10	9 6	5	4 1	0
1 1 0 0 0 0 0	1	size	1 0 1	opc	1 1 1 0 0 0	Zn	0	Zd	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Condition
10	000	FRINTN	FEAT_SME2
10	001	FRINTP	FEAT_SME2
10	010	FRINTM	FEAT_SME2
10	011	UNALLOCATED	-
10	100	FRINTA	FEAT_SME2
10	101	UNALLOCATED	-
10	11x	UNALLOCATED	-
!= 10	xxx	UNALLOCATED	-

SME2 multi-vec int down convert four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	0
1	1	0	0	0	0	0	1	sz	op	1	1	0	0	1	1	1	1	1	0	0	0	Zn		N	U	Zd	

op	N	U	Instruction Details	Condition
0	0	0	SQCVT (four registers)	FEAT_SME2
0	0	1	UQCVT (four registers)	FEAT_SME2
0	1	0	SQCVTN	FEAT_SME2
0	1	1	UQCVTN	FEAT_SME2
1	x	1	UNALLOCATED	-
1	0	0	SQCVTU (four registers)	FEAT_SME2
1	1	0	SQCVTUN	FEAT_SME2

SME2 multi-vec unpack four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9	6	5	4	2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	0	1	1	1	1	0	0	0	Zn		0	Zd		0	U

U	Instruction Details	Condition
0	SUNPK	FEAT_SME2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Condition
1	UUNPK	FEAT_SME2

SME2 multi-vec ZIP four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21	20		18	17	16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	1	0	1	1	1	0	0	0	Zn			0	0	Zd			op	0

op	Instruction Details	Condition
0	ZIP (four registers)	FEAT_SME2
1	UZP (four registers)	FEAT_SME2

SME2 multi-vec FRINT four registers

The encodings in this section are decoded from SME2 Multi-vector - SVE Constructive Unary.

31							24	23	22	21			19	18		16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	1	1		opc			1	1	1	0	0	0	Zn			0	0	Zd			0	0

size	opc	Instruction Details	Condition
10	000	FRINTN	FEAT_SME2
10	001	FRINTP	FEAT_SME2
10	010	FRINTM	FEAT_SME2
10	011	UNALLOCATED	-
10	100	FRINTA	FEAT_SME2
10	101	UNALLOCATED	-
10	11x	UNALLOCATED	-
!= 10	xxx	UNALLOCATED	-

SME2 Multi-vector - FP Multiply

SME2 Multi-vector - FP Multiply

The encodings in this section are decoded from SME encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20		18	17	16	15						10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1			1				op0		1	1	1	0	0	1					op1	0				op2	0

op0	op1	op2	Instruction details
01	0	0	SME2 multi-vec FP multiply (four registers)
01	0	1	UNALLOCATED
01	1	x	UNALLOCATED
11	x	x	UNALLOCATED
x0	x	x	SME2 multi-vec FP multiply (two registers)

SME2 multi-vec FP multiply (four registers)

The encodings in this section are decoded from SME2 Multi-vector - FP Multiply.

31							24	23	22	21	20		18	17	16	15						10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	1	1	1	1	0	0	1	Zn				0	0	Zd			0	0

size	Instruction Details	Condition
00	BFMUL (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple vectors)	FEAT_SME2p2

SME2 multi-vec FP multiply (two registers)

The encodings in this section are decoded from SME2 Multi-vector - FP Multiply.

31							24	23	22	21	20				17	16						10	9				6	5	4				1	0
1	1	0	0	0	0	0	1	size		1	Zm					0	1	1	1	0	0	1	Zn					0	Zd					0

size	Instruction Details	Condition
00	BFMUL (multiple vectors)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple vectors)	FEAT_SME2p2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 Multiple and Single Vector - FP multiply

SME2 Multiple and Single Vector - FP multiply

The encodings in this section are decoded from SME encodings.

31–24	23–22	21	20–17	16	15–10	9–7	6	5	4–2	1	0
11000001		1		op0	110110		op1	0		op2	0

op0	op1	op2	Instruction details
0	x	x	SME2 multiple and single vector FP multiply (two registers)
1	0	0	SME2 multiple and single vector FP multiply (four registers)
1	0	1	UNALLOCATED
1	1	x	UNALLOCATED

SME2 multiple and single vector FP multiply (two registers)

The encodings in this section are decoded from SME2 Multiple and Single Vector - FP multiply.

31–25	24–22	21	20–16	15	14	13	12–10	9–5	4–0
1110010	!= 110 (opc)	o2	Rm	0	1	0	Pg	Rn	Zt

size	Instruction Details	Condition
00	BFMUL (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple and single vector)	FEAT_SME2p2

SME2 multiple and single vector FP multiply (four registers)

The encodings in this section are decoded from SME2 Multiple and Single Vector - FP multiply.

31–24	23–22	21	20–17	16	15–10	9–7	6	5	4–2	1	0
11000001	size	1	Zm	1	110110	Zn	0	0	Zd	0	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Condition
00	BFMUL (multiple and single vector)	FEAT_SME2 && FEAT_SVE_BFSCALE
!= 00	FMUL (multiple and single vector)	FEAT_SME2p2

SME Memory

SME Memory

The encodings in this section are decoded from SME encodings.

31	25	24	21	20	16	15	14	10	9	5	4	2	1	0
1 1 1 0 0 0 0		op0		op1		op2	op3				op4			

op0	op1	op2	op3	op4	Instruction details	Condition
0xx0	xxxxx	x	xxxxx	0xx	SME load array vector (elements)	-
0xx1	xxxxx	x	xxxxx	0xx	SME store array vector (elements)	-
100x	00000	0	xx000	0xx	SME save and restore array	-
100x	xxxxx	1	00000	000	SME2 lookup table load/store	-
100x	xxxxx	1	00000	001	UNALLOCATED	-
100x	xxxxx	1	00000	01x	UNALLOCATED	-
100x	xxxxx	1	01000	0xx	UNALLOCATED	-
100x	xxxxx	1	1x000	0xx	UNALLOCATED	-
100x	xxxxx	x	!= xx000	0xx	UNALLOCATED	-
100x	!= 00000	0	xx000	0xx	UNALLOCATED	-
101x	xxxxx	x	xxxxx	0xx	UNALLOCATED	-
110x	xxxxx	x	xxxxx	0xx	UNALLOCATED	-
1110	xxxxx	x	xxxxx	0xx	LD1Q	FEAT_SME
1111	xxxxx	x	xxxxx	0xx	ST1Q	FEAT_SME
xxxx	xxxxx	x	xxxxx	1xx	UNALLOCATED	-

SME load array vector (elements)

The encodings in this section are decoded from SME Memory.

31	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	0
1 1 1 0 0 0 0	0	msz		0	Rm		V	Rs		Pg		Rn		0	opc	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Condition
00	LD1B (scalar plus scalar, tile slice)	FEAT_SME
01	LD1H (scalar plus scalar, tile slice)	FEAT_SME
10	LD1W (scalar plus scalar, tile slice)	FEAT_SME
11	LD1D (scalar plus scalar, tile slice)	FEAT_SME

SME store array vector (elements)

The encodings in this section are decoded from SME Memory.

31							24	23	22	21	20			16	15	14	13	12		10	9			5	4	3		0
1	1	1	0	0	0	0	0	msz		1	Rm				V	Rs		Pg			Rn				0	opc		

msz	Instruction Details	Condition
00	ST1B (scalar plus scalar, tile slice)	FEAT_SME
01	ST1H (scalar plus scalar, tile slice)	FEAT_SME
10	ST1W (scalar plus scalar, tile slice)	FEAT_SME
11	ST1D (scalar plus scalar, tile slice)	FEAT_SME

SME save and restore array

The encodings in this section are decoded from SME Memory.

31									22	21	20				16	15	14	13	12			10	9			5	4	3		0
1	1	1	0	0	0	0	1	0	0	op	0	0	0	0	0	0	Rv		0	0	0		Rn				0	imm4		

op	Instruction Details	Condition
0	LDR (array vector)	FEAT_SME
1	STR (array vector)	FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SME2 lookup table load/store

The encodings in this section are decoded from SME Memory.

31–22	21–16	15	14–10	9–5	4–2	1–0
1 1 1 0 0 0 0 1 0 0	opc	1	0 0 0 0 0	Rn	0 0 0	opc2

opc	opc2	Instruction Details	Condition
x11111	!=00	UNALLOCATED	-
!=x11111	xx	UNALLOCATED	-
011111	00	LDR (table)	FEAT_SME2
111111	00	STR (table)	FEAT_SME2

SVE encodings

SVE encodings

The encodings in this section are decoded from A64 instruction set encoding.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	1 1	imm5	1 0 1	Pg	Zn	Zt

op0	op1	op2	op3	Instruction details	Condition
000	0xx0xxxx	000xxx	x	SVE Integer Binary Arithmetic - Predicated	-
000	0xx0xxxx	001xxx	x	SVE Integer Reduction	-
000	0xx0xxxx	100xxx	x	SVE Bitwise Shift - Predicated	-
000	0xx0xxxx	101xxx	x	SVE Integer Unary Arithmetic - Predicated	-
000	0xx0xxxx	x1xxxx	x	SVE Integer Multiply-Add - Predicated	-
000	0xx1xxxx	001xxx	x	SVE Bitwise Logical - Unpredicated	-
000	0xx1xxxx	0100xx	x	SVE Index Generation	-
000	0xx1xxxx	0101xx	x	SVE Stack Allocation	-
000	0xx1xxxx	011xxx	x	SVE2 Integer Arithmetic - Unpredicated	-
000	0xx1xxxx	100xxx	x	SVE Bitwise Shift - Unpredicated	-
000	0xx1xxxx	1011xx	x	SVE Integer Misc - Unpredicated	-
000	0xx1xxxx	11xxxx	x	SVE Element Count	-
000	10x1xxxx	000xxx	x	SVE Permute Vector - Extract	-
000	11x1xxxx	000xxx	x	SVE Permute Vector - Segments	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
000	1xx00xxx	xxxxxx	x	SVE Bitwise Immediate	-
000	1xx01xxx	xxxxxx	x	SVE Integer Wide Immediate - Predicated	-
000	1xx1xxxx	001001	x	SVE Permute Vector - One Source Quadwords	-
000	1xx1xxxx	001110	x	SVE Permute Vector - Unpredicated	-
000	1xx1xxxx	001111	x	UNALLOCATED	-
000	1xx1xxxx	010xxx	x	SVE Permute Predicate	-
000	1xx1xxxx	10xxxx	x	SVE Permute Vector - Predicated	-
001	0xx0xxxx	xxxxxx	x	SVE Integer Compare - Vectors	-
001	1xx00xxx	11xxxx	x	SVE Propagate Break	-
001	1xx01xxx	01xxxx	x	SVE Partition Break	-
001	1xx01xxx	11xxxx	x	SVE Predicate Misc	-
001	1xx100xx	10xxxx	x	SVE Predicate Count	-
001	1xx101xx	1000xx	x	SVE Inc/Dec by Predicate Count	-
001	1xx101xx	1001xx	x	SVE Write FFR	-
001	1xx101xx	101xxx	x	UNALLOCATED	-
001	1xx11xxx	10xxxx	x	UNALLOCATED	-
001	1xx1xxxx	00xxxx	x	SVE Integer Compare - Scalars	-
001	1xx1xxxx	01xxxx	1	SVE Scalar Integer Compare - Predicate-as-counter	-
001	1xx1xxxx	11xxxx	x	SVE Integer Wide Immediate - Unpredicated	-
010	0x10xxxx	11001x	x	UNALLOCATED	-
010	0xx0xxxx	0xxxxx	x	SVE Integer Multiply-Add - Unpredicated	-
010	0xx0xxxx	10xxxx	x	SVE2 Integer - Predicated	-
010	0xx0xxxx	1101x1	x	UNALLOCATED	-
010	0xx1xxxx	xxxxxx	x	SVE Multiply - Indexed	-
010	1xx0xxxx	0xxxxx	x	SVE2 Widening Integer Arithmetic	-
010	1xx0xxxx	10xxxx	x	SVE Misc	-
010	1xx0xxxx	11xxxx	x	SVE2 Accumulate	-
010	1xx1xxxx	0xxxxx	x	SVE2 Narrowing	-
010	1xx1xxxx	101xxx	x	SVE2 Histogram (Segment) and Lookup Table	-
010	1xx1xxxx	111xxx	x	SVE2 Crypto Extensions	-
011	01x1xxxx	111000	x	UNALLOCATED	-
011	0x01xxxx	0111xx	x	UNALLOCATED	-
011	0x01xxxx	10xx10	x	SVE2 FP8 widening multiply-add	-
011	0x01xxxx	10xx11	x	UNALLOCATED	-
011	0x01xxxx	11x1xx	x	UNALLOCATED	-
011	0x11xxxx	10xx1x	x	UNALLOCATED	-
011	0x11xxxx	x1x1xx	x	UNALLOCATED	-
011	0xx00001	100xxx	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
011	0xx0010x	100xxx	x	UNALLOCATED	-
011	0xx00x0x	11xxxx	x	UNALLOCATED	-
011	0xx00x1x	1xxxxx	x	UNALLOCATED	-
011	0xx010xx	11xxxx	x	UNALLOCATED	-
011	0xx011xx	1xxxxx	x	SVE2 floating-point unary operations - zeroing predicated	-
011	0xx1xxxx	001011	x	UNALLOCATED	-
011	0xx1xxxx	0011xx	x	UNALLOCATED	-
011	0xx1xxxx	01x0xx	x	SVE floating-point widening multiply-add - indexed	-
011	0xx1xxxx	10x00x	x	SVE floating-point widening multiply-add	-
011	0xx1xxxx	10x10x	x	UNALLOCATED	-
011	0xx1xxxx	11101x	x	UNALLOCATED	-
011	1xx001xx	0010xx	x	UNALLOCATED	-
011	1xx001xx	0011xx	x	SVE floating-point unary operations - unpredicated	-
011	1xx010xx	001xxx	x	SVE floating-point compare - with zero	-
011	1xx011xx	001xxx	x	SVE floating-point accumulating reduction	-
011	1xx0xxxx	100xxx	x	SVE floating-point arithmetic - predicated	-
011	1xx0xxxx	101xxx	x	SVE floating-point unary operations - merging predicated	-
011	1xx1xxxx	xxxxxx	x	SVE floating-point multiply-add	-
100	xxxxxxxx	xxxxxx	x	SVE Memory - 32-bit Gather and Unsized Contiguous	-
101	xxxxxxxx	xxxxxx	x	SVE Memory - Contiguous Load	-
110	xxxxxxxx	xxxxxx	x	SVE Memory - 64-bit Gather	-
111	xxxxxxxx	001xxx	x	SVE Memory - Non-temporal and Quadword Scatter Store	-
111	xxxxxxxx	011xxx	x	SVE Memory - Non-temporal and Multi-register Contiguous Store	-
111	xxxxxxxx	0x0xxx	x	SVE Memory - Contiguous Store and Unsized Contiguous	-
111	xxxxxxxx	101xxx	x	SVE Memory - Scatter	-
111	xxxxxxxx	111xxx	x	SVE Memory - Contiguous Store with Immediate Offset	-
111	xxxxxxxx	1x0xxx	x	SVE Memory - Scatter with Optional Sign Extend	-
000	0xx1xxxx	000xxx	x	SVE integer add/subtract vectors (unpredicated)	-
000	0xx1xxxx	1010xx	x	SVE address generation	-
000	1xx1xxxx	001000	x	DUP (indexed)	FEAT_SVE || FEAT_SME
000	1xx1xxxx	00101x	x	SVE table lookup (three sources)	-
000	1xx1xxxx	001100	x	TBL — Single register table	FEAT_SVE || FEAT_SME
000	1xx1xxxx	001101	x	TBXQ	FEAT_SVE2p1 || FEAT_SME2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
000	1xx1xxxx	011xxx	x	SVE permute vector elements	-
000	1xx1xxxx	11xxxx	x	SEL (vectors)	FEAT_SVE || FEAT_SME
001	0xx1xxxx	xxxxxx	x	SVE integer compare with unsigned immediate	-
001	1xx00xxx	01xxxx	x	SVE predicate logical operations	-
001	1xx0xxxx	x0xxxx	x	SVE integer compare with signed immediate	-
001	1xx1xxxx	01xxxx	0	SVE broadcast predicate element	-
010	0000xxxx	11001x	x	SVE two-way dot product	-
010	0100xxxx	11001x	x	SVE two-way dot product (indexed)	-
010	0xx0xxxx	11000x	x	SVE integer clamp	-
010	0xx0xxxx	1101x0	x	SVE2 multiply-add (checked pointer)	-
010	0xx0xxxx	111xxx	x	SVE permute vector elements (quadwords)	-
010	1xx1xxxx	100xxx	x	SVE2 character match	-
010	1xx1xxxx	110xxx	x	HISTCNT	FEAT_SVE2
011	00x1xxxx	111000	x	SVE2 FP8 matrix multiply-accumulate	-
011	0x01xxxx	0101xx	x	SVE2 FP8 multiply-add long (indexed)	-
011	0xx00000	100xxx	x	FCADD	FEAT_SVE || FEAT_SME
011	0xx0000x	101xxx	x	SVE floating-point convert (top, predicated)	-
011	0xx0010x	101xxx	x	SVE floating-point convert precision odd elements	-
011	0xx010xx	100xxx	x	SVE2 floating-point pairwise operations	-
011	0xx010xx	101xxx	x	SVE floating-point recursive reduction (quadwords)	-
011	0xx0xxxx	0xxxxx	x	FCMLA (vectors)	FEAT_SVE || FEAT_SME
011	0xx1xxxx	0000xx	x	SVE floating-point multiply-add (indexed)	-
011	0xx1xxxx	0001xx	x	SVE floating-point complex multiply-add (indexed)	-
011	0xx1xxxx	001001	x	SVE FP clamp	-
011	0xx1xxxx	0010x0	x	SVE floating-point multiply (indexed)	-
011	0xx1xxxx	1100xx	x	SVE2 FP8 multiply-add long long (indexed)	-
011	0xx1xxxx	111001	x	SVE floating-point matrix multiply accumulate	-
011	1xx000xx	001xxx	x	SVE floating-point recursive reduction	-
011	1xx0xxxx	000xxx	x	SVE floating-point arithmetic (unpredicated)	-
011	1xx0xxxx	x1xxxx	x	SVE floating-point compare vectors	-

SVE Integer Binary Arithmetic - Predicated

SVE Integer Binary Arithmetic - Predicated

The encodings in this section are decoded from SVE encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20		18	17	16	15			13	12											0
0	0	0	0	0	1	0	0			0	op0					0	0	0													

op0	Instruction details
00x	SVE integer add/subtract vectors (predicated)
01x	SVE integer min/max/difference (predicated)
100	SVE integer multiply vectors (predicated)
101	SVE integer divide vectors (predicated)
11x	SVE bitwise logical operations (predicated)

SVE integer add/subtract vectors (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21			19	18			16	15			13	12			10	9				5	4					0
0	0	0	0	0	1	0	0	size		0	0	0		opc				0	0	0		Pg				Zm					Zdn					

size	opc	Instruction Details	Condition
xx	000	ADD (vectors, predicated)	FEAT_SVE || FEAT_SME
xx	001	SUB (vectors, predicated)	FEAT_SVE || FEAT_SME
xx	010	UNALLOCATED	-
xx	011	SUBR (vectors)	FEAT_SVE || FEAT_SME
11	100	ADDPT (predicated)	FEAT_SVE && FEAT_CPA
11	101	SUBPT (predicated)	FEAT_SVE && FEAT_CPA
11	11x	UNALLOCATED	-
!= 11	1xx	UNALLOCATED	-

SVE integer min/max/difference (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21			19	18	17	16	15			13	12			10	9				5	4					0
0	0	0	0	0	1	0	0	size		0	0	1		opc		U	0	0	0		Pg				Zm					Zdn					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	U	Instruction Details	Condition
00	0	SMAX (vectors)	FEAT_SVE || FEAT_SME
00	1	UMAX (vectors)	FEAT_SVE || FEAT_SME
01	0	SMIN (vectors)	FEAT_SVE || FEAT_SME
01	1	UMIN (vectors)	FEAT_SVE || FEAT_SME
10	0	SABD	FEAT_SVE || FEAT_SME
10	1	UABD	FEAT_SVE || FEAT_SME
11	x	UNALLOCATED	-

SVE integer multiply vectors (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21	20			18	17	16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	1	0	0		H	U	0	0	0		Pg			Zm			Zdn		

H	U	Instruction Details	Condition
0	0	MUL (vectors, predicated)	FEAT_SVE || FEAT_SME
0	1	UNALLOCATED	-
1	0	SMULH (predicated)	FEAT_SVE || FEAT_SME
1	1	UMULH (predicated)	FEAT_SVE || FEAT_SME

SVE integer divide vectors (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21	20			18	17	16	15			13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	1	0	1		R	U	0	0	0		Pg			Zm			Zdn		

R	U	Instruction Details	Condition
0	0	SDIV	FEAT_SVE || FEAT_SME
0	1	UDIV	FEAT_SVE || FEAT_SME
1	0	SDIVR	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R	U	Instruction Details	Condition
1	1	UDIVR	FEAT_SVE || FEAT_SME

SVE bitwise logical operations (predicated)

The encodings in this section are decoded from SVE Integer Binary Arithmetic - Predicated.

31							24	23	22	21		19	18		16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0	size		0	1	1	opc			0	0	0	Pg			Zm			Zdn		

opc	Instruction Details	Condition
000	ORR (vectors, predicated)	FEAT_SVE || FEAT_SME
001	EOR (vectors, predicated)	FEAT_SVE || FEAT_SME
010	AND (vectors, predicated)	FEAT_SVE || FEAT_SME
011	BIC (vectors, predicated)	FEAT_SVE || FEAT_SME
1xx	UNALLOCATED	-

SVE Integer Reduction

SVE Integer Reduction

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20		18	17	16	15		13	12		0
0	0	0	0	0	1	0	0			0	op0					0	0	1			

op0	Instruction details
000	SVE integer add reduction (predicated)
001	SVE integer add reduction (quadwords)
010	SVE integer min/max reduction (predicated)
011	SVE integer min/max reduction (quadwords)
10x	SVE constructive prefix (predicated)
110	SVE bitwise logical reduction (predicated)
111	SVE bitwise logical reduction (quadwords)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE integer add reduction (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31–24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0 0 0 1 0 0		0	0 0 0	op	U	0 0 1	Pg	Zn	Vd

op	U	Instruction Details	Condition
0	0	SADDV	FEAT_SVE || FEAT_SME
0	1	UADDV	FEAT_SVE || FEAT_SME
1	x	UNALLOCATED	-

SVE integer add reduction (quadwords)

The encodings in this section are decoded from SVE Integer Reduction.

31–24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0 0 0 1 0 0		0	0 0 1	op	U	0 0 1	Pg	Zn	Vd

op	U	Instruction Details	Condition
0	0	UNALLOCATED	-
0	1	ADDQV	FEAT_SVE2p1 || FEAT_SME2p1
1	x	UNALLOCATED	-

SVE integer min/max reduction (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31–24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0 0 0 1 0 0		0	0 1 0	op	U	0 0 1	Pg	Zn	Vd

op	U	Instruction Details	Condition
0	0	SMAXV	FEAT_SVE || FEAT_SME
0	1	UMAXV	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	Instruction Details	Condition
1	0	SMINV	FEAT_SVE || FEAT_SME
1	1	UMINV	FEAT_SVE || FEAT_SME

SVE integer min/max reduction (quadwords)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20			18	17	16	15			13	12			10	9					5	4					0
0	0	0	0	0	1	0	0			0	0	1	1		op	U	0	0	1		Pg				Zn						Vd					

op	U	Instruction Details	Condition
0	0	SMAXQV	FEAT_SVE2p1 || FEAT_SME2p1
0	1	UMAXQV	FEAT_SVE2p1 || FEAT_SME2p1
1	0	SMINQV	FEAT_SVE2p1 || FEAT_SME2p1
1	1	UMINQV	FEAT_SVE2p1 || FEAT_SME2p1

SVE constructive prefix (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21			19	18		17	16	15			13	12			10	9					5	4					0
0	0	0	0	0	1	0	0			0	1	0			opc		M	0	0	1		Pg				Zn						Zd					

opc	Instruction Details	Condition
00	MOVPRFX (predicated)	FEAT_SVE || FEAT_SME
!= 00	UNALLOCATED	-

SVE bitwise logical reduction (predicated)

The encodings in this section are decoded from SVE Integer Reduction.

31							24	23	22	21	20			18	17		16	15			13	12			10	9					5	4					0
0	0	0	0	0	1	0	0			0	1	1	0		opc			0	0	1		Pg				Zn						Vd					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
00	ORV	FEAT_SVE || FEAT_SME
01	EORV	FEAT_SVE || FEAT_SME
10	ANDV	FEAT_SVE || FEAT_SME
11	UNALLOCATED	-

SVE bitwise logical reduction (quadwords)

The encodings in this section are decoded from SVE Integer Reduction.

31	24	23	22	21	20	18	17	16	15	13	12	10	9	5	4	0
0 0 0 0 0 1 0 0				0	1 1 1		opc		0 0 1		Pg		Zn		Vd	

opc	Instruction Details	Condition
00	ORQV	FEAT_SVE2p1 || FEAT_SME2p1
01	EORQV	FEAT_SVE2p1 || FEAT_SME2p1
10	ANDQV	FEAT_SVE2p1 || FEAT_SME2p1
11	UNALLOCATED	-

SVE Bitwise Shift - Predicated

SVE Bitwise Shift - Predicated

The encodings in this section are decoded from SVE encodings.

31	24	23	22	21	20	19	18	16	15	13	12	0
0 0 0 0 0 1 0 0				0	op0				1 0 0			

op0	Instruction details
0x	SVE bitwise shift by immediate (predicated)
10	SVE bitwise shift by vector (predicated)
11	SVE bitwise shift by wide elements (predicated)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE bitwise shift by immediate (predicated)

The encodings in this section are decoded from SVE Bitwise Shift - Predicated.

31							24	23	22	21	20	19	18	17	16	15		13	12		10	9	8	7		5	4		0
0	0	0	0	0	1	0	0	tszh		0	0	opc		L	U	1	0	0	Pg			tszl		imm3			Zdn		

opc	L	U	Instruction Details	Condition
00	0	0	ASR (immediate, predicated)	FEAT_SVE || FEAT_SME
00	0	1	LSR (immediate, predicated)	FEAT_SVE || FEAT_SME
00	1	0	UNALLOCATED	-
00	1	1	LSL (immediate, predicated)	FEAT_SVE || FEAT_SME
01	0	0	ASRD	FEAT_SVE || FEAT_SME
01	0	1	UNALLOCATED	-
01	1	0	SQSHL (immediate)	FEAT_SVE2 || FEAT_SME
01	1	1	UQSHL (immediate)	FEAT_SVE2 || FEAT_SME
10	x	x	UNALLOCATED	-
11	0	0	SRSHR	FEAT_SVE2 || FEAT_SME
11	0	1	URSHR	FEAT_SVE2 || FEAT_SME
11	1	0	UNALLOCATED	-
11	1	1	SQSHLU	FEAT_SVE2 || FEAT_SME

SVE bitwise shift by vector (predicated)

The encodings in this section are decoded from SVE Bitwise Shift - Predicated.

31							24	23	22	21	20	19	18	17	16	15		13	12		10	9		5	4		0
0	0	0	0	0	1	0	0			0	1	0	R	L	U	1	0	0	Pg			Zm			Zdn		

R	L	U	Instruction Details	Condition
x	1	0	UNALLOCATED	-
0	0	0	ASR (vectors)	FEAT_SVE || FEAT_SME
0	0	1	LSR (vectors)	FEAT_SVE || FEAT_SME
0	1	1	LSL (vectors)	FEAT_SVE || FEAT_SME
1	0	0	ASRR	FEAT_SVE || FEAT_SME
1	0	1	LSRR	FEAT_SVE || FEAT_SME
1	1	1	LSLR	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE bitwise shift by wide elements (predicated)

The encodings in this section are decoded from SVE Bitwise Shift - Predicated.

31–24	23–22	21	20–19	18	17	16	15–13	12–10	9–5	4–0
0 0 0 0 0 1 0 0		0	1 1	R	L	U	1 0 0	Pg	Zm	Zdn

R	L	U	Instruction Details	Condition
0	0	0	ASR (wide elements, predicated)	FEAT_SVE || FEAT_SME
0	0	1	LSR (wide elements, predicated)	FEAT_SVE || FEAT_SME
0	1	0	UNALLOCATED	-
0	1	1	LSL (wide elements, predicated)	FEAT_SVE || FEAT_SME
1	x	x	UNALLOCATED	-

SVE Integer Unary Arithmetic - Predicated

SVE Integer Unary Arithmetic - Predicated

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21	20–19	18–16	15–13	12–0
0 0 0 0 0 1 0 0		0	op0		1 0 1	

op0	Instruction details
x0	SVE integer unary operations (predicated)
x1	SVE bitwise unary operations (predicated)

SVE integer unary operations (predicated)

The encodings in this section are decoded from SVE Integer Unary Arithmetic - Predicated.

31–24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0 0 0 1 0 0	size	0	M	0	opc	1 0 1	Pg	Zn	Zd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	opc	Instruction Details	Condition
0	000	SXTB, SXTH, SXTW — Byte, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	001	UXTB, UXTH, UXTW — Byte, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	010	SXTB, SXTH, SXTW — Halfword, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	011	UXTB, UXTH, UXTW — Halfword, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	100	SXTB, SXTH, SXTW — Word, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	101	UXTB, UXTH, UXTW — Word, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	110	ABS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	111	NEG — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	000	SXTB, SXTH, SXTW — Byte, merging	FEAT_SVE || FEAT_SME
1	001	UXTB, UXTH, UXTW — Byte, merging	FEAT_SVE || FEAT_SME
1	010	SXTB, SXTH, SXTW — Halfword, merging	FEAT_SVE || FEAT_SME
1	011	UXTB, UXTH, UXTW — Halfword, merging	FEAT_SVE || FEAT_SME
1	100	SXTB, SXTH, SXTW — Word, merging	FEAT_SVE || FEAT_SME
1	101	UXTB, UXTH, UXTW — Word, merging	FEAT_SVE || FEAT_SME
1	110	ABS — Merging	FEAT_SVE || FEAT_SME
1	111	NEG — Merging	FEAT_SVE || FEAT_SME

SVE bitwise unary operations (predicated)

The encodings in this section are decoded from SVE Integer Unary Arithmetic - Predicated.

31	24	23 22	21	20	19	18 16	15 13	12 10	9 5	4 0
0 0 0 0 0 1 0 0		size	0	M	1	opc	1 0 1	Pg	Zn	Zd

M	opc	Instruction Details	Condition
x	111	UNALLOCATED	-
0	000	CLS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	001	CLZ — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	010	CNT — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	011	CNOT — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	100	FABS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	101	FNEG — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	110	NOT (vector) — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	000	CLS — Merging	FEAT_SVE || FEAT_SME
1	001	CLZ — Merging	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	opc	Instruction Details	Condition
1	010	CNT — Merging	FEAT_SVE || FEAT_SME
1	011	CNOT — Merging	FEAT_SVE || FEAT_SME
1	100	FABS — Merging	FEAT_SVE || FEAT_SME
1	101	FNEG — Merging	FEAT_SVE || FEAT_SME
1	110	NOT (vector) — Merging	FEAT_SVE || FEAT_SME

SVE Integer Multiply-Add - Predicated

SVE Integer Multiply-Add - Predicated

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20	16	15	14	13	0
0	0	0	0	0	1	0	0			0			op0	1		

op0	Instruction details
0	SVE integer multiply-accumulate writing addend (predicated)
1	SVE integer multiply-add writing multiplicand (predicated)

SVE integer multiply-accumulate writing addend (predicated)

The encodings in this section are decoded from SVE Integer Multiply-Add - Predicated.

31							24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
0	0	0	0	0	1	0	0	size		0	Zm		0	1	op	Pg		Zn		Zda	

op	Instruction Details	Condition
0	MLA (vectors)	FEAT_SVE || FEAT_SME
1	MLS (vectors)	FEAT_SVE || FEAT_SME

SVE integer multiply-add writing multiplicand (predicated)

The encodings in this section are decoded from SVE Integer Multiply-Add - Predicated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
0 0 0 0 0 1 0 0	size	0	Zm	1	1	op	Pg	Za	Zdn

op	Instruction Details	Condition
0	MAD	FEAT_SVE || FEAT_SME
1	MSB	FEAT_SVE || FEAT_SME

SVE Bitwise Logical - Unpredicated

SVE Bitwise Logical - Unpredicated

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23	22	21	20–18	17–16	15–14	13–10	9	8–5	4	3–0
0 0 1	0 0 1 0 1	i1	tszh	1	tszl	Rv	0 1	Pn	S	Pm	0	Pd

op0	Instruction details	Condition
0xx	UNALLOCATED	-
100	SVE bitwise logical operations (unpredicated)	-
101	XAR	FEAT_SVE2 || FEAT_SME
11x	SVE2 bitwise ternary operations	-

SVE bitwise logical operations (unpredicated)

The encodings in this section are decoded from SVE Bitwise Logical - Unpredicated.

31–24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 0 0 0 0 1 0 0	opc	1	Zm	0 0 1	1 0 0	Zn	Zd

opc	Instruction Details	Condition
00	AND (vectors, unpredicated)	FEAT_SVE || FEAT_SME
01	ORR (vectors, unpredicated)	FEAT_SVE || FEAT_SME
10	EOR (vectors, unpredicated)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
11	BIC (vectors, unpredicated)	FEAT_SVE || FEAT_SME

SVE2 bitwise ternary operations

The encodings in this section are decoded from SVE Bitwise Logical - Unpredicated.

31–24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0 0 0 1 0 0	opc	1	Zm	0 0 1 1 1	o2	Zk	Zdn

opc	o2	Instruction Details	Condition
00	0	EOR3	FEAT_SVE2 || FEAT_SME
00	1	BSL	FEAT_SVE2 || FEAT_SME
01	0	BCAX	FEAT_SVE2 || FEAT_SME
01	1	BSL1N	FEAT_SVE2 || FEAT_SME
1x	0	UNALLOCATED	-
10	1	BSL2N	FEAT_SVE2 || FEAT_SME
11	1	NBSL	FEAT_SVE2 || FEAT_SME

SVE Index Generation

SVE Index Generation

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21	20–16	15–12	11–10	9–0
0 0 0 0 0 1 0 0		1		0 1 0 0	op0	

op0	Instruction details	Condition
00	INDEX (immediates)	FEAT_SVE || FEAT_SME
01	INDEX (scalar, immediate)	FEAT_SVE || FEAT_SME
10	INDEX (immediate, scalar)	FEAT_SVE || FEAT_SME
11	INDEX (scalars)	FEAT_SVE || FEAT_SME

SVE Stack Allocation

SVE Stack Allocation

The encodings in this section are decoded from SVE encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20				16	15			12	11	10								0
0	0	0	0	0	1	0	0	op0		1						0	1	0	1	op1									

op0	op1	Instruction details
0	0	SVE stack frame adjustment
0	1	Streaming SVE stack frame adjustment
1	0	SVE stack frame size
1	1	Streaming SVE stack frame size

SVE stack frame adjustment

The encodings in this section are decoded from SVE Stack Allocation.

31							24	23	22	21	20				16	15			12	11	10				5	4			0
0	0	0	0	0	1	0	0	0	op	1			Rn			0	1	0	1	0			imm6				Rd		

op	Instruction Details	Condition
0	ADDVL	FEAT_SVE || FEAT_SME
1	ADDPL	FEAT_SVE || FEAT_SME

Streaming SVE stack frame adjustment

The encodings in this section are decoded from SVE Stack Allocation.

31							24	23	22	21	20				16	15			12	11	10				5	4			0
0	0	0	0	0	1	0	0	0	op	1			Rn			0	1	0	1	1			imm6				Rd		

op	Instruction Details	Condition
0	ADDSVL	FEAT_SME
1	ADDSPL	FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE stack frame size

The encodings in this section are decoded from SVE Stack Allocation.

31–24	23	22	21	20–16	15–12	11	10–5	4–0
0 0 0 0 0 1 0 0	1	op	1	opc2	0 1 0 1	0	imm6	Rd

op	opc2	Instruction Details	Condition
0	11111	RDVL	FEAT_SVE || FEAT_SME
0	!= 11111	UNALLOCATED	-
1	xxxxx	UNALLOCATED	-

Streaming SVE stack frame size

The encodings in this section are decoded from SVE Stack Allocation.

31–24	23	22	21	20–16	15–12	11	10–5	4–0
0 0 0 0 0 1 0 0	1	op	1	opc2	0 1 0 1	1	imm6	Rd

op	opc2	Instruction Details	Condition
0	11111	RDSVL	FEAT_SME
0	!= 11111	UNALLOCATED	-
1	xxxxx	UNALLOCATED	-

SVE2 Integer Arithmetic - Unpredicated

SVE2 Integer Arithmetic - Unpredicated

The encodings in this section are decoded from SVE encodings.

31–25	24	23–22	21	20–16	15–13	12–10	9–0
0 0 0 0 0 1 0	0		1		0 1 1	op0	

op0	Instruction details
0xx	SVE2 integer multiply vectors (unpredicated)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
10x	SVE2 signed saturating doubling multiply high (unpredicated)
11x	UNALLOCATED

SVE2 integer multiply vectors (unpredicated)

The encodings in this section are decoded from SVE2 Integer Arithmetic - Unpredicated.

31							24	23	22	21	20				16	15			12	11	10	9				5	4				0
0	0	0	0	0	1	0	0	size		1	Zm					0	1	1	0	opc		Zn					Zd				

size	opc	Instruction Details	Condition
xx	00	MUL (vectors, unpredicated)	FEAT_SVE2 || FEAT_SME
xx	10	SMULH (unpredicated)	FEAT_SVE2 || FEAT_SME
xx	11	UMULH (unpredicated)	FEAT_SVE2 || FEAT_SME
00	01	PMUL	FEAT_SVE2 || FEAT_SME
!= 00	01	UNALLOCATED	-

SVE2 signed saturating doubling multiply high (unpredicated)

The encodings in this section are decoded from SVE2 Integer Arithmetic - Unpredicated.

31							24	23	22	21	20				16	15				11	10	9				5	4				0
0	0	0	0	0	1	0	0	size		1	Zm					0	1	1	1	0	R	Zn					Zd				

R	Instruction Details	Condition
0	SQDMULH (vectors)	FEAT_SVE2 || FEAT_SME
1	SQRDMULH (vectors)	FEAT_SVE2 || FEAT_SME

SVE Bitwise Shift - Unpredicated

SVE Bitwise Shift - Unpredicated

The encodings in this section are decoded from SVE encodings.

31							25	24	23	22	21	20				16	15			13	12	11										0
0	0	0	0	0	1	0		0			1						1	0		0	op0											

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE bitwise shift by wide elements (unpredicated)
1	SVE bitwise shift by immediate (unpredicated)

SVE bitwise shift by wide elements (unpredicated)

The encodings in this section are decoded from SVE Bitwise Shift - Unpredicated.

31							24	23	22	21	20				16	15			13	12	11	10	9				5	4			0
0	0	0	0	0	1	0	0			1	Zm					1	0	0		0	opc		Zn					Zd			

opc	Instruction Details	Condition
00	ASR (wide elements, unpredicated)	FEAT_SVE || FEAT_SME
01	LSR (wide elements, unpredicated)	FEAT_SVE || FEAT_SME
10	UNALLOCATED	-
11	LSL (wide elements, unpredicated)	FEAT_SVE || FEAT_SME

SVE bitwise shift by immediate (unpredicated)

The encodings in this section are decoded from SVE Bitwise Shift - Unpredicated.

31							24	23	22	21	20	19	18			16	15			13	12	11	10	9				5	4			0
0	0	0	0	0	1	0	0	tszh		1	tszl		imm3				1	0	0		1	opc		Zn					Zd			

opc	Instruction Details	Condition
00	ASR (immediate, unpredicated)	FEAT_SVE || FEAT_SME
01	LSR (immediate, unpredicated)	FEAT_SVE || FEAT_SME
10	UNALLOCATED	-
11	LSL (immediate, unpredicated)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE Integer Misc - Unpredicated

SVE Integer Misc - Unpredicated

The encodings in this section are decoded from SVE encodings.

31–25	24	23–22	21	20–16	15–12	11–10	9–0
0 0 0 0 0 1 0	0		1		1 0 1 1	op0	

op0	Instruction details
0x	SVE floating-point trig select coefficient
10	SVE floating-point exponential accelerator
11	SVE constructive prefix (unpredicated)

SVE floating-point trig select coefficient

The encodings in this section are decoded from SVE Integer Misc - Unpredicated.

31–24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0 0 0 1 0 0		1	Zm	1 0 1 1 0	op	Zn	Zd

op	Instruction Details	Condition
0	FTSSEL	FEAT_SVE
1	UNALLOCATED	-

SVE floating-point exponential accelerator

The encodings in this section are decoded from SVE Integer Misc - Unpredicated.

31–29	28–24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0 0		0 1 0	opc	1 0 0	Pg	Zm	Zdn

opc	Instruction Details	Condition
00000	FEXPA	FEAT_SVE || FEAT_SSVE_FEXPA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
!= 00000	UNALLOCATED	-

SVE constructive prefix (unpredicated)

The encodings in this section are decoded from SVE Integer Misc - Unpredicated.

31	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0 0 0 1 0 0		opc	1	opc2	1 0 1 1	1 1	Zn	Zd

opc	opc2	Instruction Details	Condition
00	00000	MOVPRFX (unpredicated)	FEAT_SVE || FEAT_SME
00	!= 00000	UNALLOCATED	-
!= 00	xxxxx	UNALLOCATED	-

SVE Element Count

SVE Element Count

The encodings in this section are decoded from SVE encodings.

31	24	23 22	21	20	19 16	15 14	13 11	10 0
0 0 0 0 0 1 0 0			1	op0		1 1	op1	

op0	op1	Instruction details
0	00x	SVE saturating inc/dec vector by element count
0	100	SVE element count
0	101	UNALLOCATED
1	000	SVE inc/dec vector by element count
1	100	SVE inc/dec register by element count
1	x01	UNALLOCATED
x	01x	UNALLOCATED
x	11x	SVE saturating inc/dec register by element count

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE saturating inc/dec vector by element count

The encodings in this section are decoded from SVE Element Count.

31							24	23	22	21	20	19			16	15			12	11	10	9			5	4			0
0	0	0	0	0	1	0	0	size		1	0	imm4				1	1	0	0	D	U	pattern				Zdn			

size	D	U	Instruction Details	Condition
00	x	x	UNALLOCATED	-
01	0	0	SQINCH (vector)	FEAT_SVE || FEAT_SME
01	0	1	UQINCH (vector)	FEAT_SVE || FEAT_SME
01	1	0	SQDECH (vector)	FEAT_SVE || FEAT_SME
01	1	1	UQDECH (vector)	FEAT_SVE || FEAT_SME
10	0	0	SQINCW (vector)	FEAT_SVE || FEAT_SME
10	0	1	UQINCW (vector)	FEAT_SVE || FEAT_SME
10	1	0	SQDECW (vector)	FEAT_SVE || FEAT_SME
10	1	1	UQDECW (vector)	FEAT_SVE || FEAT_SME
11	0	0	SQINCD (vector)	FEAT_SVE || FEAT_SME
11	0	1	UQINCD (vector)	FEAT_SVE || FEAT_SME
11	1	0	SQDECD (vector)	FEAT_SVE || FEAT_SME
11	1	1	UQDECD (vector)	FEAT_SVE || FEAT_SME

SVE element count

The encodings in this section are decoded from SVE Element Count.

31							24	23	22	21	20	19			16	15	14	13			11	10	9			5	4			0
0	0	0	0	0	1	0	0	size		1	0	imm4				1	1	1	0	0		op	pattern				Rd			

size	op	Instruction Details	Condition
xx	1	UNALLOCATED	-
00	0	CNTB, CNTD, CNTH, CNTW — Byte	FEAT_SVE || FEAT_SME
01	0	CNTB, CNTD, CNTH, CNTW — Halfword	FEAT_SVE || FEAT_SME
10	0	CNTB, CNTD, CNTH, CNTW — Word	FEAT_SVE || FEAT_SME
11	0	CNTB, CNTD, CNTH, CNTW — Doubleword	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE inc/dec vector by element count

The encodings in this section are decoded from SVE Element Count.

31–24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0 0 0 1 0 0	size	1	1	imm4	1 1	0 0 0	D	pattern	Zdn

size	D	Instruction Details	Condition
00	x	UNALLOCATED	-
01	0	INCD, INCH, INCW (vector) — Halfword	FEAT_SVE || FEAT_SME
01	1	DECD, DECH, DECW (vector) — Halfword	FEAT_SVE || FEAT_SME
10	0	INCD, INCH, INCW (vector) — Word	FEAT_SVE || FEAT_SME
10	1	DECD, DECH, DECW (vector) — Word	FEAT_SVE || FEAT_SME
11	0	INCD, INCH, INCW (vector) — Doubleword	FEAT_SVE || FEAT_SME
11	1	DECD, DECH, DECW (vector) — Doubleword	FEAT_SVE || FEAT_SME

SVE inc/dec register by element count

The encodings in this section are decoded from SVE Element Count.

31–24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0 0 0 1 0 0	size	1	1	imm4	1 1	1 0 0	D	pattern	Rdn

size	D	Instruction Details	Condition
00	0	INCB, INCD, INCH, INCW (scalar) — Byte	FEAT_SVE || FEAT_SME
00	1	DECB, DECD, DECH, DECW (scalar) — Byte	FEAT_SVE || FEAT_SME
01	0	INCB, INCD, INCH, INCW (scalar) — Halfword	FEAT_SVE || FEAT_SME
01	1	DECB, DECD, DECH, DECW (scalar) — Halfword	FEAT_SVE || FEAT_SME
10	0	INCB, INCD, INCH, INCW (scalar) — Word	FEAT_SVE || FEAT_SME
10	1	DECB, DECD, DECH, DECW (scalar) — Word	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	D	Instruction Details	Condition
11	0	INCB, INCD, INCH, INCW (scalar) — Doubleword	FEAT_SVE || FEAT_SME
11	1	DECB, DECD, DECH, DECW (scalar) — Doubleword	FEAT_SVE || FEAT_SME

SVE saturating inc/dec register by element count

The encodings in this section are decoded from SVE Element Count.

31							24	23	22	21	20	19			16	15			12	11	10	9			5	4			0
0	0	0	0	0	1	0	0	size		1	sf	imm4				1	1	1	1	D	U	pattern				Rdn			

size	sf	D	U	Instruction Details	Condition
00	0	0	0	SQINCB — 32-bit	FEAT_SVE || FEAT_SME
00	0	0	1	UQINCB — 32-bit	FEAT_SVE || FEAT_SME
00	0	1	0	SQDECB — 32-bit	FEAT_SVE || FEAT_SME
00	0	1	1	UQDECB — 32-bit	FEAT_SVE || FEAT_SME
00	1	0	0	SQINCB — 64-bit	FEAT_SVE || FEAT_SME
00	1	0	1	UQINCB — 64-bit	FEAT_SVE || FEAT_SME
00	1	1	0	SQDECB — 64-bit	FEAT_SVE || FEAT_SME
00	1	1	1	UQDECB — 64-bit	FEAT_SVE || FEAT_SME
01	0	0	0	SQINCH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	0	0	1	UQINCH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	0	1	0	SQDECH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	0	1	1	UQDECH (scalar) — 32-bit	FEAT_SVE || FEAT_SME
01	1	0	0	SQINCH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
01	1	0	1	UQINCH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
01	1	1	0	SQDECH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
01	1	1	1	UQDECH (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	0	0	0	SQINCW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	0	0	1	UQINCW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	0	1	0	SQDECW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	0	1	1	UQDECW (scalar) — 32-bit	FEAT_SVE || FEAT_SME
10	1	0	0	SQINCW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	1	0	1	UQINCW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	1	1	0	SQDECW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
10	1	1	1	UQDECW (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	0	0	0	SQINCD (scalar) — 32-bit	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	sf	D	U	Instruction Details	Condition
11	0	0	1	UQINCD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	0	1	0	SQDECD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	0	1	1	UQDECD (scalar) — 32-bit	FEAT_SVE || FEAT_SME
11	1	0	0	SQINCD (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	1	0	1	UQINCD (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	1	1	0	SQDECD (scalar) — 64-bit	FEAT_SVE || FEAT_SME
11	1	1	1	UQDECD (scalar) — 64-bit	FEAT_SVE || FEAT_SME

SVE Permute Vector - Extract

SVE Permute Vector - Extract

The encodings in this section are decoded from SVE encodings.

31								23	22	21	20			16	15			13	12													0
0	0	0	0	0	1	0	1	0	op0	1					0	0	0															

op0	Instruction details	Condition
0	EXT — Destructive	FEAT_SVE || FEAT_SME
1	EXT — Constructive	FEAT_SVE2 || FEAT_SME

SVE Permute Vector - Segments

SVE Permute Vector - Segments

The encodings in this section are decoded from SVE encodings.

31								23	22	21	20			16	15			13	12													0
0	0	0	0	0	1	0	1	1	op0	1					0	0	0															

op0	Instruction details
0	SVE permute vector segments
1	UNALLOCATED

SVE permute vector segments

The encodings in this section are decoded from SVE Permute Vector - Segments.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–23	22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0 0 0 1 0 1 1	0	1	Zm	0 0 0	opc	H	Zn	Zd

opc	H	Instruction Details	Condition
00	0	ZIP1, ZIP2 (vectors) — Low halves (quadwords)	FEAT_F64MM
00	1	ZIP1, ZIP2 (vectors) — High halves (quadwords)	FEAT_F64MM
01	0	UZP1, UZP2 (vectors) — Even (quadwords)	FEAT_F64MM
01	1	UZP1, UZP2 (vectors) — Odd (quadwords)	FEAT_F64MM
10	x	UNALLOCATED	-
11	0	TRN1, TRN2 (vectors) — Even (quadwords)	FEAT_F64MM
11	1	TRN1, TRN2 (vectors) — Odd (quadwords)	FEAT_F64MM

SVE Bitwise Immediate

SVE Bitwise Immediate

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21–20	19–18	17–0
0 0 0 0 0 1 0 1	op0	0 0	op1	

op0	op1	Instruction details	Condition
11	00	DUPM	FEAT_SVE || FEAT_SME
xx	!= 00	UNALLOCATED	-
!= 11	00	SVE bitwise logical with immediate (unpredicated)	-

SVE bitwise logical with immediate (unpredicated)

The encodings in this section are decoded from SVE Bitwise Immediate.

31–24	23–22	21–20	19–18	17–5	4–0
0 0 0 0 0 1 0 1	!= 11	0 0	0 0	imm13	Zdn
	opc				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following constraints also apply to this encoding: opc != '11'

opc	Instruction Details	Condition
00	ORR (immediate)	FEAT_SVE || FEAT_SME
01	EOR (immediate)	FEAT_SVE || FEAT_SME
10	AND (immediate)	FEAT_SVE || FEAT_SME

SVE Integer Wide Immediate - Predicated

SVE Integer Wide Immediate - Predicated

The encodings in this section are decoded from SVE encodings.

| 31 | | | | | | 25 | 24 | 23 | 22 | 21 | 20 | 19 | | | 16 | 15 | | | 13 | 12 | | | | | |

31–29	28–24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 1 1	0 0 1 0 1	size	0	Zm	op	1	o2	Pg	Zn	o3	Pd

op0	op1	Instruction details	Condition
00	x	DUPQ	FEAT_SVE2p1 || FEAT_SME2p1
01	0	EXTQ	FEAT_SVE2p1 || FEAT_SME2p1
01	1	UNALLOCATED	-
1x	x	UNALLOCATED	-

SVE Permute Vector - Unpredicated

SVE Permute Vector - Unpredicated

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21	20–19	18–16	15–10	9	8–5	4	3–0
0 0 0 0 0 1 0 1		1	op0	op1	0 0 1 1 1 0	op2		op3	

op0	op1	op2	op3	Instruction details	Condition
00	000	x	x	DUP (scalar)	FEAT_SVE || FEAT_SME
00	100	x	x	INSR (scalar)	FEAT_SVE || FEAT_SME
00	!= x00	x	x	UNALLOCATED	-
01	xx0	x	0	SVE move predicate from vector	-
01	xx0	x	1	UNALLOCATED	-
01	xx1	0	x	SVE move predicate into vector	-
01	xx1	1	x	UNALLOCATED	-
10	0xx	x	x	SVE unpack vector elements	-
10	100	x	x	INSR (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
10	101	x	x	UNALLOCATED	-
10	11x	x	x	UNALLOCATED	-
11	000	x	x	REV (vector)	FEAT_SVE || FEAT_SME
11	!= 000	x	x	UNALLOCATED	-

SVE move predicate from vector

The encodings in this section are decoded from SVE Permute Vector - Unpredicated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19	18	17	16						10	9	5	4	3	0
0	0	0	0	0	1	0	1	opc		1	0	1	opc2		0	0	0	1	1	1	0	Zn		0	Pd	

opc	opc2	Instruction Details	Condition
00	00	UNALLOCATED	-
00	01	PMOV (to predicate) — Byte	FEAT_SVE2p1 || FEAT_SME2p1
00	1x	PMOV (to predicate) — Halfword	FEAT_SVE2p1 || FEAT_SME2p1
01	xx	PMOV (to predicate) — Word	FEAT_SVE2p1 || FEAT_SME2p1
1x	xx	PMOV (to predicate) — Doubleword	FEAT_SVE2p1 || FEAT_SME2p1

SVE move predicate into vector

The encodings in this section are decoded from SVE Permute Vector - Unpredicated.

31	30	29	28					23	22	21	20	5	4	0
sf	1	1	1	0	0	1	1	1	opc		imm16		Rd	

opc	opc2	Instruction Details	Condition
00	00	UNALLOCATED	-
00	01	PMOV (to vector) — Byte	FEAT_SVE2p1 || FEAT_SME2p1
00	1x	PMOV (to vector) — Halfword	FEAT_SVE2p1 || FEAT_SME2p1
01	xx	PMOV (to vector) — Word	FEAT_SVE2p1 || FEAT_SME2p1
1x	xx	PMOV (to vector) — Doubleword	FEAT_SVE2p1 || FEAT_SME2p1

SVE unpack vector elements

The encodings in this section are decoded from SVE Permute Vector - Unpredicated.

31							24	23	22	21	20	19	18	17	16	15					10	9	5	4	0
0	0	0	0	0	1	0	1	size		1	1	0	0	U	H	0	0	1	1	1	0	Zn		Zd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	H	Instruction Details	Condition
0	0	SUNPKHI, SUNPKLO — Low half	FEAT_SVE || FEAT_SME
0	1	SUNPKHI, SUNPKLO — High half	FEAT_SVE || FEAT_SME
1	0	UUNPKHI, UUNPKLO — Low half	FEAT_SVE || FEAT_SME
1	1	UUNPKHI, UUNPKLO — High half	FEAT_SVE || FEAT_SME

SVE Permute Predicate

SVE Permute Predicate

The encodings in this section are decoded from SVE encodings.

31							24	23 22	21	20 16	15 13	12 9	8 5	4	3 0
0	0	0	0	0	1	0	1	op0	1	op1	0 1 0	op2		op3	

op0	op1	op2	op3	Instruction details	Condition
00	1000x	0000	0	SVE unpack predicate elements	-
xx	0xxxx	xxx0	0	SVE permute predicate elements	-
xx	10100	0000	0	REV (predicate)	FEAT_SVE || FEAT_SME
xx	10101	0000	0	UNALLOCATED	-
xx	10x0x	0010	0	UNALLOCATED	-
xx	10x0x	01x0	0	UNALLOCATED	-
xx	10x0x	1xx0	0	UNALLOCATED	-
xx	10x1x	xxx0	0	UNALLOCATED	-
xx	11xxx	xxx0	0	UNALLOCATED	-
xx	xxxxx	xxx0	1	UNALLOCATED	-
xx	xxxxx	xxx1	x	UNALLOCATED	-
!= 00	1000x	0000	0	UNALLOCATED	-

SVE unpack predicate elements

The encodings in this section are decoded from SVE Permute Predicate.

31	30	29	28 23	22	21 10	9 5	4 0
sf	op	S	1 0 0 0 1 0	sh	imm12	Rn	Rd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H	Instruction Details	Condition
0	PUNPKHI, PUNPKLO — Low half	FEAT_SVE || FEAT_SME
1	PUNPKHI, PUNPKLO — High half	FEAT_SVE || FEAT_SME

SVE permute predicate elements

The encodings in this section are decoded from SVE Permute Predicate.

31							24	23 22	21	20	19 16	15		13	12 11	10	9	8 5	4	3 0
0	0	0	0	0	1	0	1	size	1	0	Pm	0	1	0	opc	H	0	Pn	0	Pd

opc	H	Instruction Details	Condition
00	0	ZIP1, ZIP2 (predicates) — Low halves	FEAT_SVE || FEAT_SME
00	1	ZIP1, ZIP2 (predicates) — High halves	FEAT_SVE || FEAT_SME
01	0	UZP1, UZP2 (predicates) — Even	FEAT_SVE || FEAT_SME
01	1	UZP1, UZP2 (predicates) — Odd	FEAT_SVE || FEAT_SME
10	0	TRN1, TRN2 (predicates) — Even	FEAT_SVE || FEAT_SME
10	1	TRN1, TRN2 (predicates) — Odd	FEAT_SVE || FEAT_SME
11	x	UNALLOCATED	-

SVE Permute Vector - Predicated

SVE Permute Vector - Predicated

The encodings in this section are decoded from SVE encodings.

31							24	23 22	21	20	19 17	16	15 14	13	12 0
0	0	0	0	0	1	0	1		1	op0	op1	op2	1 0	op3	

op0	op1	op2	op3	Instruction details	Condition
0	000	0	0	CPY (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
0	000	1	0	SVE compress Active elements	-
0	000	x	1	SVE extract element to general register	-
0	001	1	1	UNALLOCATED	-
0	001	x	0	SVE extract element to SIMD&FP scalar register	-
0	01x	x	x	SVE reverse within elements	-
0	100	0	1	CPY (scalar)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
0	100	x	0	SVE conditionally broadcast element to vector	-
0	101	x	0	SVE conditionally extract element to SIMD&FP scalar	-
0	110	0	0	SPLICE — Destructive	FEAT_SVE || FEAT_SME
0	110	0	1	UNALLOCATED	-
0	110	1	0	SPLICE — Constructive	FEAT_SVE2 || FEAT_SME
0	111	0	x	SVE reverse doublewords	-
0	111	1	0	UNALLOCATED	-
0	1xx	1	1	UNALLOCATED	-
0	x01	0	1	UNALLOCATED	-
1	000	0	0	UNALLOCATED	-
1	000	1	0	EXPAND	FEAT_SVE2p2 || FEAT_SME2p2
1	000	x	1	SVE conditionally extract element to general register	-
1	!= 000	x	x	UNALLOCATED	-

SVE compress Active elements

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23 22	21	20	19		17	16	15	14	13	12 10	9 5	4 0
0	0	0	0	0	1	0	1	size	1	0	0	0	0	1	1	0	0	Pg	Zn	Zd

size	Instruction Details	Condition
0x	COMPACT — Byte and halfword	FEAT_SVE2p2 || FEAT_SME2p2
1x	COMPACT — Word and doubleword	FEAT_SVE || FEAT_SME2p2

SVE extract element to general register

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23 22	21	20	19		17	16	15	14	13	12 10	9 5	4 0
0	0	0	0	0	1	0	1	size	1	0	0	0	0	B	1	0	1	Pg	Zn	Rd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B	Instruction Details	Condition
0	LASTA (scalar)	FEAT_SVE || FEAT_SME
1	LASTB (scalar)	FEAT_SVE || FEAT_SME

SVE extract element to SIMD&FP scalar register

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19			17	16	15	14	13	12			10	9					5	4					0
0	0	0	0	0	1	0	1	size		1	0	0	0	1		B	1	0	0	Pg				Zn						Vd					

B	Instruction Details	Condition
0	LASTA (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
1	LASTB (SIMD&FP scalar)	FEAT_SVE || FEAT_SME

SVE reverse within elements

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19	18	17	16	15	14	13	12			10	9					5	4					0
0	0	0	0	0	1	0	1	size		1	0	0	1	opc		1	0	Z	Pg					Zn						Zd					

opc	Z	Instruction Details	Condition
00	0	REVB, REVH, REVW — Byte, merging	FEAT_SVE || FEAT_SME
00	1	REVB, REVH, REVW — Byte, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	0	REVB, REVH, REVW — Halfword, merging	FEAT_SVE || FEAT_SME
01	1	REVB, REVH, REVW — Halfword, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	0	REVB, REVH, REVW — Word, merging	FEAT_SVE || FEAT_SME
10	1	REVB, REVH, REVW — Word, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	RBIT — Merging	FEAT_SVE || FEAT_SME
11	1	RBIT — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE conditionally broadcast element to vector

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19		17	16	15	14	13	12		10	9		5	4		0
0	0	0	0	0	1	0	1	size		1	0	1	0	0	B	1	0	0	Pg			Zm			Zdn		

B	Instruction Details	Condition
0	CLASTA (vectors)	FEAT_SVE || FEAT_SME
1	CLASTB (vectors)	FEAT_SVE || FEAT_SME

SVE conditionally extract element to SIMD&FP scalar

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19		17	16	15	14	13	12		10	9		5	4		0
0	0	0	0	0	1	0	1	size		1	0	1	0	1	B	1	0	0	Pg			Zm			Vdn		

B	Instruction Details	Condition
0	CLASTA (SIMD&FP scalar)	FEAT_SVE || FEAT_SME
1	CLASTB (SIMD&FP scalar)	FEAT_SVE || FEAT_SME

SVE reverse doublewords

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19		17	16	15	14	13	12		10	9		5	4		0
0	0	0	0	0	1	0	1	size		1	0	1	1	1	0	1	0	Z	Pg			Zn			Zd		

size	Z	Instruction Details	Condition
00	0	REVD — Merging	FEAT_SME || FEAT_SVE2p1
00	1	REVD — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
!= 00	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE conditionally extract element to general register

The encodings in this section are decoded from SVE Permute Vector - Predicated.

31							24	23	22	21	20	19			17	16	15	14	13	12		10	9				5	4				0
0	0	0	0	0	1	0	1	size		1	1	0	0	0		B	1	0	1	Pg			Zm						Rdn			

B	Instruction Details	Condition
0	CLASTA (scalar)	FEAT_SVE || FEAT_SME
1	CLASTB (scalar)	FEAT_SVE || FEAT_SME

SVE Integer Compare - Vectors

SVE Integer Compare - Vectors

The encodings in this section are decoded from SVE encodings.

31							25	24	23	22	21	20			15	14	13																0
0	0	1	0	0	1	0		0			0					op0																	

op0	Instruction details
0	SVE integer compare vectors
1	SVE integer compare with wide elements

SVE integer compare vectors

The encodings in this section are decoded from SVE Integer Compare - Vectors.

31							24	23	22	21	20				16	15	14	13	12		10	9				5	4	3				0
0	0	1	0	0	1	0	0	size		0	Zm					op	0	o2	Pg			Zn					ne	Pd				

op	o2	ne	Instruction Details	Condition
0	0	0	CMP<cc> (vectors) — Higher or same	FEAT_SVE || FEAT_SME
0	0	1	CMP<cc> (vectors) — Higher	FEAT_SVE || FEAT_SME
0	1	0	CMP<cc> (wide elements) — Equal	FEAT_SVE || FEAT_SME
0	1	1	CMP<cc> (wide elements) — Not equal	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	o2	ne	Instruction Details	Condition
1	0	0	CMP<cc> (vectors) — Greater than or equal	FEAT_SVE || FEAT_SME
1	0	1	CMP<cc> (vectors) — Greater than	FEAT_SVE || FEAT_SME
1	1	0	CMP<cc> (vectors) — Equal	FEAT_SVE || FEAT_SME
1	1	1	CMP<cc> (vectors) — Not equal	FEAT_SVE || FEAT_SME

SVE integer compare with wide elements

The encodings in this section are decoded from SVE Integer Compare - Vectors.

31–24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 0 1 0 0 1 0 0	size	0	Zm	U	1	lt	Pg	Zn	ne	Pd

U	lt	ne	Instruction Details	Condition
0	0	0	CMP<cc> (wide elements) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	CMP<cc> (wide elements) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	CMP<cc> (wide elements) — Less than	FEAT_SVE || FEAT_SME
0	1	1	CMP<cc> (wide elements) — Less than or equal	FEAT_SVE || FEAT_SME
1	0	0	CMP<cc> (wide elements) — Higher or same	FEAT_SVE || FEAT_SME
1	0	1	CMP<cc> (wide elements) — Higher	FEAT_SVE || FEAT_SME
1	1	0	CMP<cc> (wide elements) — Lower	FEAT_SVE || FEAT_SME
1	1	1	CMP<cc> (wide elements) — Lower or same	FEAT_SVE || FEAT_SME

SVE Propagate Break

SVE Propagate Break

The encodings in this section are decoded from SVE encodings.

31–25	24	23–22	21–20	19–16	15–14	13–10	9	8–0
0 0 1 0 0 1 0	1		0 0		1 1		op0	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE propagate break from previous partition
1	UNALLOCATED

SVE propagate break from previous partition

The encodings in this section are decoded from SVE Propagate Break.

31							24	23	22	21	20	19			16	15	14	13			10	9	8			5	4	3			0
0	0	1	0	0	1	0	1	op	S	0	0	Pm				1	1	Pg				0	Pn				B	Pd			

op	S	B	Instruction Details	Condition
0	0	0	BRKPA	FEAT_SVE || FEAT_SME
0	0	1	BRKPB	FEAT_SVE || FEAT_SME
0	1	0	BRKPAS	FEAT_SVE || FEAT_SME
0	1	1	BRKPBS	FEAT_SVE || FEAT_SME
1	x	x	UNALLOCATED	-

SVE Partition Break

SVE Partition Break

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19			16	15	14	13			10	9	8			5	4	3			0
0	0	1	0	0	1	0	1	op0		0	1	op1				0	1					op2					op3				

op0	op1	op2	op3	Instruction details
0	1000	0	0	SVE propagate break to next partition
0	1000	0	1	UNALLOCATED
1	1000	0	x	UNALLOCATED
x	0000	0	x	SVE partition break condition
x	x000	1	x	UNALLOCATED
x	!= x000	x	x	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE propagate break to next partition

The encodings in this section are decoded from SVE Partition Break.

31–24	23	22	21–20	19–16	15–14	13–10	9	8–5	4	3–0
00100101	0	S	01	1000	01	Pg	0	Pn	0	Pdm

S	Instruction Details	Condition
0	BRKN	FEAT_SVE || FEAT_SME
1	BRKNS	FEAT_SVE || FEAT_SME

SVE partition break condition

The encodings in this section are decoded from SVE Partition Break.

31–24	23	22	21–20	19–16	15–14	13–10	9	8–5	4	3–0
00100101	B	S	01	0000	01	Pg	0	Pn	M	Pd

B	S	M	Instruction Details	Condition
x	1	1	UNALLOCATED	-
0	0	x	BRKA	FEAT_SVE || FEAT_SME
0	1	0	BRKAS	FEAT_SVE || FEAT_SME
1	0	x	BRKB	FEAT_SVE || FEAT_SME
1	1	0	BRKBS	FEAT_SVE || FEAT_SME

SVE Predicate Misc

SVE Predicate Misc

The encodings in this section are decoded from SVE encodings.

31–24	23–22	21–20	19–16	15–14	13–11	10–9	8–5	4	3–0
00100101		01	op0	11	op1	op2	op3	op4	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	op4	Instruction details	Condition
0000	xxx	x0	xxxx	0	SVE predicate test	-
0000	xxx	x1	xxxx	0	UNALLOCATED	-
0001	xxx	xx	xxxx	0	UNALLOCATED	-
1000	000	00	xxxx	0	SVE predicate first active	-
1000	000	10	xxxx	0	UNALLOCATED	-
1000	100	10	0000	0	SVE predicate zero	-
1000	100	10	!= 0000	0	UNALLOCATED	-
1000	110	00	xxxx	0	SVE predicate read from FFR (predicated)	-
1001	000	00	xxxx	0	UNALLOCATED	-
1001	000	10	xxxx	0	PNEXT	FEAT_SVE || FEAT_SME
1001	100	10	xxxx	0	UNALLOCATED	-
1001	110	00	0000	0	SVE predicate read from FFR (unpredicated)	-
1001	110	00	!= 0000	0	UNALLOCATED	-
100x	010	x0	xxxx	0	UNALLOCATED	-
100x	0x0	x1	xxxx	0	UNALLOCATED	-
100x	100	0x	xxxx	0	SVE predicate initialize	-
100x	100	11	xxxx	0	UNALLOCATED	-
100x	110	!= 00	xxxx	0	UNALLOCATED	-
100x	xx1	xx	xxxx	0	UNALLOCATED	-
x00x	xxx	xx	xxxx	1	UNALLOCATED	-
!= x00x	xxx	xx	xxxx	x	UNALLOCATED	-

SVE predicate test

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13	10	9	8	5	4	3	0
0	0	1	0	0	1	0	1	op	S	0	1	0	0	0	0	1	1	Pg		0	Pn		0	opc2	

op	S	opc2	Instruction Details	Condition
0	0	xxxx	UNALLOCATED	-
0	1	0000	PTEST	FEAT_SVE || FEAT_SME
0	1	!= 0000	UNALLOCATED	-
1	x	xxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE predicate first active

The encodings in this section are decoded from SVE Predicate Misc.

31–24	23	22	21–20	19–16	15–14	13–11	10–9	8–5	4	3–0
0 0 1 0 0 1 0 1	op	S	0 1	1 0 0 0	1 1	0 0 0	0 0	Pg	0	Pdn

op	S	Instruction Details	Condition
0	0	UNALLOCATED	-
0	1	PFIRST	FEAT_SVE || FEAT_SME
1	x	UNALLOCATED	-

SVE predicate zero

The encodings in this section are decoded from SVE Predicate Misc.

31–24	23	22	21–20	19–16	15–14	13–11	10–9	8–5	4	3–0
0 0 1 0 0 1 0 1	op	S	0 1	1 0 0 0	1 1	1 0 0	1 0	0 0 0 0	0	Pd

op	S	Instruction Details	Condition
0	0	PFALSE	FEAT_SVE || FEAT_SME
0	1	UNALLOCATED	-
1	x	UNALLOCATED	-

SVE predicate read from FFR (predicated)

The encodings in this section are decoded from SVE Predicate Misc.

31–24	23	22	21–20	19–16	15–14	13–11	10–9	8–5	4	3–0
0 0 1 0 0 1 0 1	op	S	0 1	1 0 0 0	1 1	1 1 0	0 0	Pg	0	Pd

op	S	Instruction Details	Condition
0	0	RDFFR (predicated)	FEAT_SVE
0	1	RDFFRS	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	Instruction Details	Condition
1	x	UNALLOCATED	-

SVE predicate read from FFR (unpredicated)

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23	22	21	20	19			16	15	14	13			11	10	9	8			5	4	3				0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	1	1	1	1	1	0		0	0	0	0	0	0	0	Pd				

op	S	Instruction Details	Condition
0	0	RDFFR (unpredicated)	FEAT_SVE
0	1	UNALLOCATED	-
1	x	UNALLOCATED	-

SVE predicate initialize

The encodings in this section are decoded from SVE Predicate Misc.

31							24	23		22	21				17	16	15		14	13			11	10	9				5	4	3				0
0	0	1	0	0	1	0	1	size			0	1	1	0	0	S	1		1	1	0	0		0	pattern					0	Pd			

S	Instruction Details	Condition
0	PTRUE (predicate)	FEAT_SVE || FEAT_SME
1	PTRUES	FEAT_SVE || FEAT_SME

SVE Predicate Count

SVE Predicate Count

The encodings in this section are decoded from SVE encodings.

31		29	28						22	21	20	19	18		16	15			12	11			8	7		5	4				0
1	1	0	1	0	1	0	1	0	1	L	0	1	op1			CRn				CRm				op2			Rt				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
000	1	SVE predicate count (predicate-as-counter)
xxx	0	SVE predicate count
!= 000	1	UNALLOCATED

SVE predicate count (predicate-as-counter)

The encodings in this section are decoded from SVE Predicate Count.

31							24	23	22	21		19	18		16	15	14	13			11	10	9	8		5	4		0
0	0	1	0	0	1	0	1	size		1	0	0	opc			1	0	0	0	0		vl	1	PNn			Rd		

opc	Instruction Details	Condition
000	CNTP (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
!= 000	UNALLOCATED	-

SVE predicate count

The encodings in this section are decoded from SVE Predicate Count.

31							24	23	22	21		19	18		16	15	14	13			10	9	8		5	4		0
0	0	1	0	0	1	0	1	size		1	0	0	opc			1	0	Pg				0	Pn			Rd		

opc	Instruction Details	Condition
000	CNTP (predicate)	FEAT_SVE || FEAT_SME
001	FIRSTP	FEAT_SVE2p2 || FEAT_SME2p2
010	LASTP	FEAT_SVE2p2 || FEAT_SME2p2
011	UNALLOCATED	-
1xx	UNALLOCATED	-

SVE Inc/Dec by Predicate Count

SVE Inc/Dec by Predicate Count

The encodings in this section are decoded from SVE encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21		19	18	17	16	15			12	11	10										0
0	0	1	0	0	1	0	1			1	0	1	op0			1	0	0	0	op1											

op0	op1	Instruction details
0	0	SVE saturating inc/dec vector by predicate count
0	1	SVE saturating inc/dec register by predicate count
1	0	SVE inc/dec vector by predicate count
1	1	SVE inc/dec register by predicate count

SVE saturating inc/dec vector by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23	22	21		19	18	17	16	15			12	11	10	9	8		5	4		0
0	0	1	0	0	1	0	1	size		1	0	1	0	D	U	1	0	0	0	0	opc		Pm			Zdn		

D	U	opc	Instruction Details	Condition
x	x	!= 00	UNALLOCATED	-
0	0	00	SQINCP (vector)	FEAT_SVE || FEAT_SME
0	1	00	UQINCP (vector)	FEAT_SVE || FEAT_SME
1	0	00	SQDECP (vector)	FEAT_SVE || FEAT_SME
1	1	00	UQDECP (vector)	FEAT_SVE || FEAT_SME

SVE saturating inc/dec register by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23	22	21		19	18	17	16	15			12	11	10	9	8		5	4		0
0	0	1	0	0	1	0	1	size		1	0	1	0	D	U	1	0	0	0	1	sf	op	Pm			Rdn		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D	U	sf	op	Instruction Details	Condition
x	x	x	1	UNALLOCATED	-
0	0	0	0	SQINCP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
0	0	1	0	SQINCP (scalar) — 64-bit	FEAT_SVE || FEAT_SME
0	1	0	0	UQINCP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
0	1	1	0	UQINCP (scalar) — 64-bit	FEAT_SVE || FEAT_SME
1	0	0	0	SQDECP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
1	0	1	0	SQDECP (scalar) — 64-bit	FEAT_SVE || FEAT_SME
1	1	0	0	UQDECP (scalar) — 32-bit	FEAT_SVE || FEAT_SME
1	1	1	0	UQDECP (scalar) — 64-bit	FEAT_SVE || FEAT_SME

SVE inc/dec vector by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23	22	21		19	18	17	16	15			12	11	10	9	8	5	4	0
0	0	1	0	0	1	0	1	size		1	0	1	1	op	D	1	0	0	0	0	opc2		Pm		Zdn	

op	D	opc2	Instruction Details	Condition
0	x	!= 00	UNALLOCATED	-
0	0	00	INCP (vector)	FEAT_SVE || FEAT_SME
0	1	00	DECP (vector)	FEAT_SVE || FEAT_SME
1	x	xx	UNALLOCATED	-

SVE inc/dec register by predicate count

The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

31							24	23	22	21		19	18	17	16	15			12	11	10	9	8	5	4	0
0	0	1	0	0	1	0	1	size		1	0	1	1	op	D	1	0	0	0	1	opc2		Pm		Rdn	

op	D	opc2	Instruction Details	Condition
0	x	!= 00	UNALLOCATED	-
0	0	00	INCP (scalar)	FEAT_SVE || FEAT_SME
0	1	00	DECP (scalar)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	D	opc2	Instruction Details	Condition
1	x	xx	UNALLOCATED	-

SVE Write FFR

SVE Write FFR

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21		19	18	17	16	15			12	11		9	8		5	4		0
0	0	1	0	0	1	0	1			1	0	1	op0	op1		1	0	0	1	op2			op3			op4		

op0	op1	op2	op3	op4	Instruction details
0	00	000	xxxx	00000	SVE FFR write from predicate
1	00	000	0000	00000	SVE FFR initialize
1	00	000	!= 0000	00000	UNALLOCATED
x	00	000	xxxx	!= 00000	UNALLOCATED
x	00	!= 000	xxxx	xxxxx	UNALLOCATED
x	!= 00	xxx	xxxx	xxxxx	UNALLOCATED

SVE FFR write from predicate

The encodings in this section are decoded from SVE Write FFR.

31							24	23	22	21		19	18	17	16	15			12	11		9	8		5	4		0
0	0	1	0	0	1	0	1	opc		1	0	1	0	0	0	1	0	0	1	0	0	0	Pn			0	0	0	0	0

opc	Instruction Details	Condition
00	WRFFR	FEAT_SVE
!= 00	UNALLOCATED	-

SVE FFR initialize

The encodings in this section are decoded from SVE Write FFR.

31							24	23	22	21		19	18	17	16	15			12	11		9	8		5	4		0
0	0	1	0	0	1	0	1	opc		1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
00	SETFFR	FEAT_SVE
!=00	UNALLOCATED	-

SVE Integer Compare - Scalars

SVE Integer Compare - Scalars

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15	14	13	12	11	10	9					4	3				0
0	0	1	0	0	1	0	1			1						0	0	op0		op1								op2				

op0	op1	op2	Instruction details
0x	xx	xxxx	SVE integer compare scalar count and limit
10	00	0000	SVE conditionally terminate scalars
10	00	!=0000	UNALLOCATED
11	00	xxxx	SVE pointer conflict compare
1x	!=00	xxxx	UNALLOCATED

SVE integer compare scalar count and limit

The encodings in this section are decoded from SVE Integer Compare - Scalars.

31							24	23	22	21	20				16	15			13	12	11	10	9				5	4	3				0
0	0	1	0	0	1	0	1	size		1	Rm					0	0	0		sf	U	lt	Rn					eq	Pd				

U	lt	eq	Instruction Details	Condition
0	0	0	WHILEGE (predicate)	FEAT_SVE2 || FEAT_SME
0	0	1	WHILEGT (predicate)	FEAT_SVE2 || FEAT_SME
0	1	0	WHILELT (predicate)	FEAT_SVE || FEAT_SME
0	1	1	WHILELE (predicate)	FEAT_SVE || FEAT_SME
1	0	0	WHILEHS (predicate)	FEAT_SVE2 || FEAT_SME
1	0	1	WHILEHI (predicate)	FEAT_SVE2 || FEAT_SME
1	1	0	WHILELO (predicate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	lt	eq	Instruction Details	Condition
1	1	1	WHILELS (predicate)	FEAT_SVE || FEAT_SME

SVE conditionally terminate scalars

The encodings in this section are decoded from SVE Integer Compare - Scalars.

31							24	23	22	21	20				16	15	14	13	12	11	10	9				5	4	3			0
0	0	1	0	0	1	0	1	op	sz	1			Rm			0	0	1	0	0	0			Rn			ne	0	0	0	0

op	ne	Instruction Details	Condition
0	x	UNALLOCATED	-
1	0	CTERMEQ, CTERMNE — Equal	FEAT_SVE || FEAT_SME
1	1	CTERMEQ, CTERMNE — Not equal	FEAT_SVE || FEAT_SME

SVE pointer conflict compare

The encodings in this section are decoded from SVE Integer Compare - Scalars.

31							24	23		22	21	20				16	15	14	13	12	11	10	9				5	4	3			0
0	0	1	0	0	1	0	1	size			1			Rm			0	0	1	1	0	0			Rn			rw		Pd		

rw	Instruction Details	Condition
0	WHILEWR	FEAT_SVE2 || FEAT_SME
1	WHILERW	FEAT_SVE2 || FEAT_SME

SVE Scalar Integer Compare - Predicate-as-counter

SVE Scalar Integer Compare - Predicate-as-counter

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15	14	13			11	10					5	4	3	2			0
0	0	1	0	0	1	0	1			1			op0			0	1		op1					op2				1	op3			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
00000	110	xxxxxx	x	SVE extract mask predicate from predicate-as-counter	-
00000	111	000000	0	PTRUE (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
00000	111	000000	1	UNALLOCATED	-
00000	111	!= 000000	x	UNALLOCATED	-
xxxxx	01x	xxxxxx	x	SVE integer compare scalar count and limit (predicate pair)	-
xxxxx	x0x	xxxxxx	x	SVE integer compare scalar count and limit (predicate-as-counter)	-
!= 00000	11x	xxxxxx	x	UNALLOCATED	-

SVE extract mask predicate from predicate-as-counter

The encodings in this section are decoded from SVE Scalar Integer Compare - Predicate-as-counter.

31–24	23–22	21	20–16	15–14	13–11	10–8	7–5	4	3–0
0 0 1 0 0 1 0 1	size	1	0 0 0 0 0	0 1	1 1 0	opc	PNn	1	Pd

opc	Instruction Details	Condition
0xx	PEXT (predicate)	FEAT_SME2 || FEAT_SVE2p1
10x	PEXT (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
11x	UNALLOCATED	-

SVE integer compare scalar count and limit (predicate pair)

The encodings in this section are decoded from SVE Scalar Integer Compare - Predicate-as-counter.

31–24	23–22	21	20–16	15–12	11	10	9–5	4	3–1	0
0 0 1 0 0 1 0 1	size	1	Rm	0 1 0 1	U	lt	Rn	1	Pd	eq

U	lt	eq	Instruction Details	Condition
0	0	0	WHILEGE (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
0	0	1	WHILEGT (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
0	1	0	WHILELT (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
0	1	1	WHILELE (predicate pair)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	lt	eq	Instruction Details	Condition
1	0	0	WHILEHS (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	0	1	WHILEHI (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	1	0	WHILELO (predicate pair)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	WHILELS (predicate pair)	FEAT_SME2 || FEAT_SVE2p1

SVE integer compare scalar count and limit (predicate-as-counter)

The encodings in this section are decoded from SVE Scalar Integer Compare - Predicate-as-counter.

31							24	23 22	21	20 16	15	14	13	12	11	10	9 5	4	3	2 0
0	0	1	0	0	1	0	1	size	1	Rm	0	1	vl	0	U	lt	Rn	1	eq	PNd

U	lt	eq	Instruction Details	Condition
0	0	0	WHILEGE (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
0	0	1	WHILEGT (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
0	1	0	WHILELT (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
0	1	1	WHILELE (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	0	0	WHILEHS (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	0	1	WHILEHI (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	1	0	WHILELO (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	WHILELS (predicate as counter)	FEAT_SME2 || FEAT_SVE2p1

SVE Integer Wide Immediate - Unpredicated

SVE Integer Wide Immediate - Unpredicated

The encodings in this section are decoded from SVE encodings.

31							24	23 22	21	20 19	18 17	16	15	14	13 0
0	0	1	0	0	1	0	1		1	op0		op1	1	1	

op0	op1	Instruction details
00	x	SVE integer add/subtract immediate (unpredicated)
01	x	SVE integer min/max immediate (unpredicated)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
10	x	SVE integer multiply immediate (unpredicated)
11	0	SVE broadcast integer immediate (unpredicated)
11	1	SVE broadcast floating-point immediate (unpredicated)

SVE integer add/subtract immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	0 0	opc	1 1	sh	imm8	Zdn

opc	Instruction Details	Condition
000	ADD (immediate)	FEAT_SVE || FEAT_SME
001	SUB (immediate)	FEAT_SVE || FEAT_SME
010	UNALLOCATED	-
011	SUBR (immediate)	FEAT_SVE || FEAT_SME
100	SQADD (immediate)	FEAT_SVE || FEAT_SME
101	UQADD (immediate)	FEAT_SVE || FEAT_SME
110	SQSUB (immediate)	FEAT_SVE || FEAT_SME
111	UQSUB (immediate)	FEAT_SVE || FEAT_SME

SVE integer min/max immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	0 1	opc	1 1	o2	imm8	Zdn

opc	o2	Instruction Details	Condition
0xx	1	UNALLOCATED	-
000	0	SMAX (immediate)	FEAT_SVE || FEAT_SME
001	0	UMAX (immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Instruction Details	Condition
010	0	SMIN (immediate)	FEAT_SVE || FEAT_SME
011	0	UMIN (immediate)	FEAT_SVE || FEAT_SME
1xx	x	UNALLOCATED	-

SVE integer multiply immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	1 0	opc	1 1	o2	imm8	Zdn

opc	o2	Instruction Details	Condition
000	0	MUL (immediate)	FEAT_SVE || FEAT_SME
000	1	UNALLOCATED	-
!= 000	x	UNALLOCATED	-

SVE broadcast integer immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	1 1	opc	0	1 1	sh	imm8	Zd

opc	Instruction Details	Condition
00	DUP (immediate)	FEAT_SVE || FEAT_SME
!= 00	UNALLOCATED	-

SVE broadcast floating-point immediate (unpredicated)

The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated.

31							24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0	0	1	0	0	1	0	1	size	1	1 1	opc	1	1 1	o2	imm8	Zd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Instruction Details	Condition
00	0	FDUP	FEAT_SVE || FEAT_SME
00	1	UNALLOCATED	-
!= 00	x	UNALLOCATED	-

SVE Integer Multiply-Add - Unpredicated

SVE Integer Multiply-Add - Unpredicated

The encodings in this section are decoded from SVE encodings.

31	30	29	28	27 25	24 21	20 15	14	13 10	9 5	4	3 0
sf	op	S	1	1 0 1	0 0 0 0	opcode2	sz	0 0 1 0	Rn	o3	mask

op0	Instruction details	Condition
0000x	SVE integer dot product (unpredicated)	-
0001x	SVE2 saturating multiply-add interleaved long	-
001xx	CDOT (vectors)	FEAT_SVE2 || FEAT_SME
01xxx	SVE2 complex integer multiply-add	-
10xxx	SVE2 integer multiply-add long	-
110xx	SVE2 saturating multiply-add long	-
1110x	SVE2 saturating multiply-add high	-
11110	SVE mixed sign dot product	-
11111	UNALLOCATED	-

SVE integer dot product (unpredicated)

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31 24	23 22	21	20 16	15 11	10	9 5	4 0
0 1 0 0 0 1 0 0	size	0	Zm	0 0 0 0 0	U	Zn	Zda

size	U	Instruction Details	Condition
0x	x	UNALLOCATED	-
1x	0	SDOT (4-way, vectors)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	U	Instruction Details	Condition
1x	1	UDOT (4-way, vectors)	FEAT_SVE || FEAT_SME

SVE2 saturating multiply-add interleaved long

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31							24	23	22	21	20				16	15				11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	0	0	1	S	Zn					Zda				

S	Instruction Details	Condition
0	SQDMLALBT	FEAT_SVE2 || FEAT_SME
1	SQDMLSLBT	FEAT_SVE2 || FEAT_SME

SVE2 complex integer multiply-add

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31							24	23	22	21	20				16	15			13	12	11		10	9				5	4				0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	1		op	rot			Zn					Zda				

op	Instruction Details	Condition
0	CMLA (vectors)	FEAT_SVE2 || FEAT_SME
1	SQRDCMLAH (vectors)	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply-add long

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31							24	23	22	21	20				16	15			13	12	11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		0	Zm					0	1	0		S	U	T		Zn					Zda				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	U	T	Instruction Details	Condition
0	0	0	SMLALB (vectors)	FEAT_SVE2 || FEAT_SME
0	0	1	SMLALT (vectors)	FEAT_SVE2 || FEAT_SME
0	1	0	UMLALB (vectors)	FEAT_SVE2 || FEAT_SME
0	1	1	UMLALT (vectors)	FEAT_SVE2 || FEAT_SME
1	0	0	SMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
1	0	1	SMLSLT (vectors)	FEAT_SVE2 || FEAT_SME
1	1	0	UMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
1	1	1	UMLSLT (vectors)	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply-add long

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31–24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 0 0 0 1 0 0	size	0	Zm	0 1 1 0	S	T	Zn	Zda

S	T	Instruction Details	Condition
0	0	SQDMLALB (vectors)	FEAT_SVE2 || FEAT_SME
0	1	SQDMLALT (vectors)	FEAT_SVE2 || FEAT_SME
1	0	SQDMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
1	1	SQDMLSLT (vectors)	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply-add high

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31–24	23–22	21	20–16	15–11	10	9–5	4–0
0 1 0 0 0 1 0 0	size	0	Zm	0 1 1 1 0	S	Zn	Zda

S	Instruction Details	Condition
0	SQRDMLAH (vectors)	FEAT_SVE2 || FEAT_SME
1	SQRDMLSH (vectors)	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE mixed sign dot product

The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.

31	24	23 22	21	20 16	15	14 10	9 5	4 0
0 1 0 0 0 1 0 0		size	0	Zm	0	1 1 1 1 0	Zn	Zda

size	Instruction Details	Condition
`10`	USDOT (vectors)	(FEAT_SVE && FEAT_I8MM) || (FEAT_SME && FEAT_I8MM)
`!=10`	UNALLOCATED	-

SVE2 Integer - Predicated

SVE2 Integer - Predicated

The encodings in this section are decoded from SVE encodings.

31	24	23 22	21	20 17	16	15 14	13	12 0
0 1 0 0 0 1 0 0			0	op0		1 0	op1	

op0	op1	Instruction details
`0010`	`1`	SVE2 integer pairwise add and accumulate long
`0011`	`1`	UNALLOCATED
`011x`	`1`	UNALLOCATED
`0x0x`	`1`	SVE2 integer unary operations (predicated)
`0xxx`	`0`	SVE2 saturating/rounding bitwise shift left (predicated)
`10xx`	`0`	SVE2 integer halving add/subtract (predicated)
`10xx`	`1`	SVE2 integer pairwise arithmetic
`11xx`	`0`	SVE2 saturating add/subtract
`11xx`	`1`	UNALLOCATED

SVE2 integer pairwise add and accumulate long

The encodings in this section are decoded from SVE2 Integer - Predicated.

31	24	23 22	21	20 17	16	15 14	13	12 10	9 5	4 0
0 1 0 0 0 1 0 0		size	0	0 0 1 0	U	1 0	1	Pg	Zn	Zda

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Condition
0	SADALP	FEAT_SVE2 || FEAT_SME
1	UADALP	FEAT_SVE2 || FEAT_SME

SVE2 integer unary operations (predicated)

The encodings in this section are decoded from SVE2 Integer - Predicated.

31	30	29	28				24	23	22			19	18	17	16	12	11	10	9	5	4	0
0	1	U	1	1	1	1	0	a	1	1	1	1	0	0	opcode		1	0	Rn		Rd	

Q	Z	op	Instruction Details	Condition
0	0	0	URECPE — Merging	FEAT_SVE2 || FEAT_SME
0	0	1	URSQRTE — Merging	FEAT_SVE2 || FEAT_SME
0	1	0	URECPE — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	1	1	URSQRTE — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	0	0	SQABS — Merging	FEAT_SVE2 || FEAT_SME
1	0	1	SQNEG — Merging	FEAT_SVE2 || FEAT_SME
1	1	0	SQABS — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	1	1	SQNEG — Zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE2 saturating/rounding bitwise shift left (predicated)

The encodings in this section are decoded from SVE2 Integer - Predicated.

31	30	29	28				24	23	22	21	20	16	15	14	11	10	9	5	4	0
0	1	U	1	1	1	1	0	size		0	Rm		1	opcode		1	Rn		Rd	

Q	R	N	U	Instruction Details	Condition
0	x	0	x	UNALLOCATED	-
0	0	1	0	SRSHL	FEAT_SVE2 || FEAT_SME
0	0	1	1	URSHL	FEAT_SVE2 || FEAT_SME
0	1	1	0	SRSHLR	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	R	N	U	Instruction Details	Condition
0	1	1	1	URSHLR	FEAT_SVE2 || FEAT_SME
1	0	0	0	SQSHL (vectors)	FEAT_SVE2 || FEAT_SME
1	0	0	1	UQSHL (vectors)	FEAT_SVE2 || FEAT_SME
1	0	1	0	SQRSHL	FEAT_SVE2 || FEAT_SME
1	0	1	1	UQRSHL	FEAT_SVE2 || FEAT_SME
1	1	0	0	SQSHLR	FEAT_SVE2 || FEAT_SME
1	1	0	1	UQSHLR	FEAT_SVE2 || FEAT_SME
1	1	1	0	SQRSHLR	FEAT_SVE2 || FEAT_SME
1	1	1	1	UQRSHLR	FEAT_SVE2 || FEAT_SME

SVE2 integer halving add/subtract (predicated)

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9		5	4		0
0	1	0	0	0	1	0	0	size		0	1	0	R	S	U	1	0	0	Pg			Zm			Zdn		

R	S	U	Instruction Details	Condition
0	0	0	SHADD	FEAT_SVE2 || FEAT_SME
0	0	1	UHADD	FEAT_SVE2 || FEAT_SME
0	1	0	SHSUB	FEAT_SVE2 || FEAT_SME
0	1	1	UHSUB	FEAT_SVE2 || FEAT_SME
1	0	0	SRHADD	FEAT_SVE2 || FEAT_SME
1	0	1	URHADD	FEAT_SVE2 || FEAT_SME
1	1	0	SHSUBR	FEAT_SVE2 || FEAT_SME
1	1	1	UHSUBR	FEAT_SVE2 || FEAT_SME

SVE2 integer pairwise arithmetic

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23	22	21		19	18	17	16	15	14	13	12		10	9		5	4		0
0	1	0	0	0	1	0	0	size		0	1	0	opc		U	1	0	1	Pg			Zm			Zdn		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	U	Instruction Details	Condition
00	0	UNALLOCATED	-
00	1	ADDP	FEAT_SVE2 || FEAT_SME
01	x	UNALLOCATED	-
10	0	SMAXP	FEAT_SVE2 || FEAT_SME
10	1	UMAXP	FEAT_SVE2 || FEAT_SME
11	0	SMINP	FEAT_SVE2 || FEAT_SME
11	1	UMINP	FEAT_SVE2 || FEAT_SME

SVE2 saturating add/subtract

The encodings in this section are decoded from SVE2 Integer - Predicated.

31							24	23 22	21		19	18	17	16	15	14	13	12 10	9 5	4 0
0	1	0	0	0	1	0	0	size	0	1	1	op	S	U	1	0	0	Pg	Zm	Zdn

op	S	U	Instruction Details	Condition
0	0	0	SQADD (vectors, predicated)	FEAT_SVE2 || FEAT_SME
0	0	1	UQADD (vectors, predicated)	FEAT_SVE2 || FEAT_SME
0	1	0	SQSUB (vectors, predicated)	FEAT_SVE2 || FEAT_SME
0	1	1	UQSUB (vectors, predicated)	FEAT_SVE2 || FEAT_SME
1	0	0	SUQADD	FEAT_SVE2 || FEAT_SME
1	0	1	USQADD	FEAT_SVE2 || FEAT_SME
1	1	0	SQSUBR	FEAT_SVE2 || FEAT_SME
1	1	1	UQSUBR	FEAT_SVE2 || FEAT_SME

SVE Multiply - Indexed

SVE Multiply - Indexed

The encodings in this section are decoded from SVE encodings.

31							25	24	23 22	21	20 16	15 10	9 0
0	1	0	0	0	1	0		0		1		op0	

op0	Instruction details
00000x	SVE integer dot product (indexed)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
00001x	SVE2 integer multiply-add (indexed)
00010x	SVE2 saturating multiply-add high (indexed)
00011x	SVE mixed sign dot product (indexed)
001xxx	SVE2 saturating multiply-add (indexed)
0100xx	SVE2 complex integer dot product (indexed)
0101xx	UNALLOCATED
0110xx	SVE2 complex integer multiply-add (indexed)
0111xx	SVE2 complex saturating multiply-add (indexed)
10xxxx	SVE2 integer multiply-add long (indexed)
110xxx	SVE2 integer multiply long (indexed)
1110xx	SVE2 saturating multiply (indexed)
11110x	SVE2 saturating multiply high (indexed)
111110	SVE2 integer multiply (indexed)
111111	UNALLOCATED

SVE integer dot product (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23 22	21	20 16	15				11	10	9 5	4 0
0	1	0	0	0	1	0	0	size	1	opc	0	0	0	0	0	U	Zn	Zda

size	U	Instruction Details	Condition
0x	x	UNALLOCATED	-
10	0	SDOT (4-way, indexed) — 8-bit to 32-bit	FEAT_SVE || FEAT_SME
10	1	UDOT (4-way, indexed) — 8-bit to 32-bit	FEAT_SVE || FEAT_SME
11	0	SDOT (4-way, indexed) — 16-bit to 64-bit	FEAT_SVE || FEAT_SME
11	1	UDOT (4-way, indexed) — 16-bit to 64-bit	FEAT_SVE || FEAT_SME

SVE2 integer multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					0	0	0	0	1		S	Zn					Zda				

size	S	Instruction Details	Condition
0x	0	MLA (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
0x	1	MLS (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	0	MLA (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	MLS (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	MLA (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	MLS (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply-add high (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					0	0	0	1	0		S	Zn					Zda				

size	S	Instruction Details	Condition
0x	0	SQRDMLAH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
0x	1	SQRDMLSH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	0	SQRDMLAH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	SQRDMLSH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	SQRDMLAH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	SQRDMLSH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE mixed sign dot product (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					0	0	0	1	1		U	Zn					Zda				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	U	Instruction Details	Condition
10	0	USDOT (indexed)	(FEAT_SVE && FEAT_I8MM) || (FEAT_SME && FEAT_I8MM)
10	1	SUDOT	(FEAT_SVE && FEAT_I8MM) || (FEAT_SME && FEAT_I8MM)
!=10	x	UNALLOCATED	-

SVE2 saturating multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31	30	29	28			25	24	23	22	21	20	16	15	14	11	10	9	5	4	0
0	Q	op	0	1	1	1	0	0	0	0	imm5		0	imm4		1	Rn		Rd	

size	S	T	Instruction Details	Condition
0x	x	x	UNALLOCATED	-
10	0	0	SQDMLALB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	SQDMLALT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	SQDMLSLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	SQDMLSLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	0	SQDMLALB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	SQDMLALT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	SQDMLSLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	SQDMLSLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 complex integer dot product (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20	16	15			12	11	10	9	5	4	0
0	1	0	0	0	1	0	0	size		1	opc		0	1	0	0	rot		Zn		Zda	

size	Instruction Details	Condition
0x	UNALLOCATED	-
10	CDOT (indexed) — 8-bit to 32-bit	FEAT_SVE2 || FEAT_SME
11	CDOT (indexed) — 16-bit to 64-bit	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 complex integer multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 1 0 0 0 1 0 0		size	1	opc	0 1 1 0	rot	Zn	Zda

size	Instruction Details	Condition
0x	UNALLOCATED	-
10	CMLA (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
11	CMLA (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME

SVE2 complex saturating multiply-add (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 1 0 0 0 1 0 0		size	1	opc	0 1 1 1	rot	Zn	Zda

size	Instruction Details	Condition
0x	UNALLOCATED	-
10	SQRDCMLAH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
11	SQRDCMLAH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply-add long (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31	24	23 22	21	20 16	15 14	13	12	11	10	9 5	4 0
0 1 0 0 0 1 0 0		size	1	opc	1 0	S	U	i1	T	Zn	Zda

size	S	U	T	Instruction Details	Condition
0x	x	x	x	UNALLOCATED	-
10	0	0	0	SMLALB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	S	U	T	Instruction Details	Condition
10	0	0	1	SMLALT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	0	UMLALB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	1	UMLALT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	0	SMLSLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	1	SMLSLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	0	UMLSLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	1	UMLSLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	0	0	SMLALB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	0	1	SMLALT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	0	UMLALB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	1	UMLALT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	0	SMLSLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	1	SMLSLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	0	UMLSLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	1	UMLSLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply long (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23 22	21	20 16	15 13	12	11	10	9 5	4 0
0	1	0	0	0	1	0	0	size	1	opc	1 1 0	U	il	T	Zn	Zd

size	U	T	Instruction Details	Condition
0x	x	x	UNALLOCATED	-
10	0	0	SMULLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	0	1	SMULLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	0	UMULLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	1	UMULLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	0	SMULLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	0	1	SMULLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	0	UMULLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	1	UMULLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28				24	23	22	21	20				16	15				12	11	10	9				5	4				0
0	Q	U	0	1	1	1	0	size		1	Rm					opcode					0	0	Rn					Rd				

size	T	Instruction Details	Condition
0x	x	UNALLOCATED	-
10	0	SQDMULLB (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	SQDMULLT (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	SQDMULLB (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	SQDMULLT (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 saturating multiply high (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15					11	10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					1	1	1	1	0		R	Zn					Zd				

size	R	Instruction Details	Condition
0x	0	SQDMULH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
0x	1	SQRDMULH (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	0	SQDMULH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
10	1	SQRDMULH (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	0	SQDMULH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME
11	1	SQRDMULH (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply (indexed)

The encodings in this section are decoded from SVE Multiply - Indexed.

31							24	23	22	21	20				16	15						10	9				5	4				0
0	1	0	0	0	1	0	0	size		1	opc					1	1	1	1	1		0	Zn					Zd				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Condition
0x	MUL (indexed) — 16-bit	FEAT_SVE2 || FEAT_SME
10	MUL (indexed) — 32-bit	FEAT_SVE2 || FEAT_SME
11	MUL (indexed) — 64-bit	FEAT_SVE2 || FEAT_SME

SVE2 Widening Integer Arithmetic

SVE2 Widening Integer Arithmetic

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20		16	15	14	13	12		0
0	1	0	0	0	1	0	1			0				0	op0				

op0	Instruction details
0x	SVE2 integer add/subtract long
10	SVE2 integer add/subtract wide
11	SVE2 integer multiply long

SVE2 integer add/subtract long

The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

31							24	23	22	21	20		16	15	14	13	12	11	10	9		5	4		0
0	1	0	0	0	1	0	1	size		0	Zm			0	0	op	S	U	T	Zn			opc		

op	S	U	T	Instruction Details	Condition
0	0	0	0	SADDLB	FEAT_SVE2 || FEAT_SME
0	0	0	1	SADDLT	FEAT_SVE2 || FEAT_SME
0	0	1	0	UADDLB	FEAT_SVE2 || FEAT_SME
0	0	1	1	UADDLT	FEAT_SVE2 || FEAT_SME
0	1	0	0	SSUBLB	FEAT_SVE2 || FEAT_SME
0	1	0	1	SSUBLT	FEAT_SVE2 || FEAT_SME
0	1	1	0	USUBLB	FEAT_SVE2 || FEAT_SME
0	1	1	1	USUBLT	FEAT_SVE2 || FEAT_SME
1	0	x	x	UNALLOCATED	-
1	1	0	0	SABDLB	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	U	T	Instruction Details	Condition
1	1	0	1	SABDLT	FEAT_SVE2 || FEAT_SME
1	1	1	0	UABDLB	FEAT_SVE2 || FEAT_SME
1	1	1	1	UABDLT	FEAT_SVE2 || FEAT_SME

SVE2 integer add/subtract wide

The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

31							24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	1	0	0	0	1	0	1	size	0	Zm	0	1 0	S	U	T	Zn	Zd

S	U	T	Instruction Details	Condition
0	0	0	SADDWB	FEAT_SVE2 || FEAT_SME
0	0	1	SADDWT	FEAT_SVE2 || FEAT_SME
0	1	0	UADDWB	FEAT_SVE2 || FEAT_SME
0	1	1	UADDWT	FEAT_SVE2 || FEAT_SME
1	0	0	SSUBWB	FEAT_SVE2 || FEAT_SME
1	0	1	SSUBWT	FEAT_SVE2 || FEAT_SME
1	1	0	USUBWB	FEAT_SVE2 || FEAT_SME
1	1	1	USUBWT	FEAT_SVE2 || FEAT_SME

SVE2 integer multiply long

The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

31							24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	1	0	0	0	1	0	1	size	0	Zm	0	1 1	op	U	T	Zn	Zd

size	op	U	T	Instruction Details	Condition
xx	0	0	0	SQDMULLB (vectors)	FEAT_SVE2 || FEAT_SME
xx	0	0	1	SQDMULLT (vectors)	FEAT_SVE2 || FEAT_SME
xx	1	0	0	SMULLB (vectors)	FEAT_SVE2 || FEAT_SME
xx	1	0	1	SMULLT (vectors)	FEAT_SVE2 || FEAT_SME
xx	1	1	0	UMULLB (vectors)	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	op	U	T	Instruction Details	Condition
xx	1	1	1	UMULLT (vectors)	FEAT_SVE2 || FEAT_SME
00	0	1	0	PMULLB — 128-bit element	FEAT_SVE_PMULL128
00	0	1	1	PMULLT — 128-bit element	FEAT_SVE_PMULL128
!= 00	0	1	0	PMULLB — 16-bit or 64-bit elements	FEAT_SVE2 || FEAT_SME
!= 00	0	1	1	PMULLT — 16-bit or 64-bit elements	FEAT_SVE2 || FEAT_SME

SVE Misc

SVE Misc

The encodings in this section are decoded from SVE encodings.

31–24	23	22	21	20–16	15	14	13–10	9–0
0 1 0 0 0 1 0 1	op0		0		1	0	op1	

op0	op1	Instruction details
0	10xx	SVE2 bitwise shift left long
1	10xx	UNALLOCATED
x	00xx	SVE2 integer add/subtract interleaved long
x	010x	SVE2 bitwise exclusive-OR interleaved
x	0110	SVE integer matrix multiply accumulate
x	0111	UNALLOCATED
x	11xx	SVE2 bitwise permute

SVE2 bitwise shift left long

The encodings in this section are decoded from SVE Misc.

31–24	23	22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 0 0 0 1 0 1	0	tszh	0	tszl	imm3	1 0 1 0	U	T	Zn	Zd

U	T	Instruction Details	Condition
0	0	SSHLLB	FEAT_SVE2 || FEAT_SME
0	1	SSHLLT	FEAT_SVE2 || FEAT_SME
1	0	USHLLB	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	T	Instruction Details	Condition
1	1	USHLLT	FEAT_SVE2 || FEAT_SME

SVE2 integer add/subtract interleaved long

The encodings in this section are decoded from SVE Misc.

31	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0 1 0 0 0 1 0 1		size	0	Zm	1 0 0 0	S	tb	Zn	Zd

S	tb	Instruction Details	Condition
0	0	SADDLBT	FEAT_SVE2 || FEAT_SME
0	1	UNALLOCATED	-
1	0	SSUBLBT	FEAT_SVE2 || FEAT_SME
1	1	SSUBLTB	FEAT_SVE2 || FEAT_SME

SVE2 bitwise exclusive-OR interleaved

The encodings in this section are decoded from SVE Misc.

31	24	23 22	21	20 16	15 11	10	9 5	4 0
0 1 0 0 0 1 0 1		size	0	Zm	1 0 0 1 0	tb	Zn	Zd

tb	Instruction Details	Condition
0	EORBT	FEAT_SVE2 || FEAT_SME
1	EORTB	FEAT_SVE2 || FEAT_SME

SVE integer matrix multiply accumulate

The encodings in this section are decoded from SVE Misc.

31	24	23 22	21	20 16	15 14	13 10	9 5	4 0
0 1 0 0 0 1 0 1		uns	0	Zm	1 0	0 1 1 0	Zn	Zd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

uns	Instruction Details	Condition
00	SMMLA	FEAT_SVE && FEAT_I8MM
01	UNALLOCATED	-
10	USMMLA	FEAT_SVE && FEAT_I8MM
11	UMMLA	FEAT_SVE && FEAT_I8MM

SVE2 bitwise permute

The encodings in this section are decoded from SVE Misc.

31	24	23	22	21	20	16	15	12	11	10	9	5	4	0
0 1 0 0 0 1 0 1		size		0	Zm		1 0 1 1		opc		Zn		Zd	

opc	Instruction Details	Condition
00	BEXT	FEAT_SVE_BitPerm
01	BDEP	FEAT_SVE_BitPerm
10	BGRP	FEAT_SVE_BitPerm
11	UNALLOCATED	-

SVE2 Accumulate

SVE2 Accumulate

The encodings in this section are decoded from SVE encodings.

31	24	23	22	21	20	17	16	15	14	13	11	10	0
0 1 0 0 0 1 0 1				0	op0			1	1	op1			

op0	op1	Instruction details
0000	011	SVE2 complex integer add
xxxx	00x	SVE2 integer absolute difference and accumulate long
xxxx	010	SVE2 integer add/subtract long with carry
xxxx	10x	SVE2 bitwise shift right and accumulate
xxxx	110	SVE2 bitwise shift and insert
xxxx	111	SVE2 integer absolute difference and accumulate
!= 0000	011	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 complex integer add

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20				17	16	15	14	13			11	10	9				5	4				0
0	1	0	0	0	1	0	1	size		0	0	0	0	0		op	1	1	0	1	1		rot	Zm						Zdn			

op	Instruction Details	Condition
0	CADD	FEAT_SVE2 || FEAT_SME
1	SQCADD	FEAT_SVE2 || FEAT_SME

SVE2 integer absolute difference and accumulate long

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20					16	15				12	11	10	9				5	4				0
0	1	0	0	0	1	0	1	size		0	Zm						1	1	0	0		U	T	Zn						Zda			

U	T	Instruction Details	Condition
0	0	SABALB	FEAT_SVE2 || FEAT_SME
0	1	SABALT	FEAT_SVE2 || FEAT_SME
1	0	UABALB	FEAT_SVE2 || FEAT_SME
1	1	UABALT	FEAT_SVE2 || FEAT_SME

SVE2 integer add/subtract long with carry

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20					16	15		14	13			11	10	9				5	4				0
0	1	0	0	0	1	0	1	size		0	Zm						1		1	0	1		0	T	Zn						Zda			

size	T	Instruction Details	Condition
0x	0	ADCLB	FEAT_SVE2 || FEAT_SME
0x	1	ADCLT	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	T	Instruction Details	Condition
1x	0	SBCLB	FEAT_SVE2 || FEAT_SME
1x	1	SBCLT	FEAT_SVE2 || FEAT_SME

SVE2 bitwise shift right and accumulate

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20	19	18		16	15			12	11	10	9		5	4		0
0	1	0	0	0	1	0	1	tszh		0	tszl		imm3			1	1	1	0	R	U	Zn			Zda		

R	U	Instruction Details	Condition
0	0	SSRA	FEAT_SVE2 || FEAT_SME
0	1	USRA	FEAT_SVE2 || FEAT_SME
1	0	SRSRA	FEAT_SVE2 || FEAT_SME
1	1	URSRA	FEAT_SVE2 || FEAT_SME

SVE2 bitwise shift and insert

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20	19	18		16	15	14	13		11	10	9		5	4		0
0	1	0	0	0	1	0	1	tszh		0	tszl		imm3			1	1	1	1	0	op	Zn			Zd		

op	Instruction Details	Condition
0	SRI	FEAT_SVE2 || FEAT_SME
1	SLI	FEAT_SVE2 || FEAT_SME

SVE2 integer absolute difference and accumulate

The encodings in this section are decoded from SVE2 Accumulate.

31							24	23	22	21	20				16	15	14	13			11	10	9		5	4		0
0	1	0	0	0	1	0	1	size		0	Zm					1	1	1	1	1		U	Zn			Zda		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Condition
0	SABA	FEAT_SVE2 || FEAT_SME
1	UABA	FEAT_SVE2 || FEAT_SME

SVE2 Narrowing

SVE2 Narrowing

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9		6	5	4		0
0	1	0	0	0	1	0	1	op0		1			op1		op2	0	op3				op4				op5			

op0	op1	op2	op3	op4	op5	Instruction details
0	00	0	10	x	x	SVE2 saturating extract narrow
0	00	1	10	0	0	SME2 multi-vec extract narrow
0	00	1	10	0	1	UNALLOCATED
0	00	1	10	1	x	UNALLOCATED
0	xx	x	0x	x	x	SVE2 bitwise shift right narrow
0	!=00	x	10	x	x	UNALLOCATED
1	xx	x	0x	0	0	SME2 multi-vec shift narrow
1	xx	x	0x	0	1	UNALLOCATED
1	xx	x	0x	1	x	UNALLOCATED
1	xx	x	10	x	x	UNALLOCATED
x	xx	x	11	x	x	SVE2 integer add/subtract narrow high part

SVE2 saturating extract narrow

The encodings in this section are decoded from SVE2 Narrowing.

31							24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9		5	4		0
0	1	0	0	0	1	0	1	0		1	tszl		0	0	0	0	1	0	opc		T	Zn			Zd		
									tszh																		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	T	Instruction Details	Condition
00	0	SQXTNB	FEAT_SVE2 || FEAT_SME
00	1	SQXTNT	FEAT_SVE2 || FEAT_SME
01	0	UQXTNB	FEAT_SVE2 || FEAT_SME
01	1	UQXTNT	FEAT_SVE2 || FEAT_SME
10	0	SQXTUNB	FEAT_SVE2 || FEAT_SME
10	1	SQXTUNT	FEAT_SVE2 || FEAT_SME
11	x	UNALLOCATED	-

SME2 multi-vec extract narrow

The encodings in this section are decoded from SVE2 Narrowing.

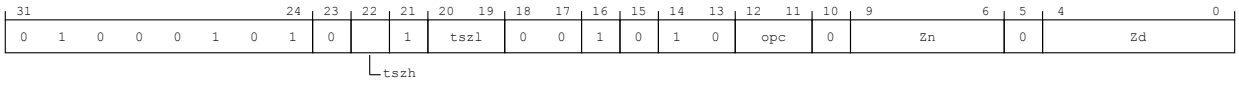

tszh	tszl	opc	Instruction Details	Condition
0	10	00	SQCVTN	FEAT_SME2 || FEAT_SVE2p1
0	10	01	UQCVTN	FEAT_SME2 || FEAT_SVE2p1
0	10	10	SQCVTUN	FEAT_SME2 || FEAT_SVE2p1
0	10	11	UNALLOCATED	-
0	!=10	xx	UNALLOCATED	-
1	xx	xx	UNALLOCATED	-

SVE2 bitwise shift right narrow

The encodings in this section are decoded from SVE2 Narrowing.

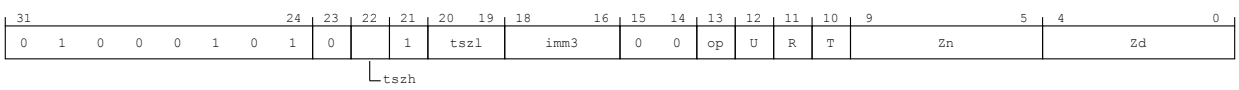

op	U	R	T	Instruction Details	Condition
0	0	0	0	SQSHRUNB	FEAT_SVE2 || FEAT_SME
0	0	0	1	SQSHRUNT	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	R	T	Instruction Details	Condition
0	0	1	0	SQRSHRUNB	FEAT_SVE2 || FEAT_SME
0	0	1	1	SQRSHRUNT	FEAT_SVE2 || FEAT_SME
0	1	0	0	SHRNB	FEAT_SVE2 || FEAT_SME
0	1	0	1	SHRNT	FEAT_SVE2 || FEAT_SME
0	1	1	0	RSHRNB	FEAT_SVE2 || FEAT_SME
0	1	1	1	RSHRNT	FEAT_SVE2 || FEAT_SME
1	0	0	0	SQSHRNB	FEAT_SVE2 || FEAT_SME
1	0	0	1	SQSHRNT	FEAT_SVE2 || FEAT_SME
1	0	1	0	SQRSHRNB	FEAT_SVE2 || FEAT_SME
1	0	1	1	SQRSHRNT	FEAT_SVE2 || FEAT_SME
1	1	0	0	UQSHRNB	FEAT_SVE2 || FEAT_SME
1	1	0	1	UQSHRNT	FEAT_SVE2 || FEAT_SME
1	1	1	0	UQRSHRNB	FEAT_SVE2 || FEAT_SME
1	1	1	1	UQRSHRNT	FEAT_SVE2 || FEAT_SME

SME2 multi-vec shift narrow

The encodings in this section are decoded from SVE2 Narrowing.

31								23	22	21	20				16	15	14	13	12	11	10	9				6	5	4				0
0	1	0	0	0	1	0	1	1	op0	1	opc					0	0	op1	U	R	0	Zn					0	Zd				

op0	opc	op1	U	R	Instruction Details	Condition
0	0xxxx	x	x	x	UNALLOCATED	-
0	1xxxx	x	x	0	UNALLOCATED	-
0	1xxxx	0	0	1	SQRSHRUN	FEAT_SME2 || FEAT_SVE2p1
0	1xxxx	0	1	1	UNALLOCATED	-
0	1xxxx	1	0	1	SQRSHRN	FEAT_SME2 || FEAT_SVE2p1
0	1xxxx	1	1	1	UQRSHRN	FEAT_SME2 || FEAT_SVE2p1
1	xxxxx	x	x	x	UNALLOCATED	-

SVE2 integer add/subtract narrow high part

The encodings in this section are decoded from SVE2 Narrowing.

31								24	23		22	21	20				16	15	14		13	12	11	10	9				5	4				0
0	1	0	0	0	1	0	1	size				1	Zm					0	1		1	S	R	T	Zn					Zd				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	R	T	Instruction Details	Condition
0	0	0	ADDHNB	FEAT_SVE2 || FEAT_SME
0	0	1	ADDHNT	FEAT_SVE2 || FEAT_SME
0	1	0	RADDHNB	FEAT_SVE2 || FEAT_SME
0	1	1	RADDHNT	FEAT_SVE2 || FEAT_SME
1	0	0	SUBHNB	FEAT_SVE2 || FEAT_SME
1	0	1	SUBHNT	FEAT_SVE2 || FEAT_SME
1	1	0	RSUBHNB	FEAT_SVE2 || FEAT_SME
1	1	1	RSUBHNT	FEAT_SVE2 || FEAT_SME

SVE2 Histogram (Segment) and Lookup Table

SVE2 Histogram (Segment) and Lookup Table

The encodings in this section are decoded from SVE encodings.

31–24	23	22	21	20–16	15–13	12–10	9–0
01000101		op0	1		101	op1	

op0	op1	Instruction details	Condition
0	0x1	UNALLOCATED	-
1	001	LUTI4 (8-bit and 16-bit) — Byte, single register table	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)
1	011	UNALLOCATED	-
x	000	HISTSEG	FEAT_SVE2
x	100	LUTI2 (8-bit and 16-bit) — Byte	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)
x	1x1	SVE2 lookup table with 4-bit indices and 16-bit element size	-
x	x10	LUTI2 (8-bit and 16-bit) — Halfword	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)

SVE2 lookup table with 4-bit indices and 16-bit element size

The encodings in this section are decoded from SVE2 Histogram (Segment) and Lookup Table.

31–24	23–22	21	20–16	15–12	11	10	9–5	4–0
01000101	i2	1	Zm	1011	op	1	Zn	Zd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details	Condition
0	LUTI4 (8-bit and 16-bit) — Halfword, two register table	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)
1	LUTI4 (8-bit and 16-bit) — Halfword, single register table	(FEAT_SVE2 && FEAT_LUT) || (FEAT_SME2 && FEAT_LUT)

SVE2 Crypto Extensions

SVE2 Crypto Extensions

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20		18	17	16	15			13	12		10	9			5	4			1	0
0	1	0	0	0	1	0	1			1	op0			op1		1	1	1		op2			op3								op4

op0	op1	op2	op3	op4	Instruction details
000	00	00x	00000	x	SVE2 crypto unary operations
000	00	00x	!= 00000	x	UNALLOCATED
000	01	00x	xxxxx	x	UNALLOCATED
000	1x	00x	xxxxx	x	SVE2 crypto destructive binary operations
xx0	1x	01x	xxxxx	x	SVE2 multi-vector AES single round (two registers)
xx1	1x	01x	xxxxx	x	SVE2 multi-vector AES single round (four registers)
xxx	xx	10x	xxxxx	x	SVE2 crypto constructive binary operations
xxx	xx	110	xxxxx	0	SVE2 Multi-vector polynomial multiply long
xxx	xx	111	xxxxx	0	SVE2 Multi-vector polynomial multiply long and accumulate vectors
xxx	xx	11x	xxxxx	1	UNALLOCATED
xxx	!= 1x	01x	xxxxx	x	UNALLOCATED
!= 000	xx	00x	xxxxx	x	UNALLOCATED

SVE2 crypto unary operations

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20		18	17	16	15					11	10	9				5	4			0
0	1	0	0	0	1	0	1	size		1	0	0	0	0	0	1	1	1	0	0		op	0	0	0	0	0	Zdn			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	op	Instruction Details	Condition
00	0	AESMC	FEAT_SVE_AES
00	1	AESIMC	FEAT_SVE_AES
!= 00	x	UNALLOCATED	-

SVE2 crypto destructive binary operations

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23 22	21	20		18	17	16	15				11	10	9	5	4	0
0	1	0	0	0	1	0	1	size	1	0	0	0	1	op	1	1	1	0	0	o2	Zm		Zdn	

size	op	o2	Instruction Details	Condition
00	0	0	AESE (vectors)	FEAT_SVE_AES
00	0	1	AESD (vectors)	FEAT_SVE_AES
00	1	0	SM4E	FEAT_SVE_SM4
00	1	1	UNALLOCATED	-
!= 00	x	x	UNALLOCATED	-

SVE2 multi-vector AES single round (two registers)

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23 22	21	20 19	18	17	16	15				11	10	9	5	4	1	0
0	1	0	0	0	1	0	1	size	1	i2	0	1	op	1	1	1	0	1	o2	Zm		Zdn		o3

size	op	o2	o3	Instruction Details	Condition
00	x	x	1	UNALLOCATED	-
00	0	0	0	AESE (indexed)	FEAT_SVE_AES2
00	0	1	0	AESD (indexed)	FEAT_SVE_AES2
00	1	0	0	AESEMC	FEAT_SVE_AES2
00	1	1	0	AESDIMC	FEAT_SVE_AES2
!= 00	x	x	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 multi-vector AES single round (four registers)

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20	19	18	17	16	15					11	10	9			5	4			2	1	0
0	1	0	0	0	1	0	1	size		1	i2		1	1	op	1	1	1	0	1		o2	Zm				Zdn				opc3	

size	op	o2	opc3	Instruction Details	Condition
00	x	x	!= 00	UNALLOCATED	-
00	0	0	00	AESE (indexed)	FEAT_SVE_AES2
00	0	1	00	AESD (indexed)	FEAT_SVE_AES2
00	1	0	00	AESEMC	FEAT_SVE_AES2
00	1	1	00	AESDIMC	FEAT_SVE_AES2
!= 00	x	x	xx	UNALLOCATED	-

SVE2 crypto constructive binary operations

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20				16	15					11	10	9			5	4					0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1	1	0		op	Zn				Zd					

size	op	Instruction Details	Condition
00	0	SM4EKEY	FEAT_SVE_SM4
00	1	RAX1	FEAT_SVE_SHA3
!= 00	x	UNALLOCATED	-

SVE2 Multi-vector polynomial multiply long

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20				16	15			13	12			10	9			5	4			1	0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1		1	1	0		Zn				Zd				0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Condition
00	PMULL	FEAT_SVE_AES2
!= 00	UNALLOCATED	-

SVE2 Multi-vector polynomial multiply long and accumulate vectors

The encodings in this section are decoded from SVE2 Crypto Extensions.

31							24	23	22	21	20				16	15		13	12		10	9			5	4			1	0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1	1	1	1	Zn				Zda				0

size	Instruction Details	Condition
00	PMLAL	FEAT_SVE_AES2
!= 00	UNALLOCATED	-

SVE2 FP8 widening multiply-add

SVE2 FP8 widening multiply-add

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15	14	13	12	11	10	9							0
0	1	1	0	0	1	0	0	op0	0	1						1	0	op1		1	0								

op0	op1	Instruction details
0	x	SVE2 FP8 multiply-add long long
1	0	SVE2 FP8 multiply-add long
1	1	UNALLOCATED

SVE2 FP8 multiply-add long long

The encodings in this section are decoded from SVE2 FP8 widening multiply-add.

31							24	23	22	21	20				16	15	14	13		12	11	10	9				5	4			0
0	1	1	0	0	1	0	0	0	0	1	Zm					1	0	TT			1	0	Zn					Zda			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TT	Instruction Details	Condition
00	FMLALLBB (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
01	FMLALLBT (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
10	FMLALLTB (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
11	FMLALLTT (vectors)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

SVE2 FP8 multiply-add long

The encodings in this section are decoded from SVE2 FP8 widening multiply-add.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	1	0	0	1	0	0	1	0	1	Zm				1	0	0	T	1	0	Zn				Zda			

T	Instruction Details	Condition
0	FMLALB (vectors, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
1	FMLALT (vectors, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

SVE2 floating-point unary operations - zeroing predicated

SVE2 floating-point unary operations - zeroing predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21			19	18			16	15	14										0
0	1	1	0	0	1	0	0			0	1	1		op0				1											

op0	Instruction details
00x	Floating-point round to integral value (predicated)
010	Floating-point convert (predicated)
011	Floating-point square root (predicated)
10x	Floating-point round and convert from integer (predicated)
11x	Floating-point log and convert to integer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Floating-point round to integral value (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23	22	21					17	16	15	14	13	12			10	9					5	4					0
0	1	1	0	0	1	0	0	size		0	1	1	0	0		op	1	opc2		Pg				Zn						Zd					

op	opc2	Instruction Details	Condition
0	00	FRINT<r> — Nearest with ties to even, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	01	FRINT<r> — Toward plus infinity, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	10	FRINT<r> — Toward minus infinity, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
0	11	FRINT<r> — Toward zero, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	00	FRINT<r> — Nearest with ties to away, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	01	UNALLOCATED	-
1	10	FRINT<r> — Current mode signalling inexact, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
1	11	FRINT<r> — Current mode, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Floating-point convert (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23	22	21		19	18		16	15	14	13	12			10	9					5	4					0
0	1	1	0	0	1	0	0	opc		0	1	1	0	1	0	1	opc2			Pg				Zn						Zd					

opc	opc2	Instruction Details	Condition
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTX	FEAT_SVE2p2 || FEAT_SME2p2
01	xx	UNALLOCATED	-
10	00	FCVT — Single-precision to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	01	FCVT — Half-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	10	BFCVT	FEAT_SVE2p2 || FEAT_SME2p2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	Instruction Details	Condition
11	00	FCVT — Double-precision to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	01	FCVT — Half-precision to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	10	FCVT — Double-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	11	FCVT — Single-precision to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Floating-point square root (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23 22	21 19	18 16	15	14 13	12 10	9 5	4 0
0	1	1	0	0	1	0	0	size	0 1 1	0 1 1	1	opc	Pg	Zn	Zd

opc	Instruction Details	Condition
00	FRECPX	FEAT_SVE2p2 || FEAT_SME2p2
01	FSQRT	FEAT_SVE2p2 || FEAT_SME2p2
1x	UNALLOCATED	-

Floating-point round and convert from integer (predicated)

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23 22	21 17	16	15	14	13	12 10	9 5	4 0
0	1	1	0	0	1	0	0	opc	0 1 1 1 0	o2	1	o3	U	Pg	Zn	Zd

opc	o2	o3	U	Instruction Details	Condition
00	0	x	0	FRINT32Z	FEAT_SVE2p2 || FEAT_SME2p2
00	0	x	1	FRINT32X	FEAT_SVE2p2 || FEAT_SME2p2
00	1	x	0	FRINT64Z	FEAT_SVE2p2 || FEAT_SME2p2
00	1	x	1	FRINT64X	FEAT_SVE2p2 || FEAT_SME2p2
01	0	0	x	UNALLOCATED	-
01	0	1	0	SCVTF (predicated) — 16-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	o3	U	Instruction Details	Condition
01	0	1	1	UCVTF (predicated) — 16-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	0	SCVTF (predicated) — 32-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	1	UCVTF (predicated) — 32-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	0	SCVTF (predicated) — 64-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	1	UCVTF (predicated) — 64-bit to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	0	x	x	UNALLOCATED	-
10	1	0	0	SCVTF (predicated) — 32-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	0	1	UCVTF (predicated) — 32-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	1	x	UNALLOCATED	-
11	0	0	0	SCVTF (predicated) — 32-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	0	1	UCVTF (predicated) — 32-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	1	x	UNALLOCATED	-
11	1	0	0	SCVTF (predicated) — 64-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	0	1	UCVTF (predicated) — 64-bit to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	0	SCVTF (predicated) — 64-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	1	UCVTF (predicated) — 64-bit to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

Floating-point log and convert to integer

The encodings in this section are decoded from SVE2 floating-point unary operations - zeroing predicated.

31							24	23	22	21				17	16	15	14	13	12		10	9		5	4		0
0	1	1	0	0	1	0	0	opc		0	1	1	1	1	o2	1	o3	U	Pg			Zn			Zd		

opc	o2	o3	U	Instruction Details	Condition
00	0	x	x	FLOGB	FEAT_SVE2p2 || FEAT_SME2p2
00	1	x	x	UNALLOCATED	-
01	0	0	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	o3	U	Instruction Details	Condition
01	0	1	0	FCVTZS — Half-precision to 16-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	0	1	1	FCVTZU — Half-precision to 16-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	0	FCVTZS — Half-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	0	1	FCVTZU — Half-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	0	FCVTZS — Half-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
01	1	1	1	FCVTZU — Half-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	0	x	x	UNALLOCATED	-
10	1	0	0	FCVTZS — Single-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	0	1	FCVTZU — Single-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	1	1	x	UNALLOCATED	-
11	0	0	0	FCVTZS — Double-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	0	1	FCVTZU — Double-precision to 32-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	0	1	x	UNALLOCATED	-
11	1	0	0	FCVTZS — Single-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	0	1	FCVTZU — Single-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	0	FCVTZS — Double-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	1	1	1	FCVTZU — Double-precision to 64-bit, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE floating-point widening multiply-add - indexed

SVE floating-point widening multiply-add - indexed

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20			16	15	14	13	12	11									0
0	1	1	0	0	1	0	0	op0		1					0	1	op1	0										

op0	op1	Instruction details
0	0	SVE BFloat16 floating-point dot product (indexed)
0	1	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
1	x	SVE floating-point multiply-add long (indexed)

SVE BFloat16 floating-point dot product (indexed)

The encodings in this section are decoded from SVE floating-point widening multiply-add - indexed.

31							24	23	22	21	20 19	18 16	15	14	13	12	11 10	9 5	4 0
0	1	1	0	0	1	0	0	0	op	1	i2	Zm	0	1	0	0	opc2	Zn	Zda

op	opc2	Instruction Details	Condition
0	x1	FDOT (2-way, indexed, FP8 to FP16)	FEAT_SSVE_FP8DOT2 || (FEAT_SVE2 && FEAT_FP8DOT2)
0	00	FDOT (2-way, indexed, FP16 to FP32)	FEAT_SME2 || FEAT_SVE2p1
0	10	UNALLOCATED	-
1	00	BFDOT (indexed)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	01	FDOT (4-way, indexed)	FEAT_SSVE_FP8DOT4 || (FEAT_SVE2 && FEAT_FP8DOT4)
1	1x	UNALLOCATED	-

SVE floating-point multiply-add long (indexed)

The encodings in this section are decoded from SVE floating-point widening multiply-add - indexed.

31							24	23	22	21	20 19	18 16	15	14	13	12	11	10	9 5	4 0
0	1	1	0	0	1	0	0	1	o2	1	i3h	Zm	0	1	op	0	i3l	T	Zn	Zda

o2	op	T	Instruction Details	Condition
0	0	0	FMLALB (indexed, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	0	1	FMLALT (indexed, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	1	0	FMLSLB (indexed)	FEAT_SVE2 || FEAT_SME
0	1	1	FMLSLT (indexed)	FEAT_SVE2 || FEAT_SME
1	0	0	BFMLALB (indexed)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	0	1	BFMLALT (indexed)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	1	0	BFMLSLB (indexed)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	BFMLSLT (indexed)	FEAT_SME2 || FEAT_SVE2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE floating-point widening multiply-add

SVE floating-point widening multiply-add

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20			16	15	14	13	12	11	10						0
0	1	1	0	0	1	0	0	op0		1					1	0	op1	0	0							

op0	op1	Instruction details
0	0	SVE BFloat16 floating-point dot product
0	1	UNALLOCATED
1	x	SVE floating-point multiply-add long

SVE BFloat16 floating-point dot product

The encodings in this section are decoded from SVE floating-point widening multiply-add.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	1	0	0	1	0	0	0	op	1	Zm				1	0	0	0	0	o2	Zn				Zda			

op	o2	Instruction Details	Condition
0	0	FDOT (2-way, vectors, FP16 to FP32)	FEAT_SME2 || FEAT_SVE2p1
0	1	FDOT (2-way, vectors, FP8 to FP16)	FEAT_SSVE_FP8DOT2 || (FEAT_SVE2 && FEAT_FP8DOT2)
1	0	BFDOT (vectors)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	1	FDOT (4-way, vectors)	FEAT_SSVE_FP8DOT4 || (FEAT_SVE2 && FEAT_FP8DOT4)

SVE floating-point multiply-add long

The encodings in this section are decoded from SVE floating-point widening multiply-add.

31							24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	1	0	0	1	0	0	1	o2	1	Zm				1	0	op	0	0	T	Zn				Zda			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o2	op	T	Instruction Details	Condition
0	0	0	FMLALB (vectors, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	0	1	FMLALT (vectors, FP16 to FP32)	FEAT_SVE2 || FEAT_SME
0	1	0	FMLSLB (vectors)	FEAT_SVE2 || FEAT_SME
0	1	1	FMLSLT (vectors)	FEAT_SVE2 || FEAT_SME
1	0	0	BFMLALB (vectors)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	0	1	BFMLALT (vectors)	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
1	1	0	BFMLSLB (vectors)	FEAT_SME2 || FEAT_SVE2p1
1	1	1	BFMLSLT (vectors)	FEAT_SME2 || FEAT_SVE2p1

SVE floating-point unary operations - unpredicated

SVE floating-point unary operations - unpredicated

The encodings in this section are decoded from SVE encodings.

31							24	23 22	21		19	18		16	15			12	11 10	9		6	5	4		0
0	1	1	0	0	1	0	1	op0	0	0	1	op1			0	0	1	1	op2				op3			

op0	op1	op2	op3	Instruction details
00	00x	xx	x	SVE2 FP8 upconverts
00	010	xx	0	SVE2 FP8 downconverts
00	010	xx	1	UNALLOCATED
00	011	xx	x	UNALLOCATED
xx	10x	xx	x	UNALLOCATED
xx	11x	00	x	SVE floating-point reciprocal estimate (unpredicated)
xx	11x	!= 00	x	UNALLOCATED
!= 00	0xx	xx	x	UNALLOCATED

SVE2 FP8 upconverts

The encodings in this section are decoded from SVE floating-point unary operations - unpredicated.

31							24	23	22	21					17	16	15			12	11 10	9				5	4				0
0	1	1	0	0	1	0	1	0	0	0	0	1	0	0		L	0	0	1	1	opc			Zn					Zd		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	opc	Instruction Details	Condition
0	00	F1CVT, F2CVT — F1CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
0	01	F1CVT, F2CVT — F2CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
0	10	BF1CVT, BF2CVT — BF1CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
0	11	BF1CVT, BF2CVT — BF2CVT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	00	F1CVTLT, F2CVTLT — F1CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	01	F1CVTLT, F2CVTLT — F2CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	10	BF1CVTLT, BF2CVTLT — BF1CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
1	11	BF1CVTLT, BF2CVTLT — BF2CVTLT	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)

SVE2 FP8 downconverts

The encodings in this section are decoded from SVE floating-point unary operations - unpredicated.

31–24	23–22	21–19	18–16	15–12	11–10	9–6	5	4–0
01100101	00	001	010	0011	opc	Zn	0	Zd

opc	Instruction Details	Condition
00	FCVTN	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
01	FCVTNB	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
10	BFCVTN	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)
11	FCVTNT (unpredicated)	(FEAT_SVE2 && FEAT_FP8) || (FEAT_SME2 && FEAT_FP8)

SVE floating-point reciprocal estimate (unpredicated)

The encodings in this section are decoded from SVE floating-point unary operations - unpredicated.

31–24	23–22	21–17	16	15–12	11–10	9–5	4–0
01100101	size	00111	op	0011	00	Zn	Zd

op	Instruction Details	Condition
0	FRECPE	FEAT_SVE || FEAT_SME
1	FRSQRTE	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE floating-point compare - with zero

SVE floating-point compare - with zero

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21		19	18	17	16	15		13	12												0
0	1	1	0	0	1	0	1			0	1	0	op0			0	0	1													

op0	Instruction details
0	SVE floating-point compare with zero
1	UNALLOCATED

SVE floating-point compare with zero

The encodings in this section are decoded from SVE floating-point compare - with zero.

31							24	23	22	21		19	18	17	16	15		13	12		10	9				5	4	3			0
0	1	1	0	0	1	0	1	size		0	1	0	0	eq	lt	0	0	1	Pg			Zn					ne	Pd			

eq	lt	ne	Instruction Details	Condition
0	0	0	FCM<cc> (zero) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	FCM<cc> (zero) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	FCM<cc> (zero) — Less than	FEAT_SVE || FEAT_SME
0	1	1	FCM<cc> (zero) — Less than or equal	FEAT_SVE || FEAT_SME
1	x	1	UNALLOCATED	-
1	0	0	FCM<cc> (zero) — Equal	FEAT_SVE || FEAT_SME
1	1	0	FCM<cc> (zero) — Not equal	FEAT_SVE || FEAT_SME

SVE floating-point accumulating reduction

SVE floating-point accumulating reduction

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21		19	18	17	16	15		13	12												0
0	1	1	0	0	1	0	1			0	1	1	op0			0	0	1													

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
0	SVE floating-point serial reduction (predicated)
1	UNALLOCATED

SVE floating-point serial reduction (predicated)

The encodings in this section are decoded from SVE floating-point accumulating reduction.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	size		0	1	1	0	opc		0	0	1	Pg			Zm			Vdn		

opc	Instruction Details	Condition
00	FADDA	FEAT_SVE
!= 00	UNALLOCATED	-

SVE floating-point arithmetic - predicated

SVE floating-point arithmetic - predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20	19	18		16	15		13	12		10	9		6	5		0
0	1	1	0	0	1	0	1			0	op0					1	0	0	op1			op2					

op0	op1	op2	Instruction details	Condition
0x	xxx	xxxx	SVE floating-point arithmetic (predicated)	-
10	000	xxxx	FTMAD	FEAT_SVE
10	!= 000	xxxx	UNALLOCATED	-
11	xxx	0000	SVE floating-point arithmetic with immediate (predicated)	-
11	xxx	!= 0000	UNALLOCATED	-

SVE floating-point arithmetic (predicated)

The encodings in this section are decoded from SVE floating-point arithmetic - predicated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20	19			16	15			13	12			10	9					5	4					0
0	1	1	0	0	1	0	1	size		0	0	opc				1	0	0		Pg				Zm						Zdn					

size	opc	Instruction Details	Condition
xx	0011	FSUBR (vectors)	FEAT_SVE || FEAT_SME
xx	1000	FABD	FEAT_SVE || FEAT_SME
xx	1010	FMULX	FEAT_SVE || FEAT_SME
xx	1011	UNALLOCATED	-
xx	1100	FDIVR	FEAT_SVE || FEAT_SME
xx	1101	FDIV	FEAT_SVE || FEAT_SME
xx	1110	FAMAX	(FEAT_SVE2 && FEAT_FAMINMAX) || (FEAT_SME2 && FEAT_FAMINMAX)
xx	1111	FAMIN	(FEAT_SVE2 && FEAT_FAMINMAX) || (FEAT_SME2 && FEAT_FAMINMAX)
00	0000	BFADD (predicated)	FEAT_SVE_B16B16
00	0001	BFSUB (predicated)	FEAT_SVE_B16B16
00	0010	BFMUL (vectors, predicated)	FEAT_SVE_B16B16
00	0100	BFMAXNM	FEAT_SVE_B16B16
00	0101	BFMINNM	FEAT_SVE_B16B16
00	0110	BFMAX	FEAT_SVE_B16B16
00	0111	BFMIN	FEAT_SVE_B16B16
00	1001	BFSCALE	FEAT_SVE_BFSCALE
!= 00	0000	FADD (vectors, predicated)	FEAT_SVE || FEAT_SME
!= 00	0001	FSUB (vectors, predicated)	FEAT_SVE || FEAT_SME
!= 00	0010	FMUL (vectors, predicated)	FEAT_SVE || FEAT_SME
!= 00	0100	FMAXNM (vectors)	FEAT_SVE || FEAT_SME
!= 00	0101	FMINNM (vectors)	FEAT_SVE || FEAT_SME
!= 00	0110	FMAX (vectors)	FEAT_SVE || FEAT_SME
!= 00	0111	FMIN (vectors)	FEAT_SVE || FEAT_SME
!= 00	1001	FSCALE	FEAT_SVE || FEAT_SME

SVE floating-point arithmetic with immediate (predicated)

The encodings in this section are decoded from SVE floating-point arithmetic - predicated.

31							24	23	22	21	20	19	18		16	15			13	12			10	9			6	5	4					0
0	1	1	0	0	1	0	1	size		0	1	1	opc			1	0	0		Pg				0	0	0	0	i1	Zdn					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
000	FADD (immediate)	FEAT_SVE || FEAT_SME
001	FSUB (immediate)	FEAT_SVE || FEAT_SME
010	FMUL (immediate)	FEAT_SVE || FEAT_SME
011	FSUBR (immediate)	FEAT_SVE || FEAT_SME
100	FMAXNM (immediate)	FEAT_SVE || FEAT_SME
101	FMINNM (immediate)	FEAT_SVE || FEAT_SME
110	FMAX (immediate)	FEAT_SVE || FEAT_SME
111	FMIN (immediate)	FEAT_SVE || FEAT_SME

SVE floating-point unary operations - merging predicated

SVE floating-point unary operations - merging predicated

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20		18	17	16	15			13	12											0
0	1	1	0	0	1	0	1			0	op0					1	0	1													

op0	Instruction details
00x	SVE floating-point round to integral value
010	SVE floating-point convert precision
011	SVE floating-point unary operations
10x	SVE integer convert to floating-point
11x	SVE floating-point convert to integer

SVE floating-point round to integral value

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21			19	18			16	15			13	12			10	9					5	4							0
0	1	1	0	0	1	0	1	size		0	0	0		opc				1	0	1		Pg				Zn						Zd							

opc	Instruction Details	Condition
000	FRINT<r> — Nearest with ties to even, merging	FEAT_SVE || FEAT_SME
001	FRINT<r> — Toward plus infinity, merging	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
010	FRINT<r> — Toward minus infinity, merging	FEAT_SVE || FEAT_SME
011	FRINT<r> — Toward zero, merging	FEAT_SVE || FEAT_SME
100	FRINT<r> — Nearest with ties to away, merging	FEAT_SVE || FEAT_SME
101	UNALLOCATED	-
110	FRINT<r> — Current mode signalling inexact, merging	FEAT_SVE || FEAT_SME
111	FRINT<r> — Current mode, merging	FEAT_SVE || FEAT_SME

SVE floating-point convert precision

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23 22	21	20		18	17 16	15		13	12	10	9	5	4	0
0	1	1	0	0	1	0	1	opc	0	0	1	0	opc2	1	0	1	Pg		Zn		Zd	

opc	opc2	Instruction Details	Condition
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTX	FEAT_SVE2 || FEAT_SME
01	xx	UNALLOCATED	-
10	00	FCVT — Single-precision to half-precision, merging	FEAT_SVE || FEAT_SME
10	01	FCVT — Half-precision to single-precision, merging	FEAT_SVE || FEAT_SME
10	10	BFCVT	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
11	00	FCVT — Double-precision to half-precision, merging	FEAT_SVE || FEAT_SME
11	01	FCVT — Half-precision to double-precision, merging	FEAT_SVE || FEAT_SME
11	10	FCVT — Double-precision to single-precision, merging	FEAT_SVE || FEAT_SME
11	11	FCVT — Single-precision to double-precision, merging	FEAT_SVE || FEAT_SME

SVE floating-point unary operations

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31							24	23	22	21	20		18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	size		0	0	1	1	opc		1	0	1	Pg			Zn			Zd		

opc	Instruction Details	Condition
00	FRECPX	FEAT_SVE || FEAT_SME
01	FSQRT	FEAT_SVE || FEAT_SME
1x	UNALLOCATED	-

SVE integer convert to floating-point

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	opc		0	1	0	opc2		U	1	0	1	Pg			Zn			Zd		

opc	opc2	U	Instruction Details	Condition
00	0x	0	FRINT32Z	FEAT_SVE2p2 || FEAT_SME2p2
00	0x	1	FRINT32X	FEAT_SVE2p2 || FEAT_SME2p2
00	1x	0	FRINT64Z	FEAT_SVE2p2 || FEAT_SME2p2
00	1x	1	FRINT64X	FEAT_SVE2p2 || FEAT_SME2p2
01	00	x	UNALLOCATED	-
01	01	0	SCVTF (predicated) — 16-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	01	1	UCVTF (predicated) — 16-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	10	0	SCVTF (predicated) — 32-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	10	1	UCVTF (predicated) — 32-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	11	0	SCVTF (predicated) — 64-bit to half-precision, merging	FEAT_SVE || FEAT_SME
01	11	1	UCVTF (predicated) — 64-bit to half-precision, merging	FEAT_SVE || FEAT_SME
10	10	0	SCVTF (predicated) — 32-bit to single-precision, merging	FEAT_SVE || FEAT_SME
10	10	1	UCVTF (predicated) — 32-bit to single-precision, merging	FEAT_SVE || FEAT_SME
10	!= 10	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	U	Instruction Details	Condition
11	00	0	SCVTF (predicated) — 32-bit to double-precision, merging	FEAT_SVE || FEAT_SME
11	00	1	UCVTF (predicated) — 32-bit to double-precision, merging	FEAT_SVE || FEAT_SME
11	01	x	UNALLOCATED	-
11	10	0	SCVTF (predicated) — 64-bit to single-precision, merging	FEAT_SVE || FEAT_SME
11	10	1	UCVTF (predicated) — 64-bit to single-precision, merging	FEAT_SVE || FEAT_SME
11	11	0	SCVTF (predicated) — 64-bit to double-precision, merging	FEAT_SVE || FEAT_SME
11	11	1	UCVTF (predicated) — 64-bit to double-precision, merging	FEAT_SVE || FEAT_SME

SVE floating-point convert to integer

The encodings in this section are decoded from SVE floating-point unary operations - merging predicated.

31							24	23	22	21		19	18	17	16	15		13	12		10	9		5	4		0
0	1	1	0	0	1	0	1	opc		0	1	1	opc2		U	1	0	1	Pg			Zn			Zd		

opc	opc2	U	Instruction Details	Condition
00	xx	0	FLOGB	FEAT_SVE2 || FEAT_SME
00	xx	1	UNALLOCATED	-
01	00	x	UNALLOCATED	-
01	01	0	FCVTZS — Half-precision to 16-bit, merging	FEAT_SVE || FEAT_SME
01	01	1	FCVTZU — Half-precision to 16-bit, merging	FEAT_SVE || FEAT_SME
01	10	0	FCVTZS — Half-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
01	10	1	FCVTZU — Half-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
01	11	0	FCVTZS — Half-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
01	11	1	FCVTZU — Half-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
10	10	0	FCVTZS — Single-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
10	10	1	FCVTZU — Single-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
10	!= 10	x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	U	Instruction Details	Condition
11	00	0	FCVTZS — Double-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
11	00	1	FCVTZU — Double-precision to 32-bit, merging	FEAT_SVE || FEAT_SME
11	01	x	UNALLOCATED	-
11	10	0	FCVTZS — Single-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
11	10	1	FCVTZU — Single-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
11	11	0	FCVTZS — Double-precision to 64-bit, merging	FEAT_SVE || FEAT_SME
11	11	1	FCVTZU — Double-precision to 64-bit, merging	FEAT_SVE || FEAT_SME

SVE floating-point multiply-add

SVE floating-point multiply-add

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20	16	15	14	0
0	1	1	0	0	1	0	1			1			op0		

op0	Instruction details
0	SVE floating-point multiply-accumulate writing addend
1	SVE floating-point multiply-accumulate writing multiplicand

SVE floating-point multiply-accumulate writing addend

The encodings in this section are decoded from SVE floating-point multiply-add.

31	30	29		27	26	25	24	23	22	21	20	16	15	13	12	11	10	9	5	4	0
size		1	1	1	VR	0	0	opc		1	Rm		option		S	1	0	Rn		Rt	

size	opc	Instruction Details	Condition
00	00	BFMLA (vectors)	FEAT_SVE_B16B16
00	01	BFMLS (vectors)	FEAT_SVE_B16B16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Condition
00	1x	UNALLOCATED	-
!= 00	00	FMLA (vectors)	FEAT_SVE || FEAT_SME
!= 00	01	FMLS (vectors)	FEAT_SVE || FEAT_SME
!= 00	10	FNMLA	FEAT_SVE || FEAT_SME
!= 00	11	FNMLS	FEAT_SVE || FEAT_SME

SVE floating-point multiply-accumulate writing multiplicand

The encodings in this section are decoded from SVE floating-point multiply-add.

31	24	23 22	21	20 16	15	14 13	12 10	9 5	4 0
0 1 1 0 0 1 0	1	size	1	Za	1	opc	Pg	Zm	Zdn

opc	Instruction Details	Condition
00	FMAD	FEAT_SVE || FEAT_SME
01	FMSB	FEAT_SVE || FEAT_SME
10	FNMAD	FEAT_SVE || FEAT_SME
11	FNMSB	FEAT_SVE || FEAT_SME

SVE Memory - 32-bit Gather and Unsized Contiguous

SVE Memory - 32-bit Gather and Unsized Contiguous

The encodings in this section are decoded from SVE encodings.

31	25	24 23	22 21	20 16	15 13	12 5	4	3 0
1 0 0 0 0 1 0		op0	op1		op2		op3	

op0	op1	op2	op3	Instruction details	Condition
00	x1	0xx	0	SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)	-
00	x1	0xx	1	UNALLOCATED	-
01	x1	0xx	x	SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)	-
10	x1	0xx	x	SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)	-
11	0x	000	0	LDR (predicate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
11	0x	000	1	UNALLOCATED	-
11	0x	010	x	LDR (vector)	FEAT_SVE || FEAT_SME
11	0x	0x1	x	UNALLOCATED	-
11	1x	0xx	0	SVE contiguous prefetch (scalar plus immediate)	-
11	1x	0xx	1	UNALLOCATED	-
xx	00	10x	x	SVE2 32-bit gather non-temporal load (vector plus scalar)	-
xx	00	110	0	SVE contiguous prefetch (scalar plus scalar)	-
xx	00	111	0	SVE 32-bit gather prefetch (vector plus immediate)	-
xx	00	11x	1	UNALLOCATED	-
xx	01	1xx	x	SVE 32-bit gather load (vector plus immediate)	-
xx	1x	1xx	x	SVE load and broadcast element	-
!= 11	x0	0xx	x	SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)	-

SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3				0
1	0	0	0	0	1	0	0	0	xs	1	Zm				0	msz		Pg			Rn				0	prfop				

msz	Instruction Details	Condition
00	PRFB (scalar plus vector)	FEAT_SVE
01	PRFH (scalar plus vector)	FEAT_SVE
10	PRFW (scalar plus vector)	FEAT_SVE
11	PRFD (scalar plus vector)	FEAT_SVE

SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4					0
1	0	0	0	0	1	0	0	1	xs	1	Zm				0	U	ff	Pg			Rn				Zt					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	ff	Instruction Details	Condition
0	0	LD1SH (scalar plus vector)	FEAT_SVE
0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
1	0	LD1H (scalar plus vector)	FEAT_SVE
1	1	LDFF1H (scalar plus vector)	FEAT_SVE

SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1	0	0	0	0	1	0	1	0	xs	1	Zm		0	U	ff	Pg		Rn		Zt	

U	ff	Instruction Details	Condition
0	x	UNALLOCATED	-
1	0	LD1W (scalar plus vector)	FEAT_SVE
1	1	LDFF1W (scalar plus vector)	FEAT_SVE

SVE contiguous prefetch (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24	23	22	21	16	15	14	13	12	10	9	5	4	3	0
1	0	0	0	0	1	0	1	1	1	imm6		0	msz		Pg		Rn		0	prfop	

msz	Instruction Details	Condition
00	PRFB (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	PRFH (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	PRFW (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	PRFD (scalar plus immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 32-bit gather non-temporal load (vector plus scalar)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31–25	24–23	22–21	20–16	15–14	13	12–10	9–5	4–0
1 0 0 0 0 1 0	msz	0 0	Rm	1 0	U	Pg	Zn	Zt

msz	U	Instruction Details	Condition
00	0	LDNT1SB	FEAT_SVE2
00	1	LDNT1B (vector plus scalar)	FEAT_SVE2
01	0	LDNT1SH	FEAT_SVE2
01	1	LDNT1H (vector plus scalar)	FEAT_SVE2
10	0	UNALLOCATED	-
10	1	LDNT1W (vector plus scalar)	FEAT_SVE2
11	x	UNALLOCATED	-

SVE contiguous prefetch (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0 0 0 1 0	msz	0 0	Rm	1 1 0	Pg	Rn	0	prfop

msz	Rm	Instruction Details	Condition
xx	11111	UNALLOCATED	-
00	!= 11111	PRFB (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	!= 11111	PRFH (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	!= 11111	PRFW (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	!= 11111	PRFD (scalar plus scalar)	FEAT_SVE || FEAT_SME

SVE 32-bit gather prefetch (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0 0 0 1 0	msz	0 0	imm5	1 1 1	Pg	Zn	0	prfop

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Condition
00	PRFB (vector plus immediate)	FEAT_SVE
01	PRFH (vector plus immediate)	FEAT_SVE
10	PRFW (vector plus immediate)	FEAT_SVE
11	PRFD (vector plus immediate)	FEAT_SVE

SVE 32-bit gather load (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1	0	0	0	0	1	0	msz	0 1	imm5	1	U	ff	Pg	Zn	Zt

msz	U	ff	Instruction Details	Condition
00	0	0	LD1SB (vector plus immediate)	FEAT_SVE
00	0	1	LDFF1SB (vector plus immediate)	FEAT_SVE
00	1	0	LD1B (vector plus immediate)	FEAT_SVE
00	1	1	LDFF1B (vector plus immediate)	FEAT_SVE
01	0	0	LD1SH (vector plus immediate)	FEAT_SVE
01	0	1	LDFF1SH (vector plus immediate)	FEAT_SVE
01	1	0	LD1H (vector plus immediate)	FEAT_SVE
01	1	1	LDFF1H (vector plus immediate)	FEAT_SVE
10	0	x	UNALLOCATED	-
10	1	0	LD1W (vector plus immediate)	FEAT_SVE
10	1	1	LDFF1W (vector plus immediate)	FEAT_SVE
11	x	x	UNALLOCATED	-

SVE load and broadcast element

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

31						25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1	0	0	0	0	1	0	dtypeh	1	imm6	1	dtypel	Pg	Rn	Zt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dtypeh	dtypel	Instruction Details	Condition
00	00	LD1RB — 8-bit element	FEAT_SVE || FEAT_SME
00	01	LD1RB — 16-bit element	FEAT_SVE || FEAT_SME
00	10	LD1RB — 32-bit element	FEAT_SVE || FEAT_SME
00	11	LD1RB — 64-bit element	FEAT_SVE || FEAT_SME
01	00	LD1RSW	FEAT_SVE || FEAT_SME
01	01	LD1RH — 16-bit element	FEAT_SVE || FEAT_SME
01	10	LD1RH — 32-bit element	FEAT_SVE || FEAT_SME
01	11	LD1RH — 64-bit element	FEAT_SVE || FEAT_SME
10	00	LD1RSH — 64-bit element	FEAT_SVE || FEAT_SME
10	01	LD1RSH — 32-bit element	FEAT_SVE || FEAT_SME
10	10	LD1RW — 32-bit element	FEAT_SVE || FEAT_SME
10	11	LD1RW — 64-bit element	FEAT_SVE || FEAT_SME
11	00	LD1RSB — 64-bit element	FEAT_SVE || FEAT_SME
11	01	LD1RSB — 32-bit element	FEAT_SVE || FEAT_SME
11	10	LD1RSB — 16-bit element	FEAT_SVE || FEAT_SME
11	11	LD1RD	FEAT_SVE || FEAT_SME

SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsized Contiguous.

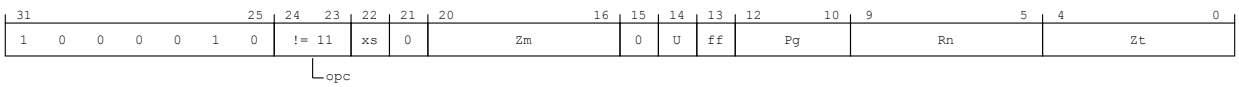

The following constraints also apply to this encoding: opc != ’11’

opc	U	ff	Instruction Details	Condition
00	0	0	LD1SB (scalar plus vector)	FEAT_SVE
00	0	1	LDFF1SB (scalar plus vector)	FEAT_SVE
00	1	0	LD1B (scalar plus vector)	FEAT_SVE
00	1	1	LDFF1B (scalar plus vector)	FEAT_SVE
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	x	UNALLOCATED	-
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE Memory - Contiguous Load

SVE Memory - Contiguous Load

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20	19		16	15		13	12		0
1	0	1	0	0	1	0			op0		op1				op2					

op0	op1	op2	Instruction details
00	0	111	SVE contiguous non-temporal load (scalar plus immediate)
00	1	001	SVE contiguous load (quadwords, scalar plus immediate)
00	1	111	SVE load multiple structures (quadwords, scalar plus immediate)
00	x	100	SVE contiguous load (quadwords, scalar plus scalar)
00	x	110	SVE contiguous non-temporal load (scalar plus scalar)
01	x	100	SVE load multiple structures (quadwords, scalar plus scalar)
1x	x	100	UNALLOCATED
xx	0	001	SVE load and broadcast quadword (scalar plus immediate)
xx	0	101	SVE contiguous load (scalar plus immediate)
xx	1	101	SVE contiguous non-fault load (scalar plus immediate)
xx	x	000	SVE load and broadcast quadword (scalar plus scalar)
xx	x	010	SVE contiguous load (scalar plus scalar)
xx	x	011	SVE contiguous first-fault load (scalar plus scalar)
!= 00	0	111	SVE load multiple structures (scalar plus immediate)
!= 00	1	001	UNALLOCATED
!= 00	1	111	UNALLOCATED
!= 00	x	110	SVE load multiple structures (scalar plus scalar)

SVE contiguous non-temporal load (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31						25	24	23	22	21	20	19			16	15			13	12		10	9			5	4			0
1	0	1	0	0	1	0	msz		0	0	0	imm4				1	1	1		Pg			Rn				Zt			

msz	Instruction Details	Condition
00	LDNT1B (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
01	LDNT1H (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
10	LDNT1W (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
11	LDNT1D (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

SVE contiguous load (quadwords, scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31						25	24	23	22	21	20	19			16	15			13	12		10	9			5	4			0
1	0	1	0	0	1	0	dtype		0	0	1	imm4				0	0	1		Pg			Rn				Zt			

dtype	Instruction Details	Condition
0x	UNALLOCATED	-
10	LD1W (scalar plus immediate, single register)	FEAT_SVE2p1
11	LD1D (scalar plus immediate, single register)	FEAT_SVE2p1

SVE load multiple structures (quadwords, scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31						25	24	23	22	21	20	19			16	15			13	12		10	9			5	4			0
1	0	1	0	0	1	0	num		0	0	1	imm4				1	1	1		Pg			Rn				Zt			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

num	Instruction Details	Condition
00	UNALLOCATED	-
01	LD2Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
10	LD3Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
11	LD4Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1

SVE contiguous load (quadwords, scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		dtype		0	0	Rm		1 0 0		Pg		Rn		Zt	

dtype	Rm	Instruction Details	Condition
0x	xxxxx	UNALLOCATED	-
1x	11111	UNALLOCATED	-
10	!= 11111	LD1W (scalar plus scalar, single register)	FEAT_SVE2p1
11	!= 11111	LD1D (scalar plus scalar, single register)	FEAT_SVE2p1

SVE contiguous non-temporal load (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		msz		0	0	Rm		1 1 0		Pg		Rn		Zt	

msz	Rm	Instruction Details	Condition
xx	11111	UNALLOCATED	-
00	!= 11111	LDNT1B (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
01	!= 11111	LDNT1H (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
10	!= 11111	LDNT1W (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
11	!= 11111	LDNT1D (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE load multiple structures (quadwords, scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		num		0	1	Rm		1 0 0		Pg		Rn		Zt	

num	Rm	Instruction Details	Condition
00	xxxxx	UNALLOCATED	-
01	!= 11111	LD2Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
10	!= 11111	LD3Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
11	!= 11111	LD4Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
!= 00	11111	UNALLOCATED	-

SVE load and broadcast quadword (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	23	22	21	20	19	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		msz		ssz		0	imm4		0 0 1		Pg		Rn		Zt	

msz	ssz	Instruction Details	Condition
xx	1x	UNALLOCATED	-
00	00	LD1RQB (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	01	LD1ROB (scalar plus immediate)	FEAT_F64MM
01	00	LD1RQH (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	01	LD1ROH (scalar plus immediate)	FEAT_F64MM
10	00	LD1RQW (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	01	LD1ROW (scalar plus immediate)	FEAT_F64MM
11	00	LD1RQD (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	01	LD1ROD (scalar plus immediate)	FEAT_F64MM

SVE contiguous load (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	25	24	21	20	19	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		dtype		0	imm4		1 0 1		Pg		Rn		Zt	

dtype	Instruction Details	Condition
0000	LD1B (scalar plus immediate, single register) — 8-bit element	FEAT_SVE || FEAT_SME
0001	LD1B (scalar plus immediate, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0010	LD1B (scalar plus immediate, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0011	LD1B (scalar plus immediate, single register) — 64-bit element	FEAT_SVE || FEAT_SME
0100	LD1SW (scalar plus immediate)	FEAT_SVE || FEAT_SME
0101	LD1H (scalar plus immediate, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0110	LD1H (scalar plus immediate, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0111	LD1H (scalar plus immediate, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1000	LD1SH (scalar plus immediate) — 64-bit element	FEAT_SVE || FEAT_SME
1001	LD1SH (scalar plus immediate) — 32-bit element	FEAT_SVE || FEAT_SME
1010	LD1W (scalar plus immediate, single register) — 32-bit element	FEAT_SVE || FEAT_SME
1011	LD1W (scalar plus immediate, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1100	LD1SB (scalar plus immediate) — 64-bit element	FEAT_SVE || FEAT_SME
1101	LD1SB (scalar plus immediate) — 32-bit element	FEAT_SVE || FEAT_SME
1110	LD1SB (scalar plus immediate) — 16-bit element	FEAT_SVE || FEAT_SME
1111	LD1D (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

SVE contiguous non-fault load (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31	25	24	21	20	19	16	15	13	12	10	9	5	4	0
1 0 1 0 0 1 0		dtype		1	imm4		1 0 1		Pg		Rn		Zt	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dtype	Instruction Details	Condition
0000	LDNF1B — 8-bit element	FEAT_SVE
0001	LDNF1B — 16-bit element	FEAT_SVE
0010	LDNF1B — 32-bit element	FEAT_SVE
0011	LDNF1B — 64-bit element	FEAT_SVE
0100	LDNF1SW	FEAT_SVE
0101	LDNF1H — 16-bit element	FEAT_SVE
0110	LDNF1H — 32-bit element	FEAT_SVE
0111	LDNF1H — 64-bit element	FEAT_SVE
1000	LDNF1SH — 64-bit element	FEAT_SVE
1001	LDNF1SH — 32-bit element	FEAT_SVE
1010	LDNF1W — 32-bit element	FEAT_SVE
1011	LDNF1W — 64-bit element	FEAT_SVE
1100	LDNF1SB — 64-bit element	FEAT_SVE
1101	LDNF1SB — 32-bit element	FEAT_SVE
1110	LDNF1SB — 16-bit element	FEAT_SVE
1111	LDNF1D	FEAT_SVE

SVE load and broadcast quadword (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31 … 25	24 23	22 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 0 1 0 0 1 0	msz	ssz	Rm	0 0 0	Pg	Rn	Zt

msz	ssz	Rm	Instruction Details	Condition
xx	0x	11111	UNALLOCATED	-
xx	1x	xxxxx	UNALLOCATED	-
00	00	!= 11111	LD1RQB (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	01	!= 11111	LD1ROB (scalar plus scalar)	FEAT_F64MM
01	00	!= 11111	LD1RQH (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	01	!= 11111	LD1ROH (scalar plus scalar)	FEAT_F64MM
10	00	!= 11111	LD1RQW (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	01	!= 11111	LD1ROW (scalar plus scalar)	FEAT_F64MM
11	00	!= 11111	LD1RQD (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	01	!= 11111	LD1ROD (scalar plus scalar)	FEAT_F64MM

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE contiguous load (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31 … 25	24 … 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 0 1 0 0 1 0	dtype	Rm	0 1 0	Pg	Rn	Zt

dtype	Rm	Instruction Details	Condition
xxxx	11111	UNALLOCATED	-
0000	!= 11111	LD1B (scalar plus scalar, single register) — 8-bit element	FEAT_SVE || FEAT_SME
0001	!= 11111	LD1B (scalar plus scalar, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0010	!= 11111	LD1B (scalar plus scalar, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0011	!= 11111	LD1B (scalar plus scalar, single register) — 64-bit element	FEAT_SVE || FEAT_SME
0100	!= 11111	LD1SW (scalar plus scalar)	FEAT_SVE || FEAT_SME
0101	!= 11111	LD1H (scalar plus scalar, single register) — 16-bit element	FEAT_SVE || FEAT_SME
0110	!= 11111	LD1H (scalar plus scalar, single register) — 32-bit element	FEAT_SVE || FEAT_SME
0111	!= 11111	LD1H (scalar plus scalar, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1000	!= 11111	LD1SH (scalar plus scalar) — 64-bit element	FEAT_SVE || FEAT_SME
1001	!= 11111	LD1SH (scalar plus scalar) — 32-bit element	FEAT_SVE || FEAT_SME
1010	!= 11111	LD1W (scalar plus scalar, single register) — 32-bit element	FEAT_SVE || FEAT_SME
1011	!= 11111	LD1W (scalar plus scalar, single register) — 64-bit element	FEAT_SVE || FEAT_SME
1100	!= 11111	LD1SB (scalar plus scalar) — 64-bit element	FEAT_SVE || FEAT_SME
1101	!= 11111	LD1SB (scalar plus scalar) — 32-bit element	FEAT_SVE || FEAT_SME
1110	!= 11111	LD1SB (scalar plus scalar) — 16-bit element	FEAT_SVE || FEAT_SME
1111	!= 11111	LD1D (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME

SVE contiguous first-fault load (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31 … 25	24 … 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 0 1 0 0 1 0	dtype	Rm	0 1 1	Pg	Rn	Zt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dtype	Instruction Details	Condition
0000	LDFF1B (scalar plus scalar) — 8-bit element	FEAT_SVE
0001	LDFF1B (scalar plus scalar) — 16-bit element	FEAT_SVE
0010	LDFF1B (scalar plus scalar) — 32-bit element	FEAT_SVE
0011	LDFF1B (scalar plus scalar) — 64-bit element	FEAT_SVE
0100	LDFF1SW (scalar plus scalar)	FEAT_SVE
0101	LDFF1H (scalar plus scalar) — 16-bit element	FEAT_SVE
0110	LDFF1H (scalar plus scalar) — 32-bit element	FEAT_SVE
0111	LDFF1H (scalar plus scalar) — 64-bit element	FEAT_SVE
1000	LDFF1SH (scalar plus scalar) — 64-bit element	FEAT_SVE
1001	LDFF1SH (scalar plus scalar) — 32-bit element	FEAT_SVE
1010	LDFF1W (scalar plus scalar) — 32-bit element	FEAT_SVE
1011	LDFF1W (scalar plus scalar) — 64-bit element	FEAT_SVE
1100	LDFF1SB (scalar plus scalar) — 64-bit element	FEAT_SVE
1101	LDFF1SB (scalar plus scalar) — 32-bit element	FEAT_SVE
1110	LDFF1SB (scalar plus scalar) — 16-bit element	FEAT_SVE
1111	LDFF1D (scalar plus scalar)	FEAT_SVE

SVE load multiple structures (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

31						25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1	0	1	0	0	1	0	msz	!= 00	0	imm4	1 1 1	Pg	Rn	Zt
								opc						

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Instruction Details	Condition
00	01	LD2B (scalar plus immediate)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	opc	Instruction Details	Condition
00	10	LD3B (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	11	LD4B (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	01	LD2H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	10	LD3H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	11	LD4H (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	01	LD2W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	10	LD3W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	11	LD4W (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	01	LD2D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	10	LD3D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	11	LD4D (scalar plus immediate)	FEAT_SVE || FEAT_SME

SVE load multiple structures (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load.

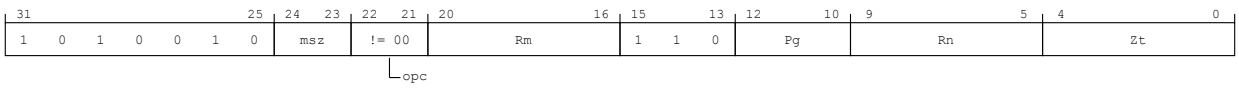

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Rm	Instruction Details	Condition
xx	!= 00	11111	UNALLOCATED	-
00	01	!= 11111	LD2B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	10	!= 11111	LD3B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	11	!= 11111	LD4B (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	01	!= 11111	LD2H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	10	!= 11111	LD3H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	11	!= 11111	LD4H (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	01	!= 11111	LD2W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	10	!= 11111	LD3W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	11	!= 11111	LD4W (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	01	!= 11111	LD2D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	10	!= 11111	LD3D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	11	!= 11111	LD4D (scalar plus scalar)	FEAT_SVE || FEAT_SME

SVE Memory - 64-bit Gather

SVE Memory - 64-bit Gather

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20		16	15		13	12		5	4	3		0
1	1	0	0	0	1	0	op0		op1					op2						op3			

op0	op1	op2	op3	Instruction details	Condition
00	00	101	x	LD1Q	FEAT_SVE2p1
00	01	0xx	1	UNALLOCATED	-
00	11	1xx	0	SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)	-
00	11	xxx	1	UNALLOCATED	-
00	x1	0xx	0	SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)	-
xx	00	111	0	SVE 64-bit gather prefetch (vector plus immediate)	-
xx	00	111	1	UNALLOCATED	-
xx	00	1x0	x	SVE2 64-bit gather non-temporal load (vector plus scalar)	-
xx	01	1xx	x	SVE 64-bit gather load (vector plus immediate)	-
xx	10	1xx	x	SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)	-
xx	x0	0xx	x	SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)	-
!=00	00	101	x	UNALLOCATED	-
!=00	11	1xx	x	SVE 64-bit gather load (scalar plus 64-bit scaled offsets)	-
!=00	x1	0xx	x	SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)	-

SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20		16	15	14	13	12		10	9		5	4	3		0
1	1	0	0	0	1	0	0	0	1	1		Zm		1	msz			Pg			Rn		0		prfop	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Condition
00	PRFB (scalar plus vector)	FEAT_SVE
01	PRFH (scalar plus vector)	FEAT_SVE
10	PRFW (scalar plus vector)	FEAT_SVE
11	PRFD (scalar plus vector)	FEAT_SVE

SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14		13	12			10	9				5	4	3				0
1	1	0	0	0	1	0	0	0	xs	1	Zm				0	msz			Pg				Rn					0	prfop				

msz	Instruction Details	Condition
00	PRFB (scalar plus vector)	FEAT_SVE
01	PRFH (scalar plus vector)	FEAT_SVE
10	PRFW (scalar plus vector)	FEAT_SVE
11	PRFD (scalar plus vector)	FEAT_SVE

SVE 64-bit gather prefetch (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15			13	12			10	9				5	4	3				0
1	1	0	0	0	1	0	msz		0	0	imm5				1	1		1	Pg				Zn					0	prfop				

msz	Instruction Details	Condition
00	PRFB (vector plus immediate)	FEAT_SVE
01	PRFH (vector plus immediate)	FEAT_SVE
10	PRFW (vector plus immediate)	FEAT_SVE
11	PRFD (vector plus immediate)	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 64-bit gather non-temporal load (vector plus scalar)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1	1	0	0	0	1	0	msz		0	0	Rm		1	U	0	Pg		Zn		Zt	

msz	U	Instruction Details	Condition
00	0	LDNT1SB	FEAT_SVE2
00	1	LDNT1B (vector plus scalar)	FEAT_SVE2
01	0	LDNT1SH	FEAT_SVE2
01	1	LDNT1H (vector plus scalar)	FEAT_SVE2
10	0	LDNT1SW	FEAT_SVE2
10	1	LDNT1W (vector plus scalar)	FEAT_SVE2
11	0	UNALLOCATED	-
11	1	LDNT1D (vector plus scalar)	FEAT_SVE2

SVE 64-bit gather load (vector plus immediate)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1	1	0	0	0	1	0	msz		0	1	imm5		1	U	ff	Pg		Zn		Zt	

msz	U	ff	Instruction Details	Condition
00	0	0	LD1SB (vector plus immediate)	FEAT_SVE
00	0	1	LDFF1SB (vector plus immediate)	FEAT_SVE
00	1	0	LD1B (vector plus immediate)	FEAT_SVE
00	1	1	LDFF1B (vector plus immediate)	FEAT_SVE
01	0	0	LD1SH (vector plus immediate)	FEAT_SVE
01	0	1	LDFF1SH (vector plus immediate)	FEAT_SVE
01	1	0	LD1H (vector plus immediate)	FEAT_SVE
01	1	1	LDFF1H (vector plus immediate)	FEAT_SVE
10	0	0	LD1SW (vector plus immediate)	FEAT_SVE
10	0	1	LDFF1SW (vector plus immediate)	FEAT_SVE
10	1	0	LD1W (vector plus immediate)	FEAT_SVE
10	1	1	LDFF1W (vector plus immediate)	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	U	ff	Instruction Details	Condition
11	0	x	UNALLOCATED	-
11	1	0	LD1D (vector plus immediate)	FEAT_SVE
11	1	1	LDFF1D (vector plus immediate)	FEAT_SVE

SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4			0
1	1	0	0	0	1	0	msz		1	0	Zm				1	U	ff	Pg			Rn				Zt			

msz	U	ff	Instruction Details	Condition
00	0	0	LD1SB (scalar plus vector)	FEAT_SVE
00	0	1	LDFF1SB (scalar plus vector)	FEAT_SVE
00	1	0	LD1B (scalar plus vector)	FEAT_SVE
00	1	1	LDFF1B (scalar plus vector)	FEAT_SVE
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31						25	24	23	22	21	20			16	15	14	13	12		10	9			5	4			0
1	1	0	0	0	1	0	msz		xs	0	Zm				0	U	ff	Pg			Rn				Zt			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	U	ff	Instruction Details	Condition
00	0	0	LD1SB (scalar plus vector)	FEAT_SVE
00	0	1	LDFF1SB (scalar plus vector)	FEAT_SVE
00	1	0	LD1B (scalar plus vector)	FEAT_SVE
00	1	1	LDFF1B (scalar plus vector)	FEAT_SVE
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

SVE 64-bit gather load (scalar plus 64-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

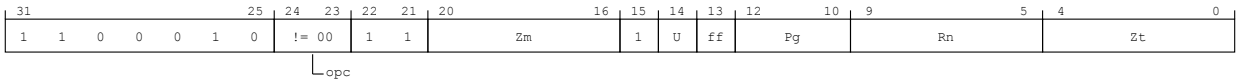

The following constraints also apply to this encoding: opc != ’00’

opc	U	ff	Instruction Details	Condition
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

The encodings in this section are decoded from SVE Memory - 64-bit Gather.

31	25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 0 0 0 1 0		!= 00 (opc)	xs	1	Zm	0	U	ff	Pg	Rn	Zt

The following constraints also apply to this encoding: opc != '00'

opc	U	ff	Instruction Details	Condition
01	0	0	LD1SH (scalar plus vector)	FEAT_SVE
01	0	1	LDFF1SH (scalar plus vector)	FEAT_SVE
01	1	0	LD1H (scalar plus vector)	FEAT_SVE
01	1	1	LDFF1H (scalar plus vector)	FEAT_SVE
10	0	0	LD1SW (scalar plus vector)	FEAT_SVE
10	0	1	LDFF1SW (scalar plus vector)	FEAT_SVE
10	1	0	LD1W (scalar plus vector)	FEAT_SVE
10	1	1	LDFF1W (scalar plus vector)	FEAT_SVE
11	0	x	UNALLOCATED	-
11	1	0	LD1D (scalar plus vector)	FEAT_SVE
11	1	1	LDFF1D (scalar plus vector)	FEAT_SVE

SVE Memory - Non-temporal and Quadword Scatter Store

SVE Memory - Non-temporal and Quadword Scatter Store

The encodings in this section are decoded from SVE encodings.

31	25	24 22	21	20 16	15 13	12 0
1 1 1 0 0 1 0		op0	op1		0 0 1	

op0	op1	Instruction details	Condition
000	1	ST1Q	FEAT_SVE2p1
xx0	0	SVE2 64-bit scatter non-temporal store (vector plus scalar)	-
xx1	0	SVE2 32-bit scatter non-temporal store (vector plus scalar)	-
!= 000	1	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 64-bit scatter non-temporal store (vector plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Quadword Scatter Store.

31	25	24 23	22	21	20 16	15 13	12 10	9 5	4 0
1 1 1 0 0 1 0		msz	0	0	Rm	0 0 1	Pg	Zn	Zt

msz	Instruction Details	Condition
00	STNT1B (vector plus scalar)	FEAT_SVE2
01	STNT1H (vector plus scalar)	FEAT_SVE2
10	STNT1W (vector plus scalar)	FEAT_SVE2
11	STNT1D (vector plus scalar)	FEAT_SVE2

SVE2 32-bit scatter non-temporal store (vector plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Quadword Scatter Store.

31	25	24 23	22	21	20 16	15 13	12 10	9 5	4 0
1 1 1 0 0 1 0		msz	1	0	Rm	0 0 1	Pg	Zn	Zt

msz	Instruction Details	Condition
00	STNT1B (vector plus scalar)	FEAT_SVE2
01	STNT1H (vector plus scalar)	FEAT_SVE2
10	STNT1W (vector plus scalar)	FEAT_SVE2
11	UNALLOCATED	-

SVE Memory - Non-temporal and Multi-register Contiguous Store

SVE Memory - Non-temporal and Multi-register Contiguous Store

The encodings in this section are decoded from SVE encodings.

31	25	24 23	22 21	20 16	15 13	12 0
1 1 1 0 0 1 0			op0		0 1 1	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	Instruction details
00	SVE contiguous non-temporal store (scalar plus scalar)
!= 00	SVE store multiple structures (scalar plus scalar)

SVE contiguous non-temporal store (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Contiguous Store.

31 … 25	24 23	22 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 1 1 0 0 1 0	msz	0 0	Rm	0 1 1	Pg	Rn	Zt

msz	Rm	Instruction Details	Condition
xx	11111	UNALLOCATED	-
00	!= 11111	STNT1B (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
01	!= 11111	STNT1H (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
10	!= 11111	STNT1W (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
11	!= 11111	STNT1D (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME

SVE store multiple structures (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Contiguous Store.

31 … 25	24 23	22 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 1 1 0 0 1 0	msz	!= 00 (opc)	Rm	0 1 1	Pg	Rn	Zt

The following constraints also apply to this encoding: opc != ’00’

msz	opc	Rm	Instruction Details	Condition
xx	!= 00	11111	UNALLOCATED	-
00	01	!= 11111	ST2B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	10	!= 11111	ST3B (scalar plus scalar)	FEAT_SVE || FEAT_SME
00	11	!= 11111	ST4B (scalar plus scalar)	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	opc	Rm	Instruction Details	Condition
01	01	!= 11111	ST2H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	10	!= 11111	ST3H (scalar plus scalar)	FEAT_SVE || FEAT_SME
01	11	!= 11111	ST4H (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	01	!= 11111	ST2W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	10	!= 11111	ST3W (scalar plus scalar)	FEAT_SVE || FEAT_SME
10	11	!= 11111	ST4W (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	01	!= 11111	ST2D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	10	!= 11111	ST3D (scalar plus scalar)	FEAT_SVE || FEAT_SME
11	11	!= 11111	ST4D (scalar plus scalar)	FEAT_SVE || FEAT_SME

SVE Memory - Contiguous Store and Unsized Contiguous

SVE Memory - Contiguous Store and Unsized Contiguous

The encodings in this section are decoded from SVE encodings.

31						25	24	22	21	20	19	16	15	14	13	12	5	4	3	0
1	1	1	0	0	1	0	op0		op1				0	op2	0			op3		

op0	op1	op2	op3	Instruction details	Condition
0xx	00	0	x	SVE store multiple structures (quadwords, scalar plus immediate)	-
0xx	01	0	x	UNALLOCATED	-
0xx	1x	0	x	SVE store multiple structures (quadwords, scalar plus scalar)	-
10x	xx	0	x	UNALLOCATED	-
110	xx	0	0	STR (predicate)	FEAT_SVE || FEAT_SME
110	xx	0	1	UNALLOCATED	-
110	xx	1	x	STR (vector)	FEAT_SVE || FEAT_SME
111	xx	0	x	UNALLOCATED	-
!= 110	xx	1	x	SVE contiguous store (scalar plus scalar)	-

SVE store multiple structures (quadwords, scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous.

31							24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	0
1	1	1	0	0	1	0	0	num		0	0	imm4		0	0	0	Pg		Rn		Zt	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

num	Instruction Details	Condition
00	UNALLOCATED	-
01	ST2Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
10	ST3Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1
11	ST4Q (scalar plus immediate)	FEAT_SVE2p1 || FEAT_SME2p1

SVE store multiple structures (quadwords, scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous.

31							24	23 22	21	20 16	15	14	13	12 10	9 5	4 0
1	1	1	0	0	1	0	0	num	1	Rm	0	0	0	Pg	Rn	Zt

num	Rm	Instruction Details	Condition
00	xxxxx	UNALLOCATED	-
01	!= 11111	ST2Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
10	!= 11111	ST3Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
11	!= 11111	ST4Q (scalar plus scalar)	FEAT_SVE2p1 || FEAT_SME2p1
!= 00	11111	UNALLOCATED	-

SVE contiguous store (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous.

31							25	24 22	21	20 16	15	14	13	12 10	9 5	4 0
1	1	1	0	0	1	0		!= 110	o2	Rm	0	1	0	Pg	Rn	Zt
								opc								

The following constraints also apply to this encoding: opc != ’110’

opc	o2	Rm	Instruction Details	Condition
0xx	x	11111	UNALLOCATED	-
00x	x	!= 11111	ST1B (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
01x	x	!= 11111	ST1H (scalar plus scalar, single register)	FEAT_SVE || FEAT_SME
10x	0	11111	UNALLOCATED	-
100	0	!= 11111	ST1W (scalar plus scalar, single register) — SVE2	FEAT_SVE2p1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	o2	Rm	Instruction Details	Condition
100	1	xxxxx	UNALLOCATED	-
101	x	!= 11111	ST1W (scalar plus scalar, single register) — SVE	FEAT_SVE || FEAT_SME
101	1	11111	UNALLOCATED	-
111	x	11111	UNALLOCATED	-
111	0	!= 11111	ST1D (scalar plus scalar, single register) — SVE2	FEAT_SVE2p1
111	1	!= 11111	ST1D (scalar plus scalar, single register) — SVE	FEAT_SVE || FEAT_SME

SVE Memory - Scatter

SVE Memory - Scatter

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20				16	15			13	12															0
1	1	1	0	0	1	0			op0							1	0	1																	

op0	Instruction details
00	SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)
10	SVE 64-bit scatter store (vector plus immediate)
11	SVE 32-bit scatter store (vector plus immediate)

SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter.

31						25	24	23	22	21	20				16	15			13	12			10	9					5	4				0
1	1	1	0	0	1	0	msz		0	0	Zm					1	0	1		Pg				Rn						Zt					

msz	Instruction Details	Condition
00	ST1B (scalar plus vector)	FEAT_SVE
01	ST1H (scalar plus vector)	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Condition
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter.

31 … 25	24 … 23	22 … 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 1 1 0 0 1 0	msz	0 1	Zm	1 0 1	Pg	Rn	Zt

msz	Instruction Details	Condition
00	UNALLOCATED	-
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 64-bit scatter store (vector plus immediate)

The encodings in this section are decoded from SVE Memory - Scatter.

31 … 25	24 … 23	22 … 21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 1 1 0 0 1 0	msz	1 0	imm5	1 0 1	Pg	Zn	Zt

msz	Instruction Details	Condition
00	ST1B (vector plus immediate)	FEAT_SVE
01	ST1H (vector plus immediate)	FEAT_SVE
10	ST1W (vector plus immediate)	FEAT_SVE
11	ST1D (vector plus immediate)	FEAT_SVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE 32-bit scatter store (vector plus immediate)

The encodings in this section are decoded from SVE Memory - Scatter.

31–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	1 1	imm5	1 0 1	Pg	Zn	Zt

msz	Instruction Details	Condition
00	ST1B (vector plus immediate)	FEAT_SVE
01	ST1H (vector plus immediate)	FEAT_SVE
10	ST1W (vector plus immediate)	FEAT_SVE
11	UNALLOCATED	-

SVE Memory - Contiguous Store with Immediate Offset

SVE Memory - Contiguous Store with Immediate Offset

The encodings in this section are decoded from SVE encodings.

31–25	24–23	22–21	20	19–16	15–13	12–0
1 1 1 0 0 1 0		op0	op1		1 1 1	

op0	op1	Instruction details
00	1	SVE contiguous non-temporal store (scalar plus immediate)
xx	0	SVE contiguous store (scalar plus immediate)
!=00	1	SVE store multiple structures (scalar plus immediate)

SVE contiguous non-temporal store (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset.

31–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1 0 0 1 0	msz	0 0	1	imm4	1 1 1	Pg	Rn	Zt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Condition
00	STNT1B (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
01	STNT1H (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
10	STNT1W (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
11	STNT1D (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME

SVE contiguous store (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset.

31 … 25	24 … 23	22 … 21	20	19 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 1 1 0 0 1 0	msz	opc	0	imm4	1 1 1	Pg	Rn	Zt

msz	opc	Instruction Details	Condition
00	xx	ST1B (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
01	xx	ST1H (scalar plus immediate, single register)	FEAT_SVE || FEAT_SME
10	00	ST1W (scalar plus immediate, single register) — SVE2	FEAT_SVE2p1
10	01	UNALLOCATED	-
10	1x	ST1W (scalar plus immediate, single register) — SVE	FEAT_SVE || FEAT_SME
11	0x	UNALLOCATED	-
11	10	ST1D (scalar plus immediate, single register) — SVE2	FEAT_SVE2p1
11	11	ST1D (scalar plus immediate, single register) — SVE	FEAT_SVE || FEAT_SME

SVE store multiple structures (scalar plus immediate)

The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset.

31 … 25	24 … 23	22 … 21	20	19 … 16	15 … 13	12 … 10	9 … 5	4 … 0
1 1 1 0 0 1 0	msz	!= 00	1	imm4	1 1 1	Pg	Rn	Zt
		opc						

The following constraints also apply to this encoding: opc != ’00’

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	opc	Instruction Details	Condition
00	01	ST2B (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	10	ST3B (scalar plus immediate)	FEAT_SVE || FEAT_SME
00	11	ST4B (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	01	ST2H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	10	ST3H (scalar plus immediate)	FEAT_SVE || FEAT_SME
01	11	ST4H (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	01	ST2W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	10	ST3W (scalar plus immediate)	FEAT_SVE || FEAT_SME
10	11	ST4W (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	01	ST2D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	10	ST3D (scalar plus immediate)	FEAT_SVE || FEAT_SME
11	11	ST4D (scalar plus immediate)	FEAT_SVE || FEAT_SME

SVE Memory - Scatter with Optional Sign Extend

SVE Memory - Scatter with Optional Sign Extend

The encodings in this section are decoded from SVE encodings.

31						25	24	23	22	21	20			16	15	14	13	12														0
1	1	1	0	0	1	0			op0						1		0															

op0	Instruction details
00	SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)
10	SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)
11	SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24	23	22	21	20			16	15	14	13	12			10	9				5	4				0
1	1	1	0	0	1	0	msz		0	0	Zm				1	xs	0	Pg				Rn					Zt				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	Instruction Details	Condition
00	ST1B (scalar plus vector)	FEAT_SVE
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1	1	1	0	0	1	0	msz	0 1	Zm	1	xs	0	Pg	Rn	Zt

msz	Instruction Details	Condition
00	UNALLOCATED	-
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	ST1D (scalar plus vector)	FEAT_SVE

SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1	1	1	0	0	1	0	msz	1 0	Zm	1	xs	0	Pg	Rn	Zt

msz	Instruction Details	Condition
00	ST1B (scalar plus vector)	FEAT_SVE
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend.

31						25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	0
1	1	1	0	0	1	0	msz		1	1	Zm		1	xs	0	Pg		Rn		Zt	

msz	Instruction Details	Condition
00	UNALLOCATED	-
01	ST1H (scalar plus vector)	FEAT_SVE
10	ST1W (scalar plus vector)	FEAT_SVE
11	UNALLOCATED	-

SVE integer add/subtract vectors (unpredicated)

The encodings in this section are decoded from SVE encodings.

31		29	28				24	23	22	21	20	16	15		13	12	10	9	5	4	0
0	0	0	0	0	1	0	0	size		1	Zm		0	0	0	opc		Zn		Zd	

size	opc	Instruction Details	Condition
xx	000	ADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	001	SUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	100	SQADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	101	UQADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	110	SQSUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
xx	111	UQSUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
11	010	ADDPT (unpredicated)	FEAT_SVE && FEAT_CPA
11	011	SUBPT (unpredicated)	FEAT_SVE && FEAT_CPA
!= 11	01x	UNALLOCATED	-

SVE address generation

The encodings in this section are decoded from SVE encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–29	28–24	23–22	21	20–16	15–12	11–10	9–5	4–0
0 0 0	0 0 1 0 0	opc	1	Zm	1 0 1 0	msz	Zn	Zd

opc	Instruction Details	Condition
00	ADR — Unpacked 32-bit signed offsets	FEAT_SVE
01	ADR — Unpacked 32-bit unsigned offsets	FEAT_SVE
1x	ADR — Packed offsets	FEAT_SVE

SVE table lookup (three sources)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0	0 0 1 0 1	size	1	Zm	0 0 1 0 1	op	Zn	Zd

op	Instruction Details	Condition
0	TBL	FEAT_SVE2 || FEAT_SME
1	TBX	FEAT_SVE2 || FEAT_SME

SVE permute vector elements

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0 1	size	1	Zm	0 1 1	opc	Zn	Zd

opc	Instruction Details	Condition
000	ZIP1, ZIP2 (vectors) — Low halves	FEAT_SVE || FEAT_SME
001	ZIP1, ZIP2 (vectors) — High halves	FEAT_SVE || FEAT_SME
010	UZP1, UZP2 (vectors) — Even	FEAT_SVE || FEAT_SME
011	UZP1, UZP2 (vectors) — Odd	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
100	TRN1, TRN2 (vectors) — Even	FEAT_SVE || FEAT_SME
101	TRN1, TRN2 (vectors) — Odd	FEAT_SVE || FEAT_SME
11x	UNALLOCATED	-

SVE integer compare with unsigned immediate

The encodings in this section are decoded from SVE encodings.

31	29	28	24	23	22	21	20	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0 0				1	imm7		lt	Pg		Zn		ne	Pd	

lt	ne	Instruction Details	Condition
0	0	CMP<cc> (immediate) — Higher or same	FEAT_SVE || FEAT_SME
0	1	CMP<cc> (immediate) — Higher	FEAT_SVE || FEAT_SME
1	0	CMP<cc> (immediate) — Lower	FEAT_SVE || FEAT_SME
1	1	CMP<cc> (immediate) — Lower or same	FEAT_SVE || FEAT_SME

SVE predicate logical operations

The encodings in this section are decoded from SVE encodings.

31	29	28	24	23	22	21	20	19	16	15	14	13	10	9	8	5	4	3	0
0 0 1		0 0 1 0 1		op	S	0	0	Pm		0	1	Pg		o2	Pn		o3	Pd	

op	S	o2	o3	Instruction Details	Condition
0	0	0	0	AND (predicates)	FEAT_SVE || FEAT_SME
0	0	0	1	BIC (predicates)	FEAT_SVE || FEAT_SME
0	0	1	0	EOR (predicates)	FEAT_SVE || FEAT_SME
0	0	1	1	SEL (predicates)	FEAT_SVE || FEAT_SME
0	1	0	0	ANDS	FEAT_SVE || FEAT_SME
0	1	0	1	BICS	FEAT_SVE || FEAT_SME
0	1	1	0	EORS	FEAT_SVE || FEAT_SME
0	1	1	1	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	S	o2	o3	Instruction Details	Condition
1	0	0	0	ORR (predicates)	FEAT_SVE || FEAT_SME
1	0	0	1	ORN (predicates)	FEAT_SVE || FEAT_SME
1	0	1	0	NOR	FEAT_SVE || FEAT_SME
1	0	1	1	NAND	FEAT_SVE || FEAT_SME
1	1	0	0	ORRS	FEAT_SVE || FEAT_SME
1	1	0	1	ORNS	FEAT_SVE || FEAT_SME
1	1	1	0	NORS	FEAT_SVE || FEAT_SME
1	1	1	1	NANDS	FEAT_SVE || FEAT_SME

SVE integer compare with signed immediate

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0 1	size	0	imm5	op	0	o2	Pg	Zn	ne	Pd

op	o2	ne	Instruction Details	Condition
0	0	0	CMP<cc> (immediate) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	CMP<cc> (immediate) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	CMP<cc> (immediate) — Less than	FEAT_SVE || FEAT_SME
0	1	1	CMP<cc> (immediate) — Less than or equal	FEAT_SVE || FEAT_SME
1	0	0	CMP<cc> (immediate) — Equal	FEAT_SVE || FEAT_SME
1	0	1	CMP<cc> (immediate) — Not equal	FEAT_SVE || FEAT_SME
1	1	x	UNALLOCATED	-

SVE broadcast predicate element

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23	22	21	20–18	17–16	15–14	13–10	9	8–5	4	3–0
0 0 1	0 0 1 0 1	i1	tszh	1	tszl	Rv	0 1	Pn	S	Pm	0	Pd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	Instruction Details	Condition
0	PSEL	FEAT_SME || FEAT_SVE2p1
1	UNALLOCATED	-

SVE two-way dot product

The encodings in this section are decoded from SVE encodings.

31–29	28–21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0 0 0 0 0	Zm	1 1 0 0 1	U	Zn	Zda

U	Instruction Details	Condition
0	SDOT (2-way, vectors)	FEAT_SME2 || FEAT_SVE2p1
1	UDOT (2-way, vectors)	FEAT_SME2 || FEAT_SVE2p1

SVE two-way dot product (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0 0 1 0 0	opc	1 1 0 0 1	U	Zn	Zda

U	Instruction Details	Condition
0	SDOT (2-way, indexed)	FEAT_SME2 || FEAT_SVE2p1
1	UDOT (2-way, indexed)	FEAT_SME2 || FEAT_SVE2p1

SVE integer clamp

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0 0	size	0	Zm	1 1 0 0 0	U	Zn	Zd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Instruction Details	Condition
0	SCLAMP	FEAT_SME || FEAT_SVE2p1
1	UCLAMP	FEAT_SME || FEAT_SVE2p1

SVE2 multiply-add (checked pointer)

The encodings in this section are decoded from SVE encodings.

31							24	23	22	21	20				16	15			12	11	10	9				5	4				0
0	1	0	0	0	1	0	0	opc		0	Zm					1	1	0	1	o2	0	Zn					Zda				

opc	o2	Instruction Details	Condition
11	0	MLAPT	FEAT_SVE && FEAT_CPA
11	1	MADPT	FEAT_SVE && FEAT_CPA
!= 11	x	UNALLOCATED	-

SVE permute vector elements (quadwords)

The encodings in this section are decoded from SVE encodings.

31		29	28				24	23	22	21	20				16	15		13	12		10	9				5	4				0
0	1	0	0	0	1	0	0	size		0	Zm					1	1	1	opc			Zn					Zd				

opc	Instruction Details	Condition
000	ZIPQ1	FEAT_SVE2p1 || FEAT_SME2p1
001	ZIPQ2	FEAT_SVE2p1 || FEAT_SME2p1
010	UZPQ1	FEAT_SVE2p1 || FEAT_SME2p1
011	UZPQ2	FEAT_SVE2p1 || FEAT_SME2p1
10x	UNALLOCATED	-
110	TBLQ	FEAT_SVE2p1 || FEAT_SME2p1
111	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE2 character match

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–13	12–10	9–5	4	3–0
0 1 0	0 0 1 0 1	opc	1	Zm	1 0 0	Pg	Zn	op	Pd

opc	op	Instruction Details	Condition
0x	0	MATCH	FEAT_SVE2
0x	1	NMATCH	FEAT_SVE2
1x	x	UNALLOCATED	-

SVE2 FP8 matrix multiply-accumulate

The encodings in this section are decoded from SVE encodings.

31–29	28–23	22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0 0 0	op	1	Zm	1 1 1 0 0 0	Zn	Zda

op	Instruction Details	Condition
0	FMMLA (widening, FP8 to FP32)	FEAT_SVE2 && FEAT_F8F32MM
1	FMMLA (widening, FP8 to FP16)	FEAT_SVE2 && FEAT_F8F16MM

SVE2 FP8 multiply-add long (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23	22–21	20–19	18–16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0 0	T	0 1	i4h	Zm	0 1 0 1	i4l	Zn	Zda

T	Instruction Details	Condition
0	FMLALB (indexed, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
1	FMLALT (indexed, FP8 to FP16)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE floating-point convert (top, predicated)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
011	00100	opc	0000	opc2	101	Pg	Zn	Zd

opc	opc2	Instruction Details	Condition
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTXNT	FEAT_SVE2p2 || FEAT_SME2p2
01	xx	UNALLOCATED	-
10	00	FCVTNT (predicated) — Single-precision to half-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	01	FCVTLT — Half-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
10	10	BFCVTNT	FEAT_SVE2p2 || FEAT_SME2p2
11	0x	UNALLOCATED	-
11	10	FCVTNT (predicated) — Double-precision to single-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2
11	11	FCVTLT — Single-precision to double-precision, zeroing	FEAT_SVE2p2 || FEAT_SME2p2

SVE floating-point convert precision odd elements

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
011	00100	opc	0010	opc2	101	Pg	Zn	Zd

opc	opc2	Instruction Details	Condition
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTXNT	FEAT_SVE2 || FEAT_SME
01	xx	UNALLOCATED	-
10	00	FCVTNT (predicated) — Single-precision to half-precision, merging	FEAT_SVE2 || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	opc2	Instruction Details	Condition
10	01	FCVTLT — Half-precision to single-precision, merging	FEAT_SVE2 || FEAT_SME
10	10	BFCVTNT	(FEAT_SVE && FEAT_BF16) || (FEAT_SME && FEAT_BF16)
11	0x	UNALLOCATED	-
11	10	FCVTNT (predicated) — Double-precision to single-precision, merging	FEAT_SVE2 || FEAT_SME
11	11	FCVTLT — Single-precision to double-precision, merging	FEAT_SVE2 || FEAT_SME

SVE2 floating-point pairwise operations

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0 0		0 1 0	opc	1 0 0	Pg	Zm	Zdn

opc	Instruction Details	Condition
000	FADDP	FEAT_SVE2 || FEAT_SME
001	UNALLOCATED	-
01x	UNALLOCATED	-
100	FMAXNMP	FEAT_SVE2 || FEAT_SME
101	FMINNMP	FEAT_SVE2 || FEAT_SME
110	FMAXP	FEAT_SVE2 || FEAT_SME
111	FMINP	FEAT_SVE2 || FEAT_SME

SVE floating-point recursive reduction (quadwords)

The encodings in this section are decoded from SVE encodings.

31 29	28 24	23 22	21 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0 0	size	0 1 0	opc	1 0 1	Pg	Zn	Vd

opc	Instruction Details	Condition
000	FADDQV	FEAT_SVE2p1 || FEAT_SME2p1
001	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
01x	UNALLOCATED	-
100	FMAXNMQV	FEAT_SVE2p1 || FEAT_SME2p1
101	FMINNMQV	FEAT_SVE2p1 || FEAT_SME2p1
110	FMAXQV	FEAT_SVE2p1 || FEAT_SME2p1
111	FMINQV	FEAT_SVE2p1 || FEAT_SME2p1

SVE floating-point multiply-add (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0 0	size	1	opc	0 0 0 0	o2	op	Zn	Zda

size	o2	op	Instruction Details	Condition
0x	0	0	FMLA (indexed) — Half-precision	FEAT_SVE || FEAT_SME
0x	0	1	FMLS (indexed) — Half-precision	FEAT_SVE || FEAT_SME
0x	1	0	BFMLA (indexed)	FEAT_SVE_B16B16
0x	1	1	BFMLS (indexed)	FEAT_SVE_B16B16
1x	1	x	UNALLOCATED	-
10	0	0	FMLA (indexed) — Single-precision	FEAT_SVE || FEAT_SME
10	0	1	FMLS (indexed) — Single-precision	FEAT_SVE || FEAT_SME
11	0	0	FMLA (indexed) — Double-precision	FEAT_SVE || FEAT_SME
11	0	1	FMLS (indexed) — Double-precision	FEAT_SVE || FEAT_SME

SVE floating-point complex multiply-add (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0 0	size	1	opc	0 0 0 1	rot	Zn	Zda

size	Instruction Details	Condition
0x	UNALLOCATED	-
10	FCMLA (indexed) — Half-precision	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Instruction Details	Condition
11	FCMLA (indexed) — Single-precision	FEAT_SVE || FEAT_SME

SVE FP clamp

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0 0	size	1	Zm	0 0 1 0 0 1	Zn	Zd

size	Instruction Details	Condition
00	BFCLAMP	FEAT_SVE_B16B16
!= 00	FCLAMP	FEAT_SME2 || FEAT_SVE2p1

SVE floating-point multiply (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0 0	size	1	opc	0 0 1 0	o2	0	Zn	Zd

size	o2	Instruction Details	Condition
0x	0	FMUL (indexed) — Half-precision	FEAT_SVE || FEAT_SME
0x	1	BFMUL (indexed)	FEAT_SVE_B16B16
1x	1	UNALLOCATED	-
10	0	FMUL (indexed) — Single-precision	FEAT_SVE || FEAT_SME
11	0	FMUL (indexed) — Double-precision	FEAT_SVE || FEAT_SME

SVE2 FP8 multiply-add long long (indexed)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–19	18–16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0 0	TT	1	i4h	Zm	1 1 0 0	i4l	Zn	Zda

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TT	Instruction Details	Condition
00	FMLALLBB (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
01	FMLALLBT (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
10	FMLALLTB (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)
11	FMLALLTT (indexed)	FEAT_SSVE_FP8FMA || (FEAT_SVE2 && FEAT_FP8FMA)

SVE floating-point matrix multiply accumulate

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0 0	opc	1	Zm	1 1 1 0 0 1	Zn	Zda

opc	Instruction Details	Condition
00	FMMLA (widening, FP16 to FP32)	FEAT_SVE_F16F32MM
01	BFMMLA (widening)	FEAT_SVE && FEAT_BF16
10	FMMLA (non-widening) — 32-bit element	FEAT_F32MM
11	FMMLA (non-widening) — 64-bit element	FEAT_F64MM

SVE floating-point recursive reduction

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0 1	size	0 0 0	opc	0 0 1	Pg	Zn	Vd

opc	Instruction Details	Condition
000	FADDV	FEAT_SVE || FEAT_SME
001	UNALLOCATED	-
01x	UNALLOCATED	-
100	FMAXNMV	FEAT_SVE || FEAT_SME
101	FMINNMV	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	Instruction Details	Condition
110	FMAXV	FEAT_SVE || FEAT_SME
111	FMINV	FEAT_SVE || FEAT_SME

SVE floating-point arithmetic (unpredicated)

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0 1	size	0	Zm	0 0 0	opc	Zn	Zd

size	opc	Instruction Details	Condition
xx	011	FTSMUL	FEAT_SVE
xx	10x	UNALLOCATED	-
xx	110	FRECPS	FEAT_SVE || FEAT_SME
xx	111	FRSQRTS	FEAT_SVE || FEAT_SME
00	000	BFADD (unpredicated)	FEAT_SVE_B16B16
00	001	BFSUB (unpredicated)	FEAT_SVE_B16B16
00	010	BFMUL (vectors, unpredicated)	FEAT_SVE_B16B16
!= 00	000	FADD (vectors, unpredicated)	FEAT_SVE || FEAT_SME
!= 00	001	FSUB (vectors, unpredicated)	FEAT_SVE || FEAT_SME
!= 00	010	FMUL (vectors, unpredicated)	FEAT_SVE || FEAT_SME

SVE floating-point compare vectors

The encodings in this section are decoded from SVE encodings.

31–29	28–24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 1 1	0 0 1 0 1	size	0	Zm	op	1	o2	Pg	Zn	o3	Pd

op	o2	o3	Instruction Details	Condition
0	0	0	FCM<cc> (vectors) — Greater than or equal	FEAT_SVE || FEAT_SME
0	0	1	FCM<cc> (vectors) — Greater than	FEAT_SVE || FEAT_SME
0	1	0	FCM<cc> (vectors) — Equal	FEAT_SVE || FEAT_SME

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	o2	o3	Instruction Details	Condition
0	1	1	FCM<cc> (vectors) — Not equal	FEAT_SVE || FEAT_SME
1	0	0	FCM<cc> (vectors) — Unordered	FEAT_SVE || FEAT_SME
1	0	1	FAC<cc> — Greater than or equal	FEAT_SVE || FEAT_SME
1	1	0	UNALLOCATED	-
1	1	1	FAC<cc> — Greater than	FEAT_SVE || FEAT_SME

Data Processing -- Immediate

Data Processing – Immediate

The encodings in this section are decoded from A64 instruction set encoding.

31	30 29	28 26	25 22	21 0
	op0	1 0 0	op1	

op0	op1	Instruction details
11	111x	Data-processing (1 source immediate)
xx	00xx	PC-rel. addressing
xx	010x	Add/subtract (immediate)
xx	0110	Add/subtract (immediate, with tags)
xx	0111	Min/max (immediate)
xx	100x	Logical (immediate)
xx	101x	Move wide (immediate)
xx	110x	Bitfield
!=11	111x	Extract

Data-processing (1 source immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 23	22 21	20 5	4 0
sf	1	1	1 0 0 1 1 1	opc	imm16	Rd

sf	opc	Rd	Instruction Details	Condition
0	xx	xxxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	opc	Rd	Instruction Details	Condition
1	0x	!= 11111	UNALLOCATED	-
1	00	11111	AUTIASPPC	FEAT_PAuth_LR
1	01	11111	AUTIBSPPC	FEAT_PAuth_LR
1	1x	xxxxx	UNALLOCATED	-

PC-rel. addressing

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 24	23 5	4 0
op	immlo	1 0 0 0 0	immhi	Rd

op	Instruction Details
0	ADR
1	ADRP

Add/subtract (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 23	22	21 10	9 5	4 0
sf	op	S	1 0 0 0 1 0	sh	imm12	Rn	Rd

sf	op	S	Instruction Details
0	0	0	ADD (immediate) — 32-bit
0	0	1	ADDS (immediate) — 32-bit
0	1	0	SUB (immediate) — 32-bit
0	1	1	SUBS (immediate) — 32-bit
1	0	0	ADD (immediate) — 64-bit
1	0	1	ADDS (immediate) — 64-bit
1	1	0	SUB (immediate) — 64-bit
1	1	1	SUBS (immediate) — 64-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Add/subtract (immediate, with tags)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 26	25 22	21 16	15 14	13 10	9 5	4 0
sf	op	S	1 0 0	0 1 1 0	imm6	op3	imm4	Rn	Rd

sf	op	S	Instruction Details	Condition
0	x	x	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	ADDG	FEAT_MTE
1	1	0	SUBG	FEAT_MTE

Min/max (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30	29	28 26	25 22	21 18	17 10	9 5	4 0
sf	op	S	1 0 0	0 1 1 1	opc	imm8	Rn	Rd

sf	op	S	opc	Instruction Details	Condition
x	0	0	!= 00xx	UNALLOCATED	-
x	0	1	xxxx	UNALLOCATED	-
x	1	x	xxxx	UNALLOCATED	-
0	0	0	0000	SMAX (immediate) — 32-bit	FEAT_CSSC
0	0	0	0001	UMAX (immediate) — 32-bit	FEAT_CSSC
0	0	0	0010	SMIN (immediate) — 32-bit	FEAT_CSSC
0	0	0	0011	UMIN (immediate) — 32-bit	FEAT_CSSC
1	0	0	0000	SMAX (immediate) — 64-bit	FEAT_CSSC
1	0	0	0001	UMAX (immediate) — 64-bit	FEAT_CSSC
1	0	0	0010	SMIN (immediate) — 64-bit	FEAT_CSSC
1	0	0	0011	UMIN (immediate) — 64-bit	FEAT_CSSC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Logical (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	opc	1 0 0 1 0 0	N	immr	imms	Rn	Rd

sf	opc	N	Instruction Details
0	xx	1	UNALLOCATED
0	00	0	AND (immediate) — 32-bit
0	01	0	ORR (immediate) — 32-bit
0	10	0	EOR (immediate) — 32-bit
0	11	0	ANDS (immediate) — 32-bit
1	00	x	AND (immediate) — 64-bit
1	01	x	ORR (immediate) — 64-bit
1	10	x	EOR (immediate) — 64-bit
1	11	x	ANDS (immediate) — 64-bit

Move wide (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22 21	20 5	4 0
sf	opc	1 0 0 1 0 1	hw	imm16	Rd

sf	opc	hw	Instruction Details
x	01	0x	UNALLOCATED
0	xx	1x	UNALLOCATED
0	00	0x	MOVN — 32-bit
0	10	0x	MOVZ — 32-bit
0	11	0x	MOVK — 32-bit
1	00	xx	MOVN — 64-bit
1	01	1x	UNALLOCATED
1	10	xx	MOVZ — 64-bit
1	11	xx	MOVK — 64-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bitfield

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	opc	1 0 0 1 1 0	N	immr	imms	Rn	Rd

sf	opc	N	Instruction Details
0	xx	1	UNALLOCATED
0	00	0	SBFM — 32-bit
0	01	0	BFM — 32-bit
0	10	0	UBFM — 32-bit
0	11	0	UNALLOCATED
1	xx	0	UNALLOCATED
1	00	1	SBFM — 64-bit
1	01	1	BFM — 64-bit
1	10	1	UBFM — 64-bit
1	11	1	UNALLOCATED

Extract

The encodings in this section are decoded from Data Processing – Immediate.

31	30 29	28 23	22	21	20 16	15 10	9 5	4 0
sf	!= 11	1 0 0 1 1 1	N	o0	Rm	imms	Rn	Rd

op21

The following constraints also apply to this encoding: op21 != ’11’

sf	op21	N	o0	imms	Instruction Details
x	00	x	1	xxxxxx	UNALLOCATED
x	01	x	x	xxxxxx	UNALLOCATED
x	10	x	x	xxxxxx	UNALLOCATED
0	00	0	0	0xxxxx	EXTR — 32-bit
0	00	0	0	1xxxxx	UNALLOCATED
0	00	1	0	xxxxxx	UNALLOCATED
1	00	0	0	xxxxxx	UNALLOCATED
1	00	1	0	xxxxxx	EXTR — 64-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Branches, Exception Generating and System instructions

Branches, Exception Generating and System instructions

The encodings in this section are decoded from A64 instruction set encoding.

31–29	28–26	25–12	11–5	4–0
op0	1 0 1	op1		op2

op0	op1	op2	Instruction details
010	00xxxxxxxxxxxx	xxxxx	Conditional branch (immediate)
010	01xxxxxxxxxxxx	xxxxx	Miscellaneous branch (immediate)
011	00xxxxxxxx1xxx	xxxxx	Compare bytes/halfwords in registers and branch
01x	1xxxxxxxxxxxxx	xxxxx	UNALLOCATED
110	00xxxxxxxxxxxx	xxxxx	Exception generation
110	010000000x00xx	xxxxx	UNALLOCATED
110	010000001000xx	xxxxx	UNALLOCATED
110	01000000110000	xxxxx	UNALLOCATED
110	01000000110001	xxxxx	System instructions with register argument
110	01000000110010	11111	Hints
110	01000000110010	!= 11111	UNALLOCATED
110	01000000110011	xxxxx	Barriers
110	01000001xx00xx	xxxxx	UNALLOCATED
110	0100000xxx0100	xxxxx	PSTATE
110	0100000xxx0101	xxxxx	UNALLOCATED
110	0100000xxx011x	xxxxx	UNALLOCATED
110	0100000xxx1xxx	xxxxx	UNALLOCATED
110	0100100xxxxxxx	xxxxx	UNALLOCATED
110	0100x01xxxxxxx	xxxxx	System instructions
110	0100x1xxxxxxxx	xxxxx	System register move
110	0101x00xxxxxxx	xxxxx	UNALLOCATED
110	0101x01xxxxxxx	xxxxx	System pair instructions
110	0101x1xxxxxxxx	xxxxx	System register pair move
110	011xxxxxxxxxxx	xxxxx	UNALLOCATED
110	1xxxxxxxxxxxxx	xxxxx	Unconditional branch (register)
111	00xxxxxxxx1xxx	xxxxx	UNALLOCATED
111	1xxxxxxxxxxxxx	xxxxx	UNALLOCATED
x00	xxxxxxxxxxxxxx	xxxxx	Unconditional branch (immediate)
x01	0xxxxxxxxxxxxx	xxxxx	Compare and branch (immediate)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
x01	1xxxxxxxxxxxxx	xxxxx	Test and branch (immediate)
x11	00xxxxxxxx00xx	xxxxx	Compare registers and branch
x11	00xxxxxxxx01xx	xxxxx	UNALLOCATED
x11	01xxxxxxxxx0xx	xxxxx	Compare register with immediate and branch
x11	01xxxxxxxxx1xx	xxxxx	UNALLOCATED

Conditional branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	29	28	24	23	5	4	3	0
0 1 0		1 0 1 0 0		imm19		o0	cond	

o0	Instruction Details	Condition
0	B.cond	-
1	BC.cond	FEAT_HBC

Miscellaneous branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	29	28	24	23	21	20	5	4	0
0 1 0		1 0 1 0 1		opc		imm16		op2	

opc	op2	Instruction Details			Condition
00x	!= 11111	UNALLOCATED			-
000	11111	RETAASPPC, RETAASPPC	RETABSPPC	—	FEAT_PAuth_LR
001	11111	RETAASPPC, RETABSPPC	RETABSPPC	—	FEAT_PAuth_LR
!= 00x	xxxxx	UNALLOCATED			-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Compare bytes/halfwords in registers and branch

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1 0 0	cc	Rm	1	H	imm9	Rt

cc	H	Instruction Details	Condition
000	0	CBB<cc> — Greater than	FEAT_CMPBR
000	1	CBH<cc> — Greater than	FEAT_CMPBR
001	0	CBB<cc> — Greater than or equal	FEAT_CMPBR
001	1	CBH<cc> — Greater than or equal	FEAT_CMPBR
010	0	CBB<cc> — Higher	FEAT_CMPBR
010	1	CBH<cc> — Higher	FEAT_CMPBR
011	0	CBB<cc> — Higher or same	FEAT_CMPBR
011	1	CBH<cc> — Higher or same	FEAT_CMPBR
10x	x	UNALLOCATED	-
110	0	CBB<cc> — Equal	FEAT_CMPBR
110	1	CBH<cc> — Equal	FEAT_CMPBR
111	0	CBB<cc> — Not equal	FEAT_CMPBR
111	1	CBH<cc> — Not equal	FEAT_CMPBR

Exception generation

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–24	23–21	20–5	4–2	1–0
1 1 0	1 0 1 0 0	opc	imm16	op2	LL

opc	op2	LL	Instruction Details
xxx	!= 000	xx	UNALLOCATED
x11	000	xx	UNALLOCATED
000	000	00	UNALLOCATED
000	000	01	SVC
000	000	10	HVC
000	000	11	SMC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	op2	LL	Instruction Details
001	000	00	BRK
001	000	!= 00	UNALLOCATED
010	000	00	HLT
010	000	!= 00	UNALLOCATED
1x0	000	xx	UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	DCPS1
101	000	10	DCPS2
101	000	11	DCPS3

System instructions with register argument

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 0 1	CRm	op2	Rt

CRm	op2	Instruction Details	Condition
0000	000	WFET	FEAT_WFxT
0000	001	WFIT	FEAT_WFxT
0000	!= 00x	UNALLOCATED	-
!= 0000	xxx	UNALLOCATED	-

Hints

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	CRm	op2	1 1 1 1 1

CRm	op2	Instruction Details	Condition
xxxx	xxx	HINT	-
0000	000	NOP	-
0000	001	YIELD	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	op2	Instruction Details	Condition
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0000	110	DGH	FEAT_DGH
0000	111	XPACD, XPACI, XPACLRI	FEAT_PAuth
0001	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716	FEAT_PAuth
0001	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716	FEAT_PAuth
0001	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716	FEAT_PAuth
0001	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716	FEAT_PAuth
0010	000	ESB	FEAT_RAS
0010	001	PSB	FEAT_SPE
0010	010	TSB	FEAT_TRF
0010	011	GCSB	FEAT_GCS
0010	100	CSDB	-
0010	110	CLRBHB	FEAT_CLRBHB
0011	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ	FEAT_PAuth
0011	001	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP	FEAT_PAuth
0011	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ	FEAT_PAuth
0011	011	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP	FEAT_PAuth
0011	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ	FEAT_PAuth
0011	101	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP	FEAT_PAuth
0011	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ	FEAT_PAuth
0011	111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP	FEAT_PAuth
0100	xx0	BTI	FEAT_BTI
0100	111	PACM	FEAT_PAuth_LR
0101	000	CHKFEAT	FEAT_CHK
0110	00x	STSHH	FEAT_PCDPHINT

Barriers

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	op2	Rt

CRm	op2	Rt	Instruction Details	Condition
xxxx	xxx	!= 11111	UNALLOCATED	-
xxxx	000	11111	UNALLOCATED	-
xxxx	010	11111	CLREX	-
xxxx	100	11111	DSB — Memory barrier	-
xxxx	101	11111	DMB	-
xxxx	110	11111	ISB	-
xxxx	111	11111	SB	FEAT_SB
xx10	001	11111	DSB — Memory nXS barrier	FEAT_XS
xx10	011	11111	UNALLOCATED	-
!= xx10	0x1	11111	UNALLOCATED	-

PSTATE

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1 0 1 0 0 0 0 0	op1	0 1 0 0	CRm	op2	Rt

op1	op2	Rt	Instruction Details	Condition
xxx	xxx	!= 11111	UNALLOCATED	-
000	000	11111	CFINV	FEAT_FlagM
000	001	11111	XAFLAG	FEAT_FlagM2
000	010	11111	AXFLAG	FEAT_FlagM2
000	011	11111	MSR (immediate)	-
000	1xx	11111	MSR (immediate)	-
!= 000	xxx	11111	MSR (immediate)	-

System instructions

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 29	28 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1 0 1 0 0	L	0 1	op1	CRn	CRm	op2	Rt

L	Instruction Details
0	SYS
1	SYSL

System register move

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 22	21	20	19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1 0 1 0 0	L	1	o0	op1	CRn	CRm	op2	Rt

L	Instruction Details
0	MSR (register)
1	MRS

System pair instructions

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31 29	28 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1 0 1 0 1	L	0 1	op1	CRn	CRm	op2	Rt

L	Instruction Details	Condition
0	SYSP	FEAT_SYSINSTR128
1	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System register pair move

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1 0 1 0 1	L	1	o0	op1	CRn	CRm	op2	Rt

L	Instruction Details	Condition
0	MSRR	FEAT_SYSREG128
1	MRRS	FEAT_SYSREG128

Unconditional branch (register)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31–29	28–25	24–21	20–16	15–10	9–5	4–0
1 1 0	1 0 1 1	opc	op2	op3	Rn	op4

opc	op2	op3	Rn	op4	Instruction Details	Condition
xxxx	11111	!= 0000xx	xxxxx	xxxxx	UNALLOCATED	-
xxxx	!= 11111	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
00x0	11111	000000	xxxxx	!= 00000	UNALLOCATED	-
00x0	11111	000001	xxxxx	xxxxx	UNALLOCATED	-
000x	11111	00001x	xxxxx	!= 11111	UNALLOCATED	-
0000	11111	000000	xxxxx	00000	BR	-
0000	11111	000010	xxxxx	11111	BRAA, BRAAZ, BRAB, BRABZ — Key A, zero modifier	FEAT_PAuth
0000	11111	000011	xxxxx	11111	BRAA, BRAAZ, BRAB, BRABZ — Key B, zero modifier	FEAT_PAuth
0001	11111	000000	xxxxx	00000	BLR	-
0001	11111	000000	xxxxx	!= 00000	UNALLOCATED	-
0001	11111	000001	xxxxx	xxxxx	UNALLOCATED	-
0001	11111	000010	xxxxx	11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key A, zero modifier	FEAT_PAuth
0001	11111	000011	xxxxx	11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key B, zero modifier	FEAT_PAuth
0010	11111	000000	xxxxx	00000	RET	-
0010	11111	00001x	!= 11111	xxxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	op2	op3	Rn	op4	Instruction Details	Condition
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	FEAT_PAuth
0010	11111	000010	11111	!= 11111	RETAASPPCR, RETABSPPCR — RETAASPPCR	FEAT_PAuth_LR
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	FEAT_PAuth
0010	11111	000011	11111	!= 11111	RETAASPPCR, RETABSPPCR — RETABSPPCR	FEAT_PAuth_LR
0011	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-
010x	11111	0000xx	!= 11111	xxxxx	UNALLOCATED	-
010x	11111	000000	11111	!= 00000	UNALLOCATED	-
010x	11111	000001	11111	xxxxx	UNALLOCATED	-
010x	11111	00001x	11111	xxxx0	UNALLOCATED	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	00001x	11111	0xxx1	UNALLOCATED	-
0100	11111	00001x	11111	10xx1	UNALLOCATED	-
0100	11111	00001x	11111	110x1	UNALLOCATED	-
0100	11111	00001x	11111	11101	UNALLOCATED	-
0100	11111	000010	11111	11111	ERETAA, ERETAB — ERETAA	FEAT_PAuth
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB	FEAT_PAuth
0101	11111	000000	11111	00000	DRPS	-
0101	11111	00001x	11111	xxxx1	UNALLOCATED	-
011x	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-
100x	11111	00000x	xxxxx	xxxxx	UNALLOCATED	-
1000	11111	000010	xxxxx	xxxxx	BRAA, BRAAZ, BRAB, BRABZ — Key A, register modifier	FEAT_PAuth
1000	11111	000011	xxxxx	xxxxx	BRAA, BRAAZ, BRAB, BRABZ — Key B, register modifier	FEAT_PAuth
1001	11111	000010	xxxxx	xxxxx	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key A, register modifier	FEAT_PAuth
1001	11111	000011	xxxxx	xxxxx	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key B, register modifier	FEAT_PAuth
101x	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-
11xx	11111	0000xx	xxxxx	xxxxx	UNALLOCATED	-

Unconditional branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30				26	25	0
op	0	0	1	0	1	imm26	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	Instruction Details
0	B
1	BL

Compare and branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30					25	24	23 … 5	4 … 0
sf	0	1	1	0	1	0	op	imm19	Rt

sf	op	Instruction Details
0	0	CBZ — 32-bit
0	1	CBNZ — 32-bit
1	0	CBZ — 64-bit
1	1	CBNZ — 64-bit

Test and branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30					25	24	23 … 19	18 … 5	4 … 0
b5	0	1	1	0	1	1	op	b40	imm14	Rt

op	Instruction Details
0	TBZ
1	TBNZ

Compare registers and branch

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30						24	23 … 21	20 … 16	15	14	13 … 5	4 … 0
sf	1	1	1	0	1	0	0	cc	Rm	0	0	imm9	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	cc	Instruction Details	Condition
x	10x	UNALLOCATED	-
0	000	CB<cc> (register) — 32-bit greater than	FEAT_CMPBR
0	001	CB<cc> (register) — 32-bit greater than or equal	FEAT_CMPBR
0	010	CB<cc> (register) — 32-bit higher	FEAT_CMPBR
0	011	CB<cc> (register) — 32-bit higher or same	FEAT_CMPBR
0	110	CB<cc> (register) — 32-bit equal	FEAT_CMPBR
0	111	CB<cc> (register) — 32-bit not equal	FEAT_CMPBR
1	000	CB<cc> (register) — 64-bit greater than	FEAT_CMPBR
1	001	CB<cc> (register) — 64-bit greater than or equal	FEAT_CMPBR
1	010	CB<cc> (register) — 64-bit higher	FEAT_CMPBR
1	011	CB<cc> (register) — 64-bit higher or same	FEAT_CMPBR
1	110	CB<cc> (register) — 64-bit equal	FEAT_CMPBR
1	111	CB<cc> (register) — 64-bit not equal	FEAT_CMPBR

Compare register with immediate and branch

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

31	30						24	23	21	20	15	14	13	5	4	0
sf	1	1	1	0	1	0	1	cc		imm6		0	imm9		Rt	

sf	cc	Instruction Details	Condition
x	10x	UNALLOCATED	-
0	000	CB<cc> (immediate) — 32-bit greater than	FEAT_CMPBR
0	001	CB<cc> (immediate) — 32-bit less than	FEAT_CMPBR
0	010	CB<cc> (immediate) — 32-bit higher	FEAT_CMPBR
0	011	CB<cc> (immediate) — 32-bit lower	FEAT_CMPBR
0	110	CB<cc> (immediate) — 32-bit equal	FEAT_CMPBR
0	111	CB<cc> (immediate) — 32-bit not equal	FEAT_CMPBR
1	000	CB<cc> (immediate) — 64-bit greater than	FEAT_CMPBR
1	001	CB<cc> (immediate) — 64-bit less than	FEAT_CMPBR
1	010	CB<cc> (immediate) — 64-bit higher	FEAT_CMPBR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	cc	Instruction Details	Condition
1	011	CB<cc> (immediate) — 64-bit lower	FEAT_CMPBR
1	110	CB<cc> (immediate) — 64-bit equal	FEAT_CMPBR
1	111	CB<cc> (immediate) — 64-bit not equal	FEAT_CMPBR

Data Processing -- Register

Data Processing – Register

The encodings in this section are decoded from A64 instruction set encoding.

31	30	29	28	27	25	24	21	20	16	15	10	9	0
	op0		op1	1	0 1	op2				op3			

op0	op1	op2	op3	Instruction details
0	1	0110	xxxxxx	Data-processing (2 source)
1	1	0110	xxxxxx	Data-processing (1 source)
x	0	0xxx	xxxxxx	Logical (shifted register)
x	0	1xx0	xxxxxx	Add/subtract (shifted register)
x	0	1xx1	xxxxxx	Add/subtract (extended register)
x	1	0000	000000	Add/subtract (with carry)
x	1	0000	001xxx	Add/subtract (checked pointer)
x	1	0000	011xxx	UNALLOCATED
x	1	0000	100000	UNALLOCATED
x	1	0000	1x1xxx	UNALLOCATED
x	1	0000	x00001	Rotate right into flags
x	1	0000	x0010x	UNALLOCATED
x	1	0000	x10x0x	UNALLOCATED
x	1	0000	xx0010	Evaluate into flags
x	1	0000	xx0011	UNALLOCATED
x	1	0000	xx011x	UNALLOCATED
x	1	0010	xxxx0x	Conditional compare (register)
x	1	0010	xxxx1x	Conditional compare (immediate)
x	1	0100	xxxxxx	Conditional select
x	1	0xx1	xxxxxx	UNALLOCATED
x	1	1xxx	xxxxxx	Data-processing (3 source)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Data-processing (2 source)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	10	9	5	4	0
sf	0	S	1	1	0 1	0	1 1 0	Rm		opcode		Rn		Rd	

sf	S	opcode	Instruction Details	Condition
x	x	1xxxxx	UNALLOCATED	-
x	0	00011x	UNALLOCATED	-
x	1	00001x	UNALLOCATED	-
x	1	0001xx	UNALLOCATED	-
x	1	001xxx	UNALLOCATED	-
x	1	01xxxx	UNALLOCATED	-
0	x	00000x	UNALLOCATED	-
0	0	0x11xx	UNALLOCATED	-
0	0	000010	UDIV (quotient) — 32-bit	-
0	0	000011	SDIV (quotient) — 32-bit	-
0	0	00010x	UNALLOCATED	-
0	0	001000	LSLV — 32-bit	-
0	0	001001	LSRV — 32-bit	-
0	0	001010	ASRV — 32-bit	-
0	0	001011	RORV — 32-bit	-
0	0	010x11	UNALLOCATED	-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B	FEAT_CRC32
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H	FEAT_CRC32
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W	FEAT_CRC32
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB	FEAT_CRC32
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH	FEAT_CRC32
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW	FEAT_CRC32
0	0	011000	SMAX (register) — 32-bit	FEAT_CSSC
0	0	011001	UMAX (register) — 32-bit	FEAT_CSSC
0	0	011010	SMIN (register) — 32-bit	FEAT_CSSC
0	0	011011	UMIN (register) — 32-bit	FEAT_CSSC
1	x	000001	UNALLOCATED	-
1	0	000000	SUBP	FEAT_MTE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode	Instruction Details	Condition
1	0	000010	UDIV (quotient) — 64-bit	-
1	0	000011	SDIV (quotient) — 64-bit	-
1	0	000100	IRG	FEAT_MTE
1	0	000101	GMI	FEAT_MTE
1	0	001000	LSLV — 64-bit	-
1	0	001001	LSRV — 64-bit	-
1	0	001010	ASRV — 64-bit	-
1	0	001011	RORV — 64-bit	-
1	0	001100	PACGA	FEAT_PAuth
1	0	001101	UNALLOCATED	-
1	0	00111x	UNALLOCATED	-
1	0	010xx0	UNALLOCATED	-
1	0	010001	UNALLOCATED	-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X	FEAT_CRC32
1	0	010101	UNALLOCATED	-
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX	FEAT_CRC32
1	0	011000	SMAX (register) — 64-bit	FEAT_CSSC
1	0	011001	UMAX (register) — 64-bit	FEAT_CSSC
1	0	011010	SMIN (register) — 64-bit	FEAT_CSSC
1	0	011011	UMIN (register) — 64-bit	FEAT_CSSC
1	0	0111xx	UNALLOCATED	-
1	1	000000	SUBPS	FEAT_MTE

Data-processing (1 source)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27		25	24			21	20	16	15	10	9	5	4	0
sf	1	S	1	1	0	1	0	1	1	0	opcode2		opcode		Rn		Rd	

sf	S	opcode2	opcode	Rn	Rd	Instruction Details	Condition
x	0	00000	001001	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	00101x	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	0011xx	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	01xxxx	xxxxx	xxxxx	UNALLOCATED	-
x	0	00000	1xxxxx	xxxxx	xxxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode2	opcode	Rn	Rd	Instruction Details	Condition
x	0	!= 0000x	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
x	1	xxxxx	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
0	0	00000	000000	xxxxx	xxxxx	RBIT — 32-bit	-
0	0	00000	000001	xxxxx	xxxxx	REV16 — 32-bit	-
0	0	00000	000010	xxxxx	xxxxx	REV — 32-bit	-
0	0	00000	000011	xxxxx	xxxxx	UNALLOCATED	-
0	0	00000	000100	xxxxx	xxxxx	CLZ — 32-bit	-
0	0	00000	000101	xxxxx	xxxxx	CLS — 32-bit	-
0	0	00000	000110	xxxxx	xxxxx	CTZ — 32-bit	FEAT_CSSC
0	0	00000	000111	xxxxx	xxxxx	CNT — 32-bit	FEAT_CSSC
0	0	00000	001000	xxxxx	xxxxx	ABS — 32-bit	FEAT_CSSC
0	0	00001	xxxxxx	xxxxx	xxxxx	UNALLOCATED	-
1	0	00000	000000	xxxxx	xxxxx	RBIT — 64-bit	-
1	0	00000	000001	xxxxx	xxxxx	REV16 — 64-bit	-
1	0	00000	000010	xxxxx	xxxxx	REV32	-
1	0	00000	000011	xxxxx	xxxxx	REV — 64-bit	-
1	0	00000	000100	xxxxx	xxxxx	CLZ — 64-bit	-
1	0	00000	000101	xxxxx	xxxxx	CLS — 64-bit	-
1	0	00000	000110	xxxxx	xxxxx	CTZ — 64-bit	FEAT_CSSC
1	0	00000	000111	xxxxx	xxxxx	CNT — 64-bit	FEAT_CSSC
1	0	00000	001000	xxxxx	xxxxx	ABS — 64-bit	FEAT_CSSC
1	0	00001	000000	xxxxx	xxxxx	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	FEAT_PAuth
1	0	00001	000001	xxxxx	xxxxx	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	FEAT_PAuth
1	0	00001	000010	xxxxx	xxxxx	PACDA, PACDZA — PACDA	FEAT_PAuth
1	0	00001	000011	xxxxx	xxxxx	PACDB, PACDZB — PACDB	FEAT_PAuth
1	0	00001	000100	xxxxx	xxxxx	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	FEAT_PAuth
1	0	00001	000101	xxxxx	xxxxx	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	FEAT_PAuth
1	0	00001	000110	xxxxx	xxxxx	AUTDA, AUTDZA — AUTDA	FEAT_PAuth
1	0	00001	000111	xxxxx	xxxxx	AUTDB, AUTDZB — AUTDB	FEAT_PAuth
1	0	00001	001xxx	!= 11111	xxxxx	UNALLOCATED	-
1	0	00001	001000	11111	xxxxx	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	FEAT_PAuth
1	0	00001	001001	11111	xxxxx	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	FEAT_PAuth
1	0	00001	001010	11111	xxxxx	PACDA, PACDZA — PACDZA	FEAT_PAuth
1	0	00001	001011	11111	xxxxx	PACDB, PACDZB — PACDZB	FEAT_PAuth
1	0	00001	001100	11111	xxxxx	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA	FEAT_PAuth
1	0	00001	001101	11111	xxxxx	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB	FEAT_PAuth

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode2	opcode	Rn	Rd	Instruction Details	Condition
1	0	00001	001110	11111	xxxxx	AUTDA, AUTDZA — AUTDZA	FEAT_PAuth
1	0	00001	001111	11111	xxxxx	AUTDB, AUTDZB — AUTDZB	FEAT_PAuth
1	0	00001	01000x	!= 11111	xxxxx	UNALLOCATED	-
1	0	00001	010000	11111	xxxxx	XPACD, XPACI, XPACLRI — XPACI	FEAT_PAuth
1	0	00001	010001	11111	xxxxx	XPACD, XPACI, XPACLRI — XPACD	FEAT_PAuth
1	0	00001	01001x	xxxxx	xxxxx	UNALLOCATED	-
1	0	00001	0101xx	xxxxx	xxxxx	UNALLOCATED	-
1	0	00001	011xxx	xxxxx	xxxxx	UNALLOCATED	-
1	0	00001	10xxxx	xxxxx	!= 11110	UNALLOCATED	-
1	0	00001	1000xx	!= 11111	11110	UNALLOCATED	-
1	0	00001	100000	11111	11110	PACNBIASPPC	FEAT_PAuth_LR
1	0	00001	100001	11111	11110	PACNBIBSPPC	FEAT_PAuth_LR
1	0	00001	100010	11111	11110	PACIA171615	FEAT_PAuth_LR
1	0	00001	100011	11111	11110	PACIB171615	FEAT_PAuth_LR
1	0	00001	100100	xxxxx	11110	AUTIASPPCR	FEAT_PAuth_LR
1	0	00001	100101	xxxxx	11110	AUTIBSPPCR	FEAT_PAuth_LR
1	0	00001	10011x	xxxxx	11110	UNALLOCATED	-
1	0	00001	101xxx	!= 11111	11110	UNALLOCATED	-
1	0	00001	101000	11111	11110	PACIASPPC	FEAT_PAuth_LR
1	0	00001	101001	11111	11110	PACIBSPPC	FEAT_PAuth_LR
1	0	00001	10101x	11111	11110	UNALLOCATED	-
1	0	00001	10110x	11111	11110	UNALLOCATED	-
1	0	00001	101110	11111	11110	AUTIA171615	FEAT_PAuth_LR
1	0	00001	101111	11111	11110	AUTIB171615	FEAT_PAuth_LR
1	0	00001	11xxxx	xxxxx	xxxxx	UNALLOCATED	-

Logical (shifted register)

The encodings in this section are decoded from Data Processing – Register.

31	30 29	28	27 24	23 22	21	20 16	15 10	9 5	4 0
sf	opc	0	1 0 1 0	shift	N	Rm	imm6	Rn	Rd

sf	opc	N	Instruction Details
0	00	0	AND (shifted register) — 32-bit
0	00	1	BIC (shifted register) — 32-bit
0	01	0	ORR (shifted register) — 32-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	opc	N	Instruction Details
0	01	1	ORN (shifted register) — 32-bit
0	10	0	EOR (shifted register) — 32-bit
0	10	1	EON (shifted register) — 32-bit
0	11	0	ANDS (shifted register) — 32-bit
0	11	1	BICS (shifted register) — 32-bit
1	00	0	AND (shifted register) — 64-bit
1	00	1	BIC (shifted register) — 64-bit
1	01	0	ORR (shifted register) — 64-bit
1	01	1	ORN (shifted register) — 64-bit
1	10	0	EOR (shifted register) — 64-bit
1	10	1	EON (shifted register) — 64-bit
1	11	0	ANDS (shifted register) — 64-bit
1	11	1	BICS (shifted register) — 64-bit

Add/subtract (shifted register)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27			24	23	22	21	20	16	15	10	9	5	4	0
sf	op	S	0	1	0	1	1	shift		0	Rm		imm6		Rn		Rd	

sf	op	S	Instruction Details
0	0	0	ADD (shifted register) — 32-bit
0	0	1	ADDS (shifted register) — 32-bit
0	1	0	SUB (shifted register) — 32-bit
0	1	1	SUBS (shifted register) — 32-bit
1	0	0	ADD (shifted register) — 64-bit
1	0	1	ADDS (shifted register) — 64-bit
1	1	0	SUB (shifted register) — 64-bit
1	1	1	SUBS (shifted register) — 64-bit

Add/subtract (extended register)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27			24	23	22	21	20	16	15	13	12	10	9	5	4	0
sf	op	S	0	1	0	1	1	opt		1	Rm		option		imm3		Rn		Rd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	opt	Instruction Details
x	x	x	!=00	UNALLOCATED
0	0	0	00	ADD (extended register) — 32-bit
0	0	1	00	ADDS (extended register) — 32-bit
0	1	0	00	SUB (extended register) — 32-bit
0	1	1	00	SUBS (extended register) — 32-bit
1	0	0	00	ADD (extended register) — 64-bit
1	0	1	00	ADDS (extended register) — 64-bit
1	1	0	00	SUB (extended register) — 64-bit
1	1	1	00	SUBS (extended register) — 64-bit

Add/subtract (with carry)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	10	9	5	4	0
sf	op	S	1	1 0 1		0 0 0 0		Rm		0 0 0 0 0 0		Rn		Rd	

sf	op	S	Instruction Details
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

Add/subtract (checked pointer)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	13	12	10	9	5	4	0
sf	op	S	1	1 0 1		0 0 0 0		Rm		0 0 1		imm3		Rn		Rd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	Instruction Details	Condition
0	x	x	UNALLOCATED	-
1	x	1	UNALLOCATED	-
1	0	0	ADDPT	FEAT_CPA
1	1	0	SUBPT	FEAT_CPA

Rotate right into flags

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	15	14	10	9	5	4	3	0
sf	op	S	1	1 0 1		0 0 0 0		imm6		0 0 0 0 1		Rn		o2	mask	

sf	op	S	o2	Instruction Details	Condition
0	x	x	x	UNALLOCATED	-
1	0	0	x	UNALLOCATED	-
1	0	1	0	RMIF	FEAT_FlagM
1	0	1	1	UNALLOCATED	-
1	1	x	x	UNALLOCATED	-

Evaluate into flags

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	15	14	13	10	9	5	4	3	0
sf	op	S	1	1 0 1		0 0 0 0		opcode2		sz	0 0 1 0		Rn		o3	mask	

sf	op	S	opcode2	sz	o3	mask	Instruction Details	Condition
0	0	0	xxxxxx	x	x	xxxx	UNALLOCATED	-
0	0	1	000000	x	0	!= 1101	UNALLOCATED	-
0	0	1	000000	x	1	xxxx	UNALLOCATED	-
0	0	1	000000	0	0	1101	SETF8, SETF16 — SETF8	FEAT_FlagM
0	0	1	000000	1	0	1101	SETF8, SETF16 — SETF16	FEAT_FlagM

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	opcode2	sz	o3	mask	Instruction Details	Condition
0	0	1	!=000000	x	x	xxxx	UNALLOCATED	-
0	1	x	xxxxxx	x	x	xxxx	UNALLOCATED	-
1	x	x	xxxxxx	x	x	xxxx	UNALLOCATED	-

Conditional compare (register)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	12	11	10	9	5	4	3	0
sf	op	S	1	1	0 1	0	0 1 0	Rm		cond		0	o2	Rn		o3	nzcv	

sf	op	S	o2	o3	Instruction Details
x	x	0	x	x	UNALLOCATED
x	x	1	0	1	UNALLOCATED
x	x	1	1	x	UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

Conditional compare (immediate)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27	25	24	21	20	16	15	12	11	10	9	5	4	3	0
sf	op	S	1	1	0 1	0	0 1 0	imm5		cond		1	o2	Rn		o3	nzcv	

sf	op	S	o2	o3	Instruction Details
x	x	0	x	x	UNALLOCATED
x	x	1	0	1	UNALLOCATED
x	x	1	1	x	UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	o2	o3	Instruction Details
1	1	1	0	0	CCMP (immediate) — 64-bit

Conditional select

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27						21	20		16	15		12	11	10	9		5	4		0
sf	op	S	1	1	0	1	0	1	0	0	Rm			cond			op2		Rn			Rd		

sf	op	S	op2	Instruction Details
x	x	0	1x	UNALLOCATED
x	x	1	xx	UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

Data-processing (3 source)

The encodings in this section are decoded from Data Processing – Register.

31	30	29	28	27			24	23		21	20		16	15	14		10	9		5	4		0
sf	op54		1	1	0	1	1	op31			Rm			o0	Ra			Rn			Rd		

sf	op54	op31	o0	Instruction Details	Condition
x	00	100	x	UNALLOCATED	-
x	!= 00	xxx	x	UNALLOCATED	-
0	00	!= x00	x	UNALLOCATED	-
0	00	000	0	MADD — 32-bit	-
0	00	000	1	MSUB — 32-bit	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op54	op31	o0	Instruction Details	Condition
1	00	x10	1	UNALLOCATED	-
1	00	000	0	MADD — 64-bit	-
1	00	000	1	MSUB — 64-bit	-
1	00	001	0	SMADDL	-
1	00	001	1	SMSUBL	-
1	00	010	0	SMULH	-
1	00	011	0	MADDPT	FEAT_CPA
1	00	011	1	MSUBPT	FEAT_CPA
1	00	101	0	UMADDL	-
1	00	101	1	UMSUBL	-
1	00	110	0	UMULH	-
1	00	111	x	UNALLOCATED	-

Data Processing -- Scalar Floating-Point and Advanced SIMD

Data Processing – Scalar Floating-Point and Advanced SIMD

The encodings in this section are decoded from A64 instruction set encoding.

31 28	27 25	24 23	22 19	18 10	9 0
op0	1 1 1	op1	op2	op3	

op0	op1	op2	op3	Instruction details	Condition
00x0	0x	x101	00xxxxx10	UNALLOCATED	-
00x0	11	xxxx	xxxxxxxx1	UNALLOCATED	-
0100	0x	x101	00xxxxx10	Cryptographic AES	-
0101	0x	x0xx	xxx0xxx00	Cryptographic three-register SHA	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0101	0x	x101	00xxxxx10	Cryptographic two-register SHA	-
0111	0x	x0xx	xxxxxxxx0	UNALLOCATED	-
011x	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
01x1	0x	x0xx	xxx1xxxx1	Advanced SIMD scalar three same extra	-
01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	-
01x1	0x	x1xx	01xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxxx1	Advanced SIMD scalar three same	-
01x1	10	xxxx	xxxxxxxx1	Advanced SIMD scalar shift by immediate	-
01x1	1x	xxxx	xxxxxxxx0	Advanced SIMD scalar x indexed element	-
01xx	11	xxxx	xxxxxxxx1	UNALLOCATED	-
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	-
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	Advanced SIMD three-register extension	-
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	-
0xx0	0x	x1xx	01xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxxx1	Advanced SIMD modified immediate	-
0xx0	10	!= 0000	xxxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	1x	xxxx	xxxxxxxx0	Advanced SIMD vector x indexed element	-
10x0	xx	xxxx	xxxxxxxxx	UNALLOCATED	-
1100	00	0xxx	xxx1xxxxx	UNALLOCATED	-
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	-
1100	00	10xx	xxx11xxxx	UNALLOCATED	-
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	-
1100	00	11xx	xxx1x01xx	UNALLOCATED	-
1100	00	11xx	xxx1x1xxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Condition
1100	00	xxxx	xxx0xxxxx	Cryptographic four-register	-
1100	01	00xx	xxxxxxxxx	XAR	FEAT_SHA3
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	-
1100	01	1000	!= 0001000xx	UNALLOCATED	-
1100	01	1001	xxxxxxxxx	UNALLOCATED	-
1100	01	101x	xxxxxxxxx	UNALLOCATED	-
1100	01	x1xx	xxxxxxxxx	UNALLOCATED	-
1100	1x	xxxx	xxxxxxxxx	UNALLOCATED	-
1110	xx	xxxx	xxxxxxxxx	UNALLOCATED	-
11x1	xx	xxxx	xxxxxxxxx	UNALLOCATED	-
x0x1	0x	x0xx	xxxxxxxxx	Conversion between floating-point and fixed-point	-
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	-
x0x1	0x	x1xx	xxx100000	UNALLOCATED	-
x0x1	0x	x1xx	xxxx10000	Floating-point data-processing (1 source)	-
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	-
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	-
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	-
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	-
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	-
x0x1	1x	xxxx	xxxxxxxxx	Floating-point data-processing (3 source)	-

Cryptographic AES

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–24	23–22	21–17	16–12	11–10	9–5	4–0
0 1 0 0	1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

size	opcode	Instruction Details	Condition
00	00100	AESE	FEAT_AES
00	00101	AESD	FEAT_AES
00	00110	AESMC	FEAT_AES
00	00111	AESIMC	FEAT_AES
00	!= 001xx	UNALLOCATED	-
!= 00	xxxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cryptographic three-register SHA

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–24	23–22	21	20–16	15	14–12	11–10	9–5	4–0
0 1 0 1	1 1 1 0	size	0	Rm	0	opcode	0 0	Rn	Rd

size	opcode	Instruction Details	Condition
00	000	SHA1C	FEAT_SHA1
00	001	SHA1P	FEAT_SHA1
00	010	SHA1M	FEAT_SHA1
00	011	SHA1SU0	FEAT_SHA1
00	100	SHA256H	FEAT_SHA256
00	101	SHA256H2	FEAT_SHA256
00	110	SHA256SU1	FEAT_SHA256
00	111	UNALLOCATED	-
!=00	xxx	UNALLOCATED	-

Cryptographic two-register SHA

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–24	23–22	21–17	16–12	11–10	9–5	4–0
0 1 0 1	1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

size	opcode	Instruction Details	Condition
00	00000	SHA1H	FEAT_SHA1
00	00001	SHA1SU1	FEAT_SHA1
00	00010	SHA256SU0	FEAT_SHA256
00	00011	UNALLOCATED	-
00	001xx	UNALLOCATED	-
00	01xxx	UNALLOCATED	-
00	1xxxx	UNALLOCATED	-
!=00	xxxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Advanced SIMD scalar copy

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22	21	20	16	15	14	11	10	9	5	4	0
0	1	op	1	1	1	1	0	0	0	0	imm5		0	imm4		1	Rn		Rd	

op	imm4	Instruction Details	Condition
0	0000	DUP (element)	FEAT_AdvSIMD
0	!= 0000	UNALLOCATED	-
1	xxxx	UNALLOCATED	-

Advanced SIMD scalar three same FP16

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			24	23	22	21	20	16	15	14	13	11	10	9	5	4	0
0	1	U	1	1	1	1	0	a	1	0	Rm		0	0	opcode		1	Rn		Rd

U	a	opcode	Instruction Details	Condition
x	x	00x	UNALLOCATED	-
x	0	010	UNALLOCATED	-
0	x	101	UNALLOCATED	-
0	0	011	FMULX	FEAT_AdvSIMD && FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_AdvSIMD && FEAT_FP16
0	0	110	UNALLOCATED	-
0	0	111	FRECPS	FEAT_AdvSIMD && FEAT_FP16
0	1	x10	UNALLOCATED	-
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	111	FRSQRTS	FEAT_AdvSIMD && FEAT_FP16
1	x	011	UNALLOCATED	-
1	x	11x	UNALLOCATED	-
1	0	100	FCMGE (register)	FEAT_AdvSIMD && FEAT_FP16
1	0	101	FACGE	FEAT_AdvSIMD && FEAT_FP16
1	1	010	FABD	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Condition
1	1	100	FCMGT (register)	FEAT_AdvSIMD && FEAT_FP16
1	1	101	FACGT	FEAT_AdvSIMD && FEAT_FP16

Advanced SIMD scalar two-register miscellaneous FP16

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31 30	29	28 24	23	22 19	18 17	16 12	11 10	9 5	4 0
0 1	U	1 1 1 1 0	a	1 1 1 1	0 0	opcode	1 0	Rn	Rd

U	a	opcode	Instruction Details	Condition
x	x	x0xxx	UNALLOCATED	-
x	x	1100x	UNALLOCATED	-
x	0	01xxx	UNALLOCATED	-
x	1	010xx	UNALLOCATED	-
x	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	0	1111x	UNALLOCATED	-
0	1	01100	FCMGT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01111	UNALLOCATED	-
0	1	11010	FCVTPS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	11101	FRECPE	FEAT_AdvSIMD && FEAT_FP16
0	1	11110	UNALLOCATED	-
0	1	11111	FRECPX	FEAT_AdvSIMD && FEAT_FP16
1	x	1111x	UNALLOCATED	-
1	0	11010	FCVTNU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Condition
1	1	0111x	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	11101	FRSQRTE	FEAT_AdvSIMD && FEAT_FP16

Advanced SIMD scalar three same extra

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14			11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		0	Rm				1	opcode				1	Rn				Rd			

U	opcode	Instruction Details	Condition
0	xxxx	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM
1	!= 000x	UNALLOCATED	-

Advanced SIMD scalar two-register miscellaneous

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16				12	11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	0	0	0	0	opcode					1	0	Rn				Rd		

U	size	opcode	Instruction Details	Condition
x	xx	00xx0	UNALLOCATED	-
x	xx	00x01	UNALLOCATED	-
x	xx	1x000	UNALLOCATED	-
x	xx	10xx1	UNALLOCATED	-
x	xx	11001	UNALLOCATED	-
x	0x	01xxx	UNALLOCATED	-
x	0x	1111x	UNALLOCATED	-
x	1x	01111	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
x	1x	11100	UNALLOCATED	-
x	10	010x1	UNALLOCATED	-
x	10	01000	UNALLOCATED	-
0	xx	00011	SUQADD	FEAT_AdvSIMD
0	xx	00111	SQABS	FEAT_AdvSIMD
0	xx	10010	UNALLOCATED	-
0	xx	10100	SQXTN, SQXTN2	FEAT_AdvSIMD
0	xx	10110	UNALLOCATED	-
0	0x	11010	FCVTNS (vector)	FEAT_AdvSIMD
0	0x	11011	FCVTMS (vector)	FEAT_AdvSIMD
0	0x	11100	FCVTAS (vector)	FEAT_AdvSIMD
0	0x	11101	SCVTF (vector, integer)	FEAT_AdvSIMD
0	1x	01100	FCMGT (zero)	FEAT_AdvSIMD
0	1x	01101	FCMEQ (zero)	FEAT_AdvSIMD
0	1x	01110	FCMLT (zero)	FEAT_AdvSIMD
0	1x	11010	FCVTPS (vector)	FEAT_AdvSIMD
0	1x	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD
0	1x	11101	FRECPE	FEAT_AdvSIMD
0	1x	11110	UNALLOCATED	-
0	1x	11111	FRECPX	FEAT_AdvSIMD
0	10	01010	UNALLOCATED	-
0	11	01000	CMGT (zero)	FEAT_AdvSIMD
0	11	01001	CMEQ (zero)	FEAT_AdvSIMD
0	11	01010	CMLT (zero)	FEAT_AdvSIMD
0	11	01011	ABS	FEAT_AdvSIMD
1	xx	00011	USQADD	FEAT_AdvSIMD
1	xx	00111	SQNEG	FEAT_AdvSIMD
1	xx	10010	SQXTUN, SQXTUN2	FEAT_AdvSIMD
1	xx	10100	UQXTN, UQXTN2	FEAT_AdvSIMD
1	0x	11010	FCVTNU (vector)	FEAT_AdvSIMD
1	0x	11011	FCVTMU (vector)	FEAT_AdvSIMD
1	0x	11100	FCVTAU (vector)	FEAT_AdvSIMD
1	0x	11101	UCVTF (vector, integer)	FEAT_AdvSIMD
1	00	10110	UNALLOCATED	-
1	01	10110	FCVTXN, FCVTXN2	FEAT_AdvSIMD
1	1x	01x10	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	FEAT_AdvSIMD
1	1x	01101	FCMLE (zero)	FEAT_AdvSIMD
1	1x	1x110	UNALLOCATED	-
1	1x	11010	FCVTPU (vector)	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
1	1x	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD
1	1x	11101	FRSQRTE	FEAT_AdvSIMD
1	1x	11111	UNALLOCATED	-
1	11	01000	CMGE (zero)	FEAT_AdvSIMD
1	11	01001	CMLE (zero)	FEAT_AdvSIMD
1	11	01011	NEG (vector)	FEAT_AdvSIMD

Advanced SIMD scalar pairwise

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	1	0	0	0	opcode				1	0	Rn				Rd			

U	size	opcode	Instruction Details	Condition
x	xx	x0xxx	UNALLOCATED	-
x	xx	01110	UNALLOCATED	-
x	xx	111xx	UNALLOCATED	-
x	10	01101	UNALLOCATED	-
0	x1	011x1	UNALLOCATED	-
0	x1	01100	UNALLOCATED	-
0	00	01100	FMAXNMP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	01101	FADDP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	01111	FMAXP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	10	01100	FMINNMP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	10	01111	FMINP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	11	x100x	UNALLOCATED	-
0	11	x1010	UNALLOCATED	-
0	11	01011	UNALLOCATED	-
0	11	11011	ADDP (scalar)	FEAT_AdvSIMD
0	!=11	x10xx	UNALLOCATED	-
1	xx	x10xx	UNALLOCATED	-
1	0x	01100	FMAXNMP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	0x	01101	FADDP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	0x	01111	FMAXP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
1	1x	01100	FMINNMP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	1x	01111	FMINP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	11	01101	UNALLOCATED	-

Advanced SIMD scalar three different

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			12	11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	Rm					opcode				0	0	Rn				Rd			

U	opcode	Instruction Details	Condition
0	0xxx	UNALLOCATED	-
0	1xx0	UNALLOCATED	-
0	1001	SQDMLAL, SQDMLAL2 (vector)	FEAT_AdvSIMD
0	1011	SQDMLSL, SQDMLSL2 (vector)	FEAT_AdvSIMD
0	1101	SQDMULL, SQDMULL2 (vector)	FEAT_AdvSIMD
0	1111	UNALLOCATED	-
1	xxxx	UNALLOCATED	-

Advanced SIMD scalar three same

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			11	10	9			5	4			0
0	1	U	1	1	1	1	0	size		1	Rm					opcode				1	Rn				Rd			

U	size	opcode	Instruction Details	Condition
x	xx	x0011	UNALLOCATED	-
x	xx	011x1	UNALLOCATED	-
x	xx	11001	UNALLOCATED	-
x	xx	11110	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
x	x0	0x1x0	UNALLOCATED	-
x	x1	00100	UNALLOCATED	-
x	x1	011x0	UNALLOCATED	-
x	0x	xx0x0	UNALLOCATED	-
x	0x	10x01	UNALLOCATED	-
x	01	00110	UNALLOCATED	-
x	1x	10010	UNALLOCATED	-
x	1x	11000	UNALLOCATED	-
x	10	0x0x0	UNALLOCATED	-
x	10	10x0x	UNALLOCATED	-
x	11	00000	UNALLOCATED	-
x	11	00010	UNALLOCATED	-
x	11	10101	UNALLOCATED	-
x	!=10	10100	UNALLOCATED	-
x	!=11	00111	UNALLOCATED	-
0	xx	00001	SQADD	FEAT_AdvSIMD
0	xx	00101	SQSUB	FEAT_AdvSIMD
0	xx	01001	SQSHL (register)	FEAT_AdvSIMD
0	xx	01011	SQRSHL	FEAT_AdvSIMD
0	xx	10110	SQDMULH (vector)	FEAT_AdvSIMD
0	xx	10111	UNALLOCATED	-
0	xx	11101	UNALLOCATED	-
0	0x	11011	FMULX	FEAT_AdvSIMD
0	0x	11100	FCMEQ (register)	FEAT_AdvSIMD
0	0x	11111	FRECPS	FEAT_AdvSIMD
0	1x	11010	UNALLOCATED	-
0	1x	11011	UNALLOCATED	-
0	1x	11100	UNALLOCATED	-
0	1x	11111	FRSQRTS	FEAT_AdvSIMD
0	11	00110	CMGT (register)	FEAT_AdvSIMD
0	11	00111	CMGE (register)	FEAT_AdvSIMD
0	11	01000	SSHL	FEAT_AdvSIMD
0	11	01010	SRSHL	FEAT_AdvSIMD
0	11	10000	ADD (vector)	FEAT_AdvSIMD
0	11	10001	CMTST	FEAT_AdvSIMD
1	xx	00001	UQADD	FEAT_AdvSIMD
1	xx	00101	UQSUB	FEAT_AdvSIMD
1	xx	01001	UQSHL (register)	FEAT_AdvSIMD
1	xx	01011	UQRSHL	FEAT_AdvSIMD
1	xx	1x111	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
1	xx	10110	SQRDMULH (vector)	FEAT_AdvSIMD
1	xx	11011	UNALLOCATED	-
1	0x	11100	FCMGE (register)	FEAT_AdvSIMD
1	0x	11101	FACGE	FEAT_AdvSIMD
1	1x	11010	FABD	FEAT_AdvSIMD
1	1x	11100	FCMGT (register)	FEAT_AdvSIMD
1	1x	11101	FACGT	FEAT_AdvSIMD
1	11	00110	CMHI (register)	FEAT_AdvSIMD
1	11	00111	CMHS (register)	FEAT_AdvSIMD
1	11	01000	USHL	FEAT_AdvSIMD
1	11	01010	URSHL	FEAT_AdvSIMD
1	11	10000	SUB (vector)	FEAT_AdvSIMD
1	11	10001	CMEQ (register)	FEAT_AdvSIMD

Advanced SIMD scalar shift by immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22	19	18	16	15	11	10	9	5	4	0
0	1	U	1	1	1	1	1	0	immh		immb		opcode		1	Rn		Rd	

U	immh	opcode	Instruction Details	Condition
x	xxxx	0xxx1	UNALLOCATED	-
x	xxxx	101xx	UNALLOCATED	-
x	xxxx	11101	UNALLOCATED	-
x	0xxx	x10x0	UNALLOCATED	-
x	0xxx	00xx0	UNALLOCATED	-
x	0xxx	110x1	UNALLOCATED	-
x	0000	x1110	UNALLOCATED	-
x	0000	1001x	UNALLOCATED	-
x	0000	11100	UNALLOCATED	-
x	0000	11111	UNALLOCATED	-
x	1xxx	110xx	UNALLOCATED	-
x	!=0000	11110	UNALLOCATED	-
0	xxxx	01100	UNALLOCATED	-
0	xxxx	1000x	UNALLOCATED	-
0	1xxx	00000	SSHR	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	immh	opcode	Instruction Details	Condition
0	1xxx	00010	SSRA	FEAT_AdvSIMD
0	1xxx	00100	SRSHR	FEAT_AdvSIMD
0	1xxx	00110	SRSRA	FEAT_AdvSIMD
0	1xxx	01000	UNALLOCATED	-
0	1xxx	01010	SHL	FEAT_AdvSIMD
0	!= 0000	01110	SQSHL (immediate)	FEAT_AdvSIMD
0	!= 0000	10010	SQSHRN, SQSHRN2	FEAT_AdvSIMD
0	!= 0000	10011	SQRSHRN, SQRSHRN2	FEAT_AdvSIMD
0	!= 0000	11100	SCVTF (vector, fixed-point)	FEAT_AdvSIMD
0	!= 0000	11111	FCVTZS (vector, fixed-point)	FEAT_AdvSIMD
1	0000	01100	UNALLOCATED	-
1	0000	1000x	UNALLOCATED	-
1	1xxx	00000	USHR	FEAT_AdvSIMD
1	1xxx	00010	USRA	FEAT_AdvSIMD
1	1xxx	00100	URSHR	FEAT_AdvSIMD
1	1xxx	00110	URSRA	FEAT_AdvSIMD
1	1xxx	01000	SRI	FEAT_AdvSIMD
1	1xxx	01010	SLI	FEAT_AdvSIMD
1	!= 0000	01100	SQSHLU	FEAT_AdvSIMD
1	!= 0000	01110	UQSHL (immediate)	FEAT_AdvSIMD
1	!= 0000	10000	SQSHRUN, SQSHRUN2	FEAT_AdvSIMD
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2	FEAT_AdvSIMD
1	!= 0000	10010	UQSHRN, UQSHRN2	FEAT_AdvSIMD
1	!= 0000	10011	UQRSHRN, UQRSHRN2	FEAT_AdvSIMD
1	!= 0000	11100	UCVTF (vector, fixed-point)	FEAT_AdvSIMD
1	!= 0000	11111	FCVTZU (vector, fixed-point)	FEAT_AdvSIMD

Advanced SIMD scalar x indexed element

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	19	16	15	12	11	10	9	5	4	0
0	1	U	1	1	1	1	1	size		L	M	Rm		opcode		H	0	Rn		Rd	

U	size	opcode	Instruction Details	Condition
x	01	1001	UNALLOCATED	-
0	xx	0xx0	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
0	xx	0011	SQDMLAL, SQDMLAL2 (by element)	FEAT_AdvSIMD
0	xx	0111	SQDMLSL, SQDMLSL2 (by element)	FEAT_AdvSIMD
0	xx	10x0	UNALLOCATED	-
0	xx	1011	SQDMULL, SQDMULL2 (by element)	FEAT_AdvSIMD
0	xx	1100	SQDMULH (by element)	FEAT_AdvSIMD
0	xx	1101	SQRDMULH (by element)	FEAT_AdvSIMD
0	xx	111x	UNALLOCATED	-
0	00	0001	FMLA (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	0101	FMLS (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	1001	FMUL (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	01	0x01	UNALLOCATED	-
0	1x	0001	FMLA (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
0	1x	0101	FMLS (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
0	1x	1001	FMUL (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
1	xx	0xxx	UNALLOCATED	-
1	xx	1xx0	UNALLOCATED	-
1	xx	1011	UNALLOCATED	-
1	xx	1101	SQRDMLAH (by element)	FEAT_RDM
1	xx	1111	SQRDMLSH (by element)	FEAT_RDM
1	00	1001	FMULX (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
1	1x	1001	FMULX (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD

Advanced SIMD table lookup

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23 22	21	20 16	15	14 13	12	11 10	9 5	4 0
0	Q	0	0	1	1	1	0	op2	0	Rm	0	len	op	0 0	Rn	Rd

Q	op2	len	op	Instruction Details	Condition
x	00	00	0	TBL — Single register table	FEAT_AdvSIMD
x	00	00	1	TBX — Single register table	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	op2	len	op	Instruction Details	Condition
x	00	01	0	TBL — Two register table	FEAT_AdvSIMD
x	00	01	1	TBX — Two register table	FEAT_AdvSIMD
x	00	10	0	TBL — Three register table	FEAT_AdvSIMD
x	00	10	1	TBX — Three register table	FEAT_AdvSIMD
x	00	11	0	TBL — Four register table	FEAT_AdvSIMD
x	00	11	1	TBX — Four register table	FEAT_AdvSIMD
0	!= 00	xx	x	UNALLOCATED	-
1	01	xx	1	LUTI4 — Halfword	FEAT_AdvSIMD && FEAT_LUT
1	01	x0	0	UNALLOCATED	-
1	01	x1	0	LUTI4 — Byte	FEAT_AdvSIMD && FEAT_LUT
1	10	xx	0	UNALLOCATED	-
1	10	xx	1	LUTI2 — Byte	FEAT_AdvSIMD && FEAT_LUT
1	11	xx	x	LUTI2 — Halfword	FEAT_AdvSIMD && FEAT_LUT

Advanced SIMD permute

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23	22	21	20				16	15	14			12	11	10	9				5	4				0
0	Q	0	0	1	1	1	0	size		0	Rm					0	opcode				1	0	Rn						Rd			

opcode	Instruction Details	Condition
x00	UNALLOCATED	-
001	UZP1	FEAT_AdvSIMD
010	TRN1	FEAT_AdvSIMD
011	ZIP1	FEAT_AdvSIMD
101	UZP2	FEAT_AdvSIMD
110	TRN2	FEAT_AdvSIMD
111	ZIP2	FEAT_AdvSIMD

Advanced SIMD extract

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23	22	21	20				16	15	14				11	10	9				5	4				0
0	Q	1	0	1	1	1	0	op2		0	Rm					0	imm4					0	Rn						Rd			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op2	Instruction Details	Condition
00	EXT	FEAT_AdvSIMD
!= 00	UNALLOCATED	-

Advanced SIMD copy

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22	21	20	16	15	14	11	10	9	5	4	0
0	Q	op	0	1	1	1	0	0	0	0	imm5		0	imm4		1	Rn		Rd	

Q	op	imm5	imm4	Instruction Details	Condition
x	0	xxxxx	0000	DUP (element)	FEAT_AdvSIMD
x	0	xxxxx	0001	DUP (general)	FEAT_AdvSIMD
x	0	xxxxx	01x0	UNALLOCATED	-
x	0	xxxxx	1xxx	UNALLOCATED	-
0	0	xxxxx	001x	UNALLOCATED	-
0	0	xxxxx	0101	SMOV — 32-bit	FEAT_AdvSIMD
0	0	xxxxx	0111	UMOV — 32-bit	FEAT_AdvSIMD
0	1	xxxxx	xxxx	UNALLOCATED	-
1	0	xxxxx	0010	UNALLOCATED	-
1	0	xxxxx	0011	INS (general)	FEAT_AdvSIMD
1	0	xxxxx	0101	SMOV — 64-bit	FEAT_AdvSIMD
1	0	x1000	0111	UMOV — 64-bit	FEAT_AdvSIMD
1	0	!= x1000	0111	UNALLOCATED	-
1	1	xxxxx	xxxx	INS (element)	FEAT_AdvSIMD

Advanced SIMD three same (FP16)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	16	15	14	13	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	a	1	0	Rm		0	0	opcode		1	Rn		Rd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Condition
0	0	000	FMAXNM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	001	FMLA (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	010	FADD (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	011	FMULX	FEAT_AdvSIMD && FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_AdvSIMD && FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	111	FRECPS	FEAT_AdvSIMD && FEAT_FP16
0	1	000	FMINNM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	001	FMLS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	010	FSUB (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	011	FAMAX	FEAT_AdvSIMD && FEAT_FAMINMAX
0	1	10x	UNALLOCATED	-
0	1	110	FMIN (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	111	FRSQRTS	FEAT_AdvSIMD && FEAT_FP16
1	x	001	UNALLOCATED	-
1	0	000	FMAXNMP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	010	FADDP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	011	FMUL (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	100	FCMGE (register)	FEAT_AdvSIMD && FEAT_FP16
1	0	101	FACGE	FEAT_AdvSIMD && FEAT_FP16
1	0	110	FMAXP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	111	FDIV (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	000	FMINNMP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	010	FABD	FEAT_AdvSIMD && FEAT_FP16
1	1	011	FAMIN	FEAT_AdvSIMD && FEAT_FAMINMAX
1	1	100	FCMGT (register)	FEAT_AdvSIMD && FEAT_FP16
1	1	101	FACGT	FEAT_AdvSIMD && FEAT_FP16
1	1	110	FMINP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	111	FSCALE	FEAT_FP8

Advanced SIMD two-register miscellaneous (FP16)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22			19	18	17	16	12	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	a	1	1	1	1	0	0	opcode		1	0	Rn		Rd	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Condition
x	x	x0xxx	UNALLOCATED	-
x	0	01xxx	UNALLOCATED	-
x	0	1111x	UNALLOCATED	-
x	1	010xx	UNALLOCATED	-
x	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11001	FRINTM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11010	FCVTNS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01111	FABS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11000	FRINTP (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11001	FRINTZ (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	11101	FRECPE	FEAT_AdvSIMD && FEAT_FP16
0	1	1111x	UNALLOCATED	-
1	0	11000	FRINTA (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11001	FRINTX (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	x1110	UNALLOCATED	-
1	1	01100	FCMGE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01111	FNEG (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11000	UNALLOCATED	-
1	1	11001	FRINTI (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11010	FCVTPU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	11101	FRSQRTE	FEAT_AdvSIMD && FEAT_FP16
1	1	11111	FSQRT (vector)	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Advanced SIMD three-register extension

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14			11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		0	Rm				1	opcode				1	Rn				Rd			

Q	U	size	opcode	Instruction Details	Condition
x	0	xx	0000	UNALLOCATED	-
x	0	xx	0010	SDOT (vector)	FEAT_DotProd
x	0	0x	1010	UNALLOCATED	-
x	0	0x	1100	UNALLOCATED	-
x	0	00	1110	FCVTN, FCVTN2 (single-precision to 8-bit floating-point)	FEAT_FP8
x	0	00	1111	FDOT (8-bit floating-point to single-precision, vector)	FEAT_FP8DOT4
x	0	01	1110	FCVTN (half-precision to 8-bit floating-point)	FEAT_FP8
x	0	01	1111	FDOT (8-bit floating-point to half-precision, vector)	FEAT_FP8DOT2
x	0	10	0001	UNALLOCATED	-
x	0	10	0011	USDOT (vector)	FEAT_I8MM
x	0	10	1x11	UNALLOCATED	-
x	0	10	1001	UNALLOCATED	-
x	0	!=0x	1xx0	UNALLOCATED	-
x	0	!=10	x0x1	UNALLOCATED	-
x	1	xx	0000	SQRDMLAH (vector)	FEAT_RDM
x	1	xx	0001	SQRDMLSH (vector)	FEAT_RDM
x	1	xx	0010	UDOT (vector)	FEAT_DotProd
x	1	xx	0011	UNALLOCATED	-
x	1	xx	10xx	FCMLA	FEAT_FCMA
x	1	xx	11x0	FCADD	FEAT_FCMA
x	1	x0	1111	UNALLOCATED	-
x	1	01	1111	BFDOT (vector)	FEAT_BF16
x	1	11	1111	BFMLALB, BFMLALT (vector)	FEAT_BF16
0	x	xx	01xx	UNALLOCATED	-
0	x	xx	1101	UNALLOCATED	-
0	0	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLBB	FEAT_FP8FMA
0	0	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLBT	FEAT_FP8FMA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Condition
0	0	11	1111	FMLALB, FMLALT (vector) — FMLALB	FEAT_FP8FMA
1	x	10	011x	UNALLOCATED	-
1	x	!=10	01xx	UNALLOCATED	-
1	0	0x	1101	UNALLOCATED	-
1	0	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLTB	FEAT_FP8FMA
1	0	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector) — FMLALLTT	FEAT_FP8FMA
1	0	10	0100	SMMLA (vector)	FEAT_I8MM
1	0	10	0101	USMMLA (vector)	FEAT_I8MM
1	0	10	1101	UNALLOCATED	-
1	0	11	1101	UNALLOCATED	-
1	0	11	1111	FMLALB, FMLALT (vector) — FMLALT	FEAT_FP8FMA
1	1	00	1101	FMMLA (widening, 8-bit floating-point to half-precision)	FEAT_F8F16MM
1	1	01	1101	BFMMLA (widening)	FEAT_BF16
1	1	10	0100	UMMLA (vector)	FEAT_I8MM
1	1	10	0101	UNALLOCATED	-
1	1	10	1101	FMMLA (widening, 8-bit floating-point to single-precision)	FEAT_F8F32MM
1	1	11	1101	UNALLOCATED	-

Advanced SIMD two-register miscellaneous

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		1	0	0	0	0	opcode				1	0	Rn				Rd			

U	size	opcode	Instruction Details	Condition
x	xx	1000x	UNALLOCATED	-
x	xx	10101	UNALLOCATED	-
x	0x	011xx	UNALLOCATED	-
x	10	11110	UNALLOCATED	-
x	11	1x110	UNALLOCATED	-
0	xx	00000	REV64	FEAT_AdvSIMD
0	xx	00001	REV16 (vector)	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
0	xx	00010	SADDLP	FEAT_AdvSIMD
0	xx	00011	SUQADD	FEAT_AdvSIMD
0	xx	00100	CLS (vector)	FEAT_AdvSIMD
0	xx	00101	CNT	FEAT_AdvSIMD
0	xx	00110	SADALP	FEAT_AdvSIMD
0	xx	00111	SQABS	FEAT_AdvSIMD
0	xx	01000	CMGT (zero)	FEAT_AdvSIMD
0	xx	01001	CMEQ (zero)	FEAT_AdvSIMD
0	xx	01010	CMLT (zero)	FEAT_AdvSIMD
0	xx	01011	ABS	FEAT_AdvSIMD
0	xx	10010	XTN, XTN2	FEAT_AdvSIMD
0	xx	10011	UNALLOCATED	-
0	xx	10100	SQXTN, SQXTN2	FEAT_AdvSIMD
0	0x	10110	FCVTN, FCVTN2 (double to single-precision, single to half-precision)	FEAT_AdvSIMD
0	0x	10111	FCVTL, FCVTL2	FEAT_AdvSIMD
0	0x	11000	FRINTN (vector)	FEAT_AdvSIMD
0	0x	11001	FRINTM (vector)	FEAT_AdvSIMD
0	0x	11010	FCVTNS (vector)	FEAT_AdvSIMD
0	0x	11011	FCVTMS (vector)	FEAT_AdvSIMD
0	0x	11100	FCVTAS (vector)	FEAT_AdvSIMD
0	0x	11101	SCVTF (vector, integer)	FEAT_AdvSIMD
0	0x	11110	FRINT32Z (vector)	FEAT_FRINTTS
0	0x	11111	FRINT64Z (vector)	FEAT_FRINTTS
0	1x	01100	FCMGT (zero)	FEAT_AdvSIMD
0	1x	01101	FCMEQ (zero)	FEAT_AdvSIMD
0	1x	01110	FCMLT (zero)	FEAT_AdvSIMD
0	1x	01111	FABS (vector)	FEAT_AdvSIMD
0	1x	1x111	UNALLOCATED	-
0	1x	11000	FRINTP (vector)	FEAT_AdvSIMD
0	1x	11001	FRINTZ (vector)	FEAT_AdvSIMD
0	1x	11010	FCVTPS (vector)	FEAT_AdvSIMD
0	1x	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD
0	1x	11100	URECPE	FEAT_AdvSIMD
0	1x	11101	FRECPE	FEAT_AdvSIMD
0	10	10110	BFCVTN, BFCVTN2	FEAT_BF16
1	xx	00000	REV32 (vector)	FEAT_AdvSIMD
1	xx	00010	UADDLP	FEAT_AdvSIMD
1	xx	00011	USQADD	FEAT_AdvSIMD
1	xx	00100	CLZ (vector)	FEAT_AdvSIMD
1	xx	00110	UADALP	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
1	xx	00111	SQNEG	FEAT_AdvSIMD
1	xx	01000	CMGE (zero)	FEAT_AdvSIMD
1	xx	01001	CMLE (zero)	FEAT_AdvSIMD
1	xx	01010	UNALLOCATED	-
1	xx	01011	NEG (vector)	FEAT_AdvSIMD
1	xx	10010	SQXTUN, SQXTUN2	FEAT_AdvSIMD
1	xx	10011	SHLL, SHLL2	FEAT_AdvSIMD
1	xx	10100	UQXTN, UQXTN2	FEAT_AdvSIMD
1	x0	10110	UNALLOCATED	-
1	0x	00001	UNALLOCATED	-
1	0x	11000	FRINTA (vector)	FEAT_AdvSIMD
1	0x	11001	FRINTX (vector)	FEAT_AdvSIMD
1	0x	11010	FCVTNU (vector)	FEAT_AdvSIMD
1	0x	11011	FCVTMU (vector)	FEAT_AdvSIMD
1	0x	11100	FCVTAU (vector)	FEAT_AdvSIMD
1	0x	11101	UCVTF (vector, integer)	FEAT_AdvSIMD
1	0x	11110	FRINT32X (vector)	FEAT_FRINTTS
1	0x	11111	FRINT64X (vector)	FEAT_FRINTTS
1	00	00101	NOT	FEAT_AdvSIMD
1	00	10111	F1CVTL, F1CVTL2, F2CVTL, F2CVTL2 — F1CVTL{2}	FEAT_FP8
1	01	00101	RBIT (vector)	FEAT_AdvSIMD
1	01	10110	FCVTXN, FCVTXN2	FEAT_AdvSIMD
1	01	10111	F1CVTL, F1CVTL2, F2CVTL, F2CVTL2 — F2CVTL{2}	FEAT_FP8
1	1x	00x01	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	FEAT_AdvSIMD
1	1x	01101	FCMLE (zero)	FEAT_AdvSIMD
1	1x	01110	UNALLOCATED	-
1	1x	01111	FNEG (vector)	FEAT_AdvSIMD
1	1x	11000	UNALLOCATED	-
1	1x	11001	FRINTI (vector)	FEAT_AdvSIMD
1	1x	11010	FCVTPU (vector)	FEAT_AdvSIMD
1	1x	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD
1	1x	11100	URSQRTE	FEAT_AdvSIMD
1	1x	11101	FRSQRTE	FEAT_AdvSIMD
1	1x	11111	FSQRT (vector)	FEAT_AdvSIMD
1	10	10111	BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2 — BF1CVTL{2}	FEAT_FP8
1	11	10111	BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2 — BF2CVTL{2}	FEAT_FP8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Advanced SIMD across lanes

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21				17	16			12	11	10	9				5	4				0
0	Q	U	0	1	1	1	0	size		1	1	0	0	0	opcode				1	0	Rn					Rd				

Q	U	size	opcode	Instruction Details	Condition
x	x	xx	x100x	UNALLOCATED	-
x	x	xx	000x0	UNALLOCATED	-
x	x	xx	00001	UNALLOCATED	-
x	x	xx	01011	UNALLOCATED	-
x	x	xx	100xx	UNALLOCATED	-
x	x	x0	001xx	UNALLOCATED	-
x	x	x0	01101	UNALLOCATED	-
x	x	x0	101xx	UNALLOCATED	-
x	x	x0	111xx	UNALLOCATED	-
x	x	x1	xx1xx	UNALLOCATED	-
x	0	xx	00011	SADDLV	FEAT_AdvSIMD
x	0	xx	01010	SMAXV	FEAT_AdvSIMD
x	0	xx	11010	SMINV	FEAT_AdvSIMD
x	0	xx	11011	ADDV	FEAT_AdvSIMD
x	0	x0	01110	UNALLOCATED	-
x	0	00	01100	FMAXNMV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	01111	FMAXV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	10	01100	FMINNMV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	10	01111	FMINV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	1	xx	00011	UADDLV	FEAT_AdvSIMD
x	1	xx	01010	UMAXV	FEAT_AdvSIMD
x	1	xx	11010	UMINV	FEAT_AdvSIMD
x	1	xx	11011	UNALLOCATED	-
0	1	x0	011x0	UNALLOCATED	-
0	1	x0	01111	UNALLOCATED	-
1	1	x0	01110	UNALLOCATED	-
1	1	00	01100	FMAXNMV — Single-precision	FEAT_AdvSIMD
1	1	00	01111	FMAXV — Single-precision	FEAT_AdvSIMD
1	1	10	01100	FMINNMV — Single-precision	FEAT_AdvSIMD
1	1	10	01111	FMINV — Single-precision	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Advanced SIMD three different

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15			12	11	10	9			5	4			0
0	Q	U	0	1	1	1	0	size		1	Rm				opcode				0	0	Rn				Rd			

U	opcode	Instruction Details	Condition
0	0000	SADDL, SADDL2	FEAT_AdvSIMD
0	0001	SADDW, SADDW2	FEAT_AdvSIMD
0	0010	SSUBL, SSUBL2	FEAT_AdvSIMD
0	0011	SSUBW, SSUBW2	FEAT_AdvSIMD
0	0100	ADDHN, ADDHN2	FEAT_AdvSIMD
0	0101	SABAL, SABAL2	FEAT_AdvSIMD
0	0110	SUBHN, SUBHN2	FEAT_AdvSIMD
0	0111	SABDL, SABDL2	FEAT_AdvSIMD
0	1000	SMLAL, SMLAL2 (vector)	FEAT_AdvSIMD
0	1001	SQDMLAL, SQDMLAL2 (vector)	FEAT_AdvSIMD
0	1010	SMLSL, SMLSL2 (vector)	FEAT_AdvSIMD
0	1011	SQDMLSL, SQDMLSL2 (vector)	FEAT_AdvSIMD
0	1100	SMULL, SMULL2 (vector)	FEAT_AdvSIMD
0	1101	SQDMULL, SQDMULL2 (vector)	FEAT_AdvSIMD
0	1110	PMULL, PMULL2	FEAT_AdvSIMD
0	1111	UNALLOCATED	-
1	0000	UADDL, UADDL2	FEAT_AdvSIMD
1	0001	UADDW, UADDW2	FEAT_AdvSIMD
1	0010	USUBL, USUBL2	FEAT_AdvSIMD
1	0011	USUBW, USUBW2	FEAT_AdvSIMD
1	0100	RADDHN, RADDHN2	FEAT_AdvSIMD
1	0101	UABAL, UABAL2	FEAT_AdvSIMD
1	0110	RSUBHN, RSUBHN2	FEAT_AdvSIMD
1	0111	UABDL, UABDL2	FEAT_AdvSIMD
1	1xx1	UNALLOCATED	-
1	1000	UMLAL, UMLAL2 (vector)	FEAT_AdvSIMD
1	1010	UMLSL, UMLSL2 (vector)	FEAT_AdvSIMD
1	1100	UMULL, UMULL2 (vector)	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	opcode	Instruction Details	Condition
1	1110	UNALLOCATED	-

Advanced SIMD three same

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15				11	10	9				5	4				0
0	Q	U	0	1	1	1	0	size		1	Rm					opcode					1	Rn					Rd				

U	size	opcode	Instruction Details	Condition
0	xx	00000	SHADD	FEAT_AdvSIMD
0	xx	00001	SQADD	FEAT_AdvSIMD
0	xx	00010	SRHADD	FEAT_AdvSIMD
0	xx	00100	SHSUB	FEAT_AdvSIMD
0	xx	00101	SQSUB	FEAT_AdvSIMD
0	xx	00110	CMGT (register)	FEAT_AdvSIMD
0	xx	00111	CMGE (register)	FEAT_AdvSIMD
0	xx	01000	SSHL	FEAT_AdvSIMD
0	xx	01001	SQSHL (register)	FEAT_AdvSIMD
0	xx	01010	SRSHL	FEAT_AdvSIMD
0	xx	01011	SQRSHL	FEAT_AdvSIMD
0	xx	01100	SMAX	FEAT_AdvSIMD
0	xx	01101	SMIN	FEAT_AdvSIMD
0	xx	01110	SABD	FEAT_AdvSIMD
0	xx	01111	SABA	FEAT_AdvSIMD
0	xx	10000	ADD (vector)	FEAT_AdvSIMD
0	xx	10001	CMTST	FEAT_AdvSIMD
0	xx	10010	MLA (vector)	FEAT_AdvSIMD
0	xx	10011	MUL (vector)	FEAT_AdvSIMD
0	xx	10100	SMAXP	FEAT_AdvSIMD
0	xx	10101	SMINP	FEAT_AdvSIMD
0	xx	10110	SQDMULH (vector)	FEAT_AdvSIMD
0	xx	10111	ADDP (vector)	FEAT_AdvSIMD
0	x1	11101	UNALLOCATED	-
0	0x	11000	FMAXNM (vector)	FEAT_AdvSIMD
0	0x	11001	FMLA (vector)	FEAT_AdvSIMD
0	0x	11010	FADD (vector)	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
0	0x	11011	FMULX	FEAT_AdvSIMD
0	0x	11100	FCMEQ (register)	FEAT_AdvSIMD
0	0x	11110	FMAX (vector)	FEAT_AdvSIMD
0	0x	11111	FRECPS	FEAT_AdvSIMD
0	00	00011	AND (vector)	FEAT_AdvSIMD
0	00	11101	FMLAL, FMLAL2 (vector) — FMLAL	FEAT_FHM
0	01	00011	BIC (vector, register)	FEAT_AdvSIMD
0	1x	11000	FMINNM (vector)	FEAT_AdvSIMD
0	1x	11001	FMLS (vector)	FEAT_AdvSIMD
0	1x	11010	FSUB (vector)	FEAT_AdvSIMD
0	1x	11011	FAMAX	FEAT_AdvSIMD && FEAT_FAMINMAX
0	1x	11100	UNALLOCATED	-
0	1x	11110	FMIN (vector)	FEAT_AdvSIMD
0	1x	11111	FRSQRTS	FEAT_AdvSIMD
0	10	00011	ORR (vector, register)	FEAT_AdvSIMD
0	10	11101	FMLSL, FMLSL2 (vector) — FMLSL	FEAT_FHM
0	11	00011	ORN (vector)	FEAT_AdvSIMD
1	xx	00000	UHADD	FEAT_AdvSIMD
1	xx	00001	UQADD	FEAT_AdvSIMD
1	xx	00010	URHADD	FEAT_AdvSIMD
1	xx	00100	UHSUB	FEAT_AdvSIMD
1	xx	00101	UQSUB	FEAT_AdvSIMD
1	xx	00110	CMHI (register)	FEAT_AdvSIMD
1	xx	00111	CMHS (register)	FEAT_AdvSIMD
1	xx	01000	USHL	FEAT_AdvSIMD
1	xx	01001	UQSHL (register)	FEAT_AdvSIMD
1	xx	01010	URSHL	FEAT_AdvSIMD
1	xx	01011	UQRSHL	FEAT_AdvSIMD
1	xx	01100	UMAX	FEAT_AdvSIMD
1	xx	01101	UMIN	FEAT_AdvSIMD
1	xx	01110	UABD	FEAT_AdvSIMD
1	xx	01111	UABA	FEAT_AdvSIMD
1	xx	10000	SUB (vector)	FEAT_AdvSIMD
1	xx	10001	CMEQ (register)	FEAT_AdvSIMD
1	xx	10010	MLS (vector)	FEAT_AdvSIMD
1	xx	10011	PMUL	FEAT_AdvSIMD
1	xx	10100	UMAXP	FEAT_AdvSIMD
1	xx	10101	UMINP	FEAT_AdvSIMD
1	xx	10110	SQRDMULH (vector)	FEAT_AdvSIMD
1	xx	10111	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Condition
1	x1	11001	UNALLOCATED	-
1	0x	11000	FMAXNMP (vector)	FEAT_AdvSIMD
1	0x	11010	FADDP (vector)	FEAT_AdvSIMD
1	0x	11011	FMUL (vector)	FEAT_AdvSIMD
1	0x	11100	FCMGE (register)	FEAT_AdvSIMD
1	0x	11101	FACGE	FEAT_AdvSIMD
1	0x	11110	FMAXP (vector)	FEAT_AdvSIMD
1	0x	11111	FDIV (vector)	FEAT_AdvSIMD
1	00	00011	EOR (vector)	FEAT_AdvSIMD
1	00	11001	FMLAL, FMLAL2 (vector) — FMLAL2	FEAT_FHM
1	01	00011	BSL	FEAT_AdvSIMD
1	1x	11000	FMINNMP (vector)	FEAT_AdvSIMD
1	1x	11010	FABD	FEAT_AdvSIMD
1	1x	11011	FAMIN	FEAT_AdvSIMD && FEAT_FAMINMAX
1	1x	11100	FCMGT (register)	FEAT_AdvSIMD
1	1x	11101	FACGT	FEAT_AdvSIMD
1	1x	11110	FMINP (vector)	FEAT_AdvSIMD
1	1x	11111	FSCALE	FEAT_FP8
1	10	00011	BIT	FEAT_AdvSIMD
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	FEAT_FHM
1	11	00011	BIF	FEAT_AdvSIMD

Advanced SIMD modified immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22			19	18	17	16	15			12	11	10	9	8	7	6	5	4			0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				o2	1	d	e	f	g	h	Rd			

Q	op	cmode	o2	Instruction Details	Condition
x	0	0xx0	0	MOVI — 32-bit shifted immediate	FEAT_AdvSIMD
x	0	0xx1	0	ORR (vector, immediate) — 32-bit	FEAT_AdvSIMD
x	0	10x0	0	MOVI — 16-bit shifted immediate	FEAT_AdvSIMD
x	0	10x1	0	ORR (vector, immediate) — 16-bit	FEAT_AdvSIMD
x	0	110x	0	MOVI — 32-bit shifting ones	FEAT_AdvSIMD
x	0	1110	0	MOVI — 8-bit	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	op	cmode	o2	Instruction Details	Condition
x	0	1111	0	FMOV (vector, immediate) — Single-precision	FEAT_AdvSIMD
x	0	1111	1	FMOV (vector, immediate) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	!= 1111	1	UNALLOCATED	-
x	1	xxxx	1	UNALLOCATED	-
x	1	0xx0	0	MVNI — 32-bit shifted immediate	FEAT_AdvSIMD
x	1	0xx1	0	BIC (vector, immediate) — 32-bit	FEAT_AdvSIMD
x	1	10x0	0	MVNI — 16-bit shifted immediate	FEAT_AdvSIMD
x	1	10x1	0	BIC (vector, immediate) — 16-bit	FEAT_AdvSIMD
x	1	110x	0	MVNI — 32-bit shifting ones	FEAT_AdvSIMD
0	1	1110	0	MOVI — 64-bit scalar	FEAT_AdvSIMD
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	MOVI — 64-bit vector	FEAT_AdvSIMD
1	1	1111	0	FMOV (vector, immediate) — Double-precision	FEAT_AdvSIMD

Advanced SIMD shift by immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28			25	24	23	22	19	18	16	15	11	10	9	5	4	0
0	Q	U	0	1	1	1	1	0	!= 0000		immb		opcode		1	Rn		Rd	
									immh										

The following constraints also apply to this encoding: immh != ’0000’

U	immh	opcode	Instruction Details	Condition
x	!= 0000	0xxx1	UNALLOCATED	-
x	!= 0000	1x110	UNALLOCATED	-
x	!= 0000	101x1	UNALLOCATED	-
x	!= 0000	110xx	UNALLOCATED	-
x	!= 0000	11101	UNALLOCATED	-
0	!= 0000	00000	SSHR	FEAT_AdvSIMD
0	!= 0000	00010	SSRA	FEAT_AdvSIMD
0	!= 0000	00100	SRSHR	FEAT_AdvSIMD
0	!= 0000	00110	SRSRA	FEAT_AdvSIMD
0	!= 0000	01x00	UNALLOCATED	-
0	!= 0000	01010	SHL	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	immh	opcode	Instruction Details	Condition
0	!=0000	01110	SQSHL (immediate)	FEAT_AdvSIMD
0	!=0000	10000	SHRN, SHRN2	FEAT_AdvSIMD
0	!=0000	10001	RSHRN, RSHRN2	FEAT_AdvSIMD
0	!=0000	10010	SQSHRN, SQSHRN2	FEAT_AdvSIMD
0	!=0000	10011	SQRSHRN, SQRSHRN2	FEAT_AdvSIMD
0	!=0000	10100	SSHLL, SSHLL2	FEAT_AdvSIMD
0	!=0000	11100	SCVTF (vector, fixed-point)	FEAT_AdvSIMD
0	!=0000	11111	FCVTZS (vector, fixed-point)	FEAT_AdvSIMD
1	!=0000	00000	USHR	FEAT_AdvSIMD
1	!=0000	00010	USRA	FEAT_AdvSIMD
1	!=0000	00100	URSHR	FEAT_AdvSIMD
1	!=0000	00110	URSRA	FEAT_AdvSIMD
1	!=0000	01000	SRI	FEAT_AdvSIMD
1	!=0000	01010	SLI	FEAT_AdvSIMD
1	!=0000	01100	SQSHLU	FEAT_AdvSIMD
1	!=0000	01110	UQSHL (immediate)	FEAT_AdvSIMD
1	!=0000	10000	SQSHRUN, SQSHRUN2	FEAT_AdvSIMD
1	!=0000	10001	SQRSHRUN, SQRSHRUN2	FEAT_AdvSIMD
1	!=0000	10010	UQSHRN, UQSHRN2	FEAT_AdvSIMD
1	!=0000	10011	UQRSHRN, UQRSHRN2	FEAT_AdvSIMD
1	!=0000	10100	USHLL, USHLL2	FEAT_AdvSIMD
1	!=0000	11100	UCVTF (vector, fixed-point)	FEAT_AdvSIMD
1	!=0000	11111	FCVTZU (vector, fixed-point)	FEAT_AdvSIMD

Advanced SIMD vector x indexed element

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	19	16	15	12	11	10	9	5	4	0
0	Q	U	0	1	1	1	1	size		L	M	Rm		opcode		H	0	Rn		Rd	

Q	U	size	opcode	Instruction Details	Condition
x	x	01	1001	UNALLOCATED	-
x	0	xx	0010	SMLAL, SMLAL2 (by element)	FEAT_AdvSIMD
x	0	xx	0011	SQDMLAL, SQDMLAL2 (by element)	FEAT_AdvSIMD
x	0	xx	0110	SMLSL, SMLSL2 (by element)	FEAT_AdvSIMD
x	0	xx	0111	SQDMLSL, SQDMLSL2 (by element)	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Condition
x	0	xx	1000	MUL (by element)	FEAT_AdvSIMD
x	0	xx	1010	SMULL, SMULL2 (by element)	FEAT_AdvSIMD
x	0	xx	1011	SQDMULL, SQDMULL2 (by element)	FEAT_AdvSIMD
x	0	xx	1100	SQDMULH (by element)	FEAT_AdvSIMD
x	0	xx	1101	SQRDMULH (by element)	FEAT_AdvSIMD
x	0	xx	1110	SDOT (by element)	FEAT_DotProd
x	0	00	0000	FDOT (8-bit floating-point to single-precision, by element)	FEAT_FP8DOT4
x	0	00	0001	FMLA (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	0101	FMLS (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	1001	FMUL (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	0	00	1111	SUDOT (by element)	FEAT_I8MM
x	0	01	0x01	UNALLOCATED	-
x	0	01	0000	FDOT (8-bit floating-point to half-precision, by element)	FEAT_FP8DOT2
x	0	01	1111	BFDOT (by element)	FEAT_BF16
x	0	1x	0001	FMLA (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	0	1x	0101	FMLS (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	0	1x	1001	FMUL (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	0	10	0000	FMLAL, FMLAL2 (by element) — FMLAL	FEAT_FHM
x	0	10	0100	FMLSL, FMLSL2 (by element) — FMLSL	FEAT_FHM
x	0	10	1111	USDOT (by element)	FEAT_I8MM
x	0	11	1111	BFMLALB, BFMLALT (by element)	FEAT_BF16
x	0	!=10	0100	UNALLOCATED	-
x	1	xx	0xx1	FCMLA (by element)	FEAT_FCMA
x	1	xx	0000	MLA (by element)	FEAT_AdvSIMD
x	1	xx	0010	UMLAL, UMLAL2 (by element)	FEAT_AdvSIMD
x	1	xx	0100	MLS (by element)	FEAT_AdvSIMD
x	1	xx	0110	UMLSL, UMLSL2 (by element)	FEAT_AdvSIMD
x	1	xx	1010	UMULL, UMULL2 (by element)	FEAT_AdvSIMD
x	1	xx	1011	UNALLOCATED	-
x	1	xx	1101	SQRDMLAH (by element)	FEAT_RDM
x	1	xx	1110	UDOT (by element)	FEAT_DotProd
x	1	xx	1111	SQRDMLSH (by element)	FEAT_RDM
x	1	0x	1100	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Condition
x	1	00	1001	FMULX (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
x	1	1x	1001	FMULX (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
x	1	10	1000	FMLAL, FMLAL2 (by element) — FMLAL2	FEAT_FHM
x	1	10	1100	FMLSL, FMLSL2 (by element) — FMLSL2	FEAT_FHM
x	1	11	1x00	UNALLOCATED	-
0	0	11	0000	FMLALB, FMLALT (by element) — FMLALB	FEAT_FP8FMA
0	1	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLBB	FEAT_FP8FMA
0	1	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLBT	FEAT_FP8FMA
1	0	11	0000	FMLALB, FMLALT (by element) — FMLALT	FEAT_FP8FMA
1	1	00	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLTB	FEAT_FP8FMA
1	1	01	1000	FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element) — FMLALLTT	FEAT_FP8FMA

Cryptographic three-register, imm2

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–21	20–16	15–14	13–12	11–10	9–5	4–0
1100	111	00	10	Rm	10	imm2	opcode	Rn	Rd

opcode	Instruction Details	Condition
00	SM3TT1A	FEAT_SM3
01	SM3TT1B	FEAT_SM3
10	SM3TT2A	FEAT_SM3
11	SM3TT2B	FEAT_SM3

Cryptographic three-register SHA 512

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27–25	24–23	22–21	20–16	15	14	13–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 0	1 1	Rm	1	O	0 0	opcode	Rn	Rd

O	opcode	Instruction Details	Condition
0	00	SHA512H	FEAT_SHA512
0	01	SHA512H2	FEAT_SHA512
0	10	SHA512SU1	FEAT_SHA512
0	11	RAX1	FEAT_SHA3
1	00	SM3PARTW1	FEAT_SM3
1	01	SM3PARTW2	FEAT_SM3
1	10	SM4EKEY	FEAT_SM4
1	11	UNALLOCATED	-

Cryptographic four-register

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–21	20–16	15	14–10	9–5	4–0
1 1 0 0	1 1 1	0 0	Op0	Rm	0	Ra	Rn	Rd

Op0	Instruction Details	Condition
00	EOR3	FEAT_SHA3
01	BCAX	FEAT_SHA3
10	SM3SS1	FEAT_SM3
11	UNALLOCATED	-

Cryptographic two-register SHA 512

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31–28	27–25	24–23	22–19	18–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 1	1 0 0 0	0 0 0 1 0 0 0	opcode	Rn	Rd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opcode	Instruction Details	Condition
00	SHA512SU0	FEAT_SHA512
01	SM4E	FEAT_SM4
1x	UNALLOCATED	-

Conversion between floating-point and fixed-point

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28	24	23 22	21	20 19	18 16	15 10	9 5	4 0
sf	0	S	1 1 1 1	0	ftype	0	rmode	opcode	scale	Rn	Rd

sf	S	ftype	rmode	opcode	Instruction Details	Condition
x	0	xx	xx	1xx	UNALLOCATED	-
x	0	10	xx	0xx	UNALLOCATED	-
x	0	!= 10	x1	01x	UNALLOCATED	-
x	0	!= 10	0x	00x	UNALLOCATED	-
x	0	!= 10	10	0xx	UNALLOCATED	-
x	1	xx	xx	xxx	UNALLOCATED	-
0	0	00	00	010	SCVTF (scalar, fixed-point) — 32-bit to single-precision	FEAT_FP
0	0	00	00	011	UCVTF (scalar, fixed-point) — 32-bit to single-precision	FEAT_FP
0	0	00	11	000	FCVTZS (scalar, fixed-point) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	001	FCVTZU (scalar, fixed-point) — Single-precision to 32-bit	FEAT_FP
0	0	01	00	010	SCVTF (scalar, fixed-point) — 32-bit to double-precision	FEAT_FP
0	0	01	00	011	UCVTF (scalar, fixed-point) — 32-bit to double-precision	FEAT_FP
0	0	01	11	000	FCVTZS (scalar, fixed-point) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	001	FCVTZU (scalar, fixed-point) — Double-precision to 32-bit	FEAT_FP
0	0	11	00	010	SCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, fixed-point) — Half-precision to 32-bit	FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
0	0	11	11	001	FCVTZU (scalar, fixed-point) — Half-precision to 32-bit	FEAT_FP16
1	0	00	00	010	SCVTF (scalar, fixed-point) — 64-bit to single-precision	FEAT_FP
1	0	00	00	011	UCVTF (scalar, fixed-point) — 64-bit to single-precision	FEAT_FP
1	0	00	11	000	FCVTZS (scalar, fixed-point) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	001	FCVTZU (scalar, fixed-point) — Single-precision to 64-bit	FEAT_FP
1	0	01	00	010	SCVTF (scalar, fixed-point) — 64-bit to double-precision	FEAT_FP
1	0	01	00	011	UCVTF (scalar, fixed-point) — 64-bit to double-precision	FEAT_FP
1	0	01	11	000	FCVTZS (scalar, fixed-point) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	001	FCVTZU (scalar, fixed-point) — Double-precision to 64-bit	FEAT_FP
1	0	11	00	010	SCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, fixed-point) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, fixed-point) — Half-precision to 64-bit	FEAT_FP16

Conversion between floating-point and integer

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20	19	18	16	15					10	9	5	4	0
sf	0	S	1	1	1	1	0	ftype		1	rmode		opcode		0	0	0	0	0	0	Rn		Rd	

sf	S	ftype	rmode	opcode	Instruction Details	Condition
x	0	10	x0	xxx	UNALLOCATED	-
x	0	10	x1	!=11x	UNALLOCATED	-
x	0	11	x1	11x	UNALLOCATED	-
x	0	!=10	01	10x	UNALLOCATED	-
x	1	xx	xx	xxx	UNALLOCATED	-
0	0	x0	01	11x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
0	0	00	00	000	FCVTNS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	001	FCVTNU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	010	SCVTF (scalar, integer) — 32-bit to single-precision	FEAT_FP
0	0	00	00	011	UCVTF (scalar, integer) — 32-bit to single-precision	FEAT_FP
0	0	00	00	100	FCVTAS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	101	FCVTAU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	110	FMOV (general) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	111	FMOV (general) — 32-bit to single-precision	FEAT_FP
0	0	00	01	000	FCVTPS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	01	001	FCVTPU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	1x	1xx	UNALLOCATED	-
0	0	00	10	000	FCVTMS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	10	001	FCVTMU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	000	FCVTZS (scalar, integer) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	001	FCVTZU (scalar, integer) — Single-precision to 32-bit	FEAT_FP
0	0	00	!= 00	01x	UNALLOCATED	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	FCVTNS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	001	FCVTNU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	010	SCVTF (scalar, integer) — 32-bit to double-precision	FEAT_FP
0	0	01	00	011	UCVTF (scalar, integer) — 32-bit to double-precision	FEAT_FP
0	0	01	00	100	FCVTAS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	101	FCVTAU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	000	FCVTPS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	001	FCVTPU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	010	FCVTNS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
0	0	01	01	011	FCVTNU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	000	FCVTMS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	10	001	FCVTMU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	10	010	FCVTPS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	011	FCVTPU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	100	FCVTMS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	101	FCVTMU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	110	FCVTZS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	10	111	FCVTZU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	11	000	FCVTZS (scalar, integer) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	001	FCVTZU (scalar, integer) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	010	FCVTAS (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	11	011	FCVTAU (scalar SIMD&FP) — Double-precision to 32-bit	FEAT_FPRCVT
0	0	01	11	100	SCVTF (scalar SIMD&FP) — 32-bit to double-precision	FEAT_FPRCVT
0	0	01	11	101	UCVTF (scalar SIMD&FP) — 32-bit to double-precision	FEAT_FPRCVT
0	0	01	11	110	FJCVTZS	FEAT_JSCVT
0	0	01	11	111	UNALLOCATED	-
0	0	10	11	11x	UNALLOCATED	-
0	0	11	00	000	FCVTNS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	001	FCVTNU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	100	FCVTAS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	101	FCVTAU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	110	FMOV (general) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
0	0	11	01	000	FCVTPS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	01	001	FCVTPU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	01	010	FCVTNS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	01	011	FCVTNU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	000	FCVTMS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	10	001	FCVTMU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	10	010	FCVTPS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	011	FCVTPU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	100	FCVTMS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	101	FCVTMU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	110	FCVTZS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	10	111	FCVTZU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	11	000	FCVTZS (scalar, integer) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, integer) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	010	FCVTAS (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	11	011	FCVTAU (scalar SIMD&FP) — Half-precision to 32-bit	FEAT_FPRCVT
0	0	11	11	100	SCVTF (scalar SIMD&FP) — 32-bit to half-precision	FEAT_FPRCVT
0	0	11	11	101	UCVTF (scalar SIMD&FP) — 32-bit to half-precision	FEAT_FPRCVT
1	0	x0	11	11x	UNALLOCATED	-
1	0	00	0x	11x	UNALLOCATED	-
1	0	00	00	000	FCVTNS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	001	FCVTNU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	010	SCVTF (scalar, integer) — 64-bit to single-precision	FEAT_FP
1	0	00	00	011	UCVTF (scalar, integer) — 64-bit to single-precision	FEAT_FP
1	0	00	00	100	FCVTAS (scalar) — Single-precision to 64-bit	FEAT_FP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
1	0	00	00	101	FCVTAU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	000	FCVTPS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	001	FCVTPU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	010	FCVTNS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	01	011	FCVTNU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	000	FCVTMS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	10	001	FCVTMU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	10	010	FCVTPS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	011	FCVTPU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	100	FCVTMS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	101	FCVTMU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	110	FCVTZS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	10	111	FCVTZU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	11	000	FCVTZS (scalar, integer) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	001	FCVTZU (scalar, integer) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	010	FCVTAS (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	11	011	FCVTAU (scalar SIMD&FP) — Single-precision to 64-bit	FEAT_FPRCVT
1	0	00	11	100	SCVTF (scalar SIMD&FP) — 64-bit to single-precision	FEAT_FPRCVT
1	0	00	11	101	UCVTF (scalar SIMD&FP) — 64-bit to single-precision	FEAT_FPRCVT
1	0	01	00	000	FCVTNS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	001	FCVTNU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	010	SCVTF (scalar, integer) — 64-bit to double-precision	FEAT_FP
1	0	01	00	011	UCVTF (scalar, integer) — 64-bit to double-precision	FEAT_FP
1	0	01	00	100	FCVTAS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	101	FCVTAU (scalar) — Double-precision to 64-bit	FEAT_FP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
1	0	01	00	110	FMOV (general) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	111	FMOV (general) — 64-bit to double-precision	FEAT_FP
1	0	01	01	x1x	UNALLOCATED	-
1	0	01	01	000	FCVTPS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	01	001	FCVTPU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	1x	!= 00x	UNALLOCATED	-
1	0	01	10	000	FCVTMS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	10	001	FCVTMU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	000	FCVTZS (scalar, integer) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	001	FCVTZU (scalar, integer) — Double-precision to 64-bit	FEAT_FP
1	0	10	01	110	FMOV (general) — Top half of 128-bit to 64-bit	FEAT_FP
1	0	10	01	111	FMOV (general) — 64-bit to top half of 128-bit	FEAT_FP
1	0	11	00	000	FCVTNS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	001	FCVTNU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	100	FCVTAS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	101	FCVTAU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	110	FMOV (general) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	FEAT_FP16
1	0	11	01	000	FCVTPS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	01	001	FCVTPU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	01	010	FCVTNS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	01	011	FCVTNU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	000	FCVTMS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	10	001	FCVTMU (scalar) — Half-precision to 64-bit	FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Condition
1	0	11	10	010	FCVTPS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	011	FCVTPU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	100	FCVTMS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	101	FCVTMU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	110	FCVTZS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	10	111	FCVTZU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	11	000	FCVTZS (scalar, integer) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, integer) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	010	FCVTAS (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	11	011	FCVTAU (scalar SIMD&FP) — Half-precision to 64-bit	FEAT_FPRCVT
1	0	11	11	100	SCVTF (scalar SIMD&FP) — 64-bit to half-precision	FEAT_FPRCVT
1	0	11	11	101	UCVTF (scalar SIMD&FP) — 64-bit to half-precision	FEAT_FPRCVT

Floating-point data-processing (1 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				15	14				10	9			5	4			0
M	0	S	1	1	1	1	0	ftype		1	opcode					1	0	0	0	0	Rn				Rd			

M	S	ftype	opcode	Instruction Details	Condition
0	0	xx	1xxxxx	UNALLOCATED	-
0	0	0x	0101xx	UNALLOCATED	-
0	0	0x	011xxx	UNALLOCATED	-
0	0	00	000000	FMOV (register) — Single-precision	FEAT_FP
0	0	00	000001	FABS (scalar) — Single-precision	FEAT_FP
0	0	00	000010	FNEG (scalar) — Single-precision	FEAT_FP
0	0	00	000011	FSQRT (scalar) — Single-precision	FEAT_FP
0	0	00	0001x0	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Condition
0	0	00	000101	FCVT — Single-precision to double-precision	FEAT_FP
0	0	00	000111	FCVT — Single-precision to half-precision	FEAT_FP
0	0	00	001000	FRINTN (scalar) — Single-precision	FEAT_FP
0	0	00	001001	FRINTP (scalar) — Single-precision	FEAT_FP
0	0	00	001010	FRINTM (scalar) — Single-precision	FEAT_FP
0	0	00	001011	FRINTZ (scalar) — Single-precision	FEAT_FP
0	0	00	001100	FRINTA (scalar) — Single-precision	FEAT_FP
0	0	00	001110	FRINTX (scalar) — Single-precision	FEAT_FP
0	0	00	001111	FRINTI (scalar) — Single-precision	FEAT_FP
0	0	00	010000	FRINT32Z (scalar) — Single-precision	FEAT_FRINTTS
0	0	00	010001	FRINT32X (scalar) — Single-precision	FEAT_FRINTTS
0	0	00	010010	FRINT64Z (scalar) — Single-precision	FEAT_FRINTTS
0	0	00	010011	FRINT64X (scalar) — Single-precision	FEAT_FRINTTS
0	0	01	000000	FMOV (register) — Double-precision	FEAT_FP
0	0	01	000001	FABS (scalar) — Double-precision	FEAT_FP
0	0	01	000010	FNEG (scalar) — Double-precision	FEAT_FP
0	0	01	000011	FSQRT (scalar) — Double-precision	FEAT_FP
0	0	01	000100	FCVT — Double-precision to single-precision	FEAT_FP
0	0	01	000101	UNALLOCATED	-
0	0	01	000110	BFCVT	FEAT_BF16
0	0	01	000111	FCVT — Double-precision to half-precision	FEAT_FP
0	0	01	001000	FRINTN (scalar) — Double-precision	FEAT_FP
0	0	01	001001	FRINTP (scalar) — Double-precision	FEAT_FP
0	0	01	001010	FRINTM (scalar) — Double-precision	FEAT_FP
0	0	01	001011	FRINTZ (scalar) — Double-precision	FEAT_FP
0	0	01	001100	FRINTA (scalar) — Double-precision	FEAT_FP
0	0	01	001110	FRINTX (scalar) — Double-precision	FEAT_FP
0	0	01	001111	FRINTI (scalar) — Double-precision	FEAT_FP
0	0	01	010000	FRINT32Z (scalar) — Double-precision	FEAT_FRINTTS
0	0	01	010001	FRINT32X (scalar) — Double-precision	FEAT_FRINTTS
0	0	01	010010	FRINT64Z (scalar) — Double-precision	FEAT_FRINTTS
0	0	01	010011	FRINT64X (scalar) — Double-precision	FEAT_FRINTTS
0	0	10	0xxxxx	UNALLOCATED	-
0	0	11	000000	FMOV (register) — Half-precision	FEAT_FP16
0	0	11	000001	FABS (scalar) — Half-precision	FEAT_FP16
0	0	11	000010	FNEG (scalar) — Half-precision	FEAT_FP16
0	0	11	000011	FSQRT (scalar) — Half-precision	FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Condition
0	0	11	000100	FCVT — Half-precision to single-precision	FEAT_FP
0	0	11	000101	FCVT — Half-precision to double-precision	FEAT_FP
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — Half-precision	FEAT_FP16
0	0	11	001001	FRINTP (scalar) — Half-precision	FEAT_FP16
0	0	11	001010	FRINTM (scalar) — Half-precision	FEAT_FP16
0	0	11	001011	FRINTZ (scalar) — Half-precision	FEAT_FP16
0	0	11	001100	FRINTA (scalar) — Half-precision	FEAT_FP16
0	0	11	001110	FRINTX (scalar) — Half-precision	FEAT_FP16
0	0	11	001111	FRINTI (scalar) — Half-precision	FEAT_FP16
0	0	11	01xxxx	UNALLOCATED	-
0	0	!= 10	001101	UNALLOCATED	-
0	1	xx	xxxxxx	UNALLOCATED	-
1	x	xx	xxxxxx	UNALLOCATED	-

Floating-point compare

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20			16	15	14	13			10	9			5	4			0
M	0	S	1	1	1	1	0	ftype		1	Rm				op		1	0	0	0	Rn				opcode2			

M	S	ftype	op	opcode2	Instruction Details	Condition
0	0	xx	00	!= xx000	UNALLOCATED	-
0	0	xx	!= 00	xxxxx	UNALLOCATED	-
0	0	00	00	00000	FCMP — Single-precision	FEAT_FP
0	0	00	00	01000	FCMP — Single-precision, zero	FEAT_FP
0	0	00	00	10000	FCMPE — Single-precision	FEAT_FP
0	0	00	00	11000	FCMPE — Single-precision, zero	FEAT_FP
0	0	01	00	00000	FCMP — Double-precision	FEAT_FP
0	0	01	00	01000	FCMP — Double-precision, zero	FEAT_FP
0	0	01	00	10000	FCMPE — Double-precision	FEAT_FP
0	0	01	00	11000	FCMPE — Double-precision, zero	FEAT_FP
0	0	10	00	xx000	UNALLOCATED	-
0	0	11	00	00000	FCMP — Half-precision	FEAT_FP16
0	0	11	00	01000	FCMP — Half-precision, zero	FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	op	opcode2	Instruction Details	Condition
0	0	11	00	10000	FCMPE — Half-precision	FEAT_FP16
0	0	11	00	11000	FCMPE — Half-precision, zero	FEAT_FP16
0	1	xx	xx	xxxxx	UNALLOCATED	-
1	x	xx	xx	xxxxx	UNALLOCATED	-

Floating-point immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				13	12		10	9			5	4			0
M	0	S	1	1	1	1	0	ftype		1	imm8					1	0	0	imm5				Rd			

M	S	ftype	imm5	Instruction Details	Condition
0	0	xx	!= 00000	UNALLOCATED	-
0	0	00	00000	FMOV (scalar, immediate) — Single-precision	FEAT_FP
0	0	01	00000	FMOV (scalar, immediate) — Double-precision	FEAT_FP
0	0	10	00000	UNALLOCATED	-
0	0	11	00000	FMOV (scalar, immediate) — Half-precision	FEAT_FP16
0	1	xx	xxxxx	UNALLOCATED	-
1	x	xx	xxxxx	UNALLOCATED	-

Floating-point conditional compare

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			12	11	10	9				5	4	3			0
M	0	S	1	1	1	1	0	ftype		1	Rm					cond				0	1	Rn					op	nzcv			

M	S	ftype	op	Instruction Details	Condition
0	0	00	0	FCCMP — Single-precision	FEAT_FP
0	0	00	1	FCCMPE — Single-precision	FEAT_FP
0	0	01	0	FCCMP — Double-precision	FEAT_FP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	op	Instruction Details	Condition
0	0	01	1	FCCMPE — Double-precision	FEAT_FP
0	0	10	x	UNALLOCATED	-
0	0	11	0	FCCMP — Half-precision	FEAT_FP16
0	0	11	1	FCCMPE — Half-precision	FEAT_FP16
0	1	xx	x	UNALLOCATED	-
1	x	xx	x	UNALLOCATED	-

Floating-point data-processing (2 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20				16	15			12	11	10	9				5	4				0
M	0	S	1	1	1	1	0	ftype		1	Rm					opcode				1	0	Rn					Rd				

M	S	ftype	opcode	Instruction Details	Condition
0	0	00	0000	FMUL (scalar) — Single-precision	FEAT_FP
0	0	00	0001	FDIV (scalar) — Single-precision	FEAT_FP
0	0	00	0010	FADD (scalar) — Single-precision	FEAT_FP
0	0	00	0011	FSUB (scalar) — Single-precision	FEAT_FP
0	0	00	0100	FMAX (scalar) — Single-precision	FEAT_FP
0	0	00	0101	FMIN (scalar) — Single-precision	FEAT_FP
0	0	00	0110	FMAXNM (scalar) — Single-precision	FEAT_FP
0	0	00	0111	FMINNM (scalar) — Single-precision	FEAT_FP
0	0	00	1000	FNMUL (scalar) — Single-precision	FEAT_FP
0	0	01	0000	FMUL (scalar) — Double-precision	FEAT_FP
0	0	01	0001	FDIV (scalar) — Double-precision	FEAT_FP
0	0	01	0010	FADD (scalar) — Double-precision	FEAT_FP
0	0	01	0011	FSUB (scalar) — Double-precision	FEAT_FP
0	0	01	0100	FMAX (scalar) — Double-precision	FEAT_FP
0	0	01	0101	FMIN (scalar) — Double-precision	FEAT_FP
0	0	01	0110	FMAXNM (scalar) — Double-precision	FEAT_FP
0	0	01	0111	FMINNM (scalar) — Double-precision	FEAT_FP
0	0	01	1000	FNMUL (scalar) — Double-precision	FEAT_FP
0	0	10	xxxx	UNALLOCATED	-
0	0	11	0000	FMUL (scalar) — Half-precision	FEAT_FP16
0	0	11	0001	FDIV (scalar) — Half-precision	FEAT_FP16
0	0	11	0010	FADD (scalar) — Half-precision	FEAT_FP16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Condition
0	0	11	0011	FSUB (scalar) — Half-precision	FEAT_FP16
0	0	11	0100	FMAX (scalar) — Half-precision	FEAT_FP16
0	0	11	0101	FMIN (scalar) — Half-precision	FEAT_FP16
0	0	11	0110	FMAXNM (scalar) — Half-precision	FEAT_FP16
0	0	11	0111	FMINNM (scalar) — Half-precision	FEAT_FP16
0	0	11	1000	FNMUL (scalar) — Half-precision	FEAT_FP16
0	0	!=10	1001	UNALLOCATED	-
0	0	!=10	101x	UNALLOCATED	-
0	0	!=10	11xx	UNALLOCATED	-
0	1	xx	xxxx	UNALLOCATED	-
1	x	xx	xxxx	UNALLOCATED	-

Floating-point conditional select

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 16	15 12	11 10	9 5	4 0
M	0	S	1	1	1	1	0	ftype	1	Rm	cond	1 1	Rn	Rd

M	S	ftype	Instruction Details	Condition
0	0	00	FCSEL — Single-precision	FEAT_FP
0	0	01	FCSEL — Double-precision	FEAT_FP
0	0	10	UNALLOCATED	-
0	0	11	FCSEL — Half-precision	FEAT_FP16
0	1	xx	UNALLOCATED	-
1	x	xx	UNALLOCATED	-

Floating-point data-processing (3 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 16	15	14 10	9 5	4 0
M	0	S	1	1	1	1	1	ftype	o1	Rm	o0	Ra	Rn	Rd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	o1	o0	Instruction Details	Condition
0	0	00	0	0	FMADD — Single-precision	FEAT_FP
0	0	00	0	1	FMSUB — Single-precision	FEAT_FP
0	0	00	1	0	FNMADD — Single-precision	FEAT_FP
0	0	00	1	1	FNMSUB — Single-precision	FEAT_FP
0	0	01	0	0	FMADD — Double-precision	FEAT_FP
0	0	01	0	1	FMSUB — Double-precision	FEAT_FP
0	0	01	1	0	FNMADD — Double-precision	FEAT_FP
0	0	01	1	1	FNMSUB — Double-precision	FEAT_FP
0	0	10	x	x	UNALLOCATED	-
0	0	11	0	0	FMADD — Half-precision	FEAT_FP16
0	0	11	0	1	FMSUB — Half-precision	FEAT_FP16
0	0	11	1	0	FNMADD — Half-precision	FEAT_FP16
0	0	11	1	1	FNMSUB — Half-precision	FEAT_FP16
0	1	xx	x	x	UNALLOCATED	-
1	x	xx	x	x	UNALLOCATED	-

Loads and Stores

Loads and Stores

The encodings in this section are decoded from A64 instruction set encoding.

31–28	27	26	25	24–10	9–0
op0	1	op1	0	op2	

op0	op1	op2	Instruction details
0000	0	1xx1xxxxxxxxxxx	UNALLOCATED
0001	1	1xx1xxxxxxxxxxx	UNALLOCATED
0x00	0	00x1xxxxxxxxxxx	Compare and swap pair
0x00	0	10x0xxxxxxxxxxx	UNALLOCATED
0x00	0	11x0xxxxxxxxxxx	Compare and swap pair (unprivileged)
0x00	1	00x000000xxxxxx	Advanced SIMD load/store multiple structures
0x00	1	00x000001xxxxxx	UNALLOCATED
0x00	1	01x0xxxxxxxxxxx	Advanced SIMD load/store multiple structures (post-indexed)
0x00	1	0xx1xxxxxxxxxxx	UNALLOCATED
0x00	1	10x10001xxxxxxx	UNALLOCATED
0x00	1	10x1001xxxxxxxx	UNALLOCATED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
0x00	1	10x101xxxxxxxxx	UNALLOCATED
0x00	1	10x11xxxxxxxxxx	UNALLOCATED
0x00	1	10xx0000xxxxxxx	Advanced SIMD load/store single structure
0x00	1	11xxxxxxxxxxxxx	Advanced SIMD load/store single structure (post-indexed)
0x00	1	x0x00001xxxxxxx	UNALLOCATED
0x00	1	x0x0001xxxxxxxx	UNALLOCATED
0x00	1	x0x001xxxxxxxxx	UNALLOCATED
0x00	1	x0x01xxxxxxxxxx	UNALLOCATED
0x01	0	1xx0xxxxxxxxx11	UNALLOCATED
0x01	0	1xx1xxxxx000010	RCW compare and swap
0x01	0	1xx1xxxxx000011	RCW compare and swap pair
0x01	0	1xx1xxxxx00011x	UNALLOCATED
0x01	0	1xx1xxxxx001x1x	UNALLOCATED
0x01	0	1xx1xxxxx01xx1x	UNALLOCATED
0x01	0	1xx1xxxxx1xxx1x	UNALLOCATED
0x01	0	1xx1xxxxxxxxx00	128-bit atomic memory operations
0x01	0	1xx1xxxxxxxxx01	Atomic memory operations (unprivileged)
1001	0	1xx0xxxxxxxxx11	UNALLOCATED
1001	1	1xx1xxxxxxxxxxx	UNALLOCATED
100x	0	1xx1xxxxxxxxxxx	UNALLOCATED
1101	0	1000111110xxx11	GCS load/store
1101	0	1100111110xxx11	UNALLOCATED
1101	0	1x000xxxxxxxx11	UNALLOCATED
1101	0	1x0010xxxxxxx11	UNALLOCATED
1101	0	1x00110xxxxxx11	UNALLOCATED
1101	0	1x001110xxxxx11	UNALLOCATED
1101	0	1x0011110xxxx11	UNALLOCATED
1101	0	1x00111111xxx11	UNALLOCATED
1101	0	1x10xxxxxxxx11	UNALLOCATED
1101	0	1xx1xxxxxxxxxxx	Load/store memory tags
1x00	0	00x1xxxxxxxxxxx	Load/store exclusive pair
1x00	0	10x0xxxxxxxxxxx	Load/store exclusive register (unprivileged)
1x00	0	11x0xxxxxxxxxxx	Compare and swap (unprivileged)
1x00	1	xxxxxxxxxxxxxxx	UNALLOCATED
x100	0	1xx1xxxxxxxxxxx	UNALLOCATED
x101	1	1xx1xxxxxxxxxxx	UNALLOCATED
xx00	0	00x0xxxxxxxxxxx	Load/store exclusive register
xx00	0	01x0xxxxxxxxxxx	Load/store ordered

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
xx00	0	01x1xxxxxxxxxx	Compare and swap
xx01	0	10x0xxxxxxxxx10	Load/store ordered register pair
xx01	0	11x000000000010	Load/store ordered (writeback)
xx01	0	11x000000000110	UNALLOCATED
xx01	0	11x000000001x10	UNALLOCATED
xx01	0	11x00000001xx10	UNALLOCATED
xx01	0	11x0000001xxx10	UNALLOCATED
xx01	0	11x000001xxxx10	UNALLOCATED
xx01	0	11x00001xxxxx10	UNALLOCATED
xx01	0	11x0001xxxxxx10	UNALLOCATED
xx01	0	11x001xxxxxxx10	UNALLOCATED
xx01	0	11x01xxxxxxxx10	UNALLOCATED
xx01	0	1xx0xxxxxxxx00	Load/store ordered (unscaled immediate)
xx01	1	1xx0xxxxxxxx00	UNALLOCATED
xx01	1	1xx0xxxxxxxx10	Load/store ordered (SIMD&FP)
xx01	1	1xx0xxxxxxxx11	UNALLOCATED
xx01	x	0xxxxxxxxxxxxx	Load register (literal)
xx01	x	1xx0xxxxxxxx01	Memory Copy and Memory Set
xx10	x	00xxxxxxxxxxxx	Load/store no-allocate pair (offset)
xx10	x	01xxxxxxxxxxxx	Load/store register pair (post-indexed)
xx10	x	10xxxxxxxxxxxx	Load/store register pair (offset)
xx10	x	11xxxxxxxxxxxx	Load/store register pair (pre-indexed)
xx11	x	0xx0xxxxxxxx00	Load/store register (unscaled immediate)
xx11	x	0xx0xxxxxxxx01	Load/store register (immediate post-indexed)
xx11	x	0xx0xxxxxxxx10	Load/store register (unprivileged)
xx11	x	0xx0xxxxxxxx11	Load/store register (immediate pre-indexed)
xx11	x	0xx1xxxxxxxx00	Atomic memory operations
xx11	x	0xx1xxxxxxxx10	Load/store register (register offset)
xx11	x	0xx1xxxxxxxxx1	Load/store register (pac)
xx11	x	1xxxxxxxxxxxxx	Load/store register (unsigned immediate)

Compare and swap pair

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
0	sz	0	0 1	0	0	0 0	L	1	Rs		o0	Rt2		Rn		Rt	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	L	o0	Rt2	Instruction Details	Condition
x	x	x	!= 11111	UNALLOCATED	-
0	0	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP	FEAT_LSE
0	0	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL	FEAT_LSE
0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA	FEAT_LSE
0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL	FEAT_LSE
1	0	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP	FEAT_LSE
1	0	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL	FEAT_LSE
1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA	FEAT_LSE
1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL	FEAT_LSE

Compare and swap pair (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
0	sz	0	0 1	0	0	1 1	L	0	Rs		o0	Rt2		Rn		Rt	

sz	L	o0	Rt2	Instruction Details	Condition
0	x	x	xxxxx	UNALLOCATED	-
1	x	x	!= 11111	UNALLOCATED	-
1	0	0	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPT	FEAT_LSUI
1	0	1	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPLT	FEAT_LSUI
1	1	0	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPAT	FEAT_LSUI
1	1	1	11111	CASPT, CASPAT, CASPALT, CASPLT — CASPALT	FEAT_LSUI

Advanced SIMD load/store multiple structures

The encodings in this section are decoded from Loads and Stores.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	27	26	25	23	22	21	16	15	12	11	10	9	5	4	0
0	Q	0	0 1	1	0	0 0	L	0	0 0 0 0 0	opcode		size		Rn		Rt	

L	opcode	Instruction Details	Condition
x	x0x1	UNALLOCATED	-
x	0101	UNALLOCATED	-
x	11xx	UNALLOCATED	-
0	0000	ST4 (multiple structures)	FEAT_AdvSIMD
0	0010	ST1 (multiple structures) — Four registers	FEAT_AdvSIMD
0	0100	ST3 (multiple structures)	FEAT_AdvSIMD
0	0110	ST1 (multiple structures) — Three registers	FEAT_AdvSIMD
0	0111	ST1 (multiple structures) — One register	FEAT_AdvSIMD
0	1000	ST2 (multiple structures)	FEAT_AdvSIMD
0	1010	ST1 (multiple structures) — Two registers	FEAT_AdvSIMD
1	0000	LD4 (multiple structures)	FEAT_AdvSIMD
1	0010	LD1 (multiple structures) — Four registers	FEAT_AdvSIMD
1	0100	LD3 (multiple structures)	FEAT_AdvSIMD
1	0110	LD1 (multiple structures) — Three registers	FEAT_AdvSIMD
1	0111	LD1 (multiple structures) — One register	FEAT_AdvSIMD
1	1000	LD2 (multiple structures)	FEAT_AdvSIMD
1	1010	LD1 (multiple structures) — Two registers	FEAT_AdvSIMD

Advanced SIMD load/store multiple structures (post-indexed)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	12	11	10	9	5	4	0
0	Q	0	0 1	1	0	0 1	L	0	Rm		opcode		size		Rn		Rt	

L	Rm	opcode	Instruction Details	Condition
x	xxxxx	x0x1	UNALLOCATED	-
x	xxxxx	0101	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	Rm	opcode	Instruction Details	Condition
x	xxxxx	11xx	UNALLOCATED	-
0	11111	0000	ST4 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	0010	ST1 (multiple structures) — Four registers, immediate offset	FEAT_AdvSIMD
0	11111	0100	ST3 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	0110	ST1 (multiple structures) — Three registers, immediate offset	FEAT_AdvSIMD
0	11111	0111	ST1 (multiple structures) — One register, immediate offset	FEAT_AdvSIMD
0	11111	1000	ST2 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	1010	ST1 (multiple structures) — Two registers, immediate offset	FEAT_AdvSIMD
0	!= 11111	0000	ST4 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!= 11111	0010	ST1 (multiple structures) — Four registers, register offset	FEAT_AdvSIMD
0	!= 11111	0100	ST3 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!= 11111	0110	ST1 (multiple structures) — Three registers, register offset	FEAT_AdvSIMD
0	!= 11111	0111	ST1 (multiple structures) — One register, register offset	FEAT_AdvSIMD
0	!= 11111	1000	ST2 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!= 11111	1010	ST1 (multiple structures) — Two registers, register offset	FEAT_AdvSIMD
1	11111	0000	LD4 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	0010	LD1 (multiple structures) — Four registers, immediate offset	FEAT_AdvSIMD
1	11111	0100	LD3 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	0110	LD1 (multiple structures) — Three registers, immediate offset	FEAT_AdvSIMD
1	11111	0111	LD1 (multiple structures) — One register, immediate offset	FEAT_AdvSIMD
1	11111	1000	LD2 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	1010	LD1 (multiple structures) — Two registers, immediate offset	FEAT_AdvSIMD
1	!= 11111	0000	LD4 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!= 11111	0010	LD1 (multiple structures) — Four registers, register offset	FEAT_AdvSIMD
1	!= 11111	0100	LD3 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!= 11111	0110	LD1 (multiple structures) — Three registers, register offset	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	Rm	opcode	Instruction Details	Condition
1	!=11111	0111	LD1 (multiple structures) — One register, register offset	FEAT_AdvSIMD
1	!=11111	1000	LD2 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!=11111	1010	LD1 (multiple structures) — Two registers, register offset	FEAT_AdvSIMD

Advanced SIMD load/store single structure

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	17	16	15	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	L	R	0	0	o2	opcode		S	size		Rn		Rt	

L	R	o2	opcode	S	size	Instruction Details	Condition
x	x	0	01x	x	x1	UNALLOCATED	-
x	x	0	10x	x	1x	UNALLOCATED	-
x	x	0	10x	1	01	UNALLOCATED	-
x	0	1	100	x	!=01	UNALLOCATED	-
x	0	1	100	1	01	UNALLOCATED	-
x	0	1	!=100	x	xx	UNALLOCATED	-
x	1	1	xxx	x	xx	UNALLOCATED	-
0	x	0	11x	x	xx	UNALLOCATED	-
0	0	0	000	x	xx	ST1 (single structure) — 8-bit	FEAT_AdvSIMD
0	0	0	001	x	xx	ST3 (single structure) — 8-bit	FEAT_AdvSIMD
0	0	0	010	x	x0	ST1 (single structure) — 16-bit	FEAT_AdvSIMD
0	0	0	011	x	x0	ST3 (single structure) — 16-bit	FEAT_AdvSIMD
0	0	0	100	x	00	ST1 (single structure) — 32-bit	FEAT_AdvSIMD
0	0	0	100	0	01	ST1 (single structure) — 64-bit	FEAT_AdvSIMD
0	0	0	101	x	00	ST3 (single structure) — 32-bit	FEAT_AdvSIMD
0	0	0	101	0	01	ST3 (single structure) — 64-bit	FEAT_AdvSIMD
0	0	1	100	0	01	STL1 (SIMD&FP)	FEAT_AdvSIMD && FEAT_LRCPC3
0	1	0	000	x	xx	ST2 (single structure) — 8-bit	FEAT_AdvSIMD
0	1	0	001	x	xx	ST4 (single structure) — 8-bit	FEAT_AdvSIMD
0	1	0	010	x	x0	ST2 (single structure) — 16-bit	FEAT_AdvSIMD
0	1	0	011	x	x0	ST4 (single structure) — 16-bit	FEAT_AdvSIMD
0	1	0	100	x	00	ST2 (single structure) — 32-bit	FEAT_AdvSIMD
0	1	0	100	0	01	ST2 (single structure) — 64-bit	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	o2	opcode	S	size	Instruction Details	Condition
0	1	0	101	x	00	ST4 (single structure) — 32-bit	FEAT_AdvSIMD
0	1	0	101	0	01	ST4 (single structure) — 64-bit	FEAT_AdvSIMD
1	x	0	11x	1	xx	UNALLOCATED	-
1	0	0	000	x	xx	LD1 (single structure) — 8-bit	FEAT_AdvSIMD
1	0	0	001	x	xx	LD3 (single structure) — 8-bit	FEAT_AdvSIMD
1	0	0	010	x	x0	LD1 (single structure) — 16-bit	FEAT_AdvSIMD
1	0	0	011	x	x0	LD3 (single structure) — 16-bit	FEAT_AdvSIMD
1	0	0	100	x	00	LD1 (single structure) — 32-bit	FEAT_AdvSIMD
1	0	0	100	0	01	LD1 (single structure) — 64-bit	FEAT_AdvSIMD
1	0	0	101	x	00	LD3 (single structure) — 32-bit	FEAT_AdvSIMD
1	0	0	101	0	01	LD3 (single structure) — 64-bit	FEAT_AdvSIMD
1	0	0	110	0	xx	LD1R	FEAT_AdvSIMD
1	0	0	111	0	xx	LD3R	FEAT_AdvSIMD
1	0	1	100	0	01	LDAP1 (SIMD&FP)	FEAT_AdvSIMD && FEAT_LRCPC3
1	1	0	000	x	xx	LD2 (single structure) — 8-bit	FEAT_AdvSIMD
1	1	0	001	x	xx	LD4 (single structure) — 8-bit	FEAT_AdvSIMD
1	1	0	010	x	x0	LD2 (single structure) — 16-bit	FEAT_AdvSIMD
1	1	0	011	x	x0	LD4 (single structure) — 16-bit	FEAT_AdvSIMD
1	1	0	100	x	00	LD2 (single structure) — 32-bit	FEAT_AdvSIMD
1	1	0	100	0	01	LD2 (single structure) — 64-bit	FEAT_AdvSIMD
1	1	0	101	x	00	LD4 (single structure) — 32-bit	FEAT_AdvSIMD
1	1	0	101	0	01	LD4 (single structure) — 64-bit	FEAT_AdvSIMD
1	1	0	110	0	xx	LD2R	FEAT_AdvSIMD
1	1	0	111	0	xx	LD4R	FEAT_AdvSIMD

Advanced SIMD load/store single structure (post-indexed)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	0	1	1	L	R	Rm		opcode		S	size	Rn		Rt

L	R	Rm	opcode	S	size	Instruction Details	Condition
x	x	xxxxx	01x	x	x1	UNALLOCATED	-
x	x	xxxxx	10x	x	1x	UNALLOCATED	-
x	x	xxxxx	10x	1	01	UNALLOCATED	-
0	x	xxxxx	11x	x	xx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Condition
0	0	11111	000	x	xx	ST1 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	001	x	xx	ST3 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	010	x	x0	ST1 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	011	x	x0	ST3 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	100	x	00	ST1 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	100	0	01	ST1 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	101	x	00	ST3 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	101	0	01	ST3 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	0	!= 11111	000	x	xx	ST1 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	001	x	xx	ST3 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	010	x	x0	ST1 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	011	x	x0	ST3 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	100	x	00	ST1 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	100	0	01	ST1 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	101	x	00	ST3 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	101	0	01	ST3 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	1	11111	000	x	xx	ST2 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	001	x	xx	ST4 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	010	x	x0	ST2 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	011	x	x0	ST4 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	100	x	00	ST2 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	100	0	01	ST2 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	101	x	00	ST4 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	101	0	01	ST4 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	1	!= 11111	000	x	xx	ST2 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	001	x	xx	ST4 (single structure) — 8-bit, register offset	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Condition
0	1	!= 11111	010	x	x0	ST2 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	011	x	x0	ST4 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	100	x	00	ST2 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	100	0	01	ST2 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	101	x	00	ST4 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	101	0	01	ST4 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	x	xxxxx	11x	1	xx	UNALLOCATED	-
1	0	11111	000	x	xx	LD1 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	001	x	xx	LD3 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	010	x	x0	LD1 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	011	x	x0	LD3 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	100	x	00	LD1 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	100	0	01	LD1 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	101	x	00	LD3 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	101	0	01	LD3 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	110	0	xx	LD1R — Immediate offset	FEAT_AdvSIMD
1	0	11111	111	0	xx	LD3R — Immediate offset	FEAT_AdvSIMD
1	0	!= 11111	000	x	xx	LD1 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	001	x	xx	LD3 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	010	x	x0	LD1 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	011	x	x0	LD3 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	100	x	00	LD1 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	100	0	01	LD1 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	101	x	00	LD3 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	101	0	01	LD3 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	110	0	xx	LD1R — Register offset	FEAT_AdvSIMD
1	0	!= 11111	111	0	xx	LD3R — Register offset	FEAT_AdvSIMD
1	1	11111	000	x	xx	LD2 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Condition
1	1	11111	001	x	xx	LD4 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	010	x	x0	LD2 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	011	x	x0	LD4 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	100	x	00	LD2 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	100	0	01	LD2 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	101	x	00	LD4 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	101	0	01	LD4 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	110	0	xx	LD2R — Immediate offset	FEAT_AdvSIMD
1	1	11111	111	0	xx	LD4R — Immediate offset	FEAT_AdvSIMD
1	1	!= 11111	000	x	xx	LD2 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	001	x	xx	LD4 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	010	x	x0	LD2 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	011	x	x0	LD4 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	100	x	00	LD2 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	100	0	01	LD2 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	101	x	00	LD4 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	101	0	01	LD4 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	110	0	xx	LD2R — Register offset	FEAT_AdvSIMD
1	1	!= 11111	111	0	xx	LD4R — Register offset	FEAT_AdvSIMD

RCW compare and swap

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
0	S	0	1 1	0	0	1	A	R	1	Rs		0 0 0 0 1 0		Rn		Rt	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	Instruction Details	Condition
0	0	0	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCAS	FEAT_THE
0	0	1	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCASL	FEAT_THE
0	1	0	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCASA	FEAT_THE
0	1	1	RCWCAS, RCWCASA, RCWCASAL, RCWCASL — RCWCASAL	FEAT_THE
1	0	0	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCAS	FEAT_THE
1	0	1	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCASL	FEAT_THE
1	1	0	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCASA	FEAT_THE
1	1	1	RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL — RCWSCASAL	FEAT_THE

RCW compare and swap pair

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	10	9	5	4	0
0	S	0	1	1	0	0	1	A	R	1	Rs	0	0	0	0	1	1	Rn	Rt

S	A	R	Instruction Details	Condition
0	0	0	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASP	FEAT_D128 && FEAT_THE
0	0	1	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASPL	FEAT_D128 && FEAT_THE
0	1	0	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASPA	FEAT_D128 && FEAT_THE
0	1	1	RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL — RCWCASPAL	FEAT_D128 && FEAT_THE
1	0	0	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASP	FEAT_D128 && FEAT_THE
1	0	1	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASPL	FEAT_D128 && FEAT_THE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	Instruction Details	Condition
1	1	0	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASPA	FEAT_D128 && FEAT_THE
1	1	1	RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL — RCWSCASPAL	FEAT_D128 && FEAT_THE

128-bit atomic memory operations

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	S	0	1	1	0	0	1	A	R	1	Rt2		o3	opc	0	0	Rn		Rt	

S	A	R	o3	opc	Instruction Details	Condition
x	x	x	x	1xx	UNALLOCATED	-
0	x	x	0	0x0	UNALLOCATED	-
0	0	0	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRP	FEAT_LSE128
0	0	0	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETP	FEAT_LSE128
0	0	0	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPP	FEAT_LSE128
0	0	0	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRP	FEAT_D128 && FEAT_THE
0	0	0	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPP	FEAT_D128 && FEAT_THE
0	0	0	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETP	FEAT_D128 && FEAT_THE
0	0	1	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPL	FEAT_LSE128
0	0	1	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPL	FEAT_LSE128
0	0	1	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPPL	FEAT_LSE128
0	0	1	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRPL	FEAT_D128 && FEAT_THE
0	0	1	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPPL	FEAT_D128 && FEAT_THE
0	0	1	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETPL	FEAT_D128 && FEAT_THE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	o3	opc	Instruction Details	Condition
0	1	0	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPA	FEAT_LSE128
0	1	0	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPA	FEAT_LSE128
0	1	0	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPPA	FEAT_LSE128
0	1	0	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRPA	FEAT_D128 && FEAT_THE
0	1	0	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPPA	FEAT_D128 && FEAT_THE
0	1	0	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETPA	FEAT_D128 && FEAT_THE
0	1	1	0	001	LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPAL	FEAT_LSE128
0	1	1	0	011	LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPAL	FEAT_LSE128
0	1	1	1	000	SWPP, SWPPA, SWPPAL, SWPPL — SWPPAL	FEAT_LSE128
0	1	1	1	001	RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL — RCWCLRPAL	FEAT_D128 && FEAT_THE
0	1	1	1	010	RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL — RCWSWPPAL	FEAT_D128 && FEAT_THE
0	1	1	1	011	RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL — RCWSETPAL	FEAT_D128 && FEAT_THE
1	x	x	0	0xx	UNALLOCATED	-
1	x	x	1	000	UNALLOCATED	-
1	0	0	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRP	FEAT_D128 && FEAT_THE
1	0	0	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPP	FEAT_D128 && FEAT_THE
1	0	0	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETP	FEAT_D128 && FEAT_THE
1	0	1	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRPL	FEAT_D128 && FEAT_THE
1	0	1	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPPL	FEAT_D128 && FEAT_THE
1	0	1	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETPL	FEAT_D128 && FEAT_THE
1	1	0	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRPA	FEAT_D128 && FEAT_THE
1	1	0	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPPA	FEAT_D128 && FEAT_THE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	A	R	o3	opc	Instruction Details	Condition
1	1	0	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETPA	FEAT_D128 && FEAT_THE
1	1	1	1	001	RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL — RCWSCLRPAL	FEAT_D128 && FEAT_THE
1	1	1	1	010	RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL — RCWSSWPPAL	FEAT_D128 && FEAT_THE
1	1	1	1	011	RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL — RCWSSETPAL	FEAT_D128 && FEAT_THE

Atomic memory operations (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	sz	0	1	1	0	0	1	A	R	1	Rs	o3	opc		0	1	Rn		Rt	

sz	A	R	o3	opc	Instruction Details	Condition
x	x	x	x	010	UNALLOCATED	-
x	x	x	x	1xx	UNALLOCATED	-
x	x	x	1	0x1	UNALLOCATED	-
0	0	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit no memory ordering	FEAT_LSUI
0	0	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit no memory ordering	FEAT_LSUI
0	0	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit no memory ordering	FEAT_LSUI
0	0	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPT	FEAT_LSUI
0	0	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit release	FEAT_LSUI
0	0	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit release	FEAT_LSUI
0	0	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit release	FEAT_LSUI
0	0	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPTL	FEAT_LSUI
0	1	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit acquire	FEAT_LSUI
0	1	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit acquire	FEAT_LSUI

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	A	R	o3	opc	Instruction Details	Condition
0	1	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit acquire	FEAT_LSUI
0	1	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPTA	FEAT_LSUI
0	1	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 32-bit acquire-release	FEAT_LSUI
0	1	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 32-bit acquire-release	FEAT_LSUI
0	1	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 32-bit acquire-release	FEAT_LSUI
0	1	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 32-bit SWPTAL	FEAT_LSUI
1	0	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit no memory ordering	FEAT_LSUI
1	0	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit no memory ordering	FEAT_LSUI
1	0	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit no memory ordering	FEAT_LSUI
1	0	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPT	FEAT_LSUI
1	0	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit release	FEAT_LSUI
1	0	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit release	FEAT_LSUI
1	0	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit release	FEAT_LSUI
1	0	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPTL	FEAT_LSUI
1	1	0	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit acquire	FEAT_LSUI
1	1	0	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit acquire	FEAT_LSUI
1	1	0	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit acquire	FEAT_LSUI
1	1	0	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPTA	FEAT_LSUI
1	1	1	0	000	LDTADD, LDTADDA, LDTADDAL, LDTADDL — 64-bit acquire-release	FEAT_LSUI
1	1	1	0	001	LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL — 64-bit acquire-release	FEAT_LSUI
1	1	1	0	011	LDTSET, LDTSETA, LDTSETAL, LDTSETL — 64-bit acquire-release	FEAT_LSUI
1	1	1	1	000	SWPT, SWPTA, SWPTAL, SWPTL — 64-bit SWPTAL	FEAT_LSUI

GCS load/store

The encodings in this section are decoded from Loads and Stores.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27	26	25–15	14–12	11–10	9–5	4–0
1101	1	0	01000111110	opc	11	Rn	Rt

opc	Instruction Details	Condition
000	GCSSTR	FEAT_GCS
001	GCSSTTR	FEAT_GCS
!= 00x	UNALLOCATED	-

Load/store memory tags

The encodings in this section are decoded from Loads and Stores.

31–28	27	26	25–24	23–22	21	20–12	11–10	9–5	4–0
1101	1	0	01	opc	1	imm9	op2	Rn	Rt

opc	imm9	op2	Instruction Details	Condition
00	xxxxxxxxx	01	STG — Post-index	FEAT_MTE
00	xxxxxxxxx	10	STG — Signed offset	FEAT_MTE
00	xxxxxxxxx	11	STG — Pre-index	FEAT_MTE
00	000000000	00	STZGM	FEAT_MTE2
01	xxxxxxxxx	00	LDG	FEAT_MTE
01	xxxxxxxxx	01	STZG — Post-index	FEAT_MTE
01	xxxxxxxxx	10	STZG — Signed offset	FEAT_MTE
01	xxxxxxxxx	11	STZG — Pre-index	FEAT_MTE
10	xxxxxxxxx	01	ST2G — Post-index	FEAT_MTE
10	xxxxxxxxx	10	ST2G — Signed offset	FEAT_MTE
10	xxxxxxxxx	11	ST2G — Pre-index	FEAT_MTE
10	000000000	00	STGM	FEAT_MTE2
11	xxxxxxxxx	01	STZ2G — Post-index	FEAT_MTE
11	xxxxxxxxx	10	STZ2G — Signed offset	FEAT_MTE
11	xxxxxxxxx	11	STZ2G — Pre-index	FEAT_MTE
11	000000000	00	LDGM	FEAT_MTE2
!= 01	!= 000000000	00	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Load/store exclusive pair

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0	0 1	0	0	0 0	L	1	Rs		o0	Rt2		Rn		Rt	

sz	L	o0	Instruction Details
0	0	0	STXP — 32-bit
0	0	1	STLXP — 32-bit
0	1	0	LDXP — 32-bit
0	1	1	LDAXP — 32-bit
1	0	0	STXP — 64-bit
1	0	1	STLXP — 64-bit
1	1	0	LDXP — 64-bit
1	1	1	LDAXP — 64-bit

Load/store exclusive register (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0	0 1	0	0	1 0	L	0	Rs		o0	Rt2		Rn		Rt	

sz	L	o0	Instruction Details	Condition
0	0	0	STTXR — 32-bit	FEAT_LSUI
0	0	1	STLTXR — 32-bit	FEAT_LSUI
0	1	0	LDTXR — 32-bit	FEAT_LSUI
0	1	1	LDATXR — 32-bit	FEAT_LSUI
1	0	0	STTXR — 64-bit	FEAT_LSUI
1	0	1	STLTXR — 64-bit	FEAT_LSUI
1	1	0	LDTXR — 64-bit	FEAT_LSUI
1	1	1	LDATXR — 64-bit	FEAT_LSUI

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Compare and swap (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31	30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0 0	1	0	0 1	1	L	0	Rs		o0	Rt2		Rn		Rt	

sz	L	o0	Rt2	Instruction Details	Condition
0	x	x	xxxxx	UNALLOCATED	-
1	x	x	!= 11111	UNALLOCATED	-
1	0	0	11111	CAST, CASAT, CASALT, CASLT — CAST	FEAT_LSUI
1	0	1	11111	CAST, CASAT, CASALT, CASLT — CASLT	FEAT_LSUI
1	1	0	11111	CAST, CASAT, CASALT, CASLT — CASAT	FEAT_LSUI
1	1	1	11111	CAST, CASAT, CASALT, CASLT — CASALT	FEAT_LSUI

Load/store exclusive register

The encodings in this section are decoded from Loads and Stores.

31 30	29	27	26	25	23	22	21	20	16	15	14	10	9	5	4	0
size	0 0	1	0	0 0	0	L	0	Rs		o0	Rt2		Rn		Rt	

size	L	o0	Instruction Details
00	0	0	STXRB
00	0	1	STLXRB
00	1	0	LDXRB
00	1	1	LDAXRB
01	0	0	STXRH
01	0	1	STLXRH
01	1	0	LDXRH
01	1	1	LDAXRH
10	0	0	STXR — 32-bit
10	0	1	STLXR — 32-bit
10	1	0	LDXR — 32-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	L	o0	Instruction Details
10	1	1	LDAXR — 32-bit
11	0	0	STXR — 64-bit
11	0	1	STLXR — 64-bit
11	1	0	LDXR — 64-bit
11	1	1	LDAXR — 64-bit

Load/store ordered

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15	14 10	9 5	4 0
size	0 0 1	0	0 0 1	L	0	Rs	o0	Rt2	Rn	Rt

size	L	o0	Instruction Details	Condition
00	0	0	STLLRB	FEAT_LOR
00	0	1	STLRB	-
00	1	0	LDLARB	FEAT_LOR
00	1	1	LDARB	-
01	0	0	STLLRH	FEAT_LOR
01	0	1	STLRH	-
01	1	0	LDLARH	FEAT_LOR
01	1	1	LDARH	-
10	0	0	STLLR — 32-bit	FEAT_LOR
10	0	1	STLR — 32-bit	-
10	1	0	LDLAR — 32-bit	FEAT_LOR
10	1	1	LDAR — 32-bit	-
11	0	0	STLLR — 64-bit	FEAT_LOR
11	0	1	STLR — 64-bit	-
11	1	0	LDLAR — 64-bit	FEAT_LOR
11	1	1	LDAR — 64-bit	-

Compare and swap

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15	14 10	9 5	4 0
size	0 0 1	0	0 0 1	L	1	Rs	o0	Rt2	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	L	o0	Rt2	Instruction Details	Condition
xx	x	x	!= 11111	UNALLOCATED	-
00	0	0	11111	CASB, CASAB, CASALB, CASLB — CASB	FEAT_LSE
00	0	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	FEAT_LSE
00	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	FEAT_LSE
00	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	FEAT_LSE
01	0	0	11111	CASH, CASAH, CASALH, CASLH — CASH	FEAT_LSE
01	0	1	11111	CASH, CASAH, CASALH, CASLH — CASLH	FEAT_LSE
01	1	0	11111	CASH, CASAH, CASALH, CASLH — CASAH	FEAT_LSE
01	1	1	11111	CASH, CASAH, CASALH, CASLH — CASALH	FEAT_LSE
10	0	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	FEAT_LSE
10	0	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	FEAT_LSE
10	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CASA	FEAT_LSE
10	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASAL	FEAT_LSE
11	0	0	11111	CAS, CASA, CASAL, CASL — 64-bit CAS	FEAT_LSE
11	0	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASL	FEAT_LSE
11	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CASA	FEAT_LSE
11	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASAL	FEAT_LSE

Load/store ordered register pair

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21	20 16	15 12	11 10	9 5	4 0
size	0 1 1	0	0 1 0	L	0	Rt2	opc2	1 0	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	L	opc2	Instruction Details	Condition
0x	x	xxxx	UNALLOCATED	-
1x	x	!= 000x	UNALLOCATED	-
10	0	0000	STILP — 32-bit pre-index	FEAT_LRCPC3
10	0	0001	STILP — 32-bit	FEAT_LRCPC3
10	1	0000	LDIAPP — 32-bit post-index	FEAT_LRCPC3
10	1	0001	LDIAPP — 32-bit	FEAT_LRCPC3
11	0	0000	STILP — 64-bit pre-index	FEAT_LRCPC3
11	0	0001	STILP — 64-bit	FEAT_LRCPC3
11	1	0000	LDIAPP — 64-bit post-index	FEAT_LRCPC3
11	1	0001	LDIAPP — 64-bit	FEAT_LRCPC3

Load/store ordered (writeback)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 10	9 5	4 0
size	0 1 1	0	0 1 1	L	0 0 0 0 0 0 0 0 0 0 1 0	Rn	Rt

size	L	Instruction Details	Condition
0x	x	UNALLOCATED	-
10	0	STLR — 32-bit	FEAT_LRCPC3
10	1	LDAPR — 32-bit	FEAT_LRCPC3
11	0	STLR — 64-bit	FEAT_LRCPC3
11	1	LDAPR — 64-bit	FEAT_LRCPC3

Load/store ordered (unscaled immediate)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	0 1 1	0	0 1	opc	0	imm9	0 0	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Condition
00	00	STLURB	FEAT_LRCPC2
00	01	LDAPURB	FEAT_LRCPC2
00	10	LDAPURSB — 64-bit	FEAT_LRCPC2
00	11	LDAPURSB — 32-bit	FEAT_LRCPC2
01	00	STLURH	FEAT_LRCPC2
01	01	LDAPURH	FEAT_LRCPC2
01	10	LDAPURSH — 64-bit	FEAT_LRCPC2
01	11	LDAPURSH — 32-bit	FEAT_LRCPC2
10	00	STLUR — 32-bit	FEAT_LRCPC2
10	01	LDAPUR — 32-bit	FEAT_LRCPC2
10	10	LDAPURSW	FEAT_LRCPC2
10	11	UNALLOCATED	-
11	00	STLUR — 64-bit	FEAT_LRCPC2
11	01	LDAPUR — 64-bit	FEAT_LRCPC2
11	1x	UNALLOCATED	-

Load/store ordered (SIMD&FP)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	0 1 1	1	0 1	opc	0	imm9	1 0	Rn	Rt

size	opc	Instruction Details	Condition
00	00	STLUR (SIMD&FP) — 8-bit	FEAT_FP && FEAT_LRCPC3
00	01	LDAPUR (SIMD&FP) — 8-bit	FEAT_FP && FEAT_LRCPC3
00	10	STLUR (SIMD&FP) — 128-bit	FEAT_FP && FEAT_LRCPC3
00	11	LDAPUR (SIMD&FP) — 128-bit	FEAT_FP && FEAT_LRCPC3
01	00	STLUR (SIMD&FP) — 16-bit	FEAT_FP && FEAT_LRCPC3
01	01	LDAPUR (SIMD&FP) — 16-bit	FEAT_FP && FEAT_LRCPC3
10	00	STLUR (SIMD&FP) — 32-bit	FEAT_FP && FEAT_LRCPC3
10	01	LDAPUR (SIMD&FP) — 32-bit	FEAT_FP && FEAT_LRCPC3
11	00	STLUR (SIMD&FP) — 64-bit	FEAT_FP && FEAT_LRCPC3
11	01	LDAPUR (SIMD&FP) — 64-bit	FEAT_FP && FEAT_LRCPC3
!= 00	1x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Load register (literal)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 5	4 0
opc	0 1 1	VR	0 0	imm19	Rt

opc	VR	Instruction Details	Condition
00	0	LDR (literal) — 32-bit	-
00	1	LDR (literal, SIMD&FP) — 32-bit	FEAT_FP
01	0	LDR (literal) — 64-bit	-
01	1	LDR (literal, SIMD&FP) — 64-bit	FEAT_FP
10	0	LDRSW (literal)	-
10	1	LDR (literal, SIMD&FP) — 128-bit	FEAT_FP
11	0	PRFM (literal)	-
11	1	UNALLOCATED	-

Memory Copy and Memory Set

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
size	0 1 1	o0	0 1	op1	0	Rs	op2	0 1	Rn	Rd

o0	op1	op2	Instruction Details	Condition
x	11	11xx	UNALLOCATED	-
0	00	0000	CPYFP, CPYFM, CPYFE — Prologue	FEAT_MOPS
0	00	0001	CPYFPWT, CPYFMWT, CPYFEWT — Prologue	FEAT_MOPS
0	00	0010	CPYFPRT, CPYFMRT, CPYFERT — Prologue	FEAT_MOPS
0	00	0011	CPYFPT, CPYFMT, CPYFET — Prologue	FEAT_MOPS
0	00	0100	CPYFPWN, CPYFMWN, CPYFEWN — Prologue	FEAT_MOPS
0	00	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — Prologue	FEAT_MOPS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Condition
0	00	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — Prologue	FEAT_MOPS
0	00	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — Prologue	FEAT_MOPS
0	00	1000	CPYFPRN, CPYFMRN, CPYFERN — Prologue	FEAT_MOPS
0	00	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — Prologue	FEAT_MOPS
0	00	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — Prologue	FEAT_MOPS
0	00	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — Prologue	FEAT_MOPS
0	00	1100	CPYFPN, CPYFMN, CPYFEN — Prologue	FEAT_MOPS
0	00	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — Prologue	FEAT_MOPS
0	00	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — Prologue	FEAT_MOPS
0	00	1111	CPYFPTN, CPYFMTN, CPYFETN — Prologue	FEAT_MOPS
0	01	0000	CPYFP, CPYFM, CPYFE — Main	FEAT_MOPS
0	01	0001	CPYFPWT, CPYFMWT, CPYFEWT — Main	FEAT_MOPS
0	01	0010	CPYFPRT, CPYFMRT, CPYFERT — Main	FEAT_MOPS
0	01	0011	CPYFPT, CPYFMT, CPYFET — Main	FEAT_MOPS
0	01	0100	CPYFPWN, CPYFMWN, CPYFEWN — Main	FEAT_MOPS
0	01	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — Main	FEAT_MOPS
0	01	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — Main	FEAT_MOPS
0	01	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — Main	FEAT_MOPS
0	01	1000	CPYFPRN, CPYFMRN, CPYFERN — Main	FEAT_MOPS
0	01	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — Main	FEAT_MOPS
0	01	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — Main	FEAT_MOPS
0	01	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — Main	FEAT_MOPS
0	01	1100	CPYFPN, CPYFMN, CPYFEN — Main	FEAT_MOPS
0	01	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — Main	FEAT_MOPS
0	01	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — Main	FEAT_MOPS
0	01	1111	CPYFPTN, CPYFMTN, CPYFETN — Main	FEAT_MOPS
0	10	0000	CPYFP, CPYFM, CPYFE — Epilogue	FEAT_MOPS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Condition
0	10	0001	CPYFPWT, CPYFMWT, CPYFEWT — Epilogue	FEAT_MOPS
0	10	0010	CPYFPRT, CPYFMRT, CPYFERT — Epilogue	FEAT_MOPS
0	10	0011	CPYFPT, CPYFMT, CPYFET — Epilogue	FEAT_MOPS
0	10	0100	CPYFPWN, CPYFMWN, CPYFEWN — Epilogue	FEAT_MOPS
0	10	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — Epilogue	FEAT_MOPS
0	10	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — Epilogue	FEAT_MOPS
0	10	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — Epilogue	FEAT_MOPS
0	10	1000	CPYFPRN, CPYFMRN, CPYFERN — Epilogue	FEAT_MOPS
0	10	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — Epilogue	FEAT_MOPS
0	10	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — Epilogue	FEAT_MOPS
0	10	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — Epilogue	FEAT_MOPS
0	10	1100	CPYFPN, CPYFMN, CPYFEN — Epilogue	FEAT_MOPS
0	10	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — Epilogue	FEAT_MOPS
0	10	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — Epilogue	FEAT_MOPS
0	10	1111	CPYFPTN, CPYFMTN, CPYFETN — Epilogue	FEAT_MOPS
0	11	0000	SETP, SETM, SETE — Prologue	FEAT_MOPS
0	11	0001	SETPT, SETMT, SETET — Prologue	FEAT_MOPS
0	11	0010	SETPN, SETMN, SETEN — Prologue	FEAT_MOPS
0	11	0011	SETPTN, SETMTN, SETETN — Prologue	FEAT_MOPS
0	11	0100	SETP, SETM, SETE — Main	FEAT_MOPS
0	11	0101	SETPT, SETMT, SETET — Main	FEAT_MOPS
0	11	0110	SETPN, SETMN, SETEN — Main	FEAT_MOPS
0	11	0111	SETPTN, SETMTN, SETETN — Main	FEAT_MOPS
0	11	1000	SETP, SETM, SETE — Epilogue	FEAT_MOPS
0	11	1001	SETPT, SETMT, SETET — Epilogue	FEAT_MOPS
0	11	1010	SETPN, SETMN, SETEN — Epilogue	FEAT_MOPS
0	11	1011	SETPTN, SETMTN, SETETN — Epilogue	FEAT_MOPS
1	00	0000	CPYP, CPYM, CPYE — Prologue	FEAT_MOPS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Condition
1	00	0001	CPYPWT, CPYMWT, CPYEWT — Prologue	FEAT_MOPS
1	00	0010	CPYPRT, CPYMRT, CPYERT — Prologue	FEAT_MOPS
1	00	0011	CPYPT, CPYMT, CPYET — Prologue	FEAT_MOPS
1	00	0100	CPYPWN, CPYMWN, CPYEWN — Prologue	FEAT_MOPS
1	00	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — Prologue	FEAT_MOPS
1	00	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — Prologue	FEAT_MOPS
1	00	0111	CPYPTWN, CPYMTWN, CPYETWN — Prologue	FEAT_MOPS
1	00	1000	CPYPRN, CPYMRN, CPYERN — Prologue	FEAT_MOPS
1	00	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — Prologue	FEAT_MOPS
1	00	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — Prologue	FEAT_MOPS
1	00	1011	CPYPTRN, CPYMTRN, CPYETRN — Prologue	FEAT_MOPS
1	00	1100	CPYPN, CPYMN, CPYEN — Prologue	FEAT_MOPS
1	00	1101	CPYPWTN, CPYMWTN, CPYEWTN — Prologue	FEAT_MOPS
1	00	1110	CPYPRTN, CPYMRTN, CPYERTN — Prologue	FEAT_MOPS
1	00	1111	CPYPTN, CPYMTN, CPYETN — Prologue	FEAT_MOPS
1	01	0000	CPYP, CPYM, CPYE — Main	FEAT_MOPS
1	01	0001	CPYPWT, CPYMWT, CPYEWT — Main	FEAT_MOPS
1	01	0010	CPYPRT, CPYMRT, CPYERT — Main	FEAT_MOPS
1	01	0011	CPYPT, CPYMT, CPYET — Main	FEAT_MOPS
1	01	0100	CPYPWN, CPYMWN, CPYEWN — Main	FEAT_MOPS
1	01	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — Main	FEAT_MOPS
1	01	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — Main	FEAT_MOPS
1	01	0111	CPYPTWN, CPYMTWN, CPYETWN — Main	FEAT_MOPS
1	01	1000	CPYPRN, CPYMRN, CPYERN — Main	FEAT_MOPS
1	01	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — Main	FEAT_MOPS
1	01	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — Main	FEAT_MOPS
1	01	1011	CPYPTRN, CPYMTRN, CPYETRN — Main	FEAT_MOPS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Condition
1	01	1100	CPYPN, CPYMN, CPYEN — Main	FEAT_MOPS
1	01	1101	CPYPWTN, CPYMWTN, CPYEWTN — Main	FEAT_MOPS
1	01	1110	CPYPRTN, CPYMRTN, CPYERTN — Main	FEAT_MOPS
1	01	1111	CPYPTN, CPYMTN, CPYETN — Main	FEAT_MOPS
1	10	0000	CPYP, CPYM, CPYE — Epilogue	FEAT_MOPS
1	10	0001	CPYPWT, CPYMWT, CPYEWT — Epilogue	FEAT_MOPS
1	10	0010	CPYPRT, CPYMRT, CPYERT — Epilogue	FEAT_MOPS
1	10	0011	CPYPT, CPYMT, CPYET — Epilogue	FEAT_MOPS
1	10	0100	CPYPWN, CPYMWN, CPYEWN — Epilogue	FEAT_MOPS
1	10	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — Epilogue	FEAT_MOPS
1	10	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — Epilogue	FEAT_MOPS
1	10	0111	CPYPTWN, CPYMTWN, CPYETWN — Epilogue	FEAT_MOPS
1	10	1000	CPYPRN, CPYMRN, CPYERN — Epilogue	FEAT_MOPS
1	10	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — Epilogue	FEAT_MOPS
1	10	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — Epilogue	FEAT_MOPS
1	10	1011	CPYPTRN, CPYMTRN, CPYETRN — Epilogue	FEAT_MOPS
1	10	1100	CPYPN, CPYMN, CPYEN — Epilogue	FEAT_MOPS
1	10	1101	CPYPWTN, CPYMWTN, CPYEWTN — Epilogue	FEAT_MOPS
1	10	1110	CPYPRTN, CPYMRTN, CPYERTN — Epilogue	FEAT_MOPS
1	10	1111	CPYPTN, CPYMTN, CPYETN — Epilogue	FEAT_MOPS
1	11	0000	SETGP, SETGM, SETGE — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0001	SETGPT, SETGMT, SETGET — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0010	SETGPN, SETGMN, SETGEN — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0011	SETGPTN, SETGMTN, SETGETN — Prologue	FEAT_MOPS && FEAT_MTE
1	11	0100	SETGP, SETGM, SETGE — Main	FEAT_MOPS && FEAT_MTE
1	11	0101	SETGPT, SETGMT, SETGET — Main	FEAT_MOPS && FEAT_MTE
1	11	0110	SETGPN, SETGMN, SETGEN — Main	FEAT_MOPS && FEAT_MTE
1	11	0111	SETGPTN, SETGMTN, SETGETN — Main	FEAT_MOPS && FEAT_MTE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

o0	op1	op2	Instruction Details	Condition
1	11	1000	SETGP, SETGM, SETGE — Epilogue	FEAT_MOPS && FEAT_MTE
1	11	1001	SETGPT, SETGMT, SETGET — Epilogue	FEAT_MOPS && FEAT_MTE
1	11	1010	SETGPN, SETGMN, SETGEN — Epilogue	FEAT_MOPS && FEAT_MTE
1	11	1011	SETGPTN, SETGMTN, SETGETN — Epilogue	FEAT_MOPS && FEAT_MTE

Load/store no-allocate pair (offset)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 0 0	L	imm7	Rt2	Rn	Rt

opc	VR	L	Instruction Details	Condition
00	0	0	STNP — 32-bit	-
00	0	1	LDNP — 32-bit	-
00	1	0	STNP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDNP (SIMD&FP) — 32-bit	FEAT_FP
01	0	x	UNALLOCATED	-
01	1	0	STNP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDNP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STNP — 64-bit	-
10	0	1	LDNP — 64-bit	-
10	1	0	STNP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDNP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTNP	FEAT_LSUI
11	0	1	LDTNP	FEAT_LSUI
11	1	0	STTNP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTNP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register pair (post-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 0 1	L	imm7	Rt2	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	VR	L	Instruction Details	Condition
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTP	FEAT_LSUI
11	0	1	LDTP	FEAT_LSUI
11	1	0	STTP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register pair (offset)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 1 0	L	imm7	Rt2	Rn	Rt

opc	VR	L	Instruction Details	Condition
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	VR	L	Instruction Details	Condition
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTP	FEAT_LSUI
11	0	1	LDTP	FEAT_LSUI
11	1	0	STTP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register pair (pre-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	VR	0 1 1	L	imm7	Rt2	Rn	Rt

opc	VR	L	Instruction Details	Condition
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11	0	0	STTP	FEAT_LSUI
11	0	1	LDTP	FEAT_LSUI
11	1	0	STTP (SIMD&FP)	FEAT_FP && FEAT_LSUI
11	1	1	LDTP (SIMD&FP)	FEAT_FP && FEAT_LSUI

Load/store register (unscaled immediate)

The encodings in this section are decoded from Loads and Stores.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	0 0	Rn	Rt

size	VR	opc	Instruction Details	Condition
00	0	00	STURB	-
00	0	01	LDURB	-
00	0	10	LDURSB — 64-bit	-
00	0	11	LDURSB — 32-bit	-
00	1	00	STUR (SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDUR (SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STUR (SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDUR (SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STURH	-
01	0	01	LDURH	-
01	0	10	LDURSH — 64-bit	-
01	0	11	LDURSH — 32-bit	-
01	1	00	STUR (SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDUR (SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	UNALLOCATED	-
10	0	00	STUR — 32-bit	-
10	0	01	LDUR — 32-bit	-
10	0	10	LDURSW	-
10	1	00	STUR (SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDUR (SIMD&FP) — 32-bit	FEAT_FP
11	0	00	STUR — 64-bit	-
11	0	01	LDUR — 64-bit	-
11	0	10	PRFUM	-
11	1	00	STUR (SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDUR (SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	UNALLOCATED	-

Load/store register (immediate post-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	0 1	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Condition
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	1x	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	0	11	UNALLOCATED	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	1x	UNALLOCATED	-
11	x	1x	UNALLOCATED	-
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP

Load/store register (unprivileged)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	1 0	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details
xx	1	xx	UNALLOCATED
00	0	00	STTRB
00	0	01	LDTRB
00	0	10	LDTRSB — 64-bit
00	0	11	LDTRSB — 32-bit
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH — 64-bit
01	0	11	LDTRSH — 32-bit
10	0	00	STTR — 32-bit
10	0	01	LDTR — 32-bit
10	0	10	LDTRSW
10	0	11	UNALLOCATED
11	0	00	STTR — 64-bit
11	0	01	LDTR — 64-bit
11	0	1x	UNALLOCATED

Load/store register (immediate pre-indexed)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	0	imm9	1 1	Rn	Rt

size	VR	opc	Instruction Details	Condition
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Condition
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	1x	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	0	11	UNALLOCATED	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	1x	UNALLOCATED	-
11	x	1x	UNALLOCATED	-
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP

Atomic memory operations

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
size	1 1 1	VR	0 0	A	R	1	Rs	o3	opc	0 0	Rn	Rt

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
xx	0	0	x	xxxxx	1	100	xxxxx	UNALLOCATED	-
xx	0	0	x	xxxxx	1	11x	xxxxx	UNALLOCATED	-
xx	0	1	1	xxxxx	1	100	xxxxx	UNALLOCATED	-
xx	1	x	x	xxxxx	0	001	xxxxx	UNALLOCATED	-
xx	1	x	x	xxxxx	0	01x	xxxxx	UNALLOCATED	-
xx	1	0	x	xxxxx	1	xxx	!= 11111	UNALLOCATED	-
xx	1	0	x	xxxxx	1	001	11111	UNALLOCATED	-
xx	1	0	x	xxxxx	1	01x	11111	UNALLOCATED	-
xx	1	1	x	xxxxx	1	xxx	xxxxx	UNALLOCATED	-
0x	0	x	0	xxxxx	1	101	xxxxx	UNALLOCATED	-
0x	0	0	1	xxxxx	1	101	xxxxx	UNALLOCATED	-
0x	0	1	x	xxxxx	1	11x	xxxxx	UNALLOCATED	-
0x	0	1	1	xxxxx	1	101	xxxxx	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
00	0	0	0	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — No memory ordering	FEAT_LSE
00	0	0	0	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPB	FEAT_LSE
00	0	0	0	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLR	FEAT_THE
00	0	0	0	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWP	FEAT_THE
00	0	0	0	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSET	FEAT_THE
00	0	0	1	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Release	FEAT_LSE
00	0	0	1	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Release	FEAT_LSE
00	0	0	1	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPLB	FEAT_LSE
00	0	0	1	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLRL	FEAT_THE
00	0	0	1	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWPL	FEAT_THE
00	0	0	1	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSETL	FEAT_THE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
00	0	1	0	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Acquire	FEAT_LSE
00	0	1	0	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPAB	FEAT_LSE
00	0	1	0	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLRA	FEAT_THE
00	0	1	0	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWPA	FEAT_THE
00	0	1	0	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSETA	FEAT_THE
00	0	1	0	xxxxx	1	100	xxxxx	LDAPRB	FEAT_LRCPC
00	0	1	1	xxxxx	0	000	xxxxx	LDADDB, LDADDAB, LDADDALB, LDADDLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	001	xxxxx	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	010	xxxxx	LDEORB, LDEORAB, LDEORALB, LDEORLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	011	xxxxx	LDSETB, LDSETAB, LDSETALB, LDSETLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	100	xxxxx	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	101	xxxxx	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	110	xxxxx	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	0	111	xxxxx	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Acquire-release	FEAT_LSE
00	0	1	1	xxxxx	1	000	xxxxx	SWPB, SWPAB, SWPALB, SWPLB — SWPALB	FEAT_LSE
00	0	1	1	xxxxx	1	001	xxxxx	RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL — RCWCLRAL	FEAT_THE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
00	0	1	1	xxxxx	1	010	xxxxx	RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL — RCWSWPAL	FEAT_THE
00	0	1	1	xxxxx	1	011	xxxxx	RCWSET, RCWSETA, RCWSETAL, RCWSETL — RCWSETAL	FEAT_THE
00	1	0	0	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	000	11111	STBFADD, STBFADDL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	100	11111	STBFMAX, STBFMAXL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	101	11111	STBFMIN, STBFMINL — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	110	11111	STBFMAXNM, STBFMAXNML — No memory ordering	FEAT_LSFE
00	1	0	0	xxxxx	1	111	11111	STBFMINNM, STBFMINNML — No memory ordering	FEAT_LSFE
00	1	0	1	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — Release	FEAT_LSFE
00	1	0	1	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	000	11111	STBFADD, STBFADDL — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	100	11111	STBFMAX, STBFMAXL — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	101	11111	STBFMIN, STBFMINL — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	110	11111	STBFMAXNM, STBFMAXNML — Release	FEAT_LSFE
00	1	0	1	xxxxx	1	111	11111	STBFMINNM, STBFMINNML — Release	FEAT_LSFE
00	1	1	0	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — Acquire	FEAT_LSFE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
00	1	1	0	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — Acquire	FEAT_LSFE
00	1	1	0	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — Acquire	FEAT_LSFE
00	1	1	0	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — Acquire	FEAT_LSFE
00	1	1	0	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — Acquire	FEAT_LSFE
00	1	1	1	xxxxx	0	000	xxxxx	LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	100	xxxxx	LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	101	xxxxx	LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	110	xxxxx	LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML — Acquire-release	FEAT_LSFE
00	1	1	1	xxxxx	0	111	xxxxx	LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML — Acquire-release	FEAT_LSFE
01	0	0	0	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — No memory ordering	FEAT_LSE
01	0	0	0	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPH	FEAT_LSE
01	0	0	0	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLR	FEAT_THE
01	0	0	0	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWP	FEAT_THE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
01	0	0	0	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSET	FEAT_THE
01	0	0	1	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Release	FEAT_LSE
01	0	0	1	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Release	FEAT_LSE
01	0	0	1	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPLH	FEAT_LSE
01	0	0	1	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLRL	FEAT_THE
01	0	0	1	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWPL	FEAT_THE
01	0	0	1	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSETL	FEAT_THE
01	0	1	0	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Acquire	FEAT_LSE
01	0	1	0	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPAH	FEAT_LSE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
01	0	1	0	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLRA	FEAT_THE
01	0	1	0	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWPA	FEAT_THE
01	0	1	0	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSETA	FEAT_THE
01	0	1	0	xxxxx	1	100	xxxxx	LDAPRH	FEAT_LRCPC
01	0	1	1	xxxxx	0	000	xxxxx	LDADDH, LDADDAH, LDADDALH, LDADDLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	001	xxxxx	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	010	xxxxx	LDEORH, LDEORAH, LDEORALH, LDEORLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	011	xxxxx	LDSETH, LDSETAH, LDSETALH, LDSETLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	100	xxxxx	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	101	xxxxx	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	110	xxxxx	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	0	111	xxxxx	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Acquire-release	FEAT_LSE
01	0	1	1	xxxxx	1	000	xxxxx	SWPH, SWPAH, SWPALH, SWPLH — SWPALH	FEAT_LSE
01	0	1	1	xxxxx	1	001	xxxxx	RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL — RCWSCLRAL	FEAT_THE
01	0	1	1	xxxxx	1	010	xxxxx	RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL — RCWSSWPAL	FEAT_THE
01	0	1	1	xxxxx	1	011	xxxxx	RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL — RCWSSETAL	FEAT_THE
01	1	0	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision no memory ordering	FEAT_LSFE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
01	1	0	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	000	11111	STFADD, STFADDL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	100	11111	STFMAX, STFMAXL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	101	11111	STFMIN, STFMINL — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Half-precision no memory ordering	FEAT_LSFE
01	1	0	0	xxxxx	1	111	11111	STFMINNM, STFMINNML — Half-precision no memory ordering	FEAT_LSFE
01	1	0	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	000	11111	STFADD, STFADDL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	100	11111	STFMAX, STFMAXL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	101	11111	STFMIN, STFMINL — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Half-precision release	FEAT_LSFE
01	1	0	1	xxxxx	1	111	11111	STFMINNM, STFMINNML — Half-precision release	FEAT_LSFE
01	1	1	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision acquire	FEAT_LSFE
01	1	1	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision acquire	FEAT_LSFE
01	1	1	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Half-precision acquire-release	FEAT_LSFE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
01	1	1	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Half-precision acquire-release	FEAT_LSFE
01	1	1	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Half-precision acquire-release	FEAT_LSFE
01	1	1	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Half-precision acquire-release	FEAT_LSFE
01	1	1	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Half-precision acquire-release	FEAT_LSFE
1x	0	1	x	xxxxx	1	!= x00	xxxxx	UNALLOCATED	-
10	0	0	x	xxxxx	1	x01	xxxxx	UNALLOCATED	-
10	0	0	x	xxxxx	1	010	xxxxx	UNALLOCATED	-
10	0	0	x	xxxxx	1	011	xxxxx	UNALLOCATED	-
10	0	0	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWP	FEAT_LSE
10	0	0	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit release	FEAT_LSE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
10	0	0	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit release	FEAT_LSE
10	0	0	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWPL	FEAT_LSE
10	0	1	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit acquire	FEAT_LSE
10	0	1	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWPA	FEAT_LSE
10	0	1	0	xxxxx	1	100	xxxxx	LDAPR — 32-bit	FEAT_LRCPC
10	0	1	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL	FEAT_LSE
10	1	0	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision no memory ordering	FEAT_LSFE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
10	1	0	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	000	11111	STFADD, STFADDL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	100	11111	STFMAX, STFMAXL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	101	11111	STFMIN, STFMINL — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	0	xxxxx	1	111	11111	STFMINNM, STFMINNML — Single-precision no memory ordering	FEAT_LSFE
10	1	0	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	000	11111	STFADD, STFADDL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	100	11111	STFMAX, STFMAXL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	101	11111	STFMIN, STFMINL — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Single-precision release	FEAT_LSFE
10	1	0	1	xxxxx	1	111	11111	STFMINNM, STFMINNML — Single-precision release	FEAT_LSFE
10	1	1	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision acquire	FEAT_LSFE
10	1	1	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision acquire	FEAT_LSFE
10	1	1	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision acquire	FEAT_LSFE
10	1	1	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision acquire	FEAT_LSFE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
10	1	1	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision acquire	FEAT_LSFE
10	1	1	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Single-precision acquire-release	FEAT_LSFE
10	1	1	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Single-precision acquire-release	FEAT_LSFE
11	0	0	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWP	FEAT_LSE
11	0	0	0	xxxxx	1	010	xxxxx	ST64BV0	FEAT_LS64_ACCDATA
11	0	0	0	xxxxx	1	011	xxxxx	ST64BV	FEAT_LS64_V
11	0	0	0	11111	1	001	xxxxx	ST64B	FEAT_LS64
11	0	0	0	11111	1	101	xxxxx	LD64B	FEAT_LS64
11	0	0	0	!= 11111	1	x01	xxxxx	UNALLOCATED	-
11	0	0	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit release	FEAT_LSE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
11	0	0	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit release	FEAT_LSE
11	0	0	1	xxxxx	1	x01	xxxxx	UNALLOCATED	-
11	0	0	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWPL	FEAT_LSE
11	0	0	1	xxxxx	1	010	xxxxx	UNALLOCATED	-
11	0	0	1	xxxxx	1	011	xxxxx	UNALLOCATED	-
11	0	1	0	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit acquire	FEAT_LSE
11	0	1	0	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWPA	FEAT_LSE
11	0	1	0	xxxxx	1	100	xxxxx	LDAPR — 64-bit	FEAT_LRCPC
11	0	1	1	xxxxx	0	000	xxxxx	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	001	xxxxx	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	010	xxxxx	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	011	xxxxx	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	100	xxxxx	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	101	xxxxx	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	0	110	xxxxx	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit acquire-release	FEAT_LSE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
11	0	1	1	xxxxx	0	111	xxxxx	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	xxxxx	1	000	xxxxx	SWP, SWPA, SWPAL, SWPL — 64-bit SWPAL	FEAT_LSE
11	1	0	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	000	11111	STFADD, STFADDL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	100	11111	STFMAX, STFMAXL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	101	11111	STFMIN, STFMINL — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	0	xxxxx	1	111	11111	STFMINNM, STFMINNML — Double-precision no memory ordering	FEAT_LSFE
11	1	0	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	000	11111	STFADD, STFADDL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	100	11111	STFMAX, STFMAXL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	101	11111	STFMIN, STFMINL — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	110	11111	STFMAXNM, STFMAXNML — Double-precision release	FEAT_LSFE
11	1	0	1	xxxxx	1	111	11111	STFMINNM, STFMINNML — Double-precision release	FEAT_LSFE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	Rs	o3	opc	Rt	Instruction Details	Condition
11	1	1	0	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision acquire	FEAT_LSFE
11	1	1	0	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision acquire	FEAT_LSFE
11	1	1	1	xxxxx	0	000	xxxxx	LDFADD, LDFADDA, LDFADDAL, LDFADDL — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	100	xxxxx	LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	101	xxxxx	LDFMIN, LDFMINA, LDFMINAL, LDFMINL — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	110	xxxxx	LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML — Double-precision acquire-release	FEAT_LSFE
11	1	1	1	xxxxx	0	111	xxxxx	LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML — Double-precision acquire-release	FEAT_LSFE

Load/store register (register offset)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1 1	VR	0 0	opc	1	Rm	option	S	1 0	Rn	Rt

size	VR	opc	option	Rt	Instruction Details	Condition
00	0	00	011	xxxxx	STRB (register) — Shifted register	-
00	0	00	!= 011	xxxxx	STRB (register) — Extended register	-
00	0	01	011	xxxxx	LDRB (register) — Shifted register	-
00	0	01	!= 011	xxxxx	LDRB (register) — Extended register	-
00	0	10	011	xxxxx	LDRSB (register) — 64-bit with shifted register offset	-
00	0	10	!= 011	xxxxx	LDRSB (register) — 64-bit with extended register offset	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	option	Rt	Instruction Details	Condition
00	0	11	011	xxxxx	LDRSB (register) — 32-bit with shifted register offset	-
00	0	11	!= 011	xxxxx	LDRSB (register) — 32-bit with extended register offset	-
00	1	00	011	xxxxx	STR (register, SIMD&FP) — 8-bit with shifted register offset	FEAT_FP
00	1	00	!= 011	xxxxx	STR (register, SIMD&FP) — 8-bit with extended register offset	FEAT_FP
00	1	01	011	xxxxx	LDR (register, SIMD&FP) — 8-bit with shifted register offset	FEAT_FP
00	1	01	!= 011	xxxxx	LDR (register, SIMD&FP) — 8-bit with extended register offset	FEAT_FP
00	1	10	xxx	xxxxx	STR (register, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	xxx	xxxxx	LDR (register, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	xxx	xxxxx	STRH (register)	-
01	0	01	xxx	xxxxx	LDRH (register)	-
01	0	10	xxx	xxxxx	LDRSH (register) — 64-bit	-
01	0	11	xxx	xxxxx	LDRSH (register) — 32-bit	-
01	1	00	xxx	xxxxx	STR (register, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	xxx	xxxxx	LDR (register, SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	xxx	xxxxx	UNALLOCATED	-
10	0	00	xxx	xxxxx	STR (register) — 32-bit	-
10	0	01	xxx	xxxxx	LDR (register) — 32-bit	-
10	0	10	xxx	xxxxx	LDRSW (register)	-
10	1	00	xxx	xxxxx	STR (register, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	xxx	xxxxx	LDR (register, SIMD&FP) — 32-bit	FEAT_FP
11	0	00	xxx	xxxxx	STR (register) — 64-bit	-
11	0	01	xxx	xxxxx	LDR (register) — 64-bit	-
11	0	10	x0x	xxxxx	UNALLOCATED	-
11	0	10	x1x	11xxx	RPRFM	FEAT_RPRFM
11	0	10	x1x	!= 11xxx	PRFM (register)	-
11	1	00	xxx	xxxxx	STR (register, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	xxx	xxxxx	LDR (register, SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	xxx	xxxxx	UNALLOCATED	-

Load/store register (pac)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23	22	21	20 12	11	10	9 5	4 0
size	1 1 1	VR	0 0	M	S	1	imm9	W	1	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	M	W	Instruction Details	Condition
11	0	0	0	LDRAA, LDRAB — Key A, offset	FEAT_PAuth
11	0	0	1	LDRAA, LDRAB — Key A, pre-indexed	FEAT_PAuth
11	0	1	0	LDRAA, LDRAB — Key B, offset	FEAT_PAuth
11	0	1	1	LDRAA, LDRAB — Key B, pre-indexed	FEAT_PAuth
11	1	x	x	UNALLOCATED	-
!=11	x	x	x	UNALLOCATED	-

Load/store register (unsigned immediate)

The encodings in this section are decoded from Loads and Stores.

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
size	1 1 1	VR	0 1	opc	imm12	Rn	Rt

size	VR	opc	Instruction Details	Condition
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Condition
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	0	10	PRFM (immediate)	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	UNALLOCATED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C5
The A64 System Instruction Class

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following sections:

- The System instruction class encoding space
- Special-purpose registers
- A64 System instructions for cache maintenance
- A64 System instructions for address translation
- A64 System instructions for TLB maintenance
- A64 System instructions for prediction restriction
- A64 System instructions for the Branch Record Buffer Extension
- A64 System instructions for the Trace Extension
- A64 System instructions for the Guarded Control Stack
- A64 System instructions introduced by FEAT_RME_GPC3
- A64 IMPLEMENTATION DEFINED System instructions

See General information about the A64 instruction descriptions for information about entries used in the instruction encoding descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.1 The System instruction class encoding space

Part of the A64 instruction encoding space is assigned to instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the general model for accessing this functionality.

Note

- See Fixed values in AArch64 instruction and System register descriptions for information about abbreviations used in the System instruction descriptions.
- In AArch32 state much of this functionality is provided through the System register interface described in The AArch32 System register interface. In AArch64 state, the parameters used to characterize the System register encoding space are {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in Background to the System register interface. There is no particular significance to the naming of these parameters, and no functional distinction between the `opn` parameters and the `CRx` parameters.

Principles of the System instruction class encoding describes some general properties of these encodings. System instruction class encoding overview then describes the top-level encoding of these instructions, and the following sections then describe the next level of the encoding hierarchy of System instructions and Special-purpose registers:

- op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access.
- op0 == `0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, Guarded Control Stack instructions, and FEAT_RME_GPC3 instructions.
- op0 == `0b11`, Moves to and from Special-purpose registers.

For the description of the next level of encoding hierarchy of System registers, see:

- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.
- Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.1 Principles of the System instruction class encoding

An encoding in the System instruction space is identified by a set of arguments,{`op0`, `op1`, `CRn`, `CRm`, `op2`}. These form an encoding hierarchy, where:

`op0` Defines the top-level division of the encoding space, see System instruction class encoding overview.

`op1` Identifies the lowest Exception level at which the encoding is accessible, as follows:

Accessible at EL0 `op1` has the value `0b011`.

Accessible at EL1 `op1` has the value `0b000`, `0b001`, or `0b010`. The value is the same as the `op1` value used to access the equivalent AArch32 register.

Accessible at Secure EL1 `op1` has the value `0b111`.

Accessible at EL2 `op1` has the value `0b100` or `0b101`. The value `0b101` is used for the EL12 encodings that access EL1 System registers used when FEAT_VHE is implemented and the Effective value of HCR_EL2.E2H is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible at EL3 `op1` has the value `0b110`.

Arm strongly recommends that implementers adopt this use of `op1` when using the IMPLEMENTATION DEFINED regions of the encoding space described in Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.2 System instruction class encoding overview

The encoding of the System instruction class describes each instruction as being either:

- A transfer to a System register. This is a System instruction with the semantics of a write.
- A transfer from a System register. This is a System instruction with the semantics of a read.

A System instruction that initiates an operation operates as if it was making a transfer to a register.

In the AArch64 instruction set, the decode structure for the System instruction class is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	op0	op1	CRn	CRm	op2	Rt

The value of `L` indicates the transfer direction:

0 Transfer to System register.

1 Transfer from System register.

The `op0` field is the top-level encoding of the System instruction type. Its possible values are:

`0b00` These encodings provide:

- Instructions with an immediate field for accessing PSTATE, the current PE state.
- The architectural hint instructions.
- Barriers and the `CLREX` instruction.

For more information about these encodings, see op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access.

`0b01` These encodings provide:

- Cache maintenance instructions.
- TLB maintenance instructions.
- Address translation instructions.
- Prediction restriction instructions.
- BRBE instructions.
- Instrumentation Trace Extension instructions.
- Guarded Control Stack instructions.
- FEAT_RME_GPC3 instructions.

Note

These are equivalent to operations in the AArch32 (`coproc` == `0b1111`) encoding space.

For more information, see op0 == `0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, Guarded Control Stack instructions, and FEAT_RME_GPC3 instructions.

`0b10` These encodings provide moves to and from:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Legacy AArch32 System registers for execution environments, to provide access to these registers from higher Exception levels that are using AArch64.
- Debug, PMU, System PMU, and trace registers.

Note

These are equivalent to the registers in the AArch32 (`coproc` == `0b1110`) encoding space.

For more information, see Moves to and from debug and trace System registers.

`0b11` These encodings provide:

- Moves to and from Non-debug System registers. The accessed registers provide system control, and system status information.

  Note

  The accessed registers are equivalent to the registers in the AArch32 (`coproc` == `0b1111`) encoding space.

- Access to Special-purpose registers.

For more information, see Instructions for accessing Special-purpose registers and Instructions for accessing non-debug System registers.

C5.1.2.1 UNDEFINED behaviors

In the System register instruction encoding space, the following principles apply:

- All unallocated encodings are treated as UNDEFINED.
- All encodings with `L` == 1 and `op0` == `0b0x` are UNDEFINED, except for encodings in the area reserved for IMPLEMENTATION DEFINED use, see Reserved encoding space for IMPLEMENTATION DEFINED instructions.

For registers and operations that are accessible from a particular Exception level, any attempt to access those registers from a lower Exception level is UNDEFINED.

If a particular Exception level:

- Defines a register to be RO, then any attempt to write to that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L` == 0 is UNDEFINED.
- Defines a register to be WO, then any attempt to read from that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L` == 1 is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED, but see the recommendation in Principles of the System instruction class encoding.

C5.1.3 op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access

The different groups of System register instructions with `op0` == `0b00`:

- Are identified by the value of `CRn`.
- Are always encoded with a value of `0b11111` in the `Rt` field.

The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 16	15 … 12	11 … 8	7 … 5	4	0 … 0
1	1	0	1	0	1	0	1	0	0	L	0	0	op1	CRn	CRm	op2	1	1 1 1 1
op0																		Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encoding of the `CRn` field is as follows:

`0b0010` See Architectural hint instructions.

`0b0011` See Barriers and CLREX.

`0b0100` See Instructions for accessing the PSTATE fields.

C5.1.3.1 Architectural hint instructions

Within the `op0` == `0b00` encodings, the architectural hint instructions are identified by `CRn` having the value `0b0010`. The encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	5	4	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1 1	0	0 1 0	Op<6:0>		1	1 1 1 1
											op0		op1		CRn		CRm	op2		Rt

The value of `op<6:0>`, formed by concatenating the `CRm` and `op2` fields, determines the hint instruction as follows:

`0b0000000` `NOP` instruction.

`0b0000001` `YIELD` instruction.

`0b0000010` `WFE` instruction.

`0b0000011` `WFI` instruction.

`0b0000100` `SEV` instruction.

`0b0000101` `SEVL` instruction.

`0b0000110` `DGH` instruction.

`0b0000111` `XPACD, XPACI, XPACLRI` instruction.

`0b0001000` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIA1716 variant.

`0b0001010` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIB1716 variant.

`0b0001100` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIA1716 variant.

`0b0001110` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIB1716 variant.

`0b0010000` `ESB` instruction.

`0b0010001`

: `PSB` instruction.

`0b0010010`

: `TSB` instruction.

`0b0010100` `CSDB` instruction.

`0b0010110` `CLRBHB` instruction.

`0b0011000` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIAZ variant.

`0b0011001` `PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA` instruction, PACIASP variant.

`0b0011010` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIBZ variant.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b0011011` `PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB` instruction, PACIBSP variant.

`0b0011100` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIAZ variant.

`0b0011101` `AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA` instruction, AUTIASP variant.

`0b0011110` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIBZ variant.

`0b0011111` `AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB` instruction, AUTIBSP variant.

`0b0101000` `CHKFEAT` instruction.

`0b0100xx0` `BTI` instruction.

`0b011000x` `STSHH` instruction.

These instructions are described in A64 Base Instruction Descriptions.

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.
- The operation of the A64 instructions for architectural hints are identical to the corresponding T32 and A32 instructions.

For more information about:

- The `WFE`, `WFI`, `SEV`, and `SEVL` instructions, see Mechanisms for entering a low-power state.
- The `YIELD` instruction, see Software control features and EL0.
- The `STSHH` instruction, see Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) instructions.

C5.1.3.2 Barriers and CLREX

Within the `op0` == `0b00` encodings, the barriers and `CLREX` instructions are identified by `CRn` having the value `0b0011`. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 0	0 1 1	0 0 1 1	CRm	op2	1 1 1 1 1
		op0	op1	CRn			Rt

The value of `op2` determines the instruction, as follows.

`0b001` `DSB` instruction, Memory nXS barrier variant.

`0b010` `CLREX` instruction.

`0b100` `DSB` instruction, Memory barrier variant.

`0b101` `DMB` instruction.

`0b110` `ISB` instruction.

`0b111` `SB` instruction.

`0b000`; `0b011` UNDEFINED.

These instructions are described in A64 Base Instruction Descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.
- The operation of the A64 instructions for barriers and `CLREX` are identical to the corresponding T32 and A32 instructions.

For more information about:

- The barrier instructions, see Memory barriers.
- The `CLREX` instruction, see Synchronization and semaphores.

C5.1.3.3 Instructions for accessing the PSTATE fields

Within the `op0` == `0b00` encodings, the instructions that can be used to modify PSTATE fields directly are identified by `CRn` having the value `0b0100`. The encoding of these instructions is shown in Figure C5-1 and Figure C5-2:

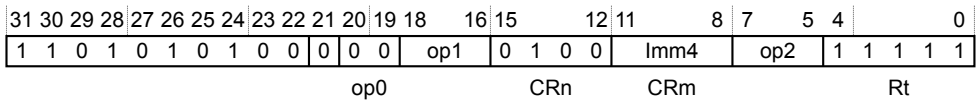

Figure C5-1 Instructions using #Imm4

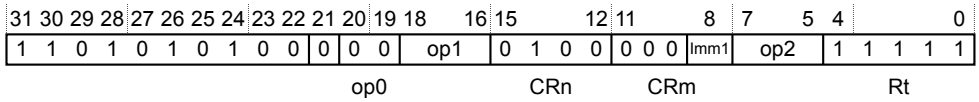

Figure C5-2 Instructions using #Imm1

These instructions are:

```
CFINV                 ; Inverts the value of PSTATE.C
MSR DAIFSet, #Imm4    ; Used to set any or all of DAIF to 1
MSR DAIFClr, #Imm4    ; Used to clear any or all of DAIF to 0
MSR SPSel, #Imm4      ; Used to select the Stack Pointer, between SP_EL0 and SP_ELx
MSR UAO, #Imm4        ; Used to set the value of PSTATE.UAO
MSR PAN, #Imm4        ; Used to set the value of PSTATE.PAN
MSR DIT, #Imm4        ; Used to set the value of PSTATE.DIT
MSR SSBS, #Imm4       ; Used to set the value of PSTATE.SSBS
MSR TCO, #Imm4        ; Used to set the value of PSTATE.TCO
MSR ALLINT, #Imm1     ; Used to set the value of PSTATE.ALLINT
```

The value of `op2` selects the instruction form, which defines the constraints on the values of the `op1` and `Imm4` or `Imm1` arguments, as follows:

`op2` == `0b000` Selects the `CFINV` instruction.

`op2` == `0b011` Selects the `MSR UAO` instruction.

`op2` == `0b100` Selects the `MSR PAN` instruction.

`op2` == `0b101` Selects the `MSR SPSel` instruction.

`op2` == `0b001` Selects the `MSR SSBS` instruction.

`op2` == `0b010` Selects the `MSR DIT` instruction.

`op2` == `0b100` Selects the `MSR TCO` instruction.

`op2` == `0b110` Selects the `MSR DAIFSet` instruction, that sets the specified PSTATE.{D, A, I, F} bits to 1.

`op2` == `0b111` Selects the `MSR DAIFClr` instruction, that clears the specified PSTATE.{D, A, I, F} bits to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`op2 == 0b000` Selects the `MSR ALLINT` instruction.

All other combinations of `op1` and `op2` are reserved, and the corresponding instructions are UNDEFINED.

Note

For PSTATE updates, instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

Writes to PSTATE occur in program order without the need for additional synchronization. Changing PSTATE.SP to use SP_EL0 synchronizes any updates to SP_EL0 that have been written by an `MSR` to SP_EL0, without the need for additional synchronization.

C5.1.4 op0 == `0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, Guarded Control Stack, and FEAT_RME_GPC3 instructions

The System instructions are encoded with `op0== 0b01`. The different groups of System instructions are identified by the values of `CRn` and `CRm`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 0	1 op1	CRn	CRm	op2	Xt
		op0					

The grouping of these instructions depending on the `CRn` and `CRm` fields is as follows:

`CRn == 0b0111` The instruction group is determined by the value of `CRm`, as follows:

- `CRm == 0b0000` Associate a physical address with a memory-mapped location, APAS instruction.

  See FEAT_RME_GPC3 instructions.

- `CRm == {0b0001, 0b0101}` Instruction cache maintenance instructions.

  See Cache maintenance instructions, and data cache zero operations.

- `CRm == 0b0010` Branch Record Buffer and Trace Extension instructions.

  See Branch Record Buffer instructions and Trace Extension instructions.

- `CRm == 0b0011` Prediction restriction instructions.

  See Prediction restriction instructions.

- `CRm == 0b0100` Data cache zero operations.

  See Cache maintenance instructions, and data cache zero operations.

- `CRm == {0b0110, 0b1010, 0b1011, 0b1100, 0b1101, 0b1110, 0b1111}` Data cache maintenance instructions.

  See Cache maintenance instructions, and data cache zero operations.

- `CRm == 0b0111` Guarded Control Stack instructions.

  See Guarded Control Stack instructions.

- `CRm == {0b1000, 0b1001}` See Address translation instructions.

`CRn == {0b1000, 0b1001}` See TLB maintenance instructions.

`CRn == {0b1011, 0b1111}` See Reserved encoding space for IMPLEMENTATION DEFINED instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.1.4.1 Cache maintenance instructions, and data cache zero operations

Table C5-1 lists the Cache maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-1 Cache maintenance instructions and data cache zero operations

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b0001	0b000	IC IALLUIS
0b01	0b000	0b0111	0b0101	0b000	IC IALLU
0b01	0b000	0b0111	0b0110	0b001	DC IVAC
0b01	0b000	0b0111	0b0110	0b010	DC ISW
0b01	0b000	0b0111	0b0110	0b011	DC IGVAC
0b01	0b000	0b0111	0b0110	0b100	DC IGSW
0b01	0b000	0b0111	0b0110	0b101	DC IGDVAC
0b01	0b000	0b0111	0b0110	0b110	DC IGDSW
0b01	0b000	0b0111	0b1010	0b010	DC CSW
0b01	0b000	0b0111	0b1010	0b100	DC CGSW
0b01	0b000	0b0111	0b1010	0b110	DC CGDSW
0b01	0b000	0b0111	0b1110	0b010	DC CISW
0b01	0b000	0b0111	0b1110	0b100	DC CIGSW
0b01	0b000	0b0111	0b1110	0b110	DC CIGDSW
0b01	0b000	0b0111	0b1111	0b001	DC CIVAPS
0b01	0b000	0b0111	0b1111	0b101	DC CIGDVAPS
0b01	0b011	0b0111	0b0100	0b001	DC ZVA
0b01	0b011	0b0111	0b0100	0b011	DC GVA
0b01	0b011	0b0111	0b0100	0b100	DC GZVA
0b01	0b011	0b0111	0b0101	0b001	IC IVAU
0b01	0b011	0b0111	0b1010	0b001	DC CVAC
0b01	0b011	0b0111	0b1010	0b011	DC CGVAC
0b01	0b011	0b0111	0b1010	0b101	DC CGDVAC
0b01	0b011	0b0111	0b1011	0b000	DC CVAOC
0b01	0b011	0b0111	0b1011	0b001	DC CVAU
0b01	0b011	0b0111	0b1011	0b111	DC CGDVAOC
0b01	0b011	0b0111	0b1100	0b001	DC CVAP
0b01	0b011	0b0111	0b1100	0b011	DC CGVAP
0b01	0b011	0b0111	0b1100	0b101	DC CGDVAP
0b01	0b011	0b0111	0b1101	0b001	DC CVADP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b011	0b0111	0b1101	0b011	DC CGVADP
0b01	0b011	0b0111	0b1101	0b101	DC CGDVADP
0b01	0b011	0b0111	0b1110	0b001	DC CIVAC
0b01	0b011	0b0111	0b1110	0b011	DC CIGVAC
0b01	0b011	0b0111	0b1110	0b101	DC CIGDVAC
0b01	0b011	0b0111	0b1111	0b000	DC CIVAOC
0b01	0b011	0b0111	0b1111	0b111	DC CIGDVAOC
0b01	0b100	0b0111	0b1110	0b000	DC CIPAE
0b01	0b100	0b0111	0b1110	0b111	DC CIGDPAE
0b01	0b110	0b0111	0b1110	0b001	DC CIPAPA
0b01	0b110	0b0111	0b1110	0b101	DC CIGDPAPA

For more information about these instructions, see About cache maintenance in AArch64 state and A64 Cache maintenance instructions.

C5.1.4.2 Prediction restriction instructions

Table C5-2 lists the Prediction restriction instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax.

Table C5-2 Prediction restriction instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b011	0b0111	0b0011	0b100	CFP RCTX
0b01	0b011	0b0111	0b0011	0b101	DVP RCTX
0b01	0b011	0b0111	0b0011	0b110	COSP RCTX
0b01	0b011	0b0111	0b0011	0b111	CPP RCTX

For more information about these instructions, see Execution, data prediction and prefetching restriction System instructions.

C5.1.4.3 Address translation instructions

Table C5-3 lists the Address translation instructions and their encodings. The syntax of the instructions includes `Xt`, that provides the address to be translated.

Table C5-3 Address translation instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b1000	0b000	AT S1E1R
0b01	0b000	0b0111	0b1000	0b001	AT S1E1W
0b01	0b000	0b0111	0b1000	0b010	AT S1E0R

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b1000	0b011	AT S1E0W
0b01	0b000	0b0111	0b1001	0b000	AT S1E1RP
0b01	0b000	0b0111	0b1001	0b001	AT S1E1WP
0b01	0b000	0b0111	0b1001	0b010	AT S1E1A
0b01	0b100	0b0111	0b1000	0b000	AT S1E2R
0b01	0b100	0b0111	0b1000	0b001	AT S1E2W
0b01	0b100	0b0111	0b1000	0b100	AT S12E1R
0b01	0b100	0b0111	0b1000	0b101	AT S12E1W
0b01	0b100	0b0111	0b1000	0b110	AT S12E0R
0b01	0b100	0b0111	0b1000	0b111	AT S12E0W
0b01	0b100	0b0111	0b1001	0b010	AT S1E2A
0b01	0b110	0b0111	0b1000	0b000	AT S1E3R
0b01	0b110	0b0111	0b1000	0b001	AT S1E3W
0b01	0b110	0b0111	0b1001	0b010	AT S1E3A

For more information about these instructions, see Address translation instructions.

C5.1.4.4 TLB maintenance instructions

Table C5-4 lists the TLB maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-4 TLB maintenance instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1000	0b0001	0b000	TLBI VMALLE1OS
0b01	0b000	0b1000	0b0001	0b001	TLBI VAE1OS
0b01	0b000	0b1000	0b0001	0b001	TLBIP VAE1OS
0b01	0b000	0b1000	0b0001	0b010	TLBI ASIDE1OS
0b01	0b000	0b1000	0b0001	0b011	TLBI VAAE1OS
0b01	0b000	0b1000	0b0001	0b011	TLBIP VAAE1OS
0b01	0b000	0b1000	0b0001	0b101	TLBI VALE1OS
0b01	0b000	0b1000	0b0001	0b101	TLBIP VALE1OS
0b01	0b000	0b1000	0b0001	0b111	TLBI VAALE1OS
0b01	0b000	0b1000	0b0001	0b111	TLBIP VAALE1OS
0b01	0b000	0b1000	0b0010	0b001	TLBI RVAE1IS
0b01	0b000	0b1000	0b0010	0b001	TLBIP RVAE1IS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1000	0b0010	0b011	TLBI RVAAE1IS
0b01	0b000	0b1000	0b0010	0b011	TLBIP RVAAE1IS
0b01	0b000	0b1000	0b0010	0b101	TLBI RVALE1IS
0b01	0b000	0b1000	0b0010	0b101	TLBIP RVALE1IS
0b01	0b000	0b1000	0b0010	0b111	TLBI RVAALE1IS
0b01	0b000	0b1000	0b0010	0b111	TLBIP RVAALE1IS
0b01	0b000	0b1000	0b0011	0b000	TLBI VMALLE1IS
0b01	0b000	0b1000	0b0011	0b001	TLBI VAE1IS
0b01	0b000	0b1000	0b0011	0b001	TLBIP VAE1IS
0b01	0b000	0b1000	0b0011	0b010	TLBI ASIDE1IS
0b01	0b000	0b1000	0b0011	0b011	TLBI VAAE1IS
0b01	0b000	0b1000	0b0011	0b011	TLBIP VAAE1IS
0b01	0b000	0b1000	0b0011	0b101	TLBI VALE1IS
0b01	0b000	0b1000	0b0011	0b101	TLBIP VALE1IS
0b01	0b000	0b1000	0b0011	0b111	TLBI VAALE1IS
0b01	0b000	0b1000	0b0011	0b111	TLBIP VAALE1IS
0b01	0b000	0b1000	0b0101	0b001	TLBI RVAE1OS
0b01	0b000	0b1000	0b0101	0b001	TLBIP RVAE1OS
0b01	0b000	0b1000	0b0101	0b011	TLBI RVAAE1OS
0b01	0b000	0b1000	0b0101	0b011	TLBIP RVAAE1OS
0b01	0b000	0b1000	0b0101	0b101	TLBI RVALE1OS
0b01	0b000	0b1000	0b0101	0b101	TLBIP RVALE1OS
0b01	0b000	0b1000	0b0101	0b111	TLBI RVAALE1OS
0b01	0b000	0b1000	0b0101	0b111	TLBIP RVAALE1OS
0b01	0b000	0b1000	0b0110	0b001	TLBI RVAE1
0b01	0b000	0b1000	0b0110	0b001	TLBIP RVAE1
0b01	0b000	0b1000	0b0110	0b011	TLBI RVAAE1
0b01	0b000	0b1000	0b0110	0b011	TLBIP RVAAE1
0b01	0b000	0b1000	0b0110	0b101	TLBI RVALE1
0b01	0b000	0b1000	0b0110	0b101	TLBIP RVALE1
0b01	0b000	0b1000	0b0110	0b111	TLBI RVAALE1
0b01	0b000	0b1000	0b0110	0b111	TLBIP RVAALE1
0b01	0b000	0b1000	0b0111	0b000	TLBI VMALLE1
0b01	0b000	0b1000	0b0111	0b001	TLBI VAE1
0b01	0b000	0b1000	0b0111	0b001	TLBIP VAE1
0b01	0b000	0b1000	0b0111	0b010	TLBI ASIDE1
0b01	0b000	0b1000	0b0111	0b011	TLBI VAAE1
0b01	0b000	0b1000	0b0111	0b011	TLBIP VAAE1
0b01	0b000	0b1000	0b0111	0b101	TLBI VALE1
0b01	0b000	0b1000	0b0111	0b101	TLBIP VALE1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1000	0b0111	0b111	TLBI VAALE1
0b01	0b000	0b1000	0b0111	0b111	TLBIP VAALE1
0b01	0b000	0b1001	0b0001	0b000	TLBI VMALLE1OSNXS
0b01	0b000	0b1001	0b0001	0b001	TLBI VAE1OSNXS
0b01	0b000	0b1001	0b0001	0b001	TLBIP VAE1OSNXS
0b01	0b000	0b1001	0b0001	0b010	TLBI ASIDE1OSNXS
0b01	0b000	0b1001	0b0001	0b011	TLBI VAAE1OSNXS
0b01	0b000	0b1001	0b0001	0b011	TLBIP VAAE1OSNXS
0b01	0b000	0b1001	0b0001	0b101	TLBI VALE1OSNXS
0b01	0b000	0b1001	0b0001	0b101	TLBIP VALE1OSNXS
0b01	0b000	0b1001	0b0001	0b111	TLBI VAALE1OSNXS
0b01	0b000	0b1001	0b0001	0b111	TLBIP VAALE1OSNXS
0b01	0b000	0b1001	0b0010	0b001	TLBI RVAE1ISNXS
0b01	0b000	0b1001	0b0010	0b001	TLBIP RVAE1ISNXS
0b01	0b000	0b1001	0b0010	0b011	TLBI RVAAE1ISNXS
0b01	0b000	0b1001	0b0010	0b011	TLBIP RVAAE1ISNXS
0b01	0b000	0b1001	0b0010	0b101	TLBI RVALE1ISNXS
0b01	0b000	0b1001	0b0010	0b101	TLBIP RVALE1ISNXS
0b01	0b000	0b1001	0b0010	0b111	TLBI RVAALE1ISNXS
0b01	0b000	0b1001	0b0010	0b111	TLBIP RVAALE1ISNXS
0b01	0b000	0b1001	0b0011	0b000	TLBI VMALLE1ISNXS
0b01	0b000	0b1001	0b0011	0b001	TLBI VAE1ISNXS
0b01	0b000	0b1001	0b0011	0b001	TLBIP VAE1ISNXS
0b01	0b000	0b1001	0b0011	0b010	TLBI ASIDE1ISNXS
0b01	0b000	0b1001	0b0011	0b011	TLBI VAAE1ISNXS
0b01	0b000	0b1001	0b0011	0b011	TLBIP VAAE1ISNXS
0b01	0b000	0b1001	0b0011	0b101	TLBI VALE1ISNXS
0b01	0b000	0b1001	0b0011	0b101	TLBIP VALE1ISNXS
0b01	0b000	0b1001	0b0011	0b111	TLBI VAALE1ISNXS
0b01	0b000	0b1001	0b0011	0b111	TLBIP VAALE1ISNXS
0b01	0b000	0b1001	0b0101	0b001	TLBI RVAE1OSNXS
0b01	0b000	0b1001	0b0101	0b001	TLBIP RVAE1OSNXS
0b01	0b000	0b1001	0b0101	0b011	TLBI RVAAE1OSNXS
0b01	0b000	0b1001	0b0101	0b011	TLBIP RVAAE1OSNXS
0b01	0b000	0b1001	0b0101	0b101	TLBI RVALE1OSNXS
0b01	0b000	0b1001	0b0101	0b101	TLBIP RVALE1OSNXS
0b01	0b000	0b1001	0b0101	0b111	TLBI RVAALE1OSNXS
0b01	0b000	0b1001	0b0101	0b111	TLBIP RVAALE1OSNXS
0b01	0b000	0b1001	0b0110	0b001	TLBI RVAE1NXS
0b01	0b000	0b1001	0b0110	0b001	TLBIP RVAE1NXS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1001	0b0110	0b011	TLBI RVAAE1NXS
0b01	0b000	0b1001	0b0110	0b011	TLBIP RVAAE1NXS
0b01	0b000	0b1001	0b0110	0b101	TLBI RVALE1NXS
0b01	0b000	0b1001	0b0110	0b101	TLBIP RVALE1NXS
0b01	0b000	0b1001	0b0110	0b111	TLBI RVAALE1NXS
0b01	0b000	0b1001	0b0110	0b111	TLBIP RVAALE1NXS
0b01	0b000	0b1001	0b0111	0b000	TLBI VMALLE1NXS
0b01	0b000	0b1001	0b0111	0b001	TLBI VAE1NXS
0b01	0b000	0b1001	0b0111	0b001	TLBIP VAE1NXS
0b01	0b000	0b1001	0b0111	0b010	TLBI ASIDE1NXS
0b01	0b000	0b1001	0b0111	0b011	TLBI VAAE1NXS
0b01	0b000	0b1001	0b0111	0b011	TLBIP VAAE1NXS
0b01	0b000	0b1001	0b0111	0b101	TLBI VALE1NXS
0b01	0b000	0b1001	0b0111	0b101	TLBIP VALE1NXS
0b01	0b000	0b1001	0b0111	0b111	TLBI VAALE1NXS
0b01	0b000	0b1001	0b0111	0b111	TLBIP VAALE1NXS
0b01	0b100	0b1000	0b0000	0b001	TLBI IPAS2E1IS
0b01	0b100	0b1000	0b0000	0b001	TLBIP IPAS2E1IS
0b01	0b100	0b1000	0b0000	0b010	TLBI RIPAS2E1IS
0b01	0b100	0b1000	0b0000	0b010	TLBIP RIPAS2E1IS
0b01	0b100	0b1000	0b0000	0b101	TLBI IPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b101	TLBIP IPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b110	TLBI RIPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b110	TLBIP RIPAS2LE1IS
0b01	0b100	0b1000	0b0001	0b000	TLBI ALLE2OS
0b01	0b100	0b1000	0b0001	0b001	TLBI VAE2OS
0b01	0b100	0b1000	0b0001	0b001	TLBIP VAE2OS
0b01	0b100	0b1000	0b0001	0b100	TLBI ALLE1OS
0b01	0b100	0b1000	0b0001	0b101	TLBI VALE2OS
0b01	0b100	0b1000	0b0001	0b101	TLBIP VALE2OS
0b01	0b100	0b1000	0b0001	0b110	TLBI VMALLS12E1OS
0b01	0b100	0b1000	0b0010	0b001	TLBI RVAE2IS
0b01	0b100	0b1000	0b0010	0b001	TLBIP RVAE2IS
0b01	0b100	0b1000	0b0010	0b010	TLBI VMALLWS2E1IS
0b01	0b100	0b1000	0b0010	0b101	TLBI RVALE2IS
0b01	0b100	0b1000	0b0010	0b101	TLBIP RVALE2IS
0b01	0b100	0b1000	0b0011	0b000	TLBI ALLE2IS
0b01	0b100	0b1000	0b0011	0b001	TLBI VAE2IS
0b01	0b100	0b1000	0b0011	0b001	TLBIP VAE2IS
0b01	0b100	0b1000	0b0011	0b100	TLBI ALLE1IS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1000	0b0011	0b101	TLBI VALE2IS
0b01	0b100	0b1000	0b0011	0b101	TLBIP VALE2IS
0b01	0b100	0b1000	0b0011	0b110	TLBI VMALLS12E1IS
0b01	0b100	0b1000	0b0100	0b000	TLBI IPAS2E1OS
0b01	0b100	0b1000	0b0100	0b000	TLBIP IPAS2E1OS
0b01	0b100	0b1000	0b0100	0b001	TLBI IPAS2E1
0b01	0b100	0b1000	0b0100	0b001	TLBIP IPAS2E1
0b01	0b100	0b1000	0b0100	0b010	TLBI RIPAS2E1
0b01	0b100	0b1000	0b0100	0b010	TLBIP RIPAS2E1
0b01	0b100	0b1000	0b0100	0b011	TLBI RIPAS2E1OS
0b01	0b100	0b1000	0b0100	0b011	TLBIP RIPAS2E1OS
0b01	0b100	0b1000	0b0100	0b100	TLBI IPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b100	TLBIP IPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b101	TLBI IPAS2LE1
0b01	0b100	0b1000	0b0100	0b101	TLBIP IPAS2LE1
0b01	0b100	0b1000	0b0100	0b110	TLBI RIPAS2LE1
0b01	0b100	0b1000	0b0100	0b110	TLBIP RIPAS2LE1
0b01	0b100	0b1000	0b0100	0b111	TLBI RIPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b111	TLBIP RIPAS2LE1OS
0b01	0b100	0b1000	0b0101	0b001	TLBI RVAE2OS
0b01	0b100	0b1000	0b0101	0b001	TLBIP RVAE2OS
0b01	0b100	0b1000	0b0101	0b010	TLBI VMALLWS2E1OS
0b01	0b100	0b1000	0b0101	0b101	TLBI RVALE2OS
0b01	0b100	0b1000	0b0101	0b101	TLBIP RVALE2OS
0b01	0b100	0b1000	0b0110	0b001	TLBI RVAE2
0b01	0b100	0b1000	0b0110	0b001	TLBIP RVAE2
0b01	0b100	0b1000	0b0110	0b010	TLBI VMALLWS2E1
0b01	0b100	0b1000	0b0110	0b101	TLBI RVALE2
0b01	0b100	0b1000	0b0110	0b101	TLBIP RVALE2
0b01	0b100	0b1000	0b0111	0b000	TLBI ALLE2
0b01	0b100	0b1000	0b0111	0b001	TLBI VAE2
0b01	0b100	0b1000	0b0111	0b001	TLBIP VAE2
0b01	0b100	0b1000	0b0111	0b100	TLBI ALLE1
0b01	0b100	0b1000	0b0111	0b101	TLBI VALE2
0b01	0b100	0b1000	0b0111	0b101	TLBIP VALE2
0b01	0b100	0b1000	0b0111	0b110	TLBI VMALLS12E1
0b01	0b100	0b1001	0b0000	0b001	TLBI IPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b001	TLBIP IPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b010	TLBI RIPAS2E1ISNXS
0b01	0b100	0b1001	0b0000	0b010	TLBIP RIPAS2E1ISNXS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1001	0b0000	0b101	TLBI IPAS2LE1ISNXS
0b01	0b100	0b1001	0b0000	0b101	TLBIP IPAS2LE1ISNXS
0b01	0b100	0b1001	0b0000	0b110	TLBI RIPAS2LE1ISNXS
0b01	0b100	0b1001	0b0000	0b110	TLBIP RIPAS2LE1ISNXS
0b01	0b100	0b1001	0b0001	0b000	TLBI ALLE2OSNXS
0b01	0b100	0b1001	0b0001	0b001	TLBI VAE2OSNXS
0b01	0b100	0b1001	0b0001	0b001	TLBIP VAE2OSNXS
0b01	0b100	0b1001	0b0001	0b100	TLBI ALLE1OSNXS
0b01	0b100	0b1001	0b0001	0b101	TLBI VALE2OSNXS
0b01	0b100	0b1001	0b0001	0b101	TLBIP VALE2OSNXS
0b01	0b100	0b1001	0b0001	0b110	TLBI VMALLS12E1OSNXS
0b01	0b100	0b1001	0b0010	0b001	TLBI RVAE2ISNXS
0b01	0b100	0b1001	0b0010	0b001	TLBIP RVAE2ISNXS
0b01	0b100	0b1001	0b0010	0b010	TLBI VMALLWS2E1ISNXS
0b01	0b100	0b1001	0b0010	0b101	TLBI RVALE2ISNXS
0b01	0b100	0b1001	0b0010	0b101	TLBIP RVALE2ISNXS
0b01	0b100	0b1001	0b0011	0b000	TLBI ALLE2ISNXS
0b01	0b100	0b1001	0b0011	0b001	TLBI VAE2ISNXS
0b01	0b100	0b1001	0b0011	0b001	TLBIP VAE2ISNXS
0b01	0b100	0b1001	0b0011	0b100	TLBI ALLE1ISNXS
0b01	0b100	0b1001	0b0011	0b101	TLBI VALE2ISNXS
0b01	0b100	0b1001	0b0011	0b101	TLBIP VALE2ISNXS
0b01	0b100	0b1001	0b0011	0b110	TLBI VMALLS12E1ISNXS
0b01	0b100	0b1001	0b0100	0b000	TLBI IPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b000	TLBIP IPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b001	TLBI IPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b001	TLBIP IPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b010	TLBI RIPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b010	TLBIP RIPAS2E1NXS
0b01	0b100	0b1001	0b0100	0b011	TLBI RIPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b011	TLBIP RIPAS2E1OSNXS
0b01	0b100	0b1001	0b0100	0b100	TLBI IPAS2LE1OSNXS
0b01	0b100	0b1001	0b0100	0b100	TLBIP IPAS2LE1OSNXS
0b01	0b100	0b1001	0b0100	0b101	TLBI IPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b101	TLBIP IPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b110	TLBI RIPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b110	TLBIP RIPAS2LE1NXS
0b01	0b100	0b1001	0b0100	0b111	TLBI RIPAS2LE1OSNXS
0b01	0b100	0b1001	0b0100	0b111	TLBIP RIPAS2LE1OSNXS
0b01	0b100	0b1001	0b0101	0b001	TLBI RVAE2OSNXS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1001	0b0101	0b001	TLBIP RVAE2OSNXS
0b01	0b100	0b1001	0b0101	0b010	TLBI VMALLWS2E1OSNXS
0b01	0b100	0b1001	0b0101	0b101	TLBI RVALE2OSNXS
0b01	0b100	0b1001	0b0101	0b101	TLBIP RVALE2OSNXS
0b01	0b100	0b1001	0b0110	0b001	TLBI RVAE2NXS
0b01	0b100	0b1001	0b0110	0b001	TLBIP RVAE2NXS
0b01	0b100	0b1001	0b0110	0b010	TLBI VMALLWS2E1NXS
0b01	0b100	0b1001	0b0110	0b101	TLBI RVALE2NXS
0b01	0b100	0b1001	0b0110	0b101	TLBIP RVALE2NXS
0b01	0b100	0b1001	0b0111	0b000	TLBI ALLE2NXS
0b01	0b100	0b1001	0b0111	0b001	TLBI VAE2NXS
0b01	0b100	0b1001	0b0111	0b001	TLBIP VAE2NXS
0b01	0b100	0b1001	0b0111	0b100	TLBI ALLE1NXS
0b01	0b100	0b1001	0b0111	0b101	TLBI VALE2NXS
0b01	0b100	0b1001	0b0111	0b101	TLBIP VALE2NXS
0b01	0b100	0b1001	0b0111	0b110	TLBI VMALLS12E1NXS
0b01	0b110	0b1000	0b0001	0b000	TLBI ALLE3OS
0b01	0b110	0b1000	0b0001	0b001	TLBI VAE3OS
0b01	0b110	0b1000	0b0001	0b001	TLBIP VAE3OS
0b01	0b110	0b1000	0b0001	0b100	TLBI PAALLOS
0b01	0b110	0b1000	0b0001	0b101	TLBI VALE3OS
0b01	0b110	0b1000	0b0001	0b101	TLBIP VALE3OS
0b01	0b110	0b1000	0b0010	0b001	TLBI RVAE3IS
0b01	0b110	0b1000	0b0010	0b001	TLBIP RVAE3IS
0b01	0b110	0b1000	0b0010	0b101	TLBI RVALE3IS
0b01	0b110	0b1000	0b0010	0b101	TLBIP RVALE3IS
0b01	0b110	0b1000	0b0011	0b000	TLBI ALLE3IS
0b01	0b110	0b1000	0b0011	0b001	TLBI VAE3IS
0b01	0b110	0b1000	0b0011	0b001	TLBIP VAE3IS
0b01	0b110	0b1000	0b0011	0b101	TLBI VALE3IS
0b01	0b110	0b1000	0b0011	0b101	TLBIP VALE3IS
0b01	0b110	0b1000	0b0100	0b011	TLBI RPAOS
0b01	0b110	0b1000	0b0100	0b111	TLBI RPALOS
0b01	0b110	0b1000	0b0101	0b001	TLBI RVAE3OS
0b01	0b110	0b1000	0b0101	0b001	TLBIP RVAE3OS
0b01	0b110	0b1000	0b0101	0b101	TLBI RVALE3OS
0b01	0b110	0b1000	0b0101	0b101	TLBIP RVALE3OS
0b01	0b110	0b1000	0b0110	0b001	TLBI RVAE3
0b01	0b110	0b1000	0b0110	0b001	TLBIP RVAE3
0b01	0b110	0b1000	0b0110	0b101	TLBI RVALE3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b110	0b1000	0b0110	0b101	TLBIP RVALE3
0b01	0b110	0b1000	0b0111	0b000	TLBI ALLE3
0b01	0b110	0b1000	0b0111	0b001	TLBI VAE3
0b01	0b110	0b1000	0b0111	0b001	TLBIP VAE3
0b01	0b110	0b1000	0b0111	0b100	TLBI PAALL
0b01	0b110	0b1000	0b0111	0b101	TLBI VALE3
0b01	0b110	0b1000	0b0111	0b101	TLBIP VALE3
0b01	0b110	0b1001	0b0001	0b000	TLBI ALLE3OSNXS
0b01	0b110	0b1001	0b0001	0b001	TLBI VAE3OSNXS
0b01	0b110	0b1001	0b0001	0b001	TLBIP VAE3OSNXS
0b01	0b110	0b1001	0b0001	0b101	TLBI VALE3OSNXS
0b01	0b110	0b1001	0b0001	0b101	TLBIP VALE3OSNXS
0b01	0b110	0b1001	0b0010	0b001	TLBI RVAE3ISNXS
0b01	0b110	0b1001	0b0010	0b001	TLBIP RVAE3ISNXS
0b01	0b110	0b1001	0b0010	0b101	TLBI RVALE3ISNXS
0b01	0b110	0b1001	0b0010	0b101	TLBIP RVALE3ISNXS
0b01	0b110	0b1001	0b0011	0b000	TLBI ALLE3ISNXS
0b01	0b110	0b1001	0b0011	0b001	TLBI VAE3ISNXS
0b01	0b110	0b1001	0b0011	0b001	TLBIP VAE3ISNXS
0b01	0b110	0b1001	0b0011	0b101	TLBI VALE3ISNXS
0b01	0b110	0b1001	0b0011	0b101	TLBIP VALE3ISNXS
0b01	0b110	0b1001	0b0101	0b001	TLBI RVAE3OSNXS
0b01	0b110	0b1001	0b0101	0b001	TLBIP RVAE3OSNXS
0b01	0b110	0b1001	0b0101	0b101	TLBI RVALE3OSNXS
0b01	0b110	0b1001	0b0101	0b101	TLBIP RVALE3OSNXS
0b01	0b110	0b1001	0b0110	0b001	TLBI RVAE3NXS
0b01	0b110	0b1001	0b0110	0b001	TLBIP RVAE3NXS
0b01	0b110	0b1001	0b0110	0b101	TLBI RVALE3NXS
0b01	0b110	0b1001	0b0110	0b101	TLBIP RVALE3NXS
0b01	0b110	0b1001	0b0111	0b000	TLBI ALLE3NXS
0b01	0b110	0b1001	0b0111	0b001	TLBI VAE3NXS
0b01	0b110	0b1001	0b0111	0b001	TLBIP VAE3NXS
0b01	0b110	0b1001	0b0111	0b101	TLBI VALE3NXS
0b01	0b110	0b1001	0b0111	0b101	TLBIP VALE3NXS

For more information about these instructions, see TLB maintenance instructions.

C5.1.4.5 Branch Record Buffer instructions

Table C5-5 lists the Branch Record Buffer instructions and their encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C5-5 Branch Record Buffer instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b001	0b0111	0b0010	0b100	BRB IALL
0b01	0b001	0b0111	0b0010	0b101	BRB INJ

For more information, see The Branch Record Buffer Extension.

C5.1.4.6 Instrumentation Trace Extension instructions

Table C5-6 lists the Instrumentation Trace Extension instructions and their encodings.

Table C5-6 Trace Extension instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b011	0b0111	0b0010	0b111	TRCIT

For more information, see Instrumentation extension.

C5.1.4.7 Guarded Control Stack instructions

Table C5-7 lists the Guarded Control Stack instructions and their encodings.

Table C5-7 Guarded Control Stack instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b0111	0b100	GCSPUSHX
0b01	0b000	0b0111	0b0111	0b101	GCSPOPCX
0b01	0b000	0b0111	0b0111	0b110	GCSPOPX
0b01	0b011	0b0111	0b0111	0b000	GCSPUSHM
0b01	0b011	0b0111	0b0111	0b001	GCSPOPM
0b01	0b011	0b0111	0b0111	0b010	GCSSS1
0b01	0b011	0b0111	0b0111	0b011	GCSSS2

For more information, see The Guarded Control Stack.

C5.1.4.8 FEAT_RME_GPC3 instructions

Table C5-8 lists the instructions introduced by FEAT_RME_GPC3 and their encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C5-8 FEAT_RME_GPC3 instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b110	0b0111	0b0000	0b000	APAS

C5.1.4.9 Reserved encoding space for IMPLEMENTATION DEFINED instructions

The A64 instruction set reserves the following encoding space for IMPLEMENTATION DEFINED instructions:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	0 1	op1	1 x 1 1	CRm	op2	Rt
				op0		CRn			

The value of `L` defines the use of `Rt` as follows:

0 `Rt` is an argument supplied to the instruction.

1 `Rt` is a result returned by the instruction.

IMPLEMENTATION DEFINED instructions in this encoding space are accessed using the `SYS`, `SYSL`, and `SYSP` instructions, see SYS, SYSL and SYSP.

Accesses to any of the IMPLEMENTATION DEFINED System instruction encodings that are not trapped are permitted to be UNDEFINED.

See also Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.5 op0 == `0b11`, Moves to and from Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with `op0` == `0b11`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	CRn	CRm	op2	Rt
				op0					

C5.1.5.1 Instructions for accessing Special-purpose registers

The value of `CRn` provides the next level of decode of these instructions. For Special-purpose registers, the value of CRn is 4.

The A64 instructions for accessing Special-purpose registers are:

```
MSR <Special-purpose_register>, Xt    ; Write to Special-purpose register
MRS Xt, <Special-purpose_register>    ; Read from Special-purpose register
```

Where `<Special-purpose_register>` is the register name, for example SP_EL0.

For these accesses, `CRn` has the value `0b0100`. The encoding for Special-purpose register accesses is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	0 1 0 0	CRm	op2	Rt
				op0		CRn			

The full list of Special-purpose registers is in Table C5-9. The characteristic of a Special-purpose register is that all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Table C5-9 lists the encodings for `op1`, `CRm`, and `op2` fields for accesses to the Special-purpose registers in AArch64.

Table C5-9 Instruction encodings for Special-purpose register accesses

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0100	0b0000	0b000	RW	SPSR_EL1	SPSR_EL1 SPSR_EL2
0b000	0b0100	0b0000	0b001	RW	ELR_EL1	ELR_EL1 ELR_EL2
0b000	0b0100	0b0001	0b000	RW	SP_EL0	SP_EL0
0b000	0b0100	0b0010	0b000	RW	SPSel	SPSel
0b000	0b0100	0b0010	0b010	RO	CurrentEL	CurrentEL
0b000	0b0100	0b0010	0b011	RW	PAN	PAN
0b000	0b0100	0b0010	0b100	RW	UAO	UAO
0b000	0b0100	0b0011	0b000	RW	ALLINT	ALLINT
0b000	0b0100	0b0011	0b001	RW	PM	PM
0b000	0b0100	0b0110	0b000	RW	ICC_PMR_EL1	ICC_PMR_EL1 ICV_PMR_EL1
0b011	0b0100	0b0010	0b000	RW	NZCV	NZCV
0b011	0b0100	0b0010	0b001	RW	DAIF	DAIF
0b011	0b0100	0b0010	0b010	RW	SVCR	SVCR
0b011	0b0100	0b0010	0b101	RW	DIT	DIT
0b011	0b0100	0b0010	0b110	RW	SSBS	SSBS
0b011	0b0100	0b0010	0b111	RW	TCO	TCO
0b011	0b0100	0b0100	0b000	RW	FPCR	FPCR
0b011	0b0100	0b0100	0b001	RW	FPSR	FPSR
0b011	0b0100	0b0100	0b010	RW	FPMR	FPMR
0b011	0b0100	0b0101	0b000	RW	DSPSR_EL0	DSPSR_EL0
0b011	0b0100	0b0101	0b001	RW	DLR_EL0	DLR_EL0
0b100	0b0100	0b0000	0b000	RW	SPSR_EL2	SPSR_EL1 SPSR_EL2
0b100	0b0100	0b0000	0b001	RW	ELR_EL2	ELR_EL1 ELR_EL2
0b100	0b0100	0b0001	0b000	RW	SP_EL1	SP_EL1
0b100	0b0100	0b0011	0b000	RW	SPSR_irq	SPSR_irq

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b0100	0b0011	0b001	RW	SPSR_abt	SPSR_abt
0b100	0b0100	0b0011	0b010	RW	SPSR_und	SPSR_und
0b100	0b0100	0b0011	0b011	RW	SPSR_fiq	SPSR_fiq
0b101	0b0100	0b0000	0b000	RW	SPSR_EL12	SPSR_EL1
0b101	0b0100	0b0000	0b001	RW	ELR_EL12	ELR_EL1
0b110	0b0100	0b0000	0b000	RW	SPSR_EL3	SPSR_EL3
0b110	0b0100	0b0000	0b001	RW	ELR_EL3	ELR_EL3
0b110	0b0100	0b0001	0b000	RW	SP_EL2	SP_EL2

All direct and indirect reads and writes to Special-purpose registers appear to occur in program order relative to other instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2 Special-purpose registers

This section describes the following Special-purpose registers:

- ALLINT, that holds the PSTATE.ALLINT bit.
- CurrentEL, that holds PSTATE.EL, and that software can read to determine the current Exception level.
- DAIF, that holds the current PSTATE.{D, A, I, F} interrupt mask bits.
- DIT, that holds the PSTATE.DIT bit.
- ELR_EL1, that holds the address to return to for an exception return from EL1.
- ELR_EL2, that holds the address to return to for an exception return from EL2.
- ELR_EL3, that holds the address to return to for an exception return from EL3.
- FPCR, that provides control of floating-point operation.
- FPMR, that controls behaviors of the FP8 instructions.
- FPSR, that provides floating-point status information.
- NZCV, that holds the PSTATE.{N, Z, C, V} condition flags.
- PAN, that holds the PSTATE.PAN state bit.
- PM, that provides access to the Profiling exception mask bit.
- SP_EL0, that holds the stack pointer for EL0.
- SP_EL1, that holds the stack pointer for EL1.
- SP_EL2, that holds the stack pointer for EL2.
- SP_EL3, that holds the stack pointer for EL3.
- SPSel, that holds PSTATE.SP, that at EL1 or higher selects the current SP.
- SPSR_abt, that holds process state on taking an exception to AArch32 Abort mode.
- SPSR_EL1, that holds process state on taking an exception to AArch64 EL1.
- SPSR_EL2, that holds process state on taking an exception to AArch64 EL2.
- SPSR_EL3, that holds process state on taking an exception to AArch64 EL3.
- SPSR_fiq, that holds process state on taking an exception to AArch32 FIQ mode.
- SPSR_irq, that holds process state on taking an exception to AArch32 IRQ mode.
- SPSR_und, that holds process state on taking an exception to AArch32 Undefined mode.
- SSBS, that holds the PSTATE.SSBS bit.
- SVCR, controls Streaming SVE mode and SME behavior.
- TCO, that holds the PSTATE.TCO bit.
- UAO, that holds the PSTATE.UAO bit.

The following registers are also Special-purpose registers:

- DLR_EL0, that holds the address to return to for a return from Debug state.
- DSPSR_EL0, that holds process state on entry to Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.1 ALLINT, All Interrupt Mask Bit

The ALLINT characteristics are:

Purpose

Allows access to the all interrupt mask bit.

Configuration

This register is present only when FEAT_NMI is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ALLINT are UNDEFINED.

Attributes

ALLINT is a 64-bit register.

Field descriptions

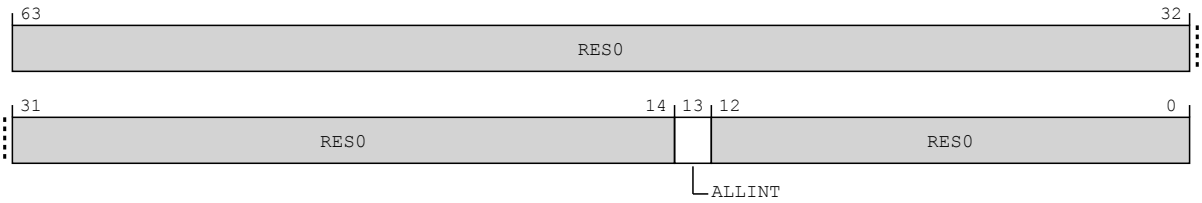

Bits [63:14]

Reserved, RES0.

ALLINT, bit [13]

All interrupt mask. An interrupt is controlled by PSTATE.ALLINT when all of the following apply:

- SCTLR_ELx.NMI is 1.
- The interrupt is targeted at ELx.
- Execution is at ELx.

ALLINT	Meaning
0b0	This control does not cause any interrupts to be masked.
0b1	If SCTLR_ELx.NMI is 1 and execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx, with or without Superpriority, is masked.

The value of this bit is set to the inverse value in the SCTLR_ELx.SPINTMASK field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ALLINT

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ALLINT
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_NMI) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{50} :: PSTATE.ALLINT :: Zeros{13};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{50} :: PSTATE.ALLINT :: Zeros{13};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{50} :: PSTATE.ALLINT :: Zeros{13};
end;
```

```
MSR ALLINT, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_NMI) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsHCRXEL2Enabled() && HCRX_EL2().TALLINT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        PSTATE.ALLINT = X{64}(t)[13];
    end;
elsif PSTATE.EL == EL2 then
    PSTATE.ALLINT = X{64}(t)[13];
elsif PSTATE.EL == EL3 then
    PSTATE.ALLINT = X{64}(t)[13];
end;
```

```
MSR ALLINT, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b001	0b0100	0b000x	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.2 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

Purpose

Holds the current Exception level.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CurrentEL are UNDEFINED.

Attributes

CurrentEL is a 64-bit register.

Field descriptions

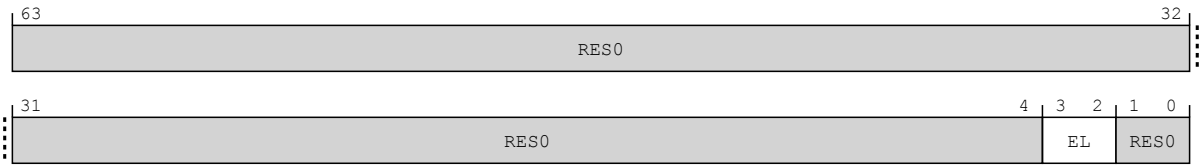

Bits [63:4]

Reserved, RES0.

EL, bits [3:2]

Current Exception level.

EL	Meaning
`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

When the Effective value of HCR_EL2.NV is 1, EL1 read accesses to the CurrentEL register return the value of `0b10` in this field.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'01'`.
  - When the highest implemented Exception level is EL2, this field resets to `'10'`.
  - Otherwise, this field resets to `'11'`.

Bits [1:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CurrentEL

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CurrentEL
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        X{64}(t) = Zeros{60} :: '10' :: Zeros{2};
    else
        X{64}(t) = Zeros{60} :: PSTATE.EL :: Zeros{2};
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{60} :: PSTATE.EL :: Zeros{2};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{60} :: PSTATE.EL :: Zeros{2};
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.3 DAIF, Interrupt Mask Bits

The DAIF characteristics are:

Purpose

Allows access to the interrupt mask bits.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DAIF are UNDEFINED.

Attributes

DAIF is a 64-bit register.

Field descriptions

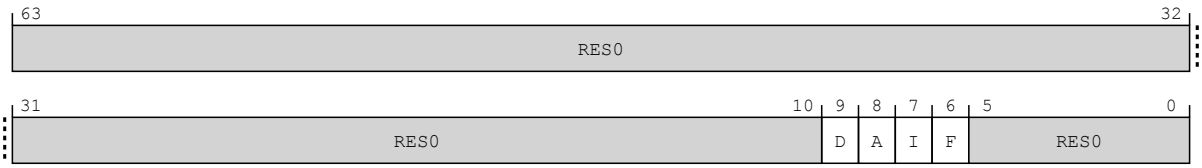

Bits [63:10]

Reserved, RES0.

D, bit [9]

Process state D mask.

D	Meaning
`0b0`	Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
`0b1`	Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

A, bit [8]

SError exception mask bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

I, bit [7]

IRQ mask bit.

I	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

F, bit [6]

FIQ mask bit.

F	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [5:0]

Reserved, RES0.

Accessing DAIF

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DAIF
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ELIsInHost(EL0) || SCTLR_EL1().UMA == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        X{64}(t) = Zeros{54} :: PSTATE.[D,A,I,F] :: Zeros{6};
    end;
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{54} :: PSTATE.[D,A,I,F] :: Zeros{6};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{54} :: PSTATE.[D,A,I,F] :: Zeros{6};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{54} :: PSTATE.[D,A,I,F] :: Zeros{6};
end;
```

MSR DAIF, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ELIsInHost(EL0) || SCTLR_EL1().UMA == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        PSTATE.[D,A,I,F] = X{64}(t)[9:6];
    end;
elsif PSTATE.EL == EL1 then
    PSTATE.[D,A,I,F] = X{64}(t)[9:6];
elsif PSTATE.EL == EL2 then
    PSTATE.[D,A,I,F] = X{64}(t)[9:6];
elsif PSTATE.EL == EL3 then
    PSTATE.[D,A,I,F] = X{64}(t)[9:6];
end;
```

MSR DAIFSet, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR DAIFClr, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.4 DIT, Data Independent Timing

The DIT characteristics are:

Purpose

Allows access to the Data Independent Timing bit.

Configuration

This register is present only when FEAT_DIT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DIT are UNDEFINED.

Attributes

DIT is a 64-bit register.

Field descriptions

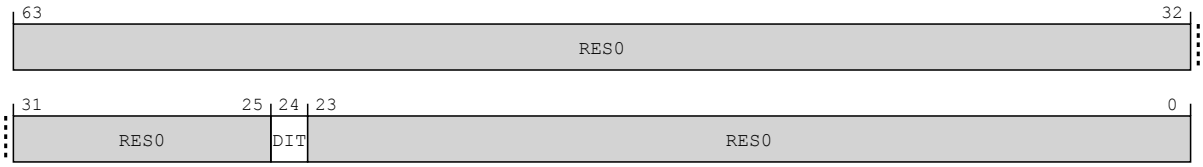

Bits [63:25]

Reserved, RES0.

DIT, bit [24]

Data Independent Timing.

DIT	Meaning
0b0	The architecture makes no statement about the timing properties of any instructions.
0b1	The architecture requires that the execution time of a data-independent-time sequence of code must be independent of all data-independent-time values. For more information, see About PSTATE.DIT.

The list of data-independent-time instructions is:

- Flag Manipulation: CFINV.
- Data Processing – Immediate:
  - Add/subtract (immediate): ADD, ADDS, SUB, and SUBS.
  - Bitfield: BFM, SBFM, and UBFM.
  - Extract: EXTR.
  - Logical (immediate): AND, ANDS, EOR, and ORR.
  - Min/max (immediate): SMAX, SMIN, UMAX, and UMIN.
- Data Processing – Register:
  - Add/subtract (extended register): ADD, ADDS, SUB, and SUBS.
  - Add/subtract (shifted register): ADD, ADDS, SUB, and SUBS.
  - Add/subtract (with carry): ADC, ADCS, SBC, and SBCS.
  - Conditional compare (immediate): CCMN, and CCMP.
  - Conditional compare (register): CCMN, and CCMP.
  - Conditional select: CSEL, CSINC, CSINV, and CSNEG.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - Data-processing (1 source): ABS, CLS, CLZ, CNT, CTZ, RBIT, REV16, REV32, and REV.
    - Data-processing (2 source): ASRV, CRC32B, CRC32CB, CRC32CH, CRC32CW, CRC32CX, CRC32H, CRC32W, CRC32X, LSLV, LSRV, RORV, SMAX, SMIN, UMAX, and UMIN.
    - Data-processing (3 source): MADD, MSUB, SMADDL, SMSUBL, SMULH, UMADDL, UMSUBL, and UMULH.
    - Evaluate into flags: SETF16, and SETF8.
    - Logical (shifted register): AND, ANDS, BIC, BICS, EON, EOR, ORN, and ORR.
    - Rotate right into flags: RMIF.
- Data Processing – Scalar Floating-Point and Advanced SIMD:
    - Advanced SIMD across lanes: ADDV, SADDLV, SMAXV, SMINV, UADDLV, UMAXV, and UMINV.
    - Advanced SIMD copy: DUP, INS, SMOV, and UMOV.
    - Advanced SIMD extract: EXT.
    - Advanced SIMD modified immediate: BIC, and ORR.
    - Advanced SIMD permute: TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2.
    - Advanced SIMD scalar copy: DUP.
    - Advanced SIMD scalar pairwise: ADDP.
    - Advanced SIMD scalar shift by immediate: Advanced SIMD scalar shift by immediate: SHL, SLI, SRSHR, SRI, SRSRA, SSHR, SSRA, URSHR, URSRA, USHR, and USRA.
    - Advanced SIMD scalar three same: ADD, CMEQ, CMGE, CMGT, CMHI, CMHS, CMTST, SQDMULH, SQRDMULH, SSHL, SUB, and USHL.
    - Advanced SIMD scalar three same extra: SQRDMLAH.
    - Advanced SIMD scalar two-register miscellaneous: ABS, CMEQ, CMGE, CMGT, CMLE, CMLT, and NEG.
    - Advanced SIMD scalar x indexed element: SQDMULH, SQRDMLAH, and SQRDMULH.
    - Advanced SIMD shift by immediate: RSHRN2, RSHRN, SHL, SHRN2, SHRN, SLI, SRI, SSHLL2, SSHLL, SSHR, SSRA, USHLL2, USHLL, USHR, and USRA.
    - Advanced SIMD table lookup: LUTI2, LUTI4, TBL, and TBX.
    - Advanced SIMD three different: ADDHN2, ADDHN, PMULL2, PMULL, RADDHN2, RADDHN, RSUBHN2, RSUBHN, SABAL2, SABAL, SABDL2, SABDL, SADDL2, SADDL, SADDW2, SADDW, SMLAL2, SMLAL, SMLSL2, SMLSL, SMULL2, SMULL, SSUBL2, SSUBL, SSUBW2, SSUBW, SUBHN2, SUBHN, UABAL2, UABAL, UABDL2, UABDL, UADDL2, UADDL, UADDW2, UADDW, UMLAL2, UMLAL, UMLSL2, UMLSL, UMULL2, UMULL, USUBL2, USUBL, USUBW2, and USUBW.
    - Advanced SIMD three same: ADD, ADDP, AND, BIC, BIF, BIT, BSL, CMEQ, CMGE, CMGT, CMHI, CMHS, CMTST, EOR, MLA, MLS, MUL, ORN, ORR, PMUL, SABA, SABD, SHADD, SHSUB, SMAX, SMAXP, SMIN, SMINP, SQDMULH, SQRDMULH, SSHL, SUB, UABA, UABD, UHADD, UHSUB, UMAX, UMAXP, UMIN, UMINP, and USHL.
    - Advanced SIMD three-register extension: SDOT, SMMLA, SQRDMLAH, UDOT, UMMLA, USDOT, and USMMLA.
    - Advanced SIMD two-register miscellaneous: ABS, CLS, CLZ, CMEQ, CMGE, CMGT, CMLE, CMLT, CNT, NEG, NOT, RBIT, REV16, REV32, REV64, SADALP, SADDLP, SHLL2, SHLL, UADALP, UADDLP, XTN2, and XTN.
    - Advanced SIMD vector x indexed element: MLA, MLS, MUL, SDOT, SMLAL2, SMLAL, SMLSL2, SMLSL, SMULL2, SMULL, SQDMULH, SQRDMLAH, SQRDMULH, SUDOT, UDOT, UMLAL2, UMLAL, UMLSL2, UMLSL, UMULL2, UMULL, and USDOT.
    - Cryptographic AES: AESD, AESE, AESIMC, and AESMC.
    - Cryptographic four-register: BCAX, EOR3, and SM3SS1.
    - Cryptographic three-register SHA: SHA1C, SHA1M, SHA1P, SHA1SU0, SHA256H2, SHA256H, and SHA256SU1.
    - Cryptographic three-register SHA 512: RAX1, SHA512H2, SHA512H, SHA512SU1, SM3PARTW1, SM3PARTW2, and SM4EKEY.
    - Cryptographic three-register, imm2: SM3TT1A, SM3TT1B, SM3TT2A, and SM3TT2B.
    - Cryptographic three-register, imm6: XAR.
    - Cryptographic two-register SHA: SHA1H, SHA1SU1, and SHA256SU0.
    - Cryptographic two-register SHA 512: SHA512SU0, and SM4E.
    - Floating-point conditional select: FCSEL.
- Loads and Stores:
    - 128-bit atomic memory operations: LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL, LDSETP, LDSETPA, LDSETPAL, LDSETPL, SWPP, SWPPA, SWPPAL, and SWPPL.
    - Advanced SIMD load/store multiple structures: LD1, LD2, LD3, LD4, ST1, ST2, ST3, and ST4.
    - Advanced SIMD load/store multiple structures (post-indexed): LD1, LD2, LD3, LD4, ST1, ST2, ST3, and ST4.
    - Advanced SIMD load/store single structure: LD1, LD1R, LD2, LD2R, LD3, LD3R, LD4, LD4R, LDAP1, ST1, ST2, ST3, ST4, and STL1.
    - Advanced SIMD load/store single structure (post-indexed): LD1, LD1R, LD2, LD2R, LD3, LD3R, LD4, LD4R, ST1, ST2, ST3, and ST4.
    - Atomic memory operations: LDADD, LDADDA, LDADDAB, LDADDAH, LDADDAL, LDADDALB, LDADDALH, LDADDB, LDADDH, LDADDL, LDADDLB, LDADDLH, LDAPR, LDAPRB, LDAPRH, LDCLR, LDCLRA, LDCLRAB, LDCLRAH, LDCLRAL, LDCLRALB, LDCLRALH, LDCLRB, LDCLRH, LDCLRL, LDCLRLB, LDCLRLH, LDEOR, LDEORA, LDEORAB, LDEORAH, LDEORAL, LDEORALB, LDEORALH, LDEORB, LDEORH, LDEORL, LDEORLB,

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LDEORLH, LDSET, LDSETA, LDSETAB, LDSETAH, LDSETAL, LDSETALB, LDSETALH, LDSETB, LDSETH, LDSETL, LDSETLB, LDSETLH, LDSMAX, LDSMAXA, LDSMAXAB, LDSMAXAH, LDSMAXAL, LDSMAXALB, LDSMAXALH, LDSMAXB, LDSMAXH, LDSMAXL, LDSMAXLB, LDSMAXLH, LDSMIN, LDSMINA, LDSMINAB, LDSMINAH, LDSMINAL, LDSMINALB, LDSMINALH, LDSMINB, LDSMINH, LDSMINL, LDSMINLB, LDSMINLH, LDUMAX, LDUMAXA, LDUMAXAB, LDUMAXAH, LDUMAXAL, LDUMAXALB, LDUMAXALH, LDUMAXB, LDUMAXH, LDUMAXL, LDUMAXLB, LDUMAXLH, LDUMIN, LDUMINA, LDUMINAB, LDUMINAH, LDUMINAL, LDUMINALB, LDUMINALH, LDUMINB, LDUMINH, LDUMINL, LDUMINLB, and LDUMINLH.

- LDAPR/STLR (SIMD&FP): LDAPUR, and STLUR.
- LDAPR/STLR (unscaled immediate): LDAPUR, LDAPURB, LDAPURH, LDAPURSB, LDAPURSH, LDAPURSW, STLUR, STLURB, and STLURH.
- LDAPR/STLR (writeback): LDAPR, and STLR.
- LDIAPP/STILP: LDIAPP, and STILP.
- Load/store exclusive pair: LDAXP, LDXP, STLXP, and STXP.
- Load/store exclusive register: LDAXR, LDAXRB, LDAXRH, LDXR, LDXRB, LDXRH, STLXR, STLXRB, STLXRH, STXR, STXRB and STXRH.
- Load/store no-allocate pair (offset): LDNP, and STNP.
- Load/store ordered: LDAR, LDARB, LDARH, LDLAR, LDLARB, LDLARH, STLLR, STLLRB, STLLRH, STLR, STLRB, and STLRH.
- Load/store register (immediate post-indexed): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (immediate pre-indexed): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (pac): LDRAA, and LDRAB.
- Load/store register (register offset): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (unprivileged): LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, STTR, STTRB, and STTRH.
- Load/store register (unscaled immediate): LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, STUR, STURB, and STURH.
- Load/store register (unsigned immediate): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register pair (offset): LDP, LDPSW, and STP.
- Load/store register pair (post-indexed): LDP, LDPSW, and STP.
- Load/store register pair (pre-indexed): LDP, LDPSW, and STP.

- If FEAT_SME is implemented, SME encodings:
  - SME Add Vector to Array: ADDHA, and ADDVA.
  - SME Integer Outer Product - 32 bit: SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS.
  - SME Memory: LD1B, LD1D, LD1H, LD1Q, LD1W, LDR, ST1B, ST1D, ST1H, ST1Q, ST1W, and STR.
  - SME Move from Array: MOVA, and MOVAZ.
  - SME Move into Array: MOVA.
  - SME Outer Product - 64 bit:
    - SME Int16 outer product: SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS.
  - SME zero array: ZERO.
  - SME2 Expand Lookup Table (Contiguous): LUTI2, and LUTI4.
  - SME2 Expand Lookup Table (Non-contiguous): LUTI2, and LUTI4.
  - SME2 Move Lookup Table: MOVT.
  - SME2 Multi-vector - Indexed (Four registers):
    - SME2 multi-vec indexed long MLA four sources: SMLAL, SMLSL, UMLAL, and UMLSL.
    - SME2 multi-vec indexed long long MLA four sources 32-bit: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
    - SME2 multi-vec indexed long long MLA four sources 64-bit: SMLALL, SMLSLL, UMLALL, and UMLSLL.
    - SME2 multi-vec ternary indexed four registers 32-bit: SDOT, SUDOT, SUVDOT, SVDOT, UDOT, USDOT, USVDOT, and UVDOT.
    - SME2 multi-vec ternary indexed four registers 64-bit: SDOT, SVDOT, UDOT, and UVDOT.
  - SME2 Multi-vector - Indexed (One register):
    - SME2 multi-vec indexed long MLA one source: SMLAL, SMLSL, UMLAL, and UMLSL.
    - SME2 multi-vec indexed long long MLA one source 32-bit: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
    - SME2 multi-vec indexed long long MLA one source 64-bit: SMLALL, SMLSLL, UMLALL, and UMLSLL.
  - SME2 Multi-vector - Indexed (Two registers):
    - SME2 multi-vec indexed long MLA two sources: SMLAL, SMLSL, UMLAL, and UMLSL.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SME2 multi-vec indexed long long MLA two sources 32-bit: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
  - SME2 multi-vec indexed long long MLA two sources 64-bit: SMLALL, SMLSLL, UMLALL, and UMLSLL.
  - SME2 multi-vec ternary indexed two registers 32-bit: SDOT, SUDOT, SVDOT, UDOT, USDOT, and UVDOT.
  - SME2 multi-vec ternary indexed two registers 64-bit: SDOT, and UDOT.
- SME2 Multi-vector - Memory (Contiguous): LD1B, LD1D, LD1H, LD1W, LDNT1B, LDNT1D, LDNT1H, LDNT1W, ST1B, ST1D, ST1H, ST1W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SME2 Multi-vector - Memory (Strided): LD1B, LD1D, LD1H, LD1W, LDNT1B, LDNT1D, LDNT1H, LDNT1W, ST1B, ST1D, ST1H, ST1W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SME2 Multi-vector - Multiple Array Vectors (Four registers):
  - SME2 multiple vectors binary int four registers: ADD, and SUB.
  - SME2 multiple vectors four-way dot product four registers: SDOT, and UDOT.
  - SME2 multiple vectors long MLA four sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 multiple vectors long long MLA four sources: SMLALL, SMLSLL, UMLALL, UMLSLL, and USMLALL.
  - SME2 multiple vectors mixed dot product four registers: USDOT.
  - SME2 multiple vectors ternary int four registers: ADD, and SUB.
  - SME2 multiple vectors two-way dot product four registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple Array Vectors (Two registers):
  - SME2 multiple vectors binary int two registers: ADD, and SUB.
  - SME2 multiple vectors four-way dot product two registers: SDOT, and UDOT.
  - SME2 multiple vectors long MLA two sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 multiple vectors long long MLA two sources: SMLALL, SMLSLL, UMLALL, UMLSLL, and USMLALL.
  - SME2 multiple vectors mixed dot product two registers: USDOT.
  - SME2 multiple vectors ternary int two registers: ADD, and SUB.
  - SME2 multiple vectors two-way dot product two registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers):
  - SME2 multiple vectors int min/max four registers: SMAX, SMIN, UMAX, and UMIN.
- SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers):
  - SME2 multiple vectors int min/max two registers: SMAX, SMIN, UMAX, and UMIN.
- SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Four registers): SQDMULH.
- SME2 Multi-vector - Multiple Vectors SVE Saturating Multiply (Two registers): SQDMULH.
- SME2 Multi-vector - Multiple and Single Array Vectors (Four registers):
  - SME2 single-multi four-way dot product four registers: SDOT, and UDOT.
  - SME2 single-multi long MLA four sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 single-multi long long MLA four sources: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
  - SME2 single-multi mixed dot product four registers: SUDOT, and USDOT.
  - SME2 single-multi ternary int four registers: ADD, and SUB.
  - SME2 single-multi two-way dot product four registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple and Single Array Vectors (Two registers):
  - SME2 multiple and single vector long MLA one source: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 multiple and single vector long long FMA one source: SMLALL, SMLSLL, UMLALL, UMLSLL, and USMLALL.
  - SME2 single-multi four-way dot product two registers: SDOT, and UDOT.
  - SME2 single-multi long MLA two sources: SMLAL, SMLSL, UMLAL, and UMLSL.
  - SME2 single-multi long long MLA two sources: SMLALL, SMLSLL, SUMLALL, UMLALL, UMLSLL, and USMLALL.
  - SME2 single-multi mixed dot product two registers: SUDOT, and USDOT.
  - SME2 single-multi ternary int two registers: ADD, and SUB.
  - SME2 single-multi two-way dot product two registers: SDOT, and UDOT.
- SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers):
  - SME2 single-multi add four registers: ADD.
  - SME2 single-multi int min/max four registers: SMAX, SMIN, UMAX, and UMIN.
  - SME2 single-multi signed saturating doubling multiply high four registers: SQDMULH.
- SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers):
  - SME2 single-multi add two registers: ADD.
  - SME2 single-multi int min/max two registers: SMAX, SMIN, UMAX, and UMIN.
  - SME2 single-multi signed saturating doubling multiply high two registers: SQDMULH.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SME2 Multi-vector - SVE Constructive Binary:
  - SME2 multi-vec CLAMP four registers: `SCLAMP`, and `UCLAMP`.
  - SME2 multi-vec CLAMP two registers: `SCLAMP`, and `UCLAMP`.
  - SME2 multi-vec ZIP two registers: `UZP`, and `ZIP`.
  - SME2 multi-vec quadwords ZIP two registers: `UZP`, and `ZIP`.
- SME2 Multi-vector - SVE Constructive Unary:
  - SME2 multi-vec ZIP four registers: `UZP`, and `ZIP`.
  - SME2 multi-vec quadwords ZIP four registers: `UZP`, and `ZIP`.
  - SME2 multi-vec unpack four registers: `SUNPK`, and `UUNPK`.
  - SME2 multi-vec unpack two registers: `SUNPK`, and `UUNPK`.
- SME2 Multi-vector - SVE Select: `SEL`.
- SME2 Multiple Zero: `ZERO`.
- SME2 Outer Product - Misc:
  - SME2 32-bit binary outer product: `BMOPA`, and `BMOPS`.

- If FEAT_SVE is implemented, SVE encodings:
  - SVE Address Generation: `ADR`.
  - SVE Bitwise Immediate: `AND`, `EOR`, and `ORR`.
  - SVE Bitwise Logical - Unpredicated: `AND`, `BCAX`, `BIC`, `BSL1N`, `BSL2N`, `BSL`, `EOR3`, `EOR`, `NBSL`, `ORR`, and `XAR`.
  - SVE Bitwise Shift - Predicated:
    - SVE bitwise shift by immediate (predicated): `ASR`, `ASRD`, `LSL`, `LSR`, `SRSHR`, and `URSHR`.
    - SVE bitwise shift by vector (predicated): `ASR`, `ASRR`, `LSL`, `LSLR`, `LSR`, and `LSRR`.
    - SVE bitwise shift by wide elements (predicated): `ASR`, `LSL`, and `LSR`.
  - SVE Bitwise Shift - Unpredicated: `ASR`, `LSL`, and `LSR`.
  - SVE Element Count:
    - SVE inc/dec register by element count: `DECB`, `DECD`, `DECH`, `DECW`, `INCB`, `INCD`, `INCH`, and `INCW`.
    - SVE inc/dec vector by element count: `DECD`, `DECH`, `DECW`, `INCD`, `INCH`, and `INCW`.
  - SVE Integer Arithmetic - Unpredicated:
    - SVE integer add/subtract vectors (unpredicated): `ADD`, and `SUB`.
  - SVE Integer Binary Arithmetic - Predicated:
    - SVE bitwise logical operations (predicated): `AND`, `BIC`, `EOR`, and `ORR`.
    - SVE integer add/subtract vectors (predicated): `ADD`, `SUB`, and `SUBR`.
    - SVE integer min/max/difference (predicated): `SABD`, `SMAX`, `SMIN`, `UABD`, `UMAX`, and `UMIN`.
    - SVE integer multiply vectors (predicated): `MUL`, `SMULH`, and `UMULH`.
  - SVE Integer Compare - Scalars: `CTERMEQ`, and `CTERMNE`.
  - SVE Integer Compare - Signed Immediate: `CMP<cc>`.
  - SVE Integer Compare - Unsigned Immediate: `CMP<cc>`.
  - SVE Integer Compare - Vectors: `CMP<cc>`.
  - SVE Integer Misc - Unpredicated: `MOVPRFX`.
  - SVE Integer Multiply-Add - Predicated: `MAD`, `MLA`, `MLS`, and `MSB`.
  - SVE Integer Multiply-Add - Unpredicated:
    - SVE integer dot product (unpredicated): `SDOT`, and `UDOT`.
    - SVE mixed sign dot product: `USDOT`.
    - SVE2 complex integer multiply-add: `CMLA`.
    - SVE2 integer multiply-add long: `SMLALB`, `SMLALT`, `SMLSLB`, `SMLSLT`, `UMLALB`, `UMLALT`, `UMLSLB`, and `UMLSLT`.
    - SVE2 saturating multiply-add high: `SQRDMLAH`.
  - SVE Integer Reduction: `ADDQV`, `ANDQV`, `ANDV`, `EORQV`, `EORV`, `MOVPRFX`, `ORQV`, `ORV`, `SADDV`, `SMAXQV`, `SMAXV`, `SMINQV`, `SMINV`, `UADDV`, `UMAXQV`, `UMAXV`, `UMINQV`, and `UMINV`.
  - SVE Integer Unary Arithmetic - Predicated:
    - SVE bitwise unary operations (predicated): `CLS`, `CLZ`, `CNOT`, `CNT`, and `NOT`.
    - SVE integer unary operations (predicated): `ABS`, `NEG`, `SXTB`, `SXTH`, `SXTW`, `UXTB`, `UXTH`, and `UXTW`.
  - SVE Integer Wide Immediate - Unpredicated:
    - SVE integer add/subtract immediate (unpredicated): `ADD`, `SUB`, and `SUBR`.
    - SVE integer min/max immediate (unpredicated): `SMAX`, `SMIN`, `UMAX`, and `UMIN`.
    - SVE integer multiply immediate (unpredicated): `MUL`.
  - SVE Memory - 32-bit Gather and Unsized Contiguous:
    - SVE 32-bit gather load (scalar plus 32-bit unscaled offsets): `LD1B`, `LD1H`, `LD1SB`, `LD1SH`, and `LD1W`.
    - SVE 32-bit gather load (vector plus immediate): `LD1B`, `LD1H`, `LD1SB`, `LD1SH`, and `LD1W`.
    - SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets): `LD1H`, and `LD1SH`.
    - SVE 32-bit gather load words (scalar plus 32-bit scaled offsets): `LD1W`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SVE load and broadcast element: LD1RB, LD1RD, LD1RH, LD1RSB, LD1RSH, LD1RSW, and LD1RW.
  - SVE load vector register: LDR.
  - SVE2 32-bit gather non-temporal load (vector plus scalar): LDNT1B, LDNT1H, LDNT1SB, LDNT1SH, and LDNT1W.
- SVE Memory - 64-bit Gather:
  - SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets): LD1D, LD1H, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (scalar plus 64-bit scaled offsets): LD1D, LD1H, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (scalar plus 64-bit unscaled offsets): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE 64-bit gather load (vector plus immediate): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE2 128-bit gather load (vector plus scalar): LD1Q.
  - SVE2 64-bit gather non-temporal load (vector plus scalar): LDNT1B, LDNT1D, LDNT1H, LDNT1SB, LDNT1SH, LDNT1SW, and LDNT1W.
- SVE Memory - Contiguous Load:
  - SVE contiguous load (quadwords, scalar plus immediate): LD1D, and LD1W.
  - SVE contiguous load (quadwords, scalar plus scalar): LD1D, and LD1W.
  - SVE contiguous load (scalar plus immediate): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE contiguous load (scalar plus scalar): LD1B, LD1D, LD1H, LD1SB, LD1SH, LD1SW, and LD1W.
  - SVE contiguous non-temporal load (scalar plus immediate): LDNT1B, LDNT1D, LDNT1H, and LDNT1W.
  - SVE contiguous non-temporal load (scalar plus scalar): LDNT1B, LDNT1D, LDNT1H, and LDNT1W.
  - SVE load and broadcast quadword (scalar plus immediate): LD1ROB, LD1ROD, LD1ROH, LD1ROW, LD1RQB, LD1RQD, LD1RQH, and LD1RQW.
  - SVE load and broadcast quadword (scalar plus scalar): LD1ROB, LD1ROD, LD1ROH, LD1ROW, LD1RQB, LD1RQD, LD1RQH, and LD1RQW.
  - SVE load multiple structures (quadwords, scalar plus immediate): LD2Q, LD3Q, and LD4Q.
  - SVE load multiple structures (quadwords, scalar plus scalar): LD2Q, LD3Q, and LD4Q.
  - SVE load multiple structures (scalar plus immediate): LD2B, LD2D, LD2H, LD2W, LD3B, LD3D, LD3H, LD3W, LD4B, LD4D, LD4H, and LD4W.
  - SVE load multiple structures (scalar plus scalar): LD2B, LD2D, LD2H, LD2W, LD3B, LD3D, LD3H, LD3W, LD4B, LD4D, LD4H, and LD4W.
- SVE Memory - Contiguous Store and Unsized Contiguous: ST1B, ST1D, ST1H, ST1W, ST2Q, ST3Q, and ST4Q.
- SVE Memory - Contiguous Store with Immediate Offset: ST1B, ST1D, ST1H, ST1W, ST2B, ST2D, ST2H, ST2W, ST3B, ST3D, ST3H, ST3W, ST4B, ST4D, ST4H, ST4W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SVE Memory - Non-temporal and Multi-register Contiguous Store: ST2B, ST2D, ST2H, ST2W, ST3B, ST3D, ST3H, ST3W, ST4B, ST4D, ST4H, ST4W, STNT1B, STNT1D, STNT1H, and STNT1W.
- SVE Memory - Non-temporal and Quadword Scatter Store: ST1Q, STNT1B, STNT1D, STNT1H, and STNT1W.
- SVE Memory - Scatter: ST1B, ST1D, ST1H, and ST1W.
- SVE Memory - Scatter with Optional Sign Extend: ST1B, ST1D, ST1H, and ST1W.
- SVE Misc: BDEP, BEXT, BGRP, EORBT, EORTB, SADDLBT, SMMLA, SSHLLB, SSHLLT, SSUBLBT, SSUBLTB, UMMLA, USHLLB, USHLLT, and USMMLA.
- SVE Multiply - Indexed:
  - SVE integer dot product (indexed): SDOT, and UDOT.
  - SVE mixed sign dot product (indexed): SUDOT, and USDOT.
  - SVE2 complex integer multiply-add (indexed): CMLA.
  - SVE2 integer multiply (indexed): MUL.
  - SVE2 integer multiply long (indexed): SMULLB, SMULLT, UMULLB, and UMULLT.
  - SVE2 integer multiply-add (indexed): MLA, and MLS.
  - SVE2 integer multiply-add long (indexed): SMLALB, SMLALT, SMLSLB, SMLSLT, UMLALB, UMLALT, UMLSLB, and UMLSLT.
  - SVE2 saturating multiply high (indexed): SQDMULH, and SQRDMULH.
  - SVE2 saturating multiply-add high (indexed): SQRDMLAH.
- SVE Permute Vector - Extract: EXT.
- SVE Permute Vector - Indexed DUP: DUP.
- SVE Permute Vector - Interleaving: TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2.
- SVE Permute Vector - One Source Quadwords: DUPQ, and EXTQ.
- SVE Permute Vector - Predicated:
  - SVE copy SIMD&FP scalar register to vector (predicated): CPY.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SVE copy general register to vector (predicated): `CPY`.
  - SVE reverse doublewords: `REVD`.
  - SVE reverse within elements: `RBIT`, `REVB`, `REVH`, and `REVW`.
- SVE Permute Vector - Segments: `TRN1`, `TRN2`, `UZP1`, `UZP2`, `ZIP1`, and `ZIP2`.
- SVE Permute Vector - TBXQ: `TBXQ`.
- SVE Permute Vector - Three Sources TBL: `TBL`, and `TBX`.
- SVE Permute Vector - Two Sources Quadwords: `TBLQ`, `UZPQ1`, `UZPQ2`, `ZIPQ1`, and `ZIPQ2`.
- SVE Permute Vector - Two Sources TBL: `TBL`.
- SVE Permute Vector - Unpredicated:
  - SVE broadcast general register: `DUP`.
  - SVE insert SIMD&FP scalar register: `INSR`.
  - SVE insert general register: `INSR`.
  - SVE reverse vector elements: `REV`.
  - SVE unpack vector elements: `SUNPKHI`, `SUNPKLO`, `UUNPKHI`, and `UUNPKLO`.
- SVE Predicate Select: `PSEL`.
- SVE Stack Allocation: `ADDPL`, `ADDSPL`, `ADDSVL`, and `ADDVL`.
- SVE Vector Select: `SEL`.
- SVE integer clamp: `SCLAMP`, and `UCLAMP`.
- SVE two-way dot product: `SDOT`, and `UDOT`.
- SVE two-way dot product (indexed): `SDOT`, and `UDOT`.
- SVE2 Accumulate:
  - SVE2 bitwise shift and insert: `SLI`, and `SRI`.
  - SVE2 bitwise shift right and accumulate: `SRSRA`, `SSRA`, `URSRA`, and `USRA`.
  - SVE2 complex integer add: `CADD`.
  - SVE2 integer absolute difference and accumulate: `SABA`, and `UABA`.
  - SVE2 integer absolute difference and accumulate long: `SABALB`, `SABALT`, `UABALB`, and `UABALT`.
  - SVE2 integer add/subtract long with carry: `ADCLB`, `ADCLT`, `SBCLB`, and `SBCLT`.
- SVE2 Crypto Extensions: `AESD`, `AESE`, `AESIMC`, `AESMC`, `RAX1`, `SM4E`, and `SM4EKEY`.
- SVE2 Histogram (Segment) and Lookup Table: `LUTI2`, and `LUTI4`.
- SVE2 Integer - Predicated:
  - SVE2 integer halving add/subtract (predicated): `SHADD`, `SHSUB`, `SHSUBR`, `UHADD`, `UHSUB`, and `UHSUBR`.
  - SVE2 integer pairwise add and accumulate long: `SADALP`, and `UADALP`.
  - SVE2 integer pairwise arithmetic: `ADDP`, `SMAXP`, `SMINP`, `UMAXP`, and `UMINP`.
- SVE2 Integer Multiply - Unpredicated: `MUL`, `PMUL`, `SMULH`, `SQDMULH`, `SQRDMULH`, and `UMULH`.
- SVE2 Narrowing:
  - SVE2 bitwise shift right narrow: `RSHRNB`, `RSHRNT`, `SHRNB`, and `SHRNT`.
  - SVE2 integer add/subtract narrow high part: `ADDHNB`, `ADDHNT`, `RADDHNB`, `RADDHNT`, `RSUBHNB`, `RSUBHNT`, `SUBHNB`, and `SUBHNT`.
- SVE2 Widening Integer Arithmetic:
  - SVE2 integer add/subtract long: `SABDLB`, `SABDLT`, `SADDLB`, `SADDLT`, `SSUBLB`, `SSUBLT`, `UABDLB`, `UABDLT`, `UADDLB`, `UADDLT`, `USUBLB`, and `USUBLT`.
  - SVE2 integer add/subtract wide: `SADDWB`, `SADDWT`, `SSUBWB`, `SSUBWT`, `UADDWB`, `UADDWT`, `USUBWB`, and `USUBWT`.
  - SVE2 integer multiply long: `PMULLB`, `PMULLT`, `SMULLB`, `SMULLT`, `UMULLB`, and `UMULLT`.
- If FEAT_SVE or FEAT_SME is implemented, SVE encodings:
  - SVE predicate logical operations: `AND (predicates)`, `BIC (predicates)`, `EOR (predicates)`, `SEL (predicates)`, `ANDS`, `BICS`, `EORS`, `ORR (predicates)`, `ORN (predicates)`, `NOR`, `NAND`, `ORRS`, `ORNS`, `NORS`, `NANDS`.
  - SVE Predicate Misc.: `PTEST`.
  - SVE Memory - Contiguous Store and Unsized Contiguous: `STR (predicate)`.
  - SVE Memory - 32-bit Gather and Unsized Contiguous: `LDR (predicate)`.
- If FEAT_SVE2p1 or FEAT_SME2p1 is implemented, SVE encodings:
  - SVE Permute Vector - Unpredicated: `PMOV (to vector)`, `PMOV (to predicates)`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bits [23:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DIT

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DIT
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_DIT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    X{64}(t) = Zeros{39} :: PSTATE.DIT :: Zeros{24};
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{39} :: PSTATE.DIT :: Zeros{24};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{39} :: PSTATE.DIT :: Zeros{24};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{39} :: PSTATE.DIT :: Zeros{24};
end;
```

```
MSR DIT, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_DIT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    PSTATE.DIT = X{64}(t)[24];
elsif PSTATE.EL == EL1 then
    PSTATE.DIT = X{64}(t)[24];
elsif PSTATE.EL == EL2 then
    PSTATE.DIT = X{64}(t)[24];
elsif PSTATE.EL == EL3 then
    PSTATE.DIT = X{64}(t)[24];
end;
```

```
MSR DIT, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.5 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

Purpose

When taking an exception to EL1, holds the address to return to.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ELR_EL1 are UNDEFINED.

Attributes

ELR_EL1 is a 64-bit register.

Field descriptions

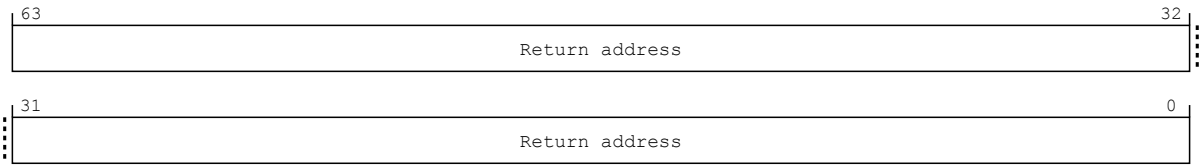

ADDR, bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `ELR_EL1` or `ELR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ELR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x230);
    else
        X{64}(t) = ELR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ELR_EL2();
    else
        X{64}(t) = ELR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ELR_EL1();
end;
```

MSR ELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪!(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x230) = X{64}(t);
    else
        ELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        ELR_EL2() = X{64}(t);
    else
        ELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ELR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, ELR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x230);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ELR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = ELR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR ELR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x230) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        ELR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        ELR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, ELR_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = ELR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = ELR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ELR_EL2();
end;
```

When FEAT_VHE is implemented MSR ELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        ELR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪then
        EXLOCKException();
    else
        ELR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ELR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.6 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

Purpose

When taking an exception to EL2, holds the address to return to.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register ELR_EL2 bits [31:0] are architecturally mapped to AArch32 System register ELR_hyp[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ELR_EL2 are UNDEFINED.

Attributes

ELR_EL2 is a 64-bit register.

Field descriptions

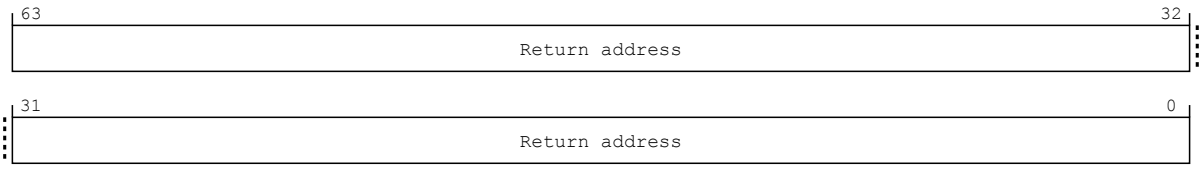

ADDR, bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `ELR_EL2` or `ELR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ELR_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = ELR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = ELR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ELR_EL2();
end;
```

MSR ELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        ELR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        ELR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ELR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, ELR_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x230);
    else
        X{64}(t) = ELR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ELR_EL2();
    else
        X{64}(t) = ELR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ELR_EL1();
end;
```

When FEAT_VHE is implemented MSR ELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
     ↪&& !(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x230) = X{64}(t);
    else
        ELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
     ↪&& ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        ELR_EL2() = X{64}(t);
    else
        ELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ELR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.7 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

Purpose

When taking an exception to EL3, holds the address to return to.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ELR_EL3 are UNDEFINED.

Attributes

ELR_EL3 is a 64-bit register.

Field descriptions

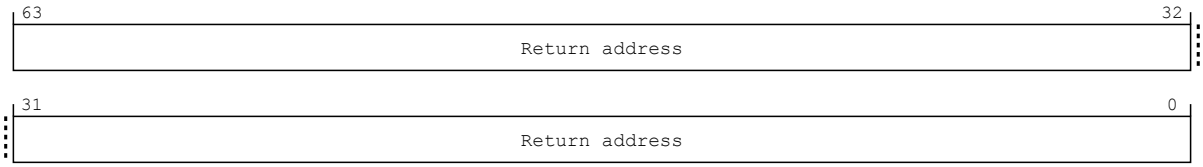

ADDR, bits [63:0]

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ELR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ELR_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ELR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        ELR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.8 FPCR, Floating-point Control Register

The FPCR characteristics are:

Purpose

Controls behavior of the floating-point instructions.

Configuration

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to nonzero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

AArch64 System register FPCR bits [26:15] are architecturally mapped to AArch32 System register FPSCR[26:15].

AArch64 System register FPCR bits [12:8] are architecturally mapped to AArch32 System register FPSCR[12:8].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to FPCR are UNDEFINED.

Attributes

FPCR is a 64-bit register.

Field descriptions

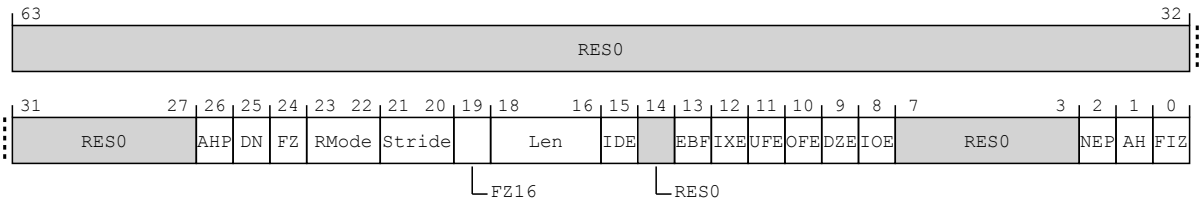

Bits [63:27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision control bit.

AHP	Meaning
0b0	IEEE half-precision format selected.
0b1	Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DN, bit [25]

Default NaN use for NaN propagation.

DN	Meaning
0b0	NaN operands propagate through to the output of a floating-point operation.
0b1	Any operation involving one or more NaNs returns the Default NaN. This bit has no effect on the output of the FABS and FNEG instructions. This bit has no effect on the output of the FMAX, FMAXP, FMAXV, FMIN, FMINP, and FMINV instructions when FPCR.AH is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flushing denormalized numbers to zero control bit.

FZ	Meaning
0b0	If FPCR.AH is 0, the flushing to zero of single-precision and double-precision denormalized inputs to, and outputs of, floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero. If FPCR.AH is 1, the flushing to zero of single-precision and double-precision denormalized outputs of floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.
0b1	If FPCR.AH is 0, denormalized single-precision and double-precision inputs to, and outputs from, floating-point instructions are flushed to zero. If FPCR.AH is 1, denormalized single-precision and double-precision outputs from floating-point instructions are flushed to zero.

For more information, see ‘Flushing denormalized numbers to zero’ and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field.

RMode	Meaning
0b00	Round to Nearest (RN) mode.
0b01	Round towards Plus Infinity (RP) mode.
0b10	Round towards Minus Infinity (RM) mode.
0b11	Round towards Zero (RZ) mode.

The following instructions ignore this field and use Round to Nearest mode:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All FP8 instructions, as described in 'Summary of FP8 instruction behaviors'.
- When FPCR.AH is 1, then the floating-point reciprocal estimate, step and exponent instructions, floating-point square root estimate and step instructions, and instructions that follow Alternate BFloat16 behaviors, as defined in 'Rounding'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Stride field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.{Stride,Len} as RAZ, access to this field is RAZ/WI.

FZ16, bit [19]

When FEAT_FP16 is implemented:

Flushing denormalized numbers to zero control bit on half-precision data-processing instructions.

FZ16	Meaning
`0b0`	For some instructions, this bit disables flushing to zero of inputs and outputs that are half-precision denormalized numbers.
`0b1`	Flushing denormalized numbers to zero enabled. For some instructions that do not convert a half-precision input to a higher precision output, this bit enables flushing to zero of inputs and outputs that are half-precision denormalized numbers.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Len field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.{Stride,Len} as RAZ, access to this field is RAZ/WI.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IDC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IDE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bit [14]

Reserved, RES0.

EBF, bit [13]

When FEAT_EBF16 is implemented:

The value of this bit controls the numeric behaviors of BFloat16 dot product calculations performed by the BFDOT, BFMMLA, BFMOPA, and BFMOPS instructions. If FEAT_SME2 is implemented, this also controls BFVDOT instruction.

When ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 are 0b0010, the PE supports the FPCR.EBF field. Otherwise, FPCR.EBF is RES0.

EBF	Meaning
0b0	These instructions use the standard BFloat16 behaviors: • Ignoring the FPCR.RMode control and using the rounding mode defined for BFloat16. For more information, see 'Round to Odd mode'. • Flushing denormalized inputs and outputs to zero, as if the FPCR.FZ and FPCR.FIZ controls had the value '1'. • Performing unfused multiplies and additions with intermediate rounding of all products and sums.
0b1	These instructions use the extended BFloat16 behaviors: • Supporting all four IEEE 754 rounding modes selected by the FPCR.RMode control. • Optionally, flushing denormalized inputs and outputs to zero, as governed by the FPCR.FZ and FPCR.FIZ controls. • Performing a fused two-way sum-of-products for each pair of adjacent BFloat16 elements, without intermediate rounding of the products, but rounding the single-precision sum before addition to the accumulator. • Generating the default NaN as intermediate sum-of-products when any multiplier input is a NaN, or any product is infinity × 0.0, or there are infinite products with differing signs. • Generating an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable.

IXE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IXC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IXE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFE, bit [11]

Underflow floating-point exception trap enable.

UFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.UFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the FPSR.UFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.UFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFE, bit [10]

Overflow floating-point exception trap enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.OFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.OFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.OFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

DZE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.DZC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.DZE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOE, bit [8]

Invalid Operation floating-point exception trap enable.

IOE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IOC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IOC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IOE is treated as 0 for all purposes other than a direct read or write of the FPCR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

Bits [7:3]

Reserved, RES0.

NEP, bit [2]

When FEAT_AFP is implemented:

Controls how the output elements other than the lowest element of the vector are determined for Advanced SIMD scalar instructions.

NEP	Meaning
`0b0`	Does not affect how the output elements other than the lowest are determined for Advanced SIMD scalar instructions.

`0b1`	The output elements other than the lowest are taken from the following registers:
	• For 3-input scalar versions of the FMLA (by element) and FMLS (by element) instructions, the <Hd>, <Sd>, or <Dd> register.
	• For 3-input versions of the FMADD, FMSUB, FNMADD, and FNMSUB instructions, the <Ha>, <Sa>, or <Da> register.
	• For 2-input scalar versions of the FACGE, FACGT, FCMEQ (register), FCMGE (register), and FCMGT (register) instructions, the <Hm>, <Sm>, or <Dm> register.
	• For 2-input scalar versions of the FABD, FADD (scalar), FDIV (scalar), FMAX (scalar), FMAXNM (scalar), FMIN (scalar), FMINNM (scalar), FMUL (by element), FMUL (scalar), FMULX (by element), FMULX, FNMUL (scalar), FRECPS, FRSQRTS, and FSUB (scalar) instructions, the <Hn>, <Sn>, or <Dn> register.
	• For 1-input scalar versions of the following instructions, the <Hd>, <Sd>, or <Dd> register:
	  • The (vector) versions of the FCVTAS, FCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, and FCVTPU instructions.
	  • The (vector, fixed-point) and (vector, integer) versions of the FCVTZS, FCVTZU, SCVTF, and UCVTF instructions.
	  • The (scalar) versions of the FABS, FNEG, FRINT32X, FRINT32Z, FRINT64X, FRINT64Z, FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ, and FSQRT instructions.
	  • The (scalar, fixed-point) and (scalar, integer) versions of the SCVTF and UCVTF instructions.
	  • The BFCVT, FCVT, FCVTXN, FRECPE, FRECPX, and FRSQRTE instructions.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.NEP is treated as 0 for all purposes other than a direct read or write of the FPCR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AH, bit [1]

When FEAT_AFP is implemented:

Alternate Handling. Controls alternate handling of floating-point numbers.

The Arm architecture supports two models for handling some of the corner cases of the floating-point behaviors, such as the nature of flushing of denormalized numbers, the detection of tininess and other exceptions and a range of other behaviors. The value of the FPCR.AH bit selects between these models.

For more information on the FPCR.AH bit, see 'Flushing denormalized numbers to zero', 'Floating-point exceptions and exception traps' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FIZ, bit [0]

When FEAT_AFP is implemented:

Flush Inputs to Zero. Controls whether single-precision, double-precision and BFloat16 input operands that are denormalized numbers are flushed to zero.

FIZ	Meaning
0b0	The flushing to zero of single-precision and double-precision denormalized inputs to floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.
0b1	Denormalized single-precision and double-precision inputs to most floating-point instructions flushed to zero.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing FPCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPCR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x00);
        else
            AArch64_SystemAccessTrap(EL1, 0x07);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().FPEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPCR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif CPACR_EL1().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPCR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        X{64}(t) = FPCR();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
MSR FPCR, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x00);
        else
            AArch64_SystemAccessTrap(EL1, 0x07);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().FPEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif CPACR_EL1().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        FPCR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.9 FPMR, Floating-point Mode Register

The FPMR characteristics are:

Purpose

Controls behaviors of the FP8 instructions.

Configuration

A direct or indirect read of this register occurs in program order relative to a direct write of this register without explicit synchronization.

On entry to or exit from Streaming SVE mode, FPMR is set to 0.

This register is present only when FEAT_FPMR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to FPMR are UNDEFINED.

Attributes

FPMR is a 64-bit register.

Field descriptions

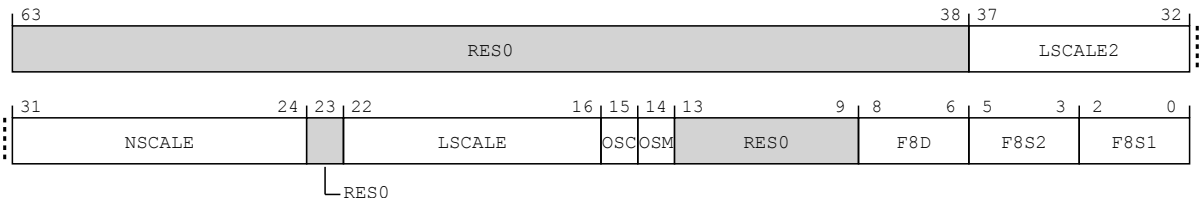

Bits [63:38]

Reserved, RES0.

LSCALE2, bits [37:32]

Downscaling value for instructions that convert the second FP8 input data stream to other floating-point formats.

This value is an unsigned integer that is subtracted from the result exponent.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSCALE, bits [31:24]

Scaling value for instructions that convert other floating-point formats to an FP8 format.

This value is a signed integer that is added to the operand exponent.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [23]

Reserved, RES0.

LSCALE, bits [22:16]

Downscaling value.

This value is an unsigned integer that is subtracted from:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The product or the sum-of-products exponent, for multiplication instructions with FP8 operands.
- The result exponent, for instructions that convert the first FP8 input data stream to other floating-point formats.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OSC, bit [15]

Overflow saturation for FP8 convert instructions. Specifies the result when a floating-point Overflow exception is detected.

OSC	Meaning
`0b0`	Infinity or NaN is generated.
`0b1`	Maximum normal number is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OSM, bit [14]

Overflow saturation for FP8 multiplication instructions. Specifies the result when a floating-point Overflow exception is detected.

OSM	Meaning
`0b0`	Infinity is generated.
`0b1`	Maximum normal number is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [13:9]

Reserved, RES0.

F8D, bits [8:6]

Destination result format for instructions that convert other floating-point values to an FP8 format.

F8D	Meaning
`0b000`	OFP8 E5M2 format.
`0b001`	OFP8 E4M3 format.

All other values are reserved.

Reserved values identify an unsupported format and behave as described in Reserved values in System and memory-mapped registers and translation table entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additionally, FP8 instructions are permitted to set an FP8 result with an unsupported format to `0xFF` and signal an Invalid Operation floating-point exception.

It is software's responsibility to check that a format value is supported in ID_AA64FPFR0_EL1[7:0], before writing it to this field.

For more information about the FP8 formats, see the *OCP 8-bit Floating Point Specification (OFP8)*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F8S2, bits [5:3]

Second FP8 input data stream format for multiplication instructions with FP8 operands, and the corresponding instructions that convert an FP8 format to other floating-point formats.

F8S2	Meaning
`0b000`	OFP8 E5M2 format.
`0b001`	OFP8 E4M3 format.

All other values are reserved.

Reserved values identify an unsupported format and behave as described in Reserved values in System and memory-mapped registers and translation table entries.

Additionally FP8 instructions are permitted to treat FP8 input values with an unsupported format as a signaling NaN.

It is software's responsibility to check that a format value is supported in ID_AA64FPFR0_EL1[7:0], before writing it to this field.

For more information about the FP8 formats, see the *OCP 8-bit Floating Point Specification (OFP8)*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F8S1, bits [2:0]

First FP8 input data stream format for multiplication instructions with FP8 operands, and the corresponding instructions that convert an FP8 format to other floating-point formats.

F8S1	Meaning
`0b000`	OFP8 E5M2 format.
`0b001`	OFP8 E4M3 format.

All other values are reserved.

Reserved values identify an unsupported format and behave as described in Reserved values in System and memory-mapped registers and translation table entries.

Additionally FP8 instructions are permitted to treat FP8 input values with an unsupported format as a signaling NaN.

It is software's responsibility to check that a format value is supported in ID_AA64FPFR0_EL1[7:0], before writing it to this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the FP8 formats, see the *OCP 8-bit Floating Point Specification (OFP8)*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPMR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPMR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_FPMR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnFPM == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().EnFPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnFPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && (!IsHCRXEL2Enabled() || HCRX_EL2().EnFPM == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnFPM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !ELIsInHost(EL0) && CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x00);
        else
            AArch64_SystemAccessTrap(EL1, 0x07);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().FPEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPMR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnFPM == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && (!IsHCRXEL2Enabled() || HCRX_EL2().EnFPM == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnFPM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif CPACR_EL1().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPMR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnFPM == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnFPM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPMR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        X{64}(t) = FPMR();
    end;
end;
```

```
MSR FPMR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_FPMR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnFPM == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().EnFPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnFPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && (!IsHCRXEL2Enabled() || HCRX_EL2().EnFPM == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnFPM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !ELIsInHost(EL0) && CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x00);
        else
            AArch64_SystemAccessTrap(EL1, 0x07);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().FPEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPMR() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnFPM == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && (!IsHCRXEL2Enabled() || HCRX_EL2().EnFPM == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnFPM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif CPACR_EL1().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x07);
        elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x07);
            end;
        else
            FPMR() = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnFPM == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnFPM == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
            AArch64_SystemAccessTrap(EL2, 0x07);
        elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x07);
        elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x07);
            end;
        else
            FPMR() = X{64}(t);
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TFP == '1' then
            AArch64_SystemAccessTrap(EL3, 0x07);
        else
            FPMR() = X{64}(t);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.10 FPSR, Floating-point Status Register

The FPSR characteristics are:

Purpose

Provides floating-point system status information.

Configuration

On entry to or exit from Streaming SVE mode, FPSR.{IOC, DZC, OFC, UFC, IXC, IDC, QC} are set to 1 and the remaining bits are set to 0.

AArch64 System register FPSR bits [31:27] are architecturally mapped to AArch32 System register FPSCR[31:27].

AArch64 System register FPSR bit [7] is architecturally mapped to AArch32 System register FPSCR[7].

AArch64 System register FPSR bits [4:0] are architecturally mapped to AArch32 System register FPSCR[4:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to FPSR are UNDEFINED.

Attributes

FPSR is a 64-bit register.

Field descriptions

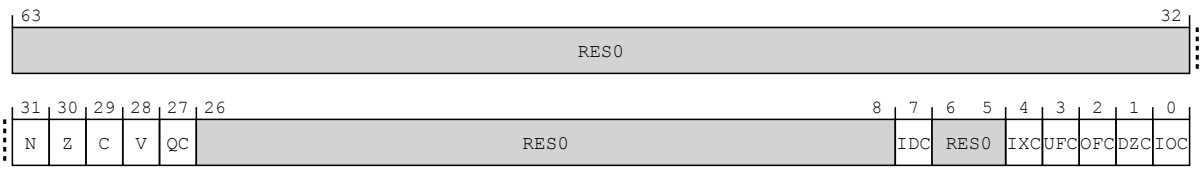

Bits [63:32]

Reserved, RES0.

N, bit [31]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Negative condition flag for AArch32 floating-point comparison operations.

Note

AArch64 floating-point comparisons set the PSTATE.N flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Z, bit [30]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Zero condition flag for AArch32 floating-point comparison operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

AArch64 floating-point comparisons set the PSTATE.Z flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [29]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Carry condition flag for AArch32 floating-point comparison operations.

Note

AArch64 floating-point comparisons set the PSTATE.C flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

V, bit [28]

When FEAT_AA32 is implemented and FEAT_FP is implemented:

Overflow condition flag for AArch32 floating-point comparison operations.

Note

AArch64 floating-point comparisons set the PSTATE.V flag instead.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:8]

Reserved, RES0.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IDE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IDE is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IXE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IXE is 0.

The criteria for the Inexact floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.UFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.UFE is 0 or if flushing denormalized numbers to zero is enabled.

The criteria for the Underflow floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.OFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.OFE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.DZE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.DZE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IOE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IOE is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The criteria for the Invalid Operation floating-point exception to occur are affected by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPSR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x00);
        else
            AArch64_SystemAccessTrap(EL1, 0x07);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().FPEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPSR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif CPACR_EL1().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPSR();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPSR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        X{64}(t) = FPSR();
    end;
end;
```

```
MSR FPSR, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x00);
        else
            AArch64_SystemAccessTrap(EL1, 0x07);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().FPEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPSR() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif CPACR_EL1().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPSR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPSR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        FPSR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.11 NZCV, Condition Flags

The NZCV characteristics are:

Purpose

Allows access to the condition flags.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to NZCV are UNDEFINED.

Attributes

NZCV is a 64-bit register.

Field descriptions

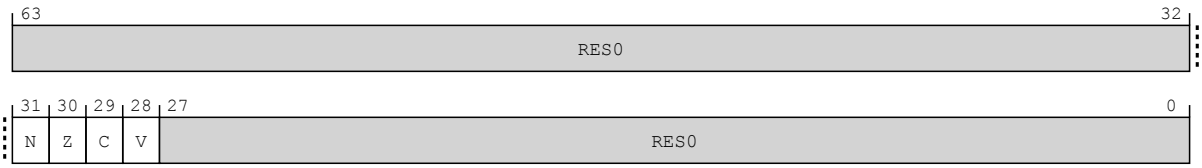

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative condition flag. Set to 1 if the result of the last flag-setting instruction was negative.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Bits [27:0]

Reserved, RES0.

Accessing NZCV

The MSR (register) and MRS instructions used to access NZCV are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, NZCV
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    X{64}(t) = Zeros{32} :: PSTATE.[N,Z,C,V] :: Zeros{28};
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{32} :: PSTATE.[N,Z,C,V] :: Zeros{28};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{32} :: PSTATE.[N,Z,C,V] :: Zeros{28};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{32} :: PSTATE.[N,Z,C,V] :: Zeros{28};
end;
```

```
MSR NZCV, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    PSTATE.[N,Z,C,V] = X{64}(t)[31:28];
elsif PSTATE.EL == EL1 then
    PSTATE.[N,Z,C,V] = X{64}(t)[31:28];
elsif PSTATE.EL == EL2 then
    PSTATE.[N,Z,C,V] = X{64}(t)[31:28];
elsif PSTATE.EL == EL3 then
    PSTATE.[N,Z,C,V] = X{64}(t)[31:28];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.12 PAN, Privileged Access Never

The PAN characteristics are:

Purpose

Allows access to the Privileged Access Never bit.

Configuration

This register is present only when FEAT_PAN is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PAN are UNDEFINED.

Attributes

PAN is a 64-bit register.

Field descriptions

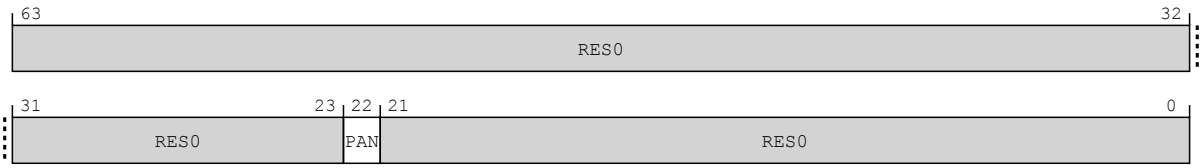

Bits [63:23]

Reserved, RES0.

PAN, bit [22]

Privileged Access Never.

PAN	Meaning
0b0	Privileged reads and write are not disabled by this mechanism.
0b1	Disables privileged read and write accesses to addresses accessible at EL0 for an enabled stage 1 translation regime that defines the EL0 permissions.

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the SCTLR_EL1.SPAN bit is 0, this bit is set to 1.
- When the target of the exception is EL2, the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and the value of the SCTLR_EL2.SPAN bit is 0, this bit is set to 1.
- When the target of the exception is EL2, the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}, and the value of the SCTLR_EL2.SPAN bit is 0, it is IMPLEMENTATION SPECIFIC whether this bit is set to 1. Arm recommends that it is set to 1 in this case.

Bits [21:0]

Reserved, RES0.

Accessing PAN

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PAN
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_PAN) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{41} :: PSTATE.PAN :: Zeros{22};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{41} :: PSTATE.PAN :: Zeros{22};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{41} :: PSTATE.PAN :: Zeros{22};
end;
```

```
MSR PAN, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_PAN) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    PSTATE.PAN = X{64}(t)[22];
elsif PSTATE.EL == EL2 then
    PSTATE.PAN = X{64}(t)[22];
elsif PSTATE.EL == EL3 then
    PSTATE.PAN = X{64}(t)[22];
end;
```

```
MSR PAN, #<imm>
```

op0	op1	CRn	op2
0b00	0b000	0b0100	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.13 PM, Profiling Exception Mask

The PM characteristics are:

Purpose

Allows access to the Profiling exception Mask bit.

Configuration

This register is present only when FEAT_EBEP is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PM are UNDEFINED.

Attributes

PM is a 64-bit register.

Field descriptions

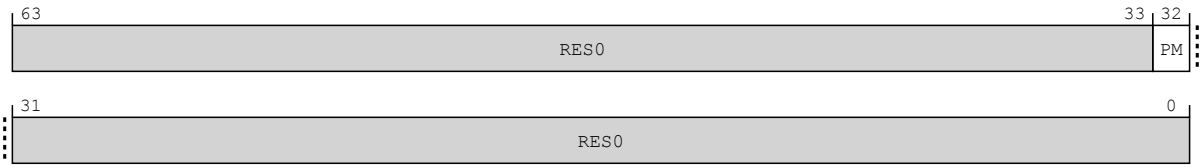

Bits [63:33]

Reserved, RES0.

PM, bit [32]

PMU Exception Mask.

PM	Meaning
0b0	Does not cause the Profiling exception to be masked.
0b1	Causes the Profiling exception to be masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

Accessing PM

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PM
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_EBEP) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{31} :: PSTATE.PM :: Zeros{32};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{31} :: PSTATE.PM :: Zeros{32};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{31} :: PSTATE.PM :: Zeros{32};
end;
```

```
MSR PM, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_EBEP) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    PSTATE.PM = X{64}(t)[32];
elsif PSTATE.EL == EL2 then
    PSTATE.PM = X{64}(t)[32];
elsif PSTATE.EL == EL3 then
    PSTATE.PM = X{64}(t)[32];
end;
```

```
MSR PM, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b001	0b0100	0b001x	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.14 SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

Purpose

Holds the stack pointer associated with EL0. At higher Exception levels, this is used as the current stack pointer when the value of SPSel.SP is 0.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL0 are UNDEFINED.

Attributes

SP_EL0 is a 64-bit register.

Field descriptions

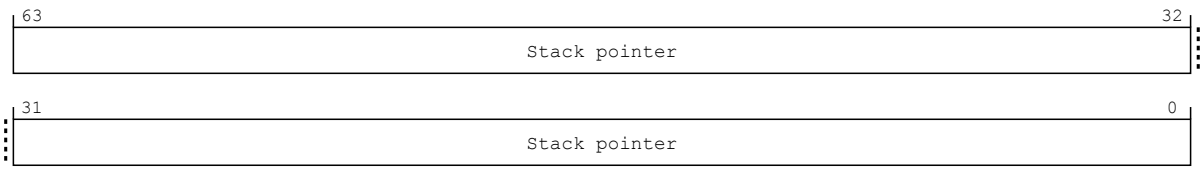

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL0

When the value of PSTATE.SP is 0, this register is accessible at all Exception levels as the current stack pointer.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        Undefined();
    else
        X{64}(t) = SP_EL0();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if PSTATE.SP == '0' then
        Undefined();
    else
        X{64}(t) = SP_EL0();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        Undefined();
    else
        X{64}(t) = SP_EL0();
    end;
end;
```

```
MSR SP_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        Undefined();
    else
        SP_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        Undefined();
    else
        SP_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        Undefined();
    else
        SP_EL0() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.15 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

Purpose

Holds the stack pointer associated with EL1. The value of SPSel.SP determines the current stack pointer.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL1 are UNDEFINED.

Attributes

SP_EL1 is a 64-bit register.

Field descriptions

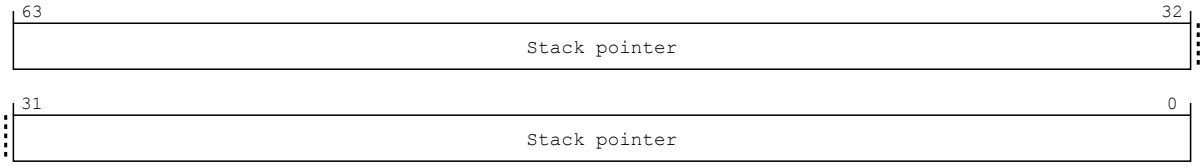

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL1

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x240);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SP_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SP_EL1();
end;
```

```
MSR SP_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x240) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    SP_EL1() = X{64}(t);
elsif PSTATE.EL == EL3 then
    SP_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.16 SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

Purpose

Holds the stack pointer associated with EL2. The value of SPSel.SP determines the current stack pointer.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL2 are UNDEFINED.

Attributes

SP_EL2 is a 64-bit register.

Field descriptions

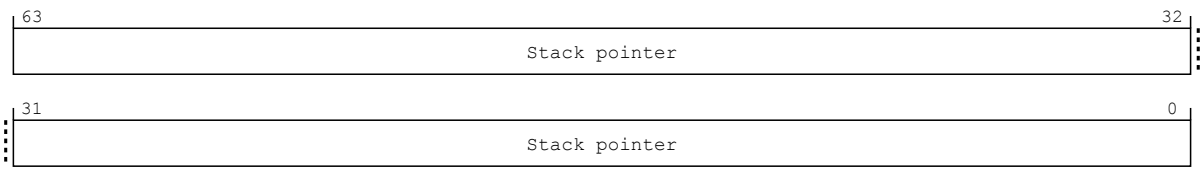

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL2

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SP_EL2();
end;
```

```
MSR SP_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    SP_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.17 SP_EL3, Stack Pointer (EL3)

The SP_EL3 characteristics are:

Purpose

Holds the stack pointer associated with EL3. The value of SPSel.SP determines the current stack pointer.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SP_EL3 are UNDEFINED.

Attributes

SP_EL3 is a 64-bit register.

Field descriptions

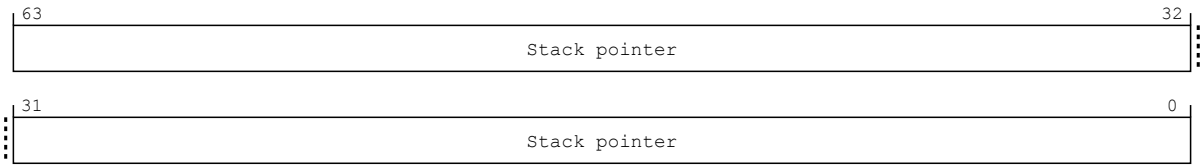

StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL3

This register is not accessible using MRS and MSR instructions.

When the value of SPSel.SP is 1, this register is accessible at EL3 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.18 SPSel, Stack Pointer Select

The SPSel characteristics are:

Purpose

Allows the Stack Pointer to be selected between SP_EL0 and SP_ELx.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSel are UNDEFINED.

Attributes

SPSel is a 64-bit register.

Field descriptions

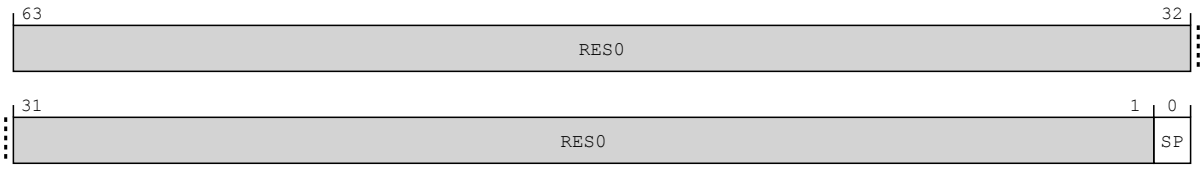

Bits [63:1]

Reserved, RES0.

SP, bit [0]

Stack pointer to use. Possible values of this bit are:

SP	Meaning
0b0	Use SP_EL0 at all Exception levels.
0b1	Use SP_ELx for Exception level ELx. When FEAT_NMI is implemented and SCTLR_ELx.SPINTMASK is 1, if execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx is masked regardless of any indication of Superpriority.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Accessing SPSel

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSel
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{63} :: PSTATE.SP;
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{63} :: PSTATE.SP;
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{63} :: PSTATE.SP;
end;
```

```
MSR SPSel, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    PSTATE.SP = X{64}(t)[0];
elsif PSTATE.EL == EL2 then
    PSTATE.SP = X{64}(t)[0];
elsif PSTATE.EL == EL3 then
    PSTATE.SP = X{64}(t)[0];
end;
```

```
MSR SPSel, #<imm>
```

op0	op1	CRn	op2
0b00	0b000	0b0100	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.19 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_abt bits [31:0] are architecturally mapped to AArch32 System register SPSR_abt[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_abt are UNDEFINED.

Attributes

SPSR_abt is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

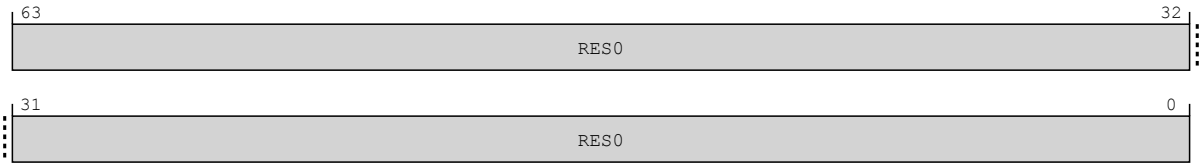

Bits [63:0]

Reserved, RES0.

Otherwise:

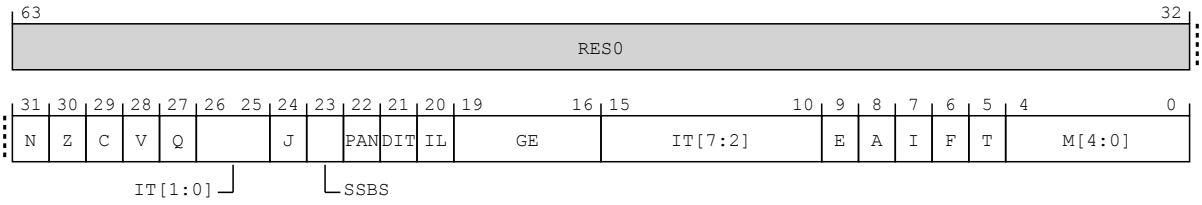

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Abort mode, and copied to PSTATE.IT on executing an exception return operation in Abort mode.

On executing an exception return operation in Abort mode, SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and copied to PSTATE.SSBS on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and copied to PSTATE.PAN on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in ‘Illegal return events from AArch32 state’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_abt

The MSR (register) and MRS instructions used to access SPSR_abt are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_abt
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SPSR_abt();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_abt();
end;
```

```
MSR SPSR_abt, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    SPSR_abt() = X{64}(t);
elsif PSTATE.EL == EL3 then
    SPSR_abt() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.20 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL1.

Configuration

AArch64 System register SPSR_EL1 bits [31:0] are architecturally mapped to AArch32 System register SPSR_svc[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_EL1 are UNDEFINED.

Attributes

SPSR_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exception taken from AArch32 state:

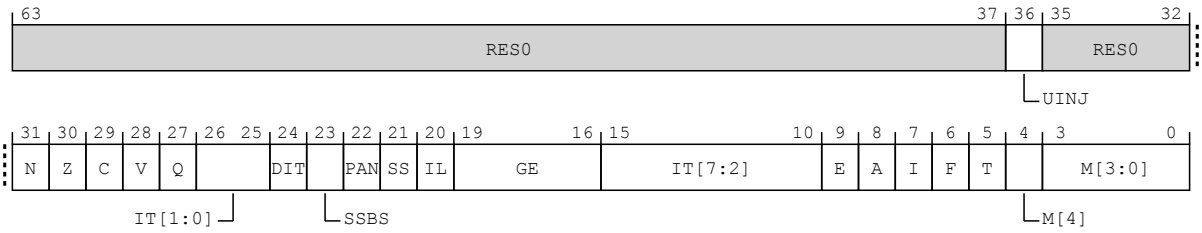

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL1, and copied to PSTATE.UINJ on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL1, and copied to PSTATE.Q on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to EL1, and copied to PSTATE.IT on executing an exception return operation in EL1.

On executing an exception return operation in EL1, SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL1, and copied to PSTATE.GE on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL1, and copied to PSTATE.E on executing an exception return operation in EL1.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL1, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL1, and copied to PSTATE.T on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 1, the value of PSTATE.nRW, on taking an exception to EL1 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

M[4]	Meaning
`0b1`	AArch32 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL1, and copied to PSTATE.M[3:0] on executing an exception return operation in EL1.

M[3:0]	Meaning
`0b0000`	User.
`0b0001`	FIQ.
`0b0010`	IRQ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[3:0]	Meaning
0b0011	Supervisor.
0b0111	Abort.
0b1011	Undefined.
0b1111	System.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

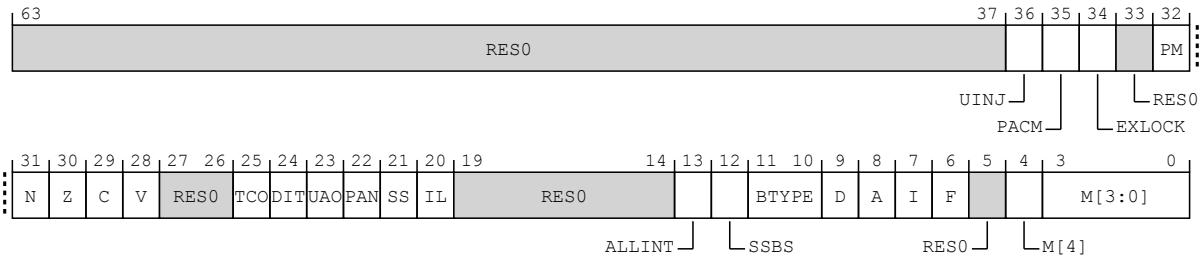

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL1, and copied to PSTATE.UINJ on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on taking an exception to EL1, and copied to PSTATE.PACM on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on taking an exception to EL1, and copied to PSTATE.EXLOCK on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [33]

Reserved, RES0.

PM, bit [32]

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on taking an exception to EL1, and copied to PSTATE.PM on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL1, and copied to PSTATE.TCO on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL1, and copied to PSTATE.UAO on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL1, and copied to PSTATE.ALLINT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL1, and copied to PSTATE.BTYPE on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL1, and copied to PSTATE.D on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0, the value of PSTATE.nRW, on taking an exception to EL1 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (EL1t).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL1 with SP_EL0 (EL1t) and either HCR_EL2.{NV, NV1} is {1,0} or HCR_EL2.{NV, NV2} is {1,1}.
0b1001	EL1 with SP_EL1 (EL1h) and either HCR_EL2.{NV, NV1} is {1,0} or HCR_EL2.{NV, NV2} is {1,1}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2]: On an exception to EL1:
  - If the exception is taken from EL0:
    - M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
  - If the exception is taken from EL1:
    - If the Effective value of HCR_EL2.{NV, NV1} is not {1,0} and the Effective value of HCR_EL2.{NV, NV2} is not {1,1}, then M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
    - If the Effective value of HCR_EL2.{NV, NV1} is {1,0} or if the Effective value of HCR_EL2.{NV, NV2} is {1,1}, then M[3:2] is set to `0b10`.
  - M[3:2] is copied to PSTATE.EL on executing a legal exception return operation in EL1.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL1 and copied to PSTATE.SP on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `SPSR_EL1` or `SPSR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

The MSR (register) and MRS instructions used to access SPSR_EL1 are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x160);
    else
        X{64}(t) = SPSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SPSR_EL2();
    else
        X{64}(t) = SPSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SPSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪!(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x160) = X{64}(t);
    else
        SPSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        SPSR_EL2() = X{64}(t);
    else
        SPSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPSR_EL1() = X{64}(t);
end;
```

```
When FEAT_VHE is implemented MRS <Xt>, SPSR_EL12
```

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x160);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SPSR_EL1();
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = SPSR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR SPSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EffectiveHCR_EL2_NVx() == '101' then
            NVMem(0x160) = X{64}(t);
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            SPSR_EL1() = X{64}(t);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            SPSR_EL1() = X{64}(t);
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MRS <Xt>, SPSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EffectiveHCR_EL2_NVx() IN {'1x1'} then
            X{64}(t) = SPSR_EL1();
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SPSR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_EL2();
end;
```

When FEAT_VHE is implemented MSR SPSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        SPSR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪then
        EXLOCKException();
    else
        SPSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPSR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.21 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL2.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register SPSR_EL2 bits [31:0] are architecturally mapped to AArch32 System register SPSR_hyp[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_EL2 are UNDEFINED.

Attributes

SPSR_EL2 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exception taken from AArch32 state:

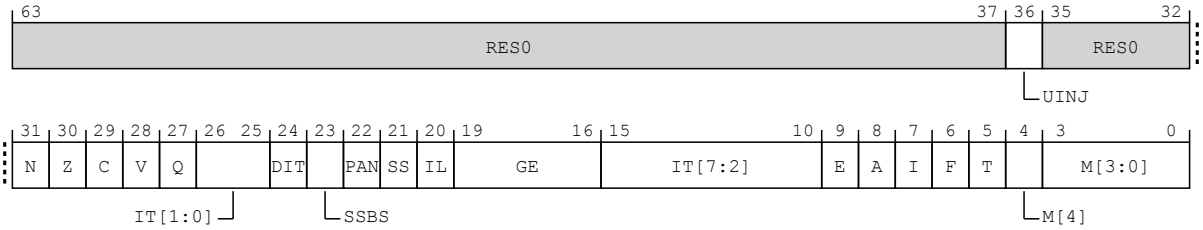

An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL2, and copied to PSTATE.UINJ on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL2, and copied to PSTATE.Q on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to EL2, and copied to PSTATE.IT on executing an exception return operation in EL2.

On executing an exception return operation in EL2, SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL2, and copied to PSTATE.GE on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL2, and copied to PSTATE.E on executing an exception return operation in EL2.

If the implementation does not support big-endian operation, SPSR_EL2.E is RES0. If the implementation does not support little-endian operation, SPSR_EL2.E is RES1. On executing an exception return operation in EL2, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL2.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL2.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL2, and copied to PSTATE.T on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 1, the value of PSTATE.nRW, on taking an exception to EL2 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

M[4]	Meaning
`0b1`	AArch32 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL2, and copied to PSTATE.M[3:0] on executing an exception return operation in EL2.

M[3:0]	Meaning
`0b0000`	User.
`0b0001`	FIQ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[3:0]	Meaning
0b0010	IRQ.
0b0011	Supervisor.
0b0111	Abort.
0b1010	Hyp.
0b1011	Undefined.
0b1111	System.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

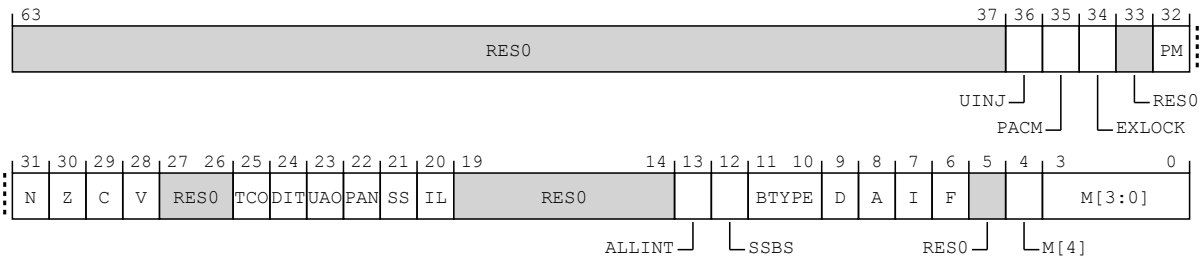

An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL2, and copied to PSTATE.UINJ on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on taking an exception to EL2, and copied to PSTATE.PACM on executing an exception return operation in EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on taking an exception to EL2, and copied to PSTATE.EXLOCK on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [33]

Reserved, RES0.

PM, bit [32]

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on taking an exception to EL2, and copied to PSTATE.PM on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL2, and copied to PSTATE.TCO on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL2, and copied to PSTATE.UAO on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL2, and copied to PSTATE.ALLINT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL2, and copied to PSTATE.BTYPE on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL2, and copied to PSTATE.D on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0, the value of PSTATE.nRW, on taking an exception to EL2 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

M[4]	Meaning
`0b0`	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
`0b0000`	EL0.
`0b0100`	EL1 with SP_EL0 (EL1t).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[3:0]	Meaning
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL2 with SP_EL0 (EL2t).
0b1001	EL2 with SP_EL2 (EL2h).

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL2 and copied to PSTATE.EL on executing an exception return operation in EL2.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL2 and copied to PSTATE.SP on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `SPSR_EL2` or `SPSR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

The MSR (register) and MRS instructions used to access SPSR_EL2 are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = SPSR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SPSR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SPSR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' &&
        ↪EffectiveHCR_EL2_NVx() IN {'xx1'} then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        SPSR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        SPSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPSR_EL2() = X{64}(t);
end;
```

```
When FEAT_VHE is implemented MRS <Xt>, SPSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x160);
    else
        X{64}(t) = SPSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SPSR_EL2();
    else
        X{64}(t) = SPSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_VHE is implemented MSR SPSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& !(EffectiveHCR_EL2_NVx() IN {'x11'}) then
        EXLOCKException();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x160) = X{64}(t);
    else
        SPSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1'
        ↪&& ELIsInHost(EL2) then
        EXLOCKException();
    elsif ELIsInHost(EL2) then
        SPSR_EL2() = X{64}(t);
    else
        SPSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPSR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.22 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_EL3 are UNDEFINED.

Attributes

SPSR_EL3 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exception taken from AArch32 state:

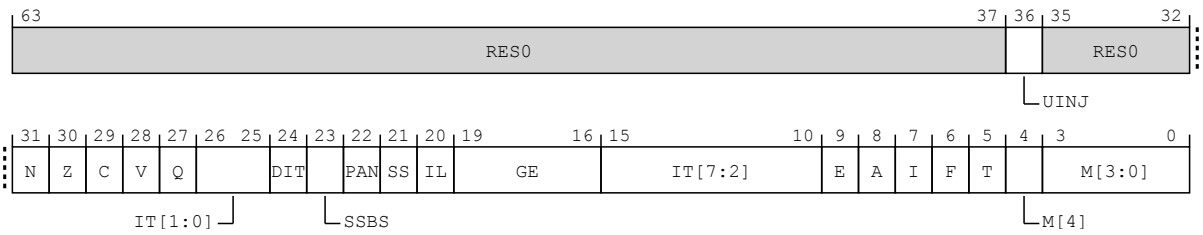

An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL3, and copied to PSTATE.UINJ on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL3, and copied to PSTATE.Q on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to EL3, and copied to PSTATE.IT on executing an exception return operation in EL3.

On executing an exception return operation in EL3, SPSR_EL3.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL3, and copied to PSTATE.GE on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL3, and copied to PSTATE.E on executing an exception return operation in EL3.

If the implementation does not support big-endian operation, SPSR_EL3.E is RES0. If the implementation does not support little-endian operation, SPSR_EL3.E is RES1. On executing an exception return operation in EL3, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL3.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL3.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL3, and copied to PSTATE.T on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 1, the value of PSTATE.nRW, on taking an exception to EL3 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

M[4]	Meaning
`0b1`	AArch32 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL3, and copied to PSTATE.M[3:0] on executing an exception return operation in EL3.

M[3:0]	Meaning	Applies when
`0b0000`	User.	
`0b0001`	FIQ.	
`0b0010`	IRQ.	
`0b0011`	Supervisor.	
`0b0110`	Monitor.	FEAT_EL3 is implemented
`0b0111`	Abort.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[3:0]	Meaning	Applies when
0b1010	Hyp.	
0b1011	Undefined.	
0b1111	System.	

Other values are reserved. If SPSR_EL3.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in ‘Illegal return events from AArch64 state’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

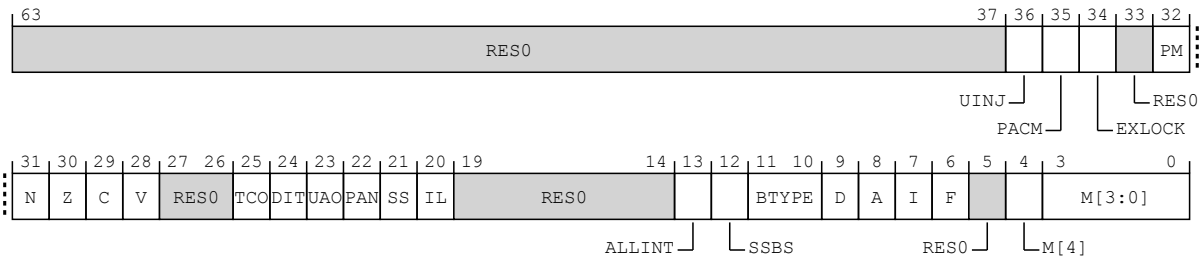

An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to 0 on taking an exception to EL3, and copied to PSTATE.UINJ on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on taking an exception to EL3, and copied to PSTATE.PACM on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on taking an exception to EL3, and copied to PSTATE.EXLOCK on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [33]

Reserved, RES0.

PM, bit [32]

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on taking an exception to EL3, and copied to PSTATE.PM on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL3, and copied to PSTATE.TCO on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL3, and copied to PSTATE.UAO on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL3, and copied to PSTATE.ALLINT on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL3, and copied to PSTATE.BTYPE on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL3, and copied to PSTATE.D on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0, the value of PSTATE.nRW, on taking an exception to EL3 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

M[4]	Meaning
`0b0`	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning	Applies when
`0b0000`	EL0.	
`0b0100`	EL1 with SP_EL0 (EL1t).	
`0b0101`	EL1 with SP_EL1 (EL1h).	
`0b1000`	EL2 with SP_EL0 (EL2t).	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[3:0]	Meaning	Applies when
0b1001	EL2 with SP_EL2 (EL2h).	
0b1100	EL3 with SP_EL0 (EL3t).	FEAT_EL3 is implemented
0b1101	EL3 with SP_EL3 (EL3h).	FEAT_EL3 is implemented

Other values are reserved. If SPSR_EL3.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL3 and copied to PSTATE.EL on executing an exception return operation in EL3.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL3 and copied to PSTATE.SP on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL3

The MSR (register) and MRS instructions used to access SPSR_EL3 are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_EL3();
end;
```

```
MSR SPSR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    else
        SPSR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.23 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_fiq bits [31:0] are architecturally mapped to AArch32 System register SPSR_fiq[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_fiq are UNDEFINED.

Attributes

SPSR_fiq is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

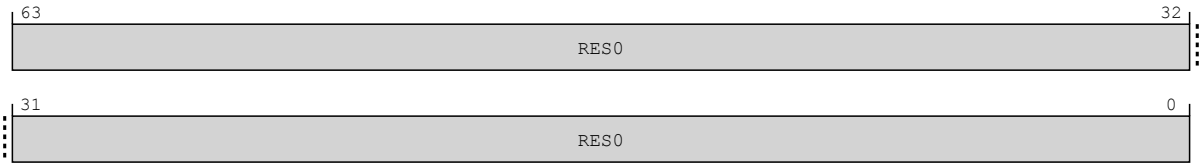

Bits [63:0]

Reserved, RES0.

Otherwise:

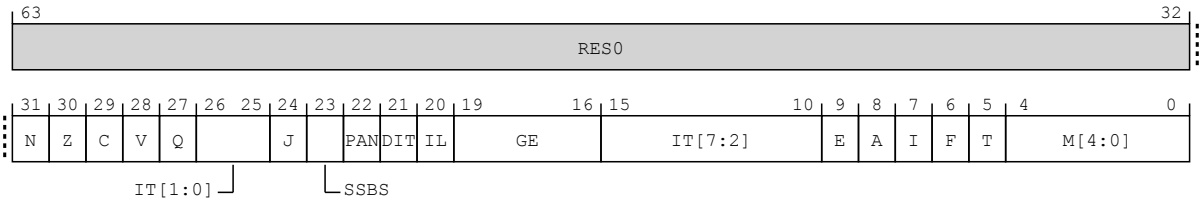

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to FIQ mode, and copied to PSTATE.IT on executing an exception return operation in FIQ mode.

On executing an exception return operation in FIQ mode, SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in ‘Illegal return events from AArch32 state’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_fiq

The MSR (register) and MRS instructions used to access SPSR_fiq are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_fiq
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SPSR_fiq();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_fiq();
end;
```

```
MSR SPSR_fiq, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    SPSR_fiq() = X{64}(t);
elsif PSTATE.EL == EL3 then
    SPSR_fiq() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.24 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_irq bits [31:0] are architecturally mapped to AArch32 System register SPSR_irq[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_irq are UNDEFINED.

Attributes

SPSR_irq is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

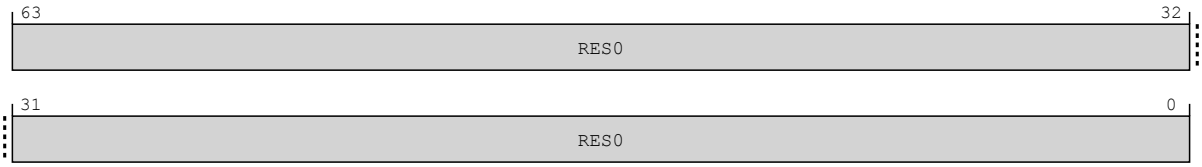

Bits [63:0]

Reserved, RES0.

Otherwise:

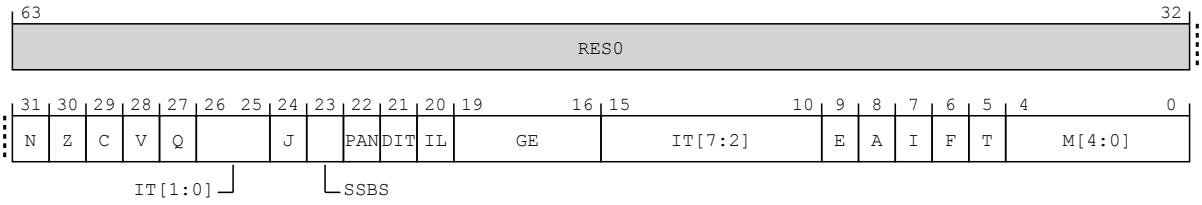

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to IRQ mode, and copied to PSTATE.IT on executing an exception return operation in IRQ mode.

On executing an exception return operation in IRQ mode, SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in ‘Illegal return events from AArch32 state’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_irq

The MSR (register) and MRS instructions used to access SPSR_irq are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_irq
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SPSR_irq();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_irq();
end;
```

```
MSR SPSR_irq, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    SPSR_irq() = X{64}(t);
elsif PSTATE.EL == EL3 then
    SPSR_irq() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.25 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configuration

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

AArch64 System register SPSR_und bits [31:0] are architecturally mapped to AArch32 System register SPSR_und[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SPSR_und are UNDEFINED.

Attributes

SPSR_und is a 64-bit register.

Field descriptions

When FEAT_AA32EL1 is not implemented:

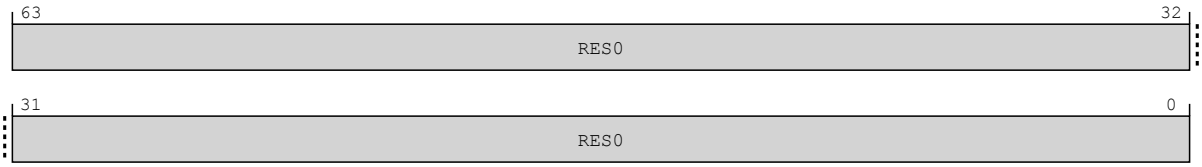

Bits [63:0]

Reserved, RES0.

Otherwise:

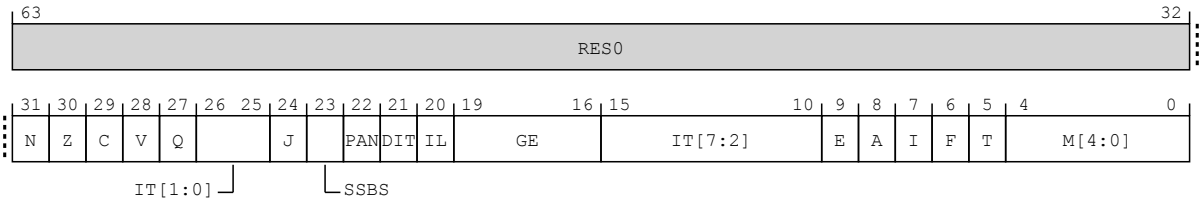

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Undefined mode, and copied to PSTATE.IT on executing an exception return operation in Undefined mode.

On executing an exception return operation in Undefined mode, SPSR_und.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

M[4:0]	Meaning
`0b10000`	User.
`0b10001`	FIQ.
`0b10010`	IRQ.
`0b10011`	Supervisor.
`0b10111`	Abort.
`0b11011`	Undefined.
`0b11111`	System.

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_und

The MSR (register) and MRS instructions used to access SPSR_und are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPSR_und
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SPSR_und();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPSR_und();
end;
```

```
MSR SPSR_und, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    SPSR_und() = X{64}(t);
elsif PSTATE.EL == EL3 then
    SPSR_und() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.26 SSBS, Speculative Store Bypass Safe

The SSBS characteristics are:

Purpose

Allows access to the Speculative Store Bypass Safe bit.

Configuration

This register is present only when FEAT_SSBS2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SSBS are UNDEFINED.

Attributes

SSBS is a 64-bit register.

Field descriptions

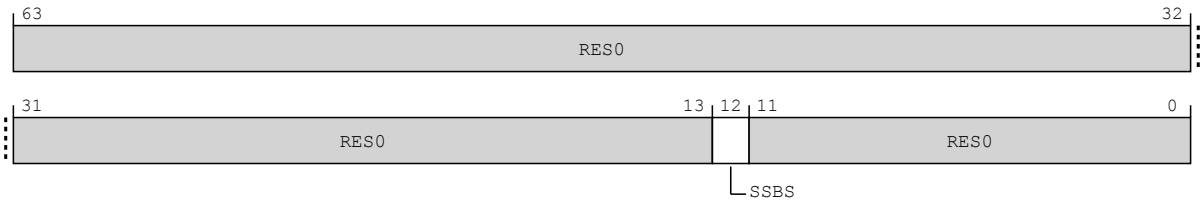

Bits [63:13]

Reserved, RES0.

SSBS, bit [12]

Speculative Store Bypass Safe.

Prohibits speculative loads or stores which might practically allow a cache timing side channel.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

SSBS	Meaning
0b0	Hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.
0b1	When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

The value of this bit is set to the value in the SCTLR_ELx.DSSBS field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [11:0]

Reserved, RES0.

Accessing SSBS

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SSBS
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if !(IsFeatureImplemented(FEAT_SSBS2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    X{64}(t) = Zeros{51} :: PSTATE.SSBS :: Zeros{12};
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{51} :: PSTATE.SSBS :: Zeros{12};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{51} :: PSTATE.SSBS :: Zeros{12};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{51} :: PSTATE.SSBS :: Zeros{12};
end;
```

```
MSR SSBS, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if !(IsFeatureImplemented(FEAT_SSBS2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    PSTATE.SSBS = X{64}(t)[12];
elsif PSTATE.EL == EL1 then
    PSTATE.SSBS = X{64}(t)[12];
elsif PSTATE.EL == EL2 then
    PSTATE.SSBS = X{64}(t)[12];
elsif PSTATE.EL == EL3 then
    PSTATE.SSBS = X{64}(t)[12];
end;
```

```
MSR SSBS, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.27 SVCR, Streaming Vector Control Register

The SVCR characteristics are:

Purpose

Allows access to the Streaming SVE mode in PSTATE.SM, and the SME storage enable in PSTATE.ZA.

Configuration

This register is present only when FEAT_SME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SVCR are UNDEFINED.

Attributes

SVCR is a 64-bit register.

Field descriptions

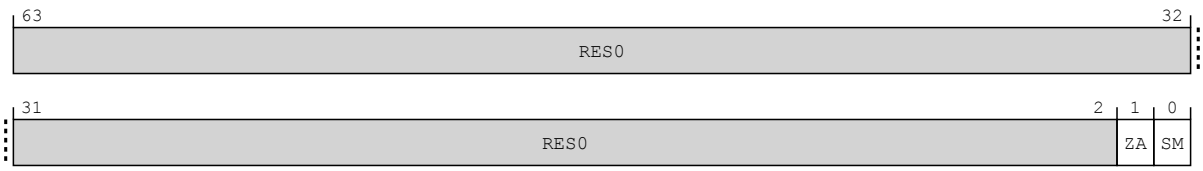

Bits [63:2]

Reserved, RES0.

ZA, bit [1]

Access to PSTATE.ZA, SME storage enable.

When this storage is disabled, execution of an instruction which can access it is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {`0x1D`, `0x3`}.

The possible values of this bit are:

ZA	Meaning
`0b0`	SME ZA storage and, if implemented, ZT0 storage are invalid and not accessible. This control causes execution at any Exception level of instructions that can access this storage to be trapped.
`0b1`	SME ZA storage and, if implemented, ZT0 storage are valid and accessible. This control does not cause execution of any instructions to be trapped.

When a write to SVCR.ZA changes the value of PSTATE.ZA from 0 to 1, all implemented bits of the storage are set to zero.

Changes to this field do not have an effect on the SVE vector and predicate registers and FPSR.

A direct or indirect read of ZA appears to occur in program order relative to a direct write of SVCR, and to `MSR SVCRZA` and `MSR SVCRSMZA` instructions, without the need for explicit synchronization.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, PSTATE.ZA resets to ‘0’.

SM, bit [0]

Access to PSTATE.SM, Streaming SVE mode.

When the PE is in Streaming SVE mode, the Streaming SVE vector length (SVL) applies to SVE instructions, and execution at any Exception level of an instruction which is illegal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {`0x1D`, `0x1`}.

When the PE is not in Streaming SVE mode, the SVE vector length (VL) applies to SVE instructions, and execution at any Exception level of an instruction which is only legal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {`0x1D`, `0x2`}.

The possible values of this bit are:

SM	Meaning
`0b0`	The PE is not in Streaming SVE mode.
`0b1`	The PE is in Streaming SVE mode.

When a write to SVCR.SM changes the value of PSTATE.SM, the following applies:

- When changed from 0 to 1, an entry to Streaming SVE mode is performed.
- When changed from 1 to 0, an exit from Streaming SVE mode is performed.
- All implemented bits of the SVE registers Z0-Z31, P0-P15, and FFR in the new mode are set to zero.
- All bits in FPMR are set to zero.
- FPSR in the new mode is set to `0x0000`_0000_0800_009f, in which all cumulative status bits are set to 1.

Changes to this field do not have an effect on SME ZA storage or, if implemented, ZT0 storage.

A direct or indirect read of SM appears to occur in program order relative to a direct write of SVCR, and to `MSR SVCRSM` and `MSR SVCRSMZA` instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, PSTATE.SM resets to ‘0’.

Accessing SVCR

SVCR is read/write and can be accessed from any Exception level.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SVCR
```

op0	op1	CRn	CRm	op2
`0b11`	`0b011`	`0b0100`	`0b0010`	`0b010`

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().SMEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x1D);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL1, 0x1D);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().SMEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        X{64}(t) = SVCR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif CPACR_EL1().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        X{64}(t) = SVCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        X{64}(t) = SVCR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        X{64}(t) = SVCR();
    end;
end;
```

```
MSR SVCR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().SMEN != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x1D);
        else
            AArch64_SystemAccessTrap(EL1, 0x1D);
        end;
    elsif ELIsInHost(EL0) && CPTR_EL2().SMEN != '11' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        SVCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif CPACR_EL1().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        SVCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        SVCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        SVCR() = X{64}(t);
    end;
end;
```

```
MSR SVCRSM, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b001x	0b011

```
MSR SVCRZA, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b010x	0b011

```
MSR SVCRSMZA, #<imm>
```

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b011x	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.28 TCO, Tag Check Override

The TCO characteristics are:

Purpose

When FEAT_MTE is implemented, this register allows tag checks to be disabled globally.

When FEAT_MTE2 is not implemented, it is IMPLEMENTATION DEFINED if accesses to this register access PSTATE.TCO or are RAZ/WI.

Configuration

This register is present only when FEAT_MTE is implemented. Otherwise, direct accesses to TCO are UNDEFINED.

Attributes

TCO is a 64-bit register.

Field descriptions

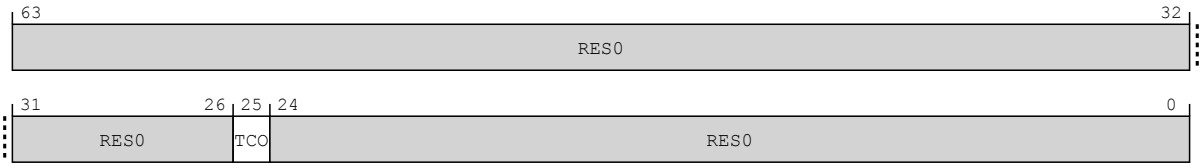

Bits [63:26]

Reserved, RES0.

TCO, bit [25]

Allows memory tag checks to be globally disabled.

TCO	Meaning
0b0	Loads and Stores are not affected by this control.
0b1	Loads and Stores are unchecked.

Bits [24:0]

Reserved, RES0.

Accessing TCO

For information about the operation of the MSR (immediate) accessor, see MSR (immediate).

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCO
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    X{64}(t) = Zeros{38} :: PSTATE.TCO :: Zeros{25};
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{38} :: PSTATE.TCO :: Zeros{25};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{38} :: PSTATE.TCO :: Zeros{25};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{38} :: PSTATE.TCO :: Zeros{25};
end;
```

```
MSR TCO, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    PSTATE.TCO = X{64}(t)[25];
elsif PSTATE.EL == EL1 then
    PSTATE.TCO = X{64}(t)[25];
elsif PSTATE.EL == EL2 then
    PSTATE.TCO = X{64}(t)[25];
elsif PSTATE.EL == EL3 then
    PSTATE.TCO = X{64}(t)[25];
end;
```

```
MSR TCO, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.29 UAO, User Access Override

The UAO characteristics are:

Purpose

Allows access to the User Access Override bit.

Configuration

This register is present only when FEAT_UAO is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to UAO are UNDEFINED.

Attributes

UAO is a 64-bit register.

Field descriptions

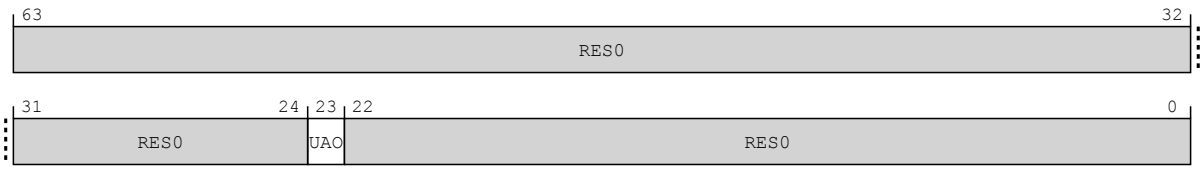

Bits [63:24]

Reserved, RES0.

UAO, bit [23]

User Access Override.

UAO	Meaning
0b0	Explicit Memory Effects generated by unprivileged memory access instructions and described as unprivileged are unprivileged.
0b1	Explicit Memory Effects generated by unprivileged memory access instructions and described as unprivileged are privileged.

The Effective value of PSTATE.UAO is 0 when execution is at EL0.

The Effective value of PSTATE.UAO is 1 when any of the following is true:

- Execution is at EL1, and HCR_EL2.{NV, NV1} is {1, 1}.
- Execution is at EL2, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- Execution is at EL3.

Bits [22:0]

Reserved, RES0.

Accessing UAO

For more information about the operation of the MSR (immediate) accessor, see ‘MSR (immediate)’.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, UAO
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_UAO) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = Zeros{40} :: PSTATE.UAO :: Zeros{23};
elsif PSTATE.EL == EL2 then
    X{64}(t) = Zeros{40} :: PSTATE.UAO :: Zeros{23};
elsif PSTATE.EL == EL3 then
    X{64}(t) = Zeros{40} :: PSTATE.UAO :: Zeros{23};
end;
```

```
MSR UAO, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_UAO) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    PSTATE.UAO = X{64}(t)[23];
elsif PSTATE.EL == EL2 then
    PSTATE.UAO = X{64}(t)[23];
elsif PSTATE.EL == EL3 then
    PSTATE.UAO = X{64}(t)[23];
end;
```

```
MSR UAO, #<imm>
```

op0	op1	CRn	op2
0b00	0b000	0b0100	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3 A64 System instructions for cache maintenance

This section lists the A64 System instructions for cache maintenance.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.1 DC CGDSW, Clean of Data and Allocation Tags by Set/Way

The DC CGDSW characteristics are:

Purpose

Clean data and Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CGDSW are UNDEFINED.

Attributes

DC CGDSW is a 64-bit System instruction.

Field descriptions

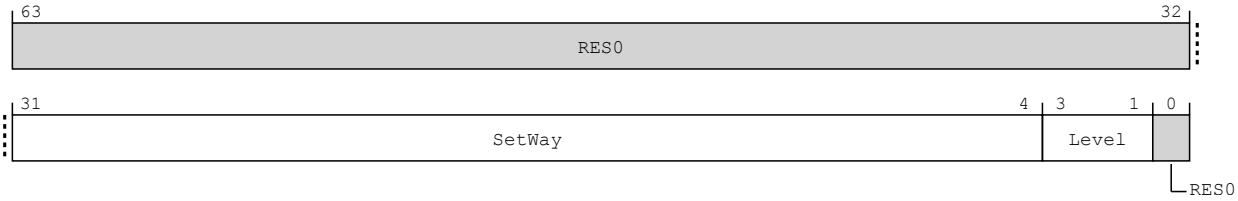

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CGDSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.2 DC CGDVAC, Clean of Data and Allocation Tags by VA to PoC

The DC CGDVAC characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVAC are UNDEFINED.

Attributes

DC CGDVAC is a 64-bit System instruction.

Field descriptions

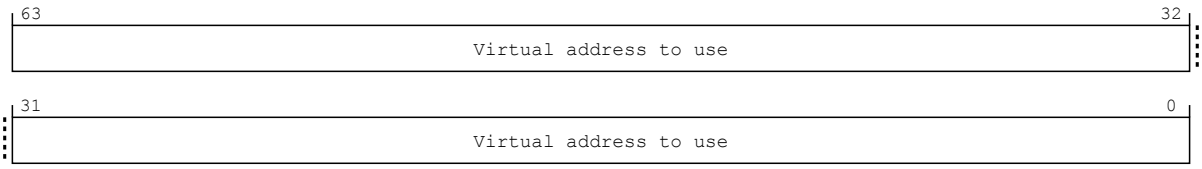

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b101

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.3 DC CGDVADP, Clean of Data and Allocation Tags by VA to PoDP

The DC CGDVADP characteristics are:

Purpose

Clean Allocation Tags and data in data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGDVAP.

Configuration

This system instruction is present only when FEAT_DPB2 is implemented and FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVADP are UNDEFINED.

Attributes

DC CGDVADP is a 64-bit System instruction.

Field descriptions

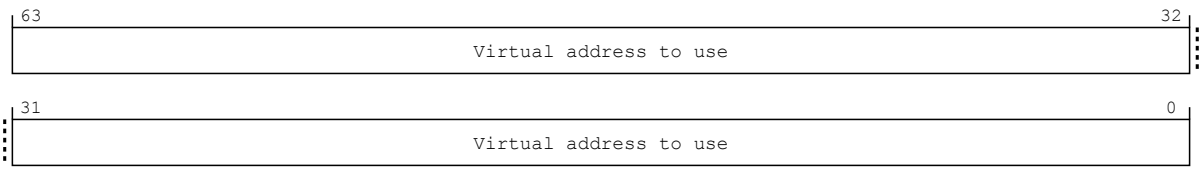

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVADP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b101

```
if !(IsFeatureImplemented(FEAT_DPB2) && IsFeatureImplemented(FEAT_MTE)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVADP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVADP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.4 DC CGDVAOC, Clean of Data and Allocation Tags by VA to Outer Cache level

The DC CGDVAOC characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented, FEAT_MTE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CGDVAOC are UNDEFINED.

Attributes

DC CGDVAOC is a 64-bit System instruction.

Field descriptions

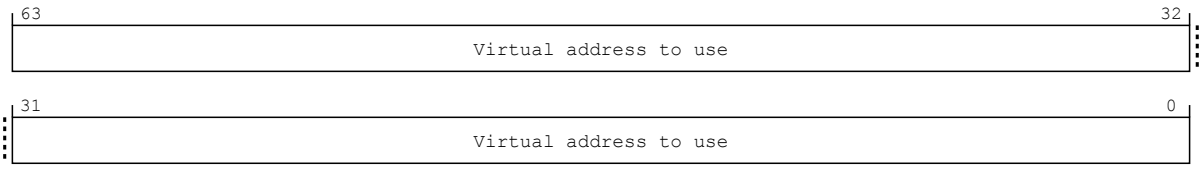

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b111

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_MTE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_OuterCache);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.5 DC CGDVAP, Clean of Data and Allocation Tags by VA to PoP

The DC CGDVAP characteristics are:

Purpose

Clean data and Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGDVAC.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVAP are UNDEFINED.

Attributes

DC CGDVAP is a 64-bit System instruction.

Field descriptions

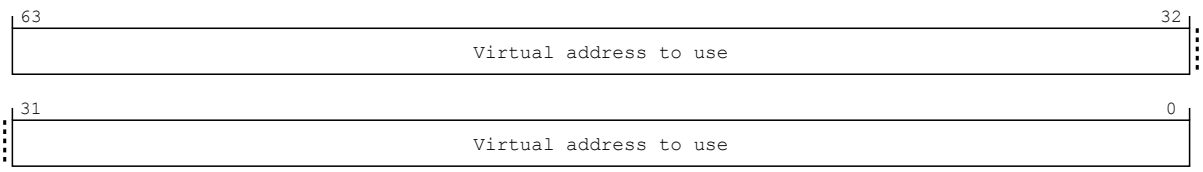

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGDVAP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b101

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGITR_EL2().DCCVAP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.6 DC CGSW, Clean of Allocation Tags by Set/Way

The DC CGSW characteristics are:

Purpose

Clean Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CGSW are UNDEFINED.

Attributes

DC CGSW is a 64-bit System instruction.

Field descriptions

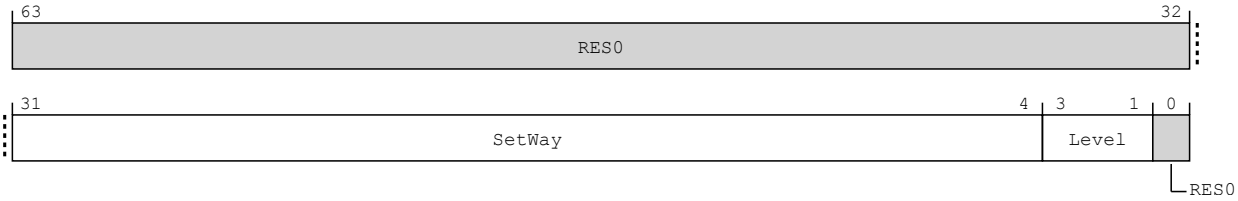

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CGSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.7 DC CGVAC, Clean of Allocation Tags by VA to PoC

The DC CGVAC characteristics are:

Purpose

Clean Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVAC are UNDEFINED.

Attributes

DC CGVAC is a 64-bit System instruction.

Field descriptions

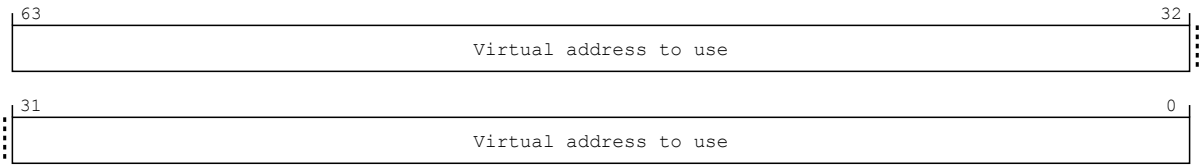

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.8 DC CGVADP, Clean of Allocation Tags by VA to PoDP

The DC CGVADP characteristics are:

Purpose

Clean Allocation tags by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGVAP.

Configuration

This system instruction is present only when FEAT_DPB2 is implemented and FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVADP are UNDEFINED.

Attributes

DC CGVADP is a 64-bit System instruction.

Field descriptions

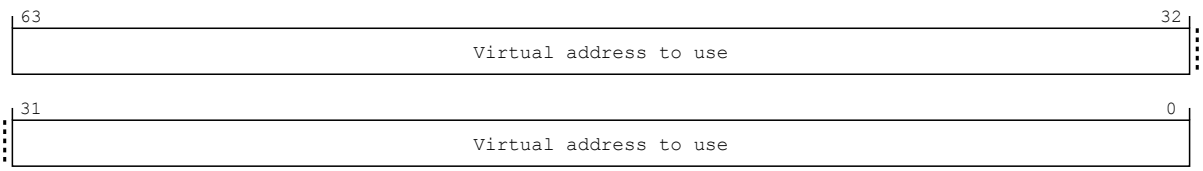

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGVADP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_DPB2) && IsFeatureImplemented(FEAT_MTE)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVADP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGITR_EL2().DCCVADP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.9 DC CGVAP, Clean of Allocation Tags by VA to PoP

The DC CGVAP characteristics are:

Purpose

Clean Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGVAC.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVAP are UNDEFINED.

Attributes

DC CGVAP is a 64-bit System instruction.

Field descriptions

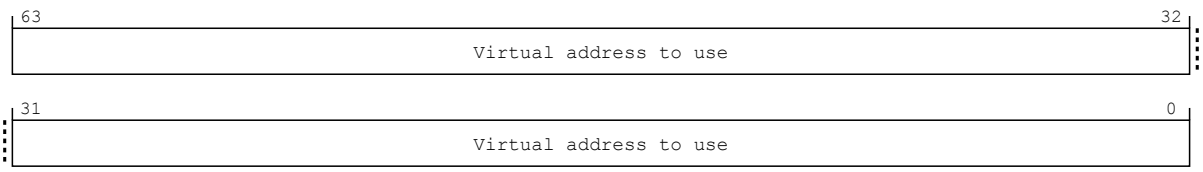

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CGVAP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGITR_EL2().DCCVAP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.10 DC CIGDPAE, Clean and invalidate of data and allocation tags by PA to PoE

The DC CIGDPAE characteristics are:

Purpose

Clean and invalidate of data and allocation tags by PA to PoE.

Configuration

This system instruction is present only when FEAT_MEC is implemented, FEAT_MTE2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDPAE are UNDEFINED.

Attributes

DC CIGDPAE is a 64-bit System instruction.

Field descriptions

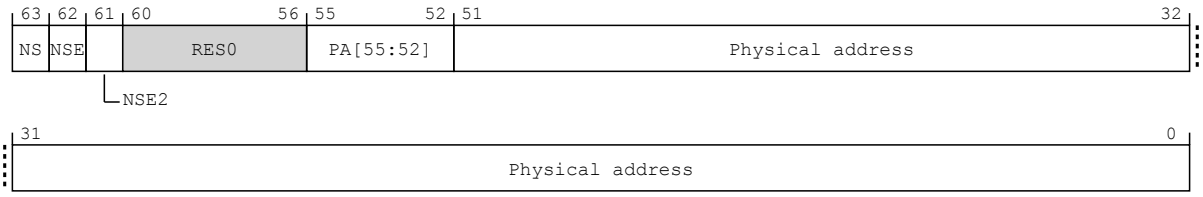

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	Reserved.
0b0	0b0	0b1	Reserved.
0b0	0b1	0b0	Reserved.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b0	0b0	Reserved.
0b0	0b1	Reserved.
0b1	0b0	Reserved.
0b1	0b1	Realm.

If {NSE, NS} is not {1, 1}, then no cache entries are required to be cleaned or invalidated.

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIGDPAE.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIGDPAE.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information, see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIGDPAE

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDPAE, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1110	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_MTE2) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoE);
    end;
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoE);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.11 DC CIGDPAPA, Clean and Invalidate of Data and Allocation Tags by PA to PoPA

The DC CIGDPAPA characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by physical address to the Point of Physical Aliasing.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_RME is implemented, FEAT_MTE2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDPAPA are UNDEFINED.

Attributes

DC CIGDPAPA is a 64-bit System instruction.

Field descriptions

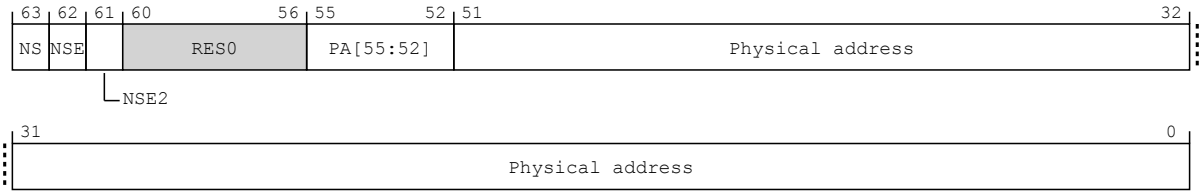

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b0	0b1	Non-secure.
0b0	0b1	0b0	Root.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

If FEAT_SEL2 is not implemented, and {NSE, NS} == {0, 0}, then no cache entries are required to be cleaned or invalidated

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIGDPAPA.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIGDPAPA.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIGDPAPA

- This instruction is not subject to any translation, permission checks, or granule protection checks.
- This instruction affects all caches in the Outer Shareable shareability domain.
- This instruction has the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
DC CIGDPAPA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1110	0b101

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_MTE2) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPA);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.12 DC CIGDSW, Clean and Invalidate of Data and Allocation Tags by Set/Way

The DC CIGDSW characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGDSW are UNDEFINED.

Attributes

DC CIGDSW is a 64-bit System instruction.

Field descriptions

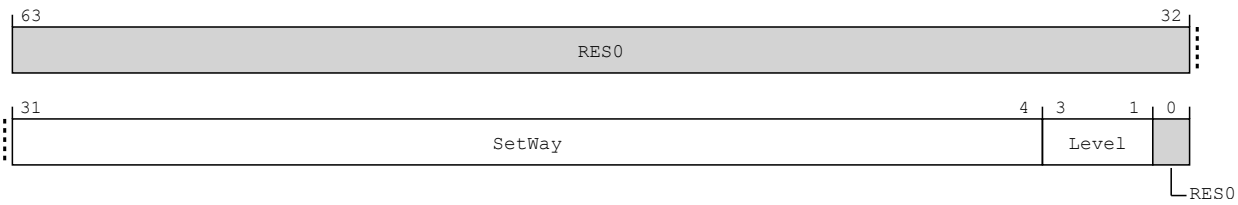

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CIGDSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCISW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.13 DC CIGDVAC, Clean and Invalidate of Data and Allocation Tags by VA to PoC

The DC CIGDVAC characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CIGDVAC are UNDEFINED.

Attributes

DC CIGDVAC is a 64-bit System instruction.

Field descriptions

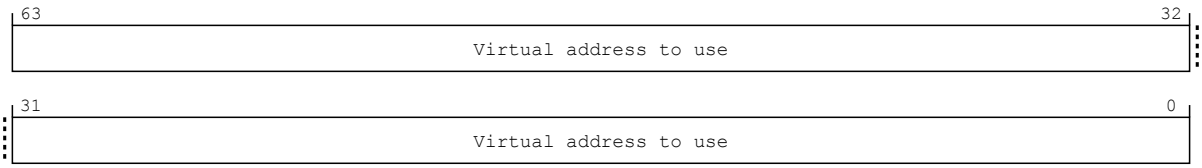

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGDVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b101

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.14 DC CIGDVAOC, Clean and Invalidate of Data and Allocation Tags by VA to Outer Cache level

The DC CIGDVAOC characteristics are:

Purpose

Clean and invalidate of data and Allocation Tags in data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented, FEAT_MTE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDVAOC are UNDEFINED.

Attributes

DC CIGDVAOC is a 64-bit System instruction.

Field descriptions

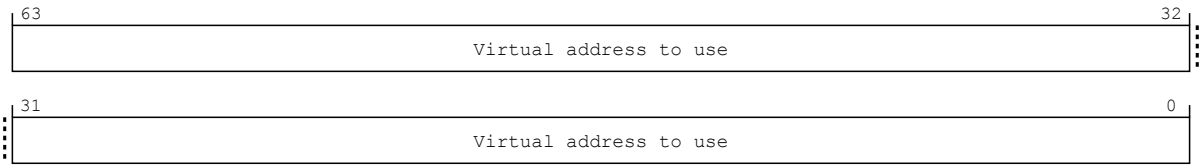

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGDVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1111	0b111

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_MTE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.15 DC CIGDVAPS, Clean and Invalidate of Data and Allocation Tags by VA to PoPS

The DC CIGDVAPS characteristics are:

Purpose

Clean and Invalidate data and Allocation Tags in data cache by virtual address to the Point of Physical Storage.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_PoPS is implemented, FEAT_MTE2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIGDVAPS are UNDEFINED.

Attributes

DC CIGDVAPS is a 64-bit System instruction.

Field descriptions

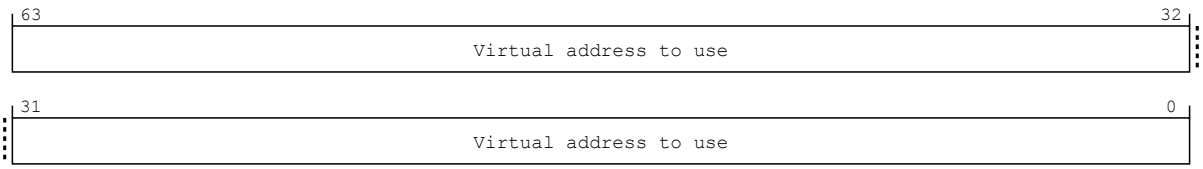

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGDVAPS

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGDVAPS, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1111	0b101

```
if !(IsFeatureImplemented(FEAT_PoPS) && IsFeatureImplemented(FEAT_MTE2) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TPCP == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGITR2_EL2().nDCCIVAPS == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.16 DC CIGSW, Clean and Invalidate of Allocation Tags by Set/Way

The DC CIGSW characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGSW are UNDEFINED.

Attributes

DC CIGSW is a 64-bit System instruction.

Field descriptions

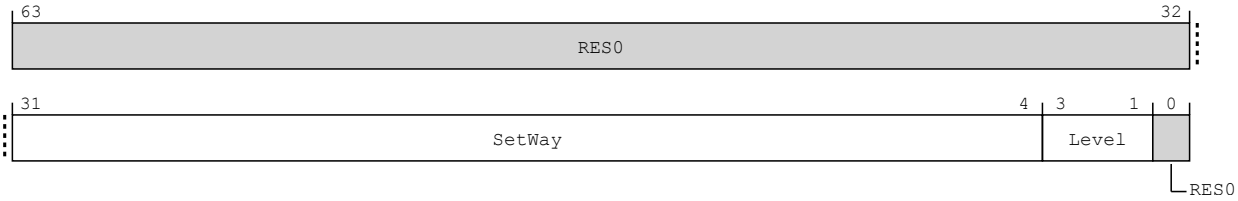

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CIGSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCISW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.17 DC CIGVAC, Clean and Invalidate of Allocation Tags by VA to PoC

The DC CIGVAC characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CIGVAC are UNDEFINED.

Attributes

DC CIGVAC is a 64-bit System instruction.

Field descriptions

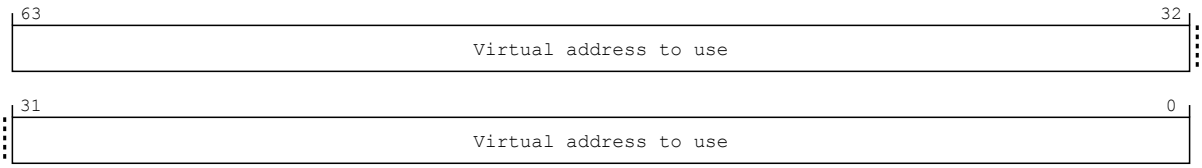

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIGVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b011

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.18 DC CIPAE, Data or unified Cache line Clean and Invalidate by PA to PoE

The DC CIPAE characteristics are:

Purpose

Data or unified Cache line Clean and Invalidate by PA to PoE.

Configuration

This system instruction is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIPAE are UNDEFINED.

Attributes

DC CIPAE is a 64-bit System instruction.

Field descriptions

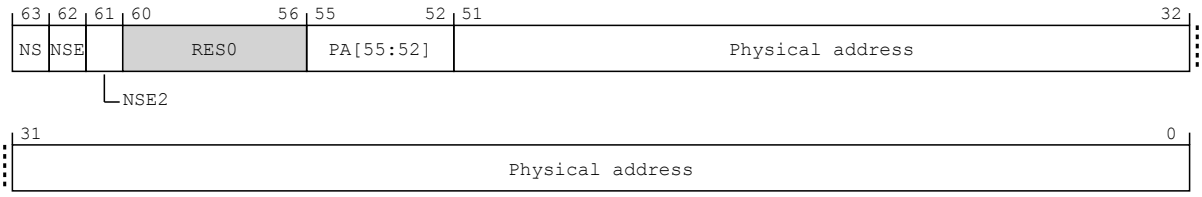

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	Reserved.
0b0	0b0	0b1	Reserved.
0b0	0b1	0b0	Reserved.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b0	0b0	Reserved.
0b0	0b1	Reserved.
0b1	0b0	Reserved.
0b1	0b1	Realm.

If {NSE, NS} is not {1, 1}, then no cache entries are required to be cleaned or invalidated.

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIPAE.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIPAE.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIPAE

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIPAE, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1110	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoE);
    end;
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoE);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.19 DC CIPAPA, Data or unified Cache line Clean and Invalidate by PA to PoPA

The DC CIPAPA characteristics are:

Purpose

Clean and Invalidate data cache by physical address to the Point of Physical Aliasing.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIPAPA are UNDEFINED.

Attributes

DC CIPAPA is a 64-bit System instruction.

Field descriptions

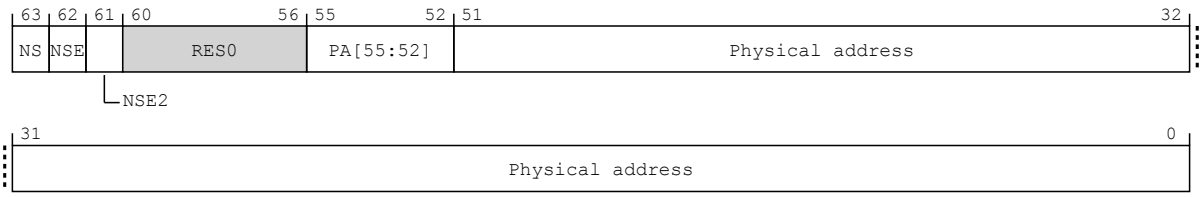

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with the NSE2 and NSE field, this field specifies the target physical address space.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b0	0b1	Non-secure.
0b0	0b1	0b0	Root.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

If {NSE2, NSE, NS} is reserved, then no cache entries are required to be cleaned or invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Together with the NSE field, this field specifies the target physical address space.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

If FEAT_SEL2 is not implemented, and {NSE, NS} == {0, 0}, then no cache entries are required to be cleaned or invalidated

NSE, bit [62]

If FEAT_RME_GDI is implemented, this field together with the NS and NSE2 fields, specifies the target physical address space.

Otherwise, this field and the NS field specify the physical address space

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see DC CIPAPA.NS.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with the NS and NSE field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIPAPA.NS.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to PA[51:0] if ID_AA64MMFR0_EL1.PARange = 0111. For more information see PA[51:0].

Otherwise:

Reserved, RES0.

PA, bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

Executing DC CIPAPA

- This instruction is not subject to any translation, permission checks, or granule protection checks.
- This instruction affects all caches in the Outer Shareable shareability domain.
- This instruction has the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
DC CIPAPA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPA);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.20 DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way

The DC CISW characteristics are:

Purpose

Clean and Invalidate data cache by set/way.

Configuration

AArch64 System instruction DC CISW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CISW are UNDEFINED.

Attributes

DC CISW is a 64-bit System instruction.

Field descriptions

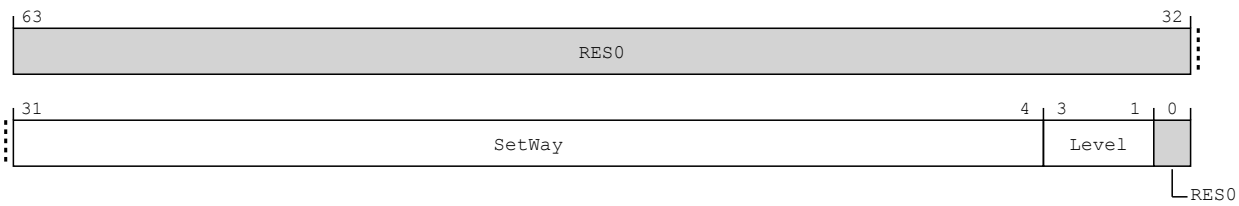

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CISW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCISW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.21 DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC

The DC CIVAC characteristics are:

Purpose

Clean and Invalidate data cache by address to Point of Coherency.

Configuration

AArch64 System instruction DC CIVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIVAC are UNDEFINED.

Attributes

DC CIVAC is a 64-bit System instruction.

Field descriptions

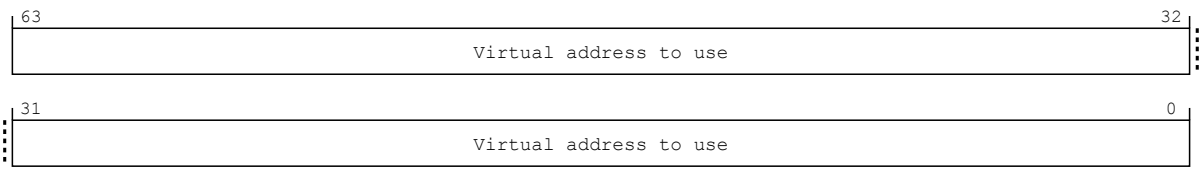

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.22 DC CIVAOC, Data or unified Cache line Clean and Invalidate by VA to Outer Cache level

The DC CIVAOC characteristics are:

Purpose

Clean and Invalidate data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIVAOC are UNDEFINED.

Attributes

DC CIVAOC is a 64-bit System instruction.

Field descriptions

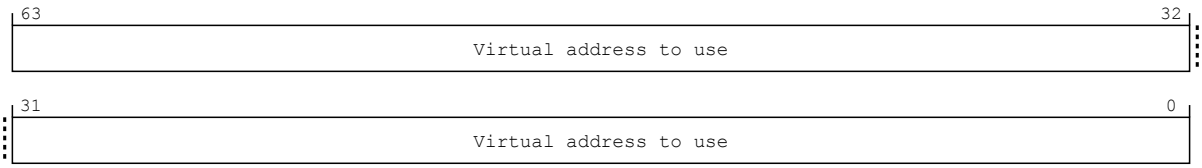

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1111	0b000

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_OuterCache);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.23 DC CIVAPS, Clean and Invalidate of Data by VA to PoPS

The DC CIVAPS characteristics are:

Purpose

Clean and Invalidate data in data cache by virtual address to the Point of Physical Storage.

Note

This instruction cleans and invalidates all copies of the Location specified in the Xt argument, irrespective of any MECID associated with the Location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.

Configuration

This system instruction is present only when FEAT_PoPS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CIVAPS are UNDEFINED.

Attributes

DC CIVAPS is a 64-bit System instruction.

Field descriptions

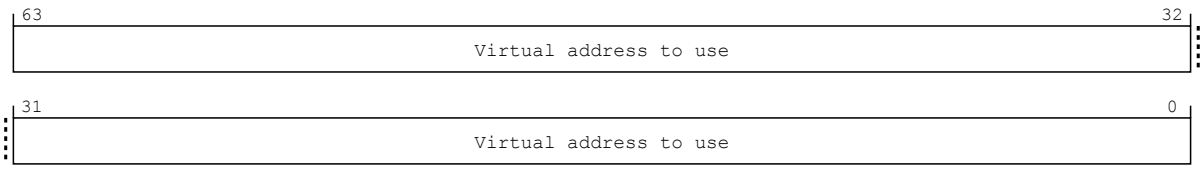

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAPS

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CIVAPS, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1111	0b001

```
if !(IsFeatureImplemented(FEAT_PoPS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGITR2_EL2().nDCCIVAPS == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoPS);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.24 DC CSW, Data or unified Cache line Clean by Set/Way

The DC CSW characteristics are:

Purpose

Clean data cache by set/way.

Configuration

AArch64 System instruction DC CSW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CSW are UNDEFINED.

Attributes

DC CSW is a 64-bit System instruction.

Field descriptions

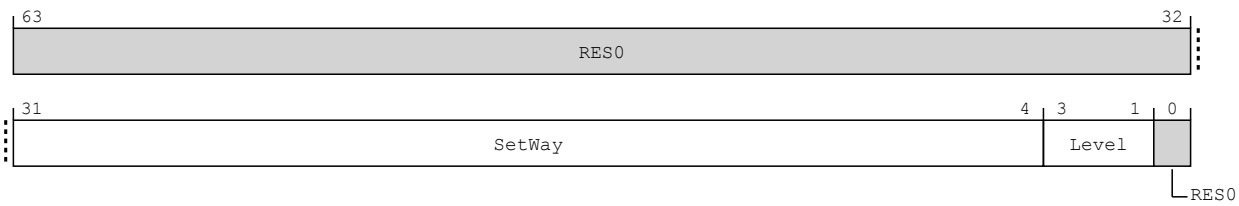

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.25 DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

Purpose

Clean data cache by address to Point of Coherency.

Configuration

AArch64 System instruction DC CVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAC are UNDEFINED.

Attributes

DC CVAC is a 64-bit System instruction.

Field descriptions

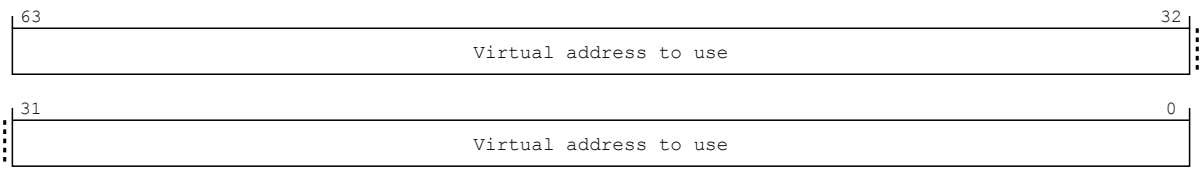

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.26 DC CVADP, Data or unified Cache line Clean by VA to PoDP

The DC CVADP characteristics are:

Purpose

Clean data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CVAP.

Configuration

This system instruction is present only when FEAT_DPB2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVADP are UNDEFINED.

Attributes

DC CVADP is a 64-bit System instruction.

Field descriptions

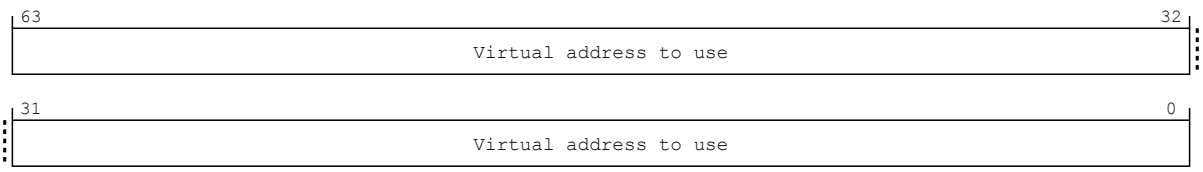

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVADP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b001

```
if !(IsFeatureImplemented(FEAT_DPB2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVADP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVADP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.27 DC CVAOC, Data or unified Cache line Clean by VA to Outer Cache level

The DC CVAOC characteristics are:

Purpose

Clean data cache by address to Outer cache level.

Configuration

This system instruction is present only when FEAT_OCCMO is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAOC are UNDEFINED.

Attributes

DC CVAOC is a 64-bit System instruction.

Field descriptions

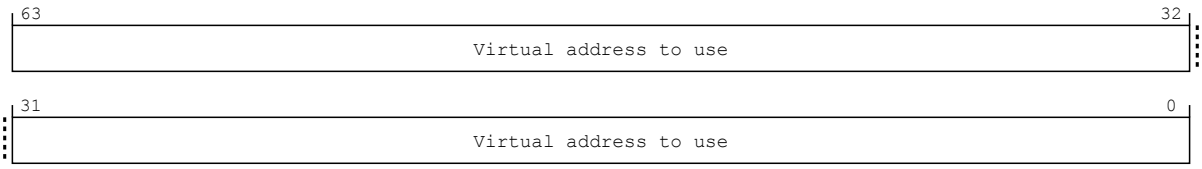

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAOC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAOC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b000

```
if !(IsFeatureImplemented(FEAT_OCCMO) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_OuterCache);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.28 DC CVAP, Data or unified Cache line Clean by VA to PoP

The DC CVAP characteristics are:

Purpose

Clean data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CVAC.

Configuration

This system instruction is present only when FEAT_DPB is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAP are UNDEFINED.

Attributes

DC CVAP is a 64-bit System instruction.

Field descriptions

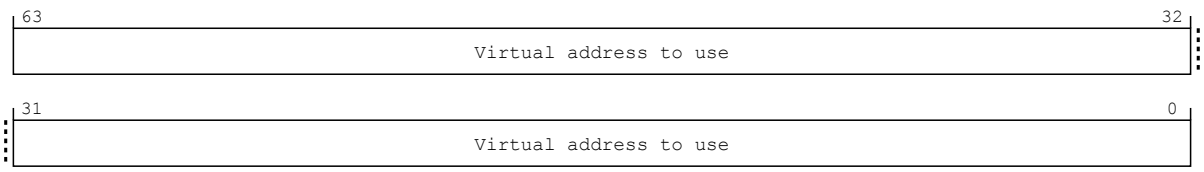

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_DPB) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGITR_EL2().DCCVAP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.29 DC CVAU, Data or unified Cache line Clean by VA to PoU

The DC CVAU characteristics are:

Purpose

Clean data cache by address to Point of Unification.

Configuration

AArch64 System instruction DC CVAU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC CVAU are UNDEFINED.

Attributes

DC CVAU is a 64-bit System instruction.

Field descriptions

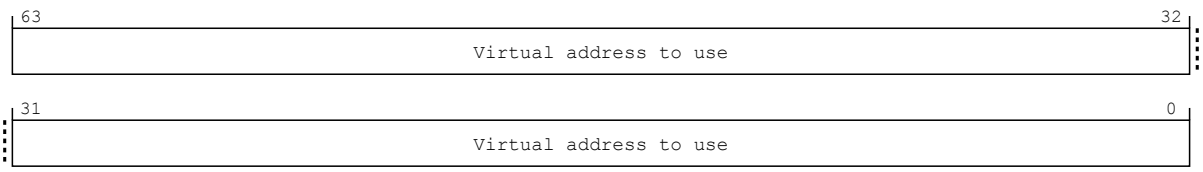

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAU

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CVAU, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TOCU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCCVAU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().TOCU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCCVAU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.30 DC GVA, Data Cache set Allocation Tag by VA

The DC GVA characteristics are:

Purpose

Write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC GVA are UNDEFINED.

Attributes

DC GVA is a 64-bit System instruction.

Field descriptions

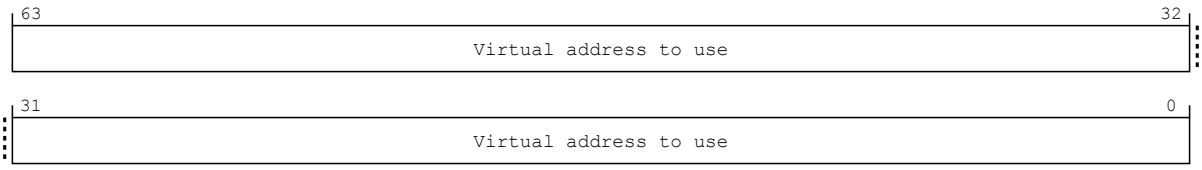

VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC GVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

For a DC GVA instruction executed to any type of Device memory, it is CONSTRAINED UNPREDICTABLE whether it generates an Alignment Fault determined by the memory type. Allocation tags at the specified addresses are not modified.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of stores to each Allocation Tag within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC GVA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().DZE == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TDZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCZVA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().DZE == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_MemZero(X{64}(t), CacheType_Tag);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TDZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCZVA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_MemZero(X{64}(t), CacheType_Tag);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_MemZero(X{64}(t), CacheType_Tag);
elsif PSTATE.EL == EL3 then
    AArch64_MemZero(X{64}(t), CacheType_Tag);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.31 DC GZVA, Data Cache set Allocation Tags and Zero by VA

The DC GZVA characteristics are:

Purpose

Zero data and write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

This instruction might have the poison-atomic property for IMPLEMENTATION DEFINED regions of physical memory.

Configuration

This system instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC GZVA are UNDEFINED.

Attributes

DC GZVA is a 64-bit System instruction.

Field descriptions

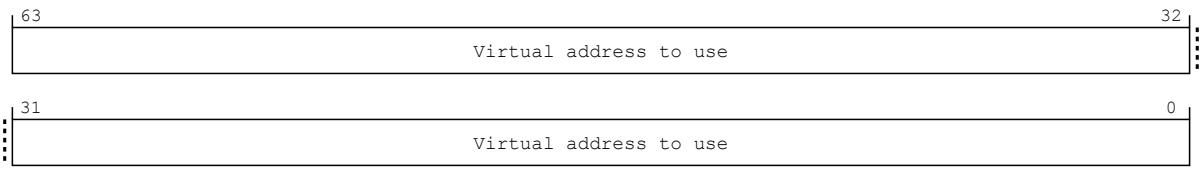

VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC GZVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being zeroed is any type of Device memory that does not support unaligned accesses, this instruction generates an alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte and Allocation tag within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC GZVA, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_MTE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().DZE == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TDZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCZVA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().DZE == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_MemZero(X{64}(t), CacheType_Data_Tag);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TDZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCZVA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_MemZero(X{64}(t), CacheType_Data_Tag);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_MemZero(X{64}(t), CacheType_Data_Tag);
elsif PSTATE.EL == EL3 then
    AArch64_MemZero(X{64}(t), CacheType_Data_Tag);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.32 DC IGDSW, Invalidate of Data and Allocation Tags by Set/Way

The DC IGDSW characteristics are:

Purpose

Invalidate data and Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGDSW are UNDEFINED.

Attributes

DC IGDSW is a 64-bit System instruction.

Field descriptions

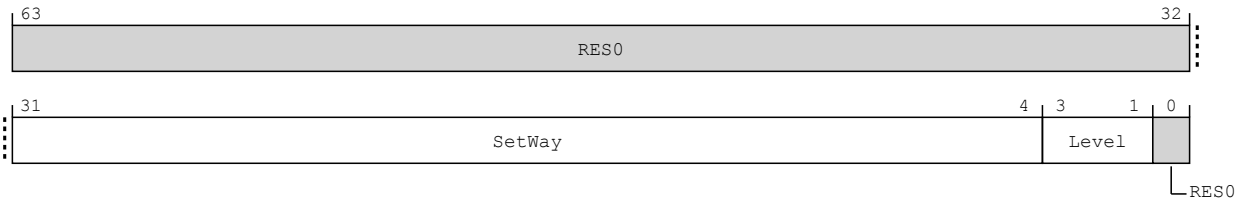

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC IGDSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- No cache lines.
- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGDSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCISW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.33 DC IGDVAC, Invalidate of Data and Allocation Tags by VA to PoC

The DC IGDVAC characteristics are:

Purpose

Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGDVAC are UNDEFINED.

Attributes

DC IGDVAC is a 64-bit System instruction.

Field descriptions

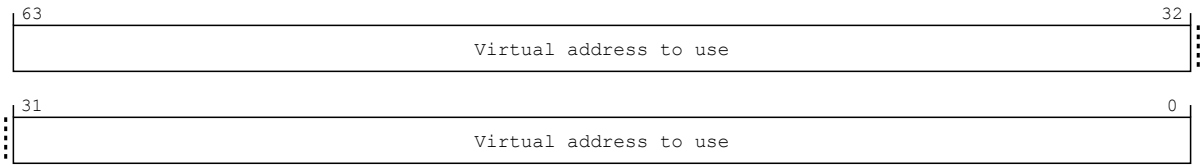

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IGDVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGDVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b101

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.34 DC IGSW, Invalidate of Allocation Tags by Set/Way

The DC IGSW characteristics are:

Purpose

Invalidate Allocation Tags in data cache by set/way.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGSW are UNDEFINED.

Attributes

DC IGSW is a 64-bit System instruction.

Field descriptions

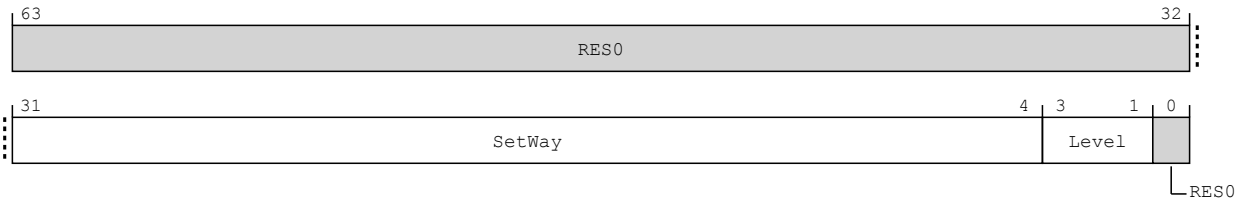

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC IGSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- No cache lines.
- A single arbitrary cache line.
- Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCISW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.35 DC IGVAC, Invalidate of Allocation Tags by VA to PoC

The DC IGVAC characteristics are:

Purpose

Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configuration

This system instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGVAC are UNDEFINED.

Attributes

DC IGVAC is a 64-bit System instruction.

Field descriptions

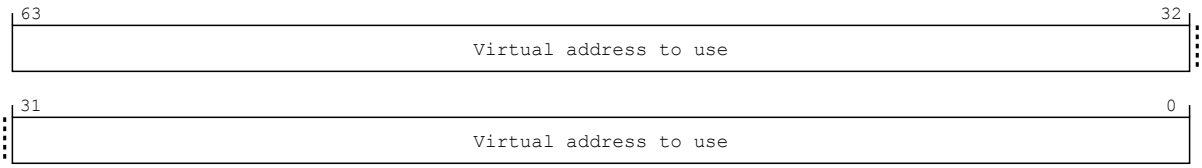

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IGVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in MMU faults generated by cache maintenance operations.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC).

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IGVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b011

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.36 DC ISW, Data or unified Cache line Invalidate by Set/Way

The DC ISW characteristics are:

Purpose

Invalidate data cache by set/way.

When FEAT_MTE2 is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

Configuration

AArch64 System instruction DC ISW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC ISW are UNDEFINED.

Attributes

DC ISW is a 64-bit System instruction.

Field descriptions

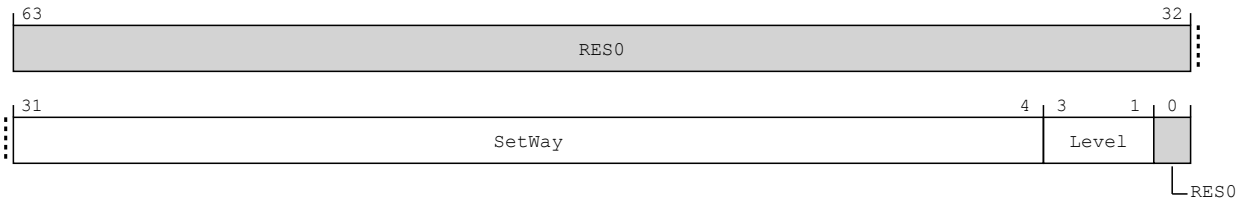

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing DC ISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC ISW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TSW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCISW == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.37 DC IVAC, Data or unified Cache line Invalidate by VA to PoC

The DC IVAC characteristics are:

Purpose

Invalidate data cache by address to Point of Coherency.

When FEAT_MTE2 is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

Configuration

AArch64 System instruction DC IVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC IVAC are UNDEFINED.

Attributes

DC IVAC is a 64-bit System instruction.

Field descriptions

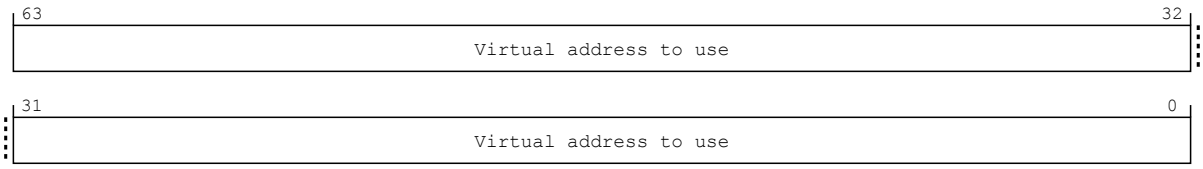

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The data cache maintenance instruction (DC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCIVAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapDC(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch64_DC(X{64}(t), CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.38 DC ZVA, Data Cache Zero by VA

The DC ZVA characteristics are:

Purpose

Zero data cache by address. Zeroes a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0.

This instruction might have the poison-atomic property for IMPLEMENTATION DEFINED regions of physical memory.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DC ZVA are UNDEFINED.

Attributes

DC ZVA is a 64-bit System instruction.

Field descriptions

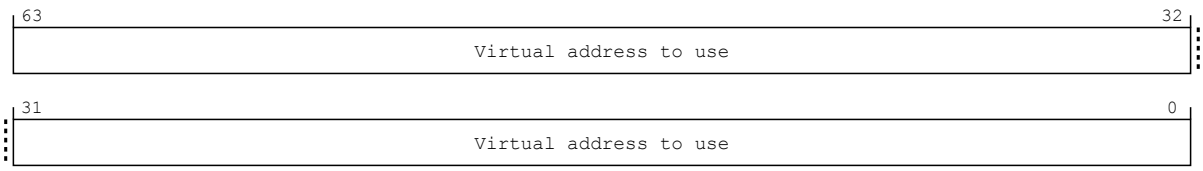

VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC ZVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 0.

If the memory region being zeroed is any type of Device memory that does not support unaligned accesses, this instruction generates an Alignment fault which is prioritized in the same way as other Alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC ZVA, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().DZE == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TDZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DCZVA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().DZE == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_MemZero(X{64}(t), CacheType_Data);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TDZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DCZVA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_MemZero(X{64}(t), CacheType_Data);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_MemZero(X{64}(t), CacheType_Data);
elsif PSTATE.EL == EL3 then
    AArch64_MemZero(X{64}(t), CacheType_Data);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.39 IC IALLU, Instruction Cache Invalidate All to PoU

The IC IALLU characteristics are:

Purpose

Invalidate all instruction caches of the PE executing the instruction to the Point of Unification.

Configuration

AArch64 System instruction IC IALLU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to IC IALLU are UNDEFINED.

Attributes

IC IALLU is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing IC IALLU

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_IC()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IALLU{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().TOCU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ICIALLU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ExecuteAsNOP();
        else
            AArch64_IC(CacheOpScope_ALLU);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_IC(CacheOpScope_ALLU);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64_IC(CacheOpScope_ALLU);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.40 IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The IC IALLUIS characteristics are:

Purpose

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification.

Configuration

AArch64 System instruction IC IALLUIS bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to IC IALLUIS are UNDEFINED.

Attributes

IC IALLUIS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing IC IALLUIS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IALLUIS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().TICAB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ICIALLUIS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_IC(CacheOpScope_ALLUIS);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_IC(CacheOpScope_ALLUIS);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64_IC(CacheOpScope_ALLUIS);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.41 IC IVAU, Instruction Cache line Invalidate by VA to PoU

The IC IVAU characteristics are:

Purpose

Invalidate instruction cache by address to Point of Unification.

Configuration

AArch64 System instruction IC IVAU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to IC IVAU are UNDEFINED.

Attributes

IC IVAU is a 64-bit System instruction.

Field descriptions

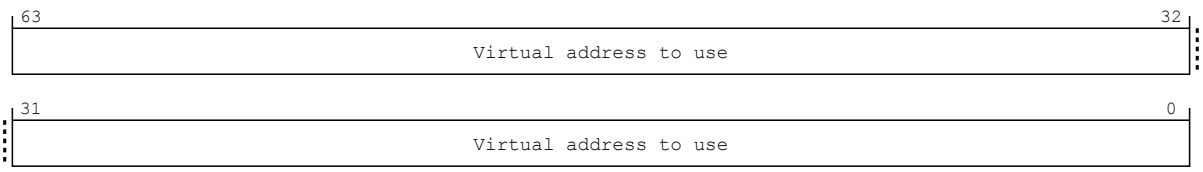

VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing IC IVAU

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in ‘MMU faults generated by cache maintenance operations’.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see ‘The instruction cache maintenance instruction (IC)’.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IVAU{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().UCI == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TPU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TOCU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().ICIVAU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCI == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_IC(X{64}(t), CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HCR_EL2().TPU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().TOCU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGITR_EL2().ICIVAU == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_IC(X{64}(t), CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch64_CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64_IC(X{64}(t), CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch64_TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch64_IC(X{64}(t), CacheOpScope_PoU);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4 A64 System instructions for address translation

This section lists the A64 System instructions for address translation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.1 AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E0R are UNDEFINED.

Attributes

AT S12E0R is a 64-bit System instruction.

Field descriptions

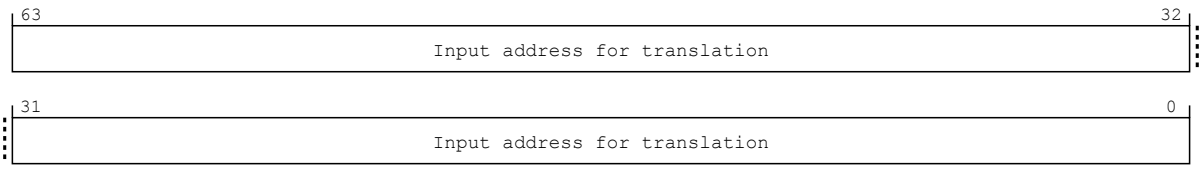

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E0R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E0R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00' then
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Read);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL0, ATAccess_Read);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
     ↪then
        Undefined();
    elsif !EL2Enabled() then
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Read);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00') then
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Read);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL0, ATAccess_Read);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E0W are UNDEFINED.

Attributes

AT S12E0W is a 64-bit System instruction.

Field descriptions

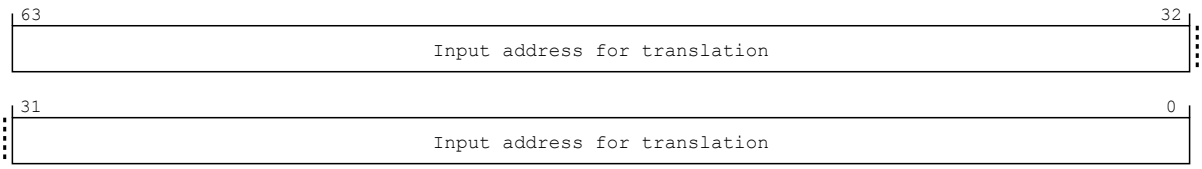

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E0W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E0W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00' then
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Write);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL0, ATAccess_Write);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
     ↪then
        Undefined();
    elsif !EL2Enabled() then
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Write);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00') then
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Write);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL0, ATAccess_Write);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E1R are UNDEFINED.

Attributes

AT S12E1R is a 64-bit System instruction.

Field descriptions

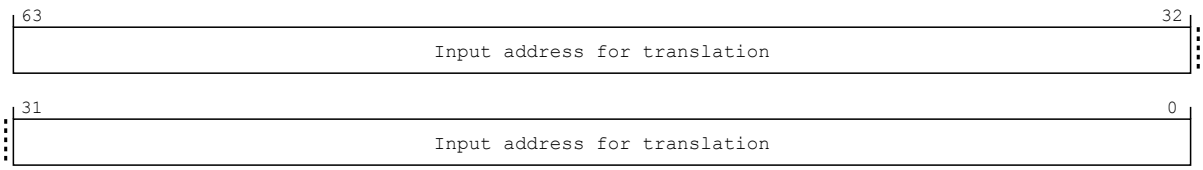

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E1R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E1R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00' then
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Read);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL1, ATAccess_Read);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
     ↪then
        Undefined();
    elsif !EL2Enabled() then
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Read);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00') then
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Read);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL1, ATAccess_Read);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S12E1W are UNDEFINED.

Attributes

AT S12E1W is a 64-bit System instruction.

Field descriptions

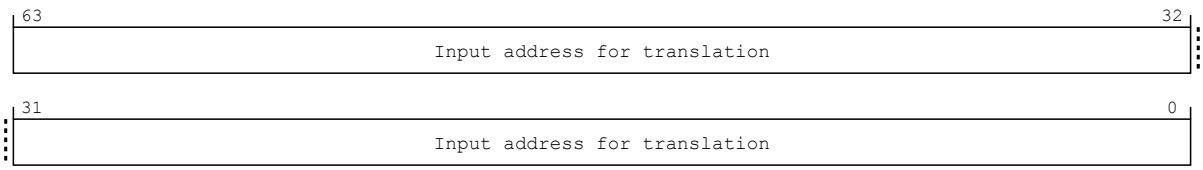

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S12E1W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S12E1W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00' then
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Write);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL1, ATAccess_Write);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
     ↪then
        Undefined();
    elsif !EL2Enabled() then
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Write);
    elsif EL2Enabled() && (ELIsInHost(EL0) || HCR_EL2().[DC,VM] == '00') then
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Write);
    else
        AArch64_AT(X{64}(t), TranslationStage_12, EL1, ATAccess_Write);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.5 AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E0R are UNDEFINED.

Attributes

AT S1E0R is a 64-bit System instruction.

Field descriptions

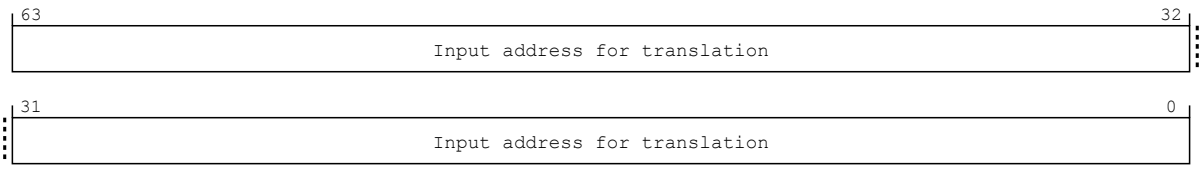

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E0R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E0R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E0R == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Read);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Read);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E0W are UNDEFINED.

Attributes

AT S1E0W is a 64-bit System instruction.

Field descriptions

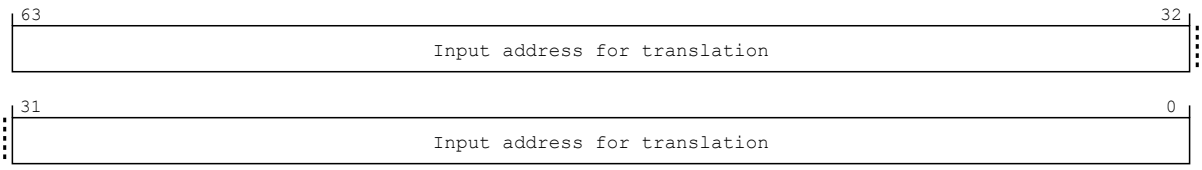

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E0W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E0W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E0W == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Write);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL0, ATAccess_Write);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.7 AT S1E1A, Address Translate Stage 1 EL1 Without Permission checks

The AT S1E1A characteristics are:

Purpose

Performs a stage 1 address translation, while ignoring the permission checks using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_ATS1A is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1A are UNDEFINED.

Attributes

AT S1E1A is a 64-bit System instruction.

Field descriptions

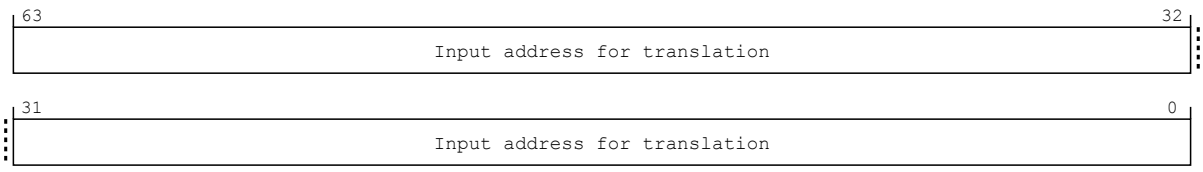

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E1A

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1A, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ATS1A) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E1A == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Any);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Any);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Any);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.8 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1R are UNDEFINED.

Attributes

AT S1E1R is a 64-bit System instruction.

Field descriptions

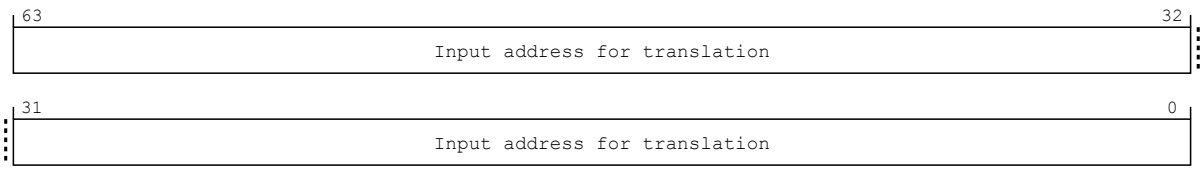

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E1R == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Read);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Read);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.9 AT S1E1RP, Address Translate Stage 1 EL1 Read PAN

The AT S1E1RP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a read from a location will generate a Permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_PAN2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1RP are UNDEFINED.

Attributes

AT S1E1RP is a 64-bit System instruction.

Field descriptions

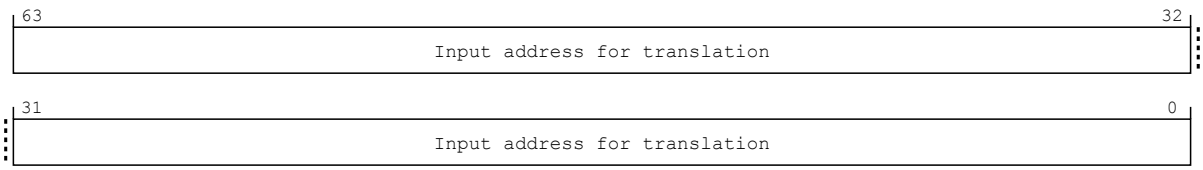

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1RP

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1RP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_PAN2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E1RP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_ReadPAN);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_ReadPAN);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.10 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1W are UNDEFINED.

Attributes

AT S1E1W is a 64-bit System instruction.

Field descriptions

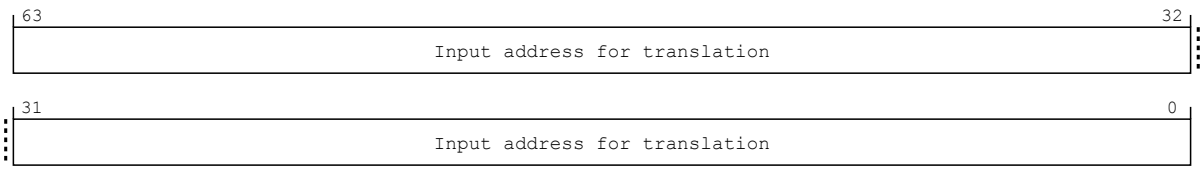

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E1W == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Write);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_Write);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.11 AT S1E1WP, Address Translate Stage 1 EL1 Write PAN

The AT S1E1WP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a write to a location will generate a Permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_PAN2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E1WP are UNDEFINED.

Attributes

AT S1E1WP is a 64-bit System instruction.

Field descriptions

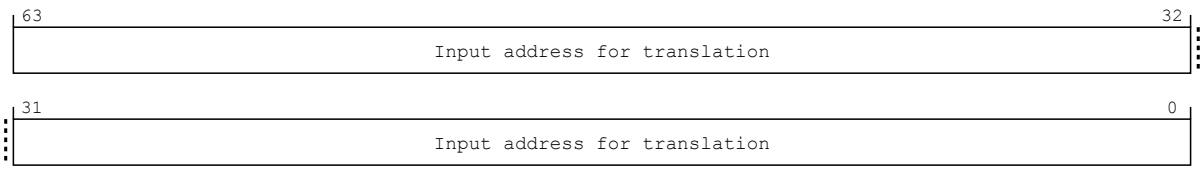

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E1WP

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E1WP, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_PAN2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().AT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().ATS1E1WP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_WritePAN);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
         ↪then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL1, ATAccess_WritePAN);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.12 AT S1E2A, Address Translate Stage 1 EL2 Without Permission checks

The AT S1E2A characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, while ignoring permissions checks from the given virtual address.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_ATS1A is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E2A are UNDEFINED.

Attributes

AT S1E2A is a 64-bit System instruction.

Field descriptions

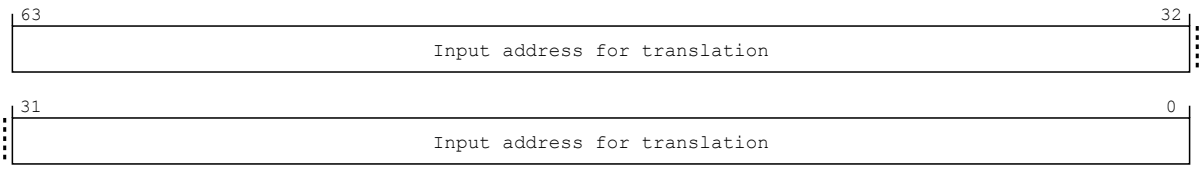

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E2A

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E2A, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1001	0b010

```
if !(IsFeatureImplemented(FEAT_ATS1A) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL2, ATAccess_Any);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
        ↪then
        Undefined();
    elsif !EL2Enabled() then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL2, ATAccess_Any);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.13 AT S1E2R, Address Translate Stage 1 EL2 Read

The AT S1E2R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if reading from the given virtual address.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E2R are UNDEFINED.

Attributes

AT S1E2R is a 64-bit System instruction.

Field descriptions

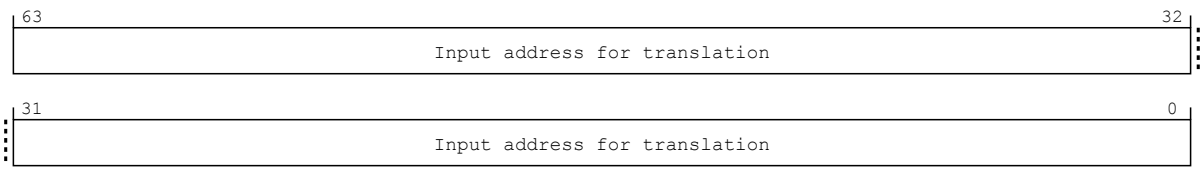

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E2R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E2R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL2, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
        ↪then
        Undefined();
    elsif !EL2Enabled() then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL2, ATAccess_Read);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.14 AT S1E2W, Address Translate Stage 1 EL2 Write

The AT S1E2W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if writing to the given virtual address.

When FEAT_RME is implemented, if the Effective value of SCR_EL3.{NSE, NS} is a reserved value, this instruction is UNDEFINED at EL3.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E2W are UNDEFINED.

Attributes

AT S1E2W is a 64-bit System instruction.

Field descriptions

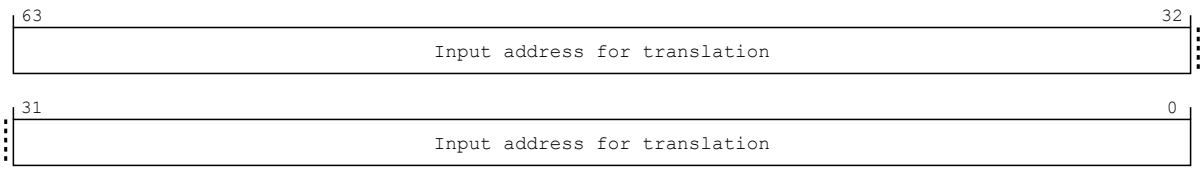

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E2W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E2W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL2, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && EffectiveSCR_EL3_NS() == '0' && EffectiveSCR_EL3_NSE() == '1'
        ↪then
        Undefined();
    elsif !EL2Enabled() then
        Undefined();
    else
        AArch64_AT(X{64}(t), TranslationStage_1, EL2, ATAccess_Write);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.15 AT S1E3A, Address Translate Stage 1 EL3 Without Permission checks

The AT S1E3A characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, while ignoring permissions checks from the given virtual address.

Configuration

This system instruction is present only when FEAT_ATS1A is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E3A are UNDEFINED.

Attributes

AT S1E3A is a 64-bit System instruction.

Field descriptions

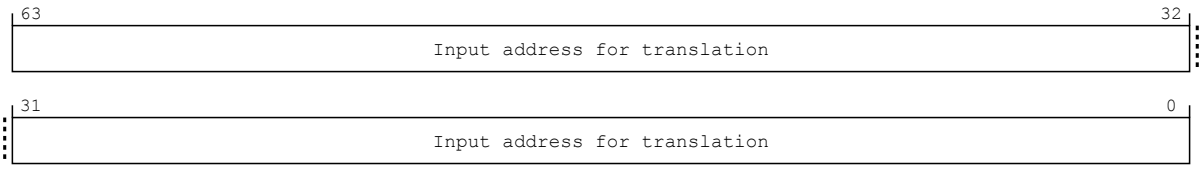

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E3A

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E3A, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1001	0b010

```
if !(IsFeatureImplemented(FEAT_ATS1A) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL3, ATAccess_Any);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.16 AT S1E3R, Address Translate Stage 1 EL3 Read

The AT S1E3R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E3R are UNDEFINED.

Attributes

AT S1E3R is a 64-bit System instruction.

Field descriptions

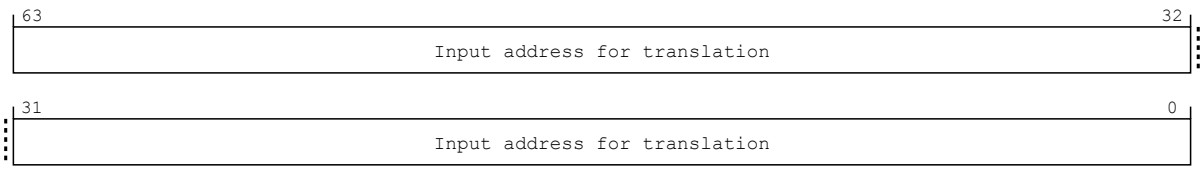

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E3R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E3R, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL3, ATAccess_Read);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.17 AT S1E3W, Address Translate Stage 1 EL3 Write

The AT S1E3W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AT S1E3W are UNDEFINED.

Attributes

AT S1E3W is a 64-bit System instruction.

Field descriptions

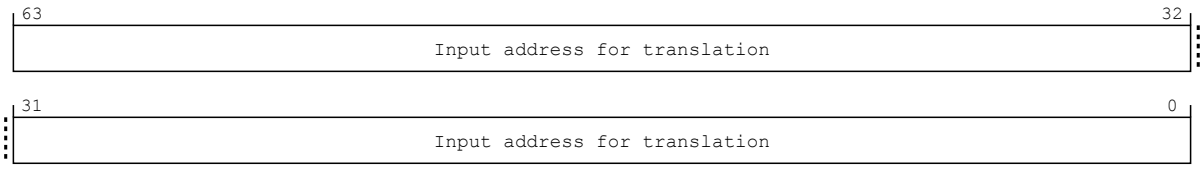

IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E3W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
AT S1E3W, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_AT(X{64}(t), TranslationStage_1, EL3, ATAccess_Write);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5 A64 System instructions for TLB maintenance

This section lists the A64 System instructions for TLB maintenance.

For more information about these instructions see TLB maintenance instructions. In particular, for the full description of the scope of each instruction see TLB maintenance instruction scope.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.1 TLBI ALLE1, TLBI ALLE1NXS, TLB Invalidate All, EL1

The TLBI ALLE1, TLBI ALLE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation only applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE1, TLBI ALLE1NXS are UNDEFINED.

Attributes

TLBI ALLE1, TLBI ALLE1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1, TLBI ALLE1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI ALLE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_ExcludeXS,
            ↪X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.2 TLBI ALLE1IS, TLBI ALLE1ISNXS, TLB Invalidate All, EL1, Inner Shareable

The TLBI ALLE1IS, TLBI ALLE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE1IS, TLBI ALLE1ISNXS are UNDEFINED.

Attributes

TLBI ALLE1IS, TLBI ALLE1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1IS, TLBI ALLE1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI ALLE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_ExcludeXS,
            ↪X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.3 TLBI ALLE1OS, TLBI ALLE1OSNXS, TLB Invalidate All, EL1, Outer Shareable

The TLBI ALLE1OS, TLBI ALLE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE1OS, TLBI ALLE1OSNXS are UNDEFINED.

Attributes

TLBI ALLE1OS, TLBI ALLE1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1OS, TLBI ALLE1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI ALLE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_ExcludeXS,
            ↪X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.4 TLBI ALLE2, TLBI ALLE2NXS, TLB Invalidate All, EL2

The TLBI ALLE2, TLBI ALLE2NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE2, TLBI ALLE2NXS are UNDEFINED.

Attributes

TLBI ALLE2, TLBI ALLE2NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2, TLBI ALLE2NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI ALLE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_ExcludeXS,
                ↪X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.5 TLBI ALLE2IS, TLBI ALLE2ISNXS, TLB Invalidate All, EL2, Inner Shareable

The TLBI ALLE2IS, TLBI ALLE2ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE2IS, TLBI ALLE2ISNXS are UNDEFINED.

Attributes

TLBI ALLE2IS, TLBI ALLE2ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2IS, TLBI ALLE2ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI ALLE2ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_ExcludeXS,
                ↪X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.6 TLBI ALLE2OS, TLBI ALLE2OSNXS, TLB Invalidate All, EL2, Outer Shareable

The TLBI ALLE2OS, TLBI ALLE2OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE2OS, TLBI ALLE2OSNXS are UNDEFINED.

Attributes

TLBI ALLE2OS, TLBI ALLE2OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2OS, TLBI ALLE2OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI ALLE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI ALLE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_ExcludeXS,
                ↪X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.7 TLBI ALLE3, TLBI ALLE3NXS, TLB Invalidate All, EL3

The TLBI ALLE3, TLBI ALLE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE3, TLBI ALLE3NXS are UNDEFINED.

Attributes

TLBI ALLE3, TLBI ALLE3NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3, TLBI ALLE3NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_NSH, TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI ALLE3NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_NSH, TLBI_ExcludeXS,
            ↪X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.8 TLBI ALLE3IS, TLBI ALLE3ISNXS, TLB Invalidate All, EL3, Inner Shareable

The TLBI ALLE3IS, TLBI ALLE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE3IS, TLBI ALLE3ISNXS are UNDEFINED.

Attributes

TLBI ALLE3IS, TLBI ALLE3ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3IS, TLBI ALLE3ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_ISH, TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI ALLE3ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_ISH, TLBI_ExcludeXS,
            ↪X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.9 TLBI ALLE3OS, TLBI ALLE3OSNXS, TLB Invalidate All, EL3, Outer Shareable

The TLBI ALLE3OS, TLBI ALLE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ALLE3OS, TLBI ALLE3OSNXS are UNDEFINED.

Attributes

TLBI ALLE3OS, TLBI ALLE3OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3OS, TLBI ALLE3OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_OSH, TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI ALLE3OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Broadcast_OSH, TLBI_ExcludeXS,
            ↪X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.10 TLBI ASIDE1, TLBI ASIDE1NXS, TLB Invalidate by ASID, EL1

The TLBI ASIDE1, TLBI ASIDE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ASIDE1, TLBI ASIDE1NXS are UNDEFINED.

Attributes

TLBI ASIDE1, TLBI ASIDE1NXS is a 64-bit System instruction.

Field descriptions

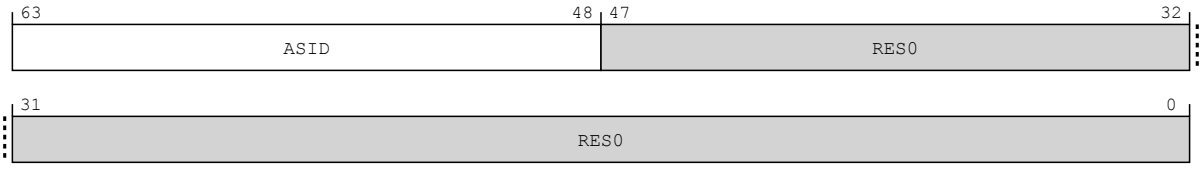

ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1, TLBI ASIDE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ASID()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIASIDE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI ASIDE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIASIDE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_ExcludeXS,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.11 TLBI ASIDE1IS, TLBI ASIDE1ISNXS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS, TLBI ASIDE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ASIDE1IS, TLBI ASIDE1ISNXS are UNDEFINED.

Attributes

TLBI ASIDE1IS, TLBI ASIDE1ISNXS is a 64-bit System instruction.

Field descriptions

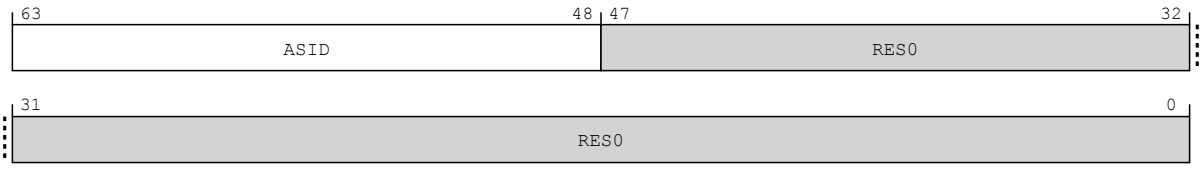

ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1IS, TLBI ASIDE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ASID()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIASIDE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI ASIDE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIASIDE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_ExcludeXS,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.12 TLBI ASIDE1OS, TLBI ASIDE1OSNXS, TLB Invalidate by ASID, EL1, Outer Shareable

The TLBI ASIDE1OS, TLBI ASIDE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI ASIDE1OS, TLBI ASIDE1OSNXS are UNDEFINED.

Attributes

TLBI ASIDE1OS, TLBI ASIDE1OSNXS is a 64-bit System instruction.

Field descriptions

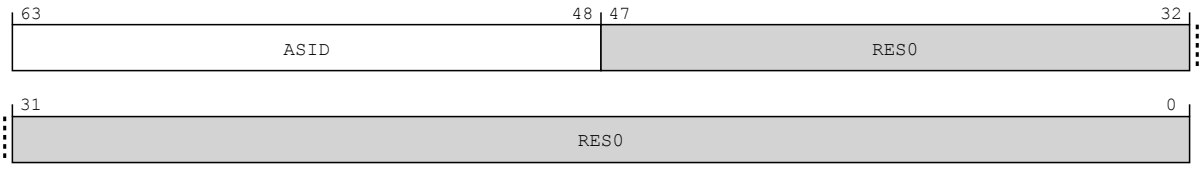

ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1OS, TLBI ASIDE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_ASID()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIASIDE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI ASIDE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIASIDE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_ExcludeXS,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.13 TLBI IPAS2E1, TLBI IPAS2E1NXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI IPAS2E1, TLBI IPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2E1, TLBI IPAS2E1NXS are UNDEFINED.

Attributes

TLBI IPAS2E1, TLBI IPAS2E1NXS is a 64-bit System instruction.

Field descriptions

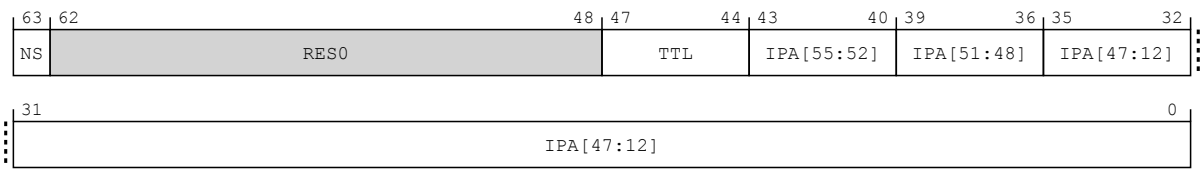

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2E1, TLBI IPAS2E1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI IPAS2E1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.14 TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS are UNDEFINED.

Attributes

TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

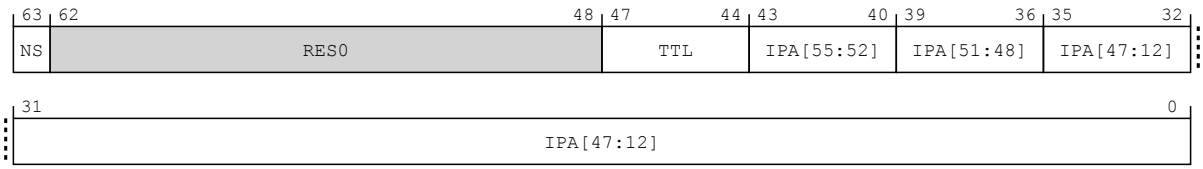

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI IPAS2E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.15 TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS are UNDEFINED.

Attributes

TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

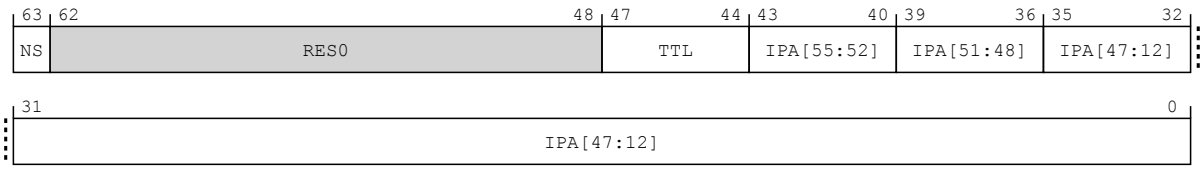

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI IPAS2E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.16 TLBI IPAS2LE1, TLBI IPAS2LE1NXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI IPAS2LE1, TLBI IPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2LE1, TLBI IPAS2LE1NXS are UNDEFINED.

Attributes

TLBI IPAS2LE1, TLBI IPAS2LE1NXS is a 64-bit System instruction.

Field descriptions

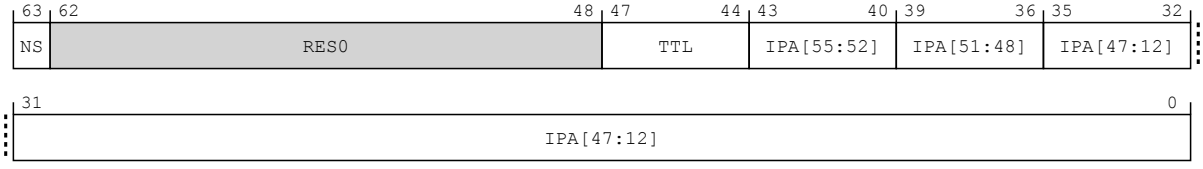

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2LE1, TLBI IPAS2LE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2LE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI IPAS2LE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.17 TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS are UNDEFINED.

Attributes

TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

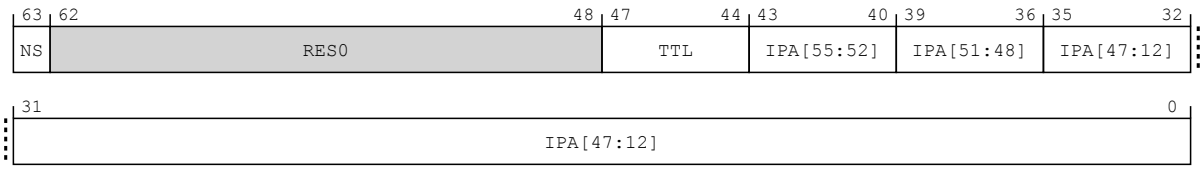

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2LE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI IPAS2LE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.18 TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

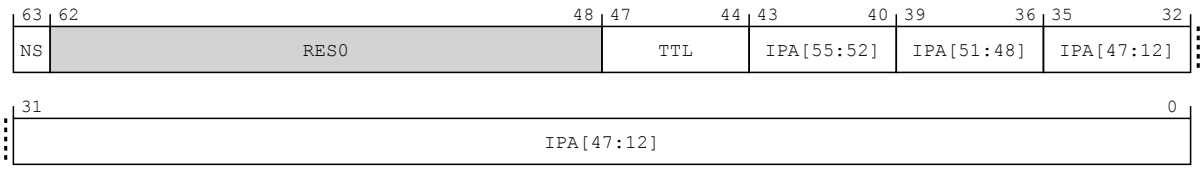

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

IPA[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is `0b0111`, bits IPA[55:48] form the upper part of the address value.

If ID_AA64MMFR0_EL1.PARange is `0b0110`, bits IPA[51:48] form the upper part of the address value and bits IPA[55:52] are RES0.

If ID_AA64MMFR0_EL1.PARange is not `0b0110` and not `0b0111`, bits IPA[55:48] are RES0.

Executing TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI IPAS2LE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI IPAS2LE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.19 TLBI PAALL, TLB Invalidate GPT Information by PA, All Entries, Local

The TLBI PAALL characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation applies to all TLB entries containing GPT information.
- The invalidation affects only the TLBs for the PE executing the operation.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI PAALL are UNDEFINED.

Attributes

TLBI PAALL is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI PAALL

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI PAALL{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b100

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_TLBI_PAALL(Broadcast_NSH);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.20 TLBI PAALLOS, TLB Invalidate GPT Information by PA, All Entries, Outer Shareable

The TLBI PAALLOS characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation applies to all TLB entries containing GPT information.
- The invalidation affects all TLBs in the Outer Shareable domain.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI PAALLOS are UNDEFINED.

Attributes

TLBI PAALLOS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI PAALLOS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI PAALLOS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_TLBI_PAALL(Broadcast_OSH);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.21 TLBI RIPAS2E1, TLBI RIPAS2E1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI RIPAS2E1, TLBI RIPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2E1, TLBI RIPAS2E1NXS are UNDEFINED.

Attributes

TLBI RIPAS2E1, TLBI RIPAS2E1NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

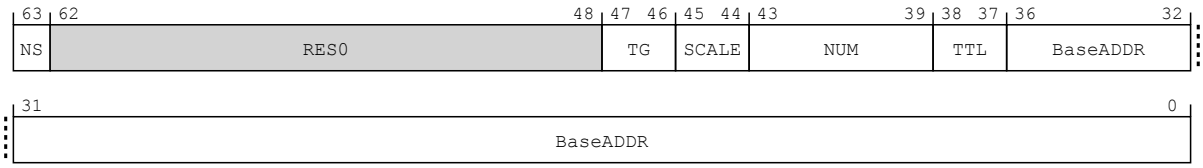

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2E1, TLBI RIPAS2E1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI RIPAS2E1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.22 TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS is a 64-bit System instruction.

Field descriptions

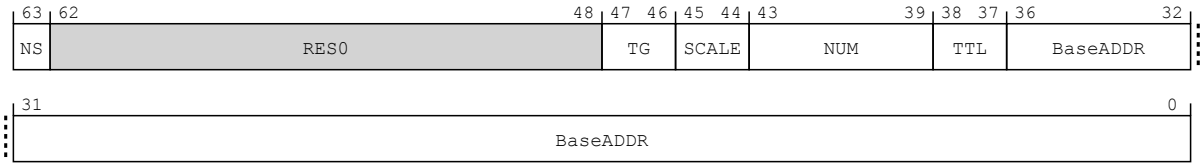

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI RIPAS2E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.23 TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS is a 64-bit System instruction.

Field descriptions

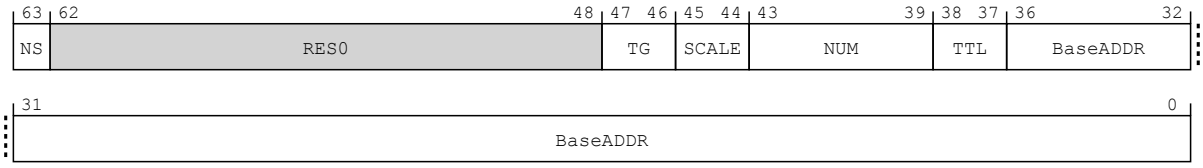

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI RIPAS2E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.24 TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS is a 64-bit System instruction.

Field descriptions

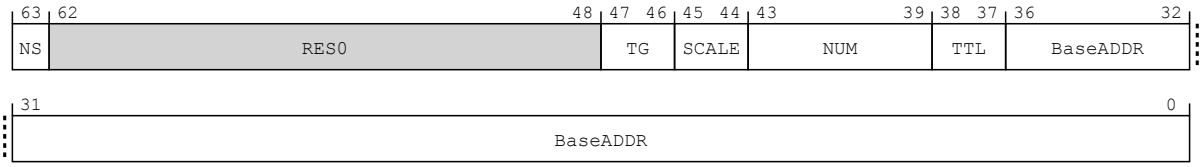

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2LE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI RIPAS2LE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.25 TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS is a 64-bit System instruction.

Field descriptions

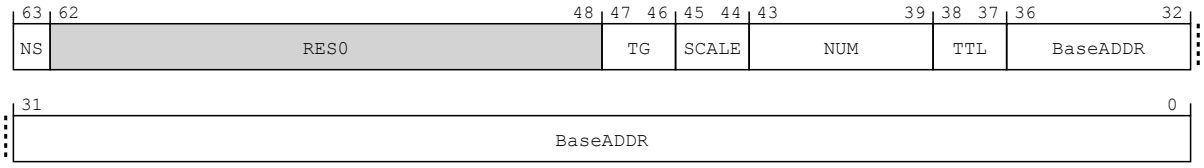

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2LE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI RIPAS2LE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.26 TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS is a 64-bit System instruction.

Field descriptions

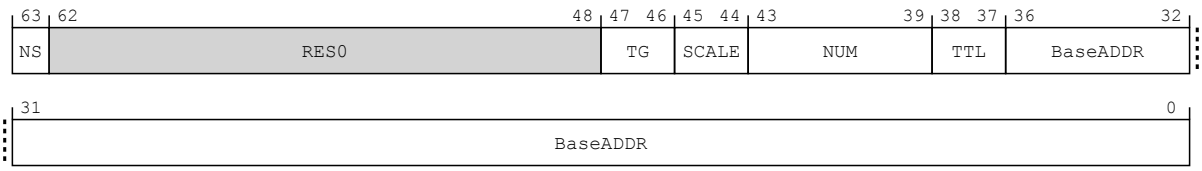

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (VTCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (VTCR_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as `0b0000`.

When using a 16KB translation granule, BaseADDR[15:14] is treated as `0b00`.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2LE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI RIPAS2LE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.27 TLBI RPALOS, TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable

The TLBI RPALOS characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation affects all TLBs in the Outer Shareable domain.
- Invalidates TLB entries containing GPT information from the final level of the GPT walk that relates to the supplied physical address.
- Invalidations are range-based, invalidating TLB entries starting from the address in BaseADDR, within the range as specified by SIZE.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RPALOS are UNDEFINED.

Attributes

TLBI RPALOS is a 64-bit System instruction.

Field descriptions

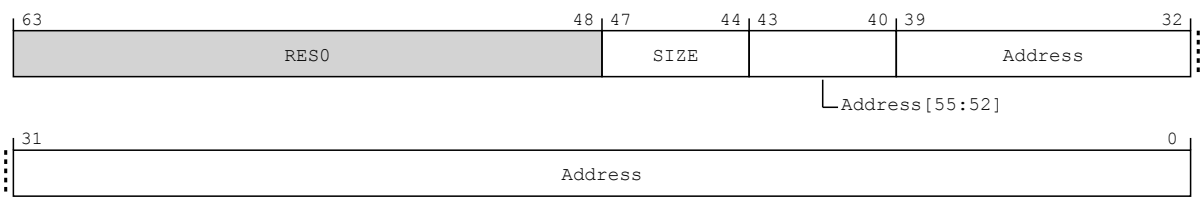

Bits [63:48]

Reserved, RES0.

SIZE, bits [47:44]

Size of the range for invalidation.

If SIZE is a reserved value, no TLB entries are required to be invalidated.

SIZE	Meaning
0b0000	4KB.
0b0001	16KB.
0b0010	64KB.
0b0011	2MB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIZE	Meaning
0b0100	32MB.
0b0101	512MB.
0b0110	1GB.
0b0111	16GB.
0b1000	64GB.
0b1001	512GB.

All other values are reserved.

If SIZE gives a range smaller than the configured physical granule size in GPCCR_EL3.PGS, then the Effective value of SIZE is taken to be the size configured by GPCCR_EL3.PGS.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

If GPCCR_EL3.PGS is configured to different values at the broadcasting PE and receiving PE, no TLB entries are required to be invalidated at the receiving PE.

Address[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to Address. For more information, see Address.

Otherwise:

Reserved, RES0.

Address, bits [39:0]

The starting address for the range of the maintenance instruction.

This field is decoded with reference to the value of GPCCR_EL3.PGS to give BaseADDR as follows:

GPCCR_EL3.PGS	BaseADDR
0b00 (4KB)	BaseADDR[51:12] = Xt[39:0]
0b10 (16KB)	BaseADDR[51:14] = Xt[39:2]
0b01 (64KB)	BaseADDR[51:16] = Xt[39:4]

Other bits of BaseADDR are treated as zero, to give the Effective value of BaseADDR.

If the Effective value of BaseADDR is not aligned to the size of the Effective value of SIZE, no TLB entries are required to be invalidated.

If the Effective value of BaseADDR targets an address above the implemented PA range that ID_AA64MMFR0_EL1.PARange indicates, no TLB entries are required to be invalidated.

If ID_AA64MMFR0_EL1.PARange is 0b0111, Address[55:52] form the upper part of the BaseADDR value. Otherwise, Address[55:52] are RES0.

Executing TLBI RPALOS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI RPALOS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_TLBI_RPA(TLBILevel_Last, X{64}(t), Broadcast_OSH);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.28 TLBI RPAOS, TLB Range Invalidate GPT Information by PA, Outer Shareable

The TLBI RPAOS characteristics are:

Purpose

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation affects all TLBs in the Outer Shareable domain.
- Invalidates TLB entries containing GPT information from all levels of the GPT walk that relates to the supplied physical address.
- Invalidations are range-based, invalidating TLB entries starting from the address in BaseADDR, within the range as specified by SIZE.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

Configuration

This system instruction is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RPAOS are UNDEFINED.

Attributes

TLBI RPAOS is a 64-bit System instruction.

Field descriptions

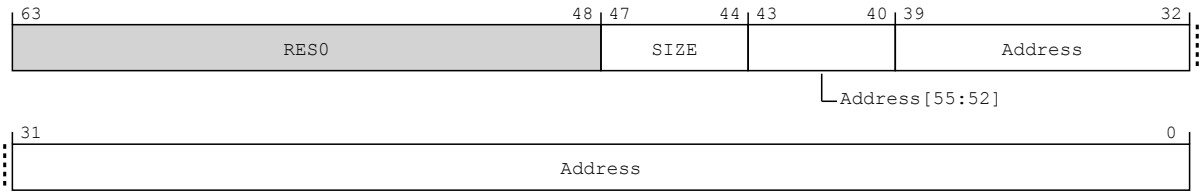

Bits [63:48]

Reserved, RES0.

SIZE, bits [47:44]

Size of the range for invalidation.

If SIZE is a reserved value, no TLB entries are required to be invalidated.

SIZE	Meaning
0b0000	4KB.
0b0001	16KB.
0b0010	64KB.
0b0011	2MB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIZE	Meaning
0b0100	32MB.
0b0101	512MB.
0b0110	1GB.
0b0111	16GB.
0b1000	64GB.
0b1001	512GB.

All other values are reserved.

If SIZE gives a range smaller than the configured physical granule size in GPCCR_EL3.PGS, then the Effective value of SIZE is taken to be the size configured by GPCCR_EL3.PGS.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

If GPCCR_EL3.PGS is configured to different values at the broadcasting PE and receiving PE, no TLB entries are required to be invalidated at the receiving PE.

Address[55:52], bits [43:40]

When FEAT_D128 is implemented:

Extension to Address. For more information, see Address.

Otherwise:

Reserved, RES0.

Address, bits [39:0]

The starting address for the range of the maintenance instruction.

This field is decoded with reference to the value of GPCCR_EL3.PGS to give BaseADDR as follows:

GPCCR_EL3.PGS	BaseADDR
0b00 (4KB)	BaseADDR[51:12] = Xt[39:0]
0b10 (16KB)	BaseADDR[51:14] = Xt[39:2]
0b01 (64KB)	BaseADDR[51:16] = Xt[39:4]

Other bits of BaseADDR are treated as zero, to give the Effective value of BaseADDR.

If the Effective value of BaseADDR is not aligned to the size of the Effective value of SIZE, no TLB entries are required to be invalidated.

If the Effective value of BaseADDR targets an address above the implemented PA range that ID_AA64MMFR0_EL1.PARange indicates, no TLB entries are required to be invalidated.

If ID_AA64MMFR0_EL1.PARange is 0b0111, Address[55:52] form the upper part of the BaseADDR value. Otherwise, Address[55:52] are RES0.

Executing TLBI RPAOS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI RPAOS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_TLBI_RPA(TLBILevel_Any, X{64}(t), Broadcast_OSH);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.29 TLBI RVAAE1, TLBI RVAAE1NXS, TLB Range Invalidate by VA, All ASID, EL1

The TLBI RVAAE1, TLBI RVAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAAE1, TLBI RVAAE1NXS are UNDEFINED.

Attributes

TLBI RVAAE1, TLBI RVAAE1NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

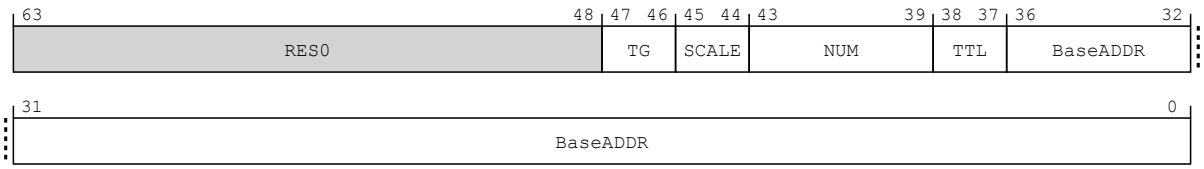

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1, TLBI RVAAE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVAA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAAE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.30 TLBI RVAAE1IS, TLBI RVAAE1ISNXS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI RVAAE1IS, TLBI RVAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAAE1IS, TLBI RVAAE1ISNXS are UNDEFINED.

Attributes

TLBI RVAAE1IS, TLBI RVAAE1ISNXS is a 64-bit System instruction.

Field descriptions

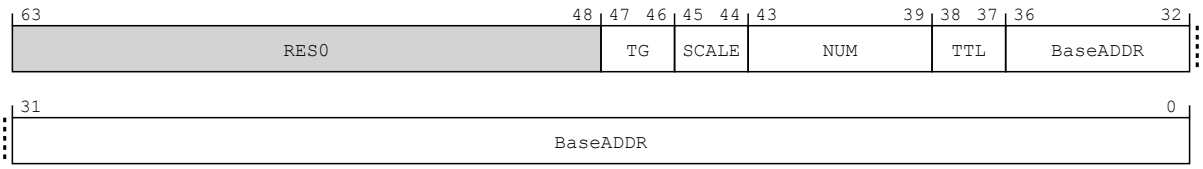

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1IS, TLBI RVAAE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVAA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGITR_EL2().TLBIRVAAE1IS == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAAE1ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.31 TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI RVAAE1OS, TLBI RVAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAAE1OS, TLBI RVAAE1OSNXS are UNDEFINED.

Attributes

TLBI RVAAE1OS, TLBI RVAAE1OSNXS is a 64-bit System instruction.

Field descriptions

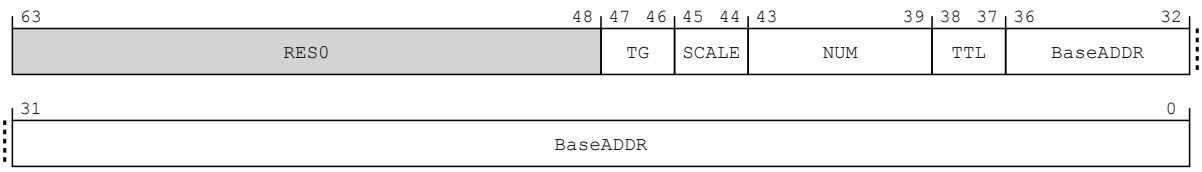

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1OS, TLBI RVAAE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVAA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGITR_EL2().TLBIRVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAAE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.32 TLBI RVAALE1, TLBI RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBI RVAALE1, TLBI RVAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAALE1, TLBI RVAALE1NXS are UNDEFINED.

Attributes

TLBI RVAALE1, TLBI RVAALE1NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

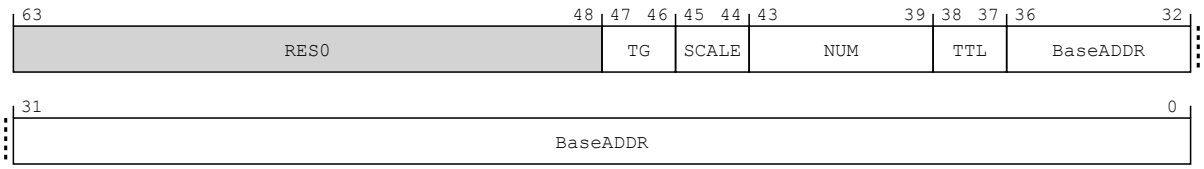

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1, TLBI RVAALE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVAA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAALE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.33 TLBI RVAALE1IS, TLBI RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI RVAALE1IS, TLBI RVAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAALE1IS, TLBI RVAALE1ISNXS are UNDEFINED.

Attributes

TLBI RVAALE1IS, TLBI RVAALE1ISNXS is a 64-bit System instruction.

Field descriptions

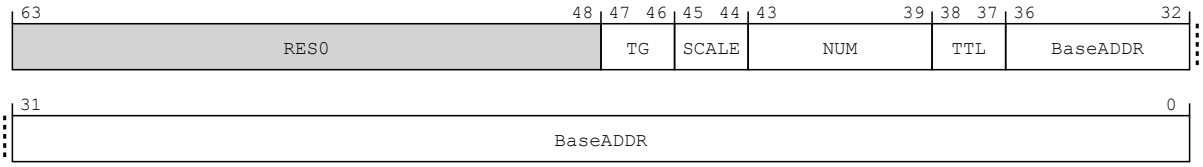

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1IS, TLBI RVAALE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVAA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGITR_EL2().TLBIRVAALE1IS == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAALE1ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.34 TLBI RVAALE1OS, TLBI RVAALE1OSNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI RVAALE1OS, TLBI RVAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAALE1OS, TLBI RVAALE1OSNXS are UNDEFINED.

Attributes

TLBI RVAALE1OS, TLBI RVAALE1OSNXS is a 64-bit System instruction.

Field descriptions

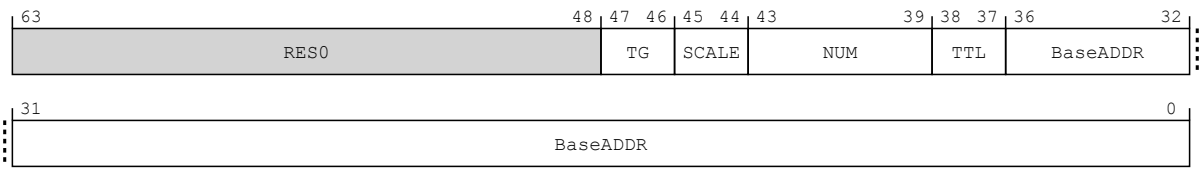

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1OS, TLBI RVAALE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVAA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGITR_EL2().TLBIRVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAALE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.35 TLBI RVAE1, TLBI RVAE1NXS, TLB Range Invalidate by VA, EL1

The TLBI RVAE1, TLBI RVAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE1, TLBI RVAE1NXS are UNDEFINED.

Attributes

TLBI RVAE1, TLBI RVAE1NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

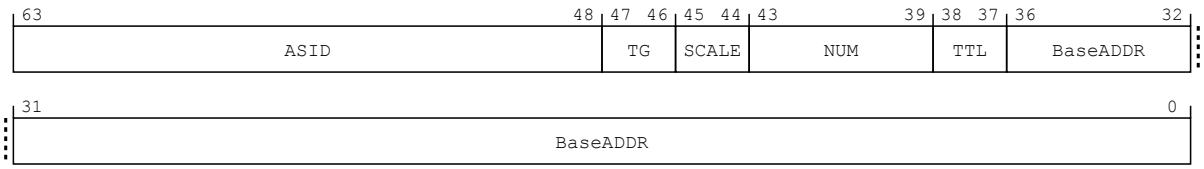

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
`0b00`	Reserved.
`0b01`	4K translation granule.
`0b10`	16K translation granule.
`0b11`	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1, TLBI RVAE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.36 TLBI RVAE1IS, TLBI RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBI RVAE1IS, TLBI RVAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE1IS, TLBI RVAE1ISNXS are UNDEFINED.

Attributes

TLBI RVAE1IS, TLBI RVAE1ISNXS is a 64-bit System instruction.

Field descriptions

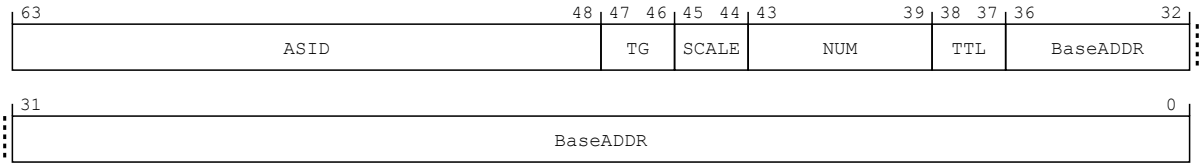

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
`0b00`	The entries in the range can be using any level for the translation table entries.
`0b01`	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.
`0b10`	The TTL hint indicates level 2.
`0b11`	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as `0b0000`.

When using a 16KB translation granule, BaseADDR[15:14] is treated as `0b00`.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1IS, TLBI RVAE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0010`	`0b001`

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVAE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIRVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.37 TLBI RVAE1OS, TLBI RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBI RVAE1OS, TLBI RVAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE1OS, TLBI RVAE1OSNXS are UNDEFINED.

Attributes

TLBI RVAE1OS, TLBI RVAE1OSNXS is a 64-bit System instruction.

Field descriptions

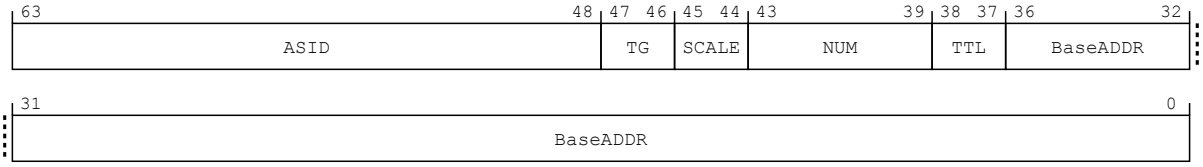

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1OS, TLBI RVAE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVAE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIRVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.38 TLBI RVAE2, TLBI RVAE2NXS, TLB Range Invalidate by VA, EL2

The TLBI RVAE2, TLBI RVAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE2, TLBI RVAE2NXS are UNDEFINED.

Attributes

TLBI RVAE2, TLBI RVAE2NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

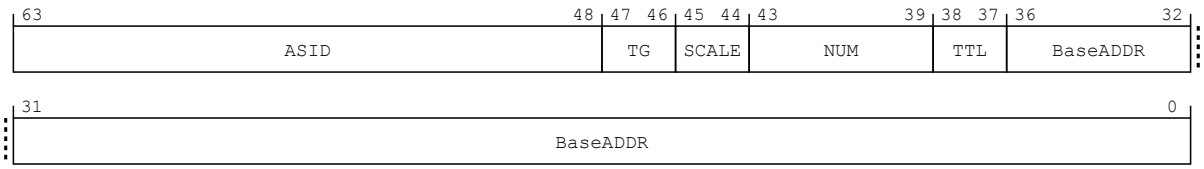

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2, TLBI RVAE2NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

When FEAT_XS is implemented TLBI RVAE2NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.39 TLBI RVAE2IS, TLBI RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBI RVAE2IS, TLBI RVAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE2IS, TLBI RVAE2ISNXS are UNDEFINED.

Attributes

TLBI RVAE2IS, TLBI RVAE2ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

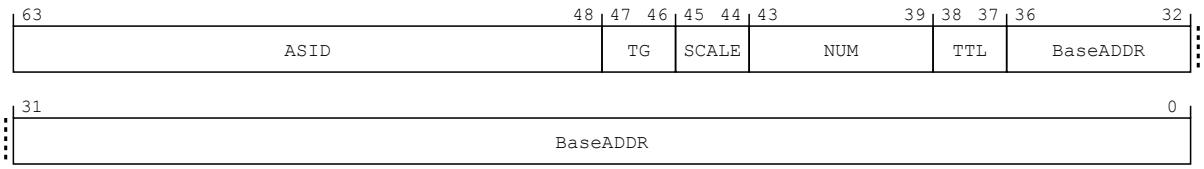

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2IS, TLBI RVAE2ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

When FEAT_XS is implemented TLBI RVAE2ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.40 TLBI RVAE2OS, TLBI RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBI RVAE2OS, TLBI RVAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE2OS, TLBI RVAE2OSNXS are UNDEFINED.

Attributes

TLBI RVAE2OS, TLBI RVAE2OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

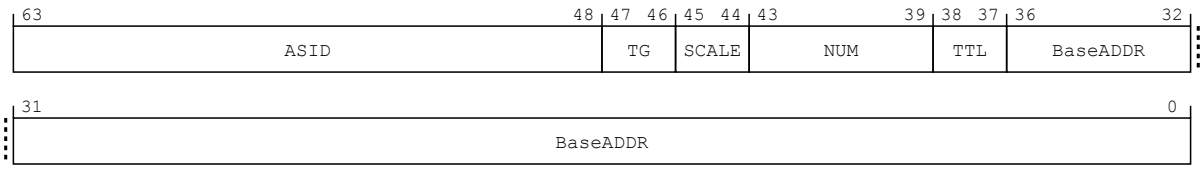

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2OS, TLBI RVAE2OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

When FEAT_XS is implemented TLBI RVAE2OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.41 TLBI RVAE3, TLBI RVAE3NXS, TLB Range Invalidate by VA, EL3

The TLBI RVAE3, TLBI RVAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE3, TLBI RVAE3NXS are UNDEFINED.

Attributes

TLBI RVAE3, TLBI RVAE3NXS is a 64-bit System instruction.

Field descriptions

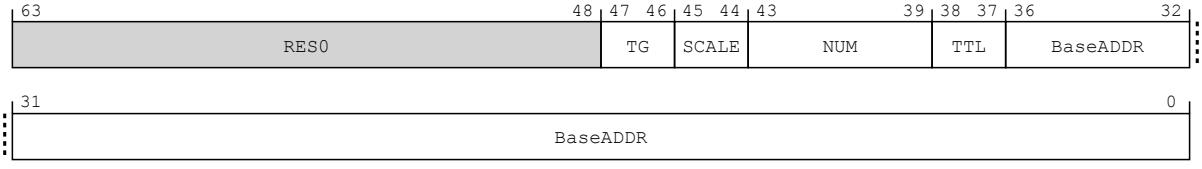

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3, TLBI RVAE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVAE3NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.42 TLBI RVAE3IS, TLBI RVAE3ISNXS, TLB Range Invalidate by VA, EL3, Inner Shareable

The TLBI RVAE3IS, TLBI RVAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE3IS, TLBI RVAE3ISNXS are UNDEFINED.

Attributes

TLBI RVAE3IS, TLBI RVAE3ISNXS is a 64-bit System instruction.

Field descriptions

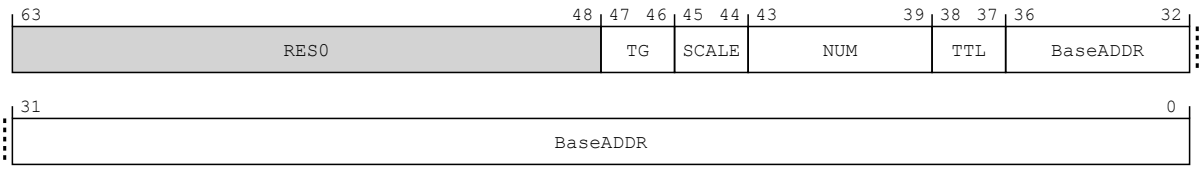

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3IS, TLBI RVAE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVAE3ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.43 TLBI RVAE3OS, TLBI RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBI RVAE3OS, TLBI RVAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVAE3OS, TLBI RVAE3OSNXS are UNDEFINED.

Attributes

TLBI RVAE3OS, TLBI RVAE3OSNXS is a 64-bit System instruction.

Field descriptions

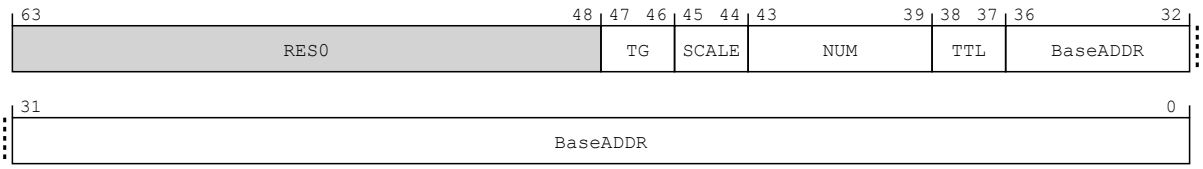

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3OS, TLBI RVAE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVAE3OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.44 TLBI RVALE1, TLBI RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1

The TLBI RVALE1, TLBI RVALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE1, TLBI RVALE1NXS are UNDEFINED.

Attributes

TLBI RVALE1, TLBI RVALE1NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

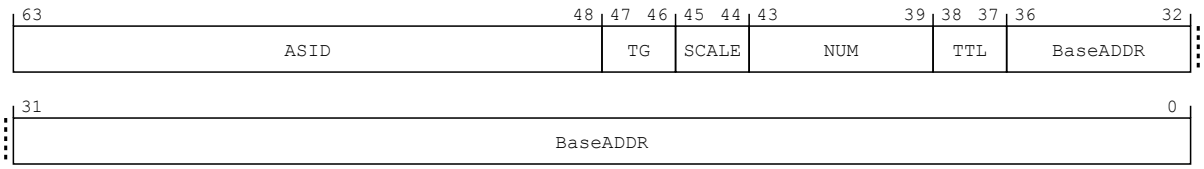

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
`0b00`	Reserved.
`0b01`	4K translation granule.
`0b10`	16K translation granule.
`0b11`	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1, TLBI RVALE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVALE1NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.45 TLBI RVALE1IS, TLBI RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI RVALE1IS, TLBI RVALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE1IS, TLBI RVALE1ISNXS are UNDEFINED.

Attributes

TLBI RVALE1IS, TLBI RVALE1ISNXS is a 64-bit System instruction.

Field descriptions

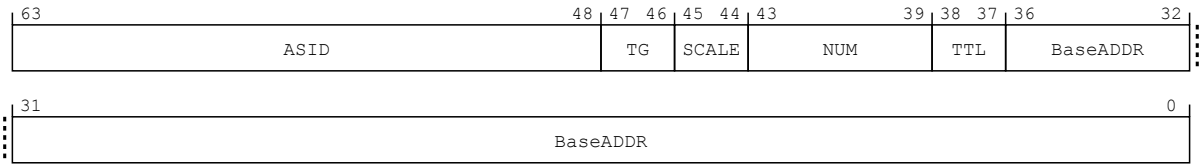

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1IS, TLBI RVALE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGITR_EL2().TLBIRVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVALE1ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.46 TLBI RVALE1OS, TLBI RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI RVALE1OS, TLBI RVALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE1OS, TLBI RVALE1OSNXS are UNDEFINED.

Attributes

TLBI RVALE1OS, TLBI RVALE1OSNXS is a 64-bit System instruction.

Field descriptions

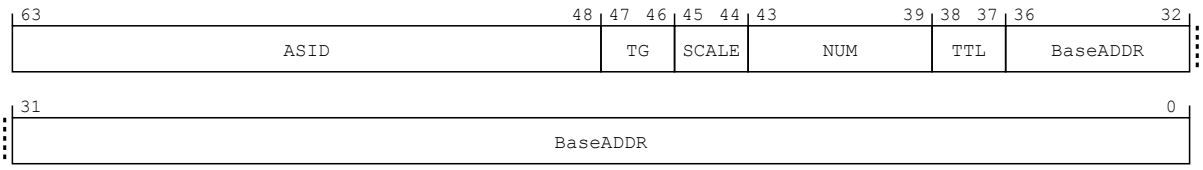

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL1.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL1.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1OS, TLBI RVALE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGITR_EL2().TLBIRVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI RVALE1OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
        ↪'0') && HFGITR_EL2().TLBIRVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.47 TLBI RVALE2, TLBI RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2

The TLBI RVALE2, TLBI RVALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE2, TLBI RVALE2NXS are UNDEFINED.

Attributes

TLBI RVALE2, TLBI RVALE2NXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

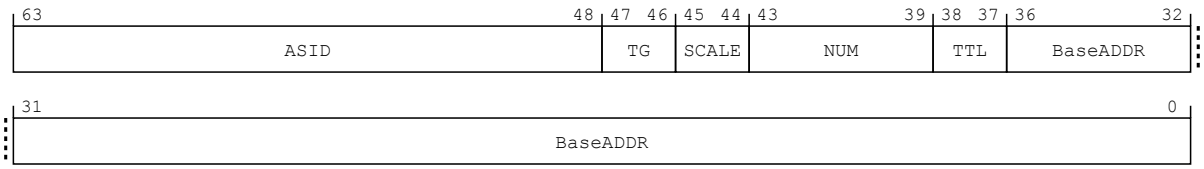

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2, TLBI RVALE2NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

When FEAT_XS is implemented TLBI RVALE2NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.48 TLBI RVALE2IS, TLBI RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI RVALE2IS, TLBI RVALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE2IS, TLBI RVALE2ISNXS are UNDEFINED.

Attributes

TLBI RVALE2IS, TLBI RVALE2ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

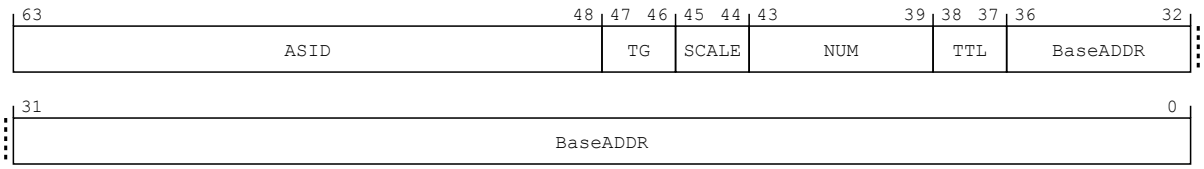

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2IS, TLBI RVALE2ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

When FEAT_XS is implemented TLBI RVALE2ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.49 TLBI RVALE2OS, TLBI RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI RVALE2OS, TLBI RVALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE2OS, TLBI RVALE2OSNXS are UNDEFINED.

Attributes

TLBI RVALE2OS, TLBI RVALE2OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

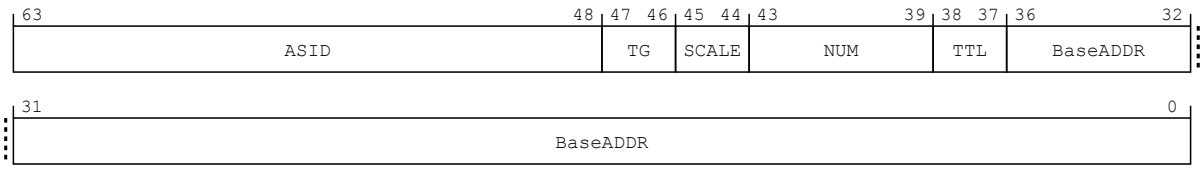

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL2.DS == '1')) || (IsFeatureImplemented(FEAT_D128) && (TCR2_EL2.D128 == '1')):

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2OS, TLBI RVALE2OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see AArch64_TLBI_RVA() in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

When FEAT_XS is implemented TLBI RVALE2OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
    ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.50 TLBI RVALE3, TLBI RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3

The TLBI RVALE3, TLBI RVALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 0000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE3, TLBI RVALE3NXS are UNDEFINED.

Attributes

TLBI RVALE3, TLBI RVALE3NXS is a 64-bit System instruction.

Field descriptions

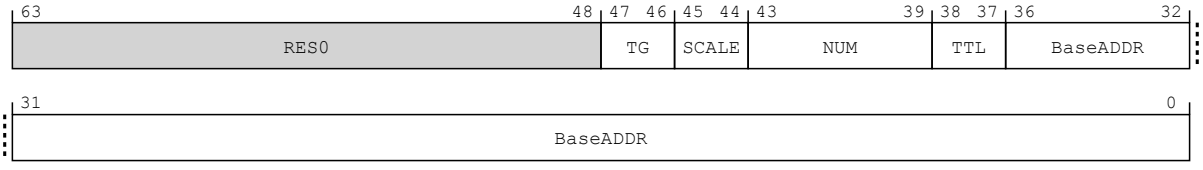

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3, TLBI RVALE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVALE3NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.51 TLBI RVALE3IS, TLBI RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI RVALE3IS, TLBI RVALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE3IS, TLBI RVALE3ISNXS are UNDEFINED.

Attributes

TLBI RVALE3IS, TLBI RVALE3ISNXS is a 64-bit System instruction.

Field descriptions

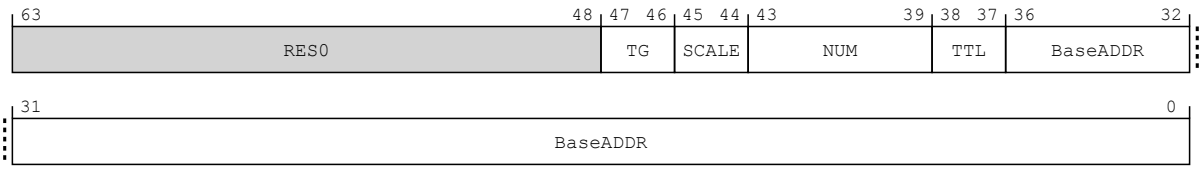

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3IS, TLBI RVALE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVALE3ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.52 TLBI RVALE3OS, TLBI RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI RVALE3OS, TLBI RVALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 64-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented, FEAT_TLBIOS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI RVALE3OS, TLBI RVALE3OSNXS are UNDEFINED.

Attributes

TLBI RVALE3OS, TLBI RVALE3OSNXS is a 64-bit System instruction.

Field descriptions

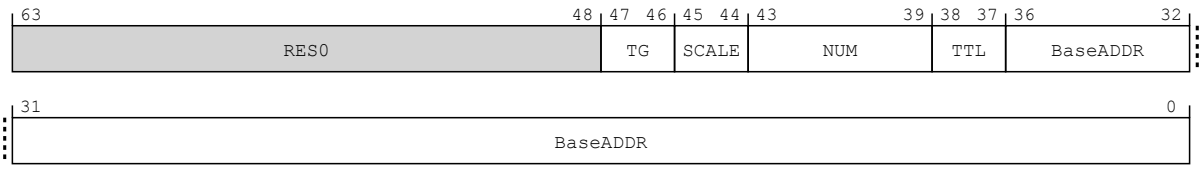

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

**When (IsFeatureImplemented(FEAT_LPA2) && (TCR_EL3.DS == '1')) ||
(IsFeatureImplemented(FEAT_D128) && (TCR_EL3.D128 == '1')):**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.

When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3OS, TLBI RVALE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI RVALE3OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.53 TLBI VAAE1, TLBI VAAE1NXS, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1, TLBI VAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAAE1, TLBI VAAE1NXS are UNDEFINED.

Attributes

TLBI VAAE1, TLBI VAAE1NXS is a 64-bit System instruction.

Field descriptions

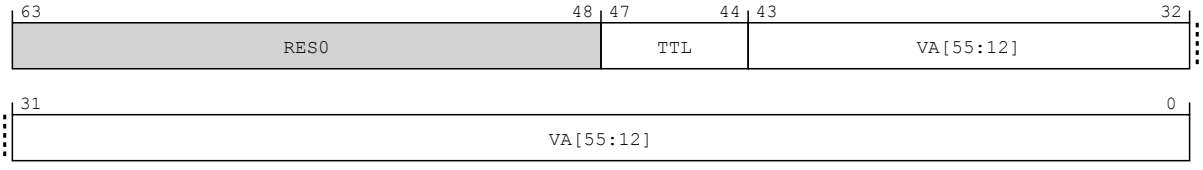

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1, TLBI VAAE1NXS

This system instruction is an alias of the SYS instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAAE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.54 TLBI VAAE1IS, TLBI VAAE1ISNXS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS, TLBI VAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAAE1IS, TLBI VAAE1ISNXS are UNDEFINED.

Attributes

TLBI VAAE1IS, TLBI VAAE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

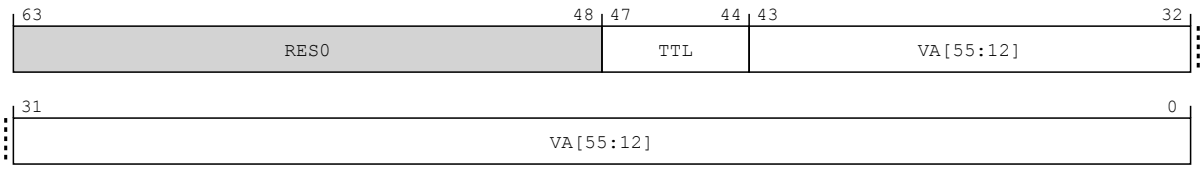

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1IS, TLBI VAAE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAAE1ISNXS{, <Xt>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.55 TLBI VAAE1OS, TLBI VAAE1OSNXS, TLB Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI VAAE1OS, TLBI VAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAAE1OS, TLBI VAAE1OSNXS are UNDEFINED.

Attributes

TLBI VAAE1OS, TLBI VAAE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

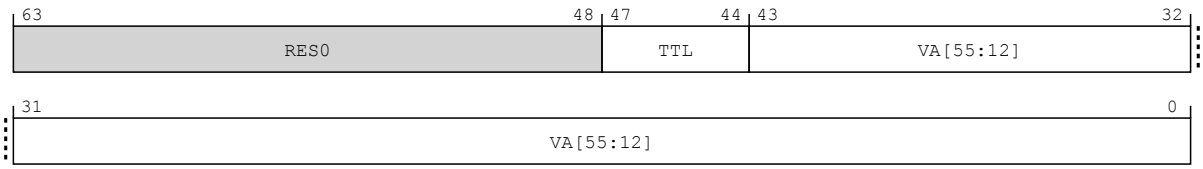

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1OS, TLBI VAAE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAAE1OSNXS{, <Xt>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.56 TLBI VAALE1, TLBI VAALE1NXS, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1, TLBI VAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAALE1, TLBI VAALE1NXS are UNDEFINED.

Attributes

TLBI VAALE1, TLBI VAALE1NXS is a 64-bit System instruction.

Field descriptions

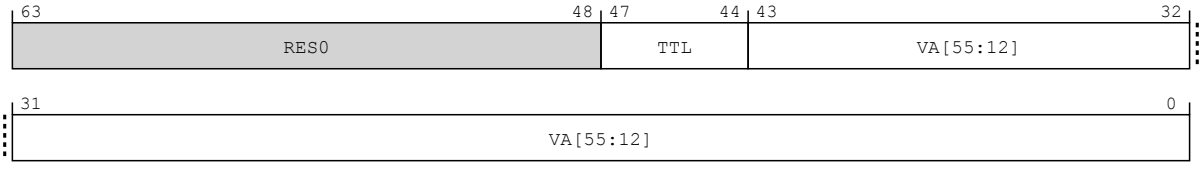

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1, TLBI VAALE1NXS

This system instruction is an alias of the SYS instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAALE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.57 TLBI VAALE1IS, TLBI VAALE1ISNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI VAALE1IS, TLBI VAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAALE1IS, TLBI VAALE1ISNXS are UNDEFINED.

Attributes

TLBI VAALE1IS, TLBI VAALE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

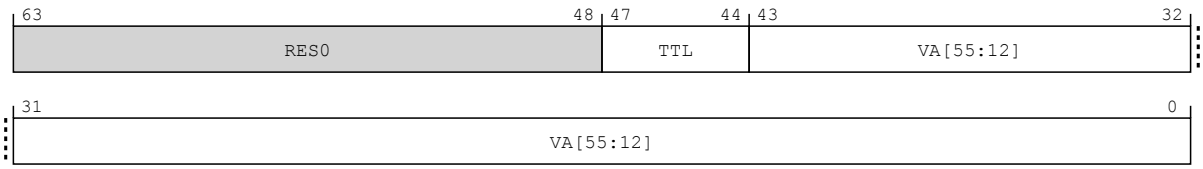

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1IS, TLBI VAALE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAALE1ISNXS{, <Xt>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.58 TLBI VAALE1OS, TLBI VAALE1OSNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI VAALE1OS, TLBI VAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAALE1OS, TLBI VAALE1OSNXS are UNDEFINED.

Attributes

TLBI VAALE1OS, TLBI VAALE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

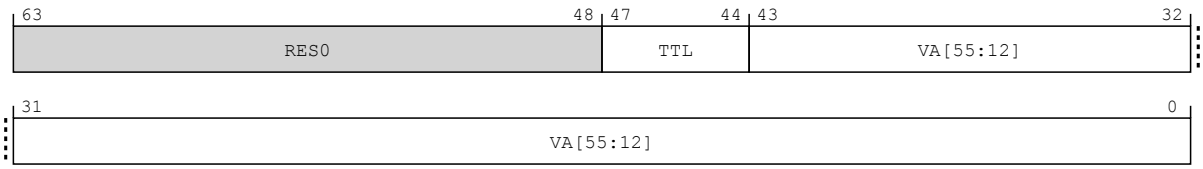

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1OS, TLBI VAALE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAALE1OSNXS{, <Xt>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
             ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.59 TLBI VAE1, TLBI VAE1NXS, TLB Invalidate by VA, EL1

The TLBI VAE1, TLBI VAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE1, TLBI VAE1NXS are UNDEFINED.

Attributes

TLBI VAE1, TLBI VAE1NXS is a 64-bit System instruction.

Field descriptions

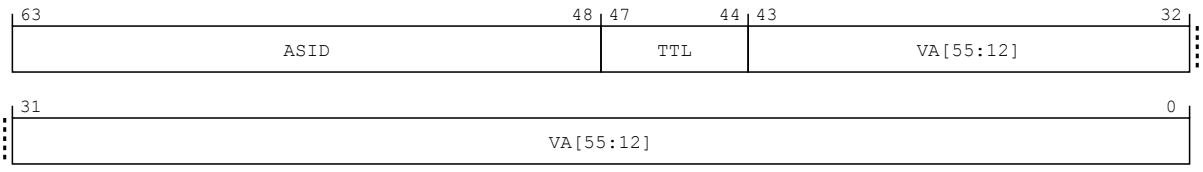

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VAE1, TLBI VAE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.60 TLBI VAE1IS, TLBI VAE1ISNXS, TLB Invalidate by VA, EL1, Inner Shareable

The TLBI VAE1IS, TLBI VAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE1IS, TLBI VAE1ISNXS are UNDEFINED.

Attributes

TLBI VAE1IS, TLBI VAE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

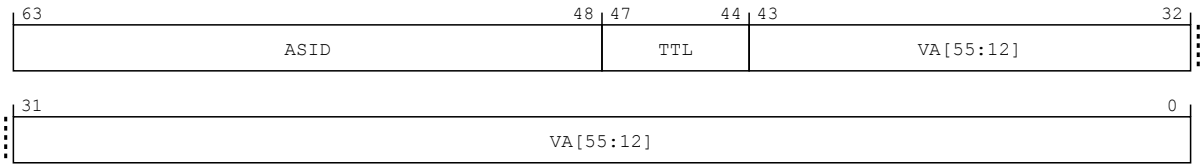

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1IS, TLBI VAE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VAE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.61 TLBI VAE1OS, TLBI VAE1OSNXS, TLB Invalidate by VA, EL1, Outer Shareable

The TLBI VAE1OS, TLBI VAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE1OS, TLBI VAE1OSNXS are UNDEFINED.

Attributes

TLBI VAE1OS, TLBI VAE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

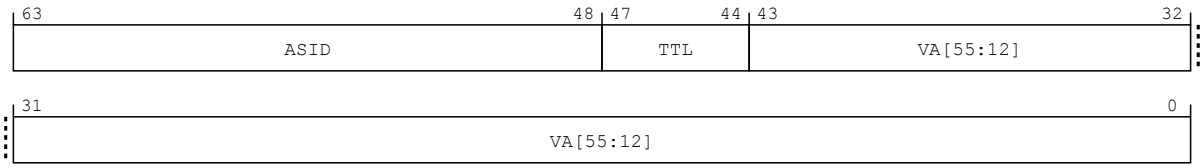

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1OS, TLBI VAE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VAE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.62 TLBI VAE2, TLBI VAE2NXS, TLB Invalidate by VA, EL2

The TLBI VAE2, TLBI VAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE2, TLBI VAE2NXS are UNDEFINED.

Attributes

TLBI VAE2, TLBI VAE2NXS is a 64-bit System instruction.

Field descriptions

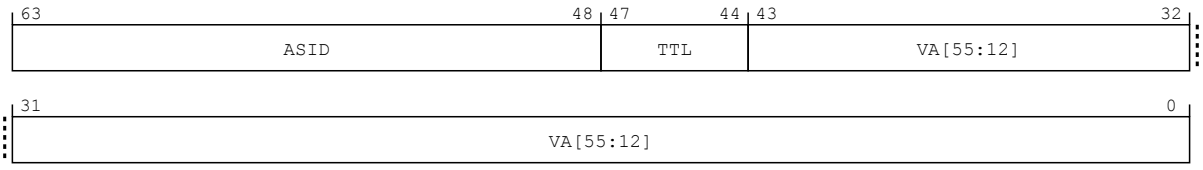

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VAE2, TLBI VAE2NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.63 TLBI VAE2IS, TLBI VAE2ISNXS, TLB Invalidate by VA, EL2, Inner Shareable

The TLBI VAE2IS, TLBI VAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE2IS, TLBI VAE2ISNXS are UNDEFINED.

Attributes

TLBI VAE2IS, TLBI VAE2ISNXS is a 64-bit System instruction.

Field descriptions

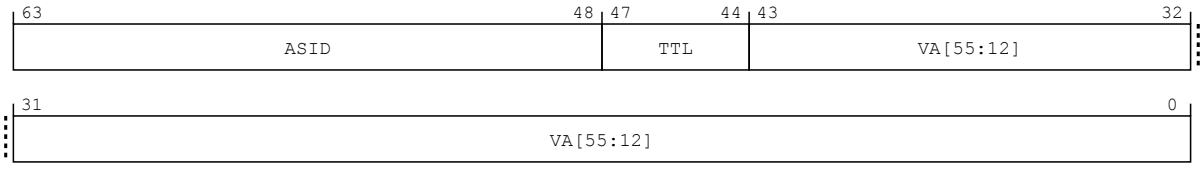

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VAE2IS, TLBI VAE2ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAE2ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.64 TLBI VAE2OS, TLBI VAE2OSNXS, TLB Invalidate by VA, EL2, Outer Shareable

The TLBI VAE2OS, TLBI VAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE2OS, TLBI VAE2OSNXS are UNDEFINED.

Attributes

TLBI VAE2OS, TLBI VAE2OSNXS is a 64-bit System instruction.

Field descriptions

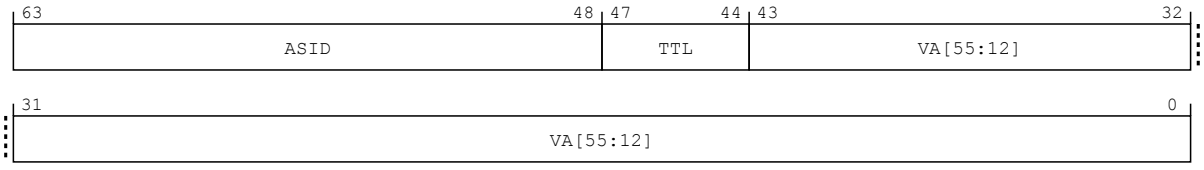

ASID, bits [63:48]

When ELIsInHost(EL2):

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2OS, TLBI VAE2OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VAE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.65 TLBI VAE3, TLBI VAE3NXS, TLB Invalidate by VA, EL3

The TLBI VAE3, TLBI VAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE3, TLBI VAE3NXS are UNDEFINED.

Attributes

TLBI VAE3, TLBI VAE3NXS is a 64-bit System instruction.

Field descriptions

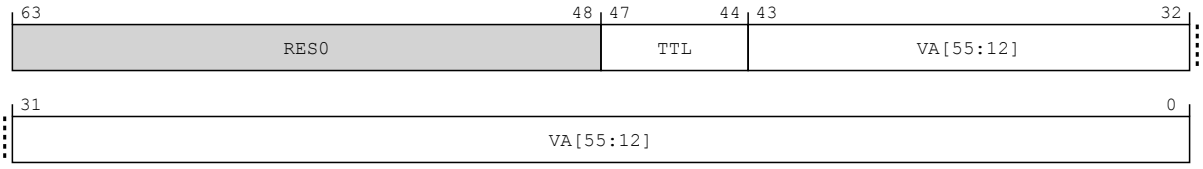

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3, TLBI VAE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0111`	`0b001`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI VAE3NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.66 TLBI VAE3IS, TLBI VAE3ISNXS, TLB Invalidate by VA, EL3, Inner Shareable

The TLBI VAE3IS, TLBI VAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE3IS, TLBI VAE3ISNXS are UNDEFINED.

Attributes

TLBI VAE3IS, TLBI VAE3ISNXS is a 64-bit System instruction.

Field descriptions

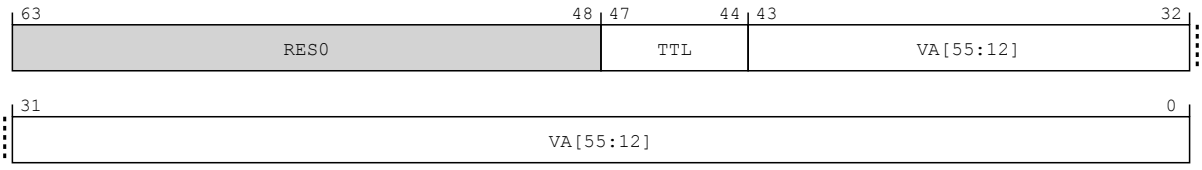

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3IS, TLBI VAE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0011`	`0b001`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI VAE3ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.67 TLBI VAE3OS, TLBI VAE3OSNXS, TLB Invalidate by VA, EL3, Outer Shareable

The TLBI VAE3OS, TLBI VAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VAE3OS, TLBI VAE3OSNXS are UNDEFINED.

Attributes

TLBI VAE3OS, TLBI VAE3OSNXS is a 64-bit System instruction.

Field descriptions

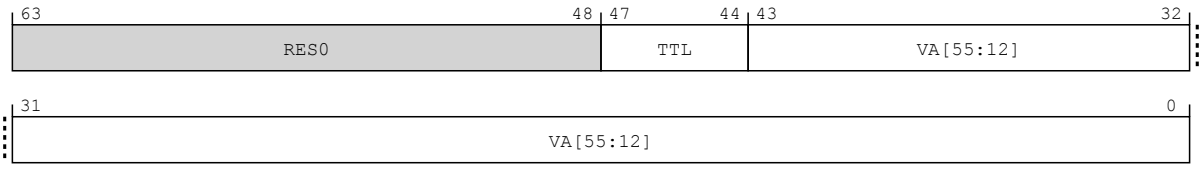

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3OS, TLBI VAE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0001`	`0b001`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI VAE3OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Any, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.68 TLBI VALE1, TLBI VALE1NXS, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1, TLBI VALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE1, TLBI VALE1NXS are UNDEFINED.

Attributes

TLBI VALE1, TLBI VALE1NXS is a 64-bit System instruction.

Field descriptions

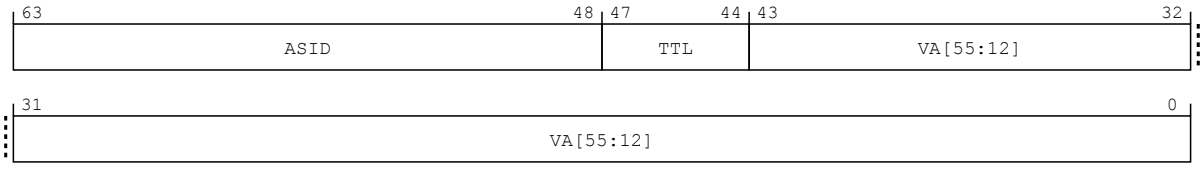

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VALE1, TLBI VALE1NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VALE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.69 TLBI VALE1IS, TLBI VALE1ISNXS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS, TLBI VALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE1IS, TLBI VALE1ISNXS are UNDEFINED.

Attributes

TLBI VALE1IS, TLBI VALE1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

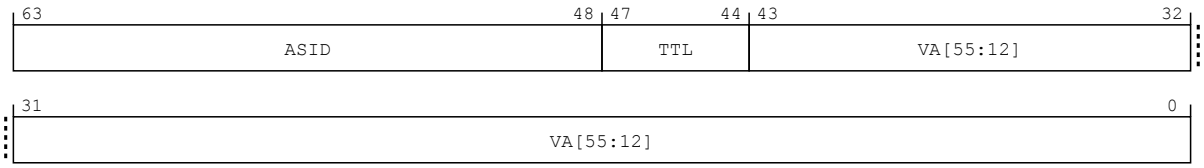

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1IS, TLBI VALE1ISNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VALE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.70 TLBI VALE1OS, TLBI VALE1OSNXS, TLB Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI VALE1OS, TLBI VALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE1OS, TLBI VALE1OSNXS are UNDEFINED.

Attributes

TLBI VALE1OS, TLBI VALE1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

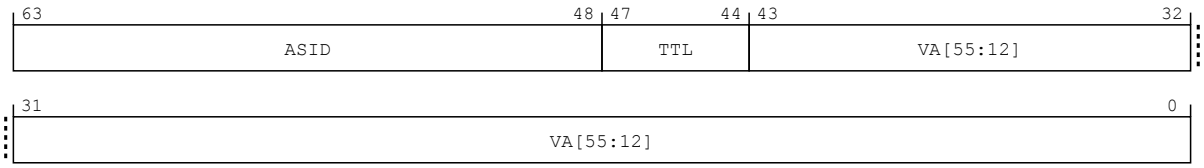

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1OS, TLBI VALE1OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_XS is implemented TLBI VALE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.71 TLBI VALE2, TLBI VALE2NXS, TLB Invalidate by VA, Last level, EL2

The TLBI VALE2, TLBI VALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE2, TLBI VALE2NXS are UNDEFINED.

Attributes

TLBI VALE2, TLBI VALE2NXS is a 64-bit System instruction.

Field descriptions

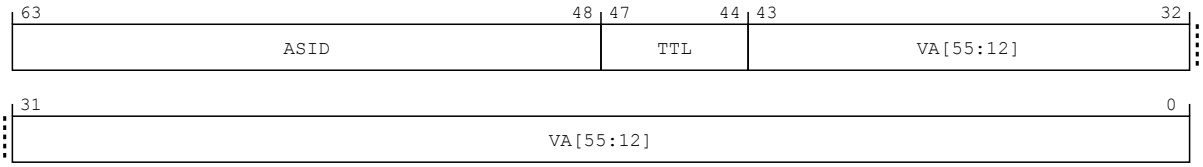

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2, TLBI VALE2NXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VALE2NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.72 TLBI VALE2IS, TLBI VALE2ISNXS, TLB Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI VALE2IS, TLBI VALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE2IS, TLBI VALE2ISNXS are UNDEFINED.

Attributes

TLBI VALE2IS, TLBI VALE2ISNXS is a 64-bit System instruction.

Field descriptions

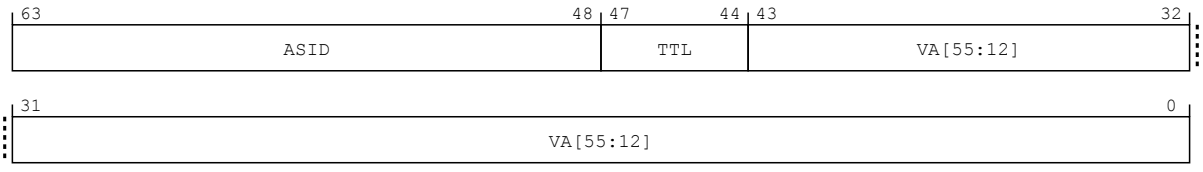

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2IS, TLBI VALE2ISNXS

This system instruction is an alias of the SYS instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VALE2ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.73 TLBI VALE2OS, TLBI VALE2OSNXS, TLB Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI VALE2OS, TLBI VALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE2OS, TLBI VALE2OSNXS are UNDEFINED.

Attributes

TLBI VALE2OS, TLBI VALE2OSNXS is a 64-bit System instruction.

Field descriptions

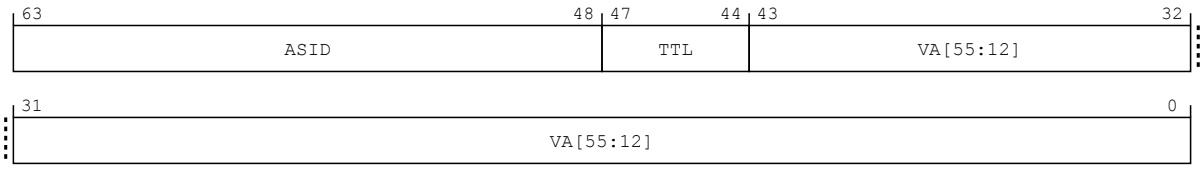

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2OS, TLBI VALE2OSNXS

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VALE2OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.74 TLBI VALE3, TLBI VALE3NXS, TLB Invalidate by VA, Last level, EL3

The TLBI VALE3, TLBI VALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE3, TLBI VALE3NXS are UNDEFINED.

Attributes

TLBI VALE3, TLBI VALE3NXS is a 64-bit System instruction.

Field descriptions

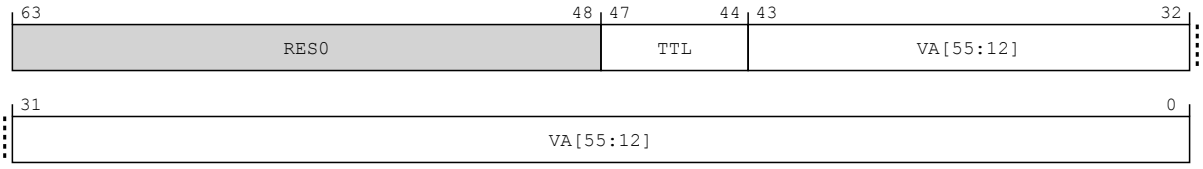

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3, TLBI VALE3NXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0111`	`0b101`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI VALE3NXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.75 TLBI VALE3IS, TLBI VALE3ISNXS, TLB Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI VALE3IS, TLBI VALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE3IS, TLBI VALE3ISNXS are UNDEFINED.

Attributes

TLBI VALE3IS, TLBI VALE3ISNXS is a 64-bit System instruction.

Field descriptions

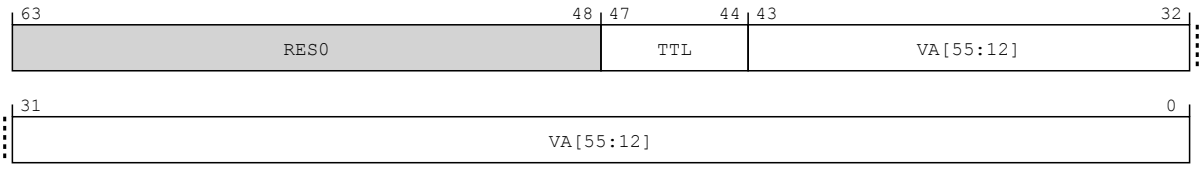

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3IS, TLBI VALE3ISNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0011`	`0b101`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI VALE3ISNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.76 TLBI VALE3OS, TLBI VALE3OSNXS, TLB Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI VALE3OS, TLBI VALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk.
  - If FEAT_D128 is implemented, a 128-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VALE3OS, TLBI VALE3OSNXS are UNDEFINED.

Attributes

TLBI VALE3OS, TLBI VALE3OSNXS is a 64-bit System instruction.

Field descriptions

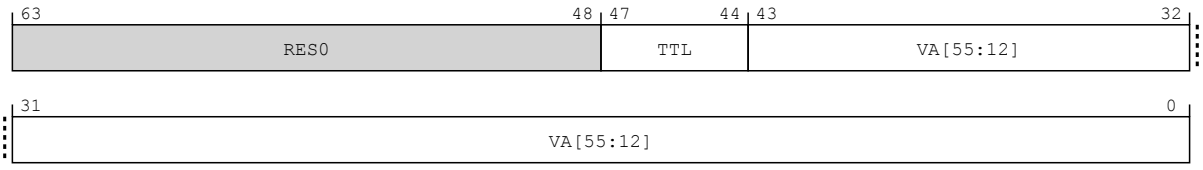

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is `0b00`. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3OS, TLBI VALE3OSNXS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b110`	`0b1000`	`0b0001`	`0b101`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{64}(t));
    end;
end;
```

When FEAT_XS is implemented TLBI VALE3OSNXS{, <Xt>}

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH,
            ↪TLBILevel_Last, TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.77 TLBI VMALLE1, TLBI VMALLE1NXS, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1, TLBI VMALLE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLE1, TLBI VMALLE1NXS are UNDEFINED.

Attributes

TLBI VMALLE1, TLBI VMALLE1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1, TLBI VMALLE1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VMALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBI VMALLE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVMALLE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLE1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVMALLE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_ExcludeXS,
         ↪X{64}(t));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
            ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.78 TLBI VMALLE1IS, TLBI VMALLE1ISNXS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS, TLBI VMALLE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLE1IS, TLBI VMALLE1ISNXS are UNDEFINED.

Attributes

TLBI VMALLE1IS, TLBI VMALLE1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLE1IS, TLBI VMALLE1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VMALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVMALLE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLE1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVMALLE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_ExcludeXS,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.79 TLBI VMALLE1OS, TLBI VMALLE1OSNXS, TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable

The TLBI VMALLE1OS, TLBI VMALLE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLE1OS, TLBI VMALLE1OSNXS are UNDEFINED.

Attributes

TLBI VMALLE1OS, TLBI VMALLE1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLE1OS, TLBI VMALLE1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBI_VMALL()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVMALLE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
             ↪X{64}(t));
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLE1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS ==
         ↪'0') && HFGITR_EL2().TLBIVMALLE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_ExcludeXS,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
             ↪TLBI_ExcludeXS, X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.80 TLBI VMALLS12E1, TLBI VMALLS12E1NXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1

The TLBI VMALLS12E1, TLBI VMALLS12E1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLS12E1, TLBI VMALLS12E1NXS are UNDEFINED.

Attributes

TLBI VMALLS12E1, TLBI VMALLS12E1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1, TLBI VMALLS12E1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VMALLS12E1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
        ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_NSH, TLBI_AllAttr,
            ↪X{64}(t));
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

`When FEAT_XS is implemented TLBI VMALLS12E1NXS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_NSH,
            ↪TLBI_ExcludeXS, X{64}(t));
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.81 TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable

The TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS are UNDEFINED.

Attributes

TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
        ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_ISH, TLBI_AllAttr,
            ↪X{64}(t));
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLS12E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_ISH,
            ↪TLBI_ExcludeXS, X{64}(t));
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.82 TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable

The TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS are UNDEFINED.

Attributes

TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS is a 64-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
        ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_OSH, TLBI_AllAttr,
            ↪X{64}(t));
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLS12E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIOS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Broadcast_OSH,
            ↪TLBI_ExcludeXS, X{64}(t));
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.83 TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS, TLB Invalidate stage 2 dirty state by VMID, EL1&0

The TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS characteristics are:

Purpose

Invalidates stage 2 write permissions from cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIW is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS are UNDEFINED.

Attributes

TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS is a 64-bit System instruction.

Field descriptions

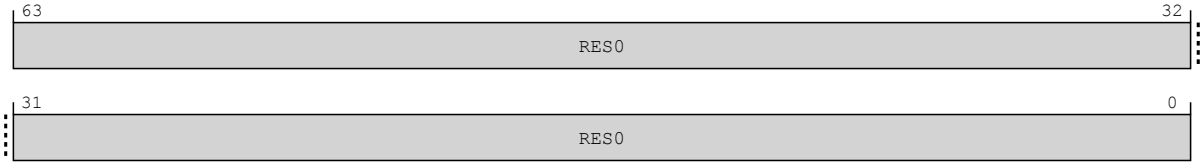

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:0]

Reserved, RES0.

Executing TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLWS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
         ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLWS2E1NXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
        ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.84 TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS, TLB Invalidate stage 2 dirty state by VMID, EL1&0, Inner Shareable

The TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS characteristics are:

Purpose

Invalidates stage 2 write permissions from cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIW is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS are UNDEFINED.

Attributes

TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLWS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
        ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLWS2E1ISNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.85 TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS, TLB Invalidate stage 2 write permission by VMID, EL1&0, Outer Shareable

The TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS characteristics are:

Purpose

Invalidates stage 2 write permissions from cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_TLBIW is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS are UNDEFINED.

Attributes

TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLWS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBI_AllAttr,
        ↪X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBI_AllAttr, X{64}(t));
        end;
    end;
end;
```

```
When FEAT_XS is implemented TLBI VMALLWS2E1OSNXS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_TLBIW) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
         ↪TLBI_ExcludeXS, X{64}(t));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBI_VMALLWS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBI_ExcludeXS, X{64}(t));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.86 TLBIP IPAS2E1, TLBIP IPAS2E1NXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1

The TLBIP IPAS2E1, TLBIP IPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2E1, TLBIP IPAS2E1NXS are UNDEFINED.

Attributes

TLBIP IPAS2E1, TLBIP IPAS2E1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

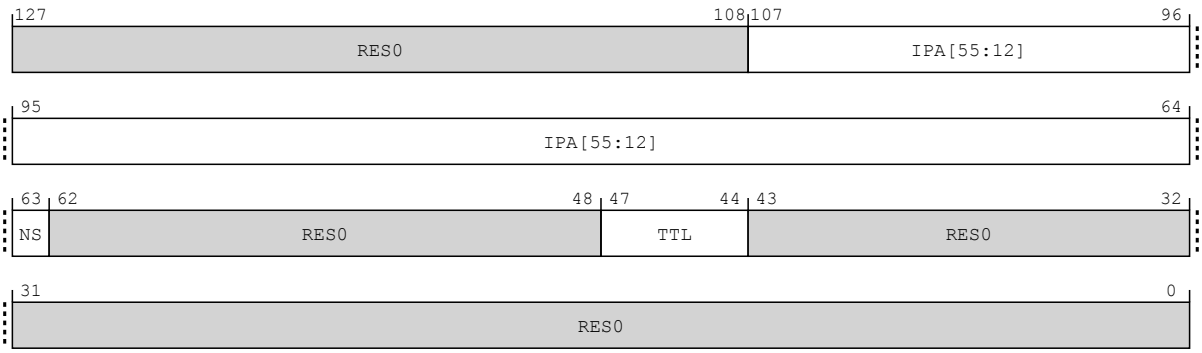

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2E1, TLBIP IPAS2E1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2E1{, <Xt>, <Xt2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

TLBIP IPAS2E1NXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.87 TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS are UNDEFINED.

Attributes

TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

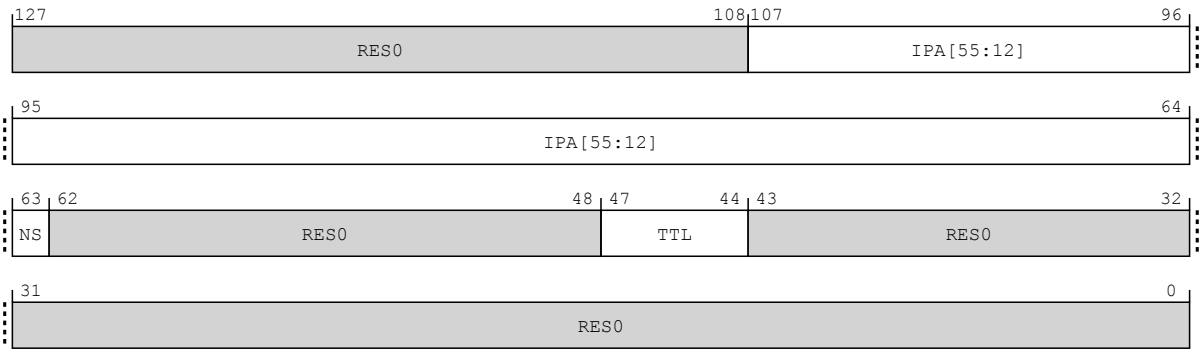

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2E1IS{, <Xt>, <Xt2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

TLBIP IPAS2E1ISNXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.88 TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS are UNDEFINED.

Attributes

TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

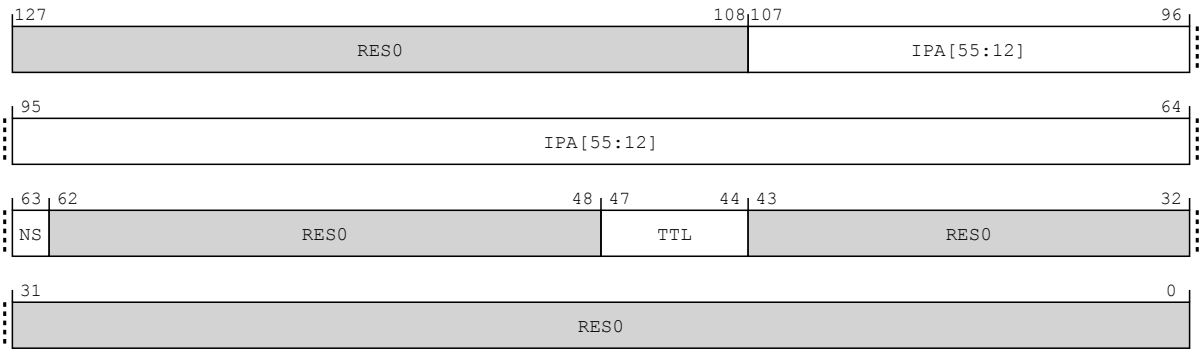

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2E1OS{, <Xt>, <Xt2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

TLBIP IPAS2E1OSNXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.89 TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS are UNDEFINED.

Attributes

TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

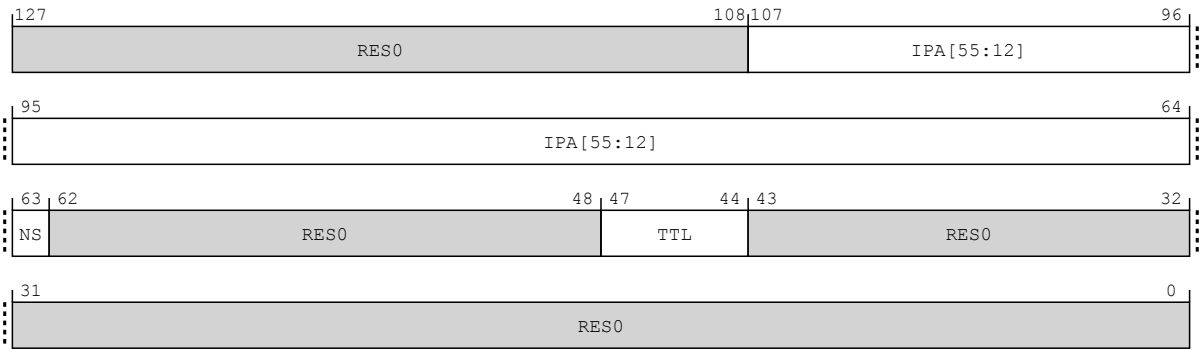

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2LE1{, <Xt>, <Xt2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

TLBIP IPAS2LE1NXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.90 TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS are UNDEFINED.

Attributes

TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

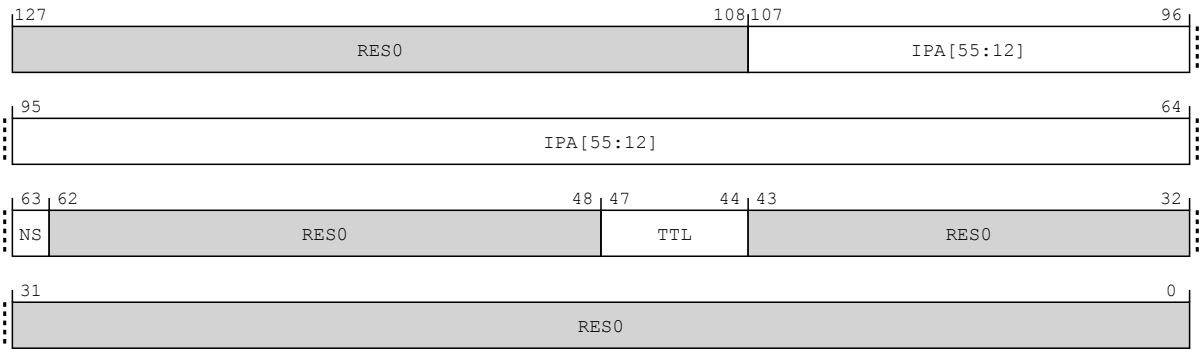

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2LE1IS{, <Xt>, <Xt2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

TLBIP IPAS2LE1ISNXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.91 TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS, TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 2 only translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

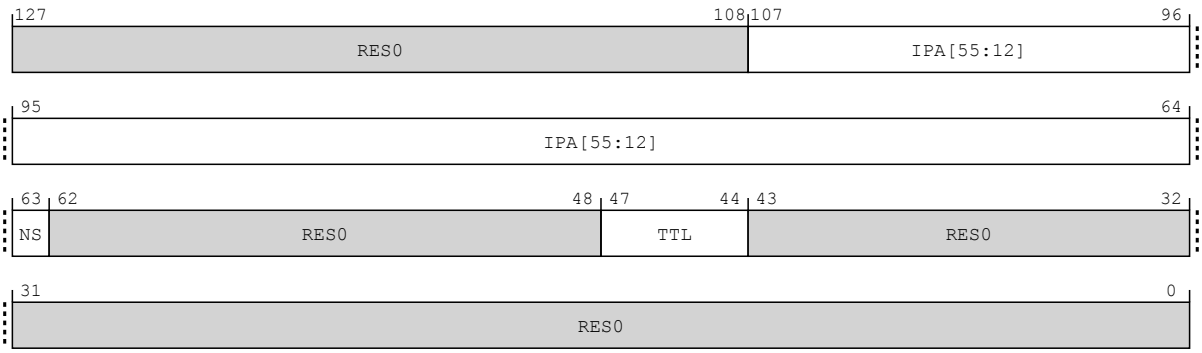

Bits [127:108]

Reserved, RES0.

IPA[55:12], bits [107:64]

Bits[55:12] of the intermediate physical address to match.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP IPAS2LE1OS{, <Xt>, <Xt2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

TLBIP IPAS2LE1OSNXS{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.92 TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS are UNDEFINED.

Attributes

TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

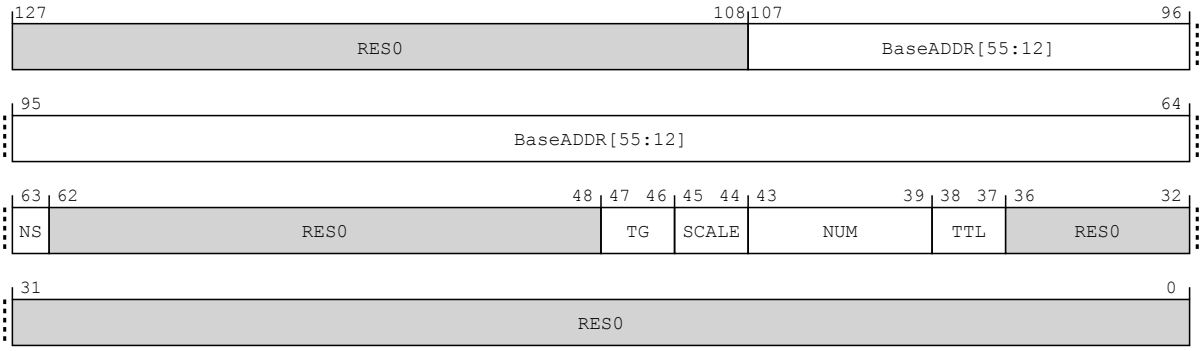

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2E1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RIPAS2E1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.93 TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS are UNDEFINED.

Attributes

TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

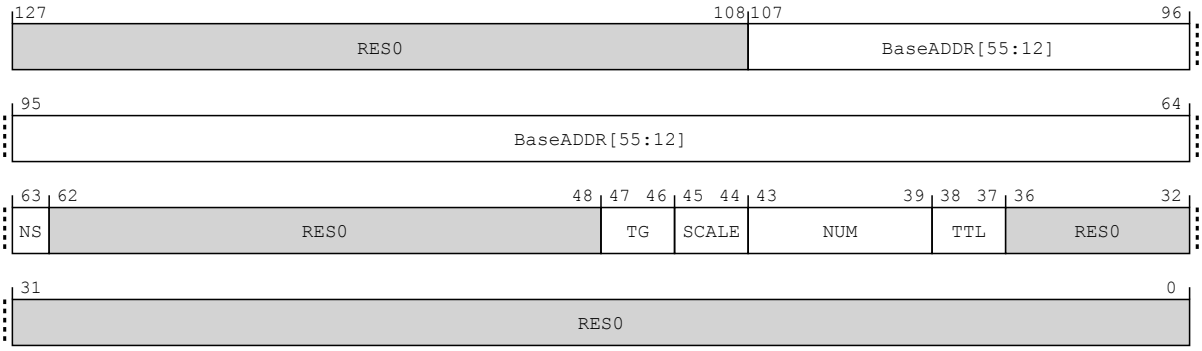

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2E1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RIPAS2E1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.94 TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 2 only translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS are UNDEFINED.

Attributes

TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

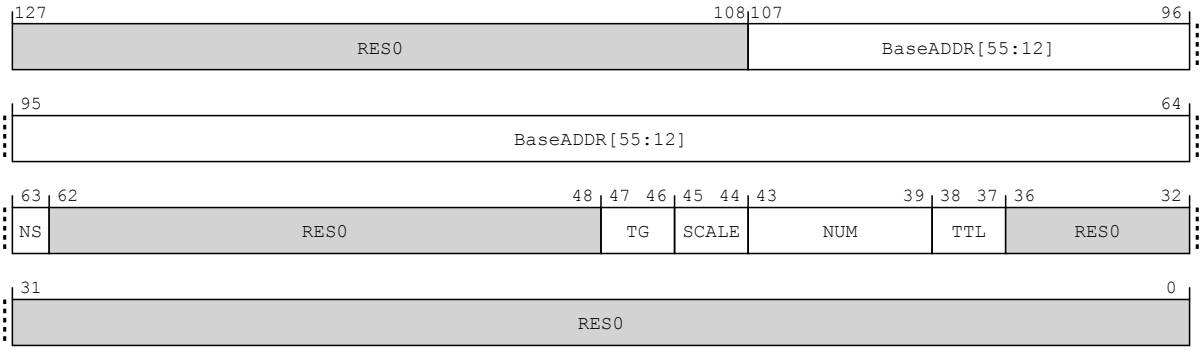

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2E1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RIPAS2E1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.95 TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS are UNDEFINED.

Attributes

TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

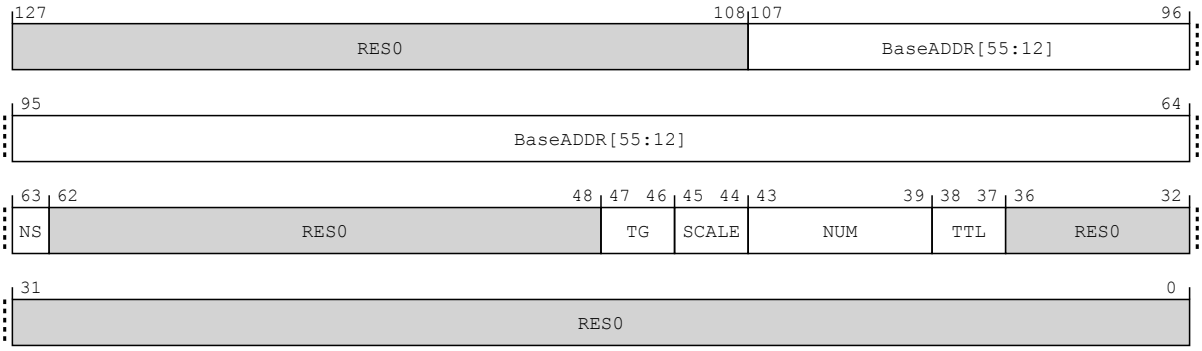

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2LE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RIPAS2LE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.96 TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS are UNDEFINED.

Attributes

TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

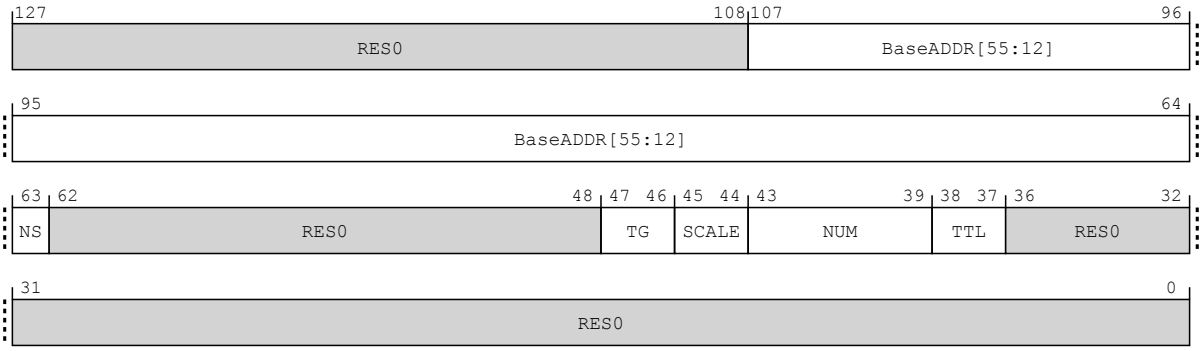

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RIPAS2LE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RIPAS2LE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.97 TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - If FEAT_D128 is implemented, a 64-bit stage 2 only translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS are UNDEFINED.

Attributes

TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS is a 128-bit System instruction.

Field descriptions

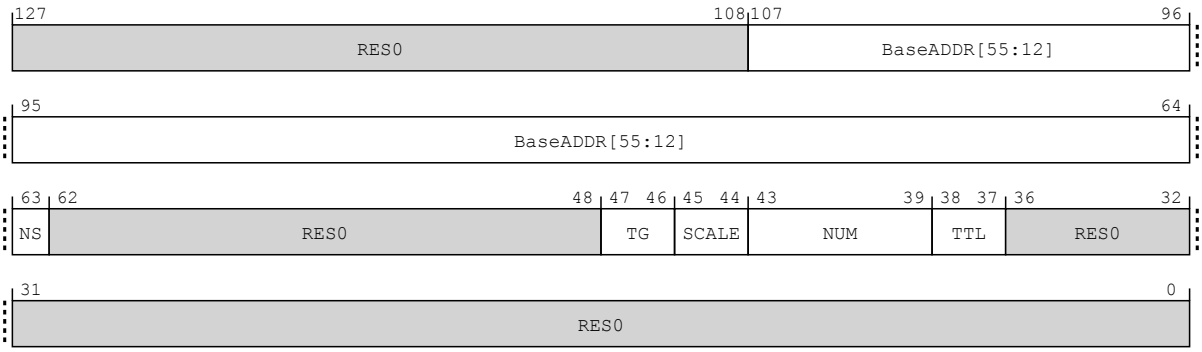

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RIPAS2LE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RIPAS2LE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0100	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
            return;
        else
            AArch64_TLBIP_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.98 TLBIP RVAAE1, TLBIP RVAAE1NXS, TLB Range Invalidate by VA, All ASID, EL1

The TLBIP RVAAE1, TLBIP RVAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAAE1, TLBIP RVAAE1NXS are UNDEFINED.

Attributes

TLBIP RVAAE1, TLBIP RVAAE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

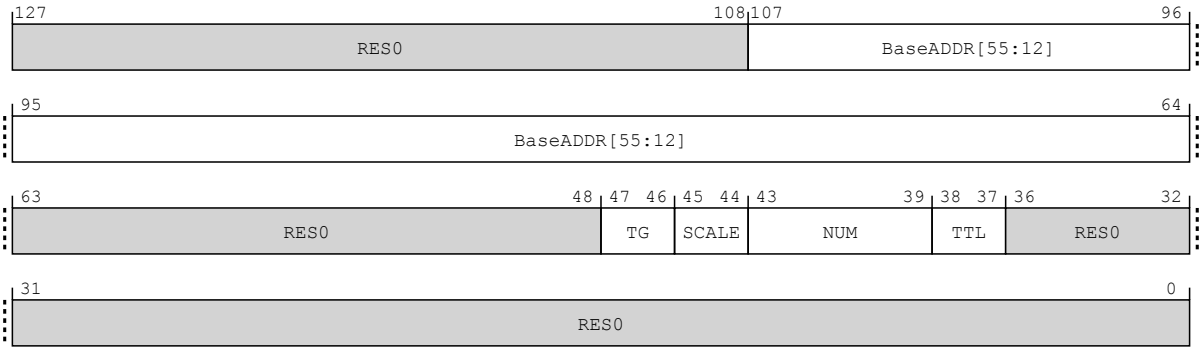

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAAE1, TLBIP RVAAE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.99 TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS is a 128-bit System instruction.

Field descriptions

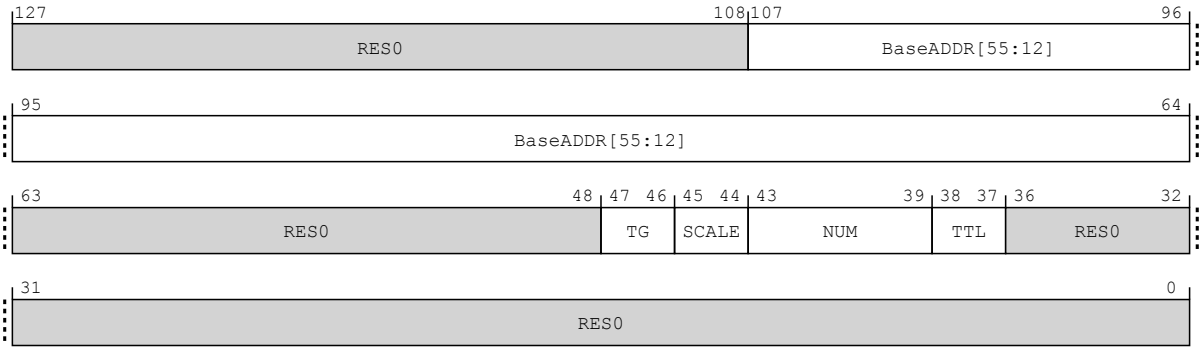

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVAAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.100 TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS is a 128-bit System instruction.

Field descriptions

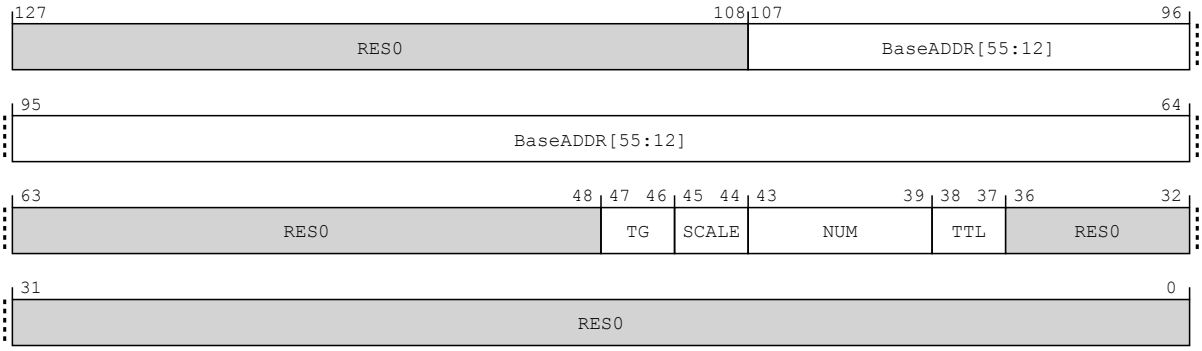

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVAAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.101 TLBIP RVAALE1, TLBIP RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBIP RVAALE1, TLBIP RVAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAALE1, TLBIP RVAALE1NXS are UNDEFINED.

Attributes

TLBIP RVAALE1, TLBIP RVAALE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

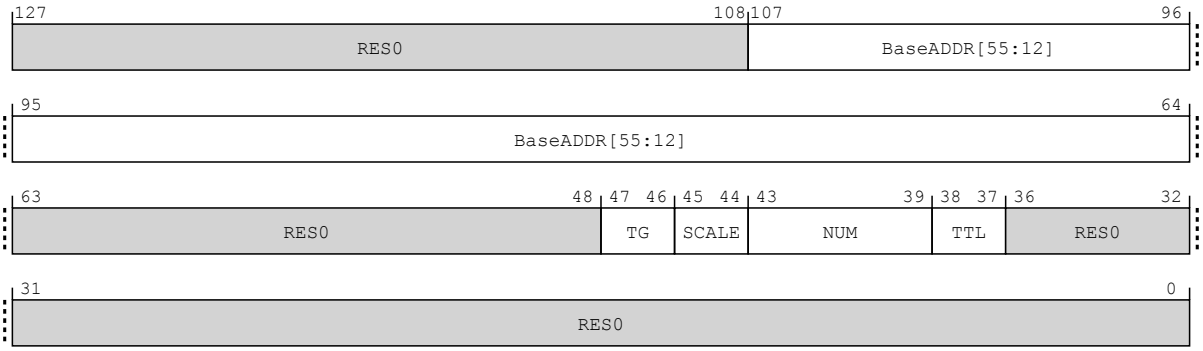

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAALE1, TLBIP RVAALE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.102 TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS is a 128-bit System instruction.

Field descriptions

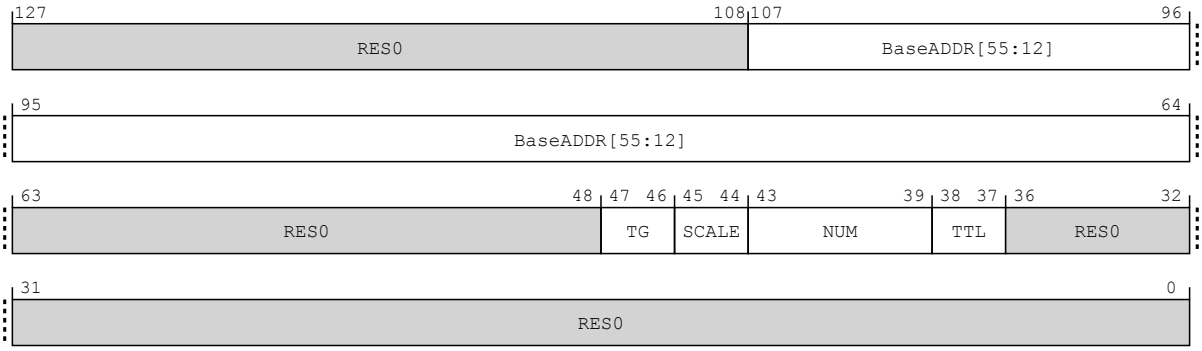

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVAALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.103 TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
  - A 64-bit stage 1 translation table entry, from the leaf level of the translation table walk, if TTL is `0b00`.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS is a 128-bit System instruction.

Field descriptions

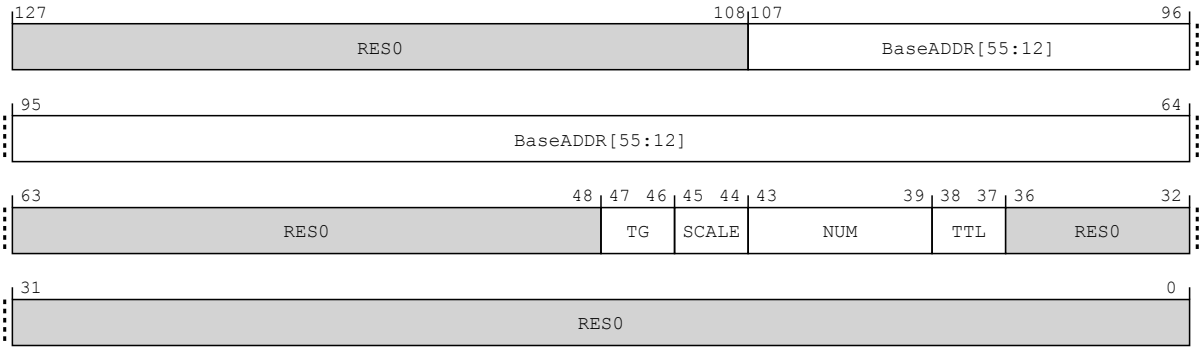

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVAALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.104 TLBIP RVAE1, TLBIP RVAE1NXS, TLB Range Invalidate by VA, EL1

The TLBIP RVAE1, TLBIP RVAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE1, TLBIP RVAE1NXS are UNDEFINED.

Attributes

TLBIP RVAE1, TLBIP RVAE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

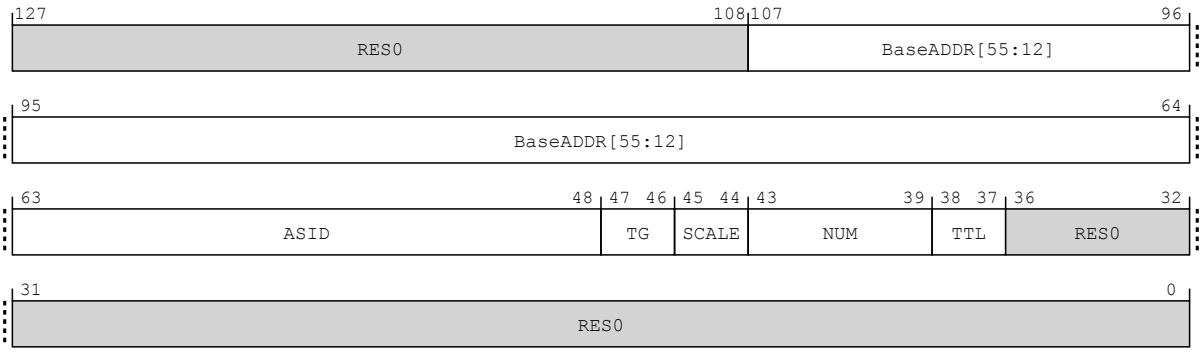

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE1, TLBIP RVAE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.105 TLBIP RVAE1IS, TLBIP RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBIP RVAE1IS, TLBIP RVAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE1IS, TLBIP RVAE1ISNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAE1IS, TLBIP RVAE1ISNXS is a 128-bit System instruction.

Field descriptions

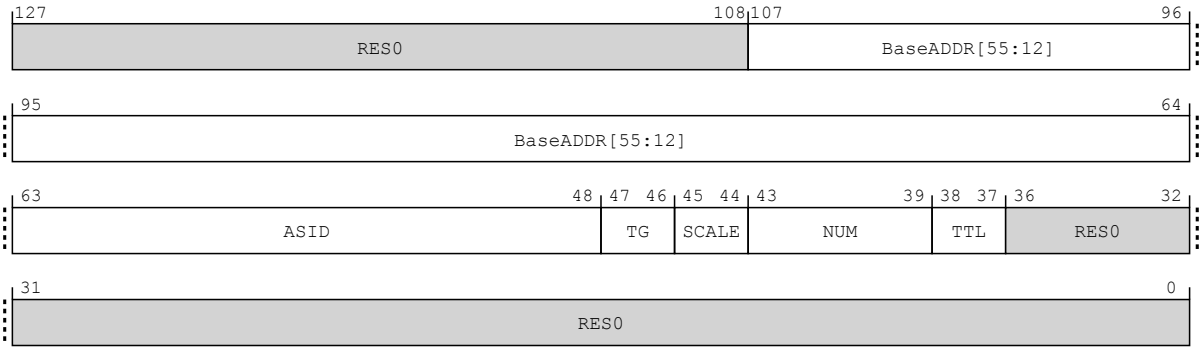

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE1IS, TLBIP RVAE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
        ↪HFGITR_EL2().TLBIRVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.106 TLBIP RVAE1OS, TLBIP RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBIP RVAE1OS, TLBIP RVAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE1OS, TLBIP RVAE1OSNXS are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

TLBIP RVAE1OS, TLBIP RVAE1OSNXS is a 128-bit System instruction.

Field descriptions

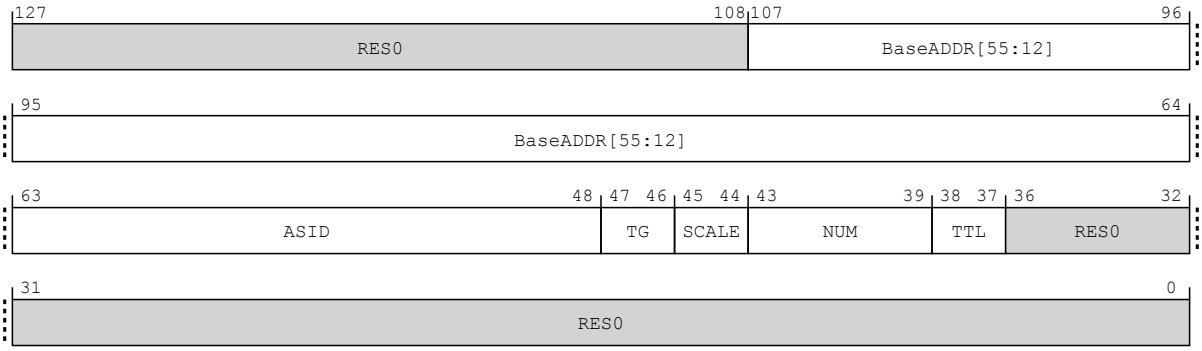

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE1OS, TLBIP RVAE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
        ↪HFGITR_EL2().TLBIRVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.107 TLBIP RVAE2, TLBIP RVAE2NXS, TLB Range Invalidate by VA, EL2

The TLBIP RVAE2, TLBIP RVAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE2, TLBIP RVAE2NXS are UNDEFINED.

Attributes

TLBIP RVAE2, TLBIP RVAE2NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

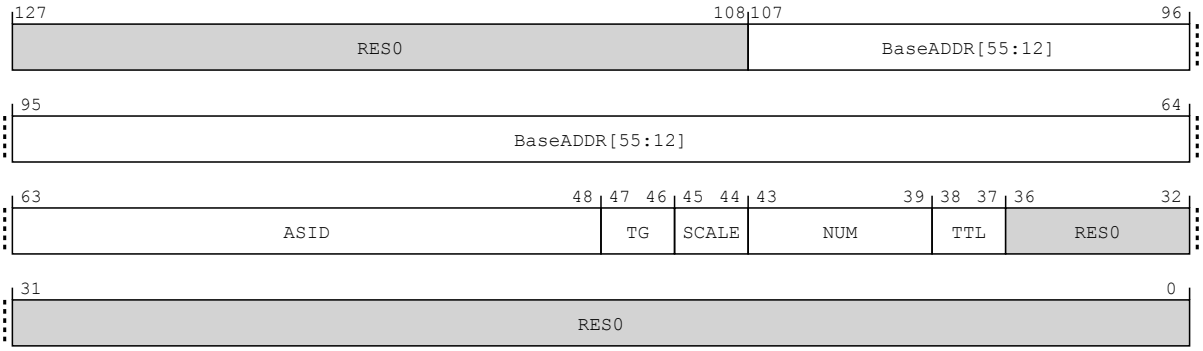

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE2, TLBIP RVAE2NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RVAE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.108 TLBIP RVAE2IS, TLBIP RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBIP RVAE2IS, TLBIP RVAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE2IS, TLBIP RVAE2ISNXS are UNDEFINED.

Attributes

TLBIP RVAE2IS, TLBIP RVAE2ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

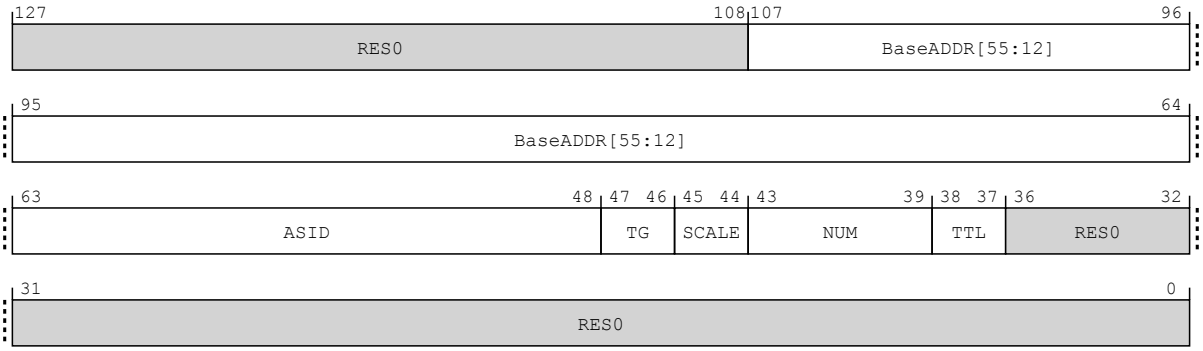

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE2IS, TLBIP RVAE2ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RVAE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.109 TLBIP RVAE2OS, TLBIP RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBIP RVAE2OS, TLBIP RVAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE2OS, TLBIP RVAE2OSNXS are UNDEFINED.

Attributes

TLBIP RVAE2OS, TLBIP RVAE2OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

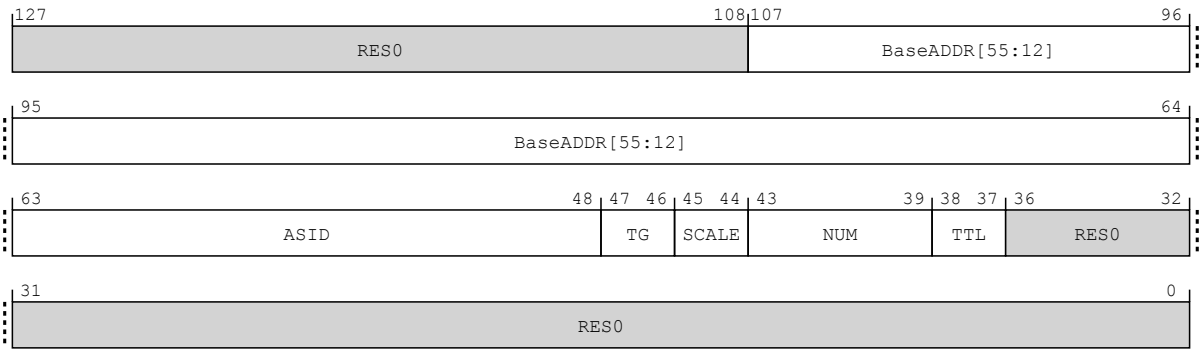

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE2OS, TLBIP RVAE2OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVAE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RVAE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.110 TLBIP RVAE3, TLBIP RVAE3NXS, TLB Range Invalidate by VA, EL3

The TLBIP RVAE3, TLBIP RVAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE3, TLBIP RVAE3NXS are UNDEFINED.

Attributes

TLBIP RVAE3, TLBIP RVAE3NXS is a 128-bit System instruction.

Field descriptions

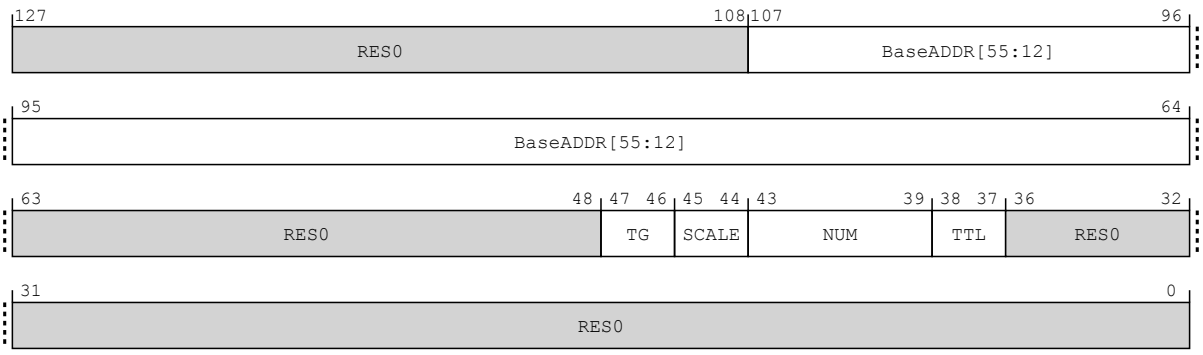

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE3, TLBIP RVAE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVAE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.111 TLBIP RVAE3IS, TLBIP RVAE3ISNXS, TLB Range Invalidate by VA, EL3, Inner Shareable

The TLBIP RVAE3IS, TLBIP RVAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE3IS, TLBIP RVAE3ISNXS are UNDEFINED.

Attributes

TLBIP RVAE3IS, TLBIP RVAE3ISNXS is a 128-bit System instruction.

Field descriptions

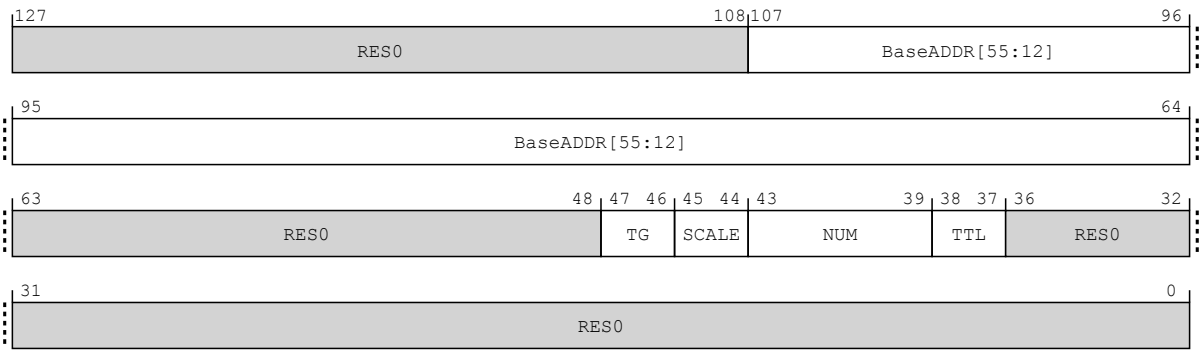

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE3IS, TLBIP RVAE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVAE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.112 TLBIP RVAE3OS, TLBIP RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBIP RVAE3OS, TLBIP RVAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVAE3OS, TLBIP RVAE3OSNXS are UNDEFINED.

Attributes

TLBIP RVAE3OS, TLBIP RVAE3OSNXS is a 128-bit System instruction.

Field descriptions

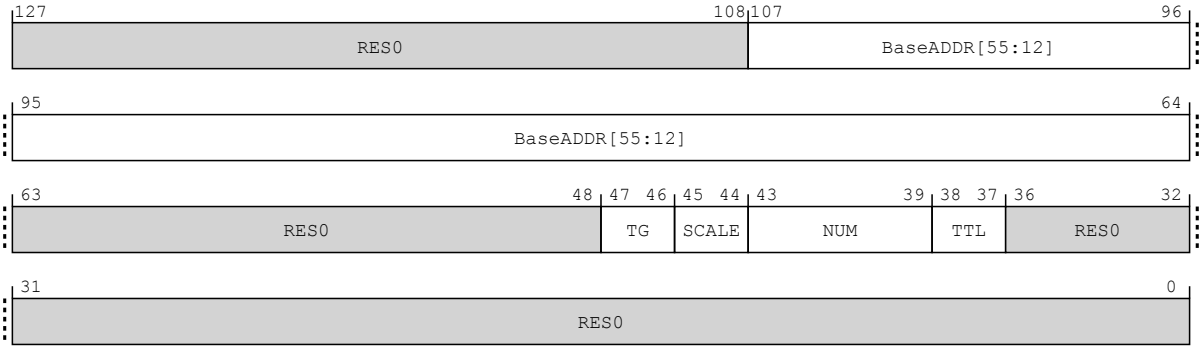

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVAE3OS, TLBIP RVAE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVAE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVAE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.113 TLBIP RVALE1, TLBIP RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1

The TLBIP RVALE1, TLBIP RVALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE1, TLBIP RVALE1NXS are UNDEFINED.

Attributes

TLBIP RVALE1, TLBIP RVALE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

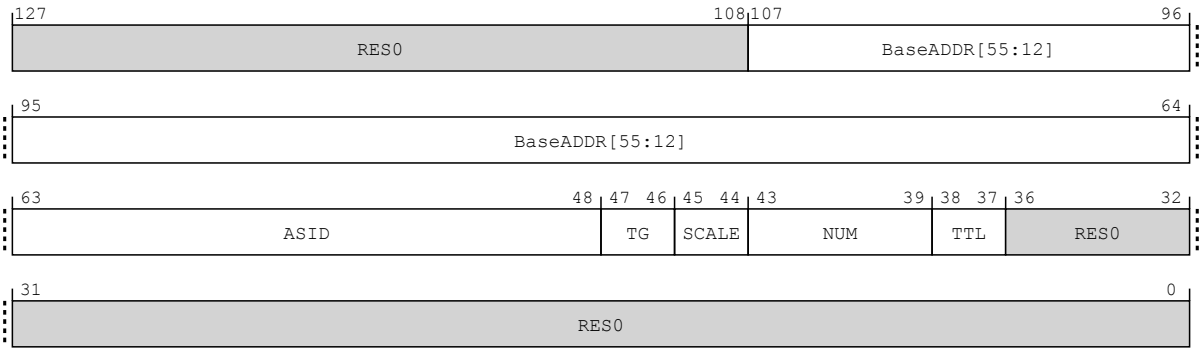

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE1, TLBIP RVALE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.114 TLBIP RVALE1IS, TLBIP RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBIP RVALE1IS, TLBIP RVALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE1IS, TLBIP RVALE1ISNXS are UNDEFINED.

Attributes

TLBIP RVALE1IS, TLBIP RVALE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

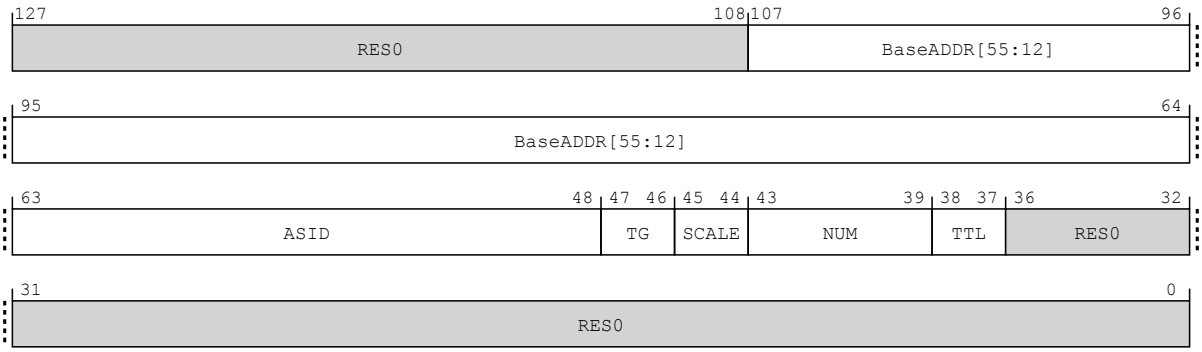

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE1IS, TLBIP RVALE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.115 TLBIP RVALE1OS, TLBIP RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBIP RVALE1OS, TLBIP RVALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE1OS, TLBIP RVALE1OSNXS are UNDEFINED.

Attributes

TLBIP RVALE1OS, TLBIP RVALE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

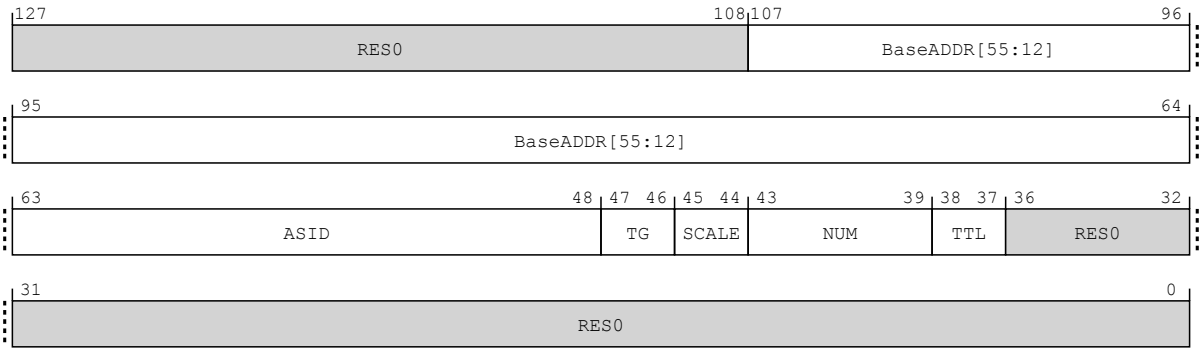

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE1OS, TLBIP RVALE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIRVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

```
TLBIP RVALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIRVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.116 TLBIP RVALE2, TLBIP RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2

The TLBIP RVALE2, TLBIP RVALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE2, TLBIP RVALE2NXS are UNDEFINED.

Attributes

TLBIP RVALE2, TLBIP RVALE2NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

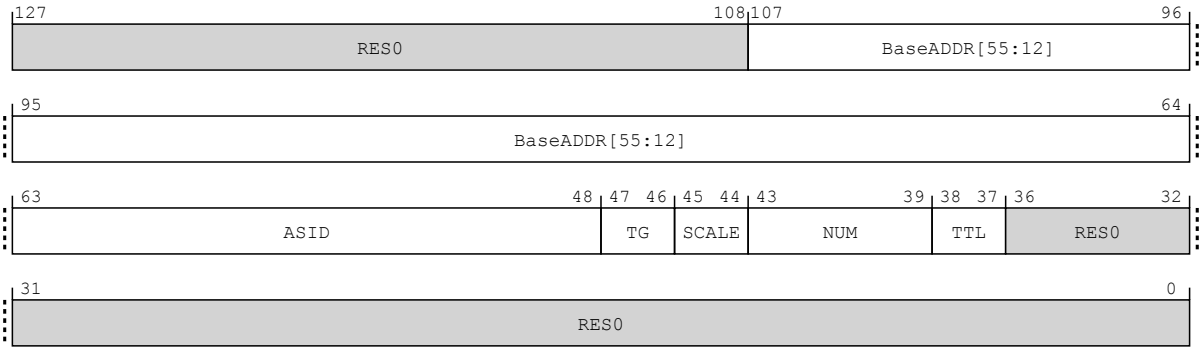

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE2, TLBIP RVALE2NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RVALE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.117 TLBIP RVALE2IS, TLBIP RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBIP RVALE2IS, TLBIP RVALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE2IS, TLBIP RVALE2ISNXS are UNDEFINED.

Attributes

TLBIP RVALE2IS, TLBIP RVALE2ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

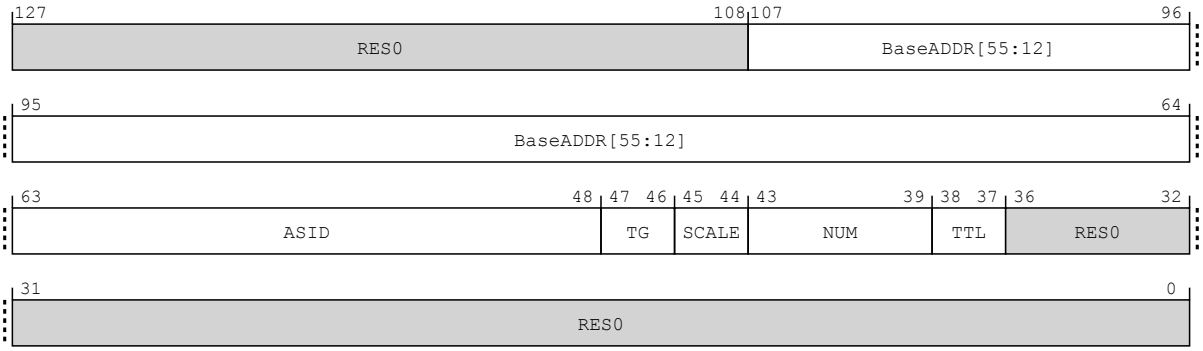

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE2IS, TLBIP RVALE2ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RVALE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.118 TLBIP RVALE2OS, TLBIP RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBIP RVALE2OS, TLBIP RVALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, if TTL is `0b00`.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE2OS, TLBIP RVALE2OSNXS are UNDEFINED.

Attributes

TLBIP RVALE2OS, TLBIP RVALE2OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

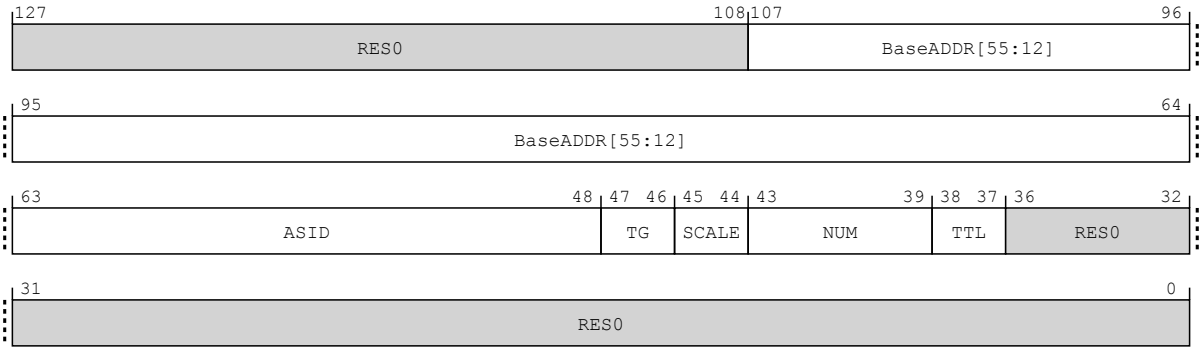

Bits [127:108]

Reserved, RES0.

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE2OS, TLBIP RVALE2OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_RVA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP RVALE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP RVALE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.119 TLBIP RVALE3, TLBIP RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3

The TLBIP RVALE3, TLBIP RVALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE3, TLBIP RVALE3NXS are UNDEFINED.

Attributes

TLBIP RVALE3, TLBIP RVALE3NXS is a 128-bit System instruction.

Field descriptions

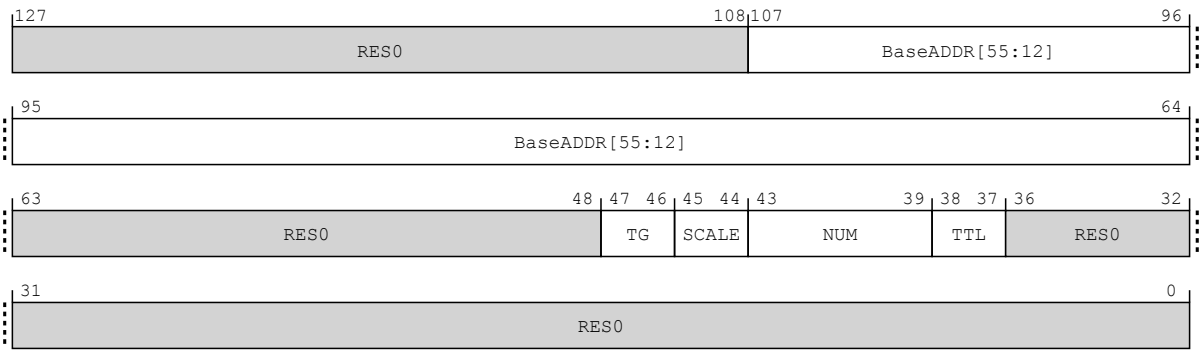

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE3, TLBIP RVALE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVALE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVALE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0110	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.120 TLBIP RVALE3IS, TLBIP RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBIP RVALE3IS, TLBIP RVALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE3IS, TLBIP RVALE3ISNXS are UNDEFINED.

Attributes

TLBIP RVALE3IS, TLBIP RVALE3ISNXS is a 128-bit System instruction.

Field descriptions

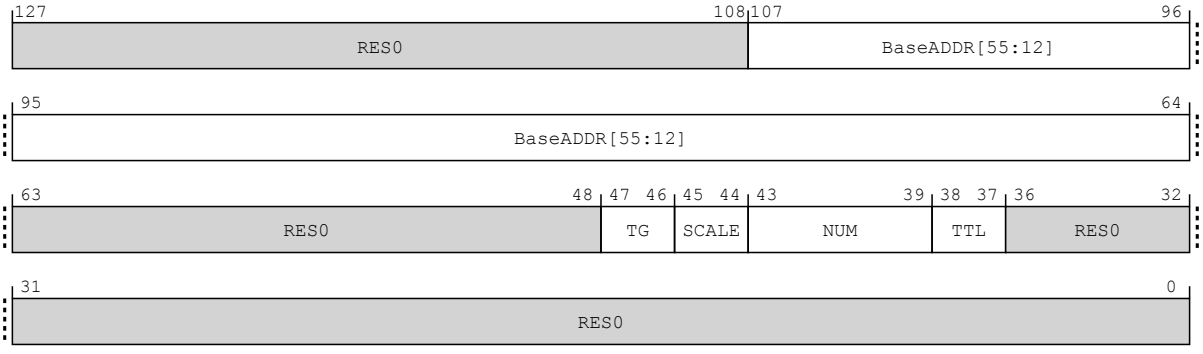

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE3IS, TLBIP RVALE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVALE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVALE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.121 TLBIP RVALE3OS, TLBIP RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

The TLBIP RVALE3OS, TLBIP RVALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL is `0b00`.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For 128-bit translation table entry, the range of addresses invalidated is UNPREDICTABLE when Block or Page size corresponding to TTL and TG, for the translation system is not aligned.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP RVALE3OS, TLBIP RVALE3OSNXS are UNDEFINED.

Attributes

TLBIP RVALE3OS, TLBIP RVALE3OSNXS is a 128-bit System instruction.

Field descriptions

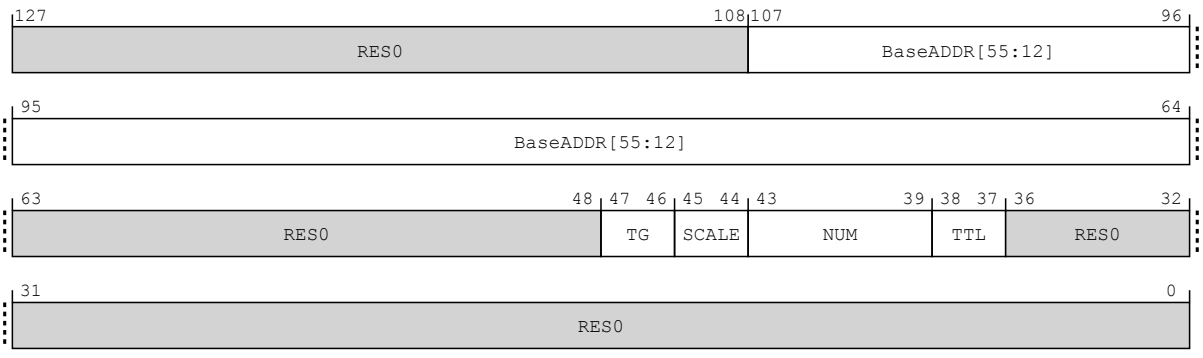

Bits [127:108]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR[55:12], bits [107:64]

The starting address for the range of the maintenance instructions. This field is BaseADDR[55:12] for all translation granules.

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	The TTL hint indicates level 1.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Bits [36:0]

Reserved, RES0.

Executing TLBIP RVALE3OS, TLBIP RVALE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP RVALE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP RVALE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.122 TLBIP VAAE1, TLBIP VAAE1NXS, TLB Invalidate Pair by VA, All ASID, EL1

The TLBIP VAAE1, TLBIP VAAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAAE1, TLBIP VAAE1NXS are UNDEFINED.

Attributes

TLBIP VAAE1, TLBIP VAAE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

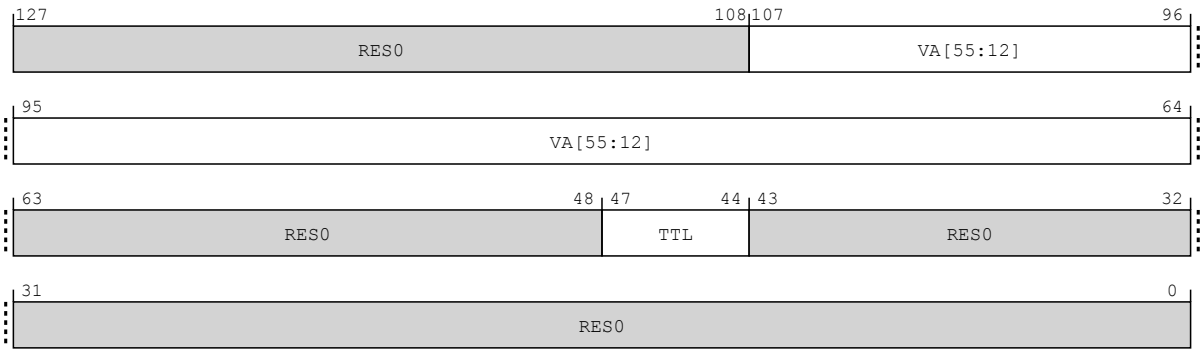

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAAE1, TLBIP VAAE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0111`	`0b011`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.123 TLBIP VAAE1IS, TLBIP VAAE1ISNXS, TLB Invalidate Pair by VA, All ASID, EL1, Inner Shareable

The TLBIP VAAE1IS, TLBIP VAAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAAE1IS, TLBIP VAAE1ISNXS are UNDEFINED.

Attributes

TLBIP VAAE1IS, TLBIP VAAE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

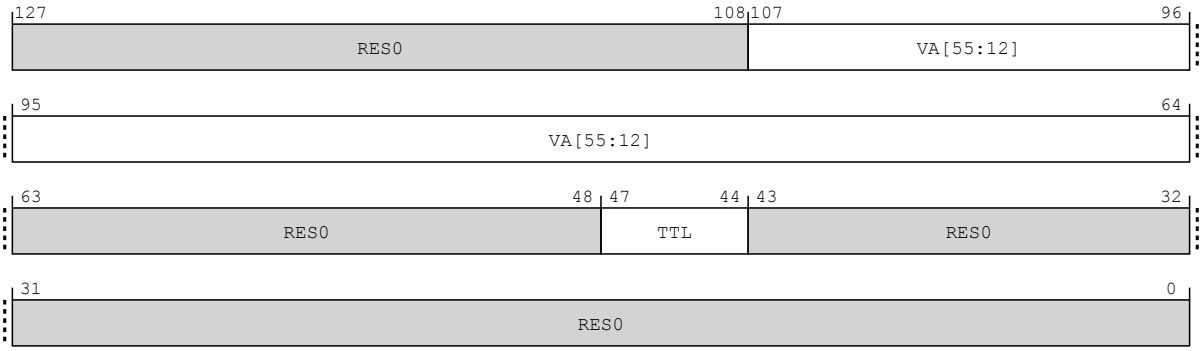

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAAE1IS, TLBIP VAAE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b011`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.124 TLBIP VAAE1OS, TLBIP VAAE1OSNXS, TLB Invalidate Pair by VA, All ASID, EL1, Outer Shareable

The TLBIP VAAE1OS, TLBIP VAAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAAE1OS, TLBIP VAAE1OSNXS are UNDEFINED.

Attributes

TLBIP VAAE1OS, TLBIP VAAE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

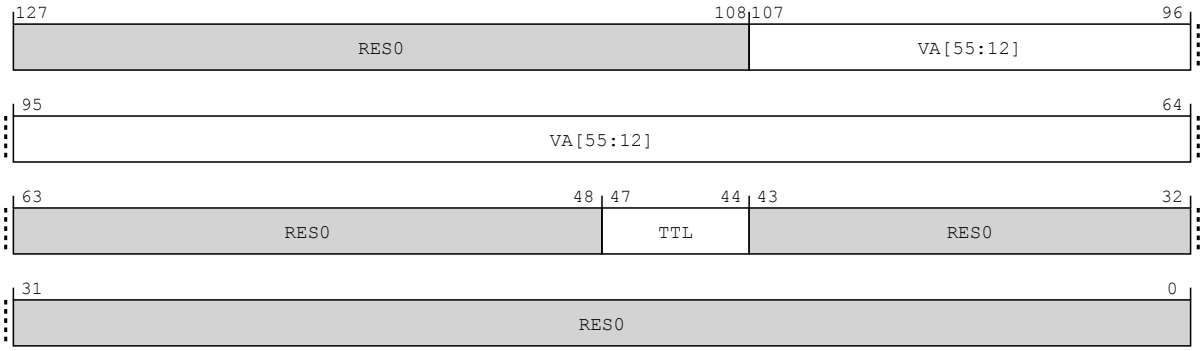

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAAE1OS, TLBIP VAAE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0001`	`0b011`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.125 TLBIP VAALE1, TLBIP VAALE1NXS, TLB Invalidate Pair by VA, All ASID, Last level, EL1

The TLBIP VAALE1, TLBIP VAALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAALE1, TLBIP VAALE1NXS are UNDEFINED.

Attributes

TLBIP VAALE1, TLBIP VAALE1NXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

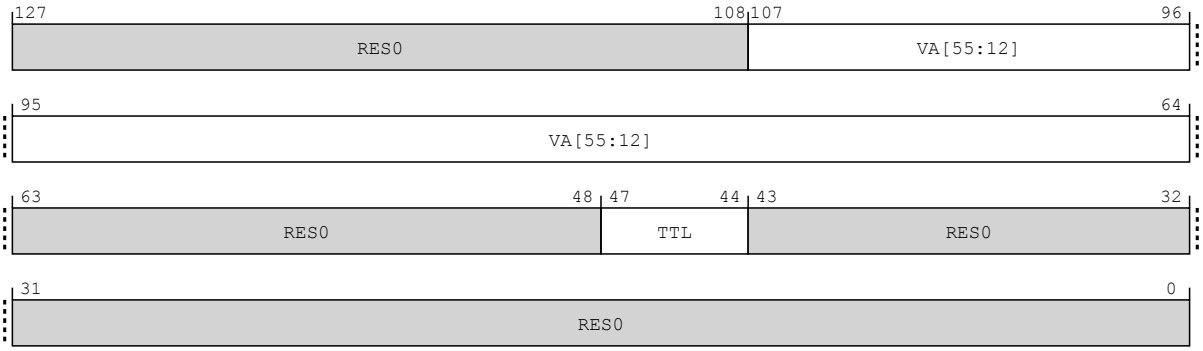

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAALE1, TLBIP VAALE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0111`	`0b111`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.126 TLBIP VAALE1IS, TLBIP VAALE1ISNXS, TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBIP VAALE1IS, TLBIP VAALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAALE1IS, TLBIP VAALE1ISNXS are UNDEFINED.

Attributes

TLBIP VAALE1IS, TLBIP VAALE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

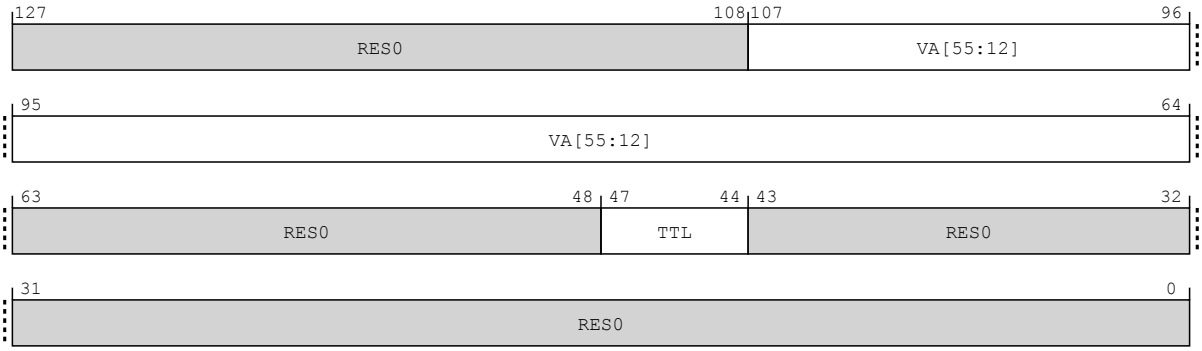

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAALE1IS, TLBIP VAALE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b111`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.127 TLBIP VAALE1OS, TLBIP VAALE1OSNXS, TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBIP VAALE1OS, TLBIP VAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAALE1OS, TLBIP VAALE1OSNXS are UNDEFINED.

Attributes

TLBIP VAALE1OS, TLBIP VAALE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

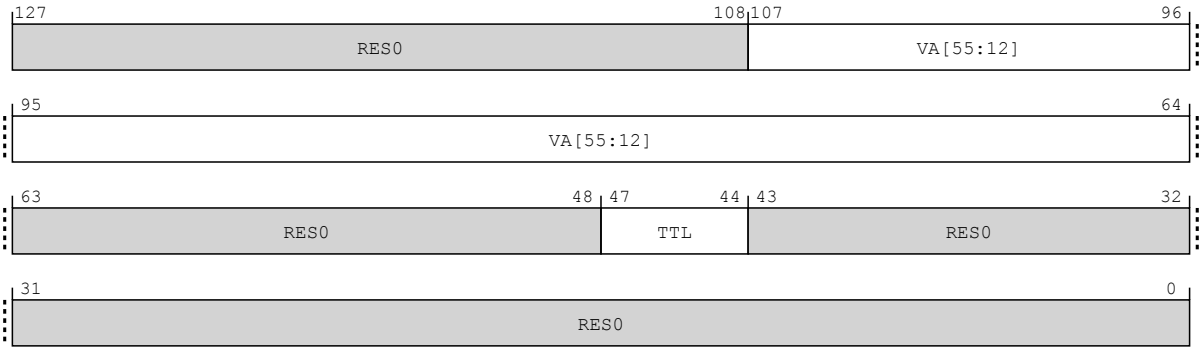

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAALE1OS, TLBIP VAALE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VAA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0001`	`0b111`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.128 TLBIP VAE1, TLBIP VAE1NXS, TLB Invalidate Pair by VA, EL1

The TLBIP VAE1, TLBIP VAE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE1, TLBIP VAE1NXS are UNDEFINED.

Attributes

TLBIP VAE1, TLBIP VAE1NXS is a 128-bit System instruction.

Field descriptions

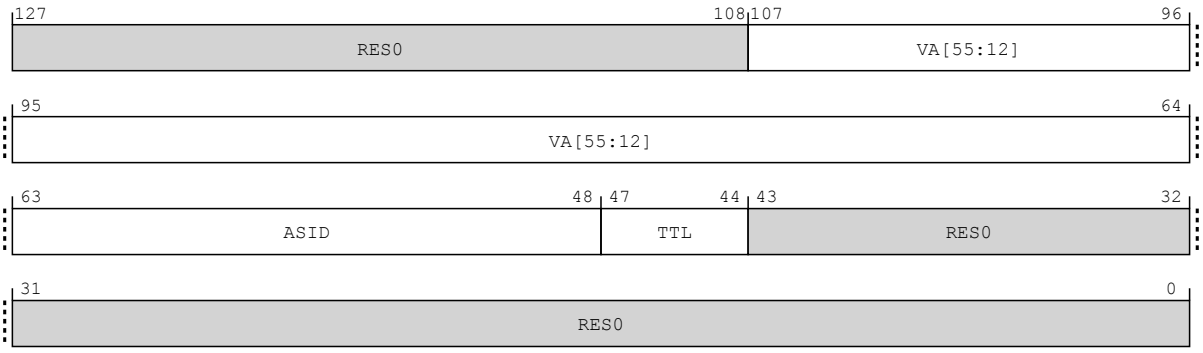

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE1, TLBIP VAE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVAE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.129 TLBIP VAE1IS, TLBIP VAE1ISNXS, TLB Invalidate Pair by VA, EL1, Inner Shareable

The TLBIP VAE1IS, TLBIP VAE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE1IS, TLBIP VAE1ISNXS are UNDEFINED.

Attributes

TLBIP VAE1IS, TLBIP VAE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

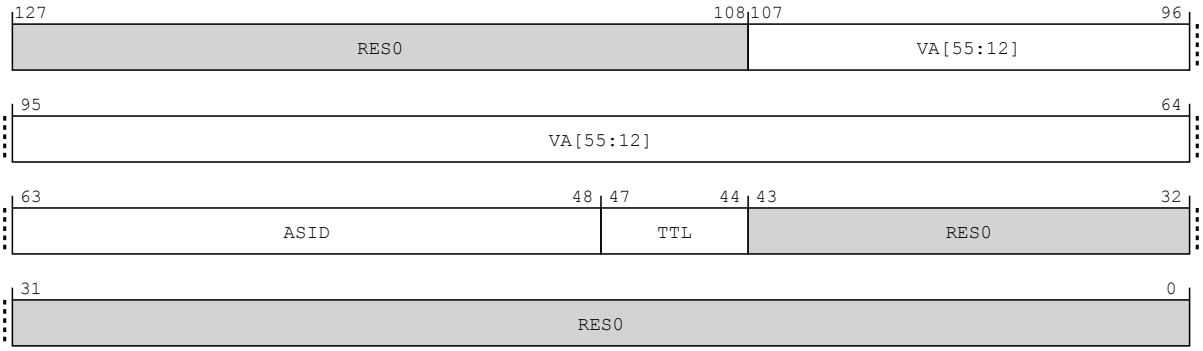

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE1IS, TLBIP VAE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b001`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
        ↪HFGITR_EL2().TLBIVAE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.130 TLBIP VAE1OS, TLBIP VAE1OSNXS, TLB Invalidate Pair by VA, EL1, Outer Shareable

The TLBIP VAE1OS, TLBIP VAE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE1OS, TLBIP VAE1OSNXS are UNDEFINED.

Attributes

TLBIP VAE1OS, TLBIP VAE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

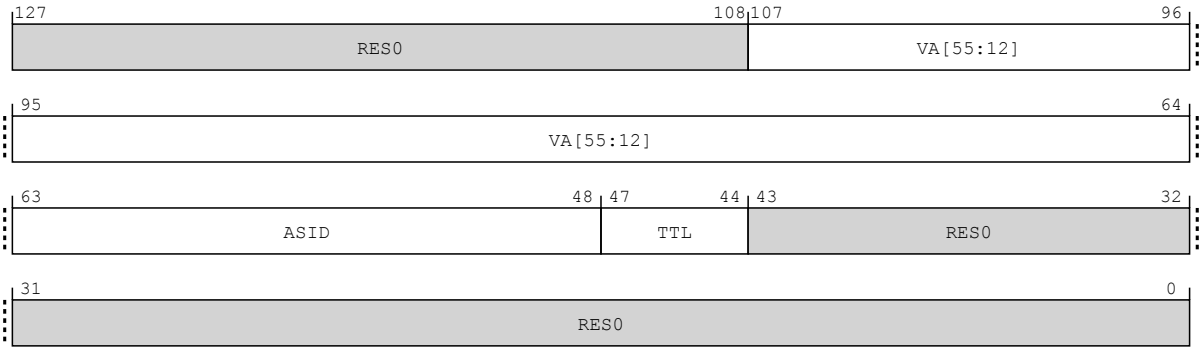

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE1OS, TLBIP VAE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0001`	`0b001`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
        ↪HFGITR_EL2().TLBIVAE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.131 TLBIP VAE2, TLBIP VAE2NXS, TLB Invalidate Pair by VA, EL2

The TLBIP VAE2, TLBIP VAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE2, TLBIP VAE2NXS are UNDEFINED.

Attributes

TLBIP VAE2, TLBIP VAE2NXS is a 128-bit System instruction.

Field descriptions

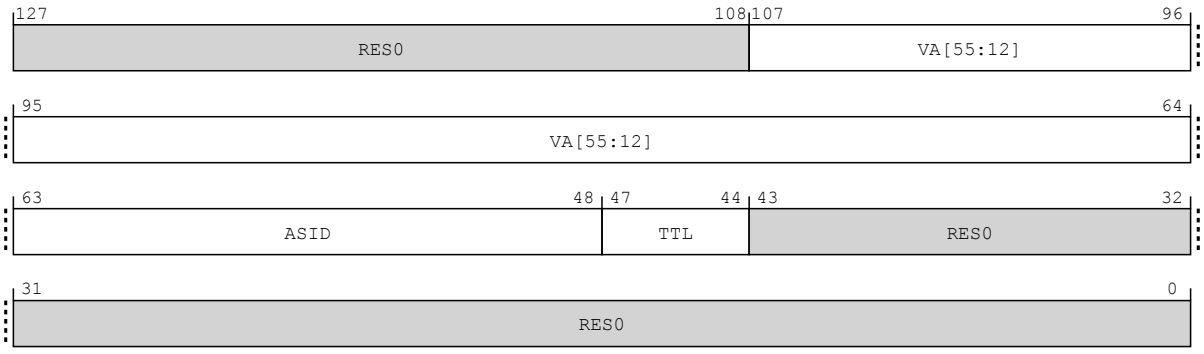

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE2, TLBIP VAE2NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP VAE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.132 TLBIP VAE2IS, TLBIP VAE2ISNXS, TLB Invalidate Pair by VA, EL2, Inner Shareable

The TLBIP VAE2IS, TLBIP VAE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE2IS, TLBIP VAE2ISNXS are UNDEFINED.

Attributes

TLBIP VAE2IS, TLBIP VAE2ISNXS is a 128-bit System instruction.

Field descriptions

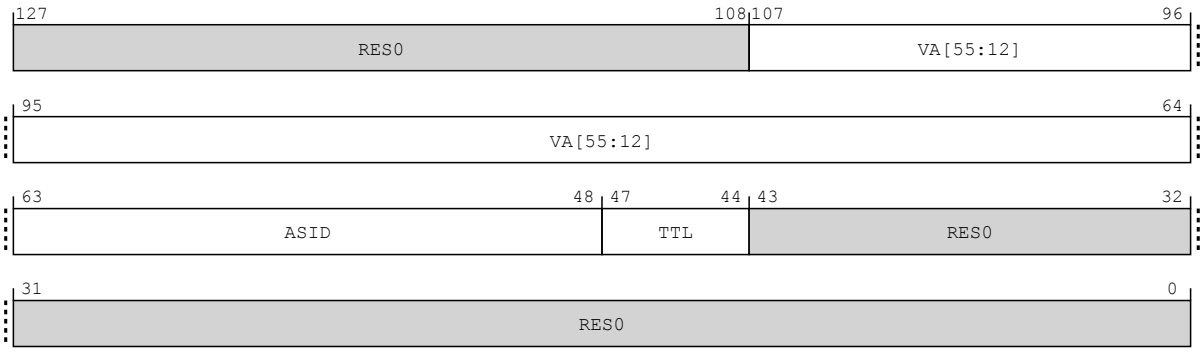

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE2IS, TLBIP VAE2ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP VAE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.133 TLBIP VAE2OS, TLBIP VAE2OSNXS, TLB Invalidate Pair by VA, EL2, Outer Shareable

The TLBIP VAE2OS, TLBIP VAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from any level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE2OS, TLBIP VAE2OSNXS are UNDEFINED.

Attributes

TLBIP VAE2OS, TLBIP VAE2OSNXS is a 128-bit System instruction.

Field descriptions

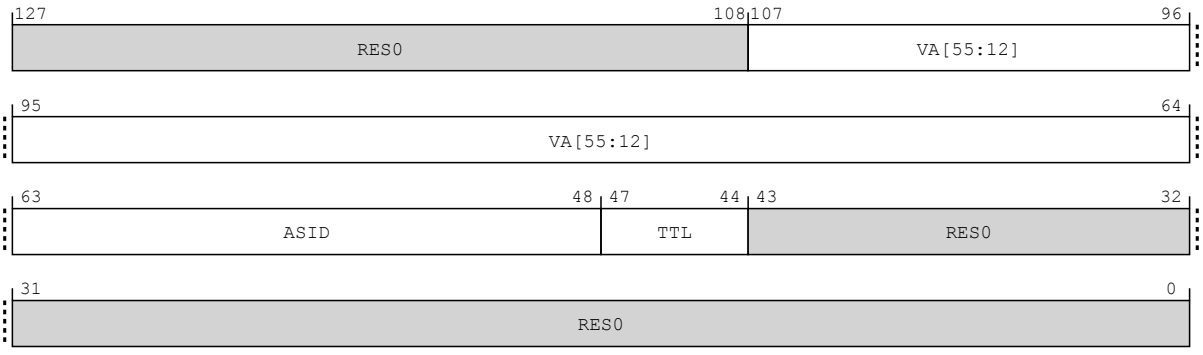

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE2OS, TLBIP VAE2OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VAE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b100`	`0b1000`	`0b0001`	`0b001`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Any, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Any, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.134 TLBIP VAE3, TLBIP VAE3NXS, TLB Invalidate Pair by VA, EL3

The TLBIP VAE3, TLBIP VAE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE3, TLBIP VAE3NXS are UNDEFINED.

Attributes

TLBIP VAE3, TLBIP VAE3NXS is a 128-bit System instruction.

Field descriptions

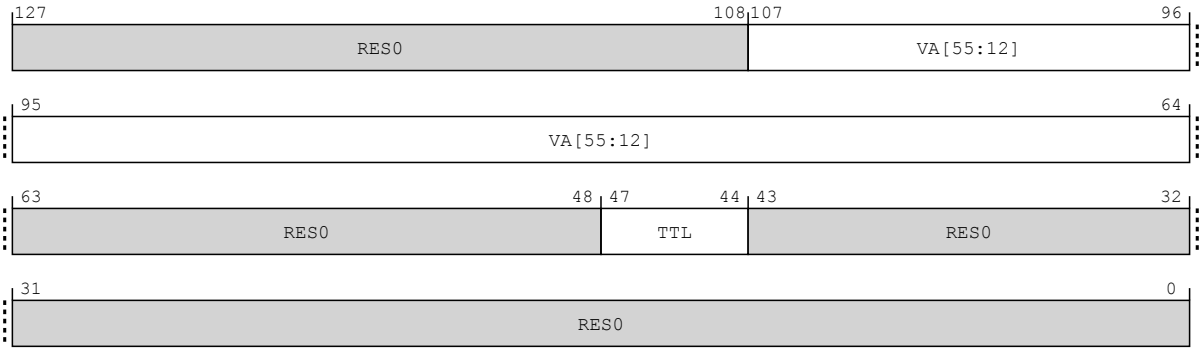

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE3, TLBIP VAE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.135 TLBIP VAE3IS, TLBIP VAE3ISNXS, TLB Invalidate Pair by VA, EL3, Inner Shareable

The TLBIP VAE3IS, TLBIP VAE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE3IS, TLBIP VAE3ISNXS are UNDEFINED.

Attributes

TLBIP VAE3IS, TLBIP VAE3ISNXS is a 128-bit System instruction.

Field descriptions

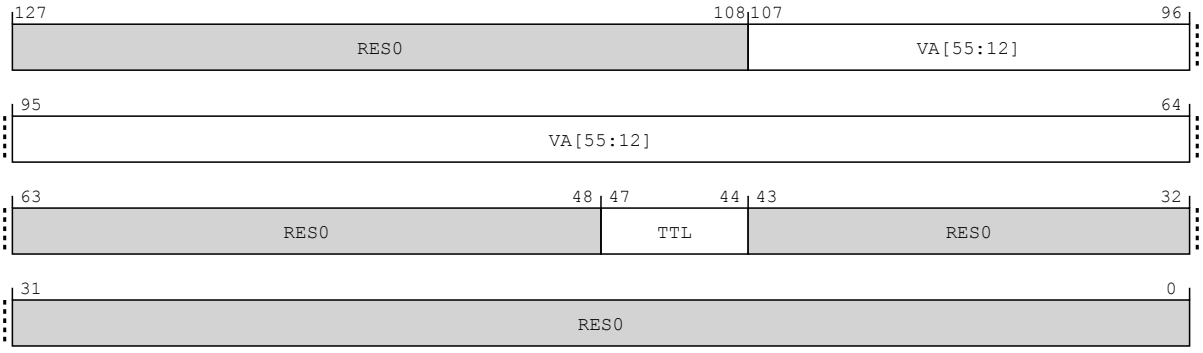

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE3IS, TLBIP VAE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.136 TLBIP VAE3OS, TLBIP VAE3OSNXS, TLB Invalidate Pair by VA, EL3, Outer Shareable

The TLBIP VAE3OS, TLBIP VAE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from any level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from any level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VAE3OS, TLBIP VAE3OSNXS are UNDEFINED.

Attributes

TLBIP VAE3OS, TLBIP VAE3OSNXS is a 128-bit System instruction.

Field descriptions

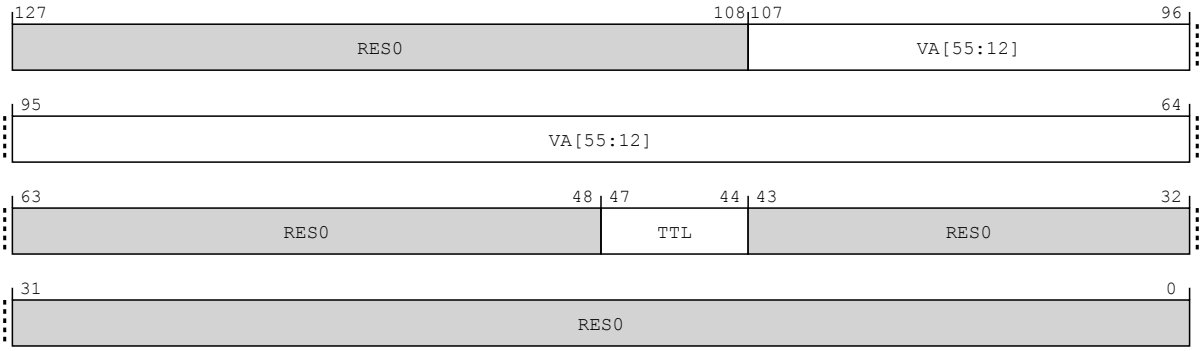

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VAE3OS, TLBIP VAE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VAE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VAE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.137 TLBIP VALE1, TLBIP VALE1NXS, TLB Invalidate Pair by VA, Last level, EL1

The TLBIP VALE1, TLBIP VALE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE1, TLBIP VALE1NXS are UNDEFINED.

Attributes

TLBIP VALE1, TLBIP VALE1NXS is a 128-bit System instruction.

Field descriptions

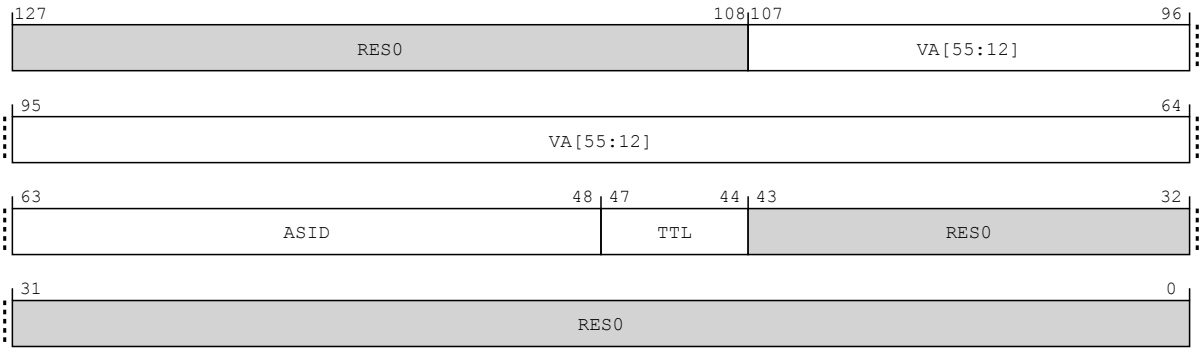

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE1, TLBIP VALE1NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE1{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VALE1NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0111	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TTLB == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
         ↪HFGITR_EL2().TLBIVALE1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
             ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.138 TLBIP VALE1IS, TLBIP VALE1ISNXS, TLB Invalidate Pair by VA, Last level, EL1, Inner Shareable

The TLBIP VALE1IS, TLBIP VALE1ISNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE1IS, TLBIP VALE1ISNXS are UNDEFINED.

Attributes

TLBIP VALE1IS, TLBIP VALE1ISNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

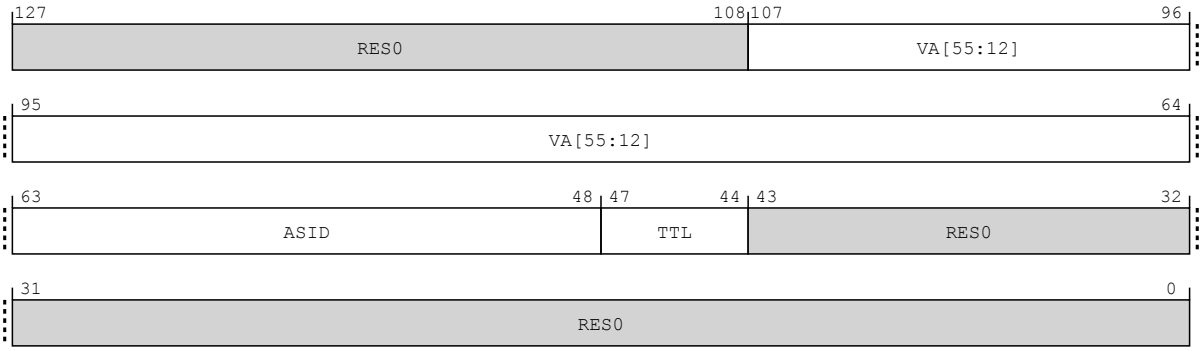

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE1IS, TLBIP VALE1ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE1IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0011`	`0b101`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VALE1ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBIS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
        ↪HFGITR_EL2().TLBIVALE1IS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.139 TLBIP VALE1OS, TLBIP VALE1OSNXS, TLB Invalidate Pair by VA, Last level, EL1, Outer Shareable

The TLBIP VALE1OS, TLBIP VALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE1OS, TLBIP VALE1OSNXS are UNDEFINED.

Attributes

TLBIP VALE1OS, TLBIP VALE1OSNXS is a 128-bit System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

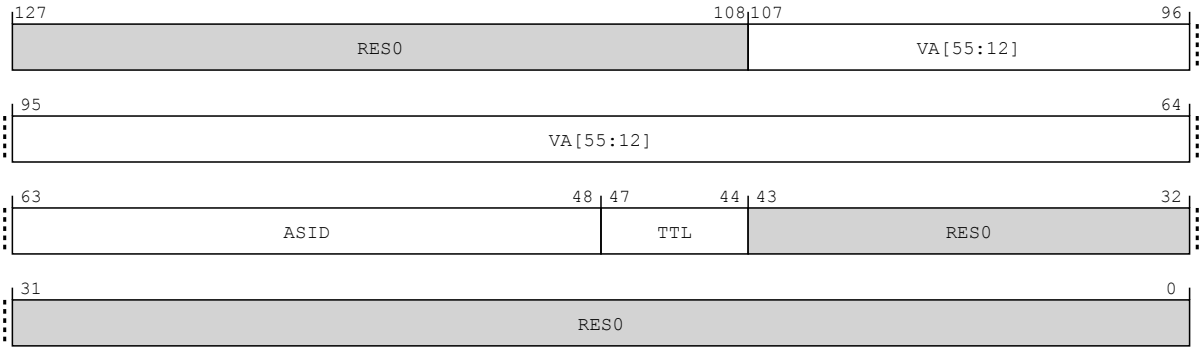

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE1OS, TLBIP VALE1OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE1OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b000`	`0b1000`	`0b0001`	`0b101`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().TLBIVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VALE1OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().TTLB == '1' || HCR_EL2().TTLBOS == '1') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2().FGTnXS == '0') &&
        ↪HFGITR_EL2().TLBIVALE1OS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL1) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.140 TLBIP VALE2, TLBIP VALE2NXS, TLB Invalidate Pair by VA, Last level, EL2

The TLBIP VALE2, TLBIP VALE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE2, TLBIP VALE2NXS are UNDEFINED.

Attributes

TLBIP VALE2, TLBIP VALE2NXS is a 128-bit System instruction.

Field descriptions

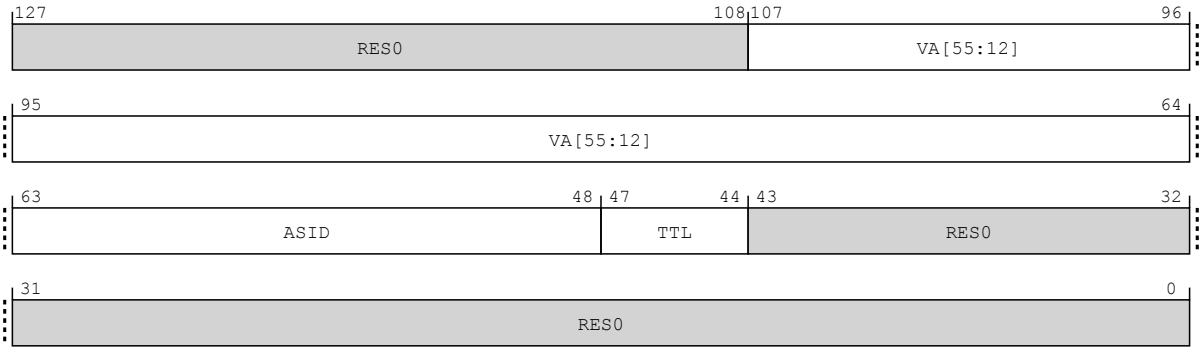

Bits [127:108]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE2, TLBIP VALE2NXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE2{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP VALE2NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0111	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.141 TLBIP VALE2IS, TLBIP VALE2ISNXS, TLB Invalidate Pair by VA, Last level, EL2, Inner Shareable

The TLBIP VALE2IS, TLBIP VALE2ISNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE2IS, TLBIP VALE2ISNXS are UNDEFINED.

Attributes

TLBIP VALE2IS, TLBIP VALE2ISNXS is a 128-bit System instruction.

Field descriptions

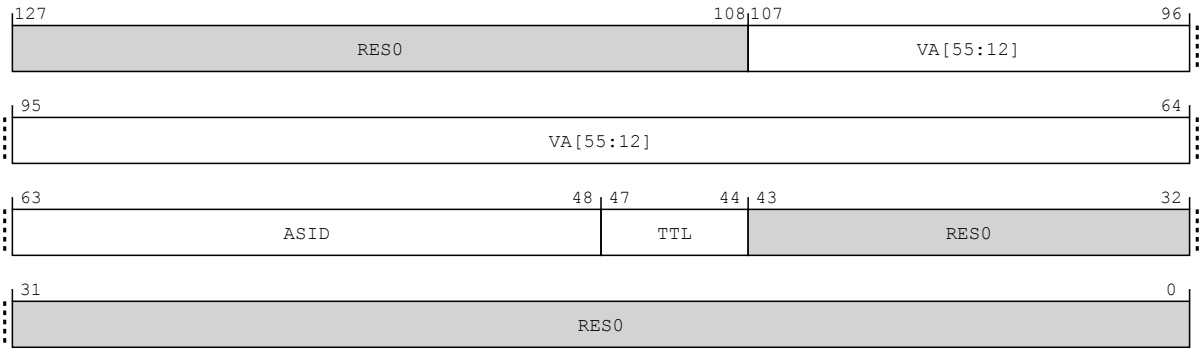

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE2IS, TLBIP VALE2ISNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE2IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```

```
TLBIP VALE2ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0011	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH,
                ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.142 TLBIP VALE2OS, TLBIP VALE2OSNXS, TLB Invalidate Pair by VA, Last level, EL2, Outer Shareable

The TLBIP VALE2OS, TLBIP VALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry.
  - A 64-bit stage 1 translation table entry, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the Effective value of HCR_EL2.E2H, for the Security state.
- If the Effective value of HCR_EL2.E2H is not 1, the entry is from the final level of the translation table walk.
- If the Effective value of HCR_EL2.E2H is 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE2OS, TLBIP VALE2OSNXS are UNDEFINED.

Attributes

TLBIP VALE2OS, TLBIP VALE2OSNXS is a 128-bit System instruction.

Field descriptions

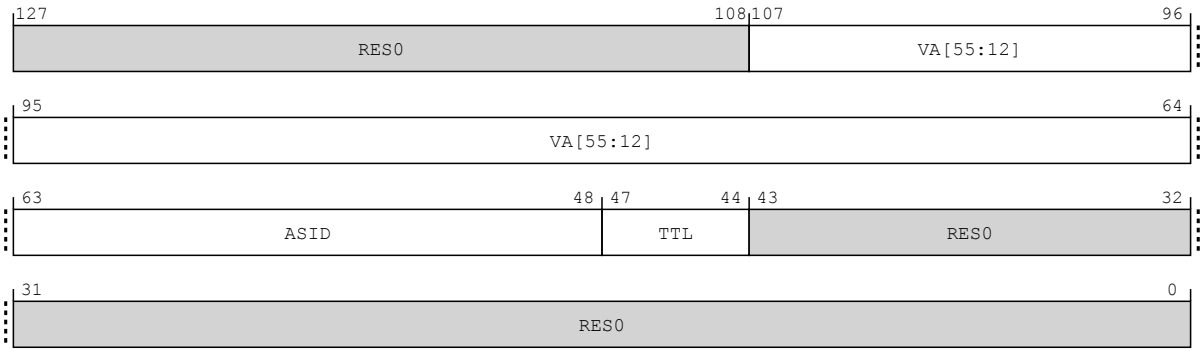

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

ASID, bits [63:48]

When ELIsInHost(EL2):

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE2OS, TLBIP VALE2OSNXS

This system instruction is an alias of the SYSP instruction.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see `AArch64_TLBIP_VA()` in the Pseudocode for AArch64 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBIP VALE2OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
`0b01`	`0b100`	`0b1000`	`0b0001`	`0b101`

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                ↪TLBILevel_Last, TLBI_AllAttr, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE2OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_ExcludeXS, X{128}(t, t2));
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        Undefined();
    else
        if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL2) then
            return;
        else
            AArch64_TLBIP_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH,
                 ↪TLBILevel_Last, TLBI_ExcludeXS, X{128}(t, t2));
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.143 TLBIP VALE3, TLBIP VALE3NXS, TLB Invalidate Pair by VA, Last level, EL3

The TLBIP VALE3, TLBIP VALE3NXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE3, TLBIP VALE3NXS are UNDEFINED.

Attributes

TLBIP VALE3, TLBIP VALE3NXS is a 128-bit System instruction.

Field descriptions

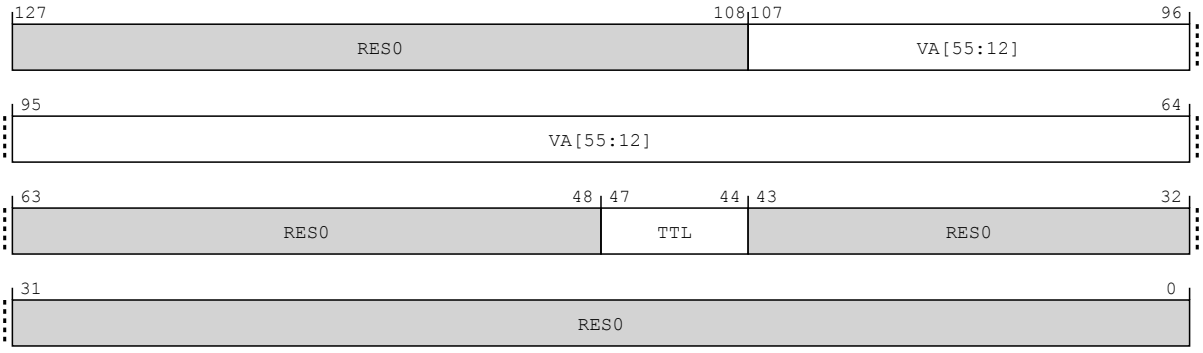

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE3, TLBIP VALE3NXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE3{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VALE3NXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.144 TLBIP VALE3IS, TLBIP VALE3ISNXS, TLB Invalidate Pair by VA, Last level, EL3, Inner Shareable

The TLBIP VALE3IS, TLBIP VALE3ISNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE3IS, TLBIP VALE3ISNXS are UNDEFINED.

Attributes

TLBIP VALE3IS, TLBIP VALE3ISNXS is a 128-bit System instruction.

Field descriptions

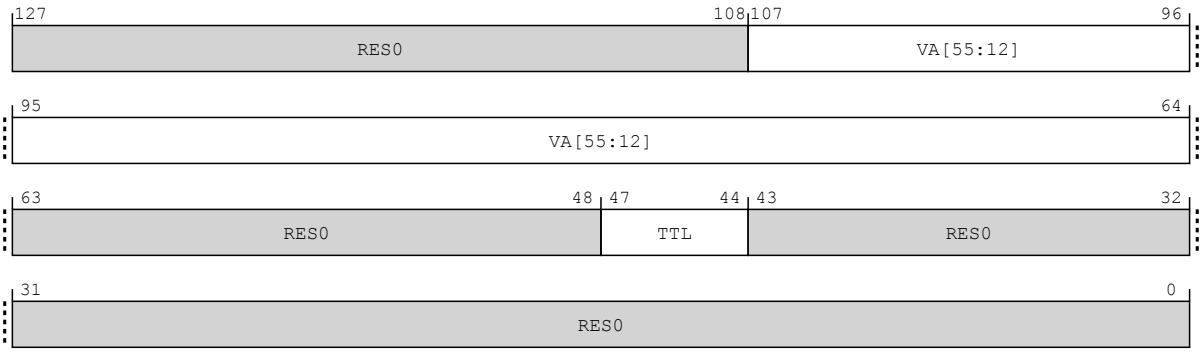

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE3IS, TLBIP VALE3ISNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE3IS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VALE3ISNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.145 TLBIP VALE3OS, TLBIP VALE3OSNXS, TLB Invalidate Pair by VA, Last level, EL3, Outer Shareable

The TLBIP VALE3OS, TLBIP VALE3OSNXS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is one of the following:
  - A 128-bit stage 1 translation table entry, from the final level of the translation table walk.
  - A 64-bit stage 1 translation table entry, from the final level of the translation table walk, if TTL[3:2] is `0b00`.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

It is IMPLEMENTATION SPECIFIC whether the TLBI System instruction with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

The TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configuration

This system instruction is present only when FEAT_D128 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TLBIP VALE3OS, TLBIP VALE3OSNXS are UNDEFINED.

Attributes

TLBIP VALE3OS, TLBIP VALE3OSNXS is a 128-bit System instruction.

Field descriptions

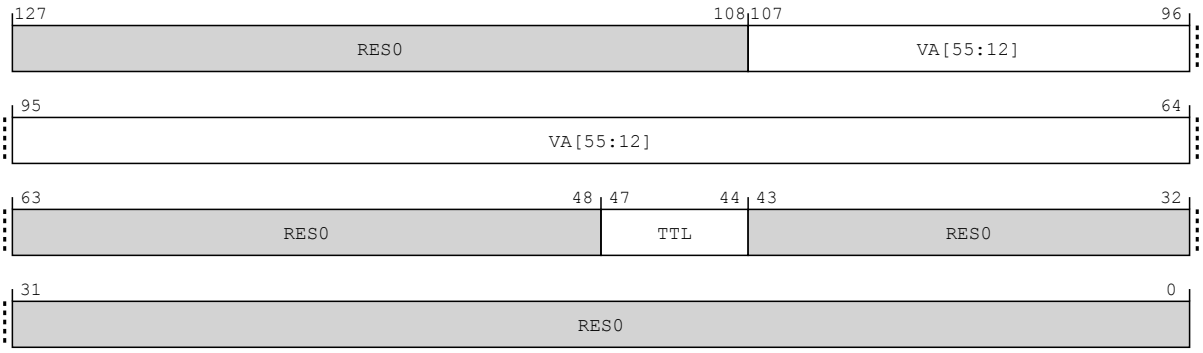

Bits [127:108]

Reserved, RES0.

VA[55:12], bits [107:64]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
`0b00xx`	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
`0b01xx`	The entry comes from a 4KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Level 0. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b10xx`	The entry comes from a 16KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.
`0b11xx`	The entry comes from a 64KB translation granule. The level of walk for the leaf level `0bxx` is encoded as: `0b00` : Reserved. Treat as if TTL<3:2> is `0b00`. `0b01` : Level 1. `0b10` : Level 2. `0b11` : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:0]

Reserved, RES0.

Executing TLBIP VALE3OS, TLBIP VALE3OSNXS

This system instruction is an alias of the SYSP instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIP VALE3OS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X{128}(t, t2));
    end;
end;
```

```
TLBIP VALE3OSNXS{, <Xt>, <Xt2>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_D128) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif !IsFeatureImplemented(FEAT_XS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RME) && !ValidSecurityStateAtEL(EL3) then
        return;
    else
        AArch64_TLBIP_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_ExcludeXS, X{128}(t, t2));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6 A64 System instructions for prediction restriction

This section lists the A64 System instructions for prediction restriction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.1 CFP RCTX, Control Flow Prediction Restriction by Context

The CFP RCTX characteristics are:

Purpose

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_SPECRES is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CFP RCTX are UNDEFINED.

Attributes

CFP RCTX is a 64-bit System instruction.

Field descriptions

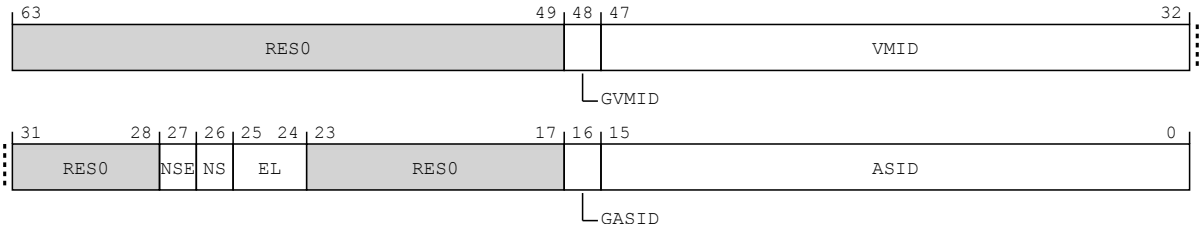

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see CFP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This instruction is treated as a NOP when executed at EL3 and either:

- CFP_RCTX.{NSE, NS} selects a reserved value.
- CFP_RCTX.{NSE, NS} == {1, 0} and CFP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing CFP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
CFP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b100

```
if !(IsFeatureImplemented(FEAT_SPECRES) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().CFPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_ControlFlow);
    end;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().CFPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_ControlFlow);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_ControlFlow);
elsif PSTATE.EL == EL3 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_ControlFlow);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.2 COSP RCTX, Clear Other Speculative Prediction Restriction by Context

The COSP RCTX characteristics are:

Purpose

Clear Other Speculative Prediction Restriction by Context applies to all prediction resources not managed by other speculation restriction System instructions.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any predictions occurring after the instruction is complete and synchronized.

This instruction applies to all speculative access except:

- Cache Prefetch predictions.
- Control Flow predictions.
- Data Value predictions.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the PE that executed the original restriction instruction, and a subsequent Context Synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations, the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.
This instruction applies to speculative accesses resulting from address predictions.

Configuration

This system instruction is present only when FEAT_SPECRES2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to COSP RCTX are UNDEFINED.

Attributes

COSP RCTX is a 64-bit System instruction.

Field descriptions

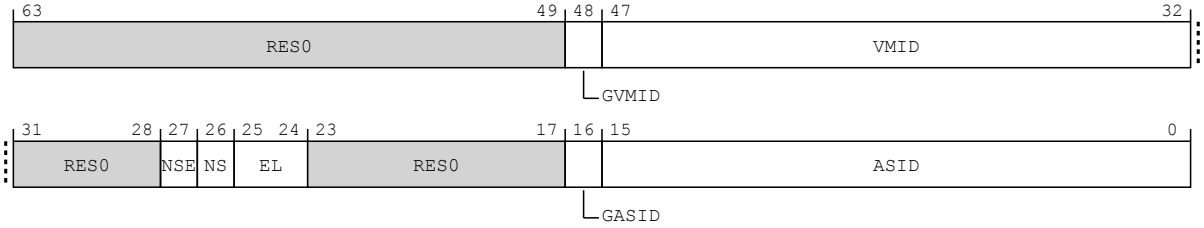

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 and EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see COSP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

This instruction is treated as a NOP when executed at EL3 and either:

- COSP_RCTX.{NSE, NS} selects a reserved value.
- COSP_RCTX.{NSE, NS} == {1, 0} and COSP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs, or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies to an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing COSP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
COSP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b110

```
if !(IsFeatureImplemented(FEAT_SPECRES2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().COSPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_Other);
    end;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().COSPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_Other);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_Other);
elsif PSTATE.EL == EL3 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_Other);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.3 CPP RCTX, Cache Prefetch Prediction Restriction by Context

The CPP RCTX characteristics are:

Purpose

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

This instruction applies to all:

- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_SPECRES is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CPP RCTX are UNDEFINED.

Attributes

CPP RCTX is a 64-bit System instruction.

Field descriptions

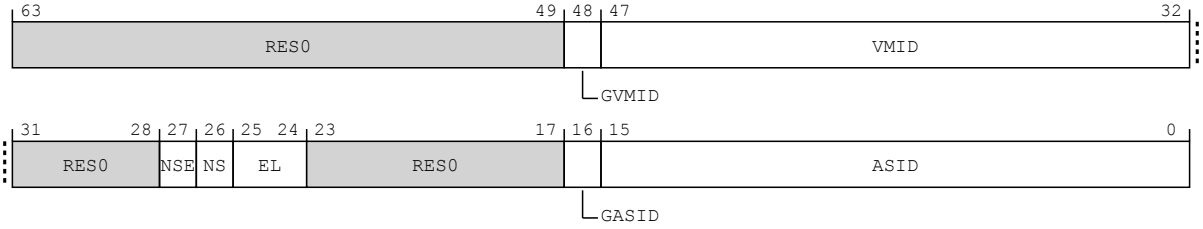

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 and EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see CPP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

This instruction is treated as a NOP when executed at EL3 and either:

- CPP_RCTX.{NSE, NS} selects a reserved value.
- CPP_RCTX.{NSE, NS} == {1, 0} and CPP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing CPP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
CPP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b111

```
if !(IsFeatureImplemented(FEAT_SPECRES) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().CPPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_CachePrefetch);
    end;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().CPPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_CachePrefetch);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_CachePrefetch);
elsif PSTATE.EL == EL3 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_CachePrefetch);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.4 DVP RCTX, Data Value Prediction Restriction by Context

The DVP RCTX characteristics are:

Purpose

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Note

The prediction of the PSTATE.{N,Z,C,V} values is not considered a data value for this purpose.

Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_SPECRES is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DVP RCTX are UNDEFINED.

Attributes

DVP RCTX is a 64-bit System instruction.

Field descriptions

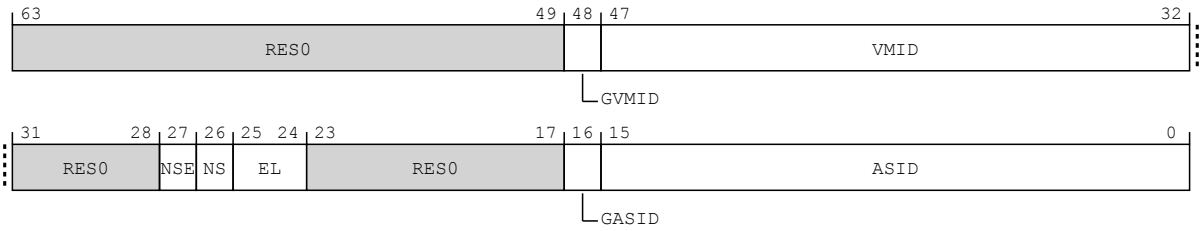

Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see DVP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

This instruction is treated as a NOP when executed at EL3 and either:

- DVP_RCTX.{NSE, NS} selects a reserved value.
- DVP_RCTX.{NSE, NS} == {1, 0} and DVP_RCTX.EL has a value other than `0b11`.

Otherwise:

Security State.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
`0b00`	EL0.	
`0b01`	EL1	
`0b10`	EL2	FEAT_EL2 is implemented
`0b11`	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
`0b0`	Applies to specified ASID for an EL0 target execution context.
`0b1`	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing DVP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DVP RCTX, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_SPECRES) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGITR_EL2().DVPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_DataValue);
    end;
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().DVPRCTX == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_RestrictPrediction(X{64}(t), RestrictType_DataValue);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_DataValue);
elsif PSTATE.EL == EL3 then
    AArch64_RestrictPrediction(X{64}(t), RestrictType_DataValue);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7 A64 System instructions for the Branch Record Buffer Extension

This section lists the A64 System instructions for the Branch Record Buffer Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7.1 BRB IALL, Invalidate the Branch Record Buffer

The BRB IALL characteristics are:

Purpose

Invalidates all Branch records in the Branch Record Buffer.

Configuration

This system instruction is present only when FEAT_BRBE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to BRB IALL are UNDEFINED.

Attributes

BRB IALL is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing BRB IALL

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
BRB IALL
```

op0	op1	CRn	CRm	op2
0b01	0b001	0b0111	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_BRBE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().nBRBIALL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRB_IALL();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRB_IALL();
    end;
elsif PSTATE.EL == EL3 then
    BRB_IALL();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7.2 BRB INJ, Branch Record Injection into the Branch Record Buffer

The BRB INJ characteristics are:

Purpose

Injects the Branch Record held in BRBINFINJ_EL1, BRBSRCINJ_EL1, and BRBTGTINJ_EL1 into the Branch Record Buffer.

Configuration

This system instruction is present only when FEAT_BRBE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to BRB INJ are UNDEFINED.

Attributes

BRB INJ is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing BRB INJ

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
BRB INJ
```

op0	op1	CRn	CRm	op2
0b01	0b001	0b0111	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_BRBE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().nBRBINJ == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRB_INJ();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRB_INJ();
    end;
elsif PSTATE.EL == EL3 then
    BRB_INJ();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.8 A64 System instructions for the Trace Extension

This section lists the A64 System instructions for the Trace Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.8.1 TRCIT, Trace Instrumentation

The TRCIT characteristics are:

Purpose

Generates an instrumentation packet in the trace.

Configuration

This system instruction is present only when FEAT_ITE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TRCIT are UNDEFINED.

Attributes

TRCIT is a 64-bit System instruction.

Field descriptions

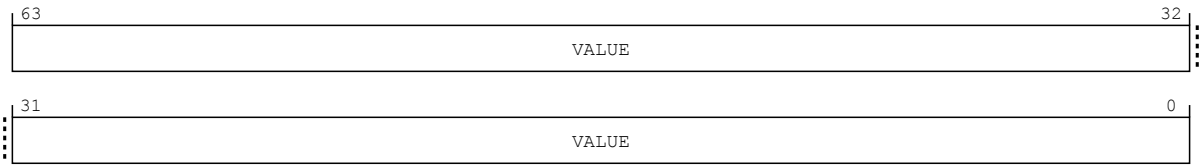

VALUE, bits [63:0]

Value to be included in the Instrumentation packet.

Executing TRCIT

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TRCIT <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    AArch64_TRCIT(X{64}(t));
elsif PSTATE.EL == EL1 then
    AArch64_TRCIT(X{64}(t));
elsif PSTATE.EL == EL2 then
    AArch64_TRCIT(X{64}(t));
elsif PSTATE.EL == EL3 then
    AArch64_TRCIT(X{64}(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9 A64 System instructions for the Guarded Control Stack

This section lists the A64 System instructions for the Guarded Control Stack.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.1 GCSPOPCX, Guarded Control Stack Pop and Compare exception return record

The GCSPOPCX characteristics are:

Purpose

Loads a GCS exception return record from the location indicated by the current GCS pointer register, compares the values loaded with the current ELR_ELx, SPSR_ELx, and LR, and increments the current GCS pointer register by the size of a GCS exception return record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPOPCX are UNDEFINED.

Attributes

GCSPOPCX is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing GCSPOPCX

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`GCSPOPCX`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0111	0b101

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().nGCSEPP == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL1) then
        GCSPOPCX();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    elsif GCSEnabled(EL2) then
        GCSPOPCX();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '1' then
        EXLOCKException();
    elsif GCSEnabled(EL3) then
        GCSPOPCX();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.2 GCSPOPM, Guarded Control Stack Pop

The GCSPOPM characteristics are:

Purpose

Loads the 64-bit doubleword that is pointed to by the current GCS pointer register, writes it to the destination register, and increments the current GCS pointer register by the size of a GCS procedure return record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPOPM are UNDEFINED.

Attributes

GCSPOPM is a 64-bit System instruction.

Field descriptions

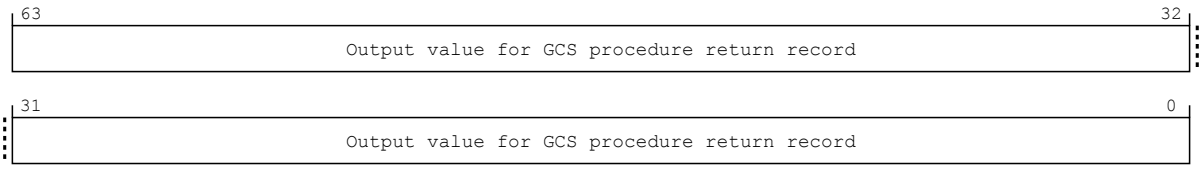

Output, bits [63:0]

Output value for GCS procedure return record.

Executing GCSPOPM

This system instruction is an alias of the SYSL instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSPOPM {<Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b001

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if GCSEnabled(EL0) then
        X{64}(t) = GCSPOPM();
    end;
elsif PSTATE.EL == EL1 then
    if GCSEnabled(EL1) then
        X{64}(t) = GCSPOPM();
    end;
elsif PSTATE.EL == EL2 then
    if GCSEnabled(EL2) then
        X{64}(t) = GCSPOPM();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if GCSEnabled(EL3) then
        X{64}(t) = GCSPOPM();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.3 GCSPOPX, Guarded Control Stack Pop exception return record

The GCSPOPX characteristics are:

Purpose

Loads a GCS exception return record from the location indicated by the current GCS pointer register, checks that the record is a GCS exception return record, and increments the current GCS pointer register by the size of a GCS exception return record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPOPX are UNDEFINED.

Attributes

GCSPOPX is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing GCSPOPX

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`GCSPOPX`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0111	0b110

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if GCSEnabled(EL1) then
        GCSPOPX();
    end;
elsif PSTATE.EL == EL2 then
    if GCSEnabled(EL2) then
        GCSPOPX();
    end;
elsif PSTATE.EL == EL3 then
    if GCSEnabled(EL3) then
        GCSPOPX();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.4 GCSPUSHM, Guarded Control Stack Push

The GCSPUSHM characteristics are:

Purpose

Decrements the current GCS pointer register by the size of a GCS procedure return record and stores an entry to the Guarded Control Stack.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPUSHM are UNDEFINED.

Attributes

GCSPUSHM is a 64-bit System instruction.

Field descriptions

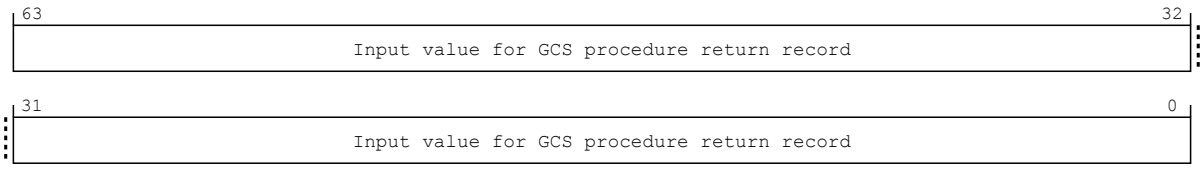

Input, bits [63:0]

Input value for GCS procedure return record.

Executing GCSPUSHM

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSPUSHM <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b000

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if (!EL2Enabled() || HCR_EL2().TGE != '1') && GCSCRE0_EL1().PUSHMEn == '0' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2().TGE == '1' && GCSCRE0_EL1().PUSHMEn == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL0) then
        GCSPUSHM(X{64}(t));
    end;
elsif PSTATE.EL == EL1 then
    if GCSCR_EL1().PUSHMEn == '0' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().nGCSPUSHM_EL1 == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL1) then
        GCSPUSHM(X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    if GCSCR_EL2().PUSHMEn == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL2) then
        GCSPUSHM(X{64}(t));
    end;
elsif PSTATE.EL == EL3 then
    if GCSCR_EL3().PUSHMEn == '0' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif GCSEnabled(EL3) then
        GCSPUSHM(X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.5 GCSPUSHX, Guarded Control Stack Push exception return record

The GCSPUSHX characteristics are:

Purpose

Decrements the current GCS pointer register by the size of a GCS exception return record and stores a GCS exception return record to the Guarded Control Stack.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSPUSHX are UNDEFINED.

Attributes

GCSPUSHX is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing GCSPUSHX

Rt should be encoded as `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`GCSPUSHX`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0111	0b100

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '0' then
        EXLOCKException();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGITR_EL2().nGCSEPP == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif GCSEnabled(EL1) then
        GCSPUSHX();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '0' then
        EXLOCKException();
    elsif GCSEnabled(EL2) then
        GCSPUSHX();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_GCS) && GetCurrentEXLOCKEN() && !Halted() && PSTATE.EXLOCK == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        EXLOCKException();
    elsif GCSEnabled(EL3) then
        GCSPUSHX();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.6 GCSSS1, Guarded Control Stack Switch Stack 1

The GCSSS1 characteristics are:

Purpose

Validates that the stack being switched to contains a Valid GCS cap record, stores an In-progress GCS cap record on to the stack that is getting switched to and sets the current GCS pointer register to the stack that is getting switched to.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSSS1 are UNDEFINED.

Attributes

GCSSS1 is a 64-bit System instruction.

Field descriptions

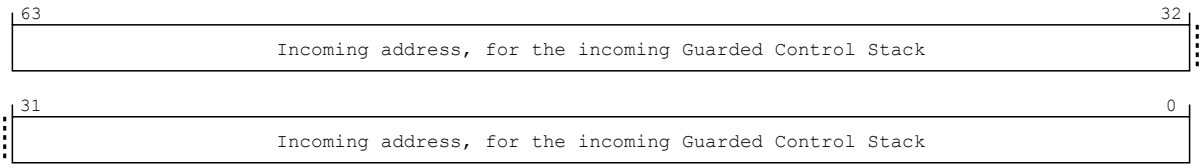

IA, bits [63:0]

Incoming address, for the incoming Guarded Control Stack.

Executing GCSSS1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSSS1 <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if GCSEnabled(EL0) then
        GCSSS1(X{64}(t));
    end;
elsif PSTATE.EL == EL1 then
    if GCSEnabled(EL1) then
        GCSSS1(X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    if GCSEnabled(EL2) then
        GCSSS1(X{64}(t));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if GCSEnabled(EL3) then
        GCSSS1(X{64}(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.9.7 GCSSS2, Guarded Control Stack Switch Stack 2

The GCSSS2 characteristics are:

Purpose

Validates that the most recent entry of the Guarded Control Stack that is getting switched to contains an In-progress GCS cap record, stores a Valid GCS cap record to the Guarded Control Stack that is getting switched from, and sets Xt to the address of that Valid GCS cap record.

Configuration

This system instruction is present only when FEAT_GCS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GCSSS2 are UNDEFINED.

Attributes

GCSSS2 is a 64-bit System instruction.

Field descriptions

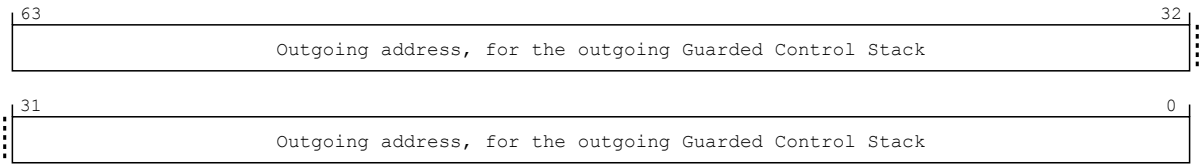

OA, bits [63:0]

Outgoing address, for the outgoing Guarded Control Stack.

Executing GCSSS2

This system instruction is an alias of the SYSL instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
GCSSS2 <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_GCS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if GCSEnabled(EL0) then
        X{64}(t) = GCSSS2();
    end;
elsif PSTATE.EL == EL1 then
    if GCSEnabled(EL1) then
        X{64}(t) = GCSSS2();
    end;
elsif PSTATE.EL == EL2 then
    if GCSEnabled(EL2) then
        X{64}(t) = GCSSS2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if GCSEnabled(EL3) then
        X{64}(t) = GCSSS2();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.10 A64 System instructions introduced by FEAT_RME_GPC3

This section lists the A64 System instructions introduced by FEAT_RME_GPC3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.10.1 APAS, Associate PA space

The APAS characteristics are:

Purpose

Associate a location with a specified PA space, for locations that are protected by a memory-side PAS filter.

Configuration

This system instruction is present only when FEAT_RME_GPC3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APAS are UNDEFINED.

Attributes

APAS is a 64-bit System instruction.

Field descriptions

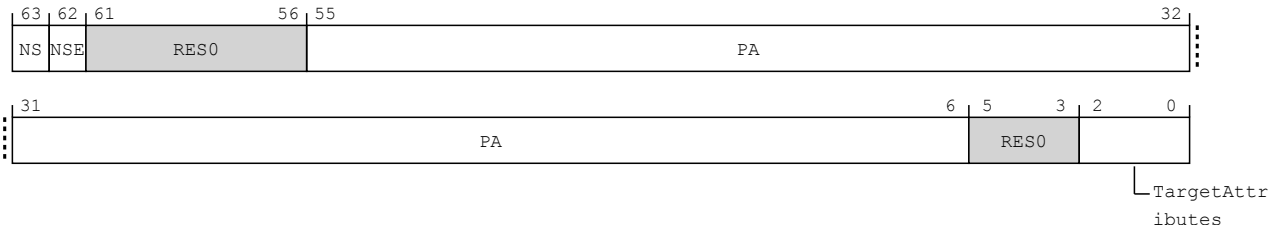

NS, bit [63]

Together with the NSE field, selects the target physical address space.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

NSE, bit [62]

Together with the NS field, selects the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see APAS.NS.

Bits [61:56]

Reserved, RES0.

PA, bits [55:6]

Bits [55:6] of the Physical Address.

Bits of Xt corresponding to physical address bits above the implemented physical address size indicated in ID_AA64MMFR0_EL1.PARange are RES0. For example, if 44 bits of PA space are implemented, then Xt[55:44] are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [5:3]

Reserved, RES0.

TargetAttributes, bits [2:0]

The Target Attributes field is encoded as follows:

TargetAttributes	Meaning
0b000	Default behavior applies.
0b001..0b111	IMPLEMENTATION DEFINED.

Executing APAS

This instruction is not subject to any translation, permission checks, or granule protection checks.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
APAS <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b0000	0b000

```
if !(IsFeatureImplemented(FEAT_RME_GPC3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    AArch64_APAS(X{64}(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.11 A64 IMPLEMENTATION DEFINED System instructions

This section lists the A64 IMPLEMENTATION DEFINED System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.11.1 SYS|SYSL|SYSP S1_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED System instructions

The SYS|SYSL|SYSP S1_<op1><*Cn*><Cm>_<op2> characteristics are:

Purpose

This area of the System instruction encoding space is reserved for IMPLEMENTATION DEFINED System instructions.

Configuration

This system instruction is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SYS|SYSL|SYSP S1_<op1>_<Cn>_<Cm>_<op2> are UNDEFINED.

Attributes

SYS|SYSL|SYSP S1_<op1><*Cn*><Cm>_<op2> is a:

- 128-bit System instruction when FEAT_SYSINSTR128 is implemented.
- 64-bit System instruction otherwise.

Field descriptions

When FEAT_SYSINSTR128 is implemented:

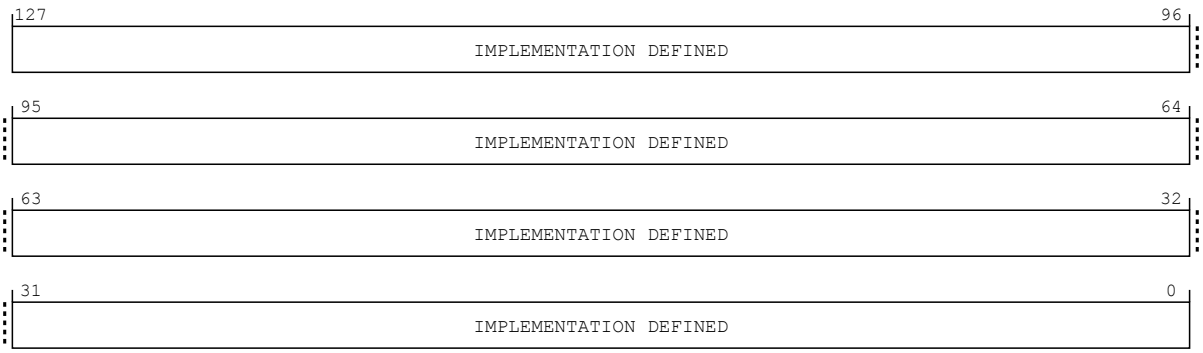

IMPLEMENTATION DEFINED, bits [127:0]

IMPLEMENTATION DEFINED.

Otherwise:

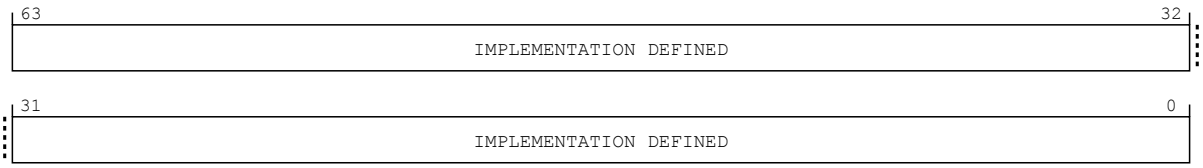

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Executing SYS|SYSL|SYSP S1_<op1>_<Cn>_<Cm>_<op2>

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL3 then
    AArch64_ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
end;
```

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    AArch64_ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL3 then
    AArch64_ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
end;
```

When FEAT_SYSINSTR128 is implemented SYSP #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>, <Xt2>}

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        else
            AArch64_SystemAccessTrap(EL1, 0x14);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        AArch64_ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
elsif PSTATE.EL == EL3 then
    AArch64_ImpDefSysInstr128(op0, op1, CRn, CRm, op2, t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C6
A64 Base Instruction Descriptions

This chapter describes the A64 base instructions.

It contains the following sections:

- About the A64 base instructions.
- Alphabetical list of A64 base instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.1 About the A64 base instructions

Alphabetical list of A64 base instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Branch, Exception generation, and System instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

A64 instruction set encoding provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The rest of this section is general description of the base instructions. It contains the following subsections:

- Register size.
- Use of the PC.
- Use of the stack pointer.
- Condition flags and related instructions.

C6.1.1 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The Condition flags, where set by the instruction, are calculated from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits calculated by the equivalent 64-bit instruction form. For example, a 32-bit bitwise `ORR` could be performed using a 64-bit `ORR` and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the `ORR` instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an `ADD`, `SUB`, `ADDS`, `or SUBS` instruction and the index register of a load/store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See Register names and Register indexed addressing.

C6.1.2 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

- Conditional and unconditional branch and return instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Exception generation and exception return instructions.

For more details of instructions that can modify the PC, see Branches, Exception generating, and System instructions.

C6.1.3 Use of the stack pointer

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned, That is, it must be aligned to 16 bytes. Misalignment generates an SP alignment fault. See SP alignment checking for more information.
- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.
- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.

C6.1.4 Condition flags and related instructions

The A64 base instructions that use the Condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.
- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.
- Conditional select with increment, negate, or invert. This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:
  - Conditional select or move. The Condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.
  - Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the Condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.
- Conditional compare. This instruction type sets the Condition flags to the result of a comparison if the original condition is true, otherwise it sets the Condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers. The conditional compare instructions are `CCMP` and `CCMN`.

The A64 base instructions that update the Condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `RMIF`, `SBCS`, `SETF8`,`SETF16`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.
- Conditional compare instructions such as `CCMN`, `CCMP`.
- The random number generation instructions `MRS RNDR` and `MRS RNDRRS`, see Effect of random number generation instructions on Condition flags.

The A64 base instructions that manipulate the Condition flags are:

- The flag manipulation instruction `CFINV`, which inverts the value of the Carry flag.
- If FEAT_FlagM2 is implemented, the base instructions `AXFLAG` and `XAFLAG`. These instructions convert between the Arm floating point comparison PSTATE condition flag format and an alternative format shown in Table C6-1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C6-1 Relationship between ARM format and alternative format PSTATE condition flags

	ARM format				Alternative format			
Result	N	Z	C	V	N	Z	C	V
Greater than	0	0	1	0	0	0	1	0
Less than	1	0	0	0	0	0	0	0
Equal	0	1	1	0	0	1	1	0
Unordered	0	0	1	1	0	1	0	0

The flags can be directly accessed for a read/write using the NZCV.

The A64 base instructions also include conditional branch instructions that do not manipulate the Condition flags:

- Compare and branch if a register is zero or nonzero, `CBZ` and `CBNZ`.
- Test a single bit in a register and branch if the bit is zero or nonzero, `TBZ` and `TBNZ`.
- Compare and branch instructions, `CB<cc>` (immediate), `CB<cc>` (register), `CBB<cc>`, and `CBH<cc>`.

C6.1.4.1 Effect of random number generation instructions on Condition flags

If FEAT_RNG is implemented, then:

- When a valid random number is returned, the PSTATE.NZCV flags are set to `0b0000`.
- If the random number hardware is not capable of returning a random number in a reasonable period of time, the PSTATE.NZCV flags are set to `0b0100`, and the random number generation instructions return the value 0.

Note

The definition of "reasonable period of time" is IMPLEMENTATION DEFINED. The expectation is that software might use this as an opportunity to reschedule or run a different routine, perhaps after a small number of retries have failed to return a valid value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2 Alphabetical list of A64 base instructions

This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see Understanding the A64 instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.1 ABS

Absolute value

This instruction calculates the absolute value of the signed integer value in the source register, and writes the result to the destination register.

Integer

(FEAT_CSSC)

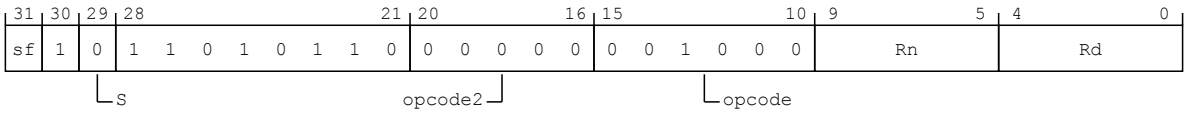

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ABS  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ABS  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 32 << UInt(sf);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = Abs(SInt(operand1));
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.2 ADC

Add with carry

This instruction adds two register values and the Carry flag value, and writes the result to the destination register.

31	30	29	28	21	20	16	15	10	9	5	4	0
sf	0	0	1 1 0 1 0 0 0 0		Rm		0 0 0 0 0 0		Rn		Rd	
	op	S										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADC  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADC  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
var result : bits(datasize);

(result, -) = AddWithCarry{datasize}(operand1, operand2, PSTATE.C);

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.3 ADCS

Add with carry, setting flags

This instruction adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	0	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0		Rn					Rd				
	op			S																												

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADCS  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADCS  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, PSTATE.C);

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.4 ADD (extended register)

Add extended and scaled register

This instruction adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword.

31	30	29	28	24	23	22	21	20	16	15	13	12	10	9	5	4	0
sf	0	0	0 1 0 1 1		0	0	1	Rm		option		imm3		Rn		Rd	

op, S, opt

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADD  <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADD  <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let shift : integer{} = UInt(imm3) as integer{0..4};
let datasize : integer{} = 32 << UInt(sf);
let extend_type : ExtendType = DecodeRegExtend(option);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTX when “option” is ‘011’.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
010	W
x11	X
10x	W
110	W

<m>

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = ExtendReg{}(m, extend_type, shift);
var result : bits(datasize);

(result, -) = AddWithCarry{datasize}(operand1, operand2, '0');

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.5 ADD (immediate)

Add immediate value

This instruction adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP).

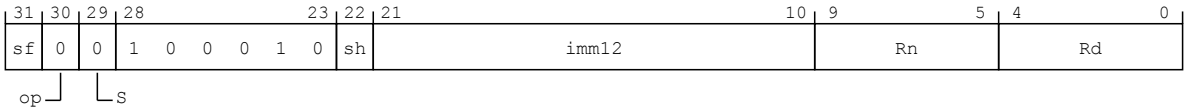

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADD  <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADD  <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);

let imm : bits(24) = if sh == '0' then Zeros{12}::imm12 else imm12::Zeros{12};
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
MOV (to/from SP)	sh == '0' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = ZeroExtend{}(imm);
var result : bits(datasize);
(result, -) = AddWithCarry{datasize}(operand1, operand2, '0');

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.6 ADD (shifted register)

Add optionally-shifted register

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	24	23 22	21	20	16	15	10	9	5	4	0
sf	0	0	0 1 0 1 1		shift	0	Rm		imm6		Rn		Rd	
	op	S												

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADD  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADD  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);
var result : bits(datasize);

(result, -) = AddWithCarry{datasize}(operand1, operand2, '0');

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.7 ADDG

Add with tag

This instruction adds an immediate value scaled by the Tag Granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer

(FEAT_MTE)

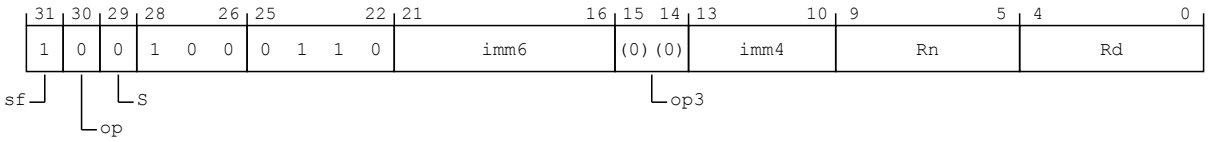

Encoding

```
ADDG  <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let tag_offset : bits(4) = imm4;
let offset : bits(64) = LSL(ZeroExtend{64}(imm6), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<uimm6>

Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the “imm6” field.

<uimm4>

Is an unsigned immediate, in the range 0 to 15, encoded in the “imm4” field.

Operation

```
let operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let start_tag : bits(4) = AArch64_AllocationTagFromAddress(operand1);
let exclude : bits(16) = GCR_EL1().Exclude;
let rtag : bits(4) = AArch64_ChooseNonExcludedTagOrZero(start_tag, tag_offset, exclude);
var result : bits(64);

(result, -) = AddWithCarry{64}(operand1, offset, '0');

result = AArch64_AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP{64}() = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    X{64}(d) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.8 ADDPT

Add checked pointer

This instruction adds a base address register value and an optionally-shifted register value, and writes the result to the destination register. The optionally-shifted register value is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

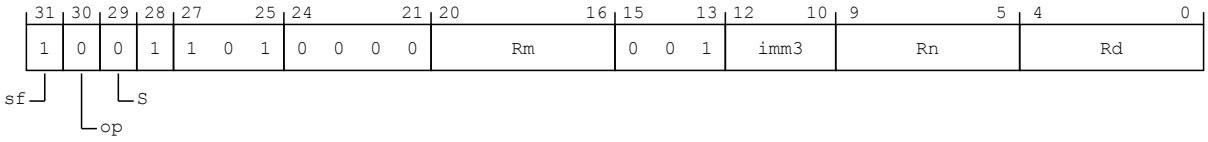

Encoding

```
ADDPT  <Xd|SP>, <Xn|SP>, <Xm>{, LSL #<amount>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let shift : integer = UInt(imm3);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the general-purpose destination register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first general-purpose source register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the left shift amount, in the range 0 to 7, defaulting to 0, encoded in the “imm3” field.

Operation

```
var result : bits(64);
let base : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let offset : bits(64) = LSL(X{64}(m), shift);

result = base + offset;
result = PointerAddCheck(result, base);

if d == 31 then
    SP{64}() = result;
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(d) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.9 ADDS (extended register)

Add extended and scaled register, setting flags

This instruction adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register).

31	30	29	28 … 24	23	22	21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
sf	0	1	0 1 0 1 1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S			opt						

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADDS  <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADDS  <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let shift : integer{} = UInt(imm3) as integer{0..4};
let datasize : integer{} = 32 << UInt(sf);
let extend_type : ExtendType = DecodeRegExtend(option);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTX when “option” is ‘011’.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m>

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMN (extended register)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = ExtendReg{}(m, extend_type, shift);
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, '0');

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.10 ADDS (immediate)

Add immediate value, setting flags

This instruction adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate).

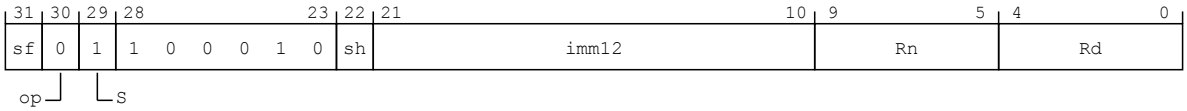

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADDS  <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADDS  <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);

let imm : bits(24) = if sh == '0' then Zeros{12}::imm12 else imm12::Zeros{12};
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
CMN (immediate)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = ZeroExtend{}(imm);
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, '0');

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.11 ADDS (shifted register)

Add optionally-shifted register, setting flags

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register).

31	30	29	28				24	23	22	21	20			16	15				10	9				5	4				0
sf	0	1	0	1	0	1	1	shift		0	Rm				imm6					Rn					Rd				

op S

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ADDS  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ADDS  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMN (shifted register)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, '0');

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.12 ADR

Form PC-relative address

This instruction adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

31	30 29	28 26	25 24	23 5	4 0
0	immlo	1 0 0	0 0	immhi	Rd
op					

Encoding

```
ADR  <Xd>, <label>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let imm : bits(64) = SignExtend{}(immhi::immlo);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<label>

Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in “immhi:immlo”.

Operation

```
X{64}(d) = PC64() + imm;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.13 ADRP

Form PC-relative address to 4KB page

This instruction adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.

31	30 29	28 26	25 24	23 5	4 0
1	immlo	1 0 0	0 0	immhi	Rd
op					

Encoding

```
ADRP  <Xd>, <label>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let imm : bits(64) = SignExtend{}(immhi::immlo::Zeros{12});
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<label>

Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as “immhi:immlo” times 4096.

Operation

```
let base : bits(64) = PC64()[63:12]::Zeros{12};
X{64}(d) = base + imm;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.14 AND (immediate)

Bitwise AND (immediate)

This instruction performs a bitwise AND between a register value and an immediate value, and writes the result to the destination register.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
AND  <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
AND  <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
var imm : bits(datasize);
(imm, -) = DecodeBitMasks{datasize}(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is the bitmask immediate. For encoding information, see Logical (immediate).

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = imm;

let result : bits(datasize) = operand1 AND operand2;

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.15 AND (shifted register)

Bitwise AND (shifted register)

This instruction performs a bitwise AND between a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 0 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
AND  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
AND  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

X{datasize}(d) = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.16 ANDS (immediate)

Bitwise AND (immediate), setting flags

This instruction performs a bitwise AND between a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate).

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 1	1 0 0 1 0 0	N	immr	imms	Rn	Rd
	opc						

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
ANDS  <Wd>, <Wn>, #<imm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ANDS  <Xd>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
var imm : bits(datasize);
(imm, -) = DecodeBitMasks{datasize}(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is the bitmask immediate. For encoding information, see Logical (immediate).

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
TST (immediate)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = imm;

let result : bits(datasize) = operand1 AND operand2;

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = result[datasize-1]::IsZeroBit{datasize}(result)::'00';
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.17 ANDS (shifted register)

Bitwise AND (shifted register), setting flags

This instruction performs a bitwise AND between a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register).

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ANDS  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ANDS  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
TST (shifted register)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

let result : bits(datasize) = operand1 AND operand2;
X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = result[datasize-1]::IsZeroBit{datasize}(result)::'00';
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.18 APAS

Associate physical address space

This instruction associates a physical address space with a memory-mapped location that is protected by a memory-side physical address space filter.

For more information, see APAS, Associate PA space.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_RME_GPC3)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	1 1 0	0 1 1 1	0 0 0 0	0 0 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
APAS  <Xt>
```

is equivalent to

```
SYS  #6, C7, C0, #0, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.19 ASR (register)

Arithmetic shift right (register)

This instruction shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of ASRV. This means:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		1	0	Rn					Rd			
		S																				op2									

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ASR  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
ASRV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ASR  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
ASRV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Operational Information

The description of ASRV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.20 ASR (immediate)

Arithmetic shift right (immediate)

This instruction shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 1 0	N	immr	x 1 1 1 1 1 (imms)	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 011111)`

```
ASR  <Wd>, <Wn>, #<shift>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1 && imms == 111111)`

```
ASR  <Xd>, <Xn>, #<shift>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, encoded in the “immr” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.21 ASRV

Arithmetic shift right variable

This instruction shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register).

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		1	0	Rn					Rd			
		S																				op2									

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ASRV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ASRV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
ASR (register)	Unconditionally

Operation

```
let operand2 : bits(datasize) = X{}(m);

X{datasize}(d) = ShiftReg{datasize}(n, shift_type, UInt(operand2) MOD datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.22 AT

Address translate

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

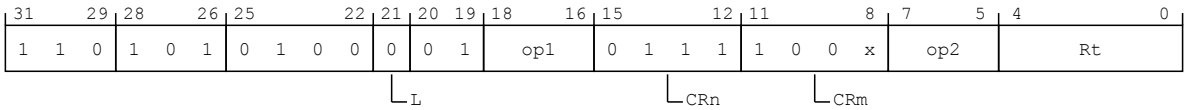

Encoding

```
AT  <at_op>, <Xt>
```

is equivalent to

```
SYS  #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_AT`.

Assembler Symbols

<at_op>

Is an AT operation name, as listed for the AT system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<at_op>	Architectural Feature	Description
000	1000	000	S1E1R	-	For more information, see AT S1E1R.
000	1000	001	S1E1W	-	For more information, see AT S1E1W.
000	1000	010	S1E0R	-	For more information, see AT S1E0R.
000	1000	011	S1E0W	-	For more information, see AT S1E0W.
000	1001	000	S1E1RP	FEAT_PAN2	For more information, see AT S1E1RP.
000	1001	001	S1E1WP	FEAT_PAN2	For more information, see AT S1E1WP.
000	1001	010	S1E1A	FEAT_ATS1A	For more information, see AT S1E1A.
100	1000	000	S1E2R	-	For more information, see AT S1E2R.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRm	op2	<at_op>	Architectural Feature	Description
100	1000	001	S1E2W	-	For more information, see AT S1E2W.
100	1000	100	S12E1R	-	For more information, see AT S12E1R.
100	1000	101	S12E1W	-	For more information, see AT S12E1W.
100	1000	110	S12E0R	-	For more information, see AT S12E0R.
100	1000	111	S12E0W	-	For more information, see AT S12E0W.
100	1001	010	S1E2A	FEAT_ATS1A	For more information, see AT S1E2A.
110	1000	000	S1E3R	-	For more information, see AT S1E3R.
110	1000	001	S1E3W	-	For more information, see AT S1E3W.
110	1001	010	S1E3A	FEAT_ATS1A	For more information, see AT S1E3A.

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.23 AUTDA, AUTDZA

Authenticate data address, using key A

This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>`, for `AUTDA`.
- The value zero, for `AUTDZA`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth)

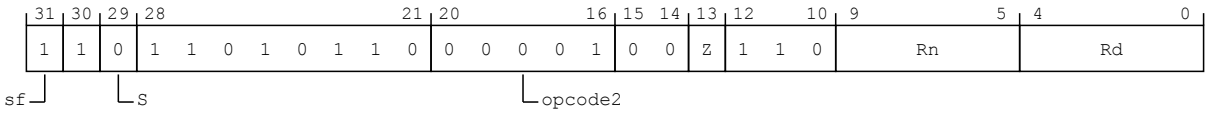

Encoding for the AUTDA variant

Applies when `(Z == 0)`

```
AUTDA  <Xd>, <Xn|SP>
```

Encoding for the AUTDZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTDZA  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF); end;
let auth_combined : boolean = FALSE;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let source_is_sp : boolean = Z == '0' && n == 31;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    X{64}(d) = AuthDA(X{64}(d), SP{64}(), auth_combined);
else
    X{64}(d) = AuthDA(X{64}(d), X{64}(n), auth_combined);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.24 AUTDB, AUTDZB

Authenticate data address, using key B

This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTDB`.
- The value zero, for `AUTDZB`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth)

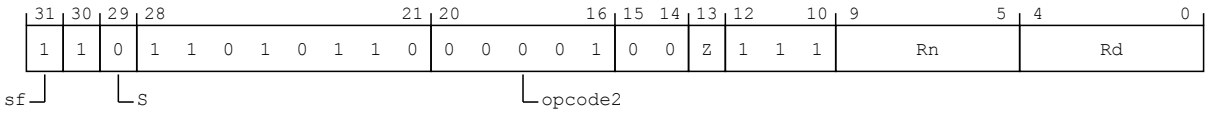

Encoding for the AUTDB variant

Applies when `(Z == 0)`

```
AUTDB  <Xd>, <Xn|SP>
```

Encoding for the AUTDZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTDZB  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF); end;
let auth_combined : boolean = FALSE;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let source_is_sp : boolean = Z == '0' && n == 31;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    X{64}(d) = AuthDB(X{64}(d), SP{64}(), auth_combined);
else
    X{64}(d) = AuthDB(X{64}(d), X{64}(n), auth_combined);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.25 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate instruction address, using key A

This instruction authenticates an instruction address, using a modifier and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `AUTIA` and `AUTIZA`.
- In X17, for `AUTIA1716`.
- In X30, for `AUTIASP` and `AUTIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTIA`.
- The value zero, for `AUTIZA` and `AUTIAZ`.
- In X16, for `AUTIA1716`.
- In SP, for `AUTIASP`.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `AUTIA1716` and `AUTIASP` include a second modifier that is:

- In X15, for `AUTIA1716`.
- In X16, for `AUTIASP`.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

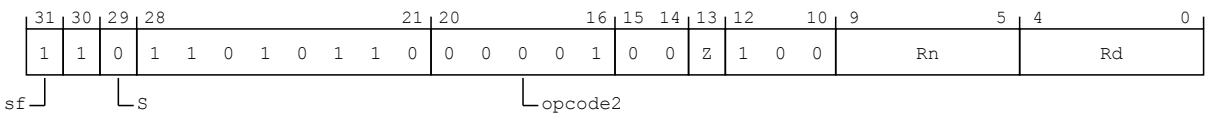

Encoding for the AUTIA variant

Applies when `(Z == 0)`

```
AUTIA  <Xd>, <Xn|SP>
```

Encoding for the AUTIZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTIZA  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF); end;
let autia1716 : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let auth_combined : boolean = FALSE;

let source_is_sp : boolean = Z == '0' && n == 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System

(FEAT_PAuth)

31																				12	11			8	7			5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0		0	0	x	1	1	0	x		1	1	1	1	1
																					CRm					op2							

Encoding for the AUTIA1716 variant

Applies when `(CRm == 0001 && op2 == 100)`

```
AUTIA1716
```

Encoding for the AUTIASP variant

Applies when `(CRm == 0011 && op2 == 101)`

```
AUTIASP
```

Encoding for the AUTIAZ variant

Applies when `(CRm == 0011 && op2 == 100)`

```
AUTIAZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP); end;
var d : integer;
var n : integer;
var source_is_sp : boolean = FALSE;
var autia1716 : boolean = FALSE;
let auth_combined : boolean = FALSE;

case CRm::op2 of
    when '0011 100' => // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101' => // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100' => // AUTIA1716
        d = 17;
        n = 16;
        autia1716 = TRUE;
end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X{64}(d) = AuthIA2(X{64}(d), SP{64}(), X{64}(16), auth_combined);
    else
        X{64}(d) = AuthIA(X{64}(d), SP{64}(), auth_combined);
    end;
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && autia1716 then
        X{64}(d) = AuthIA2(X{64}(d), X{64}(n), X{64}(15), auth_combined);
    else
        X{64}(d) = AuthIA(X{64}(d), X{64}(n), auth_combined);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.26 AUTIA171615

Authenticate instruction address, using key A

This instruction authenticates an instruction address, using two modifiers and key A. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth_LR)

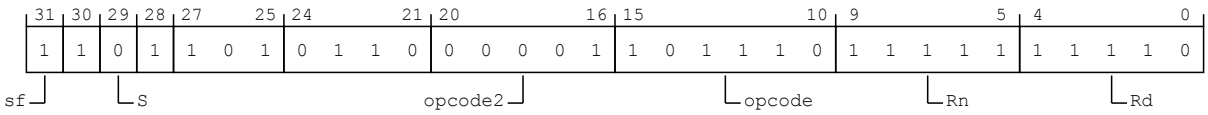

Encoding

```
AUTIA171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
let is_combined : boolean = FALSE;
X{64}(17) = AuthIA2(X{64}(17), X{64}(16), X{64}(15), is_combined);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.27 AUTIASPPC

Authenticate return address using key A, using an immediate offset

This instruction authenticates an instruction address, using two modifiers and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is the address of a program label.

Integer

(FEAT_PAuth_LR)

31	30 29	28 26	25 23	22 21	20 5	4 0
1	1 1	1 0 0	1 1 1	0 0	imm16	1 1 1 1 1
sf				opc		Rd

Encoding

```
AUTIASPPC  <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
let offset : bits(64) = ZeroExtend{}(imm16::'00');
let auth_combined : boolean = FALSE;
```

Assembler Symbols

<label>

Is the program label whose address is to be calculated. Its negative offset from the address of this instruction, a multiple of 4 in the range -262140 to 0, is encoded as an unsigned value in the “imm16” field as <label>/4.

Operation

```
let pac_addr : bits(64) = PC64() - offset;

X{64}(d) = AuthIA2(X{64}(d), SP{64}(), pac_addr, auth_combined);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.28 AUTIASPPCR

Authenticate return address using key A, using a register

This instruction authenticates an instruction address, using two modifiers and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is in the general-purpose register that is specified by `<Xn>`.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	1	0	0	Rn					1	1	1	1	0
sf		S									opcode2					opcode											Rd				

Encoding

```
AUTIASPPCR  <Xn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
let n : integer = UInt(Rn);
let auth_combined : boolean = FALSE;
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
X{64}(d) = AuthIA2(X{64}(d), SP{64}(), X{64}(n), auth_combined);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.29 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate instruction address, using key B

This instruction authenticates an instruction address, using a modifier and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `AUTIB` and `AUTIZB`.
- In X17, for `AUTIB1716`.
- In X30, for `AUTIBSP` and `AUTIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `AUTIB`.
- The value zero, for `AUTIZB` and `AUTIBZ`.
- In X16, for `AUTIB1716`.
- In SP, for `AUTIBSP`.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `AUTIB1716` and `AUTIBSP` include a second modifier that is:

- In X15, for `AUTIB1716`.
- In X16, for `AUTIBSP`.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

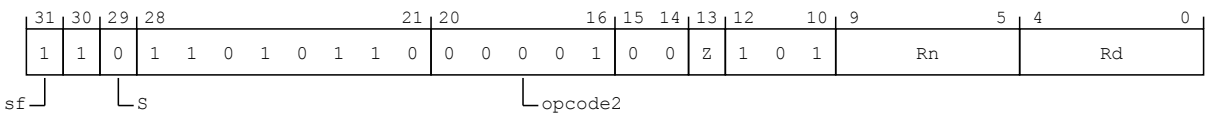

Encoding for the AUTIB variant

Applies when `(Z == 0)`

```
AUTIB  <Xd>, <Xn|SP>
```

Encoding for the AUTIZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
AUTIZB  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF); end;
let autib1716 : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let auth_combined : boolean = FALSE;

let source_is_sp : boolean = Z == '0' && n == 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System

(FEAT_PAuth)

31																				12	11			8	7			5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0		0	0	x	1	1	1	x		1	1	1	1	1
																					CRm						op2						

Encoding for the AUTIB1716 variant

Applies when `(CRm == 0001 && op2 == 110)`

```
AUTIB1716
```

Encoding for the AUTIBSP variant

Applies when `(CRm == 0011 && op2 == 111)`

```
AUTIBSP
```

Encoding for the AUTIBZ variant

Applies when `(CRm == 0011 && op2 == 110)`

```
AUTIBZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP); end;
var d : integer;
var n : integer;
var source_is_sp : boolean = FALSE;
var autib1716 : boolean = FALSE;
let auth_combined : boolean = FALSE;

case CRm::op2 of
    when '0011 110' => // AUTIBZ
        d = 30;
        n = 31;
    when '0011 111' => // AUTIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 110' => // AUTIB1716
        d = 17;
        n = 16;
        autib1716 = TRUE;
end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X{64}(d) = AuthIB2(X{64}(d), SP{64}(), X{64}(16), auth_combined);
    else
        X{64}(d) = AuthIB(X{64}(d), SP{64}(), auth_combined);
    end;
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && autib1716 then
        X{64}(d) = AuthIB2(X{64}(d), X{64}(n), X{64}(15), auth_combined);
    else
        X{64}(d) = AuthIB(X{64}(d), X{64}(n), auth_combined);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.30 AUTIB171615

Authenticate instruction address, using key B

This instruction authenticates an instruction address, using two modifiers and key B. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer

(FEAT_PAuth_LR)

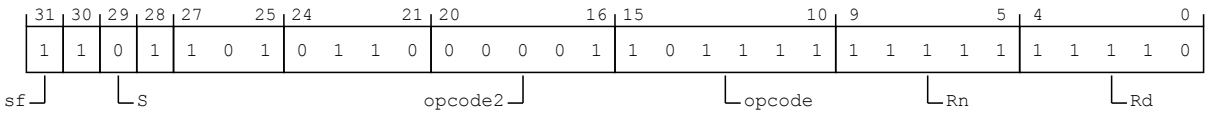

Encoding

```
AUTIB171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
let is_combined : boolean = FALSE;
X{64}(17) = AuthIB2(X{64}(17), X{64}(16), X{64}(15), is_combined);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.31 AUTIBSPPC

Authenticate return address using key B, using an immediate offset

This instruction authenticates an instruction address, using two modifiers and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is the address of a program label.

Integer

(FEAT_PAuth_LR)

31	30 29	28 26	25 23	22 21	20 5	4 0
1	1 1	1 0 0	1 1 1	0 1	imm16	1 1 1 1 1
sf				opc		Rd

Encoding

```
AUTIBSPPC  <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
let offset : bits(64) = ZeroExtend{}(imm16::'00');
let auth_combined : boolean = FALSE;
```

Assembler Symbols

<label>

Is the program label whose address is to be calculated. Its negative offset from the address of this instruction, a multiple of 4 in the range -262140 to 0, is encoded as an unsigned value in the “imm16” field as <label>/4.

Operation

```
let pac_addr : bits(64) = PC64() - offset;

X{64}(d) = AuthIB2(X{64}(d), SP{64}(), pac_addr, auth_combined);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.32 AUTIBSPPCR

Authenticate return address using key B, using a register

This instruction authenticates an instruction address, using two modifiers and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is in X30.

The first modifier is in SP.

The second modifier is in the general-purpose register that is specified by `<Xn>`.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	1	0	1	Rn					1	1	1	1	0
sf		S									opcode2					opcode											Rd				

Encoding

```
AUTIBSPPCR  <Xn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
let n : integer = UInt(Rn);
let auth_combined : boolean = FALSE;
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
X{64}(d) = AuthIB2(X{64}(d), SP{64}(), X{64}(n), auth_combined);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.33 AXFLAG

Convert floating-point condition flags from Arm to external format

This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

System

(FEAT_FlagM2)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 0 0	0 1 0 0	(0) (0) (0) (0)	0 1 0	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
AXFLAG
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM2) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
let z : bit = PSTATE.Z OR PSTATE.V;
let c : bit = PSTATE.C AND NOT(PSTATE.V);

PSTATE.[N,Z,C,V] = '0' :: z :: c :: '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.34 B.cond

Branch conditionally

This instruction branches conditionally to a label at a PC-relative offset. This instruction provides a hint that this is not a subroutine call or return.

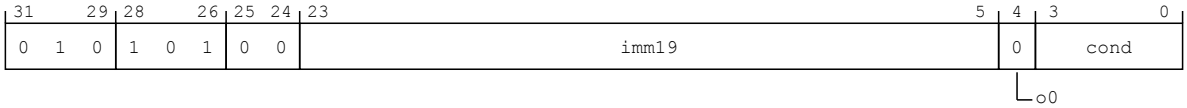

Encoding

```
B.<cond>  <label>
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm19::'00');
let condition : bits(4) = cond;
```

Assembler Symbols

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let branch_conditional : boolean = TRUE;
if ConditionHolds(condition) then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.35 B

Branch

This instruction branches unconditionally to a label at a PC-relative offset. This instruction provides a hint that this is not a subroutine call or return.

31	30	29	28		26	25 … 0
0	0	0	1	0	1	imm26
op						

Encoding

```
B  <label>
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm26::'00');
```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26” times 4.

Operation

```
let branch_conditional : boolean = FALSE;
BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.36 BC.cond

Branch consistent conditionally

This instruction branches conditionally to a label at a PC-relative offset, with a hint that this branch is very unlikely to change direction. This instruction provides a hint that this is not a subroutine call or return.

19-bit signed PC-relative branch offset

(FEAT_HBC)

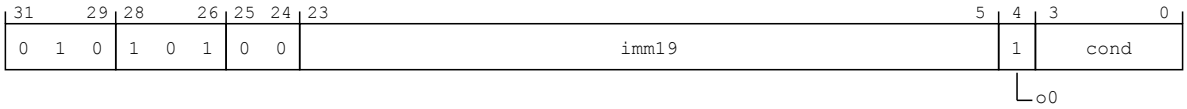

Encoding

```
BC.<cond>  <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_HBC) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm19::'00');
let condition : bits(4) = cond;
```

Assembler Symbols

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
let branch_conditional : boolean = TRUE;
if ConditionHolds(condition) then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.37 BFC

Bitfield clear

This instruction sets a bitfield of `<width>` bits at bit position `<lsb>` of the destination register to zero, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Leaving other bits unchanged

(FEAT_ASMv8p2)

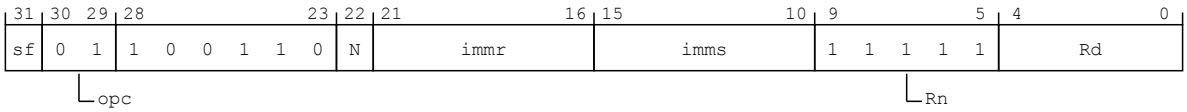

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFC  <Wd>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, WZR, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFC  <Xd>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, XZR, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.38 BFI

Bitfield insert

This instruction copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	!=11111 (Rn)	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFI  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFI  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.39 BFM

Bitfield move

This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the other bits of the destination register remain unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFM  <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFM  <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && (N != '0' || immr[5] != '0' || imms[5] != '0') then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let s : integer = UInt(imms);
let r : integer = UInt(immr);

var wmask : bits(datasize);
var tmask : bits(datasize);
(wmask, tmask) = DecodeBitMasks{datasize}(N, imms, immr, FALSE);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<immr>

For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imms>

For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
BFC	`Rn == '11111' && UInt(imms) < UInt(immr)`
BFI	`Rn != '11111' && UInt(imms) < UInt(immr)`
BFXIL	`UInt(imms) >= UInt(immr)`

Operation

```
let dst : bits(datasize) = X{}(d);
let src : bits(datasize) = X{}(n);

// Perform bitfield move on low bits
let bot : bits(datasize) = (dst AND NOT(wmask)) OR (ROR(src, r) AND wmask);

// Combine extension bits and result bits
X{datasize}(d) = (dst AND NOT(tmask)) OR (bot AND tmask);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.40 BFXIL

Bitfield extract and insert at low end

This instruction copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 1 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
BFXIL  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
BFXIL  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.41 BIC (shifted register)

Bitwise bit clear (shifted register)

This instruction performs a bitwise AND between a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 0	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
BIC  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
BIC  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

X{datasize}(d) = operand1 AND NOT(operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.42 BICS (shifted register)

Bitwise bit clear (shifted register), setting flags

This instruction performs a bitwise AND between a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1 (opc)	0 1 0 1 0	shift	1 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
BICS  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
BICS  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

let result : bits(datasize) = operand1 AND NOT(operand2);
X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = result[datasize-1]::IsZeroBit{datasize}(result)::'00';
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.43 BL

Branch with link

This instruction branches to a PC-relative offset, setting register X30 to PC+4. This instruction provides a hint that this is a subroutine call.

31	30 29	28 26	25 0
1	0 0	1 0 1	imm26
op			

Encoding

```
BL  <label>
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm26::'00');
let d : integer = 30;
```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26” times 4.

Operation

```
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64() + 4);
end;
X{64}(d) = PC64() + 4;

let branch_conditional : boolean = FALSE;
BranchTo{64}(PC64() + offset, BranchType_DIRCALL, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.44 BLR

Branch with link to register

This instruction calls a subroutine at an address in a register, setting register X30 to PC+4. This instruction provides a hint that this is a subroutine call.

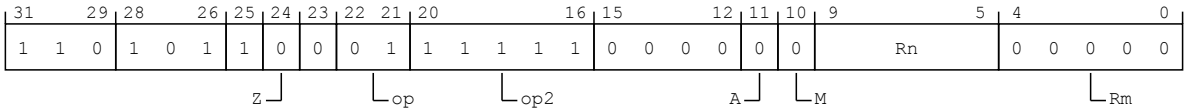

Encoding

```
BLR  <Xn>
```

Decode for this encoding

```
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

Operation

```
let target : bits(64) = X{}(n);
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64() + 4);
end;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';

X{64}(30) = PC64() + 4;

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_INDCALL, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.45 BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with link to register, with pointer authentication

This instruction authenticates the address in the general-purpose register that is specified by `<Xn>`, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xm|SP>`, for `BLRAA` and `BLRAB`.
- The value zero, for `BLRAAZ` and `BLRABZ`.

Key A is used for `BLRAA` and `BLRAAZ`. Key B is used for `BLRAB` and `BLRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the general-purpose register.

This instruction provides a hint that this is a subroutine call.

Integer

(FEAT_PAuth)

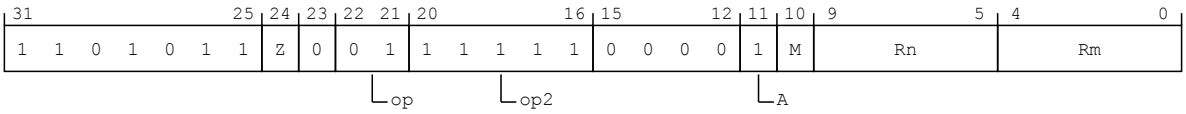

Encoding for the Key A, register modifier variant

Applies when `(Z == 1 && M == 0)`

```
BLRAA  <Xn>, <Xm|SP>
```

Encoding for the Key A, zero modifier variant

Applies when `(Z == 0 && M == 0 && Rm == 11111)`

```
BLRAAZ  <Xn>
```

Encoding for the Key B, register modifier variant

Applies when `(Z == 1 && M == 1)`

```
BLRAB  <Xn>, <Xm|SP>
```

Encoding for the Key B, zero modifier variant

Applies when `(Z == 0 && M == 1 && Rm == 11111)`

```
BLRABZ  <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
if Z == '0' && Rm != '11111' then EndOfDecode(Decode_UNDEF); end;

let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let use_key_a : boolean = (M == '0');
let source_is_sp : boolean = ((Z == '1') && (m == 31));
let auth_then_branch : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the “Rm” field.

Operation

```
var target : bits(64) = X{}(n);
let modifier : bits(64) = if source_is_sp then SP{64}() else X{64}(m);

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);
end;
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64() + 4);
end;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';

X{64}(30) = PC64() + 4;

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_INDCALL, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.46 BR

Branch to register

This instruction branches unconditionally to an address in a register. This instruction provides a hint that this is not a subroutine call or return.

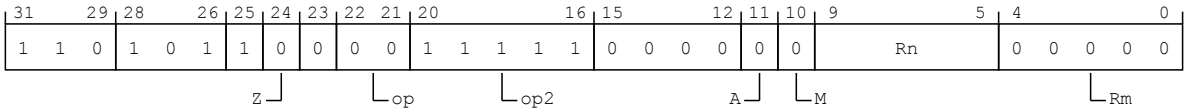

Encoding

```
BR  <Xn>
```

Decode for this encoding

```
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

Operation

```
let target : bits(64) = X{}(n);
// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
    end;
else
    BTypeNext = '01';
end;

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_INDIR, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.47 BRAA, BRAAZ, BRAB, BRABZ

Branch to register, with pointer authentication

This instruction authenticates the address in the general-purpose register that is specified by `<Xn>`, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xm|SP>`, for `BRAA` and `BRAB`.
- The value zero, for `BRAAZ` and `BRABZ`.

Key A is used for `BRAA` and `BRAAZ`. Key B is used for `BRAB` and `BRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the general-purpose register.

This instruction provides a hint that this is not a subroutine call or return.

Integer

(FEAT_PAuth)

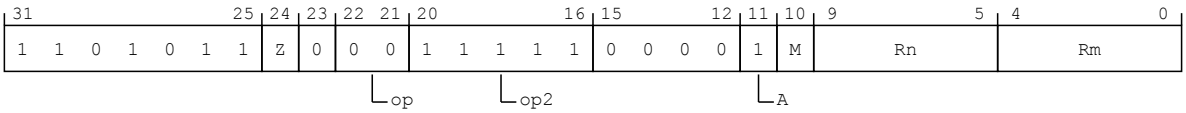

Encoding for the Key A, register modifier variant

Applies when `(Z == 1 && M == 0)`

```
BRAA  <Xn>, <Xm|SP>
```

Encoding for the Key A, zero modifier variant

Applies when `(Z == 0 && M == 0 && Rm == 11111)`

```
BRAAZ  <Xn>
```

Encoding for the Key B, register modifier variant

Applies when `(Z == 1 && M == 1)`

```
BRAB  <Xn>, <Xm|SP>
```

Encoding for the Key B, zero modifier variant

Applies when `(Z == 0 && M == 1 && Rm == 11111)`

```
BRABZ  <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
if Z == '0' && Rm != '11111' then EndOfDecode(Decode_UNDEF); end;

let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let use_key_a : boolean = (M == '0');
let source_is_sp : boolean = ((Z == '1') && (m == 31));
let auth_then_branch : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the “Rm” field.

Operation

```
var target : bits(64) = X{}(n);
let modifier : bits(64) = if source_is_sp then SP{64}() else X{64}(m);

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);
end;
// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
    end;
else
    BTypeNext = '01';
end;

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_INDIR, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.48 BRB

Branch record buffer

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_BRBE)

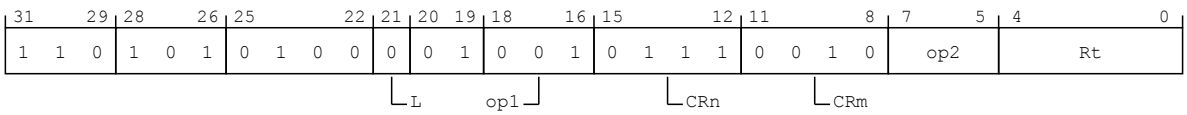

Encoding

```
BRB  <brb_op>
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp('001', '0111', '0010', op2) == Sys_BRB`.

Assembler Symbols

<brb_op>

Is a BRB operation name, as listed for the BRB system instruction group, encoded in “op2”:

op2	<brb_op>	Architectural Feature	Description
100	IALL	FEAT_BRBE	For more information, see BRB IALL.
101	INJ	FEAT_BRBE	For more information, see BRB INJ.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.49 BRK

Breakpoint instruction

This instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value `0x3c`, and captures the value of the immediate argument in ESR_ELx.ISS.

Within a guarded memory region, while PSTATE.BTYPE != `0b00`, a `BRK` instruction will not generate a Branch Target exception and will generate a Breakpoint Instruction exception as normal. For more information, see PSTATE.BTYPE.

31–29	28–26	25–24	23–21	20–5	4–2	1–0
1 1 0	1 0 1	0 0	0 0 1	imm16	0 0 0	0 0
			opc		op2	LL

Encoding

```
BRK  #<imm>
```

Decode for this encoding

```
let comment : bits(16) = imm16;
if IsFeatureImplemented(FEAT_BTI) then
    SetBTypeCompatible(TRUE);
end;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
AArch64_SoftwareBreakpoint(comment);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.50 BTI

Branch target identification

This instruction is used to guard against the execution of instructions that are not the intended target of a branch.

Outside of a guarded memory region, a `BTI` instruction executes as a `NOP`. Within a guarded memory region, while PSTATE.BTYPE != `0b00`, a `BTI` instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region. For more information, see PSTATE.BTYPE.

The operand `<targets>` passed to a `BTI` instruction determines the values of PSTATE.BTYPE that the `BTI` instruction is compatible with.

System

(FEAT_BTI)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 0 (CRm)	x x 0 (op2)	1 1 1 1 1

Encoding

```
BTI  {<targets>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BTI) then EndOfDecode(Decode_NOP); end;

// Check branch target compatibility between BTI instruction and PSTATE.BTYPE
SetBTypeCompatible(BTypeCompatible_BTI(op2[2:1]));
```

Assembler Symbols

<targets>

Is the type of indirection, encoded in “op2<2:1>”:

op2<2:1>	<targets>
00	[absent]
01	c
10	j
11	jc

Operation

```
SetBTypeNext('00');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.51 CAS, CASA, CASAL, CASL

Compare and swap word or doubleword in memory

This instruction reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of `WZR` or `XZR`, `CASA` and `CASAL` load from memory with acquire semantics.
  - `CASL` and `CASAL` store to memory with release semantics.
  - `CAS` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Ws>`, or `<Xs>`, is restored to the value held in the register before the instruction was executed.

For a `CAS` or `CASA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CAS`, `CASA`, `CASAL`, or `CASL` access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent `CAS`, `CASA`, `CASAL`, or `CASL` access to succeed when it does occur.

A code sequence starting with a `CAS` or `CASA` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CAS`, `CASA`, `CASAL`, or `CASL` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CAS` or `CASA` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CAS` or `CASA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CAS` or `CASA` either fails its compare or writes the same value back to memory.
For performance reasons, Arm recommends that software not use a `CAS` or `CASA` instruction when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` in a routine that includes a subsequent `CAS`, `CASA`, `CASAL`, or `CASL` for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent `CAS`, `CASA`, `CASAL`, or `CASL`.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction that conditionally determines whether or not to execute the subsequent `CAS`, `CASA`, `CASAL`, or `CASL`.

This guidance does not apply when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` for a `CAS` or `CASA` instruction and the compare between the `CAS` or `CASA` and the subsequent `CAS`, `CASA`, `CASAL`, or `CASL` is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent `CAS`, `CASA`, `CASAL`, or `CASL` from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LSE)

31	30	29						23	22	21	20				16	15	14				10	9			5	4			0
1	x	0	0	1	0	0	0	1	L	1	Rs					o0	1	1	1	1	1	Rn				Rt			

size

Rt2

Encoding for the 32-bit CAS variant

Applies when `(size == 10 && L == 0 && o0 == 0)`

```
CAS  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASA variant

Applies when `(size == 10 && L == 1 && o0 == 0)`

```
CASA  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASAL variant

Applies when `(size == 10 && L == 1 && o0 == 1)`

```
CASAL  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASL variant

Applies when `(size == 10 && L == 0 && o0 == 1)`

```
CASL  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CAS variant

Applies when `(size == 11 && L == 0 && o0 == 0)`

```
CAS  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASA variant

Applies when `(size == 11 && L == 1 && o0 == 0)`

```
CASA  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASAL variant

Applies when `(size == 11 && L == 1 && o0 == 1)`

```
CASAL  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASL variant

Applies when `(size == 11 && L == 0 && o0 == 1)`

```
CASL  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;
let acquire : boolean = L == '1' && s != 31;
let release : boolean = o0 == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

Operation

```
var address : bits(64);
var comparevalue : bits(datasize);
var newvalue : bits(datasize);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                       tagchecked, privileged, t, s);
comparevalue = X{datasize}(s);
newvalue = X{datasize}(t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(datasize) = MemAtomic{}(address, comparevalue, newvalue, accdesc);
X{regsize}(s) = ZeroExtend{regsize}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.52 CASB, CASAB, CASALB, CASLB

Compare and swap byte in memory

This instruction reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of `WZR` or `XZR`, `CASAB` and `CASALB` load from memory with acquire semantics.
  - `CASLB` and `CASALB` store to memory with release semantics.
  - `CASB` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Ws>`, is restored to the values held in the register before the instruction was executed.

For a `CASB` or `CASAB` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` access to succeed when it does occur.

A code sequence starting with a `CASB` or `CASAB` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASB` or `CASAB` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASB` or `CASAB` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASB` or `CASAB` either fails its compare or writes the same value back to memory.
For performance reasons, Arm recommends that software not use a `CASB` or `CASAB` instruction when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` in a routine that includes a subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB`.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction that conditionally determines whether or not to execute the subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB`.

This guidance does not apply when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` for a `CASB` or `CASAB` instruction and the compare between the `CASB` or `CASAB` and the subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent `CASB`, `CASAB`, `CASALB`, or `CASLB` from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LSE)

31 30	29 … 23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 0 (size)	0 0 1 0 0 0 1	L	1	Rs	o0	1 1 1 1 1 (Rt2)	Rn	Rt

Encoding for the CASB variant

Applies when `(L == 0 && o0 == 0)`

```
CASB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAB variant

Applies when `(L == 1 && o0 == 0)`

```
CASAB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASALB variant

Applies when `(L == 1 && o0 == 1)`

```
CASALB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLB variant

Applies when `(L == 0 && o0 == 1)`

```
CASLB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = L == '1' && s != 31;
let release : boolean = o0 == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
var comparevalue : bits(8);
var newvalue : bits(8);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                       tagchecked, privileged, t, s);
comparevalue = X{8}(s);
newvalue = X{8}(t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(8) = MemAtomic{}(address, comparevalue, newvalue, accdesc);
X{32}(s) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.53 CASH, CASAH, CASALH, CASLH

Compare and swap halfword in memory

This instruction reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of `WZR` or `XZR`, `CASAH` and `CASALH` load from memory with acquire semantics.
  - `CASLH` and `CASALH` store to memory with release semantics.
  - `CASH` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Ws>`, is restored to the values held in the register before the instruction was executed.

For a `CASH` or `CASAH` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` access to succeed when it does occur.

A code sequence starting with a `CASH` or `CASAH` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASH` or `CASAH` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASH` or `CASAH` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASH` or `CASAH` either fails its compare or writes the same value back to memory.
For performance reasons, Arm recommends that software not use a `CASH` or `CASAH` instruction when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` in a routine that includes a subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH`.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction that conditionally determines whether or not to execute the subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH`.

This guidance does not apply when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` for a `CASH` or `CASAH` instruction and the compare between the `CASH` or `CASAH` and the subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent `CASH`, `CASAH`, `CASALH`, or `CASLH` from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LSE)

31 30	29 … 23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 1 (size)	0 0 1 0 0 0 1	L	1	Rs	o0	1 1 1 1 1 (Rt2)	Rn	Rt

Encoding for the CASH variant

Applies when `(L == 0 && o0 == 0)`

```
CASH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAH variant

Applies when `(L == 1 && o0 == 0)`

```
CASAH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASALH variant

Applies when `(L == 1 && o0 == 1)`

```
CASALH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLH variant

Applies when `(L == 0 && o0 == 1)`

```
CASLH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = L == '1' && s != 31;
let release : boolean = o0 == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
var comparevalue : bits(16);
var newvalue : bits(16);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                       tagchecked, privileged, t, s);
comparevalue = X{16}(s);
newvalue = X{16}(t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(16) = MemAtomic{}(address, comparevalue, newvalue, accdesc);
X{32}(s) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.54 CASP, CASPA, CASPAL, CASPL

Compare and swap pair of words or doublewords in memory

This instruction reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPA` and `CASPAL` load from memory with acquire semantics.
  - `CASPL` and `CASPAL` store to memory with release semantics.
  - `CASP` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is `<Ws>` and `<W(s+1)>`, or `<Xs>` and `<X(s+1)>`, are restored to the values held in the registers before the instruction was executed.

For a `CASP` or `CASPA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` access to succeed when it does occur.

A code sequence starting with a `CASP` or `CASPA` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASP` or `CASPA` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASP` or `CASPA` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASP` or `CASPA` either fails its compare or writes the same value back to memory.
For performance reasons, Arm recommends that software not use a `CASP` or `CASPA` instruction when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` in a routine that includes a subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL`.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction that conditionally determines whether or not to execute the subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL`.

This guidance does not apply when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` for a `CASP` or `CASPA` instruction and the compare between the `CASP` or `CASPA` and the subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent `CASP`, `CASPA`, `CASPAL`, or `CASPL` from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LSE)

31	30	29						23	22	21	20			16	15	14				10	9			5	4			0
0	sz	0	0	1	0	0	0	0	L	1	Rs				o0	1	1	1	1	1	Rn				Rt			
																Rt2												

Encoding for the 32-bit CASP variant

Applies when `(sz == 0 && L == 0 && o0 == 0)`

```
CASP  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPA variant

Applies when `(sz == 0 && L == 1 && o0 == 0)`

```
CASPA  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPAL variant

Applies when `(sz == 0 && L == 1 && o0 == 1)`

```
CASPAL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPL variant

Applies when `(sz == 0 && L == 0 && o0 == 1)`

```
CASPL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASP variant

Applies when `(sz == 1 && L == 0 && o0 == 0)`

```
CASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPA variant

Applies when `(sz == 1 && L == 1 && o0 == 0)`

```
CASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPAL variant

Applies when `(sz == 1 && L == 1 && o0 == 1)`

```
CASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPL variant

Applies when `(sz == 1 && L == 0 && o0 == 1)`

```
CASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
if Rs[0] == '1' || Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sz);
let acquire : boolean = L == '1';
let release : boolean = o0 == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Ws> must be an even-numbered register.

<W(s+1)>

Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Wt>

Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Wt> must be an even-numbered register.

<W(t+1)>

Is the 32-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

Operation

```
var address : bits(64);
var comparevalue : bits(2*datasize);
var newvalue : bits(2*datasize);
var data : bits(2*datasize);

let s1 : bits(datasize) = X{}(s);
let s2 : bits(datasize) = X{}(s+1);
let t1 : bits(datasize) = X{}(t);
let t2 : bits(datasize) = X{}(t+1);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                       tagchecked, privileged, t, t+1, s, s+1);

comparevalue = if BigEndian(accdesc.acctype) then s1::s2 else s2::s1;
newvalue     = if BigEndian(accdesc.acctype) then t1::t2 else t2::t1;
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

data = MemAtomic{2*datasize}(address, comparevalue, newvalue, accdesc);

if BigEndian(accdesc.acctype) then
    X{datasize}(s)   = data[2*datasize-1:datasize];
    X{datasize}(s+1) = data[datasize-1:0];
else
    X{datasize}(s)   = data[datasize-1:0];
    X{datasize}(s+1) = data[2*datasize-1:datasize];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.55 CASPT, CASPAT, CASPALT, CASPLT

Compare and swap pair unprivileged

This instruction reads a pair of 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPAT` and `CASPALT` load from memory with acquire semantics.
  - `CASPLT` and `CASPALT` store to memory with release semantics.
  - `CASPT` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is `<Xs>` and `<X(s+1)>`, are restored to the values held in the registers before the instruction was executed.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For a `CASPT` or `CASPAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` access to succeed when it does occur.

A code sequence starting with a `CASPT` or `CASPAT` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CASPT` or `CASPAT` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CASPT` or `CASPAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CASPT` or `CASPAT` either fails its compare or writes the same value back to memory.
For performance reasons, Arm recommends that software not use a `CASPT` or `CASPAT` instruction when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` in a routine that includes a subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT`.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction that conditionally determines whether or not to execute the subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT`.

This guidance does not apply when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` for a `CASPT` or `CASPAT` instruction and the compare between the `CASPT` or `CASPAT` and the subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent `CASPT`, `CASPAT`, `CASPALT`, or `CASPLT` from executing under normal operating conditions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

31	30	29							23	22	21	20				16	15	14				10	9				5	4				0
0	1	0	0	1	0	0	1	1		L	0		Rs				o0	1	1	1	1	1		Rn					Rt		
	sz																				Rt2										

Encoding for the CASPT variant

Applies when `(L == 0 && o0 == 0)`

```
CASPT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the CASPAT variant

Applies when `(L == 1 && o0 == 0)`

```
CASPAT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the CASPALT variant

Applies when `(L == 1 && o0 == 1)`

```
CASPALT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the CASPLT variant

Applies when `(L == 0 && o0 == 1)`

```
CASPLT  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
if Rs[0] == '1' || Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 64;
let acquire : boolean = L == '1';
let release : boolean = o0 == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var comparevalue : bits(2*datasize);
var newvalue : bits(2*datasize);
var data : bits(2*datasize);

let s1 : bits(datasize) = X{}(s);
let s2 : bits(datasize) = X{}(s+1);
let t1 : bits(datasize) = X{}(t);
let t2 : bits(datasize) = X{}(t+1);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                       tagchecked, privileged, t, t+1, s, s+1);

comparevalue = if BigEndian(accdesc.acctype) then s1::s2 else s2::s1;
newvalue     = if BigEndian(accdesc.acctype) then t1::t2 else t2::t1;
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

data = MemAtomic{2*datasize}(address, comparevalue, newvalue, accdesc);

if BigEndian(accdesc.acctype) then
    X{datasize}(s)   = data[2*datasize-1:datasize];
    X{datasize}(s+1) = data[datasize-1:0];
else
    X{datasize}(s)   = data[datasize-1:0];
    X{datasize}(s+1) = data[2*datasize-1:datasize];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.56 CAST, CASAT, CASALT, CASLT

Compare and swap unprivileged

This instruction reads a 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of `WZR` or `XZR`, `CASAT` and `CASALT` load from memory with acquire semantics.
  - `CASLT` and `CASALT` store to memory with release semantics.
  - `CAST` has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `<Xs>`, is restored to the value held in the register before the instruction was executed.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For a `CAST` or `CASAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, this signals to the memory system that an additional subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` access to succeed when it does occur.

A code sequence starting with a `CAST` or `CASAT` instruction for which `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, and ending with a subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or `ISB` barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in `<Ws>` or `<Xs>` of the first `CAST` or `CASAT` is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a `CAST` or `CASAT` instruction, when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>`, the value in memory is not modified, because the `CAST` or `CASAT` either fails its compare or writes the same value back to memory.
For performance reasons, Arm recommends that software not use a `CAST` or `CASAT` instruction when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` in a routine that includes a subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT`.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction that conditionally determines whether or not to execute the subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT`.

This guidance does not apply when `<Ws>` or `<Xs>` specifies the same register as `<Wt>` or `<Xt>` for a `CAST` or `CASAT` instruction and the compare between the `CAST` or `CASAT` and the subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent `CAST`, `CASAT`, `CASALT`, or `CASLT` from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LSUI)

31	30	29							23	22	21	20					16	15	14				10	9				5	4				0
1	1	0	0	1	0	0	1	1		L	0			Rs				o0	1	1	1	1	1			Rn					Rt		
	sz																				Rt2												

Encoding for the CAST variant

Applies when `(L == 0 && o0 == 0)`

```
CAST  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAT variant

Applies when `(L == 1 && o0 == 0)`

```
CASAT  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the CASALT variant

Applies when `(L == 1 && o0 == 1)`

```
CASALT  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLT variant

Applies when `(L == 0 && o0 == 1)`

```
CASLT  <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = L == '1' && s != 31;
let release : boolean = o0 == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
var comparevalue : bits(64);
var newvalue : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                       tagchecked, privileged, t, s);
comparevalue = X{64}(s);
newvalue = X{64}(t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

X{64}(s) = MemAtomic{64}(address, comparevalue, newvalue, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.57 CBB<cc>

Compare bytes and branch

This instruction compares the byte values in two registers, and conditionally branches to a label at a PC-relative offset if the condition is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBBLE, CBBLO, CBBLS, and CBBLT.

Branch

(FEAT_CMPBR)

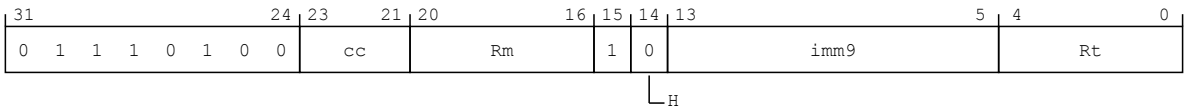

Encoding for the Greater than variant

Applies when `(cc == 000)`

```
CBBGT  <Wt>, <Wm>, <label>
```

Encoding for the Greater than or equal variant

Applies when `(cc == 001)`

```
CBBGE  <Wt>, <Wm>, <label>
```

Encoding for the Higher variant

Applies when `(cc == 010)`

```
CBBHI  <Wt>, <Wm>, <label>
```

Encoding for the Higher or same variant

Applies when `(cc == 011)`

```
CBBHS  <Wt>, <Wm>, <label>
```

Encoding for the Equal variant

Applies when `(cc == 110)`

```
CBBEQ  <Wt>, <Wm>, <label>
```

Encoding for the Not equal variant

Applies when `(cc == 111)`

```
CBBNE  <Wt>, <Wm>, <label>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 8 << UInt(H);
let t : integer = UInt(Rt);
let m : integer = UInt(Rm);
let offset : bits(64) = SignExtend{}(imm9::'00');
var op : CmpOp;
var unsigned : boolean;

case cc of
    when '000' => op = Cmp_GT; unsigned = FALSE;
    when '001' => op = Cmp_GE; unsigned = FALSE;
    when '010' => op = Cmp_GT; unsigned = TRUE;
    when '011' => op = Cmp_GE; unsigned = TRUE;
    when '110' => op = Cmp_EQ; unsigned = TRUE;
    when '111' => op = Cmp_NE; unsigned = TRUE;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

```
let operand1 : bits(datasize) = X{}(t);
let operand2 : bits(datasize) = X{}(m);
let branch_conditional : boolean = TRUE;

let value1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
let value2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
var cond : boolean;
case op of
    when Cmp_EQ => cond = value1 == value2;
    when Cmp_NE => cond = value1 != value2;
    when Cmp_GE => cond = value1 >= value2;
    when Cmp_GT => cond = value1 >  value2;
end;

if cond then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.58 CBBLE

Compare signed less than or equal to bytes and branch

This instruction compares the signed byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than or equal to the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 0 1	Rm	1	0	imm9	Rt
			cc			H		

Encoding

```
CBBLE  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBGE  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.59 CBBLO

Compare unsigned lower than bytes and branch

This instruction compares the unsigned byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 0 (cc)	Rm	1	0 (H)	imm9	Rt

Encoding

```
CBBLO  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBHI  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.60 CBBLS

Compare unsigned lower than or same as bytes and branch

This instruction compares the unsigned byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than or the same as the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 1 (cc)	Rm	1	0 (H)	imm9	Rt

Encoding

```
CBBLS  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBHS  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.61 CBBLT

Compare signed less than bytes and branch

This instruction compares the signed byte values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBB<cc>. This means:

- The encodings in this description are named to match the encodings of CBB<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBB<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31 29	28 26	25 24	23 21	20 16	15	14	13 5	4 0
0 1 1	1 0 1	0 0	0 0 0 (cc)	Rm	1	0 (H)	imm9	Rt

Encoding

```
CBBLT  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBBGT  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBB<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.62 CB<cc> (immediate)

Compare register with immediate and branch

This instruction compares the value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the comparison is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBGE (immediate), CBHS (immediate), CBLE (immediate), and CBLS (immediate).

Branch

(FEAT_CMPBR)

31	30	24	23 21	20 15	14	13 5	4 0
sf	1 1 1 0 1 0 1		cc	imm6	0	imm9	Rt

Encoding for the 32-bit greater than variant

Applies when `(sf == 0 && cc == 000)`

```
CBGT  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit less than variant

Applies when `(sf == 0 && cc == 001)`

```
CBLT  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit higher variant

Applies when `(sf == 0 && cc == 010)`

```
CBHI  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit lower variant

Applies when `(sf == 0 && cc == 011)`

```
CBLO  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit equal variant

Applies when `(sf == 0 && cc == 110)`

```
CBEQ  <Wt>, #<imm>, <label>
```

Encoding for the 32-bit not equal variant

Applies when `(sf == 0 && cc == 111)`

```
CBNE  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1 && cc == 000)`

```
CBGT  <Xt>, #<imm>, <label>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit less than variant

Applies when `(sf == 1 && cc == 001)`

```
CBLT  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1 && cc == 010)`

```
CBHI  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit lower variant

Applies when `(sf == 1 && cc == 011)`

```
CBLO  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit equal variant

Applies when `(sf == 1 && cc == 110)`

```
CBEQ  <Xt>, #<imm>, <label>
```

Encoding for the 64-bit not equal variant

Applies when `(sf == 1 && cc == 111)`

```
CBNE  <Xt>, #<imm>, <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 32 << UInt(sf);
let t : integer = UInt(Rt);
let offset : bits(64) = SignExtend{}(imm9::'00');
var op : CmpOp;
var unsigned : boolean;

case cc of
    when '000' => op = Cmp_GT; unsigned = FALSE;
    when '001' => op = Cmp_LT; unsigned = FALSE;
    when '010' => op = Cmp_GT; unsigned = TRUE;
    when '011' => op = Cmp_LT; unsigned = TRUE;
    when '110' => op = Cmp_EQ; unsigned = TRUE;
    when '111' => op = Cmp_NE; unsigned = TRUE;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
let value2 : integer = UInt(imm6);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<imm>

Is an unsigned immediate, in the range 0 to 63, encoded in the “imm6” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(t);
let branch_conditional : boolean = TRUE;

let value1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
var cond : boolean;
case op of
    when Cmp_EQ => cond = value1 == value2;
    when Cmp_NE => cond = value1 != value2;
    when Cmp_LT => cond = value1 <  value2;
    when Cmp_GT => cond = value1 >  value2;
end;

if cond then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.63 CB<cc> (register)

Compare registers and branch

This instruction compares the values in two registers, and conditionally branches to a label at a PC-relative offset if the condition is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBLE (register), CBLO (register), CBLS (register), and CBLT (register).

Branch

(FEAT_CMPBR)

31	30	24	23 21	20 16	15 14	13 5	4 0
sf	1 1 1 0 1 0 0		cc	Rm	0 0	imm9	Rt

Encoding for the 32-bit greater than variant

Applies when `(sf == 0 && cc == 000)`

```
CBGT  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit greater than or equal variant

Applies when `(sf == 0 && cc == 001)`

```
CBGE  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit higher variant

Applies when `(sf == 0 && cc == 010)`

```
CBHI  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit higher or same variant

Applies when `(sf == 0 && cc == 011)`

```
CBHS  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit equal variant

Applies when `(sf == 0 && cc == 110)`

```
CBEQ  <Wt>, <Wm>, <label>
```

Encoding for the 32-bit not equal variant

Applies when `(sf == 0 && cc == 111)`

```
CBNE  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1 && cc == 000)`

```
CBGT  <Xt>, <Xm>, <label>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit greater than or equal variant

Applies when `(sf == 1 && cc == 001)`

```
CBGE  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1 && cc == 010)`

```
CBHI  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit higher or same variant

Applies when `(sf == 1 && cc == 011)`

```
CBHS  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit equal variant

Applies when `(sf == 1 && cc == 110)`

```
CBEQ  <Xt>, <Xm>, <label>
```

Encoding for the 64-bit not equal variant

Applies when `(sf == 1 && cc == 111)`

```
CBNE  <Xt>, <Xm>, <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 32 << UInt(sf);
let t : integer = UInt(Rt);
let m : integer = UInt(Rm);
let offset : bits(64) = SignExtend{}(imm9::'00');
var op : CmpOp;
var unsigned : boolean;

case cc of
    when '000' => op = Cmp_GT; unsigned = FALSE;
    when '001' => op = Cmp_GE; unsigned = FALSE;
    when '010' => op = Cmp_GT; unsigned = TRUE;
    when '011' => op = Cmp_GE; unsigned = TRUE;
    when '110' => op = Cmp_EQ; unsigned = TRUE;
    when '111' => op = Cmp_NE; unsigned = TRUE;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(t);
let operand2 : bits(datasize) = X{}(m);
let branch_conditional : boolean = TRUE;

let value1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
let value2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
var cond : boolean;
case op of
    when Cmp_EQ => cond = value1 == value2;
    when Cmp_NE => cond = value1 != value2;
    when Cmp_GE => cond = value1 >= value2;
    when Cmp_GT => cond = value1 >  value2;
end;

if cond then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.64 CBGE (immediate)

Compare signed greater than or equal to immediate and branch

This instruction compares the signed value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is greater than or equal to the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20	15	14	13	5	4	0
sf	1	1	1	0	1	0	1	0	0	0	imm6		0	imm9		Rt	
										cc							

Encoding for the 32-bit greater than variant

Applies when `(sf == 0)`

```
CBGE  <Wt>, #<immp1>, <label>
```

is equivalent to

```
CBGT  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1)`

```
CBGE  <Xt>, #<immp1>, <label>
```

is equivalent to

```
CBGT  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<immp1>

Is an unsigned immediate, in the range 1 to 64, encoded as “imm6” plus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.65 CBH<cc>

Compare halfwords and branch

This instruction compares the halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the condition is true. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This instruction is used by the pseudo-instructions CBHLE, CBHLO, CBHLS, and CBHLT.

Branch

(FEAT_CMPBR)

31	24	23	21	20	16	15	14	13	5	4	0
0 1 1 1 0 1 0 0		cc		Rm		1	1 (H)	imm9		Rt	

Encoding for the Greater than variant

Applies when `(cc == 000)`

```
CBHGT  <Wt>, <Wm>, <label>
```

Encoding for the Greater than or equal variant

Applies when `(cc == 001)`

```
CBHGE  <Wt>, <Wm>, <label>
```

Encoding for the Higher variant

Applies when `(cc == 010)`

```
CBHHI  <Wt>, <Wm>, <label>
```

Encoding for the Higher or same variant

Applies when `(cc == 011)`

```
CBHHS  <Wt>, <Wm>, <label>
```

Encoding for the Equal variant

Applies when `(cc == 110)`

```
CBHEQ  <Wt>, <Wm>, <label>
```

Encoding for the Not equal variant

Applies when `(cc == 111)`

```
CBHNE  <Wt>, <Wm>, <label>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CMPBR) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 8 << UInt(H);
let t : integer = UInt(Rt);
let m : integer = UInt(Rm);
let offset : bits(64) = SignExtend{}(imm9::'00');
var op : CmpOp;
var unsigned : boolean;

case cc of
    when '000' => op = Cmp_GT; unsigned = FALSE;
    when '001' => op = Cmp_GE; unsigned = FALSE;
    when '010' => op = Cmp_GT; unsigned = TRUE;
    when '011' => op = Cmp_GE; unsigned = TRUE;
    when '110' => op = Cmp_EQ; unsigned = TRUE;
    when '111' => op = Cmp_NE; unsigned = TRUE;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

```
let operand1 : bits(datasize) = X{}(t);
let operand2 : bits(datasize) = X{}(m);
let branch_conditional : boolean = TRUE;

let value1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
let value2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
var cond : boolean;
case op of
    when Cmp_EQ => cond = value1 == value2;
    when Cmp_NE => cond = value1 != value2;
    when Cmp_GE => cond = value1 >= value2;
    when Cmp_GT => cond = value1 >  value2;
end;

if cond then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.66 CBHLE

Compare signed less than or equal to halfwords and branch

This instruction compares the signed halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than or equal to the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 0 1	Rm	1	1	imm9	Rt
			cc			H		

Encoding

```
CBHLE  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHGE  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.67 CBHLO

Compare unsigned lower than halfwords and branch

This instruction compares the unsigned halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 0 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLO  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHHI  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.68 CBHLS

Compare unsigned lower than or same as halfwords and branch

This instruction compares the unsigned halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than or the same as the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 1 1 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLS  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHHS  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.69 CBHLT

Compare signed less than halfwords and branch

This instruction compares the signed halfword values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CBH<cc>. This means:

- The encodings in this description are named to match the encodings of CBH<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CBH<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31–29	28–26	25–24	23–21	20–16	15	14	13–5	4–0
0 1 1	1 0 1	0 0	0 0 0 (cc)	Rm	1	1 (H)	imm9	Rt

Encoding

```
CBHLT  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHGT  <Wt>, <Wm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

Operation

The description of CBH<cc> gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.70 CBHS (immediate)

Compare unsigned higher than or same as immediate and branch

This instruction compares the unsigned value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is higher than or the same as the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20		15	14	13		5	4		0
sf	1	1	1	0	1	0	1	0	1	0	imm6			0	imm9			Rt		
										cc										

Encoding for the 32-bit higher variant

Applies when `(sf == 0)`

```
CBHS  <Wt>, #<immp1>, <label>
```

is equivalent to

```
CBHI  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1)`

```
CBHS  <Xt>, #<immp1>, <label>
```

is equivalent to

```
CBHI  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<immp1>

Is an unsigned immediate, in the range 1 to 64, encoded as “imm6” plus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.71 CBLE (immediate)

Compare signed less than or equal to immediate and branch

This instruction compares the signed value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is less than or equal to the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20	15	14	13	5	4	0
sf	1	1	1	0	1	0	1	0	0	1	imm6		0	imm9		Rt	
										cc							

Encoding for the 32-bit less than variant

Applies when `(sf == 0)`

```
CBLE  <Wt>, #<imms1>, <label>
```

is equivalent to

```
CBLT  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit less than variant

Applies when `(sf == 1)`

```
CBLE  <Xt>, #<imms1>, <label>
```

is equivalent to

```
CBLT  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<imms1>

Is a signed immediate, in the range -1 to 62, encoded as “imm6” minus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.72 CBLE (register)

Compare signed less than or equal to register and branch

This instruction compares the signed values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than or equal to the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13					5	4				0
sf	1	1	1	0	1	0	0	0	0	1	Rm				0	0	imm9						Rt				
										cc																	

Encoding for the 32-bit greater than or equal variant

Applies when `(sf == 0)`

```
CBLE  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBGE  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit greater than or equal variant

Applies when `(sf == 1)`

```
CBLE  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBGE  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.73 CBLO (register)

Compare unsigned lower than register and branch

This instruction compares the unsigned values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13					5	4				0
sf	1	1	1	0	1	0	0	0	1	0	Rm				0	0	imm9						Rt				
										cc																	

Encoding for the 32-bit higher variant

Applies when `(sf == 0)`

```
CBLO  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHI  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit higher variant

Applies when `(sf == 1)`

```
CBLO  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBHI  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.74 CBLS (immediate)

Compare unsigned lower than or same as immediate and branch

This instruction compares the unsigned value in a register with an immediate, and conditionally branches to a label at a PC-relative offset if the register value is lower than or the same as the immediate. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (immediate). This means:

- The encodings in this description are named to match the encodings of CB<cc> (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20		15	14	13		5	4		0
sf	1	1	1	0	1	0	1	0	1	1	imm6			0	imm9			Rt		
										cc										

Encoding for the 32-bit lower variant

Applies when `(sf == 0)`

```
CBLS  <Wt>, #<imms1>, <label>
```

is equivalent to

```
CBLO  <Wt>, #<imm>, <label>
```

Encoding for the 64-bit lower variant

Applies when `(sf == 1)`

```
CBLS  <Xt>, #<imms1>, <label>
```

is equivalent to

```
CBLO  <Xt>, #<imm>, <label>
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<imms1>

Is a signed immediate, in the range -1 to 62, encoded as “imm6” minus 1.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of CB<cc> (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.75 CBLS (register)

Compare unsigned lower than or same as register and branch

This instruction compares the unsigned values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is lower than or the same as the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13					5	4				0
sf	1	1	1	0	1	0	0	0	1	1	Rm				0	0	imm9						Rt				
										cc																	

Encoding for the 32-bit higher or same variant

Applies when `(sf == 0)`

```
CBLS  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBHS  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit higher or same variant

Applies when `(sf == 1)`

```
CBLS  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBHS  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.76 CBLT (register)

Compare signed less than register and branch

This instruction compares the signed values in two registers, and conditionally branches to a label at a PC-relative offset if the second value is less than the first. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

This is a pseudo-instruction of CB<cc> (register). This means:

- The encodings in this description are named to match the encodings of CB<cc> (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CB<cc> (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Branch

(FEAT_CMPBR)

31	30						24	23		21	20			16	15	14	13								5	4								0
sf	1	1	1	0	1	0	0	0	0	0	Rm				0	0	imm9									Rt							
										cc																							

Encoding for the 32-bit greater than variant

Applies when `(sf == 0)`

```
CBLT  <Wm>, <Wt>, <label>
```

is equivalent to

```
CBGT  <Wt>, <Wm>, <label>
```

Encoding for the 64-bit greater than variant

Applies when `(sf == 1)`

```
CBLT  <Xm>, <Xt>, <label>
```

is equivalent to

```
CBGT  <Xt>, <Xm>, <label>
```

Assembler Symbols

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range -1024 to 1020, is encoded as “imm9” times 4.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

The description of CB<cc> (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.77 CBNZ

Compare and branch on nonzero

This instruction compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

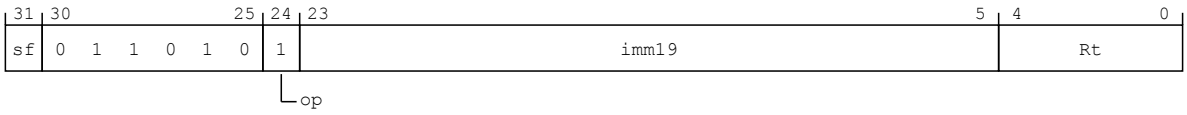

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CBNZ  <Wt>, <label>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CBNZ  <Xt>, <label>
```

Decode for all variants of this encoding

```
let t : integer = UInt(Rt);
let datasize : integer{} = 32 << UInt(sf);
let offset : bits(64) = SignExtend{}(imm19::'00');
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

```
let branch_conditional : boolean = TRUE;
let operand1 : bits(datasize) = X{}(t);
if !IsZero(operand1) then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.78 CBZ

Compare and branch on zero

This instruction compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

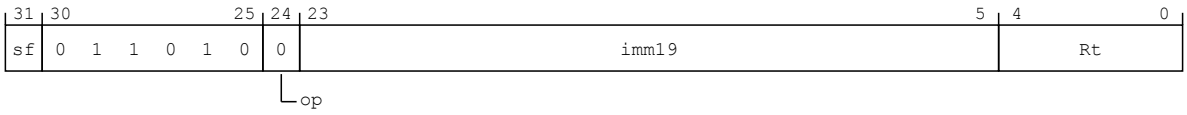

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CBZ  <Wt>, <label>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CBZ  <Xt>, <label>
```

Decode for all variants of this encoding

```
let t : integer = UInt(Rt);
let datasize : integer{} = 32 << UInt(sf);
let offset : bits(64) = SignExtend{}(imm19::'00');
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the “Rt” field.

Operation

```
let branch_conditional : boolean = TRUE;
let operand1 : bits(datasize) = X{}(t);
if IsZero(operand1) then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.79 CCMN (immediate)

Conditional compare negative (immediate)

This instruction sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

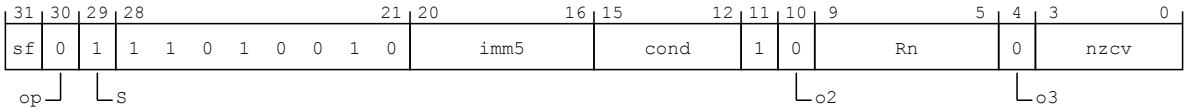

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMN  <Wn>, #<imm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMN  <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
var flags : bits(4) = nzcv;
let imm : bits(datasize) = ZeroExtend{}(imm5);
```

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is a five bit unsigned (positive) immediate encoded in the “imm5” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionHolds(condition) then
    let operand1 : bits(datasize) = X{}(n);
    let operand2 : bits(datasize) = imm;
    (-, flags) = AddWithCarry{datasize}(operand1, operand2, '0');
end;
PSTATE.[N,Z,C,V] = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.80 CCMN (register)

Conditional compare negative (register)

This instruction sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

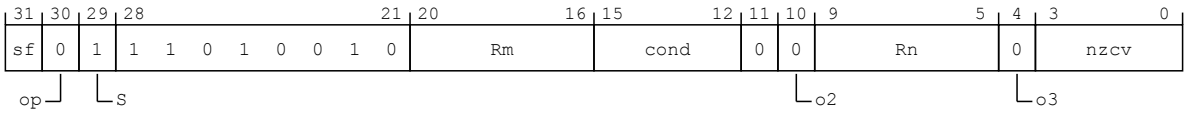

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMN  <Wn>, <Wm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMN  <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
var flags : bits(4) = nzcv;
```

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionHolds(condition) then
    let operand1 : bits(datasize) = X{}(n);
    let operand2 : bits(datasize) = X{datasize}(m);
    (-, flags) = AddWithCarry{datasize}(operand1, operand2, '0');
end;
PSTATE.[N,Z,C,V] = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.81 CCMP (immediate)

Conditional compare (immediate)

This instruction sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

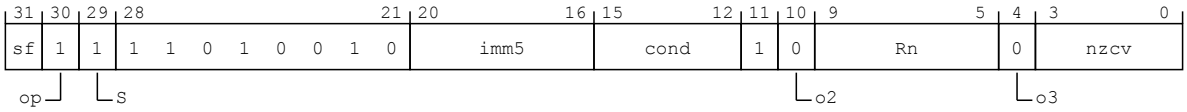

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMP  <Wn>, #<imm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMP  <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
var flags : bits(4) = nzcv;
let imm : bits(datasize) = ZeroExtend{}(imm5);
```

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is a five bit unsigned (positive) immediate encoded in the “imm5” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionHolds(condition) then
    let operand1 : bits(datasize) = X{}(n);
    let operand2 : bits(datasize) = imm;
    (-, flags) = AddWithCarry{datasize}(operand1, NOT(operand2), '1');
end;
PSTATE.[N,Z,C,V] = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.82 CCMP (register)

Conditional compare (register)

This instruction sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

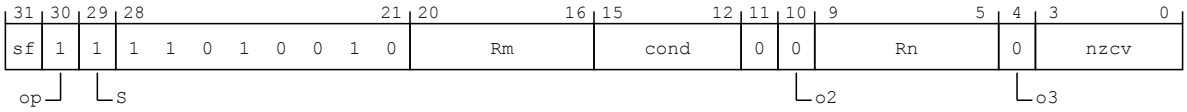

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CCMP  <Wn>, <Wm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CCMP  <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
var flags : bits(4) = nzcv;
```

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionHolds(condition) then
    let operand1 : bits(datasize) = X{}(n);
    let operand2 : bits(datasize) = X{datasize}(m);
    (-, flags) = AddWithCarry{datasize}(operand1, NOT(operand2), '1');
end;
PSTATE.[N,Z,C,V] = flags;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.83 CFINV

Invert carry flag

This instruction inverts the value of the PSTATE.C flag.

System

(FEAT_FlagM)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 0 0	0 1 0 0	(0) (0) (0) (0)	0 0 0	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
CFINV
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.84 CFP

Control flow prediction restriction by context

This instruction prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see CFP RCTX, Control Flow Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 0 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
CFP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #4, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.85 CHKFEAT

Check feature status

This instruction indicates the status of features. For more information, see Check Feature.

If FEAT_CHK is not implemented, this instruction executes as a `NOP`.

System

(FEAT_CHK)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 1 (CRm)	0 0 0 (op2)	1 1 1 1 1

Encoding

```
CHKFEAT  X16
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CHK) then EndOfDecode(Decode_NOP); end;
```

Operation

```
X{64}(16) = AArch64_ChkFeat(X{64}(16));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.86 CINC

Conditional increment

This instruction returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

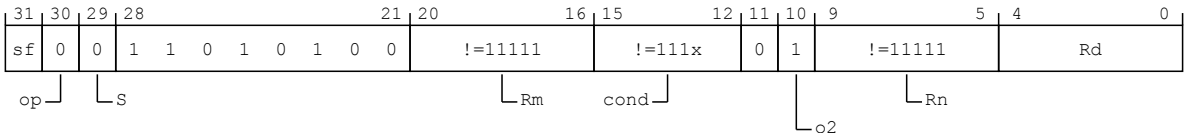

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CINC  <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSINC  <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CINC  <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSINC  <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational Information

The description of CSINC gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.87 CINV

Conditional invert

This instruction returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

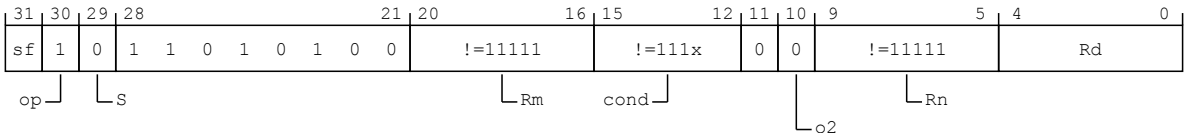

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CINV  <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSINV  <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CINV  <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSINV  <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational Information

The description of CSINV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.88 CLRBHB

Clear branch history

This instruction can be used to clear the branch history.

For more information, see Branch history.

System

(FEAT_CLRBHB)

31	29	28	26	25	12	11	8	7	5	4	0
1 1 0		1 0 1		0 1 0 0 0 0 0 0 1 1 0 0 1 0		0 0 1 0 (CRm)		1 1 0 (op2)		1 1 1 1 1	

Encoding

```
CLRBHB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then EndOfDecode(Decode_NOP); end;
```

Operation

```
Hint_CLRBHB();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.89 CLREX

Clear exclusive

This instruction clears the local monitor of the executing PE.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	0 1 0 (op2)	1 1 1 1 1 (Rt)

Encoding

```
CLREX  {#<imm>}
```

Decode for this encoding

```
// CRm field is ignored
```

Assembler Symbols

<imm>

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the “CRm” field.

Operation

```
ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.90 CLS

Count leading sign bits

This instruction determines the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. The result does not include the most significant bit of the source register.

31	30	29	28	21	20	16	15	11	10	9	5	4	0
sf	1	0	1 1 0 1 0 1 1 0		0 0 0 0 0		0 0 0 1 0		1	Rn		Rd	
		S			opcode2				op				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CLS  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CLS  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = CountLeadingSignBits(operand1);

X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.91 CLZ

Count leading zeros

This instruction determines the number of consecutive binary zero bits, starting from the most significant bit in the source register, and places the result in the destination register.

31	30	29	28							21	20					16	15					11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	0	1	0		0	Rn					Rd				
		S													opcode2								op										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CLZ  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CLZ  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = CountLeadingZeroBits(operand1);

X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.92 CMN (extended register)

Compare negative (extended register)

This instruction adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28				24	23	22	21	20			16	15		13	12		10	9			5	4				0
sf	0	1	0	1	0	1	1	0	0	1	Rm				option			imm3			Rn				1	1	1	1	1
op		S						opt																					Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMN  <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS  WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMN  <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS  XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTX when “option” is ‘011’.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
x11	X
10x	W
110	W

<m>

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (extended register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.93 CMN (immediate)

Compare negative (immediate)

This instruction adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

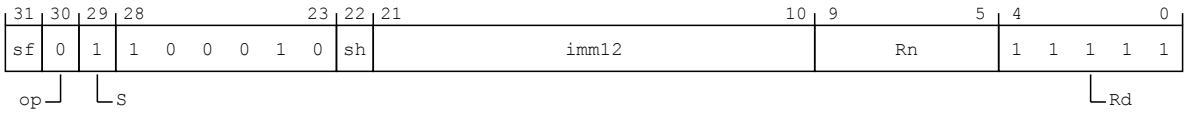

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMN  <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
ADDS  WZR, <Wn|WSP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMN  <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
ADDS  XZR, <Xn|SP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sh	<shift>
0	LSL #0
1	LSL #12

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.94 CMN (shifted register)

Compare negative (shifted register)

This instruction adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28				24	23	22	21	20			16	15					10	9				5	4				0
sf	0	1	0	1	0	1	1	shift		0	Rm				imm6						Rn					1	1	1	1	1
op		S																												Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMN  <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS  WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMN  <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS  XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.95 CMP (extended register)

Compare (extended register)

This instruction subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (extended register). This means:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28 … 24	23 22	21	20 … 16	15 … 13	12 … 10	9 … 5	4 … 0
sf	1	1	0 1 0 1 1	0 0	1	Rm	option	imm3	Rn	1 1 1 1 1
	op	S		opt						Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMP  <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS  WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMP  <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS  XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTX when “option” is ‘011’.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
x11	X
10x	W
110	W

<m>

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (extended register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.96 CMP (immediate)

Compare (immediate)

This instruction subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

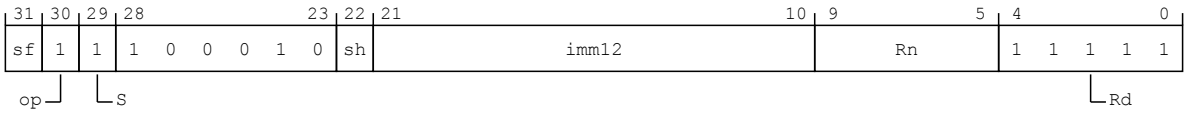

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMP  <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS  WZR, <Wn|WSP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMP  <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS  XZR, <Xn|SP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sh	<shift>
0	LSL #0
1	LSL #12

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.97 CMP (shifted register)

Compare (shifted register)

This instruction subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28				24	23	22	21	20			16	15					10	9				5	4				0
sf	1	1	0	1	0	1	1	shift		0	Rm				imm6						Rn					1	1	1	1	1
op		S																												Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CMP  <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CMP  <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.98 CMPP

Compare with tag

This instruction subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of SUBPS. This means:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	1	1	1 0 1	0 1 1 0	Rm	0 0 0 0 0 0	Rn	1 1 1 1 1
sf		S					opcode		Rd

Encoding

```
CMPP  <Xn|SP>, <Xm|SP>
```

is equivalent to

```
SUBPS  <Xd>, <Xn|SP>, <Xm|SP>
```

and is always the preferred disassembly.

Assembler Symbols

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Operation

The description of SUBPS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.99 CNEG

Conditional negate

This instruction returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	0	0	Rm					!=111x					0	1	Rn					Rd				
op		S														cond						o2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CNEG  <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSNEG  <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CNEG  <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSNEG  <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Operational Information

The description of CSNEG gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.100 CNT

Count bits

This instruction determines the number of binary one bits in the value of the source register, and writes the result to the destination register.

Integer

(FEAT_CSSC)

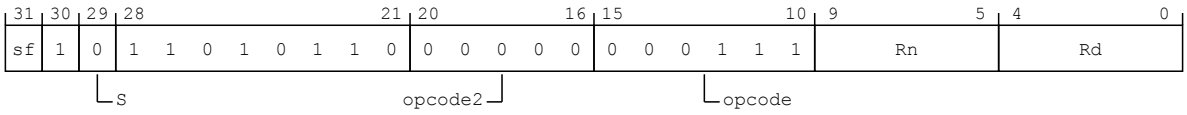

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CNT  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CNT  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 32 << UInt(sf);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = BitCount(operand1);
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.101 COSP

Clear other speculative prediction restriction by context

This instruction prevents predictions, other than Cache prefetch, Control flow, and Data Value predictions, that predict execution addresses based on information gathered from earlier execution within a particular execution context. Predictions, other than Cache prefetch, Control flow, and Data Value predictions, determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any speculative access occurring after the instruction is complete and synchronized.

For more information, see COSP RCTX, Clear Other Speculative Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES2)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 1 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
COSP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #6, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.102 CPP

Cache prefetch prediction restriction by context

This instruction prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

For more information, see CPP RCTX, Cache Prefetch Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 1 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
CPP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #7, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.103 CPYFP, CPYFM, CPYFE

Memory copy forward-only

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE.

CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFM copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFE copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFP:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFP, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFP, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFM, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFM, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFE, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFE, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFP  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFM  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFE  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.104 CPYFPN, CPYFMN, CPYFEN

Memory copy forward-only, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.105 CPYFPRN, CPYFMRN, CPYFERN

Memory copy forward-only, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.106 CPYFPRT, CPYFMRT, CPYFERT

Memory copy forward-only, reads unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRT:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRT, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRT, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 0 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERT  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.107 CPYFPRTN, CPYFMRTN, CPYFERTN

Memory copy forward-only, reads unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRTN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRTN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRTN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERTN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.108 CPYFPRTRN, CPYFMRTRN, CPYFERTRN

Memory copy forward-only, reads unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN.

CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRTRN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRTRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRTRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRTRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRTRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERTRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERTRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

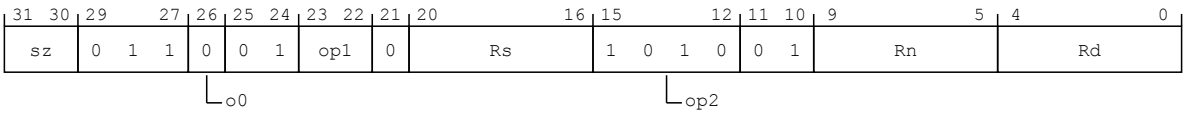

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.109 CPYFPRTWN, CPYFMRTWN, CPYFERTWN

Memory copy forward-only, reads unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMRTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFERTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPRTWN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPRTWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPRTWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMRTWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMRTWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFERTWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFERTWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFERTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.110 CPYFPT, CPYFMT, CPYFET

Memory copy forward-only, reads and writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFET copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPT:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPT, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPT, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFET, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFET, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFET  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.111 CPYFPTN, CPYFMTN, CPYFETN

Memory copy forward-only, reads and writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFETN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPTN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPTN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPTN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFETN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFETN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFETN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.112 CPYFPTRN, CPYFMTRN, CPYFETRN

Memory copy forward-only, reads and writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFETRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPTRN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPTRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPTRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMTRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMTRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFETRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFETRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFETRN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.113 CPYFPTWN, CPYFMTWN, CPYFETWN

Memory copy forward-only, reads and writes unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFETWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPTWN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPTWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPTWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMTWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMTWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFETWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFETWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

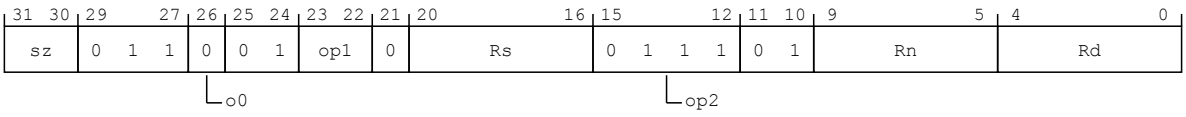

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFETWN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.114 CPYFPWN, CPYFMWN, CPYFEWN

Memory copy forward-only, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN, then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

<Xs>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.115 CPYFPWT, CPYFMWT, CPYFEWT

Memory copy forward-only, writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWT:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWT, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWT, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

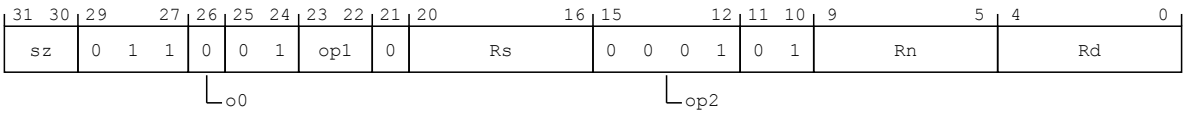

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWT  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.116 CPYFPWTN, CPYFMWTN, CPYFEWTN

Memory copy forward-only, writes unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWTN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWTN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWTN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.117 CPYFPWTRN, CPYFMWTRN, CPYFEWTRN

Memory copy forward-only, writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWTRN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWTRN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWTRN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWTRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWTRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWTRN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWTRN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	1 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.118 CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory copy forward-only, writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address in a forward direction. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN.

CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYFMWTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYFEWTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than or equal to the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYFPWTWN:

- If Xn[63] == ‘1’, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of CPYFPWTWN, option A:

- Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the original Xs + saturated copy size.
- Xd holds the original Xd + saturated copy size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of CPYFPWTWN, option B:

- Xn holds the number of bytes in the saturated copy size remaining to be copied.
- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For CPYFMWTWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.

For CPYFMWTWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be copied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds the lowest address that has not been copied from.
- Xd holds the lowest address that has not been copied to.

For CPYFEWTWN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from - Xn.
- Xd holds the lowest address to be copied to - Xn.
- On completion of the instruction, Xn holds 0.

For CPYFEWTWN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be copied.
- Xs holds the lowest address to be copied from.
- Xd holds the lowest address to be copied to.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	op1	0	Rs	0 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYFPWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYFMWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYFEWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);
memcpy.cpysize     = SInt(X{64}(memcpy.n));
memcpy.implements_option_a = CPYFOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize[63] == '1' then memcpy.cpysize = ArchMaxMOPSBlockSize; end;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize     = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
memcpy.forward = TRUE;
var fault : boolean  = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   + memcpy.cpysize,
                                                                memcpy.fromaddress + memcpy.cpysize,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize      = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
        end;
    end;
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                memcpy.fromaddress,
                                                                memcpy.forward, B,
                                                                raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress  = memcpy.fromaddress + B;
            memcpy.toaddress    = memcpy.toaddress   + B;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.cpysize      = memcpy.cpysize     - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if IsFault(memstatus) then
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.119 CPYP, CPYM, CPYE

Memory copy

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE.

CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYM copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYE copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYP:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYP, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYP, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYM, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYM, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYE, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYE, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYP  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYM  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYE  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.120 CPYPN, CPYMN, CPYEN

Memory copy, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYEN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.121 CPYPRN, CPYMRN, CPYERN

Memory copy, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYERN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.122 CPYPRT, CPYMRT, CPYERT

Memory copy, reads unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRT:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRT, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRT, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYERT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.123 CPYPRTN, CPYMRTN, CPYERTN

Memory copy, reads unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRTN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRTN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRTN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYERTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.124 CPYPRTRN, CPYMRTRN, CPYERTRN

Memory copy, reads unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRTRN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRTRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRTRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYERTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.125 CPYPRTWN, CPYMRTWN, CPYERTWN

Memory copy, reads unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMRTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYERTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPRTWN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPRTWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPRTWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMRTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMRTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYERTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYERTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Read effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Read effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 1 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMRTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYERTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.126 CPYPT, CPYMT, CPYET

Memory copy, reads and writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYET copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPT:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPT, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPT, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYET, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYET, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYET  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.127 CPYPTN, CPYMTN, CPYETN

Memory copy, reads and writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYETN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPTN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPTN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPTN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYETN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYETN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYETN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.128 CPYPTRN, CPYMTRN, CPYETRN

Memory copy, reads and writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYETRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPTRN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPTRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPTRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYETRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYETRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYETRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.129 CPYPTWN, CPYMTWN, CPYETWN

Memory copy, reads and writes unprivileged, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYETWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPTWN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPTWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPTWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYETWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYETWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 1 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYETWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.130 CPYPWN, CPYMWN, CPYEWN

Memory copy, writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYEWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 0 0	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.131 CPYPWT, CPYMWT, CPYEWT

Memory copy, writes unprivileged

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWT copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWT copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWT:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWT, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWT, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWT, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYEWT, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWT  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.132 CPYPWTN, CPYMWTN, CPYEWTN

Memory copy, writes unprivileged, reads and writes non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWTN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWTN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWTN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWTN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWTN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWTN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYEWTN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWTN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.133 CPYPWTRN, CPYMWTRN, CPYEWTRN

Memory copy, writes unprivileged, reads non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWTRN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWTRN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWTRN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWTRN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWTRN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWTRN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYEWTRN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	1 0 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWTRN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize     = 0 - memcpy.cpysize;
        end;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';
        end;
    end;
end;

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
    CheckMemCpyParams(memcpy, options);
end;

var copied : integer;
var iswrite : boolean;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var fault : boolean = FALSE;
var B : MOPSBlockSize = 0;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B,
                                                        raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize      = memcpy.cpysize      - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                        memcpy.toaddress   + memcpy.cpysize,
                                                        memcpy.fromaddress + memcpy.cpysize,
                                                        memcpy.forward, B, raccdesc,
                                                        waccdesc);
            if copied != B then
                fault               = TRUE;
                memcpy.cpysize      = memcpy.cpysize      + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.134 CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory copy, writes unprivileged and non-temporal

These instructions copy a requested number of bytes in memory from a source address to a destination address. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and copies an IMPLEMENTATION DEFINED portion of the requested number of bytes. CPYMWTWN copies a further IMPLEMENTATION DEFINED portion of the remaining bytes. CPYEWTWN copies any final remaining bytes.

Note

The ability to copy an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being copied are divided between the different instructions.

For more information on exceptions specific to memory copy instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For CPYPWTWN:

- If Xn[63:55] != ‘000000000’, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.
- After saturation is performed, the direction of the memory copy is based on the following:
  If (Xs[55:0] > Xd[55:0]) and (Xd[55:0] + saturated copy size) > Xs[55:0], then the direction is forward.
  If (Xs[55:0] < Xd[55:0]) and (Xs[55:0] + saturated copy size) > Xd[55:0], then the direction is backward.
  Otherwise, the direction is an IMPLEMENTATION DEFINED choice between forward and backward.

On completion of CPYPWTWN, option A:

- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.
- If the copy is in the forward direction, then:
  - Xn holds -1 times the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the original Xs + saturated copy size.
  - Xd holds the original Xd + saturated copy size.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs and Xd are unchanged.

On completion of CPYPWTWN, option B:

- If the copy is in the forward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the lowest address that has not been copied from.
  - Xd holds the lowest address that has not been copied to.
  - PSTATE.{N,Z,C,V} are set to {0,0,1,0}.
- If the copy is in the backward direction, then:
  - Xn holds the number of bytes in the saturated copy size remaining to be copied.
  - Xs holds the highest address that has not been copied from + 1.
  - Xd holds the highest address that has not been copied to + 1.
  - PSTATE.{N,Z,C,V} are set to {1,0,1,0}.

For CPYMWTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
  - On completion of the instruction, Xn holds -1 times the number of bytes remaining to be copied.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
  - On completion of the instruction, Xn holds the number of bytes remaining to be copied.

For CPYMWTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the lowest address that has not been copied from.
    - Xd holds the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
  - On completion of the instruction:
    - Xn holds the number of bytes remaining to be copied.
    - Xs holds the highest address that has not been copied from + 1.
    - Xd holds the highest address that has not been copied to + 1.

For CPYEWTWN, option A, when PSTATE.C = 0:

- Xn holds a signed 64-bit integer.
- If the copy is in the forward direction (Xn holds a negative number), then:
  - Xn holds -1 times the number of bytes remaining to be copied.
  - Xs holds the lowest address to be copied from - Xn.
  - Xd holds the lowest address to be copied to - Xn.
- If the copy is in the backward direction (Xn holds a positive number), then:
  - Xn holds the number of bytes remaining to be copied.
  - Xs holds the highest address to be copied from + 1 - Xn.
  - Xd holds the highest address to be copied to + 1 - Xn.
- On completion of the instruction:
  - Xn holds 0.
  - Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied from.
    - The highest address that has been copied from + 1.
  - Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
    - The lowest address that has been copied to.
    - The highest address that has not been copied to + 1.

For CPYEWTWN, option B, when PSTATE.C = 1:

- Xn holds the number of bytes remaining to be copied.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address to be copied from.
  - Xd holds the lowest address to be copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address to be copied from + 1.
  - Xd holds the highest address to be copied to + 1.
- On completion of the instruction:
  - Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xs holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied from.
  - The highest address that has been copied from + 1.
- Xd holds an IMPLEMENTATION SPECIFIC value in the range with bounds:
  - The lowest address that has been copied to.
  - The highest address that has not been copied to + 1.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	op1	0	Rs	0 1 0 1	0 1	Rn	Rd
		o0					op2			

Encoding for the Prologue variant

Applies when `(op1 == 00)`

```
CPYPWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Main variant

Applies when `(op1 == 01)`

```
CPYMWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Encoding for the Epilogue variant

Applies when `(op1 == 10)`

```
CPYEWTWN  [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memcpy : CPYParams;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
let options : bits(4) = op2;
let rnontemporal : boolean = options[3] == '1';
let wnontemporal : boolean = options[2] == '1';
case op1 of
    when '00' => memcpy.stage = MOPSStage_Prologue;
    when '01' => memcpy.stage = MOPSStage_Main;
    when '10' => memcpy.stage = MOPSStage_Epilogue;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set CPY* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the “Rd” field.

<Xs>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the “Rs” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the “Rs” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero on completion of the instruction, encoded in the “Rn” field.

Operation

```
CheckMOPSEnabled();

CheckCPYConstrainedUnpredictable(memcpy.n, memcpy.d, memcpy.s);

memcpy.nzcv        = PSTATE.[N,Z,C,V];
memcpy.toaddress   = X{64}(memcpy.d);
memcpy.fromaddress = X{64}(memcpy.s);

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X{64}(memcpy.n));
else
    memcpy.cpysize = SInt(X{64}(memcpy.n));
end;

memcpy.implements_option_a = CPYOptionA();

let rprivileged : boolean = (if options[1] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);
let wprivileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                             else PSTATE.EL != EL0);

let raccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_LOAD,  rprivileged, rnontemporal);
let waccdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > ArchMaxMOPSCPYSize then
        memcpy.cpysize = ArchMaxMOPSCPYSize;
    end;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.cpysize     = 0 - memcpy.cpysize;
            end;
        else
            if !memcpy.forward then
                // Copy in the reverse direction offsets the arguments.
                memcpy.toaddress   = memcpy.toaddress   + memcpy.cpysize;
                memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
                memcpy.nzcv = '1010';
            else
                memcpy.nzcv = '0010';
            end;
        end;
    end;

    memcpy.stagecpysize = MemCpyStageSize(memcpy);

    if memcpy.stage != MOPSStage_Prologue then
        memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv[3] == '0');
        CheckMemCpyParams(memcpy, options);
    end;

    var copied : integer;
    var iswrite : boolean;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var fault : boolean = FALSE;
    var B : MOPSBlockSize = 0;

    if memcpy.implements_option_a then
        while memcpy.stagecpysize != 0 && !fault looplimit ArchMaxMOPSCPYSize do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(memcpy);

            if memcpy.forward then
                assert B <= -1 * memcpy.stagecpysize;
                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B,
                                                            raccdesc, waccdesc);
                if copied != B then
                    fault = TRUE;
                else
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;

            else
                assert B <= memcpy.stagecpysize;
                memcpy.cpysize      = memcpy.cpysize      - B;
                memcpy.stagecpysize = memcpy.stagecpysize - B;

                (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(
                                                            memcpy.toaddress   + memcpy.cpysize,
                                                            memcpy.fromaddress + memcpy.cpysize,
                                                            memcpy.forward, B, raccdesc,
                                                            waccdesc);
                if copied != B then
                    fault               = TRUE;
                    memcpy.cpysize      = memcpy.cpysize      + B;
                    memcpy.stagecpysize = memcpy.stagecpysize + B;
                end;
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    while memcpy.stagecpysize > 0 && !fault looplimit ArchMaxMOPSCPYSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
                                                                    memcpy.fromaddress,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress   = memcpy.toaddress   + B;
            end;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress   - B,
                                                                    memcpy.fromaddress - B,
                                                                    memcpy.forward, B,
                                                                    raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress   = memcpy.toaddress   - B;
            end;
        end;

        if !fault then
            memcpy.cpysize      = memcpy.cpysize        - B;
            memcpy.stagecpysize = memcpy.stagecpysize   - B;
        end;
    end;
end;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let accdesc : AccessDescriptor = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memcpy.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.135 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum

This instruction performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x04C11DB7` is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)

31	30	29	28							21	20				16	15			13	12	11		10	9					5	4					0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	0		0	sz			Rn						Rd					
		S																		C															

Encoding for the CRC32B variant

Applies when `(sf == 0 && sz == 00)`

```
CRC32B  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32H variant

Applies when `(sf == 0 && sz == 01)`

```
CRC32H  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32W variant

Applies when `(sf == 0 && sz == 10)`

```
CRC32W  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32X variant

Applies when `(sf == 1 && sz == 11)`

```
CRC32X  <Xd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if sf == '1' && sz != '11' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && sz == '11' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = 8 << UInt(sz);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose accumulator output register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose accumulator input register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the general-purpose data source register, encoded in the “Rm” field.

<Xm>

Is the 64-bit name of the general-purpose data source register, encoded in the “Rm” field.

Operation

```
let acc : bits(32)      = X{}(n);      // accumulator
let val : bits(size)      = X{}(m);    // input value
let poly : bits(32)     = 0x04C11DB7[31:0];

let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
let tempval : bits(size+32) = BitReverse{size}(val)::Zeros{32};

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X{32}(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.136 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32C checksum

This instruction performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)

31	30	29	28							21	20			16	15			13	12	11	10	9				5	4			0
sf	0	0	1	1	0	1	0	1	1	0	Rm				0	1	0		1	sz		Rn					Rd			
		S																	C											

Encoding for the CRC32CB variant

Applies when `(sf == 0 && sz == 00)`

```
CRC32CB  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CH variant

Applies when `(sf == 0 && sz == 01)`

```
CRC32CH  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CW variant

Applies when `(sf == 0 && sz == 10)`

```
CRC32CW  <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CX variant

Applies when `(sf == 1 && sz == 11)`

```
CRC32CX  <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if sf == '1' && sz != '11' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && sz == '11' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = 8 << UInt(sz);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose accumulator output register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose accumulator input register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the general-purpose data source register, encoded in the “Rm” field.

<Xm>

Is the 64-bit name of the general-purpose data source register, encoded in the “Rm” field.

Operation

```
let acc : bits(32)     = X{}(n);     // accumulator
let val : bits(size)     = X{}(m);   // input value
let poly : bits(32)    = 0x1EDC6F41[31:0];

let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
let tempval : bits(size+32) = BitReverse{size}(val)::Zeros{32};

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X{32}(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.137 CSDB

Consumption of speculative data barrier

This instruction is a memory barrier that controls speculative execution arising from data value prediction. For more information and details of the semantics, see Consumption of Speculative Data Barrier (CSDB).

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	1 0 0 (op2)	1 1 1 1 1

Encoding

```
CSDB
```

Decode for this encoding

```
// Empty.
```

Operation

```
ConsumptionOfSpeculativeDataBarrier();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.138 CSEL

Conditional select

This instruction writes the value of the first source register to the destination register if the condition is TRUE. If the condition is FALSE, it writes the value of the second source register to the destination register.

31	30	29	28 … 21	20 … 16	15 … 12	11	10	9 … 5	4 … 0
sf	0	0	1 1 0 1 0 1 0 0	Rm	cond	0	0	Rn	Rd
	op	S					o2		

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSEL  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSEL  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
var result : bits(datasize);
if ConditionHolds(condition) then
    result = X{datasize}(n);
else
    result = X{datasize}(m);
end;

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.139 CSET

Conditional set

This instruction sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

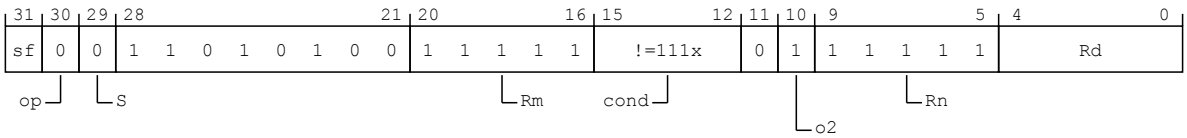

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSET  <Wd>, <invcond>
```

is equivalent to

```
CSINC  <Wd>, WZR, WZR, <cond>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSET  <Xd>, <invcond>
```

is equivalent to

```
CSINC  <Xd>, XZR, XZR, <cond>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational Information

The description of CSINC gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.140 CSETM

Conditional set mask

This instruction sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

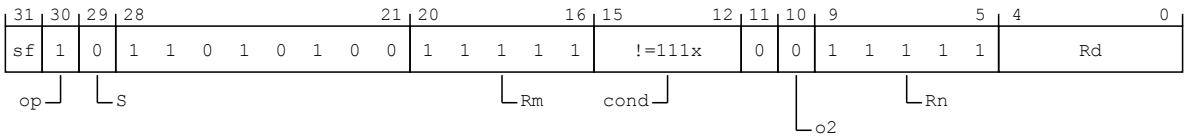

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSETM  <Wd>, <invcond>
```

is equivalent to

```
CSINV  <Wd>, WZR, WZR, <cond>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSETM  <Xd>, <invcond>
```

is equivalent to

```
CSINV  <Xd>, XZR, XZR, <cond>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<invcond>

Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in “cond”:

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational Information

The description of CSINV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.141 CSINC

Conditional select increment

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET.

31	30	29	28							21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1	1	0	1	0	1	0	0	Rm		cond		0	1	Rn		Rd	
	op	S														o2				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSINC  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSINC  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CINC	Rm != '11111' && Rn != '11111' && !(cond IN {'111x'}) && Rn == Rm
CSET	Rm == '11111' && Rn == '11111' && !(cond IN {'111x'})

Operation

```
var result : bits(datasize);
if ConditionHolds(condition) then
    result = X{datasize}(n);
else
    result = X{datasize}(m) + 1;
end;

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.142 CSINV

Conditional select invert

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV and CSETM.

31	30	29	28							21	20			16	15			12	11	10	9			5	4			0
sf	1	0	1	1	0	1	0	1	0	0	Rm				cond				0	0	Rn				Rd			
	op	S																		o2								

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSINV  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSINV  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CINV	Rm != '11111' && Rn != '11111' && !(cond IN {'111x'}) && Rn == Rm
CSETM	Rm == '11111' && Rn == '11111' && !(cond IN {'111x'})

Operation

```
var result : bits(datasize);
if ConditionHolds(condition) then
    result = X{datasize}(n);
else
    result = NOT(X{datasize}(m));
end;

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.143 CSNEG

Conditional select negation

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG.

31	30	29	28							21	20			16	15			12	11	10	9			5	4			0
sf	1	0	1	1	0	1	0	1	0	0	Rm				cond				0	1	Rn				Rd			
	op			S																								
																				o2								

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CSNEG  <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CSNEG  <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let condition : bits(4) = cond;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CNEG	`!(cond IN {'111x'}) && Rn == Rm`

Operation

```
var result : bits(datasize);
if ConditionHolds(condition) then
    result = X{datasize}(n);
else
    result = NOT(X{datasize}(m)) + 1;
end;

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.144 CTZ

Count trailing zeros

This instruction determines the number of consecutive binary zero bits, starting from the least significant bit in the source register, and places the result in the destination register.

Integer

(FEAT_CSSC)

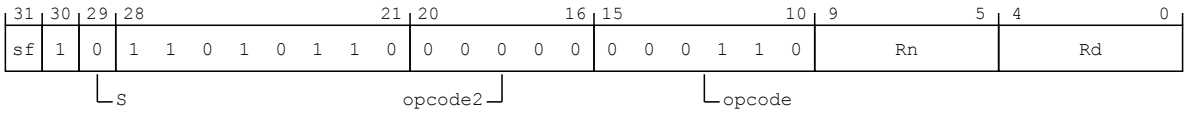

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
CTZ  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
CTZ  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let datasize : integer{} = 32 << UInt(sf);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = CountLeadingZeroBits(BitReverse{datasize}(operand1));
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.145 DC

Data cache operation

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

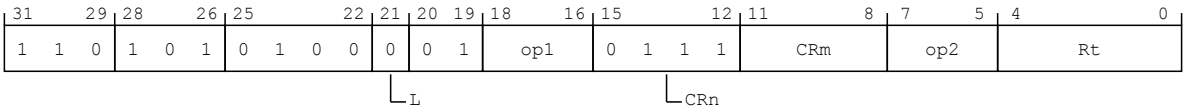

Encoding

```
DC  <dc_op>, <Xt>
```

is equivalent to

```
SYS  #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_DC`.

Assembler Symbols

<dc_op>

Is a DC operation name, as listed for the DC system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<dc_op>	Architectural Feature	Description
000	0110	001	IVAC	-	For more information, see DC IVAC.
000	0110	010	ISW	-	For more information, see DC ISW.
000	0110	011	IGVAC	FEAT_MTE2	For more information, see DC IGVAC.
000	0110	100	IGSW	FEAT_MTE2	For more information, see DC IGSW.
000	0110	101	IGDVAC	FEAT_MTE2	For more information, see DC IGDVAC.
000	0110	110	IGDSW	FEAT_MTE2	For more information, see DC IGDSW.
000	1010	010	CSW	-	For more information, see DC CSW.
000	1010	100	CGSW	FEAT_MTE2	For more information, see DC CGSW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRm	op2	<dc_op>	Architectural Feature	Description
000	1010	110	CGDSW	FEAT_MTE2	For more information, see DC CGDSW.
000	1110	010	CISW	-	For more information, see DC CISW.
000	1110	100	CIGSW	FEAT_MTE2	For more information, see DC CIGSW.
000	1110	110	CIGDSW	FEAT_MTE2	For more information, see DC CIGDSW.
000	1111	001	CIVAPS	FEAT_PoPS	For more information, see DC CIVAPS.
000	1111	101	CIGDVAPS	FEAT_PoPS && FEAT_MTE2	For more information, see DC CIGDVAPS.
011	0100	001	ZVA	-	For more information, see DC ZVA.
011	0100	011	GVA	FEAT_MTE	For more information, see DC GVA.
011	0100	100	GZVA	FEAT_MTE	For more information, see DC GZVA.
011	1010	001	CVAC	-	For more information, see DC CVAC.
011	1010	011	CGVAC	FEAT_MTE	For more information, see DC CGVAC.
011	1010	101	CGDVAC	FEAT_MTE	For more information, see DC CGDVAC.
011	1011	000	CVAOC	FEAT_OCCMO	For more information, see DC CVAOC.
011	1011	001	CVAU	-	For more information, see DC CVAU.
011	1011	111	CGDVAOC	FEAT_OCCMO && FEAT_MTE	For more information, see DC CGDVAOC.
011	1100	001	CVAP	FEAT_DPB	For more information, see DC CVAP.
011	1100	011	CGVAP	FEAT_MTE	For more information, see DC CGVAP.
011	1100	101	CGDVAP	FEAT_MTE	For more information, see DC CGDVAP.
011	1101	001	CVADP	FEAT_DPB2	For more information, see DC CVADP.
011	1101	011	CGVADP	FEAT_MTE	For more information, see DC CGVADP.
011	1101	101	CGDVADP	FEAT_MTE	For more information, see DC CGDVADP.
011	1110	001	CIVAC	-	For more information, see DC CIVAC.
011	1110	011	CIGVAC	FEAT_MTE	For more information, see DC CIGVAC.
011	1110	101	CIGDVAC	FEAT_MTE	For more information, see DC CIGDVAC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRm	op2	<dc_op>	Architectural Feature	Description
011	1111	000	CIVAOC	FEAT_OCCMO	For more information, see DC CIVAOC.
011	1111	111	CIGDVAOC	FEAT_OCCMO && FEAT_MTE	For more information, see DC CIGDVAOC.
100	1110	000	CIPAE	FEAT_MEC	For more information, see DC CIPAE.
100	1110	111	CIGDPAE	FEAT_MEC && FEAT_MTE2	For more information, see DC CIGDPAE.
110	1110	001	CIPAPA	FEAT_RME	For more information, see DC CIPAPA.
110	1110	101	CIGDPAPA	FEAT_RME && FEAT_MTE2	For more information, see DC CIGDPAPA.

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.146 DCPS1

Debug change PE state to EL1

This instruction, when executed in Debug state:

- If executed at EL0, changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target Exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at EL0 if EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE == 1.

For more information on the operation of the DCPS<n> instructions, see DCPS.

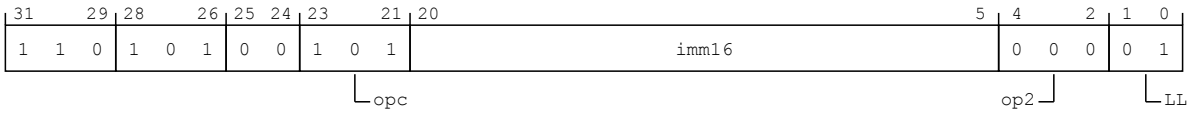

Encoding

```
DCPS1  {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm>

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the “imm16” field.

Operation

```
if !Halted() then Undefined(); end;
DCPSInstruction(EL1);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.147 DCPS2

Debug change PE state to EL2

This instruction, when executed in Debug state:

- If executed at EL0 or EL1, changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target Exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an Exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at the following Exception levels:

- All Exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

For more information on the operation of the DCPS<n> instructions, see DCPS.

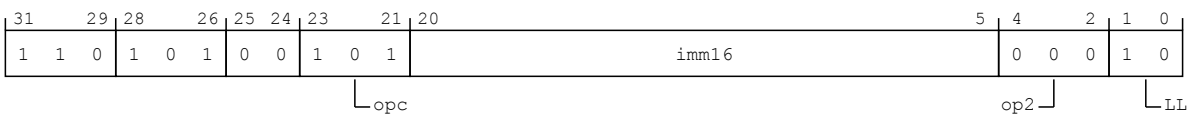

Encoding

```
DCPS2  {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm>

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the “imm16” field.

Operation

```
if !Halted() then Undefined(); end;
DCPSInstruction(EL2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.148 DCPS3

Debug change PE state to EL3

This instruction, when executed in Debug state:

- If executed at EL3, selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target Exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at all Exception levels if either:

- EDSCR.SDD == 1.
- EL3 is not implemented.

For more information on the operation of the DCPS<n> instructions, see DCPS.

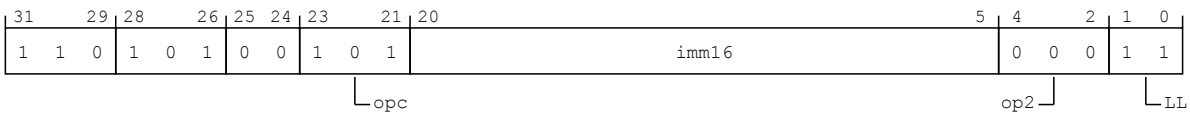

Encoding

```
DCPS3  {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm>

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the “imm16” field.

Operation

```
if !Halted() then Undefined(); end;
DCPSInstruction(EL3);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.149 DGH

Data gathering hint

This instruction is a hint instruction that indicates that it is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

System

(FEAT_DGH)

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0	1 1 0	1 1 1 1 1
			CRm	op2	

Encoding

```
DGH
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DGH) then EndOfDecode(Decode_NOP); end;
```

Operation

```
Hint_DGH();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.150 DMB

Data memory barrier

This instruction is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	1	0 1 (opc)	1 1 1 1 1 (Rt)

Encoding

```
DMB  (<option>|#<imm>)
```

Decode for this encoding

```
var types : MBReqTypes;
case CRm[1:0] of
    when '00' => types = MBReqTypes_All;
    when '01' => types = MBReqTypes_Reads;
    when '10' => types = MBReqTypes_Writes;
    when '11' => types = MBReqTypes_All;
end;
```

Assembler Symbols

<option>

Specifies the limitation on the Memory Effects ordered by the barrier operation, and is encoded in “CRm”.

For more information on whether a Memory Effect is before or after a barrier instruction, see Data Memory Barrier (DMB).

CRm	<option>	Description
0001	OSHLD	This option has the same behavior as LD and is deprecated.
0010	OSHST	This option has the same behavior as ST and is deprecated.
0011	OSH	This option has the same behavior as SY and is deprecated.
0101	NSHLD	This option has the same behavior as LD and is deprecated.
0110	NSHST	This option has the same behavior as ST and is deprecated.
0111	NSH	This option has the same behavior as SY and is deprecated.
xx00	RESERVED	•
1001	ISHLD	This option has the same behavior as LD and is deprecated.
1010	ISHST	This option has the same behavior as ST and is deprecated.
1011	ISH	This option has the same behavior as SY and is deprecated.
1101	LD	Reads are the required access type before the barrier instruction. Reads and writes are the required access types after the barrier instruction.
1110	ST	Writes are the required access type, both before and after the barrier instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	<option>	Description
1111	SY	Reads and writes are the required access types, both before and after the barrier instruction.

All encodings of “CRm” that are listed as reserved can be encoded using the #<imm> syntax. All reserved options have the same behavior as SY, but software must not rely on this behavior.

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Operation

```
DataMemoryBarrier(types);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.151 DRPS

Debug restore PE state

This instruction restores PSTATE from the SPSR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

This instruction is UNDEFINED in Non-debug state.

This instruction is UNDEFINED at EL0.

For more information on the operation of DRPS, see DRPS.

31 29	28 26	25	24 21	20 16	15 10	9 5	4 0
1 1 0	1 0 1	1	0 1 0 1	1 1 1 1 1	0 0 0 0 0 0	1 1 1 1 1	0 0 0 0 0
			opc	op2	op3	Rn	op4

Encoding

```
DRPS
```

Decode for this encoding

```
// Empty.
```

Operation

```
if !Halted() || PSTATE.EL == EL0 then Undefined(); end;
DRPSInstruction();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.152 DSB

Data synchronization barrier

This instruction is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

This instruction is used by the aliases PSSBB and SSBB.

It has encodings from 2 classes: Memory barrier and Memory nXS barrier

Memory barrier

31 29	28 26	25 12	11 8	7	6 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	1	0 0	1 1 1 1 1
					opc	Rt

Encoding

```
DSB  (<option>|#<imm>)
```

Decode for this encoding

```
var nXS : boolean = FALSE;

var alias : DSBAlias;
case CRm of
    when '0000' => alias = DSBAlias_SSBB;
    when '0100' => alias = DSBAlias_PSSBB;
    otherwise   => alias = DSBAlias_DSB;
end;


var scope : MBMaintenanceScope;
var types : MBReqTypes;
case CRm[1:0] of
    when '00' =>
        types = MBReqTypes_All;
        scope = MBMaintenanceScope_OuterShareable;
    when '01' =>
        types = MBReqTypes_Reads;
        scope = MBMaintenanceScope_None;
    when '10' =>
        types = MBReqTypes_Writes;
        scope = MBMaintenanceScope_None;
    when '11' =>
        types = MBReqTypes_All;
        case CRm[3:2] of
            when '00' => scope = MBMaintenanceScope_OuterShareable;
            when '01' => scope = MBMaintenanceScope_Nonshareable;
            when '10' => scope = MBMaintenanceScope_InnerShareable;
            when '11' => scope = MBMaintenanceScope_OuterShareable;
        end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Memory nXS barrier

(FEAT_XS)

31 29	28 26	25 12	11 10	9 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	imm2	1 0	0 0 1	1 1 1 1 1
					op2	Rt

Encoding

```
DSB  <option>nXS
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_XS) then EndOfDecode(Decode_UNDEF); end;
let types : MBReqTypes = MBReqTypes_All;
var nXS : boolean = TRUE;
let alias : DSBAlias = DSBAlias_DSB;
var scope : MBMaintenanceScope;
case imm2 of
    when '00' => scope = MBMaintenanceScope_OuterShareable;
    when '01' => scope = MBMaintenanceScope_Nonshareable;
    when '10' => scope = MBMaintenanceScope_InnerShareable;
    when '11' => scope = MBMaintenanceScope_OuterShareable;
end;
```

Assembler Symbols

<option>

For the “Memory barrier” variant: specifies the limitation on the Memory Effects and maintenance operations ordered by the barrier operation, and is encoded in “CRm”.

Note

The value `0b0000` is used to encode `SSBB` and the value `0b0100` is used to encode `PSSBB`.

For more information on whether an access is before or after a barrier instruction, see Data Synchronization Barrier (DSB).

CRm	<option>	Description
0001	OSHLD	This option has the same behavior as LD and is deprecated.
0010	OSHST	This option has the same behavior as ST and is deprecated.
0011	OSH	This option has the same behavior as SY.
0101	NSHLD	This option has the same behavior as LD and is deprecated.
0110	NSHST	This option has the same behavior as ST and is deprecated.
0111	NSH	Non-shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.
1001	ISHLD	This option has the same behavior as LD and is deprecated.
1010	ISHST	This option has the same behavior as ST and is deprecated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	<option>	Description
1011	ISH	Inner Shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.
1x00	RESERVED	•
1101	LD	Reads are the required access type before the barrier instruction. Reads and writes are the required access types after the barrier instruction.
1110	ST	Writes are the required access type, both before and after the barrier instruction.
1111	SY	Outer Shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.

All encodings of “CRm” that are listed as reserved can be encoded using the #<imm> syntax. All reserved options have the same behavior as SY, but software must not rely on this behavior.

For the “Memory nXS barrier” variant: specifies the limitation on the barrier operation, and is encoded in “imm2”:

imm2	<option>	Description
00	OSH	This option has the same behavior as SY.
01	NSH	Non-shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.
10	ISH	Inner Shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.
11	SY	Outer Shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Alias Conditions

Alias	Is preferred when
PSSBB	CRm == '0100'
SSBB	CRm == '0000'

Operation

```
case alias of
    when DSBAlias_SSBB =>
        SpeculativeStoreBypassBarrierToVA();
    when DSBAlias_PSSBB =>
        SpeculativeStoreBypassBarrierToPA();
    when DSBAlias_DSB =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !nXS && IsFeatureImplemented(FEAT_XS) then
            nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2().FnXS == '1';
        end;
        DataSynchronizationBarrier(scope, types, nXS);
    otherwise =>
        unreachable;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.153 DVP

Data value prediction restriction by context

This instruction prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see DVP RCTX, Data Value Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 1	1 0 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
DVP  RCTX, <Xt>
```

is equivalent to

```
SYS  #3, C7, C3, #5, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.154 EON (shifted register)

Bitwise exclusive-OR NOT (shifted register)

This instruction performs a bitwise exclusive-OR NOT between a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 0 (opc)	0 1 0 1 0	shift	1 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
EON  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
EON  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

X{datasize}(d) = operand1 XOR NOT(operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.155 EOR (immediate)

Bitwise exclusive-OR (immediate)

This instruction performs a bitwise exclusive-OR between a register value and an immediate value, and writes the result to the destination register.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
EOR  <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
EOR  <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
var imm : bits(datasize);
(imm, -) = DecodeBitMasks{datasize}(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is the bitmask immediate. For encoding information, see Logical (immediate).

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = imm;

let result : bits(datasize) = operand1 XOR operand2;

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.156 EOR (shifted register)

Bitwise exclusive-OR (shifted register)

This instruction performs a bitwise exclusive-OR between a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 0	0 1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc			N				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
EOR  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
EOR  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

X{datasize}(d) = operand1 XOR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.157 ERET

Exception return

This instruction restores PSTATE from the SPSR, and branches to the address held in the ELR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

ERET is UNDEFINED at EL0.

31–29	28–26	25	24–21	20–16	15–12	11	10	9–5	4–0
1 1 0	1 0 1	1	0 1 0 0 (opc)	1 1 1 1 1 (op2)	0 0 0 0	0 (A)	0 (M)	1 1 1 1 1 (Rn)	0 0 0 0 0 (op4)

Encoding

```
ERET
```

Decode for this encoding

```
let pac : boolean = FALSE;
let use_key_a : boolean = TRUE;
```

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;
AArch64_CheckForERetTrap(pac, use_key_a);
let target : bits(64) = ELR_ELx();

AArch64_ExceptionReturn(target, SPSR_ELx());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.158 ERETAA, ERETAB

Exception return, with pointer authentication

This instruction authenticates the address in ELR, using SP as the modifier and the specified key, restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for `ERETAA`. Key B is used for `ERETAB`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to ELR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

`ERETAA` and `ERETAB` are UNDEFINED at EL0.

Integer

(FEAT_PAuth)

31–25	24–21	20–16	15–12	11	10	9–5	4–0
1 1 0 1 0 1 1	0 1 0 0	1 1 1 1 1	0 0 0 0	1	M	1 1 1 1 1	1 1 1 1 1
	opc	op2		A		Rn	op4

Encoding for the ERETAA variant

Applies when `(M == 0)`

```
ERETAA
```

Encoding for the ERETAB variant

Applies when `(M == 1)`

```
ERETAB
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;

let pac : boolean = TRUE;
let use_key_a : boolean = (M == '0');
let auth_then_branch : boolean = TRUE;
```

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;
AArch64_CheckForERetTrap(pac, use_key_a);
var target : bits(64) = ELR_ELx();
let modifier : bits(64) = SP{}();

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);
end;

AArch64_ExceptionReturn(target, SPSR_ELx());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.159 ESB

Error synchronization barrier

This instruction is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. For more information, see RAS PE architecture and Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100).

If FEAT_RAS is not implemented, this instruction executes as a `NOP`.

System

(FEAT_RAS)

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 0 0 (op2)	1 1 1 1 1

Encoding

```
ESB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then EndOfDecode(Decode_NOP); end;
```

Operation

```
SynchronizeErrors();
AArch64_ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    AArch64_vESBOperation();
elsif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 then
    AArch64_dESBOperation();
end;
TakeUnmaskedSErrorInterrupts();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.160 EXTR

Extract register

This instruction extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate).

31	30 29	28 23	22	21	20 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 1	N	0	Rm	imms	Rn	Rd
	op21			o0				

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 0xxxxx)`

```
EXTR  <Wd>, <Wn>, <Wm>, #<lsb>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
EXTR  <Xd>, <Xn>, <Xm>, #<lsb>
```

Decode for all variants of this encoding

```
if N != sf then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && imms[5] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let lsb : integer{} = UInt(imms);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<lsb>

For the “32-bit” variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
`ROR (immediate)`	`Rn == Rm`

Operation

```
var result : bits(datasize);
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let concat : bits(2*datasize) = operand1::operand2;

result = concat[(lsb+datasize)-1:lsb];

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.161 GCSB

Guarded Control Stack barrier

This instruction generates a GCSB effect.

If FEAT_GCS is not implemented, this instruction executes as a `NOP`.

System

(FEAT_GCS)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 1 1 (op2)	1 1 1 1 1

Encoding

```
GCSB  DSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_NOP); end;
```

Operation

```
GCSSynchronizationBarrier();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.162 GCSPOPCX

Guarded Control Stack pop and compare exception return record

This instruction loads an exception return record from the location indicated by the current GCS pointer register, compares the loaded values with the current ELR_ELx, SPSR_ELx, and LR, and increments the pointer by the size of a GCS exception return record.

For more information, see GCSPOPCX, Guarded Control Stack Pop and Compare exception return record.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

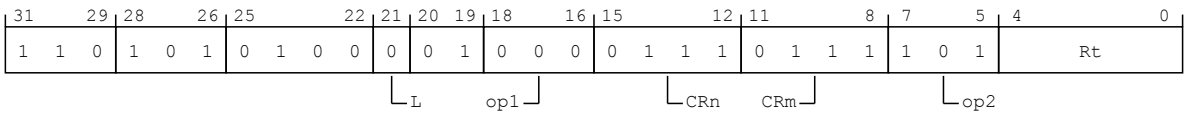

Encoding

```
GCSPOPCX
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is always the preferred disassembly.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.163 GCSPOPM

Guarded Control Stack pop

This instruction loads the 64-bit doubleword that is pointed to by the current GCS pointer register, writes it to the destination register, and increments the current GCS pointer register by the size of a GCS procedure return record.

For more information, see GCSPOPM, Guarded Control Stack Pop.

This is an alias of SYSL. This means:

- The encodings in this description are named to match the encodings of SYSL.
- The description of SYSL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	1	0 1	0 1 1	0 1 1 1	0 1 1 1	0 0 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSPOPM     {<Xt>}
```

is equivalent to

```
SYSL  <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the optional general-purpose destination register, encoded in the “Rt” field. Defaults to XZR if absent.

Operation

The description of SYSL gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.164 GCSPOPX

Guarded Control Stack pop exception return record

This instruction loads an exception return record from the location indicated by the current GCS pointer register, checks that the record is an exception return record, and increments the pointer by the size of a Guarded Control Stack exception return record.

For more information, see GCSPOPX, Guarded Control Stack Pop exception return record.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

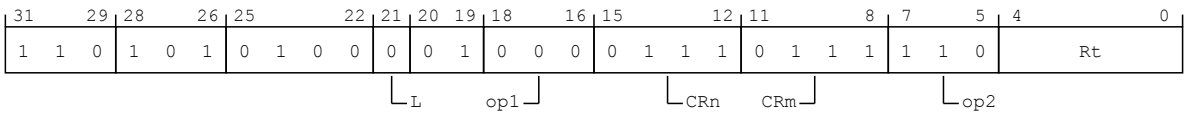

Encoding

```
GCSPOPX
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is always the preferred disassembly.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.165 GCSPUSHM

Guarded Control Stack push

This instruction decrements the current GCS pointer register by the size of a GCS procedure return record and stores an entry to the Guarded Control Stack.

For more information, see GCSPUSHM, Guarded Control Stack Push.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

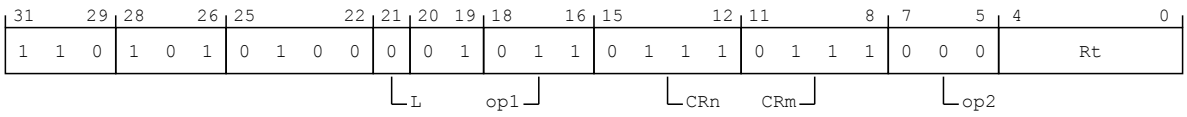

Encoding

```
GCSPUSHM  <Xt>
```

is equivalent to

```
SYS  #3, C7, C7, #0, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.166 GCSPUSHX

Guarded Control Stack push exception return record

This instruction decrements the current GCS pointer register by the size of a Guarded Control Stack exception return record and stores an exception return record to the Guarded Control Stack.

For more information, see GCSPUSHX, Guarded Control Stack Push exception return record.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 0 0	0 1 1 1	0 1 1 1	1 0 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSPUSHX
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is always the preferred disassembly.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.167 GCSSS1

Guarded Control Stack switch stack 1

This instruction validates that the stack being switched to contains a Valid GCS cap record, stores an In-progress GCS cap record to the stack that is being switched to, and sets the current GCS pointer register to the stack that is being switched to.

If the instruction generates a synchronous Data Abort exception, Watchpoint exception, GPC exception, or GCS Data Check exception, the value of GCSPR_ELx for the current Exception level is restored to the value held in the register before the instruction was executed.

For more information, see GCSSS1, Guarded Control Stack Switch Stack 1.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 1 1 1	0 1 0	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSSS1  <Xt>
```

is equivalent to

```
SYS  #3, C7, C7, #2, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.168 GCSSS2

Guarded Control Stack switch stack 2

This instruction validates that the most recent entry of the Guarded Control Stack being switched to contains an In-progress GCS cap record, stores a Valid GCS cap record to the Guarded Control Stack that is being switched from, and sets Xt to the Guarded Control Stack pointer that is being switched from.

For more information, see GCSSS2, Guarded Control Stack Switch Stack 2.

This is an alias of SYSL. This means:

- The encodings in this description are named to match the encodings of SYSL.
- The description of SYSL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	1	0 1	0 1 1	0 1 1 1	0 1 1 1	0 1 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
GCSSS2  <Xt>
```

is equivalent to

```
SYSL  <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

Operation

The description of SYSL gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.169 GCSSTR

Guarded Control Stack store register

This instruction stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Integer

(FEAT_GCS)

31			28	27	26	25	24									15	14		12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	0	0	1	1	1	1	1	0	0	0	0	1	1	Rn		Rt	
																	opc								

Encoding

```
GCSSTR  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let effective_el : bits(2) = PSTATE.EL;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();
end;

let privileged : boolean = effective_el != EL0;
let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_STORE, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

Mem{64}(address, accdesc) = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.170 GCSSTTR

Guarded Control Stack store register (unprivileged)

This instruction stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1 and the Effective value of HCR_EL2.{NV1, NV} is not {1, 1}.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_GCS)

31			28	27	26	25	24									15	14		12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	0	0	1	1	1	1	1	0	0	0	1	1	1	Rn		Rt	
																	opc								

Encoding

```
GCSSTTR  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
var address : bits(64);

let effective_el : bits(2) = if AArch64_IsUnprivAccessPriv() then PSTATE.EL else EL0;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();
end;

let privileged : boolean = effective_el != EL0;
let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_STORE, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = X{64}(n);
end;

Mem{64}(address, accdesc) = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.171 GMI

Tag mask insert

This instruction inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	0	1	1 0 1	0 1 1 0	Rm	0 0 0 1 0 1	Rn	Rd
sf		S					opcode		

Encoding

```
GMI  <Xd>, <Xn|SP>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
let address : bits(64) = if n == 31 then SP{64}() else X{64}(n);
var mask : bits(64) = X{}(m);
let tag : bits(4) = AArch64_AllocationTagFromAddress(address);

mask[UInt(tag)] = '1';
X{64}(d) = mask;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.172 HINT

Hint instruction

This instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	CRm	op2	1 1 1 1 1

Encoding

```
HINT  #<imm>
```

Decode for this encoding

```
var op : SystemHintOp;

var stream : boolean;
case CRm::op2 of
    when '0000 000' => op = SystemHintOp_NOP;
    when '0000 001' => op = SystemHintOp_YIELD;
    when '0000 010' => op = SystemHintOp_WFE;
    when '0000 011' => op = SystemHintOp_WFI;
    when '0000 100' => op = SystemHintOp_SEV;
    when '0000 101' => op = SystemHintOp_SEVL;
    when '0000 110' =>
        if !IsFeatureImplemented(FEAT_DGH) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_DGH;
    when '0000 111' => See("XPACLRI");
    when '0001 xxx' =>
        case op2 of
            when '000' => See("PACIA1716");
            when '010' => See("PACIB1716");
            when '100' => See("AUTIA1716");
            when '110' => See("AUTIB1716");
            otherwise => EndOfDecode(Decode_NOP);
        end;
    when '0010 000' =>
        if !IsFeatureImplemented(FEAT_RAS) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_ESB;
    when '0010 001' =>
        if !IsFeatureImplemented(FEAT_SPE) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_PSB;
    when '0010 010' =>
        if !IsFeatureImplemented(FEAT_TRF) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_TSB;
    when '0010 011' =>
        if !IsFeatureImplemented(FEAT_GCS) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_GCSB;
    when '0010 100' =>
        op = SystemHintOp_CSDB;
    when '0010 110' =>
        if !IsFeatureImplemented(FEAT_CLRBHB) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_CLRBHB;
    when '0011 xxx' =>
        case op2 of
            when '000' => See("PACIAZ");
            when '001' => See("PACIASP");
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '010' => See("PACIBZ");
            when '011' => See("PACIBSP");
            when '100' => See("AUTIAZ");
            when '101' => See("AUTIASP");
            when '110' => See("AUTIBZ");
            when '111' => See("AUTIBSP");
        end;
    when '0100 xx0' =>
        if !IsFeatureImplemented(FEAT_BTI) then EndOfDecode(Decode_NOP); end;

        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2[2:1]));
        op = SystemHintOp_BTI;
    when '0100 111' => See("PACM");
    when '0101 000' =>
        if !IsFeatureImplemented(FEAT_CHK) then EndOfDecode(Decode_NOP); end;
        op = SystemHintOp_CHKFEAT;
    when '0110 00x' =>
        if !IsFeatureImplemented(FEAT_PCDPHINT) then EndOfDecode(Decode_NOP); end;
        stream = op2[0] == '1';
        op = SystemHintOp_STSHH;
    otherwise => EndOfDecode(Decode_NOP);
end;
```

Assembler Symbols

<imm>

Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in the “CRm:op2” field.

The encodings that are allocated to architectural hint functionality are described in the ‘Hints’ table in the ‘Index by Encoding’.

Note

For allocated encodings of “CRm:op2”:

- A disassembler will disassemble the allocated instruction, rather than the `HINT` instruction.
- An assembler may support assembly of allocated encodings using `HINT` with the corresponding `<imm>` value, but it is not required to do so.

Operation

```
case op of
    when SystemHintOp_YIELD =>
        Hint_Yield();

    when SystemHintOp_DGH =>
        Hint_DGH();

    when SystemHintOp_WFE =>
        Hint_WFE();

    when SystemHintOp_WFI =>
        Hint_WFI();

    when SystemHintOp_SEV =>
        SendEvent();

    when SystemHintOp_SEVL =>
        SendEventLocal();

    when SystemHintOp_ESB =>
        SynchronizeErrors();
        AArch64_ESBOperation();
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_vESBOperation();
        elsif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 then
            AArch64_dESBOperation();
        end;
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSB =>
        if IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_SPEv1p5) then
            let trap_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                         !IsInHost() &&
                                         (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
                                         HFGITR_EL2().PSBCSYNC == '1');
            if trap_to_el2 then
                let target_el : bits(2) = EL2;
                let iss : bits(25)      = 0x3[24:0];
                AArch64_OtherInstrTrap(target_el, iss);
            end;
        end;

        ProfilingSynchronizationBarrier();

    when SystemHintOp_TSB =>
        if IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_TRBEv1p1) then
            let trap_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                         !IsInHost() &&
                                         (!HaveEL(EL3) || SCR_EL3().FGTEn2 == '1') &&
                                         HFGITR2_EL2().TSBCSYNC == '1');
            if trap_to_el2 then
                let target_el : bits(2) = EL2;
                let iss : bits(25)      = 0x4[24:0];
                AArch64_OtherInstrTrap(target_el, iss);
            end;
        end;

        TraceSynchronizationBarrier();

    when SystemHintOp_GCSB =>
        GCSSynchronizationBarrier();

    when SystemHintOp_CHKFEAT =>
        X{64}(16) = AArch64_ChkFeat(X{64}(16));

    when SystemHintOp_CSDB =>
        ConsumptionOfSpeculativeDataBarrier();

    when SystemHintOp_CLRBHB =>
        Hint_CLRBHB();

    when SystemHintOp_BTI =>
        SetBTypeNext('00');

    when SystemHintOp_STSHH =>
        Hint_StoreShared(stream);

    when SystemHintOp_NOP =>
        return; // Do nothing

    otherwise =>
        unreachable;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.173 HLT

Halt instruction

This instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

Within a guarded memory region, while PSTATE.BTYPE != `0b00`, an `HLT` instruction that would cause entry into Debug state will not generate a Branch Target exception and will cause entry into Debug state as normal. For more information, see PSTATE.BTYPE.

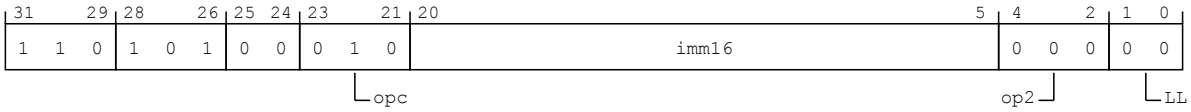

Encoding

```
HLT  #<imm>
```

Decode for this encoding

```
if EDSCR().HDE == '0' || !HaltingAllowed() then
    EndOfDecode(Decode_UNDEF);
elsif IsFeatureImplemented(FEAT_BTI) then
    SetBTypeCompatible(TRUE);
end;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
let is_async : boolean = FALSE;
let fault : FaultRecord = NoFault();
Halt(DebugHalt_HaltInstruction, is_async, fault);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.174 HVC

Hypervisor call

This instruction causes an exception to EL2. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The `HVC` instruction is UNDEFINED:

- When EL3 is implemented and SCR_EL3.HCE is set to 0.
- When EL3 is not implemented and HCR_EL2.HCD is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- At EL0.

On executing an `HVC` instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC syndrome value `0x16`, and the value of the immediate argument.

31 29	28 26	25 24	23 21	20 5	4 2	1 0
1 1 0	1 0 1	0 0	0 0 0	imm16	0 0 0	1 0
			opc		op2	LL

Encoding

```
HVC  #<imm>
```

Decode for this encoding

```
if !HaveEL(EL2) then EndOfDecode(Decode_UNDEF); end;
let imm : bits(16) = imm16;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;
if PSTATE.EL == EL1 && !EL2Enabled() then Undefined(); end;
if !HaveEL(EL3) && HCR_EL2().HCD == '1' then Undefined(); end;
if HaveEL(EL3) && SCR_EL3().HCE == '0' then Undefined(); end;

AArch64_CallHypervisor(imm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.175 IC

Instruction cache operation

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

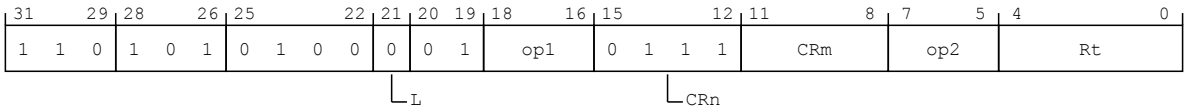

Encoding

```
IC  <ic_op>{, <Xt>}
```

is equivalent to

```
SYS  #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_IC`.

Assembler Symbols

<ic_op>

Is an IC operation name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

op1	CRm	op2	<ic_op>	Description
000	0001	000	IALLUIS	For more information, see IC IALLUIS.
000	0101	000	IALLU	For more information, see IC IALLU.
011	0101	001	IVAU	For more information, see IC IVAU.

<Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.176 IRG

Insert random tag

This instruction inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

Integer

(FEAT_MTE)

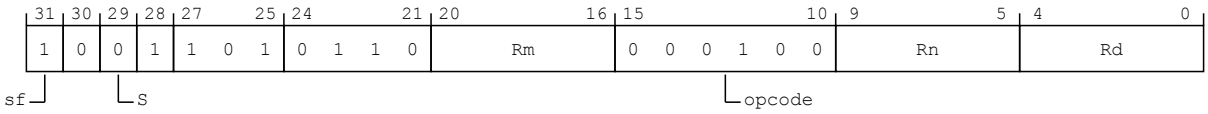

Encoding

```
IRG  <Xd|SP>, <Xn|SP>{, <Xm>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field. Defaults to XZR if absent.

Operation

```
let operand : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let exclude_reg : bits(64) = X{}(m);
let exclude : bits(16) = exclude_reg[15:0] OR GCR_EL1().Exclude;

let rtag : bits(4) = AArch64_ChooseTagOrZero(exclude);
let result : bits(64) = AArch64_AddressWithAllocationTag(operand, rtag);

if d == 31 then
    SP{64}() = result;
else
    X{64}(d) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.177 ISB

Instruction synchronization barrier

This instruction flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).

31 29	28 26	25 12	11 8	7	6 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	CRm	1	1 0	1 1 1 1 1
					opc	Rt

Encoding

```
ISB  {<option>|#<imm>}
```

Decode for this encoding

```
// no additional decoding required
```

Assembler Symbols

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY Full system barrier operation, encoded as CRm = `0b1111`. Can be omitted.

All other encodings of "CRm" are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm>

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
InstructionSynchronizationBarrier();

if IsFeatureImplemented(FEAT_BRBE) && BRBEBranchOnISB() then
    BRBEISB();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.178 LD64B

Single-copy atomic 64-byte Load

This instruction derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers. The load starts at register `Xt`, with the data being read as `X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt = Data[511:0]`. The data is loaded atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `LD64B` also supports `ST64B`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64)

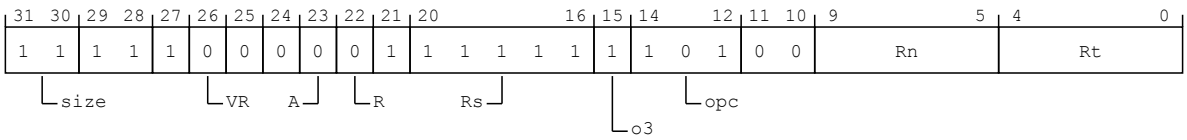

Encoding

```
LD64B  <Xt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64) then EndOfDecode(Decode_UNDEF); end;
if Rt[4:3] == '11' || Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let withstatus : boolean = FALSE;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let tagchecked : boolean = n != 31;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LD64B.

Assembler Symbols

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckLDST64BEnabled();

var data : bits(512);
var address : bits(64);
var value : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLS64(MemOp_LOAD, withstatus, tagchecked);

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP{64}();
else
    address = X{64}(n);
end;
data = MemLoad64B(address, accdesc);

for i = 0 to 7 do
    value = data[63+64*i : 64*i];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse{64}(value); end;
    X{64}(t+i) = value;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.179 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
  - `LDADDL` and `LDADDAL` store to memory with release semantics.
  - `LDADD` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADD, STADDL.

Integer

(FEAT_LSE)

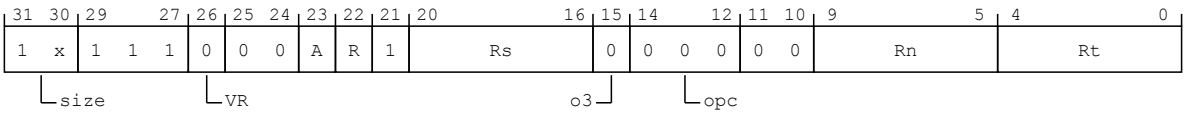

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDADD  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDADDA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDADDAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDADDL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDADD  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDADDA  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDADDAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDADDL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STADD, STADDL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.180 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte

This instruction atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAB` and `LDADDALB` load from memory with acquire semantics.
  - `LDADDLB` and `LDADDALB` store to memory with release semantics.
  - `LDADDB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADDB, STADDLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDADDB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDADDAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDADDALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDADDLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STADDB, STADDLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.181 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword

This instruction atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
  - `LDADDLH` and `LDADDALH` store to memory with release semantics.
  - `LDADDH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADDH, STADDLH.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDADDH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDADDAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDADDALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDADDLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STADDH, STADDLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.182 LDAPR

Load-acquire RCpc register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPR` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

It has encodings from 2 classes: Post-index and No offset

Post-index

(FEAT_LRCPC3)

31	30	29						23	22	21										10	9			5	4			0
1	x	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn			Rt			
size									L																			

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAPR  <Wt>, [<Xn|SP>], #4
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAPR  <Xt>, [<Xn|SP>], #8
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
var wback : boolean = TRUE;
let acquirepc : boolean = TRUE;
let regsize : integer{} = if size == '11' then 64 else 32;
let datasize : integer{} = 8 << UInt(size);
let offset : integer = 1 << UInt(size);

let tagchecked : boolean = TRUE;

var wb_unknown : boolean = FALSE;

if n == t && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset

(FEAT_LRCPC)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	1	0	1	(1)(1)(1)(1)(1)		1	1	0 0	0	0	Rn		Rt	
size				VR		A		R		Rs		o3	opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAPR  <Wt>, [<Xn|SP> {, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAPR  <Xt>, [<Xn|SP> {, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let wback : boolean = FALSE;
let offset : integer = 0;
let wb_unknown : boolean = FALSE;
let elsize : integer{} = 8 << UInt(size);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let datasize : integer{} = elsize;
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
var address : bits(64);
var data : bits(datasize);
let dbytes : integer{} = datasize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = X{64}(n);
end;

data = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    else
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.183 LDAPRB

Load-acquire RCpc register byte

This instruction derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPRB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC)

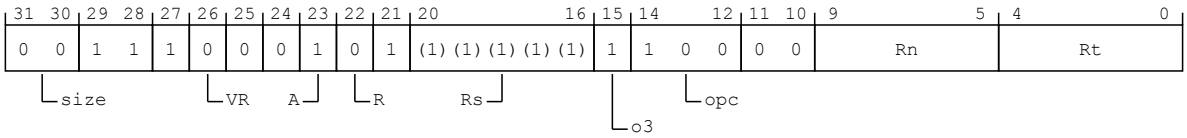

Encoding

```
LDAPRB  <Wt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(8);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

data = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.184 LDAPRH

Load-acquire RCpc register halfword

This instruction derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPRH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	1	0	1	(1)(1)(1)(1)(1)	1	1 0 0	0	0	Rn	Rt
size					VR		A		R		Rs	o3	opc				

Encoding

```
LDAPRH  <Wt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(16);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

data = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.185 LDAPUR

Load-acquire RCpc register (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPUR` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29					24	23	22	21	20						12	11	10	9				5	4				0
1	x	0	1	1	0	0	1	0	1	0	imm9							0	0	Rn					Rt				
size								opc																					

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAPUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAPUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let regsize : integer{} = if datasize == 64 then 64 else 32;
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.186 LDAPURB

Load-acquire RCpc register byte (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPURB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	0	1	1	0	0	1	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
LDAPURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8;
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.187 LDAPURH

Load-acquire RCpc register halfword (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPURH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

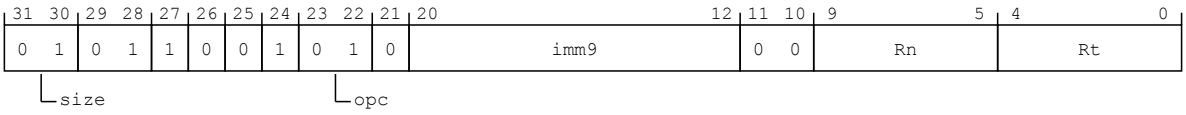

Encoding

```
LDAPURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 16;
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.188 LDAPURSB

Load-acquire RCpc register signed byte (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPURSB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31 30	29 … 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	0 1 1 0 0 1	1 x	0	imm9	0 0	Rn	Rt
size		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDAPURSB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDAPURSB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8;
let regsize : integer{} = 64 >> UInt(opc[0]);
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.189 LDAPURSH

Load-acquire RCpc register signed halfword (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPURSH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31 30	29 … 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	0 1 1 0 0 1	1 x	0	imm9	0 0	Rn	Rt
size		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDAPURSH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDAPURSH  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 16;
let regsize : integer{} = 64 >> UInt(opc[0]);
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.190 LDAPURSW

Load-acquire RCpc register signed word (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAPURSW` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	0	0	1	1	0	0	1	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
LDAPURSW  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32;
let acquirepc : boolean = TRUE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{64}(t) = SignExtend{64}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.191 LDAR

Load-acquire register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDAR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

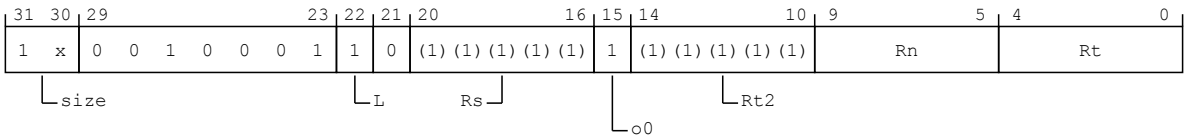

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let elsize : integer{} = 8 << UInt(size);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let acquire : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
var address : bits(64);
let dbytes : integer{} = elsize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(elsize) = Mem{elsize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.192 LDARB

Load-acquire register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDARB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

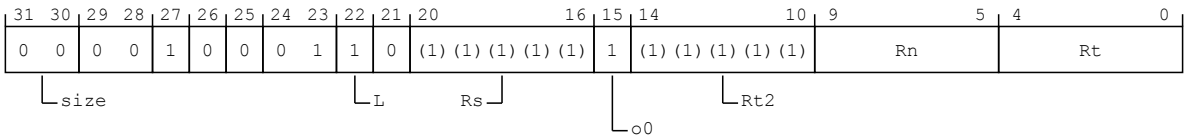

Encoding

```
LDARB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(8) = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.193 LDARH

Load-acquire register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDARH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

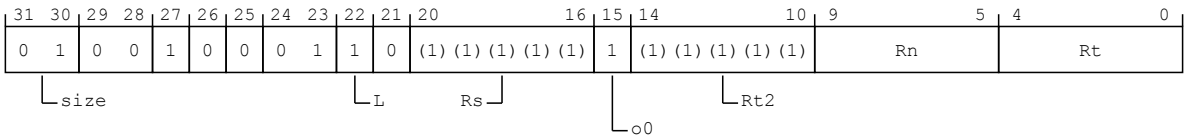

Encoding

```
LDARH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(16) = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.194 LDATXR

Load-acquire unprivileged exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `LDATXR` is equivalent to `LDXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

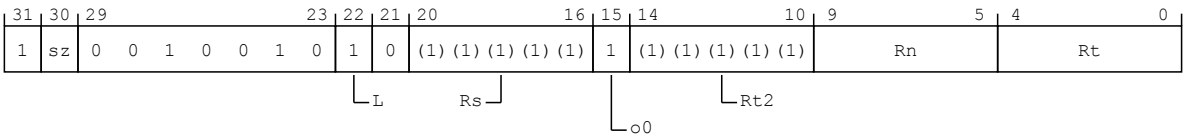

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDATXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDATXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem{elsize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.195 LDAXP

Load-acquire exclusive pair of registers

This instruction derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination registers are not both `WZR` or not both `XZR`, `LDAXP` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20					16	15	14				10	9					5	4					0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)		1	Rt2					Rn						Rt					
									L		Rs						o0																	

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDAXP  <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDAXP  <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let datasize : integer{} = elsize * 2;
let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDAXP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
var address : bits(64);
var data : bits(datasize);

let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = elsize == 64;   // When elsize is 32, the access is single-copy atomic
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem{datasize}(address, accdesc);
if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X{datasize}(t)  = ARBITRARY : bits(datasize);        // In this case t = t2
elsif BigEndian(accdesc.acctype) then
    X{elsize}(t)  = data[elsize +: elsize];
    X{elsize}(t2) = data[0 +: elsize];
else
    X{elsize}(t)  = data[0 +: elsize];
    X{elsize}(t2) = data[elsize +: elsize];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.196 LDAXR

Load-acquire exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination register is not one of `WZR` or `XZR`, `LDAXR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

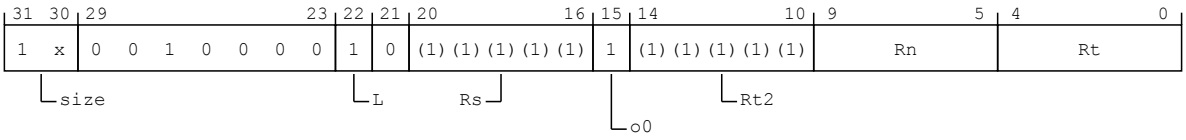

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDAXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDAXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 8 << UInt(size);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem{elsize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.197 LDAXRB

Load-acquire exclusive register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination register is not one of `WZR` or `XZR`, `LDAXRB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

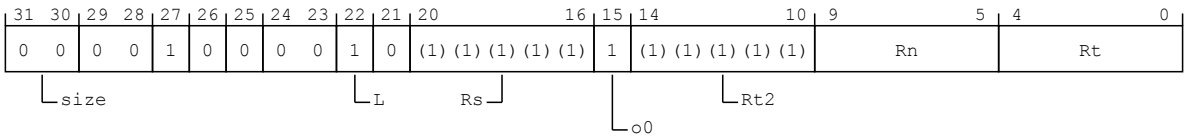

Encoding

```
LDAXRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(8);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, 1, accdesc);

data = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.198 LDAXRH

Load-acquire exclusive register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination register is not one of `WZR` or `XZR`, `LDAXRH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

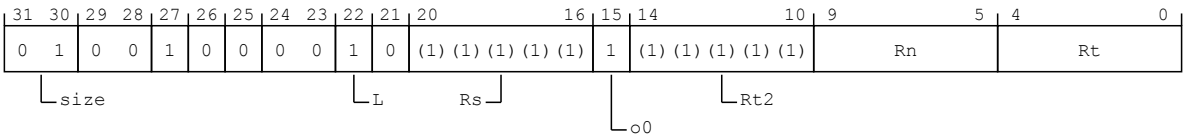

Encoding

```
LDAXRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(16);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, 2, accdesc);

data = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.199 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
  - `LDCLRL` and `LDCLRAL` store to memory with release semantics.
  - `LDCLR` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLR, STCLRL.

Integer

(FEAT_LSE)

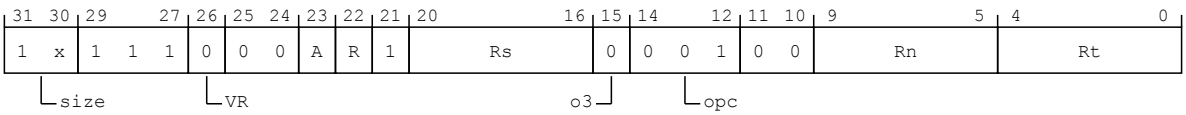

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDCLR  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDCLRA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDCLRAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDCLRL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDCLR  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STCLR, STCLRL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.200 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte

This instruction atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAB` and `LDCLRALB` load from memory with acquire semantics.
  - `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
  - `LDCLRB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLRB, STCLRLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDCLRB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDCLRAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDCLRALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDCLRLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STCLRB, STCLRLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.201 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
  - `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
  - `LDCLRH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLRH, STCLRLH.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDCLRH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDCLRAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDCLRALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDCLRLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STCLRH, STCLRLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.202 LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL

Atomic bit clear on quadword

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- `LDCLRPA` and `LDCLRPAL` load from memory with acquire semantics.
  - `LDCLRPL` and `LDCLRPAL` store to memory with release semantics.
  - `LDCLRP` has neither acquire nor release semantics.

Integer

(FEAT_LSE128)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rt2				0	0	0	1		0	0	Rn					Rt		
	S														o3		opc												

Encoding for the LDCLRP variant

Applies when `(A == 0 && R == 0)`

```
LDCLRP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDCLRPA variant

Applies when `(A == 1 && R == 0)`

```
LDCLRPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDCLRPAL variant

Applies when `(A == 1 && R == 1)`

```
LDCLRPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDCLRPL variant

Applies when `(A == 0 && R == 1)`

```
LDCLRPL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then EndOfDecode(Decode_UNDEF); end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
let value1 : bits(64) = X{}(t);
let value2 : bits(64) = X{}(t2);
var data : bits(128);
var store_value : bits(128);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                       tagchecked, privileged, t, t2, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let comparevalue : bits(128) = ARBITRARY : bits(128); // Irrelevant when not executing CAS
data = MemAtomic{128}(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t)  = data[127:64];
    X{64}(t2) = data[63:0];
else
    X{64}(t)  = data[63:0];
    X{64}(t2) = data[127:64];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.203 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive-OR on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
  - `LDEORL` and `LDEORAL` store to memory with release semantics.
  - `LDEOR` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEOR, STEORL.

Integer

(FEAT_LSE)

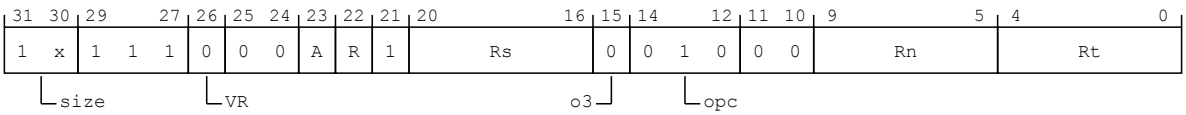

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDEOR  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDEORA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDEORAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDEORL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDEOR  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDEORA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDEORAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDEORL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STEOR, STEORL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.204 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive-OR on byte

This instruction atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAB` and `LDEORALB` load from memory with acquire semantics.
  - `LDEORLB` and `LDEORALB` store to memory with release semantics.
  - `LDEORB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEORB, STEORLB.

Integer

(FEAT_LSE)

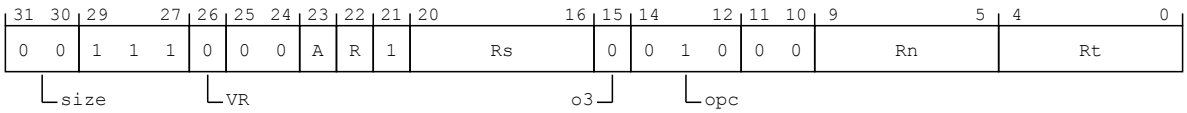

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDEORB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDEORAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDEORALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDEORLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STEORB, STEORLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.205 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive-OR on halfword

This instruction atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
  - `LDEORLH` and `LDEORALH` store to memory with release semantics.
  - `LDEORH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEORH, STEORLH.

Integer

(FEAT_LSE)

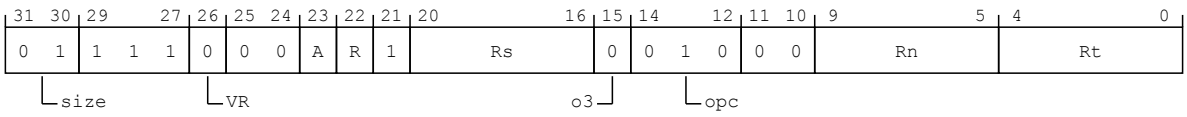

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDEORH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDEORAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDEORALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDEORLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STEORH, STEORLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.206 LDG

Load Allocation Tag

This instruction loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag Granule.

Integer

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	1	1	imm9		0	0	Rn		Rt	
						opc					op2					

Encoding

```
LDG  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);
var tag : bits(4);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let stzgm : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_LOAD, stzgm, t);

address = AddressAdd(address, offset, accdesc);
address = AlignDownSize(address, TAG_GRANULE);

tag = AArch64_MemTag(address, accdesc);
X{64}(t) = AArch64_AddressWithAllocationTag(X{64}(t), tag);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.207 LDGM

Load tag multiple

This instruction reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

31			28	27	26	25	24	23	22	21	20									12	11	10	9			5	4			0
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn				Rt			
								opc			imm9										op2									

Encoding

```
LDGM  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;

var data : bits(64) = Zeros{};
var address : bits(64);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let size : integer{} = 4 * (2 ^ (UInt(GMID_EL1().BS)));
address = AlignDownSize(address, size);
let count : integer = size >> LOG2_TAG_GRANULE;
var index : integer = UInt(address[LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE]);
let stzgm : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_LOAD, stzgm, t);

for i = 0 to count-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let tag : bits(4) = AArch64_MemTag(address, accdesc);
    data[index*:4] = tag;
    address = AddressIncrement(address, TAG_GRANULE, accdesc);
    index = index + 1;
end;

X{64}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.208 LDIAPP

Load-Acquire RCpc ordered pair of registers

This instruction calculates an address from a base register value and an optional offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- The Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.
- If the destination registers are not both `WZR` or not both `XZR`, `LDIAPP` loads from memory with Acquire semantics.
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC3)

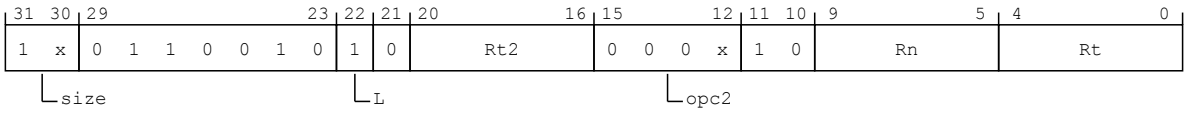

Encoding for the 32-bit post-index variant

Applies when `(size == 10 && opc2 == 0000)`

```
LDIAPP  <Wt1>, <Wt2>, [<Xn|SP>], #8
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc2 == 0001)`

```
LDIAPP  <Wt1>, <Wt2>, [<Xn|SP>]
```

Encoding for the 64-bit post-index variant

Applies when `(size == 11 && opc2 == 0000)`

```
LDIAPP  <Xt1>, <Xt2>, [<Xn|SP>], #16
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc2 == 0001)`

```
LDIAPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF); end;
let ispair : boolean = TRUE;
let postindex : boolean = opc2[0] == '0';
var wback : boolean = opc2[0] == '0';
```

`LDIAPP` has the same CONSTRAINED UNPREDICTABLE behavior as `LDP`. For information about this CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP and LDIAPP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let scale : integer{} = 2 + UInt(size[0]);
let datasize : integer{} = 8 << scale;
let offset : integer = if opc2[0] == '0' then (2 << scale) else 0;
let acqrel : boolean = TRUE;
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;

var wb_unknown : boolean = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;
let accdesc : AccessDescriptor = CreateAccDescLDAcqPC(tagchecked, ispair, acqrel, t, t2);

if n == 31 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

var full_data : bits(2*datasize);
full_data = Mem{2*datasize}(address, accdesc);
if BigEndian(accdesc.acctype) then
    data2 = full_data[(datasize-1):0];
    data1 = full_data[(2*datasize-1):datasize];
else
    data1 = full_data[(datasize-1):0];
    data2 = full_data[(2*datasize-1):datasize];
end;

if rt_unknown then
    data1 = ARBITRARY : bits(datasize);
    data2 = ARBITRARY : bits(datasize);
end;

X{datasize}(t) = data1;
X{datasize}(t2) = data2;

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.209 LDLAR

Load LOAcquire register

This instruction loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDLAR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load LOAcquire, Store LORelease and Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

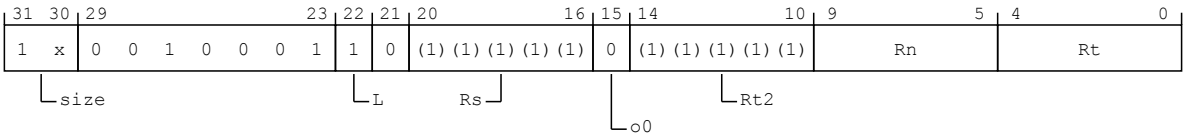

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDLAR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDLAR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let elsize : integer{} = 8 << UInt(size);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let acquire : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
let dbytes : integer{} = elsize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescLOR(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(elsize) = Mem{elsize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.210 LDLARB

Load LOAcquire register byte

This instruction loads a byte from memory, zero-extends it and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDLARB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load LOAcquire, Store LORelease and Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

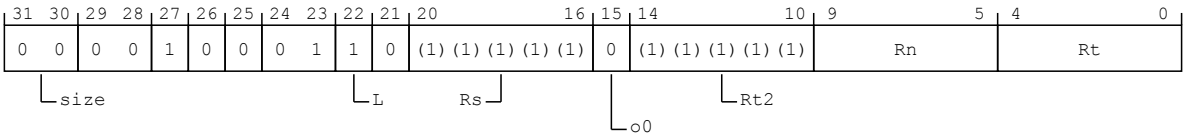

Encoding

```
LDLARB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLOR(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(8) = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.211 LDLARH

Load LOAcquire register halfword

This instruction loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of `WZR` or `XZR`, `LDLARH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load LOAcquire, Store LORelease and Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

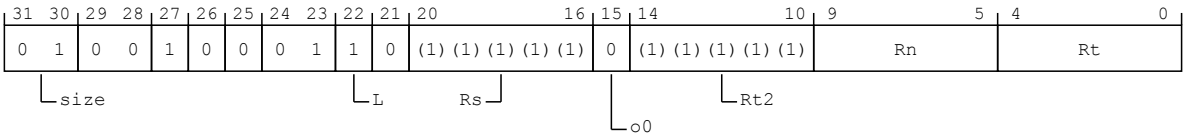

Encoding

```
LDLARH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLOR(MemOp_LOAD, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let data : bits(16) = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.212 LDNP

Load pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

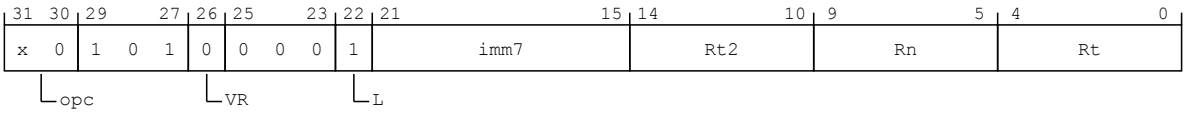

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDNP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
let data :  bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    X{datasize}(t) = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    X{datasize}(t2) = data[(datasize-1):0];
    X{datasize}(t)  = data[(2*datasize-1):datasize];
else
    X{datasize}(t)  = data[(datasize-1):0];
    X{datasize}(t2) = data[(2*datasize-1):datasize];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.213 LDP

Load pair of registers

This instruction calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

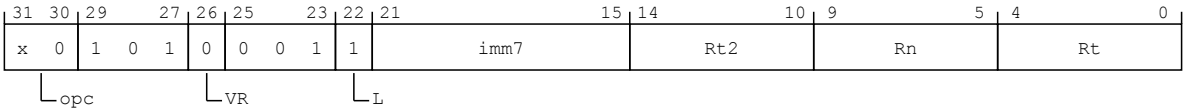

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

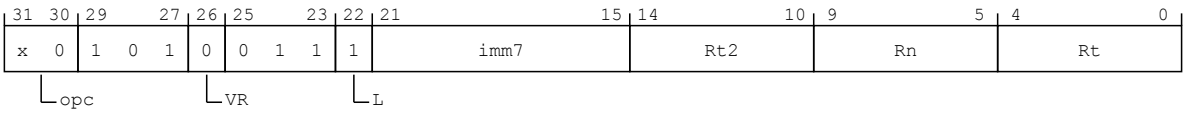

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

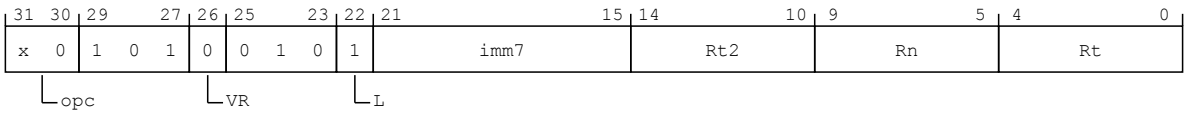

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;
var wb_unknown : boolean = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;        // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    data1 = ARBITRARY : bits(datasize);
    data2 = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    data2 = data[(datasize-1):0];
    data1 = data[(2*datasize-1):datasize];
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    data1 = data[(datasize-1):0];
    data2 = data[(2*datasize-1):datasize];
end;

X{datasize}(t) = data1;
X{datasize}(t2) = data2;

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.214 LDPSW

Load pair of registers signed word

This instruction calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

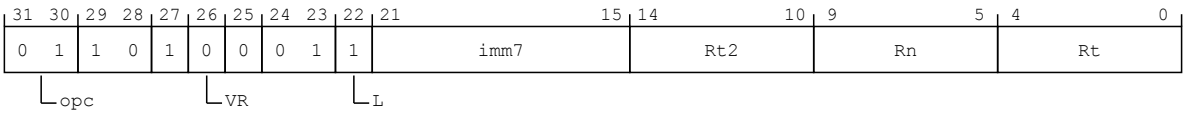

Encoding

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

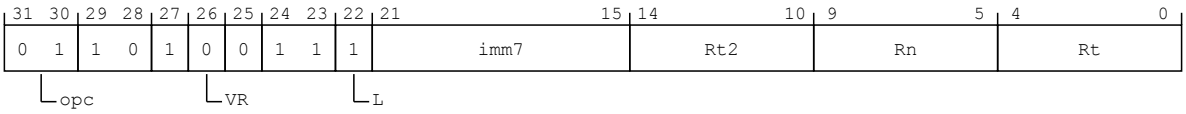

Encoding

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
```

Signed offset

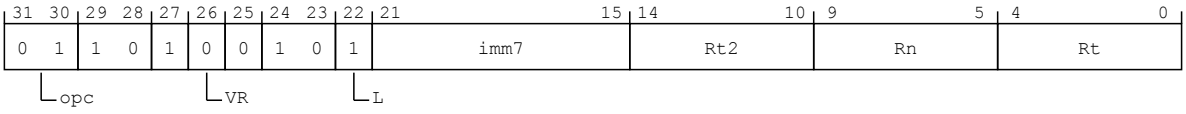

Encoding

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDPSW.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index” and “Pre-index” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Signed offset” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 32;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;
var wb_unknown : boolean = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;        // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    data1 = ARBITRARY : bits(datasize);
    data2 = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    data2 = data[(datasize-1):0];
    data1 = data[(2*datasize-1):datasize];
else
    data1 = data[(datasize-1):0];
    data2 = data[(2*datasize-1):datasize];
end;

X{64}(t) = SignExtend{64}(data1);
X{64}(t2) = SignExtend{64}(data2);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.215 LDR (immediate)

Load register (immediate)

This instruction loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	0	imm9	0 1	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	0	imm9	1 1	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31	30	29	27	26	25	24	23	22	21	10	9	5	4	0
1	x	1	1	1	0	0	1	0	1	imm12	Rn		Rt	
size				VR				opc						

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
let scale : integer{} = UInt(size);
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDR (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

For the “32-bit” variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

For the “64-bit” variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let regsize : integer{} = if datasize == 64 then 64 else 32;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.216 LDR (literal)

Load register (literal)

This instruction calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23 … 5	4 … 0
0	x	0	1 1	0	0	0	imm19	Rt
opc				VR				

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDR  <Wt>, <label>
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDR  <Xt>, <label>
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let size : integer{} = 4 << UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = FALSE;

let offset : bits(64) = SignExtend{}(imm19::'00');
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
let address : bits(64) = PC64() + offset;
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

X{size * 8}(t) = Mem{size * 8}(address, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.217 LDR (register)

Load register (register)

This instruction calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23	22	21	20	16	15	13	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	1	1	Rm	option		S	1	0	Rn		Rt	
size				VR				opc												

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDR  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDR  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let scale : integer{} = UInt(size);
let shift : integer{} = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 8 << scale;
let regsize : integer{} = if datasize == 64 then 64 else 32;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.218 LDRAA, LDRAB

Load register, with pointer authentication

This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for `LDRAA`. Key B is used for `LDRAB`.

If the authentication passes, the PE behaves the same as for an `LDR` instruction. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about addressing modes, see Load/Store addressing modes.

PAC

(FEAT_PAuth)

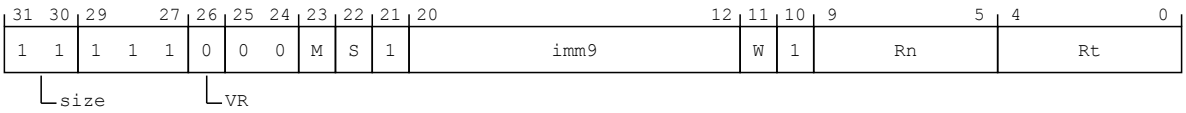

Encoding for the Key A, offset variant

Applies when `(M == 0 && W == 0)`

```
LDRAA  <Xt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the Key A, pre-indexed variant

Applies when `(M == 0 && W == 1)`

```
LDRAA  <Xt>, [<Xn|SP>{, #<simm>}]!
```

Encoding for the Key B, offset variant

Applies when `(M == 1 && W == 0)`

```
LDRAB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the Key B, pre-indexed variant

Applies when `(M == 1 && W == 1)`

```
LDRAB  <Xt>, [<Xn|SP>{, #<simm>}]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
var wback : boolean = W == '1';
let use_key_a : boolean = M == '0';
let S10 : bits(10) = S::imm9;
let offset : bits(64) = LSL(SignExtend{64}(S10), 3);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the “S:imm9” field as <simm>/8.

Operation

```
var address : bits(64);
let privileged : boolean = PSTATE.EL != EL0;
let auth_then_branch : boolean = TRUE;

let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);
if n == 31 then
    address = SP{64}();
else
    address = X{64}(n);
end;

if use_key_a then
    address = AuthDA(address, X{64}(31), auth_then_branch);
else
    address = AuthDB(address, X{64}(31), auth_then_branch);
end;

if n == 31 then
    CheckSPAlignment();
end;

address = AddressAdd(address, offset, accdesc);
X{64}(t) = Mem{64}(address, accdesc);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.219 LDRB (immediate)

Load register byte (immediate)

This instruction loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRB  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRB  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	0	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size					VR			opc				

Encoding

```
LDRB  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let data : bits(8) = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.220 LDRB (register)

Load register byte (register)

This instruction calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the Extended register variant

Applies when `(option != 011)`

```
LDRB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the Shifted register variant

Applies when `(option == 011)`

```
LDRB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(8) = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.221 LDRH (immediate)

Load register halfword (immediate)

This instruction loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRH  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
LDRH  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	1	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size					VR			opc				

Encoding

```
LDRH  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let data : bits(16) = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.222 LDRH (register)

Load register halfword (register)

This instruction calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15 … 13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
size					VR			opc									

Encoding

```
LDRH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(16) = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.223 LDRSB (immediate)

Load register signed byte (immediate)

This instruction loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 1	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSB  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSB  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	1 1	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSB  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSB  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
0 0	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSB  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSB  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSB (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(8) = Mem{8}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.224 LDRSB (register)

Load register signed byte (register)

This instruction calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit with extended register offset variant

Applies when `(opc == 11 && option != 011)`

```
LDRSB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 32-bit with shifted register offset variant

Applies when `(opc == 11 && option == 011)`

```
LDRSB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 64-bit with extended register offset variant

Applies when `(opc == 10 && option != 011)`

```
LDRSB  <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 64-bit with shifted register offset variant

Applies when `(opc == 10 && option == 011)`

```
LDRSB  <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<extend>

Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount>

Is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(8) = Mem{8}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.225 LDRSH (immediate)

Load register signed halfword (immediate)

This instruction loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

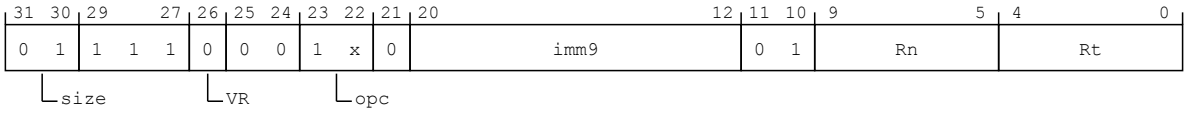

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

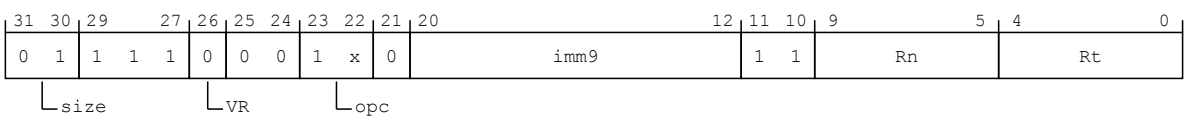

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
0 1	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(16) = Mem{16}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.226 LDRSH (register)

Load register signed halfword (register)

This instruction calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDRSH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDRSH  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
011	LSL
110	SXTW
111	SXTX

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(16) = Mem{16}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.227 LDRSW (immediate)

Load register signed word (immediate)

This instruction loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	0 1	Rn	Rt
size			VR			opc					

Encoding

```
LDRSW  <Xt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	1 1	Rn	Rt
size			VR			opc					

Encoding

```
LDRSW  <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31 30	29 28	27	26	25	24	23 22	21 … 10	9 … 5	4 … 0
1 0	1 1	1	0	0	1	1 0	imm12	Rn	Rt
size			VR			opc			

Encoding

```
LDRSW  <Xt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 2);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSW (immediate).

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var wb_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let data : bits(32) = Mem{32}(address, accdesc);
X{64}(t) = SignExtend{64}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.228 LDRSW (literal)

Load register signed word (literal)

This instruction calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23 … 5	4 … 0
1	0	0	1	1	0	0	0	imm19	Rt
opc					VR				

Encoding

```
LDRSW  <Xt>, <label>
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = FALSE;

let offset : bits(64) = SignExtend{}(imm19::'00');
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
let address : bits(64) = PC64() + offset;
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

let data : bits(32) = Mem{32}(address, accdesc);
X{64}(t) = SignExtend{64}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.229 LDRSW (register)

Load register signed word (register)

This instruction calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 0	1 1	1	0	0	0	1 0	1	Rm	option	S	1 0	Rn	Rt
size			VR			opc							

Encoding

```
LDRSW  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then 2 else 0;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(32) = Mem{32}(address, accdesc);
X{64}(t) = SignExtend{64}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.230 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
  - `LDSETL` and `LDSETAL` store to memory with release semantics.
  - `LDSET` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSET, STSETL.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	A	R	1	Rs		0	0	1 1	0	0	Rn		Rt	
size				VR								o3	opc							

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSET  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSETA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSETAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSETL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSET  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STSET, STSETL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.231 LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte

This instruction atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
  - `LDSETLB` and `LDSETALB` store to memory with release semantics.
  - `LDSETB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSETB, STSETLB.

Integer

(FEAT_LSE)

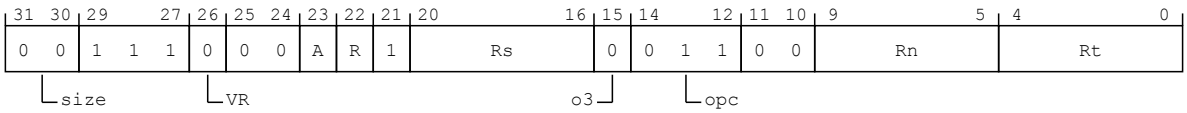

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSETB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSETAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSETALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSETLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSETB, STSETLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.232 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAH` and `LDSETALH` load from memory with acquire semantics.
  - `LDSETLH` and `LDSETALH` store to memory with release semantics.
  - `LDSETH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSETH, STSETLH.

Integer

(FEAT_LSE)

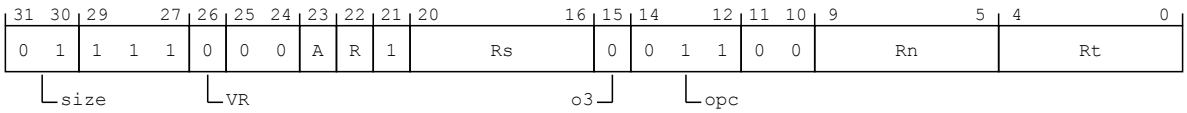

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSETH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSETAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSETALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSETLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSETH, STSETLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.233 LDSETP, LDSETPA, LDSETPAL, LDSETPL

Atomic bit set on quadword

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- `LDSETPA` and `LDSETPAL` load from memory with acquire semantics.
  - `LDSETPL` and `LDSETPAL` store to memory with release semantics.
  - `LDSETP` has neither acquire nor release semantics.

Integer

(FEAT_LSE128)

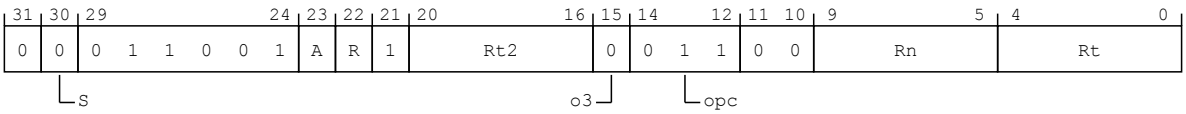

Encoding for the LDSETP variant

Applies when `(A == 0 && R == 0)`

```
LDSETP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDSETPA variant

Applies when `(A == 1 && R == 0)`

```
LDSETPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDSETPAL variant

Applies when `(A == 1 && R == 1)`

```
LDSETPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the LDSETPL variant

Applies when `(A == 0 && R == 1)`

```
LDSETPL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then EndOfDecode(Decode_UNDEF); end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
let value1 : bits(64) = X{}(t);
let value2 : bits(64) = X{}(t2);
var data : bits(128);
var store_value : bits(128);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                       tagchecked, privileged, t, t2, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let comparevalue : bits(128) = ARBITRARY : bits(128); // Irrelevant when not executing CAS
data = MemAtomic{128}(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t)  = data[127:64];
    X{64}(t2) = data[63:0];
else
    X{64}(t)  = data[63:0];
    X{64}(t2) = data[127:64];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.234 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
  - `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
  - `LDSMAX` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAX, STSMAXL.

Integer

(FEAT_LSE)

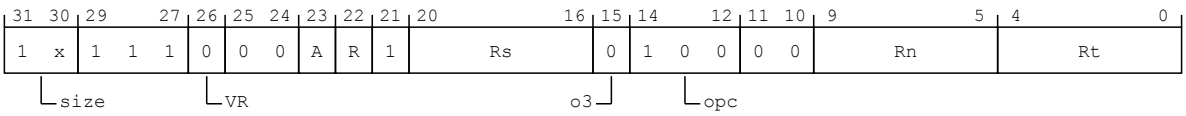

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSMAX  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSMAXA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSMAXAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSMAX  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSMAXA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSMAXAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STSMAX, STSMAXL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.235 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAB` and `LDSMAXALB` load from memory with acquire semantics.
  - `LDSMAXLB` and `LDSMAXALB` store to memory with release semantics.
  - `LDSMAXB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMAXAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMAXALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMAXB, STSMAXLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.236 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
  - `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
  - `LDSMAXH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMAXAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMAXALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMAXH, STSMAXLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.237 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
  - `LDSMINL` and `LDSMINAL` store to memory with release semantics.
  - `LDSMIN` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMIN, STSMINL.

Integer

(FEAT_LSE)

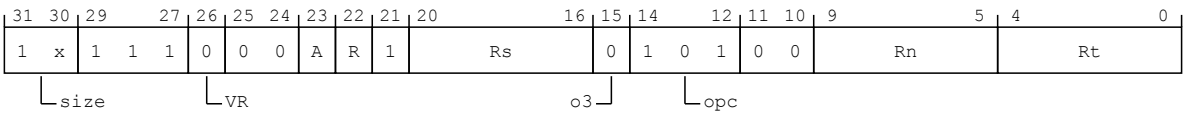

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSMIN  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSMINA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSMINAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSMINL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSMIN  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSMINA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSMINAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSMINL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STSMIN, STSMINL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.238 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAB` and `LDSMINALB` load from memory with acquire semantics.
  - `LDSMINLB` and `LDSMINALB` store to memory with release semantics.
  - `LDSMINB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMINB, STSMINLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMINB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMINAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMINALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMINB, STSMINLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.239 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
  - `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
  - `LDSMINH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMINH, STSMINLH.

Integer

(FEAT_LSE)

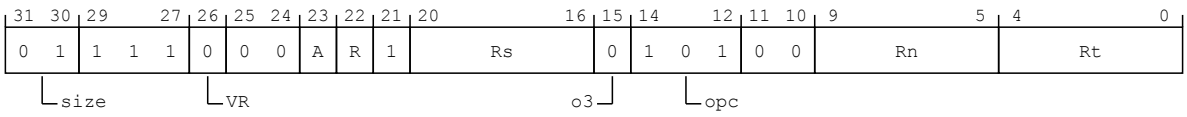

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDSMINH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDSMINAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDSMINALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDSMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STSMINH, STSMINLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.240 LDTADD, LDTADDA, LDTADDAL, LDTADDL

Atomic add unprivileged

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR`, or `XZR`, `LDTADDA` and `LDTADDAL` load from memory with acquire semantics.
  - `LDTADDL` and `LDTADDAL` store to memory with release semantics.
  - `LDTADD` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

This instruction is used by the alias STTADD, STTADDL.

Integer

(FEAT_LSUI)

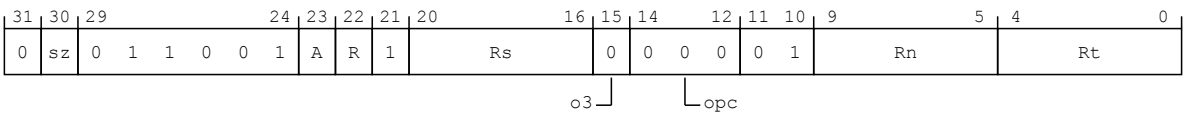

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
LDTADD  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
LDTADDA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
LDTADDAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
LDTADDL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
LDTADD  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
LDTADDA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
LDTADDAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
LDTADDL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 32 << UInt(sz);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STTADD, STTADDL	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.241 LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL

Atomic bit clear unprivileged

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR`, or `XZR`, `LDTCLRA` and `LDTCLRAL` load from memory with acquire semantics.
  - `LDTCLRL` and `LDTCLRAL` store to memory with release semantics.
  - `LDTCLR` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

This instruction is used by the alias STTCLR, STTCLRL.

Integer

(FEAT_LSUI)

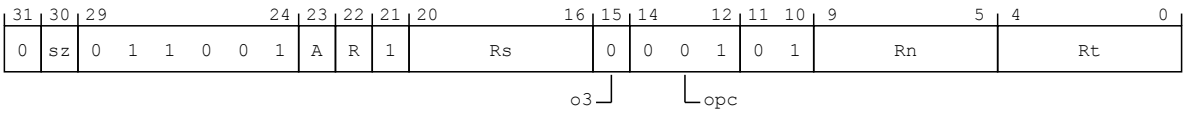

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
LDTCLR  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
LDTCLRA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
LDTCLRAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
LDTCLRL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
LDTCLR  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
LDTCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
LDTCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
LDTCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 32 << UInt(sz);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STTCLR, STTCLRL	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.242 LDTNP

Load unprivileged pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, loads two 64-bit doublewords from memory, and writes them to two registers. For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Signed offset

(FEAT_LSUI)

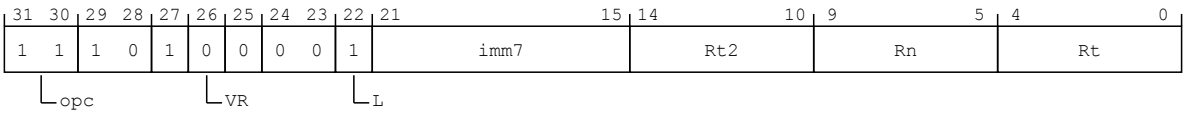

Encoding

```
LDTNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;

let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let datasize : integer{} = 64;
let offset : bits(64) = LSL(SignExtend{64}(imm7), 3);
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

LDTNP has the same CONSTRAINED UNPREDICTABLE behavior as LDNP. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Operation

```
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
let data :  bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    X{datasize}(t) = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    X{datasize}(t2) = data[(datasize-1):0];
    X{datasize}(t)  = data[(2*datasize-1):datasize];
else
    X{datasize}(t)  = data[(datasize-1):0];
    X{datasize}(t2) = data[(2*datasize-1):datasize];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.243 LDTP

Load unprivileged pair of registers

This instruction calculates an address from a base register value and an immediate offset, loads two 64-bit doublewords from memory, and writes them to two registers.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_LSUI)

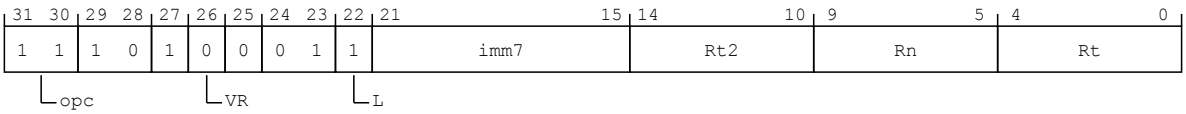

Encoding

```
LDTP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
var wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_LSUI)

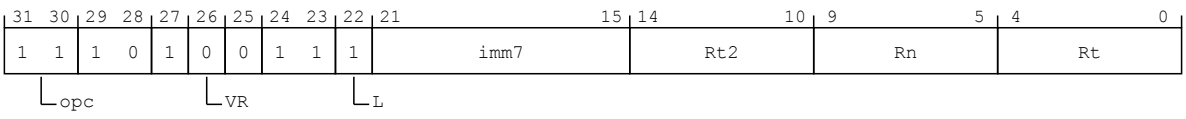

Encoding

```
LDTP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
var wback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	0	0	1 0	1	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
LDTP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
var wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

LDTP has the same CONSTRAINED UNPREDICTABLE behavior as LDP. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Load/store register pair (post-indexed)” and “Load/store register pair (pre-indexed)” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Load/store register pair (offset)” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 64;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;
var wb_unknown : boolean = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS => wback = FALSE;        // Writeback is suppressed
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_UNKNOWN =>    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    data1 = ARBITRARY : bits(datasize);
    data2 = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    data2 = data[(datasize-1):0];
    data1 = data[(2*datasize-1):datasize];
else
    data1 = data[(datasize-1):0];
    data2 = data[(2*datasize-1):datasize];
end;

X{datasize}(t) = data1;
X{datasize}(t2) = data2;

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.244 LDTR

Load register (unprivileged)

This instruction loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

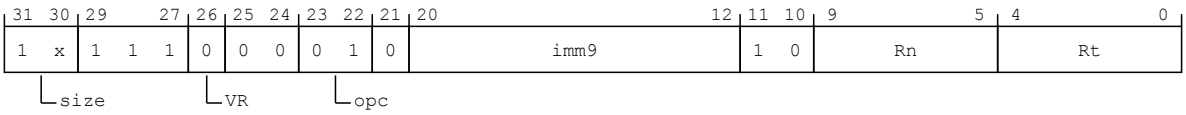

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDTR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDTR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << scale;
let regsize : integer{} = if datasize == 64 then 64 else 32;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.245 LDTRB

Load register byte (unprivileged)

This instruction loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1	1	0	0	0	0 1	0	imm9	1 0	Rn	Rt
size			VR			opc					

Encoding

```
LDTRB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.246 LDTRH

Load register halfword (unprivileged)

This instruction loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size					VR			opc							

Encoding

```
LDTRH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 16;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.247 LDTRSB

Load register signed byte (unprivileged)

This instruction loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

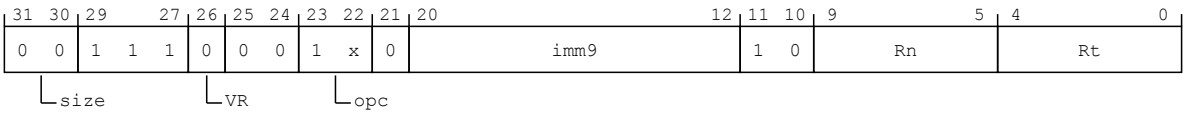

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDTRSB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDTRSB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8;
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.248 LDTRSH

Load register signed halfword (unprivileged)

This instruction loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

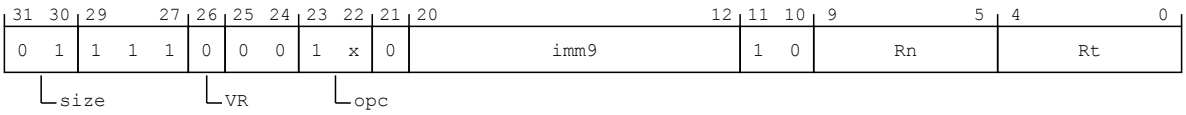

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDTRSH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDTRSH  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 16;
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.249 LDTRSW

Load register signed word (unprivileged)

This instruction loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	1 0	Rn	Rt
size			VR			opc					

Encoding

```
LDTRSW  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 32;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{64}(t) = SignExtend{64}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.250 LDTSET, LDTSETA, LDTSETAL, LDTSETL

Atomic bit set unprivileged

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR`, or `XZR`, `LDTSETA` and `LDTSETAL` load from memory with acquire semantics.
  - `LDTSETL` and `LDTSETAL` store to memory with release semantics.
  - `LDTSET` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

This instruction is used by the alias STTSET, STTSETL.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20	16	15	14			12	11	10	9	5	4	0
0	sz	0	1	1	0	0	1	A	R	1	Rs		0	0	1		1	0	1	Rn		Rt	
													o3			opc							

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
LDTSET  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
LDTSETA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
LDTSETAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
LDTSETL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
LDTSET  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
LDTSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
LDTSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
LDTSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 32 << UInt(sz);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STTSET, STTSETL	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.251 LDTXR

Load unprivileged exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `LDTXR` is equivalent to `LDXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

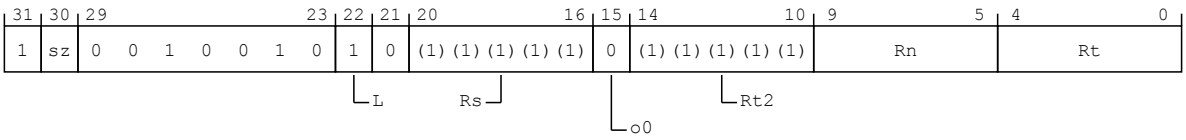

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDTXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDTXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem{elsize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.252 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
  - `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
  - `LDUMAX` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAX, STUMAXL.

Integer

(FEAT_LSE)

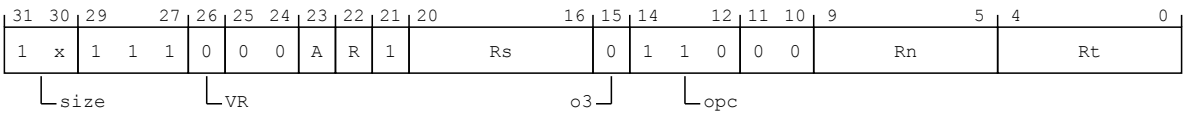

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDUMAX  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDUMAXA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDUMAXAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDUMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDUMAX  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDUMAXA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDUMAXAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDUMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STUMAX, STUMAXL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.253 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAB` and `LDUMAXALB` load from memory with acquire semantics.
  - `LDUMAXLB` and `LDUMAXALB` store to memory with release semantics.
  - `LDUMAXB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMAXAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMAXALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMAXB, STUMAXLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.254 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
  - `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
  - `LDUMAXH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMAXAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMAXALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMAXH, STUMAXLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.255 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
  - `LDUMINL` and `LDUMINAL` store to memory with release semantics.
  - `LDUMIN` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMIN, STUMINL.

Integer

(FEAT_LSE)

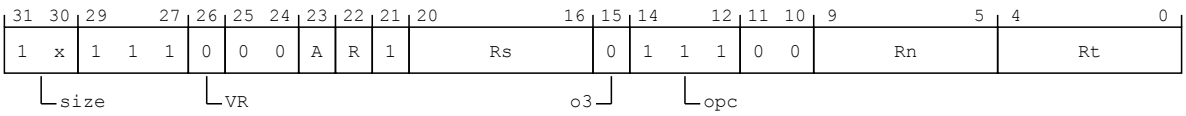

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDUMIN  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDUMINA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDUMINAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDUMINL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDUMIN  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDUMINA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDUMINAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDUMINL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
STUMIN, STUMINL	A == '0' && Rt == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
let value : bits(datasize) = X{datasize}(s);
let data : bits(datasize) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{regsize}(t) = ZeroExtend{regsize}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.256 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
  - `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
  - `LDUMINB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINLB.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMINB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMINAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMINALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMINB, STUMINLB	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
let value : bits(8) = X{8}(s);
let data : bits(8) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.257 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
  - `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
  - `LDUMINH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMINH, STUMINLH.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDUMINH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDUMINAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDUMINALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDUMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
STUMINH, STUMINLH	A == '0' && Rt == '11111'

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
let value : bits(16) = X{16}(s);
let data : bits(16) = MemAtomic{}(address, comparevalue, value, accdesc);

if t != 31 then
    X{32}(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.258 LDUR

Load register (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	1	0	imm9		0	0	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 8 << scale;
let regsize : integer{} = if datasize == 64 then 64 else 32;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.259 LDURB

Load register byte (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1	1	0	0	0	0 1	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
LDURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 8;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.260 LDURH

Load register halfword (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1	1	0	0	0	0 1	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
LDURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 16;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.261 LDURSB

Load register signed byte (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDURSB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDURSB  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 8;
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.262 LDURSH

Load register signed halfword (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size		VR		opc					

Encoding for the 32-bit variant

Applies when `(opc == 11)`

```
LDURSH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
LDURSH  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 16;
let regsize : integer{} = 64 >> UInt(opc[0]);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = SignExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.263 LDURSW

Load register signed word (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 0	1 1	1	0	0	0	1 0	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
LDURSW  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 32;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{64}(t) = SignExtend{64}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.264 LDXP

Load exclusive pair of registers

This instruction derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20					16	15	14				10	9				5	4				0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)		0	Rt2					Rn					Rt				
									L		Rs						o0															

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
LDXP  <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
LDXP  <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let datasize : integer{} = elsize * 2;
let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDXP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
var address : bits(64);
var data : bits(datasize);

let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = elsize == 64;   // When elsize is 32, the access is single-copy atomic
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem{datasize}(address, accdesc);
if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X{datasize}(t)  = ARBITRARY : bits(datasize);        // In this case t = t2
elsif BigEndian(accdesc.acctype) then
    X{elsize}(t)  = data[elsize +: elsize];
    X{elsize}(t2) = data[0 +: elsize];
else
    X{elsize}(t)  = data[0 +: elsize];
    X{elsize}(t2) = data[elsize +: elsize];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.265 LDXR

Load exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

For information about addressing modes, see Load/Store addressing modes.

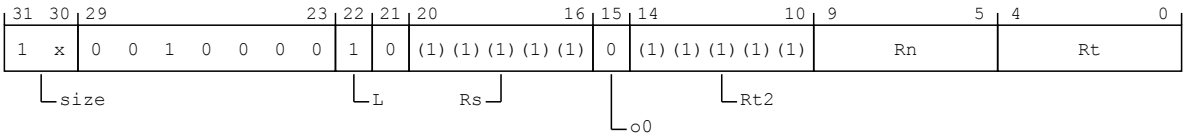

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
LDXR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
LDXR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 8 << UInt(size);
let regsize : integer{} = if elsize == 64 then 64 else 32;
let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem{elsize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.266 LDXRB

Load exclusive register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.

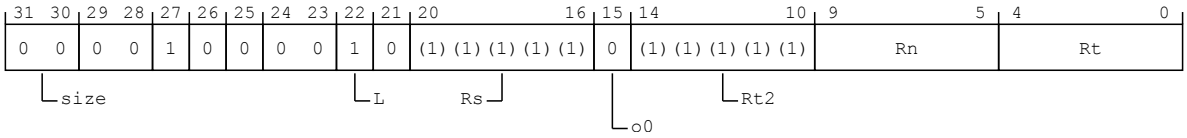

Encoding

```
LDXRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(8);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, 1, accdesc);

data = Mem{8}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.267 LDXRH

Load exclusive register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.

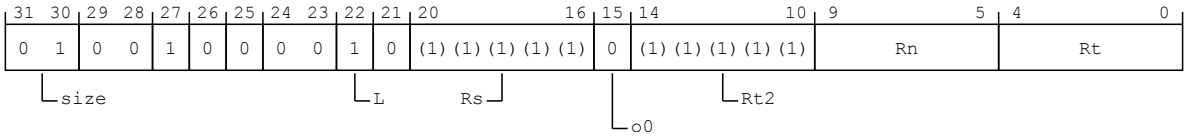

Encoding

```
LDXRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(16);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                     privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

AArch64_SetExclusiveMonitors(address, 2, accdesc);

data = Mem{16}(address, accdesc);
X{32}(t) = ZeroExtend{32}(data);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.268 LSL (register)

Logical shift left (register)

This instruction shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		0	0	Rn					Rd				
		S																				op2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSL  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSLV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSL  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSLV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Operational Information

The description of LSLV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.269 LSL (immediate)

Logical shift left (immediate)

This instruction shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	!IN{x11111}	Rn	Rd
	opc				imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms != 011111)`

```
LSL  <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #(-<shift>  MOD  32), #(31-<shift>)
```

and is the preferred disassembly when `UInt(imms) + 1 == UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1 && imms != 111111)`

```
LSL  <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #(-<shift>  MOD  64), #(63-<shift>)
```

and is the preferred disassembly when `UInt(imms) + 1 == UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

For the “32-bit” variant: is the shift amount, in the range 0 to 31.

For the “64-bit” variant: is the shift amount, in the range 0 to 63.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.270 LSLV

Logical shift left variable

This instruction shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).

31	30	29	28	21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 1 0		0	0	Rn		Rd	
		S								op2				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSLV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSLV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
LSL (register)	Unconditionally

Operation

```
let operand2 : bits(datasize) = X{}(m);

X{datasize}(d) = ShiftReg{datasize}(n, shift_type, UInt(operand2) MOD datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.271 LSR (register)

Logical shift right (register)

This instruction shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

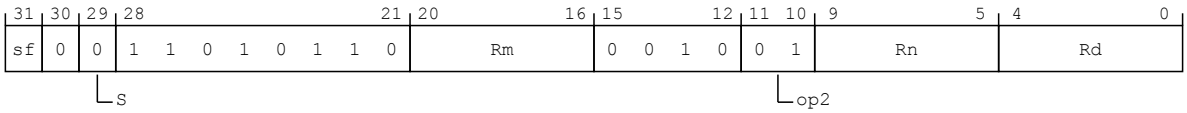

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSR  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSRV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSR  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSRV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Operational Information

The description of LSRV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.272 LSR (immediate)

Logical shift right (immediate)

This instruction shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	x 1 1 1 1 1	Rn	Rd
	opc				imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 011111)`

```
LSR  <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1 && imms == 111111)`

```
LSR  <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, encoded in the “immr” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.273 LSRV

Logical shift right variable

This instruction shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).

31	30	29	28	21	20	16	15	12	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 1 0		0	1	Rn		Rd	
		S								op2				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
LSRV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
LSRV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
LSR (register)	Unconditionally

Operation

```
let operand2 : bits(datasize) = X{}(m);

X{datasize}(d) = ShiftReg{datasize}(n, shift_type, UInt(operand2) MOD datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.274 MADD

Multiply-add

This instruction multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.

31	30 29	28 24	23 21	20 16	15	14 10	9 5	4 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	0	Ra	Rn	Rd
	op54		op31		o0			

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MADD  <Wd>, <Wn>, <Wm>, <Wa>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MADD  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Wa>

Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
MUL	Ra == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let operand3 : bits(datasize) = X{}(a);

let result : integer = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.275 MADDPT

Multiply-add checked pointer

This instruction multiplies two register values, adds a third register value, and writes the result to the destination register. The intermediate product is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

If the intermediate product cannot be correctly represented as a 64-bit two's complement value, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

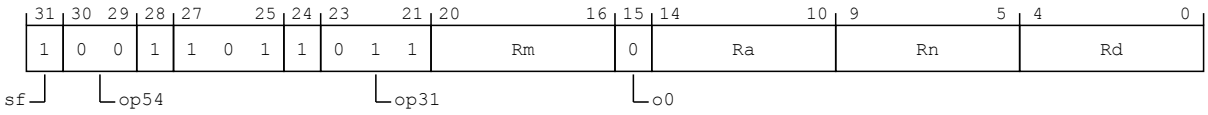

Encoding

```
MADDPT  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
let operand1 : bits(64) = X{}(n);
let operand2 : bits(64) = X{}(m);
let operand3 : bits(64) = X{}(a);

var result : bits(64);

let product : integer = SInt(operand1) * SInt(operand2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Signed and unsigned twos complement arithmetic are equivalent if only a
// fixed number of bits are considered.
result = operand3 + product[63:0];

let overflow : boolean = (product != SInt(product[63:0]));
result = PointerMultiplyAddCheck(result, operand3, overflow);

X{64}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.276 MNEG

Multiply-negate

This instruction multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 24	23 … 21	20 … 16	15	14 … 10	9 … 5	4 … 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	1	1 1 1 1 1	Rn	Rd
	op54		op31		o0	Ra		

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MNEG  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB  <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MNEG  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB  <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational Information

The description of MSUB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.277 MOV (to/from SP)

Move register value to or from SP

This instruction copies the value of a register to or from the stack pointer.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

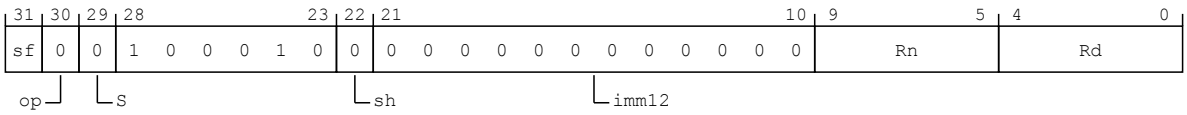

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MOV  <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD  <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `Rd == '11111' || Rn == '11111'`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD  <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `Rd == '11111' || Rn == '11111'`.

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of ADD (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.278 MOV (inverted wide immediate)

Move inverted wide immediate value

This instruction moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22 21	20 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOV  <Wd>, #<imm>
```

is equivalent to

```
MOVN  <Wd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00') && !IsOnes(imm16)`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd>, #<imm>
```

is equivalent to

```
MOVN  <Xd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

For the “32-bit” variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in “imm16:hw”, but excluding 0xffff0000 and `0x0000`ffff

For the “64-bit” variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in “imm16:hw”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.279 MOV (wide immediate)

Move wide immediate value

This instruction moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22 21	20 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOV  <Wd>, #<imm>
```

is equivalent to

```
MOVZ  <Wd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd>, #<imm>
```

is equivalent to

```
MOVZ  <Xd>, #<imm16>, LSL  #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

For the “32-bit” variant: is a 32-bit immediate which can be encoded in “imm16:hw”.

For the “64-bit” variant: is a 64-bit immediate which can be encoded in “imm16:hw”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.280 MOV (bitmask immediate)

Move bitmask immediate value

This instruction writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 1	1 0 0 1 0 0	N	immr	imms	1 1 1 1 1	Rd
	opc					Rn	

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
MOV  <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR  <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `!MoveWidePreferred(sf, N, imms, immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd|SP>, #<imm>
```

is equivalent to

```
ORR  <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `!MoveWidePreferred(sf, N, imms, immr)`.

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<imm>

For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”, but excluding values which could be encoded by MOVZ or MOVN.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”, but excluding values which could be encoded by MOVZ or MOVN.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of ORR (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.281 MOV (register)

Move register value

This instruction copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

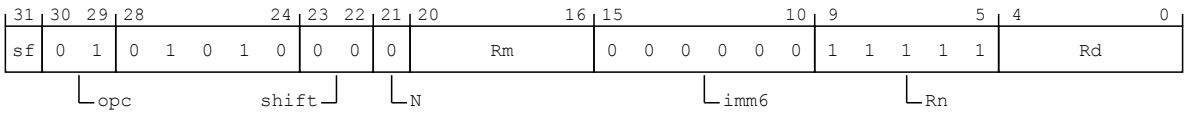

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MOV  <Wd>, <Wm>
```

is equivalent to

```
ORR  <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOV  <Xd>, <Xm>
```

is equivalent to

```
ORR  <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of ORR (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.282 MOVK

Move wide with keep

This instruction moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30 29	28 23	22 21	20 5	4 0
sf	1 1 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOVK  <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOVK  <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw[1] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
let datasize : integer{} = 32 << UInt(sf);
let imm : bits(16) = imm16;
let pos : integer{} = UInt(hw) << 4;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<shift>

For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
var result : bits(datasize) = X{}(d);
result[pos+15:pos] = imm;
X{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.283 MOVN

Move wide with NOT

This instruction moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate).

31	30 29	28 … 23	22 21	20 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 0 1	hw	imm16	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOVN  <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOVN  <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw[1] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
let datasize : integer{} = 32 << UInt(sf);
let imm : bits(16) = imm16;
let pos : integer{} = UInt(hw) << 4;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<shift>

For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Of variant	Is preferred when
MOV (inverted wide immediate)	32-bit	!(IsZero(imm16) && hw != '00') && !IsOnes(imm16)
MOV (inverted wide immediate)	64-bit	!(IsZero(imm16) && hw != '00')

Operation

```
var result : bits(datasize) = Zeros{};
result[pos+15:pos] = imm;
X{datasize}(d) = NOT(result);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.284 MOVZ

Move wide with zero

This instruction moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate).

31	30 29	28 23	22 21	20 5	4 0
sf	1 0	1 0 0 1 0 1	hw	imm16	Rd
	opc				

Encoding for the 32-bit variant

Applies when `(sf == 0 && hw == 0x)`

```
MOVZ  <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MOVZ  <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw[1] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
let datasize : integer{} = 32 << UInt(sf);
let imm : bits(16) = imm16;
let pos : integer{} = UInt(hw) << 4;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<shift>

For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (wide immediate)	!(IsZero(imm16) && hw != '00')

Operation

```
var result : bits(datasize) = Zeros{};
result[pos+15:pos] = imm;
X{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.285 MRRS

Move System register to two adjacent general-purpose registers

This instruction allows the PE to read an AArch64 128-bit System register into two adjacent 64-bit general-purpose registers.

System

(FEAT_SYSREG128)

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 1	1	1	o0	op1	CRn	CRm	op2	Rt

L

Encoding

```
MRRS  <Xt>, <Xt+1>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSREG128) then EndOfDecode(Decode_UNDEF); end;
if Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;

let t : integer       = UInt(Rt);
let t2 : integer      = UInt(Rt+1);
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '1' :: o0;
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the first general-purpose destination register, encoded in the “Rt” field.

<Xt+1>

Is the 64-bit name of the second general-purpose destination register, encoded as “Rt” +1.

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Operation

```
AArch64_SysRegRead128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.286 MRS

Move System register to general-purpose register

This instruction allows the PE to read an AArch64 System register into a general-purpose register.

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	1	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MRS  <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
let t : integer       = UInt(Rt);
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '1' :: o0;
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

The System register names are defined in AArch64 System Register Descriptions.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Operation

```
AArch64_SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.287 MSR (immediate)

Move immediate value to special register

This instruction moves an immediate value to selected bits of the PSTATE. For more information, see Process state, PSTATE.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If FEAT_SSBS is implemented, PSTATE.SSBS.
- If FEAT_PAN is implemented, PSTATE.PAN.
- If FEAT_UAO is implemented, PSTATE.UAO.
- If FEAT_DIT is implemented, PSTATE.DIT.
- If FEAT_MTE is implemented, PSTATE.TCO.
- If FEAT_NMI is implemented, PSTATE.ALLINT.
- If FEAT_SME is implemented, PSTATE.SM and PSTATE.ZA.
- If FEAT_EBEP is implemented, PSTATE.PM.

If FEAT_MTE is implemented and FEAT_MTE2 is not implemented, it is IMPLEMENTATION DEFINED whether writes to PSTATE.TCO by this instruction are ignored.

This instruction is used by the aliases SMSTART and SMSTOP.

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	op1	0 1 0 0	CRm	op2	1 1 1 1 1 (Rt)

Encoding

Applies when `(!(op1 == 000 && op2 IN {00x, 010}))`

```
MSR  <pstatefield>, #<imm>
```

Decode for this encoding

```
if op1 == '000' && op2 == '000' then See("CFINV"); end;
if op1 == '000' && op2 == '001' then See("XAFLAG"); end;
if op1 == '000' && op2 == '010' then See("AXFLAG"); end;

var min_EL : bits(2);
var need_secure : boolean = FALSE;

case op1 of
    when '00x' =>
        min_EL = EL1;
    when '010' =>
        min_EL = EL1;
    when '011' =>
        min_EL = EL0;
    when '100' =>
        min_EL = EL2;
    when '101' =>
        if !IsFeatureImplemented(FEAT_VHE) then EndOfDecode(Decode_UNDEF); end;
        min_EL = EL2;
    when '110' =>
        min_EL = EL3;
    when '111' =>
        min_EL = EL1;
        need_secure = TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let operand : bits(4) = CRm;
var field : PSTATEField;
case op1::op2 of
    when '000 011' =>
        if !IsFeatureImplemented(FEAT_UAO) then EndOfDecode(Decode_UNDEF); end;
        field = PSTATEField_UAO;
    when '000 100' =>
        if !IsFeatureImplemented(FEAT_PAN) then EndOfDecode(Decode_UNDEF); end;
        field = PSTATEField_PAN;
    when '000 101' => field = PSTATEField_SP;
    when '001 000' =>
        case CRm of
            when '000x' =>
                if !IsFeatureImplemented(FEAT_NMI) then EndOfDecode(Decode_UNDEF); end;
                field = PSTATEField_ALLINT;
            when '001x' =>
                if !IsFeatureImplemented(FEAT_EBEP) then EndOfDecode(Decode_UNDEF); end;
                field = PSTATEField_PM;
            otherwise =>
                EndOfDecode(Decode_UNDEF);
        end;
    when '011 010' =>
        if !IsFeatureImplemented(FEAT_DIT) then EndOfDecode(Decode_UNDEF); end;
        field = PSTATEField_DIT;
    when '011 011' =>
        case CRm of
            when '001x' =>
                if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
                field = PSTATEField_SVCRSM;
            when '010x' =>
                if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
                field = PSTATEField_SVCRZA;
            when '011x' =>
                if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
                field = PSTATEField_SVCRSMZA;
            otherwise =>
                EndOfDecode(Decode_UNDEF);
        end;
    when '011 100' =>
        if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
        field = PSTATEField_TCO;
    when '011 110' => field = PSTATEField_DAIFSet;
    when '011 111' => field = PSTATEField_DAIFClr;
    when '011 001' =>
        if !IsFeatureImplemented(FEAT_SSBS) then EndOfDecode(Decode_UNDEF); end;
        field = PSTATEField_SSBS;
    otherwise =>      EndOfDecode(Decode_UNDEF);
end;
```

Assembler Symbols

<pstatefield>

Is a PSTATE field name. For the MSR instruction, this is encoded in “CRm:op1:op2”:

CRm	op1	op2	<pstatefield>	Architectural Feature
000x	001	000	ALLINT	FEAT_NMI
000x	011	011	RESERVED	-
001x	001	000	PM	FEAT_EBEP

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	op1	op2	<pstatefield>	Architectural Feature
001x	011	011	SVCRSM	FEAT_SME
010x	011	011	SVCRZA	FEAT_SME
01xx	001	000	RESERVED	-
011x	011	011	SVCRSMZA	FEAT_SME
xxxx	000	011	UAO	FEAT_UAO
xxxx	000	100	PAN	FEAT_PAN
xxxx	000	101	SPSel	-
xxxx	000	11x	RESERVED	-
xxxx	001	001	RESERVED	-
xxxx	001	01x	RESERVED	-
xxxx	001	1xx	RESERVED	-
xxxx	010	xxx	RESERVED	-
xxxx	011	000	RESERVED	-
xxxx	011	001	SSBS	FEAT_SSBS
xxxx	011	010	DIT	FEAT_DIT
xxxx	011	100	TCO	FEAT_MTE
xxxx	011	101	RESERVED	-
xxxx	011	110	DAIFSet	-
xxxx	011	111	DAIFClr	-
xxxx	1xx	xxx	RESERVED	-
1xxx	001	000	RESERVED	-
1xxx	011	011	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in “CRm<2:1>”, where `0b01` specifies SVCRSM, `0b10` specifies SVCRZA, and `0b11` specifies SVCRSMZA.

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field. Restricted to the range 0 to 1, encoded in “CRm<0>”, when <pstatefield> is ALLINT, PM, SVCRSM, SVCRSMZA, or SVCRZA.

Alias Conditions

Alias	Is preferred when
SMSTART	op1 == '011' && CRm IN {'0xx1'} && op2 == '011'
SMSTOP	op1 == '011' && CRm IN {'0xx0'} && op2 == '011'

Operation

```
if UInt(PSTATE.EL) < UInt(min_EL) then Undefined(); end;
if need_secure && CurrentSecurityState() != SS_Secure then Undefined(); end;

case field of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when PSTATEField_SSBS =>
        PSTATE.SSBS = operand[0];
    when PSTATEField_SP =>
        PSTATE.SP = operand[0];
    when PSTATEField_DAIFSet =>
        AArch64_CheckDAIFAccess(PSTATEField_DAIFSet);
        PSTATE.D = PSTATE.D OR operand[3];
        PSTATE.A = PSTATE.A OR operand[2];
        PSTATE.I = PSTATE.I OR operand[1];
        PSTATE.F = PSTATE.F OR operand[0];
    when PSTATEField_DAIFClr =>
        AArch64_CheckDAIFAccess(PSTATEField_DAIFClr);
        PSTATE.D = PSTATE.D AND NOT(operand[3]);
        PSTATE.A = PSTATE.A AND NOT(operand[2]);
        PSTATE.I = PSTATE.I AND NOT(operand[1]);
        PSTATE.F = PSTATE.F AND NOT(operand[0]);
    when PSTATEField_PAN =>
        PSTATE.PAN = operand[0];
    when PSTATEField_UAO =>
        PSTATE.UAO = operand[0];
    when PSTATEField_DIT =>
        PSTATE.DIT = operand[0];
    when PSTATEField_TCO =>
        PSTATE.TCO = operand[0];
    when PSTATEField_ALLINT =>
        if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() &&
              HCRX_EL2().TALLINT == '1' && operand[0] == '1') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
        PSTATE.ALLINT = operand[0];
    when PSTATEField_SVCRSM =>
        CheckSMEAccess();
        SetPSTATE_SM(operand[0]);
    when PSTATEField_SVCRZA =>
        CheckSMEAccess();
        SetPSTATE_ZA(operand[0]);
    when PSTATEField_SVCRSMZA =>
        CheckSMEAccess();
        SetPSTATE_SM(operand[0]);
        SetPSTATE_ZA(operand[0]);
    when PSTATEField_PM =>
        PSTATE.PM = operand[0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.288 MSR (register)

Move general-purpose register to System register

This instruction allows the PE to write an AArch64 System register from a general-purpose register.

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MSR  (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

Decode for this encoding

```
let t : integer       = UInt(Rt);
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '1' :: o0;
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

The System register names are defined in AArch64 System Register Descriptions.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

```
AArch64_SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.289 MSRR

Move two adjacent general-purpose registers to System register

This instruction allows the PE to write an AArch64 128-bit System register from two adjacent 64-bit general-purpose registers.

System

(FEAT_SYSREG128)

31–29	28–26	25–22	21	20	19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 1	0	1	o0	op1	CRn	CRm	op2	Rt
			L							

Encoding

```
MSRR  (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>, <Xt+1>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSREG128) then EndOfDecode(Decode_UNDEF); end;
if Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;

let t : integer       = UInt(Rt);
let t2 : integer      = UInt(Rt+1);
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '1' :: o0;
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<systemreg>

Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

<op0>

Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<Xt>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rt” field.

<Xt+1>

Is the 64-bit name of the second general-purpose source register, encoded as “Rt” +1.

Operation

```
AArch64_SysRegWrite128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.290 MSUB

Multiply-subtract

This instruction multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.

31	30 29	28 24	23 21	20 16	15	14 10	9 5	4 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	1	Ra	Rn	Rd
	op54		op31		o0			

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MSUB  <Wd>, <Wn>, <Wm>, <Wa>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MSUB  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Wa>

Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
MNEG	Ra == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let operand3 : bits(datasize) = X{}(a);

let result : integer = UInt(operand3) - (UInt(operand1) * UInt(operand2));

X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.291 MSUBPT

Multiply-subtract checked pointer

This instruction multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register. The intermediate product is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

If the intermediate product cannot be correctly represented as a 64-bit two's complement value, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

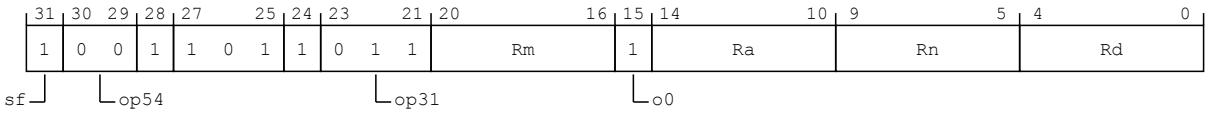

Encoding

```
MSUBPT  <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
let operand1 : bits(64) = X{}(n);
let operand2 : bits(64) = X{}(m);
let operand3 : bits(64) = X{}(a);

var result : bits(64);

let product : integer = SInt(operand1) * SInt(operand2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Signed and unsigned twos complement arithmetic are equivalent if only a
// fixed number of bits are considered.
result = operand3 - product[63:0];

let overflow : boolean = (product != SInt(product[63:0]));
result = PointerMultiplyAddCheck(result, operand3, overflow);

X{64}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.292 MUL

Multiply

This instruction multiplies two register values and writes the result to the destination register.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

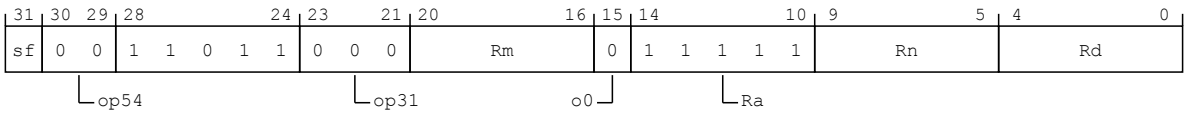

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MUL  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MADD  <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MUL  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MADD  <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of MADD gives the operational pseudocode for this instruction.

Operational Information

The description of MADD gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.293 MVN

Bitwise NOT

This instruction writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

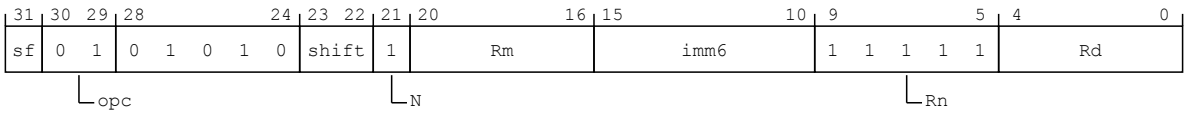

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
MVN  <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN  <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
MVN  <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN  <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
01	LSR
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ORN (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.294 NEG (shifted register)

Negate (shifted register)

This instruction negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

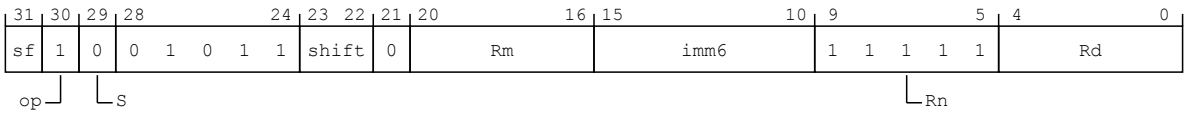

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NEG  <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NEG  <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUB (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.295 NEGS

Negate, setting flags

This instruction negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	24	23 22	21	20	16	15	10	9	5	4	0
sf	1	1	0 1 0 1 1		shift	0	Rm		imm6		1 1 1 1 1		!=11111	
	op	S										Rn		Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NEGS  <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NEGS  <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS  <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.296 NGC

Negate with carry

This instruction negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28							21	20				16	15					10	9				5	4				0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	1	1	1	1	1	Rd				
op		S																						Rn							

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NGC  <Wd>, <Wm>
```

is equivalent to

```
SBC  <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NGC  <Xd>, <Xm>
```

is equivalent to

```
SBC  <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of SBC gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.297 NGCS

Negate with carry, setting flags

This instruction negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

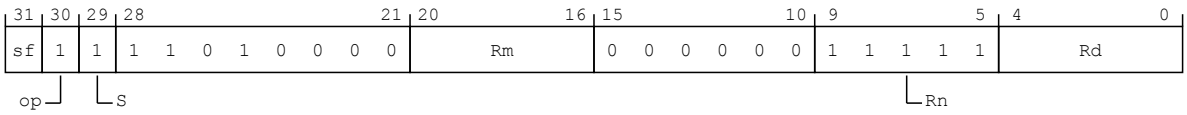

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
NGCS  <Wd>, <Wm>
```

is equivalent to

```
SBCS  <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
NGCS  <Xd>, <Xm>
```

is equivalent to

```
SBCS  <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wm>

Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of SBCS gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.298 NOP

No operation

This instruction does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 0 0 (op2)	1 1 1 1 1

Encoding

```
NOP
```

Decode for this encoding

```
// Empty.
```

Operation

```
return; // Do nothing
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.299 ORN (shifted register)

Bitwise OR NOT (shifted register)

This instruction performs a bitwise (inclusive) OR between a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 1	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ORN  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ORN  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
MVN	Rn == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

X{datasize}(d) = operand1 OR NOT(operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.300 ORR (immediate)

Bitwise OR (immediate)

This instruction performs a bitwise (inclusive) OR between a register value and an immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate).

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 1 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
ORR  <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ORR  <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
var imm : bits(datasize);
(imm, -) = DecodeBitMasks{datasize}(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is the bitmask immediate. For encoding information, see Logical (immediate).

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (bitmask immediate)	Rn == '11111' && !MoveWidePreferred(sf, N, imms, immr)

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = imm;

let result : bits(datasize) = operand1 OR operand2;

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.301 ORR (shifted register)

Bitwise OR (shifted register)

This instruction performs a bitwise (inclusive) OR between a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register).

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	0 1 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ORR  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ORR  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
MOV (register)	shift == '00' && imm6 == '000000' && Rn == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = ShiftReg{}(m, shift_type, shift_amount);

X{datasize}(d) = operand1 OR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.302 PACDA, PACDZA

Pointer Authentication Code for data address, using key A

This instruction calculates and inserts a Pointer Authentication Code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACDA`.
- The value zero, for `PACDZA`.

Integer

(FEAT_PAuth)

31	30	29	28							21	20				16	15	14	13	12		10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	0	Rn		Rd	
sf		S									opcode2														

Encoding for the PACDA variant

Applies when `(Z == 0)`

```
PACDA  <Xd>, <Xn|SP>
```

Encoding for the PACDZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACDZA  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;

var source_is_sp : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE; end;
else // PACDZA
    if n != 31 then EndOfDecode(Decode_UNDEF); end;
end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    X{64}(d) = AddPACDA(X{64}(d), SP{64}());
else
    X{64}(d) = AddPACDA(X{64}(d), X{64}(n));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.303 PACDB, PACDZB

Pointer Authentication Code for data address, using key B

This instruction calculates and inserts a Pointer Authentication Code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACDB`.
- The value zero, for `PACDZB`.

Integer

(FEAT_PAuth)

31	30	29	28							21	20				16	15	14	13	12		10	9	5	4	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	1	Rn		Rd	
sf		S									opcode2														

Encoding for the PACDB variant

Applies when `(Z == 0)`

```
PACDB  <Xd>, <Xn|SP>
```

Encoding for the PACDZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACDZB  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;

var source_is_sp : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE; end;
else // PACDZB
    if n != 31 then EndOfDecode(Decode_UNDEF); end;
end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if source_is_sp then
    X{64}(d) = AddPACDB(X{64}(d), SP{64}());
else
    X{64}(d) = AddPACDB(X{64}(d), X{64}(n));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.304 PACGA

Pointer Authentication Code, using generic key

This instruction calculates the Pointer Authentication Code for a 64-bit value in the first source register, using a modifier in the second source register, and the generic key. The calculated Pointer Authentication Code is written to the most significant 32 bits of the destination register, and the least significant 32 bits of the destination register are set to zero.

Integer

(FEAT_PAuth)

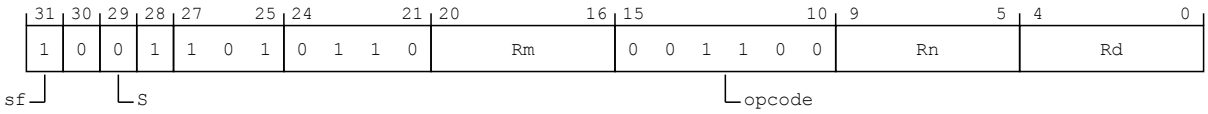

Encoding

```
PACGA  <Xd>, <Xn>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;

var source_is_sp : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

if m == 31 then source_is_sp = TRUE; end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Operation

```
if source_is_sp then
    X{64}(d) = AddPACGA(X{64}(n), SP{64}());
else
    X{64}(d) = AddPACGA(X{64}(n), X{64}(m));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.305 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for instruction address, using key A

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `PACIA` and `PACIZA`.
- In X17, for `PACIA1716`.
- In X30, for `PACIASP` and `PACIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACIA`.
- The value zero, for `PACIZA` and `PACIAZ`.
- In X16, for `PACIA1716`.
- In SP, for `PACIASP`.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `PACIA1716` and `PACIASP` include a second modifier that is:

- In X15, for `PACIA1716`.
- The value of PC, for `PACIASP`.

A `PACIASP` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIASP` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIASP` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

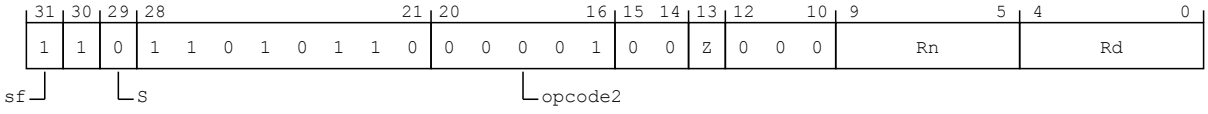

Encoding for the PACIA variant

Applies when `(Z == 0)`

```
PACIA  <Xd>, <Xn|SP>
```

Encoding for the PACIZA variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACIZA  <Xd>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
var source_is_sp : boolean = FALSE;
let pacia1716 : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE; end;
else // PACIZA
    if n != 31 then EndOfDecode(Decode_UNDEF); end;
end;
```

System

(FEAT_PAuth)

31																			12	11			8	7			5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	0	x		1	1	1	1	1
																				CRm						op2						

Encoding for the PACIA1716 variant

Applies when `(CRm == 0001 && op2 == 000)`

```
PACIA1716
```

Encoding for the PACIASP variant

Applies when `(CRm == 0011 && op2 == 001)`

```
PACIASP
```

Encoding for the PACIAZ variant

Applies when `(CRm == 0011 && op2 == 000)`

```
PACIAZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP); end;
var d : integer;
var n : integer;
var source_is_sp : boolean = FALSE;
var pacia1716 : boolean = FALSE;

case CRm::op2 of
    when '0011 000' => // PACIAZ
        d = 30;
        n = 31;
    when '0011 001' => // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if IsFeatureImplemented(FEAT_BTI) then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIASP instruction.
            let pacinst : PACInstType = PACIxSP;
            SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '0001 000' => // PACIA1716
        d = 17;
        n = 16;
        pacia1716 = TRUE;
end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X{64}(d) = AddPACIA2(X{64}(d), SP{64}(), PC64());
    else
        X{64}(d) = AddPACIA(X{64}(d), SP{64}());
    end;
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && pacia1716 then
        X{64}(d) = AddPACIA2(X{64}(d), X{64}(n), X{64}(15));
    else
        X{64}(d) = AddPACIA(X{64}(d), X{64}(n));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.306 PACIA171615

Pointer Authentication Code for instruction address, using key A

This instruction calculates and inserts a pointer authentication code for an instruction address, using two modifiers and key A. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

Integer

(FEAT_PAuth_LR)

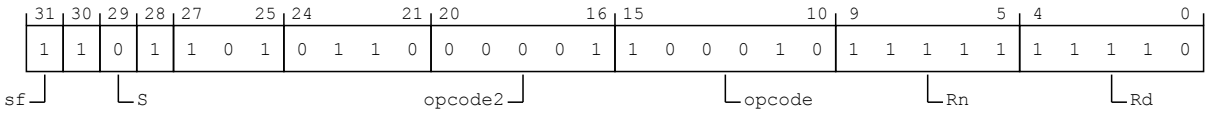

Encoding

```
PACIA171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
X{64}(17) = AddPACIA2(X{64}(17), X{64}(16), X{64}(15));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.307 PACIASPPC

Pointer Authentication Code for return address, using key A

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key A.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

A `PACIASPPC` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIASPPC` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIASPPC` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

Integer

(FEAT_PAuth_LR)

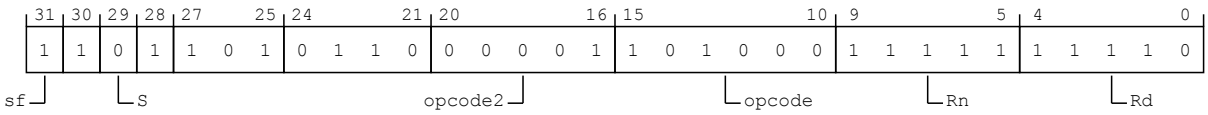

Encoding

```
PACIASPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
if IsFeatureImplemented(FEAT_BTI) then
    // Check for branch target compatibility between PSTATE.BTYPE
    // and implicit branch target of PACIxSPPC instruction.
    let pacinst : PACInstType = PACIxSPPC;
    SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
end;
```

Operation

```
X{64}(d) = AddPACIA2(X{64}(d), SP{64}(), PC64());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.308 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for instruction address, using key B

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by `<Xd>` for `PACIB` and `PACIZB`.
- In X17, for `PACIB1716`.
- In X30, for `PACIBSP` and `PACIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for `PACIB`.
- The value zero, for `PACIZB` and `PACIBZ`.
- In X16, for `PACIB1716`.
- In SP, for `PACIBSP`.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `PACIB1716` and `PACIBSP` include a second modifier that is:

- In X15, for `PACIB1716`.
- The value of PC, for `PACIBSP`.

A `PACIBSP` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIBSP` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIBSP` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

31	30	29	28							21	20				16	15	14	13	12		10	9		5	4		0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	1	Rn			Rd		
sf			S								opcode2																

Encoding for the PACIB variant

Applies when `(Z == 0)`

```
PACIB  <Xd>, <Xn|SP>
```

Encoding for the PACIZB variant

Applies when `(Z == 1 && Rn == 11111)`

```
PACIZB  <Xd>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
var source_is_sp : boolean = FALSE;
let pacib1716 : boolean = FALSE;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE; end;
else // PACIZB
    if n != 31 then EndOfDecode(Decode_UNDEF); end;
end;
```

System

(FEAT_PAuth)

31																			12	11			8	7		5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	1	x	1	1	1	1	1
																				CRm					op2						

Encoding for the PACIB1716 variant

Applies when `(CRm == 0001 && op2 == 010)`

```
PACIB1716
```

Encoding for the PACIBSP variant

Applies when `(CRm == 0011 && op2 == 011)`

```
PACIBSP
```

Encoding for the PACIBZ variant

Applies when `(CRm == 0011 && op2 == 010)`

```
PACIBZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP); end;
var d : integer;
var n : integer;
var source_is_sp : boolean = FALSE;
var pacib1716 : boolean = FALSE;

case CRm::op2 of
    when '0011 010' => // PACIBZ
        d = 30;
        n = 31;
    when '0011 011' => // PACIBSP
        d = 30;
        source_is_sp = TRUE;
        if IsFeatureImplemented(FEAT_BTI) then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIBSP instruction.
            let pacinst : PACInstType = PACIxSP;
            SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '0001 010' => // PACIB1716
        d = 17;
        n = 16;
        pacib1716 = TRUE;
end;
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
        X{64}(d) = AddPACIB2(X{64}(d), SP{64}(), PC64());
    else
        X{64}(d) = AddPACIB(X{64}(d), SP{64}());
    end;
else
    if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' && pacib1716 then
        X{64}(d) = AddPACIB2(X{64}(d), X{64}(n), X{64}(15));
    else
        X{64}(d) = AddPACIB(X{64}(d), X{64}(n));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.309 PACIB171615

Pointer Authentication Code for instruction address, using key B

This instruction calculates and inserts a pointer authentication code for an instruction address, using two modifiers and key B. The address is in X17. The 64-bit value of modifier1 is the value in X16. The 64-bit value of modifier2 is the value in X15.

Integer

(FEAT_PAuth_LR)

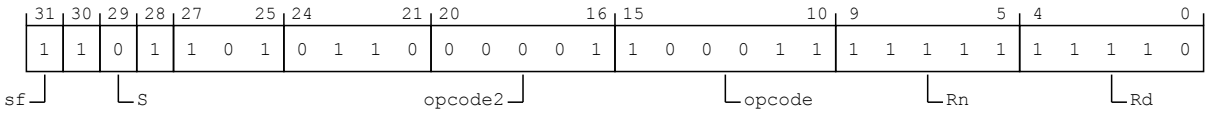

Encoding

```
PACIB171615
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
X{64}(17) = AddPACIB2(X{64}(17), X{64}(16), X{64}(15));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.310 PACIBSPPC

Pointer Authentication Code for return address, using key B

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key B.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

A `PACIBSPPC` instruction has an implicit `BTI` instruction. The implicit `BTI` instruction of a `PACIBSPPC` instruction is always compatible with PSTATE.BTYPE == `0b01` and PSTATE.BTYPE == `0b10`. Controls in SCTLR_ELx configure whether the implicit `BTI` instruction of a `PACIBSPPC` instruction is compatible with PSTATE.BTYPE == `0b11`. For more information, see PSTATE.BTYPE.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0
sf		S									opcode2					opcode						Rn					Rd				

Encoding

```
PACIBSPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
if IsFeatureImplemented(FEAT_BTI) then
    // Check for branch target compatibility between PSTATE.BTYPE
    // and implicit branch target of PACIxSPPC instruction.
    let pacinst : PACInstType = PACIxSPPC;
    SetBTypeCompatible(BTypeCompatible_PAC(pacinst));
end;
```

Operation

```
X{64}(d) = AddPACIB2(X{64}(d), SP{64}(), PC64());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.311 PACM

Pointer authentication modifier

This instruction is used to set the value of PSTATE.PACM to 1.

If FEAT_PAuth_LR is not implemented, this instruction behaves as a NOP.

System

(FEAT_PAuth_LR)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 0 (CRm)	1 1 1 (op2)	1 1 1 1 1

Encoding

```
PACM
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_NOP); end;
```

Operation

```
PSTATE.PACM = if IsPACMEnabled() then '1' else '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.312 PACNBIASPPC

Pointer Authentication Code for return address, using key A, not a branch target

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key A.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	1	0	1	1 0 1	0 1 1 0	0 0 0 0 1	1 0 0 0 0 0	1 1 1 1 1	1 1 1 1 0
sf		S				opcode2	opcode	Rn	Rd

Encoding

```
PACNBIASPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
```

Operation

```
X{64}(d) = AddPACIA2(X{64}(d), SP{64}(), PC64());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.313 PACNBIBSPPC

Pointer Authentication Code for return address, using key B, not a branch target

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using two modifiers and key B.

The address is in X30.

The first modifier is in SP.

The second modifier is the value of PC.

Integer

(FEAT_PAuth_LR)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	1	0	1	1 0 1	0 1 1 0	0 0 0 0 1	1 0 0 0 0 1	1 1 1 1 1	1 1 1 1 0
sf		S				opcode2	opcode	Rn	Rd

Encoding

```
PACNBIBSPPC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let d : integer = 30;
```

Operation

```
X{64}(d) = AddPACIB2(X{64}(d), SP{64}(), PC64());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.314 PRFM (immediate)

Prefetch memory (immediate)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from a base register value and an immediate offset. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - When FEAT_PCDPHINT is implemented, IR for intent to read on update.
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

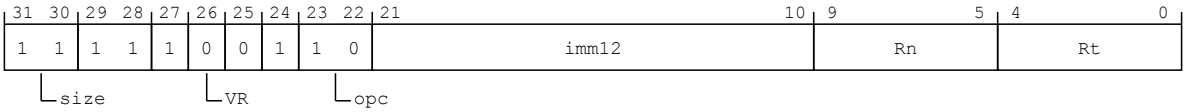

Encoding

```
PRFM  (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 3);
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = FALSE;
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00010	PLDL2KEEP	-
00011	PLDL2STRM	-
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC
11000	IR	FEAT_PCDPHINT

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
let supports_ir : boolean = IsFeatureImplemented(FEAT_PCDPHINT);
Prefetch(address, t[4:0], supports_ir);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.315 PRFM (literal)

Prefetch memory (literal)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from the PC value and an immediate offset. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

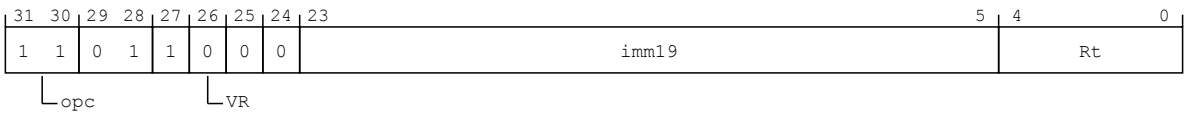

Encoding

```
PRFM  (<prfop>|#<imm5>), <label>
```

Decode for this encoding

```
let t : integer = UInt(Rt);

let offset : bits(64) = SignExtend{}(imm19::'00');
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-
00010	PLDL2KEEP	-
00011	PLDL2STRM	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
let address : bits(64) = PC64() + offset;

let supports_ir : boolean = FALSE;
Prefetch(address, t[4:0], supports_ir);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.316 PRFM (register)

Prefetch memory (register)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from a base register value and an offset register value. The offset register value can optionally be shifted and extended. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 1	1 1	1	0	0	0	1 0	1	Rm	x 1 x	S	1 0	Rn	!=11xxx
size			VR			opc			option				Rt

Encoding

```
PRFM  (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then 3 else 0;
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let m : integer = UInt(Rm);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = FALSE;
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-
00010	PLDL2KEEP	-
00011	PLDL2STRM	-
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Operation

```
var address : bits(64);

let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
let supports_ir : boolean = FALSE;
Prefetch(address, t[4:0], supports_ir);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.317 PRFUM

Prefetch memory (unscaled offset)

This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
  - When FEAT_PRFMSLC is implemented, SLC for system level cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFUM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 1	1 1	1	0	0	0	1 0	0	imm9	0 0	Rn	Rt
size			VR			opc					

Encoding

```
PRFUM  (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = FALSE;
```

Assembler Symbols

<prfop>

Is the prefetch operation, encoded in “Rt”:

Rt	<prfop>	Architectural Feature
00000	PLDL1KEEP	-
00001	PLDL1STRM	-
00010	PLDL2KEEP	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>	Architectural Feature
00011	PLDL2STRM	-
00100	PLDL3KEEP	-
00101	PLDL3STRM	-
00110	PLDSLCKEEP	FEAT_PRFMSLC
00111	PLDSLCSTRM	FEAT_PRFMSLC
01000	PLIL1KEEP	-
01001	PLIL1STRM	-
01010	PLIL2KEEP	-
01011	PLIL2STRM	-
01100	PLIL3KEEP	-
01101	PLIL3STRM	-
01110	PLISLCKEEP	FEAT_PRFMSLC
01111	PLISLCSTRM	FEAT_PRFMSLC
10000	PSTL1KEEP	-
10001	PSTL1STRM	-
10010	PSTL2KEEP	-
10011	PSTL2STRM	-
10100	PSTL3KEEP	-
10101	PSTL3STRM	-
10110	PSTSLCKEEP	FEAT_PRFMSLC
10111	PSTSLCSTRM	FEAT_PRFMSLC

For other encodings of the “Rt” field, use <imm5>.

<imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using `<prfop>`.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    address = SP{64}();
else
    address = X{64}(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

address = AddressAdd(address, offset, accdesc);
let supports_ir : boolean = FALSE;
Prefetch(address, t[4:0], supports_ir);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.318 PSB

Profiling synchronization barrier

This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If FEAT_SPE is not implemented, this instruction executes as a `NOP`.

System

(FEAT_SPE)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 0 1 (op2)	1 1 1 1 1

Encoding

```
PSB  CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SPE) then EndOfDecode(Decode_NOP); end;
```

Operation

```
if IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_SPEv1p5) then
    let trap_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                 !IsInHost() &&
                                 (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
                                 HFGITR_EL2().PSBCSYNC == '1');
    if trap_to_el2 then
        let target_el : bits(2) = EL2;
        let iss : bits(25)      = 0x3[24:0];
        AArch64_OtherInstrTrap(target_el, iss);
    end;
end;

ProfilingSynchronizationBarrier();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.319 PSSBB

Physical speculative store bypass barrier

This instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see Physical Speculative Store Bypass Barrier (PSSBB).

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	0 1 0 0 (CRm)	1	0 0 (opc)	1 1 1 1 1 (Rt)

Encoding

```
PSSBB
```

is equivalent to

```
DSB  #4
```

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.320 RBIT

Reverse bits

This instruction reverses the bit order in a register.

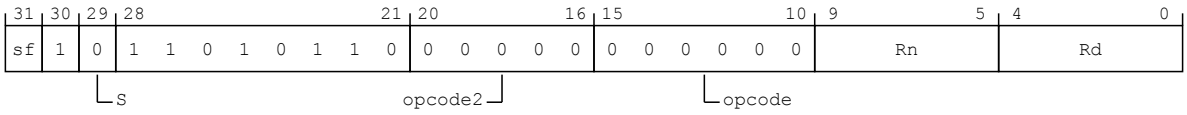

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
RBIT  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
RBIT  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand : bits(datasize) = X{}(n);
var result : bits(datasize);

for i = 0 to datasize-1 do
    result[(datasize-1)-i] = operand[i];
end;

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.321 RCWCAS, RCWCASA, RCWCASAL, RCWCASL

Read check write compare and swap doubleword in memory

This instruction reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the compare fails or the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWCASA` and `RCWCASAL` load from memory with acquire semantics.
  - `RCWCASL` and `RCWCASAL` store to memory with release semantics.
  - `RCWCAS` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29					24	23	22	21	20				16	15					10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	0	Rn				Rt			
	S																												

Encoding for the RCWCAS variant

Applies when `(A == 0 && R == 0)`

```
RCWCAS  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCASA variant

Applies when `(A == 1 && R == 0)`

```
RCWCASA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCASAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCASAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCASL variant

Applies when `(A == 0 && R == 1)`

```
RCWCASL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1' && s != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
let newdata : bits(64) = X{}(t);
let compdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(s) = readdata;   // Return the old value when s!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.322 RCWCASP, RCWCASPA, RCWCASPAL, RCWCASPL

Read check write compare and swap quadword in memory

This instruction reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the compare fails or the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- `RCWCASPA` and `RCWCASPAL` load from memory with acquire semantics.
  - `RCWCASPL` and `RCWCASPAL` store to memory with release semantics.
  - `RCWCASP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20				16	15					10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	1	Rn				Rt			
	S																												

Encoding for the RCWCASP variant

Applies when `(A == 0 && R == 0)`

```
RCWCASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWCASPA variant

Applies when `(A == 1 && R == 0)`

```
RCWCASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWCASPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWCASPL variant

Applies when `(A == 0 && R == 1)`

```
RCWCASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rs[0] == '1' then EndOfDecode(Decode_UNDEF); end;
if Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let soft : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var newdata : bits(128);
var compdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let s1 : bits(64) = X{}(s);
let s2 : bits(64) = X{}(s+1);
let t1 : bits(64) = X{}(t);
let t2 : bits(64) = X{}(t+1);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                  tagchecked, t, t+1, s, s+1);

compdata = if BigEndian(accdesc.acctype) then s1::s2 else s2::s1;
newdata  = if BigEndian(accdesc.acctype) then t1::t2 else t2::t1;

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if BigEndian(accdesc.acctype) then
    X{64}(s)   = readdata[127:64];
    X{64}(s+1) = readdata[63:0];
else
    X{64}(s)   = readdata[63:0];
    X{64}(s+1) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.323 RCWCLR, RCWCLRA, RCWCLRAL, RCWCLRL

Read check write atomic bit clear on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. If the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks.

- If the destination register is not `XZR`, `RCWCLRA` and `RCWCLRAL` load from memory with acquire semantics.
  - `RCWCLRL` and `RCWCLRAL` store to memory with release semantics.
  - `RCWCLR` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	0	1	1	1	0	0	0	A	R	1	Rs	1	0	0	1	0	0	Rn	Rt	
	S				VR							o3		opc						

Encoding for the RCWCLR variant

Applies when `(A == 0 && R == 0)`

```
RCWCLR  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCLRA variant

Applies when `(A == 1 && R == 0)`

```
RCWCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCLRAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWCLRL variant

Applies when `(A == 0 && R == 1)`

```
RCWCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1' && t != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;

var address : bits(64);
let newdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let compdata : bits(64) = ARBITRARY : bits(64);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(t) = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.324 RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL

Read check write atomic bit clear on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory.
Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. If the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks.

- `RCWCLRPA` and `RCWCLRPAL` load from memory with acquire semantics.
  - `RCWCLRPL` and `RCWCLRPAL` store to memory with release semantics.
  - `RCWCLRP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

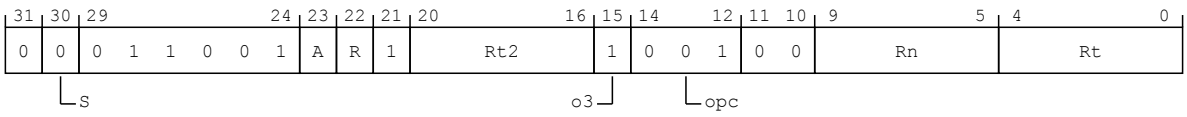

Encoding for the RCWCLRP variant

Applies when `(A == 0 && R == 0)`

```
RCWCLRP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWCLRPA variant

Applies when `(A == 1 && R == 0)`

```
RCWCLRPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWCLRPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWCLRPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWCLRPL variant

Applies when `(A == 0 && R == 1)`

```
RCWCLRPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var value1 : bits(64);
var value2 : bits(64);
var newdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                  tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

value1 = X{64}(t);
value2 = X{64}(t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
newdata = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let compdata : bits(128) = ARBITRARY : bits(128);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if rt_unknown then
    readdata = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t) = readdata[127:64];
    X{64}(t2) = readdata[63:0];
else
    X{64}(t) = readdata[63:0];
    X{64}(t2) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.325 RCWSCAS, RCWSCASA, RCWSCASAL, RCWSCASL

Read check write software compare and swap doubleword in memory

This instruction reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the compare fails, the RCW Checks fail, or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- If the destination register is not `XZR`, `RCWSCASA` and `RCWSCASAL` load from memory with acquire semantics.
  - `RCWSCASL` and `RCWSCASAL` store to memory with release semantics.
  - `RCWSCAS` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29					24	23	22	21	20				16	15					10	9			5	4			0
0	1	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	0	Rn				Rt			
	S																												

Encoding for the RCWSCAS variant

Applies when `(A == 0 && R == 0)`

```
RCWSCAS  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCASA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCASA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCASAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCASAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCASL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCASL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1' && s != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
let newdata : bits(64) = X{}(t);
let compdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(s) = readdata;   // Return the old value when s!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.326 RCWSCASP, RCWSCASPA, RCWSCASPAL, RCWSCASPL

Read check write software compare and swap quadword in memory

This instruction reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the compare fails, the RCW Checks fail, or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSCASPA` and `RCWSCASPAL` load from memory with acquire semantics.
  - `RCWSCASPL` and `RCWSCASPAL` store to memory with release semantics.
  - `RCWSCASP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20	16	15					10	9	5	4	0
0	1	0	1	1	0	0	1	A	R	1	Rs		0	0	0	0	1	1	Rn		Rt	

S

Encoding for the RCWSCASP variant

Applies when `(A == 0 && R == 0)`

```
RCWSCASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWSCASPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWSCASPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Encoding for the RCWSCASPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rs[0] == '1' then EndOfDecode(Decode_UNDEF); end;
if Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let soft : boolean = TRUE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<X(s+1)>

Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>

Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<X(t+1)>

Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var newdata : bits(128);
var compdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let s1 : bits(64) = X{}(s);
let s2 : bits(64) = X{}(s+1);
let t1 : bits(64) = X{}(t);
let t2 : bits(64) = X{}(t+1);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_CAS, soft, acquire, release,
                                                  tagchecked, t, t+1, s, s+1);

compdata = if BigEndian(accdesc.acctype) then s1::s2 else s2::s1;
newdata  = if BigEndian(accdesc.acctype) then t1::t2 else t2::t1;

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if BigEndian(accdesc.acctype) then
    X{64}(s)   = readdata[127:64];
    X{64}(s+1) = readdata[63:0];
else
    X{64}(s)   = readdata[63:0];
    X{64}(s+1) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.327 RCWSCLR, RCWSCLRA, RCWSCLRAL, RCWSCLRL

Read check write software atomic bit clear on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. If the RCW Checks fail or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks and RCWS Checks.

- If the destination register is not `XZR`, `RCWSCLRA` and `RCWSCLRAL` load from memory with acquire semantics.
  - `RCWSCLRL` and `RCWSCLRAL` store to memory with release semantics.
  - `RCWSCLR` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

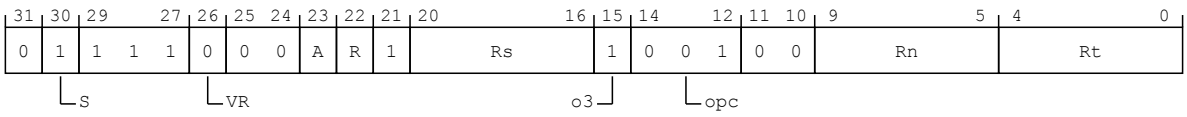

Encoding for the RCWSCLR variant

Applies when `(A == 0 && R == 0)`

```
RCWSCLR  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCLRA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCLRA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCLRAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCLRAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSCLRL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCLRL  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1' && t != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;

var address : bits(64);
let newdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let compdata : bits(64) = ARBITRARY : bits(64);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(t) = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.328 RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL

Read check write software atomic bit clear on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. If the RCW Checks fail or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks and RCWS Checks.

- `RCWSCLRPA` and `RCWSCLRPAL` load from memory with acquire semantics.
  - `RCWSCLRPL` and `RCWSCLRPAL` store to memory with release semantics.
  - `RCWSCLRP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

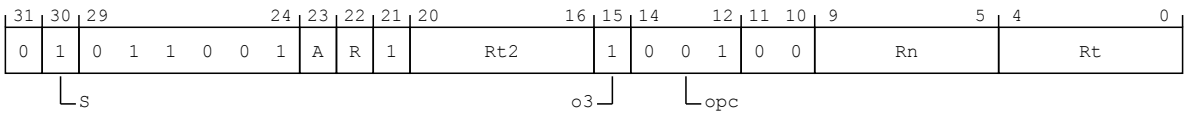

Encoding for the RCWSCLRP variant

Applies when `(A == 0 && R == 0)`

```
RCWSCLRP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSCLRPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSCLRPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSCLRPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSCLRPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSCLRPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSCLRPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var value1 : bits(64);
var value2 : bits(64);
var newdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_BIC, soft, acquire, release,
                                                  tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

value1 = X{64}(t);
value2 = X{64}(t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
newdata = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let compdata : bits(128) = ARBITRARY : bits(128);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if rt_unknown then
    readdata = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t) = readdata[127:64];
    X{64}(t2) = readdata[63:0];
else
    X{64}(t) = readdata[63:0];
    X{64}(t2) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.329 RCWSET, RCWSETA, RCWSETAL, RCWSETL

Read check write atomic bit set on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. If the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks.

- If the destination register is not `XZR`, `RCWSETA` and `RCWSETAL` load from memory with acquire semantics.
  - `RCWSETL` and `RCWSETAL` store to memory with release semantics.
  - `RCWSET` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

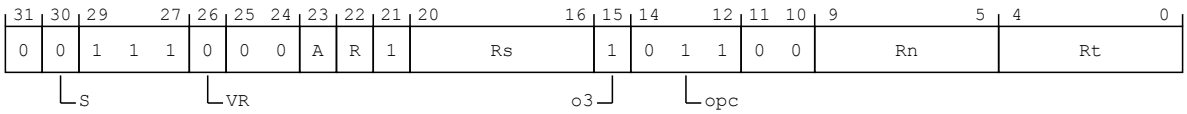

Encoding for the RCWSET variant

Applies when `(A == 0 && R == 0)`

```
RCWSET  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSETA variant

Applies when `(A == 1 && R == 0)`

```
RCWSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSETAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSETL variant

Applies when `(A == 0 && R == 1)`

```
RCWSETL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1' && t != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;

var address : bits(64);
let newdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let compdata : bits(64) = ARBITRARY : bits(64);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(t) = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.330 RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL

Read check write atomic bit set on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. If the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks.

- `RCWSETPA` and `RCWSETPAL` load from memory with acquire semantics.
  - `RCWSETPL` and `RCWSETPAL` store to memory with release semantics.
  - `RCWSETP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9				5	4				0
0	0	0	1	1	0	0	1	A	R	1	Rt2				1	0	1	1		0	0	Rn					Rt				
	S														o3		opc														

Encoding for the RCWSETP variant

Applies when `(A == 0 && R == 0)`

```
RCWSETP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSETPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSETPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSETPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSETPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSETPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSETPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var value1 : bits(64);
var value2 : bits(64);
var newdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                  tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

value1 = X{64}(t);
value2 = X{64}(t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
newdata = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let compdata : bits(128) = ARBITRARY : bits(128);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if rt_unknown then
    readdata = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t) = readdata[127:64];
    X{64}(t2) = readdata[63:0];
else
    X{64}(t) = readdata[63:0];
    X{64}(t2) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.331 RCWSSET, RCWSSETA, RCWSSETAL, RCWSSETL

Read check write software atomic bit set on doubleword in memory

This instruction atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. If the RCW Checks fail or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks and RCWS Checks.

- If the destination register is not `XZR`, `RCWSSETA` and `RCWSSETAL` load from memory with acquire semantics.
  - `RCWSSETL` and `RCWSSETAL` store to memory with release semantics.
  - `RCWSSET` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	1	0	1	1	0	0	Rn	Rt	
	S				VR							o3	opc							

Encoding for the RCWSSET variant

Applies when `(A == 0 && R == 0)`

```
RCWSSET  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSETA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSETA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSETAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSETAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSETL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSETL  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1' && t != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;

var address : bits(64);
let newdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let compdata : bits(64) = ARBITRARY : bits(64);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(t) = readdata;    // Return the old value when t!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.332 RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL

Read check write software atomic bit set on quadword in memory

This instruction atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. If the RCW Checks fail or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks and RCWS Checks.

- `RCWSSETPA` and `RCWSSETPAL` load from memory with acquire semantics.
  - `RCWSSETPL` and `RCWSSETPAL` store to memory with release semantics.
  - `RCWSSETP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	1	0	1	1	0	0	1	A	R	1	Rt2				1	0	1	1		0	0	Rn				Rt			
	S														o3		opc												

Encoding for the RCWSSETP variant

Applies when `(A == 0 && R == 0)`

```
RCWSSETP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSETPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSETPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSETPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSETPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSETPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSETPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var value1 : bits(64);
var value2 : bits(64);
var newdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_ORR, soft, acquire, release,
                                                  tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

value1 = X{64}(t);
value2 = X{64}(t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
newdata = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let compdata : bits(128) = ARBITRARY : bits(128);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if rt_unknown then
    readdata = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t) = readdata[127:64];
    X{64}(t2) = readdata[63:0];
else
    X{64}(t) = readdata[63:0];
    X{64}(t2) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.333 RCWSSWP, RCWSSWPA, RCWSSWPAL, RCWSSWPL

Read check write software swap doubleword in memory

This instruction atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. If the RCW Checks fail or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks and RCWS Checks.

- If the destination register is not `XZR`, `RCWSSWPA` and `RCWSSWPAL` load from memory with acquire semantics.
  - `RCWSSWPL` and `RCWSSWPAL` store to memory with release semantics.
  - `RCWSSWP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

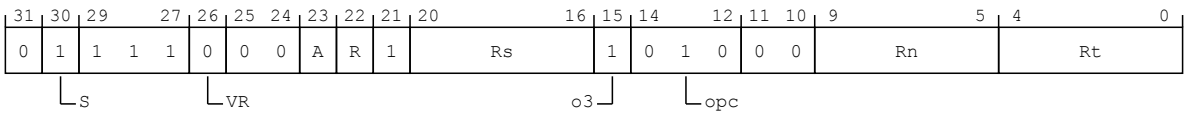

Encoding for the RCWSSWP variant

Applies when `(A == 0 && R == 0)`

```
RCWSSWP  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSWPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSWPA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSWPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSWPAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSSWPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSWPL  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1' && t != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
let newdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let compdata : bits(64) = ARBITRARY : bits(64);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(t) = readdata;   // Return the old value when t!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.334 RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL

Read check write software swap quadword in memory

This instruction atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. If the RCW Checks fail or the RCWS Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks and RCWS Checks.

- `RCWSSWPPA` and `RCWSSWPPAL` load from memory with acquire semantics.
  - `RCWSSWPPL` and `RCWSSWPPAL` store to memory with release semantics.
  - `RCWSSWPP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

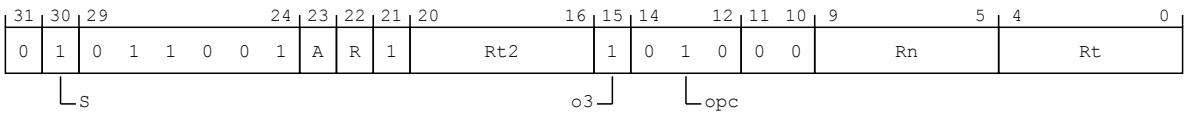

Encoding for the RCWSSWPP variant

Applies when `(A == 0 && R == 0)`

```
RCWSSWPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSWPPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSSWPPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSWPPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSSWPPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSSWPPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSSWPPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let soft : boolean = TRUE;

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var value1 : bits(64);
var value2 : bits(64);
var newdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                  tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

value1 = X{64}(t);
value2 = X{64}(t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
newdata = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let compdata : bits(128) = ARBITRARY : bits(128); // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if rt_unknown then
    readdata = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t)  = readdata[127:64];
    X{64}(t2) = readdata[63:0];
else
    X{64}(t)  = readdata[63:0];
    X{64}(t2) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.335 RCWSWP, RCWSWPA, RCWSWPAL, RCWSWPL

Read check write swap doubleword in memory

This instruction atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. If the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks.

- If the destination register is not `XZR`, `RCWSWPA` and `RCWSWPAL` load from memory with acquire semantics.
  - `RCWSWPL` and `RCWSWPAL` store to memory with release semantics.
  - `RCWSWP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_THE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	0	1	1	1	0	0	A	R	1	Rs		1	0	1	0	0	0	Rn	Rt	
	S				VR							o3		opc						

Encoding for the RCWSWP variant

Applies when `(A == 0 && R == 0)`

```
RCWSWP  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSWPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSWPA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSWPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSWPAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the RCWSWPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSWPL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1' && t != 31;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
let newdata : bits(64) = X{}(s);
var readdata : bits(64);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                  tagchecked, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let compdata : bits(64) = ARBITRARY : bits(64);    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{64}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
X{64}(t) = readdata;   // Return the old value when t!=31
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.336 RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL

Read check write swap quadword in memory

This instruction atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. If the RCW Checks fail, the architecture permits writing the value read from the location to memory. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the RCW Checks.

- `RCWSWPPA` and `RCWSWPPAL` load from memory with acquire semantics.
  - `RCWSWPPL` and `RCWSWPPAL` store to memory with release semantics.
  - `RCWSWPP` has neither acquire nor release semantics.

Note

This instruction is for performing atomic updates of translation table entries and not for general use.

Integer

(FEAT_D128 && FEAT_THE)

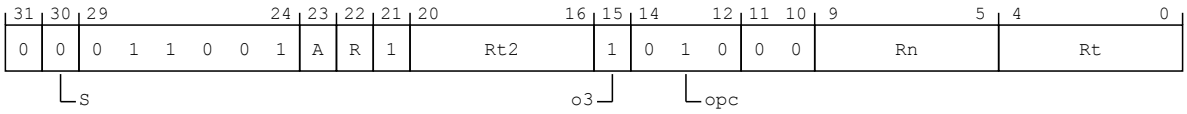

Encoding for the RCWSWPP variant

Applies when `(A == 0 && R == 0)`

```
RCWSWPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSWPPA variant

Applies when `(A == 1 && R == 0)`

```
RCWSWPPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSWPPAL variant

Applies when `(A == 1 && R == 1)`

```
RCWSWPPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the RCWSWPPL variant

Applies when `(A == 0 && R == 1)`

```
RCWSWPPL  <Xt1>, <Xt2>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then
    EndOfDecode(Decode_UNDEF);
end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let soft : boolean = FALSE;

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then Undefined(); end;
var address : bits(64);
var value1 : bits(64);
var value2 : bits(64);
var newdata : bits(128);
var readdata : bits(128);
var nzcv : bits(4);

let accdesc : AccessDescriptor = CreateAccDescRCW(MemAtomicOp_SWP, soft, acquire, release,
                                                  tagchecked, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

value1 = X{64}(t);
value2 = X{64}(t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
newdata = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let compdata : bits(128) = ARBITRARY : bits(128); // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW{128}(address, compdata, newdata, accdesc);

PSTATE.[N,Z,C,V] = nzcv;
if rt_unknown then
    readdata = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t)  = readdata[127:64];
    X{64}(t2) = readdata[63:0];
else
    X{64}(t)  = readdata[63:0];
    X{64}(t2) = readdata[127:64];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.337 RET

Return from subroutine

This instruction branches unconditionally to an address in a register. This instruction provides a hint that this is a subroutine return.

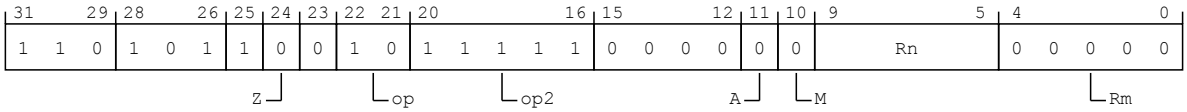

Encoding

```
RET  {<Xn>}
```

Decode for this encoding

```
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field. Defaults to X30 if absent.

Operation

```
var target : bits(64) = X{}(n);
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    target = LoadCheckGCSRecord(target, GCSInstType_PRET);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);
end;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.338 RETAA, RETAB

Return from subroutine, with pointer authentication

This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, and branches to the authenticated address.

Key A is used for `RETAA`. Key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

If FEAT_PAuth_LR is implemented and PSTATE.PACM is 1, then `RETAA` and `RETAB` include a second modifier that is in X16.

Integer

(FEAT_PAuth)

31						25	24	23	22	21	20				16	15			12	11	10	9				5	4				0
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
							Z			op			op2							A					Rn					Rm	

Encoding for the RETAA variant

Applies when `(M == 0)`

```
RETAA
```

Encoding for the RETAB variant

Applies when `(M == 1)`

```
RETAB
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;

let use_key_a : boolean = (M == '0');
let auth_then_branch : boolean = TRUE;
```

Operation

```
var inst_type : GCSInstruction;
var target : bits(64) = X{}(30);

let modifier : bits(64) = SP{}();
var modifier2 : bits(64);
var use_modifier2 : boolean = FALSE;
if IsFeatureImplemented(FEAT_PAuth_LR) && PSTATE.PACM == '1' then
    modifier2 = X{64}(16);
    use_modifier2 = TRUE;
end;

if use_key_a then
    if use_modifier2 && IsFeatureImplemented(FEAT_PAuth_LR) then
        target = AuthIA2(target, modifier, modifier2, auth_then_branch);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        target = AuthIA(target, modifier, auth_then_branch);
    end;
else
    if use_modifier2 && IsFeatureImplemented(FEAT_PAuth_LR) then
        target = AuthIB2(target, modifier, modifier2, auth_then_branch);
    else
        target = AuthIB(target, modifier, auth_then_branch);
    end;
end;

if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);
end;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.339 RETAASPPC, RETABSPPC

Return from subroutine, with enhanced pointer authentication using an immediate offset

This instruction authenticates the address that is held in LR, using SP as the first modifier, the specified immediate subtracted from PC as the second modifier, and the specified key, and branches to the authenticated address.

Key A is used for `RETAASPPC`. Key B is used for `RETABSPPC`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

Integer

(FEAT_PAuth_LR)

31							24	23		21	20												5	4				0
0	1	0	1	0	1	0	1	0	0	x	imm16													1	1	1	1	1
										opc																		op2

Encoding for the RETAASPPC variant

Applies when `(opc == 000)`

```
RETAASPPC  <label>
```

Encoding for the RETABSPPC variant

Applies when `(opc == 001)`

```
RETABSPPC  <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;

let use_key_a : boolean = opc[0] == '0';
let offset : bits(64) = ZeroExtend{}(imm16::'00');
let auth_then_branch : boolean = TRUE;
```

Assembler Symbols

<label>

Is the program label whose address is to be calculated. Its negative offset from the address of this instruction, a multiple of 4 in the range -262140 to 0, is encoded as an unsigned value in the “imm16” field as <label>/4.

Operation

```
var inst_type : GCSInstruction;
var target : bits(64) = X{}(30);

let modifier : bits(64) = SP{}();
let modifier2 : bits(64) = PC64() - offset;

if use_key_a then
    target = AuthIA2(target, modifier, modifier2, auth_then_branch);
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    target = AuthIB2(target, modifier, modifier2, auth_then_branch);
end;

if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);
end;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.340 RETAASPPCR, RETABSPPCR

Return from subroutine, with enhanced pointer authentication using a register

This instruction authenticates the address that is held in LR, using SP as the first modifier, the value in the specified register as the second modifier, and the specified key, and branches to the authenticated address.

Key A is used for `RETAASPPCR`. Key B is used for `RETABSPPCR`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

Integer

(FEAT_PAuth_LR)

31–25	24–21	20–16	15–11	10	9–5	4–0
1 1 0 1 0 1 1	0 0 1 0	1 1 1 1 1	0 0 0 0 1	M	1 1 1 1 1	!=11111
	opc	op2			Rn	Rm

Encoding for the RETAASPPCR variant

Applies when `(M == 0)`

```
RETAASPPCR  <Xm>
```

Encoding for the RETABSPPCR variant

Applies when `(M == 1)`

```
RETABSPPCR  <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth_LR) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Rm);
let use_key_a : boolean = M == '0';
let auth_then_branch : boolean = TRUE;
```

Assembler Symbols

<Xm>

Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

```
var inst_type : GCSInstruction;
var target : bits(64) = X{}(30);

let modifier : bits(64) = SP{}();
let modifier2 : bits(64) = X{}(m);

if use_key_a then
    target = AuthIA2(target, modifier, modifier2, auth_then_branch);
else
    target = AuthIB2(target, modifier, modifier2, auth_then_branch);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);
end;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

let branch_conditional : boolean = FALSE;
BranchTo{64}(target, BranchType_RET, branch_conditional);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.341 REV

Reverse bytes

This instruction reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

31	30	29	28							21	20					16	15				12	11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0		0	0	0	0		1	x	Rn					Rd				
		S									opcode2											opc											

Encoding for the 32-bit variant

Applies when `(sf == 0 && opc == 10)`

```
REV  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && opc == 11)`

```
REV  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let container_size : integer{} = 8 << UInt(opc);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand : bits(datasize) = X{}(n);
var result : bits(datasize);

let containers : integer = datasize DIV container_size;
for c = 0 to containers-1 do
    let container : bits(container_size) = operand[c*:container_size];
    result[c*:container_size] = Reverse{container_size}(container, 8);
end;

X{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.342 REV16

Reverse bytes in 16-bit halfwords

This instruction reverses the byte order in each 16-bit halfword of a register.

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	Rn					Rd				
		S													opcode2							opc										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
REV16  <Wd>, <Wn>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
REV16  <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let container_size : integer{} = 8 << UInt(opc);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand : bits(datasize) = X{}(n);
var result : bits(datasize);

let containers : integer = datasize DIV container_size;
for c = 0 to containers-1 do
    let container : bits(container_size) = operand[c*:container_size];
    result[c*:container_size] = Reverse{container_size}(container, 8);
end;

X{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.343 REV32

Reverse bytes in 32-bit words

This instruction reverses the byte order in each 32-bit word of a register.

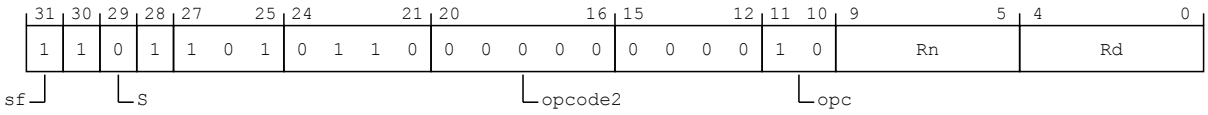

Encoding

```
REV32  <Xd>, <Xn>
```

Decode for this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let container_size : integer{} = 8 << UInt(opc);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand : bits(datasize) = X{}(n);
var result : bits(datasize);

let containers : integer = datasize DIV container_size;
for c = 0 to containers-1 do
    let container : bits(container_size) = operand[c*:container_size];
    result[c*:container_size] = Reverse{container_size}(container, 8);
end;

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.344 REV64

Reverse bytes

This instruction reverses the byte order in a 64-bit general-purpose register.

When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

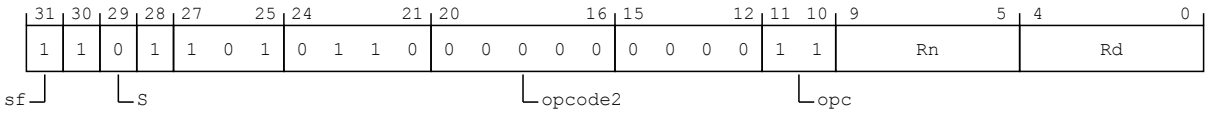

Encoding

```
REV64  <Xd>, <Xn>
```

is equivalent to

```
REV  <Xd>, <Xn>
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational Information

The description of REV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.345 RMIF

Rotate, mask insert flags

This instruction performs a rotation right of a value held in a general-purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer

(FEAT_FlagM)

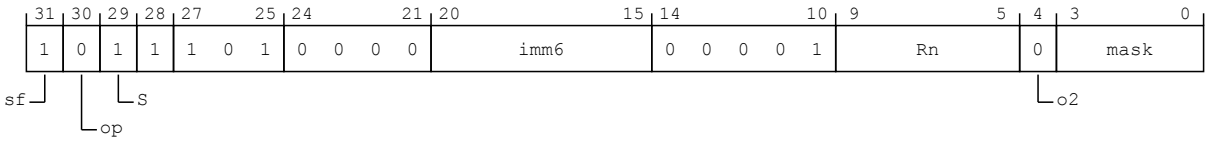

Encoding

```
RMIF  <Xn>, #<shift>, #<mask>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF); end;
let imm : integer{} = UInt(imm6);
let flagmask : bits(4) = mask;
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<shift>

Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<mask>

Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the “mask” field.

Operation

```
let reg : bits(64) = X{}(n);
let flags : bits(4) = (reg::reg)[imm+3:imm];
if flagmask[3] == '1' then PSTATE.N = flags[3]; end;
if flagmask[2] == '1' then PSTATE.Z = flags[2]; end;
if flagmask[1] == '1' then PSTATE.C = flags[1]; end;
if flagmask[0] == '1' then PSTATE.V = flags[0]; end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.346 ROR (immediate)

Rotate right (immediate)

This instruction provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21	20 … 16	15 … 10	9 … 5	4 … 0
sf	0 0	1 0 0 1 1 1	N	0	Rm	imms	Rn	Rd
	op21			o0				

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0 && imms == 0xxxxx)`

```
ROR  <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR  <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
ROR  <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR  <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Ws>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

<shift>

For the “32-bit” variant: is the amount by which to rotate, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the amount by which to rotate, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xs>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational Information

The description of EXTR gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.347 ROR (register)

Rotate right (register)

This instruction provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

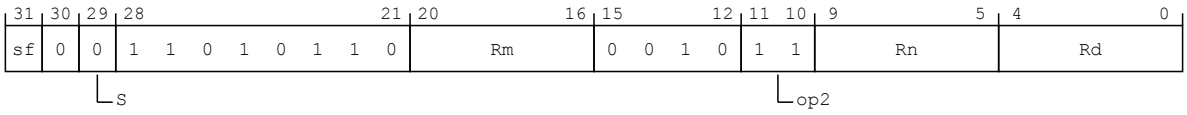

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
ROR  <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV  <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
ROR  <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV  <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational Information

The description of RORV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.348 RORV

Rotate right variable

This instruction provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).

31	30	29	28							21	20				16	15				12	11	10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0		1	1	Rn					Rd				
		S																				op2										

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
RORV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
RORV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(op2);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
ROR (register)	Unconditionally

Operation

```
let operand2 : bits(datasize) = X{}(m);

X{datasize}(d) = ShiftReg{datasize}(n, shift_type, UInt(operand2) MOD datasize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.349 RPRFM

Range prefetch memory

This instruction signals the memory system that data memory accesses from a specified range of addresses are likely to occur in the near future. The instruction may also signal the memory system about the likelihood of data reuse of the specified range of addresses. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as prefetching locations within the specified address ranges into one or more caches. The memory system may also exploit the data reuse hints to decide whether to retain the data in other caches upon eviction from the innermost caches or to discard it.

The effect of an `RPRFM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

An `RPRFM` instruction specifies the type of accesses and range of addresses using the following parameters:

- ‘Access type’, encoded in `Rt<0>`, specifies:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- ‘Policy’, encoded in `Rt<2:1>`, specifies:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once. If a streaming prefetch is specified, then the ‘ReuseDistance’ parameter is ignored.
- ‘BaseAddress’, in the 64-bit base register, holds the initial block address for the accesses.
- ‘ReuseDistance’, in the metadata register bits[63:60], indicates the maximum number of bytes to be accessed by this PE before executing the next `RPRFM` instruction that specifies the same range. This includes the total number of bytes inside and outside of the range that will be accessed by the same PE. This parameter can be used to influence cache eviction and replacement policies, in order to retain the data in the most optimal levels of the memory hierarchy after each access. If software cannot easily determine the amount of other memory that will be accessed, these bits can be set to zero to indicate that ‘ReuseDistance’ is not known. Otherwise, these four bits encode decreasing powers of two in the range 512MiB (`0b0001`) to 32KiB (`0b1111`).
- ‘Stride’, in the metadata register bits[59:38], is a signed, two’s complement integer encoding of the number of bytes to advance the block address after ‘Length’ bytes have been accessed, in the range -2MiB to +2MiB-1B. A negative value indicates that the block address is advanced in a descending direction.
- ‘Count’, in the metadata register bits[37:22], is an unsigned integer encoding of the number of blocks of data to be accessed minus 1, representing the range 1 to 65536 blocks. If ‘Count’ is 0, then the ‘Stride’ parameter is ignored and only a single block of contiguous bytes from ‘BaseAddress’ to (‘BaseAddress’ + ‘Length’ - 1) is described.
- ‘Length’, in the metadata register bits[21:0], is a signed, two’s complement integer encoding of the number of contiguous bytes to be accessed starting from the current block address, without changing the block address, in the range -2MiB to +2MiB-1B. A negative value indicates that the bytes are accessed in a descending direction.

Note

Software is expected to honor the parameters it provides to the `RPRFM` instruction, and the same PE should access all locations in the range, in the direction specified by the sign of the ‘Length’ and ‘Stride’ parameters. A range prefetch is considered active on a PE until all locations in the range have been accessed by the PE. A range prefetch might also be inactivated by the PE prior to completion, for example due to a software context switch or lack of hardware resources. Software should not specify overlapping addresses in multiple active ranges. If a range is expected to be accessed by both load and store instructions (read-modify-write), then a single range with an ‘Access type’ parameter of `PST` (prefetch for store) should be specified.

Integer

(FEAT_RPRFM)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	Rm	x	1	x	S	1	0	Rn	1	1	x	x	x
size					VR			opc				option							Rt				

Encoding

```
RPRFM   (<rprfop>|#<imm6>), <Xm>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RPRFM) then EndOfDecode(Decode_NOP); end;
let operation : bits(6) = option[2]::option[0]::S::Rt[2:0];
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
```

Assembler Symbols

<rprfop>

Is the range prefetch operation, encoded in “option:S:Rt”:

option	S	Rt	<rprfop>
010	0	11000	PLDKEEP
010	0	11001	PSTKEEP
010	0	11100	PLDSTRM
010	0	11101	PSTSTRM

For other encodings, use <imm6>.

<imm6>

Is the range prefetch operation encoding as an immediate, in the range 0 to 63, encoded in “option<2>:option<0>:S:Rt<2:0>”. This syntax is only for encodings that are not representable using `<rprfop>`.

<Xm>

Is the 64-bit name of the general-purpose register that holds an encoding of the metadata, encoded in the “Rm” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
let address : bits(64)  = if n == 31 then SP{64}() else X{64}(n);
let metadata : bits(64) = X{}(m);
let stride : integer = SInt(metadata[59:38]);
let count : integer  = UInt(metadata[37:22]) + 1;
let length : integer = SInt(metadata[21:0]);
var reuse : integer;

if metadata[63:60] == '0000' then
    reuse = -1; // Not known
else
    reuse = 32768 << (15 - UInt(metadata[63:60]));
end;

Hint_RangePrefetch(address, length, stride, count, reuse, operation);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.350 SB

Speculation barrier

This instruction is a barrier that controls speculation. For more information and details of the semantics, see Speculation Barrier (SB).

System

(FEAT_SB)

31 29	28 26	25 12	11 8	7	6 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	(0) (0) (0) (0)	1	1 1	1 1 1 1 1
			CRm		opc	Rt

Encoding

```
SB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SB) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
SpeculationBarrier();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.351 SBC

Subtract with carry

This instruction subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0		Rn					Rd				
	op	S																														

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SBC  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SBC  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
NGC	Rn == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = NOT(X{datasize}(m));
var result : bits(datasize);

(result, -) = AddWithCarry{datasize}(operand1, operand2, PSTATE.C);

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.352 SBCS

Subtract with carry, setting flags

This instruction subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	1	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0		Rn					Rd				
	op																															
		S																														

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SBCS  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SBCS  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
NGCS	Rn == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = NOT(X{datasize}(m));
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, PSTATE.C);

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.353 SBFIZ

Signed bitfield insert in zeros

This instruction copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SBFIZ  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SBFIZ  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.354 SBFM

Signed bitfield move

This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd

opc

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SBFM  <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SBFM  <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && (N != '0' || immr[5] != '0' || imms[5] != '0') then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let s : integer = UInt(imms);
let r : integer = UInt(immr);

var wmask : bits(datasize);
var tmask : bits(datasize);
(wmask, tmask) = DecodeBitMasks{datasize}(N, imms, immr, FALSE);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<immr>

For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

<imms>

For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Of variant	Is preferred when
ASR (immediate)	32-bit	imms == '011111'
ASR (immediate)	64-bit	imms == '111111'
SBFIZ		UInt(imms) < UInt(immr)
SBFX		BFXPreferred(sf, opc<1>, imms, immr)
SXTB		immr == '000000' && imms == '000111'
SXTH		immr == '000000' && imms == '001111'
SXTW		immr == '000000' && imms == '011111'

Operation

```
let src : bits(datasize) = X{}(n);

// Perform bitfield move on low bits
let bot : bits(datasize) = ROR(src, r) AND wmask;

let top : bits(datasize) = Replicate{}(src[s]);

// Combine extension bits and result bits
X{datasize}(d) = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.355 SBFX

Signed bitfield extract

This instruction copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	0 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd
	opc						

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SBFX  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SBFX  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.356 SDIV (quotient)

Signed divide

This instruction divides the first signed source register value by the second signed source register value, and writes the result to the destination register. Dividing by zero writes the value zero to the destination register. The condition flags are not affected.

31	30	29	28	21	20	16	15	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 0 0 1		1	Rn		Rd	
		S							o1				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SDIV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SDIV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let dividend : integer = SInt(operand1);
let divisor : integer  = SInt(operand2);
var result : integer;
if divisor == 0 then
    result = 0;
elsif (dividend < 0) == (divisor < 0) then
    result = Abs(dividend) DIVRM Abs(divisor);    // same signs - positive result
else
    result = -(Abs(dividend) DIVRM Abs(divisor)); // different signs - negative result
end;
X{datasize}(d) = result[datasize-1:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.357 SETF8, SETF16

Evaluation of 8-bit or 16-bit flag values

This instruction sets the PSTATE.NZV flags based on the value in the specified general-purpose register. `SETF8` treats the value as an 8-bit value. `SETF16` treats the value as a 16-bit value.

The PSTATE.C flag is not affected by these instructions.

Integer

(FEAT_FlagM)

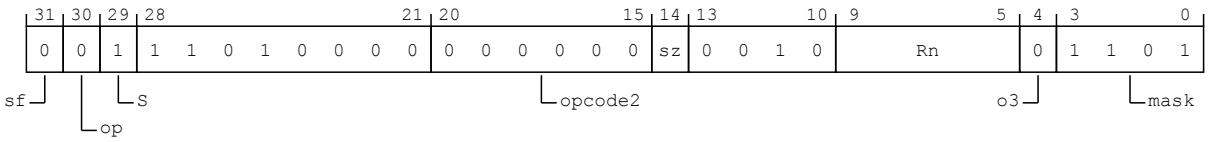

Encoding for the SETF8 variant

Applies when `(sz == 0)`

```
SETF8  <Wn>
```

Encoding for the SETF16 variant

Applies when `(sz == 1)`

```
SETF16  <Wn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = 8 << UInt(sz);
let n : integer = UInt(Rn);
```

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let reg : bits(32) = X{}(n);
PSTATE.N = reg[size-1];
PSTATE.Z = if (reg[size-1:0] == Zeros{size}) then '1' else '0';
PSTATE.V = reg[size] XOR reg[size-1];
//PSTATE.C unchanged;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.358 SETGP, SETGM, SETGE

Memory set with tag setting

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGM sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGE sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGP:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGP, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGP, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGM, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGM, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGE, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On completion of the instruction, Xn holds 0.

For SETGE, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS && FEAT_MTE)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	1 1	0	Rs	x x 0 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0000)`

```
SETGP  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0100)`

```
SETGM  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1000)`

```
SETGE  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);
end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';

case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

For the “Main” and “Prologue” variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the “Rs” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

memset.is_setg = TRUE;

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETGOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;
    end;

    if ((memset.setsize != 0 && !IsAlignedSize(memset.toaddress, TAG_GRANULE)) ||
            !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, memset.toaddress);
        AArch64_Abort(fault);
    end;

    if memset.implements_option_a then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    var fault_address : bits(64);
    if memset.implements_option_a then
        fault_address = memset.toaddress + memset.setsize;
    else
        fault_address = memset.toaddress;
    end;

    if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          !IsAlignedSize(memset.toaddress, TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
           !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
end;

var tags_set : integer;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var tagmemaddrdesc : AddressDescriptor;
var tagmemstatus : PhysMemRetStatus;
var fault : boolean    = FALSE;
var tagfault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= -1 * memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                  data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress + memset.setsize);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                              tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64_Abort(tagmemaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.359 SETGPN, SETGMN, SETGEN

Memory set with tag setting, non-temporal

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGMN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGEN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGPN:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGPN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGPN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGMN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGMN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGEN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

For SETGEN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS && FEAT_MTE)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	1 1	0	Rs	x x 1 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0010)`

```
SETGPN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0110)`

```
SETGMN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1010)`

```
SETGEN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);
end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';

case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

For the “Main” and “Prologue” variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the “Rs” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

memset.is_setg = TRUE;

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETGOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;
    end;

    if ((memset.setsize != 0 && !IsAlignedSize(memset.toaddress, TAG_GRANULE)) ||
            !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, memset.toaddress);
        AArch64_Abort(fault);
    end;

    if memset.implements_option_a then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    var fault_address : bits(64);
    if memset.implements_option_a then
        fault_address = memset.toaddress + memset.setsize;
    else
        fault_address = memset.toaddress;
    end;

    if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          !IsAlignedSize(memset.toaddress, TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
           !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
end;

var tags_set : integer;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var tagmemaddrdesc : AddressDescriptor;
var tagmemstatus : PhysMemRetStatus;
var fault : boolean    = FALSE;
var tagfault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= -1 * memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                  data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress + memset.setsize);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                              tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64_Abort(tagmemaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.360 SETGPT, SETGMT, SETGET

Memory set with tag setting, unprivileged

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGMT sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGET sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGPT:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGPT, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGPT, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGMT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGMT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGET, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

For SETGET, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS && FEAT_MTE)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	1 1	0	Rs	x x 0 1	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0001)`

```
SETGPT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0101)`

```
SETGMT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1001)`

```
SETGET  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);
end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

For the "Main" and "Prologue" variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

memset.is_setg = TRUE;

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETGOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescSTGMOPS(privileged, nontemporal);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;
    end;

    if ((memset.setsize != 0 && !IsAlignedSize(memset.toaddress, TAG_GRANULE)) ||
            !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, memset.toaddress);
        AArch64_Abort(fault);
    end;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    var fault_address : bits(64);
    if memset.implements_option_a then
        fault_address = memset.toaddress + memset.setsize;
    else
        fault_address = memset.toaddress;
    end;

    if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          !IsAlignedSize(memset.toaddress, TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
           !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
end;

var tags_set : integer;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var tagmemaddrdesc : AddressDescriptor;
var tagmemstatus : PhysMemRetStatus;
var fault : boolean    = FALSE;
var tagfault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= -1 * memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                  data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress + memset.setsize);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                              tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64_Abort(tagmemaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.361 SETGPTN, SETGMTN, SETGETN

Memory set with tag setting, unprivileged and non-temporal

These instructions set a requested number of bytes in memory to the value in the least significant byte of the source data register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register that holds the first address to be set. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETGMTN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETGETN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETGPTN:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.

On completion of SETGPTN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETGPTN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETGMTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETGMTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETGETN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

For SETGETN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS && FEAT_MTE)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	1	0 1	1 1	0	Rs	x x 1 1	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0011)`

```
SETGPTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0111)`

```
SETGMTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1011)`

```
SETGETN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then
    EndOfDecode(Decode_UNDEF);
end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

For the "Main" and "Prologue" variants: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

memset.is_setg = TRUE;

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETGOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescSTGMOPS(privileged, nontemporal);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSSETGSize then
        memset.setsize = ArchMaxMOPSSETGSize;
    end;

    if ((memset.setsize != 0 && !IsAlignedSize(memset.toaddress, TAG_GRANULE)) ||
            !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, memset.toaddress);
        AArch64_Abort(fault);
    end;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    var fault_address : bits(64);
    if memset.implements_option_a then
        fault_address = memset.toaddress + memset.setsize;
    else
        fault_address = memset.toaddress;
    end;

    if (memset.setsize != 0 && (memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          !IsAlignedSize(memset.toaddress, TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
           !IsAlignedSize(memset.setsize[63:0], TAG_GRANULE)) then
        let fault : FaultRecord = AlignmentFault(accdesc, fault_address);
        AArch64_Abort(fault);
    end;
end;

var tags_set : integer;
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var tagmemaddrdesc : AddressDescriptor;
var tagmemstatus : PhysMemRetStatus;
var fault : boolean    = FALSE;
var tagfault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= -1 * memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                  data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress + memset.setsize);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress + memset.setsize,
                                                              tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault && !tagfault looplimit ArchMaxMOPSSETGSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, TAG_GRANULE);
        assert B <= memset.stagesetsize && B[3:0] == '0000';

        (-, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);
        let tag : bits(4) = AArch64_AllocationTagFromAddress(memset.toaddress);
        (tags_set, tagmemaddrdesc, tagmemstatus) = MemSetTags(memset.toaddress, tag, B, accdesc);

        fault    = IsFault(memaddrdesc)    || IsFault(memstatus);
        tagfault = IsFault(tagmemaddrdesc) || IsFault(tagmemstatus);

        if !fault && !tagfault then
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault || tagfault, tags_set * TAG_GRANULE);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
elsif tagfault then
    if IsFault(tagmemaddrdesc) then
        AArch64_Abort(tagmemaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(tagmemstatus, iswrite, tagmemaddrdesc, 1, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.362 SETP, SETM, SETE

Memory set

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETM sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETE sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETP:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETP, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETP, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETM, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETM, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETE, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETE, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 0 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0000)`

```
SETP  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0100)`

```
SETM  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1000)`

```
SETE  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';

case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;
    end;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var memory_set : integer;
var fault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.363 SETPN, SETMN, SETEN

Memory set, non-temporal

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETMN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETEN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETPN:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETPN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETPN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETMN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETMN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETEN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETEN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 1 0	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0010)`

```
SETPN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0110)`

```
SETMN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1010)`

```
SETEN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';

case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the “Rd” field.

For the “Epilogue” and “Main” variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the “Rd” field.

<Xn>

For the “Prologue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Main” variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the “Rn” field.

For the “Epilogue” variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the “Rs” field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;
    end;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var memory_set : integer;
var fault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.364 SETPT, SETMT, SETET

Memory set, unprivileged

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETMT sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETET sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETPT:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETPT, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETPT, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETMT, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETMT, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETET, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETET, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 0 1	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0001)`

```
SETPT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0101)`

```
SETMT  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1001)`

```
SETET  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';

case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;
    end;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
end;

var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var memory_set : integer;
var fault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.365 SETPTN, SETMTN, SETETN

Memory set, unprivileged and non-temporal

These instructions set a required number of bytes in memory to the value in the least significant byte of the source data register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and sets an IMPLEMENTATION DEFINED portion of the requested number of bytes. SETMTN sets a further IMPLEMENTATION DEFINED portion of the remaining bytes. SETETN sets any final remaining bytes.

Note

The ability to set an IMPLEMENTATION DEFINED number of bytes allows an implementation to optimize how the bytes being set are divided between the different instructions.

For more information on exceptions specific to memory set instructions, see Memory Copy and Memory Set exceptions.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

For SETPTN:

- If Xn[63] == ‘1’, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.

On completion of SETPTN, option A:

- Xn holds -1 times the number of bytes in the saturated set size remaining to be set.
- Xd holds the original Xd + saturated set size.
- PSTATE.{N,Z,C,V} are set to {0,0,0,0}.

On completion of SETPTN, option B:

- Xn holds the number of bytes in the saturated set size remaining to be set.
- Xd holds the lowest address that has not been set.
- PSTATE.{N,Z,C,V} are set to {0,0,1,0}.

For SETMTN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds -1 times the number of bytes remaining to be set.

For SETMTN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds the number of bytes remaining to be set.
  - Xd holds the lowest address that has not been set.

For SETETN, option A, when PSTATE.C = ‘0’:

- Xn holds a signed 64-bit integer.
- Xn holds -1 times the number of bytes remaining to be set.
- Xd holds the lowest address to be set - Xn.
- On completion of the instruction, Xn holds 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SETETN, option B, when PSTATE.C = ‘1’:

- Xn holds the number of bytes remaining to be set.
- Xd holds the lowest address to be set.
- On completion of the instruction:
  - Xn holds 0.
  - Xd holds the lowest address that has not been set.

Explicit Memory Write effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is ‘11’.

Otherwise, the Explicit Memory Write effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_MOPS)

31 30	29 27	26	25 24	23 22	21	20 16	15 12	11 10	9 5	4 0
sz	0 1 1	0	0 1	1 1	0	Rs	x x 1 1	0 1	Rn	Rd
		o0		op1			op2			

Encoding for the Prologue variant

Applies when `(op2 == 0011)`

```
SETPTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Main variant

Applies when `(op2 == 0111)`

```
SETMTN  [<Xd>]!, <Xn>!, <Xs>
```

Encoding for the Epilogue variant

Applies when `(op2 == 1011)`

```
SETETN  [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then EndOfDecode(Decode_UNDEF); end;

var memset : SETParams;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
let options : bits(2) = op2[1:0];
let nontemporal : boolean = options[1] == '1';

case op2[3:2] of
    when '00' => memset.stage = MOPSStage_Prologue;
    when '01' => memset.stage = MOPSStage_Main;
    when '10' => memset.stage = MOPSStage_Epilogue;
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly Memory Copy and Memory Set SET* and Crossing a page boundary with different memory types or Shareability attributes.

Assembler Symbols

<Xd>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

For the "Epilogue" and "Main" variants: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

<Xn>

For the "Prologue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Main" variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the "Epilogue" variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero on completion of the instruction, encoded in the "Rn" field.

<Xs>

Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

CheckSETConstrainedUnpredictable(memset);

let data : bits(8) = X{}(memset.s);
var B : MOPSBlockSize = 0;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.[N,Z,C,V];
memset.toaddress = X{64}(memset.d);
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X{64}(memset.n));
else
    memset.setsize = SInt(X{64}(memset.n));
end;
memset.implements_option_a = SETOptionA();

let privileged : boolean = (if options[0] == '1' then AArch64_IsUnprivAccessPriv()
                            else PSTATE.EL != EL0);

let accdesc : AccessDescriptor = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > ArchMaxMOPSBlockSize then
        memset.setsize = ArchMaxMOPSBlockSize;
    end;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize   = 0 - memset.setsize;
    else
        memset.nzcv = '0010';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
end;

var memaddrdesc : AddressDescriptor;
var memstatus : PhysMemRetStatus;
var memory_set : integer;
var fault : boolean = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize,
                                                            data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize      = memset.setsize      + B;
            memset.stagesetsize = memset.stagesetsize + B;
        end;
    end;

else
    while memset.stagesetsize > 0 && !fault looplimit ArchMaxMOPSBlockSize do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress    = memset.toaddress    + B;
            memset.setsize      = memset.setsize      - B;
            memset.stagesetsize = memset.stagesetsize - B;
        end;
    end;
end;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    else
        let iswrite : boolean = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);
    end;
end;

if memset.stage == MOPSStage_Prologue then
    PSTATE.[N,Z,C,V] = memset.nzcv;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.366 SEV

Send event

This instruction is a hint instruction that causes an event to be signaled to all PEs in the multiprocessor system. For more information, see Wait for Event mechanism and Send event.

31 29	28 26	25 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	1 0 0 (op2)	1 1 1 1 1

Encoding

```
SEV
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEvent();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.367 SEVL

Send event local

This instruction is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop that starts with a `WFE` instruction.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	1 0 1 (op2)	1 1 1 1 1

Encoding

```
SEVL
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEventLocal();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.368 SMADDL

Signed multiply-add long

This instruction multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL (32-bit).

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	0	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
SMADDL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
SMULL (32-bit)	Ra == '11111'

Operation

```
let operand1 : bits(32) = X{}(n);
let operand2 : bits(32) = X{}(m);
let operand3 : bits(64) = X{}(a);

let result : integer = SInt(operand3) + (SInt(operand1) * SInt(operand2));

X{64}(d) = result[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.369 SMAX (immediate)

Signed maximum (immediate)

This instruction determines the signed maximum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	22	21	18	17	10	9	5	4	0
sf	0	0	1 0 0 0 1 1 1		0 0 0 0		imm8		Rn		Rd	
	op	S				opc						

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMAX  <Wd>, <Wn>, #<simm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMAX  <Xd>, <Xn>, #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let imm : integer = SInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<simm>

Is a signed immediate, in the range -128 to 127, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = Max(SInt(operand1), imm);
X{datasize}(d) = result[datasize-1:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.370 SMAX (register)

Signed maximum (register)

This instruction determines the signed maximum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

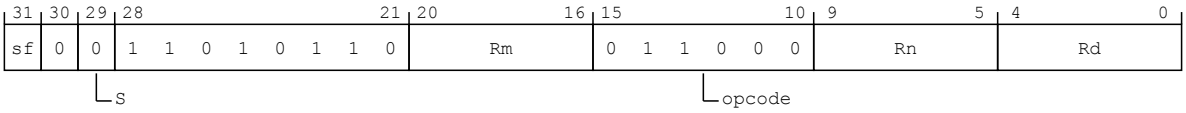

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMAX  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMAX  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let result : integer = Max(SInt(operand1), SInt(operand2));
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.371 SMC

Secure monitor call

This instruction causes an exception to EL3.

`SMC` is available only for software executing at EL1 or higher. It is UNDEFINED at EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an `SMC` instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC syndrome value `0x17`, that is taken to EL3.

If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an `SMC` instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

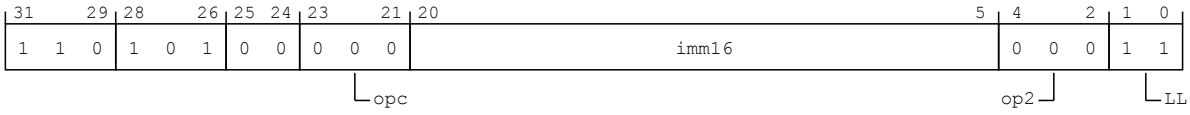

Encoding

```
SMC  #<imm>
```

Decode for this encoding

```
let imm : bits(16) = imm16;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;
AArch64_CheckForSMCUndefOrTrap(imm);
AArch64_CallSecureMonitor(imm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.372 SMIN (immediate)

Signed minimum (immediate)

This instruction determines the signed minimum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28						22	21			18	17						10	9				5	4					0
sf	0	0	1	0	0	0	1	1	1	0	0	1	0	imm8							Rn					Rd					
op		S										opc																			

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMIN  <Wd>, <Wn>, #<simm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMIN  <Xd>, <Xn>, #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let imm : integer = SInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<simm>

Is a signed immediate, in the range -128 to 127, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = Min(SInt(operand1), imm);
X{datasize}(d) = result[datasize-1:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.373 SMIN (register)

Signed minimum (register)

This instruction determines the signed minimum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	21	20	16	15	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 1 1 0 1 0		Rn		Rd	
		S					opcode					

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SMIN  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SMIN  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let result : integer = Min(SInt(operand1), SInt(operand2));
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.374 SMNEGL

Signed multiply-negate long

This instruction multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	1	1 1 1 1 1	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
SMNEGL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMSUBL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational Information

The description of SMSUBL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.375 SMSTART

Enable access to Streaming SVE mode and SME architectural state

This instruction enables access to Streaming SVE mode and SME architectural state.

SMSTART enters Streaming SVE mode, and enables the SME ZA storage.

SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.

SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

This is an alias of MSR (immediate). This means:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SME)

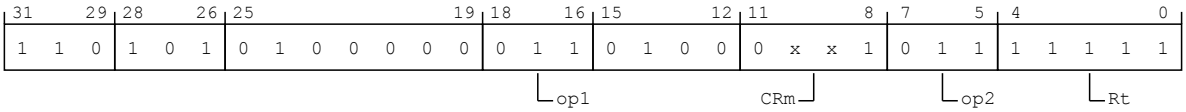

Encoding

```
SMSTART  {<option>}
```

is equivalent to

```
MSR  <pstatefield>, #1
```

and is always the preferred disassembly.

Assembler Symbols

<option>

Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>	Description
00	RESERVED	•
01	SM	Maps to <pstatefield> SVCRSM.
10	ZA	Maps to <pstatefield> SVCRZA.
11	[absent]	Maps to <pstatefield> SVCRSMZA.

Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.376 SMSTOP

Disable access to Streaming SVE mode and SME architectural state

This instruction disables access to Streaming SVE mode and SME architectural state.

SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.

SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.

SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

This is an alias of MSR (immediate). This means:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SME)

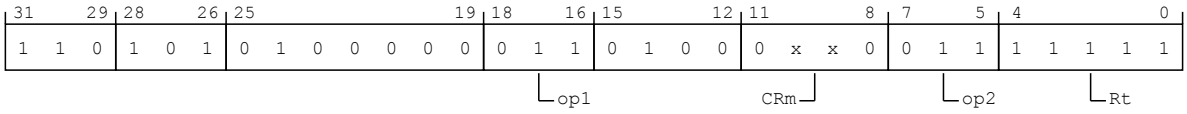

Encoding

```
SMSTOP  {<option>}
```

is equivalent to

```
MSR  <pstatefield>, #0
```

and is always the preferred disassembly.

Assembler Symbols

<option>

Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>	Description
00	RESERVED	•
01	SM	Maps to <pstatefield> SVCRSM.
10	ZA	Maps to <pstatefield> SVCRZA.
11	[absent]	Maps to <pstatefield> SVCRSMZA.

Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.377 SMSUBL

Signed multiply-subtract long

This instruction multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	1	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
SMSUBL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
SMNEGL	Ra == '11111'

Operation

```
let operand1 : bits(32) = X{}(n);
let operand2 : bits(32) = X{}(m);
let operand3 : bits(64) = X{}(a);

let result : integer = SInt(operand3) - (SInt(operand1) * SInt(operand2));

X{64}(d) = result[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.378 SMULH

Signed multiply high

This instruction multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	1 0	Rm	0	(1)(1)(1)(1)(1)	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
SMULH  <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
let operand1 : bits(64) = X{}(n);
let operand2 : bits(64) = X{}(m);

let result : integer = SInt(operand1) * SInt(operand2);

X{64}(d) = result[127:64];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.379 SMULL (32-bit)

Signed multiply long

This instruction multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	0	0 1	Rm	0	1 1 1 1 1	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
SMULL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMADDL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational Information

The description of SMADDL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.380 SSBB

Speculative store bypass barrier

This instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see Speculative Store Bypass Barrier (SSBB).

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31–29	28–26	25–12	11–8	7	6–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 1	0 0 0 0	1	0 0	1 1 1 1 1
			CRm		opc	Rt

Encoding

```
SSBB
```

is equivalent to

```
DSB  #0
```

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.381 ST2G

Store Allocation Tags

This instruction stores an Allocation Tag to two Tag Granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag Granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

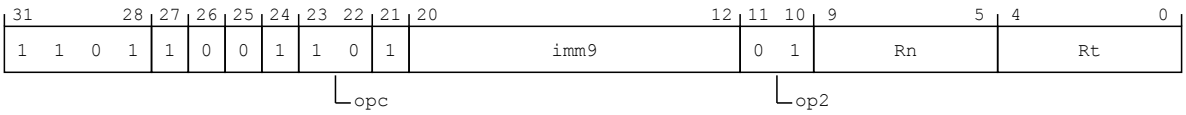

Encoding

```
ST2G  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_MTE)

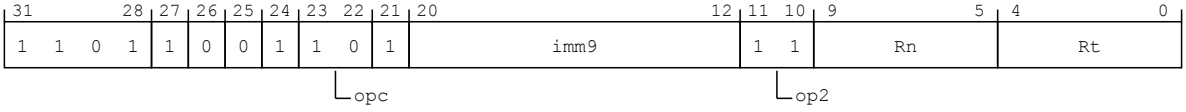

Encoding

```
ST2G  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	0	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
ST2G  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = FALSE;
let postindex : boolean = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);
var address2 : bits(64);
let data : bits(64) = if t == 31 then SP{64}() else X{64}(t);
let tag : bits(4) = AArch64_AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let stzgm : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

address2 = AddressIncrement(address, TAG_GRANULE, accdesc);

AArch64_MemTag(address , accdesc) = tag;
AArch64_MemTag(address2, accdesc) = tag;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;

    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.382 ST64B

Single-copy atomic 64-byte store without status result

This instruction stores eight 64-bit doublewords from consecutive registers to a memory location. The store starts at register `Xt`, with the data being formed as `Data[511:0] = X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt`. The data is stored atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `ST64B` also supports `LD64B`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64)

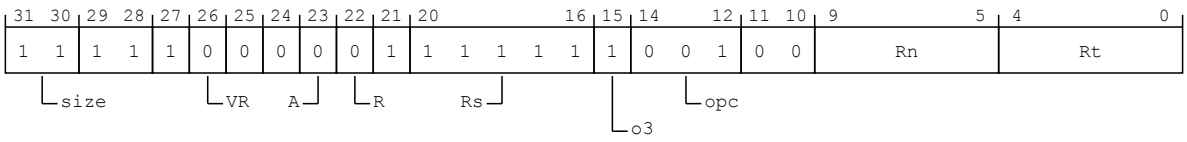

Encoding

```
ST64B  <Xt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64) then EndOfDecode(Decode_UNDEF); end;
if Rt[4:3] == '11' || Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let withstatus : boolean = FALSE;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let tagchecked : boolean = n != 31;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly ST64B.

Assembler Symbols

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckLDST64BEnabled();

var data : bits(512);
var address : bits(64);
var value : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLS64(MemOp_STORE, withstatus, tagchecked);

for i = 0 to 7 do
    value = X{64}(t+i);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if BigEndian(accdesc.acctype) then value = BigEndianReverse{64}(value); end;
    data[63+64*i : 64*i] = value;
end;

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

MemStore64B(address, data, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.383 ST64BV

Single-copy atomic 64-byte store with status result

This instruction stores eight 64-bit doublewords from consecutive registers to a memory location, and writes the status result of the store to a register. The store starts at register `Xt`, with the data being formed as `Data[511:0] = X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt`. The data is stored atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `ST64BV` also supports `ST64BV0`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64_V)

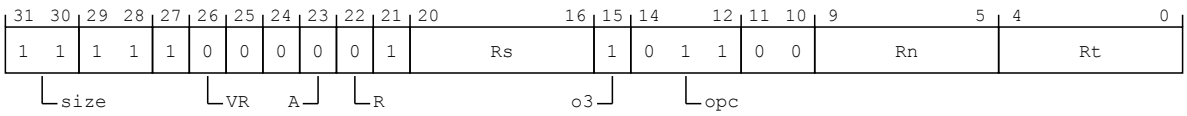

Encoding

```
ST64BV  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64_V) then EndOfDecode(Decode_UNDEF); end;
if Rt[4:3] == '11' || Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let withstatus : boolean = TRUE;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the “Rs” field.

The value returned is:

`0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= `0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckST64BVEnabled();

var data : bits(512);
var address : bits(64);
var value : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLS64(MemOp_STORE, withstatus, tagchecked);

for i = 0 to 7 do
    value = X{64}(t+i);
    if BigEndian(accdesc.acctype) then value = BigEndianReverse{64}(value); end;
    data[63+64*i : 64*i] = value;
end;

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let status : bits(64) = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X{64}(s) = status; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.384 ST64BV0

Single-copy atomic 64-byte EL0 store with status result

This instruction stores eight 64-bit doublewords from consecutive registers to a memory location, with the bottom 32 bits taken from ACCDATA_EL1, and writes the status result of the store to a register. The store starts at register `Xt`, with the data being formed as `Data[511:0] = X(t+7)::X(t+6)::X(t+5)::X(t+4)::X(t+3)::X(t+2)::X(t+1)::Xt[63:32]::ACCDATA_EL1()[31:0]`. The data is stored atomically and is required to be 64-byte aligned.

It is IMPLEMENTATION DEFINED which memory locations support this instruction. A memory location that supports `ST64BV0` also supports `ST64BV`.

For more information, including about the memory types accessible and how the accesses are performed, see Single-copy atomic 64-byte load/store.

Integer

(FEAT_LS64_ACCDATA)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 12	11	10	9 … 5	4 … 0
1	1	1	1	1	0	0	0	0	0	1	Rs	1	0 1 0	0	0	Rn	Rt
size					VR		A		R			o3	opc				

Encoding

```
ST64BV0  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64_ACCDATA) then EndOfDecode(Decode_UNDEF); end;
if Rt[4:3] == '11' || Rt[0] == '1' then EndOfDecode(Decode_UNDEF); end;
let withstatus : boolean = TRUE;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xs>

Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the “Rs” field.

The value returned is:

`0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= `0xFFFFFFFF`**_FFFFFFFF** If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckST64BV0Enabled();

var data : bits(512);
var address : bits(64);
var value : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLS64(MemOp_STORE, withstatus, tagchecked);

let Xt : bits(64) = X{}(t);
value[31:0] = ACCDATA_EL1()[31:0];
value[63:32] = Xt[63:32];
if BigEndian(accdesc.acctype) then value = BigEndianReverse{64}(value); end;
data[63:0] = value;

for i = 1 to 7 do
    value = X{64}(t+i);
    if BigEndian(accdesc.acctype) then value = BigEndianReverse{64}(value); end;
    data[63+64*i : 64*i] = value;
end;

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let status : bits(64) = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X{64}(s) = status; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.385 STADD, STADDL

Atomic add on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADD` does not have release semantics.
- `STADDL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

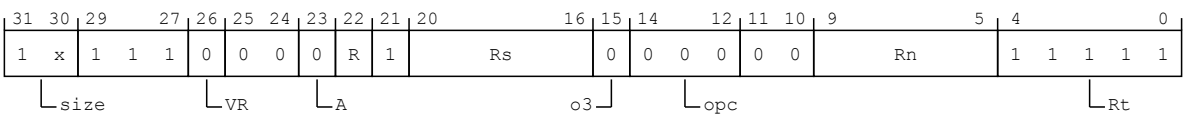

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STADD  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADD  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STADDL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STADD  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADD  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STADDL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADDL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational Information

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.386 STADDB, STADDLB

Atomic add on byte, without return

This instruction atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDB` does not have release semantics.
- `STADDLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STADDB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STADDLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.387 STADDH, STADDLH

Atomic add on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDH` does not have release semantics.
- `STADDLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

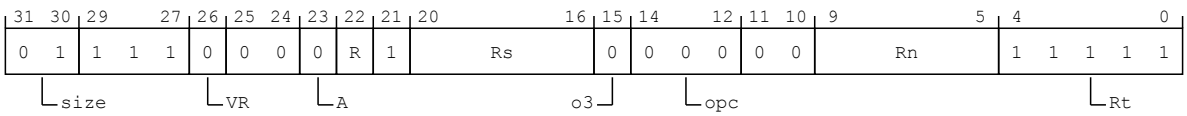

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STADDH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STADDLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.388 STCLR, STCLRL

Atomic bit clear on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLR` does not have release semantics.
- `STCLRL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

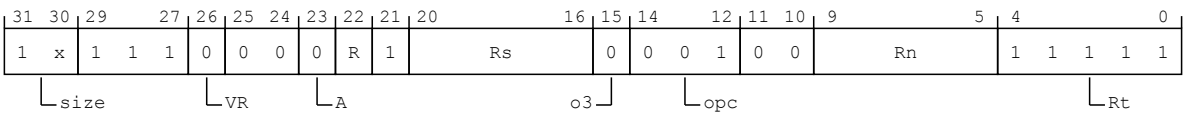

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STCLR  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLR  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STCLRL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STCLR  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLR  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STCLRL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.389 STCLRB, STCLRLB

Atomic bit clear on byte, without return

This instruction atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRB` does not have release semantics.
- `STCLRLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB. This means:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STCLRB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STCLRLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.390 STCLRH, STCLRLH

Atomic bit clear on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRH` does not have release semantics.
- `STCLRLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs	0	0	0	1	0	0	Rn	1	1	1	1	1
size					VR			A				o3	opc						Rt				

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STCLRH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STCLRLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.391 STEOR, STEORL

Atomic exclusive-OR on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- `STEOR` does not have release semantics.
- `STEORL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

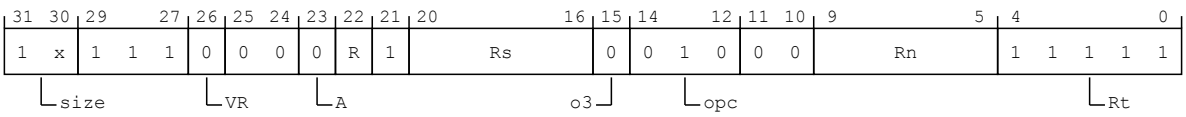

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STEOR  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEOR  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STEORL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STEOR  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEOR  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STEORL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEORL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational Information

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.392 STEORB, STEORLB

Atomic exclusive-OR on byte, without return

This instruction atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- `STEORB` does not have release semantics.
- `STEORLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STEORB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STEORLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.393 STEORH, STEORLH

Atomic exclusive-OR on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- `STEORH` does not have release semantics.
- `STEORLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STEORH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STEORLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.394 STG

Store Allocation Tag

This instruction stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag Granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	0	1	imm9		0	1	Rn		Rt	
						opc					op2					

Encoding

```
STG  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_MTE)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	0	0	1	imm9		1	1	Rn		Rt	
						opc					op2					

Encoding

```
STG  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	0	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
STG  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = FALSE;
let postindex : boolean = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let stzgm : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(64) = if t == 31 then SP{64}() else X{64}(t);
let tag : bits(4) = AArch64_AllocationTagFromAddress(data);
AArch64_MemTag(address, accdesc) = tag;

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.395 STGM

Store Allocation Tag multiple

This instruction writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

31	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1 1 0 1		1	0	0	1	1	0	1	0 0 0 0 0 0 0 0 0		0	0	Rn		Rt	
						opc			imm9		op2					

Encoding

```
STGM  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;

let data : bits(64) = X{}(t);
var address : bits(64);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let size : integer{} = 4 * (2 ^ (UInt(GMID_EL1().BS)));
address = AlignDownSize(address, size);
let count : integer = size >> LOG2_TAG_GRANULE;
var index : integer = UInt(address[LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE]);
let curraddress : bits(64) = address;
let stzgm : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

for i = 0 to count-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let tag : bits(4) = data[index*:4];
    AArch64_MemTag(address, accdesc) = tag;
    address = AddressIncrement(address, TAG_GRANULE, accdesc);
    index = index + 1;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.396 STGP

Store Allocation Tag and pair of registers

This instruction stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag Granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
0 1	1 0	1	0	0	0 1	0	simm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
STGP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(simm7), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_MTE)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
0 1	1 0	1	0	0	1 1	0	simm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
STGP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(simm7), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
0	1	1	0	1	0	0	1	0	0	simm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
STGP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(simm7), LOG2_TAG_GRANULE);
let writeback : boolean = FALSE;
let postindex : boolean = FALSE;
```

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index” and “Pre-index” variants: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the “simm7” field.

For the “Signed offset” variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “simm7” field.

Operation

```
var address : bits(64);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let stzgm : boolean = FALSE;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, ispair, t, t2);

if !postindex then
    address = AddressAdd(address, offset, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

if !IsAlignedSize(address, TAG_GRANULE) then
    let fault : FaultRecord = AlignmentFault(accdesc, address);
    AArch64_Abort(fault);
end;

let data : bits(128) = (if BigEndian(accdesc.acctype) then X{64}(t) ::X{64}(t2)
                                                      else X{64}(t2)::X{64}(t));
Mem{128}(address, accdesc) = data;

AArch64_MemTag(address, accdesc) = AArch64_AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;

    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.397 STILP

Store-release ordered pair of registers

This instruction calculates an address from a base register value and an optional offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release, with the additional requirement that:

- When using the pre-index addressing mode, the Memory effects associated with Xt2/Wt2 are Ordered-before the Memory effects associated with Xt1/Wt1.
- For all other addressing modes, the Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LRCPC3)

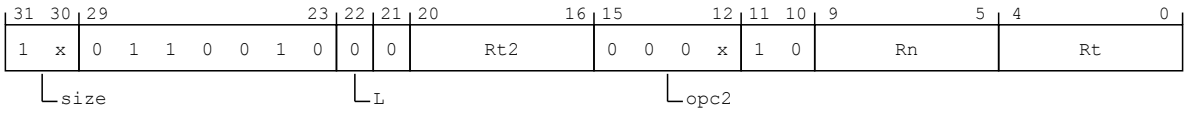

Encoding for the 32-bit pre-index variant

Applies when `(size == 10 && opc2 == 0000)`

```
STILP  <Wt1>, <Wt2>, [<Xn|SP>, #-8]!
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc2 == 0001)`

```
STILP  <Wt1>, <Wt2>, [<Xn|SP>]
```

Encoding for the 64-bit pre-index variant

Applies when `(size == 11 && opc2 == 0000)`

```
STILP  <Xt1>, <Xt2>, [<Xn|SP>, #-16]!
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc2 == 0001)`

```
STILP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF); end;
let ispair : boolean = TRUE;
let wback : boolean = opc2[0] == '0';
```

STILP has the same CONSTRAINED UNPREDICTABLE behavior as STP. For information about this CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP and STILP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let scale : integer{} = 2 + UInt(size[0]);
let datasize : integer{} = 8 << scale;
let offset : integer = if opc2[0] == '0' then -1 * (2 << scale) else 0;
let acqrel : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>    rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;
var accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked,
                                                     ispair, acqrel, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = ARBITRARY : bits(datasize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    data1 = X{datasize}(t);
end;
if rt_unknown && t2 == n then
    data2 = ARBITRARY : bits(datasize);
else
    data2 = X{datasize}(t2);
end;

var full_data : bits(2*datasize);
if BigEndian(accdesc.acctype) then
    full_data = data1::data2;
else
    full_data = data2::data1;
end;
accdesc.highestaddressfirst = offset < 0;
Mem{2*datasize}(address, accdesc) = full_data;

if wback then
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.398 STLLR

Store LORelease register

This instruction stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

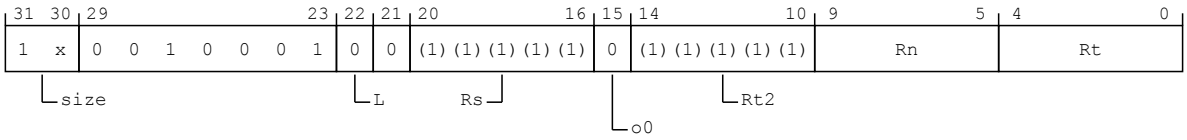

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLLR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLLR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let elsize : integer{} = 8 << UInt(size);
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
let dbytes : integer{} = elsize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescLOR(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

Mem{elsize}(address, accdesc) = X{elsize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.399 STLLRB

Store LORelease register byte

This instruction stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

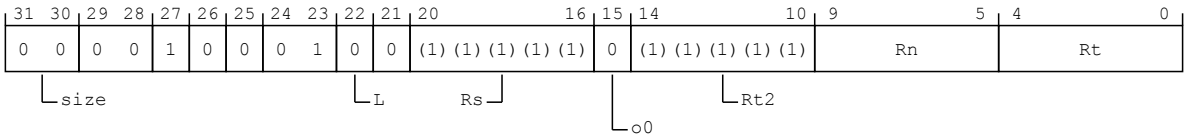

Encoding

```
STLLRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLOR(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

Mem{8}(address, accdesc) = X{8}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.400 STLLRH

Store LORelease register halfword

This instruction stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LOR)

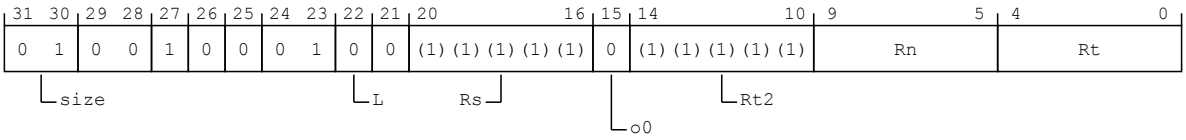

Encoding

```
STLLRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LOR) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescLOR(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

Mem{16}(address, accdesc) = X{16}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.401 STLR

Store-release register

This instruction stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 2 classes: No offset and Pre-index

No offset

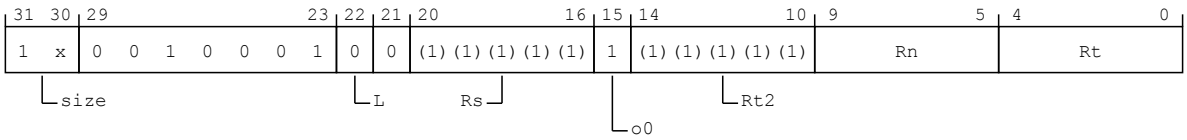

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLR  <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLR  <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let wback : boolean = FALSE;
let offset : integer = 0;
let rt_unknown : boolean = FALSE;

let elsize : integer{} = 8 << UInt(size);
let datasize : integer{} = elsize;
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Pre-index

(FEAT_LRCPC3)

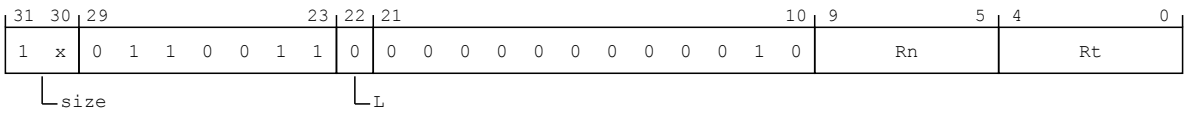

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLR  <Wt>, [<Xn|SP>, #-4]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLR  <Xt>, [<Xn|SP>, #-8]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC3) then EndOfDecode(Decode_UNDEF); end;
let wback : boolean = TRUE;

let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let offset : integer = -1 * (1 << UInt(size));
let acquire : boolean = FALSE;
let tagchecked : boolean = TRUE;

var rt_unknown : boolean = FALSE;

if n == t && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>    rt_unknown = FALSE;   // value stored is original value
        when Constraint_UNKNOWN => rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
var data : bits(datasize);
if rt_unknown then
    data = ARBITRARY : bits(datasize);
else
    data = X{datasize}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
Mem{datasize}(address, accdesc) = data;
if wback then
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.402 STLRB

Store-release register byte

This instruction stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size						L		Rs	o0	Rt2		

Encoding

```
STLRB  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

Mem{8}(address, accdesc) = X{8}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.403 STLRH

Store-release register halfword

This instruction stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	1	0	0	1	0	0	0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

Encoding

```
STLRH  <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

Mem{16}(address, accdesc) = X{16}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.404 STLTXR

Store-release unprivileged exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `STLTXR` is equivalent to `STXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

31	30	29						23	22	21	20			16	15	14				10	9			5	4			0
1	sz	0	0	1	0	0	1	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt			
									L						o0					Rt2								

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STLTXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STLTXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

STLTXR has the same CONSTRAINED UNPREDICTABLE behavior as STLXR. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64_ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(elsize);
else
    data = X{elsize}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{elsize}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.405 STLUR

Store-release register (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29					24	23	22	21	20				12	11	10	9			5	4			0
1	x	0	1	1	0	0	1	0	0	0	imm9					0	0	Rn				Rt			
size								opc																	

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);

let datasize : integer{} = 8 << scale;
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.406 STLURB

Store-release register byte (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
STLURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);

let datasize : integer{} = 8;
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.407 STLURH

Store-release register halfword (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset

(FEAT_LRCPC2)

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Encoding

```
STLURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF); end;
let offset : bits(64) = SignExtend{}(imm9);
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);

let datasize : integer{} = 16;
let acquire : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.408 STLXP

Store-release exclusive pair of registers

This instruction stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

31	30	29						23	22	21	20			16	15	14			10	9			5	4			0
1	sz	0	0	1	0	0	0	0	0	1	Rs				1	Rt2				Rn				Rt			
									L						o0												

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STLXP  <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STLXP  <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let datasize : integer{} = elsize * 2;
let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t || (s == t2) then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXP.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64_ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(datasize);

let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

if rt_unknown then
    data = ARBITRARY : bits(datasize);
else
    let el1 : bits(datasize DIV 2) = X{}(t);
    let el2 : bits(datasize DIV 2) = X{}(t2);
    data = if BigEndian(accdesc.acctype) then el1::el2 else el2::el1;
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{datasize}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.409 STLXR

Store-release exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

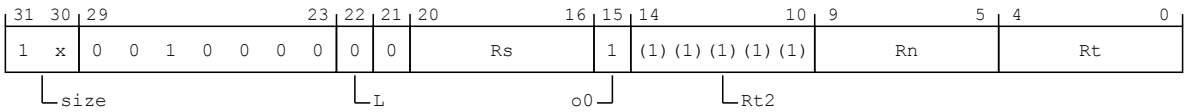

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STLXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STLXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 8 << UInt(size);
let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64_ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(elsize);
else
    data = X{elsize}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{elsize}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.410 STLXRB

Store-release exclusive register byte

This instruction stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 0	0	0	Rs	1	(1) (1) (1) (1) (1)	Rn	Rt
size						L			o0	Rt2		

Encoding

```
STLXRB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(8);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(8);
else
    data = X{8}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{8}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.411 STLXRH

Store-release exclusive register halfword

This instruction stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.

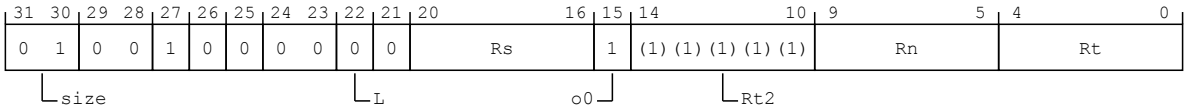

Encoding

```
STLXRH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = TRUE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRH.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64_ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(16);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(16);
else
    data = X{16}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{16}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.412 STNP

Store pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

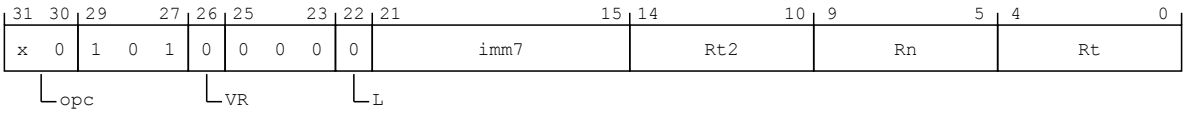

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STNP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
let data : bits(2*datasize)= (if BigEndian(accdesc.acctype)
                                  then X{datasize}(t) ::X{datasize}(t2)
                                  else X{datasize}(t2)::X{datasize}(t));
Mem{2*datasize}(address, accdesc) = data;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.413 STP

Store pair of registers

This instruction calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

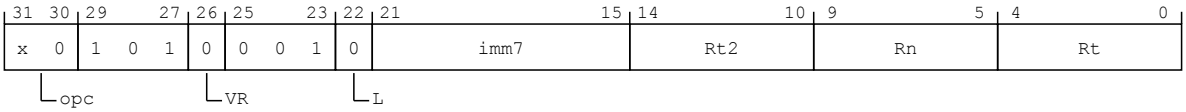

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

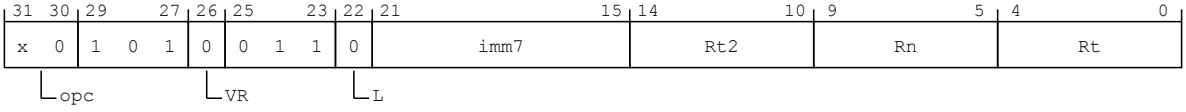

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

31	30	29			27	26	25			23	22	21	15	14	10	9	5	4	0
x	0	1	0	1		0	0	1	0		0	imm7		Rt2		Rn		Rt	
opc						VR					L								

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 10)`

```
STP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

Assembler Symbols

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>       rt_unknown = FALSE;   // Value stored is pre-writeback
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Value stored is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

if rt_unknown && t == n then
    data1 = ARBITRARY : bits(datasize);
else
    data1 = X{datasize}(t);
end;
if rt_unknown && t2 == n then
    data2 = ARBITRARY : bits(datasize);
else
    data2 = X{datasize}(t2);
end;

let data :  bits(2*datasize) = (if BigEndian(accdesc.acctype) then data1::data2
                                                               else data2::data1);
Mem{2 * datasize}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.414 STR (immediate)

Store register (immediate)

This instruction stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	0	imm9		0	1	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	0	imm9		1	1	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
1 x	1 1 1	0	0 1	0 0	imm12	Rn	Rt
size		VR		opc			

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
let scale : integer{} = UInt(size);
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), scale);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<pimm>

For the “32-bit” variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

For the “64-bit” variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var rt_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN => rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

var data : bits(datasize);
if rt_unknown then
    data = ARBITRARY : bits(datasize);
else
    data = X{datasize}(t);
end;

Mem{datasize}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.415 STR (register)

Store register (register)

This instruction calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size		VR		opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STR  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STR  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;         // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let scale : integer{} = UInt(size);
let shift : integer{} = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.416 STRB (immediate)

Store register byte (immediate)

This instruction stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
STRB  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
STRB  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	0	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size					VR			opc				

Encoding

```
STRB  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRB (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var rt_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN => rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var data : bits(8);
if rt_unknown then
    data = ARBITRARY : bits(8);
else
    data = X{8}(t);
end;

Mem{8}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.417 STRB (register)

Store register byte (register)

This instruction calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

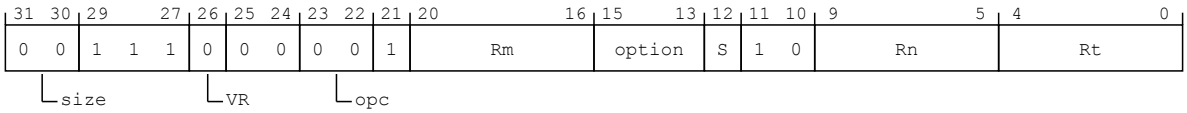

Encoding for the Extended register variant

Applies when `(option != 011)`

```
STRB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the Shifted register variant

Applies when `(option == 011)`

```
STRB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
110	SXTW
111	SXTX

<amount>

Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{8}(address, accdesc) = X{8}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.418 STRH (immediate)

Store register halfword (immediate)

This instruction stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size					VR			opc							

Encoding

```
STRH  <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size					VR			opc							

Encoding

```
STRH  <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	1	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size					VR			opc				

Encoding

```
STRH  <Wt>, [<Xn|SP>{, #<pimm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRH (immediate).

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<pimm>

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;

var c : Constraint;
var rt_unknown : boolean = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN => rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF =>   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>     EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var data : bits(16);
if rt_unknown then
    data = ARBITRARY : bits(16);
else
    data = X{16}(t);
end;

Mem{16}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.419 STRH (register)

Store register halfword (register)

This instruction calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

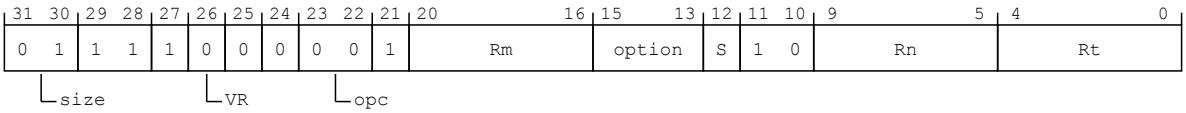

Encoding

```
STRH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;          // sub-word index
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{16}(address, accdesc) = X{16}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.420 STSET, STSETL

Atomic bit set on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSET` does not have release semantics.
- `STSETL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

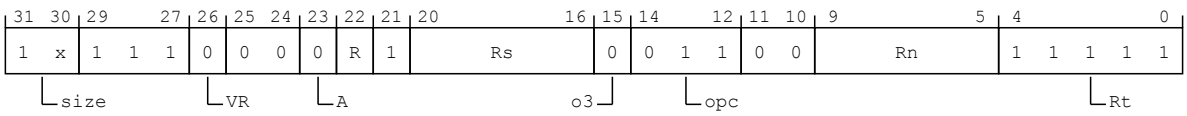

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STSET  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSET  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STSETL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STSET  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSET  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STSETL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSETL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.421 STSETB, STSETLB

Atomic bit set on byte, without return

This instruction atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETB` does not have release semantics.
- `STSETLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSETB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSETLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.422 STSETH, STSETLH

Atomic bit set on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETH` does not have release semantics.
- `STSETLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSETH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSETLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.423 STSHH

Store shared hint

This instruction signals to the memory system that if the next instruction in program order generates an explicit write effect, then it is to a location that one or more other threads of execution will observe, and there is a performance benefit to ensuring that the updated value from the write to that location propagates to those other observers with minimal latency.

The thread of execution on the other observers might be polling the location using load or Load-Exclusive instructions, or may have executed a `PRFM IR` instruction targeting the location.

System

(FEAT_PCDPHINT)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 1 1 0 (CRm)	0 0 x (op2)	1 1 1 1 1

Encoding

```
STSHH  <policy>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PCDPHINT) then EndOfDecode(Decode_NOP); end;
let stream : boolean = op2[0] == '1';
```

Assembler Symbols

<policy>

`<policy>` is one of:

KEEP Signals to the memory system that there may be a performance benefit to retaining the updated location in the local cache of the updating PE.

STRM Signals to the memory system that there may be a performance benefit to ensuring the updated location is not retained in the local cache of the updating PE.

op2<0>	<policy>
0	KEEP
1	STRM

Operation

```
Hint_StoreShared(stream);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.424 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` does not have release semantics.
- `STSMAXL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

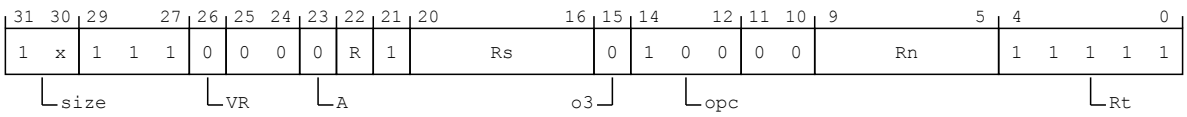

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STSMAX  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STSMAXL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STSMAX  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STSMAXL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.425 STSMAXB, STSMAXLB

Atomic signed maximum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXB` does not have release semantics.
- `STSMAXLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMAXB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMAXLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.426 STSMAXH, STSMAXLH

Atomic signed maximum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` does not have release semantics.
- `STSMAXLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMAXH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMAXLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.427 STSMIN, STSMINL

Atomic signed minimum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` does not have release semantics.
- `STSMINL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	R	1	Rs	0	1	0	1	0	0	Rn	1	1	1	1	1
size				VR			A					o3	opc						Rt				

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STSMIN  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STSMINL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STSMIN  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STSMINL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.428 STSMINB, STSMINLB

Atomic signed minimum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` does not have release semantics.
- `STSMINLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMINB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMINLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.429 STSMINH, STSMINLH

Atomic signed minimum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINH` does not have release semantics.
- `STSMINLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STSMINH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STSMINLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.430 STTADD, STTADDL

Atomic add unprivileged, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STTADD` does not have release semantics.
- `STTADDL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDTADD, LDTADDA, LDTADDAL, LDTADDL. This means:

- The encodings in this description are named to match the encodings of LDTADD, LDTADDA, LDTADDAL, LDTADDL.
- The description of LDTADD, LDTADDA, LDTADDAL, LDTADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9				5	4				0
0	sz	0	1	1	0	0	1	0	R	1	Rs				0	0	0	0	0	0	1	Rn					1	1	1	1	1
								A							o3		opc														Rt

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && R == 0)`

```
STTADD  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTADD  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(sz == 0 && R == 1)`

```
STTADDL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTADDL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && R == 0)`

```
STTADD  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTADD  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(sz == 1 && R == 1)`

```
STTADDL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTADDL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDTADD, LDTADDA, LDTADDAL, LDTADDL gives the operational pseudocode for this instruction.

Operational Information

The description of LDTADD, LDTADDA, LDTADDAL, LDTADDL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.431 STTCLR, STTCLRL

Atomic bit clear unprivileged, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STTCLR` does not have release semantics.
- `STTCLRL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL. This means:

- The encodings in this description are named to match the encodings of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL.
- The description of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20				16	15	14			12	11	10	9				5	4				0
0	sz	0	1	1	0	0	1	0	R	1	Rs					0	0	0	0	1	0	1	Rn					1	1	1	1	1
								A								o3				opc												Rt

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && R == 0)`

```
STTCLR  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTCLR  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(sz == 0 && R == 1)`

```
STTCLRL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTCLRL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && R == 0)`

```
STTCLR  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTCLR  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(sz == 1 && R == 1)`

```
STTCLRL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTCLRL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL gives the operational pseudocode for this instruction.

Operational Information

The description of LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.432 STTNP

Store unprivileged pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, and stores two 64-bit doublewords to the calculated address, from two registers. For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Signed offset

(FEAT_LSUI)

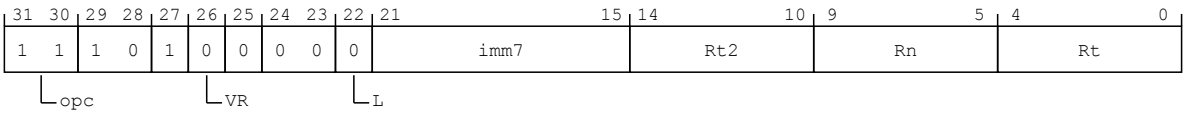

Encoding

```
STTNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;

let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let datasize : integer{} = 64;
let offset : bits(64) = LSL(SignExtend{64}(imm7), 3);
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);
let data : bits(2*datasize)= (if BigEndian(accdesc.acctype)
                                  then X{datasize}(t) ::X{datasize}(t2)
                                  else X{datasize}(t2)::X{datasize}(t));
Mem{2*datasize}(address, accdesc) = data;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.433 STTP

Store unprivileged pair of registers

This instruction calculates an address from a base register value and an immediate offset, and stores two 64-bit doublewords to the calculated address, from two registers.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	0	0	0 1	0	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
STTP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	0	0	1 1	0	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
STTP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_LSUI)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
1	1	1	0	1	0	0	1	0	0	imm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
STTP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

STTP has the same CONSTRAINED UNPREDICTABLE behavior as STP. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Load/store register pair (post-indexed)” and “Load/store register pair (pre-indexed)” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Load/store register pair (offset)” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + UInt(opc[1]);
let datasize : integer{} = 64;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE =>       rt_unknown = FALSE;   // Value stored is pre-writeback
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Value stored is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
var address : bits(64);
var data1 : bits(datasize);
var data2 : bits(datasize);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, ispair, t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

if rt_unknown && t == n then
    data1 = ARBITRARY : bits(datasize);
else
    data1 = X{datasize}(t);
end;
if rt_unknown && t2 == n then
    data2 = ARBITRARY : bits(datasize);
else
    data2 = X{datasize}(t2);
end;

let data :  bits(2*datasize) = (if BigEndian(accdesc.acctype) then data1::data2
                                                                 else data2::data1);
Mem{2 * datasize}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.434 STTR

Store register (unprivileged)

This instruction stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	0	0	imm9		1	0	Rn		Rt	
size				VR			opc										

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STTR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STTR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.435 STTRB

Store register byte (unprivileged)

This instruction stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1	1	0	0	0	0 0	0	imm9	1 0	Rn	Rt
size			VR			opc					

Encoding

```
STTRB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.436 STTRH

Store register halfword (unprivileged)

This instruction stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size					VR			opc							

Encoding

```
STTRH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 16;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.437 STTSET, STTSETL

Atomic bit set unprivileged, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STTSET` does not have release semantics.
- `STTSETL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDTSET, LDTSETA, LDTSETAL, LDTSETL. This means:

- The encodings in this description are named to match the encodings of LDTSET, LDTSETA, LDTSETAL, LDTSETL.
- The description of LDTSET, LDTSETA, LDTSETAL, LDTSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9				5	4				0
0	sz	0	1	1	0	0	1	0	R	1	Rs				0	0	1	1		0	1	Rn					1	1	1	1	1
								A							o3		opc													Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(sz == 0 && R == 0)`

```
STTSET  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTSET  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(sz == 0 && R == 1)`

```
STTSETL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDTSETL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(sz == 1 && R == 0)`

```
STTSET  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTSET  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(sz == 1 && R == 1)`

```
STTSETL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDTSETL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDTSET, LDTSETA, LDTSETAL, LDTSETL gives the operational pseudocode for this instruction.

Operational Information

The description of LDTSET, LDTSETA, LDTSETAL, LDTSETL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.438 STTXR

Store unprivileged exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Note

For the purposes of the Exclusives monitors, and the forward progress guarantees for Load-Exclusive and Store-Exclusive loops, `STTXR` is equivalent to `STXR`.

For information about addressing modes, see Load/Store addressing modes.

No offset

(FEAT_LSUI)

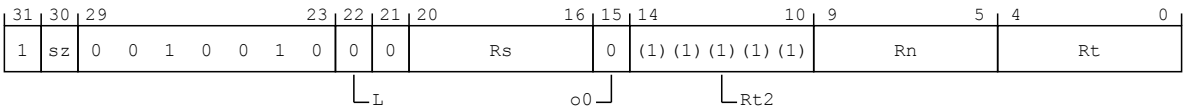

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STTXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STTXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

STTXR has the same CONSTRAINED UNPREDICTABLE behavior as STXR. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64_ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(elsize);
else
    data = X{elsize}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{elsize}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.439 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAX` does not have release semantics.
- `STUMAXL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	1	1 0	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STUMAX  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STUMAXL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STUMAX  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STUMAXL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.440 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXB` does not have release semantics.
- `STUMAXLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMAXB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMAXLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.441 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXH` does not have release semantics.
- `STUMAXLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMAXH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMAXLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.442 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` does not have release semantics.
- `STUMINL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1 1	0	0	0	0	R	1	Rs		0	1	1 1	0	0	Rn		1 1 1 1 1	
size				VR			A					o3	opc						Rt	

Encoding for the 32-bit no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STUMIN  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant

Applies when `(size == 10 && R == 1)`

```
STUMINL  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STUMIN  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit release variant

Applies when `(size == 11 && R == 1)`

```
STUMINL  <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL  <Xs>, <Xt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.443 STUMINB, STUMINLB

Atomic unsigned minimum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` does not have release semantics.
- `STUMINLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size		VR		A				o3	opc			Rt

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMINB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMINLB  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLB  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.444 STUMINH, STUMINLH

Atomic unsigned minimum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINH` does not have release semantics.
- `STUMINLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)

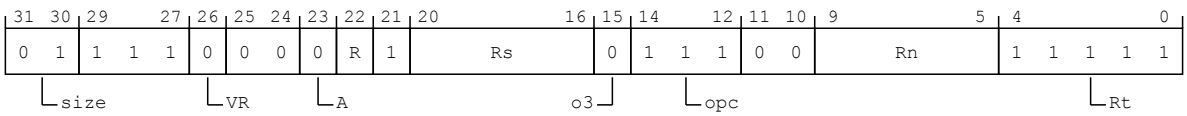

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STUMINH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant

Applies when `(R == 1)`

```
STUMINLH  <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLH  <Ws>, <Wt>, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.445 STUR

Store register (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size					VR			opc							

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STUR  <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STUR  <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
let scale : integer{} = UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.446 STURB

Store register byte (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size					VR			opc							

Encoding

```
STURB  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 8;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.447 STURH

Store register halfword (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about addressing modes, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size					VR			opc							

Encoding

```
STURH  <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
let n : integer = UInt(Rn);
let t : integer = UInt(Rt);
let datasize : integer{} = 16;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = X{datasize}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.448 STXP

Store exclusive pair of registers

This instruction stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about addressing modes, see Load/Store addressing modes.

31	30	29	23	22	21	20	16	15	14	10	9	5	4	0
1	sz	0 0 1 0 0 0 0		0	1	Rs		0	Rt2		Rn		Rt	
				L				o0						

Encoding for the 32-bit variant

Applies when `(sz == 0)`

```
STXP  <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(sz == 1)`

```
STXP  <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 32 << UInt(sz);
let datasize : integer{} = elsize * 2;
let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t || (s == t2) then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt1>

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2>

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64_ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(datasize);

let dbytes : integer{} = datasize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, ispair, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(datasize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    let el1 : bits(datasize DIV 2) = X{}(t);
    let el2 : bits(datasize DIV 2) = X{}(t2);
    data = if BigEndian(accdesc.acctype) then el1::el2 else el2::el1;
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{datasize}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.449 STXR

Store exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores.

For information about addressing modes, see Load/Store addressing modes.

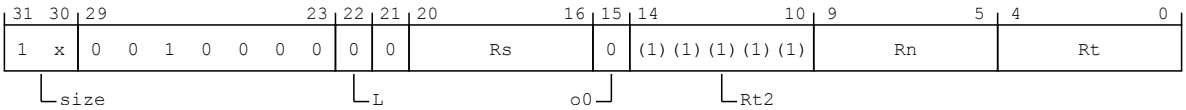

Encoding for the 32-bit variant

Applies when `(size == 10)`

```
STXR  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant

Applies when `(size == 11)`

```
STXR  <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let elsize : integer{} = 8 << UInt(size);
let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64_ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(elsize);

let dbytes : integer{} = elsize DIV 8;
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(elsize);
else
    data = X{elsize}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{elsize}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.450 STXRB

Store exclusive register byte

This instruction stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about addressing modes, see Load/Store addressing modes.

31 30	29 28	27	26	25	24 23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0	1	0	0	0 0	0	0	Rs	0	(1) (1) (1) (1) (1)	Rn	Rt
size						L			o0	Rt2		

Encoding

```
STXRB  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXRB.

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(8);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(8);
else
    data = X{8}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{8}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.451 STXRH

Store exclusive register halfword

This instruction stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about addressing modes, see Load/Store addressing modes.

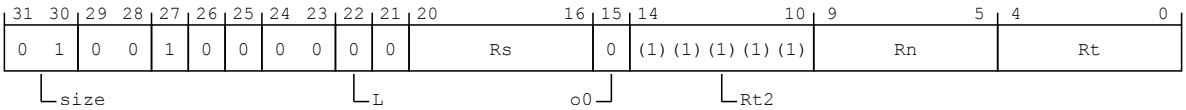

Encoding

```
STXRH  <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acqrel : boolean = FALSE;
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;
var rn_unknown : boolean = FALSE;
if s == t then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
if s == n && n != 31 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64_ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64_ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
var address : bits(64);
var data : bits(16);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                     tagchecked, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
elsif rn_unknown then
    address = ARBITRARY : bits(64);
else
    address = X{64}(n);
end;

if rt_unknown then
    data = ARBITRARY : bits(16);
else
    data = X{16}(t);
end;
var status : bit = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64_ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem is not called.

// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64_ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64_ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem{16}(address, accdesc) = data;
    status = ExclusiveMonitorsStatus();
end;
X{32}(s) = ZeroExtend{32}(status);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.452 STZ2G

Store Allocation Tags, zeroing

This instruction stores an Allocation Tag to two Tag Granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag Granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

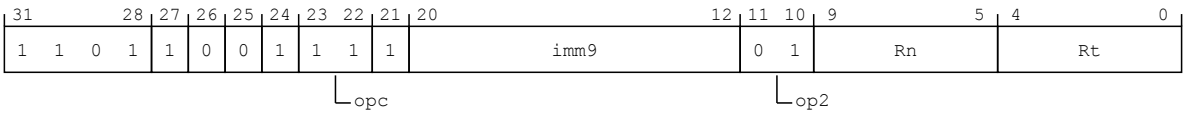

Encoding

```
STZ2G  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_MTE)

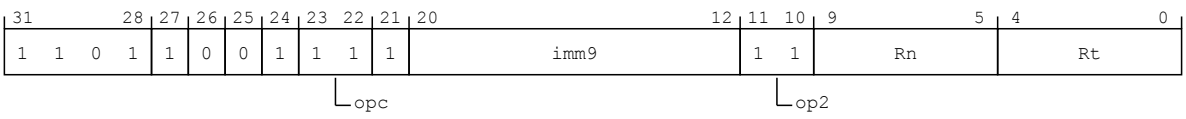

Encoding

```
STZ2G  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	1	1	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
STZ2G  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = FALSE;
let postindex : boolean = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);
var address2 : bits(64);
let data : bits(64) = if t == 31 then SP{64}() else X{64}(t);
let tag : bits(4) = AArch64_AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let stzgm : boolean = FALSE;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, ispair, t, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

address2 = AddressIncrement(address, TAG_GRANULE, accdesc);

if !IsAlignedSize(address, TAG_GRANULE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let fault : FaultRecord = AlignmentFault(accdesc, address);
    AArch64_Abort(fault);
end;

Mem{16*TAG_GRANULE}(address, accdesc) = Zeros{TAG_GRANULE * 16};

AArch64_MemTag(address , accdesc) = tag;
AArch64_MemTag(address2, accdesc) = tag;

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;

    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.453 STZG

Store Allocation Tag, zeroing

This instruction stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag Granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_MTE)

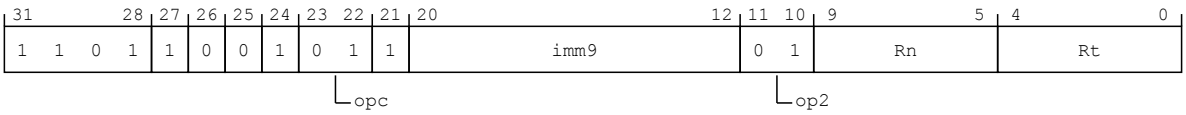

Encoding

```
STZG  <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_MTE)

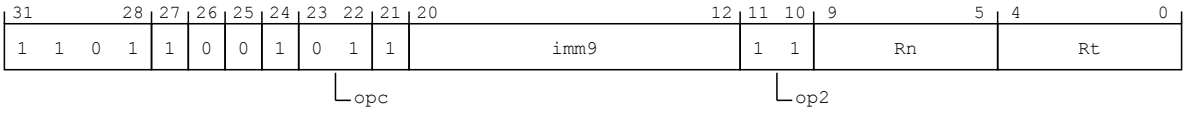

Encoding

```
STZG  <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = TRUE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Signed offset

(FEAT_MTE)

31			28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	1	0	1	1	0	0	1	0	1	1	imm9	1	0	Rn	Rt
								opc				op2			

Encoding

```
STZG  <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let offset : bits(64) = LSL(SignExtend{64}(imm9), LOG2_TAG_GRANULE);
let writeback : boolean = FALSE;
let postindex : boolean = FALSE;
```

Assembler Symbols

<Xt|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the “imm9” field.

Operation

```
var address : bits(64);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let stzgm : boolean = FALSE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

if !IsAlignedSize(address, TAG_GRANULE) then
    let fault : FaultRecord = AlignmentFault(accdesc, address);
    AArch64_Abort(fault);
end;

Mem{8*TAG_GRANULE}(address, accdesc) = Zeros{TAG_GRANULE * 8};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let data : bits(64) = if t == 31 then SP{64}() else X{64}(t);
let tag : bits(4) = AArch64_AllocationTagFromAddress(data);
AArch64_MemTag(address, accdesc) = tag;

if writeback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;

    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.454 STZGM

Store Allocation Tag and zero multiple

This instruction writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0. This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

31			28	27	26	25	24	23	22	21	20									12	11	10	9	5	4	0
1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn		Rt	
								opc			imm9										op2					

Encoding

```
STZGM  <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
if PSTATE.EL == EL0 then Undefined(); end;

let data : bits(64) = X{}(t);
let tag : bits(4) = data[3:0];
var address : bits(64);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let size : integer{} = 4 * (2 ^ (UInt(DCZID_EL0().BS)));
address = AlignDownSize(address, size);
let count : integer = size >> LOG2_TAG_GRANULE;
let stzgm : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescLDGSTG(MemOp_STORE, stzgm, t);

for i = 0 to count-1 do
    AArch64_MemTag(address, accdesc) = tag;
    Mem{8*TAG_GRANULE}(address, accdesc) = Zeros{8*TAG_GRANULE};
    address = AddressIncrement(address, TAG_GRANULE, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.455 SUB (extended register)

Subtract extended and scaled register

This instruction subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword.

31	30	29	28				24	23	22	21	20		16	15		13	12		10	9		5	4		0
sf	1	0	0	1	0	1	1	0	0	1	Rm			option			imm3			Rn			Rd		
op		S						opt																	

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SUB  <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SUB  <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let shift : integer{} = UInt(imm3) as integer{0..4};
let datasize : integer{} = 32 << UInt(sf);
let extend_type : ExtendType = DecodeRegExtend(option);
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTX when “option” is ‘011’.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

option	<R>
00x	W

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
010	W
x11	X
10x	W
110	W

<m>

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = NOT(ExtendReg{datasize}(m, extend_type, shift));
var result : bits(datasize);

(result, -) = AddWithCarry{datasize}(operand1, operand2, '1');

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.456 SUB (immediate)

Subtract immediate value

This instruction subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

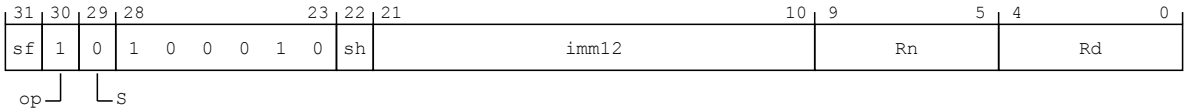

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SUB  <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SUB  <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);

let imm : bits(24) = if sh == '0' then Zeros{12}::imm12 else imm12::Zeros{12};
```

Assembler Symbols

<Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = ZeroExtend{}(imm);
var result : bits(datasize);
(result, -) = AddWithCarry{datasize}(operand1, NOT(operand2), '1');

if d == 31 then
    SP{64}() = ZeroExtend{64}(result);
else
    X{datasize}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.457 SUB (shifted register)

Subtract optionally-shifted register

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register).

31	30	29	28 … 24	23 22	21	20 … 16	15 … 10	9 … 5	4 … 0
sf	1	0	0 1 0 1 1	shift	0	Rm	imm6	Rn	Rd
	op	S							

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SUB  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SUB  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
NEG (shifted register)	Rn == '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = NOT(ShiftReg{datasize}(m, shift_type, shift_amount));
var result : bits(datasize);

(result, -) = AddWithCarry{datasize}(operand1, operand2, '1');

X{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.458 SUBG

Subtract with tag

This instruction subtracts an immediate value scaled by the Tag Granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer

(FEAT_MTE)

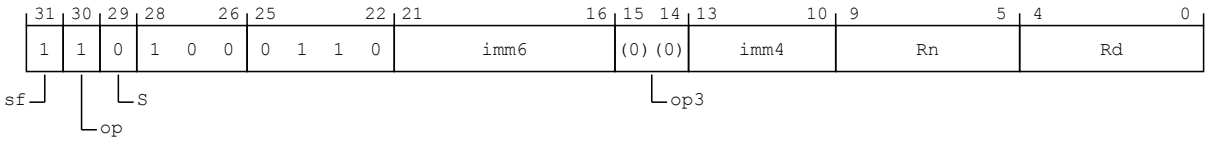

Encoding

```
SUBG  <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let tag_offset : bits(4) = imm4;
let offset : bits(64) = LSL(ZeroExtend{64}(imm6), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<uimm6>

Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the “imm6” field.

<uimm4>

Is an unsigned immediate, in the range 0 to 15, encoded in the “imm4” field.

Operation

```
let operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let start_tag : bits(4) = AArch64_AllocationTagFromAddress(operand1);
let exclude : bits(16) = GCR_EL1().Exclude;
let rtag : bits(4) = AArch64_ChooseNonExcludedTagOrZero(start_tag, tag_offset, exclude);
var result : bits(64);

(result, -) = AddWithCarry{64}(operand1, NOT(offset), '1');

result = AArch64_AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP{64}() = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    X{64}(d) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.459 SUBP

Subtract pointer

This instruction subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64 bits, and writes the result to the destination register.

Integer

(FEAT_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Rm					0	0	0	0	0	0	Rn					Rd				

sf, S, opcode

Encoding

```
SUBP  <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Operation

```
var operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
var operand2 : bits(64) = if m == 31 then SP{64}() else X{64}(m);
operand1 = SignExtend{64}(operand1[55:0]);
operand2 = NOT(SignExtend{64}(operand2[55:0]));
var result : bits(64);

(result, -) = AddWithCarry{64}(operand1, operand2, '1');

X{64}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.460 SUBPS

Subtract pointer, setting flags

This instruction subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64 bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias CMPP.

Integer

(FEAT_MTE)

31	30	29	28	27 25	24 21	20 16	15 10	9 5	4 0
1	0	1	1	1 0 1	0 1 1 0	Rm	0 0 0 0 0 0	Rn	Rd
sf		S					opcode		

Encoding

```
SUBPS  <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Xm|SP>

Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMPP	S == '1' && Rd == '11111'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
var operand2 : bits(64) = if m == 31 then SP{64}() else X{64}(m);
operand1 = SignExtend{64}(operand1[55:0]);
operand2 = NOT(SignExtend{64}(operand2[55:0]));
var result : bits(64);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{64}(operand1, operand2, '1');

X{64}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.461 SUBPT

Subtract checked pointer

This instruction subtracts an optionally-shifted register value from a base address register value, and writes the result to the destination register. The optionally-shifted register value is treated as the offset.

If the operation would have generated a result where the most significant 8 bits of the result register differ from the most significant 8 bits of the base register, then the result is modified such that it is likely to be non-canonical when used as an address.

Integer

(FEAT_CPA)

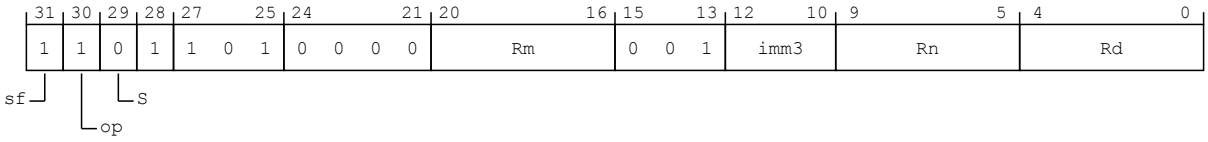

Encoding

```
SUBPT  <Xd|SP>, <Xn|SP>, <Xm>{, LSL #<amount>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CPA) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let shift : integer = UInt(imm3);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the general-purpose destination register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first general-purpose source register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the left shift amount, in the range 0 to 7, defaulting to 0, encoded in the “imm3” field.

Operation

```
var result : bits(64);
let base : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let offset : bits(64) = LSL(X{64}(m), shift);

result = base - offset;
result = PointerAddCheck(result, base);

if d == 31 then
    SP{64}() = result;
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(d) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.462 SUBS (extended register)

Subtract extended and scaled register, setting flags

This instruction subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the `<Rm>` register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register).

31	30	29	28				24	23	22	21	20		16	15		13	12		10	9		5	4		0
sf	1	1	0	1	0	1	1	0	0	1	Rm			option			imm3			Rn			Rd		
op		S						opt																	

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SUBS  <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SUBS  <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let shift : integer{} = UInt(imm3) as integer{0..4};
let datasize : integer{} = 32 << UInt(sf);
let extend_type : ExtendType = DecodeRegExtend(option);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<extend>

For the “32-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when “option” is ‘010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTX when “option” is ‘011’.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<R>

Is a width specifier, encoded in “option”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m>

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMP (extended register)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = NOT(ExtendReg{datasize}(m, extend_type, shift));
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, '1');

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.463 SUBS (immediate)

Subtract immediate value, setting flags

This instruction subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate).

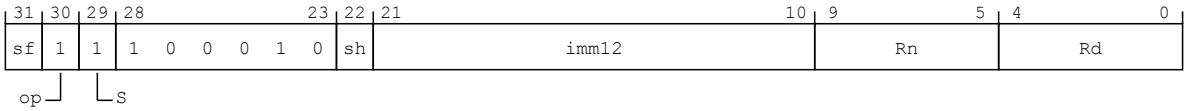

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SUBS  <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SUBS  <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);

let imm : bits(24) = if sh == '0' then Zeros{12}::imm12 else imm12::Zeros{12};
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
CMP (immediate)	Rd == '11111'

Operation

```
let operand1 : bits(datasize) = if n == 31 then SP{datasize}() else X{datasize}(n);
let operand2 : bits(datasize) = ZeroExtend{}(imm);
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, NOT(operand2), '1');

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.464 SUBS (shifted register)

Subtract optionally-shifted register, setting flags

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register) and NEGS.

31	30	29	28				24	23	22	21	20			16	15				10	9				5	4				0
sf	1	1	0	1	0	1	1	shift		0	Rm				imm6					Rn					Rd				

op S

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
SUBS  <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
SUBS  <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && imm6[5] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
let shift_type : ShiftType = DecodeShift(shift);
let shift_amount : integer = UInt(imm6);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMP (shifted register)	Rd == '11111'
NEGS	Rn == '11111' && Rd != '11111'

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = NOT(ShiftReg{datasize}(m, shift_type, shift_amount));
var result : bits(datasize);
var nzcv : bits(4);

(result, nzcv) = AddWithCarry{datasize}(operand1, operand2, '1');

X{datasize}(d) = result;
PSTATE.[N,Z,C,V] = nzcv;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.465 SVC

Supervisor call

This instruction causes an exception to be taken to EL1.

On executing an `SVC` instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC syndrome value `0x15`, and the value of the immediate argument.

31–29	28–26	25–24	23–21	20–5	4–2	1–0
1 1 0	1 0 1	0 0	0 0 0	imm16	0 0 0	0 1
			opc		op2	LL

Encoding

```
SVC  #<imm>
```

Decode for this encoding

```
let imm : bits(16) = imm16;
```

Assembler Symbols

<imm>

Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
AArch64_CheckForSVCTrap(imm);
AArch64_CallSupervisor(imm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.466 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory

This instruction atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `SWPA` and `SWPAL` load from memory with acquire semantics.
  - `SWPL` and `SWPAL` store to memory with release semantics.
  - `SWP` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LSE)

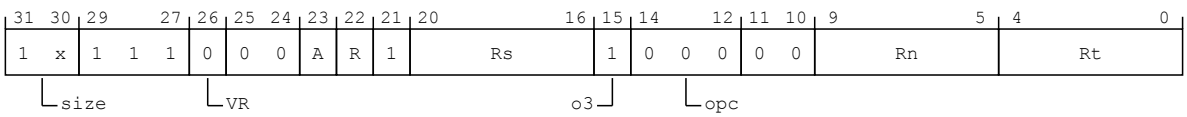

Encoding for the 32-bit SWP variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
SWP  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPA variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
SWPA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPAL variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
SWPAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPL variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
SWPL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit SWP variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
SWP  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPA variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
SWPA  <Xs>, <Xt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SWPAL variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
SWPAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPL variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
SWPL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;

let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let regsize : integer{} = if datasize == 64 then 64 else 32;
let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
var address : bits(64);
var data : bits(datasize);
var store_value : bits(datasize);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = X{datasize}(s);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, store_value, accdesc);

X{regsize}(t) = ZeroExtend{regsize}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.467 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory

This instruction atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAB` and `SWPALB` load from memory with acquire semantics.
  - `SWPLB` and `SWPALB` store to memory with release semantics.
  - `SWPB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LSE)

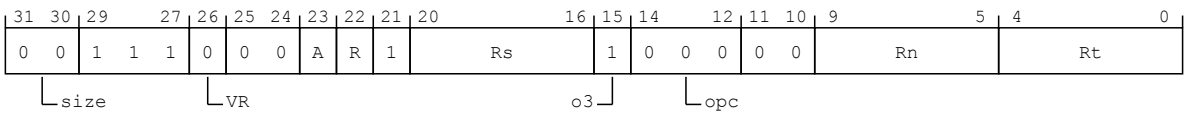

Encoding for the SWPB variant

Applies when `(A == 0 && R == 0)`

```
SWPB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPAB variant

Applies when `(A == 1 && R == 0)`

```
SWPAB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPALB variant

Applies when `(A == 1 && R == 1)`

```
SWPALB  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPLB variant

Applies when `(A == 0 && R == 1)`

```
SWPLB  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;

let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(8);
var store_value : bits(8);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = X{8}(s);

let comparevalue : bits(8) = ARBITRARY : bits(8); // Irrelevant when not executing CAS
data = MemAtomic{8}(address, comparevalue, store_value, accdesc);

X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.468 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory

This instruction atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAH` and `SWPALH` load from memory with acquire semantics.
  - `SWPLH` and `SWPALH` store to memory with release semantics.
  - `SWPH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer

(FEAT_LSE)

31	30	29	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	1	0	0	0	0	0	Rn	Rt	
size				VR								o3	opc							

Encoding for the SWPH variant

Applies when `(A == 0 && R == 0)`

```
SWPH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPAH variant

Applies when `(A == 1 && R == 0)`

```
SWPAH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPALH variant

Applies when `(A == 1 && R == 1)`

```
SWPALH  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPLH variant

Applies when `(A == 0 && R == 1)`

```
SWPLH  <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF); end;

let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
var data : bits(16);
var store_value : bits(16);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = X{16}(s);

let comparevalue : bits(16) = ARBITRARY : bits(16); // Irrelevant when not executing CAS
data = MemAtomic{16}(address, comparevalue, store_value, accdesc);

X{32}(t) = ZeroExtend{32}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.469 SWPP, SWPPA, SWPPAL, SWPPL

Swap quadword in memory

This instruction atomically loads a 128-bit quadword from a memory location, and stores the value held in a pair of registers back to the same memory location. The value initially loaded from memory is returned in the same pair of registers.

- `SWPPA` and `SWPPAL` load from memory with acquire semantics.
  - `SWPPL` and `SWPPAL` store to memory with release semantics.
  - `SWPP` has neither acquire nor release semantics.

Integer

(FEAT_LSE128)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	0	0	1	1	0	0	1	A	R	1	Rt2				1	0	0	0		0	0	Rn				Rt			
	S														o3		opc												

Encoding for the SWPP variant

Applies when `(A == 0 && R == 0)`

```
SWPP  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the SWPPA variant

Applies when `(A == 1 && R == 0)`

```
SWPPA  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the SWPPAL variant

Applies when `(A == 1 && R == 1)`

```
SWPPAL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Encoding for the SWPPL variant

Applies when `(A == 0 && R == 1)`

```
SWPPL  <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then EndOfDecode(Decode_UNDEF); end;
if Rt  == '11111' then EndOfDecode(Decode_UNDEF); end;
if Rt2 == '11111' then EndOfDecode(Decode_UNDEF); end;

let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);

let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly 128-bit Atomic Memory Operations.

Assembler Symbols

<Xt1>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
var address : bits(64);
let value1 : bits(64) = X{}(t);
let value2 : bits(64) = X{}(t2);
var data : bits(128);
var store_value : bits(128);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                       tagchecked, privileged, t, t2, t, t2);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = if BigEndian(accdesc.acctype) then value1::value2 else value2::value1;

let comparevalue : bits(128) = ARBITRARY : bits(128); // Irrelevant when not executing CAS
data = MemAtomic{128}(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = ARBITRARY : bits(128);
end;

if BigEndian(accdesc.acctype) then
    X{64}(t)  = data[127:64];
    X{64}(t2) = data[63:0];
else
    X{64}(t)  = data[63:0];
    X{64}(t2) = data[127:64];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.470 SWPT, SWPTA, SWPTAL, SWPTL

Swap unprivileged

This instruction atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `SWPTA` and `SWPTAL` load from memory with acquire semantics.
  - `SWPTL` and `SWPTAL` store to memory with release semantics.
  - `SWPT` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_LSUI)

31	30	29					24	23	22	21	20			16	15	14			12	11	10	9			5	4			0
0	sz	0	1	1	0	0	1	A	R	1	Rs				1	0	0	0		0	1	Rn				Rt			
															o3	opc													

Encoding for the 32-bit SWPT variant

Applies when `(sz == 0 && A == 0 && R == 0)`

```
SWPT  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPTA variant

Applies when `(sz == 0 && A == 1 && R == 0)`

```
SWPTA  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPTAL variant

Applies when `(sz == 0 && A == 1 && R == 1)`

```
SWPTAL  <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPTL variant

Applies when `(sz == 0 && A == 0 && R == 1)`

```
SWPTL  <Ws>, <Wt>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SWPT variant

Applies when `(sz == 1 && A == 0 && R == 0)`

```
SWPT  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPTA variant

Applies when `(sz == 1 && A == 1 && R == 0)`

```
SWPTA  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPTAL variant

Applies when `(sz == 1 && A == 1 && R == 1)`

```
SWPTAL  <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPTL variant

Applies when `(sz == 1 && A == 0 && R == 1)`

```
SWPTL  <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSUI) then EndOfDecode(Decode_UNDEF); end;

let s : integer{} = UInt(Rs);
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);

let datasize : integer{} = 32 << UInt(sz);
let regsize : integer{} = if datasize == 64 then 64 else 32;
let acquire : boolean = A == '1' && Rt != '11111';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs>

Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
var address : bits(64);
var data : bits(datasize);
var store_value : bits(datasize);

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let accdesc : AccessDescriptor = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                       tagchecked, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

store_value = X{datasize}(s);

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, store_value, accdesc);

X{regsize}(t) = ZeroExtend{regsize}(data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.471 SXTB

Signed extend byte

This instruction extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0	1 0 0 1 1 0	N	0 0 0 0 0 0	0 0 0 1 1 1	Rn	Rd
	opc			immr	imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SXTB  <Wd>, <Wn>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SXTB  <Xd>, <Wn>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.472 SXTH

Sign extend halfword

This instruction extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	0 0	1 0 0 1 1 0	N	0 0 0 0 0 0	0 0 1 1 1 1	Rn	Rd
	opc			immr	imms		

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
SXTH  <Wd>, <Wn>
```

is equivalent to

```
SBFM  <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
SXTH  <Xd>, <Wn>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.473 SXTW

Sign extend word

This instruction sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

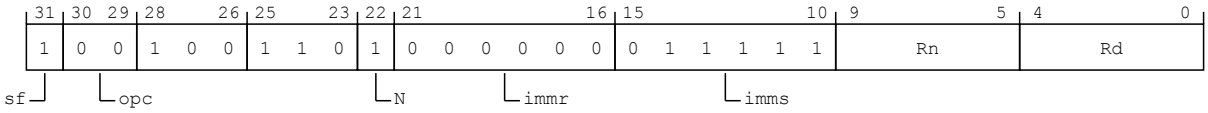

Encoding

```
SXTW  <Xd>, <Wn>
```

is equivalent to

```
SBFM  <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.474 SYS

System instruction

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions for the encodings of System instructions.

This instruction is used by the aliases APAS, AT, BRB, CFP, COSP, CPP, DC, DVP, GCSPOPCX, GCSPOPX, GCSPUSHM, GCSPUSHX, GCSSS1, IC, TLBI, and TRCIT.

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	0	0 1	op1	CRn	CRm	op2	Rt
			L						

Encoding

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

Decode for this encoding

```
let t : integer       = UInt(Rt);
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '01';
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
APAS	op1 == '110' && CRn == '0111' && CRm == '0000' && op2 == '000'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
AT	CRn == '0111' && CRm IN {'100x'} && SysOp(op1, '0111', CRm, op2) == Sys_AT
BRB	op1 == '001' && CRn == '0111' && CRm == '0010' && SysOp('001', '0111', '0010', op2) == Sys_BRB
CFP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'
COSP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '110'
CPP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'
DC	CRn == '0111' && SysOp(op1, '0111', CRm, op2) == Sys_DC
DVP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'
GCSPOPCX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '101'
GCSPOPX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '110'
GCSPUSHM	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '000'
GCSPUSHX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '100'
GCSSS1	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '010'
IC	CRn == '0111' && SysOp(op1, '0111', CRm, op2) == Sys_IC
TLBI	CRn IN {'100x'} && SysOp(op1, CRn, CRm, op2) == Sys_TLBI
TRCIT	op1 == '011' && CRn == '0111' && CRm == '0010' && op2 == '111'

Operation

```
AArch64_SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.475 SYSL

System instruction with result

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions for the encodings of System instructions.

This instruction is used by the aliases GCSPOPM and GCSSS2.

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0	1 (L)	0 1	op1	CRn	CRm	op2	Rt

Encoding

```
SYSL  <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

Decode for this encoding

```
let t : integer       = UInt(Rt);
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '01';
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Alias Conditions

Alias	Is preferred when
GCSPOPM	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '001'
GCSSS2	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '011'

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.476 SYSP

128-bit system instruction.

This instruction is used by the alias TLBIP.

System

(FEAT_SYSINSTR128)

31–29	28–26	25–22	21	20–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 1	0 (L)	0 1	op1	CRn	CRm	op2	Rt

Encoding

```
SYSP  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSINSTR128) then EndOfDecode(Decode_UNDEF); end;
if Rt[0] == '1' && Rt != '11111' then EndOfDecode(Decode_UNDEF); end;

let t : integer       = UInt(Rt);
let t2 : integer      = if t == 31 then 31 else t + 1;
let sys_L : bits(1)   = L;
let sys_op0 : bits(2) = '01';
let sys_op1 : bits(3) = op1;
let sys_op2 : bits(3) = op2;
let sys_crn : bits(4) = CRn;
let sys_crm : bits(4) = CRm;
```

Assembler Symbols

<op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn>

Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm>

Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “CRm” field.

<op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<Xt1>

Is the 64-bit name of the first optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second optional general-purpose source register, defaulting to ‘11111’, encoded as “Rt” +1. Defaults to ‘11111’ if “Rt” = ‘11111’.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
TLBIP	CRn IN {'100x'} && SysOp128(op1, CRn, CRm, op2) == Sys_TLBIP

Operation

```
AArch64_SysInstr128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.477 TBNZ

Test bit and branch if nonzero

This instruction compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

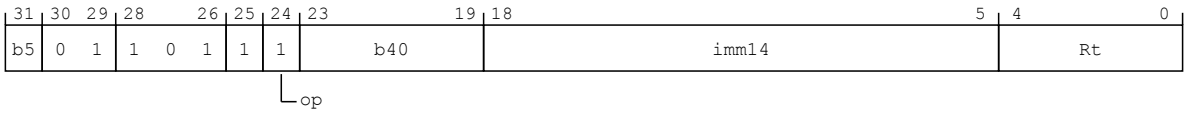

Encoding

```
TBNZ  <R><t>, #<imm>, <label>
```

Decode for this encoding

```
let t : integer = UInt(Rt);

let datasize : integer{} = 32 << UInt(b5);
let bit_pos : integer = UInt(b5::b40);
let offset : bits(64) = SignExtend{}(imm14::'00');
```

Assembler Symbols

<R>

Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an ‘X’ specifier is always permitted, but a ‘W’ specifier is only permitted when the bit number is less than 32.

<t>

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

<imm>

Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as “imm14” times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand : bits(datasize) = X{}(t);
let branch_conditional : boolean = TRUE;
if operand[bit_pos] != '0' then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.478 TBZ

Test bit and branch if zero

This instruction compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

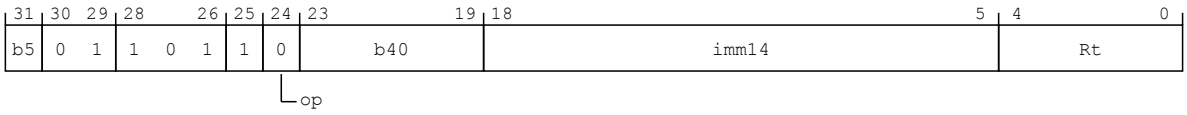

Encoding

```
TBZ  <R><t>, #<imm>, <label>
```

Decode for this encoding

```
let t : integer = UInt(Rt);

let datasize : integer{} = 32 << UInt(b5);
let bit_pos : integer = UInt(b5::b40);
let offset : bits(64) = SignExtend{}(imm14::'00');
```

Assembler Symbols

<R>

Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an ‘X’ specifier is always permitted, but a ‘W’ specifier is only permitted when the bit number is less than 32.

<t>

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

<imm>

Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

<label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as “imm14” times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand : bits(datasize) = X{}(t);
let branch_conditional : boolean = TRUE;
if operand[bit_pos] == '0' then
    BranchTo{64}(PC64() + offset, BranchType_DIR, branch_conditional);
else
    BranchNotTaken(BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.479 TLBI

TLB invalidate operation

For more information, see op0 == 0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

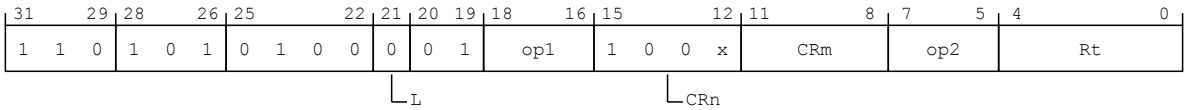

Encoding

```
TLBI  <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1, CRn, CRm, op2) == Sys_TLBI`.

Assembler Symbols

<tlbi_op>

Is a TLBI operation name, as listed for the TLBI system instruction group, encoded in “op1:CRn:CRm:op2”:

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1000	0001	000	VMALLE1OS	FEAT_TLBIOS	For more information, see TLBI VMALLE1OS, TLBI VMALLE1OSNXS.
000	1000	0001	001	VAE1OS	FEAT_TLBIOS	For more information, see TLBI VAE1OS, TLBI VAE1OSNXS.
000	1000	0001	010	ASIDE1OS	FEAT_TLBIOS	For more information, see TLBI ASIDE1OS, TLBI ASIDE1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1000	0001	011	VAAE1OS	FEAT_TLBIOS	For more information, see TLBI VAAE1OS, TLBI VAAE1OSNXS.
000	1000	0001	101	VALE1OS	FEAT_TLBIOS	For more information, see TLBI VALE1OS, TLBI VALE1OSNXS.
000	1000	0001	111	VAALE1OS	FEAT_TLBIOS	For more information, see TLBI VAALE1OS, TLBI VAALE1OSNXS.
000	1000	0010	001	RVAE1IS	FEAT_TLBIRANGE	For more information, see TLBI RVAE1IS, TLBI RVAE1ISNXS.
000	1000	0010	011	RVAAE1IS	FEAT_TLBIRANGE	For more information, see TLBI RVAAE1IS, TLBI RVAAE1ISNXS.
000	1000	0010	101	RVALE1IS	FEAT_TLBIRANGE	For more information, see TLBI RVALE1IS, TLBI RVALE1ISNXS.
000	1000	0010	111	RVAALE1IS	FEAT_TLBIRANGE	For more information, see TLBI RVAALE1IS, TLBI RVAALE1ISNXS.
000	1000	0011	000	VMALLE1IS	-	For more information, see TLBI VMALLE1IS, TLBI VMALLE1ISNXS.
000	1000	0011	001	VAE1IS	-	For more information, see TLBI VAE1IS, TLBI VAE1ISNXS.
000	1000	0011	010	ASIDE1IS	-	For more information, see TLBI ASIDE1IS, TLBI ASIDE1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1000	0011	011	VAAE1IS	-	For more information, see TLBI VAAE1IS, TLBI VAAE1ISNXS.
000	1000	0011	101	VALE1IS	-	For more information, see TLBI VALE1IS, TLBI VALE1ISNXS.
000	1000	0011	111	VAALE1IS	-	For more information, see TLBI VAALE1IS, TLBI VAALE1ISNXS.
000	1000	0101	001	RVAE1OS	FEAT_TLBIRANGE	For more information, see TLBI RVAE1OS, TLBI RVAE1OSNXS.
000	1000	0101	011	RVAAE1OS	FEAT_TLBIRANGE	For more information, see TLBI RVAAE1OS, TLBI RVAAE1OSNXS.
000	1000	0101	101	RVALE1OS	FEAT_TLBIRANGE	For more information, see TLBI RVALE1OS, TLBI RVALE1OSNXS.
000	1000	0101	111	RVAALE1OS	FEAT_TLBIRANGE	For more information, see TLBI RVAALE1OS, TLBI RVAALE1OSNXS.
000	1000	0110	001	RVAE1	FEAT_TLBIRANGE	For more information, see TLBI RVAE1, TLBI RVAE1NXS.
000	1000	0110	011	RVAAE1	FEAT_TLBIRANGE	For more information, see TLBI RVAAE1, TLBI RVAAE1NXS.
000	1000	0110	101	RVALE1	FEAT_TLBIRANGE	For more information, see TLBI RVALE1, TLBI RVALE1NXS.
000	1000	0110	111	RVAALE1	FEAT_TLBIRANGE	For more information, see TLBI RVAALE1, TLBI RVAALE1NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1000	0111	000	VMALLE1	-	For more information, see TLBI VMALLE1, TLBI VMALLE1NXS.
000	1000	0111	001	VAE1	-	For more information, see TLBI VAE1, TLBI VAE1NXS.
000	1000	0111	010	ASIDE1	-	For more information, see TLBI ASIDE1, TLBI ASIDE1NXS.
000	1000	0111	011	VAAE1	-	For more information, see TLBI VAAE1, TLBI VAAE1NXS.
000	1000	0111	101	VALE1	-	For more information, see TLBI VALE1, TLBI VALE1NXS.
000	1000	0111	111	VAALE1	-	For more information, see TLBI VAALE1, TLBI VAALE1NXS.
000	1001	0001	000	VMALLE1OSNXS	FEAT_XS	For more information, see TLBI VMALLE1OS, TLBI VMALLE1OSNXS.
000	1001	0001	001	VAE1OSNXS	FEAT_XS	For more information, see TLBI VAE1OS, TLBI VAE1OSNXS.
000	1001	0001	010	ASIDE1OSNXS	FEAT_XS	For more information, see TLBI ASIDE1OS, TLBI ASIDE1OSNXS.
000	1001	0001	011	VAAE1OSNXS	FEAT_XS	For more information, see TLBI VAAE1OS, TLBI VAAE1OSNXS.
000	1001	0001	101	VALE1OSNXS	FEAT_XS	For more information, see TLBI VALE1OS, TLBI VALE1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1001	0001	111	VAALE1OSNXS	FEAT_XS	For more information, see TLBI VAALE1OS, TLBI VAALE1OSNXS.
000	1001	0010	001	RVAE1ISNXS	FEAT_XS	For more information, see TLBI RVAE1IS, TLBI RVAE1ISNXS.
000	1001	0010	011	RVAAE1ISNXS	FEAT_XS	For more information, see TLBI RVAAE1IS, TLBI RVAAE1ISNXS.
000	1001	0010	101	RVALE1ISNXS	FEAT_XS	For more information, see TLBI RVALE1IS, TLBI RVALE1ISNXS.
000	1001	0010	111	RVAALE1ISNXS	FEAT_XS	For more information, see TLBI RVAALE1IS, TLBI RVAALE1ISNXS.
000	1001	0011	000	VMALLE1ISNXS	FEAT_XS	For more information, see TLBI VMALLE1IS, TLBI VMALLE1ISNXS.
000	1001	0011	001	VAE1ISNXS	FEAT_XS	For more information, see TLBI VAE1IS, TLBI VAE1ISNXS.
000	1001	0011	010	ASIDE1ISNXS	FEAT_XS	For more information, see TLBI ASIDE1IS, TLBI ASIDE1ISNXS.
000	1001	0011	011	VAAE1ISNXS	FEAT_XS	For more information, see TLBI VAAE1IS, TLBI VAAE1ISNXS.
000	1001	0011	101	VALE1ISNXS	FEAT_XS	For more information, see TLBI VALE1IS, TLBI VALE1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1001	0011	111	VAALE1ISNXS	FEAT_XS	For more information, see TLBI VAALE1IS, TLBI VAALE1ISNXS.
000	1001	0101	001	RVAE1OSNXS	FEAT_XS	For more information, see TLBI RVAE1OS, TLBI RVAE1OSNXS.
000	1001	0101	011	RVAAE1OSNXS	FEAT_XS	For more information, see TLBI RVAAE1OS, TLBI RVAAE1OSNXS.
000	1001	0101	101	RVALE1OSNXS	FEAT_XS	For more information, see TLBI RVALE1OS, TLBI RVALE1OSNXS.
000	1001	0101	111	RVAALE1OSNXS	FEAT_XS	For more information, see TLBI RVAALE1OS, TLBI RVAALE1OSNXS.
000	1001	0110	001	RVAE1NXS	FEAT_XS	For more information, see TLBI RVAE1, TLBI RVAE1NXS.
000	1001	0110	011	RVAAE1NXS	FEAT_XS	For more information, see TLBI RVAAE1, TLBI RVAAE1NXS.
000	1001	0110	101	RVALE1NXS	FEAT_XS	For more information, see TLBI RVALE1, TLBI RVALE1NXS.
000	1001	0110	111	RVAALE1NXS	FEAT_XS	For more information, see TLBI RVAALE1, TLBI RVAALE1NXS.
000	1001	0111	000	VMALLE1NXS	FEAT_XS	For more information, see TLBI VMALLE1, TLBI VMALLE1NXS.
000	1001	0111	001	VAE1NXS	FEAT_XS	For more information, see TLBI VAE1, TLBI VAE1NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
000	1001	0111	010	ASIDE1NXS	FEAT_XS	For more information, see TLBI ASIDE1, TLBI ASIDE1NXS.
000	1001	0111	011	VAAE1NXS	FEAT_XS	For more information, see TLBI VAAE1, TLBI VAAE1NXS.
000	1001	0111	101	VALE1NXS	FEAT_XS	For more information, see TLBI VALE1, TLBI VALE1NXS.
000	1001	0111	111	VAALE1NXS	FEAT_XS	For more information, see TLBI VAALE1, TLBI VAALE1NXS.
100	1000	0000	001	IPAS2E1IS	-	For more information, see TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS.
100	1000	0000	010	RIPAS2E1IS	FEAT_TLBIRANGE	For more information, see TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS.
100	1000	0000	101	IPAS2LE1IS	-	For more information, see TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS.
100	1000	0000	110	RIPAS2LE1IS	FEAT_TLBIRANGE	For more information, see TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS.
100	1000	0001	000	ALLE2OS	FEAT_TLBIOS	For more information, see TLBI ALLE2OS, TLBI ALLE2OSNXS.
100	1000	0001	001	VAE2OS	FEAT_TLBIOS	For more information, see TLBI VAE2OS, TLBI VAE2OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1000	0001	100	ALLE1OS	FEAT_TLBIOS	For more information, see TLBI ALLE1OS, TLBI ALLE1OSNXS.
100	1000	0001	101	VALE2OS	FEAT_TLBIOS	For more information, see TLBI VALE2OS, TLBI VALE2OSNXS.
100	1000	0001	110	VMALLS12E1OS	FEAT_TLBIOS	For more information, see TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS.
100	1000	0010	001	RVAE2IS	FEAT_TLBIRANGE	For more information, see TLBI RVAE2IS, TLBI RVAE2ISNXS.
100	1000	0010	010	VMALLWS2E1IS	FEAT_TLBIW	For more information, see TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS.
100	1000	0010	101	RVALE2IS	FEAT_TLBIRANGE	For more information, see TLBI RVALE2IS, TLBI RVALE2ISNXS.
100	1000	0011	000	ALLE2IS	-	For more information, see TLBI ALLE2IS, TLBI ALLE2ISNXS.
100	1000	0011	001	VAE2IS	-	For more information, see TLBI VAE2IS, TLBI VAE2ISNXS.
100	1000	0011	100	ALLE1IS	-	For more information, see TLBI ALLE1IS, TLBI ALLE1ISNXS.
100	1000	0011	101	VALE2IS	-	For more information, see TLBI VALE2IS, TLBI VALE2ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1000	0011	110	VMALLS12E1IS	-	For more information, see TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS.
100	1000	0100	000	IPAS2E1OS	FEAT_TLBIOS	For more information, see TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS.
100	1000	0100	001	IPAS2E1	-	For more information, see TLBI IPAS2E1, TLBI IPAS2E1NXS.
100	1000	0100	010	RIPAS2E1	FEAT_TLBIRANGE	For more information, see TLBI RIPAS2E1, TLBI RIPAS2E1NXS.
100	1000	0100	011	RIPAS2E1OS	FEAT_TLBIRANGE	For more information, see TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS.
100	1000	0100	100	IPAS2LE1OS	FEAT_TLBIOS	For more information, see TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS.
100	1000	0100	101	IPAS2LE1	-	For more information, see TLBI IPAS2LE1, TLBI IPAS2LE1NXS.
100	1000	0100	110	RIPAS2LE1	FEAT_TLBIRANGE	For more information, see TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS.
100	1000	0100	111	RIPAS2LE1OS	FEAT_TLBIRANGE	For more information, see TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS.
100	1000	0101	001	RVAE2OS	FEAT_TLBIRANGE	For more information, see TLBI RVAE2OS, TLBI RVAE2OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1000	0101	010	VMALLWS2E1OS	FEAT_TLBIW	For more information, see TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS.
100	1000	0101	101	RVALE2OS	FEAT_TLBIRANGE	For more information, see TLBI RVALE2OS, TLBI RVALE2OSNXS.
100	1000	0110	001	RVAE2	FEAT_TLBIRANGE	For more information, see TLBI RVAE2, TLBI RVAE2NXS.
100	1000	0110	010	VMALLWS2E1	FEAT_TLBIW	For more information, see TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS.
100	1000	0110	101	RVALE2	FEAT_TLBIRANGE	For more information, see TLBI RVALE2, TLBI RVALE2NXS.
100	1000	0111	000	ALLE2	-	For more information, see TLBI ALLE2, TLBI ALLE2NXS.
100	1000	0111	001	VAE2	-	For more information, see TLBI VAE2, TLBI VAE2NXS.
100	1000	0111	100	ALLE1	-	For more information, see TLBI ALLE1, TLBI ALLE1NXS.
100	1000	0111	101	VALE2	-	For more information, see TLBI VALE2, TLBI VALE2NXS.
100	1000	0111	110	VMALLS12E1	-	For more information, see TLBI VMALLS12E1, TLBI VMALLS12E1NXS.
100	1001	0000	001	IPAS2E1ISNXS	FEAT_XS	For more information, see TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_XS	For more information, see TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS.
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_XS	For more information, see TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS.
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_XS	For more information, see TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS.
100	1001	0001	000	ALLE2OSNXS	FEAT_XS	For more information, see TLBI ALLE2OS, TLBI ALLE2OSNXS.
100	1001	0001	001	VAE2OSNXS	FEAT_XS	For more information, see TLBI VAE2OS, TLBI VAE2OSNXS.
100	1001	0001	100	ALLE1OSNXS	FEAT_XS	For more information, see TLBI ALLE1OS, TLBI ALLE1OSNXS.
100	1001	0001	101	VALE2OSNXS	FEAT_XS	For more information, see TLBI VALE2OS, TLBI VALE2OSNXS.
100	1001	0001	110	VMALLS12E1OSNXS	FEAT_XS	For more information, see TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS.
100	1001	0010	001	RVAE2ISNXS	FEAT_XS	For more information, see TLBI RVAE2IS, TLBI RVAE2ISNXS.
100	1001	0010	010	VMALLWS2E1ISNXS	FEAT_TLBIW	For more information, see TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1001	0010	101	RVALE2ISNXS	FEAT_XS	For more information, see TLBI RVALE2IS, TLBI RVALE2ISNXS.
100	1001	0011	000	ALLE2ISNXS	FEAT_XS	For more information, see TLBI ALLE2IS, TLBI ALLE2ISNXS.
100	1001	0011	001	VAE2ISNXS	FEAT_XS	For more information, see TLBI VAE2IS, TLBI VAE2ISNXS.
100	1001	0011	100	ALLE1ISNXS	FEAT_XS	For more information, see TLBI ALLE1IS, TLBI ALLE1ISNXS.
100	1001	0011	101	VALE2ISNXS	FEAT_XS	For more information, see TLBI VALE2IS, TLBI VALE2ISNXS.
100	1001	0011	110	VMALLS12E1ISNXS	FEAT_XS	For more information, see TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS.
100	1001	0100	000	IPAS2E1OSNXS	FEAT_XS	For more information, see TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS.
100	1001	0100	001	IPAS2E1NXS	FEAT_XS	For more information, see TLBI IPAS2E1, TLBI IPAS2E1NXS.
100	1001	0100	010	RIPAS2E1NXS	FEAT_XS	For more information, see TLBI RIPAS2E1, TLBI RIPAS2E1NXS.
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_XS	For more information, see TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_XS	For more information, see TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS.
100	1001	0100	101	IPAS2LE1NXS	FEAT_XS	For more information, see TLBI IPAS2LE1, TLBI IPAS2LE1NXS.
100	1001	0100	110	RIPAS2LE1NXS	FEAT_XS	For more information, see TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS.
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_XS	For more information, see TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS.
100	1001	0101	001	RVAE2OSNXS	FEAT_XS	For more information, see TLBI RVAE2OS, TLBI RVAE2OSNXS.
100	1001	0101	010	VMALLWS2E1OSNXS	FEAT_TLBIW	For more information, see TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS.
100	1001	0101	101	RVALE2OSNXS	FEAT_XS	For more information, see TLBI RVALE2OS, TLBI RVALE2OSNXS.
100	1001	0110	001	RVAE2NXS	FEAT_XS	For more information, see TLBI RVAE2, TLBI RVAE2NXS.
100	1001	0110	010	VMALLWS2E1NXS	FEAT_TLBIW	For more information, see TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS.
100	1001	0110	101	RVALE2NXS	FEAT_XS	For more information, see TLBI RVALE2, TLBI RVALE2NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
100	1001	0111	000	ALLE2NXS	FEAT_XS	For more information, see TLBI ALLE2, TLBI ALLE2NXS.
100	1001	0111	001	VAE2NXS	FEAT_XS	For more information, see TLBI VAE2, TLBI VAE2NXS.
100	1001	0111	100	ALLE1NXS	FEAT_XS	For more information, see TLBI ALLE1, TLBI ALLE1NXS.
100	1001	0111	101	VALE2NXS	FEAT_XS	For more information, see TLBI VALE2, TLBI VALE2NXS.
100	1001	0111	110	VMALLS12E1NXS	FEAT_XS	For more information, see TLBI VMALLS12E1, TLBI VMALLS12E1NXS.
110	1000	0001	000	ALLE3OS	FEAT_TLBIOS	For more information, see TLBI ALLE3OS, TLBI ALLE3OSNXS.
110	1000	0001	001	VAE3OS	FEAT_TLBIOS	For more information, see TLBI VAE3OS, TLBI VAE3OSNXS.
110	1000	0001	100	PAALLOS	FEAT_RME	For more information, see TLBI PAALLOS.
110	1000	0001	101	VALE3OS	FEAT_TLBIOS	For more information, see TLBI VALE3OS, TLBI VALE3OSNXS.
110	1000	0010	001	RVAE3IS	FEAT_TLBIRANGE	For more information, see TLBI RVAE3IS, TLBI RVAE3ISNXS.
110	1000	0010	101	RVALE3IS	FEAT_TLBIRANGE	For more information, see TLBI RVALE3IS, TLBI RVALE3ISNXS.
110	1000	0011	000	ALLE3IS	-	For more information, see TLBI ALLE3IS, TLBI ALLE3ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
110	1000	0011	001	VAE3IS	-	For more information, see TLBI VAE3IS, TLBI VAE3ISNXS.
110	1000	0011	101	VALE3IS	-	For more information, see TLBI VALE3IS, TLBI VALE3ISNXS.
110	1000	0100	011	RPAOS	FEAT_RME	For more information, see TLBI RPAOS.
110	1000	0100	111	RPALOS	FEAT_RME	For more information, see TLBI RPALOS.
110	1000	0101	001	RVAE3OS	FEAT_TLBIRANGE	For more information, see TLBI RVAE3OS, TLBI RVAE3OSNXS.
110	1000	0101	101	RVALE3OS	FEAT_TLBIRANGE	For more information, see TLBI RVALE3OS, TLBI RVALE3OSNXS.
110	1000	0110	001	RVAE3	FEAT_TLBIRANGE	For more information, see TLBI RVAE3, TLBI RVAE3NXS.
110	1000	0110	101	RVALE3	FEAT_TLBIRANGE	For more information, see TLBI RVALE3, TLBI RVALE3NXS.
110	1000	0111	000	ALLE3	-	For more information, see TLBI ALLE3, TLBI ALLE3NXS.
110	1000	0111	001	VAE3	-	For more information, see TLBI VAE3, TLBI VAE3NXS.
110	1000	0111	100	PAALL	FEAT_RME	For more information, see TLBI PAALL.
110	1000	0111	101	VALE3	-	For more information, see TLBI VALE3, TLBI VALE3NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
110	1001	0001	000	ALLE3OSNXS	FEAT_XS	For more information, see TLBI ALLE3OS, TLBI ALLE3OSNXS.
110	1001	0001	001	VAE3OSNXS	FEAT_XS	For more information, see TLBI VAE3OS, TLBI VAE3OSNXS.
110	1001	0001	101	VALE3OSNXS	FEAT_XS	For more information, see TLBI VALE3OS, TLBI VALE3OSNXS.
110	1001	0010	001	RVAE3ISNXS	FEAT_XS	For more information, see TLBI RVAE3IS, TLBI RVAE3ISNXS.
110	1001	0010	101	RVALE3ISNXS	FEAT_XS	For more information, see TLBI RVALE3IS, TLBI RVALE3ISNXS.
110	1001	0011	000	ALLE3ISNXS	FEAT_XS	For more information, see TLBI ALLE3IS, TLBI ALLE3ISNXS.
110	1001	0011	001	VAE3ISNXS	FEAT_XS	For more information, see TLBI VAE3IS, TLBI VAE3ISNXS.
110	1001	0011	101	VALE3ISNXS	FEAT_XS	For more information, see TLBI VALE3IS, TLBI VALE3ISNXS.
110	1001	0101	001	RVAE3OSNXS	FEAT_XS	For more information, see TLBI RVAE3OS, TLBI RVAE3OSNXS.
110	1001	0101	101	RVALE3OSNXS	FEAT_XS	For more information, see TLBI RVALE3OS, TLBI RVALE3OSNXS.
110	1001	0110	001	RVAE3NXS	FEAT_XS	For more information, see TLBI RVAE3, TLBI RVAE3NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature	Description
110	1001	0110	101	RVALE3NXS	FEAT_XS	For more information, see TLBI RVALE3, TLBI RVALE3NXS.
110	1001	0111	000	ALLE3NXS	FEAT_XS	For more information, see TLBI ALLE3, TLBI ALLE3NXS.
110	1001	0111	001	VAE3NXS	FEAT_XS	For more information, see TLBI VAE3, TLBI VAE3NXS.
110	1001	0111	101	VALE3NXS	FEAT_XS	For more information, see TLBI VALE3, TLBI VALE3NXS.

<Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.480 TLBIP

TLB invalidate pair operation.

This is an alias of SYSP. This means:

- The encodings in this description are named to match the encodings of SYSP.
- The description of SYSP gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SYSINSTR128)

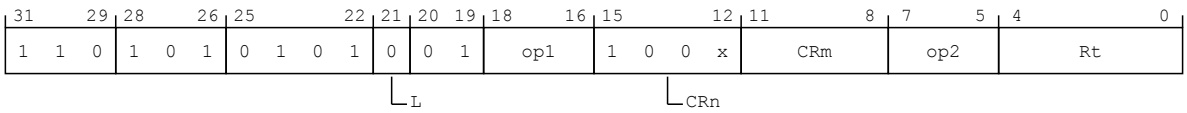

Encoding

```
TLBIP  <tlbip_op>{, <Xt1>, <Xt2>}
```

is equivalent to

```
SYSP  #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}
```

and is the preferred disassembly when `SysOp128(op1, CRn, CRm, op2) == Sys_TLBIP`.

Assembler Symbols

<tlbip_op>

Is a TLBIP operation name, as listed for the TLBIP system pair instruction group, encoded in "op1:CRn:CRm:op2":

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
000	1000	0001	001	VAE1OS	FEAT_D128	For more information, see TLBIP VAE1OS, TLBIP VAE1OSNXS.
000	1000	0001	011	VAAE1OS	FEAT_D128	For more information, see TLBIP VAAE1OS, TLBIP VAAE1OSNXS.
000	1000	0001	101	VALE1OS	FEAT_D128	For more information, see TLBIP VALE1OS, TLBIP VALE1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
000	1000	0001	111	VAALE1OS	FEAT_D128	For more information, see TLBIP VAALE1OS, TLBIP VAALE1OSNXS.
000	1000	0010	001	RVAE1IS	FEAT_D128	For more information, see TLBIP RVAE1IS, TLBIP RVAE1ISNXS.
000	1000	0010	011	RVAAE1IS	FEAT_D128	For more information, see TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS.
000	1000	0010	101	RVALE1IS	FEAT_D128	For more information, see TLBIP RVALE1IS, TLBIP RVALE1ISNXS.
000	1000	0010	111	RVAALE1IS	FEAT_D128	For more information, see TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS.
000	1000	0011	001	VAE1IS	FEAT_D128	For more information, see TLBIP VAE1IS, TLBIP VAE1ISNXS.
000	1000	0011	011	VAAE1IS	FEAT_D128	For more information, see TLBIP VAAE1IS, TLBIP VAAE1ISNXS.
000	1000	0011	101	VALE1IS	FEAT_D128	For more information, see TLBIP VALE1IS, TLBIP VALE1ISNXS.
000	1000	0011	111	VAALE1IS	FEAT_D128	For more information, see TLBIP VAALE1IS, TLBIP VAALE1ISNXS.
000	1000	0101	001	RVAE1OS	FEAT_D128	For more information, see TLBIP RVAE1OS, TLBIP RVAE1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
000	1000	0101	011	RVAAE1OS	FEAT_D128	For more information, see TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS.
000	1000	0101	101	RVALE1OS	FEAT_D128	For more information, see TLBIP RVALE1OS, TLBIP RVALE1OSNXS.
000	1000	0101	111	RVAALE1OS	FEAT_D128	For more information, see TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS.
000	1000	0110	001	RVAE1	FEAT_D128	For more information, see TLBIP RVAE1, TLBIP RVAE1NXS.
000	1000	0110	011	RVAAE1	FEAT_D128	For more information, see TLBIP RVAAE1, TLBIP RVAAE1NXS.
000	1000	0110	101	RVALE1	FEAT_D128	For more information, see TLBIP RVALE1, TLBIP RVALE1NXS.
000	1000	0110	111	RVAALE1	FEAT_D128	For more information, see TLBIP RVAALE1, TLBIP RVAALE1NXS.
000	1000	0111	001	VAE1	FEAT_D128	For more information, see TLBIP VAE1, TLBIP VAE1NXS.
000	1000	0111	011	VAAE1	FEAT_D128	For more information, see TLBIP VAAE1, TLBIP VAAE1NXS.
000	1000	0111	101	VALE1	FEAT_D128	For more information, see TLBIP VALE1, TLBIP VALE1NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
000	1000	0111	111	VAALE1	FEAT_D128	For more information, see TLBIP VAALE1, TLBIP VAALE1NXS.
000	1001	0001	001	VAE1OSNXS	FEAT_D128	For more information, see TLBIP VAE1OS, TLBIP VAE1OSNXS.
000	1001	0001	011	VAAE1OSNXS	FEAT_D128	For more information, see TLBIP VAAE1OS, TLBIP VAAE1OSNXS.
000	1001	0001	101	VALE1OSNXS	FEAT_D128	For more information, see TLBIP VALE1OS, TLBIP VALE1OSNXS.
000	1001	0001	111	VAALE1OSNXS	FEAT_D128	For more information, see TLBIP VAALE1OS, TLBIP VAALE1OSNXS.
000	1001	0010	001	RVAE1ISNXS	FEAT_D128	For more information, see TLBIP RVAE1IS, TLBIP RVAE1ISNXS.
000	1001	0010	011	RVAAE1ISNXS	FEAT_D128	For more information, see TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS.
000	1001	0010	101	RVALE1ISNXS	FEAT_D128	For more information, see TLBIP RVALE1IS, TLBIP RVALE1ISNXS.
000	1001	0010	111	RVAALE1ISNXS	FEAT_D128	For more information, see TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS.
000	1001	0011	001	VAE1ISNXS	FEAT_D128	For more information, see TLBIP VAE1IS, TLBIP VAE1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
000	1001	0011	011	VAAE1ISNXS	FEAT_D128	For more information, see TLBIP VAAE1IS, TLBIP VAAE1ISNXS.
000	1001	0011	101	VALE1ISNXS	FEAT_D128	For more information, see TLBIP VALE1IS, TLBIP VALE1ISNXS.
000	1001	0011	111	VAALE1ISNXS	FEAT_D128	For more information, see TLBIP VAALE1IS, TLBIP VAALE1ISNXS.
000	1001	0101	001	RVAE1OSNXS	FEAT_D128	For more information, see TLBIP RVAE1OS, TLBIP RVAE1OSNXS.
000	1001	0101	011	RVAAE1OSNXS	FEAT_D128	For more information, see TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS.
000	1001	0101	101	RVALE1OSNXS	FEAT_D128	For more information, see TLBIP RVALE1OS, TLBIP RVALE1OSNXS.
000	1001	0101	111	RVAALE1OSNXS	FEAT_D128	For more information, see TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS.
000	1001	0110	001	RVAE1NXS	FEAT_D128	For more information, see TLBIP RVAE1, TLBIP RVAE1NXS.
000	1001	0110	011	RVAAE1NXS	FEAT_D128	For more information, see TLBIP RVAAE1, TLBIP RVAAE1NXS.
000	1001	0110	101	RVALE1NXS	FEAT_D128	For more information, see TLBIP RVALE1, TLBIP RVALE1NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
000	1001	0110	111	RVAALE1NXS	FEAT_D128	For more information, see TLBIP RVAALE1, TLBIP RVAALE1NXS.
000	1001	0111	001	VAE1NXS	FEAT_D128	For more information, see TLBIP VAE1, TLBIP VAE1NXS.
000	1001	0111	011	VAAE1NXS	FEAT_D128	For more information, see TLBIP VAAE1, TLBIP VAAE1NXS.
000	1001	0111	101	VALE1NXS	FEAT_D128	For more information, see TLBIP VALE1, TLBIP VALE1NXS.
000	1001	0111	111	VAALE1NXS	FEAT_D128	For more information, see TLBIP VAALE1, TLBIP VAALE1NXS.
100	1000	0000	001	IPAS2E1IS	FEAT_D128	For more information, see TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS.
100	1000	0000	010	RIPAS2E1IS	FEAT_D128	For more information, see TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS.
100	1000	0000	101	IPAS2LE1IS	FEAT_D128	For more information, see TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS.
100	1000	0000	110	RIPAS2LE1IS	FEAT_D128	For more information, see TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS.
100	1000	0001	001	VAE2OS	FEAT_D128	For more information, see TLBIP VAE2OS, TLBIP VAE2OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
100	1000	0001	101	VALE2OS	FEAT_D128	For more information, see TLBIP VALE2OS, TLBIP VALE2OSNXS.
100	1000	0010	001	RVAE2IS	FEAT_D128	For more information, see TLBIP RVAE2IS, TLBIP RVAE2ISNXS.
100	1000	0010	101	RVALE2IS	FEAT_D128	For more information, see TLBIP RVALE2IS, TLBIP RVALE2ISNXS.
100	1000	0011	001	VAE2IS	FEAT_D128	For more information, see TLBIP VAE2IS, TLBIP VAE2ISNXS.
100	1000	0011	101	VALE2IS	FEAT_D128	For more information, see TLBIP VALE2IS, TLBIP VALE2ISNXS.
100	1000	0100	000	IPAS2E1OS	FEAT_D128	For more information, see TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS.
100	1000	0100	001	IPAS2E1	FEAT_D128	For more information, see TLBIP IPAS2E1, TLBIP IPAS2E1NXS.
100	1000	0100	010	RIPAS2E1	FEAT_D128	For more information, see TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS.
100	1000	0100	011	RIPAS2E1OS	FEAT_D128	For more information, see TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS.
100	1000	0100	100	IPAS2LE1OS	FEAT_D128	For more information, see TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
100	1000	0100	101	IPAS2LE1	FEAT_D128	For more information, see TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS.
100	1000	0100	110	RIPAS2LE1	FEAT_D128	For more information, see TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS.
100	1000	0100	111	RIPAS2LE1OS	FEAT_D128	For more information, see TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS.
100	1000	0101	001	RVAE2OS	FEAT_D128	For more information, see TLBIP RVAE2OS, TLBIP RVAE2OSNXS.
100	1000	0101	101	RVALE2OS	FEAT_D128	For more information, see TLBIP RVALE2OS, TLBIP RVALE2OSNXS.
100	1000	0110	001	RVAE2	FEAT_D128	For more information, see TLBIP RVAE2, TLBIP RVAE2NXS.
100	1000	0110	101	RVALE2	FEAT_D128	For more information, see TLBIP RVALE2, TLBIP RVALE2NXS.
100	1000	0111	001	VAE2	FEAT_D128	For more information, see TLBIP VAE2, TLBIP VAE2NXS.
100	1000	0111	101	VALE2	FEAT_D128	For more information, see TLBIP VALE2, TLBIP VALE2NXS.
100	1001	0000	001	IPAS2E1ISNXS	FEAT_D128	For more information, see TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_D128	For more information, see TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS.
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_D128	For more information, see TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS.
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_D128	For more information, see TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS.
100	1001	0001	001	VAE2OSNXS	FEAT_D128	For more information, see TLBIP VAE2OS, TLBIP VAE2OSNXS.
100	1001	0001	101	VALE2OSNXS	FEAT_D128	For more information, see TLBIP VALE2OS, TLBIP VALE2OSNXS.
100	1001	0010	001	RVAE2ISNXS	FEAT_D128	For more information, see TLBIP RVAE2IS, TLBIP RVAE2ISNXS.
100	1001	0010	101	RVALE2ISNXS	FEAT_D128	For more information, see TLBIP RVALE2IS, TLBIP RVALE2ISNXS.
100	1001	0011	001	VAE2ISNXS	FEAT_D128	For more information, see TLBIP VAE2IS, TLBIP VAE2ISNXS.
100	1001	0011	101	VALE2ISNXS	FEAT_D128	For more information, see TLBIP VALE2IS, TLBIP VALE2ISNXS.
100	1001	0100	000	IPAS2E1OSNXS	FEAT_D128	For more information, see TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
100	1001	0100	001	IPAS2E1NXS	FEAT_D128	For more information, see TLBIP IPAS2E1, TLBIP IPAS2E1NXS.
100	1001	0100	010	RIPAS2E1NXS	FEAT_D128	For more information, see TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS.
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_D128	For more information, see TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS.
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_D128	For more information, see TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS.
100	1001	0100	101	IPAS2LE1NXS	FEAT_D128	For more information, see TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS.
100	1001	0100	110	RIPAS2LE1NXS	FEAT_D128	For more information, see TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS.
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_D128	For more information, see TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS.
100	1001	0101	001	RVAE2OSNXS	FEAT_D128	For more information, see TLBIP RVAE2OS, TLBIP RVAE2OSNXS.
100	1001	0101	101	RVALE2OSNXS	FEAT_D128	For more information, see TLBIP RVALE2OS, TLBIP RVALE2OSNXS.
100	1001	0110	001	RVAE2NXS	FEAT_D128	For more information, see TLBIP RVAE2, TLBIP RVAE2NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
100	1001	0110	101	RVALE2NXS	FEAT_D128	For more information, see TLBIP RVALE2, TLBIP RVALE2NXS.
100	1001	0111	001	VAE2NXS	FEAT_D128	For more information, see TLBIP VAE2, TLBIP VAE2NXS.
100	1001	0111	101	VALE2NXS	FEAT_D128	For more information, see TLBIP VALE2, TLBIP VALE2NXS.
110	1000	0001	001	VAE3OS	FEAT_D128	For more information, see TLBIP VAE3OS, TLBIP VAE3OSNXS.
110	1000	0001	101	VALE3OS	FEAT_D128	For more information, see TLBIP VALE3OS, TLBIP VALE3OSNXS.
110	1000	0010	001	RVAE3IS	FEAT_D128	For more information, see TLBIP RVAE3IS, TLBIP RVAE3ISNXS.
110	1000	0010	101	RVALE3IS	FEAT_D128	For more information, see TLBIP RVALE3IS, TLBIP RVALE3ISNXS.
110	1000	0011	001	VAE3IS	FEAT_D128	For more information, see TLBIP VAE3IS, TLBIP VAE3ISNXS.
110	1000	0011	101	VALE3IS	FEAT_D128	For more information, see TLBIP VALE3IS, TLBIP VALE3ISNXS.
110	1000	0101	001	RVAE3OS	FEAT_D128	For more information, see TLBIP RVAE3OS, TLBIP RVAE3OSNXS.
110	1000	0101	101	RVALE3OS	FEAT_D128	For more information, see TLBIP RVALE3OS, TLBIP RVALE3OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
110	1000	0110	001	RVAE3	FEAT_D128	For more information, see TLBIP RVAE3, TLBIP RVAE3NXS.
110	1000	0110	101	RVALE3	FEAT_D128	For more information, see TLBIP RVALE3, TLBIP RVALE3NXS.
110	1000	0111	001	VAE3	FEAT_D128	For more information, see TLBIP VAE3, TLBIP VAE3NXS.
110	1000	0111	101	VALE3	FEAT_D128	For more information, see TLBIP VALE3, TLBIP VALE3NXS.
110	1001	0001	001	VAE3OSNXS	FEAT_D128	For more information, see TLBIP VAE3OS, TLBIP VAE3OSNXS.
110	1001	0001	101	VALE3OSNXS	FEAT_D128	For more information, see TLBIP VALE3OS, TLBIP VALE3OSNXS.
110	1001	0010	001	RVAE3ISNXS	FEAT_D128	For more information, see TLBIP RVAE3IS, TLBIP RVAE3ISNXS.
110	1001	0010	101	RVALE3ISNXS	FEAT_D128	For more information, see TLBIP RVALE3IS, TLBIP RVALE3ISNXS.
110	1001	0011	001	VAE3ISNXS	FEAT_D128	For more information, see TLBIP VAE3IS, TLBIP VAE3ISNXS.
110	1001	0011	101	VALE3ISNXS	FEAT_D128	For more information, see TLBIP VALE3IS, TLBIP VALE3ISNXS.
110	1001	0101	001	RVAE3OSNXS	FEAT_D128	For more information, see TLBIP RVAE3OS, TLBIP RVAE3OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature	Description
110	1001	0101	101	RVALE3OSNXS	FEAT_D128	For more information, see TLBIP RVALE3OS, TLBIP RVALE3OSNXS.
110	1001	0110	001	RVAE3NXS	FEAT_D128	For more information, see TLBIP RVAE3, TLBIP RVAE3NXS.
110	1001	0110	101	RVALE3NXS	FEAT_D128	For more information, see TLBIP RVALE3, TLBIP RVALE3NXS.
110	1001	0111	001	VAE3NXS	FEAT_D128	For more information, see TLBIP VAE3, TLBIP VAE3NXS.
110	1001	0111	101	VALE3NXS	FEAT_D128	For more information, see TLBIP VALE3, TLBIP VALE3NXS.

<Xt1>

Is the 64-bit name of the first optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

<Xt2>

Is the 64-bit name of the second optional general-purpose source register, defaulting to ‘11111’, encoded as “Rt” +1. Defaults to ‘11111’ if “Rt” = ‘11111’.

Operation

The description of SYSP gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.481 TRCIT

Trace instrumentation

This instruction generates an instrumentation trace packet that contains the value of the provided register.

For more information, see TRCIT, Trace Instrumentation.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_ITE)

31 29	28 26	25 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0	1 0 1	0 1 0 0	0	0 1	0 1 1	0 1 1 1	0 0 1 0	1 1 1	Rt
			L		op1	CRn	CRm	op2	

Encoding

```
TRCIT  <Xt>
```

is equivalent to

```
SYS  #3, C7, C2, #7, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.482 TSB

Trace synchronization barrier

This instruction is a barrier that synchronizes the trace operations of instructions, see Trace Synchronization Barrier (TSB).

If FEAT_TRF is not implemented, this instruction executes as a `NOP`.

System

(FEAT_TRF)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 1 0 (CRm)	0 1 0 (op2)	1 1 1 1 1

Encoding

```
TSB  CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then EndOfDecode(Decode_NOP); end;
```

Operation

```
if IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_TRBEv1p1) then
    let trap_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                 !IsInHost() &&
                                 (!HaveEL(EL3) || SCR_EL3().FGTEn2 == '1') &&
                                 HFGITR2_EL2().TSBCSYNC == '1');
    if trap_to_el2 then
        let target_el : bits(2) = EL2;
        let iss : bits(25)      = 0x4[24:0];
        AArch64_OtherInstrTrap(target_el, iss);
    end;
end;

TraceSynchronizationBarrier();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.483 TST (immediate)

Test bits (immediate)

This instruction performs a bitwise AND between a register value and an immediate value, and discards the results. It updates the condition flags based on the result.

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 1 (opc)	1 0 0 1 0 0	N	immr	imms	Rn	1 1 1 1 1 (Rd)

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
TST  <Wn>, #<imm>
```

is equivalent to

```
ANDS  WZR, <Wn>, #<imm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
TST  <Xn>, #<imm>
```

is equivalent to

```
ANDS  XZR, <Xn>, #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<imm>

Is the bitmask immediate. For encoding information, see Logical (immediate).

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.484 TST (shifted register)

Test (shifted register)

This instruction performs a bitwise AND between a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	1 1 (opc)	0 1 0 1 0	shift	0 (N)	Rm	imm6	Rn	1 1 1 1 1 (Rd)

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
TST  <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS  WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
TST  <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS  XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS (shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.485 UBFIZ

Unsigned bitfield insert in zeros

This instruction copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
UBFIZ  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #(-<lsb>  MOD  32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
UBFIZ  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #(-<lsb>  MOD  64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.486 UBFM

Unsigned bitfield move

This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.

31	30 29	28 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	imms	Rn	Rd

opc

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
UBFM  <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
UBFM  <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && (N != '0' || immr[5] != '0' || imms[5] != '0') then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let s : integer = UInt(imms);
let r : integer = UInt(immr);

var wmask : bits(datasize);
var tmask : bits(datasize);
(wmask, tmask) = DecodeBitMasks{datasize}(N, imms, immr, FALSE);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<immr>

For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imms>

For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Of variant	Is preferred when
LSL (immediate)	32-bit	imms != '011111' && UInt(imms) + 1 == UInt(immr)
LSL (immediate)	64-bit	imms != '111111' && UInt(imms) + 1 == UInt(immr)
LSR (immediate)	32-bit	imms == '011111'
LSR (immediate)	64-bit	imms == '111111'
UBFIZ		UInt(imms) < UInt(immr)
UBFX		BFXPreferred(sf, opc<1>, imms, immr)
UXTB		immr == '000000' && imms == '000111'
UXTH		immr == '000000' && imms == '001111'

Operation

```
let src : bits(datasize) = X{}(n);

// Perform bitfield move on low bits
let bot : bits(datasize) = ROR(src, r) AND wmask;

// Combine extension bits and result bits
X{datasize}(d) = bot AND tmask;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.487 UBFX

Unsigned bitfield extract

This instruction copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28 … 23	22	21 … 16	15 … 10	9 … 5	4 … 0
sf	1 0 (opc)	1 0 0 1 1 0	N	immr	imms	Rn	Rd

Encoding for the 32-bit variant

Applies when `(sf == 0 && N == 0)`

```
UBFX  <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Encoding for the 64-bit variant

Applies when `(sf == 1 && N == 1)`

```
UBFX  <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc[1], imms, immr)`.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “32-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the “64-bit” variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<width>

For the “32-bit” variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the “64-bit” variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.488 UDF

Permanently undefined

This instruction generates an Undefined Instruction exception (ESR_ELx.EC = `0b000000`). The encodings for `UDF` used in this section are defined as permanently UNDEFINED.

31	30	29	28	25	24	16	15	0
0	0	0	0 0 0 0		0 0 0 0 0 0 0 0 0		imm16	

Encoding

```
UDF  #<imm>
```

Decode for this encoding

```
// The imm16 field is ignored by hardware.
EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<imm>

is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field. The PE ignores the value of this constant.

Operation

```
// No operation.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.489 UDIV (quotient)

Unsigned divide

This instruction divides the first unsigned source register value by the second unsigned source register value, and writes the result to the destination register. Dividing by zero writes the value zero to the destination register. The condition flags are not affected.

31	30	29	28	21	20	16	15	11	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 0 0 0 1		0	Rn		Rd	
		S							o1				

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UDIV  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UDIV  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let dividend : integer = UInt(operand1);
let divisor : integer  = UInt(operand2);
var result : integer;
if divisor == 0 then
    result = 0;
else
    result = dividend DIVRM divisor;
end;
X{datasize}(d) = result[datasize-1:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.490 UMADDL

Unsigned multiply-add long

This instruction multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL (32-bit).

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	0	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
UMADDL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
UMULL (32-bit)	Ra == '11111'

Operation

```
let operand1 : bits(32) = X{}(n);
let operand2 : bits(32) = X{}(m);
let operand3 : bits(64) = X{}(a);

let result : integer = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X{64}(d) = result[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.491 UMAX (immediate)

Unsigned maximum (immediate)

This instruction determines the unsigned maximum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	22	21	18	17	10	9	5	4	0
sf	0	0	1 0 0 0 1 1 1		0 0 0 1		imm8		Rn		Rd	
	op	S				opc						

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMAX  <Wd>, <Wn>, #<uimm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMAX  <Xd>, <Xn>, #<uimm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let imm : integer = UInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<uimm>

Is an unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = Max(UInt(operand1), imm);
X{datasize}(d) = result[datasize-1:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.492 UMAX (register)

Unsigned maximum (register)

This instruction determines the unsigned maximum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	21	20	16	15	10	9	5	4	0
sf	0	0	1 1 0 1 0 1 1 0		Rm		0 1 1 0 0 1		Rn		Rd	
		S					opcode					

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMAX  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMAX  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let result : integer = Max(UInt(operand1), UInt(operand2));
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.493 UMIN (immediate)

Unsigned minimum (immediate)

This instruction determines the unsigned minimum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28	22	21	18	17	10	9	5	4	0
sf	0	0	1 0 0 0 1 1 1		0 0 1 1		imm8		Rn		Rd	
	op	S				opc						

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMIN  <Wd>, <Wn>, #<uimm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMIN  <Xd>, <Xn>, #<uimm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 32 << UInt(sf);
let imm : integer = UInt(imm8);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<uimm>

Is an unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
let operand1 : bits(datasize) = X{}(n);
let result : integer = Min(UInt(operand1), imm);
X{datasize}(d) = result[datasize-1:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.494 UMIN (register)

Unsigned minimum (register)

This instruction determines the unsigned minimum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)

31	30	29	28							21	20				16	15						10	9				5	4				0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	1	1		Rn					Rd				
		S																	opcode													

Encoding for the 32-bit variant

Applies when `(sf == 0)`

```
UMIN  <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant

Applies when `(sf == 1)`

```
UMIN  <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 32 << UInt(sf);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
let operand1 : bits(datasize) = X{}(n);
let operand2 : bits(datasize) = X{}(m);
let result : integer = Min(UInt(operand1), UInt(operand2));
X{datasize}(d) = result[datasize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.495 UMNEGL

Unsigned multiply-negate long

This instruction multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	1	1 1 1 1 1	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
UMNEGL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational Information

The description of UMSUBL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.496 UMSUBL

Unsigned multiply-subtract long

This instruction multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	1	Ra	Rn	Rd
sf	op54				U			o0			

Encoding

```
UMSUBL  <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Xa>

Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
UMNEGL	Ra == '11111'

Operation

```
let operand1 : bits(32) = X{}(n);
let operand2 : bits(32) = X{}(m);
let operand3 : bits(64) = X{}(a);

let result : integer = UInt(operand3) - (UInt(operand1) * UInt(operand2));

X{64}(d) = result[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.497 UMULH

Unsigned multiply high

This instruction multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	1 0	Rm	0	(1)(1)(1)(1)(1)	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
UMULH  <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
let operand1 : bits(64) = X{}(n);
let operand2 : bits(64) = X{}(m);

let result : integer = UInt(operand1) * UInt(operand2);

X{64}(d) = result[127:64];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.498 UMULL (32-bit)

Unsigned multiply long

This instruction multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30 29	28	27 25	24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1	1 0 1	1	1	0 1	Rm	0	1 1 1 1 1	Rn	Rd
sf	op54				U			o0	Ra		

Encoding

```
UMULL  <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL  <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Wm>

Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational Information

The description of UMADDL gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.499 UXTB

Unsigned extend byte

This instruction extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

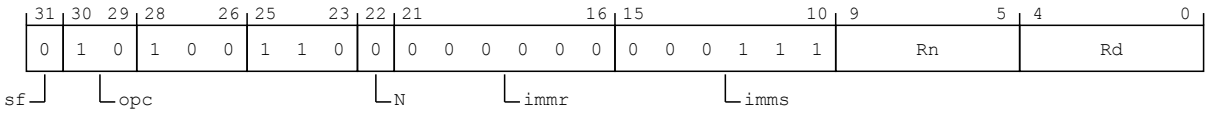

Encoding

```
UXTB  <Wd>, <Wn>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.500 UXTH

Unsigned extend halfword

This instruction extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

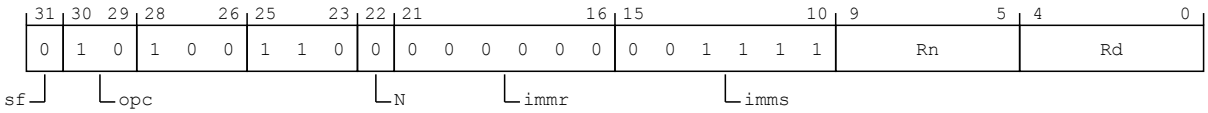

Encoding

```
UXTH  <Wd>, <Wn>
```

is equivalent to

```
UBFM  <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.501 WFE

Wait for event

This instruction is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see Wait For Event mechanism and Send event.

As described in Wait For Event mechanism and Send event, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 1 0 (op2)	1 1 1 1 1

Encoding

```
WFE
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_WFE();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.502 WFET

Wait for event with timeout

This instruction provides a hint that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see Wait For Event mechanism and Send event.

As described in Wait For Event mechanism and Send event, the execution of a `WFET` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 0 1	0 0 0 0 (CRm)	0 0 0 (op2)	Rd

Encoding

```
WFET  <Xt>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_WFxT) then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rd” field.

Operation

```
let localtimeout : integer = UInt(X{64}(d));

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    ExecuteAsNOP();
end;

Hint_WFET(localtimeout);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.503 WFI

Wait for interrupt

This instruction is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see Wait For Interrupt.

As described in Wait For Interrupt, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 1 1 (op2)	1 1 1 1 1

Encoding

```
WFI
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_WFI();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.504 WFIT

Wait for interrupt with timeout

This instruction provides a hint that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see Wait For Interrupt.

As described in Wait For Interrupt, the execution of a `WFIT` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 0 1	0 0 0 0 (CRm)	0 0 1 (op2)	Rd

Encoding

```
WFIT  <Xt>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_WFxT) then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose source register, encoded in the “Rd” field.

Operation

```
let localtimeout : integer = UInt(X{64}(d));

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    ExecuteAsNOP();
end;

Hint_WFIT(localtimeout);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.505 XAFLAG

Convert floating-point condition flags from external format to Arm format

This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System

(FEAT_FlagM2)

31–29	28–26	25–19	18–16	15–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0	0 0 0	0 1 0 0	(0) (0) (0) (0)	0 0 1	1 1 1 1 1
			op1		CRm	op2	Rt

Encoding

```
XAFLAG
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM2) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
let n : bit = NOT(PSTATE.C) AND NOT(PSTATE.Z);
let z : bit = PSTATE.Z AND PSTATE.C;
let c : bit = PSTATE.C OR PSTATE.Z;
let v : bit = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = n;
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = v;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.506 XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code

This instruction removes the Pointer Authentication Code from an address. The address is in the specified general-purpose register for `XPACI` and `XPACD`, and is in LR for `XPACLRI`.

The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLRI` are used for instruction addresses.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)

31	30	29	28							21	20					16	15					11	10	9					5	4				0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1		0	1	0	0	0		D	1	1	1	1	1		Rd				
sf							S									opcode2												Rn						

Encoding for the XPACD variant

Applies when `(D == 1)`

```
XPACD  <Xd>
```

Encoding for the XPACI variant

Applies when `(D == 0)`

```
XPACI  <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let data : boolean = (D == '1');
```

System

(FEAT_PAuth)

31		29	28		26	25											12	11				8	7		5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1
																				CRm					op2					

Encoding

```
XPACLRI
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP); end;
let d : integer = 30;
let data : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
X{64}(d) = Strip(X{64}(d), data);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.507 YIELD

Yield

This instruction is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see The YIELD instruction.

31–29	28–26	25–12	11–8	7–5	4–0
1 1 0	1 0 1	0 1 0 0 0 0 0 0 1 1 0 0 1 0	0 0 0 0 (CRm)	0 0 1 (op2)	1 1 1 1 1

Encoding

```
YIELD
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_Yield();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C7
A64 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes the A64 Advanced SIMD and floating-point instructions.

It contains the following sections:

- About the A64 Advanced SIMD and floating-point instructions.
- Alphabetical list of A64 Advanced SIMD and floating-point instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.1 About the A64 Advanced SIMD and floating-point instructions

Alphabetical list of A64 Advanced SIMD and floating-point instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Loads and store instructions associated with the SIMD and floating-point registers.
- Data processing instructions with SIMD and floating-point registers.

A64 instruction set encoding in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.

The rest of this section is a general description of the SIMD and floating-point instructions. It contains the following subsections:

- Register size.
- Output element control.
- Data types.
- Condition flags and related instructions.

C7.1.1 Register size

A64 provides a comprehensive set of packed Single Instruction Multiple Data (SIMD) and scalar operations using data held in the 32 entry 128-bit wide SIMD and floating-point register file.

Each SIMD and floating-point register can be used to hold:

- A single scalar value of the floating-point or integer type.
- A 64-bit wide vector containing one or more elements.
- A 128-bit wide vector containing two or more elements.

Where the entire 128-bit wide register is not fully utilized, the vector or scalar quantity is held in the least significant bits of the register, with the most significant bits being cleared to zero on a write, see Advanced SIMD vector formats.

The following instructions can insert data into individual elements within a SIMD and floating-pointer register without clearing the remaining bits to zero:

- Insert vector element from another vector element or general-purpose register, `INS`.
- Load structure into a single lane, for example `LD3`.
- All second-part narrowing operations, for example `SHRN2`.

C7.1.2 Output element control

When FEAT_AFP is implemented, the FPCR.NEP bit controls how output elements are determined for the scalar Advanced SIMD instructions for elements other than the lowest element of the vector.

If FPCR.NEP == 1, the following instructions determine output elements as follows:

- The 3-input floating-point scalar versions of FMLA (by element) and FMLS (by element) take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.
- The 3-input floating-point FMADD, FMSUB, FNMADD, and FNMSUB instructions take output elements other than the lowest element from the <Ha>, <Sa>, or <Da> register.
- The 2-input floating-point scalar versions of FCMGE (register), FCMGT (register), FCMEQ (register), FACGE, FACGT, take output elements other than the lowest element from the <Hm>, <Sm>, or <Dm> register.
- The 2-input floating-point scalar versions of FMULX, FRECPS, FRSQRTS, FABD, FMUL (by element), FMUL (scalar), FDIV (scalar), FADD (scalar), FSUB (scalar), FMAX (scalar), FMIN (scalar), FMAXNM (scalar), FMINNM (scalar), FNMUL (scalar), take output elements other than the lowest element from the <Hn>, <Sn>, or <Dn> register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For 1-input floating-point scalar versions of the instructions FCVTNS (vector), FCVTMS (vector), FCVTAS (vector), FCVTPS (vector), SCVTF (vector, integer), UCVTF (vector, integer) FCVTZS (vector, integer), FCVTZU (vector, integer), FCVTNU (vector), FCVTMU (vector), FCVTAU (vector), FCVTPU (vector), SCVTF (vector, fixed-point), UCVTF (vector, fixed-point), FCVTZS (vector, fixed-point), FCVTZU (vector, fixed-point), SCVTF (scalar, integer), UCVTF (scalar, integer), SCVTF (scalar, fixed-point), UCVTF (scalar, fixed-point), BFCVT, FCVT, `FCVTXN`, FRECPE, FRECPX, FRSQRTE, FABS (scalar), FNEG (scalar), FSQRT (scalar), FRINTN (scalar), FRINTP (scalar), FRINTM (scalar), FRINTZ (scalar), FRINTA (scalar), FRINTI (scalar), FRINTX (scalar), FRINT32Z (scalar), FRINT32X (scalar), FRINT64Z (scalar), FRINT64X (scalar), take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.

C7.1.3 Data types

The A64 instruction set provides support for arithmetic, conversion, and bitwise operations on:

- Half-precision, single-precision, and double-precision floating-points.
- Signed and unsigned integers.
- Polynomials over {0, 1}.
- When FEAT_FCMA is implemented, complex numbers.

For all AArch64 floating-point operations, including SIMD operations, the rounding mode and exception trap handling are controlled by the FPCR.

Note

- AArch32 Advanced SIMD operations always use Arm standard floating-point arithmetic, regardless of the rounding mode specified by the AArch64 FPCR or the AArch32 FPSCR.
- In AArch64 state, floating-point multiply-add operations are always performed as fused operations, but AArch32 state provides both fused and chained multiply-add instructions.

In addition to operations that consume and produce values of the same width and type, the A64 instruction set supports SIMD and scalar operations that produce a wider or narrower vector result:

- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the A64 instruction set provides a second-part operation, for example `SHRN2`, that can pack the result of a second operation into the upper part of the same destination register.
- Where a SIMD operation widens a 64-bit vector to a 128-bit vector, the A64 instruction set provides a second-part operation, for example `SMLAL2`, that can extract the source from the upper 64 bits of the source registers.

All SIMD operations that could produce side-effects that are not limited to the destination SIMD and floating-point register, for example a potential update of FPSR.Q or FPSR.IDC, have a dedicated scalar variant to support the use of SIMD with loops requiring specialized head or tail handling, or both.

C7.1.4 Condition flags and related instructions

The A64 instruction set provides support for flag setting and conditional operations on the SIMD and floating-point register file:

- Floating-point `FCSEL` and `FCCMP` instructions are equivalent to the integer `CSEL` and `CCMP` instructions.
- Floating-point `FCMP`, `FCMPE`, `FCCMP`, and `FCCMP` instructions set the PSTATE.{N, Z, C, V} flags based on the result of the floating-point comparison.
- Floating-point `FJCVTZS` instruction sets the PSTATE.Z flag if the result of the conversion, when converted back to a double-precision floating-point number, gives precisely the same value as the original. Other PSTATE flags are cleared by this instruction.
- Floating-point and integer instructions provide a means of producing either a scalar or a vector mask based on a comparison in a SIMD and floating-point register, for example `FCMEQ`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

`FCMP` and `FCMPE` differ from the A32/T32 `VCMP` and `VCMPE` instructions, which use the dedicated FPSCR.NZCV field for the result. A64 instructions store the result of an `FCMP` or `FCMPE` operation in the PSTATE.{N, Z, C, V} field.
If FEAT_FlagM2 is implemented, base instructions XAFLAG and AXFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See Table C6-1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

This section lists every section in the Advanced SIMD and floating-point categories of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.1 ABS

Absolute value (vector)

This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1 0 0 0 0	0 1 0 1 1	1 0	Rn	Rd
		U				size		opcode			

Encoding

```
ABS  D<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	0 1 0 1 1	1 0	Rn	Rd
		U						opcode			

Encoding

```
ABS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    element = Abs(element);
    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.2 ADD (vector)

Add (vector)

This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	1	1	1	1	0	1	1	1	Rm		1	0	0	0	0	1	Rn		Rd	
		U						size					opcode									

Encoding

```
ADD  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	size		1	Rm		1	0	0	0	0	1	Rn		Rd	
		U											opcode									

Encoding

```
ADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = element1 + element2;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.3 ADDHN, ADDHN2

Add returning high narrow

This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see RADDHN.

The `ADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `ADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1	0	0	0 0	Rn	Rd
		U								o1				

Encoding

```
ADDHN{2}  <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
let round : boolean = FALSE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(2*datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:(2*esize)]);
    element2 = UInt(operand2[e*:(2*esize)]);
    sum = element1 + element2;
    sum = RShr(sum, esize, round);
    result[e*:esize] = sum[esize-1:0];
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.4 ADDP (scalar)

Add pair of elements (scalar)

This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	1	1	1	1	0	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						size							opcode								

Encoding

```
ADDP  D<d>, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 64;
let datasize : integer{} = 128;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
V{esize}(d) = IntReduce{esize, datasize}(ReduceOp_ADD, operand);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.5 ADDP (vector)

Add pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

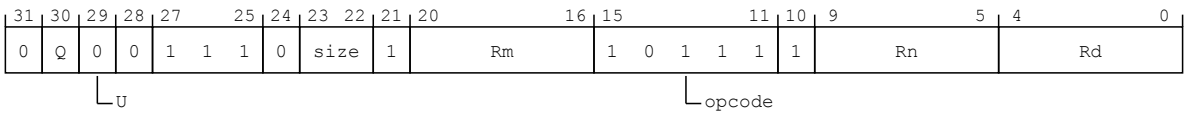

Encoding

```
ADDP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = concat[(2*e)*:esize];
    element2 = concat[((2*e)+1)*:esize];
    result[e*:esize] = element1 + element2;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.6 ADDV

Add across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 1 0 0 0	1 1 0 1 1	1 0	Rn	Rd
		U						opcode			

Encoding

```
ADDV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
V{esize}(d) = IntReduce{esize, datasize}(ReduceOp_ADD, operand);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.7 AESD

AES single round decryption.

Advanced SIMD

(FEAT_AES)

31–28	27–25	24	23–22	21–17	16–13	12	11–10	9–5	4–0
0 1 0 0	1 1 1	0	0 0 (size)	1 0 1 0 0	0 0 1 0	1 (D)	1 0	Rn	Rd

Encoding

```
AESD  <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(d);
let operand2 : bits(128) = V{}(n);
var result : bits(128) = operand1 XOR operand2;
result = AESInvShiftRows(result);
V{128}(d) = AESInvSubBytes(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.8 AESE

AES single round encryption.

Advanced SIMD

(FEAT_AES)

31–28	27–25	24	23–22	21–17	16–13	12	11–10	9–5	4–0
0 1 0 0	1 1 1	0	0 0 (size)	1 0 1 0 0	0 0 1 0	0 (D)	1 0	Rn	Rd

Encoding

```
AESE  <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(d);
let operand2 : bits(128) = V{}(n);
var result : bits(128) = operand1 XOR operand2;
result = AESShiftRows(result);
V{128}(d) = AESSubBytes(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.9 AESIMC

AES inverse mix columns.

Advanced SIMD

(FEAT_AES)

31–28	27–25	24	23–22	21–17	16–13	12	11–10	9–5	4–0
0 1 0 0	1 1 1	0	0 0 (size)	1 0 1 0 0	0 0 1 1	1 (D)	1 0	Rn	Rd

Encoding

```
AESIMC  <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand : bits(128) = V{}(n);
V{128}(d) = AESInvMixColumns(operand);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.10 AESMC

AES mix columns.

Advanced SIMD

(FEAT_AES)

31	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0 1 0 0		1 1 1		0	0	0	1 0 1 0 0		0 0 1 1		0	1	0	Rn		Rd	
					size						D						

Encoding

```
AESMC  <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand : bits(128) = V{}(n);
V{128}(d) = AESMixColumns(operand);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.11 AND (vector)

Bitwise AND (vector)

This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

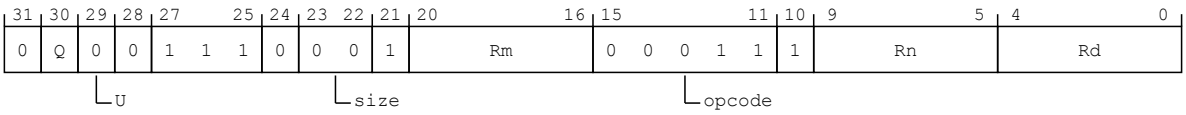

Encoding

```
AND  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
V{datasize}(d) = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.12 BCAX

Bit clear and exclusive-OR

This instruction performs a bitwise AND between the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive-OR between the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA3)

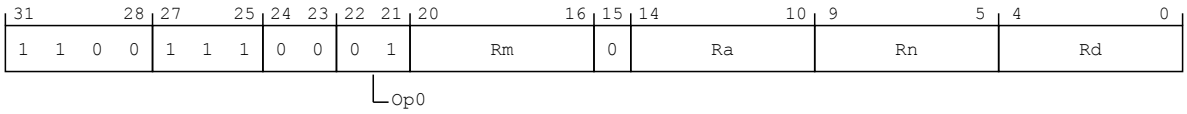

Encoding

```
BCAX  <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va>

Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(m);
let operand2 : bits(128) = V{}(n);
let operand3 : bits(128) = V{}(a);

V{128}(d) = operand2 XOR (operand1 AND NOT(operand3));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.13 BF1CVTL, BF1CVTL2, BF2CVTL, BF2CVTL2

8-bit floating-point convert to BFloat16 (vector)

This instruction converts each 8-bit floating-point element from the lower or upper half of the source vector to BFloat16 while downscaling the value, and places the results in the 16-bit elements of the destination vector. `BF1CVTL` and `BF2CVTL` convert the elements from the lower half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$, respectively. `BF1CVTL2` and `BF2CVTL2` convert the elements from the upper half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$, respectively.

The 8-bit floating-point encoding format for `BF1CVTL` and `BF1CVTL2` is selected by FPMR.F8S1. The 8-bit floating-point encoding format for `BF2CVTL` and `BF2CVTL2` is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8)

31	30	29	28				24	23	22	21				17	16				12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	x	1	0	0	0	0	1	0	1	1	1	1	0	Rn		Rd	
		U						size							opcode										

Encoding for the BF1CVTL{2} variant

Applies when `(size == 10)`

```
BF1CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Encoding for the BF2CVTL{2} variant

Applies when `(size == 11)`

```
BF2CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let part : integer = UInt(Q);
let elements : integer = 64 DIV 8;
let issrc2 : boolean = size == '11';
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(64) = Vpart{}(n, part);
var result : bits(128);

for e = 0 to elements-1 do
    result[e*:16] = FP8ConvertBF(operand[e*:8], issrc2, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.14 BFCVT

Single-precision convert to BFloat16 (scalar)

This instruction converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Single-precision to BFloat16

(FEAT_BF16)

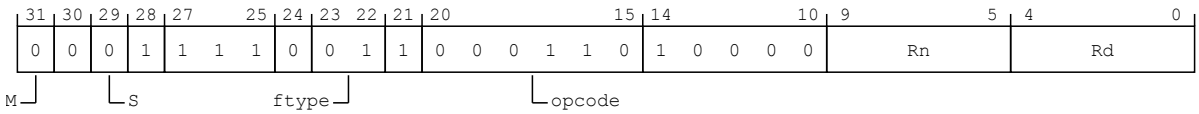

Encoding

```
BFCVT  <Hd>, <Sn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(32) = V{}(n);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

result[0+:16] = FPConvertBF(operand, FPCR());
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.15 BFCVTN, BFCVTN2

Single-precision convert to BFloat16 (vector)

This instruction reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The `BFCVTN` instruction writes the half-width results to the lower half of the 128-bit SIMD&FP destination vector and clears the upper half to zero. The `BFCVTN2` instruction writes the results to the upper half of the 128-bit SIMD&FP destination vector without affecting the lower half.

Vector single-precision to BFloat16

(FEAT_BF16)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1 0	1 0 0 0 0	1 0 1 1 0	1 0	Rn	Rd
		U				size		opcode			

Encoding

```
BFCVTN{2}  <Vd>.<Ta>, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let part : integer = UInt(Q);
let elements : integer = 64 DIV 16;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Ta>
0	4H
1	8H

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(128) = V{}(n);
var result : bits(64);

for e = 0 to elements-1 do
    result[e*:16] = FPConvertBF(operand[e*:32], FPCR());
end;

Vpart{64}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.16 BFDOT (by element)

BFloat16 dot product to single-precision (vector, by element)

This instruction delimits the source vectors into pairs of BFloat16 elements. The BFloat16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Advanced SIMD

(FEAT_BF16)

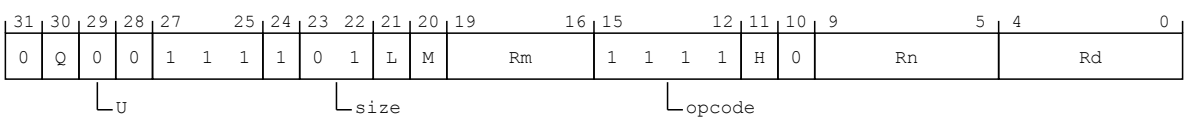

Encoding

```
BFDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(M::Rm);
let d : integer = UInt(Rd);
let i : integer = UInt(H::L);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the “H:L” fields.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * i + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * i + 1)*:16];

    let sum : bits(32) = operand3[e*:32];
    result[e*:32] = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.17 BFDOT (vector)

BFloat16 dot product to single-precision (vector)

This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Advanced SIMD

(FEAT_BF16)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0 1	0	Rm	1	1 1 1 1	1	Rn	Rd
		U				size				opcode			

Encoding

```
BFDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];

    let sum : bits(32) = operand3[e*:32];
    result[e*:32] = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.18 BFMLALB, BFMLALT (by element)

BFloat16 multiply-add to single-precision (by element)

This instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(FEAT_BF16)

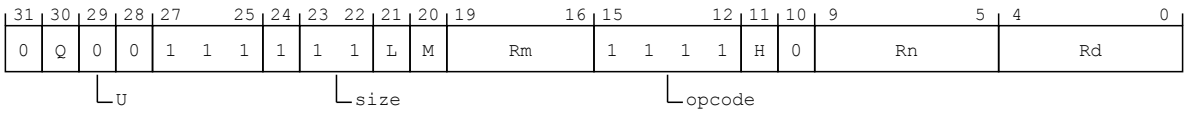

Encoding

```
BFMLAL<bt>  <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt('0'::Rm);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M);

let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

Assembler Symbols

<bt>

Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
var result : bits(128);
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);
let element2 : bits(16)  = operand2[index*:16];

for e = 0 to elements-1 do
    let element1 : bits(16)   = operand1[(2 * e + sel)*:16];
    let addend : bits(32)     = operand3[e*:32];
    result[e*:32] = BFMulAddH(addend, element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.19 BFMLALB, BFMLALT (vector)

BFloat16 multiply-add to single-precision (vector)

This instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(FEAT_BF16)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1 1	0	Rm	1	1 1 1 1	1	Rn	Rd
		U				size				opcode			

Encoding

```
BFMLAL<bt>  <Vd>.4S, <Vn>.8H, <Vm>.8H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

Assembler Symbols

<bt>

Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);
var result : bits(128);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[(2 * e + sel)*:16];
    let element2 : bits(16) = operand2[(2 * e + sel)*:16];
    let addend : bits(32)   = operand3[e*:32];
    result[e*:32] = BFMulAddH(addend, element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.20 BFMMLA (widening)

BFloat16 matrix multiply-accumulate to single-precision

This instruction multiplies the 2x4 matrix of BFloat16 values held in the first source vector by the 4x2 matrix of BFloat16 values in the second source vector.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Calculates the intermediate single-precision results as a sum of two unfused sum-of-products using two pairs of adjacent BFloat16 values from each of the source vectors, and then rounds the results before accumulation into the 2x2 matrix of single-precision results in the destination vector. This is equivalent to accumulating a 4-way dot product per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, this instruction:

- Calculates the intermediate single-precision results as a sum of two fused sum-of-products using two pairs of adjacent BFloat16 values from each of the source vectors, and then rounds the results before accumulation into the 2x2 matrix of single-precision results in the destination vector. This is equivalent to accumulating a 4-way dot product per destination element.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Advanced SIMD

(FEAT_BF16)

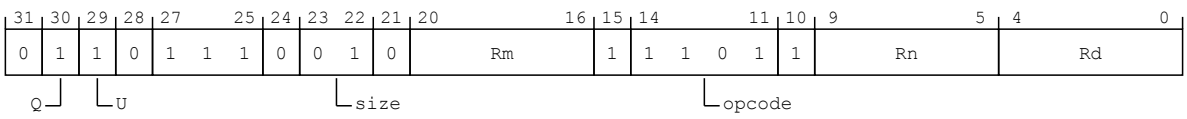

Encoding

```
BFMMLA  <Vd>.4S, <Vn>.8H, <Vm>.8H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_BF16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let op1 : bits(128) = V{}(n);
let op2 : bits(128) = V{}(m);
let acc : bits(128) = V{}(d);

V{128}(d) = BFMatMulAddH(acc, op1, op2, FPCR());
```

Operational Information

Arm expects that the BFMMLA instruction will deliver a peak BFloat16 multiply throughput that is at least as high as can be achieved using two BFDOT (vector) instructions, with a goal that it should have significantly higher throughput.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.21 BIC (vector, immediate)

Bitwise bit clear (vector, immediate)

This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Shifted immediate

(FEAT_AdvSIMD)

31	30	29	28									19	18	17	16	15			12	11	10	9	8	7	6	5	4	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd	
		op														cmode				o2								

Encoding for the 16-bit variant

Applies when `(cmode == 10x1)`

```
BIC  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit variant

Applies when `(cmode == 0xx1)`

```
BIC  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let rd : integer = UInt(Rd);
let datasize : integer{} = 64 << UInt(Q);
let imm64 : bits(64) = AdvSIMDExpandImm(op, cmode, a::b::c::d::e::f::g::h);
let imm : bits(datasize) = Replicate{}(imm64);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP register, encoded in the “Rd” field.

<T>

For the “16-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1	8H

For the “32-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8>

Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<amount>

For the “16-bit” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(rd);
V{datasize}(rd) = operand AND NOT(imm);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.22 BIC (vector, register)

Bitwise bit clear (vector, register)

This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

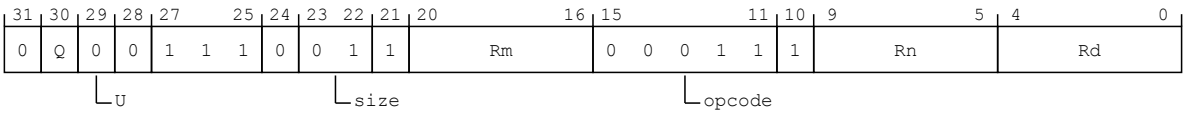

Encoding

```
BIC  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = NOT(V{datasize}(m));
V{datasize}(d) = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.23 BIF

Bitwise insert if false

This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

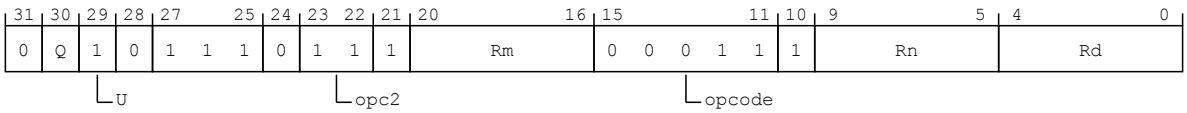

Encoding

```
BIF  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(datasize) = V{}(d);
let operand2 : bits(datasize) = NOT(V{datasize}(m));
let operand3 : bits(datasize) = V{}(n);
V{datasize}(d) = operand1 XOR ((operand1 XOR operand3) AND operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.24 BIT

Bitwise insert if true

This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

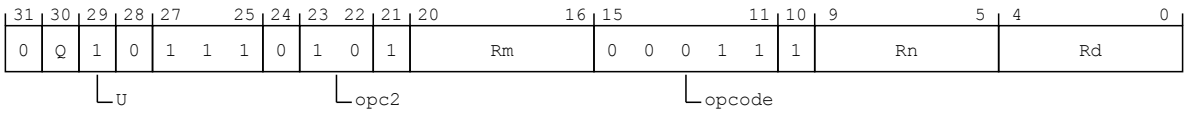

Encoding

```
BIT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(datasize) = V{}(d);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(n);
V{datasize}(d) = operand1 XOR ((operand1 XOR operand3) AND operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.25 BSL

Bitwise select

This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

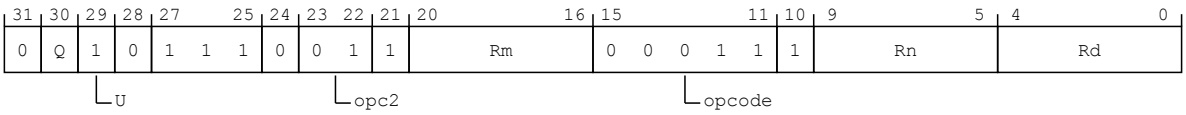

Encoding

```
BSL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(datasize) = V{}(m);
let operand2 : bits(datasize) = V{}(d);
let operand3 : bits(datasize) = V{}(n);
V{datasize}(d) = operand1 XOR ((operand1 XOR operand3) AND operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.26 CLS (vector)

Count leading sign bits (vector)

This instruction determines the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The result does not include the most significant bit itself.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	0 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
CLS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    let count : integer = CountLeadingSignBits(operand[e*:esize]);
    result[e*:esize] = count[esize-1:0];
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.27 CLZ (vector)

Count leading zero bits (vector)

This instruction determines the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1 0 0 0 0	0 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
CLZ  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    let count : integer = CountLeadingZeroBits(operand[e*:esize]);
    result[e*:esize] = count[esize-1:0];
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.28 CMEQ (register)

Compare bitwise equal (vector)

This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	0	1	1	1	Rm	1	0	0	0	1	1	Rn	Rd
		U						size				opcode							

Encoding

```
CMEQ  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	0	size		1	Rm	1	0	0	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
CMEQ  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = if element1 == element2 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.29 CMEQ (zero)

Compare bitwise equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	1	1	0	Rn					Rd				
		U						size											op												

Encoding

```
CMEQ  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	1	0	0	1	1	0	Rn					Rd				
		U																	op												

Encoding

```
CMEQ  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    result[e*:esize] = if element == 0 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.30 CMGE (register)

Compare signed greater than or equal (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1	Rm	0 0 1 1	1	1	Rn	Rd
		U				size				eq			

Encoding

```
CMGE  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 1 1	1	1	Rn	Rd
		U								eq			

Encoding

```
CMGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var element2 : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    result[e*:esize] = if element1 >= element2 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.31 CMGE (zero)

Compare signed greater than or equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	1	0	Rn					Rd				
		U						size											op												

Encoding

```
CMGE  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	1	0	0	0	1	0	Rn					Rd				
		U																	op												

Encoding

```
CMGE  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    result[e*:esize] = if element >= 0 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.32 CMGT (register)

Compare signed greater than (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1	Rm	0 0 1 1	0	1	Rn	Rd
		U				size				eq			

Encoding

```
CMGT  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 1 1	0	1	Rn	Rd
		U								eq			

Encoding

```
CMGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var element2 : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    result[e*:esize] = if element1 > element2 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.33 CMGT (zero)

Compare signed greater than zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	1	0	Rn					Rd				
		U						size											op												

Encoding

```
CMGT  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	1	0	0	0	1	0	Rn					Rd				
		U																	op												

Encoding

```
CMGT  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    result[e*:esize] = if element > 0 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.34 CMHI (register)

Compare unsigned higher (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1 1	1	Rm	0 0 1 1	0	1	Rn	Rd
		U				size				eq			

Encoding

```
CMHI  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 1 1	0	1	Rn	Rd
		U								eq			

Encoding

```
CMHI  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var element2 : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    result[e*:esize] = if element1 > element2 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.35 CMHS (register)

Compare unsigned higher or same (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1 1	1	Rm	0 0 1 1	1	1	Rn	Rd
		U				size				eq			

Encoding

```
CMHS  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 1 1	1	1	Rn	Rd
		U								eq			

Encoding

```
CMHS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var element2 : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    result[e*:esize] = if element1 >= element2 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.36 CMLE (zero)

Compare signed less than or equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	1	1	0	Rn	Rd
		U						size											op				

Encoding

```
CMLE  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	1	0	0	1	1	0	Rn	Rd
		U																	op				

Encoding

```
CMLE  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    result[e*:esize] = if element <= 0 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.37 CMLT (zero)

Compare signed less than zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	1	0	1	0	Rn					Rd				
		U						size							opcode																

Encoding

```
CMLT  D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	1	0	1	0	1	0	Rn					Rd				
		U													opcode																

Encoding

```
CMLT  <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    result[e*:esize] = if element < 0 then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.38 CMTST

Compare bitwise test bits nonzero (vector)

This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	0	1 1	1	Rm	1 0 0 0 1	1	Rn	Rd
		U				size			opcode			

Encoding

```
CMTST  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0 0 0 1	1	Rn	Rd
		U							opcode			

Encoding

```
CMTST  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = if !IsZero(element1 AND element2) then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.39 CNT

Population count per byte

This instruction determines the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

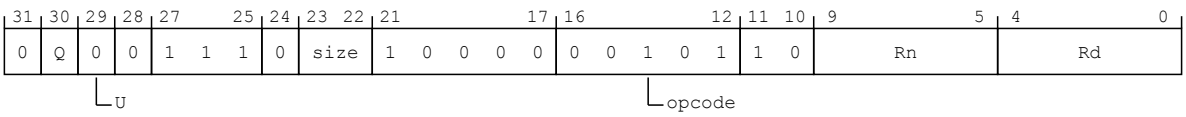

Encoding

```
CNT  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 8;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    let count : integer = BitCount(operand[e*:esize]);
    result[e*:esize] = count[esize-1:0];
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.40 DUP (element)

Duplicate vector element to vector or scalar

This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	0	0	0	0	imm5	0	0	0	0	0	1	Rn	Rd
		op														imm4			

Encoding

```
DUP  <V><d>, <Vn>.<T>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[3:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let index : integer = UInt(imm5[4:size+1]);
let idxdsize : integer{} = 64 << UInt(imm5[4]);
let esize : integer{} = 8 << size;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	0	0	0	imm5	0	0	0	0	0	1	Rn	Rd
		op														imm4			

Encoding

```
DUP  <Vd>.<T>, <Vn>.<Ts>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x1000' && Q == '0' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[3:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let index : integer = UInt(imm5[4:size+1]);
let idxdsize : integer{} = 64 << UInt(imm5[4]);
let esize : integer{} = 8 << size;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

For the “Scalar” variant: is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the “Vector” variant: is an arrangement specifier, encoded in “imm5:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<index>

Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (scalar)	Unconditionally

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(idxdsize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

element = operand[index*:esize];
for e = 0 to elements-1 do
    result[e*:esize] = element;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.41 DUP (general)

Duplicate general-purpose register to vector

This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

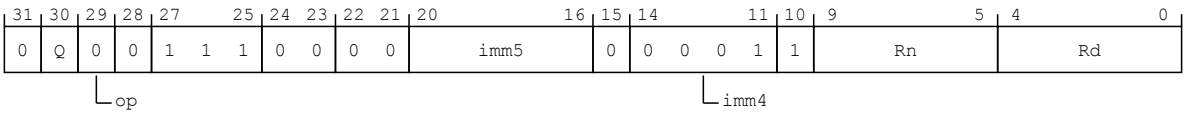

Encoding

```
DUP  <Vd>.<T>, <R><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x1000' && Q == '0' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[3:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
// imm5<4:size+1> is IGNORED
let esize : integer{} = 8 << size;
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	Q	<T>
x1000	0	RESERVED
x1000	1	2D

<R>

Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in “imm5” are ignored but should be set to zero by an assembler.

<n>

Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let element : bits(esize) = X{}(n);
let elements : integer = datasize DIV esize;
var result : bits(datasize);

for e = 0 to elements-1 do
    result[e*:esize] = element;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.42 EOR3

Three-way exclusive-OR

This instruction performs a three-way exclusive-OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA3)

31–28	27–25	24–23	22–21	20–16	15	14–10	9–5	4–0
1 1 0 0	1 1 1	0 0	0 0 (Op0)	Rm	0	Ra	Rn	Rd

Encoding

```
EOR3  <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va>

Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(m);
let operand2 : bits(128) = V{}(n);
let operand3 : bits(128) = V{}(a);
V{128}(d) = operand2 XOR operand1 XOR operand3;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.43 EOR (vector)

Bitwise exclusive-OR (vector)

This instruction performs a bitwise exclusive-OR between the first source SIMD&FP register and the second source SIMD&FP register, and places the result in the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

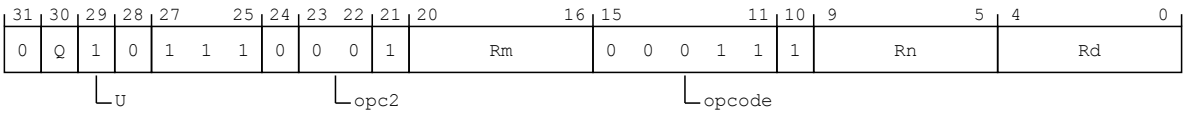

Encoding

```
EOR  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(datasize) = V{}(m);
let operand2 : bits(datasize) = Zeros{};
let operand3 : bits(datasize) = Ones{};
let operand4 : bits(datasize) = V{}(n);
V{datasize}(d) = operand1 XOR ((operand2 XOR operand4) AND operand3);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.44 EXT

Extract vector from pair of vectors

This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

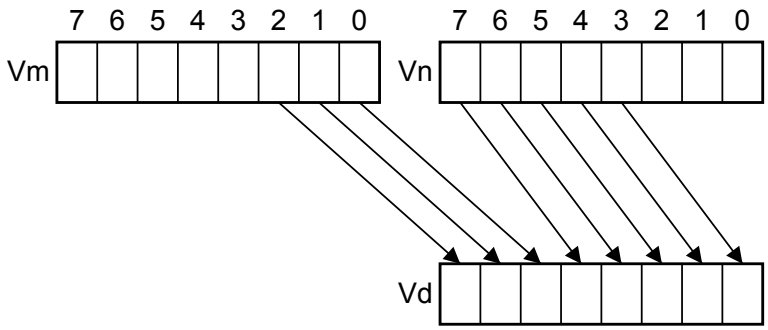

Figure C7-1 EXT doubleword operation for Q = 0 and imm4<2:0> = 3

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1 0	1 1 1	0	0 0	0	Rm	0	imm4	0	Rn	Rd
					op2							

Encoding

```
EXT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if Q == '0' && imm4[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
let position : integer{} = 8 * UInt(imm4);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<index>

Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

Q	imm4<3>	<index>
0	0	UInt(imm4<2:0>)
0	1	RESERVED
1	x	UInt(imm4)

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let hi : bits(datasize) = V{}(m);
let lo : bits(datasize) = V{}(n);
let concat : bits(datasize*2) = hi :: lo;

V{datasize}(d) = concat[(position+datasize)-1:position];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.45 F1CVTL, F1CVTL2, F2CVTL, F2CVTL2

8-bit floating-point convert to half-precision (vector)

This instruction converts each 8-bit floating-point element from the lower or upper half of the source vector to half-precision while downscaling the value, and places the results in the 16-bit elements of the destination vector. `F1CVTL` and `F2CVTL` convert the elements from the lower half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$, respectively. `F1CVTL2` and `F2CVTL2` convert the elements from the upper half of the source vector while scaling the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ and $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$, respectively.

The 8-bit floating-point encoding format for `F1CVTL` and `F1CVTL2` is selected by FPMR.F8S1. The 8-bit floating-point encoding format for `F2CVTL` and `F2CVTL2` is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8)

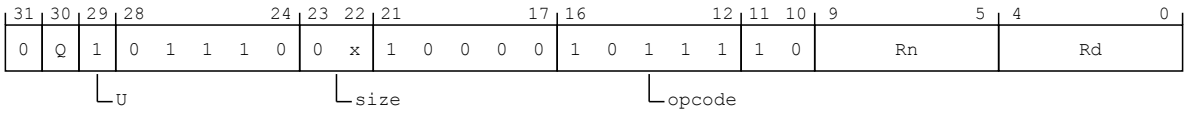

Encoding for the F1CVTL{2} variant

Applies when `(size == 00)`

```
F1CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Encoding for the F2CVTL{2} variant

Applies when `(size == 01)`

```
F2CVTL{2}  <Vd>.8H, <Vn>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let part : integer = UInt(Q);
let elements : integer = 64 DIV 8;
let issrc2 : boolean = size == '01';
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(64) = Vpart{}(n, part);
var result : bits(128);

for e = 0 to elements-1 do
    result[e*:16] = FP8ConvertFP(operand[e*:8], issrc2, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.46 FABD

Floating-point absolute difference (vector)

This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13–11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	1	0	Rm	0	0	0 1 0	1	Rn	Rd
		U				a						opcode			

Encoding

```
FABD  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	sz	1	Rm	1 1 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
FABD  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 0	Rm	0 0	0 1 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FABD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1	Rm	1 1 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
FABD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(n) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    let diff : bits(esize) = FPSub{}(element1, element2, FPCR());
    result[e*:esize] = FPAbs{esize}(diff, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.47 FABS (vector)

Floating-point absolute value (vector)

This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn	Rd
		U						a							opcode								

Encoding

```
FABS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	1	1	0	Rn	Rd
		U													opcode								

Encoding

```
FABS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPAbs{esize}(element, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.48 FABS (scalar)

Floating-point absolute value (scalar)

This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

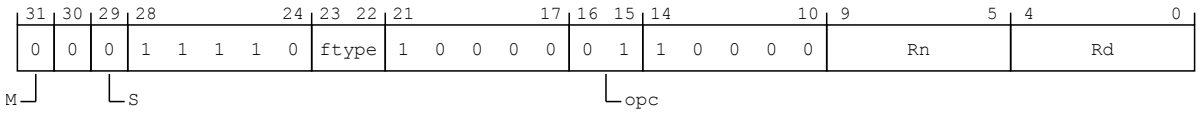

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FABS  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FABS  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FABS  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPAbs{esize}(operand, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.49 FACGE

Floating-point absolute compare greater than or equal (vector)

This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	0	1	0	Rm	0	0	1	0	1	1	Rn	Rd
		U				E								ac			

Encoding

```
FACGE  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	0	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGE  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 0	Rm	0 0	1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
var test_passed : boolean;
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(m) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    element1 = FPAbs{esize}(element1, FPCR());
    element2 = FPAbs{esize}(element2, FPCR());
    test_passed = FPCompareGE{esize}(element1, element2, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.50 FACGT

Floating-point absolute compare greater than (vector)

This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	1	0	Rm	0	0	1	0	1	1	Rn	Rd
		U				E								ac			

Encoding

```
FACGT  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGT  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 0	Rm	0 0	1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1	Rm	1 1 1 0	1	1	Rn	Rd
		U				E					ac			

Encoding

```
FACGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
var test_passed : boolean;
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(m) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    element1 = FPAbs{esize}(element1, FPCR());
    element2 = FPAbs{esize}(element2, FPCR());
    test_passed = FPCompareGT{esize}(element1, element2, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.51 FADD (vector)

Floating-point add (vector)

This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 0	Rm	0 0	0 1 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1	Rm	1 1 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
FADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPAdd{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.52 FADD (scalar)

Floating-point add (scalar)

This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

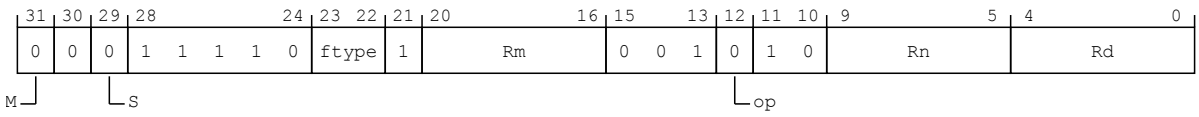

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FADD  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FADD  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FADD  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPAdd{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.53 FADDP (scalar)

Floating-point add pair of elements (scalar)

This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U							sz						opcode								

Encoding

```
FADDP  H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 32;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U													opcode								

Encoding

```
FADDP  <V><d>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize * 2;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<T>

Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand :  bits(datasize) = V{}(n);
let element1 :  bits(esize) = operand[0*:esize];
let element2 : bits(esize) = operand[1*:esize];
V{esize}(d) = FPAdd{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.54 FADDP (vector)

Floating-point add pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22–21	20–16	15–14	13–11	10	9–5	4–0
0	Q	1	0	1 1 1	0	0	1 0	Rm	0 0	0 1 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FADDP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–11	10	9–5	4–0
0	Q	1	0	1 1 1	0	0	sz	1	Rm	1 1 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
FADDP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = concat[(2*e)*:esize];
    element2 = concat[((2*e)+1)*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:esize] = FPAdd{esize}(element1, element2, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.55 FAMAX

Floating-point absolute maximum

This instruction determines the maximum absolute value from floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and places the results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

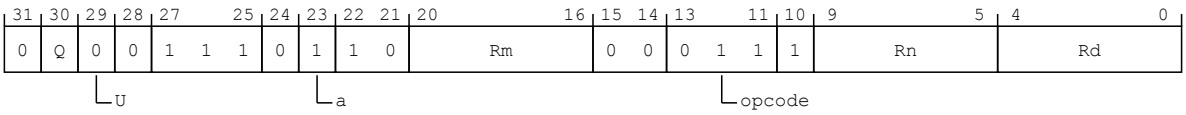

Encoding

```
FAMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

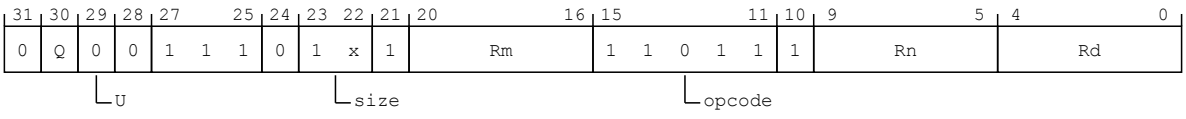

Encoding

```
FAMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “size<0>:Q”:

size<0>	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPAbsMax{esize}(op1, op2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.56 FAMIN

Floating-point absolute minimum

This instruction determines the minimum absolute value from floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and places the results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

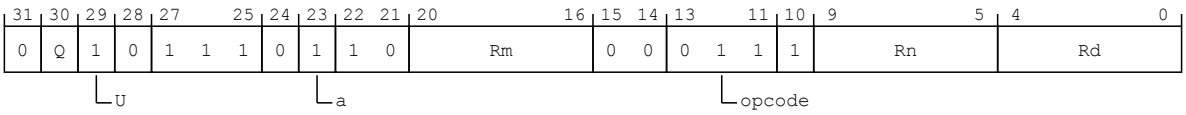

Encoding

```
FAMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD && FEAT_FAMINMAX)

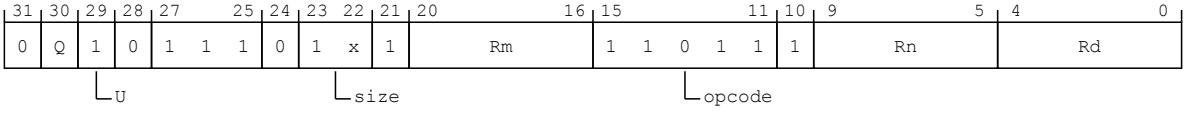

Encoding

```
FAMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “size<0>:Q”:

size<0>	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPAbsMin{esize}(op1, op2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.57 FCADD

Floating-point complex add

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	0	Rm	1	1 1	rot	0	1	Rn	Rd
		U													

Encoding

```
FCADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<rotate>

Is the rotation, encoded in “rot”:

rot	<rotate>
0	90
1	270

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element3 : bits(esize);

for e = 0 to (elements DIV 2)-1 do
    case rot of
        when '0' =>
            element1 = FPNeg{esize}(operand2[(e*2+1)*:esize], FPCR());
            element3 = operand2[(e*2)*:esize];
        when '1' =>
            element1 = operand2[(e*2+1)*:esize];
            element3 = FPNeg{esize}(operand2[(e*2)*:esize], FPCR());
    end;
    result[(e*2)*:esize] = FPAdd{esize}(operand1[(e*2)*:esize], element1, FPCR());
    result[(e*2+1)*:esize] = FPAdd{esize}(operand1[(e*2+1)*:esize], element3, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.58 FCCMP

Floating-point conditional quiet compare (scalar)

This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass, then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

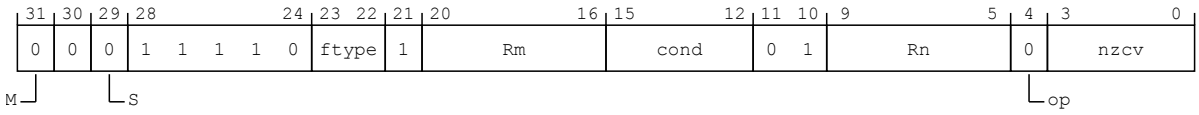

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FCCMP  <Hn>, <Hm>, #<nzcv>, <cond>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FCCMP  <Sn>, <Sm>, #<nzcv>, <cond>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FCCMP  <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 8 << UInt(ftype XOR '10');
let condition : bits(4) = cond;
var flags : bits(4) = nzcv;
```

Assembler Symbols

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();

let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

if ConditionHolds(condition) then
    let signal_all_nans : boolean = FALSE;
    flags = FPCompare{datasize}(operand1, operand2, signal_all_nans, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

PSTATE.[N,Z,C,V] = flags;
```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.59 FCCMPE

Floating-point conditional signaling compare (scalar)

This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass, then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

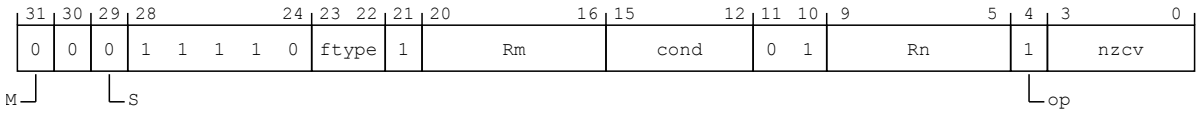

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FCCMPE  <Hn>, <Hm>, #<nzcv>, <cond>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FCCMPE  <Sn>, <Sm>, #<nzcv>, <cond>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FCCMPE  <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 8 << UInt(ftype XOR '10');
let condition : bits(4) = cond;
var flags : bits(4) = nzcv;
```

Assembler Symbols

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<nzcv>

Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();

let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

if ConditionHolds(condition) then
    let signal_all_nans : boolean = TRUE;
    flags = FPCompare{datasize}(operand1, operand2, signal_all_nans, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

PSTATE.[N,Z,C,V] = flags;
```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.60 FCMEQ (register)

Floating-point compare equal (vector)

This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

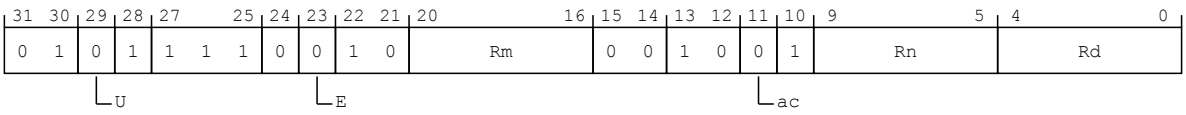

Encoding

```
FCMEQ  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

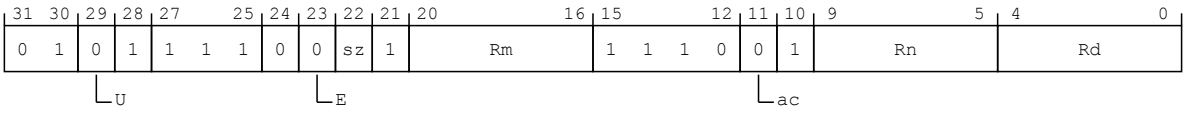

Encoding

```
FCMEQ  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 0	Rm	0 0	1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMEQ  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1	Rm	1 1 1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMEQ  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
var test_passed : boolean;
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(m) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    test_passed = FPCompareEQ{esize}(element1, element2, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.61 FCMEQ (zero)

Floating-point compare equal to zero (vector)

This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22–19	18–17	16–13	12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	1	1 1 1 1	0 0	0 1 1 0	1	1 0	Rn	Rd
		U				a				op			

Encoding

```
FCMEQ  <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21–17	16–13	12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	1	sz	1 0 0 0 0	0 1 1 0	1	1 0	Rn	Rd
		U								op			

Encoding

```
FCMEQ  <V><d>, <V><n>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn	Rd
		U						a											op				

Encoding

```
FCMEQ  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U																	op				

Encoding

```
FCMEQ  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let zero : bits(esize) = FPZero{}('0');
var element : bits(esize);
var test_passed : boolean;

for e = 0 to elements-1 do
    element = operand[e*:esize];
    test_passed = FPCompareEQ{esize}(element, zero, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.62 FCMGE (register)

Floating-point compare greater than or equal (vector)

This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	0	1	0	Rm	0	0	1	0	0	1	Rn	Rd
		U				E								ac			

Encoding

```
FCMGE  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	0	sz	1	Rm	1 1 1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMGE  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 0	Rm	0 0	1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1	Rm	1 1 1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMGE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
var test_passed : boolean;
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(m) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    test_passed = FPCompareGE{esize}(element1, element2, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.63 FCMGE (zero)

Floating-point compare greater than or equal to zero (vector)

This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U						a											op						

Encoding

```
FCMGE  <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn		Rd	
		U																	op						

Encoding

```
FCMGE  <V><d>, <V><n>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	25	24	23	22	19	18	17	16	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	0	Rn		Rd	
		U						a						op						

Encoding

```
FCMGE  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	1	sz	1 0 0 0 0	0	1 1 0	0	1	0	Rn		Rd	
		U											op						

Encoding

```
FCMGE  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let zero : bits(esize) = FPZero{}('0');
var element : bits(esize);
var test_passed : boolean;

for e = 0 to elements-1 do
    element = operand[e*:esize];
    test_passed = FPCompareGE{esize}(element, zero, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.64 FCMGT (register)

Floating-point compare greater than (vector)

This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	1	0	Rm	0	0	1	0	0	1	Rn	Rd
		U				E								ac			

Encoding

```
FCMGT  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	0	1	sz	1	Rm	1 1 1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMGT  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 0	Rm	0 0	1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1	Rm	1 1 1 0	0	1	Rn	Rd
		U				E					ac			

Encoding

```
FCMGT  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
var test_passed : boolean;
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(m) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    test_passed = FPCompareGT{esize}(element1, element2, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.65 FCMGT (zero)

Floating-point compare greater than zero (vector)

This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U						a											op												

Encoding

```
FCMGT  <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Encoding

```
FCMGT  <V><d>, <V><n>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn	Rd
		U						a											op				

Encoding

```
FCMGT  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn	Rd
		U																	op				

Encoding

```
FCMGT  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let zero : bits(esize) = FPZero{}('0');
var element : bits(esize);
var test_passed : boolean;

for e = 0 to elements-1 do
    element = operand[e*:esize];
    test_passed = FPCompareGT{esize}(element, zero, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.66 FCMLA (by element)

Floating-point complex multiply accumulate (by element)

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14 13	12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	size	L	M	Rm	0	rot	1	H	0	Rn	Rd
		U														

Encoding

```
FCMLA  <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) && size == '10' then EndOfDecode(Decode_UNDEF); end;
if size == '10' && (L == '1' || Q == '0') then EndOfDecode(Decode_UNDEF); end;
if size == '01' && H == '1' && Q == '0' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(M::Rm);
var index : integer;
if size == '01' then index = UInt(H::L); end;
if size == '10' then index = UInt(H); end;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L”:

size	<index>
00	RESERVED
01	UInt(H:L)
10	UInt(H)
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<rotate>

Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to (elements DIV 2)-1 do
    var element1 : bits(esize);
    var element2 : bits(esize);
    var element3 : bits(esize);
    var element4 : bits(esize);
    case rot of
        when '00' =>
            element1 = operand2[(index*2)*:esize];
            element2 = operand1[(e*2)*:esize];
            element3 = operand2[(index*2+1)*:esize];
            element4 = operand1[(e*2)*:esize];
        when '01' =>
            element1 = FPNeg{esize}(operand2[(index*2+1)*:esize], FPCR());
            element2 = operand1[(e*2+1)*:esize];
            element3 = operand2[(index*2)*:esize];
            element4 = operand1[(e*2+1)*:esize];
        when '10' =>
            element1 = FPNeg{esize}(operand2[(index*2)*:esize], FPCR());
            element2 = operand1[(e*2)*:esize];
            element3 = FPNeg{esize}(operand2[(index*2+1)*:esize], FPCR());
            element4 = operand1[(e*2)*:esize];
        when '11' =>
            element1 = operand2[(index*2+1)*:esize];
            element2 = operand1[(e*2+1)*:esize];
            element3 = FPNeg{esize}(operand2[(index*2)*:esize], FPCR());
            element4 = operand1[(e*2+1)*:esize];
    end;

    result[(e*2)*:esize] = FPMulAdd{esize}(operand3[(e*2)*:esize], element2, element1, FPCR());
    result[(e*2+1)*:esize] = FPMulAdd{esize}(operand3[(e*2+1)*:esize], element4, element3, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.67 FCMLA

Floating-point complex multiply accumulate

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 13	12 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	0	Rm	1	1 0	rot	1	Rn	Rd
		U												

Encoding

```
FCMLA  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) && size == '01' then EndOfDecode(Decode_UNDEF); end;
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<rotate>

Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
var element3 : bits(esize);
var element4 : bits(esize);

for e = 0 to (elements DIV 2)-1 do
    case rot of
        when '00' =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            element1 = operand2[(e*2)*:esize];
            element2 = operand1[(e*2)*:esize];
            element3 = operand2[(e*2+1)*:esize];
            element4 = operand1[(e*2)*:esize];
        when '01' =>
            element1 = FPNeg{esize}(operand2[(e*2+1)*:esize], FPCR());
            element2 = operand1[(e*2+1)*:esize];
            element3 = operand2[(e*2)*:esize];
            element4 = operand1[(e*2+1)*:esize];
        when '10' =>
            element1 = FPNeg{esize}(operand2[(e*2)*:esize], FPCR());
            element2 = operand1[(e*2)*:esize];
            element3 = FPNeg{esize}(operand2[(e*2+1)*:esize], FPCR());
            element4 = operand1[(e*2)*:esize];
        when '11' =>
            element1 = operand2[(e*2+1)*:esize];
            element2 = operand1[(e*2+1)*:esize];
            element3 = FPNeg{esize}(operand2[(e*2)*:esize], FPCR());
            element4 = operand1[(e*2+1)*:esize];
    end;

    result[(e*2)*:esize] = FPMulAdd{esize}(operand3[(e*2)*:esize], element2, element1, FPCR());
    result[(e*2+1)*:esize] = FPMulAdd{esize}(operand3[(e*2+1)*:esize], element4, element3, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.68 FCMLE (zero)

Floating-point compare less than or equal to zero (vector)

This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn					Rd				
		U						a											op												

Encoding

```
FCMLE  <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn					Rd				
		U																	op												

Encoding

```
FCMLE  <V><d>, <V><n>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn	Rd
		U						a											op				

Encoding

```
FCMLE  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U																	op				

Encoding

```
FCMLE  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let zero : bits(esize) = FPZero{}('0');
var element : bits(esize);
var test_passed : boolean;

for e = 0 to elements-1 do
    element = operand[e*:esize];
    test_passed = FPCompareGE{esize}(zero, element, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.69 FCMLT (zero)

Floating-point compare less than zero (vector)

This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn					Rd				
		U						a							opcode																

Encoding

```
FCMLT  <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn					Rd				
		U													opcode																

Encoding

```
FCMLT  <V><d>, <V><n>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn	Rd
		U						a							opcode								

Encoding

```
FCMLT  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn	Rd
		U													opcode								

Encoding

```
FCMLT  <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let zero : bits(esize) = FPZero{}('0');
var element : bits(esize);
var test_passed : boolean;

for e = 0 to elements-1 do
    element = operand[e*:esize];
    test_passed = FPCompareGT{esize}(zero, element, FPCR());
    result[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.70 FCMP

Floating-point quiet compare (scalar)

This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21	20				16	15	14	13			10	9				5	4	3	2			0
0	0	0	1	1	1	1	0	ftype		1	Rm					0	0	1	0	0	0	Rn					0	x	0	0	0	0
M		S														op											opc					

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11 && opc == 00)`

```
FCMP  <Hn>, <Hm>
```

Encoding for the Half-precision, zero variant

(FEAT_FP16) Applies when `(ftype == 11 && Rm == (00000) && opc == 01)`

```
FCMP  <Hn>, #0.0
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00 && opc == 00)`

```
FCMP  <Sn>, <Sm>
```

Encoding for the Single-precision, zero variant

(FEAT_FP) Applies when `(ftype == 00 && Rm == (00000) && opc == 01)`

```
FCMP  <Sn>, #0.0
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01 && opc == 00)`

```
FCMP  <Dn>, <Dm>
```

Encoding for the Double-precision, zero variant

(FEAT_FP) Applies when `(ftype == 01 && Rm == (00000) && opc == 01)`

```
FCMP  <Dn>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let n : integer = UInt(Rn);
let m : integer = UInt(Rm);   // ignored when opc<0> == '1'

let datasize : integer{} = 8 << UInt(ftype XOR '10');
let signal_all_nans : boolean = FALSE;
let cmp_with_zero : boolean = (opc[0] == '1');
```

Assembler Symbols

<Hn>

For the “Half-precision” variant: is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Half-precision, zero” variant: is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Sn>

For the “Single-precision” variant: is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Single-precision, zero” variant: is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dn>

For the “Double-precision” variant: is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Double-precision, zero” variant: is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();

let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = if cmp_with_zero then FPZero{datasize}('0') else V{datasize}(m);

PSTATE.[N,Z,C,V] = FPCompare{datasize}(operand1, operand2, signal_all_nans, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.71 FCMPE

Floating-point signaling compare (scalar)

This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21	20				16	15	14	13			10	9				5	4	3	2			0
0	0	0	1	1	1	1	0	ftype		1	Rm					0	0	1	0	0	0	Rn					1	x	0	0	0
M		S														op											opc					

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11 && opc == 10)`

```
FCMPE  <Hn>, <Hm>
```

Encoding for the Half-precision, zero variant

(FEAT_FP16) Applies when `(ftype == 11 && Rm == (00000) && opc == 11)`

```
FCMPE  <Hn>, #0.0
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00 && opc == 10)`

```
FCMPE  <Sn>, <Sm>
```

Encoding for the Single-precision, zero variant

(FEAT_FP) Applies when `(ftype == 00 && Rm == (00000) && opc == 11)`

```
FCMPE  <Sn>, #0.0
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01 && opc == 10)`

```
FCMPE  <Dn>, <Dm>
```

Encoding for the Double-precision, zero variant

(FEAT_FP) Applies when `(ftype == 01 && Rm == (00000) && opc == 11)`

```
FCMPE  <Dn>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let n : integer = UInt(Rn);
let m : integer = UInt(Rm);   // ignored when opc<0> == '1'

let datasize : integer{} = 8 << UInt(ftype XOR '10');
let signal_all_nans : boolean = TRUE;
let cmp_with_zero : boolean = (opc[0] == '1');
```

Assembler Symbols

<Hn>

For the “Half-precision” variant: is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Half-precision, zero” variant: is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Sn>

For the “Single-precision” variant: is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Single-precision, zero” variant: is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dn>

For the “Double-precision” variant: is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Double-precision, zero” variant: is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();

let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = if cmp_with_zero then FPZero{datasize}('0') else V{datasize}(m);

PSTATE.[N,Z,C,V] = FPCompare{datasize}(operand1, operand2, signal_all_nans, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.72 FCSEL

Floating-point conditional select (scalar)

This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20	16	15	12	11 10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1	Rm		cond		1 1	Rn		Rd	

M S

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FCSEL  <Hd>, <Hn>, <Hm>, <cond>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FCSEL  <Sd>, <Sn>, <Sm>, <cond>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FCSEL  <Dd>, <Dn>, <Dm>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let datasize : integer{} = 8 << UInt(ftype XOR '10');
let condition : bits(4) = cond;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<cond>

Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
var result : bits(datasize);

let condition_holds : boolean = ConditionHolds(condition);
result = if condition_holds then V{datasize}(n) else V{datasize}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.73 FCVT

Floating-point convert precision (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FP)

31	30	29	28 … 24	23 22	21 … 17	16 15	14 … 10	9 … 5	4 … 0
0	0	0	1 1 1 1 0	ftype	1 0 0 0 1	opc	1 0 0 0 0	Rn	Rd
M		S							

Encoding for the Half-precision to single-precision variant

Applies when `(ftype == 11 && opc == 00)`

```
FCVT  <Sd>, <Hn>
```

Encoding for the Half-precision to double-precision variant

Applies when `(ftype == 11 && opc == 01)`

```
FCVT  <Dd>, <Hn>
```

Encoding for the Single-precision to half-precision variant

Applies when `(ftype == 00 && opc == 11)`

```
FCVT  <Hd>, <Sn>
```

Encoding for the Single-precision to double-precision variant

Applies when `(ftype == 00 && opc == 01)`

```
FCVT  <Dd>, <Sn>
```

Encoding for the Double-precision to half-precision variant

Applies when `(ftype == 01 && opc == 11)`

```
FCVT  <Hd>, <Dn>
```

Encoding for the Double-precision to single-precision variant

Applies when `(ftype == 01 && opc == 00)`

```
FCVT  <Sd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == opc || ftype == '10' || opc == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let srcsize : integer{} = 8 << UInt(ftype XOR '10');
let dstsize : integer{} = 8 << UInt(opc XOR '10');
```

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(srcsize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:dstsize] = FPConvert{dstsize, srcsize}(operand, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.74 FCVTAS (vector)

Floating-point convert to signed integer, rounding to nearest with ties to away (vector)

This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U						a							opcode																

Encoding

```
FCVTAS  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = FALSE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U													opcode																

Encoding

```
FCVTAS  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = FALSE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn	Rd
		U						a									opcode						

Encoding

```
FCVTAS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = FALSE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn	Rd
		U															opcode						

Encoding

```
FCVTAS  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.75 FCVTAS (scalar)

Floating-point convert to signed integer, rounding to nearest with ties to away (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	0 0	1	0 0	0 0 0 0 0	0	Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTAS  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTAS  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTAS  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTAS  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTAS  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTAS  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean    = FALSE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.76 FCVTAS (scalar SIMD&FP)

Floating-point convert to signed integer, rounding to nearest with ties to away (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 1	0 1	0	0 0 0 0 0	0	Rn		Rd	
		S				rmode		opcode							

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTAS  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTAS  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTAS  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTAS  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, FALSE, FPCR(), FPRounding_TIEAWAY);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.77 FCVTAU (vector)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector)

This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

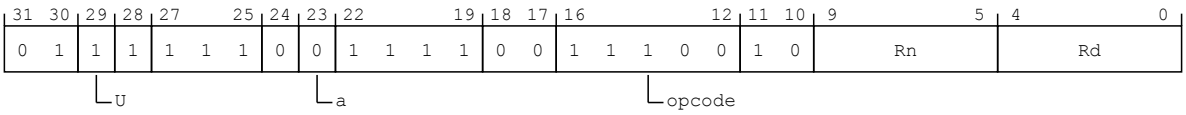

Encoding

```
FCVTAU  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = TRUE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

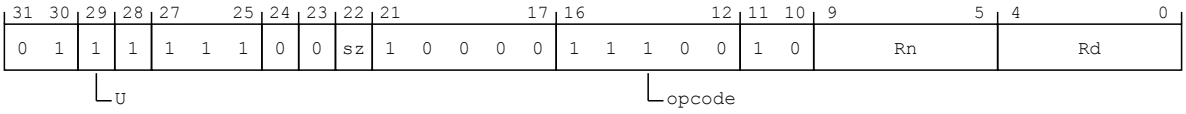

Encoding

```
FCVTAU  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = TRUE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 1 1 1	0 0	1 1 1 0 0	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
FCVTAU  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = TRUE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 1 0 0	1 0	Rn	Rd
		U							opcode			

Encoding

```
FCVTAU  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.78 FCVTAU (scalar)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	0 0	1	0 1	0 0 0 0 0	0	Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTAU  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTAU  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTAU  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTAU  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTAU  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTAU  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_TIEAWAY;
let unsigned : boolean    = TRUE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.79 FCVTAU (scalar SIMD&FP)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 1	0	1 1	0 0 0 0 0	0	Rn		Rd	
		S				rmode			opcode						

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTAU  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTAU  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTAU  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTAU  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, TRUE, FPCR(), FPRounding_TIEAWAY);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.80 FCVTL, FCVTL2

Floating-point convert to higher precision long (vector)

This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the `FCVTL2` variant operates on the elements in the top 64 bits of the source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	1	1	0	Rn	Rd
		U													opcode								

Encoding

```
FCVTL{2}  <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 16 << UInt(sz);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = Vpart{}(n, part);
var result : bits(2*datasize);

for e = 0 to elements-1 do
    result[e*:(2*esize)] = FPConvert{2*esize, esize}(operand[e*:esize], FPCR());
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.81 FCVTMS (vector)

Floating-point convert to signed integer, rounding toward minus infinity (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTMS  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTMS  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	25	24	23	22	19	18	17	16	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	Rn	Rd
		U						o2								o1				

Encoding

```
FCVTMS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd
		U						o2								o1				

Encoding

```
FCVTMS  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.82 FCVTMS (scalar)

Floating-point convert to signed integer, rounding toward minus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 0	0 0	0	0 0 0 0 0	0	Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTMS  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTMS  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTMS  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTMS  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTMS  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTMS  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_NEGINF;
let unsigned : boolean    = FALSE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.83 FCVTMS (scalar SIMD&FP)

Floating-point convert to signed integer, rounding toward minus infinity (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round toward Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 0	1	0 0	0 0 0 0 0	0	Rn		Rd	
		S				rmode		opcode							

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTMS  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTMS  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTMS  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTMS  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, FALSE, FPCR(), FPRounding_NEGINF);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.84 FCVTMU (vector)

Floating-point convert to unsigned integer, rounding toward minus infinity (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Encoding

```
FCVTMU  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn		Rd	
		U						o2											o1						

Encoding

```
FCVTMU  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 1 1 1	0 0	1 1 0 1	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTMU  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 0 1	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTMU  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.85 FCVTMU (scalar)

Floating-point convert to unsigned integer, rounding toward minus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 0	0 0	1	0 0 0 0 0	0	Rn		Rd	
		S					rmode			opcode					

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTMU  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTMU  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTMU  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTMU  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTMU  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTMU  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_NEGINF;
let unsigned : boolean    = TRUE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.86 FCVTMU (scalar SIMD&FP)

Floating-point convert to unsigned integer, rounding toward minus infinity (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round toward Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 0	1 0	1	0 0 0 0 0	0	Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTMU  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTMU  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTMU  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTMU  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, TRUE, FPCR(), FPRounding_NEGINF);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.87 FCVTN, FCVTN2 (double to single-precision, single to half-precision)

Floating-point convert to lower precision narrow (vector)

This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

`FCVTN` writes the vector to the lower half of the destination register and clears the upper half. `FCVTN2` writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				
		U													opcode																

Encoding

```
FCVTN{2}  <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 16 << UInt(sz);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(2*datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    result[e*:esize] = FPConvert{esize, 2*esize}(operand[e*:(2*esize)], FPCR());
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.88 FCVTN (half-precision to 8-bit floating-point)

Half-precision convert to 8-bit floating-point (vector)

This instruction converts half-precision elements of the two source vectors to 8-bit floating-point while scaling the values by $2^{\text{SInt(FPMR.NSCALE[4:0])}}$, and places the in-order results in the 8-bit elements of the destination vector.

The 8-bit floating-point encoding format is selected by FPMR.F8D.

Advanced SIMD

(FEAT_FP8)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0 1	0	Rm	1	1 1 1 0	1	Rn	Rd
		U				size				opcode			

Encoding

```
FCVTN  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV 16;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Tb>
0	4H
1	8H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    result[(0*elements + e)*:8] = FPConvertFP8{8, 16}(operand1[e*:16], FPCR(), FPMR());
    result[(1*elements + e)*:8] = FPConvertFP8{8, 16}(operand2[e*:16], FPCR(), FPMR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.89 FCVTN, FCVTN2 (single-precision to 8-bit floating-point)

Single-precision convert to 8-bit floating-point (vector)

This instruction converts each single-precision element of the two source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE)}}$, and places the in-order results in the 8-bit elements of the lower or upper half of the destination vector. `FCVTN` writes the results to the lower half of the destination vector and clears the upper half. `FCVTN2` writes the results to the upper half of the destination vector without affecting the other bits of the vector.

The 8-bit floating-point encoding format is selected by FPMR.F8D.

Advanced SIMD

(FEAT_FP8)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0 0	0	Rm	1	1 1 1 0	1	Rn	Rd
		U				size				opcode			

Encoding

```
FCVTN{2}  <Vd>.<Ta>, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let part : integer = UInt(Q);
let elements : integer = 128 DIV 32;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Ta>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
var result : bits(64);

for e = 0 to elements-1 do
    result[(0*elements + e)*:8] = FPConvertFP8{8, 32}(operand1[e*:32], FPCR(), FPMR());
    result[(1*elements + e)*:8] = FPConvertFP8{8, 32}(operand2[e*:32], FPCR(), FPMR());
end;

Vpart{64}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.90 FCVTNS (vector)

Floating-point convert to signed integer, rounding to nearest with ties to even (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTNS  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTNS  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 1 1 1	0 0	1 1 0 1	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTNS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 0 1	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTNS  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.91 FCVTNS (scalar)

Floating-point convert to signed integer, rounding to nearest with ties to even (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	0	0	0	0	0	0	0	0	0	0	0	Rn			Rd		
		S								rmode						opcode											

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTNS  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTNS  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTNS  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTNS  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTNS  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTNS  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_TIEEVEN;
let unsigned : boolean    = FALSE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.92 FCVTNS (scalar SIMD&FP)

Floating-point convert to signed integer, rounding to nearest with ties to even (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Even rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	0	1	0	1	0	0	0	0	0	0	0	Rn			Rd		
		S									rmode					opcode											

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTNS  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTNS  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTNS  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTNS  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, FALSE, FPCR(), FPRounding_TIEEVEN);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.93 FCVTNU (vector)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTNU  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTNU  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22–19	18–17	16–13	12	11–10	9–5	4–0
0	Q	1	0	1 1 1	0	0	1 1 1 1	0 0	1 1 0 1	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTNU  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21–17	16–13	12	11–10	9–5	4–0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 0 1	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTNU  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.94 FCVTNU (scalar)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1 0		ftype	1	0 0	0 0 1		0 0 0 0 0 0		Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTNU  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTNU  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTNU  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTNU  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTNU  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTNU  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_TIEEVEN;
let unsigned : boolean    = TRUE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.95 FCVTNU (scalar SIMD&FP)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Even rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	0 1	0 1	1	0 0 0 0 0	0	Rn		Rd	
		S				rmode		opcode							

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTNU  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTNU  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTNU  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTNU  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, TRUE, FPCR(), FPRounding_TIEEVEN);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.96 FCVTPS (vector)

Floating-point convert to signed integer, rounding toward plus infinity (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTPS  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTPS  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTPS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTPS  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.97 FCVTPS (scalar)

Floating-point convert to signed integer, rounding toward plus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1 0		ftype	1	0 1	0 0	0	0 0 0 0 0 0		Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTPS  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTPS  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTPS  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTPS  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTPS  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTPS  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_POSINF;
let unsigned : boolean    = FALSE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.98 FCVTPS (scalar SIMD&FP)

Floating-point convert to signed integer, rounding toward plus infinity (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round toward Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1 0		ftype	1	1 0	0 1 0		0 0 0 0 0 0		Rn		Rd	
		S				rmode		opcode							

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTPS  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTPS  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTPS  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTPS  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, FALSE, FPCR(), FPRounding_POSINF);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.99 FCVTPU (vector)

Floating-point convert to unsigned integer, rounding toward plus infinity (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTPU  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTPU  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 1 1 1	0 0	1 1 0 1	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTPU  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 0 1	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTPU  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.100 FCVTPU (scalar)

Floating-point convert to unsigned integer, rounding toward plus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	0	1	0	0	1	0	0	0	0	0	0	Rn			Rd		
		S									rmode					opcode											

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTPU  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTPU  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTPU  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTPU  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTPU  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTPU  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_POSINF;
let unsigned : boolean    = TRUE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.101 FCVTPU (scalar SIMD&FP)

Floating-point convert to unsigned integer, rounding toward plus infinity (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round toward Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	1	0	0	1	1	0	0	0	0	0	0	Rn			Rd		
		S									rmode				opcode												

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTPU  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTPU  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTPU  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTPU  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, TRUE, FPCR(), FPRounding_POSINF);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.102 FCVTXN, FCVTXN2

Floating-point convert to lower precision narrow, rounding to odd (vector)

This instruction reads each double-precision element in the source SIMD&FP register, converts each value to single-precision format using the Round to Odd rounding mode, and writes the results to the destination SIMD&FP register.

Note

This instruction uses the Round to Odd rounding mode, which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact, the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The Vector variant of `FCVTXN` writes the half-width results to the lower half of the destination register and clears the upper half to 0. `FCVTXN2` writes the half-width results to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	0	1	1	0	0	0	0	1	0	1	1	0	1	0	Rn	Rd
		U						size							opcode								

Encoding

```
FCVTXN  S<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32;
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				
		U						size							opcode																

Encoding

```
FCVTXN{2}  <Vd>.<Tb>, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32;
let datasize : integer{} = 64;
let elements : integer = 2;
let part : integer = UInt(Q);
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Tb>
0	2S
1	4S

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand : bits(2*datasize) = V{}(n);
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

for e = 0 to elements-1 do
    result[e*:esize] = FPConvert{esize, 2*esize}(operand[e*:(2*esize)], FPCR(), FPRounding_ODD);
end;

if merge then
    V{128}(d) = result;
else
    Vpart{datasize}(d, part) = result[0+:datasize];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.103 FCVTZS (vector, fixed-point)

Floating-point convert to signed fixed-point, rounding toward zero (vector)

This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	!=0000	immb	1 1 1 1 1	1	Rn	Rd
		U				immh		opcode			

Encoding

```
FCVTZS  <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = esize;
let elements : integer = 1;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	1 1 1 1 1	1	Rn	Rd
		U				immh		opcode			

Encoding

```
FCVTZS  <Vd>.<T>, <Vn>.<T>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<fbits>

For the “Scalar” variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the “Vector” variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.104 FCVTZS (vector, integer)

Floating-point convert to signed integer, rounding toward zero (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTZS  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTZS  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTZS  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTZS  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.105 FCVTZS (scalar, fixed-point)

Floating-point convert to signed fixed-point, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1 0		ftype	0	1 1	0 0 0		scale		Rn		Rd	
		S				rmode		opcode							

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZS  <Wd>, <Hn>, #<fbits>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZS  <Xd>, <Hn>, #<fbits>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZS  <Wd>, <Sn>, #<fbits>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZS  <Xd>, <Sn>, #<fbits>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZS  <Wd>, <Dn>, #<fbits>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZS  <Xd>, <Dn>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && scale[5] == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let decode_fltsize : integer{} = 8 << UInt(ftype XOR '10');

let fracbits : integer = 64 - UInt(scale);

let rounding : FPRounding = FPRounding_ZERO;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<fbits>

For the “Double-precision to 32-bit”, “Half-precision to 32-bit”, and “Single-precision to 32-bit” variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus “scale”.

For the “Double-precision to 64-bit”, “Half-precision to 64-bit”, and “Single-precision to 64-bit” variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus “scale”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(decode_fltsize) = V{}(n);
X{intsize}(d) = FPToFixed{intsize, decode_fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.106 FCVTZS (scalar, integer)

Floating-point convert to signed integer, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 1	0 0	0	0 0 0 0 0	0	Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZS  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZS  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZS  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZS  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZS  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZS  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_ZERO;
let unsigned : boolean    = FALSE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.107 FCVTZS (scalar SIMD&FP)

Floating-point convert to signed integer, rounding toward zero (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round toward Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	1	0	1	1	0	0	0	0	0	0	0	Rn			Rd		
		S									rmode				opcode												

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTZS  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTZS  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTZS  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTZS  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, FALSE, FPCR(), FPRounding_ZERO);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.108 FCVTZU (vector, fixed-point)

Floating-point convert to unsigned fixed-point, rounding toward zero (vector)

This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–11	10	9–5	4–0
0	1	1	1	1 1 1	1 0	!=0000	immb	1 1 1 1 1	1	Rn	Rd
		U				immh		opcode			

Encoding

```
FCVTZU  <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = esize;
let elements : integer = 1;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–11	10	9–5	4–0
0	Q	1	0	1 1 1	1 0	!=0000	immb	1 1 1 1 1	1	Rn	Rd
		U				immh		opcode			

Encoding

```
FCVTZU  <Vd>.<T>, <Vn>.<T>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<fbits>

For the “Scalar” variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the “Vector” variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.109 FCVTZU (vector, integer)

Floating-point convert to unsigned integer, rounding toward zero (vector)

This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTZU  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn	Rd
		U						o2											o1				

Encoding

```
FCVTZU  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 1 1 1	0 0	1 1 0 1	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTZU  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 0 1	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FCVTZU  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : FPRounding = FPDecodeRounding(o1::o2);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPToFixed{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.110 FCVTZU (scalar, fixed-point)

Floating-point convert to unsigned fixed-point, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	0	1 1	0 0	1	scale		Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZU  <Wd>, <Hn>, #<fbits>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZU  <Xd>, <Hn>, #<fbits>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZU  <Wd>, <Sn>, #<fbits>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZU  <Xd>, <Sn>, #<fbits>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZU  <Wd>, <Dn>, #<fbits>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZU  <Xd>, <Dn>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && scale[5] == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let decode_fltsize : integer{} = 8 << UInt(ftype XOR '10');

let fracbits : integer = 64 - UInt(scale);

let rounding : FPRounding = FPRounding_ZERO;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<fbits>

For the “Double-precision to 32-bit”, “Half-precision to 32-bit”, and “Single-precision to 32-bit” variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus “scale”.

For the “Double-precision to 64-bit”, “Half-precision to 64-bit”, and “Single-precision to 64-bit” variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus “scale”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(decode_fltsize) = V{}(n);
X{intsize}(d) = FPToFixed{intsize, decode_fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.111 FCVTZU (scalar, integer)

Floating-point convert to unsigned integer, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	1 1	0 0	1	0 0 0 0 0	0	Rn		Rd	
		S					rmode			opcode					

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZU  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZU  <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZU  <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZU  <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZU  <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZU  <Xd>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');

let rounding : FPRounding = FPRounding_ZERO;
let unsigned : boolean    = TRUE;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltval : bits(fltsize) = V{}(n);
let fracbits : integer = 0;

X{intsize}(d) = FPToFixed{intsize, fltsize}(fltval, fracbits, unsigned, FPCR(), rounding);
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.112 FCVTZU (scalar SIMD&FP)

Floating-point convert to unsigned integer, rounding toward zero (scalar SIMD&FP)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round toward Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FPRCVT)

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1 0		ftype	1	1 0	1 1 1		0 0 0 0 0 0		Rn		Rd	
		S				rmode		opcode							

Encoding for the Half-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 11)`

```
FCVTZU  <Sd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 11)`

```
FCVTZU  <Dd>, <Hn>
```

Encoding for the Single-precision to 64-bit variant

Applies when `(sf == 1 && ftype == 00)`

```
FCVTZU  <Dd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant

Applies when `(sf == 0 && ftype == 01)`

```
FCVTZU  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

fltval = V{fltsize}(n);
intval = FPToFixed{intsize, fltsize}(fltval, 0, TRUE, FPCR(), FPRounding_ZERO);
result[0+:intsize] = intval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.113 FDIV (vector)

Floating-point divide (vector)

This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

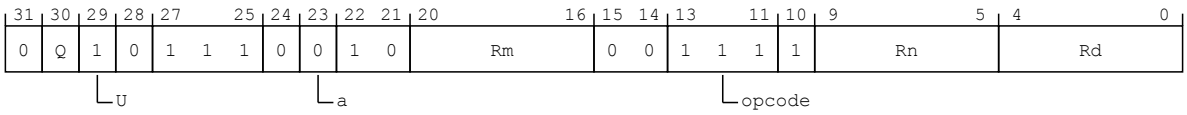

Encoding

```
FDIV  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

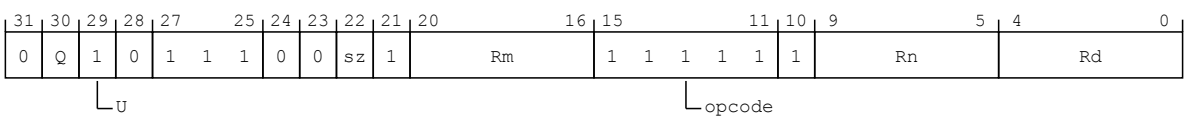

Encoding

```
FDIV  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPDiv{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.114 FDIV (scalar)

Floating-point divide (scalar)

This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20	16	15	12	11 10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1	Rm		0 0 0	1	1 0	Rn		Rd	

M S opcode

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FDIV  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FDIV  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FDIV  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPDiv{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.115 FDOT (8-bit floating-point to half-precision, by element)

8-bit floating-point dot product to half-precision (vector, by element)

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in each 16-bit element of the first source vector and a pair of 8-bit floating-point values in an indexed 16-bit element of the second source vector. The half-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the destination vector.

The 8-bit floating-point groups within the second source vector are specified using an immediate index.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8DOT2)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	0 1	L	M	Rm	0 0 0 0	H	0	Rn	Rd
		U				size				opcode				

Encoding

```
FDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8DOT2) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let m : integer = UInt('0'::Rm);
let i : integer = UInt(H::L::M);

let datasize : integer{} = if Q == '1' then 128 else 64;
let esize : integer{} = 16;
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	4H
1	8H

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

<index>

Is the immediate index of a pair of 8-bit elements, in the range 0 to 7, encoded in the “H:L:M” fields.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : bits(esize) = operand2[i*:esize];
    let sum : bits(esize) = operand3[e*:esize];

    result[e*:esize] = FP8DotAddFP{esize, esize}(sum, op1, op2, FPCR(), FPMR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.116 FDOT (8-bit floating-point to half-precision, vector)

8-bit floating-point dot product to half-precision (vector)

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in each 16-bit element of the first and second source vectors. The half-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the destination vector.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8DOT2)

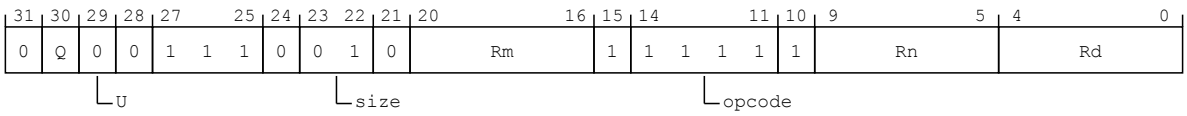

Encoding

```
FDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8DOT2) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = if Q == '1' then 128 else 64;
let esize : integer{} = 16;
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	4H
1	8H

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : bits(esize) = operand2[e*:esize];
    var sum : bits(esize) = operand3[e*:esize];

    sum = FP8DotAddFP{esize, esize}(sum, op1, op2, FPCR(), FPMR());
    result[e*:esize] = sum;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.117 FDOT (8-bit floating-point to single-precision, by element)

8-bit floating-point dot product to single-precision (vector, by element)

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in each 32-bit element of the first source vector and a group of four 8-bit floating-point values in an indexed 32-bit element of the second source vector. The single-precision sum-of-products are scaled by $2^{-UInt(FPMR.LSCALE)}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the destination vector.

The 8-bit floating-point groups within the second source vector are specified using an immediate index.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8DOT4)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	0 0	L	M	Rm	0 0 0 0	H	0	Rn	Rd
		U				size				opcode				

Encoding

```
FDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8DOT4) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let m : integer = UInt(M::Rm);
let i : integer = UInt(H::L);

let datasize : integer{} = if Q == '1' then 128 else 64;
let esize : integer{} = 32;
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the immediate index of a group of four 8-bit values, in the range 0 to 3, encoded in the “H:L” fields.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : bits(esize) = operand2[i*:esize];
    let sum : bits(esize) = operand3[e*:esize];

    result[e*:esize] = FP8DotAddFP{esize, esize}(sum, op1, op2, FPCR(), FPMR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.118 FDOT (8-bit floating-point to single-precision, vector)

8-bit floating-point dot product to single-precision (vector)

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in each 32-bit element of the first and second source vectors. The single-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the destination vector.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8DOT4)

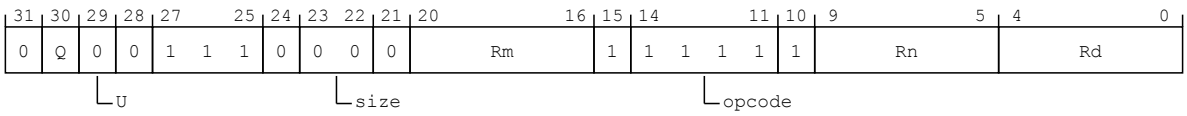

Encoding

```
FDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8DOT4) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = if Q == '1' then 128 else 64;
let esize : integer{} = 32;
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : bits(esize) = operand2[e*:esize];
    var sum : bits(esize) = operand3[e*:esize];

    sum = FP8DotAddFP{esize, esize}(sum, op1, op2, FPCR(), FPMR());
    result[e*:esize] = sum;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.119 FJCVTZS

Floating-point JavaScript convert to signed fixed-point, rounding toward zero

This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit

(FEAT_JSCVT)

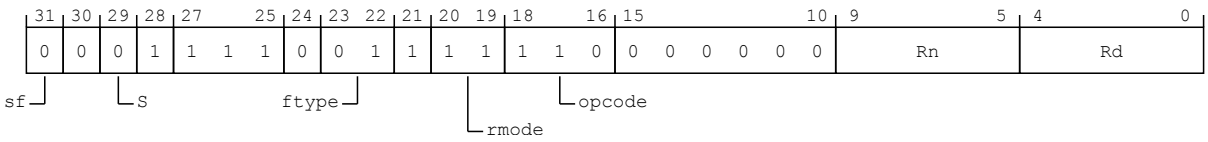

Encoding

```
FJCVTZS  <Wd>, <Dn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let fltval : bits(64) = V{}(n);

var intval : bits(32);
var z : bit;
(intval, z) = FPToFixedJS(fltval, FPCR());

X{32}(d) = intval;
PSTATE.[N,Z,C,V] = '0'::z::'00';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.120 FMADD

Floating-point fused multiply-add (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28 … 24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1 1 1 1 1	ftype	0	Rm	0	Ra	Rn	Rd
M		S			o1		o0			

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMADD  <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMADD  <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMADD  <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ha>

Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Sa>

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Da>

Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPEnabled();

let addend : bits(esize)   = V{esize}(a);
let operand1 : bits(esize) = V{esize}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(a) else Zeros{128};

result[0+:esize] = FPMulAdd{esize}(addend, operand1, operand2, FPCR());
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.121 FMAX (vector)

Floating-point maximum (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 0	Rm	0 0	1 1 0	1	Rn	Rd
		U				o1				opcode			

Encoding

```
FMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1	Rm	1 1 1 1 0	1	Rn	Rd
		U				o1				opcode			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPMax{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.122 FMAX (scalar)

Floating-point maximum (scalar)

This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

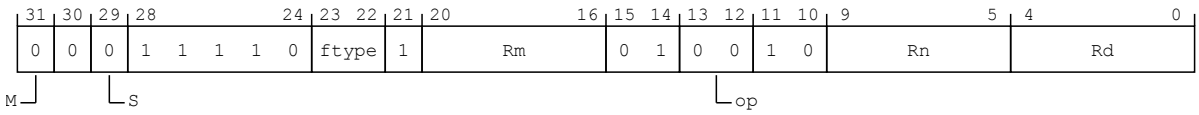

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMAX  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMAX  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMAX  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPMax{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.123 FMAXNM (vector)

Floating-point maximum number (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 0	Rm	0 0	0 0 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMAXNM  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1	Rm	1 1 0 0 0	1	Rn	Rd
		U				o1				opcode			

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMAXNM  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPMaxNum{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.124 FMAXNM (scalar)

Floating-point maximum number (scalar)

This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

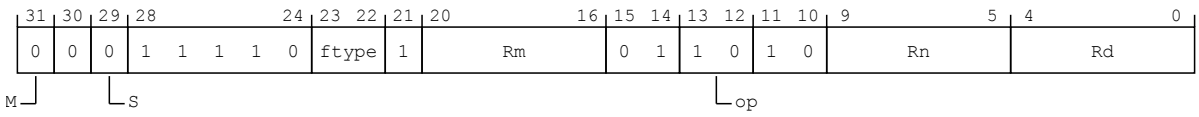

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMAXNM  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMAXNM  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMAXNM  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPMaxNum{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.125 FMAXNMP (scalar)

Floating-point maximum number of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U						o1	sz						opcode																

Encoding

```
FMAXNMP  H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 32;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U						o1							opcode																

Encoding

```
FMAXNMP  <V><d>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize * 2;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<T>

Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
let element1 : bits(esize) = operand[0*:esize];
let element2 : bits(esize) = operand[1*:esize];
V{esize}(d) = FPMaxNum{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.126 FMAXNMP (vector)

Floating-point maximum number pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 0	Rm	0 0	0 0 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMAXNMP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1	Rm	1 1 0 0 0	1	Rn	Rd
		U				o1				opcode			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMAXNMP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : bits(esize);
var element2 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = concat[(2*e)*:esize];
    element2 = concat[((2*e)+1)*:esize];
    result[e*:esize] = FPMaxNum{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.127 FMAXNMV

Floating-point maximum number across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

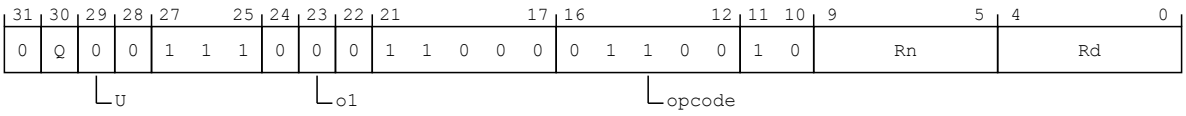

Encoding

```
FMAXNMV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
```

Single-precision

(FEAT_AdvSIMD)

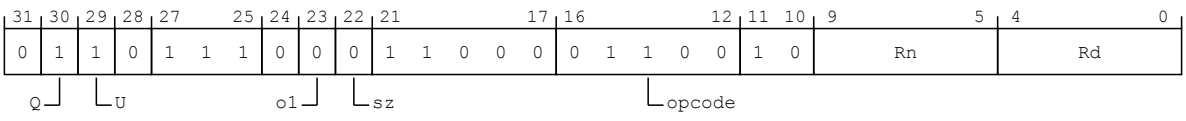

Encoding

```
FMAXNMV  S<d>, <Vn>.4S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q != '01' then EndOfDecode(Decode_UNDEF); end;    // .4S only

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<V>

Is the destination width specifier, H.

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
V{esize}(d) = FPReduce{esize, datasize}(ReduceOp_FMAXNUM, operand, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.128 FMAXP (scalar)

Floating-point maximum of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21–17	16–12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	0	0	1 1 0 0 0	0 1 1 1 1	1 0	Rn	Rd
		U				o1	sz		opcode			

Encoding

```
FMAXP  H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 32;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21–17	16–12	11–10	9–5	4–0
0	1	1	1	1 1 1	0	0	sz	1 1 0 0 0	0 1 1 1 1	1 0	Rn	Rd
		U				o1			opcode			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMAXP  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize * 2;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<T>

Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand :  bits(datasize) = V{}(n);
let element1 : bits(esize) = operand[0*:esize];
let element2 : bits(esize) = operand[1*:esize];
V{esize}(d) = FPMax{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.129 FMAXP (vector)

Floating-point maximum pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

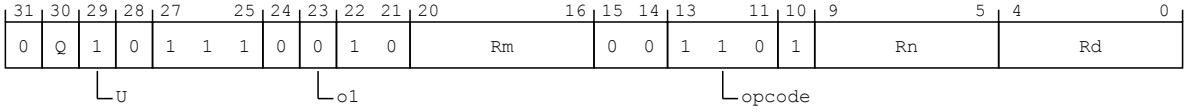

Encoding

```
FMAXP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	21	20	16	15	11	10	9	5	4	0
0	Q	1	0	1 1	1	0	0	sz	1	Rm		1 1 1 1	0	1	Rn		Rd	

U, o1, opcode

Encoding

```
FMAXP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = concat[(2*e)*:esize];
    element2 = concat[((2*e)+1)*:esize];
    result[e*:esize] = FPMax{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.130 FMAXV

Floating-point maximum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

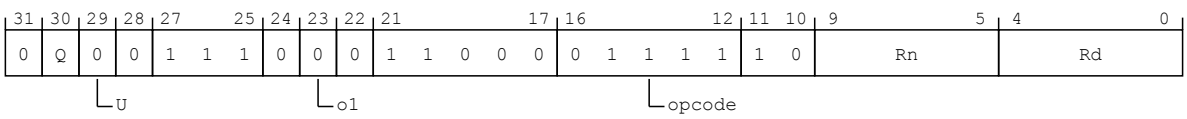

Encoding

```
FMAXV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
```

Single-precision

(FEAT_AdvSIMD)

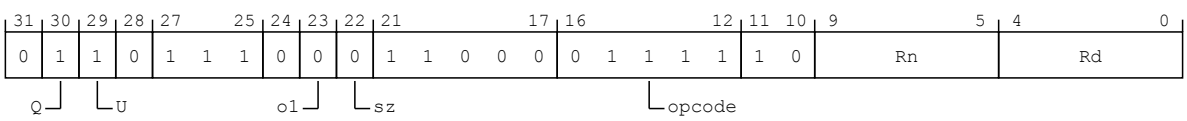

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMAXV  S<d>, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q != '01' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<V>

Is the destination width specifier, H.

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand :  bits(datasize) = V{}(n);
V{esize}(d) = FPReduce{esize, datasize}(ReduceOp_FMAX, operand, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.131 FMIN (vector)

Floating-point minimum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 0	Rm	0 0	1 1 0	1	Rn	Rd
		U				o1				opcode			

Encoding

```
FMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1	Rm	1 1 1 1 0	1	Rn	Rd
		U				o1				opcode			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPMin{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.132 FMIN (scalar)

Floating-point minimum (scalar)

This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	0	0	1	1	1	1	0	ftype		1	Rm				0	1	0	1	1	0	Rn				Rd			
M		S															op											

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMIN  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMIN  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMIN  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPMin{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.133 FMINNM (vector)

Floating-point minimum number (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 0	Rm	0 0	0 0 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMINNM  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1	Rm	1 1 0 0 0	1	Rn	Rd
		U				o1				opcode			

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMINNM  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPMinNum{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.134 FMINNM (scalar)

Floating-point minimum number (scalar)

This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28 … 24	23 22	21	20 … 16	15 14	13 12	11 10	9 … 5	4 … 0
0	0	0	1 1 1 1 0	ftype	1	Rm	0 1	1 1	1 0	Rn	Rd
M		S						op			

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMINNM  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMINNM  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMINNM  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPMinNum{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.135 FMINNMP (scalar)

Floating-point minimum number of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U						o1	sz						opcode																

Encoding

```
FMINNMP  H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 32;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U						o1							opcode																

Encoding

```
FMINNMP  <V><d>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize * 2;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<T>

Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
let element1 : bits(esize) = operand[0*:esize];
let element2 : bits(esize) = operand[1*:esize];
V{esize}(d) = FPMinNum{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.136 FMINNMP (vector)

Floating-point minimum number pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22–21	20–16	15–14	13–11	10	9–5	4–0
0	Q	1	0	1 1 1	0	1	1 0	Rm	0 0	0 0 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMINNMP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–11	10	9–5	4–0
0	Q	1	0	1 1 1	0	1	sz	1	Rm	1 1 0 0 0	1	Rn	Rd
		U				o1				opcode			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMINNMP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : bits(esize);
var element2 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = concat[(2*e)*:esize];
    element2 = concat[((2*e)+1)*:esize];
    result[e*:esize] = FPMinNum{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.137 FMINNMV

Floating-point minimum number across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

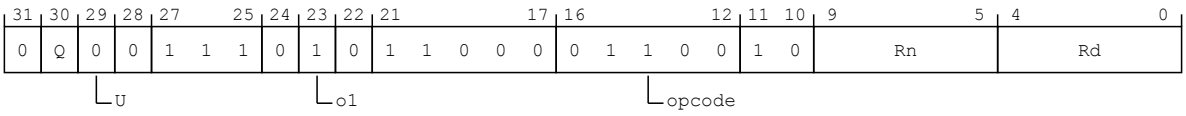

Encoding

```
FMINNMV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
```

Single-precision

(FEAT_AdvSIMD)

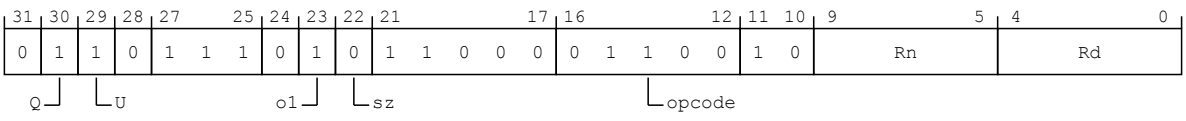

Encoding

```
FMINNMV  S<d>, <Vn>.4S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q != '01' then EndOfDecode(Decode_UNDEF); end;    // .4S only

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<V>

Is the destination width specifier, H.

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
V{esize}(d) = FPReduce{esize, datasize}(ReduceOp_FMINNUM, operand, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.138 FMINP (scalar)

Floating-point minimum of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21–17	16–12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	1	0	1 1 0 0 0	0 1 1 1 1	1 0	Rn	Rd
		U				o1	sz		opcode			

Encoding

```
FMINP  H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 32;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21–17	16–12	11–10	9–5	4–0
0	1	1	1	1 1 1	0	1	sz	1 1 0 0 0	0 1 1 1 1	1 0	Rn	Rd
		U				o1			opcode			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMINP  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize * 2;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<T>

Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand :  bits(datasize) = V{}(n);
let element1 : bits(esize) = operand[0*:esize];
let element2 : bits(esize) = operand[1*:esize];
V{esize}(d) = FPMin{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.139 FMINP (vector)

Floating-point minimum pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

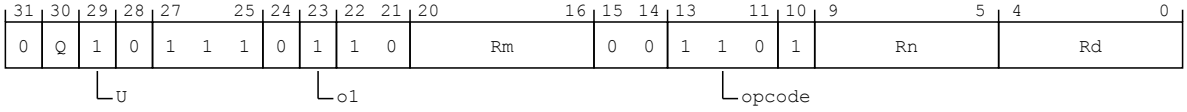

Encoding

```
FMINP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
		U						o1								opcode															

Encoding

```
FMINP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = concat[(2*e)*:esize];
    element2 = concat[((2*e)+1)*:esize];
    result[e*:esize] = FPMin{esize}(element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.140 FMINV

Floating-point minimum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

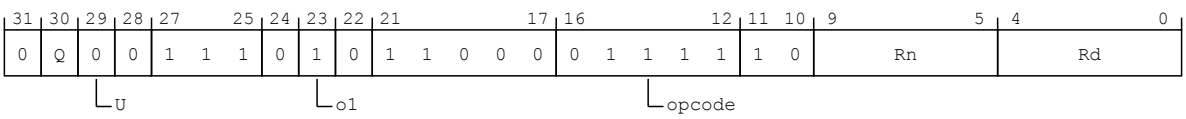

Encoding

```
FMINV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
```

Single-precision

(FEAT_AdvSIMD)

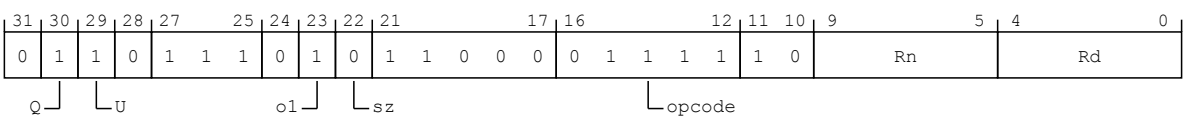

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMINV  S<d>, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q != '01' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<V>

Is the destination width specifier, H.

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand :  bits(datasize) = V{}(n);
V{esize}(d) = FPReduce{esize, datasize}(ReduceOp_FMIN, operand, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.141 FMLA (by element)

Floating-point fused multiply-add to accumulator (by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision, and Vector, single-precision and double-precision

Scalar, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	0	0	L	M	Rm	0	0	0	1	H	0	Rn	Rd
		U						size						o2						

Encoding

```
FMLA  <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm	0	0	0	1	H	0	Rn	Rd
		U												o2						

Encoding

```
FMLA  <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	0 0	L	M	Rm	0	0	0 1	H	0	Rn	Rd
		U				size					o2					

Encoding

```
FMLA  <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	1	sz	L	M	Rm	0	0	0 1	H	0	Rn	Rd
		U										o2					

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMLA  <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<V>

Is a width specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<Ts>

Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector, half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var element1 : bits(esize);
let element2 : bits(esize) = operand2[index*:esize];
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    result[e*:esize] = FPMulAdd{esize}(operand3[e*:esize], element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.142 FMLA (vector)

Floating-point fused multiply-add to accumulator (vector)

This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 0	Rm	0 0	0 0 1	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMLA  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1	Rm	1 1 0 0 1	1	Rn	Rd
		U				op				opcode			

Encoding

```
FMLA  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:esize] = FPMulAdd{esize}(operand3[e*:esize], element1, element2, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.143 FMLAL, FMLAL2 (by element)

Floating-point fused multiply-add long to accumulator (by element)

This instruction multiplies the half-precision vector elements in the first source SIMD&FP register by the specified half-precision value in the second source SIMD&FP register, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4, it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	1	0	L	M	Rm	0	0	0	0	H	0	Rn	Rd
		U							sz					S						

Encoding

```
FMLAL  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt('0'::Rm);    // Vm can only be in bottom 16 registers.
let index : integer = UInt(H::L::M);

let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let part : integer = 0;
```

FMLAL2

(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	1	1	0	L	M	Rm	1	0	0	0	H	0	Rn	Rd
		U							sz					S						

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMLAL2  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt('0'::Rm);    // Vm can only be in bottom 16 registers.
let index : integer = UInt(H::L::M);

let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let part : integer = 1;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<index>

Is the element index, encoded in the “H:L:M” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize DIV 2) = Vpart{}(n, part);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize DIV 2);
let element2 : bits(esize DIV 2) = operand2[index*:(esize DIV 2)];

for e = 0 to elements-1 do
    element1 = operand1[e*:(esize DIV 2)];
    result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.144 FMLAL, FMLAL2 (vector)

Floating-point fused multiply-add long to accumulator (vector)

This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4, it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(FEAT_FHM)

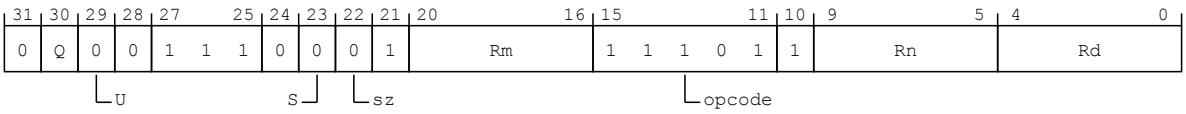

Encoding

```
FMLAL  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = 0;
```

FMLAL2

(FEAT_FHM)

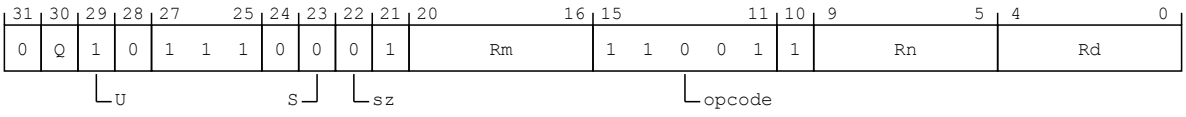

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMLAL2  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = 1;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize DIV 2) = Vpart{}(n, part);
let operand2 : bits(datasize DIV 2) = Vpart{}(m, part);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = operand1[e*:(esize DIV 2)];
    element2 = operand2[e*:(esize DIV 2)];
    result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.145 FMLALB, FMLALT (by element)

8-bit floating-point multiply-add to half-precision (vector, by element)

This instruction widens the even-numbered (bottom) or odd-numbered (top) 8-bit elements in the first source vector and the indexed element from the second source vector to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the half-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8FMA)

31	30	29	28				24	23	22	21	20	19			16	15			12	11	10	9			5	4			0
0	Q	0	0	1	1	1	1	1	1	L	M	Rm				0	0	0	0	H	0	Rn				Rd			
		U						size								opcode													

Encoding for the FMLALB variant

Applies when `(Q == 0)`

```
FMLALB  <Vd>.8H, <Vn>.16B, <Vm>.B[<index>]
```

Encoding for the FMLALT variant

Applies when `(Q == 1)`

```
FMLALT  <Vd>.8H, <Vn>.16B, <Vm>.B[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8FMA) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt('00'::Rm[2:0]);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M::Rm[3]);
let elements : integer = 128 DIV 16;
let sel : integer = UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, in the range V0 to V7, encoded in the “Rm<2:0>” field.

<index>

Is the element index, in the range 0 to 15, encoded in the “H:L:M:Rm<3>” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);
var result : bits(128);

for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(2 * e + sel)*:8];
    let element2 : bits(8) = operand2[index*:8];
    let element3 : bits(16) = operand3[e*:16];
    result[e*:16] = FP8MulAddFP{16}(element3, element1, element2, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.146 FMLALB, FMLALT (vector)

8-bit floating-point multiply-add to half-precision (vector)

This instruction widens the even-numbered (bottom) or odd-numbered (top) 8-bit elements in the first and second source vectors to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the half-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8FMA)

31	30	29	28				24	23	22	21	20			16	15	14				11	10	9			5	4			0
0	Q	0	0	1	1	1	0	1	1	0	Rm				1	1	1	1	1		1	Rn				Rd			
		U						size								opcode													

Encoding for the FMLALB variant

Applies when `(Q == 0)`

```
FMLALB  <Vd>.8H, <Vn>.16B, <Vm>.16B
```

Encoding for the FMLALT variant

Applies when `(Q == 1)`

```
FMLALT  <Vd>.8H, <Vn>.16B, <Vm>.16B
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8FMA) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let elements : integer = 128 DIV 16;
let sel : integer = UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);
var result : bits(128);

for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(2 * e + sel)*:8];
    let element2 : bits(8) = operand2[(2 * e + sel)*:8];
    let element3 : bits(16) = operand3[e*:16];
    result[e*:16] = FP8MulAddFP{16}(element3, element1, element2, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.147 FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (by element)

8-bit floating-point multiply-add to single-precision (vector, by element)

This instruction widens the first (bottom bottom), second (bottom top), third (top bottom), or fourth (top top) 8-bit element of each 32-bit container in the first source vector and the indexed element from the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$, before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8FMA)

31	30	29	28				24	23	22	21	20	19			16	15			12	11	10	9			5	4			0
0	Q	1	0	1	1	1	1	0	x	L	M	Rm				1	0	0	0	H	0	Rn				Rd			
		U						size								opcode													

Encoding for the FMLALLBB variant

Applies when `(Q == 0 && size == 00)`

```
FMLALLBB  <Vd>.4S, <Vn>.16B, <Vm>.B[<index>]
```

Encoding for the FMLALLBT variant

Applies when `(Q == 0 && size == 01)`

```
FMLALLBT  <Vd>.4S, <Vn>.16B, <Vm>.B[<index>]
```

Encoding for the FMLALLTB variant

Applies when `(Q == 1 && size == 00)`

```
FMLALLTB  <Vd>.4S, <Vn>.16B, <Vm>.B[<index>]
```

Encoding for the FMLALLTT variant

Applies when `(Q == 1 && size == 01)`

```
FMLALLTT  <Vd>.4S, <Vn>.16B, <Vm>.B[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8FMA) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt('00'::Rm[2:0]);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M::Rm[3]);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q::size[0]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, in the range V0 to V7, encoded in the “Rm<2:0>” field.

<index>

Is the element index, in the range 0 to 15, encoded in the “H:L:M:Rm<3>” fields.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);
var result : bits(128);

for e = 0 to elements-1 do
    let element1 : bits(8)  = operand1[(4 * e + sel)*:8];
    let element2 : bits(8)  = operand2[index*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.148 FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT (vector)

8-bit floating-point multiply-add to single-precision (vector)

This instruction widens the first (bottom bottom), second (bottom top), third (top bottom), or fourth (top top) 8-bit element of each 32-bit container in the first and second source vectors to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_FP8FMA)

31	30	29	28				24	23	22	21	20			16	15	14				11	10	9			5	4			0
0	Q	0	0	1	1	1	0	0	x	0	Rm				1	1	0	0	0		1	Rn				Rd			
		U						size								opcode													

Encoding for the FMLALLBB variant

Applies when `(Q == 0 && size == 00)`

```
FMLALLBB  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Encoding for the FMLALLBT variant

Applies when `(Q == 0 && size == 01)`

```
FMLALLBT  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Encoding for the FMLALLTB variant

Applies when `(Q == 1 && size == 00)`

```
FMLALLTB  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Encoding for the FMLALLTT variant

Applies when `(Q == 1 && size == 01)`

```
FMLALLTT  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP8FMA) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q::size[0]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled(); AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);
var result : bits(128);

for e = 0 to elements-1 do
    let element1 : bits(8)  = operand1[(4 * e + sel)*:8];
    let element2 : bits(8)  = operand2[(4 * e + sel)*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.149 FMLS (by element)

Floating-point fused multiply-subtract from accumulator (by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision, and Vector, single-precision and double-precision

Scalar, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	0	0	L	M	Rm	0	1	0	1	H	0	Rn	Rd
		U						size						o2						

Encoding

```
FMLS  <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm	0	1	0	1	H	0	Rn	Rd
		U												o2						

Encoding

```
FMLS  <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	0 0	L	M	Rm	0	1	0 1	H	0	Rn	Rd
		U				size					o2					

Encoding

```
FMLS  <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Rd);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	1	sz	L	M	Rm	0	1	0 1	H	0	Rn	Rd
		U										o2					

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMLS  <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<V>

Is a width specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<Ts>

Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector, half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var element1 : bits(esize);
let element2 : bits(esize) = operand2[index*:esize];
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

for e = 0 to elements-1 do
    element1 = FPNeg{esize}(operand1[e*:esize], FPCR());
    result[e*:esize] = FPMulAdd{esize}(operand3[e*:esize], element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.150 FMLS (vector)

Floating-point fused multiply-subtract from accumulator (vector)

This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 0	Rm	0 0	0 0 1	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMLS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1	Rm	1 1 0 0 1	1	Rn	Rd
		U				op				opcode			

Encoding

```
FMLS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = FPNeg{esize}(operand1[e*:esize], FPCR());
    element2 = operand2[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:esize] = FPMulAdd{esize}(operand3[e*:esize], element1, element2, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.151 FMLSL, FMLSL2 (by element)

Floating-point fused multiply-subtract long from accumulator (by element)

This instruction multiplies the negated half-precision vector elements in the first source SIMD&FP register by the specified half-precision value in the second source SIMD&FP register, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4, it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL

(FEAT_FHM)

31	30	29	28	27 25	24	23	22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	1	0	L	M	Rm	0	1	0 0	H	0	Rn	Rd
		U					sz					S					

Encoding

```
FMLSL  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt('0'::Rm);    // Vm can only be in bottom 16 registers.
let index : integer = UInt(H::L::M);

let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let part : integer = 0;
```

FMLSL2

(FEAT_FHM)

31	30	29	28	27 25	24	23	22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	1	0	L	M	Rm	1	1	0 0	H	0	Rn	Rd
		U					sz					S					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMLSL2  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt('0'::Rm);    // Vm can only be in bottom 16 registers.
let index : integer = UInt(H::L::M);

let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let part : integer = 1;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<index>

Is the element index, encoded in the “H:L:M” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize DIV 2) = Vpart{}(n, part);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize DIV 2);
let element2 : bits(esize DIV 2) = operand2[index*:(esize DIV 2)];

for e = 0 to elements-1 do
    element1 = FPNeg{esize DIV 2}(operand1[e*:(esize DIV 2)], FPCR());
    result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.152 FMLSL, FMLSL2 (vector)

Floating-point fused multiply-subtract long from accumulator (vector)

This instruction negates the half-precision values in the vector of one SIMD&FP register, multiplies these with the corresponding half-precision values in another vector, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4, it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL

(FEAT_FHM)

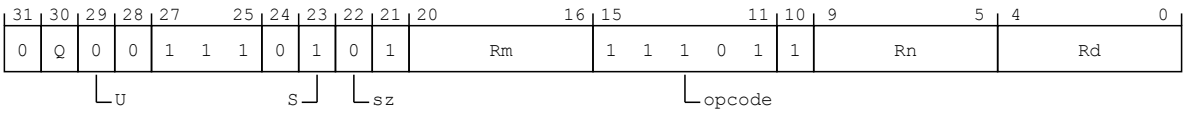

Encoding

```
FMLSL  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = 0;
```

FMLSL2

(FEAT_FHM)

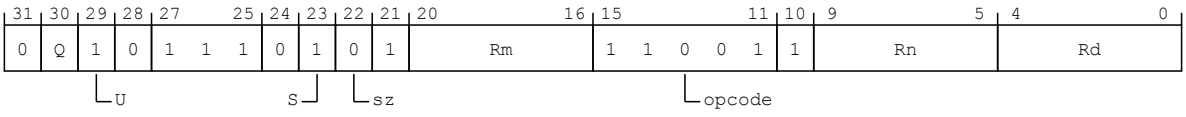

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMLSL2  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = 1;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize DIV 2) = Vpart{}(n, part);
let operand2 : bits(datasize DIV 2) = Vpart{}(m, part);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    element1 = FPNeg{esize DIV 2}(operand1[e*:(esize DIV 2)], FPCR());
    element2 = operand2[e*:(esize DIV 2)];
    result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, FPCR());
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.153 FMMLA (widening, 8-bit floating-point to half-precision)

8-bit floating-point matrix multiply-accumulate to half-precision

This instruction multiplies the 2×4 matrix of 8-bit floating-point values held in each 64-bit segment of the first source vector by the 4×2 matrix of 8-bit floating-point values in the corresponding segment of the second source vector. The intermediate half-precision results, calculated as a fused sum-of-products using four adjacent 8-bit floating-point values from each of the source vectors, are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the 2x2 half-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating 4-way dot product per destination element.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_F8F16MM)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	1	1	0	1 1 1	0	0 0	0	Rm	1	1 1 0 1	1	Rn	Rd
	Q	U				size				opcode			

Encoding

```
FMMLA  <Vd>.8H, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F8F16MM) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Rd);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled();
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(128) = V{}(d);

var result : bits(128);
var op1, op2, acc : bits(64);

for s = 0 to 1 do
    op1 = operand1[s*:64];
    op2 = operand2[s*:64];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    acc = operand3[s*:64];
    result[s*:64] = FP8MatMulAddFP{64}(acc, op1, op2, 4, FPCR(), FPMR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.154 FMMLA (widening, 8-bit floating-point to single-precision)

8-bit floating-point matrix multiply-accumulate to single-precision

This instruction multiplies the 2×8 matrix of 8-bit floating-point values held in the first source vector by the 8×2 matrix of 8-bit floating-point values in the second source vector. The intermediate single-precision results, calculated as a fused sum-of-products using eight adjacent 8-bit floating-point values from each of the source vectors, are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$ before being destructively added without intermediate rounding to the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating 8-way dot product per destination element.

The 8-bit floating-point encoding format for the elements of the first source vector is selected by FPMR.F8S1. The 8-bit floating-point encoding format for the elements of the second source vector is selected by FPMR.F8S2.

Advanced SIMD

(FEAT_F8F32MM)

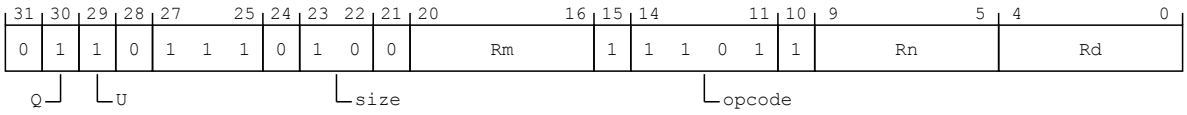

Encoding

```
FMMLA  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F8F32MM) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Rd);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPMREnabled();
AArch64_CheckFPAdvSIMDEnabled();
let op1 : bits(128) = V{}(n);
let op2 : bits(128) = V{}(m);
let acc : bits(128) = V{}(d);

V{128}(d) = FP8MatMulAddFP{128}(acc, op1, op2, 8, FPCR(), FPMR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.155 FMOV (vector, immediate)

Floating-point move immediate (vector)

This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24 23	22 19	18	17	16	15 12	11	10	9	8	7	6	5	4 0
0	Q	0	0	1 1 1	1 0	0 0 0 0	a	b	c	1 1 1 1	1	1	d	e	f	g	h	Rd
		op								cmode	o2							

Encoding

```
FMOV  <Vd>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let rd : integer = UInt(Rd);

let datasize : integer{} = 64 << UInt(Q);
let imm8 : bits(8) = a::b::c::d::e::f::g::h;
let imm16 : bits(16) = imm8[7]::NOT(imm8[6])::Replicate{2}(imm8[6])::imm8[5:0]::Zeros{6};
let imm : bits(datasize) = Replicate{}(imm16);
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28 19	18	17	16	15 12	11	10	9	8	7	6	5	4 0
0	Q	op	0 1 1 1 1 0 0 0 0 0	a	b	c	1 1 1 1	0	1	d	e	f	g	h	Rd
							cmode	o2							

Encoding for the Single-precision variant

Applies when `(op == 0)`

```
FMOV  <Vd>.<T>, #<imm>
```

Encoding for the Double-precision variant

Applies when `(Q == 1 && op == 1)`

```
FMOV  <Vd>.2D, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if cmode::op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then EndOfDecode(Decode_UNDEF); end;
end;

let rd : integer = UInt(Rd);
let datasize : integer{} = 64 << UInt(Q);
let imm64 : bits(64) = AdvSIMDExpandImm(op, cmode, a::b::c::d::e::f::g::h);
let imm : bits(datasize) = Replicate{}(imm64);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm>

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in “a:b:c:d:e:f:g:h”. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in A64 floating-point instructions.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
V{datasize}(rd) = imm;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.156 FMOV (register)

Floating-point move register without conversion

This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	17	16 15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1 0 0 0	0	0 0	1 0 0 0	0	Rn		Rd	
M		S						opc						

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMOV  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMOV  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMOV  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = Zeros{};

result[0+:esize] = operand;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.157 FMOV (general)

Floating-point move to or from general-purpose register without conversion

This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	0 x	1 1	x	0 0 0 0 0 0		Rn		Rd	
		S					rmode		opcode						

Encoding for the Half-precision to 32-bit variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11 && rmode == 00 && opcode == 110)`

```
FMOV  <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11 && rmode == 00 && opcode == 110)`

```
FMOV  <Xd>, <Hn>
```

Encoding for the 32-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11 && rmode == 00 && opcode == 111)`

```
FMOV  <Hd>, <Wn>
```

Encoding for the 32-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)`

```
FMOV  <Sd>, <Wn>
```

Encoding for the Single-precision to 32-bit variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)`

```
FMOV  <Wd>, <Sn>
```

Encoding for the 64-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11 && rmode == 00 && opcode == 111)`

```
FMOV  <Hd>, <Xn>
```

Encoding for the 64-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)`

```
FMOV  <Dd>, <Xn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit to top half of 128-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)`

```
FMOV  <Vd>.D[1], <Xn>
```

Encoding for the Double-precision to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)`

```
FMOV  <Xd>, <Dn>
```

Encoding for the Top half of 128-bit to 64-bit variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 10 && rmode == 01 && opcode == 110)`

```
FMOV  <Xd>, <Vn>.D[1]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' && opcode[2:1]::rmode != '11 01' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = if ftype == '10' then 64 else (8 << UInt(ftype XOR '10'));
let part : integer = UInt(rmode[0]);
var op : FPConvOp;

case opcode[2:1]::rmode of
    when '11 00' =>        // FMOV
        if fltsize != 16 && fltsize != intsize then EndOfDecode(Decode_UNDEF); end;
        op = if opcode[0] == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    when '11 01' =>        // FMOV D[1]
        if intsize != 64 || ftype != '10' then EndOfDecode(Decode_UNDEF); end;
        op = if opcode[0] == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    otherwise =>
        unreachable;
end;
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);

case op of
    when FPConvOp_MOV_FtoI =>
        fltval = Vpart{fltsize}(n, part);
        X{intsize}(d) = ZeroExtend{intsize}(fltval);
    when FPConvOp_MOV_ItoF =>
        intval = X{intsize}(n);
        Vpart{fltsize}(d, part) = intval[fltsize-1:0];
    otherwise =>
        unreachable;
end;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.158 FMOV (scalar, immediate)

Floating-point move immediate (scalar)

This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

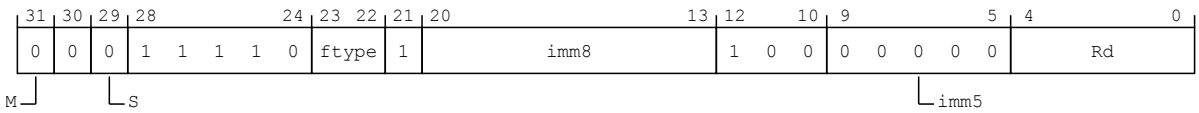

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMOV  <Hd>, #<imm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMOV  <Sd>, #<imm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMOV  <Dd>, #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);

let datasize : integer{} = 8 << UInt(ftype XOR '10');
let imm : bits(datasize) = VFPExpandImm{}(imm8);
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<imm>

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the “imm8” field. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in A64 floating-point instructions.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPEnabled();

V{datasize}(d) = imm;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.159 FMSUB

Floating-point fused multiply-subtract (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28 … 24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1 1 1 1 1	ftype	0	Rm	1	Ra	Rn	Rd
M		S			o1		o0			

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMSUB  <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMSUB  <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMSUB  <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ha>

Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Sa>

Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Da>

Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPEnabled();

let addend : bits(esize)   = V{esize}(a);
let operand1 : bits(esize) = FPNeg{esize}(V{esize}(n), FPCR());
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(a) else Zeros{128};

result[0+:esize] = FPMulAdd{esize}(addend, operand1, operand2, FPCR());
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.160 FMUL (by element)

Floating-point multiply (by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision, and Vector, single-precision and double-precision

Scalar, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	0	0	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U						size					opcode							

Encoding

```
FMUL  <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = esize;

let elements : integer = 1;
```

Scalar, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U											opcode							

Encoding

```
FMUL  <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	0 0	L	M	Rm	1 0 0 1	H	0	Rn	Rd
		U				size				opcode				

Encoding

```
FMUL  <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	1	sz	L	M	Rm	1 0 0 1	H	0	Rn	Rd
		U									opcode				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMUL  <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<V>

Is a width specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<Ts>

Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector, half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
var element1 : bits(esize);
let element2 : bits(esize) = operand2[index*:esize];

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(n) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    result[e*:esize] = FPMul{esize}(element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.161 FMUL (vector)

Floating-point multiply (vector)

This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 0	Rm	0 0	0 1 1	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMUL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1	Rm	1 1 0 1 1	1	Rn	Rd
		U								opcode			

Encoding

```
FMUL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPMul{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.162 FMUL (scalar)

Floating-point multiply (scalar)

This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20	16	15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1	Rm		0	0 0 0 1	0	Rn		Rd	
M		S							op						

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FMUL  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FMUL  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FMUL  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize]  = FPMul{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.163 FMULX (by element)

Floating-point multiply extended (by element)

This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision, and Vector, single-precision and double-precision

Scalar, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20	19–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	1	0	0	L	M	Rm	1 0 0 1	H	0	Rn	Rd
		U				size					opcode				

Encoding

```
FMULX  <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = esize;

let elements : integer = 1;
```

Scalar, single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20	19–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	1	1	sz	L	M	Rm	1 0 0 1	H	0	Rn	Rd
		U									opcode				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMULX  <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector, half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	0 0	L	M	Rm	1 0 0 1	H	0	Rn	Rd
		U				size				opcode				

Encoding

```
FMULX  <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let idxdsize : integer{} = 64 << UInt(H);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let index : integer = UInt(H::L::M);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector, single-precision and double-precision (FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	1	1	sz	L	M	Rm	1	0	0	1	H	0	Rn	Rd
		U														opcode				

Encoding

```
FMULX  <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
let Rmhi : bit = M;
case sz::L of
    when '0x' => index = UInt(H::L);
    when '10' => index = UInt(H);
    when '11' => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

For the “Scalar, half-precision” and “Vector, half-precision” variants: is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

For the “Scalar, single-precision and double-precision” and “Vector, single-precision and double-precision” variants: is the element index, encoded in “sz:L:H”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<Ts>

Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector, half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
var element1 : bits(esize);
let element2 : bits(esize) = operand2[index*:esize];

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(n) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    result[e*:esize] = FPMulX{esize}(element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.164 FMULX

Floating-point multiply extended

This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	0	0	1	0	Rm	0	0	0	1	1	1	Rn	Rd
		U						a						opcode					

Encoding

```
FMULX  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	0	0	sz	1	Rm	1	1	0	1	1	1	Rn	Rd
		U												opcode					

Encoding

```
FMULX  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 0	Rm	0 0	0 1 1	1	Rn	Rd
		U				a				opcode			

Encoding

```
FMULX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1	Rm	1 1 0 1 1	1	Rn	Rd
		U								opcode			

Encoding

```
FMULX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
if elements == 1 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(n) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPMulX{esize}(element1, element2, FPCR());
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.165 FNEG (vector)

Floating-point negate (vector)

This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn	Rd
		U						a							opcode								

Encoding

```
FNEG  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	1	1	0	Rn	Rd
		U													opcode								

Encoding

```
FNEG  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPNeg{esize}(element, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.166 FNEG (scalar)

Floating-point negate (scalar)

This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

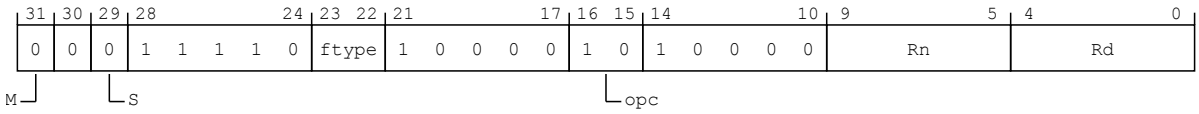

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FNEG  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FNEG  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FNEG  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPNeg{esize}(operand, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.167 FNMADD

Floating-point negated fused multiply-add (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

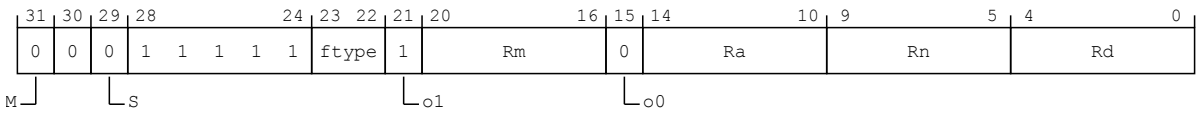

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FNMADD  <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FNMADD  <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FNMADD  <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ha>

Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Sa>

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Da>

Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPEnabled();

let addend : bits(esize)   = FPNeg{esize}(V{esize}(a), FPCR());
let operand1 : bits(esize) = FPNeg{esize}(V{esize}(n), FPCR());
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(a) else Zeros{128};

result[0+:esize] = FPMulAdd{esize}(addend, operand1, operand2, FPCR());
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.168 FNMSUB

Floating-point negated fused multiply-subtract (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28 … 24	23 22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	1 1 1 1 1	ftype	1	Rm	1	Ra	Rn	Rd
M		S			o1		o0			

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FNMSUB  <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FNMSUB  <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FNMSUB  <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ha>

Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Sa>

Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<Da>

Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPEnabled();

let addend : bits(esize)   = FPNeg{esize}(V{esize}(a), FPCR());
let operand1 : bits(esize) = V{esize}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(a) else Zeros{128};

result[0+:esize] = FPMulAdd{esize}(addend, operand1, operand2, FPCR());
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.169 FNMUL (scalar)

Floating-point multiply-negate (scalar)

This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20	16	15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1	Rm		1	0 0 0 1	0	Rn		Rd	
M		S							op						

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FNMUL  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FNMUL  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FNMUL  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

let product : bits(esize) = FPMul{}(operand1, operand2, FPCR());
result[0+:esize]  = FPNeg{esize}(product, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.170 FRECPE

Floating-point reciprocal estimate

This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U						a							opcode																

Encoding

```
FRECPE  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
FRECPE  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 1 1 1	0 0	1 1 1 0 1	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
FRECPE  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 1 0 1	1 0	Rn	Rd
		U							opcode			

Encoding

```
FRECPE  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if elements == 1 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;
let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPRecipEstimate{esize}(element, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.171 FRECPS

Floating-point reciprocal step

This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13–11	10	9–5	4–0
0	1	0	1	1 1 1	0	0	1	0	Rm	0	0	1 1 1	1	Rn	Rd
		U				a						opcode			

Encoding

```
FRECPS  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	0	0	sz	1	Rm	1 1 1 1 1	1	Rn	Rd
		U								opcode			

Encoding

```
FRECPS  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	0	1	0	Rm	0	0	1	1	1	1	Rn	Rd
		U						a						opcode					

Encoding

```
FRECPS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	0	0	sz	1	Rm	1	1	1	1	1	1	Rn	Rd
		U										opcode							

Encoding

```
FRECPS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
if elements == 1 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(n) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPRecipStepFused{esize}(element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.172 FRECPX

Floating-point reciprocal exponent (scalar)

This instruction finds an approximate reciprocal exponent for the source SIMD&FP register and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn					Rd				
		U						a							opcode																

Encoding

```
FRECPX  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 16;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
FRECPX  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
let operand : bits(esize) = V{}(n);

let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

result[0+:esize] = FPRecpX{esize}(operand, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.173 FRINT32X (vector)

Floating-point round to 32-bit integer, using current rounding mode (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision (FEAT_FRINTTS)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 1 1	0	1 0	Rn	Rd
		U								op			

Encoding

```
FRINT32X  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let intsize : integer{} = 32;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let rounding : FPRounding = FPRoundingMode(FPCR());
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.174 FRINT32X (scalar)

Floating-point round to 32-bit integer, using current rounding mode (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FRINTTS)

31	30	29	28	24	23	22	21	17	16	15	14	10	9	5	4	0
0	0	0	1 1 1 1 0		0	x	1 0 1 0 0		0	1	1 0 0 0 0		Rn		Rd	
M		S			ftype				op							

Encoding for the Single-precision variant

Applies when `(ftype == 00)`

```
FRINT32X  <Sd>, <Sn>
```

Encoding for the Double-precision variant

Applies when `(ftype == 01)`

```
FRINT32X  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if ftype IN {'1x'} then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(ftype[0]);
let intsize : integer{} = 32;
```

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundIntN{esize}(operand, FPCR(), rounding, intsize);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.175 FRINT32Z (vector)

Floating-point round to 32-bit integer toward zero (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_FRINTTS)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 1 1	0	1 0	Rn	Rd
		U								op			

Encoding

```
FRINT32Z  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let intsize : integer{} = 32;
let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.176 FRINT32Z (scalar)

Floating-point round to 32-bit integer toward zero (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FRINTTS)

31	30	29	28				24	23	22	21				17	16	15	14				10	9			5	4			0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	0	0	1	0	0	0	0	Rn				Rd			
M		S						ftype							op														

Encoding for the Single-precision variant

Applies when `(ftype == 00)`

```
FRINT32Z  <Sd>, <Sn>
```

Encoding for the Double-precision variant

Applies when `(ftype == 01)`

```
FRINT32Z  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if ftype IN {'1x'} then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(ftype[0]);
let intsize : integer{} = 32;

let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundIntN{esize}(operand, FPCR(), rounding, intsize);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.177 FRINT64X (vector)

Floating-point round to 64-bit integer, using current rounding mode (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision (FEAT_FRINTTS)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 1 1	1	1 0	Rn	Rd
		U								op			

Encoding

```
FRINT64X  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let intsize : integer{} = 64;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
let rounding : FPRounding = FPRoundingMode(FPCR());
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.178 FRINT64X (scalar)

Floating-point round to 64-bit integer, using current rounding mode (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FRINTTS)

31	30	29	28 … 24	23	22	21 … 17	16	15	14 … 10	9 … 5	4 … 0
0	0	0	1 1 1 1 0	0	x	1 0 1 0 0	1	1	1 0 0 0 0	Rn	Rd
M		S		ftype			op				

Encoding for the Single-precision variant

Applies when `(ftype == 00)`

```
FRINT64X  <Sd>, <Sn>
```

Encoding for the Double-precision variant

Applies when `(ftype == 01)`

```
FRINT64X  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if ftype IN {'1x'} then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(ftype[0]);
let intsize : integer{} = 64;
```

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundIntN{esize}(operand, FPCR(), rounding, intsize);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.179 FRINT64Z (vector)

Floating-point round to 64-bit integer toward zero (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision (FEAT_FRINTTS)

31	30	29	28	27	25	24	23	22	21	17	16	13	12	11	10	9	5	4	0
0	Q	0	0	1 1 1		0	0	sz	1 0 0 0 0		1 1 1 1		1	1	0	Rn		Rd	
		U											op						

Encoding

```
FRINT64Z  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let intsize : integer{} = 64;
let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.180 FRINT64Z (scalar)

Floating-point round to 64-bit integer toward zero (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FRINTTS)

31	30	29	28 … 24	23	22	21 … 17	16	15	14 … 10	9 … 5	4 … 0
0	0	0	1 1 1 1 0	0	x	1 0 1 0 0	1	0	1 0 0 0 0	Rn	Rd
M		S		ftype			op				

Encoding for the Single-precision variant

Applies when `(ftype == 00)`

```
FRINT64Z  <Sd>, <Sn>
```

Encoding for the Double-precision variant

Applies when `(ftype == 01)`

```
FRINT64Z  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FRINTTS) then EndOfDecode(Decode_UNDEF); end;
if ftype IN {'1x'} then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(ftype[0]);
let intsize : integer{} = 64;

let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundIntN{esize}(operand, FPCR(), rounding, intsize);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.181 FRINTA (vector)

Floating-point round to integral, to nearest with ties to away (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Encoding

```
FRINTA  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Encoding

```
FRINTA  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.182 FRINTA (scalar)

Floating-point round to integral, to nearest with ties to away (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	18	17	15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1 0 0	1	1 0	0	1 0 0 0	0	Rn		Rd	

M S rmode

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTA  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTA  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTA  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.183 FRINTI (vector)

Floating-point round to integral, using current rounding mode (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 1 1 1	0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTI  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTI  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);
let rounding : FPRounding = FPRoundingMode(FPCR());

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.184 FRINTI (scalar)

Floating-point round to integral, using current rounding mode (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

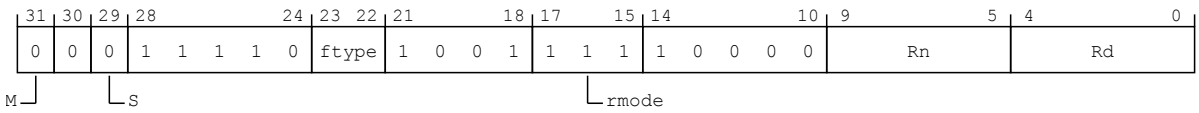

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTI  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTI  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTI  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.185 FRINTM (vector)

Floating-point round to integral, toward minus infinity (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 1 1 1	0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTM  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTM  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.186 FRINTM (scalar)

Floating-point round to integral, toward minus infinity (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	18	17	15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1 0 0	1	0 1	0	1 0 0 0	0	Rn		Rd	

M S rmode

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTM  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTM  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTM  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.187 FRINTN (vector)

Floating-point round to integral, to nearest with ties to even (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 1 1 1	0 0	1 1 0 0	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTN  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 0 0	0	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTN  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.188 FRINTN (scalar)

Floating-point round to integral, to nearest with ties to even (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21			18	17		15	14				10	9		5	4		0
0	0	0	1	1	1	1	0	ftype		1	0	0	1	0	0	0	1	0	0	0	0	Rn			Rd		
M		S														rmode											

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTN  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTN  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTN  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.189 FRINTP (vector)

Floating-point round to integral, toward plus infinity (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Encoding

```
FRINTP  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Encoding

```
FRINTP  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.190 FRINTP (scalar)

Floating-point round to integral, toward plus infinity (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	18	17	15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1 0 0	1	0 0	1	1 0 0 0	0	Rn		Rd	

M, S, rmode

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTP  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTP  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTP  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.191 FRINTX (vector)

Floating-point round to integral exact, using current rounding mode (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 1 1 1	0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTX  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = TRUE;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTX  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = TRUE;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);
let rounding : FPRounding = FPRoundingMode(FPCR());

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.192 FRINTX (scalar)

Floating-point round to integral exact, using current rounding mode (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21			18	17		15	14				10	9		5	4		0
0	0	0	1	1	1	1	0	ftype		1	0	0	1	1	1	0	1	0	0	0	0	Rn			Rd		
M		S														rmode											

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTX  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTX  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTX  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.193 FRINTZ (vector)

Floating-point round to integral, toward zero (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 1 1 1	0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTZ  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 0 0	1	1 0	Rn	Rd
		U				o2				o1			

Encoding

```
FRINTZ  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.194 FRINTZ (scalar)

Floating-point round to integral, toward zero (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

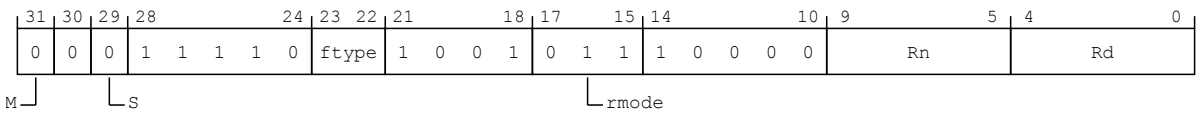

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTZ  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTZ  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTZ  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPRoundInt{esize}(operand, FPCR(), rounding, exact);

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.195 FRSQRTE

Floating-point reciprocal square root estimate

This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U						a							opcode																

Encoding

```
FRSQRTE  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
FRSQRTE  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 1 1 1	0 0	1 1 1 0 1	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
FRSQRTE  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 1 0 1	1 0	Rn	Rd
		U							opcode			

Encoding

```
FRSQRTE  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if elements == 1 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;
let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPRSqrtEstimate{esize}(element, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.196 FRSQRTS

Floating-point reciprocal square root step

This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27–25	24	23	22	21	20–16	15	14	13–11	10	9–5	4–0
0	1	0	1	1 1 1	0	1	1	0	Rm	0	0	1 1 1	1	Rn	Rd
		U				a						opcode			

Encoding

```
FRSQRTS  <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22	21	20–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	0	1	sz	1	Rm	1 1 1 1 1	1	Rn	Rd
		U								opcode			

Encoding

```
FRSQRTS  <V><d>, <V><n>, <V><m>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	1	0	Rm	0	0	1	1	1	1	Rn	Rd
		U						a						opcode					

Encoding

```
FRSQRTS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	1	sz	1	Rm	1	1	1	1	1	1	Rn	Rd
		U										opcode							

Encoding

```
FRSQRTS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
if elements == 1 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(n) else Zeros{128};

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = FPRSqrtStepFused{esize}(element1, element2, FPCR());
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.197 FSCALE

Floating-point adjust exponent by vector

This instruction multiplies the floating-point elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP8)

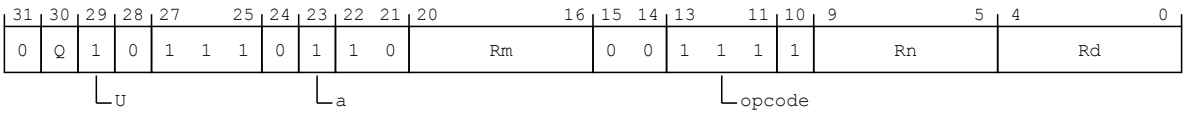

Encoding

```
FSCALE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_FP8)

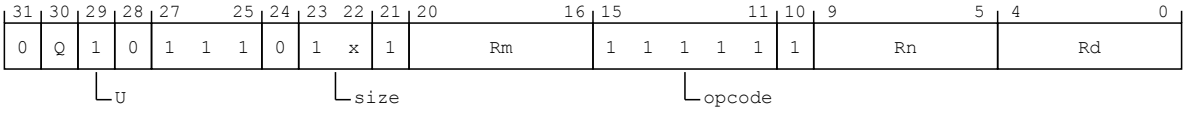

Encoding

```
FSCALE  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP8) then EndOfDecode(Decode_UNDEF); end;
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = if Q == '1' then 128 else 64;
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “size<0>:Q”:

size<0>	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    let op1 : bits(esize) = operand1[e*:esize];
    let op2 : integer     = SInt(operand2[e*:esize]);
    result[e*:esize] = FPScale{esize}(op1, op2, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.198 FSQRT (vector)

Floating-point square root (vector)

This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	1 1 1 1	0 0	1 1 1 1 1	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
FSQRT  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	1	sz	1 0 0 0 0	1 1 1 1 1	1 0	Rn	Rd
		U							opcode			

Encoding

```
FSQRT  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPSqrt{esize}(element, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.199 FSQRT (scalar)

Floating-point square root (scalar)

This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	17	16 15	14	10	9	5	4	0
0	0	0	1 1 1 1	0	ftype	1 0 0 0	0	1 1	1 0 0 0	0	Rn		Rd	
M		S						opc						

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FSQRT  <Hd>, <Hn>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FSQRT  <Sd>, <Sn>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FSQRT  <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let operand : bits(esize) = V{}(n);
var result : bits(128) = if IsMerging(FPCR()) then V{128}(d) else Zeros{128};

result[0+:esize] = FPSqrt{esize}(operand, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.200 FSUB (vector)

Floating-point subtract (vector)

This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 21	20 16	15 14	13 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	1 0	Rm	0 0	0 1 0	1	Rn	Rd
		U				a				opcode			

Encoding

```
FSUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1	sz	1	Rm	1 1 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
FSUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);

var element1 : bits(esize);
var element2 : bits(esize);
var result : bits(datasize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:esize] = FPSub{esize}(element1, element2, FPCR());
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.201 FSUB (scalar)

Floating-point subtract (scalar)

This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

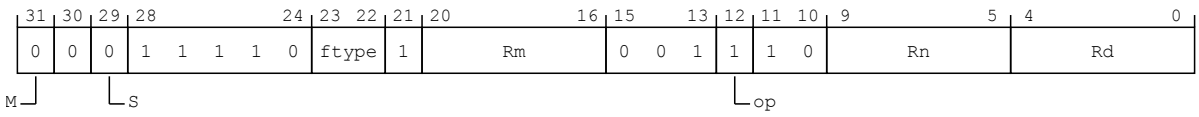

Encoding for the Half-precision variant

(FEAT_FP16) Applies when `(ftype == 11)`

```
FSUB  <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant

(FEAT_FP) Applies when `(ftype == 00)`

```
FSUB  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant

(FEAT_FP) Applies when `(ftype == 01)`

```
FSUB  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(ftype XOR '10');
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Hm>

Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPEnabled();
let operand1 : bits(esize) = V{}(n);
let operand2 : bits(esize) = V{}(m);

var result : bits(128) = if IsMerging(FPCR()) then V{128}(n) else Zeros{128};

result[0+:esize] = FPSub{esize}(operand1, operand2, FPCR());

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.202 INS (element)

Insert vector element from another vector element

This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element).

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 21	20 16	15	14 11	10	9 5	4 0
0	1	1	0	1 1 1	0 0	0 0	imm5	0	imm4	1	Rn	Rd
	Q	op										

Encoding

```
INS  <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[3:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let dst_index : integer = UInt(imm5[4:size+1]);
let src_index : integer = UInt(imm4[3:size]);
let idxdsize : integer{} = 64 << UInt(imm4[3]);
// imm4<size-1:0> is IGNORED
let esize : integer{} = 8 << size;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<Ts>
xx100	S
x1000	D

<index1>

Is the destination element index encoded in “imm5”:

imm5	<index1>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<index2>

Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	UInt(imm4)
xxx10	UInt(imm4<3:1>)
xx100	UInt(imm4<3:2>)
x1000	UInt(imm4<3>)

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.

Alias Conditions

Alias	Is preferred when
MOV (element)	Unconditionally

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(idxdsize) = V{}(n);
var result : bits(128);

result = V{128}(d);
result[dst_index*:esize] = operand[src_index*:esize];
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.203 INS (general)

Insert vector element from general-purpose register

This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general).

Advanced SIMD

(FEAT_AdvSIMD)

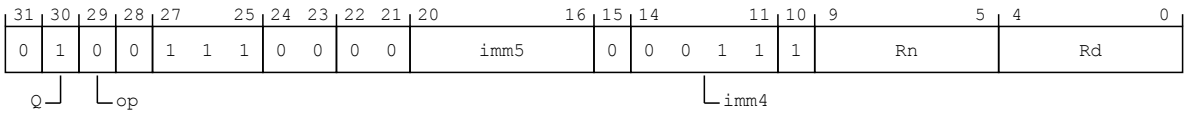

Encoding

```
INS  <Vd>.<Ts>[<index>], <R><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[3:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let index : integer = UInt(imm5[4:size+1]);
let esize : integer{} = 8 << size;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<R>

Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n>

Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
MOV (from general)	Unconditionally

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let element : bits(esize) = X{}(n);
var result : bits(128) = V{}(d);

result[index*:esize] = element;
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.204 LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers

This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29						23	22	21					16	15			12	11 10	9	5	4	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size	Rn		Rt	
									L									opcode						

Encoding for the One register variant

Applies when `(opcode == 0111)`

```
LD1  { <Vt>.<T> }, [<Xn|SP>]
```

Encoding for the Two registers variant

Applies when `(opcode == 1010)`

```
LD1  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Encoding for the Three registers variant

Applies when `(opcode == 0110)`

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Encoding for the Four registers variant

Applies when `(opcode == 0010)`

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

31	30	29						23	22	21	20			16	15			12	11	10	9			5	4			0
0	Q	0	0	1	1	0	0	1	1	0	Rm				x	x	1	x	size		Rn				Rt			
									L								opcode											

Encoding for the One register, immediate offset variant

Applies when `(Rm == 11111 && opcode == 0111)`

```
LD1  { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the One register, register offset variant

Applies when `(Rm != 11111 && opcode == 0111)`

```
LD1  { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Two registers, immediate offset variant

Applies when `(Rm == 11111 && opcode == 1010)`

```
LD1  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Two registers, register offset variant

Applies when `(Rm != 11111 && opcode == 1010)`

```
LD1  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Three registers, immediate offset variant

Applies when `(Rm == 11111 && opcode == 0110)`

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Three registers, register offset variant

Applies when `(Rm != 11111 && opcode == 0110)`

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Four registers, immediate offset variant

Applies when `(Rm == 11111 && opcode == 0010)`

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Four registers, register offset variant

Applies when `(Rm != 11111 && opcode == 0010)`

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<imm>

For the “One register, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<imm>
1	#16

For the “Two registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the “Three registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the “Four registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

var rpt : integer;                // number of iterations
let selem : integer = 1; // structure elements

case opcode of
    when '0010' => rpt = 4;    // LD/ST1 (4 registers)
    when '0110' => rpt = 3;    // LD/ST1 (3 registers)
    when '1010' => rpt = 2;    // LD/ST1 (2 registers)
    when '0111' => rpt = 1;    // LD/ST1 (1 register)
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            rval[e*:esize] = Mem{esize}(eaddr, accdesc);
            V{datasize}(tt) = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.205 LD1 (single structure)

Load one single-element structure to one lane of one register

This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

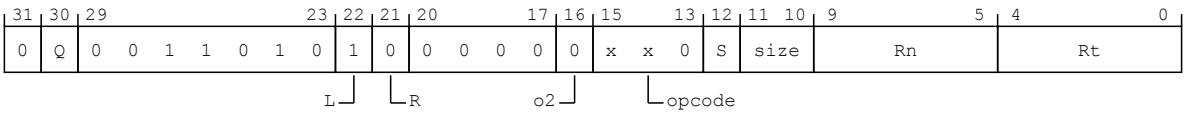

Encoding for the 8-bit variant

Applies when `(opcode == 000)`

```
LD1  { <Vt>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 010 && size == x0)`

```
LD1  { <Vt>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 100 && size == 00)`

```
LD1  { <Vt>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 100 && S == 0 && size == 01)`

```
LD1  { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

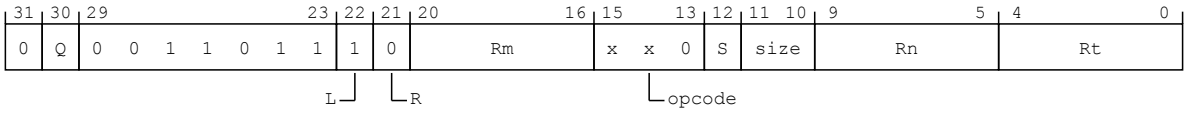

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 000)`

```
LD1  { <Vt>.B }[<index>], [<Xn|SP>], #1
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 000)`

```
LD1  { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD1  { <Vt>.D }[<index>], [<Xn|SP>], #8
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD1  { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
LD1  { <Vt>.H }[<index>], [<Xn|SP>], #2
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
LD1  { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
LD1  { <Vt>.S }[<index>], [<Xn|SP>], #4
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
LD1  { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.206 LD1R

Load one single-element structure and replicate to all lanes (of one register)

This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

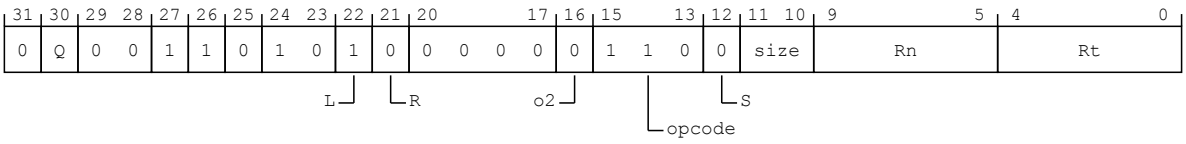

Encoding

```
LD1R  { <Vt>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

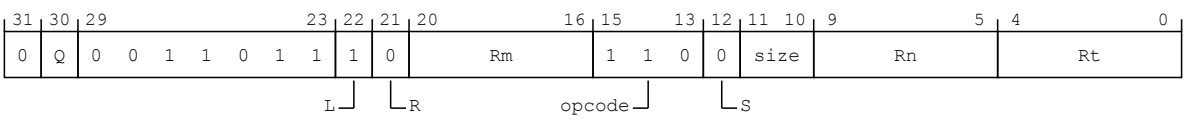

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD1R  { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD1R  { <Vt>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.207 LD2 (multiple structures)

Load multiple 2-element structures to two registers

This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29 28	27	26	25	24 23	22	21 16	15 12	11 10	9 5	4 0
0	Q	0 0	1	1	0	0 0	1	0 0 0 0 0 0	1 0 0 0	size	Rn	Rt
							L		opcode			

Encoding

```
LD2  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

31	30	29 23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0 0 1 1 0 0 1	1	0	Rm	1 0 0 0	size	Rn	Rt
			L			opcode			

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD2  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD2  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

let rpt : integer = 1;
let selem : integer = 2;

// .1D format only permitted with LD1 & ST1
if size::Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            rval[e*:esize] = Mem{esize}(eaddr, accdesc);
            V{datasize}(tt) = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.208 LD2 (single structure)

Load single 2-element structure to one lane of two registers

This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

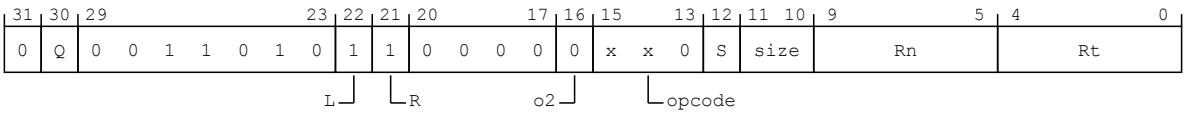

Encoding for the 8-bit variant

Applies when `(opcode == 000)`

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 010 && size == x0)`

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 100 && size == 00)`

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 100 && S == 0 && size == 01)`

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

31	30	29						23	22	21	20			16	15			13	12	11	10	9				5	4				0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x		0	S	size		Rn					Rt				
									L	R								opcode													

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 000)`

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 000)`

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.209 LD2R

Load single 2-element structure and replicate to all lanes of two registers

This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

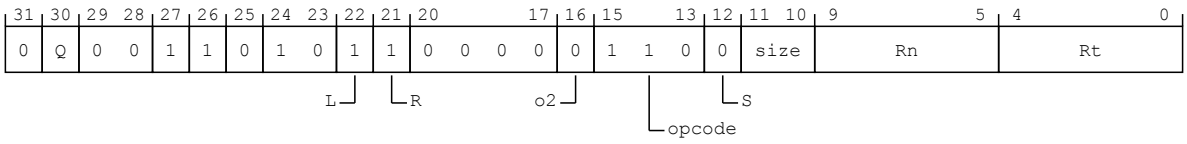

Encoding

```
LD2R  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

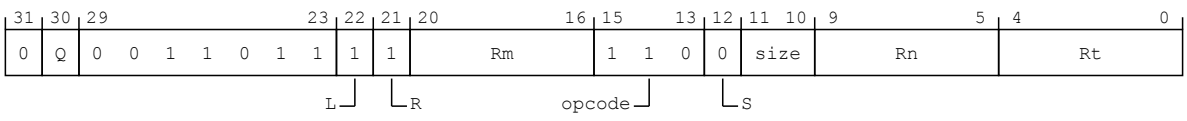

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD2R  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD2R  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#2
01	#4
10	#8
11	#16

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.210 LD3 (multiple structures)

Load multiple 3-element structures to three registers

This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

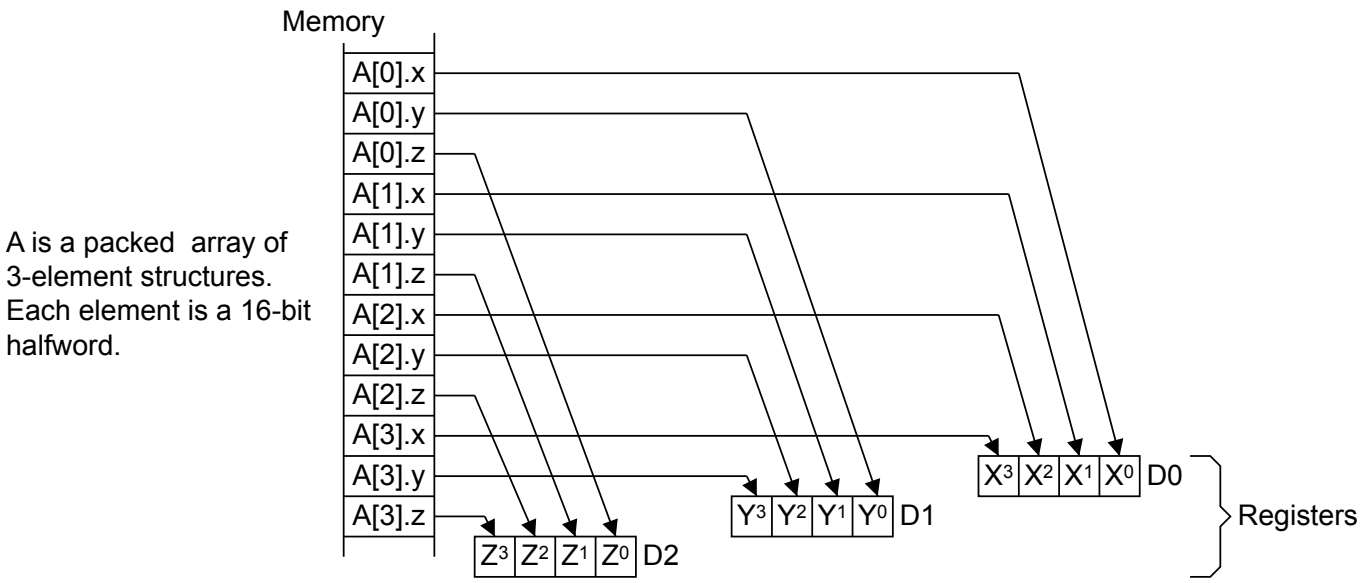

Figure C7-2 de-interleaving of an LD3.16 (multiple 3-element structures) instruction

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29 28	27	26	25	24 23	22	21 16	15 12	11 10	9 5	4 0
0	Q	0 0	1	1	0	0 0	1	0 0 0 0 0 0	0 1 0 0	size	Rn	Rt
							L		opcode			

Encoding

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

31	30	29						23	22	21	20			16	15			12	11	10	9			5	4			0
0	Q	0	0	1	1	0	0	1	1	0	Rm				0	1	0	0	size		Rn				Rt			
									L									opcode										

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

let rpt : integer = 1;
let selem : integer = 3;

// .1D format only permitted with LD1 & ST1
if size::Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            rval[e*:esize] = Mem{esize}(eaddr, accdesc);
            V{datasize}(tt) = rval;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.211 LD3 (single structure)

Load single 3-element structure to one lane of three registers

This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

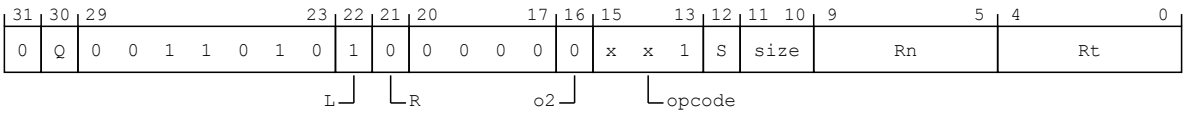

Encoding for the 8-bit variant

Applies when `(opcode == 001)`

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 011 && size == x0)`

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 101 && size == 00)`

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 101 && S == 0 && size == 01)`

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

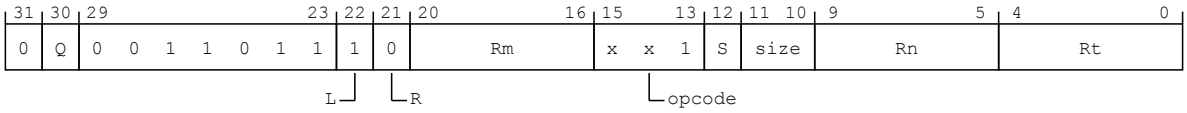

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 001)`

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 001)`

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.212 LD3R

Load single 3-element structure and replicate to all lanes of three registers

This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

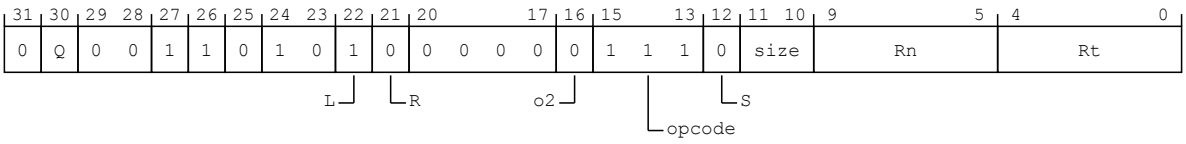

Encoding

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

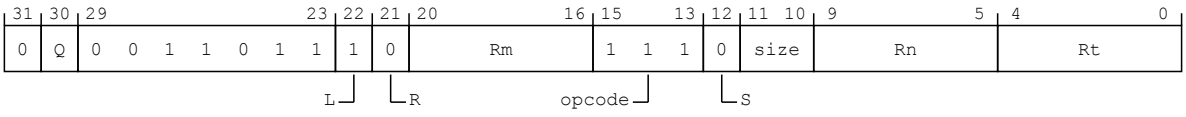

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#3
01	#6
10	#12
11	#24

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.213 LD4 (multiple structures)

Load multiple 4-element structures to four registers

This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

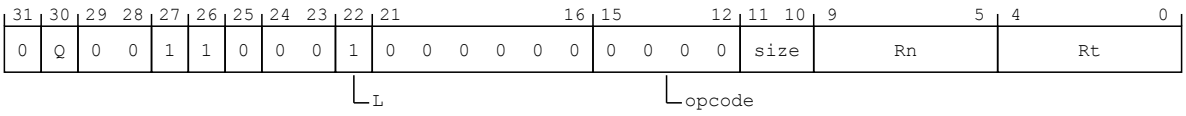

Encoding

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

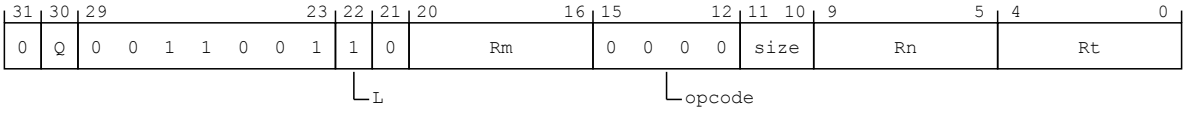

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<imm>
1	#64

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

let rpt : integer = 1;
let selem : integer = 4;

// .1D format only permitted with LD1 & ST1
if size::Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            rval[e*:esize] = Mem{esize}(eaddr, accdesc);
            V{datasize}(tt) = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.214 LD4 (single structure)

Load single 4-element structure to one lane of four registers

This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29						23	22	21	20				17	16	15			13	12	11	10	9			5	4			0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0		0	x	x		1	S	size		Rn				Rt			
									L	R						o2				opcode											

Encoding for the 8-bit variant

Applies when `(opcode == 001)`

```
LD4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 011 && size == x0)`

```
LD4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 101 && size == 00)`

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 101 && S == 0 && size == 01)`

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

31	30	29						23	22	21	20			16	15			13	12	11	10	9				5	4				0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x		1	S	size		Rn					Rt				
									L	R								opcode													

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 001)`

```
LD4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 001)`

```
LD4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
LD4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
LD4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.215 LD4R

Load single 4-element structure and replicate to all lanes of four registers

This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

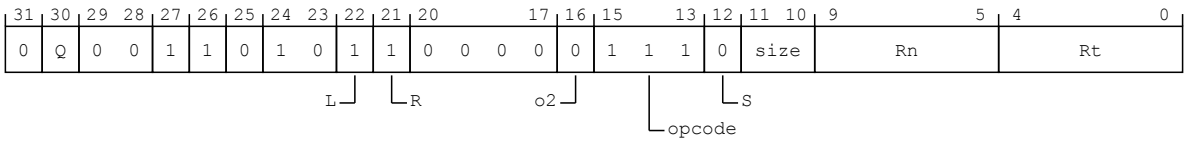

Encoding

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

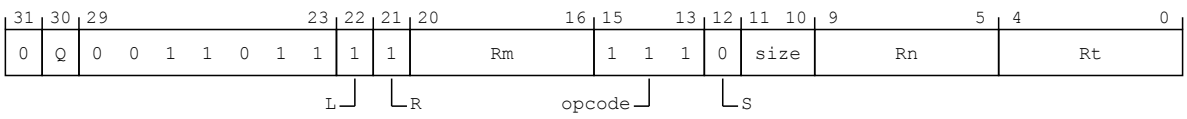

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<imm>
01	#8
10	#16
11	#32

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.216 LDAP1 (SIMD&FP)

Load-acquire RCpc one single-element structure to one lane of one register

This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

The instruction has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about addressing modes, see Load/Store addressing modes.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

64-bit

(FEAT_AdvSIMD && FEAT_LRCPC3)

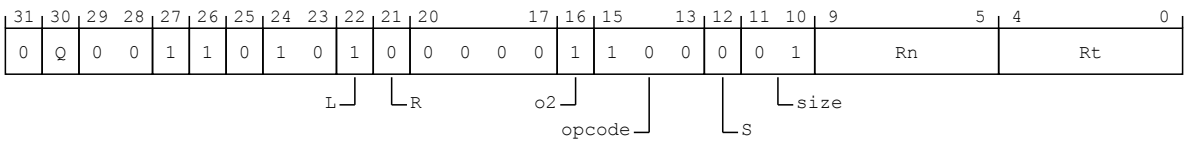

Encoding

```
LDAP1  { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_LRCPC3) then
    EndOfDecode(Decode_UNDEF);
end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<index>

Is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescASIMDAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        eaddr = AddressIncrement(address, offs, accdesc);
        rval[index*:esize] = Mem{esize}(eaddr, accdesc);
        V{128}(t) = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.217 LDAPUR (SIMD&FP)

Load-acquire RCpc SIMD&FP register (unscaled offset)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset

(FEAT_FP && FEAT_LRCPC3)

31 30	29 24	23 22	21	20 12	11 10	9 5	4 0
size	0 1 1 1 0 1	x 1	0	imm9	1 0	Rn	Rt
		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 01)`

```
LDAPUR  <Bt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 01)`

```
LDAPUR  <Ht>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 01)`

```
LDAPUR  <St>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 01)`

```
LDAPUR  <Dt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 11)`

```
LDAPUR  <Qt>, [<Xn|SP>{, #<simm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LRCPC3) then
    EndOfDecode(Decode_UNDEF);
end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescASIMDAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

V{datasize}(t) = Mem{datasize}(address, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.218 LDBFADD, LDBFADDA, LDBFADDAL, LDBFADDL

Atomic BFloat16 add

This instruction atomically loads a 16-bit value from memory, adds it to the BFloat16 value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDBFADDA` and `LDBFADDAL` load from memory with acquire semantics.
- `LDBFADDL` and `LDBFADDAL` store to memory with release semantics.
- `LDBFADD` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	0	1	1	1	1	0	0	A	R	1	Rs	0	0	0	0	0	0	Rn	Rt
size					VR							o3	opc						

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDBFADD  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDBFADDA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDBFADDAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDBFADDL  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 16;
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFADD, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.219 LDBFMAX, LDBFMAXA, LDBFMAXAL, LDBFMAXL

Atomic BFloat16 maximum

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 maximum with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDBFMAXA` and `LDBFMAXAL` load from memory with acquire semantics.
- `LDBFMAXL` and `LDBFMAXAL` store to memory with release semantics.
- `LDBFMAX` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	1	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		VR						o3	opc			

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDBFMAX  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDBFMAXA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDBFMAXAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDBFMAXL  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 16;
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMAX, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.220 LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, LDBFMAXNML

Atomic BFloat16 maximum number

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 maximum number with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDBFMAXNMA` and `LDBFMAXNMAL` load from memory with acquire semantics.
- `LDBFMAXNML` and `LDBFMAXNMAL` store to memory with release semantics.
- `LDBFMAXNM` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

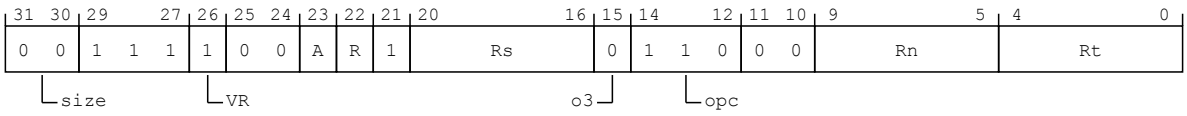

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDBFMAXNM  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDBFMAXNMA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDBFMAXNMAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDBFMAXNML  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 16;
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMAXNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.221 LDBFMIN, LDBFMINA, LDBFMINAL, LDBFMINL

Atomic BFloat16 minimum

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 minimum with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDBFMINA` and `LDBFMINAL` load from memory with acquire semantics.
- `LDBFMINL` and `LDBFMINAL` store to memory with release semantics.
- `LDBFMIN` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

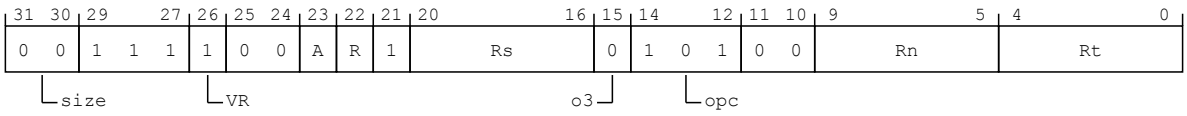

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDBFMIN  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDBFMINA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDBFMINAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDBFMINL  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 16;
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMIN, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.222 LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, LDBFMINNML

Atomic BFloat16 minimum number

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 minimum number with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDBFMINNMA` and `LDBFMINNMAL` load from memory with acquire semantics.
- `LDBFMINNML` and `LDBFMINNMAL` store to memory with release semantics.
- `LDBFMINNM` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

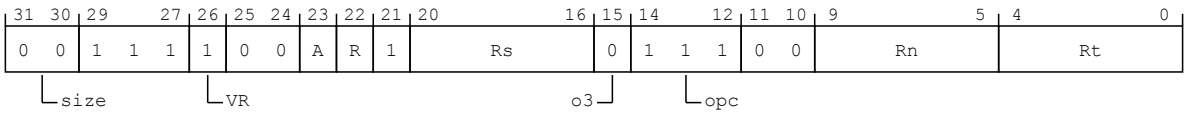

Encoding for the No memory ordering variant

Applies when `(A == 0 && R == 0)`

```
LDBFMINNM  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire variant

Applies when `(A == 1 && R == 0)`

```
LDBFMINNMA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Acquire-release variant

Applies when `(A == 1 && R == 1)`

```
LDBFMINNMAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(A == 0 && R == 1)`

```
LDBFMINNML  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 16;
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMINNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.223 LDFADD, LDFADDA, LDFADDAL, LDFADDL

Atomic floating-point add

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, performs a floating-point add with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDFADDA` and `LDFADDAL` load from memory with acquire semantics.
- `LDFADDL` and `LDFADDAL` store to memory with release semantics.
- `LDFADD` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

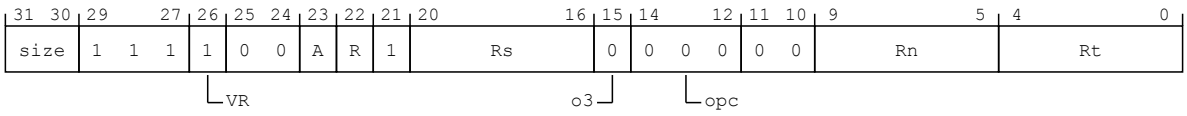

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && A == 0 && R == 0)`

```
LDFADD  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire variant

Applies when `(size == 01 && A == 1 && R == 0)`

```
LDFADDA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire-release variant

Applies when `(size == 01 && A == 1 && R == 1)`

```
LDFADDAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && A == 0 && R == 1)`

```
LDFADDL  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDFADD  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDFADDA  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDFADDAL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDFADDL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDFADD  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDFADDA  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDFADDAL  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDFADDL  <Ds>, <Dt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<St>

Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPADD, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.224 LDFMAX, LDFMAXA, LDFMAXAL, LDFMAXL

Atomic floating-point maximum

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point maximum with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDFMAXA` and `LDFMAXAL` load from memory with acquire semantics.
- `LDFMAXL` and `LDFMAXAL` store to memory with release semantics.
- `LDFMAX` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

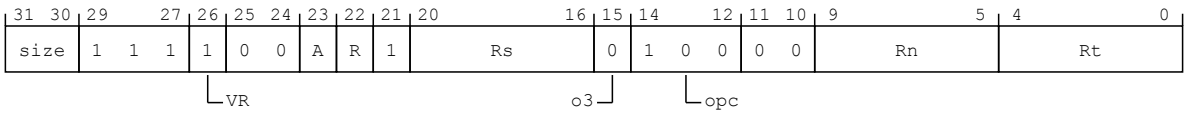

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && A == 0 && R == 0)`

```
LDFMAX  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire variant

Applies when `(size == 01 && A == 1 && R == 0)`

```
LDFMAXA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire-release variant

Applies when `(size == 01 && A == 1 && R == 1)`

```
LDFMAXAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && A == 0 && R == 1)`

```
LDFMAXL  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDFMAX  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDFMAXA  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDFMAXAL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDFMAXL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDFMAX  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDFMAXA  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDFMAXAL  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDFMAXL  <Ds>, <Dt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<St>

Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMAX, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.225 LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, LDFMAXNML

Atomic floating-point maximum number

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point maximum number with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDFMAXNMA` and `LDFMAXNMAL` load from memory with acquire semantics.
- `LDFMAXNML` and `LDFMAXNMAL` store to memory with release semantics.
- `LDFMAXNM` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
size	1 1 1	1	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
		VR						o3	opc			

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && A == 0 && R == 0)`

```
LDFMAXNM  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire variant

Applies when `(size == 01 && A == 1 && R == 0)`

```
LDFMAXNMA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire-release variant

Applies when `(size == 01 && A == 1 && R == 1)`

```
LDFMAXNMAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && A == 0 && R == 1)`

```
LDFMAXNML  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDFMAXNM  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDFMAXNMA  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDFMAXNMAL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDFMAXNML  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDFMAXNM  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDFMAXNMA  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDFMAXNMAL  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDFMAXNML  <Ds>, <Dt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<St>

Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMAXNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.226 LDFMIN, LDFMINA, LDFMINAL, LDFMINL

Atomic floating-point minimum

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point minimum with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDFMINA` and `LDFMINAL` load from memory with acquire semantics.
- `LDFMINL` and `LDFMINAL` store to memory with release semantics.
- `LDFMIN` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
size	1 1 1	1	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
		VR						o3	opc			

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && A == 0 && R == 0)`

```
LDFMIN  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire variant

Applies when `(size == 01 && A == 1 && R == 0)`

```
LDFMINA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire-release variant

Applies when `(size == 01 && A == 1 && R == 1)`

```
LDFMINAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && A == 0 && R == 1)`

```
LDFMINL  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDFMIN  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDFMINA  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDFMINAL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDFMINL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDFMIN  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDFMINA  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDFMINAL  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDFMINL  <Ds>, <Dt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<St>

Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMIN, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.227 LDFMINNM, LDFMINNMA, LDFMINNMAL, LDFMINNML

Atomic floating-point minimum number

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point minimum number with the value held in a register, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- `LDFMINNMA` and `LDFMINNMAL` load from memory with acquire semantics.
- `LDFMINNML` and `LDFMINNMAL` store to memory with release semantics.
- `LDFMINNM` has neither acquire nor release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

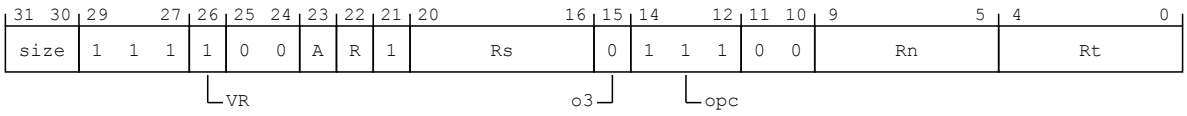

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && A == 0 && R == 0)`

```
LDFMINNM  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire variant

Applies when `(size == 01 && A == 1 && R == 0)`

```
LDFMINNMA  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision acquire-release variant

Applies when `(size == 01 && A == 1 && R == 1)`

```
LDFMINNMAL  <Hs>, <Ht>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && A == 0 && R == 1)`

```
LDFMINNML  <Hs>, <Ht>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && A == 0 && R == 0)`

```
LDFMINNM  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire variant

Applies when `(size == 10 && A == 1 && R == 0)`

```
LDFMINNMA  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision acquire-release variant

Applies when `(size == 10 && A == 1 && R == 1)`

```
LDFMINNMAL  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && A == 0 && R == 1)`

```
LDFMINNML  <Ss>, <St>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && A == 0 && R == 0)`

```
LDFMINNM  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire variant

Applies when `(size == 11 && A == 1 && R == 0)`

```
LDFMINNMA  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision acquire-release variant

Applies when `(size == 11 && A == 1 && R == 1)`

```
LDFMINNMAL  <Ds>, <Dt>, [<Xn|SP>]
```

Encoding for the Double-precision release variant

Applies when `(size == 11 && A == 0 && R == 1)`

```
LDFMINNML  <Ds>, <Dt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let s : integer = UInt(Rs);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = A == '1';
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<St>

Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMINNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);

V{datasize}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.228 LDNP (SIMD&FP)

Load pair of SIMD&FP registers, with non-temporal hint

This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Signed offset

(FEAT_FP)

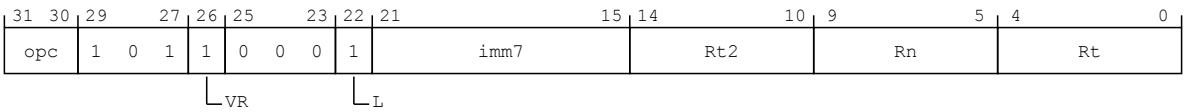

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDNP  <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDNP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
LDNP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP (SIMD&FP).

Assembler Symbols

<St1>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<St2>

Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<Dt1>

Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt2>

Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let scale : integer{} = 2 + (UInt(opc) as integer{0..2});
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = n != 31;
var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    V{datasize}(t) = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    V{datasize}(t2) = data[(datasize-1):0];
    V{datasize}(t)  = data[(2*datasize-1):datasize];
else
    V{datasize}(t)  = data[(datasize-1):0];
    V{datasize}(t2) = data[(2*datasize-1):datasize];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.229 LDP (SIMD&FP)

Load pair of SIMD&FP registers

This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_FP)

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	1	0 0 1	1	imm7	Rt2	Rn	Rt
		VR		L				

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <St1>, <St2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDP  <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
LDP  <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;

let wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_FP)

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	1	0 1 1	1	imm7	Rt2	Rn	Rt
		VR		L				

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <St1>, <St2>, [<Xn|SP>, #<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDP  <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
LDP  <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;

let wback : boolean = TRUE;
let postindex : boolean = FALSE;
```

Signed offset

(FEAT_FP)

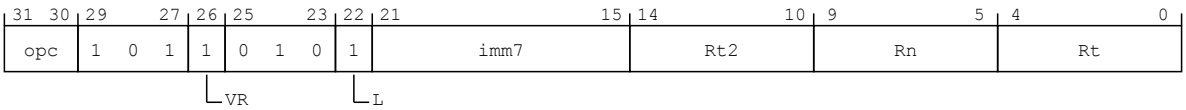

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDP  <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
LDP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;

let wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<St1>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<St2>

Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Post-index 128-bit” and “Pre-index 128-bit” variants: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “Signed offset 128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<Dt1>

Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt2>

Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + (UInt(opc) as integer{0..2});
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                    tagchecked, privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    V{datasize}(t) = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    V{datasize}(t2) = data[(datasize-1):0];
    V{datasize}(t)  = data[(2*datasize-1):datasize];
else
    V{datasize}(t)  = data[(datasize-1):0];
    V{datasize}(t2) = data[(2*datasize-1):datasize];
end;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.230 LDR (immediate, SIMD&FP)

Load SIMD&FP register (immediate offset)

This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 1	0	imm9	0 1	Rn	Rt
		VR		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 01)`

```
LDR  <Bt>, [<Xn|SP>], #<simm>
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 01)`

```
LDR  <Ht>, [<Xn|SP>], #<simm>
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 01)`

```
LDR  <St>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 01)`

```
LDR  <Dt>, [<Xn|SP>], #<simm>
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 11)`

```
LDR  <Qt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-index

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 1	0	imm9	1 1	Rn	Rt
		VR		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 01)`

```
LDR  <Bt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 01)`

```
LDR  <Ht>, [<Xn|SP>, #<simm>]!
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 01)`

```
LDR  <St>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 01)`

```
LDR  <Dt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 11)`

```
LDR  <Qt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Unsigned offset

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
size	1 1 1	1	0 1	x 1	imm12	Rn	Rt
		VR		opc			

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 01)`

```
LDR  <Bt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 01)`

```
LDR  <Ht>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 01)`

```
LDR  <St>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 01)`

```
LDR  <Dt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 11)`

```
LDR  <Qt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), scale);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<pimm>

For the “8-bit” variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “16-bit” variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

For the “32-bit” variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

For the “64-bit” variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

For the “128-bit” variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the “imm12” field as <pimm>/16.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

V{datasize}(t) = Mem{datasize}(address, accdesc);

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.231 LDR (literal, SIMD&FP)

Load SIMD&FP register (PC-relative literal)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Literal

(FEAT_FP)

31 30	29 27	26	25 24	23 5	4 0
opc	0 1 1	1	0 0	imm19	Rt

VR

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
LDR  <St>, <label>
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
LDR  <Dt>, <label>
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
LDR  <Qt>, <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
if opc == '11' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = 4 << (UInt(opc) as integer{0..2});
let nontemporal : boolean = FALSE;
let tagchecked : boolean = FALSE;

let offset : bits(64) = SignExtend{}(imm19::'00');
```

Assembler Symbols

<St>

Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<label>

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<Dt>

Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qt>

Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
let address : bits(64) = PC64() + offset;
AArch64_CheckFPEnabled();
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                    tagchecked, privileged);
let data : bits(size*8) = Mem{size*8}(address, accdesc);
V{size*8}(t) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.232 LDR (register, SIMD&FP)

Load SIMD&FP register (register offset)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

SIMD&FP registers

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 1	1	Rm	option	S	1 0	Rn	Rt
		VR		opc							

Encoding for the 8-bit with extended register offset variant

Applies when `(size == 00 && opc == 01 && option != 011)`

```
LDR  <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 8-bit with shifted register offset variant

Applies when `(size == 00 && opc == 01 && option == 011)`

```
LDR  <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 01)`

```
LDR  <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 01)`

```
LDR  <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 01)`

```
LDR  <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 11)`

```
LDR  <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;         // sub-word index
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

For the “8-bit with extended register offset” variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the “128-bit”, “16-bit”, “32-bit”, and “64-bit” variants: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the “8-bit with extended register offset” and “8-bit with shifted register offset” variants: is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

For the “16-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	<amount>
0	#0
1	#1

For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the “128-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
AArch64_CheckFPEnabled();
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

V{datasize}(t) = Mem{datasize}(address, accdesc);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.233 LDTNP (SIMD&FP)

Load unprivileged pair of SIMD&FP registers, with non-temporal hint

This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Signed offset

(FEAT_FP && FEAT_LSUI)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
1	1	1	0	1	1	0	0	0	1	imm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
LDTNP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let datasize : integer{} = 128;
let offset : bits(64) = LSL(SignExtend{64}(imm7), 4);
let tagchecked : boolean = n != 31;
var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

`LDTNP (SIMD&FP)` has the same CONSTRAINED UNPREDICTABLE behavior as `LDNP (SIMD&FP)`. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP (SIMD&FP).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    V{datasize}(t) = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    V{datasize}(t2) = data[(datasize-1):0];
    V{datasize}(t)  = data[(2*datasize-1):datasize];
else
    V{datasize}(t)  = data[(datasize-1):0];
    V{datasize}(t2) = data[(2*datasize-1):datasize];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.234 LDTP (SIMD&FP)

Load unprivileged pair of SIMD&FP registers

This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_FP && FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	1	0	0 1	1	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
LDTP  <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;

let wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_FP && FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	1	0	1 1	1	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
LDTP  <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;

let wback : boolean = TRUE;
let postindex : boolean = FALSE;
```

Signed offset

(FEAT_FP && FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	1	0	1 0	1	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
LDTP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;

let wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

`LDTP` has the same CONSTRAINED UNPREDICTABLE behavior as `LDP (SIMD&FP)`. See Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Assembler Symbols

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Load/store register pair (post-indexed)” and “Load/store register pair (pre-indexed)” variants: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.

For the “Load/store register pair (offset)” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let datasize : integer{} = 128;
let offset : bits(64) = LSL(SignExtend{64}(imm7), 4);
let tagchecked : boolean = wback || n != 31;

var rt_unknown : boolean = FALSE;

if t == t2 then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN =>    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF =>      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP =>        EndOfDecode(Decode_NOP);
    end;
end;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                    tagchecked, privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(2*datasize) = Mem{2*datasize}(address, accdesc);
if rt_unknown then
    V{datasize}(t) = ARBITRARY : bits(datasize);
elsif BigEndian(accdesc.acctype) then
    V{datasize}(t2) = data[(datasize-1):0];
    V{datasize}(t)  = data[(2*datasize-1):datasize];
else
    V{datasize}(t)  = data[(datasize-1):0];
    V{datasize}(t2) = data[(2*datasize-1):datasize];
end;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.235 LDUR (SIMD&FP)

Load SIMD&FP register (unscaled offset)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset

(FEAT_FP)

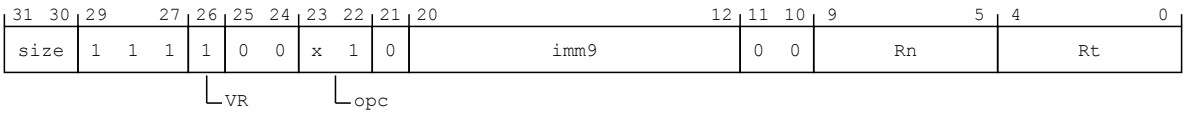

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 01)`

```
LDUR  <Bt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 01)`

```
LDUR  <Ht>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 01)`

```
LDUR  <St>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 01)`

```
LDUR  <Dt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 11)`

```
LDUR  <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

V{datasize}(t) = Mem{datasize}(address, accdesc);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.236 LUTI2

Lookup table read with 2-bit indices

This instruction copies indexed 8-bit or 16-bit elements from the table vector to the destination vector using packed 2-bit indices from a segment of the source vector. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index.

Advanced SIMD

(FEAT_AdvSIMD && FEAT_LUT)

31	30	29					24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	0	0	1	1	1	0	1	x	0	Rm				0	len		op	0	0	Rn				Rd			
	Q							op2																				

Encoding for the Byte variant

Applies when `(op2 == 10 && op == 1)`

```
LUTI2  <Vd>.16B, { <Vn>.16B }, <Vm>[<index>]
```

Encoding for the Halfword variant

Applies when `(op2 == 11)`

```
LUTI2  <Vd>.8H, { <Vn>.8H }, <Vm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_LUT) then
    EndOfDecode(Decode_UNDEF);
end;
if op2 == '10' && op == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let isize : integer{} = 2;
let esize : integer{} = if op2 == '10' then 8 else 16;
let part : integer = if op2 == '10' then UInt(len) else UInt(len::op);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP table register, encoded in the “Rn” field.

<Vm>

Is the name of the SIMD&FP source register, encoded in the “Rm” field.

<index>

For the “Byte” variant: is the vector segment index, in the range 0 to 3, encoded in the “len” field.

For the “Halfword” variant: is the vector segment index, in the range 0 to 7, encoded in the “len:op” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let elements : integer = 128 DIV esize;
let ibase : integer = elements * part;
let indices : bits(128) = V{}(m);
let table : bits(128)   = V{}(n);
var result : bits(128);

for e = 0 to elements-1 do
    let index : integer = UInt(indices[(ibase + e)*:isize]);
    result[e*:esize] = table[index*:esize];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.237 LUTI4

Lookup table read with 4-bit indices

This instruction copies indexed 8-bit or 16-bit elements from the one or two table vectors to the destination vector using packed 4-bit indices from a segment of the source vector. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index.

Advanced SIMD

(FEAT_AdvSIMD && FEAT_LUT)

31	30	29					24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	0	0	1	1	1	0	0	1	0	Rm		0	len		op	0	0	Rn		Rd	
	Q								op2													

Encoding for the Byte variant

Applies when `(len == x1 && op == 0)`

```
LUTI4  <Vd>.16B, { <Vn>.16B }, <Vm>[<index>]
```

Encoding for the Halfword variant

Applies when `(op == 1)`

```
LUTI4  <Vd>.8H, { <Vn1>.8H, <Vn2>.8H }, <Vm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_LUT) then
    EndOfDecode(Decode_UNDEF);
end;
if len[0] == '0' && op == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let isize : integer{} = 4;
let esize : integer{} = 8 << UInt(op);
let ntblr : integer = 1 << UInt(op);
let part : integer = if op == '0' then UInt(len[1]) else UInt(len);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP table register, encoded in the “Rn” field.

<Vm>

Is the name of the SIMD&FP source register, encoded in the “Rm” field.

<index>

For the “Byte” variant: is the vector segment index, in the range 0 to 1, encoded in the “len<1>” field.

For the “Halfword” variant: is the vector segment index, in the range 0 to 3, encoded in the “len” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn1>

Is the name of the first SIMD&FP table register, encoded in the “Rn” field.

<Vn2>

Is the name of the second SIMD&FP table register, encoded as “Rn” plus 1 modulo 32.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let elements : integer = 128 DIV esize;
let ibase : integer = elements * part;
let indices : bits(128) = V{}(m);
let table1 : bits(128)  = V{}(n+0);
let table2 : bits(128)  = if ntblr == 2 then V{128}((n+1) MOD 32) else Zeros{128};
var result : bits(128);
var res : bits(esize);

for e = 0 to elements-1 do
    let index : integer = UInt(indices[(ibase + e)*:isize]);
    if index < elements then
        res = table1[index*:esize];
    else
        assert ntblr == 2;
        res = table2[(index - elements)*:esize];
    end;
    result[e*:esize] = res;
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.238 MLA (by element)

Multiply-add to accumulator (vector, by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	size	L	M	Rm	0	0	0 0	H	0	Rn	Rd
		U									o2					

Encoding

```
MLA  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);    Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : bits(esize);

element2 = UInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    product = (element1 * element2)[esize-1:0];
    result[e*:esize] = operand3[e*:esize] + product;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.239 MLA (vector)

Multiply-add to accumulator (vector)

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

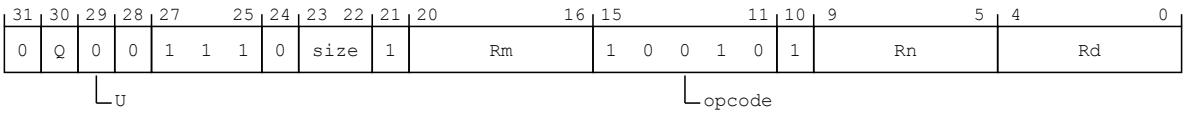

Encoding

```
MLA  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
var product : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    product = (UInt(element1) * UInt(element2))[esize-1:0];
    result[e*:esize] = operand3[e*:esize] + product;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.240 MLS (by element)

Multiply-subtract from accumulator (vector, by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	size	L	M	Rm	0	1	0 0	H	0	Rn	Rd
		U									o2					

Encoding

```
MLS  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);    Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : bits(esize);

element2 = UInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    product = (element1 * element2)[esize-1:0];
    result[e*:esize] = operand3[e*:esize] - product;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.241 MLS (vector)

Multiply-subtract from accumulator (vector)

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

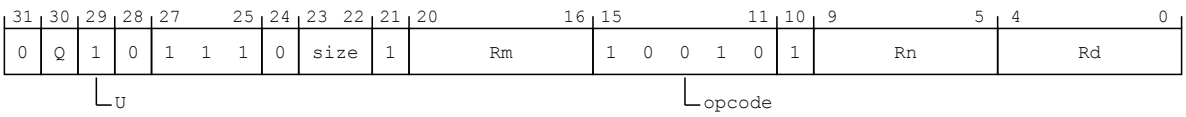

Encoding

```
MLS  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
var product : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    product = (UInt(element1) * UInt(element2))[esize-1:0];
    result[e*:esize] = operand3[e*:esize] - product;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.242 MOV (scalar)

Move vector element to scalar

This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of DUP (element). This means:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	0	0	0	0	imm5	0	0	0	0	0	1	Rn	Rd
		op														imm4			

Encoding

```
MOV  <V><d>, <Vn>.<T>[<index>]
```

is equivalent to

```
DUP  <V><d>, <Vn>.<T>[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<V>

Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index>

Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

Operational Information

The description of DUP (element) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.243 MOV (element)

Move vector element to another vector element

This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Advanced SIMD

(FEAT_AdvSIMD)

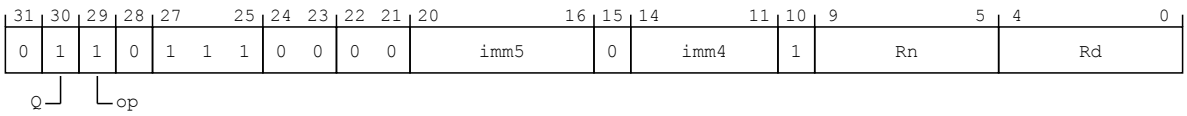

Encoding

```
MOV  <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

is equivalent to

```
INS  <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

and is always the preferred disassembly.

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index1>

Is the destination element index encoded in “imm5”:

imm5	<index1>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<index2>

Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	UInt(imm4)
xxx10	UInt(imm4<3:1>)
xx100	UInt(imm4<3:2>)
x1000	UInt(imm4<3>)

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.

Operational Information

The description of INS (element) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.244 MOV (from general)

Move general-purpose register to a vector element

This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (general). This means:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Advanced SIMD

(FEAT_AdvSIMD)

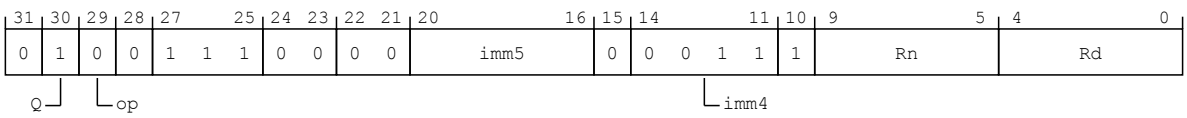

Encoding

```
MOV  <Vd>.<Ts>[<index>], <R><n>
```

is equivalent to

```
INS  <Vd>.<Ts>[<index>], <R><n>
```

and is always the preferred disassembly.

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<R>

Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n>

Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.

Operational Information

The description of INS (general) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.245 MOV (vector)

Move vector

This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Three registers of the same type

(FEAT_AdvSIMD)

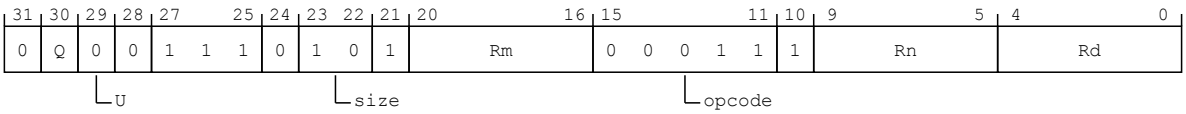

Encoding

```
MOV  <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
ORR  <Vd>.<T>, <Vn>.<T>, <Vn>.<T>
```

and is the preferred disassembly when `Rm == Rn`.

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (vector, register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.246 MOV (to general)

Move vector element to general-purpose register

This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28							21	20					16	15	14				11	10	9				5	4				0
0	Q	0	0	1	1	1	0	0	0	0	x	x	x	0	0		0	0	1	1	1		1	Rn					Rd				
		op												imm5						imm4													

Encoding for the 32-bit variant

Applies when `(Q == 0 && imm5 == xx100)`

```
MOV  <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV  <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

Encoding for the 64-bit variant

Applies when `(Q == 1 && imm5 == x1000)`

```
MOV  <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV  <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<index>

For the “32-bit” variant: is the element index encoded in “imm5<4:3>”.

For the “64-bit” variant: is the element index encoded in “imm5<4>”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational Information

The description of UMOV gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.247 MOVI

Move immediate (vector)

This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28								19	18	17	16	15			12	11	10	9	8	7	6	5	4			0
0	Q	op	0	1	1	1	1	0	0	0	0	a	b	c	cmode				0	1	d	e	f	g	h	Rd			
																			o2										

Encoding for the 8-bit variant

Applies when `(op == 0 && cmode == 1110)`

```
MOVI  <Vd>.<T>, #<imm8>{, LSL #0}
```

Encoding for the 16-bit shifted immediate variant

Applies when `(op == 0 && cmode == 10x0)`

```
MOVI  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifted immediate variant

Applies when `(op == 0 && cmode == 0xx0)`

```
MOVI  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifting ones variant

Applies when `(op == 0 && cmode == 110x)`

```
MOVI  <Vd>.<T>, #<imm8>, MSL #<amount>
```

Encoding for the 64-bit scalar variant

Applies when `(Q == 0 && op == 1 && cmode == 1110)`

```
MOVI  <Dd>, #<imm>
```

Encoding for the 64-bit vector variant

Applies when `(Q == 1 && op == 1 && cmode == 1110)`

```
MOVI  <Vd>.2D, #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let rd : integer = UInt(Rd);
let datasize : integer{} = 64 << UInt(Q);
let imm64 : bits(64) = AdvSIMDExpandImm(op, cmode, a::b::c::d::e::f::g::h);
let imm : bits(datasize) = Replicate{}(imm64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “8-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

For the “16-bit shifted immediate” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “32-bit shifted immediate” and “32-bit shifting ones” variants: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8>

Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<amount>

For the “16-bit shifted immediate” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit shifted immediate” variant: is the shift amount encoded in “cmode<2:1>”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the “32-bit shifting ones” variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<imm>

Is a 64-bit immediate ‘aaaaaaaabbbbbbbbcccccccddddddddeeeeeeeeffffffffggggggggghhhhhhhh’, encoded in “a:b:c:d:e:f:g:h”.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
V{datasize}(rd) = imm;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.248 MUL (by element)

Multiply (vector, by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

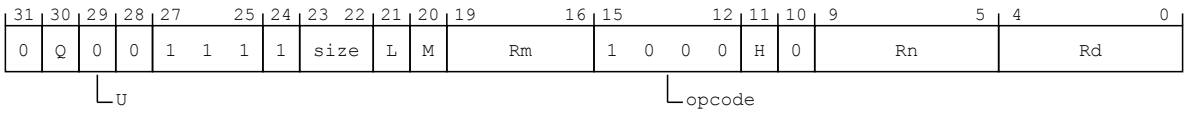

Encoding

```
MUL  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);    Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : bits(esize);

element2 = UInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    product = (element1 * element2)[esize-1:0];
    result[e*:esize] = product;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.249 MUL (vector)

Multiply (vector)

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0 0 1 1	1	Rn	Rd
		U							opcode			

Encoding

```
MUL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if U == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
var product : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    product = (UInt(element1) * UInt(element2))[esize-1:0];
    result[e*:esize] = product;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.250 MVN

Bitwise NOT (vector)

This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of NOT. This means:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0 0	1 0 0 0 0	0 0 1 0 1	1 0	Rn	Rd
		U				size		opcode			

Encoding

```
MVN  <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
NOT  <Vd>.<T>, <Vn>.<T>
```

and is always the preferred disassembly.

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

The description of NOT gives the operational pseudocode for this instruction.

Operational Information

The description of NOT gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.251 MVNI

Move inverted immediate (vector)

This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28									19	18	17	16	15			12	11	10	9	8	7	6	5	4				0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				0	1	d	e	f	g	h	Rd				
		op																		o2											

Encoding for the 16-bit shifted immediate variant

Applies when `(cmode == 10x0)`

```
MVNI  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifted immediate variant

Applies when `(cmode == 0xx0)`

```
MVNI  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifting ones variant

Applies when `(cmode == 110x)`

```
MVNI  <Vd>.<T>, #<imm8>, MSL #<amount>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let rd : integer = UInt(Rd);
let datasize : integer{} = 64 << UInt(Q);
let imm64 : bits(64) = AdvSIMDExpandImm(op, cmode, a::b::c::d::e::f::g::h);
let imm : bits(datasize) = Replicate{}(imm64);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “16-bit shifted immediate” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
1	8H

For the “32-bit shifted immediate” and “32-bit shifting ones” variants: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8>

Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<amount>

For the “16-bit shifted immediate” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit shifted immediate” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the “32-bit shifting ones” variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
V{datasize}(rd) = NOT(imm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.252 NEG (vector)

Negate (vector)

This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	1	1	1	0	Rn					Rd				
		U						size							opcode																

Encoding

```
NEG  D<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	1	0	1	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
NEG  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    element = -element;
    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.253 NOT

Bitwise NOT (vector)

This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0 0	1 0 0 0 0	0 0 1 0 1	1 0	Rn	Rd
		U				size		opcode			

Encoding

```
NOT  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 8;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
MVN	Unconditionally

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = NOT(element);
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.254 ORN (vector)

Bitwise inclusive OR NOT (vector)

This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

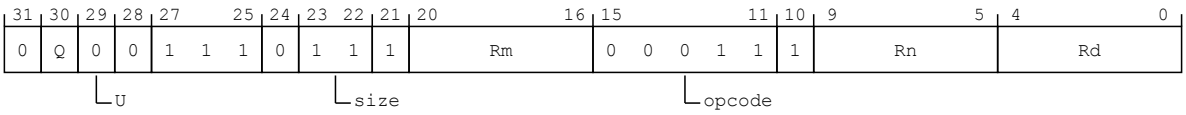

Encoding

```
ORN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = NOT(V{datasize}(m));
V{datasize}(d) = operand1 OR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.255 ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate)

This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Shifted immediate

(FEAT_AdvSIMD)

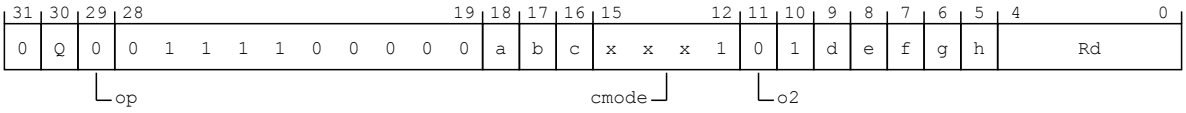

Encoding for the 16-bit variant

Applies when `(cmode == 10x1)`

```
ORR  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit variant

Applies when `(cmode == 0xx1)`

```
ORR  <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let rd : integer = UInt(Rd);
let datasize : integer{} = 64 << UInt(Q);
let imm64 : bits(64) = AdvSIMDExpandImm(op, cmode, a::b::c::d::e::f::g::h);
let imm : bits(datasize) = Replicate{}(imm64);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP register, encoded in the “Rd” field.

<T>

For the “16-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1	8H

For the “32-bit” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8>

Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<amount>

For the “16-bit” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(rd);
V{datasize}(rd) = operand OR imm;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.256 ORR (vector, register)

Bitwise inclusive OR (vector, register)

This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	1 0	1	Rm	0 0 0 1 1	1	Rn	Rd
		U				size			opcode			

Encoding

```
ORR  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
MOV (vector)	Rm == Rn

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
V{datasize}(d) = operand1 OR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.257 PMUL

Polynomial multiply

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see Polynomial arithmetic over {0, 1}.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	1 0 0 1 1	1	Rn	Rd
		U							opcode			

Encoding

```
PMUL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if U == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);
var product : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    product = PolynomialMult{esize, esize}(element1, element2)[esize-1:0];
    result[e*:esize] = product;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.258 PMULL, PMULL2

Polynomial multiply long

This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials, see Polynomial arithmetic over {0, 1}.

The `PMULL` instruction extracts each source vector from the lower half of each source register. The `PMULL2` instruction extracts each source vector from the upper half of each source register.

The `PMULL` and `PMULL2` variants that operate on 64-bit source elements are defined only when FEAT_PMULL is implemented.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 1 1 0	0 0	Rn	Rd
		U							opcode			

Encoding

```
PMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '01' || size == '10' then EndOfDecode(Decode_UNDEF); end;
if size == '11' && !IsFeatureImplemented(FEAT_PMULL) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	RESERVED
10	RESERVED
11	1Q

The ‘1Q’ arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	x	RESERVED
10	x	RESERVED
11	0	1D
11	1	2D

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:(2*esize)] = PolynomialMult{esize, esize}(element1, element2);
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.259 RADDHN, RADDHN2

Rounding add returning high narrow

This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN.

The `RADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `RADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 1	0	0	0 0	Rn	Rd
		U								o1				

Encoding

```
RADDHN{2}  <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
let round : boolean = TRUE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(2*datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:(2*esize)]);
    element2 = UInt(operand2[e*:(2*esize)]);
    sum = element1 + element2;
    sum = RShr(sum, esize, round);
    result[e*:esize] = sum[esize-1:0];
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.260 RAX1

Rotate and exclusive-OR

This instruction rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive-OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA3)

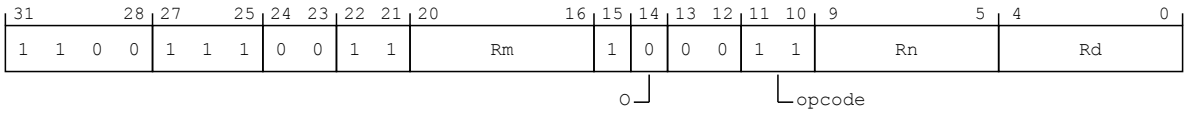

Encoding

```
RAX1  <Vd>.2D, <Vn>.2D, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
V{128}(d) = Vn XOR (ROL(Vm[127:64], 1)::ROL(Vm[63:0], 1));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.261 RBIT (vector)

Reverse bit order (vector)

This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

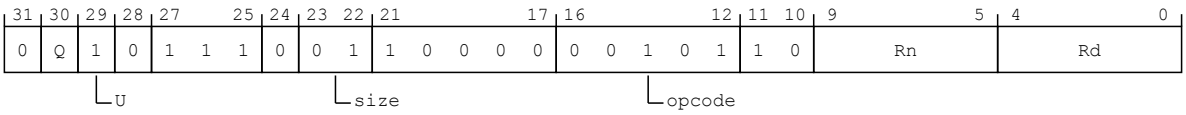

Encoding

```
RBIT  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 8;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(esize);
var rev : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    for i = 0 to esize-1 do
        rev[(esize-1)-i] = element[i];
    end;
    result[e*:esize] = rev;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.262 REV16 (vector)

Reverse elements in 16-bit halfwords (vector)

This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	0 0 0 0	1	1 0	Rn	Rd
		U							o0			

Encoding

```
REV16  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let csize : integer{} = 64 >> UInt(o0::U);
let esize : integer{} = 8 << UInt(size);
if csize <= esize then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 64 << UInt(Q);
let containers : integer = datasize DIV csize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
for c = 0 to containers-1 do
    let container : bits(csize) = operand[c*:csize];
    result[c*:csize] = Reverse{csize}(container, esize);
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.263 REV32 (vector)

Reverse elements in 32-bit words (vector)

This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1 0 0 0 0	0 0 0 0	0	1 0	Rn	Rd
		U							o0			

Encoding

```
REV32  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let csize : integer{} = 64 >> UInt(o0::U);
let esize : integer{} = 8 << UInt(size);
if csize <= esize then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 64 << UInt(Q);
let containers : integer = datasize DIV csize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
for c = 0 to containers-1 do
    let container : bits(csize) = operand[c*:csize];
    result[c*:csize] = Reverse{csize}(container, esize);
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.264 REV64

Reverse elements in 64-bit doublewords (vector)

This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	0 0 0 0	0	1 0	Rn	Rd
		U							o0			

Encoding

```
REV64  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let csize : integer{} = 64 >> UInt(o0::U);
let esize : integer{} = 8 << UInt(size);
if csize <= esize then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 64 << UInt(Q);
let containers : integer = datasize DIV csize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
for c = 0 to containers-1 do
    let container : bits(csize) = operand[c*:csize];
    result[c*:csize] = Reverse{csize}(container, esize);
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.265 RSHRN, RSHRN2

Rounding shift right narrow (immediate)

This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN.

The `RSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `RSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1 0	!=0000	immb	1 0 0 0	1	1	Rn	Rd
		U				immh			op			

Encoding

```
RSHRN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift>

Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = RShr(UInt(operand[e*:(2*esize)]), shift, round);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:esize] = element[esize-1:0];
end;

Vpart{datasize}(d, part) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.266 RSUBHN, RSUBHN2

Rounding subtract returning high narrow

This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN.

The `RSUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `RSUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 1	1	0	0 0	Rn	Rd
		U								o1				

Encoding

```
RSUBHN{2}  <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
let round : boolean = TRUE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(2*datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:(2*esize)]);
    element2 = UInt(operand2[e*:(2*esize)]);
    sum = element1 - element2;
    sum = RShr(sum, esize, round);
    result[e*:esize] = sum[esize-1:0];
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.267 SABA

Signed absolute difference and accumulate

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1 1 1	1	1	Rn	Rd
		U								ac			

Encoding

```
SABA  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize) = V{datasize}(d);
var element1 : integer;
var element2 : integer;
var absdiff : bits(esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.268 SABAL, SABAL2

Signed absolute difference and accumulate long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABAL` instruction extracts each source vector from the lower half of each source register. The `SABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

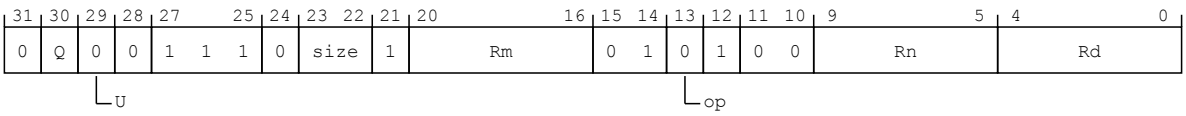

Encoding

```
SABAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize) = V{2*datasize}(d);
var element1 : integer;
var element2 : integer;
var absdiff : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    absdiff = Abs(element1-element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + absdiff;
end;
V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.269 SABD

Signed absolute difference

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1 1 1	0	1	Rn	Rd
		U								ac			

Encoding

```
SABD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize) = Zeros{datasize};
var element1 : integer;
var element2 : integer;
var absdiff : bits(esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.270 SABDL, SABDL2

Signed absolute difference long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABDL` instruction extracts each source vector from the lower half of each source register. The `SABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1	1	1	0 0	Rn	Rd
		U								op				

Encoding

```
SABDL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize) = Zeros{2*datasize};
var element1 : integer;
var element2 : integer;
var absdiff : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    absdiff = Abs(element1-element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + absdiff;
end;
V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.271 SADALP

Signed add and accumulate long pairwise

This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

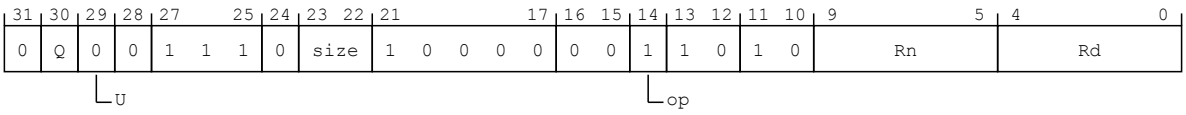

Encoding

```
SADALP  <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV (2 * esize);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize)  = V{}(d);

var sum : bits(2*esize);
var op1 : integer;
var op2 : integer;

for e = 0 to elements-1 do
    op1 = SInt(operand[(2*e+0)*:esize]);
    op2 = SInt(operand[(2*e+1)*:esize]);
    sum = (op1+op2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + sum;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.272 SADDL, SADDL2

Signed add long (vector)

This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SADDL` instruction extracts each source vector from the lower half of each source register. The `SADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0	0	0	0 0	Rn	Rd
		U								o1				

Encoding

```
SADDL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    sum = element1 + element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.273 SADDLP

Signed add long pairwise

This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 15	14	13 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	0 0	0	1 0	1 0	Rn	Rd
		U							op				

Encoding

```
SADDLP  <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV (2 * esize);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

var sum : bits(2*esize);
var op1 : integer;
var op2 : integer;

for e = 0 to elements-1 do
    op1 = SInt(operand[(2*e+0)*:esize]);
    op2 = SInt(operand[(2*e+1)*:esize]);
    sum = (op1+op2)[2*esize-1:0];
    result[e*:(2*esize)] = sum;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.274 SADDLV

Signed add long across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 1 0 0 0	0 0 0 1 1	1 0	Rn	Rd
		U						opcode			

Encoding

```
SADDLV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var opelt : bits(esize) = operand[0+:esize];
var sum : integer = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1 do
    opelt = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    sum = sum + element;
end;

V{2*esize}(d) = sum[2*esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.275 SADDW, SADDW2

Signed add wide

This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The `SADDW` instruction extracts the second source vector from the lower half of the second source register. The `SADDW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0	0	1	0 0	Rn	Rd
		U								o1				

Encoding

```
SADDW{2}  <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:(2*esize)]);
    element2 = SInt(operand2[e*:esize]);
    sum = element1 + element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.276 SCVTF (vector, fixed-point)

Signed fixed-point convert to floating-point (vector)

This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!=0000				immb			1	1	1	0	0	1	Rn					Rd				
		U							immh							opcode															

Encoding

```
SCVTF  <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = esize;
let elements : integer = 1;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!=0000				immb			1	1	1	0	0	1	Rn					Rd				
		U							immh							opcode															

Encoding

```
SCVTF  <Vd>.<T>, <Vn>.<T>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<fbits>

For the “Scalar” variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the “Vector” variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FixedToFP{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.277 SCVTF (vector, integer)

Signed integer convert to floating-point (vector)

This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

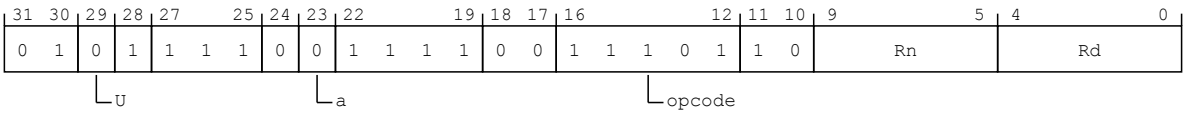

Encoding

```
SCVTF  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;

let unsigned : boolean = FALSE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

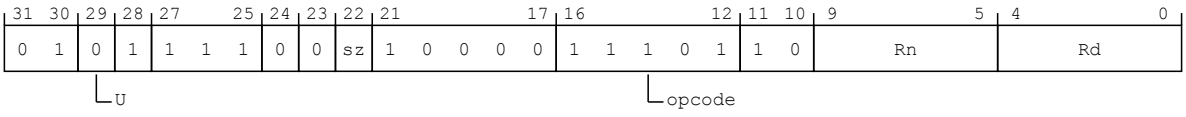

Encoding

```
SCVTF  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = FALSE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	1 1 1 1	0 0	1 1 1 0 1	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
SCVTF  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = FALSE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 1 0 1	1 0	Rn	Rd
		U							opcode			

Encoding

```
SCVTF  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let rounding : FPRounding = FPRoundingMode(FPCR());
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FixedToFP{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.278 SCVTF (scalar, fixed-point)

Signed fixed-point convert to floating-point (scalar)

This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	0	0 0	0 1	0	scale		Rn		Rd	
		S				rmode		opcode							

Encoding for the 32-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
SCVTF  <Hd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
SCVTF  <Hd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
SCVTF  <Sd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
SCVTF  <Sd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
SCVTF  <Dd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
SCVTF  <Dd>, <Xn>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && scale[5] == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let decode_fltsize : integer{} = 8 << UInt(ftype XOR '10');

let fracbits : integer = 64 - UInt(scale);

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<fbits>

For the “32-bit to double-precision”, “32-bit to half-precision”, and “32-bit to single-precision” variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus “scale”.

For the “64-bit to double-precision”, “64-bit to half-precision”, and “64-bit to single-precision” variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus “scale”.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Operation

```
AArch64_CheckFPEnabled();

let merge : boolean = IsMerging(FPCR());
let fltsize : integer{} = if merge then 128 else decode_fltsize;
var fltval : bits(fltsize) = if merge then V{fltsize}(d) else Zeros{fltsize};
let intval : bits(intsize) = X{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());

fltval[0+:decode_fltsize] = FixedToFP{decode_fltsize, intsize}(intval, fracbits, unsigned,
                                                               FPCR(), rounding);

V{fltsize}(d) = fltval;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.279 SCVTF (scalar, integer)

Signed integer convert to floating-point (scalar)

This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1	0	ftype	1	0 0	0 1	0	0 0 0 0 0	0	Rn		Rd	
		S					rmode		opcode						

Encoding for the 32-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
SCVTF  <Hd>, <Wn>
```

Encoding for the 32-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
SCVTF  <Sd>, <Wn>
```

Encoding for the 32-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
SCVTF  <Dd>, <Wn>
```

Encoding for the 64-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
SCVTF  <Hd>, <Xn>
```

Encoding for the 64-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
SCVTF  <Sd>, <Xn>
```

Encoding for the 64-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
SCVTF  <Dd>, <Xn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let decode_fltsize : integer{} = 8 << UInt(ftype XOR '10');
let unsigned : boolean    = FALSE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltsize : integer{} = if IsMerging(FPCR()) then 128 else decode_fltsize;

let intval : bits(intsize) = X{}(n);
let rounding : FPRounding  = FPRoundingMode(FPCR());
let fracbits : integer = 0;
var fltval : bits(fltsize) = if IsMerging(FPCR()) then V{fltsize}(d) else Zeros{fltsize};
fltval[0+:decode_fltsize] = FixedToFP{decode_fltsize, intsize}(intval, fracbits, unsigned,
                                                               FPCR(), rounding);

V{fltsize}(d) = fltval;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.280 SCVTF (scalar SIMD&FP)

Signed integer convert to floating-point (scalar SIMD&FP)

This instruction converts the signed integer value in the SIMD&FP source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Integer

(FEAT_FPRCVT)

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	1	1	1	0	0	0	0	0	0	0	0	Rn			Rd		
		S									rmode				opcode												

Encoding for the 32-bit to half-precision variant

Applies when `(sf == 0 && ftype == 11)`

```
SCVTF  <Hd>, <Sn>
```

Encoding for the 32-bit to double-precision variant

Applies when `(sf == 0 && ftype == 01)`

```
SCVTF  <Dd>, <Sn>
```

Encoding for the 64-bit to half-precision variant

Applies when `(sf == 1 && ftype == 11)`

```
SCVTF  <Hd>, <Dn>
```

Encoding for the 64-bit to single-precision variant

Applies when `(sf == 1 && ftype == 00)`

```
SCVTF  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
let rounding : FPRounding = FPRoundingMode(FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

intval = V{intsize}(n);
fltval = FixedToFP{fltsize, intsize}(intval, 0, FALSE, FPCR(), rounding);
result[0+:fltsize] = fltval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.281 SDOT (by element)

Dot product signed arithmetic (vector, by element)

This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	size	L	M	Rm	1 1 1 0	H	0	Rn	Rd
		U								opcode				

Encoding

```
SDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF); end;
if size  != '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(M::Rm);
let index : integer = UInt(H::L);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the element index, encoded in the “H:L” fields.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
var result : bits(datasize) = V{}(d);
for e = 0 to elements-1 do
    var res : integer = 0;
    var element1, element2 : integer;
    for i = 0 to 3 do
        element1 = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        element2 = SInt(operand2[(4 * index + i)*:(esize DIV 4)]);

        res = res + element1 * element2;
    end;
    result[e*:esize] = result[e*:esize] + res;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.282 SDOT (vector)

Dot product signed arithmetic (vector)

This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	0	Rm	1	0 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
SDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF); end;
if size != '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

result = V{datasize}(d);
for e = 0 to elements-1 do
    var res : integer = 0;
    var element1, element2 : integer;
    for i = 0 to 3 do
        element1 = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        element2 = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = result[e*:esize] + res;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.283 SHA1C

SHA1 hash update (choose).

Advanced SIMD

(FEAT_SHA1)

31	28	27	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0 1 0 1		1 1 1		0	0	0	0	Rm		0	0 0 0		0	0	Rn		Rd	
					size						opcode							

Encoding

```
SHA1C  <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var x : bits(128) = V{}(d);
var y : bits(32) = V{}(n);    // Note: 32 not 128 bits wide
let w : bits(128) = V{}(m);

for e = 0 to 3 do
    let t : bits(32) = SHAchoose(x[63:32], x[95:64], x[127:96]);
    y = y + ROL(x[31:0], 5) + t + w[e*:32];
    x[63:32] = ROL(x[63:32], 30);
    let yx : bits(160) = ROL(y::x, 32);
    (y, x) = (yx[159:128], yx[127:0]);
end;
V{128}(d) = x;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.284 SHA1H

SHA1 fixed rotate.

Advanced SIMD

(FEAT_SHA1)

31–28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0 1 0 1	1 1 1	0	0 0 (size)	1 0 1 0 0	0 0 0 0 0 (opcode)	1 0	Rn	Rd

Encoding

```
SHA1H  <Sd>, <Sn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand : bits(32) = V{}(n);        // read element [0] only,  [1-3] zeroed
V{32}(d) = ROL(operand, 30);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.285 SHA1M

SHA1 hash update (majority).

Advanced SIMD

(FEAT_SHA1)

31	28	27	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0 1 0 1		1 1 1		0	0	0	0	Rm		0	0 1 0		0	0	Rn		Rd	
					size						opcode							

Encoding

```
SHA1M  <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var x : bits(128) = V{}(d);
var y : bits(32) = V{}(n);    // Note: 32 not 128 bits wide
let w : bits(128) = V{}(m);

for e = 0 to 3 do
    let t : bits(32) = SHAmajority{32}(x[63:32], x[95:64], x[127:96]);
    y = y + ROL(x[31:0], 5) + t + w[e*:32];
    x[63:32] = ROL(x[63:32], 30);
    let yx : bits(160) = ROL(y::x, 32);
    (y, x) = (yx[159:128], yx[127:0]);
end;
V{128}(d) = x;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.286 SHA1P

SHA1 hash update (parity).

Advanced SIMD

(FEAT_SHA1)

31–28	27–25	24	23 22	21	20–16	15	14–12	11 10	9–5	4–0
0 1 0 1	1 1 1	0	0 0 (size)	0	Rm	0	0 0 1 (opcode)	0 0	Rn	Rd

Encoding

```
SHA1P  <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var x : bits(128) = V{}(d);
var y : bits(32) = V{}(n);    // Note: 32 not 128 bits wide
let w : bits(128) = V{}(m);

for e = 0 to 3 do
    let t : bits(32) = SHAparity(x[63:32], x[95:64], x[127:96]);
    y = y + ROL(x[31:0], 5) + t + w[e*:32];
    x[63:32] = ROL(x[63:32], 30);
    let yx : bits(160) = ROL(y::x, 32);
    (y, x) = (yx[159:128], yx[127:0]);
end;
V{128}(d) = x;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.287 SHA1SU0

SHA1 schedule update 0.

Advanced SIMD

(FEAT_SHA1)

31–28	27–25	24	23–22	21	20–16	15	14–12	11–10	9–5	4–0
0 1 0 1	1 1 1	0	0 0 (size)	0	Rm	0	0 1 1 (opcode)	0 0	Rn	Rd

Encoding

```
SHA1SU0  <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(d);
let operand2 : bits(128) = V{}(n);
let operand3 : bits(128) = V{}(m);
var result : bits(128) = operand2[63:0] :: operand1[127:64];
result = result XOR operand1 XOR operand3;
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.288 SHA1SU1

SHA1 schedule update 1.

Advanced SIMD

(FEAT_SHA1)

31	28	27	25	24	23	22	21	17	16	12	11	10	9	5	4	0
0 1 0 1		1 1 1		0	0	0	1 0 1 0 0		0 0 0 0 1		1	0	Rn		Rd	
					size				opcode							

Encoding

```
SHA1SU1  <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(d);
let operand2 : bits(128) = V{}(n);
let T : bits(128) = operand1 XOR LSR(operand2, 32);

var result : bits(128);
result[31:0]   = ROL(T[31:0],   1);
result[63:32]  = ROL(T[63:32],  1);
result[95:64]  = ROL(T[95:64],  1);
result[127:96] = ROL(T[127:96], 1) XOR ROL(T[31:0], 2);
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.289 SHA256H2

SHA-256 hash update (part 2).

Advanced SIMD

(FEAT_SHA256)

31–28	27–25	24	23–22	21	20–16	15	14–13	12	11–10	9–5	4–0
0 1 0 1	1 1 1	0	0 0	0	Rm	0	1 0	1	0 0	Rn	Rd
			size					P			

Encoding

```
SHA256H2  <Qd>, <Qn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Qn>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let part1 : boolean = FALSE;
V{128}(d) = SHA256hash(V{128}(n), V{128}(d), V{128}(m), part1);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.290 SHA256H

SHA-256 hash update (part 1).

Advanced SIMD

(FEAT_SHA256)

31			28	27			25	24	23	22	21	20			16	15	14	13	12	11	10	9			5	4			0
0	1	0	1	1	1	1	0	0	0		0	Rm				0	1	0	0	0	0	Rn				Rd			
									size										P										

Encoding

```
SHA256H  <Qd>, <Qn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Qn>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let part1 : boolean = TRUE;
V{128}(d) = SHA256hash(V{128}(d), V{128}(n), V{128}(m), part1);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.291 SHA256SU0

SHA-256 schedule update 0.

Advanced SIMD

(FEAT_SHA256)

31	28	27	25	24	23	22	21	17	16	12	11	10	9	5	4	0
0 1 0 1		1 1 1		0	0	0	1 0 1 0 0		0 0 0 1 0		1	0	Rn		Rd	
					size				opcode							

Encoding

```
SHA256SU0  <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(d);
let operand2 : bits(128) = V{}(n);
let T : bits(128) = operand2[31:0] :: operand1[127:32];
var result : bits(128);
var elt : bits(32);

for e = 0 to 3 do
    elt = T[e*:32];
    elt = ROR(elt, 7) XOR ROR(elt, 18) XOR LSR(elt, 3);
    result[e*:32] = elt + operand1[e*:32];
end;
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.292 SHA256SU1

SHA-256 schedule update 1.

Advanced SIMD

(FEAT_SHA256)

31	28	27	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0 1 0 1		1 1 1		0	0	0	0	Rm		0	1 1 0		0	0	Rn		Rd	
					size						opcode							

Encoding

```
SHA256SU1  <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(128) = V{}(d);
let operand2 : bits(128) = V{}(n);
let operand3 : bits(128) = V{}(m);
let T0 : bits(128) = operand3[31:0] :: operand2[127:32];

var T1 : bits(64);
var elt : bits(32);
var result : bits(128);

T1 = operand3[127:64];
for e = 0 to 1 do
    elt = T1[e*:32];
    elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
    elt = elt + operand1[e*:32] + T0[e*:32];
    result[e*:32] = elt;
end;

T1 = result[63:0];
for e = 2 to 3 do
    elt = T1[(e - 2)*:32];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
    elt = elt + operand1[e*:32] + T0[e*:32];
    result[e*:32] = elt;
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.293 SHA512H2

SHA-512 hash update part 2

This instruction takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA-512 calculation. It returns this value to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA512)

31 28	27 25	24 23	22 21	20 16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1	0 0	1 1	Rm	1	0	0 0	0 1	Rn	Rd
						O		opcode		

Encoding

```
SHA512H2  <Qd>, <Qn>, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Qn>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var Vtmp : bits(128);
var NSigma0 : bits(64);
let x : bits(128) = V{}(n);
let y : bits(128) = V{}(m);
let w : bits(128) = V{}(d);

NSigma0 =  ROR(y[63:0], 28) XOR ROR(y[63:0], 34) XOR ROR(y[63:0], 39);
Vtmp[127:64] = SHAmajority{64}(x[63:0], y[127:64], y[63:0]);
Vtmp[127:64] = (Vtmp[127:64] + NSigma0 +  w[127:64]);
NSigma0 =  ROR(Vtmp[127:64], 28) XOR ROR(Vtmp[127:64], 34) XOR ROR(Vtmp[127:64], 39);
Vtmp[63:0] = SHAmajority{64}(Vtmp[127:64], y[63:0], y[127:64]);
Vtmp[63:0] =   (Vtmp[63:0] + NSigma0 + w[63:0]);

V{128}(d) = Vtmp;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.294 SHA512H

SHA-512 hash update part 1

This instruction takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA-512 calculation. It returns this value to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA512)

31 28	27 25	24 23	22 21	20 16	15	14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1	0 0	1 1	Rm	1	0	0 0	0 0	Rn	Rd
						O		opcode		

Encoding

```
SHA512H  <Qd>, <Qn>, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Qn>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var Vtmp : bits(128);
var MSigma1 : bits(64);
var tmp : bits(64);
let x : bits(128) = V{}(n);
let y : bits(128) = V{}(m);
let w : bits(128) = V{}(d);

MSigma1 =  ROR(y[127:64], 14) XOR ROR(y[127:64], 18) XOR ROR(y[127:64], 41);
Vtmp[127:64] =  (y[127:64] AND x[63:0]) XOR (NOT(y[127:64]) AND x[127:64]);
Vtmp[127:64] = (Vtmp[127:64] + MSigma1 +  w[127:64]);
tmp = Vtmp[127:64] + y[63:0];
MSigma1 = ROR(tmp, 14) XOR ROR(tmp, 18) XOR ROR(tmp, 41);
Vtmp[63:0] = (tmp AND y[127:64]) XOR (NOT(tmp) AND x[63:0]);
Vtmp[63:0] = (Vtmp[63:0] + MSigma1 + w[63:0]);
V{128}(d) =  Vtmp;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.295 SHA512SU0

SHA-512 schedule update 0

This instruction takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA-512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA512)

31–28	27–25	24–23	22–19	18–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 1	1 0 0 0	0 0 0 1 0 0 0	0 0 (opcode)	Rn	Rd

Encoding

```
SHA512SU0  <Vd>.2D, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var sig0 : bits(64);
var Vtmp : bits(128);
let x : bits(128) = V{}(n);
let w : bits(128) = V{}(d);
sig0 = ROR(w[127:64], 1) XOR ROR(w[127:64], 8) XOR ('0000000'::w[127:71]);
Vtmp[63:0] = w[63:0] + sig0;
sig0 = ROR(x[63:0], 1) XOR ROR(x[63:0], 8) XOR ('0000000'::x[63:7]);
Vtmp[127:64] = w[127:64] + sig0;
V{128}(d) = Vtmp;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.296 SHA512SU1

SHA-512 schedule update 1

This instruction takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA-512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA512)

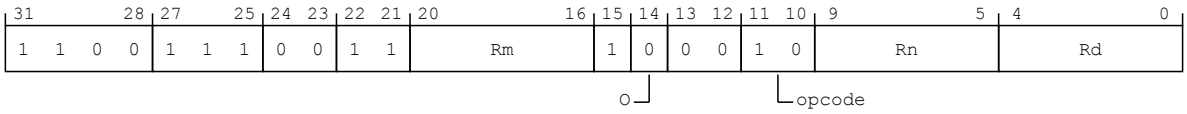

Encoding

```
SHA512SU1  <Vd>.2D, <Vn>.2D, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var sig1 : bits(64);
var Vtmp : bits(128);
let x : bits(128) = V{}(n);
let y : bits(128) = V{}(m);
let w : bits(128) = V{}(d);

sig1 = ROR(x[127:64], 19) XOR ROR(x[127:64], 61) XOR ('000000'::x[127:70]);
Vtmp[127:64] = w[127:64] + sig1 + y[127:64];
sig1 = ROR(x[63:0], 19) XOR ROR(x[63:0], 61) XOR ('000000'::x[63:6]);
Vtmp[63:0] = w[63:0] + sig1 + y[63:0];
V{128}(d) = Vtmp;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.297 SHADD

Signed halving add

This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SRHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 0 0 0	1	Rn	Rd
		U							opcode			

Encoding

```
SHADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    sum = (element1 + element2) >> 1;
    result[e*:esize] = sum[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.298 SHL

Shift left (immediate)

This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	1 x x x	immb	0 1 0 1 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
SHL  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;
let shift : integer = UInt(immh::immb) - esize;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	0 1 0 1 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
SHL  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let shift : integer = UInt(immh::immb) - esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the left shift amount, in the range 0 to 63, encoded as `UInt("immh:immb") - 64`.

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	**Q**	**<T>**

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

for e = 0 to elements-1 do
    result[e*:esize] = LSL(operand[e*:esize], shift);
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.299 SHLL, SHLL2

Shift left long (by element size)

This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SHLL` instruction extracts vector elements from the lower half of the source register. The `SHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1 0 0 0 0	1 0 0 1 1	1 0	Rn	Rd
		U						opcode			

Encoding

```
SHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<shift>

Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

size	<shift>
00	8
01	16
10	32
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = Vpart{}(n, part);
var result : bits(2*datasize);
var element : integer;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]) << shift;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:(2*esize)] = element[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.300 SHRN, SHRN2

Shift right narrow (immediate)

This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see RSHRN.

The `SHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1 0	!=0000	immb	1 0 0 0	0	1	Rn	Rd
		U				immh			op			

Encoding

```
SHRN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift>

Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    element = RShr(UInt(operand[e*:(2*esize)]), shift, round);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:esize] = element[esize-1:0];
end;

Vpart{datasize}(d, part) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.301 SHSUB

Signed halving subtract

This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 1 0 0	1	Rn	Rd
		U							opcode			

Encoding

```
SHSUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var diff : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    diff = (element1 - element2) >> 1;
    result[e*:esize] = diff[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.302 SLI

Shift left and insert (immediate)

This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

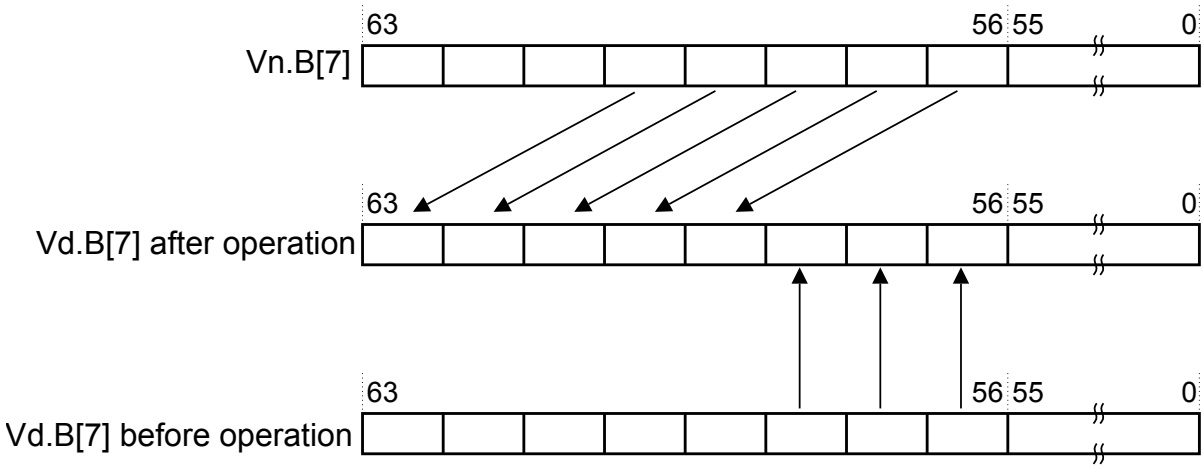

Figure C7-3 shift left by 3 for an 8-bit vector element

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23	22–19	18–16	15–11	10	9–5	4–0
0	1	1	1	1 1 1	1	0	1 x x x	immb	0 1 0 1 0	1	Rn	Rd
		U					immh		opcode			

Encoding

```
SLI  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;
let shift : integer = UInt(immh::immb) - esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	1 0	!=0000	immb	0 1 0 1 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
SLI  <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let shift : integer = UInt(immh::immb) - esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the left shift amount, in the range 0 to 63, encoded as `UInt("immh:immb") - 64`.

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let operand2 : bits(datasize) = V{}(d);
let mask : bits(esize) = LSL(Ones{esize}, shift);
var result : bits(datasize);
var shifted : bits(esize);

for e = 0 to elements-1 do
    shifted = LSL(operand[e*:esize], shift);
    result[e*:esize] = (operand2[e*:esize] AND NOT(mask)) OR shifted;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.303 SM3PARTW1

SM3PARTW1

This instruction takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive-OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

Advanced SIMD

(FEAT_SM3)

31–28	27–25	24–23	22–21	20–16	15	14	13–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 0	1 1	Rm	1	1	0 0	0 0	Rn	Rd
						O		opcode		

Encoding

```
SM3PARTW1  <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Vd : bits(128) = V{}(d);
var result : bits(128);

result[95:0] = (Vd XOR Vn)[95:0] XOR (ROL(Vm[127:96], 15)::ROL(Vm[95:64], 15)::ROL(Vm[63:32], 15));

for i = 0 to 3 do
    if i == 3 then
        result[127:96] = (Vd XOR Vn)[127:96] XOR (ROL(result[31:0], 15));
    end;
    result[(32*i)+31:(32*i)] = (result[(32*i)+31:(32*i)] XOR ROL(result[(32*i)+31:(32*i)], 15) XOR
                                ROL(result[(32*i)+31:(32*i)], 23));
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.304 SM3PARTW2

SM3PARTW2

This instruction takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive-OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

Advanced SIMD

(FEAT_SM3)

31–28	27–25	24–23	22–21	20–16	15	14	13–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 0	1 1	Rm	1	1	0 0	0 1	Rn	Rd
						O		opcode		

Encoding

```
SM3PARTW2  <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Vd : bits(128) = V{}(d);

var result : bits(128);
var tmp : bits(128);
var tmp2 : bits(32);

tmp[127:0] = (Vn XOR
               (ROL(Vm[127:96], 7) :: ROL(Vm[95:64], 7) :: ROL(Vm[63:32], 7) :: ROL(Vm[31:0], 7)));
result[127:0] = Vd[127:0] XOR tmp[127:0];
tmp2 = ROL(tmp[31:0], 15);
tmp2 = tmp2 XOR ROL(tmp2, 15) XOR ROL(tmp2, 23);
result[127:96] = result[127:96] XOR tmp2;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.305 SM3SS1

SM3SS1

This instruction rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

Advanced SIMD

(FEAT_SM3)

31–28	27–25	24–23	22–21	20–16	15	14–10	9–5	4–0
1 1 0 0	1 1 1	0 0	1 0	Rm	0	Ra	Rn	Rd
			Op0					

Encoding

```
SM3SS1  <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let a : integer{} = UInt(Ra);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va>

Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Va : bits(128) = V{}(a);

var result : bits(128);
result[127:96] = ROL((ROL(Vn[127:96], 12) + Vm[127:96] + Va[127:96]), 7);
result[95:0] = Zeros{96};
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.306 SM3TT1A

SM3TT1A

This instruction takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive-OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive-OR.
- The result of the exclusive-OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

Advanced SIMD

(FEAT_SM3)

31 28	27 25	24 23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1	0 0	1 0	Rm	1 0	imm2	0 0	Rn	Rd
							opcode		

Encoding

```
SM3TT1A  <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let i : integer = UInt(imm2);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2>

Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Vd : bits(128) = V{}(d);

var WjPrime : bits(32);
var result : bits(128);
var TT1 : bits(32);
var SS2 : bits(32);

WjPrime = Vm[i*:32];
SS2 = Vn[127:96] XOR ROL(Vd[127:96], 12);
TT1 = Vd[63:32] XOR (Vd[127:96] XOR Vd[95:64]);
TT1 = (TT1 + Vd[31:0] + SS2 + WjPrime)[31:0];
result[31:0] = Vd[63:32];
result[63:32] = ROL(Vd[95:64], 9);
result[95:64] = Vd[127:96];
result[127:96] = TT1;
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.307 SM3TT1B

SM3TT1B

This instruction takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive-OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

Advanced SIMD

(FEAT_SM3)

31–28	27–25	24–23	22–21	20–16	15–14	13–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 0	1 0	Rm	1 0	imm2	0 1	Rn	Rd
							opcode		

Encoding

```
SM3TT1B  <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let i : integer = UInt(imm2);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2>

Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Vd : bits(128) = V{}(d);

var WjPrime : bits(32);
var result : bits(128);
var TT1 : bits(32);
var SS2 : bits(32);

WjPrime = Vm[i*:32];
SS2 = Vn[127:96] XOR ROL(Vd[127:96], 12);
TT1 = (Vd[127:96] AND Vd[63:32]) OR (Vd[127:96] AND Vd[95:64]) OR (Vd[63:32] AND Vd[95:64]);
TT1 = (TT1 + Vd[31:0] + SS2 + WjPrime)[31:0];
result[31:0] = Vd[63:32];
result[63:32] = ROL(Vd[95:64], 9);
result[95:64] = Vd[127:96];
result[127:96] = TT1;
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.308 SM3TT2A

SM3TT2A

This instruction takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive-OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive-OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive-OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive-OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

Advanced SIMD

(FEAT_SM3)

31 28	27 25	24 23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1	0 0	1 0	Rm	1 0	imm2	1 0	Rn	Rd
							opcode		

Encoding

```
SM3TT2A  <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let i : integer = UInt(imm2);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2>

Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Vd : bits(128) = V{}(d);

var Wj : bits(32);
var result : bits(128);
var TT2 : bits(32);

Wj = Vm[i*:32];
TT2 = Vd[63:32] XOR (Vd[127:96] XOR Vd[95:64]);
TT2 = (TT2 + Vd[31:0] + Vn[127:96] + Wj)[31:0];

result[31:0] = Vd[63:32];
result[63:32] = ROL(Vd[95:64], 19);
result[95:64] = Vd[127:96];
result[127:96] = TT2 XOR ROL(TT2, 9) XOR ROL(TT2, 17);
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.309 SM3TT2B

SM3TT2B

This instruction takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive-OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive-OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

Advanced SIMD

(FEAT_SM3)

31 28	27 25	24 23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
1 1 0 0	1 1 1	0 0	1 0	Rm	1 0	imm2	1 1	Rn	Rd
							opcode		

Encoding

```
SM3TT2B  <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let i : integer = UInt(imm2);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2>

Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let Vd : bits(128) = V{}(d);

var Wj : bits(32);
var result : bits(128);
var TT2 : bits(32);

Wj = Vm[i*:32];
TT2 = (Vd[127:96] AND Vd[95:64]) OR (NOT(Vd[127:96]) AND Vd[63:32]);
TT2 = (TT2 + Vd[31:0] + Vn[127:96] + Wj)[31:0];

result[31:0] = Vd[63:32];
result[63:32] = ROL(Vd[95:64], 19);
result[95:64] = Vd[127:96];
result[127:96] = TT2 XOR ROL(TT2, 9) XOR ROL(TT2, 17);
V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.310 SM4E

SM4 encode

This instruction takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SM4)

31–28	27–25	24–23	22–19	18–12	11–10	9–5	4–0
1 1 0 0	1 1 1	0 1	1 0 0 0	0 0 0 1 0 0 0	0 1	Rn	Rd
					opcode		

Encoding

```
SM4E  <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM4) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vn : bits(128) = V{}(n);
var intval : bits(32);
var roundresult : bits(128);
var roundkey : bits(32);

roundresult = V{128}(d);
for index = 0 to 3 do
    roundkey = Vn[index*:32];

    intval = roundresult[127:96] XOR roundresult[95:64] XOR roundresult[63:32] XOR roundkey;

    for i = 0 to 3 do
        intval[i*:8]  = Sbox(intval[i*:8]);
    end;

    intval = intval XOR ROL(intval, 2) XOR ROL(intval, 10) XOR ROL(intval, 18) XOR ROL(intval, 24);
    intval = intval XOR roundresult[31:0];

    roundresult[31:0]   = roundresult[63:32];
    roundresult[63:32]  = roundresult[95:64];
    roundresult[95:64]  = roundresult[127:96];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    roundresult[127:96] = intval;
end;

V{128}(d) = roundresult;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.311 SM4EKEY

SM4 key

This instruction takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SM4)

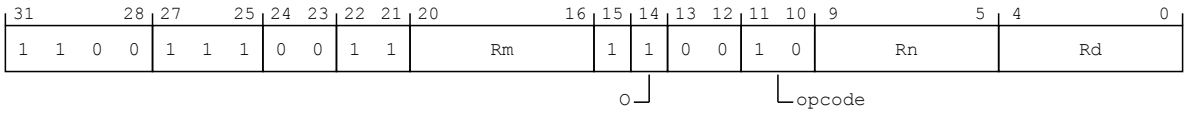

Encoding

```
SM4EKEY  <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM4) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
var intval : bits(32);
var const : bits(32);
var roundresult : bits(128);

roundresult = V{128}(n);
for index = 0 to 3 do
    const = Vm[index*:32];

    intval = roundresult[127:96] XOR roundresult[95:64] XOR roundresult[63:32] XOR const;

    for i = 0 to 3 do
        intval[i*:8] = Sbox(intval[i*:8]);
    end;

    intval = intval XOR ROL(intval, 13) XOR ROL(intval, 23);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    intval = intval XOR roundresult[31:0];

    roundresult[31:0]   = roundresult[63:32];
    roundresult[63:32]  = roundresult[95:64];
    roundresult[95:64]  = roundresult[127:96];
    roundresult[127:96] = intval;
end;

V{128}(d) = roundresult;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.312 SMAX

Signed maximum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

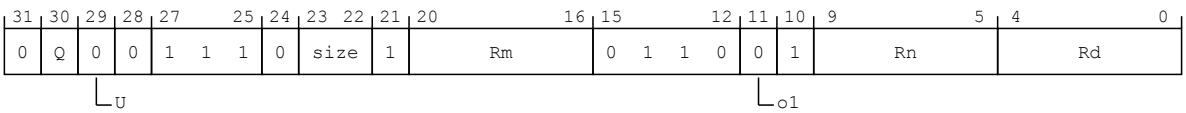

Encoding

```
SMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var max : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    max = Max(element1, element2);
    result[e*:esize] = max[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.313 SMAXP

Signed maximum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0 1 0	0	1	Rn	Rd
		U								o1			

Encoding

```
SMAXP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : integer;
var element2 : integer;
var max : integer;

for e = 0 to elements-1 do
    element1 = SInt(concat[(2*e)*:esize]);
    element2 = SInt(concat[((2*e)+1)*:esize]);
    max = Max(element1, element2);
    result[e*:esize] = max[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.314 SMAXV

Signed maximum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16	15 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 1 0 0 0	0	1 0 1 0	1 0	Rn	Rd
		U						op				

Encoding

```
SMAXV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var opelt : bits(esize) = operand[0+:esize];
var max : integer = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1 do
    opelt = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    max = Max(max, element);
end;

V{esize}(d) = max[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.315 SMIN

Signed minimum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

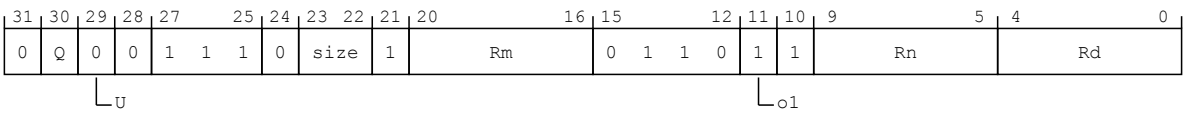

Encoding

```
SMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var min : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    min = Min(element1, element2);
    result[e*:esize] = min[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.316 SMINP

Signed minimum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0 1 0	1	1	Rn	Rd
		U								o1			

Encoding

```
SMINP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : integer;
var element2 : integer;
var min : integer;

for e = 0 to elements-1 do
    element1 = SInt(concat[(2*e)*:esize]);
    element2 = SInt(concat[((2*e)+1)*:esize]);
    min = Min(element1, element2);
    result[e*:esize] = min[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.317 SMINV

Signed minimum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16	15 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 1 0 0 0	1	1 0 1 0	1 0	Rn	Rd
		U						op				

Encoding

```
SMINV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var opelt : bits(esize) = operand[0+:esize];
var min : integer = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1 do
    opelt = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    min = Min(min, element);
end;

V{esize}(d) = min[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.318 SMLAL, SMLAL2 (by element)

Signed multiply-add long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The `SMLAL` instruction extracts vector elements from the lower half of the first source register. The `SMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	size	L	M	Rm	0	0	1 0	H	0	Rn	Rd
		U									o2					

Encoding

```
SMLAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = SInt(operand2[index*:esize]);
var product : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = operand3[e*:(2*esize)] + product;
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.319 SMLAL, SMLAL2 (vector)

Signed multiply-add long (vector)

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLAL` instruction extracts each source vector from the lower half of each source register. The `SMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0	0	0	0 0	Rn	Rd
		U								o1				

Encoding

```
SMLAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var accum : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    accum = operand3[e*:(2*esize)] + product;
    result[e*:(2*esize)] = accum;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.320 SMLSL, SMLSL2 (by element)

Signed multiply-subtract long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts vector elements from the lower half of the first source register. The `SMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	size	L	M	Rm	0	1	1 0	H	0	Rn	Rd
		U									o2					

Encoding

```
SMLSL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = SInt(operand2[index*:esize]);
var product : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = operand3[e*:(2*esize)] - product;
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.321 SMLSL, SMLSL2 (vector)

Signed multiply-subtract long (vector)

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts each source vector from the lower half of each source register. The `SMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0	1	0	0 0	Rn	Rd
		U								o1				

Encoding

```
SMLSL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var accum : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    accum = operand3[e*:(2*esize)] - product;
    result[e*:(2*esize)] = accum;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.322 SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate to 32-bit integer (vector)

This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)

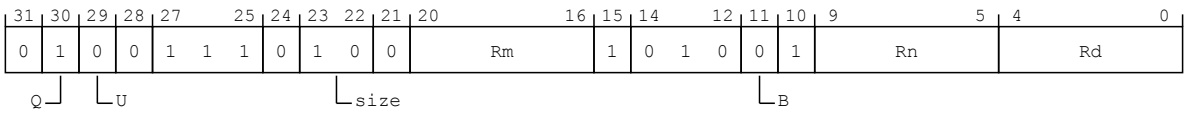

Encoding

```
SMMLA  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_I8MM) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let addend : bits(128)   = V{}(d);

V{128}(d) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Operational Information

Arm expects that the SMMLA (vector) instruction will deliver a peak integer multiply throughput that is at least as high as can be achieved using two SDOT (vector) instructions, with a goal that it should have significantly higher throughput.

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.323 SMOV

Signed move vector element to general-purpose register

This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

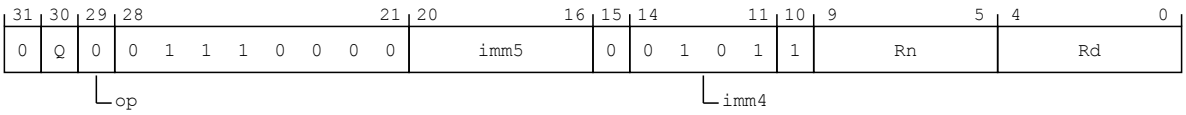

Encoding for the 32-bit variant

Applies when `(Q == 0)`

```
SMOV  <Wd>, <Vn>.<Ts>[<index>]
```

Encoding for the 64-bit variant

Applies when `(Q == 1)`

```
SMOV  <Xd>, <Vn>.<Ts>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'xx000' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[2:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << size;
let datasize : integer{} = 32 << UInt(Q);
if datasize <= esize then EndOfDecode(Decode_UNDEF); end;
let index : integer = UInt(imm5[4:size+1]);
let idxdsize : integer{} = 64 << UInt(imm5[4]);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ts>

For the “32-bit” variant: is an element size specifier, encoded in “imm5”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<Ts>
xxx00	RESERVED
xxxx1	B
xxx10	H

For the “64-bit” variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<index>

For the “32-bit” variant: is the element index encoded in “imm5”:

imm5	<index>
xxx00	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)

For the “64-bit” variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
if index == 0 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;
let operand : bits(idxdsize) = V{}(n);

X{datasize}(d) = SignExtend{datasize}(operand[index*:esize]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.324 SMULL, SMULL2 (by element)

Signed multiply long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts vector elements from the lower half of the first source register. The `SMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

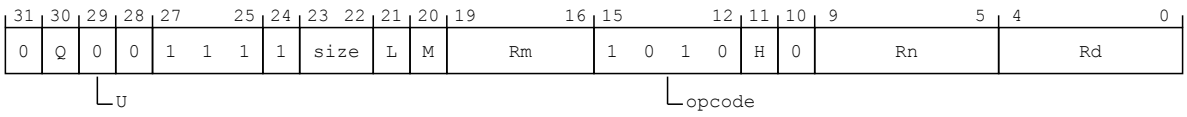

Encoding

```
SMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = SInt(operand2[index*:esize]);
var product : bits(2*esize);

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = product;
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.325 SMULL, SMULL2 (vector)

Signed multiply long (vector)

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts each source vector from the lower half of each source register. The `SMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

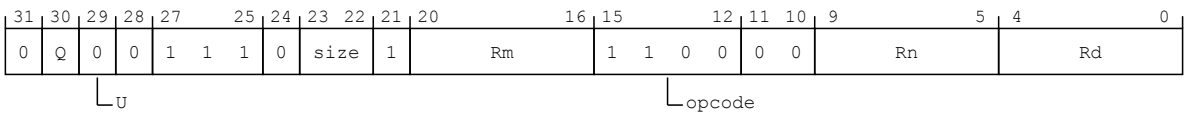

Encoding

```
SMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    result[e*:(2*esize)] = (element1 * element2)[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.326 SQABS

Signed saturating absolute value

This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U															opcode														

Encoding

```
SQABS  <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U															opcode														

Encoding

```
SQABS  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    element = Abs(element);
    (result[e*:esize], sat) = SignedSatQ{esize}(element);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.327 SQADD

Signed saturating add

This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	size		1	Rm	0	0	0	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
SQADD  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size		1	Rm	0	0	0	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
SQADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    sum = element1 + element2;
    (result[e*:esize], sat) = SatQ{esize}(sum, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.328 SQDMLAL, SQDMLAL2 (by element)

Signed saturating doubling multiply-add long (by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register. The `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	1	size		L	M	Rm	0	0	1	1	H	0	Rn	Rd
		U												o2						

Encoding

```
SQDMLAL  <Va><d>, <Vb><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	1	size		L	M	Rm		0	0	1	1	H	0	Rn		Rd	
		U													o2								

Encoding

```
SQDMLAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Va>

Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>

Is the source width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var element2 : integer;
var product : bits(2*esize);
var accum : integer;
var sat1 : boolean;
var sat2 : boolean;

element2 = SInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    (product, sat1) = SignedSatQ{2*esize}(2 * element1 * element2);
    accum = SInt(operand3[e*:(2*esize)]) + SInt(product);
    (result[e*:(2*esize)], sat2) = SignedSatQ{2*esize}(accum);
    if sat1 || sat2 then FPSR().QC = '1'; end;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.329 SQDMLAL, SQDMLAL2 (vector)

Signed saturating doubling multiply-add long

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register. The `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–14	13	12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	size	1	Rm	1 0	0	1	0 0	Rn	Rd
		U								o1				

Encoding

```
SQDMLAL  <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–14	13	12	11–10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0	0	1	0 0	Rn	Rd
		U								o1				

Encoding

```
SQDMLAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Va>

Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>

Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var accum : integer;
var sat1 : boolean;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var sat2 : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    (product, sat1) = SignedSatQ{2*esize}(2 * element1 * element2);
    accum = SInt(operand3[e*:(2*esize)]) + SInt(product);
    (result[e*:(2*esize)], sat2) = SignedSatQ{2*esize}(accum);
    if sat1 || sat2 then FPSR().QC = '1'; end;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.330 SQDMLSL, SQDMLSL2 (by element)

Signed saturating doubling multiply-subtract long (by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register. The `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	1	size		L	M	Rm	0	1	1	1	H	0	Rn	Rd
		U												o2						

Encoding

```
SQDMLSL  <Va><d>, <Vb><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15	14	13 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	size	L	M	Rm	0	1	1 1	H	0	Rn	Rd
		U									o2					

Encoding

```
SQDMLSL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Va>

Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>

Is the source width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var element2 : integer;
var product : bits(2*esize);
var accum : integer;
var sat1 : boolean;
var sat2 : boolean;

element2 = SInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    (product, sat1) = SignedSatQ{2*esize}(2 * element1 * element2);
    accum = SInt(operand3[e*:(2*esize)]) - SInt(product);
    (result[e*:(2*esize)], sat2) = SignedSatQ{2*esize}(accum);
    if sat1 || sat2 then FPSR().QC = '1'; end;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.331 SQDMLSL, SQDMLSL2 (vector)

Signed saturating doubling multiply-subtract long

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register. The `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	1	0	1	1 1 1	0	size	1	Rm	1 0	1	1	0 0	Rn	Rd
		U								o1				

Encoding

```
SQDMLSL  <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0	1	1	0 0	Rn	Rd
		U								o1				

Encoding

```
SQDMLSL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Va>

Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>

Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var accum : integer;
var sat1 : boolean;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var sat2 : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    (product, sat1) = SignedSatQ{2*esize}(2 * element1 * element2);
    accum = SInt(operand3[e*:(2*esize)]) - SInt(product);
    (result[e*:(2*esize)], sat2) = SignedSatQ{2*esize}(accum);
    if sat1 || sat2 then FPSR().QC = '1'; end;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.332 SQDMULH (by element)

Signed saturating doubling multiply returning high half (by element)

This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	1	0	1	1	1	1	1	size	L	M	Rm	1	1	0	0	H	0	Rn	Rd
		U													op				

Encoding

```
SQDMULH  <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);    Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;

let round : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23 22	21	20	19 16	15	14	13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	1	0	0	H	0	Rn	Rd
		U													op				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SQDMULH  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);   Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let round : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<m>
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : integer;
var sat : boolean;

element2 = SInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, round);
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (result[e*:esize], sat) = SignedSatQ{esize}(product);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.333 SQDMULH (vector)

Signed saturating doubling multiply returning high half

This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0	1	0	1	1 1 1	0	size	1	Rm	1 0 1 1 0	1	Rn	Rd
		U							opcode			

Encoding

```
SQDMULH  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	1 0 1 1 0	1	Rn	Rd
		U							opcode			

Encoding

```
SQDMULH  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, rounding);
    (result[e*:esize], sat) = SignedSatQ{esize}(product);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.334 SQDMULL, SQDMULL2 (by element)

Signed saturating doubling multiply long (by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register. The `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Vector and Scalar

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	size	L	M	Rm	1 0 1 1	H	0	Rn	Rd
		U								opcode				

Encoding

```
SQDMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	1	0	1	1 1 1	1	size	L	M	Rm	1 0 1 1	H	0	Rn	Rd
		U								opcode				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SQDMULL  <Va><d>, <Vb><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<index>
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<Va>

Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>

Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = SInt(operand2[index*:esize]);
var product : bits(2*esize);
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    (product, sat) = SignedSatQ{2*esize}(2 * element1 * element2);
    result[e*:(2*esize)] = product;
    if sat then FPSR().QC = '1'; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.335 SQDMULL, SQDMULL2 (vector)

Signed saturating doubling multiply long

This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the result, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts each source vector from the lower half of each source register. The `SQDMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	size	1	Rm	1 1 0 1	0 0	Rn	Rd
		U							opcode			

Encoding

```
SQDMULL  <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–12	11–10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	1 1 0 1	0 0	Rn	Rd
		U							opcode			

Encoding

```
SQDMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Va>

Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>

Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var sat : boolean;

for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    (product, sat) = SignedSatQ{2*esize}(2 * element1 * element2);
    result[e*:(2*esize)] = product;
    if sat then FPSR().QC = '1'; end;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.336 SQNEG

Signed saturating negate

This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U															opcode														

Encoding

```
SQNEG  <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U															opcode														

Encoding

```
SQNEG  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element = SInt(operand[e*:esize]);
    element = -element;
    (result[e*:esize], sat) = SignedSatQ{esize}(element);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.337 SQRDMLAH (by element)

Signed saturating rounding doubling multiply accumulate returning high half (by element)

This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 14	13	12	11	10	9 5	4 0
0	1	1	1	1 1 1	1	size	L	M	Rm	1 1	0	1	H	0	Rn	Rd
		U									S					

Encoding

```
SQRDMLAH  <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;

case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;

let rounding : boolean = TRUE;
```

Vector

(FEAT_RDM)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 14	13	12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	size	L	M	Rm	1 1	0	1	H	0	Rn	Rd
		U									S					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SQRDMLAH  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;

case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let rounding : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<m>
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var element3 : integer;
var accum : integer;
var sat : boolean;

element2 = SInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element3 = SInt(operand3[e*:esize]);
    accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (result[e*:esize], sat) = SignedSatQ{esize}(accum);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.338 SQRDMLAH (vector)

Signed saturating rounding doubling multiply accumulate returning high half (vector)

This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

31	30	29	28	27	25	24	23 22	21	20 16	15	14 12	11	10	9 5	4 0
0	1	1	1	1 1	1	0	size	0	Rm	1	0 0 0	0	1	Rn	Rd
		U										S			

Encoding

```
SQRDMLAH  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : boolean = TRUE;
```

Vector

(FEAT_RDM)

31	30	29	28	27	25	24	23 22	21	20 16	15	14 12	11	10	9 5	4 0
0	Q	1	0	1 1	1	0	size	0	Rm	1	0 0 0	0	1	Rn	Rd
		U										S			

Encoding

```
SQRDMLAH  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var element3 : integer;
var accum : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    element3 = SInt(operand3[e*:esize]);
    accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (result[e*:esize], sat) = SignedSatQ{esize}(accum);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.339 SQRDMLSH (by element)

Signed saturating rounding doubling multiply subtract returning high half (by element)

This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 14	13	12	11	10	9 5	4 0
0	1	1	1	1 1 1	1	size	L	M	Rm	1 1	1	1	H	0	Rn	Rd
		U									S					

Encoding

```
SQRDMLSH  <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;

case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;

let rounding : boolean = TRUE;
```

Vector

(FEAT_RDM)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 14	13	12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	size	L	M	Rm	1 1	1	1	H	0	Rn	Rd
		U									S					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SQRDMLSH  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;

case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let rounding : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<m>
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var element3 : integer;
var accum : integer;
var sat : boolean;

element2 = SInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element3 = SInt(operand3[e*:esize]);
    accum = (element3 << esize) - 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (result[e*:esize], sat) = SignedSatQ{esize}(accum);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.340 SQRDMLSH (vector)

Signed saturating rounding doubling multiply subtract returning high half (vector)

This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

31	30	29	28	27	25	24	23 22	21	20 16	15	14 12	11	10	9 5	4 0
0	1	1	1	1 1	1	0	size	0	Rm	1	0 0 0	1	1	Rn	Rd
		U										S			

Encoding

```
SQRDMLSH  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : boolean = TRUE;
```

Vector

(FEAT_RDM)

31	30	29	28	27	25	24	23 22	21	20 16	15	14 12	11	10	9 5	4 0
0	Q	1	0	1 1	1	0	size	0	Rm	1	0 0 0	1	1	Rn	Rd
		U										S			

Encoding

```
SQRDMLSH  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var element3 : integer;
var accum : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    element3 = SInt(operand3[e*:esize]);
    accum = (element3 << esize) - 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (result[e*:esize], sat) = SignedSatQ{esize}(accum);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.341 SQRDMULH (by element)

Signed saturating rounding doubling multiply returning high half (by element)

This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQDMULH.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	1	0	1	1	1	1	1	size		L	M	Rm	1	1	0	1	H	0	Rn	Rd
		U														op				

Encoding

```
SQRDMULH  <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);    Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;

let round : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11	10	9–5	4–0
0	Q	0	0	1	1	1	1	size		L	M	Rm	1	1	0	1	H	0	Rn	Rd
		U														op				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SQRDMULH  <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L);   Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let round : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<m>
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(idxdsize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : integer;
var sat : boolean;

element2 = SInt(operand2[index*:esize]);
for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, round);
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (result[e*:esize], sat) = SignedSatQ{esize}(product);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.342 SQRDMULH (vector)

Signed saturating rounding doubling multiply returning high half

This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQDMULH.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	1	1	1	1 1 1	0	size	1	Rm	1 0 1 1 0	1	Rn	Rd
		U							opcode			

Encoding

```
SQRDMULH  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let rounding : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	1 0 1 1 0	1	Rn	Rd
		U							opcode			

Encoding

```
SQRDMULH  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let rounding : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var product : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, rounding);
    (result[e*:esize], sat) = SignedSatQ{esize}(product);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.343 SQRSHL

Signed saturating rounding shift left (register)

This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	0	size	1	Rm	0 1 0	1	1	1	Rn	Rd
		U								R	S			

Encoding

```
SQRSHL  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = FALSE;
let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1 0	1	1	1	Rn	Rd
		U								R	S			

Encoding

```
SQRSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = FALSE;
let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    var element : integer = SInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    (result[e*:esize], sat) = SatQ{esize}(element, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.344 SQRSHRN, SQRSHRN2

Signed saturating rounded shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQSHRN.

The `SQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
		U				immh			op			

Encoding

```
SQRSHRN  <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
		U				immh			op			

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SQRSHRN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<Tb>
1xxx	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(2*esize) = operand[e*:(2*esize)];
    if unsigned then
        let element : integer = RShr(UInt(opelt), shift, round);
        (result[e*:esize], sat) = UnsignedSatQ{esize}(element);
    else
        let element : integer = RShr(SInt(opelt), shift, round);
        (result[e*:esize], sat) = SignedSatQ{esize}(element);
    end;
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.345 SQRSHRUN, SQRSHRUN2

Signed saturating rounded shift right unsigned narrow (immediate)

This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQSHRUN.

The `SQRSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQRSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	19	18	16	15	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000	immb		1	0	0	0	1	1	Rn	Rd
		U							immh							op			

Encoding

```
SQRSHRUN  <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	19	18	16	15	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000	immb		1	0	0	0	1	1	Rn	Rd
		U							immh							op			

Encoding

```
SQRSHRUN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var element : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element = RShr(SInt(operand[e*:(2*esize)]), shift, round);
    (result[e*:esize], sat) = UnsignedSatQ{esize}(element);
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.346 SQSHL (immediate)

Signed saturating shift left (immediate)

This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	!=0000	immb	0 1 1	1	0	1	Rn	Rd
		U				immh			op				

Encoding

```
SQSHL  <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = esize;
let elements : integer = 1;
let shift : integer = UInt(immh::immb) - esize;

let src_unsigned : boolean = FALSE;
let dst_unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	0 1 1	1	0	1	Rn	Rd
		U				immh			op				

Encoding

```
SQSHL  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let shift : integer = UInt(immh::immb) - esize;

let src_unsigned : boolean = FALSE;
let dst_unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>
0001	B
001x	H
01xx	S
1xxx	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
    (result[e*:esize], sat) = SatQ{esize}(element << shift, dst_unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.347 SQSHL (register)

Signed saturating shift left (register)

This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	0	size	1	Rm	0 1 0	0	1	1	Rn	Rd
		U								R	S			

Encoding

```
SQSHL  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = FALSE;
let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21	20–16	15–13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1 0	0	1	1	Rn	Rd
		U								R	S			

Encoding

```
SQSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = FALSE;
let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    var element : integer = SInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    (result[e*:esize], sat) = SatQ{esize}(element, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.348 SQSHLU

Signed saturating shift left unsigned (immediate)

This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	1 0	!=0000	immb	0 1 1	0	0	1	Rn	Rd
		U				immh			op				

Encoding

```
SQSHLU  <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = esize;
let elements : integer = 1;
let shift : integer = UInt(immh::immb) - esize;

let src_unsigned : boolean = FALSE;
let dst_unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–13	12	11	10	9–5	4–0
0	Q	1	0	1 1 1	1 0	!=0000	immb	0 1 1	0	0	1	Rn	Rd
		U				immh			op				

Encoding

```
SQSHLU  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let shift : integer = UInt(immh::immb) - esize;

let src_unsigned : boolean = FALSE;
let dst_unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>
0001	B
001x	H
01xx	S
1xxx	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
    (result[e*:esize], sat) = SatQ{esize}(element << shift, dst_unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.349 SQSHRN, SQSHRN2

Signed saturating shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see SQRSHRN.

The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	!=0000	immb	1 0 0 1	0	1	Rn	Rd
		U				immh			op			

Encoding

```
SQSHRN  <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	1 0 0 1	0	1	Rn	Rd
		U				immh			op			

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SQSHRN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<Tb>
1xxx	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(2*esize) = operand[e*:(2*esize)];
    if unsigned then
        let element : integer = RShr(UInt(opelt), shift, round);
        (result[e*:esize], sat) = UnsignedSatQ{esize}(element);
    else
        let element : integer = RShr(SInt(opelt), shift, round);
        (result[e*:esize], sat) = SignedSatQ{esize}(element);
    end;
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.350 SQSHRUN, SQSHRUN2

Signed saturating shift right unsigned narrow (immediate)

This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSHRUN.

The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 12	11	10	9 5	4 0
0	1	1	1	1 1 1	1 0	!=0000	immb	1 0 0 0	0	1	Rn	Rd
		U				immh			op			

Encoding

```
SQSHRUN  <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1 0	!=0000	immb	1 0 0 0	0	1	Rn	Rd
		U				immh			op			

Encoding

```
SQSHRUN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var element : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element = RShr(SInt(operand[e*:(2*esize)]), shift, round);
    (result[e*:esize], sat) = UnsignedSatQ{esize}(element);
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.351 SQSUB

Signed saturating subtract

This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	0	1	1	1	1	0	size		1	Rm	0	0	1	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
SQSUB  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	0	0	1	1	1	0	size		1	Rm	0	0	1	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
SQSUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var diff : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    diff = element1 - element2;
    (result[e*:esize], sat) = SatQ{esize}(diff, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.352 SQXTN, SQXTN2

Signed saturating extract narrow

This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0	1	0	1	1 1 1	0	size	1 0 0 0 0	1 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
SQXTN  <Vb><d>, <Va><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let part : integer = 0;
let elements : integer = 1;
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	1 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
SQXTN{2}  <Vd>.<Tb>, <Vn>.<Ta>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(2*datasize) = V{}(n);
var result : bits(datasize);
var element : bits(2*esize);
var sat : boolean;

for e = 0 to elements-1 do
    element = operand[e*:(2*esize)];
    (result[e*:esize], sat) = SatQ{esize}(SInt(element), unsigned);
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.353 SQXTUN, SQXTUN2

Signed saturating extract unsigned narrow

This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd				
		U													opcode																

Encoding

```
SQXTUN  <Vb><d>, <Va><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let part : integer = 0;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd				
		U													opcode																

Encoding

```
SQXTUN{2}  <Vd>.<Tb>, <Vn>.<Ta>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(2*datasize) = V{}(n);
var result : bits(datasize);
var element : bits(2*esize);
var sat : boolean;

for e = 0 to elements-1 do
    element = operand[e*:(2*esize)];
    (result[e*:esize], sat) = UnsignedSatQ{esize}(SInt(element));
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.354 SRHADD

Signed rounding halving add

This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0 0 1 0	1	Rn	Rd
		U							opcode			

Encoding

```
SRHADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    sum = (element1 + element2 + 1) >> 1;
    result[e*:esize] = sum[esize-1:0];
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.355 SRI

Shift right and insert (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

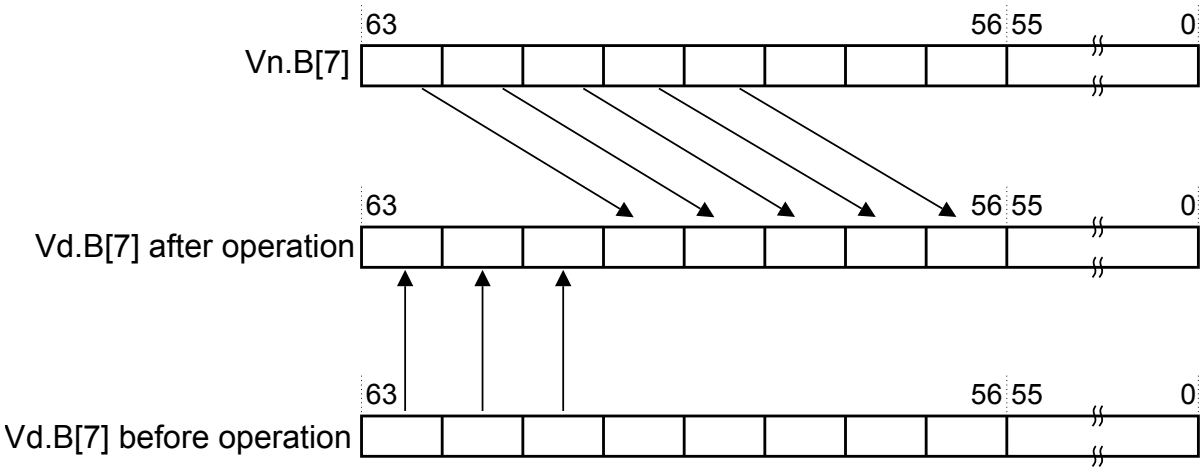

Figure C7-4 shift right by 3 for an 8-bit vector element

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

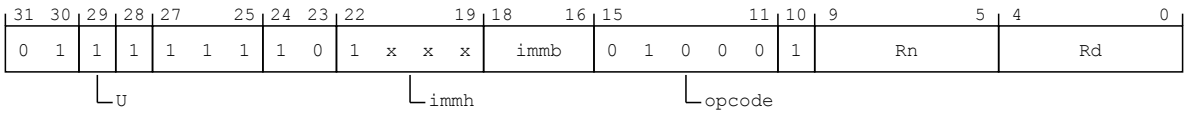

Encoding

```
SRI  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;
let shift : integer = (esize * 2) - UInt(immh::immb);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	1 0	!=0000	immb	0 1 0 0 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
SRI  <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let shift : integer = (esize * 2) - UInt(immh::immb);
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt("immh:immb")`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let operand2 : bits(datasize) = V{}(d);
let mask : bits(esize) = LSR(Ones{esize}, shift);
var result : bits(datasize);
var shifted : bits(esize);

for e = 0 to elements-1 do
    shifted = LSR(operand[e*:esize], shift);
    result[e*:esize] = (operand2[e*:esize] AND NOT(mask)) OR shifted;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.356 SRSHL

Signed rounding shift left (register)

This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see SSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

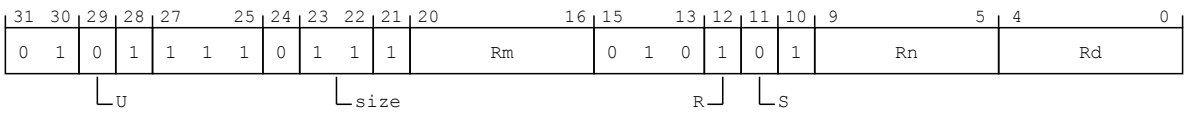

Encoding

```
SRSHL  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

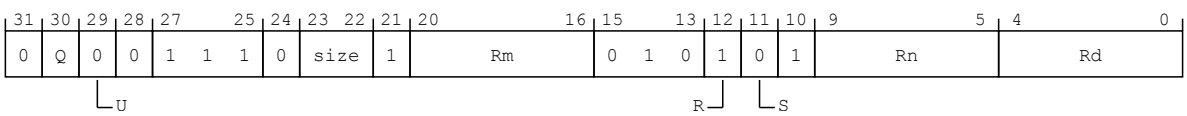

Encoding

```
SRSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    var element : integer = SInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.357 SRSHR

Signed rounding shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	1 x x x	immb	0 0	1	0	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SRSHR  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	0 0	1	0	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SRSHR  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = TRUE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.358 SRSRA

Signed rounding shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	1 x x x	immb	0 0	1	1	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SRSRA  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	0 0	1	1	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SRSRA  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = TRUE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let operand2 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.359 SSHL

Signed shift left (register)

This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see SRSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

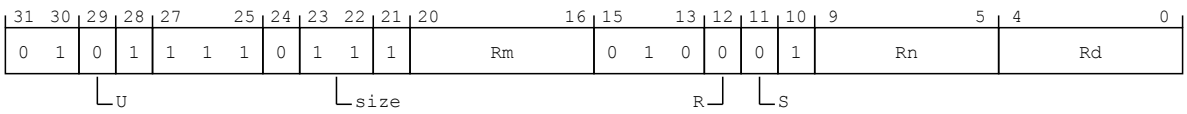

Encoding

```
SSHL  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

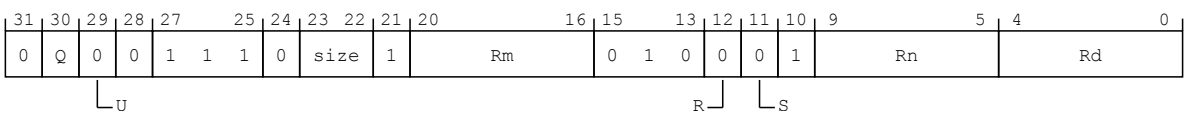

Encoding

```
SSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    var element : integer = SInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.360 SSHLL, SSHLL2

Signed shift left long (immediate)

This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SSHLL` instruction extracts vector elements from the lower half of the source register. The `SSHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	0	0	1 1 1	1 0	!=0000	immb	1 0 1 0 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
SSHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = UInt(immh::immb) - esize;
let unsigned : boolean = FALSE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift>

Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	RESERVED

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
SXTL, SXTL2	immb == '000' && BitCount(immh) == 1

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = Vpart{}(n, part);
var result : bits(datasize*2);

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    result[e*:(2*esize)] = (element << shift)[2*esize-1:0];
end;

V{datasize*2}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.361 SSHR

Signed shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	1 x x x	immb	0 0	0	0	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SSHR  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	0 0	0	0	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SSHR  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = FALSE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.362 SSRA

Signed shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	1	0	1	1 1 1	1 0	1 x x x	immb	0 0	0	1	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SSRA  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	Q	0	0	1 1 1	1 0	!=0000	immb	0 0	0	1	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
SSRA  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = FALSE;
let round : boolean = FALSE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let operand2 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.363 SSUBL, SSUBL2

Signed subtract long

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The `SSUBL` instruction extracts each source vector from the lower half of each source register. The `SSUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0	1	0	0 0	Rn	Rd
		U								o1				

Encoding

```
SSUBL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:esize]);
    element2 = SInt(operand2[e*:esize]);
    sum = element1 - element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.364 SSUBW, SSUBW2

Signed subtract wide

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The `SSUBW` instruction extracts the second source vector from the lower half of the second source register. The `SSUBW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 0	1	1	0 0	Rn	Rd
		U									o1			

Encoding

```
SSUBW{2}  <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = SInt(operand1[e*:(2*esize)]);
    element2 = SInt(operand2[e*:esize]);
    sum = element1 - element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.365 ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers

This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

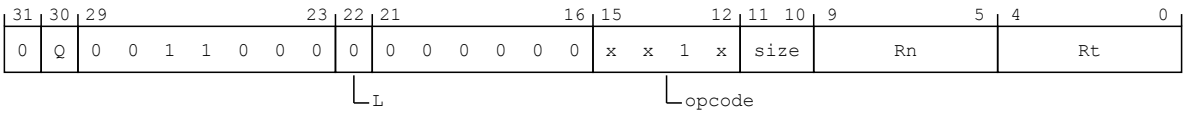

Encoding for the One register variant

Applies when `(opcode == 0111)`

```
ST1  { <Vt>.<T> }, [<Xn|SP>]
```

Encoding for the Two registers variant

Applies when `(opcode == 1010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Encoding for the Three registers variant

Applies when `(opcode == 0110)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Encoding for the Four registers variant

Applies when `(opcode == 0010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

31	30	29						23	22	21	20	16	15			12	11 10	9	5	4	0
0	Q	0	0	1	1	0	0	1	0	0	Rm		x	x	1	x	size	Rn		Rt	
									L						opcode						

Encoding for the One register, immediate offset variant

Applies when `(Rm == 11111 && opcode == 0111)`

```
ST1  { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the One register, register offset variant

Applies when `(Rm != 11111 && opcode == 0111)`

```
ST1  { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Two registers, immediate offset variant

Applies when `(Rm == 11111 && opcode == 1010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Two registers, register offset variant

Applies when `(Rm != 11111 && opcode == 1010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Three registers, immediate offset variant

Applies when `(Rm == 11111 && opcode == 0110)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Three registers, register offset variant

Applies when `(Rm != 11111 && opcode == 0110)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Four registers, immediate offset variant

Applies when `(Rm == 11111 && opcode == 0010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Four registers, register offset variant

Applies when `(Rm != 11111 && opcode == 0010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<imm>

For the “One register, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<imm>
1	#16

For the “Two registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the “Three registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the “Four registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

var rpt : integer;                // number of iterations
let selem : integer = 1; // structure elements

case opcode of
    when '0010' => rpt = 4;    // LD/ST1 (4 registers)
    when '0110' => rpt = 3;    // LD/ST1 (3 registers)
    when '1010' => rpt = 2;    // LD/ST1 (2 registers)
    when '0111' => rpt = 1;    // LD/ST1 (1 register)
    otherwise => EndOfDecode(Decode_UNDEF);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem{esize}(eaddr, accdesc) = rval[e*:esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.366 ST1 (single structure)

Store a single-element structure from one lane of one register

This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29						23	22	21	20				17	16	15			13	12	11	10	9			5	4			0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	x	x	0		S	size		Rn				Rt			
									L	R						o2			opcode											

Encoding for the 8-bit variant

Applies when `(opcode == 000)`

```
ST1  { <Vt>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 010 && size == x0)`

```
ST1  { <Vt>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 100 && size == 00)`

```
ST1  { <Vt>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 100 && S == 0 && size == 01)`

```
ST1  { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

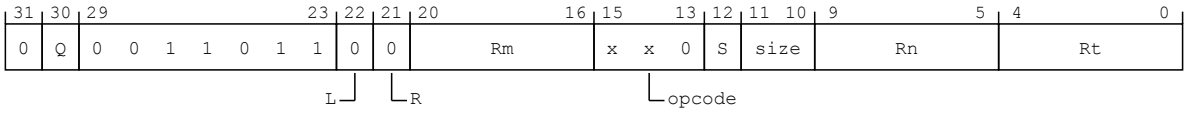

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 000)`

```
ST1  { <Vt>.B }[<index>], [<Xn|SP>], #1
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 000)`

```
ST1  { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
ST1  { <Vt>.H }[<index>], [<Xn|SP>], #2
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
ST1  { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
ST1  { <Vt>.S }[<index>], [<Xn|SP>], #4
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
ST1  { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST1  { <Vt>.D }[<index>], [<Xn|SP>], #8
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST1  { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem{esize}(eaddr, accdesc) = rval[index*:esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.367 ST2 (multiple structures)

Store multiple 2-element structures from two registers

This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29 28	27	26	25	24 23	22	21 16	15 12	11 10	9 5	4 0
0	Q	0 0	1	1	0	0 0	0	0 0 0 0 0 0	1 0 0 0	size	Rn	Rt
							L		opcode			

Encoding

```
ST2  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

31	30	29 23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0 0 1 1 0 0 1	0	0	Rm	1 0 0 0	size	Rn	Rt
			L			opcode			

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
ST2  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
ST2  { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

let rpt : integer = 1;
let selem : integer = 2;

// .1D format only permitted with LD1 & ST1
if size::Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem{esize}(eaddr, accdesc) = rval[e*:esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.368 ST2 (single structure)

Store single 2-element structure from one lane of two registers

This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29						23	22	21	20				17	16	15			13	12	11	10	9			5	4			0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0		0	x	x		0	S	size		Rn				Rt			
									L	R						o2				opcode											

Encoding for the 8-bit variant

Applies when `(opcode == 000)`

```
ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 010 && size == x0)`

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 100 && size == 00)`

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 100 && S == 0 && size == 01)`

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

31	30	29						23	22	21	20			16	15			13	12	11	10	9				5	4				0
0	Q	0	0	1	1	0	1	1	0	1	Rm				x	x	0		S	size		Rn					Rt				

L R opcode

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 000)`

```
ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 000)`

```
ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem{esize}(eaddr, accdesc) = rval[index*:esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.369 ST3 (multiple structures)

Store multiple 3-element structures from three registers

This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29 28	27	26	25	24 23	22	21 16	15 12	11 10	9 5	4 0
0	Q	0 0	1	1	0	0 0	0	0 0 0 0 0 0	0 1 0 0	size	Rn	Rt
							L		opcode			

Encoding

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

31	30	29 23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0 0 1 1 0 0 1	0	0	Rm	0 1 0 0	size	Rn	Rt
			L			opcode			

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

let rpt : integer = 1;
let selem : integer = 3;

// .1D format only permitted with LD1 & ST1
if size::Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem{esize}(eaddr, accdesc) = rval[e*:esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.370 ST3 (single structure)

Store single 3-element structure from one lane of three registers

This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29						23	22	21	20				17	16	15			13	12	11	10	9			5	4			0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	x	x	1		S	size		Rn				Rt			
									L	R						o2			opcode											

Encoding for the 8-bit variant

Applies when `(opcode == 001)`

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 011 && size == x0)`

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 101 && size == 00)`

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 101 && S == 0 && size == 01)`

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

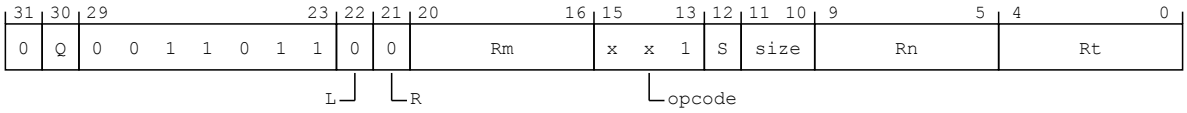

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 001)`

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 001)`

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem{esize}(eaddr, accdesc) = rval[index*:esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.371 ST4 (multiple structures)

Store multiple 4-element structures from four registers

This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29 28	27	26	25	24 23	22	21 16	15 12	11 10	9 5	4 0
0	Q	0 0	1	1	0	0 0	0	0 0 0 0 0 0	0 0 0 0	size	Rn	Rt
							L		opcode			

Encoding

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Post-index

(FEAT_AdvSIMD)

31	30	29 23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0 0 1 1 0 0 1	0	0	Rm	0 0 0 0	size	Rn	Rt
			L			opcode			

Encoding for the Immediate offset variant

Applies when `(Rm == 11111)`

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant

Applies when `(Rm != 11111)`

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<imm>
1	#64

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(size);
let elements : integer = datasize DIV esize;

let rpt : integer = 1;
let selem : integer = 4;

// .1D format only permitted with LD1 & ST1
if size::Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(datasize);
var tt : integer;
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

for r = 0 to rpt-1 do
    for e = 0 to elements-1 do
        tt = (t + r) MOD 32;
        for s = 0 to selem-1 do
            rval = V{datasize}(tt);
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem{esize}(eaddr, accdesc) = rval[e*:esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
        end;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.372 ST4 (single structure)

Store single 4-element structure from one lane of four registers

This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

(FEAT_AdvSIMD)

31	30	29						23	22	21	20				17	16	15			13	12	11	10	9			5	4			0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0		0	x	x		1	S	size		Rn				Rt			
									L	R						o2				opcode											

Encoding for the 8-bit variant

Applies when `(opcode == 001)`

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant

Applies when `(opcode == 011 && size == x0)`

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant

Applies when `(opcode == 101 && size == 00)`

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

Encoding for the 64-bit variant

Applies when `(opcode == 101 && S == 0 && size == 01)`

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-index

(FEAT_AdvSIMD)

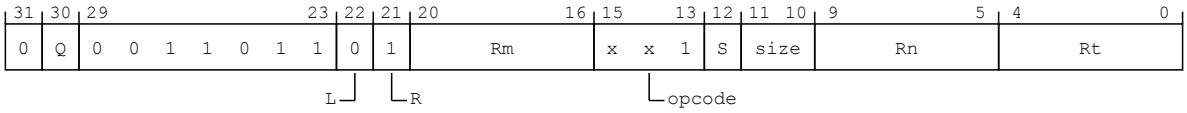

Encoding for the 8-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 001)`

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

Encoding for the 8-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 001)`

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

Encoding for the 16-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

Encoding for the 32-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant

Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

Encoding for the 64-bit, register offset variant

Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<Vt3>

Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<Vt4>

Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<index>

For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem{esize}(eaddr, accdesc) = rval[index*:esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.373 STBFADD, STBFADDL

Atomic BFloat16 floating-point add, without return

This instruction atomically loads a 16-bit value from memory, adds it to the BFloat16 value held in a register, and stores the result back to memory.

- `STBFADDL` stores to memory with release semantics.
- `STBFADD` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

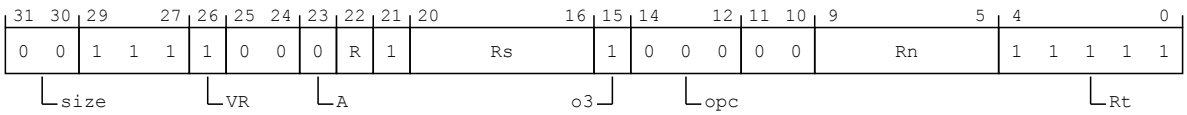

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STBFADD  <Hs>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(R == 1)`

```
STBFADDL  <Hs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 16;
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFADD, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.374 STBFMAX, STBFMAXL

Atomic BFloat16 floating-point maximum, without return

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 maximum with the value held in a register, and stores the result back to memory.

- `STBFMAXL` stores to memory with release semantics.
- `STBFMAX` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

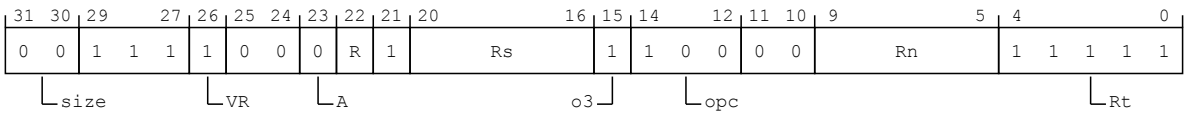

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STBFMAX  <Hs>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(R == 1)`

```
STBFMAXL  <Hs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 16;
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMAX, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.375 STBFMAXNM, STBFMAXNML

Atomic BFloat16 floating-point maximum number, without return

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 maximum number with the value held in a register, and stores the result back to memory.

- `STBFMAXNML` stores to memory with release semantics.
- `STBFMAXNM` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

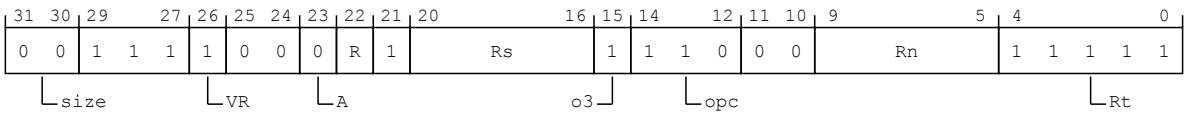

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STBFMAXNM  <Hs>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(R == 1)`

```
STBFMAXNML  <Hs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 16;
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMAXNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.376 STBFMIN, STBFMINL

Atomic BFloat16 floating-point minimum, without return

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 minimum with the value held in a register, and stores the result back to memory.

- `STBFMINL` stores to memory with release semantics.
- `STBFMIN` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

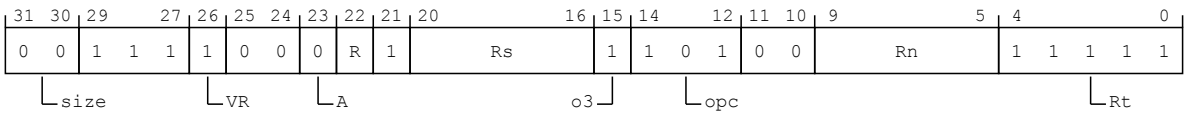

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STBFMIN  <Hs>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(R == 1)`

```
STBFMINL  <Hs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 16;
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMIN, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.377 STBFMINNM, STBFMINNML

Atomic BFloat16 floating-point minimum number, without return

This instruction atomically loads a 16-bit value from memory, calculates the BFloat16 minimum number with the value held in a register, and stores the result back to memory.

- `STBFMINNML` stores to memory with release semantics.
- `STBFMINNM` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	0	R	1	Rs					1	1	1	1	0	0	Rn					1	1	1	1	1
size					VR			A								o3	opc										Rt				

Encoding for the No memory ordering variant

Applies when `(R == 0)`

```
STBFMINNM  <Hs>, [<Xn|SP>]
```

Encoding for the Release variant

Applies when `(R == 1)`

```
STBFMINNML  <Hs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 16;
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_BFMINNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.378 STFADD, STFADDL

Atomic floating-point add, without return

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, performs a floating-point add with the value held in a register, and stores the result back to memory.

- `STFADDL` stores to memory with release semantics.
- `STFADD` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

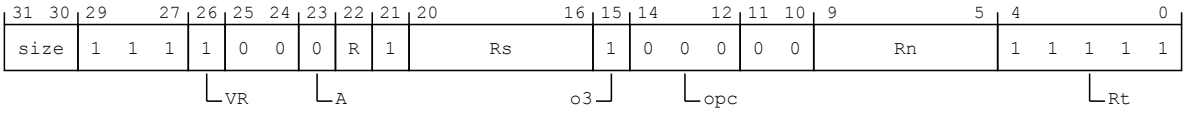

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && R == 0)`

```
STFADD  <Hs>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && R == 1)`

```
STFADDL  <Hs>, [<Xn|SP>]
```

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STFADD  <Ss>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && R == 1)`

```
STFADDL  <Ss>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STFADD  <Ds>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision release variant

Applies when `(size == 11 && R == 1)`

```
STFADDL  <Ds>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPADD, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.379 STFMAX, STFMAXL

Atomic floating-point maximum, without return

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point maximum with the value held in a register, and stores the result back to memory.

- `STFMAXL` stores to memory with release semantics.
- `STFMAX` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

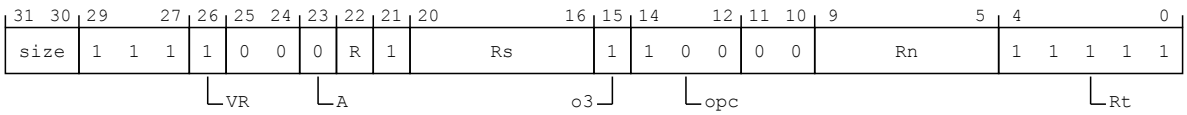

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && R == 0)`

```
STFMAX  <Hs>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && R == 1)`

```
STFMAXL  <Hs>, [<Xn|SP>]
```

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STFMAX  <Ss>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && R == 1)`

```
STFMAXL  <Ss>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STFMAX  <Ds>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision release variant

Applies when `(size == 11 && R == 1)`

```
STFMAXL  <Ds>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMAX, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.380 STFMAXNM, STFMAXNML

Atomic floating-point maximum number, without return

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point maximum number with the value held in a register, and stores the result back to memory.

- `STFMAXNML` stores to memory with release semantics.
- `STFMAXNM` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

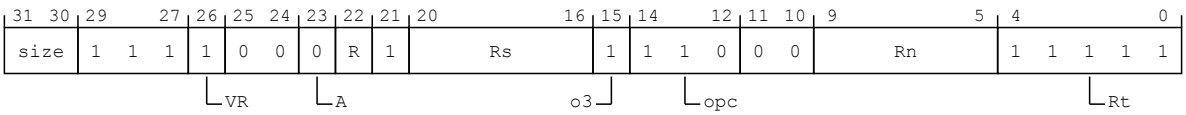

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && R == 0)`

```
STFMAXNM  <Hs>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && R == 1)`

```
STFMAXNML  <Hs>, [<Xn|SP>]
```

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STFMAXNM  <Ss>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && R == 1)`

```
STFMAXNML  <Ss>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STFMAXNM  <Ds>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision release variant

Applies when `(size == 11 && R == 1)`

```
STFMAXNML  <Ds>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMAXNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.381 STFMIN, STFMINL

Atomic floating-point minimum, without return

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point minimum with the value held in a register, and stores the result back to memory.

- `STFMINL` stores to memory with release semantics.
- `STFMIN` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

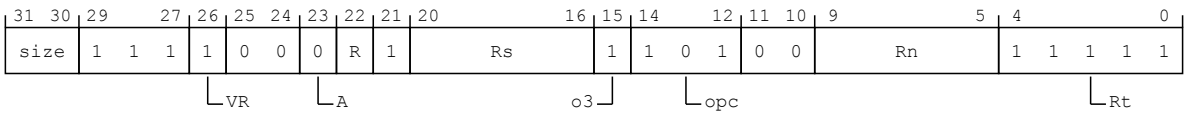

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && R == 0)`

```
STFMIN  <Hs>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && R == 1)`

```
STFMINL  <Hs>, [<Xn|SP>]
```

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STFMIN  <Ss>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && R == 1)`

```
STFMINL  <Ss>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STFMIN  <Ds>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision release variant

Applies when `(size == 11 && R == 1)`

```
STFMINL  <Ds>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMIN, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.382 STFMINNM, STFMINNML

Atomic floating-point minimum number, without return

This instruction atomically loads a 16-bit, 32-bit, or 64-bit value from memory, calculates the floating-point minimum number with the value held in a register, and stores the result back to memory.

- `STFMINNML` stores to memory with release semantics.
- `STFMINNM` has no release semantics.

This instruction:

- Disables alternative floating-point behaviors, as if FPCR.AH is 0.
- Generates only the default NaN, as if FPCR.DN is 1.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Floating-point

(FEAT_LSFE)

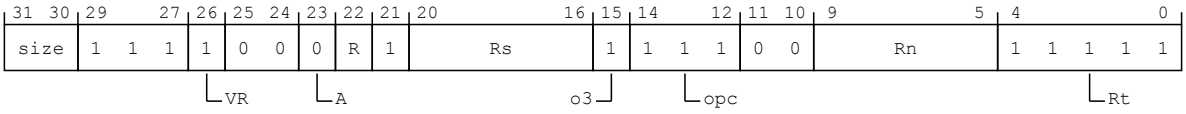

Encoding for the Half-precision no memory ordering variant

Applies when `(size == 01 && R == 0)`

```
STFMINNM  <Hs>, [<Xn|SP>]
```

Encoding for the Half-precision release variant

Applies when `(size == 01 && R == 1)`

```
STFMINNML  <Hs>, [<Xn|SP>]
```

Encoding for the Single-precision no memory ordering variant

Applies when `(size == 10 && R == 0)`

```
STFMINNM  <Ss>, [<Xn|SP>]
```

Encoding for the Single-precision release variant

Applies when `(size == 10 && R == 1)`

```
STFMINNML  <Ss>, [<Xn|SP>]
```

Encoding for the Double-precision no memory ordering variant

Applies when `(size == 11 && R == 0)`

```
STFMINNM  <Ds>, [<Xn|SP>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision release variant

Applies when `(size == 11 && R == 1)`

```
STFMINNML  <Ds>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSFE) then EndOfDecode(Decode_UNDEF); end;

let s : integer = UInt(Rs);
let n : integer = UInt(Rn);

let datasize : integer{} = 8 << UInt(size);
let acquire : boolean = FALSE;
let release : boolean = R == '1';
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Hs>

Is the 16-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Ss>

Is the 32-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<Ds>

Is the 64-bit name of the SIMD&FP register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
var value : bits(datasize);
var data : bits(datasize);
let accdesc : AccessDescriptor = CreateAccDescFPAtomicOp(MemAtomicOp_FPMINNM, acquire,
                                                         release, tagchecked);

value = V{datasize}(s);
if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

let comparevalue : bits(datasize) = ARBITRARY : bits(datasize); // Irrelevant when not executing CAS
data = MemAtomic{datasize}(address, comparevalue, value, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.383 STL1 (SIMD&FP)

Store-release a single-element structure from one lane of one register

This instruction stores the specified element of a SIMD&FP register to memory.

The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about addressing modes, see Load/Store addressing modes.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

64-bit

(FEAT_AdvSIMD && FEAT_LRCPC3)

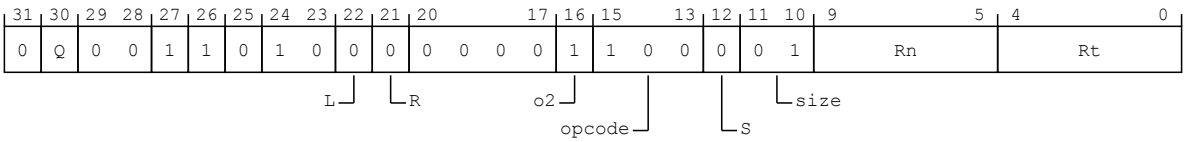

Encoding

```
STL1  { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_LRCPC3) then
    EndOfDecode(Decode_UNDEF);
end;
var t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = ARBITRARY : integer;
let wback : boolean = FALSE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Assembler Symbols

<Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<index>

Is the element index, encoded in “Q”.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Shared Decode

```
var scale : bits(2) = opcode[2:1];
let selem : integer = UInt(opcode[0]::R) + 1;
var replicate : boolean = FALSE;
var index : integer;

case scale of
    when '11' =>
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        scale = size;
        replicate = TRUE;
    when '00' =>
        index = UInt(Q::S::size);     // B[0-15]
    when '01' =>
        if size[0] == '1' then EndOfDecode(Decode_UNDEF); end;
        index = UInt(Q::S::size[1]);  // H[0-7]
    when '10' =>
        if size[1] == '1' then EndOfDecode(Decode_UNDEF); end;
        if size[0] == '0' then
            index = UInt(Q::S);       // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF); end;
            index = UInt(Q);          // D[0-1]
            scale = '11';
        end;
end;

let datasize : integer{} = 64 << UInt(Q);
let esize : integer{} = 8 << UInt(scale);
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

var address : bits(64);
var eaddr : bits(64);
var rval : bits(128);
var element : bits(esize);
var offs : bits(64) = Zeros{64};
let ebytes : integer{} = esize DIV 8;

let accdesc : AccessDescriptor = CreateAccDescASIMDAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1 do
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem{esize}(eaddr, accdesc);
        // replicate to fill 128- or 64-bit register
        V{datasize}(t) = Replicate{datasize}(element);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    end;
else
    // load/store one element per register
    for s = 0 to selem-1 do
        rval = V{128}(t);
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem{esize}(eaddr, accdesc) = rval[index*:esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
    end;
end;
if wback then
    if m != 31 then
        offs = X{64}(m);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.384 STLUR (SIMD&FP)

Store-release SIMD&FP register (unscaled offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

The instruction has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset

(FEAT_FP && FEAT_LRCPC3)

31 30	29 24	23 22	21	20 12	11 10	9 5	4 0
size	0 1 1 1 0 1	x 0	0	imm9	1 0	Rn	Rt
		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 00)`

```
STLUR  <Bt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 00)`

```
STLUR  <Ht>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 00)`

```
STLUR  <St>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 00)`

```
STLUR  <Dt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 10)`

```
STLUR  <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LRCPC3) then
    EndOfDecode(Decode_UNDEF);
end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
var address : bits(64);

let accdesc : AccessDescriptor = CreateAccDescASIMDAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = V{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.385 STNP (SIMD&FP)

Store pair of SIMD&FP registers, with non-temporal hint

This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Signed offset

(FEAT_FP)

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	1	0 0 0	0	imm7	Rt2	Rn	Rt
		VR		L				

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STNP  <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
STNP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
STNP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Assembler Symbols

<St1>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<St2>

Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<Dt1>

Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt2>

Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let scale : integer{} = 2 + (UInt(opc) as integer{0..2});
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = n != 31;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

var data : bits(2*datasize);
if BigEndian(accdesc.acctype) then
    data = V{datasize}(t) :: V{datasize}(t2);
else
    data = V{datasize}(t2) :: V{datasize}(t);
end;
Mem{2*datasize}(address, accdesc) = data;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.386 STP (SIMD&FP)

Store pair of SIMD&FP registers

This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_FP)

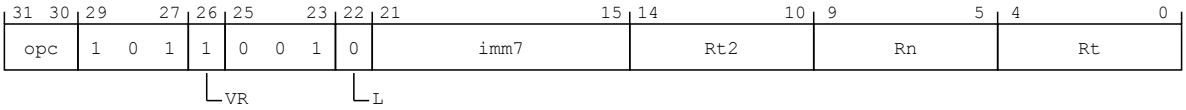

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <St1>, <St2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
STP  <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
STP  <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;

let wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_FP)

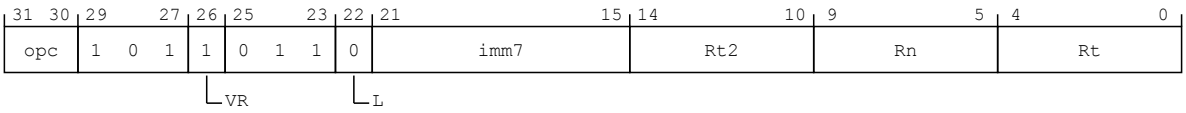

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <St1>, <St2>, [<Xn|SP>, #<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
STP  <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
STP  <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;

let wback : boolean = TRUE;
let postindex : boolean = FALSE;
```

Signed offset

(FEAT_FP)

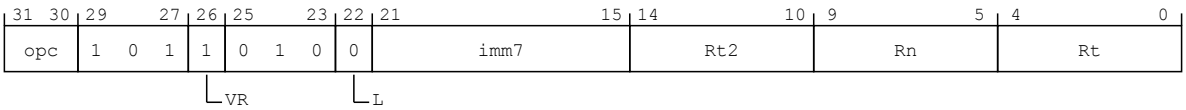

Encoding for the 32-bit variant

Applies when `(opc == 00)`

```
STP  <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant

Applies when `(opc == 01)`

```
STP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant

Applies when `(opc == 10)`

```
STP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;

let wback : boolean = FALSE;
let postindex : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<St1>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<St2>

Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Post-index 128-bit” and “Pre-index 128-bit” variants: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “Signed offset 128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<Dt1>

Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt2>

Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let scale : integer{} = 2 + (UInt(opc) as integer{0..2});
let datasize : integer{} = 8 << scale;
let offset : bits(64) = LSL(SignExtend{64}(imm7), scale);
let tagchecked : boolean = wback || n != 31;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let privileged : boolean = PSTATE.EL != EL0;
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                    tagchecked, privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

var data : bits(2*datasize);
if BigEndian(accdesc.acctype) then
    data = V{datasize}(t) :: V{datasize}(t2);
else
    data = V{datasize}(t2) :: V{datasize}(t);
end;

Mem{2*datasize}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.387 STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index, and Unsigned offset

Post-index

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	0	imm9	0 1	Rn	Rt
		VR		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 00)`

```
STR  <Bt>, [<Xn|SP>], #<simm>
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 00)`

```
STR  <Ht>, [<Xn|SP>], #<simm>
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 00)`

```
STR  <St>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 00)`

```
STR  <Dt>, [<Xn|SP>], #<simm>
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 10)`

```
STR  <Qt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let wback : boolean = TRUE;
let postindex : boolean = TRUE;
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-index

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	0	imm9	1 1	Rn	Rt
		VR		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 00)`

```
STR  <Bt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 00)`

```
STR  <Ht>, [<Xn|SP>, #<simm>]!
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 00)`

```
STR  <St>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 00)`

```
STR  <Dt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 10)`

```
STR  <Qt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let wback : boolean = TRUE;
let postindex : boolean = FALSE;
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Unsigned offset

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
size	1 1 1	1	0 1	x 0	imm12	Rn	Rt
		VR		opc			

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 00)`

```
STR  <Bt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 00)`

```
STR  <Ht>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 00)`

```
STR  <St>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 00)`

```
STR  <Dt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 10)`

```
STR  <Qt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let wback : boolean = FALSE;
let postindex : boolean = FALSE;
let offset : bits(64) = LSL(ZeroExtend{64}(imm12), scale);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the signed immediate byte offset, in the range -256 to 255, encoded in the “imm9” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<pimm>

For the “8-bit” variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “16-bit” variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the “imm12” field as <pimm>/2.

For the “32-bit” variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the “imm12” field as <pimm>/4.

For the “64-bit” variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

For the “128-bit” variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the “imm12” field as <pimm>/16.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = wback || n != 31;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

Mem{datasize}(address, accdesc) = V{datasize}(t);

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.388 STR (register, SIMD&FP)

Store SIMD&FP register (register offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

SIMD&FP registers

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	1	Rm	option	S	1 0	Rn	Rt
		VR		opc							

Encoding for the 8-bit with extended register offset variant

Applies when `(size == 00 && opc == 00 && option != 011)`

```
STR  <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 8-bit with shifted register offset variant

Applies when `(size == 00 && opc == 00 && option == 011)`

```
STR  <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 00)`

```
STR  <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 00)`

```
STR  <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 00)`

```
STR  <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 10)`

```
STR  <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if option[1] == '0' then EndOfDecode(Decode_UNDEF); end;         // sub-word index
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let extend_type : ExtendType = DecodeRegExtend(option);
let shift : integer{} = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm>

When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm>

When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend>

For the “8-bit with extended register offset” variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the “128-bit”, “16-bit”, “32-bit”, and “64-bit” variants: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the “8-bit with extended register offset” and “8-bit with shifted register offset” variants: is the index shift amount, it must be `#0`, encoded in “S” as 0 if omitted, or as 1 if present.

For the “16-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	<amount>
0	#0
1	#1

For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the “128-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
```

Operation

```
AArch64_CheckFPEnabled();
let offset : bits(64) = ExtendReg{}(m, extend_type, shift);
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = V{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.389 STTNP (SIMD&FP)

Store unprivileged pair of SIMD&FP registers, with non-temporal hint

This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

Signed offset

(FEAT_FP && FEAT_LSUI)

31 30	29 28	27	26	25	24 23	22	21 15	14 10	9 5	4 0
1 1	1 0	1	1	0	0 0	0	imm7	Rt2	Rn	Rt
opc			VR			L				

Encoding

```
STTNP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = TRUE;
let datasize : integer{} = 128;
let offset : bits(64) = LSL(SignExtend{64}(imm7), 4);
let tagchecked : boolean = n != 31;
```

Assembler Symbols

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

var data : bits(2*datasize);
if BigEndian(accdesc.acctype) then
    data = V{datasize}(t) :: V{datasize}(t2);
else
    data = V{datasize}(t2) :: V{datasize}(t);
end;
Mem{2*datasize}(address, accdesc) = data;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.390 STTP (SIMD&FP)

Store unprivileged pair of SIMD&FP registers

This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Explicit Memory effects produced by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2().{E2H, TGE} is '11'.

Otherwise, the Explicit Memory effects operate with the restrictions determined by the Exception level at which the instruction is executed.

It has encodings from 3 classes: Post-index, Pre-index, and Signed offset

Post-index

(FEAT_FP && FEAT_LSUI)

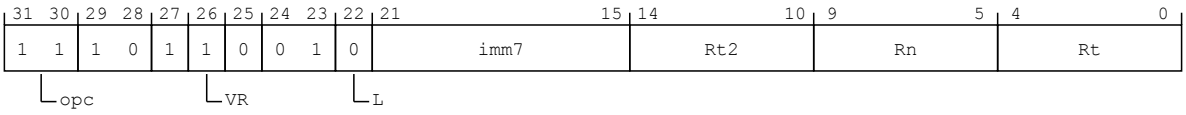

Encoding

```
STTP  <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;

let wback : boolean = TRUE;
let postindex : boolean = TRUE;
```

Pre-index

(FEAT_FP && FEAT_LSUI)

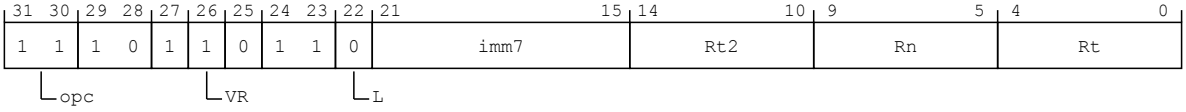

Encoding

```
STTP  <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;

let wback : boolean = TRUE;
let postindex : boolean = FALSE;
```

Signed offset

(FEAT_FP && FEAT_LSUI)

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
1	1	1	0	1	1	0	1	0	0	imm7	Rt2	Rn	Rt
opc					VR				L				

Encoding

```
STTP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP) || !IsFeatureImplemented(FEAT_LSUI) then
    EndOfDecode(Decode_UNDEF);
end;

let wback : boolean = FALSE;
let postindex : boolean = FALSE;
```

Assembler Symbols

<Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Load/store register pair (post-indexed)” and “Load/store register pair (pre-indexed)” variants: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.

For the “Load/store register pair (offset)” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

Shared Decode

```
let t : integer{} = UInt(Rt);
let t2 : integer{} = UInt(Rt2);
let n : integer{} = UInt(Rn);
let nontemporal : boolean = FALSE;
let datasize : integer{} = 128;
let offset : bits(64) = LSL(SignExtend{64}(imm7), 4);
let tagchecked : boolean = wback || n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);
let dbytes : integer{} = datasize DIV 8;

let privileged : boolean = AArch64_IsUnprivAccessPriv();
let ispair : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                    tagchecked, privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

var data : bits(2*datasize);
if BigEndian(accdesc.acctype) then
    data = V{datasize}(t) :: V{datasize}(t2);
else
    data = V{datasize}(t2) :: V{datasize}(t);
end;

Mem{2*datasize}(address, accdesc) = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.391 STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset

(FEAT_FP)

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	0	imm9	0 0	Rn	Rt
		VR		opc					

Encoding for the 8-bit variant

Applies when `(size == 00 && opc == 00)`

```
STUR  <Bt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 16-bit variant

Applies when `(size == 01 && opc == 00)`

```
STUR  <Ht>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 32-bit variant

Applies when `(size == 10 && opc == 00)`

```
STUR  <St>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant

Applies when `(size == 11 && opc == 00)`

```
STUR  <Dt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 128-bit variant

Applies when `(size == 00 && opc == 10)`

```
STUR  <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if opc[1] == '1' && size != '00' then EndOfDecode(Decode_UNDEF); end;
let scale : integer{} = if opc[1] == '1' then 4 else UInt(size);
let offset : bits(64) = SignExtend{}(imm9);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Bt>

Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<Ht>

Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<St>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Dt>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<Qt>

Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Shared Decode

```
let t : integer{} = UInt(Rt);
let n : integer{} = UInt(Rn);
let datasize : integer{} = 8 << scale;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
```

Operation

```
AArch64_CheckFPEnabled();
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked,
                                                    privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

address = AddressAdd(address, offset, accdesc);

Mem{datasize}(address, accdesc) = V{datasize}(t);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.392 SUB (vector)

Subtract (vector)

This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	1	1	1	1	1	1	0	1	1	1	Rm	1	0	0	0	0	1	Rn	Rd
		U						size				opcode							

Encoding

```
SUB  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
0	Q	1	0	1	1	1	0	size		1	Rm	1	0	0	0	0	1	Rn	Rd
		U										opcode							

Encoding

```
SUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    element1 = operand1[e*:esize];
    element2 = operand2[e*:esize];
    result[e*:esize] = element1 - element2;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.393 SUBHN, SUBHN2

Subtract returning high narrow

This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see RSUBHN.

The `SUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1	Rm	0 1	1	0	0 0	Rn	Rd
		U								o1				

Encoding

```
SUBHN{2}  <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
let round : boolean = FALSE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(2*datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:(2*esize)]);
    element2 = UInt(operand2[e*:(2*esize)]);
    sum = element1 - element2;
    sum = RShr(sum, esize, round);
    result[e*:esize] = sum[esize-1:0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Vpart{datasize}(d, part) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.394 SUDOT (by element)

Dot product with signed and unsigned integers (vector, by element)

This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)

31	30	29	28	27 25	24	23	22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	0	0	L	M	Rm	1 1 1 1	H	0	Rn	Rd
		U				US					opcode				

Encoding

```
SUDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_I8MM) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(M::Rm);
let d : integer = UInt(Rd);
let i : integer = UInt(H::L);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 32;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the immediate index of a group of four 8-bit values, in the range 0 to 3, encoded in the “H:L” fields.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    var res : bits(32) = operand3[e*:32];
    for b = 0 to 3 do
        let element1 : integer = SInt(operand1[(4 * e + b)*:8]);
        let element2 : integer = UInt(operand2[(4 * i + b)*:8]);
        res = res + element1 * element2;
    end;
    result[e*:32] = res;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.395 SUQADD

Signed saturating accumulate of unsigned value

This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
SUQADD  <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
SUQADD  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

let operand2 : bits(datasize) = V{}(d);
var op1 : integer;
var op2 : integer;
var sat : boolean;

for e = 0 to elements-1 do
    op1 = UInt(operand[e*:esize]);
    op2 = SInt(operand2[e*:esize]);
    (result[e*:esize], sat) = SatQ{esize}(op1 + op2, unsigned);
    if sat then FPSR().QC = '1'; end;
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.396 SXTL, SXTL2

Signed extend long

This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SXTL` instruction extracts the source vector from the lower half of the source register. The `SXTL2` instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSHLL, SSHLL2. This means:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Vector

(FEAT_AdvSIMD)

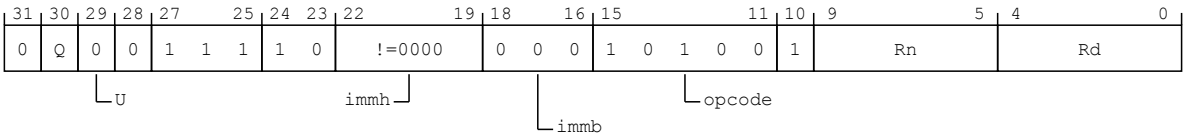

Encoding

```
SXTL{2}  <Vd>.<Ta>, <Vn>.<Tb>
```

is equivalent to

```
SSHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #0
```

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

Operational Information

The description of SSHLL, SSHLL2 gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.397 TBL

Table vector lookup

This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29					24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	0	Rm		0	len		0	0	0	Rn		Rd	
								op2								op						

Encoding for the Single register table variant

Applies when `(len == 00)`

```
TBL  <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

Encoding for the Two register table variant

Applies when `(len == 01)`

```
TBL  <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Encoding for the Three register table variant

Applies when `(len == 10)`

```
TBL  <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Encoding for the Four register table variant

Applies when `(len == 11)`

```
TBL  <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 8;
let regs : integer{} = UInt(len) + 1;
let is_tbl : boolean = (op == '0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn>

For the “Single register table” variant: is the name of the SIMD&FP table register, encoded in the “Rn” field.

For the “Four register table”, “Three register table”, and “Two register table” variants: is the name of the first SIMD&FP table register, encoded in the “Rn” field.

<Vm>

Is the name of the SIMD&FP index register, encoded in the “Rm” field.

<Vn+1>

Is the name of the second SIMD&FP table register, encoded as “Rn” plus 1 modulo 32.

<Vn+2>

Is the name of the third SIMD&FP table register, encoded as “Rn” plus 2 modulo 32.

<Vn+3>

Is the name of the fourth SIMD&FP table register, encoded as “Rn” plus 3 modulo 32.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let indices : bits(datasize) = V{}(m);
var table : bits(128*regs) = Zeros{};
var result : bits(datasize);
var index : integer;

// Create table from registers
for i = 0 to regs - 1 do
    table[i*:128] = V{128}((n+i) MOD 32);
end;

result = if is_tbl then Zeros{datasize} else V{datasize}(d);
for i = 0 to elements - 1 do
    index = UInt(indices[i*:8]);
    if index < 16 * regs then
        result[i*:8] = table[index*:8];
    end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.398 TBX

Table vector lookup extension

This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29					24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	0	0	1	1	1	0	0	0	0	Rm		0	len		1	0	0	Rn		Rd	
								op2								op						

Encoding for the Single register table variant

Applies when `(len == 00)`

```
TBX  <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

Encoding for the Two register table variant

Applies when `(len == 01)`

```
TBX  <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Encoding for the Three register table variant

Applies when `(len == 10)`

```
TBX  <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Encoding for the Four register table variant

Applies when `(len == 11)`

```
TBX  <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 8;
let regs : integer{} = UInt(len) + 1;
let is_tbl : boolean = (op == '0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn>

For the “Single register table” variant: is the name of the SIMD&FP table register, encoded in the “Rn” field.

For the “Four register table”, “Three register table”, and “Two register table” variants: is the name of the first SIMD&FP table register, encoded in the “Rn” field.

<Vm>

Is the name of the SIMD&FP index register, encoded in the “Rm” field.

<Vn+1>

Is the name of the second SIMD&FP table register, encoded as “Rn” plus 1 modulo 32.

<Vn+2>

Is the name of the third SIMD&FP table register, encoded as “Rn” plus 2 modulo 32.

<Vn+3>

Is the name of the fourth SIMD&FP table register, encoded as “Rn” plus 3 modulo 32.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let indices : bits(datasize) = V{}(m);
var table : bits(128*regs) = Zeros{};
var result : bits(datasize);
var index : integer;

// Create table from registers
for i = 0 to regs - 1 do
    table[i*:128] = V{128}((n+i) MOD 32);
end;

result = if is_tbl then Zeros{datasize} else V{datasize}(d);
for i = 0 to elements - 1 do
    index = UInt(indices[i*:8]);
    if index < 16 * regs then
        result[i*:8] = table[index*:8];
    end;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.399 TRN1

Transpose vectors (primary)

This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with `TRN2`, a 2 x 2 matrix can be transposed.

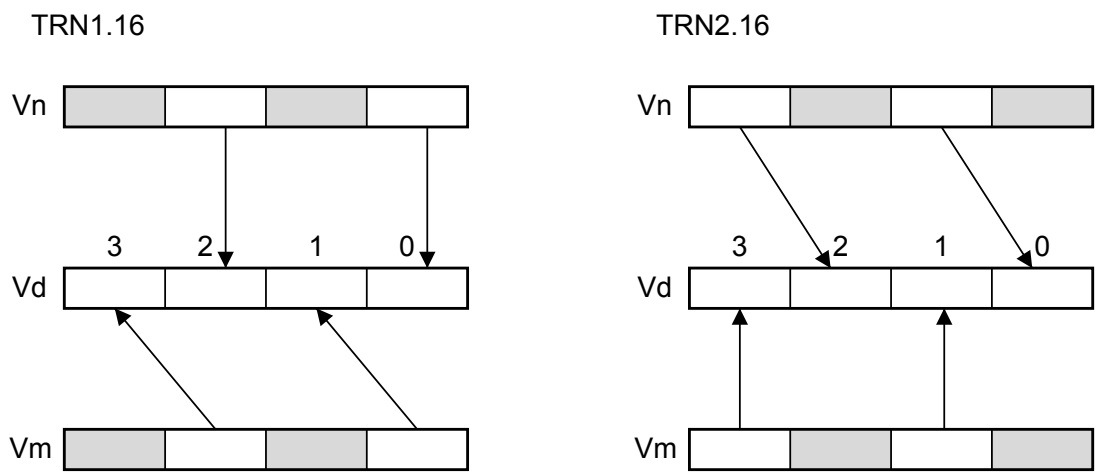

Figure C7-5 TRN1 and TRN2 halfword operations where Q = 0

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1	0	size	0	Rm	0	0	1 0	1 0	Rn	Rd
									op				

Encoding

```
TRN1  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = UInt(op);
let pairs : integer = elements DIV 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for p = 0 to pairs-1 do
    result[(2*p+0)*:esize] = operand1[(2*p+part)*:esize];
    result[(2*p+1)*:esize] = operand2[(2*p+part)*:esize];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.400 TRN2

Transpose vectors (secondary)

This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with `TRN1`, a 2 x 2 matrix can be transposed.

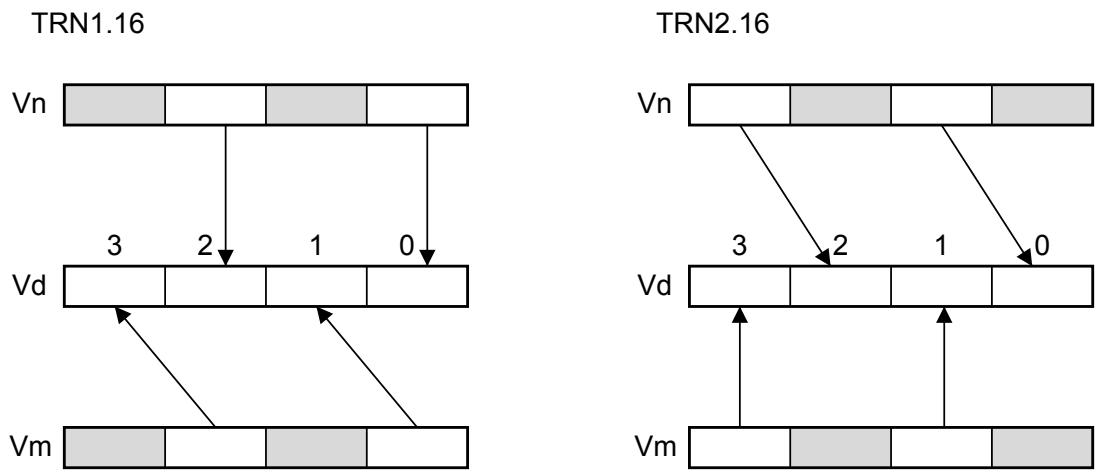

Figure C7-6 TRN1 and TRN2 halfword operations where Q = 0

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1	0	size	0	Rm	0	1	1 0	1 0	Rn	Rd
									op				

Encoding

```
TRN2  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = UInt(op);
let pairs : integer = elements DIV 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for p = 0 to pairs-1 do
    result[(2*p+0)*:esize] = operand1[(2*p+part)*:esize];
    result[(2*p+1)*:esize] = operand2[(2*p+part)*:esize];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.401 UABA

Unsigned absolute difference and accumulate

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 1 1 1	1	1	Rn	Rd
		U								ac			

Encoding

```
UABA  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize) = V{datasize}(d);
var element1 : integer;
var element2 : integer;
var absdiff : bits(esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.402 UABAL, UABAL2

Unsigned absolute difference and accumulate long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABAL` instruction extracts each source vector from the lower half of each source register. The `UABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 1	0	1	0 0	Rn	Rd
		U								op				

Encoding

```
UABAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize) = V{2*datasize}(d);
var element1 : integer;
var element2 : integer;
var absdiff : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    absdiff = Abs(element1-element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + absdiff;
end;
V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.403 UABD

Unsigned absolute difference (vector)

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

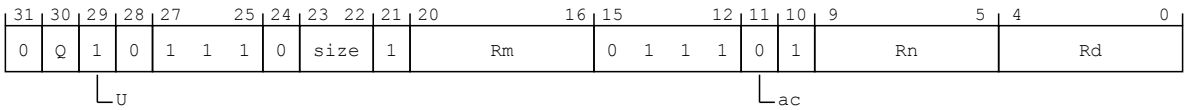

Encoding

```
UABD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize) = Zeros{datasize};
var element1 : integer;
var element2 : integer;
var absdiff : bits(esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.404 UABDL, UABDL2

Unsigned absolute difference long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABDL` instruction extracts each source vector from the lower half of each source register. The `UABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 1	1	1	0 0	Rn	Rd
		U								op				

Encoding

```
UABDL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize) = Zeros{2*datasize};
var element1 : integer;
var element2 : integer;
var absdiff : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    absdiff = Abs(element1-element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + absdiff;
end;
V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.405 UADALP

Unsigned add and accumulate long pairwise

This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

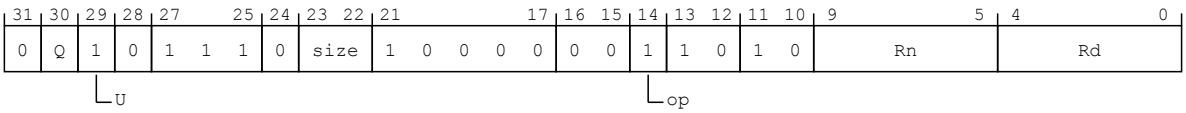

Encoding

```
UADALP  <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV (2 * esize);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize)  = V{}(d);

var sum : bits(2*esize);
var op1 : integer;
var op2 : integer;

for e = 0 to elements-1 do
    op1 = UInt(operand[(2*e+0)*:esize]);
    op2 = UInt(operand[(2*e+1)*:esize]);
    sum = (op1+op2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + sum;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.406 UADDL, UADDL2

Unsigned add long (vector)

This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UADDL` instruction extracts each source vector from the lower half of each source register. The `UADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0	0	0	0 0	Rn	Rd
		U								o1				

Encoding

```
UADDL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    sum = element1 + element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.407 UADDLP

Unsigned add long pairwise

This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

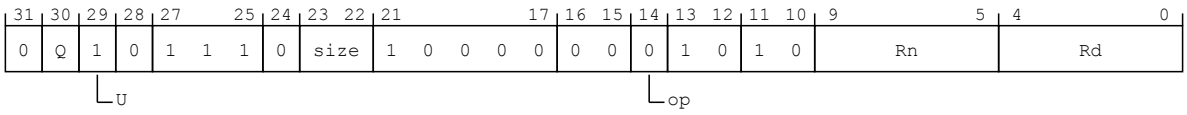

Encoding

```
UADDLP  <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV (2 * esize);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

var sum : bits(2*esize);
var op1 : integer;
var op2 : integer;

for e = 0 to elements-1 do
    op1 = UInt(operand[(2*e+0)*:esize]);
    op2 = UInt(operand[(2*e+1)*:esize]);
    sum = (op1+op2)[2*esize-1:0];
    result[e*:(2*esize)] = sum;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.408 UADDLV

Unsigned sum long across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1 1 0 0 0	0 0 0 1 1	1 0	Rn	Rd
		U						opcode			

Encoding

```
UADDLV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var opelt : bits(esize) = operand[0+:esize];
var sum : integer = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1 do
    opelt = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    sum = sum + element;
end;

V{2*esize}(d) = sum[2*esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.409 UADDW, UADDW2

Unsigned add wide

This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The `UADDW` instruction extracts vector elements from the lower half of the second source register. The `UADDW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0	0	1	0 0	Rn	Rd
		U								o1				

Encoding

```
UADDW{2}  <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:(2*esize)]);
    element2 = UInt(operand2[e*:esize]);
    sum = element1 + element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.410 UCVTF (vector, fixed-point)

Unsigned fixed-point convert to floating-point (vector)

This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	1	1	1	1 1 1	1 0	!=0000	immb	1 1 1 0 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
UCVTF  <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = esize;
let elements : integer = 1;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	1 0	!=0000	immb	1 1 1 0 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
UCVTF  <Vd>.<T>, <Vn>.<T>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let fracbits : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<fbits>

For the “Scalar” variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the “Vector” variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FixedToFP{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.411 UCVTF (vector, integer)

Unsigned integer convert to floating-point (vector)

This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision, Scalar single-precision and double-precision, Vector half-precision, and Vector single-precision and double-precision

Scalar half-precision

(FEAT_AdvSIMD && FEAT_FP16)

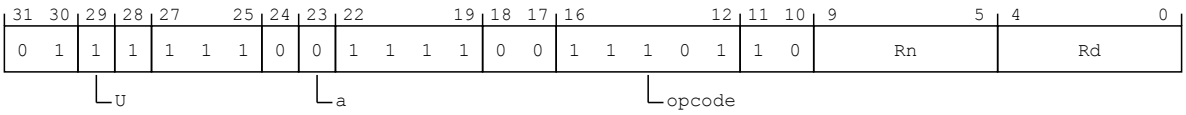

Encoding

```
UCVTF  <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let esize : integer{} = 16;
let datasize : integer{} = esize;
let elements : integer = 1;

let unsigned : boolean = TRUE;
```

Scalar single-precision and double-precision

(FEAT_AdvSIMD)

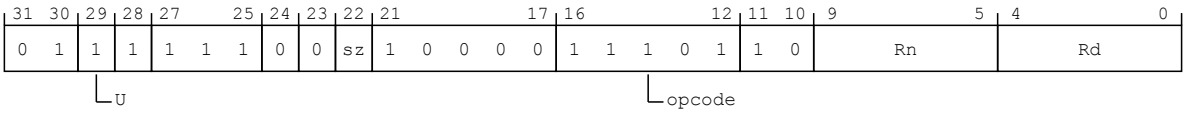

Encoding

```
UCVTF  <V><d>, <V><n>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = TRUE;
```

Vector half-precision

(FEAT_AdvSIMD && FEAT_FP16)

31	30	29	28	27 25	24	23	22 19	18 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	1 1 1 1	0 0	1 1 1 0 1	1 0	Rn	Rd
		U				a			opcode			

Encoding

```
UCVTF  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let esize : integer{} = 16;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = TRUE;
```

Vector single-precision and double-precision

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23	22	21 17	16 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	0	sz	1 0 0 0 0	1 1 1 0 1	1 0	Rn	Rd
		U							opcode			

Encoding

```
UCVTF  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if sz::Q == '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 && IsFeatureImplemented(FEAT_FPRCVT) then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;

let operand : bits(datasize) = V{}(n);

let merge : boolean = elements == 1 && IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};
let rounding : FPRounding = FPRoundingMode(FPCR());
let fracbits : integer = 0;
var element : bits(esize);

for e = 0 to elements-1 do
    element = operand[e*:esize];
    result[e*:esize] = FixedToFP{esize, esize}(element, fracbits, unsigned, FPCR(), rounding);
end;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.412 UCVTF (scalar, fixed-point)

Unsigned fixed-point convert to floating-point (scalar)

This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	24	23 22	21	20 19	18	16	15	10	9	5	4	0
sf	0	0	1 1 1 1 0		ftype	0	0 0	0 1 1		scale		Rn		Rd	
		S				rmode		opcode							

Encoding for the 32-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
UCVTF  <Hd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
UCVTF  <Hd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
UCVTF  <Sd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
UCVTF  <Sd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
UCVTF  <Dd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
UCVTF  <Dd>, <Xn>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;
if sf == '0' && scale[5] == '0' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let decode_fltsize : integer{} = 8 << UInt(ftype XOR '10');

let fracbits : integer = 64 - UInt(scale);

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<fbits>

For the “32-bit to double-precision”, “32-bit to half-precision”, and “32-bit to single-precision” variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus “scale”.

For the “64-bit to double-precision”, “64-bit to half-precision”, and “64-bit to single-precision” variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus “scale”.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Operation

```
AArch64_CheckFPEnabled();

let merge : boolean = IsMerging(FPCR());
let fltsize : integer{} = if merge then 128 else decode_fltsize;
var fltval : bits(fltsize) = if merge then V{fltsize}(d) else Zeros{fltsize};
let intval : bits(intsize) = X{}(n);
let rounding : FPRounding = FPRoundingMode(FPCR());

fltval[0+:decode_fltsize] = FixedToFP{decode_fltsize, intsize}(intval, fracbits, unsigned,
                                                               FPCR(), rounding);

V{fltsize}(d) = fltval;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.413 UCVTF (scalar, integer)

Unsigned integer convert to floating-point (scalar)

This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	0	0	0	1	1	0	0	0	0	0	0	Rn			Rd		
		S									rmode					opcode											

Encoding for the 32-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
UCVTF  <Hd>, <Wn>
```

Encoding for the 32-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
UCVTF  <Sd>, <Wn>
```

Encoding for the 32-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
UCVTF  <Dd>, <Wn>
```

Encoding for the 64-bit to half-precision variant

(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
UCVTF  <Hd>, <Xn>
```

Encoding for the 64-bit to single-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
UCVTF  <Sd>, <Xn>
```

Encoding for the 64-bit to double-precision variant

(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
UCVTF  <Dd>, <Xn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF); end;
if ftype == '10' then EndOfDecode(Decode_UNDEF); end;
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let decode_fltsize : integer{} = 8 << UInt(ftype XOR '10');
let unsigned : boolean    = TRUE;
```

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Wn>

Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Xn>

Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPEnabled();

let fltsize : integer{} = if IsMerging(FPCR()) then 128 else decode_fltsize;

let intval : bits(intsize) = X{}(n);
let rounding : FPRounding  = FPRoundingMode(FPCR());
let fracbits : integer = 0;
var fltval : bits(fltsize) = if IsMerging(FPCR()) then V{fltsize}(d) else Zeros{fltsize};
fltval[0+:decode_fltsize] = FixedToFP{decode_fltsize, intsize}(intval, fracbits, unsigned,
                                                               FPCR(), rounding);

V{fltsize}(d) = fltval;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.414 UCVTF (scalar SIMD&FP)

Unsigned integer convert to floating-point (scalar SIMD&FP)

This instruction converts the unsigned integer value in the SIMD&FP source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Integer

(FEAT_FPRCVT)

31	30	29	28				24	23	22	21	20	19	18		16	15					10	9		5	4		0
sf	0	0	1	1	1	1	0	ftype		1	1	1	1	0	1	0	0	0	0	0	0	Rn			Rd		
		S									rmode					opcode											

Encoding for the 32-bit to half-precision variant

Applies when `(sf == 0 && ftype == 11)`

```
UCVTF  <Hd>, <Sn>
```

Encoding for the 32-bit to double-precision variant

Applies when `(sf == 0 && ftype == 01)`

```
UCVTF  <Dd>, <Sn>
```

Encoding for the 64-bit to half-precision variant

Applies when `(sf == 1 && ftype == 11)`

```
UCVTF  <Hd>, <Dn>
```

Encoding for the 64-bit to single-precision variant

Applies when `(sf == 1 && ftype == 00)`

```
UCVTF  <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FPRCVT) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

let intsize : integer{} = 32 << UInt(sf);
let fltsize : integer{} = 8 << UInt(ftype XOR '10');
let rounding : FPRounding = FPRoundingMode(FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn>

Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Operation

```
AArch64_CheckFPEnabled();

var fltval : bits(fltsize);
var intval : bits(intsize);
let merge : boolean = IsMerging(FPCR());
var result : bits(128) = if merge then V{128}(d) else Zeros{128};

intval = V{intsize}(n);
fltval = FixedToFP{fltsize, intsize}(intval, 0, TRUE, FPCR(), rounding);
result[0+:fltsize] = fltval;

V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.415 UDOT (by element)

Dot product unsigned arithmetic (vector, by element)

This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

31	30	29	28	27 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	1	size	L	M	Rm	1 1 1 0	H	0	Rn	Rd
		U								opcode				

Encoding

```
UDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF); end;
if size  != '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(M::Rm);
let index : integer = UInt(H::L);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the element index, encoded in the “H:L” fields.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
var result : bits(datasize) = V{}(d);
for e = 0 to elements-1 do
    var res : integer = 0;
    var element1, element2 : integer;
    for i = 0 to 3 do
        element1 = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        element2 = UInt(operand2[(4 * index + i)*:(esize DIV 4)]);

        res = res + element1 * element2;
    end;
    result[e*:esize] = result[e*:esize] + res;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.416 UDOT (vector)

Dot product unsigned arithmetic (vector)

This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	0	Rm	1	0 0 1 0	1	Rn	Rd
		U								opcode			

Encoding

```
UDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF); end;
if size != '10' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

result = V{datasize}(d);
for e = 0 to elements-1 do
    var res : integer = 0;
    var element1, element2 : integer;
    for i = 0 to 3 do
        element1 = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        element2 = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = result[e*:esize] + res;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.417 UHADD

Unsigned halving add

This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see URHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 0 0 0	1	Rn	Rd
		U							opcode			

Encoding

```
UHADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    sum = (element1 + element2) >> 1;
    result[e*:esize] = sum[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.418 UHSUB

Unsigned halving subtract

This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 1 0 0	1	Rn	Rd
		U							opcode			

Encoding

```
UHSUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var diff : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    diff = (element1 - element2) >> 1;
    result[e*:esize] = diff[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.419 UMAX

Unsigned maximum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

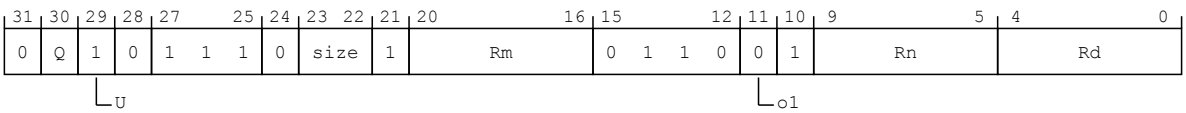

Encoding

```
UMAX  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var max : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    max = Max(element1, element2);
    result[e*:esize] = max[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.420 UMAXP

Unsigned maximum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	1 0 1 0	0	1	Rn	Rd
		U								o1			

Encoding

```
UMAXP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : integer;
var element2 : integer;
var max : integer;

for e = 0 to elements-1 do
    element1 = UInt(concat[(2*e)*:esize]);
    element2 = UInt(concat[((2*e)+1)*:esize]);
    max = Max(element1, element2);
    result[e*:esize] = max[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.421 UMAXV

Unsigned maximum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16	15 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1 1 0 0 0	0	1 0 1 0	1 0	Rn	Rd
		U						op				

Encoding

```
UMAXV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var opelt : bits(esize) = operand[0+:esize];
var max : integer = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1 do
    opelt = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    max = Max(max, element);
end;

V{esize}(d) = max[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.422 UMIN

Unsigned minimum (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

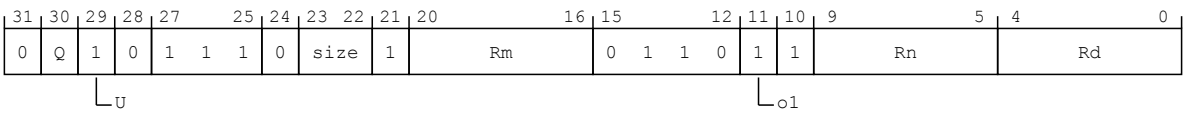

Encoding

```
UMIN  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var min : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    min = Min(element1, element2);
    result[e*:esize] = min[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.423 UMINP

Unsigned minimum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	1 0 1 0	1	1	Rn	Rd
		U								o1			

Encoding

```
UMINP  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
let concat : bits(2*datasize) = operand2::operand1;
var element1 : integer;
var element2 : integer;
var min : integer;

for e = 0 to elements-1 do
    element1 = UInt(concat[(2*e)*:esize]);
    element2 = UInt(concat[((2*e)+1)*:esize]);
    min = Min(element1, element2);
    result[e*:esize] = min[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.424 UMINV

Unsigned minimum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16	15 12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1 1 0 0 0	1	1 0 1 0	1 0	Rn	Rd
		U						op				

Encoding

```
UMINV  <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '100' then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var opelt : bits(esize) = operand[0+:esize];
var min : integer = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1 do
    opelt = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    min = Min(min, element);
end;

V{esize}(d) = min[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.425 UMLAL, UMLAL2 (by element)

Unsigned multiply-add long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register. The `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

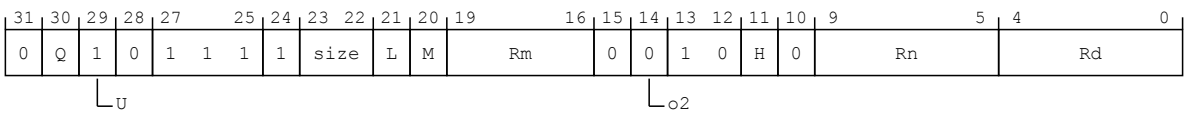

Encoding

```
UMLAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = UInt(operand2[index*:esize]);
var product : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = operand3[e*:(2*esize)] + product;
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.426 UMLAL, UMLAL2 (vector)

Unsigned multiply-add long (vector)

This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register. The `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

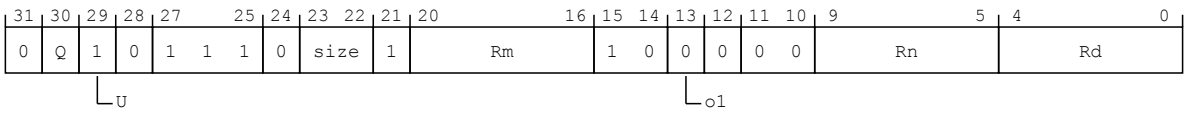

Encoding

```
UMLAL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var accum : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    accum = operand3[e*:(2*esize)] + product;
    result[e*:(2*esize)] = accum;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.427 UMLSL, UMLSL2 (by element)

Unsigned multiply-subtract long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register. The `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

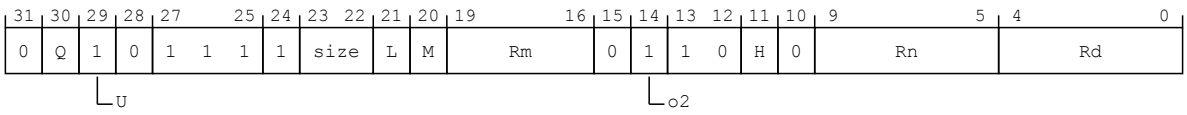

Encoding

```
UMLSL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = UInt(operand2[index*:esize]);
var product : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = operand3[e*:(2*esize)] - product;
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.428 UMLSL, UMLSL2 (vector)

Unsigned multiply-subtract long (vector)

This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMLSL` instruction extracts each source vector from the lower half of each source register. The `UMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	1 0	1	0	0 0	Rn	Rd
		U								o1				

Encoding

```
UMLSL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
let operand3 : bits(2*datasize) = V{}(d);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var product : bits(2*esize);
var accum : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    accum = operand3[e*:(2*esize)] - product;
    result[e*:(2*esize)] = accum;
end;

V{2*datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.429 UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate to 32-bit integer (vector)

This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)

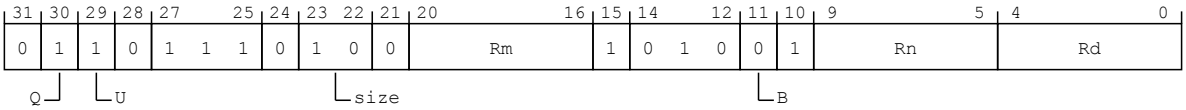

Encoding

```
UMMLA  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_I8MM) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let addend : bits(128)   = V{}(d);

V{128}(d) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Operational Information

Arm expects that the UMMLA (vector) instruction will deliver a peak integer multiply throughput that is at least as high as can be achieved using two UDOT (vector) instructions, with a goal that it should have significantly higher throughput.

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.430 UMOV

Unsigned move vector element to general-purpose register

This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general).

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29	28							21	20				16	15	14				11	10	9				5	4				0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	1	1	1		1	Rn					Rd				
		op																	imm4													

Encoding for the 32-bit variant

Applies when `(Q == 0)`

```
UMOV  <Wd>, <Vn>.<Ts>[<index>]
```

Encoding for the 64-bit variant

Applies when `(Q == 1 && imm5 == x1000)`

```
UMOV  <Xd>, <Vn>.D[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF); end;
let size : integer{} = LowestSetBitNZ(imm5[3:0]);

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << size;
let datasize : integer{} = 32 << UInt(Q);
if datasize == 64 && esize < 64 then EndOfDecode(Decode_UNDEF); end;
if datasize == 32 && esize >= 64 then EndOfDecode(Decode_UNDEF); end;
let index : integer = UInt(imm5[4:size+1]);
let idxdsize : integer{} = 64 << UInt(imm5[4]);
```

Assembler Symbols

<Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ts>

Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<index>

For the “32-bit” variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)

For the “64-bit” variant: is the element index encoded in “imm5<4>”.

<Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Alias	Of variant	Is preferred when
MOV (to general)	32-bit	imm5 IN {'xx100'}
MOV (to general)	64-bit	imm5 IN {'x1000'}

Operation

```
if index == 0 then
    AArch64_CheckFPEnabled();
else
    AArch64_CheckFPAdvSIMDEnabled();
end;
let operand : bits(idxdsize) = V{}(n);

X{datasize}(d) = ZeroExtend{datasize}(operand[index*:esize]);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.431 UMULL, UMULL2 (by element)

Unsigned multiply long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMULL` instruction extracts vector elements from the lower half of the first source register. The `UMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

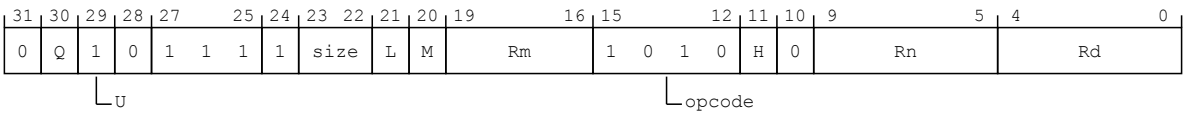

Encoding

```
UMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let idxdsize : integer{} = 64 << UInt(H);
var index : integer;
var Rmhi : bit;
case size of
    when '01' => index = UInt(H::L::M); Rmhi = '0';
    when '10' => index = UInt(H::L); Rmhi = M;
    otherwise => EndOfDecode(Decode_UNDEF);
end;

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rmhi::Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<m>

Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<Ts>

Is an element size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>

Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(idxdsize) = V{}(m);
var result : bits(2*datasize);
var element1 : integer;
let element2 : integer = UInt(operand2[index*:esize]);
var product : bits(2*esize);

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    product = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = product;
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.432 UMULL, UMULL2 (vector)

Unsigned multiply long (vector)

This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMULL` instruction extracts each source vector from the lower half of each source register. The `UMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

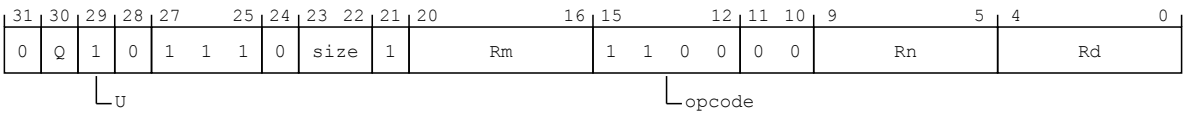

Encoding

```
UMULL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    result[e*:(2*esize)] = (element1 * element2)[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.433 UQADD

Unsigned saturating add

This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	1	1	1	1	1	1	0	size		1	Rm	0	0	0	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
UQADD  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
0	Q	1	0	1	1	1	0	size		1	Rm	0	0	0	0	1	1	Rn	Rd
		U										opcode							

Encoding

```
UQADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    sum = element1 + element2;
    (result[e*:esize], sat) = SatQ{esize}(sum, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.434 UQRSHL

Unsigned saturating rounding shift left (register)

This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see UQSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

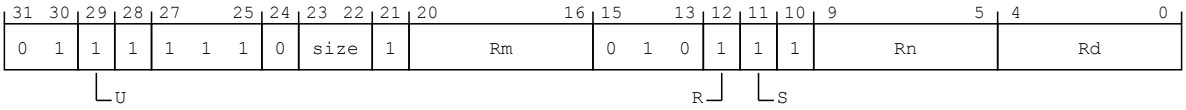

Encoding

```
UQRSHL  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = TRUE;
let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

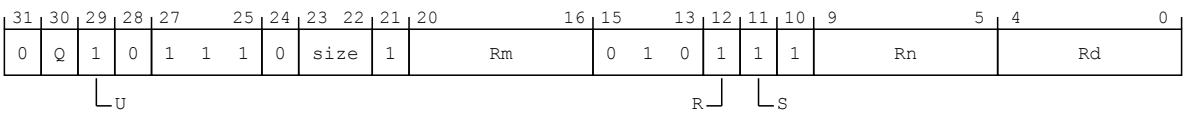

Encoding

```
UQRSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = TRUE;
let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    var element : integer = UInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    (result[e*:esize], sat) = SatQ{esize}(element, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.435 UQRSHRN, UQRSHRN2

Unsigned saturating rounded shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see UQSHRN.

The `UQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `UQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
		U				immh			op			

Encoding

```
UQRSHRN  <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	Q	1	0	1 1 1	1 0	!=0000	immb	1 0 0 1	1	1	Rn	Rd
		U				immh			op			

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
UQRSHRN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = TRUE;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<Tb>
1xxx	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(2*esize) = operand[e*:(2*esize)];
    if unsigned then
        let element : integer = RShr(UInt(opelt), shift, round);
        (result[e*:esize], sat) = UnsignedSatQ{esize}(element);
    else
        let element : integer = RShr(SInt(opelt), shift, round);
        (result[e*:esize], sat) = SignedSatQ{esize}(element);
    end;
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.436 UQSHL (immediate)

Unsigned saturating shift left (immediate)

This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	19	18	16	15	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	1	0	!=0000	immb		0	1	1	1	0	1	Rn	Rd	
		U							immh					op						

Encoding

```
UQSHL  <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = esize;
let elements : integer = 1;
let shift : integer = UInt(immh::immb) - esize;

let src_unsigned : boolean = TRUE;
let dst_unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	25	24	23	22	19	18	16	15	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	1	0	!=0000	immb		0	1	1	1	0	1	Rn	Rd	
		U							immh					op						

Encoding

```
UQSHL  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let shift : integer = UInt(immh::immb) - esize;

let src_unsigned : boolean = TRUE;
let dst_unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “immh”:

immh	<V>
0001	B
001x	H
01xx	S
1xxx	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
    (result[e*:esize], sat) = SatQ{esize}(element << shift, dst_unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.437 UQSHL (register)

Unsigned saturating shift left (register)

This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

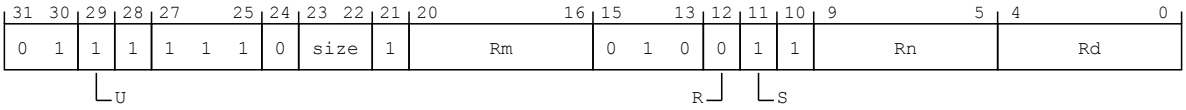

Encoding

```
UQSHL  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = TRUE;
let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

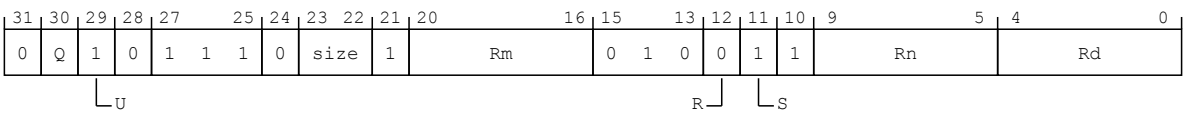

Encoding

```
UQSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let unsigned : boolean = TRUE;
let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    var element : integer = UInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    (result[e*:esize], sat) = SatQ{esize}(element, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.438 UQSHRN, UQSHRN2

Unsigned saturating shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN.

The `UQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `UQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	1	1	1	1 1 1	1 0	!=0000	immb	1 0 0 1	0	1	Rn	Rd
		U				immh			op			

Encoding

```
UQSHRN  <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = esize;
let elements : integer = 1;
let part : integer = 0;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–12	11	10	9–5	4–0
0	Q	1	0	1 1 1	1 0	!=0000	immb	1 0 0 1	0	1	Rn	Rd
		U				immh			op			

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
UQSHRN{2}  <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (2 * esize) - UInt(immh::immb);
let round : boolean = FALSE;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<Tb>
1xxx	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize*2) = V{}(n);
var result : bits(datasize);
var sat : boolean;

for e = 0 to elements-1 do
    let opelt : bits(2*esize) = operand[e*:(2*esize)];
    if unsigned then
        let element : integer = RShr(UInt(opelt), shift, round);
        (result[e*:esize], sat) = UnsignedSatQ{esize}(element);
    else
        let element : integer = RShr(SInt(opelt), shift, round);
        (result[e*:esize], sat) = SignedSatQ{esize}(element);
    end;
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.439 UQSUB

Unsigned saturating subtract

This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	1	1	1	1	1	1	0	size		1	Rm		0	0	1	0	1	1	Rn		Rd	

U

opcode

Encoding

```
UQSUB  <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	Q	1	0	1	1	1	0	size		1	Rm		0	0	1	0	1	1	Rn		Rd	

U

opcode

Encoding

```
UQSUB  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var diff : integer;
var sat : boolean;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    diff = element1 - element2;
    (result[e*:esize], sat) = SatQ{esize}(diff, unsigned);
    if sat then FPSR().QC = '1'; end;
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.440 UQXTN, UQXTN2

Unsigned saturating extract narrow

This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `UQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `UQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0	1	1	1	1 1 1	0	size	1 0 0 0 0	1 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
UQXTN  <Vb><d>, <Va><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let part : integer = 0;
let elements : integer = 1;
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24	23–22	21–17	16–12	11–10	9–5	4–0
0	Q	1	0	1 1 1	0	size	1 0 0 0 0	1 0 1 0 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
UQXTN{2}  <Vd>.<Tb>, <Vn>.<Ta>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Vb>

Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Va>

Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(2*datasize) = V{}(n);
var result : bits(datasize);
var element : bits(2*esize);
var sat : boolean;

for e = 0 to elements-1 do
    element = operand[e*:(2*esize)];
    (result[e*:esize], sat) = SatQ{esize}(UInt(element), unsigned);
    if sat then FPSR().QC = '1'; end;
end;

Vpart{datasize}(d, part) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.441 URECPE

Unsigned reciprocal estimate

This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

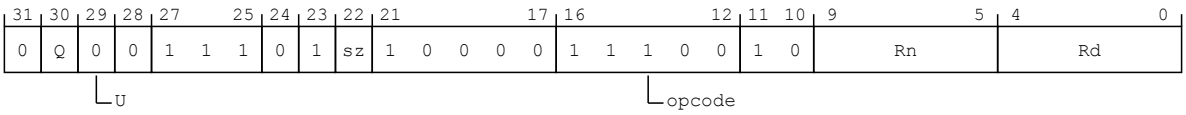

Encoding

```
URECPE  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(32);

for e = 0 to elements-1 do
    element = operand[e*:32];
    result[e*:32] = UnsignedRecipEstimate{32}(element);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.442 URHADD

Unsigned rounding halving add

This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see UHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0 0 1 0	1	Rn	Rd
		U							opcode			

Encoding

```
URHADD  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    sum = (element1 + element2 + 1) >> 1;
    result[e*:esize] = sum[esize-1:0];
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.443 URSHL

Unsigned rounding shift left (register)

This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

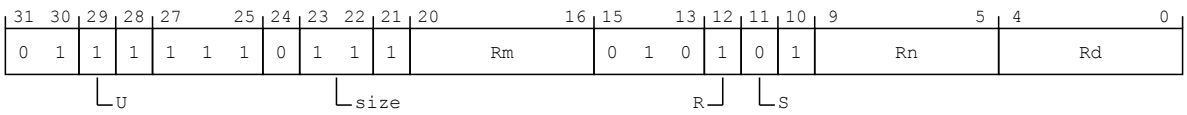

Encoding

```
URSHL  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

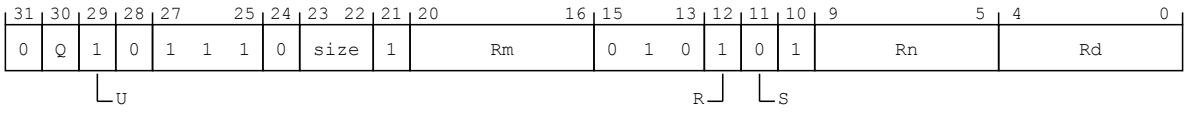

Encoding

```
URSHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    var element : integer = UInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.444 URSHR

Unsigned rounding shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	1	x	x	x	immb			0	0	1	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Encoding

```
URSHR  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!=0000				immb			0	0	1	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Encoding

```
URSHR  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = TRUE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.445 URSQRTE

Unsigned reciprocal square root estimate

This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

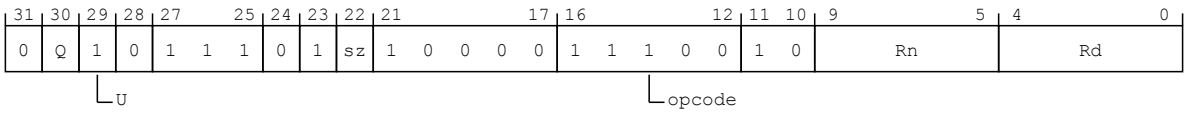

Encoding

```
URSQRTE  <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);

if sz == '1' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);
var element : bits(32);

for e = 0 to elements-1 do
    element = operand[e*:32];
    result[e*:32] = UnsignedRSqrtEstimate{32}(element);
end;

V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.446 URSRA

Unsigned rounding shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

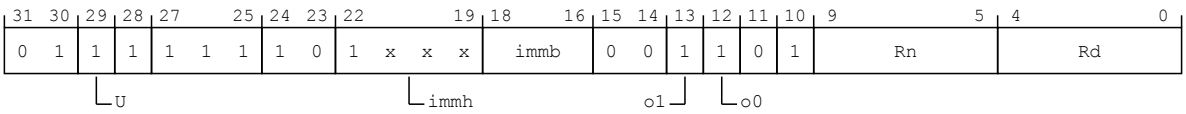

Encoding

```
URSRA  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

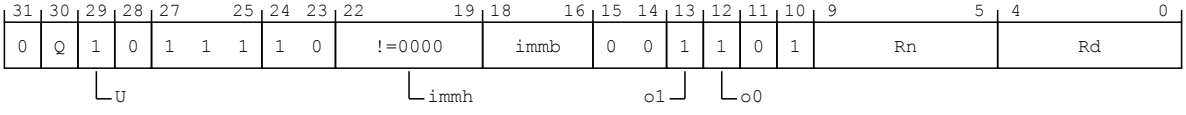

Encoding

```
URSRA  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = TRUE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let operand2 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.447 USDOT (by element)

Dot product with unsigned and signed integers (vector, by element)

This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)

31	30	29	28	27 25	24	23	22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	0	0	1 1 1	1	1	0	L	M	Rm	1 1 1 1	H	0	Rn	Rd
		U				US					opcode				

Encoding

```
USDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_I8MM) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(M::Rm);
let d : integer = UInt(Rd);
let i : integer = UInt(H::L);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 32;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<index>

Is the immediate index of a group of four 8-bit values, in the range 0 to 3, encoded in the “H:L” fields.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(128) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    var res : bits(32) = operand3[e*:32];
    for b = 0 to 3 do
        let element1 : integer = UInt(operand1[(4 * e + b)*:8]);
        let element2 : integer = SInt(operand2[(4 * i + b)*:8]);
        res = res + element1 * element2;
    end;
    result[e*:32] = res;
end;
V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.448 USDOT (vector)

Dot product with unsigned and signed integers (vector)

This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)

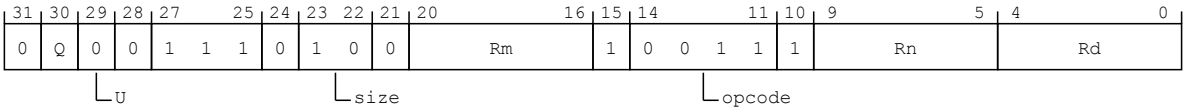

Encoding

```
USDOT  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_I8MM) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV 32;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “Q”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<Tb>
0	8B
1	16B

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
let operand3 : bits(datasize) = V{}(d);
var result : bits(datasize);

for e = 0 to elements-1 do
    var res : bits(32) = operand3[e*:32];
    for b = 0 to 3 do
        let element1 : integer = UInt(operand1[(4 * e + b)*:8]);
        let element2 : integer = SInt(operand2[(4 * e + b)*:8]);
        res = res + element1 * element2;
    end;
    result[e*:32] = res;
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.449 USHL

Unsigned shift left (register)

This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see URSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

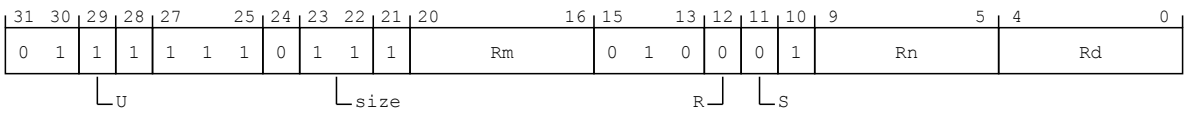

Encoding

```
USHL  D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
```

Vector

(FEAT_AdvSIMD)

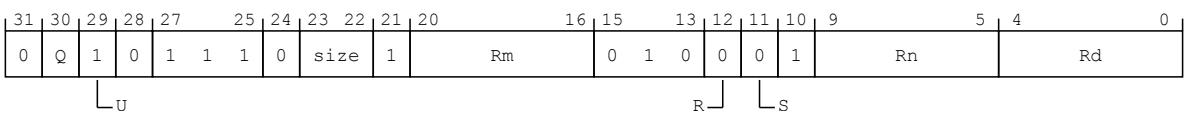

Encoding

```
USHL  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let rounding : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<m>

Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

for e = 0 to elements-1 do
    var element : integer = UInt(operand1[e*:esize]);
    var shift : integer = ShiftSat(SInt(operand2[e*:esize][7:0]), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);
    end;

    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.450 USHLL, USHLL2

Unsigned shift left long (immediate)

This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `USHLL` instruction extracts vector elements from the lower half of the source register. The `USHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	1	0	1 1 1	1 0	!=0000	immb	1 0 1 0 0	1	Rn	Rd
		U				immh		opcode			

Encoding

```
USHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] == '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh[2:0]);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = UInt(immh::immb) - esize;
let unsigned : boolean = TRUE;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift>

Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	RESERVED

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
UXTL, UXTL2	immb == '000' && BitCount(immh) == 1

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = Vpart{}(n, part);
var result : bits(datasize*2);

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
    result[e*:(2*esize)] = (element << shift)[2*esize-1:0];
end;

V{datasize*2}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.451 USHR

Unsigned shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

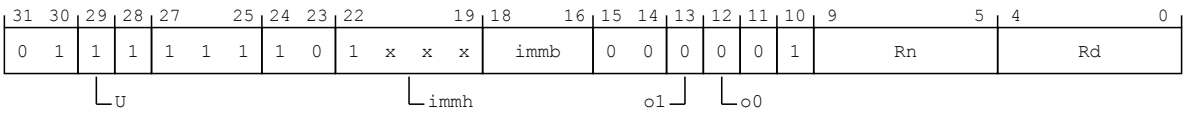

Encoding

```
USHR  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

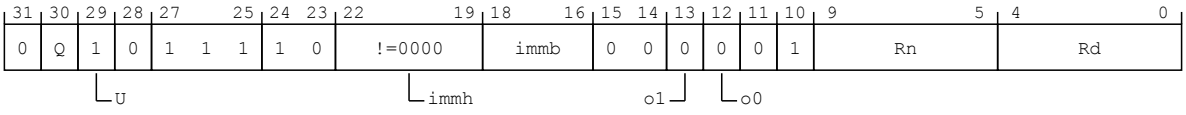

Encoding

```
USHR  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = FALSE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.452 USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate to 32-bit integer (vector)

This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)

31	30	29	28	27 25	24	23 22	21	20 16	15	14 12	11	10	9 5	4 0
0	1	0	0	1 1 1	0	1 0	0	Rm	1	0 1 0	1	1	Rn	Rd
	Q	U				size					B			

Encoding

```
USMMLA  <Vd>.4S, <Vn>.16B, <Vm>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_I8MM) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(128) = V{}(n);
let operand2 : bits(128) = V{}(m);
let addend : bits(128)   = V{}(d);

V{128}(d) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Operational Information

Arm expects that the USMMLA (vector) instruction will deliver a peak integer multiply throughput that is at least as high as can be achieved using two USDOT (vector) instructions, with a goal that it should have significantly higher throughput.

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.453 USQADD

Unsigned saturating accumulate of signed value

This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
USQADD  <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = esize;
let elements : integer = 1;
let unsigned : boolean = TRUE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U													opcode																

Encoding

```
USQADD  <Vd>.<T>, <Vn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize) = V{}(n);
var result : bits(datasize);

let operand2 : bits(datasize) = V{}(d);
var op1 : integer;
var op2 : integer;
var sat : boolean;

for e = 0 to elements-1 do
    op1 = SInt(operand[e*:esize]);
    op2 = UInt(operand2[e*:esize]);
    (result[e*:esize], sat) = SatQ{esize}(op1 + op2, unsigned);
    if sat then FPSR().QC = '1'; end;
end;
V{datasize}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.454 USRA

Unsigned shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	1	1	1	1 1 1	1 0	1 x x x	immb	0 0	0	1	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
USRA  D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3] != '1' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << 3;
let datasize : integer{} = esize;
let elements : integer = 1;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = FALSE;
```

Vector

(FEAT_AdvSIMD)

31	30	29	28	27–25	24–23	22–19	18–16	15–14	13	12	11	10	9–5	4–0
0	Q	1	0	1 1 1	1 0	!=0000	immb	0 0	0	1	0	1	Rn	Rd
		U				immh			o1	o0				

Encoding

```
USRA  <Vd>.<T>, <Vn>.<T>, #<shift>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if immh[3]::Q == '10' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let esize : integer{} = 8 << HighestSetBitNZ(immh);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;

let shift : integer = (esize * 2) - UInt(immh::immb);
let unsigned : boolean = TRUE;
let round : boolean = FALSE;
```

Assembler Symbols

<d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as `128 - UInt(“immh:immb”)`.

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(datasize)  = V{}(n);
let operand2 : bits(datasize) = V{}(d);
var result : bits(datasize);
var element : integer;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand[e*:esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    end;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.455 USUBL, USUBL2

Unsigned subtract long

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The `USUBL` instruction extracts each source vector from the lower half of each source register. The `USUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0	1	0	0 0	Rn	Rd
		U								o1				

Encoding

```
USUBL{2}  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = Vpart{}(n, part);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:esize]);
    element2 = UInt(operand2[e*:esize]);
    sum = element1 - element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.456 USUBW, USUBW2

Unsigned subtract wide

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are unsigned integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The `USUBW` instruction extracts vector elements from the lower half of the second source register. The `USUBW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21	20 16	15 14	13	12	11 10	9 5	4 0
0	Q	1	0	1 1 1	0	size	1	Rm	0 0	1	1	0 0	Rn	Rd
		U								o1				

Encoding

```
USUBW{2}  <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(2*datasize) = V{}(n);
let operand2 : bits(datasize) = Vpart{}(m, part);
var result : bits(2*datasize);
var element1 : integer;
var element2 : integer;
var sum : integer;

for e = 0 to elements-1 do
    element1 = UInt(operand1[e*:(2*esize)]);
    element2 = UInt(operand2[e*:esize]);
    sum = element1 - element2;
    result[e*:(2*esize)] = sum[2*esize-1:0];
end;

V{2*datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.457 UXTL, UXTL2

Unsigned extend long

This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `UXTL` instruction extracts vector elements from the lower half of the source register. The `UXTL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of USHLL, USHLL2. This means:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Vector

(FEAT_AdvSIMD)

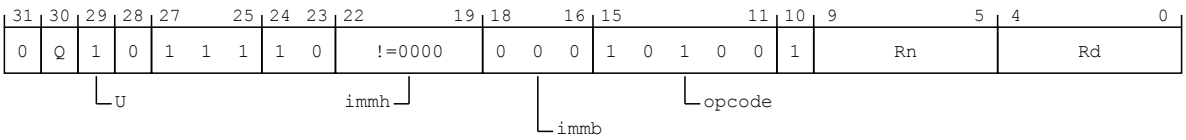

Encoding

```
UXTL{2}  <Vd>.<Ta>, <Vn>.<Tb>
```

is equivalent to

```
USHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #0
```

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Ta>

Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

Operational Information

The description of USHLL, USHLL2 gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.458 UZP1

Unzip vectors (primary)

This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with `UZP2` to de-interleave two vectors.

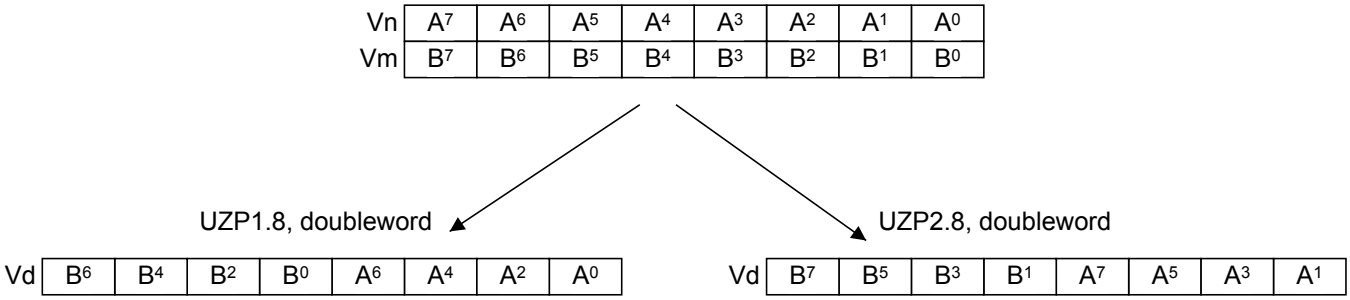

Figure C7-7 UZP1 and UZP2 with the arrangement specifier 8B

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1	0	size	0	Rm	0	0	0 1	1 0	Rn	Rd
									op				

Encoding

```
UZP1  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = UInt(op);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operandl : bits(datasize) = V{}(n);
let operandh : bits(datasize) = V{}(m);
var result : bits(datasize);

let zipped : bits(datasize*2) = operandh::operandl;
for e = 0 to elements-1 do
    result[e*:esize] = zipped[(2*e+part)*:esize];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.459 UZP2

Unzip vectors (secondary)

This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with `UZP1` to de-interleave two vectors.

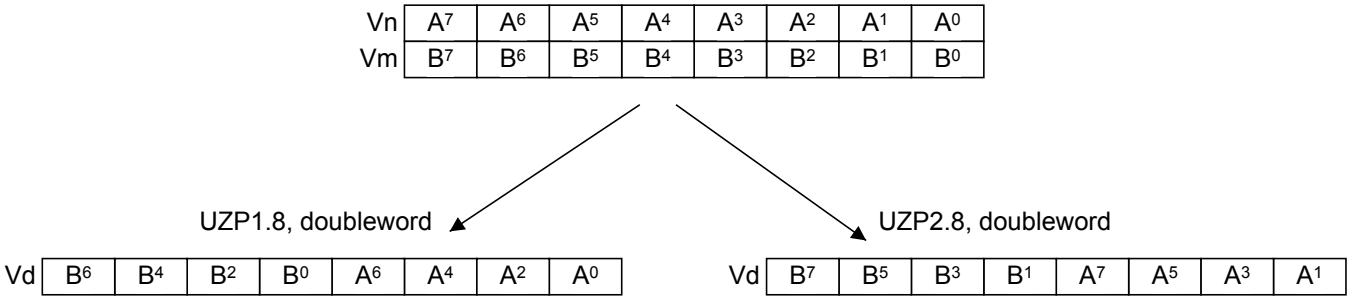

Figure C7-8 UZP1 and UZP2 with the arrangement specifier 8B

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1	0	size	0	Rm	0	1	0 1	1 0	Rn	Rd
									op				

Encoding

```
UZP2  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = UInt(op);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operandl : bits(datasize) = V{}(n);
let operandh : bits(datasize) = V{}(m);
var result : bits(datasize);

let zipped : bits(datasize*2) = operandh::operandl;
for e = 0 to elements-1 do
    result[e*:esize] = zipped[(2*e+part)*:esize];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.460 XAR

Exclusive-OR and rotate

This instruction performs a bitwise exclusive-OR between the 128-bit vector in the first source SIMD&FP register and the 128-bit vector in the second source SIMD&FP register, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

Advanced SIMD

(FEAT_SHA3)

31 28	27 25	24 23	22 21	20 16	15 10	9 5	4 0
1 1 0 0	1 1 1	0 1	0 0	Rm	imm6	Rn	Rd

Encoding

```
XAR  <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
```

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<imm6>

Is a rotation right, encoded in “imm6”.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();

let Vm : bits(128) = V{}(m);
let Vn : bits(128) = V{}(n);
let tmp : bits(128) = Vn XOR Vm;
V{128}(d) = ROR(tmp[127:64], UInt(imm6))::ROR(tmp[63:0], UInt(imm6));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.461 XTN, XTN2

Extract narrow

This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The `XTN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `XTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_AdvSIMD)

31	30	29	28	27 25	24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	0	0	1 1 1	0	size	1 0 0 0 0	1 0 0 1 0	1 0	Rn	Rd
		U						opcode			

Encoding

```
XTN{2}  <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);

let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64;
let part : integer = UInt(Q);
let elements : integer = datasize DIV esize;
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand : bits(2*datasize) = V{}(n);
var result : bits(datasize);
var element : bits(2*esize);

for e = 0 to elements-1 do
    element = operand[e*:(2*esize)];
    result[e*:esize] = element[esize-1:0];
end;
Vpart{datasize}(d, part) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.462 ZIP1

Zip vectors (primary)

This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with `ZIP2` to interleave two vectors.

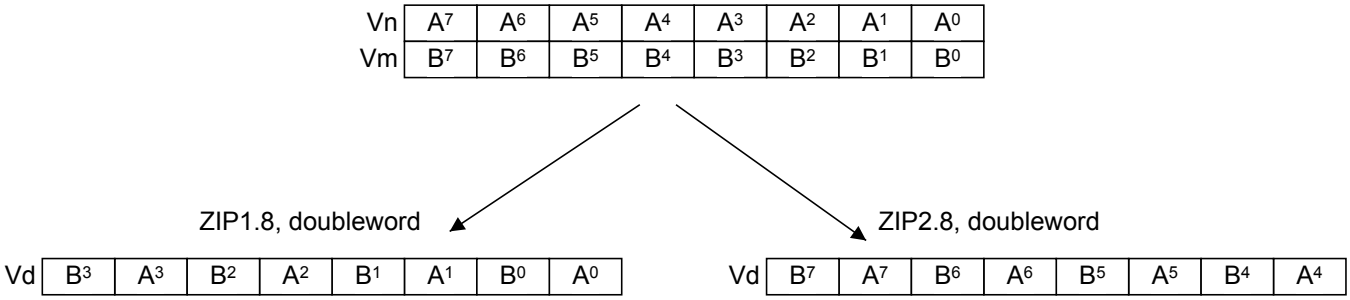

Figure C7-9 ZIP1 and ZIP2 with the arrangement specifier 8B

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1	0	size	0	Rm	0	0	1 1	1 0	Rn	Rd
									op				

Encoding

```
ZIP1  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = UInt(op);
let pairs : integer = elements DIV 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

let base : integer = part * pairs;

for p = 0 to pairs-1 do
    result[(2*p+0)*:esize] = operand1[(base+p)*:esize];
    result[(2*p+1)*:esize] = operand2[(base+p)*:esize];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.463 ZIP2

Zip vectors (secondary)

This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with `ZIP1` to interleave two vectors.

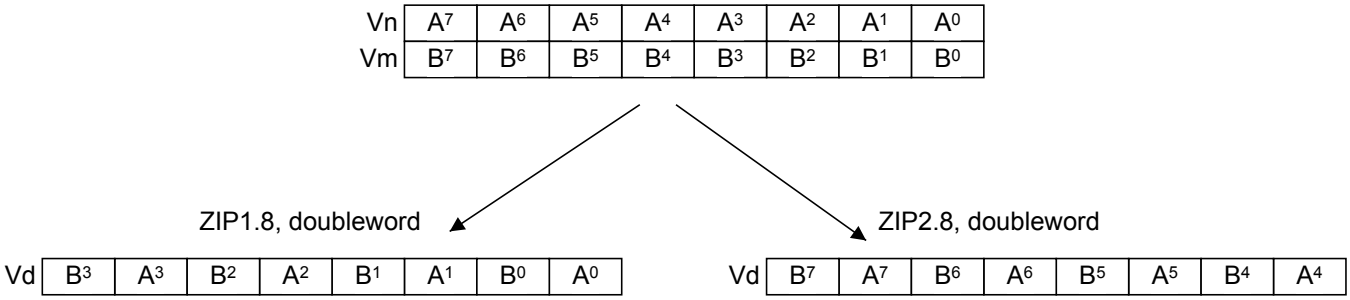

Figure C7-10 ZIP1 and ZIP2 with the arrangement specifier 8B

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

(FEAT_AdvSIMD)

31	30	29 28	27 25	24	23 22	21	20 16	15	14	13 12	11 10	9 5	4 0
0	Q	0 0	1 1 1	0	size	0	Rm	0	1	1 1	1 0	Rn	Rd
									op				

Encoding

```
ZIP2  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF); end;
if size::Q == '110' then EndOfDecode(Decode_UNDEF); end;

let d : integer{} = UInt(Rd);
let n : integer{} = UInt(Rn);
let m : integer{} = UInt(Rm);
let esize : integer{} = 8 << UInt(size);
let datasize : integer{} = 64 << UInt(Q);
let elements : integer = datasize DIV esize;
let part : integer = UInt(op);
let pairs : integer = elements DIV 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64_CheckFPAdvSIMDEnabled();
let operand1 : bits(datasize) = V{}(n);
let operand2 : bits(datasize) = V{}(m);
var result : bits(datasize);

let base : integer = part * pairs;

for p = 0 to pairs-1 do
    result[(2*p+0)*:esize] = operand1[(base+p)*:esize];
    result[(2*p+1)*:esize] = operand2[(base+p)*:esize];
end;

V{datasize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C8
SVE Instruction Descriptions

This chapter describes the SVE instructions. It contains the following sections:

- About the SVE instructions.
- Alphabetical list of SVE instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.1 About the SVE instructions

Alphabetical list of SVE instructions gives full descriptions of the SVE instructions that are in the following instruction groups:

- Loads and store instructions associated with the SVE registers.
- Data processing instructions with SVE registers.

A64 instruction set encoding provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2 Alphabetical list of SVE instructions

This section lists every section in the SVE category of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.1 ABS

Absolute value (predicated)

This instruction calculates the absolute value of the signed integer in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	0	1 1 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
ABS  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	0	1 1 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
ABS  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        var element : integer = SInt(operand[e*:esize]);
        element = Abs(element);
        result[e*:esize] = element[esize-1:0];

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.2 ADCLB

Add with carry long (bottom)

This instruction adds the even-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–14	13–11	10	9–5	4–0
0 1 0	0 0 1 0	1	0	sz	0	Zm	1 1	0 1 0	0 (T)	Zn	Zda

Encoding

```
ADCLB  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (esize * 2);
let operand : bits(VL) = Z{}(n);
let carries : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for p = 0 to pairs-1 do
    let element1 : bits(esize) = result[(2*p + 0)*:esize];
    let element2 : bits(esize) = operand[(2*p + 0)*:esize];
    let carry_in : bit = carries[(2*p + 1)*:esize][0];

    let (res, nzcv) = AddWithCarry{esize}(element1, element2, carry_in);
    let carry_out : bit = nzcv[1];

    result[(2*p + 0)*:esize] = res;
    result[(2*p + 1)*:esize] = ZeroExtend{esize}(carry_out);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.3 ADCLT

Add with carry long (top)

This instruction adds the odd-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–14	13–11	10	9–5	4–0
0 1 0	0 0 1 0	1	0	sz	0	Zm	1 1	0 1 0	1 (T)	Zn	Zda

Encoding

```
ADCLT  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (esize * 2);
let operand : bits(VL) = Z{}(n);
let carries : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for p = 0 to pairs-1 do
    let element1 : bits(esize) = result[(2*p + 0)*:esize];
    let element2 : bits(esize) = operand[(2*p + 1)*:esize];
    let carry_in : bit = carries[(2*p + 1)*:esize][0];

    let (res, nzcv) = AddWithCarry{esize}(element1, element2, carry_in);
    let carry_out : bit = nzcv[1];

    result[(2*p + 0)*:esize] = res;
    result[(2*p + 1)*:esize] = ZeroExtend{esize}(carry_out);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.4 ADD (vectors, predicated)

Add (predicated)

This instruction adds active elements of the first source vector to the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0 0	0 0 0 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
ADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element1 + element2;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.5 ADD (immediate)

Add immediate (unpredicated)

This instruction adds an unsigned immediate to each element of the source vector, and destructively places the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–14	13	12–5	4–0
0 0 1	0 0 1 0	1	size	1	0 0	0 0 0 (opc)	1 1	sh	imm8	Zdn

Encoding

```
ADD  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    result[e*:esize] = element1 + imm;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.6 ADD (vectors, unpredicated)

Add (unpredicated)

This instruction adds all elements of the first source vector to the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

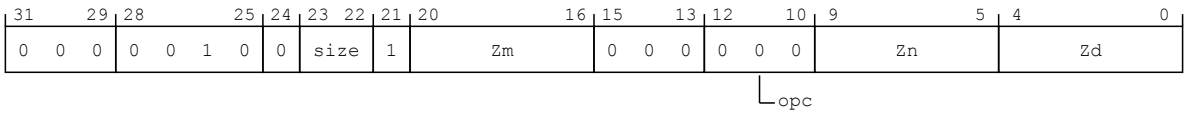

Encoding

```
ADD  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = element1 + element2;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.7 ADDHNB

Add narrow high part (bottom)

This instruction adds each vector element of the first source vector to the corresponding vector element of the second source vector, and places the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

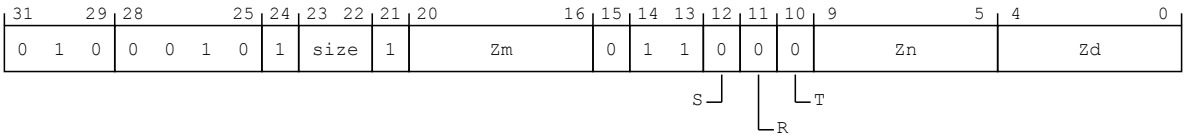

Encoding

```
ADDHNB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = (element1 + element2) >> halfesize;
    result[(2*e + 0)*:halfesize] = res[halfesize-1:0];
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.8 ADDHNT

Add narrow high part (top)

This instruction adds each vector element of the first source vector to the corresponding vector element of the second source vector, and places the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

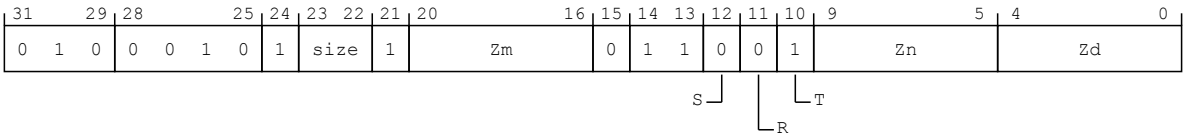

Encoding

```
ADDHNT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = (element1 + element2) >> halfesize;
    result[(2*e + 1)*:halfesize] = res[halfesize-1:0];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.9 ADDP

Add pairwise

This instruction adds pairs of adjacent elements within each source vector, and destructively places the interleaved results in the first source vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	1 0	0 0 (opc)	1 (U)	1 0	1	Pg	Zm	Zdn

Encoding

```
ADDP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[e*:esize];
    else
        if IsEven(e) then
            element1 = UInt(operand1[(e + 0)*:esize]);
            element2 = UInt(operand1[(e + 1)*:esize]);
        else
            element1 = UInt(operand2[(e - 1)*:esize]);
            element2 = UInt(operand2[(e + 0)*:esize]);
        end;
        let res : integer = element1 + element2;
        result[e*:esize] = res[esize-1:0];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.10 ADDPL

Add multiple of predicate register size to scalar register

This instruction adds the current predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and places the result in the 64-bit destination general-purpose register or current stack pointer.

SVE

(FEAT_SVE || FEAT_SME)

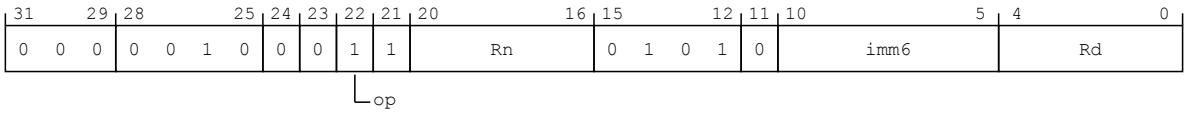

Encoding

```
ADDPL  <Xd|SP>, <Xn|SP>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let imm : integer = SInt(imm6);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is the signed immediate operand, in the range -32 to 31, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let result : bits(64) = operand1 + (imm * (PL DIV 8));

if d == 31 then
    SP{64}() = result;
else
    X{64}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.11 ADDPT (predicated)

Add checked pointer vectors (predicated)

This instruction adds, with pointer check, Active elements of the first source vector to the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE && FEAT_CPA)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	1 1 (size)	0	0 0	1 0 0 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
ADDPT  <Zdn>.D, <Pg>/M, <Zdn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_CPA) then
    EndOfDecode(Decode_UNDEF);
end;
let g : integer =  UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer =  UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 64;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 64) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand1[e*:64];
    let element2 : bits(64) = operand2[e*:64];
    if ActivePredicateElement{PL}(mask, e, 64) then
        let res : bits(64) = element1 + element2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:64] = PointerAddCheck(res, element1);
    else
        result[e*:64] = operand1[e*:64];
    end;

    Z{VL}(dn) = result;
end;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.12 ADDPT (unpredicated)

Add checked pointer vectors (unpredicated)

This instruction adds, with pointer check, all elements of the first source vector to the corresponding elements of the second source vector and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE && FEAT_CPA)

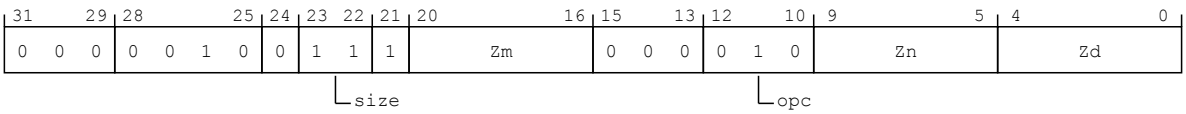

Encoding

```
ADDPT  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_CPA) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand1[e*:64];
    let element2 : bits(64) = operand2[e*:64];
    let res : bits(64) = element1 + element2;
    result[e*:64] = PointerAddCheck(res, element1);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.13 ADDQV

Add reduction of quadword vector segments

This instruction adds the elements of the same position from each 128-bit source vector segment, and places each result into the corresponding element of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as zero.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

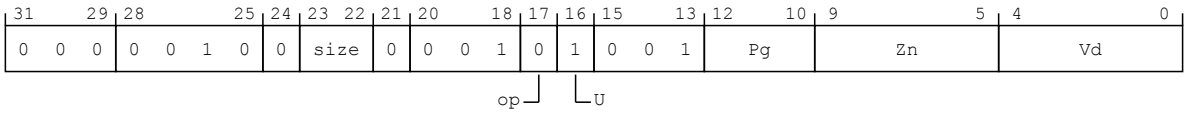

Encoding

```
ADDQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : integer;

for e = 0 to elempersegment-1 do
    dtmp = 0;
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = dtmp + UInt(stmp[e*:esize]);
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.14 ADDVL

Add multiple of vector register size to scalar register

This instruction adds the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and places the result in the 64-bit destination general-purpose register or current stack pointer.

SVE

(FEAT_SVE || FEAT_SME)

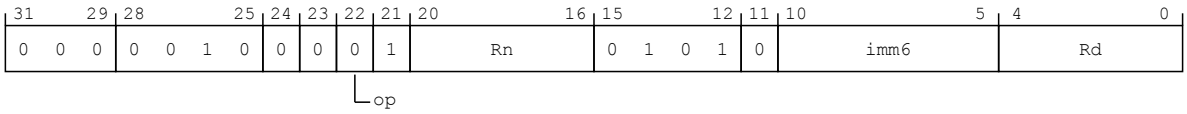

Encoding

```
ADDVL  <Xd|SP>, <Xn|SP>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let imm : integer = SInt(imm6);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is the signed immediate operand, in the range -32 to 31, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let result : bits(64) = operand1 + (imm * (VL DIV 8));

if d == 31 then
    SP{64}() = result;
else
    X{64}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.15 ADR

Calculate vector address

This instruction optionally sign-extends or zero-extends the least significant 32 bits of each element from a vector of offsets or indices in the second source vector, and scales each index by 2, 4, or 8. The scaled indices are then added to a vector of base addresses from the first source vector, and the resulting addresses are placed in the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: Packed offsets, Unpacked 32-bit signed offsets, and Unpacked 32-bit unsigned offsets

Packed offsets

(FEAT_SVE)

31–29	28–25	24	23	22	21	20–16	15–12	11–10	9–5	4–0
0 0 0	0 0 1 0	0	1	sz	1	Zm	1 0 1 0	msz	Zn	Zd

Encoding

```
ADR  <Zd>.<T>, [<Zn>.<T>, <Zm>.<T>{, <mod> <amount>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let osize : integer{} = esize;
let unsigned : boolean = TRUE;
let mbytes : integer{} = 1 << UInt(msz);
```

Unpacked 32-bit signed offsets

(FEAT_SVE)

31–29	28–25	24	23 22	21	20–16	15–12	11–10	9–5	4–0
0 0 0	0 0 1 0	0	0 0	1	Zm	1 0 1 0	msz	Zn	Zd
			opc						

Encoding

```
ADR  <Zd>.D, [<Zn>.D, <Zm>.D, SXTW{<amount>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let osize : integer{} = 32;
let unsigned : boolean = FALSE;
let mbytes : integer{} = 1 << UInt(msz);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Unpacked 32-bit unsigned offsets

(FEAT_SVE)

31 29	28 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	0 1	1	Zm	1 0 1 0	msz	Zn	Zd
			opc						

Encoding

```
ADR  <Zd>.D, [<Zn>.D, <Zm>.D, UXTW{<amount>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let osize : integer{} = 32;
let unsigned : boolean = TRUE;
let mbytes : integer{} = 1 << UInt(msz);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “msz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

msz	<mod>
00	[absent]
x1	LSL
10	LSL

<amount>

Is the index shift amount, encoded in “msz”:

msz	<amount>
00	[absent]
01	#1
10	#2
11	#3

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let base : bits(VL) = Z{}(n);
let offs : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let addr : bits(esize) = base[e*:esize];
    let offselt : bits(osize) = offs[e*:esize][osize-1:0];
    let offset : integer = if unsigned then UInt(offselt) else SInt(offselt);
    result[e*:esize] = addr + (offset * mbytes);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.16 AESD (indexed)

Multi-vector AES single round decryption

This instruction reads a 16-byte state array from each 128-bit segment of the two or four first source vectors, together with a round key from the indexed 128-bit segment of the corresponding 512-bit portion of the second source vector. Each state array undergoes a single round of the AddRoundKey(), InvShiftRows(), and InvSubBytes() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the two or four first source vectors.

When the vector length is less than 512 bits, the most significant bits of the index are ignored to select the indexed 128-bit segment of the second source vector. This instruction is unpredicated.

This instruction is legal when executed in Streaming SVE mode if both FEAT_SSVE_AES and FEAT_SVE_AES2 are implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SVE_AES2)

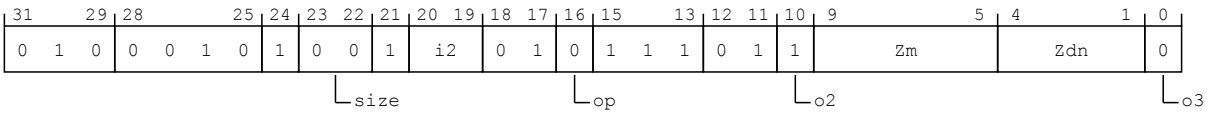

Encoding

```
AESD  { <Zdn1>.B-<Zdn2>.B }, { <Zdn1>.B-<Zdn2>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'0');
var index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SVE_AES2)

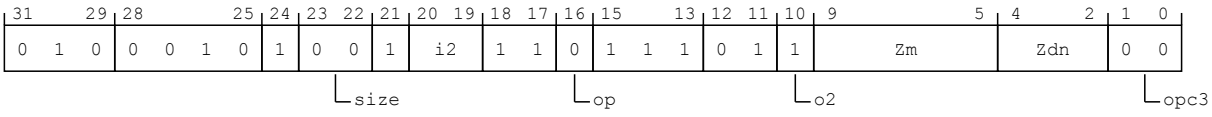

Encoding

```
AESD  { <Zdn1>.B-<Zdn4>.B }, { <Zdn1>.B-<Zdn4>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'00');
var index : integer = UInt(i2);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<index>

Is the round key index, in the range 0 to 3, encoded in the “i2” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL == 128 then index = 0; end;
if VL == 256 then index = index MOD 2; end;
let segments : integer = VL DIV 128;
let operand2 : bits(VL) = Z{}(m);
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn + r);
    for s = 0 to segments-1 do
        let keyindex : integer = (s - (s MOD 4)) + index;
        let res : bits(128) = operand1[s*:128] XOR operand2[keyindex*:128];
        results[[r]][s*:128] = AESInvSubBytes(AESInvShiftRows(res));

    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn + r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.17 AESD (vectors)

AES single round decryption

This instruction reads a 16-byte state array from each 128-bit segment of the first source vector, together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the AddRoundKey(), InvShiftRows(), and InvSubBytes() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is legal when executed in Streaming SVE mode if one of the following is true:

- Both FEAT_SSVE_AES and FEAT_SVE_AES are implemented.
- FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_AES)

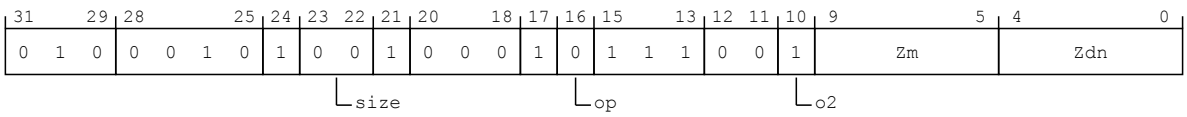

Encoding

```
AESD  <Zdn>.B, <Zdn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

result = operand1 XOR operand2;
for s = 0 to segments-1 do
    result[s*:128] = AESInvSubBytes(AESInvShiftRows(result[s*:128]));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.18 AESDIMC

Multi-vector AES single round decryption and inverse mix columns

This instruction reads a 16-byte state array from each 128-bit segment of the two or four first source vectors, together with a round key from the indexed 128-bit segment of the corresponding 512-bit portion of the second source vector. Each state array undergoes a single round of the AddRoundKey(), InvShiftRows(), InvSubBytes(), and InvMixColumns() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the two or four first source vectors.

When the vector length is less than 512 bits, the most significant bits of the index are ignored to select the indexed 128-bit segment of the second source vector. This instruction is unpredicated.

This instruction is legal when executed in Streaming SVE mode if both FEAT_SSVE_AES and FEAT_SVE_AES2 are implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SVE_AES2)

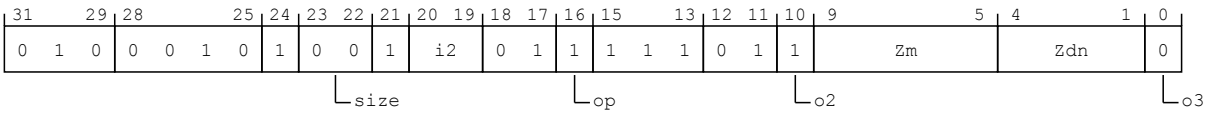

Encoding

```
AESDIMC  { <Zdn1>.B-<Zdn2>.B }, { <Zdn1>.B-<Zdn2>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'0');
var index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SVE_AES2)

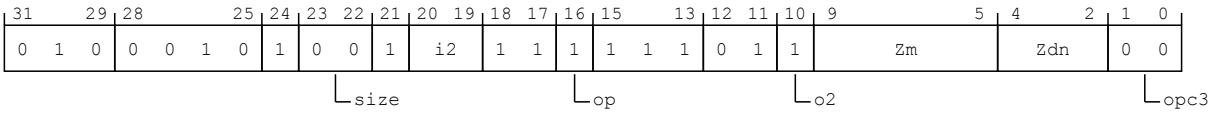

Encoding

```
AESDIMC  { <Zdn1>.B-<Zdn4>.B }, { <Zdn1>.B-<Zdn4>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'00');
var index : integer = UInt(i2);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<index>

Is the round key index, in the range 0 to 3, encoded in the “i2” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL == 128 then index = 0; end;
if VL == 256 then index = index MOD 2; end;
let segments : integer = VL DIV 128;
let operand2 : bits(VL) = Z{}(m);
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn + r);
    for s = 0 to segments-1 do
        let keyindex : integer = (s - (s MOD 4)) + index;
        let res : bits(128) = operand1[s*:128] XOR operand2[keyindex*:128];
        results[[r]][s*:128] = AESInvMixColumns(AESInvSubBytes(AESInvShiftRows(res)));

    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn + r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.19 AESE (indexed)

Multi-vector AES single round encryption

This instruction reads a 16-byte state array from each 128-bit segment of the two or four first source vectors, together with a round key from the indexed 128-bit segment of the corresponding 512-bit portion of the second source vector. Each state array undergoes a single round of the AddRoundKey(), ShiftRows(), and SubBytes() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the two or four first source vectors.

When the vector length is less than 512 bits, the most significant bits of the index are ignored to select the indexed 128-bit segment of the second source vector. This instruction is unpredicated.

This instruction is legal when executed in Streaming SVE mode if both FEAT_SSVE_AES and FEAT_SVE_AES2 are implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SVE_AES2)

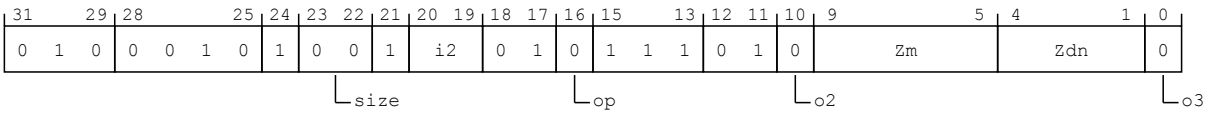

Encoding

```
AESE  { <Zdn1>.B-<Zdn2>.B }, { <Zdn1>.B-<Zdn2>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'0');
var index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SVE_AES2)

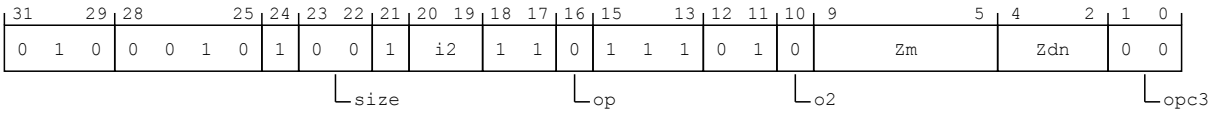

Encoding

```
AESE  { <Zdn1>.B-<Zdn4>.B }, { <Zdn1>.B-<Zdn4>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'00');
var index : integer = UInt(i2);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<index>

Is the round key index, in the range 0 to 3, encoded in the “i2” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL == 128 then index = 0; end;
if VL == 256 then index = index MOD 2; end;
let segments : integer = VL DIV 128;
let operand2 : bits(VL) = Z{}(m);
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn + r);
    for s = 0 to segments-1 do
        let keyindex : integer = (s - (s MOD 4)) + index;
        let res : bits(128) = operand1[s*:128] XOR operand2[keyindex*:128];
        results[[r]][s*:128] = AESSubBytes(AESShiftRows(res));

    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn + r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.20 AESE (vectors)

AES single round encryption

This instruction reads a 16-byte state array from each 128-bit segment of the first source vector together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the AddRoundKey(), ShiftRows(), and SubBytes() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is legal when executed in Streaming SVE mode if one of the following is true:

- Both FEAT_SSVE_AES and FEAT_SVE_AES are implemented.
- FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_AES)

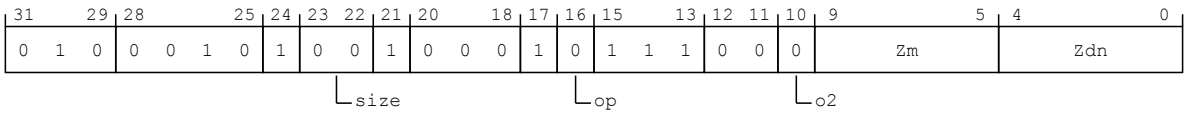

Encoding

```
AESE  <Zdn>.B, <Zdn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

result = operand1 XOR operand2;
for s = 0 to segments-1 do
    result[s*:128] = AESSubBytes(AESShiftRows(result[s*:128]));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.21 AESEMC

Multi-vector AES single round encryption and mix columns

This instruction reads a 16-byte state array from each 128-bit segment of the two or four first source vectors, together with a round key from the indexed 128-bit segment of the corresponding 512-bit portion of the second source vector. Each state array undergoes a single round of the AddRoundKey(), ShiftRows(), SubBytes(), and MixColumns() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the two or four first source vectors.

When the vector length is less than 512 bits, the most significant bits of the index are ignored to select the indexed 128-bit segment of the second source vector. This instruction is unpredicated.

This instruction is legal when executed in Streaming SVE mode if both FEAT_SSVE_AES and FEAT_SVE_AES2 are implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SVE_AES2)

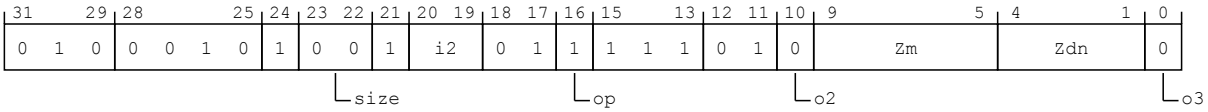

Encoding

```
AESEMC  { <Zdn1>.B-<Zdn2>.B }, { <Zdn1>.B-<Zdn2>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'0');
var index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SVE_AES2)

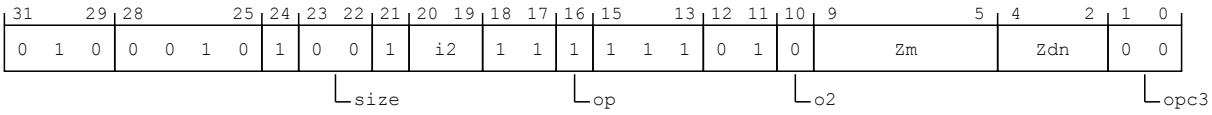

Encoding

```
AESEMC  { <Zdn1>.B-<Zdn4>.B }, { <Zdn1>.B-<Zdn4>.B }, <Zm>.Q[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn::'00');
var index : integer = UInt(i2);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<index>

Is the round key index, in the range 0 to 3, encoded in the “i2” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL == 128 then index = 0; end;
if VL == 256 then index = index MOD 2; end;
let segments : integer = VL DIV 128;
let operand2 : bits(VL) = Z{}(m);
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn + r);
    for s = 0 to segments-1 do
        let keyindex : integer = (s - (s MOD 4)) + index;
        let res : bits(128) = operand1[s*:128] XOR operand2[keyindex*:128];
        results[[r]][s*:128] = AESMixColumns(AESSubBytes(AESShiftRows(res)));

    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn + r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.22 AESIMC

AES inverse mix columns

This instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the InvMixColumns() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is legal when executed in Streaming SVE mode if one of the following is true:

- Both FEAT_SSVE_AES and FEAT_SVE_AES are implemented.
- FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_AES)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 11	10	9 5	4 0
0 1 0	0 0 1 0	1	0 0	1	0 0 0	0 0	1 1 1	0 0	1	0 0 0 0 0	Zdn
			size						op		

Encoding

```
AESIMC  <Zdn>.B, <Zdn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES) then EndOfDecode(Decode_UNDEF); end;
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand : bits(VL) = Z{}(dn);
var result : bits(VL);

for s = 0 to segments-1 do
    result[s*:128] = AESInvMixColumns(operand[s*:128]);
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.23 AESMC

AES mix columns

This instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the MixColumns() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is legal when executed in Streaming SVE mode if one of the following is true:

- Both FEAT_SSVE_AES and FEAT_SVE_AES are implemented.
- FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_AES)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 11	10	9 5	4 0
0 1 0	0 0 1 0	1	0 0	1	0 0 0	0 0	1 1 1	0 0	0	0 0 0 0 0	Zdn
			size						op		

Encoding

```
AESMC  <Zdn>.B, <Zdn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES) then EndOfDecode(Decode_UNDEF); end;
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand : bits(VL) = Z{}(dn);
var result : bits(VL);

for s = 0 to segments-1 do
    result[s*:128] = AESMixColumns(operand[s*:128]);
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.24 AND (predicates)

Bitwise AND predicates

This instruction performs a bitwise AND between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This instruction is used by the alias MOV (predicate, zeroing).

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

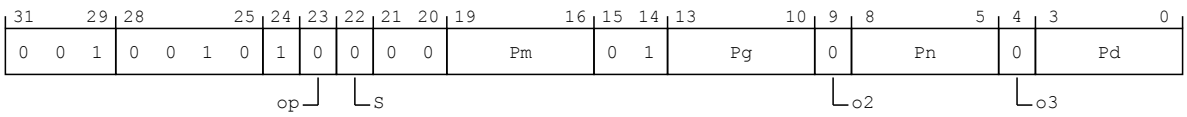

Encoding

```
AND  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Alias	Is preferred when
MOV (predicate, zeroing)	S == '0' && Pn == Pm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 AND element2);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.25 AND (vectors, predicated)

Bitwise AND (predicated)

This instruction performs a bitwise AND between the active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 1	0 1 0 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
AND  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element1 AND element2;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.26 AND (immediate)

Bitwise AND with immediate (unpredicated)

This instruction performs a bitwise AND between an immediate and each 64-bit element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction BIC (immediate).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 18	17 5	4 0
0 0 0	0 0 1 0	1	1 0	0 0	0 0	imm13	Zdn
			opc				

Encoding

```
AND  <Zdn>.<T>, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn);
var imm : bits(64);
(imm, -) = DecodeBitMasks{64}(imm13[12], imm13[5:0], imm13[11:6], TRUE);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand[e*:64];
    result[e*:64] = element1 AND imm;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.27 AND (vectors, unpredicated)

Bitwise AND (unpredicated)

This instruction performs a bitwise AND between the elements of the first source vector and the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	0 0	1	Zm	0 0 1	1 0 0	Zn	Zd
			opc						

Encoding

```
AND  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);

Z{VL}(d) = operand1 AND operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.28 ANDQV

Bitwise AND reduction of quadword vector segments

This instruction performs a bitwise AND between elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as all ones.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 1 1	1 0	0 0 1	Pg	Zn	Vd
						opc				

Encoding

```
ANDQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : bits(esize);

for e = 0 to elempersegment-1 do
    dtmp = Ones{esize};
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = dtmp AND stmp[e*:esize];
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.29 ANDS

Bitwise AND predicates, setting the condition flags

This instruction performs a bitwise AND between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This instruction is used by the alias MOVS (predicated).

Setting the condition flags

(FEAT_SVE || FEAT_SME)

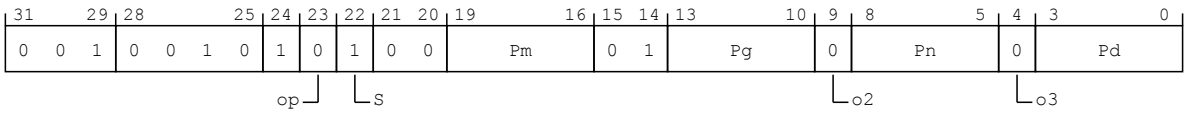

Encoding

```
ANDS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOVS (predicated)	S == '1' && Pn == Pm

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 AND element2);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.30 ANDV

Bitwise AND reduction to scalar

This instruction performs a bitwise AND between the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as all ones.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 1 0	1 0 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
ANDV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(esize) = Ones{esize};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result = result AND operand[e*:esize];

    end;
end;
V{esize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.31 ASR (immediate, predicated)

Arithmetic shift right by immediate (predicated)

This instruction shifts right by immediate, preserving the sign bit, each active element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

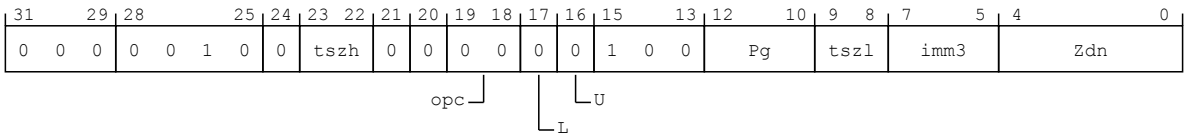

Encoding

```
ASR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let PL : integer{} = VL DIV 8;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = ASR(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.32 ASR (wide elements, predicated)

Arithmetic shift right by 64-bit wide elements (predicated)

This instruction shifts right, preserving the sign bit, active elements of the first source vector by the corresponding overlapping 64-bit elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

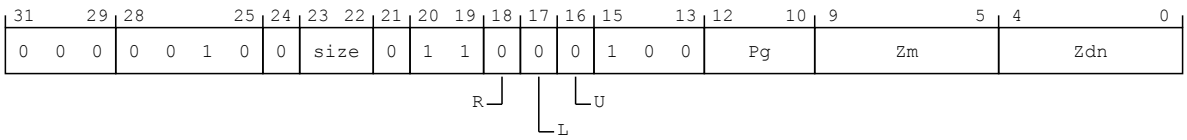

Encoding

```
ASR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
        let shift : integer = Min(UInt(element2), esize);
        result[e*:esize] = ASR(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.33 ASR (vectors)

Arithmetic shift right by vector (predicated)

This instruction shifts right, preserving the sign bit, active elements of the first source vector by the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

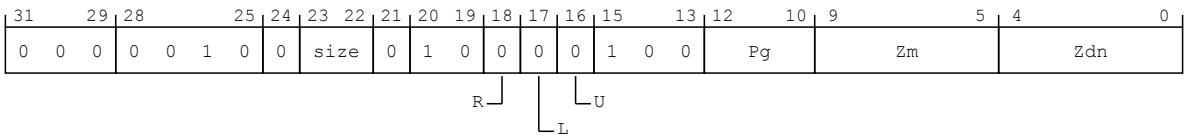

Encoding

```
ASR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let shift : integer = Min(UInt(element2), esize);
        result[e*:esize] = ASR(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.34 ASR (immediate, unpredicated)

Arithmetic shift right by immediate (unpredicated)

This instruction shifts right by immediate, preserving the sign bit, each element of the source vector, and places the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12	11	10	9 5	4 0
0 0 0	0 0 1 0	0	tszh	1	tszl	imm3	1 0 0	1	0	0	Zn	Zd
										U		

Encoding

```
ASR  <Zd>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    result[e*:esize] = ASR(element1, shift);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.35 ASR (wide elements, unpredicated)

Arithmetic shift right by 64-bit wide elements (unpredicated)

This instruction shifts right, preserving the sign bit, all elements of the first source vector by the corresponding overlapping 64-bit elements of the second source vector and places the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	size	1	Zm	1 0 0	0	0	0 (U)	Zn	Zd

Encoding

```
ASR  <Zd>.<T>, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
    let shift : integer = Min(UInt(element2), esize);
    result[e*:esize] = ASR(element1, shift);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.36 ASRD

Arithmetic shift right for divide by immediate (predicated)

This instruction shifts right by immediate, preserving the sign bit, each active element of the source vector, and destructively places the results in the corresponding elements of the source vector. The result rounds toward zero as in a signed division. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

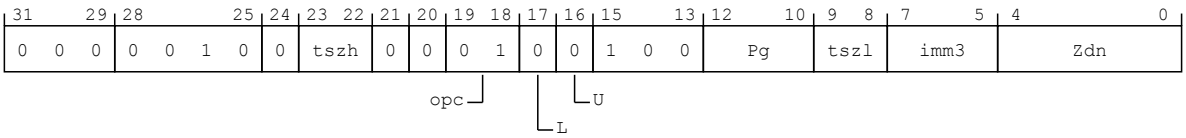

Encoding

```
ASRD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        var element1 : integer = SInt(operand1[e*:esize]);
        if element1 < 0 then
            element1 = element1 + ((1 << shift) - 1);
        end;
        result[e*:esize] = (element1 >> shift)[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.37 ASRR

Reversed arithmetic shift right by vector (predicated)

This instruction reverse shifts right, preserving the sign bit, active elements of the second source vector by the corresponding elements of the first source vector, and destructively places the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

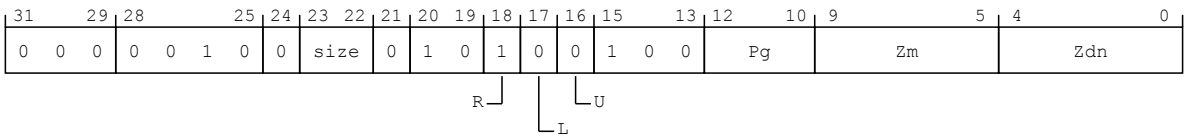

Encoding

```
ASRR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let shift : integer = Min(UInt(element1), esize);
        result[e*:esize] = ASR(element2, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.38 BCAX

Bitwise clear and exclusive-OR

This instruction performs a bitwise AND between elements of the second source vector and the inverted corresponding elements of the third source vector, then an exclusive-OR between the results and the corresponding elements of the first source vector. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	0 1	1	Zm	0 0 1	1 1	0	Zk	Zdn
			opc					o2		

Encoding

```
BCAX  <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let m : integer = UInt(Zm);
let k : integer = UInt(Zk);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the third source scalable vector register, encoded in the “Zk” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(k);

Z{VL}(dn) = operand1 XOR (operand2 AND NOT(operand3));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.39 BDEP

Scatter lower bits into positions selected by bitmask

This instruction scatters the lowest-numbered contiguous bits within each element of the first source vector to the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector, preserving their order, and sets the bits corresponding to a zero mask bit to zero. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SSVE_BitPerm is implemented.

SVE2

(FEAT_SVE_BitPerm)

31 29	28 25	24	23 22	21	20 16	15 14	13 12	11 10	9 5	4 0
0 1 0	0 0 1 0	1	size	0	Zm	1 0	1 1	0 1 (opc)	Zn	Zd

Encoding

```
BDEP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_BitPerm) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SSVE_BitPerm) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let data : bits(VL) = Z{}(n);
let mask : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements - 1 do
    result[e*:esize] = BitDeposit{esize}(data[e*:esize], mask[e*:esize]);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.40 BEXT

Gather lower bits from positions selected by bitmask

This instruction gathers bits in each element of the first source vector from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order, and sets the remaining higher-numbered bits to zero. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SSVE_BitPerm is implemented.

SVE2

(FEAT_SVE_BitPerm)

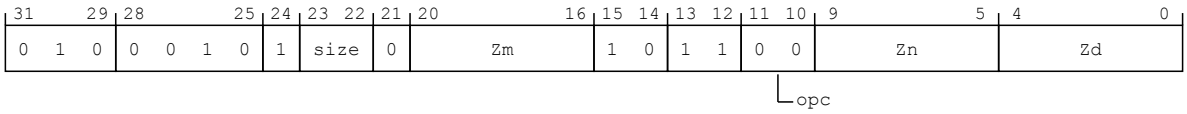

Encoding

```
BEXT  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_BitPerm) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SSVE_BitPerm) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let data : bits(VL) = Z{}(n);
let mask : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements - 1 do
    result[e*:esize] = BitExtract{esize}(data[e*:esize], mask[e*:esize]);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.41 BF1CVT, BF2CVT

8-bit floating-point convert to BFloat16

This instruction converts each even-numbered 8-bit floating-point element of the source vector to BFloat16 while downscaling the value, and places the results in the overlapping 16-bit elements of the destination vector. BF1CVT scales the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$. BF2CVT scales the values by $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$.

The 8-bit floating-point encoding format for BF1CVT is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for BF2CVT is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: BF1CVT and BF2CVT

BF1CVT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	0	0 0 1 1	1 0	Zn	Zd
						L		opc		

Encoding

```
BF1CVT  <Zd>.H, <Zn>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 8;
let d_esize : integer{} = 16;
let issrc2 : boolean = FALSE;
```

BF2CVT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	0	0 0 1 1	1 1	Zn	Zd
						L		opc		

Encoding

```
BF2CVT  <Zd>.H, <Zn>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 8;
let d_esize : integer{} = 16;
let issrc2 : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    let res : bits(d_esize) = FP8ConvertBF(element[s_esize-1:0], issrc2, FPCR(), FPMR());
    result[e*:esize] = ZeroExtend{esize}(res);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.42 BF1CVTLT, BF2CVTLT

8-bit floating-point convert to BFloat16 (top)

This instruction converts each odd-numbered 8-bit floating-point element of the source vector to BFloat16 while downscaling the value, and places the results in the overlapping 16-bit elements of the destination vector. BF1CVTLT scales the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$. BF2CVTLT scales the values by $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$.

The 8-bit floating-point encoding format for BF1CVTLT is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for BF2CVTLT is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: BF1CVTLT and BF2CVTLT

BF1CVTLT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	1	0 0 1 1	1 0	Zn	Zd
						L		opc		

Encoding

```
BF1CVTLT  <Zd>.H, <Zn>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let issrc2 : boolean = FALSE;
```

BF2CVTLT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	1	0 0 1 1	1 1	Zn	Zd
						L		opc		

Encoding

```
BF2CVTLT  <Zd>.H, <Zn>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let issrc2 : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(esize DIV 2) = operand[(2*e + 1)*:(esize DIV 2)];
    result[e*:esize] = FP8ConvertBF(element, issrc2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.43 BFADD (predicated)

BFloat16 add (predicated)

This instruction adds active BFloat16 elements of the first source vector to the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31 29	28 25	24	23 22	21	20	19 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 0	0	0	0 0 0 0	1 0 0	Pg	Zm	Zdn
			size			opc				

Encoding

```
BFADD  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFAdd{16}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.44 BFADD (unpredicated)

BFloat16 add (unpredicated)

This instruction adds all BFloat16 elements of the first source vector to the corresponding elements of the second source vector and places the results in the corresponding elements of the destination vector.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (size)	0	Zm	0 0 0	0 0 0 (opc)	Zn	Zd

Encoding

```
BFADD  <Zd>.H, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[e*:16] = BFAdd{16}(element1, element2, FPCR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.45 BFCLAMP

BFloat16 clamp to minimum/maximum number

This instruction clamps each BFloat16 element in the destination vector to between the BFloat16 minimum value in the corresponding element of the first source vector and the BFloat16 maximum value in the corresponding element of the second source vector and destructively places the clamped results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows for each minimum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31 29	28 25	24	23 22	21	20 16	15 10	9 5	4 0
0 1 1	0 0 1 0	0	0 0 (size)	1	Zm	0 0 1 0 0 1	Zn	Zd

Encoding

```
BFCLAMP  <Zd>.H, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var result : bits(VL);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    let element3 : bits(16) = operand3[e*:16];
    let maxelement : bits(16) = BFMaxNum{}(element1, element3, FPCR());
    result[e*:16] = BFMinNum{16}(maxelement, element2, FPCR());
end;
Z{VL}(d) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.46 BFCVT

Single-precision convert to BFloat16 (predicated)

This instruction converts to BFloat16 from single-precision in each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

Since the result type is smaller than the input type, the results are zero-extended to fill each destination element.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Merging and Zeroing

Merging

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 0 (opc)	0	0 1 0	1 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
BFCVT  <Zd>.H, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 1 1	0 1 0	1	1 0 (opc2)	Pg	Zn	Zd

Encoding

```
BFCVT  <Zd>.H, <Pg>/Z, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 32;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, 32) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 32) then
        let element : bits(32) = operand[e*:32];
        result[(2*e)*:16] = FPConvertBF(element, FPCR());
        result[(2*e+1)*:16] = Zeros{16};

    end;
end;
Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.47 BFCVTN

BFloat16 convert to interleaved 8-bit floating-point

This instruction converts each BFloat16 element of the pair of source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE)}}$, and places the two-way interleaved results in the corresponding 8-bit elements of the destination vector. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–16	15–12	11–10	9–6	5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 1 0	0 0 1 1	1 0	Zn	0	Zd
							opc			

Encoding

```
BFCVTN  <Zd>.B, { <Zn1>.H-<Zn2>.H }
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[(2*e + 0)*:8] = BFConvertFP8(element1, FPCR(), FPMR());
    result[(2*e + 1)*:8] = BFConvertFP8(element2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.48 BFCVTNT

Single-precision convert to BFloat16 (top, predicated)

This instruction converts each active floating-point element of the source vector from single-precision to BFloat16 floating-point, and places the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Merging and Zeroing

Merging

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 0 1 0	1 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
BFCVTNT  <Zd>.H, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 0 0 0	1 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
BFCVTNT  <Zd>.H, <Pg>/Z, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 32;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, 32) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 32) then
        let element : bits(32) = operand[e*:32];
        result[(2*e+1)*:16] = FPConvertBF(element, FPCR());
    elsif !merging then
        result[(2*e+1)*:16] = Zeros{16};

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.49 BFDOT (vectors)

BFloat16 dot product to single-precision

This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

31 29	28 25	24	23	22	21	20 16	15 14	13	12 11	10	9 5	4 0
0 1 1	0 0 1 0	0	0	1	1	Zm	1 0	0	0 0	0	Zn	Zda
				op						o2		

Encoding

```
BFDOT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
    var sum : bits(32) = operand3[e*:32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
    result[e*:32] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.50 BFDOT (indexed)

BFloat16 dot product by indexed element to single-precision

This instruction delimits the source vectors into pairs of BFloat16 elements. The BFloat16 pairs within the second source vector are specified using an immediate index that selects the same BFloat16 pair position within each 128-bit vector segment. The index range is from 0 to 3.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

31 29	28 25	24	23	22	21	20 19	18 16	15 14	13	12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	1	1	i2	Zm	0 1	0	0	0 0	Zn	Zda
				op							opc2		

Encoding

```
BFDOT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * s + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * s + 1)*:16];
    var sum : bits(32) = operand3[e*:32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
    result[e*:32] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.51 BFMAX

BFloat16 maximum (predicated)

This instruction determines the maximum of active BFloat16 elements of the first source vector and the corresponding BFloat16 elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (size)	0	0	0 1 1 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
BFMAX  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFMax{16}(element1, element2, FPCR());
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.52 BFMAXNM

BFloat16 maximum number (predicated)

This instruction determines the maximum number value of active BFloat16 elements of the first source vector and the corresponding BFloat16 elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

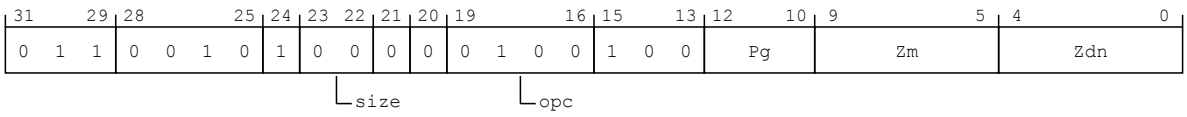

Encoding

```
BFMAXNM  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFMaxNum{16}(element1, element2, FPCR());
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.53 BFMIN

BFloat16 minimum (predicated)

This instruction determines the minimum of active BFloat16 elements of the first source vector and the corresponding BFloat16 elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (size)	0	0	0 1 1 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
BFMIN  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFMin{16}(element1, element2, FPCR());
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.54 BFMINNM

BFloat16 minimum number (predicated)

This instruction determines the minimum number value of active BFloat16 elements of the first source vector and the corresponding BFloat16 elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

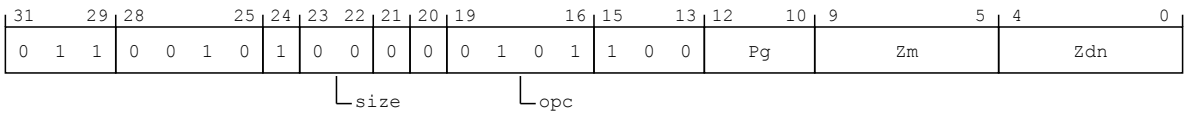

Encoding

```
BFMINNM  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFMinNum{16}(element1, element2, FPCR());
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.55 BFMLA (vectors)

BFloat16 fused multiply-add

This instruction multiplies the active BFloat16 elements of the first source vector by the corresponding BFloat16 elements of the second source vector. The results are then added to elements of the third source (addend) vector without intermediate rounding and destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

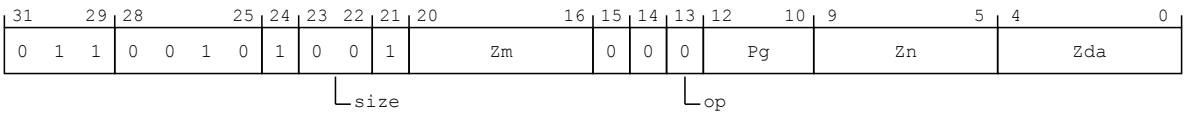

Encoding

```
BFMLA  <Zda>.H, <Pg>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 16) then
        let elem1 : bits(16) = if op1_neg then BFNeg(op1[e*:16]) else op1[e*:16];
        let elem2 : bits(16) = op2[e*:16];
        let elem3 : bits(16) = if op3_neg then BFNeg(op3[e*:16]) else op3[e*:16];

        result[e*:16] = BFMulAdd(elem3, elem1, elem2, FPCR());
    else
        result[e*:16] = op3[e*:16];

    end;
end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.56 BFMLA (indexed)

BFloat16 fused multiply-add by indexed element

This instruction multiplies all BFloat16 elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31 29	28 25	24	23	22	21	20 19	18 16	15 12	11	10	9 5	4 0
0 1 1	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 0 0	1	0	Zn	Zda
									o2	op		

Encoding

```
BFMLA  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let eltspersegment : integer = 128 DIV 16;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elem2 : bits(16) = op2[s*:16];
    let elem1 : bits(16) = if op1_neg then BFNeg(op1[e*:16]) else op1[e*:16];
    let elem3 : bits(16) = if op3_neg then BFNeg(result[e*:16]) else result[e*:16];
    result[e*:16] = BFMulAdd(elem3, elem1, elem2, FPCR());

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.57 BFMLALB (vectors)

BFloat16 multiply-add to single-precision (bottom)

This instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

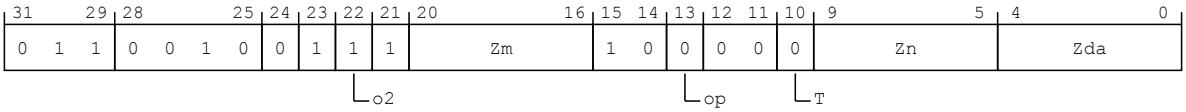

Encoding

```
BFMLALB  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 0)*:16])
                            else op1[(2*e + 0)*:16]);
    let elem2 : bits(16) = op2[(2*e + 0)*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.58 BFMLALB (indexed)

BFloat16 multiply-add by indexed element to single-precision (bottom)

This instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

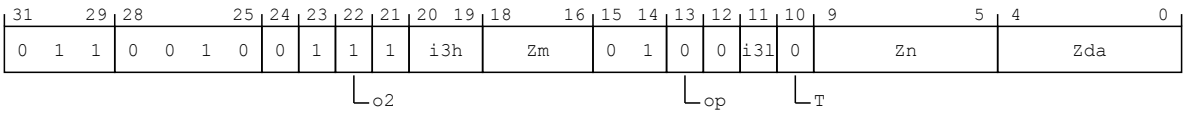

Encoding

```
BFMLALB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 0)*:16])
                            else op1[(2*e + 0)*:16]);
    let elem2 : bits(16) = op2[s*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.59 BFMLALT (vectors)

BFloat16 multiply-add to single-precision (top)

This instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

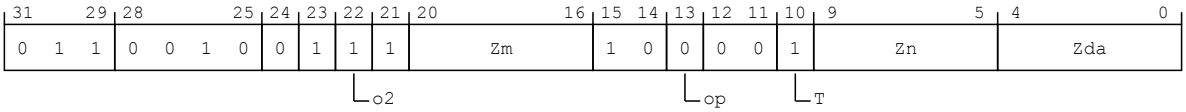

Encoding

```
BFMLALT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 1)*:16])
                            else op1[(2*e + 1)*:16]);
    let elem2 : bits(16) = op2[(2*e + 1)*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.60 BFMLALT (indexed)

BFloat16 multiply-add by indexed element to single-precision (top)

This instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_BF16)

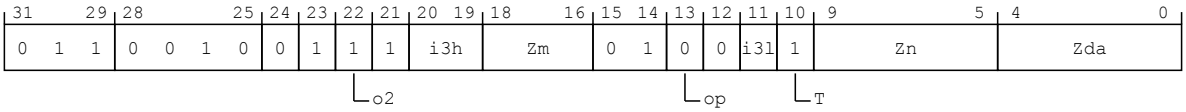

Encoding

```
BFMLALT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_BF16)) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 1)*:16])
                            else op1[(2*e + 1)*:16]);
    let elem2 : bits(16) = op2[s*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.61 BFMLS (vectors)

BFloat16 fused multiply-subtract

This instruction multiplies the active BFloat16 elements of the first source vector by the corresponding BFloat16 elements of the second source vector. The results are then subtracted from elements of the third source (addend) vector without intermediate rounding and destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

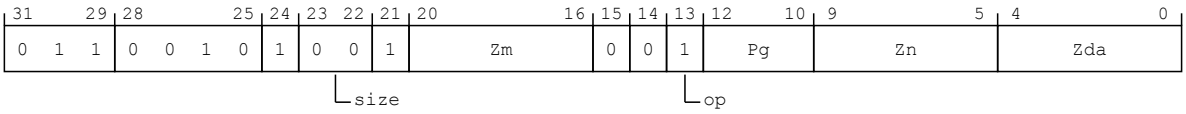

Encoding

```
BFMLS  <Zda>.H, <Pg>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 16) then
        let elem1 : bits(16) = if op1_neg then BFNeg(op1[e*:16]) else op1[e*:16];
        let elem2 : bits(16) = op2[e*:16];
        let elem3 : bits(16) = if op3_neg then BFNeg(op3[e*:16]) else op3[e*:16];

        result[e*:16] = BFMulAdd(elem3, elem1, elem2, FPCR());
    else
        result[e*:16] = op3[e*:16];

    end;
end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.62 BFMLS (indexed)

BFloat16 fused multiply-subtract by indexed element

This instruction multiplies all BFloat16 elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

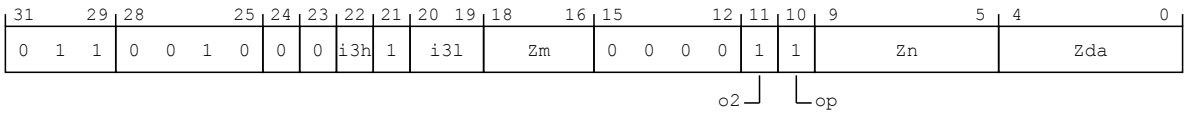

Encoding

```
BFMLS  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let eltspersegment : integer = 128 DIV 16;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elem2 : bits(16) = op2[s*:16];
    let elem1 : bits(16) = if op1_neg then BFNeg(op1[e*:16]) else op1[e*:16];
    let elem3 : bits(16) = if op3_neg then BFNeg(result[e*:16]) else result[e*:16];
    result[e*:16] = BFMulAdd(elem3, elem1, elem2, FPCR());

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.63 BFMLSLB (vectors)

BFloat16 multiply-subtract from single-precision (bottom)

This instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

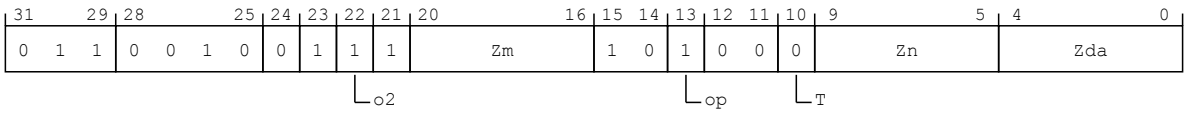

Encoding

```
BFMLSLB  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 0)*:16])
                                else op1[(2*e + 0)*:16]);
    let elem2 : bits(16) = op2[(2*e + 0)*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.64 BFMLSLB (indexed)

BFloat16 multiply-subtract by indexed element from single-precision (bottom)

This instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

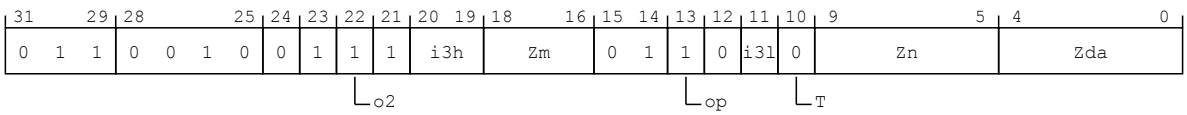

Encoding

```
BFMLSLB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 0)*:16])
                            else op1[(2*e + 0)*:16]);
    let elem2 : bits(16) = op2[s*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.65 BFMLSLT (vectors)

BFloat16 multiply-subtract from single-precision (top)

This instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

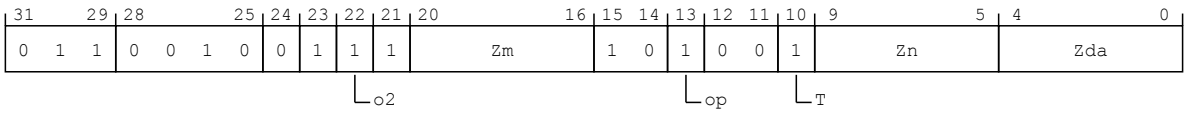

Encoding

```
BFMLSLT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 1)*:16])
                                else op1[(2*e + 1)*:16]);
    let elem2 : bits(16) = op2[(2*e + 1)*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.66 BFMLSLT (indexed)

BFloat16 multiply-subtract by indexed element from single-precision (top)

This instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

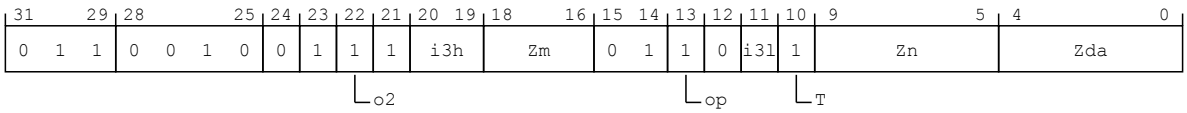

Encoding

```
BFMLSLT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(16) = (if op1_neg then BFNeg(op1[(2*e + 1)*:16])
                            else op1[(2*e + 1)*:16]);
    let elem2 : bits(16) = op2[s*:16];
    let elem3 : bits(32) = op3[e*:32];
    result[e*:32] = BFMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.67 BFMMLA (widening)

BFloat16 matrix multiply-accumulate to single-precision

This instruction multiplies the 2×4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4×2 matrix of BFloat16 values in the corresponding segment of the second source vector.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Calculates the intermediate single-precision results as a sum of two unfused sum-of-products using two pairs of adjacent BFloat16 values from each of the source vectors, and rounds the results before accumulation into the 2×2 matrix of single-precision results in the corresponding segment of the destination vector. This is equivalent to accumulating a 4-way dot product per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Honors FPCR.AH when generating a default NaN result, otherwise disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, this instruction:

- Calculates the intermediate single-precision results as a sum of two fused sum-of-products using two pairs of adjacent BFloat16 values from each of the source vectors, and rounds the results before accumulation into the 2×2 matrix of single-precision results in the destination vector. This is equivalent to accumulating a 4-way dot product per destination element.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

This instruction is unpredicated and vector length agnostic.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE && FEAT_BF16)

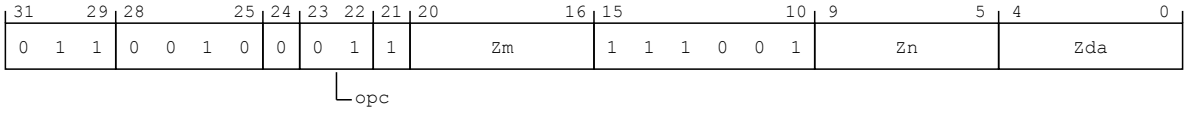

Encoding

```
BFMMLA  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_BF16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer =  VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
var op1, op2 : bits(128);
var res, addend : bits(128);

for s = 0 to segments-1 do
    op1    = operand1[s*:(128)];
    op2    = operand2[s*:(128)];
    addend = operand3[s*:(128)];
    res    = BFMatMulAddH(addend, op1, op2,  FPCR());
    result[s*:(128)] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.68 BFMUL (vectors, predicated)

BFloat16 multiply (predicated)

This instruction multiplies active BFloat16 elements of the first source vector by the corresponding BFloat16 elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

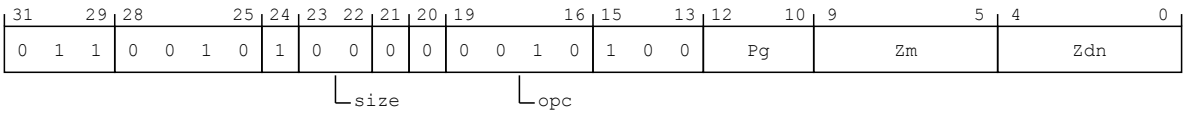

Encoding

```
BFMUL  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFMul(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.69 BFMUL (vectors, unpredicated)

BFloat16 multiply (unpredicated)

This instruction multiplies all BFloat16 elements of the first source vector by the corresponding BFloat16 elements of the second source vector, and places the results in the corresponding elements of the destination vector.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (size)	0	Zm	0 0 0	0 1 0 (opc)	Zn	Zd

Encoding

```
BFMUL  <Zd>.H, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[e*:16] = BFMul(element1, element2, FPCR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.70 BFMUL (indexed)

BFloat16 multiply by indexed element

This instruction multiplies all BFloat16 elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment and places the results in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

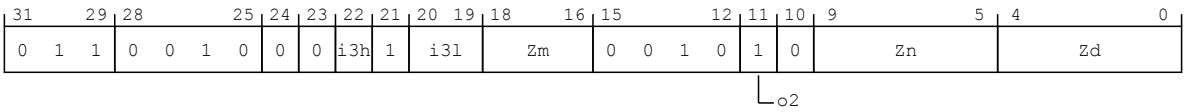

Encoding

```
BFMUL  <Zd>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let eltspersegment : integer = 128 DIV 16;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elem2 : bits(16) = op2[s*:16];
    let elem1 : bits(16) = op1[e*:16];
    result[e*:16] = BFMul(elem1, elem2, FPCR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.71 BFSCALE

BFloat16 adjust exponent (predicated)

This instruction multiplies the active BFloat16 elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified. This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

SVE2

(FEAT_SVE_BFSCALE)

31 29	28 25	24	23 22	21	20	19 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 0	0	0	1 0 0 1	1 0 0	Pg	Zm	Zdn
			size			opc				

Encoding

```
BFSCALE  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_BFSCALE) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:(16)];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : integer = SInt(operand2[e*:(16)]);
        result[e*:(16)] = BFScale(element1, element2, FPCR());
    else
        result[e*:(16)] = element1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.72 BFSUB (predicated)

BFloat16 subtract (predicated)

This instruction subtracts active BFloat16 elements of the second source vector from corresponding BFloat16 elements of the first source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (size)	0	0	0 0 0 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
BFSUB  <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 16;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 16) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    if ActivePredicateElement{PL}(mask, e, 16) then
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFSub(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:16] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.73 BFSUB (unpredicated)

BFloat16 subtract (unpredicated)

This instruction subtracts all BFloat16 elements of the second source vector from corresponding BFloat16 elements of the first source vector and places the results in the corresponding elements of the destination vector.

This instruction follows SVE2 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

SVE2

(FEAT_SVE_B16B16)

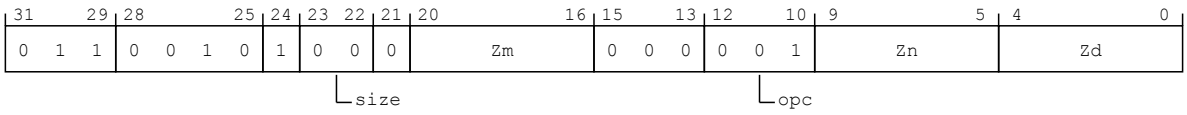

Encoding

```
BFSUB  <Zd>.H, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_B16B16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[e*:16] = BFSub(element1, element2, FPCR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.74 BGRP

Group bits to right or left as selected by bitmask

This instruction separates bits in each element of the first source vector by gathering from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, and from positions indicated by zero bits to the highest-numbered bits of the destination element, preserving the bit order within each group. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SSVE_BitPerm is implemented.

SVE2

(FEAT_SVE_BitPerm)

31 29	28 25	24	23 22	21	20 16	15 14	13 12	11 10	9 5	4 0
0 1 0	0 0 1 0	1	size	0	Zm	1 0	1 1	1 0 (opc)	Zn	Zd

Encoding

```
BGRP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_BitPerm) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SSVE_BitPerm) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let data : bits(VL) = Z{}(n);
let mask : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements - 1 do
    result[e*:esize] = BitGroup{esize}(data[e*:esize], mask[e*:esize]);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.75 BIC (immediate)

Bitwise clear bits using immediate (unpredicated)

This instruction performs a bitwise clear bits using an immediate with each 64-bit element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of AND (immediate). This means:

- The encodings in this description are named to match the encodings of AND (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of AND (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–20	19–18	17–5	4–0
0 0 0	0 0 1 0	1	1 0 (opc)	0 0	0 0	imm13	Zdn

Encoding

```
BIC  <Zdn>.<T>, <Zdn>.<T>, #<const>
```

is equivalent to

```
AND  <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of AND (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of AND (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.76 BIC (predicates)

Bitwise clear predicates

This instruction performs a bitwise AND between the active elements of the first source predicate and the inverted corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

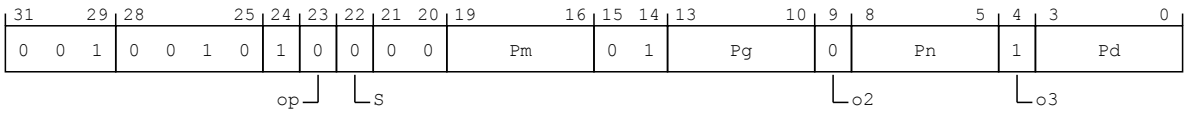

Encoding

```
BIC  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 AND (NOT element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.77 BIC (vectors, predicated)

Bitwise clear (predicated)

This instruction performs a bitwise AND between the active elements of the first source vector and the inverted corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 1	0 1 1 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
BIC  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element1 AND (NOT element2);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.78 BIC (vectors, unpredicated)

Bitwise clear (unpredicated)

This instruction performs a bitwise AND between the elements of the first source vector and the inverted corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	1 1	1	Zm	0 0 1	1 0 0	Zn	Zd
			opc						

Encoding

```
BIC  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);

Z{VL}(d) = operand1 AND (NOT operand2);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.79 BICS

Bitwise clear predicates, setting the condition flags

This instruction performs a bitwise AND between the active elements of the first source predicate and the inverted corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

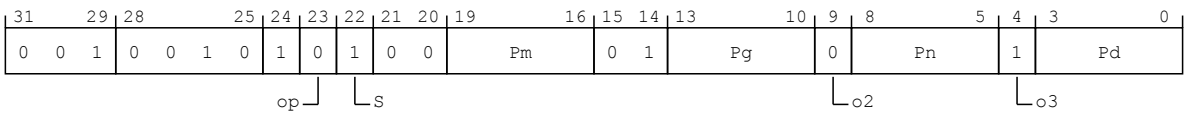

Encoding

```
BICS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 AND (NOT element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.80 BRKA

Break after first true condition

This instruction sets the destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false.

Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 10	9	8 5	4	3 0
0 0 1	0 0 1 0	1	0	0	0 1	0 0 0 0	0 1	Pg	0	Pn	M	Pd

B S

Encoding

```
BRKA  <Pd>.B, <Pg>/<ZM>, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
let merging : boolean = (M == '1');
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<ZM>

Is the predication qualifier, encoded in “M”:

M	<ZM>
0	Z
1	M

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
let operand2 : bits(PL) = if merging then P{PL}(d) else Zeros{PL};
var break : boolean = FALSE;
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element : boolean = ActivePredicateElement{PL}(operand, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let pbit : bit = if !break then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
        break = break || element;
    elsif merging then
        let pbit : bit = PredicateElement{PL}(operand2, e, esize);
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.81 BRKAS

Break after first true condition, setting the condition flags

This instruction sets the destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

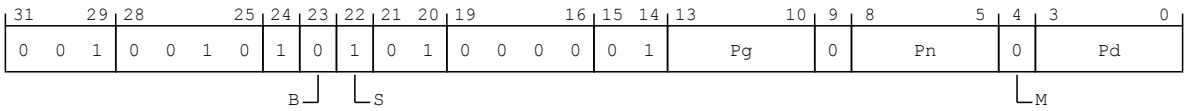

Encoding

```
BRKAS  <Pd>.B, <Pg>/Z, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
let merging : boolean = FALSE;
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
let operand2 : bits(PL) = if merging then P{PL}(d) else Zeros{PL};
var break : boolean = FALSE;
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let element : boolean = ActivePredicateElement{PL}(operand, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let pbit : bit = if !break then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
        break = break || element;
    elsif merging then
        let pbit : bit = PredicateElement{PL}(operand2, e, esize);
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.82 BRKB

Break before first true condition

This instruction sets the destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false.

Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21–20	19–16	15–14	13–10	9	8–5	4	3–0
0 0 1	0 0 1 0	1	1 (B)	0 (S)	0 1	0 0 0 0	0 1	Pg	0	Pn	M	Pd

Encoding

```
BRKB  <Pd>.B, <Pg>/<ZM>, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
let merging : boolean = (M == '1');
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<ZM>

Is the predication qualifier, encoded in “M”:

M	<ZM>
0	Z
1	M

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
let operand2 : bits(PL) = if merging then P{PL}(d) else Zeros{PL};
var break : boolean = FALSE;
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element : boolean = ActivePredicateElement{PL}(operand, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        break = break || element;
        let pbit : bit = if !break then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    elsif merging then
        let pbit : bit = PredicateElement{PL}(operand2, e, esize);
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.83 BRKBS

Break before first true condition, setting the condition flags

This instruction sets the destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

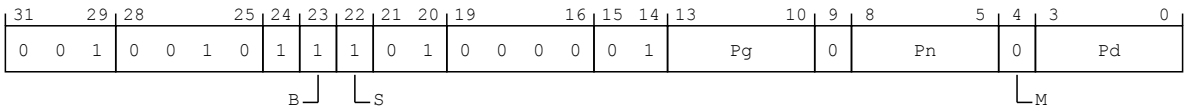

Encoding

```
BRKBS  <Pd>.B, <Pg>/Z, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
let merging : boolean = FALSE;
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
let operand2 : bits(PL) = if merging then P{PL}(d) else Zeros{PL};
var break : boolean = FALSE;
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let element : boolean = ActivePredicateElement{PL}(operand, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        break = break || element;
        let pbit : bit = if !break then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    elsif merging then
        let pbit : bit = PredicateElement{PL}(operand2, e, esize);
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.84 BRKN

Propagate break to next partition

This instruction propagates break to next partition. If the Last active element of the first source predicate is false, then the destination predicate is set to all-false. Otherwise, the destination and second source predicates are left unchanged. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23	22	21	20	19	16	15	14	13	10	9	8	5	4	3	0
0 0 1		0 0 1 0		1	0	0	0	1	1 0 0 0		0	1	Pg		0	Pn		0	Pdm	

S

Encoding

```
BRKN  <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let dm : integer = UInt(Pdm);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pdm>

Is the name of the second source and destination scalable predicate register, encoded in the “Pdm” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(dm);
var result : bits(PL);

if LastActive{PL}(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros{PL};
end;

if setflags then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.[N,Z,C,V] = PredTest{PL}(Ones{PL}, result, 8);
end;
P{PL}(dm) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.85 BRKNS

Propagate break to next partition, setting the condition flags

This instruction propagates break to next partition and sets the condition flags. If the Last active element of the first source predicate is false, then the destination predicate is set to all-false. Otherwise, the destination and second source predicates are left unchanged. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21–20	19–16	15–14	13–10	9	8–5	4	3–0
0 0 1	0 0 1 0	1	0	1	0 1	1 0 0 0	0 1	Pg	0	Pn	0	Pdm
				S								

Encoding

```
BRKNS  <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let dm : integer = UInt(Pdm);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pdm>

Is the name of the second source and destination scalable predicate register, encoded in the “Pdm” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(dm);
var result : bits(PL);

if LastActive{PL}(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros{PL};
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(Ones{PL}, result, 8);
end;
P{PL}(dm) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.86 BRKPA

Break after first true condition, propagating from previous partition

This instruction breaks after first true condition, propagating from previous partition. If the Last active element of the first source predicate is false, then the destination predicate is set to all-false. Otherwise, the destination predicate elements up to and including the first active and true source element are set to true, then the subsequent elements are set to false. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

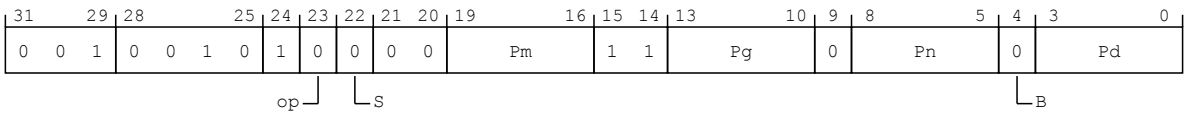

Encoding

```
BRKPA  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
var last : boolean = (LastActive{PL}(mask, operand1, 8) == '1');

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 8) then
        let pbit : bit = if last then '1' else '0';
        result[e*:1] = ZeroExtend{1}(pbit);
        last = last && (!ActivePredicateElement{PL}(operand2, e, 8));
    else
        result[e*:1] = ZeroExtend{1}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.87 BRKPAS

Break after first true condition, propagating from previous partition and setting the condition flags

This instruction breaks after first true condition, propagating from previous partition, and sets the condition flags. If the Last active element of the first source predicate is false, then the destination predicate is set to all-false. Otherwise, the destination predicate elements up to and including the first active and true source element are set to true, then the subsequent elements are set to false. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

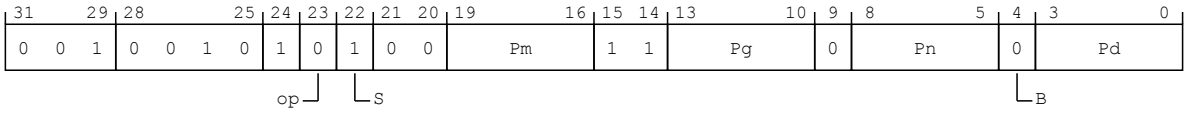

Encoding

```
BRKPAS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
var last : boolean = (LastActive{PL}(mask, operand1, 8) == '1');

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 8) then
        let pbit : bit = if last then '1' else '0';
        result[e*:1] = ZeroExtend{1}(pbit);
        last = last && (!ActivePredicateElement{PL}(operand2, e, 8));
    else
        result[e*:1] = ZeroExtend{1}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.88 BRKPB

Break before first true condition, propagating from previous partition

This instruction breaks before first true condition, propagating from previous partition. If the Last active element of the first source predicate is false, then the destination predicate is set to all-false. Otherwise, the destination predicate elements up to but not including the first active and true source element are set to true, then the subsequent elements are set to false. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

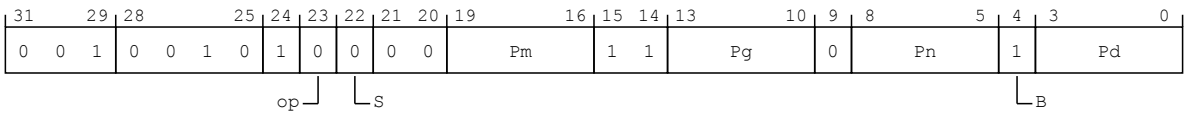

Encoding

```
BRKPB  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
var last : boolean = (LastActive{PL}(mask, operand1, 8) == '1');

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 8) then
        last = last && (!ActivePredicateElement{PL}(operand2, e, 8));
        let pbit : bit = if last then '1' else '0';
        result[e*:1] = ZeroExtend{1}(pbit);
    else
        result[e*:1] = ZeroExtend{1}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.89 BRKPBS

Break before first true condition, propagating from previous partition and setting the condition flags

This instruction breaks before first true condition, propagating from previous partition, and sets the condition flags. If the Last active element of the first source predicate is false, then the destination predicate is set to all-false. Otherwise, the destination predicate elements up to but not including the first active and true source element are set to true, then the subsequent elements are set to false. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

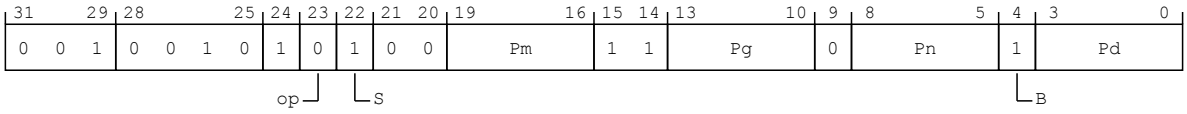

Encoding

```
BRKPBS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
var last : boolean = (LastActive{PL}(mask, operand1, 8) == '1');

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 8) then
        last = last && (!ActivePredicateElement{PL}(operand2, e, 8));
        let pbit : bit = if last then '1' else '0';
        result[e*:1] = ZeroExtend{1}(pbit);
    else
        result[e*:1] = ZeroExtend{1}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.90 BSL1N

Bitwise select with first input inverted

This instruction selects bits from the inverted first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is destructively placed in the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 1 (opc)	1	Zm	0 0 1	1 1	1 (o2)	Zk	Zdn

Encoding

```
BSL1N  <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let m : integer = UInt(Zm);
let k : integer = UInt(Zk);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zk>

Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(k);

Z{VL}(dn) = (NOT(operand1) AND operand3) OR (operand2 AND NOT(operand3));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.91 BSL2N

Bitwise select with second input inverted

This instruction selects bits from the first source vector where the corresponding bit in the third source vector is ‘1’, and from the inverted second source vector where the corresponding bit in the third source vector is ‘0’. The result is destructively placed in the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 0 (opc)	1	Zm	0 0 1	1 1	1 (o2)	Zk	Zdn

Encoding

```
BSL2N  <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let m : integer = UInt(Zm);
let k : integer = UInt(Zk);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the third source scalable vector register, encoded in the “Zk” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(k);

Z{VL}(dn) = (operand1 AND operand3) OR (NOT(operand2) AND NOT(operand3));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.92 BSL

Bitwise select

This instruction selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is destructively placed in the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 0 (opc)	1	Zm	0 0 1	1 1	1 (o2)	Zk	Zdn

Encoding

```
BSL  <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let m : integer = UInt(Zm);
let k : integer = UInt(Zk);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zk>

Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(k);

Z{VL}(dn) = (operand1 AND operand3) OR (operand2 AND NOT(operand3));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.93 CADD

Complex integer add

This instruction adds the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector that have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. The results are destructively placed in the corresponding elements of the first source vector. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–17	16	15–14	13–11	10	9–5	4–0
0 1 0	0 0 1 0	1	size	0	0 0 0 0	0	1 1	0 1 1	rot	Zm	Zdn
						op					

Encoding

```
CADD  <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
let sub_i : boolean = (rot == '0');
let sub_r : boolean = (rot == '1');
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
0	#90
1	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for p = 0 to pairs-1 do
    var acc_r : integer  = SInt(operand1[(2 * p + 0)*:esize]);
    var acc_i : integer  = SInt(operand1[(2 * p + 1)*:esize]);
    let elt2_r : integer = SInt(operand2[(2 * p + 0)*:esize]);
    let elt2_i : integer = SInt(operand2[(2 * p + 1)*:esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    end;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;
    end;

    result[(2 * p + 0)*:esize] = acc_r[esize-1:0];
    result[(2 * p + 1)*:esize] = acc_i[esize-1:0];
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.94 CDOT (vectors)

Complex integer dot product

This instruction delimits the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element that overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14 12	11 10	9 5	4 0
0 1 0	0 0 1 0	0	size	0	Zm	0	0 0 1	rot	Zn	Zda

Encoding

```
CDOT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_i : boolean = (rot[0] == rot[1]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size<0>”:

size<0>	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 1 do
        let elt1_r : integer = SInt(operand1[(4 * e + 2 * i + 0)*:(esize DIV 4)]);
        let elt1_i : integer = SInt(operand1[(4 * e + 2 * i + 1)*:(esize DIV 4)]);
        let elt2_a : integer = SInt(operand2[(4 * e + 2 * i + sel_a)*:(esize DIV 4)]);
        let elt2_b : integer = SInt(operand2[(4 * e + 2 * i + sel_b)*:(esize DIV 4)]);
        if sub_i then
            res = (res + (elt1_r * elt2_a)) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
        end;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.95 CDOT (indexed)

Complex integer dot product by indexed element

This instruction delimits the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element that overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of these instructions select a single pair of complex numbers within each 128-bit segment of the second source vector as the multiplier for all pairs of complex numbers within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index that selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 8-bit to 32-bit and 16-bit to 64-bit

8-bit to 32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–12	11–10	9–5	4–0
0 1 0	0 0 1 0	0	1 0 (size)	1	i2	Zm	0 1 0 0	rot	Zn	Zda

Encoding

```
CDOT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let sel_b : integer = UInt(NOT(rot[0]));
let sub_i : boolean = (rot[0] == rot[1]);
```

16-bit to 64-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 12	11 10	9 5	4 0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	0 1 0 0	rot	Zn	Zda
			size							

Encoding

```
CDOT  <Zda>.D, <Zn>.H, <Zm>.H[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_i : boolean = (rot[0] == rot[1]);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “8-bit to 32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “16-bit to 64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “8-bit to 32-bit” variant: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “16-bit to 64-bit” variant: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

<const>

Is the const specifier, encoded in “rot”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 1 do
        let elt1_r : integer = SInt(operand1[(4 * e + 2 * i + 0)*:(esize DIV 4)]);
        let elt1_i : integer = SInt(operand1[(4 * e + 2 * i + 1)*:(esize DIV 4)]);
        let elt2_a : integer = SInt(operand2[(4 * s + 2 * i + sel_a)*:(esize DIV 4)]);
        let elt2_b : integer = SInt(operand2[(4 * s + 2 * i + sel_b)*:(esize DIV 4)]);
        if sub_i then
            res = (res + (elt1_r * elt2_a)) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
        end;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.96 CLASTA (scalar)

Conditionally extract element after last to general-purpose register

This instruction extracts and zero-extends the element after the Last active element, or if the Last active element is the final element it extracts element zero, from the source vector register, and destructively places the zero-extended value in the destination and first source general-purpose register. If there are no Active elements, the least significant element-size bits of the destination and first source general-purpose register are destructively zero-extended.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	1	0 0 0	0	1 0	1	Pg	Zm	Rdn
							B					

Encoding

```
CLASTA  <R><dn>, <Pg>, <R><dn>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Rdn);
let m : integer = UInt(Zm);
let csize : integer{} = if esize < 64 then 32 else 64;
let isBefore : boolean = FALSE;
```

Assembler Symbols

<R>

Is a width specifier, encoded in “size”:

size	<R>
00	W
01	W
10	W
11	X

<dn>

Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the “Rdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(esize) = X{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(csize);
var last : integer = LastActiveElement{PL}(mask, esize);

if last < 0 then
    result = ZeroExtend{csize}(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0; end;
    end;
    result = ZeroExtend{csize}(operand2[last*:esize]);

end;
X{csize}(dn) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.97 CLASTA (SIMD&FP scalar)

Conditionally extract element after last to SIMD&FP scalar register

This instruction extracts and zero-extends the element after the Last active element, or if the Last active element is the final element it extracts element zero, from the source vector register, and destructively places the zero-extended value in the destination and first source SIMD & floating-point scalar register. If there are no Active elements, the least significant element-size bits of the destination and first source SIMD & floating-point scalar register are destructively zero-extended.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	0	1 0 1	0	1 0	0	Pg	Zm	Vdn

B

Encoding

```
CLASTA  <V><dn>, <Pg>, <V><dn>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Vdn);
let m : integer = UInt(Zm);
let isBefore : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<dn>

Is the number [0-31] of the source and destination SIMD&FP register, encoded in the “Vdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(esize) = V{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(esize);
var last : integer = LastActiveElement{PL}(mask, esize);

if last < 0 then
    result = ZeroExtend{esize}(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0; end;
    end;
    result = operand2[last*:esize];

end;
V{esize}(dn) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.98 CLASTA (vectors)

Conditionally extract element after last to vector register

This instruction extracts the element after the Last active element, or if the Last active element is the final element it extracts element zero, from the second source vector register, and then replicates the extracted element to destructively fill the destination and first source vector. If there are no Active elements, the destination and source vector are unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	0	1 0 0	0	1 0	0	Pg	Zm	Zdn
							B					

Encoding

```
CLASTA  <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let isBefore : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);
var last : integer = LastActiveElement{PL}(mask, esize);

if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0; end;
    end;
    for e = 0 to elements-1 do
        result[e*:esize] = operand2[last*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.99 CLASTB (scalar)

Conditionally extract last element to general-purpose register

This instruction extracts and zero-extends the Last active element from the source vector register, and destructively places the zero-extended value in the destination and first source general-purpose register. If there are no Active elements, the least significant element-size bits of the destination and first source general-purpose register are destructively zero-extended.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	1	0 0 0	1	1 0	1	Pg	Zm	Rdn
							B					

Encoding

```
CLASTB  <R><dn>, <Pg>, <R><dn>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Rdn);
let m : integer = UInt(Zm);
let csize : integer{} = if esize < 64 then 32 else 64;
let isBefore : boolean = TRUE;
```

Assembler Symbols

<R>

Is a width specifier, encoded in “size”:

size	<R>
00	W
01	W
10	W
11	X

<dn>

Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the “Rdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(esize) = X{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(csize);
var last : integer = LastActiveElement{PL}(mask, esize);

if last < 0 then
    result = ZeroExtend{csize}(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0; end;
    end;
    result = ZeroExtend{csize}(operand2[last*:esize]);

end;
X{csize}(dn) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.100 CLASTB (SIMD&FP scalar)

Conditionally extract last element to SIMD&FP scalar register

This instruction extracts and zero-extends the Last active element from the source vector register, and destructively places the zero-extended value in the destination and first source SIMD & floating-point scalar register. If there are no Active elements, the least significant element-size bits of the destination and first source SIMD & floating-point scalar register are destructively zero-extended.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	0	1 0 1	1	1 0	0	Pg	Zm	Vdn
							B					

Encoding

```
CLASTB  <V><dn>, <Pg>, <V><dn>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Vdn);
let m : integer = UInt(Zm);
let isBefore : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<dn>

Is the number [0-31] of the source and destination SIMD&FP register, encoded in the “Vdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(esize) = V{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(esize);
var last : integer = LastActiveElement{PL}(mask, esize);

if last < 0 then
    result = ZeroExtend{esize}(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0; end;
    end;
    result = operand2[last*:esize];

end;
V{esize}(dn) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.101 CLASTB (vectors)

Conditionally extract last element to vector register

This instruction extracts the Last active element from the second source vector register, and then replicates the extracted element to destructively fill the destination and first source vector. If there are no Active elements, the destination and source vector are unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	1 0 0	1	1 0	0	Pg	Zm	Zdn
							B					

Encoding

```
CLASTB  <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let isBefore : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);
var last : integer = LastActiveElement{PL}(mask, esize);

if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0; end;
    end;
    for e = 0 to elements-1 do
        result[e*:esize] = operand2[last*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.102 CLS

Count leading sign bits (predicated)

This instruction determines the number of consecutive sign bits, starting from the most significant bit in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	0 0 0	1 0 1	Pg	Zn	Zd

M: 20 — opc: 18–16

Encoding

```
CLS  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	0 0 0	1 0 1	Pg	Zn	Zd

M: 20 — opc: 18–16

Encoding

```
CLS  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = CountLeadingSignBits(element)[esize-1:0];

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.103 CLZ

Count leading zero bits (predicated)

This instruction determines the number of consecutive binary zero bits, starting from the most significant bit in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	0 0 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
CLZ  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	0 0 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
CLZ  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = CountLeadingZeroBits(element)[esize-1:0];

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.104 CMLA (vectors)

Complex integer multiply-add

This instruction multiplies the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

This instruction then adds the products to the corresponding components of the complex numbers in the addend vector, and destructively places the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

SVE2

(FEAT_SVE2 || FEAT_SME)

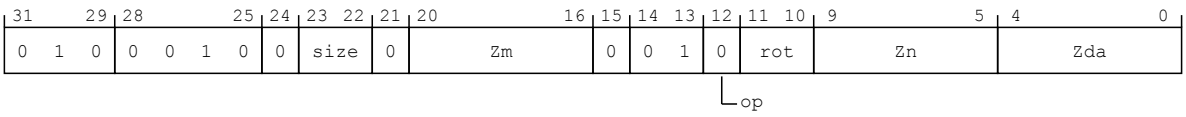

Encoding

```
CMLA  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_r : boolean = (rot[0] != rot[1]);
let sub_i : boolean = (rot[1] == '1');
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for p = 0 to pairs-1 do
    let elt1_a : integer = SInt(operand1[(2 * p + sel_a)*:esize]);
    let elt2_a : integer = SInt(operand2[(2 * p + sel_a)*:esize]);
    let elt2_b : integer = SInt(operand2[(2 * p + sel_b)*:esize]);
    let elt3_r : bits(esize) = operand3[(2 * p + 0)*:esize];
    let elt3_i : bits(esize) = operand3[(2 * p + 1)*:esize];
    let product_r : integer =  elt1_a * elt2_a;
    let product_i : integer =  elt1_a * elt2_b;
    if sub_r then
        result[(2 * p + 0)*:esize] = elt3_r - product_r;
    else
        result[(2 * p + 0)*:esize] = elt3_r + product_r;
    end;
    if sub_i then
        result[(2 * p + 1)*:esize] = elt3_i - product_i;
    else
        result[(2 * p + 1)*:esize] = elt3_i + product_i;
    end;

end;
Z{VL}(da) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.105 CMLA (indexed)

Complex integer multiply-add by indexed element

This instruction multiplies the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

This instruction then adds the products to the corresponding components of the complex numbers in the addend vector, and destructively places the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 16-bit and 32-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–12	11–10	9–5	4–0
0 1 0	0 0 1 0	0	1 0 (size)	1	i2	Zm	0 1 1 0	rot	Zn	Zda

Encoding

```
CMLA  <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_r : boolean = (rot[0] != rot[1]);
let sub_i : boolean = (rot[1] == '1');
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–12	11–10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (size)	1	i1	Zm	0 1 1 0	rot	Zn	Zda

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CMLA  <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_r : boolean = (rot[0] != rot[1]);
let sub_i : boolean = (rot[1] == '1');
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “32-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let pairspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for p = 0 to pairs-1 do
    let segmentbase : integer = p - (p MOD pairspersegment);
    let s : integer = segmentbase + index;
    let elt1_a : integer = SInt(operand1[(2 * p + sel_a)*:esize]);
    let elt2_a : integer = SInt(operand2[(2 * s + sel_a)*:esize]);
    let elt2_b : integer = SInt(operand2[(2 * s + sel_b)*:esize]);
    let elt3_r : bits(esize) = operand3[(2 * p + 0)*:esize];
    let elt3_i : bits(esize) = operand3[(2 * p + 1)*:esize];
    let product_r : integer =  elt1_a * elt2_a;
    let product_i : integer =  elt1_a * elt2_b;
    if sub_r then
        result[(2 * p + 0)*:esize] = elt3_r - product_r;
    else
        result[(2 * p + 0)*:esize] = elt3_r + product_r;
    end;
    if sub_i then
        result[(2 * p + 1)*:esize] = elt3_i - product_i;
    else
        result[(2 * p + 1)*:esize] = elt3_i + product_i;
    end;

end;
Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.106 CMP<cc> (immediate)

Compare vector to immediate

This instruction compares active integer elements in the source vector with an immediate, and places the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
LE	signed less than or equal
LO	unsigned lower than
LS	unsigned lower than or same
LT	signed less than
NE	not equal

It has encodings from 10 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same, Less than, Less than or equal, Lower, Lower or same, and Not equal

Equal

(FEAT_SVE || FEAT_SME)

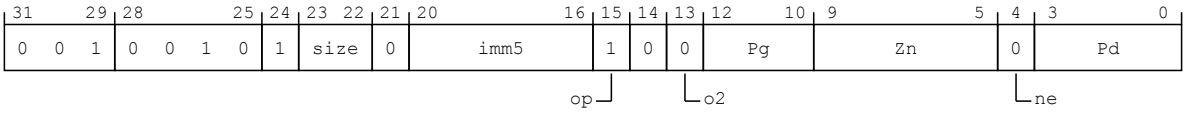

Encoding

```
CMPEQ  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_EQ;
let imm : integer = SInt(imm5);
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Greater than

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		1	size	0	imm5		0	0	0	Pg		Zn		1	Pd	
									op		lt					ne		

Encoding

```
CMPGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
let imm : integer = SInt(imm5);
let unsigned : boolean = FALSE;
```

Greater than or equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		1	size	0	imm5		0	0	0	Pg		Zn		0	Pd	
									op		lt					ne		

Encoding

```
CMPGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
let imm : integer = SInt(imm5);
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Higher

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 14	13	12 10	9 5	4	3 0
0 0 1	0 0 1 0	0	size	1	imm7	0	Pg	Zn	1	Pd
						lt			ne	

Encoding

```
CMPHI  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
let imm : integer = UInt(imm7);
let unsigned : boolean = TRUE;
```

Higher or same

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 14	13	12 10	9 5	4	3 0
0 0 1	0 0 1 0	0	size	1	imm7	0	Pg	Zn	0	Pd
						lt			ne	

Encoding

```
CMPHS  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
let imm : integer = UInt(imm7);
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Less than

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0	1	size	0	imm5	0	0	1	Pg	Zn	0	Pd
						op		lt			ne	

Encoding

```
CMPLT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LT;
let imm : integer = SInt(imm5);
let unsigned : boolean = FALSE;
```

Less than or equal

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0	1	size	0	imm5	0	0	1	Pg	Zn	1	Pd
						op		lt			ne	

Encoding

```
CMPLE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LE;
let imm : integer = SInt(imm5);
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Lower

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0	0	size	1	imm7	1 (lt)	Pg	Zn	0 (ne)	Pd

Encoding

```
CMPLO  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LT;
let imm : integer = UInt(imm7);
let unsigned : boolean = TRUE;
```

Lower or same

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0	0	size	1	imm7	1 (lt)	Pg	Zn	1 (ne)	Pd

Encoding

```
CMPLS  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LE;
let imm : integer = UInt(imm7);
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Not equal

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 0 1	0 0 1 0	1	size	0	imm5	1	0	0	Pg	Zn	1	Pd
						op		o2			ne	

Encoding

```
CMPNE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_NE;
let imm : integer = SInt(imm5);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<imm>

For the “Equal”, “Greater than”, “Greater than or equal”, “Less than”, “Less than or equal”, and “Not equal” variants: is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

For the “Higher”, “Higher or same”, “Lower”, and “Lower or same” variants: is the unsigned immediate operand, in the range 0 to 127, encoded in the “imm7” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    if ActivePredicateElement{PL}(mask, e, esize) then
        var cond : boolean;
        case cmp_op of
            when Cmp_EQ => cond = element1 == imm;
            when Cmp_NE => cond = element1 != imm;
            when Cmp_GE => cond = element1 >= imm;
            when Cmp_LT => cond = element1 <  imm;
            when Cmp_GT => cond = element1 >  imm;
            when Cmp_LE => cond = element1 <= imm;
        end;
        let pbit : bit = if cond then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.107 CMP<cc> (wide elements)

Compare vector to 64-bit wide elements

This instruction compares active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector, and places the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
LE	signed less than or equal
LO	unsigned lower than
LS	unsigned lower than or same
LT	signed less than
NE	not equal

It has encodings from 10 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same, Less than, Less than or equal, Lower, Lower or same, and Not equal

Equal

(FEAT_SVE || FEAT_SME)

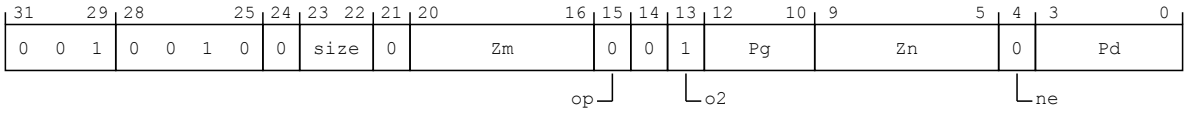

Encoding

```
CMPEQ  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_EQ;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Greater than

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		0	1	0	Pg		Zn		1	Pd	
									U		lt					ne		

Encoding

```
CMPGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
let unsigned : boolean = FALSE;
```

Greater than or equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		0	1	0	Pg		Zn		0	Pd	
									U		lt					ne		

Encoding

```
CMPGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Higher

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		1	1	0	Pg		Zn		1	Pd	
									U		lt					ne		

Encoding

```
CMPHI  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
let unsigned : boolean = TRUE;
```

Higher or same

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		1	1	0	Pg		Zn		0	Pd	
									U		lt					ne		

Encoding

```
CMPHS  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Less than

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		0	1	1	Pg		Zn		0	Pd	
									U		lt					ne		

Encoding

```
CMPLT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LT;
let unsigned : boolean = FALSE;
```

Less than or equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		0	1	1	Pg		Zn		1	Pd	
									U		lt					ne		

Encoding

```
CMPLE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LE;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Lower

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		1	1	1	Pg		Zn		0	Pd	
									U		lt					ne		

Encoding

```
CMPLO  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LT;
let unsigned : boolean = TRUE;
```

Lower or same

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		1	1	1	Pg		Zn		1	Pd	
									U		lt					ne		

Encoding

```
CMPLS  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_LE;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Not equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size		0	Zm		0	0	1	Pg		Zn		1	Pd	
										op		o2					ne		

Encoding

```
CMPNE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_NE;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    if ActivePredicateElement{PL}(mask, e, esize) then
        var cond : boolean;
        let op2elt : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        case cmp_op of
            when Cmp_EQ => cond = element1 == element2;
            when Cmp_NE => cond = element1 != element2;
            when Cmp_GE => cond = element1 >= element2;
            when Cmp_LT => cond = element1 <  element2;
            when Cmp_GT => cond = element1 >  element2;
            when Cmp_LE => cond = element1 <= element2;
        end;
        let pbit : bit = if cond then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.108 CMP<cc> (vectors)

Compare vectors

This instruction compares active integer elements in the first source vector with the corresponding elements in the second source vector, and places the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
NE	not equal

This instruction is used by the pseudo-instructions CMPLE (vectors), CMPLO (vectors), CMPLS (vectors), and CMPLT (vectors).

It has encodings from 6 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same, and Not equal

Equal

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4	3–0
0 0 1	0 0 1 0	0	size	0	Zm	1	0	1	Pg	Zn	0	Pd
						op		o2			ne	

Encoding

```
CMPEQ  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_EQ;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Greater than

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 0 1	0 0 1 0	0	size	0	Zm	1	0	0	Pg	Zn	1	Pd
						op		o2			ne	

Encoding

```
CMPGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
let unsigned : boolean = FALSE;
```

Greater than or equal

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 0 1	0 0 1 0	0	size	0	Zm	1	0	0	Pg	Zn	0	Pd
						op		o2			ne	

Encoding

```
CMPGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Higher

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		0	0	0	Pg		Zn		1	Pd	
									op		o2					ne		

Encoding

```
CMPHI  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
let unsigned : boolean = TRUE;
```

Higher or same

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		0	0	0	Pg		Zn		0	Pd	
									op		o2					ne		

Encoding

```
CMPHS  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Not equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 0 1		0 0 1 0		0	size	0	Zm		1	0	1	Pg		Zn		1	Pd	
									op		o2					ne		

Encoding

```
CMPNE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_NE;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    if ActivePredicateElement{PL}(mask, e, esize) then
        var cond : boolean;
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        case cmp_op of
            when Cmp_EQ => cond = element1 == element2;
            when Cmp_NE => cond = element1 != element2;
            when Cmp_GE => cond = element1 >= element2;
            when Cmp_LT => cond = element1 <  element2;
            when Cmp_GT => cond = element1 >  element2;
            when Cmp_LE => cond = element1 <= element2;
        end;
        let pbit : bit = if cond then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.109 CMPLE (vectors)

Compare signed less than or equal to vector, setting the condition flags

This instruction compares active signed integer elements in the first source vector being less than or equal to the corresponding signed elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Greater than or equal

(FEAT_SVE || FEAT_SME)

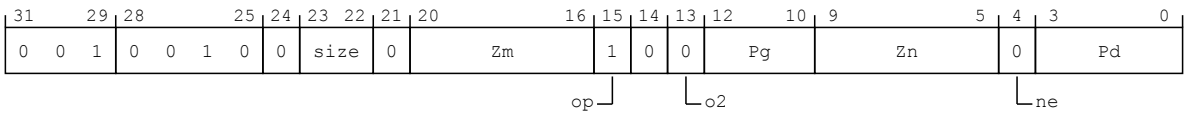

Encoding

```
CMPLE  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
CMPGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of CMP<cc> (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.110 CMPLO (vectors)

Compare unsigned lower than vector, setting the condition flags

This instruction compares active unsigned integer elements in the first source vector being lower than the corresponding unsigned elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Higher

(FEAT_SVE || FEAT_SME)

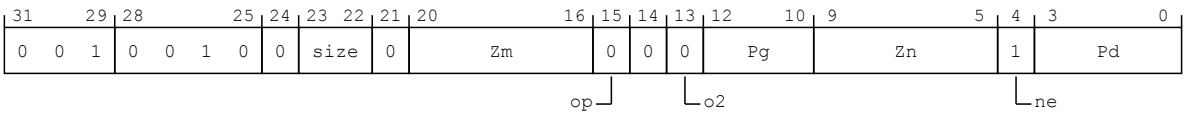

Encoding

```
CMPLO  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
CMPHI  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of CMP<cc> (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.111 CMPLS (vectors)

Compare unsigned lower than or same as vector, setting the condition flags

This instruction compares active unsigned integer elements in the first source vector being lower than or the same as the corresponding unsigned elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Higher or same

(FEAT_SVE || FEAT_SME)

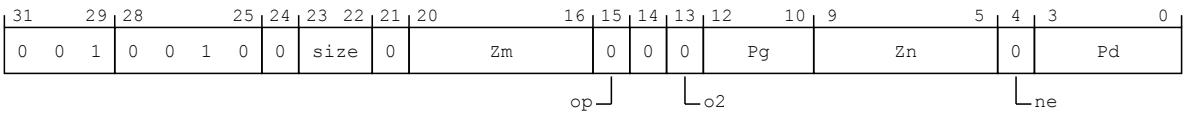

Encoding

```
CMPLS  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
CMPHS  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of CMP<cc> (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.112 CMPLT (vectors)

Compare signed less than vector, setting the condition flags

This instruction compares active signed integer elements in the first source vector being less than the corresponding signed elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Greater than

(FEAT_SVE || FEAT_SME)

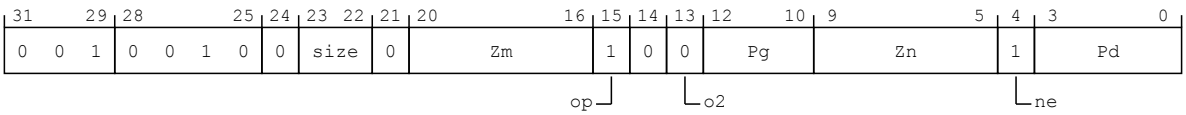

Encoding

```
CMPLT  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
CMPGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of CMP<cc> (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.113 CNOT

Logically invert boolean condition (predicated)

This instruction logically inverts the boolean value in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

Boolean TRUE is any non-zero value in a source, and one in a result element. Boolean FALSE is always zero.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	0 1 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
CNOT  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	0 1 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
CNOT  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = ZeroExtend{esize}(IsZeroBit{esize}(element));

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.114 CNT

Count non-zero bits (predicated)

This instruction determines the number of non-zero bits in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	0 1 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
CNT  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	0 1 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
CNT  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = BitCount(element)[esize-1:0];

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.115 CNTB, CNTD, CNTH, CNTW

Set scalar to multiple of predicate constraint element count

This instruction determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then places the result in the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword, and Word

Byte

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 0	1	0	imm4	1 1	1 0 0	0	pattern	Rd
			size						op		

Encoding

```
CNTB  <Xd>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let d : integer = UInt(Rd);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Doubleword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1	1	0	imm4	1 1	1 0 0	0	pattern	Rd
			size						op		

Encoding

```
CNTD  <Xd>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let d : integer = UInt(Rd);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Halfword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 1	1	0	imm4	1 1	1 0 0	0	pattern	Rd
			size						op		

Encoding

```
CNTH  <Xd>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let d : integer = UInt(Rd);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Word

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 0	1	0	imm4	1 1	1 0 0	0	pattern	Rd
			size						op		

Encoding

```
CNTW  <Xd>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let d : integer = UInt(Rd);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
X{64}(d) = (count * imm)[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.116 CNTP (predicate)

Set scalar to count of true predicate elements

This instruction determines the number of active and true elements in the source predicate and places the scalar result in the destination general-purpose register. Inactive predicate elements are not counted.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–14	13–10	9	8–5	4–0
0 0 1	0 0 1 0	1	size	1 0 0	0 0 0 (opc)	1 0	Pg	0	Pn	Rd

Encoding

```
CNTP  <Xd>, <Pg>, <Pn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Rd);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
var sum : bits(64) = Zeros{};

for e = 0 to elements-1 do
    if (ActivePredicateElement{PL}(mask, e, esize)
         && ActivePredicateElement{PL}(operand, e, esize)) then
        sum = sum + 1;
    end;
end;
X{64}(d) = sum;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.117 CNTP (predicate as counter)

Set scalar to count from predicate-as-counter

This instruction determines the number of true elements in the source predicate, up to the total number of elements in two or four vectors, and places the scalar result in the destination general-purpose register.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31–29	28–25	24	23–22	21–19	18–16	15–14	13–11	10	9	8–5	4–0
0 0 1	0 0 1 0	1	size	1 0 0	0 0 0 (opc)	1 0	0 0 0	vl	1	PNn	Rd

Encoding

```
CNTP  <Xd>, <PNn>.<T>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(PNn);
let d : integer = UInt(Rd);
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<PNn>

Is the name of the source scalable predicate register, with predicate-as-counter encoding, encoded in the “PNn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<vl>

Is the vl specifier, encoded in “vl”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let pred : bits(PL) = P{}(n);
let mask : bits(PL*4) = CounterToPredicate{}(pred[15:0]);
var sum : bits(64) = Zeros{};
let maxelements : integer = elements * width;

for e = 0 to maxelements-1 do
    if ActivePredicateElement{PL*4}(mask, e, esize) then
        sum = sum + 1;
    end;
end;
X{64}(d) = sum;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.118 COMPACT

Copy Active vector elements to lower-numbered elements

This instruction copies Active elements from the source vector to consecutive elements of the destination vector, in increasing order of element number, and sets any remaining elements to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SME2p2 is implemented.

It has encodings from 2 classes: Byte and halfword and Word and doubleword

Byte and halfword

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23	22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	0	sz	1	0	0 0 0	1	1 0	0	Pg	Zn	Zd

Encoding

```
COMPACT  <Zd>.<T>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Word and doubleword

(FEAT_SVE || FEAT_SME2p2)

31 29	28 25	24	23	22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	1	sz	1	0	0 0 0	1	1 0	0	Pg	Zn	Zd

Encoding

```
COMPACT  <Zd>.<T>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

For the “Byte and halfword” variant: is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

For the “Word and doubleword” variant: is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2p2) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = Zeros{};
var x : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand1[e*:esize];
        result[x*:esize] = element;
        x = x + 1;
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.119 CPY (immediate, zeroing)

Copy signed integer immediate to vector elements (zeroing)

This instruction copies a signed integer immediate to each Active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

This instruction is used by the alias MOV (immediate, zeroing).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 5	4 0
0 0 0	0 0 1 0	1	size	0 1	Pg	0	0	sh	imm8	Zd
							M			

Encoding

```
CPY  <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
var imm : integer = SInt(imm8);
if sh == '1' then imm = imm << 8; end;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<imm>

Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Alias Conditions

Alias	Is preferred when
MOV (immediate, zeroing)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let dest : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = imm[esize-1:0];
    else
        result[e*:esize] = dest[e*:esize];
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.120 CPY (immediate, merging)

Copy signed integer immediate to vector elements (merging)

This instruction copies a signed integer immediate to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

This instruction is used by the alias MOV (immediate, merging).

This instruction is used by the pseudo-instruction FMOV (zero, predicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 5	4 0
0 0 0	0 0 1 0	1	size	0 1	Pg	0	1	sh	imm8	Zd
							M			

Encoding

```
CPY  <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
var imm : integer = SInt(imm8);
if sh == '1' then imm = imm << 8; end;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<imm>

Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Alias Conditions

Alias	Is preferred when
FMOV (zero, predicated)	Never
MOV (immediate, merging)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let dest : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = imm[esize-1:0];
    else
        result[e*:esize] = dest[e*:esize];
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.121 CPY (scalar)

Copy general-purpose register to vector elements (predicated)

This instruction copies the general-purpose scalar source register to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (scalar, predicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	1 0 0	0	1 0	1	Pg	Rn	Zd

Encoding

```
CPY  <Zd>.<T>, <Pg>/M, <R><n|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<n|SP>

Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
MOV (scalar, predicated)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var result : bits(VL) = Z{}(d);
if AnyActiveElement{PL}(mask, esize) then
    var operand1 : bits(64);
    if n == 31 then
        operand1 = SP{64}();
    else
        operand1 = X{64}(n);
    end;

    for e = 0 to elements-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            result[e*:esize] = operand1[esize-1:0];
        end;
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.122 CPY (SIMD&FP scalar)

Copy SIMD&FP scalar register to vector elements (predicated)

This instruction copies the SIMD & floating-point scalar source register to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (SIMD&FP scalar, predicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 0 0	0	1 0	0	Pg	Vn	Zd

Encoding

```
CPY  <Zd>.<T>, <Pg>/M, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Vn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<V>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<V>
00	B
01	H
10	S
11	D

<n>

Is the number [0-31] of the source SIMD&FP register, encoded in the “Vn” field.

Alias Conditions

Alias	Is preferred when
MOV (SIMD&FP scalar, predicated)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(esize) = (if AnyActiveElement{PL}(mask, esize) then
                              V{esize}(n) else Zeros{esize});
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1;
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.123 CTERMEQ, CTERMNE

Compare and terminate loop

This instruction detects termination conditions in serialized vector loops by testing whether the comparison between the scalar source operands holds true. If the comparison is not successful, the instruction tests the state of the !Last condition flag (C), which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.

The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:

Condition	N	V	Meaning
GE	0	0	Continue loop (compare failed and last element not selected)
LT	0	1	Terminate loop (last element selected)
LT	1	0	Terminate loop (compare succeeded)
GE	1	1	Never generated

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.

It has encodings from 2 classes: Equal and Not equal

Equal

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–14	13–12	11–10	9–5	4	3–0
0 0 1	0 0 1 0	1	1	sz	1	Rm	0 0	1 0	0 0	Rn	0	0 0 0 0
			op								ne	

Encoding

```
CTERMEQ  <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let cmp_op : CmpOp = Cmp_EQ;
```

Not equal

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–14	13–12	11–10	9–5	4	3–0
0 0 1	0 0 1 0	1	1	sz	1	Rm	0 0	1 0	0 0	Rn	1	0 0 0 0
			op								ne	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
CTERMNE  <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let cmp_op : CmpOp = Cmp_NE;
```

Assembler Symbols

<R>

Is a width specifier, encoded in “sz”:

sz	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let operand1 : bits(esize) = X{}(n);
let operand2 : bits(esize) = X{}(m);
let element1 : integer = UInt(operand1);
let element2 : integer = UInt(operand2);
var term : boolean;

case cmp_op of
    when Cmp_EQ => term = element1 == element2;
    when Cmp_NE => term = element1 != element2;
end;
if term then
    PSTATE.N = '1';
    PSTATE.V = '0';
else
    PSTATE.N = '0';
    PSTATE.V = (NOT PSTATE.C);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.124 DECB, DECD, DECH, DECW (scalar)

Decrement scalar by multiple of predicate constraint element count

This instruction determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword, and Word

Byte

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 0 (size)	1	1	imm4	1 1	1 0 0	1 (D)	pattern	Rdn

Encoding

```
DECB  <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Doubleword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1 (size)	1	1	imm4	1 1	1 0 0	1 (D)	pattern	Rdn

Encoding

```
DECD  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Halfword

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 14	13 11	10	9 5	4 0
0 0 0	0 0 1 0	0	0 1	1	1	imm4	1 1	1 0 0	1	pattern	Rdn
			size						D		

Encoding

```
DECH  <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Word

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 14	13 11	10	9 5	4 0
0 0 0	0 0 1 0	0	1 0	1	1	imm4	1 1	1 0 0	1	pattern	Rdn
			size						D		

Encoding

```
DECW  <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let VL : integer{} = CurrentVL();
let operand1 : bits(64) = X{}(dn);
X{64}(dn) = operand1 - (count * imm);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.125 DECD, DECH, DECW (vector)

Decrement vector by multiple of predicate constraint element count

This instruction determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword, Halfword, and Word

Doubleword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1	1	1	imm4	1 1	0 0 0	1	pattern	Zdn
			size						D		

Encoding

```
DECD  <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Halfword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 1	1	1	imm4	1 1	0 0 0	1	pattern	Zdn
			size						D		

Encoding

```
DECH  <Zdn>.H{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Word

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 14	13 11	10	9 5	4 0
0 0 0	0 0 1 0	0	1 0	1	1	imm4	1 1	0 0 0	1	pattern	Zdn
			size						D		

Encoding

```
DECW  <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    result[e*:esize] = operand1[e*:esize] - (count * imm);

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.126 DECP (scalar)

Decrement scalar by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to decrement the scalar destination.

SVE

(FEAT_SVE || FEAT_SME)

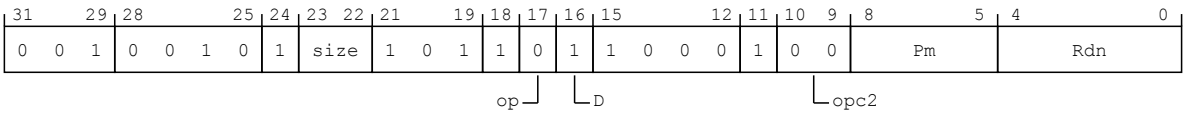

Encoding

```
DECP  <Xdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(64) = X{}(dn);
let operand2 : bits(PL) = P{}(m);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

X{64}(dn) = operand1 - count;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.127 DECP (vector)

Decrement vector by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to decrement all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

SVE

(FEAT_SVE || FEAT_SME)

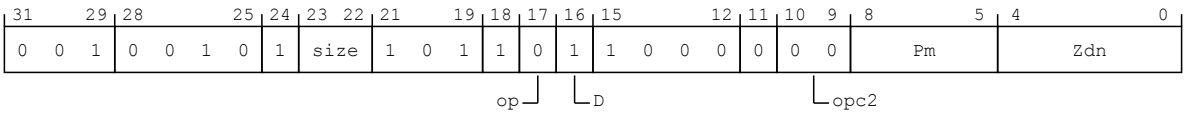

Encoding

```
DECP  <Zdn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(VL);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

for e = 0 to elements-1 do
    result[e*:esize] = operand1[e*:esize] - count;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.128 DUP (immediate)

Broadcast signed immediate to vector elements (unpredicated)

This instruction unconditionally broadcasts the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

This instruction is used by the alias MOV (immediate, unpredicated).

This instruction is used by the pseudo-instruction FMOV (zero, unpredicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0 0 1	0 0 1 0	1	size	1	1 1	0 0	0	1 1	sh	imm8	Zd
						opc					

Encoding

```
DUP  <Zd>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd);
var imm : integer = SInt(imm8);
if sh == '1' then imm = imm << 8; end;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Alias Conditions

Alias	Is preferred when
FMOV (zero, unpredicated)	Never
MOV (immediate, unpredicated)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let result : bits(VL) = Replicate{}(imm[esize-1:0]);
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.129 DUP (scalar)

Broadcast general-purpose register to vector elements (unpredicated)

This instruction unconditionally broadcasts the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (scalar, unpredicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0 0	0 0 0	0 0 1 1 1 0	Rn	Zd

Encoding

```
DUP  <Zd>.<T>, <R><n|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Rn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<n|SP>

Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
MOV (scalar, unpredicated)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var operand : bits(64);
if n == 31 then
    operand = SP{64}();
else
    operand = X{64}(n);
end;
var result : bits(VL);

for e = 0 to elements-1 do
    result[e*:esize] = operand[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.130 DUP (indexed)

Broadcast indexed element to vector (unpredicated)

This instruction unconditionally broadcasts the indexed source vector element into each element of the destination vector. This instruction is unpredicated.

The immediate element index is in the range of 0 to 63 (bytes), 31 (halfwords), 15 (words), 7 (doublewords) or 3 (quadwords). Selecting an element beyond the accessible vector length causes the destination vector to be set to zero.

This instruction is used by the alias MOV (SIMD&FP scalar, unpredicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	imm2	1	tsz	0 0 1 0 0 0	Zn	Zd

Encoding

```
DUP  <Zd>.<T>, <Zn>.<T>[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if tsz == '00000' then EndOfDecode(Decode_UNDEF); end;
let lsb : integer{} = LowestSetBit(tsz);
let esize : integer{} = 8 << lsb;
let imm : bits(7) = imm2::tsz;
let index : integer = UInt(imm[6:(lsb+1)]);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tsz”:

tsz	<T>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<imm>

Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in “imm2:tsz”.

Alias Conditions

Alias	Is preferred when
MOV (SIMD&FP scalar, unpredicated)	BitCount(imm2 : tsz) == 1
MOV (SIMD&FP scalar, unpredicated)	BitCount(imm2 : tsz) > 1

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = if index < elements then Z{VL}(n) else Zeros{VL};
var result : bits(VL);
var element : bits(esize);

if index >= elements then
    element = Zeros{esize};
else
    element = operand1[index*:esize];
end;
result = Replicate{VL}(element);

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.131 DUPM

Broadcast logical bitmask immediate to vector (unpredicated)

This instruction unconditionally broadcasts the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This instruction is used by the alias MOV.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 18	17 5	4 0
0 0 0	0 0 1 0	1	1 1	0 0	0 0	imm13	Zd

Encoding

```
DUPM  <Zd>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let d : integer = UInt(Zd);
var imm : bits(esize);
(imm, -) = DecodeBitMasks{esize}(imm13[12], imm13[5:0], imm13[11:6], TRUE);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
MOV	SVEMoveMaskPreferred(imm13)

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let result : bits(VL) = Replicate{}(imm);
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.132 DUPQ

Broadcast indexed element within each quadword vector segment (unpredicated)

This instruction unconditionally broadcasts the indexed element within each 128-bit source vector segment to all elements of the corresponding destination vector segment. This instruction is unpredicated.

The immediate element index is in the range of 0 to 15 (bytes), 7 (halfwords), 3 (words) or 1 (doublewords).

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20	19 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	0 0	1	i1	tsz	0 0 1 0 0 1	Zn	Zd

Encoding

```
DUPQ  <Zd>.<T>, <Zn>.<T>[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
if tsz == '0000' then EndOfDecode(Decode_UNDEF); end;
let lsb : integer{} = LowestSetBit(tsz);
let esize : integer{} = 8 << lsb;
let imm : bits(5) = i1::tsz;
let index : integer = UInt(imm[4:(lsb+1)]);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tsz”:

tsz	<T>
0000	RESERVED
xxx1	B
xx10	H
x100	S
1000	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate index, in the range 0 to one less than the number of elements in 128 bits, encoded in “i1:tsz”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
var element : bits(esize);

for s = 0 to segments-1 do
    element = operand[(s * elements + index)*:esize];
    result[s*:128] = Replicate{128}(element);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.133 EON

Bitwise exclusive-OR with inverted immediate (unpredicated)

This instruction performs a bitwise exclusive-OR between an inverted immediate and each 64-bit element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of EOR (immediate). This means:

- The encodings in this description are named to match the encodings of EOR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of EOR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 18	17 5	4 0
0 0 0	0 0 1 0	1	0 1	0 0	0 0	imm13	Zdn
			opc				

Encoding

```
EON  <Zdn>.<T>, <Zdn>.<T>, #<const>
```

is equivalent to

```
EOR  <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of EOR (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of EOR (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.134 EOR3

Bitwise exclusive-OR between three vectors

This instruction performs a bitwise exclusive-OR between the corresponding elements of the three source vectors, and destructively places the results in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	0 0	1	Zm	0 0 1	1 1	0	Zk	Zdn
			opc					o2		

Encoding

```
EOR3  <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let m : integer = UInt(Zm);
let k : integer = UInt(Zk);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the third source scalable vector register, encoded in the “Zk” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(k);

Z{VL}(dn) = operand1 XOR operand2 XOR operand3;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.135 EOR (predicates)

Bitwise exclusive-OR predicates

This instruction performs a bitwise exclusive-OR between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This instruction is used by the alias NOT (predicate).

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 10	9	8 5	4	3 0
0 0 1	0 0 1 0	1	0	0	0 0	Pm	0 1	Pg	1	Pn	0	Pd
		op		S					o2		o3	

Encoding

```
EOR  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
NOT (predicate)	Pm == Pg

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 XOR element2);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.136 EOR (vectors, predicated)

Bitwise exclusive-OR (predicated)

This instruction performs a bitwise exclusive-OR between the active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 1	0 0 1	0 0 0	Pg	Zm	Zdn
						opc				

Encoding

```
EOR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element1 XOR element2;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.137 EOR (immediate)

Bitwise exclusive-OR with immediate (unpredicated)

This instruction performs a bitwise exclusive-OR between an immediate and each 64-bit element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction EON.

SVE

(FEAT_SVE || FEAT_SME)

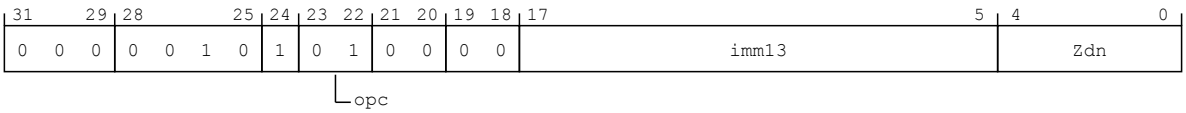

Encoding

```
EOR  <Zdn>.<T>, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn);
var imm : bits(64);
(imm, -) = DecodeBitMasks{64}(imm13[12], imm13[5:0], imm13[11:6], TRUE);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand[e*:64];
    result[e*:64] = element1 XOR imm;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.138 EOR (vectors, unpredicated)

Bitwise exclusive-OR (unpredicated)

This instruction performs a bitwise exclusive-OR between the elements of the first source vector and the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

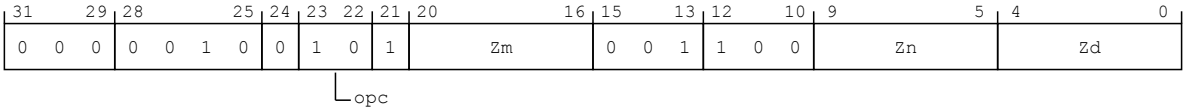

Encoding

```
EOR  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);

Z{VL}(d) = operand1 XOR operand2;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.139 EORBT

Interleaving exclusive-OR (bottom, top)

This instruction performs an interleaving exclusive-OR between the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register, placing the result in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 14	13 11	10	9 5	4 0
0 1 0	0 0 1 0	1	size	0	Zm	1 0	0 1 0	0	Zn	Zd
								tb		

Encoding

```
EORBT  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[(2*e + sel1)*:esize];
    let element2 : bits(esize) = operand2[(2*e + sel2)*:esize];
    result[(2*e + sel1)*:esize] = element1 XOR element2;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.140 EORQV

Bitwise exclusive-OR reduction of quadword vector segments

This instruction performs a bitwise exclusive-OR between elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as all zeros.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 1 1	0 1 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
EORQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : bits(esize);

for e = 0 to elempersegment-1 do
    dtmp = Zeros{esize};
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = dtmp XOR stmp[e*:esize];
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.141 EORS

Bitwise exclusive-OR predicates, setting the condition flags

This instruction performs a bitwise exclusive-OR between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This instruction is used by the alias NOTS.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

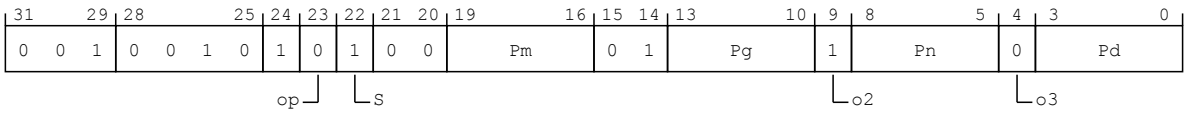

Encoding

```
EORS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
NOTS	Pm == Pg

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 XOR element2);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.142 EORTB

Interleaving exclusive-OR (top, bottom)

This instruction performs an interleaving exclusive-OR between the odd-numbered elements of the first source vector register and the even-numbered elements of the second source vector register, placing the result in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

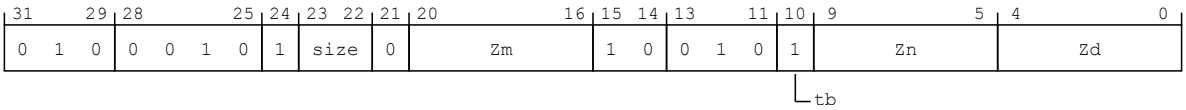

Encoding

```
EORTB  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 1;
let sel2 : integer = 0;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[(2*e + sel1)*:esize];
    let element2 : bits(esize) = operand2[(2*e + sel2)*:esize];
    result[(2*e + sel1)*:esize] = element1 XOR element2;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.143 EORV

Bitwise exclusive-OR reduction to scalar

This instruction performs a bitwise exclusive-OR between the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 1 0	0 1 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
EORV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(esize) = Zeros{esize};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result = result XOR operand[e*:esize];

    end;
end;
V{esize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.144 EXPAND

Copy lower-numbered vector elements to Active elements

This instruction copies consecutive elements from the source vector to Active elements of the destination vector, in increasing order of element number. Inactive elements in the destination vector register are set to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SME2p2 is implemented.

SVE2

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	1	0 0 0	1	1 0	0	Pg	Zn	Zd

Encoding

```
EXPAND  <Zd>.<T>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SME2p2) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL);
var x : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[x*:esize];
        x = x + 1;
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.145 EXT

Extract vector from pair of vectors

This instruction copies the indexed byte up to the last byte of the first source vector to the bottom of the result vector, then fills the remainder of the result starting from the first byte of the second source vector. The result is destructively placed in the destination and first source vector, or constructively placed in the destination vector. This instruction is unpredicated.

An index that is greater than or equal to the vector length in bytes is treated as zero, resulting in the first source vector being copied to the result unchanged.

It has encodings from 2 classes: Constructive and Destructive

Constructive

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	1 0	1	1	imm8h	0 0 0	imm8l	Zn	Zd

Encoding

```
EXT  <Zd>.B, { <Zn1>.B, <Zn2>.B }, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dst : integer = UInt(Zd);
let s1 : integer = UInt(Zn);
let s2 : integer = (s1 + 1) MOD 32;
let position : integer{} = UInt(imm8h::imm8l) * 8;
```

Destructive

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	1 0	0	1	imm8h	0 0 0	imm8l	Zm	Zdn

Encoding

```
EXT  <Zdn>.B, <Zdn>.B, <Zm>.B, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dst : integer = UInt(Zdn);
let s1 : integer = dst;
let s2 : integer = UInt(Zm);
let position : integer{} = UInt(imm8h::imm8l) * 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded in the “Zn” field.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded in the “Zn” field.

<imm>

Is the unsigned immediate operand, in the range 0 to 255, encoded in the “imm8h:imm8l” fields.

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(s1);
let operand2 : bits(VL) = Z{}(s2);
var result : bits(VL);

let concat : bits(VL*2) = operand2 :: operand1;

if position >= VL then
    result = concat[VL-1:0];
else
    result = concat[(position+VL)-1:position];
end;

Z{VL}(dst) = result;
```

Operational Information

For the Constructive variant:

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

For the Destructive variant:

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The destructive variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the destructive variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.146 EXTQ

Extract vector segment from each pair of quadword vector segments

This instruction copies, for each 128-bit vector segment of the result, the indexed byte up to and including the last byte of the corresponding first source vector segment to the bottom of the result segment, and then fills the remainder of the result segment starting from the first byte of the corresponding second source vector segment. The result segments are destructively placed in the corresponding first source vector segment. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20	19 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	0 1	1	0	imm4	0 0 1 0 0 1	Zm	Zdn

Encoding

```
EXTQ  <Zdn>.B, <Zdn>.B, <Zm>.B, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let position : integer{} = UInt(imm4) << 3;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<imm>

Is the unsigned immediate operand, in the range 0 to 15, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let concat : bits(256) = operand2[s*:128] :: operand1[s*:128];
    result[s*:128] = concat[position+127:position];
end;

Z{VL}(dn) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.147 F1CVT, F2CVT

8-bit floating-point convert to half-precision

This instruction converts each even-numbered 8-bit floating-point element of the source vector to half-precision while downscaling the value, and places the results in the overlapping 16-bit elements of the destination vector. F1CVT scales the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$. F2CVT scales the values by $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$.

The 8-bit floating-point encoding format for F1CVT is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for F2CVT is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: F1CVT and F2CVT

F1CVT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	0 (L)	0 0 1 1	0 0 (opc)	Zn	Zd

Encoding

```
F1CVT  <Zd>.H, <Zn>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 8;
let d_esize : integer{} = 16;
let issrc2 : boolean = FALSE;
```

F2CVT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	0 (L)	0 0 1 1	0 1 (opc)	Zn	Zd

Encoding

```
F2CVT  <Zd>.H, <Zn>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 8;
let d_esize : integer{} = 16;
let issrc2 : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    let res : bits(d_esize) = FP8ConvertFP(element[s_esize-1:0], issrc2, FPCR(), FPMR());
    result[e*:esize] = ZeroExtend{esize}(res);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.148 F1CVTLT, F2CVTLT

8-bit floating-point convert to half-precision (top)

This instruction converts each odd-numbered 8-bit floating-point element of the source vector to half-precision while downscaling the value, and places the results in the overlapping 16-bit elements of the destination vector. F1CVTLT scales the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$. F2CVTLT scales the values by $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$.

The 8-bit floating-point encoding format for F1CVTLT is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for F2CVTLT is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: F1CVTLT and F2CVTLT

F1CVTLT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	1	0 0 1 1	0 0	Zn	Zd
						L		opc		

Encoding

```
F1CVTLT  <Zd>.H, <Zn>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let issrc2 : boolean = FALSE;
```

F2CVTLT

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 0	1	0 0 1 1	0 1	Zn	Zd
						L		opc		

Encoding

```
F2CVTLT  <Zd>.H, <Zn>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let issrc2 : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(esize DIV 2) = operand[(2*e + 1)*:(esize DIV 2)];
    result[e*:esize] = FP8ConvertFP(element, issrc2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.149 FABD

Floating-point absolute difference (predicated)

This instruction calculates the absolute difference of active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	size	0	0	1 0 0 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FABD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPAbs{esize}(FPSub{esize}(element1, element2, FPCR()), FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.150 FABS

Floating-point absolute value (predicated)

This instruction takes the absolute value of each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	1 0 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
FABS  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	1 0 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
FABS  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPAbs{esize}(element, FPCR());

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.151 FAC<cc>

Floating-point absolute compare

This instruction compares active absolute values of floating-point elements in the first source vector with the corresponding absolute values of elements in the second source vector, and places the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

<cc>	Comparison
GE	greater than or equal
GT	greater than
LE	less than or equal
LT	less than

This instruction is used by the pseudo-instructions FACLE and FACLT.

It has encodings from 2 classes: Greater than and Greater than or equal

Greater than

(FEAT_SVE || FEAT_SME)

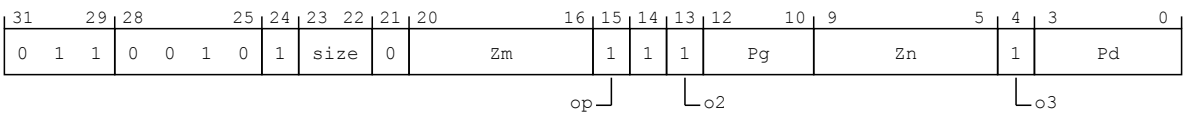

Encoding

```
FACGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
```

Greater than or equal

(FEAT_SVE || FEAT_SME)

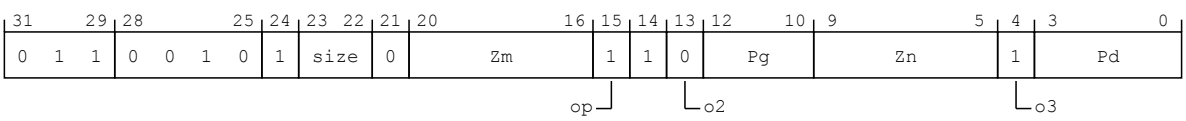

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FACGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element1 : bits(esize) = FPAbs{}(operand1[e*:esize], FPCR());
        let element2 : bits(esize) = FPAbs{}(operand2[e*:esize], FPCR());
        var res : boolean;
        case cmp_op of
            when Cmp_GE => res = FPCompareGE{esize}(element1, element2, FPCR());
            when Cmp_GT => res = FPCompareGT{esize}(element1, element2, FPCR());
        end;
        let pbit : bit = if res then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.152 FACLE

Floating-point absolute compare less than or equal

This instruction compares active absolute values of floating-point elements in the first source vector being less than or equal to the corresponding absolute values of elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Greater than or equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 1 1		0 0 1 0		1	size	0	Zm		1	1	0	Pg		Zn		1	Pd	
									op		o2					o3		

Encoding

```
FACLE  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
FACGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.

Operational Information

The description of FAC<cc> gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.153 FACLT

Floating-point absolute compare less than

This instruction compares active absolute values of floating-point elements in the first source vector being less than the corresponding absolute values of elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Greater than

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 1 1		0 0 1 0		1	size	0	Zm		1	1	1	Pg		Zn		1	Pd	
									op		o2					o3		

Encoding

```
FACLT  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
FACGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.

Operational Information

The description of FAC<cc> gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.154 FADD (immediate)

Floating-point add immediate (predicated)

This instruction adds an immediate to each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 6	5	4 0
0 1 1	0 0 1 0	1	size	0	1 1	0 0 0	1 0 0	Pg	0 0 0 0	i1	Zdn
						opc					

Encoding

```
FADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then FPPointFive{esize}('0') else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPAdd{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.155 FADD (vectors, predicated)

Floating-point add (predicated)

This instruction adds active floating-point elements of the first source vector to the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

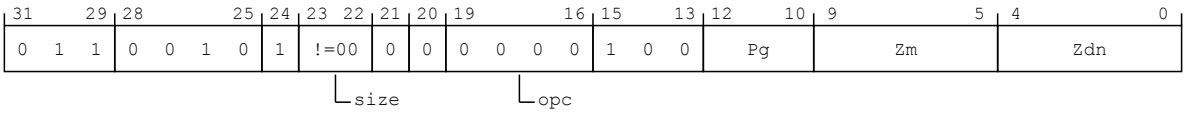

Encoding

```
FADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPAdd{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.156 FADD (vectors, unpredicated)

Floating-point add (unpredicated)

This instruction adds all floating-point elements of the first source vector to the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	Zm	0 0 0	0 0 0 (opc)	Zn	Zd

Encoding

```
FADD  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPAdd{esize}(element1, element2, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.157 FADDA

Floating-point add strictly-ordered reduction, accumulating in scalar

This instruction adds the floating-point values in a SIMD&FP scalar source and all active lanes of the vector source and destructively places the result in the SIMD&FP scalar source register. Vector elements are processed strictly in order from low to high, with the scalar source providing the initial value. Inactive elements in the source vector are ignored.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24	23–22	21–19	18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 1 1	0	0 0 (opc)	0 0 1	Pg	Zm	Vdn

Encoding

```
FADDA  <V><dn>, <Pg>, <V><dn>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Vdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<dn>

Is the number [0-31] of the source and destination SIMD&FP register, encoded in the “Vdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(esize) = V{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(esize) = operand1;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand2[e*:esize];
        result = FPAdd{esize}(result, element, FPCR());
    end;
end;

V{esize}(dn) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.158 FADDP

Floating-point add pairwise

This instruction adds pairs of adjacent floating-point elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	0 0 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FADDP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL) = Z{}(dn);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if IsEven(e) then
            element1 = operand1[(e + 0)*:esize];
            element2 = operand1[(e + 1)*:esize];
        else
            element1 = operand2[(e - 1)*:esize];
            element2 = operand2[(e + 0)*:esize];
        end;
        result[e*:esize] = FPAdd{esize}(element1, element2, FPCR());
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.159 FADDQV

Floating-point add recursive reduction of quadword vector segments

This instruction performs a floating-point addition of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as +0.0.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	0 0 0 (opc)	1 0 1	Pg	Zn	Vd

Encoding

```
FADDQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	RESERVED
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer{} = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let segbits : integer{} = segments*esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPZero{esize}('0');
var result : bits(128) = Zeros{};

for e = 0 to elempersegment-1 do
    var stmp : bits(segbits);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp[s*:esize] = operand[(s * elempersegment + e)*:esize];
        else
            stmp[s*:esize] = identity;
        end;
    end;
    result[e*:esize] = FPReduce{esize, segbits}(ReduceOp_FADD, stmp, FPCR());
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.160 FADDV

Floating-point add recursive reduction to scalar

This instruction adds horizontally the floating-point values over all lanes of a vector using a recursive pairwise reduction, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 0	0 0 0 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
FADDV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPZero{esize}('0');

V{esize}(d) = FPReducePredicated{esize, VL, PL}(ReduceOp_FADD, operand, mask,
                                                                identity, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.161 FAMAX

Floating-point absolute maximum (predicated)

This instruction determines the maximum absolute value from active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

Inactive elements in the destination vector register remain unmodified.

SVE2

((FEAT_SVE2 || FEAT_SME2) && FEAT_FAMINMAX)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0	1 1 1 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FAMAX  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FAMINMAX)) then EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPAbsMax{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.162 FAMIN

Floating-point absolute minimum (predicated)

This instruction determines the minimum absolute value from active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

Inactive elements in the destination vector register remain unmodified.

SVE2

((FEAT_SVE2 || FEAT_SME2) && FEAT_FAMINMAX)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0	1 1 1 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FAMIN  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FAMINMAX)) then EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPAbsMin{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.163 FCADD

Floating-point complex add (predicated)

This instruction adds the real and imaginary components of the active floating-point complex numbers from the first source vector to the complex numbers from the second source vector that have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. The results are destructively placed in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	size	0 0 0 0 0	rot	1 0 0	Pg	Zm	Zdn

Encoding

```
FCADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let sub_i : boolean = (rot == '0');
let sub_r : boolean = (rot == '1');
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
0	#90
1	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let pairs : integer = VL DIV (2 * esize);
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for p = 0 to pairs-1 do
    var acc_r : bits(esize) = operand1[(2 * p + 0)*:esize];
    var acc_i : bits(esize) = operand1[(2 * p + 1)*:esize];
    if ActivePredicateElement{PL}(mask, 2 * p + 0, esize) then
        var elt2_i : bits(esize) = operand2[(2 * p + 1)*:esize];
        if sub_i then elt2_i = FPNeg{esize}(elt2_i, FPCR()); end;
        acc_r = FPAdd{esize}(acc_r, elt2_i, FPCR());
    end;
    if ActivePredicateElement{PL}(mask, 2 * p + 1, esize) then
        var elt2_r : bits(esize) = operand2[(2 * p + 0)*:esize];
        if sub_r then elt2_r = FPNeg{esize}(elt2_r, FPCR()); end;
        acc_i = FPAdd{esize}(acc_i, elt2_r, FPCR());
    end;
    result[(2 * p + 0)*:esize] = acc_r;
    result[(2 * p + 1)*:esize] = acc_i;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.164 FCLAMP

Floating-point clamp to minimum/maximum number

This instruction clamps each floating-point element in the destination vector to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively places the clamped results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows for each minimum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31–29	28–25	24	23–22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0	0	!=00 (size)	1	Zm	0 0 1 0 0 1	Zn	Zd

Encoding

```
FCLAMP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var result : bits(VL);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    let element3 : bits(esize) = operand3[e*:esize];
    let maxelement : bits(esize) = FPMaxNum{}(element1, element3, FPCR());
    result[e*:esize] = FPMinNum{esize}(maxelement, element2, FPCR());
end;
Z{VL}(d) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.165 FCM<cc> (zero)

Floating-point compare with zero

This instruction compares active floating-point elements in the source vector with zero, and places the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

<cc>	Comparison
EQ	equal
GE	greater than or equal
GT	greater than
LE	less than or equal
LT	less than
NE	not equal
UO	unordered

It has encodings from 6 classes: Equal, Greater than, Greater than or equal, Less than, Less than or equal, and Not equal

Equal

(FEAT_SVE || FEAT_SME)

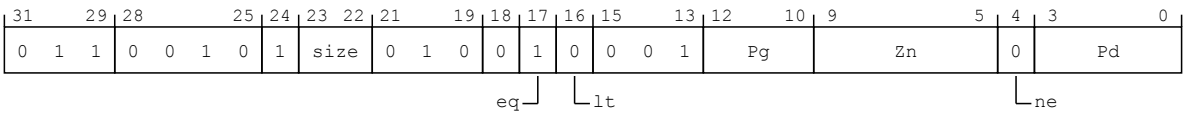

Encoding

```
FCMEQ  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let op : CmpOp = Cmp_EQ;
```

Greater than

(FEAT_SVE || FEAT_SME)

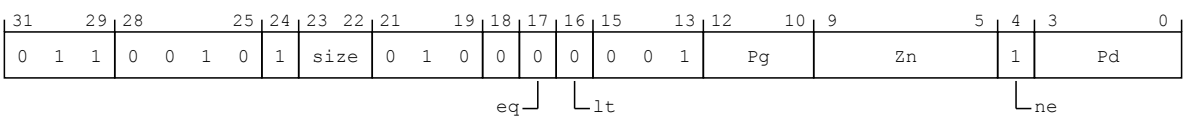

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FCMGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let op : CmpOp = Cmp_GT;
```

Greater than or equal

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 19	18	17	16	15 13	12 10	9 5	4	3 0
0 1 1	0 0 1 0	1	size	0 1 0	0	0	0	0 0 1	Pg	Zn	0	Pd
						eq	lt				ne	

Encoding

```
FCMGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let op : CmpOp = Cmp_GE;
```

Less than

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 19	18	17	16	15 13	12 10	9 5	4	3 0
0 1 1	0 0 1 0	1	size	0 1 0	0	0	1	0 0 1	Pg	Zn	0	Pd
						eq	lt				ne	

Encoding

```
FCMLT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let op : CmpOp = Cmp_LT;
```

Less than or equal

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18	17	16	15–13	12–10	9–5	4	3–0
0 1 1	0 0 1 0	1	size	0 1 0	0	0	1	0 0 1	Pg	Zn	1	Pd
						eq	lt				ne	

Encoding

```
FCMLE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let op : CmpOp = Cmp_LE;
```

Not equal

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18	17	16	15–13	12–10	9–5	4	3–0
0 1 1	0 0 1 0	1	size	0 1 0	0	1	1	0 0 1	Pg	Zn	0	Pd
						eq	lt				ne	

Encoding

```
FCMNE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let op : CmpOp = Cmp_NE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        var res : boolean;
        case op of
            when Cmp_EQ => res = FPCompareEQ{esize}(element, 0[esize-1:0], FPCR());
            when Cmp_GE => res = FPCompareGE{esize}(element, 0[esize-1:0], FPCR());
            when Cmp_GT => res = FPCompareGT{esize}(element, 0[esize-1:0], FPCR());
            when Cmp_NE => res = FPCompareNE{esize}(element, 0[esize-1:0], FPCR());
            when Cmp_LT => res = FPCompareGT{esize}(0[esize-1:0], element, FPCR());
            when Cmp_LE => res = FPCompareGE{esize}(0[esize-1:0], element, FPCR());
        end;
        let pbit : bit = if res then '1' else '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.166 FCM<cc> (vectors)

Floating-point compare

This instruction compares active floating-point elements in the first source vector with the corresponding elements in the second source vector, and places the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

<cc>	Comparison
EQ	equal
GE	greater than or equal
GT	greater than
NE	not equal
UO	unordered

This instruction is used by the pseudo-instructions FCMLE (vectors) and FCMLT (vectors).

It has encodings from 5 classes: Equal, Greater than, Greater than or equal, Not equal, and Unordered

Equal

(FEAT_SVE || FEAT_SME)

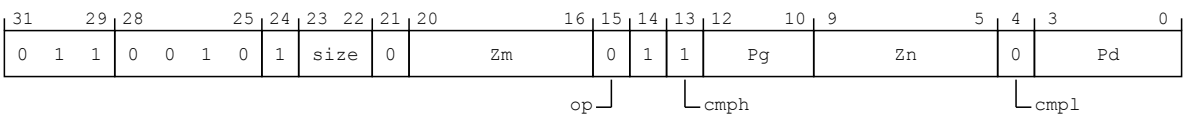

Encoding

```
FCMEQ  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_EQ;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Greater than

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 1 1		0 0 1 0		1	size	0	Zm		0	1	0	Pg		Zn		1	Pd	
									op		cmph					cmpl		

Encoding

```
FCMGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GT;
```

Greater than or equal

(FEAT_SVE || FEAT_SME)

31	29	28	25	24	23 22	21	20	16	15	14	13	12	10	9	5	4	3	0
0 1 1		0 0 1 0		1	size	0	Zm		0	1	0	Pg		Zn		0	Pd	
									op		cmph					cmpl		

Encoding

```
FCMGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_GE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Not equal

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 1 1	0 0 1 0	1	size	0	Zm	0	1	1	Pg	Zn	1	Pd
						op		cmph			cmpl	

Encoding

```
FCMNE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_NE;
```

Unordered

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4	3 0
0 1 1	0 0 1 0	1	size	0	Zm	1	1	0	Pg	Zn	0	Pd
						op		o2			o3	

Encoding

```
FCMUO  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Pd);
let cmp_op : CmpOp = Cmp_UN;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        var res : boolean;
        case cmp_op of
            when Cmp_EQ => res = FPCompareEQ{esize}(element1, element2, FPCR());
            when Cmp_GE => res = FPCompareGE{esize}(element1, element2, FPCR());
            when Cmp_GT => res = FPCompareGT{esize}(element1, element2, FPCR());
            when Cmp_UN => res = FPCompareUN{esize}(element1, element2, FPCR());
            when Cmp_NE => res = FPCompareNE{esize}(element1, element2, FPCR());
            when Cmp_LT => res = FPCompareGT{esize}(element2, element1, FPCR());
            when Cmp_LE => res = FPCompareGE{esize}(element2, element1, FPCR());
        end;
        let pbit : bit = if res then '1' else '0';
        result[e*:psize] = ZeroExtend{psize}(pbit);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.167 FCMLA (vectors)

Floating-point complex multiply-add (predicated)

This instruction multiplies the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

This instruction then destructively adds the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	size	0	Zm	0	rot	Pg	Zn	Zda

Encoding

```
FCMLA  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let neg_i : boolean = (rot[1] == '1');
let neg_r : boolean = (rot[0] != rot[1]);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let pairs : integer = VL DIV (2 * esize);
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for p = 0 to pairs-1 do
    var addend_r : bits(esize) = op3[(2 * p + 0)*:esize];
    var addend_i : bits(esize) = op3[(2 * p + 1)*:esize];
    if ActivePredicateElement{PL}(mask, 2 * p + 0, esize) then
        let elt1_a : bits(esize) = op1[(2*p + sel_a)*:esize];
        let elt2_a : bits(esize) = (if neg_r then FPNeg{esize}(op2[(2 * p + sel_a)*:esize], FPCR())
                                    else op2[(2 * p + sel_a)*:esize]);
        addend_r = FPMulAdd{esize}(addend_r, elt1_a, elt2_a, FPCR());
    end;
    if ActivePredicateElement{PL}(mask, 2 * p + 1, esize) then
        let elt1_a : bits(esize) = op1[(2 * p + sel_a)*:esize];
        let elt2_b : bits(esize) = (if neg_i then FPNeg{esize}(op2[(2 * p + sel_b)*:esize], FPCR())
                                    else op2[(2 * p + sel_b)*:esize]);
        addend_i = FPMulAdd{esize}(addend_i, elt1_a, elt2_b, FPCR());
    end;
    result[(2 * p + 0)*:esize] = addend_r;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[(2 * p + 1)*:esize] = addend_i;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.168 FCMLA (indexed)

Floating-point complex multiply-add by indexed element

This instruction multiplies the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

This instruction then destructively adds the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

The complex numbers within the second source vector are specified using an immediate index that selects the same complex number position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex numbers per 128-bit segment. This instruction is unpredicated.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 1	rot	Zn	Zda
			size							

Encoding

```
FCMLA  <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let neg_i : boolean = (rot[1] == '1');
let neg_r : boolean = (rot[0] != rot[1]);
```

Single-precision

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 1	rot	Zn	Zda
			size							

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FCMLA  <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let neg_i : boolean = (rot[1] == '1');
let neg_r : boolean = (rot[0] != rot[1]);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “Half-precision” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “Single-precision” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “Half-precision” variant: is the index of a Real and Imaginary pair, in the range 0 to 3, encoded in the “i2” field.

For the “Single-precision” variant: is the index of a Real and Imaginary pair, in the range 0 to 1, encoded in the “i1” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let pairspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for p = 0 to pairs-1 do
    let segmentbase : integer = p - (p MOD pairspersegment);
    let s : integer = segmentbase + index;
    var addend_r : bits(esize) = operand3[(2 * p + 0)*:esize];
    var addend_i : bits(esize) = operand3[(2 * p + 1)*:esize];
    let elt1_a : bits(esize) = operand1[(2 * p + sel_a)*:esize];
    var elt2_a : bits(esize)   = operand2[(2 * s + sel_a)*:esize];
    var elt2_b : bits(esize)   = operand2[(2 * s + sel_b)*:esize];
    if neg_r then elt2_a = FPNeg{esize}(elt2_a, FPCR()); end;
    if neg_i then elt2_b = FPNeg{esize}(elt2_b, FPCR()); end;
    addend_r = FPMulAdd{esize}(addend_r, elt1_a, elt2_a, FPCR());
    addend_i = FPMulAdd{esize}(addend_i, elt1_a, elt2_b, FPCR());
    result[(2 * p + 0)*:esize] = addend_r;
    result[(2 * p + 1)*:esize] = addend_i;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.169 FCMLE (vectors)

Floating-point compare less than or equal to vector

This instruction compares active floating-point elements in the first source vector being less than or equal to the corresponding elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Greater than or equal

(FEAT_SVE || FEAT_SME)

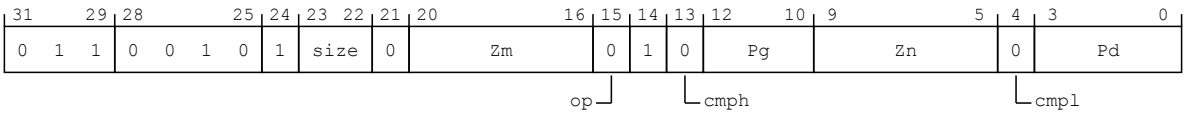

Encoding

```
FCMLE  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
FCMGE  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of FCM<cc> (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.170 FCMLT (vectors)

Floating-point compare less than vector

This instruction compares active floating-point elements in the first source vector being less than the corresponding elements in the second source vector, and places the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Greater than

(FEAT_SVE || FEAT_SME)

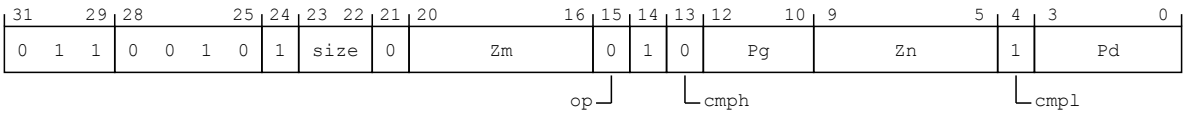

Encoding

```
FCMLT  <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
FCMGT  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of FCM<cc> (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.171 FCPY

Copy floating-point immediate to vector elements (predicated)

This instruction copies a floating-point immediate into each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias FMOV (immediate, predicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15 13	12 5	4 0
0 0 0	0 0 1 0	1	size	0 1	Pg	1 1 0	imm8	Zd

Encoding

```
FCPY  <Zd>.<T>, <Pg>/M, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let d : integer = UInt(Zd);
let imm : bits(esize) = VFPExpandImm{}(imm8);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<const>

Is a floating-point immediate value expressible as $\pm n \div 16 \times 2^r$, where n and r are integers such that $16 \leq n \leq 31$ and $-3 \leq r \leq 4$, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
FMOV (immediate, predicated)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = imm;
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.172 FCVT

Floating-point convert (predicated)

This instruction converts the size and precision of each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 12 classes: Half-precision to single-precision, merging, Half-precision to single-precision, zeroing, Half-precision to double-precision, merging, Half-precision to double-precision, zeroing, Single-precision to half-precision, merging, Single-precision to half-precision, zeroing, Single-precision to double-precision, merging, Single-precision to double-precision, zeroing, Double-precision to half-precision, merging, Double-precision to half-precision, zeroing, Double-precision to single-precision, merging, and Double-precision to single-precision, zeroing

Half-precision to single-precision, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 0 (opc)	0	0 1 0	0 1 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVT  <Zd>.S, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 32;
let merging : boolean = TRUE;
```

Half-precision to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 1 1	0 1 0	1	0 1 (opc2)	Pg	Zn	Zd

Encoding

```
FCVT  <Zd>.S, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 32;
let merging : boolean = FALSE;
```

Half-precision to double-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	0 1 0	0 1	1 0 1	Pg	Zn	Zd
			opc			opc2				

Encoding

```
FCVT  <Zd>.D, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 64;
let merging : boolean = TRUE;
```

Half-precision to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	0 1 0	1	0 1	Pg	Zn	Zd
			opc				opc2			

Encoding

```
FCVT  <Zd>.D, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 64;
let merging : boolean = FALSE;
```

Single-precision to half-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 0	0	0 1 0	0 0	1 0 1	Pg	Zn	Zd
			opc			opc2				

Encoding

```
FCVT  <Zd>.H, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 16;
let merging : boolean = TRUE;
```

Single-precision to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	0 1 1	0 1 0	1	0 0	Pg	Zn	Zd
			opc				opc2			

Encoding

```
FCVT  <Zd>.H, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 16;
let merging : boolean = FALSE;
```

Single-precision to double-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	0 1 0	1 1	1 0 1	Pg	Zn	Zd
			opc			opc2				

Encoding

```
FCVT  <Zd>.D, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let merging : boolean = TRUE;
```

Single-precision to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	0 1 0	1	1 1	Pg	Zn	Zd
			opc				opc2			

Encoding

```
FCVT  <Zd>.D, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let merging : boolean = FALSE;
```

Double-precision to half-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	0 1 0	0 0	1 0 1	Pg	Zn	Zd
			opc			opc2				

Encoding

```
FCVT  <Zd>.H, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 16;
let merging : boolean = TRUE;
```

Double-precision to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	0 1 0	1	0 0	Pg	Zn	Zd
			opc				opc2			

Encoding

```
FCVT  <Zd>.H, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 16;
let merging : boolean = FALSE;
```

Double-precision to single-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1 (opc)	0	0 1 0	1 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVT  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let merging : boolean = TRUE;
```

Double-precision to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1 (opc)	0 1 1	0 1 0	1	1 0 (opc2)	Pg	Zn	Zd

Encoding

```
FCVT  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        let res : bits(d_esize) = FPConvertSVE{d_esize, s_esize}(element[s_esize-1:0], FPCR());
        result[e*:esize] = ZeroExtend{esize}(res);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Half-precision to single-precision, merging, Half-precision to double-precision, merging, Single-precision to half-precision, merging, Single-precision to double-precision, merging, Double-precision to half-precision, merging, Double-precision to single-precision, merging variants:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.173 FCVTLT

Floating-point widening convert (top, predicated)

This instruction converts odd-numbered floating-point elements from the source vector to the next higher precision, and places the results in the active overlapping double-width elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 4 classes: Half-precision to single-precision, merging, Half-precision to single-precision, zeroing, Single-precision to double-precision, merging, and Single-precision to double-precision, zeroing

Half-precision to single-precision, merging

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 0 1 0	0 1 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTLT  <Zd>.S, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Half-precision to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 0 0 0	0 1 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTLT  <Zd>.S, <Pg>/Z, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision to double-precision, merging

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1 (opc)	0 0 1 0	1 1 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTLT  <Zd>.D, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Single-precision to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1 (opc)	0 0 0 0	1 1 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTLT  <Zd>.D, <Pg>/Z, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let halfesize : integer{} = esize DIV 2;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(halfesize) = operand[(2*e + 1)*:halfesize];
        result[e*:esize] = FPConvertSVE{esize, halfesize}(element, FPCR());
    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.174 FCVTN

Half-precision convert to interleaved 8-bit floating-point

This instruction converts each half-precision element of the pair of source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE[4:0])}}$, and places the two-way interleaved results in the corresponding 8-bit elements of the destination vector. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–16	15–12	11–10	9–6	5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 1 0	0 0 1 1	0 0	Zn	0	Zd
							opc			

Encoding

```
FCVTN  <Zd>.B, { <Zn1>.H-<Zn2>.H }
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[(2*e + 0)*:8] = FPConvertFP8{8, 16}(element1, FPCR(), FPMR());
    result[(2*e + 1)*:8] = FPConvertFP8{8, 16}(element2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.175 FCVTNB

Single-precision convert to interleaved 8-bit floating-point (bottom)

This instruction converts each single-precision element of the pair of source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE)}}$, and places the two-way interleaved results in the corresponding even-numbered 8-bit elements of the destination vector, zeroing the odd-numbered elements. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31–29	28–25	24	23–22	21–19	18–16	15–12	11	10	9–6	5	4–0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 1 0	0 0 1 1	0	1	Zn	0	Zd
							T				

Encoding

```
FCVTNB  <Zd>.B, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let res1 : bits(8) = FPConvertFP8{8, 32}(element1, FPCR(), FPMR());
    let res2 : bits(8) = FPConvertFP8{8, 32}(element2, FPCR(), FPMR());
    result[(2*e + 0)*:16] = ZeroExtend{16}(res1);
    result[(2*e + 1)*:16] = ZeroExtend{16}(res2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.176 FCVTNT (unpredicated)

Single-precision convert to interleaved 8-bit floating-point (top)

This instruction converts each single-precision element of the pair of source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE)}}$, and places the two-way interleaved results in the corresponding odd-numbered 8-bit elements of the destination vector, leaving the even-numbered elements unchanged. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 || FEAT_SME2) && FEAT_FP8)

31 29	28 25	24	23 22	21 19	18 16	15 12	11	10	9 6	5	4 0
0 1 1	0 0 1 0	1	0 0	0 0 1	0 1 0	0 0 1 1	1	1	Zn	0	Zd
							T				

Encoding

```
FCVTNT  <Zd>.B, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_FP8)) then EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL) = Z{}(d);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    result[(4*e + 1)*:8] = FPConvertFP8{8, 32}(element1, FPCR(), FPMR());
    result[(4*e + 3)*:8] = FPConvertFP8{8, 32}(element2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.177 FCVTNT (predicated)

Floating-point narrowing convert (top, predicated)

This instruction converts active floating-point elements from the source vector to the next lower precision, and places the results in the odd-numbered half-width elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 4 classes: Single-precision to half-precision, merging, Single-precision to half-precision, zeroing, Double-precision to single-precision, merging, and Double-precision to single-precision, zeroing

Single-precision to half-precision, merging

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 0 1 0	0 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTNT  <Zd>.H, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Single-precision to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	0 0 0 0	0 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTNT  <Zd>.H, <Pg>/Z, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision to single-precision, merging

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 0 1 0	1 0	1 0 1	Pg	Zn	Zd
			opc		opc2				

Encoding

```
FCVTNT  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Double-precision to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 18	17 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 0 0 0	1 0	1 0 1	Pg	Zn	Zd
			opc		opc2				

Encoding

```
FCVTNT  <Zd>.S, <Pg>/Z, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let halfesize : integer{} = esize DIV 2;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[(2*e + 1)*:halfesize] = FPConvertSVE{halfesize, esize}(element, FPCR());

    elsif !merging then
        result[(2*e + 1)*:halfesize] = Zeros{halfesize};

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.178 FCVTX

Double-precision convert to single-precision, rounding to odd (predicated)

This instruction converts active double-precision elements from the source vector to single-precision, rounding to Odd, and places the results in the even-numbered 32-bit elements of the destination vector, while setting the odd-numbered elements to zero. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

It has encodings from 2 classes: Double-precision to single-precision, merging and Double-precision to single-precision, zeroing

Double-precision to single-precision, merging

(FEAT_SVE2 || FEAT_SME)

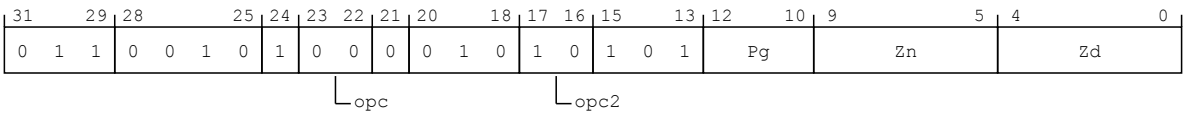

Encoding

```
FCVTX  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let merging : boolean = TRUE;
```

Double-precision to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

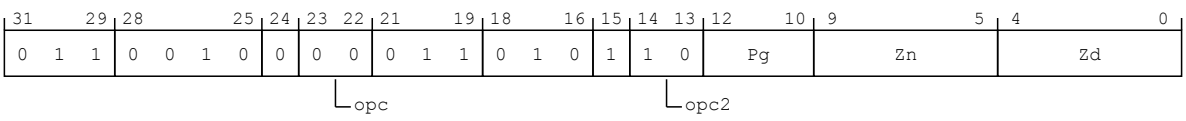

Encoding

```
FCVTX  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        let res : bits(d_esize) = FPConvertSVE{d_esize, s_esize}(element[s_esize-1:0], FPCR(),
                                                                 FPRounding_ODD);
        result[e*:esize] = ZeroExtend{esize}(res);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Double-precision to single-precision, merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.179 FCVTXNT

Double-precision convert to single-precision, rounding to odd (top, predicated)

This instruction converts active double-precision elements from the source vector to single-precision, rounding to Odd, and places the results in the odd-numbered 32-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

It has encodings from 2 classes: Double-precision to single-precision, merging and Double-precision to single-precision, zeroing

Double-precision to single-precision, merging

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 0 (opc)	0 0 1 0	1 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTXNT  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Double-precision to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 0 (opc)	0 0 0 0	1 0 (opc2)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTXNT  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let halfesize : integer{} = esize DIV 2;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[(2*e + 1)*:halfesize] = FPConvertSVE{halfesize, esize}(element, FPCR(),
                                                                      FPRounding_ODD);

    elsif !merging then
        result[(2*e + 1)*:halfesize] = Zeros{halfesize};

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.180 FCVTZS

Floating-point convert to signed integer, rounding toward zero (predicated)

This instruction converts to the signed integer nearer to zero from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.

It has encodings from 14 classes: Half-precision to 16-bit, merging, Half-precision to 16-bit, zeroing, Half-precision to 32-bit, merging, Half-precision to 32-bit, zeroing, Half-precision to 64-bit, merging, Half-precision to 64-bit, zeroing, Single-precision to 32-bit, merging, Single-precision to 32-bit, zeroing, Single-precision to 64-bit, merging, Single-precision to 64-bit, zeroing, Double-precision to 32-bit, merging, Double-precision to 32-bit, zeroing, Double-precision to 64-bit, merging, and Double-precision to 64-bit, zeroing

Half-precision to 16-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 1	0 1	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZS  <Zd>.H, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Half-precision to 16-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 1	0	1	1	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZS  <Zd>.H, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Half-precision to 32-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 1	1 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZS  <Zd>.S, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Half-precision to 32-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 1	1	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZS  <Zd>.S, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Half-precision to 64-bit, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 1	0	1 1	1 1	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZS  <Zd>.D, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Half-precision to 64-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 1	1	1	1	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZS  <Zd>.D, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Single-precision to 32-bit, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 0	0	1 1	1 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZS  <Zd>.S, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Single-precision to 32-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	0 1 1	1 1	1	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZS  <Zd>.S, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Single-precision to 64-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 1	0	1 1	1 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZS  <Zd>.D, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Single-precision to 64-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 1	1	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZS  <Zd>.D, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Double-precision to 32-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 1	0	1 1	0 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZS  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Double-precision to 32-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 1	0	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZS  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Double-precision to 64-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 1 (opc)	0	1 1	1 1 (opc2)	0 (int_U)	1 0 1	Pg	Zn	Zd

Encoding

```
FCVTZS  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Double-precision to 64-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 1 (opc)	0 1 1	1 1	1 (o2)	1	1 (o3)	0 (int_U)	Pg	Zn	Zd

Encoding

```
FCVTZS  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        let res : bits(d_esize) = FPToFixed{d_esize, s_esize}(element[s_esize-1:0], 0, unsigned,
                                                                FPCR(), rounding);
        result[e*:esize] = Extend{esize}(res, unsigned);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Half-precision to 16-bit, merging, Half-precision to 32-bit, merging, Half-precision to 64-bit, merging, Single-precision to 32-bit, merging, Single-precision to 64-bit, merging, Double-precision to 32-bit, merging, Double-precision to 64-bit, merging variants:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.181 FCVTZU

Floating-point convert to unsigned integer, rounding toward zero (predicated)

This instruction converts to the unsigned integer nearer to zero from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 14 classes: Half-precision to 16-bit, merging, Half-precision to 16-bit, zeroing, Half-precision to 32-bit, merging, Half-precision to 32-bit, zeroing, Half-precision to 64-bit, merging, Half-precision to 64-bit, zeroing, Single-precision to 32-bit, merging, Single-precision to 32-bit, zeroing, Single-precision to 64-bit, merging, Single-precision to 64-bit, zeroing, Double-precision to 32-bit, merging, Double-precision to 32-bit, zeroing, Double-precision to 64-bit, merging, and Double-precision to 64-bit, zeroing

Half-precision to 16-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 1	0 1	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.H, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Half-precision to 16-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 1	0	1	1	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.H, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Half-precision to 32-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 1	1 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.S, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Half-precision to 32-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 1	1	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.S, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Half-precision to 64-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 1	1 1	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.D, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Half-precision to 64-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 1	1	1	1	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.D, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Single-precision to 32-bit, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 0	0	1 1	1 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.S, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Single-precision to 32-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	0 1 1	1 1	1	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.S, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Single-precision to 64-bit, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 1	1 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.D, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Single-precision to 64-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 1	1	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.D, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Double-precision to 32-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 1	0	1 1	0 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Double-precision to 32-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 1	0	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Double-precision to 64-bit, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	1 1	0	1 1	1 1	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
FCVTZU  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Double-precision to 64-bit, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 1	1	1	1	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
FCVTZU  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        let res : bits(d_esize) = FPToFixed{d_esize, s_esize}(element[s_esize-1:0], 0, unsigned,
                                                                FPCR(), rounding);
        result[e*:esize] = Extend{esize}(res, unsigned);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Half-precision to 16-bit, merging, Half-precision to 32-bit, merging, Half-precision to 64-bit, merging, Single-precision to 32-bit, merging, Single-precision to 64-bit, merging, Double-precision to 32-bit, merging, Double-precision to 64-bit, merging variants:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.182 FDIV

Floating-point divide (predicated)

This instruction divides active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector and destructively places the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0	1 1 0 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FDIV  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPDiv{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.183 FDIVR

Floating-point reversed divide (predicated)

This instruction performs a reversed divide on active floating-point elements of the second source vector by the corresponding floating-point elements of the first source vector, and destructively places the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	size	0	0	1 1 0 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FDIVR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPDiv{esize}(element2, element1, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.184 FDOT (4-way, vectors)

8-bit floating-point dot product to single-precision

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in each 32-bit element of the first source and second source vectors. The single-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the destination vector.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8DOT4) || FEAT_SSVE_FP8DOT4)

31 29	28 25	24	23	22	21	20 16	15 14	13	12 11	10	9 5	4 0
0 1 1	0 0 1 0	0	0	1	1	Zm	1 0	0	0 0	1	Zn	Zda
				op						o2		

Encoding

```
FDOT  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8DOT4() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8DOT4) && IsFeatureImplemented(FEAT_FP8DOT4) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8DOT4) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1 : bits(32) = operand1[e*:(32)];
    let op2 : bits(32) = operand2[e*:(32)];
    var sum : bits(32)  = operand3[e*:(32)];

    sum = FP8DotAddFP{32, 32}(sum, op1, op2, FPCR(), FPMR());
    result[e*:(32)] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.185 FDOT (4-way, indexed)

8-bit floating-point dot product by indexed element to single-precision

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in each 32-bit element of the first source vector and a group of four 8-bit floating-point values in an indexed 32-bit element of the second source vector. The single-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the destination vector.

The 8-bit floating-point groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8DOT4) || FEAT_SSVE_FP8DOT4)

31 29	28 25	24	23	22	21	20 19	18 16	15 14	13	12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	1	1	i2	Zm	0 1	0	0	0 1	Zn	Zda
				op							opc2		

Encoding

```
FDOT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8DOT4() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i2);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8DOT4) && IsFeatureImplemented(FEAT_FP8DOT4) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8DOT4) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let op1 : bits(32) = operand1[e*:(32)];
    let op2 : bits(32) = operand2[s*:(32)];
    var sum : bits(32)  = operand3[e*:(32)];

    sum = FP8DotAddFP{32, 32}(sum, op1, op2, FPCR(), FPMR());
    result[e*:(32)] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.186 FDOT (2-way, vectors, FP8 to FP16)

8-bit floating-point dot product to half-precision

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in each 16-bit element of the first source and second source vectors. The half-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the destination vector.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8DOT2) || FEAT_SSVE_FP8DOT2)

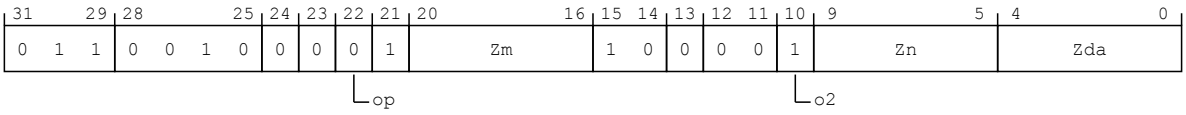

Encoding

```
FDOT  <Zda>.H, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8DOT2() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8DOT2) && IsFeatureImplemented(FEAT_FP8DOT2) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8DOT2) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1 : bits(16) = operand1[e*:(16)];
    let op2 : bits(16) = operand2[e*:(16)];
    var sum : bits(16)  = operand3[e*:(16)];

    sum = FP8DotAddFP{16, 16}(sum, op1, op2, FPCR(), FPMR());
    result[e*:(16)] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.187 FDOT (2-way, indexed, FP8 to FP16)

8-bit floating-point dot product by indexed element to half-precision

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in each 16-bit element of the first source vector and a pair of 8-bit floating-point values in an indexed 16-bit element of the second source vector. The half-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the destination vector.

The 8-bit floating-point groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8DOT2) || FEAT_SSVE_FP8DOT2)

31 29	28 25	24	23	22	21	20 19	18 16	15 14	13	12	11	10	9 5	4 0
0 1 1	0 0 1 0	0	0	0	1	i3h	Zm	0 1	0	0	i3l	1	Zn	Zda
				op										

Encoding

```
FDOT  <Zda>.H, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8DOT2() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a pair of 8-bit elements within each 128-bit vector segment, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8DOT2) && IsFeatureImplemented(FEAT_FP8DOT2) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8DOT2) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let eltspersegment : integer = 128 DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let op1 : bits(16) = operand1[e*:(16)];
    let op2 : bits(16) = operand2[s*:(16)];
    var sum : bits(16)  = operand3[e*:(16)];

    sum = FP8DotAddFP{16, 16}(sum, op1, op2, FPCR(), FPMR());
    result[e*:(16)] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.188 FDOT (2-way, vectors, FP16 to FP32)

Half-precision dot product to single-precision

This instruction calculates the fused sum-of-products of a pair of half-precision values held in each 32-bit element of the first source and second source vectors, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31	29	28	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0 1 1		0 0 1 0		0	0	0	1	Zm		1	0	0	0	0	0	Zn		Zda	
						op									o2				

Encoding

```
FDOT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
    var sum : bits(32) = operand3[e*:32];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
    result[e*:32] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.189 FDOT (2-way, indexed, FP16 to FP32)

Half-precision dot product by indexed element to single-precision

This instruction calculates the fused sum-of-products of a pair of half-precision values held in each 32-bit element of the first source vector and a pair of half-precision values in an indexed 32-bit element of the second source vector, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

The half-precision pairs within the second source vector are specified using an immediate index that selects the same pair position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31–29	28–25	24	23	22	21	20–19	18–16	15–14	13	12	11–10	9–5	4–0
0 1 1	0 0 1 0	0	0	0	1	i2	Zm	0 1	0	0	0 0	Zn	Zda
				op							opc2		

Encoding

```
FDOT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i2);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * s + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * s + 1)*:16];
    var sum : bits(32) = operand3[e*:32];

    sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
    result[e*:32] = sum;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.190 FDUP

Broadcast floating-point immediate to vector elements (unpredicated)

This instruction unconditionally broadcasts the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias FMOV (immediate, unpredicated).

SVE

(FEAT_SVE || FEAT_SME)

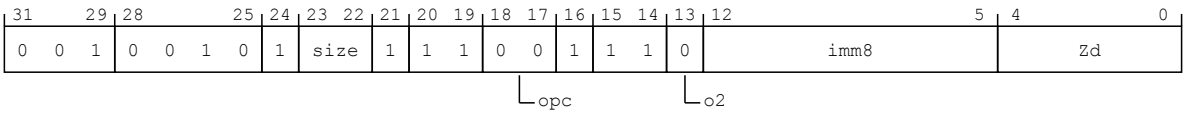

Encoding

```
FDUP  <Zd>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd);
let imm : bits(esize) = VFPExpandImm{}(imm8);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<const>

Is a floating-point immediate value expressible as ±n÷16×2^r, where n and r are integers such that $16 \leq n \leq 31$ and $-3 \leq r \leq 4$, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
FMOV (immediate, unpredicated)	Unconditionally

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var result : bits(VL);

for e = 0 to elements-1 do
    result[e*:esize] = imm;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.191 FEXPA

Floating-point exponential accelerator

This instruction calculates an exponentiation acceleration operation on each floating-point element in the source vector, where the result sign is zero, the result exponent field is copied from a set of significant bits of the input fraction, and the result fraction is inserted from a lookup table indexed by the least-significant input fraction bits, and returns each result in the corresponding element of the destination vector.

This instruction is fully defined by its bit-manipulation semantics, does not generate floating-point exceptions, and does not guarantee NaN propagation.

For double-precision variants, the result element exponent is copied from the source element bits<16:6>, and the result fraction is set based on the source element to the rounded value of $2^{52} \times (2^{bits<5:0>/64} - 1)$.

For single-precision variants, the result element exponent is copied from the source element bits<13:6>, and the result fraction is set based on the source element to the rounded value of $2^{23} \times (2^{bits<5:0>/64} - 1)$.

For half-precision variants, the result element exponent is copied from the source element bits<9:5>, and the result fraction is set based on the source element to the rounded value of $2^{10} \times (2^{bits<4:0>/32} - 1)$.

This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SSVE_FEXPA is implemented.

Note

For a double-precision input value x in the range 70,368,744,177,655 <= x < 70,368,744,179,711, the operation performed by this instruction is equivalent to calculating $2^{x-70,368,744,178,687}$.
For a single-precision input value x in the range 131,073 <= x < 131,327, the operation performed by this instruction is equivalent to calculating $2^{x-131,199}$.
For a half-precision input value x in the range 33 <= x < 63, the operation performed by this instruction is equivalent to calculating 2^{x-47}.

SVE

(FEAT_SVE || FEAT_SSVE_FEXPA)

31–29	28–25	24	23–22	21	20–16	15–12	11–10	9–5	4–0
0 0 0	0 0 1 0	0	size	1	0 0 0 0 0 (opc)	1 0 1 1	1 0	Zn	Zd

Encoding

```
FEXPA  <Zd>.<T>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SSVE_FEXPA) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_FEXPA) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPExpA{esize}(element);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.192 FIRSTP

Scalar index of first true predicate element (predicated)

This instruction determines the index of the first active and true element of the source predicate, and places the scalar result in the destination general-purpose register. If there are no active and true elements, the destination general-purpose register is updated to the value minus one.

SVE2

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–16	15–14	13–10	9	8–5	4–0
0 0 1	0 0 1 0	1	size	1 0 0	0 0 1 (opc)	1 0	Pg	0	Pn	Rd

Encoding

```
FIRSTP  <Xd>, <Pg>, <Pn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Rd);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
var index : integer = -1;

for e = 0 to elements-1 do
    if (index == -1 && ActivePredicateElement{PL}(mask, e, esize) &&
          ActivePredicateElement{PL}(operand, e, esize)) then
        index = e;
    end;
end;

X{64}(d) = index[63:0];
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.193 FLOGB

Floating-point base 2 logarithm as integer

This instruction returns the signed integer base 2 logarithm of each floating-point input element |x| after normalization.

This is the unbiased exponent of x used in the representation of the floating-point value, such that, for positive x, x = significand $\times$ $2^{exponent}$.

The integer results are placed in elements of the destination vector that have the same width (esize) as the floating-point input elements:

- If x is normal, the result is the base 2 logarithm of x.
- If x is subnormal, the result corresponds to the normalized representation.
- If x is infinite, the result is $2^{(esize-1)}$-1.
- If x is ±0.0 or NaN, the result is -$2^{(esize-1)}$.

Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (opc)	0	1 1	size	0 (U)	1 0 1	Pg	Zn	Zd

Encoding

```
FLOGB  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 0 (opc)	0 1 1	1 1	0 (o2)	1	size	Pg	Zn	Zd

Encoding

```
FLOGB  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPLogB{esize}(element, FPCR());
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.194 FMAD

Floating-point fused multiply-add to multiplicand (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and adds the result to elements of the third (addend) vector without intermediate rounding. The results are destructively placed in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

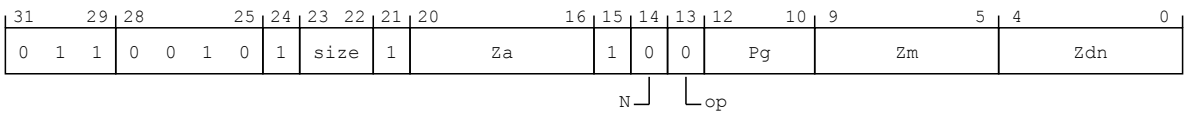

Encoding

```
FMAD  <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let a : integer = UInt(Za);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = Z{}(dn);
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(a) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.195 FMAX (immediate)

Floating-point maximum with immediate (predicated)

This instruction determines the maximum of an immediate and each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if the element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both the element and the immediate are zeros, regardless of the sign of either zero, the result is the immediate.
- If the element is a NaN, regardless of the value of FPCR.DN, the result is the immediate.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

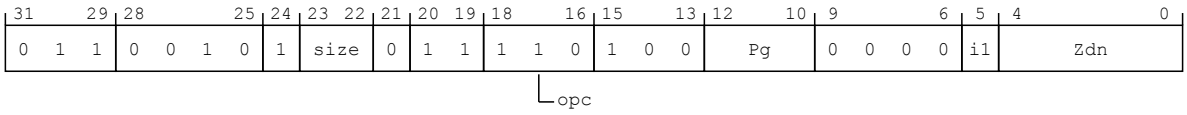

Encoding

```
FMAX  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then Zeros{esize} else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPMax{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.196 FMAX (vectors)

Floating-point maximum (predicated)

This instruction determines the maximum of active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	0 1 1 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMAX  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPMax{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.197 FMAXNM (immediate)

Floating-point maximum number with immediate (predicated)

This instruction determines the maximum number value of an immediate and each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If the element is a quiet NaN, the result is the immediate value.
- When FPCR.DN is 0, if the element is a signaling NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a signaling NaN, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–6	5	4–0
0 1 1	0 0 1 0	1	size	0	1 1	1 0 0	1 0 0	Pg	0 0 0 0	i1	Zdn
						opc					

Encoding

```
FMAXNM  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then Zeros{esize} else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPMaxNum{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.198 FMAXNM (vectors)

Floating-point maximum number (predicated)

This instruction determines the maximum number value of active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	0 1 0 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMAXNM  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPMaxNum{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.199 FMAXNMP

Floating-point maximum number pairwise

This instruction calculates the maximum value of each pair of adjacent floating-point elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 0 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMAXNMP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL) = Z{}(dn);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if IsEven(e) then
            element1 = operand1[(e + 0)*:esize];
            element2 = operand1[(e + 1)*:esize];
        else
            element1 = operand2[(e - 1)*:esize];
            element2 = operand2[(e + 0)*:esize];
        end;
        result[e*:esize] = FPMaxNum{esize}(element1, element2, FPCR());
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.200 FMAXNMQV

Floating-point maximum number recursive reduction of quadword vector segments

This instruction calculates a floating-point maximum number of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 0 0 (opc)	1 0 1	Pg	Zn	Vd

Encoding

```
FMAXNMQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	RESERVED
01	8H
10	4S
11	2D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer{} = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let segbits : integer{} = segments*esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPDefaultNaN{esize}(FPCR());
var result : bits(128) = Zeros{};

for e = 0 to elempersegment-1 do
    var stmp : bits(segbits);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp[s*:esize] = operand[(s * elempersegment + e)*:esize];
        else
            stmp[s*:esize] = identity;
        end;
    end;
    result[e*:esize] = FPReduce{esize, segbits}(ReduceOp_FMAXNUM, stmp, FPCR());
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.201 FMAXNMV

Floating-point maximum number recursive reduction to scalar

This instruction calculates a floating-point maximum number horizontally over all lanes of a vector using a recursive pairwise reduction, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 0	1 0 0 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
FMAXNMV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPDefaultNaN{esize}(FPCR());

V{esize}(d) = FPReducePredicated{esize, VL, PL}(ReduceOp_FMAXNUM, operand, mask,
                                                                identity, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.202 FMAXP

Floating-point maximum pairwise

This instruction calculates the maximum value of each pair of adjacent floating-point elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 1 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMAXP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL) = Z{}(dn);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if IsEven(e) then
            element1 = operand1[(e + 0)*:esize];
            element2 = operand1[(e + 1)*:esize];
        else
            element1 = operand2[(e - 1)*:esize];
            element2 = operand2[(e + 0)*:esize];
        end;
        result[e*:esize] = FPMax{esize}(element1, element2, FPCR());
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.203 FMAXQV

Floating-point maximum reduction of quadword vector segments

This instruction calculates a floating-point maximum of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as -Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 1 0 (opc)	1 0 1	Pg	Zn	Vd

Encoding

```
FMAXQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	RESERVED
01	8H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer{} = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let segbits : integer{} = segments*esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPInfinity{esize}('1');
var result : bits(128) = Zeros{};

for e = 0 to elempersegment-1 do
    var stmp : bits(segbits);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp[s*:esize] = operand[(s * elempersegment + e)*:esize];
        else
            stmp[s*:esize] = identity;
        end;
    end;
    result[e*:esize] = FPReduce{esize, segbits}(ReduceOp_FMAX, stmp, FPCR());
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.204 FMAXV

Floating-point maximum recursive reduction to scalar

This instruction calculates a floating-point maximum horizontally over all lanes of a vector using a recursive pairwise reduction, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as -Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 0	1 1 0 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
FMAXV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPInfinity{esize}('1');

V{esize}(d) = FPReducePredicated{esize, VL, PL}(ReduceOp_FMAX, operand, mask,
                                                                identity, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.205 FMIN (immediate)

Floating-point minimum with immediate (predicated)

This instruction determines the minimum of an immediate and each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if the element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both the element and the immediate are zeros, regardless of the sign of either zero, the result is the immediate.
- If the element is a NaN, regardless of the value of FPCR.DN, the result is the immediate.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 6	5	4 0
0 1 1	0 0 1 0	1	size	0	1 1	1 1 1	1 0 0	Pg	0 0 0 0	i1	Zdn
						opc					

Encoding

```
FMIN  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then Zeros{esize} else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPMin{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.206 FMIN (vectors)

Floating-point minimum (predicated)

This instruction determines the minimum of active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	0 1 1 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMIN  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPMin{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.207 FMINNM (immediate)

Floating-point minimum number with immediate (predicated)

This instruction determines the minimum number value of an immediate and each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If the element is a quiet NaN, the result is the immediate value.
- When FPCR.DN is 0, if the element is a signaling NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a signaling NaN, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–6	5	4–0
0 1 1	0 0 1 0	1	size	0	1 1	1 0 1 (opc)	1 0 0	Pg	0 0 0 0	i1	Zdn

Encoding

```
FMINNM  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then Zeros{esize} else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPMinNum{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.208 FMINNM (vectors)

Floating-point minimum number (predicated)

This instruction determines the minimum number value of active floating-point elements of the first source vector and the corresponding floating-point elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	0 1 0 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMINNM  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPMinNum{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.209 FMINNMP

Floating-point minimum number pairwise

This instruction calculates the minimum value of each pair of adjacent floating-point elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

SVE2

(FEAT_SVE2 || FEAT_SME)

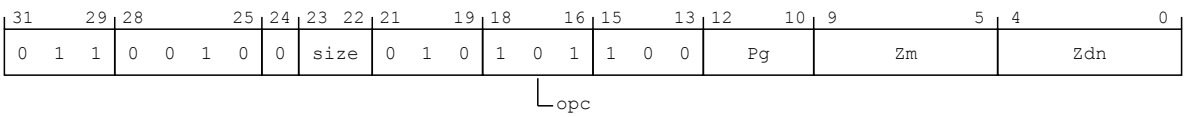

Encoding

```
FMINNMP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL) = Z{}(dn);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if IsEven(e) then
            element1 = operand1[(e + 0)*:esize];
            element2 = operand1[(e + 1)*:esize];
        else
            element1 = operand2[(e - 1)*:esize];
            element2 = operand2[(e + 0)*:esize];
        end;
        result[e*:esize] = FPMinNum{esize}(element1, element2, FPCR());
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.210 FMINNMQV

Floating-point minimum number recursive reduction of quadword vector segments

This instruction calculates a floating-point minimum number of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 0 1 (opc)	1 0 1	Pg	Zn	Vd

Encoding

```
FMINNMQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	RESERVED
01	8H
10	4S
11	2D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer{} = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let segbits : integer{} = segments*esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPDefaultNaN{esize}(FPCR());
var result : bits(128) = Zeros{};

for e = 0 to elempersegment-1 do
    var stmp : bits(segbits);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp[s*:esize] = operand[(s * elempersegment + e)*:esize];
        else
            stmp[s*:esize] = identity;
        end;
    end;
    result[e*:esize] = FPReduce{esize, segbits}(ReduceOp_FMINNUM, stmp, FPCR());
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.211 FMINNMV

Floating-point minimum number recursive reduction to scalar

This instruction calculates a floating-point minimum number horizontally over all lanes of a vector using a recursive pairwise reduction, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 0	1 0 1 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
FMINNMV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPDefaultNaN{esize}(FPCR());

V{esize}(d) = FPReducePredicated{esize, VL, PL}(ReduceOp_FMINNUM, operand, mask,
                                                                identity, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.212 FMINP

Floating-point minimum pairwise

This instruction calculates the minimum value of each pair of adjacent floating-point elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 1 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMINP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL) = Z{}(dn);
var element1 : bits(esize);
var element2 : bits(esize);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if IsEven(e) then
            element1 = operand1[(e + 0)*:esize];
            element2 = operand1[(e + 1)*:esize];
        else
            element1 = operand2[(e - 1)*:esize];
            element2 = operand2[(e + 0)*:esize];
        end;
        result[e*:esize] = FPMin{esize}(element1, element2, FPCR());
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.213 FMINQV

Floating-point minimum recursive reduction of quadword vector segments

This instruction calculates a floating-point minimum of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as +Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 0	1 1 1 (opc)	1 0 1	Pg	Zn	Vd

Encoding

```
FMINQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	RESERVED
01	8H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer{} = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let segbits : integer{} = segments*esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPInfinity{esize}('0');
var result : bits(128) = Zeros{};

for e = 0 to elempersegment-1 do
    var stmp : bits(segbits);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp[s*:esize] = operand[(s * elempersegment + e)*:esize];
        else
            stmp[s*:esize] = identity;
        end;
    end;
    result[e*:esize] = FPReduce{esize, segbits}(ReduceOp_FMIN, stmp, FPCR());
end;
V{128}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.214 FMINV

Floating-point minimum recursive reduction to scalar

This instruction calculates a floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 0	1 1 1 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
FMINV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let identity : bits(esize) = FPInfinity{esize}('0');

V{esize}(d) = FPReducePredicated{esize, VL, PL}(ReduceOp_FMIN, operand, mask,
                                                                identity, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.215 FMLA (vectors)

Floating-point fused multiply-add (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and adds the result to elements of the third source (addend) vector without intermediate rounding. The results are destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

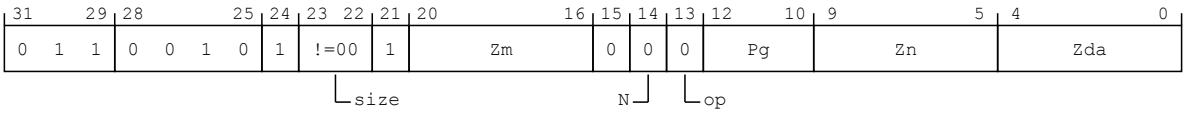

Encoding

```
FMLA  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op3[e*:esize];

    end;
end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.216 FMLA (indexed)

Floating-point fused multiply-add by indexed element

This instruction multiplies all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision, and Double-precision

Half-precision

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 0 0	0	0	Zn	Zda
									o2	op		

Encoding

```
FMLA  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Single-precision

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 0	0	0	Zn	Zda
			size					o2	op		

Encoding

```
FMLA  <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Double-precision

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 0	0	0	Zn	Zda
			size					o2	op		

Encoding

```
FMLA  <Zda>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “Half-precision” and “Single-precision” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “Double-precision” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “Half-precision” variant: is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “Single-precision” variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.

For the “Double-precision” variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elem2 : bits(esize) = op2[s*:esize];
    let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                               else op1[e*:esize]);
    let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(result[e*:esize], FPCR())
                               else result[e*:esize]);
    result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.217 FMLALB (vectors, FP8 to FP16)

8-bit floating-point multiply-add to half-precision (bottom)

This instruction widens the even 8-bit elements in the first and second source vectors to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the half-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 16	15 14	13	12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1	0 1	Zm	1 0	0	0	1 0	Zn	Zda
								T			

Encoding

```
FMLALB  <Zda>.H, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(2 * e + 0)*:8];
    let element2 : bits(8) = operand2[(2 * e + 0)*:8];
    let element3 : bits(16) = operand3[e*:16];
    result[e*:16] = FP8MulAddFP{16}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.218 FMLALB (indexed, FP8 to FP16)

8-bit floating-point multiply-add by indexed element to half-precision (bottom)

This instruction widens the even 8-bit elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the half-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	0 1	i4h	Zm	0 1 0 1	i4l	Zn	Zda
			T							

Encoding

```
FMLALB  <Zda>.H, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i4h::i4l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let elements : integer = VL DIV 16;
let eltspersegment : integer = 128 DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let element1 : bits(8) = operand1[(2 * e + 0)*:8];
    let element2 : bits(8) = operand2[s*:8];
    let element3 : bits(16) = operand3[e*:16];
    result[e*:16] = FP8MulAddFP{16}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.219 FMLALB (vectors, FP16 to FP32)

Half-precision multiply-add to single-precision (bottom)

This instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

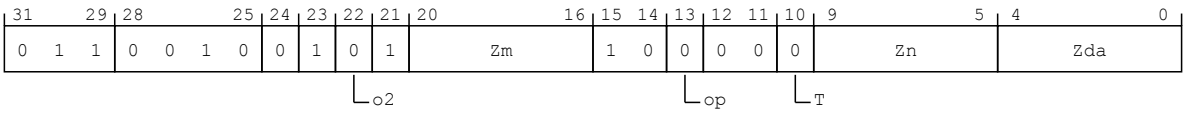

Encoding

```
FMLALB  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 0)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 0)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[(2*e + 0)*:(esize DIV 2)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.220 FMLALB (indexed, FP16 to FP32)

Half-precision multiply-add by indexed element to single-precision (bottom)

This instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Single-precision

(FEAT_SVE2 || FEAT_SME)

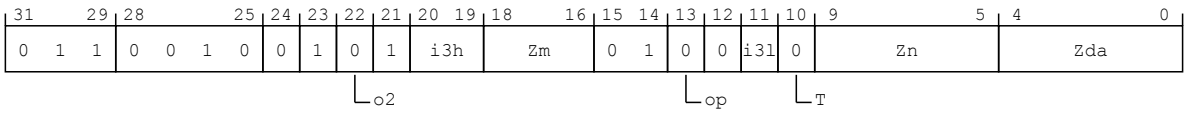

Encoding

```
FMLALB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 0)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 0)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[s*:(esize DIV 2)];
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.221 FMLALLBB (vectors)

8-bit floating-point multiply-add to single-precision (bottom bottom)

This instruction widens the first 8-bit element of each 32-bit container in the first and second source vectors to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	0 1	Zm	1 0	0 0	1 0	Zn	Zda
							TT			

Encoding

```
FMLALLBB  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(4 * e + 0)*:8];
    let element2 : bits(8) = operand2[(4 * e + 0)*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.222 FMLALLBB (indexed)

8-bit floating-point multiply-add by indexed element to single-precision (bottom bottom)

This instruction widens the first 8-bit element of each 32-bit container in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23 22	21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0 0	1	i4h	Zm	1 1 0 0	i4l	Zn	Zda
			TT							

Encoding

```
FMLALLBB  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i4h::i4l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 4 * segmentbase + index;
    let element1 : bits(8) = operand1[(4 * e + 0)*:8];
    let element2 : bits(8) = operand2[s*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.223 FMLALLBT (vectors)

8-bit floating-point multiply-add to single-precision (bottom top)

This instruction widens the second 8-bit element of each 32-bit container in the first and second source vectors to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	0 1	Zm	1 0	0 1	1 0	Zn	Zda
							TT			

Encoding

```
FMLALLBT  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(4 * e + 1)*:8];
    let element2 : bits(8) = operand2[(4 * e + 1)*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.224 FMLALLBT (indexed)

8-bit floating-point multiply-add by indexed element to single-precision (bottom top)

This instruction widens the second 8-bit element of each 32-bit container in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23 22	21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0 1	1	i4h	Zm	1 1 0 0	i4l	Zn	Zda
			TT							

Encoding

```
FMLALLBT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i4h::i4l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 4 * segmentbase + index;
    let element1 : bits(8) = operand1[(4 * e + 1)*:8];
    let element2 : bits(8) = operand2[s*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.225 FMLALLTB (vectors)

8-bit floating-point multiply-add to single-precision (top bottom)

This instruction widens the third 8-bit element of each 32-bit container in the first and second source vectors to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	0 1	Zm	1 0	1 0	1 0	Zn	Zda
							TT			

Encoding

```
FMLALLTB  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(4 * e + 2)*:8];
    let element2 : bits(8) = operand2[(4 * e + 2)*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.226 FMLALLTB (indexed)

8-bit floating-point multiply-add by indexed element to single-precision (top bottom)

This instruction widens the third 8-bit element of each 32-bit container in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE)}}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23 22	21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	1	i4h	Zm	1 1 0 0	i4l	Zn	Zda
			TT							

Encoding

```
FMLALLTB  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i4h::i4l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 4 * segmentbase + index;
    let element1 : bits(8) = operand1[(4 * e + 2)*:8];
    let element2 : bits(8) = operand2[s*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.227 FMLALLTT (vectors)

8-bit floating-point multiply-add to single-precision (top top)

This instruction widens the fourth 8-bit element of each 32-bit container in the first and second source vectors to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 16	15 14	13 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	0	0 1	Zm	1 0	1 1	1 0	Zn	Zda
							TT			

Encoding

```
FMLALLTT  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(4 * e + 3)*:8];
    let element2 : bits(8) = operand2[(4 * e + 3)*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.228 FMLALLTT (indexed)

8-bit floating-point multiply-add by indexed element to single-precision (top top)

This instruction widens the fourth 8-bit element of each 32-bit container in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$ before being destructively added without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23 22	21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	1	i4h	Zm	1 1 0 0	i4l	Zn	Zda
			TT							

Encoding

```
FMLALLTT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i4h::i4l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let elements : integer = VL DIV 32;
let eltspersegment : integer = 128 DIV 32;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 4 * segmentbase + index;
    let element1 : bits(8) = operand1[(4 * e + 3)*:8];
    let element2 : bits(8) = operand2[s*:8];
    let element3 : bits(32) = operand3[e*:32];
    result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.229 FMLALT (vectors, FP8 to FP16)

8-bit floating-point multiply-add to half-precision (top)

This instruction widens the odd 8-bit elements in the first and second source vectors to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the half-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the source vectors. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 16	15 14	13	12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1	0 1	Zm	1 0	0	1	1 0	Zn	Zda
								T			

Encoding

```
FMLALT  <Zda>.H, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bits(8) = operand1[(2 * e + 1)*:8];
    let element2 : bits(8) = operand2[(2 * e + 1)*:8];
    let element3 : bits(16) = operand3[e*:16];
    result[e*:16] = FP8MulAddFP{16}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.230 FMLALT (indexed, FP8 to FP16)

8-bit floating-point multiply-add by indexed element to half-precision (top)

This instruction widens the odd 8-bit elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the half-precision elements of the destination vector that overlap with the corresponding 8-bit floating-point elements in the first source vector. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SVE2

((FEAT_SVE2 && FEAT_FP8FMA) || FEAT_SSVE_FP8FMA)

31 29	28 25	24	23	22 21	20 19	18 16	15 12	11 10	9 5	4 0
0 1 1	0 0 1 0	0	1	0 1	i4h	Zm	0 1 0 1	i4l	Zn	Zda
			T							

Encoding

```
FMLALT  <Zda>.H, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !HaveSVE2FP8FMA() then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i4h::i4l);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

Operation

```
CheckFPMREnabled();
if IsFeatureImplemented(FEAT_SSVE_FP8FMA) && IsFeatureImplemented(FEAT_FP8FMA) then
    CheckSVEEnabled();
elsif IsFeatureImplemented(FEAT_FP8FMA) then
    CheckNonStreamingSVEEnabled();
else
    CheckStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let elements : integer = VL DIV 16;
let eltspersegment : integer = 128 DIV 16;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let element1 : bits(8) = operand1[(2 * e + 1)*:8];
    let element2 : bits(8) = operand2[s*:8];
    let element3 : bits(16) = operand3[e*:16];
    result[e*:16] = FP8MulAddFP{16}(element3, element1, element2, FPCR(), FPMR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.231 FMLALT (vectors, FP16 to FP32)

Half-precision multiply-add to single-precision (top)

This instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

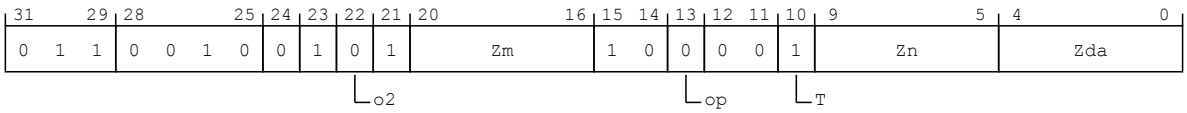

Encoding

```
FMLALT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 1)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 1)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[(2*e + 1)*:(esize DIV 2)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.232 FMLALT (indexed, FP16 to FP32)

Half-precision multiply-add by indexed element to single-precision (top)

This instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Single-precision

(FEAT_SVE2 || FEAT_SME)

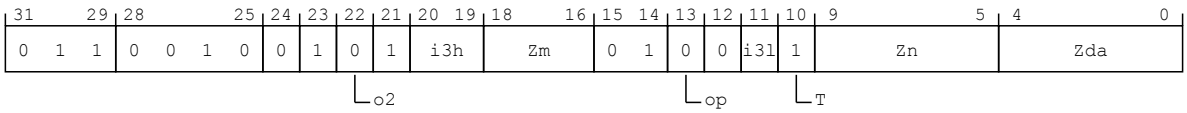

Encoding

```
FMLALT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 1)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 1)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[s*:(esize DIV 2)];
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.233 FMLS (vectors)

Floating-point fused multiply-subtract (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and subtracts the result from elements of the third source (addend) vector without intermediate rounding. The results are destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

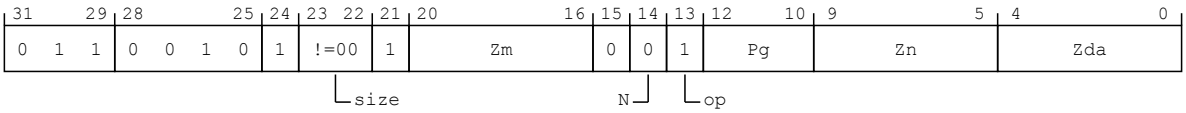

Encoding

```
FMLS  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op3[e*:esize];

    end;
end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.234 FMLS (indexed)

Floating-point fused multiply-subtract by indexed element

This instruction multiplies all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision, and Double-precision

Half-precision

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 0 0	0	1	Zn	Zda
									o2	op		

Encoding

```
FMLS  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Single-precision

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0	0	1	0	1	i2	Zm	0 0 0 0	0	1	Zn	Zda
			size						o2	op		

Encoding

```
FMLS  <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Double-precision

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 0	0	1	Zn	Zda
			size					o2	op		

Encoding

```
FMLS  <Zda>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “Half-precision” and “Single-precision” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “Double-precision” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “Half-precision” variant: is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “Single-precision” variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.

For the “Double-precision” variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elem2 : bits(esize) = op2[s*:esize];
    let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                               else op1[e*:esize]);
    let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(result[e*:esize], FPCR())
                               else result[e*:esize]);
    result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.235 FMLSLB (vectors)

Half-precision multiply-subtract from single-precision (bottom)

This instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–14	13	12–11	10	9–5	4–0
0 1 1	0 0 1 0	0	1	0	1	Zm	1 0	1	0 0	0	Zn	Zda
				o2				op		T		

Encoding

```
FMLSLB  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 0)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 0)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[(2*e + 0)*:(esize DIV 2)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.236 FMLSLB (indexed)

Half-precision multiply-subtract by indexed element from single-precision (bottom)

This instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Single-precision

(FEAT_SVE2 || FEAT_SME)

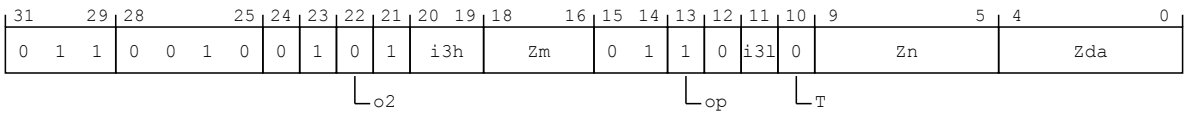

Encoding

```
FMLSLB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 0)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 0)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[s*:(esize DIV 2)];
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.237 FMLSLT (vectors)

Half-precision multiply-subtract from single-precision (top)

This instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

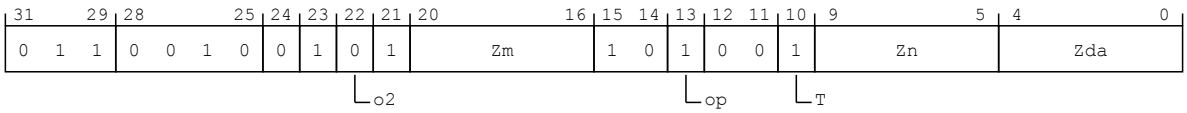

Encoding

```
FMLSLT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 1)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 1)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[(2*e + 1)*:(esize DIV 2)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.238 FMLSLT (indexed)

Half-precision multiply-subtract by indexed element from single-precision (top)

This instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Single-precision

(FEAT_SVE2 || FEAT_SME)

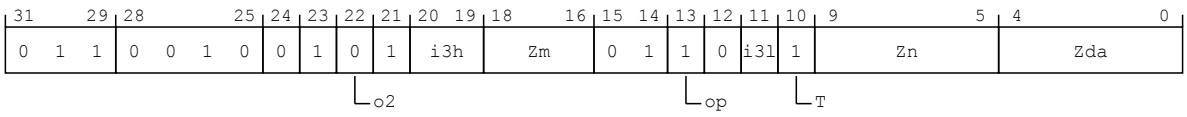

Encoding

```
FMLSLT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let index : integer = UInt(i3h::i3l);
let op1_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = 2 * segmentbase + index;
    let elem1 : bits(esize DIV 2) = (if op1_neg
                                     then FPNeg{esize DIV 2}(op1[(2*e + 1)*:(esize DIV 2)], FPCR())
                                     else op1[(2*e + 1)*:(esize DIV 2)]);
    let elem2 : bits(esize DIV 2) = op2[s*:(esize DIV 2)];
    let elem3 : bits(esize) = op3[e*:esize];
    result[e*:esize] = FPMulAddH(elem3, elem1, elem2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.239 FMMLA (widening, FP8 to FP16)

8-bit floating-point matrix multiply-accumulate to half-precision

This instruction multiplies the 2×4 matrix of 8-bit floating-point values held in each 64-bit segment of the first source vector by the 4×2 matrix of 8-bit floating-point values in the corresponding segment of the second source vector. The intermediate half-precision results calculated as a fused sum-of-products using four adjacent 8-bit floating-point values from each of the source vectors, are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the 2×2 half-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating 4-way dot product per destination element.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2 && FEAT_F8F16MM)

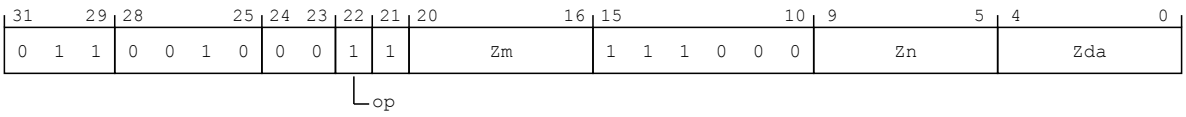

Encoding

```
FMMLA  <Zda>.H, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) || !IsFeatureImplemented(FEAT_F8F16MM) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckFPMREnabled();
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer =  VL DIV 64;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for s = 0 to segments-1 do
    let op1 : bits(64)    = operand1[s*:(64)];
    let op2 : bits(64)    = operand2[s*:(64)];
    let addend : bits(64) = operand3[s*:(64)];
    let way : integer{} = 4;
    result[s*:(64)] = FP8MatMulAddFP{64}(addend, op1, op2, way, FPCR(), FPMR());

end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.240 FMMLA (widening, FP8 to FP32)

8-bit floating-point matrix multiply-accumulate to single-precision

This instruction multiplies the 2×8 matrix of 8-bit floating-point values held in each 128-bit segment of the first source vector by the 8×2 matrix of 8-bit floating-point values in the corresponding segment of the second source vector. The intermediate single-precision results calculated as a fused sum-of-products using eight adjacent 8-bit floating-point values from each of the source vectors, are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the 2×2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating 8-way dot product per destination element.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2 && FEAT_F8F32MM)

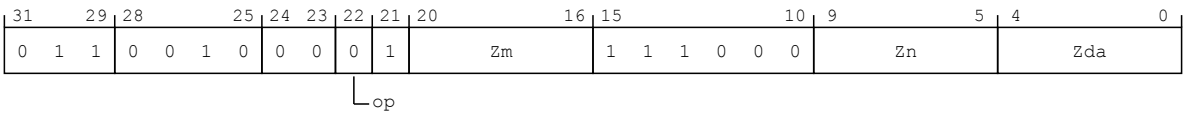

Encoding

```
FMMLA  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) || !IsFeatureImplemented(FEAT_F8F32MM) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckFPMREnabled();
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer =  VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for s = 0 to segments-1 do
    let op1 : bits(128)    = operand1[s*:(128)];
    let op2 : bits(128)    = operand2[s*:(128)];
    let addend : bits(128) = operand3[s*:(128)];
    let way : integer{} = 8;
    result[s*:(128)] = FP8MatMulAddFP{128}(addend, op1, op2, way, FPCR(), FPMR());

end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.241 FMMLA (widening, FP16 to FP32)

Half-precision matrix multiply-accumulate to single-precision

This instruction multiplies the 2×4 matrix of half-precision values held in each 128-bit segment of the first source vector by the 4×2 matrix of half-precision values in the corresponding segment of the second source vector. The intermediate single-precision results are calculated as a sum of two fused sum-of-products using two pairs of adjacent half-precision values from each of the source vectors, and rounded before accumulation into the 2×2 matrix of single-precision results in the corresponding segment of the destination vector. This is equivalent to performing a 4-way dot product per destination element.

This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_F16F32MM)

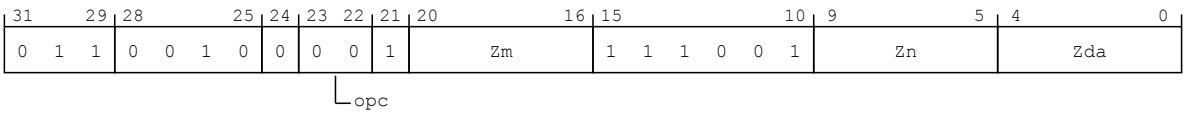

Encoding

```
FMMLA  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_F16F32MM) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer =  VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for s = 0 to segments-1 do
    let op1 : bits(128)    = operand1[s*:(128)];
    let op2 : bits(128)    = operand2[s*:(128)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let addend : bits(128) = operand3[s*:(128)];
    result[s*:(128)] = FPMatMulAddH{128}(addend, op1, op2, FPCR());
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.242 FMMLA (non-widening)

Floating-point matrix multiply-accumulate

This instruction supports single-precision and double-precision data types in a 2x2 matrix contained in segments of 128 or 256 bits, respectively. It multiplies the 2x2 matrix in each segment of the first source vector by the 2x2 matrix in the corresponding segment of the second source vector. The intermediate products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the 2x2 matrix held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the Effective SVE vector length is at least 256 bits.

ID_AA64ZFR0_EL1.F32MM indicates whether the single-precision variant is implemented.

ID_AA64ZFR0_EL1.F64MM indicates whether the double-precision variant is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_F32MM)

31–29	28–25	24	23–22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0	0	1 0 (opc)	1	Zm	1 1 1 0 0 1	Zn	Zda

Encoding

```
FMMLA  <Zda>.S, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F32MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

64-bit element

(FEAT_F64MM)

31–29	28–25	24	23–22	21	20–16	15–10	9–5	4–0
0 1 1	0 0 1 0	0	1 1 (opc)	1	Zm	1 1 1 0 0 1	Zn	Zda

Encoding

```
FMMLA  <Zda>.D, <Zn>.D, <Zm>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
if VL < esize * 4 then EndOfDecode(Decode_UNDEF); end;
let segments : integer = VL DIV (4 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL) = Zeros{};
var op1, op2 : bits(4*esize);
var res, addend : bits(4*esize);

for s = 0 to segments-1 do
    op1    = operand1[s*:(4*esize)];
    op2    = operand2[s*:(4*esize)];
    addend = operand3[s*:(4*esize)];
    res    = FPMatMulAdd{4*esize}(addend, op1, op2, esize, FPCR());
    result[s*:(4*esize)] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.243 FMOV (zero, predicated)

Move floating-point +0.0 to vector elements (predicated)

This instruction moves floating-point constant +0.0 to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is a pseudo-instruction of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CPY (immediate, merging) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

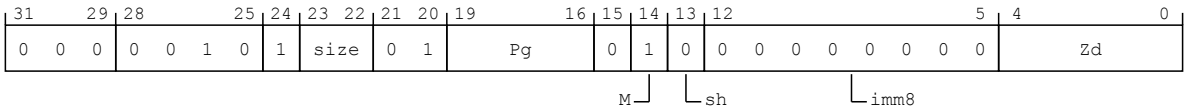

Encoding

```
FMOV  <Zd>.<T>, <Pg>/M, #0.0
```

is equivalent to

```
CPY  <Zd>.<T>, <Pg>/M, #0
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational Information

The description of CPY (immediate, merging) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.244 FMOV (zero, unpredicated)

Move floating-point +0.0 to vector elements (unpredicated)

This instruction unconditionally broadcasts the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This is a pseudo-instruction of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of DUP (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

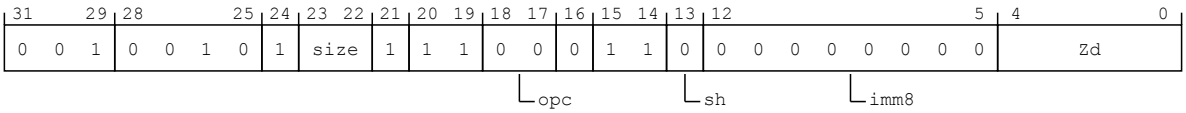

Encoding

```
FMOV  <Zd>.<T>, #0.0
```

is equivalent to

```
DUP  <Zd>.<T>, #0
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.245 FMOV (immediate, predicated)

Move floating-point immediate to vector elements (predicated)

This instruction moves a floating-point immediate into each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of FCPY. This means:

- The encodings in this description are named to match the encodings of FCPY.
- The description of FCPY gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15 13	12 5	4 0
0 0 0	0 0 1 0	1	size	0 1	Pg	1 1 0	imm8	Zd

Encoding

```
FMOV  <Zd>.<T>, <Pg>/M, #<const>
```

is equivalent to

```
FCPY  <Zd>.<T>, <Pg>/M, #<const>
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<const>

Is a floating-point immediate value expressible as ±n÷16×2^r, where n and r are integers such that $16 \leq n \leq 31$ and $-3 \leq r \leq 4$, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of FCPY gives the operational pseudocode for this instruction.

Operational Information

The description of FCPY gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.246 FMOV (immediate, unpredicated)

Move floating-point immediate to vector elements (unpredicated)

This instruction unconditionally broadcasts the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This is an alias of FDUP. This means:

- The encodings in this description are named to match the encodings of FDUP.
- The description of FDUP gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0 0 1	0 0 1 0	1	size	1	1 1	0 0 (opc)	1	1 1	0 (o2)	imm8	Zd

Encoding

```
FMOV  <Zd>.<T>, #<const>
```

is equivalent to

```
FDUP  <Zd>.<T>, #<const>
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<const>

Is a floating-point immediate value expressible as ±n÷16×2^r, where n and r are integers such that $16 \leq n \leq 31$ and $-3 \leq r \leq 4$, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Operation

The description of FDUP gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.247 FMSB

Floating-point fused multiply-subtract to multiplicand (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and subtracts the result from elements of the third (addend) vector without intermediate rounding. The results are destructively placed in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

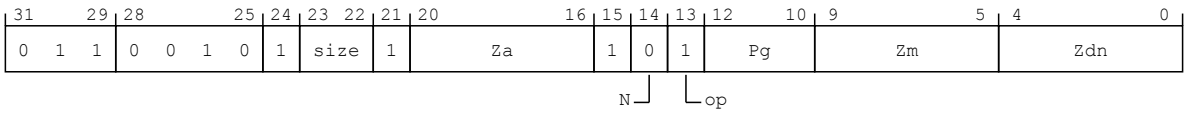

Encoding

```
FMSB  <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let a : integer = UInt(Za);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = Z{}(dn);
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(a) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.248 FMUL (immediate)

Floating-point multiply by immediate (predicated)

This instruction multiplies each active floating-point element of the source vector by an immediate, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +2.0 only. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 6	5	4 0
0 1 1	0 0 1 0	1	size	0	1 1	0 1 0 (opc)	1 0 0	Pg	0 0 0 0	i1	Zdn

Encoding

```
FMUL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then FPPointFive{esize}('0') else FPTwo{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#2.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPMul{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.249 FMUL (vectors, predicated)

Floating-point multiply (predicated)

This instruction multiplies active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	0 0 1 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMUL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPMul{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.250 FMUL (vectors, unpredicated)

Floating-point multiply (unpredicated)

This instruction multiplies all elements of the first source vector by the corresponding floating-point elements of the second source vector and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	Zm	0 0 0	0 1 0 (opc)	Zn	Zd

Encoding

```
FMUL  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPMul{esize}(element1, element2, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.251 FMUL (indexed)

Floating-point multiply by indexed element

This instruction multiplies all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision, and Double-precision

Half-precision

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 1 0	0	0	Zn	Zd
									o2			

Encoding

```
FMUL  <Zd>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Single-precision

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–12	11	10	9–5	4–0
0 1 1	0 0 1 0	0	1 0	1	i2	Zm	0 0 1 0	0	0	Zn	Zd
			size					o2			

Encoding

```
FMUL  <Zd>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Double-precision

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	1	i1	Zm	0 0 1 0	0	0	Zn	Zd
			size					o2			

Encoding

```
FMUL  <Zd>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “Half-precision” and “Single-precision” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “Double-precision” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “Half-precision” variant: is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “Single-precision” variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.

For the “Double-precision” variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let elem2 : bits(esize) = op2[s*:esize];
    let elem1 : bits(esize) = op1[e*:esize];
    result[e*:esize] = FPMul{esize}(elem1, elem2, FPCR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.252 FMULX

Floating-point multiply extended (predicated)

This instruction multiplies active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector except that $\infty \times 0.0$ gives 2.0 instead of NaN, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

The instruction can be used with `FRECPX` to safely convert arbitrary elements in mathematical vector space to unit vectors or direction vectors with length 1.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0	1 0 1 0 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FMULX  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPMulX{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.253 FNEG

Floating-point negate (predicated)

This instruction negates each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. This inverts the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	1 0 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
FNEG  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	1 0 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
FNEG  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPNeg{esize}(element, FPCR());

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.254 FNMAD

Floating-point negated fused multiply-add to multiplicand (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and adds the result to elements of the third (addend) vector without intermediate rounding. The negated results are destructively placed in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

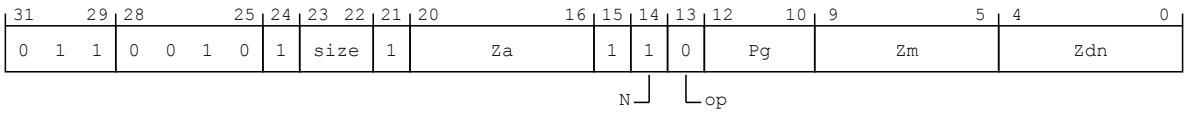

Encoding

```
FNMAD  <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let a : integer = UInt(Za);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = Z{}(dn);
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(a) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.255 FNMLA

Floating-point negated fused multiply-add (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and adds the result to elements of the third source (addend) vector without intermediate rounding. The negated results are destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

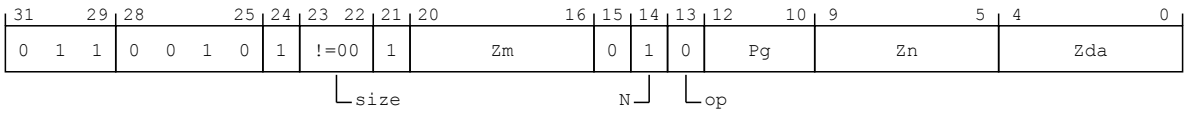

Encoding

```
FNMLA  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = TRUE;
let op3_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op3[e*:esize];

    end;
end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.256 FNMLS

Floating-point negated fused multiply-subtract (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and subtracts the result from elements of the third source (addend) vector without intermediate rounding. The negated results are destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	!=00	1	Zm	0	1	1	Pg	Zn	Zda
			size			N		op			

Encoding

```
FNMLS  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op3[e*:esize];

    end;
end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.257 FNMSB

Floating-point negated fused multiply-subtract to multiplicand (predicated)

This instruction multiplies the active floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, and subtracts the result from elements of the third (addend) vector without intermediate rounding. The negated results are destructively placed in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

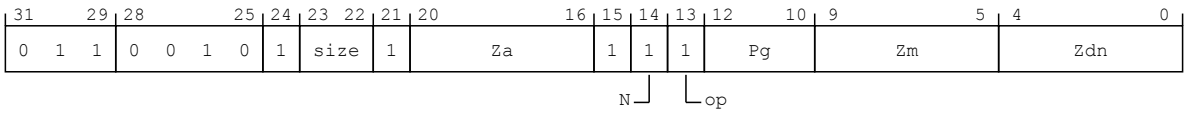

Encoding

```
FNMSB  <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let a : integer = UInt(Za);
let op1_neg : boolean = FALSE;
let op3_neg : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let op1 : bits(VL) = Z{}(dn);
let op2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let op3 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(a) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let elem1 : bits(esize) = (if op1_neg then FPNeg{esize}(op1[e*:esize], FPCR())
                                   else op1[e*:esize]);
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = (if op3_neg then FPNeg{esize}(op3[e*:esize], FPCR())
                                   else op3[e*:esize]);

        result[e*:esize] = FPMulAdd{esize}(elem3, elem1, elem2, FPCR());
    else
        result[e*:esize] = op1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.258 FRECPE

Floating-point reciprocal estimate (unpredicated)

This instruction finds the approximate reciprocal of each floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 1	1 1	0	0 0 1 1	0 0	Zn	Zd
						op				

Encoding

```
FRECPE  <Zd>.<T>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPRecipEstimate{esize}(element, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.259 FRECPS

Floating-point reciprocal step (unpredicated)

This instruction multiplies the floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, subtracts the products from 2.0 without intermediate rounding, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	size	0	Zm	0 0 0	1 1 0 (opc)	Zn	Zd

Encoding

```
FRECPS  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPRecipStepFused{esize}(element1, element2, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.260 FRECPX

Floating-point reciprocal exponent (predicated)

This instruction inverts the exponent, zeroes the fractional part of each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

The result of this instruction can be used with `FMULX` to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 1 1	0 0	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRECPX  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 1	0 1 1	1	0 0	Pg	Zn	Zd
							opc			

Encoding

```
FRECPX  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPRecpX{esize}(element, FPCR());
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.261 FRINT32X

Floating-point round to 32-bit integer (predicated)

This instruction rounds to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector.

Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2p2 || FEAT_SME2p2)

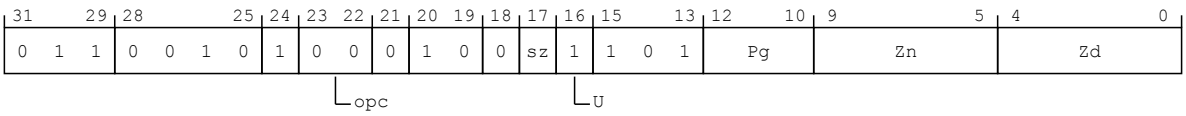

Encoding

```
FRINT32X  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 32;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

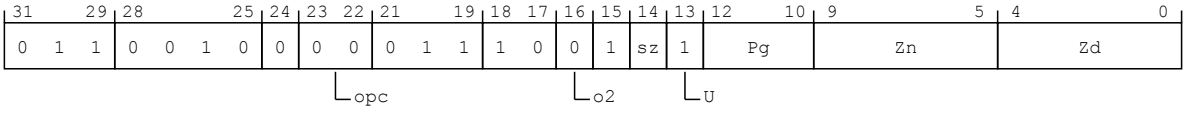

Encoding

```
FRINT32X  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 32;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.262 FRINT32Z

Floating-point round to 32-bit integer, rounding toward zero (predicated)

This instruction rounds to integral floating-point values that fit into a 32-bit integer size using the round towards zero rounding mode from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector.

Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0 (opc)	0	1 0	0	sz	0 (U)	1 0 1	Pg	Zn	Zd

Encoding

```
FRINT32Z  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 32;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 0 (opc)	0 1 1	1 0	0 (o2)	1	sz	0 (U)	Pg	Zn	Zd

Encoding

```
FRINT32Z  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 32;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.263 FRINT64X

Floating-point round to 64-bit integer (predicated)

This instruction rounds to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector.

Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0	1 0	1	sz	1	1 0 1	Pg	Zn	Zd
			opc					U				

Encoding

```
FRINT64X  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 64;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 0	0 1 1	1 0	1	1	sz	1	Pg	Zn	Zd
			opc			o2			U			

Encoding

```
FRINT64X  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 64;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.264 FRINT64Z

Floating-point round to 64-bit integer, rounding toward zero (predicated)

This instruction rounds to integral floating-point values that fit into a 64-bit integer size using the round towards zero rounding mode from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector.

Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 0	0	1 0	1	sz	0	1 0 1	Pg	Zn	Zd
			opc					U				

Encoding

```
FRINT64Z  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 64;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 0	0 1 1	1 0	1	1	sz	0	Pg	Zn	Zd
			opc			o2			U			

Encoding

```
FRINT64Z  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let intsize : integer{} = 64;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPRoundIntN{esize}(element, FPCR(), rounding, intsize);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.265 FRINT<r>

Floating-point round to integral value (predicated)

This instruction rounds to an integral floating-point value with the specified rounding option from each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

<r>	Rounding Option
N	to nearest, with ties to even
A	to nearest, with ties away from zero
M	toward minus Infinity
P	toward plus Infinity
Z	toward zero
I	current FPCR rounding mode
X	current FPCR rounding mode, signalling inexact

It has encodings from 14 classes: Current mode signalling inexact, merging, Current mode signalling inexact, zeroing, Current mode, merging, Current mode, zeroing, Nearest with ties to away, merging, Nearest with ties to away, zeroing, Nearest with ties to even, merging, Nearest with ties to even, zeroing, Toward zero, merging, Toward zero, zeroing, Toward minus infinity, merging, Toward minus infinity, zeroing, Toward plus infinity, merging, and Toward plus infinity, zeroing

Current mode signalling inexact, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 0	1 1 0 (opc)	1 0 1	Pg	Zn	Zd

Encoding

```
FRINTX  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Current mode signalling inexact, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	1	1	1 0	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTX  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

Current mode, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 0	1 1 1	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRINTI  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Current mode, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	1	1	1 1	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTI  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

Nearest with ties to away, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	size	0	0 0	1 0 0	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRINTA  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Nearest with ties to away, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	1	1	0 0	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTA  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
let merging : boolean = FALSE;
```

Nearest with ties to even, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 0	0 0 0	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRINTN  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Nearest with ties to even, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	0	1	0 0	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTN  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
let merging : boolean = FALSE;
```

Toward zero, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 0	0 1 1	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRINTZ  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Toward zero, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	0	1	1 1	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTZ  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
let merging : boolean = FALSE;
```

Toward minus infinity, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 0	0 1 0	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRINTM  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Toward minus infinity, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	0	1	1 0	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTM  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
let merging : boolean = FALSE;
```

Toward plus infinity, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	size	0	0 0	0 0 1	1 0 1	Pg	Zn	Zd
						opc				

Encoding

```
FRINTP  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
let merging : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Toward plus infinity, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14 13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	size	0 1 1	0 0	0	1	0 1	Pg	Zn	Zd
						op		opc2			

Encoding

```
FRINTP  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPRoundInt{esize}(element, FPCR(), rounding, exact);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Current mode signalling inexact, merging, Current mode, merging, Nearest with ties to away, merging, Nearest with ties to even, merging, Toward zero, merging, Toward minus infinity, merging, Toward plus infinity, merging variants:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.266 FRSQRTE

Floating-point reciprocal square root estimate (unpredicated)

This instruction finds the approximate reciprocal square root of each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18–17	16	15–12	11–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0 0 1	1 1	1	0 0 1 1	0 0	Zn	Zd
						op				

Encoding

```
FRSQRTE  <Zd>.<T>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[e*:esize];
    result[e*:esize] = FPRSqrtEstimate{esize}(element, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.267 FRSQRTS

Floating-point reciprocal square root step (unpredicated)

This instruction multiplies the floating-point elements of the first source vector by the corresponding floating-point elements of the second source vector, subtracts the products from 3.0 and divides the results by 2.0 without any intermediate rounding, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	Zm	0 0 0	1 1 1 (opc)	Zn	Zd

Encoding

```
FRSQRTS  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPRSqrtStepFused{esize}(element1, element2, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.268 FSCALE

Floating-point adjust exponent (predicated)

This instruction multiplies the active floating-point elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	1 0 0 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FSCALE  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:(esize)];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : integer = SInt(operand2[e*:(esize)]);
        result[e*:(esize)] = FPScale{esize}(element1, element2, FPCR());
    else
        result[e*:(esize)] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.269 FSQRT

Floating-point square root (predicated)

This instruction calculates the square root of each active floating-point element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	size	0	0 1 1	0 1 (opc)	1 0 1	Pg	Zn	Zd

Encoding

```
FSQRT  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–16	15	14–13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	size	0 1 1	0 1 1	1	0 1 (opc)	Pg	Zn	Zd

Encoding

```
FSQRT  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = FPSqrt{esize}(element, FPCR());
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as the merging variant this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The MOVPRFX must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.270 FSUB (immediate)

Floating-point subtract immediate (predicated)

This instruction subtracts an immediate from each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 6	5	4 0
0 1 1	0 0 1 0	1	size	0	1 1	0 0 1	1 0 0	Pg	0 0 0 0	i1	Zdn
						opc					

Encoding

```
FSUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then FPPointFive{esize}('0') else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPSub{esize}(element1, imm, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.271 FSUB (vectors, predicated)

Floating-point subtract (predicated)

This instruction subtracts active floating-point elements of the second source vector from the corresponding floating-point elements of the first source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	0	0 0 0 1 (opc)	1 0 0	Pg	Zm	Zdn

Encoding

```
FSUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPSub{esize}(element1, element2, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.272 FSUB (vectors, unpredicated)

Floating-point subtract (unpredicated)

This instruction subtracts all floating-point elements of the second source vector from the corresponding elements of the first source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	!=00 (size)	0	Zm	0 0 0	0 0 1 (opc)	Zn	Zd

Encoding

```
FSUB  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPSub{esize}(element1, element2, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.273 FSUBR (immediate)

Floating-point reversed subtract from immediate (predicated)

This instruction reverse subtracts from an immediate each active floating-point element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

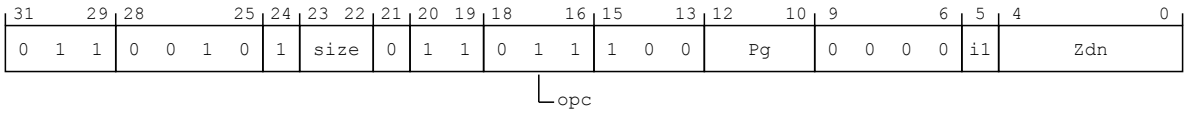

Encoding

```
FSUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let imm : bits(esize) = if i1 == '0' then FPPointFive{esize}('0') else FPOne{esize}('0');
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = FPSub{esize}(imm, element1, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.274 FSUBR (vectors)

Floating-point reversed subtract (predicated)

This instruction reverse subtracts active floating-point elements of the first source vector from the corresponding floating-point elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

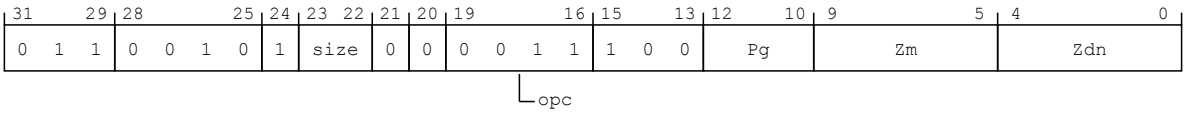

Encoding

```
FSUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);

let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPSub{esize}(element2, element1, FPCR());
    else
        result[e*:esize] = element1;
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.275 FTMAD

Floating-point trigonometric multiply-add coefficient

This instruction multiplies each element of the first source vector by the absolute value of the corresponding element of the second source vector and performs a fused addition of each product with a value obtained from a table of hard-wired coefficients, and destructively places the results in the first source vector. This instruction is unpredicated.

The coefficients are selected by a combination of the sign bit in the second source element and an immediate index in the range 0 to 7.

`FTMAD` can be combined with `FTSMUL` and `FTSSEL` to calculate values for sin(x) and cos(x). For more information, see `FTSMUL`. The coefficients are intended to provide accurate results for `FTSMUL` inputs in the range $-\pi/4 < x \leq \pi/4$.

For double-precision operations, the coefficients are:

element2<63>	Index	Hexadecimal	Approximate value
0	0	3ff0 0000 0000 0000	1
0	1	bfc5 5555 5555 5543	-1/3!
0	2	3f81 1111 1110 f30c	1/5!
0	3	bf2a 01a0 19b9 2fc6	-1/7!
0	4	3ec7 1de3 51f3 d22b	1/9!
0	5	be5a e5e2 b60f 7b91	-1/11!
0	6	3de5 d840 8868 552f	1/13!
0	7	0000 0000 0000 0000	0
1	0	3ff0 0000 0000 0000	1
1	1	bfe0 0000 0000 0000	-1/2!
1	2	3fa5 5555 5555 5536	1/4!
1	3	bf56 c16c 16c1 3a0b	-1/6!
1	4	3efa 01a0 19b1 e8d8	1/8!
1	5	be92 7e4f 7282 f468	-1/10!
1	6	3e21 ee96 d264 1b13	1/12!
1	7	bda8 f763 80fb b401	-1/14!

For single-precision operations, the coefficients are:

element2<63>	Index	Hexadecimal	Approximate value
0	0	3f80 0000	1
0	1	be2a aaab	-1/3!
0	2	3c08 8886	1/5!
0	3	b950 08b9	-1/7!
0	4	3636 9d6d	1/9!
0	5	0000 0000	0
0	6	0000 0000	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

element2<63>	Index	Hexadecimal	Approximate value
0	7	0000 0000	0
1	0	3f80 0000	1
1	1	bf00 0000	-1/2!
1	2	3d2a aaa6	1/4!
1	3	bab6 0705	-1/6!
1	4	37cd 37cc	1/8!
1	5	0000 0000	0
1	6	0000 0000	0
1	7	0000 0000	0

For half-precision operations, the coefficients are:

element2<63>	Index	Hexadecimal	Approximate value
0	0	3c00	1
0	1	b155	-1/3!
0	2	2030	1/5!
0	3	0000	0
0	4	0000	0
0	5	0000	0
0	6	0000	0
0	7	0000	0
1	0	3c00	1
1	1	b800	-1/2!
1	2	293a	1/4!
1	3	0000	0
1	4	0000	0
1	5	0000	0
1	6	0000	0
1	7	0000	0

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	size	0	1 0	imm3	1 0 0	0 0 0	Zm	Zdn

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FTMAD  <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let imm : integer = UInt(imm3);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<imm>

Is the unsigned immediate operand, in the range 0 to 7, encoded in the “imm3” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPTrigMAdd{esize}(imm, element1, element2, FPCR());
end;

Z{VL}(dn) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.276 FTSMUL

Floating-point trigonometric starting value

This instruction multiplies each element of the first source vector by itself, replaces the sign-bit of each product with the least-significant bit of the corresponding element of the second source vector, and places the results in the destination vector. This instruction is unpredicated.

`FTSMUL` can be combined with `FTMAD` and `FTSSEL` to calculate values for sin(x) and cos(x). The use of the second operand is consistent with it holding an integer corresponding to the desired sine-wave quadrant when used in conjunction with `FTMAD`.

Note: `FTSMUL`, `FTMAD`, and `FTSSEL` can be used to calculate values for sin(k) and correspondingly cos(k-π/2) via an intermediate value x, in the range $-\pi/4 < x \leq \pi/4$, and a quadrant q where $k = q\pi/2 + x$, using a Taylor Series approximation. `FTSMUL` can be used to calculate x^2, and to insert q<0> into the most-significant bit, indicating the desired odd versus even Taylor Series to be used in `FTMAD`. Repeated uses of `FTMAD` can be performed on this value with decreasing immediate index operands, to produce a single accumulated value approximating the Taylor Series result with a single outstanding factor. `FTSSEL` can be used to apply the final factor of x, 1.0, -x, or -1.0, dependent on the corresponding sine-wave quadrant q<1:0>, to produce a final sin() or cos() value.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

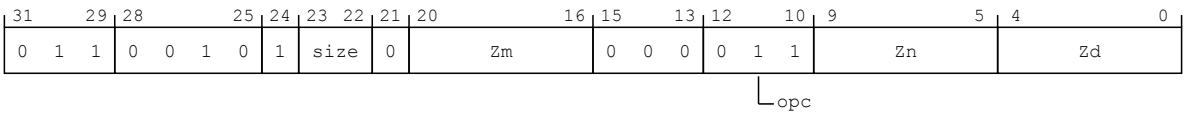

Encoding

```
FTSMUL  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPTrigSMul{esize}(element1, element2, FPCR());
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.277 FTSSEL

Floating-point trigonometric select coefficient

This instruction selects either the element of the first source vector or the value 1.0, based on bit<0> of the corresponding element of the second source element, negates the result if bit<1> of the corresponding element of the second source element is set, and places the results in the destination vector. This instruction is unpredicated.

`FTSSEL` can be combined with `FTSMUL` and `FTMAD` to calculate values for sin(k) and cos(k). For more information, see `FTSMUL`. The use of the second operand is consistent with it holding an integer corresponding to the desired sine-wave quadrant.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	size	1	Zm	1 0 1 1	0	0	Zn	Zd
								op		

Encoding

```
FTSSEL  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = FPTrigSSel{esize}(element1, element2);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.278 HISTCNT

Count matching elements in vector

This instruction compares each active 32 or 64-bit element of the first source vector with all Active elements with an element number less than or equal to its own in the second source vector, and places the count of matching elements in the corresponding element of the destination vector. Inactive elements in the destination vector are set to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
0 1 0	0 0 1 0	1	size	1	Zm	1 1 0	Pg	Zn	Zd

Encoding

```
HISTCNT  <Zd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let d : integer = UInt(Zd);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    var count : integer = 0;
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        for i = 0 to e do
            if ActivePredicateElement{PL}(mask, i, esize) then
                let element2 : bits(esize) = operand2[i*:esize];
                if element1 == element2 then
                    count = count + 1;
                end;
            end;
        end;
    end;
    result[e*:esize] = count[esize-1:0];
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.279 HISTSEG

Count matching elements in vector segments

This instruction compares each 8-bit byte element of the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector and places the count of matching elements in the corresponding element of the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
0 1 0	0 0 1 0	1	size	1	Zm	1 0 1	0 0 0	Zn	Zd

Encoding

```
HISTSEG  <Zd>.B, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
if size != '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8;
let d : integer = UInt(Zd);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for b = 0 to segments-1 do
    for s = 0 to eltspersegment-1 do
        var count : integer = 0;
        let e : integer = eltspersegment * b + s;
        let element1 : bits(esize) = operand1[e*:esize];
        for i = 0 to eltspersegment-1 do
            let e2 : integer = eltspersegment * b + i;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let element2 : bits(esize) = operand2[e2*:esize];
            if element1 == element2 then
                count = count + 1;
            end;
        end;
        result[e*:esize] = count[esize-1:0];
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.280 INCB, INCD, INCH, INCW (scalar)

Increment scalar by multiple of predicate constraint element count

This instruction determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword, and Word

Byte

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 0 (size)	1	1	imm4	1 1	1 0 0	0 (D)	pattern	Rdn

Encoding

```
INCB  <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Doubleword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1 (size)	1	1	imm4	1 1	1 0 0	0 (D)	pattern	Rdn

Encoding

```
INCD  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Halfword

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 14	13 11	10	9 5	4 0
0 0 0	0 0 1 0	0	0 1	1	1	imm4	1 1	1 0 0	0	pattern	Rdn
			size						D		

Encoding

```
INCH  <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Word

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 14	13 11	10	9 5	4 0
0 0 0	0 0 1 0	0	1 0	1	1	imm4	1 1	1 0 0	0	pattern	Rdn
			size						D		

Encoding

```
INCW  <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let VL : integer{} = CurrentVL();
let operand1 : bits(64) = X{}(dn);
X{64}(dn) = operand1 + (count * imm);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.281 INCD, INCH, INCW (vector)

Increment vector by multiple of predicate constraint element count

This instruction determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword, Halfword, and Word

Doubleword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1 (size)	1	1	imm4	1 1	0 0 0	0 (D)	pattern	Zdn

Encoding

```
INCD  <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Halfword

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 1 (size)	1	1	imm4	1 1	0 0 0	0 (D)	pattern	Zdn

Encoding

```
INCH  <Zdn>.H{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Word

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 14	13 11	10	9 5	4 0
0 0 0	0 0 1 0	0	1 0	1	1	imm4	1 1	0 0 0	0	pattern	Zdn
			size						D		

Encoding

```
INCW  <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    result[e*:esize] = operand1[e*:esize] + (count * imm);

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.282 INCP (scalar)

Increment scalar by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to increment the scalar destination.

SVE

(FEAT_SVE || FEAT_SME)

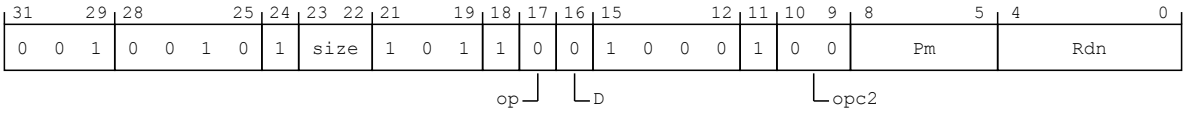

Encoding

```
INCP  <Xdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(64) = X{}(dn);
let operand2 : bits(PL) = P{}(m);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

X{64}(dn) = operand1 + count;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.283 INCP (vector)

Increment vector by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to increment all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18	17	16	15–12	11	10–9	8–5	4–0
0 0 1	0 0 1 0	1	size	1 0 1	1	0	0	1 0 0 0	0	0 0	Pm	Zdn
						op	D			opc2		

Encoding

```
INCP  <Zdn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(VL);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

for e = 0 to elements-1 do
    result[e*:esize] = operand1[e*:esize] + count;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.284 INDEX (immediates)

Create index starting from and incremented by immediate

This instruction populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	imm5b	0 1 0 0	0 0	imm5	Zd

Encoding

```
INDEX  <Zd>.<T>, #<imm1>, #<imm2>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd);
let imm1 : integer = SInt(imm5);
let imm2 : integer = SInt(imm5b);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm1>

Is the first signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

<imm2>

Is the second signed immediate operand, in the range -16 to 15, encoded in the “imm5b” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var result : bits(VL);

for e = 0 to elements-1 do
    let index : integer = imm1 + e * imm2;
    result[e*:esize] = index[esize-1:0];
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.285 INDEX (immediate, scalar)

Create index starting from immediate and incremented by general-purpose register

This instruction populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Rm	0 1 0 0	1 0	imm5	Zd

Encoding

```
INDEX  <Zd>.<T>, #<imm>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Rm);
let d : integer = UInt(Zd);
let imm : integer = SInt(imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand2 : bits(esize) = X{}(m);
let element2 : integer = SInt(operand2);
var result : bits(VL);

for e = 0 to elements-1 do
    let index : integer = imm + e * element2;
    result[e*:esize] = index[esize-1:0];
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.286 INDEX (scalar, immediate)

Create index starting from general-purpose register and incremented by immediate

This instruction populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	imm5	0 1 0 0	0 1	Rn	Zd

Encoding

```
INDEX  <Zd>.<T>, <R><n>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Rn);
let d : integer = UInt(Zd);
let imm : integer = SInt(imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<imm>

Is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(esize) = X{}(n);
let element1 : integer = SInt(operand1);
var result : bits(VL);

for e = 0 to elements-1 do
    let index : integer = element1 + e * imm;
    result[e*:esize] = index[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.287 INDEX (scalars)

Create index starting from and incremented by general-purpose register

This instruction populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Rm	0 1 0 0	1 1	Rn	Zd

Encoding

```
INDEX  <Zd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(esize) = X{}(n);
let element1 : integer = SInt(operand1);
let operand2 : bits(esize) = X{}(m);
let element2 : integer = SInt(operand2);
var result : bits(VL);

for e = 0 to elements-1 do
    let index : integer = element1 + e * element2;
    result[e*:esize] = index[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.288 INSR (scalar)

Insert general-purpose register in shifted vector

This instruction shifts the destination vector left by one element, and then places a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0 0	1 0 0	0 0 1 1 1 0	Rm	Zdn

Encoding

```
INSR  <Zdn>.<T>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Rm);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R>

Is a width specifier, encoded in “size”:

size	<R>
00	W

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
01	W
10	W
11	X

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let dest : bits(VL) = Z{}(dn);
let src : bits(esize) = X{}(m);
Z{VL}(dn) = dest[(VL-esize)-1:0] :: src;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.289 INSR (SIMD&FP scalar)

Insert SIMD&FP scalar register in shifted vector

This instruction shifts the destination vector left by one element, and then places a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	1 0	1 0 0	0 0 1 1 1 0	Vm	Zdn

Encoding

```
INSR  <Zdn>.<T>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Vm);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<V>
01	H
10	S
11	D

<m>

Is the number [0-31] of the source SIMD&FP register, encoded in the “Vm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let dest : bits(VL) = Z{}(dn);
let src : bits(esize) = V{}(m);
Z{VL}(dn) = dest[(VL-esize)-1:0] :: src;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.290 LASTA (scalar)

Extract element after last to general-purpose register

This instruction extracts the element after the Last active element to a general-purpose register. If there is an Active element, the element after the Last active element modulo the number of elements from the final source vector register is extracted. If there are no Active elements, element zero is extracted. Then the extracted element is zero-extended and placed in the destination general-purpose register.

SVE

(FEAT_SVE || FEAT_SME)

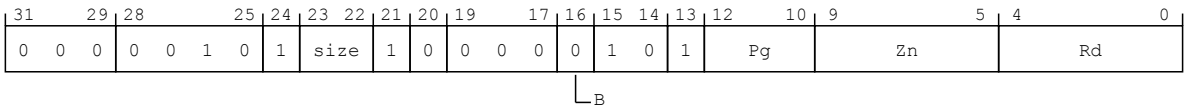

Encoding

```
LASTA  <R><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = if esize < 64 then 32 else 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Rd);
let isBefore : boolean = FALSE;
```

Assembler Symbols

<R>

Is a width specifier, encoded in “size”:

size	<R>
00	W
01	W
10	W
11	X

<d>

Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the “Rd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = Z{}(n);
var result : bits(rsize);
var last : integer = LastActiveElement{PL}(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1; end;
else
    last = last + 1;
    if last >= elements then last = 0; end;
end;
result = ZeroExtend{rsize}(operand[last*:esize]);

X{rsize}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.291 LASTA (SIMD&FP scalar)

Extract element after last to SIMD&FP scalar register

This instruction extracts the element after the Last active element to a SIMD&FP scalar register. If there is an Active element, the element after the Last active element modulo the number of elements from the final source vector register is extracted. If there are no Active elements, element zero is extracted. Then the extracted element is placed in the destination SIMD&FP scalar register.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 0 1	0	1 0	0	Pg	Zn	Vd
							B					

Encoding

```
LASTA  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let isBefore : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = Z{}(n);
var last : integer = LastActiveElement{PL}(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1; end;
else
    last = last + 1;
    if last >= elements then last = 0; end;
end;
V{esize}(d) = operand[last*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.292 LASTB (scalar)

Extract last element to general-purpose register

This instruction extracts the Last active element to a general-purpose register. If there is an Active element, the Last active element from the final source vector register is extracted. If there are no Active elements, the highest-numbered element is extracted. Then the extracted element is zero-extended and placed in the destination general-purpose register.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 0 0	1	1 0	1	Pg	Zn	Rd
							B					

Encoding

```
LASTB  <R><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = if esize < 64 then 32 else 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Rd);
let isBefore : boolean = TRUE;
```

Assembler Symbols

<R>

Is a width specifier, encoded in “size”:

size	<R>
00	W
01	W
10	W
11	X

<d>

Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the “Rd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = Z{}(n);
var result : bits(rsize);
var last : integer = LastActiveElement{PL}(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1; end;
else
    last = last + 1;
    if last >= elements then last = 0; end;
end;
result = ZeroExtend{rsize}(operand[last*:esize]);

X{rsize}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.293 LASTB (SIMD&FP scalar)

Extract last element to SIMD&FP scalar register

This instruction extracts the Last active element to a SIMD&FP scalar register. If there is an Active element, the Last active element from the final source vector register is extracted. If there are no Active elements, the highest-numbered element is extracted. Then the extracted element is placed in the destination SIMD&FP register.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	0	0 0 1	1	1 0	0	Pg	Zn	Vd
							B					

Encoding

```
LASTB  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let isBefore : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = Z{}(n);
var last : integer = LastActiveElement{PL}(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1; end;
else
    last = last + 1;
    if last >= elements then last = 0; end;
end;
V{esize}(d) = operand[last*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.294 LASTP

Scalar index of last true predicate element (predicated)

This instruction determines the index of the last active and true element of the source predicate, and places the scalar result in the destination general-purpose register. If there are no active and true elements, the destination general-purpose register is updated to the value minus one.

SVE2

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–16	15–14	13–10	9	8–5	4–0
0 0 1	0 0 1 0	1	size	1 0 0	0 1 0 (opc)	1 0	Pg	0	Pn	Rd

Encoding

```
LASTP  <Xd>, <Pg>, <Pn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let d : integer = UInt(Rd);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(PL) = P{}(n);
var index : integer = -1;

for e = elements-1 downto 0 do
    if (index == -1 && ActivePredicateElement{PL}(mask, e, esize) &&
          ActivePredicateElement{PL}(operand, e, esize)) then
        index = e;
    end;
end;

X{64}(d) = index[63:0];
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.295 LD1B (scalar plus immediate, consecutive registers)

Contiguous load of bytes to multiple consecutive vectors (immediate index)

This instruction performs a contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

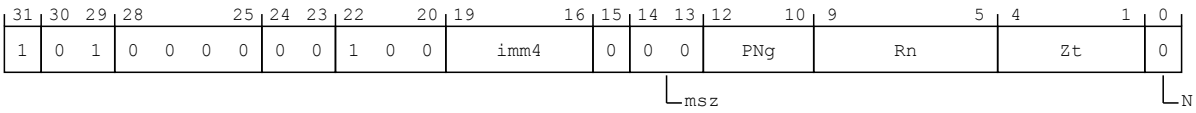

Encoding

```
LD1B  { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

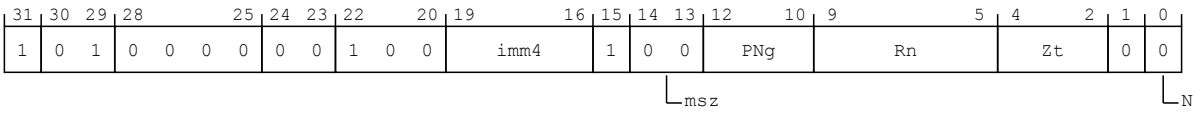

Encoding

```
LD1B  { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.296 LD1B (scalar plus scalar, consecutive registers)

Contiguous load of bytes to multiple consecutive vectors (scalar index)

This instruction performs a contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	0	0	0	PNg	Rn		Zt		0
												msz								N

Encoding

```
LD1B  { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	1	0	0	PNg	Rn		Zt		0	0
													msz								N

Encoding

```
LD1B  { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.297 LD1B (vector plus immediate)

Gather load unsigned bytes to vector (immediate index)

This instruction performs a gather load of unsigned bytes to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

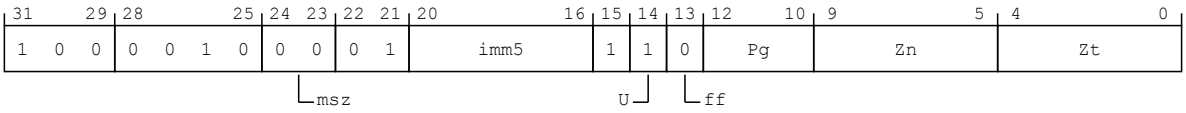

Encoding

```
LD1B  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

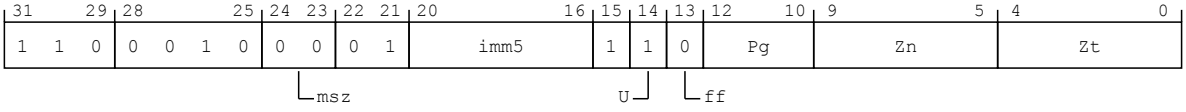

Encoding

```
LD1B  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.298 LD1B (scalar plus immediate, single register)

Contiguous load unsigned bytes to vector (immediate index)

This instruction performs a contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element, and 64-bit element

8-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 0 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1B  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 0 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1B  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

32-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1B  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.299 LD1B (scalar plus scalar, single register)

Contiguous load unsigned bytes to vector (scalar index)

This instruction performs a contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element, and 64-bit element

8-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 0 0	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1B  { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 0 1	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1B  { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

32-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 0	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1B  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 1	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.300 LD1B (scalar plus vector)

Gather load unsigned bytes to vector (vector index)

This instruction performs a gather load of unsigned bytes to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign-extended or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset, and 64-bit unscaled offset

32-bit unpacked unscaled offset

(FEAT_SVE)

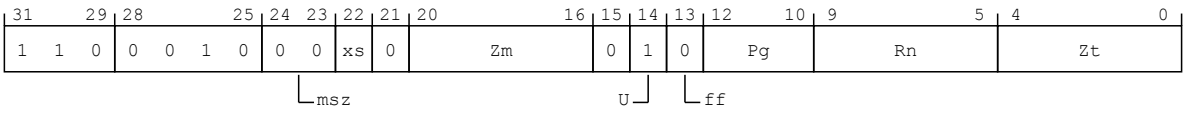

Encoding

```
LD1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

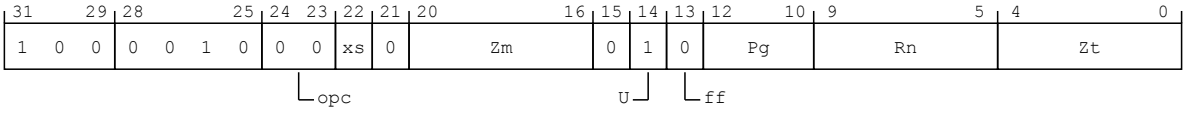

Encoding

```
LD1B  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 0	1 0	Zm	1	1	0	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LD1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.301 LD1D (scalar plus immediate, consecutive registers)

Contiguous load of doublewords to multiple consecutive vectors (immediate index)

This instruction performs a contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22		20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4		0	1	1	PNg		Rn		Zt		0
															msz								N

Encoding

```
LD1D  { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22		20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4		1	1	1	PNg		Rn		Zt		0	0
															msz									N

Encoding

```
LD1D  { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.302 LD1D (scalar plus scalar, consecutive registers)

Contiguous load of doublewords to multiple consecutive vectors (scalar index)

This instruction performs a contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	0	1	1	PNg	Rn		Zt		0
													msz							N

Encoding

```
LD1D  { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	1	1	1	PNg	Rn		Zt		0	0
													msz								N

Encoding

```
LD1D  { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.303 LD1D (vector plus immediate)

Gather load doublewords to vector (immediate index)

This instruction performs a gather load of doublewords to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

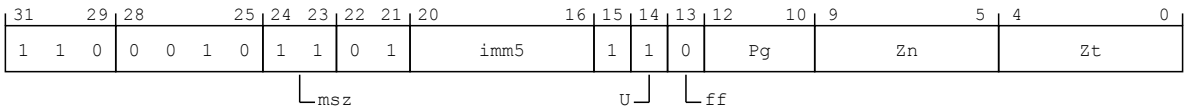

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.304 LD1D (scalar plus immediate, single register)

Contiguous load unsigned doublewords to vector (immediate index)

This instruction performs a contiguous load of unsigned doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 64-bit element and 128-bit element

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

128-bit element

(FEAT_SVE2p1)

31–29	28–25	24 23	22 21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1	0 0	1	imm4	0 0 1	Pg	Rn	Zt
		dtype							

Encoding

```
LD1D  { <Zt>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.305 LD1D (scalar plus scalar, single register)

Contiguous load unsigned doublewords to vector (scalar index)

This instruction performs a contiguous load of unsigned doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 64-bit element and 128-bit element

64-bit element

(FEAT_SVE || FEAT_SME)

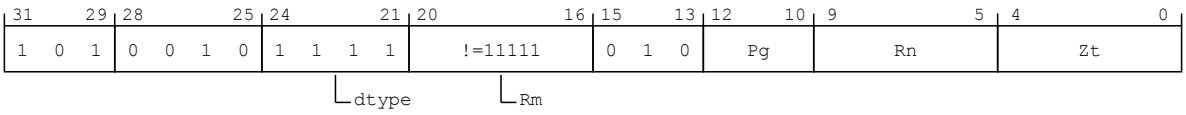

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
```

128-bit element

(FEAT_SVE2p1)

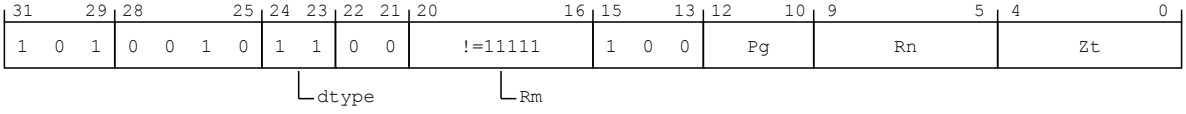

Encoding

```
LD1D  { <Zt>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.306 LD1D (scalar plus vector)

Gather load doublewords to vector (vector index)

This instruction performs a gather load of doublewords to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit unpacked scaled offset

(FEAT_SVE)

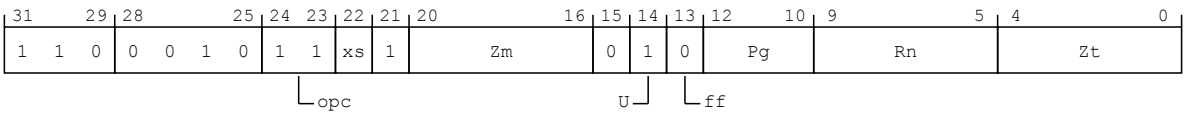

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 3;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

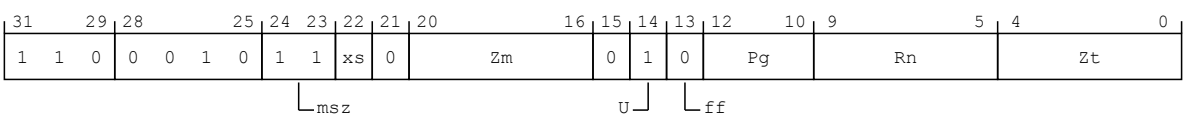

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 1	1 1	Zm	1	1	0	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 3;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 1	1 0	Zm	1	1	0	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LD1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.307 LD1H (scalar plus immediate, consecutive registers)

Contiguous load of halfwords to multiple consecutive vectors (immediate index)

This instruction performs a contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		0	0	1	PNg		Rn		Zt		0
												msz								N

Encoding

```
LD1H  { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		1	0	1	PNg		Rn		Zt		0	0
												msz									N

Encoding

```
LD1H  { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.308 LD1H (scalar plus scalar, consecutive registers)

Contiguous load of halfwords to multiple consecutive vectors (scalar index)

This instruction performs a contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	0	0	1	PNg	Rn		Zt		0
													msz							N

Encoding

```
LD1H  { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	1	0	1	PNg	Rn		Zt		0	0
													msz								N

Encoding

```
LD1H  { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.309 LD1H (vector plus immediate)

Gather load unsigned halfwords to vector (immediate index)

This instruction performs a gather load of unsigned halfwords to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

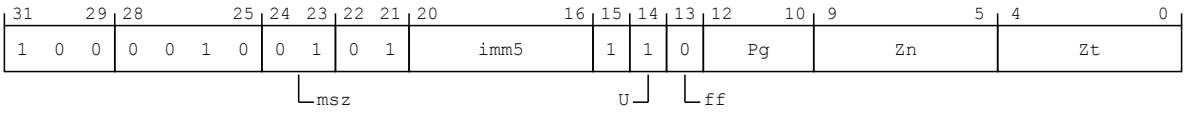

Encoding

```
LD1H  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

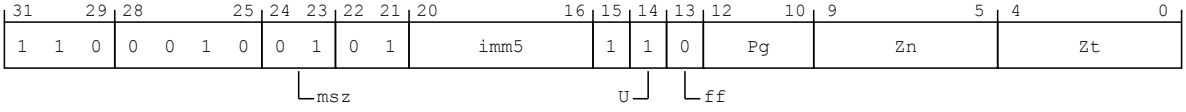

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.310 LD1H (scalar plus immediate, single register)

Contiguous load unsigned halfwords to vector (immediate index)

This instruction performs a contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 0 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1H  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 1 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 1 1 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.311 LD1H (scalar plus scalar, single register)

Contiguous load unsigned halfwords to vector (scalar index)

This instruction performs a contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 0 1	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1H  { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 1 0	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 1 1 1	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.312 LD1H (scalar plus vector)

Gather load unsigned halfwords to vector (vector index)

This instruction performs a gather load of unsigned halfwords to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

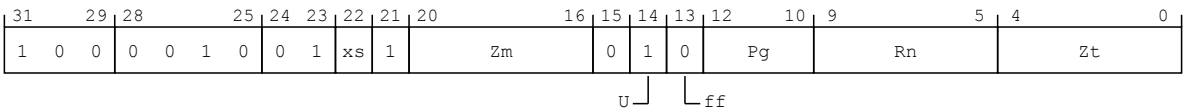

Encoding

```
LD1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked scaled offset

(FEAT_SVE)

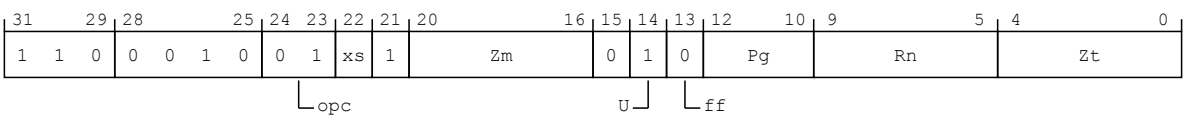

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	xs	0	Zm	0	1	0	Pg	Rn	Zt
		msz				U		ff			

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 1	xs	0	Zm	0	1	0	Pg	Rn	Zt
		opc				U		ff			

Encoding

```
LD1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	1 1	Zm	1	1	0	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 1;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	1 0	Zm	1	1	0	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.313 LD1Q

Gather load quadwords

This instruction performs a gather load of quadwords to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2p1)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 0	0 0	Rm	1 0 1	Pg	Zn	Zt

Encoding

```
LD1Q  { <Zt>.Q }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
CheckNonStreamingSVEEnabled();
let elements : integer = VL DIV 128;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, 128) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 128) then
        let baddr : bits(64) = base[(2*e)*:64];
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        result[e*:128] = Mem{128}(addr, accdesc);
    else
        result[e*:128] = Zeros{128};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.314 LD1RB

Load and broadcast unsigned byte to vector

This instruction loads a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate offset that is in the range 0 to 63.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element, and 64-bit element

8-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 0	1	imm6	1	0 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RB  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 0	1	imm6	1	0 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RB  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

32-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 0	1	imm6	1	1 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 0	1	imm6	1	1 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.315 LD1RD

Load and broadcast doubleword to vector

This instruction loads a single doubleword from a memory address generated by a 64-bit scalar base address plus an immediate offset that is a multiple of 8 in the range 0 to 504.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

SVE

(FEAT_SVE || FEAT_SME)

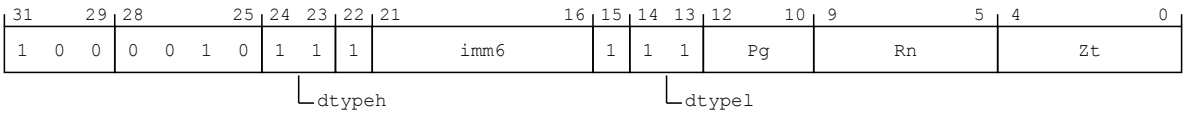

Encoding

```
LD1RD  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the “imm6” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.316 LD1RH

Load and broadcast unsigned halfword to vector

This instruction loads a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset that is a multiple of 2 in the range 0 to 126.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1	1	imm6	1	0 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RH  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1	1	imm6	1	1 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 1	1	imm6	1	1 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the “imm6” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.317 LD1ROB (scalar plus immediate)

Contiguous load and replicate thirty-two bytes (immediate index)

This instruction loads thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0	0 1	0	imm4	0 0 1	Pg	Rn	Zt
		msz	ssz						

Encoding

```
LD1ROB  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.318 LD1ROB (scalar plus scalar)

Contiguous load and replicate thirty-two bytes (scalar index)

This instruction loads thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

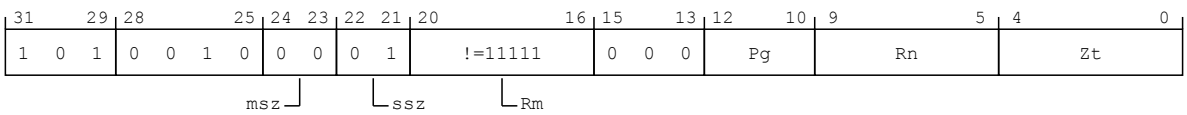

Encoding

```
LD1ROB  { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var offset : bits(64);
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.319 LD1ROD (scalar plus immediate)

Contiguous load and replicate four doublewords (immediate index)

This instruction loads four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

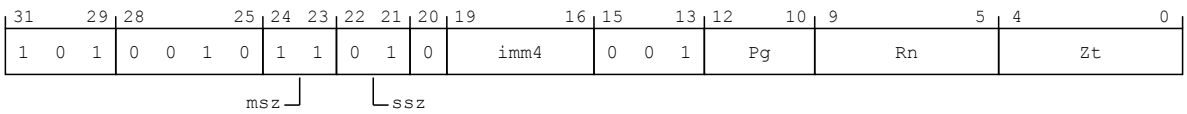

Encoding

```
LD1ROD  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.320 LD1ROD (scalar plus scalar)

Contiguous load and replicate four doublewords (scalar index)

This instruction loads four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

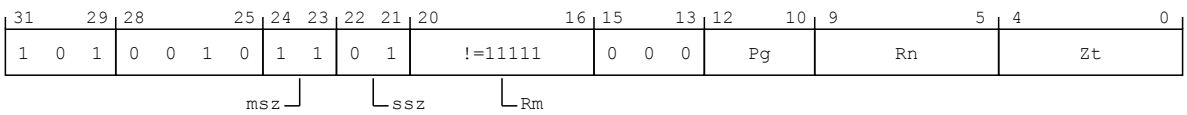

Encoding

```
LD1ROD  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var offset : bits(64);
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.321 LD1ROH (scalar plus immediate)

Contiguous load and replicate sixteen halfwords (immediate index)

This instruction loads sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1	0 1	0	imm4	0 0 1	Pg	Rn	Zt
		msz	ssz						

Encoding

```
LD1ROH  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.322 LD1ROH (scalar plus scalar)

Contiguous load and replicate sixteen halfwords (scalar index)

This instruction loads sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

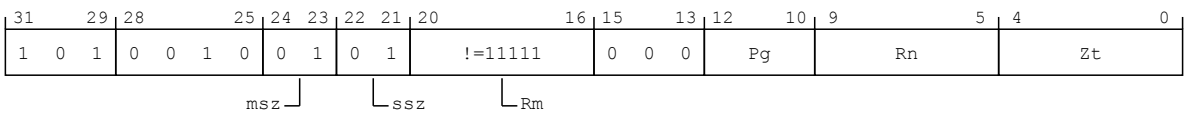

Encoding

```
LD1ROH  { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var offset : bits(64);
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.323 LD1ROW (scalar plus immediate)

Contiguous load and replicate eight words (immediate index)

This instruction loads eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 (msz)	0 1 (ssz)	0	imm4	0 0 1	Pg	Rn	Zt

Encoding

```
LD1ROW  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.324 LD1ROW (scalar plus scalar)

Contiguous load and replicate eight words (scalar index)

This instruction loads eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the Effective SVE vector length is at least 256 bits.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_F64MM)

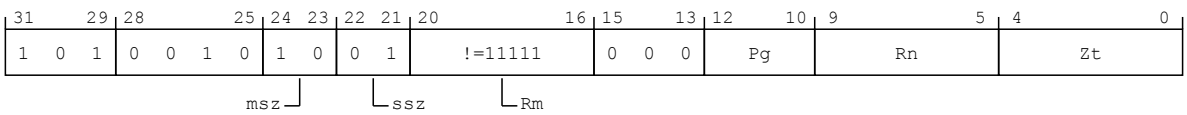

Encoding

```
LD1ROW  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
if VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = 256 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low bits only
var offset : bits(64);
var addr : bits(64);
var result : bits(256);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = ZeroExtend{VL}(Replicate{VL}(result));
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.325 LD1RQB (scalar plus immediate)

Contiguous load and replicate sixteen bytes (immediate index)

This instruction loads sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

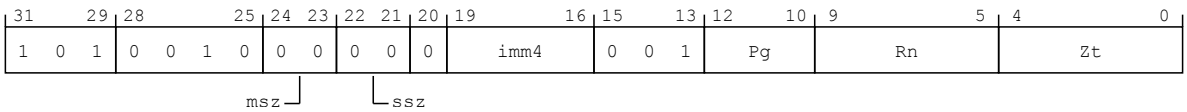

Encoding

```
LD1RQB  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var addr : bits(64);
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * 16, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.326 LD1RQB (scalar plus scalar)

Contiguous load and replicate sixteen bytes (scalar index)

This instruction loads sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

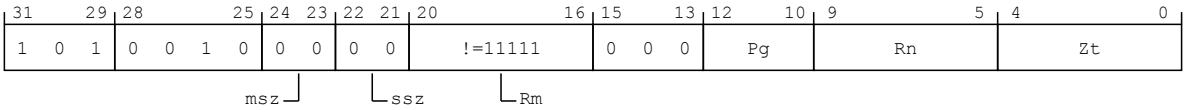

Encoding

```
LD1RQB  { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var offset : bits(64);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.327 LD1RQD (scalar plus immediate)

Contiguous load and replicate two doublewords (immediate index)

This instruction loads two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

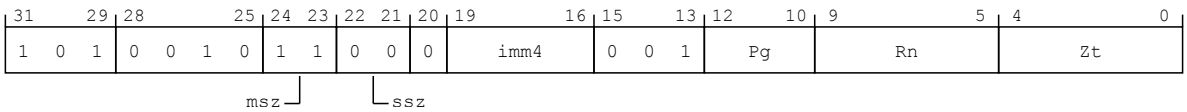

Encoding

```
LD1RQD  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var addr : bits(64);
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * 16, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.328 LD1RQD (scalar plus scalar)

Contiguous load and replicate two doublewords (scalar index)

This instruction loads two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 (msz)	0 0 (ssz)	!=11111 (Rm)	0 0 0	Pg	Rn	Zt

Encoding

```
LD1RQD  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var offset : bits(64);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.329 LD1RQH (scalar plus immediate)

Contiguous load and replicate eight halfwords (immediate index)

This instruction loads eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

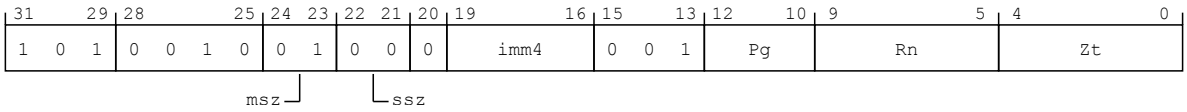

Encoding

```
LD1RQH  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var addr : bits(64);
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * 16, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.330 LD1RQH (scalar plus scalar)

Contiguous load and replicate eight halfwords (scalar index)

This instruction loads eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

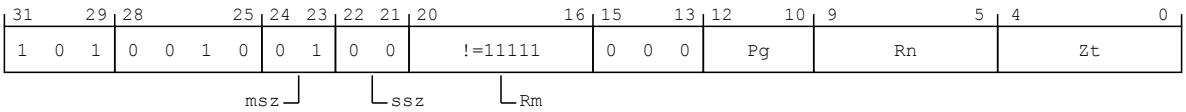

Encoding

```
LD1RQH  { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var offset : bits(64);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.331 LD1RQW (scalar plus immediate)

Contiguous load and replicate four words (immediate index)

This instruction loads four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 (msz)	0 0 (ssz)	0	imm4	0 0 1	Pg	Rn	Zt

Encoding

```
LD1RQW  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var addr : bits(64);
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * 16, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.332 LD1RQW (scalar plus scalar)

Contiguous load and replicate four words (scalar index)

This instruction loads four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index that is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 (msz)	0 0 (ssz)	!=11111 (Rm)	0 0 0	Pg	Rn	Zt

Encoding

```
LD1RQW  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = 128 DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g); // low 16 bits only
var offset : bits(64);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(128);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = Replicate{VL}(result);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.333 LD1RSB

Load and broadcast signed byte to vector

This instruction loads a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset that is in the range 0 to 63.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 1	1	imm6	1	1 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RSB  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm6);
```

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 1	1	imm6	1	0 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RSB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm6);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1 0 0	0 0 1 0	1 1	1	imm6	1	0 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RSB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the “imm6” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.334 LD1RSH

Load and broadcast signed halfword to vector

This instruction loads a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset that is a multiple of 2 in the range 0 to 126.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 0	1	imm6	1	0 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RSH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm6);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 0	1	imm6	1	0 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RSH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.335 LD1RSW

Load and broadcast signed word to vector

This instruction loads a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset that is a multiple of 4 in the range 0 to 252.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15	14 13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 1	1	imm6	1	0 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RSW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the “imm6” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.336 LD1RW

Load and broadcast unsigned word to vector

This instruction loads a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset that is a multiple of 4 in the range 0 to 252.

The loaded data is broadcast into all Active elements of the destination vector, setting the Inactive elements to zero. If all elements are inactive, then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 0	1	imm6	1	1 0	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RW  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 0	1	imm6	1	1 1	Pg	Rn	Zt
		dtypeh				dtypel			

Encoding

```
LD1RW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm6);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    let addr : bits(64) = AddressAdd(base, offset * mbytes, accdesc);
    data = Mem{msize}(addr, accdesc);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.337 LD1SB (vector plus immediate)

Gather load signed bytes to vector (immediate index)

This instruction performs a gather load of signed bytes to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

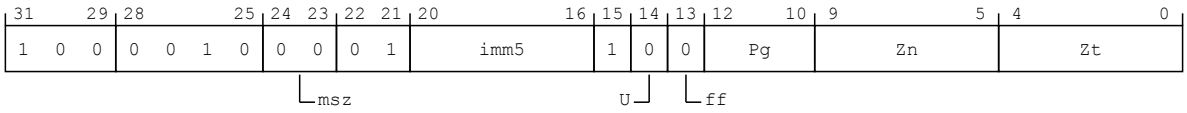

Encoding

```
LD1SB  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

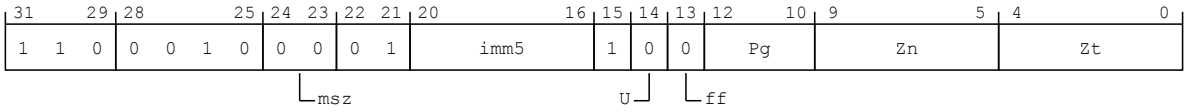

Encoding

```
LD1SB  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.338 LD1SB (scalar plus immediate)

Contiguous load signed bytes to vector (immediate index)

This instruction performs a contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1SB  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 0 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1SB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 1 0 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.339 LD1SB (scalar plus scalar)

Contiguous load signed bytes to vector (scalar index)

This instruction performs a contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 0	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1SB  { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 0 1	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1SB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

64-bit element

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 1 0 0	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.340 LD1SB (scalar plus vector)

Gather load signed bytes to vector (vector index)

This instruction performs a gather load of signed bytes to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign-extended or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset, and 64-bit unscaled offset

32-bit unpacked unscaled offset

(FEAT_SVE)

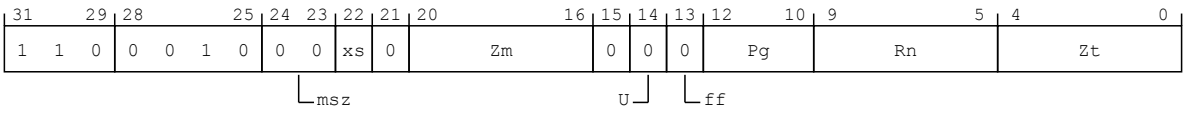

Encoding

```
LD1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

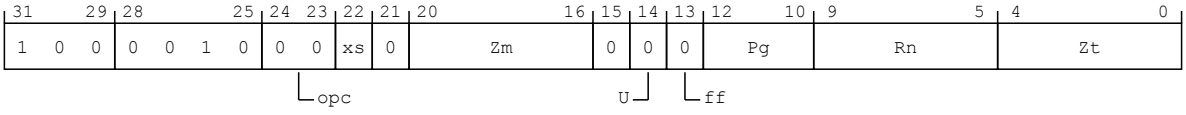

Encoding

```
LD1SB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 0	1 0	Zm	1	0	0	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LD1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.341 LD1SH (vector plus immediate)

Gather load signed halfwords to vector (immediate index)

This instruction performs a gather load of signed halfwords to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1	0 1	imm5	1	0	0	Pg	Zn	Zt
		msz				U	ff			

Encoding

```
LD1SH  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	0 1	imm5	1	0	0	Pg	Zn	Zt
		msz				U	ff			

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.342 LD1SH (scalar plus immediate)

Contiguous load signed halfwords to vector (immediate index)

This instruction performs a contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.343 LD1SH (scalar plus scalar)

Contiguous load signed halfwords to vector (scalar index)

This instruction performs a contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 1	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
```

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 0	!=11111	0 1 0	Pg	Rn	Zt
		dtype	Rm				

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.344 LD1SH (scalar plus vector)

Gather load signed halfwords to vector (vector index)

This instruction performs a gather load of signed halfwords to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

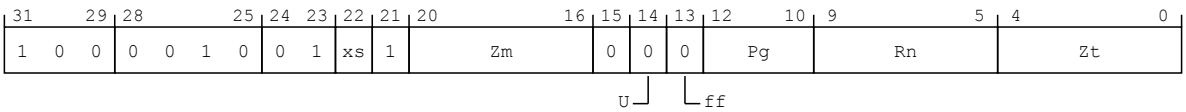

Encoding

```
LD1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked scaled offset

(FEAT_SVE)

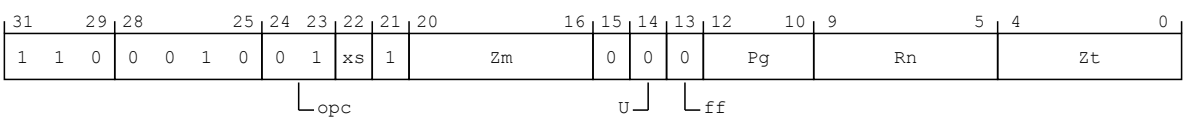

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	xs	0	Zm	0	0	0	Pg	Rn	Zt
		msz					U	ff			

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 1	xs	0	Zm	0	0	0	Pg	Rn	Zt
		opc					U	ff			

Encoding

```
LD1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	1 1	Zm	1	0	0	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 1;
```

64-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	1 0	Zm	1	0	0	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LD1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.345 LD1SW (vector plus immediate)

Gather load signed words to vector (immediate index)

This instruction performs a gather load of signed words to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0 (msz)	0 1	imm5	1	0 (U)	0 (ff)	Pg	Zn	Zt

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.346 LD1SW (scalar plus immediate)

Contiguous load signed words to vector (immediate index)

This instruction performs a contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 0 0 (dtype)	0	imm4	1 0 1	Pg	Rn	Zt

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.347 LD1SW (scalar plus scalar)

Contiguous load signed words to vector (scalar index)

This instruction performs a contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SVE

(FEAT_SVE || FEAT_SME)

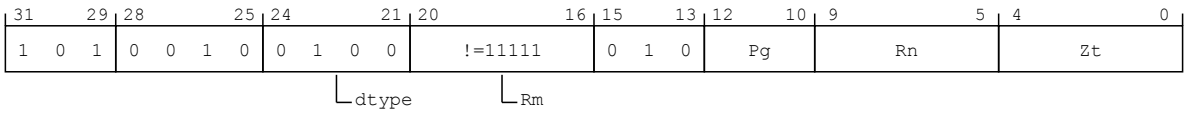

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.348 LD1SW (scalar plus vector)

Gather load signed words to vector (vector index)

This instruction performs a gather load of signed words to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit unpacked scaled offset

(FEAT_SVE)

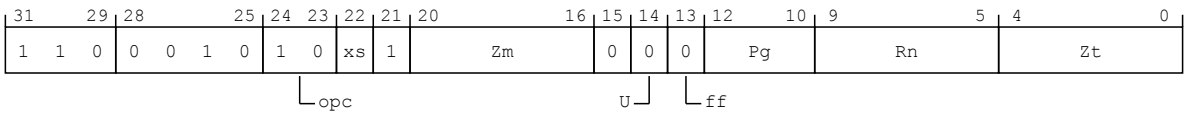

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

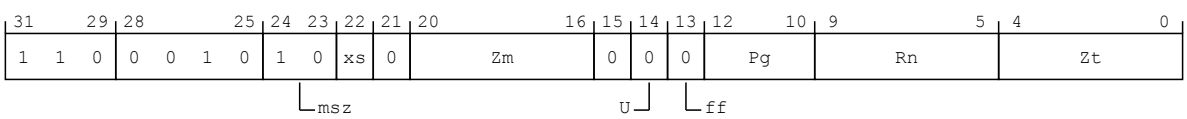

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0	1 1	Zm	1	0	0	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 2;
```

64-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0	1 0	Zm	1	0	0	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LD1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.349 LD1W (scalar plus immediate, consecutive registers)

Contiguous load of words to multiple consecutive vectors (immediate index)

This instruction performs a contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4		1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4			0	1	0	PNg			Rn			Zt			0
																msz											N

Encoding

```
LD1W  { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4		2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4			1	1	0	PNg			Rn			Zt			0	0
																msz												N

Encoding

```
LD1W  { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.350 LD1W (scalar plus scalar, consecutive registers)

Contiguous load of words to multiple consecutive vectors (scalar index)

This instruction performs a contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	0	1	0	PNg	Rn		Zt		0
													msz							N

Encoding

```
LD1W  { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	1	1	0	PNg	Rn		Zt		0	0
													msz								N

Encoding

```
LD1W  { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.351 LD1W (vector plus immediate)

Gather load unsigned words to vector (immediate index)

This instruction performs a gather load of unsigned words to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

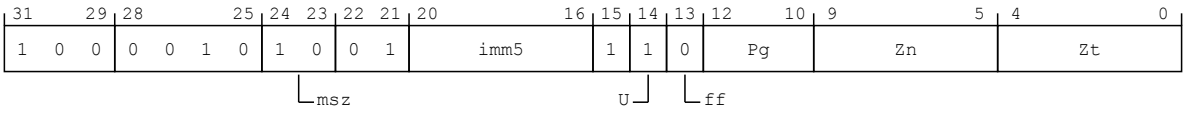

Encoding

```
LD1W  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

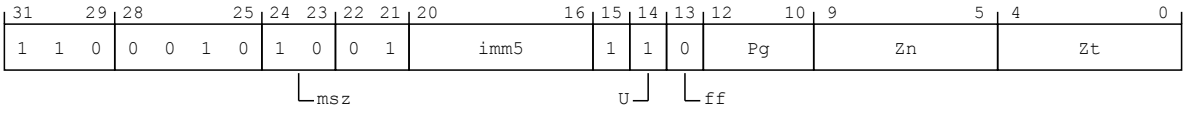

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.352 LD1W (scalar plus immediate, single register)

Contiguous load unsigned words to vector (immediate index)

This instruction performs a contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit element, 64-bit element, and 128-bit element

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 0	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 1	0	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

128-bit element

(FEAT_SVE2p1)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 0	0 0	1	imm4	0 0 1	Pg	Rn	Zt
		dtype							

Encoding

```
LD1W  { <Zt>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask :  bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.353 LD1W (scalar plus scalar, single register)

Contiguous load unsigned words to vector (scalar index)

This instruction performs a contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit element, 64-bit element, and 128-bit element

32-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 0 (dtype)	!=11111 (Rm)	0 1 0	Pg	Rn	Zt

Encoding

```
LD1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 1 (dtype)	!=11111 (Rm)	0 1 0	Pg	Rn	Zt

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

128-bit element

(FEAT_SVE2p1)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 0	0 0	!=11111	1 0 0	Pg	Rn	Zt
		dtype		Rm				

Encoding

```
LD1W  { <Zt>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.354 LD1W (scalar plus vector)

Gather load unsigned words to vector (vector index)

This instruction performs a gather load of unsigned words to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

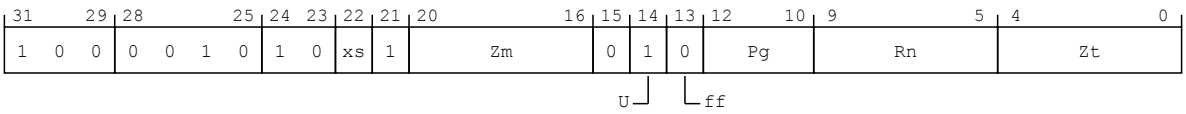

Encoding

```
LD1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked scaled offset

(FEAT_SVE)

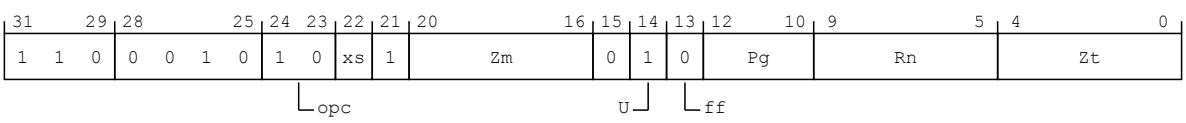

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 0	xs	0	Zm	0	1	0	Pg	Rn	Zt
		msz				U		ff			

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 0 0	0 0 1 0	1 0	xs	0	Zm	0	1	0	Pg	Rn	Zt
		opc				U		ff			

Encoding

```
LD1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

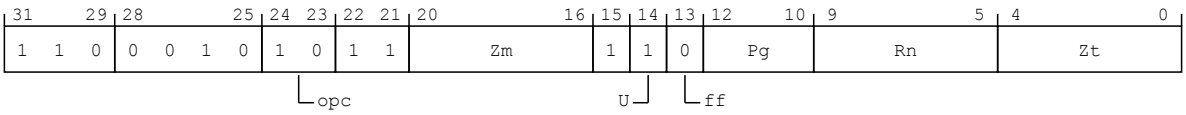

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 2;
```

64-bit unscaled offset

(FEAT_SVE)

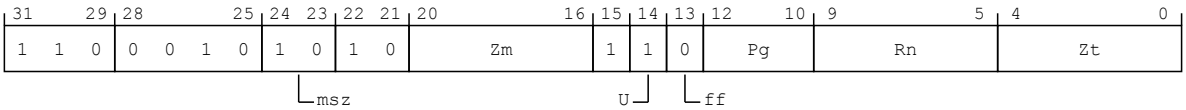

Encoding

```
LD1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.355 LD2B (scalar plus immediate)

Contiguous load two-byte structures to two vectors (immediate index)

This instruction performs a contiguous load of two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

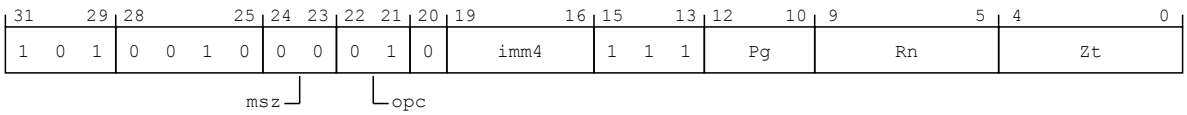

Encoding

```
LD2B  { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.356 LD2B (scalar plus scalar)

Contiguous load two-byte structures to two vectors (scalar index)

This instruction performs a contiguous load of two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

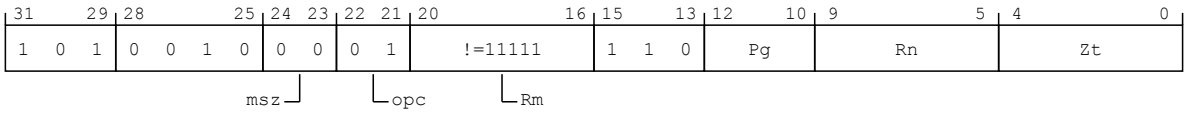

Encoding

```
LD2B  { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.357 LD2D (scalar plus immediate)

Contiguous load two-doubleword structures to two vectors (immediate index)

This instruction performs a contiguous load of two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

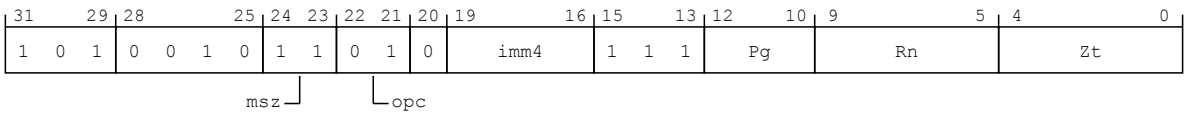

Encoding

```
LD2D  { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.358 LD2D (scalar plus scalar)

Contiguous load two-doubleword structures to two vectors (scalar index)

This instruction performs a contiguous load of two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1	0 1	!=11111	1 1 0	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
LD2D  { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.359 LD2H (scalar plus immediate)

Contiguous load two-halfword structures to two vectors (immediate index)

This instruction performs a contiguous load of two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 (msz)	0 1 (opc)	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LD2H  { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.360 LD2H (scalar plus scalar)

Contiguous load two-halfword structures to two vectors (scalar index)

This instruction performs a contiguous load of two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

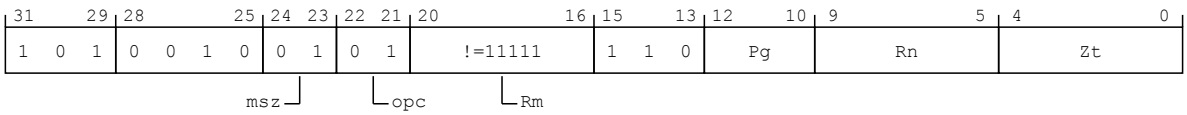

Encoding

```
LD2H  { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.361 LD2Q (scalar plus immediate)

Contiguous load two-quadword structures to two vectors (immediate index)

This instruction performs a contiguous load of two-quadword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1	0 0	1	imm4	1 1 1	Pg	Rn	Zt

num

Encoding

```
LD2Q  { <Zt1>.Q, <Zt2>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.362 LD2Q (scalar plus scalar)

Contiguous load two-quadword structures to two vectors (scalar index)

This instruction performs a contiguous load of two-quadword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1	0 1	!=11111	1 0 0	Pg	Rn	Zt
		num		Rm				

Encoding

```
LD2Q  { <Zt1>.Q, <Zt2>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.363 LD2W (scalar plus immediate)

Contiguous load two-word structures to two vectors (immediate index)

This instruction performs a contiguous load of two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 (msz)	0 1 (opc)	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LD2W  { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.364 LD2W (scalar plus scalar)

Contiguous load two-word structures to two vectors (scalar index)

This instruction performs a contiguous load of two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

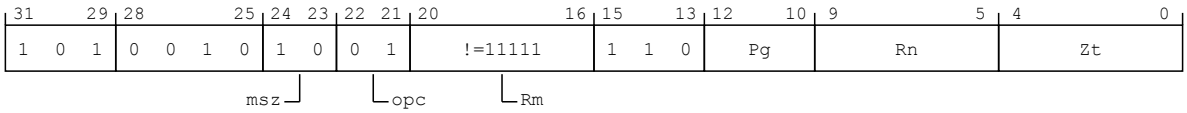

Encoding

```
LD2W  { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.365 LD3B (scalar plus immediate)

Contiguous load three-byte structures to three vectors (immediate index)

This instruction performs a contiguous load of three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

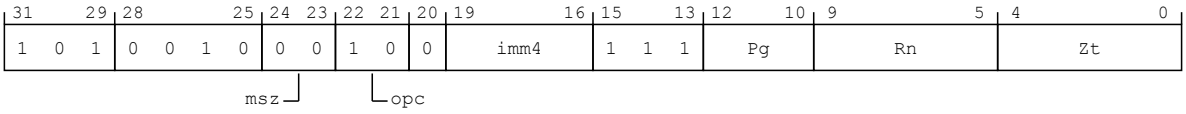

Encoding

```
LD3B  { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.366 LD3B (scalar plus scalar)

Contiguous load three-byte structures to three vectors (scalar index)

This instruction performs a contiguous load of three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

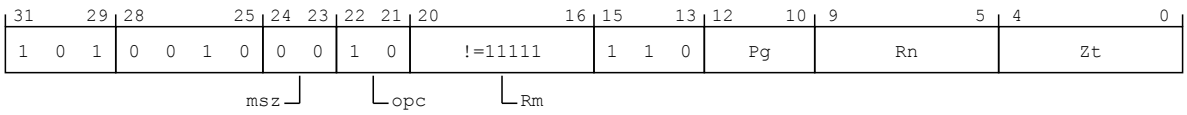

Encoding

```
LD3B  { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.367 LD3D (scalar plus immediate)

Contiguous load three-doubleword structures to three vectors (immediate index)

This instruction performs a contiguous load of three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

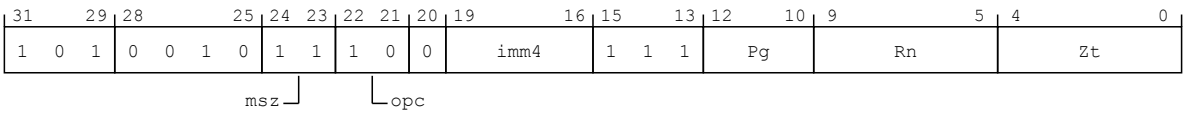

Encoding

```
LD3D  { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.368 LD3D (scalar plus scalar)

Contiguous load three-doubleword structures to three vectors (scalar index)

This instruction performs a contiguous load of three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

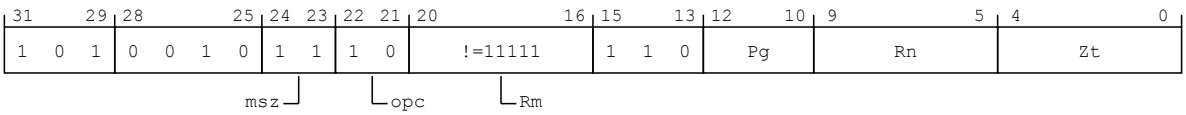

Encoding

```
LD3D  { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.369 LD3H (scalar plus immediate)

Contiguous load three-halfword structures to three vectors (immediate index)

This instruction performs a contiguous load of three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 (msz)	1 0 (opc)	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LD3H  { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.370 LD3H (scalar plus scalar)

Contiguous load three-halfword structures to three vectors (scalar index)

This instruction performs a contiguous load of three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

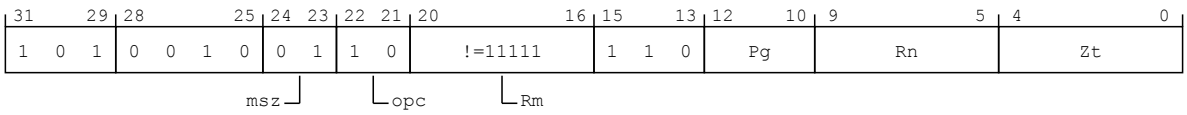

Encoding

```
LD3H  { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.371 LD3Q (scalar plus immediate)

Contiguous load three-quadword structures to three vectors (immediate index)

This instruction performs a contiguous load of three-quadword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 (num)	0 0	1	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LD3Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.372 LD3Q (scalar plus scalar)

Contiguous load three-quadword structures to three vectors (scalar index)

This instruction performs a contiguous load of three-quadword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

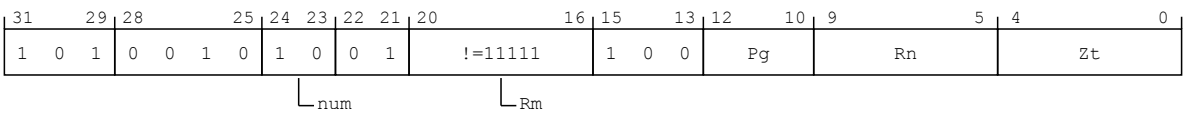

Encoding

```
LD3Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.373 LD3W (scalar plus immediate)

Contiguous load three-word structures to three vectors (immediate index)

This instruction performs a contiguous load of three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

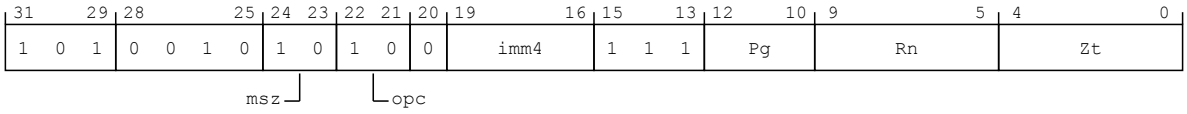

Encoding

```
LD3W  { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.374 LD3W (scalar plus scalar)

Contiguous load three-word structures to three vectors (scalar index)

This instruction performs a contiguous load of three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

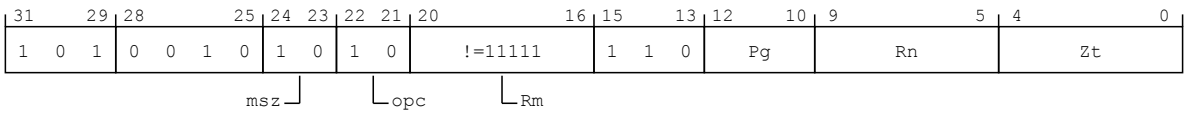

Encoding

```
LD3W  { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.375 LD4B (scalar plus immediate)

Contiguous load four-byte structures to four vectors (immediate index)

This instruction performs a contiguous load of four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

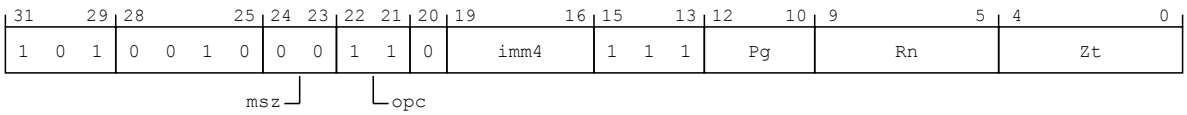

Encoding

```
LD4B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.376 LD4B (scalar plus scalar)

Contiguous load four-byte structures to four vectors (scalar index)

This instruction performs a contiguous load of four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

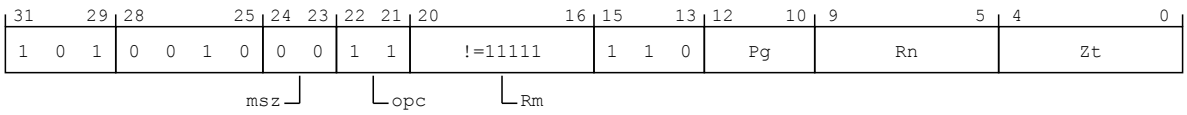

Encoding

```
LD4B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.377 LD4D (scalar plus immediate)

Contiguous load four-doubleword structures to four vectors (immediate index)

This instruction performs a contiguous load of four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

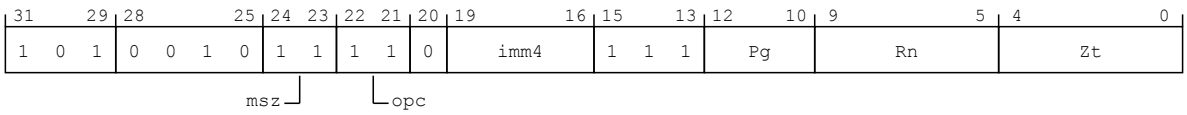

Encoding

```
LD4D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.378 LD4D (scalar plus scalar)

Contiguous load four-doubleword structures to four vectors (scalar index)

This instruction performs a contiguous load of four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

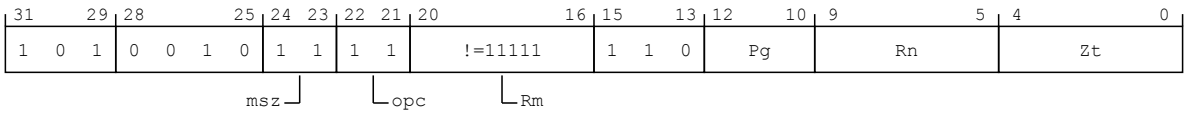

Encoding

```
LD4D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.379 LD4H (scalar plus immediate)

Contiguous load four-halfword structures to four vectors (immediate index)

This instruction performs a contiguous load of four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 (msz)	1 1 (opc)	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LD4H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.380 LD4H (scalar plus scalar)

Contiguous load four-halfword structures to four vectors (scalar index)

This instruction performs a contiguous load of four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

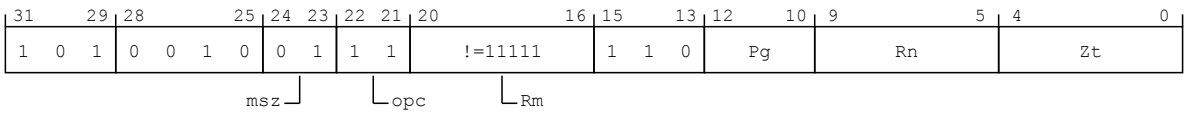

Encoding

```
LD4H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.381 LD4Q (scalar plus immediate)

Contiguous load four-quadword structures to four vectors (immediate index)

This instruction performs a contiguous load of four-quadword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 (num)	0 0	1	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LD4Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.382 LD4Q (scalar plus scalar)

Contiguous load four-quadword structures to four vectors (scalar index)

This instruction performs a contiguous load of four-quadword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1	0 1	!=11111	1 0 0	Pg	Rn	Zt
		num		Rm				

Encoding

```
LD4Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.383 LD4W (scalar plus immediate)

Contiguous load four-word structures to four vectors (immediate index)

This instruction performs a contiguous load of four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

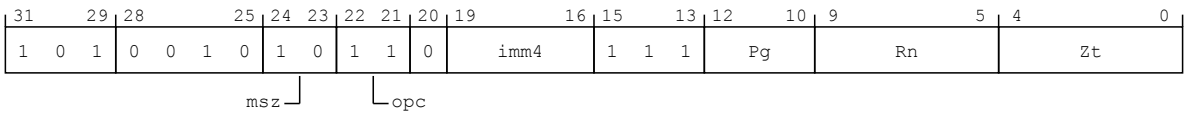

Encoding

```
LD4W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.384 LD4W (scalar plus scalar)

Contiguous load four-word structures to four vectors (scalar index)

This instruction performs a contiguous load of four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory that make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE

(FEAT_SVE || FEAT_SME)

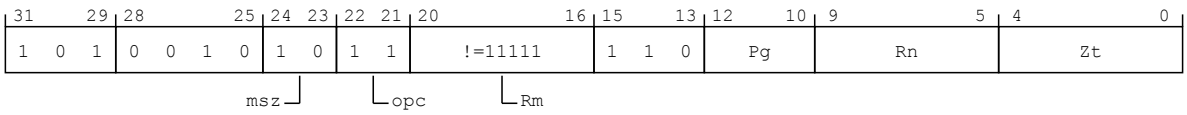

Encoding

```
LD4W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}((t+r) MOD 32) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.385 LDFF1B (vector plus immediate)

Gather load first-fault unsigned bytes to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of unsigned bytes to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 0	0 1	imm5	1	1	1	Pg	Zn	Zt
		msz				U	ff			

Encoding

```
LDFF1B  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 0	0 1	imm5	1	1	1	Pg	Zn	Zt
		msz				U	ff			

Encoding

```
LDFF1B  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.386 LDFF1B (scalar plus scalar)

Contiguous load first-fault unsigned bytes to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element, and 64-bit element

8-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 0 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1B  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

16-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 0 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1B  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

32-bit element

(FEAT_SVE)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1B  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

64-bit element

(FEAT_SVE)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.387 LDFF1B (scalar plus vector)

Gather load first-fault unsigned bytes to vector (vector index)

This instruction performs a gather load with first-faulting behavior of unsigned bytes to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign-extended or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset, and 64-bit unscaled offset

32-bit unpacked unscaled offset

(FEAT_SVE)

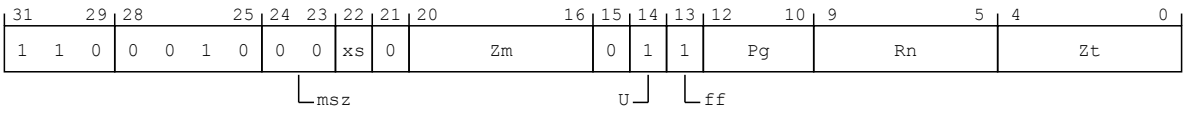

Encoding

```
LDFF1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

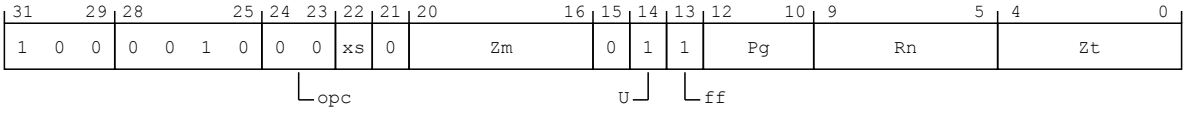

Encoding

```
LDFF1B  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 0	1 0	Zm	1	1	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.388 LDFF1D (vector plus immediate)

Gather load first-fault doublewords to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of doublewords to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

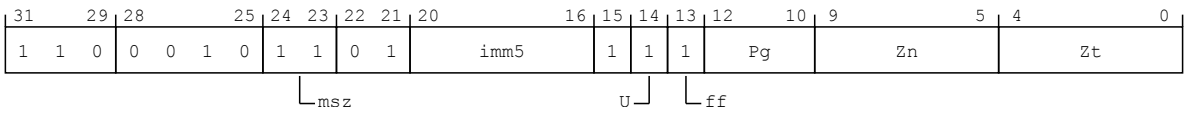

Encoding

```
LDFF1D  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.389 LDFF1D (scalar plus scalar)

Contiguous load first-fault doublewords to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.390 LDFF1D (scalar plus vector)

Gather load first-fault doublewords to vector (vector index)

This instruction performs a gather load with first-faulting behavior of doublewords to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit unpacked scaled offset

(FEAT_SVE)

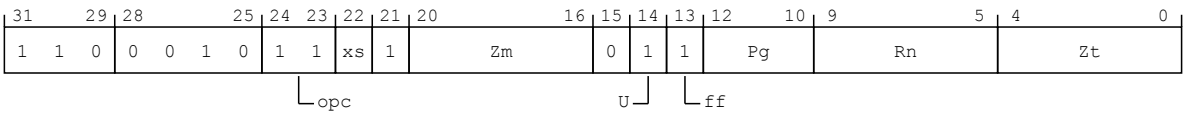

Encoding

```
LDFF1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 3;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

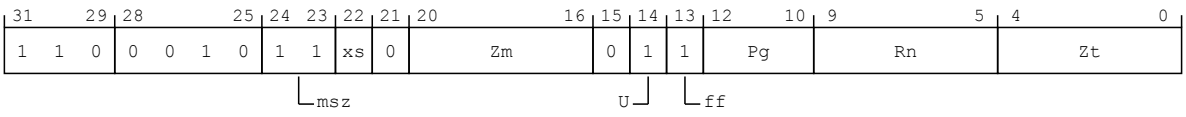

Encoding

```
LDFF1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 1	1 1	Zm	1	1	1	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LDFF1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 3;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 1	1 0	Zm	1	1	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.391 LDFF1H (vector plus immediate)

Gather load first-fault unsigned halfwords to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of unsigned halfwords to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

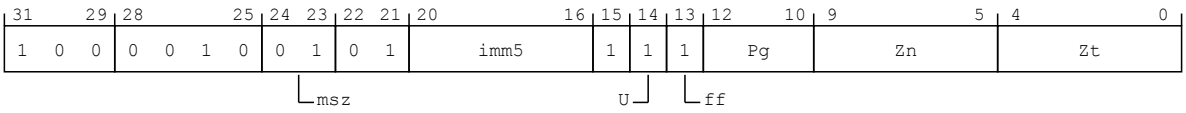

Encoding

```
LDFF1H  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

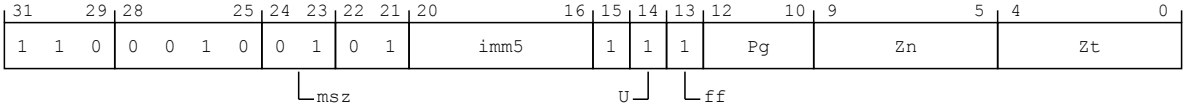

Encoding

```
LDFF1H  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
            end;
        else
            (data, fault) = (Zeros{msize}, FALSE);
        end;

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR(e, esize) = '0';
        end;

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR(e, esize) == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                result[e*:esize] = Extend{esize}(data, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                result[e*:esize] = Zeros{esize};
            else  // merge
                result[e*:esize] = orig[e*:esize];
            end;
        else
            result[e*:esize] = Extend{esize}(data, unsigned);
        end;
    end;

    Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.392 LDFF1H (scalar plus scalar)

Contiguous load first-fault unsigned halfwords to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 0 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1H  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 1 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

64-bit element

(FEAT_SVE)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 1 1 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.393 LDFF1H (scalar plus vector)

Gather load first-fault unsigned halfwords to vector (vector index)

This instruction performs a gather load with first-faulting behavior of unsigned halfwords to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22	21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1	xs	1	Zm	0	1	1	Pg	Rn	Zt
							U	ff			

Encoding

```
LDFF1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22	21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	xs	1	Zm	0	1	1	Pg	Rn	Zt
		opc					U	ff			

Encoding

```
LDFF1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	xs	0	Zm	0	1	1	Pg	Rn	Zt
		msz				U		ff			

Encoding

```
LDFF1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 1	xs	0	Zm	0	1	1	Pg	Rn	Zt
		opc				U		ff			

Encoding

```
LDFF1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	1 1	Zm	1	1	1	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LDFF1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 1;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	1 0	Zm	1	1	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.394 LDFF1SB (vector plus immediate)

Gather load first-fault signed bytes to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of signed bytes to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

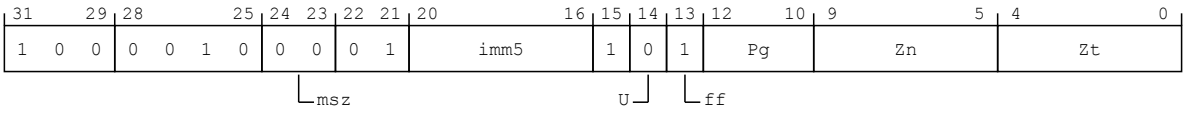

Encoding

```
LDFF1SB  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

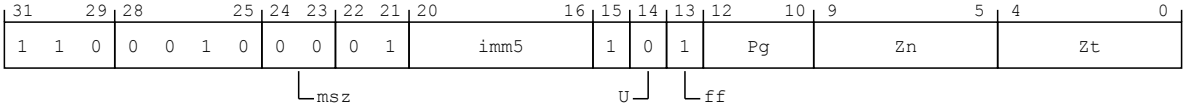

Encoding

```
LDFF1SB  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.395 LDFF1SB (scalar plus scalar)

Contiguous load first-fault signed bytes to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1SB  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 0 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1SB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

64-bit element

(FEAT_SVE)

31 29	28 25	24 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 1 0 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.396 LDFF1SB (scalar plus vector)

Gather load first-fault signed bytes to vector (vector index)

This instruction performs a gather load with first-faulting behavior of signed bytes to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign-extended or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset, and 64-bit unscaled offset

32-bit unpacked unscaled offset

(FEAT_SVE)

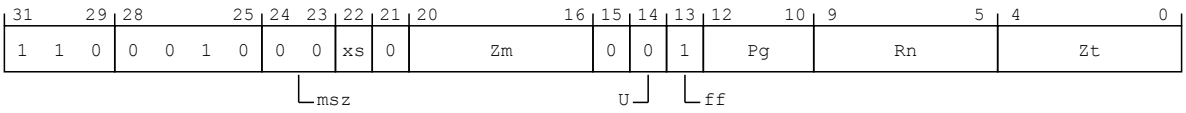

Encoding

```
LDFF1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

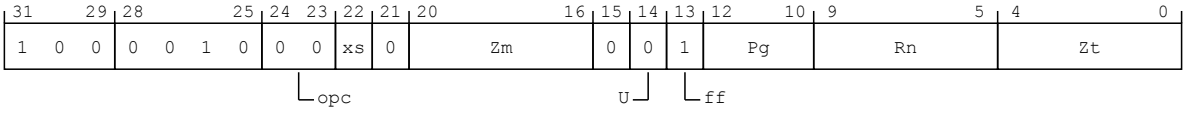

Encoding

```
LDFF1SB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 0	1 0	Zm	1	0	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.397 LDFF1SH (vector plus immediate)

Gather load first-fault signed halfwords to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of signed halfwords to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1 (msz)	0 1	imm5	1	0 (U)	1 (ff)	Pg	Zn	Zt

Encoding

```
LDFF1SH  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1 (msz)	0 1	imm5	1	0 (U)	1 (ff)	Pg	Zn	Zt

Encoding

```
LDFF1SH  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.398 LDFF1SH (scalar plus scalar)

Contiguous load first-fault signed halfwords to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if accdesc.first then
                // Mem[] will not return if a fault is detected for the First active element
                data = Mem{msize}(addr, accdesc);
                accdesc.first = FALSE;
            else
                // MemNF[] will return fault=TRUE if access is not performed for any reason
                (data, fault) = MemNF{msize}(addr, accdesc);
                faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
            end;
        else
            (data, fault) = (Zeros{msize}, FALSE);
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR(e, esize) = '0';
        end;

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR(e, esize) == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                result[e*:esize] = Extend{esize}(data, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                result[e*:esize] = Zeros{esize};
            else  // merge
                result[e*:esize] = orig[e*:esize];
            end;
        else
            result[e*:esize] = Extend{esize}(data, unsigned);
        end;
    end;

    Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.399 LDFF1SH (scalar plus vector)

Gather load first-fault signed halfwords to vector (vector index)

This instruction performs a gather load with first-faulting behavior of signed halfwords to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22	21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1	xs	1	Zm	0	0	1	Pg	Rn	Zt
							U	ff			

Encoding

```
LDFF1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22	21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	xs	1	Zm	0	0	1	Pg	Rn	Zt
		opc					U	ff			

Encoding

```
LDFF1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	0 1	xs	0	Zm	0	0	1	Pg	Rn	Zt
		msz					U		ff		

Encoding

```
LDFF1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14	13	12 10	9 5	4 0
1 0 0	0 0 1 0	0 1	xs	0	Zm	0	0	1	Pg	Rn	Zt
		opc					U		ff		

Encoding

```
LDFF1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	1 1	Zm	1	0	1	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LDFF1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 1;
```

64-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1	1 0	Zm	1	0	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.400 LDFF1SW (vector plus immediate)

Gather load first-fault signed words to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of signed words to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0	0 1	imm5	1	0	1	Pg	Zn	Zt
		msz				U	ff			

Encoding

```
LDFF1SW  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.401 LDFF1SW (scalar plus scalar)

Contiguous load first-fault signed words to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

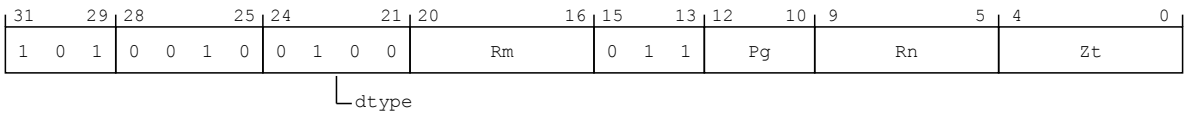

Encoding

```
LDFF1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.402 LDFF1SW (scalar plus vector)

Gather load first-fault signed words to vector (vector index)

This instruction performs a gather load with first-faulting behavior of signed words to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit unpacked scaled offset

(FEAT_SVE)

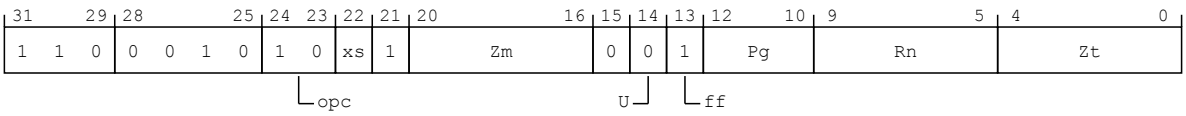

Encoding

```
LDFF1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

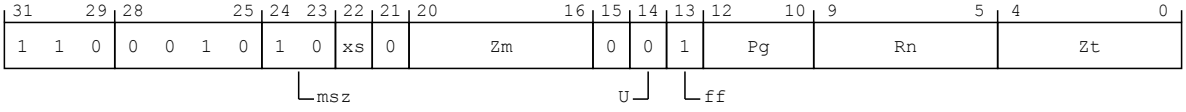

Encoding

```
LDFF1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0	1 1	Zm	1	0	1	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LDFF1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 2;
```

64-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0	1 0	Zm	1	0	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = FALSE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.403 LDFF1W (vector plus immediate)

Gather load first-fault unsigned words to vector (immediate index)

This instruction performs a gather load with first-faulting behavior of unsigned words to Active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 0 (msz)	0 1	imm5	1	1 (U)	1 (ff)	Pg	Zn	Zt

Encoding

```
LDFF1W  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0 (msz)	0 1	imm5	1	1 (U)	1 (ff)	Pg	Zn	Zt

Encoding

```
LDFF1W  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
            end;
        else
            (data, fault) = (Zeros{msize}, FALSE);
        end;

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR(e, esize) = '0';
        end;

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR(e, esize) == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                result[e*:esize] = Extend{esize}(data, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                result[e*:esize] = Zeros{esize};
            else  // merge
                result[e*:esize] = orig[e*:esize];
            end;
        else
            result[e*:esize] = Extend{esize}(data, unsigned);
        end;
    end;

    Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.404 LDFF1W (scalar plus scalar)

Contiguous load first-fault unsigned words to vector (scalar index)

This instruction performs a contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 0	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 1	Rm	0 1 1	Pg	Rn	Zt
		dtype					

Encoding

```
LDFF1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if accdesc.first then
                // Mem[] will not return if a fault is detected for the First active element
                data = Mem{msize}(addr, accdesc);
                accdesc.first = FALSE;
            else
                // MemNF[] will return fault=TRUE if access is not performed for any reason
                (data, fault) = MemNF{msize}(addr, accdesc);
                faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
            end;
        else
            (data, fault) = (Zeros{msize}, FALSE);
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR(e, esize) = '0';
        end;

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR(e, esize) == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                result[e*:esize] = Extend{esize}(data, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                result[e*:esize] = Zeros{esize};
            else  // merge
                result[e*:esize] = orig[e*:esize];
            end;
        else
            result[e*:esize] = Extend{esize}(data, unsigned);
        end;
    end;

    Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.405 LDFF1W (scalar plus vector)

Gather load first-fault unsigned words to vector (vector index)

This instruction performs a gather load with first-faulting behavior of unsigned words to Active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

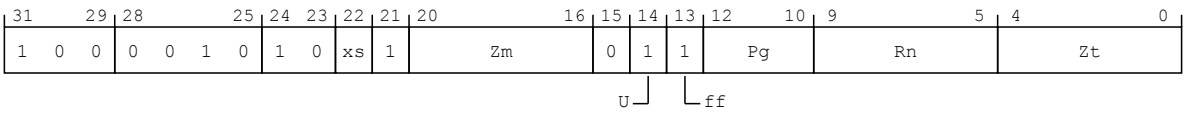

Encoding

```
LDFF1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked scaled offset

(FEAT_SVE)

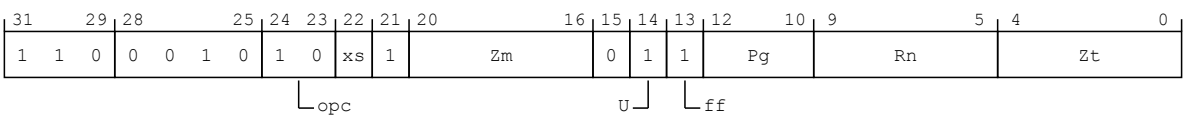

Encoding

```
LDFF1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22	21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	1 0	xs	0	Zm	0	1	1	Pg	Rn	Zt
		msz				U		ff			

Encoding

```
LDFF1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22	21	20–16	15	14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	1 0	xs	0	Zm	0	1	1	Pg	Rn	Zt
		opc				U		ff			

Encoding

```
LDFF1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 0	1 1	Zm	1	1	1	Pg	Rn	Zt
		opc				U	ff			

Encoding

```
LDFF1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 2;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0	0 0 1 0	1 0	1 0	Zm	1	1	1	Pg	Rn	Zt
		msz				U	ff			

Encoding

```
LDFF1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let unsigned : boolean = TRUE;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = FALSE;
let tagchecked : boolean = TRUE;
var accdesc : AccessDescriptor = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

assert accdesc.first;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the First active element
            data = Mem{msize}(addr, accdesc);
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF{msize}(addr, accdesc);
            faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
        end;
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.406 LDNF1B

Contiguous load non-fault unsigned bytes to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element, and 64-bit element

8-bit element

(FEAT_SVE)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 0 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1B  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

16-bit element

(FEAT_SVE)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 0 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1B  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

32-bit element

(FEAT_SVE)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1B  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0 1 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1B  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.407 LDNF1D

Contiguous load non-fault doublewords to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.408 LDNF1H

Contiguous load non-fault unsigned halfwords to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 0 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1H  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 1 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1H  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 1 1 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.409 LDNF1SB

Contiguous load non-fault signed bytes to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 16-bit element, 32-bit element, and 64-bit element

16-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 1 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1SB  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 0 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1SB  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE)

31 29	28 25	24 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 1 0 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1SB  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.410 LDNF1SH

Contiguous load non-fault signed halfwords to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1SH  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 0 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1SH  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (data, fault) = (Zeros{msize}, FALSE);
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR(e, esize) = '0';
        end;

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR(e, esize) == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                result[e*:esize] = Extend{esize}(data, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                result[e*:esize] = Zeros{esize};
            else  // merge
                result[e*:esize] = orig[e*:esize];
            end;
        else
            result[e*:esize] = Extend{esize}(data, unsigned);
        end;
    end;

    Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.411 LDNF1SW

Contiguous load non-fault signed words to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 0 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1SW  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
        (data, fault) = (Zeros{msize}, FALSE);
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR(e, esize) = '0';
    end;

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR(e, esize) == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            result[e*:esize] = Extend{esize}(data, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            result[e*:esize] = Zeros{esize};
        else  // merge
            result[e*:esize] = orig[e*:esize];
        end;
    else
        result[e*:esize] = Extend{esize}(data, unsigned);
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.412 LDNF1W

Contiguous load non-fault unsigned words to vector (immediate index)

This instruction performs a contiguous load with non-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 0	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 1 1	1	imm4	1 0 1	Pg	Rn	Zt
		dtype						

Encoding

```
LDNF1W  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
let orig : bits(VL) = Z{}(t);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
var fault : boolean = FALSE;
var faulted : boolean = FALSE;
var unknown : boolean = FALSE;
let contiguous : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF{msize}(addr, accdesc);
        faulted = faulted || ConstrainUnpredictableBool(Unpredictable_NONFAULT);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (data, fault) = (Zeros{msize}, FALSE);
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR(e, esize) = '0';
        end;

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR(e, esize) == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                result[e*:esize] = Extend{esize}(data, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                result[e*:esize] = Zeros{esize};
            else  // merge
                result[e*:esize] = orig[e*:esize];
            end;
        else
            result[e*:esize] = Extend{esize}(data, unsigned);
        end;
    end;

    Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.413 LDNT1B (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal load of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		0	0	0	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		1	0	0	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1B  { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.414 LDNT1B (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal load of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	0	0	0	PNg	Rn		Zt		1
												msz								N

Encoding

```
LDNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm	1	0	0	PNg	Rn		Zt		0	1
													msz								N

Encoding

```
LDNT1B  { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.415 LDNT1B (vector plus scalar)

Gather load non-temporal unsigned bytes

This instruction performs a non-temporal gather load of unsigned bytes to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

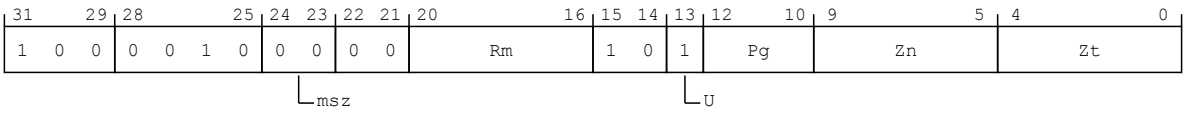

Encoding

```
LDNT1B  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

64-bit unscaled offset

(FEAT_SVE2)

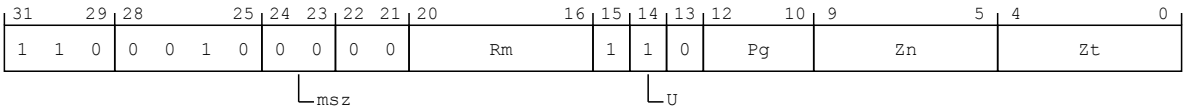

Encoding

```
LDNT1B  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.416 LDNT1B (scalar plus immediate, single register)

Contiguous load non-temporal bytes to vector (immediate index)

This instruction performs a contiguous non-temporal load of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 0 (msz)	0 0	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LDNT1B  { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.417 LDNT1B (scalar plus scalar, single register)

Contiguous load non-temporal bytes to vector (scalar index)

This instruction performs a contiguous non-temporal load of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	0 0	0 0	!=11111	1 1 0	Pg	Rn	Zt
		msz		Rm				

Encoding

```
LDNT1B  { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.418 LDNT1D (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal load of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		0	1	1	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		1	1	1	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.419 LDNT1D (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal load of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	Rm		0	1	1	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	Rm		1	1	1	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.420 LDNT1D (vector plus scalar)

Gather load non-temporal unsigned doublewords

This instruction performs a non-temporal gather load of doublewords to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2)

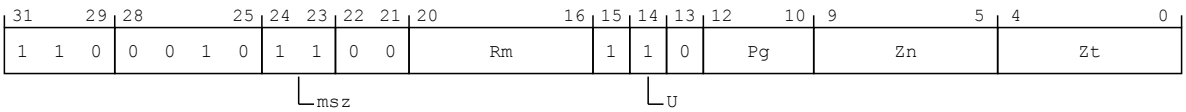

Encoding

```
LDNT1D  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.421 LDNT1D (scalar plus immediate, single register)

Contiguous load non-temporal doublewords to vector (immediate index)

This instruction performs a contiguous non-temporal load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 (msz)	0 0	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LDNT1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.422 LDNT1D (scalar plus scalar, single register)

Contiguous load non-temporal doublewords to vector (scalar index)

This instruction performs a contiguous non-temporal load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 1 (msz)	0 0	!=11111 (Rm)	1 1 0	Pg	Rn	Zt

Encoding

```
LDNT1D  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.423 LDNT1H (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal load of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0 0		0	0	1 0 0		imm4		0	0	1	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0 0		0	0	1 0 0		imm4		1	0	1	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.424 LDNT1H (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal load of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	Rm		0	0	1	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	Rm		1	0	1	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.425 LDNT1H (vector plus scalar)

Gather load non-temporal unsigned halfwords

This instruction performs a non-temporal gather load of unsigned halfwords to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

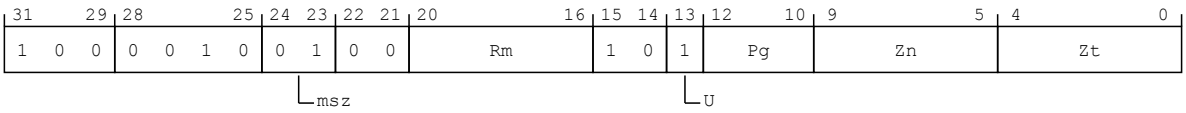

Encoding

```
LDNT1H  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

64-bit unscaled offset

(FEAT_SVE2)

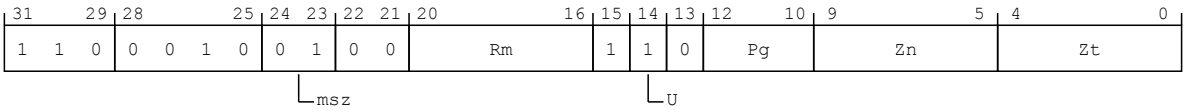

Encoding

```
LDNT1H  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.426 LDNT1H (scalar plus immediate, single register)

Contiguous load non-temporal halfwords to vector (immediate index)

This instruction performs a contiguous non-temporal load of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 (msz)	0 0	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
LDNT1H  { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.427 LDNT1H (scalar plus scalar, single register)

Contiguous load non-temporal halfwords to vector (scalar index)

This instruction performs a contiguous non-temporal load of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	0 1 (msz)	0 0	!=11111 (Rm)	1 1 0	Pg	Rn	Zt

Encoding

```
LDNT1H  { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.428 LDNT1SB

Gather load non-temporal signed bytes

This instruction performs a non-temporal gather load of signed bytes to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

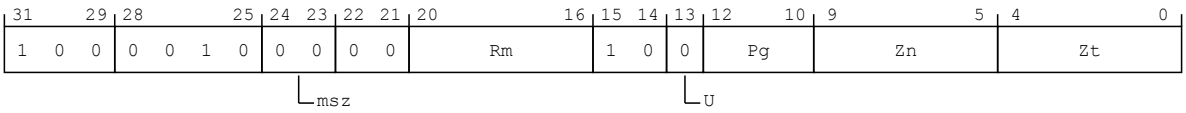

Encoding

```
LDNT1SB  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

64-bit unscaled offset

(FEAT_SVE2)

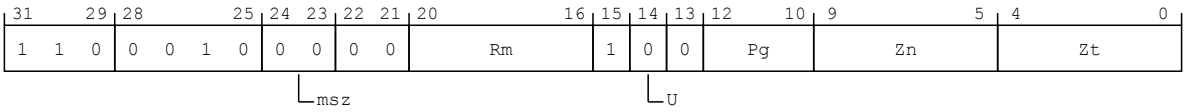

Encoding

```
LDNT1SB  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.429 LDNT1SH

Gather load non-temporal signed halfwords

This instruction performs a non-temporal gather load of signed halfwords to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

31–29	28–25	24–23	22–21	20–16	15–14	13	12–10	9–5	4–0
1 0 0	0 0 1 0	0 1 (msz)	0 0	Rm	1 0	0 (U)	Pg	Zn	Zt

Encoding

```
LDNT1SH  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
```

64-bit unscaled offset

(FEAT_SVE2)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0	0 0 1 0	0 1 (msz)	0 0	Rm	1	0 (U)	0	Pg	Zn	Zt

Encoding

```
LDNT1SH  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.430 LDNT1SW

Gather load non-temporal signed words

This instruction performs a non-temporal gather load of signed words to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2)

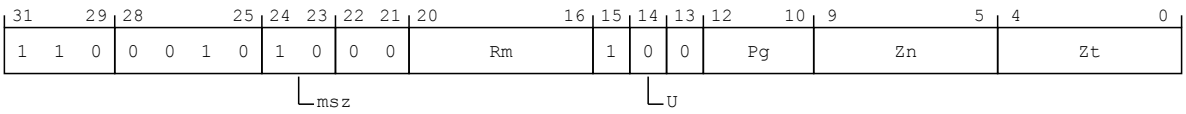

Encoding

```
LDNT1SW  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.431 LDNT1W (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal load of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		0	1	0	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 0	0	imm4		1	1	0	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1W  { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.432 LDNT1W (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal load of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	Rm		0	1	0	PNg		Rn		Zt		1
												msz								N

Encoding

```
LDNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	Rm		1	1	0	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
LDNT1W  { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer+r) = values[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.433 LDNT1W (vector plus scalar)

Gather load non-temporal unsigned words

This instruction performs a non-temporal gather load of unsigned words to Active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

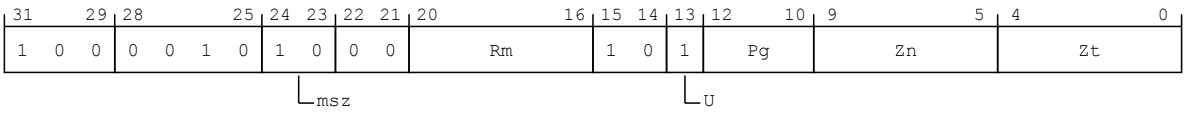

Encoding

```
LDNT1W  { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

64-bit unscaled offset

(FEAT_SVE2)

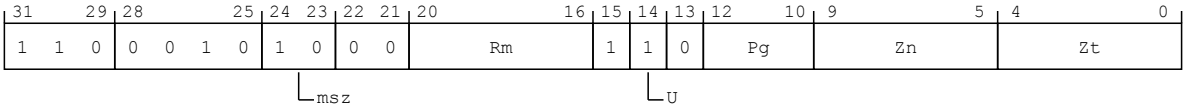

Encoding

```
LDNT1W  { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var result : bits(VL);
var data : bits(msize);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        data = Mem{msize}(addr, accdesc);
        result[e*:esize] = Extend{esize}(data, unsigned);
    else
        result[e*:esize] = Zeros{esize};
    end;
end;

Z{VL}(t) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.434 LDNT1W (scalar plus immediate, single register)

Contiguous load non-temporal words to vector (immediate index)

This instruction performs a contiguous non-temporal load of words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 0 1	0 0 1 0	1 0	0 0	0	imm4	1 1 1	Pg	Rn	Zt
		msz							

Encoding

```
LDNT1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.435 LDNT1W (scalar plus scalar, single register)

Contiguous load non-temporal words to vector (scalar index)

This instruction performs a contiguous non-temporal load of words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 0 1	0 0 1 0	1 0 (msz)	0 0	!=11111 (Rm)	1 1 0	Pg	Rn	Zt

Encoding

```
LDNT1W  { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.436 LDR (predicate)

Load predicate register

This instruction loads a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 that is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.

For programmer convenience, an assembler must also accept a predicate-as-counter register name for the destination predicate register.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15 13	12 10	9 5	4	3 0
1 0 0	0 0 1 0	1 1	0	imm9h	0 0 0	imm9l	Rn	0	Pt

Encoding

```
LDR  <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Pt);
let n : integer = UInt(Rn);
let imm : integer = SInt(imm9h::imm9l);
```

Assembler Symbols

<Pt>

Is the name of the destination scalable predicate register, encoded in the “Pt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the “imm9h:imm9l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = PL DIV 8;
var base : bits(64);
let offset : integer = imm * elements;
var result : bits(PL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP{64}();
else
    base = X{64}(n);
end;

var addr : bits(64) = AddressAdd(base, offset, accdesc);

let aligned : boolean = IsAlignedSize(base, 2);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;

for e = 0 to elements-1 do
    result[e*:8] = AArch64_MemSingle{8}(addr, accdesc, aligned);
    addr = AddressIncrement(addr, 1, accdesc);
end;

P{PL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.437 LDR (vector)

Load vector register

This instruction loads a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 that is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22	21 16	15 13	12 10	9 5	4 0
1 0 0	0 0 1 0	1 1	0	imm9h	0 1 0	imm9l	Rn	Zt

Encoding

```
LDR  <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let imm : integer = SInt(imm9h::imm9l);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the “imm9h:imm9l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 8;
var base : bits(64);
let offset : integer = imm * elements;
var result : bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = FALSE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  predicated, tagchecked);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if n == 31 then
    CheckSPAlignment();
    base = SP{64}();
else
    base = X{64}(n);
end;

var addr : bits(64) = AddressAdd(base, offset, accdesc);

let aligned : boolean = IsAlignedSize(base, 16);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;

for e = 0 to elements-1 do
    result[e*:8] = AArch64_MemSingle{8}(addr, accdesc, aligned);
    addr = AddressIncrement(addr, 1, accdesc);
end;

Z{VL}(t) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.438 LSL (immediate, predicated)

Logical shift left by immediate (predicated)

This instruction shifts left by immediate each active element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

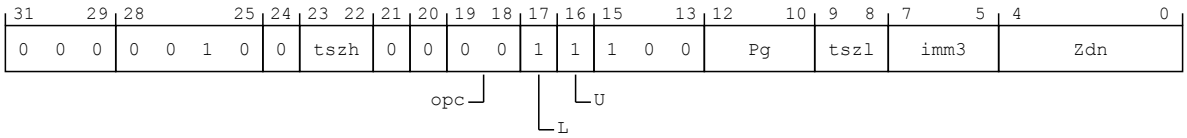

Encoding

```
LSL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let PL : integer{} = VL DIV 8;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = LSL(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.439 LSL (wide elements, predicated)

Logical shift left by 64-bit wide elements (predicated)

This instruction shifts left active elements of the first source vector by the corresponding overlapping 64-bit elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

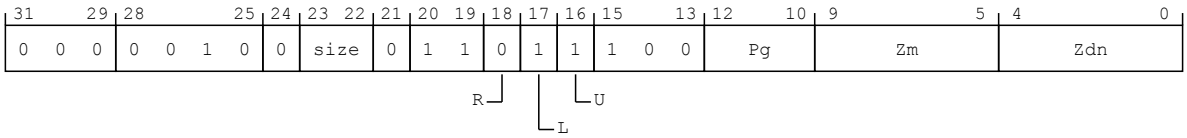

Encoding

```
LSL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
        let shift : integer = Min(UInt(element2), esize);
        result[e*:esize] = LSL(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.440 LSL (vectors)

Logical shift left by vector (predicated)

This instruction shifts left active elements of the first source vector by the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

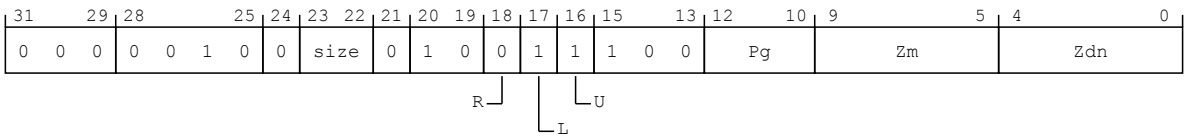

Encoding

```
LSL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let shift : integer = Min(UInt(element2), esize);
        result[e*:esize] = LSL(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.441 LSL (immediate, unpredicated)

Logical shift left by immediate (unpredicated)

This instruction shifts left by immediate each element of the source vector, and places the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	tszh	1	tszl	imm3	1 0 0	1	1 1 (opc)	Zn	Zd

Encoding

```
LSL  <Zd>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    result[e*:esize] = LSL(element1, shift);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.442 LSL (wide elements, unpredicated)

Logical shift left by 64-bit wide elements (unpredicated)

This instruction shifts left all elements of the first source vector by the corresponding overlapping 64-bit elements of the second source vector and places the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

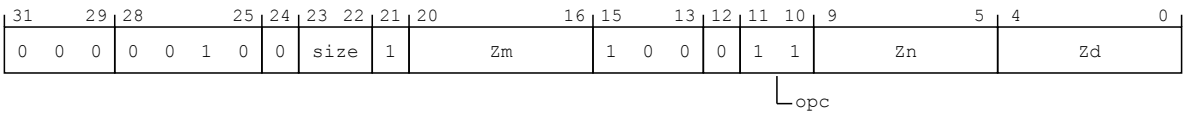

Encoding

```
LSL  <Zd>.<T>, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
    let shift : integer = Min(UInt(element2), esize);
    result[e*:esize] = LSL(element1, shift);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.443 LSLR

Reversed logical shift left by vector (predicated)

This instruction reverse shifts left active elements of the second source vector by the corresponding elements of the first source vector, and destructively places the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

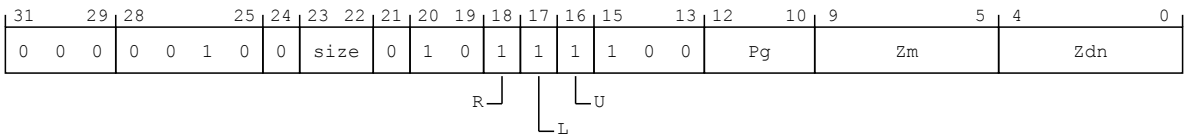

Encoding

```
LSLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let shift : integer = Min(UInt(element1), esize);
        result[e*:esize] = LSL(element2, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.444 LSR (immediate, predicated)

Logical shift right by immediate (predicated)

This instruction shifts right by immediate, inserting zeroes, each active element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

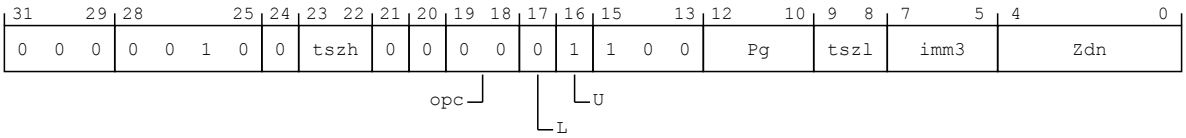

Encoding

```
LSR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let PL : integer{} = VL DIV 8;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = LSR(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.445 LSR (wide elements, predicated)

Logical shift right by 64-bit wide elements (predicated)

This instruction shifts right, inserting zeroes, active elements of the first source vector by the corresponding overlapping 64-bit elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

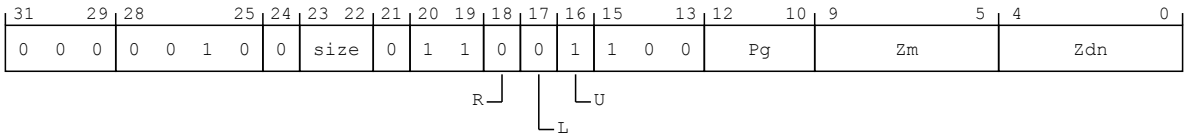

Encoding

```
LSR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
        let shift : integer = Min(UInt(element2), esize);
        result[e*:esize] = LSR(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.446 LSR (vectors)

Logical shift right by vector (predicated)

This instruction shifts right, inserting zeroes, active elements of the first source vector by the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

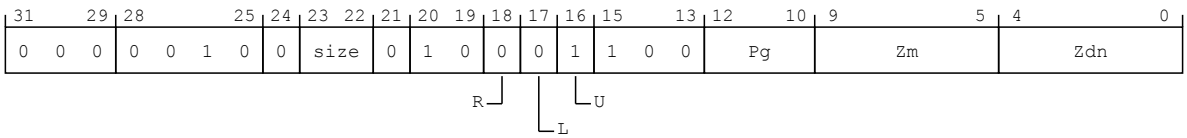

Encoding

```
LSR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let shift : integer = Min(UInt(element2), esize);
        result[e*:esize] = LSR(element1, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.447 LSR (immediate, unpredicated)

Logical shift right by immediate (unpredicated)

This instruction shifts right by immediate, inserting zeroes, each element of the source vector, and places the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	tszh	1	tszl	imm3	1 0 0	1	0	1	Zn	Zd
										U		

Encoding

```
LSR  <Zd>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    result[e*:esize] = LSR(element1, shift);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.448 LSR (wide elements, unpredicated)

Logical shift right by 64-bit wide elements (unpredicated)

This instruction shifts right, inserting zeroes, all elements of the first source vector by the corresponding overlapping 64-bit elements of the second source vector and places the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	size	1	Zm	1 0 0	0	0	1	Zn	Zd
									U		

Encoding

```
LSR  <Zd>.<T>, <Zn>.<T>, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(64) = operand2[((e * esize) DIVRM 64)*:64];
    let shift : integer = Min(UInt(element2), esize);
    result[e*:esize] = LSR(element1, shift);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.449 LSRR

Reversed logical shift right by vector (predicated)

This instruction reverse shifts right, inserting zeroes, active elements of the second source vector by the corresponding elements of the first source vector, and destructively places the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

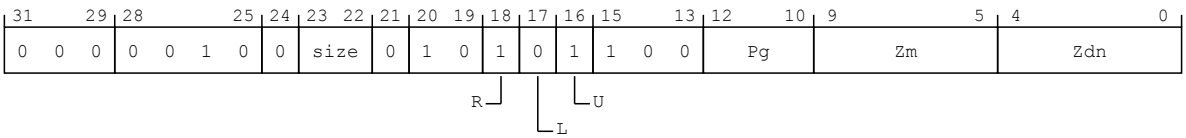

Encoding

```
LSRR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let shift : integer = Min(UInt(element1), esize);
        result[e*:esize] = LSR(element2, shift);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.450 LUTI2 (8-bit and 16-bit)

Lookup table read with 2-bit indices (8-bit and 16-bit)

This instruction copies indexed 8-bit or 16-bit elements from the low 128 bits of the table vector to the destination vector using packed 2-bit indices from a segment of the source vector. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index. This instruction is unpredicated.

It has encodings from 2 classes: Byte and Halfword

Byte

((FEAT_SVE2 || FEAT_SME2) && FEAT_LUT)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 0	0 0 1 0	1	i2	1	Zm	1 0 1	1 0 0	Zn	Zd

Encoding

```
LUTI2  <Zd>.B, { <Zn>.B }, <Zm>[<index>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_LUT)) then EndOfDecode(Decode_UNDEF);
end;
let isize : integer{} = 2;
let esize : integer{} = 8;
let m : integer = UInt(Zm);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let part : integer = UInt(i2);
```

Halfword

((FEAT_SVE2 || FEAT_SME2) && FEAT_LUT)

31–29	28–25	24	23–22	21	20–16	15–13	12	11–10	9–5	4–0
0 1 0	0 0 1 0	1	i3h	1	Zm	1 0 1	i3l	1 0	Zn	Zd

Encoding

```
LUTI2  <Zd>.H, { <Zn>.H }, <Zm>[<index>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_LUT)) then EndOfDecode(Decode_UNDEF);
end;
let isize : integer{} = 2;
let esize : integer{} = 16;
let m : integer = UInt(Zm);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let part : integer = UInt(i3h::i3l);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the table vector register, encoded in the “Zn” field.

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

<index>

For the “Byte” variant: is the vector segment index, in the range 0 to 3, encoded in the “i2” field.

For the “Halfword” variant: is the vector segment index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let ibase : integer = elements * part;
let indexes : bits(VL) = Z{}(m);
let table : bits(VL)   = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let index : integer = UInt(indexes[(ibase + e)*:isize]);
    result[e*:esize] = table[index*:esize];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.451 LUTI4 (8-bit and 16-bit)

Lookup table read with 4-bit indices (8-bit and 16-bit)

This instruction copies indexed 8-bit or 16-bit elements from the low 128 or 256 bits of the table vector, or from the low 128 bits of the two table vectors to the destination vector using packed 4-bit indices from a segment of the source vector. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index. This instruction is unpredicated.

It has encodings from 3 classes: Byte, single register table, Halfword, two register table, and Halfword, single register table

Byte, single register table

((FEAT_SVE2 || FEAT_SME2) && FEAT_LUT)

31–29	28–25	24	23	22	21	20–16	15–13	12–10	9–5	4–0
0 1 0	0 0 1 0	1	i1	1	1	Zm	1 0 1	0 0 1	Zn	Zd

Encoding

```
LUTI4  <Zd>.B, { <Zn>.B }, <Zm>[<index>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_LUT)) then EndOfDecode(Decode_UNDEF);
end;
let isize : integer{} = 4;
let esize : integer{} = 8;
let ntblr : integer = 1;
let m : integer = UInt(Zm);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let part : integer = UInt(i1);
```

Halfword, two register table

((FEAT_SVE2 || FEAT_SME2) && FEAT_LUT)

31–29	28–25	24	23 22	21	20–16	15–13	12	11	10	9–5	4–0
0 1 0	0 0 1 0	1	i2	1	Zm	1 0 1	1	0	1	Zn	Zd
								op			

Encoding

```
LUTI4  <Zd>.H, { <Zn1>.H, <Zn2>.H }, <Zm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_LUT)) then EndOfDecode(Decode_UNDEF);
end;
let isize : integer{} = 4;
let esize : integer{} = 16;
let ntblr : integer = 2;
let m : integer = UInt(Zm);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let part : integer = UInt(i2);
```

Halfword, single register table

((FEAT_SVE2 || FEAT_SME2) && FEAT_LUT)

31 29	28 25	24	23 22	21	20 16	15 13	12	11	10	9 5	4 0
0 1 0	0 0 1 0	1	i2	1	Zm	1 0 1	1	1	1	Zn	Zd
								op			

Encoding

```
LUTI4  <Zd>.H, { <Zn>.H }, <Zm>[<index>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME2)) ||
      !IsFeatureImplemented(FEAT_LUT)) then EndOfDecode(Decode_UNDEF);
end;
if MaxImplementedAnyVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let isize : integer{} = 4;
let esize : integer{} = 16;
let ntblr : integer = 1;
let m : integer = UInt(Zm);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let part : integer = UInt(i2);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the table vector register, encoded in the “Zn” field.

<Zm>

Is the name of the source scalable vector register, encoded in the “Zm” field.

<index>

For the “Byte, single register table” variant: is the vector segment index, in the range 0 to 1, encoded in the “i1” field.

For the “Halfword, single register table” and “Halfword, two register table” variants: is the vector segment index, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first table vector register, encoded as “Zn”.

<Zn2>

Is the name of the second table vector register, encoded as “Zn” plus 1 modulo 32.

Operation

```
if IsFeatureImplemented(FEAT_SME2) then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
if ntblr == 1 && esize == 16 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let elements : integer = VL DIV esize;
let tablesize : integer = if ntblr == 1 && esize == 16 then 256 else 128;
let eltspertable : integer = tablesize DIV esize;
let ibase : integer = elements * part;
let indexes : bits(VL) = Z{}(m);
let table1 : bits(VL)  = Z{}(n+0);
let table2 : bits(VL)  = if ntblr == 2 then Z{VL}((n+1) MOD 32) else Zeros{VL};
var result : bits(VL);
var res : bits(esize);

for e = 0 to elements-1 do
    let index : integer = UInt(indexes[(ibase + e)*:isize]);
    if index < eltspertable then
        res = table1[index*:esize];
    else
        assert ntblr == 2;
        res = table2[(index - eltspertable)*:esize];
    end;
    result[e*:esize] = res;

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.452 MAD

Multiply-add to multiplicand (predicated)

This instruction multiplies the active elements of the first source vector by the corresponding elements of the second source vector, and adds the result to elements of the third (addend) vector. The results are destructively placed in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	Zm	1	1	0	Pg	Za	Zdn
								op			

Encoding

```
MAD  <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let a : integer = UInt(Za);
let sub_op : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand3 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(a) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : integer = UInt(operand1[e*:esize]);
        let element2 : integer = UInt(operand2[e*:esize]);
        let product : integer = element1 * element2;
        if sub_op then
            result[e*:esize] = operand3[e*:esize] - product;
        else
            result[e*:esize] = operand3[e*:esize] + product;
        end;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.453 MADPT

Multiply-add checked pointer vectors to multiplicand

This instruction multiplies, with overflow check, the elements of the first and second source vectors and adds the results of multiplication, with pointer check, to elements of the third (addend) vector. The final results are destructively placed in the destination and first source (multiplicand) vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE && FEAT_CPA)

31–29	28–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (opc)	0	Zm	1 1 0 1	1 (o2)	0	Za	Zdn

Encoding

```
MADPT  <Zdn>.D, <Zm>.D, <Za>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_CPA) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn);
let m : integer  = UInt(Zm);
let a : integer  = UInt(Za);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(a);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:64]);
    let element2 : integer = SInt(operand2[e*:64]);
    let product : integer = element1 * element2;
    let overflow : boolean = (product != SInt(product[63:0]));
    let addend : bits(64) = operand3[e*:64];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:64] = PointerMultiplyAddCheck(addend + product, addend, overflow);
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.454 MATCH

Detect any matching elements, setting the condition flags

This instruction compares each active 8-bit or 16-bit element in the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector. For each element in the first source vector, if any matching element is found in the corresponding segment of the second source vector, the corresponding element of the destination predicate is set to true, otherwise it is set to false. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

Byte and halfword

(FEAT_SVE2)

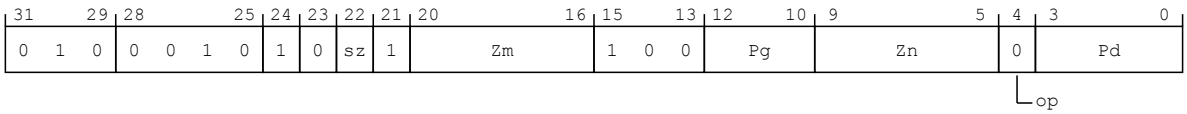

Encoding

```
MATCH  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(sz);
let g : integer = UInt(Pg);
let d : integer = UInt(Pd);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let segmentbase : integer = e - (e MOD eltspersegment);
        result[e*:psize] = ZeroExtend{psize}('0');
        let element1 : bits(esize) = operand1[e*:esize];
        for i = segmentbase to (segmentbase + eltspersegment) - 1 do
            let element2 : bits(esize) = operand2[i*:esize];
            if element1 == element2 then
                result[e*:psize] = ZeroExtend{psize}('1');
            end;
        end;
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.455 MLA (vectors)

Multiply-add (predicated)

This instruction multiplies the active elements of the first source vector by the corresponding elements of the second source vector, and adds the result to elements of the third source (addend) vector. The results are destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	Zm	0	1	0	Pg	Zn	Zda

op

Encoding

```
MLA  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sub_op : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : integer = UInt(operand1[e*:esize]);
        let element2 : integer = UInt(operand2[e*:esize]);
        let product : integer = element1 * element2;
        if sub_op then
            result[e*:esize] = operand3[e*:esize] - product;
        else
            result[e*:esize] = operand3[e*:esize] + product;
        end;
    else
        result[e*:esize] = operand3[e*:esize];
    end;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.456 MLA (indexed)

Multiply-add by indexed element

This instruction multiplies all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment. This instruction is unpredicated.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 0 0 1	0	Zn	Zda
									S		

Encoding

```
MLA  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 0 1	0	Zn	Zda
			size					S		

Encoding

```
MLA  <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (size)	1	i1	Zm	0 0 0 0 1	0 (S)	Zn	Zda

Encoding

```
MLA  <Zda>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[s*:esize]);
    let product : bits(esize) = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.457 MLAPT

Multiply-add checked pointer vectors

This instruction multiplies, with overflow check, the elements of the first and second source vectors and adds the results of multiplication, with pointer check, to elements of the third source (addend) vector. The final results are destructively placed in the destination and third source (addend) vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE && FEAT_CPA)

31–29	28–25	24	23–22	21	20–16	15–12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (opc)	0	Zm	1 1 0 1	0 (o2)	0	Zn	Zda

Encoding

```
MLAPT  <Zda>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_CPA) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:64]);
    let element2 : integer = SInt(operand2[e*:64]);
    let product : integer = element1 * element2;
    let overflow : boolean = (product != SInt(product[63:0]));
    let addend : bits(64) = operand3[e*:64];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:64] = PointerMultiplyAddCheck(addend + product, addend, overflow);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.458 MLS (vectors)

Multiply-subtract (predicated)

This instruction multiplies the active elements of the first source vector by the corresponding elements of the second source vector, and subtracts the result from elements of the third source (addend) vector. The results are destructively placed in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

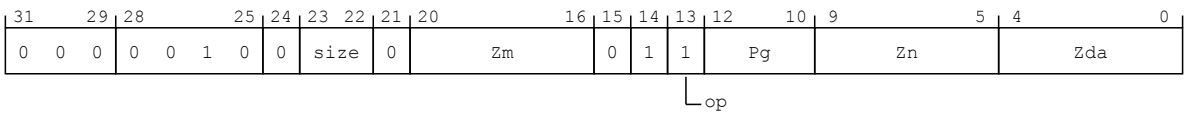

Encoding

```
MLS  <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sub_op : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : integer = UInt(operand1[e*:esize]);
        let element2 : integer = UInt(operand2[e*:esize]);
        let product : integer = element1 * element2;
        if sub_op then
            result[e*:esize] = operand3[e*:esize] - product;
        else
            result[e*:esize] = operand3[e*:esize] + product;
        end;
    else
        result[e*:esize] = operand3[e*:esize];
    end;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.459 MLS (indexed)

Multiply-subtract by indexed element

This instruction multiplies all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment. This instruction is unpredicated.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 0 0 1	1	Zn	Zda
									S		

Encoding

```
MLS  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 0 1	1	Zn	Zda
			size					S		

Encoding

```
MLS  <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (size)	1	i1	Zm	0 0 0 0 1	1 (S)	Zn	Zda

Encoding

```
MLS  <Zda>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[s*:esize]);
    let product : bits(esize) = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.460 MOV (predicate, zeroing)

Move predicates (zeroing)

This instruction reads Active elements from the source predicate and places those elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This is an alias of AND (predicates). This means:

- The encodings in this description are named to match the encodings of AND (predicates).
- The description of AND (predicates) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

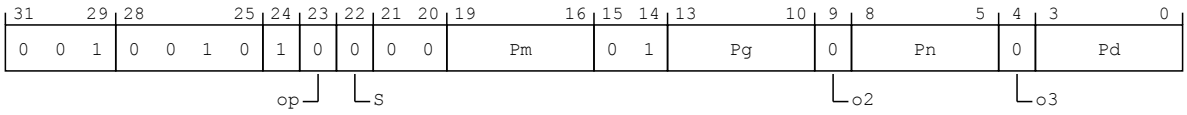

Encoding

```
MOV  <Pd>.B, <Pg>/Z, <Pn>.B
```

is equivalent to

```
AND  <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B
```

and is the preferred disassembly when `S == '0' && Pn == Pm`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of AND (predicates) gives the operational pseudocode for this instruction.

Operational Information

The description of AND (predicates) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.461 MOV (immediate, zeroing)

Move signed integer immediate to vector elements (zeroing)

This instruction moves a signed integer immediate to each Active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

This is an alias of CPY (immediate, zeroing). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, zeroing).
- The description of CPY (immediate, zeroing) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 5	4 0
0 0 0	0 0 1 0	1	size	0 1	Pg	0	0	sh	imm8	Zd
							M			

Encoding

```
MOV  <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}
```

is equivalent to

```
CPY  <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.462 MOV (immediate, merging)

Move signed integer immediate to vector elements (merging)

This instruction moves a signed integer immediate to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

This is an alias of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The description of CPY (immediate, merging) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 5	4 0
0 0 0	0 0 1 0	1	size	0 1	Pg	0	1	sh	imm8	Zd
							M			

Encoding

```
MOV  <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}
```

is equivalent to

```
CPY  <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational Information

The description of CPY (immediate, merging) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.463 MOV (scalar, predicated)

Move general-purpose register to vector elements (predicated)

This instruction moves the general-purpose scalar source register to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (scalar). This means:

- The encodings in this description are named to match the encodings of CPY (scalar).
- The description of CPY (scalar) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	1 0 0	0	1 0	1	Pg	Rn	Zd

Encoding

```
MOV  <Zd>.<T>, <Pg>/M, <R><n|SP>
```

is equivalent to

```
CPY  <Zd>.<T>, <Pg>/M, <R><n|SP>
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<n|SP>

Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the “Rn” field.

Operation

The description of CPY (scalar) gives the operational pseudocode for this instruction.

Operational Information

The description of CPY (scalar) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.464 MOV (SIMD&FP scalar, predicated)

Move SIMD&FP scalar register to vector elements (predicated)

This instruction moves the SIMD & floating-point scalar source register to each Active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (SIMD&FP scalar). This means:

- The encodings in this description are named to match the encodings of CPY (SIMD&FP scalar).
- The description of CPY (SIMD&FP scalar) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 0 0	0	1 0	0	Pg	Vn	Zd

Encoding

```
MOV  <Zd>.<T>, <Pg>/M, <V><n>
```

is equivalent to

```
CPY  <Zd>.<T>, <Pg>/M, <V><n>
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<V>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<V>
00	B
01	H
10	S
11	D

<n>

Is the number [0-31] of the source SIMD&FP register, encoded in the “Vn” field.

Operation

The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

Operational Information

The description of CPY (SIMD&FP scalar) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.465 MOV (immediate, unpredicated)

Move signed immediate to vector elements (unpredicated)

This instruction unconditionally broadcasts the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

This is an alias of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The description of DUP (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0 0 1	0 0 1 0	1	size	1	1 1	0 0	0	1 1	sh	imm8	Zd
						opc					

Encoding

```
MOV  <Zd>.<T>, #<imm>{, <shift>}
```

is equivalent to

```
DUP  <Zd>.<T>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.466 MOV (scalar, unpredicated)

Move general-purpose register to vector elements (unpredicated)

This instruction unconditionally broadcasts the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (scalar). This means:

- The encodings in this description are named to match the encodings of DUP (scalar).
- The description of DUP (scalar) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0 0	0 0 0	0 0 1 1 1 0	Rn	Zd

Encoding

```
MOV  <Zd>.<T>, <R><n|SP>
```

is equivalent to

```
DUP  <Zd>.<T>, <R><n|SP>
```

and is always the preferred disassembly.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R>

Is a width specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<R>
00	W
01	W
10	W
11	X

<n|SP>

Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the “Rn” field.

Operation

The description of DUP (scalar) gives the operational pseudocode for this instruction.

Operational Information

The description of DUP (scalar) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.467 MOV (SIMD&FP scalar, unpredicated)

Move indexed element or SIMD&FP scalar to vector (unpredicated)

This instruction unconditionally broadcasts the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (indexed). This means:

- The encodings in this description are named to match the encodings of DUP (indexed).
- The description of DUP (indexed) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 … 24	23 22	21	20 … 16	15 … 10	9 … 5	4 … 0
0 0 0 0 0 1 0 1	imm2	1	tsz	0 0 1 0 0 0	Zn	Zd

Encoding

```
MOV  <Zd>.<T>, <V><n>
```

is equivalent to

```
DUP  <Zd>.<T>, <Zn>.<T>[0]
```

and is the preferred disassembly when `BitCount(imm2 :: tsz) == 1`.

Encoding

```
MOV  <Zd>.<T>, <Zn>.<T>[<imm>]
```

is equivalent to

```
DUP  <Zd>.<T>, <Zn>.<T>[<imm>]
```

and is the preferred disassembly when `BitCount(imm2 :: tsz) > 1`.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tsz”:

tsz	<T>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

tsz	<T>
x1000	D
10000	Q

<V>

Is a width specifier, encoded in “tsz”:

tsz	<V>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

<n>

Is the number [0-31] of the source SIMD&FP register, encoded in the “Zn” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<imm>

Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in “imm2:tsz”.

Operation

The description of DUP (indexed) gives the operational pseudocode for this instruction.

Operational Information

The description of DUP (indexed) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.468 MOV

Move logical bitmask immediate to vector (unpredicated)

This instruction unconditionally broadcasts the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This is an alias of DUPM. This means:

- The encodings in this description are named to match the encodings of DUPM.
- The description of DUPM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 18	17 5	4 0
0 0 0	0 0 1 0	1	1 1	0 0	0 0	imm13	Zd

Encoding

```
MOV  <Zd>.<T>, #<const>
```

is equivalent to

```
DUPM  <Zd>.<T>, #<const>
```

and is the preferred disassembly when `SVEMoveMaskPreferred(imm13)`.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of DUPM gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.469 MOV

Move predicate (unpredicated)

This instruction reads all elements from the source predicate and places those elements in the destination predicate. This instruction is unpredicated. This instruction does not set the condition flags.

For programmer convenience, an assembler must also accept predicate-as-counter register names for the source and destination predicate registers.

This is an alias of ORR (predicates). This means:

- The encodings in this description are named to match the encodings of ORR (predicates).
- The description of ORR (predicates) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

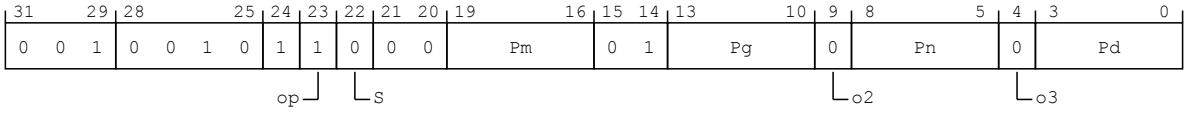

Encoding

```
MOV  <Pd>.B, <Pn>.B
```

is equivalent to

```
ORR  <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B
```

and is the preferred disassembly when `S == '0' && Pn == Pm && Pm == Pg`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of ORR (predicates) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (predicates) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.470 MOV (vector, unpredicated)

Move vector register (unpredicated)

This instruction copies the contents of the source vector register to the destination vector register. This instruction is unpredicated.

This is an alias of ORR (vectors, unpredicated). This means:

- The encodings in this description are named to match the encodings of ORR (vectors, unpredicated).
- The description of ORR (vectors, unpredicated) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	0 1 (opc)	1	Zm	0 0 1	1 0 0	Zn	Zd

Encoding

```
MOV  <Zd>.D, <Zn>.D
```

is equivalent to

```
ORR  <Zd>.D, <Zn>.D, <Zn>.D
```

and is the preferred disassembly when `Zn == Zm`.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (vectors, unpredicated) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.471 MOV (predicate, merging)

Move predicates (merging)

This instruction reads Active elements from the source predicate and places those elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. This instruction does not set the condition flags.

This is an alias of SEL (predicates). This means:

- The encodings in this description are named to match the encodings of SEL (predicates).
- The description of SEL (predicates) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

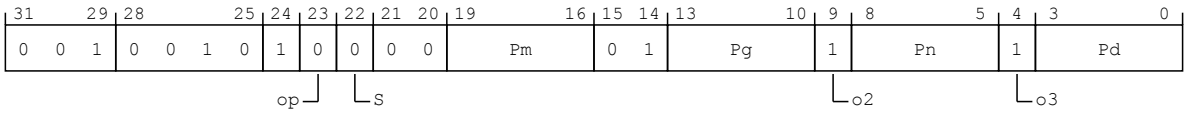

Encoding

```
MOV  <Pd>.B, <Pg>/M, <Pn>.B
```

is equivalent to

```
SEL  <Pd>.B, <Pg>, <Pn>.B, <Pd>.B
```

and is the preferred disassembly when `Pd == Pm`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of SEL (predicates) gives the operational pseudocode for this instruction.

Operational Information

The description of SEL (predicates) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.472 MOV (vector, predicated)

Move vector elements (predicated)

This instruction moves elements from the source vector to the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of SEL (vectors). This means:

- The encodings in this description are named to match the encodings of SEL (vectors).
- The description of SEL (vectors) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 14	13 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	Zm	1 1	Pv	Zn	Zd

Encoding

```
MOV  <Zd>.<T>, <Pv>/M, <Zn>.<T>
```

is equivalent to

```
SEL  <Zd>.<T>, <Pv>, <Zn>.<T>, <Zd>.<T>
```

and is the preferred disassembly when `Zd == Zm`.

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv>

Is the name of the vector select predicate register, encoded in the “Pv” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of SEL (vectors) gives the operational pseudocode for this instruction.

Operational Information

The description of SEL (vectors) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.473 MOVPRFX (predicated)

Move prefix (predicated)

This instruction is a hint to hardware that the instruction may be combined with the destructive instruction that follows it in program order to create a single constructive operation. This instruction is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated `MOVPRFX` or the unpredicated `MOVPRFX` instruction.

Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other `MOVPRFX` instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit "wide vector" operand), and the same destination vector as the `MOVPRFX` instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

SVE

(FEAT_SVE || FEAT_SME)

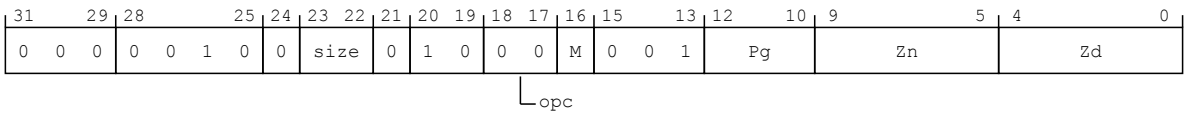

Encoding

```
MOVPRFX  <Zd>.<T>, <Pg>/<ZM>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = (M == '1');
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<ZM>

Is the predication qualifier, encoded in “M”:

M	<ZM>
0	Z
1	M

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let dest : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand1[e*:esize];
        result[e*:esize] = element;
    else
        result[e*:esize] = dest[e*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.474 MOVPRFX (unpredicated)

Move prefix (unpredicated)

This instruction is a hint to hardware that the instruction may be combined with the destructive instruction that follows it in program order to create a single constructive operation. This instruction is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Although the operation of the instruction is defined as a simple unpredicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other `MOVPRFX` instructions. The prefixed instruction must specify the same destination vector as the `MOVPRFX` instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	0 0	1	0 0 0 0 0	1 0 1 1	1 1	Zn	Zd
			opc		opc2				

Encoding

```
MOVPRFX  <Zd>, <Zn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let result : bits(VL) = Z{}(n);
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.475 MOVS (predicated)

Move predicates (zeroing), setting the condition flags

This instruction reads Active elements from the source predicate and places those elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is an alias of ANDS. This means:

- The encodings in this description are named to match the encodings of ANDS.
- The description of ANDS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

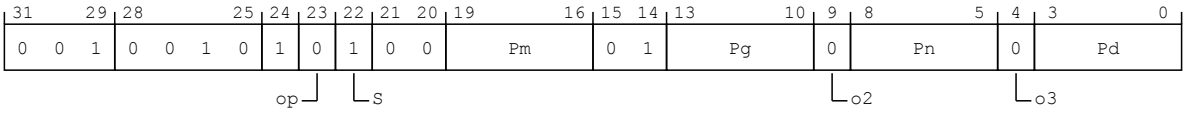

Encoding

```
MOVS  <Pd>.B, <Pg>/Z, <Pn>.B
```

is equivalent to

```
ANDS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B
```

and is the preferred disassembly when `S == '1' && Pn == Pm`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of ANDS gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.476 MOVS (unpredicated)

Move predicate (unpredicated), setting the condition flags

This instruction reads all elements from the source predicate and places those elements in the destination predicate. This instruction is unpredicated. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is an alias of ORRS. This means:

- The encodings in this description are named to match the encodings of ORRS.
- The description of ORRS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

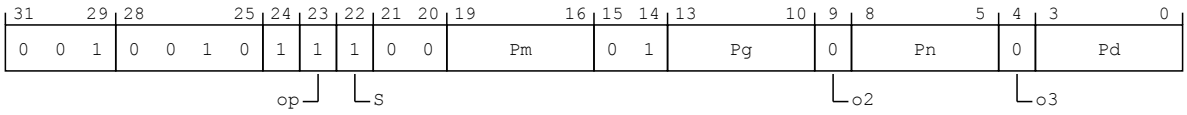

Encoding

```
MOVS  <Pd>.B, <Pn>.B
```

is equivalent to

```
ORRS  <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B
```

and is the preferred disassembly when `S == '1' && Pn == Pm && Pm == Pg`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of ORRS gives the operational pseudocode for this instruction.

Operational Information

The description of ORRS gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.477 MSB

Multiply-subtract to multiplicand

This instruction multiplies the active elements of the first source vector by the corresponding elements of the second source vector, and subtracts the result from elements of the third (addend) vector. The results are destructively placed in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	Zm	1	1	1	Pg	Za	Zdn
								op			

Encoding

```
MSB  <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
let a : integer = UInt(Za);
let sub_op : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Za>

Is the name of the third source scalable vector register, encoded in the “Za” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand3 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(a) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element1 : integer = UInt(operand1[e*:esize]);
        let element2 : integer = UInt(operand2[e*:esize]);
        let product : integer = element1 * element2;
        if sub_op then
            result[e*:esize] = operand3[e*:esize] - product;
        else
            result[e*:esize] = operand3[e*:esize] + product;
        end;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.478 MUL (vectors, predicated)

Multiply (predicated)

This instruction multiplies active elements of the first source vector by the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 0 0	0	0	0 0 0	Pg	Zm	Zdn
						H	U				

Encoding

```
MUL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let product : integer = element1 * element2;
        result[e*:esize] = product[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.479 MUL (immediate)

Multiply by immediate (unpredicated)

This instruction multiplies each element of the source vector by an immediate, and destructively places the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

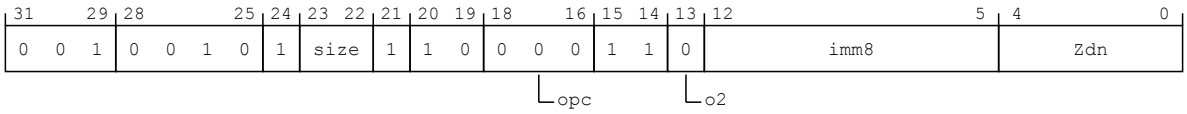

Encoding

```
MUL  <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let imm : integer = SInt(imm8);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is the signed immediate operand, in the range -128 to 127, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    result[e*:esize] = (element1 * imm)[esize-1:0];

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.480 MUL (vectors, unpredicated)

Multiply (unpredicated)

This instruction multiplies elements of the first source vector by the corresponding elements of the second source vector and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12	11–10	9–5	4–0
0 0 0	0 0 1 0	0	size	1	Zm	0 1 1	0	0 0 (opc)	Zn	Zd

Encoding

```
MUL  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let product : integer = element1 * element2;
    result[e*:esize] = product[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.481 MUL (indexed)

Multiply by indexed element

This instruction multiplies all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment. This instruction is unpredicated.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23	22	21	20 19	18 16	15 10	9 5	4 0
0 1 0	0 0 1 0	0	0	i3h	1	i3l	Zm	1 1 1 1 1 0	Zn	Zd

Encoding

```
MUL  <Zd>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	1 1 1 1 1 0	Zn	Zd
			size						

Encoding

```
MUL  <Zd>.S, <Zn>.S, <Zm>.S[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 10	9 5	4 0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	1 1 1 1 1 0	Zn	Zd
			size						

Encoding

```
MUL  <Zd>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[s*:esize]);
    let res : integer = element1 * element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.482 NAND

Bitwise NAND predicates

This instruction performs a bitwise NAND between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

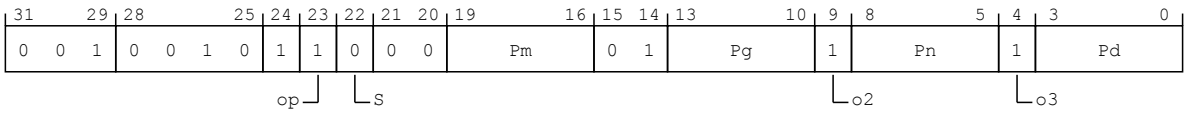

Encoding

```
NAND  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(NOT(element1 AND element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.483 NANDS

Bitwise NAND predicates, setting the condition flags

This instruction performs a bitwise NAND between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

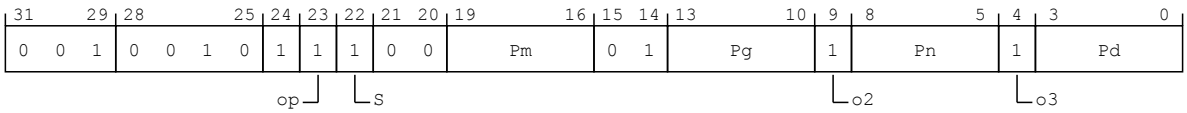

Encoding

```
NANDS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(NOT(element1 AND element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.484 NBSL

Bitwise inverted select

This instruction selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The inverted result is destructively placed in the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	1 1	1	Zm	0 0 1	1 1	1	Zk	Zdn
			opc					o2		

Encoding

```
NBSL  <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let m : integer = UInt(Zm);
let k : integer = UInt(Zk);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zk>

Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(k);

Z{VL}(dn) = NOT((operand1 AND operand3) OR (operand2 AND NOT(operand3)));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.485 NEG

Negate (predicated)

This instruction negates the signed integer value in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	0	1 1 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
NEG  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	0	1 1 1	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
NEG  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        var element : integer = SInt(operand[e*:esize]);
        element = -element;
        result[e*:esize] = element[esize-1:0];

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.486 NMATCH

Detect no matching elements, setting the condition flags

This instruction compares each active 8-bit or 16-bit element in the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector. For each element in the first source vector, if no matching element is found in the corresponding segment of the second source vector, the corresponding element of the destination predicate is set to true, otherwise it is set to false. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

Byte and halfword

(FEAT_SVE2)

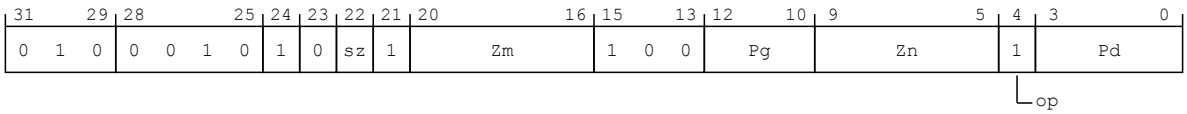

Encoding

```
NMATCH  <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(sz);
let g : integer = UInt(Pg);
let d : integer = UInt(Pd);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let segmentbase : integer = e - (e MOD eltspersegment);
        result[e*:psize] = ZeroExtend{psize}('1');
        let element1 : bits(esize) = operand1[e*:esize];
        for i = segmentbase to (segmentbase + eltspersegment) - 1 do
            let element2 : bits(esize) = operand2[i*:esize];
            if element1 == element2 then
                result[e*:psize] = ZeroExtend{psize}('0');
            end;
        end;
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.487 NOR

Bitwise NOR predicates

This instruction performs a bitwise NOR between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

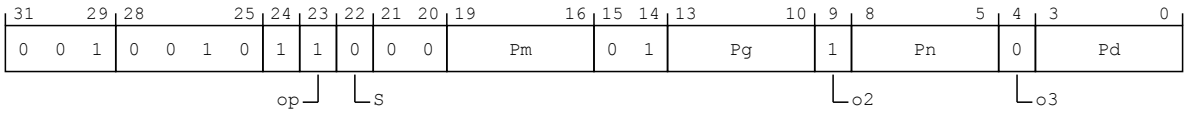

Encoding

```
NOR  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(NOT(element1 OR element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.488 NORS

Bitwise NOR predicates, setting the condition flags

This instruction performs a bitwise NOR between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

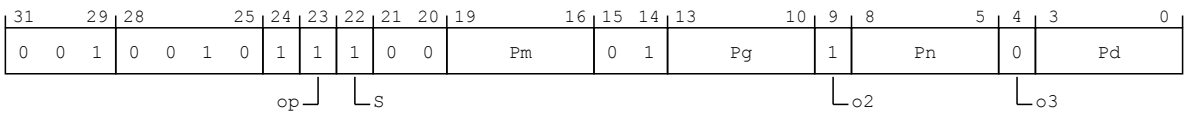

Encoding

```
NORS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(NOT(element1 OR element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.489 NOT (predicate)

Bitwise invert predicate

This instruction performs a bitwise invert of each active element of the source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This is an alias of EOR (predicates). This means:

- The encodings in this description are named to match the encodings of EOR (predicates).
- The description of EOR (predicates) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

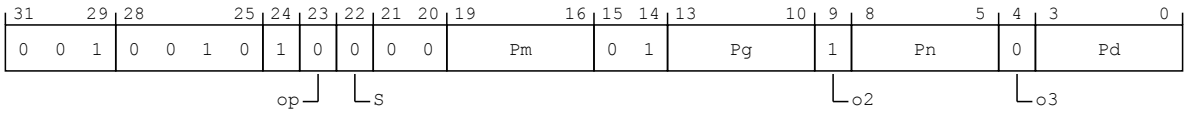

Encoding

```
NOT  <Pd>.B, <Pg>/Z, <Pn>.B
```

is equivalent to

```
EOR  <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B
```

and is the preferred disassembly when `Pm == Pg`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of EOR (predicates) gives the operational pseudocode for this instruction.

Operational Information

The description of EOR (predicates) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.490 NOT (vector)

Bitwise invert (predicated)

This instruction performs a bitwise invert of each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	1	1 1 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
NOT  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	1	1 1 0	1 0 1	Pg	Zn	Zd
					M		opc				

Encoding

```
NOT  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = NOT element;

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.491 NOTS

Bitwise invert predicate, setting the condition flags

This instruction performs a bitwise invert of each active element of the source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This is an alias of EORS. This means:

- The encodings in this description are named to match the encodings of EORS.
- The description of EORS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

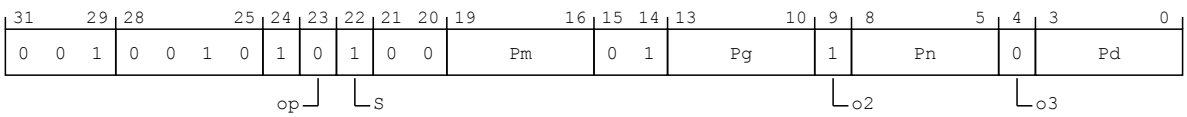

Encoding

```
NOTS  <Pd>.B, <Pg>/Z, <Pn>.B
```

is equivalent to

```
EORS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B
```

and is the preferred disassembly when `Pm == Pg`.

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

The description of EORS gives the operational pseudocode for this instruction.

Operational Information

The description of EORS gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.492 ORN (immediate)

Bitwise inclusive OR with inverted immediate (unpredicated)

This instruction performs a bitwise inclusive OR between an inverted immediate and each 64-bit element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ORR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 18	17 5	4 0
0 0 0	0 0 1 0	1	0 0 (opc)	0 0	0 0	imm13	Zdn

Encoding

```
ORN  <Zdn>.<T>, <Zdn>.<T>, #<const>
```

is equivalent to

```
ORR  <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.493 ORN (predicates)

Bitwise inclusive OR inverted predicate

This instruction performs a bitwise inclusive OR between the active elements of the first source predicate and the inverted corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

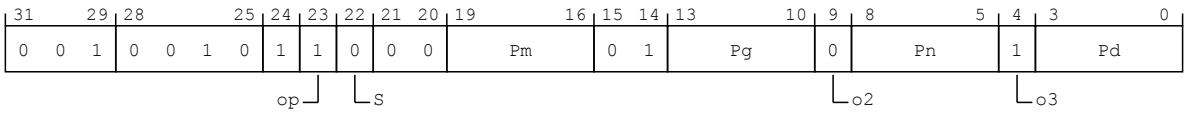

Encoding

```
ORN  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 OR (NOT element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.494 ORNS

Bitwise inclusive OR inverted predicate, setting the condition flags

This instruction performs a bitwise inclusive OR between the active elements of the first source predicate and the inverted corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

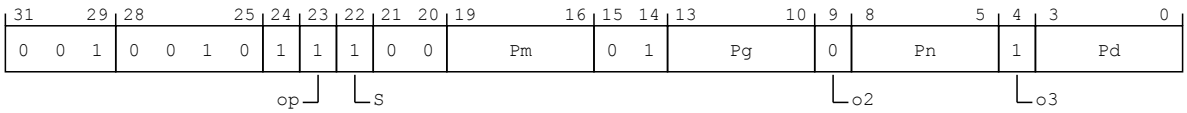

Encoding

```
ORNS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 OR (NOT element2));
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.495 ORQV

Bitwise inclusive OR reduction of quadword vector segments

This instruction performs a bitwise inclusive OR between elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as all zeros.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 1 1	0 0 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
ORQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : bits(esize);

for e = 0 to elempersegment-1 do
    dtmp = Zeros{esize};
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = dtmp OR stmp[e*:esize];
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.496 ORR (predicates)

Bitwise inclusive OR predicates

This instruction performs a bitwise inclusive OR between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This instruction is used by the alias MOV.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

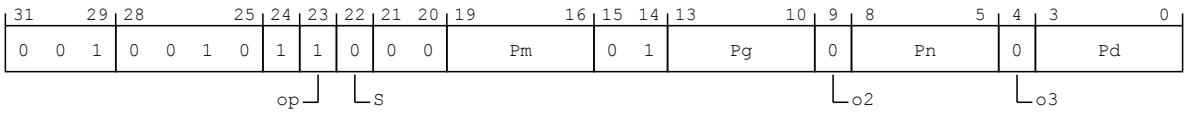

Encoding

```
ORR  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV	S == '0' && Pn == Pm && Pm == Pg

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 OR element2);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.497 ORR (vectors, predicated)

Bitwise inclusive OR (predicated)

This instruction performs a bitwise inclusive OR between the active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 1	0 0 0 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
ORR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element1 OR element2;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.498 ORR (immediate)

Bitwise inclusive OR with immediate (unpredicated)

This instruction performs a bitwise inclusive OR between an immediate and each 64-bit element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction ORN (immediate).

SVE

(FEAT_SVE || FEAT_SME)

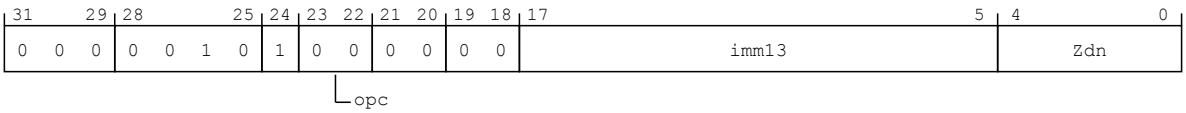

Encoding

```
ORR  <Zdn>.<T>, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn);
var imm : bits(64);
(imm, -) = DecodeBitMasks{64}(imm13[12], imm13[5:0], imm13[11:6], TRUE);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “imm13”:

imm13	<T>
0xxxxxx0xxxxx	S
0xxxxxx10xxxx	H
0xxxxxx110xxx	B
0xxxxxx1110xx	B
0xxxxxx11110x	B
0xxxxxx11111x	RESERVED
1xxxxxxxxxxxx	D

<const>

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand[e*:64];
    result[e*:64] = element1 OR imm;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.499 ORR (vectors, unpredicated)

Bitwise inclusive OR (unpredicated)

This instruction performs a bitwise inclusive OR between the elements of the first source vector and the corresponding elements of the second source vector, and places the first in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (vector, unpredicated).

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	0 1 (opc)	1	Zm	0 0 1	1 0 0	Zn	Zd

Encoding

```
ORR  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Alias Conditions

Alias	Is preferred when
MOV (vector, unpredicated)	Zn == Zm

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);

Z{VL}(d) = operand1 OR operand2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.500 ORRS

Bitwise inclusive OR predicates, setting the condition flags

This instruction performs a bitwise inclusive OR between the active elements of the first source predicate and the corresponding elements of the second source predicate, and places the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This instruction is used by the alias MOVS (unpredicated).

Setting the condition flags

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 10	9	8 5	4	3 0
0 0 1	0 0 1 0	1	1	1	0 0	Pm	0 1	Pg	0	Pn	0	Pd
		op		S					o2		o3	

Encoding

```
ORRS  <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOVS (unpredicated)	S == '1' && Pn == Pm && Pm == Pg

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1 OR element2);
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.501 ORV

Bitwise inclusive OR reduction to scalar

This instruction performs a bitwise inclusive OR between the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 1 0	0 0 (opc)	0 0 1	Pg	Zn	Vd

Encoding

```
ORV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(esize) = Zeros{esize};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result = result OR operand[e*:esize];

    end;
end;
V{esize}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.502 PEXT (predicate)

Predicate extract from predicate-as-counter

This instruction converts the source predicate-as-counter into a four register wide predicate-as-mask, and copies the portion of the mask value selected by the portion index to the destination predicate register. A portion corresponds to a one predicate register fraction of the wider predicate-as-mask value.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31 29	28 25	24	23 22	21	20 16	15 14	13 11	10	9 8	7 5	4	3 0
0 0 1	0 0 1 0	1	size	1	0 0 0 0 0	0 1	1 1 0	0	imm2	PNn	1	Pd

Encoding

```
PEXT  <Pd>.<T>, <PNn>[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt('1'::PNn);
let d : integer = UInt(Pd);
let part : integer = UInt(imm2);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<PNn>

Is the name of the first source scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNn” field.

<imm>

Is the portion index, in the range 0 to 3, encoded in the “imm2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let pred : bits(PL) = P{}(n);
let mask : bits(PL*4) = CounterToPredicate{}(pred[15:0]);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let pbit : bit = PredicateElement{PL*4}(mask, part * elements + e, esize);
    result[e*:psize] = ZeroExtend{psize}(pbit);
end;

P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.503 PEXT (predicate pair)

Predicate extract pair from predicate-as-counter

This instruction converts the source predicate-as-counter into a four register wide predicate-as-mask, and copies the two portions of the mask value selected by the portion index to the two destination predicate registers. A portion corresponds to a one predicate register fraction of the wider predicate-as-mask value.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31 29	28 25	24	23 22	21	20 16	15 14	13 11	10 9	8	7 5	4	3 0
0 0 1	0 0 1 0	1	size	1	0 0 0 0 0	0 1	1 1 0	1 0	i1	PNn	1	Pd

Encoding

```
PEXT  { <Pd1>.<T>, <Pd2>.<T> }, <PNn>[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt('1'::PNn);
let d0 : integer = UInt(Pd);
let d1 : integer = (UInt(Pd) + 1) MOD 16;
let part : integer = UInt(i1);
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded in the “Pd” field.

<PNn>

Is the name of the first source scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the portion index, in the range 0 to 1, encoded in the “i1” field.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let pred : bits(PL) = P{}(n);
let mask : bits(PL*4) = CounterToPredicate{}(pred[15:0]);
var result0 : bits(PL);
var result1 : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let pbit : bit = PredicateElement{PL*4}(mask, part * 2 * elements + e, esize);
    result0[e*:psize] = ZeroExtend{psize}(pbit);
end;

for e = 0 to elements-1 do
    let pbit : bit = PredicateElement{PL*4}(mask, part * 2 * elements + elements + e, esize);
    result1[e*:psize] = ZeroExtend{psize}(pbit);
end;

P{PL}(d0) = result0;
P{PL}(d1) = result1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.504 PFALSE

Set all predicate elements to false

This instruction sets all elements in the destination predicate to false.

For programmer convenience, an assembler must also accept predicate-as-counter register name for the destination predicate register.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 11	10 9	8 5	4	3 0
0 0 1	0 0 1 0	1	0	0	0 1	1 0 0 0	1 1	1 0 0	1 0	0 0 0 0	0	Pd
			op	S								

Encoding

```
PFALSE  <Pd>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Pd);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
P{PL}(d) = Zeros{PL};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.505 PFIRST

Set the First active predicate element to true

This instruction sets the First active element in the destination predicate to true, otherwise elements from the source predicate are passed through unchanged. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 11	10 9	8 5	4	3 0
0 0 1	0 0 1 0	1	0	1	0 1	1 0 0 0	1 1	0 0 0	0 0	Pg	0	Pdn
			op	S								

Encoding

```
PFIRST  <Pdn>.B, <Pg>, <Pdn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let dn : integer = UInt(Pdn);
```

Assembler Symbols

<Pdn>

Is the name of the source and destination scalable predicate register, encoded in the “Pdn” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var result : bits(PL) = P{}(dn);
var first : integer = -1;
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) && first == -1 then
        first = e;
    end;
end;

if first >= 0 then
    result[first*:psize] = ZeroExtend{psize}('1');
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(dn) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.506 PMLAL

Multi-vector polynomial multiply long and accumulate

This instruction performs a polynomial multiplication over [0, 1] of the even-numbered elements of the first source vector by the corresponding elements of the second source vector, and then performs a bitwise exclusive-OR between the result and the overlapping double-width elements of the first destination vector. The same operation is performed with odd-numbered elements of the source vectors, writing to the second destination vector. This instruction is unpredicated.

This instruction is legal when executed in Streaming SVE mode if both FEAT_SSVE_AES and FEAT_SVE_AES2 are implemented.

SVE2

(FEAT_SVE_AES2)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 1	0
0 1 0	0 0 1 0	1	0 0	1	Zm	1 1 1	1 1 1	Zn	Zda	0
			size							

Encoding

```
PMLAL  { <Zda1>.Q-<Zda2>.Q }, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda::'0');
```

Assembler Symbols

<Zda1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zda” times 2.

<Zda2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zda” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result_lo : bits(VL) = Z{}(da + 0);
var result_hi : bits(VL) = Z{}(da + 1);

for e = 0 to elements-1 do
    let element1_lo : bits(esize DIV 2) = operand1[(2*e + 0)*:(esize DIV 2)];
    let element2_lo : bits(esize DIV 2) = operand2[(2*e + 0)*:(esize DIV 2)];
    let element1_hi : bits(esize DIV 2) = operand1[(2*e + 1)*:(esize DIV 2)];
    let element2_hi : bits(esize DIV 2) = operand2[(2*e + 1)*:(esize DIV 2)];
    let product_lo : bits(esize) = PolynomialMult{esize DIV 2, esize DIV 2}(element1_lo,
                                                                            element2_lo);
    let product_hi : bits(esize) = PolynomialMult{esize DIV 2, esize DIV 2}(element1_hi,
                                                                            element2_hi);
    result_lo[e*:esize] = result_lo[e*:esize] XOR product_lo;
    result_hi[e*:esize] = result_hi[e*:esize] XOR product_hi;
end;

Z{VL}(da + 0) = result_lo;
Z{VL}(da + 1) = result_hi;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.507 PMOV (to predicate)

Move predicate from vector

This instruction copies a packed bitmap, where bit value `0b1` represents TRUE and bit value `0b0` represents FALSE, from a portion of the source vector register to elements of the destination SVE predicate register.

Because the number of bits in an SVE predicate element scales with the vector element size, the behavior varies according to the specified element size.

- When the predicate element specifier is B, each bit [N] from the least-significant VL/8 bits in the source vector register is copied to bit [N] of the destination predicate register. The portion index, if specified, must be 0.
- When the predicate element specifier is H, each bit [N] within the indexed block of VL/16 bits in the source vector register is copied to bit [N*2] of the destination predicate register, and the other bits in the predicate are set to zero. The portion index is in the range 0 to 1, inclusive.
- When the predicate element specifier is S, each bit [N] within the indexed block of VL/32 bits in the source vector register is copied to bit [N*4] of the destination predicate register, and the other bits in the predicate are set to zero. The portion index is in the range 0 to 3, inclusive.
- When the predicate element specifier is D, each bit [N] within the indexed block of VL/64 bits in the source vector register is copied to bit [N*8] of the destination predicate register, and the other bits in the predicate are set to zero. The portion index is in the range 0 to 7, inclusive.

The portion index is optional, defaulting to 0 if omitted.

It has encodings from 4 classes: Byte, Doubleword, Halfword, and Word

Byte

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–10	9–5	4	3–0
0 0 0	0 0 1 0	1	0 0 (opc)	1	0 1	0 1 (opc2)	0	0 0 1 1 1 0	Zn	0	Pd

Encoding

```
PMOV  <Pd>.B, <Zn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let esize : integer{} = 8;
let imm : integer = 0;
```

Doubleword

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23	22	21	20–19	18–17	16	15–10	9–5	4	3–0
0 0 0	0 0 1 0	1	1	i3h	1	0 1	i3l	0	0 0 1 1 1 0	Zn	0	Pd

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
PMOV  <Pd>.D, <Zn>{[<imm>]}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let esize : integer{} = 64;
let imm : integer = UInt(i3h::i3l);
```

Halfword

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 19	18	17	16	15 10	9 5	4	3 0
0 0 0	0 0 1 0	1	0 0	1	0 1	1	i1	0	0 0 1 1 1 0	Zn	0	Pd
			opc									

Encoding

```
PMOV  <Pd>.H, <Zn>{[<imm>]}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let esize : integer{} = 16;
let imm : integer = UInt(i1);
```

Word

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 10	9 5	4	3 0
0 0 0	0 0 1 0	1	0 1	1	0 1	i2	0	0 0 1 1 1 0	Zn	0	Pd
			opc								

Encoding

```
PMOV  <Pd>.S, <Zn>{[<imm>]}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Pd);
let esize : integer{} = 32;
let imm : integer = UInt(i2);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<imm>

For the “Doubleword” variant: is the optional portion index, in the range 0 to 7, defaulting to 0, encoded in the “i3h:i3l” fields.

For the “Halfword” variant: is the optional portion index, in the range 0 to 1, defaulting to 0, encoded in the “i1” field.

For the “Word” variant: is the optional portion index, in the range 0 to 3, defaulting to 0, encoded in the “i2” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    result[e*:psize] = ZeroExtend{psize}(operand[(elements * imm) + e]);
end;

P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.508 PMOV (to vector)

Move predicate to vector

This instruction copies the source SVE predicate register elements into the destination vector register as a packed bitmap with one bit per predicate element, where bit value `0b1` represents a TRUE predicate element, and bit value `0b0` represents a FALSE predicate element.

Because the number of bits in an SVE predicate element scales with the vector element size, the behavior varies according to the specified element size.

- When the predicate element specifier is B, every bit in the predicate register is copied to the least-significant VL/8 bits of the destination vector register. The portion index, if specified, must be 0.
- When the predicate element specifier is H, every second bit in the predicate register is copied to the indexed block of VL/16 bits in the destination vector register, where the portion index is in the range 0 to 1, inclusive.
- When the predicate element specifier is S, every fourth bit in the predicate register is copied to the indexed block of VL/32 bits in the destination vector register, where the portion index is in the range 0 to 3, inclusive.
- When the predicate element specifier is D, every eighth bit in the predicate register is copied to the indexed block of VL/64 bits in the destination vector register, where the portion index is in the range 0 to 7, inclusive.

The portion index is optional, defaulting to 0 if omitted. When the index is zero, the instruction writes zeroes to the most significant VL-(VL/esize) bits of the destination vector register. When a non-zero index is specified, the packed bitmap is inserted into the destination vector register, and the unindexed blocks remain unchanged.

It has encodings from 4 classes: Byte, Doubleword, Halfword, and Word

Byte

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–10	9	8–5	4–0
0 0 0	0 0 1 0	1	0 0 (opc)	1	0 1	0 1 (opc2)	1	0 0 1 1 1 0	0	Pn	Zd

Encoding

```
PMOV  <Zd>, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Pn);
let d : integer = UInt(Zd);
let esize : integer{} = 8;
let imm : integer = 0;
```

Doubleword

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23	22	21	20–19	18–17	16	15–10	9	8–5	4–0
0 0 0	0 0 1 0	1	1	i3h	1	0 1	i3l	1	0 0 1 1 1 0	0	Pn	Zd

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
PMOV  <Zd>{[<imm>]}, <Pn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Pn);
let d : integer = UInt(Zd);
let esize : integer{} = 64;
let imm : integer = UInt(i3h::i3l);
```

Halfword

(FEAT_SVE2p1 || FEAT_SME2p1)

31	29	28	25	24	23	22	21	20	19	18	17	16	15	10	9	8	5	4	0
0 0 0		0 0 1 0		1	0	0	1	0	1	1	i1	1	0 0 1 1 1 0		0	Pn		Zd	

opc

Encoding

```
PMOV  <Zd>{[<imm>]}, <Pn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Pn);
let d : integer = UInt(Zd);
let esize : integer{} = 16;
let imm : integer = UInt(i1);
```

Word

(FEAT_SVE2p1 || FEAT_SME2p1)

31	29	28	25	24	23	22	21	20	19	18	17	16	15	10	9	8	5	4	0
0 0 0		0 0 1 0		1	0	1	1	0	1	i2		1	0 0 1 1 1 0		0	Pn		Zd	

opc

Encoding

```
PMOV  <Zd>{[<imm>]}, <Pn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Pn);
let d : integer = UInt(Zd);
let esize : integer{} = 32;
let imm : integer = UInt(i2);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

<imm>

For the “Doubleword” variant: is the optional portion index, in the range 0 to 7, defaulting to 0, encoded in the “i3h:i3l” fields.

For the “Halfword” variant: is the optional portion index, in the range 0 to 1, defaulting to 0, encoded in the “i1” field.

For the “Word” variant: is the optional portion index, in the range 0 to 3, defaulting to 0, encoded in the “i2” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand : bits(PL) = P{}(n);
var result : bits(VL);

if imm == 0 then
    result = Zeros{VL};
else
    result = Z{VL}(d);
end;

for e = 0 to elements-1 do
    result[(elements * imm) + e] = PredicateElement{PL}(operand, e, esize);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.509 PMUL

Polynomial multiply (unpredicated)

This instruction performs a polynomial multiplication over [0, 1] of all elements of the first source vector by the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0 0	0 0 1 0	0	0 0	1	Zm	0 1 1	0	0 1	Zn	Zd
			size					opc		

Encoding

```
PMUL  <Zd>.B, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = PolynomialMult{esize, esize}(element1, element2)[esize-1:0];
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.510 PMULL

Multi-vector polynomial multiply long

This instruction performs a polynomial multiplication over [0, 1] of the even-numbered elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the first destination vector. The same operation is performed with odd-numbered elements of the source vectors, writing to the second destination vector. This instruction is unpredicated.

This instruction is legal when executed in Streaming SVE mode if both FEAT_SSVE_AES and FEAT_SVE_AES2 are implemented.

SVE2

(FEAT_SVE_AES2)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–1	0
0 1 0	0 0 1 0	1	0 0	1	Zm	1 1 1	1 1 0	Zn	Zd	0

size

Encoding

```
PMULL  { <Zd1>.Q-<Zd2>.Q }, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_AES2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SSVE_AES) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var result_lo : bits(VL);
var result_hi : bits(VL);

for e = 0 to elements-1 do
    let element1_lo : bits(esize DIV 2) = operand1[(2*e + 0)*:(esize DIV 2)];
    let element2_lo : bits(esize DIV 2) = operand2[(2*e + 0)*:(esize DIV 2)];
    let element1_hi : bits(esize DIV 2) = operand1[(2*e + 1)*:(esize DIV 2)];
    let element2_hi : bits(esize DIV 2) = operand2[(2*e + 1)*:(esize DIV 2)];
    result_lo[e*:esize] = PolynomialMult{esize DIV 2, esize DIV 2}(element1_lo, element2_lo);
    result_hi[e*:esize] = PolynomialMult{esize DIV 2, esize DIV 2}(element1_hi, element2_hi);
end;

Z{VL}(d + 0) = result_lo;
Z{VL}(d + 1) = result_hi;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.511 PMULLB

Polynomial multiply long (bottom)

This instruction performs a polynomial multiplication over [0, 1] of the even-numbered elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether the 128-bit element variant is implemented. The 128-bit element variant is legal when executed in Streaming SVE mode if one of the following is true:

- Both FEAT_SSVE_AES and FEAT_SVE_PMULL128 are implemented.
- FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 16-bit or 64-bit elements and 128-bit element

16-bit or 64-bit elements

(FEAT_SVE2 || FEAT_SME)

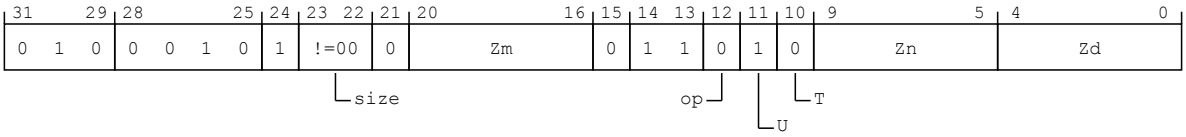

Encoding

```
PMULLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size[0] == '0' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

128-bit element

(FEAT_SVE_PMULL128)

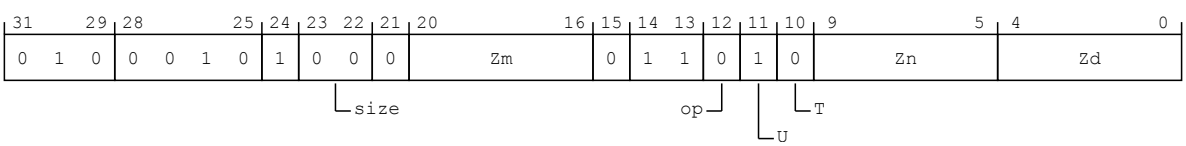

Encoding

```
PMULLB  <Zd>.Q, <Zn>.D, <Zm>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_PMULL128) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
1x	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
01	B
1x	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if esize == 128 then
    if IsFeatureImplemented(FEAT_SSVE_AES) then
        CheckSVEEnabled();
    else
        CheckNonStreamingSVEEnabled();
    end;
else
    CheckSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let element1 : bits(esize DIV 2) = operand1[(2*e + 0)*:(esize DIV 2)];
    let element2 : bits(esize DIV 2) = operand2[(2*e + 0)*:(esize DIV 2)];
    result[e*:esize] = PolynomialMult{esize DIV 2, esize DIV 2}(element1, element2);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.512 PMULLT

Polynomial multiply long (top)

This instruction performs a polynomial multiplication over [0, 1] of the odd-numbered elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether the 128-bit element variant is implemented. The 128-bit element variant is legal when executed in Streaming SVE mode if one of the following is true:

- Both FEAT_SSVE_AES and FEAT_SVE_PMULL128 are implemented.
- FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 16-bit or 64-bit elements and 128-bit element

16-bit or 64-bit elements

(FEAT_SVE2 || FEAT_SME)

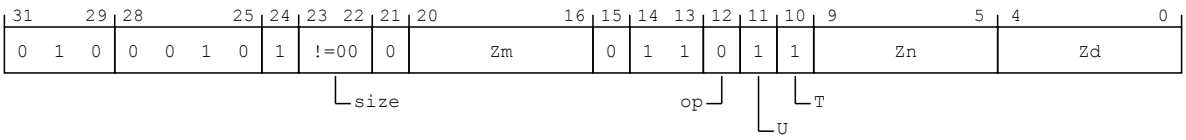

Encoding

```
PMULLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size[0] == '0' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

128-bit element

(FEAT_SVE_PMULL128)

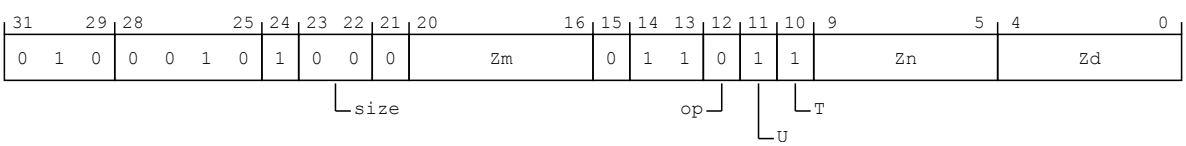

Encoding

```
PMULLT  <Zd>.Q, <Zn>.D, <Zm>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_PMULL128) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
1x	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
01	B
1x	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if esize == 128 then
    if IsFeatureImplemented(FEAT_SSVE_AES) then
        CheckSVEEnabled();
    else
        CheckNonStreamingSVEEnabled();
    end;
else
    CheckSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let element1 : bits(esize DIV 2) = operand1[(2*e + 1)*:(esize DIV 2)];
    let element2 : bits(esize DIV 2) = operand2[(2*e + 1)*:(esize DIV 2)];
    result[e*:esize] = PolynomialMult{esize DIV 2, esize DIV 2}(element1, element2);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.513 PNEXT

Find next active predicate

This instruction constructs a loop that iterates over all true elements in the vector select predicate register. If all elements in the first source predicate register are false it determines the first true element in the vector select predicate register, otherwise it determines the next true element in the vector select predicate register that follows the last true element in the first source predicate register. All elements of the destination predicate register are set to false, except the element corresponding to the determined vector select element, if any, which is set to true. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 16	15 14	13 11	10 9	8 5	4	3 0
0 0 1	0 0 1 0	1	size	0 1	1 0 0 1	1 1	0 0 0	1 0	Pv	0	Pdn

Encoding

```
PNEXT  <Pdn>.<T>, <Pv>, <Pdn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let v : integer = UInt(Pv);
let dn : integer = UInt(Pdn);
```

Assembler Symbols

<Pdn>

Is the name of the first source and destination scalable predicate register, encoded in the “Pdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv>

Is the name of the vector select predicate register, encoded in the “Pv” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer{} = VL DIV esize;
let mask : bits(PL) = P{}(v);
let operand : bits(PL) = P{}(dn);
var result : bits(PL) = Zeros{PL};
let psize : integer{} = esize DIV 8;

var next : integer = LastActiveElement{PL}(operand, esize) + 1;

while next < elements && (!ActivePredicateElement{PL}(mask, next, esize)) looplimit elements do
    next = next + 1;
end;

if next < elements then
    result[next*:psize] = ZeroExtend{psize}('1');
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(dn) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.514 PRFB (vector plus immediate)

Gather prefetch bytes (vector plus immediate)

This instruction performs a gather prefetch of bytes from the active memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	0 0	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Encoding

```
PRFB  <prfop>, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 0;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 1 0	0 0 1 0	0 0	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
PRFB  <prfop>, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 0;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let addr : bits(64) = ZeroExtend{64}(base[e*:esize]) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.515 PRFB (scalar plus immediate)

Contiguous prefetch bytes (immediate index)

This instruction prefetches contiguous byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 1	1	imm6	0	0 0 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFB  <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 0;
let offset : integer = SInt(imm6);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = (offset * elements) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.516 PRFB (scalar plus scalar)

Contiguous prefetch bytes (scalar index)

This instruction prefetches contiguous byte elements from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	0 0	0 0	!=11111	1 1 0	Pg	Rn	0	prfop
		msz		Rm					

Encoding

```
PRFB  <prfop>, <Pg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = UInt(offset) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.517 PRFB (scalar plus vector)

Gather prefetch bytes (scalar plus vector)

This instruction performs a gather prefetch of bytes from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign-extended or zero-extended from 32 to 64 bits. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, and 64-bit scaled offset

32-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 0 0	0 0 1 0	0 0	xs	1	Zm	0	0 0 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFB  <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit unpacked scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	xs	1	Zm	0	0 0 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFB  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	1 1	Zm	1	0 0 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFB  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.518 PRFD (vector plus immediate)

Gather prefetch doublewords (vector plus immediate)

This instruction performs a gather prefetch of doublewords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 1	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Encoding

```
PRFD  <prfop>, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 3;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 1 0	0 0 1 0	1 1	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
PRFD  <prfop>, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 3;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let addr : bits(64) = ZeroExtend{64}(base[e*:esize]) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.519 PRFD (scalar plus immediate)

Contiguous prefetch doublewords (immediate index)

This instruction prefetches contiguous doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

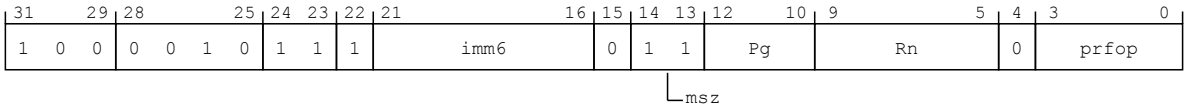

Encoding

```
PRFD  <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 3;
let offset : integer = SInt(imm6);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = (offset * elements) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.520 PRFD (scalar plus scalar)

Contiguous prefetch doublewords (scalar index)

This instruction prefetches contiguous doubleword elements from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 8 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 1 (msz)	0 0	!=11111 (Rm)	1 1 0	Pg	Rn	0	prfop

Encoding

```
PRFD  <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 3;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = UInt(offset) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.521 PRFD (scalar plus vector)

Gather prefetch doublewords (scalar plus vector)

This instruction performs a gather prefetch of doublewords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then multiplied by 8. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, and 64-bit scaled offset

32-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 0 0	0 0 1 0	0 0	xs	1	Zm	0	1 1 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFD  <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 3;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit unpacked scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	xs	1	Zm	0	1 1 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFD  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 3;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	1 1	Zm	1	1 1 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFD  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 3;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.522 PRFH (vector plus immediate)

Gather prefetch halfwords (vector plus immediate)

This instruction performs a gather prefetch of halfwords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	0 1	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Encoding

```
PRFH  <prfop>, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 1;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 1 0	0 0 1 0	0 1	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
PRFH  <prfop>, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 1;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let addr : bits(64) = ZeroExtend{64}(base[e*:esize]) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.523 PRFH (scalar plus immediate)

Contiguous prefetch halfwords (immediate index)

This instruction prefetches contiguous halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 1	1	imm6	0	0 1 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFH  <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 1;
let offset : integer = SInt(imm6);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = (offset * elements) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.524 PRFH (scalar plus scalar)

Contiguous prefetch halfwords (scalar index)

This instruction prefetches contiguous halfword elements from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	0 1	0 0	!=11111	1 1 0	Pg	Rn	0	prfop
		msz		Rm					

Encoding

```
PRFH  <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = UInt(offset) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.525 PRFH (scalar plus vector)

Gather prefetch halfwords (scalar plus vector)

This instruction performs a gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, and 64-bit scaled offset

32-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 0 0	0 0 1 0	0 0	xs	1	Zm	0	0 1	Pg	Rn	0	prfop
							msz				

Encoding

```
PRFH  <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit unpacked scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	xs	1	Zm	0	0 1 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFH  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 1;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	1 1	Zm	1	0 1 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFH  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.526 PRFW (vector plus immediate)

Gather prefetch words (vector plus immediate)

This instruction performs a gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 0	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Encoding

```
PRFW  <prfop>, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 2;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 1 0	0 0 1 0	1 0	0 0	imm5	1 1 1	Pg	Zn	0	prfop
		msz							

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
PRFW  <prfop>, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 2;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let addr : bits(64) = ZeroExtend{64}(base[e*:esize]) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.527 PRFW (scalar plus immediate)

Contiguous prefetch words (immediate index)

This instruction prefetches contiguous word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21–16	15	14–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 1	1	imm6	0	1 0 (msz)	Pg	Rn	0	prfop

Encoding

```
PRFW  <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 2;
let offset : integer = SInt(imm6);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = (offset * elements) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.528 PRFW (scalar plus scalar)

Contiguous prefetch words (scalar index)

This instruction prefetches contiguous word elements from the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

Note

Arm strongly recommends the following for this instruction:

- A PE does not perform a prefetch as a result of Inactive elements.
- Software uses the predicate operand to suppress prefetches from unwanted addresses.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4	3–0
1 0 0	0 0 1 0	1 0 (msz)	0 0	!=11111 (Rm)	1 1 0	Pg	Rn	0	prfop

Encoding

```
PRFW  <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let scale : integer = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(64);

if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = X{64}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let eoff : integer = UInt(offset) + e;
        let addr : bits(64) = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.529 PRFW (scalar plus vector)

Gather prefetch words (scalar plus vector)

This instruction performs a gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.

The `<prfop>` operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, and 64-bit scaled offset

32-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 0 0	0 0 1 0	0 0	xs	1	Zm	0	1 0	Pg	Rn	0	prfop
							msz				

Encoding

```
PRFW  <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit unpacked scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22	21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	xs	1	Zm	0	1 0	Pg	Rn	0	prfop
							msz				

Encoding

```
PRFW  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = (xs == '0');
let scale : integer = 2;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14 13	12 10	9 5	4	3 0
1 1 0	0 0 1 0	0 0	1 1	Zm	1	1 0	Pg	Rn	0	prfop
						msz				

Encoding

```
PRFW  <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let level : integer = UInt(prfop[2:1]);
let stream : boolean = (prfop[0] == '1');
let pref_hint : PrefetchHint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<prfop>

Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(64);
var offset : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if AnyActiveElement{PL}(mask, esize) then
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.530 PSEL

Predicate select between predicate register or all-false

This instruction places the contents of the first source predicate register into the destination predicate register if the indexed element of the second source predicate is true, otherwise it sets the destination predicate to all-false. The indexed element is determined by the sum of a general-purpose index register and an immediate, modulo the number of elements. This instruction does not set the condition flags. For programmer convenience, an assembler must also accept predicate-as-counter register names for the destination predicate register and the first source predicate register.

SVE2

(FEAT_SME || FEAT_SVE2p1)

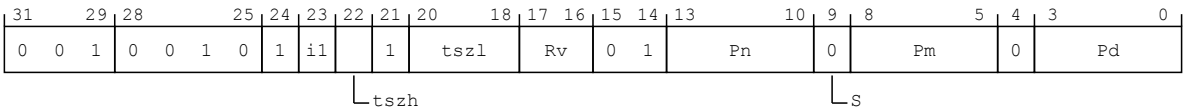

Encoding

```
PSEL  <Pd>, <Pn>, <Pm>.<T>[<Wv>, <imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let imm5 : bits(5) = i1::tszh::tszl;
var esize : integer;
var imm : integer;
case tszh::tszl of
    when '0000' => EndOfDecode(Decode_UNDEF);
    when '1000' => esize = 64;  imm = UInt(imm5[4]);
    when 'x100' => esize = 32;  imm = UInt(imm5[4:3]);
    when 'xx10' => esize = 16;  imm = UInt(imm5[4:2]);
    when 'xxx1' => esize = 8;   imm = UInt(imm5[4:1]);
end;
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let v : integer = UInt('011'::Rv);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

tszh	tszl	<T>
0	000	RESERVED
x	xx1	B
x	x10	H
x	100	S
1	000	D

<Wv>

Is the 32-bit name of the vector select register W12-W15, encoded in the “Rv” field.

<imm>

Is the element index, in the range 0 to one less than the number of vector elements in a 128-bit vector register, encoded in “i1:tszh:tszl”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
let idx : bits(32) = X{}(v);
let element : integer = (UInt(idx) + imm) MOD elements;
var result : bits(PL);

if ActivePredicateElement{PL}(operand2, element, esize) then
    result = operand1;
else
    result = Zeros{PL};
end;

P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.531 PTEST

Set condition flags for predicate

This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate source register, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

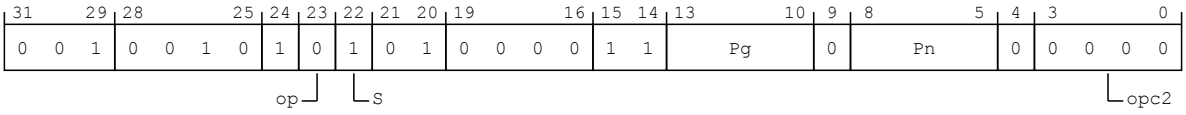

Encoding

```
PTEST  <Pg>, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
```

Assembler Symbols

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let result : bits(PL) = P{}(n);

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.532 PTRUE (predicate)

Initialize predicate from named constraint

This instruction sets elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length, then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. This instruction does not set the condition flags.

Not setting the condition flags

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21 20	19 17	16	15 14	13 11	10	9 5	4	3 0
0 0 1	0 0 1 0	1	size	0 1	1 0 0	0	1 1	1 0 0	0	pattern	0	Pd
						S						

Encoding

```
PTRUE  <Pd>.<T>{, <pattern>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
let pat : bits(5) = pattern;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let pbit : bit = if e < count then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
end;

if setflags then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.[N,Z,C,V] = PredTest{PL}(result, result, esize);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.533 PTRUE (predicate as counter)

Initialize predicate-as-counter to all active

This instruction sets the destination predicate as all Active elements, using the predicate-as-counter encoding.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

31 29	28 25	24	23 22	21	20 16	15 14	13 11	10 5	4	3	2 0
0 0 1	0 0 1 0	1	size	1	0 0 0 0 0	0 1	1 1 1	0 0 0 0 0 0	1	0	PNd

Encoding

```
PTRUE  <PNd>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt('1'::PNd);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let result : bits(PL) = EncodePredCount{}(esize, elements, elements, FALSE);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.534 PTRUES

Initialize predicate from named constraint and set the condition flags

This instruction sets elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length, then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

Setting the condition flags

(FEAT_SVE || FEAT_SME)

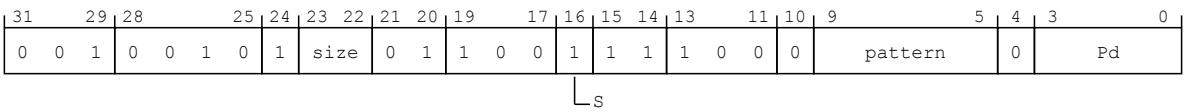

Encoding

```
PTRUES  <Pd>.<T>{, <pattern>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
let pat : bits(5) = pattern;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let pbit : bit = if e < count then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(result, result, esize);
end;
P{PL}(d) = result;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.535 PUNPKHI, PUNPKLO

Unpack and widen half of predicate

This instruction unpacks elements from the lowest or highest half of the source predicate and places them in elements of twice their size within the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

High half

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–17	16	15–13	12–9	8–5	4	3–0
0 0 0	0 0 1 0	1	0 0	1	1 0 0 0	1	0 1 0	0 0 0 0	Pn	0	Pd

H

Encoding

```
PUNPKHI  <Pd>.H, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
let hi : boolean = TRUE;
```

Low half

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–17	16	15–13	12–9	8–5	4	3–0
0 0 0	0 0 1 0	1	0 0	1	1 0 0 0	0	0 1 0	0 0 0 0	Pn	0	Pd

H

Encoding

```
PUNPKLO  <Pd>.H, <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
let hi : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand : bits(PL) = P{}(n);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let pbit : bit = PredicateElement{PL}(operand, if hi then e + elements else e, esize DIV 2);
    result[e*:psize] = ZeroExtend{psize}(pbit);
end;

P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.536 RADDHNB

Rounding add narrow high part (bottom)

This instruction adds each vector element of the first source vector to the corresponding vector element of the second source vector, and places the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

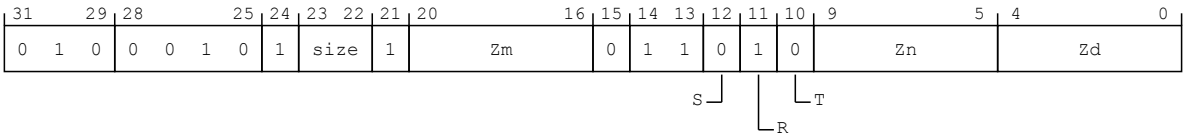

Encoding

```
RADDHNB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = ((element1 + element2) + (1 << (halfesize - 1))) >> halfesize;
    result[(2*e + 0)*:halfesize] = res[halfesize-1:0];
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.537 RADDHNT

Rounding add narrow high part (top)

This instruction adds each vector element of the first source vector to the corresponding vector element of the second source vector, and places the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

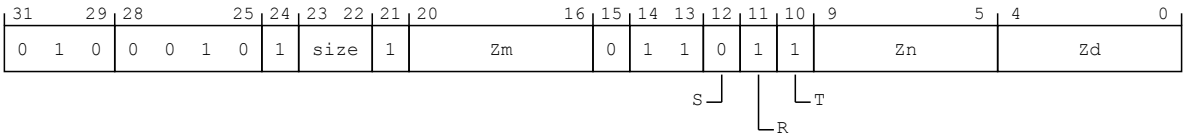

Encoding

```
RADDHNT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = ((element1 + element2) + (1 << (halfesize - 1))) >> halfesize;
    result[(2*e + 1)*:halfesize] = res[halfesize-1:0];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.538 RAX1

Bitwise rotate left by 1 and exclusive-OR

This instruction rotates each 64-bit element of the second source vector left by 1 and performs an exclusive-OR with the corresponding elements of the first source vector. The results are placed in the corresponding elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SHA3 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled, or FEAT_SME2p1 is implemented.

SVE2

(FEAT_SVE_SHA3)

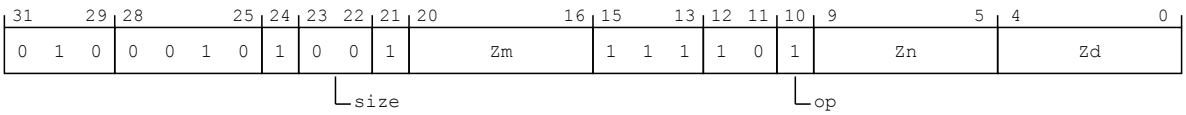

Encoding

```
RAX1  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_SHA3) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if IsFeatureImplemented(FEAT_SME2p1) then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand1[e*:64];
    let element2 : bits(64) = operand2[e*:64];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[e*:64] = element1 XOR ROL(element2, 1);
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.539 RBIT

Reverse bits (predicated)

This instruction reverses the bits in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–18	17–16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	0	0 1	1 1 (opc)	1 0	0 (Z)	Pg	Zn	Zd

Encoding

```
RBIT  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19–18	17–16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	0	0 1	1 1 (opc)	1 0	1 (Z)	Pg	Zn	Zd

Encoding

```
RBIT  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = BitReverse{esize}(element);

    end;
end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.540 RDFFR (unpredicated)

Read the first-fault register

This instruction reads the first-fault register (`FFR`) and places the read values in the destination predicate without predication.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24	23	22	21–20	19–16	15–14	13–11	10–9	8–5	4	3–0
0 0 1	0 0 1 0	1	0	0	0 1	1 0 0 1	1 1	1 1 0	0 0	0 0 0 0	0	Pd
			op	S								

Encoding

```
RDFFR  <Pd>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Pd);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let ffr : bits(PL) = FFR{}();
P{PL}(d) = ffr;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.541 RDFFR (predicated)

Return predicate of successfully loaded elements

This instruction reads the first-fault register (`FFR`) and places active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction does not set the condition flags.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

Not setting the condition flags

(FEAT_SVE)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 11	10 9	8 5	4	3 0
0 0 1	0 0 1 0	1	0	0	0 1	1 0 0 0	1 1	1 1 0	0 0	Pg	0	Pd

op, S

Encoding

```
RDFFR  <Pd>.B, <Pg>/Z
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let d : integer = UInt(Pd);
let setflags : boolean = FALSE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let ffr : bits(PL) = FFR{}();
let result : bits(PL) = ffr AND mask;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, 8);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.542 RDFFRS

Return predicate of successfully loaded elements, setting the condition flags

This instruction reads the first-fault register (`FFR`) and places active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

Setting the condition flags

(FEAT_SVE)

31 29	28 25	24	23	22	21 20	19 16	15 14	13 11	10 9	8 5	4	3 0
0 0 1	0 0 1 0	1	0	1	0 1	1 0 0 0	1 1	1 1 0	0 0	Pg	0	Pd
			op	S								

Encoding

```
RDFFRS  <Pd>.B, <Pg>/Z
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let d : integer = UInt(Pd);
let setflags : boolean = TRUE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let mask : bits(PL) = P{}(g);
let ffr : bits(PL) = FFR{}();
let result : bits(PL) = ffr AND mask;

if setflags then
    PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, 8);
end;
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.543 RDVL

Read multiple of vector register size to scalar register

This instruction multiplies the current vector register size in bytes by an immediate in the range -32 to 31 and places the result in the 64-bit destination general-purpose register.

SVE

(FEAT_SVE || FEAT_SME)

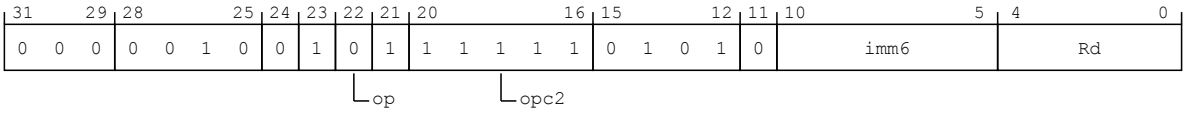

Encoding

```
RDVL  <Xd>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Rd);
let imm : integer = SInt(imm6);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<imm>

Is the signed immediate operand, in the range -32 to 31, encoded in the “imm6” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let len : integer = imm * (VL DIV 8);
X{64}(d) = len[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.544 REV (predicate)

Reverse all elements in a predicate

This instruction reads all elements in the source predicate, reverses the order of all the elements, and places the result in the destination predicate. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 9	8 5	4	3 0
0 0 0	0 0 1 0	1	size	1	1 0 1 0 0	0 1 0	0 0 0 0	Pn	0	Pd

Encoding

```
REV  <Pd>.<T>, <Pn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let d : integer = UInt(Pd);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let operand : bits(PL) = P{}(n);
let result : bits(PL) = Reverse{}(operand, esize DIV 8);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.545 REV (vector)

Reverse all elements in a vector (unpredicated)

This instruction reads all elements in the source vector, reverses the order of all the elements, and places the result in the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	1 1	0 0 0	0 0 1 1 1 0	Zn	Zd

Encoding

```
REV  <Zd>.<T>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let operand : bits(VL) = Z{}(n);
let result : bits(VL) = Reverse{}(operand, esize);
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.546 REVB, REVH, REVW

Reverse bytes / halfwords / words within elements (predicated)

This instruction reverses the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 6 classes: Byte, merging, Byte, zeroing, Halfword, merging, Halfword, zeroing, Word, merging, and Word, zeroing

Byte, merging

(FEAT_SVE || FEAT_SME)

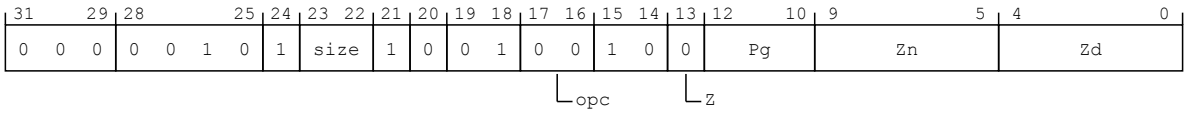

Encoding

```
REVB  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 8;
let merging : boolean = TRUE;
```

Byte, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

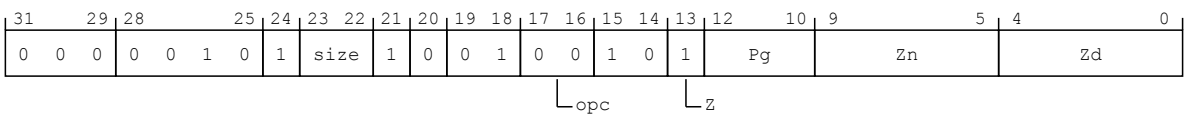

Encoding

```
REVB  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 8;
let merging : boolean = FALSE;
```

Halfword, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 18	17 16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 1	0 1	1 0	0	Pg	Zn	Zd
							opc		Z			

Encoding

```
REVH  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 16;
let merging : boolean = TRUE;
```

Halfword, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19 18	17 16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 1	0 1	1 0	1	Pg	Zn	Zd
							opc		Z			

Encoding

```
REVH  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 16;
let merging : boolean = FALSE;
```

Word, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 18	17 16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 1	1 0	1 0	0	Pg	Zn	Zd
							opc		Z			

Encoding

```
REVW  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 32;
let merging : boolean = TRUE;
```

Word, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19 18	17 16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	0 1	1 0	1 0	1	Pg	Zn	Zd
							opc		Z			

Encoding

```
REVW  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 32;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

For the “Byte, merging” and “Byte, zeroing” variants: is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

For the “Halfword, merging” and “Halfword, zeroing” variants: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = Reverse{esize}(element, swsize);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.547 REVD

Reverse 64-bit doublewords in elements (predicated)

This instruction reverses the order of 64-bit doublewords within each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SME || FEAT_SVE2p1)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	0 0	1	0	1 1 1	0	1 0	0	Pg	Zn	Zd
			size						Z			

Encoding

```
REVD  <Zd>.Q, <Pg>/M, <Zn>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 128;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 64;
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19–17	16	15–14	13	12–10	9–5	4–0
0 0 0	0 0 1 0	1	0 0	1	0	1 1 1	0	1 0	1	Pg	Zn	Zd
			size						Z			

Encoding

```
REVD  <Zd>.Q, <Pg>/Z, <Zn>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 128;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let swsize : integer = 64;
let merging : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = Reverse{esize}(element, swsize);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.548 RSHRNB

Rounding shift right narrow by immediate (bottom)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

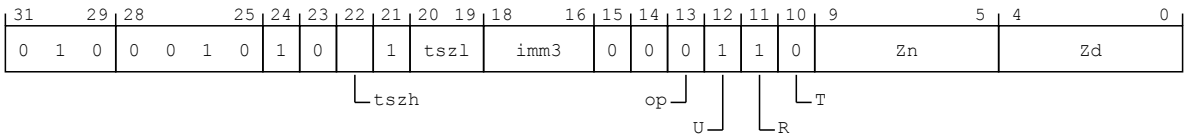

Encoding

```
RSHRNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 0)*:esize] = res[esize-1:0];
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.549 RSHRNT

Rounding shift right narrow by immediate (top)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

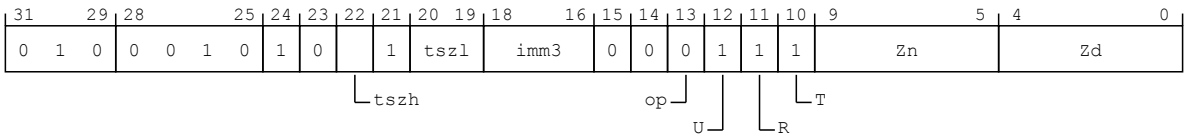

Encoding

```
RSHRNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 1)*:esize] = res[esize-1:0];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.550 RSUBHNB

Rounding subtract narrow high part (bottom)

This instruction subtracts each vector element of the second source vector from the corresponding vector element in the first source vector, and places the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered half-width destination elements to zero. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

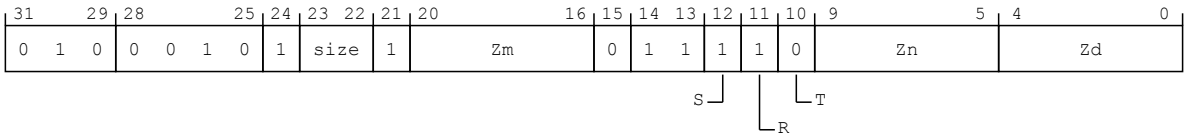

Encoding

```
RSUBHNB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = ((element1 - element2) + (1 << (halfesize - 1))) >> halfesize;
    result[(2*e + 0)*:halfesize] = res[halfesize-1:0];
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.551 RSUBHNT

Rounding subtract narrow high part (top)

This instruction subtracts each vector element of the second source vector from the corresponding vector element in the first source vector, and places the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

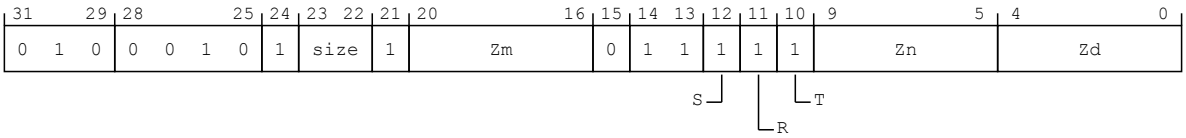

Encoding

```
RSUBHNT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = ((element1 - element2) + (1 << (halfesize - 1))) >> halfesize;
    result[(2*e + 1)*:halfesize] = res[halfesize-1:0];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.552 SABA

Signed absolute difference and accumulate

This instruction calculates the absolute difference between signed integer values in elements of the first source vector and the corresponding elements of the second source vector, and adds the difference to the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

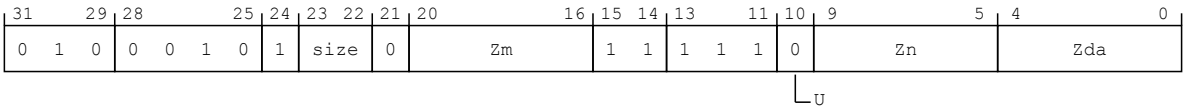

Encoding

```
SABA  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let absdiff : bits(esize) = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.553 SABALB

Signed absolute difference and accumulate long (bottom)

This instruction calculates the absolute difference between even-numbered signed integer values in elements of the first source vector and the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

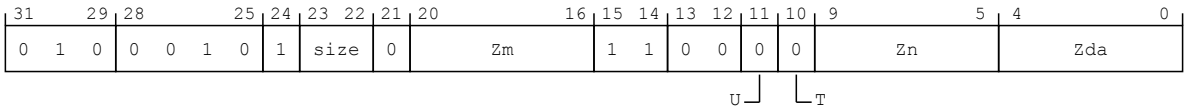

Encoding

```
SABALB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.554 SABALT

Signed absolute difference and accumulate long (top)

This instruction calculates the absolute difference between odd-numbered signed elements of the first source vector and the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

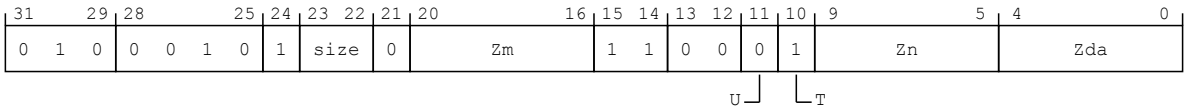

Encoding

```
SABALT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.555 SABD

Signed absolute difference (predicated)

This instruction calculates the absolute difference between signed integer values in active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0 1	1 0 (opc)	0 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
SABD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let absdiff : integer = Abs(element1 - element2);
        result[e*:esize] = absdiff[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.556 SABDLB

Signed absolute difference long (bottom)

This instruction calculates the absolute difference between even-numbered signed integer values in elements of the first source vector and the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

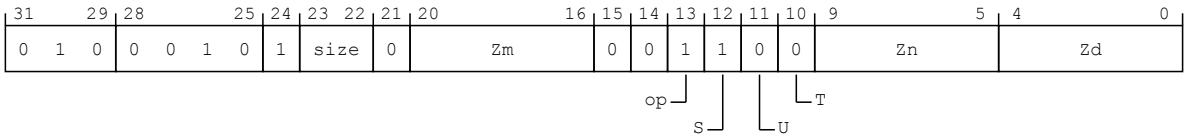

Encoding

```
SABDLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let res : integer = Abs(element1 - element2);
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.557 SABDLT

Signed absolute difference long (top)

This instruction calculates the absolute difference between odd-numbered signed integer values in elements of the first source vector and the corresponding elements of the second source vector, and places the results in overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

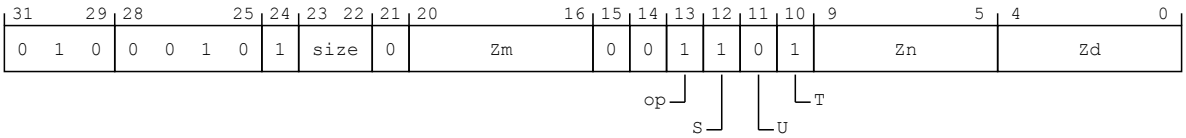

Encoding

```
SABDLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let res : integer = Abs(element1 - element2);
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.558 SADALP

Signed add and accumulate long pairwise

This instruction adds pairs of adjacent signed integer values and accumulates the results into the overlapping double-width elements of the destination vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	0 0 1 0	0	1 0	1	Pg	Zn	Zda
						U					

Encoding

```
SADALP  <Zda>.<T>, <Pg>/M, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the second source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand_acc : bits(VL) = Z{}(da);
let operand_src : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand_acc[e*:esize];
    else
        let element1 : integer = SInt(operand_src[(2*e + 0)*:(esize DIV 2)]);
        let element2 : integer = SInt(operand_src[(2*e + 1)*:(esize DIV 2)]);
        let sum : bits(esize) = (element1 + element2)[esize-1:0];
        result[e*:esize] = operand_acc[e*:esize] + sum;
    end;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.559 SADDLB

Signed add long (bottom)

This instruction adds the corresponding even-numbered signed elements of the first and second source vectors, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

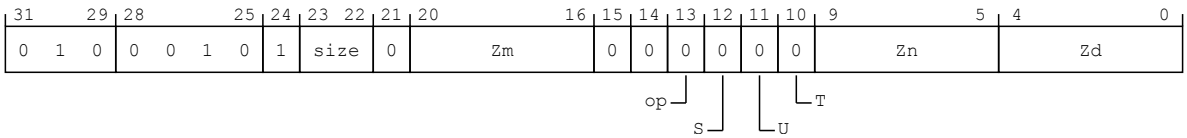

Encoding

```
SADDLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 0;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 + element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.560 SADDLBT

Signed add long (bottom + top)

This instruction adds the even-numbered signed elements of the first source vector to the odd-numbered signed elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

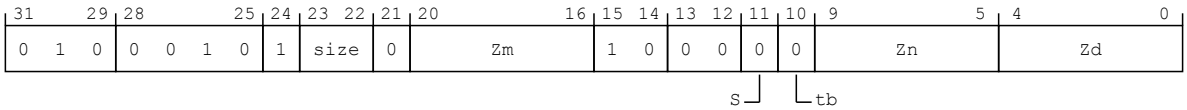

Encoding

```
SADDLBT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 1;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 + element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.561 SADDLT

Signed add long (top)

This instruction adds the corresponding odd-numbered signed elements of the first and second source vectors, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

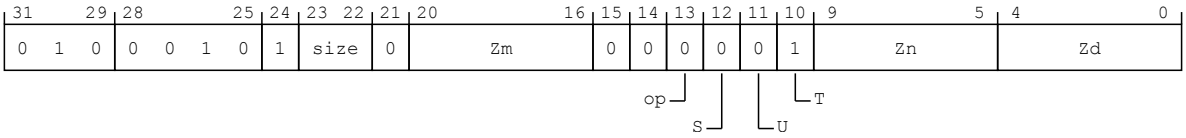

Encoding

```
SADDLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 1;
let sel2 : integer = 1;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 + element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.562 SADDV

Signed add reduction to scalar

This instruction adds all the signed elements of the source vector, and places the result in the SIMD&FP scalar destination register. Narrow elements are first sign-extended to 64 bits. Inactive elements in the source vector are treated as zero.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 0 0	0 (op)	0 (U)	0 0 1	Pg	Zn	Vd

Encoding

```
SADDV  <Dd>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Dd>

Is the 64-bit name of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = Z{}(n);
var sum : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand[e*:esize]);
        sum = sum + element;
    end;
end;

V{64}(d) = sum[63:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.563 SADDWB

Signed add wide (bottom)

This instruction adds the even-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

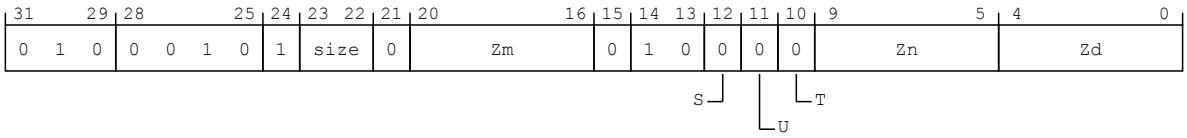

Encoding

```
SADDWB  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    result[e*:esize] = (element1 + element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.564 SADDWT

Signed add wide (top)

This instruction adds the odd-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

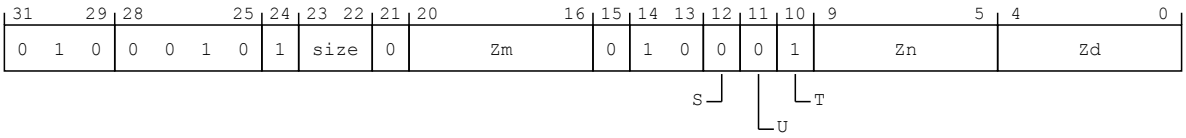

Encoding

```
SADDWT  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    result[e*:esize] = (element1 + element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.565 SBCLB

Subtract with carry long (bottom)

This instruction subtracts the even-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23	22	21	20 16	15 14	13 11	10	9 5	4 0
0 1 0	0 0 1 0	1	1	sz	0	Zm	1 1	0 1 0	0	Zn	Zda
									T		

Encoding

```
SBCLB  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (esize * 2);
let operand : bits(VL) = Z{}(n);
let carries : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for p = 0 to pairs-1 do
    let element1 : bits(esize) = result[(2*p + 0)*:esize];
    let element2 : bits(esize) = operand[(2*p + 0)*:esize];
    let carry_in : bit = carries[(2*p + 1)*:esize][0];

    let (res, nzcv) = AddWithCarry{esize}(element1, NOT(element2), carry_in);
    let carry_out : bit = nzcv[1];

    result[(2*p + 0)*:esize] = res;
    result[(2*p + 1)*:esize] = ZeroExtend{esize}(carry_out);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.566 SBCLT

Subtract with carry long (top)

This instruction subtracts the odd-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23	22	21	20 16	15 14	13 11	10	9 5	4 0
0 1 0	0 0 1 0	1	1	sz	0	Zm	1 1	0 1 0	1	Zn	Zda
									T		

Encoding

```
SBCLT  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (esize * 2);
let operand : bits(VL) = Z{}(n);
let carries : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for p = 0 to pairs-1 do
    let element1 : bits(esize) = result[(2*p + 0)*:esize];
    let element2 : bits(esize) = operand[(2*p + 1)*:esize];
    let carry_in : bit = carries[(2*p + 1)*:esize][0];

    let (res, nzcv) = AddWithCarry{esize}(element1, NOT(element2), carry_in);
    let carry_out : bit = nzcv[1];

    result[(2*p + 0)*:esize] = res;
    result[(2*p + 1)*:esize] = ZeroExtend{esize}(carry_out);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.567 SCLAMP

Signed clamp to minimum/maximum

This instruction clamps each signed element in the destination vector to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively writes the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SME || FEAT_SVE2p1)

31–29	28–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	Zm	1 1 0 0 0	0	Zn	Zd
							U		

Encoding

```
SCLAMP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(d);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    let element3 : integer = SInt(operand3[e*:esize]);
    let res : integer = Min(Max(element1, element3), element2);
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.568 SCVTF (predicated)

Signed integer convert to floating-point (predicated)

This instruction converts to floating-point from the signed integer in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 14 classes: 16-bit to half-precision, merging, 16-bit to half-precision, zeroing, 32-bit to half-precision, merging, 32-bit to half-precision, zeroing, 32-bit to single-precision, merging, 32-bit to single-precision, zeroing, 32-bit to double-precision, merging, 32-bit to double-precision, zeroing, 64-bit to half-precision, merging, 64-bit to half-precision, zeroing, 64-bit to single-precision, merging, 64-bit to single-precision, zeroing, 64-bit to double-precision, merging, and 64-bit to double-precision, zeroing

16-bit to half-precision, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 0	0 1	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.H, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

16-bit to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 0	0	1	1	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.H, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

32-bit to half-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 1	0	1 0	1 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.H, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

32-bit to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 0	1	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.H, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

32-bit to single-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 0	0	1 0	1 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.S, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

32-bit to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	0 1 1	1 0	1	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.S, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

32-bit to double-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 0	0 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.D, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

32-bit to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 0	0	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.D, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

64-bit to half-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 1	0	1 0	1 1	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.H, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

64-bit to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 0	1	1	1	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.H, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 16;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

64-bit to single-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 0	1 0	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

64-bit to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 0	1	1	0	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

64-bit to double-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 0	1 1	0	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
SCVTF  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

64-bit to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 0	1	1	1	0	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
SCVTF  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        let fpval : bits(d_esize) = FixedToFP{d_esize, s_esize}(element[s_esize-1:0], 0, unsigned,
                                                                FPCR(), rounding);
        result[e*:esize] = ZeroExtend{esize}(fpval);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the 16-bit to half-precision, merging, 32-bit to half-precision, merging, 32-bit to single-precision, merging, 32-bit to double-precision, merging, 64-bit to half-precision, merging, 64-bit to single-precision, merging, 64-bit to double-precision, merging variants:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.569 SDIV

Signed divide (predicated)

This instruction performs a signed divide on active elements of the first source vector by the corresponding elements of the second source vector and destructively places the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

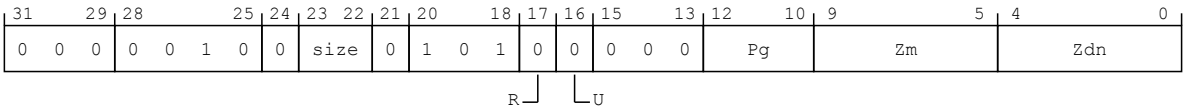

Encoding

```
SDIV  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let dividend : integer = SInt(operand1[e*:esize]);
        let divisor : integer = SInt(operand2[e*:esize]);
        var quotient : integer;
        if divisor == 0 then
            quotient = 0;
        elsif (dividend < 0) == (divisor < 0) then
            quotient = Abs(dividend) DIVRM Abs(divisor);    // same signs - positive result
        else
            quotient = -(Abs(dividend) DIVRM Abs(divisor)); // different signs - negative result
        end;
        result[e*:esize] = quotient[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.570 SDIVR

Signed reversed divide (predicated)

This instruction performs a signed reversed divide on active elements of the second source vector by the corresponding elements of the first source vector, and destructively places the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 0 1	1 (R)	0 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
SDIVR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let divisor : integer = SInt(operand1[e*:esize]);
        let dividend : integer = SInt(operand2[e*:esize]);
        var quotient : integer;
        if divisor == 0 then
            quotient = 0;
        elsif (dividend < 0) == (divisor < 0) then
            quotient = Abs(dividend) DIVRM Abs(divisor);    // same signs - positive result
        else
            quotient = -(Abs(dividend) DIVRM Abs(divisor)); // different signs - negative result
        end;
        result[e*:esize] = quotient[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.571 SDOT (2-way, vectors)

Signed integer dot product (two-way)

This instruction calculates the dot product of a pair of signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a pair of signed 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

16-bit to 32-bit

(FEAT_SME2 || FEAT_SVE2p1)

31 29	28 25	24 21	20 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0 0 0 0	Zm	1 1 0 0 1	0	Zn	Zda
					U		

Encoding

```
SDOT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 1 do
        let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
        let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.572 SDOT (2-way, indexed)

Signed integer dot product by indexed element (two-way)

This instruction calculates the dot product of a pair of signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a pair of signed 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

16-bit to 32-bit

(FEAT_SME2 || FEAT_SVE2p1)

31–29	28–25	24–21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0 1 0 0	i2	Zm	1 1 0 0 1	0	Zn	Zda
						U		

Encoding

```
SDOT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 1 do
        let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
        let element2 : integer = SInt(operand2[(2 * s + i)*:(esize DIV 2)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.573 SDOT (4-way, vectors)

Signed integer dot product (four-way)

This instruction calculates the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

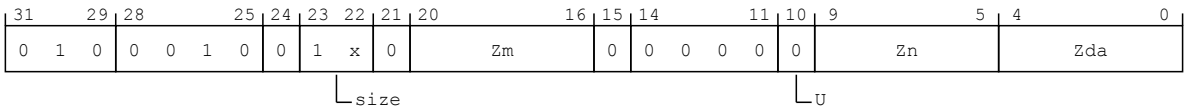

Encoding

```
SDOT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size<0>”:

size<0>	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.574 SDOT (4-way, indexed)

Signed integer dot product by indexed element (four-way)

This instruction calculates the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 32-bit and 16-bit to 64-bit

8-bit to 32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 0 0	0	Zn	Zda
			size					U		

Encoding

```
SDOT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

16-bit to 64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 0 0	0	Zn	Zda
			size					U		

Encoding

```
SDOT  <Zda>.D, <Zn>.H, <Zm>.H[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “8-bit to 32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “16-bit to 64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “8-bit to 32-bit” variant: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “16-bit to 64-bit” variant: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        let element2 : integer = SInt(operand2[(4 * s + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.575 SEL (predicates)

Conditionally select elements from two predicates

This instruction reads Active elements from the first source predicate and Inactive elements from the second source predicate and places them in the corresponding elements of the destination predicate. This instruction does not set the condition flags.

This instruction is used by the alias MOV (predicate, merging).

SVE

(FEAT_SVE || FEAT_SME)

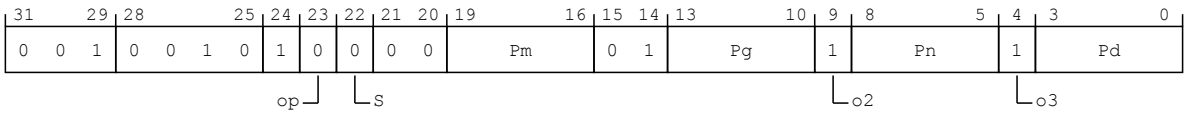

Encoding

```
SEL  <Pd>.B, <Pg>, <Pn>.B, <Pm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<Pg>

Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Alias Conditions

Alias	Is preferred when
MOV (predicate, merging)	Pd == Pm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

for e = 0 to elements-1 do
    let element1 : bit = PredicateElement{PL}(operand1, e, esize);
    let element2 : bit = PredicateElement{PL}(operand2, e, esize);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:psize] = ZeroExtend{psize}(element1);
    else
        result[e*:psize] = ZeroExtend{psize}(element2);
    end;
end;

P{PL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.576 SEL (vectors)

Conditionally select elements from two vectors

This instruction selects elements from the first source vector where the corresponding vector select predicate element is true, and from the second source vector where the predicate element is false, and places them in the corresponding elements of the destination vector.

This instruction is used by the alias MOV (vector, predicated).

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 14	13 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	Zm	1 1	Pv	Zn	Zd

Encoding

```
SEL  <Zd>.<T>, <Pv>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let v : integer = UInt(Pv);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv>

Is the name of the vector select predicate register, encoded in the “Pv” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Alias Conditions

Alias	Is preferred when
MOV (vector, predicated)	Zd == Zm

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(v);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
let operand2 : bits(VL) = if AnyActiveElement{PL}(NOT(mask), esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[e*:esize];
    else
        result[e*:esize] = operand2[e*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.577 SETFFR

Initialize the first-fault register to all true

This instruction initializes the first-fault register (`FFR`) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31 29	28 25	24	23 22	21 19	18	17 16	15 12	11 9	8 5	4 0
0 0 1	0 0 1 0	1	0 0	1 0 1	1	0 0	1 0 0 1	0 0 0	0 0 0 0	0 0 0 0 0
			opc							

Encoding

```
SETFFR
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
FFR{PL}() = Ones{PL};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.578 SHADD

Signed halving add

This instruction adds active signed elements of the first source vector to the corresponding signed elements of the second source vector, shifts right one bit, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

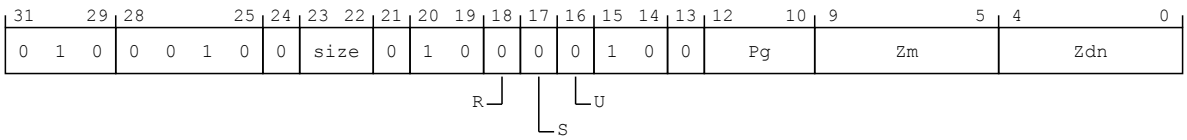

Encoding

```
SHADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 + element2) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.579 SHRNB

Shift right narrow by immediate (bottom)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

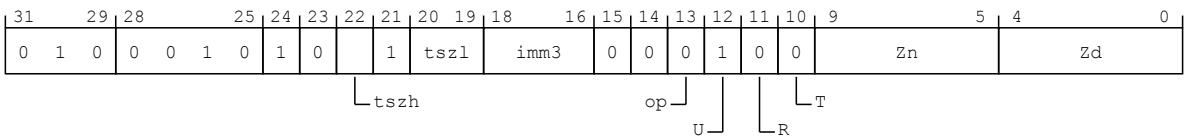

Encoding

```
SHRNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = UInt(element) >> shift;
    result[(2*e + 0)*:esize] = res[esize-1:0];
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.580 SHRNT

Shift right narrow by immediate (top)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

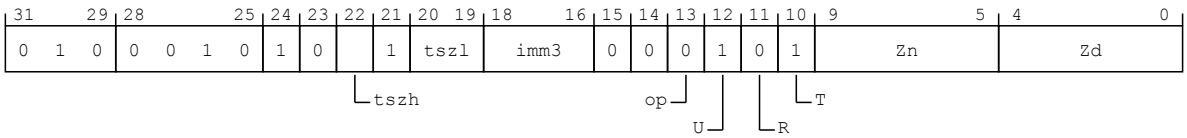

Encoding

```
SHRNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = UInt(element) >> shift;
    result[(2*e + 1)*:esize] = res[esize-1:0];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.581 SHSUB

Signed halving subtract

This instruction subtracts active signed elements of the second source vector from the corresponding signed elements of the first source vector, shifts right one bit, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

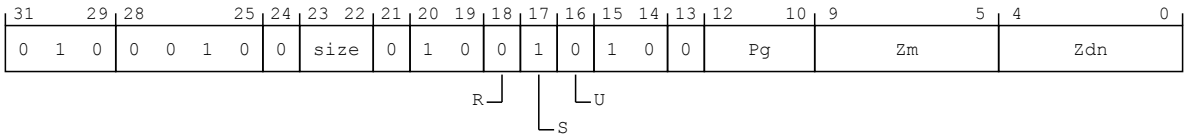

Encoding

```
SHSUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 - element2) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.582 SHSUBR

Signed halving subtract reversed

This instruction subtracts active signed elements of the first source vector from corresponding signed elements of the second source vector, shifts right one bit, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

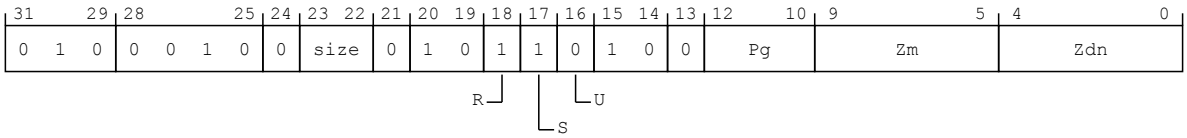

Encoding

```
SHSUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element2 - element1) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.583 SLI

Shift left and insert (immediate)

This instruction shifts each source vector element left by an immediate value, and inserts the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 14	13 11	10	9 5	4 0
0 1 0	0 0 1 0	1	tszh	0	tszl	imm3	1 1	1 1 0	1	Zn	Zd
									op		

Encoding

```
SLI  <Zd>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element1 : bits(esize) = result[e*:esize];
    let element2 : bits(esize) = operand[e*:esize];
    let mask : bits(esize) = LSL(Ones{esize}, shift);
    let shiftedval : bits(esize) = LSL(element2, shift);
    result[e*:esize] = (element1 AND (NOT mask)) OR shiftedval;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.584 SM4E

SM4 encryption and decryption

This instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, together with four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each block of data is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_SM4)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–11	10	9–5	4–0
0 1 0	0 0 1 0	1	0 0	1	0 0 0	1	1	1 1 1	0 0	0	Zm	Zdn
			size				op			o2		

Encoding

```
SM4E  <Zdn>.S, <Zdn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_SM4) then EndOfDecode(Decode_UNDEF); end;
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let key : bits(128) = operand2[s*:128];
    var intval : bits(32);
    var roundresult : bits(128) = operand1[s*:128];
    var roundkey : bits(32);

    for index = 0 to 3 do
        roundkey = key[index*:32];
        intval = roundresult[127:96] XOR roundresult[95:64] XOR roundresult[63:32] XOR roundkey;

        for i = 0 to 3 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            intval[i*:8]  = Sbox(intval[i*:8]);
        end;

        intval = (intval XOR
                  ROL(intval, 2) XOR
                  ROL(intval, 10) XOR
                  ROL(intval, 18) XOR
                  ROL(intval, 24));
        intval = intval XOR roundresult[31:0];

        roundresult[31:0] = roundresult[63:32];
        roundresult[63:32] = roundresult[95:64];
        roundresult[95:64] = roundresult[127:96];
        roundresult[127:96] = intval;
    end;

    result[s*:128] = roundresult;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.585 SM4EKEY

SM4 key updates

This instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, along with four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE_SM4)

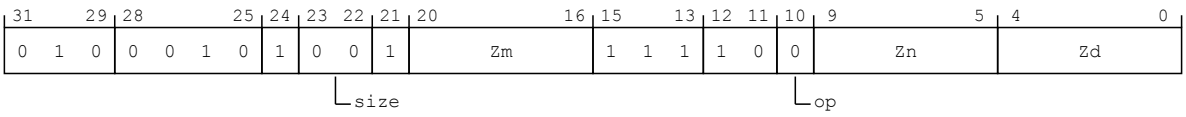

Encoding

```
SM4EKEY  <Zd>.S, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE_SM4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let source : bits(128) = operand2[s*:128];
    var intval : bits(32);
    var const : bits(32);
    var roundresult : bits(128) = operand1[s*:128];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for index = 0 to 3 do
        const = source[index*:32];
        intval = roundresult[127:96] XOR roundresult[95:64] XOR roundresult[63:32] XOR const;
        for i = 0 to 3 do
            intval[i*:8] = Sbox(intval[i*:8]);
        end;

        intval = intval XOR ROL(intval, 13) XOR ROL(intval, 23);
        intval = intval XOR roundresult[31:0];

        roundresult[31:0] = roundresult[63:32];
        roundresult[63:32] = roundresult[95:64];
        roundresult[95:64] = roundresult[127:96];
        roundresult[127:96] = intval;
    end;

    result[s*:128] = roundresult;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.586 SMAX (vectors)

Signed maximum (predicated)

This instruction determines the signed maximum of active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 1	0 0 (opc)	0 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
SMAX  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let maximum : integer = Max(element1, element2);
        result[e*:esize] = maximum[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.587 SMAX (immediate)

Signed maximum with immediate (unpredicated)

This instruction determines the signed maximum of an immediate and each element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

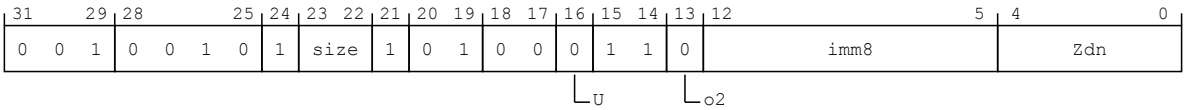

Encoding

```
SMAX  <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let unsigned : boolean = FALSE;
let imm : integer = if unsigned then UInt(imm8) else SInt(imm8);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is the signed immediate operand, in the range -128 to 127, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    result[e*:esize] = Max(element1, imm)[esize-1:0];

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.588 SMAXP

Signed maximum pairwise

This instruction calculates the maximum value of each pair of adjacent signed integer elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	1 0	1 0 (opc)	0 (U)	1 0	1	Pg	Zm	Zdn

Encoding

```
SMAXP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[e*:esize];
    else
        if IsEven(e) then
            element1 = SInt(operand1[(e + 0)*:esize]);
            element2 = SInt(operand1[(e + 1)*:esize]);
        else
            element1 = SInt(operand2[(e - 1)*:esize]);
            element2 = SInt(operand2[(e + 0)*:esize]);
        end;
        let res : integer = Max(element1, element2);
        result[e*:esize] = res[esize-1:0];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.589 SMAXQV

Signed maximum reduction of quadword vector segments

This instruction determines the signed maximum of the elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

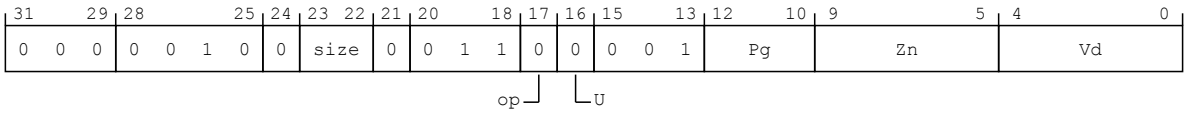

Encoding

```
SMAXQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : integer;

for e = 0 to elempersegment-1 do
    dtmp = if unsigned then 0 else -(2^(esize-1));
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = Max(dtmp, SInt(stmp[e*:esize]));
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.590 SMAXV

Signed maximum reduction to scalar

This instruction determines the signed maximum of the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

SVE

(FEAT_SVE || FEAT_SME)

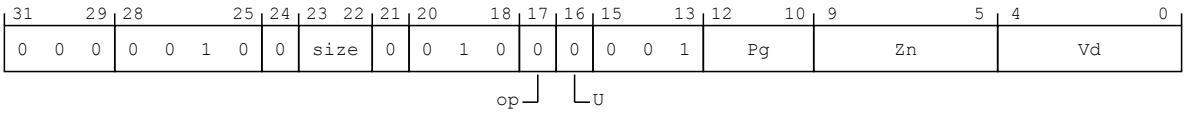

Encoding

```
SMAXV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var maximum : integer = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let opelt : bits(esize) = operand[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        maximum = Max(maximum, element);

    end;
end;
V{esize}(d) = maximum[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.591 SMIN (vectors)

Signed minimum (predicated)

This instruction determines the signed minimum of active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

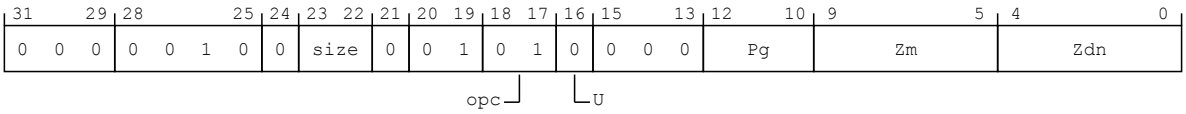

Encoding

```
SMIN  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let minimum : integer = Min(element1, element2);
        result[e*:esize] = minimum[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.592 SMIN (immediate)

Signed minimum with immediate (unpredicated)

This instruction determines the signed minimum of an immediate and each element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

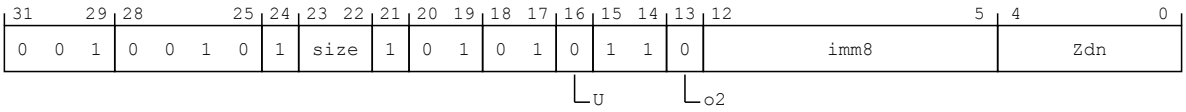

Encoding

```
SMIN  <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let unsigned : boolean = FALSE;
let imm : integer = if unsigned then UInt(imm8) else SInt(imm8);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is the signed immediate operand, in the range -128 to 127, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    result[e*:esize] = Min(element1, imm)[esize-1:0];

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.593 SMINP

Signed minimum pairwise

This instruction calculates the minimum value of each pair of adjacent signed integer elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 14	13	12 10	9 5	4 0
0 1 0	0 0 1 0	0	size	0	1 0	1 1	0	1 0	1	Pg	Zm	Zdn
						opc	U					

Encoding

```
SMINP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[e*:esize];
    else
        if IsEven(e) then
            element1 = SInt(operand1[(e + 0)*:esize]);
            element2 = SInt(operand1[(e + 1)*:esize]);
        else
            element1 = SInt(operand2[(e - 1)*:esize]);
            element2 = SInt(operand2[(e + 0)*:esize]);
        end;
        let res : integer = Min(element1, element2);
        result[e*:esize] = res[esize-1:0];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.594 SMINQV

Signed minimum reduction of quadword vector segments

This instruction determines the signed minimum of the elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 18	17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0 1 1	1	0	0 0 1	Pg	Zn	Vd
						op	U				

Encoding

```
SMINQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : integer;

for e = 0 to elempersegment-1 do
    dtmp = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = Min(dtmp, SInt(stmp[e*:esize]));
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.595 SMINV

Signed minimum reduction to scalar

This instruction determines the signed minimum of the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 1 0	1 (op)	0 (U)	0 0 1	Pg	Zn	Vd

Encoding

```
SMINV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var minimum : integer = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let opelt : bits(esize) = operand[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        minimum = Min(minimum, element);

    end;
end;
V{esize}(d) = minimum[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.596 SMLALB (vectors)

Signed multiply-add long (bottom)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

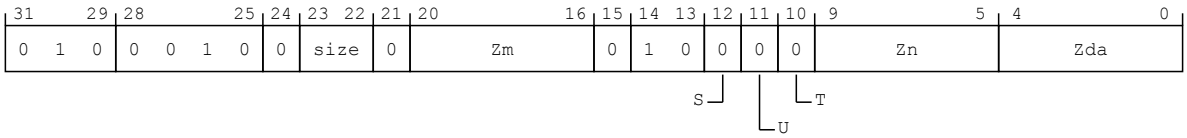

Encoding

```
SMLALB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.597 SMLALB (indexed)

Signed multiply-add long by indexed element (bottom)

This instruction multiplies the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and destructively adds the results to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

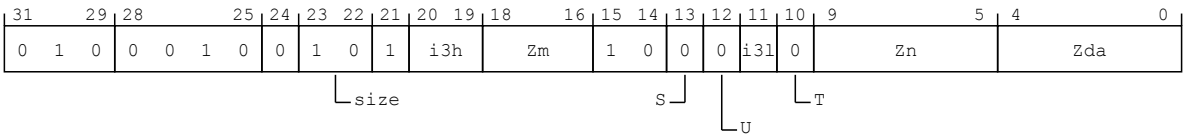

Encoding

```
SMLALB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

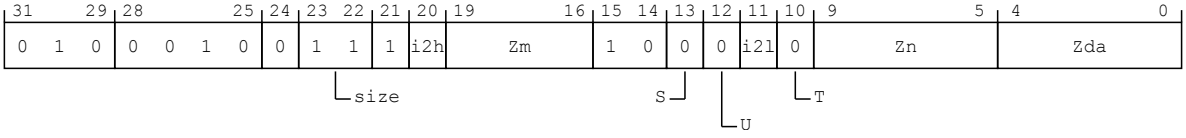

Encoding

```
SMLALB  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.598 SMLALT (vectors)

Signed multiply-add long (top)

This instruction multiplies the odd-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

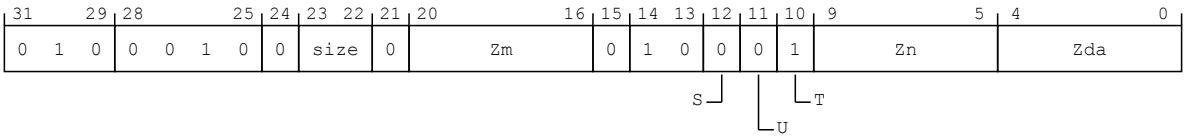

Encoding

```
SMLALT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.599 SMLALT (indexed)

Signed multiply-add long by indexed element (top)

This instruction multiplies the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and destructively adds the results to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

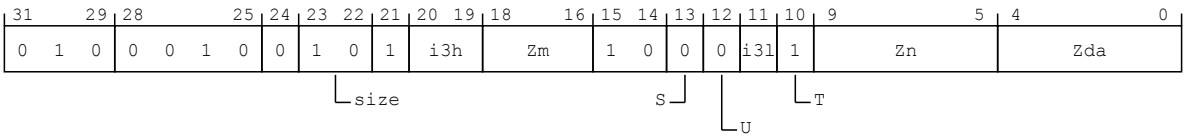

Encoding

```
SMLALT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

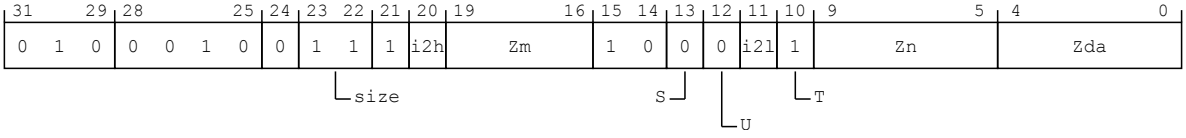

Encoding

```
SMLALT  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.600 SMLSLB (vectors)

Signed multiply-subtract long (bottom)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and destructively subtracts the results from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

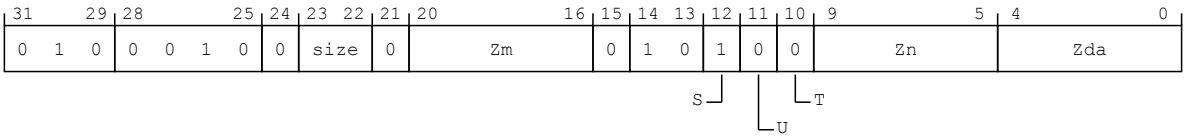

Encoding

```
SMLSLB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.601 SMLSLB (indexed)

Signed multiply-subtract long by indexed element (bottom)

This instruction multiplies the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and destructively subtracts the results from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

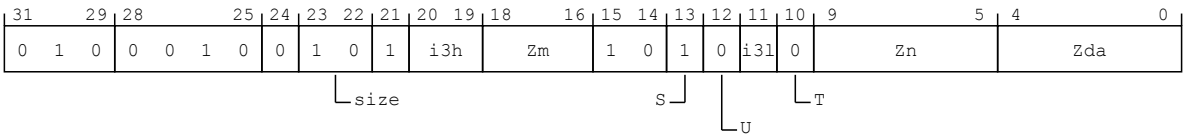

Encoding

```
SMLSLB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

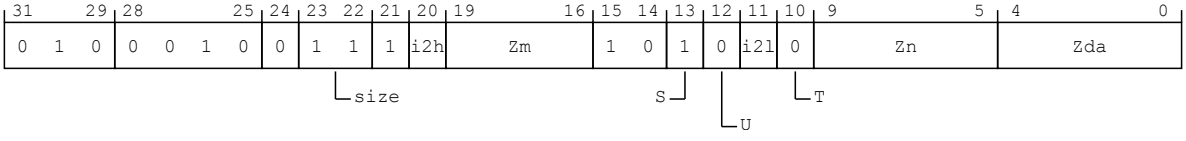

Encoding

```
SMLSLB  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.602 SMLSLT (vectors)

Signed multiply-subtract long (top)

This instruction multiplies the odd-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and destructively subtracts the results from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

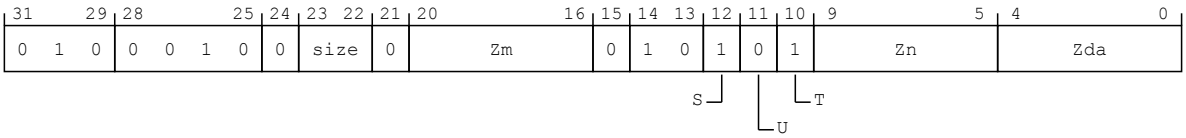

Encoding

```
SMLSLT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.603 SMLSLT (indexed)

Signed multiply-subtract long by indexed element (top)

This instruction multiplies the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and destructively subtracts the results from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

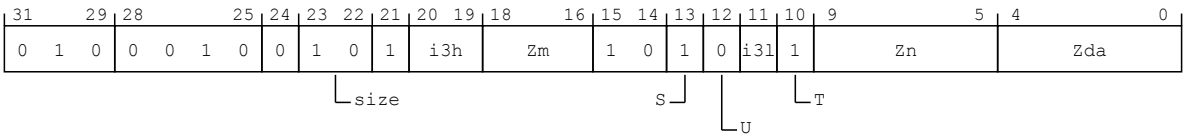

Encoding

```
SMLSLT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

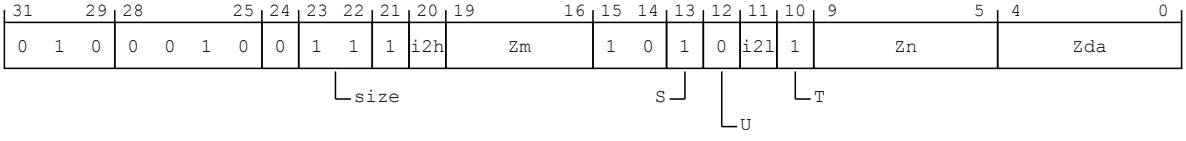

Encoding

```
SMLSLT  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.604 SMMLA

Signed 8-bit integer matrix multiply-accumulate to 32-bit integer

This instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE && FEAT_I8MM)

31–29	28–25	24	23–22	21	20–16	15–14	13–10	9–5	4–0
0 1 0	0 0 1 0	1	0 0 (uns)	0	Zm	1 0	0 1 1 0	Zn	Zda

Encoding

```
SMMLA  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_I8MM) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL) = Zeros{};
var op1, op2 : bits(128);
var res, addend : bits(128);

for s = 0 to segments-1 do
    op1    = operand1[s*:(128)];
    op2    = operand2[s*:(128)];
    addend = operand3[s*:(128)];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    result[s*:(128)] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.605 SMULH (predicated)

Signed multiply returning high half (predicated)

This instruction performs a widening multiplication of signed integer values in active elements of the first source vector by the corresponding elements of the second source vector and destructively places the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 0 0	1 (H)	0 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
SMULH  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let product : integer = (element1 * element2) >> esize;
        result[e*:esize] = product[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.606 SMULH (unpredicated)

Signed multiply returning high half (unpredicated)

This instruction performs a widening multiplication of signed integer values of all elements of the first source vector by the corresponding elements of the second source vector and places the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

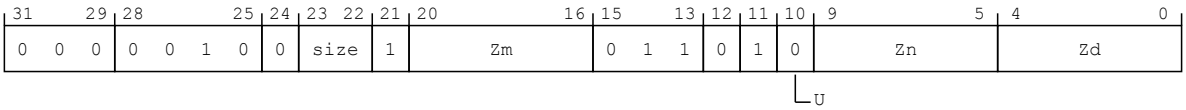

Encoding

```
SMULH  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let product : integer = (element1 * element2) >> esize;
    result[e*:esize] = product[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.607 SMULLB (vectors)

Signed multiply long (bottom)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

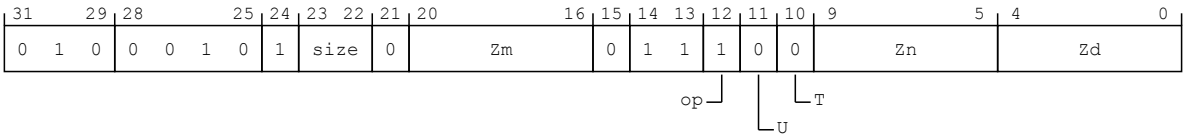

Encoding

```
SMULLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let res : integer = element1 * element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.608 SMULLB (indexed)

Signed multiply long by indexed element (bottom)

This instruction multiplies the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and places the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

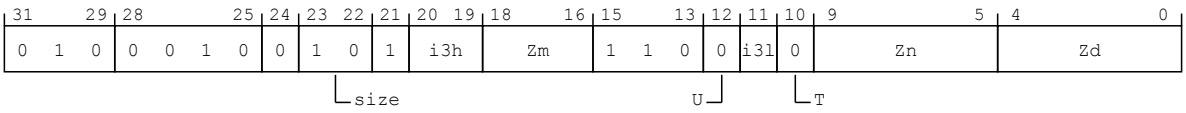

Encoding

```
SMULLB  <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

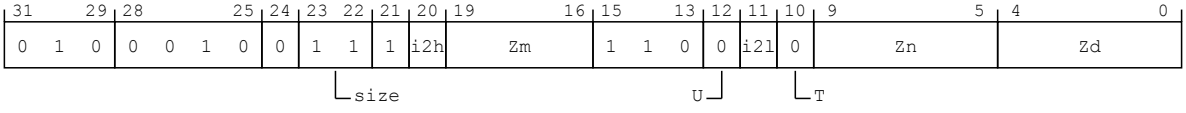

Encoding

```
SMULLB  <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 0;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let res : integer = element1 * element2;
    result[e*:(2*esize)] = res[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.609 SMULLT (vectors)

Signed multiply long (top)

This instruction multiplies the odd-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

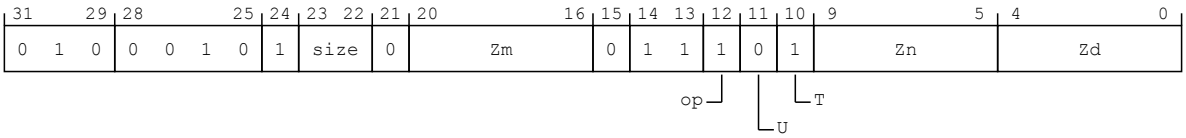

Encoding

```
SMULLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let res : integer = element1 * element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.610 SMULLT (indexed)

Signed multiply long by indexed element (top)

This instruction multiplies the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment, and places the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i3h	Zm	1 1 0	0	i3l	1	Zn	Zd
			size					U		T		

Encoding

```
SMULLT  <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1	1	i2h	Zm	1 1 0	0	i2l	1	Zn	Zd
			size					U		T		

Encoding

```
SMULLT  <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let res : integer = element1 * element2;
    result[e*:(2*esize)] = res[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.611 SPLICE

Splice two vectors under predicate control

This instruction selects a region from the first source vector and copies it to the lowest-numbered elements of the result. Any remaining elements of the result are set to a copy of the lowest-numbered elements from the second source vector. The region is selected using the first and last true elements in the vector select predicate register. The result is destructively placed in the destination and first source vector, or constructively placed in the destination vector.

It has encodings from 2 classes: Constructive and Destructive

Constructive

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	1 1 0	1	1 0	0	Pv	Zn	Zd

Encoding

```
SPLICE  <Zd>.<T>, <Pv>, { <Zn1>.<T>, <Zn2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let v : integer = UInt(Pv);
let dst : integer = UInt(Zd);
let s1 : integer = UInt(Zn);
let s2 : integer = (s1 + 1) MOD 32;
```

Destructive

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 17	16	15 14	13	12 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	0	1 1 0	0	1 0	0	Pv	Zm	Zdn

Encoding

```
SPLICE  <Zdn>.<T>, <Pv>, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let v : integer = UInt(Pv);
let dst : integer = UInt(Zdn);
let s1 : integer = dst;
let s2 : integer = UInt(Zm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv>

Is the name of the vector select predicate register P0-P7, encoded in the “Pv” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded in the “Zn” field.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded in the “Zn” field.

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(v);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(s1) else Zeros{VL};
let operand2 : bits(VL) = Z{}(s2);
var result : bits(VL);
var x : integer = 0;
var active : boolean = FALSE;
let lastnum : integer = LastActiveElement{PL}(mask, esize);

if lastnum >= 0 then
    for e = 0 to lastnum do
        active = active || ActivePredicateElement{PL}(mask, e, esize);
        if active then
            result[x*:esize] = operand1[e*:esize];
            x = x + 1;
        end;
    end;
end;

let nelements : integer = (elements - x) - 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to nelements do
    result[x*:esize] = operand2[e*:esize];
    x = x + 1;
end;

Z{VL}(dst) = result;
```

Operational Information

For the Destructive variant:

The destructive variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the destructive variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.612 SQABS

Signed saturating absolute value

This instruction calculates the absolute value of the signed integer in each Active element of the source vector, and places the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18	17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	0	1	0	0	0	1 0	1	Pg	Zn	Zd

Q, Z, op

Encoding

```
SQABS  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18	17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	0	1	0	1	0	1 0	1	Pg	Zn	Zd

Q, Z, op

Encoding

```
SQABS  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        var element : integer = SInt(operand[e*:esize]);
        element = Abs(element);
        result[e*:esize] = SignedSat{esize}(element);

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.613 SQADD (vectors, predicated)

Signed saturating add (predicated)

This instruction adds active signed elements of the first source vector to corresponding signed elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

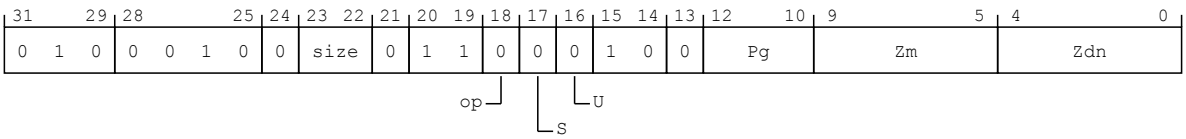

Encoding

```
SQADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = SignedSat{esize}(element1 + element2);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.614 SQADD (immediate)

Signed saturating add immediate (unpredicated)

This instruction performs a signed saturating addition of an unsigned immediate to each element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0 0 1	0 0 1 0	1	size	1	0 0	1 0	0	1 1	sh	imm8	Zdn
							U				

Encoding

```
SQADD  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    (result[e*:esize], -) = SatQ{esize}(element1 + imm, unsigned);

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.615 SQADD (vectors, unpredicated)

Signed saturating add (unpredicated)

This instruction performs a signed saturating addition of all elements of the first source vector to the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 0 0	1 0	0	Zn	Zd
								U		

Encoding

```
SQADD  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    if unsigned then
        let element1 : integer = UInt(op1elt);
        let element2 : integer = UInt(op2elt);
        (result[e*:esize], -) = UnsignedSatQ{esize}(element1 + element2);
    else
        let element1 : integer = SInt(op1elt);
        let element2 : integer = SInt(op2elt);
        (result[e*:esize], -) = SignedSatQ{esize}(element1 + element2);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.616 SQCADD

Saturating complex integer add

This instruction adds the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector that have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. The results are destructively placed in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

SVE2

(FEAT_SVE2 || FEAT_SME)

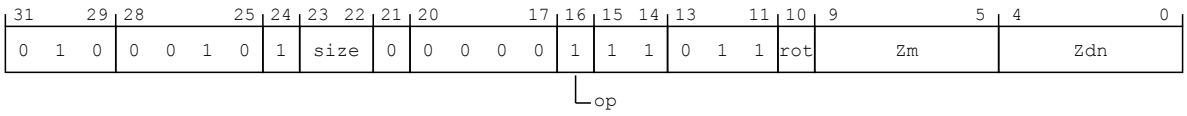

Encoding

```
SQCADD  <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
let sub_i : boolean = (rot == '0');
let sub_r : boolean = (rot == '1');
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
0	#90
1	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for p = 0 to pairs-1 do
    var acc_r : integer  = SInt(operand1[(2 * p + 0)*:esize]);
    var acc_i : integer  = SInt(operand1[(2 * p + 1)*:esize]);
    let elt2_r : integer = SInt(operand2[(2 * p + 0)*:esize]);
    let elt2_i : integer = SInt(operand2[(2 * p + 1)*:esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    end;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;
    end;

    result[(2 * p + 0)*:esize] = SignedSat{esize}(acc_r);
    result[(2 * p + 1)*:esize] = SignedSat{esize}(acc_i);
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.617 SQCVTN

Signed 32-bit integer saturating extract narrow to interleaved 16-bit integer

This instruction saturates the signed integer value in each element of the pair of source vectors to half the original source element width, and places the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

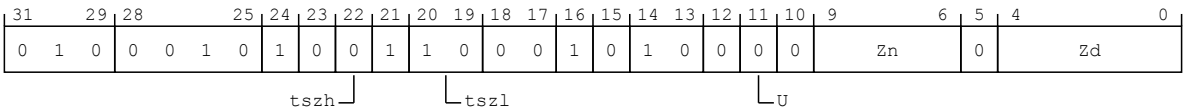

Encoding

```
SQCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 1 do
        let operand : bits(VL) = Z{}(n+i);
        let element : integer = SInt(operand[e*:(2 * esize)]);
        result[(2*e + i)*:esize] = SignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.618 SQCVTUN

Signed 32-bit integer saturating extract narrow to interleaved unsigned 16-bit integer

This instruction saturates the signed integer value in each element of the pair of source vectors to unsigned integer value that is half the original source element width, and places the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

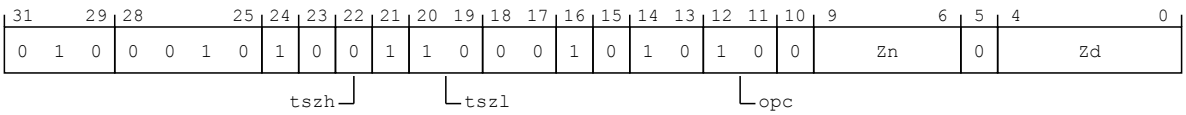

Encoding

```
SQCVTUN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 1 do
        let operand : bits(VL) = Z{}(n+i);
        let element : integer = SInt(operand[e*:(2 * esize)]);
        result[(2*e + i)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.619 SQDECB

Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count

This instruction determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

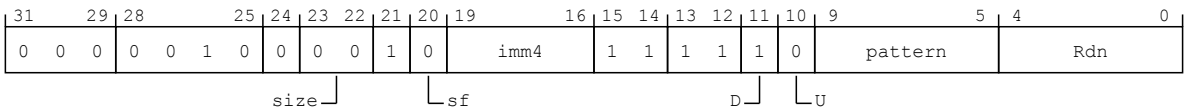

Encoding

```
SQDECB  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

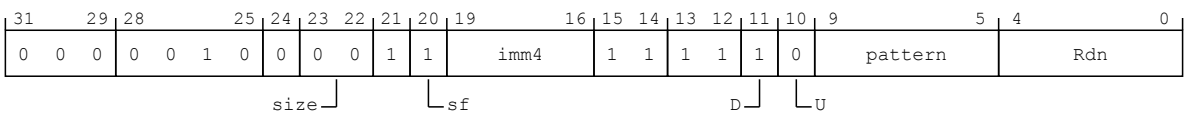

Encoding

```
SQDECB  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.620 SQDECD (scalar)

Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

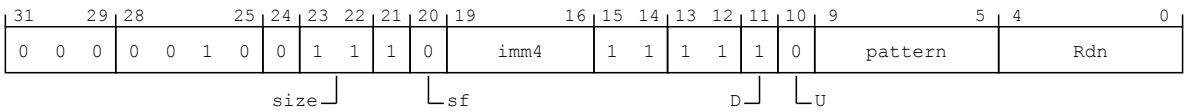

Encoding

```
SQDECD  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

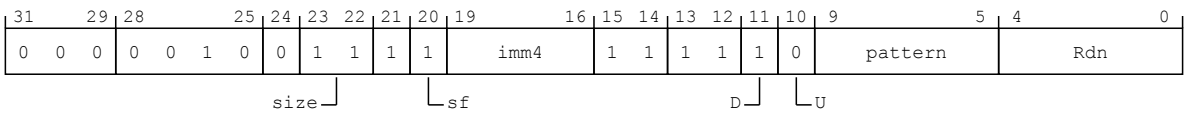

Encoding

```
SQDECD  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.621 SQDECD (vector)

Signed saturating decrement vector by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit
signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

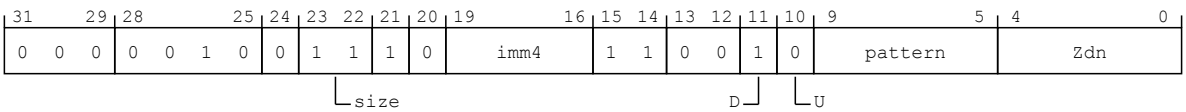

Encoding

```
SQDECD  <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) - (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) - (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.622 SQDECH (scalar)

Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

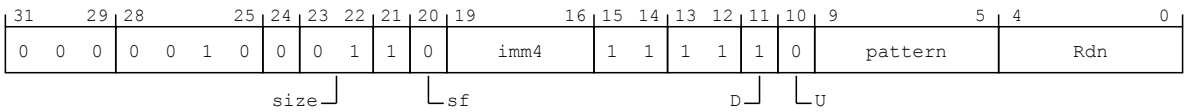

Encoding

```
SQDECH  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

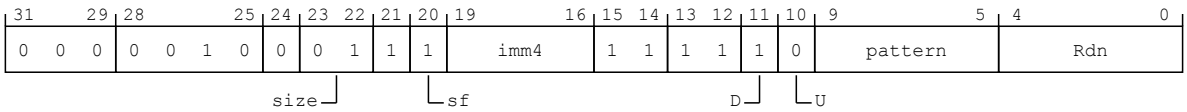

Encoding

```
SQDECH  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.623 SQDECH (vector)

Signed saturating decrement vector by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit
signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

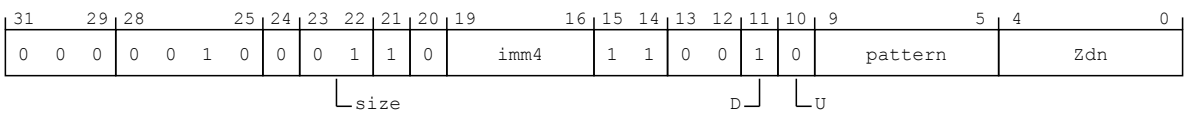

Encoding

```
SQDECH  <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) - (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) - (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.624 SQDECP (scalar)

Signed saturating decrement scalar by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

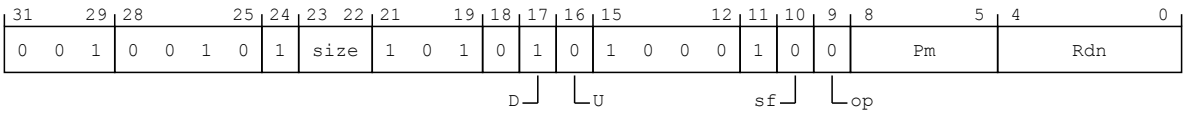

Encoding

```
SQDECP  <Xdn>, <Pm>.<T>, <Wdn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

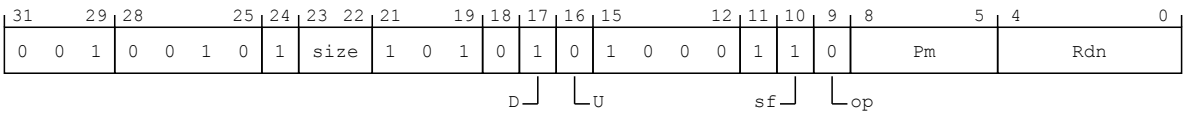

Encoding

```
SQDECP  <Xdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(ssize) = X{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(ssize);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - count);
    X{64}(dn) = ZeroExtend{64}(result);
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - count);
    X{64}(dn) = SignExtend{64}(result);
end;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.625 SQDECP (vector)

Signed saturating decrement vector by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21–19	18	17	16	15–12	11	10–9	8–5	4–0
0 0 1	0 0 1 0	1	size	1 0 1	0	1 (D)	0 (U)	1 0 0 0	0	0 0 (opc)	Pm	Zdn

Encoding

```
SQDECP  <Zdn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Zdn);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(VL);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(opelt) - count);
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(opelt) - count);
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.626 SQDECW (scalar)

Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 0	1	0	imm4	1 1	1 1	1	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQDECW  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 0	1	1	imm4	1 1	1 1	1	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQDECW  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.627 SQDECW (vector)

Signed saturating decrement vector by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit
signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

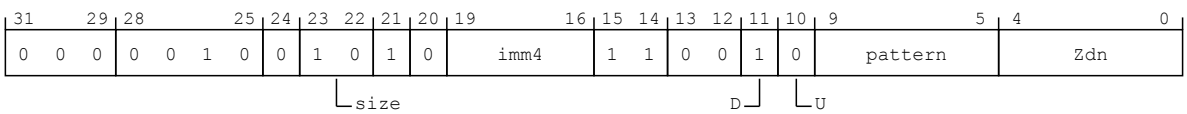

Encoding

```
SQDECW  <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) - (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) - (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.628 SQDMLALB (vectors)

Signed saturating doubling multiply-add (bottom)

This instruction multiples the even-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result of the multiplication. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively added to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14–12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	Zm	0	1 1 0	0 (S)	0 (T)	Zn	Zda

Encoding

```
SQDMLALB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel1 : integer = 0;
let sel2 : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2 * e + sel1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2 * e + sel2)*:(esize DIV 2)]);
    let element3 : integer = SInt(result[e*:esize]);
    let product : integer = SInt(SignedSat{esize}(2 * element1 * element2));
    result[e*:esize] = SignedSat{esize}(element3 + product);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.629 SQDMLALB (indexed)

Signed saturating doubling multiply-add by indexed element (bottom)

This instruction multiples the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and then doubles the result of the multiplication. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively added to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

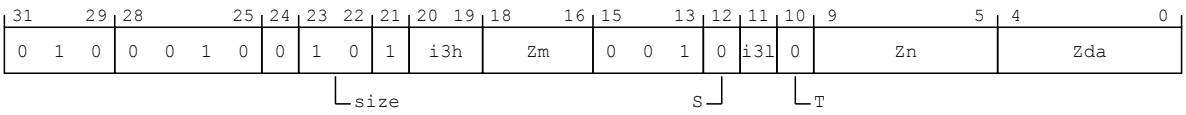

Encoding

```
SQDMLALB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

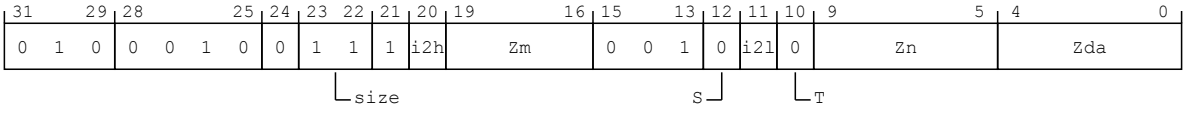

Encoding

```
SQDMLALB  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let element3 : integer = SInt(result[e*:(2*esize)]);
    let product : integer = SInt(SignedSat{2*esize}(2 * element1 * element2));
    let res : integer = element3 + product;
    result[e*:(2*esize)] = SignedSat{2*esize}(res);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.630 SQDMLALBT

Signed saturating doubling multiply-add (bottom × top)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding odd-numbered signed elements of the second source vector, and then doubles the result of the multiplication. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively added to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0 1 0	0 0 1 0	0	size	0	Zm	0	0 0 0 1	0	Zn	Zda
								S		

Encoding

```
SQDMLALBT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel1 : integer = 0;
let sel2 : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2 * e + sel1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2 * e + sel2)*:(esize DIV 2)]);
    let element3 : integer = SInt(result[e*:esize]);
    let product : integer = SInt(SignedSat{esize}(2 * element1 * element2));
    result[e*:esize] = SignedSat{esize}(element3 + product);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.631 SQDMLALT (vectors)

Signed saturating doubling multiply-add (top)

This instruction multiplies the odd-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively added to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

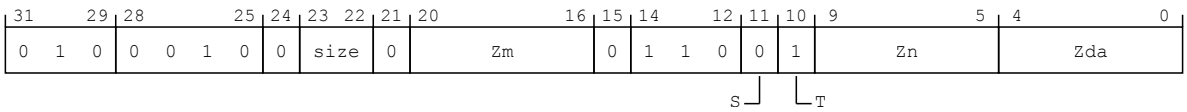

Encoding

```
SQDMLALT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel1 : integer = 1;
let sel2 : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2 * e + sel1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2 * e + sel2)*:(esize DIV 2)]);
    let element3 : integer = SInt(result[e*:esize]);
    let product : integer = SInt(SignedSat{esize}(2 * element1 * element2));
    result[e*:esize] = SignedSat{esize}(element3 + product);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.632 SQDMLALT (indexed)

Signed saturating doubling multiply-add by indexed element (top)

This instruction multiplies the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively added to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

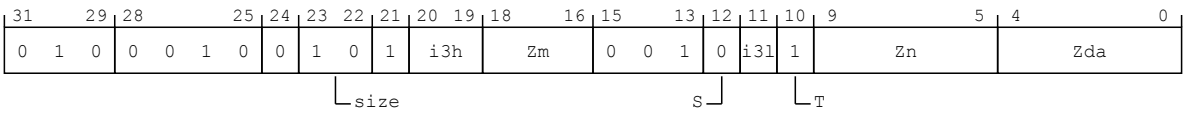

Encoding

```
SQDMLALT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

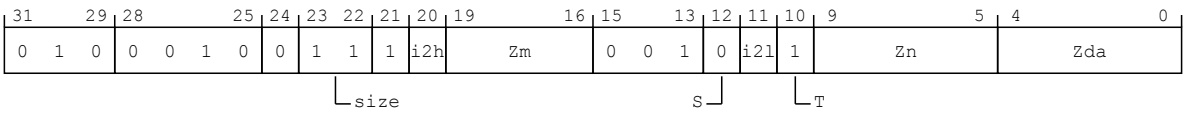

Encoding

```
SQDMLALT  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let element3 : integer = SInt(result[e*:(2*esize)]);
    let product : integer = SInt(SignedSat{2*esize}(2 * element1 * element2));
    let res : integer = element3 + product;
    result[e*:(2*esize)] = SignedSat{2*esize}(res);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.633 SQDMLSLB (vectors)

Signed saturating doubling multiply-subtract (bottom)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively subtracted from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

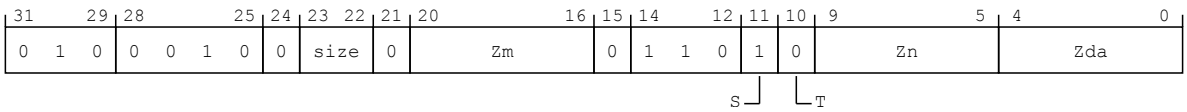

Encoding

```
SQDMLSLB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel1 : integer = 0;
let sel2 : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2 * e + sel1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2 * e + sel2)*:(esize DIV 2)]);
    let element3 : integer = SInt(result[e*:esize]);
    let product : integer = SInt(SignedSat{esize}(2 * element1 * element2));
    result[e*:esize] = SignedSat{esize}(element3 - product);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.634 SQDMLSLB (indexed)

Signed saturating doubling multiply-subtract by indexed element (bottom)

This instruction multiplies the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively subtracted from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

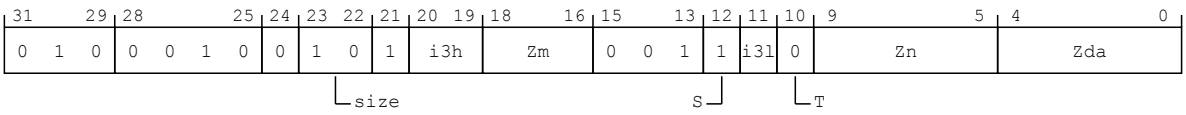

Encoding

```
SQDMLSLB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

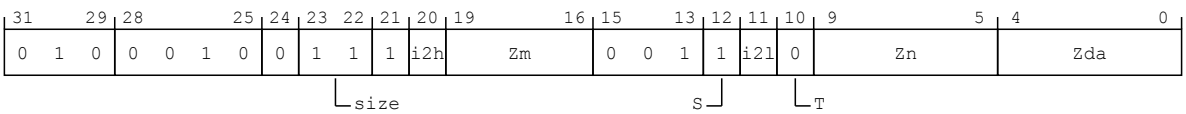

Encoding

```
SQDMLSLB  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let element3 : integer = SInt(result[e*:(2*esize)]);
    let product : integer = SInt(SignedSat{2*esize}(2 * element1 * element2));
    let res : integer = element3 - product;
    result[e*:(2*esize)] = SignedSat{2*esize}(res);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.635 SQDMLSLBT

Signed saturating doubling multiply-subtract (bottom × top)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding odd-numbered signed elements of the second source vector, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively subtracted from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

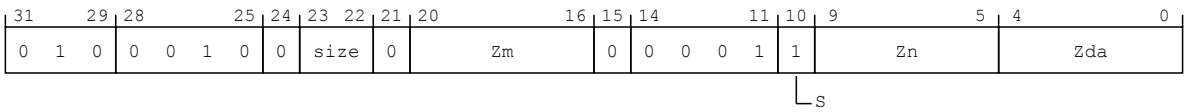

Encoding

```
SQDMLSLBT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel1 : integer = 0;
let sel2 : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2 * e + sel1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2 * e + sel2)*:(esize DIV 2)]);
    let element3 : integer = SInt(result[e*:esize]);
    let product : integer = SInt(SignedSat{esize}(2 * element1 * element2));
    result[e*:esize] = SignedSat{esize}(element3 - product);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.636 SQDMLSLT (vectors)

Signed saturating doubling multiply-subtract (top)

This instruction multiplies the odd-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively subtracted from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

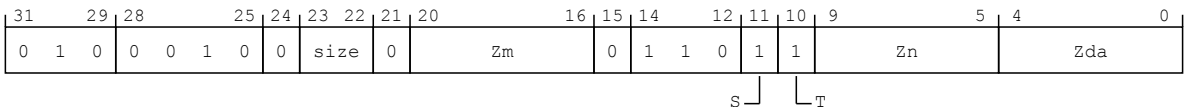

Encoding

```
SQDMLSLT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel1 : integer = 1;
let sel2 : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2 * e + sel1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2 * e + sel2)*:(esize DIV 2)]);
    let element3 : integer = SInt(result[e*:esize]);
    let product : integer = SInt(SignedSat{esize}(2 * element1 * element2));
    result[e*:esize] = SignedSat{esize}(element3 - product);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.637 SQDMLSLT (indexed)

Signed saturating doubling multiply-subtract by indexed element (top)

This instruction multiplies the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and then doubles the result. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$ and is then destructively subtracted from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

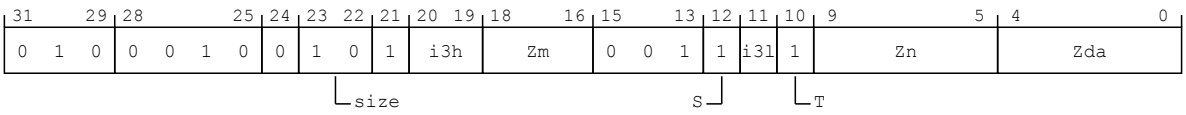

Encoding

```
SQDMLSLT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

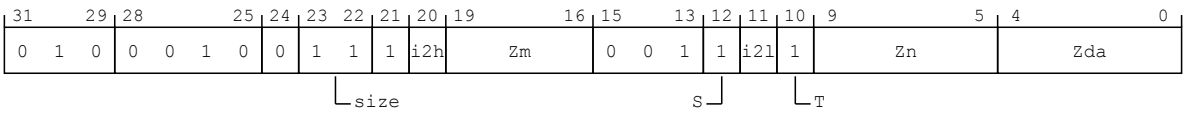

Encoding

```
SQDMLSLT  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let element3 : integer = SInt(result[e*:(2*esize)]);
    let product : integer = SInt(SignedSat{2*esize}(2 * element1 * element2));
    let res : integer = element3 - product;
    result[e*:(2*esize)] = SignedSat{2*esize}(res);
end;

Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.638 SQDMULH (vectors)

Signed saturating doubling multiply high (unpredicated)

This instruction multiplies the signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result and places the most significant half of the results in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 1 1	1 0	0	Zn	Zd
								R		

Encoding

```
SQDMULH  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    let res : integer = 2 * element1 * element2;
    result[e*:esize] = SignedSat{esize}(res >> esize);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.639 SQDMULH (indexed)

Signed saturating doubling multiply high by indexed element

This instruction multiplies all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, then doubles and places the most significant half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23	22	21	20 19	18 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0	0	i3h	1	i3l	Zm	1 1 1 1 0	0	Zn	Zd
									R		

Encoding

```
SQDMULH  <Zd>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	1 1 1 1 0	0	Zn	Zd
			size					R		

Encoding

```
SQDMULH  <Zd>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (size)	1	i1	Zm	1 1 1 1 0	0 (R)	Zn	Zd

Encoding

```
SQDMULH  <Zd>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[s*:esize]);
    let res : integer = 2 * element1 * element2;
    result[e*:esize] = SignedSat{esize}(res >> esize);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.640 SQDMULLB (vectors)

Signed saturating doubling multiply (bottom)

This instruction multiplies the even-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result and places the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

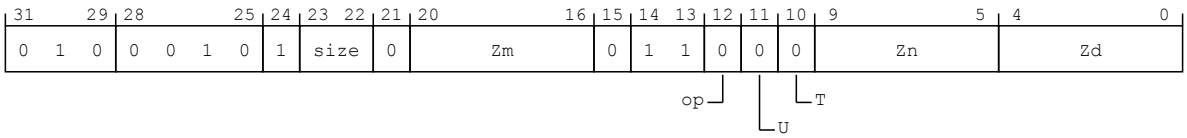

Encoding

```
SQDMULLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let res : integer = 2 * element1 * element2;
    result[e*:esize] = SignedSat{esize}(res);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.641 SQDMULLB (indexed)

Signed saturating doubling multiply by indexed element (bottom)

This instruction multiplies the even-numbered signed elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment, and then doubles and places the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

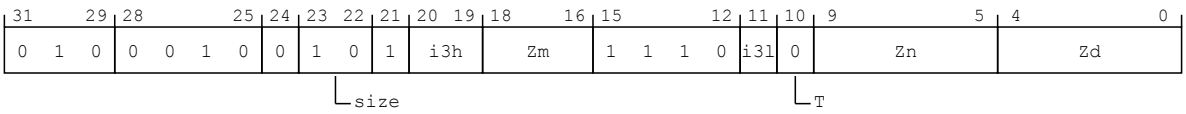

Encoding

```
SQDMULLB  <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

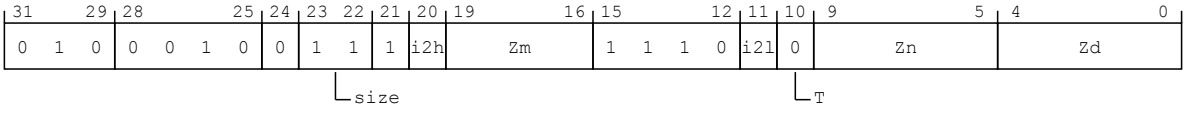

Encoding

```
SQDMULLB  <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 0;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let res : integer = 2 * element1 * element2;
    result[e*:(2*esize)] = SignedSat{2*esize}(res);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.642 SQDMULLT (vectors)

Signed saturating doubling multiply (top)

This instruction multiplies the odd-numbered signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles and places the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

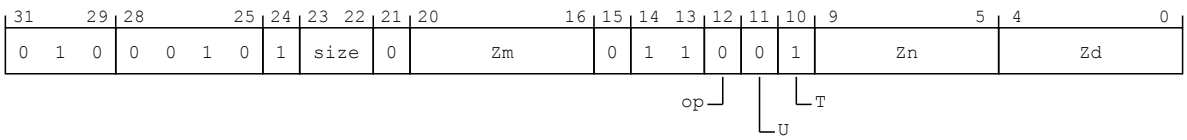

Encoding

```
SQDMULLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let res : integer = 2 * element1 * element2;
    result[e*:esize] = SignedSat{esize}(res);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.643 SQDMULLT (indexed)

Signed saturating doubling multiply by indexed element (top)

This instruction multiplies the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment, and then doubles and places the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

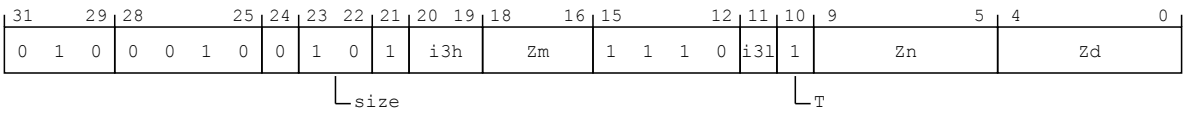

Encoding

```
SQDMULLT  <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

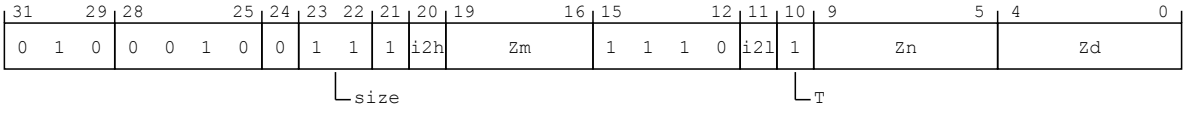

Encoding

```
SQDMULLT  <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = SInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = SInt(operand2[(2 * s + index)*:esize]);
    let res : integer = 2 * element1 * element2;
    result[e*:(2*esize)] = SignedSat{2*esize}(res);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.644 SQINCB

Signed saturating increment scalar by multiple of 8-bit predicate constraint element count

This instruction determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

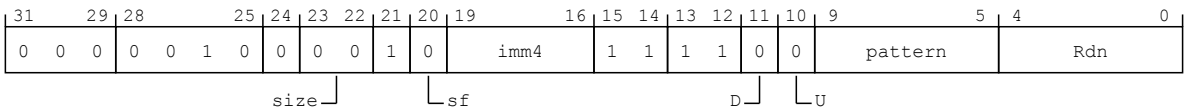

Encoding

```
SQINCB  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

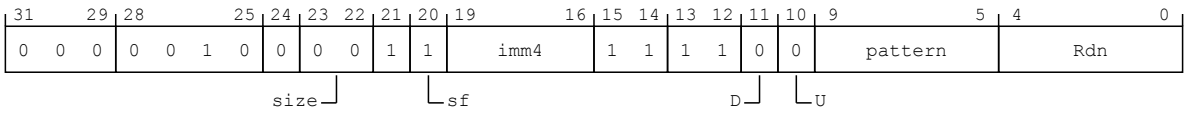

Encoding

```
SQINCB  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.645 SQINCD (scalar)

Signed saturating increment scalar by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1	1	0	imm4	1 1	1 1	0	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQINCD  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 1	1	1	imm4	1 1	1 1	0	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQINCD  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.646 SQINCD (vector)

Signed saturating increment vector by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit
signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

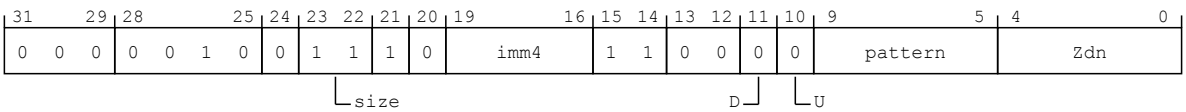

Encoding

```
SQINCD  <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) + (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) + (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.647 SQINCH (scalar)

Signed saturating increment scalar by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 1	1	0	imm4	1 1	1 1	0	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQINCH  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 1	1	1	imm4	1 1	1 1	0	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQINCH  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.648 SQINCH (vector)

Signed saturating increment vector by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit
signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

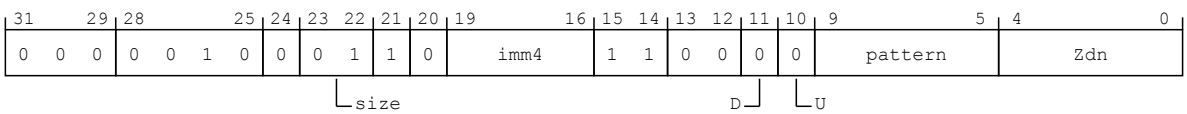

Encoding

```
SQINCH  <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) + (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) + (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.649 SQINCP (scalar)

Signed saturating increment scalar by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

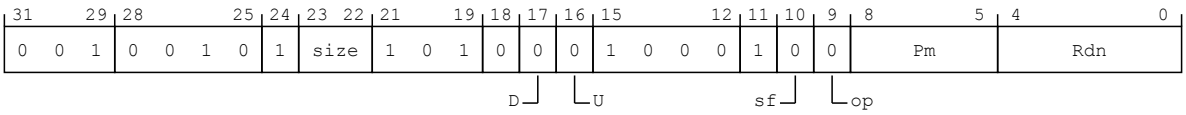

Encoding

```
SQINCP  <Xdn>, <Pm>.<T>, <Wdn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

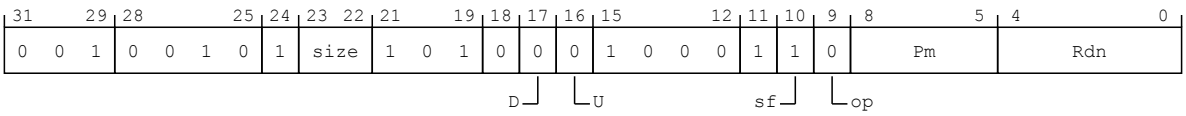

Encoding

```
SQINCP  <Xdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(ssize) = X{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(ssize);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + count);
    X{64}(dn) = ZeroExtend{64}(result);
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + count);
    X{64}(dn) = SignExtend{64}(result);
end;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.650 SQINCP (vector)

Signed saturating increment vector by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

SVE

(FEAT_SVE || FEAT_SME)

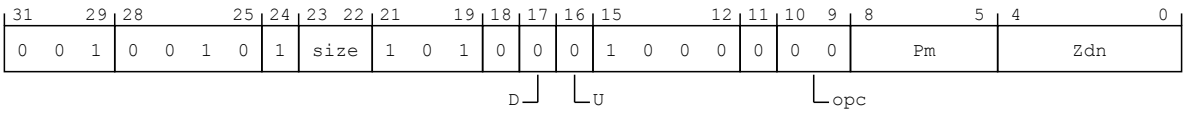

Encoding

```
SQINCP  <Zdn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Zdn);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(VL);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(opelt) + count);
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(opelt) + count);
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.651 SQINCW (scalar)

Signed saturating increment scalar by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 0	1	0	imm4	1 1	1 1	0	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQINCW  <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	1 0	1	1	imm4	1 1	1 1	0	0	pattern	Rdn
			size		sf				D	U		

Encoding

```
SQINCW  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.652 SQINCW (vector)

Signed saturating increment vector by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit
signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

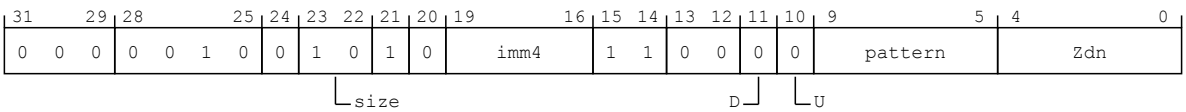

Encoding

```
SQINCW  <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) + (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) + (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.653 SQNEG

Signed saturating negate

This instruction negates the signed integer value in each Active element of the source vector, and places the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18	17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	0	1	0	0	1	1 0	1	Pg	Zn	Zd

Q, Z, op

Encoding

```
SQNEG  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18	17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	0	1	0	1	1	1 0	1	Pg	Zn	Zd

Q, Z, op

Encoding

```
SQNEG  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        var element : integer = SInt(operand[e*:esize]);
        element = -element;
        result[e*:esize] = SignedSat{esize}(element);

    end;
end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.654 SQRDCMLAH (vectors)

Saturating rounding doubling complex integer multiply-add high

This instruction multiplies without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

This instruction then doubles and adds the products to the corresponding components of the complex numbers in the addend vector, and destructively places the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14 13	12	11 10	9 5	4 0
0 1 0	0 0 1 0	0	size	0	Zm	0	0 1	1	rot	Zn	Zda
								op			

Encoding

```
SQRDCMLAH  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_r : boolean = (rot[0] != rot[1]);
let sub_i : boolean = (rot[1] == '1');
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

var res_r, res_i : integer;
for p = 0 to pairs-1 do
    let elt1_a : integer = SInt(operand1[(2 * p + sel_a)*:esize]);
    let elt2_a : integer = SInt(operand2[(2 * p + sel_a)*:esize]);
    let elt2_b : integer = SInt(operand2[(2 * p + sel_b)*:esize]);
    let elt3_r : bits(esize) = operand3[(2 * p + 0)*:esize];
    let elt3_i : bits(esize) = operand3[(2 * p + 1)*:esize];
    let product_r : integer =  elt1_a * elt2_a;
    let product_i : integer =  elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r;
    end;
    res_r = (res_r + (1 << (esize-1))) >> esize;
    result[(2 * p + 0)*:esize] = SignedSat{esize}(res_r);
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i;
    end;
    res_i = (res_i + (1 << (esize-1))) >> esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[(2 * p + 1)*:esize] = SignedSat{esize}(res_i);

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.655 SQRDCMLAH (indexed)

Saturating rounding doubling complex integer multiply-add high by indexed element

This instruction multiplies without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

This instruction then doubles and adds the products to the corresponding components of the complex numbers in the addend vector, and destructively places the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 16-bit and 32-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–12	11–10	9–5	4–0
0 1 0	0 0 1 0	0	1 0 (size)	1	i2	Zm	0 1 1 1	rot	Zn	Zda

Encoding

```
SQRDCMLAH  <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_r : boolean = (rot[0] != rot[1]);
let sub_i : boolean = (rot[1] == '1');
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–12	11–10	9–5	4–0
0 1 0	0 0 1 0	0	1 1 (size)	1	i1	Zm	0 1 1 1	rot	Zn	Zda

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SQRDCMLAH  <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel_a : integer = UInt(rot[0]);
let sel_b : integer = UInt(NOT(rot[0]));
let sub_r : boolean = (rot[0] != rot[1]);
let sub_i : boolean = (rot[1] == '1');
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “32-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

<const>

Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV (2 * esize);
let pairspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

var res_r, res_i : integer;
for p = 0 to pairs-1 do
    let segmentbase : integer = p - (p MOD pairspersegment);
    let s : integer = segmentbase + index;
    let elt1_a : integer = SInt(operand1[(2 * p + sel_a)*:esize]);
    let elt2_a : integer = SInt(operand2[(2 * s + sel_a)*:esize]);
    let elt2_b : integer = SInt(operand2[(2 * s + sel_b)*:esize]);
    let elt3_r : bits(esize) = operand3[(2 * p + 0)*:esize];
    let elt3_i : bits(esize) = operand3[(2 * p + 1)*:esize];
    let product_r : integer =  elt1_a * elt2_a;
    let product_i : integer =  elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r;
    end;
    res_r = (res_r + (1 << (esize-1))) >> esize;
    result[(2 * p + 0)*:esize] = SignedSat{esize}(res_r);
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i;
    end;
    res_i = (res_i + (1 << (esize-1))) >> esize;
    result[(2 * p + 1)*:esize] = SignedSat{esize}(res_i);

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.656 SQRDMLAH (vectors)

Signed saturating rounding doubling multiply-add high (unpredicated)

This instruction multiplies the signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result and destructively adds the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0 1 0	0 0 1 0	0	size	0	Zm	0	1 1 1 0	0	Zn	Zda
								S		

Encoding

```
SQRDMLAH  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    let element3 : integer = SInt(operand3[e*:esize]);
    let res : integer = (element3 << esize) + (2 * element1 * element2);
    result[e*:esize] = SignedSat{esize}((res + (1 << (esize - 1))) >> esize);

end;
Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.657 SQRDMLAH (indexed)

Signed saturating rounding doubling multiply-add high by indexed element

This instruction multiplies all signed elements within each 128-bit segment of the first source vector by the specified signed element of the corresponding second source vector segment, and then doubles the result and destructively adds the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

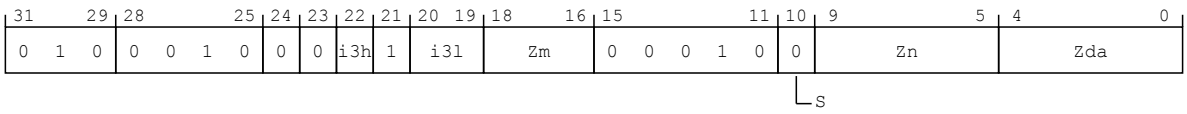

Encoding

```
SQRDMLAH  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

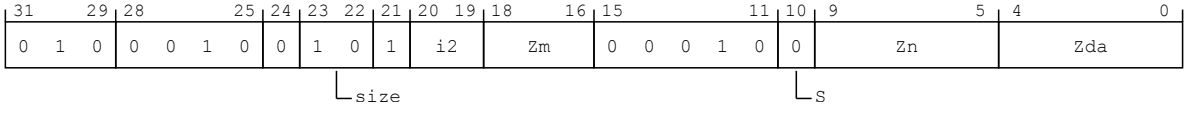

Encoding

```
SQRDMLAH  <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 1 0	0	Zn	Zda
			size					S		

Encoding

```
SQRDMLAH  <Zda>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[s*:esize]);
    let element3 : integer = SInt(operand3[e*:esize]);
    let res : integer = (element3 << esize) + (2 * element1 * element2);
    result[e*:esize] = SignedSat{esize}((res + (1 << (esize - 1))) >> esize);

end;
Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.658 SQRDMLSH (vectors)

Signed saturating rounding doubling multiply-subtract high (unpredicated)

This instruction multiplies the signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result and destructively subtracts the rounded high half of each result from the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15	14 11	10	9 5	4 0
0 1 0	0 0 1 0	0	size	0	Zm	0	1 1 1 0	1	Zn	Zda
								S		

Encoding

```
SQRDMLSH  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    let element3 : integer = SInt(operand3[e*:esize]);
    let res : integer = (element3 << esize) - (2 * element1 * element2);
    result[e*:esize] = SignedSat{esize}((res + (1 << (esize - 1))) >> esize);

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.659 SQRDMLSH (indexed)

Signed saturating rounding doubling multiply-subtract high by indexed element

This instruction multiplies all signed elements within each 128-bit segment of the first source vector by the specified signed element of the corresponding second source vector segment, and then doubles the result and destructively subtracts the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	0	i3h	1	i3l	Zm	0 0 0 1 0	1	Zn	Zda
									S		

Encoding

```
SQRDMLSH  <Zda>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 1 0	1	Zn	Zda
			size					S		

Encoding

```
SQRDMLSH  <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 1 0	1	Zn	Zda
			size					S		

Encoding

```
SQRDMLSH  <Zda>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[s*:esize]);
    let element3 : integer = SInt(operand3[e*:esize]);
    let res : integer = (element3 << esize) - (2 * element1 * element2);
    result[e*:esize] = SignedSat{esize}((res + (1 << (esize - 1))) >> esize);

end;
Z{VL}(da) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.660 SQRDMULH (vectors)

Signed saturating rounding doubling multiply high (unpredicated)

This instruction multiplies the signed elements of the first source vector by the corresponding elements of the second source vector, and then doubles the result and places the most significant rounded half of the result in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 1 1	1 0	1	Zn	Zd
								R		

Encoding

```
SQRDMULH  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    let res : integer = 2 * element1 * element2;
    result[e*:esize] = SignedSat{esize}((res + (1 << (esize - 1))) >> esize);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.661 SQRDMULH (indexed)

Signed saturating rounding doubling multiply high by indexed element

This instruction multiplies all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, then doubles and places the most significant rounded half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 3 classes: 16-bit, 32-bit, and 64-bit

16-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23	22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	0	i3h	1	i3l	Zm	1 1 1 1 0	1	Zn	Zd
									R		

Encoding

```
SQRDMULH  <Zd>.H, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	1 1 1 1 0	1	Zn	Zd
			size					R		

Encoding

```
SQRDMULH  <Zd>.S, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20	19 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	1 1 1 1 0	1	Zn	Zd
			size					R		

Encoding

```
SQRDMULH  <Zd>.D, <Zn>.D, <Zm>.D[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “16-bit” and “32-bit” variants: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “16-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “32-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2” field.

For the “64-bit” variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[s*:esize]);
    let res : integer = 2 * element1 * element2;
    result[e*:esize] = SignedSat{esize}((res + (1 << (esize - 1))) >> esize);

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.662 SQRSHL

Signed saturating rounding shift left (predicated)

This instruction shifts active signed elements of the first source vector by corresponding elements of the second source vector, and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

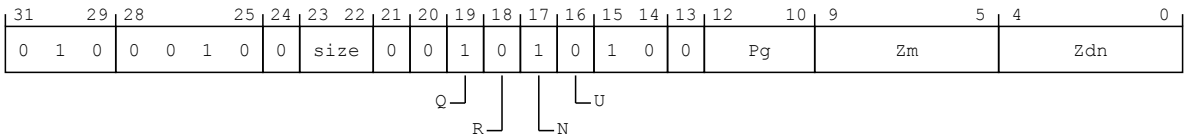

Encoding

```
SQRSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = SignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.663 SQRSHLR

Signed saturating rounding shift left reversed (predicated)

This instruction shifts active signed elements of the second source vector by corresponding elements of the first source vector and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

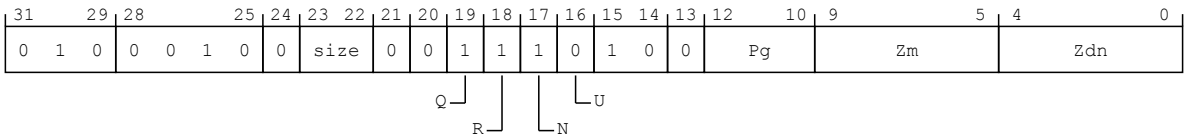

Encoding

```
SQRSHLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand2 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = SignedSat{esize}(res);
    else
        result[e*:esize] = operand2[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.664 SQRSHRN

Signed saturating rounding shift right narrow by immediate to interleaved integer

This instruction shifts right by an immediate value the signed integer value in each element of the two source vectors, and places the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

16-bit

(FEAT_SME2 || FEAT_SVE2p1)

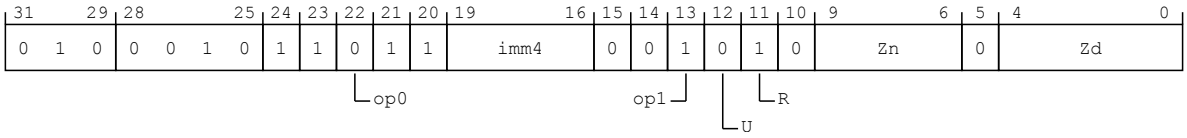

Encoding

```
SQRSHRN  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
let shift : integer = esize - UInt(imm4);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as "Zn" times 2 plus 1.

<const>

Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 1 do
        let operand : bits(VL) = Z{}(n+i);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element : bits(2 * esize) = operand[e*:(2 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(2*e + i)*:esize] = SignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.665 SQRSHRNB

Signed saturating rounding shift right narrow by immediate (bottom)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

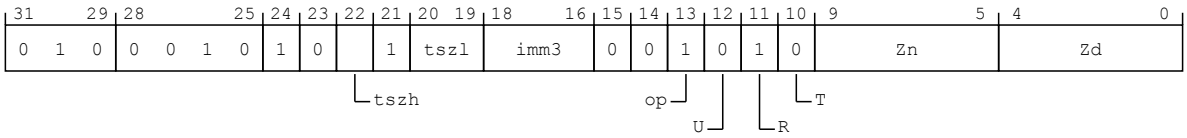

Encoding

```
SQRSHRNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 0)*:esize] = SignedSat{esize}(res);
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.666 SQRSHRNT

Signed saturating rounding shift right narrow by immediate (top)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

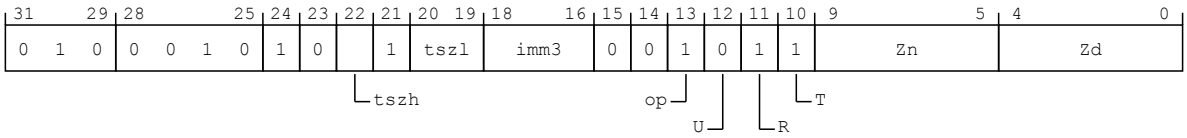

Encoding

```
SQRSHRNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 1)*:esize] = SignedSat{esize}(res);

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.667 SQRSHRUN

Signed saturating rounding shift right unsigned narrow by immediate and interleave

This instruction shifts right by an immediate value the signed integer value in each element of the two source vectors, and places the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

16-bit

(FEAT_SME2 || FEAT_SVE2p1)

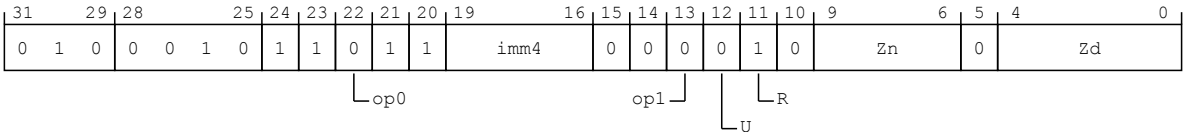

Encoding

```
SQRSHRUN  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
let shift : integer = esize - UInt(imm4);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as "Zn" times 2 plus 1.

<const>

Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 1 do
        let operand : bits(VL) = Z{}(n+i);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element : bits(2 * esize) = operand[e*:(2 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(2*e + i)*:esize] = UnsignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.668 SQRSHRUNB

Signed saturating rounding shift right narrow by immediate to unsigned integer (bottom)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

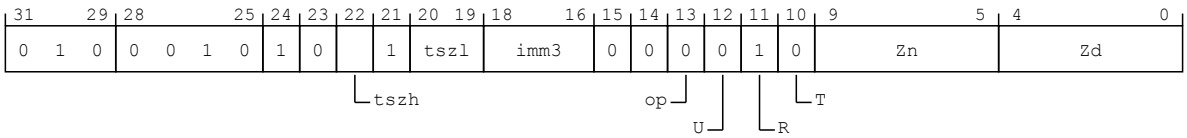

Encoding

```
SQRSHRUNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 0)*:esize] = UnsignedSat{esize}(res);
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.669 SQRSHRUNT

Signed saturating rounding shift right narrow by immediate to unsigned integer (top)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

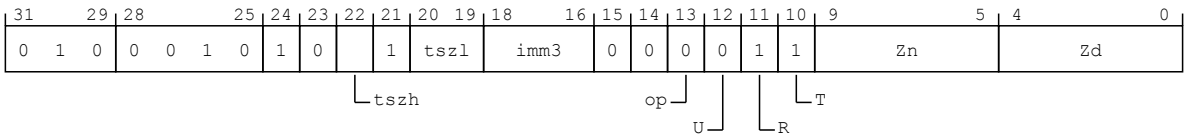

Encoding

```
SQRSHRUNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 1)*:esize] = UnsignedSat{esize}(res);

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.670 SQSHL (immediate)

Signed saturating shift left by immediate

This instruction shifts left by immediate each active signed element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

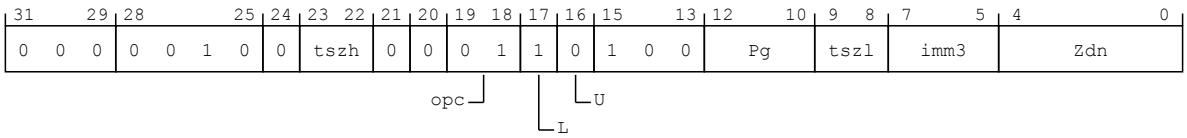

Encoding

```
SQSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = element1 << shift;
        result[e*:esize] = SignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.671 SQSHL (vectors)

Signed saturating shift left (predicated)

This instruction shifts active signed elements of the first source vector by corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

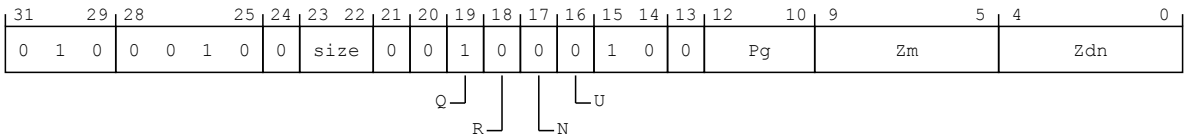

Encoding

```
SQSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = element >> shift;
        end;
        result[e*:esize] = SignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.672 SQSHLR

Signed saturating shift left reversed (predicated)

This instruction shifts active signed elements of the second source vector by the corresponding elements of the first source vector, and destructively places the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

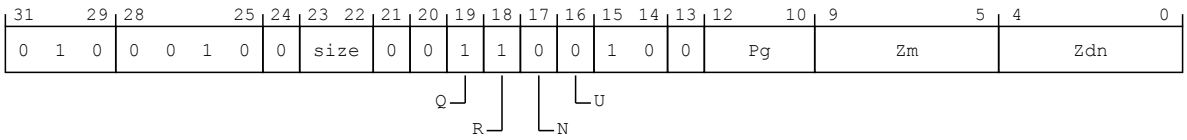

Encoding

```
SQSHLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand2 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = element >> shift;
        end;
        result[e*:esize] = SignedSat{esize}(res);
    else
        result[e*:esize] = operand2[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.673 SQSHLU

Signed saturating shift left unsigned by immediate

This instruction shifts left by immediate each active signed element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

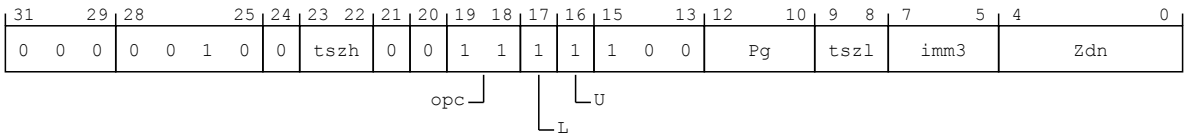

Encoding

```
SQSHLU  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = element1 << shift;
        result[e*:esize] = UnsignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.674 SQSHRNB

Signed saturating shift right narrow by immediate (bottom)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

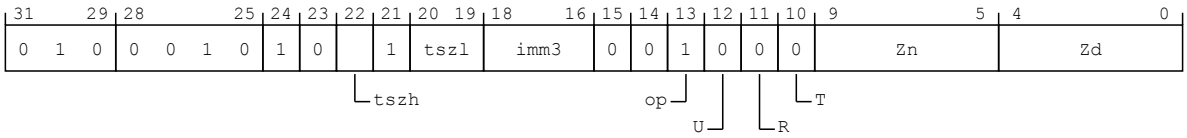

Encoding

```
SQSHRNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = SInt(element) >> shift;
    result[(2*e + 0)*:esize] = SignedSat{esize}(res);
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.675 SQSHRNT

Signed saturating shift right narrow by immediate (top)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

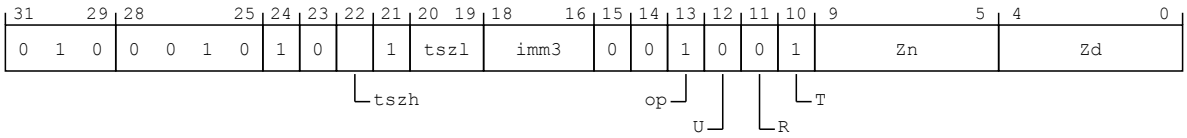

Encoding

```
SQSHRNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = SInt(element) >> shift;
    result[(2*e + 1)*:esize] = SignedSat{esize}(res);

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.676 SQSHRUNB

Signed saturating shift right narrow by immediate to unsigned integer (bottom)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

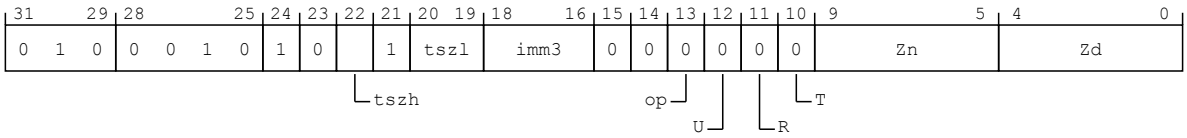

Encoding

```
SQSHRUNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = SInt(element) >> shift;
    result[(2*e + 0)*:esize] = UnsignedSat{esize}(res);
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.677 SQSHRUNT

Signed saturating shift right narrow by immediate to unsigned integer (top)

This instruction shifts each signed integer value in the source vector elements right by an immediate value, and places the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

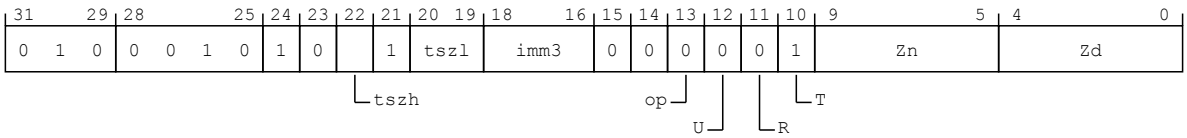

Encoding

```
SQSHRUNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = SInt(element) >> shift;
    result[(2*e + 1)*:esize] = UnsignedSat{esize}(res);

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.678 SQSUB (vectors, predicated)

Signed saturating subtract (predicated)

This instruction subtracts active signed elements of the second source vector from corresponding signed elements of the first source vector and destructively places the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

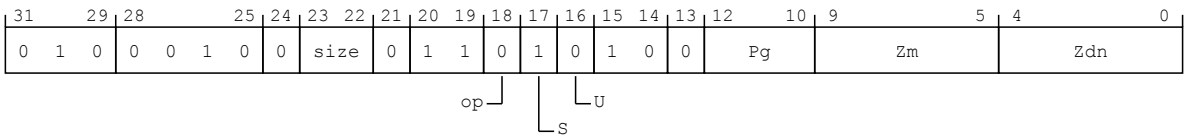

Encoding

```
SQSUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = SignedSat{esize}(element1 - element2);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.679 SQSUB (immediate)

Signed saturating subtract immediate (unpredicated)

This instruction performs a signed saturating subtraction of an unsigned immediate from each element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 14	13	12 5	4 0
0 0 1	0 0 1 0	1	size	1	0 0	1 1	0	1 1	sh	imm8	Zdn
							U				

Encoding

```
SQSUB  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    (result[e*:esize], -) = SatQ{esize}(element1 - imm, unsigned);

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.680 SQSUB (vectors, unpredicated)

Signed saturating subtract (unpredicated)

This instruction performs a signed saturating subtraction of all elements of the second source vector from the corresponding elements of the first source vector, and places the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 0 0	1 1	0	Zn	Zd
								U		

Encoding

```
SQSUB  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    if unsigned then
        let element1 : integer = UInt(op1elt);
        let element2 : integer = UInt(op2elt);
        (result[e*:esize], -) = UnsignedSatQ{esize}(element1 - element2);
    else
        let element1 : integer = SInt(op1elt);
        let element2 : integer = SInt(op2elt);
        (result[e*:esize], -) = SignedSatQ{esize}(element1 - element2);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.681 SQSUBR

Signed saturating subtract reversed (predicated)

This instruction subtracts active signed elements of the first source vector from corresponding signed elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

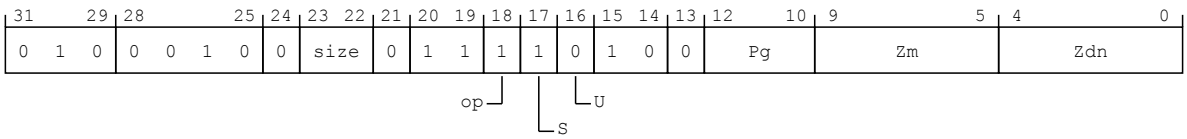

Encoding

```
SQSUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = SignedSat{esize}(element2 - element1);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.682 SQXTNB

Signed saturating extract narrow (bottom)

This instruction saturates the signed integer value in each source element to half the original source element width, and places the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

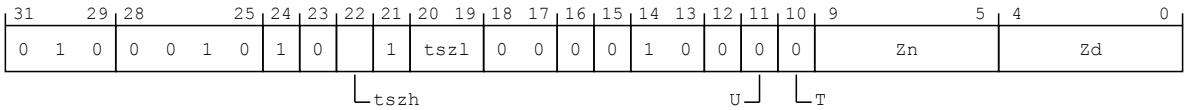

Encoding

```
SQXTNB  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if !(tsize IN {'001', '010', '100'}) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
0	11	RESERVED
1	00	S
1	01	RESERVED
1	1x	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
0	11	RESERVED
1	00	D
1	01	RESERVED
1	1x	RESERVED

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let res : bits(halfesize) = SignedSat{halfesize}(element1);
    result[(2*e + 0)*:halfesize] = res;
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.683 SQXTNT

Signed saturating extract narrow (top)

This instruction saturates the signed integer value in each source element to half the original source element width, and places the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

SVE2

(FEAT_SVE2 || FEAT_SME)

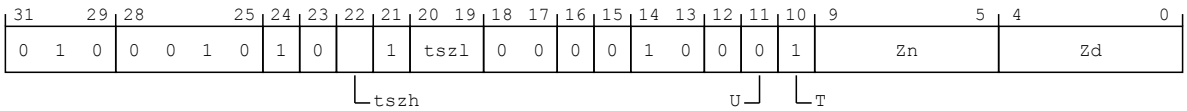

Encoding

```
SQXTNT  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if !(tsize IN {'001', '010', '100'}) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
0	11	RESERVED
1	00	S
1	01	RESERVED
1	1x	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
0	11	RESERVED
1	00	D
1	01	RESERVED
1	1x	RESERVED

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let res : bits(halfesize) = SignedSat{halfesize}(element1);
    result[(2*e + 1)*:halfesize] = res;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.684 SQXTUNB

Signed saturating extract narrow to unsigned integer (bottom)

This instruction saturates the signed integer value in each source element to an unsigned integer value that is half the original source element width, and places the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

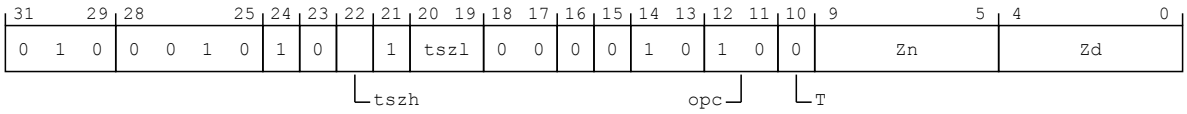

Encoding

```
SQXTUNB  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if !(tsize IN {'001', '010', '100'}) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
0	11	RESERVED
1	00	S
1	01	RESERVED
1	1x	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
0	11	RESERVED
1	00	D
1	01	RESERVED
1	1x	RESERVED

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let res : bits(halfesize) = UnsignedSat{halfesize}(element1);
    result[(2*e + 0)*:halfesize] = res;
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.685 SQXTUNT

Signed saturating extract narrow to unsigned integer (top)

This instruction saturates the signed integer value in each source element to an unsigned integer value that is half the original source element width, and places the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

SVE2

(FEAT_SVE2 || FEAT_SME)

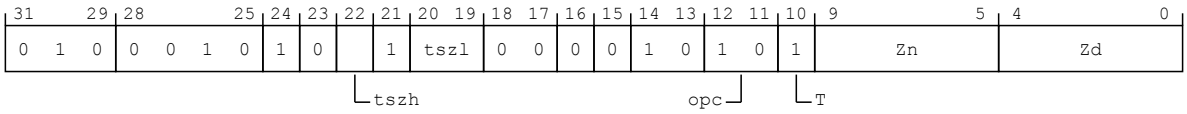

Encoding

```
SQXTUNT  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if !(tsize IN {'001', '010', '100'}) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
0	11	RESERVED
1	00	S
1	01	RESERVED
1	1x	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
0	11	RESERVED
1	00	D
1	01	RESERVED
1	1x	RESERVED

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let res : bits(halfesize) = UnsignedSat{halfesize}(element1);
    result[(2*e + 1)*:halfesize] = res;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.686 SRHADD

Signed rounding halving add

This instruction adds active signed elements of the first source vector to corresponding signed elements of the second source vector, shifts right one bit, and destructively places the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

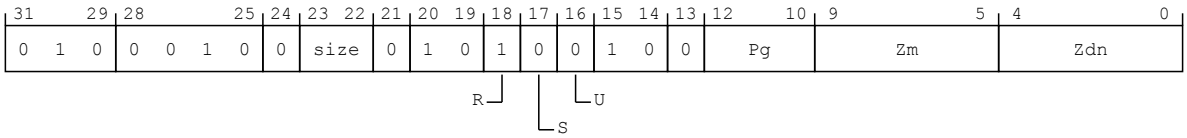

Encoding

```
SRHADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 + element2 + 1) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.687 SRI

Shift right and insert (immediate)

This instruction shifts each source vector element right by an immediate value, and inserts the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–14	13–11	10	9–5	4–0
0 1 0	0 0 1 0	1	tszh	0	tszl	imm3	1 1	1 1 0	0	Zn	Zd
									op		

Encoding

```
SRI  <Zd>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element1 : bits(esize) = result[e*:esize];
    let element2 : bits(esize) = operand[e*:esize];
    let mask : bits(esize) = LSR(Ones{esize}, shift);
    let shiftedval : bits(esize) = LSR(element2, shift);
    result[e*:esize] = (element1 AND (NOT mask)) OR shiftedval;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.688 SRSHL

Signed rounding shift left (predicated)

This instruction shifts active signed elements of the first source vector by corresponding elements of the second source vector, and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

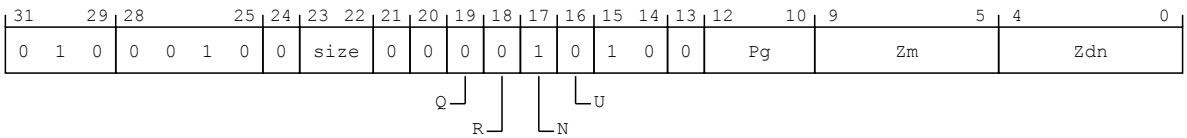

Encoding

```
SRSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.689 SRSHLR

Signed rounding shift left reversed (predicated)

This instruction shifts active signed elements of the second source vector by corresponding elements of the first source vector and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

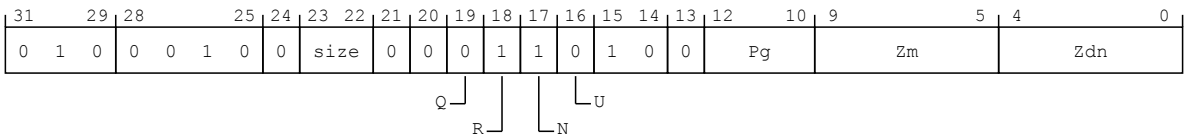

Encoding

```
SRSHLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand2 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand2[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.690 SRSHR

Signed rounding shift right by immediate

This instruction shifts right by immediate each active signed element of the source vector, and destructively places the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

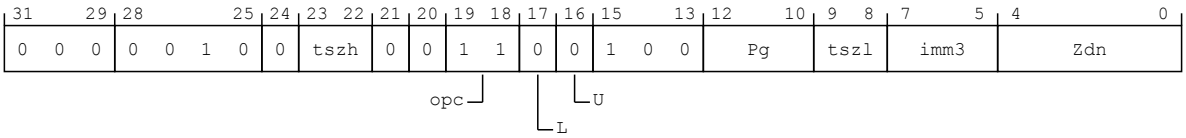

Encoding

```
SRSHR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 + (1 << (shift - 1))) >> shift;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.691 SRSRA

Signed rounding shift right and accumulate (immediate)

This instruction shifts right by immediate each signed element of the source vector, preserving the sign bit, and destructively adds the rounded intermediate result to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–14	13–12	11	10	9–5	4–0
0 1 0	0 0 1 0	1	tszh	0	tszl	imm3	1 1	1 0	1 (R)	0 (U)	Zn	Zda

Encoding

```
SRSRA  <Zda>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let da : integer = UInt(Zda);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : integer = (SInt(operand1[e*:esize]) + (1 << (shift - 1))) >> shift;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.692 SSHLLB

Signed shift left long by immediate (bottom)

This instruction shifts left by immediate each even-numbered signed element of the source vector, and places the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

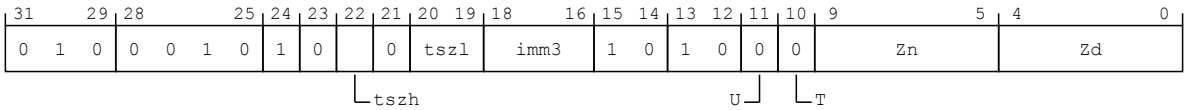

Encoding

```
SSHLLB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[(2*e + 0)*:esize];
    let shifted_value : integer = SInt(element) << shift;
    result[e*:(2*esize)] = shifted_value[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.693 SSHLLT

Signed shift left long by immediate (top)

This instruction shifts left by immediate each odd-numbered signed element of the source vector, and places the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

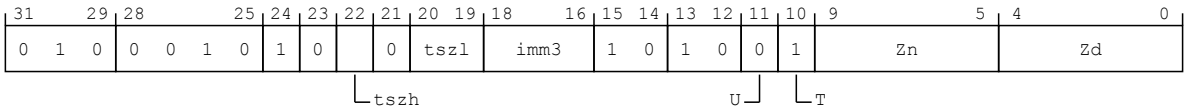

Encoding

```
SSHLLT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[(2*e + 1)*:esize];
    let shifted_value : integer = SInt(element) << shift;
    result[e*:(2*esize)] = shifted_value[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.694 SSRA

Signed shift right and accumulate (immediate)

This instruction shifts right by immediate each signed element of the source vector, preserving the sign bit, and destructively adds the truncated intermediate result to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 14	13 12	11	10	9 5	4 0
0 1 0	0 0 1 0	1	tszh	0	tszl	imm3	1 1	1 0	0 (R)	0 (U)	Zn	Zda

Encoding

```
SSRA  <Zda>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let da : integer = UInt(Zda);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : integer = SInt(operand1[e*:esize]) >> shift;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.695 SSUBLB

Signed subtract long (bottom)

This instruction subtracts the even-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

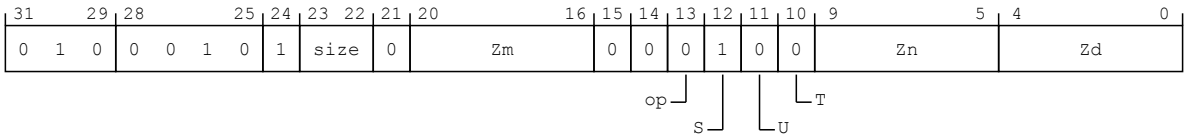

Encoding

```
SSUBLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 0;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 - element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.696 SSUBLBT

Signed subtract long (bottom - top)

This instruction subtracts the odd-numbered signed elements of the second source vector from the even-numbered signed elements of the first source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

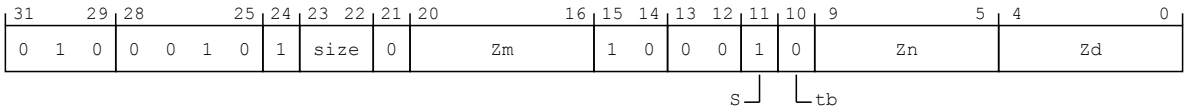

Encoding

```
SSUBLBT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 1;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 - element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.697 SSUBLT

Signed subtract long (top)

This instruction subtracts the odd-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

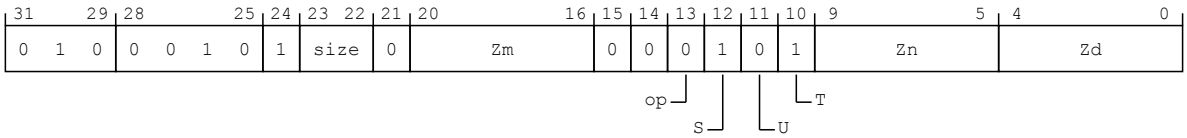

Encoding

```
SSUBLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 1;
let sel2 : integer = 1;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 - element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.698 SSUBLTB

Signed subtract long (top - bottom)

This instruction subtracts the even-numbered signed elements of the second source vector from the odd-numbered signed elements of the first source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

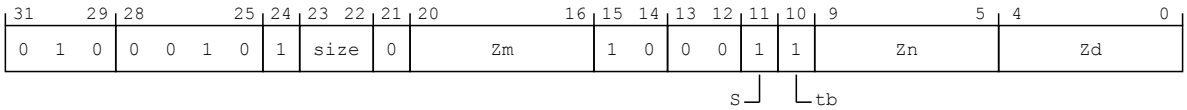

Encoding

```
SSUBLTB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 1;
let sel2 : integer = 0;
let unsigned : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 - element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.699 SSUBWB

Signed subtract wide (bottom)

This instruction subtracts the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

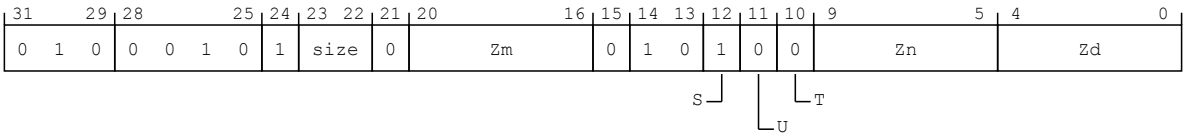

Encoding

```
SSUBWB  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    result[e*:esize] = (element1 - element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.700 SSUBWT

Signed subtract wide (top)

This instruction subtracts the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

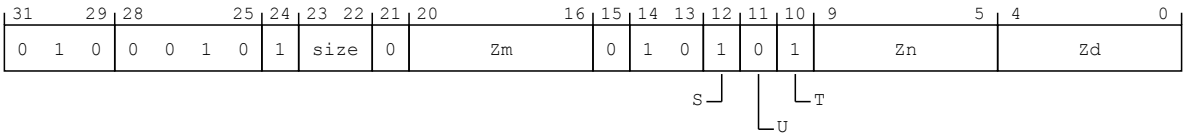

Encoding

```
SSUBWT  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = SInt(operand1[e*:esize]);
    let element2 : integer = SInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    result[e*:esize] = (element1 - element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.701 ST1B (scalar plus immediate, consecutive registers)

Contiguous store of bytes from multiple consecutive vectors (immediate index)

This instruction performs a contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		0	0	0	PNg		Rn		Zt		0
												msz								N

Encoding

```
ST1B  { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		1	0	0	PNg		Rn		Zt		0	0
												msz									N

Encoding

```
ST1B  { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.702 ST1B (scalar plus scalar, consecutive registers)

Contiguous store of bytes from multiple consecutive vectors (scalar index)

This instruction performs a contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

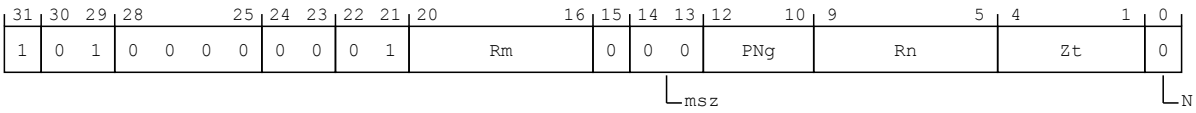

Encoding

```
ST1B  { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

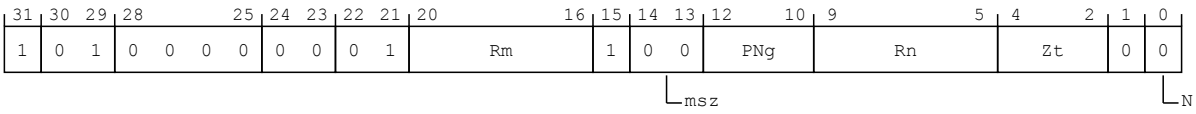

Encoding

```
ST1B  { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.703 ST1B (vector plus immediate)

Scatter store bytes from a vector (immediate index)

This instruction performs a scatter store of bytes from the Active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0	1 1	imm5	1 0 1	Pg	Zn	Zt
		msz						

Encoding

```
ST1B  { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0	1 0	imm5	1 0 1	Pg	Zn	Zt
		msz						

Encoding

```
ST1B  { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.704 ST1B (scalar plus immediate, single register)

Contiguous store bytes from vector (immediate index)

This instruction performs a contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0 (msz)	size	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
ST1B  { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8 << UInt(size);
let msize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.705 ST1B (scalar plus scalar, single register)

Contiguous store bytes from vector (scalar index)

This instruction performs a contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0	size	!=11111	0	1	0	Pg	Rn	Zt
				Rm						

Encoding

```
ST1B  { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8 << UInt(size);
let msize : integer{} = 8;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.706 ST1B (scalar plus vector)

Scatter store bytes from a vector (vector index)

This instruction performs a scatter store of bytes from the Active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign-extended or zero-extended from 32 to 64 bits. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset, and 64-bit unscaled offset

32-bit unpacked unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0 (msz)	0 0	Zm	1	xs	0	Pg	Rn	Zt

Encoding

```
ST1B  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0 (msz)	1 0	Zm	1	xs	0	Pg	Rn	Zt

Encoding

```
ST1B  { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0	0 0	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1B  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.707 ST1D (scalar plus immediate, consecutive registers)

Contiguous store of doublewords from multiple consecutive vectors (immediate index)

This instruction performs a contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

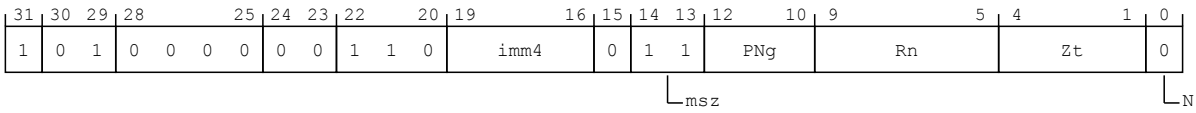

Encoding

```
ST1D  { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

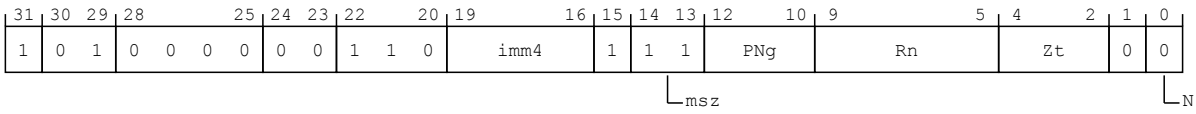

Encoding

```
ST1D  { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.708 ST1D (scalar plus scalar, consecutive registers)

Contiguous store of doublewords from multiple consecutive vectors (scalar index)

This instruction performs a contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm		0	1	1	PNg		Rn		Zt		0
														msz								N

Encoding

```
ST1D  { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm		1	1	1	PNg		Rn		Zt		0	0
														msz									N

Encoding

```
ST1D  { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.709 ST1D (vector plus immediate)

Scatter store doublewords from a vector (immediate index)

This instruction performs a scatter store of doublewords from the Active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1 (msz)	1 0	imm5	1 0 1	Pg	Zn	Zt

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offset : integer = UInt(imm5);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.710 ST1D (scalar plus immediate, single register)

Contiguous store doublewords from vector (immediate index)

This instruction performs a contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 64-bit element and 128-bit element

64-bit element

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1 (msz)	1 1 (opc)	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offset : integer = SInt(imm4);
```

128-bit element

(FEAT_SVE2p1)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1 (msz)	1 0 (opc)	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
ST1D  { <Zt>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 64;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.711 ST1D (scalar plus scalar, single register)

Contiguous store doublewords from vector (scalar index)

This instruction performs a contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 64-bit element and 128-bit element

64-bit element

(FEAT_SVE || FEAT_SME)

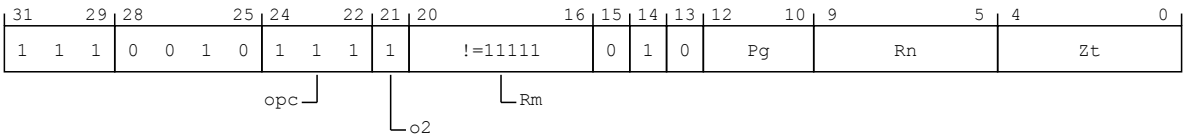

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
```

128-bit element

(FEAT_SVE2p1)

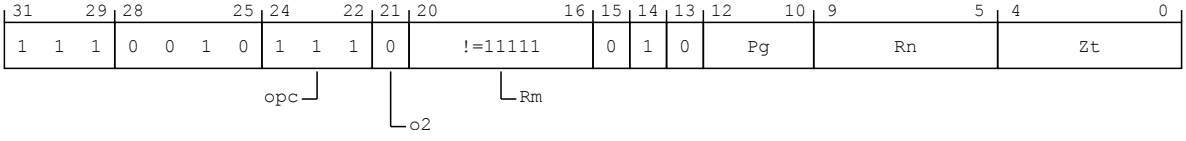

Encoding

```
ST1D  { <Zt>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 64;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.712 ST1D (scalar plus vector)

Scatter store doublewords from a vector (vector index)

This instruction performs a scatter store of doublewords from the Active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit unpacked scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1 (msz)	0 1	Zm	1	xs	0	Pg	Rn	Zt

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 3;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1 (msz)	0 0	Zm	1	xs	0	Pg	Rn	Zt

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1	0 1	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 3;
```

64-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1	0 0	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1D  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.713 ST1H (scalar plus immediate, consecutive registers)

Contiguous store of halfwords from multiple consecutive vectors (immediate index)

This instruction performs a contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		0	0	1	PNg		Rn		Zt		0
												msz								N

Encoding

```
ST1H  { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		1	0	1	PNg		Rn		Zt		0	0
												msz									N

Encoding

```
ST1H  { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.714 ST1H (scalar plus scalar, consecutive registers)

Contiguous store of halfwords from multiple consecutive vectors (scalar index)

This instruction performs a contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

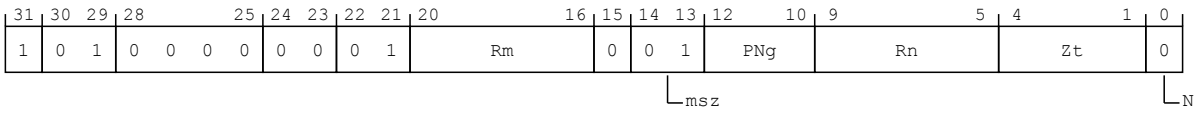

Encoding

```
ST1H  { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

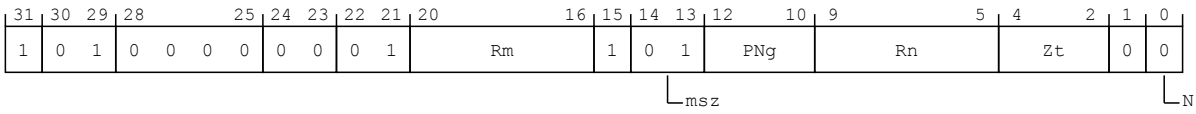

Encoding

```
ST1H  { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.715 ST1H (vector plus immediate)

Scatter store halfwords from a vector (immediate index)

This instruction performs a scatter store of halfwords from the Active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	1 1	imm5	1 0 1	Pg	Zn	Zt
		msz						

Encoding

```
ST1H  { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	1 0	imm5	1 0 1	Pg	Zn	Zt
		msz						

Encoding

```
ST1H  { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.716 ST1H (scalar plus immediate, single register)

Contiguous store halfwords from vector (immediate index)

This instruction performs a contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1 (msz)	size	0	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
ST1H  { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8 << UInt(size);
let msize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.717 ST1H (scalar plus scalar, single register)

Contiguous store halfwords from vector (scalar index)

This instruction performs a contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	size	!=11111 (Rm)	0	1	0	Pg	Rn	Zt

Encoding

```
ST1H  { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8 << UInt(size);
let msize : integer{} = 16;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.718 ST1H (scalar plus vector)

Scatter store halfwords from a vector (vector index)

This instruction performs a scatter store of halfwords from the Active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	1 1	Zm	1	xs	0	Pg	Rn	Zt
		msz								

Encoding

```
ST1H  { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	0 1	Zm	1	xs	0	Pg	Rn	Zt
		msz								

Encoding

```
ST1H  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 1;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	0 0	Zm	1	xs	0	Pg	Rn	Zt
		msz								

Encoding

```
ST1H  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	1 0	Zm	1	xs	0	Pg	Rn	Zt
		msz								

Encoding

```
ST1H  { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	0 1	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1H  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 1;
```

64-bit unscaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	0 0	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1H  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.719 ST1Q

Scatter store quadwords

This instruction performs a scatter store of quadwords from the Active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2p1)

31 29	28 25	24 22	21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0 0	1	Rm	0 0 1	Pg	Zn	Zt

Encoding

```
ST1Q  { <Zt>.Q }, <Pg>, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
CheckNonStreamingSVEEnabled();
let elements : integer = VL DIV 128;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var src : bits(VL);
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, 128) then
    base = Z{VL}(n);
    offset = X{64}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, 128) then
        let baddr : bits(64) = base[(2*e)*:64];
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        Mem{128}(addr, accdesc) = src[e*:128];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.720 ST1W (scalar plus immediate, consecutive registers)

Contiguous store of words from multiple consecutive vectors (immediate index)

This instruction performs a contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		0	1	0	PNg		Rn		Zt		0
												msz								N

Encoding

```
ST1W  { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		1	1	0	PNg		Rn		Zt		0	0
												msz									N

Encoding

```
ST1W  { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.721 ST1W (scalar plus scalar, consecutive registers)

Contiguous store of words from multiple consecutive vectors (scalar index)

This instruction performs a contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

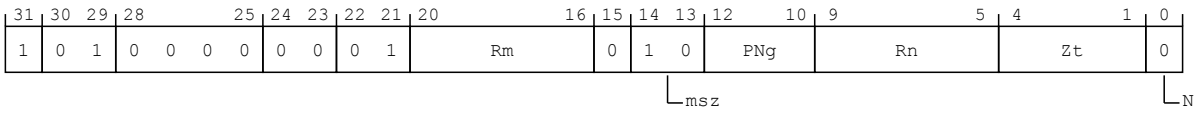

Encoding

```
ST1W  { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

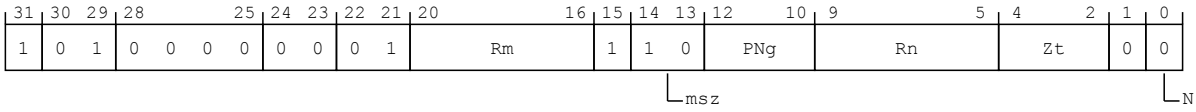

Encoding

```
ST1W  { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.722 ST1W (vector plus immediate)

Scatter store words from a vector (immediate index)

This instruction performs a scatter store of words from the Active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	1 1	imm5	1 0 1	Pg	Zn	Zt

msz

Encoding

```
ST1W  { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offset : integer = UInt(imm5);
```

64-bit element

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	1 0	imm5	1 0 1	Pg	Zn	Zt

msz

Encoding

```
ST1W  { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offset : integer = UInt(imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset * mbytes, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.723 ST1W (scalar plus immediate, single register)

Contiguous store words from vector (immediate index)

This instruction performs a contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit and 64-bit elements and 128-bit element

32-bit and 64-bit elements

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22	21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	1	sz	0	imm4	1 1 1	Pg	Rn	Zt
		msz								

Encoding

```
ST1W  { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32 << UInt(sz);
let msize : integer{} = 32;
let offset : integer = SInt(imm4);
```

128-bit element

(FEAT_SVE2p1)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	0 0	0	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST1W  { <Zt>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 32;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the “imm4” field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.724 ST1W (scalar plus scalar, single register)

Contiguous store words from vector (scalar index)

This instruction performs a contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit and 64-bit elements and 128-bit element

32-bit and 64-bit elements

(FEAT_SVE || FEAT_SME)

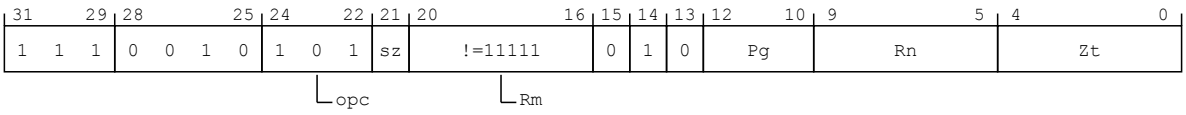

Encoding

```
ST1W  { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32 << UInt(sz);
let msize : integer{} = 32;
```

128-bit element

(FEAT_SVE2p1)

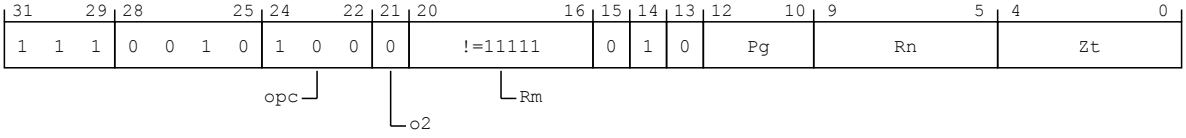

Encoding

```
ST1W  { <Zt>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let msize : integer{} = 32;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.725 ST1W (scalar plus vector)

Scatter store words from a vector (vector index)

This instruction performs a scatter store of words from the Active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign-extended or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset

32-bit scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 0	1 1	Zm	1	xs	0	Pg	Rn	Zt
		msz								

Encoding

```
ST1W  { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked scaled offset

(FEAT_SVE)

31 29	28 25	24 23	22 21	20 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 0	0 1	Zm	1	xs	0	Pg	Rn	Zt
		msz								

Encoding

```
ST1W  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 2;
```

32-bit unpacked unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0 (msz)	0 0	Zm	1	xs	0	Pg	Rn	Zt

Encoding

```
ST1W  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

32-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0 (msz)	1 0	Zm	1	xs	0	Pg	Rn	Zt

Encoding

```
ST1W  { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
let offs_size : integer{} = 32;
let offs_unsigned : boolean = xs == '0';
let scale : integer = 0;
```

64-bit scaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	0 1	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1W  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 2;
```

64-bit unscaled offset

(FEAT_SVE)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	0 0	Zm	1 0 1	Pg	Rn	Zt
		msz						

Encoding

```
ST1W  { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Zm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
let offs_size : integer{} = 64;
let offs_unsigned : boolean = TRUE;
let scale : integer = 0;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Zm>

Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod>

Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(VL);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
    base = if n == 31 then SP{64}() else X{64}(n);
    offset = Z{VL}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let offselt : bits(offs_size) = offset[e*:esize][offs_size-1:0];
        let off : integer = if offs_unsigned then UInt(offselt) else SInt(offselt);
        let addr : bits(64) = AddressAdd(base, off << scale, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.726 ST2B (scalar plus immediate)

Contiguous store two-byte structures from two vectors (immediate index)

This instruction performs a contiguous store of two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0	0 1	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST2B  { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.727 ST2B (scalar plus scalar)

Contiguous store two-byte structures from two vectors (scalar index)

This instruction performs a contiguous store of two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0	0 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST2B  { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.728 ST2D (scalar plus immediate)

Contiguous store two-doubleword structures from two vectors (immediate index)

This instruction performs a contiguous store of two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 1	0 1	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST2D  { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.729 ST2D (scalar plus scalar)

Contiguous store two-doubleword structures from two vectors (scalar index)

This instruction performs a contiguous store of two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 1	0 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST2D  { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.730 ST2H (scalar plus immediate)

Contiguous store two-halfword structures from two vectors (immediate index)

This instruction performs a contiguous store of two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	0 1	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST2H  { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.731 ST2H (scalar plus scalar)

Contiguous store two-halfword structures from two vectors (scalar index)

This instruction performs a contiguous store of two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	0 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST2H  { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.732 ST2Q (scalar plus immediate)

Contiguous store two-quadword structures from two vectors (immediate index)

This instruction performs a contiguous store of two-quadword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory that make up each structure. Inactive structures are not written to memory.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	0	0 1	0 0	imm4	0	0	0	Pg	Rn	Zt
			num								

Encoding

```
ST2Q  { <Zt1>.Q, <Zt2>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.733 ST2Q (scalar plus scalar)

Contiguous store two-quadword structures from two vectors (scalar index)

This instruction performs a contiguous store of two-quadword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory that make up each structure. Inactive structures are not written to memory.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0	0 1	1	!=11111	0	0	0	Pg	Rn	Zt
			num		Rm						

Encoding

```
ST2Q  { <Zt1>.Q, <Zt2>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.734 ST2W (scalar plus immediate)

Contiguous store two-word structures from two vectors (immediate index)

This instruction performs a contiguous store of two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 0	0 1	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST2W  { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.735 ST2W (scalar plus scalar)

Contiguous store two-word structures from two vectors (scalar index)

This instruction performs a contiguous store of two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	0 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST2W  { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let nreg : integer{} = 2;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[2]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.736 ST3B (scalar plus immediate)

Contiguous store three-byte structures from three vectors (immediate index)

This instruction performs a contiguous store of three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

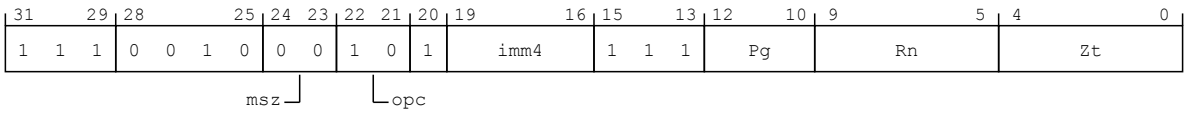

Encoding

```
ST3B  { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.737 ST3B (scalar plus scalar)

Contiguous store three-byte structures from three vectors (scalar index)

This instruction performs a contiguous store of three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0	1 0	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST3B  { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.738 ST3D (scalar plus immediate)

Contiguous store three-doubleword structures from three vectors (immediate index)

This instruction performs a contiguous store of three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

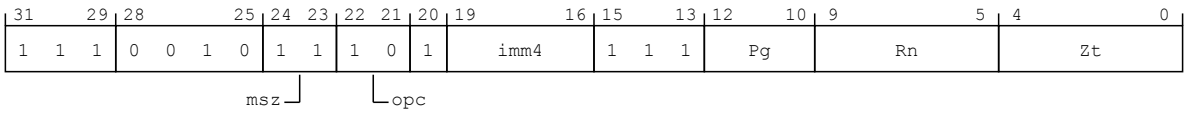

Encoding

```
ST3D  { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.739 ST3D (scalar plus scalar)

Contiguous store three-doubleword structures from three vectors (scalar index)

This instruction performs a contiguous store of three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1	1 0	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST3D  { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.740 ST3H (scalar plus immediate)

Contiguous store three-halfword structures from three vectors (immediate index)

This instruction performs a contiguous store of three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	1 0	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST3H  { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.741 ST3H (scalar plus scalar)

Contiguous store three-halfword structures from three vectors (scalar index)

This instruction performs a contiguous store of three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	1 0	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST3H  { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.742 ST3Q (scalar plus immediate)

Contiguous store three-quadword structures from three vectors (immediate index)

This instruction performs a contiguous store of three-quadword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory that make up each structure. Inactive structures are not written to memory.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	0	1 0	0 0	imm4	0	0	0	Pg	Rn	Zt
			num								

Encoding

```
ST3Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.743 ST3Q (scalar plus scalar)

Contiguous store three-quadword structures from three vectors (scalar index)

This instruction performs a contiguous store of three-quadword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory that make up each structure. Inactive structures are not written to memory.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–16	15	14	13	12–10	9–5	4–0
1 1 1	0 0 1 0	0	1 0 (num)	1	!=11111 (Rm)	0	0	0	Pg	Rn	Zt

Encoding

```
ST3Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.744 ST3W (scalar plus immediate)

Contiguous store three-word structures from three vectors (immediate index)

This instruction performs a contiguous store of three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

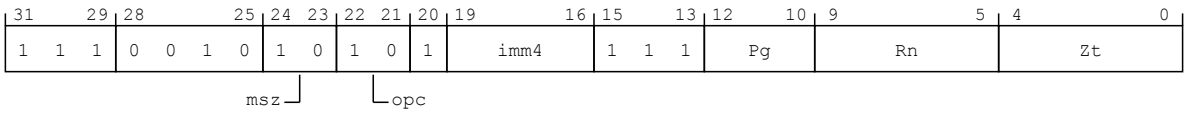

Encoding

```
ST3W  { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.745 ST3W (scalar plus scalar)

Contiguous store three-word structures from three vectors (scalar index)

This instruction performs a contiguous store of three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0 (msz)	1 0 (opc)	!=11111 (Rm)	0 1 1	Pg	Rn	Zt

Encoding

```
ST3W  { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let nreg : integer{} = 3;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[3]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.746 ST4B (scalar plus immediate)

Contiguous store four-byte structures from four vectors (immediate index)

This instruction performs a contiguous store of four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0 (msz)	1 1 (opc)	1	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
ST4B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.747 ST4B (scalar plus scalar)

Contiguous store four-byte structures from four vectors (scalar index)

This instruction performs a contiguous store of four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 0	1 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST4B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.748 ST4D (scalar plus immediate)

Contiguous store four-doubleword structures from four vectors (immediate index)

This instruction performs a contiguous store of four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 1	1 1	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST4D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.749 ST4D (scalar plus scalar)

Contiguous store four-doubleword structures from four vectors (scalar index)

This instruction performs a contiguous store of four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1	1 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST4D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.750 ST4H (scalar plus immediate)

Contiguous store four-halfword structures from four vectors (immediate index)

This instruction performs a contiguous store of four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	1 1	1	imm4	1 1 1	Pg	Rn	Zt
		msz	opc						

Encoding

```
ST4H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.751 ST4H (scalar plus scalar)

Contiguous store four-halfword structures from four vectors (scalar index)

This instruction performs a contiguous store of four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	1 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST4H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.752 ST4Q (scalar plus immediate)

Contiguous store four-quadword structures from four vectors (immediate index)

This instruction performs a contiguous store of four-quadword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory that make up each structure. Inactive structures are not written to memory.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21 20	19 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	0	1 1	0 0	imm4	0	0	0	Pg	Rn	Zt
			num								

Encoding

```
ST4Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.753 ST4Q (scalar plus scalar)

Contiguous store four-quadword structures from four vectors (scalar index)

This instruction performs a contiguous store of four-quadword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory that make up each structure. Inactive structures are not written to memory.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	0	1 1 (num)	1	!=11111 (Rm)	0	0	0	Pg	Rn	Zt

Encoding

```
ST4Q  { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 128;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.754 ST4W (scalar plus immediate)

Contiguous store four-word structures from four vectors (immediate index)

This instruction performs a contiguous store of four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index that is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

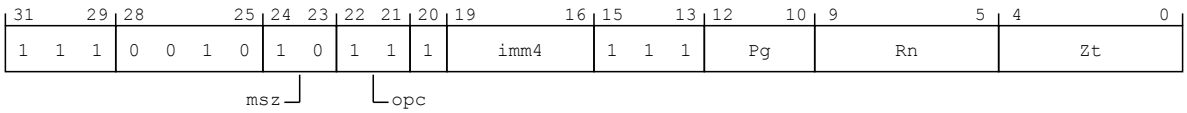

Encoding

```
ST4W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * nreg * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.755 ST4W (scalar plus scalar)

Contiguous store four-word structures from four vectors (scalar index)

This instruction performs a contiguous store of four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (`LSL` option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory that make up each structure. Inactive structures are not written to memory.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	1 1	!=11111	0 1 1	Pg	Rn	Zt
		msz	opc	Rm				

Encoding

```
ST4W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let nreg : integer{} = 4;
```

Assembler Symbols

<Zt1>

Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Zt3>

Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
let mask : bits(PL) = P{}(g);
var offset : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    values[[r]] = Z{VL}((t+r) MOD 32);
end;

for e = 0 to elements-1 do
    for r = 0 to nreg-1 do
        if ActivePredicateElement{PL}(mask, e, esize) then
            Mem{esize}(addr, accdesc) = values[[r]][e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.756 STNT1B (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		0	0	0	PNg		Rn		Zt		1
												msz								N

Encoding

```
STNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		1	0	0	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
STNT1B  { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.757 STNT1B (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	0	0	0	PNg	Rn		Zt		1
												msz								N

Encoding

```
STNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	1	0	0	PNg	Rn		Zt		0	1
													msz								N

Encoding

```
STNT1B  { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 8;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.758 STNT1B (vector plus scalar)

Scatter store non-temporal bytes

This instruction performs a non-temporal scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

31–29	28–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0 (msz)	1	0	Rm	0 0 1	Pg	Zn	Zt

Encoding

```
STNT1B  { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 8;
```

64-bit unscaled offset

(FEAT_SVE2)

31–29	28–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0 (msz)	0	0	Rm	0 0 1	Pg	Zn	Zt

Encoding

```
STNT1B  { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.759 STNT1B (scalar plus immediate, single register)

Contiguous store non-temporal bytes from vector (immediate index)

This instruction performs a contiguous non-temporal store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 0 (msz)	0 0	1	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
STNT1B  { <Zt>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var src : bits(VL);
let mask : bits(PL) = P{}(g);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.760 STNT1B (scalar plus scalar, single register)

Contiguous store non-temporal bytes from vector (scalar index)

This instruction performs a contiguous non-temporal store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

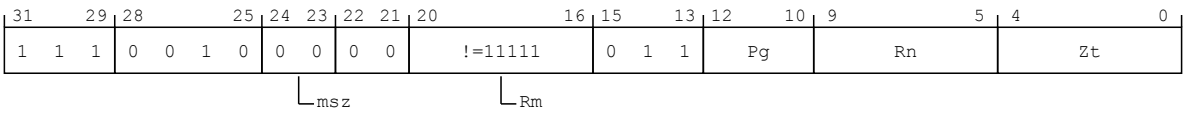

Encoding

```
STNT1B  { <Zt>.B }, <Pg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 8;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
let mask : bits(PL) = P{}(g);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated  : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.761 STNT1D (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4		1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4			0	1	1	PNg			Rn			Zt			1
																msz											N

Encoding

```
STNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4		2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4			1	1	1	PNg			Rn			Zt			0	1
																msz												N

Encoding

```
STNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.762 STNT1D (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	0	1	1	PNg	Rn		Zt		1
												msz								N

Encoding

```
STNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	1	1	1	PNg	Rn		Zt		0	1
												msz									N

Encoding

```
STNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 64;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.763 STNT1D (vector plus scalar)

Scatter store non-temporal doublewords

This instruction performs a non-temporal scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE2)

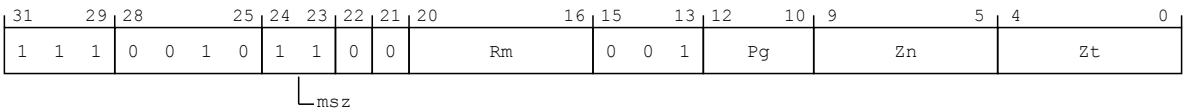

Encoding

```
STNT1D  { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 64;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.764 STNT1D (scalar plus immediate, single register)

Contiguous store non-temporal doublewords from vector (immediate index)

This instruction performs a contiguous non-temporal store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20	19–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1 (msz)	0 0	1	imm4	1 1 1	Pg	Rn	Zt

Encoding

```
STNT1D  { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var src : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mask : bits(PL) = P{}(g);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.765 STNT1D (scalar plus scalar, single register)

Contiguous store non-temporal doublewords from vector (scalar index)

This instruction performs a contiguous non-temporal store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 1	0 0	!=11111	0 1 1	Pg	Rn	Zt
		msz		Rm				

Encoding

```
STNT1D  { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mask : bits(PL) = P{}(g);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated  : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.766 STNT1H (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		0	0	1	PNg		Rn		Zt		1
												msz								N

Encoding

```
STNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		1	0	1	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
STNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.767 STNT1H (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	0	0	1	PNg	Rn		Zt		1
												msz								N

Encoding

```
STNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	1	0	1	PNg	Rn		Zt		0	1
													msz								N

Encoding

```
STNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 16;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.768 STNT1H (vector plus scalar)

Scatter store non-temporal halfwords

This instruction performs a non-temporal scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

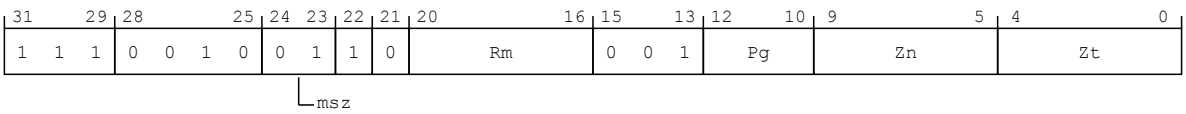

Encoding

```
STNT1H  { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 16;
```

64-bit unscaled offset

(FEAT_SVE2)

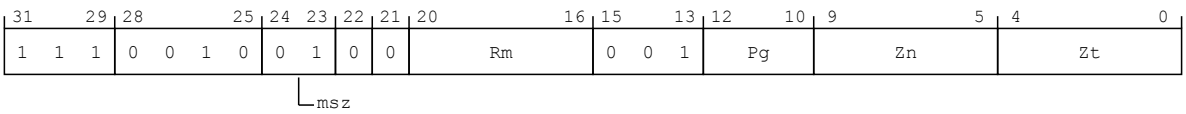

Encoding

```
STNT1H  { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 16;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.769 STNT1H (scalar plus immediate, single register)

Contiguous store non-temporal halfwords from vector (immediate index)

This instruction performs a contiguous non-temporal store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	0 1	0 0	1	imm4	1 1 1	Pg	Rn	Zt
		msz							

Encoding

```
STNT1H  { <Zt>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var src : bits(VL);
let mask : bits(PL) = P{}(g);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.770 STNT1H (scalar plus scalar, single register)

Contiguous store non-temporal halfwords from vector (scalar index)

This instruction performs a contiguous non-temporal store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–23	22–21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	0 1	0 0	!=11111	0 1 1	Pg	Rn	Zt
		msz		Rm				

Encoding

```
STNT1H  { <Zt>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 16;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mask : bits(PL) = P{}(g);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated  : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.771 STNT1W (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (immediate index)

This instruction performs a contiguous non-temporal store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		0	1	0	PNg		Rn		Zt		1
												msz								N

Encoding

```
STNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0 0 0	0	0	0	1 1	0	imm4		1	1	0	PNg		Rn		Zt		0	1
												msz									N

Encoding

```
STNT1W  { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.772 STNT1W (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (scalar index)

This instruction performs a contiguous non-temporal store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	0	1	0	PNg	Rn		Zt		1
												msz								N

Encoding

```
STNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let t : integer = UInt(Zt::'0');
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2 || FEAT_SVE2p1)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm	1	1	0	PNg	Rn		Zt		0	1
													msz								N

Encoding

```
STNT1W  { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let t : integer = UInt(Zt::'00');
let esize : integer{} = 32;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 2.

For the “Four registers” variant: is the name of the first scalable vector register to be transferred, encoded as “Zt” times 4.

<Zt2>

Is the name of the second scalable vector register to be transferred, encoded as “Zt” times 2 plus 1.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt4>

Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” times 4 plus 3.

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.773 STNT1W (vector plus scalar)

Scatter store non-temporal words

This instruction performs a non-temporal scatter store of words from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

(FEAT_SVE2)

31–29	28–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	1	0	Rm	0 0 1	Pg	Zn	Zt
		msz							

Encoding

```
STNT1W  { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let msize : integer{} = 32;
```

64-bit unscaled offset

(FEAT_SVE2)

31–29	28–25	24–23	22	21	20–16	15–13	12–10	9–5	4–0
1 1 1	0 0 1 0	1 0	0	0	Rm	0 0 1	Pg	Zn	Zt
		msz							

Encoding

```
STNT1W  { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let n : integer = UInt(Zn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 64;
let msize : integer{} = 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the base scalable vector register, encoded in the “Zn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
var base : bits(VL);
var offset : bits(64);
var src : bits(VL);
let mbytes : integer{} = msize DIV 8;
let contiguous : boolean = FALSE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if AnyActiveElement{PL}(mask, esize) then
    base = Z{VL}(n);
    offset = X{64}(m);
    src = Z{VL}(t);
end;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let baddr : bits(64) = ZeroExtend{}(base[e*:esize]);
        let addr : bits(64) = AddressAdd(baddr, offset, accdesc);
        Mem{msize}(addr, accdesc) = src[e*:esize][msize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.774 STNT1W (scalar plus immediate, single register)

Contiguous store non-temporal words from vector (immediate index)

This instruction performs a contiguous non-temporal store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20	19 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 0	0 0	1	imm4	1 1 1	Pg	Rn	Zt
		msz							

Encoding

```
STNT1W  { <Zt>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>

Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mbytes : integer{} = esize DIV 8;
var src : bits(VL);
let mask : bits(PL) = P{}(g);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated : boolean = TRUE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * elements * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.775 STNT1W (scalar plus scalar, single register)

Contiguous store non-temporal words from vector (scalar index)

This instruction performs a contiguous non-temporal store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index that is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 23	22 21	20 16	15 13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 0	0 0	!=11111	0 1 1	Pg	Rn	Zt
		msz		Rm				

Encoding

```
STNT1W  { <Zt>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt(Pg);
let esize : integer{} = 32;
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var base : bits(64);
var offset : bits(64);
var addr : bits(64);
var src : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let mask : bits(PL) = P{}(g);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
let predicated  : boolean = TRUE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if !AnyActiveElement{PL}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
    src = Z{VL}(t);
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    addr = AddressIncrement(addr, mbytes, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.776 STR (predicate)

Store predicate register

This instruction stores a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 that is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.

For programmer convenience, an assembler must also accept a predicate-as-counter register name for the source predicate register.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24–22	21–16	15	14	13	12–10	9–5	4	3–0
1 1 1	0 0 1 0	1 1 0	imm9h	0	0	0	imm9l	Rn	0	Pt

Encoding

```
STR  <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Pt);
let n : integer = UInt(Rn);
let imm : integer = SInt(imm9h::imm9l);
```

Assembler Symbols

<Pt>

Is the name of the scalable predicate transfer register, encoded in the “Pt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the “imm9h:imm9l” fields.

Operation

```
CheckSVEEnabled();
let PL : integer{} = CurrentVL() DIV 8;
let elements : integer = PL DIV 8;
var src : bits(PL);
var base : bits(64);
let offset : integer = imm * elements;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = FALSE;
let tagchecked : boolean = n != 31;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP{64}();
else
    base = X{64}(n);
end;

src = P{PL}(t);

var addr : bits(64) = AddressAdd(base, offset, accdesc);

let aligned : boolean = IsAlignedSize(base, 2);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;

for e = 0 to elements-1 do
    AArch64_MemSingle{8}(addr, accdesc, aligned) = src[e*:8];
    addr = AddressIncrement(addr, 1, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.777 STR (vector)

Store vector register

This instruction stores a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 that is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24 22	21 16	15	14	13	12 10	9 5	4 0
1 1 1	0 0 1 0	1 1 0	imm9h	0	1	0	imm9l	Rn	Zt

Encoding

```
STR  <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let t : integer = UInt(Zt);
let n : integer = UInt(Rn);
let imm : integer = SInt(imm9h::imm9l);
```

Assembler Symbols

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the “imm9h:imm9l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 8;
var src : bits(VL);
var base : bits(64);
let offset : integer = imm * elements;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let predicated : boolean = FALSE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  predicated, tagchecked);

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    base = SP{64}();
else
    base = X{64}(n);
end;

src = Z{VL}(t);

var addr : bits(64) = AddressAdd(base, offset, accdesc);

let aligned : boolean = IsAlignedSize(base, 16);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;

for e = 0 to elements-1 do
    AArch64_MemSingle{8}(addr, accdesc, aligned) = src[e*:8];
    addr = AddressIncrement(addr, 1, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.778 SUB (vectors, predicated)

Subtract (predicated)

This instruction subtracts active elements of the second source vector from the corresponding elements of the first source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 0	0 0 1 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
SUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element1 - element2;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.779 SUB (immediate)

Subtract immediate (unpredicated)

This instruction subtracts an unsigned immediate from each element of the source vector, and destructively places the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

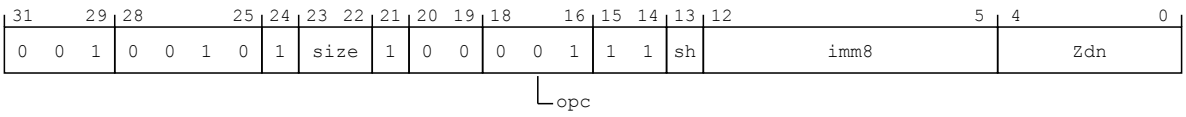

Encoding

```
SUB  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    result[e*:esize] = element1 - imm;

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.780 SUB (vectors, unpredicated)

Subtract (unpredicated)

This instruction subtracts elements of the second source vector from the corresponding elements of the first source vector, and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

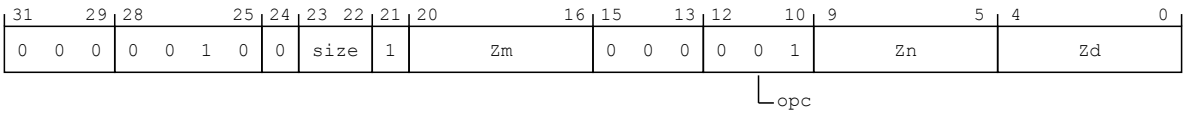

Encoding

```
SUB  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = element1 - element2;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.781 SUBHNB

Subtract narrow high part (bottom)

This instruction subtracts each vector element of the second source vector from the corresponding vector element in the first source vector, and places the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

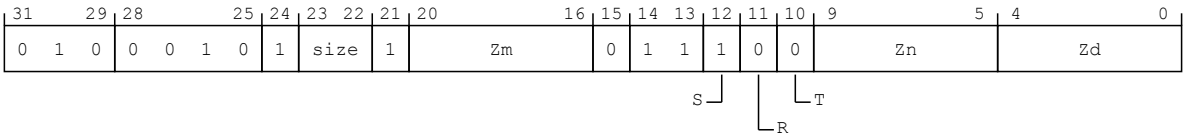

Encoding

```
SUBHNB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = (element1 - element2) >> halfesize;
    result[(2*e + 0)*:halfesize] = res[halfesize-1:0];
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.782 SUBHNT

Subtract narrow high part (top)

This instruction subtracts each vector element of the second source vector from the corresponding vector element in the first source vector, and places the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

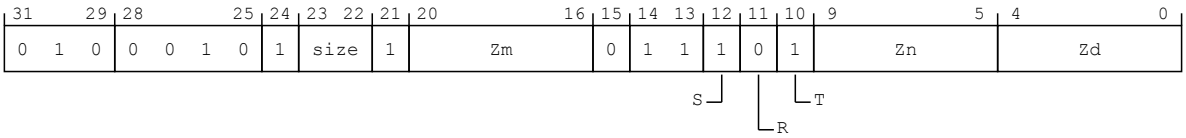

Encoding

```
SUBHNT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let res : integer = (element1 - element2) >> halfesize;
    result[(2*e + 1)*:halfesize] = res[halfesize-1:0];

end;
Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.783 SUBPT (predicated)

Subtract checked pointer vectors (predicated)

This instruction subtracts, with pointer check, Active elements of the second source vector from corresponding elements of the first source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE && FEAT_CPA)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	1 1 (size)	0	0 0	1 0 1 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
SUBPT  <Zdn>.D, <Pg>/M, <Zdn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_CPA) then
    EndOfDecode(Decode_UNDEF);
end;
let g : integer =  UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer =  UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV 64;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, 64) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand1[e*:64];
    let element2 : bits(64) = operand2[e*:64];
    if ActivePredicateElement{PL}(mask, e, 64) then
        let res : bits(64) = element1 - element2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:64] = PointerAddCheck(res, element1);
    else
        result[e*:64] = operand1[e*:64];
    end;

    Z{VL}(dn) = result;
end;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.784 SUBPT (unpredicated)

Subtract checked pointer vectors (unpredicated)

This instruction subtracts, with pointer check, all elements of the second source vector from corresponding elements of the first source vector and places the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE2

(FEAT_SVE && FEAT_CPA)

31 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	1 1	1	Zm	0 0 0	0 1 1	Zn	Zd
			size				opc		

Encoding

```
SUBPT  <Zd>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_CPA) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 64;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(64) = operand1[e*:64];
    let element2 : bits(64) = operand2[e*:64];
    let res : bits(64) = element1 - element2;
    result[e*:64] = PointerAddCheck(res, element1);
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.785 SUBR (vectors)

Reversed subtract (predicated)

This instruction performs a reversed subtraction on active elements of the first source vector from the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 0	0 1 1 (opc)	0 0 0	Pg	Zm	Zdn

Encoding

```
SUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element2 - element1;
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.786 SUBR (immediate)

Reversed subtract from immediate (unpredicated)

This instruction performs a reversed subtraction from an unsigned immediate each element of the source vector, and destructively places the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–14	13	12–5	4–0
0 0 1	0 0 1 0	1	size	1	0 0	0 1 1 (opc)	1 1	sh	imm8	Zdn

Encoding

```
SUBR  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    result[e*:esize] = (imm - element1)[esize-1:0];

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.787 SUDOT

Signed by unsigned 8-bit integer dot product by indexed element to 32-bit integer

This instruction calculates the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_I8MM)

31 29	28 25	24	23 22	21	20 19	18 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 1 1	1	Zn	Zda
			size					U		

Encoding

```
SUDOT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_I8MM)) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        let element2 : integer = UInt(operand2[(4 * s + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.788 SUNPKHI, SUNPKLO

Signed unpack and extend half of vector

This instruction unpacks and sign-extends elements from the lowest or highest half of the source vector, and writes the sign-extended values into double-width elements of the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

High half

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	1 0	0	0	1	0 0 1 1 1 0	Zn	Zd

U H

Encoding

```
SUNPKHI  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let hi : boolean = TRUE;
```

Low half

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	1 0	0	0	0	0 0 1 1 1 0	Zn	Zd

U H

Encoding

```
SUNPKLO  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let hi : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let hsize : integer{} = esize DIV 2;
let offset : integer = if hi then elements else 0;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(hsize) = operand[(e + offset)*:hsize];
    result[e*:esize] = Extend{esize}(element, unsigned);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.789 SUQADD

Signed saturating unsigned add

This instruction adds active unsigned elements of the source vector to the corresponding signed elements of the addend vector, and destructively places the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

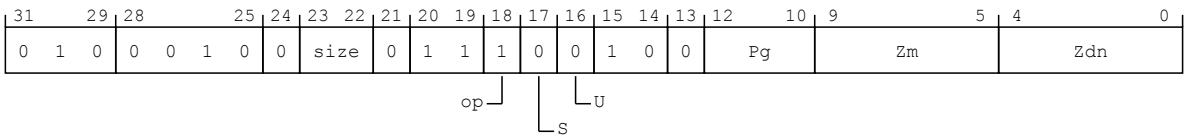

Encoding

```
SUQADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = SignedSat{esize}(SInt(element1) + UInt(element2));
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.790 SXTB, SXTH, SXTW

Signed byte / halfword / word extend (predicated)

This instruction sign-extends the least-significant sub-element of each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 6 classes: Byte, merging, Byte, zeroing, Halfword, merging, Halfword, zeroing, Word, merging, and Word, zeroing

Byte, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	0	0 0	0	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
SXTB  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let merging : boolean = TRUE;
```

Byte, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	0	0 0	0	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
SXTB  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let merging : boolean = FALSE;
```

Halfword, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1	0	0 1	0	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
SXTH  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let merging : boolean = TRUE;
```

Halfword, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0	0	0 1	0	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
SXTH  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let merging : boolean = FALSE;
```

Word, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1	0	1 0	0	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
SXTW  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let merging : boolean = TRUE;
```

Word, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0	0	1 0	0	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
SXTW  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = FALSE;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

For the “Byte, merging” and “Byte, zeroing” variants: is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

For the “Halfword, merging” and “Halfword, zeroing” variants: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = Extend{esize}(element[s_esize-1:0], unsigned);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.791 TBL

Programmable table lookup in one or two vector table (zeroing)

This instruction reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements consisting of one or two consecutive vector registers, where the first or only vector holds the lower numbered elements, and places the indexed table element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements, then it places zero in the corresponding destination vector element.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

It has encodings from 2 classes: Single register table and Two register table

Single register table

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 0 1 1 0 0	Zn	Zd

Encoding

```
TBL  <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let double_table : boolean = FALSE;
```

Two register table

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 0 1 0 1	0 (op)	Zn	Zd

Encoding

```
TBL  <Zd>.<T>, { <Zn1>.<T>, <Zn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let double_table : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded in the “Zn” field.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let indexes : bits(VL) = Z{}(m);
var result : bits(VL);
let table_size : integer{} = if double_table then VL*2 else VL;
let table_elems : integer = table_size DIV esize;
var table : bits(table_size);

if double_table then
    let top : bits(VL) = Z{}((n + 1) MOD 32);
    let bottom : bits(VL) = Z{}(n);
    table = (top::bottom)[table_size-1:0];
else
    table = Z{table_size}(n);
end;

for e = 0 to elements-1 do
    let idx : integer = UInt(indexes[e*:esize]);
    result[e*:esize] = if idx < table_elems then table[idx*:esize] else Zeros{esize};
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.792 TBLQ

Programmable table lookup within each quadword vector segment (zeroing)

This instruction reads, for each 128-bit destination vector segment, each element of the corresponding second source (index) vector segment and uses its value to select an indexed element from the corresponding first source (table) vector segment. The indexed table element is placed in the element of the destination vector that corresponds to the index vector element. If an index value is greater than or equal to the number of elements in a 128-bit vector segment then it places zero in the corresponding destination vector element. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–16	15–13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	Zm	1 1 1	1 1 0	Zn	Zd
							opc		

Encoding

```
TBLQ  <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    for e = 0 to elements-1 do
        let idx : integer = UInt(operand2[(s * elements + e)*:esize]);
        if idx < elements then
            result[(s * elements + e)*:esize] = operand1[(s * elements + idx)*:esize];
        else
            result[(s * elements + e)*:esize] = Zeros{esize};
        end;
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.793 TBX

Programmable table lookup in single vector table (merging)

This instruction reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements in the first source vector, and places the indexed element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements, then the corresponding destination vector element is left unchanged.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 0 1 0 1	1 (op)	Zn	Zd

Encoding

```
TBX  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);

for e = 0 to elements-1 do
    let element2 : integer = UInt(operand2[e*:esize]);
    if element2 < elements then
        result[e*:esize] = operand1[element2*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.794 TBXQ

Programmable table lookup within each quadword vector segment (merging)

This instruction reads, for each 128-bit destination vector segment, each element of the corresponding second source (index) vector segment and uses its value to select an indexed element from the corresponding first source (table) vector segment. The indexed table element is placed in the element of the destination vector that corresponds to the index vector element. If an index value is greater than or equal to the number of elements in a 128-bit vector segment then the corresponding destination vector element is left unchanged. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 16	15 10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	Zm	0 0 1 1 0 1	Zn	Zd

Encoding

```
TBXQ  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(d);

for s = 0 to segments-1 do
    for e = 0 to elements-1 do
        let idx : integer = UInt(operand2[(s * elements + e)*:esize]);
        if idx < elements then
            result[(s * elements + e)*:esize] = operand1[(s * elements + idx)*:esize];
        end;
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.795 TRN1, TRN2 (predicates)

Interleave even or odd elements from two predicates

This instruction interleaves alternating even or odd-numbered elements from the first and second source predicates, and places the values in the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: Even and Odd

Even

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–11	10	9	8–5	4	3–0
0 0 0	0 0 1 0	1	size	1	0	Pm	0 1 0	1 0 (opc)	0 (H)	0	Pn	0	Pd

Encoding

```
TRN1  <Pd>.<T>, <Pn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let part : integer = 0;
```

Odd

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–11	10	9	8–5	4	3–0
0 0 0	0 0 1 0	1	size	1	0	Pm	0 1 0	1 0 (opc)	1 (H)	0	Pn	0	Pd

Encoding

```
TRN2  <Pd>.<T>, <Pn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let part : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let pairs : integer = VL DIV (esize * 2);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);

for p = 0 to pairs-1 do
    result[(2*p+0)*:(esize DIV 8)] = operand1[(2*p+part)*:(esize DIV 8)];
    result[(2*p+1)*:(esize DIV 8)] = operand2[(2*p+part)*:(esize DIV 8)];
end;

P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.796 TRN1, TRN2 (vectors)

Interleave even or odd elements from two vectors

This instruction interleaves alternating even or odd-numbered elements from the first and second source vectors, and places the values in the destination vector. This instruction is unpredicated.

The 128-bit element variant requires that the Effective SVE vector length is at least 256 bits. ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: Even, Even (quadwords), Odd, and Odd (quadwords)

Even

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 1 1	1 0	0	Zn	Zd
								H		

Encoding

```
TRN1  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```

Even (quadwords)

(FEAT_F64MM)

31–29	28–25	24–23	22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1 1	0	1	Zm	0 0 0	1 1	0	Zn	Zd
							opc	H		

Encoding

```
TRN1  <Zd>.Q, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Odd

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	Zm	0 1 1	1 0	1	Zn	Zd
								H		

Encoding

```
TRN2  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```

Odd (quadwords)

(FEAT_F64MM)

31 29	28 25	24 23	22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	1 1	0	1	Zm	0 0 0	1 1	1	Zn	Zd
							opc	H		

Encoding

```
TRN2  <Zd>.Q, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if esize < 128 then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL < esize * 2 then EndOfDecode(Decode_UNDEF); end;
let pairs : integer = VL DIV (esize * 2);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Zeros{};

for p = 0 to pairs-1 do
    result[(2*p+0)*:esize] = operand1[(2*p+part)*:esize];
    result[(2*p+1)*:esize] = operand2[(2*p+part)*:esize];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.797 UABA

Unsigned absolute difference and accumulate

This instruction calculates the absolute difference between unsigned integer values in elements of the first source vector and the corresponding elements of the second source vector, and adds the difference to the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 14	13 11	10	9 5	4 0
0 1 0	0 0 1 0	1	size	0	Zm	1 1	1 1 1	1	Zn	Zda
								U		

Encoding

```
UABA  <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let absdiff : bits(esize) = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.798 UABALB

Unsigned absolute difference and accumulate long (bottom)

This instruction calculates the absolute difference between even-numbered unsigned elements of the first source vector and the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

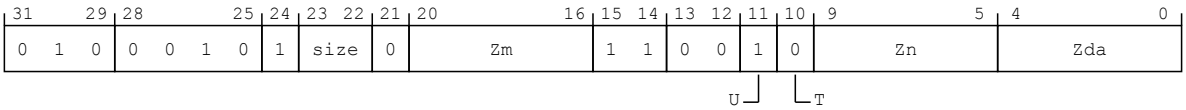

Encoding

```
UABALB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.799 UABALT

Unsigned absolute difference and accumulate long (top)

This instruction calculates the absolute difference between odd-numbered unsigned elements of the first source vector and the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

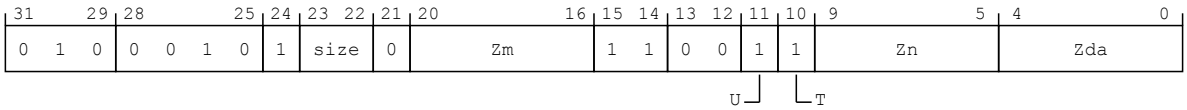

Encoding

```
UABALT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let absdiff = Abs(element1 - element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + absdiff;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.800 UABD

Unsigned absolute difference (predicated)

This instruction calculates the absolute difference between unsigned integer values in active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 1	1 0 (opc)	1 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
UABD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let absdiff : integer = Abs(element1 - element2);
        result[e*:esize] = absdiff[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.801 UABDLB

Unsigned absolute difference long (bottom)

This instruction calculates the absolute difference between the even-numbered unsigned integer values in elements of the first source vector and the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

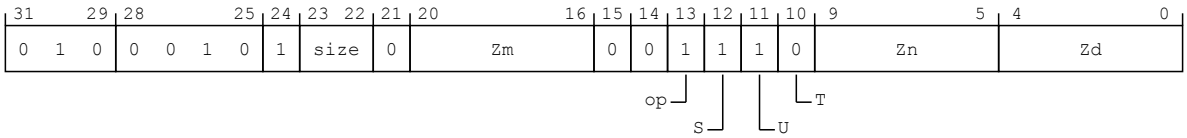

Encoding

```
UABDLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let res : integer = Abs(element1 - element2);
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.802 UABDLT

Unsigned absolute difference long (top)

This instruction calculates the absolute difference between the odd-numbered unsigned integer values in elements of the first source vector and the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

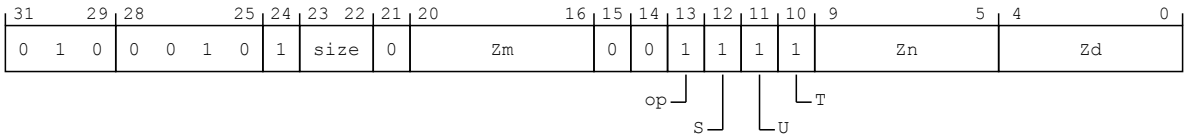

Encoding

```
UABDLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let res : integer = Abs(element1 - element2);
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.803 UADALP

Unsigned add and accumulate long pairwise

This instruction adds pairs of adjacent unsigned integer values and accumulates the results into the overlapping double-width elements of the destination vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–17	16	15–14	13	12–10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	0 0 1 0	1	1 0	1	Pg	Zn	Zda
						U					

Encoding

```
UADALP  <Zda>.<T>, <Pg>/M, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the second source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand_acc : bits(VL) = Z{}(da);
let operand_src : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand_acc[e*:esize];
    else
        let element1 : integer = UInt(operand_src[(2*e + 0)*:(esize DIV 2)]);
        let element2 : integer = UInt(operand_src[(2*e + 1)*:(esize DIV 2)]);
        let sum : bits(esize) = (element1 + element2)[esize-1:0];
        result[e*:esize] = operand_acc[e*:esize] + sum;
    end;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.804 UADDLB

Unsigned add long (bottom)

This instruction adds the corresponding even-numbered unsigned elements of the first and second source vectors, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

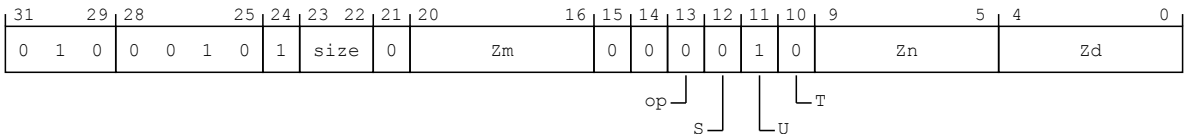

Encoding

```
UADDLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 0;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 + element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.805 UADDLT

Unsigned add long (top)

This instruction adds the corresponding odd-numbered unsigned elements of the first and second source vectors, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

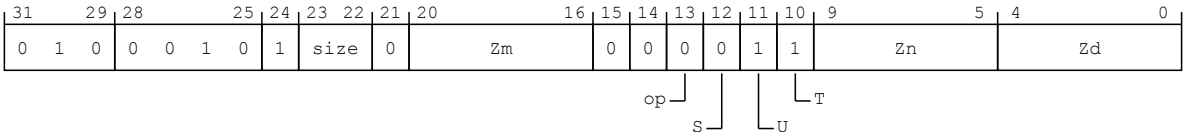

Encoding

```
UADDLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 1;
let sel2 : integer = 1;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 + element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.806 UADDV

Unsigned add reduction to scalar

This instruction adds all the unsigned elements of the source vector, and places the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 0 0	0	1	0 0 1	Pg	Zn	Vd
						op	U				

Encoding

```
UADDV  <Dd>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
```

Assembler Symbols

<Dd>

Is the 64-bit name of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var sum : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand[e*:esize]);
        sum = sum + element;

    end;
end;
V{64}(d) = sum[63:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.807 UADDWB

Unsigned add wide (bottom)

This instruction adds the even-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

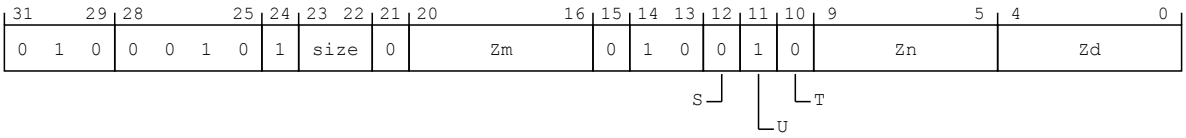

Encoding

```
UADDWB  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    result[e*:esize] = (element1 + element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.808 UADDWT

Unsigned add wide (top)

This instruction adds the odd-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

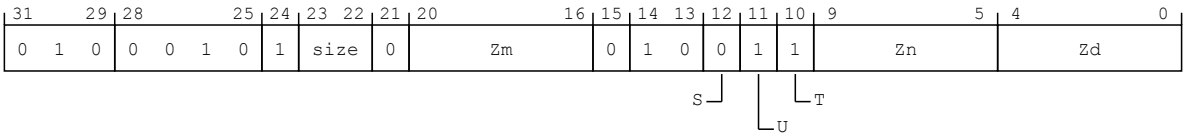

Encoding

```
UADDWT  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    result[e*:esize] = (element1 + element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.809 UCLAMP

Unsigned clamp to minimum/maximum

This instruction clamps each unsigned element in the destination vector to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively writes the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SME || FEAT_SVE2p1)

31–29	28–25	24	23–22	21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	Zm	1 1 0 0 0	1	Zn	Zd
							U		

Encoding

```
UCLAMP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(d);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    let element3 : integer = UInt(operand3[e*:esize]);
    let res : integer = Min(Max(element1, element3), element2);
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.810 UCVTF (predicated)

Unsigned integer convert to floating-point (predicated)

This instruction converts to floating-point from the unsigned integer in each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 14 classes: 16-bit to half-precision, merging, 16-bit to half-precision, zeroing, 32-bit to half-precision, merging, 32-bit to half-precision, zeroing, 32-bit to single-precision, merging, 32-bit to single-precision, zeroing, 32-bit to double-precision, merging, 32-bit to double-precision, zeroing, 64-bit to half-precision, merging, 64-bit to half-precision, zeroing, 64-bit to single-precision, merging, 64-bit to single-precision, zeroing, 64-bit to double-precision, merging, and 64-bit to double-precision, zeroing

16-bit to half-precision, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–17	16	15–13	12–10	9–5	4–0
0 1 1	0 0 1 0	1	0 1	0	1 0	0 1	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.H, <Pg>/M, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

16-bit to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21–19	18–17	16	15	14	13	12–10	9–5	4–0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 0	0	1	1	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.H, <Pg>/Z, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 16;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

32-bit to half-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 1	0	1 0	1 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.H, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

32-bit to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 0	1	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.H, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

32-bit to single-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 0	0	1 0	1 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.S, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

32-bit to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 0	0 1 1	1 0	1	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.S, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

32-bit to double-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 0	0 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.D, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

32-bit to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 0	0	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.D, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 32;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

64-bit to half-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	0 1	0	1 0	1 1	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.H, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

64-bit to half-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	0 1	0 1 1	1 0	1	1	1	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.H, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 16;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

64-bit to single-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 0	1 0	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.S, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

64-bit to single-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 0	1	1	0	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.S, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 32;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

64-bit to double-precision, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 1 1	0 0 1 0	1	1 1	0	1 0	1 1	1	1 0 1	Pg	Zn	Zd
			opc			opc2	int_U				

Encoding

```
UCVTF  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = TRUE;
```

64-bit to double-precision, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21 19	18 17	16	15	14	13	12 10	9 5	4 0
0 1 1	0 0 1 0	0	1 1	0 1 1	1 0	1	1	1	1	Pg	Zn	Zd
			opc			o2		o3	int_U			

Encoding

```
UCVTF  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let s_esize : integer{} = 64;
let d_esize : integer{} = 64;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        let fpval : bits(d_esize) = FixedToFP{d_esize, s_esize}(element[s_esize-1:0], 0, unsigned,
                                                                FPCR(), rounding);
        result[e*:esize] = ZeroExtend{esize}(fpval);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

For the 16-bit to half-precision, merging, 32-bit to half-precision, merging, 32-bit to single-precision, merging, 32-bit to double-precision, merging, 64-bit to half-precision, merging, 64-bit to single-precision, merging, 64-bit to double-precision, merging variants:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.811 UDIV

Unsigned divide (predicated)

This instruction performs an unsigned division on active elements of the first source vector by the corresponding elements of the second source vector and destructively places the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 0 1	0	1	0 0 0	Pg	Zm	Zdn
						R	U				

Encoding

```
UDIV  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let dividend : integer = UInt(operand1[e*:esize]);
        let divisor : integer = UInt(operand2[e*:esize]);
        var quotient : integer;
        if divisor == 0 then
            quotient = 0;
        else
            quotient = dividend DIVRM divisor;
        end;
        result[e*:esize] = quotient[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.812 UDIVR

Unsigned reversed divide (predicated)

This instruction performs an unsigned reversed division on active elements of the second source vector by the corresponding elements of the first source vector, and destructively places the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1 0 1	1 (R)	1 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
UDIVR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let divisor : integer = UInt(operand1[e*:esize]);
        let dividend : integer = UInt(operand2[e*:esize]);
        var quotient : integer;
        if divisor == 0 then
            quotient = 0;
        else
            quotient = dividend DIVRM divisor;
        end;
        result[e*:esize] = quotient[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.813 UDOT (2-way, vectors)

Unsigned integer dot product (two-way)

This instruction calculates the dot product of a pair of unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a pair of unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

16-bit to 32-bit

(FEAT_SME2 || FEAT_SVE2p1)

31–29	28–25	24–21	20–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0 0 0 0	Zm	1 1 0 0 1	1	Zn	Zda
					U		

Encoding

```
UDOT  <Zda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 1 do
        let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
        let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.814 UDOT (2-way, indexed)

Unsigned integer dot product by indexed element (two-way)

This instruction calculates the dot product of a pair of unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a pair of unsigned 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

16-bit to 32-bit

(FEAT_SME2 || FEAT_SVE2p1)

31 29	28 25	24 21	20 19	18 16	15 11	10	9 5	4 0
0 1 0	0 0 1 0	0 1 0 0	i2	Zm	1 1 0 0 1	1	Zn	Zda
						U		

Encoding

```
UDOT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 1 do
        let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
        let element2 : integer = UInt(operand2[(2 * s + i)*:(esize DIV 2)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.815 UDOT (4-way, vectors)

Unsigned integer dot product (four-way)

This instruction calculates the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15	14–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 x (size)	0	Zm	0	0 0 0 0	1 (U)	Zn	Zda

Encoding

```
UDOT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size<0>”:

size<0>	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.816 UDOT (4-way, indexed)

Unsigned integer dot product by indexed element (four-way)

This instruction calculates the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 32-bit and 16-bit to 64-bit

8-bit to 32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i2	Zm	0 0 0 0 0	1	Zn	Zda
			size					U		

Encoding

```
UDOT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

16-bit to 64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1	1	i1	Zm	0 0 0 0 0	1	Zn	Zda
			size					U		

Encoding

```
UDOT  <Zda>.D, <Zn>.H, <Zm>.H[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let index : integer = UInt(i1);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “8-bit to 32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “16-bit to 64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “8-bit to 32-bit” variant: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “16-bit to 64-bit” variant: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        let element2 : integer = UInt(operand2[(4 * s + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.817 UHADD

Unsigned halving add

This instruction adds active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shifts right one bit, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

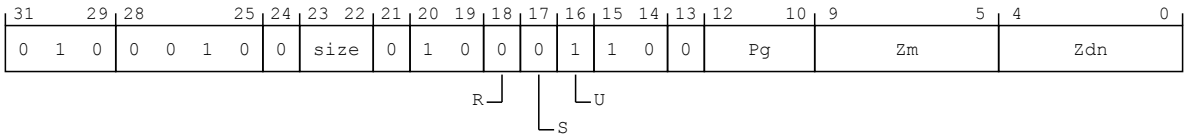

Encoding

```
UHADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 + element2) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.818 UHSUB

Unsigned halving subtract

This instruction subtracts active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector, shifts right one bit, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

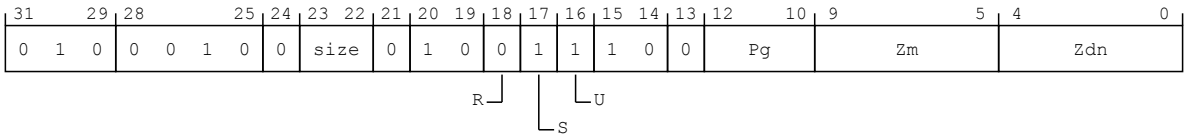

Encoding

```
UHSUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 - element2) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.819 UHSUBR

Unsigned halving subtract reversed

This instruction subtracts active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector, shifts right one bit, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

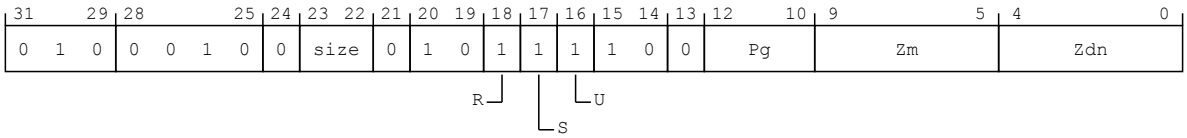

Encoding

```
UHSUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element2 - element1) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.820 UMAX (vectors)

Unsigned maximum (predicated)

This instruction determines the unsigned maximum of active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0 1	0 0 (opc)	1 (U)	0 0 0	Pg	Zm	Zdn

Encoding

```
UMAX  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let maximum : integer = Max(element1, element2);
        result[e*:esize] = maximum[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.821 UMAX (immediate)

Unsigned maximum with immediate (unpredicated)

This instruction determines the unsigned maximum of an immediate and each element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

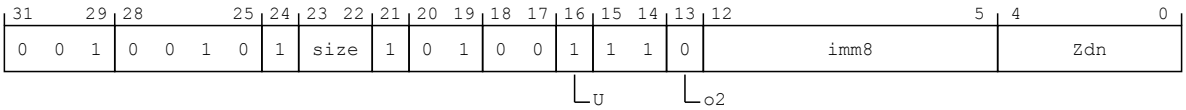

Encoding

```
UMAX  <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let unsigned : boolean = TRUE;
let imm : integer = if unsigned then UInt(imm8) else SInt(imm8);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is the unsigned immediate operand, in the range 0 to 255, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    result[e*:esize] = Max(element1, imm)[esize-1:0];

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.822 UMAXP

Unsigned maximum pairwise

This instruction calculates the maximum value of each pair of adjacent unsigned integer elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

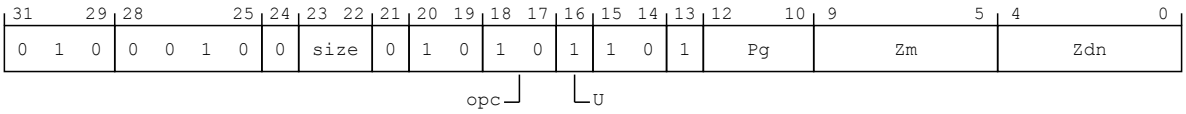

Encoding

```
UMAXP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[e*:esize];
    else
        if IsEven(e) then
            element1 = UInt(operand1[(e + 0)*:esize]);
            element2 = UInt(operand1[(e + 1)*:esize]);
        else
            element1 = UInt(operand2[(e - 1)*:esize]);
            element2 = UInt(operand2[(e + 0)*:esize]);
        end;
        let res : integer = Max(element1, element2);
        result[e*:esize] = res[esize-1:0];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.823 UMAXQV

Unsigned maximum reduction of quadword vector segments

This instruction determines the unsigned maximum of the elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as zero.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

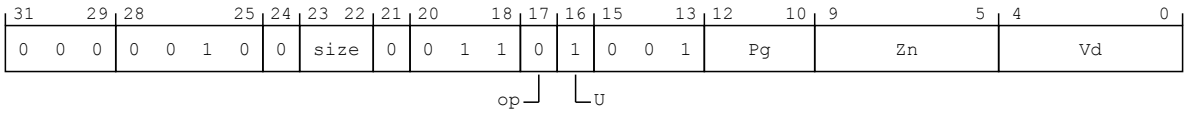

Encoding

```
UMAXQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : integer;

for e = 0 to elempersegment-1 do
    dtmp = if unsigned then 0 else -(2^(esize-1));
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = Max(dtmp, UInt(stmp[e*:esize]));
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.824 UMAXV

Unsigned maximum reduction to scalar

This instruction determines the unsigned maximum of the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

SVE

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–18	17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0 1 0	0	1	0 0 1	Pg	Zn	Vd
						op	U				

Encoding

```
UMAXV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var maximum : integer = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let opelt : bits(esize) = operand[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        maximum = Max(maximum, element);

    end;
end;
V{esize}(d) = maximum[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.825 UMIN (vectors)

Unsigned minimum (predicated)

This instruction determines the unsigned minimum of active elements of the first source vector and the corresponding elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

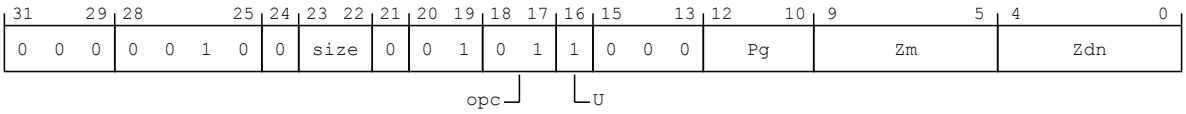

Encoding

```
UMIN  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let minimum : integer = Min(element1, element2);
        result[e*:esize] = minimum[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.826 UMIN (immediate)

Unsigned minimum with immediate (unpredicated)

This instruction determines the unsigned minimum of an immediate and each element of the source vector, and destructively places the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

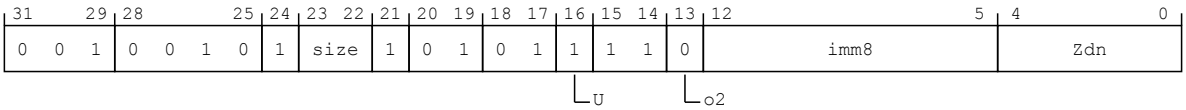

Encoding

```
UMIN  <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
let unsigned : boolean = TRUE;
let imm : integer = if unsigned then UInt(imm8) else SInt(imm8);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm>

Is the unsigned immediate operand, in the range 0 to 255, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    result[e*:esize] = Min(element1, imm)[esize-1:0];

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.827 UMINP

Unsigned minimum pairwise

This instruction calculates the minimum value of each pair of adjacent unsigned integer elements within each source vector, and interleaves the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

SVE2

(FEAT_SVE2 || FEAT_SME)

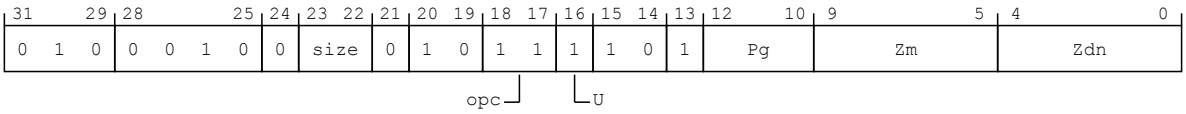

Encoding

```
UMINP  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);
var element1 : integer;
var element2 : integer;

for e = 0 to elements-1 do
    if !ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = operand1[e*:esize];
    else
        if IsEven(e) then
            element1 = UInt(operand1[(e + 0)*:esize]);
            element2 = UInt(operand1[(e + 1)*:esize]);
        else
            element1 = UInt(operand2[(e - 1)*:esize]);
            element2 = UInt(operand2[(e + 0)*:esize]);
        end;
        let res : integer = Min(element1, element2);
        result[e*:esize] = res[esize-1:0];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.828 UMINQV

Unsigned minimum reduction of quadword vector segments

This instruction determines the unsigned minimum of the elements in the same position from each 128-bit source vector segment, and places each result into the corresponding element of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31 29	28 25	24	23 22	21	20 18	17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0 1 1	1 (op)	1 (U)	0 0 1	Pg	Zn	Vd

Encoding

```
UMINQV  <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Vd>

Is the name of the destination SIMD&FP register, encoded in the “Vd” field.

<T>

Is an arrangement specifier, encoded in “size”:

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let segments : integer = VL DIV 128;
let elempersegment : integer = 128 DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(128) = Zeros{};
var stmp : bits(128) = Zeros{};

var dtmp : integer;

for e = 0 to elempersegment-1 do
    dtmp = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
    for s = 0 to segments-1 do
        if ActivePredicateElement{PL}(mask, s * elempersegment + e, esize) then
            stmp = operand[s*:128];
            dtmp = Min(dtmp, UInt(stmp[e*:esize]));
        end;
    end;
    result[e*:esize] = dtmp[esize-1:0];
end;

V{128}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.829 UMINV

Unsigned minimum reduction to scalar

This instruction determines the unsigned minimum of the elements of the source vector, and places the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

SVE

(FEAT_SVE || FEAT_SME)

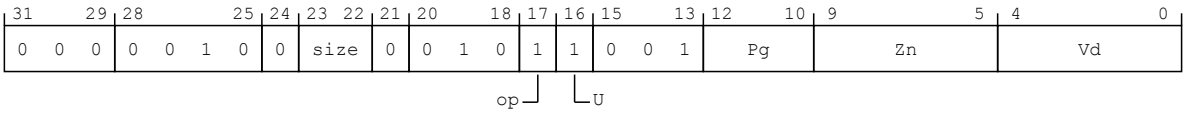

Encoding

```
UMINV  <V><d>, <Pg>, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Vd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d>

Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var minimum : integer = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let opelt : bits(esize) = operand[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        minimum = Min(minimum, element);

    end;
end;
V{esize}(d) = minimum[esize-1:0];
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.830 UMLALB (vectors)

Unsigned multiply-add long (bottom)

This instruction multiplies the even-numbered unsigned elements of the first source vector by the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

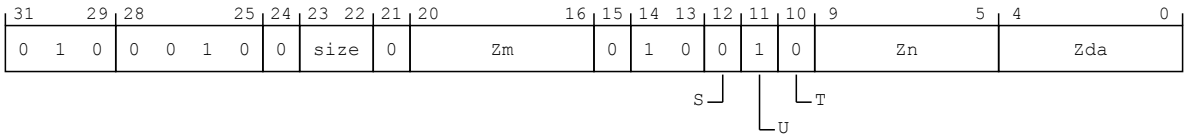

Encoding

```
UMLALB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.831 UMLALB (indexed)

Unsigned multiply-add long by indexed element (bottom)

This instruction multiplies the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and destructively adds the results to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

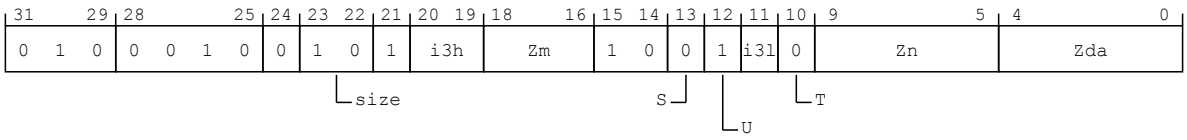

Encoding

```
UMLALB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

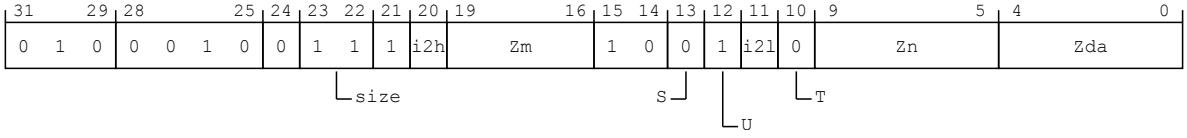

Encoding

```
UMLALB  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = UInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = UInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.832 UMLALT (vectors)

Unsigned multiply-add long (top)

This instruction multiplies the odd-numbered unsigned elements of the first source vector by the corresponding elements of the second source vector, and destructively adds the results to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

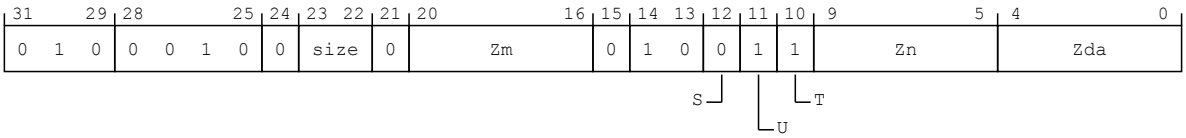

Encoding

```
UMLALT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.833 UMLALT (indexed)

Unsigned multiply-add long by indexed element (top)

This instruction multiplies the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and destructively adds the results to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

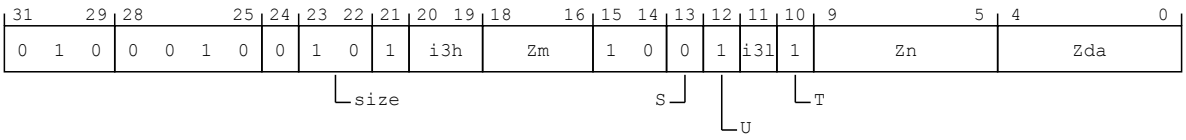

Encoding

```
UMLALT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

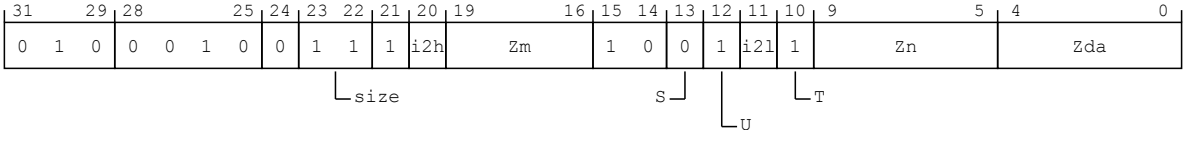

Encoding

```
UMLALT  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = UInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = UInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] + product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.834 UMLSLB (vectors)

Unsigned multiply-subtract long (bottom)

This instruction multiplies the even-numbered unsigned elements of the first source vector by the corresponding elements of the second source vector, and destructively subtracts the results from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

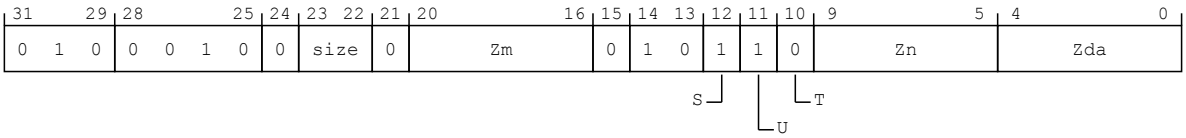

Encoding

```
UMLSLB  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.835 UMLSLB (indexed)

Unsigned multiply-subtract long by indexed element (bottom)

This instruction multiplies the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and destructively subtracts the results from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

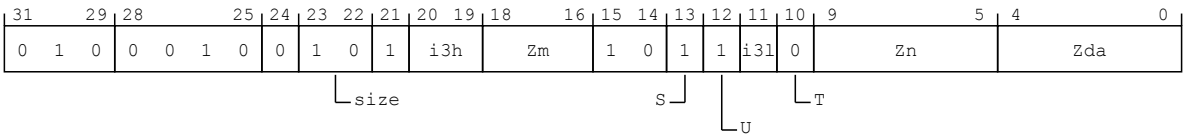

Encoding

```
UMLSLB  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

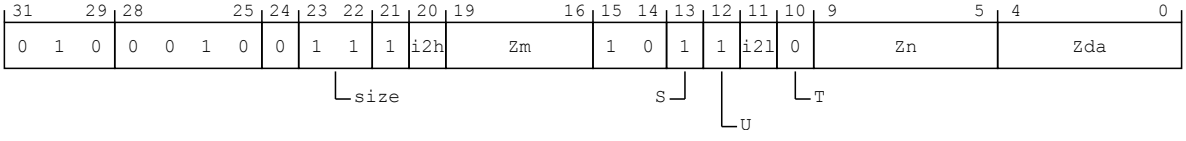

Encoding

```
UMLSLB  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 0;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = UInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = UInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.836 UMLSLT (vectors)

Unsigned multiply-subtract long (top)

This instruction multiplies the odd-numbered unsigned elements of the first source vector by the corresponding elements of the second source vector, and destructively subtracts the results from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

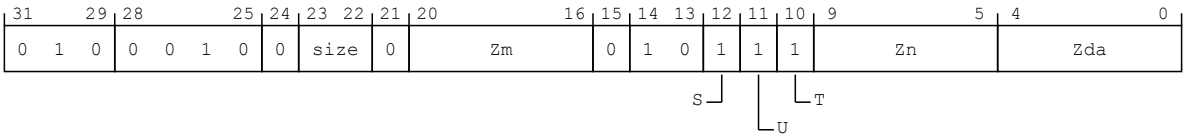

Encoding

```
UMLSLT  <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let product = (element1 * element2)[esize-1:0];
    result[e*:esize] = result[e*:esize] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.837 UMLSLT (indexed)

Unsigned multiply-subtract long by indexed element (top)

This instruction multiplies the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and destructively subtracts the results from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

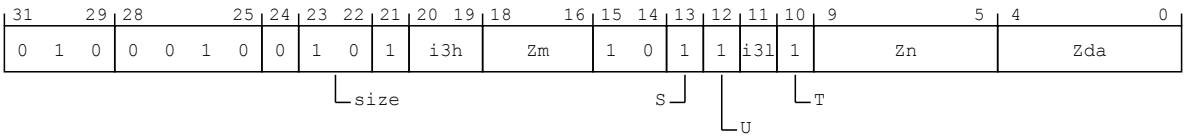

Encoding

```
UMLSLT  <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

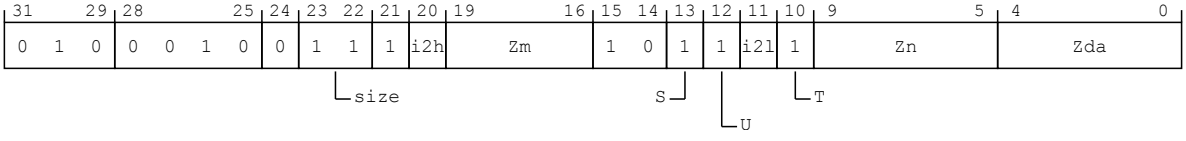

Encoding

```
UMLSLT  <Zda>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let sel : integer = 1;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Z{}(da);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = UInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = UInt(operand2[(2 * s + index)*:esize]);
    let product : bits(2*esize) = (element1 * element2)[2*esize-1:0];
    result[e*:(2*esize)] = result[e*:(2*esize)] - product;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.838 UMMLA

Unsigned 8-bit integer matrix multiply-accumulate to 32-bit integer

This instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE && FEAT_I8MM)

31–29	28–25	24	23–22	21	20–16	15–14	13–10	9–5	4–0
0 1 0	0 0 1 0	1	1 1 (uns)	0	Zm	1 0	0 1 1 0	Zn	Zda

Encoding

```
UMMLA  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_I8MM) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL) = Zeros{};
var op1, op2 : bits(128);
var res, addend : bits(128);

for s = 0 to segments-1 do
    op1    = operand1[s*:(128)];
    op2    = operand2[s*:(128)];
    addend = operand3[s*:(128)];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    result[s*:(128)] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.839 UMULH (predicated)

Unsigned multiply returning high half (predicated)

This instruction performs a widening multiplication of unsigned integer values in active elements of the first source vector by the corresponding elements of the second source vector and destructively places the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 18	17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1 0 0	1	1	0 0 0	Pg	Zm	Zdn
						H	U				

Encoding

```
UMULH  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let product : integer = (element1 * element2) >> esize;
        result[e*:esize] = product[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.840 UMULH (unpredicated)

Unsigned multiply returning high half (unpredicated)

This instruction performs a widening multiplication of unsigned integer values of all elements of the first source vector by the corresponding elements of the second source vector and places the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12	11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 1 1	0	1	1	Zn	Zd
									U		

Encoding

```
UMULH  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let product : integer = (element1 * element2) >> esize;
    result[e*:esize] = product[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.841 UMULLB (vectors)

Unsigned multiply long (bottom)

This instruction multiplies the even-numbered unsigned elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

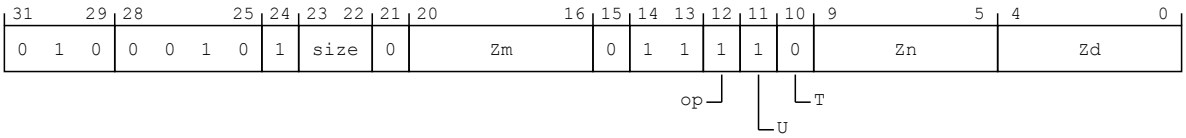

Encoding

```
UMULLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 0)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    let res : integer = element1 * element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.842 UMULLB (indexed)

Unsigned multiply long by indexed element (bottom)

This instruction multiplies the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and places the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–13	12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 0	1	i3h	Zm	1 1 0	1	i3l	0	Zn	Zd
			size					U		T		

Encoding

```
UMULLB  <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 0;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12	11	10	9–5	4–0
0 1 0	0 0 1 0	0	1 1	1	i2h	Zm	1 1 0	1	i2l	0	Zn	Zd
			size					U		T		

Encoding

```
UMULLB  <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 0;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = UInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = UInt(operand2[(2 * s + index)*:esize]);
    let res : integer = element1 * element2;
    result[e*:(2*esize)] = res[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.843 UMULLT (vectors)

Unsigned multiply long (top)

This instruction multiplies the odd-numbered unsigned elements of the first source vector by the corresponding elements of the second source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

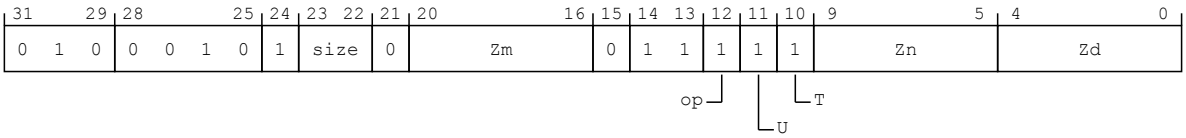

Encoding

```
UMULLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[(2*e + 1)*:(esize DIV 2)]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    let res : integer = element1 * element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.844 UMULLT (indexed)

Unsigned multiply long by indexed element (top)

This instruction multiplies the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and places the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE2 || FEAT_SME)

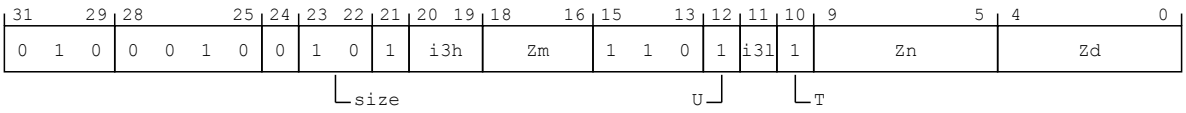

Encoding

```
UMULLT  <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let index : integer = UInt(i3h::i3l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 1;
```

64-bit

(FEAT_SVE2 || FEAT_SME)

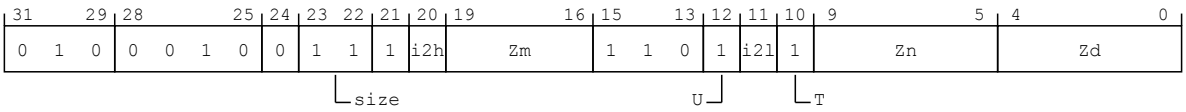

Encoding

```
UMULLT  <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let index : integer = UInt(i2h::i2l);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel : integer = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

For the “32-bit” variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the “64-bit” variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm>

For the “32-bit” variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “64-bit” variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let eltspersegment : integer = 128 DIV (2 * esize);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let s : integer = e - (e MOD eltspersegment);
    let element1 : integer = UInt(operand1[(2 * e + sel)*:esize]);
    let element2 : integer = UInt(operand2[(2 * s + index)*:esize]);
    let res : integer = element1 * element2;
    result[e*:(2*esize)] = res[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.845 UQADD (vectors, predicated)

Unsigned saturating add (predicated)

This instruction adds active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, and destructively places the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

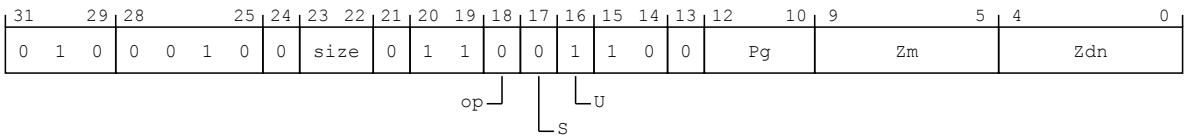

Encoding

```
UQADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = UnsignedSat{esize}(element1 + element2);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.846 UQADD (immediate)

Unsigned saturating add immediate (unpredicated)

This instruction performs an unsigned saturating addition of an unsigned immediate to each element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

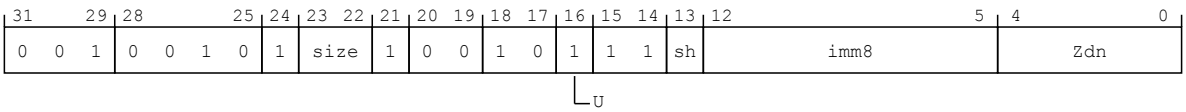

Encoding

```
UQADD  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    (result[e*:esize], -) = SatQ{esize}(element1 + imm, unsigned);

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX must be unpredicated.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.847 UQADD (vectors, unpredicated)

Unsigned saturating add (unpredicated)

This instruction performs an unsigned saturating addition of all elements of the first source vector to the corresponding elements of the second source vector, and places the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 0 0	1 0	1	Zn	Zd
								U		

Encoding

```
UQADD  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    if unsigned then
        let element1 : integer = UInt(op1elt);
        let element2 : integer = UInt(op2elt);
        (result[e*:esize], -) = UnsignedSatQ{esize}(element1 + element2);
    else
        let element1 : integer = SInt(op1elt);
        let element2 : integer = SInt(op2elt);
        (result[e*:esize], -) = SignedSatQ{esize}(element1 + element2);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.848 UQCVTN

Unsigned 32-bit integer saturating extract narrow to interleaved 16-bit integer

This instruction saturates the unsigned integer value in each element of the pair of source vectors to half the original source element width, and places the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

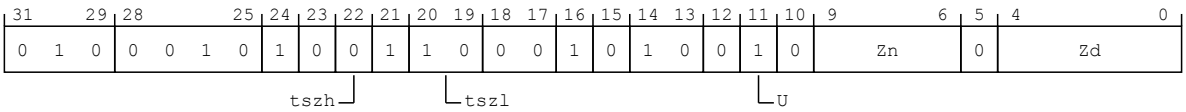

Encoding

```
UQCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 1 do
        let operand : bits(VL) = Z{}(n+i);
        let element : integer = UInt(operand[e*:(2 * esize)]);
        result[(2*e + i)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.849 UQDECB

Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count

This instruction determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

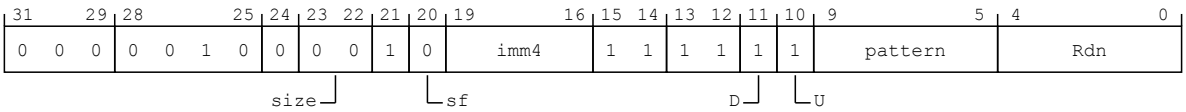

Encoding

```
UQDECB  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

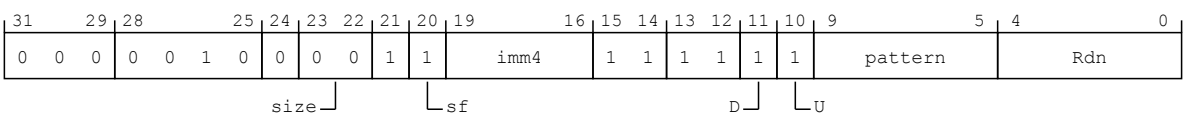

Encoding

```
UQDECB  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.850 UQDECD (scalar)

Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

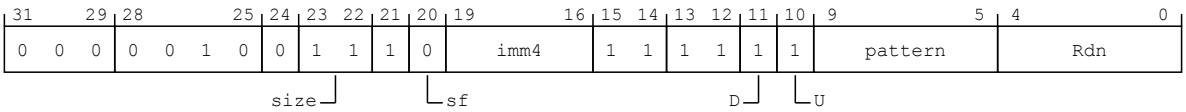

Encoding

```
UQDECD  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

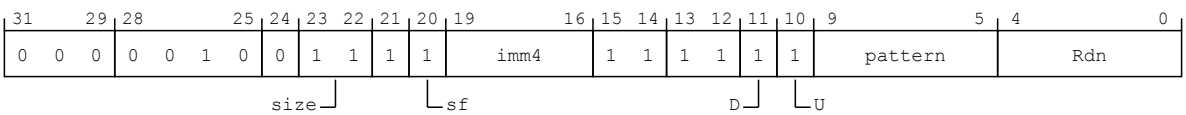

Encoding

```
UQDECD  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.851 UQDECD (vector)

Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit
unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

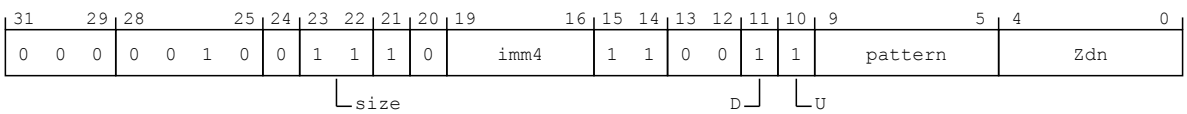

Encoding

```
UQDECD  <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) - (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) - (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.852 UQDECH (scalar)

Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

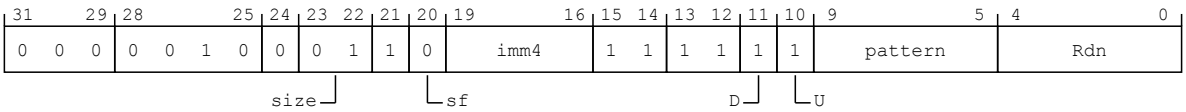

Encoding

```
UQDECH  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

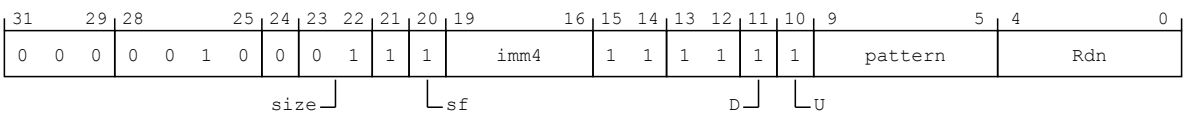

Encoding

```
UQDECH  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.853 UQDECH (vector)

Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit
unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

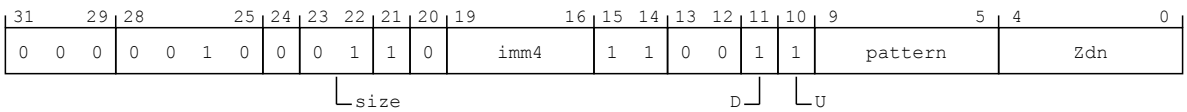

Encoding

```
UQDECH  <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) - (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) - (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.854 UQDECP (scalar)

Unsigned saturating decrement scalar by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

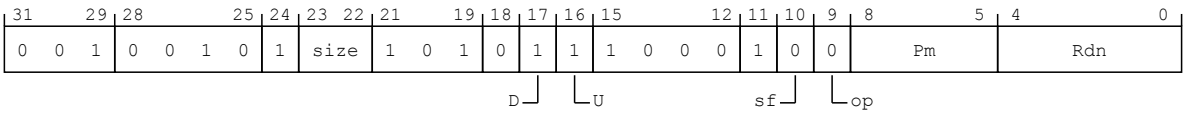

Encoding

```
UQDECP  <Wdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

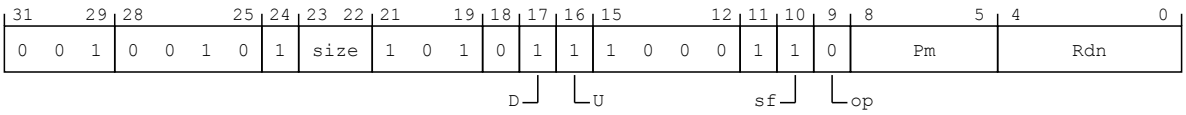

Encoding

```
UQDECP  <Xdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(ssize) = X{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(ssize);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - count);
    X{64}(dn) = ZeroExtend{64}(result);
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - count);
    X{64}(dn) = SignExtend{64}(result);
end;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.855 UQDECP (vector)

Unsigned saturating decrement vector by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

SVE

(FEAT_SVE || FEAT_SME)

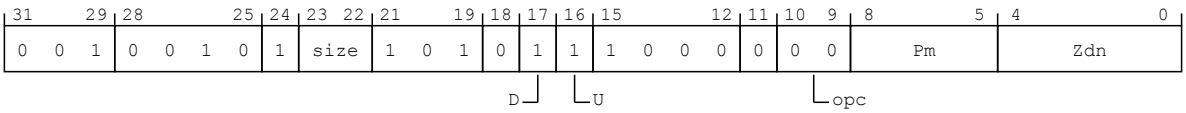

Encoding

```
UQDECP  <Zdn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Zdn);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(VL);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(opelt) - count);
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(opelt) - count);
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.856 UQDECW (scalar)

Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

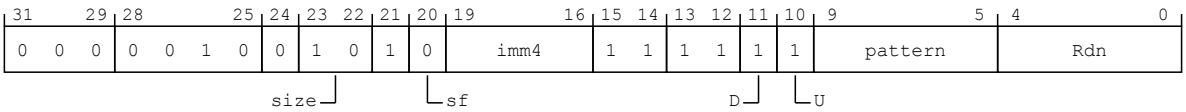

Encoding

```
UQDECW  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

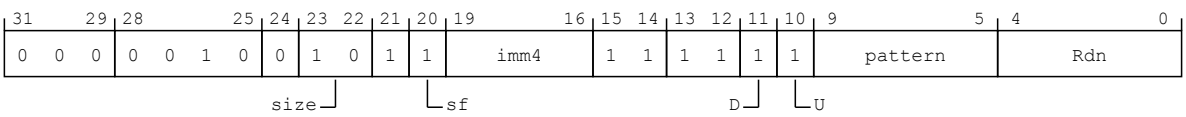

Encoding

```
UQDECW  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) - (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) - (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.857 UQDECW (vector)

Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit
unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

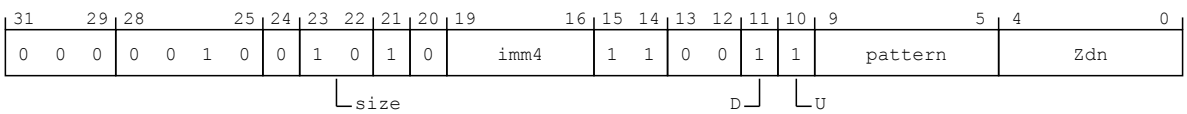

Encoding

```
UQDECW  <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) - (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) - (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.858 UQINCB

Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count

This instruction determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 0	1	0	imm4	1 1	1 1	0	1	pattern	Rdn
			size		sf				D	U		

Encoding

```
UQINCB  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–14	13–12	11	10	9–5	4–0
0 0 0	0 0 1 0	0	0 0	1	1	imm4	1 1	1 1	0	1	pattern	Rdn
			size		sf				D	U		

Encoding

```
UQINCB  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.859 UQINCD (scalar)

Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

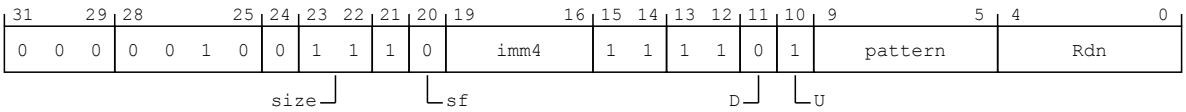

Encoding

```
UQINCD  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

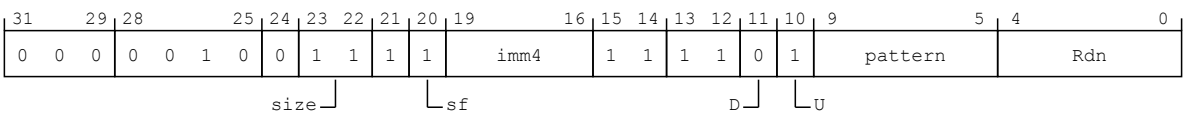

Encoding

```
UQINCD  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.860 UQINCD (vector)

Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count

This instruction determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit
unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

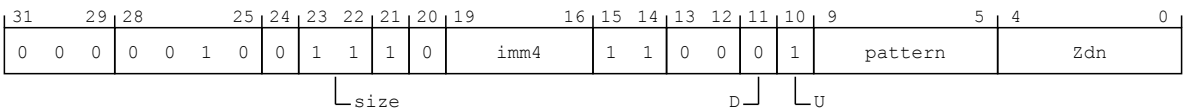

Encoding

```
UQINCD  <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) + (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) + (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.861 UQINCH (scalar)

Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

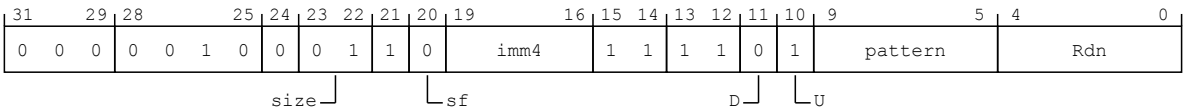

Encoding

```
UQINCH  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

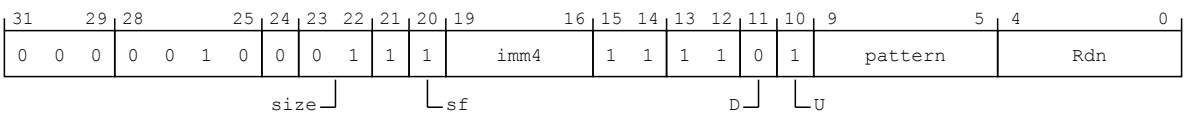

Encoding

```
UQINCH  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.862 UQINCH (vector)

Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count

This instruction determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit
unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

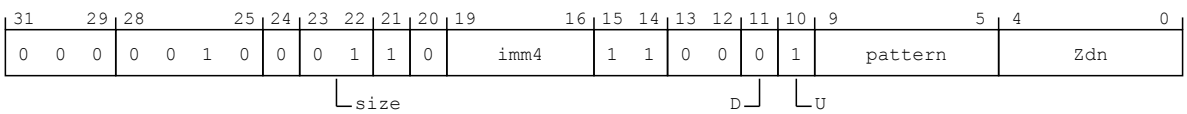

Encoding

```
UQINCH  <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) + (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) + (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.863 UQINCP (scalar)

Unsigned saturating increment scalar by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

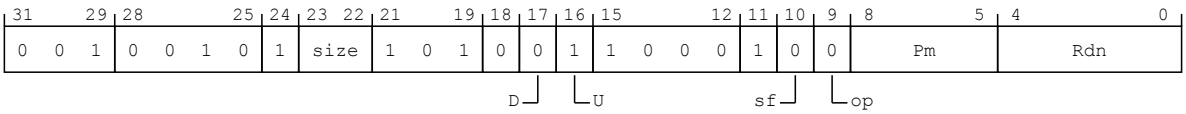

Encoding

```
UQINCP  <Wdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

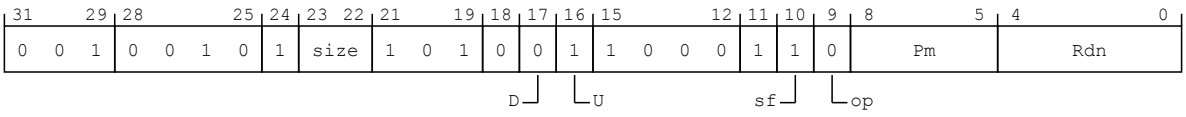

Encoding

```
UQINCP  <Xdn>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Rdn);
let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(ssize) = X{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(ssize);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + count);
    X{64}(dn) = ZeroExtend{64}(result);
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + count);
    X{64}(dn) = SignExtend{64}(result);
end;
```

Operational Information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.864 UQINCP (vector)

Unsigned saturating increment vector by count of true predicate elements

This instruction determines the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

SVE

(FEAT_SVE || FEAT_SME)

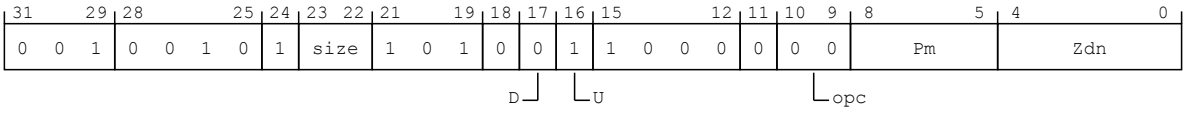

Encoding

```
UQINCP  <Zdn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = UInt(Pm);
let dn : integer = UInt(Zdn);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm>

Is the name of the source scalable predicate register, encoded in the “Pm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(PL) = P{}(m);
var result : bits(VL);
var count : integer = 0;

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(operand2, e, esize) then
        count = count + 1;
    end;
end;

for e = 0 to elements-1 do
    let opelt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(opelt) + count);
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(opelt) + count);
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.865 UQINCW (scalar)

Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SVE || FEAT_SME)

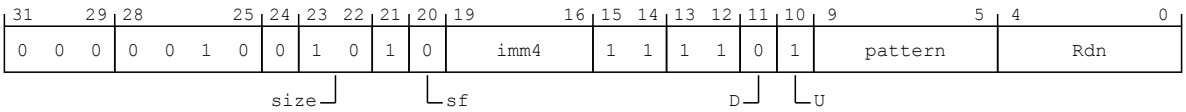

Encoding

```
UQINCW  <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 32;
```

64-bit

(FEAT_SVE || FEAT_SME)

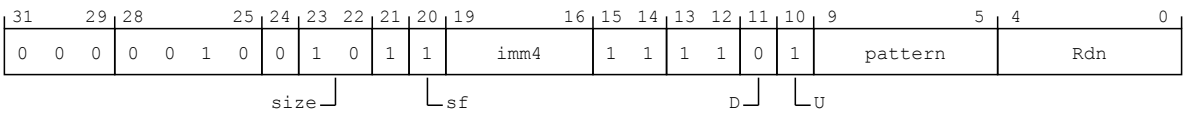

Encoding

```
UQINCW  <Xdn>{, <pattern>{, MUL #<imm>}}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Rdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
let  ssize : integer{} = 64;
```

Assembler Symbols

<Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

<Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

Operation

```
CheckSVEEnabled();
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(ssize) = X{}(dn);
var result : bits(ssize);
if unsigned then
    (result, -) = UnsignedSatQ{ssize}(UInt(operand1) + (count * imm));
else
    (result, -) = SignedSatQ{ssize}(SInt(operand1) + (count * imm));
end;
X{64}(dn) = Extend{64}(result, unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.866 UQINCW (vector)

Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count

This instruction determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit
unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

SVE

(FEAT_SVE || FEAT_SME)

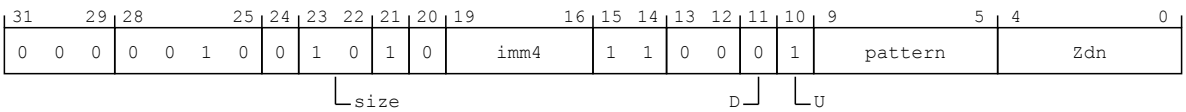

Encoding

```
UQINCW  <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 32;
let dn : integer = UInt(Zdn);
let pat : bits(5) = pattern;
let imm : integer = UInt(imm4) + 1;

let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pattern	<pattern>
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let count : integer = DecodePredCount(pat, esize);
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    if unsigned then
        (result[e*:esize], -) = UnsignedSatQ{esize}(UInt(op1elt) + (count * imm));
    else
        (result[e*:esize], -) = SignedSatQ{esize}(SInt(op1elt) + (count * imm));
    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.867 UQRSHL

Unsigned saturating rounding shift left (predicated)

This instruction shifts active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

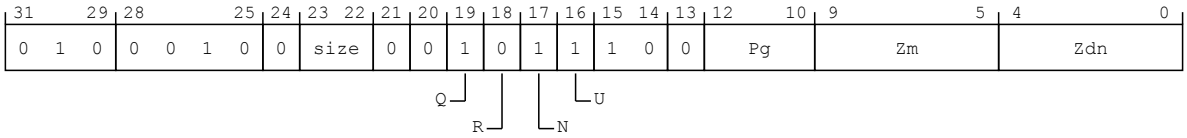

Encoding

```
UQRSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = UnsignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.868 UQRSHLR

Unsigned saturating rounding shift left reversed (predicated)

This instruction shifts active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

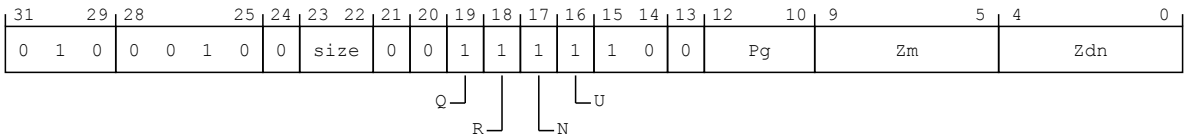

Encoding

```
UQRSHLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand2 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = UnsignedSat{esize}(res);
    else
        result[e*:esize] = operand2[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.869 UQRSHRN

Unsigned saturating rounding shift right narrow by immediate to interleaved integer

This instruction shifts right by an immediate value the unsigned integer value in each element of the two source vectors, and places the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

16-bit

(FEAT_SME2 || FEAT_SVE2p1)

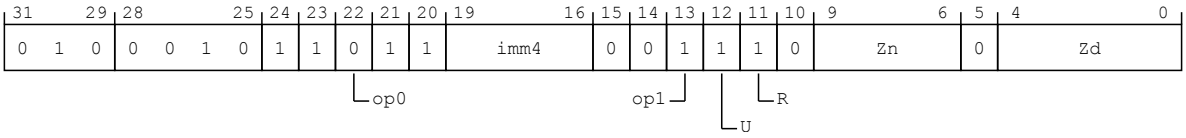

Encoding

```
UQRSHRN  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
let shift : integer = esize - UInt(imm4);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as "Zn" times 2 plus 1.

<const>

Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 1 do
        let operand : bits(VL) = Z{}(n+i);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element : bits(2 * esize) = operand[e*:(2 * esize)];
        let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
        result[(2*e + i)*:esize] = UnsignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.870 UQRSHRNB

Unsigned saturating rounding shift right narrow by immediate (bottom)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

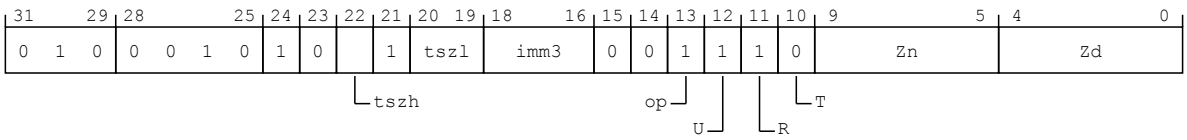

Encoding

```
UQRSHRNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 0)*:esize] = UnsignedSat{esize}(res);
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.871 UQRSHRNT

Unsigned saturating rounding shift right narrow by immediate (top)

This instruction shifts each unsigned integer value in the source vector elements by an immediate value, and places the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

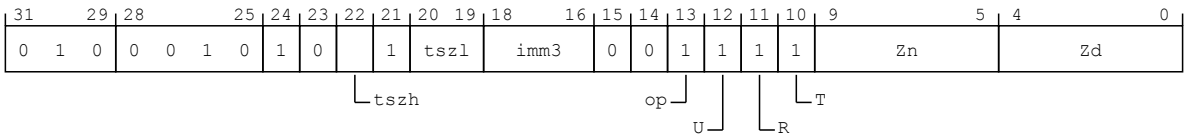

Encoding

```
UQRSHRNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
    result[(2*e + 1)*:esize] = UnsignedSat{esize}(res);

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.872 UQSHL (immediate)

Unsigned saturating shift left by immediate

This instruction shifts left by immediate each active unsigned element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

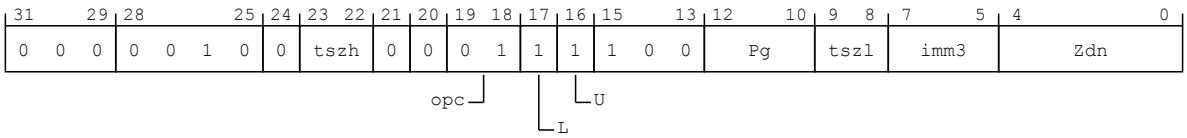

Encoding

```
UQSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = element1 << shift;
        result[e*:esize] = UnsignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.873 UQSHL (vectors)

Unsigned saturating shift left (predicated)

This instruction shifts active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

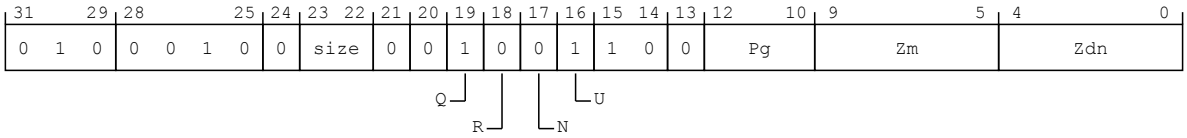

Encoding

```
UQSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = element >> shift;
        end;
        result[e*:esize] = UnsignedSat{esize}(res);
    else
        result[e*:esize] = operand1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.874 UQSHLR

Unsigned saturating shift left reversed (predicated)

This instruction shifts active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively places the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

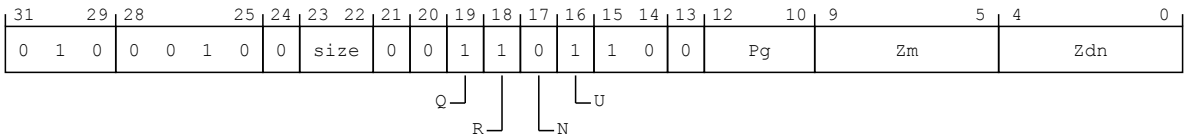

Encoding

```
UQSHLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand2 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = element >> shift;
        end;
        result[e*:esize] = UnsignedSat{esize}(res);
    else
        result[e*:esize] = operand2[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is CONSTRAINED UNPREDICTABLE:

- The MOVPRFX can be predicated or unpredicated.
- A predicated MOVPRFX must use the same governing predicate register as this instruction.
- A predicated MOVPRFX must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The MOVPRFX must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.875 UQSHRNB

Unsigned saturating shift right narrow by immediate (bottom)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

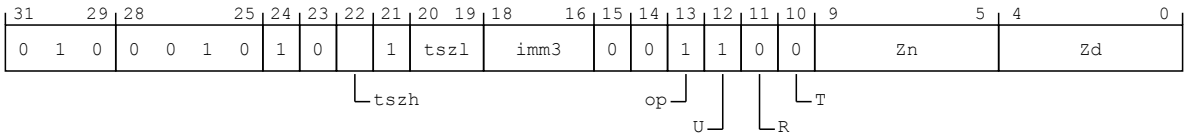

Encoding

```
UQSHRNB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = UInt(element) >> shift;
    result[(2*e + 0)*:esize] = UnsignedSat{esize}(res);
    result[(2*e + 1)*:esize] = Zeros{esize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.876 UQSHRNT

Unsigned saturating shift right narrow by immediate (top)

This instruction shifts each unsigned integer value in the source vector elements right by an immediate value, and places the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

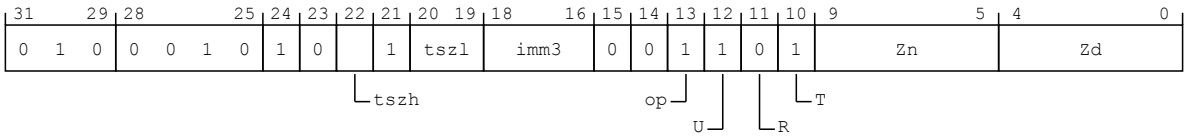

Encoding

```
UQSHRNT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn>

Is the name of the source scalable vector register, encoded in the "Zn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
for e = 0 to elements-1 do
    let element : bits(2*esize) = operand[e*:(2*esize)];
    let res : integer = UInt(element) >> shift;
    result[(2*e + 1)*:esize] = UnsignedSat{esize}(res);

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.877 UQSUB (vectors, predicated)

Unsigned saturating subtract (predicated)

This instruction subtracts active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector and destructively places the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

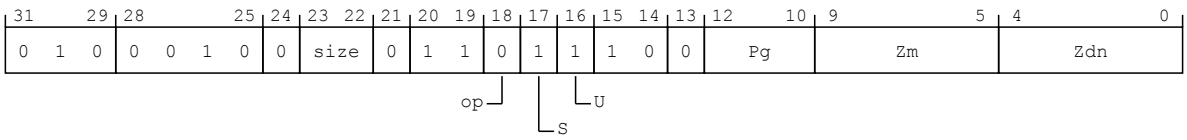

Encoding

```
UQSUB  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = UnsignedSat{esize}(element1 - element2);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.878 UQSUB (immediate)

Unsigned saturating subtract immediate (unpredicated)

This instruction performs an unsigned saturating subtraction of an unsigned immediate from each element of the source vector, and destructively places the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "`#<imm>, LSL #8`". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "`#0, LSL #8`".

SVE

(FEAT_SVE || FEAT_SME)

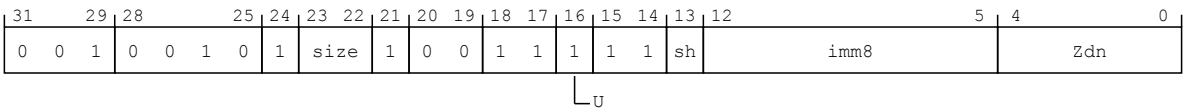

Encoding

```
UQSUB  <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size::sh == '001' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn);
var imm : integer = UInt(imm8);
if sh == '1' then imm = imm << 8; end;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

Is an unsigned immediate in the range 0 to 255, encoded in the “imm8” field.

<shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    (result[e*:esize], -) = SatQ{esize}(element1 - imm, unsigned);

end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.879 UQSUB (vectors, unpredicated)

Unsigned saturating subtract (unpredicated)

This instruction performs an unsigned saturating subtraction of all elements of the second source vector from the corresponding elements of the first source vector, and places the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.

SVE

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	0	size	1	Zm	0 0 0	1 1	1	Zn	Zd
								U		

Encoding

```
UQSUB  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize) = operand1[e*:esize];
    let op2elt : bits(esize) = operand2[e*:esize];
    if unsigned then
        let element1 : integer = UInt(op1elt);
        let element2 : integer = UInt(op2elt);
        (result[e*:esize], -) = UnsignedSatQ{esize}(element1 - element2);
    else
        let element1 : integer = SInt(op1elt);
        let element2 : integer = SInt(op2elt);
        (result[e*:esize], -) = SignedSatQ{esize}(element1 - element2);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.880 UQSUBR

Unsigned saturating subtract reversed (predicated)

This instruction subtracts active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector and destructively places the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

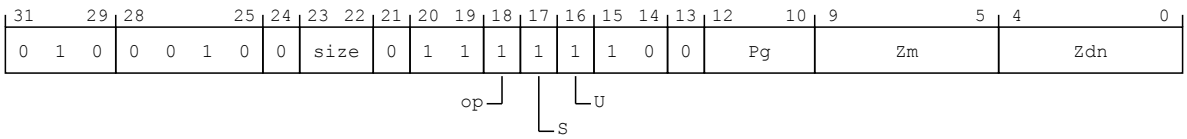

Encoding

```
UQSUBR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = UnsignedSat{esize}(element2 - element1);
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.881 UQXTNB

Unsigned saturating extract narrow (bottom)

This instruction saturates the unsigned integer value in each source element to half the original source element width, and places the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

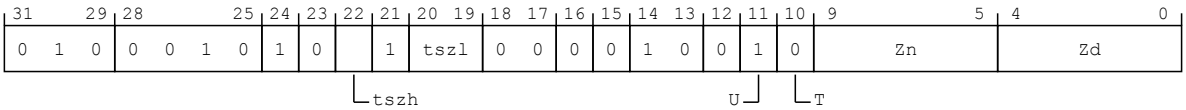

Encoding

```
UQXTNB  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if !(tsize IN {'001', '010', '100'}) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
0	11	RESERVED
1	00	S
1	01	RESERVED
1	1x	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
0	11	RESERVED
1	00	D
1	01	RESERVED
1	1x	RESERVED

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let res : bits(halfesize) = UnsignedSat{halfesize}(element1);
    result[(2*e + 0)*:halfesize] = res;
    result[(2*e + 1)*:halfesize] = Zeros{halfesize};

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.882 UQXTNT

Unsigned saturating extract narrow (top)

This instruction saturates the unsigned integer value in each source element to half the original source element width, and places the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

SVE2

(FEAT_SVE2 || FEAT_SME)

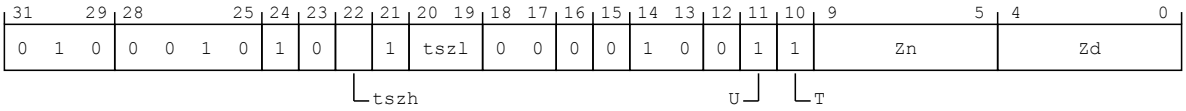

Encoding

```
UQXTNT  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if !(tsize IN {'001', '010', '100'}) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
0	11	RESERVED
1	00	S
1	01	RESERVED
1	1x	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
0	11	RESERVED
1	00	D
1	01	RESERVED
1	1x	RESERVED

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
var result : bits(VL) = Z{}(d);
let halfesize : integer{} = esize DIV 2;

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let res : bits(halfesize) = UnsignedSat{halfesize}(element1);
    result[(2*e + 1)*:halfesize] = res;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.883 URECPE

Unsigned reciprocal estimate (predicated)

This instruction finds the approximate reciprocal of each active unsigned element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2 || FEAT_SME)

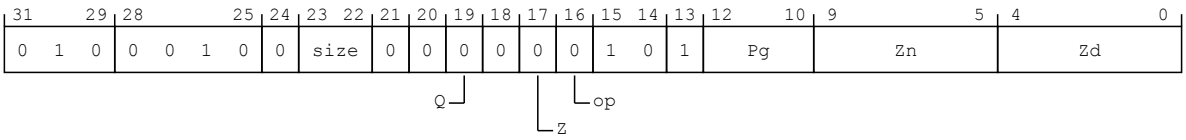

Encoding

```
URECPE  <Zd>.S, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

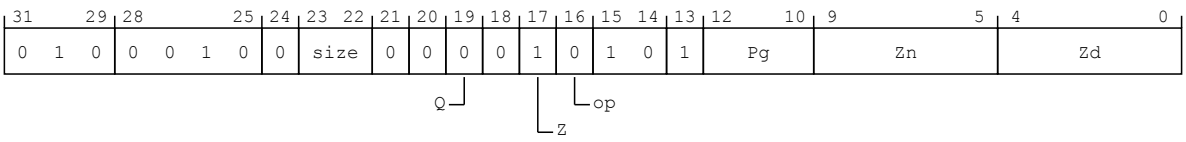

Encoding

```
URECPE  <Zd>.S, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = UnsignedRecipEstimate{esize}(element);

    end;
end;
Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.884 URHADD

Unsigned rounding halving add

This instruction adds active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shifts right one bit, and destructively places the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

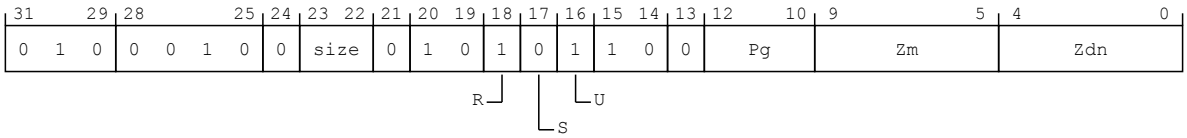

Encoding

```
URHADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 + element2 + 1) >> 1;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.885 URSHL

Unsigned rounding shift left (predicated)

This instruction shifts active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

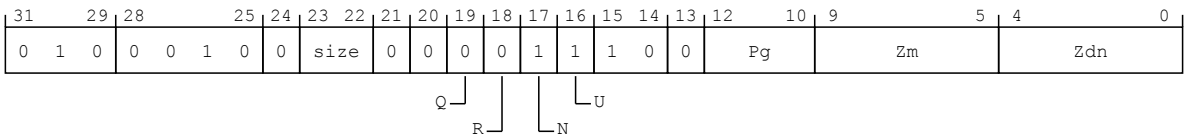

Encoding

```
URSHL  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.886 URSHLR

Unsigned rounding shift left reversed (predicated)

This instruction shifts active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively places the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

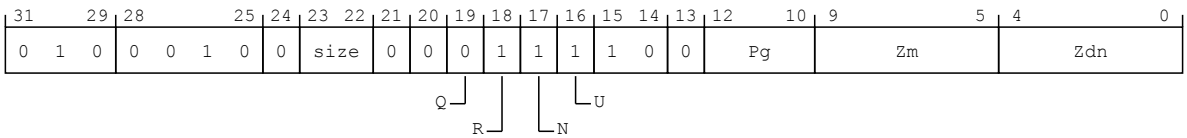

Encoding

```
URSHLR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
let operand2 : bits(VL) = Z{}(dn);
var result : bits(VL);

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : integer = UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand2[e*:esize];

    end;
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.887 URSHR

Unsigned rounding shift right by immediate

This instruction shifts right by immediate each active unsigned element of the source vector, and destructively places the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

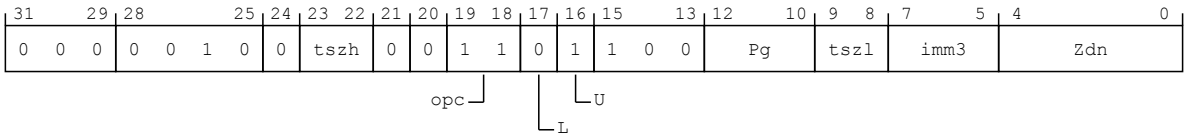

Encoding

```
URSHR  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zdn>

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let mask : bits(PL) = P{}(g);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    if ActivePredicateElement{PL}(mask, e, esize) then
        let res : integer = (element1 + (1 << (shift - 1))) >> shift;
        result[e*:esize] = res[esize-1:0];
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.888 URSQRTE

Unsigned reciprocal square root estimate (predicated)

This instruction finds the approximate reciprocal square root of each active unsigned element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 2 classes: Merging and Zeroing

Merging

(FEAT_SVE2 || FEAT_SME)

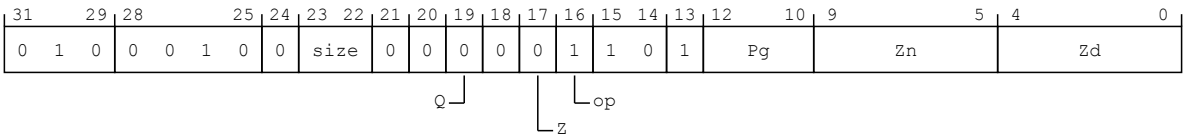

Encoding

```
URSQRTE  <Zd>.S, <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = TRUE;
```

Zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

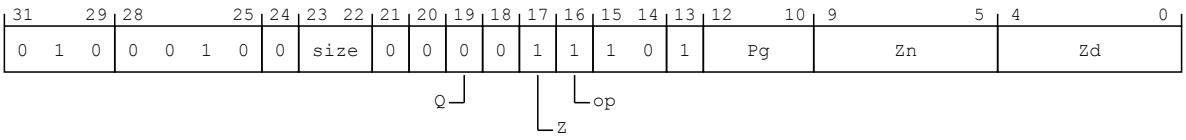

Encoding

```
URSQRTE  <Zd>.S, <Pg>/Z, <Zn>.S
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '10' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};

for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = UnsignedRSqrtEstimate{esize}(element);

    end;
end;
Z{VL}(d) = result;
```

Operational Information

For the Merging variant:

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.889 URSRA

Unsigned rounding shift right and accumulate (immediate)

This instruction shifts right by immediate each unsigned element of the source vector, inserting zeroes, and destructively adds the rounded intermediate result to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–14	13–12	11	10	9–5	4–0
0 1 0	0 0 1 0	1	tszh	0	tszl	imm3	1 1	1 0	1 (R)	1 (U)	Zn	Zda

Encoding

```
URSRA  <Zda>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let da : integer = UInt(Zda);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : integer = (UInt(operand1[e*:esize]) + (1 << (shift - 1))) >> shift;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.890 USDOT (vectors)

Unsigned by signed 8-bit integer dot product to 32-bit integer

This instruction calculates the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_I8MM)

31–29	28–25	24	23–22	21	20–16	15	14–10	9–5	4–0
0 1 0	0 0 1 0	0	1 0 (size)	0	Zm	0	1 1 1 1 0	Zn	Zda

Encoding

```
USDOT  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_I8MM)) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.891 USDOT (indexed)

Unsigned by signed 8-bit integer dot product by indexed element to 32-bit integer

This instruction calculates the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

((FEAT_SVE || FEAT_SME) && FEAT_I8MM)

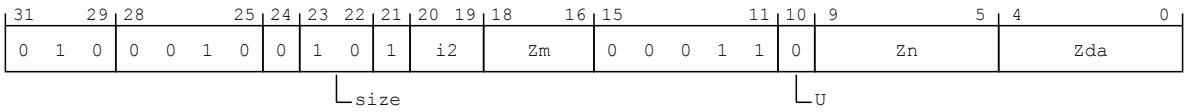

Encoding

```
USDOT  <Zda>.S, <Zn>.B, <Zm>.B[<imm>]
```

Decode for this encoding

```
if ((!IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME)) ||
    !IsFeatureImplemented(FEAT_I8MM)) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let index : integer = UInt(i2);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let eltspersegment : integer = 128 DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let segmentbase : integer = e - (e MOD eltspersegment);
    let s : integer = segmentbase + index;
    var res : bits(esize) = operand3[e*:esize];
    for i = 0 to 3 do
        let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
        let element2 : integer = SInt(operand2[(4 * s + i)*:(esize DIV 4)]);
        res = res + element1 * element2;
    end;
    result[e*:esize] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.892 USHLLB

Unsigned shift left long by immediate (bottom)

This instruction shifts left by immediate each even-numbered unsigned element of the source vector, and places the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

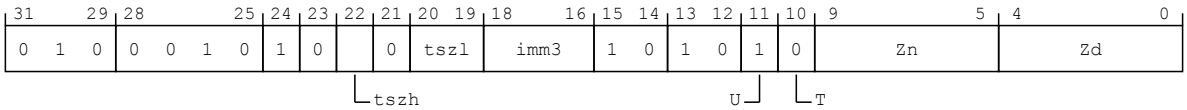

Encoding

```
USHLLB  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[(2*e + 0)*:esize];
    let shifted_value : integer = UInt(element) << shift;
    result[e*:(2*esize)] = shifted_value[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.893 USHLLT

Unsigned shift left long by immediate (top)

This instruction shifts left by immediate each odd-numbered unsigned element of the source vector, and places the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

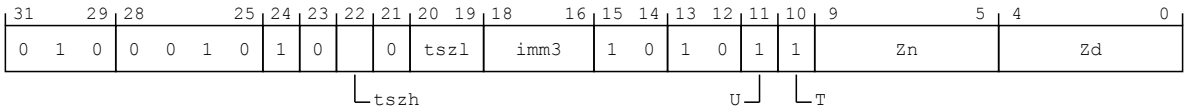

Encoding

```
USHLLT  <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(3) = tszh::tszl;
if tsize == '000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let shift : integer = UInt(tsize::imm3) - esize;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const>

Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in “tszh:tszl:imm3”.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(esize) = operand[(2*e + 1)*:esize];
    let shifted_value : integer = UInt(element) << shift;
    result[e*:(2*esize)] = shifted_value[2*esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.894 USMMLA

Unsigned by signed 8-bit integer matrix multiply-accumulate to 32-bit integer

This instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE && FEAT_I8MM)

31–29	28–25	24	23–22	21	20–16	15–14	13–10	9–5	4–0
0 1 0	0 0 1 0	1	1 0	0	Zm	1 0	0 1 1 0	Zn	Zda
			uns						

Encoding

```
USMMLA  <Zda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) || !IsFeatureImplemented(FEAT_I8MM) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(Zda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(VL) = Z{}(da);
var result : bits(VL) = Zeros{};
var op1, op2 : bits(128);
var res, addend : bits(128);

for s = 0 to segments-1 do
    op1    = operand1[s*:(128)];
    op2    = operand2[s*:(128)];
    addend = operand3[s*:(128)];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    result[s*:(128)] = res;
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.895 USQADD

Unsigned saturating signed add

This instruction adds active signed elements of the source vector to the corresponding unsigned elements of the addend vector, and destructively places the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SVE2 || FEAT_SME)

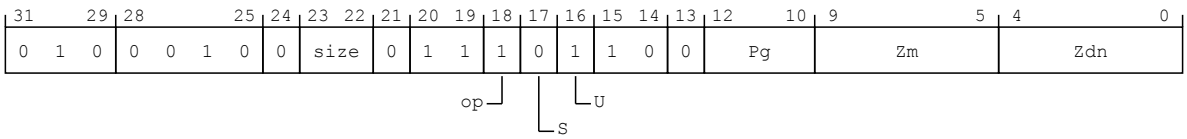

Encoding

```
USQADD  <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let g : integer = UInt(Pg);
let dn : integer = UInt(Zdn);
let m : integer = UInt(Zm);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(m) else Zeros{VL};
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = UnsignedSat{esize}(UInt(element1) + SInt(element2));
    else
        result[e*:esize] = operand1[e*:esize];
    end;
end;

Z{VL}(dn) = result;
```

Operational Information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of this instruction.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.896 USRA

Unsigned shift right and accumulate (immediate)

This instruction shifts right by immediate each unsigned element of the source vector, inserting zeroes, and destructively adds the truncated intermediate result to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18–16	15–14	13–12	11	10	9–5	4–0
0 1 0	0 0 1 0	1	tszh	0	tszl	imm3	1 1	1 0	0 (R)	1 (U)	Zn	Zda

Encoding

```
USRA  <Zda>.<T>, <Zn>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn);
let da : integer = UInt(Zda);
let shift : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zda>

Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(da);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : integer = UInt(operand1[e*:esize]) >> shift;
    result[e*:esize] = operand2[e*:esize] + element[esize-1:0];
end;

Z{VL}(da) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.897 USUBLB

Unsigned subtract long (bottom)

This instruction subtracts the even-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

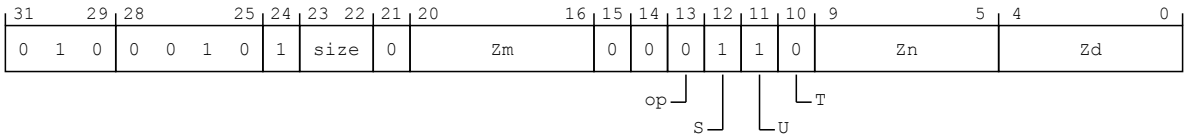

Encoding

```
USUBLB  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 0;
let sel2 : integer = 0;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 - element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.898 USUBLT

Unsigned subtract long (top)

This instruction subtracts the odd-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and places the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

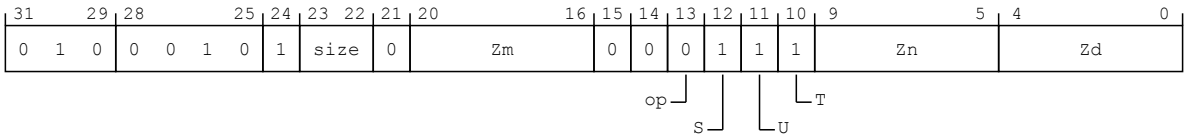

Encoding

```
USUBLT  <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let sel1 : integer = 1;
let sel2 : integer = 1;
let unsigned : boolean = TRUE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let op1elt : bits(esize DIV 2) = operand1[(2*e + sel1)*:(esize DIV 2)];
    let op2elt : bits(esize DIV 2) = operand2[(2*e + sel2)*:(esize DIV 2)];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let res : integer = element1 - element2;
    result[e*:esize] = res[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.899 USUBWB

Unsigned subtract wide (bottom)

This instruction subtracts the even-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

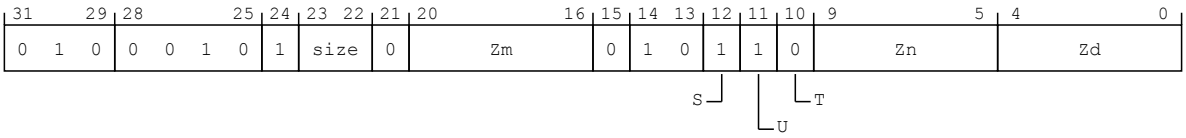

Encoding

```
USUBWB  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[(2*e + 0)*:(esize DIV 2)]);
    result[e*:esize] = (element1 - element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.900 USUBWT

Unsigned subtract wide (top)

This instruction subtracts the odd-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and places the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

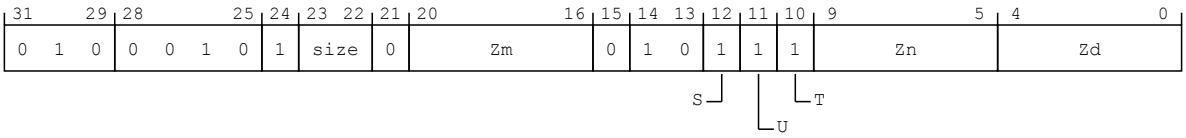

Encoding

```
USUBWT  <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : integer = UInt(operand1[e*:esize]);
    let element2 : integer = UInt(operand2[(2*e + 1)*:(esize DIV 2)]);
    result[e*:esize] = (element1 - element2)[esize-1:0];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.901 UUNPKHI, UUNPKLO

Unsigned unpack and extend half of vector

This instruction unpacks and zero-extends elements from the lowest or highest half of the source vector, and writes the zero-extended values into double-width elements of the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

High half

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	1 0	0	1	1	0 0 1 1 1 0	Zn	Zd

U H

Encoding

```
UUNPKHI  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let hi : boolean = TRUE;
```

Low half

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–19	18	17	16	15–10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	1 0	0	1	0	0 0 1 1 1 0	Zn	Zd

U H

Encoding

```
UUNPKLO  <Zd>.<T>, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let hi : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let hsize : integer{} = esize DIV 2;
let offset : integer = if hi then elements else 0;
let operand : bits(VL) = Z{}(n);
var result : bits(VL);

for e = 0 to elements-1 do
    let element : bits(hsize) = operand[(e + offset)*:hsize];
    result[e*:esize] = Extend{esize}(element, unsigned);
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.902 UXTB, UXTH, UXTW

Unsigned byte / halfword / word extend (predicated)

This instruction zero-extends the least-significant sub-element of each active element of the source vector, and places the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected.

It has encodings from 6 classes: Byte, merging, Byte, zeroing, Halfword, merging, Halfword, zeroing, Word, merging, and Word, zeroing

Byte, merging

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19	18–17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	1	0	0 0	1	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
UXTB  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let merging : boolean = TRUE;
```

Byte, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31–29	28–25	24	23–22	21	20	19	18–17	16	15–13	12–10	9–5	4–0
0 0 0	0 0 1 0	0	size	0	0	0	0 0	1	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
UXTB  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 8;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let merging : boolean = FALSE;
```

Halfword, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1	0	0 1	1	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
UXTH  <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let merging : boolean = TRUE;
```

Halfword, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0	0	0 1	1	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
UXTH  <Zd>.<T>, <Pg>/Z, <Zn>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size IN {'0x'} then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 16;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let merging : boolean = FALSE;
```

Word, merging

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	1	0	1 0	1	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
UXTW  <Zd>.D, <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let merging : boolean = TRUE;
```

Word, zeroing

(FEAT_SVE2p2 || FEAT_SME2p2)

31 29	28 25	24	23 22	21	20	19	18 17	16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	size	0	0	0	1 0	1	1 0 1	Pg	Zn	Zd
					M			U				

Encoding

```
UXTW  <Zd>.D, <Pg>/Z, <Zn>.D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p2) && !IsFeatureImplemented(FEAT_SME2p2) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let s_esize : integer{} = 32;
let g : integer = UInt(Pg);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let unsigned : boolean = TRUE;
let merging : boolean = FALSE;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

For the “Byte, merging” and “Byte, zeroing” variants: is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

For the “Halfword, merging” and “Halfword, zeroing” variants: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = if AnyActiveElement{PL}(mask, esize) then Z{VL}(n) else Zeros{VL};
var result : bits(VL) = if merging then Z{VL}(d) else Zeros{VL};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    if ActivePredicateElement{PL}(mask, e, esize) then
        let element : bits(esize) = operand[e*:esize];
        result[e*:esize] = Extend{esize}(element[s_esize-1:0], unsigned);
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

The merging variant of this instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and the merging variant of this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` can be predicated or unpredicated.
- A predicated `MOVPRFX` must use the same governing predicate register as the merging variant this instruction.
- A predicated `MOVPRFX` must use the larger of the destination element size and first source element size in the preferred disassembly of the merging variant of this instruction.
- The `MOVPRFX` must specify the same destination register as the merging variant of this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of the merging variant of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.903 UZP1, UZP2 (predicates)

Concatenate even or odd elements from two predicates

This instruction concatenates adjacent even or odd-numbered elements from the first and second source predicates, and places the values in the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: Even and Odd

Even

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–11	10	9	8–5	4	3–0
0 0 0	0 0 1 0	1	size	1	0	Pm	0 1 0	0 1	0	0	Pn	0	Pd
								opc	H				

Encoding

```
UZP1  <Pd>.<T>, <Pn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let part : integer = 0;
```

Odd

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–11	10	9	8–5	4	3–0
0 0 0	0 0 1 0	1	size	1	0	Pm	0 1 0	0 1	1	0	Pn	0	Pd
								opc	H				

Encoding

```
UZP2  <Pd>.<T>, <Pn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let part : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let pairs : integer = VL DIV (esize * 2);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);

for p = 0 to pairs - 1 do
    result[p*:(esize DIV 8)] = operand1[(2*p+part)*:(esize DIV 8)];
end;

for p = 0 to pairs - 1 do
    result[(pairs+p)*:(esize DIV 8)] = operand2[(2*p+part)*:(esize DIV 8)];
end;

P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.904 UZP1, UZP2 (vectors)

Concatenate even or odd elements from two vectors

This instruction concatenates adjacent even or odd-numbered elements from the first and second source vectors, and places the values in the destination vector. This instruction is unpredicated.

Note: UZP1 is equivalent to truncating and packing each element from two source vectors into a single destination vector with elements of half the size.

The 128-bit element variant requires that the Effective SVE vector length is at least 256 bits. ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: Even, Even (quadwords), Odd, and Odd (quadwords)

Even

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 1 1	0 1	0	Zn	Zd
								H		

Encoding

```
UZP1  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```

Even (quadwords)

(FEAT_F64MM)

31–29	28–25	24–23	22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1 1	0	1	Zm	0 0 0	0 1	0	Zn	Zd
							opc	H		

Encoding

```
UZP1  <Zd>.Q, <Zn>.Q, <Zm>.Q
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```

Odd

(FEAT_SVE || FEAT_SME)

31 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	1	size	1	Zm	0 1 1	0 1	1	Zn	Zd
								H		

Encoding

```
UZP2  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```

Odd (quadwords)

(FEAT_F64MM)

31 29	28 25	24 23	22	21	20 16	15 13	12 11	10	9 5	4 0
0 0 0	0 0 1 0	1 1	0	1	Zm	0 0 0	0 1	1	Zn	Zd
							opc	H		

Encoding

```
UZP2  <Zd>.Q, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if esize < 128 then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL < esize * 2 then EndOfDecode(Decode_UNDEF); end;
let pairs : integer = VL DIV (esize * 2);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Zeros{};

for p = 0 to pairs - 1 do
    result[p*:esize] = operand1[(2*p+part)*:esize];
end;

for p = 0 to pairs - 1 do
    result[(pairs+p)*:esize] = operand2[(2*p+part)*:esize];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.905 UZPQ1

Concatenate even elements within each pair of quadword vector segments

This instruction concatenates adjacent even-numbered elements from the corresponding 128-bit vector segments of the first and second source vectors, and places them in the corresponding elements of the destination vector segment. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

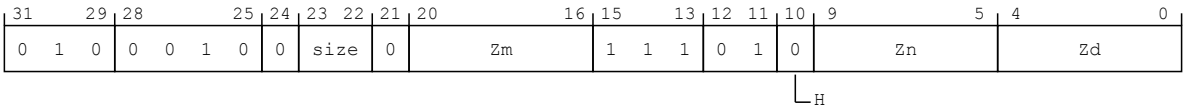

Encoding

```
UZPQ1  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let pairs : integer = elements DIV 2;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let soff : integer = s * elements;
    for p = 0 to pairs-1 do
        result[(soff + p)*:esize] = operand1[(soff + 2 * p + part)*:esize];
    end;

    for p = 0 to pairs-1 do
        result[(soff + pairs + p)*:esize] = operand2[(soff + 2 * p + part)*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.906 UZPQ2

Concatenate odd elements within each pair of quadword vector segments

This instruction concatenates adjacent odd-numbered elements from the corresponding 128-bit vector segments of the first and second source vectors, and places them in the corresponding elements of the destination vector segment. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

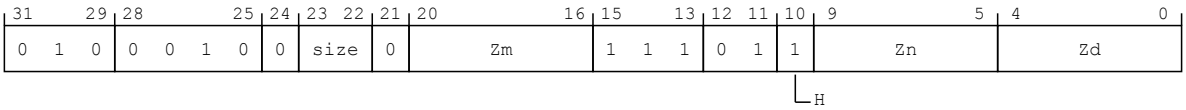

Encoding

```
UZPQ2  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let pairs : integer = elements DIV 2;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let soff : integer = s * elements;
    for p = 0 to pairs-1 do
        result[(soff + p)*:esize] = operand1[(soff + 2 * p + part)*:esize];
    end;

    for p = 0 to pairs-1 do
        result[(soff + pairs + p)*:esize] = operand2[(soff + 2 * p + part)*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.907 WHILEGE (predicate)

While decrementing signed scalar greater than or equal to scalar

This instruction generates a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum signed integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–14	13	12	11	10	9–5	4	3–0
0 0 1	0 0 1 0	1	size	1	Rm	0 0	0	sf	0	0	Rn	0	Pd
									U	lt		eq	

Encoding

```
WHILEGE  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_GE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.908 WHILEGE (predicate as counter)

While decrementing signed scalar greater than or equal to scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum signed integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

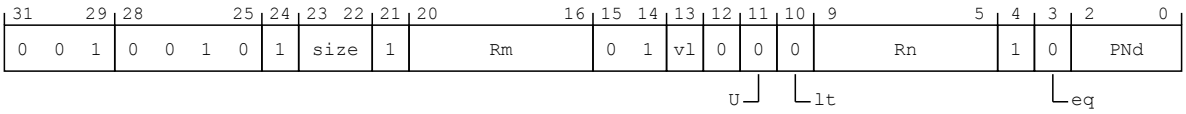

Encoding

```
WHILEGE  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = FALSE;
let invert : boolean = TRUE;
let op : CmpOp = Cmp_GE;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    var cond : boolean;
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 - 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.909 WHILEGE (predicate pair)

While decrementing signed scalar greater than or equal to scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum signed integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

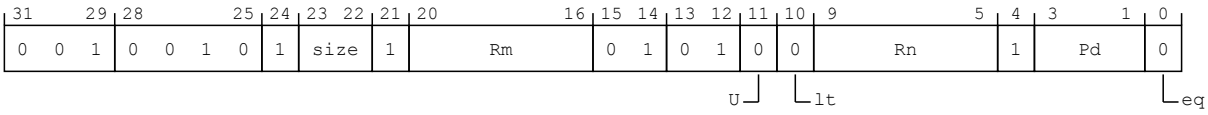

Encoding

```
WHILEGE  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_GE;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = (elements*2)-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.910 WHILEGT (predicate)

While decrementing signed scalar greater than scalar

This instruction generates a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

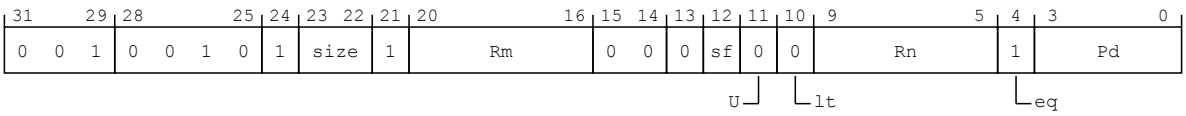

Encoding

```
WHILEGT  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_GT;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.911 WHILEGT (predicate as counter)

While decrementing signed scalar greater than scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

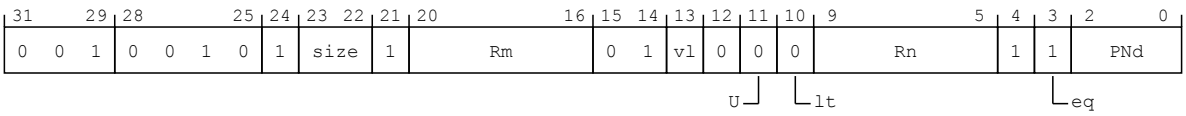

Encoding

```
WHILEGT  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = FALSE;
let invert : boolean = TRUE;
let op : CmpOp = Cmp_GT;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    var cond : boolean;
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 - 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.912 WHILEGT (predicate pair)

While decrementing signed scalar greater than scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

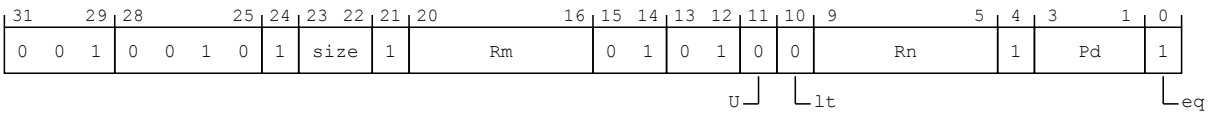

Encoding

```
WHILEGT  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_GT;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = (elements*2)-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.913 WHILEHI (predicate)

While decrementing unsigned scalar higher than scalar

This instruction generates a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

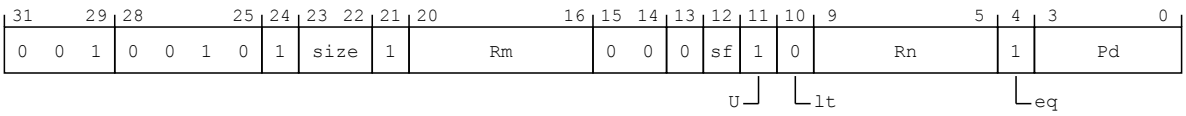

Encoding

```
WHILEHI  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_GT;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.914 WHILEHI (predicate as counter)

While decrementing unsigned scalar higher than scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

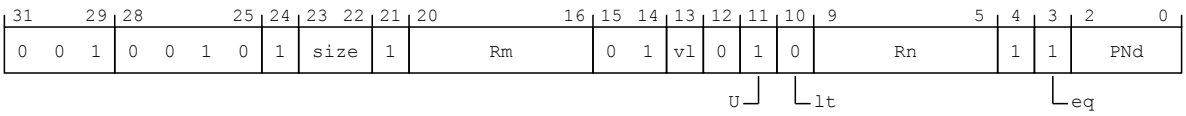

Encoding

```
WHILEHI  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = TRUE;
let invert : boolean = TRUE;
let op : CmpOp = Cmp_GT;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    var cond : boolean;
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 - 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.915 WHILEHI (predicate pair)

While decrementing unsigned scalar higher than scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

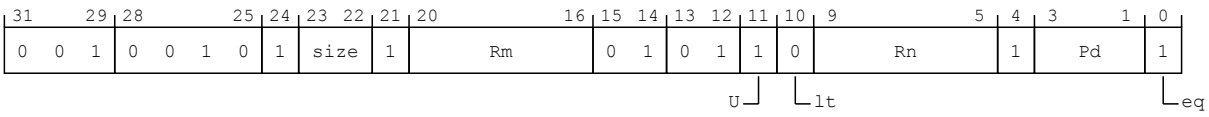

Encoding

```
WHILEHI  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_GT;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = (elements*2)-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.916 WHILEHS (predicate)

While decrementing unsigned scalar higher than or same as scalar

This instruction generates a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher than or the same as the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum unsigned integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

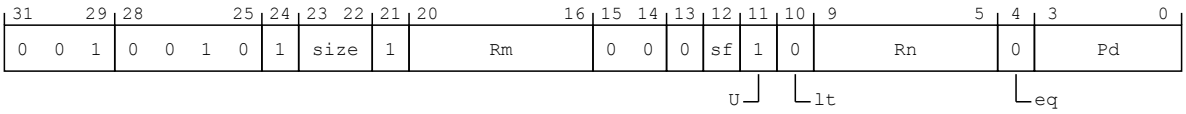

Encoding

```
WHILEHS  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_GE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.917 WHILEHS (predicate as counter)

While decrementing unsigned scalar higher than or same as scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher than or the same as the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum unsigned integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

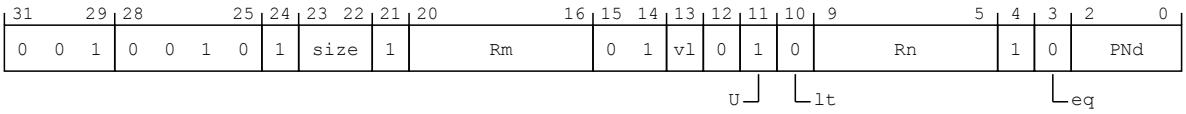

Encoding

```
WHILEHS  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = TRUE;
let invert : boolean = TRUE;
let op : CmpOp = Cmp_GE;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = elements-1 downto 0 do
    var cond : boolean;
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 - 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.918 WHILEHS (predicate pair)

While decrementing unsigned scalar higher than or same as scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher than or the same as the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum unsigned integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

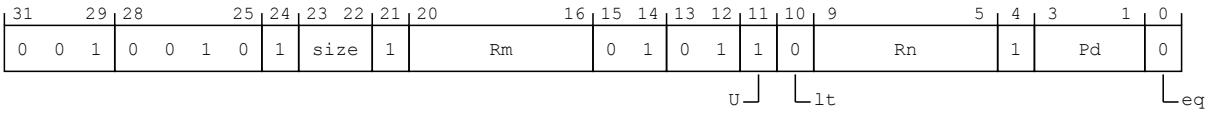

Encoding

```
WHILEHS  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_GE;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = (elements*2)-1 downto 0 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_GT => cond = (element1 >  element2);
        when Cmp_GE => cond = (element1 >= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 - 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.919 WHILELE (predicate)

While incrementing signed scalar less than or equal to scalar

This instruction generates a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum signed integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

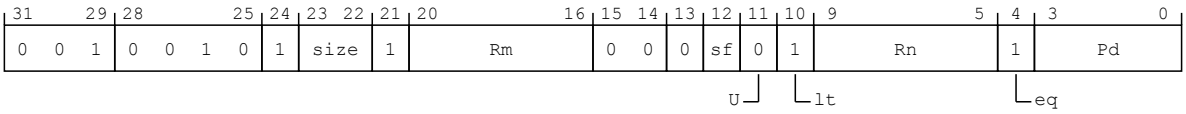

Encoding

```
WHILELE  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_LE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.920 WHILELE (predicate as counter)

While incrementing signed scalar less than or equal to scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum signed integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

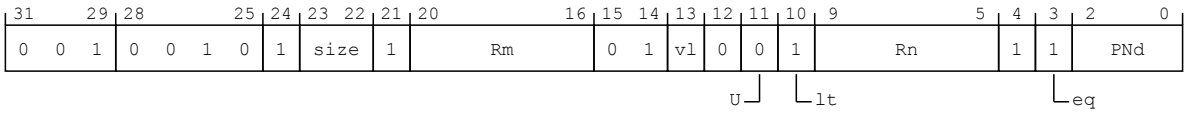

Encoding

```
WHILELE  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = FALSE;
let invert : boolean = FALSE;
let op : CmpOp = Cmp_LE;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let op2val : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let op1val : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (op1val <  op2val);
        when Cmp_LE => cond = (op1val <= op2val);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 + 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.921 WHILELE (predicate pair)

While incrementing signed scalar less than or equal to scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum signed integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

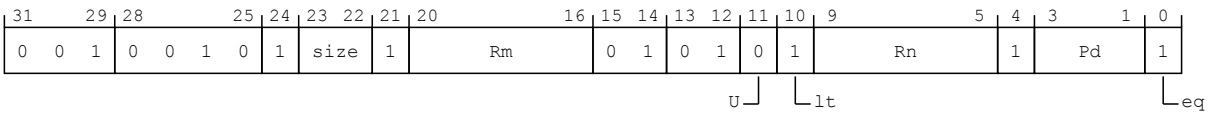

Encoding

```
WHILELE  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_LE;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to (elements*2)-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.922 WHILELO (predicate)

While incrementing unsigned scalar lower than scalar

This instruction generates a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

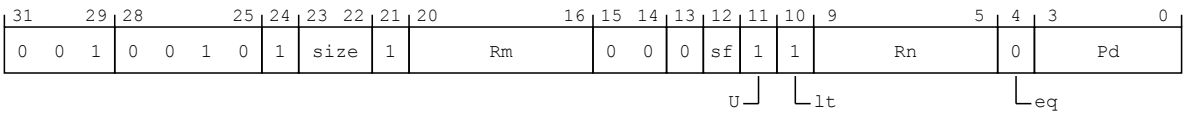

Encoding

```
WHILELO  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_LT;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.923 WHILELO (predicate as counter)

While incrementing unsigned scalar lower than scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

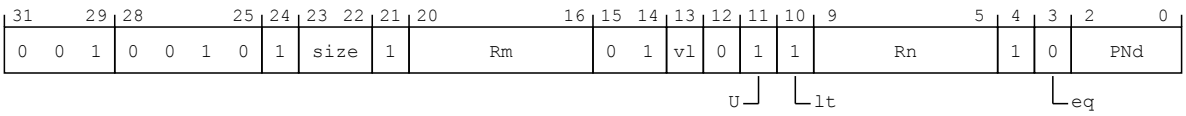

Encoding

```
WHILELO  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = TRUE;
let invert : boolean = FALSE;
let op : CmpOp = Cmp_LT;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let op2val : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let op1val : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (op1val <  op2val);
        when Cmp_LE => cond = (op1val <= op2val);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 + 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.924 WHILELO (predicate pair)

While incrementing unsigned scalar lower than scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

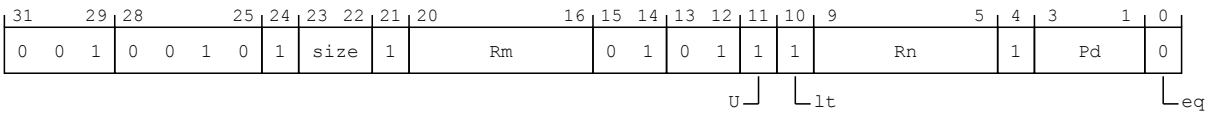

Encoding

```
WHILELO  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_LT;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to (elements*2)-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.925 WHILELS (predicate)

While incrementing unsigned scalar lower than or same as scalar

This instruction generates a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower than or the same as the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum unsigned integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

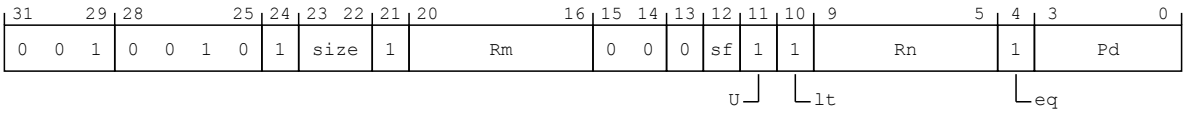

Encoding

```
WHILELS  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_LE;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.926 WHILELS (predicate as counter)

While incrementing unsigned scalar lower than or same as scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower than or the same as the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum unsigned integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

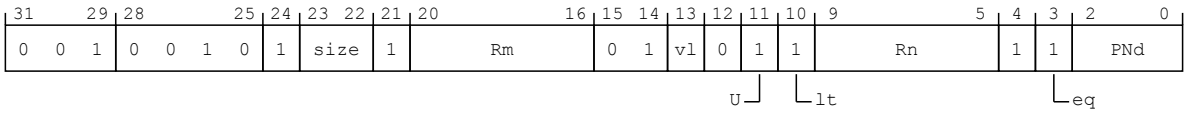

Encoding

```
WHILELS  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = TRUE;
let invert : boolean = FALSE;
let op : CmpOp = Cmp_LE;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let op2val : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let op1val : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (op1val <  op2val);
        when Cmp_LE => cond = (op1val <= op2val);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 + 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.927 WHILELS (predicate pair)

While incrementing unsigned scalar lower than or same as scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower than or the same as the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum unsigned integer value, then a condition that includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

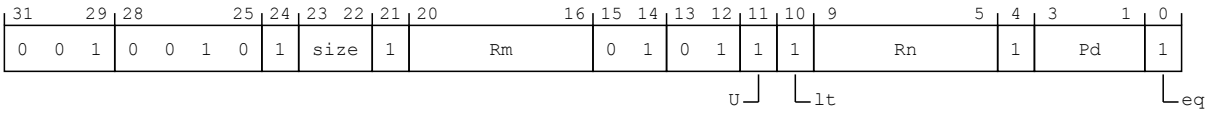

Encoding

```
WHILELS  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = TRUE;
let op : CmpOp = Cmp_LE;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
00	B
01	H
10	S
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to (elements*2)-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.928 WHILELT (predicate)

While incrementing signed scalar less than scalar

This instruction generates a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE

(FEAT_SVE || FEAT_SME)

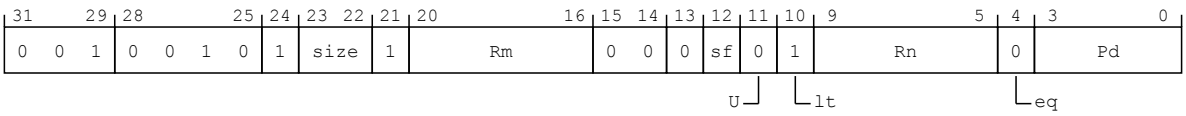

Encoding

```
WHILELT  <Pd>.<T>, <R><n>, <R><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 32 << UInt(sf);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_LT;
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<R>

Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.929 WHILELT (predicate as counter)

While incrementing signed scalar less than scalar (predicate-as-counter)

This instruction generates a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

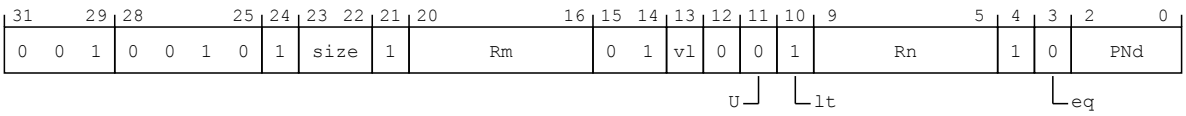

Encoding

```
WHILELT  <PNd>.<T>, <Xn>, <Xm>, <vl>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt('1'::PNd);
let unsigned : boolean = FALSE;
let invert : boolean = FALSE;
let op : CmpOp = Cmp_LT;
let width : integer = 2 << UInt(vl);
```

Assembler Symbols

<PNd>

Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

<vl>

Is the vl specifier, encoded in “vl”:

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if IsFeatureImplemented(FEAT_SVE2p1) then CheckSVEEnabled(); else CheckStreamingSVEEnabled(); end;
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = width * (VL DIV esize);
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL);
var last : boolean = TRUE;
var count : integer = 0;

let op2val : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to elements-1 do
    let op1val : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (op1val <  op2val);
        when Cmp_LE => cond = (op1val <= op2val);
    end;

    last = last && cond;
    if last then count = count + 1; end;
    operand1 = operand1 + 1;
end;

result = EncodePredCount{PL}(esize, elements, count, invert);
PSTATE.[N,Z,C,V] = PredCountTest(elements, count, invert);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.930 WHILELT (predicate pair)

While incrementing signed scalar less than scalar (pair of predicates)

This instruction generates a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SME2 || FEAT_SVE2p1)

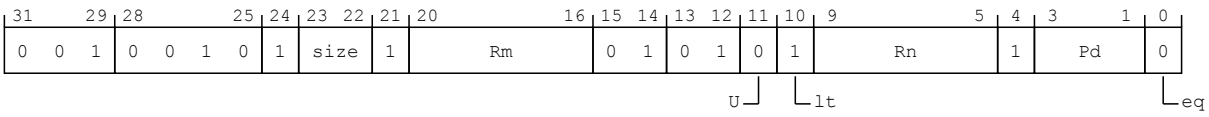

Encoding

```
WHILELT  { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) && !IsFeatureImplemented(FEAT_SVE2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let rsize : integer{} = 64;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d0 : integer = UInt(Pd::'0');
let d1 : integer = UInt(Pd::'1');
let unsigned : boolean = FALSE;
let op : CmpOp = Cmp_LT;
```

Assembler Symbols

<Pd1>

Is the name of the first destination scalable predicate register, encoded as “Pd” times 2.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<T>
11	D

<Pd2>

Is the name of the second destination scalable predicate register, encoded as “Pd” times 2 plus 1.

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL*2) = Ones{};
var operand1 : bits(rsize) = X{}(n);
let operand2 : bits(rsize) = X{}(m);
var result : bits(PL*2);
var last : boolean = TRUE;
let psize : integer{} = esize DIV 8;

let element2 : integer = if unsigned then UInt(operand2) else SInt(operand2);
for e = 0 to (elements*2)-1 do
    let element1 : integer = if unsigned then UInt(operand1) else SInt(operand1);
    var cond : boolean;
    case op of
        when Cmp_LT => cond = (element1 <  element2);
        when Cmp_LE => cond = (element1 <= element2);
    end;

    last = last && cond;
    let pbit : bit = if last then '1' else '0';
    result[e*:psize] = ZeroExtend{psize}(pbit);
    operand1 = operand1 + 1;
end;

PSTATE.[N,Z,C,V] = PredTest{PL*2}(mask, result, esize);
P{PL}(d0) = result[PL-1:0];
P{PL}(d1) = result[PL*2-1:PL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.931 WHILERW

While free of read-after-write conflicts

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr,addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

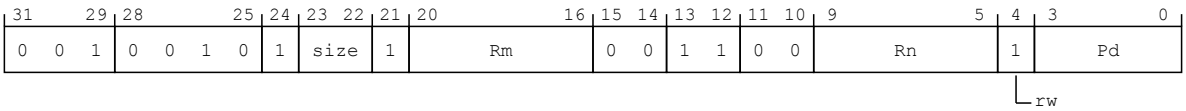

Encoding

```
WHILERW  <Pd>.<T>, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
let src1 : bits(64) = X{}(n);
let src2 : bits(64) = X{}(m);
let operand1 : integer = UInt(src1);
let operand2 : integer = UInt(src2);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

let diff : integer = Abs(operand2 - operand1) DIVRM (esize DIV 8);
for e = 0 to elements-1 do
    if diff == 0 || e < diff then
        result[e*:psize] = ZeroExtend{psize}('1');
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.932 WHILEWR

While free of write-after-read/write conflicts

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr,addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. This instruction sets the First (N), None (Z), and !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

SVE2

(FEAT_SVE2 || FEAT_SME)

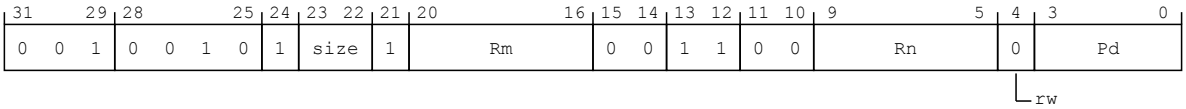

Encoding

```
WHILEWR  <Pd>.<T>, <Xn>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let d : integer = UInt(Pd);
```

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Xn>

Is the 64-bit name of the first source general-purpose register, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the second source general-purpose register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mask : bits(PL) = Ones{};
let src1 : bits(64) = X{}(n);
let src2 : bits(64) = X{}(m);
let operand1 : integer = UInt(src1);
let operand2 : integer = UInt(src2);
var result : bits(PL);
let psize : integer{} = esize DIV 8;

let diff : integer = (operand2 - operand1) DIVRM (esize DIV 8);
for e = 0 to elements-1 do
    if diff <= 0 || e < diff then
        result[e*:psize] = ZeroExtend{psize}('1');
    else
        result[e*:psize] = ZeroExtend{psize}('0');
    end;
end;

PSTATE.[N,Z,C,V] = PredTest{PL}(mask, result, esize);
P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.933 WRFFR

Write the first-fault register

This instruction reads the source predicate register, and writes the value in the register to the first-fault register (FFR). This instruction is intended to restore a saved FFR and is not recommended for general use by applications.

This instruction requires that the source predicate contains a monotonic predicate value, in which starting from bit 0 there are zero or more 1 bits, followed only by 0 bits in any remaining bit positions. If the source is not a monotonic predicate value, then the resulting value in the FFR will be UNPREDICTABLE. It is not possible to generate a non-monotonic value in FFR when using SETFFR followed by first-fault or non-fault loads.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

SVE

(FEAT_SVE)

31 29	28 25	24	23 22	21 19	18	17 16	15 12	11 9	8 5	4 0
0 0 1	0 0 1 0	1	0 0	1 0 1	0	0 0	1 0 0 1	0 0 0	Pn	0 0 0 0 0
			opc							

Encoding

```
WRFFR  <Pn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Pn);
```

Assembler Symbols

<Pn>

Is the name of the source scalable predicate register, encoded in the “Pn” field.

Operation

```
CheckNonStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let operand : bits(PL) = P{}(n);

let hsb : integer{} = HighestSetBit(operand);
if hsb < 0 || IsOnes(operand[hsb:0]) then
    FFR{PL}() = operand;
else // not a monotonic predicate
    FFR{PL}() = ARBITRARY : bits(PL);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.934 XAR

Bitwise exclusive-OR and rotate right by immediate

This instruction performs a bitwise exclusive-OR between the corresponding elements of the first source vector and the second source vector, and then rotates each result element right by an immediate amount. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

SVE2

(FEAT_SVE2 || FEAT_SME)

31 29	28 25	24	23 22	21	20 19	18 16	15 13	12 10	9 5	4 0
0 0 0	0 0 1 0	0	tszh	1	tszl	imm3	0 0 1	1 0 1	Zm	Zdn

Encoding

```
XAR  <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let tsize : bits(4) = tszh::tszl;
if tsize == '0000' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let m : integer = UInt(Zm);
let dn : integer = UInt(Zdn);
let rot : integer = (2 * esize) - UInt(tsize::imm3);
```

Assembler Symbols

<Zdn>

Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T>

Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const>

Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:tszl:imm3”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let operand1 : bits(VL) = Z{}(dn);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for e = 0 to elements-1 do
    let element1 : bits(esize) = operand1[e*:esize];
    let element2 : bits(esize) = operand2[e*:esize];
    result[e*:esize] = ROR(element1 XOR element2, rot);
end;
Z{VL}(dn) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is CONSTRAINED UNPREDICTABLE:

- The `MOVPRFX` must be unpredicated.
- The `MOVPRFX` must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.935 ZIP1, ZIP2 (predicates)

Interleave elements from two half predicates

This instruction interleaves alternating elements from the lowest or highest halves of the first and second source predicates, and places the values in the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: High halves and Low halves

High halves

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–11	10	9	8–5	4	3–0
0 0 0	0 0 1 0	1	size	1	0	Pm	0 1 0	0 0	1	0	Pn	0	Pd
								opc	H				

Encoding

```
ZIP2  <Pd>.<T>, <Pn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let part : integer = 1;
```

Low halves

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20	19–16	15–13	12–11	10	9	8–5	4	3–0
0 0 0	0 0 1 0	1	size	1	0	Pm	0 1 0	0 0	0	0	Pn	0	Pd
								opc	H				

Encoding

```
ZIP1  <Pd>.<T>, <Pn>.<T>, <Pm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Pn);
let m : integer = UInt(Pm);
let d : integer = UInt(Pd);
let part : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Pd>

Is the name of the destination scalable predicate register, encoded in the “Pd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn>

Is the name of the first source scalable predicate register, encoded in the “Pn” field.

<Pm>

Is the name of the second source scalable predicate register, encoded in the “Pm” field.

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let pairs : integer = VL DIV (esize * 2);
let operand1 : bits(PL) = P{}(n);
let operand2 : bits(PL) = P{}(m);
var result : bits(PL);

let base : integer = part * pairs;
for p = 0 to pairs-1 do
    result[(2*p+0)*:(esize DIV 8)] = operand1[(base+p)*:(esize DIV 8)];
    result[(2*p+1)*:(esize DIV 8)] = operand2[(base+p)*:(esize DIV 8)];
end;

P{PL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.936 ZIP1, ZIP2 (vectors)

Interleave elements from two half vectors

This instruction interleaves alternating elements from the lowest or highest halves of the first and second source vectors, and places the values in the destination vector. This instruction is unpredicated.

The 128-bit element variant requires that the Effective SVE vector length is at least 256 bits. ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled.

It has encodings from 4 classes: High halves, High halves (quadwords), Low halves, and Low halves (quadwords)

High halves

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 1 1	0 0	1	Zn	Zd
								H		

Encoding

```
ZIP2  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```

High halves (quadwords)

(FEAT_F64MM)

31–29	28–25	24–23	22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1 1	0	1	Zm	0 0 0	0 0	1	Zn	Zd
							opc	H		

Encoding

```
ZIP2  <Zd>.Q, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Low halves

(FEAT_SVE || FEAT_SME)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1	size	1	Zm	0 1 1	0 0	0	Zn	Zd
								H		

Encoding

```
ZIP1  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE) && !IsFeatureImplemented(FEAT_SME) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```

Low halves (quadwords)

(FEAT_F64MM)

31–29	28–25	24–23	22	21	20–16	15–13	12–11	10	9–5	4–0
0 0 0	0 0 1 0	1 1	0	1	Zm	0 0 0	0 0	0	Zn	Zd
							opc	H		

Encoding

```
ZIP1  <Zd>.Q, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_F64MM) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
if esize < 128 then
    CheckSVEEnabled();
else
    CheckNonStreamingSVEEnabled();
end;
let VL : integer{} = CurrentVL();
if VL < esize * 2 then EndOfDecode(Decode_UNDEF); end;
let pairs : integer = VL DIV (esize * 2);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL) = Zeros{};

let base : integer = part * pairs;
for p = 0 to pairs-1 do
    result[(2*p+0)*:esize] = operand1[(base+p)*:esize];
    result[(2*p+1)*:esize] = operand2[(base+p)*:esize];
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.937 ZIPQ1

Interleave elements from low halves of each pair of quadword vector segments

This instruction interleaves alternating elements from low halves of the corresponding 128-bit vector segments of the first and second source vectors, and places them in the corresponding elements of the destination vector segment. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

31–29	28–25	24	23–22	21	20–16	15–13	12–11	10	9–5	4–0
0 1 0	0 0 1 0	0	size	0	Zm	1 1 1	0 0	0 (H)	Zn	Zd

Encoding

```
ZIPQ1  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 0;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let pairs : integer = elements DIV 2;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let base : integer = s * elements + part * pairs;
    for p = 0 to pairs-1 do
        result[(s * elements + 2 * p + 0)*:esize] = operand1[(base + p)*:esize];
        result[(s * elements + 2 * p + 1)*:esize] = operand2[(base + p)*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C8.2.938 ZIPQ2

Interleave elements from high halves of each pair of quadword vector segments

This instruction interleaves alternating elements from high halves of the corresponding 128-bit vector segments of the first and second source vectors, and places them in the corresponding elements of the destination vector segment. This instruction is unpredicated.

SVE2

(FEAT_SVE2p1 || FEAT_SME2p1)

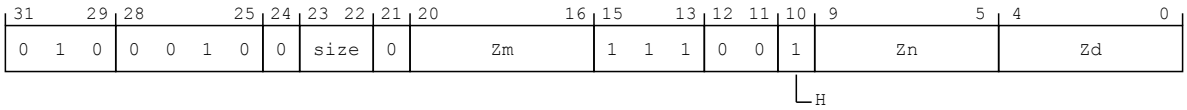

Encoding

```
ZIPQ2  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SVE2p1) && !IsFeatureImplemented(FEAT_SME2p1) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd);
let part : integer = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSVEEnabled();
let VL : integer{} = CurrentVL();
let segments : integer = VL DIV 128;
let elements : integer = 128 DIV esize;
let pairs : integer = elements DIV 2;
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for s = 0 to segments-1 do
    let base : integer = s * elements + part * pairs;
    for p = 0 to pairs-1 do
        result[(s * elements + 2 * p + 0)*:esize] = operand1[(base + p)*:esize];
        result[(s * elements + 2 * p + 1)*:esize] = operand2[(base + p)*:esize];
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C9
SME Instruction Descriptions

This chapter describes the SME instructions.

It contains the following sections:

- About the SME instructions
- Alphabetical list of SME instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.1 About the SME instructions

Alphabetical list of SME instructions gives full descriptions of the SME instructions.

A64 instruction set encoding provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2 Alphabetical list of SME instructions

This section lists every section in the SME category of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.1 ADD (to vector, multiple vectors)

Multi-vector add by vector to multi-vector

This instruction adds elements of the second source vector to the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15			11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	1	1	0	0	0	Zdn				0

op

Encoding

```
ADD  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15			11	10	9	8	7			5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	1	0	0	0	Zdn			0	0

op

Encoding

```
ADD  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = element1 + element2;
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.2 ADD (to array, array and multiple vectors)

Multi-vector accumulate to ZA array vectors

This instruction destructively adds all elements of the two or four source vectors to the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv		1	1	1	Zm				0	1	0	off3		
																												S			

Encoding

```
ADD  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv		1	1	1	Zm			0	0	1	0	off3		
																												S			

Encoding

```
ADD  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zm4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = ZAvector{}(vec);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = element1 + element2;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.3 ADD (to array, multiple and single vector)

Multi-vector add by vector to ZA array vectors

This instruction adds all the corresponding elements of the second source vector and the two or four first source vectors and places the results in the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	1	0	sz	1	0	Zm		0	Rv		1	1	Zn		1	0	off3	

S

Encoding

```
ADD  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	1	0	sz	1	1	Zm		0	Rv		1	1	Zn		1	0	off3	

S

Encoding

```
ADD  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = element1 + element2;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.4 ADD (to array, multiple vectors)

Multi-vector add to ZA array vectors

This instruction adds all the corresponding elements of the two or four second source vectors and first source vectors and places the results in the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0	0	Rv		1	1	0	Zn	0	1	0	off3

S

Encoding

```
ADD  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0	1	0	Rv		1	1	0	Zn	0	0	1	0	off3

S

Encoding

```
ADD  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = element1 + element2;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.5 ADDHA

Add horizontally vector elements to ZA tile

This instruction adds each element of the source vector to the corresponding Active element of each horizontal slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a horizontal slice is considered active if its corresponding element in the second governing predicate is TRUE and the element corresponding to its horizontal slice number in the first governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21		19	18	17	16	15		13	12		10	9		5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	Pm			Pn			Zn			0	0	0	ZAda	
									op						V														

Encoding

```
ADDHA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let da : integer = UInt(ZAda);
```

64-bit

(FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21		19	18	17	16	15		13	12		10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0	Pm			Pn			Zn			0	0	ZAda		
									op						V														

Encoding

```
ADDHA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let da : integer = UInt(ZAda);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand_src : bits(VL) = Z{}(n);
let operand_acc : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);

for col = 0 to dim-1 do
    let element : bits(esize) = operand_src[col*:esize];
    for row = 0 to dim-1 do
        var res : bits(esize) = operand_acc[(row*dim+col)*:esize];
        if (ActivePredicateElement{PL}(mask1, row, esize) &&
              ActivePredicateElement{PL}(mask2, col, esize)) then
            res = res + element;
        end;
        result[(row*dim+col)*:esize] = res;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.6 ADDSPL

Add multiple of Streaming SVE predicate register size to scalar register

This instruction adds the Streaming SVE predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and places the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME

(FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–12	11	10–5	4–0
0 0 0	0 0 1 0	0	0	1	1	Rn	0 1 0 1	1	imm6	Rd
				op						

Encoding

```
ADDSPL  <Xd|SP>, <Xn|SP>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let imm : integer = SInt(imm6);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is the signed immediate operand, in the range -32 to 31, encoded in the “imm6” field.

Operation

```
CheckSMEEnabled();
let SVL : integer = CurrentSVL();
let len : integer = imm * (SVL DIV 64);
let operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let result : bits(64) = operand1 + len;

if d == 31 then
    SP{64}() = result;
else
    X{64}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.7 ADDSVL

Add multiple of Streaming SVE vector register size to scalar register

This instruction adds the Streaming SVE vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and places the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME

(FEAT_SME)

31–29	28–25	24	23	22	21	20–16	15–12	11	10–5	4–0
0 0 0	0 0 1 0	0	0	0	1	Rn	0 1 0 1	1	imm6	Rd
				op						

Encoding

```
ADDSVL  <Xd|SP>, <Xn|SP>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let d : integer = UInt(Rd);
let imm : integer = SInt(imm6);
```

Assembler Symbols

<Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<imm>

Is the signed immediate operand, in the range -32 to 31, encoded in the “imm6” field.

Operation

```
CheckSMEEnabled();
let SVL : integer = CurrentSVL();
let len : integer = imm * (SVL DIV 8);
let operand1 : bits(64) = if n == 31 then SP{64}() else X{64}(n);
let result : bits(64) = operand1 + len;

if d == 31 then
    SP{64}() = result;
else
    X{64}(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.8 ADDVA

Add vertically vector elements to ZA tile

This instruction adds each element of the source vector to the corresponding Active element of each vertical slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a vertical slice is considered active if its corresponding element in the first governing predicate is TRUE and the element corresponding to its vertical slice number in the second governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	13	12	10	9	5	4	3	2	1	0
1	1	0	0 0 0	0	0	1	0	0 1	0	0	0	1	Pm		Pn		Zn		0	0	0	ZAda	

op V

Encoding

```
ADDVA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let da : integer = UInt(ZAda);
```

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	13	12	10	9	5	4	3	2	0
1	1	0	0 0 0	0	0	1	1	0 1	0	0	0	1	Pm		Pn		Zn		0	0	ZAda	

op V

Encoding

```
ADDVA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let da : integer = UInt(ZAda);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand_src : bits(VL) = Z{}(n);
let operand_acc : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);

for row = 0 to dim-1 do
    let element : bits(esize) = operand_src[row*:esize];
    for col = 0 to dim-1 do
        var res : bits(esize) = operand_acc[(row*dim+col)*:esize];
        if (ActivePredicateElement{PL}(mask1, row, esize) &&
              ActivePredicateElement{PL}(mask2, col, esize)) then
            res = res + element;
        end;
        result[(row*dim+col)*:esize] = res;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.9 BF1CVT, BF2CVT

Multi-vector 8-bit floating-point convert to BFloat16

This instruction converts each 8-bit floating-point element of the source vector to BFloat16 while downscaling the value, and places the results in the corresponding 16-bit elements of the destination vectors. BF1CVT scales the values by $2^{-\text{UInt(FPMR.LSCALE[5:0])}}$. BF2CVT scales the values by $2^{-\text{UInt(FPMR.LSCALE2[5:0])}}$. The 8-bit floating-point encoding format for BF1CVT is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for BF2CVT is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: BF1CVT and BF2CVT

BF1CVT

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	0	1	1	0	0	1	1	0	1	1	1	0	0	0	Zn		Zd		0
								opc																		L

Encoding

```
BF1CVT  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = FALSE;
```

BF2CVT

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	0	0	Zn		Zd		0
								opc																		L

Encoding

```
BF2CVT  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 8;
let operand : bits(VL) = Z{}(n);
var result : bits(2*VL);

for e = 0 to elements-1 do
    let element : bits(8) = operand[e*:8];
    result[e*:16] = FP8ConvertBF(element, issrc2, FPCR(), FPMR());
end;

Z{VL}(d+0) = result[VL-1:0];
Z{VL}(d+1) = result[2*VL-1:VL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.10 BF1CVTL, BF2CVTL

Multi-vector 8-bit floating-point convert to deinterleaved BFloat16

This instruction converts each 8-bit floating-point element of the source vector to BFloat16 while downscaling the value, and places the two-way deinterleaved results in the corresponding 16-bit elements of the destination vectors. BF1CVTL scales the values by $2^{-\mathrm{UInt(FPMR.LSCALE[5:0])}}$. BF2CVTL scales the values by $2^{-\mathrm{UInt(FPMR.LSCALE2[5:0])}}$. The 8-bit floating-point encoding format for BF1CVTL is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for BF2CVTL is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: BF1CVTL and BF2CVTL

BF1CVTL

(FEAT_SME2 && FEAT_FP8)

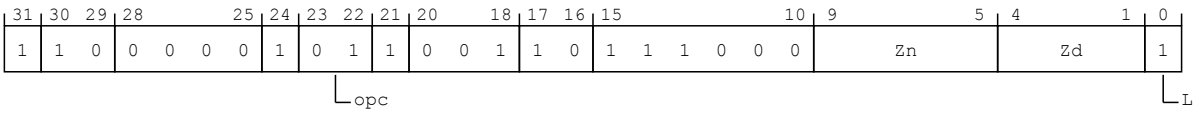

Encoding

```
BF1CVTL  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = FALSE;
```

BF2CVTL

(FEAT_SME2 && FEAT_FP8)

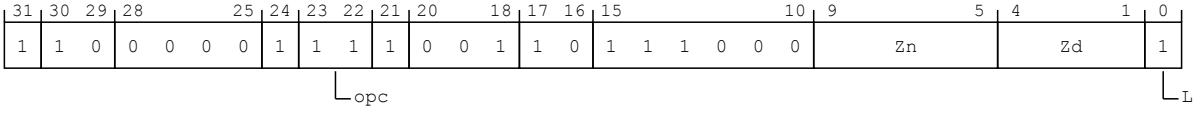

Encoding

```
BF2CVTL  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV 16;
let operand : bits(VL) = Z{}(n);
var result1 : bits(VL);
var result2 : bits(VL);

for p = 0 to pairs-1 do
    let element1 : bits(8) = operand[(2*p + 0)*:8];
    let element2 : bits(8) = operand[(2*p + 1)*:8];
    result1[p*:16] = FP8ConvertBF(element1, issrc2, FPCR(), FPMR());
    result2[p*:16] = FP8ConvertBF(element2, issrc2, FPCR(), FPMR());
end;

Z{VL}(d+0) = result1;
Z{VL}(d+1) = result2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.11 BFADD

Multi-vector BFloat16 accumulate to ZA array vectors

This instruction destructively adds all elements of the two or four source vectors to the corresponding BFloat16 elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	0	0	Rv		1	1	1	Zm			0	0	0	off3		
									sz																		S		

Encoding

```
BFADD  ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	1	0	Rv		1	1	1	Zm			0	0	0	0	off3		
									sz																			S		

Encoding

```
BFADD  ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zm4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = ZAvector{}(vec);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFAdd_ZA(element1, element2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.12 BFCLAMP

Multi-vector BFloat16 clamp to minimum/maximum number

This instruction clamps each BFloat16 element in the two or four destination vectors to between the BFloat16 minimum value in the corresponding element of the first source vector and the BFloat16 maximum value in the corresponding element of the second source vector and destructively places the clamped results in the corresponding elements of the two or four destination vectors.

Regardless of the value of FPCR.AH, the behavior is as follows for each minimum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				1	1	0	0	0	0	Zn				Zd				0
								size																					op

Encoding

```
BFCLAMP  { <Zd1>.H-<Zd2>.H }, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				1	1	0	0	1	0	Zn				Zd			0	0
								size																					op

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
BFCLAMP  { <Zd1>.H-<Zd4>.H }, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = Z{}(d+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        let element3 : bits(16) = operand3[e*:16];
        let maxelement : bits(16) = BFMaxNum{}(element1, element3, FPCR());
        results[[r]][e*:16] = BFMinNum{16}(maxelement, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.13 BFCVT (BFloat16 to 8-bit floating-point)

Multi-vector BFloat16 convert to 8-bit floating-point

This instruction converts each BFloat16 element of the two source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE)}}$, and places the results in the half-width elements of the destination vector. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SME2

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	1	1	1	0	0	0	Zn		0	Zd	

op

Encoding

```
BFCVT  <Zd>.B, { <Zn1>.H-<Zn2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[(0*elements + e)*:8] = BFConvertFP8(element1, FPCR(), FPMR());
    result[(1*elements + e)*:8] = BFConvertFP8(element2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.14 BFCVT (single-precision to BFloat16)

Multi-vector single-precision convert to BFloat16

This instruction converts each element of the two source vectors from single-precision to BFloat16 floating-point, and places the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1	1	0	0	0	Zn		0	Zd	
									op															N		

Encoding

```
BFCVT  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let res1 : bits(16) = FPConvertBF(element1, FPCR());
    let res2 : bits(16) = FPConvertBF(element2, FPCR());
    result[e*:16] = res1;
    result[(elements+e)*:16] = res2;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.15 BFCVTN

Multi-vector single-precision convert to interleaved BFloat16

This instruction converts each element of the two source vectors from single-precision to BFloat16 floating-point, and places the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9		6	5	4		0
1	1	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1	1	0	0	0	Zn			1	Zd		
									op																N			

Encoding

```
BFCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let res1 : bits(16) = FPConvertBF(element1, FPCR());
    let res2 : bits(16) = FPConvertBF(element2, FPCR());
    result[(2*e + 0)*:16] = res1;
    result[(2*e + 1)*:16] = res2;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.16 BFDOT (multiple and indexed vector)

Multi-vector BFloat16 dot product by indexed element to single-precision

This instruction calculates the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index that selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5		3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2		Zn				0	1	1	off3		
																			op								opc2				

Encoding

```
BFDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5		3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn				0	0	1	1	off3		
																			op									opc2				

Encoding

```
BFDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let elt2_a : bits(16) = operand2[(2 * s + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * s + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.17 BFDOT (multiple and single vector)

Multi-vector BFloat16 dot product by vector to single-precision

This instruction calculates the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	0	0	Zn				1	0	off3		
																										opc				

Encoding

```
BFDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	0	Zn				1	0	off3		
																										opc				

Encoding

```
BFDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
        let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.18 BFDOT (multiple vectors)

Multi-vector BFloat16 dot product to single-precision

This instruction calculates the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first and second source vectors. The single-precision dot product results are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	1	1	0	1	Zm		0	0	Rv		1	0	Zn		0	1	0	off3	

opc

Encoding

```
BFDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	1	1	0	1	Zm		0	1	0	Rv		1	0	Zn		0	0	1	0	off3	

opc

Encoding

```
BFDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
        let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.19 BFMAX (multiple and single vector)

Multi-vector BFloat16 maximum by vector

This instruction determines the maximum of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15			11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	0	0	0	1	0	0	0	Zdn			0
								size																	op					o2

Encoding

```
BFMAX  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15			11	10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	1	0	0	1	0	0	0	Zdn		0	0
								size																	op					o2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
BFMAX  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMax{16}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.20 BFMAX (multiple vectors)

Multi-vector BFloat16 maximum

This instruction determines the maximum of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	Zdn				0
								size																		opc					o2

Encoding

```
BFMAX  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm			0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn			0	0
								size																		opc					o2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
BFMAX  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMax{16}(element1, element2, FPCR());
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.21 BFMAXNM (multiple and single vector)

Multi-vector BFloat16 maximum number by vector

This instruction determines the maximum number value of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	1	0
1	1	0	0	0	0	1	0	0	1	0	Zm	1	0	1	0	0	0	0	1	0	Zdn	0
							size												op			o2

Encoding

```
BFMAXNM  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	2	1	0
1	1	0	0	0	0	1	0	0	1	0	Zm	1	0	1	0	1	0	0	1	0	Zdn	0	0
							size												op				o2

Encoding

```
BFMAXNM  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMaxNum{16}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.22 BFMAXNM (multiple vectors)

Multi-vector BFloat16 maximum number

This instruction determines the maximum number value of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0		0	1	Zdn				0
								size																			opc					o2

Encoding

```
BFMAXNM  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20		18	17							10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm			0	0	1	0	1	1	1	0	0	1	0		0	1	Zdn				0	0
								size																			opc						o2

Encoding

```
BFMAXNM  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMaxNum{16}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.23 BFMIN (multiple and single vector)

Multi-vector BFloat16 minimum by vector

This instruction determines the minimum of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15			11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	0	0	0	1	0	0	0	Zdn			1
								size																	op					o2

Encoding

```
BFMIN  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15			11	10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	1	0	0	1	0	0	0	Zdn		0	1
								size																	op					o2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
BFMIN  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMin{16}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.24 BFMIN (multiple vectors)

Multi-vector BFloat16 minimum

This instruction determines the minimum of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	Zdn				1
								size																		opc					o2

Encoding

```
BFMIN  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			18	17						10	9		7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn			0	1
								size																		opc						o2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
BFMIN  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMin{16}(element1, element2, FPCR());
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.25 BFMINNM (multiple and single vector)

Multi-vector BFloat16 minimum number by vector

This instruction determines the minimum number value of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	0		Zm			1	0	1	0	0	0	0	1	0	0	1		Zdn			1
								size																		op					o2

Encoding

```
BFMINNM  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	0		Zm			1	0	1	0	1	0	0	1	0	0	1		Zdn		0	1
								size																		op					o2

Encoding

```
BFMINNM  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMinNum{16}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.26 BFMINNM (multiple vectors)

Multi-vector BFloat16 minimum number

This instruction determines the minimum number value of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1		Zm			0	1	0	1	1	0	0	0	1	0		0	1		Zdn			1
								size																			opc					o2

Encoding

```
BFMINNM  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_B16B16)

31	30	29	28			25	24	23	22	21	20			18	17						10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	0	0	1		Zm			0	0	1	0	1	1	1	0	0	1	0	0	1		Zdn			0	1
								size																			opc						o2

Encoding

```
BFMINNM  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        results[[r]][e*:16] = BFMinNum{16}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.27 BFMLA (multiple and indexed vector)

Multi-vector BFloat16 fused multiply-add by indexed element

This instruction multiplies the indexed element of the second source vector by the corresponding BFloat16 elements of the two or four first source vectors and destructively adds these values without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				1	0	i3l	off3		
																										op		S			

Encoding

```
BFMLA  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		1	i3h		Zn				0	1	0	i3l	off3		
																											op		S			

Encoding

```
BFMLA  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 16;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(16) = op1[e*:16];
        let segmentbase : integer = e - (e MOD eltspersegment);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let s : integer = segmentbase + index;
        let elem2 : bits(16) = op2[s*:16];
        let elem3 : bits(16) = op3[e*:16];
        result[e*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.28 BFMLA (multiple and single vector)

Multi-vector BFloat16 fused multiply-add by vector

This instruction multiplies the corresponding BFloat16 elements of the two or four first source vector with the corresponding elements of the second source vector and destructively adds these values without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	1	0	Zm				0	Rv		1	1	1	Zn				0	0	off3		
									sz																		S		

Encoding

```
BFMLA  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		1	1	1	Zn				0	0	off3		
									sz																		S		

Encoding

```
BFMLA  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(16) = op1[e*:16];
        let elem2 : bits(16) = op2[e*:16];
        let elem3 : bits(16) = op3[e*:16];
        result[e*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.29 BFMLA (multiple vectors)

Multi-vector BFloat16 fused multiply-add

This instruction multiplies the corresponding BFloat16 elements of the two or four first and second source vectors and destructively adds these values without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

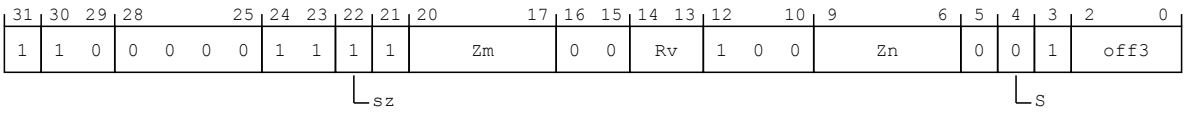

Encoding

```
BFMLA  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

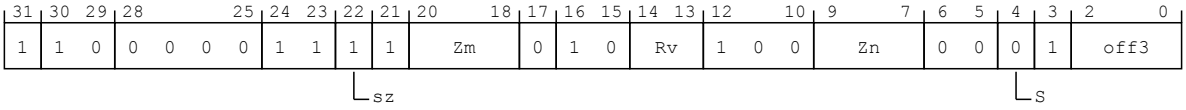

Encoding

```
BFMLA  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(16) = op1[e*:16];
        let elem2 : bits(16) = op2[e*:16];
        let elem3 : bits(16) = op3[e*:16];
        result[e*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.30 BFMLAL (multiple and indexed vector)

Multi-vector BFloat16 multiply-add by indexed element to single-precision

This instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index that selects the same element position within each 128-bit vector segment.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	1	1	0	0	0	Zm	i3h	Rv		1	i3l		Zn		1	0	off3	
																				op	S		

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	0	0	1	Zm	0	Rv		1	i3h		Zn		0	1	0	i3l	off2	
																				op		S			

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1		Zm			1		Rv		1		i3h			Zn			0	0	1	0	i3l	off2
																											op			S		

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let elem1 : bits(16) = op1[(2 * e + i)*:16];
            let elem2 : bits(16) = op2[s*:16];
            let elem3 : bits(32) = op3[e*:32];
            result[e*:32] = BFMulAddH_ZA(elem3, elem1, elem2, FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.31 BFMLAL (multiple and single vector)

Multi-vector BFloat16 multiply-add by vector to single-precision

This instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28 25	24	23	22	21	20	19 16	15	14 13	12 10	9 5	4	3	2 0
1	1	0	0 0 0 0	1	0	0	1	0	Zm	0	Rv	0 1 1	Zn	1	0	off3
														op	S	

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28 25	24	23	22	21	20	19 16	15	14 13	12 10	9 5	4	3	2	1 0
1	1	0	0 0 0 0	1	0	0	1	0	Zm	0	Rv	0 1 0	Zn	1	0	0	off2
														op	S	o2	

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				1	0	0	off2	
																										op	S	o2		

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = op1[(2 * e + i)*:16];
            let elem2 : bits(16) = op2[(2 * e + i)*:16];
            let elem3 : bits(32) = op3[e*:32];
            result[e*:32] = BFMulAddH_ZA(elem3, elem1, elem2, FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.32 BFMLAL (multiple vectors)

Multi-vector BFloat16 multiply-add to single-precision

This instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		0	1	0	Zn				0	1	0	0	off2	
																										op		S			

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	1	0	0	off2	
																											op		S		

Encoding

```
BFMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = op1[(2 * e + i)*:16];
            let elem2 : bits(16) = op2[(2 * e + i)*:16];
            let elem3 : bits(32) = op3[e*:32];
            result[e*:32] = BFMulAddH_ZA(elem3, elem1, elem2, FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.33 BFMLS (multiple and indexed vector)

Multi-vector BFloat16 fused multiply-subtract by indexed element

This instruction multiplies the indexed element of the second source vector by the corresponding BFloat16 elements of the two or four first source vectors and destructively subtracts these values without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				1	1	i3l	off3		
																										op		S			

Encoding

```
BFMLS  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		1	i3h		Zn				0	1	1	i3l	off3		
																											op		S			

Encoding

```
BFMLS  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 16;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(16) = BFNeg(op1[e*:16]);
        let segmentbase : integer = e - (e MOD eltspersegment);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let s : integer = segmentbase + index;
        let elem2 : bits(16) = op2[s*:16];
        let elem3 : bits(16) = op3[e*:16];
        result[e*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.34 BFMLS (multiple and single vector)

Multi-vector BFloat16 fused multiply-subtract by vector

This instruction multiplies the corresponding BFloat16 elements of the two or four first source vector with the corresponding elements of the second source vector and destructively subtracts these values without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

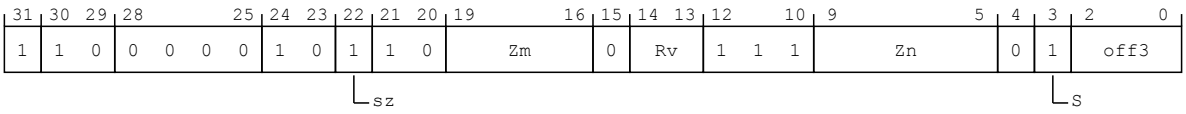

Encoding

```
BFMLS  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

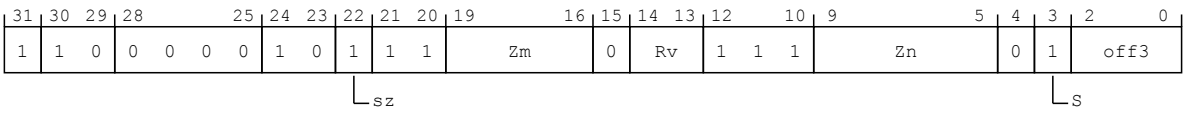

Encoding

```
BFMLS  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(16) = BFNeg(op1[e*:16]);
        let elem2 : bits(16) = op2[e*:16];
        let elem3 : bits(16) = op3[e*:16];
        result[e*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.35 BFMLS (multiple vectors)

Multi-vector BFloat16 fused multiply-subtract

This instruction multiplies the corresponding BFloat16 elements of the two or four first and second source vectors and destructively subtracts these values without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv		1	0	0		Zn				0	1	1	off3			
									sz																			S					

Encoding

```
BFMLS  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12			10	9			7	6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	1	0	Rv		1	0	0		Zn				0	0	1	1	off3			
									sz																					S				

Encoding

```
BFMLS  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(16) = BFNeg(op1[e*:16]);
        let elem2 : bits(16) = op2[e*:16];
        let elem3 : bits(16) = op3[e*:16];
        result[e*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.36 BFMLSL (multiple and indexed vector)

Multi-vector BFloat16 multiply-subtract by indexed element from single-precision

This instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index that selects the same element position within each 128-bit vector segment.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	1	1	0	0	0	Zm	i3h	Rv		1	i3l		Zn		1	1	off3	
																				op	S		

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	0	0	1	Zm	0	Rv		1	i3h		Zn		0	1	1	i3l	off2	
																				op		S			

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1		Zm			1		Rv		1		i3h			Zn		0	0	1	1	i3l	off2
																												op		S		

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let elem1 : bits(16) = BFNeg(op1[(2 * e + i)*:16]);
            let elem2 : bits(16) = op2[s*:16];
            let elem3 : bits(32) = op3[e*:32];
            result[e*:32] = BFMulAddH_ZA(elem3, elem1, elem2, FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.37 BFMLSL (multiple and single vector)

Multi-vector BFloat16 multiply-subtract by vector from single-precision

This instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	1	0	0	1	0	Zm		0	Rv		0	1	1	Zn	1	1	off3	

op, S

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	0	Zm		0	Rv		0	1	0	Zn	1	1	0	off2	

op, S, o2

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				1	1	0	off2	

op, S, o2

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = BFNeg(op1[(2 * e + i)*:16]);
            let elem2 : bits(16) = op2[(2 * e + i)*:16];
            let elem3 : bits(32) = op3[e*:32];
            result[e*:32] = BFMulAddH_ZA(elem3, elem1, elem2, FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.38 BFMLSL (multiple vectors)

Multi-vector BFloat16 multiply-subtract from single-precision

This instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	1	Zm		0	0	Rv		0	1	0	Zn	0	1	1	0	off2	
																			op		S			

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	1	1	0	off2
																						op		S		

Encoding

```
BFMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = BFNeg(op1[(2 * e + i)*:16]);
            let elem2 : bits(16) = op2[(2 * e + i)*:16];
            let elem3 : bits(32) = op3[e*:32];
            result[e*:32] = BFMulAddH_ZA(elem3, elem1, elem2, FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.39 BFMOP4A (widening)

BFloat16 quarter-tile sum of outer products to single-precision, accumulating

This instruction generates four independent quarter-tile BFloat16 sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of BFloat16 values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of BFloat16 values held in the half-vectors of the second source vectors. Each 32-bit container of the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vectors holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix.

The instruction widens the sub-matrices of BFloat16 values held in the first source vectors to single-precision values and multiplies them by the corresponding widened sub-matrices of BFloat16 values in the second source vectors to single-precision values. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ single-precision sums of outer products are then destructively added to the single-precision destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction follows SME BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
BFMOP4A  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
BFMOP4A  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
BFMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
BFMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 32;
let tilesize : integer{} = 4*dim*dim*32;
let op3 : bits(tilesize) = ZAtile{}(da, 32);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            let sum : bits(32) = op3[tile_idx*:32];

            var erow_0 : bits(16) = op1[(2*row_idx + 0)*:16];
            var erow_1 : bits(16) = op1[(2*row_idx + 1)*:16];
            let ecol_0 : bits(16) = op2[(2*col_idx + 0)*:16];
            let ecol_1 : bits(16) = op2[(2*col_idx + 1)*:16];
            if sub_op then
                let honor_altfp : boolean = FALSE;   // Alternate handling ignored
                erow_0 = BFNeg(erow_0, honor_altfp);
                erow_1 = BFNeg(erow_1, honor_altfp);
            end;

            result[tile_idx*:32] = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.40 BFMOP4A (non-widening)

BFloat16 quarter-tile outer product, accumulating

This instruction generates four independent quarter-tile BFloat16 outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 16-bit element ZA tile.

Each of the quarter-tile outer products is generated by multiplying the $SVL_H \div 2 \times 1$ sub-matrix of BFloat16 values held in the half-vectors of the first source vectors by the $1 \times SVL_H \div 2$ sub-matrix of BFloat16 values held in the half-vectors of the second source vectors.

The resulting quarter-tile $SVL_H \div 2 \times SVL_H \div 2$ BFloat16 outer products are destructively added to the destination ZA tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4A  <ZAda>.H, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Single vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4A  <ZAda>.H, <Zn>.H, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4A  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4A  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 16;
let tilesize : integer{} = 4*dim*dim*16;
let op3 : bits(tilesize) = ZAtile{}(da, 16);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer  = row_base + row;
            let col_idx : integer  = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var elem1 : bits(16) = op1[row_idx*:(16)];
            let elem2 : bits(16) = op2[col_idx*:(16)];
            let elem3 : bits(16) = op3[tile_idx*:(16)];

            if sub_op then
                let honor_altfp : boolean = FALSE;   // Alternate handling ignored
                elem1 = BFNeg(elem1, honor_altfp);
            end;
            result[tile_idx*:(16)] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.41 BFMOP4S (widening)

BFloat16 quarter-tile sum of outer products to single-precision, subtracting

This instruction generates four independent quarter-tile BFloat16 sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of BFloat16 values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of BFloat16 values held in the half-vectors of the second source vectors. Each 32-bit container of the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vectors holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix.

The instruction widens the sub-matrices of BFloat16 values held in the first source vectors to single-precision values and multiplies them by the corresponding widened sub-matrices of BFloat16 values in the second source vectors to single-precision values. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ single-precision sums of outer products are then destructively subtracted from the single-precision destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

This instruction follows SME BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
BFMOP4S  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
BFMOP4S  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
BFMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
BFMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 32;
let tilesize : integer{} = 4*dim*dim*32;
let op3 : bits(tilesize) = ZAtile{}(da, 32);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            let sum : bits(32) = op3[tile_idx*:32];

            var erow_0 : bits(16) = op1[(2*row_idx + 0)*:16];
            var erow_1 : bits(16) = op1[(2*row_idx + 1)*:16];
            let ecol_0 : bits(16) = op2[(2*col_idx + 0)*:16];
            let ecol_1 : bits(16) = op2[(2*col_idx + 1)*:16];
            if sub_op then
                let honor_altfp : boolean = FALSE;   // Alternate handling ignored
                erow_0 = BFNeg(erow_0, honor_altfp);
                erow_1 = BFNeg(erow_1, honor_altfp);
            end;

            result[tile_idx*:32] = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.42 BFMOP4S (non-widening)

BFloat16 quarter-tile outer product, subtracting

This instruction generates four independent quarter-tile BFloat16 outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 16-bit element ZA tile.

Each of the quarter-tile outer products is generated by multiplying the $SVL_H \div 2 \times 1$ sub-matrix of BFloat16 values held in the half-vectors of the first source vectors by the $1 \times SVL_H \div 2$ sub-matrix of BFloat16 values held in the half-vectors of the second source vectors.

The resulting quarter-tile $SVL_H \div 2 \times SVL_H \div 2$ BFloat16 outer products are destructively subtracted from the destination ZA tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4S  <ZAda>.H, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Single vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4S  <ZAda>.H, <Zn>.H, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4S  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	1	0	0	
											M												N					S				ZAda

Encoding

```
BFMOP4S  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 16;
let tilesize : integer{} = 4*dim*dim*16;
let op3 : bits(tilesize) = ZAtile{}(da, 16);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer  = row_base + row;
            let col_idx : integer  = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var elem1 : bits(16) = op1[row_idx*:(16)];
            let elem2 : bits(16) = op2[col_idx*:(16)];
            let elem3 : bits(16) = op3[tile_idx*:(16)];

            if sub_op then
                let honor_altfp : boolean = FALSE;   // Alternate handling ignored
                elem1 = BFNeg(elem1, honor_altfp);
            end;
            result[tile_idx*:(16)] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.43 BFMOPA (widening)

BFloat16 sum of outer products to single-precision, accumulating

This instruction works with a 32-bit element ZA tile.

This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is then destructively added to the single-precision destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME BFloat16 numerical behaviors.

SME

(FEAT_SME)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3 2	1 0
1	0 0	0 0 0 0	1	1 0	0	Zm	Pm	Pn	Zn	0	0 0	ZAda
										S		

Encoding

```
BFMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 32;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        // determine row/col predicates
        let prow_0 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 0, 16));
        let prow_1 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 1, 16));
        let pcol_0 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 0, 16));
        let pcol_1 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 1, 16));

        var sum : bits(32) = operand3[(row*dim+col)*:32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            let erow_0 : bits(16) = (if prow_0 then operand1[(2*row + 0)*:16]
                                     else FPZero{16}('0'));
            let erow_1 : bits(16) = (if prow_1 then operand1[(2*row + 1)*:16]
                                     else FPZero{16}('0'));
            let ecol_0 : bits(16) = (if pcol_0 then operand2[(2*col + 0)*:16]
                                     else FPZero{16}('0'));
            let ecol_1 : bits(16) = (if pcol_1 then operand2[(2*col + 1)*:16]
                                     else FPZero{16}('0'));
            sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());

        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;

ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.44 BFMOPA (non-widening)

BFloat16 outer product, accumulating

This instruction works with a 16-bit element ZA tile.

This instruction generates an outer product of the first source vector and the second source vector. The first source is $SVL_H \times 1$ vector and the second source is $1 \times SVL_H$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, $SVL_H \times SVL_H$, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

SME2

(FEAT_SME_B16B16)

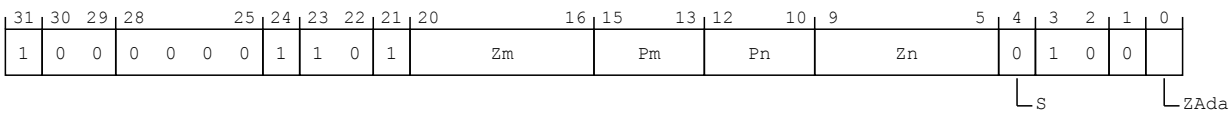

Encoding

```
BFMOPA  <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 16;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(dim*dim*16) = ZAtile{}(da, 16);
var result : bits(dim*dim*16);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        let elem2 : bits(16) = op2[col*:16];
        let elem3 : bits(16) = op3[(row*dim+col)*:16];

        if ActivePredicateElement{PL}(mask1, row, 16) &&
               ActivePredicateElement{PL}(mask2, col, 16) then
            let elem1 : bits(16) = op1[row*:16];
            result[(row*dim+col)*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
        else
            result[(row*dim+col)*:16] = elem3;
        end;
    end;
end;

ZAtile{dim*dim*16}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.45 BFMOPS (widening)

BFloat16 sum of outer products to single-precision, subtracting

This instruction works with a 32-bit element ZA tile.

This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is then destructively subtracted from the single-precision destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME BFloat16 numerical behaviors.

SME

(FEAT_SME)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3 2	1 0
1	0 0	0 0 0 0	1	1 0	0	Zm	Pm	Pn	Zn	1	0 0	ZAda
										S		

Encoding

```
BFMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 32;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        // determine row/col predicates
        let prow_0 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 0, 16));
        let prow_1 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 1, 16));
        let pcol_0 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 0, 16));
        let pcol_1 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 1, 16));

        var sum : bits(32) = operand3[(row*dim+col)*:32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            var erow_0 : bits(16) = (if prow_0 then operand1[(2*row + 0)*:16] else FPZero{16}('0'));
            var erow_1 : bits(16) = (if prow_1 then operand1[(2*row + 1)*:16] else FPZero{16}('0'));
            let ecol_0 : bits(16) = (if pcol_0 then operand2[(2*col + 0)*:16]
                                     else FPZero{16}('0'));
            let ecol_1 : bits(16) = (if pcol_1 then operand2[(2*col + 1)*:16]
                                     else FPZero{16}('0'));
            let honor_altfp : boolean = FALSE;   // Alternate handling ignored
            if prow_0 then erow_0 = BFNeg(erow_0, honor_altfp); end;
            if prow_1 then erow_1 = BFNeg(erow_1, honor_altfp); end;
            sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());

        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;

ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.46 BFMOPS (non-widening)

BFloat16 outer product, subtracting

This instruction works with a 16-bit element ZA tile.

This instruction generates an outer product of the first source vector and the second source vector. The first source is $SVL_H \times 1$ vector and the second source is $1 \times SVL_H$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, $SVL_H \times SVL_H$, is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

SME2

(FEAT_SME_B16B16)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3 2	1	0
1	0 0	0 0 0 0	1	1 0	1	Zm	Pm	Pn	Zn	1	1 0	0	
										S			ZAda

Encoding

```
BFMOPS  <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 16;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(dim*dim*16) = ZAtile{}(da, 16);
var result : bits(dim*dim*16);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        let elem2 : bits(16) = op2[col*:16];
        let elem3 : bits(16) = op3[(row*dim+col)*:16];

        if ActivePredicateElement{PL}(mask1, row, 16) &&
               ActivePredicateElement{PL}(mask2, col, 16) then
            let elem1 : bits(16) = BFNeg(op1[row*:16]);
            result[(row*dim+col)*:16] = BFMulAdd_ZA(elem3, elem1, elem2, FPCR());
        else
            result[(row*dim+col)*:16] = elem3;
        end;
    end;
end;

ZAtile{dim*dim*16}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.47 BFMUL (multiple and single vector)

Multi-vector BFloat16 multiply by vector

This instruction multiplies all the BFloat16 elements of the two or four first source vectors by the corresponding BFloat16 elements of the second source vector, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28			25	24	23	22	21	20			17	16	15					10	9			6	5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	1	1	0	1	0	Zn				0	Zd				0
								size																							

Encoding

```
BFMUL  { <Zd1>.H-<Zd2>.H }, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28			25	24	23	22	21	20			17	16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				1	1	1	1	0	1	0	Zn			0	0	Zd			0	0
								size																							

Encoding

```
BFMUL  { <Zd1>.H-<Zd4>.H }, { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:(16)];
        let element2 : bits(16) = operand2[e*:(16)];
        results[[r]][e*:(16)] = BFMul(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.48 BFMUL (multiple vectors)

Multi-vector BFloat16 multiply

This instruction multiplies all the BFloat16 elements of the two or four first source vectors by the corresponding BFloat16 elements of the two or four second source vectors, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28	25	24	23	22	21	20	17	16	15	10	9	6	5	4	1	0
1	1	0	0 0 0 0		1	0	0	1	Zm		0	1 1 1 0 0 1		Zn		0	Zd		0

size

Encoding

```
BFMUL  { <Zd1>.H-<Zd2>.H }, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	10	9	7	6	5	4	2	1	0
1	1	0	0 0 0 0		1	0	0	1	Zm		0	1	1 1 1 0 0 1		Zn		0	0	Zd		0	0

size

Encoding

```
BFMUL  { <Zd1>.H-<Zd4>.H }, { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element1 : bits(16) = operand1[e*:(16)];
        let element2 : bits(16) = operand2[e*:(16)];
        results[[r]][e*:(16)] = BFMul(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.49 BFSCALE (multiple and single vector)

Multi-vector BFloat16 adjust exponent by vector

This instruction multiplies the BFloat16 elements of the two or four first source vectors by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	0	0	0	1	1	0	0	Zdn					0
								size																								op

Encoding

```
BFSCALE  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7			5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	1	0	0	1	1	0	0	Zdn				0	0
								size																								op

Encoding

```
BFSCALE  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : integer = SInt(operand2[e*:16]);
        results[[r]][e*:16] = BFScale(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.50 BFSCALE (multiple vectors)

Multi-vector BFloat16 adjust exponent

This instruction multiplies the BFloat16 elements of the two or four first source vectors by 2.0 to the power of the signed integer values in the corresponding elements of the two or four second source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction follows SME2 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28			25	24	23	22	21	20		17	16						10	9		7	6	5	4		1	0
1	1	0	0	0	0	0	1	0	0	1	Zm			0	1	0	1	1	0	0	0	1	1	0	0	Zdn			0
								size																opc					o2

Encoding

```
BFSCALE  { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_SVE_BFSCALE)

31	30	29	28			25	24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm			0	0	1	0	1	1	1	0	0	1	1	0	0	Zdn			0	0
								size																	opc						o2

Encoding

```
BFSCALE  { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_SVE_BFSCALE) then
    EndOfDecode(Decode_UNDEF);
end;
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : integer = SInt(operand2[e*:16]);
        results[[r]][e*:16] = BFScale(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.51 BFSUB

Multi-vector BFloat16 subtract from ZA array vectors

This instruction destructively subtracts all elements of the two or four source vectors from the corresponding BFloat16 elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	0	0	Rv		1	1	1	Zm			0	0	1	off3		
									sz																		S			

Encoding

```
BFSUB  ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	1	0	Rv		1	1	1	Zm			0	0	0	1	off3		
									sz																			S		

Encoding

```
BFSUB  ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_B16B16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zm4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = ZAvector{}(vec);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(16) = operand1[e*:16];
        let element2 : bits(16) = operand2[e*:16];
        result[e*:16] = BFSub_ZA(element1, element2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.52 BFTMOPA (widening)

BFloat16 sparse sum of outer products to single-precision, accumulating

This instruction generates BFloat16 sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected 2-in-4 BFloat16 values from each overlapping 32-bit containers of the two $SVL_S \times 2$ sub-matrices in the first source vectors by the two BFloat16 values from the corresponding 32-bit container of the $2 \times SVL_S$ sub-matrix in the second source vector. The two selected elements from each overlapping 32-bit containers of the first source vectors correspond to 2-in-4 elements of rows of two $SVL_S \times 2$ sub-matrices. Each 32-bit container of the second source vector holds 2 elements of columns of a compressed $2 \times SVL_S$ sub-matrix.

The 2-in-4 BFloat16 values from overlapping 32-bit containers of the first source vectors are selected by 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to 32-bit containers of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The instruction widens the selected elements of sub-matrices of BFloat16 values held in the first source vectors to single-precision values and multiplies them by the corresponding widened elements of sub-matrix of BFloat16 values in the second source vector to single-precision values. The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is then destructively added to the single-precision destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction follows SME BFloat16 numerical behaviors.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30 29	28 25	24	23 22	21	20 16	15	14	13	12	11 10	9 6	5 4	3	2	1 0
1	0 0	0 0 0 0	1	0 1	0	Zm	0	0	0	K	Zk	Zn	i2	0	0	ZAda

Encoding

```
BFTMOPA  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 8;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var i : integer = 0;
        var erow : array [[2]] of bits(16);
        erow[[0]] = BFZero{16}('0');
        erow[[1]] = BFZero{16}('0');
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            for e = 0 to 1 do
                if i < 2 && ctrl[(4*col + 2*r + e)*:1] == '1' then
                    erow[[i]] = op1[(2*row + e)*:16];
                    i = i + 1;
                end;
            end;
        end;
        var ecol : array [[2]] of bits(16);
        ecol[[0]] = op2[(2*col + 0)*:16];
        ecol[[1]] = op2[(2*col + 1)*:16];
        let sum : bits(32) = op4[(row*dim+col)*:32];
        result[(row*dim+col)*:32] = BFDotAdd(sum, erow[[0]], erow[[1]],
                                             ecol[[0]], ecol[[1]], FPCR());
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.53 BFTMOPA (non-widening)

BFloat16 sparse outer product, accumulating

This instruction generates BFloat16 outer product by multiplying the 1-in-2 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the destination ZA tile elements.

The outer product is generated by multiplying the selected 1-in-2 BFloat16 value from each overlapping 16-bit containers of the two $SVL_H \times 1$ sub-matrices in the first source vectors by the BFloat16 value from the corresponding 16-bit container of the compressed $1 \times SVL_H$ sub-matrix in the second source vector.

The 1-in-2 BFloat16 value in the first source vectors is selected by 2-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the result. If both bits of the 2-bit control corresponding to the elements of the first source vectors are 1, only the element corresponding to the least significant bit is selected.

The instruction multiplies the selected elements of sub-matrices of BFloat16 values held in the first source vectors by the corresponding elements of sub-matrix of BFloat16 values in the second source vector. The resulting BFloat16 $SVL_H \times SVL_H$ outer product is then destructively added to the BFloat16 destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME2 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP && FEAT_SME_B16B16)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	1	1	Zm				0	0	0	K	Zk		Zn				i2		1	0	0	ZAda

Encoding

```
BFTMOPA  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, <Zm>.H, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) || !IsFeatureImplemented(FEAT_SME_B16B16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 16;
let csize : integer{} = VL DIV 8;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*16) = ZAtile{dim*dim*16}(da, 16);
var result : bits(dim*dim*16);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var i : integer = 0;
        var elem1 : bits(16) = BFZero{16}('0');
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            if i < 1 && ctrl[(2*col + r)*:1] == '1' then
                elem1 = op1[row*:(16)];
                i = i + 1;
            end;
        end;
        let elem2 : bits(16) = op2[col*:(16)];
        let sum : bits(16)   = op4[(row*dim+col)*:(16)];
        result[(row*dim+col)*:(16)] = BFMulAdd_ZA(sum, elem1, elem2, FPCR());
    end;
end;
ZAtile{dim*dim*16}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.54 BFVDOT

Multi-vector BFloat16 vertical dot product by indexed element to single-precision

This instruction calculates the sum-of-products of each vertical pair of BFloat16 values in the corresponding elements of the two first source vectors with the pair of BFloat16 values in the indexed 32-bit group of the corresponding 128-bit segment of the second source vector. The single-precision sum-of-products are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index that selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the ZA operand consists of two ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5		3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				0	1	1	off3		
																			op									opc2			

Encoding

```
BFVDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 2;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to 1 do
    let operand1a : bits(VL) = Z{}(n);
    let operand1b : bits(VL) = Z{}(n+1);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let elt1_a : bits(16) = operand1a[(2 * e + r)*:16];
        let elt1_b : bits(16) = operand1b[(2 * e + r)*:16];
        let elt2_a : bits(16) = operand2[(2 * s + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * s + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.55 BMOPA

Bitwise exclusive NOR population count outer product, accumulating

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source $SVL_S \times 1$ vector and the second source $1 \times SVL_S$ vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting $SVL_S \times SVL_S$ product is then destructively added to the destination tile.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	0	Zm		Pm		Pn		Zn		0	1	0	ZAda	
																			S				

Encoding

```
BMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);

for row = 0 to dim-1 do
    let element1 : bits(esize) = operand1[row*:esize];
    for col = 0 to dim-1 do
        let element2 : bits(esize) = operand2[col*:esize];
        let element3 : bits(esize) = operand3[(row*dim + col)*:esize];
        if (ActivePredicateElement{PL}(mask1, row, esize) &&
              ActivePredicateElement{PL}(mask2, col, esize)) then
            let res : integer = BitCount(NOT(element1 XOR element2));
            result[(row*dim + col)*:esize] = element3 + res;
        else
            result[(row*dim + col)*:esize] = element3;
        end;
    end;
end;
ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.56 BMOPS

Bitwise exclusive NOR population count outer product, subtracting

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source $SVL_S \times 1$ vector and the second source $1 \times SVL_S$ vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting $SVL_S \times SVL_S$ product is then destructively subtracted from the destination tile.

SME2

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	Zm	Pm		Pn		Zn		1	1	0	ZAda	
																	S				

Encoding

```
BMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);

for row = 0 to dim-1 do
    let element1 : bits(esize) = operand1[row*:esize];
    for col = 0 to dim-1 do
        let element2 : bits(esize) = operand2[col*:esize];
        let element3 : bits(esize) = operand3[(row*dim + col)*:esize];
        if (ActivePredicateElement{PL}(mask1, row, esize) &&
              ActivePredicateElement{PL}(mask2, col, esize)) then
            let res : integer = BitCount(NOT(element1 XOR element2));
            result[(row*dim + col)*:esize] = element3 - res;
        else
            result[(row*dim + col)*:esize] = element3;
        end;
    end;
end;
ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.57 F1CVT, F2CVT

Multi-vector 8-bit floating-point convert to half-precision

This instruction converts each 8-bit floating-point element of the source vector to half-precision while downscaling the value, and places the results in the corresponding 16-bit elements of the destination vectors. F1CVT scales the values by $2^{-UInt(FPMR.LSCALE[3:0])}$. F2CVT scales the values by $2^{-UInt(FPMR.LSCALE2[3:0])}$. The 8-bit floating-point encoding format for F1CVT is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for F2CVT is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: F1CVT and F2CVT

F1CVT

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0	0	0	Zn		Zd		0
								opc																		L

Encoding

```
F1CVT  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = FALSE;
```

F2CVT

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0	0	0	Zn		Zd		0
								opc																		L

Encoding

```
F2CVT  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 8;
let operand : bits(VL) = Z{}(n);
var result : bits(2*VL);

for e = 0 to elements-1 do
    let element : bits(8) = operand[e*:8];
    result[e*:16] = FP8ConvertFP(element, issrc2, FPCR(), FPMR());
end;

Z{VL}(d+0) = result[VL-1:0];
Z{VL}(d+1) = result[2*VL-1:VL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.58 F1CVTL, F2CVTL

Multi-vector 8-bit floating-point convert to deinterleaved half-precision

This instruction converts each 8-bit floating-point element of the source vector to half-precision while downscaling the value, and places the two-way deinterleaved results in the corresponding 16-bit elements of the destination vectors. F1CVTL scales the values by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$. F2CVTL scales the values by $2^{-\text{UInt(FPMR.LSCALE2[3:0])}}$. The 8-bit floating-point encoding format for F1CVTL is selected by `FPMR.F8S1`. The 8-bit floating-point encoding format for F2CVTL is selected by `FPMR.F8S2`.

This instruction is unpredicated.

It has encodings from 2 classes: F1CVTL and F2CVTL

F1CVTL

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0	0	0	Zn		Zd		1
								opc																		L

Encoding

```
F1CVTL  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = FALSE;
```

F2CVTL

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	5	4	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0	0	0	Zn		Zd		1
								opc																		L

Encoding

```
F2CVTL  { <Zd1>.H-<Zd2>.H }, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd:: '0');
let issrc2 : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV 16;
let operand : bits(VL) = Z{}(n);
var result1 : bits(VL);
var result2 : bits(VL);

for p = 0 to pairs-1 do
    let element1 : bits(8) = operand[(2*p + 0)*:8];
    let element2 : bits(8) = operand[(2*p + 1)*:8];
    result1[p*:16] = FP8ConvertFP(element1, issrc2, FPCR(), FPMR());
    result2[p*:16] = FP8ConvertFP(element2, issrc2, FPCR(), FPMR());
end;

Z{VL}(d+0) = result1;
Z{VL}(d+1) = result2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.59 FADD

Multi-vector floating-point accumulate to ZA array vectors

This instruction destructively adds all elements of the two or four source vectors to the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors, Two ZA single-vectors of half-precision elements, Four ZA single-vectors, and Four ZA single-vectors of half-precision elements

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv		1	1	1	Zm			0	0	0	off3		
																											S			

Encoding

```
FADD  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16 || FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	Rv		1	1	1	Zm			0	0	0	off3		
									sz																		S			

Encoding

```
FADD  ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) && !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28	25	24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv		1	1	1	Zm	0	0	0	0	off3	
																								S		

Encoding

```
FADD  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Four ZA single-vectors of half-precision elements

(FEAT_SME_F16F16 || FEAT_SME_F8F16)

31	30	29	28	25	24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	1	1	0	1	0	0	1	0	1	0	Rv		1	1	1	Zm	0	0	0	0	off3	
							sz																	S		

Encoding

```
FADD  ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) && !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zm1>

For the “Two ZA single-vectors” and “Two ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” and “Four ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zm4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = ZAvector{}(vec);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPAdd_ZA{esize}(element1, element2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.60 FAMAX

Multi-vector floating-point absolute maximum

This instruction determines the maximum absolute value from floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_FAMINMAX)

31	30	29	28			25	24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	1	0		1	0	Zdn				0
																										opc						o2

Encoding

```
FAMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_FAMINMAX)

31	30	29	28			25	24	23	22	21	20		18	17							10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	0	1	0	1	1	1	0	0	1	0		1	0	Zdn				0	0
																										opc							o2

Encoding

```
FAMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPAbsMax{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.61 FAMIN

Multi-vector floating-point absolute minimum

This instruction determines the minimum absolute value from floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is the Default NaN, with the sign bit set to 0.
- Denormalized inputs and results are never flushed to zero, as if FPCR.{FZ, FZ16, FIZ} are all 0.
- Denormalized inputs never generate an Input Denormal floating-point exception.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_FAMINMAX)

31	30	29	28			25	24	23	22	21	20			17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	1	0	1	0	Zdn				1
																									opc						o2

Encoding

```
FAMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_FAMINMAX)

31	30	29	28			25	24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	0	1	0	1	1	1	0	0	1	0	1	0	Zdn			0	1
																									opc						o2

Encoding

```
FAMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FAMINMAX) then
    EndOfDecode(Decode_UNDEF);
end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPAbsMin{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.62 FCLAMP

Multi-vector floating-point clamp to minimum/maximum number

This instruction clamps each floating-point element in the two or four destination vectors to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively places the clamped results in the corresponding elements of the two or four destination vectors.

Regardless of the value of FPCR.AH, the behavior is as follows for each minimum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4			1	0
1	1	0	0	0	0	0	1	!=00		1	Zm				1	1	0	0	0	0	Zn				Zd				0
								size																					op

Encoding

```
FCLAMP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	!=00		1	Zm				1	1	0	0	1	0	Zn				Zd			0	0
								size																					op

Encoding

```
FCLAMP  { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = Z{}(d+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        let element3 : bits(esize) = operand3[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let maxelement : bits(esize) = FPMaxNum{}(element1, element3, FPCR());
        results[[r]][e*:esize] = FPMinNum{esize}(maxelement, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.63 FCVT (widening)

Multi-vector half-precision convert to single-precision

This instruction converts each element of the source vector from half-precision to single-precision floating-point, and places the results in the double-width destination elements of the destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F16F16 indicates whether this instruction is implemented.

SME2

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20		18	17	16	15			10	9	5	4	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	1	1	0	0	0	Zn	Zd	0
																								L

Encoding

```
FCVT  { <Zd1>.S-<Zd2>.S }, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd::'0');
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let operand : bits(VL) = Z{}(n);
var result : bits(2*VL);

for e = 0 to elements-1 do
    let element : bits(16) = operand[e*:16];
    let res : bits(32) = FPConvertSVE{32, 16}(element, FPCR());
    result[e*:32] = res;
end;

Z{VL}(d+0) = result[VL-1:0];
Z{VL}(d+1) = result[2*VL-1:VL];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.64 FCVT (narrowing, FP16 to FP8)

Multi-vector half-precision convert to 8-bit floating-point

This instruction converts each half-precision element of the two source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE[4:0])}}$, and places the results in the half-width elements of the destination vector. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SME2

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	0	Zn		0	Zd	

op

Encoding

```
FCVT  <Zd>.B, { <Zn1>.H-<Zn2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(16) = operand1[e*:16];
    let element2 : bits(16) = operand2[e*:16];
    result[(0*elements + e)*:8] = FPConvertFP8{8, 16}(element1, FPCR(), FPMR());
    result[(1*elements + e)*:8] = FPConvertFP8{8, 16}(element2, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.65 FCVT (narrowing, FP32 to FP8)

Multi-vector single-precision convert to 8-bit floating-point

This instruction converts each single-precision element of the four source vectors to 8-bit floating-point while scaling the value by $2^{\text{SInt(FPMR.NSCALE)}}$, and places the results in the quarter-width elements of the destination vector. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SME2

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	0	Zn		0	0	Zd	
																									N		

Encoding

```
FCVT  <Zd>.B, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
let operand3 : bits(VL) = Z{}(n+2);
let operand4 : bits(VL) = Z{}(n+3);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let element3 : bits(32) = operand3[e*:32];
    let element4 : bits(32) = operand4[e*:32];
    result[(0*elements + e)*:8] = FPConvertFP8{8, 32}(element1, FPCR(), FPMR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[(1*elements + e)*:8] = FPConvertFP8{8, 32}(element2, FPCR(), FPMR());
    result[(2*elements + e)*:8] = FPConvertFP8{8, 32}(element3, FPCR(), FPMR());
    result[(3*elements + e)*:8] = FPConvertFP8{8, 32}(element4, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.66 FCVT (narrowing, FP32 to FP16)

Multi-vector single-precision convert to half-precision

This instruction converts each element of the two source vectors from single-precision to half-precision floating-point, and places the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9		6	5	4		0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	Zn			0	Zd		
									op																N			

Encoding

```
FCVT  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let res1 : bits(16) = FPConvertSVE{16, 32}(element1, FPCR());
    let res2 : bits(16) = FPConvertSVE{16, 32}(element2, FPCR());
    result[e*:16] = res1;
    result[(elements+e)*:16] = res2;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.67 FCVTL

Multi-vector half-precision convert to deinterleaved single-precision

This instruction converts each element of the source vector from half-precision to single-precision floating-point, and places the two-way deinterleaved results in the double-width destination elements of the destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F16F16 indicates whether this instruction is implemented.

SME2

(FEAT_SME_F16F16)

31	30	29	28				25	24	23	22	21	20			18	17	16	15					10	9			5	4			1	0
1	1	0	0	0	0	0		1	1	0	1	0	0	0		0	0	1	1	1	0	0	0	Zn				Zd				1
																																L

Encoding

```
FCVTL  { <Zd1>.S-<Zd2>.S }, <Zn>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd::'0');
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let pairs : integer = VL DIV 32;
let operand : bits(VL) = Z{}(n);
var result0 : bits(VL);
var result1 : bits(VL);

for p = 0 to pairs-1 do
    let element1 : bits(16) = operand[(2*p+0)*:16];
    let element2 : bits(16) = operand[(2*p+1)*:16];
    let res1 : bits(32) = FPConvertSVE{32, 16}(element1, FPCR());
    let res2 : bits(32) = FPConvertSVE{32, 16}(element2, FPCR());
    result0[p*:32] = res1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result1[p*:32] = res2;
end;

Z{VL}(d+0) = result0;
Z{VL}(d+1) = result1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.68 FCVTN (FP32 to FP8)

Multi-vector single-precision convert to interleaved 8-bit floating-point

This instruction converts each single-precision element of the four source vectors to 8-bit floating-point while scaling the value by $2^{SInt(FPMR.NSCALE)}$, and places the four-way interleaved results in the quarter-width elements of the destination vector. The 8-bit floating-point encoding format is selected by `FPMR.F8D`.

This instruction is unpredicated.

SME2

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	0	Zn		0	1	Zd	
																									N		

Encoding

```
FCVTN  <Zd>.B, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
let operand3 : bits(VL) = Z{}(n+2);
let operand4 : bits(VL) = Z{}(n+3);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let element3 : bits(32) = operand3[e*:32];
    let element4 : bits(32) = operand4[e*:32];
    result[(4*e + 0)*:8] = FPConvertFP8{8, 32}(element1, FPCR(), FPMR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    result[(4*e + 1)*:8] = FPConvertFP8{8, 32}(element2, FPCR(), FPMR());
    result[(4*e + 1)*:8] = FPConvertFP8{8, 32}(element3, FPCR(), FPMR());
    result[(4*e + 1)*:8] = FPConvertFP8{8, 32}(element4, FPCR(), FPMR());

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.69 FCVTN (FP32 to FP16)

Multi-vector single-precision convert to interleaved half-precision

This instruction converts each element of the two source vectors from single-precision to half-precision floating-point, and places the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	Zn		1	Zd	
									op															N		

Encoding

```
FCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var result : bits(VL);

let operand1 : bits(VL) = Z{}(n+0);
let operand2 : bits(VL) = Z{}(n+1);
for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[e*:32];
    let element2 : bits(32) = operand2[e*:32];
    let res1 : bits(16) = FPConvertSVE{16, 32}(element1, FPCR());
    let res2 : bits(16) = FPConvertSVE{16, 32}(element2, FPCR());
    result[(2*e + 0)*:16] = res1;
    result[(2*e + 1)*:16] = res2;

end;
Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.70 FCVTZS

Multi-vector single-precision convert to signed 32-bit integer, rounding toward zero

This instruction converts each element of the two or four source vectors from single-precision to the signed 32-bit integer nearer to zero, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0	0	Zn		0	Zd		0
																								U			

Encoding

```
FCVTZS  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	Zn		0	0	Zd		0	0
																									U				

Encoding

```
FCVTZS  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRounding_ZERO;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FPToFixed{32, 32}(element, 0, unsigned, FPCR(), rounding);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.71 FCVTZU

Multi-vector single-precision convert to unsigned 32-bit integer, rounding toward zero

This instruction converts each element of the two or four source vectors from single-precision to the unsigned 32-bit integer nearer to zero, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0	0	Zn		1	Zd		0
																								U			

Encoding

```
FCVTZU  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	Zn		0	1	Zd		0	0
																									U				

Encoding

```
FCVTZU  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRounding_ZERO;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FPToFixed{32, 32}(element, 0, unsigned, FPCR(), rounding);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.72 FDOT (4-way, multiple and indexed vector)

Multi-vector 8-bit floating-point dot product by indexed element to single-precision

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector. The single-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The 8-bit floating-point groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5			3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				1	1	1		off3		
																			op									opc2				

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5			3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		0	i2		Zn				0	0	0	1		off3		
																			op										opc2				

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let op1 : bits(32) = operand1[e*:(32)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let op2 : bits(32) = operand2[s*:(32)];
        var sum : bits(32) = operand3[e*:(32)];
        sum = FP8DotAddFP{32, 32}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(32)] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.73 FDOT (4-way, multiple and single vector)

Multi-vector 8-bit floating-point dot product by vector to single-precision

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector. The single-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_F8F32)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm	0	Rv	1	0	0	Zn	1	1	off3	

opc

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_F8F32)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm	0	Rv	1	0	0	Zn	1	1	off3	

opc

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let op1 : bits(32) = operand1[e*:(32)];
        let op2 : bits(32) = operand2[e*:(32)];
        var sum : bits(32) = operand3[e*:(32)];
        sum = FP8DotAddFP{32, 32}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(32)] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.74 FDOT (4-way, multiple vectors)

Multi-vector 8-bit floating-point dot product to single-precision

This instruction calculates the fused sum-of-products of a group of four 8-bit floating-point values held in the corresponding 32-bit elements of the two or four first and second source vectors. The single-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12		10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1		Zm			0	0		Rv	1	0	0		Zn			1	1	0		off3	
																										opc					

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12		10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1		Zm			0	1	0		Rv	1	0	0		Zn			0	1	1	0		off3	
																													opc				

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let op1 : bits(32) = operand1[e*:(32)];
        let op2 : bits(32) = operand2[e*:(32)];
        var sum : bits(32) = operand3[e*:(32)];
        sum = FP8DotAddFP{32, 32}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(32)] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.75 FDOT (2-way, multiple and indexed vector, FP8 to FP16)

Multi-vector 8-bit floating-point dot product by indexed element to half-precision

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in the corresponding 16-bit elements of the two or four first source vectors and the indexed 16-bit element of the second source vector. The half-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The 8-bit floating-point pairs within the second source vector are specified using an immediate index that selects the same pair position within each 128-bit vector segment.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_F8F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	1	1	1	0	1	Zm	0	Rv		0	i3h		Zn		1	0	i3l	off3	
															op									

Encoding

```
FDOT  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_F8F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	4	3	2	0
1	1	0	0	0	0	1	0	0	0	1	Zm	1	Rv		1	i3h		Zn		1	0	i3l	off3	

Encoding

```
FDOT  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 8-bit elements within each 128-bit vector segment, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 16;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let op1 : bits(16) = operand1[e*:(16)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let op2 : bits(16) = operand2[s*:(16)];
        var sum : bits(16) = operand3[e*:(16)];
        sum = FP8DotAddFP{16, 16}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(16)] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.76 FDOT (2-way, multiple and single vector, FP8 to FP16)

Multi-vector 8-bit floating-point dot product by vector to half-precision

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in the corresponding 16-bit elements of the two or four first source vectors and the second source vector. The half-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	0	0	Zn				0	1	off3		

opc

Encoding

```
FDOT  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	0	Zn				0	1	off3		

opc

Encoding

```
FDOT  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let op1 : bits(16) = operand1[e*:(16)];
        let op2 : bits(16) = operand2[e*:(16)];
        var sum : bits(16) = operand3[e*:(16)];
        sum = FP8DotAddFP{16, 16}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(16)] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.77 FDOT (2-way, multiple vectors, FP8 to FP16)

Multi-vector 8-bit floating-point dot product to half-precision

This instruction calculates the fused sum-of-products of a pair of 8-bit floating-point values held in the corresponding 16-bit elements of the two or four first and second source vectors. The half-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12		10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1		Zm			0	0		Rv	1	0	0		Zn			1	0	0		off3	
																											opc				

Encoding

```
FDOT  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12		10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1		Zm			0	1	0		Rv	1	0	0		Zn			0	1	0	0		off3	
																													opc				

Encoding

```
FDOT  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let op1 : bits(16) = operand1[e*:(16)];
        let op2 : bits(16) = operand2[e*:(16)];
        var sum : bits(16) = operand3[e*:(16)];
        sum = FP8DotAddFP{16, 16}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(16)] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.78 FDOT (2-way, multiple and indexed vector, FP16 to FP32)

Multi-vector half-precision dot product by indexed element to single-precision

This instruction calculates the fused sum-of-products of a pair of half-precision values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector, without intermediate rounding. The single-precision sum-of-products are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The half-precision pairs within the second source vector are specified using an immediate index that selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5		3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2		Zn				0	0	1	off3		
																			op									opc2			

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5		3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn				0	0	0	1	off3		
																			op										opc2		

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let elt2_a : bits(16) = operand2[(2 * s + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * s + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.79 FDOT (2-way, multiple and single vector, FP16 to FP32)

Multi-vector half-precision dot product by vector to single-precision

This instruction calculates the fused sum-of-products of a pair of half-precision values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector, without intermediate rounding. The single-precision sum-of-products are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	0	0	Zn				0	0	off3		
																										opc				

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	0	0	Zn				0	0	off3		
																										opc				

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
        let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.80 FDOT (2-way, multiple vectors, FP16 to FP32)

Multi-vector half-precision dot product to single-precision

This instruction calculates the fused sum-of-products of a pair of half-precision values held in the corresponding 32-bit elements of the two or four first and second source vectors, without intermediate rounding. The single-precision sum-of-products are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		1	0	0	Zn				0	0	0	off3		

opc

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	0	0	off3		

opc

Encoding

```
FDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
        let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.81 FMAX (multiple and single vector)

Multi-vector floating-point maximum by vector

This instruction determines the maximum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15					11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	!=00		1	0	Zm				1	0	1	0	0		0	0	1	0	0	0	Zdn				0
								size																			op					o2

Encoding

```
FMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15					11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	!=00		1	0	Zm				1	0	1	0	1		0	0	1	0	0	0	Zdn				0	0
								size																			op						o2

Encoding

```
FMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMax{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.82 FMAX (multiple vectors)

Multi-vector floating-point maximum

This instruction determines the maximum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0	1	!=00		1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	Zdn				0
								size																		opc					o2

Encoding

```
FMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	!=00		1	Zm			0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn			0	0
								size																		opc					o2

Encoding

```
FMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMax{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.83 FMAXNM (multiple and single vector)

Multi-vector floating-point maximum number by vector

This instruction determines the maximum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28				25	24	23	22	21	20	19				16	15					11	10	9	8	7	6	5	4				1	0
1	1	0	0	0	0	0		1	!=00		1	0	Zm					1	0	1	0	0		0	0	1	0	0	1	Zdn					0
									size																				op						o2

Encoding

```
FMAXNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28				25	24	23	22	21	20	19				16	15					11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0		1	!=00		1	0	Zm					1	0	1	0	1		0	0	1	0	0	1	Zdn				0	0
									size																				op						o2

Encoding

```
FMAXNM  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMaxNum{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.84 FMAXNM (multiple vectors)

Multi-vector floating-point maximum number

This instruction determines the maximum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28				25	24	23	22	21	20				17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0		1	!=00		1	Zm					0	1	0	1	1	0	0	0	1	0	0	1	Zdn				0
									size																		opc						o2

Encoding

```
FMAXNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28				25	24	23	22	21	20			18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0		1	!=00		1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	1	Zdn			0	0
									size																		opc						o2

Encoding

```
FMAXNM  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMaxNum{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.85 FMIN (multiple and single vector)

Multi-vector floating-point minimum by vector

This instruction determines the minimum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15					11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	!=00		1	0	Zm				1	0	1	0	0		0	0	1	0	0	0	Zdn				1
								size																			op					o2

Encoding

```
FMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15					11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	!=00		1	0	Zm				1	0	1	0	1		0	0	1	0	0	0	Zdn				0	1
								size																			op						o2

Encoding

```
FMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMin{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.86 FMIN (multiple vectors)

Multi-vector floating-point minimum

This instruction determines the minimum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0	1	!=00		1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	Zdn				1
								size																		opc					o2

Encoding

```
FMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	!=00		1	Zm			0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn			0	1
								size																		opc					o2

Encoding

```
FMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMin{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.87 FMINNM (multiple and single vector)

Multi-vector floating-point minimum number by vector

This instruction determines the minimum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	1	0
1	1	0	0 0 0 0		1	!=00		1	0	Zm		1 0 1 0 0		0	0	1	0	0	1	Zdn		1
						size													op			o2

Encoding

```
FMINNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	2	1	0
1	1	0	0 0 0 0		1	!=00		1	0	Zm		1 0 1 0 1		0	0	1	0	0	1	Zdn		0	1
						size													op				o2

Encoding

```
FMINNM  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMinNum{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.88 FMINNM (multiple vectors)

Multi-vector floating-point minimum number

This instruction determines the minimum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28				25	24	23	22	21	20				17	16						10	9			7	6	5	4				1	0
1	1	0	0	0	0	0		1	!=00		1	Zm					0	1	0	1	1	0	0	0	1	0		0	1	Zdn					1
									size																			opc							o2

Encoding

```
FMINNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28				25	24	23	22	21	20			18	17							10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0		1	!=00		1	Zm				0	0	1	0	1	1	1	0	0	1	0		0	1	Zdn				0	1
									size																			opc							o2

Encoding

```
FMINNM  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        results[[r]][e*:esize] = FPMinNum{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.89 FMLA (multiple and indexed vector)

Multi-vector floating-point fused multiply-add by indexed element

This instruction multiplies the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively adds these values without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 6 classes: Two ZA single-vectors of half-precision elements, Two ZA single-vectors of single-precision elements, Two ZA single-vectors of double-precision elements, Four ZA single-vectors of half-precision elements, Four ZA single-vectors of single-precision elements, and Four ZA single-vectors of double-precision elements

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	1	0	0	0	1	Zm		0	Rv		1	i3h		Zn		0	0	i3l	off3	
																				op		S		

Encoding

```
FMLA  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Two ZA single-vectors of single-precision elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				0	0	0	off3		
																			op								S				

Encoding

```
FMLA  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Two ZA single-vectors of double-precision elements (FEAT_SME2 && FEAT_SME_F64F64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		0	0	i1	Zn				0	0	0	off3		
																											S				

Encoding

```
FMLA  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_F64F64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Four ZA single-vectors of half-precision elements (FEAT_SME_F16F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	1	0	0	0	1	Zm	1	Rv		1	i3h		Zn		0	0	0	i3l	off3	
																					op		S		

Encoding

```
FMLA  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Four ZA single-vectors of single-precision elements (FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	1	0	1	0	1	Zm	1	Rv		0	i2		Zn		0	0	0	0	off3	
															op							S			

Encoding

```
FMLA  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.S-<Zn4>.S }, <Zm>.S[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Four ZA single-vectors of double-precision elements (FEAT_SME2 && FEAT_SME_F64F64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	0	1		Zm			1		Rv		0	0	i1		Zn		0	0	0	0		off3		
																				op								S					

Encoding

```
FMLA  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }, <Zm>.D[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_F64F64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors of double-precision elements”, “Two ZA single-vectors of half-precision elements”, and “Two ZA single-vectors of single-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors of double-precision elements”, “Four ZA single-vectors of half-precision elements”, and “Four ZA single-vectors of single-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA single-vectors of half-precision elements” and “Two ZA single-vectors of half-precision elements” variants: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “Four ZA single-vectors of single-precision elements” and “Two ZA single-vectors of single-precision elements” variants: is the element index, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “Four ZA single-vectors of double-precision elements” and “Two ZA single-vectors of double-precision elements” variants: is the element index, in the range 0 to 1, encoded in the “i1” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(esize) = op1[e*:esize];
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let elem2 : bits(esize) = op2[s*:esize];
        let elem3 : bits(esize) = op3[e*:esize];
        result[e*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.90 FMLA (multiple and single vector)

Multi-vector floating-point fused multiply-add by vector

This instruction multiplies the corresponding floating-point elements of the two or four first source vector with the corresponding elements of the second source vector and destructively adds these values without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors, Two ZA single-vectors of half-precision elements, Four ZA single-vectors, and Four ZA single-vectors of half-precision elements

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		1	1	0	Zn				0	0	off3		
																											S			

Encoding

```
FMLA  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	1	1	Zn				0	0	off3		
									sz																		S			

Encoding

```
FMLA  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		1	1	0	Zn				0	0	off3		
																											S			

Encoding

```
FMLA  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Four ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	1	1	Zn				0	0	off3		
									sz																		S			

Encoding

```
FMLA  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(esize) = op1[e*:esize];
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = op3[e*:esize];
        result[e*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.91 FMLA (multiple vectors)

Multi-vector floating-point fused multiply-add

This instruction multiplies the corresponding floating-point elements of the two or four first and second source vectors and destructively adds these values without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors, Two ZA single-vectors of half-precision elements, Four ZA single-vectors, and Four ZA single-vectors of half-precision elements

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12		10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv		1	1	0	Zn				0	0	0	off3		
																												S			

Encoding

```
FMLA  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12		10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		1	0	0	Zn				0	0	1	off3		
									sz																		S				

Encoding

```
FMLA  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20		18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	1	0	Rv		1	1	0	Zn			0	0	0	0	off3		
																												S			

Encoding

```
FMLA  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Four ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20		18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	0	1	off3		
									sz																		S				

Encoding

```
FMLA  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” and “Two ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” and “Four ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” and “Two ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” and “Four ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(esize) = op1[e*:esize];
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = op3[e*:esize];
        result[e*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.92 FMLAL (multiple and indexed vector, FP8 to FP16)

Multi-vector 8-bit floating-point multiply-add by indexed element to half-precision

This instruction widens all 8-bit floating-point elements in the one, two, or four first source vectors and the indexed element of the second source vector to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the overlapping 16-bit half-precision elements of the ZA double-vector groups.

The 8-bit floating-point elements within the second source vector are specified using a 4-bit immediate index that selects the same element position within each 128-bit vector segment.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME_F8F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	1	1	1	0	0	Zm		i4A	Rv		0	i4B		Zn		0	i4C	off3	

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i4A::i4B::i4C);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME_F8F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	0	1	Zm		0	Rv		1	i4h		Zn		1	1	i4l		off2	

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME_F8F16)

31	30 29	28 25	24	23 22	21 20	19 16	15	14 13	12	11 10	9 7	6 4	3 2	1 0
1	1 0	0 0 0 0	1	1 0	0 1	Zm	1	Rv	1	i4h	Zn	0 1 0	i4l	off2

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

For the “One ZA double-vector” variant: is the element index, in the range 0 to 15, encoded in the “i4A:i4B:i4C” fields.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 16;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let elem1 : bits(8) = op1[(2 * e + i)*:(8)];
            let elem2 : bits(8) = op2[s*:(8)];
            let elem3 : bits(16) = op3[e*:(16)];
            result[e*:(16)] = FP8MulAddFP{16}(elem3, elem1, elem2,
                                                FPCR(), FPMR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.93 FMLAL (multiple and single vector, FP8 to FP16)

Multi-vector 8-bit floating-point multiply-add by vector to half-precision

This instruction widens all 8-bit floating-point elements in the one, two, or four first source vectors and the second source vector to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the overlapping 16-bit half-precision elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	1	Zn				0	0	off3		

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>], <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	1	0	Zn				0	0	1	off2	
																									op		S	o2		

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				0	0	1	off2	
																										op	S	o2		

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(8) = op1[(2 * e + i)*:(8)];
            let elem2 : bits(8) = op2[(2 * e + i)*:(8)];
            let elem3 : bits(16) = op3[e*:(16)];
            result[e*:(16)] = FP8MulAddFP{16}(elem3, elem1, elem2,
                                                  FPCR(), FPMR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.94 FMLAL (multiple vectors, FP8 to FP16)

Multi-vector 8-bit floating-point multiply-add to half-precision

This instruction widens all 8-bit floating-point elements in the two or four first and second source vectors to half-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$ before being destructively added without intermediate rounding to the overlapping 16-bit half-precision elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12			10	9			6	5	4			2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		0	1	0		Zn				1	0	0	0	off2		

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12			10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	1	0	Rv		0	1	0		Zn				0	1	0	0	0	off2		

Encoding

```
FMLAL  ZA.H[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(8) = op1[(2 * e + i)*:(8)];
            let elem2 : bits(8) = op2[(2 * e + i)*:(8)];
            let elem3 : bits(16) = op3[e*:(16)];
            result[e*:(16)] = FP8MulAddFP{16}(elem3, elem1, elem2,
                                                FPCR(), FPMR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.95 FMLAL (multiple and indexed vector, FP16 to FP32)

Multi-vector half-precision multiply-add by indexed element to single-precision

This instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index that selects the same element position within each 128-bit vector segment.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	1	1	0	0	0	Zm	i3h	Rv		1	i3l		Zn		0	0	off3	
																				op	S		

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	0	0	1	Zm	0	Rv		1	i3h		Zn		0	0	0	i3l	off2	
																				op		S			

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1		Zm			1		Rv		1		i3h			Zn			0	0	i3l	off2
																											op		S		

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let elem1 : bits(16) = op1[(2 * e + i)*:(16)];
            let elem2 : bits(16) = op2[s*:(16)];
            let elem3 : bits(32) = op3[e*:(32)];
            result[e*:(32)] = FPMulAddH_ZA(elem3, elem1, elem2,
                                                   FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.96 FMLAL (multiple and single vector, FP16 to FP32)

Multi-vector half-precision multiply-add by vector to single-precision

This instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	1	0	0	1	0	Zm		0	Rv		0	1	1	Zn	0	0	off3	

op, S

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	0	Zm		0	Rv		0	1	0	Zn	0	0	0	off2	

op, S, o2

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				0	0	0	off2	

op, S, o2

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = op1[(2 * e + i)*:(16)];
            let elem2 : bits(16) = op2[(2 * e + i)*:(16)];
            let elem3 : bits(32) = op3[e*:(32)];
            result[e*:(32)] = FPMulAddH_ZA(elem3, elem1, elem2,
                                                   FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.97 FMLAL (multiple vectors, FP16 to FP32)

Multi-vector half-precision multiply-add to single-precision

This instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	1	Zm		0	0	Rv		0	1	0	Zn	0	0	0	0	off2	

op S

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	0	0	0	off2

op S

Encoding

```
FMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = op1[(2 * e + i)*:(16)];
            let elem2 : bits(16) = op2[(2 * e + i)*:(16)];
            let elem3 : bits(32) = op3[e*:(32)];
            result[e*:(32)] = FPMulAddH_ZA(elem3, elem1, elem2,
                                                   FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.98 FMLALL (multiple and indexed vector)

Multi-vector 8-bit floating-point multiply-add by indexed element to single-precision

This instruction widens all 8-bit floating-point elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$ before being destructively added without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA quad-vector groups.

The 8-bit floating-point elements within the second source vector are specified using a 4-bit immediate index that selects the same element position within each 128-bit vector segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME_F8F32)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	2	1	0
1	1	0	0	0	0	1	0	1	0	0	Zm	i4h	Rv		i4l		Zn		0	0	0	off2

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME_F8F32)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	0	0	1	Zm	0	Rv		0	i4h		Zn		1	0	0	i4l		o1

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6		4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		0	i4h		Zn			1	0	0	0	i4l		o1

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : bits(8) = operand1[(4 * e + i)*:8];
            let element2 : bits(8) = operand2[s*:8];
            let element3 : bits(32) = operand3[e*:32];
            result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.99 FMLALL (multiple and single vector)

Multi-vector 8-bit floating-point multiply-add by vector to single-precision

This instruction widens all 8-bit floating-point elements in the one, two, or four first source vectors and the second source vector to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-UInt(FPMR.LSCALE)}$ before being destructively added without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	0	1	Zn				0	0	0	off2	

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	0	0	Zn				0	0	0	1	o1

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors (FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	0	0	Zn				0	0	0	1	o1

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : bits(8) = operand1[(4 * e + i)*:8];
            let element2 : bits(8) = operand2[(4 * e + i)*:8];
            let element3 : bits(32) = operand3[e*:32];
            result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.100 FMLALL (multiple vectors)

Multi-vector 8-bit floating-point multiply-add to single-precision

This instruction widens all 8-bit floating-point elements in the two or four first and second source vectors to single-precision format and multiplies the corresponding elements. The intermediate products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE)}}$ before being destructively added without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12			10	9			6	5	4			2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		0	0	0		Zn				1	0	0	0		0	o1

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12			10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	1	0	Rv		0	0	0		Zn				0	1	0	0	0	0	o1

Encoding

```
FMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : bits(8) = operand1[(4 * e + i)*:8];
            let element2 : bits(8) = operand2[(4 * e + i)*:8];
            let element3 : bits(32) = operand3[e*:32];
            result[e*:32] = FP8MulAddFP{32}(element3, element1, element2, FPCR(), FPMR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.101 FMLS (multiple and indexed vector)

Multi-vector floating-point fused multiply-subtract by indexed element

This instruction multiplies the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively subtracts these values without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 6 classes: Two ZA single-vectors of half-precision elements, Two ZA single-vectors of single-precision elements, Two ZA single-vectors of double-precision elements, Four ZA single-vectors of half-precision elements, Four ZA single-vectors of single-precision elements, and Four ZA single-vectors of double-precision elements

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	1	0	0	0	1	Zm		0	Rv		1	i3h		Zn		0	1	i3l	off3	
																				op		S		

Encoding

```
FMLS  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Two ZA single-vectors of single-precision elements (FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	1	0	1	0	1	Zm	0	Rv		0	i2		Zn		0	1	0	off3	
															op						S			

Encoding

```
FMLS  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Two ZA single-vectors of double-precision elements (FEAT_SME2 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	1	1	1	0	1	Zm	0	Rv		0	0	i1	Zn		0	1	0	off3	
																					S			

Encoding

```
FMLS  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_F64F64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Four ZA single-vectors of half-precision elements (FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		1	i3h		Zn			0	0	1	i3l	off3		
																										op		S			

Encoding

```
FMLS  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Four ZA single-vectors of single-precision elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		0	i2		Zn			0	0	1	0	off3		
																			op								S				

Encoding

```
FMLS  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.S-<Zn4>.S }, <Zm>.S[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Four ZA single-vectors of double-precision elements (FEAT_SME2 && FEAT_SME_F64F64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	0	1		Zm			1		Rv		0	0	i1		Zn		0	0	1	0		off3		
																				op								S					

Encoding

```
FMLS  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }, <Zm>.D[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_F64F64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors of double-precision elements”, “Two ZA single-vectors of half-precision elements”, and “Two ZA single-vectors of single-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors of double-precision elements”, “Four ZA single-vectors of half-precision elements”, and “Four ZA single-vectors of single-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA single-vectors of half-precision elements” and “Two ZA single-vectors of half-precision elements” variants: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the “Four ZA single-vectors of single-precision elements” and “Two ZA single-vectors of single-precision elements” variants: is the element index, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “Four ZA single-vectors of double-precision elements” and “Two ZA single-vectors of double-precision elements” variants: is the element index, in the range 0 to 1, encoded in the “i1” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(esize) = FPNeg{}(op1[e*:esize], FPCR());
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let elem2 : bits(esize) = op2[s*:esize];
        let elem3 : bits(esize) = op3[e*:esize];
        result[e*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.102 FMLS (multiple and single vector)

Multi-vector floating-point fused multiply-subtract by vector

This instruction multiplies the corresponding floating-point elements of the two or four first source vector with the corresponding elements of the second source vector and destructively subtracts these values without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors, Two ZA single-vectors of half-precision elements, Four ZA single-vectors, and Four ZA single-vectors of half-precision elements

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		1	1	0	Zn				0	1	off3		
																											S			

Encoding

```
FMLS  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		1	1	1	Zn				0	1	off3		
									sz																		S			

Encoding

```
FMLS  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		1	1	0	Zn				0	1	off3		
																											S			

Encoding

```
FMLS  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Four ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		1	1	1	Zn				0	1	off3		
									sz																		S			

Encoding

```
FMLS  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(esize) = FPNeg{}(op1[e*:esize], FPCR());
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = op3[e*:esize];
        result[e*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.103 FMLS (multiple vectors)

Multi-vector floating-point fused multiply-subtract

This instruction multiplies the corresponding floating-point elements of the two or four first and second source vectors and destructively subtracts these values without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors, Two ZA single-vectors of half-precision elements, Four ZA single-vectors, and Four ZA single-vectors of half-precision elements

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12		10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv		1	1	0	Zn				0	0	1	off3		
																												S			

Encoding

```
FMLS  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12		10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		1	0	0	Zn				0	1	1	off3		
									sz																		S				

Encoding

```
FMLS  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20		18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	1	0	Rv		1	1	0	Zn			0	0	0	1	off3		
																												S			

Encoding

```
FMLS  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Four ZA single-vectors of half-precision elements

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20		18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	1	1	off3		
									sz																		S				

Encoding

```
FMLS  ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” and “Two ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” and “Four ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” and “Two ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” and “Four ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    let op3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let elem1 : bits(esize) = FPNeg{}(op1[e*:esize], FPCR());
        let elem2 : bits(esize) = op2[e*:esize];
        let elem3 : bits(esize) = op3[e*:esize];
        result[e*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.104 FMLSL (multiple and indexed vector)

Multi-vector half-precision multiply-subtract by indexed element from single-precision

This instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index that selects the same element position within each 128-bit vector segment.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	0	1	1	0	0	0	Zm	i3h	Rv		1	i3l		Zn		0	1	off3	
																				op	S		

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	0	0	1	Zm	0	Rv		1	i3h		Zn		0	0	1	i3l	off2	
																				op		S			

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				1	Rv		1	i3h		Zn			0	0	0	1	i3l	off2	
																											op	S			

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let elem1 : bits(16) = FPNeg{16}(op1[(2 * e + i)*:(16)], FPCR());
            let elem2 : bits(16) = op2[s*:(16)];
            let elem3 : bits(32) = op3[e*:(32)];
            result[e*:(32)] = FPMulAddH_ZA(elem3, elem1, elem2,
                                                   FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.105 FMLSL (multiple and single vector)

Multi-vector half-precision multiply-subtract by vector from single-precision

This instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2		0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	1	1	Zn				0	1	off3		
																										op	S			

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				0	Rv		0	1	0	Zn				0	1	0	off2	
																										op	S	o2		

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	1	0	Zn				0	1	0	off2	

op, S, o2

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}((n+r) MOD 32);
    let op2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = FPNeg{16}(op1[(2 * e + i)*:(16)], FPCR());
            let elem2 : bits(16) = op2[(2 * e + i)*:(16)];
            let elem3 : bits(32) = op3[e*:(32)];
            result[e*:(32)] = FPMulAddH_ZA(elem3, elem1, elem2,
                                                FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.106 FMLSL (multiple vectors)

Multi-vector half-precision multiply-subtract from single-precision

This instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		0	1	0	Zn				0	0	1	0	off2	
																											op	S			

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	0	1	0	off2	
																											op	S			

Encoding

```
FMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let op1 : bits(VL) = Z{}(n+r);
    let op2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let op3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let elem1 : bits(16) = FPNeg{16}(op1[(2 * e + i)*:(16)], FPCR());
            let elem2 : bits(16) = op2[(2 * e + i)*:(16)];
            let elem3 : bits(32) = op3[e*:(32)];
            result[e*:(32)] = FPMulAddH_ZA(elem3, elem1, elem2,
                                                FPCR());
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.107 FMOP4A (widening, 2-way, FP8 to FP16)

8-bit floating-point quarter-tile sum of outer products to half-precision, accumulating

This instruction generates four independent quarter-tile 8-bit floating-point sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 16-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_H \div 2 \times 2$ sub-matrix of 8-bit floating-point values held in the half-vectors of the first source vectors by the $2 \times SVL_H \div 2$ sub-matrix of 8-bit floating-point values held in the half-vectors of the second source vectors. Each 16-bit container of the first source vectors holds 2 elements of each row of a $SVL_H \div 2 \times 2$ sub-matrix. Similarly, each 16-bit container of the second source vectors holds 2 elements of each column of a $2 \times SVL_H \div 2$ sub-matrix.

This instruction widens the sub-matrices of 8-bit floating-point values held in the first source vectors to half-precision values and multiplies them by the corresponding widened sub-matrices of 8-bit floating-point values in the second source vectors to half-precision values. The resulting quarter-tile $SVL_H \div 2 \times SVL_H \div 2$ half-precision sums of outer products are scaled by $2^{-\text{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added to the half-precision destination tile. This is equivalent to performing a downscaled 2-way dot product and accumulate to each of the destination tile elements.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	1	0	0	
											M												N									ZAda

Encoding

```
FMOP4A  <ZAda>.H, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;

let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
```

Single vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	1	0	0	
											M												N									ZAda

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMOP4A  <ZAda>.H, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;

let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
```

Multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	0	0	
											M												N									ZAda

Encoding

```
FMOP4A  <ZAda>.H, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;

let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
```

Multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	0	0	
											M												N									ZAda

Encoding

```
FMOP4A  <ZAda>.H, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;

let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 16;
let tilesize : integer{} = 4*dim*dim*16;
let op3 : bits(tilesize) = ZAtile{}(da, 16);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(16) = op3[tile_idx*:(16)];

            var rowop : bits(16);
            var colop : bits(16);
            for i = 0 to 1 do
                rowop[i*:8] = op1[(2*row_idx + i)*:8];
                colop[i*:8] = op2[(2*col_idx + i)*:8];
            end;
            sum = FP8DotAddFP{16, 16}(sum, rowop, colop, FPCR(), FPMR());
            result[tile_idx*:(16)] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.108 FMOP4A (widening, 4-way)

8-bit floating-point quarter-tile sum of outer products to single-precision, accumulating

This instruction generates four independent quarter-tile 8-bit floating-point sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit floating-point values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit floating-point values held in the half-vectors of the second source vectors. Each 32-bit container of the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the second source vectors holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

This instruction widens the sub-matrices of 8-bit floating-point values held in the first source vectors to single-precision values and multiplies them by the corresponding widened sub-matrices of 8-bit floating-point values in the second source vectors to single-precision values. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ single-precision sums of outer products are scaled by $2^{-UInt(FPMR.LSCALE)}$, before being destructively added to the single-precision destination tile. This is equivalent to performing a downscaled 4-way dot product and accumulate to each of the destination tile elements.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N									

Encoding

```
FMOP4A  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F32) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
```

Single vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N									

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMOP4A  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F32) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
```

Multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F32)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	Zm		0	0	0	0	1	Zn		0	0	0	0	ZAda	
									M							N								

Encoding

```
FMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F32) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
```

Multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F8F32)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	1	Zm		0	0	0	0	1	Zn		0	0	0	0	ZAda	
									M							N								

Encoding

```
FMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F8F32) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 32;
let tilesize : integer{} = 4*dim*dim*32;
let op3 : bits(tilesize) = ZAtile{}(da, 32);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(32) = op3[tile_idx*:(32)];

            var rowop : bits(32);
            var colop : bits(32);
            for i = 0 to 3 do
                rowop[i*:8] = op1[(4*row_idx + i)*:8];
                colop[i*:8] = op2[(4*col_idx + i)*:8];
            end;
            sum = FP8DotAddFP{32, 32}(sum, rowop, colop, FPCR(), FPMR());
            result[tile_idx*:(32)] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.109 FMOP4A (widening, 2-way, FP16 to FP32)

Half-precision quarter-tile sum of outer products to single-precision, accumulating

This instruction generates four independent quarter-tile half-precision sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of half-precision values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of half-precision values held in the half-vectors of the second source vectors. Each 32-bit container of the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vectors holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix.

The instruction widens the sub-matrices of half-precision values held in the first source vectors to single-precision values and multiplies them by the corresponding widened sub-matrices of half-precision values in the second source vectors to single-precision values. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ single-precision sums of outer products are then destructively added to the single-precision destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4A  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMOP4A  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 32;
let tilesize : integer{} = 4*dim*dim*32;
let op3 : bits(tilesize) = ZAtile{}(da, 32);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            let sum : bits(32) = op3[tile_idx*:32];

            var erow_0 : bits(16) = op1[(2*row_idx + 0)*:16];
            var erow_1 : bits(16) = op1[(2*row_idx + 1)*:16];
            let ecol_0 : bits(16) = op2[(2*col_idx + 0)*:16];
            let ecol_1 : bits(16) = op2[(2*col_idx + 1)*:16];
            if sub_op then
                erow_0 = FPNeg{16}(erow_0, FPCR());
                erow_1 = FPNeg{16}(erow_1, FPCR());
            end;

            result[tile_idx*:32] = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.110 FMOP4A (non-widening)

Floating-point quarter-tile outer product, accumulating

This instruction generates four independent quarter-tile floating-point outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 16-bit, 32-bit, or 64-bit element ZA tile.

In case of the half-precision variant, each of the quarter-tile outer products is generated by multiplying the $SVL_H \div 2 \times 1$ sub-matrix of half-precision values held in the half-vectors of the first source vectors by the $1 \times SVL_H \div 2$ sub-matrix of half-precision values held in the half-vectors of the second source vectors. In case of the single-precision variant, each of the quarter-tile outer products is generated by multiplying the $SVL_S \div 2 \times 1$ sub-matrix of single-precision values held in the half-vectors of the first source vectors by the $1 \times SVL_S \div 2$ sub-matrix of single-precision values held in the half-vectors of the second source vectors. In case of the double-precision variant, each of the quarter-tile outer products is generated by multiplying the $SVL_D \div 2 \times 1$ sub-matrix of double-precision values held in the half-vectors of the first source vectors by the $1 \times SVL_D \div 2$ sub-matrix of double-precision values held in the half-vectors of the second source vectors.

The resulting quarter-tile $SVL_H \div 2 \times SVL_H \div 2$ half-precision outer products in case of the half-precision variant, $SVL_S \div 2 \times SVL_S \div 2$ single-precision outer products in case of the single-precision variant, or $SVL_D \div 2 \times SVL_D \div 2$ double-precision outer products in case of the double-precision variant are destructively added to the destination ZA tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 12 classes: Half-precision, single and multiple vectors, Half-precision, single vectors, Half-precision, multiple and single vectors, Half-precision, multiple vectors, Single-precision, single and multiple vectors, Single-precision, single vectors, Single-precision, multiple and single vectors, Single-precision, multiple vectors, Double-precision, single and multiple vectors, Double-precision, single vectors, Double-precision, multiple and single vectors, and Double-precision, multiple vectors

Half-precision, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		1	0	0	0	1		Zm			0	0	0	0	0	0	0	0	0		Zn			0	0	1	0	0	
												M													N						S				ZAda

Encoding

```
FMOP4A  <ZAda>.H, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Half-precision, single vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
FMOP4A  <ZAda>.H, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Half-precision, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
FMOP4A  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Half-precision, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	0	0	
											M												N					S				ZAda

Encoding

```
FMOP4A  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Single-precision, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4A  <ZAda>.S, <Zn>.S, { <Zm1>.S-<Zm2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision, single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	0	Zm		0	0	0 0 0 0 0		0	Zn		0	0	0	0	ZAda	
									M							N				S				

Encoding

```
FMOP4A  <ZAda>.S, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Single-precision, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	0	Zm		0	0	0 0 0 0 0		1	Zn		0	0	0	0	ZAda	
									M							N				S				

Encoding

```
FMOP4A  <ZAda>.S, { <Zn1>.S-<Zn2>.S }, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision, multiple vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	1	Zm		0	0	0 0 0 0 0		1	Zn		0	0	0	0	ZAda	
									M							N				S				

Encoding

```
FMOP4A  <ZAda>.S, { <Zn1>.S-<Zn2>.S }, { <Zm1>.S-<Zm2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Double-precision, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0 0 0 0		0	1	1	0	1	Zm		0 0 0 0 0 0 0		0	Zn		0	0	1	ZAda	
									M					N				S			

Encoding

```
FMOP4A  <ZAda>.D, <Zn>.D, { <Zm1>.D-<Zm2>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision, single vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0 0 0 0		0	1	1	0	0	Zm		0 0 0 0 0 0 0		0	Zn		0	0	1	ZAda	
									M					N				S			

Encoding

```
FMOP4A  <ZAda>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Double-precision, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0 0 0 0		0	1	1	0	0	Zm		0 0 0 0 0 0 0		1	Zn		0	0	1	ZAda	
									M					N				S			

Encoding

```
FMOP4A  <ZAda>.D, { <Zn1>.D-<Zn2>.D }, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28				25	24	23	22	21	20	19			17	16							10	9	8			6	5	4	3	2			0
1	0	0	0	0	0	0		0	1	1	0	1	Zm				0	0	0	0	0	0	0		1	Zn				0	0	1	ZAda			
												M													N						S					

Encoding

```
FMOP4A  <ZAda>.D, { <Zn1>.D-<Zn2>.D }, { <Zm1>.D-<Zm2>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “Half-precision, multiple and single vectors”, “Half-precision, multiple vectors”, “Half-precision, single and multiple vectors”, and “Half-precision, single vectors” variants: is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

For the “Single-precision, multiple and single vectors”, “Single-precision, multiple vectors”, “Single-precision, single and multiple vectors”, and “Single-precision, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “Double-precision, multiple and single vectors”, “Double-precision, multiple vectors”, “Double-precision, single and multiple vectors”, and “Double-precision, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer  = row_base + row;
            let col_idx : integer  = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var elem1 : bits(esize) = op1[row_idx*:(esize)];
            let elem2 : bits(esize) = op2[col_idx*:(esize)];
            let elem3 : bits(esize) = op3[tile_idx*:(esize)];

            if sub_op then elem1 = FPNeg{esize}(elem1, FPCR()); end;
            result[tile_idx*:(esize)] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.111 FMOP4S (widening)

Half-precision quarter-tile sum of outer products to single-precision, subtracting

This instruction generates four independent quarter-tile half-precision sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of half-precision values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of half-precision values held in the half-vectors of the second source vectors. Each 32-bit container of the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vectors holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix.

The instruction widens the sub-matrices of half-precision values held in the first source vectors to single-precision values and multiplies them by the corresponding widened sub-matrices of half-precision values in the second source vectors to single-precision values. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ single-precision sums of outer products are then destructively subtracted from the single-precision destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4S  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMOP4S  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV 32;
let tilesize : integer{} = 4*dim*dim*32;
let op3 : bits(tilesize) = ZAtile{}(da, 32);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            let sum : bits(32) = op3[tile_idx*:32];

            var erow_0 : bits(16) = op1[(2*row_idx + 0)*:16];
            var erow_1 : bits(16) = op1[(2*row_idx + 1)*:16];
            let ecol_0 : bits(16) = op2[(2*col_idx + 0)*:16];
            let ecol_1 : bits(16) = op2[(2*col_idx + 1)*:16];
            if sub_op then
                erow_0 = FPNeg{16}(erow_0, FPCR());
                erow_1 = FPNeg{16}(erow_1, FPCR());
            end;

            result[tile_idx*:32] = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.112 FMOP4S (non-widening)

Floating-point quarter-tile outer product, subtracting

This instruction generates four independent quarter-tile floating-point outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 16-bit, 32-bit, or 64-bit element ZA tile.

In case of the half-precision variant, each of the quarter-tile outer products is generated by multiplying the $SVL_H \div 2 \times 1$ sub-matrix of half-precision values held in the half-vectors of the first source vectors by the $1 \times SVL_H \div 2$ sub-matrix of half-precision values held in the half-vectors of the second source vectors. In case of the single-precision variant, each of the quarter-tile outer products is generated by multiplying the $SVL_S \div 2 \times 1$ sub-matrix of single-precision values held in the half-vectors of the first source vectors by the $1 \times SVL_S \div 2$ sub-matrix of single-precision values held in the half-vectors of the second source vectors. In case of the double-precision variant, each of the quarter-tile outer products is generated by multiplying the $SVL_D \div 2 \times 1$ sub-matrix of double-precision values held in the half-vectors of the first source vectors by the $1 \times SVL_D \div 2$ sub-matrix of double-precision values held in the half-vectors of the second source vectors.

The resulting quarter-tile $SVL_H \div 2 \times SVL_H \div 2$ half-precision outer products in case of the half-precision variant, $SVL_S \div 2 \times SVL_S \div 2$ single-precision outer products in case of the single-precision variant, or $SVL_D \div 2 \times SVL_D \div 2$ double-precision outer products in case of the double-precision variant are destructively subtracted from the destination ZA tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 12 classes: Half-precision, single and multiple vectors, Half-precision, single vectors, Half-precision, multiple and single vectors, Half-precision, multiple vectors, Single-precision, single and multiple vectors, Single-precision, single vectors, Single-precision, multiple and single vectors, Single-precision, multiple vectors, Double-precision, single and multiple vectors, Double-precision, single vectors, Double-precision, multiple and single vectors, and Double-precision, multiple vectors

Half-precision, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0		0	1	Zm				0	0	0	0	0	0	0	0	0	Zn				0	1	1	0	0	
												M													N						S				ZAda

Encoding

```
FMOP4S  <ZAda>.H, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Half-precision, single vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			17	16	15	14						10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm				0	0	0	0	0	0	0	0		0	Zn				0	1	1	0	0	
											M														N						S				ZAda

Encoding

```
FMOP4S  <ZAda>.H, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Half-precision, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20	19			17	16	15	14						10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm				0	0	0	0	0	0	0	0		1	Zn				0	1	1	0	0	
											M														N						S				ZAda

Encoding

```
FMOP4S  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Half-precision, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F16F16)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		1	0	0	0	1	Zm		0	0	0 0 0 0 0		1	Zn		0	1	1	0	0	
									M							N				S				ZAda

Encoding

```
FMOP4S  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 16;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Single-precision, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	1	Zm		0	0	0 0 0 0 0		0	Zn		0	1	0	0	ZAda	
									M							N				S				

Encoding

```
FMOP4S  <ZAda>.S, <Zn>.S, { <Zm1>.S-<Zm2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision, single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4S  <ZAda>.S, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Single-precision, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	0	0	ZAda	
											M												N					S				

Encoding

```
FMOP4S  <ZAda>.S, { <Zn1>.S-<Zn2>.S }, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision, multiple vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	1	Zm		0	0	0 0 0 0 0		1	Zn		0	1	0	0	ZAda	
									M							N				S				

Encoding

```
FMOP4S  <ZAda>.S, { <Zn1>.S-<Zn2>.S }, { <Zm1>.S-<Zm2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Double-precision, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0 0 0 0		0	1	1	0	1	Zm		0 0 0 0 0 0 0		0	Zn		0	1	1	ZAda	
									M					N				S			

Encoding

```
FMOP4S  <ZAda>.D, <Zn>.D, { <Zm1>.D-<Zm2>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision, single vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0 0 0 0		0	1	1	0	0	Zm		0 0 0 0 0 0 0		0	Zn		0	1	1	ZAda	
									M					N				S			

Encoding

```
FMOP4S  <ZAda>.D, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Double-precision, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	0	0 0 0 0		0	1	1	0	0	Zm		0 0 0 0 0 0 0		1	Zn		0	1	1	ZAda	
									M					N				S			

Encoding

```
FMOP4S  <ZAda>.D, { <Zn1>.D-<Zn2>.D }, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_F64F64)

31	30	29	28				25	24	23	22	21	20	19			17	16							10	9	8			6	5	4	3	2			0
1	0	0	0	0	0	0		0	1	1	0	1	Zm				0	0	0	0	0	0	0		1	Zn				0	1	1	ZAda			
												M													N						S					

Encoding

```
FMOP4S  <ZAda>.D, { <Zn1>.D-<Zn2>.D }, { <Zm1>.D-<Zm2>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_F64F64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “Half-precision, multiple and single vectors”, “Half-precision, multiple vectors”, “Half-precision, single and multiple vectors”, and “Half-precision, single vectors” variants: is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

For the “Single-precision, multiple and single vectors”, “Single-precision, multiple vectors”, “Single-precision, single and multiple vectors”, and “Single-precision, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “Double-precision, multiple and single vectors”, “Double-precision, multiple vectors”, “Double-precision, single and multiple vectors”, and “Double-precision, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer  = row_base + row;
            let col_idx : integer  = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var elem1 : bits(esize) = op1[row_idx*:(esize)];
            let elem2 : bits(esize) = op2[col_idx*:(esize)];
            let elem3 : bits(esize) = op3[tile_idx*:(esize)];

            if sub_op then elem1 = FPNeg{esize}(elem1, FPCR()); end;
            result[tile_idx*:(esize)] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.113 FMOPA (widening, 2-way, FP8 to FP16)

8-bit floating-point sum of outer products to half-precision, accumulating

This instruction works with a 16-bit element ZA tile.

This instruction widens the $SVL_H \times 2$ sub-matrix of 8-bit floating-point values held in the first source vector to half-precision values and multiplies it by the widened $2 \times SVL_H$ sub-matrix of 8-bit floating-point values in the second source vector to half-precision values.

Each source vector is independently predicated by a corresponding governing predicate. When a 8-bit source element is Inactive it is treated as having the value +0.0, but if both groups of source vector elements that correspond to a 16-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_H \times SVL_H$ half-precision sum of outer products are scaled by $2^{-UInt(FPMR.LSCALE[3:0])}$, before being destructively added to the half-precision destination tile. This is equivalent to performing a downscaled 2-way dot product and accumulate to each of the destination tile elements.

Each 16-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_H \times 2$ sub-matrix. Similarly, each 16-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_H$ sub-matrix.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

SME2

(FEAT_SME_F8F16)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3 2	1	0
1	0 0	0 0 0 0	0	1 0	1	Zm	Pm	Pn	Zn	0	1 0	0	ZAda

Encoding

```
FMOPA  <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 16;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*16) = ZAtile{}(da, 16);
var result : bits(dim*dim*16);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var prow : array [[2]] of boolean;
        var pcol : array [[2]] of boolean;
        var any_active : boolean = FALSE;
        for i = 0 to 1 do
            prow[[i]] = ActivePredicateElement{PL}(mask1, 2*row + i, 8);
            pcol[[i]] = ActivePredicateElement{PL}(mask2, 2*col + i, 8);
            any_active = any_active || (prow[[i]] && pcol[[i]]);
        end;

        if any_active then
            var sum : bits(16) = operand3[(row*dim+col)*:16];
            var rowop : bits(16) = Zeros{};
            var colop : bits(16) = Zeros{};
            for i = 0 to 1 do
                if prow[[i]] then
                    rowop[i*:8] = operand1[(2*row + i)*:8];
                end;
                if pcol[[i]] then
                    colop[i*:8] = operand2[(2*col + i)*:8];
                end;
            end;

            sum = FP8DotAddFP{16, 16}(sum, rowop, colop, FPCR(), FPMR());
            result[(row*dim+col)*:16] = sum;
        else
            result[(row*dim+col)*:16] = operand3[(row*dim+col)*:16];
        end;
    end;
end;

ZAtile{dim*dim*16}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.114 FMOPA (widening, 4-way)

8-bit floating-point sum of outer products to single-precision, accumulating

This instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 4$ sub-matrix of 8-bit floating-point values held in the first source vector to single-precision values and multiplies it by the widened $4 \times SVL_S$ sub-matrix of 8-bit floating-point values in the second source vector to single-precision values.

Each source vector is independently predicated by a corresponding governing predicate. When a 8-bit source element is Inactive it is treated as having the value +0.0, but if both groups of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision sum of outer products are scaled by $2^{-UInt(FPMR.LSCALE)}$, before being destructively added to the single-precision destination tile. This is equivalent to performing a downscaled 4-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

SME2

(FEAT_SME_F8F32)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3 2	1 0
1	0 0	0 0 0 0	0	1 0	1	Zm	Pm	Pn	Zn	0	0 0	ZAda

Encoding

```
FMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 32;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var prow : array [[4]] of boolean;
        var pcol : array [[4]] of boolean;
        var any_active : boolean = FALSE;
        for i = 0 to 3 do
            prow[[i]] = ActivePredicateElement{PL}(mask1, 4*row + i, 8);
            pcol[[i]] = ActivePredicateElement{PL}(mask2, 4*col + i, 8);
            any_active = any_active || (prow[[i]] && pcol[[i]]);
        end;

        if any_active then
            var sum : bits(32) = operand3[(row*dim+col)*:32];
            var rowop : bits(32) = Zeros{};
            var colop : bits(32) = Zeros{};
            for i = 0 to 3 do
                if prow[[i]] then
                    rowop[i*:8] = operand1[(4*row + i)*:8];
                end;
                if pcol[[i]] then
                    colop[i*:8] = operand2[(4*col + i)*:8];
                end;
            end;

            sum = FP8DotAddFP{32, 32}(sum, rowop, colop, FPCR(), FPMR());
            result[(row*dim+col)*:32] = sum;
        else
            result[(row*dim+col)*:32] = operand3[(row*dim+col)*:32];
        end;
    end;
end;

ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.115 FMOPA (widening, 2-way, FP16 to FP32)

Half-precision sum of outer products to single-precision, accumulating

This instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision values held in the first source vector to single-precision values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision values in the second source vector to single-precision values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is then destructively added to the single-precision destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	1	Zm		Pm		Pn		Zn		0	0	0	ZAda	
																			S				

Encoding

```
FMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 32;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        // determine row/col predicates
        let prow_0 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 0, 16));
        let prow_1 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 1, 16));
        let pcol_0 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 0, 16));
        let pcol_1 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 1, 16));

        var sum : bits(32) = operand3[(row*dim+col)*:32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            let erow_0 : bits(16) = (if prow_0 then operand1[(2*row + 0)*:16]
                                     else FPZero{16}('0'));
            let erow_1 : bits(16) = (if prow_1 then operand1[(2*row + 1)*:16]
                                     else FPZero{16}('0'));
            let ecol_0 : bits(16) = (if pcol_0 then operand2[(2*col + 0)*:16]
                                     else FPZero{16}('0'));
            let ecol_1 : bits(16) = (if pcol_1 then operand2[(2*col + 1)*:16]
                                     else FPZero{16}('0'));
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());

        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;

ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.116 FMOPA (non-widening)

Floating-point outer product, accumulating

The half-precision variant works with a 16-bit element ZA tile.

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

This instruction generates an outer product of the first source vector and the second source vector. In case of the half-precision variant, the first source is $SVL_H \times 1$ vector and the second source is $1 \times SVL_H$ vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, $SVL_H \times SVL_H$ in case of half-precision variant, $SVL_S \times SVL_S$ in case of single-precision variant or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 3 classes: Half-precision, Single-precision, and Double-precision

Half-precision

(FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	0	Zm				Pm			Pn			Zn				0	1	0	0	
																									S				ZAda

Encoding

```
FMOPA  <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Single-precision

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	0	Zm				Pm			Pn			Zn				0	0	0	ZAda	
																									S				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Double-precision

(FEAT_SME_F64F64)

31	30	29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3	2 0
1	0	0	0 0 0 0	0	1 1	0	Zm	Pm	Pn	Zn	0	0	ZAda
											S		

Encoding

```
FMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

For the “Half-precision” variant: is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

For the “Single-precision” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “Double-precision” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        let elem2 : bits(esize) = op2[col*:esize];
        let elem3 : bits(esize) = op3[(row*dim+col)*:esize];

        if (ActivePredicateElement{PL}(mask1, row, esize) &&
              ActivePredicateElement{PL}(mask2, col, esize)) then
            let elem1 : bits(esize) = op1[row*:esize];
            result[(row*dim+col)*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
        else
            result[(row*dim+col)*:esize] = elem3;
        end;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.117 FMOPS (widening)

Half-precision sum of outer products to single-precision, subtracting

This instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision values held in the first source vector to single-precision values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision values in the second source vector to single-precision values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is then destructively subtracted from the single-precision destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	1	Zm		Pm		Pn		Zn		1	0	0	ZAda	
																			S				

Encoding

```
FMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV 32;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        // determine row/col predicates
        let prow_0 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 0, 16));
        let prow_1 : boolean = (ActivePredicateElement{PL}(mask1, 2*row + 1, 16));
        let pcol_0 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 0, 16));
        let pcol_1 : boolean = (ActivePredicateElement{PL}(mask2, 2*col + 1, 16));

        var sum : bits(32) = operand3[(row*dim+col)*:32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            var erow_0 : bits(16) = (if prow_0 then operand1[(2*row + 0)*:16] else FPZero{16}('0'));
            var erow_1 : bits(16) = (if prow_1 then operand1[(2*row + 1)*:16] else FPZero{16}('0'));
            let ecol_0 : bits(16) = (if pcol_0 then operand2[(2*col + 0)*:16]
                                        else FPZero{16}('0'));
            let ecol_1 : bits(16) = (if pcol_1 then operand2[(2*col + 1)*:16]
                                        else FPZero{16}('0'));
            if prow_0 then erow_0 = FPNeg{16}(erow_0, FPCR()); end;
            if prow_1 then erow_1 = FPNeg{16}(erow_1, FPCR()); end;
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR());

        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;

ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.118 FMOPS (non-widening)

Floating-point outer product, subtracting

The half-precision variant works with a 16-bit element ZA tile.

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

This instruction generates an outer product of the first source vector and the second source vector. In case of the half-precision variant, the first source is $SVL_H \times 1$ vector and the second source is $1 \times SVL_H$ vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, $SVL_H \times SVL_H$ in case of half-precision variant, $SVL_S \times SVL_S$ in case of single-precision variant or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 3 classes: Half-precision, Single-precision, and Double-precision

Half-precision

(FEAT_SME_F16F16)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		1	1	0	0	
																	S				ZAda

Encoding

```
FMOPS  <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Single-precision

(FEAT_SME)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	0	0	0	0	1	0	0	Zm		Pm		Pn		Zn		1	0	0	ZAda	
																	S				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Double-precision (FEAT_SME_F64F64)

31	30	29	28 25	24	23 22	21	20 16	15 13	12 10	9 5	4	3	2 0
1	0	0	0 0 0 0	0	1 1	0	Zm	Pm	Pn	Zn	1	0	ZAda
											S		

Encoding

```
FMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

For the “Half-precision” variant: is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

For the “Single-precision” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “Double-precision” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let op1 : bits(VL) = Z{}(n);
let op2 : bits(VL) = Z{}(m);
let op3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        let elem2 : bits(esize) = op2[col*:esize];
        let elem3 : bits(esize) = op3[(row*dim+col)*:esize];

        if (ActivePredicateElement{PL}(mask1, row, esize) &&
              ActivePredicateElement{PL}(mask2, col, esize)) then
            let elem1 : bits(esize) = FPNeg{}(op1[row*:esize], FPCR());
            result[(row*dim+col)*:esize] = FPMulAdd_ZA{esize}(elem3, elem1, elem2, FPCR());
        else
            result[(row*dim+col)*:esize] = elem3;
        end;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.119 FMUL (multiple and single vector)

Multi-vector floating-point multiply by vector

This instruction multiplies all the floating-point elements of the two or four first source vectors by the corresponding floating-point elements of the second source vector, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2p2)

31	30 29	28 25	24	23 22	21	20 17	16	15 10	9 6	5	4 1	0
1	1 0	0 0 0 0	1	!=00	1	Zm	0	1 1 1 0 1 0	Zn	0	Zd	0
				size								

Encoding

```
FMUL  { <Zd1>.<T>-<Zd2>.<T> }, { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd::'0');
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2p2)

31	30 29	28 25	24	23 22	21	20 17	16	15 10	9 7	6	5	4 2	1	0
1	1 0	0 0 0 0	1	!=00	1	Zm	1	1 1 1 0 1 0	Zn	0	0	Zd	0	0
				size										

Encoding

```
FMUL  { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd::'00');
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:(esize)];
        let element2 : bits(esize) = operand2[e*:(esize)];
        results[[r]][e*:(esize)] = FPMul{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.120 FMUL (multiple vectors)

Multi-vector floating-point multiply

This instruction multiplies all the floating-point elements of the two or four first source vectors by the corresponding floating-point elements of the two or four second source vectors, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2p2)

31	30 29	28 25	24	23 22	21	20 17	16	15 10	9 6	5	4 1	0
1	1 0	0 0 0 0	1	!=00	1	Zm	0	1 1 1 0 0 1	Zn	0	Zd	0
				size								

Encoding

```
FMUL  { <Zd1>.<T>-<Zd2>.<T> }, { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd::'0');
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2p2)

31	30 29	28 25	24	23 22	21	20 18	17 16	15 10	9 7	6	5	4 2	1	0
1	1 0	0 0 0 0	1	!=00	1	Zm	0 1	1 1 1 0 0 1	Zn	0	0	Zd	0	0
				size										

Encoding

```
FMUL  { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Zd::'00');
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:(esize)];
        let element2 : bits(esize) = operand2[e*:(esize)];
        results[[r]][e*:(esize)] = FPMul{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.121 FRINTA

Multi-vector single-precision round to integral value, to nearest with ties away from zero

This instruction rounds each single-precision element of the two or four source vectors to the nearest integral floating-point value, with ties rounding away from zero, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19	18		16	15					10	9		6	5	4		1	0
1	1	0	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	0	0	Zn			0	Zd			0
								size						opc															

Encoding

```
FRINTA  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19	18		16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	1	0	1	1	1	1	0	0	1	1	1	0	0	0	Zn			0	0	Zd			0	0
								size						opc																	

Encoding

```
FRINTA  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEAWAY;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FPRoundInt{32}(element, FPCR(), rounding, exact);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.122 FRINTM

Multi-vector single-precision round to integral value, toward minus Infinity

This instruction rounds down each single-precision element of the two or four source vectors to an integral floating-point value, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	18	16	15	10	9	6	5	4	1	0
1	1	0	0 0 0 0		1	1	0	1	0	1	0 1 0		1 1 1 0 0 0		Zn		0	Zd		0
						size					opc									

Encoding

```
FRINTM  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	18	16	15	10	9	7	6	5	4	2	1	0
1	1	0	0 0 0 0		1	1	0	1	1	1	0 1 0		1 1 1 0 0 0		Zn		0	0	Zd		0	0
						size					opc											

Encoding

```
FRINTM  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_NEGINF;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FPRoundInt{32}(element, FPCR(), rounding, exact);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.123 FRINTN

Multi-vector single-precision round to integral value, to nearest with ties to even

This instruction rounds each single-precision element of the two or four source vectors to the nearest integral floating-point value, with ties rounding to an even value, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	18	16	15	10	9	6	5	4	1	0
1	1	0	0 0 0 0		1	1	0	1	0	1	0 0 0		1 1 1 0 0 0		Zn		0	Zd		0
						size					opc									

Encoding

```
FRINTN  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	18	16	15	10	9	7	6	5	4	2	1	0
1	1	0	0 0 0 0		1	1	0	1	1	1	0 0 0		1 1 1 0 0 0		Zn		0	0	Zd		0	0
						size					opc											

Encoding

```
FRINTN  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_TIEEVEN;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FPRoundInt{32}(element, FPCR(), rounding, exact);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.124 FRINTP

Multi-vector single-precision round to integral value, toward plus Infinity

This instruction rounds up each single-precision element of the two or four source vectors to an integral floating-point value, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19	18		16	15					10	9		6	5	4		1	0
1	1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	0	0	0	Zn			0	Zd			0
								size						opc															

Encoding

```
FRINTP  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19	18		16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	0	0	0	Zn			0	0	Zd			0	0
								size						opc																	

Encoding

```
FRINTP  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let exact : boolean = FALSE;
let rounding : FPRounding = FPRounding_POSINF;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FPRoundInt{32}(element, FPCR(), rounding, exact);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.125 FSCALE (multiple and single vector)

Multi-vector floating-point adjust exponent by vector

This instruction multiplies the floating-point elements of the two or four first source vectors by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_FP8)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	1	0
1	1	0	0 0 0 0		1	!=00		1	0	Zm		1 0 1 0 0		0	0	1	1 0 0		Zdn		0
						size															op

Encoding

```
FSCALE  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_FP8)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	2	1	0
1	1	0	0 0 0 0		1	!=00		1	0	Zm		1 0 1 0 1		0	0	1	1 0 0		Zdn		0	0
						size																op

Encoding

```
FSCALE  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : integer = SInt(operand2[e*:esize]);
        results[[r]][e*:esize] = FPScale{esize}(element1, element2, FPCR());
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.126 FSCALE (multiple vectors)

Multi-vector floating-point adjust exponent

This instruction multiplies the floating-point elements of the two or four first source vectors by 2.0 to the power of the signed integer values in the corresponding elements of the two or four second source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20			17	16						10	9		7	6	5	4			1	0
1	1	0	0	0	0	0	1	!=00		1	Zm				0	1	0	1	1	0	0	0	1	1	0	0	Zdn				0
								size																	opc						o2

Encoding

```
FSCALE  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2 && FEAT_FP8)

31	30	29	28			25	24	23	22	21	20		18	17							10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	!=00		1	Zm			0	0	1	0	1	1	1	0	0	1	1	0	0	Zdn			0	0
								size																	opc						o2

Encoding

```
FSCALE  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !IsFeatureImplemented(FEAT_FP8) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : integer = SInt(operand2[e*:esize]);
        results[[r]][e*:esize] = FPScale{esize}(element1, element2, FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.127 FSUB

Multi-vector floating-point subtract from ZA array vectors

This instruction destructively subtracts all elements of the two or four source vectors from the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID_AA64SMFR0_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors, Two ZA single-vectors of half-precision elements, Four ZA single-vectors, and Four ZA single-vectors of half-precision elements

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv		1	1	1	Zm			0	0	1	off3		
																											S			

Encoding

```
FSUB  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Two ZA single-vectors of half-precision elements

(FEAT_SME_F16F16 || FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	Rv		1	1	1	Zm			0	0	1	off3		
									sz																		S			

Encoding

```
FSUB  ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) && !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_F64F64))

31	30	29	28	25	24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv		1	1	1	Zm	0	0	0	1	off3	
																								S		

Encoding

```
FSUB  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_F64F64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Four ZA single-vectors of half-precision elements

(FEAT_SME_F16F16 || FEAT_SME_F8F16)

31	30	29	28	25	24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	1	1	0	1	0	0	1	0	1	0	Rv		1	1	1	Zm	0	0	0	1	off3	
							sz																	S		

Encoding

```
FSUB  ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F16F16) && !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 16;
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zm1>

For the “Two ZA single-vectors” and “Two ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” and “Four ZA single-vectors of half-precision elements” variants: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zm4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = ZAvector{}(vec);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = FPSub_ZA{esize}(element1, element2, FPCR());
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.128 FTMOPA (widening, 2-way, FP8 to FP16)

8-bit floating-point sparse sum of outer products to half-precision, accumulating

This instruction generates 8-bit floating-point sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 16-bit element ZA tile.

The sum of outer products is generated by multiplying the selected 2-in-4 8-bit floating-point values from each overlapping 16-bit containers of the two $SVL_H \times 2$ sub-matrices in the first source vectors by the two 8-bit floating point values from the corresponding 16-bit container of the $2 \times SVL_H$ sub-matrix in the second source vector. The two selected elements from each overlapping 16-bit containers of the first source vectors correspond to 2-in-4 elements of rows of two $SVL_H \times 2$ sub-matrices. Each 16-bit container of the second source vector holds 2 elements of columns of a compressed $2 \times SVL_H$ sub-matrix.

The 2-in-4 8-bit floating values from overlapping 16-bit containers of the first source vectors are selected by 4-bit control in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to 16-bit containers of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The instruction widens the selected elements of sub-matrices of 8-bit floating-point values held in the first source vectors to half-precision values and multiplies them by the corresponding widened elements of sub-matrix of 8-bit floating-point values in the second source vector to half-precision values. The resulting $SVL_H \times SVL_H$ half-precision sum of outer products is scaled by $2^{-UInt(FPMR.LSCALE[3:0])}$, before being destructively added to the half-precision destination tile. This is equivalent to performing a downscaled 2-way dot product and accumulate to each of the destination tile elements.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP && FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5		4	3	2	1	0
1	0	0	0	0	0	0	0	0	1	1	Zm				0	0	0	K	Zk		Zn				i2			1	0	0	ZAda

Encoding

```
FTMOPA  <ZAda>.H, { <Zn1>.B-<Zn2>.B }, <Zm>.B, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) || !IsFeatureImplemented(FEAT_SME_F8F16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 16;
let csize : integer{} = VL DIV 4;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*16) = ZAtile{}(da, 16);
var result : bits(dim*dim*16);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var i : integer = 0;
        var rowop : bits(16) = Zeros{};
        var colop : bits(16) = Zeros{};
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            for e = 0 to 1 do
                if i < 2 && ctrl[(4*col + 2*r + e)*:1] == '1' then
                    rowop[i*:8] = op1[(2*row + e)*:8];
                    i = i + 1;
                end;
            end;
        end;
        for j = 0 to 1 do
            colop[j*:8] = op2[(2*col + j)*:8];
        end;
        let sum : bits(16) = op4[(row*dim+col)*:16];
        result[(row*dim+col)*:16] = FP8DotAddFP{16, 16}(sum, rowop, colop, FPCR(), FPMR());
    end;
end;
ZAtile{dim*dim*16}(da, 16) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.129 FTMOPA (widening, 4-way)

8-bit floating-point sparse sum of outer products to single-precision, accumulating

This instruction generates 8-bit floating-point sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected two sets of 2-in-4 8-bit floating-point values from each overlapping 32-bit containers of the two $SVL_S \times 4$ sub-matrices in the first source vectors by the four 8-bit floating-point values from the corresponding 32-bit container of the $4 \times SVL_S$ sub-matrix in the second source vector. The four selected elements from the overlapping 32-bit containers of the first source vectors correspond to two sets of 2-in-4 elements of rows of two $SVL_S \times 4$ sub-matrices. Each 32-bit container of the second source vector holds 4 elements of columns of a compressed $4 \times SVL_S$ sub-matrix.

The two sets of 2-in-4 8-bit floating values from overlapping 32-bit containers of the first source vectors are selected by pairs of 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to each 32-bit container of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The instruction widens the selected elements of sub-matrices of 8-bit floating-point values held in the first source vectors to single-precision values and multiplies them by the corresponding widened elements of sub-matrix of 8-bit floating-point values in the second source vector to single-precision values. The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is scaled by $2^{-UInt(FPMR.LSCALE)}$, before being destructively added to the single-precision destination tile. This is equivalent to performing a downscaled 4-way dot product and accumulate to each of the destination tile elements.

The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP && FEAT_SME_F8F32)

31	30 29	28 25	24	23 22	21	20 16	15	14	13	12	11 10	9 6	5 4	3	2	1 0
1	0 0	0 0 0 0	0	0 1	1	Zm	0	0	0	K	Zk	Zn	i2	0	0	ZAda

Encoding

```
FTMOPA  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) || !IsFeatureImplemented(FEAT_SME_F8F32) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 4;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var rowop : bits(32) = Zeros{};
        var colop : bits(32) = Zeros{};
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            var i : integer = 0;
            for e = 0 to 3 do
                if i < 2 && ctrl[(8*col + 4*r + e)*:1] == '1' then
                    rowop[(2*r + i)*:8] = op1[(4*row + e)*:8];
                    i = i + 1;
                end;
            end;
        end;
        for j = 0 to 3 do
            colop[j*:8] = op2[(4*col + j)*:8];
        end;
        let sum : bits(32) = op4[(row*dim+col)*:32];
        result[(row*dim+col)*:32] = FP8DotAddFP{32, 32}(sum, rowop, colop, FPCR(), FPMR());
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.130 FTMOPA (widening, 2-way, FP16 to FP32)

Half-precision sparse sum of outer products to single-precision, accumulating

This instruction generates half-precision sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected 2-in-4 half-precision values from each overlapping 32-bit containers of the two $SVL_S \times 2$ sub-matrices in the first source vectors by the two half-precision values from the corresponding 32-bit container of the $2 \times SVL_S$ sub-matrix in the second source vector. The two selected elements from each overlapping 32-bit containers of the first source vectors correspond to 2-in-4 elements of rows of two $SVL_S \times 2$ sub-matrices. Each 32-bit container of the second source vector holds 2 elements of columns of a compressed $2 \times SVL_S$ sub-matrix.

The 2-in-4 half-precision values from overlapping 32-bit containers of the first source vectors are selected by 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to 32-bit containers of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The instruction widens the selected elements of sub-matrices of half-precision values held in the first source vectors to single-precision values and multiplies them by the corresponding widened elements of sub-matrix of half-precision values in the second source vector to single-precision values. The resulting $SVL_S \times SVL_S$ single-precision sum of outer products is then destructively added to the single-precision destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30 29	28 25	24	23 22	21	20 16	15	14	13	12	11 10	9 6	5 4	3	2	1 0
1	0 0	0 0 0 0	1	0 1	1	Zm	0	0	0	K	Zk	Zn	i2	0	0	ZAda

Encoding

```
FTMOPA  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 8;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var i : integer = 0;
        var erow : array [[2]] of bits(16);
        erow[[0]] = FPZero{16}('0');
        erow[[1]] = FPZero{16}('0');
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            for e = 0 to 1 do
                if i < 2 && ctrl[(4*col + 2*r + e)*:1] == '1' then
                    erow[[i]] = op1[(2*row + e)*:16];
                    i = i + 1;
                end;
            end;
        end;
        var ecol : array [[2]] of bits(16);
        ecol[[0]] = op2[(2*col + 0)*:16];
        ecol[[1]] = op2[(2*col + 1)*:16];
        let sum : bits(32) = op4[(row*dim+col)*:32];
        result[(row*dim+col)*:32] = FPDotAdd_ZA(sum, erow[[0]], erow[[1]],
                                                ecol[[0]], ecol[[1]], FPCR());
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.131 FTMOPA (non-widening)

Floating-point sparse outer product, accumulating

This instruction generates floating-point outer product by multiplying the 1-in-2 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the destination ZA tile elements.

In case of the half-precision variant, the outer product is generated by multiplying the selected 1-in-2 half-precision value from each overlapping 16-bit containers of the two $SVL_H \times 1$ sub-matrices in the first source vectors by the half-precision value from the corresponding 16-bit container of the compressed $1 \times SVL_H$ sub-matrix in the second source vector. In case of the single-precision variant, the outer product is generated by multiplying the selected 1-in-2 single-precision value from each overlapping 32-bit containers of the two $SVL_S \times 1$ sub-matrices in the first source vectors by the single-precision value from the corresponding 32-bit container of the compressed $1 \times SVL_S$ sub-matrix in the second source vector.

The 1-in-2 floating-point value in the first source vectors is selected by 2-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the result. If both bits of the 2-bit control corresponding to the elements of the first source vectors are 1, only the element corresponding to the least significant bit is selected.

The instruction multiplies the selected elements of sub-matrices of floating-point values held in the first source vectors by the corresponding elements of sub-matrix of floating-point values in the second source vector. The resulting outer product, $SVL_H \times SVL_H$ in case of the half-precision variant or $SVL_S \times SVL_S$ in case of the single-precision variant, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision

(FEAT_SME_TMOP && FEAT_SME_F16F16)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	1	0	Zm				0	0	0	K	Zk		Zn				i2		1	0	0	ZAda

Encoding

```
FTMOPA  <ZAda>.H, { <Zn1>.H-<Zn2>.H }, <Zm>.H, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) || !IsFeatureImplemented(FEAT_SME_F16F16) then
    EndOfDecode(Decode_UNDEF);
end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let esize : integer{} = 16;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Single-precision

(FEAT_SME_TMOP)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	1	0	Zm				0	0	0	K	Zk		Zn				i2		0	0	ZAda	

Encoding

```
FTMOPA  <ZAda>.S, { <Zn1>.S-<Zn2>.S }, <Zm>.S, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let esize : integer{} = 32;
```

Assembler Symbols

<ZAda>

For the “Half-precision” variant: is the name of the ZA tile ZA0-ZA1, encoded in the “ZAda” field.

For the “Single-precision” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV esize;
let csize : integer{} = (VL * 2) DIV esize;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*esize) = ZAtile{dim*dim*esize}(da, esize);
var result : bits(dim*dim*esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var i : integer = 0;
        var elem1 : bits(esize) = FPZero{esize}('0');
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            if i < 1 && ctrl[(2*col + r)*:1] == '1' then
                elem1 = op1[row*:(esize)];
                i = i + 1;
            end;
        end;
        let elem2 : bits(esize) = op2[col*:(esize)];
        let sum : bits(esize)   = op4[(row*dim+col)*:(esize)];
        result[(row*dim+col)*:(esize)] = FPMulAdd_ZA{esize}(sum, elem1, elem2, FPCR());
    end;
end;
ZAtile{dim*dim*esize}(da, esize) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.132 FVDOT (FP8 to FP16)

Multi-vector 8-bit floating-point vertical dot product by indexed element to half-precision

This instruction calculates the fused sum-of-products of each vertical pair of 8-bit floating-point values held in the corresponding elements of the two first source vectors with horizontal pair of 8-bit floating-point values in the indexed 16-bit group of the corresponding 128-bit segment of the second source vector. The half-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE[3:0])}}$, before being destructively added without intermediate rounding to the corresponding half-precision elements of the two ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The 8-bit floating-point pairs within the second source vector are specified using an immediate index that selects the same pair position within each 128-bit vector segment.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the ZA operand consists of two ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME_F8F16)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		1	i3h		Zn				1	0	i3l	off3		
																			op												

Encoding

```
FVDOT  ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F16) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i3h::i3l);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 8-bit elements within each 128-bit vector segment, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 16;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 2;
let eltspersegment : integer = 128 DIV 16;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to 1 do
    let operand1a : bits(VL) = Z{}(n+0);
    let operand1b : bits(VL) = Z{}(n+1);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var op1 : bits(16);
        op1[0+:8] = operand1a[(2 * e + r)*:8];
        op1[1*:8] = operand1b[(2 * e + r)*:8];
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let op2 : bits(16) = operand2[s*:16];
        var sum : bits(16)  = operand3[e*:(16)];
        sum = FP8DotAddFP{16, 16}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(16)] = sum;
    end;

    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.133 FVDOT (FP16 to FP32)

Multi-vector half-precision vertical dot product by indexed element to single-precision

This instruction calculates the fused sum-of-products of each vertical pair of half-precision values in the corresponding elements of the two first source vectors with the pair of half-precision values in the indexed 32-bit group of the corresponding 128-bit segment of the second source vector, without intermediate rounding. The single-precision sum-of-products are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The half-precision pairs within the second source vector are specified using an immediate index that selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the ZA operand consists of two ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5		3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				0	0	1	off3			
																			op									opc2				

Encoding

```
FVDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 2;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to 1 do
    let operand1a : bits(VL) = Z{}(n);
    let operand1b : bits(VL) = Z{}(n+1);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        let elt1_a : bits(16) = operand1a[(2 * e + r)*:16];
        let elt1_b : bits(16) = operand1b[(2 * e + r)*:16];
        let elt2_a : bits(16) = operand2[(2 * s + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * s + 1)*:16];
        var sum : bits(32) = operand3[e*:32];
        sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR());
        result[e*:32] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.134 FVDOTB

Multi-vector 8-bit floating-point vertical dot product by indexed element to single-precision (bottom)

This instruction calculates the fused sum-of-products of each vertical pair of 8-bit floating-point values held in the corresponding elements of the two first source vectors with the lower-numbered horizontal pair of 8-bit floating-point values in the indexed 32-bit group of the corresponding 128-bit segment of the second source vector. The single-precision sum-of-products are scaled by $2^{-\mathrm{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the four ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The 8-bit floating-point groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the ZA operand consists of four ZA single-vector groups.

This instruction is unpredicated.

SME2

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		0	1	i2h	Zn				0	0	i2l	off3		
																											T				

Encoding

```
FVDOTB  ZA.S[<Wv>, <offs>, VGx4], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2h::i2l);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 4;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to 3 do
    let operand1a : bits(VL) = Z{}(n+0);
    let operand1b : bits(VL) = Z{}(n+1);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var op1 : bits(16);
        op1[0+:8] = operand1a[(4 * e + r)*:8];
        op1[1*:8] = operand1b[(4 * e + r)*:8];
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = 2*(segmentbase + index);
        let op2 : bits(16) = operand2[s*:16];
        var sum : bits(32)  = operand3[e*:(32)];
        sum = FP8DotAddFP{32, 16}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(32)] = sum;
    end;

    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.135 FVDOTT

Multi-vector 8-bit floating-point vertical dot product by indexed element to single-precision (top)

This instruction calculates the fused sum-of-products of each vertical pair of 8-bit floating-point values held in the corresponding elements of the two first source vectors with the higher-numbered horizontal pair of 8-bit floating-point values in the indexed 32-bit group of the corresponding 128-bit segment of the second source vector. The single-precision sum-of-products are scaled by $2^{-\text{UInt(FPMR.LSCALE)}}$, before being destructively added without intermediate rounding to the corresponding single-precision elements of the four ZA single-vector groups. The 8-bit floating-point encoding format for the elements of the first source vector and the second source vector is selected by `FPMR.F8S1` and `FPMR.F8S2` respectively.

The 8-bit floating-point groups within the second source vector are specified using an immediate index that selects the same group position within each 128-bit vector segment.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the ZA operand consists of four ZA single-vector groups.

This instruction is unpredicated.

SME2

(FEAT_SME_F8F32)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		0	1	i2h	Zn				0	1	i2l	off3		
																											T				

Encoding

```
FVDOTT  ZA.S[<Wv>, <offs>, VGx4], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_F8F32) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2h::i2l);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2h:i2l” fields.

Operation

```
CheckFPMREnabled();
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 4;
let eltspersegment : integer = 128 DIV 32;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to 3 do
    let operand1a : bits(VL) = Z{}(n+0);
    let operand1b : bits(VL) = Z{}(n+1);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var op1 : bits(16);
        op1[0+:8] = operand1a[(4 * e + r)*:8];
        op1[1*:8] = operand1b[(4 * e + r)*:8];
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = 2*(segmentbase + index) + 1;
        let op2 : bits(16) = operand2[s*:16];
        var sum : bits(32)  = operand3[e*:(32)];
        sum = FP8DotAddFP{32, 16}(sum, op1, op2, FPCR(), FPMR());
        result[e*:(32)] = sum;
    end;

    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.136 LD1B (scalar plus immediate, strided registers)

Contiguous load of bytes to multiple strided vectors (immediate index)

This instruction performs a contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			0	0	0	PNg			Rn			T	0	Zt		
																msz									N			

Encoding

```
LD1B  { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			1	0	0	PNg			Rn			T	0	0	Zt	
																msz									N			

Encoding

```
LD1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.137 LD1B (scalar plus scalar, strided registers)

Contiguous load of bytes to multiple strided vectors (scalar index)

This instruction performs a contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	0	0	Rm		0	0	0	PNg		Rn		T	0	Zt	
												msz							N		

Encoding

```
LD1B  { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	0	0	Rm		1	0	0	PNg		Rn		T	0	0	Zt	
												msz							N			

Encoding

```
LD1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.138 LD1B (scalar plus scalar, tile slice)

Contiguous load of bytes to 8-bit element ZA tile slice

This instruction performs a contiguous load of bytes to an 8-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	0
1	1	1	0	0	0	0	0	0	0	0	Rm		V	Rs		Pg		Rn		0	off4	

msz

Encoding

```
LD1B  { ZA0<HV>.B[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = 0;
let offset : integer = UInt(off4);
let esize : integer{} = 8;
let vertical : boolean = V == '1';
```

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

for e = 0 to dim - 1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    moffs = moffs + 1;
end;

ZAslice{VL}(t, esize, vertical, slice) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.139 LD1D (scalar plus immediate, strided registers)

Contiguous load of doublewords to multiple strided vectors (immediate index)

This instruction performs a contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

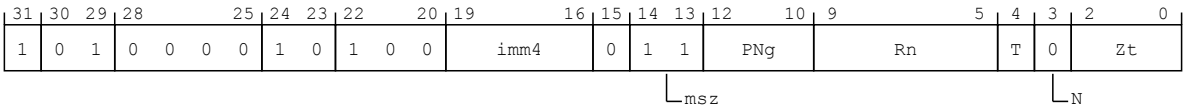

Encoding

```
LD1D  { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

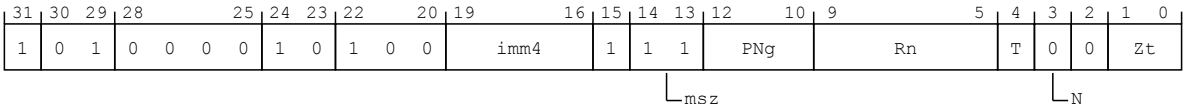

Encoding

```
LD1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.140 LD1D (scalar plus scalar, strided registers)

Contiguous load of doublewords to multiple strided vectors (scalar index)

This instruction performs a contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3	2		0
1	0	1	0	0	0	0	1	0	0	0	Rm				0	1	1	PNg			Rn				T	0	Zt		
																msz										N			

Encoding

```
LD1D  { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm				1	1	1	PNg			Rn				T	0	0	Zt	
																msz										N			

Encoding

```
LD1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.141 LD1D (scalar plus scalar, tile slice)

Contiguous load of doublewords to 64-bit element ZA tile slice

This instruction performs a contiguous load of doublewords to a 64-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	1	0
1	1	1	0	0	0	0	0	1	1	0	Rm		V	Rs		Pg		Rn		0	ZAt		o1

msz

Encoding

```
LD1D  { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = UInt(o1);
let esize : integer{} = 64;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

for e = 0 to dim - 1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    moffs = moffs + 1;
end;

ZAslice{VL}(t, esize, vertical, slice) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.142 LD1H (scalar plus immediate, strided registers)

Contiguous load of halfwords to multiple strided vectors (immediate index)

This instruction performs a contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

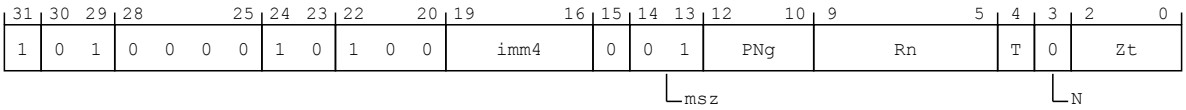

Encoding

```
LD1H  { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

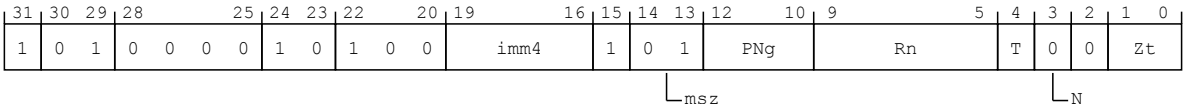

Encoding

```
LD1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.143 LD1H (scalar plus scalar, strided registers)

Contiguous load of halfwords to multiple strided vectors (scalar index)

This instruction performs a contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3	2		0
1	0	1	0	0	0	0	1	0	0	0	Rm				0	0	1	PNg			Rn				T	0	Zt		
																msz										N			

Encoding

```
LD1H  { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm				1	0	1	PNg			Rn				T	0	0	Zt	
																msz										N			

Encoding

```
LD1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.144 LD1H (scalar plus scalar, tile slice)

Contiguous load of halfwords to 16-bit element ZA tile slice

This instruction performs a contiguous load of halfwords to a 16-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	1	1	0	0	0	0	0	0	1	0	Rm		V	Rs		Pg		Rn		0	ZAt	off3	

msz

Encoding

```
LD1H  { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = UInt(off3);
let esize : integer{} = 16;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

for e = 0 to dim - 1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    moffs = moffs + 1;
end;

ZAslice{VL}(t, esize, vertical, slice) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.145 LD1Q

Contiguous load of quadwords to 128-bit element ZA tile slice

This instruction performs a contiguous load of quadwords to a 128-bit element ZA tile slice. The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset that is multiplied by 16 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

31	30	29	28			25	24			21	20			16	15	14	13	12		10	9			5	4	3			0
1	1	1	0	0	0	0	1	1	1	0	Rm				V	Rs		Pg			Rn				0	ZAt			

Encoding

```
LD1Q  { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #4}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = 0;
let esize : integer{} = 128;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, with implicit value 0.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

for e = 0 to dim - 1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    moffs = moffs + 1;
end;

ZAslice{VL}(t, esize, vertical, slice) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.146 LD1W (scalar plus immediate, strided registers)

Contiguous load of words to multiple strided vectors (immediate index)

This instruction performs a contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			0	1	0	PNg			Rn			T	0	Zt		
																msz									N			

Encoding

```
LD1W  { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			1	1	0	PNg			Rn			T	0	0	Zt	
																msz									N			

Encoding

```
LD1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.147 LD1W (scalar plus scalar, strided registers)

Contiguous load of words to multiple strided vectors (scalar index)

This instruction performs a contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	0	0	Rm		0	1	0	PNg		Rn		T	0	Zt	
												msz							N		

Encoding

```
LD1W  { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	0	0	Rm		1	1	0	PNg		Rn		T	0	0	Zt	
												msz							N			

Encoding

```
LD1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.148 LD1W (scalar plus scalar, tile slice)

Contiguous load of words to 32-bit element ZA tile slice

This instruction performs a contiguous load of words to a 32-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12			10	9				5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	0	0	Rm				V	Rs		Pg				Rn					0	ZAt		off2	
								msz																							

Encoding

```
LD1W  { <ZAt><HV>.S[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = UInt(off2);
let esize : integer{} = 32;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var result : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

for e = 0 to dim - 1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = Mem{esize}(addr, accdesc);
    else
        result[e*:esize] = Zeros{esize};
    end;
    moffs = moffs + 1;
end;

ZAslice{VL}(t, esize, vertical, slice) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.149 LDNT1B (scalar plus immediate, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal load of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			0	0	0	PNg			Rn			T	1	Zt		
																msz									N			

Encoding

```
LDNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			1	0	0	PNg			Rn			T	1	0	Zt	
																msz									N			

Encoding

```
LDNT1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.150 LDNT1B (scalar plus scalar, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal load of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	0	0	Rm		0	0	0	PNg		Rn		T	1	Zt	
												msz							N		

Encoding

```
LDNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	0	0	Rm		1	0	0	PNg		Rn		T	1	0	Zt	
												msz							N			

Encoding

```
LDNT1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.151 LDNT1D (scalar plus immediate, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal load of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			0	1	1	PNg			Rn			T	1	Zt		
																msz									N			

Encoding

```
LDNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			1	1	1	PNg			Rn			T	1	0	Zt	
																msz									N			

Encoding

```
LDNT1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.152 LDNT1D (scalar plus scalar, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal load of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	0	Rm	0	1	1	PNg		Rn	T	1	Zt	
													msz						N		

Encoding

```
LDNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm	1	1	1	PNg		Rn	T	1	0	Zt	
													msz						N			

Encoding

```
LDNT1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.153 LDNT1H (scalar plus immediate, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal load of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			0	0	1	PNg			Rn			T	1	Zt		
																msz									N			

Encoding

```
LDNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			1	0	1	PNg			Rn			T	1	0	Zt	
																msz									N			

Encoding

```
LDNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.154 LDNT1H (scalar plus scalar, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal load of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	0	0	Rm		0	0	1	PNg		Rn		T	1	Zt	
												msz							N		

Encoding

```
LDNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	0	0	Rm		1	0	1	PNg		Rn		T	1	0	Zt	
												msz							N			

Encoding

```
LDNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.155 LDNT1W (scalar plus immediate, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal load of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			0	1	0	PNg			Rn			T	1	Zt		
																msz									N			

Encoding

```
LDNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4			1	1	0	PNg			Rn			T	1	0	Zt	
																msz									N			

Encoding

```
LDNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.156 LDNT1W (scalar plus scalar, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal load of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	0	0	Rm		0	1	0	PNg		Rn		T	1	Zt	
												msz							N		

Encoding

```
LDNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	0	0	Rm		1	1	0	PNg		Rn		T	1	0	Zt	
												msz							N			

Encoding

```
LDNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
var values : array [[4]] of bits(VL);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            values[[r]][e*:esize] = Mem{esize}(addr, accdesc);
        else
            values[[r]][e*:esize] = Zeros{esize};
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(transfer) = values[[r]];
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.157 LDR (array vector)

Load ZA array vector

This instruction performs a contiguous load of bytes to a ZA array vector from the memory address generated by a 64-bit scalar base plus an optional immediate offset multiplied by the current vector length in bytes. The ZA array vector is selected by the sum of the vector select register and the same immediate offset, modulo the number of bytes in a Streaming SVE vector.

This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME

(FEAT_SME)

31	30	29	28			25	24			22	21	20				16	15	14	13	12			10	9			5	4	3			0
1	1	1	0	0	0	0	1	0	0		0	0	0	0	0	0	0	Rv		0	0	0		Rn				0	off4			
											op																					

Encoding

```
LDR  ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let v : integer = UInt('011'::Rv);
let offset : integer = UInt(off4);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W12-W15, encoded in the “Rv” field.

<offs>

Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the “off4” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckSMEAndZAEnabled();
let SVL : integer{} = CurrentSVL();
let dim : integer{} = SVL DIV 8;
var base : bits(64);
let moffs : integer = offset * dim;
var result : bits(SVL);
let vbase : bits(32) = X{}(v);
let vec : integer = (UInt(vbase) + offset) MOD dim;
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP{64}();
else
    base = X{64}(n);
end;

var addr : bits(64) = AddressAdd(base, moffs, accdesc);

let aligned : boolean = IsAlignedSize(base, 16);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;

for e = 0 to dim-1 do
    result[e*:8] = AArch64_MemSingle{8}(addr, accdesc, aligned);
    addr = AddressIncrement(addr, 1, accdesc);
end;

ZAvector{SVL}(vec) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.158 LDR (table)

Load ZT0 register

This instruction loads the 64-byte ZT0 register from the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2

(FEAT_SME2)

31	30	29	28			25	24		22	21					16	15	14				10	9		5	4		2	1	0
1	1	1	0	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	Rn			0	0	0	0	0
										opc																		opc2	

Encoding

```
LDR  ZT0, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
```

Assembler Symbols

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
let elements : integer = 512 DIV 8;
var addr : bits(64);
var result : bits(512);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    CheckSPAlignment();
    addr = SP{64}();
else
    addr = X{64}(n);
end;

let aligned : boolean = IsAlignedSize(addr, 16);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    result[e*:8] = AArch64_MemSingle{8}(addr, accdesc, aligned);
    addr = AddressIncrement(addr, 1, accdesc);
end;

ZT0{512}() = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.159 LUTI2 (two registers)

Lookup table read with 2-bit indexes (two registers)

This instruction copies 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: Consecutive and Strided

Consecutive

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17		15	14	13	12	11	10	9		5	4		1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	1	i3			1	size		0	0	Zn			Zd			0

opc2

Encoding

```
LUTI2  { <Zd1>.<T>-<Zd2>.<T> }, ZT0, <Zn>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 2;
let n : integer = UInt(Zn);
let dstride : integer = 1;
let d : integer = UInt(Zd::'0');
let imm : integer = UInt(i3);
let nreg : integer{} = 2;
```

Strided

(FEAT_SME2p1)

31	30	29	28			25	24					19	18	17		15	14	13	12	11	10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	0	0	1	1	1	i3			1	size		0	0	Zn			D	0	Zd		

opc2

Encoding

```
LUTI2  { <Zd1>.<T>, <Zd2>.<T> }, ZT0, <Zn>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
if size == '10' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 2;
let n : integer = UInt(Zn);
let dstride : integer = 8;
let d : integer = UInt(D::'0'::Zd);
let imm : integer = UInt(i3);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zd1>

For the “Consecutive” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Strided” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 of the destination multi-vector group, encoded as “D:‘0’:Zd”.

<T>

For the “Consecutive” variant: is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

For the “Strided” variant: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	B
1	H

<Zd2>

For the “Consecutive” variant: is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

For the “Strided” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 of the destination multi-vector group, encoded as “D:‘1’:Zd”.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<index>

Is the vector segment index, in the range 0 to 7, encoded in the “i3” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let segments : integer = (esize DIV (isize * nreg)) as integer{1, 2, 4, 8, 16};
let segment : integer = imm MOD segments;
let indexes : bits(VL) = Z{}(n);
var dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = (segment * nreg + r) * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst) = result;
    dst = dst + dstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.160 LUTI2 (four registers)

Lookup table read with 2-bit indexes (four registers)

This instruction copies 8-bit, 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: Consecutive and Strided

Consecutive

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12	11	10	9		5	4		2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	1	i2		1	0	size		0	0	Zn			Zd			0	0

opc2

Encoding

```
LUTI2  { <Zd1>.<T>-<Zd4>.<T> }, ZT0, <Zn>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 2;
let n : integer = UInt(Zn);
let dstride : integer = 1;
let d : integer = UInt(Zd::'00');
let imm : integer = UInt(i2);
let nreg : integer{} = 4;
```

Strided

(FEAT_SME2p1)

31	30	29	28			25	24					19	18	17	16	15	14	13	12	11	10	9		5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	1	i2		1	0	size		0	0	Zn			D	0	0	Zd	

opc2

Encoding

```
LUTI2  { <Zd1>.<T>, <Zd2>.<T>, <Zd3>.<T>, <Zd4>.<T> }, ZT0, <Zn>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
if size == '10' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 2;
let n : integer = UInt(Zn);
let dstride : integer = 4;
let d : integer = UInt(D::'00'::Zd);
let imm : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zd1>

For the “Consecutive” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

For the “Strided” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 of the destination multi-vector group, encoded as “D:‘00’:Zd”.

<T>

For the “Consecutive” variant: is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

For the “Strided” variant: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	B
1	H

<Zd4>

For the “Consecutive” variant: is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

For the “Strided” variant: is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 of the destination multi-vector group, encoded as “D:‘11’:Zd”.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<index>

Is the vector segment index, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zd2>

Is the name of the second scalable vector register Z4-Z7 or Z20-Z23 of the destination multi-vector group, encoded as “D:‘01’:Zd”.

<Zd3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 of the destination multi-vector group, encoded as “D:‘10’:Zd”.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let segments : integer = (esize DIV (isize * nreg)) as integer{1, 2, 4, 8, 16};
let segment : integer = imm MOD segments;
let indexes : bits(VL) = Z{}(n);
var dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = (segment * nreg + r) * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst) = result;
    dst = dst + dstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.161 LUTI2 (single)

Lookup table read with 2-bit indexes (single)

This instruction copies 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	14	13	12	11	10	9	5	4	0
1	1	0	0	0	0	0	0	1	1	0	0	1	1	i4		size		0	0	Zn		Zd	

opc2

Encoding

```
LUTI2  <Zd>.<T>, ZT0, <Zn>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 2;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let imm : integer = UInt(i4);
let nreg : integer{} = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<index>

Is the vector segment index, in the range 0 to 15, encoded in the “i4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let segments : integer = (esize DIV (isize * nreg)) as integer{1, 2, 4, 8, 16};
let segment : integer = imm MOD segments;
let indexes : bits(VL) = Z{}(n);
let dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = (segment * nreg + r) * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst+r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.162 LUTI4 (two registers)

Lookup table read with 4-bit indexes (two registers)

This instruction copies 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: Consecutive and Strided

Consecutive

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12	11	10	9		5	4		1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	i2		1	size		0	0	Zn			Zd			0

opc2

Encoding

```
LUTI4  { <Zd1>.<T>-<Zd2>.<T> }, ZT0, <Zn>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn);
let dstride : integer = 1;
let d : integer = UInt(Zd::'0');
let imm : integer = UInt(i2);
let nreg : integer{} = 2;
```

Strided

(FEAT_SME2p1)

31	30	29	28			25	24					19	18	17	16	15	14	13	12	11	10	9		5	4	3	2		0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	i2		1	size		0	0	Zn			D	0	Zd		

opc2

Encoding

```
LUTI4  { <Zd1>.<T>, <Zd2>.<T> }, ZT0, <Zn>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
if size == '10' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn);
let dstride : integer = 8;
let d : integer = UInt(D::'0'::Zd);
let imm : integer = UInt(i2);
let nreg : integer{} = 2;
```

Assembler Symbols

<Zd1>

For the “Consecutive” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Strided” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 of the destination multi-vector group, encoded as “D:‘0’:Zd”.

<T>

For the “Consecutive” variant: is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

For the “Strided” variant: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	B
1	H

<Zd2>

For the “Consecutive” variant: is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

For the “Strided” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 of the destination multi-vector group, encoded as “D:‘1’:Zd”.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<index>

Is the vector segment index, in the range 0 to 3, encoded in the “i2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let segments : integer = (esize DIV (isize * nreg)) as integer{1, 2, 4, 8, 16};
let segment : integer = imm MOD segments;
let indexes : bits(VL) = Z{}(n);
var dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = (segment * nreg + r) * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst) = result;
    dst = dst + dstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.163 LUTI4 (four registers, 8-bit)

Lookup table read with 4-bit indexes and 8-bit elements (four registers)

This instruction copies 8-bit elements from ZT0 to four destination vectors using packed 4-bit indices in the two source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Consecutive and Strided

Consecutive

(FEAT_SME_LUTv2)

31	30	29	28			25	24	23	22	21		19	18			14	13	12	11	10	9	6	5	4	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0	size	0	0	Zn		0	Zd		0	0
																			opc								

Encoding

```
LUTI4  { <Zd1>.B-<Zd4>.B }, ZT0, { <Zn1>-<Zn2> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_LUTv2) then EndOfDecode(Decode_UNDEF); end;
if size != '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn::'0');
let dstride : integer = 1;
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```

Strided

(FEAT_SME2p1 && FEAT_SME_LUTv2)

31	30	29	28			25	24					19	18		16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	size		0	0	Zn		0	D	0	0	Zd	
																				opc									

Encoding

```
LUTI4  { <Zd1>.B, <Zd2>.B, <Zd3>.B, <Zd4>.B }, ZT0, { <Zn1>-<Zn2> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) || !IsFeatureImplemented(FEAT_SME_LUTv2) then
    EndOfDecode(Decode_UNDEF);
end;
if size != '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn::'0');
let dstride : integer = 4;
let d : integer = UInt(D::'00'::Zd);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Consecutive” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

For the “Strided” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 of the destination multi-vector group, encoded as “D:‘00’:Zd”.

<Zd4>

For the “Consecutive” variant: is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

For the “Strided” variant: is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 of the destination multi-vector group, encoded as “D:‘11’:Zd”.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd2>

Is the name of the second scalable vector register Z4-Z7 or Z20-Z23 of the destination multi-vector group, encoded as “D:‘01’:Zd”.

<Zd3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 of the destination multi-vector group, encoded as “D:‘10’:Zd”.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let indexes : bits(2*VL) = Z{VL}(n+1) :: Z{VL}(n+0);
var dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = r * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst) = result;
    dst = dst + dstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.164 LUTI4 (four registers, 16-bit and 32-bit)

Lookup table read with 4-bit indexes (four registers)

This instruction copies 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: Consecutive and Strided

Consecutive

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12	11	10	9		5	4		2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	i1	1	0	size		0	0	Zn			Zd			0	0

opc2

Encoding

```
LUTI4  { <Zd1>.<T>-<Zd4>.<T> }, ZT0, <Zn>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn);
let dstride : integer = 1;
let d : integer = UInt(Zd::'00');
let imm : integer = UInt(i1);
let nreg : integer{} = 4;
```

Strided

(FEAT_SME2p1)

31	30	29	28			25	24					19	18	17	16	15	14	13	12	11	10	9		5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	i1	1	0	size		0	0	Zn			D	0	0	Zd	

opc2

Encoding

```
LUTI4  { <Zd1>.H, <Zd2>.H, <Zd3>.H, <Zd4>.H }, ZT0, <Zn>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
if size != '01' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn);
let dstride : integer = 4;
let d : integer = UInt(D::'00'::Zd);
let imm : integer = UInt(i1);
let nreg : integer{} = 4;
```

Assembler Symbols

<Zd1>

For the “Consecutive” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

For the “Strided” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 of the destination multi-vector group, encoded as “D:‘00’:Zd”.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	RESERVED

<Zd4>

For the “Consecutive” variant: is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

For the “Strided” variant: is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 of the destination multi-vector group, encoded as “D:‘11’:Zd”.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<index>

Is the vector segment index, in the range 0 to 1, encoded in the “i1” field.

<Zd2>

Is the name of the second scalable vector register Z4-Z7 or Z20-Z23 of the destination multi-vector group, encoded as “D:‘01’:Zd”.

<Zd3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 of the destination multi-vector group, encoded as “D:‘10’:Zd”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let segments : integer = (esize DIV (isize * nreg)) as integer{1, 2, 4, 8, 16};
let segment : integer = imm MOD segments;
let indexes : bits(VL) = Z{}(n);
var dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = (segment * nreg + r) * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst) = result;
    dst = dst + dstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.165 LUTI4 (single)

Lookup table read with 4-bit indexes (single)

This instruction copies 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16		14	13	12	11	10	9		5	4		0
1	1	0	0	0	0	0	0	1	1	0	0	1	0	1	i3			size		0	0	Zn			Zd		

opc2

Encoding

```
LUTI4  <Zd>.<T>, ZT0, <Zn>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let isize : integer{} = 4;
let n : integer = UInt(Zn);
let d : integer = UInt(Zd);
let imm : integer = UInt(i3);
let nreg : integer{} = 1;
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<index>

Is the vector segment index, in the range 0 to 7, encoded in the “i3” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let segments : integer = (esize DIV (isize * nreg)) as integer{1, 2, 4, 8, 16};
let segment : integer = imm MOD segments;
let indexes : bits(VL) = Z{}(n);
let dst : integer = d;
let table : bits(512) = ZT0{}();

for r = 0 to nreg-1 do
    let base : integer = (segment * nreg + r) * elements;
    var result : bits(VL);
    for e = 0 to elements-1 do
        let index : integer = UInt(indexes[(base+e)*:isize]);
        result[e*:esize] = table[index*:32][esize-1:0];
    end;
    Z{VL}(dst+r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.166 MOV (tile to vector, two registers)

Move ZA two-slice operand to Z two-vector operand

This instruction operates on a horizontal or vertical ZA two-slice operand within a ZA tile of the specified element size.

The first slice of the two-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 2, modulo the number of slices in the tile.

The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This is an alias of MOVA (tile to vector, two registers). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, two registers).
- The description of MOVA (tile to vector, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4		1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs		0	0	0	0	0	off3			Zd			0
								size																						

Encoding for the 8-bit variant

```
MOV  { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]
```

is equivalent to

```
MOVA  { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]
```

and is always the preferred disassembly.

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7	6	5	4		1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs		0	0	0	0	0	ZAn	off2		Zd			0
								size																						

Encoding for the 16-bit variant

```
MOV  { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]
```

is equivalent to

```
MOVA  { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME2)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4				1	0
1	1	0	0	0	0	0		0	1	0	0	0	0		1	1	0	V	Rs		0	0	0		0	0	ZAn		o1	Zd					0

size

Encoding for the 32-bit variant

```
MOV  { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]
```

is equivalent to

```
MOVA  { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]
```

and is always the preferred disassembly.

64-bit

(FEAT_SME2)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4				1	0
1	1	0	0	0	0	0		0	1	1	0	0	0		1	1	0	V	Rs		0	0	0		0	0	ZAn				Zd				0

size

Encoding for the 64-bit variant

```
MOV  { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]
```

is equivalent to

```
MOVA  { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off3” field times 2.

For the “16-bit” variant: is the first slice index offset, encoded as “off2” field times 2.

For the “32-bit” variant: is the first slice index offset, encoded as “o1” field times 2.

For the “64-bit” variant: is the first slice index offset, with implicit value 0.

<offs2>

For the “8-bit” variant: is the second slice index offset, encoded as “off3” field times 2 plus 1.

For the “16-bit” variant: is the second slice index offset, encoded as “off2” field times 2 plus 1.

For the “32-bit” variant: is the second slice index offset, encoded as “o1” field times 2 plus 1.

For the “64-bit” variant: is the second slice index offset, with implicit value 1.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

Operation

The description of MOVA (tile to vector, two registers) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (tile to vector, two registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.167 MOV (tile to vector, four registers)

Move ZA four-slice operand to Z four-vector operand

This instruction operates on a horizontal or vertical ZA four-slice operand within a ZA tile of the specified element size.

The first slice of the four-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 4, modulo the number of slices in the tile.

The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This is an alias of MOVA (tile to vector, four registers). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, four registers).
- The description of MOVA (tile to vector, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs		0	0	1	0	0	0	off2		Zd			0	0

size

Encoding for the 8-bit variant

```
MOV  { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]
```

is equivalent to

```
MOVA  { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]
```

and is always the preferred disassembly.

16-bit

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs		0	0	1	0	0	0	ZAn	o1	Zd			0	0

size

Encoding for the 16-bit variant

```
MOV  { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]
```

is equivalent to

```
MOVA  { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs		0	0	1	0	0	0	ZAn		Zd			0	0

size

Encoding for the 32-bit variant

```
MOV  { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]
```

is equivalent to

```
MOVA  { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]
```

and is always the preferred disassembly.

64-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs		0	0	1	0	0	ZAn			Zd			0	0

size

Encoding for the 64-bit variant

```
MOV  { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]
```

is equivalent to

```
MOVA  { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off2” field times 4.

For the “16-bit” variant: is the first slice index offset, encoded as “o1” field times 4.

For the “32-bit” and “64-bit” variants: is the first slice index offset, with implicit value 0.

<offs4>

For the “8-bit” variant: is the fourth slice index offset, encoded as “off2” field times 4 plus 3.

For the “16-bit” variant: is the fourth slice index offset, encoded as “o1” field times 4 plus 3.

For the “32-bit” and “64-bit” variants: is the fourth slice index offset, with implicit value 3.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

Operation

The description of MOVA (tile to vector, four registers) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (tile to vector, four registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.168 MOV (array to vector, two registers)

Move two ZA single-vector groups to Z two-vector operand

This instruction operates on two ZA single-vector groups.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (array to vector, two registers). This means:

- The encodings in this description are named to match the encodings of MOVA (array to vector, two registers).
- The description of MOVA (array to vector, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	0	0	0	0	0	0		1	1	0	0	Rv		0	1	0		0	0	off3				Zd			0

Encoding

```
MOV  { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
```

is equivalent to

```
MOVA  { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Operation

The description of MOVA (array to vector, two registers) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of MOVA (array to vector, two registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.169 MOV (array to vector, four registers)

Move four ZA single-vector groups to Z four-vector operand

This instruction operates on four ZA single-vector groups.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (array to vector, four registers). This means:

- The encodings in this description are named to match the encodings of MOVA (array to vector, four registers).
- The description of MOVA (array to vector, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SME2

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv		0	1	1	0	0	off3			Zd			0	0

Encoding

```
MOV  { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

is equivalent to

```
MOVA  { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Operation

The description of MOVA (array to vector, four registers) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of MOVA (array to vector, four registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.170 MOV (tile to vector, single)

Move ZA tile slice to Z vector

This instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1. Inactive elements in the destination vector remain unmodified.

This is an alias of MOVA (tile to vector, single). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, single).
- The description of MOVA (tile to vector, single) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit

8-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	8	5	4	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	V	Rs	Pg		0	off4		Zd	
						size							Q									

Encoding for the 8-bit variant

```
MOV  <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]
```

is equivalent to

```
MOVA  <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]
```

and is always the preferred disassembly.

16-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	8	7	5	4	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	V	Rs	Pg		0	ZAn	off3		Zd	
							size						Q										

Encoding for the 16-bit variant

```
MOV  <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]
```

is equivalent to

```
MOVA  <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12		10	9	8	7	6	5	4			0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	V	Rs		Pg			0	ZAn		off2		Zd			
								size								Q															

Encoding for the 32-bit variant

```
MOV  <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]
```

is equivalent to

```
MOVA  <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]
```

and is always the preferred disassembly.

64-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12		10	9	8		6	5	4			0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	V	Rs		Pg			0	ZAn			o1	Zd			
								size								Q															

Encoding for the 64-bit variant

```
MOV  <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]
```

is equivalent to

```
MOVA  <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]
```

and is always the preferred disassembly.

128-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12		10	9	8			5	4			0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	V	Rs		Pg			0	ZAn				Zd			
								size								Q															

Encoding for the 128-bit variant

```
MOV  <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]
```

is equivalent to

```
MOVA  <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

For the “8-bit” variant: is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

For the “16-bit” variant: is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

For the “32-bit” variant: is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

For the “64-bit” variant: is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

For the “128-bit” variant: is the slice index offset, with implicit value 0.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

For the “128-bit” variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAn” field.

Operation

The description of MOVA (tile to vector, single) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (tile to vector, single) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.171 MOV (vector to tile, two registers)

Move Z two-vector operand to ZA two-slice operand

This instruction operates on a horizontal or vertical ZA two-slice operand within a ZA tile of the specified element size.

The first slice of the two-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 2, modulo the number of slices in the tile.

The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This is an alias of MOVA (vector to tile, two registers). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, two registers).
- The description of MOVA (vector to tile, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs		0	0	0	Zn			0	0	0	off3		
								size																						

Encoding for the 8-bit variant

```
MOV  ZA0<HV>.B[<Ws>, <offs1>:<offs2>], { <Zn1>.B-<Zn2>.B }
```

is equivalent to

```
MOVA  ZA0<HV>.B[<Ws>, <offs1>:<offs2>], { <Zn1>.B-<Zn2>.B }
```

and is always the preferred disassembly.

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs		0	0	0	Zn			0	0	0	ZAd	off2	
								size																						

Encoding for the 16-bit variant

```
MOV  <ZAd><HV>.H[<Ws>, <offs1>:<offs2>], { <Zn1>.H-<Zn2>.H }
```

is equivalent to

```
MOVA  <ZAd><HV>.H[<Ws>, <offs1>:<offs2>], { <Zn1>.H-<Zn2>.H }
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	V	Rs		0	0	Zn		0	0	0	ZAd		o1

size

Encoding for the 32-bit variant

```
MOV  <ZAd><HV>.S[<Ws>, <offs1>:<offs2>], { <Zn1>.S-<Zn2>.S }
```

is equivalent to

```
MOVA  <ZAd><HV>.S[<Ws>, <offs1>:<offs2>], { <Zn1>.S-<Zn2>.S }
```

and is always the preferred disassembly.

64-bit

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	0	1	0	0	V	Rs		0	0	Zn		0	0	0	ZAd	

size

Encoding for the 64-bit variant

```
MOV  <ZAd><HV>.D[<Ws>, <offs1>:<offs2>], { <Zn1>.D-<Zn2>.D }
```

is equivalent to

```
MOVA  <ZAd><HV>.D[<Ws>, <offs1>:<offs2>], { <Zn1>.D-<Zn2>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off3” field times 2.

For the “16-bit” variant: is the first slice index offset, encoded as “off2” field times 2.

For the “32-bit” variant: is the first slice index offset, encoded as “o1” field times 2.

For the “64-bit” variant: is the first slice index offset, with implicit value 0.

<offs2>

For the “8-bit” variant: is the second slice index offset, encoded as “off3” field times 2 plus 1.

For the “16-bit” variant: is the second slice index offset, encoded as “off2” field times 2 plus 1.

For the “32-bit” variant: is the second slice index offset, encoded as “o1” field times 2 plus 1.

For the “64-bit” variant: is the second slice index offset, with implicit value 1.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<ZAd>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAd” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAd” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAd” field.

Operation

The description of MOVA (vector to tile, two registers) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (vector to tile, two registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.172 MOV (vector to tile, four registers)

Move Z four-vector operand to ZA four-slice operand

This instruction operates on a horizontal or vertical ZA four-slice operand within a ZA tile of the specified element size.

The first slice of the four-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 4, modulo the number of slices in the tile.

The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This is an alias of MOVA (vector to tile, four registers). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, four registers).
- The description of MOVA (vector to tile, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs		0	0	1	Zn			0	0	0	0	0	off2	

size

Encoding for the 8-bit variant

```
MOV  ZA0<HV>.B[<Ws>, <offs1>:<offs4>], { <Zn1>.B-<Zn4>.B }
```

is equivalent to

```
MOVA  ZA0<HV>.B[<Ws>, <offs1>:<offs4>], { <Zn1>.B-<Zn4>.B }
```

and is always the preferred disassembly.

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs		0	0	1	Zn			0	0	0	0	0	ZAd	o1

size

Encoding for the 16-bit variant

```
MOV  <ZAd><HV>.H[<Ws>, <offs1>:<offs4>], { <Zn1>.H-<Zn4>.H }
```

is equivalent to

```
MOVA  <ZAd><HV>.H[<Ws>, <offs1>:<offs4>], { <Zn1>.H-<Zn4>.H }
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME2)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9			7	6	5	4	3	2	1	0
1	1	0	0	0	0	0		0	1	0	0	0	0		1	0	0	V	Rs		0	0	1		Zn				0	0	0	0	0	ZAd	

size

Encoding for the 32-bit variant

```
MOV  <ZAd><HV>.S[<Ws>, <offs1>:<offs4>], { <Zn1>.S-<Zn4>.S }
```

is equivalent to

```
MOVA  <ZAd><HV>.S[<Ws>, <offs1>:<offs4>], { <Zn1>.S-<Zn4>.S }
```

and is always the preferred disassembly.

64-bit

(FEAT_SME2)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0		0	1	1	0	0	0		1	0	0	V	Rs		0	0	1		Zn				0	0	0	0	ZAd		

size

Encoding for the 64-bit variant

```
MOV  <ZAd><HV>.D[<Ws>, <offs1>:<offs4>], { <Zn1>.D-<Zn4>.D }
```

is equivalent to

```
MOVA  <ZAd><HV>.D[<Ws>, <offs1>:<offs4>], { <Zn1>.D-<Zn4>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off2” field times 4.

For the “16-bit” variant: is the first slice index offset, encoded as “o1” field times 4.

For the “32-bit” and “64-bit” variants: is the first slice index offset, with implicit value 0.

<offs4>

For the “8-bit” variant: is the fourth slice index offset, encoded as “off2” field times 4 plus 3.

For the “16-bit” variant: is the fourth slice index offset, encoded as “o1” field times 4 plus 3.

For the “32-bit” and “64-bit” variants: is the fourth slice index offset, with implicit value 3.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<ZAd>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAd” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAd” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAd” field.

Operation

The description of MOVA (vector to tile, four registers) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (vector to tile, four registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.173 MOV (vector to array, two registers)

Move Z two-vector operand to two ZA single-vector groups

This instruction operates on two ZA single-vector groups.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (vector to array, two registers). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to array, two registers).
- The description of MOVA (vector to array, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SME2

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	0	Zn				0	0	0	off3		

Encoding

```
MOV  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }
```

is equivalent to

```
MOVA  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

The description of MOVA (vector to array, two registers) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (vector to array, two registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.174 MOV (vector to array, four registers)

Move Z four-vector operand to four ZA single-vector groups

This instruction operates on four ZA single-vector groups.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (vector to array, four registers). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to array, four registers).
- The description of MOVA (vector to array, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	1	Zn			0	0	0	0	off3		

Encoding

```
MOV  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }
```

is equivalent to

```
MOVA  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

The description of MOVA (vector to array, four registers) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (vector to array, four registers) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.175 MOV (vector to tile, single)

Move Z vector to ZA tile slice

This instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1. Inactive elements in the destination slice remain unmodified.

This is an alias of MOVA (vector to tile, single). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, single).
- The description of MOVA (vector to tile, single) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit

8-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	5	4	3	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	V	Rs	Pg		Zn		0	off4	
							size						Q									

Encoding for the 8-bit variant

```
MOV  ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B
```

is equivalent to

```
MOVA  ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B
```

and is always the preferred disassembly.

16-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	V	Rs	Pg		Zn	0	ZAd	off3	
								size						Q									

Encoding for the 16-bit variant

```
MOV  <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H
```

is equivalent to

```
MOVA  <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12		10	9			5	4	3		2	1		0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	V	Rs		Pg			Zn				0	ZAd			off2		
								size								Q																	

Encoding for the 32-bit variant

```
MOV  <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S
```

is equivalent to

```
MOVA  <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S
```

and is always the preferred disassembly.

64-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12		10	9			5	4	3		1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	V	Rs		Pg			Zn				0	ZAd			o1
								size								Q															

Encoding for the 64-bit variant

```
MOV  <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D
```

is equivalent to

```
MOVA  <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D
```

and is always the preferred disassembly.

128-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12		10	9			5	4	3		0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	V	Rs		Pg			Zn				0	ZAd		
								size								Q														

Encoding for the 128-bit variant

```
MOV  <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q
```

is equivalent to

```
MOVA  <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

For the “8-bit” variant: is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

For the “16-bit” variant: is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

For the “32-bit” variant: is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

For the “64-bit” variant: is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

For the “128-bit” variant: is the slice index offset, with implicit value 0.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<ZAd>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAd” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAd” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAd” field.

For the “128-bit” variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAd” field.

Operation

The description of MOVA (vector to tile, single) gives the operational pseudocode for this instruction.

Operational Information

The description of MOVA (vector to tile, single) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.176 MOVA (tile to vector, two registers)

Move ZA two-slice operand to Z two-vector operand

This instruction operates on a horizontal or vertical ZA two-slice operand within a ZA tile of the specified element size.

The first slice of the two-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 2, modulo the number of slices in the tile.

The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias MOV (tile to vector, two registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7			5	4			1	0
1	1	0	0	0	0	0	0	0	0	0	0	0		1	1	0	V	Rs		0	0	0		0	0	off3				Zd			0

size

Encoding

```
MOVA  { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 8;
let d : integer = UInt(Zd::'0');
let n : integer = 0;
let offset : integer = UInt(off3::'0');
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	0	0	1	0	0	0		1	1	0	V	Rs		0	0	0		0	0	ZAn	off2			Zd			0

size

Encoding

```
MOVA  { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 16;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(ZAn);
let offset : integer = UInt(off2::'0');
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	8	7	6	5	4	1	0
1	1	0	0 0 0	0	0	1	0	0 0	0	1	1	0	V	Rs		0 0	0	0	0	ZAn		o1	Zd		0

size

Encoding

```
MOVA  { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 32;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(ZAn);
let offset : integer = UInt(o1::'0');
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	8	7	5	4	1	0
1	1	0	0 0 0	0	0	1	1	0 0	0	1	1	0	V	Rs		0 0	0	0	0	ZAn		Zd		0

size

Encoding

```
MOVA  { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 64;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(ZAn);
let offset : integer = 0;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off3” field times 2.

For the “16-bit” variant: is the first slice index offset, encoded as “off2” field times 2.

For the “32-bit” variant: is the first slice index offset, encoded as “o1” field times 2.

For the “64-bit” variant: is the first slice index offset, with implicit value 0.

<offs2>

For the “8-bit” variant: is the second slice index offset, encoded as “off3” field times 2 plus 1.

For the “16-bit” variant: is the second slice index offset, encoded as “off2” field times 2 plus 1.

For the “32-bit” variant: is the second slice index offset, encoded as “o1” field times 2 plus 1.

For the “64-bit” variant: is the second slice index offset, with implicit value 1.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

Alias Conditions

Alias	Is preferred when
MOV (tile to vector, two registers)	Unconditionally

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
if nreg == 4 && esize == 64 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let slices : integer = VL DIV esize;
let index : bits(32) = X{}(s);
let slice : integer = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAslice{}(n, esize, vertical, slice + r);
    Z{VL}(d + r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.177 MOVA (tile to vector, four registers)

Move ZA four-slice operand to Z four-vector operand

This instruction operates on a horizontal or vertical ZA four-slice operand within a ZA tile of the specified element size.

The first slice of the four-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 4, modulo the number of slices in the tile.

The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias MOV (tile to vector, four registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0		1	1	0	V	Rs		0	0	1		0	0	0	off2		Zd			0	0

size

Encoding

```
MOVA  { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 8;
let d : integer = UInt(Zd::'00');
let n : integer = 0;
let offset : integer = UInt(off2::'00');
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0		1	1	0	V	Rs		0	0	1		0	0	0	ZAn	o1	Zd			0	0

size

Encoding

```
MOVA  { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 16;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(ZAn);
let offset : integer = UInt(o1::'00');
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs		0	0	1	0	0	0	ZAn		Zd			0	0

size

Encoding

```
MOVA  { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 32;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(ZAn);
let offset : integer = 0;
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs		0	0	1	0	0	ZAn			Zd			0	0

size

Encoding

```
MOVA  { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 64;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(ZAn);
let offset : integer = 0;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off2” field times 4.

For the “16-bit” variant: is the first slice index offset, encoded as “o1” field times 4.

For the “32-bit” and “64-bit” variants: is the first slice index offset, with implicit value 0.

<offs4>

For the “8-bit” variant: is the fourth slice index offset, encoded as “off2” field times 4 plus 3.

For the “16-bit” variant: is the fourth slice index offset, encoded as “o1” field times 4 plus 3.

For the “32-bit” and “64-bit” variants: is the fourth slice index offset, with implicit value 3.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

Alias Conditions

Alias	Is preferred when
`MOV (tile to vector, four registers)`	`Unconditionally`

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
if nreg == 4 && esize == 64 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let slices : integer = VL DIV esize;
let index : bits(32) = X{}(s);
let slice : integer = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAslice{}(n, esize, vertical, slice + r);
    Z{VL}(d + r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.178 MOVA (array to vector, two registers)

Move two ZA single-vector groups to Z two-vector operand

This instruction operates on two ZA single-vector groups.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (array to vector, two registers).

SME2

(FEAT_SME2)

31	30 29	28 25	24	23 22	21 19	18	17	16 15	14 13	12 10	9 8	7 5	4 1	0
1	1 0	0 0 0 0	0	0 0	0 0 0	1	1	0 0	Rv	0 1 0	0 0	off3	Zd	0

Encoding

```
MOVA  { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (array to vector, two registers)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAvector{}(vec);
    Z{VL}(d + r) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.179 MOVA (array to vector, four registers)

Move four ZA single-vector groups to Z four-vector operand

This instruction operates on four ZA single-vector groups.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (array to vector, four registers).

SME2

(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv		0	1	1	0	0	off3			Zd			0	0

Encoding

```
MOVA  { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (array to vector, four registers)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAvector{}(vec);
    Z{VL}(d + r) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.180 MOVA (tile to vector, single)

Move ZA tile slice to Z vector

This instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This instruction is used by the alias MOV (tile to vector, single).

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit

8-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	8	5	4	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	V	Rs	Pg		0	off4		Zd	
							size						Q									

Encoding

```
MOVA  <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = 0;
let offset : integer = UInt(off4);
let esize : integer{} = 8;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	8	7	5	4	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	V	Rs	Pg		0	ZAn	off3		Zd	
							size						Q										

Encoding

```
MOVA  <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = UInt(off3);
let esize : integer{} = 16;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8		7	6		5	4				0
1	1	0	0	0	0	0	0	1	0	0	0	0		0	1	0	V	Rs			Pg			0	ZAn			off2			Zd			
								size								Q																			

Encoding

```
MOVA  <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = UInt(off2);
let esize : integer{} = 32;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			6	5	4				0
1	1	0	0	0	0	0	0	1	1	0	0	0		0	1	0	V	Rs			Pg			0	ZAn				o1	Zd				
								size								Q																		

Encoding

```
MOVA  <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = UInt(o1);
let esize : integer{} = 64;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

128-bit

(FEAT_SME)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4				0
1	1	0	0	0	0	0		0	1	1	0	0	0		0	1	1	V	Rs		Pg				0	ZAn				Zd				
									size								Q																	

Encoding

```
MOVA  <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = 0;
let esize : integer{} = 128;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

For the “8-bit” variant: is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

For the “16-bit” variant: is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

For the “32-bit” variant: is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

For the “64-bit” variant: is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

For the “128-bit” variant: is the slice index offset, with implicit value 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

For the “128-bit” variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAn” field.

Alias Conditions

Alias	Is preferred when
MOV (tile to vector, single)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
let operand : bits(VL) = ZAslice{}(n, esize, vertical, slice);
var result : bits(VL) = Z{}(d);

for e = 0 to dim-1 do
    let element : bits(esize) = operand[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element;
    end;
end;

Z{VL}(d) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.181 MOVA (vector to tile, two registers)

Move Z two-vector operand to ZA two-slice operand

This instruction operates on a horizontal or vertical ZA two-slice operand within a ZA tile of the specified element size.

The first slice of the two-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 2, modulo the number of slices in the tile.

The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to tile, two registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs		0	0	0	Zn			0	0	0	off3		

size

Encoding

```
MOVA  ZA0<HV>.B[<Ws>, <offs1>:<offs2>], { <Zn1>.B-<Zn2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 8;
let n : integer = UInt(Zn::'0');
let d : integer = 0;
let offset : integer = UInt(off3::'0');
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs		0	0	0	Zn			0	0	0	ZAd	off2	

size

Encoding

```
MOVA  <ZAd><HV>.H[<Ws>, <offs1>:<offs2>], { <Zn1>.H-<Zn2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(ZAd);
let offset : integer = UInt(off2::'0');
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	V	Rs		0	0	0	Zn			0	0	0	ZAd		o1

size

Encoding

```
MOVA  <ZAd><HV>.S[<Ws>, <offs1>:<offs2>], { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(ZAd);
let offset : integer = UInt(o1::'0');
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	V	Rs		0	0	0	Zn			0	0	0	ZAd		

size

Encoding

```
MOVA  <ZAd><HV>.D[<Ws>, <offs1>:<offs2>], { <Zn1>.D-<Zn2>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 64;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(ZAd);
let offset : integer = 0;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off3” field times 2.

For the “16-bit” variant: is the first slice index offset, encoded as “off2” field times 2.

For the “32-bit” variant: is the first slice index offset, encoded as “o1” field times 2.

For the “64-bit” variant: is the first slice index offset, with implicit value 0.

<offs2>

For the “8-bit” variant: is the second slice index offset, encoded as “off3” field times 2 plus 1.

For the “16-bit” variant: is the second slice index offset, encoded as “off2” field times 2 plus 1.

For the “32-bit” variant: is the second slice index offset, encoded as “o1” field times 2 plus 1.

For the “64-bit” variant: is the second slice index offset, with implicit value 1.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<ZAd>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAd” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAd” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAd” field.

Alias Conditions

Alias	Is preferred when
MOV (vector to tile, two registers)	Unconditionally

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
if nreg == 4 && esize == 64 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let slices : integer = VL DIV esize;
let index : bits(32) = X{}(s);
let slice : integer = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1 do
    let result : bits(VL) = Z{}(n + r);
    ZAslice{VL}(d, esize, vertical, slice + r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.182 MOVA (vector to tile, four registers)

Move Z four-vector operand to ZA four-slice operand

This instruction operates on a horizontal or vertical ZA four-slice operand within a ZA tile of the specified element size.

The first slice of the four-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 4, modulo the number of slices in the tile.

The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to tile, four registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs		0	0	1	Zn			0	0	0	0	0	off2	

size

Encoding

```
MOVA  ZA0<HV>.B[<Ws>, <offs1>:<offs4>], { <Zn1>.B-<Zn4>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 8;
let n : integer = UInt(Zn::'00');
let d : integer = 0;
let offset : integer = UInt(off2::'00');
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs		0	0	1	Zn			0	0	0	0	0	ZAd	o1

size

Encoding

```
MOVA  <ZAd><HV>.H[<Ws>, <offs1>:<offs4>], { <Zn1>.H-<Zn4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(ZAd);
let offset : integer = UInt(o1::'00');
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	V	Rs		0	0	1	Zn			0	0	0	0	0	ZAd	

size

Encoding

```
MOVA  <ZAd><HV>.S[<Ws>, <offs1>:<offs4>], { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(ZAd);
let offset : integer = 0;
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	V	Rs		0	0	1	Zn			0	0	0	0	ZAd		

size

Encoding

```
MOVA  <ZAd><HV>.D[<Ws>, <offs1>:<offs4>], { <Zn1>.D-<Zn4>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(ZAd);
let offset : integer = 0;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off2” field times 4.

For the “16-bit” variant: is the first slice index offset, encoded as “o1” field times 4.

For the “32-bit” and “64-bit” variants: is the first slice index offset, with implicit value 0.

<offs4>

For the “8-bit” variant: is the fourth slice index offset, encoded as “off2” field times 4 plus 3.

For the “16-bit” variant: is the fourth slice index offset, encoded as “o1” field times 4 plus 3.

For the “32-bit” and “64-bit” variants: is the fourth slice index offset, with implicit value 3.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<ZAd>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAd” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAd” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAd” field.

Alias Conditions

Alias	Is preferred when
MOV (vector to tile, four registers)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
if nreg == 4 && esize == 64 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let slices : integer = VL DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let index : bits(32) = X{}(s);
let slice : integer = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1 do
    let result : bits(VL) = Z{}(n + r);
    ZAslice{VL}(d, esize, vertical, slice + r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.183 MOVA (vector to array, two registers)

Move Z two-vector operand to two ZA single-vector groups

This instruction operates on two ZA single-vector groups.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to array, two registers).

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9		6	5	4	3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	0	Zn			0	0	0	off3		

Encoding

```
MOVA  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let n : integer = UInt(Zn::'0');
let nreg : integer{} = 2;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (vector to array, two registers)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1 do
    let result : bits(VL) = Z{}(n + r);
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.184 MOVA (vector to array, four registers)

Move Z four-vector operand to four ZA single-vector groups

This instruction operates on four ZA single-vector groups.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to array, four registers).

SME2

(FEAT_SME2)

31	30 29	28 25	24	23 22	21 19	18	17	16 15	14 13	12 10	9 7	6 5	4	3	2 0
1	1 0	0 0 0 0	0	0 0	0 0 0	1	0	0 0	Rv	0 1 1	Zn	0 0	0	0	off3

Encoding

```
MOVA  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let n : integer = UInt(Zn::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
MOV (vector to array, four registers)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1 do
    let result : bits(VL) = Z{}(n + r);
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.185 MOVA (vector to tile, single)

Move Z vector to ZA tile slice

This instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This instruction is used by the alias MOV (vector to tile, single).

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit

8-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	5	4	3	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	V	Rs		Pg		Zn	0	off4	
							size						Q									

Encoding

```
MOVA  ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(Zn);
let d : integer = 0;
let offset : integer = UInt(off4);
let esize : integer{} = 8;
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	19	18	17	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	V	Rs		Pg		Zn	0	ZAd	off3
								size						Q									

Encoding

```
MOVA  <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(Zn);
let d : integer = UInt(ZAd);
let offset : integer = UInt(off3);
let esize : integer{} = 16;
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	V	Rs		Pg		Zn		0	ZAd		off2	
								size							Q												

Encoding

```
MOVA  <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(Zn);
let d : integer = UInt(ZAd);
let offset : integer = UInt(off2);
let esize : integer{} = 32;
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12	10	9	5	4	3	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	V	Rs		Pg		Zn		0	ZAd		o1
								size							Q											

Encoding

```
MOVA  <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(Zn);
let d : integer = UInt(ZAd);
let offset : integer = UInt(o1);
let esize : integer{} = 64;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

128-bit

(FEAT_SME)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9				5	4	3				0
1	1	0	0	0	0	0		0	1	1	0	0	0		0	0	1	V	Rs		Pg				Zn					0	ZAd				
									size								Q																		

Encoding

```
MOVA  <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let g : integer = UInt(Pg);
let s : integer = UInt('011'::Rs);
let n : integer = UInt(Zn);
let d : integer = UInt(ZAd);
let offset : integer = 0;
let esize : integer{} = 128;
let vertical : boolean = V == '1';
```

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

For the “8-bit” variant: is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

For the “16-bit” variant: is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

For the “32-bit” variant: is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

For the “64-bit” variant: is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

For the “128-bit” variant: is the slice index offset, with implicit value 0.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<ZAd>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAd” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAd” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAd” field.

For the “128-bit” variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAd” field.

Alias Conditions

Alias	Is preferred when
MOV (vector to tile, single)	Unconditionally

Operation

```
CheckStreamingSVEAndZAEnabled();
let  VL : integer{} = CurrentVL();
let  PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
let mask : bits(PL) = P{}(g);
let operand : bits(VL) = Z{}(n);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var result : bits(VL) = ZAslice{}(d, esize, vertical, slice);

for e = 0 to dim-1 do
    let element : bits(esize) = operand[e*:esize];
    if ActivePredicateElement{PL}(mask, e, esize) then
        result[e*:esize] = element;
    end;
end;

ZAslice{VL}(d, esize, vertical, slice) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.186 MOVAZ (tile to vector, two registers)

Move and zero ZA two-slice operand to Z two-vector operand

This instruction operates on a horizontal or vertical ZA two-slice operand within a ZA tile of the specified element size. The tile slices are zeroed after moving their contents to the destination vectors.

The first slice of the two-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 2, modulo the number of slices in the tile.

The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs		0	0	0	1	0	off3			Zd				0

size

Encoding

```
MOVAZ  { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 8;
let d : integer = UInt(Zd::'0');
let n : integer = 0;
let offset : integer = UInt(off3::'0');
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs		0	0	0	1	0	ZAn	off2		Zd				0

size

Encoding

```
MOVAZ  { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 16;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(ZAn);
let offset : integer = UInt(off2::'0');
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7	6	5	4	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs		0	0	0	1	0	ZAn		o1	Zd		0

size

Encoding

```
MOVAZ  { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 32;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(ZAn);
let offset : integer = UInt(o1::'0');
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs		0	0	0	1	0	ZAn			Zd		0

size

Encoding

```
MOVAZ  { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 2;
let esize : integer{} = 64;
let d : integer = UInt(Zd::'0');
let n : integer = UInt(ZAn);
let offset : integer = 0;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off3” field times 2.

For the “16-bit” variant: is the first slice index offset, encoded as “off2” field times 2.

For the “32-bit” variant: is the first slice index offset, encoded as “o1” field times 2.

For the “64-bit” variant: is the first slice index offset, with implicit value 0.

<offs2>

For the “8-bit” variant: is the second slice index offset, encoded as “off3” field times 2 plus 1.

For the “16-bit” variant: is the second slice index offset, encoded as “off2” field times 2 plus 1.

For the “32-bit” variant: is the second slice index offset, encoded as “o1” field times 2 plus 1.

For the “64-bit” variant: is the second slice index offset, with implicit value 1.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
if nreg == 4 && esize == 64 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let slices : integer = VL DIV esize;
let index : bits(32) = X{}(s);
let slice : integer = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAslice{}(n, esize, vertical, slice + r);
    ZAslice{VL}(n, esize, vertical, slice + r) = Zeros{VL};
    Z{VL}(d + r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.187 MOVAZ (tile to vector, four registers)

Move and zero ZA four-slice operand to Z four-vector operand

This instruction operates on a horizontal or vertical ZA four-slice operand within a ZA tile of the specified element size. The tile slices are zeroed after moving their contents to the destination vectors.

The first slice of the four-slice operand is selected by rounding down the sum of the slice index register and the immediate offset to the nearest lower multiple of 4, modulo the number of slices in the tile.

The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit, and 64-bit

8-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0		1	1	0	V	Rs		0	0	1		1	0	0	off2		Zd			0	0

size

Encoding

```
MOVAZ  { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 8;
let d : integer = UInt(Zd::'00');
let n : integer = 0;
let offset : integer = UInt(off2::'00');
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0		1	1	0	V	Rs		0	0	1		1	0	0	ZAn	o1	Zd			0	0

size

Encoding

```
MOVAZ  { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 16;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(ZAn);
let offset : integer = UInt(o1::'00');
let vertical : boolean = V == '1';
```

32-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs		0	0	1	1	0	0	ZAn		Zd			0	0

size

Encoding

```
MOVAZ  { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 32;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(ZAn);
let offset : integer = 0;
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21		19	18	17	16	15	14	13	12		10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs		0	0	1	1	0	ZAn			Zd			0	0

size

Encoding

```
MOVAZ  { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let nreg : integer{} = 4;
let esize : integer{} = 64;
let d : integer = UInt(Zd::'00');
let n : integer = UInt(ZAn);
let offset : integer = 0;
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs1>

For the “8-bit” variant: is the first slice index offset, encoded as “off2” field times 4.

For the “16-bit” variant: is the first slice index offset, encoded as “o1” field times 4.

For the “32-bit” and “64-bit” variants: is the first slice index offset, with implicit value 0.

<offs4>

For the “8-bit” variant: is the fourth slice index offset, encoded as “off2” field times 4 plus 3.

For the “16-bit” variant: is the fourth slice index offset, encoded as “o1” field times 4 plus 3.

For the “32-bit” and “64-bit” variants: is the fourth slice index offset, with implicit value 3.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
if nreg == 4 && esize == 64 && VL < 256 then EndOfDecode(Decode_UNDEF); end;
let slices : integer = VL DIV esize;
let index : bits(32) = X{}(s);
let slice : integer = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAslice{}(n, esize, vertical, slice + r);
    ZAslice{VL}(n, esize, vertical, slice + r) = Zeros{VL};
    Z{VL}(d + r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.188 MOVAZ (array to vector, two registers)

Move and zero two ZA single-vector groups to Z two-vector operand

This instruction operates on two ZA single-vector groups. The ZA single-vector groups are zeroed after moving their contents to the destination vectors.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME2p1)

31	30 29	28 25	24	23 22	21 19	18	17	16 15	14 13	12 10	9 8	7 5	4 1	0
1	1 0	0 0 0 0	0	0 0	0 0 0	1	1	0 0	Rv	0 1 0	1 0	off3	Zd	0

Encoding

```
MOVAZ  { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAvector{}(vec);
    ZAvector{VL}(vec) = Zeros{VL};
    Z{VL}(d + r) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.189 MOVAZ (array to vector, four registers)

Move and zero four ZA single-vector groups to Z four-vector operand

This instruction operates on four ZA single-vector groups. The ZA single-vector groups are zeroed after moving their contents to the destination vectors.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The `vector group` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME2p1)

31	30 29	28 25	24	23 22	21 19	18	17	16 15	14 13	12 10	9 8	7 5	4 2	1 0
1	1 0	0 0 0 0	0	0 0	0 0 0	1	1	0 0	Rv	0 1 1	1 0	off3	Zd	0 0

Encoding

```
MOVAZ  { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1 do
    let result : bits(VL) = ZAvector{}(vec);
    ZAvector{VL}(vec) = Zeros{VL};
    Z{VL}(d + r) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.190 MOVAZ (tile to vector, single)

Move and zero ZA tile slice to Z vector

This instruction operates on a horizontal or vertical slice within a named ZA tile of the specified element size. The tile slice is zeroed after moving its contents to the destination vector.

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

This instruction is unpredicated.

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit

8-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4			0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	V	Rs		0	0	0		1	off4				Zd		
								size								Q															

Encoding

```
MOVAZ  <Zd>.B, ZA0<HV>.B[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let n : integer = 0;
let offset : integer = UInt(off4);
let esize : integer{} = 8;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

16-bit

(FEAT_SME2p1)

31	30	29	28			25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7		5	4			0
1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	V	Rs		0	0	0		1	ZAn	off3			Zd		
								size								Q															

Encoding

```
MOVAZ  <Zd>.H, <ZAn><HV>.H[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = UInt(off3);
let esize : integer{} = 16;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit

(FEAT_SME2p1)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8	7	6	5	4				0
1	1	0	0	0	0	0		0	1	0	0	0	0		0	1	0	V	Rs		0	0	0		1	ZAn		off2		Zd				
									size								Q																	

Encoding

```
MOVAZ  <Zd>.S, <ZAn><HV>.S[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = UInt(off2);
let esize : integer{} = 32;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

64-bit

(FEAT_SME2p1)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			6	5	4				0
1	1	0	0	0	0	0		0	1	1	0	0	0		0	1	0	V	Rs		0	0	0		1	ZAn				o1	Zd			
									size								Q																	

Encoding

```
MOVAZ  <Zd>.D, <ZAn><HV>.D[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = UInt(o1);
let esize : integer{} = 64;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

128-bit

(FEAT_SME2p1)

31	30	29	28				25	24	23	22	21			19	18	17	16	15	14	13	12			10	9	8			5	4			0
1	1	0	0	0	0	0		0	1	1	0	0	0		0	1	1	V	Rs		0	0	0		1	ZAn				Zd			
									size								Q																

Encoding

```
MOVAZ  <Zd>.Q, <ZAn><HV>.Q[<Ws>, <offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let s : integer = UInt('011'::Rs);
let n : integer = UInt(ZAn);
let offset : integer = 0;
let esize : integer{} = 128;
let d : integer = UInt(Zd);
let vertical : boolean = V == '1';
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

For the “8-bit” variant: is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

For the “16-bit” variant: is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

For the “32-bit” variant: is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

For the “64-bit” variant: is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

For the “128-bit” variant: is the slice index offset, with implicit value 0.

<ZAn>

For the “16-bit” variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAn” field.

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAn” field.

For the “128-bit” variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAn” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
let operand : bits(VL) = ZAslice{}(n, esize, vertical, slice);
ZAslice{VL}(n, esize, vertical, slice) = Zeros{VL};
Z{VL}(d) = operand;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.191 MOVT (table to scalar)

Move 8 bytes from ZT0 to general-purpose register

This instruction moves 8 bytes to a general-purpose register from the ZT0 register at the byte offset specified by the immediate index. This instruction is UNDEFINED in Non-debug state.

SME2

(FEAT_SME2)

31	30	29	28			25	24						18	17	16	15	14		12	11						5	4		0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	off3			0	0	1	1	1	1	1	Rt		
																							opc						

Encoding

```
MOVT  <Xt>, ZT0[<offs>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !Halted() then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let offset : integer = UInt(off3);
```

Assembler Symbols

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<offs>

Is the immediate byte offset, a multiple of 8 in the range of 0 to 56, encoded in the “off3” field as <offs>/8.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
let operand : bits(512) = ZT0{}();

X{64}(t) = operand[offset*:64];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.192 MOVT (scalar to table)

Move 8 bytes from general-purpose register to ZT0

This instruction moves 8 bytes to the ZT0 register at the byte offset specified by the immediate index from a general-purpose register. This instruction is UNDEFINED in Non-debug state.

SME2

(FEAT_SME2)

31	30	29	28			25	24						18	17	16	15	14		12	11						5	4		0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	off3			0	0	1	1	1	1	1	Rt		
																							opc						

Encoding

```
MOVT  ZT0[<offs>], <Xt>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) || !Halted() then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Rt);
let offset : integer = UInt(off3);
```

Assembler Symbols

<offs>

Is the immediate byte offset, a multiple of 8 in the range of 0 to 56, encoded in the “off3” field as <offs>/8.

<Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
var result : bits(512) = ZT0{}();

result[offset*:64] = X{64}(t);
ZT0{512}() = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.193 MOVT (vector to table)

Move vector register to ZT0

This instruction copies the source vector register to ZT0 at the vector length offset specified by the immediate index. When the index is zero, the instruction writes zeroes to the most significant (512-VL) bits of the ZT0 register. When the index is not zero, the unindexed portions of ZT0 remain unchanged.

This instruction is unpredicated.

SME2

(FEAT_SME_LUTv2)

31	30	29	28			25	24						18	17	16	15	14	13	12	11						5	4				0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	0	off2		0	0	1	1	1	1	1	Zt			

opc

Encoding

```
MOVT  ZT0{[<offs>, MUL VL]}, <Zt>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_LUTv2) then EndOfDecode(Decode_UNDEF); end;
let t : integer = UInt(Zt);
let imm : integer = UInt(off2);
```

Assembler Symbols

<offs>

Is the vector length offset, in the range 0 to 3, defaulting to 0 when omitted, encoded in the “off2” field.

<Zt>

Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
let VL : integer{} = CurrentVL();
let tsize : integer{} = (if VL <= 512 then VL else 512) as integer{128, 256, 512};
let offset : integer = imm MOD (512 DIV tsize);
var result : bits(512) = if imm == 0 then Zeros{512} else ZT0{512}();

result[offset*:tsize] = Z{VL}(t)[tsize-1:0];
ZT0{512}() = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.194 RDSVL

Read multiple of Streaming SVE vector register size to scalar register

This instruction multiplies the Streaming SVE vector register size in bytes by an immediate in the range -32 to 31 and places the result in the 64-bit destination general-purpose register.

This instruction does not require the PE to be in Streaming SVE mode.

SME

(FEAT_SME)

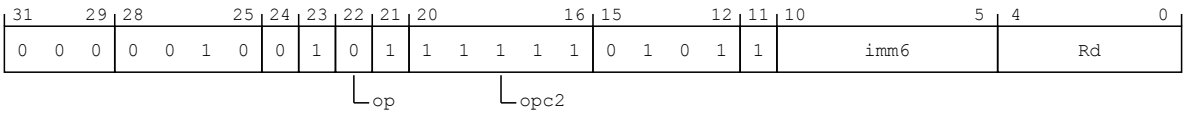

Encoding

```
RDSVL  <Xd>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let d : integer = UInt(Rd);
let imm : integer = SInt(imm6);
```

Assembler Symbols

<Xd>

Is the 64-bit name of the destination general-purpose register, encoded in the “Rd” field.

<imm>

Is the signed immediate operand, in the range -32 to 31, encoded in the “imm6” field.

Operation

```
CheckSMEEnabled();
let SVL : integer = CurrentSVL();
let len : integer = imm * (SVL DIV 8);
X{64}(d) = len[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.195 SCLAMP

Multi-vector signed clamp to minimum/maximum

This instruction clamps each signed element in the two or four destination vectors to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively places the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	0	1	Zn				Zd				0

U

Encoding

```
SCLAMP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	1	1	Zn				Zd			0	0

U

Encoding

```
SCLAMP  { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = Z{}(d+r);
    for e = 0 to elements-1 do
        let element1 : integer = SInt(operand1[e*:esize]);
        let element2 : integer = SInt(operand2[e*:esize]);
        let element3 : integer = SInt(operand3[e*:esize]);
        let res : integer = Min(Max(element1, element3), element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.196 SCVTF

Multi-vector signed 32-bit integer convert to single-precision

This instruction converts each element of the two or four source vectors from signed 32-bit integer to single-precision, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	1	1	0	0	0	Zn		0	Zd		0
																								U			

Encoding

```
SCVTF  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	1	0	1	1	1	0	0	0	Zn		0	0	Zd		0	0
																									U				

Encoding

```
SCVTF  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
let rounding : FPRounding = FPRoundingMode(FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FixedToFP{32, 32}(element, 0, unsigned, FPCR(), rounding);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.197 SDOT (2-way, multiple and indexed vector)

Multi-vector signed 16-bit integer dot product by indexed element to 32-bit integer

This instruction calculates the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2		Zn				0	0	0	off3		
																			op								U				

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn				0	0	0	0	off3		
																			op									U				

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        for i = 0 to 1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * s + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.198 SDOT (2-way, multiple and single vector)

Multi-vector signed 16-bit integer dot product by vector to 32-bit integer

This instruction calculates the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	1	0	1	Zn	0	1	off3	

U

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm	0	Rv	1	0	1	Zn	0	1	off3	

U

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.199 SDOT (2-way, multiple vectors)

Multi-vector signed 16-bit integer dot product to 32-bit integer

This instruction calculates the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	1	Zm		0	0	Rv		1	0	1	Zn	0	0	1	off3

U

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	1	Zm		0	1	0	Rv		1	0	1	Zn		0	0	0	1	off3

U

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.200 SDOT (4-way, multiple and indexed vector)

Multi-vector signed integer dot product by indexed element

This instruction calculates the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors of 32-bit elements, Two ZA single-vectors of 64-bit elements, Four ZA single-vectors of 32-bit elements, and Four ZA single-vectors of 64-bit elements

Two ZA single-vectors of 32-bit elements

(FEAT_SME2)

31	30	29	28–25	24	23	22	21	20	19–16	15	14–13	12	11–10	9–6	5	4	3	2–0
1	1	0	0 0 0 0	1	0	1	0	1	Zm	0	Rv	1	i2	Zn	1	0	0	off3
												op				U		

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Two ZA single-vectors of 64-bit elements

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28–25	24	23	22	21	20	19–16	15	14–13	12	11	10	9–6	5	4	3	2–0
1	1	0	0 0 0 0	1	1	1	0	1	Zm	0	Rv	0	0	i1	Zn	0	0	1	off3
																	U		

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SDOT  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 2;
```

Four ZA single-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn			0	1	0	0	off3		
																			op								U				

Encoding

```
SDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Four ZA single-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		0	0	i1	Zn			0	0	0	1	off3		
																				op							U				

Encoding

```
SDOT  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors of 32-bit elements” and “Two ZA single-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors of 32-bit elements” and “Four ZA single-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA single-vectors of 32-bit elements” and “Two ZA single-vectors of 32-bit elements” variants: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “Four ZA single-vectors of 64-bit elements” and “Two ZA single-vectors of 64-bit elements” variants: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        for i = 0 to 3 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.201 SDOT (4-way, multiple and single vector)

Multi-vector signed integer dot product by vector

This instruction calculates the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm	0	Rv	1	0	1	Zn	0	0	off3	

U

Encoding

```
SDOT  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm	0	Rv	1	0	1	Zn	0	0	off3	

U

Encoding

```
SDOT  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.202 SDOT (4-way, multiple vectors)

Multi-vector signed integer dot product

This instruction calculates the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30 29	28 25	24 23	22	21	20 17	16 15	14 13	12 10	9 6	5	4	3	2 0
1	1 0	0 0 0 0	1 1	sz	1	Zm	0 0	Rv	1 0 1	Zn	0	0	0	off3
												U		

Encoding

```
SDOT  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30 29	28 25	24 23	22	21	20 18	17	16 15	14 13	12 10	9 7	6 5	4	3	2 0
1	1 0	0 0 0 0	1 1	sz	1	Zm	0	1 0	Rv	1 0 1	Zn	0 0	0	0	off3
													U		

Encoding

```
SDOT  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.203 SEL

Multi-vector conditional select

This instruction selects consecutive elements from the two or four first source vectors where vector select predicate elements are true, and places them in the corresponding elements of the two or four destination vectors. Where vector select predicate elements are false, the corresponding consecutive elements are selected from the two or four second source vectors and placed in the remaining elements of the destination vectors.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	13	12	10	9	6	5	4	1	0
1	1	0	0	0	1	size		1	Zm		0	1	0	PNv		Zn		0	Zd		0

Encoding

```
SEL  { <Zd1>.<T>-<Zd2>.<T> }, <PNv>, { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let d : integer = UInt(Zd::'0');
let v : integer = UInt('1'::PNv);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	13	12	10	9	7	6	5	4	2	1	0
1	1	0	0	0	1	size		1	Zm		0	1	1	0	PNv		Zn		0	0	Zd		0	0

Encoding

```
SEL  { <Zd1>.<T>-<Zd4>.<T> }, <PNv>, { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let d : integer = UInt(Zd::'00');
let v : integer = UInt('1'::PNv);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<PNv>

Is the name of the vector select predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNv” field.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
var results : array[[4]] of bits(VL);
let pred : bits(PL) = P{}(v);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            results[[r]][e*:esize] = operand1[e*:esize];
        else
            results[[r]][e*:esize] = operand2[e*:esize];
        end;
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.204 SMAX (multiple and single vector)

Multi-vector signed maximum by vector

This instruction determines the signed maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	1	0
1	1	0	0	0	0	1	size	1	0	Zm		1	0	1	0	0	0	0	0	Zdn		0
																			op			U

Encoding

```
SMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	6	5	4	2	1	0
1	1	0	0	0	0	1	size	1	0	Zm		1	0	1	0	1	0	0	0	Zdn		0	0
																			op				U

Encoding

```
SMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Max(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.205 SMAX (multiple vectors)

Multi-vector signed maximum

This instruction determines the signed maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	0	0		0	0	Zdn				0
																										opc						U

Encoding

```
SMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	0	1	0	1	1	1	0	0	0	0		0	0	Zdn				0	0
																										opc							U

Encoding

```
SMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Max(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.206 SMIN (multiple and single vector)

Multi-vector signed minimum by vector

This instruction determines the signed minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	0	0	0	0	1	Zdn				0
																										op					U

Encoding

```
SMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	1	Zdn				0	0
																										op						U

Encoding

```
SMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Min(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.207 SMIN (multiple vectors)

Multi-vector signed minimum

This instruction determines the signed minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23 22	21	20 17	16 10	9 7	6 5	4 1	0
1	1	0	0 0 0 0		1	size	1	Zm	0 1 0 1 1 0 0	0 0 0	0 1 (opc)	Zdn	0 (U)

Encoding

```
SMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23 22	21	20 18	17 10	9 7	6 5	4 2	1	0
1	1	0	0 0 0 0		1	size	1	Zm	0 0 1 0 1 1 1 0	0 0 0	0 1 (opc)	Zdn	0	0 (U)

Encoding

```
SMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Min(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.208 SMLAL (multiple and indexed vector)

Multi-vector signed 16-bit integer multiply-add by indexed element to 32-bit integer

This instruction multiplies each signed 16-bit element in the one, two, or four first source vectors by each signed 16-bit indexed element of the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	0	0		Zm			i3h		Rv	1		i3l		Zn			0	0		off3	
																										U	S			

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1		Zm			0		Rv	1		i3h		Zn			0	0	0	i3l	off2	
																											U	S			

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	0	1	Zm		1	Rv		1	i3h		Zn		0	0	0	0	i3l	off2	

U S

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[s*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.209 SMLAL (multiple and single vector)

Multi-vector signed 16-bit integer multiply-add by vector to 32-bit integer

This instruction multiplies each signed 16-bit element in the one, two, or four first source vectors by each signed 16-bit element in the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	1	Zn	0	0	off3	
																			U	S		

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	0	Zn	0	0	0	off2	
																			U	S	op		

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		0	1	0	Zn				0	0	0	off2	

U, S, op

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.210 SMLAL (multiple vectors)

Multi-vector signed 16-bit integer multiply-add to 32-bit integer

This instruction multiplies each signed 16-bit element in the two or four first source vectors by each signed 16-bit element in the two or four second source vectors, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	0	Rv		0	1	0	Zn	0	0	0	0	off2	
																				U	S			

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	0	0	0	off2
																						U	S			

Encoding

```
SMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.211 SMLALL (multiple and indexed vector)

Multi-vector signed integer multiply-add long long by indexed element

This instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors by each signed 8-bit or 16-bit indexed element of the second source vector, widens each product to 32 bits or 64 bits, and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Two ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 32-bit elements, and Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l			Zn				0	0	0	off2	

U, S, op

Encoding

```
SMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

One ZA quad-vector of 64-bit elements

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv		0	i3l		Zn				0	0	0	off2	

U, S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SMLALL  ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn				0	0	0	i4l			o1
																										op	U	S				

Encoding

```
SMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Two ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				0	Rv		0	0	i3h	Zn				0	0	0	i3l			o1
																										U		S				

Encoding

```
SMLALL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		0	i4h		Zn			0	0	0	0	i4l		o1

op, U, S

Encoding

```
SMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				1	Rv		0	0	i3h	Zn			0	0	0	0	i3l		o1

U, S

Encoding

```
SMLALL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA quad-vectors of 32-bit elements”, “One ZA quad-vector of 32-bit elements”, and “Two ZA quad-vectors of 32-bit elements” variants: is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

For the “Four ZA quad-vectors of 64-bit elements”, “One ZA quad-vector of 64-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA quad-vectors of 32-bit elements” and “Two ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors of 32-bit elements” and “Four ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[s*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.212 SMLALL (multiple and single vector)

Multi-vector signed integer multiply-add long long by vector

This instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors by each signed 8-bit or 16-bit element in the second source vector, widens each product to 32 bits or 64 bits, and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm			0		Rv	0	0	1		Zn			0	0	0	off2	
																										U	S	op		

Encoding

```
SMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm			0		Rv	0	0	0		Zn			0	0	0	0	o1
																										U	S	op		

Encoding

```
SMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		0	0	0	Zn				0	0	0	0	o1
																										U	S	op		

Encoding

```
SMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.213 SMLALL (multiple vectors)

Multi-vector signed integer multiply-add long long

This instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors by each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32 bits or 64 bits, and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv		0	0	0		Zn				0	0	0	0	0	o1

U, S, op

Encoding

```
SMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12			10	9			7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	1	0	Rv		0	0	0		Zn				0	0	0	0	0	0	o1

U, S, op

Encoding

```
SMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.214 SMLSL (multiple and indexed vector)

Multi-vector signed 16-bit integer multiply-subtract by indexed element from 32-bit integer

This instruction multiplies each signed 16-bit element in the one, two, or four first source vectors by each signed 16-bit indexed element of the second source vector, widens each product to 32 bits, and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	0	0		Zm			i3h		Rv	1		i3l		Zn			0	1		off3	
																										U	S			

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1		Zm			0		Rv	1		i3h		Zn			0	0	1	i3l	off2	
																											U	S			

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		1	i3h		Zn			0	0	0	1	i3l	off2	
																											U	S			

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[s*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.215 SMLSL (multiple and single vector)

Multi-vector signed 16-bit integer multiply-subtract by vector from 32-bit integer

This instruction multiplies each signed 16-bit element in the one, two, or four first source vectors by each signed 16-bit element in the second source vector, widens each product to 32 bits, and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	1	Zn	0	1	off3	
																			U	S		

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	0	Zn	0	1	0	off2	
																			U	S	op		

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		0	1	0	Zn				0	1	0	off2	
																										U	S	op		

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.216 SMLSL (multiple vectors)

Multi-vector signed 16-bit integer multiply-subtract from 32-bit integer

This instruction multiplies each signed 16-bit element in the two or four first source vectors by each signed 16-bit element in the two or four second source vectors, widens each product to 32 bits, and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	0	Rv		0	1	0	Zn	0	0	1	0	off2	
																				U	S			

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	0	1	0	off2
																						U	S			

Encoding

```
SMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.217 SMLSLL (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long long by indexed element

This instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors by each signed 8-bit or 16-bit indexed element of the second source vector, widens each product to 32 bits or 64 bits, and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Two ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 32-bit elements, and Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l			Zn				0	1	0	off2	

U, S, op

Encoding

```
SMLSLL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

One ZA quad-vector of 64-bit elements

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv		0	i3l		Zn				0	1	0	off2	

U, S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
SMLSLL  ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn				0	0	1	i4l		o1

op, U, S

Encoding

```
SMLSLL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Two ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				0	Rv		0	0	i3h	Zn				0	0	1	i3l		o1

U, S

Encoding

```
SMLSLL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	Zm		1	Rv		0	i4h		Zn		0	0	0	1	i4l		o1

op, U, S

Encoding

```
SMLSLL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	0	1	Zm		1	Rv		0	0	i3h	Zn		0	0	0	1	i3l		o1

U, S

Encoding

```
SMLSLL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA quad-vectors of 32-bit elements”, “One ZA quad-vector of 32-bit elements”, and “Two ZA quad-vectors of 32-bit elements” variants: is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

For the “Four ZA quad-vectors of 64-bit elements”, “One ZA quad-vector of 64-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA quad-vectors of 32-bit elements” and “Two ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors of 32-bit elements” and “Four ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[s*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.218 SMLSLL (multiple and single vector)

Multi-vector signed integer multiply-subtract long long by vector

This instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors by each signed 8-bit or 16-bit element in the second source vector, widens each product to 32 bits or 64 bits, and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		0	0	1	Zn				0	1	0	off2	

U, S, op

Encoding

```
SMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm				0	Rv		0	0	0	Zn				0	1	0	0	o1

U, S, op

Encoding

```
SMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		0	0	0	Zn				0	1	0	0	o1

U, S, op

Encoding

```
SMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.219 SMLSLL (multiple vectors)

Multi-vector signed integer multiply-subtract long long

This instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors by each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32 bits or 64 bits, and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	sz	1	Zm		0	0	Rv		0	0	Zn		0	0	1	0	0	o1

U, S, op

Encoding

```
SMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	sz	1	Zm		0	1	0	Rv		0	0	Zn		0	0	0	1	0	0	o1

U, S, op

Encoding

```
SMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.220 SMOP4A (2-way)

Signed 16-bit integer quarter-tile sum of outer products to 32-bit integer, accumulating

This instruction generates four independent quarter-tile signed integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of 16-bit signed values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of 16-bit signed values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products are then destructively added to the 32-bit integer destination tile.

This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	1	Zm			0	1	0	0	0	0	0	0	0	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
SMOP4A  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	Zm			0	1	0	0	0	0	0	0	0	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
SMOP4A  <ZAda>.S, <Zn>.H, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	Zm			0	1	0	0	0	0	0		1	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
SMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	1	Zm			0	1	0	0	0	0	0		1	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
SMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 1 do
                let op1elt : bits(esize DIV 2) = op1[(2*row_idx + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = op2[(2*col_idx + k)*:(esize DIV 2)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.221 SMOP4A (4-way)

Signed integer quarter-tile sum of outer products, accumulating

This instruction generates four independent quarter-tile signed integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit signed values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit signed values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively added to the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	0	1	Zm				0	1	0	0	0	0	0		0	Zn				0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
SMOP4A  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	0	0		Zm			0	1	0	0	0	0	0		0		Zn			0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
SMOP4A  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	0	0		Zm			0	1	0	0	0	0	0		1		Zn			0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
SMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

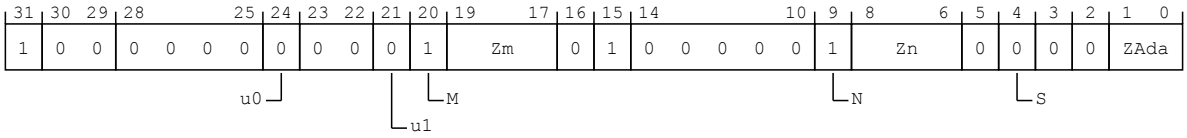

Encoding

```
SMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

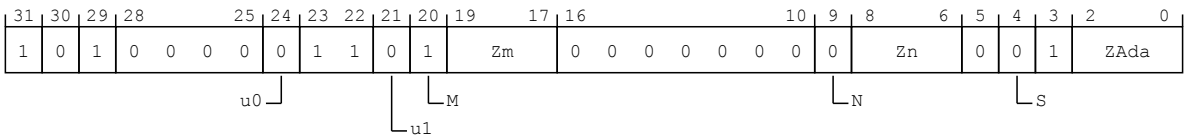

Encoding

```
SMOP4A  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19		17	16							10	9	8		6	5	4	3	2		0
1	0	1	0	0	0	0	0	1	1	0	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	0	1	ZAda		

u0, u1, M, N, S

Encoding

```
SMOP4A  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19		17	16							10	9	8		6	5	4	3	2		0
1	0	1	0	0	0	0	0	1	1	0	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	0	1	ZAda		

u0, u1, M, N, S

Encoding

```
SMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

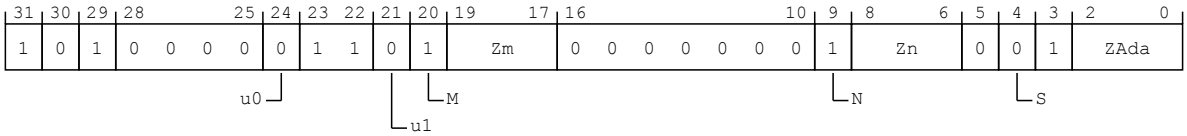

Encoding

```
SMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.222 SMOP4S (2-way)

Signed 16-bit integer quarter-tile sum of outer products to 32-bit integer, subtracting

This instruction generates four independent quarter-tile signed integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of 16-bit signed values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of 16-bit signed values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products are then destructively subtracted from the 32-bit integer destination tile.

This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	0	1		Zm			0	1	0	0	0	0	0		0		Zn			0	1	1	0	ZAda	
								u0				M													N						S				

Encoding

```
SMOP4S  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	0	0		Zm			0	1	0	0	0	0	0		0		Zn			0	1	1	0	ZAda	
								u0				M													N						S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SMOP4S  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	Zm			0	1	0	0	0	0	0		1	Zn			0	1	1	0	ZAda	
							u0				M												N					S				

Encoding

```
SMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	1	Zm			0	1	0	0	0	0	0		1	Zn			0	1	1	0	ZAda	
							u0				M												N					S				

Encoding

```
SMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 1 do
                let op1elt : bits(esize DIV 2) = op1[(2*row_idx + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = op2[(2*col_idx + k)*:(esize DIV 2)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.223 SMOP4S (4-way)

Signed integer quarter-tile sum of outer products, subtracting

This instruction generates four independent quarter-tile signed integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit signed values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit signed values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively subtracted from the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0	0	0	0	0	0	1	Zm		0	1	0 0 0 0	0	0	Zn		0	1	0	0	ZAda	

u0 u1 M N S

Encoding

```
SMOP4S  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	0	Zm		0	1	0 0 0 0 0		0	Zn		0	1	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
SMOP4S  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	0	0	Zm		0	1	0 0 0 0 0		1	Zn		0	1	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
SMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

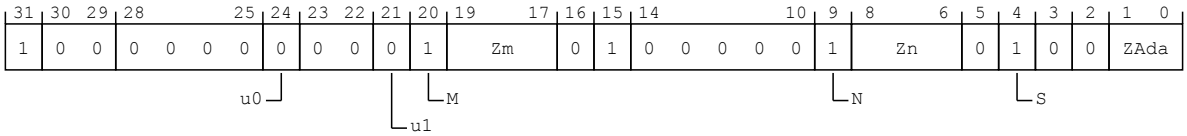

Encoding

```
SMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

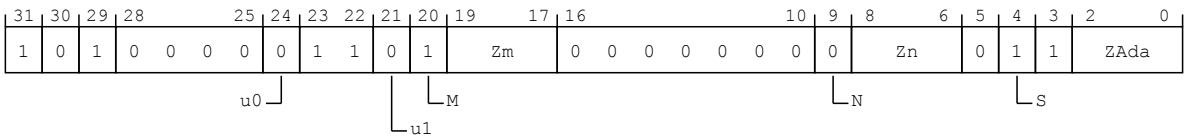

Encoding

```
SMOP4S  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	0	0	Zm		0 0 0 0 0 0	0	0	Zn		0	1	1	ZAda	

u0 u1 M N S

Encoding

```
SMOP4S  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	0	0	Zm		0 0 0 0 0 0	0	1	Zn		0	1	1	ZAda	

u0 u1 M N S

Encoding

```
SMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

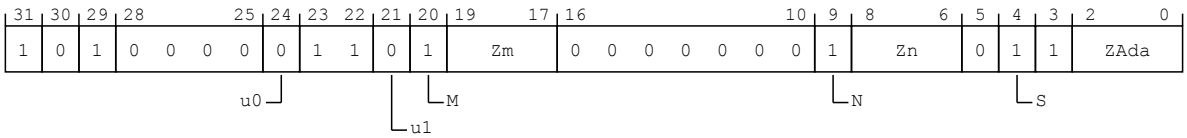

Encoding

```
SMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.224 SMOPA (2-way)

Signed 16-bit integer sum of outer products to 32-bit integer, accumulating

This instruction works with a 32-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of signed 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	0	1	0	0	Zm		Pm		Pn		Zn		0	1	0	ZAda	

u0 S

Encoding

```
SMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 1 do
            if (ActivePredicateElement{PL}(mask1, 2*row + k, esize DIV 2) &&
                  ActivePredicateElement{PL}(mask2, 2*col + k, esize DIV 2)) then
                let op1elt : bits(esize DIV 2) = operand1[(2*row + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = operand2[(2*col + k)*:(esize DIV 2)];
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum + prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.225 SMOPA (4-way)

Signed integer sum of outer products, accumulating

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	0	0	Zm		Pm		Pn		Zn		0	0	0	ZAda	

u0 u1 S

Encoding

```
SMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	0	Zm		Pm		Pn		Zn		0	0	ZAda	
					u0			u1									S			

Encoding

```
SMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum + prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.226 SMOPS (2-way)

Signed 16-bit integer sum of outer products to 32-bit integer, subtracting

This instruction works with a 32-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of signed 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2

(FEAT_SME2)

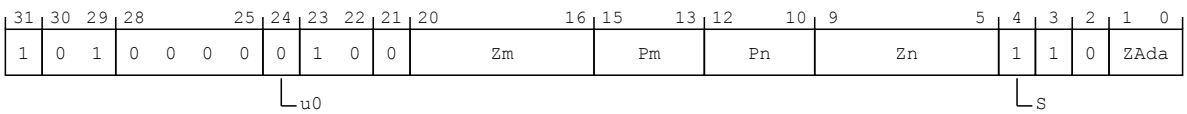

Encoding

```
SMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 1 do
            if (ActivePredicateElement{PL}(mask1, 2*row + k, esize DIV 2) &&
                  ActivePredicateElement{PL}(mask2, 2*col + k, esize DIV 2)) then
                let op1elt : bits(esize DIV 2) = operand1[(2*row + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = operand2[(2*col + k)*:(esize DIV 2)];
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum - prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.227 SMOPS (4-way)

Signed integer sum of outer products, subtracting

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	0	0	Zm		Pm		Pn		Zn		1	0	0	ZAda	
							u0			u1									S				

Encoding

```
SMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	0	Zm		Pm		Pn		Zn		1	0	ZAda	
					u0			u1									S			

Encoding

```
SMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum - prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.228 SQCVT (two registers)

Multi-vector signed 32-bit integer saturating extract narrow to 16-bit integer

This instruction saturates the signed 32-bit integer value in each element of the two source vectors to half the original source element width, and places the 16-bit results in the half-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	6	5	4	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1	1	1	1	0	0	0	Zn		0	Zd	
									op															U		

Encoding

```
SQCVT  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for r = 0 to 1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : integer = SInt(operand[e*:(2 * esize)]);
        result[(r*elements + e)*:esize] = SignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.229 SQCVT (four registers)

Multi-vector signed saturating extract narrow

This instruction saturates the signed integer value in each element of the four source vectors to quarter the original source element width, and places the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

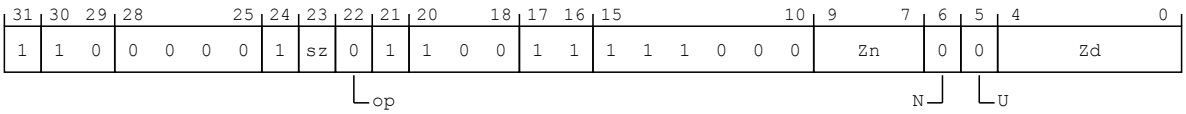

Encoding

```
SQCVT  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(sz);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Tb>
0	S
1	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : integer = SInt(operand[e*:(4 * esize)]);
        result[(r*elements + e)*:esize] = SignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.230 SQCVTN

Multi-vector signed saturating extract narrow to interleaved integer

This instruction saturates the signed integer value in each element of the four source vectors to quarter the original source element width, and places the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

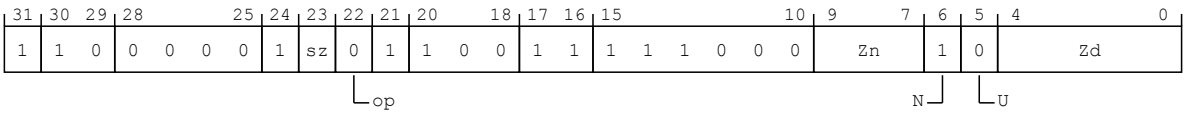

Encoding

```
SQCVTN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(sz);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Tb>
0	S
1	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 3 do
        let operand : bits(VL) = Z{}(n+i);
        let element : integer = SInt(operand[e*:(4 * esize)]);
        result[(4*e + i)*:esize] = SignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.231 SQCVTU (two registers)

Multi-vector signed 32-bit integer saturating extract narrow to unsigned 16-bit integer

This instruction saturates the signed 32-bit integer value in each element of the two source vectors to unsigned integer value that is half the original source element width, and places the 16-bit results in the half-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

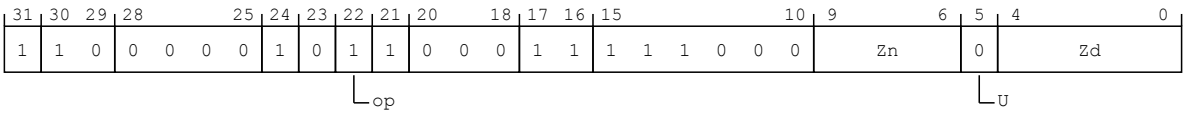

Encoding

```
SQCVTU  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for r = 0 to 1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : integer = SInt(operand[e*:(2 * esize)]);
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.232 SQCVTU (four registers)

Multi-vector signed saturating extract narrow to unsigned integer

This instruction saturates the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the original source element width, and places the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

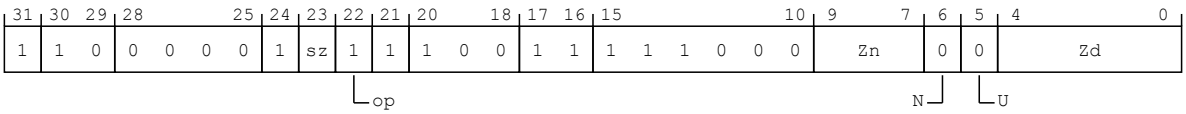

Encoding

```
SQCVTU  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(sz);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Tb>
0	S
1	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : integer = SInt(operand[e*:(4 * esize)]);
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.233 SQCVTUN

Multi-vector signed saturating extract narrow to interleaved unsigned integer

This instruction saturates the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the original source element width, and places the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

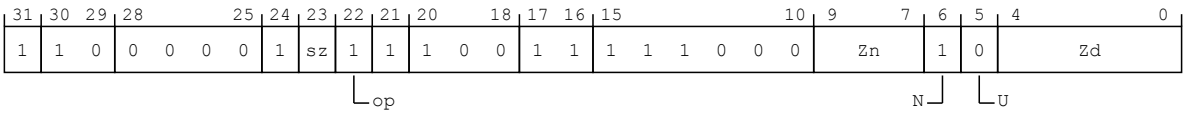

Encoding

```
SQCVTUN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(sz);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Tb>
0	S
1	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 3 do
        let operand : bits(VL) = Z{}(n+i);
        let element : integer = SInt(operand[e*:(4 * esize)]);
        result[(4*e + i)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.234 SQDMULH (multiple and single vector)

Multi-vector signed saturating doubling multiply high by vector

This instruction multiplies the signed elements of the two or four first source vectors by the corresponding elements of the second source vector, and then doubles the result and destructively places the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	1	0
1	1	0	0 0 0	0	1	size		1	0	Zm		1 0 1 0 0		1	0	0	0 0 0		Zdn		0

op

Encoding

```
SQDMULH  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	11	10	9	8	7	5	4	2	1	0
1	1	0	0 0 0	0	1	size		1	0	Zm		1 0 1 0 1		1	0	0	0 0 0		Zdn		0	0

op

Encoding

```
SQDMULH  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element1 : integer = SInt(operand1[e*:esize]);
        let element2 : integer = SInt(operand2[e*:esize]);
        let res : integer = 2 * element1 * element2;
        results[[r]][e*:esize] = SignedSat{esize}(res >> esize);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.235 SQDMULH (multiple vectors)

Multi-vector signed saturating doubling multiply high

This instruction multiplies the corresponding signed elements of the two or four first and second source vectors, and then doubles the result and destructively places the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9				5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	1	0	0	0	0	0	Zdn				0

op

Encoding

```
SQDMULH  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9				5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	1	0	0	0	0	0	Zdn		0	0

op

Encoding

```
SQDMULH  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : integer = SInt(operand1[e*:esize]);
        let element2 : integer = SInt(operand2[e*:esize]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let res : integer = 2 * element1 * element2;
        results[[r]][e*:esize] = SignedSat{esize}(res >> esize);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.236 SQRSHR (two registers)

Multi-vector signed 32-bit integer saturating rounding shift right narrow by immediate to 16-bit integer

This instruction shifts right by an immediate value the signed integer value in each element of the two source vectors, and places the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15		13	12		10	9			6	5	4			0
1	1	0	0	0	0	0	1	1	1	1	0	imm4				1	1	0	1	0	1	Zn				0	Zd			
											op															U				

Encoding

```
SQRSHR  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
let shift : integer = esize - UInt(imm4);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as "Zn" times 2 plus 1.

<const>

Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for r = 0 to 1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(2 * esize) = operand[e*:(2 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(r*elements + e)*:esize] = SignedSat{esize}(res);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.237 SQRSHR (four registers)

Multi-vector signed saturating rounding shift right narrow by immediate

This instruction shifts right by an immediate value the signed integer value in each element of the four source vectors, and places the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 7	6	5	4 0
1	1 0	0 0 0 0	1	tsize	1	imm5	1 1 0	1 1	0	Zn	0	0	Zd
									N		op	U	

Encoding

```
SQRSHR  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if tsize == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
let shift : integer = (8 * esize) - UInt(tsize::imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tsize”:

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<const>

Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in “tsize:imm5”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(4 * esize) = operand[e*:(4 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(r*elements + e)*:esize] = SignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.238 SQRSHRN

Multi-vector signed saturating rounding shift right narrow by immediate to interleaved integer

This instruction shifts right by an immediate value the signed integer value in each element of the four source vectors, and places the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	11	10	9	7	6	5	4	0
1	1	0	0	0	0	1	tsize	1	imm5		1	1	0	1	1	Zn		0	0	Zd	
															N			op	U		

Encoding

```
SQRSHRN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if tsize == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
let shift : integer = (8 * esize) - UInt(tsize::imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tsize”:

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<const>

Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in “tsize:imm5”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 3 do
        let operand : bits(VL) = Z{}(n+i);
        let element : bits(4 * esize) = operand[e*:(4 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(4*e + i)*:esize] = SignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.239 SQRSHRU (two registers)

Multi-vector signed 32-bit integer saturating rounding shift right narrow by immediate to unsigned 16-bit integer

This instruction shifts right by an immediate value the signed integer value in each element of the two source vectors, and places the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15		13	12		10	9			6	5	4			0
1	1	0	0	0	0	0	1	1	1	1	1	imm4				1	1	0	1	0	1	Zn				0	Zd			
											op															U				

Encoding

```
SQRSHRU  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
let shift : integer = esize - UInt(imm4);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as "Zn" times 2 plus 1.

<const>

Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for r = 0 to 1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(2 * esize) = operand[e*:(2 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(res);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.240 SQRSHRU (four registers)

Multi-vector signed saturating rounding shift right narrow by immediate to unsigned integer

This instruction shifts right by an immediate value the signed integer value in each element of the four source vectors, and places the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30 29	28 25	24	23 22	21	20 16	15 13	12 11	10	9 7	6	5	4 0
1	1 0	0 0 0 0	1	tsize	1	imm5	1 1 0	1 1	0	Zn	1	0	Zd
									N		op	U	

Encoding

```
SQRSHRU  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if tsize == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
let shift : integer = (8 * esize) - UInt(tsize::imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tsize”:

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<const>

Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in “tsize:imm5”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(4 * esize) = operand[e*:(4 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.241 SQRSHRUN

Multi-vector signed saturating rounding shift right narrow by immediate to interleaved unsigned integer

This instruction shifts right by an immediate value the signed integer value in each element of the four source vectors, and places the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12	11	10	9		7	6	5	4		0
1	1	0	0	0	0	0	1	tsize		1	imm5				1	1	0	1	1	1	Zn			1	0	Zd		
																				N				op	U			

Encoding

```
SQRSHRUN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if tsize == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
let shift : integer = (8 * esize) - UInt(tsize::imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tsize”:

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<const>

Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in “tsize:imm5”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 3 do
        let operand : bits(VL) = Z{}(n+i);
        let element : bits(4 * esize) = operand[e*:(4 * esize)];
        let res : integer = (SInt(element) + (1 << (shift-1))) >> shift;
        result[(4*e + i)*:esize] = UnsignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.242 SRSHL (multiple and single vector)

Multi-vector signed rounding shift left by vector

This instruction shifts the signed elements of the two or four first source vectors by the corresponding elements of the second source vector, and destructively places the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7		5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	1	0	0	0	1	Zdn				0
																									opc						U

Encoding

```
SRSHL  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	0	0	0	1	Zdn			0	0
																									opc						U

Encoding

```
SRSHL  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element : integer =  SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.243 SRSHL (multiple vectors)

Multi-vector signed rounding shift left

This instruction shifts the signed elements of the two or four first source vectors by the corresponding elements of the two or four second source vectors and destructively places the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9	8	7		5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	1	0	0	0	1	Zdn				0
																								opc							U

Encoding

```
SRSHL  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	0	1	0	0	0	1	Zdn		0	0
																									opc						U

Encoding

```
SRSHL  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element : integer =  SInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.244 ST1B (scalar plus immediate, strided registers)

Contiguous store of bytes from multiple strided vectors (immediate index)

This instruction performs a contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			0	0	0	PNg			Rn			T	0	Zt		
																msz									N			

Encoding

```
ST1B  { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			1	0	0	PNg			Rn			T	0	0	Zt	
																msz									N			

Encoding

```
ST1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.245 ST1B (scalar plus scalar, strided registers)

Contiguous store of bytes from multiple strided vectors (scalar index)

This instruction performs a contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

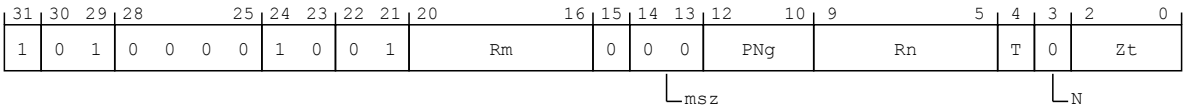

Encoding

```
ST1B  { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2)

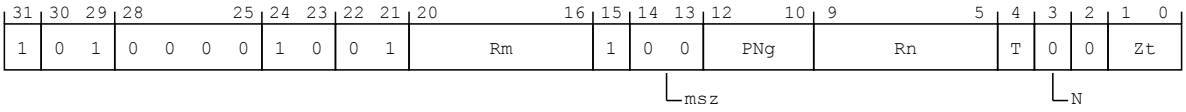

Encoding

```
ST1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.246 ST1B (scalar plus scalar, tile slice)

Contiguous store of bytes from 8-bit element ZA tile slice

This instruction performs a contiguous store of bytes from an 8-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)

31	30 29	28 25	24	23 22	21	20 16	15	14 13	12 10	9 5	4	3 0
1	1 1	0 0 0 0	0	0 0	1	Rm	V	Rs	Pg	Rn	0	off4

msz

Encoding

```
ST1B  { ZA0<HV>.B[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = 0;
let offset : integer = UInt(off4);
let esize : integer{} = 8;
let vertical : boolean = V == '1';
```

Assembler Symbols

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 15, encoded in the “off4” field.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var src : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

src = ZAslice{VL}(t, esize, vertical, slice);
for e = 0 to dim-1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    moffs = moffs + 1;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.247 ST1D (scalar plus immediate, strided registers)

Contiguous store of doublewords from multiple strided vectors (immediate index)

This instruction performs a contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			0	1	1	PNg			Rn			T	0	Zt		
																msz									N			

Encoding

```
ST1D  { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			1	1	1	PNg			Rn			T	0	0	Zt	
																msz									N			

Encoding

```
ST1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.248 ST1D (scalar plus scalar, strided registers)

Contiguous store of doublewords from multiple strided vectors (scalar index)

This instruction performs a contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3	2		0
1	0	1	0	0	0	0	1	0	0	1	Rm				0	1	1	PNg			Rn				T	0	Zt		
																msz										N			

Encoding

```
ST1D  { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12		10	9			5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm				1	1	1	PNg			Rn				T	0	0	Zt	
																msz										N			

Encoding

```
ST1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.249 ST1D (scalar plus scalar, tile slice)

Contiguous store of doublewords from 64-bit element ZA tile slice

This instruction performs a contiguous store of doublewords from a 64-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is multiplied by 8 and added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)

31	30 29	28 25	24	23 22	21	20 16	15	14 13	12 10	9 5	4	3 1	0
1	1 1	0 0 0 0	0	1 1	1	Rm	V	Rs	Pg	Rn	0	ZAt	o1
				msz									

Encoding

```
ST1D  { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #3}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = UInt(o1);
let esize : integer{} = 64;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 1, encoded in the “o1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var src : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

src = ZAslice{VL}(t, esize, vertical, slice);
for e = 0 to dim-1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    moffs = moffs + 1;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.250 ST1H (scalar plus immediate, strided registers)

Contiguous store of halfwords from multiple strided vectors (immediate index)

This instruction performs a contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			0	0	1	PNg			Rn			T	0	Zt		
																msz									N			

Encoding

```
ST1H  { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			1	0	1	PNg			Rn			T	0	0	Zt	
																msz									N			

Encoding

```
ST1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.251 ST1H (scalar plus scalar, strided registers)

Contiguous store of halfwords from multiple strided vectors (scalar index)

This instruction performs a contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm	0	0	1	PNg		Rn	T	0	Zt	
													msz						N		

Encoding

```
ST1H  { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm	1	0	1	PNg		Rn	T	0	0	Zt	
													msz						N			

Encoding

```
ST1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.252 ST1H (scalar plus scalar, tile slice)

Contiguous store of halfwords from 16-bit element ZA tile slice

This instruction performs a contiguous store of halfwords from a 16-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is multiplied by 2 and added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	1	1	0	0	0	0	0	0	1	1	Rm		V	Rs		Pg		Rn		0	ZAt	off3	

msz

Encoding

```
ST1H  { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #1}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = UInt(off3);
let esize : integer{} = 16;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 7, encoded in the “off3” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var src : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

src = ZAslice{VL}(t, esize, vertical, slice);
for e = 0 to dim-1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    moffs = moffs + 1;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.253 ST1Q

Contiguous store of quadwords from 128-bit element ZA tile slice

This instruction performs a contiguous store of quadwords from a 128-bit element ZA tile slice. The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset that is multiplied by 16 and added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)

31	30 29	28 25	24 21	20 16	15	14 13	12 10	9 5	4	3 0
1	1 1	0 0 0 0	1 1 1 1	Rm	V	Rs	Pg	Rn	0	ZAt

Encoding

```
ST1Q  { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #4}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = 0;
let esize : integer{} = 128;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, with implicit value 0.

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var src : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

src = ZAslice{VL}(t, esize, vertical, slice);
for e = 0 to dim-1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    moffs = moffs + 1;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.254 ST1W (scalar plus immediate, strided registers)

Contiguous store of words from multiple strided vectors (immediate index)

This instruction performs a contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			0	1	0	PNg			Rn			T	0	Zt		
																msz									N			

Encoding

```
ST1W  { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			1	1	0	PNg			Rn			T	0	0	Zt	
																msz									N			

Encoding

```
ST1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.255 ST1W (scalar plus scalar, strided registers)

Contiguous store of words from multiple strided vectors (scalar index)

This instruction performs a contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

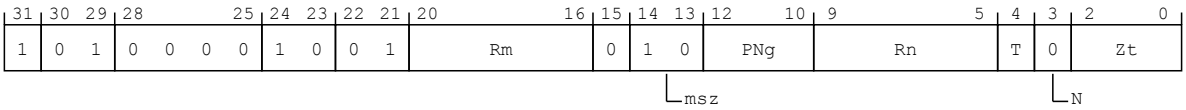

Encoding

```
ST1W  { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2)

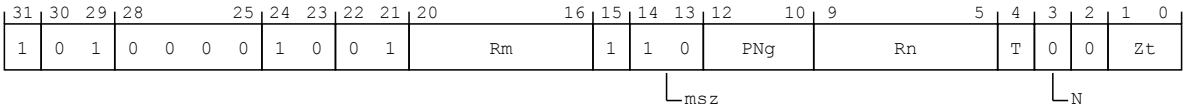

Encoding

```
ST1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.256 ST1W (scalar plus scalar, tile slice)

Contiguous store of words from 32-bit element ZA tile slice

This instruction performs a contiguous store of words from a 32-bit element ZA tile slice. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset that is multiplied by 4 and added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)

31	30 29	28 25	24	23 22	21	20 16	15	14 13	12 10	9 5	4	3 2	1 0
1	1 1	0 0 0 0	0	1 0	1	Rm	V	Rs	Pg	Rn	0	ZAt	off2
				msz									

Encoding

```
ST1W  { <ZAt><HV>.S[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #2}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('0'::Pg);
let s : integer = UInt('011'::Rs);
let t : integer = UInt(ZAt);
let offset : integer = UInt(off2);
let esize : integer{} = 32;
let vertical : boolean = V == '1';
```

Assembler Symbols

<ZAt>

Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the “ZAt” field.

<HV>

Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws>

Is the 32-bit name of the slice index register W12-W15, encoded in the “Rs” field.

<offs>

Is the slice index offset, in the range 0 to 3, encoded in the “off2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the “Rm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer = VL DIV esize;
var base : bits(64);
var addr : bits(64);
let mask : bits(PL) = P{}(g);
var moffs : bits(64) = X{}(m);
let index : bits(32) = X{}(s);
let slice : integer = (UInt(index) + offset) MOD dim;
var src : bits(VL);
let mbytes : integer{} = esize DIV 8;
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = TRUE;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    if (AnyActiveElement{PL}(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)) then
        CheckSPAlignment();
    end;
    base = SP{64}();
else
    base = X{64}(n);
end;

src = ZAslice{VL}(t, esize, vertical, slice);
for e = 0 to dim-1 do
    addr = AddressAdd(base, UInt(moffs) * mbytes, accdesc);
    if ActivePredicateElement{PL}(mask, e, esize) then
        Mem{esize}(addr, accdesc) = src[e*:esize];
    end;
    moffs = moffs + 1;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.257 STMOPA (2-way)

Signed 16-bit integer sparse sum of outer products to 32-bit integer, accumulating

This instruction generates signed integer sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected 2-in-4 16-bit signed values from each overlapping 32-bit containers of the two $SVL_S \times 2$ sub-matrices in the first source vectors by the two 16-bit signed values from the corresponding 32-bit container of the $2 \times SVL_S$ sub-matrix in the second source vector. The two selected elements from each overlapping 32-bit containers of the first source vectors correspond to 2-in-4 elements of rows of two $SVL_S \times 2$ sub-matrices. Each 32-bit container of the second source vector holds 2 elements of columns of a compressed $2 \times SVL_S$ sub-matrix.

The 2-in-4 16-bit signed values from overlapping 32-bit containers of the first source vectors are selected by 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to 32-bit containers of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30 29	28 25	24	23 22	21	20 16	15	14	13	12	11 10	9 6	5 4	3	2	1 0
1	0 0	0 0 0 0	0	0 1	0	Zm	1	0	0	K	Zk	Zn	i2	1	0	ZAda
			u0													

Encoding

```
STMOPA  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 8;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var erow : array [[2]] of bits(16);
        var ecol : array [[2]] of bits(16);
        for j = 0 to 1 do
            erow[[j]] = Zeros{16};
            ecol[[j]] = op2[(2*col + j)*:16];
        end;
        var i : integer = 0;
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            for e = 0 to 1 do
                if i < 2 && ctrl[(4*col + 2*r + e)*:1] == '1' then
                    erow[[i]] = op1[(2*row + e)*:16];
                    i = i + 1;
                end;
            end;
        end;
        var sum : bits(32) = op4[(row*dim+col)*:32];
        for j = 0 to 1 do
            let erowval : integer = if unsigned then UInt(erow[[j]]) else SInt(erow[[j]]);
            let ecolval : integer = if unsigned then UInt(ecol[[j]]) else SInt(ecol[[j]]);
            sum = sum + (erowval * ecolval);
        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.258 STMOPA (4-way)

Signed 8-bit integer sparse sum of outer products to 32-bit integer, accumulating

This instruction generates signed integer sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected two sets of 2-in-4 8-bit signed values from each overlapping 32-bit containers of the two $SVL_S \times 4$ sub-matrices in the first source vectors by the four 8-bit signed values from the corresponding 32-bit container of the $4 \times SVL_S$ sub-matrix in the second source vector. The four selected elements from the overlapping 32-bit containers of the first source vectors correspond to two sets of 2-in-4 elements of rows of two $SVL_S \times 4$ sub-matrices. Each 32-bit container of the second source vector holds 4 elements of columns of a compressed $4 \times SVL_S$ sub-matrix.

The two sets of 2-in-4 8-bit signed values from overlapping 32-bit containers of the first source vectors are selected by pairs of 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to each 32-bit container of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5		4	3	2	1	0
1	0	0	0	0	0	0	0	0	1	0	Zm				1	0	0	K	Zk		Zn				i2			0	0	ZAda	
							u0			u1																					

Encoding

```
STMOPA  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 4;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var erow : array [[4]] of bits(8);
        var ecol : array [[4]] of bits(8);
        for j = 0 to 3 do
            erow[[j]] = Zeros{8};
            ecol[[j]] = op2[(4*col + j)*:8];
        end;
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            var i : integer = 0;
            for e = 0 to 3 do
                if i < 2 && ctrl[(8*col + 4*r + e)*:1] == '1' then
                    erow[[2*r + i]] = op1[(4*row + e)*:8];
                    i = i + 1;
                end;
            end;
        end;
        var sum : bits(32) = op4[(row*dim+col)*:32];
        for j = 0 to 3 do
            let erowval : integer = if op1_unsigned then UInt(erow[[j]]) else SInt(erow[[j]]);
            let ecolval : integer = if op2_unsigned then UInt(ecol[[j]]) else SInt(ecol[[j]]);
            sum = sum + (erowval * ecolval);
        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.259 STNT1B (scalar plus immediate, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	1 1	0	imm4		0	0	0	PNg		Rn		T	1	Zt	
												msz							N		

Encoding

```
STNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	1 1	0	imm4		1	0	0	PNg		Rn		T	1	0	Zt	
												msz							N			

Encoding

```
STNT1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.260 STNT1B (scalar plus scalar, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm	0	0	0	PNg		Rn	T	1	Zt	
													msz						N		

Encoding

```
STNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 8;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm	1	0	0	PNg		Rn	T	1	0	Zt	
													msz						N			

Encoding

```
STNT1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 8;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.261 STNT1D (scalar plus immediate, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			0	1	1	PNg			Rn			T	1	Zt		
																msz									N			

Encoding

```
STNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			1	1	1	PNg			Rn			T	1	0	Zt	
																msz									N			

Encoding

```
STNT1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.262 STNT1D (scalar plus scalar, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm	0	1	1	PNg		Rn	T	1	Zt	
													msz						N		

Encoding

```
STNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 64;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm	1	1	1	PNg		Rn	T	1	0	Zt	
													msz						N			

Encoding

```
STNT1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 64;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.263 STNT1H (scalar plus immediate, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2		0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			0	0	1	PNg			Rn			T	1	Zt		
																msz									N			

Encoding

```
STNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22		20	19		16	15	14	13	12		10	9		5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4			1	0	1	PNg			Rn			T	1	0	Zt	
																msz									N			

Encoding

```
STNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.264 STNT1H (scalar plus scalar, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm	0	0	1	PNg		Rn	T	1	Zt	
													msz						N		

Encoding

```
STNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 16;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm	1	0	1	PNg		Rn	T	1	0	Zt	
													msz						N			

Encoding

```
STNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 16;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.265 STNT1W (scalar plus immediate, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (immediate index)

This instruction performs a contiguous non-temporal store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index that is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	0	1 1	0	imm4		0	1	0	PNg		Rn		T	1	Zt	
												msz							N		

Encoding

```
STNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0 0 0	0	1	0	1 1	0	imm4		1	1	0	PNg		Rn		T	1	0	Zt	
												msz							N			

Encoding

```
STNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
let offset : integer = SInt(imm4);
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<imm>

For the “Two registers” variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the “imm4” field.

For the “Four registers” variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the “imm4” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
addr = AddressAdd(base, offset * nreg * elements * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.266 STNT1W (scalar plus scalar, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (scalar index)

This instruction performs a contiguous non-temporal store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index that is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	0
1	0	1	0	0	0	0	1	0	0	1	Rm	0	1	0	PNg		Rn	T	1	Zt	
													msz						N		

Encoding

```
STNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 2;
let tstride : integer = 8;
let t : integer = UInt(T::'0'::Zt);
let esize : integer{} = 32;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	14	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm	1	1	0	PNg		Rn	T	1	0	Zt	
													msz						N			

Encoding

```
STNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let g : integer = UInt('1'::PNg);
let nreg : integer{} = 4;
let tstride : integer = 4;
let t : integer = UInt(T::'00'::Zt);
let esize : integer{} = 32;
```

Assembler Symbols

<Zt1>

For the “Two registers” variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as “T:‘0’:Zt”.

For the “Four registers” variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as “T:‘00’:Zt”.

<Zt2>

For the “Two registers” variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as “T:‘1’:Zt”.

For the “Four registers” variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as “T:‘01’:Zt”.

<PNg>

Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the “PNg” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xm>

Is the 64-bit name of the general-purpose offset register, encoded in the “Rm” field.

<Zt3>

Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as “T:‘10’:Zt”.

<Zt4>

Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as “T:‘11’:Zt”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let elements : integer = VL DIV esize;
let mbytes : integer{} = esize DIV 8;
var offset : bits(64);
var base : bits(64);
var addr : bits(64);
var src : bits(VL);
let pred : bits(PL) = P{}(g);
let mask : bits(PL * nreg) = CounterToPredicate{}(pred[15:0]);
let contiguous : boolean = TRUE;
let nontemporal : boolean = TRUE;
var transfer : integer = t;
let tagchecked : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let accdesc : AccessDescriptor = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if !AnyActiveElement{PL*nreg}(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    end;
else
    if n == 31 then CheckSPAlignment(); end;
end;

base = if n == 31 then SP{64}() else X{64}(n);
offset = X{64}(m);
addr = AddressAdd(base, UInt(offset) * mbytes, accdesc);

for r = 0 to nreg-1 do
    src = Z{VL}(transfer);
    for e = 0 to elements-1 do
        if ActivePredicateElement{PL*nreg}(mask, r * elements + e, esize) then
            Mem{esize}(addr, accdesc) = src[e*:esize];
        end;
        addr = AddressIncrement(addr, mbytes, accdesc);
    end;
    transfer = transfer + tstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.267 STR (array vector)

Store ZA array vector

This instruction performs a contiguous store of bytes from a ZA array vector to the memory address generated by a 64-bit scalar base plus an optional immediate offset multiplied by the current vector length in bytes. The ZA array vector is selected by the sum of the vector select register and the same immediate offset, modulo the number of bytes in a Streaming SVE vector.

This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME

(FEAT_SME)

31	30	29	28			25	24			22	21	20				16	15	14	13	12			10	9			5	4	3			0
1	1	1	0	0	0	0	1	0	0		1	0	0	0	0	0	0	Rv		0	0	0		Rn				0	off4			
											op																					

Encoding

```
STR  ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
let v : integer = UInt('011'::Rv);
let offset : integer = UInt(off4);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W12-W15, encoded in the “Rv” field.

<offs>

Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the “off4” field.

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckSMEAndZAEnabled();
let SVL : integer{} = CurrentSVL();
let dim : integer{} = SVL DIV 8;
var base : bits(64);
let moffs : integer = offset * dim;
var src : bits(SVL);
let vbase : bits(32) = X{}(v);
let vec : integer = (UInt(vbase) + offset) MOD dim;
let contiguous : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP{64}();
else
    base = X{64}(n);
end;

src = ZAvector{SVL}(vec);

var addr : bits(64) = AddressAdd(base, moffs, accdesc);

let aligned : boolean = IsAlignedSize(base, 16);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;

for e = 0 to dim-1 do
    AArch64_MemSingle{8}(addr, accdesc, aligned) = src[e*:8];
    addr = AddressIncrement(addr, 1, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.268 STR (table)

Store ZT0 register

This instruction stores the 64-byte ZT0 register to the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2

(FEAT_SME2)

31	30	29	28			25	24		22	21					16	15	14				10	9		5	4		2	1	0
1	1	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	Rn			0	0	0	0	0
										opc																		opc2	

Encoding

```
STR  ZT0, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Rn);
```

Assembler Symbols

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
let elements : integer = 512 DIV 8;
var addr : bits(64);
let table : bits(512) = ZT0{}();
let contiguous : boolean = TRUE;
let nontemporal : boolean = FALSE;
let tagchecked : boolean = n != 31;
let accdesc : AccessDescriptor = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
                                                  tagchecked);

if n == 31 then
    CheckSPAlignment();
    addr = SP{64}();
else
    addr = X{64}(n);
end;

let aligned : boolean = IsAlignedSize(addr, 16);

if !aligned && AlignmentEnforced() then
    let fault : FaultRecord = AlignmentFault(accdesc, addr);
    AArch64_Abort(fault);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1 do
    AArch64_MemSingle{8}(addr, accdesc, aligned) = table[e*:8];
    addr = AddressIncrement(addr, 1, accdesc);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.269 SUB (to array, array and multiple vectors)

Multi-vector subtract from ZA array vectors

This instruction destructively subtracts all elements of the two or four source vectors from the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv	1	1	1	Zm	0	1	1	off3
																								S	

Encoding

```
SUB  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm	0	0	1	1	off3
																									S	

Encoding

```
SUB  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zm4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = ZAvector{}(vec);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = element1 - element2;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.270 SUB (to array, multiple and single vector)

Multi-vector subtract by vector to ZA array vectors

This instruction subtracts all the corresponding elements of the second source vector from the two or four first source vectors and places the results in the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm	0	Rv	1	1	0	Zn	1	1	off3	
																				S		

Encoding

```
SUB  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm	0	Rv	1	1	0	Zn	1	1	off3	
																				S		

Encoding

```
SUB  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = element1 - element2;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.271 SUB (to array, multiple vectors)

Multi-vector subtract to ZA array vectors

This instruction subtracts all the corresponding elements of the two or four second source vectors from first source vectors and places the results in the corresponding elements of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	1	1	sz	1	Zm		0	0	Rv		1	1	Zn		0	1	1	off3	

S

Encoding

```
SUB  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	1	1	sz	1	Zm		0	1	0	Rv		1	1	0	Zn		0	0	1	1	off3

S

Encoding

```
SUB  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element1 : bits(esize) = operand1[e*:esize];
        let element2 : bits(esize) = operand2[e*:esize];
        result[e*:esize] = element1 - element2;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.272 SUDOT (4-way, multiple and indexed vector)

Multi-vector signed by unsigned 8-bit integer dot product by indexed elements to 32-bit integer

This instruction calculates the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2		Zn				1	1	1	off3			
																			op								U					

Encoding

```
SUDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn				0	1	1	1	off3			
																			op									U					

Encoding

```
SUDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        for i = 0 to 3 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.273 SUDOT (4-way, multiple and single vector)

Multi-vector signed by unsigned 8-bit integer dot product by vector to 32-bit integer

This instruction calculates the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm	0	Rv	1	0	1	Zn	1	1	off3	

U

Encoding

```
SUDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm	0	Rv	1	0	1	Zn	1	1	off3	

U

Encoding

```
SUDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.274 SUMLALL (multiple and indexed vector)

Multi-vector signed by unsigned 8-bit integer multiply-add by indexed element to 32-bit integer

This instruction multiplies each signed 8-bit element in the one, two, or four first source vectors by each unsigned 8-bit indexed element of the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l			Zn				1	0	1	off2	

U, S, op

Encoding

```
SUMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn				1	1	0	i4l		o1

op, U, S

Encoding

```
SUMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0 0 0	0	1	0	0	0	1	Zm		1	Rv		0	i4h		Zn		0	1	1	0	i4l		o1

op, U, S

Encoding

```
SUMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[s*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.275 SUMLALL (multiple and single vector)

Multi-vector signed by unsigned 8-bit integer multiply-add by vector to 32-bit integer

This instruction multiplies each signed 8-bit element in the two or four first source vectors by each unsigned 8-bit element in the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME2)

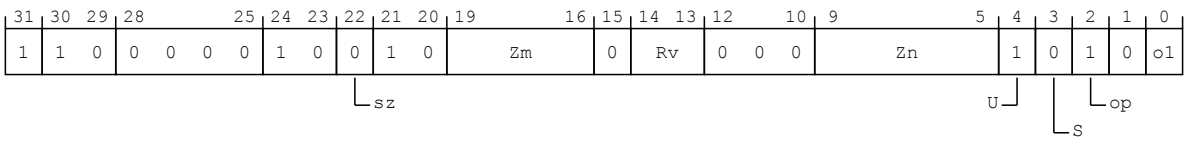

Encoding

```
SUMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2)

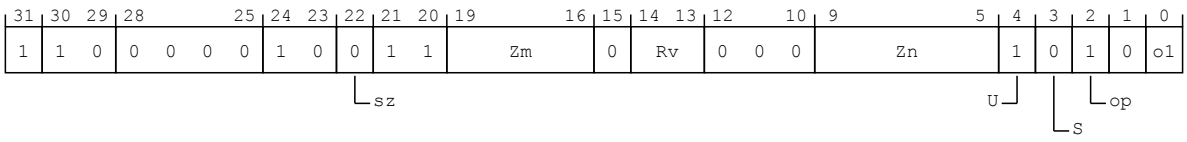

Encoding

```
SUMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.276 SUMOP4A

Signed by unsigned integer quarter-tile sum of outer products, accumulating

This instruction generates four independent quarter-tile signed by unsigned integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit unsigned values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively added to the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0		1	1		Zm			0	1	0	0	0	0	0		0		Zn			0	0		0		ZAda

u0 u1 M N S

Encoding

```
SUMOP4A  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	1	0		Zm			0	1	0	0	0	0	0		0		Zn			0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
SUMOP4A  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28				25	24	23	22	21	20	19			17	16	15	14					10	9	8			6	5	4	3	2	1	0
1	0	0	0	0	0	0		0	0	0	1	0		Zm			0	1	0	0	0	0	0		1		Zn			0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
SUMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

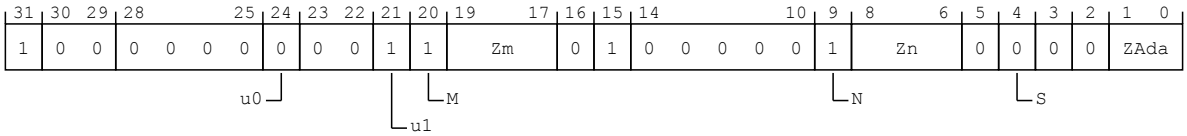

Encoding

```
SUMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

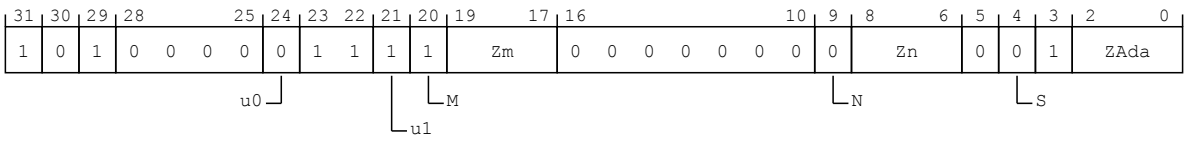

Encoding

```
SUMOP4A  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	1	0	Zm		0 0 0 0 0 0	0	0	Zn		0	0	1	ZAda	

u0 u1 M N S

Encoding

```
SUMOP4A  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	17	16	10	9	8	6	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	1	0	Zm		0 0 0 0 0 0	0	1	Zn		0	0	1	ZAda	

u0 u1 M N S

Encoding

```
SUMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

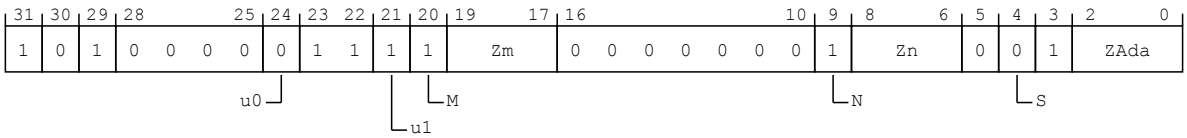

Encoding

```
SUMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.277 SUMOP4S

Signed by unsigned integer quarter-tile sum of outer products, subtracting

This instruction generates four independent quarter-tile signed by unsigned integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit unsigned values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit signed values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively subtracted from the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

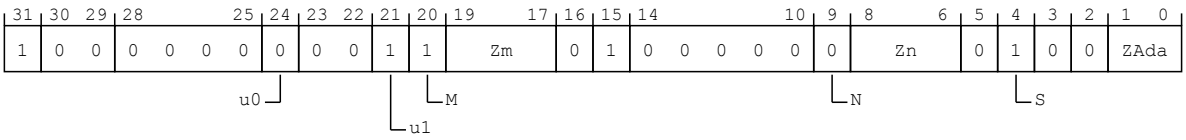

Encoding

```
SUMOP4S  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	1	0	Zm		0	1	0 0 0 0 0		0	Zn		0	1	0	0	ZAda	

u0 u1 M N S

Encoding

```
SUMOP4S  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		0	0	0	1	0	Zm		0	1	0 0 0 0 0		1	Zn		0	1	0	0	ZAda	

u0 u1 M N S

Encoding

```
SUMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

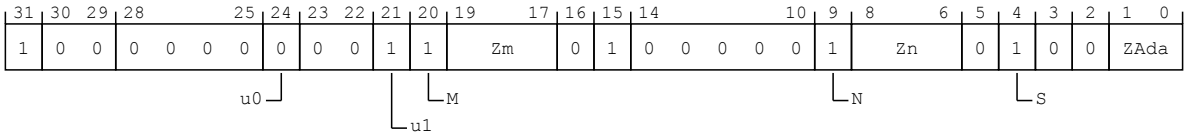

Encoding

```
SUMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

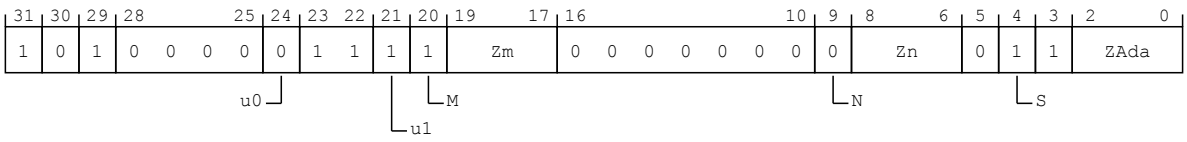

Encoding

```
SUMOP4S  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19		17	16							10	9	8		6	5	4	3	2		0
1	0	1	0	0	0	0	0	1	1	1	0	Zm			0	0	0	0	0	0	0	0	0	Zn			0	1	1	ZAda		

u0 u1 M N S

Encoding

```
SUMOP4S  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19		17	16							10	9	8		6	5	4	3	2		0
1	0	1	0	0	0	0	0	1	1	1	0	Zm			0	0	0	0	0	0	0	0	1	Zn			0	1	1	ZAda		

u0 u1 M N S

Encoding

```
SUMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

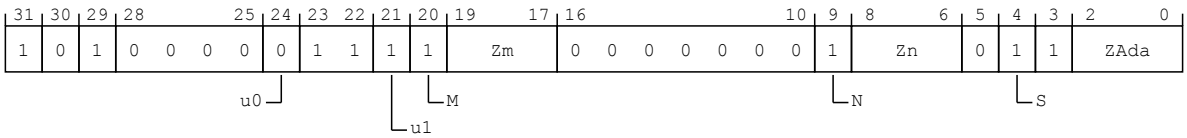

Encoding

```
SUMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.278 SUMOPA (4-way)

Signed by unsigned integer sum of outer products, accumulating

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	0	1	Zm		Pm		Pn		Zn		0	0	0	ZAda	

u0 u1 S

Encoding

```
SUMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	1	Zm		Pm		Pn		Zn		0	0	ZAda	
					u0			u1									S			

Encoding

```
SUMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum + prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.279 SUMOPS

Signed by unsigned integer sum of outer products, subtracting

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	Zm		Pm		Pn		Zn	1	0	0	ZAda	

u0 u1 S

Encoding

```
SUMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	0	1	1	1	Zm		Pm		Pn		Zn		1	0	ZAda	
					u0			u1									S			

Encoding

```
SUMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum - prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.280 SUNPK

Unpack and sign-extend multi-vector elements

This instruction unpacks and sign-extends elements from one or two source vectors, and writes the sign-extended values into the double-width elements of two or four destination vectors. This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	10	9	5	4	1	0
1	1	0	0	0	1	size		1	0	1	0	1	1	0	Zn		Zd		0

U

Encoding

```
SUNPK  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let unsigned : boolean = FALSE;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	10	9	6	5	4	2	1	0
1	1	0	0	0	1	size		1	1	1	0	1	1	0	Zn		0	Zd		0	0

U

Encoding

```
SUNPK  { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<Tb>-<Zn2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let hsize : integer{} = esize DIV 2;
let sreg : integer = nreg DIV 2;
var results : array [[4]] of bits(VL);

for r = 0 to sreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for i = 0 to 1 do
        for e = 0 to elements-1 do
            let element : bits(hsize) = operand[(i*elements + e)*:hsize];
            results[[2*r+i]][e*:esize] = Extend{esize}(element, unsigned);
        end;
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.281 SUTMOPA

Signed by unsigned 8-bit integer sparse sum of outer products to 32-bit integer, accumulating

This instruction generates signed by unsigned integer sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected two sets of 2-in-4 8-bit signed values from each overlapping 32-bit containers of the two $SVL_S \times 4$ sub-matrices in the first source vectors by the four 8-bit unsigned values from the corresponding 32-bit container of the $4 \times SVL_S$ sub-matrix in the second source vector. The four selected elements from the overlapping 32-bit containers of the first source vectors correspond to two sets of 2-in-4 elements of rows of two $SVL_S \times 4$ sub-matrices. Each 32-bit container of the second source vector holds 4 elements of columns of a compressed $4 \times SVL_S$ sub-matrix.

The two sets of 2-in-4 8-bit signed values from overlapping 32-bit containers of the first source vectors are selected by pairs of 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to each 32-bit container of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	1	1	Zm				1	0	0	K	Zk		Zn				i2		0	0	ZAda	
							u0			u1																				

Encoding

```
SUTMOPA  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = FALSE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 4;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var erow : array [[4]] of bits(8);
        var ecol : array [[4]] of bits(8);
        for j = 0 to 3 do
            erow[[j]] = Zeros{8};
            ecol[[j]] = op2[(4*col + j)*:8];
        end;
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            var i : integer = 0;
            for e = 0 to 3 do
                if i < 2 && ctrl[(8*col + 4*r + e)*:1] == '1' then
                    erow[[2*r + i]] = op1[(4*row + e)*:8];
                    i = i + 1;
                end;
            end;
        end;
        var sum : bits(32) = op4[(row*dim+col)*:32];
        for j = 0 to 3 do
            let erowval : integer = if op1_unsigned then UInt(erow[[j]]) else SInt(erow[[j]]);
            let ecolval : integer = if op2_unsigned then UInt(ecol[[j]]) else SInt(ecol[[j]]);
            sum = sum + (erowval * ecolval);
        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.282 SUVDOT

Multi-vector signed by unsigned 8-bit integer vertical dot product by indexed element to 32-bit integer

This instruction calculates the vertical dot product of the corresponding signed 8-bit elements from the four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the ZA operand consists of four ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1		Zm			1		Rv	0		i2		Zn		0	1	1	1		off3	
																			op								U				

Encoding

```
SUVDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 4;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 3 do
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let operand1 : bits(VL) = Z{}(n+i);
            let element1 : integer = SInt(operand1[(4 * e + r)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.283 SVDOT (2-way)

Multi-vector signed 16-bit integer vertical dot product by indexed element to 32-bit integer

This instruction calculates the vertical dot product of the corresponding two signed 16-bit integer values held in the two first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the ZA operand consists of two ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				1	0	0	off3		
																			op								U				

Encoding

```
SVDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 2;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 1 do
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 1 do
            let operand1 : bits(VL) = Z{}(n+i);
            let element1 : integer = SInt(operand1[(2 * e + r)*:(esize DIV 2)]);
            let element2 : integer = SInt(operand2[(2 * s + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.284 SVDOT (4-way)

Multi-vector signed integer vertical dot product by indexed element

This instruction calculates the vertical dot product of the corresponding four signed 8-bit or 16-bit integer values held in the four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the ZA operand consists of four ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 8-bit to 32-bit and 16-bit to 64-bit

8-bit to 32-bit

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0 0 0	0	1	0	1	0	1	Zm		1	Rv		0	i2		Zn		0	1	0	0	off3	
															op							U			

Encoding

```
SVDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

16-bit to 64-bit

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0 0 0	0	1	1	1	0	1	Zm		1	Rv		0	1	i1	Zn		0	0	0	1	off3	
																op						U			

Encoding

```
SVDOT  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “8-bit to 32-bit” variant: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “16-bit to 64-bit” variant: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 4;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 3 do
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let operand1 : bits(VL) = Z{}(n+i);
            let element1 : integer = SInt(operand1[(4 * e + r)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.285 UCLAMP

Multi-vector unsigned clamp to minimum/maximum

This instruction clamps each unsigned element in the two or four destination vectors to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively places the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	0	1	Zn				Zd				1

U

Encoding

```
UCLAMP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	1	1	Zn				Zd			0	1

U

Encoding

```
UCLAMP  { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = Z{}(d+r);
    for e = 0 to elements-1 do
        let element1 : integer = UInt(operand1[e*:esize]);
        let element2 : integer = UInt(operand2[e*:esize]);
        let element3 : integer = UInt(operand3[e*:esize]);
        let res : integer = Min(Max(element1, element3), element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.286 UCVTF

Multi-vector unsigned 32-bit integer convert to single-precision

This instruction converts each element of the two or four source vectors from unsigned 32-bit integer to single-precision, and places the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	10	9	6	5	4	1	0
1	1	0	0 0 0 0		1	0	0	1	0 0 0		1	0	1 1 1 0 0 0		Zn		1	Zd		0
																	U			

Encoding

```
UCVTF  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	10	9	7	6	5	4	2	1	0
1	1	0	0 0 0 0		1	0	0	1	1 0	0	1	0	1 1 1 0 0 0		Zn		0	1	Zd		0	0
																		U				

Encoding

```
UCVTF  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
let rounding : FPRounding = FPRoundingMode(FPCR());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn1>

For the “Two registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV 32;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(32) = operand[e*:32];
        results[[r]][e*:32] = FixedToFP{32, 32}(element, 0, unsigned, FPCR(), rounding);
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.287 UDOT (2-way, multiple and indexed vector)

Multi-vector unsigned 16-bit integer dot product by indexed element to 32-bit integer

This instruction calculates the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	0
1	1	0	0	0	0	1	0	1	0	1	Zm	0	Rv		1	i2		Zn		0	1	0	off3	
															op						U			

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	1	0	1	0	1	Zm	1	Rv		1	i2		Zn		0	0	1	0	off3	
															op							U			

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        for i = 0 to 1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * s + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.288 UDOT (2-way, multiple and single vector)

Multi-vector unsigned 16-bit integer dot product by vector to 32-bit integer

This instruction calculates the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	1	0	1	Zn	1	1	off3	

U

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm	0	Rv	1	0	1	Zn	1	1	off3	

U

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.289 UDOT (2-way, multiple vectors)

Multi-vector unsigned 16-bit integer dot product to 32-bit integer

This instruction calculates the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	1	Zm		0	0	Rv		1	0	1	Zn	0	1	1	off3

U

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	1	1	Zm		0	1	0	Rv		1	0	1	Zn	0	0	1	1	off3

U

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.290 UDOT (4-way, multiple and indexed vector)

Multi-vector unsigned integer dot product by indexed element

This instruction calculates the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors of 32-bit elements, Two ZA single-vectors of 64-bit elements, Four ZA single-vectors of 32-bit elements, and Four ZA single-vectors of 64-bit elements

Two ZA single-vectors of 32-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2		Zn				1	1	0	off3		
																			op								U				

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Two ZA single-vectors of 64-bit elements

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		0	0	i1	Zn				0	1	1	off3		
																											U				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
UDOT  ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 2;
```

Four ZA single-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn			0	1	1	0	off3		
																			op								U				

Encoding

```
UDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Four ZA single-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		0	0	i1	Zn			0	0	1	1	off3		
																				op							U				

Encoding

```
UDOT  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors of 32-bit elements” and “Two ZA single-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors of 32-bit elements” and “Four ZA single-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA single-vectors of 32-bit elements” and “Two ZA single-vectors of 32-bit elements” variants: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “Four ZA single-vectors of 64-bit elements” and “Two ZA single-vectors of 64-bit elements” variants: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        for i = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.291 UDOT (4-way, multiple and single vector)

Multi-vector unsigned integer dot product by vector

This instruction calculates the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm	0	Rv	1	0	1	Zn	1	0	off3	

U

Encoding

```
UDOT  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm	0	Rv	1	0	1	Zn	1	0	off3	

U

Encoding

```
UDOT  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.292 UDOT (4-way, multiple vectors)

Multi-vector unsigned integer dot product

This instruction calculates the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30 29	28 25	24 23	22	21	20 17	16 15	14 13	12 10	9 6	5	4	3	2 0
1	1 0	0 0 0 0	1 1	sz	1	Zm	0 0	Rv	1 0 1	Zn	0	1	0	off3
												U		

Encoding

```
UDOT  ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30 29	28 25	24 23	22	21	20 18	17	16 15	14 13	12 10	9 7	6 5	4	3	2 0
1	1 0	0 0 0 0	1 1	sz	1	Zm	0	1 0	Rv	1 0 1	Zn	0 0	1	0	off3
													U		

Encoding

```
UDOT  ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32 << UInt(sz);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.293 UMAX (multiple and single vector)

Multi-vector unsigned maximum by vector

This instruction determines the unsigned maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	0	0	0	0	0	Zdn				1
																										op					U

Encoding

```
UMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	0	Zdn			0	1
																										op					U

Encoding

```
UMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Max(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.294 UMAX (multiple vectors)

Multi-vector unsigned maximum

This instruction determines the unsigned maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	0	0	0	0	0	Zdn				1
																										opc						U

Encoding

```
UMAX  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9			7	6	5	4			2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	0	1	0	1	1	1	0	0	0	0	0	0	0	Zdn				0	1
																										opc							U

Encoding

```
UMAX  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Max(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.295 UMIN (multiple and single vector)

Multi-vector unsigned minimum by vector

This instruction determines the unsigned minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	0	0	0	0	1	Zdn				1
																										op					U

Encoding

```
UMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	1	Zdn			0	1
																										op					U

Encoding

```
UMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Min(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.296 UMIN (multiple vectors)

Multi-vector unsigned minimum

This instruction determines the unsigned minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively places the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9			7	6	5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	0	0	0		0	1	Zdn				1
																										opc						U

Encoding

```
UMIN  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9			7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm			0	0	1	0	1	1	1	0	0	0	0		0	1	Zdn			0	1
																										opc						U

Encoding

```
UMIN  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = operand1[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let op2elt : bits(esize) = operand2[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let res : integer = Min(element1, element2);
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.297 UMLAL (multiple and indexed vector)

Multi-vector unsigned 16-bit integer multiply-add by indexed element to 32-bit integer

This instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors by each unsigned 16-bit indexed element of the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14		13	12	11		10	9			5	4	3	2			0
1	1	0	0	0	0	0	1	1	1	0	0	Zm				i3h	Rv			1	i3l			Zn				1	0	off3			
																												U	S				

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14		13	12	11		10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv			1	i3h			Zn				0	1	0	i3l	off2	
																													U	S			

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		1	i3h		Zn			0	0	1	0	i3l	off2	
																											U	S			

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[s*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.298 UMLAL (multiple and single vector)

Multi-vector unsigned 16-bit integer multiply-add by vector to 32-bit integer

This instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors by each unsigned 16-bit element in the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	1	Zn	1	0	off3	
																			U	S		

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	0	Zn	1	0	0	off2	
																			U	S	op		

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		0	1	0	Zn				1	0	0	off2	

U, S, op

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.299 UMLAL (multiple vectors)

Multi-vector unsigned 16-bit integer multiply-add to 32-bit integer

This instruction multiplies each unsigned 16-bit element in the two or four first source vectors by each unsigned 16-bit element in the two or four second source vectors, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	0	Rv		0	1	0	Zn	0	1	0	0	off2	
																				U	S			

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	1	0	0	off2
																						U	S			

Encoding

```
UMLAL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.300 UMLALL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long long by indexed element

This instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors by each unsigned 8-bit or 16-bit indexed element of the second source vector, widens each product to 32 bits or 64 bits, and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Two ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 32-bit elements, and Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l			Zn				1	0	0	off2	

U, S, op

Encoding

```
UMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

One ZA quad-vector of 64-bit elements

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv		0	i3l		Zn				1	0	0	off2	

U, S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
UMLALL  ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn				0	1	0	i4l		o1

op, U, S

Encoding

```
UMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Two ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				0	Rv		0	0	i3h	Zn				0	1	0	i3l		o1

U, S

Encoding

```
UMLALL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	Zm		1	Rv		0	i4h		Zn		0	0	1	0	i4l		o1

op, U, S

Encoding

```
UMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	0	1	Zm		1	Rv		0	0	i3h	Zn		0	0	1	0	i3l		o1

U, S

Encoding

```
UMLALL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA quad-vectors of 32-bit elements”, “One ZA quad-vector of 32-bit elements”, and “Two ZA quad-vectors of 32-bit elements” variants: is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

For the “Four ZA quad-vectors of 64-bit elements”, “One ZA quad-vector of 64-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA quad-vectors of 32-bit elements” and “Two ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors of 32-bit elements” and “Four ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[s*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.301 UMLALL (multiple and single vector)

Multi-vector unsigned integer multiply-add long long by vector

This instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors by each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32 bits or 64 bits, and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm			0		Rv	0	0	1		Zn			1	0	0	off2	

U, S, op

Encoding

```
UMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm			0		Rv	0	0	0		Zn			1	0	0	0	o1

U, S, op

Encoding

```
UMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		0	0	0	Zn				1	0	0	0	o1

U, S, op

Encoding

```
UMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.302 UMLALL (multiple vectors)

Multi-vector unsigned integer multiply-add long long

This instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors by each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32 bits or 64 bits, and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20			17	16	15	14	13	12			10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv		0	0	0		Zn				0	1	0	0	0	o1
																												U	S	op		

Encoding

```
UMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20			18	17	16	15	14	13	12			10	9			7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	1	0	Rv		0	0	0		Zn				0	0	1	0	0	0	o1
																														U	S	op		

Encoding

```
UMLALL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.303 UMLSL (multiple and indexed vector)

Multi-vector unsigned 16-bit integer multiply-subtract by indexed element from 32-bit integer

This instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors by each unsigned 16-bit indexed element of the second source vector, widens each product to 32 bits, and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	5	4	3	2	0
1	1	0	0	0	1	1	1	0	0	Zm		i3h	Rv		1	i3l		Zn		1	1	off3	
																				U	S		

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	0	1	Zm		0	Rv		1	i3h		Zn		0	1	1	i3l	off2	
																					U	S			

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	0	1	Zm		1	Rv		1	i3h		Zn		0	0	1	1	i3l	off2	

U S

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 2 * segmentbase + index;
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[s*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.304 UMLSL (multiple and single vector)

Multi-vector unsigned 16-bit integer multiply-subtract by vector from 32-bit integer

This instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors by each unsigned 16-bit element in the second source vector, widens each product to 32 bits, and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	1	Zn	1	1	off3	
																			U	S		

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3::'0');
let nreg : integer{} = 1;
```

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	0	Zn	1	1	0	off2	
																			U	S	op		

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm				0	Rv		0	1	0	Zn				1	1	0	off2	

U S op

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.305 UMLSL (multiple vectors)

Multi-vector unsigned 16-bit integer multiply-subtract from 32-bit integer

This instruction multiplies each unsigned 16-bit element in the two or four first source vectors by each unsigned 16-bit element in the two or four second source vectors, widens each product to 32 bits, and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The double-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

Two ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	0	Rv		0	1	0	Zn	0	1	1	0	off2	
																				U	S			

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 2;
```

Four ZA double-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	1	1	0	off2
																						U	S			

Encoding

```
UMLSL  ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off2::'0');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “off2” field times 2.

<offs2>

Is the second vector select offset, encoded as “off2” field times 2 plus 1.

<Zn1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA double-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 2);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 1 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(2 * e + i)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * e + i)*:(esize DIV 2)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.306 UMLSLL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long long by indexed element

This instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors by each unsigned 8-bit or 16-bit indexed element of the second source vector, widens each product to 32 bits or 64 bits, and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Two ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 32-bit elements, and Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12			10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l				Zn				1	1	0	off2	

U, S, op

Encoding

```
UMLSLL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

One ZA quad-vector of 64-bit elements

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv		0	i3l			Zn				1	1	0	off2	

U, S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
UMLSLL  ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]
```

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn				0	1	1	i4l		o1

op, U, S

Encoding

```
UMLSLL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Two ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm				0	Rv		0	0	i3h	Zn				0	1	1	i3l		o1

U, S

Encoding

```
UMLSLL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors of 32-bit elements (FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	Zm		1	Rv		0	i4h		Zn		0	0	1	1	i4l		o1

op, U, S

Encoding

```
UMLSLL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	0	0	1	Zm		1	Rv		0	0	i3h	Zn		0	0	1	1	i3l		o1

U, S

Encoding

```
UMLSLL  ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i3h::i3l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector of 32-bit elements” and “One ZA quad-vector of 64-bit elements” variants: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors of 32-bit elements”, “Four ZA quad-vectors of 64-bit elements”, “Two ZA quad-vectors of 32-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “Four ZA quad-vectors of 32-bit elements”, “One ZA quad-vector of 32-bit elements”, and “Two ZA quad-vectors of 32-bit elements” variants: is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

For the “Four ZA quad-vectors of 64-bit elements”, “One ZA quad-vector of 64-bit elements”, and “Two ZA quad-vectors of 64-bit elements” variants: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

<Zn1>

For the “Two ZA quad-vectors of 32-bit elements” and “Two ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors of 32-bit elements” and “Four ZA quad-vectors of 64-bit elements” variants: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[s*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.307 UMLSLL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long long by vector

This instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors by each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32 bits or 64 bits, and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm			0		Rv	0	0	1		Zn			1	1	0	off2	

U S op

Encoding

```
UMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm			0		Rv	0	0	0		Zn			1	1	0	0	o1

U S op

Encoding

```
UMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm				0	Rv		0	0	0	Zn				1	1	0	0	o1

U, S, op

Encoding

```
UMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.308 UMLSLL (multiple vectors)

Multi-vector unsigned integer multiply-subtract long long

This instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors by each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32 bits or 64 bits, and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	1	0
1	1	0	0	0	1	1	sz	1	Zm		0	0	Rv		0	0	Zn		0	1	1	0	0	o1

U, S, op

Encoding

```
UMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2 && (sz == '0' || FEAT_SME_I16I64))

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	sz	1	Zm		0	1	0	Rv		0	0	Zn		0	0	1	1	0	0	o1

U, S, op

Encoding

```
UMLSLL  ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if sz == '1' && !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32 << UInt(sz);
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] - product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.309 UMOP4A (2-way)

Unsigned 16-bit integer quarter-tile sum of outer products to 32-bit integer, accumulating

This instruction generates four independent quarter-tile unsigned integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products are then destructively added to the 32-bit integer destination tile.

This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	1	0	0	0	0	0	0	0	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
UMOP4A  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	1	0	0	0	0	0	0	0	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
UMOP4A  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	1	0	0	0	0	0		1	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
UMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	1	0	0	0	0	0		1	Zn			0	0	1	0	ZAda	
							u0				M												N					S				

Encoding

```
UMOP4A  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 1 do
                let op1elt : bits(esize DIV 2) = op1[(2*row_idx + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = op2[(2*col_idx + k)*:(esize DIV 2)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.310 UMOP4A (4-way)

Unsigned integer quarter-tile sum of outer products, accumulating

This instruction generates four independent quarter-tile unsigned integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit unsigned values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively added to the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0	0	1	0	0	1	1	Zm		0	1	0 0 0 0	0	0	Zn		0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
UMOP4A  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	1	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	
							u0			u1	M												N					S				

Encoding

```
UMOP4A  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	1	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	
							u0			u1	M												N					S				

Encoding

```
UMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	1	Zm			0	1	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
UMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19		17	16						10	9	8		6	5	4	3	2		0
1	0	1	0	0	0	0	1	1	1	1	1	Zm			0	0	0	0	0	0	0	0	Zn			0	0	1	ZAda		

u0, u1, M, N, S

Encoding

```
UMOP4A  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

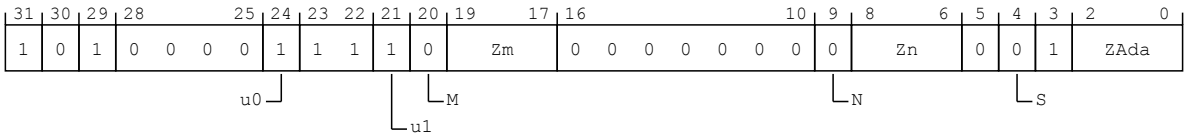

Encoding

```
UMOP4A  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

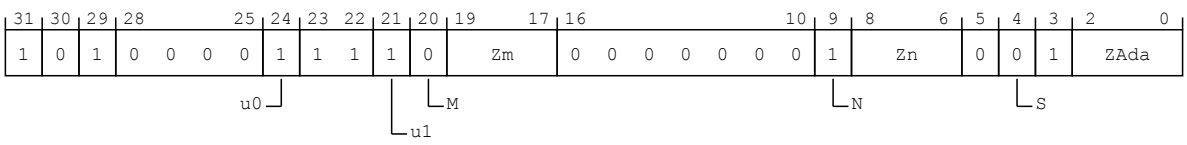

Encoding

```
UMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

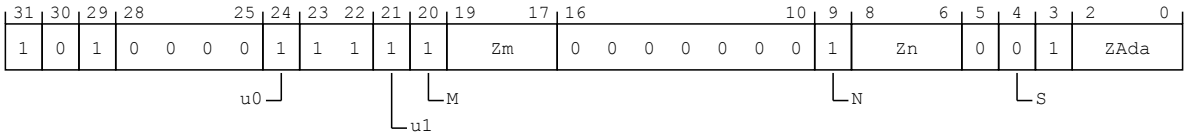

Encoding

```
UMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.311 UMOP4S (2-way)

Unsigned 16-bit integer quarter-tile sum of outer products to 32-bit integer, subtracting

This instruction generates four independent quarter-tile unsigned integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit element ZA tile.

Each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the first source vectors by the $2 \times SVL_S \div 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 2 elements of each row of a $SVL_S \div 2 \times 2$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 2 elements of each column of a $2 \times SVL_S \div 2$ sub-matrix. The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products are then destructively subtracted from the 32-bit integer destination tile.

This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 4 classes: Single and multiple vectors, Single vectors, Multiple and single vectors, and Multiple vectors

Single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	1	0	0	0	0	0	0	0	Zn			0	1	1	0	ZAda	
							u0				M												N					S				

Encoding

```
UMOP4S  <ZAda>.S, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	1	0	0	0	0	0	0	0	Zn			0	1	1	0	ZAda	
							u0				M												N					S				

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
UMOP4S  <ZAda>.S, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	1	0	0	0	0	0	1	Zn				0	1	1	0	ZAda	
							u0				M											N						S				

Encoding

```
UMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Multiple vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	1	Zm			0	1	0	0	0	0	0	1	Zn				0	1	1	0	ZAda	
							u0				M											N						S				

Encoding

```
UMOP4S  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 1 do
                let op1elt : bits(esize DIV 2) = op1[(2*row_idx + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = op2[(2*col_idx + k)*:(esize DIV 2)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.312 UMOP4S (4-way)

Unsigned integer quarter-tile sum of outer products, subtracting

This instruction generates four independent quarter-tile unsigned integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit unsigned values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit unsigned values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively subtracted from the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0	0	1	0	0	1	1	Zm		0	1	0 0 0 0	0	0	Zn		0	1	0	0	ZAda	

u0 u1 M N S

Encoding

```
UMOP4S  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	1	0	0	0	0	0	0	0	Zn			0	1	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
UMOP4S  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	1	0	Zm			0	1	0	0	0	0	0	0	1	Zn			0	1	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
UMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

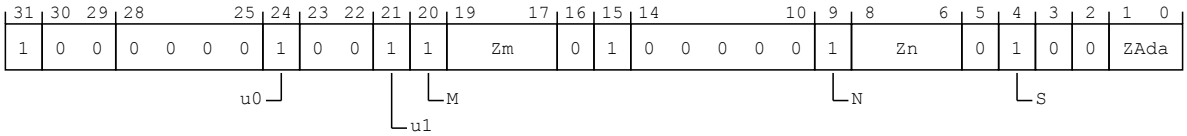

Encoding

```
UMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

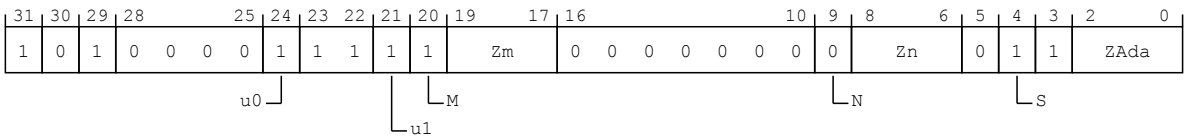

Encoding

```
UMOP4S  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

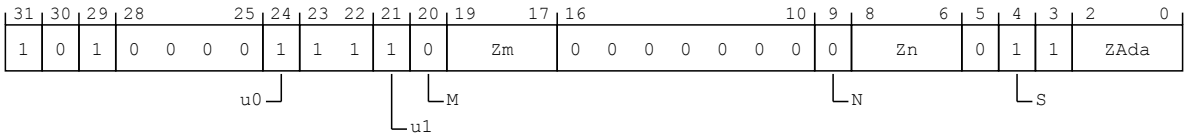

Encoding

```
UMOP4S  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

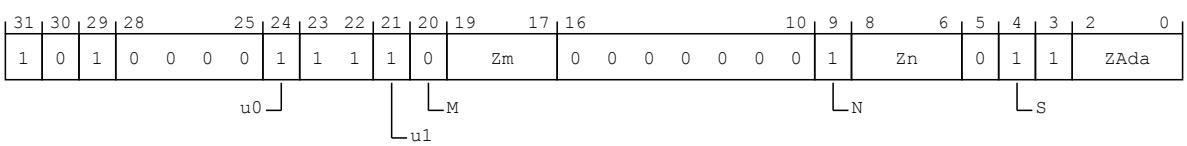

Encoding

```
UMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

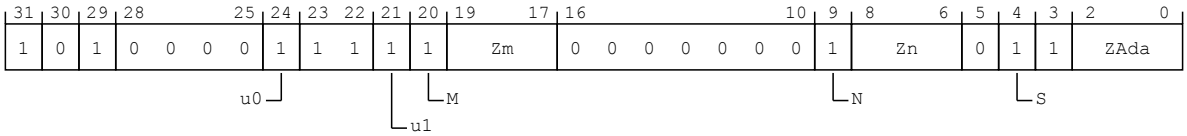

Encoding

```
UMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.313 UMOPA (2-way)

Unsigned 16-bit integer sum of outer products to 32-bit integer, accumulating

This instruction works with a 32-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of unsigned 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		0	1	0	ZAda	
							u0												S				

Encoding

```
UMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 1 do
            if (ActivePredicateElement{PL}(mask1, 2*row + k, esize DIV 2) &&
                  ActivePredicateElement{PL}(mask2, 2*col + k, esize DIV 2)) then
                let op1elt : bits(esize DIV 2) = operand1[(2*row + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = operand2[(2*col + k)*:(esize DIV 2)];
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum + prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.314 UMOPA (4-way)

Unsigned integer sum of outer products, accumulating

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	1	Zm		Pm		Pn		Zn		0	0	0	ZAda	

u0 u1 S

Encoding

```
UMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	1	1	1	Zm		Pm		Pn		Zn		0	0	ZAda	
					u0			u1									S			

Encoding

```
UMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum + prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.315 UMOPS (2-way)

Unsigned 16-bit integer sum of outer products to 32-bit integer, subtracting

This instruction works with a 32-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of unsigned 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		1	1	0	ZAda
							u0												S			

Encoding

```
UMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 1 do
            if (ActivePredicateElement{PL}(mask1, 2*row + k, esize DIV 2) &&
                  ActivePredicateElement{PL}(mask2, 2*col + k, esize DIV 2)) then
                let op1elt : bits(esize DIV 2) = operand1[(2*row + k)*:(esize DIV 2)];
                let op2elt : bits(esize DIV 2) = operand2[(2*col + k)*:(esize DIV 2)];
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum - prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.316 UMOPS (4-way)

Unsigned integer sum of outer products, subtracting

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	1	Zm		Pm		Pn		Zn		1	0	0	ZAda	

u0 u1 S

Encoding

```
UMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	1	1	1	Zm		Pm		Pn		Zn		1	0	ZAda	
					u0			u1									S			

Encoding

```
UMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum - prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.317 UQCVT (two registers)

Multi-vector unsigned 32-bit integer saturating extract narrow to 16-bit integer

This instruction saturates the unsigned 32-bit integer value in each element of the two source vectors to half the original source element width, and places the 16-bit results in the half-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

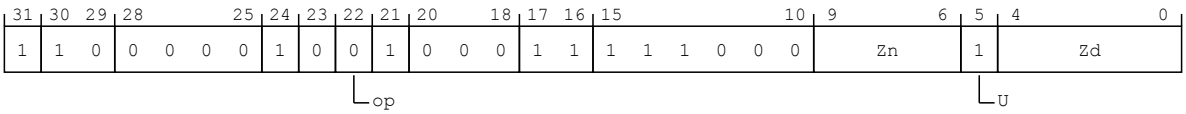

Encoding

```
UQCVT  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for r = 0 to 1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : integer = UInt(operand[e*:(2 * esize)]);
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.318 UQCVT (four registers)

Multi-vector unsigned saturating extract narrow

This instruction saturates the unsigned integer value in each element of the four source vectors to quarter the original source element width, and places the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

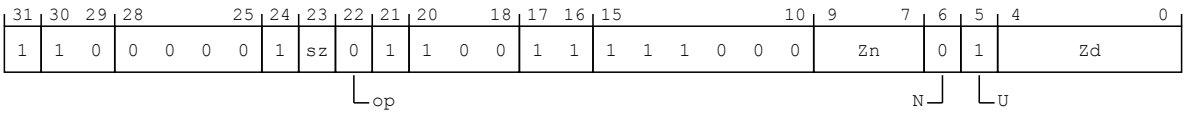

Encoding

```
UQCVT  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(sz);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Tb>
0	S
1	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : integer = UInt(operand[e*:(4 * esize)]);
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.319 UQCVTN

Multi-vector unsigned saturating extract narrow to interleaved integer

This instruction saturates the unsigned integer value in each element of the four source vectors to quarter the original source element width, and places the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2

(FEAT_SME2)

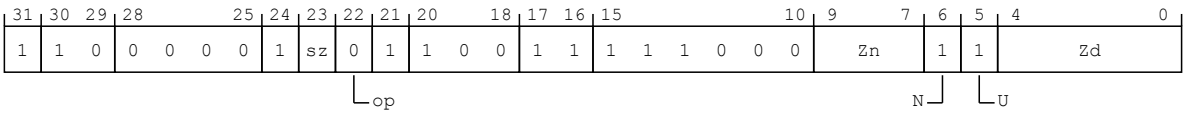

Encoding

```
UQCVTN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(sz);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>

Is the size specifier, encoded in “sz”:

sz	<T>
0	B
1	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Tb>

Is the size specifier, encoded in “sz”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Tb>
0	S
1	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 3 do
        let operand : bits(VL) = Z{}(n+i);
        let element : integer = UInt(operand[e*:(4 * esize)]);
        result[(4*e + i)*:esize] = UnsignedSat{esize}(element);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.320 UQRSHR (two registers)

Multi-vector unsigned 32-bit integer saturating rounding shift right narrow by immediate to 16-bit integer

This instruction shifts right by an immediate value the unsigned integer value in each element of the two source vectors, and places the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15		13	12		10	9			6	5	4			0
1	1	0	0	0	0	0	1	1	1	1	0	imm4				1	1	0	1	0	1	Zn				1	Zd			
											op															U				

Encoding

```
UQRSHR  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 16;
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd);
let shift : integer = esize - UInt(imm4);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as "Zn" times 2 plus 1.

<const>

Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (2 * esize);
var result : bits(VL);

for r = 0 to 1 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(2 * esize) = operand[e*:(2 * esize)];
        let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(res);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.321 UQRSHR (four registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

This instruction shifts right by an immediate value the unsigned integer value in each element of the four source vectors, and places the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	11	10	9	7	6	5	4	0
1	1	0	0	0	1	tsize		1	imm5		1	1	0	1	1	0	Zn	0	1	Zd	
															N			op	U		

Encoding

```
UQRSHR  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if tsize == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
let shift : integer = (8 * esize) - UInt(tsize::imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tsize”:

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<const>

Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in “tsize:imm5”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    for e = 0 to elements-1 do
        let element : bits(4 * esize) = operand[e*:(4 * esize)];
        let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
        result[(r*elements + e)*:esize] = UnsignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.322 UQRSHRN

Multi-vector unsigned saturating rounding shift right narrow by immediate to interleaved integer

This instruction shifts right by an immediate value the unsigned integer value in each element of the four source vectors, and places the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	11	10	9	7	6	5	4	0
1	1	0	0	0	1	tsize		1	imm5		1	1	0	1	1	Zn		0	1	Zd	
															N			op	U		

Encoding

```
UQRSHRN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if tsize == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << HighestSetBitNZ(tsize);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd);
let shift : integer = (8 * esize) - UInt(tsize::imm5);
```

Assembler Symbols

<Zd>

Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>

Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as "Zn" times 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Tb>

Is the size specifier, encoded in “tsize”:

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

<const>

Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in “tsize:imm5”.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV (4 * esize);
var result : bits(VL);

for e = 0 to elements-1 do
    for i = 0 to 3 do
        let operand : bits(VL) = Z{}(n+i);
        let element : bits(4 * esize) = operand[e*:(4 * esize)];
        let res : integer = (UInt(element) + (1 << (shift-1))) >> shift;
        result[(4*e + i)*:esize] = UnsignedSat{esize}(res);
    end;
end;

Z{VL}(d) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.323 URSHL (multiple and single vector)

Multi-vector unsigned rounding shift left by vector

This instruction shifts the unsigned elements of the two or four first source vectors by the corresponding elements of the second source vector, and destructively places the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7		5	4			1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	1	0	0	0	1	Zdn				1
																								opc							U

Encoding

```
URSHL  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15				11	10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	0	0	0	1	Zdn			0	1
																								opc							U

Encoding

```
URSHL  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt('0'::Zm);
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m);
    for e = 0 to elements-1 do
        let element : integer =  UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.324 URSHL (multiple vectors)

Multi-vector unsigned rounding shift left

This instruction shifts the unsigned elements of the two or four first source vectors by the corresponding elements of the two or four second source vectors, and destructively places the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			17	16						10	9	8	7		5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	1	0	1	1	0	0	1	0	0	0	1	Zdn				1
																								opc							U

Encoding

```
URSHL  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'0');
let m : integer = UInt(Zm::'0');
let nreg : integer{} = 2;
```

Four registers

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17							10	9	8	7		5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	0	1	0	0	0	1	Zdn		0	1
																									opc						U

Encoding

```
URSHL  { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let dn : integer = UInt(Zdn::'00');
let m : integer = UInt(Zm::'00');
let nreg : integer{} = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zdn1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zdn2>

Is the name of the second scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 2 plus 1.

<Zm1>

For the “Two registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zdn4>

Is the name of the fourth scalable vector register of the destination and first source multi-vector group, encoded as “Zdn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
var results : array [[4]] of bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(dn+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for e = 0 to elements-1 do
        let element : integer =  UInt(operand1[e*:esize]);
        var shift : integer = ShiftSat(SInt(operand2[e*:esize]), esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        var res : integer;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        end;
        results[[r]][e*:esize] = res[esize-1:0];
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(dn+r) = results[[r]];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.325 USDOT (4-way, multiple and indexed vector)

Multi-vector unsigned by signed 8-bit integer dot product by indexed element to 32-bit integer

This instruction calculates the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2		Zn				1	0	1	off3		
																			op								U				

Encoding

```
USDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn				0	1	0	1	off3		
																			op									U				

Encoding

```
USDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        for i = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.326 USDOT (4-way, multiple and single vector)

Multi-vector unsigned by signed 8-bit integer dot product by vector to 32-bit integer

This instruction calculates the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm	0	Rv	1	0	1	Zn	0	1	off3	

U

Encoding

```
USDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm	0	Rv	1	0	1	Zn	0	1	off3	

U

Encoding

```
USDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.327 USDOT (4-way, multiple vectors)

Multi-vector unsigned by signed 8-bit integer dot product to 32-bit integer

This instruction calculates the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to corresponding 32-bit element of the ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	17	16	15	14	13	12	10	9	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	0	1	Zm		0	0	Rv		1	0 1	Zn		0	0	1	off3

Encoding

```
USDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(off3);
let nreg : integer{} = 2;
```

Four ZA single-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	18	17	16	15	14	13	12	10	9	7	6	5	4	3	2	0
1	1	0	0	0	0	0	1	1	0	1	Zm		0	1	0	Rv		1	0 1	Zn		0	0	0	1	off3

Encoding

```
USDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(off3);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA single-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.328 USMLALL (multiple and indexed vector)

Multi-vector unsigned by signed 8-bit integer multiply-add by indexed element to 32-bit integer

This instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors by each signed 8-bit indexed element of the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index that selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l			Zn				0	0	1	off2	

U, S, op

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn				1	0	0	i4l		o1

op, U, S

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1		Zm			1		Rv		0		i4h		Zn	0	1	0	0	i4l		o1

op, U, S

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let index : integer = UInt(i4h::i4l);
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the element index, in the range 0 to 15, encoded in the “i4h:i4l” fields.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let segmentbase : integer = e - (e MOD eltspersegment);
            let s : integer = 4 * segmentbase + index;
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[s*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.329 USMLALL (multiple and single vector)

Multi-vector unsigned by signed 8-bit integer multiply-add by vector to 32-bit integer

This instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors by each signed 8-bit element in the second source vector, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm		0	Rv		0 0 1	Zn		0	0	1	off2

sz, U, S, op

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off2::'00');
let nreg : integer{} = 1;
```

Two ZA quad-vectors

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	19	16	15	14	13	12	10	9	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm		0	Rv		0 0 0	Zn		0	0	1	0	o1

sz, U, S, op

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors (FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12		10	9			5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm				0	Rv		0	0	0	Zn				0	0	1	0	o1
									sz																	U	S	op		

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn);
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn”.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” plus 1 modulo 32.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” plus 3 modulo 32.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}((n+r) MOD 32);
    let operand2 : bits(VL) = Z{}(m);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.330 USMLALL (multiple vectors)

Multi-vector unsigned by signed 8-bit integer multiply-add to 32-bit integer

This instruction multiplies each unsigned 8-bit element in the two or four first source vectors by each signed 8-bit element in the two or four second source vectors, widens each product to 32 bits, and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The quad-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

Two ZA quad-vectors

(FEAT_SME2)

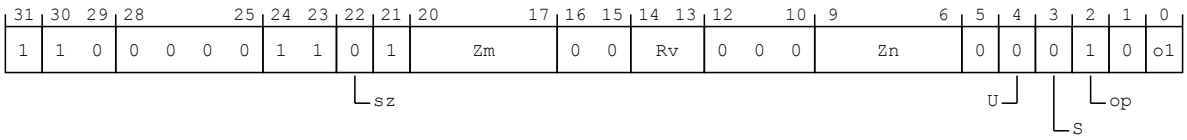

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm::'0');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 2;
```

Four ZA quad-vectors

(FEAT_SME2)

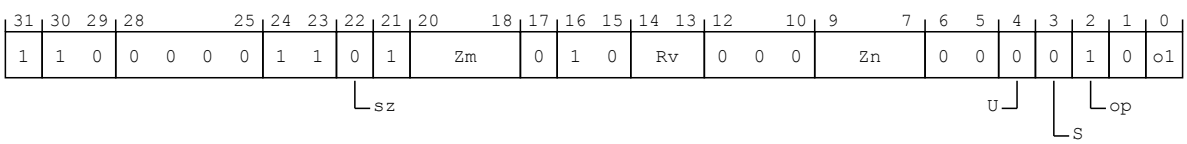

Encoding

```
USMLALL  ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let v : integer = UInt('010'::Rv);
let n : integer = UInt(Zn::'00');
let m : integer = UInt(Zm::'00');
let offset : integer = UInt(o1::'00');
let nreg : integer{} = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

Is the first vector select offset, encoded as “o1” field times 4.

<offs4>

Is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

<Zn1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm1>

For the “Two ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 2.

For the “Four ZA quad-vectors” variant: is the name of the first scalable vector register of the second source multi-vector group, encoded as “Zm” times 4.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, encoded as “Zm” times 2 plus 1.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm4>

Is the name of the fourth scalable vector register of the second source multi-vector group, encoded as “Zm” times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV nreg;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
var result : bits(VL);
vec = vec - (vec MOD 4);

for r = 0 to nreg-1 do
    let operand1 : bits(VL) = Z{}(n+r);
    let operand2 : bits(VL) = Z{}(m+r);
    for i = 0 to 3 do
        let operand3 : bits(VL) = ZAvector{}(vec + i);
        for e = 0 to elements-1 do
            let element1 : integer = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            let product : bits(esize) = (element1 * element2)[esize-1:0];
            result[e*:esize] = operand3[e*:esize] + product;
        end;
        ZAvector{VL}(vec + i) = result;
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.331 USMOP4A

Unsigned by signed integer quarter-tile sum of outer products, accumulating

This instruction generates four independent quarter-tile unsigned by signed integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and accumulates the results to the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit signed values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit signed values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively added to the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

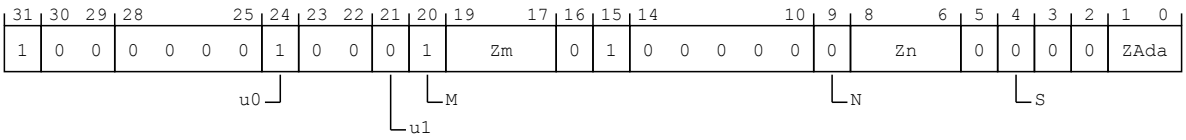

Encoding

```
USMOP4A  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	1	0	0	0	0	0	0	0	Zn			0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
USMOP4A  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28			25	24	23	22	21	20	19		17	16	15	14					10	9	8		6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	0	0	0	Zm			0	1	0	0	0	0	0	0	1	Zn			0	0	0	0	ZAda	

u0 u1 M N S

Encoding

```
USMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

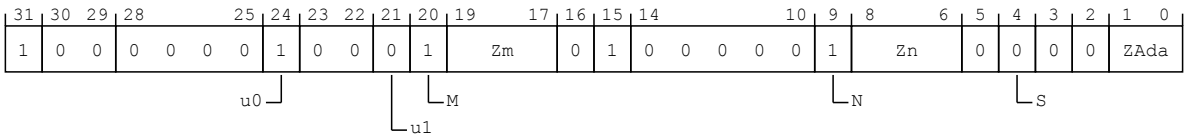

Encoding

```
USMOP4A  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

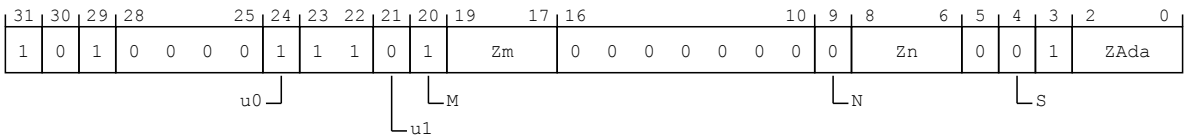

Encoding

```
USMOP4A  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

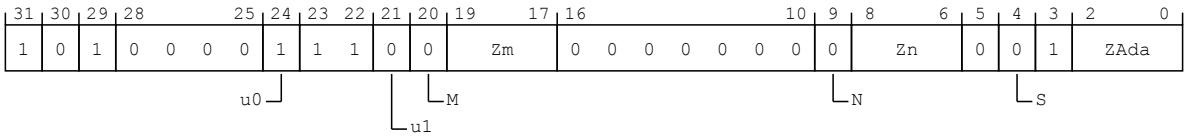

Encoding

```
USMOP4A  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

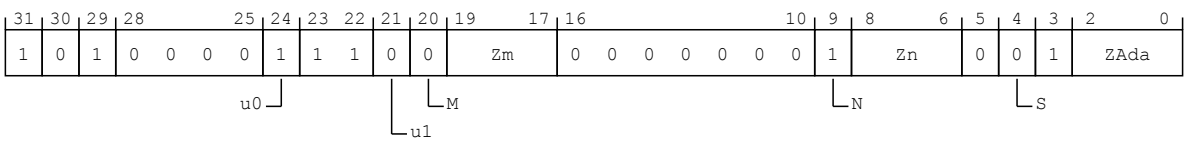

Encoding

```
USMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

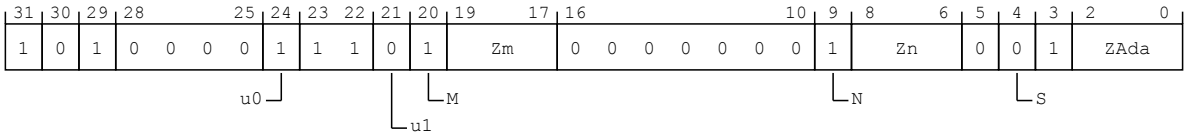

Encoding

```
USMOP4A  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = FALSE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.332 USMOP4S

Unsigned by signed integer quarter-tile sum of outer products, subtracting

This instruction generates four independent quarter-tile unsigned by signed integer sums of outer products from the sub-matrices in the half-vectors of the one or two first and second source vectors and subtracts the results from the corresponding elements of a 32-bit or 64-bit element ZA tile.

In case of the 8-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_S \div 2 \times 4$ sub-matrix of 8-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_S \div 2$ sub-matrix of 8-bit signed values held in the half-vectors of the second source vectors. Each 32-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_S \div 2 \times 4$ sub-matrix. Similarly, each 32-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_S \div 2$ sub-matrix.

In case of the 16-bit integer variant, each of the quarter-tile sums of outer products is generated by multiplying the $SVL_D \div 2 \times 4$ sub-matrix of 16-bit unsigned values held in the half-vectors of the first source vectors by the $4 \times SVL_D \div 2$ sub-matrix of 16-bit signed values held in the half-vectors of the second source vectors. Each 64-bit container of the half-vectors in the first source vectors holds 4 elements of each row of a $SVL_D \div 2 \times 4$ sub-matrix. Similarly, each 64-bit container of the half-vectors in the second source vector holds 4 elements of each column of a $4 \times SVL_D \div 2$ sub-matrix.

The resulting quarter-tile $SVL_S \div 2 \times SVL_S \div 2$ widened 32-bit integer sums of outer products in case of the 8-bit integer variant or $SVL_D \div 2 \times SVL_D \div 2$ widened 64-bit integer sums of outer products in case of the 16-bit integer variant are then destructively subtracted from the 32-bit or 64-bit integer destination tile respectively.

This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

This instruction is unpredicated.

It has encodings from 8 classes: 32-bit, single and multiple vectors, 32-bit, single vectors, 32-bit, multiple and single vectors, 32-bit, multiple vectors, 64-bit, single and multiple vectors, 64-bit, single vectors, 64-bit, multiple and single vectors, and 64-bit, multiple vectors

32-bit, single and multiple vectors

(FEAT_SME_MOP4)

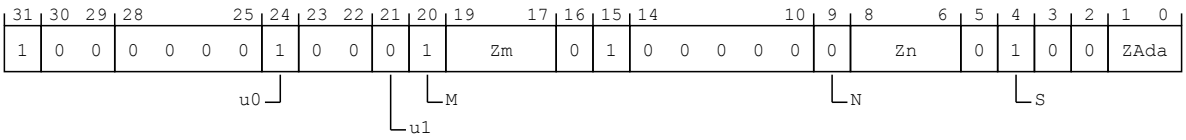

Encoding

```
USMOP4S  <ZAda>.S, <Zn>.B, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		1	0	0	0	0	Zm		0	1	0 0 0 0 0		0	Zn		0	1	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
USMOP4S  <ZAda>.S, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

32-bit, multiple and single vectors

(FEAT_SME_MOP4)

31	30	29	28	25	24	23	22	21	20	19	17	16	15	14	10	9	8	6	5	4	3	2	1	0
1	0	0	0 0 0 0		1	0	0	0	0	Zm		0	1	0 0 0 0 0		1	Zn		0	1	0	0	ZAda	

u0, u1, M, N, S

Encoding

```
USMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

32-bit, multiple vectors

(FEAT_SME_MOP4)

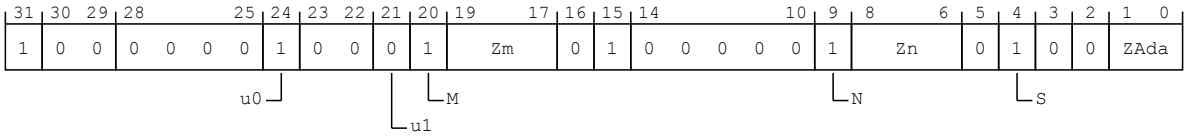

Encoding

```
USMOP4S  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

64-bit, single and multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

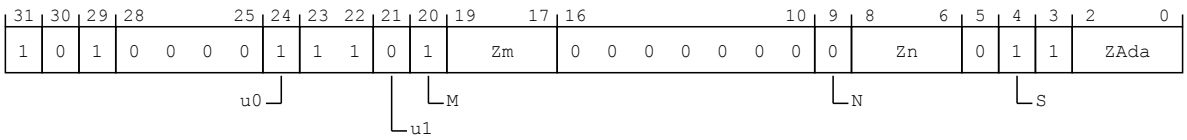

Encoding

```
USMOP4S  <ZAda>.D, <Zn>.H, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

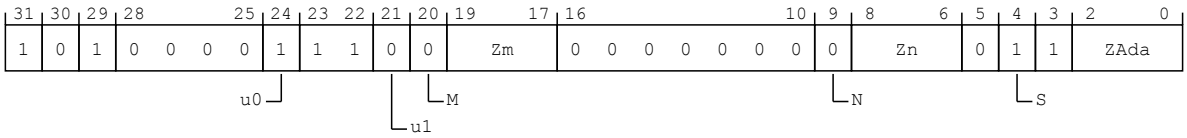

Encoding

```
USMOP4S  <ZAda>.D, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 1;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

64-bit, multiple and single vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

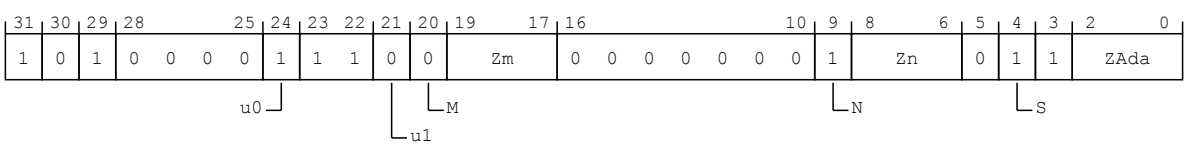

Encoding

```
USMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 1;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, multiple vectors

(FEAT_SME_MOP4 && FEAT_SME_I16I64)

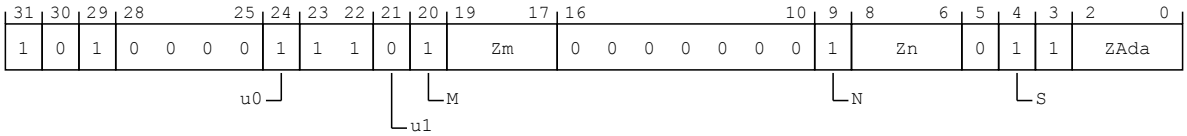

Encoding

```
USMOP4S  <ZAda>.D, { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_MOP4) || !IsFeatureImplemented(FEAT_SME_I16I64) then
    EndOfDecode(Decode_UNDEF);
end;
let esize : integer{} = 64;
let n : integer = UInt('0'::Zn::'0');
let m : integer = UInt('1'::Zm::'0');
let nreg : integer{} = 2;
let mreg : integer = 2;
let da : integer = UInt(ZAda);
let sub_op : boolean = TRUE;
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit, multiple and single vectors”, “32-bit, multiple vectors”, “32-bit, single and multiple vectors”, and “32-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit, multiple and single vectors”, “64-bit, multiple vectors”, “64-bit, single and multiple vectors”, and “64-bit, single vectors” variants: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Zn>

Is the name of the first source scalable vector register, registers in the range Z0-Z15, encoded as “Zn” times 2.

<Zm1>

Is the name of the first scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zm2>

Is the name of the second scalable vector register of the second source multi-vector group, in the range Z16-Z31, encoded as “Zm” times 2 plus 17.

<Zm>

Is the name of the second source scalable vector register, registers in the range Z16-Z31, encoded as “Zm” times 2 plus 16.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, in the range Z0-Z15, encoded as “Zn” times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let hvsize : integer{} = VL DIV 2;
let dim : integer{} = hvsize DIV esize;
let tilesize : integer{} = 4*dim*dim*esize;
let op3 : bits(tilesize) = ZAtile{}(da, esize);
var result : bits(tilesize);
var prod : integer;

for outprod = 0 to 3 do
    let row_hv : integer = outprod DIVRM 2;
    let col_hv : integer = outprod MOD 2;
    let row_base : integer = row_hv * dim;
    let col_base : integer = col_hv * dim;

    let op1 : bits(VL) = Z{}(n + (nreg-1)*col_hv);
    let op2 : bits(VL) = Z{}(m + (mreg-1)*row_hv);

    for row = 0 to dim-1 do
        for col = 0 to dim-1 do
            let row_idx : integer = row_base + row;
            let col_idx : integer = col_base + col;
            let tile_idx : integer = row_idx * dim * 2 + col_idx;

            var sum : bits(esize) = op3[tile_idx*:esize];

            for k = 0 to 3 do
                let op1elt : bits(esize DIV 4) = op1[(4*row_idx + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = op2[(4*col_idx + k)*:(esize DIV 4)];
                let element1 : integer = (if op1_unsigned then UInt(op1elt)
                                          else SInt(op1elt));
                let element2 : integer = (if op2_unsigned then UInt(op2elt)
                                          else SInt(op2elt));
                prod = element1 * element2;
                if sub_op then prod = -prod; end;
                sum = sum + prod;
            end;
            result[tile_idx*:esize] = sum;
        end;
    end;
end;
ZAtile{tilesize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.333 USMOPA (4-way)

Unsigned by signed integer sum of outer products, accumulating

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		0	0	0	ZAda	

u0 u1 S

Encoding

```
USMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	1	1	0	Zm		Pm		Pn		Zn		0	0	ZAda	
					u0			u1									S			

Encoding

```
USMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum + prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.334 USMOPS

Unsigned by signed integer sum of outer products, subtracting

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

This instruction multiplies the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

31	30	29	28			25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	0	Zm		Pm		Pn		Zn		1	0	0	ZAda	
							u0			u1									S				

Encoding

```
USMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 32;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit

(FEAT_SME_I16I64)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	3	2	0
1	0	1	0 0 0	0	1	1	1	0	Zm		Pm		Pn		Zn		1	0	ZAda	
					u0			u1									S			

Encoding

```
USMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_I16I64) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 64;
let a : integer = UInt(Pn);
let b : integer = UInt(Pm);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

For the “32-bit” variant: is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

For the “64-bit” variant: is the name of the ZA tile ZA0-ZA7, encoded in the “ZAda” field.

<Pn>

Is the name of the first governing scalable predicate register P0-P7, encoded in the “Pn” field.

<Pm>

Is the name of the second governing scalable predicate register P0-P7, encoded in the “Pm” field.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let PL : integer{} = VL DIV 8;
let dim : integer{} = VL DIV esize;
let mask1 : bits(PL) = P{}(a);
let mask2 : bits(PL) = P{}(b);
let operand1 : bits(VL) = Z{}(n);
let operand2 : bits(VL) = Z{}(m);
let operand3 : bits(dim*dim*esize) = ZAtile{}(da, esize);
var result : bits(dim*dim*esize);
var prod : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var sum : bits(esize) = operand3[(row*dim+col)*:esize];
        for k = 0 to 3 do
            if (ActivePredicateElement{PL}(mask1, 4*row + k, esize DIV 4) &&
                  ActivePredicateElement{PL}(mask2, 4*col + k, esize DIV 4)) then
                let op1elt : bits(esize DIV 4) = operand1[(4*row + k)*:(esize DIV 4)];
                let op2elt : bits(esize DIV 4) = operand2[(4*col + k)*:(esize DIV 4)];
                let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
                prod = element1 * element2;
                sum = sum - prod;
            end;
        end;
        result[(row*dim+col)*:esize] = sum;
    end;
end;

ZAtile{dim*dim*esize}(da, esize) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.335 USTMOPA

Unsigned by signed 8-bit integer sparse sum of outer products to 32-bit integer, accumulating

This instruction generates unsigned by signed integer sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected two sets of 2-in-4 8-bit unsigned values from each overlapping 32-bit containers of the two $SVL_S \times 4$ sub-matrices in the first source vectors by the four 8-bit signed values from the corresponding 32-bit container of the $4 \times SVL_S$ sub-matrix in the second source vector. The four selected elements from the overlapping 32-bit containers of the first source vectors correspond to two sets of 2-in-4 elements of rows of two $SVL_S \times 4$ sub-matrices. Each 32-bit container of the second source vector holds 4 elements of columns of a compressed $4 \times SVL_S$ sub-matrix.

The two sets of 2-in-4 8-bit unsigned values from overlapping 32-bit containers of the first source vectors are selected by pairs of 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to each 32-bit container of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30	29	28			25	24	23	22	21	20			16	15	14	13	12	11	10	9			6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	1	0	Zm				1	0	0	K	Zk		Zn				i2		0	0	ZAda	
							u0			u1																				

Encoding

```
USTMOPA  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = FALSE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 4;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var erow : array [[4]] of bits(8);
        var ecol : array [[4]] of bits(8);
        for j = 0 to 3 do
            erow[[j]] = Zeros{8};
            ecol[[j]] = op2[(4*col + j)*:8];
        end;
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            var i : integer = 0;
            for e = 0 to 3 do
                if i < 2 && ctrl[(8*col + 4*r + e)*:1] == '1' then
                    erow[[2*r + i]] = op1[(4*row + e)*:8];
                    i = i + 1;
                end;
            end;
        end;
        var sum : bits(32) = op4[(row*dim+col)*:32];
        for j = 0 to 3 do
            let erowval : integer = if op1_unsigned then UInt(erow[[j]]) else SInt(erow[[j]]);
            let ecolval : integer = if op2_unsigned then UInt(ecol[[j]]) else SInt(ecol[[j]]);
            sum = sum + (erowval * ecolval);
        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.336 USVDOT

Multi-vector unsigned by signed 8-bit integer vertical dot product by indexed element to 32-bit integer

This instruction calculates the vertical dot product of corresponding unsigned 8-bit elements from the four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the ZA operand consists of four ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			7	6	5	4	3	2			0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		0	i2		Zn				0	1	0	1	off3			
																			op									U					

Encoding

```
USVDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<index>

Is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 4;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 3 do
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let operand1 : bits(VL) = Z{}(n+i);
            let element1 : integer = UInt(operand1[(4 * e + r)*:(esize DIV 4)]);
            let element2 : integer = SInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.337 UTMOPA (2-way)

Unsigned 16-bit integer sparse sum of outer products to 32-bit integer, accumulating

This instruction generates unsigned integer sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected 2-in-4 16-bit unsigned values from each overlapping 32-bit containers of the two $SVL_S \times 2$ sub-matrices in the first source vectors by the two 16-bit unsigned values from the corresponding 32-bit container of the $2 \times SVL_S$ sub-matrix in the second source vector. The two selected elements from each overlapping 32-bit containers of the first source vectors correspond to 2-in-4 elements of rows of two $SVL_S \times 2$ sub-matrices. Each 32-bit container of the second source vector holds 2 elements of columns of a compressed $2 \times SVL_S$ sub-matrix.

The 2-in-4 16-bit unsigned values from overlapping 32-bit containers of the first source vectors are selected by 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to 32-bit containers of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

31	30 29	28 25	24	23 22	21	20 16	15	14	13	12	11 10	9 6	5 4	3	2	1 0
1	0 0	0 0 0 0	1	0 1	0	Zm	1	0	0	K	Zk	Zn	i2	1	0	ZAda
			u0													

Encoding

```
UTMOPA  <ZAda>.S, { <Zn1>.H-<Zn2>.H }, <Zm>.H, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 8;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var erow : array [[2]] of bits(16);
        var ecol : array [[2]] of bits(16);
        for j = 0 to 1 do
            erow[[j]] = Zeros{16};
            ecol[[j]] = op2[(2*col + j)*:16];
        end;
        var i : integer = 0;
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            for e = 0 to 1 do
                if i < 2 && ctrl[(4*col + 2*r + e)*:1] == '1' then
                    erow[[i]] = op1[(2*row + e)*:16];
                    i = i + 1;
                end;
            end;
        end;
        var sum : bits(32) = op4[(row*dim+col)*:32];
        for j = 0 to 1 do
            let erowval : integer = if unsigned then UInt(erow[[j]]) else SInt(erow[[j]]);
            let ecolval : integer = if unsigned then UInt(ecol[[j]]) else SInt(ecol[[j]]);
            sum = sum + (erowval * ecolval);
        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.338 UTMOPA (4-way)

Unsigned 8-bit integer sparse sum of outer products to 32-bit integer, accumulating

This instruction generates unsigned integer sum of outer products by multiplying the 2-in-4 selected elements from the dense sub-matrices in the two first source vectors with the corresponding elements of the compressed sparse sub-matrix in the second source vector and accumulates the results to the corresponding elements of a 32-bit element ZA tile.

The sum of outer products is generated by multiplying the selected two sets of 2-in-4 8-bit unsigned values from each overlapping 32-bit containers of the two $SVL_S \times 4$ sub-matrices in the first source vectors by the four 8-bit unsigned values from the corresponding 32-bit container of the $4 \times SVL_S$ sub-matrix in the second source vector. The four selected elements from the overlapping 32-bit containers of the first source vectors correspond to two sets of 2-in-4 elements of rows of two $SVL_S \times 4$ sub-matrices. Each 32-bit container of the second source vector holds 4 elements of columns of a compressed $4 \times SVL_S$ sub-matrix.

The two sets of 2-in-4 8-bit unsigned values from overlapping 32-bit containers of the first source vectors are selected by pairs of 4-bit controls in the indexed segment of the control vector register. If the control bit corresponding to an element in the first source vectors is 0, the element is discarded and does not contribute to the sum of products result. If more than two bits of the 4-bit control corresponding to each 32-bit container of the first source vectors are 1, only the elements corresponding to the least two significant bits are selected.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

This instruction is unpredicated.

SME2

(FEAT_SME_TMOP)

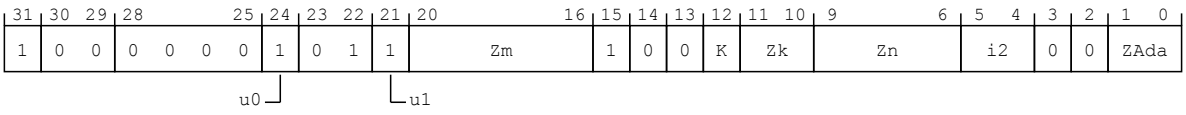

Encoding

```
UTMOPA  <ZAda>.S, { <Zn1>.B-<Zn2>.B }, <Zm>.B, <Zk>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME_TMOP) then EndOfDecode(Decode_UNDEF); end;
let n : integer = UInt(Zn::'0');
let m : integer = UInt(Zm);
let k : integer = UInt('1'::K::'1'::Zk);
let index : integer = UInt(i2);
let da : integer = UInt(ZAda);
let op1_unsigned : boolean = TRUE;
let op2_unsigned : boolean = TRUE;
```

Assembler Symbols

<ZAda>

Is the name of the ZA tile ZA0-ZA3, encoded in the “ZAda” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zk>

Is the name of the control vector register Z20-Z23 or Z28-Z31, encoded in the “K:Zk” fields.

<index>

Is the control segment index, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let dim : integer{} = VL DIV 32;
let csize : integer{} = VL DIV 4;
let op2 : bits(VL) = Z{}(m);
let op3 : bits(VL) = Z{}(k);
let ctrl : bits(csize) = op3[index*:csize];
let op4 : bits(dim*dim*32) = ZAtile{}(da, 32);
var result : bits(dim*dim*32);

for row = 0 to dim-1 do
    for col = 0 to dim-1 do
        var erow : array [[4]] of bits(8);
        var ecol : array [[4]] of bits(8);
        for j = 0 to 3 do
            erow[[j]] = Zeros{8};
            ecol[[j]] = op2[(4*col + j)*:8];
        end;
        for r = 0 to 1 do
            let op1 : bits(VL) = Z{}(n+r);
            var i : integer = 0;
            for e = 0 to 3 do
                if i < 2 && ctrl[(8*col + 4*r + e)*:1] == '1' then
                    erow[[2*r + i]] = op1[(4*row + e)*:8];
                    i = i + 1;
                end;
            end;
        end;
        var sum : bits(32) = op4[(row*dim+col)*:32];
        for j = 0 to 3 do
            let erowval : integer = if op1_unsigned then UInt(erow[[j]]) else SInt(erow[[j]]);
            let ecolval : integer = if op2_unsigned then UInt(ecol[[j]]) else SInt(ecol[[j]]);
            sum = sum + (erowval * ecolval);
        end;
        result[(row*dim+col)*:32] = sum;
    end;
end;
ZAtile{dim*dim*32}(da, 32) = result;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.339 UUNPK

Unpack and zero-extend multi-vector elements

This instruction unpacks and zero-extends elements from one or two source vectors, and writes the zero-extended values into double width elements of the two or four destination vectors. This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers

(FEAT_SME2)

31	30	29	28	25	24	23 22	21	20	18	17	16	15	10	9	5	4	1	0
1	1	0	0 0 0	0	1	size	1	0 0	1	0	1	1 1 1 0 0	0	Zn		Zd		1
																		U

Encoding

```
UUNPK  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let d : integer = UInt(Zd::'0');
let nreg : integer{} = 2;
let unsigned : boolean = TRUE;
```

Four registers

(FEAT_SME2)

31	30	29	28	25	24	23 22	21	20	18	17	16	15	10	9	6	5	4	2	1	0
1	1	0	0 0 0	0	1	size	1	1 0	1	0	1	1 1 1 0 0	0	Zn		0	Zd		0	1
																				U

Encoding

```
UUNPK  { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<Tb>-<Zn2>.<Tb> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '00' then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn::'0');
let d : integer = UInt(Zd::'00');
let nreg : integer{} = 4;
let unsigned : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

For the “Two registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

For the “Four registers” variant: is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb>

Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let hsize : integer{} = esize DIV 2;
let sreg : integer = nreg DIV 2;
var results : array [[4]] of bits(VL);

for r = 0 to sreg-1 do
    let operand : bits(VL) = Z{}(n+r);
    for i = 0 to 1 do
        for e = 0 to elements-1 do
            let element : bits(hsize) = operand[(i*elements + e)*:hsize];
            results[[2*r+i]][e*:esize] = Extend{esize}(element, unsigned);
        end;
    end;
end;

for r = 0 to nreg-1 do
    Z{VL}(d+r) = results[[r]];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.340 UVDOT (2-way)

Multi-vector unsigned 16-bit integer vertical dot product by indexed element to 32-bit integer

This instruction calculates the vertical dot product of the corresponding two unsigned 16-bit integer values held in the two first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to 3.

The single-vector group within each half of the ZA array is selected by the sum of the vector select register and offset, modulo half the number of ZA array vectors.

The `vector group` symbol `VGx2` indicates that the ZA operand consists of two ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9			6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				1	1	0	off3		
																			op								U				

Encoding

```
UVDOT  ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'0');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 2.

<Zn2>

Is the name of the second scalable vector register of the first source multi-vector group, encoded as “Zn” times 2 plus 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

Is the immediate index of a pair of 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 2;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 1 do
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 1 do
            let operand1 : bits(VL) = Z{}(n+i);
            let element1 : integer = UInt(operand1[(2 * e + r)*:(esize DIV 2)]);
            let element2 : integer = UInt(operand2[(2 * s + i)*:(esize DIV 2)]);
            sum = sum + element1 * element2;
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.341 UVDOT (4-way)

Multi-vector unsigned integer vertical dot product by indexed element

This instruction calculates the vertical dot product of the corresponding four unsigned 8-bit or 16-bit integer values held in the four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index that selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment.

The single-vector group within each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo quarter the number of ZA array vectors.

The `vector group` symbol `VGx4` indicates that the ZA operand consists of four ZA single-vector groups. The vector group symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 8-bit to 32-bit and 16-bit to 64-bit

8-bit to 32-bit

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		0	i2		Zn			0	1	1	0	off3		
																			op								U				

Encoding

```
UVDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 32;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i2);
```

16-bit to 64-bit

(FEAT_SME2 && FEAT_SME_I16I64)

31	30	29	28			25	24	23	22	21	20	19			16	15	14	13	12	11	10	9		7	6	5	4	3	2		0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		0	1	i1	Zn			0	0	1	1	off3		
																				op							U				

Encoding

```
UVDOT  ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !(IsFeatureImplemented(FEAT_SME2) && IsFeatureImplemented(FEAT_SME_I16I64)) then
    EndOfDecode(Decode_UNDEF);
end;
let v : integer = UInt('010'::Rv);
let esize : integer{} = 64;
let n : integer = UInt(Zn::'00');
let m : integer = UInt('0'::Zm);
let offset : integer = UInt(off3);
let index : integer = UInt(i1);
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

<Zn1>

Is the name of the first scalable vector register of the first source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the first source multi-vector group, encoded as “Zn” times 4 plus 3.

<Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<index>

For the “8-bit to 32-bit” variant: is the immediate index of a group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the “16-bit to 64-bit” variant: is the immediate index of a group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let elements : integer = VL DIV esize;
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV 4;
let eltspersegment : integer = 128 DIV esize;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
let operand2 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 3 do
    let operand3 : bits(VL) = ZAvector{}(vec);
    for e = 0 to elements-1 do
        let segmentbase : integer = e - (e MOD eltspersegment);
        let s : integer = segmentbase + index;
        var sum : bits(esize) = operand3[e*:esize];
        for i = 0 to 3 do
            let operand1 : bits(VL) = Z{}(n+i);
            let element1 : integer = UInt(operand1[(4 * e + r)*:(esize DIV 4)]);
            let element2 : integer = UInt(operand2[(4 * s + i)*:(esize DIV 4)]);
            sum = sum + element1 * element2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
        result[e*:esize] = sum;
    end;
    ZAvector{VL}(vec) = result;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.342 UZP (four registers)

Concatenate elements from four vectors

This instruction concatenates every fourth element from each of the four source vectors and places them in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	1	0	1	1	1	0	0	0	Zn			0	0	Zd			1	0
																														op	

Encoding

```
UZP  { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' && MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
```

128-bit element

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9		7	6	5	4		2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn			0	0	Zd			1	0
																														op	

Encoding

```
UZP  { <Zd1>.Q-<Zd4>.Q }, { <Zn1>.Q-<Zn4>.Q }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 512 then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
if VL < esize * 4 then EndOfDecode(Decode_UNDEF); end;
let quads : integer = VL DIV (esize * 4);
var result0 : bits(VL);
var result1 : bits(VL);
var result2 : bits(VL);
var result3 : bits(VL);

for r = 0 to 3 do
    let operand : bits(VL) = Z{}(n+r);
    let base : integer = r * quads;
    for q = 0 to quads-1 do
        result0[(base+q)*:esize] = operand[(4*q+0)*:esize];
        result1[(base+q)*:esize] = operand[(4*q+1)*:esize];
        result2[(base+q)*:esize] = operand[(4*q+2)*:esize];
        result3[(base+q)*:esize] = operand[(4*q+3)*:esize];
    end;
end;

Z{VL}(d+0) = result0;
Z{VL}(d+1) = result1;
Z{VL}(d+2) = result2;
Z{VL}(d+3) = result3;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.343 UZP (two registers)

Concatenate elements from two vectors

This instruction concatenates every second element from each of the first and second source vectors and places them in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	1	0
1	1	0	0	0	0	0	1	size	1	Zm	1	1	0	1	0	0	Zn	Zd	1
																			op

Encoding

```
UZP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
```

128-bit element

(FEAT_SME2)

31	30	29	28	25	24	23	22	21	20	16	15	13	12	10	9	5	4	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm	1	1	0	1	0	1	Zn	Zd	1
																			op

Encoding

```
UZP  { <Zd1>.Q-<Zd2>.Q }, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
if VL < esize * 2 then EndOfDecode(Decode_UNDEF); end;
let pairs : integer = VL DIV (esize * 2);
var result0 : bits(VL);
var result1 : bits(VL);

for r = 0 to 1 do
    let base : integer = r * pairs;
    let operand : bits(VL) = if r == 0 then Z{VL}(n) else Z{VL}(m);
    for p = 0 to pairs-1 do
        result0[(base+p)*:esize] = operand[(2*p+0)*:esize];
        result1[(base+p)*:esize] = operand[(2*p+1)*:esize];
    end;
end;

Z{VL}(d+0) = result0;
Z{VL}(d+1) = result1;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.344 ZERO (single-vector)

Zero ZA single-vector groups

This instruction zeroes two or four ZA single-vector groups.

The single-vector group within each half of or each quarter of the ZA array is selected by the sum of the vector select register and offset, modulo half or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA single-vector groups respectively.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	Rv		0	0	0	0	0	0	0	0	0	0	off3		
														opc																	

Encoding

```
ZERO  ZA.D[<Wv>, <offs>, VGx2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let ngrp : integer = 2;
```

Four ZA single-vectors

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	Rv		0	0	0	0	0	0	0	0	0	0	off3		
														opc																	

Encoding

```
ZERO  ZA.D[<Wv>, <offs>, VGx4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3);
let ngrp : integer = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs>

Is the vector select offset, in the range 0 to 7, encoded in the “off3” field.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV ngrp;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
for r = 0 to ngrp-1 do
    ZAvector{VL}(vec) = Zeros{VL};
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.345 ZERO (double-vector)

Zero ZA double-vector groups

This instruction zeroes one, two, or four ZA double-vector groups.

The double-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA double-vector groups respectively.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors, and Four ZA double-vectors

One ZA double-vector

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2		0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	Rv		0	0	0	0	0	0	0	0	0	0	off3		
														opc																	

Encoding

```
ZERO  ZA.D[<Wv>, <offs1>:<offs2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off3::'0');
let ngrp : integer = 1;
let nvec : integer = 2;
```

Two ZA double-vectors

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	Rv		0	0	0	0	0	0	0	0	0	0	0	off2	
														opc																	

Encoding

```
ZERO  ZA.D[<Wv>, <offs1>:<offs2>, VGx2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off2::'0');
let ngrp : integer = 2;
let nvec : integer = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Four ZA double-vectors

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	Rv		0	0	0	0	0	0	0	0	0	0	0	off2	
															opc																

Encoding

```
ZERO  ZA.D[<Wv>, <offs1>:<offs2>, VGx4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off2::'0');
let ngrp : integer = 4;
let nvec : integer = 2;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA double-vector” variant: is the first vector select offset, encoded as “off3” field times 2.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the first vector select offset, encoded as “off2” field times 2.

<offs2>

For the “One ZA double-vector” variant: is the second vector select offset, encoded as “off3” field times 2 plus 1.

For the “Four ZA double-vectors” and “Two ZA double-vectors” variants: is the second vector select offset, encoded as “off2” field times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV ngrp;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
vec = vec - (vec MOD nvec);
for r = 0 to ngrp-1 do
    for i = 0 to nvec-1 do
        ZAvector{VL}(vec + i) = Zeros{VL};
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.346 ZERO (quad-vector)

Zero ZA quad-vector groups

This instruction zeroes one, two, or four ZA quad-vector groups.

The quad-vector group within all of, each half of, or each quarter of the ZA array is selected by the sum of the vector select register and offset range, modulo all, half, or quarter the number of ZA array vectors.

The `vector group` symbol, `VGx2` or `VGx4`, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors, and Four ZA quad-vectors

One ZA quad-vector

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	Rv		0	0	0	0	0	0	0	0	0	0	0	off2	
														opc																	

Encoding

```
ZERO  ZA.D[<Wv>, <offs1>:<offs4>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(off2::'00');
let ngrp : integer = 1;
let nvec : integer = 4;
```

Two ZA quad-vectors

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	Rv		0	0	0	0	0	0	0	0	0	0	0	0	o1
														opc																	

Encoding

```
ZERO  ZA.D[<Wv>, <offs1>:<offs4>, VGx2]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(o1::'00');
let ngrp : integer = 2;
let nvec : integer = 4;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Four ZA quad-vectors

(FEAT_SME2p1)

31	30	29	28			25	24						18	17		15	14	13	12									3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	Rv		0	0	0	0	0	0	0	0	0	0	0	0	o1
															opc																

Encoding

```
ZERO  ZA.D[<Wv>, <offs1>:<offs4>, VGx4]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2p1) then EndOfDecode(Decode_UNDEF); end;
let v : integer = UInt('010'::Rv);
let offset : integer = UInt(o1::'00');
let ngrp : integer = 4;
let nvec : integer = 4;
```

Assembler Symbols

<Wv>

Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>

For the “One ZA quad-vector” variant: is the first vector select offset, encoded as “off2” field times 4.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the first vector select offset, encoded as “o1” field times 4.

<offs4>

For the “One ZA quad-vector” variant: is the fourth vector select offset, encoded as “off2” field times 4 plus 3.

For the “Four ZA quad-vectors” and “Two ZA quad-vectors” variants: is the fourth vector select offset, encoded as “o1” field times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
let VL : integer{} = CurrentVL();
let vectors : integer = VL DIV 8;
let vstride : integer = vectors DIV ngrp;
let vbase : bits(32) = X{}(v);
var vec : integer = (UInt(vbase) + offset) MOD vstride;
vec = vec - (vec MOD nvec);
for r = 0 to ngrp-1 do
    for i = 0 to nvec-1 do
        ZAvector{VL}(vec + i) = Zeros{VL};
    end;
    vec = vec + vstride;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.347 ZERO (tiles)

Zero a list of 64-bit element ZA tiles

This instruction zeroes all bytes within each of the up to eight listed 64-bit element tiles named `ZA0.D` to `ZA7.D`, leaving the other 64-bit element tiles unmodified.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

For programmer convenience an assembler must also accept the names of 32-bit, 16-bit, and 8-bit element tiles that are converted into the corresponding set of 64-bit element tiles.

In accordance with the architecturally defined mapping between different element size tiles:

- Zeroing the 8-bit element tile name `ZA0.B`, or the entire array name `ZA`, is equivalent to zeroing all eight 64-bit element tiles named `ZA0.D` to `ZA7.D`.
- Zeroing the 16-bit element tile name `ZA0.H` is equivalent to zeroing 64-bit element tiles named `ZA0.D`, `ZA2.D`, `ZA4.D`, and `ZA6.D`.
- Zeroing the 16-bit element tile name `ZA1.H` is equivalent to zeroing 64-bit element tiles named `ZA1.D`, `ZA3.D`, `ZA5.D`, and `ZA7.D`.
- Zeroing the 32-bit element tile name `ZA0.S` is equivalent to zeroing 64-bit element tiles named `ZA0.D` and `ZA4.D`.
- Zeroing the 32-bit element tile name `ZA1.S` is equivalent to zeroing 64-bit element tiles named `ZA1.D` and `ZA5.D`.
- Zeroing the 32-bit element tile name `ZA2.S` is equivalent to zeroing 64-bit element tiles named `ZA2.D` and `ZA6.D`.
- Zeroing the 32-bit element tile name `ZA3.S` is equivalent to zeroing 64-bit element tiles named `ZA3.D` and `ZA7.D`.

The preferred disassembly of this instruction uses the shortest list of tile names that represent the encoded immediate mask.

For example:

- An immediate that encodes 64-bit element tiles `ZA0.D`, `ZA1.D`, `ZA4.D`, and `ZA5.D` is disassembled as {`ZA0.S`, `ZA1.S`}.
- An immediate that encodes 64-bit element tiles `ZA0.D`, `ZA2.D`, `ZA4.D`, and `ZA6.D` is disassembled as {`ZA0.H`}.
- An all-ones immediate is disassembled as {`ZA`}.
- An all-zeros immediate is disassembled as an empty list { }.

SME

(FEAT_SME)

31	30	29	28			25	24						18	17								8	7								0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	imm8								

Encoding

```
ZERO  { {<mask>} }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME) then EndOfDecode(Decode_UNDEF); end;
let mask : bits(8) = imm8;
let esize : integer{} = 64;
```

Assembler Symbols

<mask>

Is the optional list of up to eight 64-bit element tile names separated by commas, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckSMEAndZAEnabled();
let SVL : integer{} = CurrentSVL();
let dim : integer{} = SVL DIV esize;
let result : bits(dim*dim*esize) = Zeros{};

for i = 0 to 7 do
    if mask[i] == '1' then ZAtile{dim*dim*esize}(i, esize) = result; end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.348 ZERO (table)

Zero ZT0

This instruction zeroes all bytes of the ZT0 register.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2

(FEAT_SME2)

31	30	29	28			25	24					18	17												4	3			0
1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
																													opc

Encoding

```
ZERO  { ZT0 }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
```

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();

ZT0{512}() = Zeros{512};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.349 ZIP (four registers)

Interleave elements from four vectors

This instruction places the four-way interleaved elements from the four source vectors in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	1	0	1	1	1	0	0	0	Zn		0	0	Zd		0	0
																												op	

Encoding

```
ZIP  { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T> }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if size == '11' && MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
```

128-bit element

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20		18	17	16	15					10	9	7	6	5	4	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn		0	0	Zd		0	0
																												op	

Encoding

```
ZIP  { <Zd1>.Q-<Zd4>.Q }, { <Zn1>.Q-<Zn4>.Q }
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 512 then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn::'00');
let d : integer = UInt(Zd::'00');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 4.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd4>

Is the name of the fourth scalable vector register of the destination multi-vector group, encoded as “Zd” times 4 plus 3.

<Zn1>

Is the name of the first scalable vector register of the source multi-vector group, encoded as “Zn” times 4.

<Zn4>

Is the name of the fourth scalable vector register of the source multi-vector group, encoded as “Zn” times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
if VL < esize * 4 then EndOfDecode(Decode_UNDEF); end;
let quads : integer = VL DIV (esize * 4);
let operand0 : bits(VL) = Z{}(n);
let operand1 : bits(VL) = Z{}(n+1);
let operand2 : bits(VL) = Z{}(n+2);
let operand3 : bits(VL) = Z{}(n+3);
var result : bits(VL);

for r = 0 to 3 do
    let base : integer = r * quads;
    for q = 0 to quads-1 do
        result[(4*q+0)*:esize] = operand0[(base+q)*:esize];
        result[(4*q+1)*:esize] = operand1[(base+q)*:esize];
        result[(4*q+2)*:esize] = operand2[(base+q)*:esize];
        result[(4*q+3)*:esize] = operand3[(base+q)*:esize];
    end;
    Z{VL}(d+r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C9.2.350 ZIP (two registers)

Interleave elements from two vectors

This instruction places the two-way interleaved elements from the first and second source vectors in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4			1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	1	0	0	Zn				Zd				0

op

Encoding

```
ZIP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 8 << UInt(size);
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
```

128-bit element

(FEAT_SME2)

31	30	29	28			25	24	23	22	21	20			16	15		13	12		10	9			5	4			1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				1	1	0	1	0	1	Zn				Zd				0

op

Encoding

```
ZIP  { <Zd1>.Q-<Zd2>.Q }, <Zn>.Q, <Zm>.Q
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SME2) then EndOfDecode(Decode_UNDEF); end;
if MaxImplementedSVL() < 256 then EndOfDecode(Decode_UNDEF); end;
let esize : integer{} = 128;
let n : integer = UInt(Zn);
let m : integer = UInt(Zm);
let d : integer = UInt(Zd::'0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Zd1>

Is the name of the first scalable vector register of the destination multi-vector group, encoded as “Zd” times 2.

<T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd2>

Is the name of the second scalable vector register of the destination multi-vector group, encoded as “Zd” times 2 plus 1.

<Zn>

Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckStreamingSVEEnabled();
let VL : integer{} = CurrentVL();
if VL < esize * 2 then EndOfDecode(Decode_UNDEF); end;
let pairs : integer = VL DIV (esize * 2);
let operand0 : bits(VL) = Z{}(n);
let operand1 : bits(VL) = Z{}(m);
var result : bits(VL);

for r = 0 to 1 do
    let base : integer = r * pairs;
    for p = 0 to pairs-1 do
        result[(2*p+0)*:esize] = operand0[(base+p)*:esize];
        result[(2*p+1)*:esize] = operand1[(base+p)*:esize];
    end;
    Z{VL}(d+r) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part D
The AArch64 System Level Architecture

Chapter D1
The AArch64 System Level Programmers' Model

This chapter describes the AArch64 system level programmers' model. It contains the following sections:

- Exception levels.
- Registers for instruction processing and exception handling.
- System register locking.
- Exceptions.
- Process state, PSTATE.
- Resets and power domains.
- Mechanisms for entering a low-power state.
- Self-hosted debug.
- Event monitors.
- Interprocessing.
- Check Feature.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.1 Exception levels

R_{FYTFG} The architecture defines four *Exception levels*: EL0, EL1, EL2, and EL3.

R_{VPSDB} EL3 is the highest Exception level and EL0 the lowest. Therefore, EL3 is higher than EL2, EL2 is higher than EL1, and EL1 is higher than EL0.

R_{NXCRB} A PE must implement EL1 and EL0.

R_{KPZJW} A PE with FEAT_RME must implement EL3 and EL2.
In a PE without FEAT_RME, all the following are true:
- Whether EL3 is implemented is IMPLEMENTATION DEFINED.
- Whether EL2 is implemented is IMPLEMENTATION DEFINED.
- The PE is not required to implement a contiguous set of Exception levels.

I_{WRRKQ} EL2 provides support for the virtualization of EL0 and EL1.

R_{CCQWK} Unprivileged execution is any execution that occurs at EL0.

R_{NZZNS} The current Exception level changes only when any of the following occur:
- Taking an exception.
- Returning from an exception.
- Processor reset.
- Exiting from Debug state.
- If in Debug state, executing a DCPSx instruction.
- If in Debug state, executing a DRPS instruction.

R_{XRQKF} The *target Exception level* is the Exception level to which an exception is taken.

R_{FFJBB} Each exception type has a target Exception level that is either:
- Implicit in the type of the exception.
- Defined by configuration bits in the System registers.

R_{TKYYF} An exception cannot be taken to EL0.

R_{QNTPB} An exception cannot cause entry to a lower Exception level.

D1.1.1 Execution state

R_{HTYJH} The architecture defines two Execution states. The AArch32 Execution state uses 32-bit processing. The AArch64 Execution state uses 64-bit processing.

I_{WKKGD} The AArch32 Execution state is compatible with Armv7-A operation.

R_{NKNVR} An Exception level is described as:
- *Using AArch64* when execution in that Exception level is in the AArch64 Execution state.
- *Using AArch32* when execution in that Exception level is in the AArch32 Execution state.

R_{VZTWD} The PE can only change Execution state at reset, or when changing Exception level.

R_{QNVJL} If an Exception level is using AArch32, then all lower Exception levels are using AArch32.

R_{BDMRT} If an Exception level is using AArch64, then all higher Exception levels are using AArch64.

R_{MWGKL} Interaction between the AArch64 and AArch32 Execution states is called *interprocessing*.

I_{KSMPY} In this chapter, the described behavior assumes the highest implemented Exception level is using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.1.2 Security states

R_{WPBBJ} The architecture defines the following *Security states*:

- Secure state.
- Non-secure state.
- If FEAT_RME is implemented, also both of:
  - Realm state.
  - Root state.

I_{XKGPG} Execution in:

- Secure state cannot be observed or modified by an agent associated with any of:
  - Non-secure state.
  - Realm state.
- Realm state cannot be observed or modified by an agent associated with any of:
  - Non-secure state.
  - Secure state.
- Root state cannot be observed or modified by an agent associated with any other Security state.

R_{NRXJT} The architecture defines the following *Physical Address* (PA) spaces:

- Secure.
- Non-secure.
- If FEAT_RME is implemented, also the following:
  - Realm.
  - Root.
  - If FEAT_RME_GDI is implemented, also both of:
    - System Agent.
    - Non-secure Protected.

I_{RKGZF} Memory assigned to the:

- Secure physical address space cannot be read or modified by an agent associated with any of:
  - Non-secure state.
  - Realm state.
- Realm physical address space cannot be read or modified by an agent associated with any of:
  - Non-secure state.
  - Secure state.
- Root physical address space cannot be read or modified by an agent associated with any other Security state.

R_{ZFPTR} EL3 is always in:

- Secure state in a PE without FEAT_RME.
- Root state in a PE with FEAT_RME.

I_{HGPBB} The Effective value of SCR_EL3.{NSE, NS} selects the Security state of EL2 and lower.

I_{YFMYJ} For details on the System Agent and Non-secure Protected PA spaces, see Granular Data Isolation.

R_{SGMFG} EL2 and lower cannot be in Root state.

R_{DVTDD} EL2 can only be in Secure state if, in addition to the Effective value of SCR_EL3.{NSE, NS} selecting Secure state, both of the following are true:

- FEAT_SEL2 is implemented.
- The Effective value of SCR_EL3.EEL2 is `0b1`.

I_{ZZCQM} The state of SCR_EL3.NS, and SCR_EL3.NSE if FEAT_RME is implemented, can only change in EL3.

I_{DJJQJ} In a PE with FEAT_RME, all the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Root state, Realm state, and Non-secure state must be implemented.
- Whether Secure state is implemented is IMPLEMENTATION DEFINED.
- If Secure state is implemented, the PE must also implement FEAT_SEL2.

IZLZWD In a PE with FEAT_RME, software can discover whether Secure state is implemented by discovering whether FEAT_SEL2 is implemented from ID_AA64PFR0_EL1.SEL2.

IYBGYV Figure D1-1 illustrates the Security states in a PE with FEAT_RME.

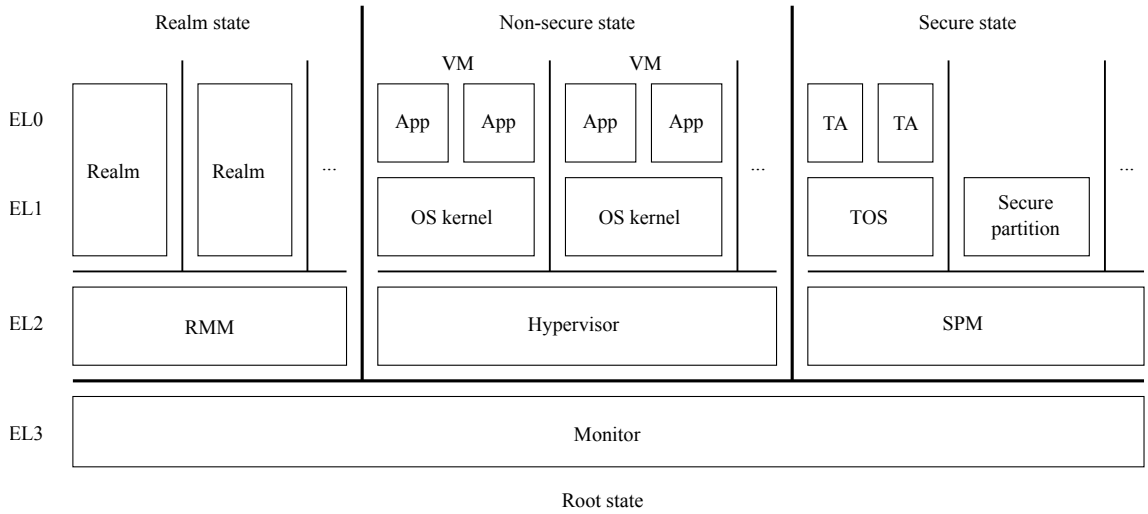

Figure D1-1 Security states

Where:

- RMM is Realm Management Monitor.
- SPM is Secure Partition Manager.
- VM is Virtual Machine.

D1.1.3 Effect of not implementing an Exception level

RKFGRW An exception cannot be taken to an unimplemented Exception level.

RHRDVX Any configurable instruction control that defines an exception to an unimplemented Exception level does not cause that exception to the unimplemented Exception level. The Effective value of the configurable exception control is the value that does not cause an exception to that Exception level.

RWPVKJ System calls made to an unimplemented Exception level from a lower Exception level are UNDEFINED.

RWPXJS The translation regime associated with an unimplemented Exception level is not implemented.

RWKQBG Any exception return that targets an unimplemented Exception level is an illegal return.

RGTCCW Every accessible register associated with an unimplemented Exception level is RES0 unless the register is associated with the Exception level only to provide the ability to transfer execution to a lower Exception level.

D1.1.3.1 Behavior when EL3 is not implemented

RBSNQL If EL3 is not implemented, SCR_EL3.NS has a fixed Effective value that is IMPLEMENTATION DEFINED.

IBWVTM An implementation can provide a configuration input that determines the Effective value of SCR_EL3.NS at reset.

RLBZCK If EL3 is not implemented, and the Effective value of SCR_EL3.NS is 0, all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Effective value of MDCR_EL3.EPMAD is 1.
- The Effective value of MDCR_EL3.EDAD is 1.
- The Effective value of MDCR_EL3.SPME is 1.
- The Effective value of MDCR_EL3.NSPB is `0b01`.
- The Effective value of MDCR_EL3.SPD32 is `0b11`.
- The Effective value of MDCR_EL3.STE is 1.

R_{ZTWJS} If EL3 is not implemented, the Effective value of MDCR_EL3.STE is the inverse of the Effective value of SCR_EL3.NS.

R_{RDYPY} If all of the following are true, the Effective value of SCR_EL3.EEL2 is 1:

- EL3 is not implemented.
- EL2 is implemented.
- The Effective value of SCR_EL3.NS is 0.

D1.1.3.2 Behavior when EL2 is not implemented

R_{PQDWN} If EL2 is not implemented, all of the following are true:

- Virtual interrupts are disabled.
- The Effective value of CNTHCTL_EL2[1:0] is `0b11`.
- The Effective value of HCR_EL2.E2H is 0.
- The Effective value of HCR_EL2.TGE is 0.
- The Effective value of MDCR_EL2.HPMN is the value of PMCR_EL0.N.

D1.1.3.3 Behavior when EL2 is not implemented and EL3 is implemented

R_{RJFFP} If EL2 is not implemented and EL3 is implemented, all of the following are true:

- Except for any of the following registers, every accessible register associated with EL2 is RES0:
  - If EL1 can use AArch32 then the following registers are not RES0:
    - DACR32_EL2.
    - DBGVCR32_EL2.
    - FPEXC32_EL2.
    - IFSR32_EL2.
  - For VPIDR_EL2:
    - Reads return the value of MIDR_EL1.
    - Writes to VPIDR_EL2 are ignored.
  - For VMPIDR_EL2:
    - Reads return the value of MPIDR_EL1.
    - Writes to VMPIDR_EL2 are ignored.
- The Effective value of HCR_EL2.RW is the value of SCR_EL3.RW.
- SCR_EL3.HCE is RES0.
- The following Address translation and TLB invalidation instructions are UNDEFINED:
  - AT S1E2R.
  - AT S1E2W.
  - TLBI VAE2, TLBI VAE2NXS.
  - TLBI VALE2, TLBI VALE2NXS.
  - TLBI VAE2IS, TLBI VAE2ISNXS.
  - TLBI VALE2IS, TLBI VALE2ISNXS.
  - TLBI VAE2OS, TLBI VAE2OSNXS.
  - TLBI VALE2OS, TLBI VALE2OSNXS.
  - TLBI ALLE2, TLBI ALLE2NXS.
  - TLBI ALLE2IS, TLBI ALLE2ISNXS.
  - TLBI ALLE2OS, TLBI ALLE2OSNXS.
  - TLBI RVAE2, TLBI RVAE2NXS.
  - TLBI RVALE2, TLBI RVALE2NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— TLBI RVAE2IS, TLBI RVAE2ISNXS.
— TLBI RVALE2IS, TLBI RVALE2ISNXS.
— TLBI RVAE2OS, TLBI RVAE2OSNXS.
— TLBI RVALE2OS, TLBI RVALE2OSNXS.

D1.1.3.4 Typical Exception level usage model

IWCCCB The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

Exception level	Usage
EL0	Applications.
EL1	OS kernel and associated functions that are typically described as privileged.
EL2	Hypervisor.
EL3	Monitor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.2 Registers for instruction processing and exception handling

IBCVRH In the Arm architecture, registers are split into two main categories:

- Registers that provide system control or status reporting.
- Registers that are used in instruction processing, for example to accumulate a result, and in handling exceptions.

For more information, see AArch64 System Register Descriptions.

D1.2.1 The general-purpose registers

RJBKTZ The general-purpose register bank comprises the 31 general-purpose registers, R0-R30.

IRKTLT The general-purpose register bank is associated with instructions in the base instruction set.

RYDKHJ The general-purpose registers can be accessed as either of the following:

- 31 64-bit registers, X0-X30.
- 31 32-bit registers, W0-W30.

IPVGXY When a Warm reset is asserted, the general-purpose registers are reset to an architecturally UNKNOWN value.
For more information, see Register size and Registers in AArch64 Execution state.

D1.2.2 The stack pointer registers

RBBRKH If in AArch64 state, each implemented Exception level has a dedicated stack pointer register. The stack pointer registers are:

- SP_EL0.
- SP_EL1.
- If EL2 is implemented, SP_EL2.
- If EL3 is implemented, SP_EL3.

IYPMHQ When a Warm reset is asserted, the stack pointers are reset to an architecturally UNKNOWN value.
For more information, see Special purpose registers and PC alignment checking.

D1.2.2.1 Stack pointer register selection

RCNNQX The PE uses the following stack pointers:

- If executing at EL0, then the PE uses the EL0 stack pointer, SP_EL0.
- If executing at EL1, EL2, or EL3, then the PE uses the stack pointer determined by PSTATE.SP:
  - If PSTATE.SP is 0, then the PE uses the EL0 stack pointer, SP_EL0.
  - If PSTATE.SP is 1, then the PE uses the stack pointer for the current Exception level.

ICSFMY When an exception is taken, the stack pointer for the target Exception level is selected.

IVYNZY The selected stack pointer can be indicated by a suffix to the Exception level:

- The t suffix, referring to thread, indicates use of the SP_EL0 stack pointer.
- The h suffix, referring to handler, indicates use of the SP_ELx stack pointer.

IQZDYD The following are the AArch64 stack pointer options:

Exception level (EL)	Stack pointer (SP) option
EL0	SP_EL0
EL1	SP_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception level (EL)	Stack pointer (SP) option
EL2	SP_EL2
EL3	SP_EL3

D1.2.3 The SIMD and floating-point registers

R_{JSCYH} The SIMD and floating-point register bank comprises the 32 quadword (128-bit) SIMD and floating-point registers, V0-V31.

I_{LMMMW} The SIMD and floating-point register bank is used for floating-point, vector, and other SIMD-related scalar operations.

R_{LFCJZ} The SIMD and floating-point registers can be accessed as any of the following:

- 32 quadword (128-bit) registers, Q0-Q31.
- 32 doubleword (64-bit) registers, D0-D31.
- 32 word (32-bit) registers, S0-S31.
- 32 halfword (16-bit) registers, H0-H31.
- 32 byte (8-bit) registers, B0-B31.
- 128-bit vectors of elements.
- 64-bit vectors of elements.

I_{KFYJJ} When a Warm reset is asserted, the SIMD and floating-point registers are reset to an architecturally UNKNOWN value.
For more information, see Registers in AArch64 Execution state.

D1.2.4 Saved Program Status Registers

R_{PJSSD} In AArch64 state, each implemented Exception level to which an exception can be taken has a Saved Program Status Register (SPSR). The following are the SPSRs:

- For exceptions taken to EL1 using AArch64, SPSR_EL1.
- For exceptions taken to EL2 using AArch64, SPSR_EL2.
- For exceptions taken to EL3 using AArch64, SPSR_EL3.

I_{DVBPS} When a Warm reset is asserted, the SPSRs are reset to an architecturally UNKNOWN value.
For more information, see Process state, PSTATE.

D1.2.5 Exception Link Registers

R_{JQBYN} Exception Link Registers hold preferred exception return addresses.

R_{VCLNF} In AArch64 state, each implemented Exception level to which an exception can be taken has an Exception Link Register (ELR). The following are the ELRs:

- For exceptions taken to EL1, ELR_EL1.
- For exceptions taken to EL2, ELR_EL2.
- For exceptions taken to EL3, ELR_EL3.

I_{SHJSC} When a Warm reset is asserted, the ELRs are reset to an architecturally UNKNOWN value.
For more information, see Preferred exception return address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3 System register locking

R_{DJDVH} All statements in this section require FEAT_SRMASK.

Note

HCR_EL2.E2H is RES1 when FEAT_SRMASK is implemented.

I_{DFJZG} Software can program the mask registers shown in the following table to mask individual control fields of the corresponding System registers, as follows:

- The EL1 mask registers mask EL1 control fields from EL1 MSR writes.
- The EL2 mask registers mask EL2 control fields from EL2 MSR writes.

For example, if TCRMASK_EL1 is programmed to mask a TCR_EL1 control field, then EL1 MSR writes to TCR_EL1 do not update that masked field.

Table D1-3 System registers and their corresponding mask registers

System register	Mask register
ACTLR_EL1	ACTLRMASK_EL1
CPACR_EL1	CPACRMASK_EL1
SCTLR_EL1	SCTLRMASK_EL1
SCTLR2_EL1	SCTLR2MASK_EL1
TCR_EL1	TCRMASK_EL1
TCR2_EL1	TCR2MASK_EL1
ACTLR_EL2	ACTLRMASK_EL2
CPTR_EL2	CPTRMASK_EL2
SCTLR_EL2	SCTLRMASK_EL2
SCTLR2_EL2	SCTLR2MASK_EL2
TCR_EL2	TCRMASK_EL2
TCR2_EL2	TCR2MASK_EL2

Using the mask registers makes it possible, for example, for EL1 to frequently change specific EL1 control fields without always trapping the EL1 MSR System register writes to EL2.

R_{FHCYV} EL1 is permitted to program an EL1 mask register when it is zero. Otherwise, the write to the mask register is UNDEFINED unless it is trapped.

EL2 is permitted to program an EL2 mask register when it is zero. Otherwise, the write to the mask register is UNDEFINED.

I_{VGNST} An attempt to program a nonzero mask register is only guaranteed to be UNDEFINED if there has been a Context synchronization event since the write that set it to the nonzero value and before the new attempt to program it. This is because the check of whether it is nonzero is an indirect read of it.

I_{QTNLT} For each EL1 mask register:

- HFGRTR2_EL2 contains a fine-grained trap control for EL1 reads.
- HFGWTR2_EL2 contains a fine-grained trap control for EL1 writes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3.1 EL1 System register alias names

I_{QBPYD} For each of the EL1 System registers in Table D1-3, FEAT_SRMASK provides an *EL1 alias name*, <System_register>ALIAS_EL1. These are listed in System and Special-purpose register aliasing.

- EL1 software that is aware of the presence of the EL1 alias names can use them when it is also aware that fine-grained traps are trapping EL1 writes to EL1 System registers.

For each of the EL2 System registers in Table D1-3, FEAT_SRMASK does not provide an EL2 alias name.

- A kernel executing at EL2 is expected to use the EL1 alias names.

I_{CWCFN} The EL1 mask registers mask the EL1 alias names in addition to masking the corresponding EL1 System registers. For example, if TCRMASK_EL1 is programmed to mask a TCR_EL1 control field, then EL1 `MSR` writes to TCRALIAS_EL1 do not update that masked field.

R_{CSSSN} `MSR` and `MRS` accesses to the EL1 alias names are direct accesses to the corresponding EL1 System registers. However, without explicit synchronization, the accesses might be reordered relative to `MSR` and `MRS` accesses to the EL1 System registers.

I_{VPCLQ} The accessibility from EL1 to each EL1 alias name is the same as for the corresponding EL1 System register, except for the opcode, and that there is a separate fine-grained trap control for each of reads and writes for the EL1 alias name:

- HFGRTR2_EL2 contains the fine-grained trap control for EL1 reads.
- HFGWTR2_EL2 contains the fine-grained trap control for EL1 writes.

D1.3.2 EL12 alias names for mask registers

I_{HFLBB} Because HCR_EL2.E2H is RES1 when FEAT_SRMASK is implemented, FEAT_SRMASK provides an EL12 alias name for each EL1 mask register in Table D1-3, for use by EL2 hypervisor software. See System and Special-purpose register aliasing.

D1.3.3 Nested virtualization

I_{GFHTJ} `MSR` writes that are transformed to stores by FEAT_NV2 are not masked by the EL1 mask register configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4 Exceptions

D1.4.1 Exception entry terminology

D1.4.1.1 Taken, taken from, and taken to

R_{VDFHD} When the PE responds to an exception, an exception is *taken*.

R_{FXGFX} The PE state immediately before taking the exception is the state the exception is *taken from*.

R_{WTMVN} The PE state immediately after taking the exception is the state the exception is *taken to*.

D1.4.1.2 Exception generating instructions

R_{QPPFN} *Exception generating instructions* intentionally produce a precise exception in the instruction stream immediately after the exception generating instruction. They are:

- `HVC`.
- `SMC`.
- `SVC`.

D1.4.1.3 Synchronous and asynchronous exceptions

R_{FQHGR} If all of the following apply, an exception is *synchronous*:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The preferred exception return address has an architecturally defined relationship with the instruction that caused the exception.
- The exception is precise.

R_{PCXDH} An exception is *asynchronous* if it is not synchronous.

R_{JPZPR} Asynchronous exceptions taken to AArch64 state are also known as interrupts.

D1.4.1.4 Definition of a precise exception and imprecise exception

R_{TNVSL} An exception is *precise* if on taking the exception, the PE state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. However, for an instruction executing immediately at the point in the instruction stream that the exception was taken from, the definition of precise also permits any of the following:

- For synchronous Data Abort and Watchpoint exceptions that are taken to AArch64 state generated by an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instruction can be UNKNOWN, if all of the following apply:
  - The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
  - The registers are not involved in the calculation of the memory address used by the instruction.
- For synchronous Data Abort and Watchpoint exceptions that are generated from load or store instructions executed in AArch64 state, all the following can occur:
  - If the instruction was a load to either the base address register or the offset register, that register is restored to the original value, and any other destination registers become UNKNOWN.
  - If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.
  - If the instruction was an atomic that attempted both a load and a store, and the load was to the base address register, a source register or a compare register, that register is restored to the original value. Otherwise, that destination register becomes UNKNOWN.
- For implementations that include synchronous exception generation for floating-point exceptions, when a floating-point exception is taken, it is permitted that the cumulative floating-point exception bits are not restored.
- For a precise exception that is taken from AArch64 state during an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instruction can be UNKNOWN, if all of the following apply:
  - The instruction is not `GCSSS2`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- — The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
- — The registers are not involved in the calculation of the memory address used by the instruction.

- For a synchronous exception or synchronous Debug state entry, generated by a Memory Copy and Memory Set instruction, and for an asynchronous exception or asynchronous Debug state entry part way through the execution of a Memory Copy and Memory Set instruction:
  - — The values of PSTATE.{N, Z, C, V} are UNKNOWN on exception entry. The values held in SPSR_ELx.{N, Z, C, V} on exception entry, and DSPSR_EL0.{N, Z, C, V} on Debug state entry, are the initial values for the instruction.
  - — All memory locations written by the pseudocode at the point at which the exception is taken are updated.
  - — If the memory locations that are not written by the pseudocode at the point at which the exception is taken are within the set of locations to be read by the instruction if it is restarted with the X[n], X[d], and X[s] register values presented as part of taking the exception, then those memory locations are not updated.
  - — For CPYF*T* and SET{G}*T* instructions, no memory locations that are not written by the pseudocode at the point at which the exception is taken are updated.
  - — For Memory Copy and Memory Set instructions other than CPYF*T* and SET{G}*T*, any memory location that does not itself generate a synchronous exception on a write becomes UNKNOWN if all the following apply:
    - — The memory location is within the set of locations to be written by the instruction if it is restarted with the X[n], X[d], and X[s] register values presented as part of taking the exception.
    - — The memory location is not within the set of locations to be read by the instruction on that restart.
  - — Only for a synchronous exception or synchronous Debug state entry, generated by a Memory Copy and Memory Set instruction:
    - — On taking the exception, the X[n], X[d], and, as appropriate, X[s] register values hold a self-consistent set of values that correspond to the first element that was not copied or set, such that return to the instruction enables resumption of the memory copy or memory set. That first element does not need to be the element with a watchpoint, but can be an earlier element, including the first element.
    - — If a memory location that generates a synchronous exception on a write is within the set of locations to be written by the instruction if it is restarted with the X[n], X[d], and X[s] register values presented as part of taking the exception, and if the memory location is not within the set of locations to be read by the instruction on that restart, then the memory location is unchanged.
    - — For a synchronous exception, ELR_ELx points to the instruction that generated the exception.
    - — For a synchronous Debug state entry, DLR_EL0 points to the instruction that generated the Debug state entry.
  - — Only for an asynchronous exception or asynchronous Debug state entry part way through the execution of a Memory Copy and Memory Set instruction:
    - — On taking the exception, the values of the registers that are updated by the instruction are presented as a self-consistent set of values that correspond to the first element that was not copied or set.

R_{ZNCXP} An exception is *imprecise* if it is not a precise exception.

R_{LSGJD} Except for SError exceptions, all exceptions taken to AArch64 state are precise. For each occurrence of an SError exception, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

I_{ZNNNY} An asynchronous Data Abort exception generated by a load that causes an SError exception to be taken at some point later in the instruction stream than the load that generated the exception, is usually imprecise. The SError exception is usually imprecise because the data returned from the load is UNKNOWN, and so can have corrupted the state that is presented at the time that the exception is taken.

D1.4.1.5 Preferred exception return address

I_{LQPWH} The *preferred exception return address* is an address that software might return to after handling an exception in order to resume execution.

I_{HBQRX} The preferred exception return address is determined by the type of the exception.

R_{DPLYN} For an exception taken to an Exception level, ELx, using AArch64, the Exception Link Register for that Exception level, ELR_ELx, is set to the preferred exception return address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{VBQMV} For asynchronous exceptions, the preferred exception return address is the address of the instruction following the instruction boundary at which the interrupt occurs.

R_{QYCWH} For synchronous exceptions other than exception generating instructions, the preferred exception return address is the address of the instruction that generates the exception.

R_{DKWPP} For an exception generating instruction that is executed, the preferred exception return address is the address of the instruction that follows the exception generating instruction.

R_{LBLBR} For an exception generating instruction that is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the exception generating instruction.

R_{XVMNH} When an exception is taken from an Exception level using AArch32 to an Exception level, ELx, using AArch64, ELR_ELx[63:32] are 0.

D1.4.1.6 Exception vectors

R_{FKKHH} When an exception is taken to an Exception level that is using AArch64, execution starts at the *exception vector*.

I_{QLHPV} The memory at the exception vector for an exception is expected to contain the handler code that corresponds to that exception category.

R_{BRCKV} The vector base address for an Exception level, ELx, is defined by the Vector Base Address Register, VBAR_ELx.

R_{PJKRC} Each exception category has an exception vector at a fixed offset from the vector base address.

I_{YCTHH} An exception taken to AArch64 is categorized for the purpose of assigning an exception vector based on the following information:

- Whether the exception is one of the following:
  - A synchronous exception, excluding synchronous External aborts.
  - A synchronous External abort.
  - An SError exception, including vSError and delegated SError exceptions.
  - An IRQ or vIRQ interrupt.
  - An FIQ or vFIQ interrupt.
- Information about the Exception level that the exception came from.
- Information about the SP in use.
- The state of the register file.

For information on synchronous exceptions, see Synchronous exception types. For information on asynchronous exceptions, see Asynchronous exception types.

R_{RYXCL} The following tables describe the offsets that are added to the vector base address to describe the exception vector:

Exception taken from	Offset for exception type		
	Synchronous exceptions, excluding External aborts	IRQ and vIRQ	FIQ and vFIQ
Current Exception level with SP_EL0.	0x000	0x080	0x100
Current Exception level with SP_ELx, x > 0.	0x200	0x280	0x300
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.	0x400	0x480	0x500
Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.	0x600	0x680	0x700

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception taken from	Offset for exception type		
	Synchronous External aborts, EASE == 0	Synchronous External aborts, EASE == 1	SError, vSError, and delegated SError
Current Exception level with SP_EL0.	`0x000`	`0x180`	`0x180`
Current Exception level with SP_ELx, x > 0.	`0x200`	`0x380`	`0x380`
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.	`0x400`	`0x580`	`0x580`
Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.	`0x600`	`0x780`	`0x780`

The applicable value of EASE is the following:

- For an exception taken to EL1, the Effective value of SCTLR2_EL1.EASE.
- For an exception taken to EL2, the Effective value of SCTLR2_EL2.EASE.
- For an exception taken to EL3, the Effective value of SCR_EL3.EASE.

I_{FSZTB} For the preceding tables, if EL3 is the target Exception level, the implemented Exception level immediately lower than EL3 is either:

- EL2, if EL2 is implemented and enabled for the Security state of the Exception level the exception was taken from.
- EL1, if EL2 is not implemented or not enabled for the Security state of the Exception level the exception was taken from.

D1.4.2 Exception entry

R_{DQXFW} When an exception is taken to an Exception level, ELx, that is using AArch64 state, all the following occur:

- Fields in SPSR_ELx are written according to their description in SPSR_ELx, based on the values in PSTATE immediately before the exception was taken.
- The preferred exception return address is written to ELR_ELx.
- The contents of PSTATE immediately after the exception is taken is as described in rule R_{WTXBY}.
- For synchronous exceptions and SError exceptions, exception syndrome information is written to ESR_ELx.
- Execution starts from the exception vector at the target Exception level.

R_{WTXBY} When an exception is taken to an Exception level, ELx, that is using AArch64 state, after recording the previous values in SPSR_ELx, the following PSTATE bits are set:

- PSTATE.EL is set to the target Exception level.
- All of PSTATE.{D, A, I, F, SP} are set to 1.
- PSTATE.{IL, nRW} are set to 0.
- PSTATE.SS is set according to the rules in AArch64 Self-hosted Debug.
- If FEAT_PAN is implemented, then for any of the following situations, PSTATE.PAN is set to 1:
  - The target Exception level is EL1 and SCTLR_EL1.SPAN is 0.
  - The target Exception level is EL2 using AArch64, the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and SCTLR_EL2.SPAN is 0.
- If FEAT_PAN is implemented, the target Exception level is EL2 using AArch64, the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}, and SCTLR_EL2.SPAN is 0, then it is IMPLEMENTATION SPECIFIC whether PSTATE.PAN is set to 1.
- If FEAT_UAO is implemented, then PSTATE.UAO is set to 0.
- If FEAT_BTI is implemented, then PSTATE.BTYPE is set to `0b00`.
- If FEAT_MTE is implemented, then PSTATE.TCO is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_SSBS is implemented, then PSTATE.SSBS is set to the value of SCTLR_ELx.DSSBS.
- If FEAT_NMI is implemented, then PSTATE.ALLINT is set to the inverse value of SCTLR_ELx.SPINTMASK.
- If FEAT_EBEP, FEAT_SPE_EXC or FEAT_TRBE_EXC is implemented, then PSTATE.PM is set to 1.
- If FEAT_GCS is implemented, PSTATE.EXLOCK is updated based on the following situations:
  — On taking an exception to the same Exception level, PSTATE.EXLOCK is set to the Effective value of GCSCR_ELx.EXLOCKEN for the current Exception level.
  — On taking an exception to a higher Exception level, PSTATE.EXLOCK is set to 0.
- If FEAT_PAuth_LR is implemented, then PSTATE.PACM is set to 0.

R_{XKSNJ} If FEAT_UINJ is implemented, then when an exception is taken to an Exception level, ELx, that is using AArch64 state, PSTATE.UINJ is set to 0, and the resulting value stored in SPSR_ELx is 0.

I_{RFWPH} When an exception is taken to an Exception level that is using AArch64 state, the following PSTATE fields are unchanged:

- If FEAT_DIT is implemented, PSTATE.DIT.
- If FEAT_SME is implemented, PSTATE.{SM, ZA}.

I_{HLPGV} PSTATE.PACM is set to 0 when executing an `SVC`, `HVC`, or `SMC` instruction that is not trapped, disabled, or UNDEFINED, and the resulting value stored in SPSR_ELx.PACM is 0.

R_{SRXVW} If the Effective value of SCTLR_ELx.IESB is 1 at the target Exception level, an exception taken to AArch64 state is an error synchronization event.

R_{BBSRF} If SCTLR_ELx.EIS is 1, exception entry is a Context synchronization event. If SCTLR_ELx.EIS is 0, exception entry is not a Context synchronization event, but the indirect writes to ESR_ELx, FAR_ELx, SPSR_ELx, ELR_ELx, and HPFAR_EL2 due to exception entry are synchronized so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.

I_{JFWCQ} For the purposes of the memory model, some exception entries are Instruction Fetch Barrier effects, regardless of the value of SCTLR_ELx.EIS. See Basic definitions for the list of exception entries.

I_{FZTLH} The ordering of error synchronization and context synchronization on exception entry is described in the pseudocode.

I_{WPBHV} Memory copy and memory set operations are guaranteed to function correctly only if the prologue, main, and epilogue Memory Copy and Memory Set instructions are executed in succession and are placed consecutively in memory. Failure to execute the three instructions in succession can result in the instructions causing exceptions. Software exception handlers are expected to require that the three instructions are placed consecutively in memory. If the three instructions are not placed consecutively in memory, exception returns to the wrong instructions might occur.

I_{LBHFM} There are additional behaviors for synchronous External aborts and SError exceptions. See also Taking error exceptions.

D1.4.2.1 Synchronous exception entry

R_{PMQBM} For any of the following synchronous exceptions taken from an Exception level using AArch64, if FEAT_BTI is implemented, then PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then PSTATE.BTYPE is set to 0:

- Software Step exception.
- PC Alignment Fault exception.
- Instruction Abort exception.
- Breakpoint exception.
- Address Matching Vector Catch exception.
- Illegal Execution state exception.
- Software Breakpoint exception.
- Branch Target exception.

When taking any other synchronous exception from an Exception level using AArch64, it is CONSTRAINED UNPREDICTABLE whether:

- SPSR_ELx.BTYPE is set to the value of PSTATE.BTYPE.
- SPSR_ELx.BTYPE is set to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSTATE.BTYPE is then set to 0.

I_{KVRPF} For synchronous exceptions that result in the exception handler emulating the instruction and returning to the following instruction, the handler typically completes by ensuring the state of the PE on returning from the exception is consistent with the emulated instruction having been executed. This usually includes ensuring the value of SPSR_ELx.BTYPE is consistent with the instruction being emulated having executed.

For `SVC`, `HVC`, and `SMC` instructions, when these instructions execute without being trapped, the value of SPSR_ELx already points to the following instruction. However SPSR_ELx.BTYPE is not guaranteed to be 0 and so might not be consistent with the `SVC`, `HVC`, or `SMC` instruction having executed. This might subsequently cause a Branch Target exception at the instruction after the `SVC`, `HVC`, or `SMC` instruction. This is unlikely to be encountered in real code since it requires the `SVC`, `HVC`, or `SMC` to be executed in a non-Guarded page with the instruction after the `SVC`, `HVC`, or `SMC` executed in a Guarded page.

R_{ZTSSH} For any of the following synchronous exception types, when an exception is taken to an Exception level, ELx, a *virtual address* (VA) that characterizes the exception is captured in FAR_ELx:

- An Instruction Abort exception.
- A Data Abort exception.
- A PC alignment fault exception.
- A Watchpoint exception.

For GPC exceptions, a VA that characterizes the exception is captured in FAR_EL3.

For synchronous External aborts that are not caused by translation table walks, it is CONSTRAINED UNPREDICTABLE whether the FAR_ELx contains a VA that characterizes the exception. The ESR_ELx.FnV bit in the ISS encoding for the External abort indicates the validity of the VA in FAR_ELx:

- If ESR_ELx.FnV in the ISS encoding for the External abort is 1, the VA in FAR_ELx is UNKNOWN.
- If ESR_ELx.FnV in the ISS encoding for the External abort is 0, the VA in FAR_ELx is valid.

For all other exceptions taken to ELx, FAR_ELx is UNKNOWN.

R_{JXPNL} When an SME load or store instruction, or an SVE contiguous vector load or store instruction, causes a Data Abort exception that sets ESR_ELx.ISV to 0:

- If the value written to FAR_ELx might not be the faulting VA, the PE sets ESR_ELx.FnP to 1 and sets FAR_ELx to any address within the naturally-aligned fault granule that contains the faulting VA. The naturally-aligned fault granule is as defined in the FAR_ELx register descriptions.
- Otherwise, the PE sets ESR_ELx.FnP to 0.

R_{FKLWR} For Instruction Abort or Data Abort exceptions caused by any of the following faults, when an exception is taken to EL2, an *intermediate physical address* (IPA) that characterizes the exception is captured in HPFAR_EL2:

- A Translation fault, Access Flag fault, or Address Size fault on a stage 2 translation not on a stage 1 translation table walk.
- A Translation fault, Access flag fault, Address Size fault, or Permission fault on stage 2 translation of an address accessed in a stage 1 translation table walk.
- A Granule Protection Fault (GPF) on an access for a stage 2 translation table.

For GPC exceptions due to a fault on an access for a stage 2 translation table walk, an IPA that characterizes the exception is captured in HPFAR_EL2.

For all other exceptions taken to EL2 using AArch64, HPFAR_EL2 is UNKNOWN.

R_{BFJJV} For GPC exceptions, a PA and physical address space that characterize the exception and a PA are captured in MFAR_EL3.

R_{TGQRC} For Instruction Abort or Data Abort exceptions caused by an External abort, when FEAT_PFAR is implemented:

- If all of the following apply, then ESR_ELx.PFV is set to 0:
  - The exception is taken to EL1.
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCR_EL2.VM is 1.
- Otherwise, ESR_ELx.PFV is set to an IMPLEMENTATION DEFINED value of 0 or 1.

For all other Instruction Abort and Data Abort exceptions, ESR_ELx.PFV is set to 0.

On taking an Instruction Abort or Data Abort exception:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If ESR_ELx.PFV is set to 1 by the PE, then a physical address space that characterizes the exception and a PA are captured in PFAR_ELx or MFAR_EL3 as applicable. The fault granule size is defined by D_{LVGRB}.
- If ESR_ELx.PFV is set to 0 by the PE, then the applicable PFAR_ELx or MFAR_EL3 register is set to an UNKNOWN value.

Note

PFAR_ELx never records the Intermediate Physical Address (IPA). PFAR_ELx might reveal a faulting physical addresses to a guest operating system if stage 2 translation is not being used and some other method is used to hide physical addresses from the guest (such as shadow page tables).

R_{DHLHF} For all exceptions other than Instruction Abort, Data Abort, and GPC exceptions, the applicable PFAR_ELx or MFAR_EL3 register is set to an UNKNOWN value.

R_{TGRZL} If an Instruction Abort or Breakpoint exception is generated part way through execution of a Memory Copy and Memory Set instruction, presentation of the exception state is as described in R_{TNVSL} for a synchronous exception or synchronous Debug state entry from a watchpoint, generated by a Memory Copy and Memory Set instruction.

R_{ZMSLS} For a CPY* Memory Copy instruction, if both the read and write generate the Data Abort or Watchpoint exception, it is CONSTRAINED UNPREDICTABLE whether the exception on the read or the exception on the write is presented. ESR_ELx.ISS.WnR is set to be consistent with whether the exception was on the read or the write.

D1.4.2.1.1 SVE MOVPRFX exception entry behavior

R_{RWVTR} When a `MOVPRFX` instruction pairs legally with another instruction and the execution of the pair generates a synchronous exception:

- If the generated exception is a Breakpoint Instruction exception from a prefixed `BRK` instruction then `MOVPRFX` updates the architectural state and ELR_ELx stores the address of `BRK` instruction.
- Otherwise, the return address that is stored in ELR_ELx is one of the following:
  - When the `MOVPRFX` instruction did not cause a change to the architectural state, the address of the `MOVPRFX` instruction is stored.
  - When the `MOVPRFX` instruction caused a change to the architectural state, the address of the prefixed instruction is stored.

R_{XRWVD} When a `MOVPRFX` instruction pairs legally with another instruction and the execution of the pair causes synchronous entry to Debug state:

- If the Debug state entry is due to a Halt Instruction debug event from a prefixed `HLT` instruction then `MOVPRFX` updates the architectural state and DLR_EL0 stores the address of `HLT` instruction.
- Otherwise, the return address that is stored in DLR_EL0 is one of the following:
  - When the `MOVPRFX` instruction did not cause a change to the architectural state, the address of the `MOVPRFX` instruction is stored.
  - When the `MOVPRFX` instruction caused a change to the architectural state, the address of the prefixed instruction is stored.

R_{TPRKM} When a `MOVPRFX` instruction pairs illegally with another instruction and execution of the pair generates a synchronous exception, the return address recorded in ELR_ELx is a CONSTRAINED UNPREDICTABLE choice one of the following:

- The address of the `MOVPRFX` instruction.
- The address of the prefixed instruction.

R_{JVNGC} When a `MOVPRFX` instruction pairs illegally with another instruction and execution of the pair causes entry to Debug state, the return address recorded in DLR_EL0 is a CONSTRAINED UNPREDICTABLE choice one of the following:

- The address of the `MOVPRFX` instruction.
- The address of the prefixed instruction.

R_{CRRPM} When a prefixed instruction generates an Instruction Abort due to an MMU fault or synchronous External abort and the `MOVPRFX` does not generate an Instruction Abort, then the address of the prefixed instruction is recorded in the appropriate FAR_ELx or HPFAR_EL2 register and the address of the `MOVPRFX` instruction is recorded in the appropriate ELR_ELx register.

R_{ZJYDX} When a prefixed instruction generates an Instruction Abort due to an MMU fault or synchronous External abort and the `MOVPRFX` also generates an Instruction Abort, then the address of the `MOVPRFX` instruction is recorded in the appropriate FAR_ELx or HPFAR_EL2 register and the appropriate ELR_ELx register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.2.2 Asynchronous exception entry

R_{BWRCL} If FEAT_BTI is implemented, then when an asynchronous exception is taken from an Exception level using AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then PSTATE.BTYPE is set to 0.

R_{JBKRW} If any of the following apply, when a physical SError exception is taken to AArch64 state, the pending state of the physical SError is cleared:

- FEAT_DoubleFault is implemented.
- If FEAT_RAS is implemented, and on taking the SError exception, the syndrome recorded in ESR_ELx indicates an SError other than IMPLEMENTATION DEFINED or uncategorized SError exception syndrome. See Taking error exceptions.

Otherwise, it is IMPLEMENTATION DEFINED whether the pending state of the physical SError is cleared. This IMPLEMENTATION DEFINED behavior might vary according to the type of the SError exception.

R_{SYZRH} When a virtual SError exception is taken to AArch64 state, HCR_EL2.VSE is set to 0.

R_{QXQQX} When a delegated SError exception is taken, SCR_EL3.DSE is set to 0.

R_{NPSDK} When FEAT_PFAR is implemented, on taking an SError exception to an Exception level using AArch64:

- If all of the following apply, ESR_ELx.PFV is set to 0:
  - The exception is taken to EL1.
  - EL2 is implemented and enabled in the current security state.
  - The Effective value of HCR_EL2.VM is 1.
- Otherwise, ESR_ELx.PFV is set to an IMPLEMENTATION DEFINED value of 0 or 1.

On taking an SError exception, if ESR_ELx.PFV is set to 1 by the PE then:

- An address within the same naturally-aligned fault granule as the faulting physical address is written to PFAR_ELx.PA or MFAR_EL3.PA as applicable. The fault granule size is defined by D_{LVGRB}.
- The faulting physical address space is written to PFAR_ELx.{NSE, NS} or MFAR_EL3.{NSE, NS} as applicable.

Note

PFAR_ELx never records the Intermediate Physical Address (IPA). PFAR_ELx might reveal a faulting physical addresses to a guest operating system if stage 2 translation is not being used and some other method is used to hide physical addresses from the guest (such as shadow page tables).

R_{YJTBL} On taking an SError exception, the following registers are UNKNOWN based on the ESR_ELx.PFV value:

- If ESR_EL1.PFV is set to 0, PFAR_EL1 is UNKNOWN.
- If ESR_EL2.PFV is set to 0, PFAR_EL2 is UNKNOWN.
- If ESR_EL3.PFV is set to 0, MFAR_EL3 is UNKNOWN.

R_{GZVBJ} An asynchronous exception or asynchronous Debug state entry part way through the execution of a Memory Copy and Memory Set instruction is permissible. For an asynchronous exception, ELR_ELx points to the instruction that the exception was taken on. For an asynchronous Debug state entry, DLR_EL0 points to the instruction that the Debug state entry was taken on.

R_{MNGMW} For an imprecise asynchronous exception part way through the execution of a Memory Copy and Memory Set instruction, the following are UNKNOWN:

- The state of the X[n], X[d], and X[s] registers.
- The state of memory that was being written to by the instruction.

D1.4.3 Exception return terminology

D1.4.3.1 Return, return from, return to

R_{MMSFW} An exception return is caused by the execution of an exception return instruction.

R_{JDJHH} The PE state immediately before an exception return instruction is executed is the state the exception *returns from*.

R_{HNVMN} The PE state immediately after the execution of an exception return instruction is the state the exception *returns to*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.4 Exception return

R_{SKNJF} In AArch64 state, the Exception return instructions are ERET, ERETAA, and ERETAB.

R_{PKJFB} The Exception return instructions are UNDEFINED in EL0.

R_{SVBYH} An exception return is either legal or illegal.

D1.4.4.1 Legal exception returns from AArch64 state

R_{BWCFK} On a legal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, and the Exception return instruction does not generate an exception, the PE inserts an error synchronization event before the Exception return instruction.
- Fields in PSTATE that have a corresponding field in SPSR_ELx are written according to their description in SPSR_ELx.
- The PC is set to the value in ELR_ELx.
- If returning to an Exception level using AArch32 state, all the following apply:
  - If SPSR_ELx.T is 0, ELR_ELx[1:0] are treated as being 0 for setting the PC.
  - If SPSR_ELx.T is 1, ELR_ELx[0] is treated as being 0 for setting the PC.
- The contents of PSTATE are set to the values held in SPSR_ELx.
- If FEAT_PAuth_LR is implemented and any of the following apply, PSTATE.PACM is set to 0:
  - The trivial implementation of PSTATE.PACM is used.
  - The effects of PACM instructions are disabled at the target Exception level.
- If the PSTATE.IL bit copied from SPSR_ELx is 1 and the target Exception level for the return is using AArch32 state, the PSTATE.{IT, T} bits are determined by an IMPLEMENTATION DEFINED choice of one of the following:
  - Set to 0.
  - For returns initiated by an Exception return instruction or DRPS instruction, copied from SPSR_ELx.
  - For debug exits, copied from DSPSR_EL0.

  The IMPLEMENTATION DEFINED choice might vary dynamically within the implementation. Software must regard the value as being an UNKNOWN choice between the two values.
- The Event Register for the PE executing the Exception return instruction is set.
- The local Exclusives monitor for the PE executing the Exception return instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.
- After the PC is set to the value held in ELR_ELx and the contents of PSTATE are set to the values held in SPSR_ELx, ELR_ELx and SPSR_ELx become UNKNOWN.
- If the exception return is from EL2, to EL0 in AArch64 state when HCR_EL2.TGE is 0, ELR_EL1 and SPSR_EL1 become UNKNOWN.
- If the Effective value of SCTLR_ELx.EOS is 1, the exception return is a Context synchronization event.
- If the Effective value of SCTLR_ELx.EOS is 0, the exception return is not a Context synchronization event.

D1.4.4.2 Illegal exception returns from AArch64 state

R_{TYTWB} If in AArch64 state, any of the following situations can cause an illegal exception return:

- A return is made to an Exception level higher than the current Exception level.
- A return is made to an Exception level that is not implemented.
- If FEAT_RME is implemented, then if SCR_EL3.{NSE, NS} is {1, 0}, an exception return from EL3 to a lower Exception level.
- If all of the following are true, and a return is made to EL1:
  - EL2 is implemented and enabled in the current Security state
  - HCR_EL2.TGE is 1.
- If all of the following are true, and a return is made to EL2:
  - EL3 is implemented.
  - SCR_EL3.NS is 0.
  - FEAT_SEL2 is not implemented.
- If all of the following are true, and a return is made to EL2:
  - EL3 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SCR_EL3.NS is 0.
  - FEAT_SEL2 is implemented.
  - SCR_EL3.EEL2 is 0.
- A return where the saved PSTATE.M[4] is 0 and at least one of the following is true:
  - M[1] is 1.
  - M[3:0] are `0b0001`.
  - The Exception level being returned to is using AArch32 state, as determined by SCR_EL3.RW, HCR_EL2.RW, or as configured from reset.
- A return where the saved PSTATE.M[4] bit is 1, indicating a return to AArch32 state, and at least one of the following are true:
  - AArch32 state is not supported at any Exception level.
  - The M field value is not a valid AArch32 state PE mode.
  - The Exception level being returned to is using AArch64 state, as determined by SCR_EL3.RW, HCR_EL2.RW, or as configured from reset.
- If the Effective value of GCSCR_ELx.EXLOCKEN is 1 and PSTATE.EXLOCK is 0, the execution of an exception return instruction to return to the current Exception level ELx.

R_{VWJHB} On an illegal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, the PE inserts an error synchronization event before the Exception return instruction.
- The PC is set to the value in ELR_ELx.
  If the saved PSTATE.M[4] bit is 1, for illegal exception returns to AArch32 state, all of the following are true:
  - The PC bits[31:2] are set to the value in ELR_ELx.
  - The PC bits[63:32, 1:0] are UNKNOWN.
- PSTATE.IL is set to 1.
- PSTATE.{EL, nRW, SP} are unchanged.
- PSTATE.{N, Z, C, V, D, A, I, F} are set to the associated values in SPSR_ELx.
- PSTATE.SS is handled as if the return is a legal exception return.
- If PSTATE.nRW is 1, indicating a return to AArch32 state, then the following PSTATE bits are also set:
  - PSTATE.{Q, IT, GE, E} are set to the associated values in SPSR_ELx.
  - It is CONSTRAINED UNPREDICTABLE whether PSTATE.T is 0 or set to the contents of SPSR_ELx.
- If FEAT_PAN is implemented, then PSTATE.PAN is set to the associated value in SPSR_ELx.
- All of the following are UNKNOWN:
  - If FEAT_UAO is implemented, then PSTATE.UAO.
  - If FEAT_DIT is implemented, then PSTATE.DIT.
  - If FEAT_MTE is implemented, then PSTATE.TCO.
  - If FEAT_SSBS is implemented, then PSTATE.SSBS.
  - If FEAT_BTI is implemented, then PSTATE.BTYPE.
  - If FEAT_PAuth_LR is implemented, then PSTATE.PACM.
- If FEAT_NMI is implemented, then PSTATE.ALLINT is set to the associated value in SPSR_ELx.
- If FEAT_EBEP, FEAT_SPE_EXC or FEAT_TRBE_EXC is implemented, then PSTATE.PM is set to the associated value in SPSR_ELx.
- If FEAT_GCS is implemented, then PSTATE.EXLOCK is unchanged.
- If FEAT_UINJ is implemented, then PSTATE.UINJ is set to 0.
- The Event Register for the PE executing the Exception return instruction is set.
- The local Exclusives monitor for the PE executing the Exception return instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.
- After the PC has been set to the value held in ELR_ELx and the contents of PSTATE have been set to the value held in SPSR_ELx, the ELR_ELx and SPSR_ELx become UNKNOWN.
- If the Effective value of SCTLR_ELx.EOS is 1, the exception return is a Context synchronization event.
- If the Effective value of SCTLR_ELx.EOS is 0, the exception return is not a Context synchronization event.

D1.4.5 Synchronous exception types

R_{BZGXV} All of the following are synchronous exceptions:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Any exception generated by attempting to execute an instruction that is UNDEFINED, including:
  - Attempts to execute instructions at an inappropriate Exception level.
  - Attempts to execute instructions when they are disabled.
  - Attempts to execute instruction bit patterns that are not allocated.
  - Any attempt to execute any instruction when PSTATE.UINJ is 1. See Undefined Instruction exceptions injected by PSTATE.UINJ.
- Any exception caused by attempts to execute an instruction when the value of PSTATE.IL is 1. These are called Illegal Execution State exceptions.
- Any exception caused by the use of a misaligned SP. These are called SP Alignment Fault exceptions.
- Any exception caused by attempting to execute an instruction with a misaligned PC. These are called PC Alignment Fault exceptions.
- If FEAT_BTI is implemented, execution is inside a guarded memory region, and PSTATE.BTYPE does not equal 0, then any exception caused by executing an instruction that is not compatible with the current value of PSTATE.BTYPE. These are called Branch Target exceptions.
- Any exception caused by a pointer authentication instruction authentication failure. These are called PAC Fail exceptions.
- Any exception caused by the exception generating instructions SVC, HVC, or SMC. These are respectively called Supervisor Call (SVC) exceptions, Hypervisor Call (HVC) exceptions, or Secure Monitor Call (SMC) exceptions.
- Traps on attempts to execute instructions that the System registers define as instructions that are trapped to a higher Exception level. These are called Trap exceptions.
- Any exception caused by Instruction Aborts that were generated by the memory address translation system that was associated with attempts to execute instructions from areas of memory that generate faults. These are called Instruction Abort exceptions.
- Any exception caused by Data Aborts that were generated by the memory address translation system that are associated with attempts to read or write memory that generate faults. These are called Data Abort exceptions.
- Any exception caused by Data Aborts because of a misaligned address. These are called Data Abort exceptions.
- If FEAT_MTE2 is implemented, any exception caused by a Data Abort as a result of a Tag Check Fault. These are called Data Abort exceptions.
- If FEAT_RME is implemented, any exception that results from a *Granule Protection Check* (GPC) fault. Depending on the cause of the exception, the GPC fault type, and the value of the routing control SCR_EL3.GPF, these are called:
  - GPC exceptions.
  - Data Abort exceptions.
  - Instruction Abort exceptions.
- Memory Copy exceptions and Memory Set exceptions. For more information, see Memory Copy and Memory Set exceptions.
- Guarded Control Stack exceptions. For more information, see Guarded Control Stack exceptions.
- All of the debug exceptions:
  - Breakpoint Instruction exceptions.
  - Breakpoint exceptions.
  - Watchpoint exceptions.
  - Vector Catch exceptions.
  - Software Step exceptions.
- In an implementation that supports the trapping of floating-point exceptions, any exception caused by trapped floating-point exceptions. These are called Trapped Floating-point exceptions.
- An IMPLEMENTATION DEFINED set of exceptions caused by External aborts.

I_{XRFSY} The architecture permits, but does not require, synchronous or asynchronous handling of External aborts. However, some situations specified by the architecture must be handled synchronously. For more information, see Taking error exceptions.

D1.4.5.1 Taking synchronous exceptions from EL0

R_{DPLSC} If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 1, when any of the following exceptions are taken from EL0, they are taken to EL2, unless routed to a higher Exception level. If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 0, when any of the following exceptions are taken from EL0, they are taken to EL1, unless routed to a higher Exception level:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Stage 1 Data Abort.
- Stage 1 Instruction Abort.
- PC Alignment fault.
- SP Alignment fault.
- Tag Check fault.
- Branch Target exception.
- Illegal Execution State exception.
- Trapped Floating-point exception.
- Supervisor Call.
- Undefined Instruction exception.
- PAC Fail exception.
- WFE trap.
- WFI trap.
- Advanced SIMD and floating-point Access trap.
- SVE Access trap.
- SME Access traps.
- SME Illegal Instruction exceptions.
- Synchronous External aborts.
- Memory Copy and Memory Set exceptions.
- PMU exceptions.
- Guarded Control Stack exceptions.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when any of the following debug exceptions are taken from EL0, they are taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, when any of the following debug exceptions are taken from EL0, they are taken to EL1:

- Breakpoint exception.
- Software Breakpoint exception.
- Software Step exception.
- Watchpoint exception.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when a Vector Catch exception is taken from EL0, it is taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, a Vector Catch exception cannot be taken.

R_{JNBTN} For an exception that is taken from EL0 to EL2 because the value of HCR_EL2.TGE is 1:

- If the exception would have been reported in ESR_EL1 using any EC value other than `0x07`, then the EC value and corresponding ISS encoding that would have been used to report the exception in ESR_EL1 are used to report the exception in ESR_EL2.
- If the exception would have been reported in ESR_EL1 using the EC value `0x07`, then the EC value `0x00` and ISS encoding value `0x00` are reported in ESR_EL2.

D1.4.5.2 Exception levels for taking a synchronous External abort

R_{QDLHN} If all of the following are true, then a synchronous External abort is taken to EL3 using AArch64:

- EL3 is implemented and using AArch64.
- Any of the following are true:
  - SCR_EL3.EA is 1 and the exception is taken from EL2, EL1, or EL0.
  - The exception is taken from EL3.

Otherwise, if all of the following are true, then a synchronous External abort exception is taken to EL2:

- EL2 is implemented and enabled in the current Security state.
- The exception is taken from EL1 or EL0.
- Any of the following are true:
  - The Effective value of HCR_EL2.TEA or HCR2.TEA is 1.
  - The exception was generated by an External abort on a stage 2 translation table walk.
  - The exception was generated by an External abort on a System register access converted to a memory access by FEAT_NV2. This can only occur at EL1.
  - The Effective value of HCR_EL2.TGE or HCR.TGE is 1. This can only occur at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise, if FEAT_DoubleFault2 is implemented and all of the following are true, then a synchronous External abort exception is taken to EL2:

- EL2 is implemented, enabled in the current Security state, and using AArch64.
- The Effective value of HCRX_EL2.TMEA is 1.
- PSTATE.A is 1.
- The exception is taken from EL1.

Otherwise, if FEAT_DoubleFault2 is implemented and all of the following are true, then a synchronous External abort exception is taken to EL3:

- EL3 is implemented and using AArch64.
- The Effective value of SCR_EL3.TMEA is 1.
- PSTATE.A is 1.
- The exception is taken from EL1 or EL2.

Otherwise, all of the following apply:

- A synchronous External abort exception taken from EL2 is taken to EL2.
- When EL1 is using AArch64, synchronous External abort exception taken from EL1 or EL0 is taken to EL1 using AArch64.

I_{MBJFW} The terms used in the tables in I_{NDBHM} have the following meanings:

SCR_EL3 The Effective value of a field in SCR_EL3.

HCR If EL2 is using AArch32, this is the Effective value of a field in HCR or HCR2. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.

HCRX_EL2 The Effective value of a field in HCRX_EL2.

PSTATE The Effective value of a field in PSTATE.

EA The Effective value of SCR_EL3.EA. If EL3 is not implemented, the Effective value of SCR_EL3.EA is 0.

TMEA The Effective value of SCR_EL3.TMEA or HCRX_EL2.TMEA.

If FEAT_DoubleFault2 is not implemented, the Effective values of SCR_EL3.TMEA and HCRX_EL2.TMEA are 0.

If EL2 is not implemented or EL2 is disabled in the current Security state, the Effective value of SCR_EL3.HXEn is 0, or EL2 is using AArch32, then the Effective value of HCRX_EL2.TMEA is 0.

TGE The Effective value of HCR.TGE. If EL2 is not implemented, or if EL2 is disabled in the current Security state, then the Effective value of HCR.TGE is 0.

A The Effective value of PSTATE.A.

EL1 The exception is taken to EL1.
If EL1 is using AArch32, then the exception is taken to the AArch32 Abort mode.

EL2 The exception is taken to EL2.
If EL2 is using AArch32, then the exception is taken to the AArch32 Hyp mode.

EL3 The exception is taken to EL3 using AArch64.

n/a Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

I_{NDBHM} The following table summarizes the destination Exception level for a synchronous External abort exception taken from ELx, and all of the following applies:

- SCR_EL3.{NS, EEL2} are not shown. I_{MBJFW} describes Effective values involving Security states.
- The error was generated by a stage 1 fault on an access that is not a System register access. See R_{QDLHN} for details of other cases.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCR_EL3		HCR		HCRX_EL2	PSTATE	Exception taken from			
EA	TMEA	TGE	TEA	TMEA	A	EL0	EL1	EL2	EL3
0b0	0b0	0b0	0b0	0b0	X	EL1	EL1	EL2	EL3
0b0	0b0	0b0	0b0	0b1	0b0	EL1	EL1	EL2	EL3
0b0	0b0	0b0	0b0	0b1	0b1	EL1	EL2	EL2	EL3
0b0	0b0	0b0	0b1	X	X	EL2	EL2	EL2	EL3
0b0	0b0	0b1	X	X	X	EL2	n/a	EL2	EL3
0b0	0b1	0b0	0b0	0b0	0b1	EL1	EL3	EL3	EL3
0b0	0b1	0b0	0b0	X	0b0	EL1	EL1	EL2	EL3
0b0	0b1	0b0	0b0	0b1	0b1	EL1	EL2	EL3	EL3
0b0	0b1	0b0	0b1	X	0b0	EL2	EL2	EL2	EL3
0b0	0b1	0b0	0b1	X	0b1	EL2	EL2	EL3	EL3
0b0	0b1	0b1	X	X	0b0	EL2	n/a	EL2	EL3
0b0	0b1	0b1	X	X	0b1	EL2	n/a	EL3	EL3
0b1	X	0b0	X	X	X	EL3	EL3	EL3	EL3
0b1	X	0b1	X	X	X	EL3	n/a	EL3	EL3

D1.4.5.3 Granule Protection Check faults

I_{YRNGX} If FEAT_RME is implemented, then when GPCCR_EL3.GPC is 1, *Granule Protection Checks* (GPCs) on accesses to Physical Address (PA) space are enabled and might result in *GPC faults*.

R_{PYTGX} GPC faults are reported as synchronous exceptions:

GPC fault	Synchronous exception type
Granule Protection Fault (GPF) at EL3	Instruction Abort exception or Data Abort exception
GPF at EL2, EL1, or EL0 when SCR_EL3.GPF is 0	Instruction Abort exception or Data Abort exception
GPF at EL2, EL1, or EL0 when SCR_EL3.GPF is 1	GPC exception
Granule Protection Table (GPT) walk faults	GPC exception
GPT address size faults	GPC exception
Synchronous External abort on GPT fetch	GPC exception

R_{FXMGJ} GPC exceptions are taken to EL3.
Instruction Abort exceptions and Data Abort exceptions resulting from GPFs at:

- EL3 are taken to EL3.
- EL2 are taken to EL2.
- EL1 and EL0 are taken to:
  - EL2 when the GPF is on an access for a stage 2 translation table, including for a hardware update of stage 2 tables.
  - Otherwise:
    - EL1 when HCR_EL2.{TGE, GPF} is {0, 0}.
    - EL2 when HCR_EL2.{TGE, GPF} is not {0, 0}.

I_{XWVCY} For GPFs at EL2, EL1, or EL0 when SCR_EL3.GPF is 1, EL3 firmware can choose to delegate the resulting GPC exception to a lower Exception level, as an Instruction Abort exception or Data Abort exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{JXSRX} When the PE is in Debug state and EDSCR.SDD is 1, SCR_EL3.GPF is treated as 0, and the following GPC faults are treated as a GPF for the purposes of causing an exception:

- GPT walk faults.
- GPT address size faults.
- Synchronous External abort on GPT fetch.

I_{ZTKNY} GPC exceptions due to a synchronous External abort on GPT fetch are subject to SCR_EL3.EASE.

I_{YJLPJ} All GPC faults are reported with a priority consistent with the GPC being performed on any access to PA space. That is, for each existing synchronous External abort, GPC faults are reported with immediately higher priority than the corresponding synchronous External abort for that access. See MMU fault prioritization from a single address translation stage.

I_{YHXKR} When GPCCR_EL3.GPCP is 1, the PE can omit GPCs on fetches of translation table entries that are Table descriptors for stage 2 translation table walks, for a performance optimization. Enabling this optimization is dependent on the security model of the system.

R_{RWGJH} If GPCCR_EL3.GPCP is 1 and the PE omits a GPC when fetching a translation table entry that is a Table descriptor for a stage 2 translation table walk, then when the entry is processed:

- If the entry is not a Table descriptor, the PE must complete a GPC on the address of the entry before the translation completes. This is permitted to occur concurrently with a GPC on the entry content, as long as the priority order for fault reporting is maintained.
- Arm strongly recommends that the PE perform a GPC on the address of the entry if the entry generates a fault that would report syndrome information from that entry. If the GPC results in a GPC fault, the fault is reported as though GPCCR_EL3.GPCP is 0.

I_{ZDKBX} When a load or store instruction results in accesses to two granules, the access to each granule is subject to a GPC.

R_{GVSNZ} If an instruction that stores to memory generates a GPC fault, the value of each memory location that instruction attempted to store to is either:

- Unchanged if access to the location triggered the GPC fault.
- UNKNOWN for any location for which access did not trigger a fault or debug event.

I_{RFMWD} A GPC fault might occur at any point in the translation process that requires access to a PA. For example, to perform a store at EL1, a PE performs:

- A stage 1 translation for the VA.
- Stage 2 translations for:
  - The output IPA from the stage 1 translation.
  - The IPA of each stage 1 descriptor.
- GPCs for all of:
  - The PA that the VA translates to.
  - The PA of each stage 1 descriptor.
  - The PA of each stage 2 descriptor.

D1.4.5.4 SVE synchronous memory faults

I_{YLCFS} In this section, the term *Memory fault* refers to the detection of an erroneous condition or debug event as a result of performing a data memory access for an SVE load or store instruction.

R_{SKNTR} When an SVE load or store instruction results in a data memory access, the detection of any of the following conditions is considered to be a *Memory fault*:

- MMU fault.
- Alignment fault, excluding the SP alignment fault.
- Synchronous External abort, including synchronous parity error or ECC error.
- Watchpoint debug event.
- When FEAT_MTE2 is implemented, a synchronous Tag Check Fault.

For more information, see Memory aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{LDTYT} The detection or generation of a *Memory fault* by an SVE load or store instruction may or may not cause a synchronous Data Abort or Watchpoint exception to be taken.

R_{LVCNH} Unless otherwise specified in this section, SVE vector load and store instructions that detect a *Memory fault* cause a Data Abort or Watchpoint exception to be taken, as described in SVE Data Abort and Watchpoint exceptions.

R_{FPVTT} A *Memory fault* detected for a memory location that can only be accessed by an *Inactive element* of a predicated SVE vector load or store instruction is ignored and does not cause a Data Abort or Watchpoint exception to be taken by that instruction.

D1.4.5.4.1 SVE Data Abort and Watchpoint exceptions

R_{KJPTS} Unless otherwise specified in this section, a Data Abort and Watchpoint exception caused by an SVE load or store instruction follows the behaviors described in Definition of a precise exception and imprecise exception, Exception entry, and Prioritization of Synchronous exceptions taken to AArch64 state.

I_{DGSNC} SVE load and store instructions can generate a sequence of single-copy atomic memory accesses that might not be completed due to a *Memory fault* causing a Data Abort or Watchpoint exception to be taken during the memory access sequence.

R_{ZKBRX} When the execution of an SVE load or store instruction detects multiple *Memory faults* caused by different single-copy atomic memory accesses, the *Memory faults* are not prioritized by the architecture.

I_{MSVYK} When an SVE load or store instruction that has not been architecturally executed is restarted after an exception return, any memory locations that it accessed before taking the exception might be accessed again. Therefore, SVE load or store instructions might perform multiple accesses to memory locations that do not cause a *Memory fault* but which are sensitive to the number of accesses, or have been modified between the accesses.

R_{ZXNXT} When execution of an SVE load instruction causes a Data Abort or Watchpoint exception to be taken and the destination is not a vector register that is also used as a base or index register by the instruction, then all elements of the destination register become UNKNOWN.

R_{SNJQR} When execution of an SVE load instruction causes a Data Abort or Watchpoint exception to be taken and the destination is a vector register that is also used as a base or index register by the instruction, then all elements of the destination vector register are restored to their original value prior to execution of the load instruction.

R_{DWYCY} When execution of an SVE predicated vector store instruction causes a Data Abort or Watchpoint exception to be taken, one or more of the following occurs:

- Memory locations that are associated with *Active elements* and which do not generate a *Memory fault* become UNKNOWN.
- Memory locations that are associated with *Active elements* and which generate a *Memory fault* are unchanged.
- Memory locations that are only associated with *Inactive elements* are unchanged.

D1.4.5.4.2 SVE First-fault and Non-fault loads

I_{JZBGW} When a memory access performed for the *First active element* of an SVE First-fault vector load instruction detects a *Memory fault*, this causes a synchronous exception to be taken as described in SVE Data Abort and Watchpoint exceptions.

I_{DXBNG} When a memory access performed for the *First active element* of an SVE First-fault vector load instruction does not detect a *Memory fault*, the other elements are handled in the same way as the elements of an SVE Non-fault vector load instruction.

R_{JKGYJ} A Data Abort or Watchpoint exception is not generated when a *Memory fault* is detected by a memory access performed for any of the following elements:

- Any *Active element* of an SVE Non-fault vector load.
- Any *Active element* of an SVE First-fault vector load except for the *First active element*.

R_{MNKNV} The PE can choose to suppress a memory access performed for any of the following elements:

- Any *Active element* of an SVE Non-fault vector load.
- Any *Active element* of an SVE First-fault vector load except for the *First active element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{YFTRN} When a memory access performed for any of the following elements detects a *Memory fault*, an asynchronous Tag check fault, or is suppressed for any other reason, the FFR predicate elements starting from that element number, up to and including the highest-numbered element, are set to FALSE:

- Any *Active element* of an SVE Non-fault vector load.
- Any *Active element* of an SVE First-fault vector load except for the *First active element*.

Note

An asynchronous Tag check fault setting elements in FFR to FALSE is required to be synchronous with respect to the instruction stream as per R_{XXMMP}.

I_{BMQVT} An FFR predicate element is never set to TRUE by an SVE vector load, therefore the fault indications are cumulative.

I_{BXNTL} A memory access performed for the First active element of an SVE First-fault vector load instruction is not precisely equivalent to the memory access for an Active element of a regular SVE predicated vector load instruction. When the initial value of the FFR element corresponding to the First active element is FALSE, then even if an access is performed for that element and does not detect a *Memory fault*, the content of the corresponding destination vector element is still CONSTRAINED UNPREDICTABLE, as described in rule R_{NGFTJ}.

R_{NGFTJ} After an SVE Non-fault vector load or First-fault vector load is executed, each destination vector element contains one of the values listed in the following table:

Table D1-8 Contents of destination vector element after an SVE Non-fault vector load or First-fault vector load

Corresponding FFR element	Vector element status	Content of destination vector element
FALSE	Active	Each byte of the element contains an independently CONSTRAINED UNPREDICTABLE choice of one of the following: • 0. • The previous value of that byte in the destination vector register. • If and only if all of the following apply, the value read from memory: — The memory access for that byte was not an access to any type of Device memory, or for a First-fault vector load was an access to Device memory inside the 64-byte window of the First active element. See Device memory for more information. — The memory access for that byte does not return information that cannot be accessed at the current or a lower level of privilege.
FALSE	Inactive	A CONSTRAINED UNPREDICTABLE choice of: • 0. • The previous value of that vector element.
TRUE	Active	The value read from memory.
TRUE	Inactive	0.

R_{WCHSR} In Table D1-8, watchpoints and MTE Tag check faults are not a mechanism for preventing access to memory.

D1.4.5.5 Undefined Instruction exceptions injected by PSTATE.UINJ

R_{NKNRG} If FEAT_UINJ is implemented, then any attempt to execute any instruction when PSTATE.UINJ is 1 causes an Undefined Instruction exception with ESR_ELx.EC `0b000000`.

I_{HPGHL} Software can set SPSR_ELx.UINJ to 1, and on exception return that value is copied to PSTATE.UINJ. Then, when the PE attempts to execute the next instruction, an Undefined Instruction exception occurs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.5.6 Prioritization of Synchronous exceptions taken to AArch64 state

RZFGJP The following list shows the priorities for synchronous exceptions taken to an Exception level using AArch64. The highest priority is 1. The AArch64 priority numbers correlate with the equivalent AArch32 and Debug prioritization lists.

Priority	Synchronous exception type
1	Reset Catch debug event. See Reset Catch debug events.
2	Exception Catch debug event if it has a priority of 2. See Exception Catch debug event.
3	Halting Step debug event. See Halting Step debug events.
4	Software Step exception. See Software Step exceptions.
5	Exception Catch debug event if it has a priority of 5. See Exception Catch debug event.
6	PC alignment fault exception. See PC alignment checking.
7	Illegal Execution state exceptions. It is IMPLEMENTATION DEFINED whether this is prioritized here or at 10. See Illegal exception returns from AArch64 state.
8	Instruction Abort exceptions, including exceptions generated by an MMU fault for the translation of an instruction fetch. If FEAT_RME is implemented, this includes GPC exceptions and Granule Protection Faults. See MMU fault prioritization from a single address translation stage.
9	Breakpoint exceptions, or AArch32 Address Matching Vector Catch exceptions. See Breakpoint exceptions and Vector Catch exceptions.
10	Illegal Execution state exceptions. It is IMPLEMENTATION DEFINED whether this is prioritized here or at 7. See Illegal exception returns from AArch64 state.
11	If FEAT_UINJ is implemented, an Undefined Instruction exception caused by an attempt to execute an instruction when PSTATE.UINJ is 1.
12	Software Breakpoint exceptions, caused by execution of a Breakpoint instruction.
13	Branch Target exception. See PSTATE.BTYPE.
14	EXLOCK Exception. See Guarded Control Stack exceptions.
15	Exceptions taken to EL3 due to the MPAM3_EL3.TRAPLOWER trap control.
16	Exceptions taken to EL3 due to the MPAMBW3_EL3.nTRAPLOWER trap control.
17	Exceptions taken from EL1 to EL2 because of the configuration of one of the following: • HSTR_EL2.Tn. • HCR_EL2.TIDCP. • If FEAT_NV is implemented, the Effective value of HCR_EL2.{NV1, NV}. Exceptions taken from EL0 to EL1 or EL2 because of the configuration of the following: • If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP. Exceptions taken from EL0 to EL2 because of the configuration of the following: • If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
18	Exceptions that occur as a result of attempting to execute an instruction that is UNDEFINED for one or more of the following reasons: • Attempting to execute a `UDF` instruction. • Attempting to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation. • Attempting to execute an instruction that is defined to never be accessible at the current Exception level and Security state, regardless of any enables or traps. • Debug state execution of an instruction encoding that is not accessible in Debug state. • Non-debug state execution of an instruction encoding that is not accessible in Non-debug state. • Attempting to execute an `HVC` instruction, when `HVC` instructions are disabled by SCR_EL3.HCE or HCR_EL2.HCD. • Attempting to execute an `MSR` or `MRS` instruction to SP_EL0, when SPSel.SP is 0. • Attempting to execute an `MSR` or `MRS` instruction using an _EL12 register name, when the Effective value of HCR_EL2.E2H is 0. • If FEAT_FGT and FEAT_PMUv3 are implemented, attempting to execute an `MSR` or `MRS` instruction in AArch64 state, or an `MCR` or `MRC` instruction in AArch32 state, that accesses a register associated with an unimplemented PMU event counter. • If FEAT_SRMASK is implemented, attempting to execute an `MSR` write access to any of the following: — An EL1 mask register from EL1, when the EL1 mask register is nonzero and the access is not trapped. — An EL2 mask register from EL2, when the EL2 mask register is nonzero. • Attempting to execute an `HLT` instruction, when `HLT` instructions are disabled by EDSCR.HDE or halting is prohibited. • Attempting to execute an SVE instruction that is UNDEFINED because the *Maximum implemented SVE vector length* in both Streaming SVE mode, and Non-streaming SVE mode, is less than the minimum required vector length for that instruction. See Maximum implemented SVE vector lengths. • Attempting to execute an SME instruction that is UNDEFINED because the *Maximum implemented Streaming SVE vector length* is less than the minimum required vector length for that instruction. See Maximum implemented SVE vector lengths. • Attempting to execute an SVE `MOVPRFX` instruction with an immediately following instruction that the `MOVPRFX` cannot predictably prefix, when the choice of resulting behavior is for either instruction to generate an Undefined Instruction exception. See Move prefix.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
18 cont'd	• If in Debug state: — Attempting to execute a `DCPS1` instruction in Non-secure EL0 HCR_EL2.TGE is 1. — Attempting to execute a `DCPS2` instruction in EL1 or EL0, when EL2 is disabled in the current Security state or is not implemented. — Attempting to execute a `DCPS3` instruction, when EDSCR.SDD is 1 or EL3 is not implemented. — Attempting to execute in EL2, EL1, or EL0 an instruction that is configured by EL3 control registers to trap to EL3, when EDSCR.SDD is 1. It is IMPLEMENTATION DEFINED whether this type of exception is prioritized at this level or has the priority of the original trap exception. • If in AArch32 state: — If SCTLR_EL1.ITD is 1, attempting to execute an instruction in an IT block. — If SCTLR_EL1.SED is 1, attempting to execute a `SETEND` instructions. — If SCTLR_EL1.CP15BEN is 0, attempting to execute a `CP15DMB`, `CP15DSB`, or `CP15ISB` barrier instruction. • Attempting to execute an instruction that is UNDEFINED because all of the following apply: — EL0 is using AArch32. — EL1 is using AArch64. — Programming FPCR.{Stride, Len} to nonzero values is supported. — One or more of FPCR.{Stride, Len} is nonzero.
19	If FEAT_SME or FEAT_SVE is implemented, an exception taken to EL1, or to EL2 if HCR_EL2.TGE is 1, due to one of the following trap controls: • CPACR_EL1.SMEN or CPACR_EL1.ZEN.
20	If all of the following apply, any exception that is generated by a configurable trap or enable: • The exception is not already covered by priorities 4-19. • The exception is taken to EL1, or, if HCR_EL2.TGE is 1, is taken to EL2.
21	As the result of one of the following configuration controls, an exception taken from EL0 to EL2: • When the exception is taken to AArch64 state: — HSTR_EL2.T*n*. — HCR_EL2.TIDCP. • When the exception is taken to AArch32 state: — HSTR.T*n*. — HCR.TIDCP.
22	If FEAT_SME or FEAT_SVE is implemented, an exception taken to EL2 due to one of the following trap controls: • CPTR_EL2.SMEN or CPTR_EL2.ZEN, when the Effective value of HCR_EL2.E2H is 1. • CPTR_EL2.TSM or CPTR_EL2.TZ, when the Effective value of HCR_EL2.E2H is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
23	As the result of one of the following configuration controls, an exception taken to EL2: • When the exception is taken to AArch64 state: controls in CPTR_EL2, other than those controls covered in priority 22. • When the exception is taken to AArch32 state: controls in HCPTR.
24	As the result of a configuration control in one of the following registers, an exception taken to EL2: • When the exception is taken to AArch64 state: — HCR_EL2: Other than a setting made in the {TIDCP, NV} fields, and MRS/MSR instruction using an _EL12 register name when the Effective value of HCR_EL2.E2H is 0. — CNTHCTL_EL2: Any setting. — MDCR_EL2: Any setting. Any of the following fine-grained traps: — HAFGRTR_EL2. — HDFGRTR_EL2. — HDFGRTR2_EL2. — HDFGWTR_EL2. — HDFGWTR2_EL2. — HFGITR_EL2. — HFGITR2_EL2. — HFGRTR_EL2. — HFGRTR2_EL2. — HFGWTR_EL2. — HFGWTR2_EL2. When the exception is taken to AArch32 state: — HCR: Other than the TIDCP bit. — CNTHCTL: Any setting. — HDCR: Any setting.
25	Other than an exception defined by priorities 4-24 inclusive, any exception that is the result of a configurable access to instructions, where the exception is taken to EL2.
26	An exception caused by the SMC instruction being UNDEFINED because SCR_EL3.SMD is 1.
27	An exception caused by any of the following Exception generating instructions: • `HVC`. • `SMC`. • `SVC`. For AArch64 State, see Branches, Exception generating, and System instructions.
28	If FEAT_SME or FEAT_SVE is implemented, an exception taken to EL3 due to one of the following trap controls: • CPTR_EL3.ESM or CPTR_EL3.EZ.
29	An exception taken to EL3 as the result of configuration controls in CPTR_EL3, other than those controls covered in priority 28. It is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
30	Exceptions Trapped to EL3 from Secure EL1 using AArch32.
31	An Exception taken to EL3 from EL0, EL1, or EL2 as a result of configuration controls in MDCR_EL3. It is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception.
32	Other than an exception defined by priorities 4-31, inclusive, any exception taken to EL3 because of a configurable access to an instruction. If an exception generated by a configurable instruction control is treated as UNDEFINED as a result of EDSCR.SDD configuration, then it is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception.
33	*SME illegal instruction exceptions* due to attempting to execute an SME instruction when PSTATE.SM is 0, or attempting to execute certain Advanced SIMD or SVE instructions when PSTATE.SM is 1. See Streaming SVE mode.
34	SME illegal instruction exceptions due to attempting to access *ZA storage* or the *ZT0 register* when PSTATE.ZA is 0. See ZA storage and SME2 ZT0 register.
35	SME illegal instruction exceptions due to attempting to access the ZT0 register when an SMCR_ELx.EZT0 control is 0. See SME2 ZT0 register and SME traps and exceptions.
36	An exception as a result of attempting to execute an SVE or SME instruction that is UNDEFINED because the Effective SVE vector length is less than the minimum required vector length for that instruction. See also Configurable SVE vector lengths.
37	When FEAT_FPAC is implemented, a PAC Fail exception. See Faulting on pointer authentication.
38	When SIMD & Floating Point is implemented, any trapped floating-point exception. See Floating-point exceptions and exception traps.
39	This priority number is used by debug events.
40	An SP alignment fault. See SP alignment checking.
41	When FEAT_MOPS is implemented, Memory Copy and Memory Set exceptions caused by the CPY* or SET* instruction algorithm option of the implementation that an exception is taken from being different from the algorithm option of the implementation that is returned to. See Memory Copy and Memory Set exceptions.
42	When FEAT_MOPS is implemented, Memory Copy and Memory Set exceptions that are not defined by priority 41. See Memory Copy and Memory Set exceptions.
For priorities 43, 44, 45, 46 or 47: when an instruction results in more than one single-copy atomic memory access, the architecture will not prioritize the individual Synchronous exceptions generated as the result of the multiple memory accesses. When FEAT_GCS is implemented, a stricter requirement is defined for the `GCSSS2` instruction in R_{XTZVS}.	
43	In descending priority order: a) Data Abort exceptions on translation table walks and translation table entry updates. b) GPC Exceptions on translation table walks and translation table entry updates, if FEAT_RME is implemented. c) Data Abort exceptions due to synchronous External aborts on translation table walks and translation table entry updates. d) Data Abort exceptions arising from an MMU fault not on a translation table walk, that are not covered by priorities 44 or 46. See also: • MMU fault prioritization from a single address translation stage. • External aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
44	If prioritized here, then in descending priority order: a) Data Abort exceptions due to a *Granule Protection Fault* (GPF) on the final physical address access of the address translation process, if FEAT_RME is implemented. b) GPC exceptions on the final physical address access of the address translation process, if FEAT_RME is implemented. **c) Any of:** i) Data Abort exceptions due to synchronous External aborts on the final physical address access of the address translation process. ii) Data Abort exceptions on Tag Check Faults, if FEAT_MTE2 is implemented. Whether these are prioritized here or as priority 46 is IMPLEMENTATION DEFINED. See also: • Tag Check Faults. • External aborts.
45	Watchpoint exception. See Watchpoint exceptions.
46	The exceptions listed for priority 44 if they are prioritized as 46.
47	Guarded Control Stack Data Check exceptions. See Guarded Control Stack exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.5.7 Trapping of floating-point exceptions

R_{TSQFJ} Execution of a floating-point instruction, or execution of an Advanced SIMD or SVE instruction that performs floating-point operations, can generate an exceptional condition, called a floating-point exception.

R_{BBSGN} For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

I_{YRVSW} The architecture does not support asynchronous reporting of floating-point exceptions. If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements on when that synchronous exception is generated do not apply.

I_{KJTLV} For any of the implemented floating-point exceptions listed in R_{BBSGN}, FPCR provides control bits to enable synchronous exception generation.

R_{VJFJR} The Exception level that a Trapped Floating-point exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

R_{TRWMV} If an implementation includes synchronous exception generation for floating-point exceptions in AArch64 state, when the execution of separate operations in separate SIMD or SVE elements causes multiple floating-point exceptions, the ESR_ELx reports only the exceptions associated with one element that the instruction uses. The architecture does not specify which element is reported.

R_{VJNYH} When a floating-point exception is trapped, all of the following apply:

- When the trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value of the FPSR immediately before the instruction that generated the trapped floating-point exception.When the trapped floating-point exception is taken, if the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which untrapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative floating-point exception bits having the value 1.
- In ESR_ELx at the target Exception level, all of the following apply:
  - The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
  - If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.
  - If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

R_{FTPRW} For trapped floating-point exceptions from Advanced SIMD or SVE instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.

D1.4.5.8 Memory Copy and Memory Set exceptions

I_{TPJXM} If an exception is taken during the execution of Memory Copy and Memory Set instructions, and execution of these instructions restarts on a physical hardware PE implementation that is different from where the exception was taken from, then a Memory Copy or Memory Set exception can be generated.

R_{JQZHK} For CPYE*, CPYFE*, CPYFM*, and CPYM* instructions, if the instruction algorithm option indicated by the value of PSTATE.C is different from the PSTATE.C value of the implementation, then a Memory Copy exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{XQQXB} For SETE*, SETGE*, SETGM*, and SETM* instructions, if the instruction algorithm option indicated by the value of PSTATE.C is different from the PSTATE.C value of the implementation, then a Memory Set exception is generated.

R_{PHLLT} For CPYE*, CPYFE*, CPYFM*, and CPYM* instructions, if the instruction alignment and size requirements of the implementation that an exception is taken from are different from the alignment and size requirements of the implementation that is returned to, then a Memory Copy exception is generated.

R_{BKCPW} For SETE*, SETGE*, SETGM*, and SETM* instructions, if the instruction alignment and size requirements of the implementation that an exception is taken from are different from the alignment and size requirements of the implementation that is returned to, then a Memory Set exception is generated.

I_{QJFTK} The Memory Copy and Memory Set exceptions that are caused by alignment and size requirement differences can be avoided if the following apply:

- The CPYFP*, CPYP*, SETGP*, or SETP* instruction is executed before the appropriate CPYFM*, CPYM*, SETGM*, or SETM* on the same implementation.
- The CPYFM*, CPYM*, SETGM*, or SETM* instruction is executed before the appropriate CPYFE*, CPYE*, SETGE*, or SETE* on the same implementation.

I_{LNTYW} The generic return routine for Memory Copy and Memory Set exceptions is to adjust the register arguments to be consistent with performing a new memory copy or memory set, and then to return to the CPYFP*, CPYP*, SETGP*, or SETP* instruction to restart the memory copy or memory set.

I_{TTVQX} The generic return routine for Memory Copy and Memory Set exceptions can be used by relying on the Memory Copy and Memory Set instructions being placed consecutively in memory - prologue, then main, and then epilogue.

I_{CNTMJ} The generic return routine for Memory Copy exceptions is shown in the following pseudocode:

```
if FormatOption == 0b00 || FormatOption == 0b11 then
   //format is from option B
   if SPSR_ELx.N=='1' then
      //backward copy
      DestReg = DestReg - SizeReg;
      SrcReg = SrcReg - SizeReg;
else // format is from option A;
   if SizeReg<63>=='1' then
      // forward copy
      DestReg = DestReg + SizeReg;
      SrcReg = SrcReg + SizeReg;
      SizeReg = -1*SizeReg;
if FromEpilogue then
      ReturnAddress = ELR_ELx - 8;
else
      ReturnAddress = ELR_ELx - 4;
```

I_{MWFQH} The generic return routine for Memory Set exceptions is shown in the following pseudocode:

```
if FormatOption == 0b01 || FormatOption == 0b10 then
   // format is from Option A
   // forward set
   DestReg = DestReg + SizeReg;
   SizeReg = -1*SizeReg;
if FromEpilogue then
   ReturnAddress = ELR_ELx - 8;
else
   ReturnAddress = ELR_ELx - 4;
```

I_{PRXVQ} The information in SPSR_ELx.C (or, equivalently, the ESR_ELx.ISS.FormatOption field), SPSR_ELx.N, and the sign of the value in the Xn register is sufficient to determine the current format of the Xd, Xs, and Xn registers, and perform any required reformatting, before restarting the memory copy or memory set. However, it is not required to use FEAT_MOPS instructions and the memory copy or memory set could alternatively be restarted using a software implementation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{ZHRJP} The Exception levels that Memory Copy and Memory Set exceptions are taken to are as follows:

- From EL0, to EL1 if HCR_EL2.TGE is 0. Otherwise, they are taken to EL2.
- From EL1, to either EL1 or EL2, depending on the value of HCRX_EL2.MCE2.
- From EL2 or EL3, to the same Exception level.

I_{NXHPS} The HCRX_EL2.MCE2 control bit allows Memory Copy and Memory Set exceptions to be handled in a system where Memory Copy and Memory Set instructions might be used at EL1 at times when taking an exception to EL1 is not acceptable. This is necessary only if there is a possibility of the migration of EL1 code between different physical hardware PE implementations when EL1 is not tolerant of taking an exception.

I_{ZWLQV} Arm expects that software running at EL1 will only enable EL0 use of the Memory Copy and Memory Set instructions with SCTLR_EL1.MSCEn if it implements the generic return routine for Memory Copy and Memory Set exceptions.

D1.4.6 Asynchronous exception types

R_{ZNTKG} The Arm architecture Exception model distinguishes the following classes of asynchronous exceptions:

- Physical Interrupts
- Virtual Interrupts
- PMU exceptions, see Exception-based event profiling.

There are the following types of physical interrupt:

- SError (also described as a System Error)
- IRQ
- FIQ

There are the following types of virtual interrupt:

- vSError (also described as a Virtual System Error)
- Delegated SError
- vIRQ
- vFIQ

R_{SLLMN} IRQ, FIQ, vIRQ, and vFIQ interrupts are precise asynchronous exceptions.

R_{GGWMZ} If FEAT_NMI is implemented, then IRQ, FIQ, vIRQ, and vFIQ interrupts can have Superpriority as an additional attribute.

I_{PYQBL} When SCTLR_ELx.NMI is 0, the attribute of Superpriority has no effect on interrupts taken to ELx.
See also:

- Physical interrupt masking.
- Virtual asynchronous exception masking.

R_{PBKNX} The mechanism by which Superpriority is assigned is IMPLEMENTATION DEFINED.

R_{DXHJZ} Each physical interrupt type can be assigned a target Exception level of EL1, EL2, or EL3.

I_{VMWYT} ISR_EL1 shows the pending status of the physical interrupts, the physical SError exception, the virtual interrupts, and the virtual and delegated SError exceptions, as follows:

- When read from EL1:
  - The pending status of whichever virtual interrupts are enabled, and of the virtual SError exception if enabled. If the virtual SError exception is not enabled and the delegated SError exception is enabled, the pending status of the delegated SError exception.
- When read from EL2:
  - The pending status of the delegated SError exception, if the delegated SError exception is enabled.

Otherwise, when read from EL1, EL2, or EL3, the pending status of the physical interrupts, and of the physical SError exception.

I_{RWYFW} An implementation might support other mechanisms for signaling a virtual interrupt.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{WRGNC} The following Profiling exceptions can also be asynchronous exceptions:

- PMU Profiling exceptions. See Exception-based event profiling.
- TRBE Profiling exceptions. See Trace buffer management.
- SPE Profiling exceptions. See Profiling Buffer management.

I_{XHCGZ} The mechanism by which physical interrupts are signaled to the PE are IMPLEMENTATION DEFINED and might be either edge or level sensitive. A common implementation choice is that the IRQ and FIQ interrupts are level sensitive, and this is supported by the Generic Interrupt Controller (GIC).

I_{YSKZH} The physical SError exception is often used, amongst other things, for communicating External aborts from the memory system that are to be taken asynchronously. For more information, see PE error handling.

R_{KTWMT} For an External abort generated by the memory system that is taken asynchronously using the SError exception, the SError exception always behaves as an edge-triggered interrupt. For any other sources of SError exceptions, it is IMPLEMENTATION DEFINED whether they are edge-triggered or level-sensitive.

R_{HSFRC} When taking the SError exception to an Exception level using AArch64, ESR_ELx at the target Exception level is updated to describe the SError exception.

If FEAT_RAS is implemented, then:

- When taking a vSError exception to an Exception level using AArch64, ESR_ELx at the target Exception level is updated with exception syndrome information from VSESR_EL2.
- If FEAT_E3DSE is implemented, then when taking a delegated SError exception, ESR_ELx at the target Exception level is updated with exception syndrome information from VSESR_EL3.

R_{RKDNV} When taking an IRQ, vIRQ, FIQ or vFIQ interrupt to an Exception level using AArch64, ESR_ELx at the target Exception level is not updated.

D1.4.6.1 Virtual asynchronous exceptions

R_{MLLVN} Each virtual interrupt type can be independently enabled from EL2. If a virtual interrupt type is enabled from EL2, the target Exception level for the corresponding physical interrupt is not EL1.

R_{RTLGR} Pending vIRQ and vFIQ interrupts can be assigned Superpriority by using controls in HCRX_EL2.

R_{BCXJB} If HCR_EL2.TGE is 0, then setting an HCR_EL2.{FMO, IMO} routing control bit to 1 enables the corresponding virtual interrupt.

If HCR_EL2.TGE is 1, then virtual interrupts are disabled.

R_{CDCKC} If HCR_EL2.TGE is 0, then setting the Effective value of either the HCR_EL2.AMO or HCRX_EL2.TMEA routing control bit to 1 enables the virtual SError exception.

If HCR_EL2.TGE is 1, then virtual SError exceptions are disabled.

R_{NVFXB} If a virtual asynchronous exception type is enabled, that type of interrupt can be generated by any one of the following:

- Execution at EL0 or EL1 if the corresponding virtual interrupt pending bit, HCR_EL2.{VSE, VI, VF}, is 1.
- For a vIRQ or a vFIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller.

R_{SDVLM} If a virtual interrupt is disabled, the virtual interrupt cannot be taken.

I_{NJPHG} The following table describes the bits that enable virtual interrupts, the bits that cause virtual asynchronous exceptions to be pending in HCR_EL2, and the bits that enable signaling of pending virtual interrupts with Superpriority in HCRX_EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Virtual asynchronous exception type	Enable control	Cause a virtual asynchronous exception to be pending	Pending virtual interrupt has Superpriority
vSError	HCR_EL2.AMO or HCRX_EL2.TMEA	HCR_EL2.VSE	n/a
vIRQ	HCR_EL2.IMO	HCR_EL2.VI	HCRX_EL2.VINMI
vFIQ	HCR_EL2.FMO	HCR_EL2.VF	HCRX_EL2.VFNMI

R_{MZWNH} When taking a vIRQ or a vFIQ interrupt, the corresponding virtual interrupt pending bit in HCR_EL2 retains its state.

I_{QVCWH} If the virtual interrupt pending bits are used, the vIRQ or vFIQ exception handler must cause software executing at EL2 or EL3 to set their corresponding virtual interrupt pending bits to 0.

I_{YPRNB} When taking a vSError exception, HCR_EL2.VSE is cleared to 0.

I_{LFSXS} A virtual SError is never taken to EL2 or EL3, regardless of the values of HCRX_EL2.TMEA and SCR_EL3.TMEA.

D1.4.6.2 Delegated SErrors

I_{NVRSW} If FEAT_E3DSE is implemented, *delegated SError exceptions* are defined by SCR_EL3.EnDSE, SCR_EL3.DSE, VSESR_EL3, and VDISR_EL3.

R_{TNWJT} If the Effective value of SCR_EL3.{EA, TMEA} is not {0, 0}, then setting the Effective value of the SCR_EL3.EnDSE control bit to 1 enables the delegated SError exception.
If the Effective value of SCR_EL3.{EA, TMEA} is {0, 0}, or the Effective value of SCR_EL3.EnDSE is 0, then delegated SError exceptions are disabled.

R_{WLQXL} If delegated SError exceptions are enabled, then setting SCR_EL3.DSE to 1 pends a delegated SError exception while executing at EL0, EL1, or EL2 .

R_{RWMSM} If the delegated SError exception is disabled, the delegated SError exception cannot be taken.

I_{QKSJR} When taking a delegated SError exception, SCR_EL3.DSE is cleared to 0.

I_{LVWLF} A delegated SError is never taken to EL3, regardless of the value of SCR_EL3.TMEA.

D1.4.6.3 Establishing the target Exception level of an asynchronous exception

R_{QZPXL} The R_{NMMXK} table uses the following terms:

Term	Meaning
SCR_EL3	The Effective value of a field in SCR_EL3.
NS	If FEAT_RME is implemented, then if Secure state is not implemented, the Effective value of SCR_EL3.NS is 1.
EEL2	If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1.
IRQ FIQ	If EL3 is not implemented, the Effective value of these fields is 0.
RW	If EL3 is not implemented, the Effective value of SCR_EL3.RW is 1.
HCR	If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.
TGE	If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.
IMO FMO	If EL2 is not implemented, the Effective value of these fields is 0.
E2H	If EL2 is not implemented, the Effective value of HCR.E2H or HCR_EL2.E2H is 0.
RW	If EL2 is not implemented, the Effective value of HCR_EL2.RW is the same as the Effective value of SCR_EL3.RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Term	Meaning
EL1	The exception is taken to EL1 using AArch64.
EL2	The exception is taken to EL2 using AArch64.
EL3	The exception is taken to EL3 using AArch64.
C	The interrupt is not taken and remains pending, regardless of the PSTATE.{I, F, ALLINT} interrupt masks.
IRQ, FIQ	The exception is taken to the AArch32 IRQ mode or the AArch32 FIQ mode accordingly.
Hyp	The exception is taken to AArch32 Hyp mode.
n/a	Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

R_{NMMXK} The following table describes the routing of physical FIQ and IRQ interrupts if the highest implemented Exception level is using AArch64 and either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Table D1-12 FIQ and IRQ interrupt exception targets

SCR_EL3				HCR				Target when taken from			
NS	EEL2	IRQ FIQ	RW	TGE	IMO FMO	E2H	RW	EL0	EL1	EL2	EL3
0	0	0	0	x	x	x	x	IRQ, FIQ	IRQ, FIQ	n/a	C
0	0	0	1	x	x	x	x	EL1	EL1	n/a	C
0	0	1	x	x	x	x	x	EL3	EL3	n/a	EL3
0	1	0	x	0	0	0	0	IRQ, FIQ	IRQ, FIQ	C	C
0	1	0	x	0	0	0	1	EL1	EL1	C	C
0	1	0	x	0	0	1	x	EL1	EL1	C	C
0	1	0	x	0	1	x	x	EL2	EL2	EL2	C
0	1	0	x	1	x	x	x	EL2	n/a	EL2	C
0	1	1	x	0	x	x	x	EL3	EL3	EL3	EL3
0	1	1	x	1	x	x	x	EL3	n/a	EL3	EL3
1	x	0	0	0	0	n/a	n/a	IRQ, FIQ	IRQ, FIQ	Hyp	C
1	x	0	0	0	1	n/a	n/a	Hyp	Hyp	Hyp	C
1	x	0	0	1	x	n/a	n/a	Hyp	n/a	Hyp	C
1	x	0	1	0	0	0	0	FIQ	FIQ	C	C
1	x	0	1	0	0	0	1	EL1	EL1	C	C
1	x	0	1	0	0	1	x	EL1	EL1	C	C
1	x	0	1	0	1	x	x	EL2	EL2	EL2	C
1	x	0	1	1	x	x	x	EL2	n/a	EL2	C
1	x	1	x	0	x	x	x	EL3	EL3	EL3	EL3
1	x	1	x	1	x	x	x	EL3	n/a	EL3	EL3

R_{ZZYHK} The R_{MGNFZ} and R_{VRBFW} tables use the following terms:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Term	Meaning
SCR_EL3	The Effective value of a field in SCR_EL3.
EA	If EL3 is not implemented, the Effective value of SCR_EL3.EA is 0.
TMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of SCR_EL3.TMEA is 0.
NMEA	If FEAT_DoubleFault is not implemented, the Effective value of SCR_EL3.NMEA is 0.
HCR_EL2	The Effective value of a field in HCR_EL2.
HCR	The Effective value of a field in HCR.
TGE	If EL2 is not implemented, or if EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.TGE or HCR.TGE is 0.
E2H	If EL2 is not implemented, or if EL2 is disabled in the current Security state, the Effective value of HCR_EL2.E2H is 0.
AMO	If EL2 is not implemented, or if EL2 is disabled in the current Security state, the Effective value of HCR_EL2.AMO or HCR.AMO is 0.
SCTLR2_EL2	The Effective value of a field in SCTLR2_EL2.
NMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of SCTLR2_EL2.NMEA is 0. If EL2 is not implemented or EL2 is disabled in the current Security state, or if the Effective value of SCR_EL3.SCTLR2En is 0, then the Effective value of SCTLR2_EL2.NMEA is 0.
HCRX_EL2	The Effective value of a field in HCRX_EL2.
TMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of HCRX_EL2.TMEA is 0. If EL2 is not implemented or EL2 is disabled in the current Security state, or the Effective value of SCR_EL3.HXEn is 0, then the Effective value of HCRX_EL2.TMEA is 0.
SCTLR2_EL1	The Effective value of a field in SCTLR2_EL1.
NMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of SCTLR2_EL1.NMEA is 0. If the Effective value of HCRX_EL2.SCTLR2En is 0 or the Effective value of SCR_EL3.SCTLR2En is 0, then the Effective value of SCTLR2_EL1.NMEA is 0.
PSTATE	The value of a field in PSTATE.
A	The value of PSTATE.A.
EL1	The exception is taken to EL1. If EL1 is using AArch32, then the exception is taken to Abort mode.
EL2	The exception is taken to EL2 using AArch64.
EL3	The exception is taken to EL3 using AArch64.
C	The interrupt is not taken and remains pending, regardless of the PSTATE.A mask.
Abt	The exception is taken to the AArch32 Abort mode.
Hyp	The exception is taken to AArch32 Hyp mode.
n/a	Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

R_{MGNFZ} The following tables describe the routing of SError exceptions taken from ELx.

All of the following apply to these tables:

- SCR_EL3.{NS, EEL2} are not shown because R_{ZZYHK} describes Effective values involving Security states.
- Either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D1-14 SError exception target when EL2 is using AArch64

SCR_EL3			HCR_EL2			SCTLR2_EL2	HCRX_EL2	SCTLR2_EL1	PSTATE	Taken from		
EA	TMEA	NMEA	TGE	E2H	AMO	NMEA	TMEA	NMEA	A	EL0	EL1	EL2
0b0	0b0	0b0	0b0	X	0b0	0b0	0b0	X	X	EL1	EL1	C
0b0	0b0	0b0	0b0	X	0b0	0b0	0b1	0b0	0b1	EL2	EL2	C
0b0	0b0	0b0	0b0	X	0b0	0b0	0b1	X	0b0	EL1	EL1	C
0b0	0b0	0b0	0b0	X	0b0	0b0	0b1	0b1	0b1	EL1	EL2	C
0b0	0b0	0b0	0b0	X	0b0	0b1	0b0	X	X	EL1	EL1	EL2
0b0	0b0	0b0	0b0	X	0b0	0b1	0b1	0b0	0b1	EL2	EL2	EL2
0b0	0b0	0b0	0b0	X	0b0	0b1	0b1	X	0b0	EL1	EL1	EL2
0b0	0b0	0b0	0b0	X	0b0	0b1	0b1	0b1	0b1	EL1	EL2	EL2
0b0	0b0	0b0	0b0	X	0b1	X	X	X	X	EL2	EL2	EL2
0b0	0b0	0b0	0b1	X	X	X	X	X	X	EL2	n/a	EL2
0b0	0b0	0b1	0b0	X	0b0	0b0	0b0	X	X	EL1	EL1	C
0b0	0b0	0b1	0b0	X	0b0	0b0	0b1	0b0	0b1	EL2	EL2	C
0b0	0b0	0b1	0b0	X	0b0	0b0	0b1	X	0b0	EL1	EL1	C
0b0	0b0	0b1	0b0	X	0b0	0b0	0b1	0b1	0b1	EL1	EL2	C
0b0	0b0	0b1	0b0	X	0b0	0b1	0b0	X	X	EL1	EL1	EL2
0b0	0b0	0b1	0b0	X	0b0	0b1	0b1	0b0	0b1	EL2	EL2	EL2
0b0	0b0	0b1	0b0	X	0b0	0b1	0b1	X	0b0	EL1	EL1	EL2
0b0	0b0	0b1	0b0	X	0b0	0b1	0b1	0b1	0b1	EL1	EL2	EL2
0b0	0b0	0b1	0b0	X	0b1	X	X	X	X	EL2	EL2	EL2
0b0	0b0	0b1	0b1	X	X	X	X	X	X	EL2	n/a	EL2
0b0	0b1	0b0	0b0	X	0b0	0b0	X	X	0b0	EL1	EL1	EL3
0b0	0b1	0b0	0b0	X	0b0	X	0b0	0b0	0b1	EL3	EL3	EL3
0b0	0b1	0b0	0b0	X	0b0	X	0b0	0b1	0b1	EL1	EL3	EL3
0b0	0b1	0b0	0b0	X	0b0	X	0b1	0b0	0b1	EL2	EL2	EL3
0b0	0b1	0b0	0b0	X	0b0	X	0b1	0b1	0b1	EL1	EL2	EL3
0b0	0b1	0b0	0b0	X	0b0	0b1	X	X	0b0	EL1	EL1	EL2
0b0	0b1	0b0	0b0	X	0b1	X	X	X	0b0	EL2	EL2	EL2
0b0	0b1	0b0	0b0	X	0b1	X	X	X	0b1	EL2	EL2	EL3
0b0	0b1	0b0	0b1	0b0	X	X	X	X	0b1	EL2	n/a	EL3
0b0	0b1	0b0	0b1	X	X	X	X	X	0b0	EL2	n/a	EL2
0b0	0b1	0b0	0b1	0b1	X	0b0	X	X	0b1	EL3	n/a	EL3
0b0	0b1	0b0	0b1	0b1	X	0b1	X	X	0b1	EL2	n/a	EL3
0b0	0b1	0b1	0b0	X	0b0	0b0	X	X	0b0	EL1	EL1	EL3
0b0	0b1	0b1	0b0	X	0b0	X	0b0	0b0	0b1	EL3	EL3	EL3
0b0	0b1	0b1	0b0	X	0b0	X	0b0	0b1	0b1	EL1	EL3	EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCR_EL3			HCR_EL2			SCTLR2_EL2	HCRX_EL2	SCTLR2_EL1	PSTATE	Taken from		
EA	TMEA	NMEA	TGE	E2H	AMO	NMEA	TMEA	NMEA	A	EL0	EL1	EL2
0b0	0b1	0b1	0b0	X	0b0	X	0b1	0b0	0b1	EL2	EL2	EL3
0b0	0b1	0b1	0b0	X	0b0	X	0b1	0b1	0b1	EL1	EL2	EL3
0b0	0b1	0b1	0b0	X	0b0	0b1	X	X	0b0	EL1	EL1	EL2
0b0	0b1	0b1	0b0	X	0b1	X	X	X	0b0	EL2	EL2	EL2
0b0	0b1	0b1	0b0	X	0b1	X	X	X	0b1	EL2	EL2	EL3
0b0	0b1	0b1	0b1	0b0	X	X	X	X	0b1	EL2	n/a	EL3
0b0	0b1	0b1	0b1	X	X	X	X	X	0b0	EL2	n/a	EL2
0b0	0b1	0b1	0b1	0b1	X	0b0	X	X	0b1	EL3	n/a	EL3
0b0	0b1	0b1	0b1	0b1	X	0b1	X	X	0b1	EL2	n/a	EL3
0b1	X	X	0b0	X	X	X	X	X	X	EL3	EL3	EL3
0b1	X	X	0b1	X	X	X	X	X	X	EL3	n/a	EL3

Table D1-15 SError exception target when EL2 is using AArch32

SCR_EL3		HCR		PSTATE	Taken from		
EA	TMEA	TGE	AMO	A	EL0	EL1	EL2
0b0	0b0	0b0	0b0	X	Abt	Abt	Hyp
0b0	0b0	0b0	0b1	X	Hyp	Hyp	Hyp
0b0	0b0	0b1	X	X	Hyp	n/a	Hyp
0b0	0b1	0b0	0b0	0b0	Abt	Abt	Hyp
0b0	0b1	0b0	0b0	0b1	EL3	EL3	EL3
0b0	0b1	0b0	0b1	0b0	Hyp	Hyp	Hyp
0b0	0b1	0b0	0b1	0b1	Hyp	Hyp	EL3
0b0	0b1	0b1	X	0b0	Hyp	n/a	Hyp
0b0	0b1	0b1	X	0b1	Hyp	n/a	EL3
0b1	X	0b0	X	X	EL3	EL3	EL3
0b1	X	0b1	X	X	EL3	n/a	EL3

D1.4.6.3.1 Establishing the target Exception level of a delegated SError exception

R_{VRBFW} The following table describes the routing of delegated SError exceptions taken from ELx. See R_{ZZYHK} for the terms used.

All of the following apply to this table:

- SCR_EL3.{NS, EEL2} are not shown because R_{ZZYHK} describes Effective values involving Security states.
- Either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D1-16 Delegated SError exception target when EL2 is using AArch64, in an implementation that includes FEAT_E3DSE

HCR_EL2		HCRX_EL2	PSTATE	Taken from		
TGE	AMO	TMEA	A	EL0	EL1	EL2
0b0	0b0	0b0	X	EL1	EL1	C
0b0	0b0	0b1	0b0	EL1	EL1	C
0b0	0b0	0b1	0b1	EL2	EL2	C
0b0	0b1	X	X	EL2	EL2	EL2
0b1	X	X	X	EL2	n/a	EL2

D1.4.6.4 Asynchronous exception masking

R_{HDMYQ} An interrupt can be masked. A masked interrupt is not taken and remains pending.

R_{PVHHQ} Whether an interrupt is masked is determined by all of the following:

- The PSTATE.{A, I, F, ALLINT, SP} bits.
- The value of AllIntMask, as described in R_{XZPDT}.
- Whether an IRQ or FIQ interrupt has Superpriority.
- The implemented Exception levels.
- The configuration of the Hypervisor Configuration Registers.
- For an SError, whether the exception is non-maskable.
- The configuration of the Secure Configuration Register.

R_{LMWZH} If in AArch64 state and the target Exception level of an interrupt is lower than the current Exception level, the interrupt is masked.

R_{XZPDT} If FEAT_NMI is implemented and SCTLR_ELx.NMI is 1, all of the following apply:

- The AllIntMask variable is created and is defined as follows:
  `AllIntMask == PSTATE.ALLINT || (PSTATE.SP && SCTLR_ELx.SPINTMASK)`
- When AllIntMask is 1 and execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx is masked regardless of Superpriority.
- When AllIntMask is 0 and execution is at ELx:
  - An IRQ or FIQ interrupt with Superpriority that is targeted to ELx is not masked by PSTATE.I or PSTATE.F, respectively.
  - An IRQ or FIQ interrupt without Superpriority that is targeted to ELx is masked by PSTATE.I or PSTATE.F, respectively, in the same way as when the Effective value of SCTLR_ELx.NMI is 0.

I_{BXXWT} AllIntMask does not affect the masking or routing of interrupts to a higher Exception level.

R_{SFMGZ} If FEAT_NMI is not implemented or SCTLR_ELx.NMI is 0, the Effective value of AllIntMask is 0.

D1.4.6.4.1 Physical interrupt masking

R_{MHWBP} If the target Exception level of a physical IRQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

PSTATE.I	SCTLR_ELx.NMI	AllIntMask	IRQ	IRQ with Superpriority
0	0	x	Not masked	Not masked
0	1	0	Not masked	Not masked
0	1	1	Masked	Masked

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSTATE.I	SCTLR_ELx.NMI	AllIntMask	IRQ	IRQ with Superpriority
1	0	x	Masked	Masked
1	1	0	Masked	Not masked
1	1	1	Masked	Masked

R_{GFXKY} If the target Exception level of a physical FIQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

PSTATE.F	SCTLR_ELx.NMI	AllIntMask	FIQ	FIQ with Superpriority
0	0	x	Not masked	Not masked
0	1	0	Not masked	Not masked
0	1	1	Masked	Masked
1	0	x	Masked	Masked
1	1	0	Masked	Not masked
1	1	1	Masked	Masked

PSTATE.F	FIQ
0	Not masked
1	Masked

R_{GCTZC} If the target Exception level of a physical SError is the current Exception level, the PSTATE.A control determines whether the interrupt is masked.

However, if the target Exception level is the current Exception level, and if any of the following are true, the PSTATE.A control is ignored, and the exception is taken:

- The target Exception level is EL3, FEAT_DoubleFault is implemented and the Effective value of SCR_EL3.NMEA is 1.
- The target Exception level is EL2, FEAT_DoubleFault2 is implemented and the Effective value of SCTLR2_EL2.NMEA is 1.
- The target Exception level is EL1, FEAT_DoubleFault2 is implemented and the Effective value of SCTLR2_EL1.NMEA is 1.

R_{XBYXL} If the target Exception level of a physical interrupt is higher than the current Exception level, all of the following apply:

- If the target Exception level is EL3, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
- If the target Exception level is EL2 and any of the following are true, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
  - The Effective value of HCR_EL2.E2H is 0.
  - The Effective value of HCR_EL2.TGE is 0.
- If the target Exception level is EL2 and all of the following are true, the interrupt can be masked by the PSTATE.{A, I, F} bits:
  - The Effective value of HCR_EL2.E2H is 1.
  - The Effective value of HCR_EL2.TGE is 1.
  - The interrupt does not have Superpriority or SCTLR_EL2.NMI is 0.
- If the target Exception level is EL1 and the interrupt does not have Superpriority or SCTLR_EL1.NMI is 0, the interrupt can be masked by the PSTATE.{A, I, F} bits.

I_{KDYCN} The ability to execute in EL0 with interrupts taken to EL1 masked is required by some user level driver code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HPRNX} The R_{SXLWJ} table uses the following terms:

Term	Meaning
SCR_EL3	The Effective value of a field in SCR_EL3.
NS	If FEAT_RME is implemented, then if Secure state is not implemented, the Effective value of SCR_EL3.NS is 1.
EEL2	If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1. If FEAT_SEL2 is not implemented, the Effective value of SCR_EL3.EEL2 is 0.
IRQ FIQ	If EL3 is not implemented, then the Effective value of these fields is 0.
RW	If EL3 is not implemented, the Effective value of SCR_EL3.RW is 1.
HCR	If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.
TGE	If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.
E2H	If EL2 is not implemented, the Effective value of HCR.E2H or HCR_EL2.E2H is 0.
IMO FMO	If EL2 is not implemented, the Effective value of these fields is 0.
A	When the interrupt is pending, it is taken regardless of the value of the PSTATE.{A, I, F, ALLINT} interrupt masks.
B	When the interrupt is pending, it might be subject to masking, as defined in R_{MHWBP} or R_{GFXKY}. If the interrupt is masked, it is not taken. If the interrupt is not masked, it is taken.
C	When the interrupt is pending, it is not taken, regardless of the value of the PSTATE.{A, I, F, ALLINT} interrupt masks.
n/a	Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.

R_{SXLWJ} The following table describes the target and masking of physical FIQ and IRQ interrupts, if the highest implemented Exception level is using AArch64.

Table D1-21 FIQ and IRQ interrupt exception targets and masking

SCR_EL3				HCR			Effect of the interrupt mask when executing at:			
NS	EEL2	IRQ FIQ	RW	TGE	E2H	IMO FMO	EL0	EL1	EL2	EL3
0	0	0	x	x	x	x	B	B	n/a	C
0	0	1	x	x	x	x	A	A	n/a	B
0	1	0	x	0	x	0	B	B	C	C
0	1	0	x	0	x	1	A	A	B	C
0	1	0	x	1	0	x	A	n/a	B	C
0	1	0	x	1	1	x	B	n/a	B	C
0	1	1	x	0	x	x	A	A	A	B
0	1	1	x	1	x	x	A	n/a	A	B
1	x	0	0	0	n/a	0	B	B	B	C
1	x	0	0	0	n/a	1	A	A	B	C
1	x	0	0	1	n/a	x	A	n/a	B	C
1	x	0	1	0	x	0	B	B	C	C
1	x	0	1	0	x	1	A	A	B	C

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCR_EL3				HCR			Effect of the interrupt mask when executing at:			
NS	EEL2	IRQ FIQ	RW	TGE	E2H	IMO FMO	EL0	EL1	EL2	EL3
1	x	0	1	1	0	x	A	n/a	B	C
1	x	0	1	1	1	x	B	n/a	B	C
1	x	1	x	0	x	x	A	A	A	B
1	x	1	x	1	x	x	A	n/a	A	B

R_{ZSRWR} The R_{JFKMF} tables use the following terms:

Term	Meaning
SCR_EL3	The Effective value of a field in SCR_EL3.
EA	If EL3 is not implemented, the Effective value of SCR_EL3.EA is 0.
TMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of SCR_EL3.TMEA is 0.
NMEA	If FEAT_DoubleFault is not implemented, the Effective value of SCR_EL3.NMEA is 0.
HCR_EL2	The Effective value of a field in HCR_EL2.
HCR	The Effective value of a field in HCR.
TGE	If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.TGE or HCR.TGE is 0.
E2H	If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.E2H is 0.
AMO	If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.AMO or HCR.AMO is 0.
SCTLR2_EL2	The Effective value of a field in SCTLR2_EL2.
NMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of SCTLR2_EL2.NMEA is 0. If EL2 is not implemented or EL2 is disabled in the current Security state, or if the Effective value of SCR_EL3.SCTLR2En is 0, then the Effective value of SCTLR2_EL2.NMEA is 0.
HCRX_EL2	The Effective value of a field in HCRX_EL2.
TMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of HCRX_EL2.TMEA is 0. If EL2 is not implemented or EL2 is disabled in the current Security state, or the Effective value of SCR_EL3.HXEn is 0, then the Effective value of HCRX_EL2.TMEA is 0.
SCTLR2_EL1	The Effective value of a field in SCTLR2_EL1.
NMEA	If FEAT_DoubleFault2 is not implemented, the Effective value of SCTLR2_EL1.NMEA is 0. If the Effective value of HCRX_EL2.SCTLR2En is 0 or the Effective value of SCR_EL3.SCTLR2En is 0, then the Effective value of SCTLR2_EL1.NMEA is 0.
A	When the exception is pending, it is taken regardless of the value of the PSTATE.A mask.
B	When the exception is pending, it might be subject to masking, as defined in R_{GCTZC}. If the interrupt is masked, it is not taken. If the interrupt is not masked, it is taken.
C	The exception is not taken and remains pending, regardless of the PSTATE.A mask.
n/a	Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

R_{JFKMF} The following tables describe the masking of SError exceptions.

All of the following apply to these tables:

- SCR_EL3.{NS, EEL2} are not shown because R_{ZSRWR} describes Effective values involving Security states.
- Either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D1-23 SError exception masking when EL2 is using AArch64

SCR_EL3			HCR_EL2			SCTLR2_EL2	HCRX_EL2	SCTLR2_EL1	Effect of the interrupt mask when executing at			
EA	TMEA	NMEA	TGE	E2H	AMO	NMEA	TMEA	NMEA	EL0	EL1	EL2	EL3
0b0	0b0	0b0	0b0	X	0b0	0b0	0b0	0b0	B	B	C	C
0b0	0b0	0b0	0b0	X	0b0	0b0	0b0	0b1	A	A	C	C
0b0	0b0	0b0	0b0	X	0b0	0b0	0b1	X	A	A	C	C
0b0	0b0	0b0	0b0	X	0b0	0b1	0b0	0b0	B	B	A	C
0b0	0b0	0b0	0b0	X	0b0	0b1	0b0	0b1	A	A	A	C
0b0	0b0	0b0	0b0	X	0b0	0b1	0b1	X	A	A	A	C
0b0	0b0	0b0	0b0	X	0b1	0b0	X	X	A	A	B	C
0b0	0b0	0b0	0b0	X	0b1	0b1	X	X	A	A	A	C
0b0	0b0	0b0	0b1	0b0	X	0b0	X	X	A	n/a	B	C
0b0	0b0	0b0	0b1	X	X	0b1	X	X	A	n/a	A	C
0b0	0b0	0b0	0b1	0b1	X	0b0	X	X	B	n/a	B	C
0b0	0b0	0b1	0b0	X	0b0	0b0	0b0	0b0	B	B	C	A
0b0	0b0	0b1	0b0	X	0b0	0b0	0b0	0b1	A	A	C	A
0b0	0b0	0b1	0b0	X	0b0	0b0	0b1	X	A	A	C	A
0b0	0b0	0b1	0b0	X	0b0	0b1	0b0	0b0	B	B	A	A
0b0	0b0	0b1	0b0	X	0b0	0b1	0b0	0b1	A	A	A	A
0b0	0b0	0b1	0b0	X	0b0	0b1	0b1	X	A	A	A	A
0b0	0b0	0b1	0b0	X	0b1	0b0	X	X	A	A	B	A
0b0	0b0	0b1	0b0	X	0b1	0b1	X	X	A	A	A	A
0b0	0b0	0b1	0b1	0b0	X	0b0	X	X	A	n/a	B	A
0b0	0b0	0b1	0b1	X	X	0b1	X	X	A	n/a	A	A
0b0	0b0	0b1	0b1	0b1	X	0b0	X	X	B	n/a	B	A
0b0	0b1	0b0	0b0	X	X	X	X	X	A	A	A	C
0b0	0b1	0b0	0b1	X	X	X	X	X	A	n/a	A	C
0b0	0b1	0b1	0b0	X	X	X	X	X	A	A	A	A
0b0	0b1	0b1	0b1	X	X	X	X	X	A	n/a	A	A
0b1	X	0b0	0b0	X	X	X	X	X	A	A	A	B
0b1	X	0b0	0b1	X	X	X	X	X	A	n/a	A	B
0b1	X	0b1	0b0	X	X	X	X	X	A	A	A	A
0b1	X	0b1	0b1	X	X	X	X	X	A	n/a	A	A

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D1-24 SError exception masking when EL2 is using AArch32

SCR_EL3		HCR		Effect of the interrupt mask when executing at		
EA	TMEA	TGE	AMO	EL0	EL1	EL2
0b0	0b0	0b0	0b0	B	B	B
0b0	0b0	0b0	0b1	A	A	B
0b0	0b0	0b1	x	A	n/a	B
0b0	0b1	0b0	x	A	A	A
0b0	0b1	0b1	x	A	n/a	A
0b1	x	0b0	x	A	A	A
0b1	x	0b1	x	A	n/a	A

D1.4.6.4.2 Virtual asynchronous exception masking

R_{SNLJH} The following controls determine whether a virtual IRQ interrupt is masked:

PSTATE.I	SCTLR_EL1.NMI	PSTATE.EL	AllIntMask	vIRQ	vIRQ with Superpriority
0	0	xx	x	Not Masked	Not Masked
0	1	EL0	x	Not Masked	Not Masked
0	1	EL1	0	Not Masked	Not Masked
0	1	EL1	1	Masked	Masked
1	0	xx	x	Masked	Masked
1	1	EL0	x	Masked	Not Masked
1	1	EL1	0	Masked	Not Masked
1	1	EL1	1	Masked	Masked

R_{XSPSG} The following controls determine whether a virtual FIQ interrupt is masked:

PSTATE.F	SCTLR_EL1.NMI	PSTATE.EL	AllIntMask	vFIQ	vFIQ with Superpriority
0	0	xx	x	Not Masked	Not Masked
0	1	EL0	x	Not Masked	Not Masked
0	1	EL1	0	Not Masked	Not Masked
0	1	EL1	1	Masked	Masked
1	0	xx	x	Masked	Masked
1	1	EL0	x	Masked	Not Masked
1	1	EL1	0	Masked	Not Masked
1	1	EL1	1	Masked	Masked

R_{GYGBD} Virtual asynchronous exceptions can only be taken from EL0 or EL1 and can only be taken to EL1. If EL2 is not enabled in the current Security state, all types of virtual asynchronous exceptions are always masked. If executing at EL2 or EL3, all types of virtual asynchronous exceptions are always masked.

R_{MMYBS} The following controls determine whether a virtual SError interrupt is masked:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSTATE.A	PSTATE.EL	vSError
0	xx	Not Masked
1	xx	Masked

I_{BNCCH} The ability to execute at EL0 with virtual interrupts taken to EL1 masked is required by some user level driver code.

R_{WVZVM} The R_{BKHXL} table uses the following terms:

Term	Meaning
SCR_EL3	The Effective value of a field in SCR_EL3.
EEL2	If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1.
NS	If FEAT_RME is implemented, then if Secure state is not implemented, the Effective value of SCR_EL3.NS is 1.
EA IRQ FIQ	If EL3 is not implemented, the Effective value of these fields is 0.
HCR	If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.
E2H	If EL2 is using AArch32, the Effective value of HCR.E2H is 0.
TGE	If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.
AMO IMO FMO	If EL2 is not implemented, the Effective value of these fields is 0.
B	When the interrupt is pending, it might be subject to masking, as defined in R_{SNLJH}, R_{XSPSG}, and R_{MMYBS}. If the interrupt is masked, it is not taken. If the interrupt is not masked, it is taken.
C	When the interrupt is pending, it is not taken, regardless of the value of the PSTATE.{A, I, F, ALLINT, SP, PM} interrupt masks.
n/a	Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.

R_{BKHXL} The following table describes the masking of virtual interrupts when the highest implemented Exception level is using AArch64:

SCR_EL3			HCR			Effect of the interrupt mask when executing at:			
EEL2	NS	EA IRQ FIQ	E2H	TGE	AMO IMO FMO	EL0	EL1	EL2	EL3
0	0	x	x	x	x	C	C	n/a	C
1	0	x	x	0	0	C	C	C	C
1	0	x	x	0	1	B	B	C	C
1	0	x	x	1	x	C	n/a	C	C
x	1	x	x	0	0	C	C	C	C
x	1	x	x	0	1	B	B	C	C
x	1	x	x	1	x	C	n/a	C	C

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.6.4.3 Delegated SError exception masking

R_{ZZXBJ} The R_{LPCYB} table uses the following terms:

Term	Meaning
HCR_EL2	The Effective value of a field in HCR_EL2.
TGE	If EL2 is not implemented, or EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.TGE is 0.
E2H	If FEAT_VHE is implemented and FEAT_E2H0 is not implemented, then the effective value of HCR_EL2.E2H is 1. Otherwise, if FEAT_VHE is not implemented, or EL2 is not implemented, or EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.E2H is 0.
AMO	If EL2 is not implemented, or EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.AMO is 0.
HCRX_EL2	The Effective value of a field in HCRX_EL2.
TMEA	If FEAT_DoubleFault2 is not implemented, EL2 is not implemented, EL2 is disabled in the current Security state, or the Effective value of SCR_EL3.HXEn is 0, then the Effective value of HCRX_EL2.TMEA is 0.
A	When the exception is pending, it is taken regardless of the value of the PSTATE.A mask. However, the target Exception level of the exception might depend on the value of PSTATE.A. See R_{VRBFW}.
B	When the exception is pending, it might be subject to masking. If PSTATE.A is 1, then the exception is masked and is not taken. If PSTATE.A is 0, then the exception is not masked and is taken.
C	The error exception is not taken and remains pending, regardless of the value of the PSTATE.A mask.
n/a	Not applicable. The Exception level is not present in this configuration.

R_{LPCYB} The following table describes the masking of delegated SError exceptions.
All of the following apply to this table:

- SCR_EL3.{NS, EEL2} are not shown because R_{ZZXBJ} describes Effective values involving Security states.
- Either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Table D1-31 Delegated SError exception masking in an implementation that includes FEAT_E3DSE

HCR_EL2			HCRX_EL2	Effect of the interrupt mask when executing at:		
TGE	E2H	AMO	TMEA	EL0	EL1	EL2
0b0	X	0b0	0b0	B	B	C
0b0	X	0b0	0b1	A	A	C
0b0	X	0b1	X	A	A	B
0b1	0b0	X	X	A	n/a	B
0b1	0b1	X	X	B	n/a	B

D1.4.6.5 Prioritization of asynchronous exceptions

R_{BKHSW} The virtual SError exception injected using HCR_EL2.VSE has priority over the delegated SError exception injected using SCR_EL3.DSE. Otherwise, the prioritization of physical asynchronous exceptions and virtual asynchronous exceptions is IMPLEMENTATION DEFINED.

R_{RBZYL} For all of the following Context synchronization events, if an asynchronous exception is pending before the Context synchronization event, and the asynchronous exception remains pending and is not masked after the Context synchronization event, then the asynchronous exception is taken before the first instruction after the Context synchronization event:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Execution of an ISB instruction.
- If FEAT_ExS is not implemented, exception entry.
- If FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is 1, exception entry.
- If FEAT_ExS is not implemented, exception return.
- If FEAT_ExS is implemented and the appropriate SCTLR_ELx.EOS bit is 1, exception exit.
- Exit from Debug state.

If the first instruction after the Context synchronization event generates a synchronous exception, the architecture does not define whether the PE takes the asynchronous exception or the synchronous exception first.

I_{RDBWJ} A RAS error synchronization event defines additional requirements for taking an SError exception. See Error synchronization event.

R_{QVRHH} Other than the behaviors described in R_{RBZYL}, an unmasked, pending asynchronous exception must be taken in finite time.

R_{ZQPLL} If an unmasked asynchronous exception was pending but is changed to not pending before it is taken, it is CONSTRAINED UNPREDICTABLE whether or not the asynchronous exception is taken. If the asynchronous exception is taken, it is taken before the first Context synchronization event after the asynchronous exception was changed to be not pending.

D1.4.6.6 Taking an asynchronous exception during a multi-access load or store

R_{ZBFSL} If in AArch64 state, asynchronous exceptions can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.

In this situation, the behavior is consistent with the requirements described in R_{TNVSL} in Definition of a precise exception and imprecise exception.

D1.4.6.7 Taking an asynchronous exception during an SVE instruction

R_{TFLTX} Permitting SVE instructions to be interrupted by asynchronous exceptions is IMPLEMENTATION DEFINED.

R_{WFMZK} When returning from an asynchronous exception, an interrupted SVE instruction is restarted and cannot resume at the point the instruction was interrupted.

I_{QRQJP} For Data Aborts taken asynchronously, see Definition of a precise exception and imprecise exception.

D1.4.7 UNDEFINED instructions

R_{ZLJDF} An instruction which is UNDEFINED generates a synchronous exception for that instruction unless there is a higher priority exception generated for that instruction.

R_{SXYZB} The Exception level that the synchronous Undefined Instruction exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

D1.4.8 Configurable instruction controls

R_{QMWPZ} *Configurable instruction controls* are control bits held in System registers that determine whether attempting to execute an instruction generates a synchronous exception at the point in the instruction stream of that instruction, and the instruction is not executed.

I_{ZSSHJ} Configurable instruction controls might be referred to by any of the following names:

- Instruction enables.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Instruction disables.
- Trap controls.

The definitions of each type overlap, and in some cases is historical. Describing a register control field as an instruction enable, an instruction disable, or a trap control, gives no indication of how an exception that is generated as a consequence of the value of that field is handled or reported. Each configurable instruction control defines how the exception that is generated as a consequence of the configurable instruction control is handled or reported.

R_{VWWGJ} An exception can only be generated as a result of a configurable instruction control if all the following apply:

- The instruction generating the exception does not also generate a higher priority exception.
- The instruction generating the exception is not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed.

R_{JTXTF} It is UNPREDICTABLE / CONSTRAINED UNPREDICTABLE whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed, with all of the following constraints:

- If the instruction description explicitly states that the configurable instruction control is applied with higher priority than the CONSTRAINED UNPREDICTABLE behavior, then the configurable instruction control generates an exception.
- The CONSTRAINED UNPREDICTABLE behaviors cannot lead to any behavior that is prohibited by the general definition of UNPREDICTABLE.

I_{BJBMF} UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of configurable instruction controls, but the architecture does not require them to do so.

Note: Many CONSTRAINED UNPREDICTABLE behaviors for instructions include an allowance that the CONSTRAINED UNPREDICTABLE instruction behaves the same way as a closely related instruction that is not CONSTRAINED UNPREDICTABLE. In those cases, the instruction enable, disable, or trap control that causes an exception on the closely related instruction will cause the same exception on the CONSTRAINED UNPREDICTABLE instruction.

I_{TFGPW} An implementation might provide more controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

R_{QQLHG} When a configurable instruction control causes an exception, the exception is taken and the instruction is not executed, and therefore all the following are true:

- The preferred exception return address of the exception is the instruction that generates the exception.
- There are no changes to the registers accessed by the instruction, including as a result of side-effects of a register access.

R_{SZCKN} When a configurable instruction control causes a conditional instruction to generate an exception in AArch32 state, it is IMPLEMENTATION DEFINED whether the exception applies to conditional AArch32 instructions that fail their condition code check.

D1.4.8.1 EL0 and EL1 configurable instruction controls

I_{TJPQP} The following EL0 and EL1 System registers contain configurable instruction controls:

Register name	Register description
AMUSERENR_EL0	Activity Monitors User Enable Register
CPACR_EL1	Architectural Feature Access Control Register
MDSCR_EL1	Monitor System Debug Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
SCTLR_EL1	System Control Register (EL1)
TCR_EL1	Translation Control Register (EL1)

R_{KYYYZ} An exception caused by configurable instruction controls in EL1 can be taken from either AArch64 state or AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.8.2 EL2 configurable instruction controls

IFFLBF The following EL2 System registers contain configurable instruction controls:

Register name	Register description
CPTR_EL2	Architectural Feature Trap Register, EL2
HAFGRTR_EL2	Hypervisor Activity Monitors Fine-Grained Read Trap Register
HCR_EL2	Hypervisor Configuration Register
HCRX_EL2	Extended Hypervisor Configuration Register
HDFGRTR_EL2	Hypervisor Debug Fine-Grained Read Trap Register
HDFGRTR2_EL2	Hypervisor Debug Fine-Grained Read Trap Register 2
HDFGWTR_EL2	Hypervisor Debug Fine-Grained Write Trap Register
HDFGWTR2_EL2	Hypervisor Debug Fine-Grained Write Trap Register 2
HFGITR_EL2	Hypervisor Fine-Grained Instruction Trap Register
HFGITR2_EL2	Hypervisor Fine-Grained Instruction Trap Register 2
HFGRTR_EL2	Hypervisor Fine-Grained Read Trap Register
HFGRTR2_EL2	Hypervisor Fine-Grained Read Trap Register 2
HFGWTR_EL2	Hypervisor Fine-Grained Write Trap Register
HFGWTR2_EL2	Hypervisor Fine-Grained Write Trap Register 2
HSTR_EL2	Hypervisor System Trap Register
MDCR_EL2	Monitor Debug Configuration Register, EL2
SCTLR_EL2	System Control Register, EL2
TCR_EL2	Translation Control Register, EL2

RJZJRG An exception caused by configurable instruction controls in EL2 can be taken from either AArch64 state or AArch32 state.

RSNLNY EL2 configurable instruction controls are ignored in Secure state if Secure EL2 is not implemented or not enabled.

D1.4.8.3 EL3 configurable instruction controls

RLTKXZ The following EL3 System registers contain configurable instruction controls:

Register name	Register description
CPTR_EL3	Architectural Feature Trap Register, EL3
FGWTE3_EL3	Fine-Grained Write Traps EL3
MDCR_EL3	Monitor Debug Configuration Register, EL3
SCTLR_EL3	System Control Register, EL3
SCR_EL3	Secure Configuration Register
TCR_EL3	Translation Control Register, EL3

RNMCCJ An exception caused by configurable instruction controls in EL3 can be taken from either AArch64 state or AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4.9 Exception generating instructions

IYRYNQ The exception generating instructions are commonly called system calls and refer to the instructions that generate any of the following synchronous exception types:

- Supervisor Call exception.
- Hypervisor Call exception.
- Secure Monitor Call exception.

RBRCQB A Supervisor Call is generated by executing an `SVC` instruction.

RTNSNV A Supervisor Call exception is taken to the following Exception levels:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

INCTDB A Supervisor Call enables software executing at EL0 to make a call to an operating system or other software executing at EL1.

RLLLGJ If EL2 is implemented and the `HVC` instruction is not UNDEFINED, a Hypervisor Call is generated by executing an `HVC` instruction.

RRWKMY A Hypervisor Call exception is taken to the following Exception levels:

- If EL2 is implemented and enabled in the current Security state, when taken from EL1, the exception is taken to EL2.
- When taken from EL2, the exception is taken to EL2.
- When taken from EL3, the exception is taken to EL3.

RZVTPR If any of the following is true, the `HVC` instruction is UNDEFINED:

- The PE is executing at EL0.
- If EL2 is not enabled in the current Security state, and the PE is executing at EL1.
- The Effective value of SCR_EL3.HCE is 0.
- EL2 is not implemented.

RNDGTC If EL3 is implemented and the `SMC` instruction is not UNDEFINED or trapped, a Secure Monitor Call is generated by executing an `SMC` instruction. A Secure Monitor Call is a synchronous exception that is taken to EL3.

RHMXQS If any of the following are true, the `SMC` instruction is UNDEFINED:

- The PE is executing at EL0.
- EL3 is not implemented and the `SMC` instruction is not trapped to EL2 by HCR_EL2.TSC.
- SCR_EL3.SMD is 1 and the `SMC` instruction is not trapped to EL2 by HCR_EL2.TSC.

D1.4.10 Program Counter and stack pointer alignment

D1.4.10.1 PC alignment checking

RQCCWG If bits [1:0] of the PC are not `0b00`, there is a misaligned PC.

RSJLDM The execution of an instruction with a misaligned PC generates a synchronous PC Alignment exception on that instruction.

RJKZZX A PC Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If HCR_EL2.TGE is 1, the exception is taken to EL2.

- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

R_{MVTQT} When a PC Alignment Fault exception is taken to an Exception level, ELx, using AArch64, the ELR_ELx and the FAR_ELx both hold the entire PC in its misaligned form.

I_{VFXPV} A misalignment of the PC is an indication of a serious error, for example software corruption of an address.

D1.4.10.2 SP alignment checking

R_{RDMXG} When the SP is used as the base address of a calculation, regardless of any offset applied by the instruction, if bits [3:0] of the SP are not `0b0000`, there is a misaligned SP.

R_{TFVSM} If SP alignment checking is enabled, then the execution of a load or store using the SP with a misaligned SP generates a synchronous SP Alignment exception on that load or store.

R_{JQXVP} PRFM instructions that use the SP do not perform stack alignment checking.

R_{STDYJ} The following bits enable SP alignment checking at each Exception level when that Exception level is using AArch64.

- If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, SCTLR_EL2.SA0 controls EL0. Otherwise, SCTLR_EL1.SA0 controls EL0.
- SCTLR_EL1.SA controls EL1.
- SCTLR_EL2.SA controls EL2.
- SCTLR_EL3.SA controls EL3.

R_{NCGYQ} An SP Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  — If HCR_EL2.TGE is 0, the exception is taken to EL1
  — If HCR_EL2.TGE is 1, the exception it taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

R_{DLDVL} If SP alignment checking is enabled and an SVE predicated load or store instruction with any *Active elements* uses the current SP as the base address, then the SP register is checked for 16-byte alignment.

R_{FNCJX} If SP alignment checking is enabled and an SVE predicated load or store instruction with no *Active elements* uses the current SP as the base address, then it is CONSTRAINED UNPREDICTABLE whether the SP register is checked for 16-byte alignment.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.5 Process state, PSTATE

R_{SPCZF} Process state, or PSTATE, is an abstraction of process state information.

I_{LPKMT} All Arm instruction sets provide instructions that operate on elements of PSTATE.

I_{KXKMY} PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

I_{ZHGST} PSTATE is defined in pseudocode as the PSTATE structure.

D1.5.1 PSTATE fields that are meaningful in AArch64 state

I_{LLLXL} PSTATE fields that are meaningful in AArch64 state are grouped into the following categories:

- Condition flags.
- Execution state controls.
- Exception mask bits.
- Access control bits.
- Timing control bits.
- Speculation control bits.

R_{PCDTX} The following PSTATE bits are meaningful in AArch64 state:

PSTATE field name	PSTATE field group	Required implemented feature	Value when a Warm reset is asserted	Additional details
N, Negative condition flag	Condition flags	-	-	-
Z, Zero condition flag	Condition flags	-	-	-
C, Carry condition flag	Condition flags	-	-	-
V, Overflow condition flag	Condition flags	-	-	-
SS, Software Step bit	Execution state controls	-	0	-
IL, Illegal Execution state bit	Execution state controls	-	0	-
nRW, Current Execution state bit	Execution state controls	-	0	If the current Execution state is AArch64, the PSTATE.nRW bit is 0.
EL, Current Exception level field	Execution state controls	-	When a Warm reset is asserted into an Exception level using AArch64, the PSTATE.EL field holds the encoding for the highest implemented Exception level.	-
SP, Stack pointer register selection bit	Execution state controls	-	1	-
D, Debug exception mask bit	Exception mask bits	-	1	-
A, I, F Asynchronous exception mask bits	Exception mask bits	-	1	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSTATE field name	PSTATE field group	Required implemented feature	Value when a Warm reset is asserted	Additional details
ALLINT, All interrupt mask bit	Exception mask bits	FEAT_NMI	-	-
PAN, Privileged Access Never state bit	Access control bits	FEAT_PAN	-	-
UAO, User Access Override bit	Access control bits	FEAT_UAO	-	-
TCO, Tag Check Override bit	Access control bits	FEAT_MTE	-	-
BTYPE, Branch target identification bit	Access control bits	FEAT_BTI	-	-
DIT, Data Independent Timing bit	Timing control bits	FEAT_DIT	0	-
SSBS, Speculative Store Bypass Safe bit	Speculation control bits	FEAT_SSBS	IMPLEMENTATION DEFINED	-
PM, PMU exception mask	Exception mask bits	FEAT_EBEP	-	-
EXLOCK, GCS exception state lock	Access control bits	FEAT_GCS	0	-
SM, Enables Streaming SVE mode	Execution state controls	FEAT_SME	0	-
ZA, Enables SME ZA storage	Access control bits	FEAT_SME	0	-
PACM. PACM instruction executed	Execution state controls	FEAT_PAuth_LR	0	-
UINJ, Inject Undefined Instruction exception	Execution state controls	FEAT_UINJ	0	-

D1.5.1.1 Accessing PSTATE fields

R_{LYGBS} In AArch64 state, PSTATE fields are accessed using Special-purpose registers that are directly read using the MRS (register) instructions, and directly written using the MSR (register) instructions.

R_{DGPCL} If in AArch64 state, the following Special-purpose registers can access the PSTATE fields that hold AArch64 state:

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
CurrentEL	EL
SPSel	SP
PAN	PAN
UAO	UAO
DIT	DIT
SSBS	SSBS
TCO	TCO
ALLINT	ALLINT
PM	PM
SVCR	SM, ZA

All other PSTATE fields do not have direct read and write access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{BRSMZ} Software can use the MSR (immediate) instructions to directly write to PSTATE.{D, A, I, F, SP, PAN, UAO, DIT, SSBS, TCO, ALLINT, PM, SM, ZA}.

R_{LDXKJ} The following PSTATE fields can be accessed at EL0:

- PSTATE.{N, Z, C, V, SSBS, DIT, TCO, SM, ZA}.
- If SCTLR_EL1.UMA is 1, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, PSTATE.{D, A, I, F}.
- PSTATE.{SM, ZA} access instructions can be executed at EL0 or higher.
- All other PSTATE access instructions can be executed at EL1 or higher and are UNDEFINED at EL0.

R_{HXYGT} Writes to the PSTATE fields have side-effects on various aspects of the PE operation. For side-effects caused by writes to a PSTATE field, all of the following are true:

- The side-effect is guaranteed not to be visible to earlier instructions in the Execution stream.
- The side-effect is guaranteed to be visible to later instructions in the Execution stream.

Other side-effects might occur but are not guaranteed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.6 Resets and power domains

I_{DQXXZ} The PE logic is split into a Debug power domain and a Core power domain. Cold and Warm resets, reset elements in the Core power domain. An External debug reset can reset the Debug power domain. Other resets might be implemented.

D1.6.1 Power domains and reset domains

R_{JMTZY} The architecture defines the following power domains:

- The Core power domain.
- The Debug power domain.
- The system counter power domain.
- If trace is implemented, a power domain for the Trace Unit.
- If FEAT_RAS is implemented, there may be one or more IMPLEMENTATION DEFINED error recovery power domains. The relationship between these power domains and other power domains is IMPLEMENTATION DEFINED. For error records associated with the PE, this is the same as the Core power domain.
- If FEAT_AMUv1 is implemented, the AMU power domain.
- For *MPAM Memory System Components* (MSC), a power domain for each MSC.

I_{QBPTS} The power domains are described as logical because the architecture does not require two physical power domains.

R_{PDMKB} It is IMPLEMENTATION DEFINED whether each power domain powers up and powers down separately or together.

R_{ZDVQQ} The Core power domain contains:

- Non-debug registers and logic.
- Self-hosted debug registers and logic.
- Shared debug registers and logic.
- Some external debug registers and logic.

R_{GTVLN} The Debug power domain contains:

- The interface between the PE and the external debugger.
- Some external debug registers.

R_{MLXWM} If an external debugger is connected to the PE, the Debug power domain is powered up.

I_{KTSWK} For more information on access permissions for the external debug interface registers, see Access permissions for the External debug interface registers or the individual register descriptions.

D1.6.2 Reset types

I_{HDQLB} The architecture defines the following resets:

- Warm reset.
- Cold reset.
- External Debug reset.
- Trace unit reset. See Resetting the trace unit.
- Timer reset.
- If FEAT_RAS is implemented, there may be one or more IMPLEMENTATION DEFINED error recovery resets. The relationship between this reset and other resets is IMPLEMENTATION DEFINED. For error records associated with the PE, the error recovery reset is the same as Warm reset.
- If the FEAT_AMUv1 is implemented, AMU reset.
- For *MPAM Memory System Components* (MSC), a reset for each MSC.

I_{FMTCY} Other resets are IMPLEMENTATION DEFINED and can be mapped onto the architecturally-defined resets.

R_{PXXYQ} Mechanisms to assert resets, other than RMR_ELx, are IMPLEMENTATION DEFINED. One such mechanism is EDPRCR.CWRR. Any reset-asserting mechanism that software can command, including hardware mechanisms directly exposed to software, must only be accessible at the highest Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HLDGD} When a Cold reset is asserted, for each register in the Core power domain, the reset sets the register to its architecturally-defined reset value.

R_{PJXVL} When a Warm reset is asserted, registers and logic in the Core power domain that are affected by the Warm reset are reset to their architecturally-defined reset value. Registers and logic that are only reset by a Cold reset are unaffected by a Warm reset.

R_{NBKHK} When an External Debug reset is asserted, the External Debug reset domain is reset and affected registers are set to their architecturally-defined External debug reset values. The Cold reset domain is unaffected.

R_{RKSZM} When a Cold reset is asserted, a Warm reset is asserted.

I_{VBZBW} Every register field that has storage belongs to exactly one reset domain. Each System register field definition defines the reset domain and the reset value.

I_{MWTXK} If RMR_ELx is implemented, writing 1 to RMR_ELx.RR requests a Warm reset.

R_{SGXSW} Writing 1 to RMR_ELx.RR is only a request for a Warm reset. The reset is not guaranteed to occur unless the following code sequence is executed:

```
; In addition, interrupts and debug requests for this PE should be disabled
; in the system before running this sequence to ensure the WFI suspends execution,
        MOV Wy, #3          ; for AArch64, #2 for AArch32; y is any register
        DSB                 ; ensure all stores etc are complete
        MSR RMR_ELx, Wy     ; request the reset
        ISB                 ; synchronize change to the RMR
Loop
        WFI                 ; enter a quiescent state
        B Loop
```

R_{DDTVZ} It is IMPLEMENTATION DEFINED whether a Warm reset can be asserted without using RMR_ELx.

I_{LWNDC} Arm recommends that a Warm reset can be asserted independently of a Cold reset.

I_{QLZGF} A Warm reset allows debugging across a reset of the PE logic in the Core power domain.

R_{SKGMH} It is IMPLEMENTATION DEFINED whether an External Debug reset and a Cold reset can be asserted independently.

I_{WTJYS} Arm recommends that when separate Core and Debug power domains are implemented, the External Debug reset and Cold reset can be asserted independently.

D1.6.3 Reset behavior

I_{TWYCF} Immediately after a reset, much of the PE state is architecturally UNKNOWN. However, some of the PE state is defined, see the individual register descriptions for more information. The state that is reset to known values is sufficient to permit predictable initial execution at the highest Exception level, such that this execution is then capable of initializing the remaining state of the system where necessary before use.

R_{JYLQV} When a Cold or Warm reset is asserted, all of the following occur:

- The PE enters the highest implemented Exception level.
- The stack pointer for the highest implemented Exception level, SP_ELx, is selected.

R_{ZBHGJ} When a Cold or Warm reset is deasserted, execution starts at an IMPLEMENTATION DEFINED address, anywhere in the *Physical Address* (PA) range. The RVBAR associated with the highest implemented Exception level RVBAR_EL1, RVBAR_EL2, or RVBAR_EL3, holds the address at which the PE starts execution.

I_{QSCPY} When a Cold or Warm reset is deasserted, the IMPLEMENTATION DEFINED address from which the PE starts execution is typically set either by a hardwired configuration of the PE or by configuration input signals.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{RRJLM} When a Warm reset is asserted, all of the following are architecturally UNKNOWN:

- All general-purpose, SVE, SIMD, and floating-point registers.
- The global exclusive monitor for the PE.
- The local exclusive monitor for the PE.
- System and Special-purpose registers with a reset value of architecturally UNKNOWN.

I_{KWHDG} When a Cold or Warm reset is asserted, PSTATE.{SS, IL, nRW, EL, SP, D, A, I, F, DIT, EXLOCK, ZA, SM, PACM, UINJ} fields are set to known values.

R_{BGSHC} When a Cold or Warm reset is asserted, the TLBs and caches are in an IMPLEMENTATION DEFINED state.

R_{XNNNH} When a Cold or Warm reset has been asserted and before the memory management system is enabled, the TLBs, caches or both might need to be invalidated by IMPLEMENTATION DEFINED invalidation sequences.

R_{TZHFZ} When a Cold or Warm reset has been asserted and before Normal memory accesses are permitted to be Cacheable, the TLBs, caches or both might need to be invalidated by IMPLEMENTATION DEFINED invalidation sequences.

R_{KMCGN} If IMPLEMENTATION DEFINED resets are implemented, each IMPLEMENTATION DEFINED reset can treat the cache and TLB state differently.

R_{TJGFQ} If IMPLEMENTATION DEFINED resets are implemented, for each IMPLEMENTATION DEFINED reset, the TLBs, caches, or both might require a different IMPLEMENTATION DEFINED invalidation sequence.

I_{PHDMZ} The IMPLEMENTATION DEFINED invalidation sequence might be a NOP.

D1.6.3.1 External debug access to registers in reset

R_{VFMKQ} If External Debug reset is asserted, for each register that can be accessed by the external debug interface, a register access by the external debug interface has one of the following IMPLEMENTATION DEFINED results:

- The external debug interface can access the register, and the access result is IMPLEMENTATION DEFINED.
- The external debug interface cannot access the register.

I_{QHHMQ} If either a Cold or Warm reset is asserted, accesses of registers or register fields might be direct accesses or indirect side-effects of an access.

R_{YKRBF} If either a Cold or Warm reset is asserted, external debug interface accesses of a register or register field have the following effects:

Access	Register or register field reset behavior	Effect on register or register field
Write	Reset by the reset signal	Set to CONSTRAINED UNPREDICTABLE choice of the reset value or the value written.
Write	Not reset by the reset signal	Set to the value written.
Read	Reset by the reset signal	Returns an UNKNOWN value.
Read	Not reset by the reset signal	Returns the value of the register or register field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.7 Mechanisms for entering a low-power state

I_{RTNZD} The architecture provides the following mechanisms that software can use to indicate that the PE can enter low-power state:

- Wait for Event.
- Wait for Interrupt.

D1.7.1 Wait for Event

I_{ZZQGW} The Wait for Event mechanism behavior depends on the interaction of all of the following:

- The Event Register for the PE. See The Event Register.
- The Wait for Event (WFE) or Wait for Event with Timeout (WFET) instruction. See The Wait for Event and Wait for Event with Timeout instructions.
- WFE wakeup events. See WFE wakeup events in AArch64 state.
- The Send Event instructions, SEV and SEVL, that can cause WFE wakeup events. See The Send Event instructions.

R_{TJTFC} The architecture does not define the exact nature of the low-power state entered by WFE or WFET, except that when a WFE or WFET instruction is executed, memory coherency and architectural state are not lost.

I_{WBMQQ} The Wait for Event mechanism is associated with suspending execution on a PE for power saving, therefore Arm recommends that the Event Register is set infrequently. Software must only use the setting of the Event Register as a hint, and must not assume that any particular message is sent as a result of the setting of the Event Register.

D1.7.1.1 The Event Register

R_{PVKKP} The Event Register is a single bit register for each PE.

I_{TZVVM} The event register is a conceptual register and cannot be explicitly accessed.

I_{PDJMB} If the Event Register is 1, then an event has occurred since the register was last cleared and the event might require some action by the PE.

R_{RJYVC} When the PE executes a WFE or WFET instruction, all of the following apply:

- If the Event Register is 0, then the PE can enter the low-power state.
- If the Event Register is 1, then all of the following apply:
  - The PE does not suspend operation.
  - The Event Register is cleared to 0.
  - The WFE or WFET instruction completes immediately.

R_{DDGLZ} The reset value of the Event Register is UNKNOWN.

R_{XRZRK} The Event Register for a PE is set by any of the following:

- A Send Event instruction, SEV, executed by any PE in the system.
- A Send Event Local instruction, SEVL, executed by the PE.
- An exception return.
- The clearing of the global monitor for the PE.
- An event from a Generic Timer event stream, see Event streams.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

I_{PJZBB} The Event Register is cleared only by a Wait for Event (WFE) instruction or a Wait for Event with Timeout (WFET) instruction.

I_{ZXJPD} Software cannot read or write the value of the Event Register directly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.7.1.2 The Wait for Event and Wait for Event with Timeout instructions

R_{QRXWZ} If FEAT_WFxT is implemented, the Wait for Event with Timeout (WFET) instruction is implemented

R_{CSKPF} If the Event Register is 0, when a WFE or WFET instruction is executed, the PE can suspend execution and enter a low-power state.

R_{GZZQM} If the PE enters a low-power state, the PE remains in the lower-power state until one of the following occurs:

- The PE detects a WFE wakeup event.
- An IMPLEMENTATION DEFINED wake event that is architecturally permitted to occur at any time.
- A reset is asserted.

I_{QGHTM} The architecture permits all of the following:

- The PE is permitted to leave the low-power state for any reason.
- The PE is permitted to treat WFE as a NOP, but this is not recommended for lowest power operation.

R_{SLQVQ} If the PE is in a low-power state, when the PE detects a WFE wakeup event, or earlier if the implementation chooses, the WFE or WFET instruction completes. If the wakeup event sets the Event Register, when execution is restarted, the state of the Event Register is IMPLEMENTATION DEFINED.

I_{DRSJV} Software using the Wait for Event mechanism must tolerate spurious wakeup events, including multiple wakeup events.

D1.7.1.3 Trapping of WFE and WFET

I_{KMHTB} The WFE and WFET instructions are available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 can be configured to trap to a higher Exception level.

R_{JWKTL} If FEAT_TWED is implemented, then all of the following apply:

- A delay before taking a WFE or WFET trap can be configured.
- If a delay before taking a WFE or WFET trap is configured, then the delay does not affect the priority of the traps.

I_{RDYQK} For example, if execution is subject to a trap at EL1 because SCTLR_EL1.nTWE is 0 and HCR_EL2.TWE is 1, the only trap that will be taken is a trap to EL1, even if the delay at EL1 is longer than the delay at EL2.

D1.7.1.4 WFE wakeup events in AArch64 state

I_{RJRLR} In this section, AllIntMask refers to the value described in R_{XZPDT}.

I_{LRPVP} If a WFE or WFET instruction puts a PE into low-power state, a WFE wakeup event received by the PE causes that PE to exit low-power state.

R_{KBMFC} All of the following are WFE wakeup events:

- The execution of an SEV instruction on any PE in a multiprocessor system.
- Any physical SError exception, physical IRQ interrupt, or physical FIQ interrupt received by the PE that is not disabled by EDSCR.INTdis and either of the following is true:
  - The interrupt is marked as A in the tables in Asynchronous exception masking, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
  - The interrupt is marked as B in the tables in Asynchronous exception masking and any of the following are true:
    - The value of the corresponding PSTATE.{A, I, F} mask bit is 0.
    - AllIntMask is 0 and the IRQ or FIQ interrupt has Superpriority.
      See R_{MHWBP} and R_{GFXKY}.
- If all of the following apply, any virtual SError exception, virtual IRQ interrupt, or virtual FIQ interrupt received by the PE:
  - The PE is executing at EL1 or EL0.
  - The interrupt is not disabled by EDSCR.INTdis.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - The interrupt or exception is marked as B in the table in Virtual asynchronous exception masking and any of the following are true:
        - The value of the corresponding PSTATE.{A, I, F} mask bit is 0.
        - AllIntMask is 0 and the vIRQ or vFIQ interrupt has Superpriority. See R_{SNLJH} and R_{XSPSG}.
- If all the following apply, any delegated SError exception received by the PE:
    - The PE is executing at EL2, EL1, or EL0.
    - The exception is not disabled by EDSCR.INTdis.
    - Either of the following is true:
        - The exception is marked as A in the table in Delegated SError exception masking.
        - The exception is marked as B in the table in Delegated SError exception masking and PSTATE.A is 0.
- If halting is allowed, an asynchronous External Debug Request debug event. For the definition of halting, see Halting allowed and halting prohibited and External Debug Request debug event.
- An event sent by the timer event stream for the PE. See Event streams.
- An event caused by the clearing of the global monitor for the PE.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- When FEAT_WFxT is implemented, for `WFET` instructions, a local timeout event caused by the virtual count threshold value, expressed in CNTVCT_EL0, being equaled or exceeded.

I_{FRZZX} Not all of the wakeup events in R_{KBMFC} set the Event Register.

D1.7.1.5 The Send Event instructions

R_{DQRMZ} The Send Event instructions are:

- `SEV`, Send Event: Causes an event to be signaled to all PEs in a multiprocessor system.
- `SEVL`, Send Event Local: Causes an event to be signaled locally without requiring the event to be signaled to other PEs in a multiprocessor system.

I_{FVQBK} An `SEVL` instruction might signal an event to other PEs by an IMPLEMENTATION DEFINED mechanism, but it is not required to do so.

R_{HTDFW} The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED.

R_{RLHHM} When an event is signaled by an `SEV` instruction, the ordering of the event with respect to the completion of memory accesses by instructions before the SEV instruction is not guaranteed.

I_{DVZXD} Arm recommends that software includes a `DSB` instruction before any `SEV` instruction. `DSB` instruction ensures that no instructions, including any SEV instructions, that appear in program order after the `DSB` instruction, can execute until the `DSB` instruction has completed. See Data Synchronization Barrier.

R_{QJDQV} The `SEVL` instruction appears to execute in program order relative to any subsequent WFE or WFET instruction executed on the same PE.

I_{HWHBP} The behavior in R_{QJDQV} applies without the need for any explicit insertion of barrier instructions.

D1.7.2 Wait for Interrupt mechanism

I_{LMJGQ} A PE can use the Wait for Interrupt mechanism to enter a low-power state.

R_{JHZBN} When FEAT_WFxT is implemented, the Wait for Interrupt with Timeout (WFIT) instruction is implemented.

R_{DBXLJ} Software can use the Wait for Interrupt (WFI) and Wait for Interrupt with Timeout (WFIT) instructions to cause the PE to enter a low-power state.

R_{HSPJD} If a Wait for Interrupt (WFI) and Wait for Interrupt with Timeout (WFIT) instructions causes the PE to enter a low-power state, the PE remains in that low-power state until any of the following occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PE detects a WFI wakeup event.
- An IMPLEMENTATION DEFINED wake event that is architecturally permitted to occur at any time.
- A reset is asserted.

I_{MFLXH} The architecture permits all of the following:

- The PE is permitted to leave the low-power state for any reason.
- The PE is permitted to treat WFI as a NOP, but this is not recommended for lowest power operation.

R_{MRVLC} When the PE leaves a low-power state that was entered because of a WFI or WFIT instruction, the WFI or WFIT instruction completes.

R_{HJPJY} Except for all or the following, the architecture does not define the exact nature of the low-power state:

- When a WFI or WFIT instruction is executed, the architecture requires that memory coherency is not lost.
- If the system is configured such that the WFI or WFIT instruction can be completed, then the architecture requires that the architectural state is not lost.

I_{WWYXS} In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wakeup events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

I_{CHMWP} The WFI and WFIT instructions are available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 can be configured to trap to a higher Exception level.

D1.7.2.1 WFI wakeup events

I_{CYKVZ} In this section, AllIntMask refers to the value described in R_{XZPDT}.

R_{VRLPB} All of the following are WFI wakeup events:

- Any physical SError exception, physical IRQ interrupt, or physical FIQ interrupt received by the PE that is marked as A or B in the tables in Asynchronous exception masking, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- If the PE is executing at EL1 or EL0, any virtual SError exception, virtual IRQ interrupt, or virtual FIQ interrupt received by the PE, that is marked as B in the tables in Virtual asynchronous exception masking, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- If the PE is executing at EL2, EL1, or EL0, any delegated SError exception received by the PE that is marked as A or B in the table in Delegated SError exception masking, regardless of the value of PSTATE.A.
- If halting is allowed, an asynchronous External Debug Request debug event. For the definition of halting, see Halting allowed and halting prohibited and External Debug Request debug event.
- An event sent by an IMPLEMENTATION DEFINED mechanism.
- When FEAT_WFxT is implemented, for `WFIT` instructions, a local timeout event caused by the virtual count threshold value, expressed in CNTVCT_EL0, being equaled or exceeded.

R_{NLVGY} WFI wakeup events are never disabled by EDSCR.INTdis, and are never masked by the PSTATE.{A, I, F} mask bits, or by the state of AllIntMask. If wakeup is invoked by an interrupt that is disabled or masked, the interrupt is not taken.

I_{VCDGW} Because debug events are WFI wakeup events, Arm recommends that Wait for Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures that the intervention of debug while waiting does not significantly change the function of the program being debugged.

I_{FXRXC} If the PE is in WFx state when a PMU Capture event occurs, it must wake-up to service the request. Arm recommends the Capture event is not treated as an IMPLEMENTATION DEFINED WFx wake-up event and the PE returns to standby after. See PMU snapshots.

I_{ZQWXZ} The architecture does not require the WFI mechanism to drain down any pending memory activity before suspending execution, and software must not rely on the WFI mechanism operating in this way.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.7.3 Pending Profiling exceptions

I_{WNMSV} A pending asynchronous Profiling exception is not required to be a WFE wakeup event or WFI wakeup event.

Arm recommends that a pending unmasked asynchronous Profiling exception is treated as an IMPLEMENTATION DEFINED wakeup event, to allow timely processing of the profiling event. For example, if other telemetry data needs to be collected by software from the system. However, if a Profiling exception becomes pending as the PE enters a low-power WFx state, then the PE might not have registered the pending Profiling exception before entering the WFx state.

A similar condition applies when using PMU, SPE, or TRBE interrupts, because the PE has to have asserted the interrupt request to the interrupt controller before entering the WFx state for it to be a wakeup event. This is due to IMPLEMENTATION SPECIFIC timings that determine whether a profiling event causes a WFx wakeup or prevents the PE from entering the WFx state.

I_{VPTQY} In I_{WNMSV}, pending means that, if the Profiling exception were unmasked, a Profiling exception would be taken by a simple sequential execution of the program before the execution of the next instruction.

D1.7.4 Using WFI to indicate an idle state on bus interfaces

R_{TZPSV} When all of the following are true, the IMPLEMENTATION DEFINED mechanism for entering powerdown state can be completed:

- The IMPLEMENTATION DEFINED mechanism forces quiescence on a PE.
- The IMPLEMENTATION DEFINED mechanism prevents any possible WFI wakeup events.

R_{VWDWJ} The mechanism for entering powerdown state is IMPLEMENTATION DEFINED.

I_{FFGGG} Although the mechanism for entering powerdown state is IMPLEMENTATION DEFINED, the WFI and WFE mechanisms are often used to enter powerdown state.

R_{RMBKR} If a WFI powerdown mechanism is implemented, when a WFI or WFIT instruction is executed, the PE completes all current operations and any associated bus activity also completes. When all current PE operations and associated bus activity are complete, the PE can signal to an external power controller that there is no ongoing bus activity.

R_{CLTZY} After a WFI instruction is executed and all currently executing instructions and related bus activity has completed, the PE is waiting for an interrupt and is in *idle state*. If in idle state, the PE can process memory-mapped or external debug interface accesses to debug registers.

I_{CYBLG} The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

R_{DHHKY} If OSDLR_EL1.DLK is 1, the PE does not signal the idle state to the control logic unless it can also guarantee that the debug interface is idle. For more information about the OS Double Lock, see Debug behavior when the OS Double Lock is locked.

I_{WRZGC} In a PE that implements separate Core and Debug power domains, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see Power domains and debug.

I_{RFLDH} The exact nature of the debug interface is IMPLEMENTATION DEFINED, but the use of Wait for Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it suitable as the preferred powerdown entry mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.8 Self-hosted debug

I_{MJGPY} The architecture supports both of the following:

Self-hosted debug

The PE itself hosts a debugger. The debugger programs the PE to generate debug exceptions. Debug exceptions are accommodated in the Arm Exception model.

External debug

The PE is controlled by an external debugger. The debugger programs the PE to generate debug events, that cause the PE to enter Debug state. In Debug state, the PE is halted.

D1.8.1 Debug exceptions

I_{PWPHY} If a debugger has programmed the PE to generate Debug exceptions, the Debug exceptions occur during normal program flow.

I_{DSWBH} For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions.

I_{BXVCF} The debug exceptions are all of the following:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

I_{VYKHK} The PE can only generate a particular debug exception if all of the following are true:

- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
- A debugger has enabled that particular debug exception. All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See The debug exception enable controls.

I_{GRWSG} Breakpoints and watchpoints can cause entry to Debug state instead of causing debug exceptions. See About External Debug.

D1.8.2 The PSTATE debug mask bit, D

R_{DYRLS} The debug exception mask bit, PSTATE.D, can mask Watchpoint, Breakpoint, and Software Step exceptions if the target Exception level is the current Exception level.

R_{RDYDM} If in AArch64 state and the target Exception level of a debug exception is lower than the current Exception level, the debug exception is masked.

R_{JWZFQ} If the target Exception level is higher than the current Exception level, debug exceptions are not masked by PSTATE.D.

R_{KCHGM} If PSTATE.D is 1 and the PE is already handling an exception in AArch64 state, the Watchpoint, Breakpoint, and Software Step exceptions are masked.

I_{LVYYH} If an exception is already being handled, masking prevents software from generating another instance of an exception and prevents recursive entry into the exception handler and loss of return state.

R_{NCHXS} Masking debug interrupts with PSTATE.D prevents the generation of new debug exceptions, therefore, any masked debug exceptions are not taken if PSTATE.D is later set to 0.

I_{VSTYD} The behavior described in R_{NCHXS} differs from the behavior for interrupts, where the PSTATE.{A, I, F} mask prevents the interrupt from being taken, but instead the interrupt remains pending.

I_{KWFMZ} When an exception is taken, PSTATE.D is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.9 Event monitors

IYVDTB The architecture supports the following non-invasive architectural components that allow for event monitoring:

Performance Monitors

The Performance Monitors have a wide feature set, flexible selection of counted events, and are read/write in operation.

Activity Monitors

The Activity Monitors have a narrow feature set, limited selection of counted events, and are read-only in operation.

D1.9.1 The Performance Monitors Extension

IZSPVV The System registers provide access to a Performance Monitoring Unit (PMU), defined as the OPTIONAL Performance Monitors Extension to the architecture, a non-invasive debug resource that provides information about the operation of the PE.

The PMU provides:

- A 64-bit cycle counter.
- An IMPLEMENTATION DEFINED number of event counters. If FEAT_PMUv3p5 is implemented, the event counters are 64-bit unsigned counters, otherwise the event counters are 32-bit event counters.Each event counter can be configured to count occurrences of a specified event. The events that can be counted are:
  - Architectural and microarchitectural events that are likely to be consistent across many microarchitectures. The PMU architecture uses event numbers to identify an event, and the PMU specification defines which event number must be used for each of these architectural and microarchitectural events.
  - IMPLEMENTATION SPECIFIC events. The PMU specification reserves event numbers for IMPLEMENTATION SPECIFIC events.
- An OPTIONAL 64-bit instructions counter.

For more information, see The Performance Monitors Extension.

D1.9.2 The Activity Monitors Extension

IPCKDC If the OPTIONAL Activity Monitors Extension is implemented, the System registers provide access to controls and counters for the Activity Monitors Unit (AMU).

For more information, see The Activity Monitors Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.10 Interprocessing

R_{FKKJY} Moving between the AArch64 and AArch32 Execution states is called *interprocessing*.

R_{NKBQL} The Execution state can only change on a change of Exception level.

I_{WNJBK} Therefore, Execution state can change only on taking an exception to a higher Exception level or returning from an exception to a lower Exception level. For an exception that is taken to the same Exception level or is returning from the same Exception level, the Execution state cannot change.

R_{GLNVF} When taking an exception to a higher Exception level, the Execution state is one of the following:

- Unchanged.
- Changed from AArch32 state to AArch64 state.

R_{KTCXV} When returning from an exception to a lower Exception level, the Execution state is one of the following:

- Unchanged.
- Changed from AArch64 state to AArch32 state.

D1.10.1 Register mappings between AArch32 state and AArch64 state

I_{XVDJR} Register mappings between AArch32 state and AArch64 state describe the following:

- For exceptions taken from AArch32 state to AArch64 state, where the AArch32 register content is found.
- For exception returns from AArch64 state to AArch32 state, how the AArch32 register content is derived.

R_{PCCLL} The AArch32 state register contents occupy the bottom 32 bits of the AArch64 state registers.

R_{CTBNQ} If in AArch32 state, the upper 32 bits of AArch64 state registers are inaccessible and are ignored.

R_{GMLYF} For bits[63:32] of AArch64 registers that are not mapped to AArch32 registers, the unmapped bits are unchanged by AArch32 state execution.

D1.10.1.1 Mapping of the general-purpose registers between the Execution states

I_{PYKVS} The following table shows the mappings of general-purpose registers between Execution states:

AArch32 register	AArch64 register
R0	X0
R1	X1
R2	X2
R3	X3
R4	X4
R5	X5
R6	X6
R7	X7
R8_usr	X8
R9_usr	X9
R10_usr	X10
R11_usr	X11
R12_usr	X12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch32 register	AArch64 register
SP_usr	X13
LR_usr	X14
SP_hyp	X15
LR_irq	X16
SP_irq	X17
LR_svc	X18
SP_svc	X19
LR_abt	X20
SP_abt	X21
LR_und	X22
SP_und	X23
R8_fiq	X24
R9_fiq	X25
R10_fiq	X26
R11_fiq	X27
R12_fiq	X28
SP_fiq	X29
LR_fiq	X30

I_{JCWRB} For some exceptions, the exception syndrome given in the ESR_ELx identifies one or more register numbers from the issued instruction that generated the exception. If one of these exceptions is taken from an Exception level using AArch32, the register numbers give the AArch64 view of the register.

I_{VSFBC} For example, if an exception is taken from AArch32 Abort mode and the faulting instruction specified R14, the ESR_ELx.ISS field would report this using the EC value `0b10100`, because register X20 provides the AArch64 view of LR_abt, which is the copy of R14 used in Abort mode.

D1.10.1.2 Mapping of the SIMD and floating-point registers between the Execution states

I_{DPNRF} The following table shows the mappings of SIMD and floating-point registers between the Execution states:

AArch32 register	AArch64 register
V0	Q0
V1	Q1
V2	Q2
.	.
.	.
.	.
V15	Q15

R_{TJJVC} The AArch64 state registers V16-V31 are not accessible from AArch32 state.

I_{VTZXL} The mapping between the V, D, and S registers in AArch64 state is not the same as the mapping between the Q, D, and S registers in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{VSTPQ} In AArch64 state, there are:

- 32 128-bit V registers, V0-V31.
- 32 64-bit D registers, D0-D31.
- 32 32-bit S registers, S0-S31.

R_{WJGSQ} A smaller register occupies the least significant bytes of the corresponding larger register.

I_{FFHKC} The following graphic shows the mapping of V, D, and S registers in AArch64 state:

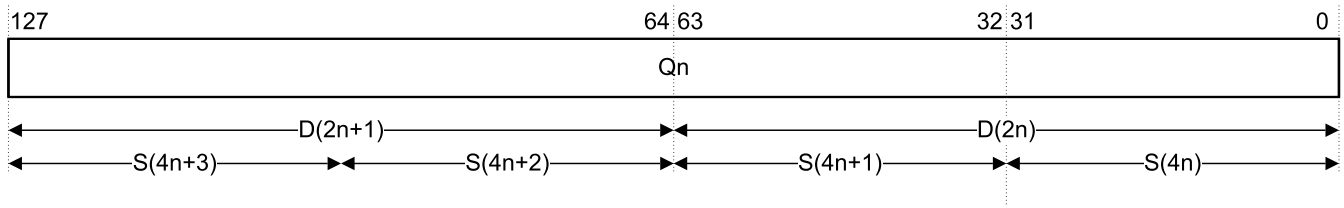

R_{DHVZF} In AArch32 state, there are:

- 16 128-bit Q registers, Q0-Q15.
- 32 64-bit D registers, D0-D31.
- 32 32-bit S registers, S0-S31.

R_{BPYPZ} Smaller registers are packed into larger registers.

I_{YHYWN} The following graphic shows the mapping of Q, D and S registers in AArch32 state:

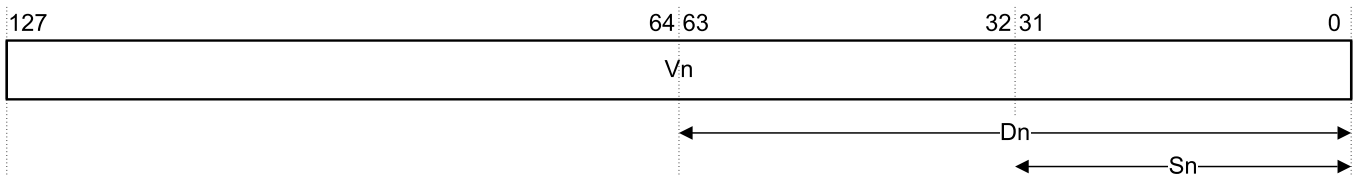

R_{GXGJK} In AArch32 state:

- There are no S registers that correspond to Q8-Q15.
- D16-D31 pack into Q8-Q15.

I_{MKXYB} If software executing in AArch64 state interprets D or S registers from AArch32 state, the software must unpack the D or S registers from the V registers before it uses them.

D1.10.1.3 Mapping of the System registers between the Execution states

I_{MPMNZ} For a full list of mappings of writable AArch64 System registers to the AArch32 System registers, see Table K14-5.

R_{BQSMK} The relationship between the AArch64 System registers and the AArch32 System registers is architecturally defined.

R_{LPDQL} Modifications made to AArch32 System registers affect only the parts of the AArch64 state registers that are mapped to the AArch32 System registers.

I_{HTCVH} Supervisory code, such as a hypervisor, executing in AArch64 state, can save, restore, and interpret the System registers belonging to a lower Exception level that is using AArch32.

I_{LYHXS} In some cases, two AArch32 System registers are packed into a single AArch64 System register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{QXNZN} If EL3 is implemented and is using AArch32, some System registers are banked between Secure and Non-secure states. In this type of banking, there is an instance of the register in Secure state and another instance of the register in Non-secure state.

R_{TVZJS} If any of the following are true, banking between Secure and Non-secure states is not supported:

- EL3 is not implemented.
- EL3 is implemented and is using AArch64.

I_{JRZXQ} If EL3 is implemented and is using AArch64, the same, non-banked, registers are accessed in the following state:

- Secure EL1 with EL1 using AArch32.
- Non-secure EL1 with EL1 using AArch32.

I_{HSRHJ} If EL3 is implemented and is using AArch64, it is not possible to architecturally determine whether an AArch64 register is mapped onto the Secure or the Non-secure instance of corresponding AArch32 register.

R_{MCZCY} If EL3 is implemented and is using AArch64, the interrupt asserted by the AArch64 CNTP_* timer is the same interrupt that is asserted by the Non-secure AArch32 CNTP_* timer when EL3 is using AArch32.

I_{JSZYM} Although not required, Arm expects that implementations map many of the AArch64 registers for use by EL3 to the Secure instances of the banked AArch32 registers, and map many of the AArch64 registers for use by EL1 to the Non-secure instances of the banked AArch32 registers. However, if EL2 and EL3 are implemented and both support use of AArch32, this mapping is not possible for the following registers:

- IFAR - This is because when EL3 is using AArch32, HIFAR is an alias of the Secure IFAR.
- DFAR - This is because when EL3 is using AArch32, HDFAR is an alias of the Secure DFAR.

I_{BYYWR} There are some AArch32 System registers that are only used in AArch32 state and do not have an equivalent AArch64 System register. However, there are AArch64 registers that can access, from a higher Exception level, the AArch32 registers that do not have an AArch64 state equivalent.

I_{VGXRB} For a full list of AArch64 registers that access AArch32 System registers with no AArch64 equivalent, see Table K14-6.

I_{WRKYQ} For a full list AArch64 registers that allow access to the AArch32 ID registers from AArch64 state, see Table K14-7.

D1.10.1.4 State of the general-purpose registers on taking an exception to AArch64 state

D1.10.1.4.1 If the general-purpose register was accessible from AArch32 state

R_{WWJWP} When an exception is taken from AArch32 state to AArch64 state and the general- purpose register was accessible from AArch32 state, the upper 32 bits have one of the following IMPLEMENTATION DEFINED values and might vary dynamically within an implementation:

- The upper 32 bits retain the value that the same architectural register held before any AArch32 execution.
- The upper 32 bits are set to zero.

The IMPLEMENTATION DEFINED behavior applies regardless of whether any execution occurred at the Exception level that was using AArch32.

I_{TKQYW} For example, the IMPLEMENTATION DEFINED behavior described in R_{WWJWP} applies even if AArch32 state was entered by an exception return from AArch64 state, and another exception was immediately taken to AArch64 state without any instruction execution in AArch32 state.

I_{QFBYM} When an exception is taken from AArch32 state to AArch64 state, software must regard the value of the upper 32 bits as a CONSTRAINED UNPREDICTABLE choice between the two values described in R_{WWJWP}.

D1.10.1.4.2 If the general-purpose register was not accessible from AArch32 state

R_{RHRQX} If all of the following apply, when an exception is taken from AArch32 state to AArch64 state, the X15 register is treated as if it is accessible and therefore the upper 32 bits of X15 might either be set to zero or retain their previous value:

- The target Exception level is EL3.
- EL2 is not implemented or EL1 is in Secure state.
- SCR_EL3.RW is 0.

Otherwise, when an exception is taken from AArch32 state to AArch64 state, for a general-purpose register that was not accessible from AArch32 state, the register retains the value it had before any AArch32 execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.10.1.4.3 Determining the upper 32 bits of AArch64 registers on taking an exception from AArch32 state

R_{ZJBSH} Whether a general-purpose register has its upper 32 bits set to zero or retained on taking an exception from AArch32 to AArch64 is affected by all of the following:

- The AArch64 state target Exception level.
- The values of both:
  - SCR_EL3.RW.
  - HCR_EL2.RW or HCR.RW, where HCR.RW is a notional bit that is RES 0.

R_{DGVGF} The following table shows which general-purpose registers can have their upper 32 bits set to zero on taking an exception from AArch32 state to AArch64 state. In this table, a dash (-) indicates that the RW values are not valid for the targeted Exception level.

SCR_EL3.RW	HCR_EL2.RW or HCR.RW	Registers when the target Exception level is:		
		EL3	EL2	EL1
0	0	X0-X30	-	-
0	1	Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.		
1	0	X0-X14, X16-X30	X0-X14, X16-X30	-
1	1	X0-X14	X0-X14	X0-X14

R_{WKWVT} If EL2 is not implemented, or the SCR_EL3.NS or SCR.NS bits prevent its use, the registers that can have their upper 32 bits set to zero on taking an exception from AArch32 state to AArch64 state are the same as when HCR_EL2.RW has the same value as SCR_EL3.RW.

R_{WPVQR} The following table shows which general-purpose registers can retain their upper 32 bits on taking an exception from AArch32 state to AArch64 state. In this table, a dash (-) indicates that the RW values are not valid for the target Exception level.

SCR_EL3.RW	HCR_EL2.RW or HCR.RW	Registers when the target Exception level is:		
		EL3	EL2	EL1
0	0	None	-	-
0	1	Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.		
1	0	X15	X15	-
1	1	X15-X30	X15-X30	X15-X30

R_{LGSLY} If EL2 is not implemented, or SCR_EL3.NS prevents its use, the registers that can have their upper 32 bits retained on taking an exception to AArch64 state from AArch32 state are the same as when HCR_EL2.RW has the same value as SCR_EL3.RW.

D1.10.1.5 SPSR, ELR, and AArch64 SP relationships on changing Execution state

I_{PSRRS} The following table shows SPSR and ELR registers that are architecturally mapped between AArch32 state and AArch64 state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch32 register	AArch64 register
SPSR_svc	SPSR_EL1
SPSR_hyp	SPSR_EL2
ELR_hyp	ELR_EL2

R_{CHHXX} When an exception is taken from an Exception level using AArch32 to EL3 using AArch64 and EL2 has been using AArch32 state, the upper 32 bits of ELR_EL2 have one of the following IMPLEMENTATION DEFINED values:

- The upper 32 bits retain the value that the same architectural register held before any AArch32 execution.
- The upper 32 bits are set to zero.

R_{WFDMR} The following AArch32 registers are used only during execution in AArch32 state. However, they retain their state if there is execution at EL1 and EL1 is using AArch64 state.

- SPSR_abt.
- SPSR_und.
- SPSR_irq.
- SPSR_fiq.

I_{TBFWN} For the purposes of context switching, the registers in R_{WFDMR} are accessible during execution in AArch64 state at Exception levels higher than EL1.

I_{WRPZP} If EL1 does not support execution in AArch32 state, the registers in R_{WFDMR} are RES0.

R_{MBVJZ} If an exception is taken from an Exception level using AArch32 to an Exception level using AArch64, the AArch64 stack pointers and Exception Link Registers associated with an Exception level that are not accessible during AArch32 execution at the Exception level from which the exception was taken, retain the state that they had before AArch32 execution. This applies to the following registers:

- SP_EL0.
- SP_EL1.
- SP_EL2.
- ELR_EL1.

D1.10.1.6 PSTATE.SM and PSTATE.ZA behaviors on changing Execution state

R_{SWQGH} The values of PSTATE.{SM, ZA} do not change when any of the following occurs:

- An exception return from AArch64 to AArch32 Execution state.
- An exception taken from AArch32 to AArch64 Execution state.

R_{GXKNK} In AArch32 state, the Effective value of PSTATE.SM is 0.

I_{MWQNV} When PSTATE.SM is 1, a change in Execution state from AArch64 to AArch32, or from AArch32 to AArch64, causes all implemented bits of the SVE registers (including SIMD&FP registers) and the FPSR to be reset to a fixed value.

I_{WYKRM} The value of PSTATE.ZA does not change in AArch32 Execution state. Therefore, a transition between AArch64 and AArch32 Execution state when PSTATE.ZA is 1 has no effect on the contents of ZA storage, or the ZT0 register when FEAT_SME2 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.11 Check Feature

IFCDFK FEAT_CHK provides CHKFEAT with the following properties:

- CHKFEAT is allocated from the Hint instruction space, to allow it to be used on any PE, regardless of whether FEAT_CHK is implemented.
- CHKFEAT takes a value as an input where one or more bits of the input value is set to 1 to test whether the requested feature is enabled, and if the feature is enabled the relevant bit is cleared to 0. Otherwise, the bit is unchanged.
- When FEAT_CHK is not implemented, the input value is unchanged, indicating the requested feature is not enabled.
- CHKFEAT returns the state of GCSEnabled().

ICZRRH For the encoding allocation for CHKFEAT, see op0 == 0b00, architectural hints, barriers and CLREX, and PSTATE access.

IQMQXS Since the CHKFEAT instruction is in the Hint instruction encoding space, it is effectively implemented on all PEs that support AArch64. However, the CHKFEAT instruction only returns a result value that is different from the input value when any of the features that are identified by the CHKFEAT are implemented.

RVXBTN An example of how CHKFEAT can be used in an implementation independent manner is shown below.

Example D1-1 Example code for using CHKFEAT

```
    MOV X16, #0x1           ; X16 has bit [0] set to select GCS
    CHKFEAT X16             ; Updates X16 with the status of GCS
    TBNZ X16, #0, skipgcs   ; Skip over GCS code if GCS is not enabled

    ...                     ; GCS related code

skipgcs:
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D2
AArch64 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch64 self-hosted debug Exception model. It is organized as follows:

Introductory information

- About self-hosted debug.
- The debug exception enable controls.

The debug Exception model

- Routing debug exceptions.
- Enabling debug exceptions from the current Exception level and Security state.
- The effect of powerdown on debug exceptions.
- Summary of the routing and enabling of debug exceptions.
- Pseudocode description of debug exceptions.

The debug exceptions

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

Synchronization requirements

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a Context Synchronization event guarantees the effects of the changes:

- Synchronization and debug exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of *debug exceptions*, which are taken using the Exception model described in The AArch64 System Level Programmers' Model. This section introduces some terms that are used in describing self-hosted debug, and then introduces the debug exceptions. See:

- Definition of a debugger in the context of self-hosted debug.
- Context ID and Process ID.
- About debug exceptions.

D2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the Debug system registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

D2.1.2 Context ID and Process ID

A CONTEXTIDR_ELx identifies the current *Context ID*, which is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

In AArch64 state, the CONTEXTIDR_ELx has a single field, PROCID, that is defined as the *Process Identifier* (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical.

D2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger enables one or more debug exceptions. The debug exceptions that can be generated in stage 1 of an AArch64 translation regime are:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions, generated by hardware breakpoints.
- Watchpoint exceptions, generated by hardware watchpoints.
- Software Step exceptions.

In addition, debug exceptions generated in an AArch32 translation regime might be routed to EL2 using AArch64. See Routing debug exceptions. AArch32 Self-hosted Debug describes the debug exceptions that can be generated in an AArch32 translation regime.

Vector Catch exceptions are exceptions that cannot be generated in an AArch64 translation regime but can be generated in stage 1 of an AArch32 translation regime and routed to EL2 using AArch64. Vector Catch exceptions describes the behavior for this case.

Other than Breakpoint Instruction exceptions, the PE can generate a particular debug exception only if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- Debug exceptions are not disabled from the current Exception level and Security state.
  See Enabling debug exceptions from the current Exception level and Security state.
- A debugger has not disabled that particular debug exception.
  All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See The debug exception enable controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Breakpoint Instruction exceptions are always enabled.

Note

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.
For the definition of halting is allowed, see Halting allowed and halting prohibited.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Exception level.
- The current Security state.

For more information, see Breakpoint Instruction exceptions.

Breakpoint exceptions

The architecture provides 2-64 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on one or more of particular instruction addresses, accesses to any address in an instruction address range, or particular PE contexts.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.

The architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch64 translation regime:

- Address.
  - Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID held in the CONTEXTIDR_EL1, and if FEAT_VHE is implemented, CONTEXTIDR_EL2.
  - VMID Match. Matches with the VMID value held in the VTTBR_EL2.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a context-matching breakpoint, so that the Address breakpoint generates a Breakpoint exception only if the PE is in a particular context when the address match occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Debug State.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see Breakpoint exceptions.

Watchpoint exceptions

The architecture provides 2-64 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000 - 0x101F`.

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a Context matching breakpoint with linking enabled. In this case, the watchpoint generates a Watchpoint exception only if the PE is in a particular context when the data address match occurs.

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is an address breakpoint with linking enabled. In this case, the watchpoint generates a Watchpoint exception only if the address of the instruction making the access matches the breakpoint.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Debug State.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see Watchpoint exceptions.

Vector Catch exceptions

These are not generated in an AArch64 translation regime. They can be generated only in an AArch32 translation regime. See Vector Catch exceptions.

Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can debug software executing at a lower Exception level, by making it single-step instructions.

After the software being debugged has single-stepped an instruction, the PE takes a Software Step exception.

Software step can be used only by a debugger executing in an Exception level that is using AArch64. However, the instruction stepped might be executed in either Execution state, and therefore Software Step exceptions can be taken from either Execution state.

If debug exceptions are enabled, Software Step exceptions can be generated.

If debug exceptions are disabled, software step is inactive.

For more information, see Software Step exceptions.

Table D2-1 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

Table D2-1 PE behavior and pseudocode for each of the debug exceptions

Debug exception	PE behavior if debug exceptions are:		Pseudocode
	Enabled	Disabled	
Breakpoint Instruction exceptions	Takes the exception	Takes the exception	Pseudocode description of Breakpoint Instruction exceptions
Breakpoint exceptions	Takes the exception[a]	Ignored	Pseudocode description of Breakpoint exceptions taken from an AArch64 stage 1 translation regime

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug exception	PE behavior if debug exceptions are:		Pseudocode
	Enabled	Disabled	
Watchpoint exceptions	Takes the exception[a]	Ignored	Pseudocode description of Watchpoint exceptions taken from AArch64 state
Vector Catch exceptions	Takes the exception	Ignored	Pseudocode description of Vector Catch exceptions
Software Step exceptions	Takes the exception	Not applicable[b]	Pseudocode description of Software Step exceptions

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Debug State.

b. Software Step is inactive if debug exceptions are disabled. No Software Step exceptions can be generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.2 Routing debug exceptions

Debug exceptions are routed to the *debug target Exception level*, EL_D, according to the following controls:

- HCR_EL2.TGE.
- MDCR_EL2.TDE.
- The Security state when the exception is taken.
- The Exception level where the exception is taken.

This means that, for all debug exceptions, EL_D is either EL1 or EL2.

Note

If EL2 is not implemented, the Effective value of HCR_EL2.TGE is 0 and the Effective value of MDCR_EL2.TDE is 0. Throughout this section, references to the values of these fields are to the Effective values of the fields.

The routing of debug exceptions is as follows:

Debug exceptions taken when EL2 is implemented and enabled in the current Security state

The routing of debug exceptions taken depends on the values of MDCR_EL2.TDE and HCR_EL2.TGE:

If the Effective value of {MDCR_EL2.TDE, HCR_EL2.TGE} is not {0, 0}

Debug exceptions are routed to EL2, EL_D is EL2.

Otherwise

Debug exceptions behave as follows:

- Debug exceptions taken from EL1 and EL0 are routed to EL1. EL_D is EL1.
- Breakpoint Instruction exceptions taken from EL2 are routed to EL2.
- All other debug exceptions are disabled from EL2 using AArch64.

When EL3 is implemented

Breakpoint Instruction exceptions taken from EL3 are routed to EL3.

All other debug exceptions are disabled from EL3 using AArch64.

Otherwise

Debug exceptions are routed to EL1. EL_D is EL1.

This means that, for all debug exceptions, the *debug target Exception level*, EL_D, is either EL1 or EL2.

Table D2-2, Table D2-3, and Table D2-4 show the routing of debug exceptions. In these tables:

NSE Means the Effective value of SCR_EL3.NSE. If FEAT_RME is not implemented, this is 0.

NS Means the Effective value of SCR_EL3.NS. If Secure state is not implemented, this is 1.

EEL2 Means the Effective value of SCR_EL3.EEL2. If FEAT_SEL2 is not implemented, this is 0.

TDE or TGE Means the logical OR of the Effective value of MDCR_EL2.TDE and the Effective value of HCR_EL2.TGE.

(EL*x*) Means EL_D is EL*x*. However:

- All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Exception level.
- Breakpoint Instruction exceptions taken when executing in this Exception level are routed to the same Exception level. This may not be the same as the EL_D Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL*x* Means EL_D is EL*x*.

Table D2-2 Routing when both EL3 and EL2 are implemented

NSE	NS	EEL2	TDE or TGE	EL_D when executing in:			
				EL0	EL1	EL2	EL3
0	0	0	x	EL1	EL1	n/a	(EL1)
0	0	1	0	EL1	EL1	(EL1)	(EL1)
0	0	1	1	EL2	EL2	EL2	(EL2)
x	1	x	0	EL1	EL1	(EL1)	(EL1)
x	1	x	1	EL2	EL2	EL2	(EL2)

Table D2-3 Routing when EL3 is implemented and EL2 is not implemented

EL_D when executing in:		
EL0	EL1	EL3
EL1	EL1	(EL1)

Table D2-4 Routing of exceptions

TDE or TGE	EL_D when executing in:		
	EL0	EL1	EL2
0	EL1	EL1	(EL1)
1	EL2	EL2	EL2

D2.2.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level.

`DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

These functions are described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.3 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each breakpoint, DBGBCR<n>_EL1.E.

If more than 16 breakpoints are implemented, then extended breakpoint and watchpoint enable controls are implemented for the additional breakpoints, MDCR_EL3.EBWE, MDCR_EL2.EBWE, MDSCR_EL1.EMBWE, and EDSCR2.EHBWE.

Watchpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each watchpoint, DBGWCR<n>_EL1.E.

If more than 16 watchpoints are implemented, then extended breakpoint and watchpoint enable controls are implemented for the additional watchpoints, MDCR_EL3.EBWE, MDCR_EL2.EBWE, MDSCR_EL1.EMBWE, and EDSCR2.EHBWE.

Vector Catch exceptions

MDSCR_EL1.MDE.

Software Step exceptions

MDSCR_EL1.SS.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

D2.3.1 Enabling debug exceptions from the current Exception level and Security state

Debug exceptions are enabled from the current Exception level, where EL_D is the Exception level defined in Routing debug exceptions, as follows:

- When executing at any Exception level that is higher than EL_D:
  - Breakpoint Instruction exceptions are enabled.
  - All other debug exceptions are disabled.
- When executing at EL_D:
  - Breakpoint Instruction exceptions are enabled.
  - All other debug exceptions are disabled if either of the following is true:
    - The Local (kernel) Debug Enable bit, MDSCR_EL1.KDE, is 0.
    - The Debug exception mask bit, PSTATE.D, is 1.

  This means that a debugger must explicitly enable these debug exceptions from EL_D by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0.
- When executing at EL1, EL_D is EL2, and FEAT_NV2 is implemented:
  - Watchpoint debug exceptions generated by a System register access converted to a memory access because the Effective value of HCR_EL2.NV2 is 1 are disabled if MDSCR_EL1.KDE is 0 and enabled if MDSCR_EL1.KDE is 1. The value of PSTATE.D is ignored.
  - All other debug exceptions are enabled.
- Otherwise all debug exceptions are enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-5 shows when debug exceptions are enabled from the current Security state.

If EL3 and Secure state are implemented, software executing at EL3 can set the *Secure Debug Disable* bit, MDCR_EL3.SDD, to 1 to disable all debug exceptions taken from AArch64 Secure state other than Breakpoint Instruction exceptions.

The A-profile architecture does not support disabling debug exceptions in Non-secure state and, if FEAT_RME is implemented, Realm state.

Table D2-5 Whether debug exceptions are enabled from the current Security state

Current Security state	Breakpoint Instruction exceptions	All other debug exceptions
Non-secure	Enabled	Enabled
Secure	Enabled	Disabled if the Effective value of MDCR_EL3.SDD is 1. Otherwise enabled.
Realm	Enabled	Enabled
Root	Enabled	Disabled

If EL3 is not implemented and the implementation is a Secure state only implementation, or Secure state is not implemented, then the Effective value of MDCR_EL3.SDD is 0.

D2.3.2 Pseudocode description of enabling debug exceptions

`AArch64_GenerateDebugException()` determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the current Exception level and Security state.

`AArch64_GenerateDebugExceptionsFrom()` determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the specified Exception level and Security state.

These functions are described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.4 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

If the OS Lock is unlocked, and `DoubleLockStatus()` == FALSE, debug exceptions other than Breakpoint Instruction exceptions are enabled.

If OS Lock is locked, or if `DoubleLockStatus()` ==TRUE, debug exceptions other than Breakpoint Instruction exceptions are disabled.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.5 Summary of the routing and enabling of debug exceptions

Behavior is as follows:

Breakpoint Instruction exceptions

These are always enabled, regardless of the current Exception level and Security state. A Breakpoint Instruction exception taken from EL3 is always routed to EL3. A Breakpoint Instruction exception taken from EL2 is routed to EL2. A Breakpoint Instruction exception taken from EL0 or EL1 is always routed to EL_D.

All other debug exceptions

Table D2-6 shows the valid combinations of MDCR_EL3.SDD, MDCR_EL2.TDE, MDSCR_EL1.KDE, and PSTATE.D, and for each combination shows where these exceptions are enabled from and where they are taken to. The table does not include the case when executing at EL1, EL_D is EL2, and FEAT_NV2 is implemented that is described in Enabling debug exceptions from the current Exception level and Security state.

In the table:

Lock Means the value of (OSLSR_EL1.OSLK == '1' || `DoubleLockStatus()`).

NSE Means the Effective value of SCR_EL3.NSE. If FEAT_RME is not implemented, this is 0.

NS Means the Effective value of SCR_EL3.NS. If Secure state is not implemented, this is 1.

SDD Means the Effective value of MDCR_EL3.SDD. See Disabling debug exceptions from Secure state.

EEL2 Means the Effective value of SCR_EL3.EEL2. If FEAT_SEL2 is not implemented, this is 0.

TGE Means the Effective value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.

TDE Means the Effective value of MDCR_EL2.TDE. If EL2 is not implemented, the PE behaves as if this is 0.

KDE Means the value of MDSCR_EL1.KDE.

D Means the value of PSTATE.D.

n/a Means not applicable. The PE cannot be executing at this Exception level.

- Means that debug exceptions are disabled from that Exception level.

Table D2-6 Routing of Breakpoint, Watchpoint, Software Step, and Vector Catch exceptions

Debug state	Lock	NSE	NS	SDD	EEL2	TGE	TDE	KDE	D	EL_D when enabled from:			
										EL0	EL1	EL2	EL3
Yes	X	X	X	X	X	X	X	X	X	-	-	-	-
No	TRUE	X	X	X	X	X	X	X	X	-	-	-	-
	FALSE	0	0	1	X	X	X	X	X	-	-	-	-
				0	0	X	X	0	X	EL1	-	n/a	-
								1	0	EL1	EL1	n/a	-
									1	EL1	-	n/a	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug state	Lock	NSE	NS	SDD	EEL2	TGE	TDE	KDE	D	EL_D when enabled from:			
										EL0	EL1	EL2	EL3
					1	0	0	0	X	EL1	-	-	-
								1	0	EL1	EL1	-	-
									1	EL1	-	-	-
No	FALSE	0	0	0	1	0	1	0	X	EL2	EL2	-	-
								1	0	EL2	EL2	EL2	-
									1	EL2	EL2	-	-
						1	X	0	X	EL2	n/a	-	-
								1	0	EL2	n/a	EL2	-
									1	EL2	n/a	-	-
		X	1	X	X	0	0	0	X	EL1	-	-	-
								1	0	EL1	EL1	-	-
									1	EL1	-	-	-
							1	0	X	EL2	EL2	-	-
								1	0	EL2	EL2	EL2	-
									1	EL2	EL2	-	-
						1	X	0	X	EL2	n/a	-	-
								1	0	EL2	n/a	EL2	-
									1	EL2	n/a	-	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.6 Pseudocode description of debug exceptions

The `AArch64_Abort()` function processes `FaultRecord` objects, as described in Abort exceptions, and generates a debug exception.

Some functions called by `AArch64_Abort()` are:

- `AArch64_BreakpointException()`.
- `AArch64_WatchpointException()`.
- `AArch64_VectorCatchException()`.
- `AArch64_InstructionAbort()`.
- `AArch64_DataAbort()`.

These functions are defined in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.7 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing one of the following:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.
- At EL0 using AArch32 when FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

For software executing in an Exception level that is using AArch64, a Breakpoint Instruction exception results from the execution of an A64 `BRK` instruction. Within the AArch64 EL1&0 or EL2&0 translation regime, executing a T32 or A32 `BKPT` instruction at EL0 using AArch32 generates a Breakpoint Instruction exception.

For more information about the T32 and A32 `BKPT` instructions, see:

- Breakpoint instruction in the T32 and A32 instruction sets.
- BKPT instructions as the first instruction in an IT block.

The following subsections describe Breakpoint Instruction exceptions in an AArch64 translation regime:

- About Breakpoint Instruction exceptions.
- Breakpoint instructions.
- Exception syndrome information and preferred return address.
- Pseudocode description of Breakpoint Instruction exceptions.

D2.7.1 About Breakpoint Instruction exceptions

A *breakpoint* is an event that results from the execution of an instruction, which is based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint Instruction exceptions, which this section describes, are software breakpoints. Breakpoint exceptions describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

- The current Exception level.
- The current Security state.
- Whether the *debug target Exception level*, EL_D, is using AArch64 or AArch32.

Note

Debuggers using breakpoint instructions must be aware of the rules for concurrent modification and execution of instructions. See Concurrent modification and execution of instructions.

D2.7.2 Breakpoint instructions

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. It is unconditional.

For details of the instruction encoding, see BRK.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The breakpoint instruction in the T32 and A32 instruction sets is `BKPT #<immediate>`.

For more information about the T32 and A32 breakpoint instruction, see Breakpoint instruction in the T32 and A32 instruction sets.

D2.7.3 Exception syndrome information and preferred return address

See the following:

- Exception syndrome information.
- Preferred return address.

D2.7.3.1 Exception syndrome information

On taking a Breakpoint Instruction exception, the PE records information about the exception in the *Exception Syndrome Register* (ESR) at the Exception level the exception is taken to in ESR_ELx

Note

Breakpoint Instruction exceptions are the only debug exception that can be taken to EL3 using AArch64.

Table D2-7 shows the information that the PE records.

Table D2-7 Information recorded in the ESR_ELx

ESR_ELx field	Information recorded in ESR_EL1, ESR_EL2, or ESR_EL3.
Exception Class, EC	Whether the breakpoint instruction was executed in AArch64 state or AArch32 state. The PE sets this to: • `0x3C` for an A64 `BRK` instruction. • `0x38` for an A32 or T32 `BKPT` instruction.
Instruction Length, IL	The PE sets this to: • 0 for a 16-bit T32 `BKPT` instruction. • 1 for an A64 `BRK` instruction, or an A32 `BKPT` instruction.
Instruction Specific Syndrome, ISS	**ISS[24:16]** RES0. **ISS[15:0]** The PE copies the instruction Comment field value into here, zero extended as necessary.

Note

- If debug exceptions are routed to EL2, it is the exception that is routed, not the instruction that is trapped. Therefore, if a Breakpoint Instruction exception is routed to EL2, ESR_EL2.EC is set to the same value as if the exception was taken to EL1.
- For information about how debug exceptions can be routed to EL2, see Routing debug exceptions.

D2.7.3.2 Preferred return address

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

D2.7.4 Pseudocode description of Breakpoint Instruction exceptions

`AArch64_SoftwareBreakpoint()` generates a Breakpoint Instruction exception that is taken to AArch64 state.

This function is defined in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.8 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing one of the following:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.
- At EL0 using AArch32 when FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} == {1, 1}.

This section contains the following subsections:

- About Breakpoint exceptions.
- Breakpoint types and linking of breakpoints.
- Execution conditions for which a breakpoint generates Breakpoint exceptions.
- Breakpoint instruction address comparisons.
- Breakpoint context comparisons.
- Breakpoint usage constraints.
- Preferred return address.
- Pseudocode description of Breakpoint exceptions taken from an AArch64 stage 1 translation regime.

D2.8.1 About Breakpoint exceptions

A *breakpoint* is an event that results from the execution of an instruction, which is based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint exceptions are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in Breakpoint Instruction exceptions.

An implementation can include between 2-64 hardware breakpoints.

If AArch32 is supported at EL1 or FEAT_Debugv8p9 is not implemented, then the PE does not implement more than 16 breakpoints.

When more than 16 breakpoints are implemented, the number of breakpoints is identified to software by ID_AA64DFR1_EL1.BRPs and EDDFR1.BRPs. Otherwise ID_AA64DFR0_EL1.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>_EL1. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>_EL1. This holds the value used for breakpoint matching, which is one of:
  - An instruction virtual address.
  - A Context ID.
  - A VMID value.
  - A concatenation of both a Context ID value and a VMID value.
  - If FEAT_VHE is implemented, a concatenation of two Context ID values.

These registers are numbered, so that:

- DBGBCR0_EL1 and DBGBVR0_EL1 are for breakpoint number zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- DBGBCR1_EL1 and DBGBVR1_EL1 are for breakpoint number one.
- . . .
- DBGBCR<n-1>_EL1 and DBGBVR<n-1>_EL1 are for breakpoint number (n-1).

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is generated only if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the Effective value of DBGBCR<n>_EL1.E is 1.
- The conditions specified in the DBGBCR<n>_EL1 are met.
- The comparison with the value held in the DBGBVR<n>_EL1 is successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

The address of each instruction is compared against the DBGBVR<n>_EL1 values of all Address matching breakpoints with linking disabled that can generate a breakpoint match at the current Exception level in the current Security state, possibly after the application of a breakpoint mask, and:

- The Boolean equality comparison results of all the address match breakpoints are OR'd together to generate a first result.
- The Boolean inequality comparison results of all the address mismatch breakpoints are AND'd together to generate a second result.

Note

Because ranges have power-of-two sizes and are aligned, it is not possible to configure strictly overlapping pairs of ranges. Only non-overlapping or subsets can be selected.

The resulting address match is:

- False, if there are no such breakpoints.
- The first result, if all such breakpoints are address match breakpoints.
- The second result, if all such breakpoints are address mismatch breakpoints.
- The logical-AND of the first result and the second result, if there is a mix of address match and address mismatch breakpoints.

This address match result is OR'd with the Boolean results of all Context matching breakpoints with linking disabled that are enabled and can generate a breakpoint match at the current Exception level in the current Security state. The breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its Condition code check.
- The instruction type.

Note

This does not apply to breakpoints with linking enabled. A breakpoint with linking enabled only delivers its result to the breakpoint(s) and/or watchpoint(s) linked to it, if any.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:

- Enabled, Breakpoint debug events generate Breakpoint exceptions.
- Disabled, Breakpoint debug events are ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions.
However, the behavior described also applies if breakpoints are causing entry to Debug state.

Example D2-1 shows an example of setting an address match breakpoint and an address mismatch breakpoint, and the conditions that cause a breakpoint exception.

Example D2-1 Breakpoint address match and breakpoint address mismatch

Two breakpoints are programmed as follows:

- Breakpoint 0 is programmed with:
  - DBGBCR0_EL1.{BT2, BT} = *Address match breakpoint*.
  - DBGBCR0_EL1.MASK = M_0, where $M_0 > 2$.
  - DBGBVR0_EL1 = A_0, where A_0 is aligned to 2^{M_0}.

  This means that breakpoint 0 will match an instruction address B if it is within a range, as follows:
  $B \in \{A_0, A_0 + 4, \ldots (A_0 + 2^{M_0} - 4)\}$
  $\Rightarrow (A_0 \leq B) \wedge (B < A_0 + 2^{M_0})$
- Breakpoint 1 is programmed with:
  - DBGBCR1_EL1.{BT2, BT} = *Address mismatch breakpoint*.
  - DBGBCR1_EL1.MASK = M_1, where $M_1 > 2$.
  - DBGBVR1_EL1 = A_1, where A1 is aligned to 2^{M_1}.

  This means that breakpoint 1 will match an instruction address B if it is outside of a range, as follows:
  $B \notin \{A_1, A_1 + 4, \ldots (A_1 + 2^{M_1} - 4)\}$
  $\Rightarrow (B < A_1) \vee (B \geq A_1 + 2^{M_1})$
- Such that $\{A_1, A_1 + 4, \ldots (A_1 + 2^{M_1} - 4)\} \subset \{A_0, A_0 + 4, \ldots (A_0 + 2^{M_0} - 4)\}$.
- Both breakpoints match at the current Exception level in the current Security state.

Then the combination of these two breakpoints will only generate a breakpoint for an instruction address B if it is both within the first range and outside of the second range, as follows:

$B \in \{A_0, A_0 + 4, \ldots (A_0 + 2^{M_0} - 1)\} \wedge B \notin \{A_1, A_1 + 4, \ldots (A_1 + 2^{M_1} - 4)\}$

$\Rightarrow B \in \{A_0, A_0 + 4, \ldots (A_1 - 4), (A_1 + 2^{M_1}), (A_1 + 2^{M_1} + 4), \ldots (A_0 + 2^{M_0} - 4)\}$

$\Rightarrow ((A_0 \leq B) \wedge (B < A_1)) \vee ((B \geq A_1 + 2^{M_1}) \wedge (B < A_0 + 2^{M_0}))$

The debug exception enable controls describes the enable controls for Breakpoint debug events.

D2.8.2 Accessing breakpoint System registers

R_{MLTYN} When any of the following are true, the Effective value of MDSELR_EL1.BANK is zero:

- Executing at EL3 and MDCR_EL3.EBWE is 0.
- Executing at EL2 and the Effective value of MDCR_EL2.EBWE is 0.
- Executing at EL1 and the Effective value of MDSCR_EL1.EMBWE is 0.

R_{JGQGB} If all of the following are true, MDSELR_EL1 is RES0:

- FEAT_Debugv8p9 is implemented.
- 16 or fewer breakpoints are implemented.
- 16 or fewer watchpoints are implemented.

R_{KSCWV} The MRS and MSR System register names for accessing breakpoint control registers DBGBCR<n>_EL1 and breakpoint value registers DBGBVR<n>_EL1 are mapped to physical registers as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The register names DBGBCR0_EL1 through DBGBCR15_EL1 are mapped to the physical registers DBGBCR<BANK×16+0>_EL1 through DBGBCR<BANK×16+15>_EL1.
- The register names DBGBVR0_EL1 through DBGBVR15_EL1 are mapped to the physical registers DBGBVR<BANK×16+0>_EL1 through DBGBVR<BANK×16+15>_EL1.

IHMNFZ MSR and MRS instructions using the breakpoint System register names in RKSCWV make indirect reads of MDSELR_EL1.BANK. This means that direct writes to MDSELR_EL1.BANK must be explicitly synchronized before a direct read or direct write using any of these System register names.

D2.8.3 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match or mismatch.
  - A Context ID match, with the value held in CONTEXTIDR_EL1 or the value held in CONTEXTIDR_EL2.
  - A VMID match, with the VMID value held in VTTBR_EL2.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not *context-aware*. These can only be programmed to generate a Breakpoint exception on an instruction address match.

If more than 16 *context-aware* breakpoints are implemented, the number of *context-aware* breakpoints is identified to software by ID_AA64DFR1_EL1.CTX_CMPs and EDDFR1.CTX_CMPs. Otherwise ID_AA64DFR0_EL1.CTX_CMPs indicates how many are implemented.

At least one breakpoint must be *context-aware*.

If 16 or fewer breakpoints are implemented, the highest numbered are *context-aware*.

If more than 16 breakpoints are implemented:

- If 16 or fewer are *context-aware*, these are the highest numbered breakpoints up to breakpoint number 15.
- Otherwise, *context-aware* breakpoints are numbered consecutively beginning at breakpoint number 0.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case, the breakpoint is called an *Unlinked breakpoint*.
- Linked to another Context matching breakpoint. In this case, the breakpoint is called a *Linked breakpoint*.
- Enabled for linking from another breakpoint or watchpoint. In this case, the breakpoint has *Linking enabled* and does not generate debug events in isolation.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that generates a Breakpoint exception only if the PE is in a particular context when an instruction address match occurs. This is shown in Example D2-2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D2-2 Linking an Address breakpoint and a Context breakpoint

For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Context matching breakpoint with linking enabled*.
3. Link these two breakpoints together. A Breakpoint exception is generated only if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* fields for a breakpoint, DBGBCR<n>_EL1.{BT2, BT}, control the breakpoint type and whether the breakpoint is enabled for linking.

If AArch32 state is implemented, Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See Specifying the halfword-aligned address that an Address breakpoint matches on.

D2.8.3.1 Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked Address breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Context matching breakpoint with linking enabled.

  The *Linked Breakpoint Number* field, DBGBCR<n>_EL1.{LBNX, LBN}, for the Linked Address breakpoint specifies the particular Context matching breakpoint that the Linked Address breakpoint links to, and:

  — The {LBNX, LBN} fields are ignored if the breakpoint is not a Linked Address breakpoint.

  — DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.

  — DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} for the Context matching breakpoint with linking enabled are ignored.
- Linked Address breakpoints cannot link to watchpoints. The {LBNX, LBN} fields always specify a breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not *context-aware* or does not have linking enabled, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See Other usage constraints for Address breakpoints.

A single Context matching breakpoint with linking enabled might be linked to by all, or any combination of, the following:

- Multiple Linked Address breakpoints.
- Multiple Linked watchpoints.

If FEAT_ABLE is implemented, a single Address matching breakpoint with linking enabled might be linked to by multiple Linked watchpoints.

Note

If FEAT_NV2 is implemented, the hypervisor must use the `{0b0, 0b1101}`, CONTEXTIDR_EL2 Match breakpoint with linking enabled type to guarantee a linked match, see Interaction with self-hosted and External debug.

If no linked breakpoints and no linked watchpoints link to a breakpoint programmed with linking enabled, then the breakpoint does not generate any Breakpoint exceptions or debug events.

Figure D2-1 shows an example of permitted breakpoint and watchpoint linking.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

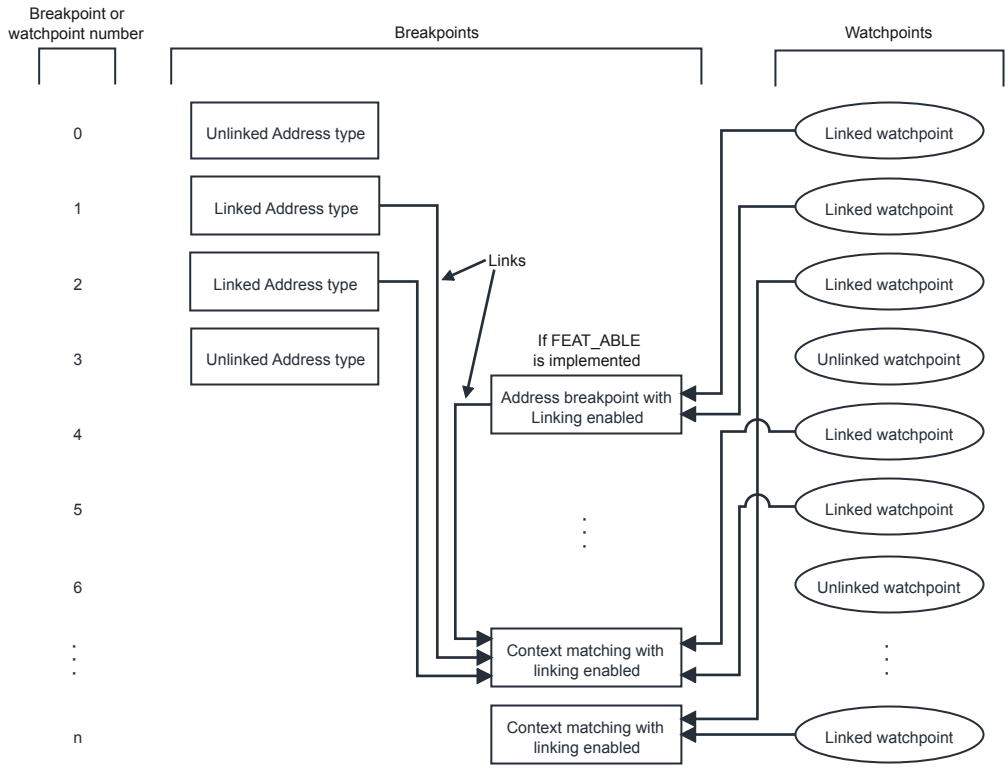

Figure D2-1 The role of linking in Breakpoint and Watchpoint exception generation

In Figure D2-1, each Linked Address breakpoint can generate a Breakpoint exception only if the comparisons made by both it, and the breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the breakpoint that it links to, are successful.

D2.8.3.2 Breakpoint types defined by DBGBCR<n>_EL1.{BT2, BT}

If FEAT_ABLE is not implemented then the Effective value of DBGBCR<n>_EL1.BT2 is 0.

The following lists provides more detail about each breakpoint type:

`{0b0, 0b0000}`, **Unlinked Address Match breakpoint**

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address match, as described in Breakpoint instruction address comparisons.

DBGBCR<n>_EL1.{LBNX, LBN} for this breakpoint are ignored.

`{0b0, 0b0001}`, **Linked Address Match breakpoint**

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A successful address match defined by this breakpoint, as described in Breakpoint instruction address comparisons.
- A successful context match defined by the Context breakpoint with linking enabled that this breakpoint links to.

DBGBCR<n>_EL1.{LBNX, LBN} for this breakpoint selects the Context breakpoint with linking enabled that this breakpoint links to.

`{0b0, 0b0010}`, **Context ID Match breakpoint**

{BT2, BT} == `{0b0, 0b0010}` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the Effective value of HCR_EL2.E2H is 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBNX, LBN, BAS} for this breakpoint are ignored

`{0b0, 0b0011}`, **Context ID Match breakpoint with linking enabled**

{BT2, BT} == `{0b0, 0b0011}` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the Effective value of HCR_EL2.E2H is 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`{0b0, 0b0100}`, **Unlinked Address Mismatch breakpoint**

{BT2, BT} == `{0b0, 0b0100}` is a reserved value unless one of the following applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_BWE is implemented.
- EL1 is using AArch32.

In stage 1 of an AArch64 translation regime, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address mismatch, as described in Breakpoint instruction address comparisons.

DBGBCR<n>_EL1.{LBNX, LBN} for this breakpoint is ignored.

`0b0100`, Unlinked Address Mismatch breakpoint describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

`{0b0, 0b0101}`, **Linked Address Mismatch breakpoint**

{BT2, BT} == {`0b0`, `0b0101`} is a reserved value unless one of the following applies:

- FEAT_BWE is implemented.
- EL1 is using AArch32.

In stage 1 of an AArch64 translation regime, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address mismatch defined by this breakpoint, as described in Breakpoint instruction address comparisons.
- A successful context match defined by the Context matching breakpoint that this breakpoint links to.

DBGBCR<n>_EL1.{LBNX, LBN} for this breakpoint selects the Context matching breakpoint that this breakpoint links to.

`0b0101`, Linked Address Mismatch breakpoint describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

`{0b0, 0b0110}`, **CONTEXTIDR_EL1 Match breakpoint**

{BT2, BT} == {`0b0`, `0b0110`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.

In an implementation that includes FEAT_VHE, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<n>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBNX, LBN, BAS} for this breakpoint are ignored.

`{0b0, 0b0111}`, **CONTEXTIDR_EL1 Match breakpoint with linking enabled**

{BT2, BT} == {`0b0`, `0b0111`} is a reserved value if either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.

In an implementation that includes FEAT_VHE, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<n>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`{0b0, 0b1000}`, **VMID Match breakpoint**

{BT2, BT} == {`0b0`, `0b1000`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful VMID match, as described in Breakpoint context comparisons.

DBGBCR<n>_EL1.{LBNX, LBN, BAS} for this breakpoint are ignored.

`{0b0, 0b1001}`, **VMID Match breakpoint with linking enabled**

{BT2, BT} == {`0b0`, `0b1001`} is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See Breakpoint instruction address comparisons.
  - A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`{0b0, 0b1010}`, **Context ID and VMID Match breakpoint**

{BT2, BT} == {`0b0`, `0b1010`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

When EL2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates a Breakpoint exception for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.
- A successful VMID match.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1.

Breakpoint context comparisons describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>_EL1.{LBNX, LBN, BAS} for this breakpoint are ignored.

`{0b0, 0b1011}`, **Context ID and VMID Match breakpoint with linking enabled**

{BT2, BT} == {`0b0`, `0b1011`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

When EL2 is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on all of the following:
  - — A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
  - — A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  - — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
  - — A successful VMID match defined by this breakpoint.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1.

Breakpoint context comparisons describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`{0b0, 0b1100}`, **CONTEXTIDR_EL2 Match breakpoint**

{BT2, BT} == {`0b0`, `0b1100`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBNX, LBN, BAS} for this breakpoint are ignored.

`{0b0, 0b1101}`**, CONTEXTIDR_EL2 Match breakpoint with linking enabled**

{BT2, BT} == {`0b0`, `0b1101`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`{0b0, 0b1110}`**, Full Context ID Match breakpoint**

{BT2, BT} == {`0b0`, `0b1110`} is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1[31:0] is compared with the Context ID value

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

held in CONTEXTIDR_EL1, and the value of DBGBVR<n>_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBNX, LBN, BAS} for this breakpoint are ignored.

`{0b0, 0b1111}`, **Full Context ID Match breakpoint with linking enabled**

{BT2, BT} == `{0b0, 0b1111}` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match, as described in Breakpoint context comparisons.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR<n>_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`{0b1, 0b0000}`, **Unlinked Address Match breakpoint with linking enabled**

{BT2, BT} == `{0b1, 0b0000}` is a reserved value if the breakpoint does not support being configured as an address matching breakpoint with linking enabled.

Otherwise, one of the following applies:

- If no Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Watchpoint exception by an instruction depends on the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful instruction address match, as described in Breakpoint instruction address comparisons.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, PMC} for this breakpoint are ignored.

`{0b1, 0b0001}`, **Linked Address Match breakpoint with linking enabled**

{BT2, BT} == `{0b1, 0b0001}` is a reserved value if the breakpoint does not support being configured as an address matching breakpoint with linking enabled.

Otherwise, one of the following applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If no Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Watchpoint exception by an instruction depends on the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful instruction address match defined by this breakpoint, as described in Breakpoint instruction address comparisons.
  - A successful context match defined by the Context matching breakpoint with linking enabled that this breakpoint links to.

DBGBCR<n>_EL1.{LBNX, LBN} for this breakpoint selects the Context matching breakpoint with linking enabled that this breakpoint links to.

DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} for this breakpoint are ignored.

`{0b1, 0b0100}`, **Unlinked Address Mismatch breakpoint with linking enabled**

{BT2, BT} == {`0b1`, `0b0100`} is a reserved value if one of the following applies:

- The breakpoint does not support being configured as an address matching breakpoint with linking enabled.
- FEAT_BWE is not implemented.

Otherwise, one of the following applies:

- If no Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Watchpoint exception by an instruction depends on the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful instruction address match, as described in Breakpoint instruction address comparisons.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, PMC} for this breakpoint are ignored.

`{0b1, 0b0101}`, **Linked Address Mismatch breakpoint with linking enabled**

{BT2, BT} == {`0b1`, `0b0101`} is a reserved value if one of the following applies:

- The breakpoint does not support being configured as an address matching breakpoint with linking enabled.
- FEAT_BWE is implemented.

Otherwise, one of the following applies:

- If no Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Watchpoint exception by an instruction depends on the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful instruction address match defined by this breakpoint, as described in Breakpoint instruction address comparisons.
  - A successful context match defined by the Context matching breakpoint with linking enabled that this breakpoint links to.

DBGBCR<n>_EL1.{LBNX, LBN} for this breakpoint selects the Context matching breakpoint that this breakpoint links to.

DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} for this breakpoint are ignored.

Note

See Reserved DBGBCR<n>_EL1.{BT2, BT} values for the behavior of breakpoints programmed with reserved {BT2, BT} values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.8.4 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it generates Breakpoint exceptions only for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at EL0 in Secure state.

DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} defines the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

Security State Control Extension, SSCE, and Security State Control, SSC

SSCE and SSC control whether the breakpoint generates Breakpoint exceptions in only one Security state or in multiple Security states.

Note

This is determined by the Security state of the PE, not from the physical address space attribute returned by the translation of the virtual address on which the breakpoint is set.

Higher Mode Control, HMC, and Privileged Mode Control, PMC

HMC and PMC together control which Exception levels the breakpoint generates Breakpoint exceptions in.

Table D2-8 shows the valid combinations of the values of HMC, SSCE, SSC, and PMC, and for each combination shows which Exception levels breakpoints generate Breakpoint exceptions in.

In the table:

NS Non-secure state.

S Secure state.

RL In implementations that include FEAT_RME, Realm state.

RT In implementations that include FEAT_RME, Root state.

Y or - Means that a breakpoint programmed with the values of HMC, SSCE, SSC, and PMC shown in that row:

- **Y** Can generate Breakpoint exceptions in that Exception level and Security state.
- **-** Cannot generate Breakpoint exceptions in that Exception level and Security state.

Table D2-8 Summary of breakpoint HMC, SSCE, SSC, and PMC encodings

HMC	SSCE	SSC	PMC	Security states	EL3[a]	EL2	EL1	EL0
0	0	00	01	RL, S, NS	-	-	Y	-
0	0	00	10		-	-	-	Y
0	0	00	11		-	-	Y	Y
0	0	01	01	NS	-	-	Y	-
0	0	01	10		-	-	-	Y
0	0	01	11		-	-	Y	Y
0	0	10	01	S	-	-	Y	-
0	0	10	10		-	-	-	Y
0	0	10	11		-	-	Y	Y

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HMC	SSCE	SSC	PMC	Security states	EL3[a]	EL2	EL1	EL0
0	0	11	00	S	-	Y	-	-
0	0	11	01		-	Y	Y	-
0	0	11	11		-	Y	Y	Y
0	1	01	01	RL	-	-	Y	-
0	1	01	10		-	-	-	Y
0	1	01	11		-	-	Y	Y
1	0	00	01	RT, RL, S, NS	Y	Y	Y	-
1	0	00	11		Y	Y	Y	Y
1	0	01	00	NS	-	Y	-	-
1	0	01	01		-	Y	Y	-
1	0	01	11		-	Y	Y	Y
1	0	10	00	RT or S[b]	Y	-	-	-
1	0	10	01	RT, S	Y	Y	Y	-
1	0	10	11		Y	Y	Y	Y
1	0	11	00	RL, S, NS	-	Y	-	-
1	0	11	01		-	Y	Y	-
1	0	11	11		-	Y	Y	Y
1	1	01	00	RL	-	Y	-	-
1	1	01	01		-	Y	Y	-
1	1	01	11		-	Y	Y	Y

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSCE, SSC, and PMC are relevant only if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSCE, SSC, and PMC that generate Breakpoint exceptions at EL3 using AArch64.

b. If FEAT_RME is implemented, this combination specifies Root state. Otherwise, Secure state.

All combinations of HMC, SSCE, SSC, and PMC that Table D2-8 does not show are reserved. A combination in Table D2-8 might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSCE, SSC and PMC are reserved if an Exception level or Security state are not implemented, See Reserved DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} values.

D2.8.5 Breakpoint instruction address comparisons

In this subsection, the term `AddrTop` represents the most significant bit of a virtual address used by breakpoint data address comparisons. `AddrTop` is:

- 55, if address tagging is used for the address. See Address tagging.
- 63, otherwise.

Note

When stage 1 translation is enabled, in AArch64 state, the virtual address size is determined by the configured input address size for the stage 1 translation. Software can configure a smaller address width for a virtual address, see Input address size configuration. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.
When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED, as described in Implemented physical address size. These faults have a higher priority than breakpoints.

An address comparison is successful if bits [`AddrTop`:2] of the current instruction virtual address are equal to DBGBVR<n>_EL1[`AddrTop`:2].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

DBGBVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits. DBGBVR<n>_EL1[1:0] are RES0 and are ignored.

If EL1 is using AArch64 and EL0 is using AArch32, T32 and A32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, the instruction addresses are zero-extended before comparison with the breakpoint.

D2.8.5.1 Specifying the halfword-aligned address that an Address breakpoint matches on

For Address Match breakpoints, if the implementation supports AArch32 state, a debugger must program the *Byte Address Selection* field, DBGBCR<n>_EL1.BAS.

Table D2-9 Programmable BAS values

BAS	Match instruction at	Constraint for debuggers
`0b0011`	DBGBVR<n>_EL1	Use for T32 instructions.
`0b1100`	DBGBVR<n>_EL1 + 2	Use for T32 instructions.
`0b1111`	DBGBVR<n>_EL1	Use for A64 and A32 instructions.

If the implementation is an AArch64-only implementation, all instructions are word-aligned and DBGBCR<n>_EL1.BAS is RES1.

Figure D2-2 shows a summary of when Address Match breakpoints programmed with particular BAS values generate Breakpoint exceptions. The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example, row 7 shows that a breakpoint with DBGBCR<n>_EL1.BAS programmed as either `0b0011` or `0b1111` generates Breakpoint exceptions for A64 instructions. A64 instructions are always at word-aligned addresses.

Note

To breakpoint on an A64 instruction, Arm recommends that the debugger programs DBGBCR<n>_EL1.BAS as `0b1111`.

In the figure:

Yes Means that the breakpoint generates a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Instruction set	Size	Location of instruction[a]								BAS[3:0]		
			-2	-1	0	+1	+2	+3	+4	+5	0b0011	0b1100	0b1111
Row 1	T32	16-bit			■	■					Yes	No	Yes
Row 2		16-bit					■	■			No	Yes	UNP
Row 3	T32	32-bit	■	■	■	■					UNP	No	UNP
Row 4		32-bit			■	■	■	■			Yes	UNP	Yes
Row 5		32-bit					■	■	■	■	No	Yes	UNP
Row 6	A32	32-bit			■	■	■	■			Yes	UNP	Yes
Row 7	A64	32-bit			■	■	■	■			Yes	UNP	Yes

a. 0 means the word-aligned address held in the DBGBVR<n>_EL1[maxAddressSize:2]:00.
The other locations are as follows:
- -2 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) - 2).
- -1 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) + 5).

The solid areas show the location of the instruction.

Figure D2-2 Summary of BAS field meanings for Address Match breakpoints

D2.8.5.2 Address Mismatch breakpoints

If FEAT_BWE is not implemented, then mismatch breakpoints do not generate a breakpoint match in AArch64 state.

If FEAT_BWE is implemented, then mismatch breakpoints can generate a breakpoint match in AArch64 state and this can cause entry to Debug state.

The behavior of an address mismatch breakpoint is the same as for an address match breakpoint, except that the result is inverted such that only addresses other than the specified address, or addresses outside of the range of addresses, generate a breakpoint match.

Other breakpoint conditions specified by DBGBCR<n>_EL1 are not inverted.

D2.8.5.3 Programming a breakpoint range with eight or more bytes

A debugger can use the DBGBCR<n>_EL1.MASK field to program a single breakpoint with an instruction address range if all of the following are true:

- It is a size that is:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

If *m* least significant address bits are masked, the breakpoint generates a Breakpoint exception on all of the following:

- Address DBGBCR<n>_EL1[`AddrTop`:*m*]:000...
- Address DBGBCR<n>_EL1[`AddrTop`:*m*]:111...
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Breakpoint exceptions are generated for all addresses between DBGBCR<n>_EL1[`AddrTop`:4]:0000 and DBGBCR<n>_EL1[`AddrTop`:4]:1111, including these addresses.

When masking address bits, a debugger must both:

- Program DBGBCR<n>_EL1.BAS to be `0b1111`. See Programming dependencies of the BAS and MASK fields.
- In DBGBVR<n>_EL1, set the masked address bits to zero. See Other usage constraints.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.8.6 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<n>_EL1.BT determines what context comparison is required, if any. Table D2-10 shows the BT values that require a comparison, and the match required for the comparison to be successful.

Table D2-10 Breakpoint Context ID and VMID comparison tests

DBGBCR<n>.BT	Test required for successful context comparison
0b001x	• When FEAT_VHE is implemented, EL2 is using AArch64, the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID value. • Otherwise, CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value.
0b011x	CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value.
0b100x	VTTBR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value.
0b101x	CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value and VTTBR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value.
0b110x	CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32].
0b111x	Both: • CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value, DBGBVR<n>_EL1[31:0]. • CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32].

No Context ID or VMID comparison is required for other valid DBGBCR.BT values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR_EL1 and any of:
  - The PE is executing at EL3 using AArch64.
  - The PE is executing at EL2.
  - FEAT_VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- The comparison uses the value of CONTEXTIDR_EL2 and any of:
  - Neither FEAT_VHE is implemented, nor FEAT_Debugv8p2 is implemented.
  - The PE is in Secure state, and either FEAT_SEL2 is not implemented, or Secure EL2 is disabled.
  - The PE is executing at EL3.
  - EL2 is using AArch32.
  - EL2 is not implemented.
- The comparison uses the current VMID value and any of:
  - EL2 is not implemented.
  - The PE is in Secure state, and either FEAT_SEL2 is not implemented, or Secure EL2 is disabled.
  - The PE is executing at EL2.
  - The PE is executing at EL3.
  - FEAT_VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Note

- For all Context breakpoints, DBGBCR<n>_EL1.BAS is RES1 and is ignored.
- For Context matching breakpoints with linking enabled, DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, PMC} are RES0 and are ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.8.7 Breakpoint usage constraints

See the following sections:

- Reserved DBGBCR<n>_EL1.{BT2, BT} values.
- Reserved DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} values.
- Programming dependencies of the BAS and MASK fields.
- Reserved DBGBCR<n>_EL1.BAS values.
- Usage constraints on DBGBCR<n>_EL1.{LBNX, LBN} values.
- Reserved DBGWCR<n>_EL1.MASK values.
- Other usage constraints for Address breakpoints.
- Other usage constraints for Context breakpoints.

D2.8.7.1 Reserved DBGBCR<n>_EL1.{BT2, BT} values

Table D2-11 shows when particular DBGBCR<n>_EL1.BT values are reserved.

Table D2-11 Reserved BT values when the Effective value of DBGBCR<n>_EL1.BT2 is 0

BT value	Breakpoint type	Reserved
`0b001x`	Context ID Match	If the breakpoint is not *context-aware*.
`0b010x`	Address Mismatch	If FEAT_BWE is not implemented and one of the following: • In stage 1 of an AArch64 translation regime. • EDSCR.HDE is 1 and halting is allowed.
`0b011x`	CONTEXTIDR_EL1 Match	If FEAT_VHE is not implemented, or the breakpoint is not *context-aware*.
`0b100x`	VMID Match	If EL2 is not implemented, or the breakpoint is not *context-aware*.
`0b101x`	Context ID and VMID Match	If EL2 is not implemented, or the breakpoint is not *context-aware*.
`0b110x`	CONTEXTIDR_EL2 Match	If FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, or if the breakpoint is not *context-aware*.
`0b111x`	Full Context ID Match	

If FEAT_ABLE is implemented and BT2 is 1, all values of BT other than `0b0x0x` are reserved.

If a breakpoint is programmed with one of these reserved {BT2, BT} values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a {BT2, BT} value that is not reserved, other than for a direct or external read of DBGBCR<n>_EL1.
- For a direct or external read of DBGBCR<n>_EL1, if the reserved {BT2, BT} value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the {BT2, BT} value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved {BT2, BT} values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.8.7.2 Reserved DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} values

Table D2-12 shows when particular combinations of DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} are reserved in stage 1 of an AArch64 translation regime.

Table D2-12 Reserved HMC, SSCE, SSC, and PMC combinations

HMC, SSCE, SSC, and PMC combination	Reserved
All combinations with SSCE set to 1.	When FEAT_RME is not implemented.
All combinations with HMC set to 0, SSCE set to 0, and SSC set to `0b01` or `0b10`.	When Secure state is not implemented.
All combinations with SSCE set to 0 and SSC set to `0b01` or `0b10`, except for the combination with HMC set to 1, SSCE set to 0, SSC set to `0b01`, and PMC set to `0b00`.	When EL3 is not implemented and EL2 is implemented.
Any combination with SSCE set to 0 where HMC or SSC is nonzero, except for the combination with HMC set to 1, SSCE set to 0, SSC set to `0b01`, and PMC set to `0b00`, or combinations when SSC is set to `0b11`.	When both of EL2 and EL3 are not implemented.
The combination with HMC set to 1, SSCE set to 0, SSC set to `0b11`, and PMC set to `0b00`.	When EL2 is not implemented.
The combinations with SSC set to `0b11` except the combination with HMC set to 1, SSCE set to 0, SSC set to `0b11` and PMC set to `0b00`.	When Secure EL2 is not implemented.
The combination with HMC set to 1, SSCE set to 0, SSC set to `0b01` and PMC set to `0b00`.	When Secure EL2 is not implemented.
Combinations not included in Table D2-8.	Always.

For all breakpoints except Context matching breakpoints with linking enabled, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Context matching breakpoint with linking enabled, then:

- The values of HMC, SSCE, SSC, and PMC are ignored.
- A direct or external read of DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} returns UNKNOWN values

The behavior of breakpoints with reserved combinations of HMC, SSCE, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.8.7.3 Programming dependencies of the BAS and MASK fields

When programming a breakpoint, a debugger must use either:

- The MASK field, to program the breakpoint with an address range that can be eight bytes to 2GB.
- The BAS field. Table D2-9 gives the valid values of the DBGBCR<n>_EL1.BAS field .

If the debugger uses the:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MASK field, it must program BAS to be `0b1111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If any of the following apply, then it is CONSTRAINED UNPREDICTABLE whether or not a Breakpoint exception is generated:

- DBGBCR<n>_EL1.MASK is programmed with a reserved value.
- DBGBCR<n>_EL1.MASK is a valid nonzero value and any of the following apply:
  - DBGBCR<n>_EL1.BAS is not `0b1111`, and AArch32 is supported at EL0.
  - DBGBVR<n>_EL1[(MASK-1):0] is nonzero.
  - DBGBCR<n>_EL1.{BT2, BT} is {`0b0`, `0b0x1x`} or {`0b0`, `0b1xxx`}, indicating a Context matching breakpoint.
- DBGBCR<n>_EL1.MASK is zero, DBGBCR<n>_EL1.{BT2, BT} is {`0b0`, `0b010x`}, indicating an address mismatch breakpoint without linking enabled, and AArch32 is not supported at EL1.

Breakpoint <n> does not match in AArch64 state. A stand-alone mismatch of a single address is not supported in AArch64 state.

When any of these conditions apply, the PE behaves as if one of the following applies:

- DBGBCR<n>_EL1.MASK has been programmed with a defined value, which might be `0b00000` (no mask), other than for a direct read of DBGBCR<n>_EL1.
- The breakpoint is disabled.

The UNPREDICTABLE behavior is constrained to the generation of Breakpoint and Watchpoint exceptions and debug events.

D2.8.7.4 Reserved DBGBCR<n>_EL1.BAS values

In an AArch64-only implementation, DBGBCR<n>_EL1.BAS for all breakpoints is RES1.

Otherwise:

For all Context breakpoints

DBGBCR<n>_EL1.BAS is RES1 and is ignored.

For all Address breakpoints

Table D2-9 gives the valid values of the DBGBCR<n>_EL1.BAS field.

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BAS value that is not reserved, other than for a direct or external read of DBGBCR<n>_EL1.
- A direct or external read of DBGBCR<n>_EL1.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

D2.8.7.5 Usage constraints on DBGBCR<n>_EL1.{LBNX, LBN} values

For Linked Address breakpoints

A Linked Address breakpoint must link to a *context-aware* breakpoint. For a Linked Address breakpoint, any DBGBCR<n>_EL1.{LBNX, LBN} value that is not for a *context-aware* breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not *context-aware*, then reads of DBGBCR<n>_EL1.{LBNX, LBN} return an UNKNOWN value and behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Disabled.
- Linked to an UNKNOWN *context-aware* breakpoint.

If a Linked Address breakpoint links to a breakpoint that is implemented and that is *context-aware*, but that is either not enabled or not programmed as a Context matching breakpoint with linking enabled, it behaves as if it is disabled.

For all other breakpoint types

DBGBCR<n>_EL1.{LBNX, LBN} reads UNKNOWN and its value is ignored.

D2.8.7.6 Other usage constraints for Address breakpoints

For all Address breakpoints

- DBGBVR<n>_EL1[1:0] are RES0 and are ignored.
- If the implementation supports AArch32 state:
  - For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
  - If DBGBCR<n>.BAS is `0b1111`, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[48:2]:00) + 2). For T32 instructions, Arm recommends that the debugger programs the BAS field with either `0b0011` or `0b1100`.

D2.8.7.7 Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n>_EL1 that are not used to specify Context ID or VMID are RES0 and are ignored.

For breakpoints with linking enabled

If no Linked Address breakpoints or Linked watchpoints link to a breakpoint with linking enabled, the breakpoint with linking enabled does not generate any Breakpoint exceptions.

D2.8.8 Preferred return address

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.8.9 Pseudocode description of Breakpoint exceptions taken from an AArch64 stage 1 translation regime

`AArch64_BreakpointValueMatch()` tests the value in DBGBVR<n>_EL1.

`AArch64_StateMatch()` tests the values in DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} and, if the breakpoint links to a breakpoint with linking enabled, also tests the Context matching breakpoint.

`AArch64_BreakpointMatch()` tests a committed instruction against all breakpoints.

`AArch64_CheckBreakpoint()` generates a Breakpoint exception if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.
- All of the conditions required for Breakpoint exception generation are met. See About Breakpoint exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

`AArch64_CheckBreakpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

`AArch64_BreakpointException()` is called to generate a Breakpoint exception.

These functions are defined in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing one of the following:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.
- At EL0 using AArch32 when FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

This section contains the following subsections:

- About Watchpoint exceptions.
- Watchpoint types and linking of watchpoints.
- Execution conditions for which a watchpoint generates Watchpoint exceptions.
- Watchpoint data address comparisons.
- Watchpoint behavior for certain instruction classes.
- Watchpoint usage constraints.
- Exception syndrome information, fault address information, and preferred return address.
- Pseudocode description of Watchpoint exceptions taken from AArch64 state.

D2.9.1 About Watchpoint exceptions

A *watchpoint* is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-64 watchpoints. In an implementation, ID_AA64DFR0_EL1.WRPs and EDDFR1.WRPs show how many are implemented.

If AArch32 is supported at EL1 or FEAT_Debugv8p9 is not implemented, then the PE does not implement more than 16 watchpoints.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>_EL1. This contains controls for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>_EL1. This holds the data virtual address used for watchpoint matching.

These registers are numbered, so that:

- DBGWCR0_EL1 and DBGWVR0_EL1 are for watchpoint number zero.
- DBGWCR1_EL1 and DBGWVR1_EL1 are for watchpoint number one.
- . . .
- DBGWCR<n-1>_EL1 and DBGWVR<n-1>_EL1 are for watchpoint number (n-1).

A watchpoint can:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Context matching breakpoint with linking enabled*, so that a Watchpoint debug event is generated only if the PE is in a particular context when the address match occurs.
- If FEAT_ABLE is implemented, link to an *Address matching breakpoint with linking enabled* so that a Watchpoint debug event is generated only if the address of the instruction that made the matching access was executed from a specific address or from a specified range of addresses.

  The Address matching breakpoint with linking enabled can also be linked to a Context matching breakpoint with linking enabled.
- If FEAT_BWE2 is implemented, link to an *Address mismatch breakpoint* so that a Watchpoint debug event is generated only if the address of the instruction that made the access was executed from an address that was not selected. See Watchpoint data address comparisons.
- Be programmed as an address match watchpoint or an address mismatch watchpoint. See Watchpoint data address comparisons.

A single watchpoint can be programmed to match or mismatch on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>_EL1.BAS, to select the bytes. See Programming a watchpoint range with eight bytes or fewer.
- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>_EL1.MASK, to program a watchpoint with eight bytes to 2GB. See Programming a watchpoint range with eight or more bytes.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint debug events is CONSTRAINED UNPREDICTABLE. See Programming dependencies of the BAS and MASK fields.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The conditions specified in the DBGWCR<n>_EL1 are met. This includes all of the following:
  - The watchpoint is enabled. See DBGWCR<n>_EL1.E.
  - The type of the access. That is, whether the access is a load, a store, or both. See DBGWCR<n>_EL1.LST.
  - The Security state, Exception level, and privilege of the access. See Execution conditions for which a watchpoint generates Watchpoint exceptions.
  - If the watchpoint links to a breakpoint with linking enabled, the comparison or comparisons made by the breakpoint. See Rules for linking watchpoints.

  Together, these define the *context* of the data access that the watchpoint matches.
- The comparison with the address held in the DBGWVR<n>_EL1 is successful. See Watchpoint data address comparisons.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.
- If the access is due to a System register access instruction executed at EL1 and transformed into a memory access by the mechanism described in Enhanced support for nested virtualization and one of the following is true:
  - EDSCR.HDE is set to 1 and halting is allowed.
  - Debug exceptions are enabled at EL2. For more information, see Enabling debug exceptions from the current Exception level and Security state.

If multiple watchpoints can generate a Watchpoint debug event for an access, then, for all watchpoints that are enabled and match the context of the data access:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Boolean equality comparison results of all the address match watchpoints are OR'd together to generate a first result.
- The Boolean inequality comparison results of all the address mismatch watchpoints are AND'd together to generate a second result.

A watchpoint debug event is generated if any of the following apply:

- The first result is true and all enabled watchpoints are address match watchpoints.
- The second result is true and all enabled watchpoints are address mismatch watchpoints.
- Both the first and second results are true, and both address match watchpoints and address mismatch watchpoints are enabled.

The pseudocode function `AArch64_CheckWatchpoint()` describes the details for combining the results of individual watchpoints to determine the final result of watchpoint match.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state. Otherwise, if debug exceptions are:

- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

Note

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions.
However, unless specified otherwise, the behavior described also applies if watchpoints are causing entry to Debug state.

The debug exception enable controls describes the enable controls for Watchpoint debug events.

D2.9.2 Accessing watchpoint System registers

R_{MLRFY} The rules in Accessing breakpoint System registers that apply to MDSELR_EL1.BANK when accessing breakpoint registers also apply to watchpoint registers. See R_{MLTYN}, R_{JGQGB}, and I_{HMNFZ}.

R_{YKKCH} The `MRS` and `MSR` System register names for accessing breakpoint control registers DBGWCR<n>_EL1 and breakpoint value registers DBGWVR<n>_EL1 are mapped to physical registers as follows:

- The register names DBGWCR0_EL1 through DBGWCR15_EL1 are mapped to the physical registers DBGWCR<BANK×16+0>_EL1 through DBGWCR<BANK×16+15>_EL1.
- The register names DBGWVR0_EL1 through DBGWVR15_EL1 are mapped to the physical registers DBGWVR<BANK×16+0>_EL1 through DBGWVR<BANK×16+15>_EL1.

I_{XMTTT} `MSR` and `MRS` instructions using the watchpoint System register names in R_{YKKCH} make indirect reads of MDSELR_EL1.BANK. This means that direct writes to MDSELR_EL1.BANK must be explicitly synchronized before a direct read or direct write using any of these System register names.

D2.9.3 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that:

- It is either:
  - Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
  - Linked to a breakpoint with linking enabled. In this case, the watchpoint is called a *Linked watchpoint*.
- And is either:
  - An address match watchpoint.
  - An address mismatch watchpoint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When a Linked watchpoint links to an Address matching breakpoint with linking enabled, the Linked watchpoint generates a Watchpoint exception only if the address of the instruction that made the matching access was executed from a specific address or in range of addresses.

When a Linked watchpoint links to a Context matching breakpoint with linking enabled, or a Linked watchpoint links to a Linked Address matching breakpoint with linking enabled that links to a Context matching breakpoint with linking enabled, the Linked watchpoint generates a Watchpoint exception only if the PE is in a particular context when the data address match occurs.

Example D2-3 Linking a watchpoint and an Address matching breakpoint

For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.
3. Link the watchpoint and the breakpoint together. A Watchpoint exception is generated only if both the data address matches and the VMID matches.

D2.9.3.1 Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can be linked to any breakpoint with linking enabled.
- A Linked watchpoint can link to any type of breakpoint with linking enabled. The *Linked Breakpoint Number* field, DBGWCR<n>_EL1.{LBNX, LBN}, for the Linked watchpoint specifies the particular breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>_EL1.{SSCE, SSC, HMC, PAC} for the Linked watchpoint defines the execution conditions that the watchpoint generates Watchpoint exceptions for. See Execution conditions for which a watchpoint generates Watchpoint exceptions.
  - DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} for the breakpoint with linking enabled are ignored.
- A Linked watchpoint cannot link to another watchpoint. The {LBNX, LBN} fields always specify a breakpoint.
- If any of the following apply, then the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE:
  - The Linked watchpoint is linked to a breakpoint that does not support linking.
  - FEAT_ABLE is not implemented and the Linked watchpoint is linked to a breakpoint that is not context aware.
  - FEAT_ABLE is not implemented and the Linked watchpoint is linked to an Address matching breakpoint.
  - The Linked watchpoint is linked to an unimplemented breakpoint.
  - The Linked watchpoint or Linked breakpoint is linked to a breakpoint that does not have linking enabled.

  See Watchpoint usage constraints.
- If the access is due to a System register access instruction executed at EL1 and transformed into a memory access by the mechanism described in Enhanced support for nested virtualization, and the watchpoint is linked to a Context matching breakpoint with linking enabled, or a Linked Address matching breakpoint with linking enabled that is linked to a Context matching breakpoint with linking enabled, then it is CONSTRAINED UNPREDICTABLE whether there is a watchpoint match.
- Multiple Linked watchpoints can link to a single breakpoint with linking enabled.

  Note

  Multiple Address breakpoints can also link to a single breakpoint with linking enabled. Breakpoint exceptions describes breakpoints.

- If an instruction address generates a breakpoint match for more than one enabled Address matching breakpoint with linking enabled, then it is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated.

Figure D2-1 shows an example of permitted watchpoint linking.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.4 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it generates Watchpoint exceptions only for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only for Non-secure EL2 accesses.

DBGWCR<n>_EL1.{SSCE, SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

Security State Control Extension, SSCE, and Security State Control, SSC

SSCE and SSC control whether the watchpoint generates Watchpoint exceptions in only one Security state or in multiple Security states.

Note

This is determined by the Security state of the PE, not from the physical address space attribute returned by the translation of the virtual address on which the watchpoint is set.

Higher Mode Control, HMC, and Privileged Access Control, PAC

HMC and PAC together control which Exception levels and privilege levels the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access and the Exception level at which the access was made:

- Unprivileged memory accesses generated by unprivileged memory access instructions are treated as unprivileged EL0 accesses.
- System register accesses executed at EL1 and transformed into a memory access by the mechanism described in Enhanced support for nested virtualization are treated as EL2 accesses.

Example D2-4

PAC condition for generating a Watchpoint exception

This means that, if the PE executes an unprivileged memory access instruction at EL1, any resulting Explicit Memory Effect that is unprivileged triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on EL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A64 unprivileged memory access instructions are `LDTR` and `STTR`.

Table D2-13 shows the valid combinations of HMC, SSCE, SSC, and PAC, and for each combination shows which Exception levels watchpoints generate Watchpoint exceptions in.

In the table:

NS Non-secure state.

S Secure state.

RL In implementations that include FEAT_RME, Realm state.

RT In implementations that include FEAT_RME, Root state.

Y or - Means that a watchpoint programmed with the values of HMC, SSCE, SSC, and PAC shown in that row:

- **Y** Can generate Watchpoint exceptions in that Exception level and Security state.
- **-** Cannot generate Watchpoint exceptions in that Exception level and Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-13 Summary of watchpoint HMC, SSCE, SSC, and PAC encodings

HMC	SSCE	SSC	PAC	Security states	EL3[a]	EL2	EL1	EL0
0	0	00	01	RL, S, NS	-	-	Y	-
0	0	00	10		-	-	-	Y
0	0	00	11		-	-	Y	Y
0	0	01	01	NS	-	-	Y	-
0	0	01	10		-	-	-	Y
0	0	01	11		-	-	Y	Y
0	0	10	01	S	-	-	Y	-
0	0	10	10		-	-	-	Y
0	0	10	11		-	-	Y	Y
0	0	11	00	S	-	Y	-	-
0	0	11	01		-	Y	Y	-
0	0	11	11		-	Y	Y	Y
0	1	01	01	RL	-	-	Y	-
0	1	01	10		-	-	-	Y
0	1	01	11		-	-	Y	Y
1	0	00	01	RT, RL, S, NS	Y	Y	Y	-
1	0	00	11		Y	Y	Y	Y
1	0	01	00	NS	-	Y	-	-
1	0	01	01		-	Y	Y	-
1	0	01	11		-	Y	Y	Y
1	0	10	00	RT or S[b]	Y	-	-	-
1	0	10	01	RT, S	Y	Y	Y	-
1	0	10	11		Y	Y	Y	Y
1	0	11	00	RL, S, NS	-	Y	-	-
1	0	11	01		-	Y	Y	-
1	0	11	11		-	Y	Y	Y
1	1	01	00	RL	-	Y	-	-
1	1	01	01		-	Y	Y	-
1	1	01	11		-	Y	Y	Y

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSCE, SSC, and PAC are relevant only if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSCE, SSC, and PAC that generate Watchpoint exceptions at EL3 using AArch64.

b. If FEAT_RME is implemented, this combination specifies Root state. Otherwise, Secure state.

All combinations of HMC, SSCE, SSC, and PAC that Table D2-13 does not show are reserved. A combination in Table D2-13 might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSCE, SSC and PAC are reserved if an Exception level or Security state are not implemented or enabled, see Reserved DBGWCR<n>_EL1.{SSCE, SSC, HMC, PAC} values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.5 Watchpoint data address comparisons

In this subsection, the term `AddrTop` represents the most significant bit of a virtual address used by watchpoint data address comparisons. `AddrTop` is:

- 55, if address tagging is used for the address. See Address tagging.
- 63, otherwise.

Note

When stage 1 translation is enabled, in AArch64 state, the virtual address size is determined by the configured input address size for the stage 1 translation. Software can configure a smaller address width for a virtual address. See Input address size configuration. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.
When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED, as described in Implemented physical address size. These faults have a higher priority than watchpoints.

Bits [`AddrTop`:2] of the data address are compared with DBGWVR<n>_EL1[`AddrTop`:2], taking into account all of the following:

- The size of the access. See Size of the data access.
- The bytes selected by DBGWVR<n>_EL1.BAS. See Programming a watchpoint range with eight bytes or fewer.
- Any address ranges indicated by DBGWVR<n>_EL1.MASK. See Programming a watchpoint range with eight or more bytes.

The comparison is successful if any of following are true:

- The watchpoint is an address match watchpoint, and the two values being compared match.
- The watchpoint is an address mismatch watchpoint, and the two values being compared do not match.

If EL1 is using AArch64 and EL0 is using AArch32, AArch32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, the 32-bit data address is zero-extended before comparison with the watchpoint.

Note

- DBGWVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits.
- DBGWVR<n>_EL1[1:0] are RES0 and are ignored.

D2.9.5.1 Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each data access must be taken into account. The address of every byte in a data access is compared, and:

- In the case of an address match watchpoint, the watchpoint generates a Watchpoint exception if any byte address matches. See Example D2-5.
- In the case of an address mismatch watchpoint, the watchpoint generates a Watchpoint exception if any byte address does not match. See Example D2-6.

Example D2-5

1. A debugger programs an address match watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint generates a Watchpoint exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D2-6

1. A debugger programs an address mismatch watchpoint to generate Watchpoint exceptions when any address other than the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint generates a Watchpoint exception.

The size of data accesses initiated by `DC ZVA` instructions is the DC ZVA block size that DCZID_EL0.BS defines.

The size of data accesses initiated by `DC IVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR_EL0.DminLine defines.
  - 2KB.
- A power-of-two.

For both of these instructions:

- The lowest address accessed by the instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.
- The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA, and DC GZVA instructions.

D2.9.5.2 Programming a watchpoint range with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>_EL1.BAS, selects bytes in the doubleword starting at the address contained in DBGWVR<n>_EL1.

If the address programmed into the DBGWVR<n>_EL1 is:

- Doubleword-aligned:
  - All eight bits of DBGWCR<n>_EL1.BAS are used.
- Word-aligned but not doubleword-aligned:
  - Only DBGWCR<n>_EL1.BAS[3:0] are used. In this case, DBGWCR<n>_EL1.BAS[7:4] are RES0.

An address match watchpoint generates a Watchpoint exception when an access accesses any byte address that DBGWCR<n>_EL1.BAS selects. An address mismatch watchpoint generates a Watchpoint exception when an access accesses any byte address that DBGWCR<n>_EL1.BAS does not select.

Table D2-14 Supported BAS values when the DBGWVR<n>_EL1 address alignment is doubleword

BAS value	Description
`0b00000000`	No byte selected.
BAS[0] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`000` selected.
BAS[1] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`001` selected.
BAS[2] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`010` selected.
BAS[3] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`011` selected.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS value	Description
BAS[4] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:100 selected.
BAS[5] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:101 selected.
BAS[6] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:110 selected.
BAS[7] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:111 selected.

Table D2-15 Supported BAS values when the DBGWVR<n>_EL1 address alignment is word

BAS value[a]	Description
0b00000000	No byte selected.
BAS[0] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:00 selected.
BAS[1] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:01 selected.
BAS[2] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:10 selected.
BAS[3] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:11 selected.

a. DBGWCR<n>_EL1.BAS[7:4] are RES0.

If the BAS field selects more than one byte, the bytes it selects must be contiguous. For watchpoint behavior when its BAS field selects non-contiguous bytes, see Other usage constraints.

When programming the BAS field with anything other than 0b11111111, a debugger must program DBGWCR<n>_EL1.MASK to 0b00000. See Programming dependencies of the BAS and MASK fields.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>_EL1[AddrTop:3]. See Example D2-5.

Example D2-7

The following are some example configurations of the BAS field:

- To program an address match watchpoint to generate a Watchpoint exception on the byte at address 0x1003, program:
  - DBGWVR<n>_EL1 with 0x1000.
  - DBGWCR<n>_EL1.BAS to be 0b00001000.
- To program an address match watchpoint to generate a Watchpoint exception on the bytes at addresses 0x2003, 0x2004 and 0x2005, program:
  - DBGWVR<n>_EL1 with 0x2000.
  - DBGWCR<n>_EL1.BAS to be 0b00111000.
- If the address programmed into the DBGWVR<n>_EL1 is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>_EL1.BAS to be 0b00001111.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>_EL1[31:3]:100 is accessed, program DBGWCR<n>_EL1.BAS to be 0b11110000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm deprecates programming a DBGWVR<n>_EL1 with an address that is not doubleword-aligned.

D2.9.5.3 Programming a watchpoint range with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>_EL1.MASK, to program a single watchpoint with a data address range. The range must meet all of the following criteria:

- It is a size that is:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked.

If *m* least significant address bits are masked, an address match watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>_EL1[`AddrTop`:*m*]:000. . .0.
- Address DBGWVR<n>_EL1[`AddrTop`:*m*]:111. . .1.
- Any address between these two addresses.

An address mismatch breakpoint generates a Watchpoint exception for any address outside of the range.

Example D2-8

If the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>_EL1[`AddrTop`:4]:0000 and DBGWVR<n>_EL1[`AddrTop`:4]:1111, including these addresses.

When masking address bits, a debugger must both:

- Program DBGWCR<n>_EL1.BAS to be `0b11111111`. See Programming dependencies of the BAS and MASK fields.
- In the DBGWVR<n>_EL1, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see Other usage constraints.

D2.9.6 Watchpoint behavior for certain instruction classes

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Prefetch memory instructions.
- All data cache maintenance instructions except `DC IVAC`.

Note

Despite their mnemonics, the DC ZVA, DC GVA, and DC GZVA instructions are not data cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an IMPLEMENTATION DEFINED subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache, address translation, and TLB operations are IMPLEMENTATION DEFINED.

The access size of the IMPLEMENTATION DEFINED instruction cache, address translation, and TLB operations which generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also the following subsections:

- Watchpoint behavior on accesses by Store-Exclusive instructions.
- Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA, and DC GZVA instructions.
- Watchpoint behavior on accesses by SVE and SME instructions.
- Watchpoint behavior on accesses by Allocation tag load and store instructions.

D2.9.6.1 Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

D2.9.6.2 Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA, and DC GZVA instructions

`DC ZVA`, `DC GVA` and `DC GZVA` operations can generate Watchpoint exceptions.

If DC IVAC is not treated as a `NOP`, it can generate Watchpoint exceptions.

`DC IVAC`, `DC ZVA`, `DC GZVA` and `DC GVA` operations are treated as data stores by DBGWCR<n>_EL1.LSC.

Note

For the size of data accesses performed by the `DC IVAC` instruction and the `DC ZVA` instruction, see Watchpoint data address comparisons. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

D2.9.6.3 Watchpoint behavior on accesses by SVE and SME instructions

R_{ZRWTZ} For SVE predicated vector load or store instructions which are not First-fault vector loads or Non-fault vector loads, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element*, a Watchpoint exception is generated.

R_{GLRCD} For SVE predicated non-contiguous vector load or store instructions, if the instruction performs an access due to an *Inactive element*, a Watchpoint exception is not generated.

R_{DYGMS} For SVE Non-fault vector load instructions, when the instruction performs an access, a Watchpoint exception is not generated other than as permitted by R_{LVLYV}.

R_{LKKQG} For SVE Non-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element*, the access is reported in the FFR.

R_{TLSCX} For SVE First-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to the *First active element*, a *Watchpoint event* is generated.

R_{XBBLW} For SVE First-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element* that is not the *First active element*, a Watchpoint exception is not generated and the access is reported in the FFR.

I_{CKZFP} Watchpoints are not a mechanism for preventing access to memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{ZHXGG} For SVE Non-fault and First-fault vector load instructions that do not generate a Watchpoint exception, an access that matches a configured watchpoint can return the data and set the appropriate FFR elements to FALSE.

D_{TDYYR} A *relaxed watchpoint access* is a memory access or set of contiguous memory accesses generated by any of:

- SIMD&FP load/store instruction when the PE is in Streaming SVE mode.
- SVE contiguous vector load/store instruction.
- SME load/store instruction.

R_{LVLYV} If all the following are true for a *relaxed watchpoint access*, then it is CONSTRAINED UNPREDICTABLE whether or not a watchpoint is triggered:

- The watchpoint matches a range where the lowest accessed address is rounded down to the nearest multiple of 16 bytes and the highest accessed address is rounded up to the nearest multiple of 16 bytes minus 1.
- The watchpoint does not match the range of the original access or set of contiguous accesses.
- Either the watchpoint generates a Watchpoint exception, or FEAT_EDHSR is implemented.

I_{YVSFL} If a Watchpoint exception is triggered by a match on a rounded access address range that would not have been triggered by the original access address range, then this may report a false-positive match. The debugger should attempt to detect and step over false-positive matches. The architecture does not permit missed, or false-negative matches.

D2.9.6.4 Watchpoint behavior on accesses by Allocation tag load and store instructions

If FEAT_MTE is implemented, an instruction that loads or stores an Allocation Tag is considered a load or store to each location in the associated Tag Granule for the purpose of triggering Watchpoint exceptions.

D2.9.7 Watchpoint usage constraints

See the following:

- Reserved DBGBCR<n>_EL1.{BT2, BT} values when using Watchpoints.
- Reserved DBGWCR<n>_EL1.{SSCE, SSC, HMC, PAC} values.
- Usage constraints on DBGWCR<n>_EL1.{LBNX, LBN} values.
- Programming dependencies of the BAS and MASK fields.
- Reserved DBGWCR<n>_EL1.BAS values.
- Reserved DBGWCR<n>_EL1.MASK values.
- Other usage constraints.

D2.9.7.1 Reserved DBGBCR<n>_EL1.{BT2, BT} values when using Watchpoints

It is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated if DBGBCR<n>_EL1.{BT2, BT} is a reserved value. See Reserved DBGBCR<n>_EL1.{BT2, BT} values .

D2.9.7.2 Reserved DBGWCR<n>_EL1.{SSCE, SSC, HMC, PAC} values

Table D2-16 shows when particular combinations of DBGWCR<n>_EL1.{SSCE, SSC, HMC, PAC} are reserved.

Table D2-16 Reserved SSCE, SSC, HMC, and PAC combinations

HMC, SSCE, SSC, and PAC combination	Reserved
All combinations with SSCE set to 1.	When FEAT_RME is not implemented.
All combinations with HMC set to 0, SSCE set to 0, and SSC set to `0b01` or `0b10`.	When Secure state is not implemented.
All combinations with SSCE set to 0 and SSC set to `0b01` or `0b10`, except for the combination with HMC set to 1, SSCE set to 0, SSC set to `0b01`, and PAC set to `0b00`.	When EL3 is not implemented and EL2 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HMC, SSCE, SSC, and PAC combination	Reserved
All combinations with SSCE set to 0 where HMC or SSC is nonzero, except for the combination with HMC set to 1, SSCE set to 0, SSC set to `0b01`, and PAC set to `0b00` or combinations with SSC set to `0b11`.	When both EL2 and EL3 are not implemented.
The combination with HMC set to 1, SSCE set to 0, SSC set to `0b11`, and PAC set to `0b00`.	When EL2 is not implemented.
The combinations with SSC set to `0b11` except the combination with HMC set to 1, SSCE set to 0, SSC set to `0b11`, and PAC set to `0b00`.	When Secure EL2 is not implemented.
The combination with HMC set to 1, SSCE set to 0, SSC set to `0b01`, and PAC set to `0b00`.	When Secure EL2 is not implemented.
Combinations not included in Table D2-13.	Always.

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct or external read of DBGWCR<n>_EL1.
- For a direct or external read of DBGWCR<n>_EL1, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of SSCE, SSC, HMC, and PMC is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSCE, SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.9.7.3 Usage constraints on DBGWCR<n>_EL1.{LBNX, LBN} values

For Linked Watchpoints

If any of the following are true for a Linked watchpoint <n>, then direct reads of DBGWCR<n>_EL1.{LBNX, LBN} return an UNKNOWN value and it is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated:

- The Linked watchpoint is linked to an unimplemented breakpoint.
- The Linked watchpoint is linked to a breakpoint which does not support linking.
- FEAT_ABLE is not implemented and the Linked watchpoint is linked to a breakpoint which is not context aware.

Otherwise, if any of the following are true for a Linked watchpoint <n>, then the Linked watchpoint behaves as if it is disabled:

- The Linked watchpoint is linked to an enabled breakpoint that does not have linking enabled.
- FEAT_ABLE is not implemented and the Linked watchpoint is linked to an enabled Address matching breakpoint.
- The Linked watchpoint is linked to a disabled breakpoint.

It is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated if an instruction address generates a breakpoint match for more than one enabled Address matching breakpoint with linking enabled.

For Unlinked Watchpoints For Unlinked watchpoints, DBGWCR<n>_EL1.{LBNX, LBN} reads UNKNOWN and its value is ignored.

D2.9.7.4 Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n>_EL1 the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is nonzero and a BAS field that is not set to `0b11111111`, then for each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception is generated.

D2.9.7.5 Reserved DBGWCR<n>_EL1.BAS values

The BAS field must be programmed with a value `Zeros(8-n-m):Ones(n):Zeros(m)`, where:

- `n` is a nonzero positive integer less-than-or-equal-to 8.
- `m` is a positive integer less-than 8.
- `n+m` is less-than-or-equal-to 8.

All other values are reserved.

Note

If `x` is zero, then `Zeros(x)` is an empty bitstring.

If DBGWVR<n>_EL1[2] is 1, DBGWCR<n>_EL1.BAS[7:4] are RES0 and are ignored.

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>_EL1.
- A direct or external read of DBGWCR<n>_EL1.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

D2.9.7.6 Reserved DBGWCR<n>_EL1.MASK values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`, other than for a direct or external read of DBGWCR<n>_EL1.
- A direct or external read of DBGWCR<n>_EL1.MASK returns an UNKNOWN value.

D2.9.7.7 Other usage constraints

For all watchpoints:

- DBGWVR<n>_EL1[1:0] are RES0 and are ignored.
- If DBGWCR<n>_EL1.MASK is nonzero, and any masked bits of DBGWVR<n>_EL1 are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>_EL1.LSC is `0b00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.8 Exception syndrome information, fault address information, and preferred return address

On taking a Watchpoint exception or entering Debug state for a Watchpoint debug event, the PE records the following:

- Information about the debug event or exception in a *syndrome register*. No information is recorded if the watchpoint causes the PE to enter Debug state and FEAT_EDHSR is not implemented.
- An address that the debugger can use to determine the memory location that caused the watchpoint, in a *Fault Address Register*. No address is recorded if the PE sets the FnV bit in the syndrome register to 1, as described in Fault address information.

The syndrome register and fault address register used are one of the following:

- For a Watchpoint exception taken to EL1, ESR_EL1 and FAR_EL1.
- For a Watchpoint exception taken to EL2, ESR_EL2 and FAR_EL2.
- For a Watchpoint debug event that causes entry to Debug state, EDHSR, if implemented, and EDWAR.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

See ISS encoding for an exception from a Watchpoint exception for more information.

Note

Watchpoint exceptions cannot be taken to EL3 using AArch64.

This section uses the following terms:

D_{RCTWV} **Naturally-aligned block of memory**

Means:

- A power-of-two size.
- No larger than the `DC ZVA` block size defined by the DCZID_EL0.BS field, if ESR_ELx.FnP or EDHSR.FnP is 0.
- No larger than the smallest implemented translation granule if ESR_ELx.FnP or EDHSR.FnP is 1.
- Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint, or a watchpointed address in the address range permitted by R_{LVLYV} in Watchpoint behavior on accesses by SVE and SME instructions.

The size of the naturally-aligned block of memory is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

D_{WNGCC} **Watchpointed address**

Is defined as follows:

- If all watchpoints are address match watchpoints, then a watchpointed address is any address for which at least one of the address match watchpoints generates a watchpoint match.
- If all watchpoints are address mismatch watchpoints, then a watchpointed address is any address for which all address mismatch watchpoints generate a watchpoint match.
- Otherwise, a watchpointed address is any address for which at least one of the address match watchpoints generates a watchpoint match and all address mismatch watchpoints generate a watchpoint match.

For more information, see the subsections that follow. These are:

- Exception syndrome information.
- Fault address information.
- Preferred return address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.8.1 Exception syndrome information

R_{SQDKJ} If an instruction generates a watchpoint match where the watchpointed data address or data address range is not accessed by the instruction, the PE:

- Sets ESR_ELx.WPF to 1, on taking a Watchpoint exception generated by the watchpoint match.
- If FEAT_EDHSR is implemented, sets EDHSR.WPF to 1, on entering Debug state on a Watchpoint debug event generated by the watchpoint match.

Otherwise, ESR_ELx.WPF or EDHSR.WPF is set to an IMPLEMENTATION DEFINED choice of 0 or 1.

Example D2-9

When R_{LVLYV} in Watchpoint behavior on accesses by SVE and SME instructions applies, a *relaxed watchpoint access* might generate a watchpoint match for a data address or data address range that the instruction does not access. Arm strongly recommends that ESR_ELx.WPF or EDHSR.WPF is set to 0 for all other cases.

R_{KSSHC} If a watchpoint matches an access that is due to a *relaxed watchpoint access*, then the PE:

- Sets ESR_ELx.FnV to an IMPLEMENTATION DEFINED value, 0 or 1, on taking a Watchpoint exception generated by the watchpoint match.
- If FEAT_EDHSR is implemented, sets EDHSR.FnV to an IMPLEMENTATION DEFINED value, 0 or 1, on entering Debug state on a Watchpoint debug event generated by the watchpoint match.

Otherwise, ESR_ELx.FnV or EDHSR.FnV is set to 0.

R_{RLWSF} When the PE sets ESR_ELx.FnV to 0 on taking a Watchpoint exception generated by a watchpoint match:

- If the lowest watchpointed address that is higher than or equal to the address recorded in FAR_ELx might not have been accessed by the instruction, other than as permitted by R_{LVLYV}, the PE sets ESR_ELx.FnP to 1.
- Otherwise, the PE sets ESR_ELx.FnP to 0.

R_{CJWYX} When FEAT_EDHSR is implemented and the PE sets EDHSR.FnV to 0 on entering Debug state on a Watchpoint debug event generated by a watchpoint match:

- If the lowest watchpointed address that is higher than or equal to the address recorded in EDWAR might not have been accessed by the instruction, other than as permitted by R_{LVLYV}, the PE sets EDHSR.FnP to 1.
- Otherwise the PE sets EDHSR.FnP to 0.

R_{TLMWP} When FEAT_Debugv8p2 is implemented and FEAT_Debugv8p9 is not implemented, the PE behavior on taking a Watchpoint exception is as follows:

- If the PE sets ESR_ELx.FnV to 1 or ESR_ELx.FnP to 1, then the PE sets ESR_ELx.WPTV to 1.
- If the address recorded in FAR_ELx is not within a naturally-aligned block of memory, then the PE sets ESR_ELx.WPTV to 1.
- Otherwise, the PE sets ESR_ELx.WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

R_{RLFQK} When FEAT_Debugv8p2 is implemented and FEAT_Debugv8p9 is not implemented, the PE behavior on entering Debug state on a Watchpoint debug event is as follows:

- If FEAT_EDHSR is implemented and the PE sets EDHSR.FnV to 1 or EDHSR.FnP to 1, then the PE sets EDHSR.WPTV to 1.
- If the address recorded in EDWAR is not within a naturally-aligned block of memory, where the naturally-aligned block of memory is defined by D_{RCTWV}, then the PE sets EDHSR.WPTV to 1.
- Otherwise, if FEAT_EDHSR is implemented, then the PE sets EDHSR.WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

R_{WVSZS} When FEAT_Debugv8p9 is implemented, on a watchpoint match the PE:

- Sets ESR_ELx.WPTV to 1 on taking a Watchpoint exception generated by the watchpoint match.
- Sets EDHSR.WPTV to 1 on entering Debug state on a Watchpoint debug event generated by the watchpoint match.

R_{PVYNL} On a watchpoint match generated by watchpoint <*n*>:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the PE sets ESR_ELx.WPTV to 1 on taking a Watchpoint exception generated by the watchpoint match, then ESR_ELx.WPT is set to <*n*>.
- If FEAT_EDHSR is implemented and the PE sets EDHSR.WPTV to 1 on entering Debug state on a Watchpoint debug event generated by the watchpoint match, then EDHSR.WPT is set to <*n*>.
- Otherwise, ESR_ELx.WPT or EDHSR.WPT is UNKNOWN.

R_{XQBQN} When an instruction generates multiple watchpoint matches and the PE sets ESR_ELx.WPTV to 1 on taking a Watchpoint exception generated by the watchpoint match, or FEAT_EDHSR is implemented and the PE sets EDHSR.WPTV to 1 on entering Debug state on a Watchpoint debug event generated by the watchpoint match, then the value <n> reported in ESR_ELx.WPT or EDHSR.WPT, is defined as follows:

- If all the matching watchpoints are address match watchpoints or all the matching watchpoints are address mismatch watchpoints, then <n> is any watchpoint that generates a watchpoint match.
- Otherwise, <n> is any address match watchpoint that generates a watchpoint match.

D2.9.8.2 Fault address information

D2.9.8.2.1 Address recorded for Watchpoint exceptions generated by Memory Copy and Memory Set instructions

R_{FMRLD} If FEAT_MOPS is implemented, when a watchpoint is triggered by a Memory Copy or Memory Set instruction, the address recorded in FAR_ELx or EDWAR is all of:

- If the watchpoint is triggered by a write made by a Memory Copy or Memory Set instruction, in the inclusive range between:
  - The lowest address written to by the instruction that triggered the watchpoint.
  - The highest watchpointed address written to by the instruction that triggered the watchpoint.
- If the watchpoint is triggered by a read made by a Memory Copy instruction, in the inclusive range between:
  - The lowest address read from by the instruction that triggered the watchpoint.
  - The highest watchpointed address read from by the instruction that triggered the watchpoint.
- If the watchpoint generates an entry to Debug state and FEAT_EDHSR is not implemented, within a naturally-aligned block of memory.
- Within the current translation granule of the watchpointed address.

D2.9.8.2.2 Address recorded for Watchpoint exceptions generated by zeroing and data cache maintenance instructions

R_{GBDMM} For a watchpoint triggered by a DC ZVA, DC GVA, DC GZVA, or STZGM instruction, or a Data Cache maintenance operation, the address recorded in FAR_ELx or EDWAR is both:

- Any address accessed by the instruction.
- Within a naturally-aligned block of memory.

Note

Despite their mnemonics, the DC ZVA, DC GVA, and DC GZVA instructions are not data cache maintenance instructions.

D2.9.8.2.3 Address recorded for Watchpoint exceptions generated by SVE and SME instructions

R_{GVKYZ} For a watchpoint triggered by an SVE contiguous load/store instruction or by an SME load/store instruction, the address recorded in FAR_ELx or EDWAR is both:

- In the inclusive range between:
  - The lowest address accessed by the vector instruction that triggered the watchpoint, or the lowest rounded address as permitted by R_{LVLYV} in Watchpoint behavior on accesses by SVE and SME instructions.
  - The highest watchpointed address accessed by the vector instruction that triggered the watchpoint, or the highest rounded address as permitted by R_{LVLYV}.
- If either the watchpoint generates an entry to Debug state and FEAT_EDHSR is not implemented, or FEAT_Debugv8p2 is not implemented, then the range is further restricted such that:
  - The range includes a watchpointed address.
  - The range does not include any Inactive elements.

  Otherwise, the range might include Inactive elements that have not been accessed by the instruction. See R_{RLWSF} and R_{CJWYX}. However, other than as permitted by R_{LVLYV}, the watchpoint was triggered by an Active element.
- Within a naturally-aligned block of memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.8.2.4 Address recorded for Watchpoint exceptions generated by other instructions

The address recorded by other instructions must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
  - The highest watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
- Within a naturally-aligned block of memory.

Example D2-10 Address recorded for a watchpoint programmed on `0x8019`

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte `0x8019`.

An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.

If the `DC ZVA` block size is:

- 32 bytes, the address that the PE records must be between `0x8004` and `0x8019` inclusive.
- 16 bytes, the address that the PE records must be between `0x8010` and `0x8019` inclusive.

D2.9.8.3 Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.9.9 Pseudocode description of Watchpoint exceptions taken from AArch64 state

`AArch64_WatchpointByteMatch()` tests an individual byte accessed by an operation.

`AArch64_StateMatch()` tests the values in DBGWCR<n>_EL1.{HMC, SSCE, SSC, PAC}, and if the watchpoint links to a breakpoint with linking enabled, also tests the breakpoint that the watchpoint links to.

`AArch64_WatchpointMatch()` tests the value in DBGWVR<n>_EL1.

`AArch64_CheckWatchpoint()` generates a `FaultRecord` that `AArch64_Abort()` raises a Watchpoint exception for if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.
- All of the conditions required for Watchpoint exception generation are met. See About Watchpoint exceptions.

Note

`AArch64_CheckWatchpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

`AArch64_WatchpointException()` is called to generate a Watchpoint exception.

These functions are defined in Pseudocode for AArch64 operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.10 Vector Catch exceptions

Vector Catch exceptions are not generated in AArch64 translation regimes.

Note

This means that they are never taken to EL1 using AArch64 and are supported only if at least EL1 using AArch32 is supported.

A debugger that is executing in EL2 using AArch64 can route Vector Catch exceptions to EL2 using AArch64. See Routing debug exceptions.

`AArch64_VectorCatchException()` is called to generate a Vector Catch exception.

Vector Catch exceptions describes Vector Catch exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.11 Software Step exceptions

The following subsections describe Software Step exceptions:

- About Software Step exceptions.
- Rules for setting MDSCR_EL1.SS to 1.
- The software step state machine.
- Entering the active-not-pending state.
- Behavior in the active-not-pending state.
- Entering the active-pending state.
- Behavior in the active-pending state.
- Stepping T32 IT instructions.
- Exception syndrome information and preferred return address.
- Additional considerations.
- Pseudocode description of Software Step exceptions.

D2.11.1 About Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can single-step instructions at a lower Exception level.

Operation is as follows:

1. A debugger:
   a. Enables software step by setting MDSCR_EL1.SS to 1. See The debug exception enable controls.
   b. Executes an exception return instruction, to branch to the instruction to be single-stepped in the software being debugged.
   c. Optionally configures the MDSTEPOP_EL1 register. See Behavior in the active-not-pending state.
2. The PE then:
   a. Executes the instruction to be single-stepped.
   b. Takes a Software Step exception on the next instruction, returning control to the debugger.

However, another exception might be generated while the instruction is being stepped. This exception is either:

- A synchronous exception that is generated by the instruction being stepped.
- An asynchronous exception that is taken before or after the instruction being stepped.

The PE can take a Software Step exception only if debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.

A state machine describes the behavior of software step, shown in The software step state machine.

Throughout this Software Step exceptions section, including in all subsections, EL_D means the Exception level that Software Step exceptions are targeting. Routing debug exceptions defines EL_D as the *debug target Exception level*.

D2.11.2 Rules for setting MDSCR_EL1.SS to 1

Debugger software must be executing in an Exception level and Security state that debug exceptions are disabled from when it sets MDSCR_EL1.SS to 1.

The Exception level that hosts the debugger software must be using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.11.3 The software step state machine

In Figure D2-3:

- The OS Lock is unlocked and `DoubleLockStatus()` == FALSE.
- The PE is not in Secure state with MDCR_EL3.SDD set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

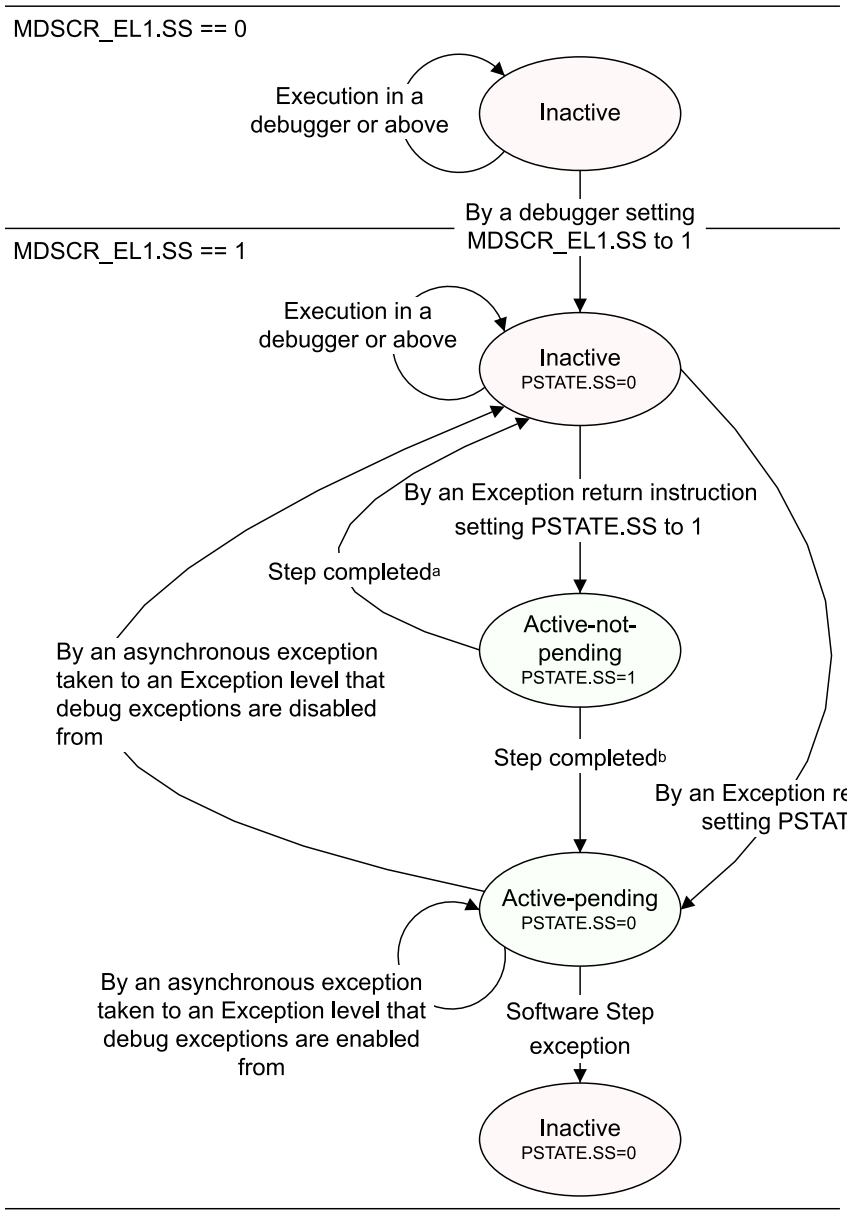

Execution is at either:

- An Exception level that is higher than EL^D.
- EL^D with (PSTATE.D == 1 || MDSCR_EL1.KDE == 0).

This is termed *execution in a debugger or above*.

To make the PE single-step an instruction, the debugger:

1. Sets SPSR_ELx.SS to 1.
2. Programs the ELR_ELx to point to the instruction to be stepped and optionally programs MDSTEPOP_EL1.
3. Executes an Exception return instruction.

Execution is in the software being debugged, at either:

- An Exception level that is lower than EL^D.
- EL^D with (PSTATE.D == 0 && MDSCR_EL1.KDE == 1).

Execution is in the software being debugged, at either:

- An Exception level that is lower than EL^D.
- EL^D with (PSTATE.D == 0 && MDSCR_EL1.KDE == 1).

A Software Step exception is pending.

Execution has returned to the debugger.

a. The step is the PE either:
   - Taking an exception to an Exception level that debug exceptions are disabled from.
   - If execution is at EL^D with MDSCR_EL1.KDE == 1, executing an instruction that sets PSTATE.D to 1.

   Software step is inactive when debug exceptions are disabled from the current Exception level, and debug exceptions are disabled from EL^D when PSTATE.D is 1.

b. The step is the PE either:
   - Executing the instruction to be stepped without taking an exception.
   - Taking an exception to an Exception level that debug exceptions are enabled from. The Exception level might be using AArch64 or AArch32.

c. Or, if execution is at EL^D with MDSCR_EL1.KDE == 1, by software setting PSTATE.D to 0.

Figure D2-3 Software step state machine

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a description of when debug exceptions are enabled or disabled from an Exception level, see Enabling debug exceptions from the current Exception level and Security state.

For more information about how a step is completed, see Behavior in the active-not-pending state.

The software step states are:

Inactive

Software step is inactive. It cannot generate any Software Step exceptions or affect PE execution. Software step is inactive whenever any of the following are true:

- MDSCR_EL1.SS is 0.
- EL_D is using AArch32.
- Debug exceptions are disabled from the current Exception level or Security state.

Active-not-pending

None of the conditions mentioned in Inactive are true, therefore software step is active.

The instruction to be stepped is usually the instruction in memory at the address specified by the PC. However, when FEAT_STEP2 is implemented, the instruction to be stepped might be specified in MDSTEPOP_EL1.

Active-pending

None of the conditions mentioned in Inactive are true, therefore software step is active.

A Software Step exception is pending on the current instruction.

Whenever software step is active, whether the state machine is in the active-not-pending state or the active-pending state depends on PSTATE.SS. Table D2-17 shows this.

Table D2-17 State machine states

EL_D using:	Debug exception enable status in the current Exception level and Security state	MDSCR_EL1.SS	PSTATE.SS	State machine state
AArch32	X	X	X	Inactive
AArch64	Disabled	X	X	Inactive
AArch64	Enabled	0	X	Inactive
AArch64	Enabled	1	1	Active-not-pending
AArch64	Enabled	1	0	Active-pending

D2.11.4 Entering the active-not-pending state

Software step can only enter the active-not-pending state from the inactive state.

Software step:

- Enters the active-not-pending state when an Exception return instruction writes 1 to PSTATE.SS, by copying from SPSR_ELx.SS when it restores PSTATE.
- Might enter the active-not-pending state on exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 1. See Exiting Debug state.

An Exception return instruction only copies 1 from SPSR_ELx.SS to PSTATE.SS if all of the following are true:

- MDSCR_EL1.SS is 1.
- EL_D is using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Debug exceptions are disabled from the current Exception level.
- Debug exceptions are enabled from the Exception level that the Exception return instruction targets.

Otherwise, Exception return instructions set PSTATE.SS to 0, regardless of the value of SPSR_ELx.SS.

Table D2-18 shows this. In the table:

Lock Means the value of (OSLSR_EL1.OSLK == '1' || `DoubleLockStatus()`).

NS Means the Effective value of SCR_EL3.NS.

SDD Means the Effective value of MDCR_EL3.SDD. See Disabling debug exceptions from Secure state.

EEL2 Means the Effective value of SCR_EL3.EEL2. If FEAT_SEL2 is not implemented, this is 0.

TGE Means the value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.

TDE Means the Effective value of MDCR_EL2.TDE. See Routing debug exceptions.

EL1 is using The Execution state when the EL_D is EL1.

EL2 is using The Execution state when the EL_D is EL2.

Table D2-18 Value an Exception return instruction writes to PSTATE.SS

MDSCR_EL1.SS	Lock	NS	SDD	EEL2	TGE	TDE	EL1 is using	EL2 is using	Value an Exception return instruction writes to PSTATE.SS
0	X	X	X	X	X	X	X	X	0
1	TRUE	X	X	X	X	X	X	X	0
	FALSE	0	1	X	X	X	X	X	0
			0	0	X	X	AArch32	n/a	0
							AArch64	n/a	See Table D2-19
				1	0	0	AArch32	n/a	0
							AArch64	AArch64	See Table D2-19
						1	AArch32	AArch32	0
							X	AArch64	See Table D2-20
					1	X	n/a	AArch32	0
							n/a	AArch64	See Table D2-20
		1	X	X	0	0	AArch32	n/a	0
							AArch64	AArch64	See Table D2-19
						1	AArch32	AArch32	0
							X	AArch64	See Table D2-20
					1	X	n/a	AArch32	0
							n/a	AArch64	See Table D2-20

For:

- If EL_D is EL1 using AArch64, Table D2-19 shows the value an Exception return instruction writes to PSTATE.SS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL_D is EL2 using AArch64, Table D2-20 shows the value an Exception return instruction writes to PSTATE.SS.

In both tables:

From EL Means the Exception level at which the PE executes the Exception return instruction.

Target EL Is the target Exception level of the Exception return instruction.

Note

If the Exception return instruction is an illegal exception return, the target Exception level of the Exception return instruction is the current Exception level. See Illegal exception returns from AArch64 state.

KDE Is MDSCR_EL1.KDE. See Enabling debug exceptions from the current Exception level and Security state

Table D2-19 Value an Exception return instruction writes to PSTATE.SS if ELD is EL1 using AArch64

From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	Software step enable status at: From EL	Software step enable status at: Target EL	Value an Exception return instruction writes to PSTATE.SS
EL3	EL3	X	X	X	Disabled	Disabled	0
	EL2	X	X	X	Disabled	Disabled	0
	EL1	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL3.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL3.SS
EL2	EL2	X	X	X	Disabled	Disabled	0
	EL1	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL2.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL2.SS
EL1	EL1	0	X	X	Disabled	Disabled	0
		1	0	X	Enabled[a]	-[b]	0
			1	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL1.SS
	EL0	0	X	X	Disabled	Enabled	SPSR_EL1.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL1.SS

a. Because MDSCR_EL1.SS == 1, it means that the Exception return instruction is itself being stepped.

b. Depends on SPSR_EL1.D.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-20 Value an Exception return instruction writes to PSTATE.SS if ELD is EL2 using AArch64

From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	Software step enable status at: From EL	Software step enable status at: Target EL	Value an Exception return instruction writes to PSTATE.SS
EL3	EL3	X	X	X	Disabled	Disabled	0
	EL2	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL3.SS
	EL1	X	X	X	Disabled	Enabled	SPSR_EL3.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL3.SS
EL2	EL2	0	X	X	Disabled	Disabled	0
		1	0	X	Enabled[a]	-[b]	0
			1	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL2.SS
	EL1	0	X	X	Disabled	Enabled	SPSR_EL2.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL2.SS
	EL0	0	X	X	Disabled	Enabled	SPSR_EL2.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL2.SS
EL1	EL1	X	X	X	Enabled[a]	Enabled	0
	EL0	X	X	X	Enabled[a]	Enabled	0

a. Because MDSCR_EL1.SS == 1, it means that the Exception return instruction is itself being stepped.

b. Depends on SPSR_EL2.D.

Note

No AArch32 instruction can set PSTATE.SS to 1.

D2.11.5 Behavior in the active-not-pending state

In this state, the PE does one of the following:

- Executes the instruction to be stepped and either:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception if the instruction generates one.
- Takes an asynchronous exception without executing any instructions.
- Enters Debug state because of a *Halting debug event*.

The instruction to be stepped is usually the instruction in memory at the address specified by the PC.

However, when FEAT_STEP2 is implemented, the instruction to be stepped might be specified in MDSTEPOP_EL1. See Software Step behavior when FEAT_STEP2 is implemented.

If the PE executes the instruction without taking any exceptions, then the PE sets PSTATE.SS to 0, meaning that after the instruction has been executed:

- If the instruction has disabled debug by setting PSTATE.D to 1 then software step advances to the inactive state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the instruction disables software step by a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS, then software step might advance to the inactive state. These writes require explicit synchronization to guarantee their effect. See Synchronization and the software step state machine.
- Otherwise, software step advances to the active-pending state. See Behavior in the active-pending state.

If the PE takes either a synchronous or an asynchronous exception, behavior is as described in one of the following:

- If the PE takes an exception to an Exception level that is using AArch64.
- If the PE takes an exception to an Exception level that is using AArch32.

If the PE enters Debug state because of a Halting debug event, behavior is as described in Entering Debug state and Software Step.

D2.11.5.1 If the PE takes an exception to an Exception level that is using AArch64

As part of exception entry, the PE does all of the following:

- Sets SPSR_ELx.SS to 0 or 1, depending on the exception. See Table D2-21.
- It is UNPREDICTABLE whether SPSR_ELx.SS is set to 0 or 1 when an SError exception is taken to ELx without executing the instruction.
- Sets PSTATE.SS to 0. This causes software step to enter either the active-pending state or the inactive state, depending on whether debug exceptions are enabled or disabled from the Exception level that the exception is taken to:

  Enabled Software step enters the active-pending state.

  Disabled Software step enters the inactive state.

  In either case, on taking the exception, a step is complete.
- Sets PSTATE.D to 1.

Table D2-21 Categorization of exceptions, for setting SPSR_ELx.SS to 0 or 1

Exception description	Exceptions	SPSR_ELx.SS
Exceptions whose preferred return address is for the instruction that follows the instruction to be stepped.	Supervisor Call (SVC) exceptions. Hypervisor Call (HVC) exceptions. Secure Monitor Call (SMC) exceptions.	0
Exceptions whose preferred return address is the address of the instruction to be stepped.	All other synchronous exceptions, and asynchronous exceptions that are taken before the instruction to be stepped.	1

Note

If an SMC instruction executed at Non-secure EL1 is trapped to EL2 because HCR_EL2.TSC is 1, the exception is a Trap exception, not a Secure Monitor Call exception, and so SPSR_ELx.SS is set to 1, not 0.

D2.11.5.2 If the PE takes an exception to an Exception level that is using AArch32

This can happen only when all of the following is true:

- EL2 is implemented and is using AArch64, and the Effective value of MDCR_EL2.TDE is 1. Because MDCR_EL2.TDE is 1, EL_D is EL2.
- The exception is taken to EL1 using AArch32.

As part of exception entry, the PE sets PSTATE.SS to 0. This causes software step to enter the active-pending state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Software step always enters the active-pending state because the exception is taken to an Exception level that debug exceptions are enabled from, EL1. Debug exceptions are enabled from EL1 because EL_D is EL2, and debug exceptions are always enabled from Exception levels that are lower than EL_D.
- AArch32 SPSRs have no SS bit.

D2.11.5.3 Software Step behavior when FEAT_STEP2 is implemented

R_{VTLLV} When FEAT_STEP2 is implemented, executing from MDSTEPOP_EL1 is enabled when all of the following are true:

- The PE is in AArch64 state.
- Software Step is in the active-not-pending state.
- MDSCR_EL1.EnSTEPOP is 1.
- Either EL3 is not implemented or MDCR_EL3.EnSTEPOP is 1.
- Any of the following apply:
  - EL2 is not implemented.
  - EL2 is disabled in the current Security state.
  - MDCR_EL2.EnSTEPOP is 1.

Otherwise, executing from MDSTEPOP_EL1 is disabled.

I_{WTPKX} The pseudocode function `SoftwareStepOpEnabled()` returns TRUE when executing from MDSTEPOP_EL1 is enabled and FALSE otherwise.

I_{SFGGB} When executing from MDSTEPOP_EL1 is disabled, the value in MDSTEPOP_EL1 is ignored.

R_{KPFLX} When executing from MDSTEPOP_EL1 is enabled, the instruction to be stepped is specified by the value of MDSTEPOP_EL1. That is, the PE executes an instruction from MDSTEPOP_EL1 instead of executing an instruction from memory.

R_{XTKVL} When the PE executes an instruction from MDSTEPOP_EL1, all of the following apply:

- The instruction address in the PC is checked by the PE as if the instruction is fetched from memory.
  For example, the instruction might generate any of the following exceptions based on the current value of PC:
  - PC alignment fault exception.
  - Instruction abort exception.
  - Breakpoint exception.
- A read of PC by the instruction returns the current value of PC.
- A write to PC updates the PC as normal.

I_{RJMRP} In R_{XTKVL}, reads of the PC includes indirect reads in the event of an exception or debug event being taken, as follows:

- If the instruction execution causes a synchronous exception, or is preempted by an asynchronous exception, then the preferred return address for the exception is based on the current PC value.
- If the instruction execution causes a synchronous entry to Debug state, or is preempted by an asynchronous entry to Debug state, then the preferred restart address for the debug event is based on the current PC value.

I_{VXWNT} When the PE executes an instruction from MDSTEPOP_EL1, it is CONSTRAINED UNPREDICTABLE whether the PE fetches an instruction from the memory location addressed by the current PC. This means that, in the case where the instruction fetch would generate a synchronous External abort, it is CONSTRAINED UNPREDICTABLE whether this exception will be generated. In the case where the instruction fetch would generate a synchronous External abort on a translation table fetch or translation table update, then the exception is generated because R_{XTKVL} requires that the translation table is checked.

If the PE does fetch an instruction from the memory location addressed by the current PC, then that instruction is ignored.

I_{LYLNF} Instruction execution properties that are derived from the current PC value use the same method whether the PE executes an instruction from MDSTEPOP_EL1 or from memory. For example, instruction execution properties that are derived from the PC being in a guarded page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Execution from MDSTEPOP_EL1 is not considered concurrent modification and execution of instructions. Tracing properties and performance monitoring of instruction execution from MDSTEPOP_EL1 are unchanged.

R_{DQFDD} The `MOVPRFX` instruction execution from MDSTEPOP_EL1 does not pair with another instruction.

I_{RDLQR} If a `BRK` instruction that is prefixed by a `MOVPRFX` instruction generates a Breakpoint Instruction exception then the ELR_ELx points to the PC of `BRK` instruction and the `MOVPRFX` instruction is required to have updated the architectural state. The debugger can use MDSTEPOP_EL1 to execute the original instruction that was replaced by the `BRK` instruction.

I_{XLQPG} Executing an instruction from MDSTEPOP_EL1 is an indirect read of MDSTEPOP_EL1.

D2.11.5.4 Summary of behavior in the active-not-pending state

Table D2-22 summarizes behavior in the active-not-pending state.

Table D2-22 Summary of behavior in the active-not-pending state

Event	Value written to PSTATE.SS	Target Exception level is using:	Details[a]	Value written to SPSR_ELx.SS	Next state
No exception	0	n/a	Disables Software step	n/a	Inactive
			Otherwise	n/a	Active-pending
Exception	0	AArch64	Supervisor Call (`SVC`) Hypervisor Call (`HVC`) Secure Monitor Call (`SMC`)	0	Active-pending or inactive[b]
			Other	1	
		AArch32	All	0	Active-pending

a. For the *No exception* rows, this column shows the effect of the event. For the *Exception* rows, this column shows the exception taken.

b. Which state software step enters depends on whether debug exceptions are enabled or disabled from the target Exception level. See Figure D2-3.

D2.11.6 Entering the active-pending state

Software step enters the active-pending state after any of the following operations, provided that both:

- MDSCR_EL1.SS is 1.
- Debug exceptions are enabled from the Exception level and Security state that execution is in after the operation.

The operations are:

While software step is in the active-not-pending state

The PE either:

- Executing the instruction to be stepped without taking any exceptions.
- Taking an exception.

While software step is in the active-pending state

The PE takes an asynchronous exception.

While software step is in the inactive state

The PE executes either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An Exception return instruction when SPSR_ELx.SS is 0.
- An instruction that enables debug by setting PSTATE.D to 0.

Note

If entry to the active-pending state is because of the PE taking an exception, it means that the exception is one that is taken to EL1 when MDCR_EL2.TDE is 1 and EL2 is implemented and enabled in the current Security state. Otherwise, debug exceptions are masked by PSTATE.D, therefore they would be disabled from the target Exception level of the exception.

In addition, software step might enter the active-pending state either:

- After a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS. These writes require explicit synchronization to guarantee their effect. See Synchronization and the software step state machine.
- On exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 0. See Exiting Debug state.

D2.11.7 Behavior in the active-pending state

When the PE is in the active-pending state, a Software Step exception is taken before the PE executes an instruction.

The Software Step exception has higher priority than all other types of synchronous exception. However, the prioritization of this exception with respect to any unmasked pending asynchronous exception is not defined by the architecture.

For more information, see the following:

- Prioritization of Synchronous exceptions taken to AArch64 state.
- Prioritization of asynchronous exceptions.
- Architectural requirements for taking asynchronous exceptions.

D2.11.8 Stepping T32 IT instructions

The A-profile architecture permits a combination of an `IT` instruction and another 16-bit T32 instruction to comprise one 32-bit instruction.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers this combination to be one instruction.
- The PE considers this combination to be two instructions.

In an implementation that supports the ITD control, which can disable some uses of the `IT` instruction, it is then IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD field. For example:

- The PE might consider this combination to be one instruction, regardless of the state of the applicable ITD field.
- The PE might consider this combination to be two instructions, regardless of the state of the applicable ITD field.
- The PE might consider this combination to be one instruction when the applicable ITD field is 1, and two instructions when it is 0.

The applicable ITD field is one of:

- SCTLR_EL1.ITD if execution is at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.ITD if execution is at EL0 or EL1 when EL1 is using AArch32.
- HSCTLR.ITD if execution is at Non-secure EL2 using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.11.9 Exception syndrome information and preferred return address

See the following:

- Exception syndrome information.
- Preferred return address.

D2.11.9.1 Exception syndrome information

On taking a Software Step exception, the PE records information about the exception in the Exception Syndrome Register (ESR_ELx) at the Exception level the exception is taken to. See ISS encoding for an exception from a Software Step exception for more information.

If no instruction was stepped because software step entered the active-pending state from the inactive state without passing through the active-not-pending state, then ESR_ELx.{ISV, EX} are set to {0, 0}.

When an instruction has been stepped, if the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code test, then ESR_ELx.ISV is set to 1 and ESR_ELx.EX is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1.

When an instruction has been stepped, if the stepped instruction was an Exception return instruction or an `ISB`, then ESR_ELx.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1, and ESR_ELx.EX is set to 0.

If the PE is executing at EL1 or EL0, and the Effective value of MDCR_EL2.TDE is 1, then a different exception might be taken to EL1 before the Software Step exception is taken to EL2. In this case, on taking the Software Step exception ESR_EL2.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1, and:

- If ESR_EL2.ISV is set to 1, then ESR_EL2.EX is set to the correct value for the instruction.
- If ESR_EL2.ISV is set to 0, then ESR_EL2.EX is set to 0.

Other than for the cases described above, when an instruction has been stepped:

- ESR_ELx.ISV is set to 1, to indicate that ESR_ELx.EX is valid.
- If the instruction stepped was a Load-Exclusive instruction, then ESR_ELx.EX is set to 1. Otherwise, ESR_ELx.EX is set to 0.

Note

If the PE cannot determine the correct value of ESR_ELx.EX for the stepped instruction, then it sets ESR_ELx.{ISV, EX} to {0, 0}. For example, the exception is taken before the PE decodes the stepped instruction, or the exception means the PE has no valid stepped instruction to decode.

Note

A Load-Exclusive instruction is any one of the following:

- In the A64 instruction set, any instruction that has a mnemonic starting with either `LDX` or `LDAX`.
- In the T32 and A32 instruction sets, any instruction that has a mnemonic starting with either `LDREX` or `LDAEX`.

Note

An implementation that always sets ISV to 0 and never sets EX is not compliant.

Table D2-23 summarizes the possible values that the PE can record in ESR_ELx.{ISV, EX}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-23 Values that the PE can record in ESR_ELx.{ISV, EX}

Description	ESR_ELx.ISV	ESR_ELx.EX
Syndrome data is not available because no instruction was stepped.	0	0
Syndrome data is available because an instruction was stepped. The instruction stepped was a conditional Load-Exclusive instruction that failed its Condition code test.	1	0 or 1
Syndrome data is available because an instruction was stepped. The instruction stepped was an Exception Return instruction or an `ISB`.	0 or 1	0
A different exception is taken before the Software Step exception.	0	0
	1	Set to the correct value for the instruction.
Syndrome data is available because an instruction was stepped. The instruction stepped was an instruction other than a Load-Exclusive instruction, an Exception Return instruction, or an `ISB`, and no other exception was taken before the Software Step exception.	1	0
Syndrome data is available because an instruction was stepped. The instruction stepped was a Load-Exclusive instruction that was either not conditional or did not fail its Condition code test.	1	1

D2.11.9.2 Preferred return address

The preferred return of a Software Step exception is the address of the instruction that was not executed because the PE took the Software Step exception instead.

D2.11.10 Additional considerations

This section contains the following:

- Behavior when an Exception return instruction is an illegal exception return.
- Behavior when the instruction stepped writes a misaligned PC value.
- Stepping code that uses Exclusives monitors.
- Synchronization and the software step state machine.

D2.11.10.1 Behavior when an Exception return instruction is an illegal exception return

If the conditions for entering the active-not-pending state in Entering the active-not-pending state are met, but the PE executes an Exception return instruction that is an illegal exception return, the exception return must be taken to the same Exception level that it was taken from. In this scenario, even though the Exception level remains the same before and after the Exception return instruction, software step can advance from the inactive state to one of the active states. Consider the following case:

1. MDSCR_EL1.SS is 1 and software step is inactive. The current Exception level is EL1 using AArch64, the OS Lock and OS Double Lock are unlocked, and MDCR_EL2.TDE is 0, MDSCR_EL1.KDE is 1, and PSTATE.D is 1.

   PSTATE.D == 1 is the reason why software step is inactive, because PSTATE.D == 1 means that debug exceptions are disabled from the current Exception level.
2. The PE executes an Exception return instruction.
3. The intended target of the Exception return instruction is EL2. This means that the Exception return instruction is an illegal exception return because the intended target is higher than the Exception level the Exception return instruction it is executed at. In this case, the Exception return instruction must target EL1 instead of EL2.

   If SPSR_EL1.D is 0, then on the Exception return instruction PSTATE.D becomes 0 and debug exceptions become enabled from the current Exception level. Software step therefore advances from the inactive state to one of the active states.

Which active state software step advances to depends on whether SPSR_ELx.SS is 1 or 0:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If SPSR_ELx.SS is 1, software step advances to the active-not-pending state.

  In this case, an Illegal Execution state exception is pending on the instruction to be stepped, and the PE takes the Illegal Execution state exception instead of executing the instruction to be stepped.
- If SPSR_ELx.SS is 0, software step advances to the active-pending state.

  In this case, a Software Step exception and an Illegal Execution state exception are both pending. The Software Step exception has higher priority. On taking the Software Step exception, the PE sets SPSR_ELx.IL to 1.

Note

Prioritization of Synchronous exceptions taken to AArch64 state shows the relative priorities of synchronous exceptions.

D2.11.10.2 Behavior when the instruction stepped writes a misaligned PC value

An indirect branch that writes a misaligned PC value might generate a PC alignment fault exception at the target of the branch. However, if the indirect branch is stepped using software step, the PE takes a Software Step exception instead, because the Software Step exception has higher priority. Behavior on returning from the Software Step exception depends on which Execution state the Exception level being returned to is using:

AArch64 A PC alignment fault exception is generated.

AArch32 The return from the Software Step exception forces the PC to the correct alignment, and no PC alignment fault exception is generated.

Debugger software must therefore take care when using software step to single-step an indirect branch instruction executed in AArch32 state, that it does not hide a PC alignment fault exception.

D2.11.10.3 Stepping code that uses Exclusives monitors

The A-profile architecture provides no mechanism for preserving the state of the Exclusives monitors when a Load-Exclusive or a Store-Exclusive instruction is stepped.

However, for certain progressions through the software step state machine, on taking a Software Step exception, the PE provides an indication of whether the instruction stepped was a Load-Exclusive instruction.

Debugger software can use this to detect the state of the Exclusives monitors. For example, if the PE reports that the instruction stepped was a Load-Exclusive instruction, the debugger is aware that the next Store-Exclusive operation will fail, because all Exclusives monitors are cleared on returning from the Software Step exception. The debugger must then take action to ensure that the code being stepped makes forwards progress.

For more information on how the PE reports whether the instruction stepped was a Load-Exclusive instruction, see Exception syndrome information and preferred return address.

D2.11.10.4 Synchronization and the software step state machine

Any of the following can cause transitions between software step states:

- A direct write to a System register.
- A direct write to a Special-purpose register.
- A write to an external debug register.

The software step state machine indirectly reads some of these registers and so is not guaranteed to observe any new values until after a Context Synchronization event has occurred.

Between a write to the register and the next Context Synchronization event, it is CONSTRAINED UNPREDICTABLE whether software step uses the state of the PE before the write, or the state of the PE after the write.

After a Context Synchronization event, the state machine must use the state of the PE after the write.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D2-11 Example of synchronization and software step state machine changing states

1. Software changes MDSCR_EL1.SS from 0 to 1 when debug exceptions are enabled.
2. The PE executes some instructions.
3. A Context Synchronization event occurs.

During step 2, it is CONSTRAINED UNPREDICTABLE whether software step remains in the inactive state, as if MDSCR_EL1.SS is 0, or enters the active-pending state because MDSCR_EL1.SS is 1. If it is in the:

- Inactive state, then after the Context Synchronization event, it must enter the active-pending state.
- Active-pending state, the PE might take a Software Step exception before the Context Synchronization event.

Note

A direct write to a Special-purpose register does not require explicit synchronization.

D2.11.11 Pseudocode description of Software Step exceptions

`SSAdvance()` advances software step from the active-not-pending state to the active-pending state, by setting PSTATE.SS to 0. It is called on completing execution of each instruction.

`CheckSoftwareStep()` checks whether software step is in the active-pending state, and if it is, generates a Software Step exception. It is called before each instruction executed, regardless of Execution state, before checking for any other synchronous exceptions.

`DebugExceptionReturnSS()` returns the value to write to PSTATE.SS on an exception return or an exit from Debug state. See Entering the active-not-pending state.

These functions are defined in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.12 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a Context Synchronization event has occurred. Similarly, the effect of the change on the software step state machine cannot be relied on until after a Context Synchronization event has occurred.

For any instructions executed between the time when the change is made and the time when the next Context Synchronization event occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example D2-12 Example of synchronization and Breakpoint exception generation

1. Software changes MDSCR_EL1.MDE from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A Context Synchronization event occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example D2-13 Example of synchronization and debug exceptions generation

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A Context Synchronization event occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

Note

For more information, see:

- Accessing PSTATE fields.
- Synchronization requirements for AArch64 System registers.
- Synchronization of changes to the external debug registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D3
AArch64 Self-hosted Trace

This chapter describes the AArch64 self-hosted trace:

Introductory information:

- About self-hosted trace.
- Trace sinks.
- Register controls to enable self-hosted trace.

Prohibited regions in trace:

- Controls to prohibit trace at Exception levels.
- Self-hosted trace and visibility of virtual data.

Timestamps and Synchronization:

- Self-hosted trace timestamps.
- Synchronization in self-hosted trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.1 About self-hosted trace

A trace unit generates trace data to describe the program flow of the PE.

The trace unit can be an implementation of any of the following:

- In Armv8, a standard Arm Embedded Trace Macrocell (ETM). See *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
- In Armv9, the Embedded Trace Extension (ETE). See The Embedded Trace Extension.
- An IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture trace unit that includes the ETM System register interface, FEAT_TRF must be implemented.

If an Armv9-compliant PE implements FEAT_ETE, FEAT_TRF must be implemented.

If any of the following are true, then Arm recommends that FEAT_TRF is implemented, but this is not mandatory:

- An Armv8.4-compliant PE or Armv9-compliant PE implements an IMPLEMENTATION DEFINED trace unit.
- An Armv8.4-compliant PE implements an ETM Architecture trace unit but does not implement the ETM System register interface.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

The trace unit and the PE must have the same view of the debug authentication interface. If FEAT_TRF is implemented, `ExternalNoninvasiveDebugEnabled()` is always TRUE.

D3.1.1 Trace sinks

The trace unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

If FEAT_TRBE is implemented, the PE includes a Trace Buffer Unit. Trace data is written directly to memory by the Trace Buffer Unit. See The Trace Buffer Extension.

In Armv8, Arm recommends that a system that includes FEAT_TRF incorporates an ETR, and follows the system architecture described by the *CoreSight Base System Architecture (CS-BSA)*.

D3.1.2 Register controls to enable self-hosted trace

R_{FHVLM} If FEAT_TRF is implemented, self-hosted trace is enabled if any of the following are true, and disabled otherwise:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, EL3 is implemented, MDCR_EL3.STE == 1, and `ExternalSecureNoninvasiveDebugEnabled()` == FALSE.
- EDSCR.TFO ==1, EL3 is not implemented, the PE executes in Secure state, and `ExternalSecureNoninvasiveDebugEnabled()` == FALSE.
- EDSCR.TFO ==1, FEAT_RME is implemented, MDCR_EL3.RLTE == 1, and `ExternalRealmNoninvasiveDebugEnabled()` == FALSE.

The pseudocode function `SelfHostedTraceEnabled()` shows these rules.

I_{ZSZXF} If FEAT_TRF is not implemented, self-hosted trace is disabled.

I_{RCXSF} While self-hosted trace is disabled, `ExternalNoninvasiveDebugAllowed()` controls whether tracing is prohibited or allowed in each Security state.

I_{KHRZF} The self-hosted trace extensions do not provide any mechanism to control software access to the trace unit external debug interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed()` indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

If self-hosted trace is enabled and EL3 is implemented, tracing is prohibited in Secure state when MDCR_EL3.STE == 0.

If self-hosted trace is enabled, tracing is prohibited in Realm state when MDCR_EL3.RLTE == 0.

If self-hosted trace is enabled, tracing is prohibited in Root state.

D3.2.1 Controls to prohibit trace at Exception levels

If self-hosted trace is enabled, TRFCR_EL1 and TRFCR_EL2 control whether trace is prohibited at an Exception level. While self-hosted trace is disabled, the registers TRFCR_EL1 and TRFCR_EL2 are ignored.

If self-hosted trace is enabled, tracing is prohibited at EL0 if one of the following is true:

- The Effective value of HCR_EL2.TGE == 0 and TRFCR_EL1.E0TRE == 0.
- The Effective value of HCR_EL2.TGE == 1 and TRFCR_EL2.E0HTRE == 0.

If self-hosted trace is enabled, tracing is prohibited at EL1 if TRFCR_EL1.E1TRE == 0.

If self-hosted trace is enabled, tracing is prohibited at EL2 if TRFCR_EL2.E2TRE == 0.

If self-hosted trace is enabled, tracing is prohibited at EL3 if one of the following is true:

- EL3 is using AArch64 state.
- EL3 is using AArch32 state and TRFCR.E1TRE == 0.

The pseudocode `TraceAllowed()` shows the above rules.

If FEAT_ETE is not implemented, when self-hosted trace is enabled, no events are exported to the trace unit when tracing is prohibited.

If FEAT_ETE is not implemented, when self-hosted trace is disabled, no events are exported to the trace unit when the PE is in Secure state and counting in Secure state is prohibited.

If FEAT_ETE is implemented, see External inputs for the rules governing export of events to the trace unit.

If FEAT_ETE is not implemented, when PMCR_EL0.X==0 or PMCR.X==0, no PMU events are exported to the trace unit.

Otherwise, PMU events are exported to the trace unit.

If self-hosted trace is enabled, Table D3-1 shows the prohibited regions by Exception level and state.

In the table:

RLTE Means the Effective value of MDCR_EL3.RLTE.

STE Means the Effective value of MDCR_EL3.STE or SDCR.STE, as applicable.

EEL2 Means the Effective value of SCR_EL3.EEL2.

TGE Means the Effective value of HCR_EL2.TGE.

P Means prohibited.

E2TRE Means prohibited if TRFCR_EL2.E2TRE == 0.

E1TRE Means prohibited if TRFCR_EL1.E1TRE == 0.

E0HTRE Means prohibited if TRFCR_EL2.E0HTRE == 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E0TRE Means prohibited if TRFCR_EL1.E0TRE == 0.

n/a Not applicable.

If FEAT_TRBE is not implemented, trace is allowed unless otherwise prohibited in Table D3-1.

If FEAT_TRBE is implemented, the requirements in this section are extended. See The owning translation regime.

Table D3-1 Prohibited regions

Controls						Tracing prohibited at			
State	**RLTE**	**STE**	**EL3 using**	**EEL2**	**TGE**	**EL3**	**EL2**	**EL1**	**EL0**
Non-secure	x	x	x	x	0	n/a	E2TRE	E1TRE	E0TRE
	x	x	x	x	1	n/a	E2TRE	n/a	E0HTRE
Secure	x	0	x	x	x	P	P	P	P
	x	1	AArch64	0	x	P	n/a	E1TRE	E0TRE
				1	0	P	E2TRE	E1TRE	E0TRE
				1	1	P	E2TRE	n/a	E0HTRE
	n/a		AArch32	x	x	E1TRE	n/a	n/a	E0TRE
Realm	0	x	x	x	x	n/a	P	P	P
	1	x	x	x	0	n/a	E2TRE	E1TRE	E0TRE
					1	n/a	E2TRE	n/a	E0HTRE
Root	x	x	x	x	x	P	n/a	n/a	n/a

D3.2.2 Self-hosted trace and visibility of virtual data

A hypervisor can use TRFCR_EL2.CX to control visibility of CONTEXTIDR_EL2 and VTTBR_EL2.VMID.

If self-hosted trace is enabled and TRFCR_EL2.CX == 0, or if EL2 is not implemented:

- The values of CONTEXTIDR_EL2 and VTTBR_EL2.VMID are not traced.
- Comparisons between CONTEXTIDR_EL2 and VTTBR_EL2.VMID do not match and results of comparison are not exposed through the comparators.

The trace unit can either prohibit trace for these values, or can record a CONTEXTIDR_EL2 or VTTBR_EL2.VMID value of zero in the trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.3 Self-hosted trace timestamps

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While self-hosted trace is disabled, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When self-hosted trace is enabled, the trace timestamp is one of the following:

- Physical time, which is defined by the physical count value returned by `PhysicalCountInt()`.
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of `PhysicalCountInt()` minus a physical offset.
  If any of the following are true, the value of physical offset is zero:
  — EL3 is using AArch32.
  — EL2 is not implemented.
  — FEAT_ECV_POFF is not implemented.
  — The Effective value of SCR_EL3.{NSE, NS, RW} is {0, 1, 0}.
  — CNTHCTL_EL2.ECV is 0.
  — EL3 is implemented and using AArch64, and SCR_EL3.ECVEn is 0.
  Otherwise the value of physical offset is the value of CNTPOFF_EL2.
- Virtual time, which is defined as the value of `PhysicalCountInt()` minus a virtual offset.
  If any of the following are true, the virtual offset is zero:
  — EL2 is not implemented.
  Otherwise, the virtual offset is always CNTVOFF_EL2, including when a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero.

The following fields control which counter is used for self-hosted trace:

- TRFCR_EL2.TS.
- TRFCR_EL1.TS.
- HTRFCR.TS.
- TRFCR.TS.

The timestamp used for trace is shown in Table D3-2.

Table D3-2 Timestamp used for trace.

`SelfHostedTraceEnabled()`	TRFCR_EL2.TS	TRFCR_EL1.TS	Timestamp traced
FALSE	xx	xx	CoreSight time
TRUE	`0b00`	`0b01`	`PhysicalCountInt()` - virtual offset
	`0b00`	`0b10`	`PhysicalCountInt()` - physical offset
	`0b00`	`0b11`	`PhysicalCountInt()`
	`0b01`	xx	`PhysicalCountInt()` - virtual offset
	`0b10`	xx	`PhysicalCountInt()` - physical offset
	`0b11`	xx	`PhysicalCountInt()`

Note

The counter value used for the trace timestamp is not affected by the Effective value of HCR_EL2.E2H, or whether EL2 is enabled or disabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.4 Synchronization in self-hosted trace

The trace unit is an indirect observer of the System registers.

If FEAT_TRBE is implemented, the requirements in this section are extended. See Synchronization and the Trace Buffer Unit.

While self-hosted trace is enabled, indirect reads of the trace filter control fields, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced. For these register fields, in addition to the standard requirements for synchronization of System register accesses, when a trace filter control value is changed and synchronization is not explicitly specified, one of the following occurs:

- The behavior of the PE must be consistent with the control value having the old value.
- The behavior of the PE must change the control value at a point in the simple sequential execution of the program, so that before that point, the behavior of the PE is consistent with the control value having the old value, and after that point the behavior of the PE is consistent with the control value having the new value.

If there are multiple direct writes to the register without explicit synchronization, the behavior is consistent with the writes occurring in program order.

The `TSB CSYNC` operation is used to ensure that a trace operation, due to a trace unit generating trace for an instruction has completed. The `TSB CSYNC` operation can be reordered with respect to other instructions, so must be combined with at least one Context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR_EL1 or TRFCR_EL2 is guaranteed to be observed by the trace unit only after a subsequent Context synchronization event. For more information, see Trace Synchronization Barrier.

While self-hosted trace is disabled, the trace unit might impose stronger synchronization requirements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D4
The Embedded Trace Extension

This chapter describes the *Embedded Trace Extension* (ETE). It contains the following sections:

- About the Embedded Trace Extension.
- Programmers' model.
- Trace elements.
- Instruction and exception classification.
- About the ETE trace unit.
- Resource operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.1 About the Embedded Trace Extension

IJTPNL FEAT_ETE describes the operation of a trace unit. The trace unit provides details about software control flow running on a PE which can be used to aid debugging or optimization. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

ILVKQS FEAT_ETE overlaps with the ETMv4 architecture, while also providing support for new architecture features. For more information on the ETMv4 architecture, see the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064)*. FEAT_ETE does not support all the features of ETMv4. For more information on the differences between FEAT_ETE and the ETMv4 architecture, see Differences between ETM and ETE.

D4.1.1 Attributes of tracing

IRGLSR The attributes of PE tracing are:

Trace is generated in real time.
Trace provides a means of observing the PE operation while the PE is running. For diagnostic purposes, this is useful as some types of erroneous behavior are only solvable by observing the system during runtime. In addition, because the PE trace can include cycle counts, trace can be used for PE profiling purposes.

Trace has a minimal effect on functional performance of the PE.
Usually, trace has no effect on the functional performance of the PE. This attribute does depend on the market use of the PE being debugged, however, and on the trace requirements for the PE and the trace solution that is adopted to meet those requirements. For some markets, some impact on PE performance is acceptable but for others, most notably in real-time systems, an impact on PE performance might be unacceptable.

Trace is available for self-hosted analysis.
The trace from a PE or process is available for analysis by software running on the target. See Self-hosted Trace.

Trace is deeply embedded in an SoC.
Trace provides a method of debugging software executing on PEs that are deeply embedded within an SoC.

Trace is available for external analysis.
The trace from a PE or process can be exported off chip for analysis by external tools.

D4.1.2 Self-hosted Trace

IFNMRL Self-hosted trace is used for various purposes, including:

Non-invasive single stepping:
The trace provides a history of execution similar to that obtained by single-stepping through code.

Failure logging:
This is similar to a stack trace dump when a failure occurs.

Performance analysis:
The trace might be used with other trace sources or performance analysis units to analyze program performance.

IVVSCJ Capturing the trace on-chip involves one of the following:

Use of system memory:
The trace output from the trace unit is directed to a buffer in main system memory through the Trace Buffer Unit defined by FEAT_TRBE.

Use of existing shared system memory, where some main system memory is reserved for trace capture:
The trace output from the trace unit is directed to the reserved memory over the main system bus, typically using CoreSight technology such as a CoreSight ETR.

Use of a dedicated on-chip buffer:
The trace output from the trace unit is directed to the dedicated memory, typically using CoreSight technology such as a CoreSight ETB. A dedicated bus such as AMBA ATB is also usually implemented between the trace unit and the dedicated memory. Use of dedicated memory means that PE tracing can be performed with zero or minimal effect on system behavior.
AMBA ATB is defined by the *AMBA® ATB Protocol Specification (ARM IHI 0032)*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.1.3 External debug

I_{TQGBH} External debug is commonly used in trace applications that require long-term logging of behavior. In addition, external debug is more likely to be used when the impact of PE tracing on system performance must be minimized.

For example, external debug might be used:

- For debugging real-time systems.
- When analyzing programs that do not frequently vary their behavior.
- For debugging software, where a history of execution is required up to the point of failure.

Exporting the trace off-chip usually involves one of the following methodologies:

D4.1.3.1 Real-time continuous export

I_{VJBMM} This can be done using either:

- A dedicated trace port capable of sustaining the bandwidth of the trace.
- An existing interface on the SoC, such as a USB or other high-speed port.

Use of a dedicated trace port means that the trace can be exported off-chip with zero or minimum effect on system behavior. An existing interface is usually used when system constraints, such as cost or package size, mean that a dedicated trace port is not possible. However, use of an existing interface might affect system behavior, because both trace and normal interface traffic use the same port.

D4.1.3.2 Short-term on-chip capture with subsequent low speed export

I_{PXSSB} This option is used when a low-cost method of exporting the trace is required, or when system constraints prevent real-time continuous export. The trace output from the trace unit is stored temporarily on-chip, and then exported using either:

- An existing debug port on the SoC, such as a JTAG-DP or SW-DP.
- Another existing interface on the SoC, such as USB.

Typically, the temporary storage is a circular buffer. If the buffer is full, newer trace overwrites older trace, which means that the buffer always contains the most recent trace. In SoCs that employ Arm CoreSight technology, a dedicated Embedded Trace Buffer (ETB) is provided for the on-chip capture of trace.

D4.1.4 Trace output

R_{SVDBD} The trace unit outputs the trace byte stream to one or more of the following:

- The TRBE.
- One or more IMPLEMENTATION DEFINED interfaces. An AMBA ATB interface to a CoreSight subsystem is an example of such an IMPLEMENTATION DEFINED interface.

R_{LGVCX} If the Trace Buffer Unit is enabled, the trace byte stream is output solely to the Trace Buffer Unit.

R_{FJFNS} If the Trace Buffer Unit is disabled, the trace byte stream is output to one or more of the other interfaces.

R_{HBBDV} The ETE Trace Output Enable is always implemented when FEAT_ETEv1p3 is implemented and an IMPLEMENTATION DEFINED trace output interface is present.

R_{RPHPJ} The ETE Trace Output Enable is optional when FEAT_ETE is implemented and an IMPLEMENTATION DEFINED trace output interface is present.

R_{FTSZG} When the ETE Trace Output Enable is not implemented and the Trace Buffer Unit is disabled, trace is output to an IMPLEMENTATION DEFINED trace output interface unless such output is disabled by an IMPLEMENTATION DEFINED mechanism.

R_{PJYKY} When the ETE Trace Output Enable is implemented and TRCEVENTCTL1R.OE is 0 or the Trace Buffer Unit is enabled, trace is never output to any IMPLEMENTATION DEFINED trace output interface.

R_{MMCKW} When the ETE Trace Output Enable is implemented and TRCEVENTCTL1R.OE is 1 and the Trace Buffer Unit is disabled, trace is output to an IMPLEMENTATION DEFINED trace output interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The addition of TRCEVENTCTL1R.OE is not backward compatible, and requires a trace analyzer to explicitly enable trace output to any IMPLEMENTATION DEFINED trace output interface.

I_{TVDMT} If the trace output is captured by a single capture point dedicated to a PE, and does not mix with any other trace streams, then the value programmed into TRCTRACEIDR.TRACEID does not need to be unique among all values programmed into all trace units in the system. For example, the value `0x01` could be programmed into all trace units that have their own dedicated trace capture point.

- One example of a dedicated trace capture point is the Trace Buffer Extension. When FEAT_TRBE is implemented and enabled, the value of TRCTRACEIDR.TRACEID can be the same for all PEs that are using the Trace Buffer Extension. For more information, see The Trace Buffer Extension.
- Another example of a dedicated trace capture point is a CoreSight ETR connected through an AMBA ATB interface and dedicated to a PE. If the AMBA ATB interface connects to the ETR without mixing with any other trace streams, TRCTRACEIDR.TRACEID does not need to be unique when using the ETR, and when ensuring the trace does not go to any other trace capture points.

R_{RWPFG} If an AMBA ATB interface is implemented, the trace unit must support all of the following:

- ATB triggers, indicated by TRCIDR5.ATBTRIG.
- A 7-bit trace ID, indicated by TRCIDR5.TRACEIDSIZE.

I_{QJKVW} Arm recommends that if the trace unit implements either an AMBA ATB interface, or an IMPLEMENTATION DEFINED interface for trace output, then the path from the interface is not affected by a Warm reset of the PE. This ensures tracing is possible through a Warm reset of the PE, which is useful for low level debugging scenarios.

R_{NLSSL} While all trace outputs are disabled, the trace unit considers any generated trace data as having been output.

D4.1.5 Trace sessions

R_{GLTKQ} At any one time, the trace unit is either enabled or disabled. For more information on the possible states of the trace unit, see Trace unit programming states.

R_{XBPSQ} A *trace session* is the period between the trace unit becoming enabled, and when the trace unit next becomes disabled.

R_{MTLFH} While the trace unit is enabled, the ViewInst() function is either *active* or *inactive*. While ViewInst() is *active*, the trace unit generates trace for instructions that are executed, unless trace generation is *inoperative*.

R_{MFNWB} Whether ViewInst() is *active* or *inactive* is independent of whether trace generation is *operative* or *inoperative*.

R_{ZVNKV} Trace generation is *operative* while neither of the following are true:

- The trace unit is disabled.
- The trace unit is recovering from a trace unit buffer overflow.

R_{RDPSW} Trace generation *becomes operative* when trace generation transitions from being *inoperative* to *operative*, and occurs when any of the following occur:

- When the trace unit transitions from being disabled to being enabled.
- When the trace unit recovers from a trace unit buffer overflow.

R_{BDRKW} Trace generation *becomes inoperative* when trace generation transitions from being *operative* to *inoperative*, and occurs either:

- When the trace unit transitions from being enabled to being disabled.
- When the trace unit encounters a trace unit buffer overflow.

R_{LDDLP} When the trace unit is unable to generate at least one trace packet which is required by the architecture, a *trace unit buffer overflow* occurs.

I_{HDJWW} A trace unit buffer overflow is typically caused when any buffering in the trace unit is unable to receive more trace packets. Inability to receive trace packets is often caused by being unable to sustain output of trace packets to any trace capture infrastructure.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A trace unit buffer overflow is independent of the Trace Buffer Unit filling or wrapping a trace buffer in memory. However a trace unit buffer overflow might be caused by the Trace Buffer Unit rejecting trace data due to insufficient capacity, and the limit of any trace unit internal buffers is subsequently reached.

D4.1.6 Trace unit programming states

R_{WMGGP} The trace unit is always in one of the states shown in Figure D4-1 and Table D4-1.

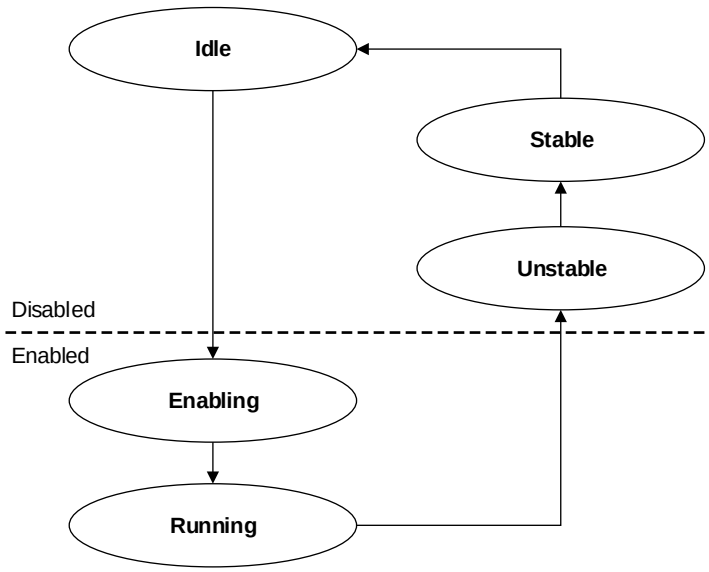

Figure D4-1 Trace unit programming states

Table D4-1 Trace unit programming states

State	TRCSTATR.IDLE	TRCSTATR.PMSTABLE	Trace unit enabled
Idle	`0b1`	`0b1`	No
Enabling	`0b1`	UNKNOWN	Yes
Running	`0b0`	UNKNOWN	Yes
Unstable	`0b0`	`0b0`	No
Stable	`0b0`	`0b1`	No

R_{DXFFN} When the trace unit becomes enabled, the trace unit transitions from the Idle state to the Enabling state.

R_{ZFPHC} The trace unit transitions from the Enabling state to the Running state in a finite amount of time.

R_{TZSRP} When the trace unit becomes disabled, the trace unit transitions from the Running state to the Unstable state.

R_{CZHWF} The trace unit transitions from the Unstable state to the Stable state in a finite amount of time.

R_{BPTKT} While the trace unit is in the Stable and Idle states, the states of the following fields do not change other than by direct writes or external writes:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCVICTLR.SSSTATUS.
- TRCSEQSTR.STATE.
- TRCCNTVR<n>.VALUE.
- TRCSSCSR<n>.STATUS.
- TRCRSR.EVENT.
- TRCRSR.EXTIN.
- TRCRSR.TA.

I_{TDLZL} The trace unit programmers' model state can be safely read when in any of the Stable or Idle states.

R_{TRZCP} When the trace unit becomes fully idle and both of the following are true, the trace unit transitions from the Stable state to the Idle state:

- The trace unit is drained of any trace.
- With the exception of the programming interfaces, all external interfaces on the trace unit are quiescent.

R_{RWYQD} While the trace unit is not in the Idle state, direct writes and external writes to the trace unit registers are CONSTRAINED UNPREDICTABLE, except for the following registers:

- TRCPRGCTLR.
- TRCCLAIMSET.
- TRCCLAIMCLR.

This CONSTRAINED UNPREDICTABLE behavior is one of the following:

- The write is ignored.
- The register takes an UNKNOWN value.

The trace byte stream might also be corrupted and analysis of the byte stream might be impossible.

R_{MDZDN} While the trace unit is not in the Idle or Running states, changing the value of TRCPRGCTLR.EN is CONSTRAINED UNPREDICTABLE.

This CONSTRAINED UNPREDICTABLE behavior is one of the following:

- The write is ignored.
- The register takes an UNKNOWN value.

I_{PRQRD} For more information, see:

- Behavior on enabling.
- Behavior on disabling.
- Access permissions on the corresponding register page.

I_{FJPCN} Figure D4-2 shows the procedure that must be used when programming the trace unit registers using the external debug interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

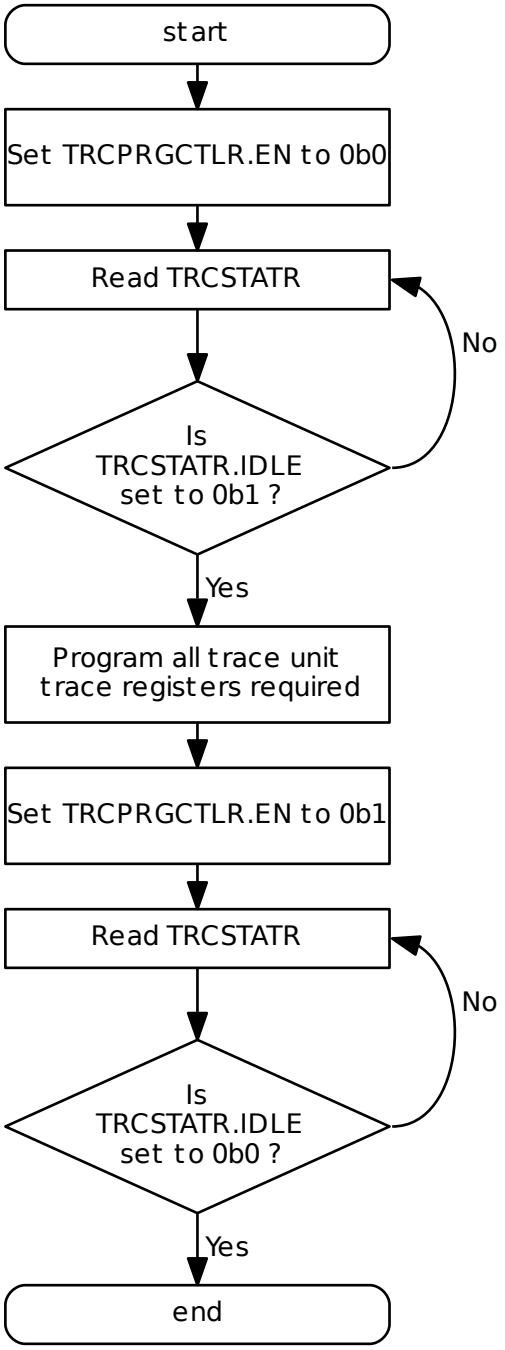

Figure D4-2 External debug interface programming procedure

IZGJRF Figure D4-3 shows the procedure that is used when programming the trace unit registers using the System instruction interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

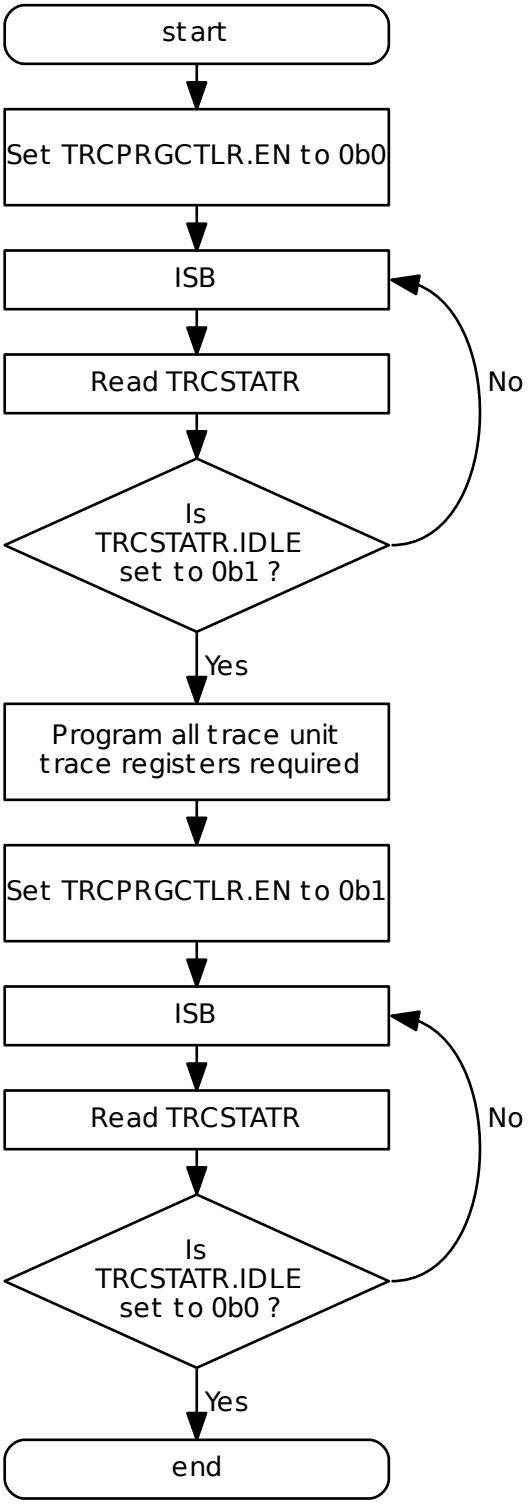

Figure D4-3 System instruction programming procedure

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.2 Programmers' model

D4.2.1 Accessing ETE registers

IVYNRB The ETE architecture provides registers for programming the ETE trace unit and reading back the programmed settings. These registers can be accessed by using one or more of the following access mechanisms:

- An external debug interface, for use by an external debugger.
- System instructions, for use by self-hosted software running on the PE being traced.

RNBPML When register accesses occur simultaneously from multiple access mechanisms, the trace unit behaves as if all accesses occur atomically in any order.

D4.2.1.1 External debug interface

IKPYGY The external debug interface defines an address-mapped peripheral that occupies 4KB of address space.

Note

The PE does not have to be in Debug state to program the ETE registers.

ITBMWG Unless otherwise stated in this section, information for supported access sizes for external debug interface is covered in Supported access sizes.

RVQWLY The trace unit supports the following access sizes:

- Word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations.
- Doubleword-aligned 64-bit accesses to access 64-bit registers mapped to a doubleword-aligned pair of adjacent 32-bit locations. The order in which the two halves are accessed is not specified.

Note

This means that a system implementing the Debug registers using a 32-bit bus, such as AMBA APB in CoreSight systems, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

RVNNPF All registers are only single-copy atomic at word granularity.

RKYDTQ The trace unit does not support the following accesses:

- Byte.
- Halfword.
- Unaligned word. These accesses are not single-copy atomic at word granularity.
- Unaligned doubleword. These accesses are not single-copy atomic at doubleword granularity.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair forming a 64-bit register.
- Quadword or higher.
- Exclusives.

RCFRFW For accesses from the external debug interface, the size of an access is determined by the interface. In an Arm Debug Interface compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register. The CoreSight APB-AP supports only word accesses.

RYFRMG The behavior of the accesses that are unsupported by the trace unit is CONSTRAINED UNPREDICTABLE and is one of the following:

- Accesses generate an External abort, and writes set the accessed register or registers to an UNKNOWN value or values.
- Reads return UNKNOWN data and writes are ignored.
- Reads return UNKNOWN data and writes set the accessed register or registers to an UNKNOWN value. This is the Arm preferred behavior.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{YSHRS} For accesses from the external debug interface which return an error response when `AllowExternalTraceAccess()` returns FALSE, EDPRSR.STAD is only set to 1 when this is the highest priority cause of the error. The following causes are higher priority than `AllowExternalTraceAccess()`:

- The trace unit Core power domain is powered down.
- The OS Lock is locked and the register is defined to return an error response due to the OS Lock being locked.

R_{KQMKX} Accesses from the external debug interface to unimplemented or Reserved trace unit registers behave as follows:

- When the trace unit core power domain is off, the access returns an error.
- Otherwise:
  - For accesses in the range of offsets `0xF00` to `0xFFC`, the access behaves as RES0H.
  - For accesses in the range of offsets `0x000` to `0xEFC`:
    - When the OS Lock is locked, the response is a CONSTRAINED UNPREDICTABLE choice of an error response or behaving as RES0H.
    - When the OS Lock is unlocked and AllowExternalTraceAccess() returns FALSE, the response is a CONSTRAINED UNPREDICTABLE choice of an error response or behaving as RES0H.
    - Otherwise, the access behaves as RES0H.

R_{WXKDP} Reads of write-only registers are Reserved.

R_{SVSNR} Writes to read-only registers are Reserved.

I_{WTJFD} For accesses that return an error, see Behavior of a not permitted memory-mapped access for more details on how this error is handled.

D4.2.1.2 System instructions

R_{VGVTS} Instructions with `CRn >= 0b1000` are not allocated for accessing trace unit registers.

R_{SGPQB} While the PE is in EL0, all accesses are UNDEFINED.

I_{WCXDT} For consistency with the Arm architecture, System instruction accesses to registers which are not implemented generate an Undefined Instruction exception. These accesses include:

- Writes to read-only registers.
- Reads from write-only registers.
- Accesses to registers which are not present due to the configuration of the trace unit.

D4.2.2 Synchronization of register updates

D4.2.2.1 System registers

I_{KWCGH} As defined in Synchronization requirements for AArch64 System registers, direct writes to trace unit registers are only guaranteed to be visible to execution after a subsequent Context synchronization event, which consists of one of the following:

- Taking an exception.
- Returning from an exception.
- Performing an Instruction Synchronization Barrier operation.
- Exit from Debug state.
- Executing a `DCPS` instruction in Debug state.
- Executing a `DRPS` instruction in Debug state.

I_{PNZZH} Direct reads of trace unit registers while the trace unit is not in the Stable or Idle states are not guaranteed to contain the results of the trace operation of execution previous to the direct read operation.

I_{QPVJQ} As defined in Synchronization requirements for AArch64 System registers, a direct write to a register using the same register number that was used by a previous System instruction to write it, the final result is the value of the second write, without requiring any context synchronization between the two write instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{WLGQK} As defined in Synchronization requirements for AArch64 System registers, a direct read of a register using the same register number that was used by an earlier direct write is guaranteed to observe the value that was written, without requiring any context synchronization between the write and read instructions.

S_{HSXGZ} Context synchronization events are important when changing the value of TRCPRGCTLR.EN or when changing the OS Lock. After writing to TRCPRGCTLR to change the value of TRCPRGCTLR.EN, one read of TRCSTATR is required before programming any other registers. A Context synchronization event is required between writing to TRCPRGCTLR and reading TRCSTATR. If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.

R_{WPWWS} When indirect writes or external writes to the registers in Table D4-2 occur, both of the following can observe the writes:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

Table D4-2 Registers with a guarantee of observability

Register	Notes
TRCCLAIMSET	Claim Tag Set Register
TRCCLAIMCLR	Claim Tag Clear Register
TRCCNTVR<n>	Counter Value Register <n>
TRCSEQSTR	Sequencer State Register
TRCSSCSR<n>	Single-shot Comparator Control Status Register

R_{JQTMC} When the trace unit becomes enabled or disabled as a result of a direct write, for any instruction in program order before the direct write, the new state of the trace unit does not affect trace operations.

R_{KNQWS} When the trace unit becomes enabled or disabled as a result of a direct write, for any instruction after a Context synchronization event in program order after the direct write, the new state of the trace unit takes effect for any trace operations.

Note

The registers which control whether the trace unit is enabled or disabled are:

- TRCPRGCTLR.
- OSLAR_EL1.

S_{YKGMP} Arm recommends that a Context synchronization event is executed after programming the Trace Unit registers, to ensure that all updates are synchronized to the trace unit before normal code execution resumes.

R_{WZQWC} When a Context synchronization event occurs while the trace unit is in the Idle or Stable states, and at no other time, indirect writes to the trace unit registers are guaranteed to be visible to direct reads.

R_{GQKGX} When either of the following events occurs, and at no other time, indirect writes to the trace unit registers are guaranteed to be visible to indirect reads or external reads:

- The trace unit transitions into the Stable state.
- The trace unit transitions into the Idle state.

R_{XLXQL} The trace unit functions perform indirect writes to the registers and indirect reads from the registers in architectural order.

I_{RCKQJ} See Trace unit programming states for more details on programming the trace unit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.2.2.2 External debugger registers

I_{KNWDX} As defined in the Synchronization of changes to the external debug registers, this section refers to accesses from the external debug interface as *external reads* and *external writes*.

I_{HMNWB} As defined in Synchronization of changes to the external debug registers, explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, so explicit synchronization is never required for registers that are accessible only in the external debug interface.

I_{YXWFD} As defined in Synchronization of changes to the external debug registers, when an external write to a register using the same register number that was used by a previous external write is performed, the final result is the value of the second write, without requiring any context synchronization between the two write accesses.

R_{PGTLX} The trace unit does not require explicit synchronization for an external write to be visible to indirect reads.

R_{DYRZC} The trace unit does not require explicit synchronization for an external write to be visible to subsequent external reads.

I_{RDFSX} As defined in Synchronization of changes to the external debug registers, explicit synchronization is required for an external write to be visible to direct reads.

R_{MMYRJ} While the trace unit is in the Stable or Idle states, the trace unit does not require explicit synchronization for indirect writes to be visible to external reads.

D4.2.2.3 Synchronization and the authentication interface

R_{WYWMJ} Changes to the authentication interface are indirect writes to TRCAUTHSTATUS by the controller of the authentication interface. It is IMPLEMENTATION DEFINED whether a change on the authentication interface is guaranteed to be observable to an external read of the register only after a Context synchronization event or in finite time.

I_{XVVKN} For more information, see Synchronization and the authentication interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3 Trace elements

IFNFWZ Trace elements form an AST which is used to describe the control flow of program execution. Different sequences of the trace elements can be used to imply the same operation. In this way FEAT_ETE can be used by different micro-architectures. This is similar to the approach used in previous trace protocols. For more information, see the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064)*.

IXXBMZ A trace unit compresses the information on the execution of the PE and outputs a trace byte stream that comprises of multiple packets of encoded data. The compression techniques that are used include:

Not having a trace element for every executed instruction in the instruction trace element stream

Instead, the trace unit generates *P0 elements* in the trace element stream when certain types of instruction are executed. These certain types of instructions are known as *P0 instructions*. A *P0 element* acts as a signpost in the program flow, indicating that execution is proceeding along a given path.

Consequently, the stream of *P0 elements* implies the execution of a greater number of instructions, and a trace analyzer can reconstruct the stream of instructions that are executed between *P0 elements* by using the P0 element stream and the program image.

Multiple trace elements in single packets

Common sequences of trace elements are encoded into single packets.

Removal of program addresses from the trace element stream

The trace analyzer can infer the addresses from the program image and previous history. This includes the targets of direct branch instructions, where the target address is encoded in the instruction itself.

Removal of predictable trace elements

Some trace elements can be removed from the AST representation if the contents of the trace element can be predicted by previous control flow choices in the software flow. For example the *Target Address element* for returning from a subroutine might not be required if the branch to the subroutine has been traced.

D4.3.1 Layer model

IXMFJT FEAT_ETE is based on a layer model. Each layer deals with a unique aspect of tracing the PE.

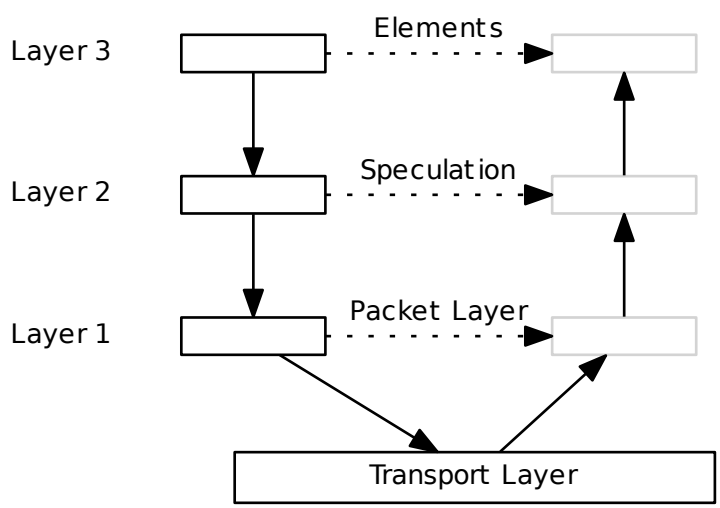

Figure D4-4 Layer model for compression and decompression

IGLPQZ Transport Layer

The transport layer either provides either:

- A path off chip.
- A path to a memory buffer for trace to be stored.

Layer 1

Layer 1 provides compression by:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Grouping trace elements together to form packets.
- Removing trace elements that can be implied.
- Compression against previous values.
- Leading zero compression.
- Reordering of trace elements.

Layer 2

Layer 2 provides:

- Speculation resolution.

Layer 3

At layer 3:

- PE behavior is converted into trace elements.
- Compression is achieved by removing the trace elements which can be predicted using the program image:
  - Direct branch target addresses.
  - Return stack optimization.
- Knowledge of the application is required in order to decompress. Processes that modify the instruction opcodes require additional information to allow debugging.

D4.3.2 Trace protocol synchronization

I_{CLTCM} The trace byte stream of a trace unit is typically stored in a circular buffer where, if the buffer is full, newer trace overwrites older trace. To ensure that a trace stream can be analyzed when the trace has been stored in a circular buffer, a trace unit must periodically generate trace protocol synchronization points in the trace byte stream.

I_{BPNSY} The following trace elements or packets are used to provide synchronization information in the different layers.

Table D4-3 Control of each layer

Layer	Control
Layer 3	*Context element* and *Target Address element*
Layer 2	*Trace Info element*
Layer 1	Trace Info packet
Transport Layer	Alignment Synchronization packet

I_{SFXXD} Whenever a trace analyzer receives a Trace Info packet, the trace analyzer receives information about the current state of the trace. However, the trace analyzer cannot begin analysis of program execution until it knows the context in which instructions are being executed and it has an instruction address to start analysis from.

R_{PGHPW} When a *Trace Info element* is generated, the trace unit generates a *Context element* and a *Target Address element* soon after the *Trace Info element*.

Note

There are common use cases where the ratio between the number of bytes associated with trace protocol synchronization and other trace bytes increases significantly, resulting in a degradation of the usability of the trace. Therefore Arm recommends that trace protocol synchronization only occurs when required.

D4.3.2.1 Non-periodic trace protocol synchronization

R_{QZRMQ} When the trace unit becomes operative, non-periodic trace protocol synchronization occurs.

R_{TTLJC} When non-periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet in the trace byte stream before any other trace packets are generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HMDGL} When non-periodic trace protocol synchronization occurs, the trace unit generates a *Trace Info element* in the trace element stream before any other trace elements are generated, except *Event elements* and *Instrumentation elements*.

I_{MQNBT} Arm recommends that if a trace protocol synchronization request occurs while ViewInst() is *inactive*, the Alignment Synchronization packet is not output in the trace byte stream until just before any of the following:

- ViewInst() becomes active.
- An Event packet is output.
- An Instrumentation packet is output.

I_{MHGVF} The generation of Instrumentation packets is independent of the ViewInst() function. Instrumentation packets can therefore be generated when ViewInst() indicates instruction tracing is inactive.

D4.3.2.2 Periodic trace protocol synchronization

I_{YPRYM} The trace unit can be programmed to generate trace protocol synchronization requests on a periodic basis, so that the trace element streams and the trace byte streams can be analyzed when stored in a circular trace buffer. TRCSYNCPR.PERIOD controls periodic trace protocol synchronization requests.

I_{NTFYC} Periodic trace protocol synchronization can also be requested by the trace capture infrastructure, for example if a trace protocol synchronization request is received on an Arm AMBA ATB interface. For more information on the Arm AMBA ATB interface, see the *AMBA® ATB Protocol Specification (ARM IHI 0032)*.

R_{QHHSY} When periodic trace protocol synchronization is requested, either by TRCSYNCPR.PERIOD or by other sources, the trace unit performs periodic trace protocol synchronization.

R_{VMPYW} When periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet. The trace unit then generates a *Trace Info element* to indicate a point where program execution trace analysis can start. A *Trace Info element* is not required if only *Event elements* and *Instrumentation elements* are output.

I_{QYQRY} Arm recommends that an Alignment Synchronization packet is only output in the trace byte stream if other trace packets have been output since the previous Alignment Synchronization packet. This reduces the risk of a circular buffer filling and overwriting trace.

I_{NQYXW} If two or more periodic trace protocol synchronization requests occur, and no trace is generated between these two requests, then Arm recommends that a non-periodic trace protocol synchronization occurs before any further trace is generated. This ensures that when tracing has been inactive for a long period of time, the trace stream is fully synchronized when tracing is re-activated.

D4.3.2.3 Synchronization of instruction trace

R_{KKQGK} When non-periodic trace protocol synchronization occurs, the trace unit generates a *Context element* and a *Target Address element* before any *P0 elements* are generated, to provide the trace analyzer with *Context information* and *Address information*.

R_{SVGNN} When periodic trace protocol synchronization occurs, and ViewInst() is active when the corresponding *Trace Info element* is generated, the trace unit generates a *Context element* and a *Target Address element* which provide the *Context information* and *Address information* for the target of the most recent non-canceled *P0 elements*.

Note

If the trace unit generates the *Context element* and *Target Address element* immediately after the *Trace Info element*, then the most recent non-canceled *P0 elements* might have occurred before the *Trace Info element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

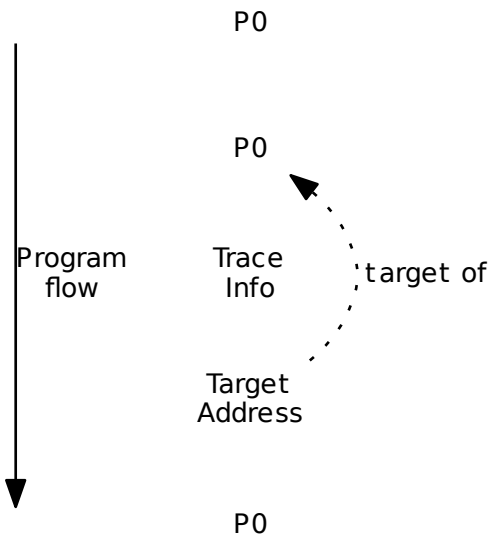

Figure D4-5 Example of Target Address element after Trace Info element.

R_{DLPYX} When periodic trace protocol synchronization occurs, and ViewInst() is inactive when the corresponding *Trace Info element* is generated, when ViewInst() becomes active and a *Trace On element* is generated, the trace unit generates a *Context element* and a *Target Address element* before any *Atom elements*, *Q elements*, or *Exception elements* are generated. This provides the trace analyzer with *Context information* and *Address information*.

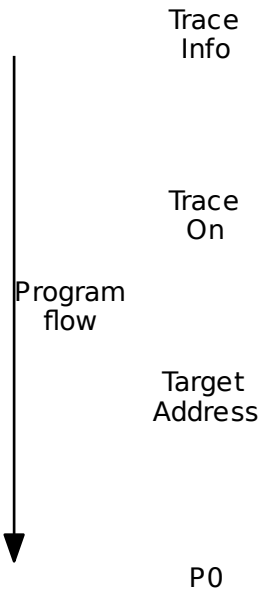

Figure D4-6 Example of Target Address element after Trace Info element in a filtered region.

I_{YZPCB} If a *Cancel element* cancels any *P0 elements* before a *Trace Info element*, then the trace analyzer discards all of the following:

- The canceled *P0 elements*.
- The *Trace Info element*.
- All elements after the *Trace Info element*, up to and including the *Cancel element*. This includes any *Context elements* or *Target Address elements*.

Note

In this scenario, information from the canceled *Trace Info element* can still be used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

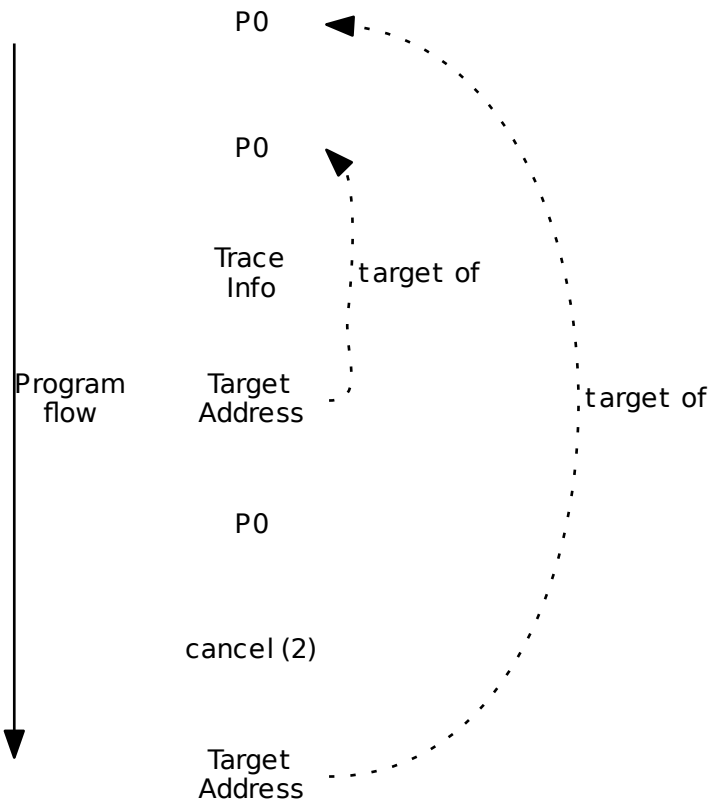

Figure D4-7 Example of Target Address element after Trace Info element in a mispredicted region.

R_{KGPTB} When a *Cancel element* is generated which cancels any *P0 elements* before a *Trace Info element*, the trace unit generates a new *Context element* and a new *Target Address element*, which indicate the target of the most recent *P0 element* that has not been canceled.

I_{CHTFM} The *Target Address element* and *Context element* might indicate the target of a *P0 element* from before the *Trace Info element*, or might be delayed until after the next *P0 element*, and therefore indicate the target of that *P0 element*.

Note

If the trace unit generates the new *Context element* and *Target Address element* prior to the next new *P0 element*, then this might prevent the indication of execution of some instructions before the *Trace Info element*.

I_{NSWTK} If the *Cancel element* cancels all *P0 elements* after a *Trace Info element*, but no *P0 elements* prior to the *Trace Info element*, then it might be necessary for the trace unit to immediately generate a *Context element* and *Target Address element*. This is because a *Context element* and *Target Address element* might have been present in the trace element stream after the *Trace Info element*, and those *Context element* and *Target Address element* are now discarded.

D4.3.3 Speculation in the trace element stream

I_{NVBWS} FEAT_ETE supports the correction of trace. This might be because of the tracing of speculative execution of instructions by a PE.

I_{RTJNK} A FEAT_ETE trace unit traces speculatively-executed instructions in the same way as all other instructions, so that both speculatively-executed instructions and architecturally-executed instructions appear in the instruction trace element stream. This means that some of the program execution information that is shown in the trace element stream might be incorrect, because some of the speculatively executed instructions might be mis-speculated.

Note

The level of speculation that is revealed in the trace is IMPLEMENTATION SPECIFIC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{XLLKT} The trace unit resolves this speculation by generating trace elements to confirm the status of each instruction in the instruction trace element stream. That is, the trace unit generates trace elements to show whether each instruction has been committed for execution, or canceled because of mis-speculation. This means that a trace analyzer does not know the status of a traced instruction until the trace analyzer receives a trace element that indicates whether the instruction has been committed for execution, or canceled because the instruction was mis-speculated.

R_{ZJJKY} When speculatively-executed instructions are traced, the trace unit subsequently generates trace elements that indicate whether the instructions have been committed for execution, or have been canceled.

I_{KYXKZ} A trace analyzer takes the appropriate action, which might involve canceling some trace elements, to determine the actual program execution.

I_{GGFML} Trace elements that resolve the status of a traced instruction are called *speculation resolution elements*. For more information on speculation elements, see Speculation Resolution Elements.

R_{KYGRF} When trace is generated for speculative execution, for mis-speculated execution, the trace unit does not trace any information that cannot be accessed by software executing at the same or at a lower level of privilege than the mis-speculated execution.

R_{QHQLY} When a Context synchronization event is speculated as being taken or executed, the trace unit does not generate trace for any speculative execution after the Context synchronization event until the Context synchronization event is resolved.

R_{LWJCQ} When a speculated Context synchronization event is resolved as being not taken or not executed, the trace unit does not generate trace for mis-speculated execution that occurred after the Context synchronization event.

R_{YGSGJ} When an exit from a Trace Prohibited region is speculated as being taken, the trace unit does not generate trace for any speculative execution after the exit from the Trace Prohibited region, until the exit from the Trace Prohibited region is resolved.

R_{SRLCG} When a speculated exit from a Trace Prohibited region is resolved as being not taken, the trace unit does not generate trace for mis-speculated execution that occurred after the exit from a Trace Prohibited region.

D4.3.4 Trace element stream

I_{LTBWR} A *Trace element stream* is a sequence of elements which describe:

- The software control path of PE execution traced by the trace unit.
- Event-based trace.
- Temporal information.

I_{LBSZF} A Trace Info element provides a point in the trace element stream where analysis of the trace element stream can begin. Trace Info elements include setup information about:

- The static trace programming that does not change during a trace session, including whether cycle counting is enabled, and if enabled, the cycle count threshold.
- Dynamic information that might change during a trace session, such as the speculation depth, which indicates how many unresolved P0 elements were traced before the Trace Info element.

D4.3.5 P0 element

I_{LLDBJ} P0 elements imply the execution of instructions.

I_{XPZXL} P0 elements are generated speculatively and must be either committed or canceled. For more information on speculation elements, see Speculation Resolution Elements.

R_{XVHWG} P0 elements must be generated in sequential execution order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3.5.1 Atom element

IXPFJG An *Atom element* implies that one or more instructions have been executed, up to and including the next *P0 instruction*. Only certain instructions generate an *Atom element*. See Instruction and exception classification for more information on these instructions.

RPRNZH The *Atom element* is one of the following types:

- E Atom.
- N Atom.

ICYMYM The meaning of the type of an *Atom element* depends on the instruction it is encoding. For example, branch instructions are represented as an E *Atom element* if the branch was taken and an N *Atom element* if not taken.

D4.3.5.2 Exception element

IYVMSC An *Exception element* indicates a change in program flow which cannot be calculated by the analysis of the program image, or which is caused by an instruction which is not a *P0 instruction*. Such a change in program flow is described as an Exceptional occurrence.

RMKPFJ An Exceptional occurrence consists of the following:

- PE Architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

IJLZPY An *Exception element* indicates:

- That an Exceptional occurrence has occurred.
- The type of Exceptional occurrence.
- The virtual address where the Exceptional occurrence was taken from, also known as the preferred exception return address.

RDXJBQ The instruction set for the preferred exception return address for an *Exception element* is one of the following:

- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

RYPPRH An *Exception element* is a *P0 element*.

D4.3.5.2.1 PE Architectural exceptions

RPZRFL The following exception types are used to indicate PE Architectural exceptions:

- IRQ.
- FIQ.
- Trap.
- Call.
- Inst fault.
- Data fault.
- Inst debug.
- Data debug.
- Alignment.
- System Error.
- Debug halt.

See Instruction and exception classification for information on the mapping between the PE Architectural exceptions and these exception types.

RSFYMW Table D4-4 defines the preferred exception return address for each exception type for PE Architectural exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-4 Preferred exception return address for PE Architectural exceptions

Exception type	Preferred exception return address
IRQ	Instruction after the last executed instruction
FIQ	Instruction after the last executed instruction
Trap	For a trapped instruction or UNDEFINED instruction, the preferred exception return address is the address of the instruction. For a trapped exception, the preferred exception return address is the address of the instruction that caused the exception.
Call	Instruction after the call instruction
Inst fault	Instruction that caused the exception
Data fault	Instruction that caused the exception
Inst debug	Instruction that caused the exception
Data debug	Instruction that caused the exception
Alignment	Instruction that caused the alignment exception
System Error	Instruction after the last executed instruction
Debug halt	The instruction after the last executed instruction, that is, the value loaded into the DLR register.

I_{GZKGC} The nature of System Error means that execution might not complete up to the preferred exception return address, or it might perform some operations after the preferred exception return address. This behavior is IMPLEMENTATION DEFINED and might vary depending on the cause of the exception.

R_{GFJZF} When an imprecise System Error exception occurs, the preferred exception return address is the address stored in the relevant ELR when the exception is taken.

S_{GKMTH} When a System Error exception occurs, the trace analyzer must be aware that the preferred exception return address might not indicate the exact point at which program execution was interrupted. The trace analyzer should not rely on the preferred exception return address for inferring exactly which instructions were executed. This behavior only occurs for imprecise System Error exceptions.

R_{LBLWT} When an imprecise Debug halt exception occurs, the preferred exception return address is the address stored in DLR or DLR_EL0 when the exception is taken.

S_{RDJXM} When an imprecise Debug halt exception occurs, the trace analyzer must be aware that the preferred exception return address might not indicate the exact point at which program execution was interrupted. The trace analyzer should not rely on the preferred exception return address for inferring exactly which instructions were executed. An imprecise Debug halt exception can only occur under direct control of a debugger, usually by controlling EDRCR.CBRRQ.

D4.3.5.2.2 ETE defined exceptions

R_{MZJTJ} In addition to the Arm Architectural exceptions, the ETE specifies the following Exceptional occurrences that are traced using *Exception elements*:

- PE Reset, which indicates that a PE Warm reset has occurred.

R_{NRJGC} Table D4-5 defines the preferred exception return address for each exception type for ETE defined exceptions.

Table D4-5 Preferred exception return address for ETE defined exceptions

Exception type	Preferred exception return address
PE Reset	UNKNOWN

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{JRNYF} When a PE Reset occurs, the preferred exception return address and context are UNKNOWN. Therefore for an *Exception element* indicating a PE Reset the preferred exception return address and context are UNKNOWN. No instruction execution is indicated between the previous *P0 element* and the *Exception element*.

I_{QJYYZ} When an *Exception element* indicating a PE Reset occurs:

- The target address and target context of the previous *P0 element* might be UNKNOWN.
- If there are no *P0 elements* between a *Trace On element* and the *Exception element*, then the initial address and context after the previous *Trace On element* might be UNKNOWN.

D4.3.5.2.3 IMPLEMENTATION DEFINED exceptions

R_{ZVYQW} ETE defines some exception types which are IMPLEMENTATION DEFINED, including but not limited to:

- ECC error correction.
- Generic replay of program execution.

I_{XHFLL} The use of the IMPLEMENTATION DEFINED exceptions is optional and IMPLEMENTATION DEFINED. IMPLEMENTATION DEFINED exceptions are not required to be traced but are intended to be used to simplify tracing of certain micro-architectural situations.

I_{DFLDJ} In general, the preferred exception return address is the address of the instruction after the last executed instruction, before the exception occurs.

D4.3.5.3 Source Address element

I_{DJTGL} The *Source Address element* indicates execution up to and including a provided *P0 instruction* address, and indicates the *P0 instruction* is taken. All *P0 instructions* except the final *P0 instruction* are not taken, which means that explicit N *Atom elements* are not required to be traced for those *P0 instructions*. A *Source Address element* indicates both of the following for the final *P0 instruction*:

- The instruction set.
- The virtual address of the instruction.

R_{HVVRK} The instruction set for a *Source Address element* is one of the following:

- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

R_{WTRBB} A *Source Address element* is a *P0 element*.

D4.3.5.4 Q element

R_{JRFYT} A *Q element* belongs to the *P0 element* group in the instruction trace element stream, and must be explicitly resolved or canceled.

I_{XPNWS} A *Q element* can optionally include a number, M. The number is a count of the instructions that are executed since the most recent *P0 element*, which might be a *Q element*. If it does not include a count of instructions, then the number of instructions that are executed since the most recent *P0 element* is UNKNOWN.

R_{XWBMW} The trace unit generates *Q elements* in the program order in which they occur, and the trace protocol encode and decode process maintains this order.

R_{JBYXC} A *Q element* does not imply Exceptional occurrences.

R_{KPNGG} When a *Q element* implies an Exception return instruction which is taken, that instruction is the last instruction that is implied by the *Q element*.

R_{YRLJR} When a *Q element* implies an executed `ISB` instruction, this is the last instruction implied by the *Q element* if execution continues from a new context after the `ISB`.

R_{LZLDH} When execution continues from a new context after a *Q element* is generated, the trace unit generates a *Context element* after the *Q element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{BTNZC} The *Context element* might be generated before or after the *Target Address element* that is also required after the *Q element*.

If a context change occurs at a point that is not a Context synchronization event, then the last instruction that is implied by a *Q element* must be the last instruction that is executed with the old context. The trace unit can then generate a *Context element* after the *Q element* to indicate the new context.

D4.3.6 Virtual Address Space Element

D4.3.6.1 Trace On element

R_{NHDCF} A *Trace On element* indicates a discontinuity in the trace element stream. The trace unit inserts a *Trace On element* after a gap in the generation of the trace element stream:

- When the trace generation becomes operative and before any *P0 elements*.
- If some instructions are filtered out of the trace.
- The first traced instructions after:
  - A Trace Prohibited region.
  - The PE leaves Debug state.
- When instruction trace is lost because a trace unit buffer overflow occurs.

R_{KMFKP} When a *Trace On element* is generated, the trace unit generates a *Target Address element* before the next *P0 element*.

R_{TJLYH} When a *Trace On element* is generated, the trace unit generates a *Context element* before the next *Atom element*, *Exception element* or *Q element*, to indicate where tracing starts, unless the context has not changed since the previous *Context element* was output.

R_{JKFBS} When the first *Trace On element* is generated, the trace unit outputs the corresponding *Context element* before the first *P0 element*.

D4.3.6.2 Target Address element

R_{QWBLT} A *Target Address element* indicates both of the following for the next instruction to be executed:

- The instruction set.
- The virtual address of the instruction.

R_{JYKHH} The instruction set for a *Target Address element* is one of the following:

- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

R_{HMWHY} The trace unit generates *Target Address elements* in program order relative to other *P0 elements*.

I_{XCKNM} *Target Address element* values can be corrected by another *Target Address element* if both *Target Address elements* are generated before the next *P0 element* or *Trace On element*.

D4.3.6.3 Context element

I_{KQKFF} The *Context element* indicates the execution context for the next instruction to be executed.

R_{VHQYV} The *Context element* provides the following *Context information*:

- The Security state.
- The Exception level.
- The Execution state of the PE.

R_{WSVRL} The *Context element* can optionally provide the following *Context information*:

- The Context identifier.
- The Virtual context identifier.

R_{WJDWF} The trace unit generates *Context elements* in program order relative to *P0 elements*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3.7 Temporal elements

I_{HHXND} Temporal elements provide information about the passage of time within the trace element stream. The following temporal elements are supported by ETE:

The Cycle Count element
Indicates the passage of PE clock cycles within the trace element stream.

The Timestamp element
Indicates the passage of time within the trace element stream.

The Timestamp Marker element
Indicates that the most recent *P0 element*, *Event element*, *Instrumentation element* has been timestamped and that a subsequent *Timestamp element* contains the timestamp value.

D4.3.7.1 Cycle Count element

I_{NVGJP} Each *Cycle Count element* is associated with a *Commit element*, and when a *Commit element* is generated, a *Cycle Count element* might also be generated.

R_{BZQWX} Each *Cycle Count element* is associated with the most recent *Commit element*.

R_{VZXNN} A *Cycle Count element* indicates the number of PE clock cycles between the two most recent *Commit elements* that both have an associated *Cycle Count element*.

I_{FHGKM} Not every *Commit element* is required to have an associated *Cycle Count element*.

R_{VNYMN} *Cycle Count elements* are generated in order relative to *Commit elements*.

D4.3.7.2 Timestamp element

I_{LKDJM} The *Timestamp element* inserts a global timestamp value into the trace element stream.

I_{BLBJX} The source for timestamp reported in the timestamp element is controlled by:

- TRFCR_EL1.TS.
- TRFCR_EL2.TS.

R_{BRJJF} A timestamp value of zero indicates that the timestamp value is UNKNOWN.

I_{VTLTF} An UNKNOWN timestamp value might occur if the system does not support timestamping or if the timestamp is temporarily unavailable.

I_{YQJDR} The source for the payload of *Timestamp elements* is controlled by the TRFCR registers and the virtual timers. It is expected that these registers will be changed by context switch software. As a result it is possible that payloads of *Timestamp elements* might appear to have discontinuities, and even go backwards, if the source of the timestamp changes, or any context switching changes the System registers which control the timestamp value.

R_{MCSGX} If FEAT_ETEv1p1 is implemented, then when there has been a *Timestamp Marker element* before the *Timestamp element*, the *Timestamp element* contains the timestamp value of the most recent of the following before the *Timestamp Marker element*:

- P0 element.
- Event element.
- Instrumentation element.

R_{DGTJZ} If FEAT_ETEv1p1 is not implemented or if there has not been a *Timestamp Marker element* before the *Timestamp element*, the *Timestamp element* contains the timestamp value of the most recent of the following before the *Timestamp element*:

- P0 element.
- Event element.
- Instrumentation element.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{PXZVX} If TRCIDR0.TSMARK is 1 and there is no previous *Timestamp Marker element*, the *Timestamp element* is for a *P0 element*, *Event element*, or *Instrumentation element* which is before the start of the trace. This scenario might occur when trace analysis starts at a *Trace Info element* which is not the first *Trace Info element*, and the *Timestamp Marker element* was generated before the *Trace Info element*.

I_{CSZYW} The requirement for a *Timestamp Marker element* for every *Timestamp element* is to avoid needing to indicate if there has been a *Timestamp Marker element* at a Trace Info point. This allows a trace analyzer to assume there is a *Timestamp Marker element* (or not) before the Trace Info, based on a static piece of information.

D4.3.7.3 Timestamp Marker element

R_{RFYPT} The *Timestamp Marker element* indicates the most recent *P0 element*, *Event element*, or *Instrumentation element* has been timestamped, and that a *Timestamp element* will follow containing the timestamp value.

R_{SZRHP} *Timestamp Marker element*s are generated in order with respect to *P0 element*s, *Event element*s, and *Instrumentation element*.

R_{DCRVK} *Timestamp Marker element*s are not canceled by *Cancel elements*.

I_{DLCLX} A *Cancel element* might cause a *P0 element* to be canceled and if there is a *Timestamp Marker element* that is associated with that *P0 element* then the *Timestamp Marker element* is not associated with any *P0 element*. The *Timestamp element* which is associated with the *Timestamp Marker element* is unaffected, and is still usable for timestamping the approximate position in the trace stream.

R_{VWJVC} If 2 *Timestamp Marker element*s occur without a *Timestamp element* between them, the oldest *Timestamp Marker element* is ignored.

R_{JNWJY} If an *Overflow element* or *Discard element* occurs after a *Timestamp Marker element* and before a *Timestamp element*, the *Timestamp Marker element* is ignored.

R_{LWZXK} If *Timestamp Marker element*s are generated by the trace unit, every *Timestamp element* must have a corresponding *Timestamp Marker element* generated before the *Timestamp element*.

I_{JGKZJ} The generation of *Timestamp Marker element*s is indicated in TRCIDR0.TSMARK.

D4.3.8 Speculation Resolution Elements

I_{YYMXT} The ETE architecture allows trace to be generated speculatively and then later committed or removed by the decompression process. Each *P0 element* is traced and is considered speculative until either committed by a *Commit element* or canceled by a *Cancel element*. This method of generating speculative trace allows for the tracing of speculative execution.

I_{SRRZZ} Speculation resolution elements provide a trace analyzer with information about which trace elements were correctly or incorrectly generated, and ensure the trace analyzer can reconstruct the program execution. The following speculation resolution elements are supported by ETE:

The Mispredict element
Corrects the most recent *Atom element*.

The Cancel element
Indicates that one or more *P0 elements* are canceled.

The Commit element
Indicates that one or more *P0 elements* are resolved for execution.

The Discard element
Removes all speculative *P0 elements*.

I_{XLHWT} TRCIDR8.MAXSPEC specifies the maximum number of uncommitted *P0 elements* which can be discarded at a later stage using *Cancel elements*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3.8.1 Commit element

I_{KQQML} A *Commit element* indicates that a number of unresolved *P0 elements* have been resolved for execution. The resolved *P0 elements* are the oldest *P0 elements*.

R_{PNBGQ} The *Commit element* resolves all types of *P0 element*.

I_{KHYLN} *Commit elements* might be merged if the total number of *P0 elements* resolved is less than TRCIDR8.MAXSPEC. *Commit elements* are merged by adding their respective commit count values together.

Figure D4-8 shows an example operation for a *Commit element*:

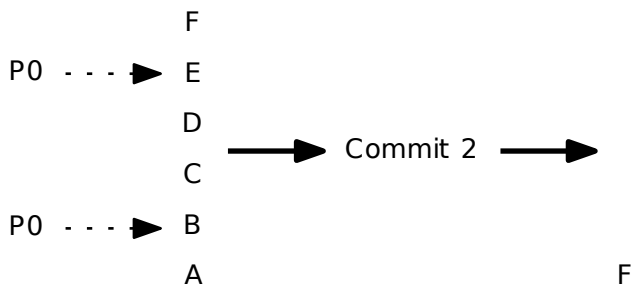

Figure D4-8 Commit Operation Example

D4.3.8.2 Cancel element

I_{MRGLC} The *Cancel element* indicates the number of youngest unresolved and uncanceled *P0 elements* that are canceled from execution. A trace unit might cancel elements because of many reasons, including but not limited to:

- A *P0 instruction* is mis-speculated.
- An exception occurs.

R_{WLTNX} The *Cancel element* cancels all types of *P0 element*.

I_{NDQKN} *Cancel elements* might be merged if no *P0 elements* have been generated in between. *Cancel elements* are merged by adding their respective cancel numbers together.

Figure D4-9 shows an example operation for a *Cancel element*:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

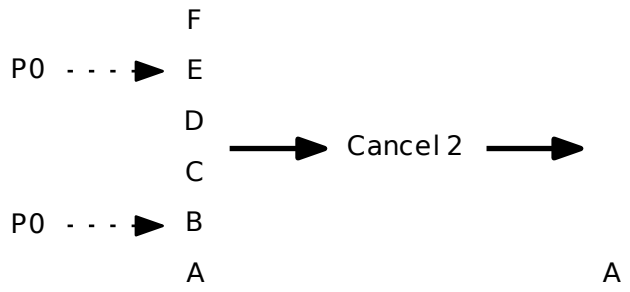

Figure D4-9 Cancel Operation Example

D4.3.8.3 Discard element

I_{TCWCN} A *Discard element* is generated if uncommitted *P0 elements* remain when trace generation becomes inoperative or if the resolution of uncommitted *P0 elements* cannot be output by the trace unit.

I_{STXQZ} If trace generation remains inoperative, the outcomes of instructions that are traced by *P0 elements*, such as conditional *P0 instructions*, cannot be resolved, and therefore a *Discard element* indicates that all uncommitted *P0 elements* must be discarded.

D4.3.8.4 Mispredict element

I_{GBKKQ} The *Mispredict element* indicates that the most recent non-canceled *Atom element* has the incorrect E or N status.

I_{RGVGL} For example, if a branch instruction is predicted as taken, it is traced with an E *Atom element*. If the prediction becomes incorrect, then a *Mispredict element* is traced to indicate to a trace analyzer that the E *Atom element* changes to an N *Atom element*.

D4.3.9 Instrumentation element

I_{ZLFFN} The *Instrumentation element* indicates execution of a `TRCIT` instruction.

R_{BXTXK} If Instrumentation tracing is prohibited, then no trace is generated by the `TRCIT` instruction, and `TRCIT` behaves as a `NOP`.

R_{JJZXW} If tracing is prohibited, then no ETE trace is generated by the `TRCIT` instruction, and `TRCIT` behaves as a `NOP`. `TraceAllowed()` defines when tracing is prohibited.

R_{BWHCS} If Instrumentation tracing is not prohibited and tracing is enabled and not prohibited, then execution of the `TRCIT` instruction requests an Instrumentation element to be generated in the ETE trace stream. `TraceInstrumentationAllowed()` defines when Instrumentation tracing is not prohibited.

R_{NHJLQ} When an Instrumentation element is requested, the Instrumentation element is generated if and only if the `TRCIT` instruction is architecturally executed.

R_{VPFGH} If the Instrumentation element cannot be generated when required, then a trace unit buffer overflow occurs.

R_{CJRNL} Instrumentation elements are not discarded by Cancel elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3.10 Other elements

D4.3.10.1 Event element

I_{RBKYZ} The *Event element* indicates when a programmed *ETEEvent* occurs and its payload contains a number to identify the *ETEEvent* number. See TRCEVENTCTL0R, and TRCEVENTCTL1R, for information about the programming of arbitrary *ETEEvents*.

R_{SMLVB} *Event elements* maintain order relative to other *Event elements*.

D4.3.10.2 Overflow element

I_{RFQKZ} The *Overflow element* indicates that the trace unit buffer has overflowed, and at least one trace element might have been lost.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.4 Instruction and exception classification

INMJBZ This section defines all of the *P0 instructions*.

RPBVZM *P0 instructions* comprise all of the following:

- All direct *P0 instructions*.
- All indirect *P0 instructions*.

RGFNRJ Direct *P0 instructions* comprise all of the following:

- All direct branch instructions.
- `ISB` instructions.
- `TSTART` instructions.
- `WFE`, `WFET`, `WFI`, and `WFIT` instructions, when indicated by TRCIDR2.WFXMODE.

RDJMQM Indirect *P0 instructions* comprise all of the following:

- All indirect branch instructions.

RKJTCL All uses of `ISB` in this section apply to all variants of the `ISB` instruction, including the `CP15ISB` instruction.

D4.4.1 AArch64 instructions

D4.4.1.1 Direct P0 instructions

RTWTMK The following table describes the A64 direct *P0 instructions*.

Table D4-6 A64 direct P0 instructions

Instruction	Description
B	Unconditional Branch
B.cond	Conditional Branch
BC.cond	Branch Consistent conditionally
BL	Branch with link
CB<cc> (immediate)	Compare register with immediate and branch
CB<cc> (register)	Compare registers and branch
CBB<cc>	Compare bytes and branch
CBH<cc>	Compare halfwords and branch
CBZ or CBNZ	Compare with zero and branch
ISB	Instruction Synchronization Barrier
TBZ or TBNZ	Test and branch
WFE, WFET	Wait For Event
WFI, WFIT	Wait For Interrupt

D4.4.1.2 Indirect P0 instructions

RLTZGC The following table describes the A64 indirect *P0 instructions*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-7 A64 indirect P0 instructions

Instruction	Description
BLR	Branch with link to register
BLRAA	Authenticate and branch with link
BLRAAZ	Authenticate and branch with link
BLRAB	Authenticate and branch with link
BLRABZ	Authenticate and branch with link
BR	Branch to register
BRAA	Authenticate and branch
BRAAZ	Authenticate and branch
BRAB	Authenticate and branch
BRABZ	Authenticate and branch
ERET	Return From Exception
ERETAA	Authenticate and Exception return
ERETAB	Authenticate and Exception return
RET	Return From subroutine
RETAA	Authenticate and function return
RETAASPPC	Authenticate and function return
RETAASPPCR	Authenticate and function return
RETAB	Authenticate and function return
RETABSPPC	Authenticate and function return
RETABSPPCR	Authenticate and function return

D4.4.1.3 Return from exception instructions

R_{BVKWS} The following table describes the A64 return from exception instructions.

Table D4-8 A64 return from exception instructions

Instruction	Description
ERET	Return From Exception
ERETAA	Authenticate and Exception return
ERETAB	Authenticate and Exception return

D4.4.1.4 Branch with link instructions

R_{DVKBK} The following table describes the A64 branch with link instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-9 A64 branch with link instructions

Instruction	Description
BL	Branch with link
BLR	Branch with link to register
BLRAA	Authenticate and branch with link
BLRAAZ	Authenticate and branch with link
BLRAB	Authenticate and branch with link
BLRABZ	Authenticate and branch with link

D4.4.1.5 Meaning of Atom elements

R_{HGDNB} The following table describes the meaning of Atom elements in AArch64 A64.

Table D4-10 Meaning of Atom elements in AArch64 A64

Instruction	E	N
B	The branch was taken	RESERVED
B.cond	The branch was taken	The branch was not taken
BC.cond	The branch was taken	The branch was not taken
BL	The branch was taken	RESERVED
BLR	The branch was taken	RESERVED
BLRAA	The branch was taken	RESERVED
BLRAAZ	The branch was taken	RESERVED
BLRAB	The branch was taken	RESERVED
BLRABZ	The branch was taken	RESERVED
BR	The branch was taken	RESERVED
BRAA	The branch was taken	RESERVED
BRAAZ	The branch was taken	RESERVED
BRAB	The branch was taken	RESERVED
BRABZ	The branch was taken	RESERVED
CB<cc> (immediate)	The branch was taken	The branch was not taken
CB<cc> (register)	The branch was taken	The branch was not taken
CBB<cc>	The branch was taken	The branch was not taken
CBH<cc>	The branch was taken	The branch was not taken
CBZ or CBNZ	The branch was taken	The branch was not taken
ERET	The branch was taken and the PE returned from the Exception	RESERVED
ERETAA	The branch was taken and the PE returned from Exception	RESERVED
ERETAB	The branch was taken and the PE returned from Exception	RESERVED
ISB	The ISB performed a Context synchronization event and is considered as taken	RESERVED
RET	The branch was taken and the PE returned from the subroutine	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	E	N
RETAA	The branch was taken and the PE returned from the subroutine	RESERVED
RETAASPPC	The branch was taken and the PE returned from the subroutine	RESERVED
RETAASPPCR	The branch was taken and the PE returned from the subroutine	RESERVED
RETAB	The branch was taken and the PE returned from the subroutine	RESERVED
RETABSPPC	The branch was taken and the PE returned from the subroutine	RESERVED
RETABSPPCR	The branch was taken and the PE returned from the subroutine	RESERVED
TBZ or TBNZ	The branch was taken	The branch was not taken
WFE, WFET	The instruction was executed and is considered as taken	RESERVED
WFI, WFIT	The instruction was executed and is considered as taken	RESERVED

D4.4.2 AArch32 A32 instructions

D4.4.2.1 Direct P0 instructions

R_{GWKSD} The following table describes the A32 direct *P0 instructions*.

Table D4-11 A32 direct P0 instructions

Instruction	Description
B{<c>}	Branch
BL	Branch with link
BLX <immed>	Branch with link and exchange
ISB	Instruction Synchronization Barrier
WFE	Wait For Event
WFI	Wait For Interrupt

D4.4.2.2 Indirect P0 instructions

R_{DTZVJ} The following table describes the A32 indirect *P0 instructions*.

Table D4-12 A32 indirect P0 instructions

Instruction	Description
BLX <reg>	Branch with Link and Exchange
BX	Branch and Exchange
BXJ	Branch and Exchange
Data processing instructions that modify the PC	-
ERET	Exception Return
LDM including the PC	Load Multiple to the PC
LDR PC	Load a word to the PC
RFE	Return From Exception

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.4.2.3 Branch with link instructions

R_{PLXGS} The following table describes the A32 branch with link instructions.

Table D4-13 A32 branch with link instructions

Instruction	Description
BL	Branch with link
BLX <immed>	Branch with link and exchange
BLX <reg>	Branch with Link and Exchange

D4.4.2.4 Meaning of Atom elements

R_{YVVSN} The following table describes the meaning of Atom elements in AArch32 A32.

Table D4-14 Meaning of Atom elements in AArch32 A32

Instruction	E	N
B{<c>}	The branch was taken	The branch was not taken
BL	The branch was taken	The branch was not taken
BLX <immed>	The branch was taken	The branch was not taken
BLX <reg>	The branch was taken	The branch was not taken
BX	The branch was taken	The branch was not taken
BXJ	The branch was taken	The branch was not taken
Data processing instructions that modify the PC	The branch was taken	The branch was not taken
ERET	The branch was taken and the PE returned from an Exception	The branch was not taken and the PE did not return from an Exception
ISB	The ISB performed a Context synchronization event and is considered as taken	The ISB did not perform a Context synchronization event and is considered as not taken
LDM including the PC	The branch was taken	The branch was not taken
LDR PC	The branch was taken	The branch was not taken
RFE	The branch was taken and the PE returned from the Exception	RESERVED.
WFE	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken	The instruction failed its condition code check and is considered as not taken
WFI	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken	The instruction failed its condition code check and is considered as not taken

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.4.3 AArch32 T32 instructions

D4.4.3.1 Direct P0 instructions

R_{NRJKR} The following table describes the T32 direct P0 instructions.

Table D4-15 T32 direct P0 instructions

Instruction	Description
B{<c>}	Branch
BL	Branch with Link
BLX <immed>	Branch with Link and Exchange
CBNZ	Compare and Branch on Nonzero
CBZ	Compare and Branch on Zero
ISB	Instruction Synchronization Barrier, including CP15 encodings
WFE	Wait For Event
WFI	Wait For Interrupt

D4.4.3.2 Indirect P0 instructions

R_{WXDRS} The following table describes the T32 indirect P0 instructions.

Table D4-16 T32 indirect P0 instructions

Instruction	Description
BLX <reg>	Branch with Link and Exchange
BX	Branch and Exchange
BXJ	Branch and Exchange
Data processing instructions that modify the PC	-
LDM including the PC	Load Multiple including to the PC
LDR to the PC	Load to the PC
POP {..,PC}	Load the PC from the stack
RFE	Return From Exception
TBB	Table Branch
TBH	Table Branch

D4.4.3.3 Branch with link instructions

R_{BHLTJ} The following table describes the T32 branch with link instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-17 T32 branch with link instructions

Instruction	Description
BL	Branch with Link
BLX <immed>	Branch with Link and Exchange
BLX <reg>	Branch with Link and Exchange

D4.4.3.4 Meaning of Atom elements

R_{PLGZG} The following table describes the meaning of Atom elements in AArch32 T32.

Table D4-18 Meaning of Atom elements in AArch32 T32

Instruction	E	N
B{<c>}	The branch was taken	The branch was not taken
BL	The branch was taken	The branch was not taken
BLX <immed>	The branch was taken	The branch was not taken
BLX <reg>	The branch was taken	The branch was not taken
BX	The branch was taken	The branch was not taken
BXJ	The branch was taken	The branch was not taken
CBNZ	The branch was taken	The branch was not taken
CBZ	The branch was taken	The branch was not taken
Data processing instructions that modify the PC	The branch was taken	The branch was not taken
ISB	The ISB performed a Context synchronization event and is considered as taken	The ISB did not perform a Context synchronization event and is considered as not taken
LDM including the PC	The branch was taken	The branch was not taken
LDR to the PC	The branch was taken	The branch was not taken
POP {..,PC}	The branch was taken	The branch was not taken
RFE	The branch was taken and the PE returned from the Exception	The branch was not taken and the PE did not return from the Exception
TBB	The branch was taken	The branch was not taken
TBH	The branch was taken	The branch was not taken
WFE	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken	The instruction failed its condition code check and is considered as not taken
WFI	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken	The instruction failed its condition code check and is considered as not taken

D4.4.4 Exceptions to Exception element encoding

R_{GZQKS} The following table shows the Exception mapping for exceptions taken to AArch64 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-19 Exception mapping for exceptions taken to AArch64 state

Reason	Type
Branch Target exception	Inst fault
Breakpoint	Inst debug
Exceptions due to SME functionality	Trap
EXLOCK exception	Inst fault
FIQ	FIQ
GPC exception due to data access	Data fault
GPC exception due to instruction access	Inst fault
Guarded Control Stack Data Check exception	Data fault
HVC	Call
Halting debug event	Debug halt
IRQ	IRQ
Illegal execution state	Trap
Instruction Abort	Inst fault
Instruction or event trapped by a control bit	Trap
MemCpy or MemSet	Trap
Misaligned PC	Alignment
PAC Fail	Data fault
PMU exception	Inst debug
SError exception	System Error
SMC due to HCR_EL2.TSC	Trap
SMC other than due to HCR_EL2.TSC	Call
SVC due to HFGITR_EL2.SVC_EL0 or HFGITR_EL2.SVC_EL1	Trap
SVC other than due to HFGITR_EL2.SVC_EL0 or HFGITR_EL2.SVC_EL1	Call
Software Breakpoint Instruction	Inst debug
Software Step	Inst debug
Stack Pointer Misalignment	Alignment
Synchronous Data Abort	Data fault
Traps due to `GCSSTR` and `GCSSTTR`	Trap
UNDEFINED instruction	Trap
Watchpoint	Data debug

R_{GPYBB} The following table shows the Exception mapping for exceptions taken to AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-20 Exception mapping for exceptions taken to AArch32 state

Reason	Type
Breakpoint	Inst fault
FIQ	FIQ
HVC	Call
Halting debug event	Debug halt
IRQ	IRQ
Illegal execution state	Trap
Instruction or event trapped by a control bit	Trap
Prefetch Abort	Inst fault
SError exception	System Error
SMC	Call
SVC	Call
Software Breakpoint Instruction	Inst fault
Synchronous Data Abort	Data fault
UNDEFINED instruction	Trap
Vector Catch exception	Inst fault
Watchpoint	Data fault

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5 About the ETE trace unit

IGBYNF Figure D4-10 shows the stages of trace generation:

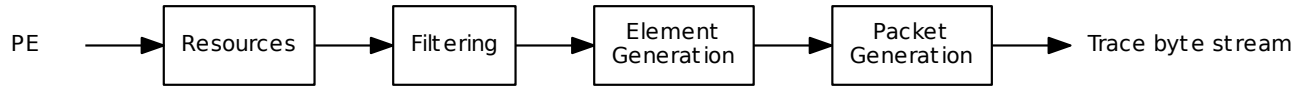

Figure D4-10 Stages of trace generation

D4.5.1 Resetting the trace unit

RPCXJC A trace unit includes a trace unit reset, which resets all of the:

- Trace unit trace registers.
- Trace unit management registers.

RPTZDH When the trace unit Core power domain is powered up, a trace unit reset is applied.

IZXRHG It is IMPLEMENTATION DEFINED whether the system has a mechanism to initiate a trace unit reset on demand.

SWVMHS In a PE with FEAT_TRF, a PE Cold reset causes EDSCR.TFO to be reset to 0 which means that tracing is prohibited after the Cold reset until explicitly permitted by software. If tracing from a Cold reset is required, the debugger needs to ensure any relevant controls, including EDSCR.TFO, are programmed to permit tracing. Programming such registers might involve causing the PE to enter Debug state to ensure the registers can be programmed before the PE starts executing instructions.

RWKLGX When a trace unit reset is applied, the trace unit resets the values of all trace unit registers to the values described in the individual register descriptions.

ICQWFH Some other Arm trace architectures support multiple types of reset for the trace unit.

D4.5.2 System behaviors

RGFMRH The trace unit outputs all of the trace byte stream, without external stimulus, within finite time.

D4.5.2.1 Behavior on enabling

RVBMLV While both of the following are true, the trace unit is enabled:

- TRCPRGCTLR.EN is set to 1.
- The OS Lock is unlocked.

ILFBBP Some Arm trace architectures have a dedicated trace unit OS Lock, whereas ETE depends on the PE OS Lock.

RKBFFQ While the trace unit is enabled, the trace unit can trace all PE execution, except when any of the following are true:

- A trace unit buffer overflow occurs.
- The authentication interface prohibits the tracing of certain pieces of code.
- FEAT_TRF or FEAT_TRBE prohibit the tracing of certain pieces of code.

IKCDMH No sequences of code or PE operations are exempt from this requirement. However, while the trace unit is transitioning from an enabled to a disabled state, or from a disabled to an enabled state, some loss of trace is permitted.

IGDNWY While the trace unit is enabled, writes to most trace unit trace registers might be ignored. It is UNKNOWN whether writes to these registers succeed. When the writes are successful, the behavior of the trace unit is UNPREDICTABLE.

SBXQJH Trace analyzers must not write to most trace unit trace registers while the trace unit is enabled, or if TRCSTATR.IDLE indicates that the trace unit is not idle.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{TPTRW} While the trace unit is enabled or idle, all resources that are visible in the programmers' model might have unstable values. Therefore, a trace analysis tool must be aware that the following values might be dynamically changing as they are being read:

- The Counter values, indicated by the TRCCNTVR<n>.
- The Sequencer state, indicated by TRCSEQSTR.
- The ViewInst start/stop function, indicated by TRCVICTLR.
- The Single-shot Comparator Control status, indicated by the TRCSSCSR<n>.

R_{VNGFG} When the trace unit becomes enabled, the trace unit does not reset the state of any of the resources in the trace unit, including the Counters, the Sequencer, and the ViewInst start/stop function.

S_{LMPNV} While the trace unit is disabled, and before it is enabled, a trace analyzer ensures the trace unit resources are programmed with a valid initial state.

D4.5.2.2 Behavior on disabling

I_{GZPBM} While the trace unit is disabled:

- The trace unit cannot generate trace.
- The trace unit resources are disabled.

For more information, see Behavior of the resources while in the Pausing state.

R_{TMLTF} While either of the following is true, the trace unit is disabled:

- TRCPRGCTLR.EN is set to 0.
- The OS Lock is locked.

I_{WSFFP} Some Arm trace architectures have a dedicated trace unit OS Lock. ETE depends on the PE OS Lock.

R_{ZDTLK} When the trace unit becomes disabled, the trace unit stops generating trace and empties the trace buffers by outputting any data in them.

R_{TNYDD} When the trace buffers are empty, after the trace unit has become idle after becoming disabled, TRCSTATR.IDLE indicates that the trace unit is idle.

R_{TMVLW} When the trace unit becomes disabled, all resources that are visible in the programmers' model retain their values and become stable at those values.

R_{QVYMJ} When the trace unit becomes disabled, when the resources are stable, TRCSTATR.PMSTABLE indicates that the programmers' model is stable.

R_{GLBHL} When the trace unit becomes disabled after the trace unit has generated Event elements, the trace unit outputs the Event elements before TRCSTATR.IDLE indicates that the trace unit is idle.

R_{YFLJT} While the trace unit is disabled, the following are true:

- No trace is generated.
- All trace unit resources and *ETEEvents* are disabled.

D4.5.2.3 Behavior on flushing

I_{XRMWS} The trace unit is allowed to buffer the trace byte stream to make efficient use of system infrastructure.

I_{WHZBD} As the trace unit is allowed to delay the output of the trace byte stream to the system infrastructure, there are system events that require all of the trace byte stream to be observable to other observers in the system.

I_{CXLCR} Making the trace byte stream visible to other observers is known as a trace unit flush.

R_{JLRQH} When any of the following occur, a trace unit flush is requested:

- The trace unit transitions from an enabled to a disabled state.
- The trace capture infrastructure requests a trace unit flush.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A `TSB CSYNC` instruction is executed in a Trace Prohibited region while the Trace Buffer Extension is implemented and enabled.

I_{KGJRL} A trace unit flush might be requested for IMPLEMENTATION DEFINED reasons. For example:

- Before the trace unit enters either:
  - The low-power state.
  - A powerdown state.
- The PE enters Debug state.

I_{ZWHKM} An example of a trace unit flush is one requested on an Arm AMBA ATB interface. For more information on the Arm AMBA ATB interface, see the *AMBA® ATB Protocol Specification (ARM IHI 0032)*.

R_{HGYLG} When a trace unit flush is requested, the trace unit performs the following tasks before responding to the flush request:

1. Encode any remaining elements into trace packets.
2. Complete any packets that are in the process of being generated.
3. Output all trace packets for all PE execution that occurred before the flush request was received.

I_{LMVMT} An example of when the trace unit might need to encode remaining elements into trace packets before a trace unit flush is when there are *Commit elements* that are not yet encoded.

R_{TWBVY} When a trace unit flush occurs while the trace unit is recovering from a trace unit buffer overflow, the trace unit outputs the corresponding *Overflow element* before responding to the flush request.

I_{GHKFH} When a trace unit flush occurs, the trace unit either continues to generate trace or stops generating trace, depending on what condition caused the trace flush. For example, if a flush occurs because the trace unit is entering a disabled state, then tracing becomes inactive after the trace flush.

R_{TTDBB} When a condition causes both a trace unit flush and the trace unit to stop generating trace, the trace unit stops generating trace before responding to the flush request, and before indicating that the trace unit is idle.

I_{NHBMZ} On entry to Debug state, Arm recommends that the *Exception element* indicating entry to Debug state is included in the flushed trace data if tracing is active.

R_{PFHWW} When a trace unit flush is requested, the trace unit outputs the data within a finite period.

R_{DKFRL} When a trace unit flush is requested and the cause of the flush request requires an acknowledgment, the trace unit generates the acknowledgment within a finite period.

I_{SCBMG} The flush request mechanism on AMBA ATB is an example of a cause of a flush request that also requires an acknowledgment.

D4.5.2.4 Low-power state

I_{GHHNW} The low-power state in the trace unit is a mechanism to improve energy efficiency during periods where trace generation is limited.

Scenarios where the trace unit might be in the low-power state are any of the following:

- The PE is in a low-power state.
- The PE is in Debug state.

R_{LHDSS} The trace unit is only permitted to be in the low-power state when any of the following are true:

- The PE is in a low-power state due to the Wait for Event mechanism.
- The PE is in a low-power state due to the Wait for Interrupt mechanism.
- The PE is in Debug state.
- The trace unit is Disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.2.5 Trace unit behavior when the PE is in a low-power state

I_{MSTWP} The PE that is being traced might support a low-power state where no execution occurs. This low-power state might be invoked, for example, when the PE executes a `WFI`, `WFIT`, `WFE`, or `WFET` instruction.

I_{DPFXN} When the PE is in a low-power state, it might be advantageous if the trace unit also enters a low-power state.

R_{FBMYZ} It is IMPLEMENTATION DEFINED whether a trace unit supports the low-power state.

R_{WMPTL} While the trace unit is in the disabled state, the trace unit does not stop the PE from entering a low-power state.

R_{YLDDV} While the trace unit is in Low-power Override Mode, the trace unit does not affect the operation of the PE.

D4.5.2.6 Trace unit behavior in the low-power state

R_{FMXFM} While the trace unit is enabled, when the trace unit enters the low-power state, the trace unit continues to appear enabled throughout the time it is in the low-power state.

R_{KQVNN} When the trace unit enters or leaves the low-power state, the trace unit does not lose resource events that are in transition through the trace unit, except those permitted when moving through the Pausing state of the resources. For more information on the resource events that are permitted to be lost when in the Pausing state, see Behavior of the resources while in the Pausing state.

I_{RVKHK} Observation of resource events that are in transition through the trace unit when it enters the low-power state might not occur until after the trace unit leaves the low-power state.

R_{RGFJY} While the trace unit is not in the low-power state, and before it enters the low-power state, the resources enter the Paused state. For more information on the pausing of resources, see Behavior of the resources while in the Paused state.

I_{MXHHN} If `WFI` and `WFE` instructions are classified as P0 instructions, see TRCIDR2.WFXMODE, and the trace unit enters the low-power state as a result of a `WFI` or `WFE` instruction, Arm strongly recommends that the following elements are generated before the trace unit enters the low-power state:

- The *Atom element* that represents a `WFI`, `WFIT`, `WFE`, or `WFET` instruction.
- Any pending *Commit elements*.

R_{LBDSM} While the trace unit is in the low-power state, the trace unit does not generate trace, including *ETEEvent* trace.

R_{MFBDT} While the trace unit is in the low-power state, the resources remain in the state that they were in before the trace unit entered the low-power state.

I_{QXBYK} The resources are:

- The Counters.
- The Sequencer.
- The ViewInst start/stop function.
- The Single-shot Comparator Controls.

R_{FHYHC} While the trace unit is in the low-power state, the trace unit drives all External Outputs low.

R_{DNKDV} While the trace-unit is in the low-power state, the PE and external debuggers ability to access the trace unit trace registers and trace unit management registers is unaffected.

R_{XTBQX} While the trace unit is in the low-power state, when a trace protocol synchronization request occurs, the trace unit handles the trace protocol synchronization request correctly. For more information on how the trace unit handles trace protocol synchronization requests, see Trace protocol synchronization.

R_{TWQJT} While the trace unit is in a retention state, external debugger accesses to the trace unit behave as if there is no power to the trace unit Core power domain.

I_{RCXZX} While the trace unit is in the low-power state, the trace unit might not recognize external events, such as the assertion of any External Inputs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{BPQTL} While the trace unit is in the low-power state, it is IMPLEMENTATION DEFINED whether the cycle counter continues to count or not.

I_{VTRBC} While the trace unit is in the low-power state, timestamp requests might be ignored.

I_{ZTDMB} It is possible that the trace unit might intermittently leave and reenter the low-power state while the PE is in a low-power state. If this happens, the trace unit resources might become intermittently active during this time. In addition, trace generation might also become intermittently active, and this means that the trace unit might output some packets. This behavior is IMPLEMENTATION DEFINED.

I_{ZVDSF} There is no additional requirement for the trace unit to generate a *Trace Info element* or *Trace On element* when leaving the low-power state. However, if the trace unit entered the low-power state because the PE was in Debug state, the normal requirements for restarting trace after leaving Debug state apply, including generation of a Trace On element. For more information on the PE Debug state, see Trace unit behavior while the PE is in Debug state.

I_{LQFRD} The trace unit can be programmed so that it does not enter the low-power state, by enabling Low-power Override Mode. Low-power Override Mode is selected using TRCEVENTCTL1R.LPOVERRIDE.

R_{VHSFL} When Low-power Override Mode is enabled, the resources continue operating and the trace unit can generate trace.

I_{FRMMP} Low-power Override Mode does not affect the operation of the PE, however it is not required to prevent the PE from entering a low-power state. This means that even though the trace unit can generate trace, it might only generate Event elements.

D4.5.3 Trace unit behavior while the PE is in Debug state

R_{XJXQS} While ViewInst is active, when the PE enters Debug state, the trace unit generates an Exception element which indicates that the PE has entered Debug state.

R_{YMJFJ} When the PE enters Debug state, ViewInst becomes inactive, and remains inactive throughout the time the PE is in Debug state.

R_{DPKSC} While the PE is in Debug state, the trace unit does not trace:

- Instructions that are executed.
- The effects of instructions that are executed.
- Exceptional occurrences.

R_{HBNFJ} When the PE exits Debug state and ViewInst becomes active, the trace unit generates a Trace On element.

R_{TGFHM} While the PE is in Debug state, the ViewInst start/stop function maintains its state.

I_{WFYLQ} If an Exceptional occurrence occurs between the PE exiting Debug state and the PE executing the first instruction, the value of TRCRSR.TA is used to determine if the Exceptional occurrence is traced. In general, if the entry to Debug state was traced then TRCRSR.TA will be set to 1, and therefore this Exceptional occurrence on exit from Debug state is traced.

I_{NPQLT} If a PE Reset occurs when the PE is in Debug state this is treated as leaving Debug state. This means that a Trace On element and an Exception element indicating a PE Reset are traced if tracing is not prohibited and either of the following are true:

- TRCRSR.TA is 1.
- Forced tracing of PE Resets is active.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.4 Trace unit behavior on a trace unit buffer overflow

R_{PQGXB} When a trace unit buffer overflow occurs, trace generation becomes inoperative until the trace unit can recover from the overflow.

R_{RQHFH} When a trace unit buffer overflow occurs, the trace unit does not output a partial trace packet, that is, the trace unit can only output complete packets.

I_{TDCNT} The *Overflow element* indicates to a trace analyzer that a trace unit buffer overflow has occurred. For more details on the generation of an *Overflow element*, see Overflow element.

R_{DQBDH} When the trace unit recovers from a trace unit buffer overflow, the following occur:

- Trace protocol synchronization is requested.
- Trace protocol synchronization occurs before the trace unit outputs any packets.

I_{VQYYH} When an Overflow packet is generated, the trace unit might output any of the following packets before it outputs an Alignment Synchronization packet:

- Event packet.
- Overflow packet.
- Discard packet.
- Ignore packet.

I_{YYNRQ} Arm recommends that the Alignment Synchronization packet is the first packet output after the Overflow packet.

D4.5.5 Instrumentation extension

I_{TLZTW} FEAT_ITE provides a mechanism to allow software to inject instrumentation information into the ETE trace stream, allowing the ETE trace to be augmented with software-defined information that can aid with debugging and interpretation of the ETE trace.

Examples of instrumentation information include the following:

- Context information about the program being executed.
- Parameters to functions.
- Data addresses and values of variables and other data structures.

I_{GRJLV} All of the following registers control the behavior of FEAT_ITE:

- TRCITECR_EL1, providing EL1 with controls over FEAT_ITE.
- TRCITECR_EL2, providing EL2 with controls over FEAT_ITE.
- TRCITEEDCR, TRCCONFIGR and EDSCR.TFO, providing an external debugger with controls over FEAT_ITE.
- MDCR_EL3.EnITE, providing an EL3 trap control over access to TRCITECR_EL1 and TRCITECR_EL2.
- HDFGRTR2_EL2.nTRCITECR_EL1 and HDFGWTR2_EL2.nTRCITECR_EL1, providing EL2 trap controls over access to TRCITECR_EL1.

I_{CMDFW} The controls provided in TRCITECR_EL1 and TRCITECR_EL2 are similar in function to TRFCR_ELx provided for FEAT_TRF.

R_{FLGVJ} If self-hosted trace is enabled, then the controls in TRCITECR_EL1 and TRCITECR_EL2 control when Instrumentation trace is prohibited.

See `TraceInstrumentationAllowed()`.

R_{RTKDL} If self-hosted trace is disabled and TRCCONFIGR.ITO is 1, then the fields in TRCITEEDCR control when Instrumentation trace is prohibited.

See `TraceInstrumentationAllowed()`.

R_{DKCFL} If self-hosted trace is disabled and TRCCONFIGR.ITO is 0, then the fields in TRCITEEDCR, TRCITECR_EL1, and TRCITECR_EL2 control when Instrumentation trace is prohibited.

See `TraceInstrumentationAllowed()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{SRTGR} TRCCONFIGR.ITO and TRCITEEDCR allow an external debugger to one of the following:

- Completely control when Instrumentation trace is allowed and prohibited.
- Share control of when Instrumentation trace is allowed with software running on the PE.

D4.5.6 Trace unit power states

I_{GNFXM} The Arm architecture defines the following power states:

Normal
The trace unit Core power domain is fully powered up and the trace unit registers are accessible.

Standby
The trace unit Core power domain is on, but there are measures to reduce energy consumption. Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

Retention
The OS takes some measures, including IMPLEMENTATION DEFINED code sequences and registers, to reduce energy consumption. Trace unit registers cannot be accessed. A trace unit reset does not occur on leaving retention.

Powerdown
The OS takes some measures to reduce energy consumption by turning the trace unit Core power domain off. Trace registers cannot be accessed. A trace unit reset occurs on leaving powerdown.

I_{MNDVQ} A trace unit might support a low-power state, which is equivalent to the standby state.

I_{BMSVN} A trace unit might support a retention state or a powerdown state, and both of these states are considered to be a state where the trace unit Core power domain is powered down.

I_{ZVBZF} If the trace unit is implemented in a power domain which is separate from the PE power domain, all of the following are true:

- The trace unit Core power domain might be able to be powered down without powering down the PE power domain.
- The trace unit Core power domain is always powered down when the PE power domain is powered down.

I_{CNZJH} A read of TRCPDSR returns information about the current state of the trace unit.
Table D4-21 describes the meanings of the returned value.

Table D4-21 Meaning of TRCPDSR values

STICKYPD	POWER	Meaning
`0b0`	`0b1`	The trace unit Core power domain is powered and the trace unit registers are accessible.
`0b1`	`0b1`	The trace unit Core power domain is powered and the trace unit registers are accessible. A trace unit reset or power interruption has occurred so the trace unit register state might not be valid.

R_{CMKXK} When the trace unit Core power domain transitions from powered down to powered up, if the trace unit register state has been preserved over the powerdown, then TRCPDSR.STICKYPD is restored to the value before powerdown.

R_{FQMXQ} When the trace unit Core power domain transitions from powered down to powered up, if the trace unit register state has not been preserved over the power down then TRCPDSR.STICKYPD is set to `0b1`.

Note

ETE only supports a single power domain and therefore TRCPDSR.POWER is always `0b1`.

I_{FRMBB} A system might support a Debug power domain that contains the interface between the trace unit and the external debugger. The Debug power domain usually needs to be powered up when the external debugger needs to connect to the system. Such a Debug power domain is described in Resets and power domains.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{GYLKD} If the trace unit Core power domain can be powered down independently of the Debug power domain, Arm recommends that the system implements an external debug component with a powerup request mechanism which can request the trace unit Core power domain to be powered up.

R_{ZNSNS} Arm strongly recommends the powerup request mechanism is a CoreSight Class `0x9` ROM Table containing a parent entry for the trace unit. A parent entry of a component is one of:

- An entry in the ROM table that locates the component.
- An entry in a first ROM table that locates a second ROM table that includes a parent entry for the component.

The second ROM table is a descendant of the first ROM table.

I_{TLYXG} This definition of a parent entry is recursive, and therefore the powerup request mechanism might be high up in a hierarchy of ROM tables. The ROM table and any descendants might describe other debug components, including debug components for other PEs. The ROM table might have parent entries in other ROM tables, and those parent entries might also have a powerup request mechanism in those ROM tables.

R_{ZPCZC} If the powerup request mechanism is implemented, in the Class `0x9` ROM Table containing the powerup request mechanism for the trace unit:

- The POWERIDVALID bit in the parent entry must be `0b1`.
- The POWERID field in the parent entry has an IMPLEMENTATION DEFINED value.

I_{DXHPS} It is IMPLEMENTATION DEFINED whether the trace unit Core power domain is the PE Core power domain or some other power domain. For more information on the CoreSight Class `0x9` ROM Table, see the Arm® CoreSight™ Architecture Specification (ARM IHI 0029).

D4.5.7 Visibility of the PE operation

I_{BMPFH} This section describes the ability of the trace unit to trace the execution of the operation of the PE.

R_{QCFMC} When the trace unit performs indirect reads of PE System registers, the trace unit complies with the rules associated with Context synchronization events.

R_{QHTYC} When the trace unit performs indirect reads of PE System registers, the trace unit complies with the rules associated with the `TSB CSYNC` instruction as defined in The Trace Buffer Extension.

R_{QCFSS} When instructions are executed outside of any Trace Prohibited region, the trace unit observes the execution.

R_{CDNKX} When observable instructions are executed, the trace unit observes all execution before a PE Context synchronization event occurs, as defined in The Trace Buffer Extension.

R_{QMBKJ} When an Exceptional occurrence occurs outside of any Trace Prohibited region, the trace unit observes the *Exceptional occurrence*.

I_{XBJFP} Executing a `TSB CSYNC` instruction generates a Trace synchronization event as defined in The Trace Buffer Extension.

R_{FCBLJ} When a `TSB CSYNC` instruction is executed in a Trace Prohibited region, the `TSB CSYNC` instruction does not become microarchitecturally-finished until the resources are in the Paused state or the trace unit is in the Idle or Stable state.

I_{JYJDZ} After a `TSB CSYNC` instruction executed inside a Trace Prohibited region generates a Trace synchronization event, the Trace synchronization event is microarchitecturally-finished when the *trace operation* has microarchitecturally-finished for every instruction before the Context synchronization event before the `TSB CSYNC` instruction that generated the Trace synchronization event.

For more details on the `TSB CSYNC` instruction, see R_{MRVPT}.

R_{XLVQM} When a `TSB CSYNC` instruction executed in a Trace Prohibited region becomes microarchitecturally-finished, the trace unit generates no more trace until the PE leaves the Trace Prohibited region.

I_{CZLXW} When a `TSB CSYNC` is executed in a Trace Prohibited region, the rules around generation of a trace flush and requiring no more trace to be generated in the Trace Prohibited region mean that only whole trace packets are output, and the last byte output is the end of a packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{GSXJJ} These rules ensure that no new trace is generated and allows various System registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

R_{XRWPV} When the trace unit becomes enabled in a Trace Prohibited region, the trace unit generates no trace until the PE leaves the Trace Prohibited region.

Note

This ensures that no trace is generated until the PE leaves the Trace Prohibited region, and therefore allows various System registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

I_{KXDDS} The trace operation as defined in The Trace Buffer Extension can be split into operations that are performed by one of the following:

- The PE.
- The ETE trace unit.
- The trace buffer.

The operation of the trace unit is defined by the *ETE trace operation*.

R_{RFJQN} If the Trace Buffer Unit is implemented and enabled, when a Trace synchronization event occurs, and after all of the trace byte stream generated by the trace unit is flushed to the trace buffer, the Trace synchronization event completes.

I_{CNJYL} Table D4-22 describes the labels used in the ordering diagrams present in this Chapter.

Table D4-22 Labels for ordering diagrams

Notation	Name	Description
po	program-order	*head* is in program order after *tail*.
rf	Reads-from	*tail* Reads-from *head*.
co	Coherence-after	*head* is Coherence-after *tail*.
fr	from-read	As *co*, except that the operation at *head* is a read.
ob	Observed-by	*tail* is Observed-by *head*. Only applies for different Observers.
tb	traced-by	*head* is the *trace operation* for the instruction at *tail*.
gb	generated by	*head* is an operation generated by the instruction at *tail*.
seo	speculative execution-order	The PE speculated that the instruction at *head* was executed after *tail*, but the instruction was later Canceled. An *seo* arrow might be paired with a *can* arrow that shows this.
can	canceled	The instruction at *tail* was Canceled when the instruction at *head* was resolved.

D4.5.7.1 ETE trace operation

R_{YCJXC} Each instruction has all of the following state information:

- PC.
- PSTATE.T.
- PSTATE.EL.
- The Security state.
- CONTEXTIDR_EL1.PROCID.
- CONTEXTIDR_EL2.PROCID.
- TRFCR_EL1.
- TRFCR_EL2.
- MDCR_EL3.STE.
- TxNestingLevel.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{GCLMS} The trace information generated contains *Address information* in *Target Address elements*, *Source Address elements*, *Exception elements*, and *Q elements*. The *Address information* contains:

- The virtual address of an instruction.
- The instruction set, known as the sub_isa.

I_{JTLPL} The trace information generated contains *Context information* in *Context elements*. The *Context information* contains:

- The current Security state.
- The current Exception level.
- The current Execution state.
- The current Context identifier, as stored in CONTEXTIDR_EL1.PROCID.
- The current Virtual context identifier, as stored in CONTEXTIDR_EL2.PROCID.

R_{WBCRN} When an instruction is executed and all the trace elements for the instruction have been generated, the trace operation for the instruction is complete.

I_{WXNXB} Trace elements generated for an instruction might include:

- Global timestamp elements.
- Cycle count elements.
- Speculation resolution elements.

I_{FKTKY} For example, the tracing of PE execution is where:

- Resolved instruction A is executed in program order before a Resolved instruction B.
- t_A is all the trace elements that are generated due to the tracing of instruction A.
- t_B is all the trace elements that are generated due to the tracing of instruction B.
- The trace elements for t_A must be observed before t_B.

This is shown in Figure D4-11.

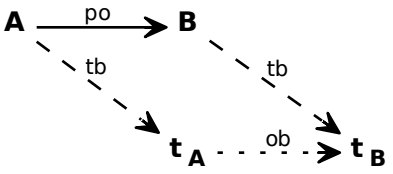

Figure D4-11 Trace operation

D4.5.7.2 Impact on PE behavior

I_{LLKFT} The ETE architecture places no requirements on the impact that trace generation has on the functional performance of a PE. Arm expects that trace unit implementations are designed according to the market requirements of the PEs being traced, and according to the trace requirements for those PEs. For some markets and trace requirements, the trace solution might always have some performance impact on the PE and the ETE architecture does not prohibit this.

D4.5.7.3 Behavior on a PE Warm reset

R_{YYHBF} A PE Warm reset does not cause a Trace unit reset.

Note

This ensures that tracing is possible through a PE Warm reset.

I_{QBSCX} A PE Warm reset might occur at the same time as a Trace unit reset, however, these are asynchronous and unrelated events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.7.4 Instruction block

I_{TYXHR} How instructions are executed can vary significantly between PE designs. To allow for these variations the ETE architecture allows some flexibility within the filtering model. Rather than applying the filtering model to individual instructions it is applied to blocks of instructions.

R_{BQTBL} An instruction block contains one or more instructions.

R_{GDZBX} An instruction block can contain zero or one *P0 instruction*.

R_{CVJQH} When an instruction block is generated which contains a *P0 instruction*, the instruction block has the *P0 instruction* as the last instruction in the block.

R_{HPJTP} *Exceptional occurrences* do not occur between instructions in an instruction block.

R_{LDJXZ} The addresses of the instructions within an instruction block are sequential.

I_{JCQHC} The number of instructions in a block can vary from block to block and can vary each time the same sequence of instructions are executed.

I_{HRBJG} For example, the tracing of an instruction block is where:

- Resolved instruction A is executed in program order before a Resolved instruction B.
- t_A is all the trace elements that are generated due to the tracing of instruction A.
- t_A is all the trace elements that are generated due to the tracing of instruction B.

This is shown in Figure D4-12.

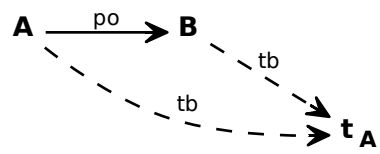

Figure D4-12 Instruction block trace operation

D4.5.7.5 Exposing speculation

I_{DCDNQ} For some PE microarchitectures, the tracing of execution-order only might not be achievable. The ETE architecture provides the ability to trace speculatively executed instructions.

R_{TRVLX} When speculative instructions are observed, the trace unit indicates whether each instruction is resolved or canceled with a resolve operation or a cancel operation.

R_{PPJSK} A resolve operation indicates that one or more instructions have, or will be, architecturally executed.

R_{WZBLY} A cancel operation indicates that one or more instructions, although observed by the trace unit, did not architecturally execute.

I_{KQYZB} There is no requirement to expose any speculation to the trace unit.

I_{DKDHD} For example, the tracing of speculation execution is where:

- S is executed in speculative execution-order after a Resolved instruction A.
- A is executed in program order before a Resolved instruction B.
- S is not in speculative execution-order after B.
- Q is executed in speculative execution-order after a Resolved instruction B.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This is shown in Figure D4-13.

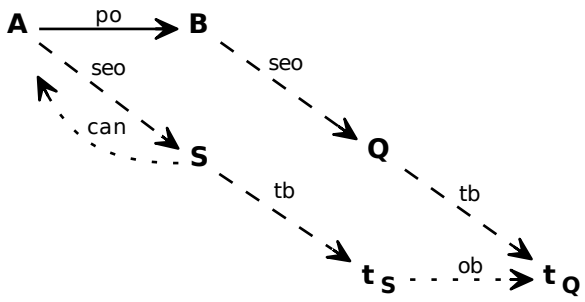

Figure D4-13 Observation of Speculative Trace operation

D4.5.7.6 Trace Prohibited Regions

I_{THCBC} Trace Prohibited regions are instruction address regions or periods of execution by the PE that are not to be traced. Instructions and Exceptional occurrences which are not prohibited are not necessarily traced because the trace unit has a number of trace filtering functions to limit the amount of trace generated to the sections or periods of interest.

I_{RJYNL} An executable program might contain regions of code that are prohibited to trace. These regions might be associated with another Security state, or with the PE entering a privileged mode so that it can execute the instructions that are contained within them.

Tracing might be prohibited while the PE is operating in certain states or modes. For example:

- Non-invasive debug might be prohibited while the PE is in Secure, Realm, or Root state.
- FEAT_TRF might prohibit tracing.
- FEAT_TRBE might prohibit tracing.

Trace might also become prohibited if, while tracing program execution, an authentication interface changes the currently permitted level of non-invasive debug. For example, if trace is permitted and active while the PE is operating in Secure state, and then the permitted level of non-invasive debug changes from being permitted for Secure state, to not permitted, then trace becomes prohibited.

I_{ZPXRJ} The Trace Buffer Extension describes when FEAT_TRBE prohibits tracing.

R_{HYZLQ} If an optional authentication interface is implemented, while non-invasive debug is disabled in a Security state other than Non-secure state according to that optional authentication interface and while self-hosted trace is disabled, then for execution in that Security state, the PE executes in a Trace Prohibited region.

I_{NVSDD} An example of an optional authentication interface is the CoreSight Authentication interface. For more information, see the Arm® CoreSight™ Architecture Specification (ARM IHI 0029).

R_{FFVYM} While the PE is executing code from a Trace Prohibited region, the trace unit does not trace instructions or Exceptional occurrences, including PE Resets.

R_{KTMLZ} While the PE is executing code from a Trace Prohibited region, instruction Address Comparators do not match on any instructions in the Trace Prohibited region.

R_{SZRZR} While cycle counting is enabled and the PE is executing code from a Trace Prohibited region, the cycle counter continues to count.

I_{MCCBH} When the PE leaves a Trace Prohibited region and tracing restarts, the cycle counter includes cycles spent in the Trace Prohibited region in the cycle count.

I_{SDSGK} The behavior of the resources when entering a Trace Prohibited region is defined in Behavior of the resources while in the Paused state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{VHRGM} While the PE is executing code from a Trace Prohibited region, the trace unit does not output any trace that might provide information about the execution in the Trace Prohibited region.

I_{JSKLD} Examples of information about execution in a Trace Prohibited region that trace might provide are the context of execution, instruction addresses, and the address of the first instruction in the Trace Prohibited region.

I_{MKQWS} The most common cause of an entry into a Trace Prohibited region is an *Exceptional occurrence* or Context synchronization event.

R_{THBVD} When an *Exceptional occurrence* that must be traced causes the PE to enter a Trace Prohibited region, the trace unit generates an *Exception element* that indicates the exception type.

R_{CKZMR} When the PE enters a Trace Prohibited region and there are unresolved speculative *P0 elements* remaining in the trace byte stream, when the resolution of the speculative elements is known the trace unit generates the appropriate *Commit elements* or *Cancel elements*.

R_{RXMJF} When the PE leaves a Trace Prohibited region and ViewInst is active, that is, any filtering applied dictates that ViewInst is active, the trace unit generates a *Trace On element*.

I_{QGFJF} The purpose of the trace unit generating a *Trace On element* when the PE exits a Trace Prohibited region and ViewInst is active is to indicate to the trace analyzer that there has been a discontinuity in the trace element stream.

I_{JBFTB} If the PE leaves a Trace Prohibited region other than when a Context synchronization event occurs, the Trace Prohibited region is permitted to extend up to the next Context synchronization event. Typically, a PE leaves a Trace Prohibited region via a Context synchronization event, but a PE might leave a Trace Prohibited region when the authentication interface changes, or when moving from a higher Security state to a lower Security state without an exception return.

I_{DMXPF} If an *Exceptional occurrence* occurs between the PE exiting a Trace Prohibited region and the PE executing the first instruction, the value of TRCRSR.TA is used to determine if the *Exceptional occurrence* is traced.

D4.5.7.7 Multi-threaded processor

R_{KBZTZ} Processors with multiple threads or PEs have a trace unit for each thread or PE.

I_{KHSWQ} The processor might support enabling and disabling of threads, either at PE Reset time or dynamically. The trace units for the threads that are disabled might behave in one of the following ways:

- The trace unit Core power domain is powered down.
- The trace unit Core power domain is held in the trace unit reset state.

I_{RFSNL} Arm recommends that the trace units for threads that are permanently disabled are not visible: either they are not included, or they are marked as not present in any ROM tables that describe the system.

D4.5.7.8 Sharing between multiple PEs

I_{TSWXS} Previous Trace architectures have allowed the trace unit to be shared between multiple PEs.

R_{TLJSQ} A trace unit only traces a single PE, that is, it cannot be shared between multiple PEs.

D4.5.8 Speculation resolution

I_{RKYCD} The trace unit implements a maximum speculation depth, that is, the maximum permitted number of *P0 elements* that can be speculative at any instance. TRCIDR8.MAXSPEC indicates the IMPLEMENTATION DEFINED maximum speculation depth.

R_{GLQPL} The trace unit never outputs more speculative *P0 elements* than the maximum speculation depth.

I_{KCFGW} If a trace unit is not exposed to any speculative execution, then Arm recommends that the trace unit implements a maximum speculation depth of zero, and in this case:

- *Cancel elements* are not generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- *Commit elements* are generated after each *P0 element*, causing each *P0 element* to be immediately resolved when it is generated. The instruction trace protocol implicitly generates these *Commit elements* for each *P0 element*, meaning that explicit Commit packets are not required.
- *Mispredict elements* are not generated.

IQLKDL The ETE architecture defines *Commit element* and *Cancel elements* to allow the speculation of the *P0 elements* to be resolved by the trace analyzer. The trace unit is required to calculate the number of *P0 elements* which are committed or canceled. There are many methods by which these numbers can be calculated, but in the generic case the trace unit can use the following mathematical procedure.

The PE can speculatively indicate blocks of instructions to the trace unit. Each block of instructions is given a tag where *tag* belongs to 0. . .*m* and *m* is the number of rewind points supported by the PE.

The number of instructions per block can be random from the set *N* and there is a maximum of one *P0 instruction* per block. The order in which the tags are used can be random, but a tag cannot be reused until the previous block with that tag has been resolved, canceled or merged.

This procedure generates a transform from the potentially random sequence of core tags to a more usable space. The transform T evolves over time as the tags are reused and provides the mapping onto the new space,

$$T_t = [c_0, \ldots, c_m]$$

c_i is the mapping for core tag *i*.

c_i belongs to 0. . .*q*, where *q* is greater than *m*.

D4.5.8.1 Initialization

IHRDQL To perform the necessary calculations, the trace unit tracks the transform of the last resolved block. γ_t is the last committed indicator. The algorithm is initialized at *t=0* to

$$\forall i \in 0, \ldots, m : T_0[i] = \gamma_0$$
$$x_0 = \gamma_0$$

D4.5.8.2 New block operation

ISMGSG The sequence of the numbers in the transformed space, x_t, is defined by the following equation:

$$x_{t+1} = \begin{cases} |(x_t + 1) \bmod q| & \text{If a traced P0 instruction} \\ x_t & \text{Otherwise} \end{cases}$$

$$T_{t+1}[tag_t] = \begin{cases} |(x_t + 1) \bmod q| & \text{If a traced P0 instruction} \\ T_t[tag_t] & \text{Otherwise} \end{cases}$$

D4.5.8.3 Resolved operation

IBJFLR The PE can resolve one or more blocks in an atomic operation. This is performed by indicating the youngest block's tag to be resolved, and by inference all older blocks.

The number required by the *Commit element* is calculated by

$$n_+ = |(T_t[l] - \gamma_t) \bmod q|$$

The state of the transform is updated by

$$\gamma_{t+1} = T_t[l]$$

where *l* is the *youngest block's tag*.

D4.5.8.4 Cancel operation

IHJYQH The PE can cancel one or more blocks in an atomic operation. This is performed by indicating the oldest block to be canceled.

The number required by the *Cancel element* is calculated by

$$n_- = |(x_t - T_t[r]) \bmod q|$$

The state of the transform is updated by

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

$$x_{t+1} = T_t[r]$$

where *r* is the *oldest block's tag*.

D4.5.9 Filtering trace generation

I_{CCWZG} The amount of trace that can be generated by the trace unit can be significant. Not all the operations of the PE are always relevant. The amount of trace generated can be reduced by the use of the trace unit filter functions.

D4.5.9.1 ViewInst function

R_{GQFSB} The filtering function of the instruction trace is expressed as a calculation evaluated for each instruction.

$$\text{ViewInst}_i = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug state} \\ S_i \wedge I_i \wedge E_i \wedge N_i & \text{Otherwise} \end{cases}$$

$$\begin{aligned} S_i &= \text{ViewInst start/stop function} \\ I_i &= \text{ViewInst include/exclude function} \\ E_i &= \text{Exception level filtering} \\ N_i &= \text{Resource event based filtering} \end{aligned}$$

R_{BZXFH} While ViewInst is high, the trace unit traces all instructions.

I_{MJMCV} Instructions for which ViewInst is low might be traced. This might be as a result of tracing the next *P0 element* or optimizations in the trace unit.

R_{XPNZL} When ViewInst is high for an instruction in an instruction block, the trace unit traces the entire instruction block.

R_{KHDNX} When ViewInst becomes high, the trace unit traces the next *P0 instruction* or Exceptional occurrence.

I_{NSMSV} Some instruction types cause the trace unit to generate *P0 elements*, so that they are explicitly traced. Other instruction types however are not explicitly traced. The execution of these other instruction types can be inferred from the *P0 elements*. This means that the following scenario is possible:

- While ViewInst is high, some instructions are executed. This means that ViewInst is indicating that those instructions must be traced. However, none of the executed instructions cause the trace unit to generate a *P0 element*, therefore none of the instructions are traced.
- ViewInst then goes low.
- The PE then executes an instruction that, whenever ViewInst is high, causes the trace unit to generate a *P0 element*.

In this scenario, although ViewInst is low when the instruction in step 3 is executed, indicating that the instruction is not traced, tracing of the instruction is permitted because this is the only way that the preceding instructions can be traced.

I_{FGSBW} There is no requirement for the target address of a *P0 instruction* or Exceptional occurrence to be traced if ViewInst has transitioned to a low state by the time program execution has moved to the target.

I_{LCXHX} Unless the target instruction block is traced, any *Target Address elements* indicating the target address of a *P0 instruction* or Exceptional occurrence cannot be relied on.

D4.5.9.1.1 Resource event based filtering

R_{BQNWP} The resource event based filtering part of the ViewInst function is expressed as the following equation:

$$N_i = \text{Fn}(\text{TRCVICTLR.EVENT.SEL}, \text{TRCVICTLR.EVENT.TYPE})$$

Where Fn(TRCVICTLR.EVENT.SEL, TRCVICTLR.EVENT.TYPE) selects the combination of Resource Selectors used for resource event based filtering.

I_{KMXNS} The timing of the resource event based filtering is IMPLEMENTATION SPECIFIC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{DWWVR} Resource event based filtering can be used to make ViewInst active based on system conditions or on trace unit resources. For example:

- Sampling based on cycle counts.
- Activating tracing on the nth function call.
- Performance Monitoring Unit events.

R_{NBYKR} When an instruction block is processed by the trace unit during a cycle, the trace unit samples the ViewInst function resource event input during that cycle.

R_{SRMMD} When no instruction blocks are processed by the trace unit during a cycle, the trace unit does not sample the ViewInst function resource event input during that cycle.

D4.5.9.1.2 Exception level filtering

I_{HNPYV} This function provides a simple method of filtering out information about different Exception levels without the need to use of additional resources.

R_{LWYJR} The Exception level based filtering part of the ViewInst function is expressed as the following equation:

$$E_i = \begin{cases} \neg\text{TRCVICTLR.EXLEVEL_S_EL0} & \text{Secure EL0} \\ \neg\text{TRCVICTLR.EXLEVEL_S_EL1} & \text{Secure EL1} \\ \neg\text{TRCVICTLR.EXLEVEL_S_EL2} & \text{Secure EL2} \\ \neg\text{TRCVICTLR.EXLEVEL_S_EL3} & \text{EL3} \\ \neg\text{TRCVICTLR.EXLEVEL_NS_EL0} & \text{Non-secure EL0} \\ \neg\text{TRCVICTLR.EXLEVEL_NS_EL1} & \text{Non-secure EL1} \\ \neg\text{TRCVICTLR.EXLEVEL_NS_EL2} & \text{Non-secure EL2} \\ \neg(\text{TRCVICTLR.EXLEVEL_RL_EL0} \oplus \text{TRCVICTLR.EXLEVEL_NS_EL0}) & \text{Realm EL0} \\ \neg(\text{TRCVICTLR.EXLEVEL_RL_EL1} \oplus \text{TRCVICTLR.EXLEVEL_NS_EL1}) & \text{Realm EL1} \\ \neg(\text{TRCVICTLR.EXLEVEL_RL_EL2} \oplus \text{TRCVICTLR.EXLEVEL_NS_EL2}) & \text{Realm EL2} \end{cases}$$

D4.5.9.2 ViewInst start/stop function filtering

I_{PJQSC} The ViewInst start/stop function is useful when the requirement is to trace a particular piece of code with all the functions that the piece of code calls.

The ViewInst start/stop function uses the Single Address Comparators and the PE Comparator Inputs to define start points and stop points.

A start point is any of the following:

- The instruction address which matches a Single Address Comparator selected for the ViewInst start/stop function using TRCVISSCTLR.START.
- The instruction address which matches a PE Comparator selected for the ViewInst start/stop function using TRCVIPCSSCTLR.START.

A stop point is any of the following:

- The instruction address which matches a Single Address Comparator selected for the ViewInst start/stop function using TRCVISSCTLR.STOP.
- The instruction address which matches a PE Comparator selected for the ViewInst start/stop function using TRCVIPCSSCTLR.STOP.

Multiple start points can be selected. Multiple stop points can be selected.

R_{MVDJT} When a start point is encountered, the ViewInst start/stop function becomes active for the instruction at the start point.

R_{CDFBP} When a stop point is encountered, the ViewInst start/stop function becomes inactive immediately after the instruction at the stop point.

R_{LMQPR} When the ViewInst start/stop function causes ViewInst to become active, the trace unit traces the instruction at the start address.

R_{BWXLS} When the ViewInst start/stop function causes ViewInst to become inactive, the trace unit traces up to and including the instruction at the stop address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{XHFYQ} While a ViewInst start/stop function start address is the same as a stop address, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

R_{KCYTR} The ViewInst start/stop function part of the ViewInst function is expressed as the following equations:

$$\text{TRCVICTLR.SSSTATUS}_{i+1} = S_i \wedge \neg\text{Stop}_i$$
$$S_i = \text{TRCVICTLR.SSSTATUS}_i \vee \text{Start}_i$$

If TRCIDR4.NUMPC == `0b0000`, then:

$$\text{Start}_i = \sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.START}[n])$$
$$\text{Stop}_i = \sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.STOP}[n])$$

If TRCIDR4.NUMPC != `0b0000`, then:

$$\text{Start}_i = \sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.START}[n]) \vee \sum_m (\text{PECMP}[m] \wedge \text{TRCVIPCSSCTLR.START}[m])$$
$$\text{Stop}_i = \sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.STOP}[n]) \vee \sum_m (\text{PECMP}[m] \wedge \text{TRCVIPCSSCTLR.STOP}[m])$$

R_{MVFYN} The following have no effect on the ViewInst start/stop function:

- Exceptional occurrences.
- Execution in Debug state.
- Execution in a Trace Prohibited region.
- A trace unit buffer overflow.

R_{GRSVY} When disabling the trace unit, the ViewInst start/stop function becomes static and retains its state until the trace unit is enabled again.

S_{XMMLP} For each of TRCVICTLR.SSSTATUS, TRCVISSCTLR, and TRCVIPCSSCTLR, if the register is implemented, it must be programmed with an initial state when the trace unit is programmed before a trace session.

R_{HYLDM} If an implementation makes speculation visible to the trace unit, the ViewInst start/stop function behaves as if no speculation has occurred. That is, when the instruction at a start or stop point is executed speculatively and is later canceled, the ViewInst start/stop function behaves as if the instruction at the start or stop point was not executed.

R_{BZSXR} When the trace unit becomes disabled and there are instructions at start points or stop points which are still speculative, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

- The ViewInst start/stop function behaves as if the instructions at the start points or stop points were incorrectly speculated. That is, the trace unit behaves as if those start points and stop points did not occur.
- The ViewInst start/stop function behaves as if the instructions at the start points or stop points were correctly speculated. That is, the trace unit updates the state of the ViewInst start/stop function as if those start points and stop points occurred.

R_{HMKSZ} When mis-speculation occurs and the PE returns to a point in execution before the trace unit was enabled, the ViewInst start/stop function reverts to the state it was in when the trace unit became enabled.

R_{PBFMY} When the state of the ViewInst start/stop function is changed by anything other than a direct write to TRCVICTLR, the PE considers the write to be an indirect write to TRCVICTLR.SSSTATUS.

D4.5.9.2.1 Instruction blocks

R_{PTTKL} When an instruction block that contains instructions at ViewInst start points and no instructions at ViewInst stop points is executed, the ViewInst start/stop function remains active for the entire instruction block.

R_{WJLMR} While the ViewInst start/stop function is active, when an instruction block is executed that contains at least one ViewInst stop address and no ViewInst start addresses, the ViewInst start/stop function remains active for the whole instruction block and becomes inactive for the next instruction block, unless the next instruction block contains a ViewInst start address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SMGJZ} When an instruction block that contains at least one instruction at a ViewInst start point and at least one instruction at a ViewInst stop point is executed, the ViewInst start/stop function obeys the order of the start and stop points in the block, with the following consequences:

- The ViewInst start/stop function is active for the whole of the instruction block.
- When the final instruction in the block at a ViewInst start or stop point is at a ViewInst start point, the ViewInst start/stop function is active for the next instruction block.
- When the final instruction in the block at a ViewInst start or stop point is at a ViewInst stop point, the ViewInst start/stop function is inactive for the next instruction block, unless the next block contains an instruction at a new ViewInst start point.

R_{SFXZB} The trace analyzer ensures that for all of the SACs selected for ViewInst start points or stop points, any SAC programmed with a lower address than another SAC is a lower-numbered SAC than the other SAC. That is, the SACs contain addresses in address order.

I_{RYPMM} While the SACs selected for ViewInst do not contain addresses in address order, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

R_{ZLDKC} The trace analyzer ensures that for all of the PE Comparator Inputs selected for ViewInst start points or stop points, any PE comparator programmed with a lower address than another PE comparator is a lower-numbered PE comparator than the other PE comparator. That is, the PE comparators contain addresses in address order.

I_{CKTWT} While the PE Comparator Inputs selected for ViewInst do not contain addresses in address order, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

R_{VFFWS} If more than one instruction Address Comparator is programmed with the same instruction address, then programming one or more of those comparators as start comparators, and one or more as stop comparators, results in the following CONSTRAINED UNPREDICTABLE behavior of the ViewInst start/stop function:

- The ViewInst start/stop function is either active or inactive for the instruction at that address.
- The ViewInst start/stop function is either active or inactive after that instruction.

D4.5.9.3 ViewInst include/exclude function filtering

I_{NDYFG} The ViewInst include/exclude function is useful if:

- Specific ranges of instructions are required to be included in the trace.
- Specific ranges of instructions are required to be excluded from the trace.
- A combination of including and excluding instruction ranges is required.

I_{LDDGL} The ViewInst include/exclude function is comprised of two functions:

Function	Meaning
ViewInst include function	Includes one or more instruction address ranges
ViewInst exclude function	Excludes one or more instruction address ranges

There are between zero and eight instruction Address Range Comparators available for the ViewInst include/exclude function. Some of these comparators can be selected for the ViewInst include function, and some for the ViewInst exclude function.

S_{VNWYT} For example, if all instructions in the address range from `0x00` to `0x2C` are required, but no other instructions are required, an Address Range Comparator can be selected for the ViewInst include function and be programmed with these two addresses. All instructions that are in this address range, including those at the start and end addresses, are traced.

I_{NKLJR} The ViewInst include/exclude function differs from the ViewInst start/stop function in the following ways:

- When the ViewInst start/stop function is used, the trace unit starts tracing on a specified start instruction address and stops tracing on a specified stop instruction address. However, if execution branches or jumps to an address between the start and stop points, without first accessing the instruction at the start address, then the instruction that it has branched or jumped to is not traced. Instructions in the start/stop range are only traced if the instruction at the start address is accessed, so that the trace unit is triggered to start tracing. When triggered, and as execution continues sequentially towards the stop address, all functions that the piece of code calls are traced.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the ViewInst include/exclude function is used, for example by programming an Address Range Comparator with an include address range, then if execution branches or jumps to any instruction address anywhere in that range, that instruction is always traced. This is true regardless of whether the instruction at the start address has been accessed or not.

In addition, unlike the ViewInst start/stop function, as program execution continues through the address range towards the end address, no functions that the piece of code calls are traced.

R_{SKZHH} The ViewInst include/exclude function part of the ViewInst function is expressed as the following equations:

$$I_i = \text{Include}_i \wedge \neg\text{Exclude}_i$$
$$\text{Include}_i = \sum_n (\text{ARC}[n] \wedge \text{TRCVIIECTLR.INCLUDE}[n]) \vee \prod_n \neg\text{TRCVIIECTLR.INCLUDE}[n]$$
$$\text{Exclude}_i = \sum_n (\text{ARC}[n] \wedge \text{TRCVIIECTLR.EXCLUDE}[n])$$

D4.5.9.3.1 Instruction blocks

R_{RNPWD} When an instruction in an instruction block is included to be traced by the ViewInst include/exclude function, the ViewInst include/exclude function includes all of the instruction block.

R_{PLCQJ} When an instruction block contains at least one instruction excluded by the ViewInst include/exclude function, and only when all the instructions in the instruction block are excluded, the ViewInst include/exclude function excludes the instruction block.

I_{LBNVM} If a block of instructions is not entirely covered by at least one individual ARC selected by TRCVIIECTLR.EXCLUDE, it is CONSTRAINED UNPREDICTABLE whether the block is excluded or not. This applies even if other ARCs selected by TRCVIIECTLR.EXCLUDE cover the rest of the block.

D4.5.9.4 Guidelines for interpreting the ViewInst function result

I_{TCGMV} The result of the ViewInst function is either:

Result	Meaning
High	Indicates that instructions being executed must be traced
Low	It is expected that instructions being executed are not traced

If it is expected that instructions being executed are not traced, then there are occasions when it is permitted to trace some of those instructions. This section provides guidelines for when it is permitted to trace instructions that ViewInst indicates are not traced.

D4.5.9.4.1 When ViewInst transitions from low to high

I_{GMYYC} If execution occurs while ViewInst is low, it is permitted for a trace unit to trace instructions in certain circumstances. For more information on ViewInst low, see Occasions when tracing instructions when ViewInst is low is permitted.

I_{RVCQT} Tracing of instructions is permitted while ViewInst is low, but if no instructions or Exceptional occurrences that occur are traced, then there is a discontinuity in the trace. When a discontinuity in the trace occurs, when ViewInst becomes high, a *Trace On element* must be generated.

I_{YXGLK} Any instructions that are executed while ViewInst is high must be traced.

D4.5.9.4.2 Occasions when tracing instructions when ViewInst is low is permitted

I_{VFYXG} ETE permits tracing of instructions when ViewInst is low, in the following scenarios:

- When the instruction that ViewInst indicates is not to be traced is in the same instruction block as an instruction that ViewInst indicates must be traced. This is because the only way to trace the instruction that must be traced is to trace the whole instruction block.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the instruction that ViewInst indicates is not to be traced is in an instruction block that precedes or follows an instruction block containing an instruction that ViewInst indicates must be traced.

An implementation always traces the instruction block that contains an instruction that must be traced. However, additional blocks of instructions might also be traced. This is more likely to occur when many instructions are executed in close proximity.

I_{CDLHB} Except for the scenarios that are mentioned, if the ViewInst function indicates that an instruction is not to be traced, then in general it is expected that it is not traced.

I_{FJMHL} In Figure D4-14, the instruction block 1 is in execution order before instruction block 2. The ViewInst calculation for the second block returns true, as indicated by the transition labeled (a). As ViewInst is true for this instruction block then all the instructions in this block must be traced, as indicated by the transition labeled (b). Instruction block 1 might be traced as it is in the same PE cycle as block 2, as indicated by the transition labeled (c).

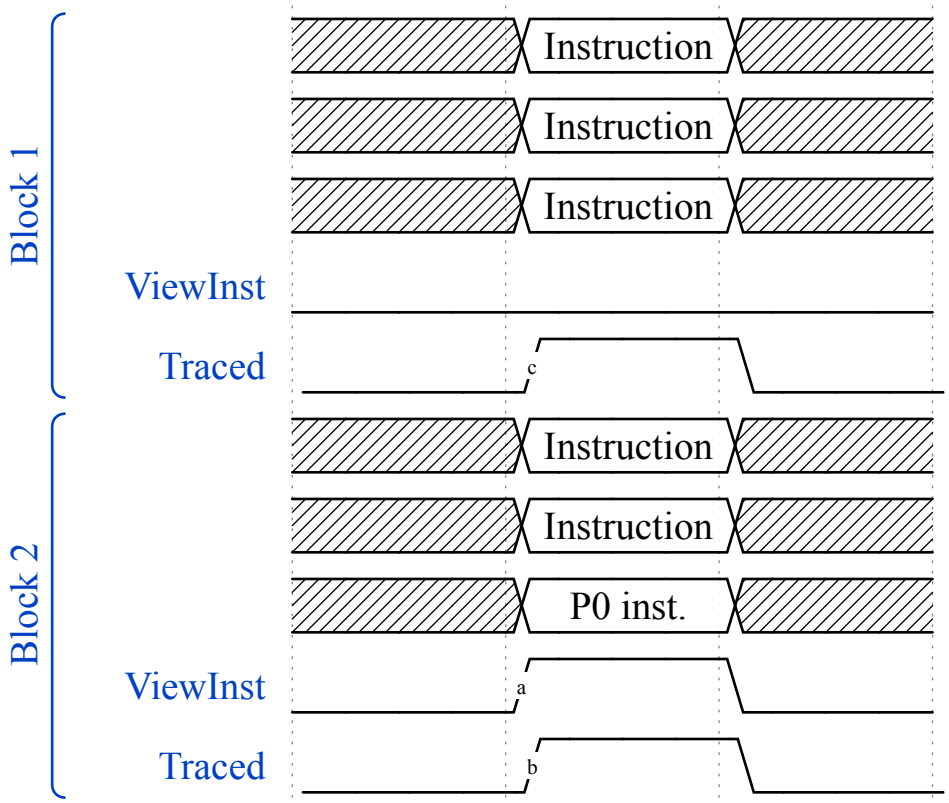

Figure D4-14 Example of close proximity

D4.5.9.5 Rules for tracing Exceptional occurrences

R_{CGVJD} When an Exceptional occurrence occurs, the Exceptional occurrence does not affect the comparators used by the ViewInst function, and none of the comparators used by the ViewInst function match.

I_{VFDMQ} The comparators used by the ViewInst function include the following:

- Single Address Comparators.
- Address Range Comparators.
- Context Identifier Comparators.
- Virtual Context Identifier Comparators.

I_{VFNZR} When an *Exception element* is traced, it might indicate execution of instructions up to a specified address. These instructions might have an impact on the comparators, but the Exceptional occurrence itself does not.

This means that when an Exceptional occurrence occurs, the ViewInst function does not indicate whether the Exceptional occurrence must be traced. However, it is useful to trace Exceptional occurrences, to determine why execution has departed from the normal program flow.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{BVGXZ} When an instruction executes or Exceptional occurrence occurs outside a Trace Prohibited region, the trace unit remembers whether the instruction or Exceptional occurrence was traced. The trace unit performs indirect writes to TRCRSR.TA to store this state. When an Exceptional occurrence occurs, the trace unit uses TRCRSR.TA to determine whether to trace the Exceptional occurrence.

R_{BFSWZ} When an instruction executes or Exceptional occurrence occurs outside a Trace Prohibited region and the instruction or Exceptional occurrence is traced, TRCRSR.TA is set to 1.

R_{MLTTK} When an instruction executes or Exceptional occurrence occurs outside a Trace Prohibited region and the instruction or Exceptional occurrence is not traced, TRCRSR.TA is set to 0.

R_{CJTJM} When an instruction or Exceptional occurrence is canceled, TRCRSR.TA is set to the value of TRCRSR.TA immediately before the canceled instruction or Exceptional occurrence.

R_{DPMBQ} When an Exceptional occurrence occurs, it is traced if all of the following are true:

- The PE is not in Debug state.
- TRCRSR.TA is 1.

R_{BJQDP} While any of the following are true, TRCRSR.TA is unchanged by any execution:

- The PE is in Debug state.
- The PE is in a Trace Prohibited region.

R_{QZPJD} When a trace unit buffer overflow occurs, the behavior of TRCRSR.TA is IMPLEMENTATION DEFINED and is one of the following:

- TRCRSR.TA is set to 0.
- TRCRSR.TA is set to the value of TRCRSR.TA for the most recent instruction or Exceptional occurrence before the trace unit buffer overflow occurred.

D4.5.9.6 Forced tracing of Exceptional occurrences

I_{MFQND} The trace unit can be programmed so that it always traces certain Exceptional occurrences, regardless of whether the instruction or Exceptional occurrence immediately before the Exceptional occurrence must be traced. This option is enabled by setting either or both:

- TRCVICTLR.TRCERR to 1. This forces the trace unit to trace System Error exceptions regardless of the value of ViewInst.
- TRCVICTLR.TRCRESET to 1. This forces the trace unit to trace PE Resets regardless of the value of ViewInst.

R_{SJXYS} While the PE is executing in a Trace Prohibited region, forced tracing of System Error exceptions is inactive.

R_{VLNMN} While the PE is not executing a Trace Prohibited region and forced tracing of System Error exceptions is enabled, forced tracing of System Error exceptions is active.

R_{LTLBB} While forced tracing of System Error exceptions is active, when a System Error exception occurs, the trace unit generates an *Exception element* indicating a System Error exception, regardless of the value of ViewInst.

R_{NCXJN} While the PE is executing in a Trace Prohibited region, forced tracing of PE Resets is inactive, regardless of whether the PE Reset causes the PE to leave a Trace Prohibited region or not.

R_{NQJNL} While the PE is not executing in a Trace Prohibited region, while forced tracing of PE Resets is enabled, forced tracing of PE Resets is active.

R_{GPKSH} While forced tracing of PE Resets is active, when a PE Reset occurs, the trace unit generates an *Exception element* indicating a PE Reset, regardless of the value of ViewInst.

R_{BBBBT} While tracing is inactive, before an *Exception element* is generated due to forced tracing of either a PE Reset of a System Error exception, the trace unit generates a *Trace On element* and then a *Target Address element*.

I_{LXLKS} When an *Exception element* is generated as a result of forced tracing, the *Trace On element* generated before the *Exception element* indicates that tracing becomes active, and the *Target Address element* indicates where tracing becomes active.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NZNDM} When a System Error exception occurs and TRCRSR.TA is 0 and the exception is traced because forced tracing of System Error exceptions is enabled, then it is IMPLEMENTATION DEFINED whether TRCRSR.TA is set to 1 or remains at 0.

R_{YFKGM} When a PE Reset occurs and TRCRSR.TA is 0 and the PE Reset is traced because forced tracing of PE Resets is enabled, then it is IMPLEMENTATION DEFINED whether TRCRSR.TA is set to 1 or remains at 0.

I_{GWHRM} In scenarios where a System Error exception occurs at approximately the same time as an exit from a Trace Prohibited region, after all execution inside the Trace Prohibited region and before any instruction execution outside the Trace Prohibited region, it is UNPREDICTABLE whether the System Error exception is considered to have occurred inside or outside the Trace Prohibited region. It is also UNPREDICTABLE whether the forced tracing of System Error exceptions is active for this exception.

These scenarios do not include scenarios where the System Error exception caused the exit from a Trace Prohibited region, because the System Error exception occurred inside the Trace Prohibited region.

I_{JGVSY} In scenarios where a System Error exception occurs at approximately the same time as an entry to a Trace Prohibited region, after all execution before the Trace Prohibited region and before any instruction execution inside the Trace Prohibited region, it is UNPREDICTABLE whether the System Error exception is considered to have occurred inside or outside the Trace Prohibited region. It is also UNPREDICTABLE whether the forced tracing of System Error exceptions is active for this exception.

These scenarios do not include scenarios where the System Error exception caused the entry to a Trace Prohibited region, because the System Error exception occurred outside the Trace Prohibited region.

R_{TSVKN} When a System Error exception occurs immediately after the PE exits a Trace Prohibited region and the System Error exception is traced, the preferred exception return address in the *Exception element* indicating the System Error exception does not include information about the Trace Prohibited region.

I_{PXJWM} In scenarios where a PE Reset occurs at approximately the same time as an exit from a Trace Prohibited region, after all execution inside the Trace Prohibited region and before any instruction execution outside the Trace Prohibited region, it is UNPREDICTABLE whether the PE Reset is considered to have occurred inside or outside the Trace Prohibited region. It is UNPREDICTABLE whether the forced tracing of PE Resets is active for this PE Reset.

These scenarios do not include scenarios where the PE Reset caused the exit from a Trace Prohibited region, because the PE Reset occurred inside the Trace Prohibited region.

I_{JKQHF} In scenarios where a PE Reset occurs at approximately the same time as an entry to a Trace Prohibited region, after all execution before the Trace Prohibited region and before any instruction execution inside the Trace Prohibited region, it is UNPREDICTABLE whether the PE Reset is considered to have occurred inside or outside the Trace Prohibited region. It is UNPREDICTABLE whether the forced tracing of PE Resets is active for this PE Reset.

These scenarios do not include scenarios where the PE Reset caused the entry to a Trace Prohibited region, because the PE Reset occurred outside the Trace Prohibited region.

R_{NRNFS} When a PE Reset occurs immediately after the PE exits a Trace Prohibited region and the PE Reset is traced, the preferred exception return address in the *Exception element* indicating the PE Reset does not include information about the Trace Prohibited region.

D4.5.10 Element Generation

D4.5.10.1 Trace Info element generation

R_{TFQRM} When a trace protocol synchronization request is serviced, the trace unit generates a *Trace Info element* to indicate where program execution trace analysis can start. A *Trace Info element* is not required to indicate where *Event element* and *Instrumentation element* analysis can start.

Note

There is no requirement to generate a new *Trace Info element* every time that ViewInst becomes active. This is because, despite the discontinuity in the trace that is caused by the filtering, the programming of the trace remains the same.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.10.2 Atom element

R_{XCJGD} When a *P0 instruction* is taken, the trace unit generates one of the following:

- An E *Atom element*.
- A *Source Address element*.

R_{SRYKV} When a *P0 instruction* is not taken, the trace unit generates one of the following:

- An N *Atom element*.
- Nothing.

R_{TZZRH} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, for all future not taken *P0 instructions* until the next taken *P0 instruction* or Exceptional occurrence, the trace unit does not generate an N *Atom element*.

R_{NZTGQ} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, when an Exceptional occurrence occurs before the next taken *P0 instruction*, the trace unit generates an *Exception element*.

R_{FWCQR} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, when no Exceptional occurrence occurs before the next taken *P0 instruction*, the trace unit generates a *Source Address element* for the next taken *P0 instruction*.

R_{NTMMM} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, and the *P0 instruction* is subsequently mispredicted, the trace unit generates a *Source Address element* and does not generate a *Mispredict element*.

R_{ZRYPK} The trace unit generates *Atom elements* in the program order in which they occur, and the trace protocol encode and decode process maintains this order.

I_{TGQNB} For conditional branch instructions, an E *Atom element* indicates that the instruction passed its condition code check, and an N *Atom element* indicates that the instruction failed its condition code check, although a trace unit might use a *Mispredict element* to modify the *Atom element*.

I_{BXBNQ} The trace unit might trace unconditional *P0 instructions* using an E *Atom element* or an N *Atom element*.

R_{RNYNV} When an unconditional *P0 instruction* is traced using an N *Atom element*, the trace unit generates either a *Mispredict element* or a *Cancel element* to correct the N *Atom element*.

R_{LMDZV} When an `ISB` instruction does not pass the condition code check, and the `ISB` instruction does not perform a Context synchronization event, the trace unit treats the `ISB` instruction as a not taken instruction.

R_{NZPLB} When an `ISB` instruction does not pass the condition code check, and the `ISB` instruction performs a Context synchronization event, the trace unit treats the `ISB` instruction as a taken instruction.

R_{CPFBS} When an `ISB` instruction passes the condition code check, the trace unit treats the `ISB` instruction as a taken instruction.

Note

For an `ISB` instruction, a trace analyzer must not infer the value of the PSTATE condition flags from an *Atom element*.

R_{QBGXJ} It is IMPLEMENTATION DEFINED whether the trace unit classifies `WFI`, `WFIT`, `WFE`, and `WFET` instructions as *P0 instructions*. When `WFI`, `WFIT`, `WFE`, and `WFET` are classified as *P0 instructions*, execution of these instructions generates an *Atom element*. See Low-power state and TRCIDR2.WFXMODE.

R_{HJHHV} When `WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as *P0 instructions* and a conditional `WFE` or `WFI` instruction is executed, if the instruction passes its condition code check then an E *Atom element* is generated.

R_{BMXDT} When `WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as *P0 instructions* and a conditional `WFE` or `WFI` instruction is executed, if the instruction fails its condition code check then either an E *Atom element* or an N *Atom element* is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For a `WFI`, `WFIT`, `WFE`, or `WFET` instruction which is classified as a *P0 instruction*, a trace analyzer must not infer the value of the PSTATE condition flags from an E *Atom element*.

I_{PZRCT} *P0 instructions* that fail or are predicted to fail their condition code check either generate an Undefined Instruction exception or are executed as a NOP, if the instruction is also UNDEFINED.

R_{YCRVD} When a *P0 instruction* fails or is predicted to fail its condition code check, and the *P0 instruction* is executed as a NOP, the trace unit generates an N *Atom element* for the *P0 instruction*.

R_{TSQGH} When a *P0 instruction* fails or is predicted to fail its condition code check, and the *P0 instruction* generates an Undefined Instruction exception, the trace unit does not generate an *Atom element* for the instruction and generates an *Exception element* instead. The preferred exception return address for the generated *Exception element* is the UNDEFINED instruction, which indicates that the instruction did not execute.

R_{NQPPX} The trace unit generates all *Atom elements* speculatively, and explicitly resolves or cancels each *Atom element* by generating *Commit elements* or *Cancel elements*.

I_{NGJYB} A trace analyzer can infer execution from an *Atom element*, but only after the *Atom element* has been resolved by a *Commit element*.

S_{MFDNZ} For taken direct *P0 instructions*, a trace analyzer must infer the target address and instruction set of the instruction from the opcode in the program image.

S_{TJSRY} If a taken direct *P0 instruction* is from a branch broadcasting region, the trace analyzer does not need to infer the target address and instruction set because this is explicitly traced using a *Target Address element*.

D4.5.10.3 Exception element

R_{QHRVX} When an Exceptional occurrence occurs, if the Exceptional occurrence is required to be traced, the trace unit generates an *Exception element*.

R_{LYYMG} The trace unit generates *Exception elements* in program order relative to other *P0 elements*.

I_{PCMYT} To be consistent with the rules for generating *Target Address elements*, under the following scenarios the trace unit must generate a *Target Address element* before an *Exception element*, unless the *Target Address element* would be removed due to a return stack match:

- The Exceptional occurrence is taken from the target of a taken indirect *P0 instruction*.
- The Exceptional occurrence is taken from the target of a taken direct *P0 instruction* in a branch broadcasting region.
- The Exceptional occurrence is taken from the target of another *Exception element*.

R_{XGXKK} When an Exceptional occurrence occurs, if the *Context information* changes at the target of the *P0 element* preceding the Exceptional occurrence, then the trace unit generates a *Context element* before the *Exception element*. The *Context element* provides *Context information* about the address and context where the Exceptional occurrence was taken from.

I_{CMRCN} An invalid address is one where bits [63:P] are not all zeros or all ones, where P is defined as one of the following:

- When FEAT_LVA3 is implemented, 56.
- When FEAT_LVA is implemented, 52.
- Otherwise, 48.

R_{RJCBT} When the PE attempts to execute an instruction at an invalid address and the trace unit generates an *Exception element*, the preferred exception return address in the *Exception element* indicates one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

I_{GYJKB} When a branch to an invalid address occurs and an *Exception element* is generated for an exception taken from that invalid address, Arm recommends that the preferred exception return address in the *Exception element* is the same as the invalid address indicated by the *Target Address* element for the branch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.10.4 Source address element

R_{DCHPQ} When a *P0 instruction* which must be traced is not taken and the trace unit does not generate an N *Atom element*, then when a subsequent *P0 instruction* is taken, the trace unit generates a *Source Address element*.

I_{KDTRV} A trace unit can generate a *Source Address element* to imply that at least one instruction has been executed, including a taken *P0 instruction*.

R_{WVBQW} When the trace unit generates a *Source Address element* to imply that a taken *P0 instruction* has been executed, the address associated with the *Source Address element* is the virtual address of the taken *P0 instruction*.

D4.5.10.5 Q element

R_{FZTZP} A trace unit can generate a *Q element* to imply that at least one instruction has been executed, possibly including *P0 instructions*.

R_{WHLPS} When a *Q element* is generated, the trace unit generates a *Target Address element* that indicates where execution is to continue after all the instructions that are implied by the *Q element* have been executed.

R_{MNWCK} When a *Q element* is generated and the last instruction implied by the *Q element* is a *P0 instruction*, the trace unit generates a *Target Address element* that indicates the target of the *P0 instruction*.

R_{BJLRS} When a *Q element* is generated and the last instruction implied by the *Q element* is not a *P0 instruction*, the trace unit generates a *Target Address element* that indicates the instruction address immediately following the last instruction that is implied by the *Q element*.

R_{FPHFM} When the PE leaves a region where *Q elements* are permitted, either by a *P0 instruction* or by sequential execution out of the region, and a *Q element* implies the execution of the last instruction in the region, the *Q element* does not imply any more instructions after the last instruction in the region.

R_{DTWYZ} When the PE enters a region where *Q elements* are permitted, either by a *P0 instruction* or by an Exceptional occurrence, the trace unit traces the *P0 instruction* or Exceptional occurrence using elements other than *Q elements*.

Note

Although the trace unit does not trace with *Q elements* a *P0 instruction* or Exceptional occurrence that causes the PE to enter a region where *Q elements* are permitted, any subsequent instructions in the region where *Q elements* are permitted might be traced using *Q elements*.

I_{YRZWY} When the PE enters by sequential execution a region where *Q elements* are permitted, any instructions that are executed since the last *P0 element* outside the permitted region might be traced using a *Q element*. These instructions can always be inferred unambiguously from the *Q element*.

R_{SJSGX} When the PE enters by sequential execution a region where *Q elements* are permitted, and *P0 instructions* executed since the last *P0 element* outside the permitted region are traced by a *Q element*, the *Q element* does not indicate execution of any *P0 instructions* outside the permitted region.

D4.5.10.6 Event element

R_{WJPTB} The trace unit generates *Event elements* independently of ViewInst.

R_{KKLYB} While TRCEVENTCTL1R.INSTEN<n> is 1 and the resource event selected by TRCEVENTCTL0R.EVENT<n> is active, while trace generation is operative, the trace unit generates an *Event element* <n> on each PE clock cycle.

R_{SPYTT} When an *Event element* is generated between two *P0 elements* or at the same time as a *P0 element* that follows another, the trace unit inserts the *Event element* after the first *P0 element* but before the *P0 element* that is an IMPLEMENTATION DEFINED number of *P0 elements* after the first *P0 element*.

I_{XLJKP} Arm recommends that the IMPLEMENTATION DEFINED number of *P0 elements* is less than or equal to the number of *P0 elements* the PE can process simultaneously.

R_{SHYMY} While trace generation is inoperative due to a trace unit buffer overflow, when a programmed *ETEEvent* <n> occurs, the trace unit generates at least one *Event element* <n> before it generates the *Overflow element* corresponding to the trace unit buffer overflow.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.10.7 Cancel element generation

R_{HYCXR} When one or more *P0 elements* are canceled, the trace unit generates a *Cancel element*.

R_{GKDYR} The trace unit generates *Cancel elements* in execution order relative to *P0 elements*.

R_{KDTVX} When a *Cancel element* causes execution to return to a point in the program flow that is not adjacent to a *P0 instruction*, the trace unit generates an *Exception element* that indicates which instructions were executed up to that point in the program flow before it generates any *P0 elements*.

D4.5.10.8 Commit element generation

R_{QNLYJ} When one or more traced *P0 elements* are resolved for execution, the trace unit generates a *Commit element*.

I_{MDSCN} An *Atom element* might be corrected using a *Mispredict element* after it has been resolved.

R_{MNXTW} The trace unit never generates more speculative *P0 elements* than the maximum speculation depth of the trace unit.

R_{XXFBC} When trace generation becomes inoperative due to the trace unit being disabled, the trace unit outputs any *Commit elements* which have not been output.

I_{BNLZT} If cycle counting is enabled some *Commit elements* have *Cycle Count elements* associated with them, that provide counts of processor clock cycles. The cycle count values given in *Cycle Count elements* can be used to obtain a cumulative count.

R_{ZHWDD} *Commit elements* with associated *Cycle Count elements* cannot be merged with later *Commit elements*.

I_{PSFTD} For more information, see Cycle Count element.

D4.5.10.9 Context element

R_{BDFDQ} The trace unit generates a *Context element* in the following situations:

- While tracing is active, when any of the *Context information* changes, before any *P0 element* which indicates execution from the new context.
- After a *Trace Info element* is generated due to a non-periodic trace protocol synchronization request, and before any *P0 element*.
- After a *Trace Info element* is generated due to a periodic trace protocol synchronization request.
- When mis-speculation results in an incorrect *Context element* being output, before any *P0 element* which indicates execution from the correct context.

R_{JNXJT} While Virtual context identifier tracing is enabled and TRFCR_EL2.CX disallows the tracing of the Virtual context identifier, when the trace unit generates a *Context element*, the Virtual context identifier in the *Context element* has the value `0x0`.

I_{TBJPL} A *Context element* might also be output at other points, which might include after all Context synchronization events, or at any other point at which the *Context information* changes.

R_{MKKZN} If the highest implemented Exception level is using AArch64, the Context identifier value is the value of CONTEXTIDR_EL1.

I_{WXVHT} Some of the *Context information* might change at points other than at Context synchronization events. These changes occur when system instructions are used to change a piece of *Context information*, including:

- Writes to the current CONTEXTIDR_EL1.
- Writes to the CONTEXTIDR_EL2.
- Changes from a higher Security state to a lower Security state without using an exception return.
- Changes in Exception level other than due to an exception or an exception return.

R_{HXNXF} When a System instruction writes to a System register and the *Context information* changes, the trace unit generates a *Context element* containing the new context value, after the *P0 element* prior to the System instruction but before the *P0 element* following a Context synchronization event after the System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If the *Context element* is output before the first *P0 element* after the System instruction, this might imply that some instructions before the System instruction were executed with the new context. This is acceptable because the code which changes the context is usually executed in a state where it does not matter whether the old or new context values are used.

I_{WTSTB} If the PE takes an exception after performing a write to a System register that changes the context, but a *P0 element* has not been generated since the write, then a *Context element* indicating the new context is not required to be output before the *Exception element*. This is because no instructions or Exceptional occurrences are indicated to have been executed from the new context. A *Context element* indicating the new context must be generated after the *Exception element* if the Exceptional occurrence is a Context synchronization event. If the Exceptional occurrence changes the context, then the *Context element* must indicate the new context. This might happen if, for example, the Security state changes.

R_{HTYGF} When a PE reset occurs, until the relevant PE registers are updated, the trace unit traces the Context identifier and Virtual context identifier as zero.

I_{KTPVJ} A trace unit is not required to generate a *Context element* if tracing becomes inactive before any instructions are executed in the new context.

I_{QWSVJ} Additional *Context elements* might be output by a trace unit in some scenarios, but these must only be output where they do no affect the analysis of the trace element stream. Such a scenario might include when a change in the *Context information* is incorrectly speculated and a subsequent *Context element* corrects the value of a previous incorrect *Context element*. Arm recommends that the generation of additional unnecessary *Context elements* is minimized to ensure trace bandwidth is minimized.

D4.5.10.10 Target Address element

R_{HLRZJ} When the trace analyzer cannot infer the address or instruction set from the trace, the trace unit generates a *Target Address element*.

I_{FRTGM} Occasions when the trace analyzer might not be able to infer the address or instruction set from previous trace include:

- At the target of an indirect *P0 instruction* which is taken.
- At the target of a direct *P0 instruction* which is taken in a branch broadcasting region, see TRCBBCTLR for more information.
- At the target of an Exceptional occurrence.
- When mis-speculation occurs and the address cannot be inferred.
- After a *Q element* is generated.

R_{ZRYSN} When the trace analyzer cannot infer the address or instruction set from the trace, the trace unit generates the resulting *Target Address element* before the next *P0 element*, unless any of the following are true:

- The *Target Address element* can be omitted because of a return stack match.
- Tracing is inactive at the target of the *P0 instruction* or Exceptional occurrence.

R_{RGPTK} When non-periodic trace protocol synchronization occurs, the trace unit generates a *Target Address element* after the *Trace Info element* and *Trace On element* corresponding to the non-periodic trace protocol synchronization, and before the next *P0 element* is generated.

R_{MDTYL} When periodic trace protocol synchronization occurs, after the corresponding *Trace Info element* has been generated, the trace unit generates a *Target Address element* containing the address of the target of the most recent *P0 element* before the *Target Address element*.

I_{YMLXQ} When non-periodic trace protocol synchronization occurs, the *Target Address element* does not need to indicate the target of the most recent *P0 element*, since tracing might not become active at the target of a *P0 element*.

I_{GBMWG} When periodic trace protocol synchronization occurs, the *Target Address element* needs to indicate the target of the most recent *P0 element*, since tracing is continuing from that *P0 element*. Furthermore, the *Target Address element* might indicate the target of a *P0 element* from before the *Trace Info element*.

R_{QCSJJ} When a *Trace On element* is generated, the trace unit generates a *Target Address element* before the next *P0 element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{DTQDH} Typically, a *Target Address element* is required after an *Exception element* to indicate the target of the Exceptional occurrence, since a trace analyzer is not usually able to infer the target of an Exceptional occurrence.

I_{YHQGL} In some scenarios, an *Exception element* might be generated in the trace where the Exceptional occurrence target address is the next sequential instruction from the last instruction before the Exceptional occurrence. This behavior depends on many factors and might only occur for IMPLEMENTATION DEFINED Exceptional occurrences. If an Exceptional occurrence is taken to the next sequential instruction, the trace unit is not required to output a *Target Address element* indicating the target address of the Exceptional occurrence because this can be inferred from the previous execution.

I_{GVZJZ} A trace analyzer needs both a *Target Address element* and a *Context element* before it can determine the instruction set in use, because the *Target Address element* provides the instruction set and the *Context element* provides information on which Execution state the PE is in.

R_{RHDMW} When a change of instruction set occurs that switches between AArch32 state and AArch64 state, the trace unit generates a *Context element* indicating the new state.

I_{KZXQW} An invalid address is one where bits [63:P] are not all zeros or all ones, where P is defined as one of the following:

- When FEAT_LVA3 is implemented, 56.
- When FEAT_LVA is implemented, 52.
- Otherwise, 48.

R_{VWVWR} When the PE attempts to execute an instruction at an invalid address and the trace unit generates a *Target Address element*, the *Target Address element* indicates one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

I_{YJYFM} When a branch to an invalid address occurs and an *Exception element* is generated for an exception taken from that invalid address, Arm recommends that the preferred exception return address in the *Exception element* is the same as the invalid address indicated by the *Target Address* element for the branch.

R_{SBCPK} While tagged addresses are in use, the virtual address in the trace element stream does not include the tag and is the PC value, that is, depending on the state of the PE at the address, bits[63:56] are one of the following:

- The sign-extension of bit[55].
- Zero.

I_{YGGGK} The Translation Control Registers, TCR_ELx, contain the TBI field for controlling whether to ignore the top byte of an address. If the current TBI field is changed from 0 to 1, and before the next Context synchronization event the PE takes an exception because of an invalid top address byte, the branch target address to the invalid address or the preferred exception return address of the *Exception element* might not contain the full invalid address and might contain the address with the top byte masked. Furthermore, the branch target address might be the invalid address and therefore might be different from the preferred exception return address. Trace analysis tools must be aware that if a branch target address is substantially different from a preferred exception return address which follows, then there might have been a change in the TBI field which caused this large change in address.

R_{HHKGB} When a pointer authentication check fails and an exception is taken from the resulting invalid address, the preferred exception return address is one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

I_{CRSTX} Arm recommends that when a pointer authentication check fails and an exception is taken from the resulting invalid address, the preferred exception return address is the full 64-bit invalid address.

R_{FSWRC} The bottom bits of an address are ignored, in the following way:

- Bits[1:0] of addresses that are used in A64 or A32 instructions are always traced as zero.
- Bit[0] of addresses that are used in T32 instructions is always traced as zero.

I_{MDZJL} Additional *Target Address elements* might be output by a trace unit in some scenarios, but these must only be output where they do not affect the analysis of the trace element stream. These scenarios include, but are not limited to:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When an instruction address is incorrectly speculated, and a subsequent *Target Address element* corrects the value of a previous incorrect *Target Address element*.
- When an instruction address can be inferred by the trace analyzer, for example for the target of a direct *P0 instruction*, but a *Target Address element* is output anyway with the same address.

Arm recommends that the generation of additional unnecessary *Target Address elements* is minimized to ensure trace bandwidth is minimized.

D4.5.10.11 Mispredict element

R_{YJCNT} When the status of the last non-canceled *Atom element* has been changed by the PE, the trace unit generates a *Mispredict element*.

R_{SCKBZ} The trace unit only generates a *Mispredict element* to change the status of an *Atom element*.

I_{XYDNP} A trace unit might generate multiple *Mispredict elements* for the same *Atom element*. A trace analyzer must use each *Mispredict element* to determine the final status of the *Atom element*. For example, if an E *Atom element* has two *Mispredict elements*, the first *Mispredict element* indicates the *Atom element* is an N *Atom element* and the second *Mispredict element* indicates it is an E *Atom element*.

I_{HVWCN} If a PE mispredicts only the target address of a *P0 element*, then it does not generate a *Mispredict element*.

The trace unit uses a *Target Address element* to correct the mispredicted target address. When analyzing a *Mispredict element*, any *Target Address elements* between the mispredicted *Atom element* and the *Mispredict element* must be discarded.

D4.5.10.12 Overflow element

R_{HRYKY} When a trace unit buffer overflow occurs, after all trace elements that were generated prior to the trace unit buffer overflow are output, the trace unit outputs an *Overflow element*.

R_{RPSPH} When a trace unit buffer overflow occurs, and the trace unit is disabled after recovering from the trace unit buffer overflow, the trace unit outputs the corresponding *Overflow element* before the trace unit becomes idle.

D4.5.10.13 Timestamp element

R_{YYWTR} While TRCCONFIGR.TS is 1 and any of the following occur, a timestamp request occurs:

- The timestamp resource event occurs, as controlled by TRCTSCTLR.
- The trace unit generates a *Trace Info element*.
- The trace unit recovers from a trace unit buffer overflow.
- When not in a Trace Prohibited region and a Context synchronization event is caused by any of the following:
  - The PE takes an exception.
  - The PE returns from an exception.
  - An `ISB` instruction is executed.
- A trace unit flush is requested.

R_{CKNFV} While TRCCONFIGR.TS is 1 and when not in a Trace Prohibited region, a timestamp request might occur when any of the following occur but do not cause a Context synchronization event:

- The PE takes an exception.
- The PE returns from an exception.
- An `ISB` instruction is executed.

R_{NGXNQ} The state of the ViewInst function does not affect whether a timestamp request occurs.

R_{HZSYP} When a timestamp request occurs and ViewInst is inactive, the timestamp request is permitted to be delayed until after the first of the following occurs:

- A *P0 element* is generated.
- An *Event element* is generated.
- An *Instrumentation element* is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{WFXVV} There is no requirement for a *Timestamp element* to be generated in the trace element stream on each occasion that ViewInst becomes active.

R_{RMSVV} When a timestamp request occurs and is not ignored, the trace unit generates a *Timestamp element*.

R_{DWCYP} When a timestamp request occurs but the trace unit does not have the capacity to generate the *Timestamp element* immediately, then the generation of the *Timestamp element* is delayed until there is capacity.

I_{TQDHQ} A trace unit might not have the capacity to generate a *Timestamp element* for multiple reasons, including avoiding a trace unit buffer overflow. A delayed *Timestamp element* means that a timestamp value might not be the exact time of the incident that resulted in the timestamp request. A timestamp is only a time indicator inserted in the trace element stream somewhere near the time of the request.

R_{XMJGY} When a timestamp request occurs while in a Trace Prohibited region, then the generation of the *Timestamp element* is delayed until the PE leaves the Trace Prohibited region.

R_{CWHHW} When the first timestamp request occurs after trace generation becomes operative, the trace unit delays generation of the corresponding *Timestamp element* until after the trace unit has generated either a *P0 element*, an *Event element*, or an *Instrumentation element*.

I_{SDPZZ} This is so that the timestamp value can correspond to the most recent of these elements.

R_{SZNMB} A timestamp request is permitted to be ignored if a previous timestamp request has not yet generated a *Timestamp element*, due to a delay in the generation.

R_{DWFTT} A trace unit might ignore the second request of two successive timestamp requests if all of the following are true:

- The second request is not caused as a result of a trace protocol synchronization request.
- No *P0 elements*, *Event elements*, or *Instrumentation elements* have been generated between the two requests.
- The timestamp value would be the same in both *Timestamp elements*.

R_{ZMQLT} While TRCCONFIGR.CCI is 1, each *Timestamp element* contains a cycle count that indicates the number of cycles between the element with which the Timestamp is associated and the most recent *Cycle Count element* before that element.

R_{MWJHD} The cycle count associated with a *Timestamp element* is different from the *Cycle Count element* in the following ways:

- The cycle count does not affect the cumulative cycle count.
- The cycle count value can be zero.

I_{HTGCM} When the cycle count associated with a *Timestamp element* is zero, this indicates that no cycles passed between the previous *Cycle Count element* and the element with which the *Timestamp element* is associated.

I_{TYVDN} The cycle count associated with a *Timestamp element* is not a *Cycle Count element*, and therefore has no effect on the cycle counter.

R_{JNSWW} When the trace unit is first enabled, while cycle counting is enabled, when a *Timestamp element* is generated before any *Cycle Count elements*, the *Timestamp element* reports the cycle count as UNKNOWN.

R_{NPYKS} When a *Timestamp element* is generated and the cycle counter has exceeded the maximum supported value, the *Timestamp element* reports the cycle count as UNKNOWN.

D4.5.10.14 Trace On element

R_{LMLSN} When an instruction block is traced immediately after an instruction block was not traced or a trace unit buffer overflow occurred, the trace unit generates a Trace On element.

I_{GPQZW} When an *Exception element* indicating a PE reset is traced, the preferred exception return address is UNKNOWN. Any instructions since the most recent unresolved *P0 element* are not traced. If ViewInst was active for these instructions, this is not considered a gap in the trace element stream and a *Trace On element* is not required.

In some scenarios where mis-speculation occurs or instructions are canceled, after *Cancel elements* have been processed there might be *Trace On elements* in the trace element stream even though no trace discontinuity occurred in the architecturally-executed instruction trace. This typically only occurs when the trace is filtered using the ViewInst function, which causes the *Trace On element* to be inserted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{MHFJB} Trace analyzers must be aware that these additional *Trace On elements* might be traced.

D4.5.10.15 Cycle Count element

R_{TYNZR} The cycle counter has an IMPLEMENTATION DEFINED size of between 12 and 20 bits, as indicated by TRCIDR2.CCSIZE. The cycle counter therefore supports values from 1 to 2^{20}-1.

R_{GWQGS} While TRCCONFIGR.CCI is 1 and the cycle count is equal to or greater than the value of TRCCCCTLR.THRESHOLD, when a Commit element is generated, a *Cycle Count element* request occurs.

R_{KJXDK} While TRCCCCTLR.THRESHOLD is programmed with a value less than TRCIDR3.CCITMIN, the generation of *Cycle Count elements* is CONSTRAINED UNPREDICTABLE.

R_{GDTBW} When a request for a *Cycle Count element* occurs, one of the following occurs:

- The trace unit generates a *Cycle Count element* immediately and before any future *Commit element*.
- The trace unit delays generation of the *Cycle Count element* until after one or more further *Commit elements* have been generated.

I_{TCFRL} Arm recommends that when a request for a Cycle Count element occurs, the *Cycle Count element* is generated immediately, and that *Cycle Count element* generation is only delayed in rare and non-repetitive circumstances.

R_{BMVKB} When a *Cycle Count element* is generated, the *Cycle Count element* contains the value of the cycle counter at the time the most recent *Commit element* was generated, and the cycle counter is reset to the number of cycles since the most recent *Commit element* was generated.

R_{PFZXK} A value of 0 indicates that the cycle count value is UNKNOWN.

R_{YVWJW} When the cycle counter exceeds the maximum supported value, the cycle count value is UNKNOWN.

R_{YFMWB} When the trace unit becomes enabled, an UNKNOWN cycle count value occurs for the first *Cycle Count element* generated.

R_{HQKWH} When a trace unit buffer overflow occurs, an UNKNOWN cycle count value occurs for the first *Cycle Count element* generated.

I_{PDBDY} The first *Cycle Count element* after the PE clock has been restarted should have an UNKNOWN cycle count.

D4.5.10.16 Discard element

R_{WDWGV} When trace generation becomes inoperative after *P0 elements* have been generated but the trace unit is unable to output the resolution of those *P0 elements*, the trace unit generates a *Discard element*.

R_{FHQDX} When trace generation becomes inoperative due to the trace unit becoming disabled, and a *Discard element* is generated, the trace unit outputs the *Discard element* after all other elements.

R_{NSMJF} When a PE reset occurs after *P0 elements* have been generated but the trace unit is unable to output the resolution of those *P0 elements*, the trace unit generates a *Discard element*.

I_{SKJSP} A trace unit might not generate a *Discard element* if no *P0 elements* are speculative.

I_{TGXKV} A trace unit might generate a *Discard element* even if no *P0 elements* are speculative.

R_{CTYFK} When a *Discard element* is generated, all uncommitted *P0 elements* are discarded, that is, the current speculation depth is set to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.10.17 Instrumentation element

R_{STQSN} The Instrumentation element for a first `TRCIT` instruction is generated before the Instrumentation element for any `TRCIT` instruction in program order after that first `TRCIT` instruction.

R_{WYYPN} The Instrumentation element contains the following information:

- The doubleword contained in the `Xt` register parameter of the `TRCIT` instruction.
- The Exception level at which the `TRCIT` instruction was executed.

I_{XWZNW} If Instrumentation tracing is prohibited and tracing is not prohibited, then Arm recommends that the behavior of the `TRCIT` instruction is unaffected by the state of the trace unit.

I_{KGBZX} The `TRCIT` instruction is added to the list of instructions that support concurrent modification and execution, to allow live patching of the `TRCIT` instruction. A `TRCIT` instruction is only permitted to be modified to be a `NOP` instruction or another encoding of the `TRCIT` instruction. Only a `NOP` instruction or a `TRCIT` instruction is permitted to be modified to be a `TRCIT` instruction.

R_{SRSGF} The architecture guarantees that for `NOP` and `TRCIT` instructions, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

R_{LGWQF} Each Instrumentation element generates an Instrumentation packet.

I_{BXHZM} When non-periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet before any Instrumentation packet. See R_{TTLJC}.

I_{YWJZW} There are no requirements on the order between Instrumentation packets and other ETE packets, except for the requirements around an Alignment Synchronization packet and the requirements for ordering Timestamp packets and Timestamp Marker packets.

D4.5.11 Trace unit features

I_{BFHRC} The architecture defines a number of optional and mandatory features that are provided to modify the trace element stream to provide additional information to aid debugging. These features include the following:

- *Q element* regions.
- Branch broadcasting.
- Context identifier tracing.
- Cycle counting.
- Event trace.
- No overflow.
- PE Stalling and overflow avoidance.
- Timestamping.
- Virtual context identifier tracing.

For the optional features, the inclusion of these optional features is indicated in TRCIDR0-TRCIDR13.

D4.5.11.1 Q regions

I_{XFPKH} The support for *Q element* is OPTIONAL, as indicated by TRCIDR0.QSUPP.

I_{FSXRY} The use of *Q elements* must be explicitly enabled if the trace unit is to use them.

I_{CGQZJ} While *Q elements* are enabled, the trace element stream might not contain enough information to determine the complete program flow, because some changes in flow might not be explicitly indicated.

I_{QLVSG} Arm recommends that *Q elements* are only used in cases where generating the full ETE instruction trace element stream might cause the performance of the PE being traced to degrade significantly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{TNPWC} The use of *Q elements* degrades the information that can be extracted from the trace element stream. Arm recommends that *Q element* filtering, as indicated by TRCIDR0.QFILT, is also implemented.

I_{CBKXZ} If TRCQCTLR is implemented, the trace unit supports the ability to control when *Q elements* are permitted in the trace element stream using ARCs. The *Q element* filtering operates in either Exclude mode, or Include mode, selected by TRCQCTLR.MODE.

R_{DSFJZ} If *Q elements* are enabled and *Q element* filtering is in Include mode, the ARCs selected by TRCQCTLR.RANGE define where *Q elements* are permitted.

R_{WXRDY} If *Q elements* are enabled and *Q element* filtering is in Exclude mode, the ARCs selected by TRCQCTLR.RANGE define where *Q elements* are not permitted.

R_{RBPJF} When an instruction block contains at least one instruction where *Q elements* are permitted, the entire instruction block is permitted to generate *Q elements*.

R_{NQSLS} The following equation is calculated for each instruction block and defines when *Q elements* are permitted.

$$\text{Q_PERMITTED}_i = \begin{cases} \sum_n \text{ARC}_i[n] \wedge \text{TRCQCTLR.RANGE}[n] & \text{Include mode} \\ \neg \sum_n \text{ARC}_i[n] \wedge \text{TRCQCTLR.RANGE}[n] & \text{Exclude mode} \end{cases}$$

R_{SGFHP} While TRCCONFIGR.QE indicates that *Q elements* are disabled, the trace unit does not generate any *Q elements*.

R_{CGHSK} While TRCCONFIGR.QE indicates that *Q elements* are disabled, the trace unit is able to generate all of the elements required to trace the instruction sequence.

D4.5.11.2 Branch broadcasting

I_{WLPLZ} The branch broadcasting feature forces the trace unit to explicitly trace the target addresses of taken direct *P0 instructions*.
The target addresses are traced using *Target Address elements* in the instruction trace stream.

I_{DCYQT} Branch broadcasting is enabled by performing both of the following actions:

- Setting TRCCONFIGR.BB to 1.
- Programming TRCBBCTLR to specify how branch broadcasting behaves. TRCBBCTLR selects Address Range Comparators to define when branch broadcasting is active, and selects the operating mode of branch broadcasting:
  - — Branch broadcasting is active for all instructions inside the selected ranges. This is known as Include mode.
  - — Branch broadcasting is active for all instructions outside the selected ranges. This is known as Exclude mode.

R_{MHYFV} When a direct *P0 instruction* for which branch broadcasting is active is taken, the trace unit generates a *Target Address element* to explicitly trace the target of the instruction, regardless of whether the *P0 instruction* is mispredicted.

R_{VQTVR} While branch broadcasting is enabled, while the return stack is enabled, the trace unit prioritizes branch broadcasting over the return stack, that is, the return stack does not match on the target of any instruction for which branch broadcasting is active.

R_{XSVSX} If TRCBBCTLR is not implemented, while branch broadcasting is enabled, branch broadcasting is active for all instructions.

D4.5.11.3 Context identifier tracing

I_{RJJZW} The trace unit can be programmed to include information about the current execution context of the program being executed on the PE, including:

- The current process identifier, stored in CONTEXTIDR_EL1. This is known as the *Context identifier*.
- The current VMID, stored in CONTEXTIDR_EL2. This is known as the *Virtual context identifier*.

R_{GVMFG} The trace unit supports tracing of the Context identifier, with TRCIDR2.CIDSIZE indicating a 32-bit Context identifier size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.11.4 Cycle counting

I_{MKQVC} The use of the cycle counting feature introduces *Cycle Count elements* into the trace element stream to indicate the passage of PE clock cycles.

I_{BMCMB} Counting the number of clock cycles the PE uses to perform a certain function can be useful as a way of measuring program performance, or for profiling the PE.

R_{JMLLY} While cycle counting is enabled, the trace unit outputs Cycle Count packets that contain processor clock cycle count values.

I_{TVCCV} *Cycle Count elements* are associated with *Commit elements*, so that when a *Commit element* is generated, a *Cycle Count element* might also be generated. Whether a *Cycle Count element* is generated when a *Commit element* is generated depends on what cycle count threshold has been specified when programming TRCCCCTLR.THRESHOLD.

When a *Commit element* is generated and the cycle count value is equal to or more than the threshold value, then a *Cycle Count element* is generated and a Cycle Count packet is output. The cycle count value that is contained in that packet is associated with the *Commit element* that triggered it.

R_{BHYWB} While cycle counting is enabled, and when a *Commit element* is generated and the cycle count value is greater than or equal to the threshold value that is programmed in TRCCCCTLR.THRESHOLD, the trace unit generates a *Cycle Count element*.

I_{LFLCZ} Also, because cycle counting is associated with *Commit elements*, a Cycle Count packet might imply the generation of *Commit elements*, and so in addition to the cycle count value, some Cycle Count packets also contain a value for the number of *Commit elements* that the trace unit has generated.

I_{MWQNZ} The value of cycle count that is given in a new Cycle Count packet indicates the number of processor clock cycles between the new *Commit element* that the packet is associated with, and the most recent *Commit element* prior to the new *Commit element* that had a *Cycle Count element* associated with it.

This means that if there is a requirement for a cumulative cycle count total, the cycle count values from the successive Cycle Count packets can be added together to obtain this.

D4.5.11.5 Event trace

I_{GGMQT} The ETE architecture supports the tracing of additional information in the trace stream. These are known as *ETEEvents*, also known as Event trace. The trace unit supports up to 4 *ETEEvents*. The generation of *ETEEvents* is controlled by selecting resources selectors. The occurrence of *ETEEvents* can be communicated in the following ways:

- To the system by External Outputs.
- To the trace analyzer by Event Packet.

D4.5.11.6 No overflow

I_{DJCLX} A trace unit might include an optional feature to prevent overflows, which is indicated by TRCIDR3.NOOVERFLOW.

R_{YSPHL} TRCSTALLCTLR.NOOVERFLOW controls the no overflow feature.

S_{BSPDF} Enabling the no overflow feature might have a significant impact on PE performance.

R_{LCGZJ} While the no overflow feature is enabled, and while the number or frequency of *ETEEvents* is below an IMPLEMENTATION DEFINED threshold, the trace unit does not overflow.

R_{VHMZX} The threshold is greater than or equal to one of each numbered *ETEEvent*, for each trace session.

R_{MYMKW} When TRCIDR3.SYSSTALL is 0 the Effective value of TRCSTALLCTLR.NOOVERFLOW is 0 which means the no overflow feature is disabled.

R_{JYYLV} When TRCSTALLCTLR.ISTALL is 0 and TRCSTALLCTLR.NOOVERFLOW is 1, it is CONSTRAINED UNPREDICTABLE whether any stalling is disabled or whether the no overflow feature is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.11.7 Stalling the execution of the PE

I_{MPBVQ} The trace unit can be programmed to reduce the likelihood of a trace unit buffer overflow. If the trace unit is configured to support PE stalling, TRCIDR3.STALLCTL indicates that PE stalling is implemented and TRCIDR3.SYSSTALL indicates that PE stalling is permitted, then the execution of the PE can be slowed.

I_{HPFQP} It is permissible that the operation of the PE can be affected by the programming of the trace unit. The amount of intrusion and when stalling occurs is IMPLEMENTATION DEFINED. Additional stalling of the PE execution can be achieved by enabling this feature.

I_{VZSVK} Trace unit stalling of the PE is independent of the operation of the PE.

R_{NVBGS} PE operations which explicitly interact with the trace unit complete independently of the programming of the ability of the trace unit to stall the PE.

R_{SCLVV} The trace unit does not stall the PE while any of the following are true:

- The trace unit is in the Disabled state.
- The PE is executing in a Trace Prohibited region (see Trace Prohibited Regions).
- The PE is in Debug State.
- The PE does not allow stalling, that is, TRCIDR3.SYSSTALL == `0b0`.
- Self-hosted trace is disabled and `ExternalInvasiveDebugEnabled()` == FALSE.
- When TRCSTALLCTLR.ISTALL == `0b0` and TRCSTALLCTLR.NOOVERFLOW == `0b0`.
- Trace output is disabled.

R_{RWTYJ} When all of the following are true, the trace unit is permitted to stall the PE:

- Stalling of the PE is not prohibited by R_{SCLVV}.
- TRCSTALLCTLR.ISTALL == `0b1`.
- Any of the following are true:
  - TRCSTALLCTLR.NOOVERFLOW == `0b1`.
  - The available space in the internal storage of the trace unit is below the level indicated in TRCSTALLCTLR.LEVEL.

Otherwise, the trace unit does not stall the PE due to the stalling feature or no overflow feature.

R_{NVKXX} The trace unit does not indefinitely stall the operation of the PE.

I_{XPRQJ} In a multi-threaded processor, if the trace unit stalls a PE, Arm recommends that stalling or disruption of the processing of other PEs is minimized. In particular, if tracing of one or more PEs in a multi-threaded processor is enabled but tracing of other PEs in the multi-threaded processor is disabled, Arm recommends that if the PEs being traced are stalled by their respective trace units then the stall has minimal effect on the PEs which are not being traced.

I_{KBXXH} The levels indicated in TRCSTALLCTLR.LEVEL are the levels of intrusion allowed.

I_{ZRQBK} A summary of the stalling and no overflow scenarios is shown in Table D4-25, when TRCIDR3.STALLCTL is 1 and TRCIDR3.SYSSTALL is 1.

Table D4-25 Summary of TRCSTALLCTLR stalling and no overflow features

ISTALL	NOOVERFLOW	LEVEL	Description
0	0	x	Stalling is disabled.
0	1	x	It is CONSTRAINED UNPREDICTABLE whether the no overflow feature is enabled or stalling is disabled.
1	0	Zero	Stalling is enabled at the minimum level.
1	0	Nonzero	Stalling is enabled and is based on the value in TRCSTALLCTLR.LEVEL.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISTALL	NOOVERFLOW	LEVEL	Description
1	1	Zero	The no overflow feature is enabled, preventing overflows.
1	1	Nonzero	The no overflow feature is enabled, preventing overflows, and TRCSTALLCTLR.LEVEL might cause stalling earlier than necessary to prevent overflows.

D4.5.11.8 Timestamping

I_{YVBBW} The trace unit supports timestamping, where a common global time value is inserted in to the trace stream. These timestamps may be used to correlate between multiple trace streams, including those from other PEs or other sources of trace. These timestamps may be used to determine the passage of time, for analyzing performance.

I_{VFWQS} When timestamping is enabled, the trace unit inserts *Timestamp elements* in to the trace stream. Each Timestamping element indicates the time that a recent *P0 element*, *Event element*, or *Instrumentation element* occurred, and can be used to accurately determine when that element occurred.

I_{YFPDZ} The time value included in *Timestamp elements* is selected by TRFCR_EL1 and TRFCR_EL2 and is one of:

- Physical time, as seen by the generic timers in the PE.
- Virtual time, as seen by the generic timers in the PE.
- An IMPLEMENTATION DEFINED time value, often supplied by a CoreSight system.

I_{CLZKR} The insertion of *Timestamp elements* is controlled by TRCCONFIGR.TS and TRCTSCTLR.

D4.5.11.9 Virtual context identifier tracing

I_{VMGBJ} Whether an implementation supports Virtual context identifier tracing is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed to output the identifier of a virtual machine that the PE is executing.

I_{FDDXM} This option is enabled by setting TRCCONFIGR.VMID to `0b1`.

R_{PTVYD} If the PE implements EL2, the trace unit supports a 32-bit Virtual context identifier, with TRCIDR2.VMIDSIZE indicating a 32-bit Virtual context identifier size. The source of the Virtual context identifier is CONTEXTIDR_EL2.PROCID.

R_{BRDYF} If the PE does not implement EL2, the trace unit does not support a Virtual context identifier, with TRCIDR2.VMIDSIZE indicating Virtual context identifier tracing is not supported.

Note

Previous trace architectures from Arm supported the ability to select the source of the Virtual context identifier. This specification does not support Virtual context identifier selection, and only permits CONTEXTIDR_EL2.PROCID as the source of the Virtual context identifier. See TRCIDR2.VMIDOPT for more details.

D4.5.12 Compression

I_{WVBMT} Additional compression of the trace byte stream is achieved by the following methods:

- Removing elements that can be implied by the trace analyzer:
  - Implying the existence of *Commit elements* based on the tracing of other elements.
  - Removing *Target Address elements* that can be calculated by the trace analyzer by analysis of previous traced PE execution.
- Combining multiple elements together into a single packet:
  - Combining *Atom elements* into a single packet.
  - Combining *Cancel elements* and *Mispredict elements* into a single packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.12.1 Implied commits

I_{BTGGX} The ETE trace protocol provides mechanisms to minimize the number of *Commit elements* which need to be explicitly output in the trace byte stream. When a *P0 element* is output in the trace byte stream, if the number of speculative *P0 elements* output exceeds TRCIDR8.MAXSPEC, then a *Commit element* is implied which resolves the oldest speculative *P0 element*.

R_{YKLRM} The trace unit does not generate commit packets for *Commit elements* that have been implied by the trace protocol.

D4.5.12.2 Atom packing

I_{QMVNP} The ETE trace protocol provides packets which allow groups of consecutive *Atom elements* to be packed into a single trace packet. Figure D4-15 shows the decision tree for generating the different formats of Atom packets.

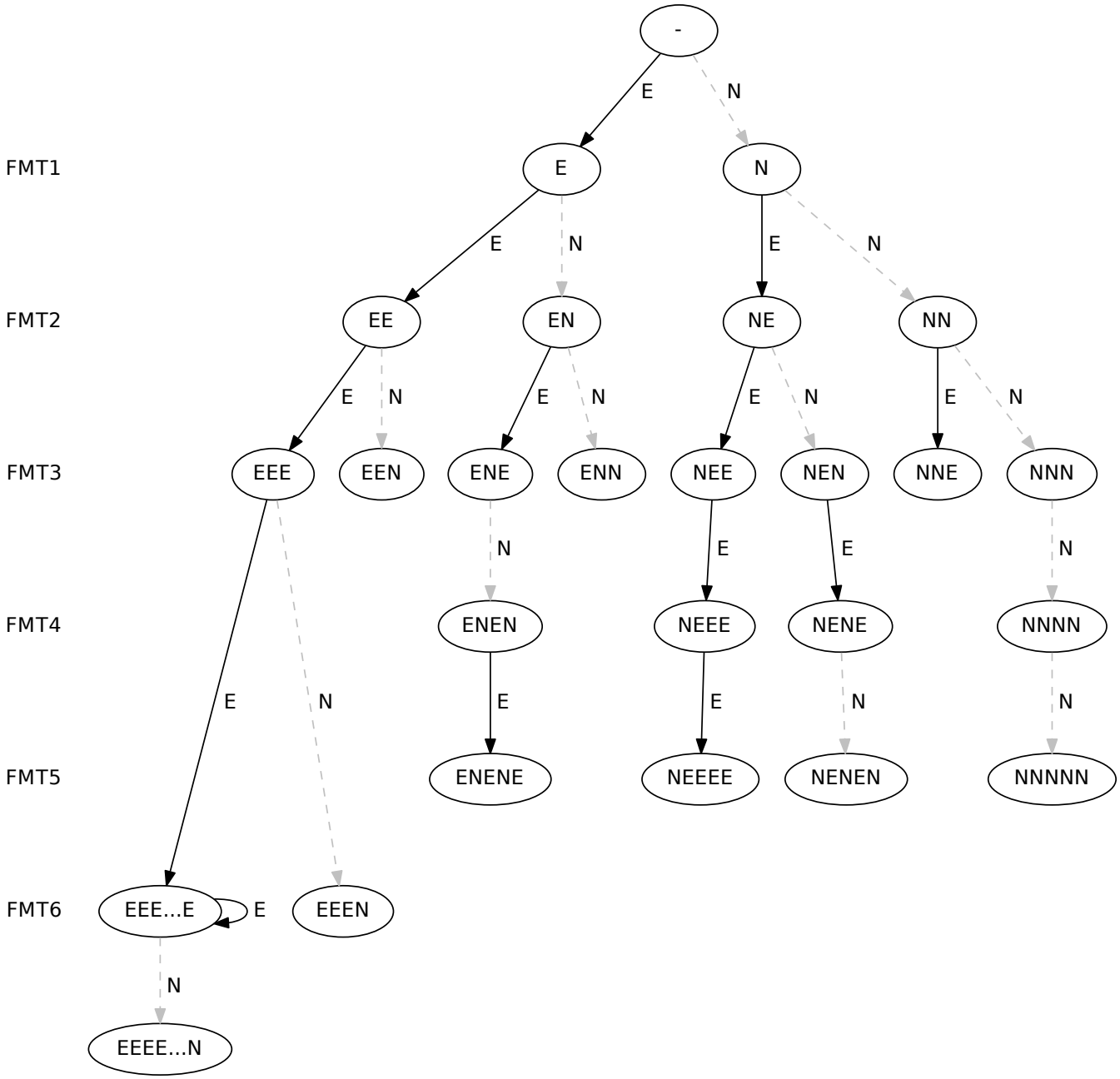

Figure D4-15 Atom packing

I_{XXSGS} Cancel Packets can indicate a number of *Atom elements* and the *Cancel element*.

I_{KWWHP} The Mispredict Packets can indicate a number of *Atom elements* and the *Mispredict element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5.12.3 Address compression

I_{LFFCR} The trace unit can remove program addresses from the trace stream. The trace analyzer can infer the addresses from the program image and previous history.

This includes the targets of direct *P0 instructions*, where the target address is encoded in the instruction itself.

R_{SJPYH} The trace unit retains the *Address information* of up to the last three addresses that were explicitly output in the trace protocol, as contained in:

- Target Address packets.
- Source Address packets.
- Exception packets.
- PE Reset packets.
- Q packets.

I_{SXNYK} The explicitly output addresses that the trace unit retains are known as the address history buffer.

I_{LPYRK} For optimized trace protocol efficiency, Arm recommends that the address history buffer is three entries deep.

R_{LMHFW} When any of the following packets are generated, the trace unit pushes the address value and sub_isa to the address history buffer:

- Target Address packet.
- Source Address packet.
- Q packet that implies a *Target Address element*.

R_{CMCRT} When an Exception packet is generated, the trace unit pushes the preferred exception return address value and sub_isa to the address history buffer.

R_{FPDFJ} When a PE Reset packet is generated with an UNKNOWN address, the trace unit pushes an address value of `0x0` and sub_isa of IS0 to the address history buffer.

R_{BPRDC} When a Target Address packet is generated, the trace unit uses the address history buffer to identify when a Target Address Exact Match packet can be used. When a Target Address Exact Match packet cannot be used, the most recent entry in the address history buffer is used for the Target Address packet selection.

R_{GWKFD} When a Source Address packet is generated, the trace unit uses the address history buffer to identify when a Source Address Exact Match packet can be used. When a Source Address Exact Match packet cannot be used, the most recent entry in the address history buffer is used for the Source Address packet selection.

R_{YLXFK} When an Exception packet is generated, the trace unit uses the address history buffer to identify when an Exception Exact Match Address packet can be used. When an Exception Exact Match Address packet cannot be used, the most recent entry in the address history buffer is used for the Exception Address packet selection.

R_{YCMCG} When a Q packet is generated which implies a *Target Address element*, the trace unit uses the address history buffer to identify when a Q with Exact Match Address packet can be used. When a Q with Exact Match Address packet cannot be used, the most recent entry in the address history buffer is used for the Q Address packet selection.

R_{BTWGD} When a Trace Info packet is generated, the trace unit sets all entries of the address history buffer to have an address value of `0x0` and sub_isa of IS0.

D4.5.12.4 Return stack address matching

I_{NHWVZ} The trace unit might contain the optional return stack function. The return stack operates when Branch with Link instructions or indirect *P0 instructions* are taken, and provides a mechanism to allow the trace unit to remove certain *Target Address elements* from the trace element stream. The trace analyzer maintains an independent copy of the return stack which is used to determine when *Target Address elements* have been removed and then infer the target of indirect *P0 instructions*.

R_{HNDJJ} The depth of the return stack is IMPLEMENTATION DEFINED from 0 to 15 entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{BLYHW} For optimized trace protocol efficiency, Arm recommends the trace unit implements the return stack with at least 3 entries.

R_{HFCTC} While TRCCONFIGR.RS is 1, when a Branch with Link instruction is predicted as taken and is traced, the trace unit pushes the following *Address information* to the return stack:

- The instruction address + the instruction size, that is, the return address for the Branch with Link instruction.
- The sub_isa from the instruction set encoding.

R_{ZKTHK} When a return stack push occurs, all existing entries are shifted down one place on the return stack and the new entry is pushed to the top entry of the return stack.

R_{ZSVDQ} While the return stack is full, when a return stack push occurs, the oldest entry on the return stack is discarded.

R_{FFXPW} When a Branch with Link instruction is predicted as taken and traced with an E *Atom element*, when a return stack push occurs, the trace unit pushes to the return stack, even if the prediction is incorrect and is subsequently corrected to an N *Atom element*.

R_{NYHFH} When a Branch with Link instruction is predicted as not taken and traced with an N *Atom element*, the trace unit does not push to the return stack, even if the prediction is incorrect and is subsequently corrected to an E *Atom element*.

R_{GVLKJ} When a Branch with Link instruction is implied by a *Q element*, the trace unit does not push to the return stack.

R_{WRXCW} When a Branch with Link instruction is executed in a branch broadcasting region, the trace unit does not push to the return stack.

R_{FVRPZ} When a Branch with Link instruction is speculatively executed and generates an E *Atom element*, if that *Atom element* is subsequently cancelled by a *Cancel element* then any corresponding entry on the trace unit return stack must be popped from the trace unit return stack.

Note

Discarding the entire contents of the trace unit return stack meets this obligation.

R_{QHSBN} When an indirect *P0 instruction* is taken and traced, and the *Address information* in the resultant *Target Address element* matches the address and sub_isa on the top of the return stack, the trace unit performs a return stack pop.

R_{HTKJS} When a return stack pop occurs, both of the following occur:

- The trace unit discards the *Target Address element* that matches the address and sub_isa on the top of the return stack.
- The trace unit removes the top entry of the return stack, and shifts each older entry up one position.

R_{WBCJG} When an indirect *P0 instruction* is implied by a *Q element*, the trace unit does not perform a return stack pop.

I_{BCWSQ} When an indirect *P0 instruction* is taken, it is possible that the target address is predicted incorrectly by the PE.

R_{YMRGB} When the target address of a taken indirect *P0 instruction* is incorrectly predicted, and the incorrect target address is traced with a *Target Address element*, the trace unit corrects the incorrect address by generating a new *Target Address element* with the correct target address, and neither of the target addresses cause a return stack pop.

R_{GBHNP} When the target address of a taken indirect *P0 instruction* is incorrectly predicted, and the incorrect target address matches the top entry of the return stack, the trace unit subsequently generates a *Target Address element* with the correct target address, and neither of the target addresses cause a return stack pop.

R_{ZCCBS} When the final status of the *Atom element* corresponding to an indirect *P0 instruction* is E, including when one or more Mispredict elements change the status of the *Atom element*, the trace unit performs a return stack pop.

Note

A return stack push only occurs if the initial *Atom element* state for the Branch with Link instruction is E. Conversely, a return stack pop only occurs if the final *Atom element* state for the indirect *P0 instruction* is E.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SLDXR} When an instruction that is both a Branch with Link instruction and an indirect *P0 instruction* is executed, the trace unit performs the following actions on the return stack, in order:

1. Determine whether a return stack push is possible and push if required.
2. Determine whether a return stack pop is possible and pop if required.

Note

Some Arm trace architectures use a different order of operations.

R_{SBZJB} When any of the following occur, the trace unit discards the contents of the return stack:

- The trace unit generates a *Trace Info element*.
- The trace unit generates a *Trace On element*.
- The PE enters a branch broadcasting region.

I_{XRQHQ} A trace unit might discard the contents of the return stack at any time.

I_{DCNGF} When the return stack contents are discarded, there is no requirement for the trace analyzer to be aware that this discard operation has occurred. This is because even though the contents of the trace unit return stack are discarded, there are no adverse consequences if the contents of the trace analyzer return stack are retained, but never used.

R_{GZSSX} After a *Trace Info element*, a *Target Address element* and a *Context element* are required but might not be generated immediately. If the *Target Address element* and the *Context element* are not generated before the next *P0 element*, then any Branch with Link instructions must not push on to the return stack until both the *Target Address element* and the *Context element* have been generated.

Note

This restriction prevents the trace unit from performing return stack pushes for instructions that the trace analyzer cannot analyze, because it is not yet fully synchronized.

D4.5.12.5 Timestamp value compression

I_{GYYNG} The trace analyzer maintains a copy of the last *Timestamp element* value broadcast. The *Timestamp element* value might be compressed relative to the last value and only the bits that have changed need to be encoded.

R_{GPGQQ} When a Trace Info packet is generated, the trace unit sets its maintained value of the last *Timestamp element* to zero, and when the trace unit generates a subsequent Timestamp packet the value is compressed relative to this new zero value. This means that the first Timestamp packet after a Trace Info packet contains all nonzero bits of the Timestamp value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6 Resource operation

I_{PKCDG} FEAT_ETE has a number of resources that can be used to provide advanced filtering functionality.

R_{PWBZK} The resources operate in one of the following states:

Running
All the resources are active.

Pausing
The resources are progressing to the Paused state.

Paused
All the resources are static and inactive except for External Input Selectors.

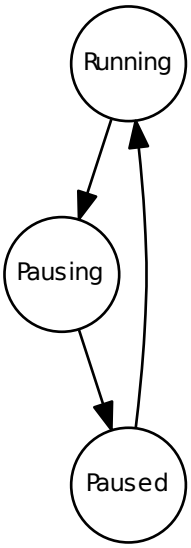

Figure D4-16 Resources operation

I_{HWYYK} As described in System behaviors, the trace unit can be disabled by either:

- Setting TRCPRGCTLR.EN to 0.
- Locking the OS Lock, by setting OSLAR_EL1.OSLK to 1.

R_{JLLVN} While the resources are in the Running state, and when any of the following occur, the resources enter the Pausing state:

- The trace unit becomes disabled.
- The trace unit enters the low-power state.
- The PE begins executing in a Trace Prohibited region.

R_{YWDVJ} While the resources are in the Pausing state, the resources enter the Paused state in finite time.

R_{LYFDT} While the trace unit is in the Paused state, when all of the following are true, the resources enter the Running state:

- The trace unit is enabled.
- The trace unit is not in the low-power state.
- The PE is not executing in a Trace Prohibited region.

R_{TMPZZ} A trace unit buffer overflow has no impact on the behavior of the resources.

D4.6.1 Behavior of the resources while in the Running state

S_{JVYQP} The time taken for the resources to operate might vary between different trace unit implementations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6.2 Behavior of the resources while in the Pausing state

R_{RDCGC} When the resources enter the Pausing state, the resources perform the following procedure:

1. All resources, except for the Sequencer and any Counters, are driven low as inputs to the Resource Selector logic. The Counters and the Sequencer behave as normal.
2. The states that the inputs were at before they were driven low are propagated through the Resource Selector logic.
3. The states of the Counters and the Sequencer are propagated through the Resource Selector logic one more time. That is, the states of the Counters and the Sequencer are propagated through the Resource Selector logic for the length of time that it takes for the state of a resource to be propagated through the Resource Selector logic.
4. The resources enter the Paused state.

I_{LGWRK} The procedure that the resources perform when the resources are in the Pausing state has the result that, for resource events that are activated by a resource that is not a Counter or a Sequencer, no activity is lost, because all those resource events are updated.

I_{CQNGN} When Counter and Sequencer states are propagated back as resources, so that a loop is created, then the following are true:

- If a Counter at zero resource is being used to activate the Sequencer or a Counter, then that Counter at zero resource might be propagating through the Resource Selector logic at the time when the procedure ends. In this case, the Sequencer state resource or other Counter at zero resource that is activated by that Counter at zero resource might be lost.
- If a Sequencer state resource is being used to activate a Counter, then that Sequencer state resource might be propagating through the Resource Selector logic at the time when the procedure ends. In this case, the Counter at zero resource that is activated by that Sequencer state resource might be lost.

I_{BRZXY} When the trace unit becomes disabled, the behavior of the resources in the Pausing state ensures that the programmers' model provides a consistent view of the state of the trace unit resources. That is, regarding the Counters and the Sequencer, the following are true:

- If the state of the Sequencer propagated back as a resource, then the view of the Sequencer as a resource event and the view of the Sequencer resource state both show the same Sequencer state.
- If the state of a Counter is propagated back as a resource, then the view of the Counter as a resource event and the view of the Counter resource state both show the same Counter state.
  The Counter state might be one of the following:
  - The Counter is at zero.
  - The Counter is not at zero.

D4.6.3 Behavior of the resources while in the Paused state

I_{YXKSQ} The behavior of the resources when the PE enters the low-power state or a Trace Prohibited region differs from other trace architectures defined by Arm.

R_{FHYQW} While the resources are in the Paused state and the trace unit is not disabled, the resources do not lose resource events that are in transition, except those permitted when moving through the Pausing state of the resources. See Behavior of the resources while in the Pausing state for details on the resource events that are permitted to be lost when in the Pausing state.

I_{HZRSS} While the resources are in the Paused state, the resources might not observe resource events that are in transition until after the resources leave the Paused state.

R_{YWQNQ} While the resources are in the Paused state, the resources remain in the state they are in.

R_{BQSMN} While the resources are in the Paused state, the trace unit drives all External Outputs low.

R_{MVZYP} When the trace unit becomes disabled and the resources enter the Paused state, and not before, the trace unit might set TRCSTATR.PMSTABLE to 1.

R_{RWNTS} While TRCSTATR.PMSTABLE is set to 1, all resources and resource events remain in a quiescent state.

I_{ZSHWT} The behavior of the External Input Selectors is detailed in Operation while in Paused state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6.4 Behavior of resources on a Trace synchronization event

R_{RFSRY} When the following resources have finished calculations for all instructions prior to the previous Context synchronization event, a Trace synchronization event completes:

- Address Comparators.
- Context Identifier Comparators.
- Virtual Context Identifier Comparators.
- Single-shot Comparator Controls.

D4.6.5 Resource organization

I_{NJLRF} There are 2 types of resources:

- Precise resources.
- Imprecise resources.

I_{JCKLL} Each resource has a current state, which is output as a *Resource state*. The Resource state is selected by Resource Selectors, and then used by various trace unit functions as a *Resource event*, see Figure D4-17.

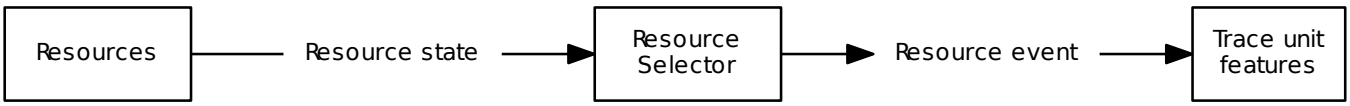

Figure D4-17 Resources organization

D4.6.5.1 Precise resources

I_{QSPKY} The precise resources are used in the evaluation of the ViewInst include/exclude function and the ViewInst start/stop function.

R_{WNGDH} The trace unit evaluates the precise resources for each instruction block. See Instruction block for more details.

R_{NFDCZ} The trace unit maintains execution order of the precise resources.

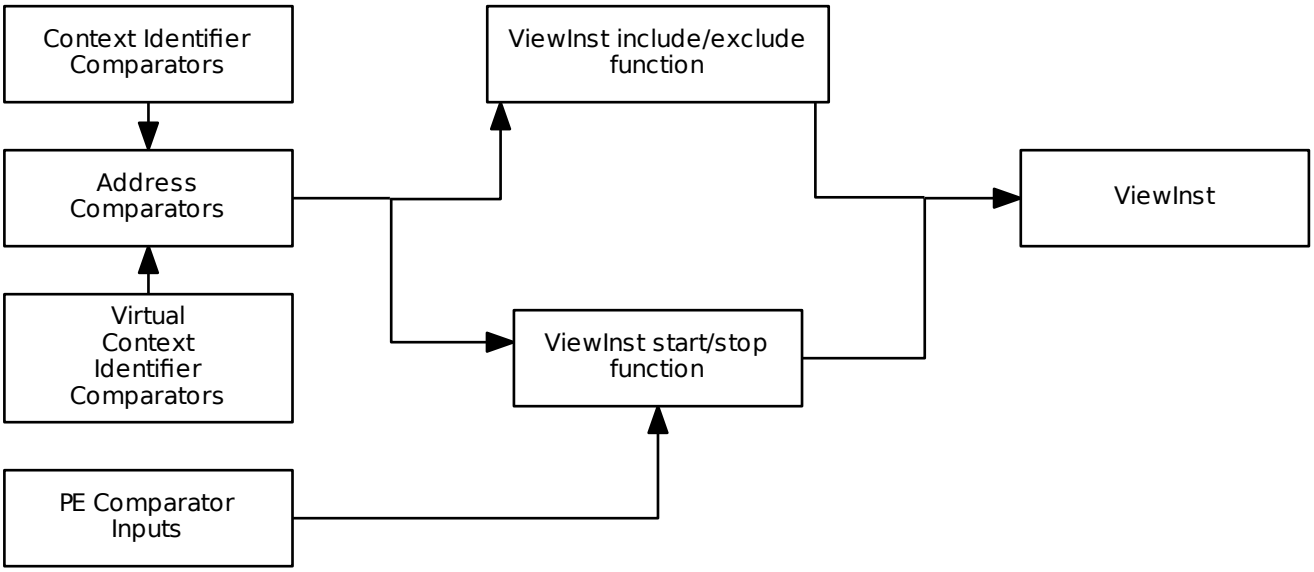

Figure D4-18 Precise resource path

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6.5.2 Imprecise resources

I_{NNHSY} If a Resource is not a precise resource, it is an imprecise resource.

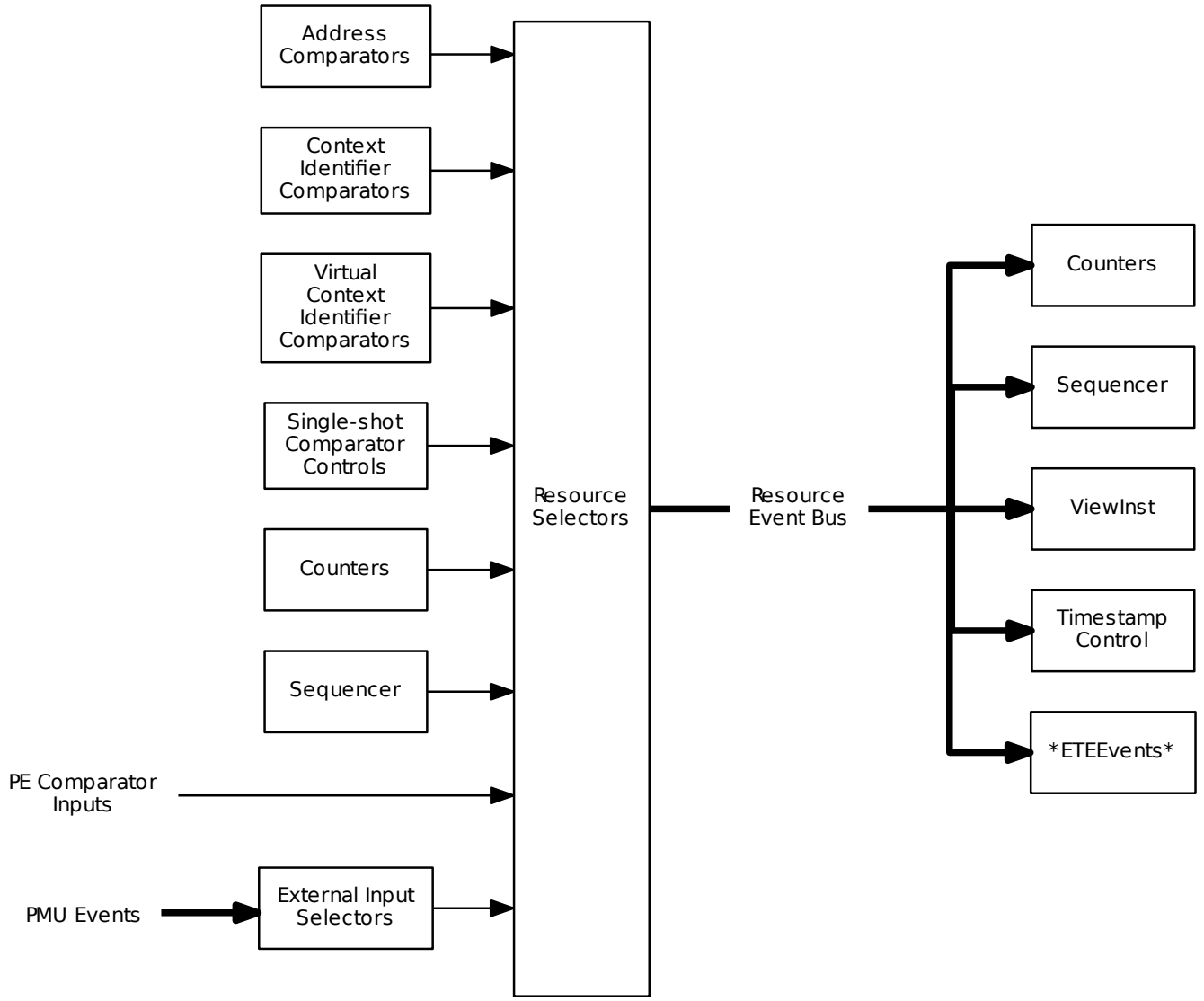

Figure D4-19 Imprecise resource path

D4.6.5.3 Selecting a resource or a pair of resources

I_{BRQFW} A resource is selected by using a Resource Selector.

R_{QDJVV} Each Resource Selector uses one of the 30 TRCRSCTLR<n> registers. The trace unit implements Resource Selectors in pairs, so that a maximum of 15 programmable pairs can be implemented.

R_{NRSGN} Resource Selector 0 always provides a FALSE result.

R_{SXSQT} While the resources are in the Running state, Resource Selector 1 provides a TRUE result.

I_{TQVKS} TRCIDR4.NUMRSPAIR indicates how many pairs of Resource Selectors are implemented.

I_{MSHWC} Resource Selectors can be used in pairs or used individually. When a pair of Resource Selectors is used, a Boolean function can be applied to the outputs of the combination of selected resources. See Figure D4-21.

R_{WZVDQ} While TRCRSCTLR<n>.SELECT[m] is 1, the Resource Selector selects the Resource Number m of the group selected by TRCRSCTLR<n>.GROUP as described in Table D4-26.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D4-26 Resource grouping

Group	Resource Number	Resource
0b0000	0-3	External Input Selectors 0-3
	4-15	Reserved
0b0001	0-7	PE Comparator Inputs 0-7
	8-15	Reserved
0b0010	0	Counter 0 at zero
	1	Counter 1 at zero
	2	Counter 2 at zero
	3	Counter 3 at zero
	4	Sequencer state 0
	5	Sequencer state 1
	6	Sequencer state 2
	7	Sequencer state 3
	8-15	Reserved
0b0011	0-7	Single-shot Comparator Control 0-7
	8-15	Reserved
0b0100	0-15	Single Address Comparator 0-15
0b0101	0-7	Address Range Comparator 0-7
	8-15	Reserved
0b0110	0-7	Context Identifier Comparator 0-7
	8-15	Reserved
0b0111	0-7	Virtual Context Identifier Comparator 0-7
	8-15	Reserved
0b1xxx	0-15	Reserved

R_{HVNQG} While TRCRSCTLR<n>.INV is set to 0 and one or more resources in a group are selected, when any of the outputs of the selected resources are high, the Resource Selector fires.

R_{WFGMY} While TRCRSCTLR<n>.INV is set to 1, when none of the outputs of the selected resources are high, the Resource Selector fires.

I_{SZZMP} Figure D4-20 summarizes the process of resource selection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

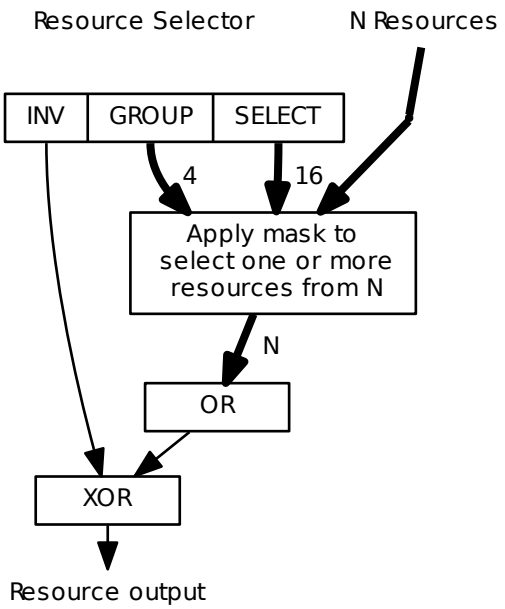

Figure D4-20 A Resource Selector

D4.6.5.4 A Resource Selector pair

I_{DLRMJ} The Resource Selectors are arranged in pairs, and the result of each of a pair of Resource Selectors can be combined using a Boolean function and used to drive other resources and events in the trace unit.

R_{KTNJM} For each TRCRSCTLR<n> register which is the lower register for a pair of Resource Selectors, the TRCRSCTLR<n> register has the TRCRSCTLR<n>.PAIRINV field.

I_{QKTSJ} For example:

- TRCRSCTLR2 and TRCRSCTLR3 constitute a Resource Selector pair. In this case:
  - TRCRSCTLR2 is the lower register.
  - TRCRSCTLR2.PAIRINV optionally inverts the result of the Boolean function that is applied to the outputs of the combination of selected resources.
  - TRCRSCTLR3 is the upper register.
  - TRCRSCTLR3.PAIRINV is RES0.

This means that, when a Resource Selector pair is used, the following scenario is possible:

- One TRCRSCTLR<n> might select only one resource within the group.
- The other TRCRSCTLR<n> might select more than one resource from the group, so that the result is a logical OR of the selected resources.
- A Boolean function, for example a logical AND, might be applied to the outputs of the combination of selected resources.
- The result of that Boolean function might be inverted by using PAIRINV.

I_{LPJXK} In Figure D4-21, the Boolean function is selected by using the INV field for each Resource Selector, with the PAIRINV field for each Resource Selector pair, see Table D4-27.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

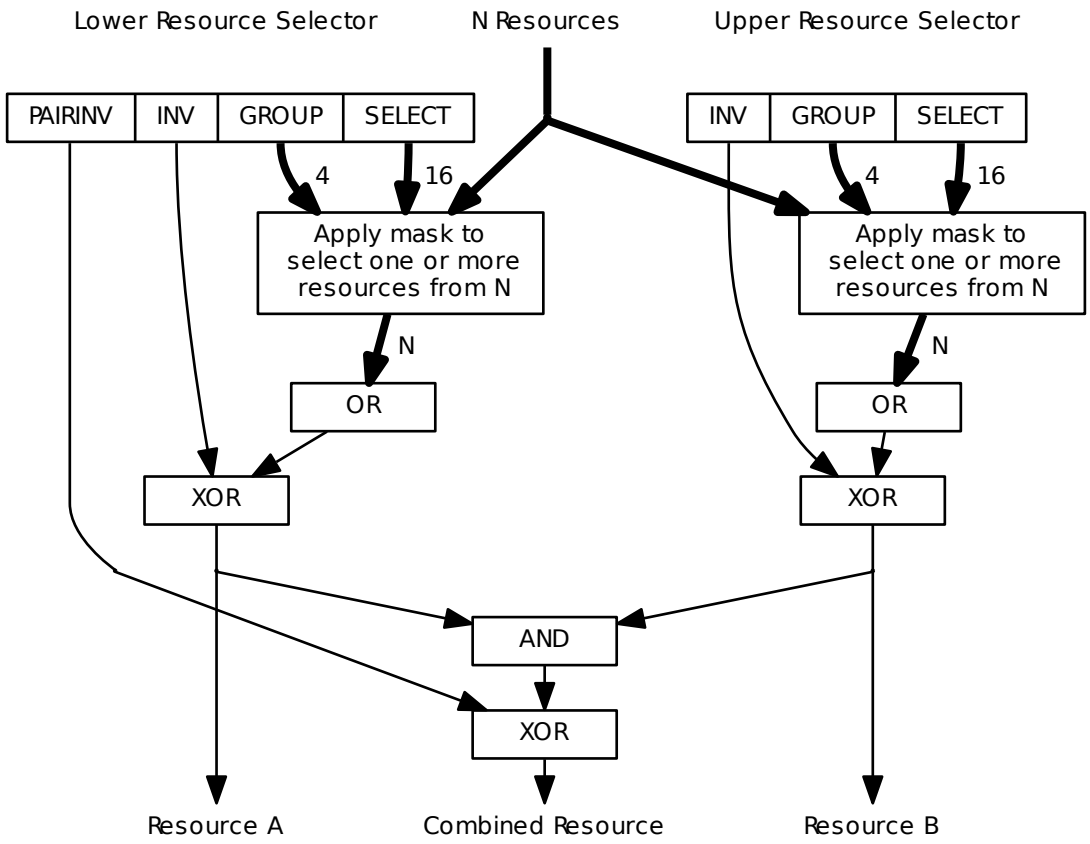

Figure D4-21 A Resource Selector pair

Table D4-27 Selecting a Boolean function

Function	PAIRINV	Resource A INV	Resource B INV
A AND B	0b0	0b0	0b0
NOT(A AND B)	0b1	0b0	0b0
Reserved	0b0	0b0	0b1
NOT(A) OR B	0b1	0b0	0b1
NOT(A) AND B	0b0	0b1	0b0
Reserved	0b1	0b1	0b0
NOT(A) AND NOT(B)	0b0	0b1	0b1
A OR B	0b1	0b1	0b1

D4.6.6 Address comparators

I_{LGGVG} An ETE trace unit provides between 0 and 16 *Single Address Comparators* (SACs), each of which compares the instruction address with a user-programmed value.

R_{YCRNP} The trace unit implements SACs in pairs, so that a trace unit implementation contains an even number of SACs.

I_{MNTCY} TRCIDR4.NUMACPAIRS indicates how many pairs of SACs are implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{YMVDZ} When the PE executes instructions in Debug state, Address Comparators do not match.

R_{YPWLR} When the PE executes instructions in a Trace Prohibited region, Address Comparators do not match.

I_{WDJPG} Address Comparators might match on Speculative execution.

D4.6.6.1 Single Address Comparators

I_{SSHHT} A SAC can be used in the following ways:

- As inputs to the ViewInst start/stop function in the ViewInst function. For more information, see ViewInst start/stop function filtering.
- As an individual resource.
- The comparator can be programmed so that, whenever the PE is in a specific Security state, the comparator only matches in certain Exception levels.

R_{DKCFF} An SAC only matches on Exception levels and Security states as indicated by TRCACATR<n>.

$$\text{SAC_EL}_i[n] = \begin{cases} \neg\text{TRCACATR<n>.EXLEVEL_S_EL0} & \text{Secure EL0} \\ \neg\text{TRCACATR<n>.EXLEVEL_S_EL1} & \text{Secure EL1} \\ \neg\text{TRCACATR<n>.EXLEVEL_S_EL2} & \text{Secure EL2} \\ \neg\text{TRCACATR<n>.EXLEVEL_S_EL3} & \text{EL3} \\ \neg\text{TRCACATR<n>.EXLEVEL_NS_EL0} & \text{Non-secure EL0} \\ \neg\text{TRCACATR<n>.EXLEVEL_NS_EL1} & \text{Non-secure EL1} \\ \neg\text{TRCACATR<n>.EXLEVEL_NS_EL2} & \text{Non-secure EL2} \\ \neg(\text{TRCACATR<n>.EXLEVEL_RL_EL0} \oplus \text{TRCACATR<n>.EXLEVEL_NS_EL0}) & \text{Realm EL0} \\ \neg(\text{TRCACATR<n>.EXLEVEL_RL_EL1} \oplus \text{TRCACATR<n>.EXLEVEL_NS_EL1}) & \text{Realm EL1} \\ \neg(\text{TRCACATR<n>.EXLEVEL_RL_EL2} \oplus \text{TRCACATR<n>.EXLEVEL_NS_EL2}) & \text{Realm EL2} \end{cases}$$

R_{QFNSK} An SAC only matches on the context indicated by TRCACATR<n>.CONTEXT and TRCACATR<n>.CONTEXTTYPE.

$$m = \text{TRCACATR<n>.CONTEXT}$$
$$\text{type} = \text{TRCACATR<n>.CONTEXTTYPE}$$

$$\text{SAC_CONTEXT}_i[n] = \begin{cases} 1 & \text{type is 0} \\ \text{CIDCOMP}[m] & \text{type is 1} \\ \text{VMIDCOMP}[m] & \text{type is 2} \\ \text{CIDCOMP}[m] \wedge \text{VMIDCOMP}[m] & \text{type is 3} \end{cases}$$

R_{PYMZV} When an instruction is executed, and the address of the lowest byte of the instruction exactly matches the programmed address of an SAC, the SAC matches.

$$\text{SAC_ADDR}_i[n] = (\text{ThisInstrAddr()}_i \equiv \text{TRCACVR<n>.ADDRESS})$$

I_{SPFXX} For example, for a 4-byte instruction at address `0x1000`:

- The lowest byte of the instruction is at `0x1000`.
- The second byte of the instruction is at `0x1001`.
- The third byte of the instruction is at `0x1002`.
- The highest byte of the instruction is at `0x1003`.

If an SAC is programmed with `0x1000`, then it always matches on that instruction at address `0x1000`.

I_{JZXFJ} It is IMPLEMENTATION DEFINED whether an SAC matches when its programmed address matches any byte of an instruction which is not the lowest byte of the instruction.

I_{VSFSS} The Arm architecture supports disabling `IT` instructions on more than one subsequent instruction, using the ITD bits in the SCTLR, HSCTLR, and SCTLR_EL1 System registers. If any of the ITD bits are set to 1 and are affecting `IT`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

operation, and a SAC is programmed to match on the address of the instruction that is immediately after an `IT` instruction, when the instruction immediately after the `IT` instruction is executed it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{TFYFT} If any of the ITD bits are set to 1, Arm recommends that a SAC is programmed to match on the address of the `IT` instruction, instead of the instruction immediately after the `IT` instruction.

S_{MLDYK} To avoid unexpected behavior from an SAC, Arm recommends that the SAC is always programmed with an address that is for the lowest byte of an instruction.

I_{MCKFH} When the instruction immediately after a `MOVPRFX` instruction executes, if a SAC is programmed to match on the address of this instruction, then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{FPTHL} Arm recommends that a SAC is programmed to match on the address of the `MOVPRFX` instruction, instead of the instruction immediately after the `MOVPRFX` instruction.

I_{TBNTJ} The operation of a SAC is as follows:

$$\mathrm{SAC}_i[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \mathrm{SAC_ADDR}_i[n] \wedge \mathrm{SAC_EL}_i[n] \wedge \mathrm{SAC_CONTEXT}_i[n] & \text{Otherwise} \end{cases}$$

D4.6.6.2 Address Range Comparators

I_{HDFQM} Pairs of SACs are arranged to form one *Address Range Comparator* (ARC). An ARC is programmed with an address range, so that whenever any address in that range is accessed, the ARC matches. A trace unit contains between zero and eight ARCs. ARCs can be used in the following ways:

- Selected for the ViewInst include/exclude function in the ViewInst function. See ViewInst include/exclude function filtering.
- As individual resources.

An ARC is programmed by programming the SACs as follows:

- The first SAC is programmed with the start address of the instruction range.
- The second SAC is programmed with the end address of the instruction range.

S_{WFSPV} The address that the second SAC is programmed with must be greater than or equal to the address that the first SAC is programmed with, that is, the end address must be greater than or equal to the start address.

R_{MXCGD} While the start address of an ARC is greater than the end address, the behavior of the ARC is CONSTRAINED UNPREDICTABLE, that is, at any time the ARC might do one of the following:

- Match.
- Not match.

R_{XYJLC} While the TRCACATR<n> registers for the SACs in an ARC are programmed to different values, the behavior of the ARC is CONSTRAINED UNPREDICTABLE.

R_{LLQPL} While an ARC is programmed with an instruction address range, when the PE executes an instruction at an address in the following range, the ARC matches:

$$\begin{aligned} \text{start_address} &= \text{TRCACVR<n>.ADDRESS} \\ \text{end_address} &= \text{TRCACVR<n+1>.ADDRESS} \\ \mathrm{ARC_ADDR}_i[n/2] &= (\text{ThisInstrAddr()}_i \geq \text{start_address}) \wedge (\text{ThisInstrAddr()}_i \leq \text{end_address}) \end{aligned}$$

R_{YYXSQ} When an instruction is executed, and the address of the lowest byte of the instruction is within the programmed address range of an ARC, the ARC matches.

I_{RPFWZ} When an instruction is executed and the programmed address range of an ARC contains addresses for one or more bytes of the instruction, but does not contain the address for the lowest byte of the instruction, it is IMPLEMENTATION SPECIFIC whether the ARC matches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{RZFPT} For example, for a 4-byte instruction at address `0x1000`:

- The lowest byte of the instruction is at `0x1000`.
- The second byte of the instruction is at `0x1001`.
- The third byte of the instruction is at `0x1002`.
- The highest byte of the instruction is at `0x1003`.

If the programmed address range contains `0x1000`, then the ARC always matches. However, if the programmed address range starts at either `0x1001`, `0x1002`, or `0x1003`, then it is IMPLEMENTATION SPECIFIC whether the ARC matches.

S_{HSFTQ} To avoid unexpected behavior from an ARC, Arm recommends that the ARC is always programmed with an address range that starts with an address for the lowest byte of an instruction.

I_{VRRHS} The Arm architecture supports disabling `IT` instructions on more than one subsequent instruction, using the ITD bits in the SCTLR, HSCTLR, and SCTLR_EL1 System registers. If any of the ITD bits are set to 1 and are affecting `IT` operation, and an ARC is programmed to include the address of the instruction that is immediately after an `IT` instruction but not include the `IT` instruction, when the instruction immediately after the `IT` instruction is executed then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{DMHQH} If any of the ITD bits are set to 1, Arm recommends that an ARC is programmed to include both the `IT` instruction and the instruction immediately after the `IT` instruction.

I_{PBKPJ} When the instruction immediately after a `MOVPRFX` instruction is executed, if an ARC is programmed to include the address of the instruction that is after the `MOVPRFX` instruction but not the `MOVPRFX` instruction, then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{HVTHL} Arm recommends that an ARC is programmed to include both the `MOVPRFX` instruction and the instruction immediately after the `MOVPRFX` instruction.

I_{HTXLT} It might be possible for multiple matches to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place with each address in the programmed range. In this case, any of the following might occur:

- An address in the range is matched more than once.
- More than one address in the range is matched simultaneously.

R_{HMYMX} When multiple ARC matches occur simultaneously for one ARC, both of the following are true:

- The ARC signals a match at least once.
- The ARC does not signal more matches than the number of instructions that are executed with an address that matches an address in the programmed range.

I_{CTBDN} Each ARC can be used with one, or a combination of, the following:

- A Context Identifier Comparator.
- A Virtual Context Identifier Comparator.

R_{TMCJX} An ARC only matches on Exception levels and Security states as indicated by TRCACATR<2n>.

$$\text{ARC_EL}_i[n] = \begin{cases} \neg\text{TRCACATR<2n>.EXLEVEL_S_EL0} & \text{Secure EL0} \\ \neg\text{TRCACATR<2n>.EXLEVEL_S_EL1} & \text{Secure EL1} \\ \neg\text{TRCACATR<2n>.EXLEVEL_S_EL2} & \text{Secure EL2} \\ \neg\text{TRCACATR<2n>.EXLEVEL_S_EL3} & \text{EL3} \\ \neg\text{TRCACATR<2n>.EXLEVEL_NS_EL0} & \text{Non-secure EL0} \\ \neg\text{TRCACATR<2n>.EXLEVEL_NS_EL1} & \text{Non-secure EL1} \\ \neg\text{TRCACATR<2n>.EXLEVEL_NS_EL2} & \text{Non-secure EL2} \\ \neg(\text{TRCACATR<2n>.EXLEVEL_RL_EL0} \oplus \text{TRCACATR<2n>.EXLEVEL_NS_EL0}) & \text{Realm EL0} \\ \neg(\text{TRCACATR<2n>.EXLEVEL_RL_EL1} \oplus \text{TRCACATR<2n>.EXLEVEL_NS_EL1}) & \text{Realm EL1} \\ \neg(\text{TRCACATR<2n>.EXLEVEL_RL_EL2} \oplus \text{TRCACATR<2n>.EXLEVEL_NS_EL2}) & \text{Realm EL2} \end{cases}$$

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{VSBJF} An ARC only matches on the context indicated by TRCACATR<2n>.CONTEXT and TRCACATR<2n>.CONTEXTTYPE.

$$m = \text{TRCACATR<2n>.CONTEXT}$$
$$\text{type} = \text{TRCACATR<2n>.CONTEXTTYPE}$$

$$\text{ARC_CONTEXT}_i[n] = \begin{cases} 1 & \text{type is 0} \\ \text{CIDCOMP}[m] & \text{type is 1} \\ \text{VMIDCOMP}[m] & \text{type is 2} \\ \text{CIDCOMP}[m] \wedge \text{VMIDCOMP}[m] & \text{type is 3} \end{cases}$$

R_{RTXJN} The operation of an ARC is as follows:

$$\text{ARC}_i[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \text{ARC_ADDR}_i[n] \wedge \text{ARC_EL}_i[n] \wedge \text{ARC_CONTEXT}_i[n] & \text{Otherwise} \end{cases}$$

D4.6.7 Context Identifier Comparator

I_{KDSNY} An ETE trace unit provides between zero and eight Context Identifier Comparators. Each Context Identifier Comparator can be used in any of the following ways:

- Associated with a SAC.
- Associated with an ARC.
- As an individual resource.

R_{DCCBY} While a Context Identifier Comparator is associated with either an SAC or an ARC, only while the PE is executing with the Context identifier that the Context Identifier Comparator is programmed with and when an address is accessed which the SAC or ARC is programmed to match on, the SAC or ARC signals a match.

R_{BKQKQ} While a Context Identifier Comparator is used as an individual resource, when an instruction block is executed with the Context identifier that the Context Identifier Comparator is programmed with, the Context Identifier Comparator matches.

I_{PBXRH} When using a Context Identifier Comparator as an independent resource to activate a resource event, the time that the resource event is activated relative to the time that the Context Identifier Comparator becomes active might be imprecise.

I_{RBLYL} It might be possible for multiple matches of a Context Identifier Comparator to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place.

R_{MPJBW} When multiple Context Identifier Comparator matches occur simultaneously for one Context Identifier Comparator, all of the following are true:

- The Context Identifier Comparator signals a match at least once.
- The Context Identifier Comparator does not signal more matches than the number of instructions that are executed with the Context identifier that the Context Identifier Comparator is programmed with.

I_{HDCJK} A Context Identifier Comparator might match on Speculative execution, that is, a Context Identifier Comparator might match if the PE speculatively changes the Context identifier.

R_{MCYYC} When the PE executes instructions in Debug state, Context Identifier Comparators do not match.

R_{SRZGJ} When the PE executes instructions in a Trace Prohibited region, Context Identifier Comparators do not match.

I_{GKDRL} The Context identifier might change at points that are not Context synchronization events, for example when a System instruction is used to write to the current Context identifier register. In these scenarios, the Context Identifier Comparator might compare against the old or new Context identifier value for any instruction after the *P0 element* before the System instruction, up to the Context synchronization event after the System instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

$$m = \begin{cases} \text{TRCCIDCCTLR0.COMP0} & n \equiv 0 \\ \text{TRCCIDCCTLR0.COMP1} & n \equiv 1 \\ \text{TRCCIDCCTLR0.COMP2} & n \equiv 2 \\ \text{TRCCIDCCTLR0.COMP3} & n \equiv 3 \\ \text{TRCCIDCCTLR1.COMP4} & n \equiv 4 \\ \text{TRCCIDCCTLR1.COMP5} & n \equiv 5 \\ \text{TRCCIDCCTLR1.COMP6} & n \equiv 6 \\ \text{TRCCIDCCTLR1.COMP7} & n \equiv 7 \end{cases}$$

$$v = \text{TRCCIDCVR<n>.VALUE}$$

$$\text{cid} = \text{CONTEXTIDR_EL1.PROCID}$$

$$\text{CIDCOMP}[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \prod_{j=0}^{7} \left((v[8j+7:8j] \equiv \text{cid}[8j+7:8j]) \vee \text{m}[j] \right) & \text{Otherwise} \end{cases}$$

D4.6.8 Virtual Context Identifier Comparators

I_{BXVPG} An ETE trace unit provides between zero and eight Virtual Context Identifier Comparators. Each Virtual Context Identifier Comparator can be used in any of the following ways:

- Associated with a SAC.
- Associated with an ARC.
- As an individual resource.

R_{RTRBM} While a Virtual Context Identifier Comparator is associated with either an SAC or an ARC, only while the PE is executing with the Virtual context identifier that the Virtual Context Identifier Comparator is programmed with and when an address is accessed which the SAC or ARC is programmed to match on, the SAC or ARC signals a match.

R_{VWYMY} While a Virtual Context Identifier Comparator is used as an individual resource, when an instruction block is executed with the Virtual context identifier that matches the Virtual Context Identifier Comparator value, the Virtual Context Identifier Comparator matches.

R_{FLXQL} While TRFCR_EL2.CX indicates that Virtual Context Identifier Comparators cannot match, Virtual Context Identifier Comparators do not match.

R_{LPKBR} When the PE executes instructions in Debug state, Virtual Context Identifier Comparators do not match.

R_{WZWLT} When the PE executes instructions in a Trace Prohibited region, Virtual Context Identifier Comparators do not match.

I_{SCPJP} When using a Virtual Context Identifier Comparator as an independent resource to activate a resource event, the time at which the resource event is activated relative to the time at which the Virtual Context Identifier Comparator becomes active might be imprecise.

R_{LJRPW} A Virtual Context Identifier Comparator is associated with an SAC by programming TRCACATR<n>.CONTEXT for the SAC.

I_{GJCRG} It might be possible for multiple matches of a Virtual context identifier to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place.

R_{JNNDL} When multiple Virtual Context Identifier Comparator matches occur simultaneously for one Virtual Context Identifier Comparator, both of the following are true:

- The Virtual Context Identifier Comparator signals a match at least once.
- The Virtual Context Identifier Comparator does not signal more matches than the number of instructions that are executed with the Virtual context identifier that the Virtual Context Identifier Comparator is programmed with.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{NPPCF} A Virtual Context Identifier Comparator might signal a match on Speculative execution, that is, a Virtual Context Identifier Comparator might signal a match when the PE speculatively changes the Virtual context identifier.

I_{PPWXT} The Virtual context identifier might change at points which are not Context synchronization events, for example when a System instruction is used to write to CONTEXTIDR_EL2. In these scenarios, the Virtual Context Identifier Comparator might compare against the old or new Virtual context identifier value for any instruction after the *P0 element* before the System instruction, up to the Context synchronization event after the System instruction.

$$m = \begin{cases} \text{TRCVMIDCCTLR0.COMP0} & n \equiv 0 \\ \text{TRCVMIDCCTLR0.COMP1} & n \equiv 1 \\ \text{TRCVMIDCCTLR0.COMP2} & n \equiv 2 \\ \text{TRCVMIDCCTLR0.COMP3} & n \equiv 3 \\ \text{TRCVMIDCCTLR1.COMP4} & n \equiv 4 \\ \text{TRCVMIDCCTLR1.COMP5} & n \equiv 5 \\ \text{TRCVMIDCCTLR1.COMP6} & n \equiv 6 \\ \text{TRCVMIDCCTLR1.COMP7} & n \equiv 7 \end{cases}$$

$$\text{VMID} = \text{CONTEXTIDR_EL2.PROCID}$$

$$v = \text{TRCVMIDCVR<n>.VALUE}$$

$$\text{VMIDCOMP}[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \prod_{j=0}^{7} \Big((v[8j+7:8j] \equiv \text{VMID}[8j+7:8j]) \vee \text{m}[j] \Big) & \text{Otherwise} \end{cases}$$

D4.6.9 Counters

I_{NCCBM} The Counters that are employed by the ETE architecture are all decrement counters.

The ETE architecture enables a trace unit to connect Counter outputs to resource events, so that a Counter at zero state can be used as a resource to activate a resource event. For example, a Counter at zero state might be used to assert an External Output or to make ViewInst active.

An ETE trace unit provides up to four 16-bit Counters. TRCIDR5.NUMCNTR indicates how many Counters are implemented. For each Counter, the following can be specified:

- The initial counter value. This can be programmed using TRCCNTVR<n>.
- The reload value. This can be programmed using TRCCNTRLDVR<n>.
- The resource event that causes the Counter to reload with the reload value. This resource event is called RLDEVENT. In addition, if required, the Counter can be programmed so that it automatically reloads whenever it reaches zero.
- The resource event that enables the Counter to decrement. This resource event is called CNTEVENT. The Counter decrements whenever CNTEVENT is active.

R_{RBMQM} The processor clock clocks the Counters in the trace unit.

R_{PZQGV} While the PE is stalled, the Counters continue to count.

R_{FHFMP} While the resources are in the Paused state, the Counters do not count.

R_{LFVYH} When a Counter value is changed by anything other than a direct write to TRCCNTVR<n>, the trace unit considers the change to be an indirect write to TRCCNTVR<n>.VALUE.

I_{MLDXC} Each Counter operates in one of the two following possible modes:

- Normal mode.
- Self-reload mode.

R_{SBQPN} While the Counter is in Normal mode, when the Counter reaches zero, the Counter remains at zero until the reload resource event, RLDEVENT, occurs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HYLGG} While the Counter is in Normal mode, the Counter-at-zero resource is active for the whole of the time that the Counter is at zero.

R_{YLYPH} While the Counter is in Self-reload mode, when the Counter reaches zero, when the decrement resource event is next active, the trace unit reloads the Counter with the reload value.

R_{VGJNL} While the Counter is in Self-reload mode, when the Counter value is zero, the decrement resource event is active and the reload resource event is not active, the Counter-at-zero resource is active for one cycle.

I_{KTRXV} The following examples show various operating scenarios for a single Counter. Each Counter is programmed with a reload value of `0x3`.

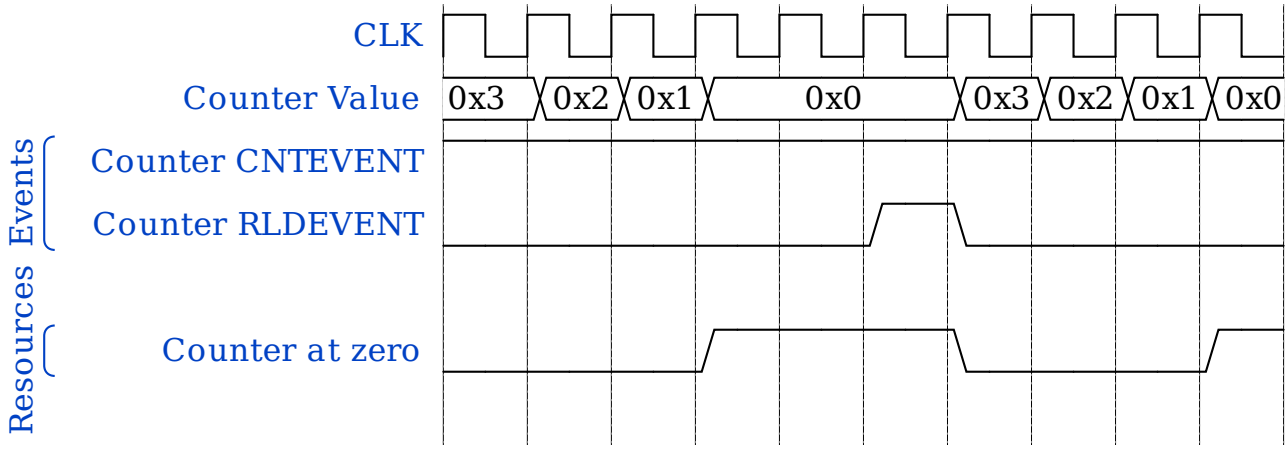

Figure D4-22 Counter Example 1, Normal mode

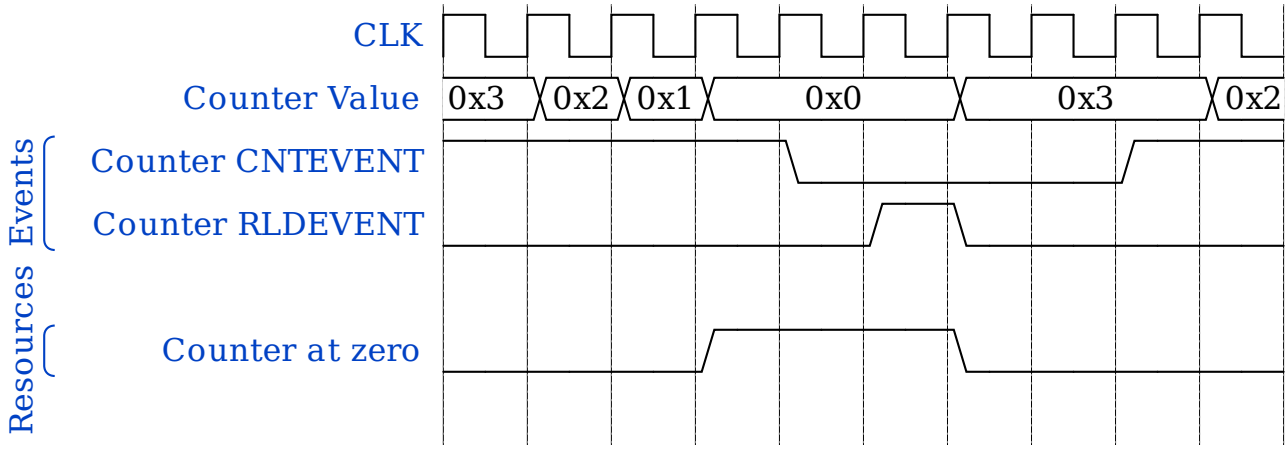

Figure D4-23 Counter Example 2, Normal mode

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

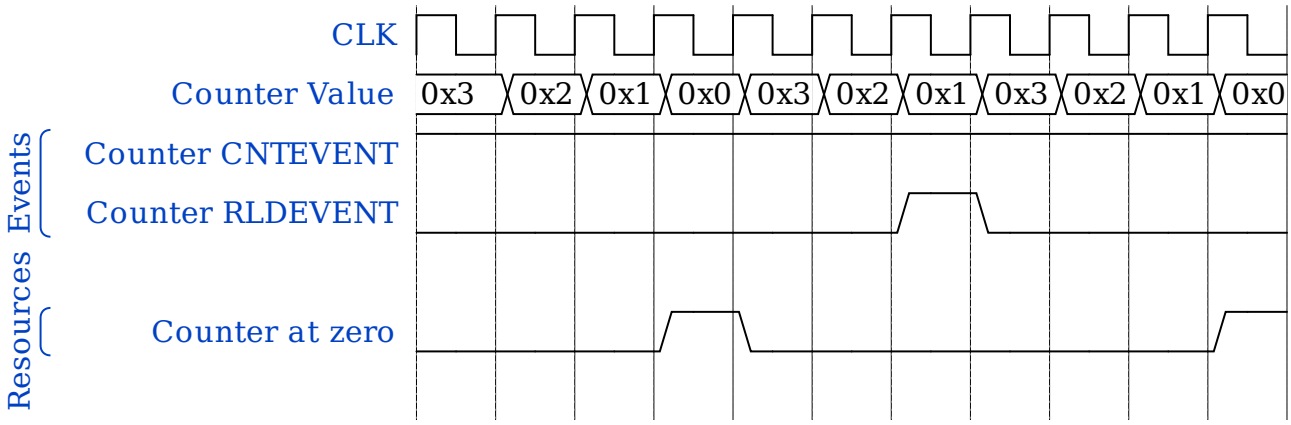

Figure D4-24 Counter Example 3, Self-reload mode

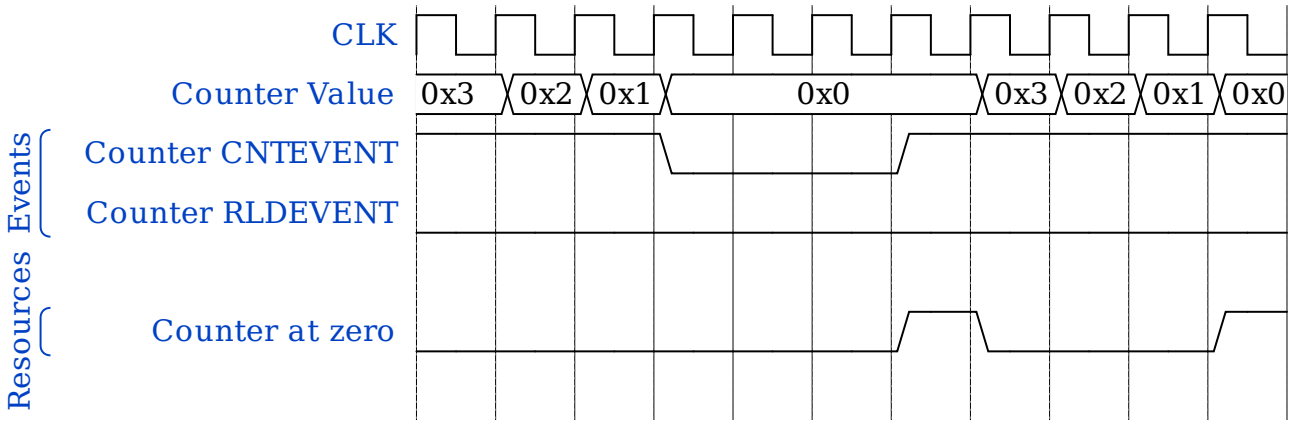

Figure D4-25 Counter Example 4, Self-reload mode

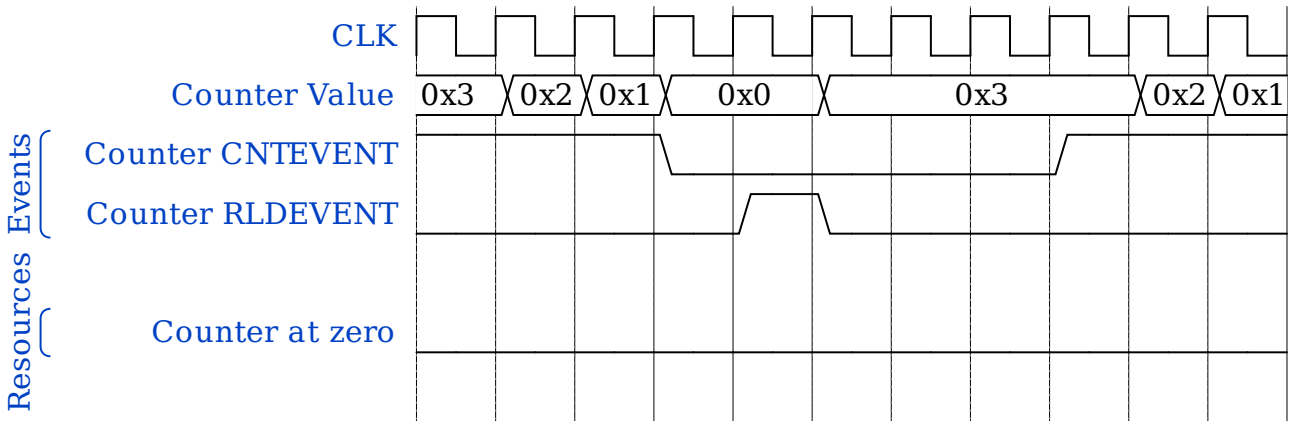

Figure D4-26 Counter Example 5, Self-reload mode

R_{KXLKC} While the decrement resource event is inactive, the Counters do not decrement.

R_{DDCDK} The trace unit prioritizes the reload resource event over the count decrement resource event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6.9.1 Counter Operation in Normal mode

IBRLYH This table describes the counter operation in Normal mode.

RLDEVENT	dec_action	Counter value	Action	Resource Active	Notes
Inactive	x	0	Stable	Yes	Resource is active while Counter is at zero and remains at zero
Inactive	0	> 0	Stable	No	No activity
Inactive	1	> 0	Decrement	No	Decrement when not zero
Active	x	0	Reload	Yes	Reload, but resource is active because Counter is at zero
Active	x	> 0	Reload	No	Reload

D4.6.9.2 Counter Operation in Self-reload mode

ISHWBT This table describes the counter operation in Self-reload mode:

RLDEVENT	dec_action	Counter value	Action	Resource Active	Notes
Inactive	0	x	Stable	No	No activity, resource is not active even if the Counter is at zero
Inactive	1	0	Reload	Yes	Reload because dec_action is active and the Counter is at zero, resource is active only in this cycle
Inactive	1	> 0	Decrement	No	Decrement when not zero
Active	x	x	Reload	No	Reload regardless of decrement action and the value of the Counter, resource is never active

D4.6.9.3 Forming a larger Counter from two separate Counters

ITYLSH Some Counters can be chained together to form a larger counter, so that every time one Counter reloads, another Counter decrements.

IMMDRW The following example shows an operating scenario for 2 Counters chained together. Counter 0 is programmed with a reload value of `0x2`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

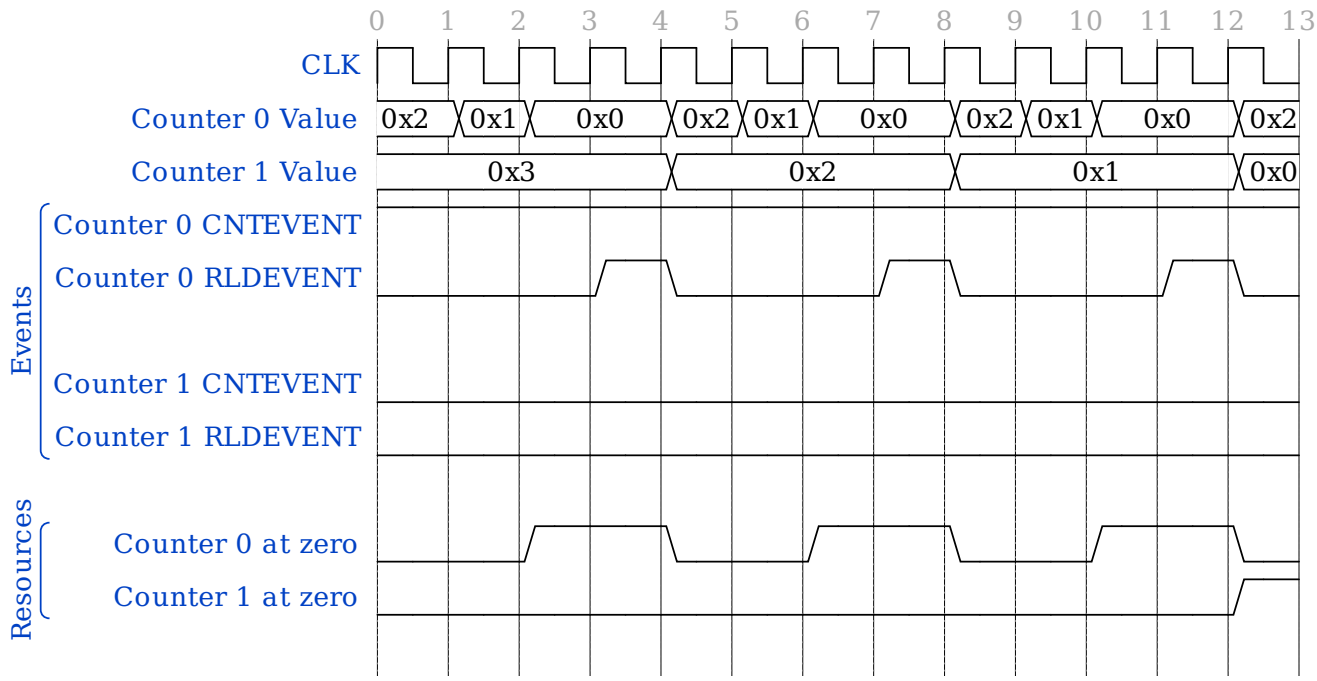

Figure D4-27 Chained Counter Example 1

R_{WPWQD} Only certain Counters can be programmed to be chained together, as follows:

- Counter 1 can be programmed to decrement when Counter 0 reloads.
- Counter 3 can be programmed to decrement when Counter 2 reloads.

R_{QHZFW} The decrement resource event for the higher Counter n is active when either of the following occurs:

- The lower Counter reloads due to one of the following:
  - The reload resource event that is selected by TRCCNTCTLR<n-1>.RLDEVENT.
  - The Self-reload mechanism that is controlled by TRCCNTCTLR<n-1>.RLDSELF.
- The decrement resource event that is selected by TRCCNTCTLR<n>.CNTEVENT is active.

R_{BDPDN} While two Counters are chained together to form a larger counter, the larger counter appears as a 32-bit counter without any tearing of the values between the two Counters.

I_{FTDHL} For example, if Counter 0 is in Self-reload mode and has a value of `0x0000` and a reload value of `0xFFFF`, and Counter 1 is in Normal mode and has a value of `0x1234`, then when a decrement resource event occurs on Counter 0, Counter 0 reloads to `0xFFFF`. The reload of Counter 0 causes Counter 1 to decrement, resulting in a value of `0x1233`. Therefore the sequence on the Counters on consecutive cycles is `0x1234_0000` and `0x1233_FFFF`.

I_{BCMGM} For Counters 1 and 3, TRCCNTCTLR<n>.CNTCHAIN is an RW field that determines whether the Counter is chained. For Counters 0 and 2, TRCCNTCTLR<n>.CNTCHAIN is RES0.

Note

The CounterAtZero resource might not be asserted at the same time that the Counter is at zero. For example, this could happen if the trace unit implementation pipelines some logic.

I_{KJSDV} The CounterAtZero resource might not be asserted at the same time that the Counter is at zero. For example, this could happen if the trace unit implementation pipelines some logic.

D4.6.9.4 Pseudocode

D4.6.9.4.1 EvalAllCounters()

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// The counter-at-zero resources
var CounterAtZero : array[[4]] of boolean;

//
// EvalAllCounters() is called each clock cycle
//

func EvalAllCounters()
begin
    var reload : array[[4]] of boolean;
    for i = 0 to UInt(TRCIDR5().NUMCNTR) - 1 do
        if (i == 1) || (i == 3) then
            reload[[i]] = EvalCounter(i, reload[[i-1]]);
        else
            reload[[i]] = EvalCounter(i, FALSE);
        end;
    end;
    return;
end;

//
// EvalCounter() is called for each counter
//

func EvalCounter(index : integer, lower_reloads : boolean) => boolean
begin
    var dec_action : boolean;
    var resource_active : boolean;
    var reload : boolean;
    var decrement : boolean;

    // A dec_action signal is constructed which indicates whether the counter
    // decrements. This is based on TRCCNTCTLR(n).CNTEVENT and, for counters
    // which support chaining, on TRCCNTCTLR(n).CNTCHAIN and on whether or not
    // the lower counter is reloading.
    dec_action = (IsEventActive(TRCCNTCTLR(index).CNTEVENT) ||
                  ((TRCCNTCTLR(index).CNTCHAIN == '1') && lower_reloads));

    // The counter-at-zero resource is active if the counter is
    // currently at zero and is either in Normal mode or in
    // Self-Reload mode and dec_action is active and the reload
    // event is not active.
    resource_active = ((TRCCNTVR(index).VALUE == Zeros{16}) &&
                       ((TRCCNTCTLR(index).RLDSELF=='0') ||
                        (dec_action && !IsEventActive(TRCCNTCTLR(index).RLDEVENT))));

    // The counter reloads if the reload event is active or the self-reload
    // mechanism causes a reload.
    reload = (IsEventActive(TRCCNTCTLR(index).RLDEVENT) ||
              ((TRCCNTCTLR(index).RLDSELF == '1') &&
                dec_action && TRCCNTVR(index).VALUE == Zeros{16}));

    // The counter only decrements if it is nonzero and does not reload and
    // dec_action is active.
    decrement = !reload && (TRCCNTVR(index).VALUE != Zeros{16}) && dec_action;

    // Determine the next value of the counter
    if reload then
        TRCCNTVR(index).VALUE = TRCCNTRLDVR(index).VALUE;
    elsif decrement then
        TRCCNTVR(index).VALUE = (UInt(TRCCNTVR(index).VALUE) - 1)[15:0];
    else
        TRCCNTVR(index).VALUE = TRCCNTVR(index).VALUE;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CounterAtZero[[index]] = resource_active;
    return reload;
end;
```

D4.6.10 Sequencer

I_{BGGRG} An ETE trace unit can contain a Sequencer state machine that has four states.

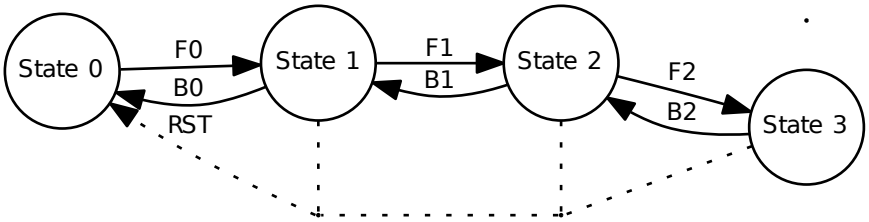

Figure D4-28 Sequencer state machine

I_{PTVBH} TRCIDR5.NUMSEQSTATE indicates whether the state machine is implemented.

R_{QYNJH} If the Sequencer state machine is implemented, it has four states, numbered 0 to 3.

R_{TQWHX} When the trace unit leaves the Disabled state, the Sequencer state machine starts in the state programmed in TRCSEQSTR.STATE.

I_{SYCBV} The Sequencer can be connected to resource events, so that the Sequencer transitions from one state to another when certain resource events occur. The TRCSEQEVR<n> registers can be used to specify which resource events cause the state machine to transition.

Each register can be used to specify the following:

- A resource event that causes the state machine to progress to the next state.
- A resource event that causes the state machine to transition backwards to the previous state.

Different resource events can be chosen to cause the Sequencer to transition between different states. For example, a particular resource event might cause an F0 transition from state 0 to state 1 on one processor clock cycle, whereas a different resource event might cause an F1 transition from state 1 to state 2 on the next processor clock cycle. A third independent resource event might cause a B1 transition backwards from state 2 to state 1 on the third clock cycle.

R_{NPVRQ} The trace unit prioritizes forward transitions over backward transitions in the Sequencer state machine. That is, when two resource events occur that result in a forward transition conflicting with a backward transition in the same processor clock cycle, the trace unit gives priority to the forward transition and ignores the backward transition.

I_{QNFJZ} The Sequencer can progress through multiple states in a single processor clock cycle. For example, if the Sequencer is in state 0 and the resource events that cause an F0 and F1 transition to take place both become active in one clock cycle, then the Sequencer progresses from state 0 to state 2.

I_{DMZGJ} The Sequencer can be reset to state 0 from any other state. TRCSEQRSTEVR can be used to specify a resource event to reset the Sequencer.

R_{HQBBF} When a resource event that causes an RST transition occurs, the Sequencer finishes the clock cycle in state 0 and does not progress to another state until the next clock cycle.

R_{KVSXC} The trace unit prioritizes RST transitions over all other transitions. That is, when a resource event that causes an RST transition is active in the same clock cycle as resource events that cause other transitions, the trace unit gives priority to the RST transition and ignores all other transitions.

R_{JDPYL} The following table defines all of the possible state transitions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	To			
From	0	1	2	3
0	RST | !F0	F0 & !F1	F0 & F1 & !F2	F0 & F1 & F2
1	RST | (B0 & !F1 & !F0)	(!B0 | F0) & !F1	F1 & !F2	F1 & F2
2	RST | (B0 & B1 & !F2 & !F1 & !F0)	B1 & (!B0 | F0) & !F1 & !F2	(!B1 | F1) & !F2	F2
3	RST | (B0 & B1 & B2 & !F2 & !F1 & !F0)	B2 & B1 & (!B0 | F0) & !F2 & !F1	B2 & (!B1 | F1) & !F2	!B2 | F2

I_{YQZGV} If multiple resource events that cause transitions become active in one processor clock cycle, there is no guarantee that the order of these resource events becoming active is observed. For example, you might program:

- F0 to be active on an instruction Address Comparator at address `0x1000`.
- F1 to be active on an instruction Address Comparator at address `0x1004`.

If the instruction at `0x1000` and the instruction at `0x1004` are executed in the same processor clock cycle, then the transition from state 0 to state 2 occurs regardless of the program order of the two instructions.

R_{VDTDP} When the Sequencer state is changed by anything other than a direct write to TRCSEQSTR, the trace unit considers the change to be an indirect write to TRCSEQSTR.STATE.

I_{WYFZH} The ETE architecture provides each Sequencer state as a resource, so that states can be used to trigger other resource events in the trace unit.

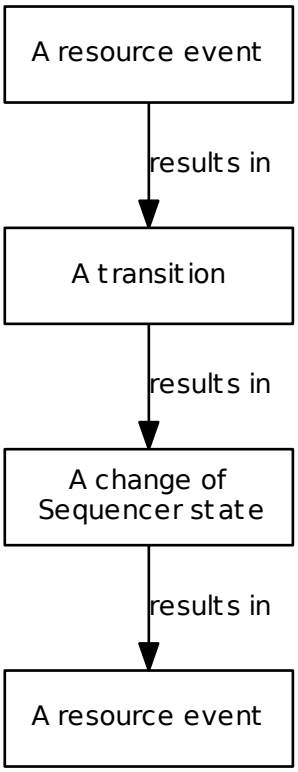

Figure D4-29 Sequencer operation

R_{HQHFT} When the Sequencer progresses through multiple states in a single processor clock cycle, for each state that it passes through, the resource state that the Sequencer triggers is active for that cycle.

I_{DCFMF} For example, if the Sequencer is in state 0, and in one processor clock cycle it moves to state 3, then the resource events that state 1 and state 2 are connected to must be active for that clock cycle. The same rule applies if the Sequencer is transitioning backwards, so that if it is in state 3, and in one processor clock cycle B2 and B1 cause it move to state 1, then the resource event that state 2 is connected to must be active for that clock cycle.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The exception to this is when an RST transition causes the Sequencer to return to state 0. For example, if the Sequencer is in state 3, and in one processor clock cycle it moves to state 0, then the resource events that state 2 and stage 1 are connected to must not become active.

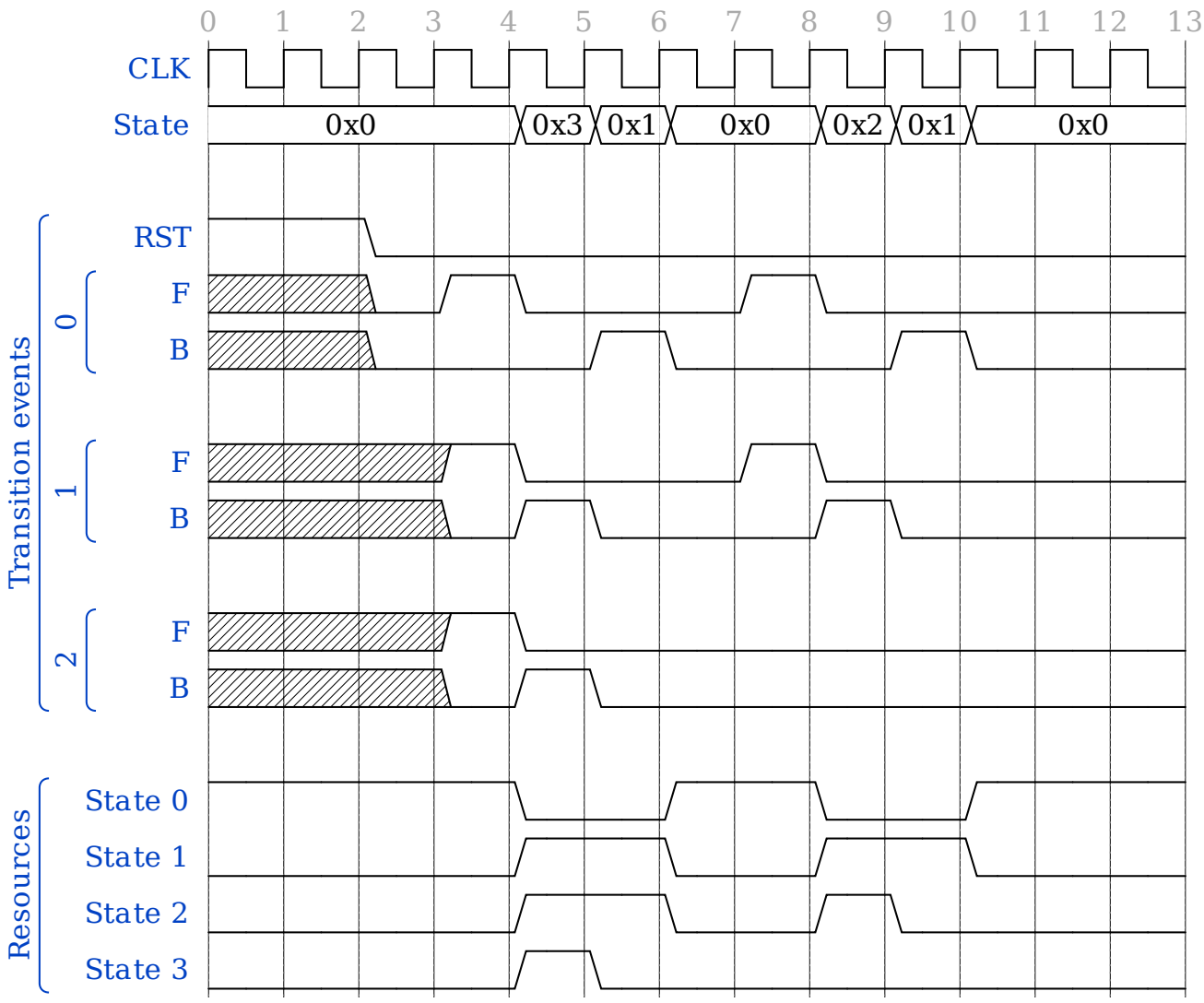

Figure D4-30 Example of State Transitions

D4.6.10.1 Pseudocode

D4.6.10.1.1 EvalSequencer()

```
// The sequencer state resources
var SequencerState : array[[4]] of boolean;

// EvalSequencer()
// ===============

func EvalSequencer()
begin
    var rst : boolean;
    var txn0 : array [[4]] of boolean;
    var txn1 : array [[4]] of boolean;
    var txn2 : array [[4]] of boolean;
    var txn3 : array [[4]] of boolean;

    (rst, txn0, txn1, txn2, txn3) = SequencerTransitions();
    // Sequencer State resources
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SequencerState[[0]] = FALSE;
    SequencerState[[1]] = FALSE;
    SequencerState[[2]] = FALSE;
    SequencerState[[3]] = FALSE;

    SequencerResource(rst, txn0, txn1, txn2, txn3);
    SequencerNextState(rst, txn0, txn1, txn2, txn3);
end;
```

D4.6.10.1.2 SequencerTransitions()

```
// SequencerTransitions()
// ======================

func SequencerTransitions() => ( boolean,
                                 array [[4]] of boolean,
                                 array [[4]] of boolean,
                                 array [[4]] of boolean,
                                 array [[4]] of boolean)
begin
    let F0 : boolean = IsEventActive(TRCSEQEVR0().F_SEL, TRCSEQEVR0().F_TYPE);
    let B0 : boolean = IsEventActive(TRCSEQEVR0().B_SEL, TRCSEQEVR0().B_TYPE);

    let F1 : boolean = IsEventActive(TRCSEQEVR1().F_SEL, TRCSEQEVR1().F_TYPE);
    let B1 : boolean = IsEventActive(TRCSEQEVR1().B_SEL, TRCSEQEVR1().B_TYPE);

    let F2 : boolean = IsEventActive(TRCSEQEVR2().F_SEL, TRCSEQEVR2().F_TYPE);
    let B2 : boolean = IsEventActive(TRCSEQEVR2().B_SEL, TRCSEQEVR2().B_TYPE);

    let rst : boolean = IsEventActive(TRCSEQRSTEVR().RST_SEL, TRCSEQRSTEVR().RST_TYPE);

    var txn0 : array [[4]] of boolean;
    var txn1 : array [[4]] of boolean;
    var txn2 : array [[4]] of boolean;
    var txn3 : array [[4]] of boolean;

    txn0[[1]] = F0 && !F1;
    txn0[[2]] = F0 &&  F1 && !F2;
    txn0[[3]] = F0 &&  F1 &&  F2;

    txn1[[0]] =   B0 && !F0  && !F1;
    txn1[[1]] = (!B0 ||  F0) && !F1;
    txn1[[2]] =                 F1 && !F2;
    txn1[[3]] =                 F1 &&  F2;

    txn2[[0]] =   B0 && !F0  &&  B1 && !F1  && !F2;
    txn2[[1]] = (!B0 ||  F0) &&  B1 && !F1  && !F2;
    txn2[[2]] =               (!B1 ||  F1) && !F2;
    txn2[[3]] =                                F2;

    txn3[[0]] =   B0 && !F0  &&  B1 && !F1  && B2 && !F2;
    txn3[[1]] = (!B0 ||  F0) &&  B1 && !F1  && B2 && !F2;
    txn3[[2]] =               (!B1 ||  F1) && B2 && !F2;
    txn3[[3]] =                            (!B2 ||  F2);

    return (rst, txn0, txn1, txn2, txn3);
end;
```

D4.6.10.1.3 SequencerResource()

```
// SequencerResource()
// ===================

func SequencerResource(rst : boolean,
                       txn0 : array [[4]] of boolean,
                       txn1 : array [[4]] of boolean,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        txn2 : array [[4]] of boolean,
                        txn3 : array [[4]] of boolean)
begin
    case TRCSEQSTR().STATE of
        when '00' => SequencerState[[0]] = TRUE;
        when '01' => SequencerState[[1]] = TRUE;
        when '10' => SequencerState[[2]] = TRUE;
        when '11' => SequencerState[[3]] = TRUE;
    end;

    // If the statemachine passes through several states in one iteration then
    // set the SequencerState as appropriate.
    if !rst then
        case TRCSEQSTR().STATE of
            when '00' =>
                if txn0[[2]] then
                    SequencerState[[1]] = TRUE;
                end;
                if txn0[[3]] then
                    SequencerState[[1]] = TRUE;
                    SequencerState[[2]] = TRUE;
                end;
            when '01' =>
                if txn1[[3]] then
                    SequencerState[[1]] = TRUE;
                    SequencerState[[2]] = TRUE;
                end;
            when '10' =>
                if txn2[[0]] then
                    SequencerState[[1]] = TRUE;
                end;
            when '11' =>
                if txn3[[0]] then
                    SequencerState[[1]] = TRUE;
                    SequencerState[[2]] = TRUE;
                end;
                if txn3[[1]] then
                    SequencerState[[2]] = TRUE;
                end;
        end;
    end;
end;
```

D4.6.10.1.4 SequencerNextState()

```
// SequencerNextState()
// ====================

func SequencerNextState(rst : boolean,
                        txn0 : array [[4]] of boolean,
                        txn1 : array [[4]] of boolean,
                        txn2 : array [[4]] of boolean,
                        txn3 : array [[4]] of boolean)
begin
    if rst then
        TRCSEQSTR().STATE = '00';
    else
        case TRCSEQSTR().STATE of
            when '00' =>
                if txn0[[1]] then
                    TRCSEQSTR().STATE = '01';
                end;
                if txn0[[2]] then
                    TRCSEQSTR().STATE = '10';
                end;
                if txn0[[3]] then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    TRCSEQSTR().STATE = '11';
                end;
            when '01' =>
                if txn1[[0]] then
                    TRCSEQSTR().STATE = '00';
                end;
                if txn1[[1]] then
                    TRCSEQSTR().STATE = '01';
                end;
                if txn1[[2]] then
                    TRCSEQSTR().STATE = '10';
                end;
                if txn1[[3]] then
                    TRCSEQSTR().STATE = '11';
                end;
            when '10' =>
                if txn2[[0]] then
                    TRCSEQSTR().STATE = '00';
                end;
                if txn2[[1]] then
                    TRCSEQSTR().STATE = '01';
                end;
                if txn2[[2]] then
                    TRCSEQSTR().STATE = '10';
                end;
                if txn2[[3]] then
                    TRCSEQSTR().STATE = '11';
                end;
            when '11' =>
                if txn3[[0]] then
                    TRCSEQSTR().STATE = '00';
                end;
                if txn3[[1]] then
                    TRCSEQSTR().STATE = '01';
                end;
                if txn3[[2]] then
                    TRCSEQSTR().STATE = '10';
                end;
                if txn3[[3]] then
                    TRCSEQSTR().STATE = '11';
                end;
        end;
    end;
end;
```

D4.6.11 Single-shot Comparator Controls

I_{QSKXC} If a trace unit is exposed to speculative execution when Address Comparators are used to activate resource events in the trace unit, then those resource events might be activated when speculative execution occurs:

- A Single Address Comparator might signal a match on speculative execution.
- An Address Range Comparator might signal a match on speculative execution.

For example, this means that if an Address Comparator is used to activate a Counter or assert an External Output, then that Counter might decrement, or that External Output might become asserted, as a result of speculative execution. The Single-shot Comparator Controls for Address Comparators make it possible for resource events in the trace unit to be activated based only on non-speculative execution, that is, only on architectural execution.

A trace unit can provide up to eight Single-shot Comparator Controls. Each Single-shot Comparator Control can be used with one or more Address Comparators.

I_{TLFLF} Single-shot Comparator Controls can be used as a trace unit resource, to activate trace unit resource events. For example, a Single-shot Comparator Control can be selected to:

- Enable or reload a trace unit Counter.
- Initiate a transition in the trace unit Sequencer state machine.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Assert an External Output.

A Single-shot Comparator Control can therefore, if programmed to assert an External Output, be used to indicate to a trace analyzer that a particular instruction has been resolved for execution. This means that a trace analyzer can start or stop trace capture that is based on the architectural execution of that instruction.

I_{RBMXW} If a trace unit contains one or more Address Comparators, Arm recommends that at least one Single-shot Comparator Control is implemented.

I_{VNBPG} A Single-shot Comparator Control works in the following way:

1. One or more Address Comparators are selected by using the TRCSSCCR<n> for the Single-shot Comparator Control. The selected Address Comparators can be all Single Address Comparators, all Address Range Comparators, or a combination of both.
2. When one of the selected Address Comparators matches, then when the instruction is confirmed to have architecturally executed, the Single-shot Comparator Control fires.

When a selected Address Comparator matches, but the instruction is confirmed to have not architecturally executed, the Single-shot Comparator Control does not fire.

R_{XVVYX} When an instruction which matches an Address Comparator is confirmed to have architecturally executed, and the Address Comparator is selected by TRCSSCCR<n>, and TRCSSCSR<n>.STATUS is 0 or TRCSSCCR<n>.RST is 1, the Single-shot Comparator Control <n> fires.

R_{XFJGB} When a `TSB CSYNC` instruction is executed while a Single-shot Comparator Control is programmed to fire due to the `TSB CSYNC` instruction, only when the related Trace synchronization event has completed, the Single-shot Comparator fires.

R_{SWNFV} When a Single-shot Comparator Control fires, the trace unit considers this an indirect write to set TRCSSCSR<n>.STATUS to 1.

R_{GBDCK} While the resources are in the Paused state, when the conditions for a Single-shot Comparator Control to fire are met:

- If TRCSSCCR<n>.RST is 1 or TRCSSCSR<n>.STATUS is 0 then TRCSSCSR<n>.PENDING is indirectly written to 1.
- If TRCSSCCR<n>.RST is 0 and TRCSSCSR<n>.STATUS is 1 then TRCSSCSR<n>.PENDING is either indirectly written to 1 or is unchanged.

R_{NKKSN} When the trace unit becomes disabled and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control does not fire.

R_{DQZSD} When tracing becomes prohibited and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control waits until the instruction speculation is resolved and fires if the instruction is architecturally executed.

R_{VTWXJ} While a Single-shot Comparator Control is used for instruction address comparisons, when the conditions for the Single-shot Comparator Control to fire are met, the Single-shot Comparator Control fires, regardless of whether either of the following are true:

- The instruction fails its condition code check.
- The instruction is a failed store-exclusive operation.

I_{XZKFW} When a Single-shot Comparator Control is used to activate a resource event, the resource event might not become activated until some time after the trace unit has traced the instruction. This is because although the trace unit traces the instruction it is executed, the PE might not confirm whether the instruction was architecturally executed or canceled because of mis-speculation until some time later, and therefore the Single-shot Comparator Control might not fire until some time later.

D4.6.11.1 Single-shot Comparator Control modes

I_{XZJSV} Each Single-shot Comparator Control operates in one of the following modes:

- Single-shot mode: The Single-shot Comparator Control only fires once. That is, after it has fired, it never fires again.
- Multi-shot mode: The Single-shot Comparator Control resets after each time it fires. That is, it can fire again when a selected Address Comparator next signals an address match for an instruction is architecturally executed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCSSCCR<n>.RST selects the mode.

R_{KJBCH} While a Single-shot Comparator Control is in multi-shot mode, when the Single-shot Comparator Control fires, it fires for a maximum of one processor clock cycle.

R_{SNDBZ} While a Single-shot Comparator Control is in multi-shot mode, when multiple of the comparators selected for the Single-shot Comparator Control match in close succession, only the first match is guaranteed to cause the Single-shot Comparator Control to fire.

I_{HSGTY} Examples of multiple comparator matches in close succession include:

- More than one of the Address Comparators that are selected signal an address match simultaneously.
- One Address Comparator matches multiple times while a first match is still waiting to be resolved.

D4.6.11.2 Operation while in Paused state

I_{SPQLS} The resolution of a speculative instruction might occur after the PE has entered a Trace Prohibited region and the resources have entered the Paused state. If the conditions for the Single-shot Comparator Control to fire are met while the resources are in the Paused state, then the Single-shot Comparator Control resource event is delayed to ensure that the Single-shot Comparator Control resource event is seen.

R_{TQHNK} While the resources are in the Paused state, the Single-shot Comparator Controls do not fire.

R_{PVRGR} When the resources enter the Running state while TRCSSCSR<n>.PENDING is 1, the following occur:

- If TRCSSCCR<n>.RST is 1 or TRCSSCSR<n>.STATUS is 0, the Single-shot Comparator Control fires.
- TRCSSCSR<n>.PENDING is indirectly written to 0.

I_{DMGDY} Some implementations might have no scenarios where TRCSSCSR<n>.PENDING can be set to 1 by an indirect write by the trace unit, in particular if the trace unit is not exposed to any speculative execution. However all implementations support software setting TRCSSCSR<n>.PENDING to 1, and the Single-shot Comparator fires when the trace unit enters the Running state. This behavior ensures the state from one trace unit can be migrated to a different trace unit.

D4.6.12 External Outputs

I_{BZHDF} The ETE architecture supports between zero and four External Outputs. The number of outputs that a trace unit has is IMPLEMENTATION DEFINED, and Arm recommends that at least one output is implemented.

I_{QPQFJ} External Outputs are used to indicate *ETEEvents* to a trace analyzer.

ETEEvents are controlled by resources events. For example, an instruction Address Comparator can be used to drive one of the resource events.

If an External Output is programmed to be asserted based on program execution, such as an Address Comparator, the External Output might not be asserted at the same time as any trace generated by that program execution is output by the trace unit.

I_{PNVWQ} Typically, the generated trace might be buffered in a trace unit which means that the External Output would be asserted before the trace is output.

S_{MFTNW} To program an External Output, use TRCEVENTCTL0R to select a resource.

S_{RBKWB} The TRCIDR0.NUMEVENT field shows how many *ETEEvents* are supported for the particular implementation.

I_{RFLGF} The External Outputs are connected to the Cross Trigger Interface (CTI) for the PE. See The Embedded Cross-Trigger Interface.

R_{RFGJD} In a PE where the Trace Unit reset is independent of the PE Warm reset and the CTI reset is independent of the PE Warm reset, transmission of External Outputs to the CTI is unaffected by a PE Warm reset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6.12.1 Operation while in Paused state

I_{NXCSB} While the resources are in the Paused state an *ETEEvent* might occur, but any associated trace packets might not be generated. TRCRSR.EVENT provides a mechanism for recording this occurrence so that the trace unit state can be saved and restored.

R_{BCMYM} While the resources are in the Paused state, the *ETEEvent* selector retains that one or more *ETEEvents* have been generated but not traced.

R_{SNKFL} When an *ETEEvent* has been generated and the associated External Output has been asserted, any associated Event packets are generated.

R_{FVCMB} When an *ETEEvent* has been generated but the associated External Output has not been asserted, any associated Event packets are not generated.

R_{GYWLS} When an *ETEEvent* occurs while the resources are in the Paused state and the Event packet is not output, the trace unit sets the associated TRCRSR.EVENT[n] to 1.

R_{DCLHJ} When an *ETEEvent* occurs while the resources are in the Paused state, this is considered an indirect write to TRCRSR.

R_{SWBRL} When the trace unit enters the Running state while TRCRSR.EVENT[n] is 1, the associated *ETEEvent* resource is active for a single PE clock cycle, and the trace unit clears TRCRSR.EVENT[n] to 0 and considers the action an indirect write to TRCRSR.

I_{KZYKM} When the trace unit enters the Running state while TRCRSR.EVENT[n] is 1, the resource event selected by TRCEVENTCTL0R.EVENT<n> might also be active on the same PE clock cycle. If this happens, the associated *ETEEvent* resource is active for the single PE clock cycle and might not generate 2 separate *ETEEvents* for these 2 causes of the *ETEEvent*.

D4.6.13 External inputs

I_{TPPSC} The trace unit uses the PMU events as External Inputs.

R_{MTGKB} If a PMU event number that is selected is not implemented, then the External Input resource event is inactive.

R_{YCNCR} Unless otherwise specified by the PMU event, the following events are selectable by the trace unit:

- If FEAT_PMUv3 is implemented, all Common architectural and microarchitectural events implemented by the PMU.
- If FEAT_PMUv3 is not implemented, all PMU events required by the Performance Monitors Extension. See Required events.

  Note

  This includes all events required by other implemented extensions.

- PMU_HOVFS and PMU_OVFS.
- If FEAT_TRBE is implemented, TRB_TRIG.

I_{XJBHV} Additional ASIC-specific events can be selected by using a number in the IMPLEMENTATION DEFINED region.

I_{VWHTZ} There is no requirement that all IMPLEMENTATION DEFINED events are visible by the trace unit, PMU counters, and the `PMUEVENT bus`.

R_{PHDQT} For ETE, export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.

R_{RFWZB} When self-hosted trace is enabled and tracing is prohibited, only the PMU events defined by rules R_{VBCBZ} and R_{KRSMY} are exported to the trace unit.

R_{WSXTC} When self-hosted trace is disabled and counting in the current Security state of the PE is prohibited, only the PMU events defined by rules R_{VBCBZ} and R_{KRSMY} are exported to the trace unit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{VBCBZ} The following PMU events are always exported to the trace unit:

- CTI_TRIGOUT4.
- CTI_TRIGOUT5.
- CTI_TRIGOUT6.
- CTI_TRIGOUT7.
- PMU_OVFS.
- TRB_TRIG.
- TRB_WRAP.

R_{KRSMY} The following PMU events are always exported to the trace unit, unless self-hosted trace is enabled and TRFCR_EL2.E2TRE is 0:

- PMU_HOVFS.

R_{QPDHK} When multiple occurrences of the same PMU event occur during the same cycle, the trace unit only observes a single occurrence of the PMU event.

I_{MHHNV} The operation of the PMU events and the generation of trace within the trace unit are not tightly coupled, and there is no guarantee that enabling ViewInst due to a PMU event will cause the instruction that caused the PMU event to be traced.

R_{XGMPN} When the PMU event SW_INCR is selected as an External Input and PMSWINC_EL0 is written from EL2 or EL3, the External Input is asserted if any bit [n] written has the value 1 and the relevant PMU counter <n> is implemented.

R_{BXPZK} When the PMU event SW_INCR is selected as an External Input and PMSWINC_EL0 is written from EL1 or EL0, the External Input is asserted if any bit [n] written has the value 1 and the relevant PMU counter <n> is implemented and any of the following are true:

- EL2 is implemented and enabled in the current Security state and <n> is less than MDCR_EL2.HPMN.
- EL2 is implemented and disabled in the current Security state.
- EL2 is not implemented.

R_{TTPPY} In a PE where the trace unit reset is independent of the PE Warm reset and the CTI reset is independent of the PE Warm reset, transmission of the CTI_TRIGOUTn events from the CTI to the trace unit is unaffected by a PE Warm reset.

D4.6.13.1 Operation while in Paused state

I_{HZLDV} The External Input Selectors are guaranteed to be active while in the Paused state. This is so that while the resources are Paused any cross trigger event is not lost but will occur when the resources resume running.

TRCRSR.EXTIN provides a mechanism to capture the sticky state of the External Input Selectors while in the Paused state so that the ETE state can be saved and restored.

I_{ZQFND} When one or more selected External Inputs have been asserted, while the resources are in the Paused state, the trace unit retains the knowledge that one or more selected External Inputs have been asserted.

R_{KCXLF} While the resources are in the Pausing or Paused states and the trace unit is not disabled and is not in the low-power state, when an External Input Selector n detects the selected External Input is asserted, the trace unit performs an indirect write to set TRCRSR.EXTIN[n] to 1.

R_{QWYSK} When the resources enter the Running state while TRCRSR.EXTIN[n] is 1, the External Input Selector resource is active for a single PE clock cycle, and the trace unit clears TRCRSR.EXTIN[n] and considers the action an indirect write to TRCRSR.

D4.6.13.2 Operation while in low-power state

R_{KVFVS} While the trace unit is in the low-power state, the External Input Selectors are inactive.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6.14 PE Comparator Inputs

I_{CXBPR} The ETE architecture provides up to eight PE Comparator Inputs, that is, inputs that can be driven from comparators within the PE. For example, a PE might contain IMPLEMENTATION DEFINED comparators.

R_{CNVSS} The number of PE Comparator Inputs is indicated by TRCIDR4.NUMPC.

R_{BDWHM} While the PE is executing in a Trace Prohibited region, the PE Comparator Inputs are inactive.

I_{DDXFB} Each PE Comparator Input can be used in any of the following ways:

- To control the ViewInst start/stop function.
- To control the Single-shot Comparator Controls.
- As an independent resource.

I_{SKDCW} The timing of the effects of the PE Comparator Inputs on the resources and the filtering of the trace unit is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D5
ETE Protocol Descriptions

This chapter describes the ETE packets. It contains the following sections:

- About the ETE protocol.
- Summary list of ETE packets.
- Alphabetical list of ETE packets.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.1 About the ETE protocol

IXCCWQ An ETE trace unit generates a trace byte stream. The protocol is a byte-based packet protocol, which means that the trace byte stream is constructed of multiple packets, where each packet contains one or more bytes of data.

RBVTNX A packet consists of a single header byte, followed by zero or more payload bytes.

D5.1.1 Encoding schemes

D5.1.1.1 Field encodings

ITGRRZ **Bit Replacement:**

The packet outputs bits which update a piece of state. Bits output by the packet replace only those bits in the piece of state. Bits not output by the packet remain unchanged in the piece of state.

INKPMZ **Unsigned LE128n:**

The data is encoded as an unsigned number. The least significant bits of the number are output in the least significant bits of the packet. Bits not output by the packet are `0`.

IWYBBG **POD:**

The encoding is specific to the packet.

IQXHHT **Unary code:**

The sequence for this variable is one of the following:

- A 0.
- A number of 1s followed by a 0.
- All 1s for the size of the variable, as defined by the packet.

For example, the permitted values for a 4-bit variable are:

- `0b0`.
- `0b10`.
- `0b110`.
- `0b1110`.
- `0b1111`.

D5.1.1.2 Instruction set encoding

RFXNDF For any virtual instruction address, the instruction set is output as a combination of the following two pieces of information:

- The SF bit encoded in Context packets.
- The sub_isa encoded by the type of the following groups of packets:
  - Target Address packets.
  - Exception packets.
  - Q packets.
  - Source Address packets.

The sub_isa indicates either:

- IS0.
- IS1.

Table D5-1 indicates how the combination of the SF bit and sub_isa indicate the instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D5-1 Instruction set encodings

SF Bit	sub_isa	Instruction Set
`0b0`	IS0	AArch32 A32
`0b0`	IS1	AArch32 T32
`0b1`	IS0	AArch64 A64

IWKMNL The sub_isa also indicates the alignment of the virtual instruction addresses. Table D5-2 indicates the alignment of each sub_isa.

Table D5-2 Virtual instruction address alignment

sub_isa	Alignment
IS0	Word-aligned
IS1	Halfword-aligned

INSZMB The following packets encode the sub_isa:

- Exception Short Address IS0 Packet.
- Exception Short Address IS1 Packet.
- Exception 32-bit Address IS0 Packet.
- Exception 32-bit Address IS1 Packet.
- Exception 64-bit Address IS0 Packet.
- Exception 64-bit Address IS1 Packet.
- Exception 32-bit Address IS0 with Context Packet.
- Exception 32-bit Address IS1 with Context Packet.
- Exception 64-bit Address IS0 with Context Packet.
- Exception 64-bit Address IS1 with Context Packet.
- Target Address Short IS0 Packet.
- Target Address Short IS1 Packet.
- Target Address 32-bit IS0 Packet.
- Target Address 32-bit IS1 Packet.
- Target Address 64-bit IS0 Packet.
- Target Address 64-bit IS1 Packet.
- Target Address with Context 32-bit IS0 Packet.
- Target Address with Context 32-bit IS1 Packet.
- Target Address with Context 64-bit IS0 Packet.
- Target Address with Context 64-bit IS1 Packet.
- Source Address Short IS0 Packet.
- Source Address Short IS1 Packet.
- Source Address 32-bit IS0 Packet.
- Source Address 32-bit IS1 Packet.
- Source Address 64-bit IS0 Packet.
- Source Address 64-bit IS1 Packet.
- Q short address IS0 Packet.
- Q short address IS1 Packet.
- Q 32-bit address IS0 Packet.
- Q 32-bit address IS1 Packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2 Summary list of ETE packets

Table D5-3 lists the ETE packets ordered by the header byte.

Table D5-3 ETE Packets

Header byte	Name	Purpose
00000000	Alignment Synchronization Packet	Identifies a packet boundary.
00000000	Discard Packet	Indicates a *Discard element*.
00000000	Overflow Packet	Indicates that a trace unit buffer overflow has occurred.
00000001	Trace Info Packet	Resets trace compression to a known architectural state.
0000001x	Timestamp Packet	Indicates a *Timestamp element*.
00000100	Trace On Packet	Indicates that there has been a discontinuity in the trace element stream.
00000110	PE Reset Packet	Indicates that a PE Reset has occurred.
00000110	Exception 32-bit Address IS0 with Context Packet	Indicates that an exception has occurred.
00000110	Exception 32-bit Address IS1 with Context Packet	Indicates that an exception has occurred.
00000110	Exception 64-bit Address IS0 with Context Packet	Indicates that an exception has occurred.
00000110	Exception 64-bit Address IS1 with Context Packet	Indicates that an exception has occurred.
00000110	Exception Exact Match Address Packet	Indicates that an exception has occurred.
00000110	Exception Short Address IS0 Packet	Indicates that an exception has occurred.
00000110	Exception Short Address IS1 Packet	Indicates that an exception has occurred.
00000110	Exception 32-bit Address IS0 Packet	Indicates that an exception has occurred.
00000110	Exception 32-bit Address IS1 Packet	Indicates that an exception has occurred.
00000110	Exception 64-bit Address IS0 Packet	Indicates that an exception has occurred.
00000110	Exception 64-bit Address IS1 Packet	Indicates that an exception has occurred.
00001001	Instrumentation Packet	Indicates an *Instrumentation element*.
00001100	Cycle Count Format 2_0 small commit Packet	Indicates a *Commit element* and a *Cycle Count element*.
00001101	Cycle Count Format 2_1 Packet	Indicates a *Cycle Count element*.
00001101	Cycle Count Format 2_0 large commit Packet	Indicates a *Commit element* and a *Cycle Count element*.
00001110	Cycle Count Format 1_1 with count Packet	Indicates a *Cycle Count element*.
00001110	Cycle Count Format 1_0 with count Packet	Indicates zero or one *Commit elements* followed by a *Cycle Count element*.
00001111	Cycle Count Format 1_1 unknown count Packet	Indicates a *Cycle Count element*.
00001111	Cycle Count Format 1_0 unknown count Packet	Indicates zero or one *Commit elements* followed by a *Cycle Count element* with an UNKNOWN cycle count value.
000100xx	Cycle Count Format 3_1 Packet	Indicates a *Cycle Count element*.
0001xxxx	Cycle Count Format 3_0 Packet	Indicates a *Commit element* and a *Cycle Count element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Header byte	Name	Purpose
00101101	Commit Packet	Indicates a *Commit element*.
0010111x	Cancel Format 1 Packet	Indicates a *Cancel element* optionally followed by a *Mispredict element*.
001100xx	Mispredict Packet	Indicates 0-2 E or N *Atom elements* followed by one *Mispredict element*.
001101xx	Cancel Format 2 Packet	Indicates zero or more E or N *Atom elements* followed by a *Cancel element* and a *Mispredict element*.
00111xxx	Cancel Format 3 Packet	Indicates zero or one E *Atom element* followed by a *Cancel element* with a payload of 2-5 and one *Mispredict element*.
01110000	Ignore Packet	To align packet boundary to memory boundary.
0111xxxx	Event Packet	Indicates 1-4 *Event elements*.
10000000	Context Same Packet	Indicates a *Context element*.
10000001	Context Packet	Indicates a *Context element*.
10000010	Target Address with Context 32-bit IS0 Packet	Indicates a *Target Address element* and a *Context element*.
10000011	Target Address with Context 32-bit IS1 Packet	Indicates a *Target Address element* and a *Context element*.
10000101	Target Address with Context 64-bit IS0 Packet	Indicates a *Target Address element* and a *Context element*.
10000110	Target Address with Context 64-bit IS1 Packet	Indicates a *Target Address element* and a *Context element*.
10001000	Timestamp Marker Packet	Indicates a *Timestamp Marker element*.
100100xx	Target Address Exact Match Packet	Indicates a *Target Address element*.
10010101	Target Address Short IS0 Packet	Indicates a *Target Address element*.
10010110	Target Address Short IS1 Packet	Indicates a *Target Address element*.
10011010	Target Address 32-bit IS0 Packet	Indicates a *Target Address element*.
10011011	Target Address 32-bit IS1 Packet	Indicates a *Target Address element*.
10011101	Target Address 64-bit IS0 Packet	Indicates a *Target Address element*.
10011110	Target Address 64-bit IS1 Packet	Indicates a *Target Address element*.
101000xx	Q with Exact match address Packet	Indicates that some instructions have executed with an address of the next instruction.
10100101	Q short address IS0 Packet	Indicates that some instructions have executed with an address of the next instruction.
10100110	Q short address IS1 Packet	Indicates that some instructions have executed with an address of the next instruction.
10101010	Q 32-bit address IS0 Packet	Indicates that some instructions have executed with an address of the next instruction.
10101011	Q 32-bit address IS1 Packet	Indicates that some instructions have executed with an address of the next instruction.
10101100	Q with count Packet	Indicates that some instructions have executed.
10101111	Q Packet	Indicates that some instructions have executed, without a count of the number of instructions.
101100xx	Source Address Exact Match Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
10110100	Source Address Short IS0 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Header byte	Name	Purpose
`10110101`	Source Address Short IS1 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10110110`	Source Address 32-bit IS0 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10110111`	Source Address 32-bit IS1 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10111000`	Source Address 64-bit IS0 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10111001`	Source Address 64-bit IS1 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`110101xx`	Atom Format 5.2 Packet	Indicates five *Atom elements*.
`110110xx`	Atom Format 2 Packet	Indicates two *Atom elements*.
`110111xx`	Atom Format 4 Packet	Indicates four *Atom elements*.
`11110101`	Atom Format 5.1 Packet	Indicates five *Atom elements*.
`1111011x`	Atom Format 1 Packet	Indicates one *Atom element*.
`11111xxx`	Atom Format 3 Packet	Indicates three *Atom elements*.
`11xxxxxx`	Atom Format 6 Packet	Indicates 3-23 E *Atom elements*, plus a subsequent E Atom or N *Atom element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3 Alphabetical list of ETE packets

This section lists each ETE packet and their description.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.1 Alignment Synchronization Packet

Purpose

Identifies a packet boundary.

Configurations

All.

This packet forms a unique bit and byte pattern. Searching for this pattern allows the trace analyzer to identify packet boundaries.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

Figure D5-1 Alignment Synchronization Packet

Additional information

For more information about the decoding of this packet see Parse_ExtensionPacket().

For more information about the generation of this packet see Trace protocol synchronization.

R_{BXZZJ} Any byte that follows this unique sequence of bits is the header byte of a new packet.

R_{VRKLP} This packet must be output before the first Trace Info packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.2 Atom Format 1 Packet

Purpose

Indicates one *Atom element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	A

Figure D5-2 Atom Format 1 Packet

Field descriptions

A

Indicates a single *Atom element*.

The encoding for this field is POD.

A	Description
`0b0`	One N *Atom element*
`0b1`	One E *Atom element*

Additional information

For more information about the decoding of this packet see AtomFormat1Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.3 Atom Format 2 Packet

Purpose

Indicates two *Atom elements*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	0	1	1	0	A	

Figure D5-3 Atom Format 2 Packet

Field descriptions

A

Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

A	Description
`0b00`	1. N *Atom element*. 2. N *Atom element*.
`0b01`	1. E *Atom element*. 2. N *Atom element*.
`0b10`	1. N *Atom element*. 2. E *Atom element*.
`0b11`	1. E *Atom element*. 2. E *Atom element*.

Additional information

For more information about the decoding of this packet see AtomFormat2Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.4 Atom Format 3 Packet

Purpose

Indicates three *Atom elements*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	1	1	1	A		

Figure D5-4 Atom Format 3 Packet

Field descriptions

A

Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

A	Description
`0b000`	1. N *Atom element*. 2. N *Atom element*. 3. N *Atom element*.
`0b001`	1. E *Atom element*. 2. N *Atom element*. 3. N *Atom element*.
`0b010`	1. N *Atom element*. 2. E *Atom element*. 3. N *Atom element*.
`0b011`	1. E *Atom element*. 2. E *Atom element*. 3. N *Atom element*.
`0b100`	1. N *Atom element*. 2. N *Atom element*. 3. E *Atom element*.
`0b101`	1. E *Atom element*. 2. N *Atom element*. 3. E *Atom element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A	Description
`0b110`	1. N *Atom element*. 2. E *Atom element*. 3. E *Atom element*.
`0b111`	1. E *Atom element*. 2. E *Atom element*. 3. E *Atom element*.

Additional information

For more information about the decoding of this packet see AtomFormat3Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.5 Atom Format 4 Packet

Purpose

Indicates four *Atom elements*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	0	1	1	1	A	

Figure D5-5 Atom Format 4 Packet

Field descriptions

A

Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

A	Description
`0b00`	1. N *Atom element*. 2. E *Atom element*. 3. E *Atom element*. 4. E *Atom element*.
`0b01`	1. N *Atom element*. 2. N *Atom element*. 3. N *Atom element*. 4. N *Atom element*.
`0b10`	1. N *Atom element*. 2. E *Atom element*. 3. N *Atom element*. 4. E *Atom element*.
`0b11`	1. E *Atom element*. 2. N *Atom element*. 3. E *Atom element*. 4. N *Atom element*.

Additional information

For more information about the decoding of this packet see AtomFormat4Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.6 Atom Format 5.1 Packet

Purpose

Indicates five *Atom elements*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1

Figure D5-6 Atom Format 5.1 Packet

Element sequence

This packet encodes the following sequence:

1. N *Atom element*.
2. E *Atom element*.
3. E *Atom element*.
4. E *Atom element*.
5. E *Atom element*.

Additional information

For more information about the decoding of this packet see AtomFormat5_1Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.7 Atom Format 5.2 Packet

Purpose

Indicates five *Atom elements*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	0	1	0	1	A	

Figure D5-7 Atom Format 5.2 Packet

Field descriptions

A

Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

A	Description
`0b01`	1. N *Atom element*. 2. N *Atom element*. 3. N *Atom element*. 4. N *Atom element*. 5. N *Atom element*.
`0b10`	1. N *Atom element*. 2. E *Atom element*. 3. N *Atom element*. 4. E *Atom element*. 5. N *Atom element*.
`0b11`	1. E *Atom element*. 2. N *Atom element*. 3. E *Atom element*. 4. N *Atom element*. 5. E *Atom element*.

Additional information

For more information about the decoding of this packet see AtomFormat5_2Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.8 Atom Format 6 Packet

Purpose

Indicates 3-23 E *Atom elements*, plus a subsequent E Atom or N *Atom element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	A	COUNT				

Figure D5-8 Atom Format 6 Packet

Field descriptions

A

Indicates an E *Atom element* or N *Atom element*, after the E *Atom elements* indicated by COUNT.

The encoding for this field is POD.

A	Description
`0b0`	One E *Atom element*
`0b1`	One N *Atom element*

COUNT

Indicates a number of E *Atom elements*. The number is 3 + COUNT. Permitted values of COUNT are `0b00000` to `0b10100`.

The encoding for this field is POD.

Additional information

For more information about the decoding of this packet see AtomFormat6Packet().

For more information about the generation of this packet see Atom packing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.9 Commit Packet

Purpose

Indicates a *Commit element*.

Configurations

TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1
C0	COMMIT[6:0]						
C0	COMMIT[13:7]						
C0	COMMIT[20:14]						
C0	COMMIT[27:21]						
(0) (0) (0) (0)				COMMIT[31:28]			

Figure D5-9 Commit Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COMMIT

The number of *P0 elements* to be resolved.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.

Additional information

For more information about the decoding of this packet see CommitPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.10 Context Packet

Purpose

Indicates a *Context element*.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
0	0	NS	SF	NSE	(0)	EL	

Figure D5-10 Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-11 Context Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-12 Context Packet (3)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-13 Context Packet (4)

Field descriptions

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL

Exception level.

The encoding for this field is POD.

EL	Description
`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
0b0	The PE is in AArch32 state.
0b1	The PE is in AArch64 state.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Context element*.

Additional information

For more information about the decoding of this packet see Parse_ContextBytes().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.11 Context Same Packet

Purpose

Indicates a *Context element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

Figure D5-14 Context Same Packet

Element sequence

This packet encodes the following sequence:

1. *Context element*.

Additional information

For more information about the decoding of this packet see Parse_ContextBytes().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.12 Cycle Count Format 1_0 unknown count Packet

Purpose

Indicates zero or one *Commit elements* followed by a *Cycle Count element* with an UNKNOWN cycle count value.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1
C0	COMMIT[6:0]						
C0	COMMIT[13:7]						
C0	COMMIT[20:14]						
C0	COMMIT[27:21]						
(0) (0) (0) (0)				COMMIT[31:28]			

Figure D5-15 Cycle Count Format 1_0 unknown count Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COMMIT

If this field is zero, there is no *Commit element*. Otherwise, there is a *Commit element* before the *Cycle Count element* and this field indicates the number of *P0 elements* committed by the *Commit element*.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element* with an unknown cycle count.

Additional information

For more information about the decoding of this packet see CycleCountFormat1Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.13 Cycle Count Format 1_0 with count Packet

Purpose

Indicates zero or one *Commit elements* followed by a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0
C0	COMMIT[6:0]						
C0	COMMIT[13:7]						
C0	COMMIT[20:14]						
C0	COMMIT[27:21]						
(0) (0) (0) (0)				COMMIT[31:28]			
C1	COUNT[6:0]						
C1	COUNT[13:7]						
(0) (0)		COUNT[19:14]					

Figure D5-16 Cycle Count Format 1_0 with count Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1

Continuation Bit.

The encoding for this field is Unary code.

C1	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMMIT

If this field is zero, there is no *Commit element*. Otherwise, there is a *Commit element* before the *Cycle Count element* and this field indicates the number of *P0 elements* committed by the *Commit element*.

The encoding for this field is unsigned LE128n.

COUNT

Indicates the number of PE clock cycles that have occurred between the 2 most recent *Commit elements* that both had a *Cycle Count element* associated with them. The cycle count is COUNT+cc_threshold.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat1Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.14 Cycle Count Format 1_1 unknown count Packet

Purpose

Indicates a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b1`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1

Figure D5-17 Cycle Count Format 1_1 unknown count Packet

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element* with an unknown cycle count.

Additional information

For more information about the decoding of this packet see CycleCountFormat1Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.15 Cycle Count Format 1_1 with count Packet

Purpose

Indicates a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b1`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0
C0	COUNT[6:0]						
C0	COUNT[13:7]						
(0) (0)	COUNT[19:14]						

Figure D5-18 Cycle Count Format 1_1 with count Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

Indicates the number of PE clock cycles that have occurred between the 2 most recent *Commit elements* that both had a *Cycle Count element* associated with them. The cycle count is COUNT+cc_threshold.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat1Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.16 Cycle Count Format 2_0 large commit Packet

Purpose

Indicates a *Commit element* and a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1
AAAA				BBBB			

Figure D5-19 Cycle Count Format 2_0 large commit Packet

Field descriptions

AAAA

Indicates the number of *P0 elements* to be resolved indicated by TRCIDR8.MAXSPEC + field - 15.

The number of P0 elements to be resolved must be greater than 0.

If the number of P0 elements to be resolved is less than 17 then it is preferred that a Cycle Count Format 2_0 small commit Packet is used.

The encoding for this field is POD.

BBBB

Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat2Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.17 Cancel Format 1 Packet

Purpose

Indicates a *Cancel element* optionally followed by a *Mispredict element*.

Configurations

TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	M
C0	CANCEL[6:0]						
C0	CANCEL[13:7]						
C0	CANCEL[20:14]						
C0	CANCEL[27:21]						
(0) (0) (0) (0)				CANCEL[31:28]			

Figure D5-20 Cancel Format 1 Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

CANCEL

The number of *P0 elements* to be canceled.

The encoding for this field is unsigned LE128n.

CANCEL	Description
`0b0`	Reserved

M

Mispredict element included in the packet.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Description
`0b0`	No *Mispredict element* occurred
`0b1`	A *Mispredict element* occurred after the *Cancel element*

Additional information

For more information about the decoding of this packet see CancelFormat1Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.18 Cancel Format 2 Packet

Purpose

Indicates zero or more E or N *Atom elements* followed by a *Cancel element* and a *Mispredict element*.

Configurations

TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	1	0	1	A	

Figure D5-21 Cancel Format 2 Packet

Field descriptions

A

Indicates the number of *Atom elements* that occurred before the *Cancel element*.

The encoding for this field is POD.

A	Description
`0b00`	1. *Cancel element*. 2. *Mispredict element*.
`0b01`	1. E *Atom element*. 2. *Cancel element*. 3. *Mispredict element*.
`0b10`	1. E *Atom element*. 2. E *Atom element*. 3. *Cancel element*. 4. *Mispredict element*.
`0b11`	1. N *Atom element*. 2. *Cancel element*. 3. *Mispredict element*.

Additional information

For more information about the decoding of this packet see CancelFormat2Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.19 Cancel Format 3 Packet

Purpose

Indicates zero or one E *Atom element* followed by a *Cancel element* with a payload of 2-5 and one *Mispredict element*.

Configurations

TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	1	1	CC		A

Figure D5-22 Cancel Format 3 Packet

Field descriptions

A

Indicates the number of *Atom elements* that occurred before the *Cancel element*.

The encoding for this field is POD.

A	Description
0b0	1. *Cancel element*.
0b1	1. E *Atom element*. 2. *Cancel element*.

CC

The number of *P0 elements* to be canceled.

The encoding for this field is POD.

CC	Description
0b00	Cancel 2 *P0 elements*
0b01	Cancel 3 *P0 elements*
0b10	Cancel 4 *P0 elements*
0b11	Cancel 5 *P0 elements*

Additional information

For more information about the decoding of this packet see CancelFormat3Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.20 Cycle Count Format 2_0 small commit Packet

Purpose

Indicates a *Commit element* and a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0
AAAA				BBBB			

Figure D5-23 Cycle Count Format 2_0 small commit Packet

Field descriptions

AAAA

Indicates the number of *P0 elements* to be resolved indicated by this field + 1.

The encoding for this field is POD.

BBBB

Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat2Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.21 Cycle Count Format 2_1 Packet

Purpose

Indicates a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b1`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1
1	1	1	1	BBBB			

Figure D5-24 Cycle Count Format 2_1 Packet

Field descriptions

BBBB

Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat2Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.22 Cycle Count Format 3_0 Packet

Purpose

Indicates a *Commit element* and a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	1	AA		BB	

Figure D5-25 Cycle Count Format 3_0 Packet

Field descriptions

AA

The number of *P0 elements* to be resolved indicated by this field + 1.

The encoding for this field is POD.

BB

Indicates the cycle value. The cycle count is calculated from cc_threshold + BB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat3Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.23 Cycle Count Format 3_1 Packet

Purpose

Indicates a *Cycle Count element*.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b1`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	1	0	0	BB	

Figure D5-26 Cycle Count Format 3_1 Packet

Field descriptions

BB

Indicates the cycle value. The cycle count is calculated from cc_threshold + BB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see CycleCountFormat3Packet().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.24 Discard Packet

Purpose

Indicates a *Discard element*.

Configurations

All.

Indicates a *Discard element*.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1

Figure D5-27 Discard Packet

Element sequence

This packet encodes the following sequence:

1. *Discard element*.

Additional information

For more information about the decoding of this packet see DiscardPacket().

For more information about the generation of this packet see Trace protocol synchronization.

I_{RTFPP} This packet is used to discard any speculative trace that the trace analyzer might still be holding onto.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.25 Event Packet

Purpose

Indicates 1-4 *Event elements*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	1	1	1	V3	V2	V1	V0

Figure D5-28 Event Packet

Field descriptions

V0

Event 0 indicator.

The encoding for this field is POD.

V0	Description
`0b0`	*ETEEvent* 0 did not occur
`0b1`	*ETEEvent* 0 occurred

V1

Event 1 indicator.

The encoding for this field is POD.

V1	Description
`0b0`	*ETEEvent* 1 did not occur
`0b1`	*ETEEvent* 1 occurred

V2

Event 2 indicator.

The encoding for this field is POD.

V2	Description
`0b0`	*ETEEvent* 2 did not occur
`0b1`	*ETEEvent* 2 occurred

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

V3

Event 3 indicator.

The encoding for this field is POD.

V3	Description
`0b0`	*ETEEvent* 3 did not occur
`0b1`	*ETEEvent* 3 occurred

Additional information

For more information about the decoding of this packet see EventTracingPacket().

Note

[V3, V2, V1, V0] != `0b0000` as this is decoded as an Ignore Packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.26 Exception 32-bit Address IS0 Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							

Figure D5-29 Exception 32-bit Address IS0 Packet

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.27 Exception 32-bit Address IS0 with Context Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-30 Exception 32-bit Address IS0 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-31 Exception 32-bit Address IS0 with Context Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-32 Exception 32-bit Address IS0 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-33 Exception 32-bit Address IS0 with Context Packet (4)

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL

Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

EL	Description
`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
0b0	The PE is in AArch32 state.
0b1	The PE is in AArch64 state.

TYPE

The exception type.

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.28 Exception 32-bit Address IS1 Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							

Figure D5-34 Exception 32-bit Address IS1 Packet

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.29 Exception 32-bit Address IS1 with Context Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-35 Exception 32-bit Address IS1 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-36 Exception 32-bit Address IS1 with Context Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-37 Exception 32-bit Address IS1 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-38 Exception 32-bit Address IS1 with Context Packet (4)

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL

Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

EL	Description
`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
0b0	The PE is in AArch32 state.
0b1	The PE is in AArch64 state.

TYPE

The exception type.

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.30 Exception 64-bit Address IS0 Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5-39 Exception 64-bit Address IS0 Packet

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.31 Exception 64-bit Address IS0 with Context Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-40 Exception 64-bit Address IS0 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-41 Exception 64-bit Address IS0 with Context Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-42 Exception 64-bit Address IS0 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-43 Exception 64-bit Address IS0 with Context Packet (4)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL

Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

EL	Description
`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
0b0	The PE is in AArch32 state.
0b1	The PE is in AArch64 state.

TYPE

The exception type.

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.32 Exception 64-bit Address IS1 Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5-44 Exception 64-bit Address IS1 Packet

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.33 Exception 64-bit Address IS1 with Context Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-45 Exception 64-bit Address IS1 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-46 Exception 64-bit Address IS1 with Context Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-47 Exception 64-bit Address IS1 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-48 Exception 64-bit Address IS1 with Context Packet (4)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL

Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

EL	Description
`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
0b0	The PE is in AArch32 state.
0b1	The PE is in AArch64 state.

TYPE

The exception type.

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.34 Exception Exact Match Address Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	0	0	A	

Figure D5-49 Exception Exact Match Address Packet

Field descriptions

A

Preferred Exception Return address.

The encoding for this field is POD.

A	Description
`0b00`	The Preferred Exception Return is the same as address history buffer entry 0.
`0b01`	The Preferred Exception Return is the same as address history buffer entry 1.
`0b10`	The Preferred Exception Return is the same as address history buffer entry 2.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.35 Exception Short Address IS0 Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	0	1	0	1
C0	A[8:2]						
A[16:9]							

Figure D5-50 Exception Short Address IS0 Packet

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E	Description
0b10	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.36 Exception Short Address IS1 Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	0	1	1	0
C0	A[7:1]						
A[15:8]							

Figure D5-51 Exception Short Address IS1 Packet

Field descriptions

A

Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (TYPE, ADDRESS).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E	Description
0b10	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE

The exception type.

The encoding for this field is POD.

TYPE	Description
0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.37 Ignore Packet

Purpose

To align packet boundary to memory boundary.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0

Figure D5-52 Ignore Packet

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.38 Instrumentation Packet

Purpose

Indicates an Instrumentation element.

Configurations

When FEAT_ITE is implemented.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1
(0) (0) (0) (0) (0) (0)						EL	
VALUE[7:0]							
VALUE[15:8]							
VALUE[23:16]							
VALUE[31:24]							
VALUE[39:32]							
VALUE[47:40]							
VALUE[55:48]							
VALUE[63:56]							

Figure D5-53 Instrumentation Packet

Field descriptions

EL

Exception level at which the corresponding TRCIT instruction was executed.

The encoding for this field is POD.

EL	Description
`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

VALUE

Value provided in Xn of the TRCIT instruction.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. Instrumentation element.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.39 Mispredict Packet

Purpose

Indicates 0-2 E or N *Atom elements* followed by one *Mispredict element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	1	0	0	A	

Figure D5-54 Mispredict Packet

Field descriptions

A

Indicates the number of *Atom elements* that occurred before the *Mispredict element*.

The encoding for this field is POD.

A	Description
`0b00`	1. *Mispredict element*.
`0b01`	1. E *Atom element*. 2. *Mispredict element*.
`0b10`	1. E *Atom element*. 2. E *Atom element*. 3. *Mispredict element*.
`0b11`	1. N *Atom element*. 2. *Mispredict element*.

Additional information

For more information about the decoding of this packet see MispredictPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.40 Overflow Packet

Purpose

Indicates that a trace unit buffer overflow has occurred.

Configurations

All.

Indicates that a trace unit buffer overflow has occurred and data might have been lost.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1

Figure D5-55 Overflow Packet

Element sequence

This packet encodes the following sequence:

1. *Overflow element*.
2. *Discard element*.

Additional information

For more information about the decoding of this packet see OverflowPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.41 PE Reset Packet

Purpose

Indicates that a PE Reset has occurred.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	0	0	0	0	0	E[0]
0	1	1	1	0	0	0	0

Figure D5-56 PE Reset Packet

Field descriptions

E

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

E	Description
`0b01`	1. *Exception element* (PE_Reset, UNKNOWN).
`0b10`	1. *Target Address element* (UNKNOWN). 2. *Exception element* (PE_Reset, UNKNOWN).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket().

For more information about the generation of this packet see Address compression.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.42 Q 32-bit address IS0 Packet

Purpose

Indicates that some instructions have executed with an address of the next instruction.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5-57 Q 32-bit address IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.43 Q 32-bit address IS1 Packet

Purpose

Indicates that some instructions have executed with an address of the next instruction.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5-58 Q 32-bit address IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.44 Q Packet

Purpose

Indicates that some instructions have executed, without a count of the number of instructions.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	1

Figure D5-59 Q Packet

Element sequence

This packet encodes the following sequence:

1. *Q element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.45 Q short address IS0 Packet

Purpose

Indicates that some instructions have executed with an address of the next instruction.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1
C0	A[8:2]						
A[16:9]							
C1	COUNT[6:0]						
C1	COUNT[13:7]						
C1	COUNT[20:14]						
C1	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5-60 Q short address IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1

Continuation Bit.

The encoding for this field is Unary code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.46 Q short address IS1 Packet

Purpose

Indicates that some instructions have executed with an address of the next instruction.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	0	1	1	0
C0	A[7:1]						
A[15:8]							
C1	COUNT[6:0]						
C1	COUNT[13:7]						
C1	COUNT[20:14]						
C1	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5-61 Q short address IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1

Continuation Bit.

The encoding for this field is Unary code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.47 Q with count Packet

Purpose

Indicates that some instructions have executed.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	0
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5-62 Q with count Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.48 Q with Exact match address Packet

Purpose

Indicates that some instructions have executed with an address of the next instruction.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	0	0	TYPE	
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5-63 Q with Exact match address Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

TYPE

The TYPE field indicates what form the rest of the Packet takes.

The encoding for this field is POD.

TYPE	Description
`0b00`	A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 0.
`0b01`	A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Description
`0b10`	A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 2.
`0b11`	RESERVED

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_QPacket().

For more information about the generation of this packet see Address compression.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.49 Source Address 32-bit IS0 Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							

Figure D5-64 Source Address 32-bit IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.50 Source Address 32-bit IS1 Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							

Figure D5-65 Source Address 32-bit IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.51 Source Address 64-bit IS0 Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5-66 Source Address 64-bit IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.52 Source Address 64-bit IS1 Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5-67 Source Address 64-bit IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.53 Source Address Exact Match Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	0	QE	

Figure D5-68 Source Address Exact Match Packet

Field descriptions

QE

Instruction virtual address.

The encoding for this field is POD.

QE	Description
`0b00`	The address is the same as address history buffer entry 0.
`0b01`	The address is the same as address history buffer entry 1.
`0b10`	The address is the same as address history buffer entry 2.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.54 Source Address Short IS0 Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	0
C0	A[8:2]						
A[16:9]							

Figure D5-69 Source Address Short IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.55 Source Address Short IS1 Packet

Purpose

Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	1
C0	A[7:1]						
A[15:8]							

Figure D5-70 Source Address Short IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.56 Timestamp Marker Packet

Purpose

Indicates a *Timestamp Marker element*.

Configurations

TRCIDR0.TSSIZE != `0b00000` and TRCIDR0.TSMARK == `0b1`

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0

Figure D5-71 Timestamp Marker Packet

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.57 Timestamp Packet

Purpose

Indicates a *Timestamp element*.

Configurations

TRCIDR0.TSSIZE != `0b00000`.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0
C0	TS[6:0]						
C0	TS[13:7]						
C0	TS[20:14]						
C0	TS[27:21]						
C0	TS[34:28]						
C0	TS[41:35]						
C0	TS[48:42]						
C0	TS[55:49]						
TS[63:56]							

Figure D5-72 Timestamp Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1
C0	TS[6:0]						
C0	TS[13:7]						
C0	TS[20:14]						
C0	TS[27:21]						
C0	TS[34:28]						
C0	TS[41:35]						
C0	TS[48:42]						
C0	TS[55:49]						
TS[63:56]							
C1	COUNT[6:0]						
C1	COUNT[13:7]						
(0) (0)	COUNT[19:14]						

Figure D5-73 Timestamp Packet (2)

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C0	Description
0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

C1

Continuation Bit.

The encoding for this field is Unary code.

C1	Description
0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

COUNT

The number of PE clock cycles between the most recent *Cycle Count element* and the element related to the Timestamp. When the COUNT field is not present, the cycle count value is UNKNOWN.

The encoding for this field is unsigned LE128n.

TS

Timestamp Value.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Timestamp element*.

Additional information

For more information about the decoding of this packet see TimestampPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.58 Target Address 32-bit IS0 Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							

Figure D5-74 Target Address 32-bit IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.59 Target Address 32-bit IS1 Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							

Figure D5-75 Target Address 32-bit IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.60 Target Address 64-bit IS0 Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5-76 Target Address 64-bit IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.61 Target Address 64-bit IS1 Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5-77 Target Address 64-bit IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.62 Target Address Exact Match Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	0	0	QE	

Figure D5-78 Target Address Exact Match Packet

Field descriptions

QE

Instruction virtual address.

The encoding for this field is POD.

QE	Description
`0b00`	The address is the same as address history buffer entry 0.
`0b01`	The address is the same as address history buffer entry 1.
`0b10`	The address is the same as address history buffer entry 2.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.63 Target Address Short IS0 Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1
C0	A[8:2]						
A[16:9]							

Figure D5-79 Target Address Short IS0 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.64 Target Address Short IS1 Packet

Purpose

Indicates a *Target Address element*.

Configurations

All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	0
C0	A[7:1]						
A[15:8]							

Figure D5-80 Target Address Short IS1 Packet

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.65 Target Address with Context 32-bit IS0 Packet

Purpose

Indicates a *Target Address element* and a *Context element*.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-81 Target Address with Context 32-bit IS0 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-82 Target Address with Context 32-bit IS0 Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-83 Target Address with Context 32-bit IS0 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-84 Target Address with Context 32-bit IS0 Packet (4)

Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL

Exception level at this address.

The encoding for this field is POD.

EL	Description
0b00	EL0
0b01	EL1
0b10	EL2
0b11	EL3

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SF	Description
`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.
2. *Context element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.66 Target Address with Context 32-bit IS1 Packet

Purpose

Indicates a *Target Address element* and a *Context element*.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-85 Target Address with Context 32-bit IS1 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-86 Target Address with Context 32-bit IS1 Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-87 Target Address with Context 32-bit IS1 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-88 Target Address with Context 32-bit IS1 Packet (4)

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL

Exception level at this address.

The encoding for this field is POD.

EL	Description
0b00	EL0
0b01	EL1
0b10	EL2
0b11	EL3

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b0	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
0b1	0b1	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

SF

AArch64 state.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SF	Description
`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.
2. *Context element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.67 Target Address with Context 64-bit IS0 Packet

Purpose

Indicates a *Target Address element* and a *Context element*.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-89 Target Address with Context 64-bit IS0 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-90 Target Address with Context 64-bit IS0 Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-91 Target Address with Context 64-bit IS0 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-92 Target Address with Context 64-bit IS0 Packet (4)

Field descriptions

A

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL

Exception level at this address.

The encoding for this field is POD.

EL	Description
`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
`0b0`	`0b0`	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
`0b0`	`0b1`	Non-secure state.
`0b1`	`0b0`	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
`0b1`	`0b1`	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.
2. *Context element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.68 Target Address with Context 64-bit IS1 Packet

Purpose

Indicates a *Target Address element* and a *Context element*.

Configurations

All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	NSE	(0)	EL	

Figure D5-93 Target Address with Context 64-bit IS1 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	NSE	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-94 Target Address with Context 64-bit IS1 Packet (2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5-95 Target Address with Context 64-bit IS1 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	NSE	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5-96 Target Address with Context 64-bit IS1 Packet (4)

Field descriptions

A

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL

Exception level at this address.

The encoding for this field is POD.

EL	Description
`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS

Security state.

The encoding for this field is POD.

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
`0b0`	`0b0`	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
`0b0`	`0b1`	Non-secure state.
`0b1`	`0b0`	Root state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.
`0b1`	`0b1`	Realm state. This value is defined only if FEAT_RME is implemented, and reserved otherwise.

NSE

Security state, for more information see the NS field.

The encoding for this field is POD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SF

AArch64 state.

The encoding for this field is POD.

SF	Description
`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.
2. *Context element*.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket().

For more information about the generation of this packet see Address compression.

For more information about the encoding of this packet see Instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.69 Trace Info Packet

Purpose

Resets trace compression to a known architectural state.

Configurations

All.

The trace info packet resets the trace compression to a known state.

Any fields which are not output are treated as if the value is zero.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0

Figure D5-97 Trace Info Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1
(0) (0) (0) (0) (0) (0) (0)							CC

Figure D5-98 Trace Info Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			

Figure D5-99 Trace Info Packet (3)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	1
(0) (0) (0) (0) (0) (0) (0)							CC
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			

Figure D5-100 Trace Info Packet (4)

Packet Layout - Variant 5

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
C1	CYCT[6:0]						
(0) (0) (0)				CYCT[11:7]			

Figure D5-101 Trace Info Packet (5)

Packet Layout - Variant 6

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	1
(0) (0) (0) (0) (0) (0) (0)							CC
C1	CYCT[6:0]						
(0) (0) (0)				CYCT[11:7]			

Figure D5-102 Trace Info Packet (6)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Packet Layout - Variant 7

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	0
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			
C1	CYCT[6:0]						
(0) (0) (0)			CYCT[11:7]				

Figure D5-103 Trace Info Packet (7)

Packet Layout - Variant 8

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	1
(0) (0) (0) (0) (0) (0) (0)							CC
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			
C1	CYCT[6:0]						
(0) (0) (0)				CYCT[11:7]			

Figure D5-104 Trace Info Packet (8)

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

C0	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1

Continuation Bit.

The encoding for this field is Unary code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1	Description
`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

CC

Cycle count enable indicator.

When this field is not output, it is treated as if it is zero.

The encoding for this field is POD.

CC	Description
`0b0`	Cycle counting is not enabled.
`0b1`	Cycle counting is enabled.

CYCT

The cycle count threshold.

When this field is not output, it is treated as if it is zero.

The encoding for this field is unsigned LE128n.

SPEC

The number of uncommitted *P0 elements* in the trace.

When this field is not output, it is treated as if it is zero.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Trace Info element*.

Additional information

For more information about the decoding of this packet see TraceInfoPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3.70 Trace On Packet

Purpose

Indicates that there has been a discontinuity in the trace element stream.

Configurations

All.

A Trace On packet indicates to a trace analyzer that the trace unit has generated a *Trace On element*.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0

Figure D5-105 Trace On Packet

Element sequence

This packet encodes the following sequence:

1. *Trace On element*.

Additional information

For more information about the decoding of this packet see TraceOnPacket().

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D6
The Trace Buffer Extension

This chapter describes the *Trace Buffer Extension* (TRBE). It contains the following sections:

- About the Trace Buffer Extension.
- The trace buffer.
- Trace buffer Self-hosted mode.
- Trace buffer External mode.
- Trace buffer management.
- Synchronization and the Trace Buffer Unit.
- Trace synchronization and memory barriers.
- Trace of Speculative execution.
- Trace in Debug state.
- Synchronization litmus tests.
- UNPREDICTABLE behavior.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.1 About the Trace Buffer Extension

IMRFPK The Trace Buffer Extension feature is identified as FEAT_TRBE.

When FEAT_TRBE is implemented, the PE includes a Trace Buffer Unit. There is one logical Trace Buffer Unit for each PE in the processor.

When the Trace Buffer Unit is enabled, Program Flow Trace generated by the trace unit is written directly to memory by the Trace Buffer Unit, rather than routing it to a trace fabric. Figure D6-1 shows this.

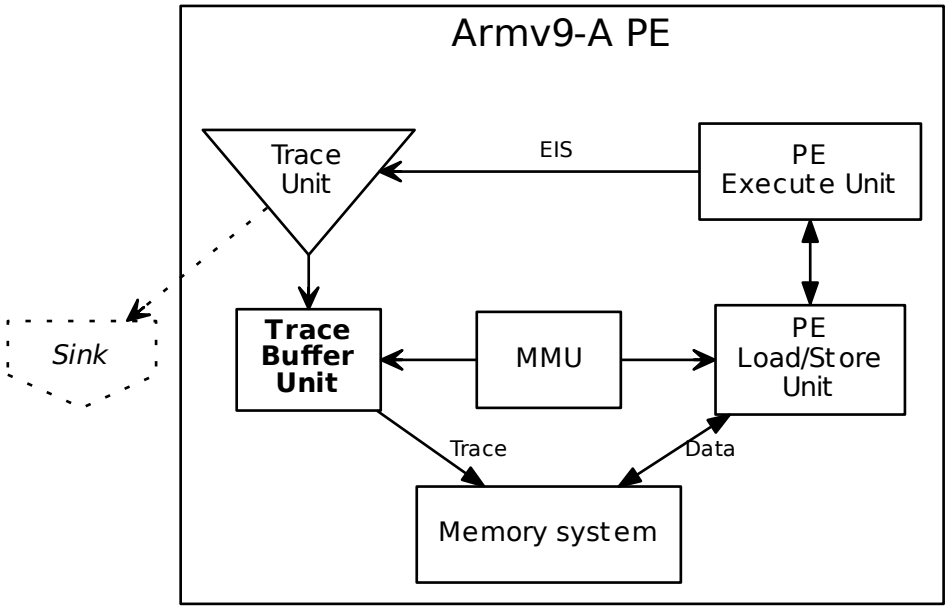

Figure D6-1 Logical organization of an Armv9-A PE including a trace unit and a Trace Buffer Unit

In this figure:

- *EIS* is an internal representation of the executed instruction stream.
- The trace unit converts the EIS into formatted trace data.
- *Sink* is described by the section Trace Buffer Unit disabled.

IFGGXR To allow use by self-hosted software in a platform OS environment, a Trace Buffer Unit or ETR must support a trace buffer that is mapped to a set of non-contiguous physical blocks in memory. The Trace Buffer Unit achieves this by using the PE VMSA-based MMU.

This means that all of the following apply:

- The trace buffer is normally virtually addressed.
- The trace buffer has an owning Exception level and owning Security state that define the translation regime the trace buffer uses.
- FEAT_TRBE provides a synchronization instruction, `TSB CSYNC`, that is used with a `DSB` operation to flush trace to the trace buffer.
- For FEAT_TRBE, trace is implicitly prohibited when the owning translation regime is not in context. That is, trace is prohibited if executing at a higher Exception level than the owning Exception level, or not executing in the owning Security state. This is in addition to the behavior in FEAT_ETE and FEAT_TRF for when trace is prohibited.

IFGPMM FEAT_TRBE also allows the trace buffer to be defined using physical addresses. This allows the Trace Buffer Unit to be used for debugging software that changes the virtual address mappings. In this configuration, the buffer must be contiguously mapped in physical memory.

ISTZGC The Trace Buffer Unit supports the following operational modes:

Trace buffer mode

Controls how the Trace Buffer Unit uses the trace buffer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trigger mode

Controls how the Trace Buffer Unit reacts to a trigger condition signaled by the trace unit.

R_{PLYXP} Other than where stated otherwise, this chapter describes a simple sequential model of the Trace Buffer Unit. That is, one which performs the simple loop of:

1. Collect a single byte of trace data from the trace unit.
2. If required, translate the *current write pointer* virtual address to a physical address.
3. If permitted, write the byte of trace data to the write address.
4. If collection is not stopped, increment the *current write pointer*.
5. If necessary, decrement the *Trigger Counter*.

Trace buffer management events are processed as part of this operation loop.

Implementations compliant with the architecture conform with the described behavior of the Trace Buffer Unit.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with this specification is the same as a simple sequential model. Trace appears to be written sequentially by the Trace Buffer Unit.

I_{PTDZC} If FEAT_TRBE_EXT is implemented, then the Trace Buffer Unit can be controlled externally from the PE instead of by self-hosted software. The following are possible methods of external Trace Buffer Unit control:

- An external trace debugger that programs the Trace Buffer Unit through an external debug interface.
- An external trace debugger executing on the same SoC as the PE being traced where trace is controlled through the memory-mapped external debug interface.

For more information, see Trace buffer External mode.

D6.1.1 The trace buffer pointers

R_{WKBRT} The trace buffer is defined by three trace buffer pointer addresses:

- The *Base pointer*.
- The *Limit pointer*.
- The *current write pointer*.

R_{FVPBS} The trace buffer starts at the Base pointer and extends to the Limit pointer. The location at the Base pointer is included in the trace buffer. The location at the Limit pointer is not included in the trace buffer.

R_{XBLPK} The Base pointer and Limit pointer must be aligned by software to the smallest implemented translation granule size.

R_{VHNTF} For each byte of trace that the Trace Buffer Unit *accepts* and writes to the trace buffer at the address in the current write pointer, one of the following applies:

- If the current write pointer is not equal to the Limit pointer minus one, then the current write pointer is incremented by one.
- If the current write pointer is equal to the Limit pointer minus one, then all of the following occur:
  - The current write pointer is wrapped by setting it to the Base pointer.
  - A trace buffer management event might be generated. See Trace buffer management.
  - TRBSR_ELx.WRAP is set to 1, where ELx is the value in the *Other event* column of Table D6-7. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged if no trace buffer management event is generated.
  - The TRB_WRAP event is generated.

R_{BGBCJ} The current write pointer is not incremented when collection is stopped.

R_{VMVJH} The required alignment of the current write pointer is IMPLEMENTATION DEFINED.

I_{BTSCF} The Trace Buffer Unit can write trace data to memory in quantized units. The behavior is as if the bytes are written sequentially.

R_{JMPCB} The Base pointer is (TRBBASER_EL1.BASE << 12). Bits [11:0] of the Base pointer are zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{LLBBS} The Limit pointer is (TRBLIMITR_EL1.LIMIT << 12). Bits [11:0] of the Limit pointer are zero.

R_{KXRTY} The current write pointer is TRBPTR_EL1.PTR[63:0].

R_{PBGNS} The *Trigger Counter* is TRBTRG_EL1.TRG.

D6.1.2 System events

I_{XDSQV} The trace unit can be configured to react to PE events and events from the CTI. The CTI is for use by external debuggers.

As part of FEAT_TRBE and FEAT_ETE, the PMU and FEAT_ETE event sources are unified into a single event number space. Unless otherwise stated, all architecturally defined common events that can be counted by the PMU are usable as an event at the trace unit.

The following architecturally-defined events are provided:

- The CTI_TRIGOUT<n> events, such as CTI_TRIGOUT4, are defined to map the system events from the CTI into the PMU event number space. As well as defining these events for the trace unit, this also provides a standard mechanism for counting *external events passed to the PE*.
- The TRCEXTOUT<n> events, such as TRCEXTOUT0, are defined to allow the PMU to count the events that a FEAT_ETE implementation of the trace unit might generate.
- The PMU_OVFS and PMU_HOVFS events are defined to allow the trace unit to trigger directly from a PMU overflow without using the Performance Monitors overflow trigger for PMU counters accessible to EL1 and EL0, and EL2, respectively.
- The TRB_WRAP event is defined to allow the trace unit to trigger when the *current write pointer* reaches the end of the trace buffer and is wrapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.2 The trace buffer

R_{SDCKT} The Trace Buffer Unit uses one of the following:

- If self-hosted trace is enabled, then Self-hosted mode. See Trace buffer Self-hosted mode.
- If FEAT_TRBE_EXT is implemented and self-hosted trace is disabled, then External mode. See Trace buffer External mode.

R_{YCHKJ} If and only if one of the following is true, then the Trace Buffer Unit is enabled:

- The Trace Buffer Unit is using Self-hosted mode and TRBLIMITR_EL1.E is 1.
- The Trace Buffer Unit is using External mode and TRBLIMITR_EL1.XE is 1.

Otherwise, the Trace Buffer Unit is disabled. See Trace Buffer Unit disabled.

The pseudocode function `TraceBufferEnabled()` describes this.

R_{BGLHT} If and only if all of the following are true, then the Trace Buffer Unit is *running*:

- The Trace Buffer Unit is enabled.
- TRBSR_EL1.S is 0.
- FEAT_TRBE_EXC is not implemented, or self-hosted trace is disabled, or all of the following are true:
  - Any of the following are true:
    - EL2 is not implemented.
    - The Effective value of SCR_EL3.{NS, EEL2} is {0, 0}.
    - TRBSR_EL2.S is 0.
    - EL3 is implemented and MDCR_EL3.TRBEE is `0b00`.
    - TRFCR_EL2.EE is either `0b00` or `0b01`.
  - Any of the following are true:
    - EL3 is not implemented.
    - TRBSR_EL3.S is 0.
    - MDCR_EL3.TRBEE is either `0b00` or `0b01`.

The pseudocode function `TraceBufferRunning()` shows this.

R_{FRHXV} If and only if all of the following are true, then *collection is stopped*:

- The Trace Buffer Unit is enabled.
- The Trace Buffer Unit is not running.

The pseudocode function `TraceBufferRunning()` shows this.

I_{YPSDY} Table D6-1 shows whether the Trace Buffer Unit is running when all of the following apply:

- The PE is executing in the trace buffer owning Security state.
- The PE is executing at either the owning Exception level or a lower Exception level.
- Self-hosted trace is enabled.

Each column in the left side of the table refers to the Effective value of a field in a System register.

Table D6-1 Trace Buffer running when self-hosted trace is enabled

TRBLIMITR_EL1.E	TRBSR_EL1.S	MDCR_EL3.TRBEE	TRBSR_EL3.S	TRFCR_EL2.EE	TRBSR_EL2.S	Running
`0b0`	`X`	`XX`	`X`	`XX`	`X`	False
`0b1`	`0b0`	`0b00`	`X`	`XX`	`X`	True
`0b1`	`0b0`	`0b01`	`X`	`0b0X`	`X`	True
`0b1`	`0b0`	`0b01`	`X`	`0b1X`	`0b0`	True
`0b1`	`0b0`	`0b01`	`X`	`0b1X`	`0b1`	False
`0b1`	`0b0`	`0b1X`	`0b0`	`0b0X`	`X`	True

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRBLIMITR_EL1.E	TRBSR_EL1.S	MDCR_EL3.TRBEE	TRBSR_EL3.S	TRFCR_EL2.EE	TRBSR_EL2.S	Running
0b1	0b0	0b1X	0b0	0b1X	0b0	True
0b1	0b0	0b1X	0b0	0b1X	0b1	False
0b1	0b0	0b1X	0b1	XX	X	False
0b1	0b1	XX	X	XX	X	False

I_{JYSQZ} While the Trace Buffer Unit is enabled, it collects trace data from the trace unit and does one of the following:

- *Accepts* the trace data and writes it to the trace buffer in memory.
- *Discards* the trace data. The trace data is lost.
- *Rejects* the trace data.

R_{YMYQX} When the Trace Buffer Unit is enabled and *running*, and the Trace Buffer Unit is able to accept the trace data, the Trace Buffer Unit *accepts* the trace data from the trace unit and writes it into the trace buffer.

R_{LNTVR} When the Trace Buffer Unit is enabled and *running*, and the Trace Buffer Unit is not able to accept the trace data, the Trace Buffer Unit *rejects* the trace data from the trace unit. The trace data might be retained by the trace unit until the Trace Buffer Unit *accepts* the trace data.

I_{SQYCT} For example, the Trace Buffer Unit might not be able to accept trace data while its internal buffers are full.

I_{TRCDR} If the Trace Buffer Unit *rejects* trace data and the trace unit is not able to retain the trace data, then the trace unit discards it and enters an Overflow state. Details of Overflow state and how the trace unit recovers from Overflow state are defined by the trace unit. See Trace unit behavior on a trace unit buffer overflow.

R_{YMVZL} When the Trace Buffer Unit is enabled and collection is stopped, the Trace Buffer Unit *discards* trace data from the trace unit. The trace data is lost.

R_{PHSKP} When used with a trace unit that implements FEAT_ETE, the Trace Buffer Unit ignores the value of the ETE TRCTRACEIDR register.

D6.2.1 Trace Buffer Unit disabled

R_{HNTLG} When the Trace Buffer Unit is disabled, the Trace Buffer Unit *discards* trace data from the trace unit.

R_{BSMLW} The Trace Buffer Unit does not prefetch and cache address translations when the Trace Buffer Unit is disabled.

I_{YHJDQ} When the Trace Buffer Unit is disabled the trace unit might send trace data to an IMPLEMENTATION DEFINED trace bus.

R_{JYTYH} The trace unit does not send trace data to the IMPLEMENTATION DEFINED trace bus when the Trace Buffer Unit is enabled.

I_{FPXHD} Figure D6-1 shows this IMPLEMENTATION DEFINED trace bus as a dotted line to an external trace *Sink*.
Details of this bus are outside the scope of this architecture, and might require further configuration. For example, if the trace unit implements FEAT_ETE and the trace bus is AMBA ATB, the ATID value is configured through the trace unit external trace registers.

D6.2.2 Restrictions on programming the Trace Buffer Unit

R_{MSPSD} A *current write pointer* value is out-of-range if any of the following are true:

- The *current write pointer* is less-than the *Base pointer*, treating both pointers as unsigned integers.
- The *current write pointer* is greater-than-or-equal-to the *Limit pointer*, treating both pointers as unsigned integers.
- Bits [63:56] of the *current write pointer* are not equal to bits [63:56] of the *Base pointer*.
- Bits [63:56] of the *current write pointer* are not equal to bits [63:56] of the *Limit pointer*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note: R_{MSPSD} means the *current write pointer* is out-of-range if the *Base pointer* is not less-than the *Limit pointer* or bits [63:56] of the *Base pointer* are not equal to bits [63:56] of the *Limit pointer*.

R_{XXZHM} A *current write pointer* or *Trigger Counter* value is misaligned if it is not a multiple of an IMPLEMENTATION DEFINED alignment specified by TRBIDR_EL1.Align.

R_{HXZZM} A *current write pointer* or *Trigger Counter* value is a valid restart value if it was previously initialized with a value that was not out-of-range and not misaligned and later read from the applicable register when all of the following are true:

- The Trace Buffer Unit is disabled.
- All trace operations are complete. See R_{NSFRQ} for the definition of complete.
- No External abort has been reported to the Trace Buffer Unit. See The External abort is reported to the Trace Buffer Unit.
- No write by the Trace Buffer Unit has generated an Alignment fault.
- No write by the Trace Buffer Unit has generated an asynchronous SError exception.

R_{XZWXQ} A *current write pointer* or *Trigger Counter* value is a fault value if it was previously initialized with a value that was not out-of-range and not misaligned or a value that was a valid restart value, and later read from the applicable register when all of the following are true:

- The Trace Buffer Unit is disabled.
- All trace operations are complete. See R_{NSFRQ} for the definition of complete.
- One of the following is true:
  - An External abort has been reported to the Trace Buffer Unit. See The External abort is reported to the Trace Buffer Unit.
  - A write by the Trace Buffer Unit has generated an Alignment fault.
  - A write by the Trace Buffer Unit has generated an asynchronous SError exception.

I_{CRSGP} An MMU fault does not generate a fault value. If software is able to fix the fault, then the Trace Buffer Unit can restart using the *current write pointer* and *Trigger Counter* values.

However, following an MMU fault:

- R_{YMVZL} means the Trace Buffer Unit *discards* trace because collection is stopped. That is, trace will be lost.
- R_{BQTGW} means that the *Trigger Counter* might be incorrect if a *Detected Trigger* has occurred.

See also S_{GTLCY}.

I_{SFTPM} Following a trace buffer management event, or on a context switch, the *current write pointer* and *Trigger Counter* might be misaligned. If TRBIDR_EL1.Align is nonzero, software should treat bits [M:0] as SBZP when writing to the applicable register, where M is (TRBIDR_EL1.Align-1) in each of the following situations:

- When first creating a trace buffer, software sets bits [M:0] to zero, meaning the registers are set to an aligned value.
- On a context switch, the definitions of a restart value and fault value mean software does not have to validate or modify the value read from hardware.

A *current write pointer* restart value or fault value will not be out-of-range.

I_{VRFQC} A fault value is for error handling purposes only. Software must not cause the Trace Buffer Unit to become enabled and *running* with the *current write pointer* having a fault value.

Software context switching the Trace Buffer Unit will avoid this issue because the trace buffer management event sets TRBSR_ELx.S to 1, meaning the Trace Buffer Unit will not become *running* following the context switch.

R_{JWWWM} If the *current write pointer* is written by a direct write with a misaligned value that is not a restart value and not a fault value, the value returned by a subsequent direct read of the *current write pointer* is UNKNOWN.

R_{MGZWR} If the *current write pointer* has an out-of-range value, or a misaligned value that is not a restart value when the Trace Buffer Unit attempts to write to the trace buffer, then any of the following might occur:

- If the value is out-of-range, the *current write pointer* might be wrapped before or after the write, and the TRB_WRAP event might be generated.
- If the value is misaligned, the write might generate an Alignment fault.
- The Trace Buffer Unit might write the trace data to any address in memory that is writable by a privileged access in the owning translation regime. These addresses are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For Self-hosted mode:
  - Virtual addresses in the owning translation regime if the Effective value of TRBLIMITR_EL1.nVM is 0.
  - Intermediate physical addresses in the owning Security state if the Effective value of TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations.
  - Physical addresses in the owning Security state if the Effective value of TRBLIMITR_EL1.nVM is 1, and either the owning translation regime has no stage 2 translation or stage 2 translation is disabled.
- For External mode, any addresses in the physical address space specified by TRBMAR_EL1.PAS.

- The write might generate a trace buffer management event and the TRB_WRAP event might be generated:
  - If FEAT_TRBE_EXC is not implemented, a buffer management event for an UNKNOWN reason:
    - TRBSR_EL1.S is either set to 1 or unchanged.
    - TRBSR_EL1.WRAP is either set to 1 or unchanged.
    - TRBSR_EL1.EC is set to an UNKNOWN value.
    - TRBSR_EL1.MSS is set to an UNKNOWN value.
  - If FEAT_TRBE_EXC is implemented, a buffer management event for one of the following reasons:
    - A Buffer full buffer management event.
    - A Buffer wrap buffer management event.
    - An Access not allowed buffer management event.
    - Any MMU fault buffer management event.

    The management event is recorded as described by R_{BNQRP}.
- All trace data is silently discarded without being written to memory.
- The Trace Buffer Unit behaves as if the trace buffer is disabled.

R_{CPDDM} If the *Trigger Counter* is written by a direct write with a misaligned value that is not a restart value, then all of the following apply:

- If the value is not a fault value, the value returned by a subsequent direct read of the *Trigger Counter* register is UNKNOWN.
- The generation of a *Trigger Event* while the Trace Buffer Unit remains enabled and *running* is UNPREDICTABLE.

I_{YSXXN} R_{XXZHM} and R_{CPDDM} mean an implementation that always keeps the *current write pointer* and/or *Trigger Counter* aligned to the IMPLEMENTATION DEFINED alignment specified by TRBIDR_EL1.Align, where TRBIDR_EL1.Align is greater-than-zero (byte alignment), can implement bits [*M*:0] of the applicable register(s) as RAZ/WI bits, where *M* is (TRBIDR_EL1.Align-1).

R_{HXZZM} allows an implementation where an External abort is reported to the Trace Buffer Unit and handled synchronously to implement TRBPTR_EL1[*M*:*N*], where *N* is IMPLEMENTATION SPECIFIC and typically determined by the minimum memory access granule, as read/write bits for the purpose of reporting an External abort fault address, but otherwise ignore the value in these bits. If *N>0*, bits [(*N-1)*:0] can be implemented as RAZ/WI.

R_{DJMDD} When the following conditions apply, the PE might ignore a direct or external write to any of certain Trace Buffer Unit registers, other than a direct write or external write to TRBLIMITR_EL1 that clears the conditions:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

The Trace Buffer Unit registers affected are:

- The *current write pointer*, TRBPTR_EL1.
- The *Base pointer*, TRBBASER_EL1.
- The *Limit pointer*, TRBLIMITR_EL1.
- The *Trigger Counter*, TRBTRG_EL1.
- TRBSR_EL1.
- TRBSR_EL2 and TRBSR_EL3, if FEAT_TRBE_EXC is implemented.
- TRBMAR_EL1.
- TRBMPAM_EL1, if FEAT_TRBE_MPAM is implemented.
- TRBITCTRL, if implemented as part of FEAT_TRBE_EXT.

Writes to TRBCR are not affected.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S_{DSSHH} Software must use appropriate Context synchronization operations to order a direct write that modifies TRBLIMITR_EL1.E with respect to other direct writes to Trace Buffer Unit registers. This includes a write to enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1.

See also:

- Synchronization and the Trace Buffer Unit.
- UNPREDICTABLE behavior.
- Context switching.

D6.2.3 Effect on the exclusive monitors

R_{DCVBN} If an operation between Load-Exclusive and Store-Exclusive instructions is traced, and the trace data is written to an unrelated address, then the write has no effect on the exclusive monitors.

R_{NWSKV} If the Trace Buffer Unit writes to the marked address of an exclusives monitor in the Exclusive Access state, then there is a CONSTRAINED UNPREDICTABLE choice of one of the following occurring:

- The write has the same effect on the exclusives monitor as a store by the PE or any other Observer to that address.
- The write has no effect on the exclusives monitor.

D6.2.4 Effect of MTE

R_{YGMLW} If FEAT_MTE is implemented, then the Trace Buffer Unit generates an Unchecked access for each access to the trace buffer.

Note: This is the case even when a Tagged Normal memory type is accessed.

See also The Memory Tagging Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.3 Trace buffer Self-hosted mode

I_{TSNMY} When the Trace Buffer Unit is using Self-hosted mode, then:

- If the Effective value of TRBLIMITR_EL1.nVM is 0, then address translation for the trace buffer in the owning translation regime is enabled. See Behavior when address translation is enabled.
- If the Effective value of TRBLIMITR_EL1.nVM is 1, then address translation for the trace buffer in the owning translation regime is disabled. See Behavior when address translation is disabled.

R_{FTWWP} If all of the following apply, then the Effective value of TRBLIMITR_EL1.nVM is 0:

- FEAT_TRBEv1p1 is implemented.
- Self-hosted trace is enabled.
- EL2 is implemented and enabled in the owning Security state.
- The owning Exception level is EL1.
- The Effective value of TRFCR_EL2.DnVM is 1.

D6.3.1 Behavior when address translation is enabled

R_{XRNCQ} If the Effective value of TRBLIMITR_EL1.nVM is 0, then the *Base pointer*, *Limit pointer*, and *current write pointer* are virtual addresses in the stage 1 translation regime of the owning translation regime.

R_{CMDTG} If the Effective value of TRBLIMITR_EL1.nVM is 0, other than as described in this section, then the stage 1 translation process for translating virtual addresses and checking for MMU faults is identical to that for any other virtual address in the owning translation regime.

R_{NPFBJ} The effect of PSTATE.PAN is not applied to the stage 1 Base permissions.

R_{BTSXD} The effects of FEAT_S1POE stage 1 Overlay permissions and POR_ELx are not applied to the memory access.

I_{GKBYK} If the Effective value of TRBLIMITR_EL1.nVM is 0, then R_{CMDTG} means all of the following apply:

- The virtual addresses are translated to stage 1 output addresses by stage 1 translation, and checked for stage 1 MMU faults. The stage 1 output addresses are:
  - Physical address in the owning Security state if the owning translation regime has no stage 2 translation.
  - Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.
- If stage 1 translation is enabled for the owning translation regime, the memory type and, as applicable, Cacheability, Shareability, and Device type attributes, for stage 1 output addresses are defined by the translation table entries for the virtual address being written to.
- If stage 1 translation is disabled for the owning translation regime, the memory type of the stage 1 output addresses is Device-nGnRnE, unless overridden by stage 2 controls.
- If SCTLR_ELx.C is 0 for the owning translation regime and stage 1 translation is enabled, then all accesses to Normal memory are Non-cacheable.
- TRBPTR_EL1[63:56] are ignored by address translation if the respective TBI bit is 1.

R_{SJFRQ} When the Trace Buffer Unit is enabled and collection is not stopped, the Trace Buffer Unit might cause speculative translations for the owning translation regime, including when the owning translation regime is out-of-context.

I_{QXJZX} R_{SJFRQ} means that, when the Trace Buffer Unit is enabled and the owning Exception level is a lower Exception level, then the Trace Buffer Unit might make memory accesses to translation table entries from the translation regime of the owning Exception level, using the settings of the System registers associated with that translation regime.

If the PE is not executing in the owning Security state, or the PE is executing at EL3 and SCR_EL3.{NSE, NS} does not indicate the owning Security state, then the translation regime of the owning Exception level might not be the owning translation regime.

These memory accesses might be observed by other observers, to the extent that those accesses are required to be observed as determined by the Shareability and Cacheability of those translation table entries.

This is an exception to the rules in the section Speculative memory accesses from out-of-context translation regimes.

See also Context switching.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.3.2 Behavior when address translation is disabled

R_{PBZRZ} If the Effective value of TRBLIMITR_EL1.nVM is 1, the *Base pointer*, *Limit pointer*, and *current write pointer* are:

- Physical addresses in the owning Security state if the owning translation regime has no stage 2 translation.
- Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.

These addresses are output directly by stage 1 without any address translation. Stage 1 translation is disabled for writes to the trace buffer made by the Trace Buffer Unit.

R_{FJKLW} If the Effective value of TRBLIMITR_EL1.nVM is 1, then unless overridden by stage 2 controls, TRBMAR_EL1 defines the memory type, and, as applicable, Cacheability, Shareability, and Device type attributes, for the stage 1 output addresses.

I_{GLNHS} If the Effective value of TRBLIMITR_EL1.nVM is 1, the values of SCTLR_ELx.{C, M} for the owning translation regime are ignored for the purposes of determining the trace buffer Cacheability attributes.

S_{ZMPXW} If the Effective value of TRBLIMITR_EL1.nVM is 1 it is possible to generate mismatched attributes for a location from within the same stage 1 translation regime, using TRBMAR_EL1.

Software must be aware of the consequences of and permitted behaviors when accessing a memory location with mismatched attributes. For more information, including a full definition of *mismatched attributes* and the permitted behaviors, see Mismatched memory attributes.

R_{MXRFD} If the Effective value of TRBLIMITR_EL1.nVM is 1, `OAMax` is greater than or equal to `AArch64_PAMax()`, and TRBPTR_EL1[`OAMax`:`AArch64_PAMax()`] is nonzero, then a stage 1 Address Size fault is generated when the Trace Buffer Unit attempts to write to memory, where:

- If FEAT_D128 is implemented, then `OAMax` is 55.
- Otherwise, if FEAT_LPA is implemented or FEAT_LPA2 is implemented , then `OAMax` is 51.
- Otherwise, `OAMax` is 47.

R_{BRRRK} If the Effective value of TRBLIMITR_EL1.nVM is 1 and TRBPTR_EL1[63:(`OAMax`+1)] is nonzero when the Trace Buffer Unit attempts to write to the trace buffer, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- A stage 1 Address Size fault is generated.
- TRBPTR_EL1[63:(`OAMax`+1)] are ignored and treated as zero.

The value of `OAMax` is as defined by R_{MXRFD}.

D6.3.3 Effect of stage 2 translation

R_{JCMKS} If the owning translation regime has stage 2 translations, the stage 2 process of translating the stage 1 output intermediate physical addresses and attributes to a physical address and attributes, and checking for MMU faults, is identical to that for any other intermediate physical address generated by the owning translation regime.

I_{ZSDMR} For example:

- The intermediate physical addresses are translated to physical addresses by stage 2 translation, and checked for stage 2 MMU faults.
- If the Effective value of TRBLIMITR_EL1.nVM is 1, meaning stage 1 translation is disabled, and the resulting IPA is translated by a stage 2 Block or Page descriptor with the AssuredOnly attribute set to 1, then the access translated by that descriptor generates a stage 2 Permission fault.
- The attributes from stage 1 are combined with the attributes from the stage 2 translation to generate the physical memory attributes.
- If the Effective value of HCR_EL2.DC in the owning translation regime is 1, then stage 1 translation is disabled and the memory type produced by stage 1 is Normal Non-shareable, Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate, regardless of the values of SCTLR_EL1.C and TRBMAR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.3.4 Accesses to the trace buffer

R_{JTVDD} Writes to the trace buffer by the Trace Buffer Unit are privileged writes within the owning translation regime.

I_{LDVQH} If ELx stage 1 translates an address and stage 1 Indirect permissions are used, then the Trace Buffer Unit indirectly reads PIR_ELx.

I_{BMKDC} If stage 2 translates an address and stage 2 Overlay permissions are used, the effect of FEAT_S2POE stage 2 Overlay permissions apply to the memory access and the Trace Buffer Unit indirectly reads S2POR_EL1.

I_{TTKZQ} For accesses made by the Trace Buffer Unit, the memory type and, as applicable, Cacheability, Shareability, and Device type attributes are determined by the translation tables or TRBMAR_EL1.

See:

- R_{CMDTG} and I_{GKBYK}, if translation is enabled.
- R_{FJKLW}, if translation is disabled.
- R_{JCMKS} and I_{ZSDMR}, if the owning translation regime has stage 2 translations.

R_{FBKCC} From Armv9.3, hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is controlled in the same way as explicit memory accesses in the trace buffer owning translation regime. Otherwise, it is IMPLEMENTATION DEFINED whether hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is controlled in the same way as explicit memory accesses in the trace buffer owning translation regime. This is discoverable by software using TRBIDR_EL1.F. See Hardware management of the dirty state and Hardware management of the Access flag.

R_{TNLBZ} If FEAT_HAFT is implemented, for an architecturally executed memory access that is translated by a translation stage with Table descriptor Access flag hardware management enabled, the Trace Buffer Unit is required to set the Access flag (AF) to 1 in all Table descriptors accessed during the translation table walk for that access that have the AF set to 0.

R_{SHWSL} If all of the following apply, the Trace Buffer Unit can speculatively update the translation table descriptor for any Page or Block in the trace buffer before writing data to it:

- Hardware management of dirty state by the Trace Buffer Unit is implemented.
- Hardware management of dirty state is enabled for the owning translation regime.
- The write is otherwise permitted.

This includes the case where a trace buffer management event means the Trace Buffer Unit stops writing data before the Page or Block is written to.

R_{BWNRF} The access granule for writes to the trace buffer by the Trace Buffer Unit is IMPLEMENTATION DEFINED, up to a maximum of 2KB, and might vary from time to time.

R_{CMSNC} Writes to any Device memory type by the Trace Buffer Unit occur once.

R_{RZTDD} A memory access from the Trace Buffer Unit that crosses a Page or Block boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- Each memory access generated by the Trace Buffer Unit uses the memory type and Shareability attribute associated with its own address.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation generated the mismatch, or there is only one stage of translation in the owning translation regime, the resulting trace buffer management event is a stage 1 Data Abort.
  - If only the stage 2 translation generated the mismatch, the resulting trace buffer management event is a stage 2 Data Abort.
  - If both stages of translation generate the mismatch, the resulting trace buffer management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- The trace data is discarded and the *current write pointer* might not be updated.

R_{XRQTN} A memory access from the Trace Buffer Unit to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Each memory accesses generated by the Trace Buffer Unit is performed as if the boundary has no effect on the memory accesses.
- Each memory accesses generated by the Trace Buffer Unit is performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between it and other memory accesses.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation causes the boundary to be crossed, or there is only one stage of translation in the owning translation regime, the resulting trace buffer management event is a stage 1 Data Abort.
  - If only the stage 2 translation causes the boundary to be crossed, the resulting trace buffer management event is a stage 2 Data Abort.
  - If both stages of translation cause the boundary to be crossed, the resulting trace buffer management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- The trace data is discarded and the *current write pointer* might not be updated.

Note: The boundary is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

I_{VKQBR} Although the Trace Buffer Unit behaves as if trace data is written a byte at a time, it is not required to do so.

For example, R_{BWNRF} and R_{CMSNC} mean that if the memory type for the trace buffer is Device-nGnRnE, then all of the following apply:

- Writes are not repeated and not reordered.
- A write Completes only after it reaches its endpoint in the memory system.
- The access granule size at the endpoint in the memory system is not defined by the architecture. However, a specific implementation might define the granule to permit interoperability with specific devices.

The access granule is not required to be fixed. For example, the Trace Buffer Unit might output a smaller granule when flushing trace data to the trace buffer.

See also I_{QQKZF}.

D6.3.5 The owning translation regime

R_{DPGJG} The owning translation regime is defined by the owning Security state and the owning Exception level.

R_{HBZNT} When the Trace Buffer Unit is enabled, the owning Security state is:

- Non-secure state if and only if at least one of the following is true:
  - EL3 is not implemented and the PE executes in Non-secure state.
  - The Effective value of MDCR_EL3.NSTBE is 0 and MDCR_EL3.NSTB is either `0b10` or `0b11`.
- Secure state if and only if Secure state is implemented and at least one of the following is true:
  - EL3 is not implemented and the PE executes in Secure state.
  - The Effective value of MDCR_EL3.NSTBE is 0 and MDCR_EL3.NSTB is either `0b00` or `0b01`.
- Realm state if and only if FEAT_RME is implemented, MDCR_EL3.NSTBE is 1, and MDCR_EL3.NSTB is either `0b10` or `0b11`.

R_{SKVWG} When the Trace Buffer Unit is enabled, the owning Exception level is:

- EL1 if and only if at least one of the following is true:
  - EL2 is not implemented in the owning Security state.
  - EL2 is disabled in the owning Security state.
  - MDCR_EL2.E2TB is either `0b10` or `0b11`.
- EL2 if and only if all of the following is true:
  - EL2 is implemented and enabled in the owning Security state.
  - MDCR_EL2.E2TB is `0b00`.

R_{XWDZV} When the Trace Buffer Unit is enabled and the owning Exception level is EL1, all of the following apply:

- The owning translation regime is EL1&0.
- If the Effective value of TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL1&0 translation regime using the current ASID from TTBRx_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of TRBLIMITR_EL1.nVM is 1, the trace buffer pointer addresses are intermediate physical addresses.
- Intermediate physical addresses (whether from the output of stage 1, or the pointers, as applicable) are subject to stage 2 translation using the current VMID if EL2 is implemented and enabled and HCR_EL2.VM is 1.
- If the Trace Buffer Unit is using Self-hosted mode, then the following are prohibited trace regions:
  - EL3.
  - EL2.
  - EL0, if EL2 is implemented and enabled and HCR_EL2.TGE is 1.

R_{SHXTV} When the Trace Buffer Unit is enabled and the owning Exception level is EL2, all of the following apply:

- If the Effective value of HCR_EL2.E2H is 0, the owning translation regime is EL2.
- If the Effective value of HCR_EL2.E2H is 1, the owning translation regime is EL2&0.
- If the Effective value of HCR_EL2.E2H is 0 and the Effective value of TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL2 translation regime.
- If the Effective value of HCR_EL2.E2H is 1 and the Effective value of TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL2&0 translation regime using the current ASID from TTBRx_EL2.
- If the Effective value of TRBLIMITR_EL1.nVM is 1, the trace buffer pointer addresses are physical addresses.
- If the Trace Buffer Unit is using Self-hosted mode, then EL3 is a prohibited trace region.

I_{QCKVZ} Table D6-2 summarizes the owning translation regime. In this table:

Enabled
is the value of the function `TraceBufferEnabled()`.

NSTBE
is the Effective value of MDCR_EL3.NSTBE.

NSTB
is the Effective value of MDCR_EL3.NSTB.

E2TB
is the Effective value of MDCR_EL2.E2TB.

EEL2
is the Effective value of SCR_EL3.EEL2.

E2H
is the Effective value of HCR_EL2.E2H.

The pseudocode function `TraceBufferOwner` describes this.

Table D6-2 Summary of owning translation regime (for all Exception levels using AArch64 state)

Enabled	NSTBE	NSTB	E2TB	EEL2	E2H	Owning translation regime
FALSE	x	x	x	x	x	Disabled
TRUE	0b0	0b0x	x	0b0	x	Secure EL1&0
TRUE	0b0	0b0x	0b00	0b1	0b0	Secure EL2
TRUE	0b0	0b0x	0b00	0b1	0b1	Secure EL2&0
TRUE	0b0	0b0x	0b1x	0b1	x	Secure EL1&0
TRUE	0b0	0b1x	0b00	x	0b0	Non-secure EL2
TRUE	0b0	0b1x	0b00	x	0b1	Non-secure EL2&0
TRUE	0b0	0b1x	0b1x	x	x	Non-secure EL1&0
TRUE	0b1	0b1x	0b00	x	0b0	Realm EL2
TRUE	0b1	0b1x	0b00	x	0b1	Realm EL2&0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Enabled	NSTBE	NSTB	E2TB	EEL2	E2H	Owning translation regime
TRUE	`0b1`	`0b1x`	`0b1x`	x	x	Realm EL1&0

R_{MFFGX} When the Trace Buffer Unit is enabled and the owning Security state is Non-secure state, Secure state and Realm state are prohibited trace regions.

R_{VGWJN} When the Trace Buffer Unit is enabled and the owning Security state is Secure state, Non-secure state and Realm state are prohibited trace regions.

R_{RRBFN} When the Trace Buffer Unit is enabled and the owning Security state is Realm state, Non-secure state and Secure state are prohibited trace regions.

I_{DCRYN} FEAT_TRF provides additional controls to define Trace Prohibited regions.

The following table summarizes the Trace Prohibited regions, by Exception level and state, when all of the following apply:

- `TraceBufferEnabled()` is TRUE.
- EL3, Non-secure EL2, and Secure EL2 are all implemented.
- EL3 is using AArch64.
- FEAT_RME is implemented.

In this table:

NSE
is the Effective value of SCR_EL3.NSE.

NS
is the Effective value of SCR_EL3.NS.

RLTE
is the Effective value of MDCR_EL3.RLTE.

STE
is the Effective value of MDCR_EL3.STE.

NSTBE
is the Effective value of MDCR_EL3.NSTBE.

NSTB
is the Effective value of MDCR_EL3.NSTB.

E2TB
is the Effective value of MDCR_EL2.E2TB.

EEL2
is the Effective value of SCR_EL3.EEL2.

TGE
is the Effective value of HCR_EL2.TGE.

Reserved values are not shown in the table.

The EL3, EL2, EL1, EL0 columns show which control, if any, enables tracing at the Exception level. In these columns:

P
means tracing is prohibited.

E2TRE
means tracing is allowed if TRFCR_EL2.E2TRE is 1 and prohibited otherwise.

E1TRE
means tracing is allowed if TRFCR_EL1.E1TRE is 1 and prohibited otherwise.

E0HTRE
means tracing is allowed if TRFCR_EL2.E0HTRE is 1 and prohibited otherwise.

E0TRE
means tracing is allowed if TRFCR_EL1.E0TRE is 1 and prohibited otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The pseudocode function `TraceAllowed` describes this.

NSE	NS	RLTE	STE	NSTBE	NSTB	E2TB	EEL2	TGE	EL3	EL2	EL1	EL0
0b0	0b0	x	0b0	x	x	x	x	x	P	P	P	P
0b0	0b0	x	0b1	0b0	0b0x	x	0b0	x	P	n/a	E1TRE	E0TRE
0b0	0b0	x	0b1	0b0	0b0x	0b00	0b1	0b0	P	E2TRE	E1TRE	E0TRE
0b0	0b0	x	0b1	0b0	0b0x	0b00	0b1	0b1	P	E2TRE	n/a	E0HTRE
0b0	0b0	x	0b1	0b0	0b0x	0b1x	0b1	0b0	P	P	E1TRE	E0TRE
0b0	0b0	x	0b1	0b0	0b0x	0b1x	0b1	0b1	P	P	n/a	P
0b0	0b0	x	0b1	x	0b1x	x	x	x	P	P	P	P
0b0	0b1	x	x	0b0	0b0x	x	x	x	P	P	P	P
0b0	0b1	x	x	0b0	0b1x	0b00	x	0b0	P	E2TRE	E1TRE	E0TRE
0b0	0b1	x	x	0b0	0b1x	0b00	x	0b1	P	E2TRE	n/a	E0HTRE
0b0	0b1	x	x	0b0	0b1x	0b1x	x	0b0	P	P	E1TRE	E0TRE
0b0	0b1	x	x	0b0	0b1x	0b1x	x	0b1	P	P	n/a	P
0b0	0b1	x	x	0b1	0b1x	x	x	x	P	P	P	P
0b1	0b1	0b0	x	x	x	x	x	x	P	P	P	P
0b1	0b1	0b1	x	0b0	x	x	x	x	P	P	P	P
0b1	0b1	0b1	x	0b1	0b1	0b00	x	0b0	P	E2TRE	E1TRE	E0TRE
0b1	0b1	0b1	x	0b1	0b1	0b00	x	0b1	P	E2TRE	n/a	E0HTRE
0b1	0b1	0b1	x	0b1	0b1	0b1x	x	0b0	P	P	E1TRE	E0TRE
0b1	0b1	0b1	x	0b1	0b1	0b1x	x	0b1	P	P	n/a	P

R_{MCYDC} When the Trace Buffer Unit is disabled or not using Self-hosted mode, the owning translation regime, owning Security state, and owning Exception level are not defined.

D6.3.6 Cache and TLB operations

I_{WYVXK} Translations used by the Trace Buffer Unit might be cached in a TLB.

R_{GQJMC} TLB maintenance operations that affect the TLB of the PE also affect any TLB caching translations for the Trace Buffer Unit of that PE.

R_{RNPNM} The PE is permitted, but not required, to cache all translations used by the Trace Buffer Unit in TLB caching structures that combine stage 1 and stage 2 of the translation. This includes when the Effective value of TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations.

S_{LSZDR} When the Effective value of TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations, the Trace Buffer Unit uses intermediate physical addresses (IPAs). R_{RNPNM} permits, but does not require, such translations to be cached in a TLB in such a way that an `IPAS2` TLB maintenance operation is *not* sufficient to invalidate the cached copies. In this case, there is no VA for the translation.

If the Effective value of TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations, then the following code executed at EL2 or above is sufficient to invalidate all cached copies of the stage 2 translations used by the Trace Buffer Unit of the IPA held in `Xt` for the current VMID:

```
TLBI IPAS2E1, Xt
DSB
TLBI VMALLE1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Equivalent architectural requirements apply to the `IPAS2L` instruction, except that the only TLB entries that must be invalidated by an `IPAS2L` instruction are those that come from the final level of the translation table lookup.

Equivalent sequences guaranteed to invalidate all entries invalidated by the above code sequence can be used, such as `TLBI ALL` or `TLBI VMALLS1S2`.

R_{DNFWB} Cache maintenance operations that affect the caches of the PE also affect data caching by the Trace Buffer Unit of that PE.

I_{MZQPT} R_{GQJMC} and R_{DNFWB} mean that the completion of any cache or TLB maintenance instruction includes its completion on all Trace Buffer Units for PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction. See Synchronization litmus tests for more information.

D6.3.7 Self-hosted mode and MEC

R_{RKSNW} If FEAT_MEC is implemented and the Trace Buffer Unit is using Self-hosted mode, accesses made by the Trace Buffer Unit to the trace buffer are associated with a MECID that is determined by the owning translation regime, owning Security state, and owning Exception level. For more information, see Memory Encryption Contexts.

D6.3.8 Self-hosted mode and MPAM

R_{WXSYG} If FEAT_MPAM is implemented and the Trace Buffer Unit is using Self-hosted mode, then the MPAM information for accesses made by the Trace Buffer Unit to the trace buffer use the MPAM values of the *owning Exception level* and *owning Security state*.

I_{XLNWD} For example, if the owning Exception level is EL2 the trace buffer writes use MPAM2_EL2.PARTID_D, MPAM2_EL2.PMG_D, with **MPAM_NS** or **MPAM_SP** reflecting the owning Security state.

See also About MPAM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4 Trace buffer External mode

D6.4.1 The external Trace Buffer debug component

R_{RGHMG} The external Trace Buffer debug component is defined by the external views of all of the following:

- TRBBASER_EL1.
- TRBIDR_EL1.
- TRBLIMITR_EL1.
- TRBMAR_EL1.
- TRBMPAM_EL1, if implemented.
- TRBPTR_EL1.
- TRBSR_EL1.
- TRBTRG_EL1.
- Trace Buffer Management Registers, if implemented.

Where applicable, the external views are mapped to the equivalent System registers.

I_{LSZMG} The Trace Buffer Management Registers are those with offsets ranging from `0xF00` through `0xFFC`, as defined by Management registers and CoreSight compliance.

I_{QSMXY} The external Trace Buffer debug component is a separate component in the external debug interface.

I_{DCWTT} An external access to a *Trace Buffer Unit* register might return an error response if one or more of the following apply:

Off

The Core power domain is powered down or is in low-power state where the registers cannot be accessed.

DLK

FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

Note

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

OSLK

OSLSR_EL1.OSLK is `0b1`. The OS Lock is locked.

ETBAD

`AllowExternalTraceBufferAccess` == FALSE for the access. The access is disabled by MDCR_EL3.ETBAD.

R_{MXBCY} For reserved accesses to the external Trace Buffer debug component:

- If the Core power domain is powered down or is in a low-power state where the registers cannot be accessed, then the access returns an error response.
- Otherwise:
  - For addresses in the range `0xF00` through `0xFFC`, the access is RES0H.
  - For all other addresses, if any of the following conditions apply, then the access is either RES0H or returns an error response:

    DLK

    FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

    Note

    The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

    OSLK

    OSLSR_EL1.OSLK is `0b1`. The OS Lock is locked.

    ETBAD

    `AllowExternalTraceBufferAccess` == FALSE for the access. The access is disabled by MDCR_EL3.ETBAD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise, the access is RES0H.

I_{RFYXF} For more information on reserved accesses, see Access requirements for reserved and unallocated registers.

I_{ZTKSS} Table D6-4 shows the external Trace Buffer debug register map.

Table D6-4 Summary of external Trace Buffer debug registers

Offset	Name	Description	Access
0x000	TRBBASER_EL1	Trace Buffer Base Address Register	RW
0x008	TRBPTR_EL1	Trace Buffer Write Pointer Register	RW
0x010	TRBLIMITR_EL1	Trace Buffer Limit Address Register	RW
0x018	TRBSR_EL1	Trace Buffer Status/syndrome Register	RW
0x020	TRBTRG_EL1	Trace Buffer Trigger Counter Register	RW
0x028	TRBMAR_EL1	Trace Buffer Memory Attribute Register	RW
0x030	TRBIDR_EL1	Trace Buffer ID Register	RO
0x038	TRBCR	Trace Buffer Control Register	RW
0x040	TRBMPAM_EL1	Trace Buffer MPAM Configuration Register	RW
0xF00	TRBITCTRL	Integration Mode Control Register	RW
0xFA8	TRBDEVAFF	Device Affinity Register	RO
0xFB0	TRBLAR	Lock Access Register	WO
0xFB4	TRBLSR	Lock Status Register	RO
0xFB8	TRBAUTHSTATUS	Authentication Status Register	RO
0xFBC	TRBDEVARCH	Trace Buffer Device Architecture Register	RO
0xFC0	TRBDEVID2	Device Configuration Register 2	RO
0xFC4	TRBDEVID1	Device Configuration Register 1	RO
0xFC8	TRBDEVID	Device Configuration Register	RO
0xFCC	TRBDEVTYPE	Device Type Register	RO
0xFD0	TRBPIDR4	Peripheral Identification Register 4	RO
0xFD4	TRBPIDR5	Peripheral Identification Register 5	RO
0xFD8	TRBPIDR6	Peripheral Identification Register 6	RO
0xFDC	TRBPIDR7	Peripheral Identification Register 7	RO
0xFE0	TRBPIDR0	Peripheral Identification Register 0	RO
0xFE4	TRBPIDR1	Peripheral Identification Register 1	RO
0xFE8	TRBPIDR2	Peripheral Identification Register 2	RO
0xFEC	TRBPIDR3	Peripheral Identification Register 3	RO
0xFF0	TRBCIDR0	Component Identification Register 0	RO
0xFF4	TRBCIDR1	Component Identification Register 1	RO
0xFF8	TRBCIDR2	Component Identification Register 2	RO
0xFFC	TRBCIDR3	Component Identification Register 3	RO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4.2 Behavior in External mode

R_{DFCSS} The rules in this section apply only when the Trace Buffer Unit is enabled and using External mode.

R_{NHTSY} The *Base pointer*, *Limit pointer*, and *current write pointer* are physical addresses.

R_{XXFVW} TRBMAR_EL1 defines the memory type, and, as applicable, Cacheability, Shareability, and Device type attributes, and the physical address space for the physical addresses.

R_{WNCWG} The IMPLEMENTATION DEFINED authentication interface controls which physical address spaces can be used.

R_{CMYYF} If the authentication interface prohibits invasive debug of the Security state corresponding to the physical address space selected by TRBMAR_EL1.PAS, or TRBMAR_EL1.PAS is set to a reserved value, then when the Trace Buffer Unit receives trace data from the trace unit, no trace is written to the trace buffer and all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- If TRBSR_EL1.S is 0, then all of the following occur:
  - TRBSR_EL1.S is set to 1, collection is stopped.
  - TRBSR_EL1.EC is set to `0x00`, other buffer management event.
  - TRBSR_EL1.BSC is set to `0b000000`, *access not allowed*.
- The other fields in TRBSR_EL1 are unchanged.

I_{PGLYR} The trace unit defines the trace prohibited regions.

When the Trace Buffer Unit is using Self-hosted mode, EL3 is always a prohibited region. This is not the case when the Trace Buffer Unit is using External mode.

R_{WMPMV} A write of 1 to TRBCR.ManStop generates a trace unit flush, and on completion of the trace unit flush all of the following apply:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- If TRBSR_EL1.S is 0, then all of the following occur:
  - TRBSR_EL1.S is set to 1, collection is stopped.
  - TRBSR_EL1.EC is set to `0x00`, other buffer management event.
  - TRBSR_EL1.BSC is set to `0b000011`, *Manual Stop event*.
- The other fields in TRBSR_EL1 are unchanged.

After the trace buffer management event is generated:

- The Trace Buffer Unit writes all trace data it has accepted from the trace unit to memory, adding padding data if necessary.
- The *current write pointer* points to one of the following:
  - If the last trace byte written to the trace buffer was the last byte in the trace buffer, then the *Base pointer*.
  - Otherwise, the first byte after the last trace byte written to the trace buffer.

I_{YBKZD} Setting TRBCR.ManStop to 1 has the effects described in R_{WMPMV} even if collection is stopped. Other than a Trigger Event, other trace buffer management events do not guarantee a trace unit flush, and following any other management events, including a Trigger Event, setting TRBCR.ManStop to 1 is required to force the Trace Buffer Unit to write all trace data to memory.

I_{KMQKF} When stopping trace at the end of a trace session using External mode, software does the following:

1. Stops and disables the trace unit.
2. Sets TRBCR.ManStop to 1.
3. Polls for TRBSR_EL1.{DAT, S} = {0, 1}.
4. Disables the Trace Buffer Unit.

To ensure the Trace Buffer Unit writes are Complete, the software is required to issue a `DSB` instruction. This might require an external debugger to halt the PE to execute the instruction. See Cross-halt trigger event.

I_{BSXVZ} If TRBPTR_EL1[63:(`OAMax`+1)] is nonzero when the Trace Buffer Unit attempts to write to the trace buffer, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- A stage 1 Address Size fault is generated.
- TRBPTR_EL1[63:(`OAMax`+1)] are ignored and treated as zero.

The value of `OAMax` is as defined by R_{MXRFD}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4.3 External mode and the Realm Management Extension

R_{WGMTL} When all of the following are true, each physically addressed access made by the Trace Buffer Unit is subject to Granule Protection Checks:

- FEAT_RME is implemented.
- The Trace Buffer Unit is using Self-hosted mode or `ExternalRootInvasiveDebugEnabled()` is FALSE.
- GPCCR_EL3.GPC is 1.

Otherwise, each physically addressed access made by the Trace Buffer Unit is not subject to Granule Protection Checks.

R_{TYJGF} When all the following are true, the Trace Buffer Unit rejects trace:

- FEAT_RME is implemented.
- The Trace Buffer Unit is using External mode and `ExternalRootInvasiveDebugEnabled()` is FALSE.
- GPCCR_EL3.{TBGPCD, GPC} is {0, 0}.

I_{LJVQH} If FEAT_RME is implemented, `ExternalRootInvasiveDebugEnabled()` is FALSE, and the Trace Buffer Unit is using External mode, then on a Warm reset, the PE enters a trace prohibited region and trace remains prohibited until the PE has left Root state and entered a trace allowed region.

I_{BWDGK} GPCCR_EL3.TBGPCD provides firmware with a control to allow trace to be written to physically addressed memory without Granule Protection Checks even when `ExternalRootInvasiveDebugEnabled()` is FALSE and the Trace Buffer Unit is using External mode. For example, if the system firmware will not enable Granule Protection Checks.

If GPCCR_EL3.GPC is 1, `ExternalRootInvasiveDebugEnabled()` is TRUE or the Trace Buffer Unit is using Self-hosted mode, then GPCCR_EL3.TBGPCD is ignored.

GPCCR_EL3.TBGPCD is reset to 0 on a Cold reset and preserved on a Warm reset.

D6.4.4 External mode and MEC

I_{YPPVQ} R_{RKSNW} defines the interaction of Self-hosted mode and MEC.

R_{MVPSY} If FEAT_MEC is implemented and the Trace Buffer Unit is using External mode, then accesses made by the Trace Buffer Unit to the trace buffer use the default MECID of zero.

D6.4.5 External mode and MPAM

R_{SFTXK} If FEAT_MPAM is implemented, FEAT_TRBE_MPAM is not implemented, and the Trace Buffer Unit is using External mode, then accesses made by the Trace Buffer Unit to the trace buffer use the Default physical PARTID and Default PMG values, and the PARTID space corresponding to the trace physical address space defined by TRBMAR_EL1.PAS, regardless of the value of MPAMn_ELx.MPAMEN.

R_{GJFRD} If FEAT_MPAM and FEAT_TRBE_MPAM are implemented and the Trace Buffer Unit is using External mode, then accesses made by the Trace Buffer Unit to the trace buffer use one of the following:

- If TRBMPAM_EL1.EN is 0, the Default physical PARTID and Default PMG values, and the PARTID space corresponding to the trace physical address space defined by TRBMAR_EL1.PAS, regardless of the value of MPAMn_ELx.MPAMEN.
- If TRBMPAM_EL1.EN is 1, then the TRBMPAM_EL1.{PARTID, PMG, MPAM_SP} MPAM values.

I_{JRQWG} The external debugger can choose any PARTID and PMG values from the selected the MPAM_SP PARTID space.

I_{QSRZX} System register access to TRBMPAM_EL1 is also provided, to allow software to save and restore the register over power-down. TRBMPAM_EL1 is not used in Self-hosted mode.

R_{ZXFRR} The IMPLEMENTATION DEFINED authentication interface controls which MPAM_SP PARTID spaces can be used.

R_{VVWKG} If the authentication interface prohibits invasive debug of the Security state corresponding to the MPAM_SP.PARTID space selected by TRBMPAM_EL1.MPAM_SP, or TRBMPAM_EL1.MPAM_SP is set to a reserved value, then when the Trace Buffer Unit receives trace data from the trace unit, no trace is written to the trace buffer and all of the following occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- If TRBSR_EL1.S is 0, then all of the following occur:
  - TRBSR_EL1.S is set to 1, collection is stopped.
  - TRBSR_EL1.EC is set to `0x00`, other buffer management event.
  - TRBSR_EL1.BSC is set to `0b000000`, *access not allowed*.
- The other fields in TRBSR_EL1 are unchanged.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.5 Trace buffer management

I_{GYHBH} FEAT_TRBE supports the following trace buffer modes:

Circular Buffer mode

In Circular Buffer mode, when the *current write pointer* reaches the *Limit pointer*, it is wrapped by setting it to the *Base pointer*.

Wrap mode

As Circular Buffer mode, except that an interrupt request is generated when the *current write pointer* is wrapped.

Fill mode

As Wrap mode, except that trace collection stops when the *current write pointer* is wrapped.

I_{FZXSV} The trace buffer mode is controlled by TRBLIMITR_EL1.FM.

I_{NFZKS} A trace buffer management event occurs:

- On an Alignment fault or MMU fault.
- On an External abort.
- On a *Trigger Event*, if enabled.
- When the *current write pointer* is wrapped to the *Base pointer* and the trace buffer mode is not Circular Buffer mode. This event is known as:
  - A buffer wrap event, if the trace buffer mode is Wrap mode.
  - A buffer full event, if the trace buffer mode is Fill mode.
- On a programming error, when permitted as an UNPREDICTABLE behavior of the PE. For more information, see Restrictions on programming the Trace Buffer Unit and UNPREDICTABLE behavior.
- On an IMPLEMENTATION DEFINED event.

Note

The trace buffer management event behavior differs from that of the SPE Profiling Buffer management event.

R_{HLKSG} On a trace buffer management event, all of the following occurs:

- The interrupt request bit, TRBSR_ELx.IRQ, is set to 1.
- Additional syndrome for the event might be written to TRBSR_ELx.MSS.

The value of ELx depends on the type of event, and is defined by R_{BNQRP}. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

R_{BNQRP} If FEAT_TRBE_EXC is implemented and self-hosted trace is enabled, then when a trace buffer management event is generated:

- If EL3 is implemented and any of the following are true, then the management event is recorded in TRBSR_EL3:
  - MDCR_EL3.TRBEE is `0b11`.
  - MDCR_EL3.TRBEE is `0b10` and the management event is due to a fault on a write to the trace buffer that would generate a Data Abort exception taken to EL3 if generated by a store instruction executed at the owning Exception level. That is, any of the following faults on a write to the trace buffer:
    - Granule Protection Check (GPC) faults, other than Granule Protection Faults (GPFs). For more information, see Granule Protection Check faults.
    - If SCR_EL3.GPF is 1, GPFs.
    - If SCR_EL3.EA is 1, External aborts. See also I_{FRNZK}.
  - MDCR_EL3.TRBEE is `0b10` and the management event is an IMPLEMENTATION DEFINED buffer management event that the implementation elects to record in TRBSR_EL3.
- Otherwise, if all of the following are true, then the management event is recorded in TRBSR_EL2:
  - EL3 is not implemented or MDCR_EL3.TRBEE is not `0b00`.
  - EL2 is implemented and any of the following are true:
    - The Effective value of SCR_EL3.NS is 1.
    - The Effective value of SCR_EL3.{NS, EEL2} is {0, 1}.

    This applies even when the current Exception level is EL3.
  - One of the following is true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— TRFCR_EL2.EE is `0b11`.
— TRFCR_EL2.EE is `0b10` and the management event is due to any fault on a write to the trace buffer that would generate a Data Abort exception taken to EL2 if generated by a store instruction executed at the owning Exception level.
If the owning Exception level is EL2, this means any fault on a write to the trace buffer.
If the owning Exception level is EL1, this means any of the following faults on a write to the trace buffer:
· Stage 2 faults.
· If HCR_EL2.GPF is 1, GPFs.
· If HCR_EL2.TEA is 1, External aborts. See also I_{FRNZK}.
— TRFCR_EL2.EE is `0b10` and the management event is due to a GPC fault other than a GPF.
— TRFCR_EL2.EE is `0b10` and the management event is an IMPLEMENTATION DEFINED buffer management event that the implementation elects to record in TRBSR_EL2.

• Otherwise, the management event is recorded in TRBSR_EL1.

If FEAT_TRBE_EXC is not implemented or self-hosted trace is disabled, then the management event is recorded in TRBSR_EL1.
The pseudocode function `ReportTRBEEvent()` shows this.

I_{FRNZK} From Armv9.3, External aborts are not reported on writes to the trace buffer. However, External aborts on translation table walks, translation table updates, and GPT walks can be reported as MMU faults. See Faults.

I_{KKVYZ} Table D6-5, Table D6-6, and Table D6-7 show which TRBSR_ELx register used to record the syndrome for each type of trace buffer management event when self-hosted trace is enabled.
Each column in the left side of the tables refers to the Effective value of a field in a System register.
Each column in the right side of the tables represents a trace buffer management event:

Other event
Means a trace buffer management event recorded with the EC value `0b000000`.

Stage 1 (GPF)
Means a trace buffer management event for a Granule Protection Fault (GPF) reported as a stage 1 abort on a write to the buffer, recorded with the EC value of `0b100100` and an FSC value indicating a GPF.

Stage 1 (External abort)
Means a trace buffer management event for an External abort reported as a stage 1 abort on a write to the buffer, recorded with the EC value of `0b100100` and an FSC value indicating an External abort.

Stage 1 (Other abort)
Means a trace buffer management event for any other stage 1 abort on a write to the buffer, recorded with the EC value `0b100100`.

Stage 2 (GPF)
Means a trace buffer management event for a GPF reported as a stage 2 abort on a write to the buffer, recorded with the EC value of `0b100101` and an FSC value indicating a GPF.

Stage 2 (External abort)
Means a trace buffer management event for an External abort reported as a stage 2 abort on a write to the buffer, recorded with the EC value of `0b100101` and an FSC value indicating an External abort. That is, a synchronous External abort on a stage 2 translation table walk or translation table update.

Stage 2 (Other abort)
Means a trace buffer management event for any other stage 2 abort on a write to the buffer, recorded with the EC value `0b100101`.

GPC fault
Means a trace buffer management event for a Granule Protection Check fault, other than a GPF, on a write to the buffer, recorded with the EC value `0b011110`.

The values in each of these cells denotes which TRBSR_ELx register is updated for that combination of control register values and trace buffer management event type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D6-5 TRBE Profiling Buffer management event reporting (Granule Protection Checks)

MDCR_EL3.TRBEE	SCR_EL3.GPF	TRFCR_EL2.EE	MDCR_EL2.E2TB	HCR_EL2.GPF	GPF		GPC fault
					Stage 1	Stage 2	
0b00	X	XX	XX	X	EL1	EL1	EL1
0b01	X	0b0X	XX	X	EL1	EL1	EL1
0b01	X	0b10	0b00	X	EL2	EL2	EL2
0b01	X	0b10	0b1X	0b0	EL1	EL2	EL2
0b01	X	0b10	0b1X	0b1	EL2	EL2	EL2
0b01	X	0b11	XX	X	EL2	EL2	EL2
0b10	0b0	0b0X	XX	X	EL1	EL1	EL3
0b10	0b0	0b10	0b00	X	EL2	EL2	EL3
0b10	0b0	0b10	0b1X	0b0	EL1	EL2	EL3
0b10	0b0	0b10	0b1X	0b1	EL2	EL2	EL3
0b10	0b0	0b11	XX	X	EL2	EL2	EL3
0b10	0b1	XX	XX	X	EL3	EL3	EL3
0b11	X	XX	XX	X	EL3	EL3	EL3

Table D6-6 TRBE Profiling Buffer management event reporting (External aborts)

MDCR_EL3.TRBEE	SCR_EL3.EA	TRFCR_EL2.EE	MDCR_EL2.E2TB	HCR_EL2.TEA	External abort	
					Stage 1	Stage 2
0b00	X	XX	XX	X	EL1	EL1
0b01	X	0b0X	XX	X	EL1	EL1
0b01	X	0b10	0b00	X	EL2	EL2
0b01	X	0b10	0b1X	0b0	EL1	EL2
0b01	X	0b10	0b1X	0b1	EL2	EL2
0b01	X	0b11	XX	X	EL2	EL2
0b10	0b0	0b0X	XX	X	EL1	EL1
0b10	0b0	0b10	0b00	X	EL2	EL2
0b10	0b0	0b10	0b1X	0b0	EL1	EL2
0b10	0b0	0b10	0b1X	0b1	EL2	EL2
0b10	0b0	0b11	XX	X	EL2	EL2
0b10	0b1	XX	XX	X	EL3	EL3
0b11	X	XX	XX	X	EL3	EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D6-7 TRBE Profiling Buffer management event reporting (All other management events)

MDCR_EL3.TRBEE	TRFCR_EL2.EE	MDCR_EL2.E2TB	Other abort		Other event
			Stage 1	Stage 2	
0b00	XX	XX	EL1	EL1	EL1
0b01	0b0X	XX	EL1	EL1	EL1
0b01	0b10	0b00	EL2	EL2	EL1
0b01	0b10	0b1X	EL1	EL2	EL1
0b01	0b11	XX	EL2	EL2	EL2
0b10	0b0X	XX	EL1	EL1	EL1
0b10	0b10	0b00	EL2	EL2	EL1
0b10	0b10	0b1X	EL1	EL2	EL1
0b10	0b11	XX	EL2	EL2	EL2
0b11	XX	XX	EL3	EL3	EL3

R_{MFMXQ} When FEAT_TRBE_EXC is implemented, self-hosted trace is enabled, and any of the following are true, then the TRBE Profiling exception is *enabled* and pending:

- All of the following are true:
  - EL3 is implemented.
  - MDCR_EL3.TRBEE is `0b10` or `0b11`.
  - TRBSR_EL3.IRQ is 1.

  In this case:
  - The target Exception level is EL3.
  - If all of the following are true, then the TRBE Profiling exception is *unmasked*:
    - The PE is executing at EL2, EL1, or EL0.
    - The PE is in Non-debug state.
- All of the following are true:
  - EL3 is not implemented or MDCR_EL3.TRBEE is not `0b00`.
  - EL2 is implemented and enabled in the current Security State.
  - TRFCR_EL2.EE is `0b10` or `0b11`.
  - TRBSR_EL2.IRQ is 1.

  In this case:
  - The target Exception level is EL2.
  - If the PE is executing in the owning Security state, the PE is in Non-debug state, and any of the following are true, then the TRBE Profiling exception is unmasked:
    - The PE is executing at EL1 or EL0.
    - The PE is executing at EL2, TRFCR_EL2.EE is `0b11`, TRFCR_EL2.KE is 1, and PSTATE.PM is 0.
- All of the following are true:
  - EL3 is not implemented or MDCR_EL3.TRBEE is not `0b00`.
  - EL2 is not implemented, or EL2 is disabled in the current Security State, or TRFCR_EL2.EE is not `0b00`.
  - TRFCR_EL1.EE is `0b11`.
  - TRBSR_EL1.IRQ is 1.

  In this case:
  - The target Exception level is EL2 if the Effective value of HCR_EL2.TGE is 1, and EL1 otherwise.
  - If the PE is executing in the owning Security state, the PE is in Non-debug state, and any of the following are true, then the TRBE Profiling exception is unmasked.
    - The PE is executing at EL0.
    - The PE is executing at EL1, TRFCR_EL1.KE is 1, and PSTATE.PM is 0.

When self-hosted trace is disabled or none of the above apply, the TRBE Profiling exception is disabled and not pending.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the TRBE Profiling exception is enabled and unmasked, the TRBE Profiling exception will be taken to the target Exception level.

When the TRBE Profiling exception is enabled and not unmasked, then the TRBE Profiling exception is *masked*. The TRBE Profiling exception is always masked at EL3.

The pseudocode function `CheckForTRBEException()` shows this.

IMTLSX

Table D6-8, Table D6-9, and Table D6-10 show the TRBE Profiling exception enable and masking for each Exception level, for each of the TRBSR_EL1.IRQ, TRBSR_EL2.IRQ, and TRBSR_EL3.IRQ bits respectively when all of the following apply:

- FEAT_TRBE_EXC is implemented.
- The PE is executing in AArch64 state.
- Self-hosted trace is enabled.

Each column in the left side of the table refers to the Effective value of a field in a System register or PSTATE.

In Table D6-8, the IRQ column represents the state of the **TRBIRQ** signal. In these cells for each row representing a combination of control bits:

HIGH

Means the TRBE Profiling exception is disabled and the TRBE interrupt request is enabled and asserted.

LOW

Means the TRBE interrupt request is not asserted.

The other columns in the right side of each table represent an Exception level the exception is taken from. In each cell for each of these columns for each row representing a combination of control bits:

None

Means the exception is disabled. For Table D6-9 and Table D6-10 this means that the TRBSR_EL3.IRQ or TRBSR_EL2.IRQ bit (as applicable) is ignored

B

Means the exception is enabled and masked by PSTATE.PM. The exception remains pending.

C

Means the exception is enabled and masked regardless of the value of PSTATE.PM. The exception remains pending.

ELx

Means the exception is enabled, is not masked, and is taken to ELx.

n/a

Not applicable. The PE cannot be executing at this Exception level for the specified value of HCR_EL2.TGE.

Table D6-8 TRBE Profiling exception and interrupt, taken from EL0, EL1, EL2 or EL3, when the Effective value of TRBSR_EL1.IRQ is 1

MDCR_EL3. TRBEE	TRFCR_EL2. EE	HCR_EL2. TGE	TRFCR_EL1		PSTATE. PM	IRQ	Taken from			
			EE	KE			EL0	EL1	EL2	EL3
0b00	XX	0b0	XX	X	X	HIGH	None	None	None	None
0b00	XX	0b1	XX	X	X	HIGH	None	n/a	None	None
!=0b00	0b00	0b0	XX	X	X	HIGH	None	None	None	None
!=0b00	0b00	0b1	XX	X	X	HIGH	None	n/a	None	None
!=0b00	!=0b00	0b0	0b0X	X	X	HIGH	None	None	None	None
!=0b00	!=0b00	0b0	0b1X	0b0	X	LOW	EL1	C	C	C
!=0b00	!=0b00	0b0	0b1X	0b1	0b0	LOW	EL1	EL1	C	C
!=0b00	!=0b00	0b0	0b1X	0b1	0b1	LOW	EL1	B	C	C

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MDCR_EL3. TRBEE	TRFCR_EL2. EE	HCR_EL2. TGE	TRFCR_EL1		PSTATE. PM	IRQ	Taken from			
			EE	KE			EL0	EL1	EL2	EL3
!=0b00	!=0b00	0b1	0b0X	X	X	HIGH	None	n/a	None	None
!=0b00	!=0b00	0b1	0b1X	X	X	LOW	EL2	n/a	C	C

Table D6-9 TRBE Profiling exception, taken from EL0, EL1, EL2 or EL3, when the Effective value of TRBSR_EL2.IRQ is 1

MDCR_EL3.TRBEE	TRFCR_EL2		HCR_EL2.TGE	PSTATE.PM	Taken from			
	EE	KE			EL0	EL1	EL2	EL3
0b00	XX	X	0b0	X	None	None	None	None
0b00	XX	X	0b1	X	None	n/a	None	None
!=0b00	0b0X	X	0b0	X	None	None	None	None
!=0b00	0b0X	X	0b1	X	None	n/a	None	None
!=0b00	0b10	X	0b0	X	EL2	EL2	C	C
!=0b00	0b10	X	0b1	X	EL2	n/a	C	C
!=0b00	0b11	0b0	0b0	X	EL2	EL2	C	C
!=0b00	0b11	0b0	0b1	X	EL2	n/a	C	C
!=0b00	0b11	0b1	0b0	0b0	EL2	EL2	EL2	C
!=0b00	0b11	0b1	0b0	0b1	EL2	EL2	B	C
!=0b00	0b11	0b1	0b1	0b0	EL2	n/a	EL2	C
!=0b00	0b11	0b1	0b1	0b1	EL2	n/a	B	C

Table D6-10 TRBE Profiling exception, taken from EL0, EL1, EL2 or EL3, when the Effective value of TRBSR_EL3.IRQ is 1

MDCR_EL3.TRBEE	HCR_EL2.TGE	Taken from			
		EL0	EL1	EL2	EL3
0b0X	0b0	None	None	None	None
0b0X	0b1	None	n/a	None	None
0b10	0b0	EL3	EL3	EL3	C
0b10	0b1	EL3	n/a	EL3	C
0b11	0b0	EL3	EL3	EL3	C
0b11	0b1	EL3	n/a	EL3	C

I_{WKGWP} A TRBE Profiling exception is never generated when self-hosted trace is disabled. This includes when FEAT_TRBE_EXT is implemented and the Trace Buffer Unit is in External mode.

I_{PRSXC} By setting MDCR_EL3.TRBEE or TRFCR_EL2.EE to 0b01, software at EL3 or EL2 (respectively) allows lower Exception levels to use the TRBE Profiling exception without any trace buffer management event being recorded in TRBSR_EL3 or TRBSR_EL2, meaning no TRBE Profiling exceptions are taken to EL3 or EL2 (as applicable).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

However, this also means that the lower Exception level has visibility of when writes to the trace buffer cause a Granule Protection Check fault, Granule Protection Fault, or stage 2 MMU fault (as applicable). Note that this is also true in FEAT_TRBE when FEAT_TRBE_EXC is not implemented.

If both MDCR_EL3.TRBEE and TRFCR_EL2.EE are `0b01`, then all trace buffer management events are recorded in TRBSR_EL1, as in FEAT_TRBE when FEAT_TRBE_EXC is not implemented. Depending on the value of TRFCR_EL1.EE, the recorded event might generate a TRBE Profiling exception taken to EL1, or cause **TRBIRQ** to be asserted.

I_{HWBCW}

By setting MDCR_EL3.TRBEE and/or TRFCR_EL2.EE to `0b10`, software at EL3 and/or EL2 (respectively) allows lower Exception levels to use the TRBE Profiling exception or trace buffer interrupt. Only trace buffer management events related to memory management by that Exception level recorded in TRBSR_EL3 or TRBSR_EL2, with EL3 having priority. This means that only TRBE Profiling exceptions for those events are taken to EL3 or EL2.

This allows trace buffer management events for aborts on writes to the trace buffer to be routed directly to the same software that would handle a similar Data Abort exception on a store instruction. R_{BNQRP} describes which trace buffer management events these are.

Software at the higher Exception level can inject those events into a lower Exception level by writing TRBSR_ELx for the lower Exception level. For example, after triaging the event. This will generate either a TRBE Profiling exception or trace buffer interrupt, depending on the value of TRFCR_ELx.EE.

R_{BNQRP} means that a lower Exception level cannot create a fake trace buffer management event for a higher Exception level.

I_{CFYMJ}

If MDCR_EL3.TRBEE is `0b01` and TRFCR_EL2.EE is `0b10`, then a hybrid of I_{PRSXC} and I_{HWBCW} is used.

This allows EL3 software to enable use of FEAT_TRBE_EXC with minimal changes from FEAT_TRBE, with EL2 managing trace buffer management events that should not be handled by the Guest operating system.

I_{TCNHM}

By setting MDCR_EL3.TRBEE or TRFCR_EL2.EE to `0b11`, software at EL3 or EL2 (respectively) forces all trace buffer management events to be recorded in TRBSR_EL3 or TRBSR_EL2, meaning TRBE Profiling exceptions for trace buffer management events are taken to EL3 or EL2 (as applicable), with EL3 having priority.

Software at EL3 or EL2 can inject those events into a lower Exception level by writing TRBSR_ELy for the lower Exception level. For example, after triaging the event.

I_{QPKBN}

The target Exception level might be the same as the owning Exception level, or a higher Exception level, or a lower Exception level. For example:

- The owning Exception level is EL1, but EL2 triages all TRBE Profiling exceptions and so programs TRFCR_EL2.EE to `0b11`.
- The owning Exception level is EL2, because EL2 has mapped the trace buffer in the EL2 translation regime, but EL1 can handle all non-MMU management events, and so EL2 programs TRFCR_EL2.EE to `0b10`.

R_{YXTTY}

When a TRBE Profiling exception is enabled and unmasked, and all of the following are true, the TRBE Profiling exception will be taken before an instruction **C** completes execution:

- Instruction **A** is one of the following:
  - A direct read of the corresponding TRBSR_ELx register that observes that the IRQ bit is 1.
  - A direct write of the corresponding TRBSR_ELx register that sets the IRQ bit to 1.
  - A `TSB CSYNC` instruction that synchronizes a trace operation that causes a trace buffer management event that sets the corresponding TRBSR_ELx.IRQ bit to 1. For more information, see Synchronization and the Trace Buffer Unit.
- **C** follows in program order immediately after a Context synchronization event **B**.
- **B** is in program order after **A**.

Otherwise, the TRBE Profiling exception is taken in finite time.

I_{PZBKW}

TRBE Profiling exceptions are treated in the same way as other Profiling exceptions by FEAT_ETE and FEAT_BRBE, if implemented.

Note

A TRBE Profiling exception might be traced by the trace unit if, for example, the trace buffer management event set TRBSR_ELx.IRQ to 1 but left TRBSR_ELx.S unchanged, meaning trace collection is not stopped.

R_{VSCDV}

If any of the following are true, then the TRBE interrupt request (**TRBIRQ**) is enabled and driven from TRBSR_EL1.IRQ:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_TRBE_EXC is not implemented.
- EL3 is implemented and MDCR_EL3.TRBEE is `0b00`.
- EL2 is implemented, TRFCR_EL2.EE is `0b00`, and any of the following are true:
  - The Effective value of SCR_EL3.NS is 1.
  - The Effective value of SCR_EL3.{NS, EEL2} is {0, 1}.

  This applies even when the current Exception level is EL3.
- TRFCR_EL1.EE is `0b00`.
- Self-hosted trace is disabled.

Otherwise, if FEAT_TRBE_EXC is implemented, then the TRBE interrupt request is disabled and driven LOW (inactive).

The pseudocode function `TRBEInterruptEnabled()` shows this.

D_{LRTBP} **TRBIRQ** is a level-triggered interrupt request.

R_{TPPCF} When a GIC is implemented, **TRBIRQ** is configured as a PPI. **TRBIRQ** is signaled by the PE that implements the Trace Buffer Unit.

I_{HHRLX} The PPI number is not defined by the architecture. Arm recommends that the PPI number is discoverable to an Operating System, for example using ACPI or devicetree interfaces.

S_{XLNJY} Software must configure the trace buffer management interrupt to be taken to the correct Exception level.

I_{HJFLC} Buffer full, Alignment fault, and MMU fault trace buffer management events are synchronous. This means that the effect of these trace buffer management events setting TRBSR_EL1.S to 1, collection is stopped, happens before any further trace is collected by the Trace Buffer Unit from the trace unit.

I_{JLZDN} When the trigger mode is *Stop on Trigger*, a *Trigger Event* initiates a trace unit flush meaning more trace might be written to the trace buffer before any collection is stopped by the Trigger Event trace buffer management event. This additional trace might cause a second trace buffer management event to be generated before the Trigger Event trace buffer management event.

I_{ZLVHR} TRBE Profiling exceptions and the **TRBIRQ** interrupt are always taken asynchronously and precisely by the PE, even if the event is reported synchronously to the Trace Buffer Unit.

I_{GVGPB} Following an Alignment fault, MMU fault, or External abort trace buffer management event, TRBPTR_EL1 serves as a Fault Address Register.

For a fault or synchronous External abort trace buffer management event, the frozen TRBPTR_EL1 is the address that generated the fault or External abort.

For an asynchronous External abort trace buffer management event, the frozen TRBPTR_EL1 is not guaranteed to be the address that generated the External abort.

D6.5.1 Prioritization of a trace buffer management event

R_{MKCHT} Where multiple synchronous trace buffer management events occur on writing trace data, the PE prioritizes them from highest to lowest priority, reporting the highest priority event as follows:

1. Synchronous fault.
2. Synchronous External abort.
3. Buffer full event.
4. Buffer wrap event.

I_{BRLXK} Do not confuse the prioritization of trace buffer management events with the prioritization of trace buffer management interrupts by an interrupt controller.

R_{GTMJD} Asynchronous and IMPLEMENTATION DEFINED trace buffer management events are not prioritized relative to synchronous trace buffer management events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.5.2 Buffer full and buffer wrap events

R_{MBSHC} If the *current write pointer* is wrapped to the *Base pointer* and the trace buffer mode is Fill mode, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1.
- TRBSR_ELx.WRAP is set to 1.
- The TRB_WRAP event is generated.
- If TRBSR_ELx.S is 0, then all of the following occur:
  - TRBSR_ELx.S is set to 1, collection is stopped.
  - TRBSR_ELx.EC is set to `0x00`, other buffer management event.
  - TRBSR_ELx.BSC is set to `0b000001`, buffer filled.

ELx is defined by R_{BNQRP}. See also Table D6-7. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

After the trace buffer management event, the *current write pointer* will point to the *Base pointer*.

R_{VBDJZ} If the *current write pointer* is wrapped to the *Base pointer* and the trace buffer mode is Wrap mode, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1.
- TRBSR_ELx.WRAP is set to 1.
- The TRB_WRAP event is generated.

Because TRBSR_ELx.S is unchanged, trace continues to be collected and written to the trace buffer.

ELx is defined by R_{BNQRP}. See also Table D6-7. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

R_{GDSTS} If the *current write pointer* is wrapped to the *Base pointer* and the trace buffer mode is Circular Buffer mode, then all of the following occur:

- TRBSR_ELx.WRAP is set to 1, where ELx is the value in the *Other event* column of Table D6-7. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.
- The TRB_WRAP event is generated.

I_{VCDNP} If collection is stopped, the *current write pointer* is not updated, meaning the *current write pointer* is never wrapped when collection is stopped.

S_{HKNBM} Software can configure the PMU to count the TRB_WRAP event and monitor how many times the *current write pointer* has wrapped, particularly in Circular Buffer mode or Wrap mode.

I_{JSQPH} See also Controlling generation of trace buffer management events.

D6.5.3 Trigger event

I_{HHLBM} The Trace Buffer Extension supports detection of a trigger condition from the trace unit. A trigger condition is typically used to stop trace capture to ensure trace is captured around a point of interest.

The trace unit defines how software programs the trace unit to generate trigger conditions.

A *Detected Trigger* is signaled to the Trace Buffer Unit by the trace unit when the trace unit detects a trigger condition. The trace unit defines whether the *Detected Trigger* is signaled synchronously or asynchronously to the trace data stream.

A *Trigger Event* occurs when the *Trigger Counter* has counted the specified number of trace bytes after a *Detected Trigger*. Software can set the *Trigger Counter* to zero to skip this step.

The *Trigger Counter* is a counter used to delay a *Trigger Event* for a specified number of trace bytes after a *Detected Trigger*.

Figure D6-2 shows this.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

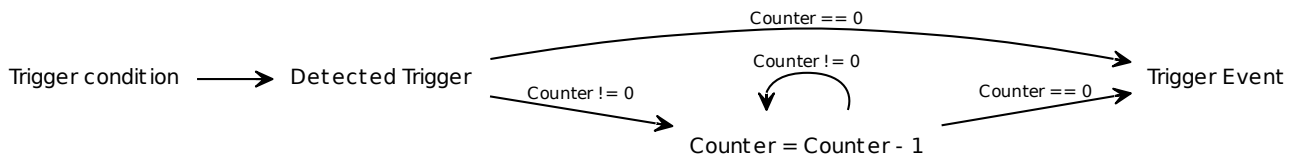

Figure D6-2 Trigger condition to Trigger Event

Figure D6-2 does not show all possible transitions. For example, software can also program TRBSR_EL1.TRG to 1 to indicate that a Detected Trigger has occurred. This might be used to start the trigger counter immediately after enabling the Trace Buffer Unit, or after restoring state due to a context switch.

I_{YRQCN} For a trace unit that implements FEAT_ETE, event 0 is the trigger condition.

S_{DHJTB} When the trigger mode is set to *Stop on trigger*, software uses the *Trigger Counter* to control how trace is collected around the *Detected Trigger* as follows:

- Set the *Trigger Counter* to zero to trace *before* the *Detected Trigger*.
- Set the *Trigger Counter* to half the size of the trace buffer to trace *around* the *Detected Trigger*.
- Set the *Trigger Counter* to the size of the trace buffer to trace *after* the *Detected Trigger*.

I_{KLRBB} Restrictions on programming the Trace Buffer Unit defines additional constraints on writing to the *Trigger Counter*.

R_{KSQJW} If a *Detected Trigger* occurs while the Trace Buffer Unit is enabled, then TRBSR_ELx.TRG is set to 1, where ELx is the value in the *Other event* column of Table D6-7. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

R_{BQTGW} If all of the following are true, then the Trace Buffer Unit decrements the *Trigger Counter* by 1 for each byte of trace data written to the trace buffer by the Trace Buffer Unit:

- The *Trigger Counter* is nonzero.
- TRBSR_ELx.TRG is set to 1, where ELx is the value in the *Other event* column of Table D6-7.

If the write generates an Alignment fault, MMU fault, or External abort, it is UNPREDICTABLE whether the *Trigger Counter* is decremented.

R_{QFGNQ} If a *Detected Trigger* occurs when the *Trigger Counter* is nonzero and TRBSR_ELx.TRG is 0, the Trace Buffer Unit might decrement the *Trigger Counter* by an amount up to the value specified by TRBIDR_EL1.Align without writing any additional trace data to the trace buffer. ELx is the value in the *Other event* column of Table D6-7.

I_{QQKZF} An implementation might include an internal buffer that collects bytes of trace data into more convenient units before writing them to memory. For example, the width of the system bus or the length of a cache line.

In such an implementation, TRBIDR_EL1.Align specifies the size of this unit, and the *Trigger Counter* is decremented by the size of this unit when the write occurs, meaning the *Trigger Counter* is always aligned to the size specified by TRBIDR_EL1.Align.

However, this means that if the *Detected Trigger* occurs when such an internal buffer is not empty, the Trace Buffer Unit will over-decrement the counter when the internal buffer is written to memory. R_{QFGNQ} permits this.

See also R_{BWNRF}.

R_{DHLQG} A *Trigger Event* is generated when the Trace Buffer Unit is enabled and one of the following occurs:

- A *Detected Trigger* occurs when the *Trigger Counter* is zero and TRBSR_ELx.TRG is 0. ELx is defined by the value in the *Other event* column of Table D6-7.
- The Trace Buffer Unit decrements the *Trigger Counter* to zero.

I_{ZGFSK} A *Trigger Event* is not generated when a *Detected Trigger* occurs, the *Trigger Counter* is set to zero. TRBSR_ELx.TRG is already 1, where ELx is the value in the *Other event* column of Table D6-7.

A *Trigger Event* is not generated when the Trace Buffer Unit is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{RWJNN} The *Detected Trigger* might be generated by a trace operation t_T. This might be the trace operation generated by an instruction that also matched the trigger condition, or might be a trace operation generated asynchronously by the trace unit to mark the trigger condition in the trace data. The trace unit defines this relationship for triggers.

R_{MLZZW} A *Trigger Event* might be generated by a trace operation as follows:

- If the *Trigger Event* is generated when a *Detected Trigger* occurs when the *Trigger Counter* is zero and TRBSR_ELx.TRG is 0, and the *Detected Trigger* is generated by a trace operation t_T then the *Trigger Event* is generated by the same trace operation t_T.
- If the *Trigger Event* is generated when the Trace Buffer Unit decrements the *Trigger Counter* to zero, then *Trigger Event* is generated by the trace operation that generated the trace data that caused the *Trigger Counter* to decrement to zero.

A *Trigger Event* is not generated by a specific trace operation if the *Trigger Event* is generated when a *Detected Trigger* occurs when the *Trigger Counter* is zero and TRBSR_ELx.TRG is 0, and the *Detected Trigger* is not generated by a specific trace operation.

ELx is the value in the *Other event* column of Table D6-7.

I_{GPHHS} The link between a *Trigger Event* and a trace operation that generated it affects when the trace operation is Microarchitecturally-finished and the behavior of the `TSB CSYNC` instruction. See R_{JQDJD}.

I_{NGLLQ} FEAT_TRBE supports the following trigger modes:

Stop on trigger
Trace collection is stopped and an interrupt request is generated after a *Trigger Event*.

IRQ on trigger
An interrupt request is generated after a *Trigger Event*.

Ignore trigger
The Trace Buffer Unit ignores the trigger condition.

In the *Stop on trigger* and *IRQ on trigger* modes, software specifies the amount of trace that is collected after the trigger condition before the *Trigger Event*.

R_{XRRWP} If a *Trigger Event* is generated when collection is not stopped and the trigger mode is *Stop on trigger*, then all of the following occur:

- The Trace Buffer Unit initiates a *trace unit flush* of the trace unit.
- The TRB_TRIG event is generated.

On completion of the trace unit flush, all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1.
- If TRBSR_ELx.S is 0, then all of the following occur:
  - TRBSR_ELx.S is set to 1, collection is stopped.
  - TRBSR_ELx.EC is set to `0x00`, other buffer management event.
  - TRBSR_ELx.BSC is set to `0b000010`, Trigger Event.
- The other fields in TRBSR_ELx are unchanged.

ELx is defined by R_{BNQRP}. See also Table D6-7.

After the trace buffer management event, the *current write pointer* will point to either the first byte after the last trace byte written to the trace buffer, or, if the last trace byte written to the trace buffer was the last byte in the trace buffer, the *Base pointer*.

I_{MCGFL} The trace unit defines the behavior and completion of a trace unit flush, including which trace operations, if any, are accepted by the Trace Buffer Unit before the trace unit flush completes.

If the *Detected Trigger* is generated by a trace operation t_T, then the trace unit flush does not complete before the Trace Buffer Unit *accepts* the trace data for t_T.

I_{VPLRF} Because the Trace Buffer Unit initiates a trace unit flush before stopping this means that, before TRBSR_ELx.S is set to 1:

- More trace might be written to the trace buffer after the *Trigger Event* is detected.
- This might generate another management event that sets TRBSR_ELx.S to 1. This means that the Trigger Event is not recorded in TRBSR_ELx.
- This might generate another management event that sets TRBSR_ELy.S to 1, where ELy is a different Exception level to ELx. This does not prevent the Trigger Event from being recorded in TRBSR_ELx.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{XYPYF} If a *Trigger Event* is generated when collection is not stopped and the trigger mode is *IRQ on trigger*, then all of the following occur:

- The Trace Buffer Unit initiates a *trace unit flush* of the trace unit.
- The TRB_TRIG event is generated.

On completion of the trace unit flush, all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1, where ELx is defined by R_{BNQRP}. See also Table D6-7.
- The other fields in TRBSR_ELx are unchanged.

Because TRBSR_ELx.S is unchanged, trace continues to be collected and written to the trace buffer.

R_{LMQHK} If a *Trigger Event* is generated and the trigger mode is *Ignore trigger*, then all of the following occur:

- TRBSR_ELx is unchanged.
- The TRB_TRIG event is generated.

R_{MWWHM} If a *Trigger Event* is generated when collection is stopped, then all of the following occur:

- TRBSR_ELx is unchanged.
- The TRB_TRIG event is generated.

I_{NZYNN} These rules mean that trace might be collected after the *Trigger Event*, but are included to ensure that trace for the instructions that caused the trigger condition is not discarded in common cases.

I_{JDWMT} The trigger mode is controlled by TRBLIMITR_EL1.TM.

I_{SJWBD} See also Controlling generation of trace buffer management events.

D6.5.4 Faults

R_{QKLXR} A write by the Trace Buffer Unit might generate one or more of the following faults:

Alignment fault

If TRBPTR_EL1 is misaligned, the behavior is UNPREDICTABLE and a write to the trace buffer by the Trace Buffer Unit might generate an Alignment fault. See also R_{MGZWR}.

Translation fault

Any access outside the virtual address or intermediate physical address space generates a Translation fault.

The translation of a virtual address or intermediate physical address to a physical address might generate Translation fault.

Writes to the trace buffer are made as privileged writes in the owning translation regime, meaning they are not affected by the TCR_ELx.E0PDy bits for the owning translation regime.

Address Size fault

The translation of a virtual address or intermediate physical address to a physical address, or use of an out-of-range physical address, might generate an Address Size fault.

Permission fault

Writes to the trace buffer are made as privileged writes in the owning translation regime. If the write does not have write permission, a Permission fault is generated. The value of PSTATE.PAN and the FEAT_S1POE stage 1 Overlay permissions are ignored.

If the Trace Buffer Unit does not manage the dirty state in translation tables, then accesses ignore the Dirty Bit Modifier bit in Page and Block descriptors and as a result, might generate a Permission fault.

Access Flag fault

If the Trace Buffer Unit does not manage the Access flag in translation tables or hardware management of the Access flag state is disabled for the owning translation regime, then any access to a Page or Block with the Access flag bit set to 0 in a descriptor will generate an Access Flag fault.

TLB Conflict fault

IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Unsupported atomic hardware update fault

If hardware update of the translation tables is not guaranteed atomic in regard to other agents that access the memory, the translation of a virtual address to a physical address might generate an Unsupported atomic hardware update fault.

External abort on translation table walk, translation table update, or Granule Protection Table (GPT) walk

The translation of a virtual address to a physical address might generate an External abort on the translation table walk, translation table update, or GPT walk, and be treated as a synchronous MMU fault. See also External aborts.

Granule Protection Check fault (GPC fault)

Writes to the trace buffer are subject to granule protection checks and might generate GPC faults.

MMU fault means any of these faults other than an Alignment fault.

R_{YQMQK} MMU faults other than GPC fault and, optionally, Address Size faults are not generated in External mode.

R_{FYBCG} Writes to the trace buffer by the Trace Buffer Unit never generate watchpoints.

I_{DTKGB} Faults do not generate an actual Data Abort exception. The ESR and FAR registers are unchanged.

S_{GTLCY} To avoid MMU faults, Arm recommends:

- Software pins the Pages or Blocks used for the trace buffer. This includes a hypervisor pinning these Pages or Blocks in the stage 2 translation.
- If the Trace Buffer Unit does not manage the Access Flag and dirty state, software marks the Pages or Blocks as accessed and dirty. Software can discover whether address translations performed by the Trace Buffer Unit manage dirty state and the Access flag from TRBIDR_EL1.F.
- The IPA corresponding to trace buffer pointer does not have the AssuredOnly attribute when TRBLIMITR_EL1.nVM is 1. Otherwise, any Trace Buffer Unit writes would generate a stage 2 permission fault.

R_{FSPBK} If a write by the Trace Buffer Unit generates an Alignment fault or MMU fault, including a Granule Protection Check (GPC) fault, and TRBSR_ELx.S is 0, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1.
- TRBSR_ELx.S is set to 1, collection is stopped.
- TRBSR_ELx.EC is set to the appropriate one of the following values:
  - `0x1E`, GPC fault on write to trace buffer, other than a Granule Protection Fault (GPF).
  - `0x24`, stage 1 Data Abort on write to trace buffer.
  - `0x25`, stage 2 Data Abort on write to trace buffer.

  A GPF on translation table walk or update is reported as either a stage 1 or stage 2 Data Abort, as appropriate. Other GPFs are reported as a stage 1 Data Abort.
- TRBSR_ELx.FSC is set to indicate the type of the fault.
- TRBPTR_EL1 is set to the address that generated the fault.
- The other fields in TRBSR_ELx are unchanged.

If the MMU fault is an External abort on a translation table walk, translation table update, or GPT walk, then it is IMPLEMENTATION DEFINED whether TRBSR_ELx.EA is set to 1 or unchanged.

The value of ELx depends on the type of event, and is defined by R_{BNQRP}. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

R_{LYHGV} For a stage 2 MMU fault on Trace Buffer Unit writes, TRBSR_ELx.TopLevel is set as follows:

- 1, for a stage 2 Permission fault due to the TopLevel attribute.
- 0, for any other stage 2 fault.

ELx is defined by R_{BNQRP}. See also Table D6-7.

R_{YCPZX} For a stage 2 MMU fault on Trace Buffer Unit writes, TRBSR_ELx.AssuredOnly is set as follows:

- 1, for a stage 2 Permission fault due to the AssuredOnly attribute.
- 0, for any other stage 2 fault.

ELx is defined by R_{BNQRP}. See also Table D6-7.

I_{ZNPQG} In the case of a stage 2 Data Abort on a write to trace buffer, the PE does not record whether the fault was due to a stage 2 fault on the access, or a stage 2 fault on a stage 1 translation table access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.5.5 External aborts

R_{GQWBF} When a write to the trace buffer generates an External abort, including an External abort on a translation table walk, translation table update, or GPT walk, then one of the following occurs, and it is IMPLEMENTATION DEFINED which:

- The External abort is ignored.
- The External abort generates an SError exception.
- For an External abort on a translation table walk, translation table update, or GPT walk only, the External abort generates a synchronous MMU fault.
- If Armv9.3 is not implemented, the External abort is reported to the Trace Buffer Unit and the Trace Buffer Unit generates a trace buffer management event.

D6.5.5.1 The External abort is ignored

I_{FBNFC} If the External abort is ignored, this has the same visible behavior as when a write does not generate any External abort. TRBSR_ELx is not modified.

D6.5.5.2 The External abort generates an SError exception

R_{VKWPM} If a write to the trace buffer generates an External abort that is taken as an SError exception, the PE takes the SError exception as normal. TRBSR_ELx is not modified.

D6.5.5.3 The External abort generates a synchronous MMU fault

I_{LNXGR} When an External abort on a translation table walk, translation table update, or GPT walk generates a synchronous MMU fault, the behavior is as described by Faults.

D6.5.5.4 The External abort is reported to the Trace Buffer Unit

I_{NHQGS} The behavior in this section is not permitted if Armv9.3 is implemented.

R_{NZTKV} When an External abort is reported to the Trace Buffer Unit, then one of the following occurs, and it is IMPLEMENTATION DEFINED which:

- The External abort is handled synchronously by the Trace Buffer Unit.
- The External abort is handled asynchronously by the Trace Buffer Unit.

R_{XRLSC} When an External abort is reported to the Trace Buffer Unit:

- The External abort bit, TRBSR_ELx.EA, is set to 1.
- The Trace Buffer Unit stops writing trace data to the trace buffer.
- If the Trace Buffer Unit handles External aborts synchronously, TRBPTR_EL1 is set to the address that generated the External abort.
  If the Trace Buffer Unit handles External aborts asynchronously, TRBPTR_EL1 is not guaranteed to be set to the address that generated the External abort.
- If TRBSR_ELx.S is 0, then all of the following occur:
  - A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1.
  - TRBSR_ELx.S is set to 1, collection is stopped.
  - TRBSR_ELx.EC is set as follows:
    - If the External abort is on a GPT fetch, `0b011110`, Granule Protection Check fault on write to buffer, other than Granule Protection Fault (GPF).
    - If the External abort is on a stage 2 translation table access, `0b100101`, stage 2 Data Abort on write to buffer.
    - Otherwise, `0b100100`, stage 1 Data Abort on write to buffer.
  - TRBSR_ELx.FSC is set to indicate the type of External abort.
- The other TRBSR_ELx fields are unchanged.

ELx is defined by R_{BNQRP}. See also Table D6-6. The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

I_{TVKJR} If the Trace Buffer Unit handles External aborts asynchronously, then all of the following apply:

- The External abort trace buffer management event might not be generated until after a first trace buffer management event has set TRBSR_ELx.S to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Writes to the trace buffer might generate a second trace buffer management event after the External abort trace buffer management event has set TRBSR_ELx.S to 1.
- The Trace Buffer Unit might collect further trace data from the trace unit and write it to memory before the External abort trace buffer management event sets TRBSR_ELx.S to 1.

D6.5.6 IMPLEMENTATION DEFINED management events

R_{CHNSL} When IMPLEMENTATION DEFINED conditions are met all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_ELx.IRQ to 1.
- TRBSR_ELx.S is set to 1, collection is stopped.
- TRBSR_ELx.EC is set to `0x1F`, IMPLEMENTATION DEFINED buffer management event.
- TRBSR_ELx.MSS is set to an IMPLEMENTATION DEFINED value.
- The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

ELx is defined by R_{BNQRP}. See also Table D6-7.

I_{LKLHH} The intent of this event is for cases such as errata workarounds to allow an implementation to report any failure to write data to the buffer that is not covered by other codes. Arm recommends that such mechanisms are disabled on reset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.6 Synchronization and the Trace Buffer Unit

I_{NMWZY} Program Flow Trace data is generated by traced instructions. When an instruction is executed:

1. The PE decides whether to create a trace operation for the instruction.
2. If created, the trace operation generates the Program Flow Trace data.

For some trace unit implementations, Speculative instructions might generate trace operations, as well as architecturally Resolved instructions.

See Stages of Execution for more information on these terms.

The trace unit might also generate asynchronous trace operations, that are not causally related to an executed instruction. If the trace unit implements the FEAT_ETE then the ETE Resources can generate trace operations that are not causally related to an instruction or Speculative instruction.

The architecture defines a Trace Synchronization event that synchronizes the operation of trace operations with the execution of instructions. Without correct use of the Trace Synchronization event, a trace operation might for instance read a stale value from a System register, causing trace data to be written to the wrong memory location, or the Trace Buffer Unit to otherwise generate UNPREDICTABLE software behavior. See also UNPREDICTABLE behavior.

R_{ZVDST} Trace operations operate independently of the instructions that are executed on the PE and make indirect reads and indirect writes of System registers as an *external agent*.

I_{TPVQR} The synchronization requirements for direct reads, direct writes, indirect reads, and indirect writes of System registers made by instructions and external agents are defined in Synchronization requirements for AArch64 System registers.

R_{NRLDV} Synchronization in self-hosted trace defines that indirect reads of the TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} trace filter controls when determining whether the current Execution stream is part of a prohibited trace region and an instruction A should generate a trace operation, are treated as indirect reads made by A.

R_{SXXQJ} Each System register access made by the trace unit is one of the following, and the trace unit defines which:

- An indirect read or indirect write rw_A made by an instruction A. For example, to determine whether to generate a trace operation for A.
- An indirect read or indirect write rw_{tA} made by a trace operation t_A. This is in addition to System register accesses defined by this manual as indirect reads or indirect writes made by the Trace Buffer Unit.
- An other indirect read or indirect write rw, not directly related to either an instruction or trace operation. The trace unit defines the synchronization requirements for these registers.

I_{DDLVC} In addition to the registers listed by R_{NRLDV}, for the ETE, the following ETE System registers are indirectly read by an instruction A to determine whether A should generate a trace operation:

- TRCPRGCTLR.EN, the Trace unit enable bit in the Programming Control Register.
- TRCOSLSR.OSLK, the Trace OS Lock Status Register.

This manual defines which System registers are indirectly read or indirectly written by the Trace Buffer Unit as part of the trace operation t_A for a traced instruction A. For example:

- Trace Buffer Unit System registers.
- VMSA System registers and SCR_EL3.NS, when translating addresses generated by the Trace Buffer Unit.

Other System registers are indirectly read or indirectly written by the trace unit as part of the trace operation t_A for a traced instruction A. For example:

- Other ETE System registers.
- If context tracing is enabled, the applicable Context ID register or registers, CONTEXTIDR_EL1 or CONTEXTIDR_EL2.
- If trace timestamping is enabled, any applicable counter offset, CNTVOFF_EL2 or CNTPOFF_EL2.

Some ETE System registers are indirectly read or indirectly written by, for example, the ETE Resources when generating trace operations or updating the ETE Resources, and are made by neither a trace operation of an instruction nor an instruction.

The behavior of the ETE is defined by FEAT_ETE. See also Trace synchronization and the Trace Unit.

I_{GVCVL} The indirect reads and indirect writes to Trace Buffer Unit, VMSA and other System registers made by the Trace Buffer Unit are made by the trace operation t_{OP}.

I_{BYSGH} To synchronize trace operations, software must use the `TSB CSYNC` instruction to generate a Trace Synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{GTCKK} In the absence of any explicit synchronization, the trace unit generates the trace data for an instruction and the Trace Buffer Unit *accepts*, *discards*, or *rejects* the trace data in finite time. However:

- If the Trace Buffer Unit *accepts* the trace data, then the write of the trace data to memory requires explicit synchronization to Complete.
- The indirect writes to System registers made by a trace operation require explicit synchronization to guarantee they are observable.

R_{NNLMC} If FEAT_TRBE_EXT is implemented and TRBSR_EL1.DAT is 0, then all Trace operations Accepted by the Trace Buffer Unit will Complete in finite time.

D6.6.1 Trace Synchronization event

R_{VWJNN} Executing a `TSB CSYNC` instruction generates a Trace Synchronization event.

R_{JQDJD} A trace operation t_{OP} is not Microarchitecturally-finished before all of the following are true:

- All indirect reads and indirect writes of System registers made by t_{OP} have been performed.
- If t_{OP} generates a *Trigger Event* that in turn initiates a *trace unit flush*, then all of the following are true:
  - The trace unit flush is complete.
  - All trace operations the Trace Buffer Unit *accepts* before the trace unit flush completes are Microarchitecturally-finished.
  - All indirect writes to System registers made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

Indirect reads and writes include but are not limited to the following:

- All indirect reads and indirect writes of the Trace Buffer Unit, VMSA System registers, and SCR_EL3.{NSE, NS} made by memory accesses performed by t_{OP}.
- All indirect writes to System registers made by a trace buffer management event generated by t_{OP}.

However, this does explicitly exclude any indirect writes of System registers made in response to an External abort by the access.

R_{NSFRQ} A trace operation t_{OP} is complete when it is Microarchitecturally-finished and all memory accesses performed by t_{OP} are Complete and any indirect writes of System registers made in response to an External abort response to the access have been performed.

R_{MRVPT} If, following a Context synchronization event CSE the PE is executing in a Trace Prohibited region, a `TSB CSYNC` executed in the Trace Prohibited region and Non-debug state in program order after CSE is not Microarchitecturally-finished before all of the following are true:

- All trace operations t_A generated by instructions A in program order before CSE are Microarchitecturally-finished.
- All trace operations t_S generated by Speculative instructions S that are not in speculative execution order after CSE are Microarchitecturally-finished.
- All trace operations t_R generated by the trace unit are Microarchitecturally-finished.
- The trace unit enters a state where the trace unit does not generate further trace operations and does not signal a *Detected Trigger*. The trace unit remains in this state while the PE is executing in the Trace Prohibited region.
- If a *trace unit flush* is initiated by a *Trigger Event* before the `TSB CSYNC` is Microarchitecturally-finished, the trace unit flush is complete, all trace operations the Trace Buffer Unit *accepts* before the trace unit flush completes are Microarchitecturally-finished, and any indirect writes made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

These trace operations are synchronized by the `TSB CSYNC`.

R_{ZCDDS} A direct write W_2 to a System register made by an instruction B is Coherence-after an indirect read or indirect write rw_1 of the same System register made by a trace operation t_A for a traced instruction A if all of the following are true:

- Either A is executed in program order before a Context synchronization event CSE, or A is CSE.
- CSE is in program order before a Trace synchronization barrier TSB.
- B is executed in program order after TSB.
- After executing CSE, the PE is in a Trace Prohibited region and TSB is executed in Non-debug state in the same Trace Prohibited region.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{FZHQG} R_{ZCDDS} emerges from the requirement in R_{MRVPT} for the trace operations to be Microarchitecturally-finished by the `TSB CSYNC` operation.

Self-hosted trace extension synchronization rules and Trace in Debug state define further rules for the operation of `TSB CSYNC`.

I_{ZKRZH} The PE does not stall indefinitely (or until interrupted) waiting for a `TSB CSYNC`. For example, the `TSB CSYNC` must not wait until there is no trace data left to write if the trace unit is capable of producing a constant stream of trace data.

I_{ZLDPS} A PE might abandon a `TSB CSYNC` executed in Non-debug state before it is Microarchitecturally-finished to take an interrupt, so long as the preferred return address is set such that the `TSB CSYNC` is re-executed when the interrupt handler completes. That is, the `TSB CSYNC` is only Speculatively executed.

R_{CKVWP} Absent any Context synchronization event or `DSB` Data synchronization barrier, a `TSB CSYNC` instruction is not required to execute in program order with respect to other instructions or memory accesses. This means that software must execute additional barriers to guarantee that the trace operations are Microarchitecturally-finished and/or complete.

D6.6.2 Trace synchronization and the Trace Unit

I_{ZYZGZ} The ETE has *Resources* that can generate trace operations that are not directly generated by an instruction or Speculative instruction.

The ETE specification defines the following rules:

- The Resources do not generate trace operations in the *Paused* state.
- If, following a Context synchronization event, the PE is executing in a Trace Prohibited region and the ETE is enabled, the ETE pauses the ETE Resources.
- How software synchronizes indirect writes to System registers made by trace operations generated by Resources.

The behavior of the ETE is defined by FEAT_ETE.

I_{GFJWK} If FEAT_RME is implemented, further rules are defined for the behavior of the trace unit when a `TSB CSYNC` instruction is executed.

R_{QVGHP} A trace operation t_R generated by ETE Resources inherits the synchronization requirements for a trace operation generated by an instruction A, even if no trace operation is generated by A, provided that one of the following applies:

- The requirement is that A is executed in program-order after CSE and tracing was prohibited before CSE and is allowed after CSE.
- The requirement is that either A is executed in program-order before CSE or A is CSE, and tracing was allowed before CSE and is prohibited after CSE.

R_{YWSDB} If the trace unit becomes enabled when the PE is executing a Trace Prohibited region, it does not generate any trace operations, including trace operations for Speculative instructions and other trace operations not generated by instructions, until the PE enters a region where tracing is allowed.

I_{XGQRM} The trace unit defines the sequence by which software enables the trace unit.

D6.6.3 Self-hosted trace extension synchronization rules

I_{CMGRF} FEAT_TRF defines further rules for the `TSB CSYNC` instruction. In particular, how a `TSB CSYNC` instruction synchronizes direct reads and indirect writes to a System register with respect to indirect reads and indirect writes of the same System register made by trace operations.

These rules restate the synchronization behavior described in Memory barriers and Synchronization in self-hosted trace.

R_{TSPXF} is similar to R_{ZCDDS} and R_{XQVZW}. However:

- R_{TSPXF} applies whether the `TSB CSYNC` operation is executed in a trace prohibited or trace allowed region, in both Non-debug state and Debug state. R_{ZCDDS} applies only when the `TSB CSYNC` is executed in a Trace Prohibited region and the PE is in Non-debug state, and R_{XQVZW} applies only when the `TSB CSYNC` is executed when the trace unit is disabled and the PE is in Debug state.
- R_{TSPXF} applies only to System registers accessed by the trace unit as part of a trace operation. R_{ZCDDS} and R_{XQVZW} apply to all System register accesses made by the trace operation and includes indirect reads and indirect writes made by the Trace Buffer Unit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See R_{SXXQJ} and I_{DDLVC}.

R_{BFJKD} An indirect read r_1 of a System register made by a trace operation t_A for a traced instruction A Reads-from a direct write W_2 to the same System register made by an instruction B if all of the following are true:

- A is executed in program order after a Context synchronization event CSE.
- B is executed in program order before CSE.

See also I_{FVBPF}.

S_{YPWVJ} R_{BFJKD} means that, if the PE enters a region where tracing is allowed by executing a Context synchronization event, such as an `ERET` instruction when SCTLR_ELx.EOS is 1, then all indirect reads and writes of System registers made by trace operations generated after entering the tracing allowed region will observe the values in those System registers written by direct writes before the Context synchronization event.

R_{MJQXM} An indirect write w_1 of a System register made by a trace operation t_A for a traced instruction A is Coherence-after a direct write W_2 of the same System register made by an instruction B if all of the following are true:

- A is executed in program order after a Context synchronization event CSE.
- B is executed in program order before CSE.

See also I_{FVBPF}.

R_{TSPXF} A direct write W_2 to a System register made by an instruction B is Coherence-after an indirect read or indirect write rw_1 of the same System register made by the trace unit as part of a trace operation t_A for a traced instruction A if all of the following are true:

- Either A is executed in program order before a Context synchronization event CSE, or A is CSE.
- CSE is in program order before a Trace synchronization barrier TSB.
- B is executed in program order after TSB.

See also I_{MQTRQ}.

R_{VTNCS} A direct read R_2 of a System register made by an instruction B Reads-from an indirect write w_1 to the same System register made by a trace operation t_A for a traced instruction A if all of the following are true:

- Either A is executed in program order before a first Context synchronization event CSE_1, or A is CSE_1.
- CSE_1 is in program order before a Trace synchronization barrier TSB.
- TSB is executed in program order before a second Context synchronization event CSE_2.
- B is executed in program order after CSE_2.

See also I_{SLWRW}.

R_{NNRHD} An instruction A in program-order after a direct write W that modifies one of the trace filter controls, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE}, Reads-from W when determining whether A should generate a trace operation, if there is no intervening direct write to the same register and any of the following is true:

- A is in program-order after a Context synchronization event CSE and CSE is in program-order after W.
- An instruction B Reads-from W when determining whether B should generate a trace operation, and A is in program-order after B.

I_{VKRGM} R_{NNRHD} means that for the instructions between a direct write to one of the trace filter controls to either enable or disable trace at the current Exception level and a following Context synchronization event, although it is UNPREDICTABLE whether each instruction observes the old or new values of the trace filter controls, once one instruction has observed the new value, all subsequent instructions also observe the new value.

However this might not happen, and all instructions might observe the old values until the Context synchronization event occurs.

This guarantees that trace switches either on and off cleanly, and is required by Program Flow Trace protocols.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.7 Trace synchronization and memory barriers

R_{VLWDM} If, following a Context synchronization event CSE, the PE is executing in a Trace Prohibited region, a `DSB` with required access types of reads and write is executed in program order after a `TSB CSYNC` operation that is executed in the Trace Prohibited region in program order after CSE, then in addition to the requirements in this manual, the `DSB` does not complete until all explicit memory accesses of the required access type made by the trace operations synchronized by the `TSB CSYNC` are Complete for the set of observers in the required Shareability domain.

See also R_{MPXXN}.

R_{DFHWV} An explicit read, explicit write, translation table walk, cache maintenance operation, or TLB invalidate operation M_1 will be Observed-by a read or a write RW_2 of a Location made by a trace operation t_A relating to a traced instruction A if all of the following are true:

- A is executed in program-order after a Context synchronization event CSE.
- CSE is in program order after a Data Synchronization Barrier DSB.
- DSB does not complete before M_1 is complete.

I_{NLHGN} In R_{DFHWV}, CSE is required to allow A to be traced as a Speculative instruction before it is Canceled or Resolved. The equivalent rule for SPE does not require CSE as SPE cannot write profiling records to memory until the profiled operation is Canceled or Resolved. See Trace of Speculative execution.

R_{JPPZZ} For the indirect writes to TRBPTR_EL1 and TRBSR_ELx that are made as a result of an External abort on a write of trace data to memory, the synchronization rules apply only after the write to memory has completed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.8 Trace of Speculative execution

I_{ZGCVG} In the standard model of execution for an instruction, instructions are executed as Speculative operations and then later become one of the following:

- Resolved. These instructions will then proceed to Complete.
- Canceled.

A trace unit might generate trace for Speculative instructions before they are Resolved or Canceled, and this trace can be written to memory. This means that, as well as indirectly reading System registers or memory, a trace operation t_S for a Speculative instruction S might perform any of the following:

- Indirectly write to System registers.
- Write to memory.

This sets trace operations apart from the normal operation of instructions, as the Arm architecture prohibits, for example, a Canceled instruction from updating a System register or memory. (Performance Monitors and Statistical Profiling can also cause speculative updates of System registers and memory.)

I_{HBYGS} The preceding rules deal with the ordering of trace operations for Resolved (and ultimately Complete) instructions, and the requirements for synchronization based on the position of those instructions in the program order of the Execution stream.

This section extends the rules to cover Speculative instructions that are later Canceled, and for trace operations generated by ETE Resources. Because these trace operations are not generated by instructions that Complete, they cannot be determined to be in *program order* with respect to architecturally Complete instructions.

R_{FBFRS} A Speculative instruction S is in speculative execution-order after an instruction A if S will be in program order after A if S is Resolved, even if S is subsequently Canceled.

I_{VFLVX} For example, a Speculative instruction S is in speculative execution-order after a Resolved instruction A if either of the following are true:

- The branch predictor mispredicted A and, had the prediction been correct, S would be in program order after A. *Branch predictor* means any structure that causes the PE to execute Speculative instructions. This includes, for example, branch history buffers, branch target caches, and instruction trace caches. It is not limited to structures that predict only the direction and/or target of branch instructions.

I_{KTCMP} Speculative execution-order does not provide a complete ordering. A pair of Speculative instructions A and B might not be ordered with respect to each other. For example, if A and B are respectively the result of different incorrect predictions by the branch predictor. However, each Speculative instruction is in speculative execution-order after at least one Resolved instruction.

I_{VBBVY} The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively.

R_{HCVVS} A trace operation t_S generated by a Speculative instruction S that is in speculative execution-order after a Resolved instruction A inherits the synchronization requirements for a trace operation generated by A, even if no trace operation is generated by A, provided that one of the following applies:

- The requirement is that A is executed in program-order after CSE, and S is in speculative execution-order after CSE.
- The requirement is that either A is executed in program-order before CSE or A is CSE, and S is not in speculative execution-order after CSE.

See also I_{WLLQH} and I_{JGXWM}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.9 Trace in Debug state

R_{WYHRT} If the trace unit is disabled, then a `TSB CSYNC` executed in Debug state is not Microarchitecturally-finished before all of the following are true:

- All trace operations t_A generated by instructions A in program order before the PE entered Debug state are Microarchitecturally-finished.
- All trace operations t_S generated by Speculative instructions S that are not in speculative execution order after the entry to Debug state are Microarchitecturally-finished.
- All trace operations t_R generated by the trace unit are Microarchitecturally-finished.
- The trace unit enters a state where the trace unit does not generate further trace operations and does not signal a *Detected Trigger*. The trace unit remains in this state while the PE is in Debug state.
- If a *trace unit flush* is initiated by a *Trigger Event* before the `TSB CSYNC` is Microarchitecturally-finished, the trace unit flush is complete, all trace operations the Trace Buffer Unit *accepts* before the trace unit flush completes are Microarchitecturally-finished, and any indirect writes made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

These trace operations are synchronized by the `TSB CSYNC`.

R_{XQVZW} A direct write W_2 to a System register made by an instruction B is Coherence-after an indirect read or indirect write RW_1 of the same System register made by a trace operation t_A for a traced instruction A if all of the following are true:

- A is executed in program order before the PE entered Debug state.
- B is executed in program order after a Trace synchronization barrier TSB.
- TSB was executed in Debug state when the trace unit was disabled.

I_{VHGVC} R_{XQVZW} emerges from the requirement in R_{WYHRT} for the trace operations to be Microarchitecturally-finished by the `TSB CSYNC` operation.

Trace Synchronization event and Self-hosted trace extension synchronization rules define further rules for the operation of `TSB CSYNC`.

R_{MPXXN} A `DSB` with required access types of reads and write is executed after a `TSB CSYNC` operation executed in Debug state, then in addition to the requirements in this manual, the `DSB` does not complete until all explicit memory accesses of the required access type made by the trace operations synchronized by the `TSB CSYNC` are Complete for the set of observers in the required Shareability domain.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.10 Synchronization litmus tests

IDJJBQ This section details example synchronization scenarios and litmus tests for `TSB CSYNC`. These are derived from FEAT_TRF and Trace Synchronization event.

This section uses the terms program order, Reads-from and Coherence-after to define the ordering of System register and memory accesses made by trace operations. These terms are defined for memory accesses in Ordering requirements defined by the formal concurrency model. For the purposes of this section, these terms are used for System registers as well as memory accesses.

The terms external agent, Observer, and Observed-by, also used in this section, are also defined in Ordering requirements defined by the formal concurrency model.

This section does not describe the synchronization rules in Debug state defined in Trace in Debug state. In general, litmus tests for Debug state can be derived by applying the following modifications:

- Where a rule mentions a Context synchronization event (CSE) coming before a `TSB CSYNC` operation, if the `TSB CSYNC` is executed in Debug state, then the entry to Debug state can replace that CSE for the rule.
- Where a rule mentions the PE executing instructions in a Trace Prohibited region following the CSE, then executing the instructions in Debug state with the trace unit disabled is sufficient for the rule.

Exit from Debug state is a Context synchronization event.

IFVBPF Figure D6-3 shows R_{BFJKD} and R_{MJQXM}.

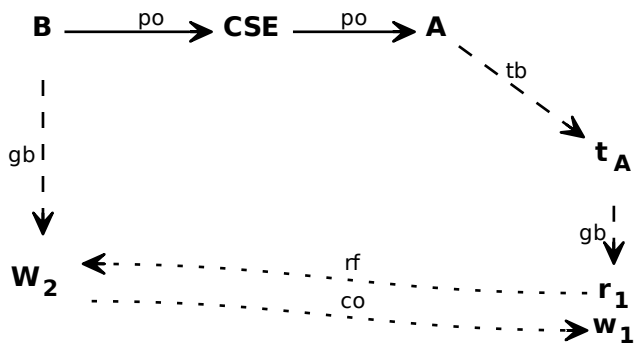

Figure D6-3 Indirect read or indirect write by Trace operation after direct write

IMQTRQ Figure D6-4 shows R_{TSPXF}.

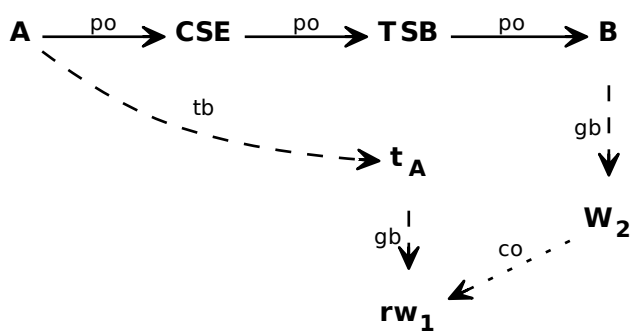

Figure D6-4 Direct write after indirect read or indirect write by Trace operation

ISLWRW In R_{VTNCS}, the second Context synchronization event CSE_2 is required to ensure the direct read B is not executed before the synchronization barrier TSB. Figure D6-5 shows this.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

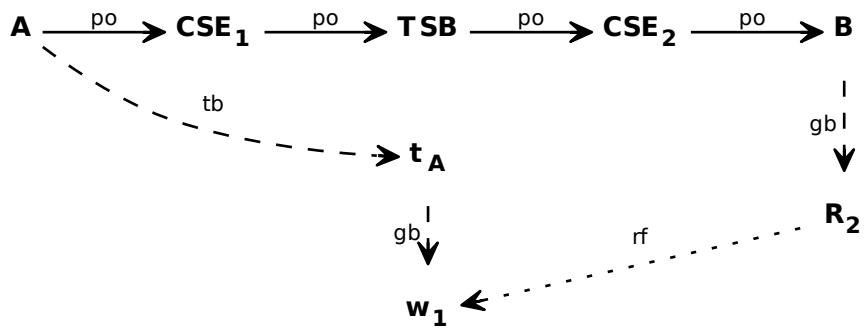

Figure D6-5 Direct read after indirect write by Trace operation

I_{LPPFQ} If the trace is not prohibited after the Context synchronization event, further trace operations might be generated that are not guaranteed to be synchronized by the `TSB CSYNC`. For I_{SLWRW}, this applies to the first Context synchronization event CSE_1.

Trace is prohibited at higher Exception levels than the owning Exception level. This means that if the PE takes an exception to a higher Exception level than the owning Exception level then trace is prohibited at by taking the exception.

I_{NWRPJ} Figure D6-6 shows R_{DFHWV} for an explicit read or a write M_1 of a Location made by an instruction B in program order before DSB.

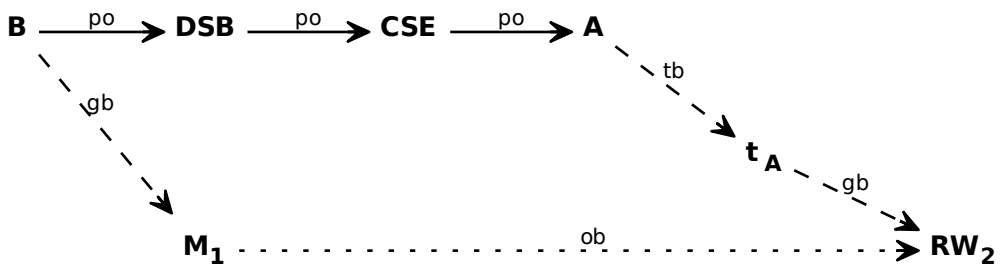

Figure D6-6 Trace operation observing memory operation

I_{JBRHG} If all of the following are true, R_{VLWDM} requires that a `DSB` with required access types of reads and writes does not complete until at least all reads or writes RW made by the Trace Buffer Unit for all trace operations t_A relating to a traced instruction A are complete for the set of the observers in the required Shareability domain:

- Either A is executed in program order before a Context synchronization event CSE, or A is CSE.
- The PE is executing in a Trace Prohibited region after CSE.
- CSE is in program order before a Trace synchronization barrier TSB.
- TSB is executed in program order before the `DSB`.

Figure D6-7 shows a read or a write RW_1 of a Location made by the Trace Buffer Unit for a trace operation t_A relating to a traced instruction A is complete and therefore will be Observed-by a read or a write RW_2 of the same Location made by an instruction B executed in program order after a `DSB ISH` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

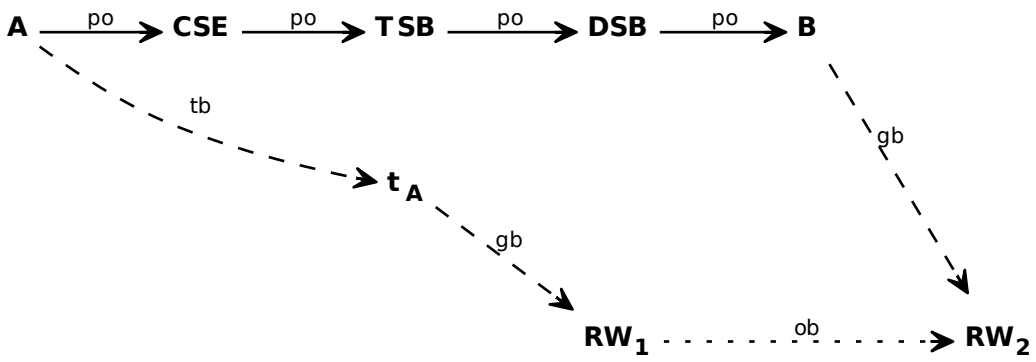

Figure D6-7 Observation of Trace operation memory access

IWLLQH R_{HCVVS} defines Trace synchronization for Speculative instructions.

For example, an indirect read r_1 of a System register made by a trace operation t_S for a traced Speculative instruction S Reads-from a direct write W_2 to the same System register made by an instruction B, if all of the following are true:

- S is executed in speculative execution-order after a Resolved instruction A.
- A is executed in program order after a Context synchronization event CSE.
- B is executed in program order before CSE.

Figure D6-8 shows this.

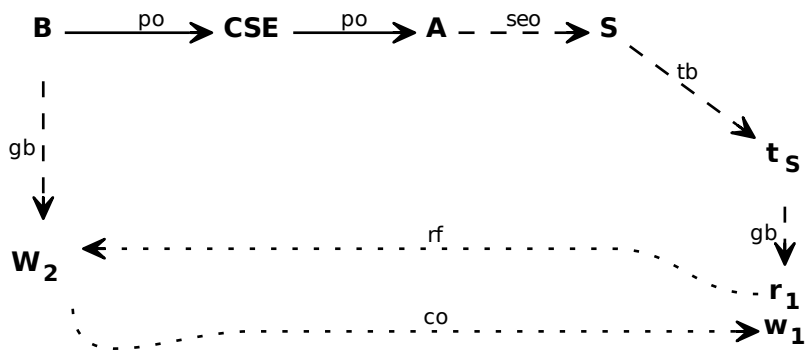

Figure D6-8 Speculative indirect read or indirect write by Trace operation after direct write

IJGXWM R_{HCVVS} defines Trace synchronization for Speculative instructions.

For example, if all of the following are true, a `DSB` with required access types of reads and writes does not complete until at least all reads or writes RW made by the Trace Buffer Unit for all trace operations t_S relating to a traced Speculative instructions S are complete for the required Shareability domain:

- S is executed in speculative execution-order after a Resolved instruction A.
- A is executed in program order before a Context synchronization event CSE.
- S is not in speculative execution-order after CSE.
- CSE is in program order before a Trace synchronization barrier TSB.
- The PE is executing in a Trace Prohibited region after CSE.
- TSB is executed in program order before the `DSB`.

Figure D6-9 shows a read or a write RW_1 of a Location made by the Trace Buffer Unit for a trace operation t_S relating to a traced Speculative instruction S is Complete and therefore will be Observed-by a read or a write RW_2 of the same Location made by an instruction B executed in program order after a `DSB ISH` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

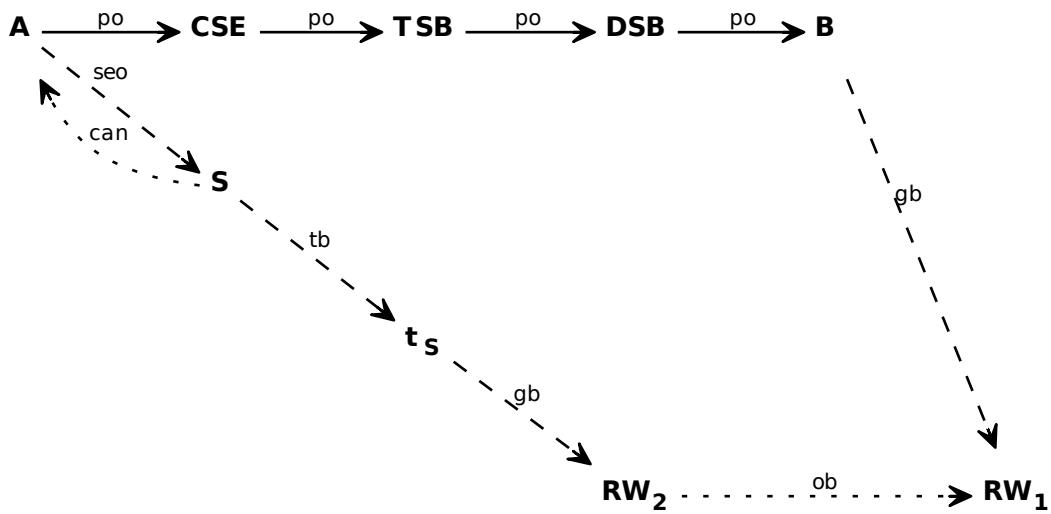

Figure D6-9 Observation of Speculative Trace operation memory access

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.11 UNPREDICTABLE behavior

R_{PFJJZ} In the absence of correct synchronization events, it is UNPREDICTABLE whether an indirect read made by a trace operation of a System register updated by a direct write will return the old or the new values. A trace operation might make multiple indirect reads of a System register, and each might return a different one of the old or the new values.

I_{GGGPH} For example, a trace operation might read MDCR_EL2.E2TB twice, as follows:

1. When the trace operation is first generated, to evaluate `TraceAllowed()` and determine whether trace is prohibited.
2. When the trace operation is complete and ready to be written to memory, to evaluate `TraceBufferOwner()` to determine the context for TRBPTR_EL1.

If MDCR_EL2.E2TB is updated without proper synchronization between these two events, both the old and new value might be used.

I_{NRQQF} In the absence of correct synchronization events, it is UNPREDICTABLE whether a direct read of a System register updated by an indirect write made by a trace operation will return the old or the new values.

I_{TRWFM} If an instruction is traced because the Trace Buffer Unit is enabled and *running*, but a later indirect read of a System register by the trace operation for the instruction determines that the trace data cannot be written to memory, because the Trace Buffer Unit now appears to be disabled, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The trace data is written to memory.
- The trace data is sent to an IMPLEMENTATION DEFINED trace bus.
- The trace data is written to memory and sent to an IMPLEMENTATION DEFINED trace bus.
- The Trace Buffer Unit *discards* the trace data.

This also includes any trace data that might be flushed by the trace unit, for example due to a `TSB CSYNC` operation.
If the Trace Buffer Unit *discards* the trace data, a trace buffer management event might be generated:

- TRBSR_ELx.EC is set to a CONSTRAINED UNPREDICTABLE choice of the following values:
  - `0x00`, other buffer management event. TRBSR_ELx.BSC is set to `0b000000`, access not allowed.
  - `0x1F`, IMPLEMENTATION DEFINED buffer management event.

It is also CONSTRAINED UNPREDICTABLE whether any of the following occur, whether or not the trace data is written to memory:

- The *current write pointer* and, if TRBSR_ELx.TRG is 1, the *Trigger Counter* are updated for the trace data.
- A trace buffer management event that would have been generated is observed and/or generated.
- A PMU event that would have been generated is generated.

R_{RRCNN} If the Effective value of SCR_EL3.{NSE, NS} does not match the owning Security state, it is CONSTRAINED UNPREDICTABLE whether any trace data:

- Is written to memory using a virtual address translated using the owning translation regime.
- Is written to memory using a virtual address translated using the translation regime formed from the owning Exception level and the Security state selected by the Effective value of SCR_EL3.{NSE, NS}.
- Is silently discarded and not written to memory.
- Is discarded and not written to memory and a trace buffer management event is generated as follows:
  - TRBSR_ELx.IRQ is set to 1, where ELx is defined by R_{BNQRP}.
  - If TRBSR_ELx.S is 0, then all of the following occur:
    - TRBSR_ELx.S is set to 1, collection is stopped.
    - TRBSR_ELx.EC is set to `0x00`, other buffer management event.
    - TRBSR_ELx.BSC is set to `0b000000`, access not allowed.
  - The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

I_{MJMWG} R_{RRCNN} means that if software executing at EL3 changes the value of SCR_EL3.NS or SCR_EL3.NSE before ensuring all trace operations are complete, this might cause CONSTRAINED UNPREDICTABLE behaviors. This means that software must execute a `TSB CSYNC` instruction to force any trace to be written to the Trace Buffer before changing context.

R_{RTGLJ} When FEAT_RME and FEAT_TRBE are implemented, if the Trace Buffer Unit is enabled and using Self-hosted mode, and MDCR_EL3.{NSTB, NSTBE} selects a reserved value, then the behavior is CONSTRAINED UNPREDICTABLE, and the Trace Buffer Unit does one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Behaves as if the Trace Buffer Unit is disabled.
- Selects an implemented Security state as the owning Security state.
- When trace data is received from the trace unit, it is not written to memory and the Trace Buffer Unit generates a trace buffer management event:
  - TRBSR_ELx.IRQ is set to 1, where ELx is defined by R_{BNQRP}.
  - If TRBSR_ELx.S is 0, then all of the following occur:
    - TRBSR_ELx.S is set to 1, collection is stopped.
    - TRBSR_ELx.EC is set to `0x00`, other buffer management event.
    - TRBSR_ELx.BSC is set to `0b000000`, *access not allowed*.
  - The other fields in TRBSR_ELx and the other TRBSR_ELx registers are unchanged.

I_{LLCLK} See also Context switching.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D7
The AArch64 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- About the memory system architecture.
- Address space.
- Mixed-endian support in AArch64.
- Memory Encryption Contexts.
- Cache support.
- External aborts.
- Memory barrier instructions.
- Pseudocode description of general memory System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see Memory hierarchy.

If FEAT_MTE2 is implemented, the definitions of the memory model which apply to data accesses and data apply to Allocation Tag accesses and Allocation tags, unless otherwise specified in The Memory Tagging Extension.

D7.1.1 Form of the memory system architecture

The A-profile architecture includes a *Virtual Memory System Architecture* (VMSA). The AArch64 Virtual Memory System Architecture describes the AArch64 view of the VMSA.

D7.1.2 Memory attributes

Memory types and attributes describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table D7-1 shows the memory attributes that are visible at the system level.

Table D7-1 Memory attribute summary

Memory type	Shareability	Cacheability
Device[a]	Outer Shareable	Non-cacheable.
Normal	One of: • Non-shareable. • Inner Shareable. • Outer Shareable.	One of[b]: • Non-cacheable. • Write-Through Cacheable. • Write-Back Cacheable.

a. Takes additional attributes, see Device memory.

b. See also Cacheability, cache allocation hints, and cache transient hints.

For more information on cacheability and shareability see Shareable Normal memory, Non-shareable Normal memory, and Caches and memory hierarchy.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.2 Address space

The architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *physical address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see Implemented physical address size.

D7.2.1 Virtual address space overflow

When a PE performs a Simple sequential execution of instructions, it calculates:

```
address_of_next_instruction = address_of_current_instruction + size_of_executed_instruction
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an instruction overflows `0xFFFF_FFFF_FFFF_FFFF`, the program counter becomes UNKNOWN. Similarly, if the address calculation based on the program counter for the value of the link register or exception link register overflows `0xFFFF_FFFF_FFFF_FFFF`, then those registers also become UNKNOWN.

Note

Address tags are not propagated to the program counter, so the tag does not affect the address calculation.

Where an instruction accesses a sequential set of bytes that crosses the `0xFFFF_FFFF_FFFF_FFFF` boundary when tagged addresses are not used, or the `0xXXFF_FFFF_FFFF_FFFF` boundary when tagged addresses are used, then the virtual address accessed for the bytes above this boundary is UNKNOWN.

The UNKNOWN virtual address behavior also applies to the set of bytes addressed by SVE and SME predicated, contiguous loads and stores that cross the `0xFFFF_FFFF_FFFF_FFFF` boundary, even if all of the virtual addresses below the boundary correspond to Inactive elements. Conversely, for SVE gather loads and scatter stores, the UNKNOWN address behavior applies only to accesses corresponding to an individual Active element that crosses the boundary.

When tagged addresses are used, the value of the tag associated with the address also becomes UNKNOWN.

Note

The behaviors described in this section apply only for the upper bound of the upper VA range, in translation regimes that have two VA ranges. They do not apply for address calculations relating to the top of the lower VA range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.3 Mixed-endian support in AArch64

A control bit, SCTLR_EL1.E0E is provided to allow the endianness of explicit data accesses made while executing at EL0 to be controlled independently of those made while executing at EL1. Table D7-2 shows the endianness of explicit data accesses and translation table walks.

Table D7-2 Endianness support

Exception level	Explicit data accesses	Stage 1 translation table walks	Stage 2 translation table walks
EL0	SCTLR_EL1.E0E[a]	SCTLR_EL1.EE	SCTLR_EL2.EE
EL1	SCTLR_EL1.EE	SCTLR_EL1.EE	SCTLR_EL2.EE
EL2	SCTLR_EL2.EE	SCTLR_EL2.EE	n/a
EL3	SCTLR_EL3.EE	SCTLR_EL3.EE	n/a

a. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the control is from SCTLR_EL2.E0E.

Note

SCTLR_EL1.E0E has no effect on the endianness of the Explicit Memory Effects generated by unprivileged memory access instructions, when these are executed at EL1.

AArch64 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR_ELx.EE is RES0, and SCTLR_EL1.E0E and SCTLR_EL2.E0E are RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR_ELx.EE is RES1, and SCTLR_EL1.E0E and SCTLR_EL2.E0E are RW.
- All Exception levels support only little-endianness:
  - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RES0.
- All Exception levels support only big-endianness:
  - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1. PSTATE.E is ignored in AArch64 state.

The `BigEndian()` function determines whether the current Exception level and Execution state are using big-endian data. This function is defined in A-profile Architecture Pseudocode.

For more information about endianness in the Arm architecture see Endian support.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.4 Memory Encryption Contexts

I_{QLZZR} The *Memory Encryption Contexts Extension* (MEC Extension) introduces *memory encryption contexts* (MECs) to all PA spaces. Multiple memory encryption contexts are provided to the Realm physical address space for assignment to Realm virtual machines, with policy controlled by Realm EL2. The Root, Secure and Non-Secure physical address spaces each have one context.

R_{LZNNK} All statements in this section and subsections require implementation of FEAT_MEC.

D7.4.1 Memory Encryption Context IDs

I_{NCBSV} A *Memory Encryption Context ID* (MECID) is used to associate a memory access with a MEC.

I_{TMSCH} The associated MEC is based on the current Security state, Exception level, and translation configuration.

R_{DWCZM} A MEC is uniquely identified by the tuple of PA space and MECID.

R_{PNKQV} All memory accesses, in addition to PA space and PA, are associated with a MECID.

I_{LGNTZ} Memory accesses associated with a MECID include reads, speculative reads, writes, atomics, instruction fetches and translation table lookups.

R_{ZSBJB} Each PA space has a default MECID, defined as MECID of zero.

I_{GHVCX} For PA spaces that support memory encryption, memory accesses associated with the default MECID of zero are encrypted.

R_{JQXXL} The Non-secure, Secure, and Root PA address spaces each support only one MEC.

R_{RWTKJ} Accesses to the Non-secure, Secure, and Root PA spaces are associated with a default MECID of zero.

R_{TPDCN} The Realm PA space supports multiple MECs.

I_{RMWYN} Accesses to the Realm PA space can be associated with multiple MECIDs.

R_{MWFGC} For accesses to Realm PA space, a MECID value of 0 is equivalent to the default MECID of zero.

I_{KZJFJ} Each MECID is bound to a cryptographic context for encrypted memory locations. A single MECID can be bound to different cryptographic contexts for different resources. For example, multiple memory banks can have dedicated encryption engines with independent keys.

I_{XJZYD} A MECID value is not a system-global identifier and it must be qualified by a PA space.

R_{TRNQY} If and how a device interprets the MECID associated with a register read or write is IMPLEMENTATION DEFINED.

I_{MXBDR} Devices are not anticipated to be MECID aware.

I_{WVTBQ} The MECIDs are configured in System registers for each supported execution context and translation regime.

I_{VVJGP} Software is permitted to manage MECID allocation independently from VMID and ASID allocation.

D7.4.1.1 MECID width

R_{WVFDT} The MECID width is 1 to 16 bits.

R_{RBXBC} For a PE, the supported MECID has an IMPLEMENTATION DEFINED bit width specified by MECIDR_EL2.MECIDWidthm1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{QDYKJ} System components may be integrated that support MECIDs of different widths. The *common MECID width* is the smallest supported MECID width for the entire system.

I_{BKQHQ} If the common MECID width is less than the value reported by MECIDR_EL2.MECIDWidthm1, then software should pad the upper bits with zeros.

I_{HGBXB} Because the MECID width has a finite size, MECIDs are expected to be used and re-used during the lifetime of a running system.

D7.4.1.2 MECID mismatch

I_{QBBQC} If the MECID associated with a memory access to a location is different than the MECID associated with a copy of the location in a cache, then a MECID mismatch occurs within that cache. A MECID mismatch can occur as the result of any memory access, including explicit read accesses, explicit write accesses, speculative read accesses, instruction fetches, and translation table lookups.

I_{YDGXJ} MECID mismatches are caused by software mis-configurations, such as:

- Multiple translation table mappings to a PA with different associated MECID values.
- Insufficient cache maintenance operations when re-assigning MECID values to a granule.
- Insufficient TLB maintenance and barriers when updating MECID registers, or MEC configuration.
- Using a common MECID width that is too large for the system.

I_{GWPRT} If MECIDR_EL2.MECIDWidthm1 is wider than the common MECID width, then all of the following can occur:

- The memory system exhibits the behaviors due to a MECID mismatch.
- Aliased sets of MECIDs result in non-uniquely-encrypted memory.

I_{BLZTF} Memory accesses with a MECID mismatch can cause either or both data corruption and data leakage across contexts.

I_{BVDCG} Recovery from a MECID mismatch requires software to correct the configuration state and perform a clean and invalidate to the PoE for all memory that might have been exposed to the MECID mismatch.

D7.4.2 Memory encryption block size

I_{ZQSGL} The block size that memory encryption engines use to encrypt data depends on the algorithm. A memory write to a location updates the entire encryption data block.

R_{LTWYZ} A memory encryption data block size is not permitted to be larger than the smallest supported translation granule size.

I_{RGMPP} Two or more memory writes with different MECID values to locations within the same smallest supported translation granule can corrupt the entire granule.

D7.4.3 Restrictions on the effects of speculation

R_{KLWFT} The MECID system registers are associated with the respective translation regime.

I_{PMLCM} The PE does not use configured MECID values from MECID system registers for speculative memory accesses for any translation regime that is out-of-context. For speculative memory accesses, the architecture requires all of the following:

- When executing at EL3 or EL2, the PE must not use the registers associated with the EL1&0 translation regime.
- When executing at EL3 the PE must not use the registers associated with the EL2 or EL2&0 translation regimes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.5 Cache support

This section describes the cache identification and control mechanisms, and the A64 cache maintenance instructions, in the following sections:

- General behavior of the caches.
- Cache identification.
- Cacheability, cache allocation hints, and cache transient hints.
- Enabling and disabling the caching of memory accesses.
- Behavior of caches at reset
- Non-cacheable accesses and instruction caches.
- About cache maintenance in AArch64 state.
- A64 Cache maintenance instructions
- Data cache zero instruction.
- Cache lockdown.
- System level caches.
- Branch prediction.
- Execution, data prediction and prefetching restriction System instructions.

See also Caches.

D7.5.1 General behavior of the caches

When a memory location has a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For more information, see The interaction of cache lockdown with cache maintenance instructions.

- Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.
- It is guaranteed that no memory location will be allocated into a Data or Unified cache if that location does not have a Normal Cacheable attribute in either:
  - The translation regime at the current Exception level.
  - The translation regime at any higher Exception level.
- For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all Requesters in its shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR_EL0 identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

D7.5.2 Cache identification

The cache identification registers describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes the cache maintenance instructions that:

- Affect the entire cache, for example IC IALLU.
- Operate by VA, for example IC IVAU.
- Operate by set/way, for example DC ISW.

The cache identification registers are:

- The Cache Type Register, CTR_EL0, that defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instruction.
  - The cache indexing and tagging policy of the Level 1 instruction cache.

  Note

  It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches see System level caches.

- A single Cache Level ID Register, CLIDR_EL1, that defines:
  - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.
  - The *Level of Coherence* (LoC) for the caches. See Terms used in describing the cache maintenance instructions for the definition of LoC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The *Level of Unification Uniprocessor* (LoUU) for the caches. See Terms used in describing the cache maintenance instructions for the definition of LoUU.
  - An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.
- A single Cache Size Selection Register, CSSELR_EL1, that selects the cache level and sort of cache (Instruction, Data/Unified/Tag) of the current Cache Size Identification Register.
- For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, CCSIDR_EL1, that defines:
  - The number of sets, associativity and line length of the cache. See Terms used in describing the cache maintenance instructions for a definition of these terms.

Note

From Armv8.3, multiple formats of the Cache Size Identification Register are supported. For more information, see Possible formats of the Cache Size Identification Register, CCSIDR_EL1.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.
3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - An instruction cache.
     - A data or unified cache.
   - Read the Cache Size Identification Register to find details of the cache.

D7.5.2.1 Possible formats of the Cache Size Identification Register, CCSIDR_EL1

From Armv8.3, the Cache Size Identification Register, CCSIDR_EL1 has two different formats available for defining the number of sets and associativity of the cache. For a definition of these terms, see Terms used in describing the cache maintenance instructions.

When FEAT_CCIDX is implemented:

- CCSIDR_EL1 is a 64-bit register.
- The length of the CCSIDR_EL1.Assoc field is 21 bits. This limits the associativity of the currently selected cache to 2^{21}.
- The length of the CCSIDR_EL1.NumSets field is 24 bits. This limits the number of sets in the currently selected cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.

When FEAT_CCIDX is not implemented:

- CCSIDR_EL1 is a 32-bit register.
- The length of the CCSIDR_EL1.Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10}.
- The length of the CCSIDR_EL1.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to 2^{15}.

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.5.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability applies only to Normal memory, and can be defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in Memory types and attributes, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a *cache transient hint* is supported, see Transient cacheability hint.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be assigned:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
- For VMSAv8-64 translation table walks, the TCR_ELx.{IRGN*n*, ORGN*n*} fields define the memory attributes of the translation tables, including the cacheability. However, this assignment supports only a subset of the cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

D7.5.3.1 Transient cacheability hint

In Armv8, it is IMPLEMENTATION DEFINED whether a Transient hint is supported. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Note

The architecture does not specify what is meant by a *relatively short period*.

The description of the AArch64 MAIR_EL1, MAIR_EL2, and MAIR_EL3 registers, and the AArch32 MAIR0, MAIR1, HMAIR0, and HMAIR1 registers, includes the assignment of the Transient hint in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

D7.5.4 Cacheable MEC transactions

R_{KLLJD} All statements in this section and subsections require implementation of FEAT_MEC.

I_{NWLXR} Each cache level is permitted to independently implement the behaviors described in this section.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NMSQG} Any memory location that has a Normal Cacheable attribute can be allocated into a cache, associated with the MECID of the translation that provided the Normal Cacheable attribute.

R_{FKNTM} In situations where Normal Non-cacheable memory is permitted to be cached in an instruction cache, it is associated with the MECID of the translation that provided the Non-cacheable attribute.

I_{RDDXP} Speculative allocation of Normal memory only occurs from the current Exception level or a higher Exception level.

R_{MVGSH} If a Normal memory location is allocated to a cache entry due to a write access or a cache writeback, then the cache entry is associated with the MECID of that write access.

R_{KZCZX} If a cache entry for a Normal memory location is updated due to a write access or a cache writeback, then it is optional to associate the cache entry with the MECID of that write access. If FEAT_RME_GDI is implemented, and the PA space is System Agent (SA) or Non-secure Protected (NSP), then the MECID value is updated on a write access or a cache writeback.

R_{PJNMC} If a write access targets a location that is cached and associated with a different MECID, and the write access is smaller than the Cache writeback granule, then:

- If the PA space is not SA or NSP, one of the following CONSTRAINED UNPREDICTABLE behaviors occur:
  - The write access succeeds, the location contents become UNKNOWN, and the MECID value associated with the cache entry is updated.
  - The write access succeeds as though the MECID values did not mismatch, and it is OPTIONAL whether the MECID value associated with the cache entry is updated.
  - The write access succeeds, the written location contents are updated, the unwritten locations within the Cache writeback granule become UNKNOWN, and the MECID value associated with the cache entry is updated.
- If the PA space is SA or NSP, then the write access succeeds, the written location contents are updated, the unwritten locations within the Cache writeback granule become UNKNOWN, and the MECID value associated with the cache entry is updated.

R_{VXMVW} If a read access, including a data read, instruction fetch, or a translation table lookup, targets a location that is cached and associated with a different MECID, then:

- If the PA space is not SA or NSP, one of the following CONSTRAINED UNPREDICTABLE behaviors occur:
  - The data returned by the read access is UNKNOWN.
  - The read access succeeds as though the MECID values did not mismatch.
- If the PA space is SA or NSP, then the data returned by the read access is UNKNOWN.

I_{HJXYJ} Arm recommends that the value of all ones is used for the UNKNOWN data, in R_{PJNMC} and R_{VXMVW}.

I_{MWGQN} FEAT_RME_GDI promotes MEC to a primary security mechanism across contexts within the NSP and SA PA spaces. R_{KZCZX}, R_{PJNMC}, R_{VXMVW} and I_{HJXYJ} are designed to prevent plain text data leakages in caches, across contexts. They do not prevent data corruption across contexts.

R_{NTHZN} A MECID mismatch that results in UNKNOWN data does not cause a plaintext data leak between MECIDs.

I_{TYGBH} If a cache detects a MECID mismatch, then it is permitted to record the PA and mismatched MECIDs and generate an IMPLEMENTATION DEFINED interrupt.

R_{BSTPQ} Associating Normal Cacheable memory accesses with a MECID is required whether or not the target location is encrypted.

D7.5.5 Enabling and disabling the caching of memory accesses

Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

The Cacheability control fields and their effects are as follows:

For the EL1&0 translation regime

- When the value of SCTLR_EL1.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR_EL1.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 2 translation tables are Non-cacheable.
- When the value of HCR_EL2.ID is 1:
  - All stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.DC is 1, all stage 1 translations and all accesses to the EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTLR_EL1.{I, C}. This applies to translations for both data and instruction accesses.

Note

- The stage 1 and stage 2 cacheability attributes are combined as described in Combining stage 1 and stage 2 Cacheability attributes for Normal memory.
- The SCTLR_EL1.{C, I} and HCR_EL2.DC fields have no effect on the EL2, EL2&0, and EL3 translation regimes.
- The HCR_EL2.{ID, CD} fields affect only stage 2 of the EL1&0 translation regime.
- When FEAT_XS is implemented, the SCTLR_EL1.{C, I} and HCR_EL2.{ID, CD} fields have no effect on the value of the XS attribute.
- When EL2 is using AArch64 and EL1 is using AArch32, the HCR_EL2.{ID, CD, DC} controls apply as described here, but the EL1 controls are SCTLR.{C, I}.

For the EL2 translation regime

- When the value of SCTLR_EL2.C is 0:
  - All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
  - All accesses to the EL2 translation tables are Non-cacheable.
- When the value of SCTLR_EL2.I is 0:
  - All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

Note

- The SCTLR_EL2.{I, C} fields have no effect on the EL1&0 and EL3 translation regimes.
- When FEAT_XS is implemented, the SCTLR_EL2.{I, C} fields have no effect on the value of the XS attribute.

For the EL2&0 translation regime

- When the value of SCTLR_EL2.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL2&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR_EL2.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When FEAT_XS is implemented, the SCTLR_EL2.{I, C} fields have no effect on the value of the XS attribute.

For the EL3 translation regime

- When the value of SCTLR_EL3.C is 0:
  - All data accesses to Normal memory using the EL3 translation regime are Non-cacheable.
  - All accesses to the EL3 translation tables are Non-cacheable.
- When the value of SCTLR_EL3.I is 0:
  - All instruction accesses to Normal memory using the EL3 translation regime are Non-cacheable.

Note

- The SCTLR_EL3{I, C} fields have no effect on the EL1&0, EL2, and EL2&0 translation regimes.
- When FEAT_XS is implemented, the SCTLR_EL3.{I, C} fields have no effect on the value of the XS attribute.

In addition:

- For translation regimes other than the EL1&0 translation regime, if the value of SCTLR_ELx.M is 0, indicating that stage 1 translations are disabled for that translation regime, then:
  - If the value of SCTLR_ELx.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
  - If the value of SCTLR_ELx.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.
- For the EL1&0 translation regime, if the value of SCTLR_EL1.M is 0, indicating that stage 1 translations are disabled for that translation regime, and the value of HCR_EL2.DC is 0:
  - If the value of SCTLR_EL1.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
  - If the value of SCTLR_EL1.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through Cacheable, Outer Write-Through Cacheable.

The effect of SCTLR_ELx.C, HCR_EL2.DC and HCR_EL2.CD is reflected in the result of the address translation instructions in the PAR when these bits have an effect on the stages of translation being reported in the PAR.

Note

- In conjunction with the requirements in Non-cacheable accesses and instruction caches, the requirements in this section mean the architecturally required effect of SCTLR_ELx.I is limited to its effect on caching instruction accesses in unified caches.
- This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in Mismatched memory attributes must be followed to manage the corresponding loss of coherency.

D7.5.6 Behavior of caches at reset

The behavior of caches at reset is as follows:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in Enabling and disabling the caching of memory accesses reset to values that force all memory locations to be treated as Non-cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This applies only to the controls that apply to the Translation regime that is used by the Exception level and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

  In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.
- Arm recommends that whenever an invalidation routine is required, it is based on the cache maintenance instructions.

See also TLB behavior at reset.

D7.5.7 Non-cacheable accesses and instruction caches

In AArch64 state, instruction accesses to Non-cacheable Normal memory can be held in instruction caches.

Correspondingly, the sequence for ensuring that modifications to instructions are available for execution must include invalidation of the modified locations from the instruction cache, even if the instructions are held in Normal Non-cacheable memory. This includes cases where System register Cacheability control fields force instruction accesses to memory to be Non-cacheable.

Therefore when using self-modified code in Non-cacheable space in a uniprocessor system, the following sequence is required:

```
; Enter this code with <Wt> containing the new 32-bit instruction
; to be held at a location pointed to by <Xn> in Normal Non-cacheable memory.
STR <Wt>, [Xn]
DSB ISH; Ensure visibility of the data stored
IC IVAU, [Xn] ; Invalidate instruction cache by VA to PoU
DSB ISH; Ensure completion of the invalidations
ISB ;
```

In a multiprocessor system, the IC IVAU for a non-cacheable location is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. This is despite non-cacheable normal memory locations being treated as Outer Shared in other parts of the architecture.

Additional software steps might be required to synchronize the threads with other PEs. This might be necessary so that the PEs executing the modified instructions can execute an `ISB` after completing the invalidation, and to avoid issues associated with concurrent modification and execution of instruction sequences. See also Concurrent modification and execution of instructions and Concurrent modification and execution of instructions.

Larger blocks of instructions can be modified using the IC IALLU instruction for a uniprocessor system, or an IC IALLUIS for a multiprocessor system.

Note

This section applies even when the Cacheability control fields force instruction accesses to memory in AArch64 state to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses.

D7.5.8 About cache maintenance in AArch64 state

The following sections give general information about cache maintenance:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Terms used in describing the cache maintenance instructions.
- Abstraction of the cache hierarchy.

The following sections describe the A64 cache maintenance instructions:

- The instruction cache maintenance instruction (IC).
- The data cache maintenance instruction (DC).

Note

Some descriptions of the cache maintenance instructions refer to the cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

D7.5.8.1 Terms used in describing the cache maintenance instructions

A *memory location* is a byte that is associated with an address in a particular address space. For example, address `0x1000` in the Root physical address space is a different memory location from address `0x1000` in the Secure physical address space.

The term *Resource* means a physical entity that can be accessed at one or more memory locations. A Resource *associated with* a physical address space is accessible in that physical address space.

Note

Examples of a Resource include:

- An MMIO register that is accessible at both the memory location with address `0x2000` in the Non-secure physical address space, and at memory location with address `0x2000` in the Secure address space.
- An SRAM that is accessible only at the memory location with address `0x3000` in the Root physical address space.
- A byte of memory that can be accessible at a fixed address but in different physical address spaces, determined by a configuration option.

Cache maintenance instructions are defined to act on particular memory locations. Depending on the instruction type, the scope is defined as one of:

- By the virtual address of the memory location to be maintained, referred to as operating *by VA*.
- By the physical address of the memory location to be maintained, referred to as operating *by PA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- Terminology for cache maintenance instructions operating by set/way.
- Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see The effects of disabling an address translation stage.

D7.5.8.1.1 Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See Memory hierarchy.

Set Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the Arm architecture, sets are numbered from 0.

Way The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

D7.5.8.1.2 Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

Clean A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the shareability domain of that memory location.

Invalidate A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Clean and Invalidate

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instructions operating by VA:
  - The instructions that FEAT_OCCMO implements operate to the outer cache level.
  - For all other instructions operating by VA, the following conceptual points are defined:

Point of Coherency (PoC)

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

Note

The presence of system caches can affect the determination of the Point of Coherency as described in System level caches.

Point of Physical Aliasing (PoPA)

The point at which updates to one memory location of a Resource are visible to all other memory locations of that Resource, for accesses to that point of any memory type or cacheability attribute, for all agents that can access memory. The relationship between the PoPA and the PoC is such that a clean of a written memory location to the PoPA means that no agent in the system can subsequently reveal an old value of the memory location by performing an invalidate operation to the PoC.

Point of Encryption (PoE)

The point in the memory system where any write that has reached that point is encrypted with the context associated with the MECID that is associated with that write.

Cache maintenance operations to the PoPA are sufficient to affect all caches before the PoE.

Point of Unification (PoU)

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

Point of Persistence (PoP)

When FEAT_DPB is implemented: The point in a memory system, if it exists, at or beyond the Point of Coherency, where a write to memory is maintained when system power is removed, and reliably recovered when power is restored to the affected locations in memory.

When FEAT_DPB and FEAT_DPB2 are implemented: The point in a memory system where there is a system guarantee that there is sufficient energy within the system to ensure that a write to memory will be persistent if system power is removed.

Note

Such memory is sometimes called non-volatile memory. For example, the Storage-class memory shown in Figure B2-1 could be used as target memory for this feature.

Point of Deep Persistence (PoDP)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The point in a memory system where any writes that have reached that point are persistent, even in the event of an instantaneous hardware failure of the power system.

Point of Physical Storage (PoPS)

The point in the memory hierarchy from where, following a clean and invalidate of a Location to the Point of Physical Storage, the first subsequent read of the Location will always return data.

Note

The Point of Physical Storage is usually after all caches in the system, including system caches that are invisible to the software for the purpose of managing coherency, such as those described in System level caches.

The following fields in the CLIDR_EL1 relate to the PoC and PoU:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

LoUU, Level of Unification, uniprocessor

This field defines the last level of data cache that must be cleaned, or the last level of instruction cache that must be invalidated, when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of data cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

- This field defines the last level of data or unified cache that must be cleaned, or the last level of instruction or unified cache that must be invalidated, when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is `0x0`, this means that no levels of data or unified cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

D7.5.8.2 Abstraction of the cache hierarchy

The following subsections describe the abstraction of the cache hierarchy:

- Cache maintenance instructions that operate by VA.
- Cache maintenance instructions that operate by set/way.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.5.8.2.1 Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being one of:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.
- When FEAT_DPB is implemented, performed to the Point of Persistence.
- When FEAT_DPB2 is implemented, performed to the Point of Deep Persistence.
- When FEAT_PoPS is implemented, performed to the Point of Physical Storage.
- When FEAT_OCCMO is implemented, performed to the Outer Cache level.

See Terms used in describing the cache maintenance instructions for definitions of these terms, and for more information about possible meanings of VA.

A64 Cache maintenance instructions lists the VA-based maintenance instructions.

The CTR_EL0 holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of VAs, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR_EL0 defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

D7.5.8.2.2 Cache maintenance instructions that operate by set/way

A64 Cache maintenance instructions lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

D7.5.9 A64 Cache maintenance instructions

The A64 cache maintenance instructions are part of the A64 System instruction class in the register encoding space. For encoding details and other general information on these System instructions, see System instructions, SYS and Cache maintenance instructions, and data cache zero operations.

The following subsections give more information about these instructions:

- The instruction cache maintenance instruction (IC).
- The data cache maintenance instruction (DC).
- EL0 accessibility of cache maintenance instructions.
- General requirements for the scope of maintenance instructions.
- Effects of instructions that operate by VA to the PoC.
- Effects of instructions that operate by VA to the Outer Cache level.
- Effects of instructions that operate by VA to the PoP.
- Effects of instructions that operate by VA to the PoPS.
- Effects of instructions that operate by VA to the PoU.
- Effects of All and set/way maintenance instructions.
- Effects of virtualization and Security state on the cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Boundary conditions for cache maintenance instructions.
- Ordering and completion of data and instruction cache instructions.
- Performing cache maintenance instructions.

D7.5.9.1 The instruction cache maintenance instruction (IC)

System instructions describes the A64 assembly syntax for this instruction.

When an IC instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, or at EL2 when the Effective value of HCR_EL2.E2H is 1, the current ASID.
- The current Security state.
- Whether the instruction was executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the instruction can generate:
  - An Access flag fault.
  - A Translation fault.
- The instruction cannot generate a Permission fault, except for:
  - The possible generation of a Permission fault by the execution of an IC IVAU instruction at EL0 when the specified address does not have read access at EL0, as described in EL0 accessibility of cache maintenance instructions.
  - When FEAT_CMOW is implemented, the possible generation of a Permission fault by:
    - The execution of an IC IVAU instruction at EL0 when the specified address has stage 1 read permission, but does not have stage 1 write permission.
    - The execution of an IC IVAU instruction at EL1 or EL0 when the specified address does not have stage 2 write permission.
  - The possible Permission fault on a Stage 2 fault on a stage 1 translation table walk.

For more information about possible faults on a cache maintenance instruction that operates by VA, see Memory aborts.

See also Ordering and completion of data and instruction cache instructions.

If CTR_EL0.DIC is 1, then it is IMPLEMENTATION DEFINED whether each of the IC instructions is treated as a NOP.

If an IC instruction that operates by VA is treated as a NOP, it does not perform translation and all the following apply:

- No MMU faults can be generated.
- If hardware update of AF is enabled, AF is not required to be set.

If an IC instruction is treated as a NOP, then it is IMPLEMENTATION DEFINED whether the execution of the instruction can be trapped.

D7.5.9.2 The data cache maintenance instruction (DC)

System instructions describes the A64 assembly syntax for this instruction.

When a DC instruction requires a set/way/level argument this takes the form of a 64-bit register, the upper 32 bits of which are RES0.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see Out of range values of the Set/Way/Index fields in cache maintenance instructions or the instruction description.

When a DC instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an instruction executed at EL1, or at EL2 when the Effective value of HCR_EL2.E2H is 1, the current ASID.
- The current Security state.
- Whether the instruction is executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, a data or unified cache maintenance instruction that operates by VA cannot generate a Permission fault except in the following cases:

- The possible generation of a Permission fault by:
  - The execution of a DC IVAC instruction when the specified address does not have write permission.
  - The execution of an enabled `DC` instruction at EL0 when the specified address does not have read access at EL0, as described in EL0 accessibility of cache maintenance instructions.

  The description of Permission faults includes possible constraints on the generation of Permission faults on cache maintenance by VA instructions.
- When FEAT_CMOW is implemented, the possible generation of a Permission fault by:
  - The execution of a DC instruction that operates by VA and performs clean and invalidate at EL0 when the specified address does not have stage 1 write permission.
  - The execution of a DC instruction that operates by VA and performs clean and invalidate at EL1 or EL0 when the specified address does not have stage 2 write permission.
- The possible Permission fault on a Stage 2 fault on a stage 1 translation table walk.

For more information about possible faults on a VA to PA translation, see Memory aborts.

When executed at EL1, a DC ISW instruction performs a clean and invalidate, meaning it performs the same maintenance as a DC CISW instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - The value of HCR_EL2.SWIO is 1, forcing a cache clean to perform a clean and invalidate.
  - The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

When executed at EL1, a DC IVAC instruction performs a clean and invalidate, meaning it performs the same maintenance as a DC CIVAC instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

Note

The forcing of a clean instruction to perform a clean invalidate applies to the AArch32 cache maintenance instructions DCIMVAC and DCISW. See AArch32 data cache maintenance instructions (DC*).

When FEAT_DPB is implemented, meaning the DC CVAP instruction is implemented, if the memory system does not support the Point of Persistence, a data cache clean to the PoP, DC CVAP, behaves as a data cache clean to the PoC, DC CVAC.

Note

- Support for the Point of Persistence does not change the definition or behavior of the CLIDR_EL1 System register.
- Because a `DSB SYS` instruction will not complete until all previous DC CVAP instructions have completed, the following sequence can be used to ensure the completion of any store to the Point of Persistence, where the store might be to Non-cacheable memory:

```
DMB             ; Note this can be any DMB that applies to both loads and stores
DC CVAP, Xt
DSB SYS
```

- If caches that are invisible to the programmer exist beyond the Point of Coherency but before the Point of Persistence and hold data that is marked as Non-cacheable, the DC CVAP operation causes the Non-cacheable locations to be cleaned from those caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If there are caches after the Point of Coherency and FEAT_PoPS is not implemented, then the DC CIVAC and DC CIGDVAC instructions are not sufficient to remove all copies of a poisoned Location and it is IMPLEMENTATION DEFINED whether any IMPLEMENTATION DEFINED mechanism exists to remove poison from a Location.

See Clearing Deferred errors from poisoned Locations for more information on removing poison from a Location.

If a memory fault that sets the FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

Note

Despite its mnemonic, DC ZVA is not a cache maintenance instruction.

See also EL0 accessibility of cache maintenance instructions and Ordering and completion of data and instruction cache instructions.

For all the following DC instructions, it is IMPLEMENTATION DEFINED whether the instruction is treated as a `NOP`:

- A DC to the PoU and the LoUIS and LoUU are 0.
- A DC to the PoC and the LoC is 0.
- A DC to the PoP and the memory system does not support the Point of Persistence, and the LoC is 0.
- A DC to the PoDP and the memory system does not support the Point of Deep Persistence, and does not support the Point of Persistence, and the LoC is 0.

If FEAT_OCCMO is implemented, and in the system the PoC is before any levels of cache, then Arm expects it is possible to implement DC CIVAOC, DC CIGDVAOC, DC CVAOC, and DC CGDVAOC as `NOP`s. In systems with no Outer cache levels, these instructions can be implemented respectively as DC CIVAC, DC CIGDVAC, DC CVAC, and DC CGDVAC.

If a DC instruction that operates by VA is treated as a `NOP`, it does not perform translation and all the following apply:

- No MMU faults can be generated.
- If hardware update of AF is enabled, AF is not required to be set.

If a DC instruction is treated as a `NOP`, then it is IMPLEMENTATION DEFINED whether the execution of the instruction can be trapped.

D7.5.9.3 EL0 accessibility of cache maintenance instructions

Software executing at EL0 can access data cache maintenance instructions and instruction cache maintenance instructions that operate by virtual address. When EL0 use of these instructions is disabled because SCTLR_EL1.UCI is 0, executing one of these instructions at EL0 generates a trap to EL1, which is reported using EC = `0x18`. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the control is from SCTLR_EL2.

Note

DC CVAP is implemented only if FEAT_DPB is implemented.

For these instructions read access permission for the virtual address is required. When the value of SCTLR_EL1.UCI is 1:

- If the `DC` instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, a Permission fault might be generated.
- For the IC IVAU instruction, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, it is IMPLEMENTATION DEFINED whether a Permission fault is generated.
- When FEAT_CMOW is implemented, for the DC instructions that operate by VA and perform clean and invalidate, and the IC instructions that operate by VA, write access permission is also required as follows:
  - When SCTLR_EL1.CMOW is 1, if the instruction executed at EL0 does not have stage 1 write permission, a Permission fault is generated.
  - When HCRX_EL2.CMOW is 1, if the instruction executed at EL0 does not have stage 2 write permission, a Permission fault is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This stage 2 access permission also applies to the DCCIMVAC and ICIMVAU AArch32 cache maintenance instructions.

For more information, see the description of Permission faults, In the case of a `DC` instruction executed at EL0 when the address specified cannot be read at EL0 the Permission fault is generated unless one of the permitted constraints described in that section applies and means the fault cannot be generated.

Software can read the CTR_EL0 to discover the stride needed for cache maintenance instructions. The SCTLR_EL1.UCT bit enables EL0 access to the CTR_EL0. When EL0 access to the Cache Type register is disabled, a register access instruction executed at EL0 is trapped to EL1 using EC = `0x18`.

D7.5.9.4 General requirements for the scope of maintenance instructions

The specification of the cache maintenance instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in General behavior of the caches, Behavior of caches at reset, and Prefetching into cache.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

Note

Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in one Security state do not have a significant performance impact on execution in another Security state.

If an implementation can overwrite Allocation Tags in memory that have been written by another observer, where the Allocation Tags have not been written by an observer in the Shareability domain of that memory location, then:

- A cache maintenance operation which cleans data from a cache level must also clean the associated Allocation Tags.
- A cache maintenance operation which invalidates, or cleans and invalidates data from a cache level, must also clean and invalidate the associated Allocation Tags.

D7.5.9.5 Effects of instructions that operate by VA to the PoC

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoC affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory or Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instructions are operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoC. Table D7-3 shows the scope of these Data and unified cache maintenance instructions.

Table D7-3 PEs affected by cache maintenance instructions to the PoC

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoC of the entire system
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoC of the entire system
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The PoC of the entire system

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D7.5.9.6 Effects of instructions that operate by VA to the Outer Cache level

The behaviors in this section apply if FEAT_OCCMO is implemented.

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the Outer cache level affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data caches to the Outer cache level. Table D7-4 shows the scope of these Data cache maintenance instructions.

Table D7-4 PEs affected by cache maintenance instructions to the Outer cache level

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The Outer cache level
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The Outer cache level
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The Outer cache level

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D7.5.9.7 Effects of instructions that operate by PA to the PoPA

Caches are affected by cache maintenance instructions to the PoPA. For those caches, the operations behave as a clean and invalidate. Data cache maintenance instructions that operate by PA to the PoPA have all of the following properties:

- The instructions affect all caches in the Outer Shareable shareability domain to the PoPA for all copies of the memory location specified by the instruction.
- The instructions are permitted to affect other memory locations of the same Resource, For example, a cached copy of the same Resource that is associated with a different PA space. If multiple memory locations of the Resource have been written, it is CONSTRAINED UNPREDICTABLE which additional copies are cleaned to the PoPA. This CONSTRAINED UNPREDICTABLE behavior is guaranteed to be avoided if granule protection checks are configured to ensure that only one memory location of the Resource is writable at any time.
- The instructions affect all caches before the PoPA, even if the caches are after the PoC and are otherwise invisible to the programmer.
- The instructions have the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain. This includes aspects relating to the minimum size of cache lines, indicated by CTR_EL0.DminLine.
- The instructions are not subject to granule protection checks.
- If the instructions target a PA above the implemented PA size, then no cache entries are required to be cleaned or invalidated.

In a system that contains caches associated with observers outside the Outer Shareable domain, for each of those caches at least one of the following properties must apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The cache is affected by `DC PAPA` operations. For that cache, it is permitted for `DC PAPA` operations to be treated as invalidate operations rather than clean and invalidate operations.
- Any accesses from the cache that propagate into the Outer Shareable domain are subject to granule protection checks, and the system additionally provides one of the following properties:
  - The cache can only store memory locations from the Non-secure physical address space.
  - Accesses from the cache are subject to translation controlled by the Security state associated with the cache line.

D7.5.9.8 Effects of instructions that operate by PA to the PoE

R_{JBRCM} Data cache maintenance instructions that operate by PA to the PoE have all of the following properties:

- The instructions affect all caches in the Outer Shareable shareability domain to the PoE for all copies of the memory location specified by the instruction.
- The instructions affect all caches before the PoE, even if the caches are after the PoC and are otherwise invisible to the programmer.
- The instructions have the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain. This includes aspects relating to the minimum size of cache lines, indicated by CTR_EL0.DminLine.
- The instructions clean and invalidate all copies of the memory location specified by the instruction, irrespective of any MECID associated with the memory location. Memory accesses resulting from the Clean operation use the MECID associated with the cache entry.
- It is IMPLEMENTATION DEFINED whether the instructions are subject to granule protection checks.
- If HCR_EL2.NV is 1, then executing the instructions at Realm EL1 is trapped to Realm EL2. Exceptions generated by these traps are reported using EC = `0x18` with its associated ISS field.

D7.5.9.9 Effects of instructions that operate by VA to the PoP

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoP affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory or Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instructions are operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoP. Table D7-5 shows the scope of these Data and unified cache maintenance to the PoP instructions.

Table D7-5 PEs affected by cache maintenance instructions to the PoP

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoP of the entire system
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoP of the entire system
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The PoP of the entire system

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D7.5.9.10 Effects of instructions that operate by VA to the PoPS

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoPS affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For Device memory or Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instructions are operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoPS. Table D7-6 shows the scope of these Data and unified cache maintenance to the PoPS instructions.

Table D7-6 PEs affected by cache maintenance instructions to the PoPS

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction.	The PoPS of the entire system.
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction.	The PoPS of the entire system.
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction.	The PoPS of the entire system.

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D7.5.9.11 Effects of instructions that operate by VA to the PoU

For cache maintenance instructions that operate by VA to the PoU, Table D7-7 shows how, for a VA in a Normal or Device memory location, the shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table D7-7 PEs affected by cache maintenance instructions to the PoU

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction
Inner Shareable or Outer Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain as the PE executing the instruction

Note

- The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable shareability domain containing the PE executing the instruction.
- It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D7.5.9.12 Effects of All and set/way maintenance instructions

The DC set/way instructions apply only to the caches of the PE that performs the instruction. IC IALLU instructions apply only to the caches of the PE that performs the instruction, unless HCR_EL2.FB=1, which causes the instructions to be broadcast within the Inner Shareable domain when executed from EL1.

The IC IALLUIS instruction can affect the caches of all PEs in the same Inner Shareable shareability domain as the PE that performs the instruction. This instruction has an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- The possible presence of system caches, as described in System level caches, means architecture does not guarantee that all levels of the cache can be maintained using set/way instructions.
- It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D7.5.9.13 Effects of virtualization and Security state on the cache maintenance instructions

Each Security state has its own physical address (PA) space, therefore cache entries are associated with PA space.

Table D7-8 shows the effects of virtualization and security on the cache maintenance instructions. In the table, the *Specified entries* are entries that the architecture requires the instruction to affect.

Note

The rules described in General behavior of the caches mean that an instruction might also affect other entries.

Table D7-8 Effects of virtualization and security on the maintenance instructions

Cache maintenance instructions	Security state	Specified entries
Data or unified cache maintenance instructions		
Invalidate, Clean, or Clean and Invalidate by VA: DC IVAC, DC CVAC, DC CVAP, DC CVAU, DC CIVAC, DC CVAP, DC CIVAPS, DC CIGDVAPS, DC CIVAOC, DC CIGDVAOC, DC CVAOC, DC CGDVAOC.	Non-secure and Secure. When FEAT_RME is implemented, Realm and Root.	All lines that hold the PA that, in the current Security state, is mapped to by the combination of all of: • The specified VA. • For an instruction executed at EL1, EL0, or at EL2 when the Effective value of HCR_EL2.E2H is 1, the current ASID if the location is mapped to by a non-global page. • For an instruction executed at EL1 when SCR_EL3.NS is 1 or SCR_EL3.EEL2 is 1, the current VMID.[a] • For an instruction executed at EL0 when (SCR_EL3.NS is 1 or SCR_EL3.EEL2 is 1) and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the current VMID.[a]
Invalidate, Clean, or Clean and Invalidate by set/way: DC ISW, DC CSW, DCCISW, DC IGSW, DC CGSW, DC CIGSW, DC IGDSW, DC CGDSW, DC CIGDSW	Non-secure	Line specified by set/way provided that the entry comes from the Non-secure PA space.
	Secure	Line specified by set/way provided that the entry comes from the Secure or Non-secure PA space.
	When FEAT_RME is implemented, the following cases are added:	
	Realm	Line specified by set/way provided that the entry comes from the Realm or Non-secure PA space.
	Root	Line specified by set/way regardless of the PA space that the entry has come from.
Instruction cache maintenance instructions		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cache maintenance instructions	Security state	Specified entries
Invalidate by VA: IC IVAU	Non-secure and Secure. When FEAT_RME is implemented, Realm and Root.	All lines corresponding to the specified VA[b] in the current translation regime and: • For an instruction executed at EL1, EL0, or at EL2 when the Effective value of HCR_EL2.E2H is 1, the current ASID. • For an instruction executed at EL1 when SCR_EL3.NS is 1 or SCR_EL3.EEL2 is 1, the current VMID.[a] • For an instruction executed at EL0 when (SCR_EL3.NS is 1 or SCR_EL3.EEL2 is 1) and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the current VMID.[a]
Invalidate All: IC IALLU, IC IALLUIS	Non-secure and Secure.	For an instruction executed at: • EL1 when the Effective value of SCR_EL3.{NSE, NS} is {0, 0} and SCR_EL3.EEL2 == 1, all instruction cache lines containing Secure or Non-secure entries associated with the current VMID. • EL1 when the Effective value of SCR_EL3.{NSE, NS} is {0, 1}, all instruction cache lines containing Non-secure entries associated with the current VMID. • EL2 when the Effective value of SCR_EL3.{NSE, NS} is {0, 1}, all instruction cache lines containing Non-secure entries. • EL1 when the Effective value of SCR_EL3.{EEL2, NSE, NS} is {0, 0, 0}, EL2 when SCR_EL3.{EEL2, NSE, NS} is {1, 0, 0}, or EL3, all instruction cache lines containing Secure or Non-secure entries.
	Realm	For an instruction executed at: • EL1 when the Effective value of SCR_EL3.{NSE, NS} is {1, 1}, all instruction cache lines containing Realm or Non-secure entries associated with the current VMID. • EL2 when the Effective value of SCR_EL3.{NSE, NS} is {1, 1}, all instruction cache lines containing Realm or Non-secure entries.
	Root	For an instruction executed at EL3, all instruction cache lines regardless of the PA space that the entries have come from.

a. Dependencies on the VMID apply even when HCR_EL2.VM is set to 0. The architecture does not define a reset value for VTTBR_EL2.VMID, and therefore, in any implementation that includes EL2, the boot software executed when reset is deasserted must initialize VTTBR_EL2.VMID.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:

- For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
- For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.

For information on types of instruction cache see Instruction caches.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in The interaction of cache lockdown with cache maintenance instructions applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, invalidate by set/way instructions can, instead, perform a clean and invalidate by set/way.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- As described in The data cache maintenance instruction (DC), the AArch64 Data cache invalidate instructions, DC IVAC and DC ISW, when executed at EL1 and EL0,

  and the AArch32 Data cache invalidate instructions DCIMVAC and DCISW, when executed at EL1,

  can be configured to perform a cache clean as well as a cache invalidation.
- TLB invalidate instructions and IC IALLU instructions executed at EL1 are broadcast across the Inner Shareable domain when all of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The value of HCR_EL2.FB is 1.

For more information about the cache maintenance instructions, see About cache maintenance in AArch64 state, A64 Cache maintenance instructions, and The AArch64 Virtual Memory System Architecture.

D7.5.9.14 Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instruction operates on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  - Is Normal memory or Device memory.
  - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

D7.5.9.15 Ordering and completion of data and instruction cache instructions

All data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address:

- Execute in program order relative to loads or stores that have all of the following properties:
  - Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  - Use an address with the same cacheability attributes as the address passed to the data cache instruction.
- Can execute in any order relative to loads or stores that have all of the following properties:
  - Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  - Use an address with different cacheability attributes as the address passed to the data cache instruction.
  - Do not have a `DMB` or `DSB` executed between the load or store instruction and the data cache instruction.
- Can execute in any order relative to loads or stores that access any address with the Device memory attribute, or with Normal memory with Inner Non-cacheable attribute unless a `DMB` or `DSB` is executed between the instructions.
- Execute in program order relative to other data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
- Can execute in any order relative to loads or stores that access an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to other data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a `DSB` is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a `DMB` or `DSB` is executed between the instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Despite their mnemonics, the DC ZVA, DC GVA, and DC GZVA instructions are not data cache maintenance instructions.

Note

- Data cache ordering rules by address are consistent with physically indexed physically tagged caches. See Data and unified caches.
- Data cache zero instruction describes the ordering and completion rules for Data Cache Zero.

All data cache maintenance instructions that do not specify an address:

- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that specify an address, other than `DC ZVA`, `DC GVA`, and `DC GZVA`, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to loads or stores unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a `DSB` is executed between the instructions.

All instruction cache maintenance instructions can execute in any order relative to other instruction cache instructions, data cache instructions, loads, and stores unless a `DSB` is executed between the instructions.

A cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a `DSB` instruction executed by the PE that executed the cache maintenance instruction. See also the requirements for cache maintenance instructions in Completion and endpoint ordering.

In all cases, where the text in this section refers to a `DMB` or a `DSB`, this means a `DMB` or `DSB` whose required access type is both loads and stores.

Note

This section specifies the memory ordering requirements of `DMB` and `DSB` based on the required access type as specified by the <option> parameter. `DSB` has additional requirements with respect to execution of instructions appearing in program order after the `DSB` as described in Data Synchronization Barrier. These additional requirements create the effect of additional memory ordering beyond what is specified in this section.

Note

These ordering requirements are extended from the requirements in AArch32 state given in:

- Ordering of cache and branch predictor maintenance instructions.
- AArch32 instruction cache maintenance instructions (IC*).

D7.5.9.15.1 Ordering and completion of Data Cache Clean to Point of Persistence

The update of the persistent memory as a result of Data Cache Clean to the Point of Persistence is guaranteed to have occurred either after:

- The execution of a `DSB` applying to both reads and writes after the execution of the Data Cache Clean to the Point of Persistence.
- The update to persistent memory caused by a different Data Cache Clean to the Point of Persistence that is ordered after a `DMB` applying to both reads and writes that appears after the original Data Cache Clean to the Point of Persistence.

Note

This second point is an aspect of the fact that the Data Cache Clean to the Point of Persistence instructions are ordered by `DMB`, and this controls the order of arrival in persistent memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The ordering effect for the Data Cache Clean to the Point of Persistence by DMB applying to both read and writes is not sufficient to ensure that in Example D7-1, observation of the value '1' in the memory location X3 implies that the Data Cache Clean to the Point of Persistence has caused an update of persistent memory:

Example D7-1 The ordering effect for the Data Cache Clean to the Point of Persistence

```
; initial condition has [X3]=0

DC CVADP, X1
DMB
MOV X2,#1
STR X2, [X3]
```

However, in Example D7-2, the ordering effects of the DMB instruction will ensure that the location pointed by P0: X1 will reach the Point of Persistence before, or at the same time as, the location pointed by P1:X8.

Example D7-2 The ordering effect for the Data Cache Clean to the Point of Persistence

```
; initial conditions has P0: X3 and P1: X3 point to the same location, which is 0 at the start of
    ↪this example

P0
    DC CVAP, X1
    DMB
    MOV X2, #1
    STR X2, [X3]

P1
loop
    LDR X2, [X3]
    CBZ X2, loop
    DMB
    DC CVAP, X8
```

D7.5.9.15.2 Ordering and completion of Data Cache Clean to Point of Deep Persistence

The update of the deep persistent memory as a result of Data Cache Clean to the Point of Deep Persistence is guaranteed to have occurred either after:

- The execution of a DSB applying to both reads and writes after the execution of the Data Cache Clean to the Point of Deep Persistence.
- The update to deep persistent memory caused by a different Data Cache Clean to the Point of Deep Persistence that is ordered after a DMB applying to both reads and writes that appears after the original Data Cache Clean to the Point of Deep Persistence.

Note

This second point is an aspect of the fact that the Data Cache Clean to the Point of Deep Persistence instructions are ordered by DMB, and this controls the order of arrival in deep persistent memory.

The ordering effect for the Data Cache Clean to the Point of Deep Persistence by DMB applying to both read and writes is not sufficient to ensure that in Example D7-3, observation of the value '1' in the memory location X3 implies that the Data Cache Clean to the Point of Deep Persistence has caused an update of deep persistent memory:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D7-3 The ordering effect for the Data Cache Clean to the Point of Deep Persistence

```
; initial conditions has [X3]=0

DC CVADP, X1
DMB
MOV X2,#1
STR X2, [X3]
```

However, in Example D7-4, the ordering effects of the `DMB` instruction will ensure that the location pointed by P0: X1 will reach the Point of Deep Persistence before, or at the same time as, the location pointed by P1: X8.

Example D7-4 The ordering effect for the Data Cache Clean to the Point of Deep Persistence

```
; initial conditions has P0: X3 and P1: X3 point to the same location, which is 0 at the start of
    ↪this example
P0
    DC CVADP, X1
    DMB
    MOV X2, #1
    STR X2, [X3]

P1
loop
    LDR X2, [X3]
    CBZ X2, loop
    DMB
    DC CVADP, X8
```

D7.5.9.16 Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

- For data or unified cache maintenance, uses the CTR_EL0.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the CTR_EL0.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

D7.5.9.16.1 Example code for cache maintenance instructions

The cache maintenance instructions by set/way can clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Since the set/way instructions are performed only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Since any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

System level caches refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes that the current Cache Size Identification Register is in 32-bit format. For more information, see Possible formats of the Cache Size Identification Register, CCSIDR_EL1.

```
        MRS     X0, CLIDR_EL1
        AND     W3, W0, #0x07000000     // Get 2 x Level of Coherence
        LSR     W3, W3, #23
        CBZ     W3, Finished
        MOV     W10, #0                 // W10 = 2 x cache level
        MOV     W8, #1                  // W8 = constant 0b1
Loop1:  ADD     W2, W10, W10, LSR #1    // Calculate 3 x cache level
        LSR     W1, W0, W2              // extract 3-bit cache type for this level
        AND     W1, W1, #0x7
        CMP     W1, #2
        B.LT    Skip                    // No data or unified cache at this level
        MSR     CSSELR_EL1, X10         // Select this cache level
        ISB                             // Synchronize change of CSSELR
        MRS     X1, CCSIDR_EL1          // Read CCSIDR
        AND     W2, W1, #7              // W2 = log2(linelen)-4
        ADD     W2, W2, #4              // W2 = log2(linelen)
        UBFX    W4, W1, #3, #10         // W4 = max way number, right aligned
        CLZ     W5, W4                  // W5 = 32-log2(ways), bit position of way in DC operand
        LSL     W9, W4, W5              // W9 = max way number, aligned to position in DC operand
        LSL     W16, W8, W5             // W16 = amount to decrement way number per iteration
Loop2:  UBFX    W7, W1, #13, #15        // W7 = max set number, right aligned
        LSL     W7, W7, W2              // W7 = max set number, aligned to position in DC operand
        LSL     W17, W8, W2             // W17 = amount to decrement set number per iteration
Loop3:  ORR     W11, W10, W9            // W11 = combine way number and cache number ...
        ORR     W11, W11, W7            //  ... and set number for DC operand
        DC      CSW, X11                // Do data cache clean by set and way
        SUBS    W7, W7, W17             // Decrement set number
        B.GE    Loop3
        SUBS    X9, X9, X16             // Decrement way number
        B.GE    Loop2
Skip:   ADD     W10, W10, #2            // Increment 2 x cache level
        CMP     W3, W10
        DSB                             // Ensure completion of previous cache maintenance instruction
        B.GT    Loop1
Finished:
```

Similar approaches can be used for all cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.5.10 Data cache zero instruction

The Data Cache Zero by Address instruction, DC ZVA, writes `0x00` to each byte of a block of *N* bytes, aligned in memory to *N* bytes in size, where:

- The block in memory is identified by the address supplied as an argument to the DC ZVA instruction. There are no alignment restrictions on this address.

  Note

  This means that each byte of the block of memory that includes the supplied address is set to zero.

- The DCZID_EL0 register indicates the block size, *N* bytes, that is written with byte values of zero.

Software can restrict access to this instruction. See Configurable instruction controls and the description of the DC ZVA instruction.

The DC ZVA instruction behaves as a set of stores to the location being accessed, and:

- Generates a Permission fault if the translation regime being used when the instruction is executed does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instruction.

In addition:

- When the instruction is executed, it can generate memory faults or watchpoints that are prioritized in the same way as other memory related faults or watchpoints. Where a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the syndrome field is not set to 1, which would be the case for all other cache maintenance instructions. See ISS encoding for an exception from a Data Abort for more information about the encoding of the associated ESR_ELx.ISS field.
- If the memory region being zeroed is any type of Device memory that does not support unaligned accesses, then `DC ZVA` generates an Alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

Note

The architecture makes no statements about whether or not a DC ZVA instruction causes allocation to any particular level of the cache, for addresses that have a cacheable attribute for those levels of cache.
Despite its mnemonic, the DC ZVA instruction is not a data cache maintenance instruction.

D7.5.11 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

D7.5.11.1 The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- Cache maintenance operations to the PoPA affect cache entries regardless of lockdown status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DC CSW.
  - Cache invalidate by set/way, DC ISW.
  - Cache clean and invalidate by set/way, DC CISW.
  - Instruction cache invalidate all, IC IALLU and IC IALLUIS.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
  2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose, see ISS encoding for an exception from a Data Abort.

  This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.
- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by virtual address, DC CVAC, DC CVAP, and DC CVAU.
  - Cache invalidate by virtual address, DC IVAC.
  - Cache clean and invalidate by virtual address, DC CIVAC.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
  2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose. See ESR_ELx.

In an implementation that includes EL2 enabled in the current Security state, if HCR_EL2.TIDCP is set to 1, any exception relating to lockdown of an entry is routed to EL2.

Note

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.
In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.
The reset setting of the cache must be that no cache entries are locked.

D7.5.11.1.1 Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED spaces reserved for Cache Lockdown, see Reserved encodings for IMPLEMENTATION DEFINED registers.

D7.5.12 System level caches

The Arm Architecture defines a *system cache* as a cache that is not described in the PE Cache Identification registers, CCSIDR_EL1 and CLIDR_EL1, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the Point of Coherency and cannot be managed by any cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the Point of Coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. System caches which lie before the Point of Coherency and can be managed by cache maintenance by address instructions that apply to the Point of Coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entirety of such a cache must be performed, as in the case for power management, it must be performed using non-architectural mechanisms.
3. System caches which lie beyond the Point of Coherency and so are invisible to the software for the purpose of managing coherency.

Note

If maintenance of the entirety of such a cache is supported, it is an IMPLEMENTATION DEFINED property of the system.

D7.5.13 Branch prediction

In the AArch64 architecture, branch predictors are not architecturally visible.

For details on the restrictions on the effects of speculation for branch predictors, see Restrictions on the effects of speculation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.6 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU or debug logic. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see RAS PE Architecture.

An External abort is one of the following:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see Exception entry terminology.

The RAS Extension provides a more granular taxonomy of aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported.

External aborts on data accesses and translation table walks on data accesses can be either synchronous or asynchronous.

When FEAT_DoubleFault is not implemented, External aborts on instruction fetches and translation table walks on instruction fetches can be either synchronous or asynchronous.

When FEAT_DoubleFault is implemented, all External abort exceptions on instruction fetches and translation table walks on instruction fetches must be synchronous.

A synchronous External abort on an instruction fetch, including a translation table walk on an instruction fetch, is taken precisely using the Instruction Abort exception.

A synchronous External abort on a data read or write, including a translation table walk on a data read or write, is taken precisely using the Data Abort exception.

See Synchronous exception types.

An asynchronous External abort is taken using the SError exception. See Asynchronous exception types.

The effect of a failed memory access is described in Definition of a precise exception and imprecise exception.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running, Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:

- Provision for the classification of External aborts.
- Parity or ECC error reporting, RAS Extension not implemented.

D7.6.1 Provision for the classification of External aborts

In AArch64 state, an implementation can use ESR_ELx.EA, ISS[9], to provide more information about synchronous External aborts. For all synchronous aborts other than synchronous External aborts, ESR_ELx.EA, ISS[9], returns a value of 0.

If the FEAT_RAS is implemented:

- The ESR_ELx.SET field provides information about the state of the PE following a synchronous External abort.
- The ESR_ELx.AET field might contain more information following an asynchronous abort taken as an SError exception.
- The implementation might define error record registers.

For more information, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ISS encoding for an exception from an Instruction Abort.
- ISS encoding for an exception from a Data Abort.
- ISS encoding for an SError exception.
- Taking error exceptions.

D7.6.2 Parity or ECC error reporting, RAS Extension not implemented

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache system. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors. However, when parity or ECC error reporting is implemented, it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.7 Memory barrier instructions

Memory barriers describes the memory barrier instructions. This section describes the system level controls of those instructions.

D7.7.1 EL2 control of the maintenance scope of Data Synchronization Barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 enabled in the current Security state and that supports maintenance scope on `DSB` instructions, the HCR_EL2.BSU field can modify the required maintenance scope of an instruction that is executed at EL0 or EL1. Table D7-9 shows the encoding of this field.

Table D7-9 EL2 control of DSB maintenance scope, for constraining both loads and stores executed at EL0 or EL1

HCR_EL2.BSU	Minimum maintenance scope of `DSB` instructions
00	No effect, maintenance scope is as specified by the instruction
01	Inner Shareable
10	Outer Shareable
11	Full system

For an instruction executed at EL0 or EL1, Table D7-10 shows how the HCR_EL2.BSU is combined with the maintenance scope specified by the argument of the `DSB` to both loads and stores to give the scope of the instruction.

Table D7-10 Effect of HCR_EL2.BSU on barrier instructions executed at EL1 or EL0

Maintenance scope specified by the `DSB` argument	HCR_EL2.BSU	Resultant maintenance scope
Full system	Any	Full system
Outer Shareable	`00`, `01`, or `10`	Outer Shareable
	`11`, Full system	Full system
Inner Shareable	`00` or `01`	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system
Non-shareable	`00`, No effect	Non-shareable
	`01`, Inner Shareable	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.8 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- Memory data type definitions.
- Basic memory access.
- Aligned memory access.
- Unaligned memory access.
- Exclusives monitors operations.
- Access permission checking.
- Abort exceptions.
- Memory barriers.

D7.8.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the `AddressDescriptor` type.
- Full address, defined by the `FullAddress` type.
- Memory attributes, defined by the `MemoryAttributes` type.
- Memory type, defined by the `MemType` enumeration.
- Device memory type, defined by the `DeviceType` enumeration.
- Normal memory attributes, defined by the `MemAttrHints` type.
- Cacheability attributes, defined by the `MemAttr_NC`, `MemAttr_WT`, and `MemAttr_WB` constants.
- Allocation hints, defined by the `MemHint_No`, `MemHint_WA`, `MemHint_RA`, and `MemHint_RWA` constants.
- Access permissions, defined by the `Permissions` type.

These types are defined in A-profile Architecture Pseudocode.

D7.8.2 Basic memory access

The `PhysMemRead()` and `PhysMemWrite()` functions perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

The attributes in `memaddrdesc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in Memory types and attributes, Ordering relations, and Atomicity in the Arm architecture.

D7.8.3 Aligned memory access

The `AArch64_MemSingle{size}()` accessor makes atomic, little-endian accesses.

D7.8.4 Unaligned memory access

The `Mem{size}()` accessor make accesses of the required type. If an access is not architecturally defined to be atomic, `Mem()` synthesizes accesses from multiple calls to `AArch64_MemSingle{}()`. It also reverses the byte order if the access is big-endian.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.8.5 Exclusives monitors operations

The `AArch64_SetExclusiveMonitors()` function sets the Exclusives monitors for a block of bytes, the size of which is determined by `size`, at the virtual address defined by `address`.

The `AArch64_ExclusiveMonitorsPass()` function checks whether the Exclusives monitors are set to include the location of a number of bytes specified by `size`, at the virtual address defined by `address`. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `AArch64_SetExclusiveMonitors()`.

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `AArch64_ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The `MarkExclusiveGlobal()` procedure takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records that the PE `processorid` has requested exclusive access covering at least `size` bytes from address `paddress`. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The `MarkExclusiveLocal()` procedure takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records in a local record that PE `processorid` has requested exclusive access to an address covering at least `size` bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

The `IsExclusiveGlobal()` function takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked in a global record an address range as exclusive access requested that covers at least `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

The `IsExclusiveLocal()` function takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of `size` bytes from address `paddress`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `ClearExclusiveByAddress()` procedure takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The procedure clears the global records of all PEs, other than `processorid`, for which an address region including any of `size` bytes starting from `paddress` has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

D7.8.6 Access permission checking

The `AArch64_S1CheckPermissions()` and `AArch64_S2CheckPermissions()` functions are used by the architecture to perform access permission checking based on attributes derived from the Translation Table descriptors.

The interpretation of access permission is shown in Memory access control.

D7.8.7 Abort exceptions

The `AArch64_Abort()` function generates either a Data Abort or an Instruction Abort exception by calling `AArch64_DataAbort()` or `AArch64_InstructionAbort()`. It also can generate a debug exception for debug

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

related faults, see AArch64 Self-hosted Debug.

The `AArch64_DataAbort()` function generates a Data Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx. See ISS encoding for an exception from a Data Abort. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64_DataAbort()` also sets the FAR to the VA of the abort.

The `AArch64_InstructionAbort()` function generates an Instruction Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx, see ISS encoding for an exception from an Instruction Abort. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64_InstructionAbort()` also sets the FAR to the VA of the abort.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault.

The function `NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a `FaultRecord` contains a fault.

D7.8.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBMaintenanceScope` and `MBReqTypes`.

These enumerations define the required shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier`, `DataSynchronizationBarrier`, and `InstructionSynchronizationBarrier` perform the memory barriers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D8
The AArch64 Virtual Memory System Architecture

This chapter provides a system level view of the AArch64 *Virtual Memory System Architecture* (VMSA), the memory system architecture of an A-profile implementation that is executing in AArch64 state. It contains the following sections:

- Address translation.
- Translation process.
- Translation table descriptor formats.
- Memory access control.
- Memory region attributes.
- Other descriptor fields.
- Address tagging.
- Pointer authentication.
- Memory Encryption Contexts extension.
- Virtualization Host Extensions.
- Nested virtualization.
- Memory aborts.
- Translation Lookaside Buffers.
- TLB maintenance.
- Caches.
- Pseudocode description of VMSA address translation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1 Address translation

R_{DWFWP} If an implementation is executing in AArch64 state, then that implementation uses either or both of the VMSAv8-64 and the VMSAv9-128 translation systems.

R_{RKHJV} All of the following determine whether a translation stage uses the VMSAv9-128 translation system:

- For stage 1 translations in the EL1&0 translation regime, the Effective value of TCR2_EL1.D128 is 1.
- For stage 1 translations in the EL2&0 translation regime, the Effective value of TCR2_EL2.D128 is 1.
- For stage 1 translations in the EL3 translation regime, the Effective value of TCR_EL3.D128 is 1.
- For stage 2 translations, the Effective value of VTCR_EL2.D128 is 1.

Otherwise, the translation stage uses the VMSAv8-64 translation system.
This applies even when the translation stage is disabled.

I_{CCTQS} Address translation converts the addresses used by instructions to the addresses used by the physical memory system.

R_{CRPHV} When a data address or instruction address is used in an instruction, it is a *virtual address* (VA). This includes any address stored in one of the following registers:

- *Program counter* (PC).
- *Stack pointers* (SP).
- *Link register* (LR).
- *Exception link register* (ELR).

R_{CRDGS} When an access is made to the physical memory system, a *physical address* (PA) is used.

R_{KCNRX} An address translation maps a VA to a PA.

R_{CCCQQ} An address translation requires one of the following:

- A single translation stage, stage 1.
- Two sequential translation stages, stage 1 and stage 2.

R_{ZKJWW} An address translation stage maps an *input address* (IA) to an *output address* (OA).

R_{BCKLH} If one address translation stage is used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the translation stage.
2. The PA is output as the OA from the translation stage.

R_{KXSMJ} If two address translation stages are used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the stage 1 translation.
2. The *intermediate physical address* (IPA) is output as the OA from the stage 1 translation.
3. The IPA is input as the IA to the stage 2 translation.
4. The PA is output as the OA from the stage 2 translation.

I_{MBNVX} If an address translation stage is disabled, then the value of the OA is the same as the IA.

I_{YXNZL} When an IA is translated to an OA, an address translation stage uses a set of memory mapped translation tables that hold all of the following information:

- The OA corresponding to the IA.
- For accesses made from Secure state, whether the OA access is to the Secure or Non-secure address map.
- The OA memory access permissions.
- The OA memory region attributes.

I_{FJFQR} When an IA is translated by an address translation stage, all of the following apply:

- A translation table lookup reads an entry from a translation table.
- A translation table entry resolves a subset of the IA.
- Multiple translation table entries can be required to completely resolve an IA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An address translation can require multiple lookups across different lookup levels and multiple translation tables.

R_{JTYJP} When memory is accessed, the *Memory Management Unit* (MMU) controls address translation, memory access permissions, memory attribute determination, and memory attribute checking.

R_{RJPRG} When the MMU cannot translate the IA, an MMU fault is generated.

R_{FLRHM} When an address translation stage generates an MMU fault, it is one of the following:

- When a stage 1 translation cannot translate an IA, a stage 1 MMU fault is generated.
- When a stage 2 translation cannot translate an IA, a stage 2 MMU fault is generated.

I_{ZWCKD} For more information, see:

- Implemented physical address size.
- Output address size configuration.
- Supported virtual address ranges.
- Input address size configuration.
- Intermediate physical address size configuration.
- Translation process.
- The effects of disabling an address translation stage.
- Memory aborts

D8.1.1 Translation granules

I_{GZDPY} The translation granule size determines the number of bits resolved by each lookup level when mapping from an IA to OA, and the maximum size of a single translation table.

R_{ZVQXW} The VMSA defines all of the following translation granule sizes:

- 4KB.
- 16KB.
- 64KB.

R_{JZCBC} For each translation granule size, and translation stage, support is IMPLEMENTATION DEFINED.

R_{VKNXT} Each address translation stage can be independently configured to use one of the supported translation granule sizes.

R_{QKCFY} A page is the smallest memory region in which an IA to OA mapping can be specified.

I_{MGPLT} The translation granule determines all of the following:

- The translation process used to resolve an IA to an OA.
- The page size of the address translation stage.
- The number of address bits required to address a page.
- The maximum translation table size of the address translation stage.
- The number of address bits that can be resolved in a single translation table lookup.

I_{KZLYC} For all of the following reasons, a larger translation granule can reduce the number of translation lookup levels:

- The larger granule uses a larger translation table with more entries.
- A single lookup can resolve more IA bits.
- The larger page size means more of the least significant address bits are used to address within a page and do not require translation because those bits are flat-mapped from IA to OA.

I_{QRLDB} Arm recommends separating memory-mapped peripherals by an integer multiple of the largest granule size supported by the PEs in the system, to allow independent management of each peripheral.

I_{XKKTX} If FEAT_GTG is not implemented, then the ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} fields specify the translation granules supported in both stage 1 and stage 2 translations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{FXFSZ} If FEAT_GTG is implemented, then support for translation granule sizes are determined by all of the following:

- The ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} fields specify the translation granules supported in stage 1 translations.
- The ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} fields specify the translation granules supported in stage 2 translations.

D8.1.2 Translation regimes

I_{CJMWY} A translation regime determines how a VA is mapped to a PA. The translation regime is affected by the current Security state, the current Exception level, the enabled Exception levels, HCR_EL2 settings, and implemented features.

I_{HLQTD} The architecture defines all of the following translation regimes:

- Non-secure EL1&0 translation regime.
- Secure EL1&0 translation regime.
- Realm EL1&0 translation regime.
- Non-secure EL2&0 translation regime.
- Secure EL2&0 translation regime.
- Realm EL2&0 translation regime.
- Non-secure EL2 translation regime.
- Secure EL2 translation regime.
- Realm EL2 translation regime.
- EL3 translation regime.

I_{DQHFH} The implemented Exception levels affect the supported translation regimes and translation stages.

R_{FVYPX} For each Security state, configuration of stage 1 and stage 2 translation can produce output addresses only in physical address spaces marked as YES in the following table:

Table D8-1 Relationship between Security state and permitted PA space

PA space	Non-secure state	Secure state	Realm state	Root state
Non-secure	Yes	Yes	Yes	Yes
Secure	No	Yes	No	Yes
Realm	No	No	Yes	Yes
Root	No	No	No	Yes

R_{TSFYG} Only the EL1&0 translation regimes support a stage 2 translation.

I_{DTPTJ} If a software agent, such as an operating system, uses or configures stage 1 translations, then that software agent might be unaware of all of the following:

- The stage 2 translation.
- The distinction between IPA and PA.

R_{ZQYNV} If the Effective value of HCR_EL2.E2H is 0, then only the EL1&0 translation regime, stage 1, can support two VA ranges.

R_{QLSFG} If the Effective value of HCR_EL2.E2H is 1, then the EL2&0 translation regime, stage 1, can support two VA ranges.

I_{QCFSC} If a stage 1 translation supports two VA ranges, then that translation stage also supports two privilege levels.

I_{MZRPG} Support for execution in Realm state at EL0 in AArch32 is IMPLEMENTATION DEFINED. Use of the Realm translation regimes at EL0 in AArch32 depends on that support for AArch32 at EL0. Support for execution in Realm state at other Exception levels is available in AArch64 only.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1.2.1 Non-secure EL1&0 translation regime

R_{QJSCR} If all of the following apply, then the Non-secure EL1&0 translation regime is used to translate addresses:

- The memory accesses occur in Non-secure state.
- One of the following applies:
  - Memory is accessed from EL1.
  - Memory is accessed from EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

R_{LLNVY} If Non-secure EL2 is not implemented, then the Non-secure EL1&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports two VA ranges and the use of ASIDs.

R_{PMRVM} If Non-secure EL2 is implemented, then the Non-secure EL1&0 translation regime supports all of the following:

- Translation stage 1 that does all the following:
  - Maps VAs to IPAs.
  - Supports two VA ranges and the use of ASIDs.
- Translation stage 2 that does all the following:
  - Maps stage 1 IPAs to PAs.
  - Supports a single IPA range.
- The translations are associated with a VMID.
- Translation stage 1 and translation stage 2 can be enabled independent of each other.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

I_{FYLSZ} Non-secure EL2 is effectively disabled if it is not implemented.

D8.1.2.2 Secure EL1&0 translation regime

R_{DKYXN} If all of the following apply, then the Secure EL1&0 translation regime is used to translate addresses:

- The memory accesses occur in Secure state.
- One of the following applies:
  - Memory is accessed from EL1.
  - Memory is accessed from EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

R_{YTZXB} If Secure EL2 is disabled or not implemented, then the Secure EL1&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports two VA ranges and the use of ASIDs.

R_{JMRWC} If Secure EL2 is enabled, then the Secure EL1&0 translation regime, supports all of the following:

- Translation stage 1 that does all the following:
  - Maps VAs to IPAs.
  - Supports two VA ranges and the use of ASIDs.
- Translation stage 2 that does all the following:
  - Maps stage 1 IPAs to PAs.
  - Supports a single Non-secure IPA range and a single Secure IPA range.
- The translations are associated with a VMID.
- Translation stage 1 and translation stage 2 can be enabled independent of each other.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

D8.1.2.3 Realm EL1&0 translation regime

R_{BTYVG} If all of the following apply, then the Realm EL1&0 translation regime is used to translate addresses:

- The memory accesses occur in Realm state.
- One of the following applies:
  - Memory is accessed from EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Memory is accessed from EL0 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

R_{YXQTB} The Realm EL1&0 translation regime supports all of the following:

- Translation stage 1 that does all the following:
  - Maps VAs to Realm IPAs.
  - Supports two VA ranges and the use of ASIDs.
- Translation stage 2 that does all the following:
  - Supports one IPA range.
  - Maps stage 1 Realm IPAs to Realm PAs or Non-secure PAs.
  - The translation is associated with a VMID.
- The translations are associated with a VMID.
- Translation stage 1 and translation stage 2 can be enabled independent of each other.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

R_{HMWYX} All features supported for the Non-secure EL1&0 stage 1 translation regime are supported for the Realm EL1&0 stage 1 translation regime.

R_{PDRZK} All features supported for the Non-secure EL1&0 stage 2 translations are supported for Realm EL1&0 stage 2 translations.

D8.1.2.4 Non-secure EL2&0 translation regime

R_{KBSNT} If all of the following apply, then the Non-secure EL2&0 translation regime is used to translate addresses:

- The memory accesses occur in Non-secure state.
- One of the following applies:
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 0} and memory accesses are from EL2.
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and memory accesses are from EL2 or EL0.

For more information, see Virtualization Host Extensions.

R_{MNQBD} The Non-secure EL2&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports two VA ranges and the use of ASIDs.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

I_{LKPNM} The Non-secure EL2&0 regime might also be used when execution is at EL1, and in some cases at EL0 when HCR_EL2.{E2H, TGE} is not {1, 1}. For example:

- At EL1 when FEAT_NV2 is in use, as described in Loads and stores generated by transforming register accesses.
- At EL1 or EL0 when the Statistical Profiling Unit is enabled, as described in The owning translation regime.
- At EL1 or EL0 when the Trace Buffer Extension is enabled, as described in The owning translation regime.

D8.1.2.5 Secure EL2&0 translation regime

R_{RJHRJ} If all of the following apply, then the Secure EL2&0 translation regime is used to translate addresses:

- The Effective value of SCR_EL3.EEL2 is 1.
- The memory accesses occur in Secure state.
- One of the following applies:
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 0} and memory accesses are from EL2.
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and memory accesses are from EL2 or EL0.

For more information, see Virtualization Host Extensions.

R_{RRQTL} The Secure EL2&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to Secure PAs or Non-secure PAs.
- Supports two VA ranges and the use of ASIDs.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

I_{LLMSY} The Secure EL2&0 regime might also be used when execution is at EL1, and in some cases at EL0 when HCR_EL2.{E2H, TGE} is not {1, 1}. For example:

- At EL1 when FEAT_NV2 is in use, as described in Loads and stores generated by transforming register accesses.
- At EL1 or EL0 when the Statistical Profiling Unit is enabled, as described in The owning translation regime.
- At EL1 or EL0 when the Trace Buffer Extension is enabled, as described in The owning translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1.2.6 Realm EL2&0 translation regime

R_{JYGWW} If all of the following apply, then the Realm EL2&0 translation regime is used to translate addresses:

- The memory accesses occur in Realm state.
- One of the following applies:
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 0} and memory accesses are from EL2.
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and memory accesses are from EL2 or EL0.

For more information, see Virtualization Host Extensions.

R_{PMTLC} The Realm EL2&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to Realm PAs or Non-secure PAs.
- Supports two VA ranges and the use of ASIDs.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

D8.1.2.7 Non-secure EL2 translation regime

R_{LCPJJ} If all of the following apply, then the Non-secure EL2 translation regime is used to translate addresses:

- The memory accesses occur in Non-secure state.
- The Effective value of HCR_EL2.E2H is 0.
- Memory is accessed from EL2.

R_{PYJCB} The Non-secure EL2 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports a single VA range.

I_{RRDHX} The Non-secure EL2 regime might also be used when execution is at EL1, and in some cases at EL0 when HCR_EL2.{E2H, TGE} is not {1, 1}. For example:

- At EL1 when FEAT_NV2 is in use, as described in Loads and stores generated by transforming register accesses.
- At EL1 or EL0 when the Statistical Profiling Unit is enabled, as described in The owning translation regime.
- At EL1 or EL0 when the Trace Buffer Extension is enabled, as described in The owning translation regime.

D8.1.2.8 Secure EL2 translation regime

R_{GFVNF} If all of the following apply, then the Secure EL2 translation regime is used to translate addresses:

- The Effective value of SCR_EL3.EEL2 is 1.
- The memory accesses occur in Secure state.
- The Effective value of HCR_EL2.E2H is 0.
- Memory is accessed from EL2.

R_{CTSYJ} The Secure EL2 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to Secure PAs or Non-secure PAs.
- Supports a single VA range.

I_{ZGLNV} The Secure EL2 regime might also be used when execution is at EL1, and in some cases at EL0 when HCR_EL2.{E2H, TGE} is not {1, 1}. For example:

- At EL1 when FEAT_NV2 is in use, as described in Loads and stores generated by transforming register accesses.
- At EL1 or EL0 when the Statistical Profiling Unit is enabled, as described in The owning translation regime.
- At EL1 or EL0 when the Trace Buffer Extension is enabled, as described in The owning translation regime.

D8.1.2.9 Realm EL2 translation regime

R_{FMHTT} If all of the following apply, then the Realm EL2 translation regime is used to translate addresses:

- The memory accesses occur in Realm state.
- The Effective value of HCR_EL2.E2H is 0.
- Memory is accessed from EL2.

R_{WYXPT} The Realm EL2 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to Realm PAs or Non-secure PAs.
- Supports a single VA range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1.2.10 EL3 translation regime

R_{PCRZX} If memory is accessed from EL3, then the EL3 translation regime is used to translate addresses.

R_{DCCTQ} The EL3 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs in all supported PA spaces.
- Supports a single VA range.

D8.1.3 Relationship between translation regimes and implemented Exception levels

I_{VNQXD} The set of translation regimes that an implementation supports depend on the set of Exception levels that the implementation supports.

R_{GFKSR} If an implementation does not include EL2, then all of the following apply:

- If the implementation does not include EL3, then the MMU provides a single EL1&0 stage 1 translation regime with an IMPLEMENTATION DEFINED Security state.
- If the implementation includes EL3, then the MMU provides an EL1&0 stage 1 translation regime in each Security state.

R_{RJWCS} For each translation stage supported by a translation regime, the following table shows what is required to support that translation stage.

Table D8-2 Translation regimes, translation stages, and associated controls

Translation stage	Requires
Non-secure EL2 stage 1	EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.{NSE, NS} is {0, 1}. The Effective value of HCR_EL2.E2H is 0.
Non-secure EL2&0 stage 1	EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.{NSE, NS} is {0, 1}. The Effective value of HCR_EL2.E2H is 1.
Non-secure EL1&0 stage 2	EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.{NSE, NS} is {0, 1}.
Non-secure EL1&0 stage 1	EL1 uses AArch64. The Effective value of SCR_EL3.{NSE, NS} is {0, 1}.
Secure EL2 stage 1	EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.EEL2 is 1. The Effective value of SCR_EL3.{NSE, NS} is {0, 0}. The Effective value of HCR_EL2.E2H is 0.
Secure EL2&0 stage 1	EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.EEL2 is 1. The Effective value of SCR_EL3.{NSE, NS} is {0, 0}. The Effective value of HCR_EL2.E2H is 1.
Secure EL1&0 stage 2	EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.EEL2 is 1. The Effective value of SCR_EL3.{NSE, NS} is {0, 0}.
Secure EL1&0 stage 1	EL1 uses AArch64. The Effective value of SCR_EL3.{NSE, NS} is {0, 0}.
Realm EL2 stage 1	FEAT_RME is implemented. SCR_EL3.{NSE, NS} is {1, 1}. The Effective value of HCR_EL2.E2H is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Translation stage	Requires
Realm EL2&0 stage 1	FEAT_RME is implemented. SCR_EL3.{NSE, NS} is {1, 1}. The Effective value of HCR_EL2.E2H is 1.
Realm EL1&0 stage 2	FEAT_RME is implemented. SCR_EL3.{NSE, NS} is {1, 1}.
Realm EL1&0 stage 1	FEAT_RME is implemented. SCR_EL3.{NSE, NS} is {1, 1}.
EL3 stage 1	EL3 is implemented and EL3 uses AArch64.

R_{VDVPR} For a stage 1 translation, if EL0 uses AArch32 state and one or more of the following applies, then VMSAv8-64 translation is required:

- EL1 uses AArch64 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- EL2 uses AArch64 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

R_{WNBMG} For a stage 2 translation, if either EL1 or EL0 uses AArch32 state and EL2 uses AArch64, then VMSAv8-64 translation is required.

R_{KXSWD} If EL0 uses AArch32, then accesses from EL0 have all of the following properties:

- Accesses use the EL1&0 or EL2&0 translation regime.
- Accesses use the AArch32 application-level memory model.
- Accesses are limited to a 32-bit VA range.

I_{MQXXQ} For more information, see The AArch64 Application Level Programmers' Model.

D8.1.4 The system registers relevant to MMU operation

I_{RJZKJ} If a System register name has a numeric suffix, then the suffix indicates the lowest Exception level that can access the register.

I_{TNCZR} Translation stages are enabled and controlled by the registers specified in the following table.

Table D8-3 Registers that enable and control translation stages

Translation Stage	Controlled from	Controlling registers
Non-secure EL2 stage 1	Non-secure EL2	SCTLR_EL2, SCTLR2_EL2, TCR_EL2, TCR2_EL2, MAIR_EL2, MAIR2_EL2, AMAIR_EL2, AMAIR2_EL2, TTBR0_EL2
Non-secure EL2&0 stage 1	Non-secure EL2	SCTLR_EL2, SCTLR2_EL2, TCR_EL2, TCR2_EL2, MAIR_EL2, MAIR2_EL2, AMAIR_EL2, AMAIR2_EL2, TTBR0_EL2, TTBR1_EL2, HCR_EL2
Non-secure EL1&0 stage 2	Non-secure EL2	SCTLR_EL2, SCTLR2_EL2, VTCR_EL2, VTTBR_EL2, HCR_EL2
Non-secure EL1&0 stage 1	Non-secure EL1	SCTLR_EL1, SCTLR2_EL1, TCR_EL1, TCR2_EL1, MAIR_EL1, MAIR2_EL1, AMAIR_EL1, AMAIR2_EL1, TTBR0_EL1, TTBR1_EL1, HCR_EL2
Secure EL2 stage 1	Secure EL2	SCTLR_EL2, SCTLR2_EL2, TCR_EL2, TCR2_EL2, MAIR_EL2, MAIR2_EL2, AMAIR_EL2, AMAIR2_EL2, TTBR0_EL2, SCR_EL3
Secure EL2&0 stage 1	Secure EL2	SCTLR_EL2, SCTLR2_EL2, TCR_EL2, TCR2_EL2, MAIR_EL2, MAIR2_EL2, AMAIR_EL2, AMAIR2_EL2, TTBR0_EL2, TTBR1_EL2, HCR_EL2, SCR_EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Translation Stage	Controlled from	Controlling registers
Secure EL1&0 stage 2	Secure EL2	SCTLR_EL2, SCTLR2_EL2, VSTCR_EL2, VSTTBR_EL2, VTCR_EL2, VTTBR_EL2, HCR_EL2, SCR_EL3
Secure EL1&0 stage 1	Secure EL1	SCTLR_EL1, SCTLR2_EL1, TCR_EL1, TCR2_EL1, MAIR_EL1, MAIR2_EL1, AMAIR_EL1, AMAIR2_EL1, TTBR0_EL1, TTBR1_EL1, HCR_EL2, SCR_EL3
Realm EL2 stage 1	Realm EL2	SCTLR_EL2, SCTLR2_EL2, TCR_EL2, TCR2_EL2, MAIR_EL2, MAIR2_EL2, AMAIR_EL2, AMAIR2_EL2, TTBR0_EL2
Realm EL2&0 stage 1	Realm EL2	SCTLR_EL2, SCTLR2_EL2, TCR_EL2, TCR2_EL2, MAIR_EL2, MAIR2_EL2, AMAIR_EL2, AMAIR2_EL2, TTBR0_EL2, TTBR1_EL2, HCR_EL2
Realm EL1&0 stage 2	Realm EL2	SCTLR_EL2, SCTLR2_EL2, VTCR_EL2, VTTBR_EL2, HCR_EL2
Realm EL1&0 stage 1	Realm EL1	SCTLR_EL1, SCTLR2_EL1, TCR_EL1, TCR2_EL1, MAIR_EL1, MAIR2_EL1, AMAIR_EL1, AMAIR2_EL1, TTBR0_EL1, TTBR1_EL1, HCR_EL2
EL3 stage 1	EL3	SCTLR_EL3, SCTLR2_EL3, SCR_EL3, TCR_EL3, MAIR_EL3, MAIR2_EL3, AMAIR_EL3, AMAIR2_EL3, TTBR0_EL3

I_{DDTVD} The following table lists the System register common abbreviations used in VMSAv8-64. The common abbreviation is used to describe features that apply to multiple translation regimes or translation stages.

Table D8-4 Common System register abbreviations used in VMSAv8-64

Common Abbreviation	Translation stage	Exception level EL1	EL2	EL3
AMAIR_ELx	-	AMAIR_EL1	AMAIR_EL2	AMAIR_EL3
AMAIR2_ELx	-	AMAIR2_EL1	AMAIR2_EL2	AMAIR2_EL3
MAIR_ELx	-	MAIR_EL1	MAIR_EL2	MAIR_EL3
MAIR2_ELx	-	MAIR2_EL1	MAIR2_EL2	MAIR2_EL3
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
SCTLR2_ELx	-	SCTLR2_EL1	SCTLR2_EL2	SCTLR2_EL3
TCR_ELx	Stage 1	TCR_EL1	TCR_EL2	TCR_EL3
	Stage 2		VTCR_EL2, VSTCR_EL2	
TCR2_ELx	Stage 1	TCR2_EL1	TCR2_EL2	
TTBR_ELx	Stage 1	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2, TTBR1_EL2	TTBR0_EL3
	Stage 2		VTTBR_EL2, VSTTBR_EL2	
TTBRn_ELx	Stage 1	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2, TTBR1_EL2	TTBR0_EL3
TTBR0_ELx	Stage 1	TTBR0_EL1	TTBR0_EL2	TTBR0_EL3
TTBR1_ELx	Stage 1	TTBR1_EL1	TTBR1_EL2	

I_{RGGJW} When software that enables or disables an address translation stage uses a PA that differs from the VA, speculative instruction fetching can cause complications. If software controls address translations that apply to the software itself, then when that software enables or disables an address translation stage, Arm recommends that the software uses a PA that is identical to the VA.

I_{CVKWB} For more information, see:

- Relationship between translation regimes and implemented Exception levels.
- Register name disambiguation by Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1.5 Out-of-context translation regimes

I_{PTDBT} Software is required to consider the effects on registers and translation tables when a context switch is made from one translation regime to another.

D8.1.5.1 Speculative memory accesses from out-of-context translation regimes

R_{NRJPP} If an implementation is executing at EL3 or EL2, the PE is not permitted to use the registers associated with the EL1&0 translation regime to speculatively access memory or translation tables.

R_{FCZXS} If an implementation is executing at EL3, the PE is not permitted to use the registers associated with the EL2 and EL2&0 translation regimes to speculatively access memory or translation tables.

R_{LFHQG} When an exception to a higher Exception level occurs, memory accesses caused by the translation table walk for the lower exception level can continue speculatively after the higher Exception level is entered.

I_{QVPYJ} When an exception to a higher Exception level occurs, executing a DSB instruction after entering that higher Exception level ensures that the memory accesses caused by the translation table walk are completed.

I_{GFKHK} If the *Statistical Profiling Unit* (SPU) is not in use at a lower Exception level, when a DSB instruction is executed and completed after entering a higher Exception level, all of the following apply regarding memory accesses caused by speculative translation table walks from the lower Exception level:

- All of the speculative memory accesses are observed as required by the Shareability and Cacheability attributes of the lower Exception level translation table entries.
- If it is impossible to determine that the speculative memory accesses have been observed by the observers, then these speculative memory accesses can still occur.
- No new memory accesses using the lower Exception level translation table entries occur.

I_{JZRDG} If the SPU is in use at a lower Exception level, when a PSB instruction and a subsequent DSB instruction are executed after entering a higher Exception level, all of the following apply regarding memory accesses caused by speculative translation table walks from a lower Exception level:

- All of the speculative memory accesses are observed as required by the Shareability and Cacheability attributes of the lower Exception level translation table entries.
- If it is impossible to determine that the speculative memory accesses have been observed by the observers, then these speculative memory accesses can still occur.
- No new memory accesses using the lower Exception level translation table entries occur.

I_{MTHPL} For exceptions to the rules in this section when the Trace Buffer Unit is enabled, see R_{SJFRQ}.

D8.1.5.2 Register changes when changing virtual machine

I_{XHBRX} If EL2 is enabled, when a switch from one virtual machine to another occurs, software executing at EL0 or EL1 expects register fields controlling or affecting the EL1&0 regime are modified consistently before the next virtual machine starts execution.

I_{SFZVV} If the registers fields controlling or affecting the EL1&0 translation regime are modified when changing a virtual machine, then all of the following apply:

- Software executing at EL2 does the modification.
- The registers are modified out-of-context from software executing at EL1 or EL0.
- When modifying the registers, no synchronization precautions are required until the exception return to EL1 or EL0.

D8.1.6 Implemented physical address size

I_{TRRHX} The implemented PA size is indicated by the value of the ID_AA64MMFR0_EL1.PARange field, as shown in the following table. All other values of the ID_AA64MMFR0_EL1.PARange field are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-5 Implemented physical address size

ID_AA64MMFR0_EL1.PARange	PA memory size	PA address size
0b0000	4GB	32 bits, PA[31:0]
0b0001	64GB	36 bits, PA[35:0]
0b0010	1TB	40 bits, PA[39:0]
0b0011	4TB	42 bits, PA[41:0]
0b0100	16TB	44 bits, PA[43:0]
0b0101	256TB	48 bits, PA[47:0]
0b0110	4PB	52 bits, PA[51:0]
0b0111	64PB	56 bits, PA[55:0]

R_{HGNFW} If the implemented PA size is greater than 48 bits, then FEAT_LPA is required.

I_{NRJPM} A PA size greater than 52 bits can only be expressed by the VMSAv9-128 translation system.

R_{ZHZNT} If the VMSAv8-64 translation system is used in an implementation that supports the VMSAv9-128 translation system, then PA bits [55:52] are set to 0b0000.

D8.1.7 Output address size configuration

R_{QQQSJ} For a translation stage that uses the VMSAv8-64 translation system, the maximum OA size that can be described by translation table entries is one of the following:

- For a translation regime controlled by TCR_ELx, if the 4KB or 16KB translation granule is used, then one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then 48 bits.
  - If the Effective value of TCR_ELx.DS is 1, then 52 bits.
- For any translation regime, if the 64KB translation granule is used, then one of the following:
  - If FEAT_LPA is not implemented, then 48 bits.
  - If FEAT_LPA is implemented, then 52 bits.

R_{LWWHG} For a translation stage that uses the VMSAv9-128 translation system, the maximum OA size that can be described by translation table entries is 56 bits.

R_{MRYSR} The name of the TCR_ELx.{I}PS field is one of the following:

- If the register is TCR_EL1, then the field is TCR_EL1.IPS.
- If the register is TCR_EL2, then one of the following:
  - If the Effective value of HCR_EL2.E2H is 0, then the field is TCR_EL2.PS.
  - If the Effective value of HCR_EL2.E2H is 1, then the field is TCR_EL2.IPS.
- If the register is TCR_EL3, then the field is TCR_EL3.PS.

I_{YDDNK} The OA size from a translation stage is configured by the associated TCR_ELx.{I}PS field, as shown in the following table.

Table D8-6 Output address size configuration

TCR_ELx.{I}PS	OA memory size	OA address size
0b000	4GB	32 bits, OA[31:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCR_ELx.{I}PS	OA memory size	OA address size
0b001	64GB	36 bits, OA[35:0]
0b010	1TB	40 bits, OA[39:0]
0b011	4TB	42 bits, OA[41:0]
0b100	16TB	44 bits, OA[43:0]
0b101	256TB	48 bits, OA[47:0]
0b110	4PB	52 bits, OA[51:0]
0b111	64PB	56 bits, OA[55:0]

Note

For the TCR_EL2.PS field, if the Effective value of HCR_EL2.E2H is 0, then the value 0b111 is reserved.

I_{MNZFV} The TCR_ELx.{I}PS field is three bits and has the same encoding as the three least significant bits of ID_AA64MMFR0_EL1.PARange.

R_{CLPQT} The OA size from a translation stage cannot be larger than the implemented PA size. If the OA size represented by TCR_ELx.{I}PS is larger than the implemented PA size, then the OA size is treated as being the same as the implemented PA size.

I_{PKBTM} Arm strongly recommends that software avoids configuring TCR_ELx.{I}PS to a value greater than the implemented PA size.

I_{HGDWP} If a translation stage is enabled and the translation stage OA is larger than the implemented PA size, then an Address size fault is generated at the lookup level in the translation stage that generated the OA.

R_{BZHGM} If all of the following apply, then a stage 2 Translation fault is generated:

- Two address translation stages are used.
- Stage 2 address translation is enabled.
- The stage 1 OA does not generate a stage 1 Address size fault.
- The stage 1 OA is larger than the specified stage 2 translation IA size.

D8.1.8 Supported virtual address ranges

I_{YMKBT} The VA size is determined by configuring the stage 1 translation IA size.
For more information, see Input address size configuration.

R_{HYPNC} For the VMSAv8-64 translation system, the maximum VA size is one of the following:

- For a translation regime controlled by TCR_ELx, if the 4KB or 16KB translation granule is used, then one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then 48 bits.
  - If the Effective value of TCR_ELx.DS is 1, then 52 bits.
- For any translation regime, if the 64KB translation granule is used, then one of the following:
  - If FEAT_LVA is not implemented, then 48 bits.
  - If FEAT_LVA is implemented, then 52 bits.

R_{XQGZR} For the VMSAv9-128 translation system, the maximum VA size is one of the following:

- If FEAT_LVA3 is implemented, then 56 bits.
- If FEAT_LVA is implemented and FEAT_LVA3 is not implemented, then 52 bits.
- If FEAT_LVA is not implemented, then 48 bits.

I_{KRQBC} Only stage 1 address translation in the EL1&0 and EL2&0 translation regimes support two VA ranges.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{BXWQW} If a translation regime supports two VA ranges, then each VA range represents an independent mapping from IA to OA.

R_{GZPNP} If a translation regime supports a single VA range, then all of the following apply:

- The maximum IA width is 56 bits.
- If the maximum VA size is 48 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.
- If the maximum VA size is 52 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x000F_FFFF_FFFF_FFFF`.
- If the maximum VA size is 56 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x00FF_FFFF_FFFF_FFFF`.

R_{HYYQH} If a translation regime supports two VA ranges, then all of the following apply:

- The maximum IA width is 55 bits.
- IA bit[55] determines one of the following VA ranges:
  - If IA bit[55] is 0, the lower VA range is used.
  - If IA bit[55] is 1, the upper VA range is used.
- The maximum lower VA range is one of the following:
  - If the maximum VA size is 48 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.
  - If the maximum VA size is 52 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x000F_FFFF_FFFF_FFFF`.
  - If the maximum VA size is 55 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x007F_FFFF_FFFF_FFFF`.
- The maximum upper VA range is one of the following:
  - If the maximum VA size is 48 bits, then the maximum VA range is `0xFFFF_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.
  - If the maximum VA size is 52 bits, then the maximum VA range is `0xFFF0_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.
  - If the maximum VA size is 55 bits, then the maximum VA range is `0xFF80_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.

R_{CSSJN} If the configured size of the upper VA range is less than the maximum, then all of the following apply:

- The range starting address is greater than the address used by the maximum VA size.
- The range ending address is `0xFFFF_FFFF_FFFF_FFFF`.

I_{THWTD} A 48-bit VA range defines an address space of 256TB. A 52-bit VA range defines an address space of 4PB. A 56-bit VA range defines an address space of 64PB.

D8.1.9 Input address size configuration

R_{TLMML} If a translation stage supports a single VA range, then TCR_ELx.T0SZ configures the IA size, translated using TTBR0_ELx.

R_{GMLRF} If a translation stage supports two VA ranges, then all of the following TCR_ELx.T*n*SZ fields specify the IA size:

- TCR_ELx.T0SZ configures the IA size of the lower VA range, translated using TTBR0_ELx.
- TCR_ELx.T1SZ configures the IA size of the upper VA range, translated using TTBR1_ELx.

I_{GRZSP} The stage 1 translation IA size is defined as 64-TCR_ELx.T*n*SZ.

R_{PLCGL} For all translation stages, the following table shows how the maximum permitted T*n*SZ value is determined:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-7 Determining the maximum permitted TnSZ value

Translation granule size	FEAT_TTST not implemented	FEAT_TTST implemented
4KB	39	48
16KB	39	48
64KB	39	47

I_{FTBXR} If TCR_ELx.T*n*SZ configures an IA size that is smaller than the maximum size, then each one-bit reduction in the IA size has one of the following effects on the lookup level that the table walk starts with:

- The translation table size is reduced by one half.
- The table walk starts one level later.

R_{YXNYW} If the value of T*n*SZ is greater than the maximum permitted value, then any use of the T*n*SZ value causes one of the following IMPLEMENTATION DEFINED behaviors to occur:

- For all purposes other than reading back the value, the implementation behaves as if the T*n*SZ field has the maximum permitted value.
- Any use generates a level 0 Translation fault.

R_{GTJBY} For a stage 1 translation using the VMSAv8-64 translation system, the minimum Effective value of T*n*SZ is one of the following:

- For the 4KB or 16KB translation granule, one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then the minimum Effective value of T*n*SZ is 16.
  - If the Effective value of TCR_ELx.DS is 1, then the minimum Effective value of T*n*SZ is 12.
- For the 64KB translation granule, one of the following:
  - If FEAT_LVA is not implemented, then the minimum Effective value of T*n*SZ is 16.
  - If FEAT_LVA is implemented, then the minimum Effective value of T*n*SZ is 12.

R_{VRKKV} For a stage 1 translation using the VMSAv9-128 translation system, the minimum Effective value of T*n*SZ is one of the following:

- If FEAT_LVA3 is implemented, then one of the following:
  - If the translation regime supports a single IA range, then 8.
  - If the translation regime supports two IA ranges, then 9.
- If FEAT_LVA is implemented and FEAT_LVA3 is not implemented, then 12.
- If FEAT_LVA is not implemented, then 16.

R_{VCDSW} If all of the following apply, then a stage 1 level 0 Address size fault is generated:

- Stage 1 translation is disabled.
- The IA size is larger than the implemented PA size.

I_{QZVXG} For an address translation stage, when an attempt is made to translate an address larger than the configured IA size, a level 0 Translation fault at that translation stage is generated.

R_{SXWGM} For a stage 1 address translation, if the value of T*n*SZ is smaller than the minimum Effective value, then when using the T*n*SZ value to translate an address, one of the following applies:

- If FEAT_LVA is not implemented, then it is IMPLEMENTATION DEFINED which of the following behaviors occurs:
  - For all uses other than reading back the value, the implementation behaves as if the T*n*SZ field has a value of 16.
  - When using the T*n*SZ value to translate an address, a stage 1 level 0 Translation fault is generated.
- If FEAT_LVA is implemented, then when using the T*n*SZ value to translate an address, a stage 1 level 0 Translation fault is generated.

I_{TFBDN} For more information, see Relationship between translation regimes and implemented Exception levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1.10 Intermediate physical address size configuration

R_{SCJMS} When a stage 2 translation occurs, the configured IPA size is specified by one of the following T0SZ values:

- If the IPA is in Non-secure or Realm address space, then VTCR_EL2.T0SZ
- If the IPA is in Secure address space, then VSTCR_EL2.T0SZ

R_{TDJSG} The implemented PA size constrains all of the following:

- The maximum IPA size.
- The effective minimum T0SZ value, VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ, used to specify the configured IPA size.
- The stage 2 initial lookup level.

R_{DRRZZ} When all of the following apply, the stage 2 translation IPA size check generates a stage 2 level 0 Translation fault:

- Stage 2 translation is enabled.
- The input address to stage 2 translation, whether or not stage 1 translation is enabled, is larger than the IA size configured in VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ, as appropriate for the IPA space.

R_{DTLMN} When a stage 2 translation occurs, the effective minimum value of T0SZ is determined by the implemented PA size as shown in the following table:

Table D8-8 Determining the effective minimum T0SZ value for stage 2 translations

Implemented PA size	Effective minimum T0SZ value
32 bits	32 if EL1 is using AArch64
32 bits	24 if EL1 is using AArch32
36 bits	28 if EL1 is using AArch64
36 bits	24 if EL1 is using AArch32
40 bits	24
42 bits	22
44 bits	20
48 bits	16
52 bits	12
56 bits	8 if the translation regime uses the VMSAv9-128 translation system

R_{SRKBC} Table D8-9 shows the permitted initial stage 2 lookup levels for each of the effective minimum T0SZ values. For the 4KB granule, an initial lookup level of 3 is only supported if FEAT_TTST is implemented and the PE is executing in AArch64 state.

Table D8-9 Implications of the effective minimum T0SZ value on the initial stage 2 lookup level

Supported PA size	Effective minimum T0SZ value	Valid initial lookup levels		
		4KB granule	16KB granule	64KB granule
32 bits	32, if EL1 is using AArch64 24, if EL1 is using AArch32	3, 2, 1	3, 2	3, 2
36 bits	28, if EL1 is using AArch64 24, if EL1 is using AArch32	3, 2, 1	3, 2	3, 2
40 bits	24	3, 2, 1	3, 2	3, 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Supported PA size	Effective minimum T0SZ value	Valid initial lookup levels		
		4KB granule	16KB granule	64KB granule
42 bits	22	3, 2, 1	3, 2, 1	3, 2
44 bits	20	3, 2, 1, 0	3, 2, 1	3, 2, 1
48 bits	16	3, 2, 1, 0	3, 2, 1	3, 2, 1
52 bits	12	3, 2, 1, 0, -1	3, 2, 1, 0	3, 2, 1
56 bits	8	3, 2, 1, 0, -1, -2	3, 2, 1, 0, -1	3, 2, 1, 0

For more information, see:

- VMSAv8-64 Stage 2 address translation using the 4KB translation granule
- VMSAv8-64 Stage 2 address translation using the 16KB translation granule
- VMSAv8-64 Stage 2 address translation using the 64KB translation granule
- Translation using the VMSAv9-128 translation system

R_{MWXYY} For a stage 2 address translation, if FEAT_LPA is not implemented and the value of T0SZ is smaller than the effective minimum value, then it is IMPLEMENTATION DEFINED which of the following behaviors occurs:

- For all purposes other than reading back the value of the field, the implementation treats the T0SZ field as having the effective minimum value.
- The implementation generates a stage 2 level 0 Translation fault.

R_{MYXHB} For a stage 2 address translation, if FEAT_LPA is implemented and the value of the T0SZ field is smaller than the effective minimum value, then a stage 2 level 0 Translation fault is generated.

I_{ZZJJF} If the T0SZ field is programmed to a value smaller than the effective minimum value, then a larger address range cannot be supported because an Address size fault is generated due to one of the following reasons:

- For a VMSAv8-64 stage 1 translation, the stage 1 OA is larger than the PA size.
- For a VMSAv8-32 stage 1 translation, the stage 1 OA is larger than 40 bits.

I_{NLYLM} For more information, see:

- Implemented physical address size.
- Translation table walk.
- Memory aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2 Translation process

I_{GDSWV} The translation of a VA to a PA begins with reading the translation table base address register, followed by a walk through the translation tables to read the descriptors associated with the mapping.

D8.2.1 Translation table walk

R_{BTTHB} A translation table walk is the set of translation table lookups that are required to do all of the following:

- For a single stage translation at stage 1, translate a VA to a PA.
- For two translation stages and stage 1 is disabled, translate an IPA to a PA.
- For a two stage translation, all of the following:
  - For a stage 1 translation, translate a VA to an IPA.
  - For a stage 2 translation, translate an IPA to a PA for each of the stage 1 translation table lookups.
  - For a stage 2 translation, translate an IPA to a PA for the stage 1 OA.

R_{RHQPX} When a translation table walk begins, the initial translation table lookup uses the translation table base address stored in the TTBR_ELx register to locate the translation table.

R_{YKQTS} When a translation table lookup occurs, the descriptor held in the translation table entry is read as one of the following:

- For the VMSAv8-64 translation system, an 8-byte single-copy atomic access.
- For the VMSAv9-128 translation system, a 16-byte single-copy atomic access.

For information on the translation table alignment requirements, see Translation table alignment.

R_{YPWQG} The descriptor type indicates one of the following:

- The translation table walk is complete and the translation table entry is the final entry.
- The translation table walk requires an extra lookup at the next higher lookup level.
- The descriptor is invalid.

R_{NMTJN} All of the following descriptor types conclude the translation process:

- Page descriptor.
- Block descriptor.
- Invalid descriptor.

For more information, see Translation table descriptor formats.

I_{WMLYG} When an Invalid descriptor is returned during the translation process, a Translation fault at the current lookup level is generated.

I_{QPHCD} When an extra lookup level is required, the descriptor contains all of the following information:

- The translation table base address of the next lookup level.
- For stage 1 descriptors, if the Effective value of TCR_ELx.HPD*n* is 0, then the descriptor has hierarchical access permissions that are applied to the final translation.
- If the translation is made in a Secure translation regime, then a stage 1 descriptor indicates whether the base address is in the Secure address space or the Non-secure address space, unless a hierarchical control at a previous lookup level has indicated that it is required to be in the Non-secure address space.

For more information, see Security state of translation table lookups.

R_{VSJGV} For a translation lookup level, the translation table base address is one of the following:

- For the initial lookup level, the aligned value of the appropriate TTBR_ELx.BADDR field.
- For subsequent lookup levels, the next-level translation table base address returned by the previous lookup level.

I_{HFBGP} For the VMSAv9-128 translation system, the Skip level (SKL) field in the descriptors and translation table base address registers permit the translation table walk to skip subsequent lookup levels.

I_{ZMMXH} When the last lookup level of a translation stage returns a valid descriptor, it contains all of the following:

- The OA from the translation stage.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The memory region access permissions.
- The memory region attributes.

For more information, see Memory region attributes and Memory access control.

RRZHGL For a translation regime that uses two translation stages, the stage 1 descriptor addresses require stage 2 translation from IPA to PA.

RNLLYJ When a translation table walk succeeds, all of the following are returned:

- The OA.
- If the translation table walk is made in a Secure translation regime, then the information returned indicates whether the OA is in the Secure IPA or PA space, or the Non-secure IPA or PA space.
- If the translation table walk is made in a Realm translation regime, then the information returned indicates whether the OA is in the Realm PA space, or the Non-secure PA space.
- If the translation table walk is made in Root state, then the OA can be in the Root, Realm, Secure, or Non-secure PA spaces.
- The output memory region attributes.
- The output memory region access permissions.

For more information, see Security state of translation table lookups.

IVPFHD The following figure is a generalized view of a single address translation stage with three lookup levels.

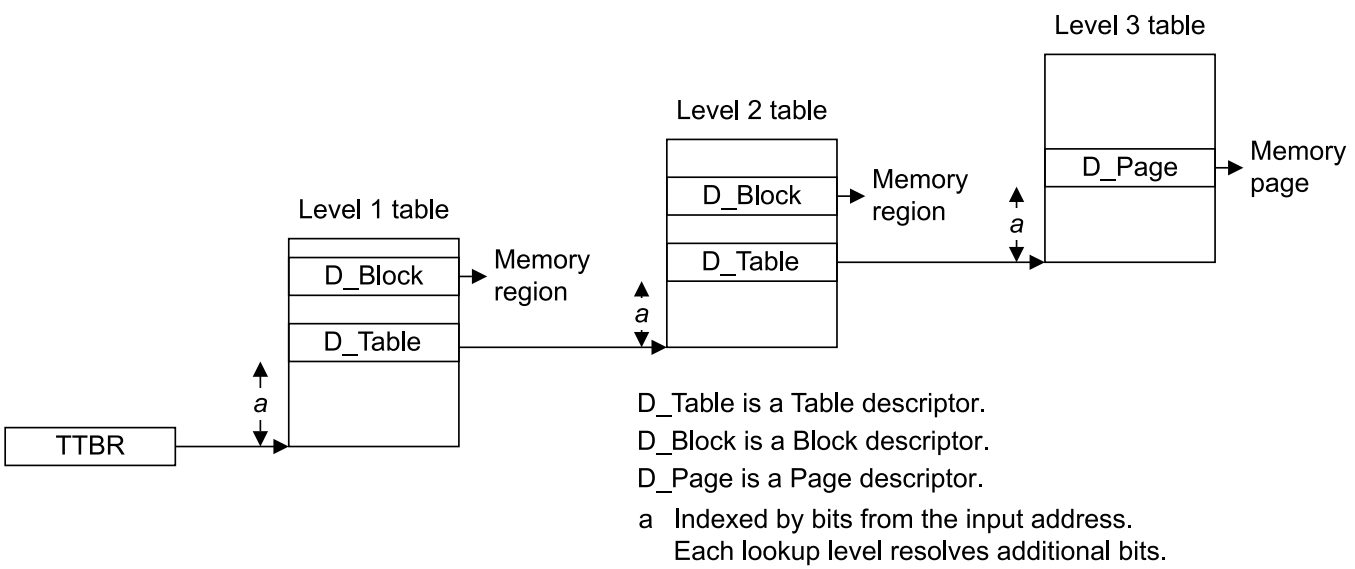

Figure D8-1 Generalized view of a single address translation stage

IJWQFM If all of the following are true, then a translation table entry is permitted to be cached in a Translation Lookaside Buffer (TLB):

- The entry is valid.
- Using the entry does not generate a Translation fault, an Address size fault, or an Access flag fault.

IZTRGT TLB caching can reduce the number of required translation table lookups.

IJQDTN If two address translation stages are used and when a full translation table walk at both stages is required, then all of the following apply:

- S1 is the number of lookup levels required for a stage 1 translation.
- S2 is the number of lookup levels required for a stage 2 translation.
- The maximum number of translation table lookups required is (S1+1)*(S2+1)-1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.2 Concatenated translation tables

R_{MCGQP} All statements in this section apply only to the VMSAv8-64 translation system.

I_{TDMHR} For the initial lookup of a stage 2 address translation, up to 16 translation tables can be concatenated.

R_{HRSBS} For a stage 2 translation, if the translation table in the initial lookup level would require 16 or fewer entries, then the stage 2 translation can be configured to have all of the following properties:

- The initial lookup of the stage 2 translation starts at the next lookup level.
- A number of translation tables corresponding to the original number of translation table entries at the previously initial lookup level are concatenated at that next lookup level.

R_{DXBSH} For the 16KB translation granule, if the Effective value of VTCR_EL2.DS is 0 and a 48-bit address size is translated by the stage 2 translation, then the initial lookup is required to start at level 1 with two concatenated translation tables.

R_{JKZFY} For the initial lookup in a stage 2 translation, if concatenated translation tables are used, then all of the following apply:

- Up to four additional IA bits are resolved by the initial lookup.
- For each additional *n* IA bits resolved by the initial lookup, the number of least significant translation table base address bits held in the TTBR_ELx is reduced by *n* bits.
- For each additional *n* IA bits resolved by the initial lookup, 2^n concatenated translation tables are required at the initial lookup level.

I_{DRXGV} Using a concatenated translation table eliminates the overhead of an extra lookup level.

I_{YXFPJ} If concatenated translation tables are used, then software is required to do all of the following:

- Align the base address of the first translation table to the sum of the size of the memory occupied by the concatenated translation tables.
- Program VTTBR_EL2 or VSTTBR_EL2 with the address of the first translation table in the set of concatenated translation tables.
- Program VTCR_EL2 or VSTCR_EL2 with the initial lookup level.

D8.2.3 Translation table base address register

I_{TLWRN} For a translation stage, the translation table base address register, TTBR_ELx, holds the translation table base address of the initial lookup.

R_{DXLKR} For a translation stage that supports two VA ranges, two translation table base address registers are required.

I_{HPFDM} Software can use TCR_ELx.TnSZ to configure the translation stage IA size to be smaller than the supported size.

R_{QKFFR} If EL2 is disabled or not implemented, then for the stage 1 EL1&0 regime, all of the following apply:

- The translation table base address held in the TTBR_ELx register is a PA.
- The translation table base address returned by a Table descriptor is a PA.

R_{QSWHS} If EL2 is enabled, then for all address translation stages other than stage 1 in EL1&0, all of the following apply:

- The translation table base address held in the TTBR_ELx register is a PA.
- The translation table base address returned by a Table descriptor is a PA.

R_{PNHHK} If EL2 is enabled, then for stage 1 address translations in EL1&0, all of the following apply:

- The stage 1 translation table base address held in the TTBR_ELx register is an IPA.
- The stage 1 translation table base address returned by a Table descriptor is an IPA.
- Accesses to stage 1 translation tables are subject to a stage 2 translation.

R_{GYQHZ} For an address translation stage, the translation table base address in TTBR_ELx is defined for the supported OA size of that stage.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{SXSYL} For the VMSAv8-64 translation system, the number of {I}PA bits held in TTBR_ELx is determined by the granule size and OA address size. The bits used for each granule size when using the maximum OA address size of 48 bits or 52 bits are shown in the following table. Software might be required to set one or more of the low-order base address bits to zero to align the table to the table size.

Table D8-10 {I}PA bits held in TTBR_ELx

Granule size	Maximum OA size	Translation table base address	{I}PA bits held in TTBR_ELx
4KB, 16KB, or 64KB	52 bits	TTBR_ELx[5:2]	{I}PA[51:48]
4KB	48 bits or 52 bits	TTBR_ELx[47:12]	{I}PA[47:12]
16KB	48 bits or 52 bits	TTBR_ELx[47:14]	{I}PA[47:14]
64KB	48 bits or 52 bits	TTBR_ELx[47:16]	{I}PA[47:16]

I_{QTPMQ} If an address translation stage uses an OA size smaller than the maximum, then the upper bits of the translation table base address in TTBR_ELx corresponding to the upper bits of OA size are required to be set by software to zero.

I_{LXTNH} If TCR_ELx.T*n*SZ specifies an IA size that is smaller than the maximum size resolved at the initial lookup level, then more low-order TTBR_ELx bits are used to hold the translation table base address.

R_{VPBBF} If an address translation stage uses an OA size smaller than the maximum and if the bits above the configured OA size of the translation table base address in TTBR_ELx are not set to zero, then an Address size fault is generated and reports all of the following:

- A translation level 0 lookup as generating the fault, regardless of whether or not the translation stage starts with a level 0 lookup.
- The translation stage that generated the fault.

R_{DYDPX} Direct writes to TTBR0_ELx and TTBR1_ELx occur in program order relative to one another, without the need for explicit synchronization. For any one translation, all indirect reads of TTBR0_ELx and TTBR1_ELx that are made as part of the translation observe only one point in that order of direct writes.

I_{RBMTM} Consistent with the general requirements for direct writes to System registers, direct writes to TTBR_ELx are not required to be observed by indirect reads until completion of a Context synchronization event.

I_{QBVNF}

Example D8-1 Changing translation table base address and ASID value when TCR_EL1.A1=0

The following is an example software sequence for changing translation table base address and ASID value when TCR_EL1.A1=0:

```
Change TTBR1 to point only at global entries
Change TTBR0 (includes changing the ASID)
Change TTBR1 to point at tables containing non-global entries
ISB
```

I_{GRBLY}

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D8-2 Changing translation table base address and ASID value when TCR_EL1.A1=1

The following is an example software sequence for changing translation table base address and ASID value when TCR_EL1.A1=1:

```
Change TTBR0 to point to no valid entries
Change TTBR1 (includes changing the ASID)
Change TTBR0 to point at tables containing valid entries
ISB
```

I_{BSFJP} For more information, see System registers relevant to MMU operation.

D8.2.4 Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported

R_{HCTPT} If a stage 1 translation regime supports two VA ranges, then the translation regime TTBR_ELx registers point to all of the following address ranges:

- For the lower VA range that begins at address `0x0000_0000_0000_0000`, TTBR0_ELx points to the translation table for the initial lookup level.
- For the upper VA range that ends at address `0xFFFF_FFFF_FFFF_FFFF`, TTBR1_ELx points to the translation table for the initial lookup level.

R_{YYVYV} If a stage 1 translation regime supports two VA ranges, then the TCR_ELx.{T0SZ, T1SZ} fields configure all of the following address range sizes:

- The lower VA range is `0x0000_0000_0000_0000` to $(2^{(64-T0SZ)} - 1)$.
- The upper VA range is $(2^{64} - 2^{(64-T1SZ)})$ to `0xFFFF_FFFF_FFFF_FFFF`.

I_{THCZK} If a stage 1 translation regime supports two VA ranges, when an accessed address is not in the lower VA range or the upper VA range, a stage 1 level 0 Translation fault is generated.

I_{BCMMD} The following figure illustrates the two address ranges translated by the tables the TTBR_ELx registers point to.

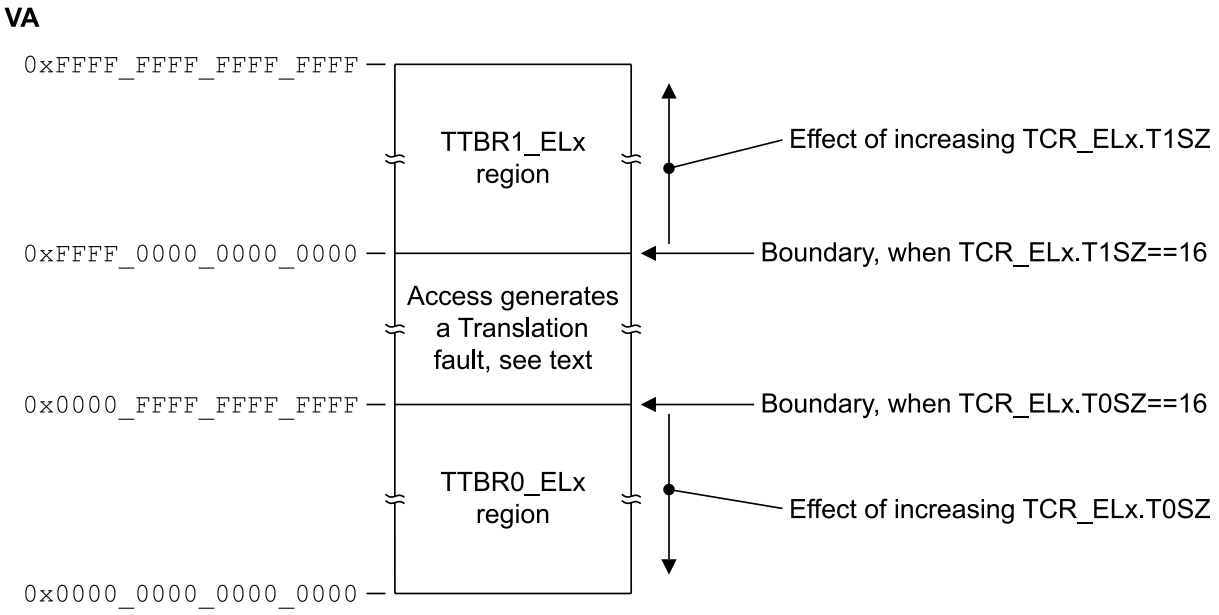

Figure D8-2 Example of a stage 1 translation that supports two VA ranges

R_{VZCSR} If a stage 1 translation regime supports two VA ranges, then all of the following are used to select the TTBR_ELx:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If VA bit[55] is zero, then TTBR0_ELx is selected.
- If VA bit[55] is one, then TTBR1_ELx is selected.

I_{ZFSYQ} If a stage 1 translation regime supports two VA ranges and TCR_ELx.EPD*n* is one, then when a TLB miss occurs based on TTBRn_ELx, a level 0 Translation fault is generated, and no translation table walk is done.

I_{SMKQB} For the VMSAv9-128 translation system, if a stage 1 translation regime supports two VA ranges, then the maximum supported VA width is 55 bits, and the smallest permitted value of the T*n*SZ field is 9.

I_{QBLKT} For more information, see Address tagging.

D8.2.4.1 Preventing EL0 access to halves of the address map

R_{BNDVG} All statements in this section require implementation of FEAT_E0PD.

I_{PZZHN} The TCR_ELx.{E0PD0, E0PD1} fields can be used to prevent EL0 access to the addresses translated by the corresponding TTBR0_ELx or TTBR1_ELx.

I_{RFRSQ} When the TCR_ELx.{E0PD0, E0PD1} fields prevent EL0 access to an address translated by TTBR0_ELx or TTBR1_ELx, then a level 0 Translation fault is generated.

R_{DDFWH} When the TCR_ELx.{E0PD0, E0PD1} fields generate a level 0 Translation fault, then all of the following apply:

- The time needed to take the fault should be the same whether or not the address accessed is present in a TLB, to mitigate attacks that use fault timing.
- The fault generated does not affect any micro-architectural state of the PE in a manner that is different if the address accessed is present in a TLB or not, to prevent this information being used to determine the presence of the address in a TLB.

R_{CZWGX} When the TCR_ELx.{E0PD0, E0PD1} fields generate a level 0 Translation fault, the fault is not counted as a TLB miss for performance monitoring features.

D8.2.5 Translation table and translation table lookup properties

I_{SLRTY} Translation table and translation table lookup properties include the table size, table alignment, table endianness, and memory attributes.

D8.2.5.1 Translation table size

R_{YDJDH} The descriptor size in a translation table entry is one of the following:

- For the VMSAv8-64 translation system, an eight-byte, or 64-bit, object.
- For the VMSAv9-128 translation system, a 16-byte, or 128-bit, object.

R_{WDRNX} If *n* is the number of bits resolved by a lookup level, then the number of translation table entries required at that lookup level is 2^n.

R_{NVTHS} The size of a translation table in bytes is determined by multiplying the number of entries by the descriptor size.

I_{ZMLQD} The maximum number of translation table entries is determined by the translation granule size, which is defined by one of the following:

- For a stage 1 translation that supports one VA range, TCR_ELx.TG0.
- For a stage 1 translation that can support two VA ranges, all of the following:
  - For the lower VA range, TCR_ELx.TG0.
  - For the upper VA range, TCR_ELx.TG1.
- For a stage 2 translation in the Non-secure EL1&0 translation regime, VTCR_EL2.TG0.
- For a stage 2 translation in the Secure EL1&0 translation regime, one of the following:
  - If the stage 2 IA is a Non-secure IPA, then VTCR_EL2.TG0.
  - If the stage 2 IA is a Secure IPA, then VSTCR_EL2.TG0.
- For a stage 2 translation in the Realm EL1&0 translation regime, VTCR_EL2.TG0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.5.2 Translation table alignment

R_{KBLCR} A translation table is required to be aligned to one of the following:

- For the VMSAv8-64 translation system, if the translation table has fewer than eight entries and an OA size greater than 48 bits is used, then the table is aligned to 64 bytes.
- Otherwise, the translation table is aligned to the size of that translation table.

I_{TBPFK} Only when all of the following are true is it possible to have fewer than 8 translation table entries:

- The translation table is at the initial lookup level.
- Concatenated translation tables are not used.

R_{VCLZN} If concatenated translation tables are used, then the concatenated translation tables are required to be aligned to the overall size of the memory occupied by the concatenated translation tables.

D8.2.5.3 Translation table lookup endianness

I_{LBMWQ} When a translation table lookup reads a translation table entry, all of the following apply:

- If the VMSAv8-64 translation system is used, the read is an 8-byte single-copy atomic access.
- If the VMSAv9-128 translation system is used, the read is a 16-byte single-copy atomic access.

R_{FDKMS} The endianness of a translation table lookup is determined by the appropriate SCTLR_ELx.EE bit.

R_{PZVGH} Changing an SCTLR_ELx.EE bit requires synchronization before the change is guaranteed to be visible to subsequent operations.

For more information, see Synchronization requirements for AArch64 System registers.

I_{QJDJV} Arm does not recommend changing an SCTLR_ELx.EE bit unless one or more of the following apply:

- The translation table lookups affected by the modified SCTLR_ELx.EE bit belong to an out-of-context translation regime.
- For any translation stage affected by the modified SCTLR_ELx.EE bit, address translation is disabled.

D8.2.5.4 Translation table lookup memory attributes

I_{TXCPH} For a translation table lookup in an address translation stage, the TCR_ELx and VTCR_EL2 registers determine the memory Cacheability and Shareability attributes that apply.

R_{ZZSYD} For a translation table lookup in an address translation stage, the required memory type is Normal memory.

I_{RMDMS} For a two stage translation regime, when a translation table lookup from stage 1 occurs, all of the following apply:

- Arm recommends that the stage 2 translation of the stage 1 translation table lookup does not map to Device memory.
- Software can configure HCR_EL2.PTW to protect stage 2 table walks from mapping stage 1 translation tables to Device memory.

For more information, see Stage 2 fault on a stage 1 translation table walk.

R_{XGXKR} When memory locations that hold the translation tables are accessed by software, and the memory accesses and the translation table walks have mismatched memory attributes, the behavior is the same as different memory accesses to the same memory location using mismatched memory attributes.

For more information, see Mismatched memory attributes.

I_{BSKKL} Arm recommends that the memory attributes applied by the TCR_ELx to the translation tables are the same as the attributes that are applied by other accesses to the memory that holds the translation tables.

I_{WVTXS} For more information on Normal and Device memory and on the Cacheability and Shareability attributes, see Memory types and attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.6 Translation table walk properties

D8.2.6.1 Ordering of memory accesses from translation table walks

R_{TGRMW} A translation table walk is considered to be a separate observer. An explicit write to the translation tables might be observed by that separate observer for either of the following:

- A translation table walk caused by a different explicit write generated by the same instruction.
- A translation table walk caused by an explicit access generated by any instruction appearing in program order after the instruction doing the explicit write to the translation table.

R_{SPVBD} The explicit write to the translation tables is guaranteed to be observable, to the extent required by the Shareability attributes, only after the execution of a DSB instruction. This DSB instruction and the instruction that performed the explicit write to the translation tables must have been executed by the same PE.

R_{RGPMV} Any writes to the translation tables are not observed by any translation table accesses associated with an explicit memory access generated by a load or store that occurs in program order before the instruction that performs the write to the translation tables.

R_{NNFPF} If FEAT_ETS2 is implemented, then an Effect E_1 is Ordered-before an Effect E_2 when all of the following apply:

- E_1 is not generated by an Advanced SIMD, floating-point, SVE, or SME instruction.
- E_2 is not generated for an instruction fetch.
- E_1 is an Explicit Memory Effect.
- E_1 appears in program order before E_3.
- E_3 is a TLBUncacheable Fault Effect.
- E_2 is Translation-intrinsically-before E_3.
- E_2 is an Implicit TTD Memory Read Effect.

I_{JBHNF} The full definition of the ordering implications of FEAT_ETS2 is covered formally in Ordering requirements defined by the formal concurrency model, and the rules in this section are not intended to contradict that section.
For more information about ordering requirements to FEAT_ETS2, see DSB-ordered-before and ETS-ordered-before.

R_{QVWHP} If FEAT_ETS3 is implemented, then an Effect E_1 is Ordered-before an Effect E_2 when all of the following apply:

- E_1 is not generated by an Advanced SIMD, floating-point, SVE, or SME instruction.
- E_2 is not generated for an instruction fetch.
- E_1 is an Explicit Memory Effect.
- E_1 appears in program order before E_3.
- E_3 is an MMU Fault Effect.
- E_2 is Translation-intrinsically-before E_3.
- E_2 is an Implicit TTD Memory Read Effect.

I_{VKSMJ} The full definition of the ordering implications of FEAT_ETS3 is covered formally in Ordering requirements defined by the formal concurrency model, and the rules in this section are not intended to contradict that section.
For more information about ordering requirements to FEAT_ETS3, see DSB-ordered-before and ETS-ordered-before.

I_{LTSNZ} For more information, see:

- External visibility requirement.
- Memory barriers.

D8.2.6.2 Security state of translation table lookups

R_{PXRPP} For a translation table walk in a Non-secure translation regime, all translation table lookups are done to Non-secure IPA or PA space.

I_{MBWLG} When EL2 is enabled, the OA space of the Secure EL1&0 stage 1 translation regime is IPA space. In all other cases, the OA space for a Secure stage 1 translation regime is PA space.

R_{SCHHQ} For a stage 1 translation table walk in a Secure Translation Regime, including the EL3 translation regime when FEAT_RME is not implemented, the OA space at stage 1 of the translation table lookups is determined by all of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The initial translation table lookup is done to Secure OA space.
- When the Table descriptor lookup is to Secure OA space, the descriptor NSTable bit determines one of the following:
  - If NSTable is zero, the next translation table lookup is done to Secure OA space.
  - If NSTable is one, the next translation table lookup is done to Non-secure OA space.
- When the Table descriptor lookup is to Non-secure OA space, the next translation table lookup is done to Non-secure OA space.

For more information, see Hierarchical control of Secure or Non-secure memory accesses.

I_{VRNWF} For a stage 2 translation in a Secure translation regime, the PA space of translation table lookups is determined by one of the following:

- When translating an address in Secure IPA space, the Effective value of the VSTCR_EL2.SW bit.
- When translating an address in Non-secure IPA space, the Effective value of the VTCR_EL2.NSW bit.

I_{KXGHL} When an IPA from a Secure translation regime is translated by stage 2 translation, the output PA space of the translation is determined by one of the following:

- When translating an address in Secure IPA space, the Effective value of the VSTCR_EL2.SA bit.
- When translating an address in Non-secure IPA space, the Effective value of the VTCR_EL2.NSA bit.

R_{DVGRP} For a stage 1 translation table walk in the Realm EL2 or Realm EL2&0 translation regime, all translation table lookups are done to the Realm PA space.

I_{TGPMX} Regardless of whether stage 1 translation is enabled or disabled in the Realm EL1&0 translation regime, Realm EL2 is always enabled and the OA of a Realm EL1&0 stage 1 translation is a Realm IPA.

R_{PGRQD} For a stage 2 translation table walk in the Realm EL1&0 translation regime, all translation table lookups are done to the Realm PA space.

I_{KCYMF} For a stage 2 translation table walk in the Realm EL1&0 translation regime, VTCR_EL2[30:29] are RES0 and there is no equivalent of the NSA, NSW fields.

R_{CFPDJ} For a stage 1 translation table walk in the EL3 translation regime, if FEAT_RME is implemented, then all translation table lookups are done to the Root PA space.

D8.2.7 Translation Hardening Extension

R_{FYNMT} All statements in this section and subsections require implementation of the OPTIONAL feature, FEAT_THE.

I_{YGPNC} To protect the integrity of kernel translations, the Translation Hardening Extension, FEAT_THE, prevents the Exception level that owns the translation tables from modifying an arbitrary subset of those translation table entries.

D8.2.7.1 Stage 1 Protected Attribute

R_{MCDSQ} For stage 1 of each translation regime using VMSAv8-64, the value of PnCH is determined as shown in the following table:

Table D8-11 Determination of PnCH

Translation regime	Field
EL1&0	TCR2_EL1.PnCH.
EL2&0	TCR2_EL2.PnCH.
EL2	TCR2_EL2.PnCH.
EL3	TCR_EL3.PnCH.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NHZZY} If one of the following is true, then Protection is enabled:

- If stage 1 translation is using VMSAv8-64 and PnCH is 1.
- If stage 1 translation is using VMSAv9-128.

Otherwise, Protection is disabled. This applies even when stage 1 translation is not enabled.

R_{NBVJL} If Protection is enabled, then the value of the Protected bit in the stage 1 descriptor indicates all of the following:

- If the Protected bit is 0, the descriptor is said to be Non-protected.
- If the Protected bit is 1, the descriptor is said to be Protected.

This applies even when stage 1 translation is not enabled.

R_{VDPDW} For a stage 1 descriptor, if Protection is enabled and the RCW checks fail, then `RCW` and `RCWS` instructions do not update that descriptor. The RCW checks are:

- RCW State Checks:
  - For a Protected descriptor, there is an attempt to change the protection or validity of that descriptor.
  - For a Non-protected descriptor, there is an attempt to change the protection of that descriptor.
- RCW Mask Check:
  - For a Protected and valid descriptor, when an attempt is made to update a bit in that descriptor and the Effective value of the corresponding RCWMASK_EL1 bit is 0.

R_{NYFNK} For a stage 1 descriptor, if RCWS checks fail, then `RCWS` instructions do not update that descriptor. The RCWS checks are:

- RCWS State check:
  - This check fails if there is an attempt to change the validity of that descriptor and one of the following applies:
    - The descriptor is valid.
    - The descriptor is invalid and Protection is disabled.
    - The descriptor is invalid and it is a Non-protected descriptor.
- RCWS Mask check:
  - This check fails if, for a valid descriptor, an attempt is made to update a bit in that descriptor and the Effective value of the corresponding RCWSMASK_EL1 bit is 0.

I_{LBYCD} Even when a descriptor is not Protected, the bitmask in RCWSMASK_EL1 controls which fields in a descriptor are permitted to be updated by an `RCWS` instruction, as follows:

- A value of 0 indicates that the corresponding bit of the descriptor cannot be changed.
- A value of 1 indicates that the corresponding bit of the descriptor can be changed.

I_{QSXGJ} It is expected that the contents of RCWMASK_EL1 and RCWSMASK_EL1 registers are set once by the hypervisor for each instance of a Virtual Machine.

R_{CRSZF} If Protection is not enabled, then `RCW` instructions can change all bits in a stage 1 descriptor.

I_{LWWYX} `RCWS` instructions are subject to the RCWS State and Mask checks, even if Protection is disabled.

R_{QVMPP} When stage 1 translation is disabled at ELx, all of the following apply:

- `RCW` instructions are subjected to RCW checks.
- `RCWS` instructions are subjected to RCW checks and RCWS checks.

I_{XSNQG} It is expected that `RCW` instructions at ELx are used to update stage 1 descriptors owned by ELx. `RCW` instructions at ELx are not expected to update stage 1 descriptors not owned by ELx, or stage 2 descriptors.

I_{MWKTV} It is expected that `RCWS` instructions are used to update bits in descriptors that are reserved for software use and, correspondingly, only bits reserved for software use are set to 1 in RCWSMASK_EL1. If descriptor changes affect only bits that are interpreted by software, then it is not necessary to do TLB maintenance to synchronize the effect on the TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.7.2 Stage 1 Reduced Coherence write

R_{NRVRT} A write has the Reduced Coherence property if and only if it is written by an `RCWS` instruction and *Permit Translation Table Walk Incoherence*, PTTWI, is 1 on the PE executing that `RCWS` instruction.

R_{QWZDZ} For stage 1 translations in a translation regime, the value of PTTWI is obtained as shown in the following table:

Table D8-12 Determination of PTTWI

Translation Regime	Field
EL1&0	If EL2 is not enabled, then TCR2_EL1.PTTWI. If EL2 is enabled, then TCR2_EL1.PTTWI && HCRX_EL2.PTTWI.
EL2&0	TCR2_EL2.PTTWI
EL2	TCR2_EL2.PTTWI
EL3	TCR_EL3.PTTWI

R_{DTFHS} If all of the following apply, a read of a memory location as part of a translation table walk is permitted to observe the most-recent write entry without the Reduced Coherence property, W1, in the coherence order of that memory location:

- The read is done as part of a translation table walk.
- The read is not the read effect due to a hardware update of a translation table entry.
- The read is permitted by the memory model to observe a write entry with the Reduced Coherence property, W2, in the coherence order of that memory location.

R_{GLBZZ} The effect of PTTWI on `RCWS` instructions applies regardless of whether translation is enabled or disabled.

R_{YHRDF} It is permitted to implement each PTTWI bit as a read-only bit with a fixed value of 0.

I_{BBTRQ} PTTWI is permitted to be cached in a TLB.

D8.2.7.3 Assured translation

R_{SSRXH} If all of the following apply, the level of the stage 1 translation table walk for a VA has the Assured Translation property:

- The EL1&0 translation regime has two translation stages enabled.
- For the initial lookup level of a walk, one of the following applies:
  - The walk is from TTBR0_EL1 and VTCR_EL2.TL0 is 1.
  - The walk is from TTBR1_EL1 and VTCR_EL2.TL1 is 1.
- For subsequent levels, all of the following apply:
  - All previous levels of the stage 1 walk have the Assured Translation property.
  - The stage 1 translation table descriptor is a Protected descriptor and is fetched from an IPA with the with stage 2 MRO permission.

Note

When Stage 2 Overlay permissions are enabled, S2OverlayPerm can provide MRO permission for an IPA. To ensure that EL1 cannot create Assured translations for itself, EL2 must either disable Stage 2 Overlay permissions or disable EL1 access to S2POR_EL1. For more information, see Stage 2 Overlay permissions.

R_{YXKRB} The IPA AssuredOnly attribute is determined as shown in the following table:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-13 Determination of AssuredOnly

Stage 2 format	VTCR_EL2.AssuredOnly	AssuredOnly attribute
VMSAv8-64	0	0
VMSAv8-64	1	Block or Page descriptor AssuredOnly field
VMSAv9-128	RES0	Block or Page descriptor AssuredOnly field

R_{WWYDL} If an IPA is translated by a stage 2 Block or Page descriptor with the AssuredOnly attribute set to 1, and any of the following apply, then any access translated by that descriptor generates a stage 2 Permission fault:

- The level of the stage 1 translation table walk that generated the IPA does not have the Assured Translation property.
- Stage 1 translation is disabled.

R_{QNQZJ} If all of the following apply, privileged Guarded control stack data accesses from EL1 to a memory location generate a stage 2 Permission fault due to an AssuredOnly check, independent of whether the stage 1 translation has the Assured Translation property:

- VTCR_EL2.GCSH is 1.
- Stage 2 translation is enabled.
- The memory location is translated using a stage 2 Block or Page descriptor with the AssuredOnly attribute set to 0.

For more information, see Guarded Control Stack data accesses.

D8.2.7.4 Stage 2 TopLevel checks

I_{ZCRKQ} If stage 2 translation is enabled, then the stage 2 TopLevel0 and TopLevel1 permissions indicate that the IPA can be used as a top-level table for TTBR0_EL1 and TTBR1_EL1 respectively.

For more information, see Stage 2 Base permissions.

R_{BCSRJ} For a VA translated by TTBR0_EL1, the stage 2 translation of the initial lookup level of the stage 1 translation table walk generates a stage 2 Permission fault due to the TopLevel checks according to the conditions in the following table:

Table D8-14 TopLevel check conditions for translations by TTBR0_EL1

VTCR_EL2.TL0	VTCR_EL2.TL1	{TopLevel0, TopLevel1}	Permission fault
0	0	{x, x}	No
0	1	{0, 1}	Yes
		{0, 0}, {1, x}	No
1	x	{0, x}	Yes
		{1, x}	No

For more information, see Stage 2 Base permissions.

R_{SWLZW} For a VA translated by TTBR1_EL1, the stage 2 translation of the initial lookup level of the stage 1 translation table walk generates a stage 2 Permission fault due to the TopLevel checks according to the conditions in the following table:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-15 TopLevel check conditions for translations by TTBR1_EL1

VTCR_EL2.TL0	VTCR_EL2.TL1	{TopLevel0, TopLevel1}	Permission fault
0	0	{x, x}	No
1	0	{1, 0}	Yes
		{0, 0}, {x, 1}	No
x	1	{x, 0}	Yes
		{x, 1}	No

For more information, see Stage 2 Base permissions.

D8.2.8 VMSAv8-64 translation using the 4KB granule

R_{LCVYF} All statements in this section and subsections require a translation stage use the VMSAv8-64 translation system.

I_{SZLLC} Address translations that use the 4KB granule have a 4KB page size. Depending on the settings and supported features, up to 40 address bits are resolved using up to 5 lookup levels.

R_{FMKVP} Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

I_{VNNCG} For the 4KB translation granule, the maximum VA and PA supported by a translation regime is one of the following:

- If the Effective value of TCR_ELx.DS is 0, then the maximum VA and PA supported is 48 bits.
- If the Effective value of TCR_ELx.DS is 1, then the maximum VA and PA supported is 52 bits.

R_{QGLQX} For the 4KB translation granule, if the Effective value of TCR_ELx.DS is 0, then OA[51:48] are treated as `0b0000`.

R_{KSBQV} For each lookup level supported by the 4KB translation granule, the following table describes the translation table properties at that level.

Table D8-16 4KB granule translation table properties at each lookup level

Lookup level	Index into translation table	Maximum entries in table	Contents of translation table entries	Additional requirements
-1	-	-	Lookup level not supported	Effective value of TCR_ELx.DS is 0.
	IA[51:48]	16	Table descriptors	Effective value of TCR_ELx.DS is 1.
0	IA[47:39]	512	Table descriptors	Effective value of TCR_ELx.DS is 0.
			Table descriptors and Block descriptors	Effective value of TCR_ELx.DS is 1.
1	IA[38:30]	512	Table descriptors and Block descriptors	-
2	IA[29:21]	512	Table descriptors and Block descriptors	-
3	IA[20:12]	512	Page descriptors	-

R_{YLGLV} For the 4KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 4KB.
- The translation can resolve a page using one of the following maximum address ranges:
  - If the Effective value of TCR_ELx.DS is 0, then IA[47:12].
  - If the Effective value of TCR_ELx.DS is 1, then IA[51:12].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The page is addressed by one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then OA[47:12].
  - If the Effective value of TCR_ELx.DS is 1, then OA[51:12].
- IA[11:0] is mapped directly to OA[11:0].

R_{NKQNK} For the 4KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

Table D8-17 4KB granule block descriptor properties at each lookup level

Lookup level	Size of memory region addressed by Block descriptor	Bit range that is direct mapped from IA to OA	IA bit range that selects Block descriptor	Effective value of TCR_ELx.DS
0	Not supported	-	-	0
	512GB	IA[38:0] maps to OA[38:0]	IA[51:39]	1
1	1GB	IA[29:0] maps to OA[29:0]	IA[47:30]	0
			IA[51:30]	1
2	2MB	IA[20:0] maps to OA[20:0]	IA[47:21]	0
			IA[51:21]	1

I_{XLXHW} For the 4KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. An IA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1.

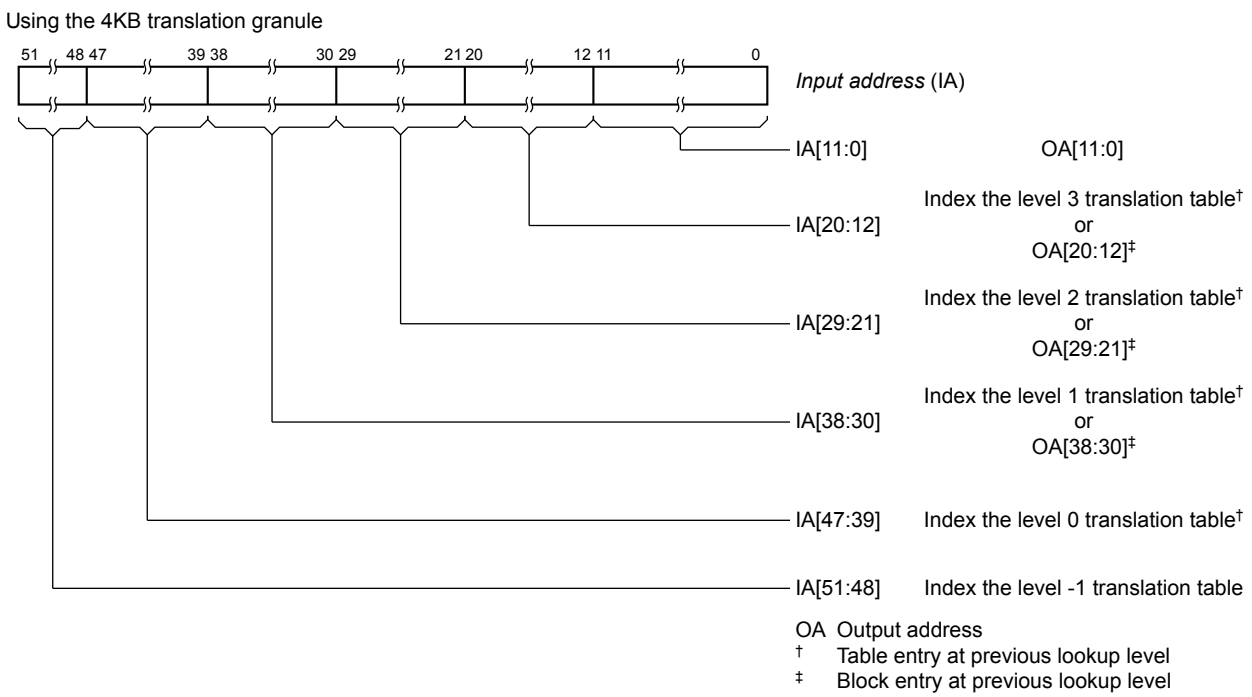

Figure D8-3 52-bit IA resolved using 4KB translation granule

D8.2.8.1 VMSAv8-64 Stage 1 address translation using the 4KB translation granule

R_{GWNBR} For the 4KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.T*n*SZ field as shown in the following table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-18 4KB granule, determining stage 1 initial lookup level

Initial lookup level	T*n*SZ minimum value	Maximum IA bits resolved	T*n*SZ maximum value	Minimum IA bits resolved	Additional requirements
-1	12	IA[51:12]	15	IA[48:12]	Effective value of TCR_ELx.DS is 1
0	16	IA[47:12]	24	IA[39:12]	-
1	25	IA[38:12]	33	IA[30:12]	-
2	34	IA[29:12]	39	IA[24:12]	-
2	40	IA[23:12]	42	IA[21:12]	FEAT_TTST is implemented
3	43	IA[20:12]	48	IA[15:12]	FEAT_TTST is implemented

IJCHLN For a stage 1 translation in the 4KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 9 bits and all remaining lookup levels are indexed by exactly 9 bits.

IVKJBP For the 4KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA and an OA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1. The 512GB block size requires that the Effective value of TCR_ELx.DS is 1.

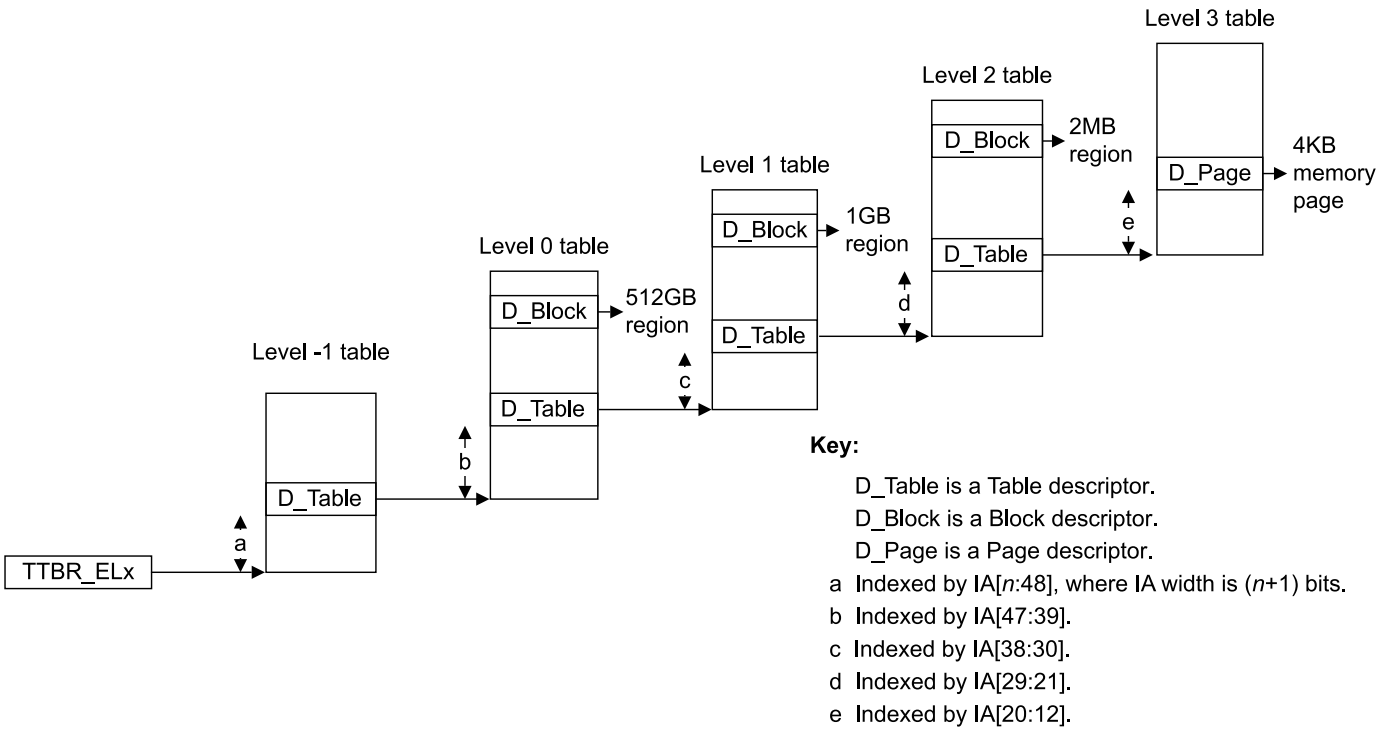

Figure D8-4 Generalized view of a stage 1 address translation using the 4KB granule

D8.2.8.2 VMSAv8-64 Stage 2 address translation using the 4KB translation granule

IGVFLG For the 4KB translation granule, when a stage 2 translation table walk is started, the initial lookup level is determined by one of the following:

- If the Effective value of VTCR_EL2.DS is 0, then the initial lookup level is determined by one of VTCR_EL2.SL0 or VSTCR_EL2.SL0, and VTCR_EL2.SL2 and VSTCR_EL2.SL2 are `RES0`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of VTCR_EL2.DS is 1, then the initial lookup level is determined by the combination of VTCR_EL2.SL0 and VTCR_EL2.SL2, or VSTCR_EL2.SL0 and VSTCR_EL2.SL2.

IPBGCS For the 4KB translation granule, when a stage 2 translation table walk is started, the following table shows the initial lookup level determined by VTCR_EL2.SL0 or VSTCR_EL2.SL0 and, when the Effective value of VTCR_EL2.DS is 1, the corresponding VTCR_EL2.SL2 or VSTCR_EL2.SL2.

Table D8-19 4KB granule, determining stage 2 initial lookup level

SL2	SL0	Initial lookup level
0b0	0b00	Level 2
0b0	0b01	Level 1
0b0	0b10	Level 0
0b0	0b11	Level 3
0b1	0b00	Level -1
0b1	0b01	Reserved
0b1	0b10	Reserved
0b1	0b11	Reserved

RDBNDG For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is -1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-20 4KB granule, stage 2 initial lookup at level -1

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[51:12]-IA[48-12]	12-15	Effective value of VTCR_EL2.DS is 1

RPZFHQ For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 0, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-21 4KB granule, stage 2 initial lookup at level 0

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[47:12]-IA[39:12]	16-24	-
2	IA[48:12]	15	Effective value of VTCR_EL2.DS is 1
4	IA[49:12]	14	Effective value of VTCR_EL2.DS is 1
8	IA[50:12]	13	Effective value of VTCR_EL2.DS is 1
16	IA[51:12]	12	Effective value of VTCR_EL2.DS is 1

RMKPBG For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-22 4KB granule, stage 2 initial lookup at level 1

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[38:12]-IA[30:12]	25-33	-
2	IA[39:12]	24	-
4	IA[40:12]	23	-
8	IA[41:12]	22	-
16	IA[42:12]	21	-

R_{KKRSZ} For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 2, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-23 4KB granule, stage 2 initial lookup at level 2

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[23:12]-IA[21:12]	40-42	FEAT_TTST is implemented
None (1 table)	IA[29:12]-IA[24:12]	34-39	-
2	IA[30:12]	33	-
4	IA[31:12]	32	-
8	IA[32:12]	31	-
16	IA[33:12]	30	-

R_{TJMNP} For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 3, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-24 4KB granule, stage 2 initial lookup at level 3

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[20:12]-IA[15:12]	43-48	FEAT_TTST is implemented
2	IA[21:12]	42	FEAT_TTST is implemented
4	IA[22:12]	41	FEAT_TTST is implemented
8	IA[23:12]	40	FEAT_TTST is implemented
16	IA[24:12]	39	FEAT_TTST is implemented

I_{SWBVJ} For the 4KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 2 address translation. An IA larger than 48 bits requires that the Effective value of VTCR_EL2.DS is 1. The 512GB block size requires that the Effective value of VTCR_EL2.DS is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VTCR_EL2.{SL2, SL0} defines the start level.

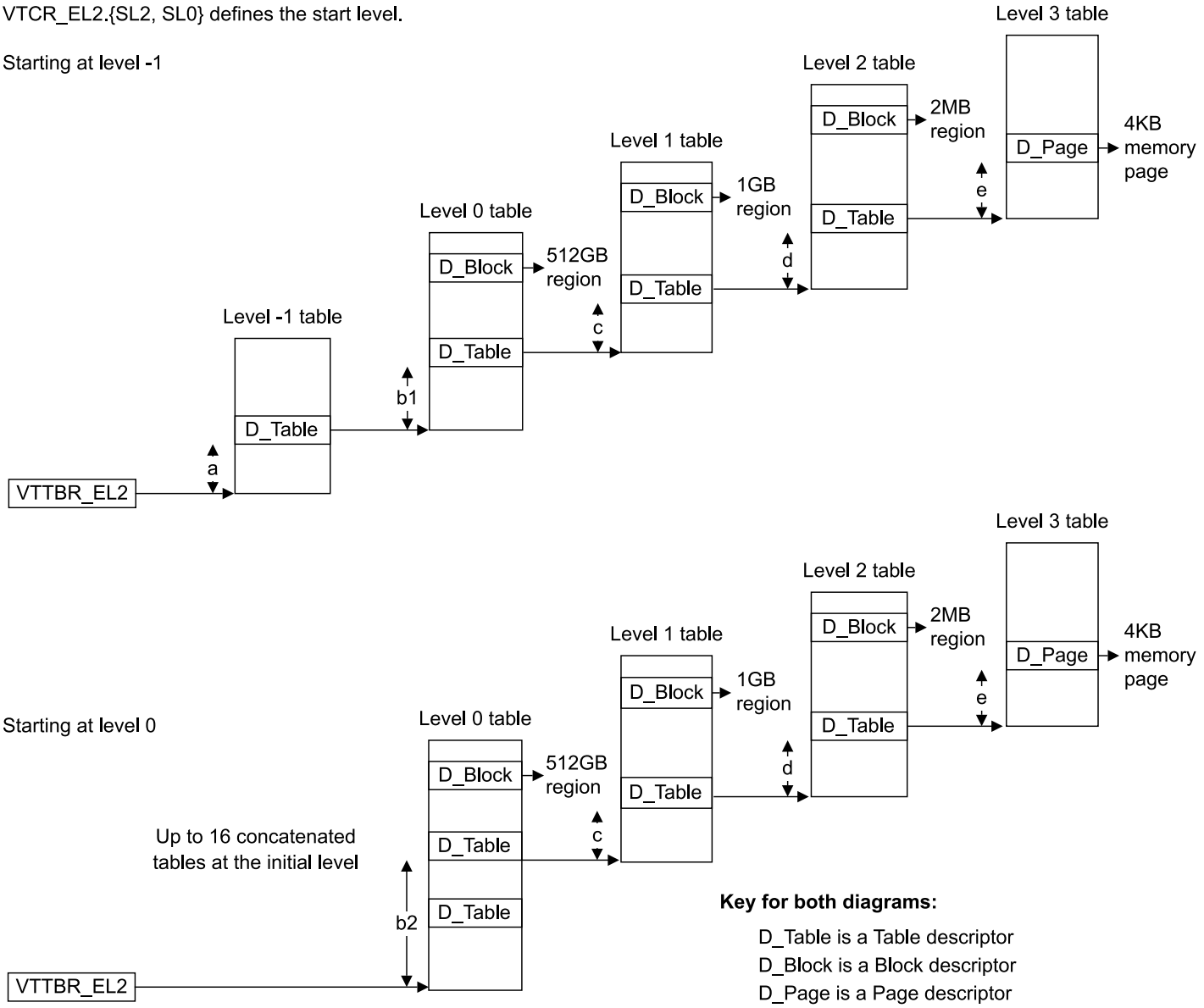

Figure D8-5 Generalized view of a stage 2 address translation using the 4KB granule

R_{KTKWK} For the 4KB translation granule, when a stage 2 translation table walk is started and one of the following is true, a stage 2 level 0 Translation fault is generated:

- If the Effective value of VTCR_EL2.DS is 0, then one or more of the following is true:
  - The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.
  - VTCR_EL2.SL0 or VSTCR_EL2.SL0 is programmed to a reserved value.
- If the Effective value of VTCR_EL2.DS is 1, then one or more of the following is true:
  - The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.{SL2, SL0} or VSTCR_EL2.{SL2, SL0} value.
  - VTCR_EL2.{SL2, SL0} or VSTCR_EL2.{SL2, SL0} is programmed to a reserved value.

I_{HBWKN} For more information, see Concatenated translation tables.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.8.3 Finding the descriptor when using the VMSAv8-64 4KB translation granule

R_{VMRQS} For the 4KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using of all of the following information:

- The translation table base address, *BaseAddr*.
- The number of bits in the supported PA size, *PAsize*.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.T*n*SZ.
- For a stage 2 translation, all of the following:
  - The value of VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ.
  - The value of VTCR_EL2.SL0 or VSTCR_EL2.SL0.
  - If the Effective value of VTCR_EL2.DS is 1, then the value of VTCR_EL2.SL2 or VSTCR_EL2.SL2.

Table D8-25 4KB granule, finding the descriptor address

Lookup level	Stage 1 translation table descriptor address	Stage 2 translation table descriptor address
-1	if 12 <= T*n*SZ <= 15 then *x* = (19 - T*n*SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+44:48]:`0b000`	if SL2==1 and SL0==0 then if 12 <= T0SZ <= 15 then *x* = (19 - T0SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+44:48]:`0b000`
0	if 16 <= T*n*SZ <= 24 then *x* = (28 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+35:39]:`0b000`	if SL0==2 then if 12 <= T0SZ <= 24 then *x* = (28 - T0SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+35:39]:`0b000`
1	if 25 <= T*n*SZ <= 33 then *x* = (37 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+26:30]:`0b000`	if SL0==1 then if 21 <= T0SZ <= 33 then *x* = (37 - T0SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+26:30]:`0b000`
2	if 34 <= T*n*SZ <= 42 then *x* = (46 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+17:21]:`0b000`	if SL0==0 then if 30 <= T0SZ <= 42 then *x* = (46 - T0SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+17:21]:`0b000`
3	if 43 <= T*n*SZ <= 48 then *x* = (55 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+8:12]:`0b000`	if SL0==3 then if 39 <= T0SZ <= 48 then *x* = (55 - T0SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+8:12]:`0b000`

D8.2.9 VMSAv8-64 translation using the 16KB granule

R_{QSQNQ} All statements in this section and subsections require a translation stage use the VMSAv8-64 translation system.

I_{TDTCY} Address translations that use the 16KB granule have a 16KB page size. Depending on the settings and supported features, up to 38 address bits are resolved using up to 4 lookup levels.

R_{VQYZX} Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

I_{JCYXY} For the 16KB translation granule, the maximum VA and PA supported by a translation regime is one of the following:

- If the Effective value of TCR_ELx.DS is 0, then the maximum VA and PA supported is 48 bits.
- If the Effective value of TCR_ELx.DS is 1, then the maximum VA and PA supported is 52 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CDLYH} For the 16KB translation granule, if the Effective value of TCR_ELx.DS is 0, then OA[51:48] are treated as `0b0000`.

R_{VGVLD} For each lookup level supported by the 16KB translation granule, the following table describes the translation table properties at that level.

Table D8-26 16KB granule translation table properties at each lookup level

Lookup level	Index into translation table	Maximum entries in table	Contents of translation table entries	Additional requirements
0	IA[47]	2	Table descriptors Lookup level 0 supported only at stage 1	Effective value of TCR_ELx.DS is 0
	IA[51:47]	32	Table descriptors Lookup level 0 supported at stages 1 and 2	Effective value of TCR_ELx.DS is 1
1	IA[46:36]	2048	Table descriptors	Effective value of TCR_ELx.DS is 0
			Table descriptors and Block descriptors	Effective value of TCR_ELx.DS is 1
2	IA[35:25]	2048	Table descriptors and Block descriptors	-
3	IA[24:14]	2048	Page descriptors	-

R_{RYDBQ} For the 16KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 16KB.
- The translation can resolve a page using one of the following maximum address ranges:
  - If the Effective value of TCR_ELx.DS is 0, then IA[47:14].
  - If the Effective value of TCR_ELx.DS is 1, then IA[51:14].
- The page is addressed by one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then OA[47:14].
  - If the Effective value of TCR_ELx.DS is 1, then OA[51:14].
- IA[13:0] is mapped directly to OA[13:0].

R_{FFHSQ} For the 16KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

Table D8-27 16KB granule block descriptor properties at each lookup level

Lookup level	Size of memory region addressed by Block descriptor	Bit range that is direct mapped from IA to OA	IA bit range that selects Block descriptor	Effective value of TCR_ELx.DS
1	Not supported	-	-	0
	64GB	IA[35:0] maps to OA[35:0]	IA[51:36]	1
2	32MB	IA[24:0] maps to OA[24:0]	IA[47:25]	0
			IA[51:25]	1

I_{MJSJR} For the 16KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. An IA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

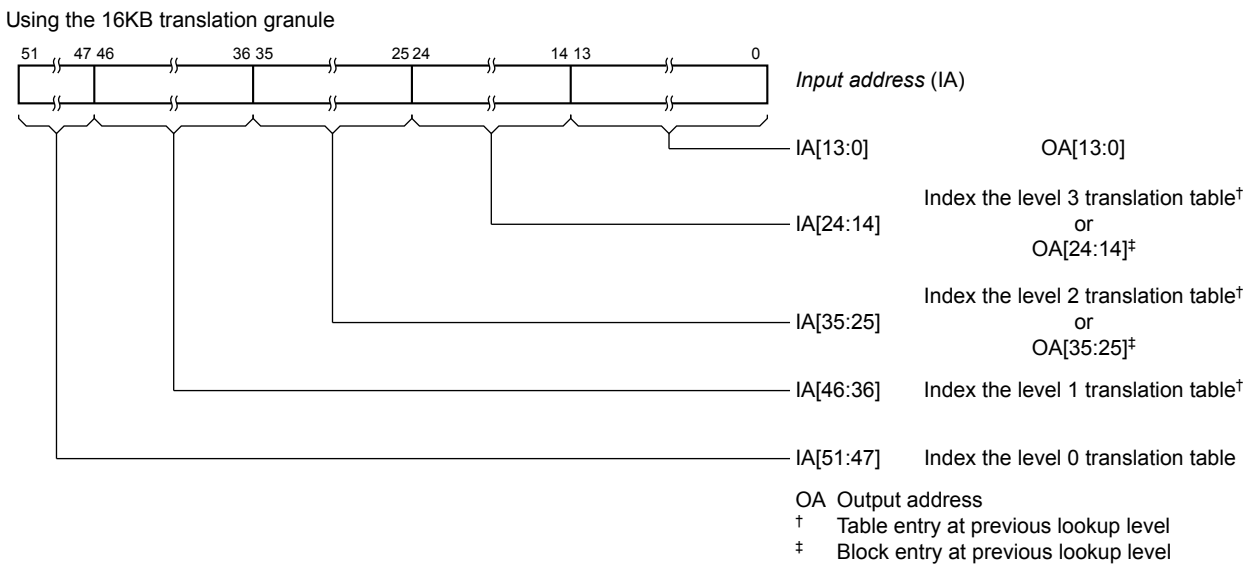

Figure D8-6 52-bit IA resolved using 16KB translation granule

D8.2.9.1 VMSAv8-64 Stage 1 address translation using the 16KB translation granule

R_{JHFFJ} For the 16KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.T*n*SZ field as shown in the following table:

Table D8-28 16KB granule, determining stage 1 initial lookup level

Initial lookup level	T*n*SZ minimum value	Maximum IA bits resolved	T*n*SZ maximum value	Minimum IA bits resolved	Additional requirements
0	12	IA[51:14]	15	IA[48:14]	Effective value of TCR_ELx.DS is 1
0	16	IA[47:14]	16	IA[47:14]	-
1	17	IA[46:14]	27	IA[36:14]	-
2	28	IA[35:14]	38	IA[25:14]	-
3	39	IA[24:14]	39	IA[24:14]	-
3	40	IA[23:14]	48	IA[15:14]	FEAT_TTST is implemented

I_{FLDKN} For a stage 1 translation in the 16KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 11 bits and all remaining lookup levels are indexed by exactly 11 bits.

I_{KSSHJ} For the 16KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA and an OA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1. The 64GB block size requires that the Effective value of TCR_ELx.DS is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

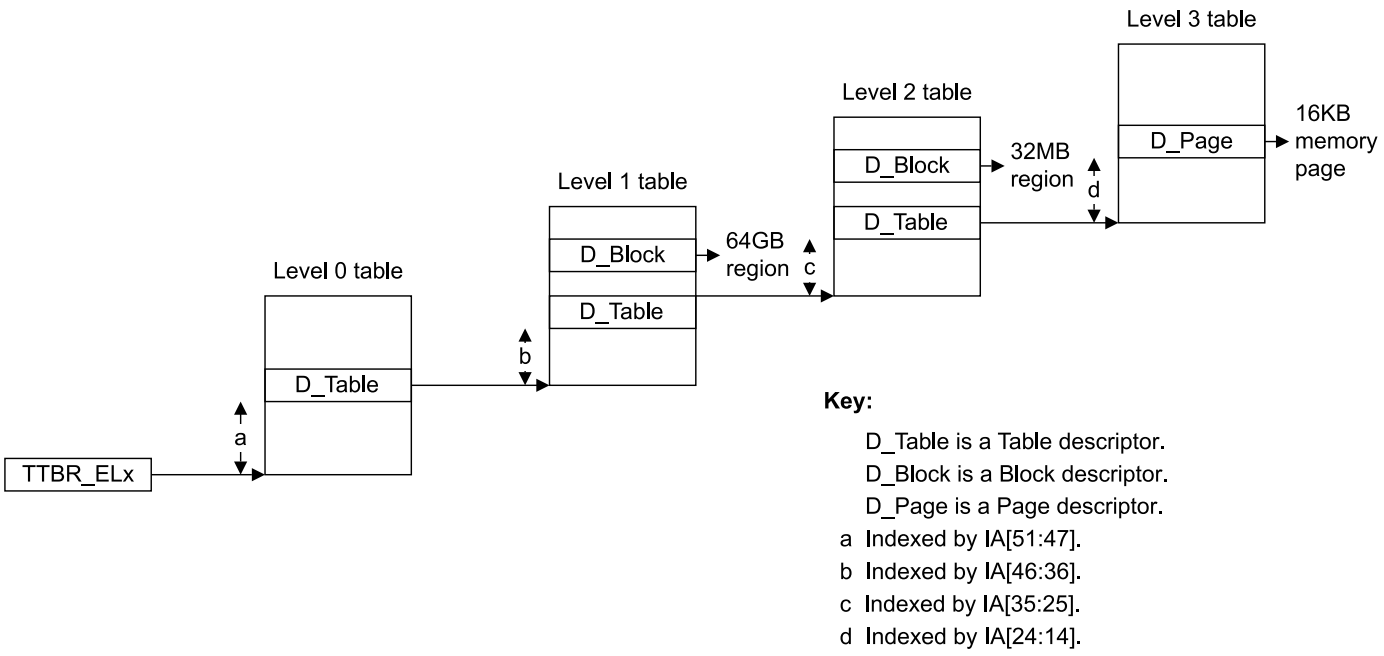

Figure D8-7 Generalized view of a stage 1 address translation using the 16KB granule

D8.2.9.2 VMSAv8-64 Stage 2 address translation using the 16KB translation granule

IQDHZV For the 16KB translation granule, when a stage 2 translation table walk is started, the initial lookup level is determined by the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.

IMKHHJ For the 16KB translation granule, when a stage 2 translation table walk is started, the following table shows the initial lookup level determined by SL0:

Table D8-29 16KB granule, determining stage 2 initial lookup level

SL0	Initial lookup level
`0b00`	Level 3
`0b01`	Level 2
`0b10`	Level 1
`0b11`	Level 0

RJKYLY For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 0, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-30 16KB granule, stage 2 initial lookup at level 0

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[51:14]-IA[47:14]	12-16	Effective value of VTCR_EL2.DS is 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{FBHPY} For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-31 16KB granule, stage 2 initial lookup at level 1

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[46:14]-IA[36:14]	17-27	-
2	IA[47:14]	16	-
4	IA[48:14]	15	Effective value of VTCR_EL2.DS is 1
8	IA[49:14]	14	Effective value of VTCR_EL2.DS is 1
16	IA[50:14]	13	Effective value of VTCR_EL2.DS is 1

R_{MJPYK} For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 2, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-32 16KB granule, stage 2 initial lookup at level 2

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[35:14]-IA[25:14]	28-38	-
2	IA[36:14]	27	-
4	IA[37:14]	26	-
8	IA[38:14]	25	-
16	IA[39:14]	24	-

R_{WDMGC} For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 3, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-33 16KB granule, stage 2 initial lookup at level 3

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[23:14]-IA[15:14]	40-48	FEAT_TTST is implemented
None (1 table)	IA[24:14]	39	-
2	IA[25:14]	38	-
4	IA[26:14]	37	-
8	IA[27:14]	36	-
16	IA[28:14]	35	-

I_{GBRMW} For the 16KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 2 address translation. An IA larger than 48 bits requires that the Effective value of VTCR_EL2.DS is 1. The 64GB block size requires that the Effective value of VTCR_EL2.DS is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

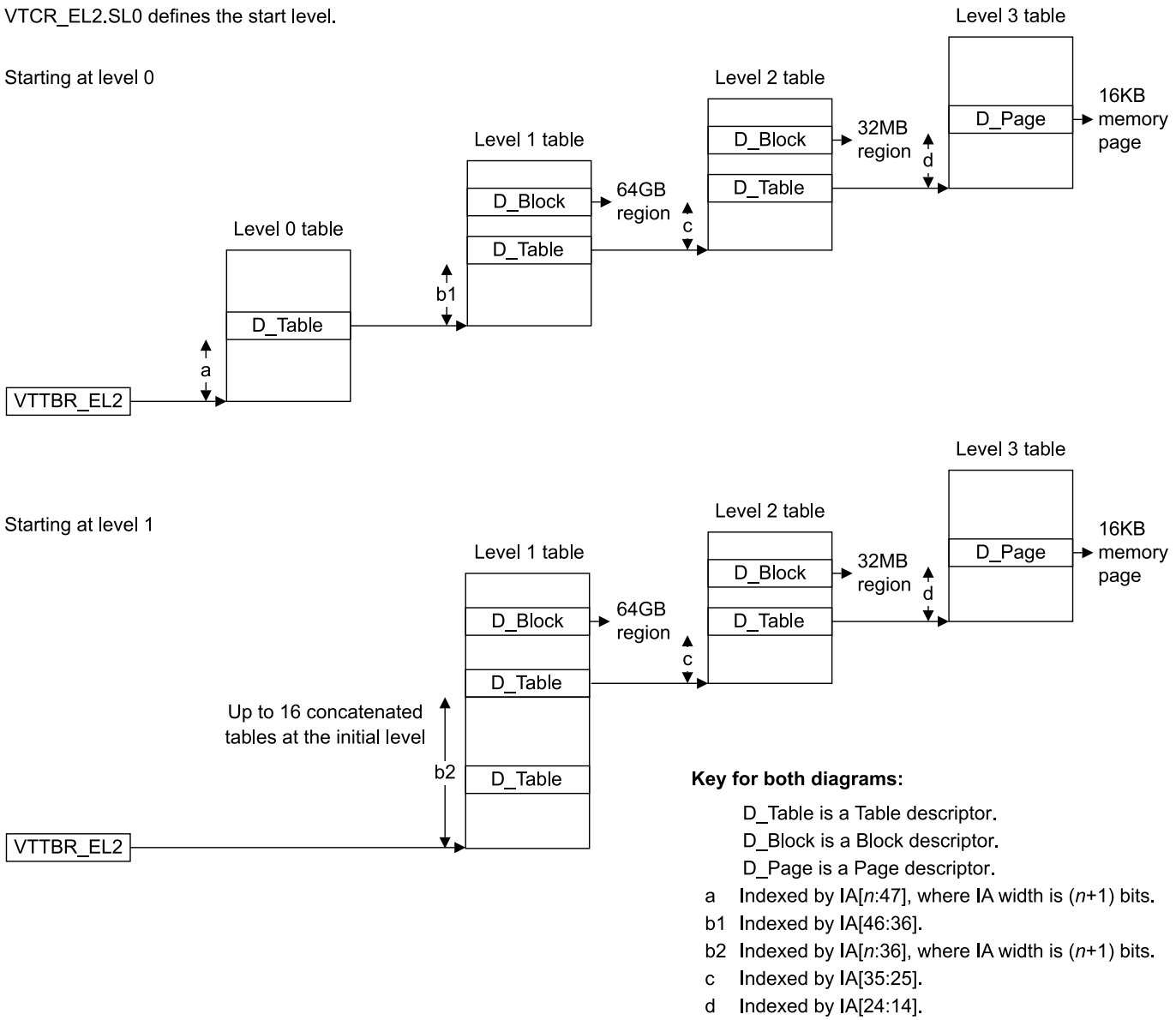

Figure D8-8 Generalized view of a stage 2 address translation using the 16KB granule

R_{NWPLD} For the 16KB translation granule, when a stage 2 translation table walk is started and one of the following is true, a stage 2 level 0 Translation fault is generated:

- The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.
- VTCR_EL2.SL0 or VSTCR_EL2.SL0 is programmed to a reserved value.

I_{FXRPG} For more information, see Concatenated translation tables.

D8.2.9.3 Finding the descriptor when using the VMSAv8-64 16KB translation granule

R_{KMTBZ} For the 16KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using of all of the following information:

- The translation table base address, *BaseAddr*.
- The number of bits in the supported PA size, *PAsize*.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.T*n*SZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a stage 2 translation, all of the following:
  - The value of VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ.
  - The value of VTCR_EL2.SL0 or VSTCR_EL2.SL0.

Table D8-34 16KB granule, finding the descriptor address

Lookup level	Stage 1 translation table descriptor address	Stage 2 translation table descriptor address
0	if 12 <= T*n*SZ <= 16 then *x* = (20 - T*n*SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+43:47]:`0b000`	if SL0==3 then if 12 <= T0SZ <= 16 then *x* = (20 - T0SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+43:47]:`0b000`
1	if 17 <= T*n*SZ <= 27 then *x* = (31 - T*n*SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+32:36]:`0b000`	if SL0==2 then if 13 <= T0SZ <= 27 then *x* = (31 - T0SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+32:36]:`0b000`
2	if 28 <= T*n*SZ <= 38 then *x* = (42 - T*n*SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+21:25]:`0b000`	if SL0==1 then if 24 <= T0SZ <= 38 then *x* = (42 - T0SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+21:25]:`0b000`
3	if 39 <= T*n*SZ <= 48 then *x* = (53 - T*n*SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+10:14]:`0b000`	if SL0==0 then if 35 <= T0SZ <= 48 then *x* = (53 - T0SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+10:14]:`0b000`

D8.2.10 VMSAv8-64 translation using the 64KB granule

R_{RGNZV} All statements in this section and subsections require a translation stage use the VMSAv8-64 translation system.

I_{VDPWL} Address translations that use the 64KB granule have a 64KB page size. Depending on the settings and supported features, up to 36 address bits are resolved using up to 3 lookup levels.

I_{LRFQD} For the 64KB translation granule, the maximum VA size supported by a translation regime is one of the following:

- If FEAT_LVA is not implemented, then the maximum VA supported is 48 bits.
- If FEAT_LVA is implemented, then the maximum VA supported is 52 bits.

I_{YZDPC} For the 64KB translation granule, the maximum PA size supported by a translation regime is one of the following:

- If FEAT_LPA is not implemented, then the maximum PA supported is 48 bits.
- If FEAT_LPA is implemented, then the maximum PA supported is 52 bits.

R_{NSQXP} For the 64KB translation granule, if FEAT_LPA is not implemented, then OA[51:48] are treated as `0b0000`.

R_{MLLGN} For each lookup level supported by the 64KB translation granule, the following table describes the translation table properties at that level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-35 64KB granule translation table properties at each lookup level

Lookup level	Index into translation table	Maximum entries in table	Contents of translation table entries	Additional requirements for stage 1	Additional requirements for stage 2
1	IA[47:42]	64	Table descriptors	-	-
			Table descriptors and Block descriptors	FEAT_LPA	FEAT_LPA
	IA[51:42]	1024	Table descriptors	FEAT_LVA	-
			Table descriptors and Block descriptors	FEAT_LVA, FEAT_LPA	FEAT_LPA
2	IA[41:29]	8192	Table descriptors and Block descriptors	-	-
3	IA[28:16]	8192	Page descriptors	-	-

R_{ZWQBH} For the 64KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 64KB.
- For stage 1, the translation can resolve a page using one of the following maximum address ranges:
  - If FEAT_LVA is not implemented, then IA[47:16].
  - If FEAT_LVA is implemented, then IA[51:16].
- For stage 2, the translation can resolve a page using one of the following maximum address ranges:
  - If FEAT_LPA is not implemented, then IA[47:16].
  - If FEAT_LPA is implemented, then IA[51:16].
- For stage 1 and stage 2, the page is addressed by one of the following:
  - If FEAT_LPA is not implemented, then OA[47:16].
  - If FEAT_LPA is implemented, then OA[51:16].
- IA[15:0] is mapped directly to OA[15:0].

R_{RBNGS} For the 64KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

Table D8-36 64KB granule block descriptor properties at each lookup level

Lookup level	Size of memory region addressed by Block descriptor	Bit range that is direct mapped from IA to OA	IA bit range that selects Block descriptor	Additional requirements for stage 1	Additional requirements for stage 2
1	4TB	IA[41:0] maps to OA[41:0]	IA[47:42]	FEAT_LPA	FEAT_LPA
			IA[51:42]	FEAT_LVA, FEAT_LPA	FEAT_LPA
2	512MB	IA[28:0] maps to OA[28:0]	IA[47:29]	-	-
			IA[51:29]	FEAT_LVA	FEAT_LPA

I_{TVDSK} For the 64KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. For a stage 1 translation, an IA larger than 48 bits requires the implementation of FEAT_LVA. For a stage 2 translation, an IA larger than 48 bits requires the implementation of FEAT_LPA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

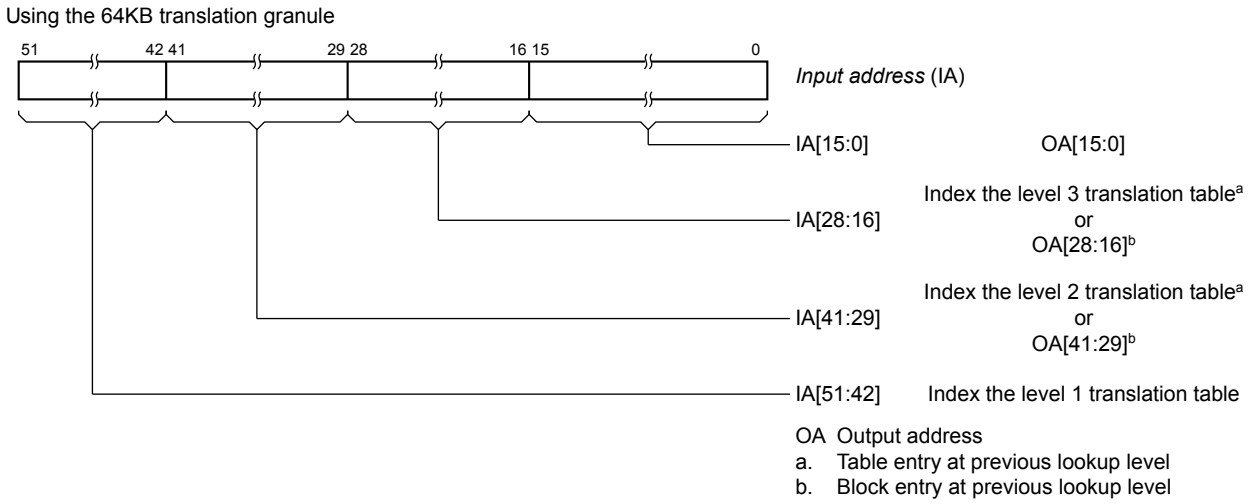

Figure D8-9 52-bit IA resolved using 64KB translation granule

D8.2.10.1 VMSAv8-64 Stage 1 address translation using the 64KB translation granule

R_{FMBKV} For the 64KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.T*n*SZ field as shown in the following table:

Table D8-37 64KB granule, determining stage 1 initial lookup level

Initial lookup level	T*n*SZ minimum value	Maximum IA bits resolved	T*n*SZ maximum value	Minimum IA bits resolved	Additional requirements
1	12	IA[51:16]	15	IA[48:16]	FEAT_LVA is implemented
1	16	IA[47:16]	21	IA[42:16]	-
2	22	IA[41:16]	34	IA[29:16]	-
3	35	IA[28:16]	39	IA[24:16]	-
3	40	IA[23:16]	47	IA[16:16]	FEAT_TTST is implemented

I_{FQFFW} For a stage 1 translation in the 64KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 13 bits and all remaining lookup levels are indexed by exactly 13 bits.

I_{MRDWZ} For the 64KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA larger than 48 bits requires implementation of FEAT_LVA, and an OA larger than 48 bits requires implementation of FEAT_LPA. The 4TB block size requires implementation of FEAT_LPA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

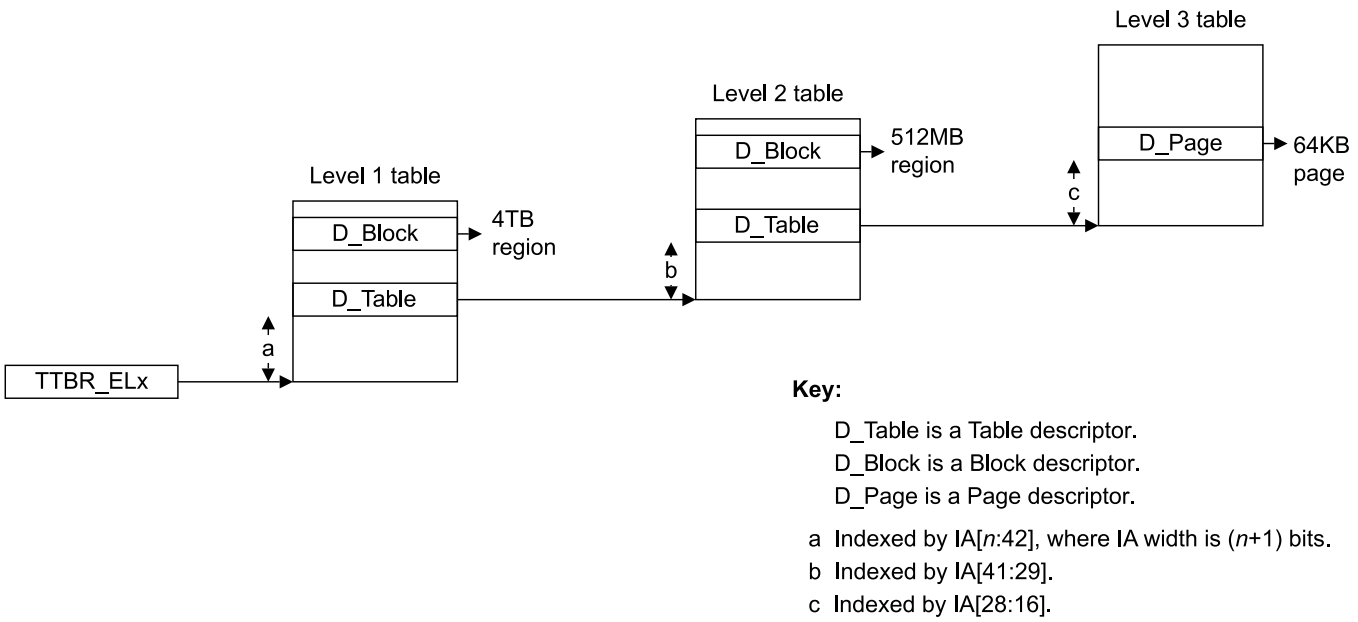

Figure D8-10 Generalized view of a stage 1 address translation using the 64KB granule

D8.2.10.2 VMSAv8-64 Stage 2 address translation using the 64KB translation granule

I_{HHFJP} For the 64KB translation granule, when a stage 2 translation table walk is started, the initial lookup level is determined by the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.

I_{VLBLQ} For the 64KB translation granule, when a stage 2 translation table walk is started, the following table shows the initial lookup level determined by SL0:

Table D8-38 64KB granule, determining stage 2 initial lookup level

SL0	Initial lookup level
`0b00`	Level 3
`0b01`	Level 2
`0b10`	Level 1
`0b11`	Reserved

R_{QCSFP} For the 64KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-39 64KB granule, stage 2 initial lookup at level 1

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[51:16]-IA[48:16]	12-15	FEAT_LPA is implemented
None (1 table)	IA[47:16]-IA[42:16]	16-21	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RWHSBD For the 64KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 2, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-40 64KB granule, stage 2 initial lookup at level 2

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[41:16]-IA[29:16]	22-34	-
2	IA[42:16]	21	-
4	IA[43:16]	20	-
8	IA[44:16]	19	-
16	IA[45:16]	18	-

RPLLHG For the 64KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 3, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

Table D8-41 64KB granule, stage 2 initial lookup at level 3

Number of concatenated translation tables	IA bits resolved	T0SZ	Additional requirements
None (1 table)	IA[23:16]-IA[16:16]	40-47	FEAT_TTST is implemented
None (1 table)	IA[28:16]-IA[24:16]	35-39	-
2	IA[29:16]	34	-
4	IA[30:16]	33	-
8	IA[31:16]	32	-
16	IA[32:16]	31	-

ISYRCT For the 64KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 2 address translation. An IA larger than 48 bits requires the implementation of FEAT_LPA. The 4TB block size requires implementation of FEAT_LPA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

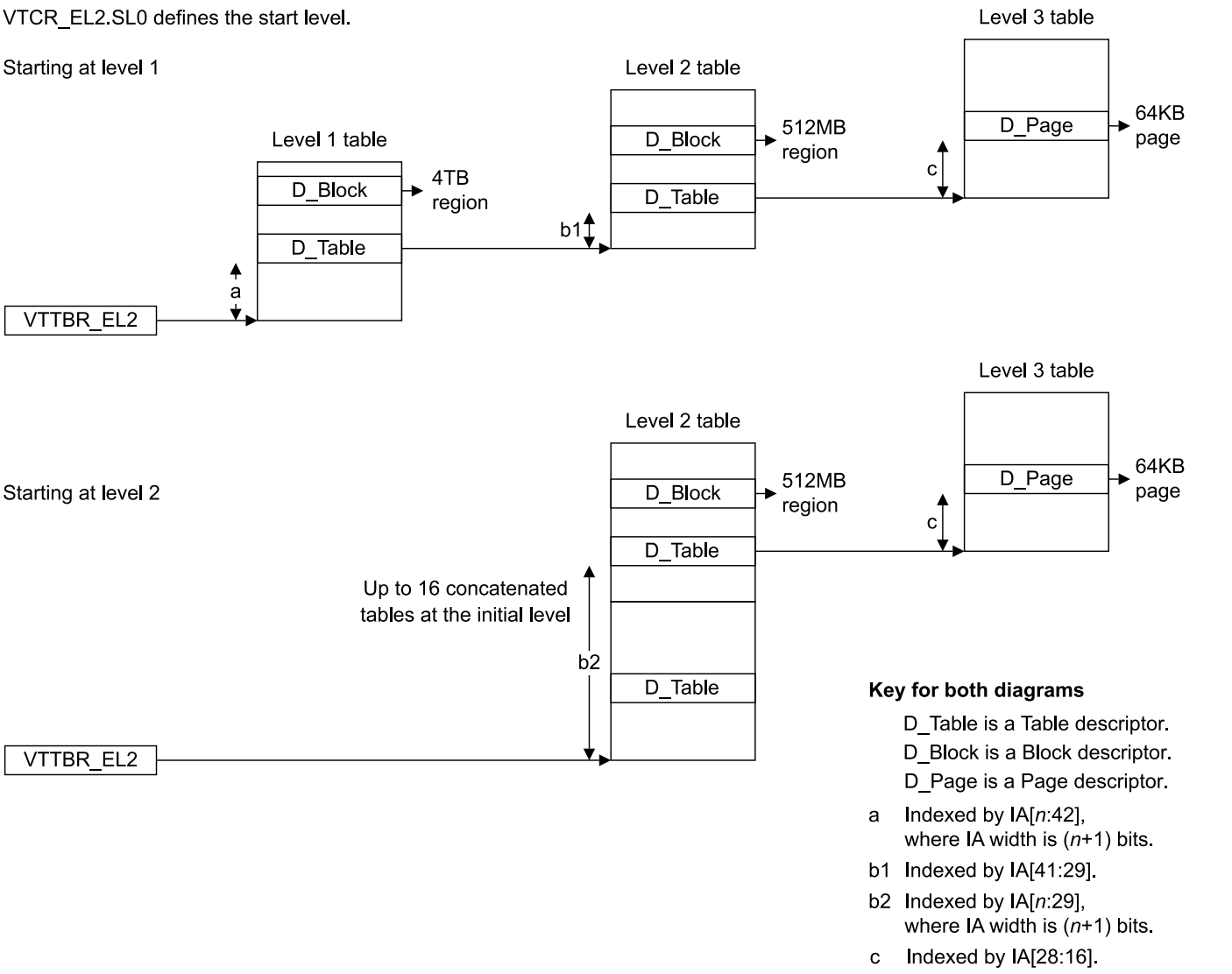

Figure D8-11 Generalized view of a stage 2 address translation using the 64KB granule

R_{SGCBS} For the 64KB translation granule, when a stage 2 translation table walk is started and one of the following is true, a stage 2 level 0 Translation fault is generated:

- The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.
- VTCR_EL2.SL0 or VSTCR_EL2.SL0 is programmed to a reserved value.

I_{DKRJG} For more information, see Concatenated translation tables.

D8.2.10.3 Finding the descriptor when using the VMSAv8-64 64KB translation granule

R_{LMDCR} For the 64KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using of all of the following information:

- The translation table base address, *BaseAddr*.
- The number of bits in the supported PA size, *PAsize*.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.T*n*SZ.
- For a stage 2 translation, all of the following:
  - The value of VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The value of VTCR_EL2.SL0 or VSTCR_EL2.SL0.

Table D8-42 64KB granule, finding the descriptor address

Lookup level	Stage 1 translation table descriptor address	Stage 2 translation table descriptor address
1	if 12 <= T*n*SZ <= 21 then *x* = (25 - T*n*SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+38:42]:`0b000`	if SL0==2 then if 12 <= T0SZ <= 21 then *x* = (25 - T0SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+38:42]:`0b000`
2	if 22 <= T*n*SZ <= 34 then *x* = (38 - T*n*SZ) else *x* = 16 BaseAddr[PAsize-1:*x*]:IA[*x*+25:29]:`0b000`	if SL0==1 then if 18 <= T0SZ <= 34 then *x* = (38 - T0SZ) else *x* = 16 BaseAddr[PAsize-1:*x*]:IA[*x*+25:29]:`0b000`
3	if 35 <= T*n*SZ <= 47 then *x* = (51 - T*n*SZ) else *x* = 16 BaseAddr[PAsize-1:*x*]:IA[*x*+12:16]:`0b000`	if SL0==0 then if 31 <= T0SZ <= 47 then *x* = (51 - T0SZ) else *x* = 16 BaseAddr[PAsize-1:*x*]:IA[*x*+12:16]:`0b000`

D8.2.11 Translation using the VMSAv9-128 translation system

R_{TZZMT} All statements in this section and subsections require a translation stage use the VMSAv9-128 translation system.

R_{KWKRB} For a translation granule used in the VMSAv9-128 translation system, all of the following properties apply:

- The *Stride* is the minimum number of bits used to index a lookup level other than the initial lookup level.
- The minimum number of translation table entries at lookup levels other than the initial lookup level is 2^{Stride}.
- The number of address bits resolved by a Page descriptor at lookup level 3, *PageIndex*, is `Stride+4`.

Those properties are shown in the following table.

Table D8-43 Translation granule properties at lookup levels other than the initial lookup level

Translation granule	Minimum number of translation table entries	Stride	PageIndex
4KB	256	8	12
16KB	1024	10	14
64KB	4096	12	16

D8.2.11.1 Starting the VMSAv9-128 translation

R_{BYPBM} The *regular start level* is determined by the translation granule size and the T*n*SZ fields in the corresponding TCR_ELx, as shown in the following table:

Table D8-44 Regular start levels in the VMSAv9-128 translation system

Translation granule	T*n*SZ maximum value	T*n*SZ minimum value	Regular start level
4KB	11	8	-2
	19	12	-1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Translation granule	T*n*SZ maximum value	T*n*SZ minimum value	Regular start level
	27	20	0
	35	28	1
	43	36	2
	48	44	3
16KB	9	8	-1
	19	10	0
	29	20	1
	39	30	2
	48	40	3
64KB	11	8	0
	23	12	1
	35	24	2
	47	36	3

R_{DBQZN} When a translation table walk begins, the *start level* is determined by adding the value of the TTBR_ELx.SKL field to the regular start level.

R_{FYTMS} If the start level is greater than 3, a level 0 Translation fault is generated.

Note

This can be due to misprogramming the TTBR_ELx.SKL and TCR_ELx.{TG*n*, T*n*SZ} fields.

I_{JHGNJ} When stage 2 uses the VMSAv9-128 translation system, the VTCR_EL2.{SL2, SL0} and VSTCR_EL2.{SL2, SL0} fields are RES0.

I_{DHBWG} For a translation granule, the properties of the start level translation table and the start level address are calculated using of all of the following information:

- The translation table base address, *BaseAddr*, in TTBR_ELx.BADDR.
- The IA supplied to the translation and used as an index into the start level translation table.
- The permitted range of values for TCR_ELx.T*n*SZ.
- The translation granule Stride value.
- The translation table base address register SKL value.

R_{JXBPC} The number of address bits resolved by the table walk process, *X*, is `64-TnSZ-PageIndex`.

R_{ZZDZK} The number of address bits resolved at all lookup levels other than the start level, *Y*, is `(3-start level)*Stride`.

R_{CNZNY} The number of address bits resolved at the start level, *Z*, is `X-Y`.

R_{YQVMK} The size of the start level translation table is 2^Z`x16` bytes.

R_{TBNJK} The start level address used to fetch the first descriptor is determined by appending BaseAddr to an index determined from the IA supplied to the translation stage, as follows:

- `Start level address = BaseAddr[55:Z+4]:IA[64-TnSZ-1:64-TnSZ-Z]:0b0000`.

R_{PMSLB} If the high order start level address bits greater than the OA size are not 0, then an Address size fault is generated at level 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.11.2 Continuing the VMSAv9-128 translation

R_{DZXSB} When a Table descriptor is returned at the current lookup level and does not generate an MMU fault, the translation table walk continues to the *next lookup level*.

R_{RCJXL} The number of address bits resolved by all remaining lookup levels, *K*, is `(3-current lookup level)*Stride`.

R_{CWNTJ} The next lookup level is `current lookup level + 1 + Table descriptor SKL field`.

R_{FSZKG} The number of address bits resolved at the next lookup level, *N*, is `(SKL+1)*Stride`.

R_{ZRZLX} The size of the next level translation table is 2^N`x16` bytes, determined according to the following table:

Table D8-45 Next level translation table sizes

Translation granule	Descriptor SKL field	Translation table size
4KB	0	4KB
	1	1MB
	2	256MB
	3	64GB
16KB	0	16KB
	1	16MB
	2	16GB
	3	Invalid descriptor
64KB	0	64KB
	1	256MB
	2	1TB
	3	Invalid descriptor

Note

For each translation granule, the list of translation table sizes in Table D8-45 is the same as the list of supported block and page sizes in Table D8-46, although the table sizes are not limited to specific lookup levels.

R_{QYWCB} If the size of the next level table is greater than the permitted translation table size, then a Translation fault at the current lookup level is generated. See Table D8-46.

R_{KMLSH} The next level address used to fetch the next descriptor is determined by appending the next level table address to an index determined from the IA supplied to the translation stage, as follows:

- `Next level address = Next level table address[55:N+4]:IA[K+PageIndex-1:K+PageIndex-N]:0b0000`.

R_{RNCRQ} If the high order next level address bits greater than the OA size are not 0, then an Address size fault is generated at the current lookup level.

D8.2.11.3 Completing the VMSAv9-128 translation

R_{WXRFL} When a Block or Page descriptor is returned at the current lookup level, the translation table walk completes and the current lookup level is the *final lookup level*.

R_{MYHXN} For the VMSAv9-128 translation system, a translation resolved by a Block or Page descriptor has all of the properties shown in the following table. In this table, all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- OA base (OAB) is the OA supplied by the descriptor at the final lookup level.
- IA is the IA supplied to the translation stage.
- Final address is the OA supplied by the translation stage.

Table D8-46 Block and Page descriptor properties in the VMSAv9-128 translation system

Translation Granule	Lookup level	Supported block or page size	Final address
4KB	0	64GB block	OAB[55:36]:IA[35:0]
	1	256MB block	OAB[55:28]:IA[27:0]
	2	1MB block	OAB[55:20]:IA[19:0]
	3	4KB page	OAB[55:12]:IA[11:0]
16KB	1	16GB block	OAB[55:34]:IA[33:0]
	2	16MB block	OAB[55:24]:IA[23:0]
	3	16KB page	OAB[55:14]:IA[13:0]
64KB	1	1TB block	OAB[55:40]:IA[39:0]
	2	256MB block	OAB[55:28]:IA[27:0]
	3	64KB page	OAB[55:16]:IA[15:0]

R_{BFVYY} For the final address supplied by a translation stage, if the address bits above the OA size are not set to zero, then an Address size fault is generated at the current lookup level.

D8.2.12 The effects of disabling an address translation stage

I_{HBZJC} Stage 1 and stage 2 translations can be disabled independently, and doing so affects the MMU behavior.

D8.2.12.1 Behavior when stage 1 address translation is disabled

R_{PPDBS} If stage 1 address translation is disabled, then all of the following apply to memory accesses that would otherwise be translated at stage 1:

- The stage 1 IA is flat mapped to the OA.
- No Translation faults, Access flag faults, or Permission faults can be generated.
- Address size faults and Alignment faults can be generated.
- No memory is guarded.
- The access is to one of the following IPA or PA spaces:
  - For accesses from Non-secure state, the access is to Non-secure IPA or PA space.
  - For accesses from Secure state, the access is to Secure IPA or PA space.
  - For accesses from Realm state, the access is to Realm IPA or PA space.
  - For accesses from Root state, the access is to Root PA space.

R_{WFZPW} If stage 1 address translation is disabled, then the stage 1 translation assigns one of the following attributes to memory accesses:

- For memory accesses using the EL1&0 regime, if the effective value of HCR_EL2.DC is 1, then all of the following memory attributes are assigned:
  - The Tagged attribute is set according to HCR_EL2.DCT.
  - Normal Inner Write-Back Cacheable Read-Allocate Write-Allocate.
  - Normal Outer Write-Back Cacheable Read-Allocate Write-Allocate.
  - Non-shareable.
  - If FEAT_XS is implemented, then the XS attribute is set to 0.
- For all other memory accesses, one of the following memory attributes are assigned:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a data access, the Device-nGnRnE memory attribute.
- For an instruction access, the Normal memory attribute and one of the following:
  - If SCTLR_ELx.I is 0, then the Non-cacheable and Outer Shareable attributes.
  - If SCTLR_ELx.I is 1, then the Cacheable, Inner Write-Through Read-Allocate No Write-Allocate, Outer Write-Through Read-Allocate No Write-Allocate Outer Shareable attribute.
- For data accesses and instruction accesses, if FEAT_XS is implemented, then the XS attribute is set to 1.

R_{CSDNK} If all of the following apply, then the stage 1 memory attribute assignments and OA can be modified by the stage 2 translation:

- EL1&0 stage 2 address translation is enabled.
- An access using the EL1&0 translation regime occurs.

R_{LBPTY} If HCR_EL2.DC is 1, then all of the following apply:

- For all purposes other than reading the value of the bit, the Effective value of SCTLR_EL1.M is 0, disabling stage 1 address translation in the EL1&0 translation regime.
- If EL2 is enabled, then for all purposes other than reading the value of the bit, the Effective value of HCR_EL2.VM is 1, enabling stage 2 address translation in the EL1&0 translation regime.

D8.2.12.2 Behavior when stage 2 address translation is disabled

R_{QVLSD} If stage 2 address translation is disabled, then all of the following apply to memory accesses that would otherwise be translated at stage 2:

- No stage 2 MMU faults can be generated.
- The IPA is flat mapped to the PA.
- The memory attributes and permissions assigned by the stage 1 translation are assigned to the PA.
- The access is to one of the following PA spaces:
  - For accesses from the Non-secure IPA space, the access is to the Non-secure PA space.
  - For accesses from the Secure IPA space, the access is to the Secure PA space.
  - For accesses from the Realm IPA space, the access is to the Realm PA space.

D8.2.12.3 Instruction fetch behavior when all translation stages are disabled

I_{DLNTB} If all associated address translation stages are disabled, then software is required to place instructions that will be executed in memory regions where those regions and the memory regions immediately following each have all the following properties:

- The memory region size is equal to the minimum implemented translation granule size.
- The memory region is tolerant to speculative accesses.
- The memory region is naturally aligned.

R_{WLVZN} If execution is in AArch64 state and all associated address translation stages are disabled, a memory location might be accessed as a result of an instruction fetch, including a speculative instruction fetch, in all of the following cases:

- The memory location is in the same memory region, or in the next contiguous memory region, as an instruction that simple sequential program execution either requires to be fetched now or has required to be fetched since the last reset.
- The memory location is the target of a direct branch that simple sequential program execution would have taken since the most recent of one of the following:
  - The last reset.
  - The last synchronization of instruction cache maintenance targeting the branch instruction address.

I_{FZQCN} If all address translation stages are disabled, then speculative instruction fetches can cause unintended memory location accesses, regardless of whether the fetched instruction is committed to execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.12.4 Effect of disabling address translation on maintenance and address translation instructions

R_{SJBMV} Cache maintenance instructions act on the target location regardless of all of the following:

- Whether or not any address translation stages are disabled.
- The memory attribute values.

R_{ZSSPZ} For an address translation stage that is disabled, cache maintenance instructions use flat address mapping.

R_{FZYGN} If an address translation stage is disabled, TLB maintenance operations are not affected.

I_{HRNPD} For more information, see:

- A64 Cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.

D8.2.13 Address translation instructions

I_{TMFCK} Address translation instructions return the result of translating an IA using a specified translation stage or regime.

I_{BDZFF} An address translation instruction has all of the following properties:

- An IA is supplied as the argument to the instruction.
- The instruction encoding determines the translation stage and regime used by the translation.
- The PAR_EL1 register is updated with the translation result.
- For the security state of the instruction as selected by SCR_EL3.{NSE, NS}, the architecture guarantees all of the following:
  - If executed in Non-secure state, then no result is returned from a Secure, Realm, or Root address translation stage.
  - If executed in Secure state, then no result is returned from a Realm or Root address translation stage.
  - If executed in Realm state, then no result is returned from a Secure or Root address translation stage.
- If an address translation stage is controlled by a higher Exception level than the Exception level at which the address translation instruction is executed, then the instruction is UNDEFINED.

I_{CSYQZ} If FEAT_MTE2 is enabled, then the results of `AT*` instructions reflect whether the translation is Tagged or Untagged. For more information, see Memory region tagging types.

I_{TSJPJ} The A64 assembly language syntax of an address translation instruction is `AT <operation>, <Xt>`.

I_{PRNLQ} The `<operation>` in `AT <operation>, <Xt>` is one of the following:

- `S1E1R`, `S1E1W`, `S1E0R`, `S1E0W`, `S12E1R`, `S12E1W`, `S12E0R`, `S12E0W`, `S1E2R`, `S1E2W`, `S1E3R`, or `S1E3W`.
- If FEAT_PAN2 is implemented, `then S1E1WP` or `S1E1RP`.
- If FEAT_ATS1A is implemented, `then S1E1A`, S1E2A, or `S1E3A`.

I_{CQWJY} The `<operation>` in `AT <operation>, <Xt>` has a structure of `<stages><el>(<read|write>{<pan>})|<ignore>` with all of the following components:

- `<stages>` is one of the following address translation stages:
  - `S1` specifies a stage 1 translation.
  - `S12` specifies a stage 1 translation followed by a stage 2 translation.
- `<el>` is one of the following Exception levels that apply to the translation:
  - `E0` specifies EL0.
  - `E1` specifies EL1.
  - `E2` specifies EL2.
  - `E3` specifies EL3.
- `<read|write>` is one of the following:
  - `R` specifies Read.
  - `W` specifies Write.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_PAN2 is implemented, then the `<pan>` component is included and has all of the following properties:
  - `P` determines whether the PSTATE.PAN value is considered when determining permissions. For more information, see PSTATE.PAN.
  - The field is optional.
  - If `<stages>=S1` and `<el>=E1`, the field is permitted.
  - If `<stages>!=S1` or `<el>!=E1`, the field is not permitted.
- If FEAT_ATS1A is implemented, then the `<ignore>` component is included and has all of the following properties:
  - `A` specifies that the permission checks are ignored.
  - The field is optional.
  - If this field is specified, the `<read|write>` and `<pan>` fields are not permitted.

I_{XLYWF} If `<el>` is higher than the current Exception level, then the address translation instruction is UNDEFINED.

I_{SYHWD} The `<Xt>` in `AT <operation>, <Xt>` specifies the 64-bit register holding the address to be translated.

I_{GSXHM} When an address translation instruction applies to a translation regime that is using AArch32, VA[63:32] is RES0.

R_{NJGLC} When an address is translated by an address translation instruction, no alignment restrictions exist and therefore an address translation instruction cannot generate an Alignment fault.

I_{LRNWF} Execution of an AT instruction may result in a successful translation reported in PAR_EL1, an MMU fault being reported in PAR_EL1, or an MMU fault generating an exception.
For more information see:
- MMU faults generated by address translation instructions
- Address translation instructions, successful address translation

R_{NWDPJ} When an address translation instruction is executed, the specified translation stage and regime determines all of the following:
- The entries in TLB caching structures that are permitted to be used.
- How the translation table walk is done.

R_{NPZWX} When an address translation instruction is executed, it is IMPLEMENTATION DEFINED whether the result can be returned from a TLB, or a translation table walk occurs.

I_{WGXJK} If TLB entries might differ from the underlying translation tables held in memory, such as when waiting for a maintenance or synchronization event to complete after updating the translation tables, Arm recommends not using the address translation instructions.

D8.2.13.1 Address translation instructions, successful address translation

R_{LRYRY} When an address translation instruction successfully translates an address, all of the following are updated in PAR_EL1:
- PAR_EL1.F is set to 0.
- The resulting OA is returned in PAR_EL1.PA, and the resulting attributes are returned in the other fields of PAR_EL1.

I_{PVCPF} When populating PAR_EL1 with the result of an address translation instruction, the fetches of stage 1 or stage 2 descriptors use the appropriate MECID register values for the translation regime.

I_{QKWHX} The architecture defines a single PAR, PAR_EL1, that is used regardless of all of the following:
- The Exception level at which the address translation instruction was executed.
- The Exception level that controls the address translation stage or stages used by the address translation instruction.

R_{YLPDG} If single-stage address translation instructions, `ATS1*`, target the VMSAv9-128 translation system, then they report their result using the 128-bit format of PAR_EL1.

R_{XVCRF} If one of the following is true, then `ATS12*` instructions report their result using the 128-bit format of PAR_EL1:
- Stage 2 of the target translation system is enabled and the stage 2 translation system uses VMSAv9-128.
- Stage 2 of the target translation system is disabled and the stage 1 translation system uses VMSAv9-128.

Otherwise, the 64-bit format of PAR_EL1 is used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.13.2 Address translation instructions, effect of translation regime

R_{FKTKP} If EL2 is disabled or not implemented, the AT S1E2R, AT S1E2W, and AT S1E2A instructions are UNDEFINED.

R_{NYXTL} If EL3 is implemented, then when an address translation instruction that applies to an Exception level lower than EL3 is executed, the Effective value of SCR_EL3.{NSE, NS} determines the target Security state that the instruction applies to, as shown in the following table:

NSE, NS	Security State
0, 0	Secure
0, 1	Non-secure
1, 0	Instruction is UNDEFINED
1, 1	Realm

R_{RKBPZ} If EL3 is implemented and EL2 is disabled or not implemented for the target Security state, then all of the following apply to the `AT S12E**` instruction behavior:

- The instruction has the same behavior as the equivalent `AT S1E**` instruction.
- The instruction behaves consistently with an implementation that has all of the following characteristics:
  - EL2 is implemented.
  - Stage 2 translation is disabled.

R_{PXNSR} If the Effective value of SCR_EL3.{NSE, NS} is {0, 0} and Secure EL2 is disabled or not implemented, then executing the following AT instructions at EL3 is UNDEFINED:

- AT S1E2R.
- AT S1E2W.
- AT S1E2A.

I_{KJDTT} The value of HCR_EL2.DC affects all of the following instructions:

- `AT S1E0*`.
- `AT S1E1*`.
- `AT S12E0*`.
- `AT S12E1*`.

D8.2.13.3 Address translation instructions, synchronization requirements

I_{JTYVP} When an address translation instruction is executed, explicit synchronization is required to guarantee the result is visible to subsequent direct reads of PAR_EL1.

For more information, see Synchronization requirements for AArch64 System registers.

I_{PCPWC} Requiring explicit synchronization after executing an address translation instruction is consistent with the general requirement that the effect of a write to a System register requires explicit synchronization before the result is guaranteed to be visible to subsequent instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.3 Translation table descriptor formats

R_{HJXKZ} If an address translation stage is controlled by an Exception level that is using AArch64, then the translation table uses one of the following:

- For the VMSAv8-64 translation system, 64-bit descriptors.
- For the VMSAv9-128 translation system, 128-bit descriptors.

R_{JXLYN} A translation descriptor has one of the following formats:

- An invalid descriptor format.
- A Table descriptor format that points to the next-level translation table.
- A Block descriptor or Page descriptor format that defines the memory access properties.
- A reserved format.

R_{KHVQT} The value of descriptor bit[0] determines one of the following:

- If descriptor bit[0] is 0, then the descriptor is invalid.
- If descriptor bit[0] is 1, then the descriptor is valid.

R_{RWMFF} For 64-bit descriptors used by the VMSAv8-64 translation system, if descriptor bit[0] is 1, then the descriptor type is determined by one of the following values of descriptor bit[1]:

- For lookup levels less than lookup level 3, one of the following:
  - If bit[1] is 0, then the descriptor is a Block descriptor.
  - If bit[1] is 1, then the descriptor is a Table descriptor.
- For lookup level 3, one of the following:
  - If bit[1] is 0, then the descriptor is reserved, and treated as invalid.
  - If bit[1] is 1, then the descriptor is a Page descriptor.

Table D8-48 Determination of descriptor type, VMSAv8-64 translation system

Descriptor bit[0]	bit[1]	Descriptor type	Condition
0	-	Invalid	-
1	0	Block	Lookup level is not 3
		Reserved, treated as invalid	Lookup level is 3
	1	Table	Lookup level is not 3
		Page	Lookup level is 3

R_{BPTPZ} For 128-bit descriptors used by the VMSAv9-128 translation system, if descriptor bit[0] is 1, then the descriptor type is determined by the sum of the current lookup level and the descriptor SKL field in descriptor bits[110:109], as follows:

- If the sum is less than 3, then one of the following:
  - If a table of the resulting size is permitted, then a Table descriptor.
  - If a table of the resulting size is not permitted, then an Invalid descriptor.
- If the sum is equal to 3, then one of the following:
  - If a block of the resulting size is permitted at the resulting level, then a Block or Page descriptor.
  - If a block of the resulting size is not permitted at the resulting level, then an Invalid descriptor.
- If the sum is greater than 3, then an Invalid descriptor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-49 Determination of descriptor type, VMSAv9-128 translation system

Descriptor bit[0]	Current lookup level + descriptor SKL field	Descriptor type	Condition
0	-	Invalid	-
1	Less than 3	Table	Table size is permitted
		Invalid	Table size is not permitted
	Equal to 3	Block or Page	Block size is permitted
		Invalid	Block size is not permitted
	Greater than 3	Invalid	-

R_{CQGPJ} For Table descriptors, if the Effective value of the SKL field is 0, then the next-level table address is one of the following:

- For a level -2 Table descriptor, the base address of a level -1 table.
- For a level -1 Table descriptor, the base address of a level 0 table.
- For a level 0 Table descriptor, the base address of a level 1 table.
- For a level 1 Table descriptor, the base address of a level 2 table.
- For a level 2 Table descriptor, the base address of a level 3 table.

R_{FRLRB} For Table descriptors, the next-level table address is one of the following:

- For the EL1&0 stage 1 translation regimes, if EL2 is enabled, then the IPA of the target table.
- For all other translation regimes, the PA of the target table.

D8.3.1 VMSAv8-64 descriptor formats

D8.3.1.1 VMSAv8-64 Table descriptor format

R_{PGHRP} Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

I_{TWKLX} The following figure shows all of following stage 1 and stage 2 VMSAv8-64 Table descriptor formats:

- 4KB, 16KB, and 64KB granules using a 48-bit next-level table address.
- If the Effective value of TCR_ELx.DS is 1, the 4KB and 16KB granules using a 52-bit next-level table address.
- If FEAT_LPA is implemented, the 64KB granule using a 52-bit next-level table address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

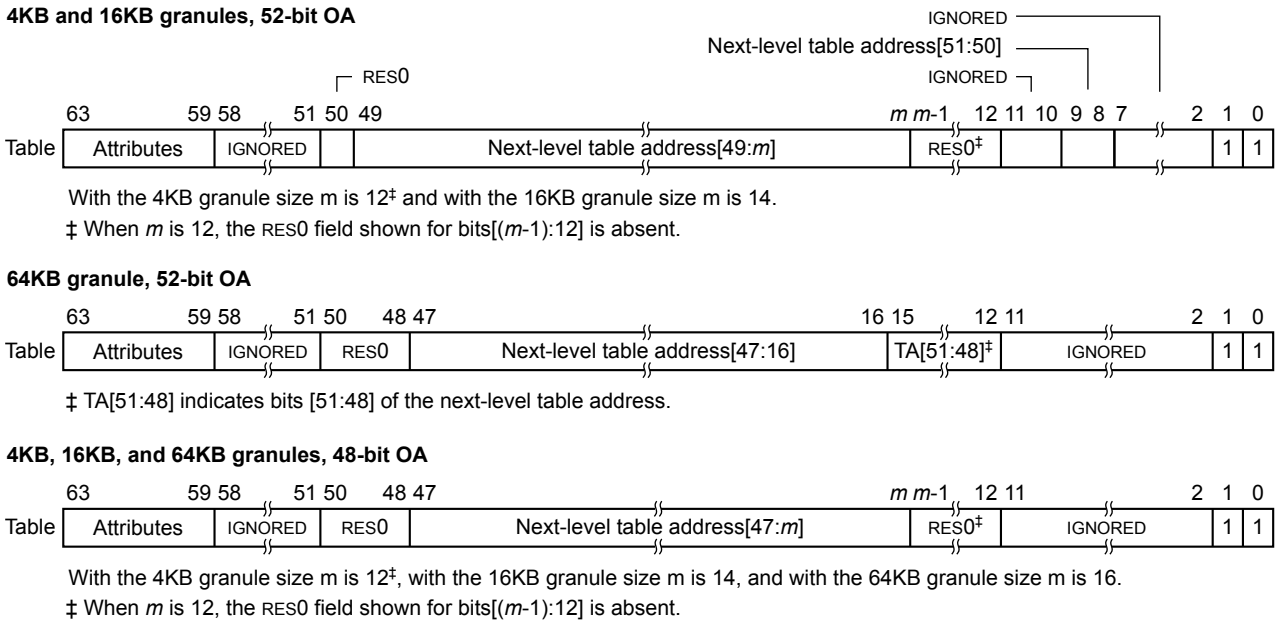

Figure D8-12 VMSAv8-64 Table descriptor formats

ILZSBS For a stage 1 translation, the following figure shows the next-level attribute fields in a VMSAv8-64 Table descriptor. For a stage 2 translation, all of the following apply:

- Bit[63] is RES0.
- Bits[62:59] are one of the following:
  - — If stage 2 Indirect permissions are disabled, then RES0.
  - — If stage 2 Indirect permissions are enabled, then IGNORED.
- Bits[58:51] are IGNORED.

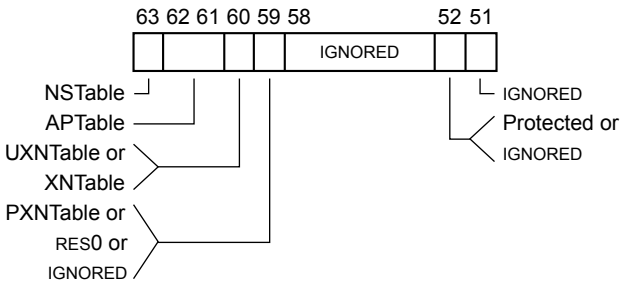

Figure D8-13 Stage 1 next-level attribute fields in a VMSAv8-64 Table descriptor

RYWCCT For stage 1 translations, the following table defines the fields in the VMSAv8-64 Table descriptor. In this table, NLTA is the next level table address.

Table D8-50 Stage 1 VMSAv8-64 Table descriptor fields

Bit position	Field	Condition
[63]	RES0	The Security state is not Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	NSTable	Secure state. See Hierarchical control of Secure or Non-secure memory accesses.
[62:61]	APTable[1:0]	Hierarchical permissions are enabled. See Hierarchical control of data access Direct permissions. For EL1&0 translations, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then APTable[0] is treated as 0 regardless of the actual value. See Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1.
[60]	XNTable	Hierarchical permissions are enabled and the translation regime supports a single privilege level. See Hierarchical control of instruction execution for Direct permissions.
	UXNTable	Hierarchical permissions are enabled and the translation regime supports two privilege levels. See Hierarchical control of instruction execution for Direct permissions.
	PXNTable	Hierarchical permissions are enabled, the EL1&0 translation regime, and the Effective value of HCR_EL2.{NV, NV1} is {1, 1}. See Hierarchical control of instruction execution for Direct permissions.
[59]	RES0	Hierarchical permissions are enabled and one of the following applies: • The translation regime supports a single privilege level. • The EL1&0 translation regime and the Effective value of HCR_EL2.{NV, NV1} is {1, 1}. See Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1.
	PXNTable	Hierarchical permissions are enabled and the translation regime supports two privilege levels. See Hierarchical control of instruction execution for Direct permissions.
[58:53]	IGNORED	-
[52]	IGNORED	The Effective value of PnCH is 0.
	Protected	The Effective value of PnCH is 1. See Stage 1 Protected Attribute.
[51]	IGNORED	-
[50]	RES0	-
[49:48]	RES0	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0, or the 64KB translation granule is used.
	NLTA[49:48]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
[47:12]	NLTA[47:12]	The 4KB translation granule is used.
	NLTA[47:14]	The 16KB translation granule is used. Descriptor bits [13:12] are RES0.
	NLTA[47:16]	The 64KB translation granule is used and FEAT_LPA is not implemented. Descriptor bits [15:12] are RES0.
	NLTA[47:16]: NLTA[51:48]	The 64KB translation granule is used and FEAT_LPA is implemented.
[11]	IGNORED	-
[10]	IGNORED	Hardware managed Table descriptor Access flag is not enabled.
	Access flag (AF)	Hardware managed Table descriptor Access flag is enabled. See Hardware management of the Table descriptor Access Flag.
[9:8]	IGNORED	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0, or the 64KB translation granule is used.
	NLTA[51:50]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
[7:2]	IGNORED	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
[1]	Table descriptor	1, for lookup levels less than lookup level 3.
[0]	Valid descriptor	1.

I_{LPCPS} For stage 1 translations in the EL3 translation regime, the removal of NSTable in Root state is a change from the behavior of EL3 in Secure state.

R_{XFNBW} For stage 2 translations, the following table defines the fields in the VMSAv8-64 Table descriptor. In this table, NLTA is the next level table address.

Table D8-51 Stage 2 VMSAv8-64 Table descriptor fields

Bit position	Field	Condition
[63]	RES0	-
[62:59]	RES0	Stage 2 Indirect permissions are disabled.
	IGNORED	Stage 2 Indirect permissions are enabled. See Stage 2 Indirect permissions.
[58:51]	IGNORED	-
[50]	RES0	-
[49:48]	RES0	The 4KB or 16KB translation granule is used, and the Effective value of VTCR_EL2.DS is 0, or the 64KB translation granule is used.
	NLTA[49:48]	The 4KB or 16KB translation granule is used, and the Effective value of VTCR_EL2.DS is 1.
[47:12]	NLTA[47:12]	The 4KB translation granule is used.
	NLTA[47:14]	The 16KB translation granule is used. Descriptor bits [13:12] are RES0.
	NLTA[47:16]	The 64KB translation granule is used and FEAT_LPA is not implemented. Descriptor bits [15:12] are RES0.
	NLTA[47:16]: NLTA[51:48]	The 64KB translation granule is used and FEAT_LPA is implemented.
[11]	IGNORED	-
[10]	IGNORED	Hardware managed Table descriptor Access flag is not enabled.
	Access flag (AF)	Hardware managed Table descriptor Access flag is enabled. See Hardware management of the Table descriptor Access Flag.
[9:8]	IGNORED	The 4KB or 16KB translation granule is used, and the Effective value of VTCR_EL2.DS is 0, or the 64KB translation granule is used.
	NLTA[51:50]	The 4KB or 16KB translation granule is used, and the Effective value of VTCR_EL2.DS is 1.
[7:2]	IGNORED	-
[1]	Table descriptor	1, for lookup levels less than lookup level 3.
[0]	Valid descriptor	1.

D8.3.1.2 VMSAv8-64 Block descriptor and Page descriptor formats

R_{WLDSK} Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

I_{GBPDK} The following figure shows all of the following VMSAv8-64 Block descriptor formats:

- 4KB, 16KB, and 64KB granules using a 48-bit OA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of TCR_ELx.DS is 1, the 4KB and 16KB granules using a 52-bit OA.
- If FEAT_LPA is supported, the 64KB granule using a 52-bit OA.

4KB and 16KB granules, 52-bit OA

64KB granule, 52-bit OA

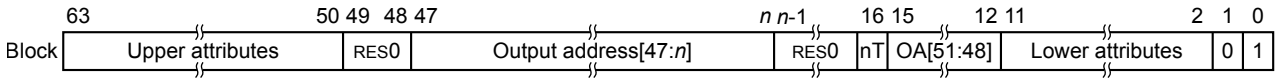

For the 4KB granule size, the level 0 descriptor *n* is 39, the level 1 descriptor *n* is 30, and the level 2 descriptor *n* is 21.

For the 16KB granule size, the level 1 descriptor *n* is 36, and the level 2 descriptor *n* is 25.

For the 64KB granule size, the level 1 descriptor *n* is 42, and the level 2 descriptor *n* is 29.

4KB, 16KB, and 64KB granules, 48-bit OA

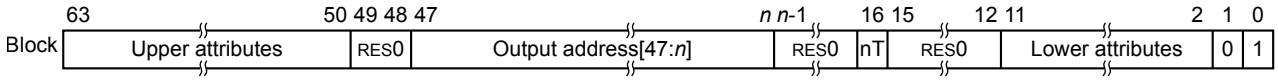

For the 4KB granule size, the level 1 descriptor *n* is 30, and the level 2 descriptor *n* is 21.

For the 16KB granule size, the level 2 descriptor *n* is 25.

For the 64KB granule size, the level 2 descriptor *n* is 29.

Figure D8-14 VMSAv8-64 Block descriptor formats

IWNCLB The following figure shows all of the following VMSAv8-64 Page descriptor formats:

- 4KB, 16KB, and 64KB granules using a 48-bit OA.
- If the Effective value of TCR_ELx.DS is 1, the 4KB and 16KB granules using a 52-bit OA.
- If FEAT_LPA is supported, the 64KB granule using a 52-bit OA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

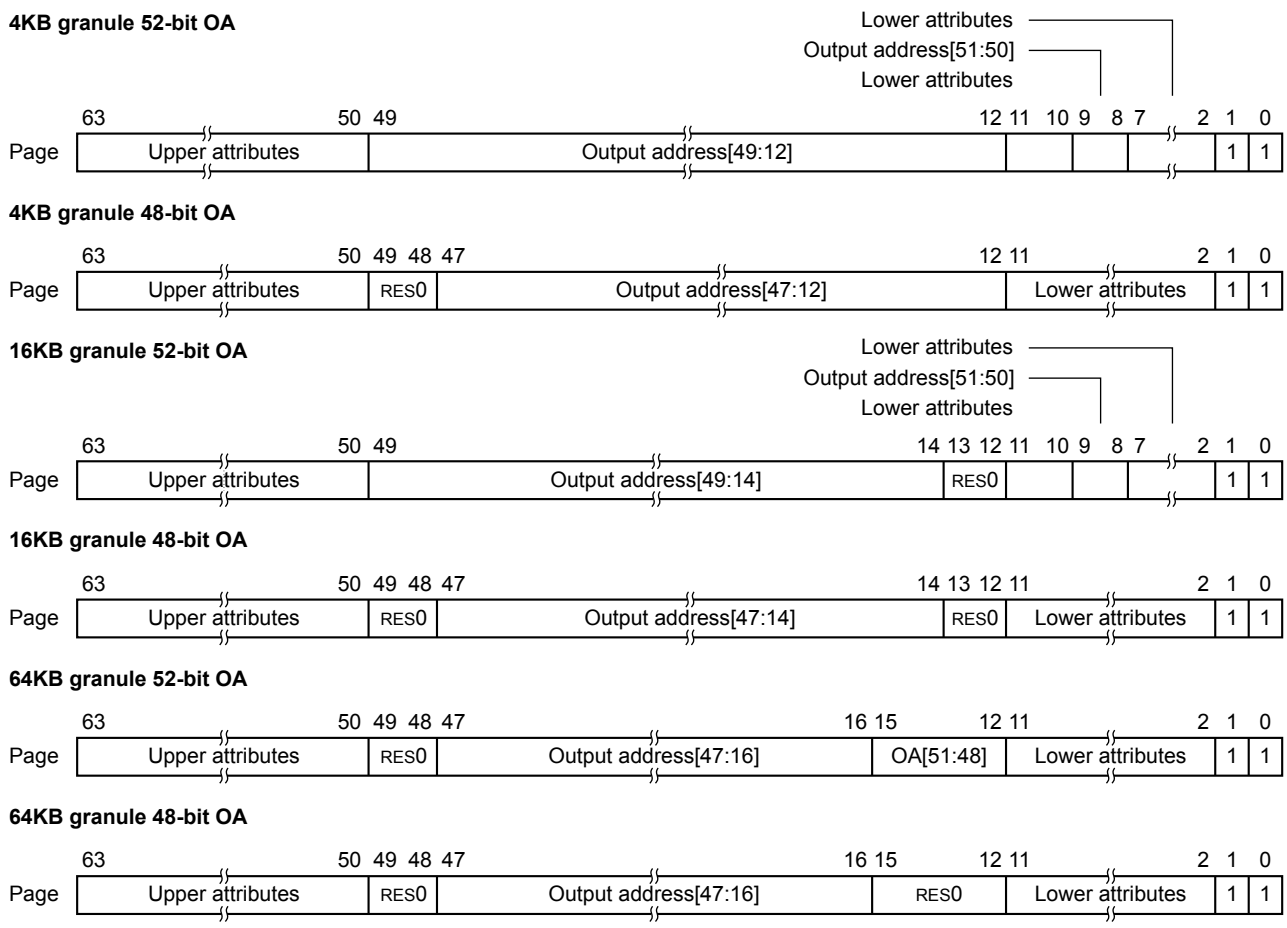

Figure D8-15 VMSAv8-64 Page descriptor formats

IGLMLD For a stage 1 translation, the following figure shows the attribute fields in VMSAv8-64 Block descriptors and Page descriptors, split into upper attributes and lower attributes.

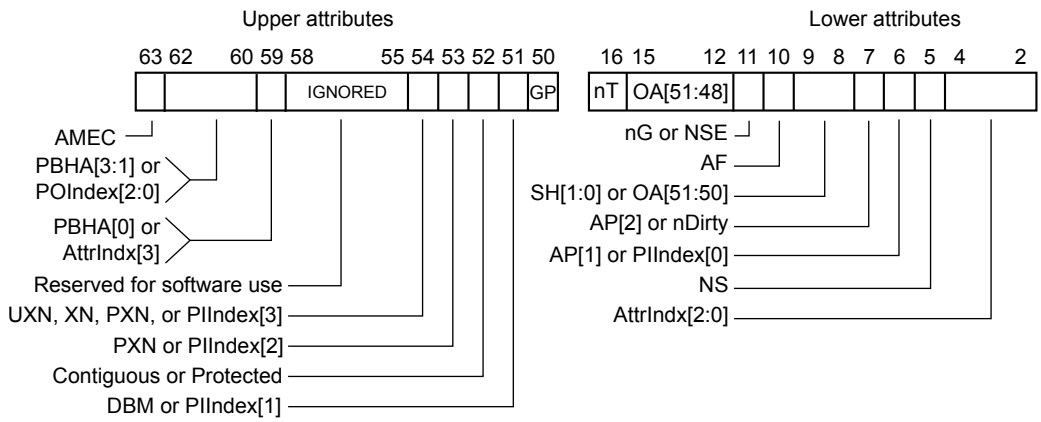

Figure D8-16 Stage 1 attribute fields in VMSAv8-64 Block and Page descriptors

RJJNHR For stage 1 translations, the following table defines the fields in the VMSAv8-64 Block and Page descriptors. In this table, OAB is the OA base that is appended to the IA supplied to the translation stage to produce the final OA supplied by the translation stage.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-52 Stage 1 VMSAv8-64 Block and Page descriptor fields

Bit position	Field	Condition
[63]	IGNORED	Secure state.
	Reserved for software use	Non-secure state, or the Realm EL1&0 translation regime.
	Alternate MECID (AMEC)	Realm EL2 or Realm EL2&0 translation regimes, and the descriptor NS field is 0. See Memory Encryption Contexts extension.
	RES0	Realm EL2 or Realm EL2&0 translation regimes, and the descriptor NS field is 1.
[62:60]	IGNORED	Both of the following apply: • FEAT_HPDS2 is not implemented or PBHA is not enabled by the corresponding TCR_ELx.HWU*nn* control bit. • FEAT_S1POE is not implemented or stage 1 Overlay permissions are not enabled.
	PBHA[3:1]	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
	POIndex[2:0]	Stage 1 Overlay permissions enabled. See Stage 1 Overlay permissions.
[59]	IGNORED	Both of the following apply: • FEAT_HPDS2 is not implemented or PBHA is not enabled by the corresponding TCR_ELx.HWU*nn* control bit. • FEAT_AIE is not implemented or the stage 1 Attribute Index Extension is not enabled.
	PBHA[0]	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
	AttrIndx[3]	FEAT_AIE is enabled. See Stage 1 memory type and Cacheability attributes.
[58:55]	Reserved for software use	-
[54]	Execute-never (XN)	The translation regime supports a single privilege level. See Stage 1 instruction execution using Direct permissions.
	Unprivileged Execute-never (UXN)	The translation regime supports two privilege levels. See Stage 1 instruction execution using Direct permissions.
	Privileged Execute-never (PXN)	The EL1&0 translation regime and the Effective value of HCR_EL2.{NV, NV1} is {1, 1}. The Effective value of UXN is 0. See Stage 1 instruction execution using Direct permissions and Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1.
	PIIndex[3]	Stage 1 Indirect permissions enabled, regardless of other feature settings. See Stage 1 Indirect permissions.
[53]	RES0	One of the following: • The translation regime supports a single privilege level. • The EL1&0 translation regime and the Effective value of HCR_EL2.{NV, NV1} is {1, 1}. See Stage 1 instruction execution using Direct permissions and Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1.
	PXN	The translation regime supports two privilege levels. See Stage 1 instruction execution using Direct permissions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	PIIndex[2]	Stage 1 Indirect permissions enabled, regardless of other feature settings. See Stage 1 Indirect permissions.
[52]	Contiguous	The Effective value of PnCH is 0. See The Contiguous bit.
	Protected	The Effective value of PnCH is 1. See Stage 1 Protected Attribute.
[51]	Dirty bit modifier (DBM)	Stage 1 Indirect permissions are disabled. See Hardware management of the dirty state.
	PIIndex[1]	Stage 1 Indirect permissions are enabled. See Stage 1 Indirect permissions.
[50]	RES0	FEAT_BTI is not implemented.
	Guarded Page (GP)	FEAT_BTI is implemented. See PSTATE.BTYPE.
[49:48]	RES0	One of the following: • The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0. • The 64KB translation granule is used.
	OAB[49:48]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
Block[47:17]	OAB[47:39]	The 4KB translation granule, the Effective value of TCR_ELx.DS is 1, and the descriptor is a level 0 Block descriptor. Descriptor bits [38:17] are RES0.
	OAB[47:30]	The 4KB translation granule and the descriptor is a level 1 Block descriptor. Descriptor bits [29:17] are RES0.
	OAB[47:21]	The 4KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [20:17] are RES0.
	OAB[47:36]	The 16KB translation granule, the Effective value of TCR_ELx.DS is 1, and the descriptor is a level 1 Block descriptor. Descriptor bits [35:17] are RES0.
	OAB[47:25]	The 16KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [24:17] are RES0.
	OAB[47:42]	The 64KB translation granule, FEAT_LPA is implemented, and the descriptor is a level 1 Block descriptor. Descriptor bits [41:17] are RES0.
	OAB[47:29]	The 64KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [28:17] are RES0.
Block[16]	RES0	FEAT_BBML1 is not implemented.
	nT	FEAT_BBML1 is implemented. See Table and Block entry.
Block[15:12]	RES0	One of the following: • FEAT_LPA is not implemented. • FEAT_LPA is implemented, and the 4KB or 16KB translation granule is used.
	OAB[51:48]	FEAT_LPA is implemented and the 64KB translation granule is used.
Page[47:12]	OAB[47:12]	The 4KB translation granule is used.
	OAB[47:14]	The 16KB translation granule is used. Descriptor bits [13:12] are RES0.
	OAB[47:16]	The 64KB translation granule and FEAT_LPA is not implemented. Descriptor bits [15:12] are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	OAB[47:16]: OAB[51:48]	The 64KB translation granule and FEAT_LPA is implemented.
[11]	RES0	The translation regime supports a single privilege level and the Security state is not Root state.
	NSE	Root state. See Controlling memory access Security state.
	Not global (nG)	The translation regime supports two privilege levels. See Global and process-specific translation table entries.
	IGNORED	The Effective value of TCR2_ELx.FNGy is 1, where y is {0, 1}. See R_{KDVGW} in Use of ASIDs and VMIDs to reduce TLB maintenance requirements.
[10]	Access flag (AF)	- See The Access flag.
[9:8]	Shareability (SH[1:0])	One of the following: • The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0. • The 64KB translation granule is used. See Stage 1 Shareability attributes.
	OAB[51:50]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
[7]	AP[2]	Stage 1 Indirect permissions are disabled. See Stage 1 data accesses using Direct permissions.
	nDirty	Stage 1 Indirect permissions are enabled. See The dirty state.
[6]	RES1	Stage 1 Indirect permissions are disabled and the translation regime supports a single privilege level.
	AP[1]	Stage 1 Indirect permissions are disabled and the translation regime supports two privilege levels. See Stage 1 data accesses using Direct permissions.
	PIIndex[0]	Stage 1 Indirect permissions are enabled. See Stage 1 Indirect permissions.
[5]	RES0	The access is from Non-secure state, or from Realm state using the EL1&0 translation regime.
	NS	The access is from Secure state, from Realm state using the EL2 or EL2&0 translation regimes, or from Root state.
[4:2]	AttrIndx[2:0]	See Stage 1 memory type and Cacheability attributes.
[1]	Block descriptor	0, for lookup levels less than lookup level 3.
	Page descriptor	1, for lookup level 3.
[0]	Valid descriptor	1.

I_{BJRZD} For a stage 2 translation, the following figure shows the attribute fields in VMSAv8-64 Block descriptors and Page descriptors, split into upper attributes and lower attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

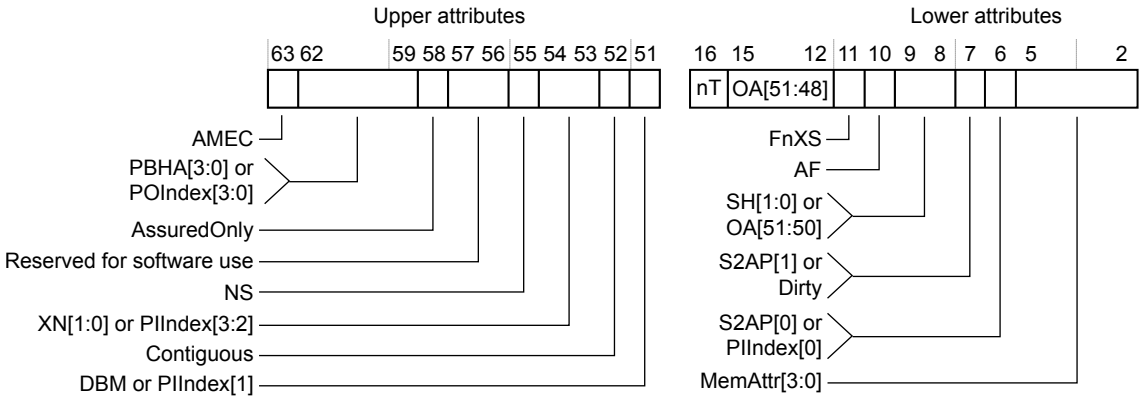

Figure D8-17 Stage 2 attribute fields in VMSAv8-64 Block and Page descriptors

R_{DMBGN} For stage 2 translations, the following table defines the fields in the VMSAv8-64 Block and Page descriptors. In this table, OAB is the OA base that is appended to the IA supplied to the translation stage to produce the final OA supplied by the translation stage.

Table D8-53 Stage 2 VMSAv8-64 Block and Page descriptor fields

Bit position	Field	Condition
[63]	RES0	One of the following: • Non-secure or Secure state. • Realm state and the descriptor NS field is 1.
	Alternate MECID (AMEC)	Realm state and the descriptor NS field is 0. See Memory Encryption Contexts extension.
[62:60]	Reserved for use by a System MMU	One of the following: • FEAT_HPDS2 is not implemented. • FEAT_HPDS2 is implemented and the PBHA bit is not enabled by the corresponding TCR_ELx.HWU*nn* control bit.
	PBHA[3:1]	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
	POIndex[3:1]	Stage 2 Overlay permissions enabled, regardless of other feature settings. See Stage 2 Overlay permissions.
[59]	IGNORED	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
	PBHA[0]	FEAT_HPDS2 is implemented and PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
	POIndex[0]	Stage 2 Overlay permissions enabled, regardless of other feature settings. See Stage 2 Overlay permissions.
[58]	Reserved for software use	VTCR_EL2.AssuredOnly is 0.
	AssuredOnly	VTCR_EL2.AssuredOnly is 1. See Assured translation.
[57:56]	Reserved for software use	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
[55]	Non-secure (NS)	Realm state. See Controlling memory access Security state.
	IGNORED	Security state other than Realm state.
[54]	Execute-never (XN)	FEAT_XNX is not implemented. See Stage 2 instruction execution using Direct permissions.
	XN[1]	FEAT_XNX is implemented. See Stage 2 instruction execution using Direct permissions.
	PIIndex[3]	Stage 2 Indirect permissions enabled, regardless of other feature settings. See Stage 2 Indirect permissions.
[53]	RES0	FEAT_XNX is not implemented.
	XN[0]	FEAT_XNX is implemented. See Stage 2 instruction execution using Direct permissions.
	PIIndex[2]	Stage 2 Indirect permissions enabled, regardless of other feature settings. See Stage 2 Indirect permissions.
[52]	Contiguous	- See The Contiguous bit.
[51]	Dirty bit modifier (DBM)	Stage 2 Indirect permissions are disabled. See Hardware management of the dirty state.
	PIIndex[1]	Stage 2 Indirect permissions are enabled. See Stage 2 Indirect permissions.
[50]	RES0	-
[49:48]	RES0	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0, or the 64KB translation granule.
	OAB[49:48]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
Block[47:17]	OAB[47:39]	The 4KB translation granule, the Effective value of TCR_ELx.DS is 1, and the descriptor is a level 0 Block descriptor. Descriptor bits [38:17] are RES0.
	OAB[47:30]	The 4KB translation granule and the descriptor is a level 1 Block descriptor. Descriptor bits [29:17] are RES0.
	OAB[47:21]	The 4KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [20:17] are RES0.
	OAB[47:36]	The 16KB translation granule, the Effective value of TCR_ELx.DS is 1, and the descriptor is a level 1 Block descriptor. Descriptor bits [35:17] are RES0.
	OAB[47:25]	The 16KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [24:17] are RES0.
	OAB[47:42]	The 64KB translation granule, FEAT_LPA is implemented, and the descriptor is a level 1 Block descriptor. Descriptor bits [41:17] are RES0.
	OAB[47:29]	The 64KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [28:17] are RES0.
Block[16]	RES0	FEAT_BBML1 is not implemented.
	nT	FEAT_BBML1 is implemented. See Table and Block entry.
Block[15:12]	RES0	One of the following: • FEAT_LPA is not implemented. • FEAT_LPA is implemented, and the 4KB or 16KB translation granule is used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	OAB[51:48]	FEAT_LPA is implemented and the 64KB translation granule is used.
Page[47:12]	OAB[47:12]	The 4KB translation granule is used.
	OAB[47:14]	The 16KB translation granule is used. Descriptor bits [13:12] are RES0.
	OAB[47:16]	The 64KB translation granule and FEAT_LPA is not implemented. Descriptor bits [15:12] are RES0.
	OAB[47:16]: OAB[51:48]	The 64KB translation granule and FEAT_LPA is implemented.
[11]	RES0	FEAT_XS is not implemented.
	FnXS	FEAT_XS is implemented. See XS attribute modifier.
[10]	Access flag (AF)	- See The Access flag.
[9:8]	Shareability (SH[1:0])	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0, or the 64KB translation granule. See Stage 2 Shareability attributes.
	OAB[51:50]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
[7]	S2AP[1]	Stage 2 Indirect permissions are disabled. See Stage 2 data accesses using Direct permissions.
	Dirty	Stage 2 Indirect permissions are enabled. See The dirty state.
[6]	S2AP[0]	Stage 2 Indirect permissions are disabled. See Stage 2 data accesses using Direct permissions.
	PIIndex[0]	Stage 2 Indirect permissions are enabled. See Stage 2 Indirect permissions.
[5]	RES0	The Effective value of HCR_EL2.FWB is 1.
	MemAttr[3]	The Effective value of HCR_EL2.FWB is 0. See Stage 2 memory type and Cacheability attributes when FWB is disabled.
[4:2]	MemAttr[2:0]	- See Stage 2 memory type and Cacheability attributes when FWB is disabled and Stage 2 memory type and Cacheability attributes when FWB is enabled.
[1]	Block descriptor	0, for lookup levels less than lookup level 3.
	Page descriptor	1, for lookup level 3.
[0]	Valid descriptor	1.

IGKSKW The VMSAv8-64 Block descriptor and Page descriptor format defines the data Access Permissions bits, AP[2:1], and does not define an AP[0] bit.

D8.3.1.3 VMSAv8-64 Invalid descriptor format

RVRYZC For stage 1 translations, the following table defines the fields in the VMSAv8-64 Invalid descriptor. IGNORED fields are available for software use.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-54 Stage 1 VMSAv8-64 Invalid descriptor fields

Bit position	Field	Condition
[63:53]	IGNORED	-
[52]	IGNORED	The Effective value of PnCH is 0.
	Protected	The Effective value of PnCH is 1. See Stage 1 Protected Attribute.
[51:1]	IGNORED	-
[0]	Invalid descriptor	0.

RJYLCV For stage 2 translations, the following table defines the fields in the VMSAv8-64 Invalid descriptor. IGNORED fields are available for software use.

Table D8-55 Stage 2 VMSAv8-64 Invalid descriptor fields

Bit position	Field	Condition
[63:1]	IGNORED	-
[0]	Invalid descriptor	0.

D8.3.2 VMSAv9-128 descriptor formats

D8.3.2.1 VMSAv9-128 Table descriptor format

IBCKNF For stage 1 translations, the following figure shows the format of the VMSAv9-128 Table descriptor.

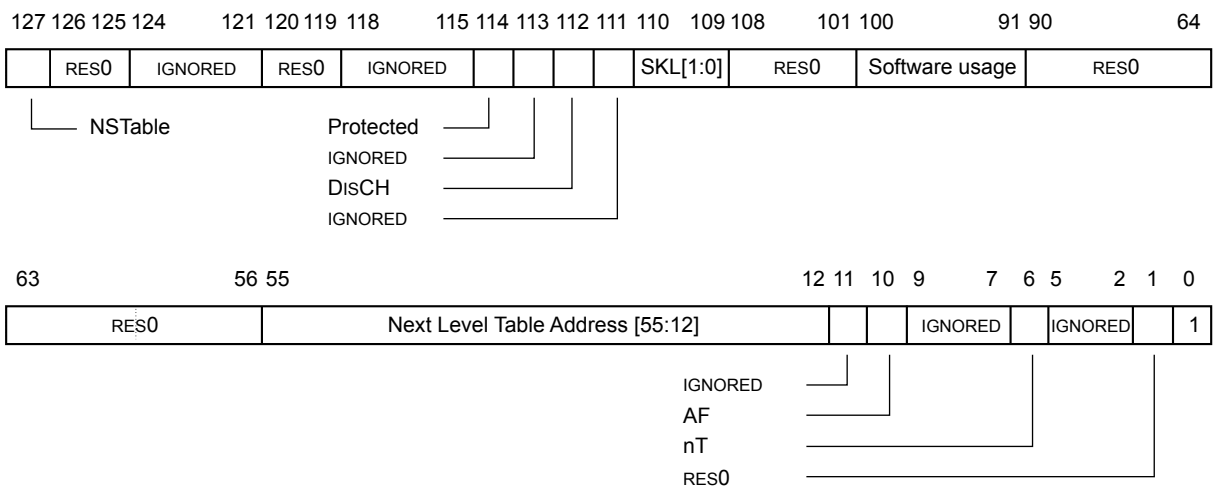

Figure D8-18 Stage 1 VMSAv9-128 Table descriptor

RFHFQV For stage 1 translations, the following table defines the fields in the VMSAv9-128 Table descriptor. In this table, NLTA is the next level table address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-56 Stage 1 VMSAv9-128 Table descriptor fields

Bit position	Field	Condition
[127]	RES0	The Security state is not Secure state.
	NSTable	Secure state. See Hierarchical control of Secure or Non-secure memory accesses.
[126:125]	RES0	-
[124:121]	RES0	Stage 1 Overlay permissions disabled.
	IGNORED	Stage 1 Overlay permissions enabled. See Stage 1 Overlay permissions.
[120:119]	RES0	-
[118:115]	IGNORED	-
[114]	RES0	FEAT_THE is not implemented.
	Protected	FEAT_THE is implemented. See Stage 1 Protected Attribute.
[113]	IGNORED	-
[112]	Disable contiguous bit (DisCH)	- See The Contiguous bit.
[111]	IGNORED	-
[110:109]	Skip level (SKL)	- See Translation using the VMSAv9-128 translation system, and to determine the descriptor type, Table D8-49.
[108:101]	RES0	-
[100:91]	Software usage	-
[90:56]	RES0	-
[55:12]	NLTA[55:12]	4KB translation granule.
	NLTA[55:14]	16KB translation granule. Descriptor bits [13:12] are RES0.
	NLTA[55:16]	64KB translation granule. Descriptor bits [15:12] are RES0.
[11]	RES0	The translation regime supports a single privilege level and the Security state is not Root state.
	IGNORED	The translation regime supports two privilege levels or the Security state is Root state.
[10]	IGNORED	Hardware managed Table descriptor Access flag is not enabled.
	Access flag (AF)	Hardware managed Table descriptor Access flag is enabled. See Hardware management of the Table descriptor Access Flag.
[9:7]	IGNORED	-
[6]	RES0	SKL is `0b00`.
	nT	SKL is not `0b00`. See Table and Block entry.
[5:2]	IGNORED	-
[1]	RES0	-
[0]	Valid descriptor	1.

ITKQXW For stage 2 translations, the following figure shows the format of the VMSAv9-128 Table descriptor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

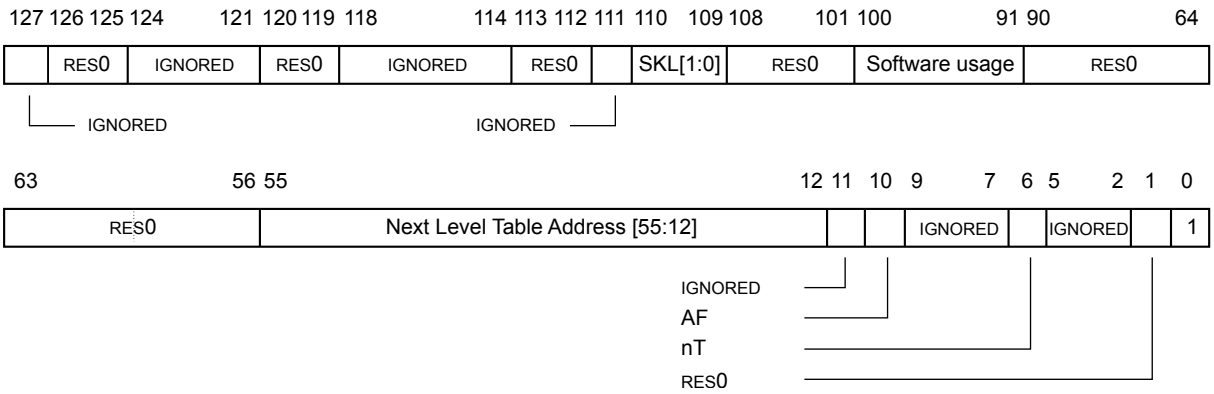

Figure D8-19 Stage 2 VMSAv9-128 Table descriptor

R_{MYLYC} For stage 2 translations, the following table defines the fields in the VMSAv9-128 Table descriptor. In this table, NLTA is the next level table address.

Table D8-57 Stage 2 VMSAv9-128 Table descriptor fields

Bit position	Field	Condition
[127]	IGNORED	Realm EL1&0 translation regime.
	RES0	Translation regime is not the Realm EL1&0 translation regime.
[126:125]	RES0	-
[124:121]	RES0	Stage 2 Overlay permissions disabled.
	IGNORED	Stage 2 Overlay permissions enabled. See Stage 2 Overlay permissions.
[120:119]	RES0	-
[118:115]	IGNORED	-
[114]	RES0	FEAT_THE is not implemented.
	IGNORED	FEAT_THE is implemented.
[113:112]	RES0	-
[111]	IGNORED	-
[110:109]	Skip level (SKL)	- See Translation using the VMSAv9-128 translation system, and to determine the descriptor type, Table D8-49.
[108:101]	RES0	-
[100:91]	Software usage	-
[90:56]	RES0	-
[55:12]	NLTA[55:12]	4KB translation granule.
	NLTA[55:14]	16KB translation granule. Descriptor bits [13:12] are RES0.
	NLTA[55:16]	64KB translation granule. Descriptor bits [15:12] are RES0.
[11]	IGNORED	-
[10]	IGNORED	Hardware managed Table descriptor Access flag is not enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	Access flag (AF)	Hardware managed Table descriptor Access flag is enabled. See Hardware management of the Table descriptor Access Flag.
[9:7]	IGNORED	-
[6]	RES0	SKL is `0b00`.
	nT	SKL is not `0b00`. See Block translation entry.
[5:2]	IGNORED	-
[1]	RES0	-
[0]	Valid descriptor	1.

D8.3.2.2 VMSAv9-128 Block descriptor and Page descriptor formats

IYCWMS For stage 1 translations, the following figure shows the format of the VMSAv9-128 Block and Page descriptor.

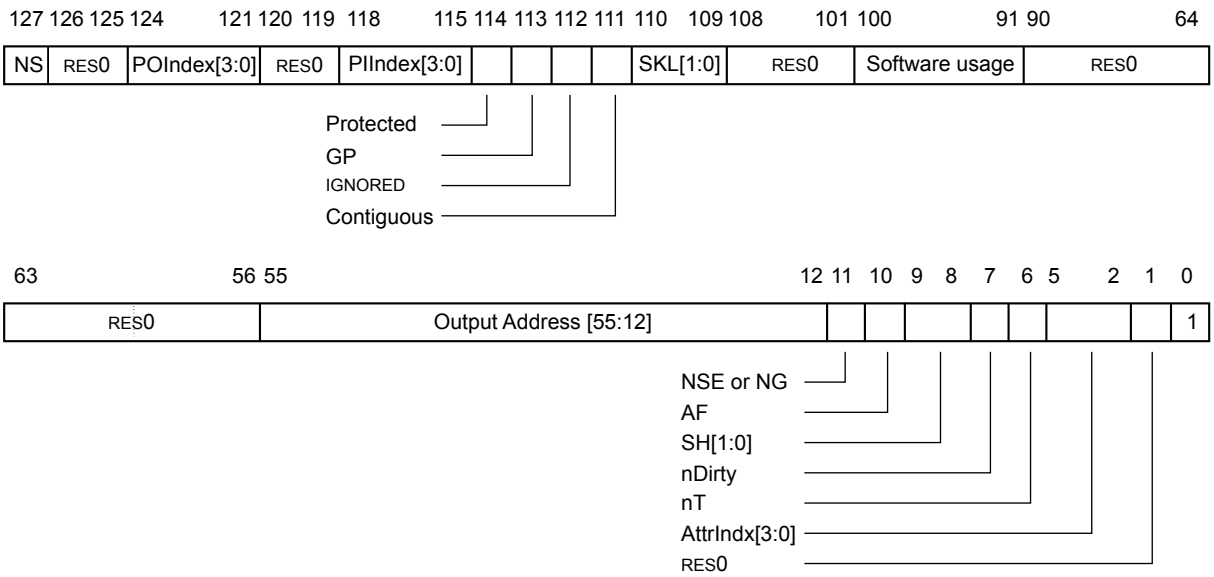

Figure D8-20 Stage 1 VMSAv9-128 Block and Page descriptor

RFPNKM For stage 1 translations, the following table defines the fields in the VMSAv9-128 Block and Page descriptor. In this table, OAB is the OA base that is appended to the IA supplied to the translation stage to produce the final OA supplied by the translation stage.

Table D8-58 Stage 1 VMSAv9-128 Block and Page descriptor fields

Bit position	Field	Condition
[127]	RES0	The access is from Non-secure state, or from Realm state using the EL1&0 translation regime.
	NS	The access is from Secure state, from Realm state using the EL2 or EL2&0 translation regimes, or from Root state.
[126:125]	RES0	-
[124:121]	RES0	Stage 1 Overlay permissions disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	POIndex[3:0]	Stage 1 Overlay permissions enabled. See Stage 1 Overlay permissions.
[120:119]	RES0	-
[118:115]	PIIndex[3:0]	-
[114]	RES0	FEAT_THE is not implemented.
	Protected	FEAT_THE is implemented. See Stage 1 Protected Attribute.
[113]	Guarded Page (GP)	- See PSTATE.BTYPE.
[112]	IGNORED	-
[111]	Contiguous	- See The Contiguous bit.
[110:109]	Skip level (SKL)	- See Translation using the VMSAv9-128 translation system, and to determine the descriptor type, Table D8-49.
[108:101]	RES0	-
[100:91]	Software usage	-
[90:56]	RES0	-
[55:12]	OAB[55:12]	4KB translation granule.
	OAB[55:14]	16KB translation granule. Descriptor bits [13:12] are RES0.
	OAB[55:16]	64KB translation granule. Descriptor bits [15:12] are RES0.
[11]	RES0	The translation regime supports a single privilege level and the Security state is not Root state.
	NSE	Root state. See Controlling memory access Security state.
	Not global (nG)	The translation regime supports two privilege levels. See Global and process-specific translation table entries.
	IGNORED	The Effective value of TCR2_ELx.FNGy is 1, where y is {0, 1}. See R_{KDVGW} in Use of ASIDs and VMIDs to reduce TLB maintenance requirements.
[10]	Access flag (AF)	- See The Access flag.
[9:8]	Shareability (SH[1:0])	- See Stage 1 Shareability attributes.
[7]	nDirty	- See The dirty state.
[6]	RES0	SKL is `0b00`.
	nT	SKL is not `0b00`. See Block translation entry.
[5]	IGNORED	FEAT_AIE is not implemented or not enabled.
	AttrIndx[3]	FEAT_AIE is implemented and enabled. See Stage 1 memory type and Cacheability attributes.
[4:2]	AttrIndx[2:0]	- See Stage 1 memory type and Cacheability attributes.
[1]	RES0	-
[0]	Valid descriptor	1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{LFZDC} For stage 2 translations, the following figure shows the format of the VMSAv9-128 Block and Page descriptor. In this table, OAB is the OA base that is appended to the IA supplied to the translation stage to produce the final OA supplied by the translation stage.

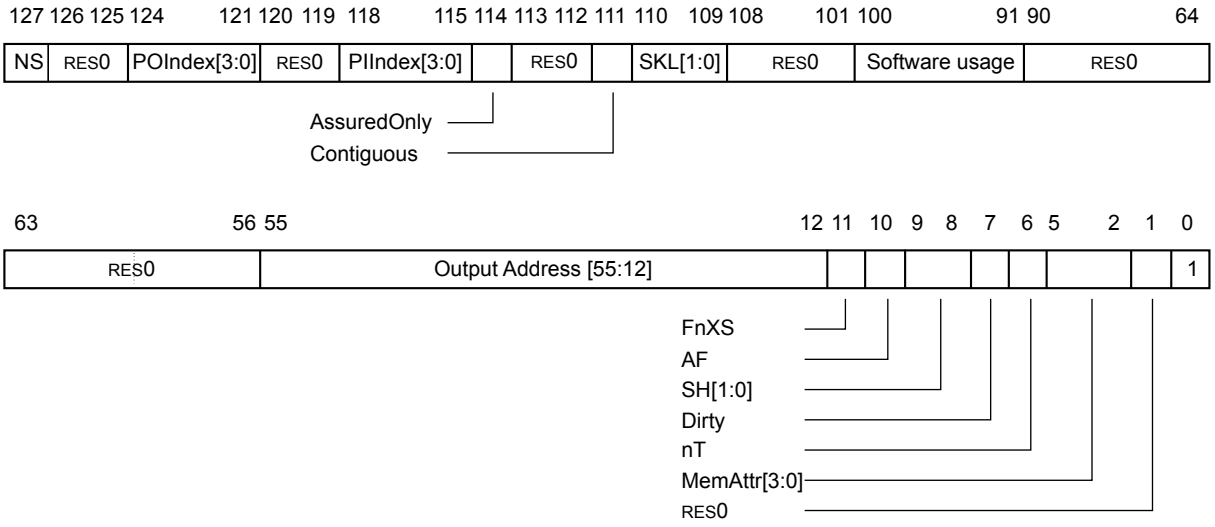

Figure D8-21 Stage 2 VMSAv9-128 Block and Page descriptor

R_{NQYBF} For stage 2 translations, the following table defines the fields in the VMSAv9-128 Block and Page descriptors:

Table D8-59 Stage 2 VMSAv9-128 Block and Page descriptor fields

Bit position	Field	Condition
[127]	RES0	The access is not from the Realm EL1&0 translation regime.
	NS	The access is from the Realm EL1&0 translation regime.
[126:125]	RES0	-
[124:121]	RES0	Stage 2 Overlay permissions disabled.
	POIndex[3:0]	Stage 2 Overlay permissions enabled. See Stage 2 Overlay permissions.
[120:119]	RES0	-
[118:115]	PIIndex[3:0]	-
[114]	RES0	FEAT_THE is not implemented.
	AssuredOnly	FEAT_THE is implemented. See Assured translation.
[113:112]	RES0	-
[111]	Contiguous	- See The Contiguous bit.
[110:109]	Skip level (SKL)	- See Translation using the VMSAv9-128 translation system, and to determine the descriptor type, Table D8-49.
[108:101]	RES0	-
[100:91]	Software usage	-
[90:56]	RES0	-
[55:12]	OAB[55:12]	4KB translation granule.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	OAB[55:14]	16KB translation granule. Descriptor bits [13:12] are RES0.
	OAB[55:16]	64KB translation granule. Descriptor bits [15:12] are RES0.
[11]	FnXS	- See XS attribute modifier.
[10]	Access flag (AF)	- See The Access flag.
[9:8]	Shareability (SH[1:0])	- See Stage 1 Shareability attributes.
[7]	Dirty	- See The dirty state.
[6]	RES0	SKL is `0b00`.
	nT	SKL is not `0b00`. See Block translation entry.
[5]	RES0	The Effective value of HCR_EL2.FWB is 1.
	MemAttr[3]	The Effective value of HCR_EL2.FWB is 0. See Stage 2 memory type and Cacheability attributes when FWB is disabled.
[4:2]	MemAttr[2:0]	- See Stage 2 memory type and Cacheability attributes when FWB is disabled and Stage 2 memory type and Cacheability attributes when FWB is enabled.
[1]	RES0	-
[0]	Invalid descriptor	0.
	Valid descriptor	1.

D8.3.2.3 VMSAv9-128 Invalid descriptor format

R_{JRLVJ} For the VMSAv9-128 translation system, the following table defines the fields in the Invalid descriptor. IGNORED fields are available for software use.

Table D8-60 VMSAv9-128 Invalid descriptor fields

Bit position	Field	Condition
[127:115]	IGNORED	-
[114]	Protected	Stage 1 and FEAT_THE is implemented. See Assured translation.
	IGNORED	Stage 2 or FEAT_THE is not implemented.
[113:1]	IGNORED	-
[0]	Invalid descriptor	0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.4 Memory access control

I_{SHRMH} Fields in the descriptors, PSTATE, and system registers determine the permissions used to control access to memory and instruction execution.

R_{MNKWG} *Direct permissions* use one or more of the following to determine the base permissions that control whether or not a location can be accessed, the type of access that is permitted, and the privilege level necessary to access the location:

- The access permissions in Block descriptors and Page descriptors.
- If enabled, then the hierarchical access permissions in Table descriptors.

R_{XRLBM} *Indirect permissions* use all of the following to determine the base permissions that control whether or not a location can be accessed, the type of access that is permitted, and the privilege level necessary to access the location:

- The *Permission Indirection Index* (PIIndex) field in Block descriptors and Page descriptors.
- For each translation regime, a Permission Indirection Register (PIR) indexed by PIIndex.

R_{SHZSG} If enabled, *Overlay permissions* use all of the following to further restrict permissions from what is configured for the base permissions:

- The *Permission Overlay Index* (POIndex) field in Block descriptors and Page descriptors.
- For each Exception level, a Permission Overlay Register (POR) indexed by POIndex.

Note

Overlay permissions allow permissions to be progressively restricted by processes running at EL0 while reducing the number of calls to supervisory software at more privileged Exception levels, and without the cost of TLB maintenance.

D8.4.1 Stage 1 permissions

R_{TDNYB} All statements in this section apply when stage 1 uses VMSAv8-64 or VMSAv9-128.

R_{WBLJK} The following table lists the stage 1 permissions and the access type that is permitted when that permission is present. If the stage 1 permission is not present, that access type is not permitted.

Table D8-61 Stage 1 permissions

Stage 1 permission	Access type permitted
UnprivRead	Unprivileged data read in the EL1&0 and EL2&0 translation regimes.
UnprivWrite	Unprivileged data write in the EL1&0 and EL2&0 translation regimes.
PrivRead	Privileged data read in the EL1&0, EL2&0, EL2, and EL3 translation regimes.
PrivWrite	Privileged data write in the EL1&0, EL2&0, EL2, and EL3 translation regimes.
UnprivGCS	Unprivileged GCS access in the EL1&0 and EL2&0 translation regimes.
PrivGCS	Privileged GCS access in the EL1&0, EL2&0, EL2, and EL3 translation regimes.
UnprivExecute	Unprivileged instruction execution in the EL1&0 and EL2&0 translation regimes.
PrivExecute	Privileged instruction execution in the EL1&0, EL2&0, EL2, and EL3 translation regimes.

I_{XFDFY} All of the following controls can ensure that memory locations are not both writable and executable:

- For privileged accesses in all translation regimes, PrivWXN.
- For unprivileged accesses in the EL1&0 and EL2&0 translation regimes, UnprivWXN.

I_{KWYWX} UnprivRead, UnprivWrite, UnprivGCS, UnprivExecute, and UnprivWXN do not apply to the EL2 and EL3 translation regimes.

R_{VBBBJ} If stage 1 translation is disabled, then stage 1 permits all data accesses, instruction executions, and GCS accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{MTSDS} If stage 1 translation is enabled, then stage 1 permissions are determined by all of the following:

- Stage 1 Base permissions. For more information, see Stage 1 Base permissions.
- Stage 1 Overlay permissions. For more information, see Stage 1 Overlay permissions.

D8.4.1.1 Stage 1 Base permissions

I_{PJNLQ} Stage 1 Base permissions can be determined by one of the following:

- Stage 1 Direct permissions. For more information, see Stage 1 Direct permissions.
- Stage 1 Indirect permissions. For more information, see Stage 1 Indirect permissions.

I_{SXTJD} If FEAT_S1PIE is not implemented, then stage 1 Direct permissions are used.

I_{VYHZC} If stage 1 translation uses VMSAv8-64, then the choice between Direct and Indirect permissions is determined by one of the following:

- If the Exception level is not EL3, then TCR2_ELx.PIE.
- If the Exception level is EL3, then TCR_EL3.PIE.

I_{BSXJT} Stage 1 Direct permissions are not supported by VMSAv9-128. If stage 1 translation uses VMSAv9-128, then it uses Indirect permissions and the Effective value of the PIE field is 1.

D8.4.1.2 Stage 1 Direct permissions

D8.4.1.2.1 Stage 1 data accesses using Direct permissions

R_{CBNKR} For a stage 1 translation that supports one Exception level, the following table shows the possible data access permissions.

Table D8-62 Data access permissions for a stage 1 translation supporting one Exception level

AP[2]	Permissions
0	PrivRead, PrivWrite
1	PrivRead

R_{MGGZM} For a stage 1 translation that supports one Exception level, AP[1] is RES1.

R_{RNGJG} For a stage 1 translation that supports two Exception levels, the following table shows the possible data access permissions at EL0 and the corresponding higher Exception level.

Table D8-63 Data access permissions for a stage 1 translation supporting two Exception levels

AP[2:1]	Permissions
00	PrivRead, PrivWrite
01	PrivRead, PrivWrite, UnprivRead, UnprivWrite
10	PrivRead
11	PrivRead, UnprivRead

R_{PSZJJ} If stage 1 is enabled and stage 1 Base permissions use Direct permissions, then GCS access is not permitted and UnprivGCS and PrivGCS are not present.

I_{ZRMQJ} For the EL1&0 translation regime in the current Security state, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then AP[1] is treated as 0 regardless of the actual value.
For more information, see Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IBVWWD If hardware management of dirty state is enabled at stage 1, then the AP[2] bit can be cleared by hardware in some situations.

For more information, see Hardware management of the dirty state.

D8.4.1.2.2 Hierarchical control of data access Direct permissions

IVZXYY Translation table entries at a given lookup level can limit data access Direct permissions at subsequent lookup levels.

IVWPGP For a translation regime, if Overlay permissions are enabled, then hierarchical data access Direct permissions are disabled. See Stage 1 Overlay permissions.

RYTXKB For a stage 1 translation, when hierarchical control is enabled, the Table descriptor APTable[1:0] field limits the data access permissions of subsequent stage 1 translation lookup levels, regardless of the Direct permissions in subsequent lookup levels, as shown in the following table:

Table D8-64 Effect of APTable[1:0] on subsequent lookup levels

APTable[1:0]	Effect at subsequent lookup levels
00	No effect on data access permissions.
01	Removes UnprivRead and UnprivWrite.
10	Removes UnprivWrite and PrivWrite.
11	Removes UnprivRead, UnprivWrite, and PrivWrite.

IGQYCH For a Permission fault, the level of the Block descriptor or Page descriptor is reported regardless of whether the lack of permissions was caused by configuration of the APTable or AP fields.

IPCLHM For translation regimes that support one Exception level, APTable[0] is RES0.

ITPCPS The APTable[1:0] settings are combined with the descriptor access permissions in subsequent lookup levels. They do not change the values entered in those descriptors, nor restrict what values can be entered.

IPKGJC For the translation regime controlled by a TCR_ELx, one or more of the following can be used to disable the Table descriptor APTable[1:0] field so that it is IGNORED by the PE and the behavior is as if the value is 0:

- If the Effective value of TCR_ELx.HPD0 is 1, hierarchical data access permission is disabled in the translation tables pointed to by TTBR0_ELx.
- If the Effective value of TCR_ELx.HPD1 is 1, hierarchical data access permission is disabled in the translation tables pointed to by TTBR1_ELx.

IYQWJF The descriptor APTable field affects all subsequent lookup levels. When an APTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.

D8.4.1.2.3 Stage 1 instruction execution using Direct permissions

IDXKST The instruction Execute-never, Unprivileged execute-never, and Privileged execute-never fields in a Block descriptor and Page descriptor are used to control instruction execution permissions under the Direct permissions scheme by a stage 1 translation.

RTRZVM For a stage 1 translation that supports one Exception level, the XN field in the Block descriptor and Page descriptor has all of the following properties:

- If the Effective value of XN is 0, then PrivExecute is not removed by this bit.
- If the Effective value of XN is 1, then PrivExecute is removed.

RRBDFT For a stage 1 translation that supports two Exception levels, the UXN field in the Block descriptor and Page descriptor has all of the following properties:

- If the Effective value of UXN is 0, then UnprivExecute is not removed by this bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of UXN is 1, then UnprivExecute is removed.

R_{LWPVL} For a stage 1 translation that supports two Exception levels, the PXN field in the Block descriptor and Page descriptor has all of the following properties:

- If the Effective value of PXN is 0, then PrivExecute is not removed by this bit.
- If the Effective value of PXN is 1, then PrivExecute is removed.

R_{VWLLR} For stage 1 translations that support two Exception levels, if UnprivWrite is present, then PrivExecute is removed.

D8.4.1.2.4 Preventing execution from writable locations

I_{KDRYB} There are register control fields that can be used to force writable memory to be treated as XN, PXN, or UXN, regardless of the value of the corresponding descriptor fields.

R_{FYMXJ} For unprivileged accesses in the EL1&0 and EL2&0 translation regimes, if SCTLR_ELx.WXN is 1, UnprivWrite is present, and UnprivExecute is present, then UnprivWXN is applied. Otherwise, UnprivWXN is removed.

R_{SYMMB} For privileged accesses in all translation regimes, if SCTLR_ELx.WXN is 1, PrivWrite is present, and PrivExecute is present, then PrivWXN is applied. Otherwise, PrivWXN is removed.

I_{TBVCH} For a stage 1 translation in a translation regime that supports two Exception levels, the corresponding SCTLR_ELx.WXN field does all of the following:

- If the value is 0, then there is no effect on access permissions.
- If the value is 1, then all of the following apply:
  - If the memory region has UnprivWrite, then UnprivExecute is removed, regardless of the value of the Block descriptor or Page descriptor UXN field.
  - If the memory region has PrivWrite, then PrivExecute is removed, regardless of the value of the Block descriptor or Page descriptor PXN field.

I_{FQVWM} For a stage 1 translation in a translation regime that applies to a single Exception level, the corresponding SCTLR_ELx.WXN field does all of the following:

- If the value is 0, then there is no effect on access permissions.
- If the value is 1 and if the memory region is PrivWrite, then PrivExecute is removed, regardless of the value of the Block descriptor or Page descriptor XN field.

I_{XLDPQ} The SCTLR_ELx.WXN field reduces the memory footprint available for code injection attacks by introducing an invariant that writable memory regions are not executable.

I_{RNXSK} For the lifetime of a given virtual machine, Arm expects the SCTLR_ELx.WXN field to remain static in normal operation.

I_{PNSFD} SCTLR_ELx.WXN is permitted to be cached in a TLB.

D8.4.1.2.5 Hierarchical control of instruction execution for Direct permissions

I_{JCPHF} Stage 1 translation table entries at a given lookup level can limit instruction execution controls expressed using Direct permissions at subsequent lookup levels.

I_{XMMSD} For a translation regime, if Overlay permissions are enabled, then hierarchical instruction execution controls expressed using Direct permissions are disabled. See Stage 1 Overlay permissions.

R_{RLQFP} For a stage 1 translation, the value of the XNTable Table descriptor field has one of the following effects:

- If the Effective value of the XNTable field is 0, then the field has no effect.
- If the Effective value of the XNTable field is 1, then all of the following apply:
  - The XN field in Block descriptors and Page descriptors is treated as 1 in subsequent lookup levels, regardless of the actual value of XN.
  - The value and interpretation of the XNTable and XN fields in all subsequent lookup levels are otherwise unaffected.

R_{WSFPB} For a stage 1 translation, the value of the UXNTable Table descriptor field has one of the following effects:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of the UXNTable field is 0, then the field has no effect.
- If the Effective value of the UXNTable field is 1, then all of the following apply:
  - The UXN field in Block descriptors and Page descriptors is treated as 1 in subsequent lookup levels, regardless of the actual value of UXN.
  - The value and interpretation of the UXNTable and UXN fields in all subsequent lookup levels are otherwise unaffected.

R_{NJJNG} For a stage 1 translation, the value of the PXNTable Table descriptor field has one of the following effects:

- If the Effective value of the PXNTable field is 0, then the field has no effect.
- If the Effective value of the PXNTable field is 1, then all of the following apply:
  - The PXN field in Block descriptors and Page descriptors is treated as 1 in subsequent lookup levels, regardless of the actual value of PXN.
  - The value and interpretation of the PXNTable and PXN fields all subsequent lookup levels are otherwise unaffected.

I_{SRRJF} The UXNTable, XNTable, and PXNTable Table descriptor fields control stage 1 translations. The corresponding bit positions are RES0 in the stage 2 Table descriptors.

I_{NYWHQ} For the translation regime controlled by a TCR_ELx, one or more of the following can be used to disable the Table descriptor XNTable, UXNTable, and PXNTable fields so they are IGNORED by the PE and the system behavior is as if the values are zero:

- If the Effective value of TCR_ELx.HPD0 is 1, then hierarchical instruction execution permission is disabled in the translation tables pointed to by TTBR0_ELx.
- If the Effective value of TCR_ELx.HPD1 is 1, then hierarchical instruction execution permission is disabled in the translation tables pointed to by TTBR1_ELx.

I_{ZGSNV} The Table descriptor UXNTable, XNTable, and PXNTable fields affect all subsequent lookup levels, including stage 1 translation output. When a UXNTable, XNTable, or PXNTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.

D8.4.1.2.6 Summary of Direct permissions for stage 1 translations

I_{WXKKQ} If a translation regime applies to EL0 and a higher Exception level, then the following table shows the possible stage 1 memory access permissions, using all of the following notations:

- PXN and UXN are the Effective values of PXN and UXN after the effects of PXNTable and UXNTable, respectively.
- AP[2:1] is the Effective value of AP[2:1] after the effects of APTable are applied.
- If hardware update of dirty state is enabled, then AP[2] is considered to be 0 for the purposes of calculating execute permission in some situations.
- WXN is in the SCTLR_ELx register at the higher Exception level to which the translation regime applies.

Table D8-65 Summary of possible memory access permissions using Direct permissions for a stage 1 translation supporting two Exception levels

UXN	PXN	AP[2:1]	WXN	Permission
0	0	00	0	PrivRead, PrivWrite, PrivExecute, UnprivExecute
0	0	00	1	PrivRead, PrivWrite, PrivWXN, UnprivExecute
0	0	01	0	PrivRead, PrivWrite, UnprivRead, UnprivWrite, UnprivExecute
0	0	01	1	PrivRead, PrivWrite, UnprivRead, UnprivWrite, UnprivWXN
0	0	10	x	PrivRead, PrivExecute, UnprivExecute
0	0	11	x	PrivRead, PrivExecute, UnprivRead, UnprivExecute
0	1	00	x	PrivRead, PrivWrite, UnprivExecute

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UXN	PXN	AP[2:1]	WXN	Permission
0	1	01	0	PrivRead, PrivWrite, UnprivRead, UnprivWrite, UnprivExecute
0	1	01	1	PrivRead, PrivWrite, UnprivRead, UnprivWrite, UnprivWXN
0	1	10	x	PrivRead, UnprivExecute
0	1	11	x	PrivRead, UnprivRead, UnprivExecute
1	0	00	0	PrivRead, PrivWrite, PrivExecute
1	0	00	1	PrivRead, PrivWrite, PrivWXN
1	0	01	x	PrivRead, PrivWrite, UnprivRead, UnprivWrite
1	0	10	x	PrivRead, PrivExecute
1	0	11	x	PrivRead, PrivExecute, UnprivRead
1	1	00	x	PrivRead, PrivWrite
1	1	01	x	PrivRead, PrivWrite, UnprivRead, UnprivWrite
1	1	10	x	PrivRead
1	1	11	x	PrivRead, UnprivRead

I_{LJHZZ} For a memory location, if a translation regime applies to a single Exception level, then the following table shows the stage 1 access Direct permissions, using all of the following notations:

- WXN is in the SCTLR_ELx register at the higher Exception level to which the translation regime applies.
- AP[2] is the effective value of AP[2] after the effects of APTable are applied.
- If hardware update of dirty state is enabled, then AP[2] is considered to be 0 for the purposes of calculating execute permission in some situations.

Table D8-66 Summary of possible memory access permissions using Direct permissions for a stage 1 translation supporting one Exception level

XN	AP[2]	WXN	Access permission
0	0	0	PrivRead, PrivWrite, PrivExecute
0	0	1	PrivRead, PrivWrite, PrivWXN
0	1	x	PrivRead, PrivExecute
1	0	x	PrivRead, PrivWrite
1	1	x	PrivRead

I_{JZPHK} For more information, see Implications of enabling the dirty state management mechanism.

D8.4.1.3 Stage 1 Indirect permissions

R_{JJWXH} All statements in this section require implementation of FEAT_S1PIE.

I_{LYWDV} For the EL1&0 and EL2&0 translation regimes, the following Permission Indirection Registers determine unprivileged stage 1 Base permissions, *S1UnprivBasePerm* :

- PIRE0_EL1.
- PIRE0_EL2.

R_{DJZHM} For all purposes other than a direct read of the register, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then the value of PIRE0_EL1 is treated as 0.

I_{BSMJB} For all translation regimes, the following Permission Indirection Registers determine privileged stage 1 Base permissions, *S1PrivBasePerm* :

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PIR_EL1.
- PIR_EL2.
- PIR_EL3.

R_{BCPGD} The following table shows how the 4-bit PIIndex field in stage 1 Block descriptors and Page descriptors is used to index into the Permission Indirection Registers to obtain S1UnprivBasePerm and S1PrivBasePerm:

Table D8-67 Using PIIndex to determine stage 1 Base permissions

Translation regime	S1UnprivBasePerm	S1PrivBasePerm
EL1&0	PIRE0_EL1[(PIIndex*4 + 3): PIIndex*4]	PIR_EL1[(PIIndex*4 + 3): PIIndex*4]
EL2&0	PIRE0_EL2[(PIIndex*4 + 3): PIIndex*4]	PIR_EL2[(PIIndex*4 + 3): PIIndex*4]
EL2		PIR_EL2[(PIIndex*4 + 3): PIIndex*4]
EL3		PIR_EL3[(PIIndex*4 + 3): PIIndex*4]

I_{MYXBN} Stage 1 Base permissions are calculated using the following steps:

1. Stage 1 Base permissions are decoded from the PIR_ELx register.
2. If PrivExecute or PrivGCS is present, and UnprivWrite or UnprivGCS is additionally present, then all Base permissions are removed. See R_{VFPJF}.
3. For the translation, if an inappropriate OA space is selected, then each of PrivExecute, UnprivExecute, PrivGCS, UnprivGCS are removed. See R_{CYSKB} and R_{YBDFF}.
4. The effects of PSTATE.PAN are applied. It is IMPLEMENTATION DEFINED whether this is prioritized here or above step 3. See R_{TBKKB}.

R_{LLZDZ} The following table shows the privileged and unprivileged stage 1 Base permissions that are applied as determined by the corresponding S1PrivBasePerm and S1UnprivBasePerm values, and whether stage 1 Overlay permissions are applied to those stage 1 Base permissions:

Table D8-68 Determination of stage 1 Base permissions and Overlay application

S1PrivBasePerm / S1UnprivBasePerm	Stage 1 Base Permissions	Stage 1 Overlay applied
`0b0000`	-	Yes
`0b0001`	PrivRead / UnprivRead	Yes
`0b0010`	PrivExecute / UnprivExecute	Yes
`0b0011`	PrivRead / UnprivRead, PrivExecute / UnprivExecute	Yes
`0b0100`	Reserved, treated as No access	Yes
`0b0101`	PrivRead / UnprivRead, PrivWrite / UnprivWrite	Yes
`0b0110`	PrivRead / UnprivRead, PrivWrite / UnprivWrite, PrivExecute / UnprivExecute	Yes
`0b0111`	PrivRead / UnprivRead, PrivWrite / UnprivWrite, PrivExecute / UnprivExecute	Yes
`0b1000`	PrivRead / UnprivRead	No

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S1PrivBasePerm / S1UnprivBasePerm	Stage 1 Base Permissions	Stage 1 Overlay applied
0b1001	PrivRead / UnprivRead, PrivGCS / UnprivGCS	No
0b1010	PrivRead / UnprivRead, PrivExecute / UnprivExecute	No
0b1011	Reserved, treated as No access	No
0b1100	PrivRead / UnprivRead, PrivWrite / UnprivWrite	No
0b1101	Reserved, treated as No access	No
0b1110	PrivRead / UnprivRead, PrivWrite / UnprivWrite, PrivExecute / UnprivExecute	No
0b1111	Reserved, treated as No access	No

Note

For each row in Table D8-68, stage 1 Base permissions are removed unless stated in the table.
Bit [3] in S1PrivBasePerm and S1UnprivBasePerm determines whether stage 1 Overlay permissions are applied to the corresponding stage 1 privileged and unprivileged permissions.

R_{HJYGR} The following table shows how the PrivWXN and UnprivWXN controls are determined by the corresponding S1PrivBasePerm and S1UnprivBasePerm values.

Table D8-69 Determination of PrivWXN and UnprivWXN

S1PrivBasePerm / S1UnprivBasePerm	PrivWXN / UnprivWXN control
0b0110	Applied
Others	Removed

I_{PMKFB} If stage 1 Indirect permissions are used, then SCTLR_ELx.WXN has no effect and is RES0.

I_{KDZDQ} PrivWXN and UnprivWXN are not permissions, but controls that affect the calculation of Write and Execute permissions at stage 1. For the details of how this is calculated, see Combining stage 1 Base permissions and Overlay permissions.

R_{RSXJG} If stage 1 translation is using Indirect permissions, then a memory region is execute-never at stage 1 if PrivExecute and UnprivExecute are not present.

R_{VFPJF} For the EL1&0 and EL2&0 translation regimes, all of the following combinations of S1PrivBasePerm and S1UnprivBasePerm are reserved, and the stage 1 Base permissions are removed:

- S1PrivBasePerm is one of {0b0010, 0b0011, 0b0110, 0b0111, 0b1001, 0b1010, 0b1110}, which apply PrivExecute or PrivGCS.
- S1UnprivBasePerm is one of {0b0101, 0b0110, 0b0111, 0b1001, 0b1100, 0b1110}, which apply UnprivWrite or UnprivGCS.

Note

The combination of unprivileged write or unprivileged GCS access to memory locations that are also privileged executable or privileged GCS accessible is considered unsafe and should never be used by software. As a safety mechanism, all access permissions are removed. This is different than the mechanism used by Direct permissions. In that case, if unprivileged write permission to a memory location is permitted, then privileged execute permission is removed, but other permissions are retained.
This behavior is independent of stage 1 Overlay permissions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CYSKB} If any of the following are true, then the stage 1 Base permissions, PrivGCS and UnprivGCS, are removed:

- For Secure translation regimes, the stage 1 OA resides in the Non-secure address space.
- For Root state, the stage 1 OA does not reside in the Root address space.
- For the Realm EL2 and Realm EL2&0 translation regimes, the stage 1 OA does not reside in the Realm address space.

R_{YBDFF} If any of the following are true, then the stage 1 Base permissions, PrivExecute and UnprivExecute, are removed:

- For Secure translation regimes when SCR_EL3.SIF is set to 1, the stage 1 OA resides in the Non-secure address space.
- For Root state, the stage 1 OA does not reside in the Root address space.
- For the Realm EL2 and Realm EL2&0 translation regimes, the stage 1 OA does not reside in the Realm address space.

R_{TBKKB} For the EL1&0 and EL2&0 translation regimes, if the Effective value of PSTATE.PAN is 1 and S1UnprivBasePerm is not `0b0000`, then the stage 1 Base permissions, PrivRead and PrivWrite, are removed.

It is IMPLEMENTATION DEFINED whether this applies when the S1UnprivBasePerm is a reserved value, which is therefore treated as `0b0000`.

It is IMPLEMENTATION DEFINED whether this applies when the effect of SCR_EL3.SIF and the PA space, as described in R_{YBDFF}, results in removal of all unprivileged stage 1 base permissions.

Note

The effect of PSTATE.PAN does not depend on the value of SCTLR_ELx.EPAN. If stage 1 uses Indirect permissions, then SCTLR_ELx.EPAN is RES1.

I_{LSZBZ} When the Effective value of PSTATE.PAN is 1, R_{TBKKB} ensures that memory with any form of user permissions is not accessible by privileged data accesses.

R_{FBQWG} Except for the impact of PSTATE.PAN, the Effective value of the stage 1 Base permissions are permitted to be cached in a TLB.

R_{JHSVW} If Indirect permissions are used, then hierarchical permissions are disabled and TCR_ELx.HPD*n* are RES1.

D8.4.1.4 Stage 1 Overlay permissions

R_{JCYSC} If FEAT_S1POE is not implemented, then S1UnprivOverlay and S1PrivOverlay are disabled.

R_{RSKVR} All remaining statements in this section require implementation of FEAT_S1POE.

R_{RQWXN} The stage 1 *Permission Overlay Enable* (POE) is determined by the following table:

Table D8-70 Determination of stage 1 POE value

Translation Regime	Field
EL1&0	TCR2_EL1.POE
EL2&0	TCR2_EL2.POE
EL2	TCR2_EL2.POE
EL3	TCR_EL3.POE

I_{KPGBV} The POE field is not permitted to be cached in a TLB.

R_{TMNYQ} For the EL1&0 translation regime, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, the value of the stage 1 *EL0 Permission Overlay Enable* (E0POE) is 0. Otherwise, the E0POE field is determined by the following table:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-71 Determination of stage 1 E0POE value

Translation Regime	Field
EL1&0	TCR2_EL1.E0POE
EL2&0	TCR2_EL2.E0POE

I_{SDBKH} The E0POE field is not permitted to be cached in a TLB.

R_{BVXDG} For a translation regime, if one or more of the following apply, then hierarchical permissions are disabled and the corresponding TCR_ELx.HPD*n* are RES1:

- For any translation regime, POE is 1.
- For the EL1&0 and EL2&0 translation regimes, E0POE is 1.

R_{GDVJK} If E0POE is 1 and one of the following applies, then stage 1 unprivileged overlay, *S1UnprivOverlay*, is enabled, otherwise it is disabled:

- Stage 1 uses Direct permissions.
- Stage 1 uses Indirect permissions and the Unprivileged Base permission configuration selects a value for which Overlay is applied.

R_{DPZHF} If POE is 1 and one of the following applies, then stage 1 privilege overlay, *S1PrivOverlay*, is enabled, otherwise it is disabled:

- Stage 1 uses Direct permissions.
- Stage 1 uses Indirect permissions and the Privileged Base permission configuration selects a value for which Overlay is applied.

I_{SXYLW} For the EL1&0 and EL2&0 translation regimes, the POR_EL0 Permission Overlay register determines the unprivileged stage 1 overlay permissions, *S1UnprivOverlayPerm*.

I_{YQTQJ} For all translation regimes, the following Permission Overlay registers determine privileged stage 1 overlay permissions, *S1PrivOverlayPerm* :

- POR_EL1.
- POR_EL2.
- POR_EL3.

I_{JGCLT} The size of the POIndex field in Block descriptors and Page descriptors is one of the following:

- In VMSAv8-64, 3-bits.
- In VMSAv9-128, 4-bits.

R_{BXJBV} The following table shows how the POIndex field in stage 1 Block descriptors and Page descriptors is used to index into the Permission Overlay registers to obtain S1UnprivOverlayPerm and S1PrivOverlayPerm:

Table D8-72 Using POIndex to determine stage 1 overlay permissions

Translation Regime	S1UnprivOverlayPerm	S1PrivOverlayPerm
EL1&0	POR_EL0[(POIndex*4 + 3): POIndex*4]	POR_EL1[(POIndex*4 + 3): POIndex*4]
EL2&0	POR_EL0[(POIndex*4 + 3): POIndex*4]	POR_EL2[(POIndex*4 + 3): POIndex*4]
EL2		POR_EL2[(POIndex*4 + 3): POIndex*4]
EL3		POR_EL3[(POIndex*4 + 3): POIndex*4]

R_{PGLZR} The following table shows the effect of applying stage 1 Overlay permissions to restrict stage 1 permissions:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-73 Effect of stage 1 Overlay permissions

Stage 1 Overlay permission	Effect when permission is 0
UnprivRead	Restricts read permissions for unprivileged data accesses in the EL1&0 and EL2&0 translation regimes. Not Applicable in the EL2 and EL3 translation regimes.
UnprivWrite	Restricts write permissions for unprivileged data accesses in the EL1&0 and EL2&0 translation regimes. Not Applicable in the EL2 and EL3 translation regimes.
PrivRead	Restricts read permissions for privileged data accesses in the EL1&0, EL2&0, EL2, and EL3 translation regimes.
PrivWrite	Restricts write permissions for privileged data accesses in the EL1&0, EL2&0, EL2, and EL3 translation regimes.
UnprivExecute	Restricts permission for unprivileged instruction execution in the EL1&0 and EL2&0 translation regimes. Not Applicable in the EL2 and EL3 translation regimes.
PrivExecute	Restricts permission for privileged instruction execution in the EL1&0, EL2&0, EL2, and EL3 translation regimes.

Note

GCS permissions are not affected by stage 1 Overlay permissions.

RWDCGP The following table shows the privileged and unprivileged stage 1 overlay permissions that are applied as determined by the corresponding S1PrivOverlayPerm and S1UnprivOverlayPerm values:

Table D8-74 Determination of stage 1 Overlay permissions

S1PrivOverlayPerm / S1UnprivOverlayPerm	Stage 1 Overlay permissions
`0b0000`	-
`0b0001`	PrivRead / UnprivRead
`0b0010`	PrivExecute / UnprivExecute
`0b0011`	PrivRead / UnprivRead PrivExecute / UnprivExecute
`0b0100`	PrivWrite / UnprivWrite
`0b0101`	PrivRead / UnprivRead PrivWrite / UnprivWrite
`0b0110`	PrivWrite / UnprivWrite PrivExecute / UnprivExecute
`0b0111`	PrivRead / UnprivRead PrivWrite / UnprivWrite PrivExecute / UnprivExecute
`0b1xxx`	Reserved, treated as `0b0000`

Note

For each row in Table D8-74, stage 1 Overlay permissions are 0 unless stated in the table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{SKPNZ} Stage 1 Overlay permissions are not permitted to be cached in a TLB. POIndex is permitted to be cached in a TLB.

I_{VCQZV} For a translation regime using VMSAv8-64 that is in-context, when TCR(2)_ELx.{HPD*n*, POE} or TCR(2)_ELx.{HPD*n*, E0P0E} transition from {0, 0} to {0, 1} or to {1, 1}, or vice-versa, the resulting stage 1 permissions are CONSTRAINED UNPREDICTABLE, as described in CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

I_{VRTYL} For a translation regime using VMSAv8-64 that is in-context, when TCR(2)_ELx.{HWU*nn*, POE} or TCR(2)_ELx.{HWU*nn*, E0P0E} transition from {1, 0} to {0, 1} or to {1, 1}, or vice-versa, the resulting stage 1 PBHA attributes and stage 1 permissions are CONSTRAINED UNPREDICTABLE, as described in CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

D8.4.1.5 Combining stage 1 Base permissions and Overlay permissions

R_{QXXPC} If PrivWXN is present, then all of the following apply to stage 1 Base permissions and stage 1 Overlay permissions:

- If S1PrivOverlay is disabled and the decoded stage 1 Base permission PrivWrite is present, then the stage 1 Base permission PrivExecute is removed.
- If S1PrivOverlay is enabled and the stage 1 Overlay permission PrivExecute is present, then the stage 1 Overlay permission PrivWrite is removed.

R_{HQCYT} If PrivWXN is removed, then it has no effect on stage 1 Base permissions and stage 1 Overlay permissions.

R_{NPBXC} If UnprivWXN is present, then all of the following apply to stage 1 Base permissions and stage 1 Overlay permissions:

- If S1UnprivOverlay is disabled and the decoded stage 1 Base permission UnprivWrite is present, then the stage 1 Base permission UnprivExecute is removed.
- If S1UnprivOverlay is enabled and the stage 1 Overlay permission UnprivExecute is present, then the stage 1 Overlay permission UnprivWrite is removed.

R_{GZLXV} If UnprivWXN is removed, then it has no effect on stage 1 Base permissions and stage 1 Overlay permissions.

R_{NHNQC} PrivWXN and UnprivWXN are permitted to be cached in a TLB.

I_{XGFTP} Before applying PSTATE.PAN, stage 1 Base permissions are permitted to be cached in a TLB, therefore the effect of PrivWXN and UnprivWXN on stage 1 Base permissions is also permitted to be cached in a TLB. Stage 1 Overlay permissions are not permitted to be cached in a TLB, therefore the effect of PrivWXN and UnprivWXN on stage 1 Overlay permissions is also not permitted to be cached in a TLB.

R_{CZCYQ} All of the following determine whether stage 1 permissions are present or removed:

- For PrivGCS and UnprivGCS, if the corresponding Effective stage 1 Base permission is present, then the permission is present. Otherwise, it is removed.
- PrivRead, PrivWrite, and PrivExecute are determined by one of the following:
  - If S1PrivOverlay is disabled, then when the corresponding Effective stage 1 Base permission is present, the permission is present. Otherwise, it is removed.
  - If S1PrivOverlay is enabled, then when both the corresponding Effective stage 1 Base permission and corresponding Effective stage 1 Overlay permission are present, the permission is present. Otherwise, it is removed.
- UnprivRead, UnprivWrite, and UnprivExecute are determined by one of the following:
  - If S1UnprivOverlay is disabled, then when the corresponding Effective stage 1 Base permission is present, the permission is present. Otherwise, it is removed.
  - If S1UnprivOverlay is enabled, then when both the corresponding Effective stage 1 Base permission and corresponding Effective stage 1 Overlay permission are present, the permission is present. Otherwise, it is removed.

R_{RPXZR} If any of the following applies, then a stage 1 Permission fault due to Overlay is generated:

- S1PrivOverlay is enabled and a privileged access is removed by the Effective value of the stage 1 Overlay permission.
- S1UnprivOverlay is enabled and an unprivileged access is removed by the Effective value of the stage 1 Overlay permission.

I_{DWBGP} For information on the priority of a stage 1 Permission fault due to Overlay, see Prioritization of Permission faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.4.2 Stage 2 permissions

R_{VHFGB} All statements in this section apply when stage 2 uses VMSAv8-64 or VMSAv9-128.

R_{BHXHT} The following table lists the stage 2 permissions and the access type that is permitted when the value of that permission is 1.

Table D8-75 Stage 2 permissions

Stage 2 permission	Permitted access
RO	Read only, including Data Read, RCW Read, MMU Read
WO	Write only, including Data Write, RCW Write, MMU Write
RW	Read and write, including Data Read, Data Write, RCW Read, RCW Write, MMU Read, MMU Write
MRO	Mostly read only, including Data Read, RCW Read, RCW Write, MMU Read, MMU Write
uX	Unprivileged Execute
pX	Privileged Execute
puX	Privileged Execute, Unprivileged Execute

R_{BBLJH} If stage 2 translation is disabled, then stage 2 permits all data accesses, instruction executions, and MMU accesses.

R_{ZRSPR} Stage 2 permissions are determined by one of the following:

- Stage 2 Direct permissions. For more information, see Stage 2 Direct permissions.
- Stage 2 Indirect permissions, which combine stage 2 Base permissions with, if enabled, stage 2 Overlay permissions. For more information, see Stage 2 Indirect permissions.

R_{KXXWX} If FEAT_S2PIE is not implemented, then stage 2 Direct permissions are used.

R_{JMGVS} If stage 2 translation uses VMSAv9-128, then the Effective value of the *stage 2 Permission Indirection Enable* (S2PIE) is 1. If the translation regime uses VMSAv8-64, then the stage 2 S2PIE value is determined by VTCR_EL2.S2PIE.

I_{JGTWJ} Stage 2 Direct permissions are not combined with stage 2 Overlay permissions described in Stage 2 Overlay permissions. This is different from stage 1 Base permissions where stage 1 Direct permissions are combined with stage 1 Overlay permissions.

D8.4.2.1 Stage 2 Direct permissions

I_{HHYYW} Stage 2 Direct permissions are not supported by VMSAv9-128.

R_{TVZYF} For stage 2 Direct permissions, then:

- Read permission implies MMU Read and RCW(S) Read permissions.
- Write permission implies MMU Write and RCW(S) Write permissions.

D8.4.2.1.1 Stage 2 data accesses using Direct permissions

R_{YLJXV} For the EL1&0 translation regime, if stage 2 translation is enabled, then the S2AP descriptor field determines the stage 2 data access permissions.

R_{HBFHL} The following table shows encoding of the S2AP descriptor field:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-76 Data access permissions for a stage 2 translation

S2AP[1:0]	Access permission from EL1&0
00	No data access
01	RO
10	WO
11	RW

I_{HXWLL} The S2AP data access permissions do not distinguish between EL1 accesses and EL0 accesses.

R_{FQNDX} If an attempt is made to access memory that is not permitted by the value of the S2AP field, then a stage 2 Permission fault is generated.

I_{NTJRL} If hardware management of dirty state is enabled at stage 2, then the S2AP[1] bit can be set to 1 by hardware in some situations.
For more information, see Hardware management of the dirty state.

D8.4.2.1.2 Stage 2 instruction execution using Direct permissions

I_{GNDQQ} The instruction Execute-never field in a Block descriptor and Page descriptor is used to control instruction execution permissions by a stage 2 translation.

R_{PRCFD} For a stage 2 translation, if FEAT_XNX is not implemented, then the XN field in the Block descriptor and Page descriptor controls instruction execution permissions as shown in the following table:

Table D8-77 Stage 2 instruction access permissions, FEAT_XNX not implemented

XN	Access
0	The stage 2 control does not forbid execution at EL1 and EL0.
1	The stage 2 control does not permit execution at EL1 or EL0.

R_{STCTY} For a stage 2 translation, if FEAT_XNX is implemented, then the XN[1:0] field in the Block descriptor and Page descriptor controls instruction execution permissions as shown in the following table:

Table D8-78 Stage 2 instruction access permissions, FEAT_XNX implemented

XN[1]	XN[0]	Access
0	0	The stage 2 control does not forbid execution at EL1 and EL0.
0	1	The stage 2 control does not permit execution at EL1, but does not forbid execution at EL0.
1	0	The stage 2 control does not permit execution at EL1 or EL0.
1	1	The stage 2 control does not forbid execution at EL1, but does not permit execution at EL0.

D8.4.2.2 Stage 2 Indirect permissions

R_{WCPLC} All statements in this section require implementation of FEAT_S2PIE.

I_{WJDFF} Stage 2 Indirect permissions determine the stage 2 permissions by combining all of the following:

- Stage 2 Base permissions. For more information, see Stage 2 Base permissions.
- Stage 2 Overlay permissions. For more information, see Stage 2 Overlay permissions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.4.2.2.1 Stage 2 Base permissions

RWKZJP Stage 2 Base permissions are determined by all of the following:

- Stage 2 General permissions.
- Stage 2 Special permissions.

RWLVPJ The following table shows the stage 2 General permissions and the permitted accesses:

Table D8-79 Accesses permitted by stage 2 General permissions

General permission	Permitted access
NoAccess	No Access
RO	Data Read, RCW Read, and MMU Read
RW	Data Read, Data Write, RCW Read, RCW Write, MMU Read, and MMU Write
uX	Unprivileged Execute
pX	Privileged Execute
puX	Privileged Execute and Unprivileged Execute

RWRFCR The following table shows the stage 2 Special permissions and the permitted accesses:

Table D8-80 Accesses permitted by stage 2 Special permissions

Permission	Permitted Access
WO	Data Write
MRO	Mostly read only, including Data Read, RCW Read, RCW Write, MMU Read, MMU Write
MRO-TL0, MRO-TL1, MRO-TL01	Data Read, RCW Read, RCW Write, MMU Read, MMU Write

Note

The stage 2 MRO* permissions do not provide data write permission.

IBPBBQ The *Mostly Read Only* (MRO) permission prevents explicit write accesses from the EL1&0 translation regime to EL1 translation tables while permitting hardware updates to the Access flag, dirty state, and the writes due to `RCW` and `RCWS` instructions.

RBQJWZ If an IPA is marked as MRO in stage 2, then it has the following *MRO access permissions*:

- A write to that IPA from an `RCW` instruction at EL1 or EL0 is permitted by stage 2.
- A write to that IPA from a stage 1 translation table walk as part of an Access flag or dirty state update is permitted by stage 2.
- Any other write to that IPA is not permitted by stage 2 and generates a stage 2 Permission fault.

IHHBFN All of the following TopLevel permissions determine the outcome of the TopLevel checks:

- For a VA translated by TTBR0_EL1, when TopLevel0 is 1, the IPA is permitted to be used by the first level of the translation table walk.
- For a VA translated by TTBR1_EL1, when TopLevel1 is 1, the IPA is permitted to be used by the first level of the translation table walk.

For more information, see Stage 2 TopLevel checks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CWZCV} The following table shows how the stage 2 MRO TopLevel permissions are determined:

Table D8-81 Stage 2 MRO TopLevel permissions

MRO TopLevel permission	TopLevel attributes
MRO-TL0	{TopLevel0, TopLevel1} is {1, 0}
MRO-TL1	{TopLevel0, TopLevel1} is {0, 1}
MRO-TL01	{TopLevel0, TopLevel1} is {1, 1}
Otherwise	{TopLevel0, TopLevel1} is {0, 0}

R_{JDVSM} For stage 2 permissions not affected by VTCR_EL2.GCSH, all of the following apply:

- If data read is permitted, then GCS read is permitted.
- If data write is permitted, then GCS write is permitted.

See also Stage 2 TopLevel checks.

I_{WMGMR} The stage 2 Base permissions, S2BasePerm, are determined by S2PIR_EL2.

R_{CYQGN} The following shows how the 4-bit PIIndex field in stage 2 Block descriptors and Page descriptors is used to index into S2PIR_EL2 to obtain S2BasePerm:

- S2BasePerm is S2PIR_EL2[(PIIndex*4 + 3): PIIndex*4]

I_{XLRNP} For information on the S2BasePerm encodings, see Table D8-82.

R_{ZCKWH} S2PIR_EL2 is permitted to be cached in a TLB, and therefore so is S2BasePerm.

D8.4.2.2.2 Stage 2 Overlay permissions

R_{QNYFF} If FEAT_S2POE is not implemented, then stage 2 Overlay permissions are disabled.

R_{MMYHQ} If stage 2 Indirect permissions are disabled, then stage 2 Overlay permissions are disabled.

I_{XFWQD} If FEAT_S2POE is implemented and stage 2 uses Indirect permissions, then VTCR_EL2.S2POE determines whether stage 2 Overlay permissions are enabled.

I_{SXTQD} The value of VTCR_EL2.S2POE is not permitted to be cached in a TLB. However, the effect of this field is permitted to be cached in a TLB to a limited extent, as described later in this section.

I_{WWPGP} The stage 2 Overlay permissions, S2OverlayPerm, are determined by S2POR_EL1.

R_{QPQFD} The following shows how the 4-bit POIndex field in stage 2 Block descriptors and Page descriptors is used to index into S2POR_EL1 to obtain S2OverlayPerm:

- S2OverlayPerm is S2POR_EL1[(POIndex*4 + 3): POIndex*4]

I_{LCWFG} For information on the S2OverlayPerm encodings, see Table D8-82.

R_{LJKKX} S2OverlayPerm is not permitted to be cached in a TLB, other than in the following limited case. If all of the following apply, then the effect of S2OverlayPerm is permitted to be cached in a TLB to a limited extent:

- Stage 1 and stage 2 translations are enabled.
- Stage 2 Overlay permissions are enabled.
- For the fetch of a stage 1 translation table entry, the stage 2 translation and S2OverlayPerm permit access to that stage 1 translation table entry.
- That stage 1 translation table entry is permitted to be cached in a TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The *limited extent* is that a subsequent change to S2POR_EL1 is not required to affect the use of that stage 1 translation table entry unless it has been explicitly invalidated from the TLB. This applies to each stage 1 translation table entry of a multi-level translation table walk.

I_{SRYDP} POIndex is permitted to be cached in a TLB.

I_{XHYZM} When VTCR_EL2.{HWU*nn*, S2POE} transitions from {1, 0} to {0, 1} or to {1, 1}, or vice-versa, without a change of VMID or appropriate TLB maintenance, the resulting stage 2 PBHA attributes and stage 2 permissions are CONSTRAINED UNPREDICTABLE, as described in CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

D8.4.2.2.3 Stage 2 permission encodings

R_{CXFCS} The following table shows the stage 2 Base permissions and Overlay permissions that are applied as determined by the corresponding other S2BasePerm and S2OverlayPerm values:

Table D8-82 Determination of stage 2 Base permissions and Overlay permissions

S2BasePerm / S2OverlayPerm	Stage 2 permissions
0b0000	NoAccess
0b0001	Reserved, treated as NoAccess
0b0010	MRO
0b0011	MRO-TL1
0b0100	WO
0b0101	Reserved, treated as NoAccess
0b0110	MRO-TL0
0b0111	MRO-TL01
0b1000	RO
0b1001	RO and uX
0b1010	RO and pX
0b1011	RO and puX
0b1100	RW
0b1101	RW and uX
0b1110	RW and pX
0b1111	RW and puX

D8.4.2.2.4 Combining stage 2 Base permissions and Overlay permissions

I_{YHPCG} S2BasePerm and S2OverlayPerm are combined to produce the stage 2 permissions, S2Perm.

R_{HPVST} If all of the following apply, then S2Perm gets the value of S2BasePerm:

- Stage 2 Indirect permissions are enabled.
- Stage 2 Overlay permissions are disabled.

R_{XLNQX} If all of the following apply, then the stage 2 permission attributes, {TopLevel0, TopLevel1} are {0, 0}, and S2Perm is determined by the bitwise-AND of the encodings of S2BasePerm and S2OverlayPerm:

- Stage 2 Indirect permissions are enabled.
- Stage 2 Overlay permissions are enabled.
- S2BasePerm are stage 2 General permissions.
- S2OverlayPerm are stage 2 General permissions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SPGVL} If all of the following apply, then S2Perm is determined by S2BasePerm and the corresponding S2OverlayPerm, as shown in Table D8-83:

- Stage 2 Indirect permissions are enabled.
- Stage 2 Overlay permissions are enabled.
- S2BasePerm are stage 2 Special permissions.
- S2OverlayPerm are stage 2 Special permissions.

Table D8-83 Determination of stage 2 permissions from Special permissions

S2BasePerm / S2OverlayPerm	S2OverlayPerm / S2BasePerm	S2Perm
WO	WO	WO
WO	MRO, MRO-TL0, MRO-TL1, or MRO-TL01	NoAccess
MRO	MRO	MRO
MRO-TL0	MRO or MRO-TL0	MRO-TL0
MRO-TL1	MRO or MRO-TL1	MRO-TL1
MRO-TL0	MRO-TL1	MRO-TL01
MRO-TL01	MRO, MRO-TL0, MRO-TL1, or MRO-TL01	MRO-TL01

R_{GZNFT} If all of the following apply, then S2Perm is determined by S2BasePerm and the corresponding S2OverlayPerm, as shown in Table D8-84:

- Stage 2 Indirect permissions are enabled.
- Stage 2 Overlay permissions are enabled.
- Only one of S2BasePerm or S2OverlayPerm are stage 2 General permissions.
- The corresponding other S2BasePerm or S2OverlayPerm are stage 2 Special permissions.

Table D8-84 Determination of stage 2 permissions from General and Special permissions

S2BasePerm / S2OverlayPerm	S2OverlayPerm / S2BasePerm	S2Perm
MRO, MRO-TL0, MRO-TL1, and MRO-TL01	NoAccess	NoAccess
WO	NoAccess	NoAccess
MRO, MRO-TL0, MRO-TL1, and MRO-TL01	RO, optionally with any of uX, pX, puX	RO
MRO	RW, optionally with any of uX, pX, puX	MRO
MRO-TL0	RW, optionally with any of uX, pX, puX	MRO-TL0
MRO-TL1	RW, optionally with any of uX, pX, puX	MRO-TL1
MRO-TL01	RW, optionally with any of uX, pX, puX	MRO-TL01
WO	RO, optionally with any of uX, pX, puX	NoAccess
WO	RW, optionally with any of uX, pX, puX	WO

R_{SYLGW} If S2BasePerm does not permit Privileged Execute and does not permit Unprivileged Execute, then stage 2 Indirect permissions are execute-never.

R_{ZBXQM} For stage 2 Realm EL1&0 translations, if the stage 2 OA is not in the Realm PA space, then the Effective value of S2BasePerm removes MMU read, MMU write, and execute permissions.

I_{PRXVM} For any stage 2 translation, if all of the following apply, then the Effective value of S2BasePerm removes MMU read and MMU write permissions:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The combined memory attributes result in the Device memory type.
- HCR_EL2.PTW is 1, enabling Protected Table Walk.

I_{PDJWZ} For the cases described in R_{ZBXQM} and I_{PRXVM}, the Effective value of S2BasePerm can be determined by removing these permissions from the Stage 2 Base permissions and caching the result in a TLB, and combining the result with the Stage 2 Overlay permissions, if enabled.

R_{RRNDT} If all of the following are true, then a stage 2 Permission fault due to Overlay is generated:

- Stage 2 Overlay permissions are enabled and an access is not permitted by S2OverlayPerm,
- The access does not generate a Permission fault due to the Stage 2 TopLevel checks,

I_{DLDFD} For more information on the priority of a stage 2 Permission fault due to Overlay with respect to other Permission faults, see Prioritization of Permission faults.

D8.4.3 Effect of both stage 1 and stage 2 on data access permissions

I_{QGWYR} The combination of both the stage 1 and stage 2 permissions determine the final data access permissions.

R_{KKXMY} If both stage 1 and stage 2 translations are enabled, then in the absence of other higher-priority MMU faults, the data access permissions are combined in one of the following ways:

1. If data access to memory is not permitted by the stage 1 permissions, then a stage 1 Permission fault is generated, regardless of the stage 2 permissions.
2. If data access to memory is permitted by the stage 1 permissions, but is not permitted by the stage 2 permissions, then a stage 2 Permission fault is generated.
3. If data access to memory is permitted by both the stage 1 permissions and the stage 2 permissions, then a Permission fault is not generated.

For more information, see MMU fault prioritization from a single address translation stage.

I_{JYHLR} If EL2 is enabled, then all of the following apply to the EL1&0 translation regime:

- For stage 1 translations, all of the following apply:
  - The stage 1 translation is configured and controlled from EL1.
  - Stage 1 MMU faults are taken to EL1.
- For enabled stage 2 translations, all of the following apply:
  - The stage 2 translation is configured and controlled from EL2.
  - The stage 2 data access permissions do not differentiate between accesses at EL1 or EL0.
  - Software executing at EL2 can assign a write-only permission to a memory region.
  - Stage 2 MMU faults are taken to EL2.

I_{BLVHL} Stage 1 Permission faults are reported with a higher priority than stage 2 Permission faults.

I_{FNZDL} For a virtualization implementation, a hypervisor can define the stage 2 translation permissions to be more restrictive than the stage 1 translation permissions defined by a Guest OS. The final access permissions are the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

I_{SZJGX} The effects of combining access permissions defined by the hypervisor are expected to be functionally transparent to the Guest OS.

I_{SMZSR} For more information, see MMU fault prioritization from a single address translation stage.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.4.4 Effects on instruction execution permissions and restrictions on instruction fetch

I_{CSNFS} The descriptors contain fields used to control instruction execution permissions in stage 1 and stage 2 translations.

R_{SKGLB} If instruction execution is not permitted by one of the following, then the memory region is execute-never:

- Stage 1 Direct permissions are in use, and XN is 1, or both UXN and PXN are 1.
- Stage 1 Indirect permissions are in use and mark the region as execute-never.
- Stage 2 Direct permissions are in use, and XN is 1, or both UXN and PXN are 1.
- Stage 2 Indirect permissions are in use and the Base permissions mark the region as execute-never.

See also R_{RSXJG} and R_{SYLGW}.

R_{BFDZQ} If a memory region is execute-never at the current Exception level, then speculative instruction fetch from that memory region is prohibited.

R_{DYKWD} If a memory region is execute-never at the current Exception level, then when an attempt to execute from that memory region occurs, a Permission fault is generated at the first translation stage that determines the region is execute-never.

I_{PTMPK} If Device memory is not execute-never, then speculative instruction fetch from that memory region is permitted.

I_{ZZSZZ} For all Exception levels from which a read-sensitive memory region can be accessed, to avoid the possibility of a speculative fetch affecting that memory region, software is expected to define that memory region as execute-never.

R_{WRCYB} Except for the purposes of PSTATE.PAN, the descriptor execute-never fields do not prevent data accesses to the memory region translated by the descriptor.

D8.4.4.1 Effect of both stage 1 and stage 2 on instruction access permissions

I_{MLMFS} Both the stage 1 and stage 2 permissions determine the final instruction access permissions.

I_{NYXWS} If both stage 1 and stage 2 translations are enabled, then in the absence of other higher-priority MMU faults, the instruction execution permissions are combined in one of the following ways:

1. If instruction execution is not permitted by the stage 1 permissions, then a stage 1 Permission fault is generated, regardless of the stage 2 permissions.
2. If instruction execution is permitted by the stage 1 permissions, but is not permitted by the stage 2 permissions, then a stage 2 Permission fault is generated.
3. If instruction execution is permitted by both the stage 1 permissions and the stage 2 permissions, then a Permission fault is not generated.

For more information, see MMU fault prioritization from a single address translation stage.

D8.4.4.2 Restriction on Secure instruction fetch

I_{SBXWZ} The SCR_EL3.SIF bit can be used to prevent instruction execution from Non-secure memory at stage 1 when the PE is executing in Secure state.

R_{FNVTF} If all of the following apply, then an attempt to execute an instruction fetched from memory defined by the stage 1 translation as Non-secure generates a Permission fault:

- The Effective value of SCR_EL3.SIF is 1.
- Execution is in Secure state.

R_{WWBFB} If FEAT_RME is implemented, then the SCR_EL3.SIF bit has no effect on execution in EL3.

R_{ZWRVD} During execution at EL3, if FEAT_RME is implemented, then any attempt to execute an instruction fetched from physical memory other than the Root physical address space causes a Permission fault.

I_{LNCZM} SCR_EL3.SIF does not prevent speculative instruction fetch. If Secure software wants to prevent speculative instruction fetch from a memory region, including Non-secure mappings of that region, then that software needs to define the region as Device, XN.

I_{SQGVJ} SCR_EL3.SIF is permitted to be cached in a TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{KYSPD} TLB entries might reflect the value of SCR_EL3.SIF, therefore any change to the value of that bit requires synchronization and TLB invalidation.

R_{JDPXL} For an implementation that does not implement EL3, the Effective value of SCR_EL3.SIF is 0.

D8.4.4.3 Restriction on Realm instruction fetch

R_{PKTDS} If execution is using the Realm EL2 or Realm EL2&0 translation regime, then an attempt to execute an instruction fetched from physical memory other than the Realm PA space generates a stage 1 Permission fault.

R_{HGXXY} For the purpose of PAN, if FEAT_PAN3 is implemented, then it is IMPLEMENTATION DEFINED whether a stage 1 translation in the Realm EL2&0 translation regime that resolves to a Non-secure address is treated as Unprivileged execute-never.

R_{YMCSL} If execution is using the Realm EL1&0 translation regime, then any attempt to execute an instruction fetched from physical memory other than the Realm PA space generates a stage 2 Permission fault.

I_{MQQXW} If stage 2 translation is disabled in the Realm EL1&0 translation regime, then all OAs are in the Realm PA space and Permission faults due to R_{YMCSL} can never occur.

D8.4.5 Effect of PSTATE on access permission

I_{LGNSH} PSTATE.PAN affects memory access permissions decoded from memory access control fields in the descriptors. PSTATE.UAO affects the privilege level of a memory access. The PSTATE.BTYPE field is part of the determination of whether a branch access to a guarded memory region generates a Branch Target exception.

For more information on PSTATE, see Process state, PSTATE.

D8.4.5.1 PSTATE.PAN

I_{KTCFZ} PSTATE.PAN is used to prevent privileged data accesses to virtual memory addresses that are accessible from EL0.

R_{NCMTC} All statements in this section require implementation of FEAT_PAN.

R_{TBYDZ} For a translation regime, if one or more of the following apply, then the PSTATE.PAN bit has no effect in that regime:

- Stage 1 translation is disabled.
- Stage 1 translation supports a single privilege level.

R_{XTSQH} If the Effective value of PSTATE.PAN is 1, then a privileged data access from any of the following Exception levels to a virtual memory address that is accessible to data accesses at EL0 generates a stage 1 Permission fault:

- A privileged data access from EL1.
- If the Effective value of HCR_EL2.E2H is 1, then a privileged data access from EL2.

R_{DGQQK} If the value of PSTATE.PAN is 0, then a privileged data access is permitted from the corresponding privileged Exception level to a virtual memory address that is accessible at EL0.

I_{ZBRBM} When taking an exception from an EL in AArch64 state to an ELx, PSTATE.PAN is saved to and restored from SPSR_ELx.PAN.

I_{PWXVY} When taking an exception from an EL in AArch32 state to an ELx in AArch64 state, CPSR.PAN is saved to and restored from SPSR_ELx.PAN.

I_{FNLWP} When entering and exiting Debug state, PSTATE.PAN is saved to and restored from DSPSR_EL0.PAN.

I_{RTHGD} When an exception occurs, one of the following determines whether PSTATE.PAN is set to 1 or the value of PSTATE.PAN is unchanged:

- When taking an exception to EL1, SCTLR_EL1.SPAN.
- When taking an exception to EL2, SCTLR_EL2.SPAN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{ZGCNT} If FEAT_PAN3 is implemented, PSTATE.PAN is 1, and unprivileged instruction execution is permitted to a VA, then all of the following prevent privileged data accesses to that VA:

- For a data access from EL1, when SCTLR_EL1.EPAN is 1, a Permission fault is generated.
- For a data access from EL2, when the Effective value of HCR_EL2.E2H is 1 and SCTLR_EL2.EPAN is 1, a Permission fault is generated.

R_{WHHYZ} For the purpose of PAN, if FEAT_PAN3 is implemented, then it is IMPLEMENTATION DEFINED whether SCR_EL3.SIF is used to determine instruction execution permission.

I_{HKGQP} If FEAT_PAN3 is implemented, then the SCTLR_EL1.EPAN and SCTLR_EL2.EPAN bits affect the AT S1E1RP and AT S1E1WP instructions, consistent with those instructions using PSTATE.PAN to determine whether the memory access generates a Permission fault.

R_{PMTWB} The PSTATE.PAN bit has no effect on all of the following:

- Instruction fetches.
- Data cache instructions, except DC ZVA, DC GZVA, and DC GVA.
- If FEAT_PAN2 is not implemented, then address translation instructions.
- If FEAT_PAN2 is implemented, then the address translation instructions other than AT S1E1RP and AT S1E1WP.

I_{YMLJD} If the current Exception level is EL1 and the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then PSTATE.PAN is treated as 0 for all purposes except reading the bit value.

I_{ZRSWJ} Software can access PSTATE.PAN using all of the following instructions:

- Software can use MSR (immediate) `PAN, #Imm4` to modify PSTATE.PAN.
- Software can use MSR (register) `PAN, Xt` to modify PSTATE.PAN.
- Software can use MRS `Xt, PAN` to read PSTATE.PAN.

I_{GYGGV} If the PE is in Debug state, then a debugger can use the DRPS instruction to modify PSTATE.PAN.

D8.4.5.2 PSTATE.UAO

I_{WZGDY} PSTATE.UAO is used to control whether, for unprivileged memory access instructions, memory accesses described as unprivileged by the instruction are treated as privileged or unprivileged.

R_{NMGWS} All statements in this section require implementation of FEAT_UAO.

R_{ZVNJS} Unprivileged memory access instructions are all of the following:

- The instructions described in Load/store unprivileged.
- If FEAT_LSUI is implemented, the instructions described in the following sections:
  - Unprivileged atomic memory operations.
  - Swap unprivileged.
  - Compare and Swap unprivileged.
  - Load-Acquire/Store-Release unprivileged.
  - Load-Exclusive/Store-Exclusive unprivileged.
  - Load/store non-temporal pair unprivileged.
  - Load/store pair unprivileged.
  - Load/store unprivileged SIMD and floating-point register pair.
  - Load/store unprivileged SIMD and floating-point non-temporal pair.
- If FEAT_MOPS is implemented, the instructions described as generating either unprivileged reads or unprivileged writes or both in Memory Copy and Memory Set instructions.
- If FEAT_GCS is implemented, `GCSSTTR`.

I_{CVRWF} When an exception occurs, all of the following apply to PSTATE.UAO:

- When taking an exception to ELx, PSTATE.UAO is saved to SPSR_ELx.UAO and then set to 0.
- When returning from an exception at ELx, PSTATE.UAO is restored from SPSR_ELx.UAO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{PHDNT} If the PE is in AArch32 state, then when an exception occurs and the exception is taken to AArch64 state, all of the following occur:

- PSTATE.UAO is set to 0.
- The corresponding SPSR_ELx.UA0 is set to 0.

I_{QWJDP} When entering and exiting Debug state, PSTATE.UAO is saved to and restored from DSPSR_EL0.UAO.

I_{JQRCG} Software can access PSTATE.UAO using all of the following instructions:

- Software can use MSR (immediate) `UAO, #Imm4` to modify PSTATE.UAO.
- Software can use MSR (register) `UAO, Xt` to modify PSTATE.UAO.
- Software can use MRS `Xt, UAO` to read PSTATE.UAO.

I_{VCMLT} If the PE is in Debug state, a debugger can use the DRPS instruction to modify PSTATE.UAO.

D8.4.5.3 PSTATE.BTYPE

I_{YYZDP} PSTATE.BTYPE is part of a mechanism used to guard memory pages against executing instructions that are not the intended target of a branch. See also Restrictions on the effects of speculation from Armv9.5.

R_{JJMBN} All statements in this section require implementation of FEAT_BTI.

R_{YWFHD} When an instruction completes execution, the value of the PSTATE.BTYPE field is determined by all of the following, as shown in the following table:

- The instruction that was executed.
- Whether or not the memory region is guarded.
- The register accessed by the instruction.

Table D8-85 Values taken by PSTATE.BTYPE on execution of an instruction

Instruction executed	Memory region	Register accessed	PSTATE.BTYPE
`BR`, `BRAA`, `BRAAZ`, `BRAB`, `BRABZ`	Guarded	Any register except X16 or X17	`0b11`
`BLR`, `BLRAA`, `BLRAAZ`, `BLRAB`, `BLRABZ`	Any	Any register	`0b10`
`BR`, `BRAA`, `BRAAZ`, `BRAB`, `BRABZ`	Guarded	X16 or X17	`0b01`
`BR`, `BRAA`, `BRAAZ`, `BRAB`, `BRABZ`	Non-guarded	Any register	`0b01`
`ERET`, `ERETAA`, `ERETAB`, and `DRPS`	Any	Any register	SPSR_ELx.BTYPE
Any other instruction	Any	Any register	`0b00`

I_{CNBPL} When the BTI instruction is executed, the `<targets>` operand determines the compatibility between the BTI instruction and different PSTATE.BTYPE values as shown in the following table:

Table D8-86 Compatibility of BTI instruction to different PSTATE.BTYPE values

<targets>	PSTATE.BTYPE value			
	`0b00`	`0b01`	`0b10`	`0b11`
(omitted)	N/A	Not compatible	Not compatible	Not compatible
c	N/A	Compatible	Compatible	Not compatible
j	N/A	Compatible	Not compatible	Compatible

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<targets>	PSTATE.BTYPE value			
	0b00	0b01	0b10	0b11
jc	N/A	Compatible	Compatible	Compatible

I_{JRBGR} If stage 1 translation is enabled, the GP bit in Block and Page descriptors indicates whether the memory region is guarded. For more information, see FEAT_BTI.

If stage 1 translation is disabled, no memory regions are guarded. For more information, see Behavior when stage 1 address translation is disabled.

R_{LJHCL} If the PSTATE.BTYPE field is not 0b00 and an attempt is made to execute an instruction within a guarded page, a Branch Target exception is generated unless the instruction is one of the following:

- A BTI instruction that is compatible with the PSTATE.BTYPE field.
- A PACIASP, PACIBSP, PACIASPPC, or PACIBSPPC instruction, and PSTATE.BTYPE is consistent with implicit BTI behavior of these instructions.

I_{ZJQDF} The Software Breakpoint instruction and, if Halting is enabled and allowed, the Halt instruction always generate a higher priority exception or debug event than a Branch Target exception.

R_{VWTMW} When a Branch Target exception occurs, it is taken to one of the following:

- When executing at EL0 and HCR_EL2.TGE is 0, the Branch Target exception is taken to EL1.
- When executing at EL0 and HCR_EL2.TGE is 1, the Branch Target exception is taken to EL2.
- When executing at ELx, where x is 1, 2, or 3, the Branch Target exception is taken to ELx.

I_{VWNCG} When a Branch Target exception occurs, ESR_ELx.EC is 0x0D.

For more information, see ISS encoding for an exception from Branch Target Identification instruction.

I_{GMGRS} When any of the following are true, the PACIASP, PACIBSP, PACIASPPC, and PACIBSPPC instructions have an implicit BTI behavior that is compatible with the same PSTATE.BTYPE values as a BTI.jc instruction:

- The PE is executing at EL0, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and SCTLR_EL1.BT0 is 0.
- The PE is executing at EL0, the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and SCTLR_EL2.BT0 is 0.
- The PE is executing at EL1, and SCTLR_EL1.BT1 is 0.
- The PE is executing at EL2, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and SCTLR_EL2.BT is 0.
- The PE is executing at EL2, the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and SCTLR_EL2.BT1 is 0.
- The PE is executing at EL3, and SCTLR_EL3.BT is 0.

Otherwise, these instructions have an implicit BTI behavior that is compatible with the same PSTATE.BTYPE values as BTI.c.

R_{DFHDM} The implicit branch target identification property of PACIASP, PACIBSP, PACIASPPC, and PACIBSPPC is independent of the setting of the SCTLR_ELx.{EnIA, EnIB} bits.

I_{CKJFH} In a non-guarded page, the BTI instruction executes as a NOP.

I_{WCDBN} The effect of a NOP on PSTATE.BTYPE is described in R_{YWFHD}.

I_{CPYQW} There is no mechanism for direct reads or direct writes of the PSTATE.BTYPE field.

D8.4.6 Controlling memory access Security state

I_{RLRVY} The Non-secure, NS, bit in Block descriptors and Page descriptors, and the Non-secure table, NSTable, bit in Table descriptors are used to provide control over accesses from Secure state to Secure and Non-secure memory.

R_{PRTVS} In this section and subsections, *IPA or PA space* refers to the OA space of stage 1 translations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{KZLVM} For a Secure translation regime, if the translation table entry is in Secure IPA or PA space, then the Block descriptor and Page descriptor NS determines all of the following:

- If NS is 0, an access to the OA specified by the descriptor is to Secure IPA or PA space.
- If NS is 1, an access to the OA specified by the descriptor is to Non-secure IPA or PA space.

R_{LXLSC} For all of the following, if a Block or Page descriptor has NS set to 1, then the OA is in the Non-secure PA space. Otherwise, the OA is in the Realm PA space:

- Stage 1 translation in the Realm EL2 translation regime.
- Stage 1 translation in the Realm EL2&0 translation regime.
- Stage 2 translation in the Realm EL1&0 translation regime.

R_{QMLYQ} If the stage 2 translation for a Realm stage 1 translation table walk resolves to an address not in the Realm PA space, it causes a stage 2 Permission fault.

R_{VZYQL} For a Non-secure translation regime, the NS and NSTable bits are RES0 and the OA or next-level table base address is in Non-secure IPA or PA space.

R_{FKJXR} For a Secure translation regime, when the descriptor is fetched from Non-secure IPA or PA space, all of the following apply to the descriptor:

- The descriptor NS and NSTable bits exist but they are SBZ instead of RES0.
- The OA or next-level table base address specified by the descriptor is in Non-secure IPA or PA space.

R_{QSJJS} For the EL3 stage 1 translation regime, if FEAT_RME is not implemented, then the NS field in a Block descriptor or Page descriptor determines the output PA space as shown in the following table:

Table D8-87 Output PA space

NS	Output PA space
0	Secure.
1	Non-secure.

R_{XTYPW} For the EL3 stage 1 translation regime, if FEAT_RME is implemented, then the NSE and NS fields in a Block descriptor or Page descriptor determine the output PA space as shown in the following table:

Table D8-88 Output PA space

NSE	NS	Output PA space
0	0	If FEAT_SEL2 is implemented, then Secure. Otherwise, Non-secure.
0	1	Non-secure.
1	0	Root.
1	1	Realm.

D8.4.6.1 Hierarchical control of Secure or Non-secure memory accesses

R_{WGTQD} For a Secure translation regime, when a Table descriptor is accessed from a stage 1 translation table in Secure IPA or PA space, the NSTable field determines all of the following:

- If NSTable is 0, the next-level translation table address in the Table descriptor is in Secure IPA or PA space.
- If NSTable is 1, the next-level translation table address in the Table descriptor is in Non-secure IPA or PA space.

R_{YPMZG} If the next-level translation table address in the Table descriptor is in Non-secure IPA or PA space, then the address specified by a descriptor in the next lookup-level translation table is in Non-secure IPA or PA space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{XGNKR} If stage 2 translation is enabled, the VSTCR_EL2.SA, VTCR_EL2.NSA, VSTCR_EL2.SW, and VTCR_EL2.NSW fields can map an IPA space to a PA space not matching the Security of the IPA space.

R_{KRBNK} If all of the following apply, then a stage 1 translation is treated as non-global, meaning the Effective value of nG is 1, regardless of the actual value of the Block descriptor or Page descriptor nG bit:

- The stage 1 translation supports two privilege levels.
- The PE is in Secure state.
- NSTable is 1 at any level of the translation table walk.

For more information, see Global and process-specific translation table entries.

I_{TKPLZ} The descriptor NSTable field affects all subsequent lookup levels and the translation IPA or PA space. When an NSTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.

For more information, see TLB maintenance and Using break-before-make when updating translation table entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.5 Hardware updates to the translation tables

I_{FNBGK} All of the following determine whether hardware can update the descriptor access flag and dirty state:

- The features implemented and enabled by the PE.
- The descriptor type.
- The translation regime.
- The translation stage.

I_{CGGMJ} Hardware updates to the translation tables are subject to memory ordering requirements and restrictions on the memory types.

D8.5.1 The Access flag

R_{XFXTY} The AF in a Block descriptor and Page descriptor indicates one of the following:

- If the value is 0, then the memory region has not been accessed since the value of AF was last set to 0.
- If the value is 1, then the memory region has been accessed since the value of AF was last set to 0.

R_{DWZCQ} Descriptors with AF set to zero can never be cached in a TLB.
For more information about when translation table entries are permitted to be cached in a TLB, see Translation Lookaside Buffers.

D8.5.1.1 Software management of the Access flag

I_{CDQDG} If the AF is not managed by hardware, software management of the AF is required.

R_{SWTYY} For an implementation that does not manage the AF in hardware, when an attempt is made to translate an address using a descriptor with an AF of 0, an Access flag fault is generated.

I_{FDYZF} For an implementation that does not manage the AF in hardware, when an Access flag fault is generated, software is expected to set the AF to 1 in the descriptor that generated the fault.

I_{KQMYT} Setting the AF to 1 prevents the Access flag fault from being generated the next time an attempt is made to translate an address using that descriptor.

I_{HQBRF} When software sets the AF to 1, there is no requirement to perform TLB invalidation after setting the flag because entries with an AF set to 0 are never held in a TLB.

I_{BMRWH} If a system incorporates components that can autonomously update translation table entries that are shared with the PE, then any changes by system software to translation table entries with an AF of 0 should avoid the possibility of simultaneous updates. This excludes changes to the AF value. For example, this can be done by using a Load-Exclusive/Store-Exclusive loop or an atomic compare-and-swap operation.

D8.5.1.2 Hardware management of the Access flag

R_{WMGCW} All statements in this section require implementation of the OPTIONAL feature, FEAT_HAF.

R_{LFTXR} When the PE performs a hardware update of the AF, it sets the AF to 1 in the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor.

R_{HDHQG} For a translation stage, if the value of the corresponding TCR_ELx.HA is 1, then AF hardware management is enabled.

I_{BDRJG} For each translation stage, AF hardware management can be enabled independently.

R_{HDTRB} If AF hardware management is enabled, then hardware updates the AF when a memory access is made using a Block descriptor or Page descriptor and all of the following apply:

- The access does not generate an Alignment fault based on the memory type, or a Permission fault.
- If AF hardware management was disabled or not implemented, the access would have generated an Access flag fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{QSPMS} If AF hardware management is enabled, then it is CONSTRAINED UNPREDICTABLE whether hardware updates the AF when a memory access is made using a Block descriptor or Page descriptor and all of the following apply:

- The access generates an Alignment fault based on the memory type, a Permission fault, or any fault that has a lower priority than a Permission fault from that stage of translation.
- If AF hardware management were disabled or not implemented, the access would have generated an Access flag fault.

I_{RGQLZ} If hardware does not update the AF from 0 to 1, then the descriptor is not permitted to be cached in a TLB.

R_{VNHWC} If AF hardware management is enabled in one or both translation stages, then when a memory access occurs, one or more of the following can be updated:

- The stage 1 AF.
- Each of the stage 2 AFs used to translate accesses to the stage 1 translation table walk and the OA.

I_{CVNQR} For stage 1 translations, if AF hardware management is enabled, then when stage 2 translation does not permit access to the OA returned by the stage 1 translation, it is permitted, but not required, for the stage 1 AF to be updated.

I_{MYQJN} For stage 2 translations, if AF hardware management is enabled, then when stage 1 translation does not permit access, it is permitted, but not required, for the stage 2 AF to be updated.

R_{LHQRX} If AF hardware management is enabled and a speculative access is successfully translated, then it is permitted, but not required, for hardware to update the AF.

R_{GGYFS} When a speculative access occurs, the rules determining the AF hardware update apply.

I_{GXGPN} When the translation of an architecturally executed memory access occurs, the architecture requires that AF is set to 1.

I_{LLCCB} When a translation occurs, the AF is permitted to be set to 1 even in the case where no architecturally executed memory access occurs. This could be due to a speculative memory access.

I_{PJWQY} When hardware updates of the Access Flag are enabled, it is permitted to update the Access Flag speculatively. This is not affected by the granule protection check on the output address of the translation.

R_{XQSMX} For an address translation stage, if AF hardware management is enabled, then when an address translation instruction is executed, all of the following apply:

- Hardware is permitted, but not required, to update the AF in the Block or Page descriptor that resolves the translation.
- If hardware attempts to update the AF and the update generates a fault, it is IMPLEMENTATION DEFINED whether this is reported as a Data Abort or not.
- If a Data Abort does not occur, then the instruction reports a result in the PAR as if the descriptors used to translate the address have an AF value of 1.

I_{ZWQRX} For more information, see Using break-before-make when updating translation table entries.

I_{KJBFT} If an implementation does not generate Access flag faults for cache maintenance by VA instructions, it is not required to perform hardware updates of AF for those cache maintenance instructions.
See also MMU faults generated by cache maintenance operations.

D8.5.1.3 Hardware management of the Table descriptor Access Flag

R_{LWKQH} All statements in this section require implementation of FEAT_HAFT.

R_{PJFRD} All statements in this section apply when the translation stage uses VMSAv8-64 or VMSAv9-128. They do not apply when the translation stage uses VMSAv8-32.

I_{HZHJT} Software management of the Table descriptor AF is not supported, and no Access Flag fault is generated from Table descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{FWSJH} For a translation stage, the *Hardware managed Access Flag for Tables* (HAFT) value is determined according to the following table:

Table D8-89 Determination of HAFT value

Translation stage	Translation regime	HAFT value
Stage 1	EL1&0	TCR2_EL1.HAFT
	EL2&0	TCR2_EL2.HAFT
	EL2	TCR2_EL2.HAFT
	EL3	TCR_EL3.HAFT
Stage 2	EL1&0	VTCR_EL2.HAFT

R_{SNVTX} For a translation stage, if the value of TCR_ELx.HA is 0, then the Effective value of the corresponding HAFT is 0.

R_{HTTKB} The Effective value of HAFT is permitted to be cached in a TLB.

R_{YCTVZ} The HAFT value determines whether the Table descriptor AF is managed by hardware, as shown in the following table:

Table D8-90 Meaning of HAFT value

HAFT value	Meaning
0	There is no AF in Table descriptors.
1	Hardware management of the Table descriptor AF is enabled.

I_{PDQYR} Table descriptor AF hardware management can be enabled for each translation stage independently.

R_{BCWSB} When the PE performs a hardware update of the Table descriptor AF, it sets the AF to 1 in the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor.

R_{ZGNSH} If Table descriptor AF hardware management is enabled at a translation stage, then a speculative memory access is permitted to cause a hardware update to set the AF to 1 in any Table descriptor that is accessed during the translation table walk for the access and for which the AF is 0.

R_{MDMYP} When a speculative access occurs, the rules determining the Table descriptor AF hardware update apply.

R_{CGKWB} For an architecturally executed memory access translated by a translation stage with Table descriptor AF hardware management enabled, the PE is required to set the AF to 1 in all Table descriptors accessed during the translation table walk for that access that have the AF set to 0.

R_{FBZPR} When an address translation instruction is executed, for a translation stage with Table descriptor AF hardware management enabled, the PE is required to set AF to 1 in all Table descriptors accessed during the translation table walk that have the AF set to 0.

R_{XYNNS} For a translation stage with Table descriptor AF hardware management enabled, if the AF is updated in a Table descriptor, Block descriptor, or Page descriptor, then all Table descriptors that are accessed from memory leading up to that descriptor in the translation table walk are required to either already have AF set to 1 or update AF to 1.

I_{QYLZN} For information on the memory ordering requirements and restrictions on the memory types when hardware updates the Table descriptor AF, see Hardware updates to the translation tables.

I_{VPNMG} Updates to the stage 1 Table descriptors are subject to stage 2 permission checks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{JFCYB} For a translation stage with Table descriptor AF hardware management enabled, a Table descriptor with AF set to 0 is not permitted to be cached in a TLB.

D8.5.2 The dirty state

I_{HTJWZ} The dirty state is used to indicate a memory block or page has been modified. When hardware update of the dirty state is enabled, the descriptor DBM field indicates whether the descriptor is a candidate for hardware updates of the dirty state.

R_{LRMSH} For the purpose of FEAT_HAFDBS dirty state management, a Block descriptor or Page descriptor can be described as having one of the following states:

- Non-writable.
- Writable-clean.
- Writable-dirty.

R_{XSTDV} If a Block descriptor or Page descriptor is not writable-clean and not writable-dirty, then it is described as *non-writable*.

R_{CLBGT} For a translation stage using Indirect permissions, software or hardware is permitted to manage the dirty state.

R_{DQQCV} For a translation stage using Direct permissions, only hardware is permitted to manage the dirty state.

R_{VRNVZ} For more information on managing the dirty state in hardware, see Hardware management of the dirty state.

I_{CPTSY} For a stage 1 translation using Indirect permissions at ELx, if the Effective value of TCR_ELx.HD is 0, then dirty state is managed by software.

I_{ZCPYX} For stage 2 translations using Indirect permissions, if the Effective value of VTCR_EL2.HD is 0, then dirty state is managed by software.

R_{QXCRT} For stage 1 translations using Indirect permissions that grant write permission, the following table shows the descriptor state based on the Block descriptor and Page descriptor nDirty field:

nDirty field	Descriptor state
0	Writable-dirty
1	Writable-clean

Note

The polarity of the nDirty field matches the polarity of the AP[2] field used by Direct permissions.

R_{XXBLY} For stage 2 translations using Indirect permissions that grant write permission, the following table shows the descriptor state based on the Block descriptor and Page descriptor Dirty field:

Dirty field	Descriptor state
0	Writable-clean
1	Writable-dirty

R_{PPGMC} For write accesses translated by a writable-clean descriptor, if the descriptor dirty state is managed by software, then a Permission fault due to dirty state is generated. For information on the priority of a Permission fault due to dirty state, see Prioritization of Permission faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.5.2.1 Hardware management of the dirty state

R_{QRTKG} All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS, which supports hardware update of the dirty state. FEAT_HAFDBS requires FEAT_HAF, which supports hardware update of the Access flag.

I_{FLYZP} For stage 1 translations, if the Effective value of the corresponding TCR_ELx.HD is 1, then dirty state hardware management is enabled.

I_{MHJZP} For stage 2 translations, if the Effective value of VTCR_EL2.HD is 1, then dirty state hardware management is enabled.

I_{DLNSK} If a translation stage uses Direct permissions and hardware update of the dirty state is enabled, then the descriptor DBM field indicates whether the descriptor is a candidate for hardware updates of the dirty state.

R_{PZFQC} For translations using Direct permissions, if the DBM field in a Block descriptor or Page descriptor is 0, then all of the following apply:

- For write accesses translated by the descriptor, the generation of Permission faults are unaffected by FEAT_HAFDBS.
- Hardware does not update the dirty state.

R_{XZFQH} For each translation stage using Direct permissions, if all of the following apply, then a Block descriptor or Page descriptor is described as *writable-clean*:

- Dirty state hardware update is enabled at that translation stage.
- The descriptor DBM field is 1.
- For a descriptor used by stage 1 translation, the AP[2] field is 1 and this is the only reason that a Permission fault would be generated due to a write access if dirty state hardware management were disabled.
- For a descriptor used by stage 2 translation, the S2AP[1] field is 0 and this is the only reason that a Permission fault would be generated due to a write access if dirty state hardware management were disabled.

R_{BRFGY} For each translation stage using Direct permissions, if all of the following apply, then a Block descriptor or Page descriptor is described as *writable-dirty*:

- Dirty state hardware update is enabled at that translation stage.
- The descriptor DBM field is 1.
- For a stage 1 translation, the AP[2] field is 0.
- For a stage 2 translation, the S2AP[1] field is 1.

I_{HFBCN} If a translation stage uses Indirect permissions, then dirty state hardware management is not separately determined by each Block descriptor and Page descriptor. This is different from the case where a translation stage uses Direct permissions, and the DBM field indicates whether the descriptor is a candidate for dirty state hardware updates.

I_{BFDYL} Only if a descriptor is writable-clean, then hardware can update the dirty state.

R_{TYCNM} For write accesses translated by a writable-clean descriptor, if the descriptor dirty state is managed by hardware, then a Permission fault due to dirty state is not generated.

R_{NSXRD} For write accesses translated by a writable-clean descriptor, if all of the following apply, then hardware updates that descriptor to writable-dirty:

- The descriptor dirty state is managed by hardware.
- The only reason that a Permission fault would be generated due to that write access would be if dirty state hardware management were disabled.

R_{PQJJR} For a translation stage, when the dirty state is updated by hardware, the PE updates the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor to change the descriptor from writable-clean to writable-dirty.

R_{PRHKD} If the requirements to update both the Access flag and dirty state by hardware are met, the PE is permitted to update the Access flag to 1 as part of the same atomic read-modify-write of the descriptor that performs the dirty state update.

I_{DSHLJ} For a translation stage, Access flag faults are reported with higher priority than Permission faults. Therefore, it is not permitted for hardware to update the dirty state so that a descriptor is writable-dirty but has the Access flag clear.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SGJBL} When the dirty state is updated by hardware, the PE is required to check that any copy of the descriptor cached in a TLB is not stale with regard to the descriptor in memory.

I_{YZSVV} If the descriptor has changed, then the PE is required to use the new information from the descriptor in memory, regardless of whether the modified descriptor has a different OA, different attributes, or generates a fault.

I_{WZRPC} For performance purposes, when the dirty state is updated by hardware, the PE should update or invalidate any TLB entry that contains a previously-cached copy of the descriptor. This is to prevent multiple writes by the PE from causing multiple updates to the descriptor.

I_{HQBGN} Arm expects many implementations to cause a translation table walk to occur when dirty state hardware management causes an update of a translation table descriptor.

R_{RNMLQ} If a write access translated by a writable-clean descriptor is performed architecturally, then hardware updates the dirty state of that descriptor.

R_{DYCFD} If a write access translated by a writable-clean descriptor is not performed architecturally, then unless specified here hardware does not update the dirty state of that descriptor. In all of the following cases, hardware is permitted to update the dirty state while attempting to translate the explicit write access:

- The translation generates an MMU fault that is reported with lower priority than a Permission fault at that translation stage.
- The access is non-speculative and the translation generates an Alignment fault only because the memory type indicated by the descriptor is Device, and the hardware update would have occurred if the Alignment fault was not generated.
- The descriptor is for a stage 1 translation, the access is non-speculative, and the stage 2 translation of the OA from stage 1 generates a stage 2 MMU fault.
- The descriptor is used to translate accesses from the Statistical Profiling Buffer. For more information, see The Profiling Buffer.
- The final OA from the translation generates a Granule Protection Check fault (GPC fault).
- FEAT_MTE_PERM is implemented and all of the following apply:
  - The descriptor is for a stage 2 translation.
  - The access is non-speculative.
  - The stage 2 translation generates a Permission fault due to the NoTagAccess attribute.
- FEAT_MTE_CANONICAL_TAGS is implemented and all of the following apply:
  - The descriptor is for a stage 1 translation.
  - The access is non-speculative.
  - The stage 1 translation generates a Permission fault due to a Canonical Tag Check operation.
- The descriptor is used to translate accesses from the Trace Buffer Unit. See Accesses to the trace buffer.

R_{BZXVS} If a tag write by an instruction that does not also write data is translated by a writable-clean descriptor, but the tag write effect is IGNORED due to one of the following reasons, then if all of the other requirements for a hardware update of Dirty state are met, it is CONSTRAINED UNPREDICTABLE whether hardware updates the dirty state of that descriptor:

- The stage 1 descriptor does not have the Tagged attribute.
- Allocation Tag Access is disabled for the instruction by SCR_EL3.ATA, HCR_EL2.ATA, or SCTLR_ELx.{ATA, ATA0}.

R_{BDJJL} If both translation stages are enabled, and the conditions to update a stage 1 descriptor by hardware are met, then the stage 2 translation of the location holding that stage 1 descriptor results in one of the following:

- The stage 2 translation grants write permissions and the stage 1 hardware update occurs, including both of the following cases:
  - The stage 2 translation already grants write permission and does not need to be updated.
  - The stage 2 translation is writable-clean and is successfully updated to writable-dirty.
- The stage 2 translation generates a Permission fault due to lack of write access and the stage 1 hardware update does not occur.

I_{TZTHF} If a speculative update of a stage 1 Access flag would otherwise be permitted, but the stage 2 translation of the stage 1 descriptor is read-only, then the speculative update of the stage 1 Access flag does not occur.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{JCXVS} If all of the following apply, hardware can update the dirty state of a stage 2 descriptor even if the stage 1 descriptor is not updated, including the case where the stage 1 descriptor is not a Block descriptor or Page descriptor:

- Stage 1 hardware updates of the Access flag or dirty state are enabled.
- The stage 2 translation of a stage 1 translation table walk uses a writable-clean descriptor.

R_{LXCKP} The dirty state hardware update mechanism does not affect the fault reporting priority except in all of the following cases:

- If all of the following stage 2 MMU faults are generated due to a non-speculative access that causes a stage 1 hardware update, then either fault is permitted to be reported:
  - That stage 1 hardware update generates a stage 2 MMU fault.
  - The stage 2 translation of the stage 1 OA generates a stage 2 MMU fault.
- If a hardware update is permitted, that update is permitted to generate a synchronous External abort or an IMPLEMENTATION DEFINED abort caused by a memory type not supporting an atomic read-modify-write.

R_{QDRWL} If an instruction that generates more than one single-copy atomic memory access has a fault on some, but not all, of those memory accesses, then all of the following apply:

- All accesses that do not fault are permitted to cause hardware updates of the dirty state.
- For faulting accesses that meet the requirements for hardware updates of the dirty state, those accesses are also permitted to cause hardware updates of the dirty state.

I_{BSLWF} For more information, see:

- MMU fault prioritization from a single address translation stage.
- Faults and watchpoints.

D8.5.2.2 Implications of enabling the dirty state management mechanism

R_{YWCDD} All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS, which supports hardware update of the dirty state.

R_{TTLCV} For stage 1 translations, translations using a descriptor in the writable-clean state are considered writable for the purposes of the corresponding SCTLR_ELx.WXN control.

I_{NSYYW} For stage 1 translations, if the corresponding SCTLR_ELx.WXN is 1, then all of the following apply:

- For a translation regime that supports a single privilege level, translations using a writable-clean descriptor are treated as execute-never.
- For a translation regime that supports two privilege levels, translations using a privileged writable-clean descriptor are treated as privileged execute-never.
- For a translation regime that supports two privilege levels, translations using an unprivileged writable-clean descriptor are treated as unprivileged execute-never.

R_{NKLQN} For a translation regime that supports two privilege levels, translations using an unprivileged writable-clean descriptor are treated as privileged execute-never.

R_{ZYGHJ} Cache invalidation instructions and address translation instructions never cause a dirty state hardware update of either:

- The stage 1 Block descriptor or Page descriptor translating the address specified in the instruction.
- The stage 2 Block descriptor or Page descriptor translating the OA from the stage 1 translation.

R_{MBXNP} Cache invalidation instructions and address translation instructions never cause a dirty state hardware update the stage 1 Block descriptor or Page descriptor translating the address specified in the instruction.

R_{RKMHW} For the following instructions that require write permission, if the address specified in the instruction is translated by a writable-clean descriptor and hardware management of Dirty state is enabled for the corresponding stage of translation, then the descriptor is considered to grant write access and the hardware does not update the Dirty state of that descriptor:

- Data cache invalidation instruction, DC IVAC.
- Data cache and instruction cache maintenance instructions that are affected by FEAT_CMOW.
- Address translation instructions, AT S12E0W, AT S12E1W, AT S1E0W, AT S1E1W, AT S1E2W, AT S1E3W.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{QXXXL} For all of the following cases, if the stage 1 descriptor is writable-clean, then it is IMPLEMENTATION DEFINED whether hardware updates the dirty state:

- A Store-Exclusive instruction to the memory location fails because the Exclusives monitor is not in the exclusive state.
- A GPC fault is generated on a write to the memory location.
- A synchronous External abort is generated on a write to the memory location.
- A Watchpoint exception is generated on a write to the memory location.
- A synchronous Tag check fault is generated on a write to the memory location.

R_{SJQGP} For all of the following cases, if the stage 2 descriptor is writable-clean, then it is IMPLEMENTATION DEFINED whether hardware updates the dirty state:

- A Store-Exclusive instruction to the memory location fails because the Exclusives monitor is not in the exclusive state.
- A GPC fault is generated on a write to the memory location.
- A synchronous External abort is generated on a write to the memory location.
- A Watchpoint exception is generated on a write to the memory location.
- A synchronous Tag check fault is generated on a write to the memory location.

R_{KLVVN} A hardware update of the dirty state performed by one PE does not require invalidation of any corresponding TLB entries in other PEs.

R_{TXGHB} For a `CAS` or `CASP` instruction to an address translated by a writable-clean descriptor, if the comparison fails and the location is therefore not updated, then it is CONSTRAINED UNPREDICTABLE whether hardware updates the dirty state.

R_{SMTCW} For an `RCW` or `RCWS` instruction to an address translated by a writable-clean descriptor, if the RCW checks or RCWS checks fail and the location is therefore not updated, then it is CONSTRAINED UNPREDICTABLE whether hardware updates the dirty state.

R_{KTXCF} For a stage 2 writable-clean descriptor, if an atomic instruction to the address translated by the descriptor generates a stage 2 Permission fault only as a result of not having read permission, then it is CONSTRAINED UNPREDICTABLE whether hardware updates the dirty state.

D8.5.2.3 Hardware dirty state tracking structure

I_{NWXLF} The *hardware dirty state tracking structure* (HDBSS) keeps a record of dirtied stage 2 Block and Page descriptors, enabling EL2 software to retrieve this information without scanning translation tables.

R_{RGTLS} All statements in this section require implementation of the OPTIONAL feature, FEAT_HDBSS.

I_{TMYVQ} HDBSS is enabled only if VTCR_EL2.{HDBSS, HA, HD} are {1, 1, 1}.

R_{GKFDY} If VTCR_EL2.HDBSS is 0, then all of the following apply:

- The PE does not update the HDBSS in response to a dirty state hardware update.
- The PE does not speculatively read the memory region configured by the HDBSSBR_EL2 fields.

I_{SXBYL} All of the following apply to the HDBSS base address:

- It is a PA configured in HDBSSBR_EL2.BADDR.
- For encodings of HDBSSBR_EL2.SZ greater than 4KB, bits [(SZ+12-1):12] of HDBSSBR_EL2.BADDR are RES0, such that the base address is aligned to its size.

R_{DBTZQ} The PA space of the HDBSS base address is determined by the Effective value of SCR_EL3.{NSE, NS}.

I_{JRDPM} The HDBSS size is configured in HDBSSBR_EL2.SZ.

R_{BWVKS} Each HDBSS entry is eight bytes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{QWSGD} The following table defines the fields in an HDBSS entry.

Table D8-93 Hardware dirty state tracking structure fields

Bit position	Field	Comment
[63:56]	RES0	-
[55:12]	IPA	The IPA translated by the stage 2 descriptor that was updated from writable-clean to writable-dirty. The IPA is aligned to the size of the Page or Block descriptor that was updated. Bits above the implemented physical address size are RES0.
[11]	NSIPA	For Secure state, indicates one of the following: • 0, Secure IPA. • 1, Non-secure IPA. For Realm and Non-secure states, this bit is RES0.
[10:4]	RES0	-
[3:1]	TTWL	Translation table walk level at which the descriptor has been marked dirty. This field is a 3-bit two's complement signed integer.
[0]	Valid entry	0, entry is invalid. 1, entry is valid.

R_{WZKFT} If HDBSS is enabled, then when hardware updates a stage 2 descriptor from writable-clean to writable-dirty, the PE writes to the HDBSS.

I_{RYKVL} The index of the next eight-byte HDBSS entry to write is indicated by HDBSSPROD_EL2.INDEX.

R_{HKCCN} The PE is permitted to mark HDBSS entries ahead of the HDBSSPROD_EL2.INDEX as zero.

R_{PCTTC} When the PE writes an HDBSS entry, all the following occur:

- The IPA that caused hardware to update the stage 2 descriptor dirty state, and the associated NSIPA and TTWL, are written to the HDBSS using the current HDBSSPROD_EL2.INDEX. The write is done as a 64-bit single-copy atomic operation.
- The value of HDBSSPROD_EL2.INDEX is incremented by 1.

R_{KFLNY} Accesses to the HDBSS use the same memory attributes as stage 2 translation table walks, including all of the following:

- The endianness configured by SCTLR_EL2.EE.
- The Normal memory type.
- The Cacheability and Shareability attributes configured by VTCR_EL2.{IRGN0, ORGN0, SH0}.
- If FEAT_MTE2 is implemented, accesses are always Tag Unchecked.
- If FEAT_MEC is implemented, the MECID used for stage 2 translation table walks.
- If FEAT_MPAM is implemented, the PARTID value, PMG value, and PARTID space used for stage 2 translation table walks.

I_{NMYYP} If FEAT_RME is implemented and GPC is enabled, accesses to the HDBSS are subject to Granule Protection Checks.

R_{TCJKB} If the translation table update or the HDBSS update is observable to the PE that performed the write that caused these updates, then completion of a `DSB` that applies to both loads and stores on the same PE, at an Exception level where EL1&0 is out-of-context, guarantees observability of the other update.

I_{KTVVZ} If the profiling buffer or trace buffer is enabled, further hardware updates of Dirty state may occur even if the EL1&0 translation regime is out-of-context. Disabling the buffer and executing `PSB CSYNC` or `TSB CSYNC` as appropriate, while the EL1&0 translation regime is out-of-context, before execution of a `DSB` that applies to both loads and stores, is sufficient to guarantee that all updates to the HDBSS are observable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{STDKM} If any of the following apply, then there is no guarantee of order between the two updates described, and atomicity and coherency might be lost for the writes:

- The HDBSS entry and the translation table entry to be updated are at the same location.
- The write to the HDBSS and the write access being translated target the same location.
- The PE writes any HDBSS entry, and any other write not performed by the HDBSS is performed to any location in the HDBSS buffer.

R_{CGCHP} On exception entry to ELx, where EL1&0 is out-of-context, outstanding HDBSS writes performed by the PE during execution at EL1&0 are guaranteed to be observable to memory accesses from ELx after completion of a `DSB` instruction that applies to both loads and stores.

I_{XXFYP} Hardware increments of HDBSSPROD_EL2.INDEX are indirect writes, and are therefore not guaranteed to be observable to direct reads of HDBSSPROD_EL2 until completion of a context synchronization event.

R_{RXCMY} The HDBSS is full when HDBSSPROD_EL2.INDEX is greater than or equal to $2^{(HDBSSBR_EL2.SZ+12)}/8$.

I_{LMSZL} If an HDBSS access generates an External abort or a GPC fault, and HDBSSPROD_EL2.FSC is updated to a non-zero value, then the update of the FSC field might not happen synchronously relative to some stage 2 hardware updates of the dirty state. Therefore, one or more HDBSS entries might be lost. See GPC fault on HDBSS access and External aborts on HDBSS access.

R_{YHLRY} If the HDBSS is full and enabled, then the PE does not update any stage 2 descriptors from writable-clean to writable-dirty.

R_{STJMN} If the HDBSS is enabled, and either the HDBSS is full or HDBSSPROD_EL2.FSC is set to any value other than `0b000000`, then all of the following apply:

- The PE cannot append entries to the HDBSS.
- Stage 2 dirty state hardware updates are prevented.
- Any access that requires a stage 2 dirty state hardware update generates a stage 2 Permission fault that is reported as it would be if VTCR_EL2.HD were 0. If the fault is reported in ESR_EL2, ESR_EL2.ISS2.HDBSSF is reported as 1, to indicate that the failed update was a result of the HDBSS being full or in an error state.

I_{KZMGK} MMU faults on speculative accesses are never reported.

If the HDBSS is full and enabled, and this generates a stage 2 Permission fault as part of resolving an `AT *E1*` operation executed at EL2, it is reported in PAR_EL1 as a stage 2 Permission fault on a stage 1 translation table walk, at the level of stage 2 walk that experienced the fault.

D8.5.2.3.1 GPC fault on HDBSS access

R_{LHLSD} If FEAT_RME is implemented and GPC is enabled, then when the hardware updates the stage 2 descriptor dirty state and the associated HDBSS access generates a GPC fault, all of the following apply:

- A GPC fault is reported.
- HDBSSPROD_EL2.INDEX points to the HDBSS entry that generated the fault.
- It is IMPLEMENTATION DEFINED which of the following reporting mechanisms is used:
  - A precise exception is reported as though the dirty state hardware update generated the GPC fault, and ESR_EL2.ISS2.HDBSSF is also set to 1.
  - The fault is reported in HDBSSPROD_EL2.FSC.

I_{ZRRMT} For an HDBSS access, if a GPC fault is reported precisely, then that fault can be reported as a Data Abort, Instruction Abort, or Granule Protection Check exception, and taken to EL2 or EL3, according to the criteria described in Granule Protection Check faults.

D8.5.2.3.2 External aborts on HDBSS access

R_{JHHQW} When an HDBSS write generates an External abort, HDBSSPROD_EL2.INDEX points to the entry that generated the fault, and it is IMPLEMENTATION DEFINED which of the following reporting mechanisms is used:

- A synchronous fault is generated with all of the following properties:
  - It is reported as an Instruction Abort or Data Abort taken to EL2 or EL3, according to the value of SCR_EL3.EA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— ESR_ELx.IFSC or ESR_ELx.DFSC indicates a synchronous External abort on a translation table walk or translation table hardware update.
— ESR_EL2.ISS2.HDBSSF is set to 1, indicating the HDBSS caused a synchronous External abort.

- The fault is reported by HDBSSPROD_EL2.FSC.

R_{MWNLK} When an HDBSS write generates a synchronous External abort, the MMU is permitted to do a hardware update of the stage 2 dirty state.

R_{QNJKV} When the hardware update of the stage 2 dirty state generates a synchronous External abort, or an Unsupported atomic hardware update fault, the MMU is permitted to write to the HDBSS.

D8.5.2.4 Hardware accelerator for cleaning dirty state

I_{DRTDK} The *hardware accelerator for cleaning dirty state* (HACDBS) accelerates the process of updating Block and Page descriptors from writable-dirty to writable-clean.

R_{QLGSL} All statements in this section require implementation of the OPTIONAL feature, FEAT_HACDBS.

I_{CYLPL} The HACDBS is an HDBSS-like structure originating from either the same PE or different PEs. The format of entries in this structure is described in Table D8-93.

I_{BGSDY} HACDBS processing is enabled only if HACDBSBR_EL2.EN is 1 and HCR_EL2.VM is 1.

R_{BBWCH} When HACDBS processing is disabled by a changing HACDBSBR_EL2.EN from 1 to 0, completion of a DSB instruction guarantees that all outstanding translation table walks have completed.

I_{BYRXM} All of the following apply to the HACDBS base address:

- It is a PA configured in HACDBSBR_EL2.BADDR.
- For encodings of HACDBSBR_EL2.SZ greater than 4KB, bits [(SZ+12-1):12] of HACDBSBR_EL2.BADDR are RES0, such that the base address is aligned to its size.

R_{HGVVT} The PA space of the HACDBS base address is determined by the Effective value of SCR_EL3.{NSE, NS}.

I_{TQKLR} The HACDBS size is configured in HACDBSBR_EL2.SZ.

R_{VRMKP} Each HACDBS entry is eight bytes.

R_{SNFDZ} When processing an entry in the HACDBS, all of the following apply:

- The PE reads the HACDBS entry at the current value of HACDBSCONS_EL2.INDEX. The read is done as a 64-bit single-copy atomic operation.
- If the entry is valid, the PE decodes the HACDBS entry to determine the IPA, NSIPA and TTWL fields.
- For a valid entry, the PE uses a stage 2 translation table walk to locate the stage 2 Block or Page descriptor that translates the IPA, and NSIPA attribute if appropriate. See Translation table walk.
- If any fault is generated, then HACDBS processing stops and the fault reporting requirements apply.
- If no fault is generated, then all of the following apply:
  — If the stage 2 descriptor is writable-dirty, then that descriptor is updated to writable-clean in a coherent manner, using an atomic read-modify-write.
  — If the stage 2 descriptor is writable-clean, it is not updated.
  — The value of HACDBSCONS_EL2.INDEX is incremented by 1.

R_{KTPXV} For the purpose of processing HACDBS entries and qualifying a descriptor as writable-clean or writable-dirty in this context, it is not required that stage 2 dirty state hardware update be enabled.

If stage 2 Direct permissions are used, then the descriptor DBM bit must be 1 for the descriptor to be considered writable-dirty.

If stage 2 Indirect permissions are used, then the descriptor is writable-dirty if the Dirty bit in the descriptor is 1, and the permissions configured for the descriptor include Data Write or MMU Write permission.

R_{ZLNFF} If an entry in the HACDBS is not valid, then all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PE does not process the invalid entry.
- HACDBSCONS_EL2.INDEX is incremented by 1.

I_{XZGCW} When the PE processes an entry and updates the corresponding stage 2 descriptor to writable-clean, the PE is not required to update any copy of that descriptor cached in a TLB.

R_{BFTGZ} Accesses to the HACDBS use the same memory attributes as stage 2 translation table walks, including all of the following:

- The endianness configured by SCTLR_EL2.EE.
- The Normal memory type.
- The Cacheability and Shareability attributes configured by VTCR_EL2.{IRGN0, ORGN0, SH0}.
- If FEAT_MTE2 is implemented, accesses are always Tag Unchecked.
- If FEAT_MEC is implemented, the MECID used for stage 2 translation table walks.
- If FEAT_MPAM is implemented, the PARTID value, PMG value, and PARTID space used for stage 2 translation table walks.

I_{PQQJG} If FEAT_RME is implemented and GPC is enabled, accesses to the HACDBS are subject to Granule Protection Checks.

I_{TDFSP} For the purpose of cleaning HACDBS entries, Overlay permissions and the AssuredOnly property are ignored.

R_{JSHZF} If the requirements in this subsection and the requirements for fault reporting are met, then the PE is permitted to process entries in the HACDBS ahead of the current value of HACDBSCONS_EL2.INDEX.

R_{HWNDK} In the absence of an error, all entries preceding the entry pointed to by HACDBSCONS_EL2.INDEX have been successfully cleaned.

R_{DPCSW} In the presence of an error, all the entries preceding the entry pointed to by HACDBSCONS_EL2.INDEX have been successfully cleaned and the entry at the current index caused a fault.

I_{WWTCD} In the presence of an error, the PE is permitted to have successfully cleaned entries after the one pointed to by HACDBSCONS_EL2.INDEX.

I_{RMLQL} If the HACDBS has empty entries, software can cause the PE to terminate processing before the end of the HACDBS is reached. This is done by programming the first empty entry in the HACDBS as follows:

- The Valid field is set to 1.
- The TTWL field is set to a reserved value.
- The remaining fields are set to any value.

Processing this entry causes the PE to raise an error, at which point software can intervene and disable processing of the HACDBS.

R_{LFZLR} Hardware has finished processing the HACDBS when HACDBSCONS_EL2.INDEX is greater than or equal to $2^{(\text{HACDBSBR_EL2.SZ}+12)}/8$.

R_{DLWRN} The HACDBS interrupt request signal, `HACDBSIRQ`, is a level-sensitive interrupt request that is asserted when the Effective value of HACDBSBR_EL2.EN is 1, and one or more of the following are true:

- Hardware has finished processing the HACDBS.
- HACDBSCONS_EL2.ERR_REASON is set to a value other than `0b00`.

The interrupt remains asserted until both conditions are no longer true.

R_{RMVBG} If a *Generic Interrupt Controller* (GIC) is implemented, `HACDBSIRQ` must be a *Private Peripheral Interrupt* (PPI) in a multiprocessor system. This interrupt is signaled by the PE that implements the HACDBS.

R_{GQLDJ} When hardware has finished processing the HACDBS, all of the following occur:

- The PE stops processing entries from the HACDBS.
- The `HACDBSIRQ` is asserted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When the PE stops processing the HACDBS, it is the responsibility of EL2 software to set HACDBSBR_EL2.EN to 0.

R_{MRBXC} If HACDBSBR_EL2.EN is 1 and HACDBSCONS_EL2 does not indicate any error state, and software modifies HACDBSBR_EL2.{BADDR, SZ} or any field in HACDBSCONS_EL2, then any of the following are permitted to occur:

- The direct write is IGNORED.
- The fields in HACDBSCONS_EL2 become UNKNOWN.
- The address used for fetches of HACDBS entries is UNKNOWN.

D8.5.2.4.1 HACDBS errors

I_{YFJTM} When an error occurs while processing an HACDBS entry, HACDBSCONS_EL2.ERR_REASON indicates the reason for the error.

R_{ZJYKS} When reading an entry from the HACDBS generates any of the following, HACDBSCONS_EL2.ERR_REASON is set to `0b01`:

- An External abort.
- If FEAT_RME is implemented, then a GPC fault.

R_{LMZLD} When an entry from the HACDBS is processed and the translation table walk for the IPA generates an MMU fault at any lookup level, HACDBSCONS_EL2.ERR_REASON is set to `0b10`. This includes all MMU faults except for the following:

- If the Access flag in the Block or Page descriptor is 0, an Access flag fault is not generated and the PE can update the descriptor to writable-clean.
- A Permission fault.

R_{HDDCT} When HACDBS entry is processed and any of the following apply during the translation table walk for the IPA, HACDBSCONS_EL2.ERR_REASON is set to `0b11`:

- The write permissions provided by the descriptor are anything other than writable-clean or writable-dirty.
- The Contiguous bit in the Block or Page descriptor is 1.
- The final lookup level does not match the TTWL field in the HACDBS entry.

R_{FQSJP} When HACDBSCONS_EL2.ERR_REASON is set to a value other than `0b00`, all of the following occur:

- The PE stops processing entries from the HACDBS.
- `HACDBSIRQ` is asserted.

R_{WHMDV} When a fault is generated, HACDBSCONS_EL2.INDEX points to either:

- In the presence of a single fault, the HACDBS entry that generated the fault.
- In the presence of multiple faults, the oldest HACDBS entry that generated a fault. HACDBSCONS_EL2.INDEX is the smallest value pointing to an entry that generated a fault.

I_{LYJMC} Arm expects that after resolving an error related to the HACDBS, EL2 software increments HACDBSCONS_EL2.INDEX by 1 and sets HACDBSCONS_EL2.ERR_REASON to `0b00`.

R_{WLNPL} When an error occurs while processing an HACDBS entry, subsequent completion of a `DSB` instruction that applies to both loads and stores on the same PE guarantees that all outstanding fetches of HACDBS entries, and stage 2 translation table walks for processing HACDBS entries, have completed.

D8.5.3 Ordering of hardware updates to the translation tables

R_{MKHJS} A hardware update to the translation table that is caused by a load or a store, including an atomic instruction, is guaranteed to be observed, to the extent required by the Shareability attributes:

- Before a load or store, including an atomic instruction, to an arbitrary address, other than the address of the translation table entry, that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load or store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Before a load to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load.
- Before a store or atomic access to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry.
- Before a cache maintenance instruction to an arbitrary address appearing in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instructing that caused the update to the translation table entry and the subsequent cache maintenance instruction.

R_{PLLLM} An update to the translation table that is caused by a load is not ordered with respect to the load itself.

R_{FSZNW} An update to the translation table that is caused by a store or an atomic access is observed by all observers, to the extent required by the Shareability attributes, before the store itself in the case that the store is to the same location as the translation table update.

R_{RZYCN} An update to the translation table that is caused by a store or an atomic access is not ordered with respect to the store itself in the case that the store is not the same location as the translation table update.

D8.5.4 Restriction on memory types for hardware updates to translation tables

I_{YTWWW} A translation table can be placed in Normal memory with any Cacheability.

I_{XPRCJ} If hardware update of translation table entries is enabled, software is required to ensure that the translation table is located in memory that supports atomic read-modify-write updates in a coherent manner.

I_{MYHXH} Atomicity properties can only be met at the system level, and some system implementations might not support this functionality in all memory regions, including all of the following:

- Any system memory type that does not support hardware cache coherency.
- Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does not support hardware cache coherency.

I_{PCHYY} The system implementation determines which memory type is treated as Non-cacheable.

R_{WLTNL} All of the following Conventional memory types architecturally guarantee that hardware updates of the translation tables are atomic:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

R_{NNNMB} If a translation table hardware update is not atomic as observed by other agents that can access memory, then the update can have one or more of the following effects:

- A synchronous External abort on the translation table walk is generated.
- The instruction generates an SError exception.
- An MMU fault is generated, reported as an Unsupported atomic hardware update using one of the following Fault status codes:
  - For Data Abort exceptions, ESR_ELx.DFSC = `0b110001`.
  - For Instruction Abort exceptions, ESR_ELx.IFSC = `0b110001`.
  - For an abort on a write to the Statistical Profiling buffer, PMBSR_EL1.FSC = `0b110001`.
- The hardware updates occur and all of the following apply:
  - There is no guarantee that the memory accesses are done atomically in regard to accesses to memory by other agents.
  - The instruction might also generate an SError exception.

R_{VSXXT} If a translation table hardware update is not atomic as observed by other agents that can access memory, and the update generates an MMU fault, all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL2 is implemented and enabled in the current Security state, then for the EL1&0 translation regime, one of the following:
  - If the atomic hardware update is not supported in the stage 1 translation due to the defined memory type, or if the stage 2 translation is not enabled, then this exception is a stage 1 abort and is taken to EL1.
  - In all other cases, the exception is a stage 2 abort and is taken to EL2.
- For a translation stage, the MMU fault priority is at an IMPLEMENTATION DEFINED point between all of the following:
  - Immediately before the priority of an Access flag fault that is generated at the same translation stage.
  - Immediately after the priority of a Permission fault that is generated at the same translation stage.

R_{YQMMR} If translation table hardware updates are not atomic as observed by other agents that can access memory, then in all of the following cases, the architecture permits, but does not require, an address translation instruction to generate an MMU fault, reported as an Unsupported atomic hardware update using PAR_EL1.FST = `0b110001`:

- For a stage 1 translation at EL1, stage 1 translation table hardware updates are enabled.
- For EL2 or EL3, translation table hardware updates are enabled.

I_{FGVJW} For more information, see Possible implementation restrictions on using atomic instructions.

D8.5.5 Use of the Contiguous bit with hardware updates to the translation tables

R_{XDPXQ} When hardware updates any of the AF, AP[2] or S2AP[1] either of the AF or AP[2] bits, then the update applies to a single translation table entry.

R_{LCGVS} If hardware updates a translation table entry, and if the Contiguous bit in that entry is 1, then the members in a group of contiguous translation table entries can have different AF, AP[2], and S2AP[1] values.

R_{GPHKF} If all of the following apply, then the architecture does not require a hardware update of the AF:

- An access is to a location translated by an entry that has the Contiguous bit set to 1.
- At least one descriptor in the set of contiguous translation table entries has AF set to 1.

R_{JKCXG} If all of the following apply, then the architecture does not require a hardware update of the AP[2] or S2AP[1] bit:

- A write is done to a location translated by a descriptor that has the Contiguous bit set to 1.
- The corresponding AP[2] or S2AP[1] bit in at least one descriptor in the set of contiguous translation table entries indicates that the entry is dirty.

I_{QDCMP} The Contiguous bit permits, but does not require, hardware to hold a single entry in a TLB for the set of translation table entries in the group, and to have updated only one or more of the AF and the AP[2] bit or S2AP[1] bit in the single translation table entry that resulted in the TLB entry.

I_{NFFSV} Software is required to combine the AF values and AP[2] or S2AP[1] values across all translation table entries in a contiguous group to determine whether any of the entries have been accessed or are dirty.

I_{QKKDC} For more information on the Contiguous bit, see The Contiguous bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.6 Memory region attributes

I_{SGDKK} The memory region attributes in a descriptor define the memory type, Memory Tagging attributes, the XS attribute, Cacheability, and whether the memory region is Shareable. For translation regimes that support two translation stages, the stage 1 and stage 2 memory region attributes are combined to produce a resultant memory type and Cacheability, and Shareability.

I_{HJZQP} All of the following apply to the memory region attributes:

- The stage 1 OA attributes are derived from a combination of the Block descriptor and Page descriptor AttrIndx[3:0] field, and the MAIR_ELx and MAIR2_ELx register fields.
- The stage 1 translation table walk attributes are derived from the TCR_ELx.{IRGN*n*, ORGN*n*, SH*n*} fields.
- The SCTLR_ELx.{C, I} fields might override stage 1 attributes.
- The HCR_EL2.PTW field might override stage 2 translation attributes of a stage 1 translation table walk.
- If EL2 is enabled, then for the EL1&0 translation regime the HCR_EL2.{CD, DC} fields might override stage 1 and stage 2 attributes.
- All of the following HCR_EL2.FWB values affect the stage 2 attributes:
  - When the Effective value of HCR_EL2.FWB is 0, stage 2 attributes are derived from the stage 2 MemAttr[3:2] descriptor bits, then combined with stage 1 attributes.
  - When the Effective value of HCR_EL2.FWB is 1, stage 2 attributes are derived from the stage 2 MemAttr[2:0] descriptor bits.
- For Device memory and Normal Non-cacheable mappings from stage 1, normalization of the Shareability that is input to stage 2 is IMPLEMENTATION DEFINED.
- For Device memory and Normal Non-cacheable mappings from stage 2, Shareability is always normalized to Outer Shareable.

I_{GJJZB} For a virtualization implementation, combining memory attributes from stage 1 and stage 2 translations support all of the following functions that are useful to a hypervisor executing at EL2:

- Reduce or increase the permitted Cacheability of a region.
- Increase the required Shareability of a region.

I_{LHJDF} The effects of combining memory attributes defined by the hypervisor are expected to be functionally transparent to the Guest OS.

D8.6.1 Stage 1 memory type and Cacheability attributes

R_{NWZCF} If FEAT_AIE is implemented, then all of the following apply:

- If a stage 1 translation regime uses VMSAv9-128, then the Effective value of the stage 1 *Attribute Index Extension* (AIE) is 1.
- If a stage 1 translation regime uses VMSAv8-64, then the AIE value is determined by the following table:

Table D8-94 Determination of AIE value

Translation regime	AIE value
EL1&0	TCR2_EL1.AIE
EL2&0	TCR2_EL2.AIE
EL2	TCR2_EL2.AIE
EL3	TCR_EL3.AIE

R_{SXRTL} If FEAT_AIE is not implemented, then the Effective value of AIE is 0.

R_{HHGNL} If FEAT_AIE is not implemented, all of the following apply:

- For a stage 1 translation, the Block descriptor and Page descriptor AttrIndx[2:0] field holds the value *n* used to select the 8-bit field MAIR_ELx.Attr<*n*> that specifies the memory region attributes of the descriptor OA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- It is IMPLEMENTATION DEFINED how AttrIndx[2:0] is used to index into AMAIR_ELx.

R_{YJLRL} If FEAT_AIE is implemented, then all of the following apply to the Block descriptor and Page descriptor AttrIndx field:

- If AIE is disabled, or if AIE is enabled and AttrIndx[3] is 0, then AttrIndx[2:0] holds the value *n* used to select the 8-bit field in all of the following registers:
  - MAIR_ELx.Attr<*n*> that specifies the memory region attributes of the descriptor OA.
  - AMAIR_ELx.Attr<*n*> that specifies the IMPLEMENTATION DEFINED memory region attributes of the descriptor OA.
- If AIE is enabled and AttrIndx[3] is 1, then AttrIndx[2:0] holds the value *n* used to select the 8-bit field in all of the following registers:
  - MAIR2_ELx.Attr<*n*> that specifies the memory region attributes of the descriptor OA.
  - AMAIR2_ELx.Attr<*n*> that specifies the IMPLEMENTATION DEFINED memory region attributes of the descriptor OA.

R_{BXFVK} If AIE is enabled, then AttrIndx[3:0] is used to index into AMAIR_ELx as follows:

- When AttrIndx[3] is 0, stage 1 Auxiliary Memory Attributes are obtained from AMAIR_ELx[AttrIndx[2:0]*8+7:AttrIndx[2:0]*8].
- When AttrIndx[3] is 1, stage 1 Auxiliary Memory Attributes are obtained from additional AMAIR2_ELx[AttrIndx[2:0]*8+7:AttrIndx[2:0]*8].

I_{DKDPN} FEAT_AIE enables use of page based IMPLEMENTATION DEFINED memory attributes to select an 8-bit field from AMAIR_ELx or AMAIR2_ELx.

I_{FVLFZ} For the memory region specified by a Block descriptor and Page descriptor, the MAIR_ELx.Attr<*n*> or MAIR2_ELx.Attr<*n*> field selected by the descriptor AttrIndx[2:0] field defines all of the following:

- The Device or Normal memory type.
- For Device memory, all of the following:
  - One of the Device-nGnRnE, Device-nGnRE, Device-nGRE, or Device-GRE Device memory types.
  - If FEAT_XS is implemented, then the XS attribute.
- For Normal memory, all of the following:
  - For both Inner and Outer Cacheability, one of the Non-cacheable, Write-Through, or Write-Back attributes.
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate hints, each of which is Allocate or No Allocate, and the Transient allocation hints.
  - If FEAT_MTE2 is implemented, then the Tagged attribute.
  - If FEAT_XS is implemented, then the XS attribute.

For more information, see XS attribute modifier and Memory region tagging types.

R_{QVNKY} If FEAT_AIE is implemented, then the value `0x00` in AMAIR_ELx.Attr<*n*> or AMAIR2_ELx.Attr<*n*> does not give IMPLEMENTATION DEFINED memory region attributes for translations using that Attr<*n*> field at stage 1.

I_{NSVYS} The memory attributes obtained from MAIR_ELx.Attr<*n*>, MAIR2_ELx.Attr<*n*>, AMAIR_ELx.Attr<*n*>, and AMAIR2_ELx.Attr<*n*> are permitted to be cached in a TLB.

I_{QNNGV} For some translation regimes, the memory region attributes determined from translation tables, MAIR_ELx, and MAIR2_ELx values might be overridden by the configuration of SCTLR_ELx.{C, I} and HCR_EL2.DC.

R_{MJGNV} For stage 1 of a translation regime using VMSAv8-64, if AIE is enabled, then hierarchical permissions are disabled and the corresponding TCR_ELx.HPD*n* are RES1.

D8.6.2 Stage 1 Shareability attributes

R_{ZWYSH} For Normal Cacheable memory, if the Effective value of TCR_ELx.DS is 0, then the SH[1:0] field in a stage 1 translation Block descriptor and Page descriptor encodes the Shareability attributes of the descriptor OA, as shown in the following table:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-95 Stage 1 Shareability attributes

SH[1:0]	Normal memory
00	Non-shareable
01	Reserved, CONSTRAINED UNPREDICTABLE as described in Reserved values in System and memory-mapped registers and translation table entries.
10	Outer Shareable
11	Inner Shareable

R_{PYFVQ} If a region is mapped as Device memory or Normal Non-cacheable memory after all enabled translation stages, then the region has an effective Shareability attribute of Outer Shareable.

R_{NSCNC} If stage 2 translation is enabled and stage 1 maps a region as Device memory or Normal Non-cacheable memory, then it is IMPLEMENTATION DEFINED whether:

- The Shareability attribute configured for stage 1 is input into stage 2 translation.
- An effective Outer Shareable attribute is input into stage 2 translation.

I_{HRNGF} If the Effective value of TCR_ELx.DS is 1, then the stage 1 translation descriptor bits[9:8] are OA[51:50] instead of SH[1:0].

I_{GJPJW} For Normal Cacheable memory, if the Effective value of TCR_ELx.DS is 1, then the stage 1 translation Shareability is treated as being mapped by one of the following:

- For the EL3 translation regime, TCR_EL3.SH0.
- For the EL2 translation regime, if the Effective value of HCR_EL2.E2H is 0, then TCR_EL2.SH0.
- For the EL2 and EL2&0 translation regimes, if the Effective value of HCR_EL2.E2H is 1 and the VA is an address that is translated using tables pointed to by TTBR0_EL2, then TCR_EL2.SH0.
- For the EL2 and EL2&0 translation regimes, if the Effective value of HCR_EL2.E2H is 1 and the VA is an address that is translated using tables pointed to by TTBR1_EL2, then TCR_EL2.SH1.
- For the EL1&0 translation regime, if the VA is an address that is translated using tables pointed to by TTBR0_EL1, then TCR_EL1.SH0.
- For the EL1&0 translation regime, if the VA is an address that is translated using tables pointed to by TTBR1_EL1, then TCR_EL1.SH1.

D8.6.3 Stage 2 memory type and Cacheability attributes

R_{WYQVN} For any stage 2 translation, if the final memory type any is Normal Cacheable type, then all of the following apply:

- If the output of stage 1 specifies a Cacheable memory type, then the final cache allocation hints are the stage 1 cache allocation hints.
- If the output of stage 1 does not specify a Cacheable memory type, then the final cache allocation hints are Read Allocate, Write Allocate.

R_{KTSYP} Stage 2 translation configuration does not assign cache allocation hints.

I_{BBJDH} If the memory type from a stage 2 translation causes a stage 1 translation table walk to a Device memory type, then one of the following occurs:

- If HCR_EL2.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory, and the walk can be done speculatively.
- If HCR_EL2.PTW is 1, then the memory access generates a stage 2 Permission fault.

D8.6.4 Stage 2 Memory Tagging attributes

R_{KVBTQ} All statements in this section and subsections require implementation of FEAT_MTE_PERM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{HKMRJ} If the stage 1 Tagged attribute is not defined for a memory region then the NoTagAccess attribute is not defined.

I_{FWGDY} The NoTagAccess attribute is a permission and has no effect on the cacheability of Allocation Tags.

R_{XSYYN} If all of the following are true, then a stage 2 Permission fault due to the NoTagAccess attribute is generated on a memory access:

- The access is an explicit Allocation Tag read, explicit Allocation Tag write or a Tag Checked access.
- The memory region tagging type for the access is Tagged. See Memory region tagging types.
- The combined stage 1 and stage 2 memory attributes specify the NoTagAccess attribute.

I_{FYRCG} A Permission fault due to the NoTagAccess attribute is not generated for a Tag Unchecked memory access.

R_{YMSKV} On a stage 2 Permission fault due to the NoTagAccess attribute, all of the following are true:

- The fault is taken to EL2.
- The fault is reported as a Data Abort from a lower Exception level.
- ESR_EL2.TagAccess is 1.
- The value of ESR_EL2.WnR is set to one of the following:
  - — If the fault is due to a Tag write effect, then 1.
  - — If the fault is due to a Tag read effect, then 0.
- FAR_EL2 is valid.

R_{PYQZB} For the purpose of determining whether a Permission fault due to the NoTagAccess attribute is generated for a Tag Checked access, if a Tag Check Fault is configured to have no effect on the PE due to SCTLR_ELx.{TCF, TCF0}, then the access is treated as Tag Unchecked.

I_{XRKQK} For more information about the prioritization of a Permission fault due to the NoTagAccess attribute, see Prioritization of Permission faults.

D8.6.5 Stage 2 memory type and Cacheability attributes when FWB is disabled

R_{MFDHF} All statements in this section and subsections require that the Effective value of HCR_EL2.FWB is 0.

R_{QPHXK} If the Effective value of HCR_EL2.FWB is 0, then the stage 1 and stage 2 memory type and Cacheability attributes are combined.

R_{HMNDG} For stage 2 translations, the MemAttr[3:0] field in a Block descriptor and Page descriptor encodes the memory type and Cacheability of the memory region addressed by the descriptor OA, as shown in the following table:

Table D8-96 Stage 2 MemAttr[3:0] encoding

MemAttr[3:0]	Memory type	Cacheability	Condition
0000	Device-nGnRnE	Not applicable	-
0001	Device-nGnRE	Not applicable	-
0010	Device-nGRE	Not applicable	-
0011	Device-GRE	Not applicable	-
0100	Normal NoTagAccess	Outer Write-Back Cacheable Inner Write-Back Cacheable	FEAT_MTE_PERM is implemented
	Reserved	Reserved, CONSTRAINED UNPREDICTABLE	FEAT_MTE_PERM is not implemented
0101	Normal	Outer Non-cacheable Inner Non-cacheable	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MemAttr[3:0]	Memory type	Cacheability	Condition
0110	Normal	Outer Non-cacheable Inner Write-Through Cacheable	-
0111	Normal	Outer Non-cacheable Inner Write-Back Cacheable	-
1000	Reserved	Reserved, CONSTRAINED UNPREDICTABLE	-
1001	Normal	Outer Write-Through Cacheable Inner Non-cacheable	-
1010	Normal	Outer Write-Through Cacheable Inner Write-Through Cacheable	-
1011	Normal	Outer Write-Through Cacheable Inner Write-Back Cacheable	-
1100	Normal	Reserved, CONSTRAINED UNPREDICTABLE	-
1101	Normal	Outer Write-Back Cacheable Inner Non-cacheable	-
1110	Normal	Outer Write-Back Cacheable Inner Write-Through Cacheable	-
1111	Normal	Outer Write-Back Cacheable Inner Write-Back Cacheable	-

I_{LJCRP} For more information, see:

- Reserved values in System and memory-mapped registers and translation table entries.
- Combining stage 1 and stage 2 memory type attributes.

D8.6.5.1 Combining stage 1 and stage 2 memory type attributes

R_{TNHFM} If the Effective value of HCR_EL2.FWB is 0, then the stage 1 and stage 2 memory type attributes are combined as shown in the following table:

Table D8-97 Combining stage 1 and stage 2 memory type attributes

Rule	If either translation stage assigns:	Resultant memory type is:
Device has precedence over Normal	Any Device memory type	Any Device memory type
Non-Gathering has precedence over Gathering	A Device-nGxx memory type	A Device-nGxx memory type
Non-Reordering has precedence over Reordering	A Device-nGnRx memory type	A Device-nGnRx memory type
No Early write acknowledge has precedence over Early write acknowledge	The Device-nGnRnE memory type	The Device-nGnRnE memory type

D8.6.5.2 Combining stage 1 and stage 2 Cacheability attributes for Normal memory

R_{GQFSF} If the Effective value of HCR_EL2.FWB is 0, then the stage 1 and stage 2 Inner Cacheability and Outer Cacheability attributes are combined as shown in the following table:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-98 Combining stage 1 and stage 2 Cacheability attributes for Normal memory

Assignment in stage 1	Assignment in stage 2	Resultant Cacheability
Non-cacheable	Any	Non-cacheable
Any	Non-cacheable	Non-cacheable
Write-Through Cacheable	Write-Through Cacheable	Write-Through Cacheable
Write-Through Cacheable	Write-Back Cacheable	Write-Through Cacheable
Write-Back Cacheable	Write-Through Cacheable	Write-Through Cacheable
Write-Back Cacheable	Write-Back Cacheable	Write-Back Cacheable

I_{MCQKW} For stage 2 translations, if the Effective value of HCR_EL2.FWB is 0 and the stage 2 MemAttr[3:0] field is `0b1111`, then the combined memory type and Cacheability attributes are the output memory type and Cacheability attributes from stage 1.

R_{JSXRX} If FEAT_MTE_PERM is implemented, the stage 2 NoTagAccess attribute is determined by the following:

- If the stage 1 memory type is not Tagged, the stage 2 NoTagAccess attribute is ignored.
- If the stage 1 memory type is Tagged, the stage 1 and stage 2 attributes are combined as shown in Table D8-99:

Table D8-99 Combined stage 1 and stage 2 attributes if FEAT_MTE_PERM is implemented

Stage 1 memory type and Cacheability attribute	Stage 2 memory type and Cacheability attribute	Resultant memory type and Cacheability attribute
Normal Write-Back, Tagged	Normal Write-Back	Normal Write-Back, Tagged
Normal Write-Back, Tagged	Normal Write-back, NoTagAccess	Normal Write-back, Tagged, NoTagAccess

D8.6.6 Stage 2 memory type and Cacheability attributes when FWB is enabled

R_{LNQJQ} All statements in this section and subsections require that the Effective value of HCR_EL2.FWB is 1.

R_{ZNGNK} For stage 2 translations, if FEAT_MTE_PERM is not implemented, then FEAT_S2FWB has all of the following effects on the MemAttr[3:2] bits:

- MemAttr[3] is RES0.
- The value of MemAttr[2] determines the interpretation of the MemAttr[1:0] bits.

R_{NQDNZ} For stage 2 translations, if FEAT_MTE_PERM is implemented, then MemAttr[3] is not RES0 and all bits of MemAttr[3:0] determine the memory region type and Cacheability attributes.

R_{JXGKQ} For stage 2 translations, if FEAT_MTE_PERM is implemented, then all of the following values of MemAttr[3:2] apply:

- `0b10` is Reserved.
- All other values determine the interpretation of the MemAttr[1:0] bits.

R_{VRJSW} For stage 2 translations, if MemAttr[2] is 0, or if FEAT_MTE_PERM is implemented and MemAttr[3:2] is `0b00`, then the MemAttr[1:0] bits define Device memory attributes as shown in the following table:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-100 Stage 2 MemAttr[1:0] encoding when MemAttr[2] is 0, FWB enabled

Stage 2 MemAttr[1:0]	Device Memory Attribute
00	Device-nGnRnE
01	Device-nGnRE
10	Device-nGRE
11	Device-GRE

R_{RHWZM} For stage 2 translations, the MemAttr[1:0] bits affect the memory type and Cacheability as shown in the following table:

Table D8-101 Stage 2 MemAttr[1:0] encoding, FWB enabled

Stage 2 MemAttr[3:2]	Stage 1 memory type and Cacheability attribute	Stage 2 MemAttr[1:0]	Resultant memory type and Cacheability attribute
01	Device<attr>	01	Device<attr>
01	Normal Non-cacheable	01	Normal Non-cacheable
01	Normal Write-Through	01	Normal Non-cacheable
01	Normal Write-Back	01	Normal Non-cacheable
01	Normal Write-Back, Tagged	01	Normal Non-cacheable
01	Normal Write-Back, Tagged	1x	Normal Write-Back, Tagged
11	-	01	Reserved
11	Normal Write-Back, Tagged	1x	Normal Write-Back, Tagged, NoTagAccess
x1	-	00	Reserved
x1	Device<attr>	10	Normal Write-Back
x1	Normal Non-cacheable	10	Normal Write-Back
x1	Normal Write-Through	10	Normal Write-Back
x1	Device<attr>	11	Device<attr>
x1	Normal Non-cacheable	11	Normal Non-cacheable
x1	Normal Write-Through	11	Normal Write-Through
x1	Normal Write-Back	1x	Normal Write-Back

R_{GSDKZ} If MemAttr[1:0] bits define Device memory attributes, then stage 2 Device memory attributes are combined with stage 1 memory attributes.

R_{THYXB} If the stage 1 translation specifies a cacheable memory type, then the stage 1 cache allocation hint is applied to the final cache allocation hint where the final memory type is cacheable.

R_{TBQKG} If the stage 1 translation does not specify a cacheable memory type, then if the final memory type is cacheable, it is treated as Read-Allocate, Write-Allocate, Non-Transient.

R_{RMTWN} If the stage 1 translation specifies a Device memory type, and the stage 2 descriptor MemAttr[2:0] field is `0b110`, then all of the following apply:

- If an atomic memory access occurs, then it is IMPLEMENTATION DEFINED whether it is supported in the same way as accesses to memory locations with a resultant Device memory type.
- If an exclusive access occurs, then it is IMPLEMENTATION DEFINED whether it is supported in the same way as accesses to memory locations with a resultant Device memory type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If a misaligned access occurs, then it is CONSTRAINED UNPREDICTABLE whether the resultant stage 1 memory type generates a stage 1 Alignment fault.
- For a `DC ZVA`, `DC GZVA`, or `DC GVA` instruction, it is CONSTRAINED UNPREDICTABLE whether the resultant stage 1 memory type generates a stage 1 Alignment fault.
- If the translation applies to any Active element of an SVE Non-fault vector load instruction, or to an Active element that is not the First active element of an SVE First-fault vector load instruction, then it is CONSTRAINED UNPREDICTABLE whether the element access is performed, or is suppressed and reported in the FFR.

I_{RRDBK} The architecture requires that CLIDR_EL1.{LoUU, LoIUS} are {0, 0} so that no data cache levels need to be cleaned to manage coherency with instruction fetches.

R_{DPDXS} If the stage 1 translation specifies a Normal memory type with Cacheability other than Write-Back Cacheable, and the stage 2 descriptor MemAttr[2:0] field is `0b110`, then it is CONSTRAINED UNPREDICTABLE whether accesses generated by the following instruction types follow the alignment requirements in the same way as accesses to a Normal memory type with Cacheability other than Write-Back Cacheable:

- Load-Exclusive, Store-Exclusive and Atomic instructions.
- Non-atomic Load-Acquire/Store-Release instructions.

I_{LJRWR} For more information, see:

- XS attribute modifier.
- Reserved values in System and memory-mapped registers and translation table entries.
- Alignment of data accesses.

D8.6.7 Stage 2 Shareability attributes

I_{CGGBV} The stage 2 Shareability attributes are not affected by the Effective value of HCR_EL2.FWB.

R_{DNZJQ} For Normal Cacheable memory, if the Effective value of VTCR_EL2.DS is 0, then the SH[1:0] field in a stage 2 translation Block descriptor and Page descriptor encodes the Shareability attributes of the descriptor OA, as shown in the following table:

Table D8-102 Stage 2 Shareability attributes

SH[1:0]	Normal memory
00	Non-shareable
01	Reserved, CONSTRAINED UNPREDICTABLE
10	Outer Shareable
11	Inner Shareable

R_{HTSGQ} For Normal Cacheable memory, if the Effective value of VTCR_EL2.DS is 1, then the stage 2 translation Shareability is determined by VTCR_EL2.SH0.

R_{YHCTP} For Device memory and Normal Non-cacheable memory, the Shareability attributes of the stage 2 translation descriptor OA are treated as Outer Shareable.

I_{LPRLS} If the Effective value of VTCR_EL2.DS is 1, then the stage 2 translation descriptor bits[9:8] are OA[51:50] instead of SH[1:0].

I_{DPSTX} For more information, see Reserved values in System and memory-mapped registers and translation table entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.6.7.1 Combining the stage 1 and stage 2 Shareability attributes for Normal memory

I_{LTFXF} The value of HCR_EL2.FWB does not affect how Shareability attributes from stage 1 and stage 2 are combined.

R_{WLQYW} If the resultant memory type from a stage 1 and stage 2 translation is one of the following, the memory region is treated as Outer Shareable:

- Any Device memory type.
- Normal Inner Non-cacheable, Outer Non-cacheable.

R_{ZNLRJ} For a memory region with a resultant memory type of Normal that is not Inner Non-cacheable, Outer Non-cacheable, the stage 1 and stage 2 Shareability attributes can be combined as shown in the following table:

Table D8-103 Combining stage 1 and stage 2 Shareability attributes

Assignment in stage 1	Assignment in stage 2	Resultant Shareability
Outer Shareable	Any	Outer Shareable
Inner Shareable	Outer Shareable	Outer Shareable
Inner Shareable	Inner Shareable	Inner Shareable
Inner Shareable	Non-shareable	Inner Shareable
Non-shareable	Outer Shareable	Outer Shareable
Non-shareable	Inner Shareable	Inner Shareable
Non-shareable	Non-shareable	Non-shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.7 Other descriptor fields

D8.7.1 The Contiguous bit

I_{LYNXC} The Contiguous bit identifies a descriptor as belonging to a group of adjacent translation table entries that point to a contiguous OA range.

R_{SKQHL} If the Effective value of the Contiguous bit in a Block descriptor or Page descriptor is 1, and the descriptor would otherwise be permitted to be cached in a TLB, then all of the following apply:

- The entry is permitted to be cached in a TLB as though it is one of a number of adjacent translation table entries that point to a contiguous OA range with consistent attributes and permissions.
- Software is required to ensure that all of the adjacent translation table entries for the contiguous region point to a contiguous OA range with consistent attributes and permissions.

R_{HMQXG} For VMSAv8-64 translation system, the Contiguous bit is RES0 and has an Effective value of 0 in all of the following Block descriptors:

- For the 4KB translation granule, if the Effective value of TCR_ELx.DS or VTCR_EL2.DS is 1, then the Contiguous bit is RES0 in the level 0 Block descriptor of that translation regime.
- For the 16KB translation granule, if the Effective value of TCR_ELx.DS or VTCR_EL2.DS is 1, then the Contiguous bit is RES0 in the level 1 Block descriptor of that translation regime.
- For the 64KB translation granules, if FEAT_LPA is implemented, then the Contiguous bit is RES0 in a level 1 Block descriptor.

R_{CBXXM} For a lookup level in a translation granule using the VMSAv8-64 translation system, if the Effective value of the Contiguous bit is 1, then the entry is permitted to be cached in a TLB as though all of the properties shown in the following table apply:

Table D8-104 Permitted properties of caching a translation in a TLB when Effective value of Contiguous bit is 1, VMSAv8-64 translation system

Translation granule	Block or Page lookup level	Number of adjacent translation table entries	Alignment boundary of adjacent translation table entries within the translation table	Alignment of contiguous OA range
4KB	1	16	128 bytes	16GB
	2	16	128 bytes	32MB
	3	16	128 bytes	64KB
16KB	2	32	256 bytes	1GB
	3	128	1KB	2MB
64KB	2	32	256 bytes	16GB
	3	32	256 bytes	2MB

R_{RLJBB} For the VMSAv9-128 translation system, the Contiguous bit is RES0 and has an Effective value of 0 in all of the following Block descriptors:

- For the 4KB translation granule, the level 0 Block descriptor of that translation regime.
- For the 64KB translation granule, the level 1 Block descriptor of that translation regime.

R_{CMKCR} For a lookup level in a translation granule using the VMSAv9-128 translation system, if the Effective value of the Contiguous bit is 1, then the entry is permitted to be cached in a TLB as though all of the properties shown in the following table apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-105 Permitted properties of caching a translation in a TLB when Effective value of Contiguous bit is 1, VMSAv9-128 translation system

Translation granule	Block or Page lookup level	Number of adjacent translation table entries	Alignment boundary of adjacent translation table entries within the translation table	Alignment of contiguous OA range
4KB	1	4	64 bytes	1GB
	2	16	256 bytes	16MB
	3	16	256 bytes	64KB
16KB	1	4	64 bytes	64GB
	2	16	256 bytes	256MB
	3	64	1KB	1MB
64KB	2	64	1KB	16GB
	3	16	256 bytes	1MB

R_{JDLPG} For the initial lookup table in a translation regime that uses the VMSAv9-128 translation system, the following table shows how the *Disable Contiguous bit*, DisCH*n*, is determined:

Table D8-106 Determination of DisCHn for the initial lookup table

Translation Regime	VA Range (VA[55])	DisCH
EL1&0	Lower (0)	TCR2_EL1.DisCH0
	Upper (1)	TCR2_EL1.DisCH1
EL2&0	Lower (0)	TCR2_EL2.DisCH0
	Upper (1)	TCR2_EL2.DisCH1
EL3	-	TCR_EL3.DisCH0

R_{WSVXT} For a translation regime that uses the VMSAv9-128 translation system, if the corresponding DisCH*n* field is 1, then the Effective value of the Contiguous bit in the initial lookup table Block and Page descriptors is 0.

R_{BBRTL} For the VMSAv9-128 translation system, if the DisCH field in the next level table descriptor is 1, then the Effective value of the Contiguous bit in the stage 1 next level Block and Page descriptors is 0.

I_{VNXYF} The architecture does not require descriptors with the Contiguous bit set to 1 to be cached as a single TLB entry for the contiguous region. To avoid TLB coherency issues, software is required to perform TLB maintenance on the entire address region that results from using the Contiguous bit.

I_{CQTNL} If the Effective value of the Contiguous bit is 1, then hardware updates to the AF and dirty state can cause members in a group of contiguous translation table entries to have different AF, AP[2], and S2AP[1] values.

For more information, see Use of the Contiguous bit with hardware updates to the translation tables.

D8.7.1.1 Misprogramming the Contiguous bit

R_{QTRPM} For all descriptors within the range indicated by one or more descriptors that have the Contiguous bit set to 1, if one or more of the contiguous translation table entries does not have the Contiguous bit set to 1, then this is misprogramming of the Contiguous bit, and a TLB might contain overlapping entries.

R_{NGLXZ} For a TLB lookup in a contiguous region mapped by translation table entries that have the Contiguous bit misprogrammed, that TLB lookup is permitted to produce one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An OA, access permissions, and memory attributes that are consistent with any of the programmed translation table values.
- If BBM support levels 1 and 2 are not implemented, then an OA, access permissions, or memory attributes that are inconsistent with any of the programmed translation table values. For more information, see Support levels for changing table or block size.
- A TLB conflict abort.

R_{JQQTC} For a TLB lookup in a contiguous region mapped by translation table entries that have consistent values for the Contiguous bit, but have the OA, attributes, or permissions misprogrammed, that TLB lookup is permitted to produce an OA, access permissions, and memory attributes that are consistent with any one of the programmed translation table values.

I_{PGVGZ} The Contiguous bit is present only in valid Block and Page translation table descriptors, and therefore neither of the following configurations are considered as misprogramming of the Contiguous bit:

- A contiguous range of descriptors that are each either invalid, or valid with Contiguous set to 1.
- A contiguous range of descriptors that are each either invalid, or valid with Contiguous set to 0.

D8.7.1.2 Architectural guarantees when the Contiguous bit is misprogrammed

R_{MWFWF} If the Contiguous bit is misprogrammed, then the architecture guarantees all of the following:

- If physical memory regions are inaccessible through translation tables programmed at EL1 when the Contiguous bit is not misprogrammed, then they remain inaccessible to software executing at EL1 or EL0.
- If memory attributes and permissions are unattainable through translation tables programmed at EL1 when the Contiguous bit is not misprogrammed, then they remain unattainable to software executing at EL1 or EL0.
- Software executing in Non-secure state cannot access Secure, Realm, or Root physical memory.
- Software executing in Secure state cannot access Realm or Root physical memory.
- Software executing in Realm state cannot access Secure or Root physical memory.

R_{XFDVL} The PE is required to generate an Address Size fault on accesses to addresses above the configured OA size, even in the case where the Contiguous bit is used to mark a set of Block descriptors as contiguous, such that the OA region translated by that contiguous set of descriptors exceeds the configured OA size.

I_{JHQPP} An implementation is permitted to ignore the Contiguous bit and treat it as 0 to ensure that OA range checking is done.

D8.7.1.3 Implementation options when the Contiguous bit is misprogrammed

I_{HQWXX} If software does all of the following, it is considered a programming error:

- The Contiguous bit is used to mark a set of translation table entries as contiguous.
- The address range translated by the set of translation table entries is larger than the IA size configured for the translation stage as determined by the TCR_ELx.T*n*SZ field.

R_{WMNPV} If the Contiguous bit is used to mark a set of translation table entries as contiguous and if the address range translated by the translation table entries is larger than the IA size supported by the translation stage, as defined by the TCR_ELx.T*n*SZ field, then an implementation is permitted, but not required to, do one or more of the following:

- Generate a Translation fault.
- When all of the following apply, an implementation is permitted to not generate a Translation fault:
  - A translation table entry within a contiguous set of translation table entries is accessed.
  - The translation table entry is valid.

D8.7.2 Page Based Hardware attributes

I_{RHDNL} The PBHA bits can be used for IMPLEMENTATION DEFINED purposes.

R_{XCJVR} All statements in this section require implementation of FEAT_HPDS2.

R_{JVMMH} For a stage 1 translation that uses TTBR0_ELx, if all of the following apply, then hardware can use the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] for IMPLEMENTATION DEFINED purposes:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The corresponding TCR_ELx.HWU*nn* bit, HWU62, HWU61, HWU60, or HWU59, is 1.
- TCR_ELx.HPD0 is 1.

R_{QSTMZ} For a stage 1 translation that uses TTBR1_ELx, if all of the following apply, then hardware can use the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] for IMPLEMENTATION DEFINED purposes:

- The corresponding TCR_ELx.HWU1*nn* bit, HWU162, HWU161, HWU160, or HWU159, is 1.
- TCR_ELx.HPD1 is 1.

R_{BHQKX} For a stage 2 translation, if the value of VTCR_EL2.HWU*nn* is 1, then hardware can use the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] for IMPLEMENTATION DEFINED purposes.

R_{VDFPM} If the PBHA bit is used for IMPLEMENTATION DEFINED purposes, then setting the PBHA bit to 0 has the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.

R_{PFZRK} If the Effective value of TCR_ELx.HWU{0, 1}*nn* is 0, then the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] is IGNORED.

R_{TDLFM} For any translation stage, if Overlay permissions are enabled, or for stage 1 translations, the Attribute Index Extension is enabled, then all of the following apply:

- The Effective value of PBHA[3:0] is `0b0000`.
- For stage 1 translations, the Effective value of the corresponding TCR_ELx.HWU*nn* bits are 0.
- For stage 2 translations, the Effective value of the corresponding VTCR_EL2.HWU*nn* bits are 0.

For more information, see Stage 1 Overlay permissions, Stage 2 Overlay permissions, and Stage 1 memory type and Cacheability attributes.

I_{TKJCZ} The VMSAv9-128 translation system does not support the PBHA fields, and the TCR_ELx.HWU{0, 1}*nn* fields are RES0.

D8.7.3 Table and Block entry

R_{DFJNG} All statements in this section require implementation of FEAT_BBML1.

I_{PJZBK} The nT bit is supported in the following descriptors:

- VMSAv8-64 Block descriptors.
- VMSAv9-128 descriptors, when the SKL field in that descriptor is not 0.

I_{XPRKH} Setting the nT bit in a Table descriptor or Block descriptor guarantees that, when changing the table or block size, accesses translated by the translation table entry do not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

R_{MRRPW} When using a Table descriptor or Block descriptor with the nT bit set to 1, all of the following apply:

- It is IMPLEMENTATION DEFINED whether a Translation fault is generated.
- If a Translation fault is not generated, a TLB conflict abort can be generated.

For more information, see Support levels for changing table or block size.

I_{DXRJK} If the nT bit in a Table descriptor or Block descriptor is 1, then the translation performance can be significantly impacted.

D8.7.4 XS attribute modifier

I_{PMZTD} The XS attribute indicates that an access to the memory region could take a long time to complete. Variants of `DSB` instructions and TLB maintenance instructions that do not depend on the completion of memory accesses with the XS attribute set to 1 are defined.

R_{DNHCL} All statements in this section require implementation of FEAT_XS.

R_{FQWJG} For a stage 2 translation, the FnXS bit in a Block descriptor and Page descriptor has all of the following properties:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the FnXS bit is 0, then the XS attribute of the resultant memory translation is not modified.
- If the FnXS bit is 1, then the XS attribute of the resultant memory translation is set to 0.

R_{TDMCS} For stage 2 translations, if the resultant memory attributes are Normal Inner Write-back, Outer Write-back Cacheable, then the XS attribute is set to 0 on the resultant memory translation.

I_{ZXKSJ} If FEAT_XS is implemented and the stage 1 memory type defined by the MAIR_ELx or TCR_ELx registers is one of the following, then the XS attribute is set to 0, otherwise the XS attribute is set to 1:

- Device memory types that use the MAIR_ELx.Attr<*n*> encoding `0b0000dd01`.
- For Normal memory, one of the following:
  - Inner Write-Back Cacheable, Outer Write-back Cacheable memory types defined in the MAIR_ELx or TCR_ELx registers, including any memory types that are treated as Write-Back Cacheable as a result of IMPLEMENTATION DEFINED choices in the architecture.
  - Inner Write-through Cacheable and Outer Write-through Cacheable memory types that use the MAIR_ELx.Attr<*n*> encoding `0b10100000`.
  - Inner Non-cacheable, Outer Non-cacheable memory types that use the MAIR_ELx.Attr<*n*> encoding `0b01000000`.

I_{SQFSD} The stage 2 impact of the FnXS bit applies for stage 1 translations in the EL1&0 translation regime from AArch32 or AArch64.

I_{MTRQS} For more information, see:

- Enabling and disabling the caching of memory accesses.
- Stage 1 memory type and Cacheability attributes.
- Stage 2 memory type and Cacheability attributes when FWB is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.8 Address tagging

IXWNGB When address tagging is enabled, the top eight bits of the VA are ignored for the purposes of address translation and they are instead described as an address tag. An address tag might be used for software purposes, a Logical Address Tag, or both.

RLVTFP If address tagging is enabled in a translation regime, then the top eight bits of the VA, bits[63:56], have all of the following properties:

- Bits[63:56] are the address tag.
- The bits are ignored during address translation.
- If the translation system is enabled, then the bits are ignored when determining whether the address is out of range and therefore generates a Translation fault.
- If the translation system is not enabled, then the bits are ignored when determining whether the address is out of range and therefore generates an Address size fault.

IPLXWZ If Logical Address Tagging is enabled, bits[59:56] are also the Logical Address Tag. For more information, see Logical Address Tagging.

D8.8.1 Address tag control

ICKLSG Address tagging is controlled using the *Top Byte Ignore* (TBI) field in the TCR_ELx register.

RBLBYZ For a stage 1 translation that supports a single VA range, TCR_ELx.TBI{0} determines whether address tagging is enabled.

RLPCSW For a stage 1 translation that supports two VA ranges, all of the following determine whether address tagging is enabled:

- TCR_ELx.TBI0 determines whether the lower address range uses address tags.
- TCR_ELx.TBI1 determines whether the upper address range uses address tags.

RKYBQL The TCR_ELx.TBI{*n*} bit controls address tagging whether or not the corresponding stage 1 translation is enabled.

IPRTHR If FEAT_PAuth is implemented and a TCR_ELx.TBI{*n*} bit enables address tagging, then the corresponding TCR_ELx.TBID{*n*} bit determines whether address tagging applies to both instruction addresses and data addresses, or just data addresses. For more information, see Pointer authentication.

D8.8.2 Effect of address tagging on the PC

IWKJBM This section describes how the top byte of an address is considered when address tagging is enabled and an address is propagated to the PC for a branch, procedure return, exception, or exception return occurs.

RBBQKQ For a stage 1 translation that supports a single VA range, if address tagging for instruction accesses is enabled in an Exception level, ELx, then bits[63:56] of the address loaded into the PC are forced to `0x00` in all of the following cases:

- A branch or procedure return within ELx.
- An exception is taken to ELx.
- When an exception return does not generate an Illegal exception return, any exception return to ELx.
- Exiting from Debug state to ELx.

RFWHLV For a stage 1 translation that supports two VA ranges, if address tagging for instruction accesses is enabled in an Exception level, ELx, then bits[63:56] of the address loaded into the PC are a sign-extension of address bit[55] in all of the following cases:

- A branch or procedure return within ELx.
- An exception is taken to ELx.
- When an exception return does not generate an Illegal exception return, any exception return to ELx.
- Exiting from Debug state to ELx.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{CCGCS} If the value of the SPSR_ELx.M[4] bit of the saved process state is 1 when an Illegal exception return occurs, indicating a return to AArch32 state, then PC bits[63:32] are UNKNOWN.

R_{WGTRC} When address tagging is enabled and an address causes a Data Abort or a *Watchpoint exception*, the address tag is included in the VA returned in the FAR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.9 Logical Address Tagging

R_{MWZSJ} All statements in this section require implementation of FEAT_MTE2.

R_{RYFYK} For the purposes of address translation, when Logical Address Tagging is enabled in a translation regime, bits[59:56] of the VA are ignored, and instead form a Logical Address Tag.

R_{HYJDX} If all of the following are true, then the VA, bits[63:60], can be used to hold a PAC:

- Logical Address Tagging is enabled.
- Address tagging is disabled.
- FEAT_PAuth is implemented.

D8.9.1 Logical Address Tag control

R_{TQHWL} For a stage 1 translation that supports a single VA range, all of the following determine whether Logical Address Tagging is enabled:

- TCR_ELx.TBI*n*.
- If FEAT_MTE_NO_ADDRESS_TAGS is implemented, then TCR_ELx.MTX*n*.

R_{SHNLM} For a stage 1 translation that supports two VA ranges, all of the following determine whether Logical Address Tagging is enabled:

- For the lower address range:
  - TCR_ELx.TBI0.
  - If FEAT_MTE_NO_ADDRESS_TAGS is implemented, then TCR_ELx.MTX0.
- For the upper address range:
  - TCR_ELx.TBI1.
  - If FEAT_MTE_NO_ADDRESS_TAGS is implemented, then TCR_ELx.MTX1.

R_{PXNMX} The TCR_ELx.TBI{n} bit, and if FEAT_MTE_NO_ADDRESS_TAGS the TCR_ELx.MTX{n} bit, controls Logical Address Tagging whether or not the corresponding stage 1 translation is enabled.

I_{QVQJP} When Address tagging is enabled for an address range, whether Logical Address Tagging is enabled has no effect on a PE other than for use in:

- Tag Checking.
- Access to Allocation Tags in memory.

R_{NPYMC} If Logical Address Tagging is enabled and Address tagging is disabled, then bits[59:56] of the VA have all of the following properties:

- For a stage 1 translation that supports a single VA range, the bits are treated as `0b0000` in all of the following cases:
  - During address translation.
  - If the translation system is enabled, then when determining whether the address is out of range and therefore generates a Translation fault.
  - If the translation system is not enabled, then when determining whether the address is out of range and therefore generates an Address size fault.
- For a stage 1 translation that supports two VA ranges, the bits are treated as a sign-extension of VA bit[55] in all of the following cases:
  - During address translation.
  - If the translation system is enabled, then when determining whether the address is out of range and therefore generates a Translation fault.
  - If the translation system is not enabled, then when determining whether the address is out of range and therefore generates an Address size fault.

I_{BBHPM} For the properties of VA bits[59:56] when Address tagging is enabled, see Address tagging.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.9.1.1 Effect of Logical Address Tagging on address translation and cache maintenance instructions

R_{PKRBM} For the purpose of determining whether bits [59:56] hold a Logical Address Tag, all address translation instructions and cache maintenance instructions that perform address translation are treated as memory accesses.

D8.9.1.2 Effect of Logical Address Tagging on the PC

R_{WCTSR} When an address is used as an instruction address and not for a data access, Logical Address Tagging is treated as disabled when loading an address into the PC in all of the following cases:

- A branch or procedure return within an Exception level.
- Taking an exception to an Exception level.
- Exception return to an Exception level.
- Exit from Debug state to an Exception level.

R_{HTSLR} If FEAT_PAuth is implemented, Logical Address Tagging is treated as disabled on execution of any of the following instructions that operate on a PAC field within an instruction address:

- `AUTI*`.
- `PACI*`.
- `XPACI`.
- `XPACLRI`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.10 Pointer authentication

I_{XDNDT} AArch64 state supports signing the contents of a 64-bit general-purpose register that contains an address with a *pointer authentication code* (PAC), and authenticating the contents of a register with a PAC before the register is used as the target of an indirect branch, or the base register of a load or store instruction.

R_{STHJF} All statements in this section and subsections require implementation of FEAT_PAuth.

R_{JZTGK} Pointer authentication is supported only in AArch64 state.

I_{PKPFN} If Pointer authentication is implemented, then both of the following are true:

- Exactly one of ID_AA64ISAR1_EL1.GPI, ID_AA64ISAR1_EL1.GPA, or ID_AA64ISAR2_EL1.GPA3 fields is nonzero.
- Exactly one of ID_AA64ISAR1_EL1.API, ID_AA64ISAR1_EL1.APA, or ID_AA64ISAR2_EL1.APA3 fields is nonzero.

Otherwise, all of these ID fields are zero.

I_{GMTCR} Pointer authentication functionality is the same whether address translation is enabled or disabled, but is useful only when address translation is enabled.

D8.10.1 PAC field

I_{KQZPC} The pointer authentication mechanism treats the upper bits of a pointer in a 64-bit general-purpose register as a PAC field.

I_{BSXBR} The TCR_ELx.{T0SZ, T1SZ} fields are used to configure the size and location of the PAC field.

R_{FWYCF} If FEAT_CONSTPACFIELD is not implemented, when a 64-bit general-purpose register, Xd, holds an address with a PAC field, the location of the PAC field is determined by all of the following:

- The `bottom_PAC_bit` is 64-TCR_ELx.T*n*SZ, and the value of *n* is determined as follows:
  - If inserting a PAC into an address and address tagging is not used, then the value of Xd[63] is the value of *n*.
  - Otherwise, the value of Xd[55] is the value of *n*.
- If address tagging is used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[55:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[54:`bottom_PAC_bit`].
- If address tagging is not used and logical address tagging is used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[63:60, 55:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[63:60, 54:`bottom_PAC_bit`].
- If both address tagging and logical address tagging are not used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[63:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[63:56, 54:`bottom_PAC_bit`].

For more information, see Address tagging and Logical Address Tagging.

Note

A TCR_ELx.T*n*SZ value less than 12 requires implementation of FEAT_LVA3.
If FEAT_LVA3 is implemented, and address tagging or logical address tagging are enabled, pointer authentication might provide software with little to no benefit due to the reduced PAC field size.

R_{XTGWV} If FEAT_CONSTPACFIELD is implemented, when a 64-bit general-purpose register, Xd, holds an address, the location of the PAC field is determined by all of the following:

- The `bottom_PAC_bit` is 64-TCR_ELx.T*n*SZ.
- The value of Xd[55] is the value of *n* used in T*n*SZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If address tagging is used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[55:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[54:`bottom_PAC_bit`].
- If address tagging is not used and logical address tagging is used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[63:60, 55:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[63:60, 54:`bottom_PAC_bit`].
- If both address tagging and logical address tagging are not used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[63:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[63:56, 54:`bottom_PAC_bit`].

R_{NQZWG} If FEAT_CONSTPACFIELD is implemented, then an implementation is permitted to use the value in Xd[55] to determine the size of the PAC field when adding a PAC to Xd, even when address tagging is not used.

I_{VZRCW} In the case where address tagging is not used, then any use of a VA that has differing values of bit [63] and bit [55] is a non-canonical VA that will generate a fault. For this reason, the presence or absence of FEAT_CONSTPACFIELD does not affect the use of pointer authentication when inserting a PAC into a canonical VA.

I_{HLQVM} If the TCR_ELx.TBID{*n*} bit is used to restrict address tagging to data addresses, then instruction addresses can use Xd[63:56] bits to produce larger pointer authentication code fields.

I_{DJNGK} For all of the following reasons, if FEAT_PAuth2 is implemented, FEAT_FPAC is not implemented, and stage 1 translation is disabled, then Arm recommends not setting the TCR_ELx.T*n*SZ values to indicate an address range that is smaller than the PA size:

- For some PAC values, if TCR_ELx.T*n*SZ is set to indicate an address range that is smaller than the PA size, the address generated by a failed PAC authentication might be an address within the PA size because the upper address bits, those between the PA size and the size indicated by the TCR_ELx.T*n*SZ field, are taken from the result of the authentication process.
- Accessing memory using such an address that has failed PAC authentication does not generate an Address size fault.
- The memory access is done using upper address bits, those between the PA size and the size indicated by the TCR_ELx.T*n*SZ field, taken from the result of the authentication process.

R_{VMZGK} If the value in TCR_ELx.T*n*SZ is outside the permitted range, then one of the following CONSTRAINED UNPREDICTABLE results occur:

- The TCR_ELx.T*n*SZ value configures the `bottom_PAC_bit` value.
- The minimum permitted TCR_ELx.T*n*SZ value configures the `bottom_PAC_bit` value.
- The maximum permitted TCR_ELx.T*n*SZ value configures the `bottom_PAC_bit` value.

R_{FCMCH} If a PE treats an out of range TCR_ELx.T*n*SZ value as the maximum permitted field value or the minimum permitted field value for all purposes except reading the field value, then that behavior also applies to determining `bottom_PAC_bit`.

D8.10.2 PAC generation and verification keys

I_{RSXKN} For pointer authentication, all of the following 128-bit PAC keys are provided:

- Instruction address PAC key A, APIAKey_EL1, is the concatenation of the register values APIAKeyHi_EL1:APIAKeyLo_EL1.
- Instruction address PAC key B, APIBKey_EL1, is the concatenation of the register values APIBKeyHi_EL1:APIBKeyLo_EL1.
- Data address PAC key A, APDAKey_EL1, is the concatenation of the register values APDAKeyHi_EL1:APDAKeyLo_EL1.
- Data address PAC key B, APDBKey_EL1, is the concatenation of the register values APDBKeyHi_EL1:APDBKeyLo_EL1.
- Generic authentication PAC key A, APGAKey_EL1, is the concatenation of the register values APGAKeyHi_EL1:APGAKeyLo_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{KBYGG} For an Exception level, all of the following bits in the SCTLR_ELx registers are used to enable PAC generation and validation:

- SCTLR_ELx.EnIA enables instruction address pointer authentication using the `APIAKey_EL1` key.
- SCTLR_ELx.EnIB enables instruction address pointer authentication using the `APIBKey_EL1` key.
- SCTLR_ELx.EnDA enables data address pointer authentication using the `APDAKey_EL1` key.
- SCTLR_ELx.EnDB enables data address pointer authentication using the `APDBKey_EL1` key.

I_{GMQNP} All Exception levels use the same set of registers to hold PAC keys.

I_{GLRSR} When switching between Exception levels, software is expected to apply all of the following:

- Software is expected to switch the PAC keys between Exception levels.
- Software is expected to not leave the values of the current keys present in memory, typically done by zeroing those locations after switching.

D8.10.3 PAC instructions

I_{VXGGK} When a 64-bit general-purpose register is used as the target of an indirect branch instruction or the base register in a load instruction, all of the following instructions can be used to authenticate the register contents and prepare the register to be used:

- Instructions that insert a PAC into the PAC field.
- Instructions that extract the value in the PAC field and check that the value is correct.
- Instructions that remove the PAC field in the register and replace it with the pointer sign-extension bits, without verification.

I_{KCKSC} An instruction that inserts a PAC into the PAC field of a 64-bit general-purpose register generates the PAC using the value of the 64-register and one or two other 64-bit diversifier values.

I_{WCZPC} FEAT_PAuth_LR provides instructions that allow signing of the 64-bit value in LR against the value of SP and a PC-relative instruction address.

R_{TMSKZ} An instruction that inserts a PAC into the PAC field of a 64-bit general-purpose register, operates on the PAC field in one of the following ways:

- If FEAT_PAuth2 is not implemented, then the pointer authentication mechanism replaces the PAC field with the PAC.
- If FEAT_PAuth2 is implemented, then the pointer authentication mechanism exclusive-ORs the bits in the PAC field with the PAC.

R_{TFBFT} When FEAT_PAuth2 is not implemented and when performing a PAC operation on a non-canonical address then the PAC is modified to make it incorrect, and one of the following applies:

- If FEAT_EPAC is implemented, then the pointer authentication mechanism sets the PAC field to 0.
- If FEAT_EPAC is not implemented, then the pointer authentication mechanism inverts one of the bits of the PAC as follows:
  - When address tagging is not used, bit [62] is inverted.
  - Otherwise, bit [54] is inverted.

I_{KBKGF} An instruction that extracts the PAC from the PAC field and checks that the value is correct does all of the following:

- The check is based on the value of the register and one or two other 64-bit diversifier values.
- If the value is correct, the PAC is replaced with the extension bits.
- If the value is incorrect, all of the following occur:
  - The PAC is replaced with the extension bits.
  - Two extension bits are set to a unique, fixed value, such that the 64-bit value represents a non-canonical VA. This is referred to as making the VA non-canonical.

I_{RZFZG} Multiple versions of pointer authentication instructions support different use cases. Some instructions combine a pointer authentication operation with another operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{WNKCJ} If PAC generation and validation is disabled, then the PACGA and `XPAC*` instructions are not affected and are always enabled.

R_{WDYDC} If PAC generation and validation is disabled, then all of the following apply:

- Instructions that insert a PAC into the address in a 64-bit general-purpose register execute as `NOPs`.
- Instructions that authenticate a PAC execute as `NOPs`.
- If an instruction combines pointer authentication with another operation, then all of the following apply:
  - The pointer authentication does not occur.
  - The combined operation behaves the same as the equivalent instruction that does not do pointer authentication.

I_{RHMHV} All of the following are examples of the resulting behavior of instructions that combine pointer authentication with another operation, if PAC generation and validation is disabled:

- A `RETAA` instruction operates as a RET instruction.
- A `LDRAA Xt, [Xn, #<simm10>]!` instruction operates as a `LDR Xt, [Xn, #<simm10>]!` instruction.

I_{ZVGYK} The PACGA instruction generates a 32-bit PAC from two 64-bit values and a generic key.

I_{ZPGRZ} The PACGA instruction can be used to protect small memory blocks. PACGA instructions can be chained to protect an arbitrary-sized memory block.

I_{PXWQC} For pointer authentication instructions other than the PACGA instruction, the PAC is generated using one of the following IMPLEMENTATION DEFINED methods:

- If FEAT_PACQARMA5 is implemented, then ID_AA64ISAR1_EL1.APA is nonzero and the QARMA5 block cipher algorithm is used.
- If FEAT_PACIMP is implemented, then ID_AA64ISAR1_EL1.API is nonzero and an IMPLEMENTATION DEFINED algorithm is used.
- If FEAT_PACQARMA3 is implemented, then ID_AA64ISAR2_EL1.APA3 is nonzero and the QARMA3 block cipher algorithm is used.

I_{WXMCQ} For the PACGA instruction, the PAC is generated using one of the following IMPLEMENTATION DEFINED methods:

- If FEAT_PACQARMA5 is implemented, then ID_AA64ISAR1_EL1.GPA is nonzero and the QARMA5 block cipher algorithm is used.
- If FEAT_PACIMP is implemented, then ID_AA64ISAR1_EL1.GPI is nonzero and an IMPLEMENTATION DEFINED algorithm is used.
- If FEAT_PACQARMA3 is implemented, then ID_AA64ISAR2_EL1.GPA3 is nonzero and the QARMA3 block cipher algorithm is used.

R_{ZYPJV} For a PAC generated using the `ComputePAC()` pseudocode function, where the PAC generation uses an IMPLEMENTATION DEFINED algorithm then the IMPLEMENTATION DEFINED algorithm uses the same arguments as the `ComputePAC()` function.
For a PAC generated using the `ComputePAC2()` pseudocode function, where the PAC generation uses an IMPLEMENTATION DEFINED algorithm then the IMPLEMENTATION DEFINED algorithm uses the same arguments as the `ComputePAC2()` function.

R_{JTRCS} For a set of arguments passed to the `ComputePAC()` or `ComputePAC2()` pseudocode function, the same result is produced by all PEs that an execution thread could migrate between.

R_{DPXKM} The `ComputePAC2()` pseudocode function generates a PAC for 64-bit data, using a 128-bit PAC key and two diversifiers.

R_{MLLDB} If `ComputePAC2()` is implemented using FEAT_PACQARMA5 or FEAT_PACQARMA3, then the two diversifiers are concatenated into a single 64-bit diversifier as follows:

- Bits[31:0] of the 64-bit diversifier hold the value modifier1<35:4>.
- Bits[63:32] of the 64-bit diversifier hold the value modifier2<36:5>.

I_{RNXPW} Arm strongly recommends that software at EL0 does not mix pointer authentication mechanisms. For example, using the PACGA instruction to create a PAC and using other pointer authentication instructions to authenticate, or using an instruction which uses a specific key to sign a pointer and using an instruction which uses a different key to authenticate that pointer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.10.4 PSTATE.PACM

R_{BYRKF} All statements in this section require implementation of FEAT_PAuth_LR.

I_{TQTGT} PSTATE and the PACM instruction provide a mechanism to produce a program that can run on a PE that implements FEAT_PAuth_LR or on a PE that does not implement FEAT_PAuth_LR, by extending the behavior of existing instructions for signing and authenticating a return address.

I_{RZKKL} `PACM` is intended to be executed immediately before a `PACIASP`, `PACIBSP`, `AUTIASP`, `AUTIBSP`, `RETAA`, `RETAB`, `AUTIA1716`, `AUTIB1716`, `PACIA1716`, or `PACIB1716` instruction, to extend the operation of those instructions to take a second diversifier.

R_{PLWLR} Unless otherwise specified, PSTATE.PACM is set to 0 after execution of any instruction..

R_{GJJYJ} If an IMPLEMENTATION DEFINED algorithm is implemented for Pointer authentication instructions other than the `PACGA` instruction, PSTATE.PACM is permitted to be a trivial implementation.

Otherwise, PSTATE.PACM is fully implemented.

I_{LYDML} For a trivial implementation of PSTATE.PACM, all of the following apply:

- The `PACM` instruction does not set PSTATE.PACM to 1.
- The effects of `PACM` instructions are disabled at all Exception levels and PSTATE.PACM behaves as 0.
- SPSR_ELx.PACM, DSPSR_EL0.PACM, HCRX_EL2.PACMEn, and SCTLR2_ELx.{EnPACM0, EnPACM} are fully implemented, although a value of 1 in these fields has no effect other than for reading back the value of the register.

Note

If FEAT_PACIMP is implemented, then use of PSTATE.PACM to modify the behavior of a subsequent instruction might not result in the same PAC field insertion or authentication as the explicit instructions. For example, use of PSTATE.PACM to modify a subsequent `PACIASP` instruction might not result in the same PAC field insertion as a `PACIASPPC` instruction at the same PC as the `PACIASP`. This means that software should not mix the methods used to sign and authenticate a pointer, and should use a single method for a specific pointer.

R_{TQKBS} For a full implementation of PSTATE.PACM, if the effects of `PACM` instructions are enabled, then PSTATE.PACM is set to 1 after execution of a `PACM` instruction.

R_{WNZMT} If PSTATE.PACM is 1, then the behavior of the `PACIASP`, `PACIBSP`, `AUTIASP`, `AUTIBSP`, `RETAA`, `RETAB`, `AUTIA1716`, `AUTIB1716`, `PACIA1716`, and `PACIB1716` instructions are extended as follows:

- The pseudocode function `ComputePAC2()` is used for the generation of the PAC. See R_{ZYPJV}.
- For the `PACIASP` and `PACIBSP` instructions, the 64-bit value of `modifier2` used in the generation of the PAC is the value specified in the PC.
- For the `AUTIASP`, `AUTIBSP`, `RETAA`, and `RETAB` instructions, the 64-bit value of `modifier2` is the value in X16.
- For the `AUTIA1716`, `AUTIB1716`, `PACIA1716`, and `PACIB1716` instructions, the 64-bit value of `modifier2` is the value in X15.

I_{SRKXB} Each of the SCTLR2_ELx.{EnPACM, EnPACM0} control bits enables or disables the effect of the `PACM` instruction at an Exception level.

I_{HBWFK} The HCRX_EL2.PACMEn bit can control the effect of a `PACM` instruction at EL1 and EL0.

I_{RVHSQ} If the effects of `PACM` instructions are disabled at an Exception level, PSTATE.PACM cannot be set to 1 at that Exception level and is consistent with being treated as 0 at that Exception level.

I_{CDWGZ} When the `PACM` instruction is used, to ensure code is fully backward compatible with a PE that does not implement FEAT_PAuth_LR, a `BTI` instruction must be used in the function prologue. For more information, see PSTATE.BTYPE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.10.5 Faulting on pointer authentication

I_{XBGSY} A PAC authentication failure for a given VA can cause a fault to be generated in the following three manners, according to the type of the instruction and whether FEAT_FPAC and FEAT_FPACCOMBINE are implemented:

- The PAC instruction makes the VA non-canonical, such that a subsequent use of the VA generates a fault. In this case, the PAC instruction does not directly generate the fault.
- The PAC instruction makes the VA non-canonical and uses that VA such that a fault is generated by that instruction.
- The PAC instruction directly generates a PAC Fail exception, with EC code `0b011100`.

R_{MLWGL} If an instruction is a combined instruction that includes pointer authentication, then when the PAC is incorrect in a given VA, one of the following behaviors occurs:

- For a combined authenticate and load instruction, then:
  - — If FEAT_FPACCOMBINE is not implemented, the VA is made non-canonical and then used as the address for the load.
  - — If FEAT_FPACCOMBINE is implemented, then the instruction generates a PAC Fail exception, with EC code `0b011100`.
- For a combined authenticate and branch instruction, then:
  - — If FEAT_FPACCOMBINE is not implemented, the VA is made non-canonical and the PC is updated to this non-canonical value.
  - — If FEAT_FPACCOMBINE is implemented, then the instruction generates a PAC Fail exception, with EC code `0b011100`.

R_{KJSRQ} For a PAC authentication instruction, `AUT*`, when the PAC is incorrect for a given VA, one of the following behaviors occurs:

- If FEAT_FPAC is not implemented, the VA is made non-canonical.
- If FEAT_FPAC is implemented, then the instruction generates a PAC Fail exception, with EC code `0b011100`.

R_{XXFXH} Predicting the result of a PAC Authentication is a form of control flow prediction.

R_{GKVYD} Prediction of the results of a PAC Authentication must adhere to the requirements of FEAT_FPACC_SPEC, if implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.11 Checked Pointer Arithmetic

R_{QRKMW} All statements in this section require implementation of FEAT_CPA.

I_{LYNBW} It is possible that bits[63:56] of the VA might be modified in an attempt to circumvent features using these bits, including but not limited to:

- Memory Tagging Extensions (MTE). See The Memory Tagging Extension.
- Pointer Authentication Codes (PAC). See Pointer authentication.
- Software only schemes using address tag bits. See Address tagging and Logical Address Tagging.

I_{KCTBC} Checked Pointer Arithmetic has all of the following properties:

- It is independent of the value of bits[63:56] of the VA.
- It is independent of the use case of bits[63:56] of the VA.
- It is independent of whether address tagging is enabled by TCR_ELx.TBI.
- If Checked Pointer Arithmetic for Multiplication is enabled, all of the following:
  - A recorded attempted modification of bits[63:56] of the VA is preserved by a pointer arithmetic instruction.
  - A recorded multiply overflow is preserved by a pointer arithmetic instruction.

R_{THDQL} If an instruction has all of the following, it is a Checked Pointer Arithmetic instruction:

- An Explicit Memory effect.
- An explicit base address register operand.
- One or more explicit transfer register operands containing data elements that are loaded from or stored to corresponding addressed memory locations.

I_{YVGDD} All of the following are not Checked Pointer Arithmetic instructions:

- `LDR (literal)`, because it implicitly uses the PC and does not have an explicit base address register operand.
- System register accesses by `MRS`, `MRRS`, `MSR`, and `MSSR` instructions that are transformed into memory accesses by the Effective value of HCR_EL2.NV2, because they implicitly use VNCR_EL2 and do not have an explicit base address register operand.
- System instructions such as `DC` and `IC`, because they do not have an explicit transfer register operand.
- Memory Copy and Memory Set instructions introduced by FEAT_MOPS, because they do not have a transfer register operand that is loaded from or stored to a corresponding addressed memory location.

R_{LHJPP} FEAT_CPA introduces all of the following Checked Pointer Arithmetic instructions:

- The scalar instructions `ADDPT`, `SUBPT`, `MADDPT`, and `MSUBPT`.
- The SVE instructions `ADDPT`, `SUBPT`, `MADPT`, and `MLAPT`.

I_{LWVFP} If FEAT_CPA2 is implemented, the Checked Pointer Arithmetic instructions are controlled by the following register fields:

- SCTLR2_EL1.{CPTM0, CPTM, CPTA0, CPTA}.
- SCTLR2_EL2.{CPTM0, CPTM, CPTA0, CPTA}.
- SCTLR2_EL3.{CPTM, CPTA}.

Note

If an unprivileged Checked Pointer Arithmetic instruction, such as `LDTR` or `STTR`, behaves as if the instruction was executed at EL0, then it is subject to the effects of SCTLR2_ELx.CPTA0. Other tag preserving pointer instructions, such as `ADDPT` and `MADDPT`, are subject to the effects of SCTLR2_ELx.CPTA.

R_{XCDRZ} For Checked Pointer Arithmetic instructions, if Checked Pointer Arithmetic for Addition is enabled, then all of the following apply:

- Bits[63:56] from the base register are preserved in the result.
- If bits[55:54] of the base register are `0b10` or `0b01` then bits[55:54] of the base register are preserved in the result.
- If bits[55:54] of the base register are `0b11` or `0b00` and the operation would have modified bits[63:56] from the base register when calculating the result, then bit[55] of the result is set to bit[55] of the base register and bit[54] of the result is set to `NOT`(bit[55]) of the base register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{GTPHB} For a pointer arithmetic instruction, a multiply overflow is defined as the expression

```
product<127:0> != SignExtend(product<63:0>, 128);
```

where

```
integer product = SInt(multiplicand1) * SInt(multiplicand2);
```

where each multiplicand is a 64 bit number.

R_{SJZHH} For Checked Pointer Arithmetic instructions, if Checked Pointer Arithmetic for Multiplication is enabled, then the following applies:

- If bits[55:54] of the base register are `0b11` or `0b00` and a multiply overflow is detected, then bit[55] of the result is set to bit[55] of the base register and bit[54] of the result is set to `NOT`(bit[55]) of the base register.

R_{WQDKJ} Predicting the results of comparing base[63:56] with result[63:56] and of comparing base[55] with base[54] is a form of control flow prediction.

Note

Categorizing the prediction of comparing base[63:56] with result[63:56] and of comparing base[55] with base[54] as a form of control flow prediction means that the speculation is not restricted by CSDB. Treating this as control flow prediction is consistent with predicting MTE tag check failures, which are also classified as control flow prediction.

R_{TSFPY} When a Checked Pointer Arithmetic instruction accesses a sequential set of bytes that crosses the `0xXXFF_FFFF_FFFF_FFFF` boundary, it is CONSTRAINED UNPREDICTABLE whether Checked Pointer Arithmetic is applied to the VA accessed for the bytes above this boundary.

R_{QVJHR} If Checked Pointer Arithmetic is not enabled, then Checked Pointer Arithmetic instructions add the offset value to the base register value without checking bits[63:56] and without any special handling of bits[55:54].

I_{JKJWP} An operation that corrupts bits[63:56] results in a pointer with bit[55] != bit[54], which implies the following:

- Consumption of the pointer by a PAC insertion instruction inserts an invalid PAC as FEAT_PAuth requires the pointer to be canonical.
- If the MMU is enabled, then consumption of the pointer by address translation likely generates a Translation fault.
- If the MMU is disabled, then consumption of the pointer by address translation likely generates a Address Size fault.

I_{QDPCR} Use of Checked Pointer Arithmetic requires a VA size less than or equal to 54 bits. This corresponds to a TCR_ELx.TnSZ field value of 10 or greater.

If FEAT_CPA2 and FEAT_LVA3 are implemented, Checked Pointer Arithmetic is enabled, and the VA size is greater than 54 bits, then Checked Pointer Arithmetic will corrupt legitimate pointers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.12 Memory Encryption Contexts extension

I_{YTBSD} The Memory Encryption Contexts extension (MEC extension) introduces memory encryption contexts (MECs) to all physical address (PA) spaces. Multiple memory encryption contexts are provided to the Realm physical address space for assignment to Realm virtual machines, with policy controlled by Realm EL2. The Root, Secure and Non-Secure physical address spaces each have one context.

R_{LFYSJ} All statements in this section and subsections require implementation of FEAT_MEC.

D8.12.1 Effect of MEC on PA spaces

R_{GKRVJ} PAs in the Root PA space are associated with the default MECID of zero.

R_{YJVST} PAs in the Secure PA space are associated with the default MECID of zero.

R_{MRKMQ} PAs in the Non-secure PA space are associated with the default MECID of zero.

R_{WGBXM} Root, Secure and Non-Secure PA spaces do not support multiple encryption contexts.

D8.12.1.1 Effect on the EL3 translation regime

I_{HHCPS} If FEAT_MEC is implemented, then FEAT_RME is also implemented and therefore execution in EL3 uses the Root Security state.

R_{FMLTL} For EL3 stage 1 translated addresses, if SCTLR2_EL3.EMEC is 0, then accesses to the Realm PA space use the default MECID of zero.

R_{CCSND} For EL3 stage 1 translated addresses, if SCTLR2_EL3.EMEC is 1, then accesses to the Realm PA space use the Realm PA space Alternate MECID for EL3, configured in MECID_RL_A_EL3.

R_{LPNPC} MECID_RL_A_EL3 is not permitted to be cached in a TLB.

I_{TZYKP} EL3 software can only access the Realm PA space using translated addresses. The {NSE, NS} fields within the Block or Page descriptors must equal {1, 1} to access the Realm PA space.

D8.12.1.2 Effect on the Realm EL2 and Realm EL2&0 translation regimes

R_{CBJVF} If SCTLR2_EL2.EMEC is 0, then Realm EL2 and EL2&0 accesses to Realm PA space use the default MECID of zero.

R_{FCSBF} If SCTLR_EL2.M is 0 and SCTLR2_EL2.EMEC is 1, then Realm EL2 and EL2&0 accesses to Realm PA space use the EL2&0 Primary 0 MECID, configured in MECID_P0_EL2.

R_{RXMFG} For Realm EL2 stage 1 translations, if SCTLR_EL2.M is 1, SCTLR2_EL2.EMEC is 1, and the Effective value of HCR_EL2.E2H is 0, then all lookup levels use the EL2&0 Primary 0 MECID, configured in MECID_P0_EL2.

R_{XBDTH} For Realm EL2&0 stage 1 TTBR0 and TTBR1 translations, if SCTLR_EL2.M is 1, SCTLR2_EL2.EMEC is 1, and the Effective value of HCR_EL2.E2H is 1, then all lookup levels use the MECID determined by the value of the TCR_EL2.A1 field.

TCR_EL2.A1	Translation lookup MECID
0	MECID_P1_EL2
1	MECID_P0_EL2

I_{TSRTP} TCR_EL2.A1 is not permitted to be cached in a TLB, and MECID values are not permitted to be cached in a TLB. However, the choice of which MECID register is used for translation table walks is permitted to be cached in a TLB, as long as it is only cached in TLB entries that are associated with the ASID that is also selected by the A1 field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{LZGSD} For Realm EL2 and EL2&0 stage 1 TTBR0 translated addresses to Realm PA space, if SCTLR_EL2.M is 1, SCTLR2_EL2.EMEC is 1, and TCR2_EL2.AMEC0 is 0, then the translation validity and MECID selection is determined by the AMEC field in the Block or Page descriptor used for that translation.

TTD.AMEC	Translation validity	Output MECID
0	Valid	MECID_P0_EL2
1	Translation fault	N/A

R_{VWKVQ} For Realm EL2&0 stage 1 TTBR1 translated addresses to Realm PA space, if SCTLR_EL2.M is 1, SCTLR2_EL2.EMEC is 1, TCR2_EL2.AMEC1 is 0, and the Effective value of HCR_EL2.E2H is 1, then the translation validity and MECID selection is determined by the AMEC field in the Block or Page descriptor used for that translation.

TTD.AMEC	Translation validity	Output MECID
0	Valid	MECID_P1_EL2
1	Translation fault	N/A

I_{PDRGM} Changing the state of either TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 from 1 to 0 requires TLB maintenance operations for the update to be visible.

R_{THGCP} For Realm EL2 and EL2&0 stage 1 TTBR0 translated addresses to Realm PA space, if SCTLR_EL2.M is 1, SCTLR2_EL2.EMEC is 1, and TCR2_EL2.AMEC0 is 1, then MECID selection is determined by the AMEC field in the Block or Page descriptor used for that translation.

TTD.AMEC	Output MECID
0	MECID_P0_EL2
1	MECID_A0_EL2

R_{MQHXQ} For Realm EL2&0 stage 1 TTBR1 translated addresses to Realm PA space, if SCTLR_EL2.M is 1, SCTLR2_EL2.EMEC is 1, TCR2_EL2.AMEC1 is 1, and the Effective value of HCR_EL2.E2H is 1, then MECID selection is determined by the AMEC field in the Block or Page descriptor used for that translation.

TTD.AMEC	Output MECID
0	MECID_P1_EL2
1	MECID_A1_EL2

R_{XVLMT} If NS is 1 in a Realm EL2 and EL2&0 stage 1 Block or Page descriptor, then the AMEC field is RES0 and does not affect MECID selection.

R_{VKDBG} MECID_P0_EL2, MECID_P1_EL2, MECID_A0_EL2, and MECID_A1_EL2 are not permitted to be cached in a TLB.

D8.12.1.3 Effect on the Realm EL1&0 translation regime

R_{HDGTR} If SCTLR2_EL2.EMEC is 0, then Realm EL1&0 accesses to Realm PA space use the default MECID of zero.

R_{YSNHS} If SCTLR2_EL2.EMEC is 1 and HCR_EL2.VM is 0, then Realm EL1&0 accesses to Realm PA space use the EL1&0 Primary MECID, configured in VMECID_P_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{FQFXK} For Realm EL1&0 stage 1 translations, if SCTLR2_EL2.EMEC is 1, HCR_EL2.VM is 0, and SCTLR_EL1.M is 1, then all lookup levels use the EL1&0 Primary MECID, configured in VMECID_P_EL2.

R_{PDCWV} For Realm EL1&0 stage 2 translations, if SCTLR2_EL2.EMEC is 1 and HCR_EL2.VM is 1, then all lookup levels use the EL1&0 Primary MECID, configured in VMECID_P_EL2.

R_{XMTZH} For Realm EL1&0 stage 2 translated addresses to Realm PA space, if SCTLR2_EL2.EMEC is 1 and HCR_EL2.VM is 1, then MECID selection is determined by the AMEC field in the Block or Page descriptor used for that translation.

TTD.AMEC	Output MECID
0	VMECID_P_EL2
1	VMECID_A_EL2

R_{DQZTR} If NS is 1 within a Realm EL1&0 stage 2 Block or Page descriptor, then the AMEC field is RES0 and does not affect MECID selection.

R_{JQSMC} VMECID_P_EL2 and VMECID_A_EL2 are not permitted to be cached in a TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.13 Virtualization Host Extensions

I_{QJTJN} Virtualization Host Extensions provide enhanced support for a Type 2 virtualization solution, where there is a Host OS that is either more privileged than the hypervisor or is the hypervisor.

R_{SJSDF} All statements in this chapter require implementation of FEAT_VHE.

R_{CSTLG} If and only if an implementation includes EL2 using AArch64, then Virtualization Host Extensions, FEAT_VHE, can apply.

R_{GNLPT} FEAT_VHE adds all of the following state:

- The HCR_EL2.E2H configuration bit.
- The CONTEXTIDR_EL2 register, which has the same format and contents as the CONTEXTIDR_EL1 register.
- The TTBR1_EL2 register, which has the same format and contents as the TTBR1_EL1 register.
- The CNTHV_CTL_EL2, CNTHV_CVAL_EL2, and CNTHV_TVAL_EL2 registers, which have the same format and contents as the CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 registers, respectively.
- An EL2 virtual timer with all of the following properties:
  - It is accessed using the CNTHV_CTL_EL2, CNTHV_CVAL_EL2, and CNTHV_TVAL_EL2 registers.
  - The virtual offset is treated as 0.

D8.13.1 Behavior of HCR_EL2.E2H

I_{SSMWM} Setting HCR_EL2.E2H to 1 enables a configuration where a host operating system is running in EL2, and the host operating system applications are running in EL0. The host operating system might also manage guest virtual machines that run in EL1&0, with stage 2 translation enabled and controlled by the host operating system.

I_{CDMXB} HCR_EL2.E2H is RES1 unless FEAT_E2H0 is implemented. Arm deprecates setting HCR_EL2.E2H to 0.

R_{VJBPM} If the Effective value of HCR_EL2.E2H is 0, then all of the following apply:

- Execution at EL2 uses the EL2 translation regime.
- All of the following effects apply to the contents of TTBR1_EL2:
  - The contents can be read by an `MRS` instruction and written by an `MSR` instruction.
  - For all other hardware operations, the contents are ignored.
- The Context ID matching breakpoint has all of the following properties:
  - It is disabled at EL2.
  - It uses the value of CONTEXTIDR_EL1 at EL0 and EL1.

R_{MNWSS} If the Effective value of HCR_EL2.E2H is 1, then the translation regime controlled by TCR_EL2 is the EL2&0 translation regime.

I_{STSYY} HCR_EL2.E2H is permitted to be cached in a TLB, and EL2 software is expected to perform a `TLBI ALLE2` operation after changing the value of HCR_EL2.E2H.

R_{QXCQP} All of the following are properties of the EL2&0 translation regime:

- The EL2&0 translation regime behaves the same as stage 1 in the EL1&0 translation regime and uses an upper address range translated by tables pointed to by TTBR1_EL2.
- TTBR0_EL2 translates the lower address range of the EL2&0 translation regime and is extended to have the same format and contents as TTBR0_EL1.
- The Privileged Access Never mechanism applies to data accesses from EL2.
- When CNTVCT_EL0 is read from EL2, the virtual offset is reported as 0.
- All of the following registers are redefined:
  - CNTHCTL_EL2.
  - CPTR_EL2.
  - TCR_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{BLXMQ} If the Effective value of HCR_EL2.E2H is 1, then all of the following effects to the Context ID apply:

- When executing at EL2, a Context ID matching breakpoint uses CONTEXTIDR_EL2.
- Both VMID and VMID plus Context ID matching breakpoints do not match at EL2.

R_{FLVFR} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}, then accesses from EL1 and EL0 use the EL1&0 translation regime.

R_{VMWQQ} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then when executing at EL0, all of the following apply:

- The EL2&0 translation regime is used and accesses are treated as unprivileged.
- A Context ID matching breakpoint uses CONTEXTIDR_EL2.
- The following timer registers, and their equivalent AArch32 registers, are redefined to access the associated EL2 register, rather than accessing the EL0 register:
  - CNTP_CTL_EL0.
  - CNTP_CVAL_EL0.
  - CNTP_TVAL_EL0.
  - CNTV_CTL_EL0.
  - CNTV_CVAL_EL0.
  - CNTV_TVAL_EL0.
- Both VMID and VMID plus Context ID matching breakpoints do not match at EL0.

R_{CFMND} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then when executing at EL2, all of the following apply:

- The EL2&0 translation regime is used.
- For permission and watchpoint checking, the behavior of unprivileged memory access instructions executed at EL2 is determined by the Effective value of PSTATE.UAO.

R_{XCLCK} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then all of the following register effects apply:

- Except for the purpose of reading the value held in the register, some fields in HCR_EL2 and all fields in HSTR_EL2 are treated as effectively having a specific value.
- SCTLR_EL2 is redefined in all of the following ways:
  - Additional fields from SCTLR_EL1 are included.
  - The register applies to execution at EL0.
- When CNTVCT_EL0 is read from EL0 or EL2, the virtual offset is treated as 0.
- SCTLR_EL1.UMA and SCTLR_EL1.A bit are both treated as 0 for all purposes other than reading the value of the register.

R_{XXZKM} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then when an exception is taken from EL0 to EL2, the value of the HCR_EL2.RW bit is ignored when determining the exception vector offset to use.

R_{QWHVM} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then the TLB maintenance and address translation instructions that apply to the EL1&0 translation regime are redefined to apply to the EL2&0 translation regime.

R_{HGWKR} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then when executing at EL2 or EL0, any physical interrupt that is configured to be taken at EL2 is subject to the PSTATE.{D, A, I, F} interrupt masks, and if FEAT_NMI is implemented, then the effect of PSTATE.ALLINT.

For more information, see The PSTATE debug mask bit, D and Asynchronous exception masking.

R_{KJKRH} If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then all of the following apply to EL1:

- Access from EL1 is not possible.
- CPACR_EL1 does not cause any instructions to be trapped to EL1.
- CNTKCTL_EL1 does not cause any instructions to be trapped to EL1, and the event stream caused by CNTKCTL_EL1 is disabled.

D8.13.2 System and Special-purpose register redirection

R_{JGGMV} If all of the conditions in the following list are true, then the System register mapping in the following table applies:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PE is executing at EL2.
- The Effective value of HCR_EL2.E2H is 1.

Table D8-113 System register redirection

Specified EL1 System register	Equivalent EL2 System register accessed at EL2
ACTLRMASK_EL1	ACTLRMASK_EL2
AFSR0_EL1	AFSR0_EL2
AFSR1_EL1	AFSR1_EL2
AMAIR_EL1	AMAIR_EL2
AMAIR2_EL1	AMAIR2_EL2
BRBCR_EL1	BRBCR_EL2
CPACRMASK_EL1	CPTRMASK_EL2
CNTKCTL_EL1	CNTHCTL_EL2
CONTEXTIDR_EL1	CONTEXTIDR_EL2
CPACR_EL1	CPTR_EL2
ESR_EL1	ESR_EL2
FAR_EL1	FAR_EL2
GCSCR_EL1	GCSCR_EL2
GCSPR_EL1	GCSPR_EL2
MAIR_EL1	MAIR_EL2
MAIR2_EL1	MAIR2_EL2
MPAM1_EL1	MPAM2_EL2
MPAMBW1_EL1	MPAMBW2_EL2
PFAR_EL1	PFAR_EL2
PIR_EL1	PIR_EL2
PIRE0_EL1	PIRE0_EL2
PMSCR_EL1	PMSCR_EL2
POR_EL1	POR_EL2
SCTLR_EL1	SCTLR_EL2
SCTLR2_EL1	SCTLR2_EL2
SCTLRMASK_EL1	SCTLRMASK_EL2
SCTLR2MASK_EL1	SCTLR2MASK_EL2
SCXTNUM_EL1	SCXTNUM_EL2
SMCR_EL1	SMCR_EL2
SPMACCESSR_EL1	SPMACCESSR_EL2
TCR_EL1	TCR_EL2
TCR2_EL1	TCR2_EL2
TCRMASK_EL1	TCRMASK_EL2
TCR2MASK_EL1	TCR2MASK_EL2
TFSR_EL1	TFSR_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Specified EL1 System register	Equivalent EL2 System register accessed at EL2
TRCITECR_EL1	TRCITECR_EL2
TRFCR_EL1	TRFCR_EL2
TTBR0_EL1	TTBR0_EL2
TTBR1_EL1	TTBR1_EL2
VBAR_EL1	VBAR_EL2
ZCR_EL1	ZCR_EL2

R_{RZRWZ} If all of the conditions in the following list are true, then the System register mapping in the following table applies:

- The PE is executing at EL2.
- The Effective value of HCR_EL2.E2H is 1.
- The Effective value of SCR_EL3.NS is 1.

Table D8-114 Additional System register redirection when the Effective value of SCR_EL3.NS is 1

Specified System register	Equivalent EL2 System register accessed at EL2
CNTP_CTL_EL0	CNTHP_CTL_EL2
CNTP_CVAL_EL0	CNTHP_CVAL_EL2
CNTP_TVAL_EL0	CNTHP_TVAL_EL2
CNTV_CTL_EL0	CNTHV_CTL_EL2
CNTV_CVAL_EL0	CNTHV_CVAL_EL2
CNTV_TVAL_EL0	CNTHV_TVAL_EL2

R_{LLSLV} If all of the conditions in the following list are true, then the System register mapping in the following table applies:

- The PE is executing at EL2.
- The Effective value of HCR_EL2.E2H is 1.
- The Effective value of SCR_EL3.NS is 0.

Table D8-115 Additional System register redirection when the Effective value of SCR_EL3.NS is 0

Specified EL1 System register	Equivalent EL2 System register accessed at EL2
CNTP_CTL_EL0	CNTHPS_CTL_EL2
CNTP_CVAL_EL0	CNTHPS_CVAL_EL2
CNTP_TVAL_EL0	CNTHPS_TVAL_EL2
CNTV_CTL_EL0	CNTHVS_CTL_EL2
CNTV_CVAL_EL0	CNTHVS_CVAL_EL2
CNTV_TVAL_EL0	CNTHVS_TVAL_EL2

R_{ZNNRM} If all of the conditions in the following list are true, then the Special-purpose register mapping in the following table applies:

- The PE is executing at EL2.
- The Effective value of HCR_EL2.E2H is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-116 Special-purpose register redirection

Specified EL1 Special-purpose register	Equivalent EL2 Special-purpose register accessed at EL2
ELR_EL1	ELR_EL2
SPSR_EL2	SPSR_EL1

I_{WJCBD} For more information, see System and Special-purpose register aliasing.

D8.13.3 System and Special-purpose register aliasing

R_{KSPRM} If all of the conditions in the following list are true, then the System register mapping in the following table applies:

- FEAT_SRMASK is implemented.
- The PE is executing at EL1.

Table D8-117 EL1 System register aliases

Alias mnemonic	EL1 System register accessed at EL1
ACTLRALIAS_EL1	ACTLR_EL1
CPACRALIAS_EL1	CPACR_EL1
SCTLRALIAS_EL1	SCTLR_EL1
SCTLR2ALIAS_EL1	SCTLR2_EL1
TCRALIAS_EL1	TCR_EL1
TCR2ALIAS_EL1	TCR2_EL1

R_{PHHPL} If all of the conditions in the following list are true, then the System register mapping in the following table applies. The aliases are UNDEFINED at EL1 and EL0.

- EL2 is enabled in the current Security state.
- The Effective value of HCR_EL2.E2H is 1.
- The PE is executing at EL2 or EL3.

Table D8-118 System register aliases

Alias mnemonic	EL1 or EL0 System register accessed at EL2 or EL3
ACTLRMASK_EL12	ACTLRMASK_EL1
AFSR0_EL12	AFSR0_EL1
AFSR1_EL12	AFSR1_EL1
AMAIR_EL12	AMAIR_EL1
AMAIR2_EL12	AMAIR2_EL1
BRBCR_EL12	BRBCR_EL1
CNTKCTL_EL12	CNTKCTL_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias mnemonic	EL1 or EL0 System register accessed at EL2 or EL3
CNTP_CTL_EL02	CNTP_CTL_EL0
CNTP_CVAL_EL02	CNTP_CVAL_EL0
CNTP_TVAL_EL02	CNTP_TVAL_EL0
CNTV_CTL_EL02	CNTV_CTL_EL0
CNTV_CVAL_EL02	CNTV_CVAL_EL0
CNTV_TVAL_EL02	CNTV_TVAL_EL0
CONTEXTIDR_EL12	CONTEXTIDR_EL1
CPACR_EL12	CPACR_EL1
CPACRMASK_EL12	CPACRMASK_EL1
ESR_EL12	ESR_EL1
FAR_EL12	FAR_EL1
GCSCR_EL12	GCSCR_EL1
GCSPR_EL12	GCSPR_EL1
MAIR_EL12	MAIR_EL1
MAIR2_EL12	MAIR2_EL1
MPAM1_EL12	MPAM1_EL1
MPAMBW1_EL12	MPAMBW1_EL1
PFAR_EL12	PFAR_EL1
PIR_EL12	PIR_EL1
PIRE0_EL12	PIRE0_EL1
PMSCR_EL12	PMSCR_EL1
POR_EL12	POR_EL1
SCTLR_EL12	SCTLR_EL1
SCTLRMASK_EL12	SCTLRMASK_EL1
SCTLR2_EL12	SCTLR2_EL1
SCTLR2MASK_EL12	SCTLR2MASK_EL1
SCXTNUM_EL12	SCXTNUM_EL1
SMCR_EL12	SMCR_EL1
SPMACCESSR_EL12	SPMACCESSR_EL1
TCR_EL12	TCR_EL1
TCRMASK_EL12	TCRMASK_EL1
TCR2_EL12	TCR2_EL1
TCR2MASK_EL12	TCR2MASK_EL1
TFSR_EL12	TFSR_EL1
TRCITECR_EL12	TRCITECR_EL1
TRFCR_EL12	TRFCR_EL1
TTBR0_EL12	TTBR0_EL1
TTBR1_EL12	TTBR1_EL1
VBAR_EL12	VBAR_EL1
ZCR_EL12	ZCR_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{KTWST} If all of the conditions in the following list are true, then the Special-purpose register mapping in the following table applies. The aliases are UNDEFINED at EL1 and EL0.

- EL2 is enabled in the current Security state.
- The Effective value of HCR_EL2.E2H is 1.
- The PE is executing at EL2 or EL3.

Table D8-119 Special-purpose register aliases

Alias mnemonic	Equivalent EL1 Special-purpose register accessed at EL2 or EL3
ELR_EL12	ELR_EL1
SPSR_EL12	SPSR_EL1

I_{ZXNTN} For more information, see System and Special-purpose register redirection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.14 Nested virtualization

I_{FDMLZ} Nested virtualization is an OPTIONAL feature that enables a Host hypervisor executing at EL2 to run a Guest hypervisor at EL1.

R_{QQDQK} All statements in this section and subsections require implementation of FEAT_NV.

I_{CDTFC} Nested virtualization adds the HCR_EL2.{NV, NV1, AT} fields.

I_{WHFTT} An implementation is permitted to cache the HCR_EL2.{NV, NV1} fields in a TLB.

I_{CQNPK} If FEAT_E2H0 is not implemented, it is permitted to implement HCR_EL2.NV1 as RES0. See ID_AA64MMFR4_EL1.

S_{RHQHT} If the Host hypervisor has set HCR_EL2.NV1 to 1 because a Guest hypervisor believes that HCR_EL2.E2H is 0, the Host hypervisor should also set both HCR_EL2.TVM and CPTR_EL2.TCPAC to 1 to trap Guest hypervisor accesses to EL1 System registers.

I_{FPHRW} Nested virtualization does not modify either self-hosted debug or the Performance Monitors Extension.

I_{FFWLP} Arm assumes that the Host hypervisor traps accesses to the Breakpoint registers and Performance Monitors registers to EL2, so that it can process any accesses to these registers made by a Guest hypervisor or by a Guest OS running under the Guest hypervisor.

D8.14.1 Behavior when HCR_EL2.NV is 1

I_{ZMLPV} Nested Virtualization traps functionality that is permitted at EL2 and would be UNDEFINED at EL1 if HCR_EL2.NV was 0.

I_{TZTZL} If HCR_EL2.NV is 1 and the current Exception level is EL1, then all of the following apply:

- If a read access or write access from EL1 to one of the following System registers or Special-purpose registers occurs and that access is not UNDEFINED at EL2 in the current Security state and not trapped to EL3 by a higher priority trap, then that access is trapped to EL2 and reported using EC syndrome value `0x18` in ESR_EL2:
  - Any System register accessed using `MRS` or `MSR` named *_EL2, except all of the following:
    - SP_EL2.
    - MECIDR_EL2, MECID_A0_EL2, MECID_A1_EL2, MECID_P0_EL2, and MECID_P1_EL2.
    - VMECID_A_EL2 and VMECID_P_EL2.
  - Any System register accessed using `MRS` or `MSR` named *_EL12.
  - Any System register accessed using `MRS` or `MSR` named *_EL02.
  - Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und, and SPSR_fiq, accessed using `MRS` or `MSR`.
  - Special-purpose register SP_EL1 accessed using the dedicated `MRS` or `MSR` instruction.
- If one of the following instructions is executed from EL1, then the instruction is trapped to EL2 and reported using EC syndrome value `0x18` in ESR_EL2:
  - Address Translation instructions that are accessible only from EL2 and above.
  - TLB maintenance instructions that are accessible only from EL2 and above.
- If the `ERET`, `ERETAA`, or `ERETAB` instruction is executed from EL1, then the instruction is trapped to EL2 and reported using EC syndrome value `0x1A` in ESR_EL2.
- If EL3 is not implemented and the Effective value of HCR_EL2.TSC is 1, then when an SMC instruction is executed at EL1, all of the following apply:
  - The SMC instruction is trapped to EL2.
  - The exception is reported using EC syndrome value `0x17` in ESR_EL2.

R_{GFLDX} If HCR_EL2.NV is 1 and the current Exception level is EL1, then EL1 read accesses to the CurrentEL register return a value of `0x2` in bits[3:2].

D8.14.2 Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 0

I_{ZJRNN} If the Effective value of HCR_EL2.{NV, NV1} is {1, 0} and the current Exception level is EL1, then any exception taken from EL1 to EL1 causes SPSR_EL1.M[3:2] to be set to `0b10` and not `0b01`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.14.3 Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1

I_{JKLJK} If the Effective value of HCR_EL2.{NV, NV1} is {1, 1} and the current Exception level is EL1, then all of the following apply:

- If an access to one of the following registers occurs, then the access is trapped to EL2 and reported using EC syndrome value `0x18` in ESR_EL2:
  - An access to VBAR_EL1.
  - An access to ELR_EL1.
  - An access to SPSR_EL1.
  - If implemented, then an access from EL1 to SCXTNUM_EL1.
- For Block descriptors and Page descriptors in the EL1&0 translation regime, all of the following apply:
  - Block descriptor and Page descriptor bit[54] holds PXN, not UXN.
  - Block descriptor and Page descriptor bit[53] is RES0.
  - Block descriptor and Page descriptor bit[6], AP[1], is treated as 0 regardless of the actual value.
- For Table descriptors in the EL1&0 translation regime, if hierarchical permissions are enabled, then all of the following apply:
  - Table descriptor bit[61], APTable[0], is treated as 0 regardless of the actual value.
  - Table descriptor bit[60] holds PXNTable, not UXNTable.
  - Table descriptor bit[59] is RES0.
- PSTATE.PAN is treated as 0 for all purposes except reading the value of the bit.
- The Effective value of PSTATE.UAO is 1.

I_{FQZDC} Arm expects software to clear the HCR_EL2.NV1 bit to 0 before permitting execution at EL0.

I_{FWVWB} If ID_AA64MMFR4_EL1.E2H0 is `0b1110`, then HCR_EL2.NV1 is RES0.

D8.14.4 Behavior when HCR_EL2.NV is 0 and HCR_EL2.NV1 is 1

I_{KZNPS} If the Effective value of HCR_EL2.{NV, NV1} is {0, 1} and the current Exception level is EL1, then one of the following CONSTRAINED UNPREDICTABLE behaviors apply:

- For all purposes other than reading back the value of the HCR_EL2.NV bit, the implementation behaves as if HCR_EL2.{NV, NV1} is {1, 1}.
- For all purposes other than reading back the value of the HCR_EL2.NV1 bit, the implementation behaves as if HCR_EL2.{NV, NV1} is {0, 0}.
- The implementation behaves as defined when HCR_EL2.NV is 0, with HCR_EL2.NV1 set to 1 having the effect of trapping to EL2 the same set of registers that are trapped when HCR_EL2.{NV, NV1} is {1, 1}, as defined in I_{JKLJK}.

D8.14.5 Effect of HCR_EL2.AT

I_{RBYYS} If HCR_EL2.AT is 1, then all of the following are trapped to EL2 and reported using EC syndrome value `0x18` in ESR_EL2:

- Executing `AT S1E0R` from EL1.
- Executing `AT S1E0W` from EL1.
- Executing `AT S1E1R` from EL1.
- Executing `AT S1E1W` from EL1.
- Executing `AT S1E1RP` from EL1.
- Executing `AT S1E1WP` from EL1.
- If FEAT_ATS1A is implemented, executing `AT S1E1A` from EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.14.6 Enhanced support for nested virtualization

I_{GDQNK} Enhanced support for nested virtualization provides a mechanism for hardware to transform reads and writes from System registers into reads and writes from memory.

R_{JXWDQ} All statements in this section and subsections require implementation of FEAT_NV2.

I_{TMCKW} If the Effective value of HCR_EL2.NV2 is 1 and the current Exception level is EL1, then all of the following apply:

- Accesses to EL2 registers are redirected to EL1 registers.
- Accesses to System registers are transformed to memory accesses.

I_{VGFWB} If the Effective value of HCR_EL2.NV2 is 0, then the behavior of HCR_EL2.{NV, NV1} is unchanged.

I_{ZSQQK} If ID_AA64MMFR4_EL1.NV_frac is `0b0001` and HCR_EL2.NV is 1, then the Effective value of HCR_EL2.NV2 is 1.

D8.14.6.1 Redirection of register accesses from EL2 to EL1

I_{RKLVD} If the Effective value of HCR_EL2.{NV, NV2} is {1, 1} and the current Exception level is EL1, then accesses to all of the following EL2 Special-purpose registers are redirected to the corresponding EL1 register:

- An access from EL1 to SPSR_EL2 is redirected to SPSR_EL1.
- An access from EL1 to ELR_EL2 is redirected to ELR_EL1.

I_{DLGPW} If the Effective value of HCR_EL2.{NV, NV2} is {1, 1} and the current Exception level is EL1, then accesses to all of the following EL2 System registers are redirected to the corresponding EL1 register:

- An access from EL1 to ESR_EL2 is redirected to ESR_EL1.
- An access from EL1 to FAR_EL2 is redirected to FAR_EL1.
- An access from EL1 to TFSR_EL2 is redirected to TFSR_EL1.

I_{HFNHT} For more information, see:

- op0 == `0b11`, Moves to and from Special-purpose registers.
- Instructions for accessing non-debug System registers.

D8.14.6.2 Loads and stores generated by transforming register accesses

I_{KYDSS} If the Effective value of HCR_EL2.{NV, NV2} is {1, 1} and the current Exception level is EL1, then when an MRS or MSR instruction is executed to access one of the registers listed in Table D8-120, all of the following apply:

- The instruction is treated as an instruction executed at EL1.
- The register accesses do not generate an exception to EL2.
- The register accesses are transformed to memory accesses.
- The memory accesses are treated as EL2 accesses.

I_{LKWDD} When register accesses are transformed to memory accesses, all of the following information is used to form the memory address:

- A base address that is stored in the VNCR_EL2 register.
- For a register that is redirected to memory, a unique memory offset value.
- The memory address is a combination of the base address and the memory offset according to the formula `SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64)`.

R_{CSRPQ} When register accesses are transformed to memory accesses, the following table shows the unique memory offset that is assigned to the register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D8-120 Memory address offset associated with transformed register access

Register access if the Effective value of HCR_EL2.NV1 is 0	Register access if the Effective value of HCR_EL2.NV1 is 1	Memory offset
VTTBR_EL2	VTTBR_EL2	0x20
VSTTBR_EL2	VSTTBR_EL2	0x30
VTCR_EL2	VTCR_EL2	0x40
VSTCR_EL2	VSTCR_EL2	0x48
VMPIDR_EL2	VMPIDR_EL2	0x50
CNTVOFF_EL2	CNTVOFF_EL2	0x60
HCR_EL2	HCR_EL2	0x78
HSTR_EL2	HSTR_EL2	0x80
VPIDR_EL2	VPIDR_EL2	0x88
TPIDR_EL2	TPIDR_EL2	0x90
HCRX_EL2	HCRX_EL2	0xA0
VNCR_EL2	VNCR_EL2	0xB0
CPACR_EL12	CPACR_EL1	0x100
CONTEXTIDR_EL12	CONTEXTIDR_EL1	0x108
SCTLR_EL12	SCTLR_EL1	0x110
ACTLR_EL1	ACTLR_EL1	0x118
TCR_EL12	TCR_EL1	0x120
AFSR0_EL12	AFSR0_EL1	0x128
AFSR1_EL12	AFSR1_EL1	0x130
ESR_EL12	ESR_EL1	0x138
MAIR_EL12	MAIR_EL1	0x140
AMAIR_EL12	AMAIR_EL1	0x148
MDSCR_EL1	MDSCR_EL1	0x158
SPSR_EL12	SPSR_EL1	0x160
CNTV_CVAL_EL02	CNTV_CVAL_EL0	0x168
CNTV_CTL_EL02	CNTV_CTL_EL0	0x170
CNTP_CVAL_EL02	CNTP_CVAL_EL0	0x178
CNTP_CTL_EL02	CNTP_CTL_EL0	0x180
SCXTNUM_EL12	SCXTNUM_EL1	0x188
TFSR_EL12	TFSR_EL1	0x190
HDFGRTR2_EL2	HDFGRTR2_EL2	0x1A0
CNTPOFF_EL2	CNTPOFF_EL2	0x1A8
HDFGWTR2_EL2	HDFGWTR2_EL2	0x1B0
HFGRTR_EL2	HFGRTR_EL2	0x1B8
HFGWTR_EL2	HFGWTR_EL2	0x1C0
HFGITR_EL2	HFGITR_EL2	0x1C8
HDFGRTR_EL2	HDFGRTR_EL2	0x1D0
HDFGWTR_EL2	HDFGWTR_EL2	0x1D8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register access if the Effective value of HCR_EL2.NV1 is 0	Register access if the Effective value of HCR_EL2.NV1 is 1	Memory offset
ZCR_EL12	ZCR_EL1	0x1E0
HAFGRTR_EL2	HAFGRTR_EL2	0x1E8
SMCR_EL12	SMCR_EL1	0x1F0
SMPRIMAP_EL2	SMPRIMAP_EL2	0x1F8
TTBR0_EL12	TTBR0_EL1	0x200
TTBR1_EL12	TTBR1_EL1	0x210
FAR_EL12	FAR_EL1	0x220
ELR_EL12	ELR_EL1	0x230
SP_EL1	SP_EL1	0x240
VBAR_EL12	VBAR_EL1	0x250
TCR2_EL12	TCR2_EL1	0x270
SCTLR2_EL12	SCTLR2_EL1	0x278
MAIR2_EL12	MAIR2_EL1	0x280
AMAIR2_EL12	AMAIR2_EL1	0x288
PIRE0_EL12	PIRE0_EL1	0x290
PIRE0_EL1	PIRE0_EL2	0x298
PIR_EL12	PIR_EL1	0x2A0
POR_EL12	POR_EL1	0x2A8
S2PIR_EL2	S2PIR_EL2	0x2B0
S2POR_EL1	S2POR_EL1	0x2B8
HFGRTR2_EL2	HFGRTR2_EL2	0x2C0
HFGWTR2_EL2	HFGWTR2_EL2	0x2C8
PFAR_EL12	PFAR_EL1	0x2D0
HFGITR2_EL2	HFGITR2_EL2	0x310
SCTLRMASK_EL12	SCTLRMASK_EL1	0x318
CPACRMASK_EL12	CPACRMASK_EL1	0x320
SCTLR2MASK_EL12	SCTLR2MASK_EL1	0x328
TCRMASK_EL12	TCRMASK_EL1	0x330
TCR2MASK_EL12	TCR2MASK_EL1	0x338
ACTLRMASK_EL12	ACTLRMASK_EL1	0x340
ICH_LR<n>_EL2	ICH_LR<n>_EL2	0x400+8*n
ICH_AP0R<n>_EL2	ICH_AP0R<n>_EL2	0x480+8*n
ICH_AP1R<n>_EL2	ICH_AP1R<n>_EL2	0x4A0+8*n
ICH_HCR_EL2	ICH_HCR_EL2	0x4C0
ICH_VMCR_EL2	ICH_VMCR_EL2	0x4C8
VDISR_EL2	VDISR_EL2	0x500
VSESR_EL2	VSESR_EL2	0x508
PMBLIMITR_EL1	PMBLIMITR_EL1	0x800
PMBPTR_EL1	PMBPTR_EL1	0x810
PMBSR_EL1	PMBSR_EL1	0x820

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register access if the Effective value of HCR_EL2.NV1 is 0	Register access if the Effective value of HCR_EL2.NV1 is 1	Memory offset
PMSCR_EL12	PMSCR_EL1	0x828
PMSEVFR_EL1	PMSEVFR_EL1	0x830
PMSICR_EL1	PMSICR_EL1	0x838
PMSIRR_EL1	PMSIRR_EL1	0x840
PMSLATFR_EL1	PMSLATFR_EL1	0x848
PMSNEVFR_EL1	PMSNEVFR_EL1	0x850
PMSDSFR_EL1	PMSDSFR_EL1	0x858
TRFCR_EL12	TRFCR_EL1	0x880
TRCITECR_EL12	TRCITECR_EL1	0x888
GCSPR_EL12	GCSPR_EL1	0x8C0
GCSCR_EL12	GCSCR_EL1	0x8D0
BRBCR_EL12	BRBCR_EL1	0x8E0
SPMACCESSR_EL12	SPMACCESSR_EL1	0x8E8
MPAM1_EL12	MPAM1_EL1	0x900
MPAMBW1_EL12	MPAMBW1_EL1	0x908
MPAMBWCAP_EL2	MPAMBWCAP_EL2	0x910
MPAMHCR_EL2	MPAMHCR_EL2	0x930
MPAMVPMV_EL2	MPAMVPMV_EL2	0x938
MPAMVPM0_EL2	MPAMVPM0_EL2	0x940
MPAMVPM1_EL2	MPAMVPM1_EL2	0x948
MPAMVPM2_EL2	MPAMVPM2_EL2	0x950
MPAMVPM3_EL2	MPAMVPM3_EL2	0x958
MPAMVPM4_EL2	MPAMVPM4_EL2	0x960
MPAMVPM5_EL2	MPAMVPM5_EL2	0x968
MPAMVPM6_EL2	MPAMVPM6_EL2	0x970
MPAMVPM7_EL2	MPAMVPM7_EL2	0x978
AMEVCNTVOFF0<n>_EL2	AMEVCNTVOFF0<n>_EL2	0xA00+8*n
AMEVCNTVOFF1<n>_EL2	AMEVCNTVOFF1<n>_EL2	0xA80+8*n

R_{QSKNK} Unallocated memory offset values up to but not including `0x1000` are reserved.

R_{JSCYJ} When a register access instruction targets a register that is not implemented, the PE treats access to that register as unallocated.

R_{DHMDM} When a System register access instruction is trapped by either or both of the Effective values of HCR_EL2.{NV, NV1}, then the instruction is transformed into a memory access instruction instead of creating a trap.

I_{TFJCH} When a System register access instruction is not trapped by either or both of the Effective values of HCR_EL2.{NV, NV1}, then the trap is subject to the exception prioritization rules.

I_{RSCBR} Accesses to all of the following registers that affect hypervisor execution by controlling the event stream are not transformed into memory accesses:

- CNTHCTL_EL2.
- If the Effective value of HCR_EL2.NV1 is 0, then CNTKCTL_EL12.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of HCR_EL2.NV1 is 1, then CNTKCTL_EL1.

R_{VFMQB} When an MSR or MRS System register access is transformed into a memory access, the memory access has all of the following properties:

- The memory access is translated by one of the following:
  - If the Effective value of HCR_EL2.E2H is 0, then the EL2 translation regime.
  - If the Effective value of HCR_EL2.E2H is 1, then the EL2&0 translation regime.
- The endianness of the memory access is defined by SCTLR_EL2.EE.
- The memory access is 64-bit single-copy atomic aligned to 64 bits.
- The memory access does not have Acquire or Release semantics.
- The memory access is ordered as if it is generated by a load or store instruction.
- The memory access behaves as if PSTATE.PAN is 0.
- For fields in a transformed System register that are defined to be RES0 or RES1, the memory access does not check that the fields are set correctly, nor are they forced to their dedicated value.

R_{BSBZP} When an MSRR or MRRS System register access is transformed into a memory access, the memory access has all of the following properties:

- The memory access is 128-bit aligned.
- If the target memory region is Inner and Outer Write-back Cacheable, then the access is 128-bit single-copy atomic.
- If the target memory region is not Inner and Outer Write-back Cacheable, then it is CONSTRAINED UNPREDICTABLE whether the access is 64-bit or 128-bit single-copy atomic.
- All other properties of the memory access are the same as for MSR and MRS operations that are transformed to memory accesses.

I_{CRPJG} All memory address offset values associated with transformed register accesses for 128-bit System registers are aligned to 16-bytes.

I_{TRBJN} For more information, see Prioritization of Synchronous exceptions taken to AArch64 state.

D8.14.6.3 Exceptions from transformed register accesses

I_{DBTLM} If the Effective value of HCR_EL2.{NV, NV2} is {1, 1}, then any exception taken from EL1 and taken to EL1 causes the SPSR_EL1.M[3:2] to be set to 0b10 and not 0b01.

R_{YWCZS} If a System register access is transformed to a memory access, then when the memory access generates a Data Abort, the resulting fault has all of the following properties:

- The fault is taken to EL2, using the standard vector offset for exceptions from EL1 to EL2.
- The fault is reported as a Data Abort from the current Exception level with the ESR_EL2.EC code 0x25.
- FAR_EL2 is updated to hold the faulting address.

R_{FRWJX} If a System register access is transformed to a memory access, then when the memory access generates a synchronous External abort and External aborts are not configured to be taken to EL3, the resulting fault has all of the following properties:

- The fault is taken to EL2, using the standard vector offset for exceptions from EL1 to EL2.
- The fault is reported as a Data Abort from the current Exception level with the ESR_EL2.EC code 0x25.

I_{ZLKQZ} The VNCR field in ESR_EL2 and ESR_EL3 identifies whether the fault came from use of VNCR_EL2 by EL1.

I_{BKFTF} For more information, see ISS encoding for an exception from a Data Abort.

D8.14.6.4 Interaction with self-hosted and External debug

R_{CJQGX} When a System register access is transformed into a memory access, all of the following operations treat the instruction as being executed at EL1:

- PMU events filtered by Exception level.
- For trace or Statistical Profiling, instructions filtered by Exception level.
- Checking the instruction address against breakpoint registers or trace resources.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{ZZSMN} When a System register access is transformed into a memory access, all of the following operations treat the memory access as being executed at EL2:

- The memory access is checked against the watchpoint registers.
- The memory address is recorded in a Statistical Profiling record.

R_{ZNPSG} If all of the following apply, then it is CONSTRAINED UNPREDICTABLE whether there is a watchpoint match:

- A System register access is transformed into a memory access.
- The memory access matches an EL2 access in the watchpoint registers.
- A watchpoint is linked to a context-aware breakpoint that is programmed to match the value held in CONTEXTIDR_EL1 or VMID.

R_{DDNDL} If all of the following apply, then a watchpoint match generates a Watchpoint debug event:

- A System register access is transformed into a memory access.
- EDSCR.HDE is 1.
- Halting is allowed.
- OS Lock is unlocked.

R_{YGWHK} If all of the following apply, then a watchpoint match generates a Watchpoint exception:

- A System register access is transformed into a memory access.
- EDSCR.HDE is 0 or halting is prohibited in current security state.
- Debug exceptions are enabled at EL2.
- OS Lock is unlocked.

R_{DGCTK} If a System register access is transformed to a memory access, then when a watchpoint match generates a Watchpoint exception, the resulting exception has all of the following properties:

- The exception is taken to EL2.
- The fault is reported as a Watchpoint from the current Exception level with the ESR_EL2.EC code `0x35`.
- FAR_EL2 is updated to hold the watchpointed address.

R_{VYZMX} If a System register access is transformed to a memory access, then when a Watchpoint exception is generated, the VNCR field in ESR_EL2 identifies whether the Watchpoint exception came from use of VNCR_EL2 by EL1.

R_{HZJLY} If a System register access is transformed to a memory access, then the resulting loads and stores are treated by the Performance Monitors as Memory-read operations and Memory-write operations.

R_{MVNHT} If a System register access is transformed to a memory access, then when the Statistical Profiling Unit selects the instruction generating the memory access for profiling, it records the operation as a Load/Store operation.

R_{ZVXYL} If a System register access is transformed to a memory access, then when the Statistical Profiling Unit selects the instruction generating the memory access for profiling and Statistical Profiling is disabled at EL2, the VA for the memory access is not recorded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.15 Memory aborts

R_{SQLMR} For a VMSAv8-64 implementation, all of the following abort mechanisms can cause the PE to take an exception due to a failed memory access:

- A *Debug exception* generated by the self-hosted debug system. For more information, see AArch64 Self-hosted Debug.
- An *Alignment fault* generated by a memory access that is not aligned as required by the operation. For more information, see Alignment support.
- An *MMU fault* generated by the fault checking sequence in the appropriate translation regime. For more information, see MMU fault types.
- An *External abort* generated by the memory system and not by self-hosted debug, alignment checking, or MMU checking. For more information, see External aborts.

R_{XXMXW} For AArch64 state, MMU faults are synchronous exceptions that are reported as one of the following:

- Data Aborts due to data memory access faults.
- Instruction Aborts due to instruction fetch faults.

For more information, see Synchronous exception types.

I_{WFYSR} In some cases, an MMU fault generated as part of `AT` instruction execution is reported in PAR_EL1 rather than as a synchronous abort.

I_{YMZWZ} When an MMU fault is generated and reported as an Instruction Abort or Data Abort, the Exception level that the exception is taken to depends on the translation regime and translation stage that caused the fault.

I_{DHWPX} When a memory access abort generates an exception, all of the following registers are used to record context information:

- A Fault Address Register, FAR_ELx.
- An Exception Syndrome Register, ESR_ELx.

I_{XLTCG} When an MMU fault is generated, the fault context saved in the ESR_ELx at the Exception level that the fault is taken to depends on all of the following:

- Whether the fault is reported as an Instruction Abort or as a Data Abort.
- Whether the exception is taken from the same or a lower Exception level.

I_{FVNYG} For an Instruction Abort, only the following two exceptions have a higher priority in the exception priority hierarchy:

- Software Step exception.
- PC alignment fault exception.

I_{SRPXB} For a Data Abort, only Watchpoint exceptions have a lower priority in the exception priority hierarchy.

D8.15.1 MMU fault types

R_{YLSQG} All of the following MMU fault types are supported, and there are no other MMU fault types:

- Alignment fault on a data access. For more information, see Alignment support.
- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- Permission fault.
- TLB conflict abort.
- Granule Protection Check fault (GPC fault). For more information, see GPC faults.

R_{GKZPW} When an attempt to access Device memory generates an MMU fault, no memory access occurs.

R_{TQDJS} If the memory attributes cannot be determined and the resultant memory region could be Device memory, then when an attempt to access that region generates an MMU fault, no memory access occurs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.15.1.1 Translation fault

R_{DYNYQ} If descriptor bit[0] is 0 in any translation table entry, the descriptor is invalid. When a translation table lookup returns an invalid descriptor, a Translation fault is generated and descriptor bits[63:1] are ignored.

R_{KRMJZ} If descriptor bit[1] is 0 at a lookup level that does not support Block descriptors, then the descriptor encoding is reserved. When a translation table lookup returns that descriptor, a Translation fault is generated.

R_{FDQJL} If FEAT_BBML1 is implemented, and the Block descriptor has the nT bit set, then the implementation is permitted to generate a Translation fault.

I_{RRSLQ} For any translation lookup level, when a Translation fault is generated, the reported fault code indicates the lookup level.

R_{VZZSZ} When one or more of the following apply, a level 0 Translation fault is generated on the relevant translation stage:

- The IA does not map onto a TTBR_ELx address range.
- If the IA maps onto the TTBR0_ELx address range and the IA contains any one bits above the configured IA size as determined by TCR_ELx.T0SZ.
- If the IA maps onto the TTBR1_ELx address range and the IA contains any zero bits above the configured IA size as determined by TCR_ELx.T1SZ.
- When a TLB miss occurs, the corresponding TCR_ELx.EPD*n* field prevents a translation table walk using TTBR*n*_ELx.
- When FEAT_E0PD is implemented, the corresponding TCR_ELx.E0PD*n* field prevents unprivileged access to an address translated by TTBR*n*_ELx.

R_{MLNTS} A translation table entry that generates a Translation fault is not cached in a TLB.

I_{XFTPJ} When a Translation fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

I_{PXJFV} For more information, see MMU faults generated by cache maintenance operations.

D8.15.1.2 Address size fault

R_{CVSBH} For a translation stage, if an address in one of the following has nonzero address bits above the configured OA size, then an Address size fault is generated:

- The TTBR_ELx used in the translation.
- A translation table entry.
- The translation OA.

R_{FTZHB} An Address size fault can be generated at any translation lookup level.

R_{WDZFC} For the 64KB translation granule, if the implementation does not support 52-bit PAs, then all of the following apply:

- It is IMPLEMENTATION DEFINED whether descriptor bits[15:12] being nonzero generates an Address size fault.
- For this case, it is deprecated to not generate an Address size fault.

R_{BFHQH} When an Address size fault is generated, the reported fault code indicates one of the following:

- If the fault was generated due to the TTBR_ELx used in the translation having nonzero address bits above the OA size, then a fault at level 0.
- If the fault was not generated due to the TTBR_ELx used in the translation, then the lookup level at which the fault occurred.

R_{GGQPR} When an Address size fault is generated, the translation table entry causing the fault is not cached in a TLB.

I_{PBYZQ} When an Address size fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

I_{WWYQH} An implementation is required to ensure that the Contiguous bit does not prevent generation of Address size faults. For more information, see Architectural guarantees when the Contiguous bit is misprogrammed.

I_{JTTDF} For more information, see:

- MMU faults generated by cache maintenance operations.
- Output address size configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.15.1.3 External abort on a translation table walk

R_{YLGBV} If an External abort on a translation table walk is generated, then it is reported in one of the following ways:

- If the External abort is synchronous, then it is reported as one of the following:
  - If the translation table walk is due to an instruction fetch, then a synchronous Instruction Abort exception.
  - If the translation table walk is due to a data access, an address translation instruction, or a cache maintenance instruction operating by VA, then a synchronous Data Abort exception.
- If the External abort is asynchronous, then it is reported using the SError exception.

I_{JVVQN} A TLB or intermediate TLB caching structure might support:

- An arbitrary number of translation table lookup levels.
- One or more translation stages that might not correspond to an address translation lookup stage.

R_{BGPQR} If a synchronous External abort is generated due to a TLB or intermediate TLB caching structure, including parity or ECC errors, then all of the following are permitted:

- If the PE cannot precisely determine the translation stage at which the error occurred, then it is reported and prioritized as a stage 1 fault.
- If the PE cannot precisely determine the lookup level at which the error occurred, then the lookup level is reported and prioritized as one of the following:
  - The lowest-numbered lookup level that could have caused the error.
  - If the PE cannot determine any information about the lookup level, then level 0.

D8.15.1.4 Access flag fault

I_{BMSDD} For an implementation that does not manage the AF in hardware, when and only when a non-speculative access is made to an address and an attempt is made to translate that address using a descriptor with an AF of 0, an Access flag fault is generated.

R_{LFGXS} For a translation lookup of a Block descriptor or Page descriptor, when an Access flag fault is generated, the reported fault code indicates the lookup level.

I_{PNQBP} When an Access flag fault is generated, the translation table entry causing the fault is not cached in a TLB.

I_{TQXPT} When an Access flag fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

I_{RGFRF} For more information, see:

- MMU faults generated by cache maintenance operations.
- Software management of the Access flag.

D8.15.1.5 Permission fault

R_{CSRQR} When a Permission fault is generated, the reported fault code indicates the lookup level of the Block or Page descriptor used in the translation, even if the lack of permission was caused by hierarchical permission controls.

R_{JVXRH} A translation table entry that generates a Permission fault is permitted to be cached in a TLB.

I_{WJLXV} If software updates a stage 1 or stage 2 translation table due to a Permission fault, then the software is required to invalidate the appropriate TLB entry to prevent stale information in a TLB from being used by a subsequent memory access.

I_{WHWDD} For more information, see:

- Prioritization of Permission faults.
- MMU faults generated by cache maintenance operations.
- Memory access control.
- Stage 2 fault on a stage 1 translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.15.1.6 TLB conflict abort

I_{CNNYQ} When a TLB has not been properly invalidated, such as when architecturally required TLB invalidation is not done, an address lookup might hit multiple TLB entries.

R_{ZQNWZ} When a lookup address hits multiple TLB entries, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

R_{WJMZJ} When a TLB conflict abort is generated, it is reported as one of the following:

- On an instruction fetch it is reported as an Instruction Abort.
- On a data access or a cache management instruction, it is reported as a Data Abort.
- For an `AT S1E0*` or `AT S1E1*` instruction executed from EL1, when a TLB conflict abort is generated on a stage 2 translation, it is reported as a Data Abort.

I_{DHRWD} For an `AT` instruction other than an `AT S1E0*` or `AT S1E1*` instruction executed from EL1, a TLB conflict does not generate an abort.

R_{QWTKL} For the EL1&0 regime, when a TLB conflict abort is generated and both stages of translation are enabled, then one of the following applies:

- If FEAT_BBML1 is implemented and the TLB conflict abort is generated due to changing the table or block size or Contiguous bit, then the abort is reported to EL2.
- Otherwise, it is IMPLEMENTATION DEFINED whether it is reported to EL1 or EL2.

R_{ZMVZT} A TLB Conflict abort cannot be generated for a disabled stage of translation.

I_{JYDXJ} When a TLB conflict abort is generated, all of the following are reported:

- A `0b110000` Fault status code.
- The lookup address that caused the fault.

D8.15.2 MMU faults generated by address translation instructions

R_{XGWVV} An address translation instruction uses the translation system and can generate all of the following faults:

- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- Permission fault.
- TLB conflict abort.
- When translating using the AArch32 translation systems, a domain fault.

I_{NPRRQ} If FEAT_RME is implemented, the following faults are added to the list of faults that can be generated by an address translation instruction:

- GPF.
- GPT address size fault.
- GPT walk fault.
- Synchronous External abort on GPT fetch.

These are collectively called Granule Protection Check faults (GPC faults). For more information, see GPC faults.

R_{NHWXL} When an address translation instruction generates a fault and when none of the following generated the fault, the fault is not taken as a synchronous exception, and PAR_EL1 is populated with PAR_EL1.F == 1 and the fault syndrome information:

- A synchronous External abort during a translation table walk.
- An `AT S1E0*` or `AT S1E1*` instruction executed from EL1 generates a stage 2 fault on a memory access during a translation table walk.

For more information, see Exceptions to reporting the fault in PAR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{ZCTWS} If an address translation instruction requires two address translation stages, then a fault can be generated from either stage 1 or stage 2.

I_{CLHMP} If an address translation instruction requires two address translation stages in the EL1&0 translation regime, then a fault can be generated on the stage 2 translation of an address accessed as part of the stage 1 translation table walk.

For more information, see Stage 2 fault on a stage 1 translation table walk.

I_{MXTJT} When populating PAR_EL1 with the result of an address translation instruction, granule protection checks are not performed on the final output address of a successful translation. However, granule protection checks are performed on fetches of stage 1 or stage 2 descriptors and these checks could result in a GPC fault.

I_{KKHBS} If FEAT_PAN2 is implemented, then the `<pan>` component of the AT instruction `<operation>` determines whether the PSTATE.PAN value is considered when determining permissions.

D8.15.2.1 Exceptions to reporting the fault in PAR_EL1

R_{QBSVM} For an address translation instruction executed at a particular Exception level, ELx, if a synchronous External abort is generated on a stage 1 translation table walk, then the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 1 translation table walk for a memory access from ELx would be taken.

R_{QCVQK} When an address translation instruction executed at EL3 generates a synchronous External abort on a stage 2 translation table walk, a Data Abort exception is taken to EL3.

R_{YWCRL} For an address translation instruction executed at the EL2 or EL1 Exception level, if a synchronous External abort is generated on a stage 2 translation table walk, then the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 2 translation table walk for a memory access from that same EL2 or EL1 Exception level would be taken.

R_{FDJKZ} When an address translation instruction causes a synchronous External abort that is reported as an exception, all of the following apply:

- PAR_EL1 is UNKNOWN.
- The ESR_ELx of the target Exception level indicates that the fault was generated by a translation table walk due to a cache maintenance operation or address translation instruction.
- The FAR_ELx of the target Exception level holds the VA supplied to the address translation instruction.

R_{ZWKYB} When an `AT S1E0*` or `AT S1E1*` instruction is executed, all of the following can generate a stage 2 fault on a memory access during a translation table walk:

- Stage 2 Translation fault.
- Stage 2 Address size fault.
- Synchronous External abort on a stage 2 translation table walk.
- Stage 2 Access fault.
- Stage 2 Permission fault.
- A TLB conflict abort that takes an IMPLEMENTATION DEFINED choice of reporting as a stage 2 abort.

R_{VMRFQ} When an `AT S1E0*` or `AT S1E1*` instruction executed from EL1 generates a stage 2 fault on a memory access during a translation table walk, one of the following occurs:

- If the fault is a synchronous External abort on a stage 2 translation table walk and SCR_EL3.EA is 1, then a synchronous External abort on a stage 2 translation table walk is taken to EL3.
- The fault is taken as an exception to EL2 and all of the following apply:
  - PAR_EL1 is UNKNOWN.
  - ESR_EL2 indicates that a cache maintenance instruction faulted during a translation table walk.
  - HPFAR_EL2 holds the IPA that faulted.
  - FAR_EL2 holds the VA supplied to the address translation instruction.

R_{ZTRDD} In addition to the cases listed in *Address translation instructions* in the Arm architecture, the following faults as a result of an address translation instruction are reported as an exception:

- GPC faults that would result in a GPC exception.
- GPC faults on fetches of stage 2 descriptors from `AT S1E0*` and `AT S1E1*` instructions executed from EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When HCR_EL2.GPF is 1, GPFs on fetches of stage 1 descriptors from `AT S1E0*` and `AT S1E1*` instructions executed from EL1.

Otherwise, faults as a result of an address translation instruction are reported using PAR_EL1.FST.

I_{VKPJB} For more information, see Stage 2 fault on a stage 1 translation table walk.

D8.15.3 MMU faults generated by cache maintenance operations

I_{QZTRG} There are no alignment requirements in cache maintenance instructions and execution of these instructions cannot generate an Alignment fault.

I_{BDCCY} Cache maintenance instructions treated as a `NOP` cannot generate MMU faults.

R_{TRTWX} It is IMPLEMENTATION DEFINED whether execution of any cache maintenance instruction by VA that is not treated as a `NOP` can generate an Access flag fault.

R_{MXZPX} If SCTLR_ELx.UCI is 1, enabling EL0 execution of the data cache maintenance instructions that operate by VA, then when the instruction is executed at EL0 to a location that does not have read permission at EL0, a Permission fault can be generated.

R_{XJRNW} If SCTLR_ELx.UCI is 1, enabling EL0 execution of the IC IVAU instruction, and if the instruction is executed at EL0 to a location that does not have read permission at EL0, then it is IMPLEMENTATION DEFINED whether a Permission fault is generated.

R_{FTXTG} If the Effective value of SCTLR_ELx.CMOW is 1, execution of any of the following at EL0 without stage 1 write permission generates a stage 1 Permission fault:

- Any DC instruction that operates by VA and performs clean and invalidate.
- Any IC instruction that operates by VA.

R_{BBLTJ} If the Effective value of HCRX_EL2.CMOW is 1, execution of any of the following at EL1 or EL0 without stage 2 write permission generates a stage 2 Permission fault:

- Any DC instruction that operates by VA and performs clean and invalidate.
- Any IC instruction that operates by VA.

I_{ZNSBH} When an implementation can generate MMU faults on execution of cache maintenance operations, Permission faults are generated according to the type of cache maintenance operation and system register configuration.

R_{JQCHQ} If a stage 2 translation of a stage 1 translation table walk is done as part of a cache maintenance instruction, then a stage 2 Permission fault can be generated.

R_{JYWZL} If a DC Invalidate by address instruction, for example, DC IVAC, DC IGVAC, or DC IGDVAC is not treated as a `NOP` and does not have write permission to the location it invalidates, then a Permission fault is generated.

I_{XQZJB} DC ZVA is not a cache maintenance instruction.

I_{LJLSP} For more information, see:

- Data and unified caches.
- Instruction caches.
- Memory aborts.
- Permission fault.
- A64 Cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.15.4 MMU fault-checking sequence

IPTYRT For a translation stage, when an IA is translated to an OA, the fault checking sequence is done using all of the following steps:

1. If the IA is subject to an alignment check, then check the alignment and do one of the following:
   - If the IA is not aligned, then an Alignment fault is generated.
   - If the IA is aligned, then continue to the next step.
2. Check that the IA maps to a translation table base address register, TTBR, and do one of the following:
   - If the IA does not map to a TTBR, then a Translation fault is generated.
   - If the IA maps to a TTBR and translation using that TTBR is disabled, then a Translation fault is generated.
   - If the IA maps to a TTBR, then get the translation table base address and continue to the next step.
3. Check that the translation table base address size is valid and do one of the following:
   - If the translation table base address size is not valid, then an Address size fault is generated.
   - If the translation table base address size is valid, then continue to the next step.
4. Fetch the descriptor, using all of the following steps:
   - If the descriptor address is an IPA from a stage 1 translation, then all of the following are done:
     - The fault checking sequence is done on the stage 2 translation, using the IPA as an IA to the stage 2 translation.
     - The OA from the stage 2 translation is the descriptor PA.
   - If the descriptor address is not an IPA from a stage 1 translation, then the descriptor address is the descriptor PA.
   - The descriptor fetch is initiated using the descriptor PA, and one of the following occurs:
     - A Synchronous External abort is generated.
     - The descriptor is returned.
5. Check that the descriptor is valid and do one of the following:
   - If the descriptor is not valid, then a Translation fault is generated.
   - If the descriptor is valid, then continue to the next step.
6. Check that the descriptor address size is valid and do one of the following:
   - If the descriptor address size is not valid, then an Address size fault is generated.
   - If the descriptor address size is valid, then continue to the next step.
7. Check the descriptor type and do one of the following:
   - For a Table descriptor, get the descriptor address, get the hierarchical permissions and attributes that apply to subsequent lookup levels, and go back to step 4 to fetch the next descriptor.
   - For a Block descriptor or Page descriptor, continue to the next step.
8. If FEAT_BBML1 is implemented, and the fetched descriptor is a block descriptor with the nT bit set, then the implementation can generate a Translation fault.
9. If AF hardware management is disabled or not implemented, check the descriptor AF bit and do one of the following:
   - If AF is 0, then an Access flag fault is generated.
   - If AF is 1, then continue to the next step.
10. If AF hardware management is enabled, then the hardware attempts to update the AF and that might result in a Permission fault at stage 2, or a Synchronous External abort.
11. Get the OA and OA space from the Block descriptor or Page descriptor returned by the translation table walk.
12. Check the alignment required for the output memory type and do one of the following:
    - If the OA alignment is not valid, then an Alignment fault is generated.
    - If the OA alignment is valid, then continue to the next step.
13. Check the OA space access permissions and do one of the following:
    - If the OA space access is not permitted, then a Permission fault is generated.
    - If the OA space access is permitted, then continue to the next step.
14. If dirty state hardware management is enabled, then the hardware attempts to update the dirty state and that might result in a Permission fault at stage 2, or a Synchronous External abort.
15. The translation stage returns the OA and region attributes.

IJFVPZ For a translation stage, the following figure illustrates the MMU fault checking sequence.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

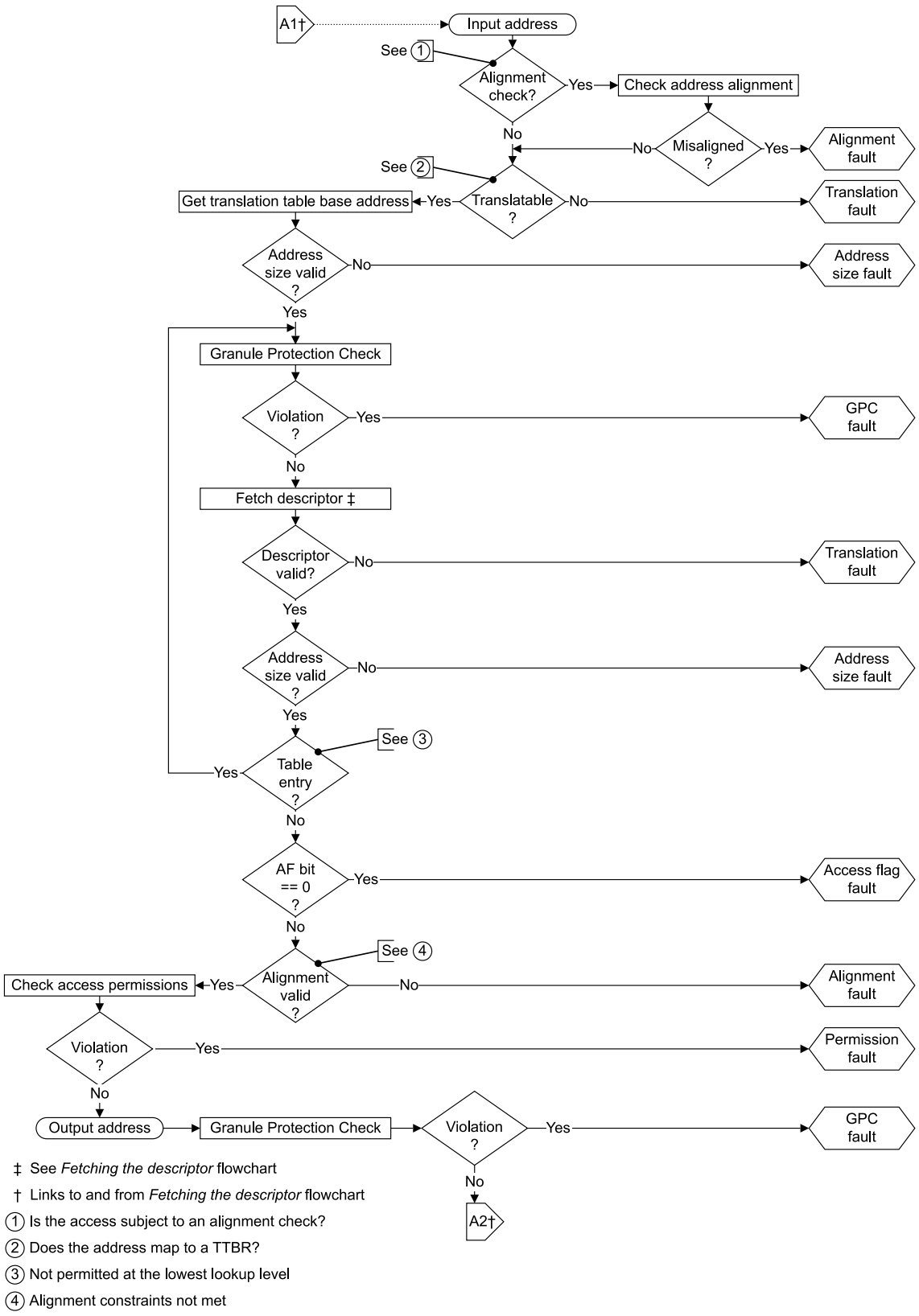

Figure D8-22 MMU fault checking sequence

IHZZMB For a translation stage, the following figure illustrates the steps taken to fetch a descriptor during a translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

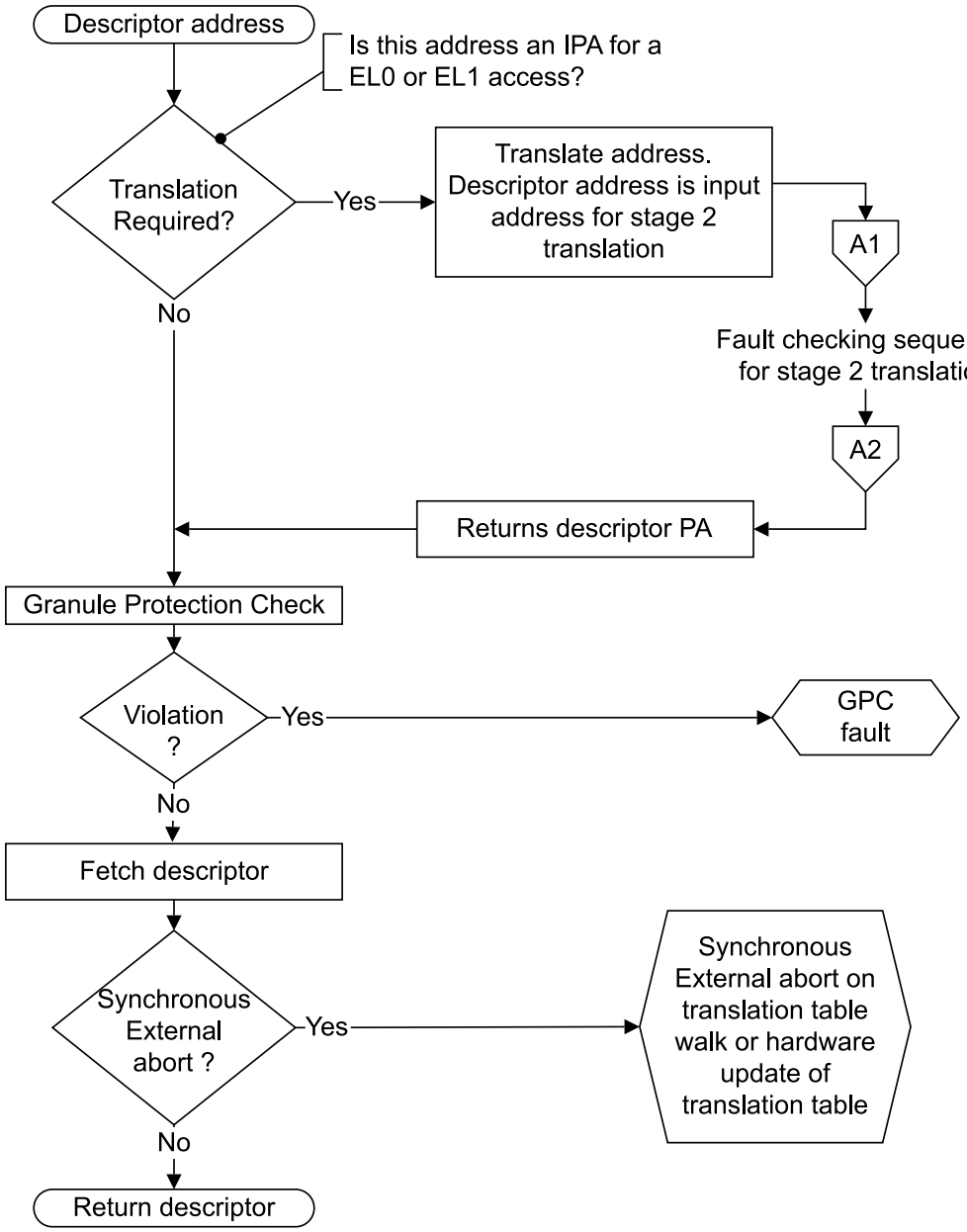

Figure D8-23 Fetching a descriptor during a translation table walk

I_{GLRRS} For more information, see Permission fault.

D8.15.4.1 Stage 2 fault on a stage 1 translation table walk

I_{MQXNK} For a translation regime that uses two translation stages, the memory access made as part of a stage 1 translation table lookup can generate one or more of the following during a stage 2 translation:

- A Translation fault, Address size fault, Access flag fault, or Permission fault.
- A GPC fault on the stage 2 descriptor access.
- A synchronous External abort on the stage 2 descriptor access.
- A synchronous External abort on the memory access.

I_{NGFYK} If an Address size fault, Translation fault, Access flag fault, or Permission fault is generated on a stage 2 translation of a stage 1 translation table walk, then all of the following apply:

- The exception is taken to EL2.
- ESR_EL2.ISS[7] is 1 to indicate a stage 2 fault during a stage 1 translation table walk.
- The part of the ISS field that might contain details of the instruction is invalid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{MRJLZ} If a synchronous External abort is generated on a stage 2 translation of a stage 1 translation table walk, then all of the following apply:

- If SCR_EL3.EA is 0, then all of the following apply:
  - The exception is taken to EL2.
  - ESR_EL2.ISS[7] is 1 to indicate a stage 2 fault during a stage 1 translation table walk.
- If SCR_EL3.EA is 1, then all of the following apply:
  - The exception is taken to EL3.
  - ESR_EL3.ISS[7] is 1 to indicate a stage 2 fault during a stage 1 translation table walk.
- The part of the ISS field that might contain details of the instruction is invalid.

I_{ZGDHT} If the stage 2 translation of a stage 1 translation table walk returns a Device memory type, then the value of HCR_EL2.PTW determines all of the following:

- If the value is 0, the translation table walk access is permitted and treated as an access to Normal, Non-cacheable memory.
- If the value is 1, the translation table walk access generates a stage 2 Permission fault.

I_{VXKVL} If the stage 2 translation of a stage 1 translation table lookup maps to Device memory, it likely indicates a Guest OS error where the stage 1 translation table is corrupted, and it is appropriate to trap this access to the hypervisor.

I_{LWGSL} If software updates HCR_EL2.PTW without changing the current VMID, then the software is required to invalidate the TLBs because they might hold entries that depend on the effect of HCR_EL2.PTW.

I_{HPGQN} For more information, see Permission fault.

D8.15.4.2 The lookup level associated with MMU faults

I_{XCCCM} For MMU faults, the *Data Fault Status Code* (DFSC) and *Instruction Fault Status Code* (IFSC) in the ESR_ELx report the translation lookup level associated with a given MMU fault type.

I_{CRKPZ} When an MMU fault is generated, all of the following determine the lookup level associated with the fault:

- For a fault generated by a translation table walk, the lookup level at which the fault occurred.
- For a Translation fault, one of the following:
  - The lookup level at which the fault occurred.
  - If a fault is generated because translation table walks are disabled by TCR_ELx.EPD*n*, the IA is outside the range specified by the associated TTBR_ELx, or FEAT_E0PD is enabled and prevents access to addresses translated by TTBRn_ELx, then the fault is reported as a level 0 fault.
- For an Access flag fault, the lookup level at which the fault occurred.
- For a Permission fault, including a Permission fault generated by hierarchical permissions, the final lookup level that returned the Block descriptor or Page descriptor.

I_{CJZVP} For more information, see Permission fault.

D8.15.5 MMU fault prioritization from a single address translation stage

R_{XCHFJ} For a single translation stage in a VMSAv8-64 translation regime, the following list shows the possible MMU faults on a memory access, from highest priority to lowest priority, and includes where stage 2 faults that are generated within a stage 1 translation table walk are prioritized:

1. For a stage 1 translation, an Alignment fault not caused by memory type.
2. If one or more of the following is true, then a Translation fault:
   - The IA out of the translated address range.
   - For a stage 1 translation, the IA requiring a TTBR_ELx that is disabled.
   - For a stage 2 translation, VTCR_EL2.SL0 is inconsistent with VTCR_EL2.T0SZ.
   - For a stage 2 translation, VSTCR_EL2.SL0 is inconsistent with VSTCR_EL2.T0SZ.
   - For a stage 2 translation, SL0 is programmed to a reserved value.
   - For a stage 2 translation, the Effective value of VTCR_EL2.DS is 1 and VTCR_EL2.{SL2, SL0} is inconsistent with VTCR_EL2.T0SZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a stage 2 translation, the Effective value of VTCR_EL2.DS is 1 and VSTCR_EL2.{SL2, SL0} is inconsistent with VSTCR_EL2.T0SZ.
- For a stage 2 translation, the Effective value of VTCR_EL2.DS is 1 and {SL2, SL0} is programmed to a reserved value.

3. If one or more of the following is true, an Address size fault on the address in TTBR_ELx:
   - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   - The programmed address is outside the implemented range.
4. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level -1 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
5. A synchronous parity or ECC error on a level -1 lookup of a translation table walk.
6. A GPC fault on a level -1 lookup of a translation table walk.
7. A synchronous External abort on a level -1 lookup of a translation table walk.
8. Translation fault on a level -1 translation table entry.
9. If one or more of the following is true, an Address size fault on the address in a level -1 lookup translation table entry:
   - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   - The OA is out of the implemented range.
10. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 0 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
11. A synchronous parity or ECC error on a level 0 lookup of a translation table walk.
12. A GPC fault on a level 0 lookup of a translation table walk.
13. A synchronous External abort on a level 0 lookup of a translation table walk.
14. Translation fault on a level 0 translation table entry.
15. If one or more of the following is true, an Address size fault on the address in a level 0 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.
16. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 1 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
17. A synchronous parity or ECC error on a level 1 lookup of a translation table walk.
18. A GPC fault on a level 1 lookup of a translation table walk.
19. A synchronous External abort on a level 1 lookup of a translation table walk.
20. Translation fault on a level 1 translation table entry.
21. If one or more of the following is true, an Address size fault on the address in a level 1 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.
22. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 2 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
23. A synchronous parity or ECC error on a level 2 lookup of a translation table walk.
24. A GPC fault on a level 2 lookup of a translation table walk.
25. A synchronous External abort on a level 2 lookup of a translation table walk.
26. Translation fault on a level 2 translation table entry.
27. If one or more of the following is true, an Address size fault on the address in a level 2 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.
28. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 3 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
29. A synchronous parity or ECC error on a level 3 lookup of a translation table walk.
30. A GPC fault on a level 3 lookup of a translation table walk.
31. A synchronous External abort on a level 3 lookup of a translation table walk.
32. Translation fault on a level 3 translation table entry.
33. If one or more of the following is true, an Address size fault on the address in a level 3 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

34. Access flag fault.
35. Alignment fault caused by the memory type.
    For an Alignment fault generated as a result of the conditions specified in Crossing a page boundary with different memory types or Shareability attributes or Crossing a peripheral boundary with a Device access, the prioritization of the Alignment fault is IMPLEMENTATION DEFINED, at a point between here and above Synchronous External abort on the memory access.
36. Permission fault. For more information, see Prioritization of Permission faults.
37. If stage 2 address translation is enabled, then a stage 2 fault on the memory access, including an Address size fault caused by a PA outside the implemented range.
38. Synchronous parity or ECC error on the memory access.
39. A GPC fault on the memory access.
40. Synchronous External abort on the memory access.

R_{MJLPH} If all the following apply, the prioritization between the stage 2 fault on the stage 1 translation table walk and the stage 1 fault caused by the stage 1 Block descriptor or Page descriptor is IMPLEMENTATION DEFINED:

- Hardware management of the stage 1 access flag or dirty state is enabled.
- The access flag or dirty state in a stage 1 descriptor needs to be updated.
- For the location of the stage 1 Block descriptor or Page descriptor, the stage 2 translation generates one of the following:
  - A Permission fault because the descriptor grants read permission but not write permission.
  - An unsupported atomic hardware update fault.
- The stage 1 Block descriptor or Page descriptor generates an MMU fault, which can be due to an Address size fault, an Alignment fault caused by memory type, or a Permission fault.

For more information, see Hardware management of the dirty state.

R_{WLLHL} When the use of a stage 1 Table descriptor generates a stage 1 Translation fault or Address Size fault, it is reported with higher priority than any stage 2 MMU faults generated as part of hardware updates for that descriptor, except in the following case:

- A synchronous External abort or an IMPLEMENTATION DEFINED abort caused by a memory type not supporting an atomic read-modify-write at stage 2 is permitted to be reported to EL2 with higher priority than a stage 1 Translation fault or Address Size fault.

R_{WLJCM} For a TLB conflict abort, all of the following apply:

- The priority of the abort is IMPLEMENTATION DEFINED.
- The priority of the abort is required to be higher than any abort that depends on a value held in a TLB.

I_{HGZXP} The cause of a TLB conflict abort depends on the TLB implementation.

I_{SYVYF} For more information on GPC fault priority, see Table D9-2.

R_{CSFKV} For an IMPLEMENTATION DEFINED MMU fault caused by an unsupported combination of memory access type and memory type and reported with ESR_ELx.DFSC value `0b110101`, the priority is IMPLEMENTATION DEFINED.

R_{KYXHT} For an MMU fault at a translation stage caused by an unsupported atomic hardware update, the priority of this fault lies at an IMPLEMENTATION DEFINED point between all of the following:

- A priority immediately higher than an Access flag fault generated by the same translation stage as this MMU fault.
- A priority immediately lower than a Permission fault generated by the same translation stage as this MMU fault.

D8.15.5.1 Prioritization of Permission faults

R_{DFGCF} The following list shows the priority order of stage 1 Permission faults, from highest priority to lowest priority:

1. Fault due to Overlay permissions.
2. Fault due to stage 1 Base permissions.
3. Fault due to write of an Allocation Tag to a Canonically Tagged memory location.
4. Fault due to the dirty state.

R_{TLHDK} The following list shows the priority order of stage 2 Permission faults, from highest priority to lowest priority:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. Fault due to TopLevel checks, causing a stage 2 abort during a stage 1 translation table walk.
2. Fault due to AssuredOnly check, including due to VTCR_EL2.GCSH, causing a stage 2 abort.
3. Fault due to stage 2 Overlay permissions.
4. Fault due to stage 2 Base permissions.
5. Fault due to NoTagAccess attribute, causing a stage 2 abort on the final access.
6. Fault due to the dirty state.

R_{LTQNW} For priority checking at each translation stage, when an access requires both read and write permission, both permissions are evaluated together at each step of the fault priority.

Note

Accesses that require both read and write permission include atomic, `RCW`, and `RCWS` instructions.
For example, if the Effective value of TCR_ELx.HD is 0 and the calculated privileged stage 1 permissions are as shown in the following table, an access will generate a stage 1 Permission fault with {Overlay, WnR, DirtyBit} fields in the syndrome register updated to {1, 1, 0}:

Table D8-121 Example condition that generate a stage 1 Permission fault

Permission type	PrivRead	PrivWrite
Stage 1 Overlay permissions	1	0
Stage 1 Base permissions	0	0
TTD.nDirty		Clean

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.16 Translation Lookaside Buffers

I_{ZVNKM} The Arm architecture does not specify any structure of *Translation Lookaside Buffers* (TLBs), and permits any structure that complies with the requirements described in this section.

R_{XCLRD} Translation table entries that generate a Translation fault, an Address size fault, or an Access flag fault are never cached in a TLB.

R_{SQBCS} When address translation is enabled, a translation table entry for an in-context translation regime that does not cause a Translation fault, an Address size fault, or an Access flag fault is permitted to be cached in a TLB or intermediate TLB caching structure as the result of an explicit or speculative access.

R_{ZVMQM} A translation table entry is not permitted to be cached in a TLB or intermediate caching structure if a synchronous external abort on the translation table walk prevents the PE from determining if the translation table entry would otherwise cause a Translation fault, an Address size fault, or an Access flag fault.

I_{RCGCH} When address translation is enabled, if a translation table entry meets all of the following requirements, then that translation table entry is permitted to be cached in a TLB or intermediate TLB caching structure at any time:

- The translation table entry itself does not generate a Translation fault, an Address size fault, or an Access flag fault.
- The translation table entry is not from a translation regime configured by an Exception level that is lower than the current Exception level.

I_{LBGNR} The Arm architecture permits TLBs to cache certain information from System control registers, including when any or all translation stages are disabled. The individual register descriptions specify System control register fields are permitted to be cached in a TLB.

For more information, see AArch64 System Register Descriptions.

R_{JBNTF} For any System register field that is described as being permitted to be cached in a TLB, if that field takes an Effective value as a result of a System register control field at a higher Exception level, then the Effective value is also permitted to be cached in a TLB.

R_{FXGNQ} The TLB entries in all of the following translation regimes are out-of-context:

- When executing at EL3 or EL2, the TLB entries associated with the EL1&0 translation regime are out-of-context.
- When executing at EL3, the TLB entries associated with the EL2 or EL2&0 translation regime are out-of-context.

I_{RLTGW} The VMSA provides TLB maintenance instructions for the management of TLB contents.

R_{NDCMB} When a translation stage is disabled and then re-enabled, TLB entries are not corrupted.

D8.16.1 TLB behavior at reset

I_{GWZWM} When a reset occurs, an implementation is not required to automatically invalidate a TLB.

R_{BQCDZ} When a reset occurs, a TLB is affected in all of the following ways:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- It is IMPLEMENTATION DEFINED whether a specific TLB invalidation routine is required to invalidate a TLB before translation is enabled after a reset.

I_{HXRSY} For the ELx reset is taken to, when a reset occurs, SCTLR_ELx.M is reset to 0. For the translation regime controlled by that SCTLR_ELx.M bit, when SCTLR_ELx.M is 0, TLB contents have no effect on address translation.

R_{QNFHZ} If an implementation requires a specific TLB invalidation routine, then all of the following apply:

- The routine is IMPLEMENTATION DEFINED.
- The implementation documentation is required to clearly document the routine.
- Arm recommends that the routine is based on the TLB maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{WVKXW} On a Cold reset or Warm reset, an implementation might require TLBs to maintain their contents from before the reset, including one or more of the following reasons:

- Power management.
- Debug requirements.

I_{WZHDM} For more information on the TLB maintenance instructions used in a TLB invalidation routine, see TLB maintenance instructions.

D8.16.2 TLB lockdown

R_{SSQZC} TLB lockdown support is IMPLEMENTATION DEFINED.

R_{XXDPS} If an implementation supports TLB lockdown, then all of the following apply:

- The lockdown mechanism is IMPLEMENTATION DEFINED.
- The implementation documentation is required to clearly document the interaction of the TLB lockdown mechanism with the architecture.
- A locked TLB entry is guaranteed to remain in the TLB, unless the locked TLB entry is affected by a TLB maintenance operation.
- An unlocked TLB entry is not guaranteed to remain in the TLB.
- If a translation table entry is modified, then it is not guaranteed that a locked TLB entry remains coherent with the modified translation table entry.

I_{DXXYV} If a translation table entry is modified, then it is not guaranteed that a locked TLB entry remains incoherent with the modified translation table entry because the lockdown mechanism might permit a TLB maintenance instruction to trigger an update of the locked TLB entry.
For more information, see The interaction of TLB lockdown with TLB maintenance instructions.

R_{DSQQW} The implementation is permitted to use the reserved IMPLEMENTATION DEFINED register encodings to implement TLB lockdown functions.

I_{LSKYV} TLB lockdown functions might include, but are not limited to, all of the following:

- Unlock all locked TLB entries.
- Prefetch a translation table entry into a specific TLB level.

R_{DSLVY} If an implementation supports TLB lockdown and EL2 is enabled, then when executing at EL1 or EL0, an exception due to TLB lockdown can be routed to one of the following:

- EL1, as a Data Abort exception.
- EL2, as a Hyp Trap exception.

D8.16.3 Use of ASIDs and VMIDs to reduce TLB maintenance requirements

I_{LHWHR} TLB maintenance can be reduced during context switches by associating the translation table lookups from some translation regimes with an *address space identifier* (ASID), *virtual machine identifier* (VMID), or both.

R_{DHRKH} The rules in this section apply only to translation regimes that support two privilege levels, and apply for each of the implemented Non-secure, Secure, and Realm Security states.

R_{WPFJJ} The rules in this section use the following definitions of *controlling ELx*:

- EL1 is the controlling ELx of stage 1 translations in the EL1&0 translation regime.
- EL2 is the controlling ELx of stage 1 translations in the EL2&0 translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.16.3.1 Global and process-specific translation table entries

I_{FRDGD} For translation regimes that use an ASID, it is possible to mark translations as global or non-global.

R_{XXNPZ} A TLB entry associated with a specific ASID value, or a specific ASID and a VMID value, can only be used to translate a VA in a context that is associated with the same ASID value, or ASID and VMID value.

R_{WTVTJ} For stage 1 of a translation regime that supports two privilege levels, non-global TLB entries are associated with a specific ASID.

I_{LCNWN} For stage 1 of a translation regime that can support two privilege levels, Arm expects that software configures translations specific to a process to be associated with a specific ASID.

R_{SHVGC} For all of the following, the translation does not support association of a translation with an ASID, and all translations are treated as global:

- A stage 1 translation that supports a single privilege level
- The stage 2 component of a two stage translation.

I_{HGKKC} The ASID permits software to switch between process-specific translation table mappings without removing previous mappings cached for another ASID from a TLB.

I_{LQMXG} For stage 1 of a translation regime that can support two privilege levels, each TTBR0_ELx and TTBR1_ELx has an ASID field.

R_{SCBSR} All the following values of TCR_ELx.A1 determine the ASID value that is used by the translations of both the lower VA range and upper VA range:

- If the value is 0, then the TTBR0_ELx.ASID is used.
- If the value is 1, then the TTBR1_ELx.ASID is used.

R_{PYWKY} For stage 1 of a translation regime that supports two privilege levels, TCR2_ELx.A2 determines all of the following:

- If the value is 0, then the one ASID selected by TCR_ELx.A1 is used.
- If the value is 1, then two ASIDs are used.

R_{XPSGZ} For a controlling ELx, if two concurrent ASIDs are used, then TCR_ELx.A1 does not select which ASID field is used for the purpose of caching translations in a TLB, and instead all of the following apply:

- Any non-global translation of an address in the TTBR0_ELx VA range is associated with TTBR0_ELx.ASID.
- Any non-global translation of an address in the TTBR1_ELx VA range is associated with TTBR1_ELx.ASID.

I_{NWLMM} The scope of `TLBI` operations is not affected by the Effective value of TCR2_ELx.A2.

This means that a `TLBI ASIDE1*` operation removes TLB entries associated with the specified ASID, regardless of which half of the VA range those entries would be used to translate, and would not be required to remove any other TLB entries.

I_{WBQGD} For execution at EL0, if FEAT_CSV2 is implemented, and either the EL1&0 or EL2&0 translation regime is used, the architecture defines a hardware-defined context as including an ASID. However, for execution at EL1 or at EL2 when using these translation regimes, the hardware-defined context does not include an ASID.

For more information see Restrictions on the effects of speculation.

R_{SCLCL} For a controlling ELx, if two concurrent ASIDs are used, then for the purposes of FEAT_CSV2, the ASID used in the hardware-defined context when executing at EL0 is one of the following:

- If TCR_ELx.A1 is 0, then TTBR0_ELx.ASID.
- If TCR_ELx.A1 is 1, then TTBR1_ELx.ASID.

R_{QGKGF} The nG bit in a Block descriptor and Page descriptor indicates one of the following:

- If the value is 0, the translation is global and the TLB entry applies to all ASID values.
- If the value is 1, the translation is non-global and the TLB entry applies to only the current ASID value.

R_{CHVGJ} For a translation regime that supports global and non-global translations, translation table entries from lookup levels other than the final lookup level are treated as non-global, regardless of the value of the nG bit in the final lookup level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{JYHZR} For a translation in Secure state, if the NSTable bit in a Table descriptor is 1 at any level of the translation table walk, then the resulting translation is treated as non-global, regardless of the nG bit value in the Block descriptor or Page descriptor for the translation.

R_{KDVGW} If TCR2_ELx.FNGy is 1, where y is {0, 1}, then all of the following apply:

- For translations in the TTBRy_ELx VA range, the Block and Page descriptor nG field is IGNORED.
- Address translations in the TTBRy_ELx VA range are cached in a TLB as non-global entries, regardless of the value of the Block or Page descriptor nG field.

I_{QVBDY} The effects of TCR2_ELx.FNGy apply regardless of the value of TCR2_ELx.A2.

R_{QNPQX} If EL2 is enabled, then for the EL1&0 translation regime, TLB entries created for the regime are associated with the current VMID.

I_{YWBQX} If EL2 is enabled, then for the EL1&0 translation regime, TLB entries are associated with the current VMID in all cases, and include all of the following:

- For both global and non-global translations of any ASID value, and regardless of whether stage 2 translation is disabled, TLB entries containing information from stage 1 translation.
- For both stage 1 translation table walks and stage 1 OA, and regardless of whether stage 1 translation is disabled, TLB entries containing information from stage 2 translations.

I_{CNTNV} For the EL1&0 translation regime, if EL2 is enabled, then TLB entries are associated with a VMID regardless of whether stage 2 translation is enabled.

I_{SLWFB} If EL2 is enabled, then for the EL1&0 translation regime, TLB entries associated with a VMID are associated with a specific virtual machine.

I_{HQXLL} The VMID permits software to switch between virtual machines that have different VMIDs without removing previous translation table mappings from a TLB.

I_{RWYWK} VTTBR_EL2.VMID holds the current VMID.

D8.16.3.2 ASID size

R_{KHXQX} The ASID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

I_{NHTBK} The maximum supported ASID size is indicated by ID_AA64MMFR0_EL1.ASIDBits.

I_{BZXYN} For an implementation that supports 16-bit ASIDs, all of the following values of TCR_ELx.AS specify whether TTBRn_ELx.ASID[15:8] are used:

- If TCR_ELx.AS is 0, then TTBRn_ELx.ASID[15:8] are not used and are treated in all of the following ways:
  - For every purpose other than direct reads of TTBR0_ELx.ASID and TTBR1_ELx.ASID, the bits are ignored by hardware.
  - When used for allocating and matching entries in a TLB, the bits are treated as if they are all zeros.
- If TCR_ELx.AS is 1, then TTBRn_ELx.ASID[15:8] are used for allocating and matching entries in a TLB.

I_{ZRWYC} Bits[15:8] of the ASID field in `TLBI{P}` instructions are considered by hardware regardless of the value of TCR_ELx.AS.

R_{NKTBP} For a translation using VMSAv8-32, if the implementation supports 16-bit ASIDs, then the 8-bit ASID used is zero-extended to 16 bits.

I_{DJKMH} The AArch32 ASID size is 8 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.16.3.3 VMID size

R_{RPSHN} The VMID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

I_{VLKQX} The maximum supported VMID size is specified by ID_AA64MMFR1_EL1.VMIDBits.

I_{ZDPJB} If and only if EL2 is using AArch64, then use of a 16-bit VMID is permitted.

I_{MDGYK} For an implementation that supports a 16-bit VMID, all of the following values of VTCR_EL2.VS specify whether VTTBR_EL2.VMID[15:8] are used:

- If VTCR_EL2.VS is 0, then VTTBR_EL2.VMID[15:8] are not used and are treated in all of the following ways:
  - For every purpose other than direct reads of VTTBR_EL2.VMID, the bits are ignored by hardware.
  - When used for allocating and matching entries in a TLB, the bits are treated as if they are all zeros.
- If VTCR_EL2.VS is 1, then VTTBR_EL2.VMID[15:8] are used.

D8.16.3.4 Common not private translations

I_{XJYLH} FEAT_TTCNP allows multiple PEs in the same Inner Shareable domain and operating in the same translation regime to use the same translation table entries in a given translation stage.

R_{FWQJZ} If all of the following conditions are met, translation table entries pointed to by TTBR_ELx are shared with all other PEs in the Inner Shareable domain, and are referred to as common not private translation tables:

- For the PEs that share the translation tables, the Effective value of the corresponding TTBR_ELx.CnP is 1.
- For the PEs that share the translation tables, the same translation regime in the same Security state applies to the corresponding TTBR_ELx.
- If an ASID applies to the corresponding translation stage, then all of the PEs that share non-global translation table entries are required to have the same current ASID.
- If a VMID applies to the corresponding translation stage, then all of the PEs that share translation table entries are required to have the same current VMID.

R_{LGDJT} For a common not private translation table, if a System register field with all of the following characteristics is set by a particular PE to a value that is different than the value set by the other PEs, then it is CONSTRAINED UNPREDICTABLE whether the System register field is interpreted using the value of the particular PE or the value of one of the other PEs that are sharing the translation table entry:

- The register field applies to the translation stage.
- The register field is permitted to be cached in a TLB.

R_{ZVRZW} For a translation regime with both stage 1 and stage 2 translations, a TLB entry can be shared between different PEs in one or more of the following cases only if the value of the TTBR_ELx.CnP bit is 1 for both stages of translation:

- The TLB entry holds information from stage 1 translation only.
- The TLB entry combines information from stage 1 and stage 2 translations.

I_{YVLVD} The TTBR_ELx.CnP bit is permitted to be cached in a TLB.

R_{QLGWZ} For a common not private translation table, if a TTBR_ELx does not point to the same translation table as the other TTBR_ELx registers, then the system is misconfigured and an address translation using that TTBR_ELx causes one of the following to occur:

- Multiple hits in a TLB, which is permitted to generate a TLB conflict abort.
- A CONSTRAINED UNPREDICTABLE result due to caching of control or data values.

For more information, see TLB conflict abort and CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.17 TLB maintenance

R_{RVJDB} For the purpose of TLB maintenance, a TLB entry is any structure that holds a translation table entry, including intermediate TLB caching structures and temporary working registers in translation table walk hardware.

I_{NDFND} When a translation table walk occurs, translation table entries that are permitted to be cached in a TLB might be held in TLB caching structures.

R_{NWYRD} Entries held in a TLB are distinguished by *translation context*. The *translation context* is all of the following context information:

- The Security state.
- The translation regime, with the exception that hardware is not required to distinguish between EL2 and EL2&0 TLB entries.
- If applicable to the translation regime, then the VMID.
- If applicable to the translation regime, then whether the translation is global or non-global.
- If the translation is non-global, then the ASID.

R_{FFWJK} TLBs, or TLB caching structures, are not guaranteed to remain coherent with changes to translation table entries, and are therefore distinct from data caches.

R_{ZYZYK} Indexing an intermediate TLB structure by the IA is permitted.

R_{BKKRB} If FEAT_nTLBPA is not implemented, then indexing an intermediate TLB structure by all of the following is also permitted:

- The PA of the location holding the translation table entry.
- For stage 1 translations, the IPA of the location holding the translation table entry.

R_{TJQVP} If FEAT_nTLBPA is not implemented, any TLB maintenance instruction that applies to a PE with the context information that is relevant to the translation table entry ensures cached copies of translation table entries are invalidated for that PE in all of the following implementations of intermediate TLB structures:

- Indexing the TLB using the PA of the location holding the translation table entry.
- For stage 1 translations, indexing the TLB using the IPA of the location holding the translation table entry.

I_{PFNFJ} FEAT_nTLBPA permits software to determine the existence of intermediate TLB caching structures that are indexed by PA or IPA and perform TLB maintenance accordingly when it would otherwise not be required.

R_{LGSCG} When a TLB maintenance instruction targets only stage 1 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include stage 1 information used to translate the address or range of addresses being invalidated.
- If the TLB maintenance targets a specific ASID, then entries in caching structures that are not tagged with the specific ASID are not invalidated.
- If the stage 1 translation information contained in a single Block or Page has been collectively cached as one or more smaller TLB entries, for example as the result of splintering, then all entries containing that stage 1 information are invalidated, regardless of whether the entry would be used to translate the address being invalidated.

R_{TVTYQ} When a TLB maintenance instruction applies only to stage 2 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include only stage 2 information used to translate the address or range of addresses being invalidated.
- If the stage 2 translation information contained in a single Block or Page has been collectively cached as one or more smaller TLB entries, for example as the result of splintering, then all entries containing that stage 2 information are invalidated, regardless of whether the entry would be used to translate the address being invalidated.
- The maintenance is not required to apply to structures combining stage 1 and stage 2 information used to translate the address or range of addresses being invalidated.

R_{GNJPZ} When a TLB maintenance instruction applies to both stage 1 and stage 2 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include stage 1 information.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The maintenance applies to any entries in caching structures that include stage 2 information.
- The maintenance applies to structures combining stage 1 and stage 2 information.

R_{PVVPF} TLB maintenance instructions that remove dirty state from stage 2 entries apply to entries that meet all of the following requirements:

- The entry would be used for stage 2 translation. This applies if the TLB entry holds information from stage 2 translation only, or combined information from stage 1 and stage 2 translation.
- The entry would be used with the current VMID.
- The entry would be used in an EL1&0 translation regime in the Security state selected by the Effective value of SCR_EL3.{NSE, NS}.

R_{SYCZJ} If the Effective value of SCR_EL3.{NSE, NS} is {0, 0} and the value of SCR_EL3.EEL2 is 0, then no TLB entries are affected by TLB maintenance instructions that only remove dirty state from stage 2 entries.

R_{NNZGV} When a TLB maintenance instruction applies only to remove dirty state from stage 2 entries, in a manner consistent with clearing the S2AP[1] bit, or the Dirty bit if Indirect permissions are used, all of the following apply:

- If VTCR_EL2.HD is 1 and the TLB entry was a cached copy of a writable-dirty descriptor, then a subsequent write access translated by that TLB entry causes hardware to update the stage 2 dirty state, including checking to ensure that the cached copy of the descriptor is not stale with regard to the descriptor in memory.
- If Direct permissions are used at stage 2, and the TLB entry was a cached copy of a descriptor that allowed write permission and had DBM = 0, then a subsequent write access translated by that TLB entry causes a check to ensure that the cached copy of the descriptor is not stale with regard to the descriptor in memory.
- If Indirect or Direct permissions are used at stage 2, but VTCR_EL2.HD is 0, then it is permitted for a subsequent write access translated by that TLB entry to generate a stage 2 Permission fault.
- In all cases, over-invalidation is permitted, and it is possible that a subsequent write access causes a new translation table walk.

These requirements apply for all write accesses, including both explicit writes and implicit writes, such as writes that are part of a hardware update of a stage 1 translation table entry.

R_{SWNLK} TLB maintenance operations that only remove dirty state from stage 2 entries apply regardless of the stage 2 Overlay permissions. Only the Base permissions in the TLB entries determine the effect of these TLBI operations.

R_{GPPYH} When a translation table entry associated with a specific VMID or ASID is modified, software is required to invalidate the corresponding TLB entry to ensure that the modified translation table entry is visible to subsequent execution, including speculative execution.

R_{VNRFW} When a System register field is modified and that field is permitted to be cached in a TLB, software is required to invalidate all TLB entries that might be affected by the field, at any address translation stage in the translation regime even if the translation stage is disabled, using the appropriate VMID and ASID, after any required System register synchronization.

For more information, see Synchronization requirements for AArch64 System registers.

I_{WZCBG} When a translation table entry that generates a Translation fault, Address size fault, or Access flag fault is changed to one that does not fault, all of the following apply to software:

- TLB invalidation is not required because an entry that generates one of the listed faults is never cached in a TLB.
- A Context synchronization event is required to ensure that the completed change to the translation table entry affects subsequent instruction fetches.

D8.17.1 Using break-before-make when updating translation table entries

R_{WHZWS} If multiple execution threads use the same translation tables, then when a translation table entry is modified in one or more of the following ways, the architecture requires software to use a break-before-make sequence:

- A change to the memory type, Shareability or Cacheability.
- The translation OA is changed and one or more of the following apply:
  - Either or both of the old and new translations grant write permission, including cases where the DBM bit is 1 and hardware updates of the dirty state is enabled.
  - The memory contents at the new OA do not match the memory contents at the previous OA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If software does not follow the sequence enabled by FEAT_BBML1 and FEAT_BBML2 is not implemented, all of the following changes to the block size used by the translation system:
  - Changing from a smaller size to a larger size, such as when a level 2 Table descriptor is replaced with a Block descriptor.
  - Changing from a larger size to a smaller size, such as when a level 2 Block descriptor is replaced with a Table descriptor.
- A global entry is created that might overlap with non-global entries in a TLB.

I_{WZRHR} A change to the translation OA space is considered a change to the translation OA.

I_{THWDH} Use of a break-before-make sequence to ensure that old and new translation table entries are never simultaneously visible to different execution threads is guaranteed to prevent all of the following problems:

- Creating multiple TLB entries that apply to the same address.
- The effects of TLB caching breaking coherency.
- The effects of TLB caching breaking single-copy atomicity properties.
- The effects of TLB caching breaking ordering guarantees or uniprocessor semantics.
- The effects of TLB caching causing a failure to clear the Exclusives monitors.

R_{DDMVT} A break-before-make sequence requires all of the following steps:

1. Replace the old translation table entry with an invalid entry.
2. Execute a DSB instruction to ensure the invalid entry is visible.
3. Invalidate TLB entries based on the translation table entry with a broadcast TLB invalidation instruction.
4. Execute a DSB instruction to ensure the invalidation completes.
5. Write the new translation table entry
6. Execute a DSB instruction to ensure the new entry is visible.

I_{GXGZY} For a translation stage with AF hardware management enabled, if a translation table entry is modified and the break-before-make sequence is not followed, then all of the following failures associated with AF hardware updates can occur:

- When a memory location associated with that translation table entry is accessed, the AF is not set.
- When hardware updates to that translation table entry are followed by stores appearing later in program order, the ordering required is not followed.

R_{RTBRL} For a translation stage with dirty state hardware management enabled, if a translation table entry is modified and the break-before-make sequence is not followed, then all of the following failures associated with dirty state hardware updates can occur:

- When a memory location associated with that translation table entry is not written, hardware modifies the AP[2] or S2AP[1] descriptor bit.
- When a memory location associated with that translation table entry is written, hardware does not modify the AP[2] or S2AP[1] descriptor bit.
- When hardware updates to that translation table entry are followed by stores appearing later in program order, the ordering required is not followed.

R_{FVQCK} If translation table entries are changed without appropriate TLB maintenance operations, including in the case where use of the break-before-make sequence is required but software does not follow the break-before-make sequence, it is possible that TLBs concurrently hold multiple different copies of those translation table entries.

In this situation, the following behaviors are permitted for a speculative or architectural access to the address resolved by those TLB entries:

- Use of the address matches multiple entries in a TLB, and a TLB conflict abort is detected. In this case, no access is made to memory based on those TLB entries. If the access is architectural, then the TLB conflict abort is reported as an exception.
- The resulting behavior is CONSTRAINED UNPREDICTABLE, and gives a behavior consistent with translation using one of the matching entries, or an amalgamation of more than one of the matching entries, but cannot permit access to memory regions with permissions or attributes that would not be possible in the current Security state at the current Exception level. This includes, for example:
  - Insufficient TLB maintenance for stage 1 translations by EL1 must not permit it to bypass the configuration of stage 2 translation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Insufficient TLB maintenance by Non-secure state must not permit it to access any memory in Secure PA space.

I_{HLHBH} For more information, see CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

D8.17.2 Support levels for changing table or block size

R_{PVTFW} All statements in this section apply only to changing table or block sizes without changing any other property requiring break-before-make as specified in the rule R_{WHZWS} in Using break-before-make when updating translation table entries.

R_{KFLJB} When a translation table entry is modified to change the table or block size, the hardware provides one of the following possible implementations affecting the break-before-make requirement to avoid breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors:

- If FEAT_BBML1 is not implemented, then software is required to use the break-before-make sequence.
- If FEAT_BBML1 is implemented, then software can use all of the following:
  - The break-before-make sequence.
  - The nT bit in the Table descriptor or Block descriptor.
- If FEAT_BBML2 is implemented, then software can change table or block size without breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors.

For more information, see Block translation entry.

R_{HHCXC} When using a Table descriptor or Block descriptor with the nT bit set to 1, any accesses translated do not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

I_{HYQMB} If the TLB entries are not invalidated after the writes that modified the translation table entries are completed, then a TLB conflict abort can be generated because in a TLB there might be multiple translation table entries that all translate the same IA.

For Table descriptors, this also applies to intermediate TLB caching structures.

For more information, see TLB conflict abort.

I_{GYJBJ} When an MMU fault is reported while changing table or block sizes but before completing the required TLBI to invalidate the old translation table entry, the syndrome information reported for the fault is consistent with one of the following:

- The old translation table structure.
- The new translation table structure.

R_{KHRBC} If FEAT_BBML1 is implemented, then changing the Contiguous bit in a set of Block descriptors or Page descriptors can be done without breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors.

R_{FCPSG} If FEAT_BBML1 is implemented and the Contiguous bit in a set of Block descriptors or Page descriptors is changed, then a TLB conflict abort can be generated because multiple translation table entries might exist within a TLB that translates the same IA.

I_{CFFVK} If FEAT_BBML1 is implemented and a TLB conflict abort is generated, then TLB maintenance is required to remove the multiple TLB entries that translate the same address.

For Table descriptors, this also applies to intermediate TLB caching structures.

For more information, see TLB maintenance due to TLB conflict.

D8.17.3 TLB maintenance due to TLB conflict

R_{BLRKC} If multiple TLB entries translate the same address, then the `ALL` or `VMALL` form of a TLB maintenance instruction that targets the given translation regime is guaranteed to remove all TLB entries within that regime.

I_{JCCRT} All of the following instructions are guaranteed to remove the TLB entries associated with a conflict:

- For the EL1&0 translation regime, when stage 2 translations are in use, either `VMALLS12E1` or `ALLE1`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For the EL1&0 translation regime, when stage 2 translations are not in use, either `VMALLE1` or `ALLE1`.
- For the EL2&0 translation regime, either `VMALLE1` or `ALLE2`.
- For the EL2 translation regime, `ALLE2`.
- For the EL3 translation regime, `ALLE3`.

R_{GRVDR} If multiple TLB entries translate the same address, then the minimum set of TLB maintenance operations required to guarantee all TLB entries associated with that address and translation regime have been invalidated is IMPLEMENTATION DEFINED.

D8.17.4 The interaction of TLB lockdown with TLB maintenance instructions

R_{VGPNS} If a TLB entry is locked and a TLB invalidate all instruction is executed and, if that entry was not locked the TLB invalidate all instruction would invalidate that entry, then one of the following IMPLEMENTATION DEFINED behaviors occur:

- The locked TLB entry is not affected.
- If the TLB entry is locked or might be locked, an IMPLEMENTATION DEFINED Data Abort exception is generated.

I_{FRJSL} The IMPLEMENTATION DEFINED Data Abort on a TLB invalidate all instruction permits TLB invalidation routines to choose to invalidate a large range of addresses, without considering whether any TLB entries are locked, or to not affect locked TLB entries.

R_{BMHZW} If a TLB entry is locked and a TLB invalidate by VA or invalidate by ASID instruction is executed and, if that entry was not locked the TLB invalidate by VA or invalidate by ASID instruction would invalidate that entry, then one of the following IMPLEMENTATION DEFINED behaviors occur:

- The locked TLB entry is invalidated.
- The locked TLB entry is not affected.
- If the TLB entry is locked or might be locked, an IMPLEMENTATION DEFINED Data Abort exception is generated.

I_{FXBBY} The exception syndrome definitions include a fault code for cache and TLB lockdown faults.

R_{BCLPC} If an implementation uses an abort mechanism when entries that can be locked down are not actually locked down, then all of the following are required:

- The IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down are documented.
- An IMPLEMENTATION DEFINED mechanism is documented that ensures no TLB entries are locked.
- For locked entries, one of the other specified alternatives is implemented.

I_{DRZHY} If an implementation supports TLB lockdown, then Arm recommends that IMPLEMENTATION DEFINED TLB maintenance instruction sequences use the architecturally-defined operations to minimize the number of customized operations required.

I_{PNDQN} For more information on TLB lockdown, see TLB lockdown.

D8.17.5 TLB maintenance instructions

I_{QJNCT} For a given translation regime, the architecture defines TLB maintenance instructions that provide all of the following functions:

- Invalidate all corresponding entries in a TLB.
- Invalidate a corresponding single TLB entry by ASID for a non-global entry.
- Invalidate all corresponding TLB entries that match a specific ASID.
- Invalidate all corresponding TLB entries that match a specific VA, regardless of the ASID.
- Invalidate all corresponding TLB entries within a range of addresses.
- Update all corresponding stage 2 TLB entries that match a specific VMID from writable-dirty to writable-clean.

I_{XMWNR} Each TLB maintenance instruction can be applied to one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Only the PE that executes the instruction
- All PEs in the same Shareability domain as the PE that executes the instruction.

I_{JFCRF} The A64 assembly language syntax of a TLB maintenance instruction is `TLBI{P} <operation>{, <Xt>}`.

I_{KTDKY} All of the following apply to the TLB maintenance instructions:

- The `TLBI` operations take zero or one register arguments, for cases where 64 bits or fewer of scope information is required.
- The `TLBIP` operations take a pair of register arguments, for cases where more than 64 bits of scope information is required.

I_{LHQBV} When VMSAv9-128 is in use, all of the following apply:

- For `TLBI RVA*` instructions, `BaseADDR[52:16]` is determined by input bits [36:0], and `BaseADDR[55:53]` is treated as having the same value as `BaseADDR[52]`.
- For `TLBI RIPA*` instructions, `BaseADDR[52:16]` is determined by input bits [36:0], and `BaseADDR[55:53]` is treated as 0.

I_{RJQTT} For the `TLBI` instructions, the invalidation scope applies to VMSAv8-64 and VMSAv9-128 TLB entries, except for all of the following:

- For `TLBI VA*` and `TLBI RVA*` instructions, when a translation table level hint is provided, the invalidation scope only applies to stage 1 VMSAv8-64 TLB entries.
- For `TLBI IPA*`, `TLBI RIPA*` instructions, when a translation table level hint is provided, the invalidation scope only applies to stage 2 VMSAv8-64 TLB entries.

I_{BDQZT} For the `TLBIP` instructions, the invalidation scope applies to VMSAv8-64 and VMSAv9-128 TLB entries, except for all of the following:

- For `TLBIP VA*` and `TLBIP RVA*` instructions, when a translation table level hint is provided, the invalidation scope only applies to stage 1 VMSAv9-128 entries.
- For `TLBIP IPA*`, `TLBIP RIPA*` instructions, when a translation table level hint is provided, the invalidation scope only applies to stage 2 VMSAv9-128 TLB entries.

I_{NDJSJ} The `<operation>` in `TLBI{P} <operation>{, <Xt>}` is one of the following:

- For `TLBI` instructions:
  - `ALLE1{NXS}`, `ALLE2{NXS}`, `ALLE3{NXS}`, `ALLE1IS{NXS}`, `ALLE2IS{NXS}`, `ALLE3IS{NXS}`, `ALLE1OS{NXS}`, `ALLE2OS{NXS}`, or `ALLE3OS{NXS}`.
  - `VMALLE1{NXS}`, `VMALLE1IS{NXS}`, `VMALLE1OS{NXS}`, `VMALLS12E1{NXS}`, `VMALLS12E1IS{NXS}`, `VMALLS12E1OS{NXS}`, `VMALLWS2E1{NXS}`, `VMALLWS2E1IS{NXS}`, or `VMALLWS2E1OS{NXS}`.
  - `ASIDE1{NXS}`, `ASIDE1IS{NXS}`, or `ASIDE1OS{NXS}`.
- For `TLBI` and `TLBIP` instructions:
  - `{R}VA{L}E1{NXS}`, `{R}VA{L}E2{NXS}`, `{R}VA{L}E3{NXS}`, `{R}VA{L}E1IS{NXS}`, `{R}VA{L}E2IS{NXS}`, `{R}VA{L}E3IS{NXS}`, `{R}VA{L}E1OS{NXS}`, `{R}VA{L}E2OS{NXS}`, or `{R}VA{L}E3OS{NXS}`.
  - `{R}VAA{L}E1{NXS}`, `{R}VAA{L}E1IS{NXS}`, or `{R}VAA{L}E1OS{NXS}`.
  - `{R}IPAS2{L}E1{NXS}`, `{R}IPAS2{L}E1IS{NXS}`, or `{R}IPAS2{L}E1OS{NXS}`.

I_{DMCXY} The `<operation>` in `TLBI{P} <operation>{, <Xt>}` has a structure of `{R}<type><regime><shareability>{NXS}` with all of the following components:

- `R` is optional and determines one of the following:
  - When present, the instruction applies to all TLB entries translating addresses within the address range.
  - When not present, the instruction applies to all TLB entries translating a single address that could be used by the PE that executes the TLB maintenance instruction.
- `<type>` is one of the following:
  - `ALL` specifies all translations at `<regime>`.
  - `VMALL` specifies all stage 1 translations for the affected translation regime, and if applicable, matching the current VMID.
  - `VMALLS12` specifies all stage 1 and stage 2 translations at EL1 with the current VMID.
  - `VMALLWS2` specifies all writable-dirty stage 2 translations at EL1 with the current VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - ASID specifies all non-global translations for the affected translation regime with the supplied ASID.
  - VA{L} specifies all translations at <regime> using the supplied address and, if an ASID is supplied, the supplied ASID or global entries.
  - VAA{L} specifies all global entries and non-global entries, regardless of the ASID value, for the affected translation regime using the supplied address.
  - IPAS2{L} specifies all stage 2 translations using the supplied IPA.
- For the VA{L}, VAA{L}, and IPAS2{L} types, L is an optional parameter specifying that the invalidation applies only to cached entries containing translation information returned by the final lookup level of the translation table walk.
- <regime> is one of the following Exception levels:
  - If E1, then EL1.
  - If E2, then EL2.
  - If E3, then EL3.
- <shareability> is one of the following:
  - IS indicates that the instruction applies to all TLBs in the Inner Shareable domain.
  - OS indicates that the instruction applies to all TLBs in the Outer Shareable domain.
  - The parameter is optional, and no value indicates that the instruction applies to all TLBs that could be used by the PE that executes the instruction.
- NXS has all of the following properties:
  - The parameter is optional.
  - When present, the instruction is considered complete after all issued memory transactions using translation information held in TLB entries that have the associated XS attribute set to 0 and are within the scope of the instruction, are complete.
  - When not present, the instruction is considered complete after all issued memory transactions using translation information held in TLB entries that are within the scope of the instruction are complete, regardless of the XS attribute value.
  - The parameter can only be present when FEAT_XS is implemented.

I_{YKBMJ} If <regime> is higher than the current Exception level, then the TLB maintenance instruction is UNDEFINED.

I_{DJQYS} All TLB maintenance instructions are UNDEFINED at EL0.

I_{XMYBL} Unless trapped by HCR_EL2.NV to EL2, then all of the following TLB maintenance instructions are UNDEFINED at EL1:

- TLBI ALLE1{NXS}, TLBI ALLE1IS{NXS}, and TLBI ALLE1OS{NXS}.
- TLBI{P} {R}IPAS2{L}E1{NXS}, TLBI{P} {R}IPAS2{L}E1IS{NXS}, and TLBI{P} {R}IPAS2{L}E1OS{NXS}.
- TLBI VMALLS12E1{NXS}, TLBI VMALLS12E1IS{NXS}, and TLBI VMALLS12E1OS{NXS}.
- TLBI VMALLWS2E1{NXS}, TLBI VMALLWS2E1IS{NXS}, and TLBI VMALLWS2E1OS{NXS}.

I_{SRWLL} If EL2 is not implemented, then all of the following TLB instructions are UNDEFINED:

- TLBI{P} {R}VA{L}E2{NXS}, TLBI{P} {R}VA{L}E2IS{NXS}, and TLBI{P} {R}VA{L}E2OS{NXS}.
- TLBI ALLE2{NXS}, TLBI ALLE2IS{NXS}, and TLBI ALLE2OS{NXS}.

I_{BWGVK} When a TLB entry is invalidated by one PE, it is inconsistent with the architecture to allow another PE to refill that TLB entry so that the new entry gives the appearance to software that the invalidation did not occur.

I_{BNSBV} The <Xt> in TLBI{P} <operation>{, <Xt>} has all of the following properties:

- It specifies a register that passes one or both of an address and an ASID as an argument.
- For operations that include a VA, if FEAT_TTL is implemented, then the register passes the level hint.
- For range-based TLB maintenance operations, then the register passes information about the address range.
- It is required by the TLBI ASID, TLBI{P} {R}VA{L}, TLBI{P} {R}VAA{L}, and TLBI{P} {R}IPAS2{L} instructions.

D8.17.5.1 TLB maintenance instructions that do not apply to a range of addresses

I_{ZVNJG} For TLB maintenance instructions that take a VA, an ASID, or both as an argument, and that do not apply to a range of addresses, the register specified by the Xt argument has the following format:

- Register bits[63:48] are one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If the instruction requires an ASID argument, the ASID.
  - If the instruction does not require an ASID argument, RES0.
- If FEAT_TTL is not implemented, then register bits[47:44] are RES0.
- If FEAT_TTL is implemented, then register bits[47:44] are one of the following:
  - If the instruction requires a VA argument, the translation table level hint, TTL.
  - If the instruction does not require a VA argument, RES0.
- Register bits[43:0] are one of the following:
  - If the instruction requires a VA argument, VA[55:12].
  - If the instruction does not require a VA argument, RES0.

R_{JGGKN} For TLB maintenance instructions that take a VA, hardware interprets VA[63:56] as each having the same value as VA[55].

R_{FGPTT} For TLB maintenance instructions that take a VA, if the instruction targets a translation regime that is using AArch32, then all of the following apply:

- Software is required to treat VA[55:32] as RES0.
- VA[55:32] are ignored when the instruction is executed, and interpreted as being zero.
- VA[63:56] are interpreted as being zero.

I_{LHKCP} If the implementation supports 16-bit ASIDs, then software is required to set the upper 8 bits of the ASID to 0 when the context being invalidated uses only 8 bits.

I_{CPNYZ} For TLB maintenance instructions that take a register argument that holds an IPA and that do not apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bit[63] is one of the following:
  - If the Effective value of SCR_EL3.{NSE, NS} is {0, 0}, the NS bit specifying the Secure or Non-secure IPA space.
  - If the Effective value of SCR_EL3.{NSE, NS} is {0, 1}, this field is RES0 and the instruction applies to the Non-secure IPA space.
  - If the Effective value of SCR_EL3.{NSE, NS} is {1, 1}, this field is RES0 and the instruction applies to the Realm IPA space.
  - If the Effective value of SCR_EL3.{NSE, NS} is {1, 0}, the instruction is not required to invalidate any TLB entries.
- If FEAT_TTL is not implemented, then register bits[62:40] are RES0.
- If FEAT_TTL is implemented, then all of the following apply:
  - Register bits[62:48] are RES0.
  - Register bits[47:44] are the translation table level hint, TTL.
  - Register bits[43:40] are RES0.
- Register bits[39:36] are one of the following:
  - If 52-bit addresses are supported, IPA[51:48].
  - If 52-bit addresses are not supported, RES0.
- Register bits[35:0] are IPA[47:12].

I_{HQTLK} For TLB maintenance instructions that take a register argument that holds a VA or an IPA, register bits that hold the address have all of the following properties:

- If the 4KB granule size is used, all address bits in the register bits are used by the invalidation instruction.
- If the 16KB granule size is used, register bits[1:0] are ignored because address bits[13:12] have no effect on the invalidation instruction.
- If the 64KB granule size is used, register bits[3:0] are ignored because address bits[15:12] have no effect on the invalidation instruction.

D8.17.5.2 TLB maintenance instructions that apply to a range of addresses

I_{QPHNP} For TLB maintenance instructions that take a VA, or a VA and an ASID, and that apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bits[63:48] are one of the following:
  - If the instruction requires an ASID argument, the ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If the instruction does not require an ASID argument, RES0.
- Register bits[47:46] are the translation granule size, TG.
- Register bits[45:44] are the SCALE field that is the exponent element of the calculation that produces the upper range.
- Register bits[43:39] are the NUM field that is the base element of the calculation that produces the upper range.
- If FEAT_TTL is not implemented, then register bits[38:37] are RES0.
- If FEAT_TTL is implemented, then register bits[38:37] are the translation table level hint, TTL.
- Register bits[36:0] give the starting address of the range of addresses, BaseADDR.

I_{XPNPP} For TLB maintenance instructions that take a register argument that holds an IPA and that apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bit[63] is one of the following:
  - If the Effective value of SCR_EL3.{NSE, NS} is {0, 0}, the NS bit specifying the Secure or Non-secure IPA space.
  - If the Effective value of SCR_EL3.{NSE, NS} is {0, 1}, this field is RES0 and the instruction applies to the Non-secure IPA space.
  - If the Effective value of SCR_EL3.{NSE, NS} is {1, 1}, this field is RES0 and the instruction applies to the Realm IPA space.
  - If the Effective value of SCR_EL3.{NSE, NS} is {1, 0}, the instruction is not required to invalidate any TLB entries.
- Register bits[62:48] are RES0.
- Register bits[47:46] are the translation granule size, TG.
- Register bits[45:44] are the SCALE field that is the exponent element of the calculation that produces the upper range.
- Register bits[43:39] are the NUM field that is the base element of the calculation that produces the upper range.
- If FEAT_TTL is not implemented, then register bits[38:37] are RES0.
- If FEAT_TTL is implemented, then register bits[38:37] are the translation table level hint, TTL.
- Register bits[36:0] specify the starting address of the range of addresses, BaseADDR.

R_{PVVXP} For TLB instructions that apply to a range of addresses, the following table shows the TG field encodings that define the translation granule size for the translations that are being invalidated, and the Translation_Granule_Size in bytes used in determining the address range.

Table D8-122 TG field encodings in TLB instructions that apply to a range of addresses

TG	Translation granule size	Translation_Granule_Size
00	Reserved	N/A
01	4KB translation granule	4096
10	16KB translation granule	16384
11	64KB translation granule	65536

R_{XKMMX} For TLB instructions that apply to a range of addresses, if the translations use a different translation granule size than the one specified by the TG field, then the architecture does not require that the instruction invalidate those entries.

R_{QKRKL} For all TLB range maintenance instructions, TLB entries that translate one or more addresses within the address range determined by the following formula are invalidated:

```
[BaseADDR <= input_address < BaseADDR + ((NUM +1)*2^(5*SCALE +1) *
Translation_Granule_Size)].
```

R_{QNPXY} For a translation regime that supports two VA ranges, if a TLB range maintenance instruction is issued with an address in the TTBR1_ELx half of the VA space, and the SCALE and NUM values cause the range to exceed the top of the address space, then the address is not considered to wrap on overflow and the PE is not required to invalidate any entries inserted for the corresponding TTBR0_ELx half of the VA space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{HKYNQ} For a TLBI range maintenance instruction, `BaseADDR` is derived from bits[36:0] in the register specified by the `Xt` argument as one of the following:

- For the 4KB granule size, one of the following:
  - If the Effective value of TCR_EL1.DS is 0, then `BaseADDR[48:12]`.
  - If the Effective value of TCR_EL1.DS is 1, then `BaseADDR[52:16]`. In this case, `BaseADDR[15:12]` is treated as all zero.
- For the 16KB granule size, one of the following:
  - If the Effective value of TCR_EL1.DS is 0, then `BaseADDR[50:14]`.
  - If the Effective value of TCR_EL1.DS is 1, then `BaseADDR[52:16]`. In this case, `BaseADDR[15:14]` is treated as all zero.
- For the 64KB granule size, `BaseADDR[52:16]`.

I_{TYBWT} For all of the following, the invalidated address range is UNPREDICTABLE:

- For the 4KB translation granule, one of the following:
  - The Effective value of TCR_EL1.DS is 1, the TTL field is `0b00`, and `BaseADDR[38:12]` are not all zero.
  - The TTL field is `0b01` and `BaseADDR[29:12]` are not all zero.
  - The TTL field is `0b10` and `BaseADDR[20:12]` are not all zero.
- For the 16KB translation granule, one of the following:
  - The Effective value of TCR_EL1.DS is 1, the TTL field is `0b01`, and `BaseADDR[35:14]` are not all zero.
  - The TTL field is `0b10` and BaseADDR[24:14] are not all zero.
- For the 64KB translation granule, one of the following:
  - The TTL field is `0b01` and `BaseADDR[41:16]` are not all zero.
  - The TTL field is `0b10` and `BaseADDR[28:16]` are not all zero.

I_{ZNXDY} For more information, see Broadcast TLB maintenance.

D8.17.5.3 Translation table level hint

R_{BLDQZ} All statements in this section require implementation of FEAT_TTL.

I_{LTZBK} For the address being invalidated by a TLB maintenance instruction, the TTL field in the register specified by `Xt` indicates the lookup level of the translation table walk holding the leaf entry.

I_{WVDLN} Hardware can use the TTL field to determine if the page might have been splintered into multiple TLB entries.

R_{SQXYZ} For an entry being invalidated by a TLB maintenance instruction, if an incorrect TTL value is specified, then the architecture does not require any entries to be invalidated from a TLB.

I_{VWPQL} For TLB instructions that apply to a single address, the following table shows the 4-bit TTL field encodings.

Table D8-123 TTL field encodings in TLB instructions that apply to a single address

TTL[3:2]	Translation granule	TTL[1:0]	Information supplied
00	-	RES0	No information supplied about the translation level. Hardware assumes that the leaf entry can be from any level.
01	4KB	00	If FEAT_LPA2 is not implemented, then this value is reserved, and hardware treats this as if TTL[3:2] is `0b00`. If FEAT_LPA2 is implemented, then the leaf entry for the address being invalidated is on level 0 of the translation table walk.
		01	The leaf entry for the address being invalidated is on level 1 of the translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL[3:2]	Translation granule	TTL[1:0]	Information supplied
		10	The leaf entry for the address being invalidated is on level 2 of the translation table walk.
		11	The leaf entry for the address being invalidated is on level 3 of the translation table walk.
10	16KB	00	This value is reserved, and hardware treats this as if TTL[3:2] is `0b00`.
		01	If FEAT_LPA2 is not implemented, then this value is reserved, and hardware treats this as if TTL[3:2] is `0b00`. If FEAT_LPA2 is implemented, then the leaf entry for the address being invalidated is on level 1 of the translation table walk.
		10	The leaf entry for the address being invalidated is on level 2 of the translation table walk.
		11	The leaf entry for the address being invalidated is on level 3 of the translation table walk.
11	64KB	00	This value is reserved, and hardware treats this as if TTL[3:2] is `0b00`.
		01	The leaf entry for the address being invalidated is on level 1 of the translation table walk.
		10	The leaf entry for the address being invalidated is on level 2 of the translation table walk.
		11	The leaf entry for the address being invalidated is on level 3 of the translation table walk.

I_{VNDBJ} For TLB instructions that apply to a range of addresses, the following table shows the 2-bit TTL field encodings.

Table D8-124 TTL field encodings in TLB instructions that apply to a range of addresses

TTL	Information supplied
00	The leaf entries in the range can be using any level for the translation table entries.
01	When the 4KB or 64KB translation granule is used, all leaf entries to invalidate are level 1 translation table entries. If FEAT_LPA2 is not implemented, then when using a 16KB translation granule, this value is reserved, and hardware treats this as if TTL is `0b00`. If FEAT_LPA2 is implemented, then when using a 16KB translation granule, all leaf entries to invalidate are level 1 translation table entries.
10	All leaf entries to invalidate are level 2 translation table entries.
11	All leaf entries to invalidate are level 3 translation table entries.

D8.17.5.4 TLB maintenance instruction scope

R_{WHKKS} If `<type>` is `ALL` in a TLB maintenance instruction, then all of the following apply:

- For all values of `<regime>`, the instruction exists.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 and stage 2 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies to cached copies of translation table entries with any VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{PMNQC} If `<type>` is `VMALL` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to all of the following:
  - When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime.
  - When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

R_{BMMCC} If `<type>` is `VMALLS12` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 and stage 2 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- The instruction applies only to cached copies of translation table entries with the current VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.
- If one of the following is true and the instruction is executed at EL3, the instruction is not UNDEFINED and has the same effect as `TLBI VMALL` because there are no stage 2 translations to invalidate:
  - EL2 is not implemented.
  - The Effective value of SCR_EL3.NS is 0 and EL2 is disabled.

R_{JQQFG} If `<type>` is `VMALLWS2` in a TLB maintenance instruction, then all of the following apply:

- The value of `<regime>` is always E1, and the instruction applies to the EL1&0 translation regime.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all writable-dirty cached copies of the stage 2 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- The instruction applies only to cached copies of translation table entries with the current VMID.
- Entries affected by the instruction are updated to writable-clean but are not required to be invalidated.

R_{FKSVM} If `<type>` is `ASID` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to all of the following:
  - When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime when executing at EL1.
  - When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime when executing at EL2.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.
- The instruction applies only to non-global entries from the final lookup level that matches the specified ASID.

R_{VMSWQ} If `<type>` is `VA` in a TLB maintenance instruction, then all of the following apply:

- For all values of `<regime>`, the instruction exists.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate the address at the specified Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.
- For instructions where the value of `<regime>` is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the Effective value of HCR_EL2.E2H.
- For entries from a translation regime in which an ASID is valid, the instruction applies only to all of the following cached copies of translation table entries:
  - A non-global entry from any lookup level that matches the specified ASID.
  - A global entry from the final lookup level.

R_{DRCZZ} If `<type>` is `VAL` in a TLB maintenance instruction, then all of the following apply:

- For all values of `<regime>`, the instruction exists.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from the final lookup level in the translation table walk required to translate the address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries from the final lookup level with the current VMID.
- For instructions where the value of `<regime>` is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the Effective value of HCR_EL2.E2H.
- For entries from a translation regime in which an ASID is valid, the instruction applies only to all of the following cached copies of translation table entries:
  - A non-global entry from the final lookup level that matches the specified ASID.
  - A global entry from the final lookup level.

R_{ZSXRQ} If `<type>` is `VAA` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to all of the following:
  - When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime when executing at EL1.
  - When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime when executing at EL2.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate the address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.
- For instructions where the value of `<regime>` is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the Effective value of HCR_EL2.E2H.
- The instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

R_{JXKNN} If `<type>` is `VAAL` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to all of the following:
  - When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime when executing at EL1.
  - When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime when executing at EL2.
- For the Security state specified by the Effective value of SCR_EL3.{NSE, NS} and for the value of SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from the final lookup level in the translation table walk required to translate the address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries from the final lookup level with the current VMID.
- For instructions where the value of `<regime>` is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the Effective value of HCR_EL2.E2H.
- The instruction applies to all of the following cached copies of translation table entries from the final lookup level in the translation table walk:
  - Global entries.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Non-global entries with any ASID.

R_{VZZVZ} If `<type>` is `IPAS2` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, and if EL2 is enabled, then the instruction applies only to the EL1&0 translation regime.
- If all of the following apply, the instruction applies to all cached copies of the stage 2 translation table entries from any lookup level in the translation table walk required to translate the address:
  - The TLB entry holds only a stage 2 translation table entry.
  - The entry applies only to the current VMID.
- The instruction is not required to invalidate TLB entries that combine stage 1 and stage 2 translation table entries.
- If one of the following is true, then the instruction executes as a NOP:
  - EL2 is not implemented.
  - The Effective value of SCR_EL3.NS is 0 and EL2 is disabled.

R_{NSDNR} If `<type>` is `IPAS2L` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, and if EL2 is enabled, then the instruction applies only to the EL1&0 translation regime.
- If all of the following apply, the instruction applies to all cached copies of the stage 2 translation table entries from the final lookup level in the translation table walk required to translate the address:
  - The TLB entry holds only a stage 2 translation table entry.
  - The entry applies only to the current VMID.
- The instruction is not required to invalidate TLB entries that combine stage 1 and stage 2 translation table entries.
- If one or more of the following is true, then the instruction executes as a NOP:
  - EL2 is not implemented.
  - The Effective value of SCR_EL3.NS is 0 and EL2 is disabled.

R_{KBSSD} It is IMPLEMENTATION SPECIFIC whether a TLBI maintenance operation with the nXS qualifier invalidates TLB entries with the XS attribute set to 1.

D8.17.6 Operation of the TLB maintenance instructions

R_{XQBCL} A TLB maintenance instruction applies whether a translation stage is enabled or disabled.

R_{KKDKL} A TLB maintenance instruction can affect any TLB entries that are not locked down.

I_{YGYRY} There is no guarantee that an unlocked TLB entry remains in the cache. Therefore, it is not possible to determine architecturally whether a TLB maintenance instruction has affected any TLB entries that were not specified by the instruction.

I_{LMGTM} The TLB maintenance instructions specify the Exception level of the translation regime to which they apply.

R_{TBLDD} A TLB maintenance instruction never generates an MMU abort.

I_{BDTLG} If EL3 is implemented, then the Effective value of SCR_EL3.{NSE, NS} modifies the effect of the TLB maintenance instructions in all of the following ways:

- For instructions that apply to the EL2 or EL2&0 translation regime, if SCR_EL3.{EEL2, NS} is not {0, 0}, then all of the following apply:
  - If the Effective value of SCR_EL3.{NSE, NS} is {0, 0}, then the instruction applies to the Secure variant of the corresponding regime.
  - If the Effective value of SCR_EL3.{NSE, NS} is {0, 1}, then the instruction applies to the Non-secure variant of the corresponding regime.
  - If the Effective value of SCR_EL3.{NSE, NS} is {1, 1}, then the instruction applies to the Realm variant of the corresponding regime.
  - If the Effective value of SCR_EL3.{NSE, NS} is {1, 0}, then the instruction is not required to invalidate any TLB entries.
- If SCR_EL3.{EEL2, NS} is {0, 0}, then instructions that apply to the EL2 or EL2&0 translation regime are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For instructions that apply to the EL3 translation regime, the Effective value of SCR_EL3.{NSE, NS} has no effect.

I_{GDZCC} A TLB maintenance instruction is not affected by the current state of all of the following control bits involved in the translation process:

- In AArch64 state, all of the following:
  - SCTLR_EL1.M.
  - SCTLR_EL2.M.
  - SCTLR_EL3.{M, RW}.
  - HCR_EL2.{VM, RW}.
  - TCR_EL1.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}.
  - If the Effective value of HCR_EL2.E2H is 0, then TCR_EL2.{TG0, T0SZ}.
  - If the Effective value of HCR_EL2.E2H is 1, then TCR_EL2.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}.
  - TCR_EL3.{TG0, T0SZ}.
  - VTCR_EL2.{SL0, T0SZ}.
  - TTBR0_EL1.ASID.
  - TTBR1_EL1.ASID.
  - If the Effective value of HCR_EL2.E2H is 1, then all of the following:
    - TTBR0_EL2.ASID.
    - TTBR1_EL2.ASID.
- In AArch32 state, all of the following:
  - SCTLR.M.
  - HCR.VM.
  - TTBCR.{EAE, PD1, PD0, N, EPD1, T1SZ, EPD0, T0SZ, A1}.
  - HTCR.T0SZ.
  - VTCR.{SL0, T0SZ}.
  - TTBR0.ASID.
  - TTBR1.ASID.
  - CONTEXTIDR.ASID.

R_{FZTHY} Address-based TLB maintenance instructions are applied to the extent of the `<shareability>` specified by the instruction regardless of the Shareability assigned by any translation of the IA or Shareability attribute stored in TLB entries.

R_{MLYGB} For TLB maintenance instructions that operate only on instruction TLBs or only on data TLBs, Arm deprecates their use in AArch32 state.

R_{LQLSD} When a TLB maintenance instruction is executed, all of the following apply:

- The minimum size of a TLB entry that is required to be invalidated from the TLB is at least the size specified by the corresponding translation table entry.
- The Contiguous bit does not affect the minimum size of a TLB entry that is required to be invalidated from the TLB.

I_{JQPNP} When invalidating a mapping greater than a translation granule size or block size, for example a mapping that is specified using the Contiguous bit, software is required to issue a TLB maintenance instruction for each block or granule within the configured contiguous region size to invalidate the entire mapping.

R_{MMZTY} An implementation is permitted to invalidate more than the minimum required TLB entries.

I_{ZWTPG} Arm recommends not invalidating entries that are not required to be invalidated to minimize the performance impact.

I_{RTRQB} Arm expects software to mostly use address-based TLB invalidation instructions that apply to entries cached from the last level of a stage 1 or a stage 2 translation table walk to avoid unnecessary loss of the cached intermediate translation table entries.

R_{TLJCY} For all of the following reasons, software is required to set VTTBR_EL2.VMID[7:0] to a known value as part of the PE initialization sequence:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The VTTBR_EL2.VMID field resets to a value that is architecturally UNKNOWN.
- If EL2 is enabled, then dependencies on the VMID in the EL1&0 translation regime apply whether stage 2 translation is enabled or disabled.

D8.17.6.1 Invalidating TLB entries from stage 2 translations

I_{LRXYX} The following code is required to invalidate all cached copies of the stage 2 translation of the IPA held in `Xt` using the current VMID, with the corresponding requirement applied to the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VMALLE1
```

I_{CKZGP} The following code is required to invalidate all cached copies of the stage 2 translations of the IPA held in `Xt` used to translate the VA, and the specified ASID when executing `TLBI VAE1`, held in `Xt2`, with the corresponding requirement applied to the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VAE1, Xt2 ; or TLBI VAAE1, Xt2
```

I_{QKHVL} The following code is required to invalidate all cached copies of the stage 2 translations of the IPA held in `Xt` used to translate the IPA produced by the last level of stage 1 translation table lookup for the VA, and the specified ASID when executing `TLBI VALE1`, held in `Xt2`, with the corresponding requirement applied to the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VALE1, Xt2 ; or TLBI VAALE1, Xt2
```

I_{XPBXT} For an EL1&0 translation regime with stage 2 translation enabled, software is required to use the entire invalidation sequences, even if stage 1 translation is disabled.

I_{RGYNM} Equivalent architectural requirements apply to the `IPAS2L` instruction, except that the only TLB entries required to be invalidated by an `IPAS2L` instruction are those that come from the final translation table lookup level.

D8.17.7 Broadcast TLB maintenance

R_{THTWM} If all of the following apply, then a TLB maintenance instruction executed by a PE in an Exception level that is using AArch64 can affect a PE that executes the same Exception level using AArch32:

- The PE lies within the target Shareability domain of the TLB maintenance instruction.
- If VA matching is required, then the VA is `0x0000_FFFF_FFFF` or lower in the memory map.
- If ASID matching is required, then one of the following:
  - — If the PE using AArch64 is using an 8-bit ASID, then the ASIDs match.
  - — If the PE using AArch64 is using a 16-bit ASID and the top 8 bits of the 16-bit ASID are zero, then the ASIDs match.
- If VMID matching is required, then one of the following:
  - — If the PE using AArch64 is using an 8-bit VMID, then the VMIDs match.
  - — If the PE using AArch64 is using a 16-bit VMID and the top 8 bits of the 16-bit VMID are zero, then the VMIDs match.

R_{VYLVH} If all of the following apply, then a TLB maintenance instruction executed by a PE in an Exception level that is using AArch32 can affect a PE that executes the same Exception level using AArch64:

- The PE lies within the same Inner Shareable Shareability domain of the TLB maintenance instruction.
- If VA matching is required, then when the supplied VA is zero-extended, the VAs match.
- If IPA matching is required, then when the supplied IPA is zero-extended, the IPAs match.
- If ASID matching is required, then one of the following:
  - — If the PE using AArch64 is using an 8-bit ASID, then the ASIDs match.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- — If the PE using AArch64 is using a 16-bit ASID, then when the supplied ASID is zero-extended, then the ASIDs match.
- If VMID matching is required, then one of the following:
  - — If the PE using AArch64 is using an 8-bit VMID, then the VMIDs match.
  - — If the PE using AArch64 is using a 16-bit VMID, then when the supplied VMID is zero-extended, the VMIDs match.

R_{WMNBC} If a PE with EL3 using AArch32 issues a broadcast AArch32 TLB maintenance instruction affecting entries in Secure state, then the instruction is not required to affect one or more of the following in the Inner Shareable domain:

- The EL3 translation regime of the PEs with EL3 using AArch64.
- If EL2 is disabled, then the EL1&0 translation regime of the PEs with EL3 using AArch64, regardless of whether the EL1&0 translation regime is using AArch64 or AArch32.

R_{SLJFD} If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting EL3 entries, then the instruction is not required to affect the EL3 translation regime of the PEs with EL3 using AArch32 in the Inner Shareable domain.

R_{GKQML} If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting Secure EL1 entries, then the instruction is not required to affect the EL3 translation regime of the PEs with EL3 using AArch32 in the Inner Shareable domain.

R_{XMZFH} In all of the following cases, a broadcast TLB maintenance instruction is not required to perform any invalidation on the recipient PE within the target Shareability domain:

- A TLB maintenance instruction specifying a VA and affecting one of the following translation regimes is broadcast from a PE using one translation granule size for that translation regime to a PE using a different translation granule size for that VA in the same translation regime:
  - — The EL2 translation regime.
  - — The EL2&0 translation regime.
  - — The EL3 translation regime.
- A TLB maintenance instruction specifying a VA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 1 translation granule size for that translation regime to a PE using EL1 and a different stage 1 translation granule size for that VA.
- If EL2 is enabled, then a TLB maintenance instruction specifying a VA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 2 translation granule size to a PE using a different stage 2 translation granule size.
- If EL2 is enabled, then a TLB maintenance instruction specifying an IPA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 2 translation granule size to a PE using a different stage 2 translation granule size.
- A TLB maintenance instruction specifying a range of VAs or IPAs is broadcast from one PE to a PE that does not support the TLB range-based operations.

R_{YNPVV} The set of Requesters containing TLBs that support TLB range-based operations is defined by the system architecture.

I_{WHFDZ} Arm strongly recommends that within an Inner Shareable domain, all PEs support the same set of broadcast TLB range maintenance instructions.

R_{TVMHK} If an Armv7 PE is in the same Inner Shareable domain as an Armv8 PE, and the Armv8 PE issues a broadcast TLB maintenance instruction that is not defined in Armv7, then that instruction is not required to affect the TLBs of the Armv7 PE.

I_{FSWKY} All of the following TLB maintenance instructions are added to the Armv8 T32 and A32 instruction sets, and do not exist in Armv7:

- For stage 1 translations, all of the following instructions that operate on TLB entries for the final level of translation table walk:
  - — TLBIMVALIS.
  - — TLBIMVAALIS.
  - — TLBIMVALHIS.
  - — TLBIMVAL.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - TLBIMVAAL.
  - TLBIMVALH.
- For stage 2 translations, all of the following instructions that operate by IPA on TLB entries:
  - TLBIIPAS2IS.
  - TLBIIPAS2LIS.
  - TLBIIPAS2.
  - TLBIIPAS2L.

D8.17.8 Ordering and completion of TLB maintenance instructions

R_{TGVWD} For all of the following instructions, the effects of a TLB maintenance instruction can be observed in any order relative to that instruction unless a `DSB` is executed between the instructions:

- Any load or store instruction.
- Another TLB maintenance instruction.
- A data or instruction cache maintenance instruction.

R_{QRNJC} A TLB maintenance instruction executed by a PE causes a TLB maintenance operation to occur on each PE within the Shareability domain specified by the instruction.

I_{NVLMG} If a TLB maintenance instruction has the nXS qualifier, then the associated TLB maintenance operations have the nXS qualifier.

I_{SFCHD} If a TLB maintenance instruction does not have the nXS qualifier and the Effective value of HCRX_EL2.FnXS is 1, then all of the following apply:

- When the instruction is not executed at EL1, the associated TLB maintenance operations do not have the nXS qualifier.
- When the instruction is executed at EL1, the associated TLB maintenance operations have the nXS qualifier.

I_{BBMSB} If a translation regime uses two translation stages, then the XS attribute used to determine the behavior of a TLB maintenance instruction with the nXS qualifier is the attribute determined after both translation stages have been applied.

R_{WWQZZ} When a TLB maintenance instruction is executed, *in-scope old translation information* is any translation information related to addresses within the scope of that instruction that is not consistent with one or more of the following:

- The architectural translation information held in the translation tables.
- Any architecture translation information that is Coherence-after the information held in the translation tables.

I_{YMDMJ} In-scope old translation information might be held in TLBs or other non-coherent caching structures.

R_{STPSQ} For a PE that handles a TLB maintenance operation without the nXS qualifier, the TLB maintenance operation is finished when all of the following apply:

- All memory accesses generated by that PE using in-scope old translation information are complete.
- All memory accesses, RWx, generated by that PE are complete, where RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I1 executed by that PE, where all of the following apply:
  - I1 uses the in-scope old translation information.
  - The use of the in-scope old translation information generates a synchronous Data Abort.
  - If I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access that RWx would be locally-ordered-before.

R_{TSJSN} For a PE that handles a TLB maintenance operation with the nXS qualifier, the TLB maintenance operation is finished when all of the following apply:

- All memory accesses with the XS attribute set to 0 generated by that PE using in-scope old translation information are complete.
- All memory accesses, RWx, generated by that PE are complete, where RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I1 executed by that PE, where all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— I1 uses the in-scope old translation information.
— The use of the in-scope old translation information generates a synchronous Data Abort.
— If I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access with the XS attribute set to 0 that RWx would be locally-ordered-before.

I_{KFTWB} For best real-time performance, Arm recommends that the completion of a TLB maintenance instruction with the nXS qualifier executed by a PE should not depend on the completion of any memory accesses with the XS attribute set to 1 generated by a second PE.

I_{TFQRW} To guarantee completion of memory accesses with the XS attribute set to 1, software must use a TLBI maintenance instruction without the nXS qualifier, and a `DSB` without the nXS qualifier.

R_{DCJQM} When the TLB maintenance operations specified by a TLB maintenance instruction are finished for all PEs, the TLB maintenance instruction is complete.

R_{LXHLD} If a TLB maintenance instruction is complete, then no new memory accesses using the in-scope old translation information are architecturally performed by any observer that is affected by the TLB maintenance instruction.

I_{CPZSM} If it is impossible for software running on any observer to determine that the memory accesses have been performed, then speculative memory accesses can be performed using in-scope old translation information.

I_{GYXJX} A `DSB` instruction can be used to ensure the completion of TLB maintenance instructions. For more information, see Data Synchronization Barrier.

R_{YNJCP} When a PE, PEx, executes a TLB maintenance instruction, the instruction can complete at any time after it is issued, but the instruction is only guaranteed to be finished for a PE other than PEx after the execution of `DSB` by the PEx.

R_{BLDZX} For the TLB of a PE that issues a TLB maintenance instruction, one of the following determines when the instruction is guaranteed to be complete:

- If FEAT_ETS2 is not implemented, then the instruction is guaranteed to be complete after the execution of DSB by that PE, followed by a Context synchronization event.
- If FEAT_ETS2 is implemented, all of the following determine when the instruction is guaranteed to be complete:
  — If the instruction applies only to translations without execute permission and where translation information that is coherence-after also do not have execute permission, then the instruction is guaranteed to be complete after the execution of `DSB` by that PE.
  — If the instruction applies to any translations with execute permission, then the instruction is guaranteed to be complete after the execution of `DSB` by that PE, followed by a Context synchronization event.

I_{MGRRK} For more information, see Ordering requirements defined by the formal concurrency model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.18 Caches

D8.18.1 Data and unified caches

R_{JHVQL} For data and unified caches, if all data accesses to an address do not use mismatched memory attributes, then the use of address translation is transparent to any data access to the address.

For more information, see Mismatched memory attributes.

R_{CTGML} For data accesses from the same observer to different VAs, if the accesses are translated to the same PA with the same memory attributes, then they are coherent and all of the following behaviors are guaranteed without requiring the use of barrier or cache maintenance instructions:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

R_{NGPVX} When cache maintenance is done to a memory location, the effect of that cache maintenance is visible to all VA or IPA aliases of that physical memory location.

I_{FHPPX} The properties of data and unified caches are consistent with implementing the caches as physically-indexed, physically-tagged caches.

D8.18.2 Instruction caches

R_{GXBMD} An instruction cache has all of the following properties:

- An instruction cache is accessed only as a result of an instruction fetch.
- An instruction cache is never written to by a load or store instruction.
- Cache invalidation is the only cache maintenance that can be done on an instruction cache.

R_{XBHSB} The Arm architecture permits all of the following instruction cache implementations:

- *Physically-indexed, physically-tagged* (PIPT) instruction cache.
- *Virtually-indexed, physically-tagged* (VIPT) instruction cache.

I_{XSPPH} The CTR_EL0.L1Ip field identifies the form of the instruction caches.

R_{WNDVG} If CTR_EL0.DIC is 1, instruction cache maintenance is not required after overwriting instructions, including for different VA aliases of the affected Locations, regardless of the value of CTR_EL0.L1Ip.

R_{XLZJK} For all permitted instruction cache implementations, the Arm IVIPT Extension is also implemented, which reduces instruction cache maintenance requirements to requiring maintenance only after writing new data to a PA that holds an instruction.

I_{QNJPN} Previous versions of the Arm architecture permitted an instruction cache option that does not implement the Arm IVIPT Extension.

I_{GWSPL} For software to be portable between implementations that might use any of the permitted instruction cache implementations, that software is required to invalidate the instruction cache whenever any condition occurs that requires instruction cache maintenance on at least one of the instruction cache types.

D8.18.2.1 Physically-indexed, physically-tagged instruction caches

R_{YXNGL} For PIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance is done on a memory location, then the effect of that instruction cache maintenance is visible to all VA or IPA aliases of that physical memory location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.18.2.2 Virtually-indexed, physically-tagged instruction caches

R_{LYZYY} For VIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance by address is done on a memory location, then the effect of that instruction cache maintenance is visible only to the VA supplied with the operation, and the effect of the invalidation might not be visible to any other VA or IPA aliases of that physical memory location.
- The only architecturally-guaranteed way to invalidate all VA or IPA aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

D8.18.3 Cache maintenance requirements due to changing memory region attributes

I_{VPPPM} If changes are made to the memory region attributes in translation table entries, TLB maintenance might be required. For more information, see TLB maintenance.

I_{ZDQCY} The behaviors caused by mismatched memory attributes mean that if any of the following changes are made to the Inner Cacheability or Outer Cacheability attributes in translation table entries, then software is required to ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the cache levels that might hold copies of the locations affected by the attribute change:

- A change from Write-Back to Write-Through.
- A change from Write-Back to Non-cacheable.
- A change from Write-Through to Non-cacheable.
- A change from Write-Through to Write-Back.

For more information, see Mismatched memory attributes and Using break-before-make when updating translation table entries.

R_{ZNPYM} If the Shareability attributes in translation table entries are changed, then software is required to execute the following sequence to prevent possible coherency errors caused by mismatched memory attributes:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the memory location from caches in the appropriate Shareability domain.
3. Change the Shareability attributes to the new values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.19 Pseudocode description of VMSA address translation

IZMHCV The Arm architecture defines pseudocode that describes the translation table walk.

ISTKYL **Full Physical Address**

A complete PA necessary to identify a location in physical memory is captured by the type `FullAddress`. This is composed of:

- A bitstring `address`, which identifies the PA.
- An enumeration `paspace` which identifies the PA space.

IMXLCQ **Address translation**

`AArch64_TranslateAddress()` acts as the entry point to VMSA and performs the required address translation based on the provided parameters and System register configurations. The function returns an `AddressDescriptor` structure holding valid data for either of the following:

- Target memory address and attributes for a non-faulting translation.
- Fault details holding data to be populated in syndrome registers.

ILTNYC `AArch64_FullTranslate()` selects the translation regime and performs first and potentially second stage of translation returning the PA and attributes of target memory. `AArch64_S1Translate()` carries out the first stage of translation when stage 1 is not disabled, mapping the VA to the IPA and carrying out permission checks. Otherwise, `AArch64_S1DisabledOutput()` assigns the appropriate memory attributes and flat maps the input address to the output address. `AArch64_S2Translate()` carries out stage 2 translation for the EL1&0 translation regime when enabled mapping the IPA to the PA. Otherwise, the IPA is the PA.

IKKVJD **Translation table walk**

Each stage of translation has a separate walk function, `AArch64_S1Walk()` and `AArch64_S2Walk()`, corresponding to the first and second stage of translation respectively. Each use walk parameters extracted from related System registers. Parameters are collected based on the active translation regime.

For instance, stage 1 EL2 translation regime parameters are obtained and returned by the function `AArch64_S1TTWParamsEL2()`. Given these parameters, a walk initializes a walk state of the type `TTWState`, holding the base address of the first translation table.

IXWTPZ The walk progressively fetches and decodes translation Table descriptors, updating the walk state to the next base address as it descends through the levels of tables until a Block or Page descriptor is discovered or an invalid descriptor is fetched. Decoding the descriptor for both stage 1 and stage 2 walks is carried out by the function `AArch64_DecodeDescriptorType()`.

IYMSFF For a non-faulting walk, three items are returned by a translation table walk:

- The final walk state.
- The final descriptor fetched.
- The address of the final descriptor.

IKGLCH The final descriptor and its address are used to update the descriptor as specified by Hardware management of the Access flag and Hardware management of the dirty state.

IRQKNH A faulting walk could report one of the following at a specified level:

- Translation Fault.
- Address size Fault.
- Access flag Fault.

IDNSWC **Hardware update of Translation Table descriptors**

The walk parameters collected from System registers indicate the ability to update the access flag or set write permissions within descriptors. This is controlled by the Dirty Bit Modifier, and the conditions specified in Hardware management of the Access flag and Hardware management of the dirty state. The translation functions `AArch64_S1Translate()` or `AArch64_S2Translate()` set the appropriate descriptor bits returned by the walk functions and call `AArch64_MemSwapTableDesc()` to swap the old descriptor for the updated one in an atomic fashion.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDKMRY

Address decoding and calculation

The walk state is initialized to hold the base address of the first translation table. For stage 1, the base address is acquired using `AArch64_S1TTBaseAddress()`, which decodes TTBR0_ELx and TTBR1_ELx registers depending on the regime and the range the VA corresponds to. For stage 2, the base address is acquired using `AArch64_S2TTBaseAddress()`, which decodes VTTBR_EL2 or VSTTBR_EL2 registers depending on the Security state of the executing PE. The walk progressively fetches and decodes translation Table descriptors, updating the walk state to the next base address utilizing `AArch64_NextTableBase()` as it descends through the levels of tables. Prior to every descriptor fetch, apart from the fetch at the start level, the base address is indexed by the function `AArch64_S1TTEntryAddress()` for stage 1 or `AArch64_S2TTEntryAddress()` for stage 2 to point to the specific table entry. Indexing at the start level is shown in `AArch64_S1SLTTEntryAddress()` for stage 1 and in `AArch64_S2SLTTEntryAddress()` for stage 2. The final walk state would hold the base address for the output block or page; this is extracted from the Leaf descriptor in `AArch64_S1LeafBase()` for stage 1 and in `AArch64_S2LeafBase()` for stage 2.

IGDTNP

Memory attribute decoding

If a stage of translation is enabled, fetched descriptors that are blocks or pages encode memory attributes assigned to the output of translation. Stage 1 memory attributes are decoded by the function `S1DecodeMemAttrs()`. Likewise, the stage 2 memory attributes are decoded by the function `S2DecodeMemAttrs()` followed by combining stage 1 and stage 2 attributes by the function `S2CombineS1MemAttrs()`. However, if FEAT_S2FWB is enabled, this behavior is overridden and memory attributes are decoded as specified in Stage 2 memory type and Cacheability attributes when FWB is enabled. This is captured by the function `AArch64_S2ApplyFWBMemAttrs()`.

IHGFRX

Fault detection

As soon as translation is invoked a reserve `FaultRecord` accompanies the process capturing the stage and level of translation as it proceeds. When a fault is detected, it is reflected in the `FaultRecord` and reported back as the result of translation with the most recent state to be reported already captured within. The following functions detect a certain type of fault:

- The `AArch64_S1CheckPermissions()` and `AArch64_S2CheckPermissions()` functions check permissions for stage 1 and stage 2 respectively.
- The `AArch64_S1HasAlignmentFaultDueToMemType()` and `AArch64_S2HasAlignmentFaultDueToMemType()` functions detect an alignment fault for stage 1 and stage 2 respectively.
- The `AArch64_S1TxSZFaults()` and `AArch64_S2TxSZFaults()` functions detect a Translation fault caused by erroneous configuration of TCR_ELx.T*n*SZ field. Also, the `AArch64_S2InconsistentSL()` and `AArch64_S2InvalidSL()` functions detect a stage 2 Translation fault caused by erroneous configuration of the VTCR_EL2.{SL2, SL0} and VSTCR_EL2.{SL2, SL0} fields.
- `AArch64_VAIsOutOfRange()` detects a stage 1 Translation fault caused by VAs larger than the address input size configured. Similarly, `AArch64_IPAIsOutOfRange()` detects a stage 2 Translation fault caused by the output of stage 1 being larger than the configured input size for stage 2.
- `AArch64_ContiguousBitFaults()` detects a stage 1 or 2 Translation fault caused by a mis-programmed contiguous bit within a fetched descriptor.

IPJBNQ

For more information, see function descriptions in aarch64/translation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D9
The Granule Protection Check Mechanism

Any access, after all enabled translation stages, targets a *physical address* (PA) in one of the four physical address spaces. This chapter describes the *Granule Protection Check* (GPC) mechanism by which accesses to those PA spaces are checked. The GPC mechanism is added by the *Realm Management Extension* (RME) and includes the following:

- Mechanism to determine the protection information for a particular PA and PA space.
- Allocation and invalidation behavior for TLBs, data caches, and instruction caches.
- Configuration registers and descriptor formats for PA space protection information.

This chapter contains the following sections:

- GPC behavior overview.
- GPC faults.
- GPT caching and invalidation.
- GPT formats.
- GPT lookup process.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.1 GPC behavior overview

I_{BDSNC} The PA space of an access is determined by all of the following:

- The Requester Security state.
- If enabled, then the stage 1 and stage 2 translation.

R_{BYRRZ} If GPCCR_EL3.GPC is 0, disabling granule protection checks, then accesses to all four address spaces are not subject to granule protection checks and cannot generate GPC faults.

R_{GRGXY} If GPCCR_EL3.GPC is 1, enabling granule protection checks, then all accesses are subject to granule protection checks, except for fetches of Granule Protection Table (GPT) information and accesses governed by the GPCCR_EL3.GPCP control.

R_{THJVJ} GPT walks are made to the Root physical address space and are not subject to granule protection checks.

R_{XSWYP} If the PoC is before any cache level and DC instructions to the PoC do not affect caches past the PoPA, then it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA to the PoC instruction can generate a GPC fault.

I_{RLDTY} If GPCCR_EL3.GPC is 1, enabling granule protection checks, then an access might generate one of the following GPC faults:

- Granule Protection Fault (GPF).
- GPT walk fault.
- GPT address size fault.
- Synchronous External abort on GPT fetch.

I_{QHSZT} The GPCCR_EL3.GPCP control governs behavior of granule protection checks on fetches of stage 2 Table descriptors.

I_{WJCYJ} The granule protection mechanism permits association of a peripheral Resource with a PA space to be performed by a Completer instead of the Requester, at a granularity finer than 4KB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.2 GPC bypass windows

R_{ZNCFP} All statements in this section require implementation of FEAT_RME_GPC3.

I_{JJLRM} *Granule Protection Table* (GPT) lookups can be skipped in portions of the memory map by using *GPC bypass windows*. All GPC bypass windows are the same size and have a base address that is aligned to a common stride value.

I_{XJPJF} The GPCBW_EL3 system register describes the base, size, and stride properties of the GPC bypass windows. The following diagram illustrates how GPC bypass windows are described using GPCBW_EL3 fields.

Example:

GPCBW_EL3.BWADDR = 2 → GPC Bypass window base address = 2GB
GPCBW_EL3.BWSIZE = 0 → GPC Bypass window size = 1GB
GPCBW_EL3.BWSTRIDE = 1 → GPC Bypass window stride = 4TB

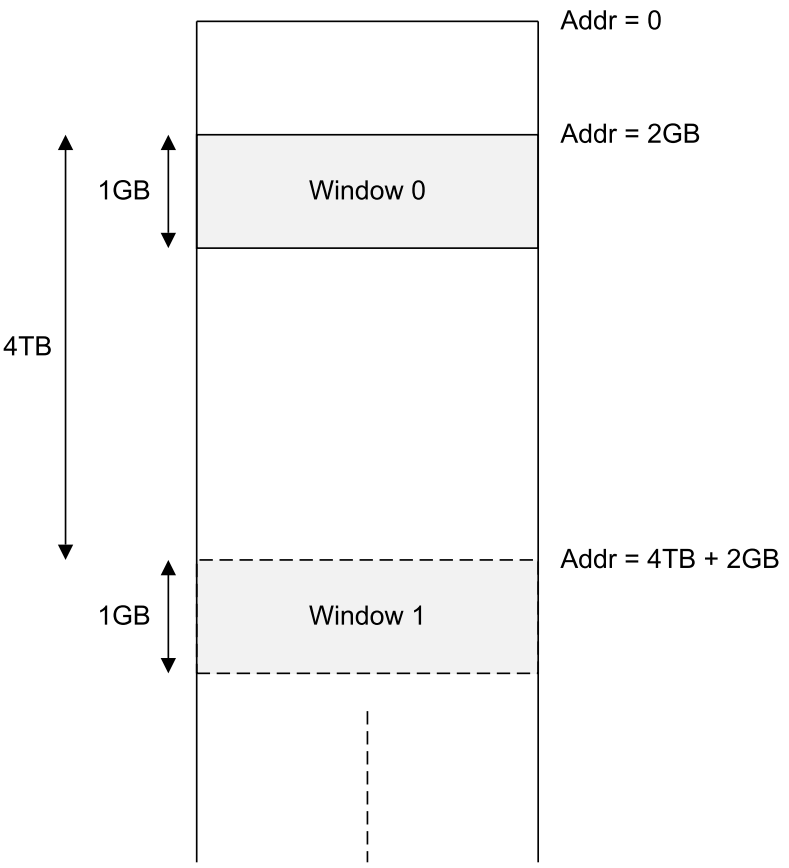

Figure D9-1 GPC bypass windows

R_{SRQWF} If GPCCR_EL3.GPCBW is set to 1 and a PA falls within a GPC bypass window range, then the GPC behaves as if the GPI encoding for the PA is *All accesses permitted*.

R_{VKXCB} GPC bypass windows are always located within the protected PA space that is configured using GPCCR_EL3.PPS.

I_{XNHTX} The GPC bypass window check does not modify the GPC fault reporting priority shown in Table D9-2, and is performed immediately after priority 3, *Granule protection fault at Level 0*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.3 Granular Data Isolation

R_{KWXTL} All statements in this section require implementation of FEAT_RME_GDI.

I_{CPKHY} The Granular Data Isolation feature enables memory isolation of non-PE data flows from the PEs, within an RME system.

I_{KCSQM} Two PA spaces are defined for non-PE data granules:

- Non-secure Protected (NSP) memory, which is managed by a PE, using the SMMU, on behalf of a NS device in protected mode.
- System Agent (SA) memory, which is fully isolated from the PEs, and any additional memory management is independent from the PEs.

Note

NSP memory is intended for media pipelines, with flexible NS software management and strong data confidentiality. SA memory is intended for use by higher security on-chip subsystems that require memory allocation on request.

I_{CQKLR} NSE2 extends NS and NSE for the addressing of PA spaces:

NSE2	NSE	NS	Target PA Space
0	0	0	S - Secure
0	0	1	NS - Non-secure
0	1	0	RT - Root
0	1	1	RL - Realm
1	0	0	SA - System Agent
1	0	1	NSP - NS Protected
1	1	0	NA6 - Reserved
1	1	1	NA7 - Reserved

R_{MXHVD} In a PE with FEAT_RME_GDI, with the exception of `DC CIPAPA`, `DC CIGDPAPA`, `DC CIPAE` and `DC CIGDPAE`, NSE2 has the effective value of zero within a PE.

R_{QCLPT} A PE is not permitted to access memory within the NSP or the SA PA spaces, from any Security state.

I_{QGYGB} Data within the NSP or the SA PA spaces is not anticipated to require PE managed coherency, through the use of cache maintenance operations to the PoC. However, cache maintenance operations to the PoPA are supported.

I_{CWGLY} FEAT_RME_GDI does not define new Security states.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.4 GPC faults

R_{JWCSM} If the GPC for an access requires use of configuration in GPCCR_EL3 or in GPCBW_EL3, and the configuration of GPCCR_EL3 or GPCBW_EL3 is invalid, then the access generates a *GPT walk fault at level 0*.

The configuration of GPCCR_EL3 or GPCBW_EL3 is invalid if any of the following are true:

- Any field in GPCCR_EL3 is programmed to a reserved value.
- Any field in GPCBW_EL3 is programmed to a reserved value when GPCCR_EL3.GPCBW is 1.
- Any field in GPCCR_EL3 is programmed to an invalid value, as specified in the definition of GPCCR_EL3.
- Any field in GPCBW_EL3 is programmed to an invalid value, as specified in the definition of GPCBW_EL3, when GPCCR_EL3.GPCBW is 1.

R_{KYZMZ} If the GPC for an access requires consumption of any field in an invalid GPT entry, then the access generates a *GPT walk fault at level x*, where x is the level of the invalid GPT entry.

R_{PGHWX} If FEAT_RME_GDI is implemented, and a GPT walk needs to consume an L1 GPT entry, then the GPT walk also reads the other GPT entry that is in the same 16-byte naturally aligned region of memory. If either entry is invalid, the following occur:

- A GPT walk fault at level 1 is generated.
- Neither GPT entry is cached in a TLB.

I_{QYQPV} In the case where a GPT walk generates a *GPT walk fault at level 1*, hardware is not required to automatically invalidate any previously-cached valid descriptors from the TLB.

R_{XVCKY} If the GPC for an access requires use of the configured base address in GPTBR_EL3.BADDR, and the base address exceeds the configured address size in GPCCR_EL3.PPS, then the access generates a *GPT address size fault at level 0*.

R_{JCGMZ} If the GPC for an access requires consumption of a GPT Table descriptor with an address that exceeds the value configured in GPCCR_EL3.PPS, then the access generates a *GPT address size fault at level 0*.

R_{DFCHJ} If a fetch of GPT information to check an access experiences an External abort, then the access generates a *synchronous External abort on GPT fetch at level x*, where x is the level of the fetch that experienced the External abort.

R_{CPDSB} If a Non-secure PA input to the GPC exceeds the PA range configured by GPCCR_EL3.PPS, then the access does not generate any GPF.

R_{TCWMD} If any of the following apply to the PA input to the GPC, the access generates a Granule protection fault at Level 0:

- The input is a Secure PA and the Effective value of GPCCR_EL3.SPAD is 1.
- The input is a Non-secure PA and the Effective value of GPCCR_EL3.NSPAD is 1.
- The input is a Realm PA and the Effective value of GPCCR_EL3.RLPAD is 1.

R_{JFFHB} If the Effective value of GPCCR_EL3.APPSAA is 0, then when a Secure, Realm, or Root PA input to the GPC exceeds the PA range configured by GPCCR_EL3.PPS, then the access generates a *Granule protection fault at Level 0*.

R_{PBPSH} If the Effective value of GPCCR_EL3.APPSAA is 1, then when a Secure, Realm, or Root PA input to the GPC exceeds the PA range configured by GPCCR_EL3.PPS, the access does not generate a GPF due to this control.

R_{DQPWS} An access is not permitted by the GPT if it is made to a PA space not permitted according to the *Granule Protection Information* (GPI) value returned by the GPT lookup.

R_{HDDNW} If an access is not permitted by the GPT, then the access generates a *Granule protection fault at Level x*, where x is the level of the GPT entry that the access was checked against.

R_{VJLXG} Accesses are checked against the GPC configuration for the physical granule being accessed, regardless of the stage 1 and stage 2 translation configuration.

I_{KLTDM} For example, if GPCCR_EL3.PGS is configured to a smaller granule size than the configured stage 1 and stage 2 translation granule size, accesses are checked at the GPCCR_EL3.PGS granule size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{GMGRR} The following table shows the possible GPC faults, from highest priority to lowest priority, and the reasons they are generated:

Table D9-2 GPC fault priority

Priority	Fault reported	Reason
1	GPT walk fault at Level 0	The configuration of GPCCR_EL3 is invalid, or GPCCR_EL3.GPCBW is 1 and the configuration of GPCBW_EL3 is invalid.
2	Granule protection fault at Level 0	An access to Secure, Non-secure, or Realm PA space is prevented by one of GPCCR_EL3.{SPAD, NSPAD, RLPAD}.
3	Granule protection fault at Level 0	The Effective value of GPCCR_EL3.APPSAA is 0 and a Secure, Realm, or Root physical address exceeds GPCCR_EL3.PPS.
4	GPT address size fault at Level 0	The base address in GPTBR_EL3.BADDR exceeds GPCCR_EL3.PPS.
5	Synchronous External abort on GPT fetch at Level 0	An L0GPT fetch experiences an external abort.
6	GPT walk fault at Level 0	An L0GPT entry is invalid.
7	GPT address size fault at Level 0	An L0GPT entry contains an address exceeding GPCCR_EL3.PPS.
8	Granule protection fault at Level 0	An L0GPT entry forbids access.
9	Synchronous External abort on GPT fetch at Level 1	An L1GPT fetch experiences an external abort.
10	GPT walk fault at Level 1	An L1GPT entry is invalid.
11	Granule protection fault at Level 1	An L1GPT entry forbids access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.5 GPT caching and invalidation

R_{YJGPL} All fetches of GPT information use Normal memory types.

R_{RKFVK} The Cacheability and Shareability attributes of GPT fetches are configured in GPCCR_EL3.

I_{CDFPQ} Fetched GPT information might be cached in a data cache, according to the Normal memory Cacheability attributes and allocation hints configured in GPCCR_EL3.

I_{ZJYLQ} The Cacheability of GPT fetches is controlled by GPCCR_EL3.{IRGN, ORGN} and is not affected by the SCTLR_ELx.C or HCR_EL2.{CD, DC} control bits.

R_{XNFGN} GPT fetches are made with behavior consistent with PBHA being disabled or programmed to zero, regardless of the PBHA configuration at stage 1 and stage 2. See Page Based Hardware attributes.

R_{YMSWK} GPT entries are permitted to be cached in TLBs combined with stage 1 and stage 2 information, as long as the requirements of TLB invalidation instructions are met.

R_{VFKSY} TLBs containing GPT information are disabled at reset. Any IMPLEMENTATION DEFINED or UNKNOWN GPT information in TLBs has no effect on accesses until granule protection checks, or any stages of translation are enabled.

R_{YMRVT} GPT information cached in a TLB is permitted to be shared across multiple PEs, except for PEs with GPCCR_EL3.GPC set to 0 and all translation stages disabled.

R_{XLDKK} For two PEs that are permitted to share GPT information cached in TLBs, if the configuration of GPCCR_EL3, GPTBR_EL3, and the GPT is not consistent across those PEs, then the behavior on one PE is a CONSTRAINED UNPREDICTABLE choice of:

- The configuration for that PE.
- The configuration of the other PE.
- A combination of the configuration of the two PEs.

I_{BSPQD} To avoid CONSTRAINED UNPREDICTABLE behavior, Root firmware must ensure that both:

- Before GPCCR_EL3.GPC is set to 1, GPCCR_EL3 and GPTBR_EL3 are otherwise configured consistently with other PEs.
- Before enabling any translation stage, GPCCR_EL3.GPC is set to 1.

R_{RQCBQ} A level 0 GPT entry is reachable if the entry is in the configured PA range of GPTBR_EL3 and GPCCR_EL3, and GPT configuration does not generate a GPC fault at level 0.

R_{MGSTK} A level 1 GPT entry is reachable if a reachable or previously cached level 0 GPT entry points to it, and that level 0 GPT entry does not generate a GPC fault.

R_{BFQRM} GPT entries may only be fetched if they are reachable.

R_{QBKYP} GPT entries may only be cached in a TLB if they are reachable and valid.

I_{JMYRB} Because GPT entries are permitted to be cached in a TLB if they are reachable and valid, translations that result in a GPF are permitted to be cached in a TLB.

I_{DRHKK} All of the following are the TLB maintenance instructions that invalidate cached GPT entries:

- `TLBI RPAOS, <Xt>`.
- `TLBI RPALOS, <Xt>`.
- `TLBI PAALLOS`.
- `TLBI PAALL`.

I_{YBBZK} The `TLBI *PA*` instructions are only present at EL3. They are UNDEFINED at EL2 and below.

R_{NBJFD} `TLBI *PA*` instructions invalidate GPT information cached in TLB entries, including in intermediate TLB caching structures, according to the requirements specified in this section.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{JJHVQ} The Arm architecture permits a range of TLB implementation styles, including TLB caching structures that store entries that combine information from stage 1 and stage 2 translation table entries.

GPT information is permitted to be cached in combination with information from stage 1 and stage 2 translation table entries, as long as the requirements for invalidation of GPT information by `TLBI *PA*` operations are met. For example:

- An implementation that caches GPT information separately from stage 1 and stage 2 information is only required to invalidate GPT information as a result of a `TLBI *PA*` operation.
- An implementation that caches entries that combine stage 2 OA information with GPT information must invalidate all such entries in response to a `TLBI PAALLOS` operation.
- An implementation that caches entries that combine information from stage 2 level 2 Table descriptors with GPT information must invalidate those entries in response to a `TLBI *PA*` operation that matches the next-level address of those level 2 Table descriptors. It is not required to invalidate those entries when a `TLBI *PA*` matches the PA that the level 2 descriptor was fetched from.

I_{XZTJV} A `TLBI RPA*` instruction applies to TLB entries containing GPT information relating to the supplied PA range.

I_{ZDVNB} A `TLBI PAALL*` instruction applies to all TLB entries containing GPT information.

I_{BKJTM} A `TLBI PAALL*` instruction also applies to any TLB entry derived from GPC configuration register fields that are permitted to be cached in a TLB.

I_{JQRVK} The remaining instruction syntax is the same as for Armv8-A. This means:

- `{R}` is a specifier denoting range-based invalidation.
- `{L}` is an optional specifier that reduces the scope of the invalidation to cached GPT entries fetched from the final level of the GPT walk.
- `{OS}` specifies that the TLBI applies to all the TLBs in the Outer Shareable domain. `TLBI *PA*` operations without `OS` are only required to apply to the PE executing the operation.
- `<Xt>` specifies that the instruction takes an `X` register as an argument to pass additional information about the invalidation scope.

R_{LRKLF} For `TLBI *PA*` instructions, Outer Shareable scope is sufficient to affect all TLBs in the system.

I_{SXTLQ} The `TLBI *PA*` operations do not have an nXS qualifier and always behave as though they are issued without an nXS qualifier.

I_{FRSHC} The Arm architecture has IMPLEMENTATION DEFINED support for TLB lockdown, and the interaction between TLB lockdown and existing data TLB maintenance instructions is IMPLEMENTATION DEFINED.

R_{BFXRL} `TLBI *PA*` operations affect TLB entries containing GPT information regardless of any TLB lockdown configuration.

I_{TMCTS} The `TLBI *PA*` operations have the same rules for ordering, observability, and completion as all other TLBI instructions.

I_{PLYZN} The `TLBI RPAOS` instruction invalidates TLB entries containing GPT information from any level of the GPT walk relating to the supplied PA range.

I_{VLLLY} The `TLBI RPALOS` instruction invalidates TLB entries containing GPT information from the final level of the GPT walk relating to the supplied PA range.

I_{ZZJVG} Consistent with all other `TLBI` instructions, over-invalidation is permitted, and under-invalidation is not.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.6 GPT formats

R_{QKHMJ} The in-memory structure that describes the association of physical granules with PA spaces is the *Granule Protection Table* (GPT).

I_{PZSYC} For implementations that choose to do so for area or performance reasons, the architecture permits caching of GPT information in a TLB.

R_{JLXZV} A successful GPT lookup resolves an input PA to the GPI for that address.

R_{TRVSY} A GPT descriptor is one of a Table, Block, Contiguous, or Granules descriptor.

R_{JXNXP} A GPT descriptor is eight bytes.

R_{BQHPD} All structures in the GPT are little-endian.

I_{KFVNY} All GPT entries are naturally-aligned in memory.

R_{VXNGT} The GPT has two levels of lookup.

R_{TRCQY} All valid entries in a level 0 GPT are GPT Block or GPT Table descriptors.

R_{TXFXH} A level 0 GPT entry that is not a GPT Block or GPT Table descriptor is invalid.

R_{DCTFM} All valid entries in a level 1 GPT are GPT Contiguous or GPT Granules descriptors.

R_{TPBZN} A level 1 GPT entry that is not a GPT Contiguous or GPT Granules descriptor is invalid.

R_{XNKFZ} A GPT entry is invalid if any of the following are true:

- A field in the entry is configured with an encoding marked as reserved.
- A bit location in the entry marked as RES0 is nonzero.

I_{VRKNJ} If FEAT_RME_GDI is implemented, the check of validity on level 1 GPT entries is performed on pairs of entries. See GPC faults.

I_{XJKRS} This is to increase the probability of detecting errors relating to a loss of integrity of the memory holding the GPT.

D9.6.1 GPT Table descriptor

R_{RCTBJ} A GPT Table descriptor contains a pointer to the base address of a next-level table, and fields describing properties relating to the remaining levels of walk.

I_{HKPQF} The following figure shows the level 0 GPT Table descriptor format:

Level 0 GPT Table descriptor

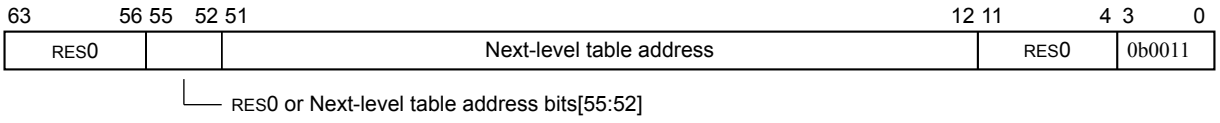

Figure D9-2 Level 0 GPT Table descriptor format

R_{KCDQS} GPT Table descriptor bits[63:56] are RES0.

R_{LHVFM} GPT Table descriptor bits[55:52] are bits [55:52] of the next-level table address, if FEAT_RME_GPC3 is implemented and GPCCR_EL3.PPS is configured to 56 bits, and are otherwise RES0.

R_{NWCYC} GPT Table descriptor bits[51:12] are bits [51:12] of the next-level table address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SFPYD} GPT Table descriptor bits[11:4] are RES0.

R_{GPPXX} GPT level 0 entry bits[3:0] are `0b0011`, indicating the entry is a GPT Table descriptor.

R_{DBTFW} The alignment of the next-level table address depends on the value of GPCCR_EL3.PGS as follows:
Descriptor bits [s-p-2:12] are RES0, where:

- *s* is derived from GPCCR_EL3.L0GPTSZ as follows:

GPCCR_EL3.L0GPTSZ	Size indicated	*s*
`0b0000`	1GB	30
`0b0100`	16GB	34
`0b0110`	64GB	36
`0b1001`	512GB	39

- *p* is derived from GPCCR_EL3.PGS as follows:

GPCCR_EL3.PGS	Size indicated	*p*
`0b00`	4KB	12
`0b10`	16KB	14
`0b01`	64KB	16

I_{BFKHR} Level 1 tables are aligned to their size in memory. The size of level 1 tables is determined by GPCCR_EL3.PGS and GPCCR_EL3.L0GPTSZ.

D9.6.2 GPT Block descriptor

I_{NKQNF} The following figure shows the level 0 GPT Block descriptor format:

Level 0 GPT Block descriptor

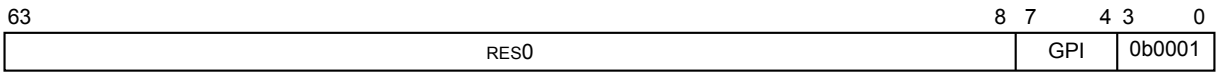

Figure D9-3 Level 0 GPT Block descriptor format

R_{PHBLQ} GPT Block descriptor bits[63:8] are RES0.

R_{FZGCP} GPT Block descriptor bits[7:4] are the GPI value. For more information, see GPI field encoding in GPT descriptors.

R_{FGPWN} GPT level 0 entry bits[3:0] are `0b0001`, indicating the entry is a GPT Block descriptor.

R_{PLSSK} GPT information from a level 0 GPT Block descriptor is permitted to be cached in a TLB as though the block is a contiguous region of granules, each of the size configured in GPCCR_EL3.PGS.

R_{YNKWN} If the range encoded in the SIZE field of the invalidation covers the full address range of the Block, as advertised in GPCCR_EL3.L0GPTSZ, a `TLBI RPA*` operation is only required to invalidate cached information from a level 0 GPT Block descriptor.

R_{GXNNL} When GPT configuration is changed between the two following structures, Granule protection checks continue to be made correctly, even if a TLBI is not issued:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A level 0 GPT Block descriptor indicating a GPI value for a region.
- A level 0 GPT Table descriptor pointing at a level 1 table of Contiguous or Granules descriptors that have the same GPI value as the level 0 Block descriptor.

When a level 0 Table descriptor is replaced with a level 0 Block descriptor, the hardware may continue to access the level 1 Table until completion of a non-Last-level TLBI by PA, targeting at least the full address range of the level 0 descriptor. This means that the memory containing the level 1 Table cannot be reclaimed for other uses until completion of that TLBI by PA operation.

D9.6.3 GPT Granules descriptor

I_{NRSQJ} The following figure shows the level 1 GPT Granules descriptor format:

Level 1 GPT Granules descriptor

63 60	59 56	55 52	51 48	47 44	43 40	39 36	35 32	31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI	GPI

Figure D9-4 Level 1 GPT Granules descriptor format

R_{HJWQH} An 8-byte GPT Granules descriptor contains the GPI values for 16 physical granules. For more information, see GPI field encoding in GPT descriptors.

R_{QDCZJ} The following table describes how the GPI values within one Granules descriptor are indexed:

GPCCR_EL3.PGS	Size indicated	Within Granules descriptor
`0b00`	4KB	i = PA[15:12]
`0b10`	16KB	i = PA[17:14]
`0b01`	64KB	i = PA[19:16]

The GPI value to use is descriptor bits $[(4*i) + 3 : (4*i)]$.

R_{GQPWL} GPT level 1 entry bits[3:0] are a valid GPI encoding, indicating the entry is a GPT Granules descriptor. For more information, see GPI field encoding in GPT descriptors.

D9.6.4 GPT Contiguous descriptor

I_{SPSCW} The following figure shows the level 1 GPT Contiguous descriptor format:

Level 1 GPT Contiguous descriptor

63 10	9 8	7 4	3 0
RES0	Contig	GPI	0b0001

Figure D9-5 Level 1 GPT Contiguous descriptor format

R_{MVVWZ} GPT Contiguous descriptor bits[63:10] are RES0.

R_{JCXJC} GPT Contiguous descriptor bits[9:8] are the Contig field.

R_{JLQNJ} GPT Contiguous descriptor bits[7:4] are the GPI value. For more information, see GPI field encoding in GPT descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{BSSVP} GPT level 1 entry bits[3:0] are `0b0001`, indicating the entry is a GPT Contiguous descriptor.

R_{BFCGF} The following table describes the Contig field encoding:

Value	Meaning
`0b00`	Reserved
`0b01`	2MB
`0b10`	32MB
`0b11`	512MB

R_{MNZWK} Information from a GPT Contiguous descriptor is permitted to be cached in a TLB or a table walk cache for an input address range up to the size indicated by the Contig field.

R_{CZJSQ} Contiguous regions are naturally-aligned.

I_{QJZQH} For example, if the Contig field in the Contiguous descriptor for address `0x8000_4000` indicates a 2MB contiguous region, the region is `0x8000_0000` to `0x801F_FFFF`.

R_{RQBNP} GPT entries marked for contiguity are permitted, but not required, to be cached as block entries.

R_{SSKBB} TLB Invalidation of GPT information is only guaranteed by TLB maintenance of the full contiguous range.

I_{KBTDW} For example, this might be done by executing a `TLBI RPALOS, <Xt>` instruction covering the full range of the contiguous GPT region.

I_{NZJDP} This requirement on TLBI scope is intended to be the same as the behavior of the Contiguous bit in the VMSA. For more information, see The Contiguous bit.

R_{SPLJH} If any of the GPI values in GPT descriptors within the range specified by a Contig field differ from each other, then the GPT Contiguous descriptor has been *misprogrammed*.

R_{SMQTZ} For an access to a memory location within the range specified by the Contig field, if a GPT Contiguous descriptor has been misprogrammed, then in the absence of other faulting conditions it is CONSTRAINED UNPREDICTABLE whether one of the following occurs:

- The access succeeds as though its PA space is permitted by a programmed GPI value in the range.
- The access experiences a GPF consistent with the access not being permitted by one of the GPI values configured for the range.

R_{NNHCF} If a GPT Contiguous descriptor has the Contig field configured to one value, and other GPT Granules descriptors or Contiguous descriptors within the range indicated by that Contig field are all configured with the same GPI values, then in the absence of both misprogramming and faulting conditions, accesses to that range are correctly checked against the GPI value programmed for the range.

I_{JMVJS} This behavior is intended to be the same as FEAT_BBML2 behavior, but with the option of TLB Conflict aborts removed.
For more information, see Support levels for changing table or block size.

I_{RQFBR} For more information, see:

- GPC faults
- GPT caching and invalidation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.6.5 GPI field encoding in GPT descriptors

R_{GYQGW} The following table describes the GPI field encoding:

Value	Meaning
`0b0000`	No accesses permitted.
`0b0100`	Accesses permitted to System Agent PA space only. This encoding is reserved if GPCCR_EL3.SA is 0.
`0b0101`	Accesses permitted to Non-secure Protected PA space only. This encoding is reserved if GPCCR_EL3.NSP is 0.
`0b0110`	No accesses permitted. This encoding is reserved if GPCCR_EL3.NA6 is 0.
`0b0111`	No accesses permitted. This encoding is reserved if GPCCR_EL3.NA7 is 0.
`0b1000`	Accesses permitted to Secure PA space only. This encoding is reserved if FEAT_SEL2 is not implemented.
`0b1001`	Accesses permitted to Non-secure PA space only
`0b1010`	Accesses permitted to Root PA space only
`0b1011`	Accesses permitted to Realm PA space only
`0b1101`	Accesses permitted to Non-secure PA space only, by Non-secure or Root Security states. This encoding is reserved if the Effective value of GPCCR_EL3.NSO is 0.
`0b1111`	All accesses permitted
Otherwise	Reserved

I_{XKBXW} Since a PE is not permitted to access memory within the NSP or the SA PA spaces, GPI encodings `0b0100` to `0b0111` are effectively “No access permitted” for PEs. These encodings are specified in the PE to allow it to distinguish between valid table entries with known encodings and invalid table entries that contain reserved encodings.

I_{YDVBY} The GPI encoding *All accesses permitted* might be used for mapping peripherals that perform register banking based on the PA space of an access.

I_{LJQWQ} The `0b1101` GPI field encoding extends the GPC to distinguish between different Security states when checking access to the Non-secure PA space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.7 GPT lookup process

R_{NGQRV} All accesses made by the MMU to the GPT are 64-bit single-copy atomic.

R_{QNGFY} Tables at each GPT level are indexed by the input PA bits, according to the values of GPCCR_EL3.{PPS, PGS, L0GPTSZ}.

I_{GRXPD} The following table describes how the index information from a PA input is decoded into the GPT lookup:

PA bits	Interpretation
PA[51:*t*]	Only applies if $t < 52$. Checked against GPCCR_EL3.PPS
PA[*t*-1:*s*]	Index into level 0 table
PA[*s*-1:*p*+4]	Index into level 1 table
PA[*p*+3:*p*]	Index of GPI within level 1 table entry

Where:

- The bit position *t* has the same value as the configured protected PA size, decoded from GPCCR_EL3.PPS.
- The bit position *s* has the same value as the supported L0GPT entry size, decoded from GPCCR_EL3.L0GPTSZ.
- The bit position *p* has the same value as the address width of the physical granule size configured in GPCCR_EL3.PGS:
  - `0b00`, 4KB, $p = 12$
  - `0b10`, 16KB, $p = 14$
  - `0b01`, 64KB, $p = 16$

R_{RDYKY} The following table describes how the level 0 table is indexed by PA bits:

GPCCR_EL3.PPS[3:0]	Level 0 index
`0b0000`	PA[31:*s*]
`0b0001`	PA[35:*s*]
`0b0010`	PA[39:*s*]
`0b0011`	PA[41:*s*]
`0b0100`	PA[43:*s*]
`0b1000`, supported when FEAT_RME_GPC3 is implemented.	PA[45:*s*]
`0b1001`, supported when FEAT_RME_GPC3 is implemented.	PA[46:*s*]
`0b0101`	PA[47:*s*]
`0b0110`	PA[51:*s*]
`0b0111`, supported when FEAT_RME_GPC3 is implemented.	PA[55:*s*]

The bit position *s* has the same value as the supported L0GPT entry size, decoded from GPCCR_EL3.L0GPTSZ.
If GPCCR_EL3.PPS is configured for a range smaller than or equal to the range advertised in GPCCR_EL3.L0GPTSZ, then the level 0 table contains only one entry, at offset zero from the configured table base.

R_{RSHYW} The following table describes how the level 1 table is indexed by PA bits:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPCCR_EL3.PGS	Size indicated	Level 1 index
`0b00`	4KB	PA[*s*-1:16]
`0b10`	16KB	PA[*s*-1:18]
`0b01`	64KB	PA[*s*-1:20]

The bit position *s* has the same value as the supported L0GPT entry size, decoded from GPCCR_EL3.L0GPTSZ.

I_{LJQCV} The following table shows how the amount of memory occupied by a level 1 table depends on GPCCR_EL3.L0GPTSZ and GPCCR_EL3.PGS:

L0GPTSZ	PGS=4KB	PGS=16KB	PGS=64KB
`0b0000`, 30 bits	128KB	32KB	8KB
`0b0100`, 34 bits	2MB	512KB	128KB
`0b0110`, 36 bits	8MB	2MB	512KB
`0b1001`, 39 bits	64MB	16MB	4MB

D9.7.1 Ordering of memory accesses from GPT walks

I_{CLGHP} If FEAT_ETS2 is implemented, and a memory access RW_1 is Ordered-before a second memory access RW_2, then RW_1 is also Ordered-before any translation table walk generated by RW_2 that generates any of the following:

- A Translation fault.
- An Address size fault.
- An Access flag fault.

For more information, see Ordering of memory accesses from translation table walks.

R_{CKTPD} If FEAT_RME is implemented, and a memory access RW_1 is Ordered-before a second memory access RW_2, then RW_1 is also Ordered-before any GPT walk generated by RW_2 that generates any of the following:

- A GPT walk fault.
- A GPT address size fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.8 Explicit PAS control

I_{HHZGR} The `APAS` instruction provides an architected mechanism for directly programming the PA space association for regions of memory that support memory-side dynamic PA space association. For example, this might be needed for memory expansion devices that maintain a local association between PA and PA space at cache-line granularity.

R_{LGVKV} If FEAT_RME_GPC3 is implemented, then an `APAS Xt` instruction is defined, which can be used to set the PA space of specific memory locations that are sensitive to the instruction.

R_{XDGGD} It is IMPLEMENTATION DEFINED which regions of memory support the `APAS` instruction. For regions that support the `APAS` instruction, the following are also IMPLEMENTATION DEFINED:

- The supported PA spaces.
- The meaning of the Target Attributes field when it is non-zero.

R_{BGQNC} The granularity that an `APAS` instruction applies to is an IMPLEMENTATION DEFINED naturally-aligned value that is at least 64 bytes.

I_{DQJHQ} If the Target Attributes field is non-zero then the instruction might indicate a granularity larger than 64B.

I_{CDCDK} Arm expects that if `APAS` is issued to a location that does not support the `APAS` instruction, or to a location that cannot be associated with the indicated PA space, then the instruction has no effect on the location and does not generate an External abort.

R_{JYZRN} An `APAS` instruction is permitted, but not required, to change the contents of the affected locations.

R_{JHVFK} An `APAS` instruction has the same ordering behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain. See Ordering and completion of data and instruction cache instructions for more information. This includes aspects relating to the minimum size of cache lines, indicated by CTR_EL0.DminLine.

Completion of a `DSB SY` guarantees that the effects of an `APAS` instruction are observable by all observers inside and outside of the Outer Shareable shareability domain.

I_{VQTWW} The `APAS` instruction is UNDEFINED at EL2 and below.

R_{NZVRX} The `APAS` instruction is not subject to granule protection checks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D10
The Memory Tagging Extension

This chapter describes the Memory Tagging Extension. It contains the following sections:

- Introduction.
- Allocation Tags.
- Memory region tagging types.
- Tag checking.
- Physical Tag locations.
- Tag Check Faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.1 Introduction

I_{QGVTK} The *Memory Tagging Extension* (MTE) describes the following concepts:

- *Allocation Tags* that are held at *Tag locations* and are associated with blocks of memory called *Tag Granules*.
- *Logical Address Tags* that form the upper bits of addresses used for *Tag Checked* memory accesses.
- *Tag Check operations* in which the *Allocation Tags* and *Logical Address Tags* are compared.
- *Tag Check Faults* that are generated from unsuccessful *Tag Check operations*.

I_{MFPTL} For more information, see:

- Allocation Tags.
- Tag Checked memory accesses.
- Tag Check operations.
- Tag Check Faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.2 Allocation Tags

D_{LLSFJ} All statements in this section require implementation of FEAT_MTE2.

R_{CWPLY} Each set of 16 naturally-aligned bytes in memory is a *Tag Granule*.

R_{XQBST} Each *Tag Granule* is associated with an *Allocation Tag* in memory.

R_{CRCLC} An *Allocation Tag* is a 4-bit value.

R_{QYCTT} All accesses to any *Allocation Tag* are single-copy atomic.

R_{CPMDB} All implicit and explicit accesses to *Allocation Tags* are regarded as streams of accesses to *Allocation Tags*, and no atomicity between accesses to different *Allocation Tags*, or between accesses to *Allocation Tags* and accesses to bytes, is ensured by the architecture.

R_{YSRGL} An instruction that explicitly accesses an *Allocation Tag* at a VA accesses the *Allocation Tag* at the VA aligned to the *Tag Granule*.

R_{PJCWS} For an instruction that explicitly accesses both *Allocation Tags* and data, any MMU faults generated by the separate accesses are not ordered.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.3 Memory region tagging types

R_{SWGZW} A *memory region tagging type* is one of the following:

- *Tagged* if FEAT_MTE2 is implemented.
- *Canonically Tagged* if FEAT_MTE_CANONICAL_TAGS is implemented.
- *Untagged*.

R_{JJRJP} The *memory region tagging type* for a memory access is:

- *Tagged* if all of the following are true:
  - FEAT_MTE2 is implemented.
  - *Allocation Tag Access* is enabled for the memory region. For more information, see Allocation Tag Access controls.
  - The stage 1 attributes define the *Tagged* attribute.
  - The combined effects of stage 1 and stage 2 translations define the memory attributes as Normal memory, Inner and Outer Write-Back, Non-Transient, Read-Allocate and Write-Allocate.
- *Otherwise it is Canonically Tagged* if all of the following are true:
  - FEAT_MTE_CANONICAL_TAGS is implemented.
  - Allocation Tag Access is enabled for the memory access.
  - *Canonical Tagging* is enabled for the memory region.
  - The stage 1 attributes do not define the *Tagged* attribute.
- Otherwise it is CONSTRAINED UNPREDICTABLE if it is *Canonically Tagged* or *Untagged* if all of the following are true:
  - FEAT_MTE_CANONICAL_TAGS is implemented.
  - Allocation Tag Access is enabled for the memory access.
  - *Canonical Tagging* is enabled for the memory region.
  - The stage 1 attributes define the *Tagged* attribute.
  - The combined effects of stage 1 and stage 2 translations do not define the memory attributes as Normal memory, Inner and Outer Write-Back, Non-Transient, Read-Allocate and Write-Allocate.
- Otherwise it is *Untagged*.

R_{XNJXP} If FEAT_MTE_CANONICAL_TAGS is implemented, *Canonical Tagging* is configured by the following controls:

- If the stage 1 translation supports two VA ranges, TCR_ELx.MTX*n*.
- If the stage 1 translation supports a single VA range, TCR_ELx.MTX.

R_{GCCBJ} For the purpose of determining the *memory region tagging type* for a memory region, an unprivileged access generated by an unprivileged memory access instruction is treated as if the access is at EL0.

R_{YSLYC} If the *memory region tagging type* for a memory region is:

- *Tagged*, then all of the following are true:
  - For the EL1&0 translation regime:
    - If HCR_EL2.DC is 0 and the stage 1 translation assigns the Non-shareable attribute, then it is IMPLEMENTATION DEFINED whether the memory region is treated as Tagged or Untagged.
    - If the combined stage 1 and stage 2 Shareability attributes are Non-shareable, then it is IMPLEMENTATION DEFINED whether the memory region is treated as Tagged or Untagged.
  - For a translation regime other than EL1&0:
    - If the stage 1 translation assigns the Non-shareable attribute, then it is IMPLEMENTATION DEFINED whether the memory region is treated as Tagged or Untagged.
  - If SCTLR_ELx.C is 0 for a memory access, it is CONSTRAINED UNPREDICTABLE whether the memory region is treated as *Tagged* or *Untagged*.
- *Canonically Tagged*, then all of the following are true:
  - For the EL1&0 translation regime:
    - If HCR_EL2.DC is 0 and the stage 1 translation assigns the Non-shareable attribute, then it is IMPLEMENTATION DEFINED whether the memory region is treated as Canonically tagged or Untagged.
    - If the combined stage 1 and stage 2 Shareability attributes are Non-shareable, then it is IMPLEMENTATION DEFINED whether the memory region is treated as Canonically tagged or Untagged.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a translation regime other than EL1&0:
  - If the stage 1 translation assigns the Non-shareable attribute, then it is IMPLEMENTATION DEFINED whether the memory region is treated as Canonically tagged or Untagged.
- SCTLR_ELx.C has no effect on whether the memory region is treated as *Canonically Tagged*.

I_{QZXCY} To ensure consistent behavior, software can set SCTLR_ELx.ATA to 0 when SCTLR_ELx.C is 0.

R_{WNMML} For a read of an *Allocation Tag* all of the following are true:

- If the read is from an *Untagged* memory region:
  - The read returns the value `0b0000`.
  - All effects of the access other than the read of the *Allocation Tag* from memory occur.
  - The read is permitted to generate an External abort if a read of data at the same VA would generate an External abort.
- If the read is from a *Canonically Tagged* memory region:
  - For a translation regime where stage 1 translation supports two VA ranges, the read returns the value:
    - `0b0000`, if the VA, bit[55], is 0.
    - `0b1111`, if the VA, bit[55], is 1.
  - For a translation regime where stage 1 translation supports a single VA range, the read returns the value `0b0000`.
  - All effects of the access other than the read of the *Allocation Tag* from memory occur.
  - The read is permitted to generate an External abort if a read of data at the same VA would generate an External abort.

R_{VBFYY} For a write of an *Allocation Tag* all of the following are true:

- If the write is to an *Untagged* memory region:
  - The write does not modify the *Allocation Tag* in memory.
  - All effects of the access other than the write of the *Allocation Tag* to memory occur.
- If the write is to a *Canonically Tagged* memory region, the write generates a stage 1 Permission fault with ESR_ELx.TnD set to 1, if the stage 1 of translation is enabled.

D10.3.1 Interactions with Device memory

I_{NLXDF} If a memory region has the Device memory type, the memory region is not *Tagged*.

R_{WHXBY} A write of data by an STZGM instruction to any type of Device memory is CONSTRAINED UNPREDICTABLE between:

- Zeroing the data at the specified locations.
- Generating an Alignment Fault determined by the memory type.

R_{JMXVK} For a DC GVA instruction to any type of Device memory it is CONSTRAINED UNPREDICTABLE whether the instruction generates an Alignment Fault determined by the memory type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.4 Tag checking

D_{XPMJF} All statements in this section require implementation of FEAT_MTE2.

D10.4.1 Tag Checked memory accesses

R_{CPKNZ} A Memory Read or Write Effect generated by an instruction is one of:

- *Tag Unchecked*.
- *Tag Checked*.

R_{DRGYL} A Memory Read or Write Effect is *Tag Unchecked* due to any of the following, otherwise it is *Tag Checked*:

- The Memory Effect is an Explicit Memory Effect, and is:
  - To an *Untagged* memory region.
  - By an instruction that explicitly loads or stores an *Allocation Tag*.
  - Due to a cache maintenance operation by VA other than DC ZVA.
  - By a prefetch instruction.
  - Due to a load or store relative to VNCR_EL2 if FEAT_NV2 is implemented.
  - Generated by the Trace Buffer Unit if FEAT_TRBE is implemented.
  - Generated by the Statistical Profiling Unit if SPE is implemented.
  - A fetch of GPT information if FEAT_RME is implemented.
  - A Guarded Control Stack data access, if FEAT_GCS is implemented.
  - A Memory Read Effect generated by an instruction for which Store-only Tag checking is enabled, if FEAT_MTE_STORE_ONLY is implemented.
  - By an instruction that uses any of the following addressing modes:
    - A base register only, where SP is the base register.
    - A base register plus immediate offset addressing form, where SP is the base register.
    - Literal (PC-relative).
- The Memory Effect is not an Explicit Memory Effect.
- The Effective value of PSTATE.TCO is 1.
- Logical Address Tagging is disabled for the VA range being accessed. See Logical Address Tagging for more information.
- Irrespective of whether the stage 1 address translation for the ELx translation regime is enabled or not, the *Logical Address Tag* has the value `0b0000` and either of the following conditions apply:
  - If the stage 1 translation supports a single VA range, TCR_ELx.TCMA is 1.
  - If the stage 1 translation supports two VA ranges, TCR_ELx.TCMA0 is 1 and the access is to the lower address range.
- Irrespective of whether the stage 1 address translation for the ELx translation regime is enabled or not, when all of the following conditions apply:
  - The *Logical Address Tag* has the value `0b1111`.
  - If the stage 1 translation supports two VA ranges, TCR_ELx.TCMA1 is 1 and the access is to the upper address range.

R_{MCGNZ} For the purpose of determining if a memory access is *Tag Checked*, an unprivileged access generated by an unprivileged memory access instruction is treated as if the access is at EL0.

R_{KVCNS} When executing a *Tag Checked* Store-Exclusive instruction, that if *Tag Unchecked* would not perform the store, and would return a status value of 1, it is CONSTRAINED UNPREDICTABLE whether the instruction generates either:

- A *Tag Checked* memory access.
- A *Tag Unchecked* memory access.

R_{BGGMD} If FEAT_SME is implemented, it is IMPLEMENTATION DEFINED whether memory accesses due to the following instructions are *Tag Checked*:

- SME, SVE, and SIMD&FP loads and stores that are executed when the PE is in Streaming SVE mode.
- SME LDR (array vector) and STR (array vector).
- SME2 LDR (table) and STR (table).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HRQCL} Memory accesses in Debug state follow the same rules as memory accesses in Non-Debug state for the purposes of determining if a memory access is *Tag Checked* or *Tag Unchecked*.

I_{PVCHY} Any MMU faults generated by a read of an *Allocation Tag* for a *Tag Checked* memory access are not ordered with respect to any MMU faults generated by the memory access.

R_{ZHPVP} If FEAT_MTE_STORE_ONLY is implemented, Store-only Tag Checking can be configured by the SCTLR_ELx{TCSO, TSCO0} controls.

R_{MNHCZ} For a Compare and swap instruction, if the compare fails, a Memory Write Effect by the instruction is permitted to be treated as *Tag Unchecked*.

R_{MLCNN} If a *Tag Check Fault* is configured to have no effect on a PE due to SCTLR_ELx.{TCF, TCF0}, whether a *Tag Checked* access is observed in performance monitors and statistical profiling is IMPLEMENTATION DEFINED.

D10.4.2 Tag Check operations

R_{FDNHC} When a *Tag Checked* memory access to a memory region occurs:

- If the memory region is *Tagged*, a *Tag Check operation* is performed.
- If the memory region is *Canonically Tagged*, a *Canonical Tag Check operation* is performed.
- If the memory region is *Untagged*, no Tag Checking occurs.

R_{QTQDK} Bits [59:56] of a 64-bit VA used for a memory access are a *Logical Address Tag*.
For more information, see Logical Address Tagging.

R_{BJWLM} A *Tag Check operation* at a VA:

- Performs an implicit read of an *Allocation Tag* at the VA aligned to the *Tag Granule*.
- Compares the *Allocation Tag* with the *Logical Address Tag*.
- Generates a *Tag Check Fault* if the two tags are not equal.

R_{VJMHZ} A *Canonical Tag Check operation* at a VA generates a *Tag Check Fault* if:

- For a translation regime where stage 1 translation supports two VA ranges the *Logical Address Tag* does not have the value:
  - `0b0000`, if the VA, bit[55], is 0.
  - `0b1111`, if the VA, bit[55], is 1.
- For a translation regime where stage 1 translation supports a single VA range, the *Logical Address Tag* does not have the value `0b0000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.5 Allocation Tag Access controls

IGJZWR *Allocation Tag Access* is used for all of the following:

- Determining access to *Allocation Tags* in memory.
- Enabling *Tag Checking* of memory accesses.
- By instructions inserting *Logical Address Tags* into addresses.

RYLWMV The following table shows the System register controls that are used to enable *Allocation Tag Access*:

Table D10-1 Allocation Tag Access controls

Exception level	System register controls
EL3	SCTLR_EL3.ATA
EL2	SCR_EL3.ATA, SCTLR_EL2.ATA
EL1	SCR_EL3.ATA, HCR_EL2.ATA, SCTLR_EL1.ATA
EL0	SCR_EL3.ATA, HCR_EL2.ATA, SCTLR_ELx.ATA0

RCWPLR When executed at an Exception level where *Allocation Tag Access* is disabled, instructions that insert *Logical Address Tags* into addresses treat the *Allocation Tag* used to generate the *Logical Address Tag* as zero.

SYHLJC Arm recommends that when software requires access to *Allocation Tags* in a context but *Tag Checking* is not required, the SCTLR_ELx.{TCF, TCF0} field affecting that context is set to `0b00`. For more information, see Tag Check Faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.6 Physical Tag locations

D_{JTNRZ} All statements in this section require implementation of FEAT_MTE2.

D10.6.1 Accessing Tag locations

R_{FHWSD} A *Tag location* is an *Allocation Tag* associated with an address in the PA space.

R_{FXRDJ} An *Allocation Tag* and a byte at the same PA are separate locations.

R_{VPTKC} An instruction accessing an *Allocation Tag* explicitly, or implicitly for a *Tag Check operation*, accesses a *Tag location*.

R_{XTTLW} An implicit or explicit access to an *Allocation Tag* uses the same translation, and has the same translation effects, as an access to the associated *Tag Granule*.

I_{VYGYV} There is no mechanism to generate an access to an *Allocation Tag* at a PA other than the PA of the start of a *Tag Granule*.

R_{TQGBX} If FEAT_RME is implemented, an access to a *Tag location* is subject to a granule protection check.

R_{FWSBS} If FEAT_MEC is implemented, an access to a *Tag location* uses the same PA space and MECID as an access to the associated *Tag Granule*.

D10.6.2 Allocation Tag Storage

R_{NGLPV} Storage is provided for each *Allocation Tag* associated with a *Tag Granule* of Conventional memory. This is *Allocation Tag storage*.

R_{BCFRK} The result of an access to an *Allocation Tag* where *Allocation Tag storage* is not provided is IMPLEMENTATION DEFINED.

D10.6.2.1 Access to Allocation Tag storage at data locations

D_{RQMQH} The rules in this section only apply if *Allocation Tag storage* can be accessed using a data memory effect at a separate PA.

R_{DYBTN} It is IMPLEMENTATION DEFINED whether *Allocation Tag storage* can be accessed using a data memory effect at a separate PA.

I_{RNGQD} It is not architecturally required for an access to *Allocation Tag storage* at a *Tag location* to be coherent with an access to the same *Allocation Tag storage* at a data memory location.

R_{JMJVY} An access to *Allocation Tag storage* at a *Tag location* can be made visible to an access to that same *Allocation Tag storage* at a data location, and vice versa, by the use of cache maintenance operations.

R_{LMRBP} If FEAT_RME is implemented, *Allocation Tag storage* that can be accessed at a data memory location is only accessible at a data memory location in the Root PA space.

R_{TJBLR} If FEAT_MEC is implemented, if *Allocation Tag storage* is accessible at a data memory location, that data location is only accessible in the Root PA space, using the default MECID of zero.

I_{VLYLK} If FEAT_MEC is implemented, encryption of *Allocation Tags* in main memory is an optional defense in depth capability for mitigating attempts to read or corrupt tags.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.6.3 Caching of Allocation Tags

R_{PMQPF} When data is evicted from a cache entry at a cache level, the evicted data can overwrite data in memory that has been written by another observer if either, or any of the following are true:

- The data has been written by an Observer in the Shareability domain of that memory location, where the maximum size of the memory that can be overwritten is defined by the Cache Write-Back Granule in CTR_EL0.
- The associated *Allocation Tags* have been written to by an Observer in the Shareability domain of that memory location, where the maximum size of the memory that can be overwritten is defined by the Cache Write-Back Granule in CTR_EL0.

R_{XPPBK} If *Allocation Tag storage* is accessible at a data memory location, when *Allocation Tags* are evicted from a cache entry at a cache level, the evicted *Allocation Tags* can overwrite *Allocation Tags* in memory that have been written by another observer if the *Allocation Tags* associated with memory within an address range of the size of the Cache Write-Back Granule, and are aligned to that size, have been written by an observer in the Shareability domain of that memory location.

R_{FYDLC} If an implementation can overwrite *Allocation Tags* in memory that have been written by another Observer, where the *Allocation Tags* have not been written by an Observer in the Shareability domain of that memory location, then a cache maintenance operation which:

- Cleans data from a cache level must also clean the associated *Allocation Tags*.
- Invalidates, or cleans and invalidates data from a cache level, must also clean and invalidate the associated *Allocation Tags*.

R_{CKJYV} If a memory region is *Untagged*, a data cache invalidation operation that would invalidate *Allocation Tags* in that memory region cleans and invalidates the *Allocation Tags*.

I_{VNMBR} For more information, see:

- Ordering requirements defined by the formal concurrency model.
- A64 System instructions for cache maintenance.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.7 Tag Check Faults

RWWXZR A *Tag Check Fault* can be configured using SCTLR_ELx.{TCF, TCF0} to have one of the following behaviors:

- Have no effect on the PE.
- Generate a synchronous *Tag Check Fault* exception. For more information see Synchronous exception types.
- If FEAT_MTE_ASYNC is implemented, asynchronously set a bit in TFSR_ELx or TFSRE0_EL1.
- If FEAT_MTE3 is implemented:
  - On a write, asynchronously set a bit in TFSR_ELx or TFSRE0_EL1.
  - On a read, generate a synchronous *Tag Check Fault* exception.

RFYYFC For the purpose of determining *Tag Check Fault* handling, an unprivileged access generated by an unprivileged memory access instruction is treated as if the access is at EL0.

RMDVXM If a *Tag Check Fault* is not configured to cause a synchronous exception, then there is no effect on either:

- The data access, that is the load or store completes unless another exception is taken.
- Any of the side effects caused by the completion of the data access.

IYHCSP A *Tag Check Fault* that is generated from a *Canonical Tag Check operation* is handled the same way as a *Tag Check Fault* generated from a *Tag Check operation*.

IRBPTM If FEAT_SME is implemented, an implementation is only expected to perform *Tag Check operations* when the PE is in Streaming SVE mode if there is a similar performance impact relative to *Tag Check operations* performed due to SVE and SIMD&FP loads and stores that are executed when the PE is not in Streaming SVE mode.

IWTXPP If FEAT_LS64WB is implemented and an LD64B or ST64B instruction accesses Inner and Outer Write-Back Cacheable, Shareable, Tagged memory, then because the instruction accesses four Tag Granules, it is possible that up to four Tag Check Faults are generated. If one or more Tag Check Faults are generated, the 64-byte atomic access does not occur and the address recorded in the FAR_ELx is the lowest faulting address.

D10.7.1 Asynchronous Tag Check Faults

DVVJZM All statements in this section require implementation of FEAT_MTE_ASYNC.

IRQYHY TFSR_ELx or TFSRE0_EL1 are the Tag Fault Status Registers.

RRJMFQ A Tag Fault Status Register is unchanged by a *Tag Check operation* that does not generate an asynchronous *Tag Check Fault*.

RJNNZS An implicit write to a Tag Fault Status Register accessible at ELx that is caused by an asynchronous *Tag Check Fault* is synchronized by any of the following:

- An exception entry to ELx if SCTLR_ELx.ITFSB is 1.
- A DSB instruction with the LD qualifier, or neither the LD nor ST qualifier, at ELx in program order, after the instruction causing the *Tag Check Fault*.

RSCVQH If an instruction that accesses memory generates both a synchronous Data Abort and an asynchronous *Tag Check Fault*, and the value of the bit written to the Tag Fault Status Register is 0, then the bit in the Tag Fault Status Register becomes UNKNOWN.

RTJDMP If an asynchronous *Tag Check Fault* has not been observed in a Tag Fault Status Register, and has not been synchronized by a DSB or by the effects of SCTLR_ELx.ITSB, then when a write to a System register affecting *Tag Check Fault* handling occurs, the effect of the *Tag Check Fault* is CONSTRAINED UNPREDICTABLE between the effect of:

- The old value of the System register.
- The new value of the System register.

RCFWVW If SVE is implemented and a load of an element in an SVE Non-fault or SVE First-fault load instruction causes an asynchronous *Tag Check Fault*:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A bit in a Tag Fault Status Register is not set if the element is not the *First active element* in an SVE First-fault load instruction or any element of an SVE Non-fault load instruction.
- A bit in a Tag Fault Status Register is set if the element is the *First active element* of an SVE First-fault load instruction.

R_{KSKCL} If SVE is implemented, it is CONSTRAINED UNPREDICTABLE whether the FFR element associated with the read of an *Active element* in an SVE Non-fault load, or an *Active element* which is not the *First active element* in an SVE First-fault load, R_2, to location X, is set to FALSE if all of the following are true:

- *Tag Check Faults* are configured as asynchronous for both reads and writes.
- A read or write RW_1 to location Y causes a *Tag Check Fault*.
- A read to location X causes a *Tag Check Fault*.
- *Tag Check Faults* for locations X and Y are reported in the same status bit, either:
  - TFSR_ELx.TFy.
  - TFSRE0_EL1.TFy.
- RW_1 is in program order before R_2, or is the *First active element* in the SVE First-fault load instruction causing R_2.
- There are no other faults caused by R_2 that are reported in FFR.
- There is not a DSB and an explicit write of 0 to that status bit appearing in program order between the instruction causing RW_1 and the instruction causing R_2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D11
The Guarded Control Stack

This chapter describes the Guarded Control Stack Extension. It contains the following sections:

- Introduction.
- The Guarded Control Stack.
- Procedure returns.
- Exception returns.
- Stage 1 permission model.
- Stage 2 Permission model.
- Guarded Control Stack switching.
- GCS record for software use.
- Guarded Control Stack exceptions.
- Guarded Control Stack data accesses.
- Detecting when FEAT_GCS is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.1 Introduction

D11.1.1 Protection for return addresses

INTVZJ *Return oriented programming* (ROP) is a common method used by attackers to execute code in a way that was not intended and to bypass security defenses. ROP techniques involve changing the return address of procedure calls to cause execution to change to a different path. Since the return address of procedures is often stored on the stack, once an attacker has basic write capabilities to data memory, it has the ability to change the return addresses. Technologies such as Pointer Authentication help to significantly reduce the scope of an attack and the scenarios in which attacks are possible.

GHMVDQ To further improve the integrity of return addresses, the Guarded Control Stack provides means to securely store return addresses such that they are immutable, and provides means to check a return address has not been compromised.

D11.1.2 Call stack recording

IZHHFX A history of the call stack provides useful information when debugging or optimizing a program. Typically, the call stack is collected at important points by unwinding the stack to collect the return address stored in each frame. This unwinding procedure can be intrusive or impractical, due to the need to analyze large amounts of memory and because the stack frames are variable in size, although some of this is mitigated by the use of frame pointers. Providing the call stack in a format which is simpler to extract at run time can substantially reduce the cost of collecting the call stack, and make new usage models possible.

GHXYWV To reduce the costs of collecting the call stack information, the Guarded Control Stack provides means to automatically collect return addresses in easily accessible memory.

D11.1.3 Overview

IVCTSV The Guarded Control Stack protects against use of procedure return instructions to return anywhere other than the return address created when the procedure was called.

ICQCMQ Optionally, the Guarded Control Stack protection of procedure return instructions does not require the instruction sequences for function calls, prologues, or epilogues to be modified.

IDMTVT The Guarded Control Stack provides protection against using exception return instructions to return anywhere other than the intended return address, for exceptions taken within the same Exception level.

IHPDGB The Guarded Control Stack provides the ability to capture the call stack for multiple concurrent applications.

IYYFJV The contents of the Guarded Control Stack are immutable to software running at the same Exception level, unless explicitly permitted by a higher Exception level.

ITGPWS The contents of the Guarded Control Stack are readable to software running at the same Exception level.

IJGNWV The management of the Guarded Control Stack does not require a call to privileged code when using various performance critical user space software constructs which manipulate the main stack or switch between multiple stacks. Such software constructs include soft exceptions, setjmp/longjmp, and software threads.

IDLBKT The Guarded Control Stack supports protection and recording in AArch64 state.

IJXYXZ The Guarded Control Stack is not supported in AArch32 state.

ILFRDH The Guarded Control Stack supports capturing the call stack for kernel and user space software independently.

IKLBVV When switching between stacks within an Exception level, FEAT_GCS provides the ability to switch between different Guarded Control Stacks without affecting the immutability of any of the Guarded Control Stacks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.2 The Guarded Control Stack

G_{THDMB} FEAT_GCS allows software to use a separate and protected stack for storing control flow information.

D_{KSDTT} The GCS pointer registers, GCSPR_ELx, are GCSPR_EL0, GCSPR_EL1, GCSPR_EL2, GCSPR_EL3.

R_{WYQWD} While executing at ELx, the Guarded Control Stack uses GCSPR_ELx as the current GCS pointer register.

R_{WCVQY} All GCSPR_ELx are 64 bits wide and hold a virtual address in the current translation regime.

D_{DWRNV} Each entry in the Guarded Control Stack is called a GCS record.

D_{NKMHW} A GCS record is one of the following:

- A GCS procedure return record.
- A GCS exception return record.
- A GCS cap record, which is one of the following:
  - Valid GCS cap record.
  - In-progress GCS cap record.
- A GCS software record.

R_{KDDMV} The size of a GCS procedure return record is 8 bytes.

R_{DNKSM} Bits [1:0] of a GCS procedure return record are set to `0b00`.

R_{KYFJK} The size of a GCS exception return record is 32 bytes.

R_{ZBFYT} Bits [63:0] of the lowest-addressed doubleword of a GCS exception return record are set to `0x0000_0000_0000_0009`.

R_{VNZWB} The size of a GCS cap record is 8 bytes.

R_{QFCRR} Bits [11:0] of a Valid GCS cap record are set to `0x001`.

R_{NCZGL} Bits [2:0] of a In-progress GCS cap record are set to `0b101`.

R_{LYPBD} The size of a GCS software record is 8 bytes.

R_{ZXQBN} Bits [11:0] of a GCS software record are set to `0x011` and bits [63:12] can take any value..

D11.2.1 Enabling the Guarded Control Stack

I_{RNKDG} FEAT_GCS augments existing procedure call and return instructions to additionally push and pop to the Guarded Control Stack.

R_{BYVYL} The Guarded Control Stack at ELx is PCR Selected if GCSCR_ELx.PCRSEL is 1, where x is 1 or greater.
The Guarded Control Stack at EL0 is PCR Selected if GCSCRE0_EL1.PCRSEL is 1.

R_{KJHKJ} The Guarded Control Stack at ELx is PCR Enabled if the Guarded Control Stack is PCR Selected and GCS Enabled at ELx.

R_{WFTRC} The Guarded Control Stack at ELx is GCS Enabled if all of the following are true:

- The PE is executing in AArch64 state.
- The Guarded Control Stack at ELx is PCR Selected.
- At EL0 and EL1, any of the following are true:
  - EL2 is not implemented or disabled in the current Security state.
  - The Effective value of HCR_EL2.{E2H,TGE} is {1, 1}.
  - SCR_EL3.HXEn is 1 and HCRX_EL2.GCSEn is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- At EL0, EL1, and EL2, any of the following are true:
  - EL3 is not implemented.
  - SCR_EL3.GCSEn is 1.

R_{GCVKH} The Guarded Control Stack at ELx is GCS Disabled if the Guarded Control Stack is not GCS Enabled at ELx.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.3 Procedure returns

G_{BSBZN} FEAT_GCS provides means to protect procedure return addresses.

G_{QLGCM} FEAT_GCS provides means to record call stacks.

R_{MSYQB} A procedure return address push operation performs the following:

1. Stores a doubleword to the virtual address defined by the current GCS pointer register minus the size of a GCS procedure return record.
2. Decrements the current GCS pointer register by the size of a GCS procedure return record.

R_{LWRMC} A procedure return address pop and compare operation performs the following:

1. Loads a doubleword from the virtual address defined by the current GCS pointer register.
2. The value in the record is compared with the value in the branch target register:
   - If the values are the same, increments the current GCS pointer register by the size of a GCS procedure return record.
   - If the values are not the same, a GCS Data Check exception is generated.

R_{BMMYF} A procedure return address pop operation performs the following:

1. Loads a doubleword from the virtual address defined by the current GCS pointer register.
2. Increments the current GCS pointer register by the size of a GCS procedure return record.

R_{QHNWF} The Guarded Control Stack at ELx has Return value checking enabled if GCSCR_ELx.RVCHKEN is 1. x is any of 1, 2, 3.
The Guarded Control Stack at EL0 has Return value checking enabled if GCSCRE0_EL1.RVCHKEN is 1.

R_{ZNPXD} The Guarded Control Stack at ELx has Return value checking disabled at ELx if the Guarded Control Stack at ELx does not have Return value checking enabled.

I_{NLMNQ} If the Guarded Control Stack at ELx has Return value checking disabled, this means the address present in the target register of a procedure return instruction is ignored and the top entry on the Guarded Control Stack is popped and used as the target address.
For example, for a `RET` instruction that uses the LR as the intended target address register, the value in LR is ignored.

I_{GNXCT} Irrespective of whether address tagging for instruction addresses is enabled or not, all 64 bits are compared in the compare operation of the procedure return address pop and compare operation.

I_{XFKQN} Guarded Control Stacks are defined as full-descending stacks and there are no controls provided to operate the Guarded Control Stacks as empty-descending stacks or ascending stacks.

D_{VKRDG} Each Memory Write effect and Memory Read effect generated by the following instructions is referred to as a Guarded Control Stack data access:

- Branch with link instructions.
- Procedure return instructions.
- `GCSPOPM`.
- `GCSSS1`.
- `GCSSS2`.
- `GCSPUSHM`.
- `GCSPUSHX`.
- `GCSPOPX`.
- `GCSPOPCX`.
- `GCSSTR`.
- `GCSSTTR`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.3.1 Procedure call and return instructions

G_{PYLXG} FEAT_GCS provides means to protect procedure return addresses without recompilation of functions or increasing the code footprint of functions.

R_{MXSDP} On executing any branch with link instruction, before branching to the target address, if the Guarded Control Stack is PCR Enabled at the current Exception level then a procedure return address push operation is generated. The data value for the store operation of the procedure return address push operation is the value of LR which is created by the branch with link instruction.

R_{MMFFJ} The branch with link instructions are defined as:

- `BL`.
- `BLR`.
- `BLRAA, BLRAAZ, BLRAB, BLRABZ`.

R_{QBJCD} On executing a procedure return instruction, before branching to the target address, if the Guarded Control Stack at the current Exception level is PCR Enabled, one of the following operation is generated:

- If the Guarded Control Stack at the current Exception level has Return value checking enabled, then a procedure return address pop and compare operation is generated. The procedure return address pop and compare operation has a lower priority than any authentication operation performed by the same instruction.
- If the Guarded Control Stack at the current Exception level has Return value checking disabled, then a procedure return address pop operation is generated. The procedure return address pop operation has lower priority than any authentication operation performed by the same instruction. The popped value is used as the target address for branching to the next instruction.

R_{MRQPD} The procedure return instructions are defined as:

- `RET`.
- `RETAA, RETAB`.
- `RETAASPPC, RETABSPPC`.
- `RETAASPPCR, RETABSPPCR`.

D11.3.2 Management of procedure return records

G_{TSLPS} FEAT_GCS provides means for software to add a GCS procedure return record to a Guarded Control Stack.

I_{MLRYX} The `GCSPUSHM` instruction is provided to enable software to add a doubleword in the Guarded Control Stack.

I_{PYRNS} The `GCSSTR` and the `GCSSTTR` instructions are provided to enable software to modify a doubleword in the Guarded Control Stack.

I_{ZBPNC} The `GCSPUSHM` instruction does not sign extend the value that is being stored. This is to maintain consistency with the `GCSSTR` instruction behavior.

I_{PPGKD} The `GCSPOPM` instruction is provided to adjust the current GCS pointer register. This instruction can be used in places where there is no return from subroutine instruction executed for a preceding branch with link instruction.

R_{QHJBM} While the Guarded Control Stack is GCS Enabled at the current Exception level, the `GCSPOPM` instruction performs the following operations:

1. Load a doubleword from the address pointed by the current GCS pointer register.
2. Check the [1:0] bits of the loaded value:

- If the value does not equal `0b00`, a GCS Data Check exception is taken.
- If the value equals `0b00`, the loaded value is written to the `Xn` register defined by `GCSPOPM` and the current GCS pointer register is incremented by the size of a GCS procedure return record.

R_{SXVRB} When the Guarded Control Stack is GCS Disabled at the current Exception level, the `GCSPOPM` instruction does not perform the load operation or the update to current GCS pointer register and executes as a `NOP`.

I_{QLKKL} Arm strongly recommends software uses guard pages between Guarded Control Stacks to help detecting the following:

- Malicious attacks that use `GCSPOPM` instruction to subvert the GCSPR_ELx value to an attacker defined value.
- An underflow or an overflow of the Guarded Control Stack for any other reasons.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.4 Exception returns

I_{TRBJS} When an exception is taken, information regarding where the exception was taken from is captured in various System registers, and is subsequently used to return from the exception. Such information is usually saved to memory and is therefore potentially vulnerable to a variety of attacks which could modify the return state. Given that exceptions can be taken from almost any location, exception returns are usually able to return to any location and therefore are quite powerful. Ensuring the exception return state is protected against unwanted modification provides additional protection against using exception returns to branch to any location.

G_{MDCFD} FEAT_GCS provides means to protect exception return state from unwanted modification, for exceptions within the same Exception level.

G_{HVKNT} FEAT_GCS uses the protection mechanisms provided by the Guarded Control Stack to protect exception return state, in addition to procedure return state.

G_{VQNZK} FEAT_GCS provides protection against mis-use of exception return code sequences.

D11.4.1 Pushing and popping exception return state

I_{KLWLQ} A set of instructions is provided to operate on the Guarded Control Stack specifically for exceptions and exception returns, to allow software to add entries and check entries.

- `GCSPUSHX`
  This instruction creates a new GCS exception return record on the Guarded Control Stack containing the values of the ELR_ELx and SPSR_ELx for the current Exception level and the LR value. The intent is that such data is the contents of the ELR_ELx and SPSR_ELx, pushed soon after taking an exception. This instruction is more limited in its usage than `GCSPUSHM`, only allowing the ELR_ELx, SPSR_ELx and LR to be pushed. This instruction also avoids the exception return state being moved to general purpose registers before adding to the Guarded Control Stack, further reducing the attack surface.
- `GCSPOPCX`
  This instruction pops a GCS exception return record from the top of the Guarded Control Stack and compares the values popped with the ELR_ELx and SPSR_ELx for the current Exception level and the LR value. If the values mismatch, a GCS Data Check exception is taken. The intent is that this instruction is used just before an exception return, to validate that the intended exception return state is the same as that which was pushed to the Guarded Control Stack on exception entry.
- `GCSPOPX`
  This instruction pops a GCS exception return record from the top of the Guarded Control Stack. If the record is not a GCS exception return record, a GCS Data Check exception is taken. This instruction is intended to be used to remove a GCS exception return record, when modification of such a record is needed.

S_{SLHZY} Exception handling software can use the `GCSPUSHX` instruction to save the exception return state to a memory location that is protected by the GCS permissions in stage 1 translations.

- A typical preemptible operating system might employ a kernel stack for kernel execution on behalf of a thread. Software can create a Guarded Control Stack for each kernel stack and these instructions can be used to save the exception return state in the Guarded Control Stack that corresponds to the kernel stack of a thread.
- An interrupt handler might have its own stack that is different from the kernel stack of any thread. Software can create a Guarded Control Stack that corresponds to the interrupt stack and these instructions can be used to store the exception return state in that Guarded Control Stack.

I_{CCMMC} PSTATE.EXLOCK is provided to allow software to ensure that the exception return state that is pushed to the Guarded Control Stack is the state of the PE when the exception was taken, and to ensure that once the exception return state is checked before an exception return that the values in ELR_ELx and SPSR_ELx are unable to be changed. When enabled, PSTATE.EXLOCK prevents `MSR` instructions from modifying the state of the ELR_ELx and SPSR_ELx, by causing such instructions to generate an EXLOCK Exception. When enabled, execution of an exception return instruction which targets the current Exception level will check PSTATE.EXLOCK and will set PSTATE.IL to cause an Illegal exception return.

I_{RZZFL} The GCS exception return record pushed by the `GCSPUSHX` instruction contains a doubleword with the value `0x0000_0000_0000_0009`. This value allows a GCS exception return record to be distinguishable from a GCS procedure return record, and allows the PE to ensure that an exception return does not use a procedure return address or a procedure return does not use an exception return address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When a GCSPOPCX or GCSPOPX instruction is attempted where the top record on the Guarded Control Stack is a GCS procedure return record, the GCSPOPCX or GCSPOPX instruction detects the value in the lowest-addressed doubleword is not 0x0000_0000_0000_0009, and takes a GCS Data Check exception.
- When a procedure return instruction is attempted where the top record on the Guarded Control Stack is not a GCS procedure return record and therefore the bits [1:0] of the doubleword are not 0b00, the procedure return will return to this location then take a PC Alignment exception.
- When a GCSPOPM instruction is attempted where the top record on the Guarded Control Stack is not a GCS procedure return record and therefore bits [1:0] of the value popped are not 0b00, the GCSPOPM instruction takes a GCS Data Check exception.

R_{TNHNB} A GCSPUSHX instruction executed at ELx, where x is 1 or greater:

- Performs a GCS exception push operation when all of the following are true:
  - The Guarded Control Stack at ELx is GCS Enabled.
  - The Effective value of GCSCR_ELx.EXLOCKEN is 0 or PSTATE.EXLOCK is 1.
- Generates an EXLOCK Exception when all of the following are true:
  - The Guarded Control Stack at ELx is GCS Enabled.
  - The Effective value of GCSCR_ELx.EXLOCKEN is 1 and PSTATE.EXLOCK is 0.
- Performs no operation if neither a GCS exception push operation nor an EXLOCK Exception is triggered.

R_{TMRMG} Execution of GCSPUSHX at EL0 is UNDEFINED.

R_{GWJCX} A GCS exception push operation:

- Stores a GCS exception return record at the virtual address defined by the current GCS pointer register minus 32 bytes, consisting of:
  - The value of the LR, stored to the highest addressed doubleword of the GCS exception return record.
  - The value of the SPSR_ELx for the current Exception level, stored to the second-highest addressed doubleword of the GCS exception return record.
  - The value of the ELR_ELx for the current Exception level, stored to the third-highest addressed doubleword of the GCS exception return record.
  - A doubleword with the value 0x0000_0000_0000_0009, stored to the lowest addressed doubleword of the GCS exception return record.
- Decrements the current GCS pointer register by the size of a GCS exception return record.
- Sets PSTATE.EXLOCK to 0.

R_{JTRFY} A GCSPOPCX instruction executed at ELx, where x is 1 or greater:

- Performs a GCS exception pop and compare operation when all of the following are true
  - Guarded Control Stacks are GCS Enabled at ELx.
  - The Effective value of GCSCR_ELx.EXLOCKEN is 0 or PSTATE.EXLOCK is 0.
- Generates an EXLOCK Exception when all of the following are true:
  - Guarded Control Stacks are GCS Enabled at ELx.
  - The Effective value of GCSCR_ELx.EXLOCKEN is 1 and PSTATE.EXLOCK is 1.
- Performs no operation if neither a GCS exception pop and compare operation nor an EXLOCK Exception is triggered.

R_{JYSCF} Execution of GCSPOPCX at EL0 is UNDEFINED.

R_{PCQXX} A GCS exception pop and compare operation:

- Loads four doublewords of a GCS exception return record from the virtual address defined by the current GCS pointer register. Each access is single-copy atomic at doubleword granularity.
- If the value loaded from the lowest addressed doubleword of the GCS exception return record is not 0x0000_0000_0000_0009 then a GCS Data Check exception is taken.
- Compares the value of LR with the value loaded from the highest addressed doubleword of the GCS exception return record, and takes a GCS Data Check exception if the values are not equal.
- Compares the value of the SPSR_ELx for the current Exception level with the value loaded from the second-highest addressed doubleword of the GCS exception return record, and takes a GCS Data Check exception if the values are not equal.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Compares the value of the ELR_ELx for the current Exception level with the value loaded from the third-highest addressed doubleword of the GCS exception return record, and takes a GCS Data Check exception if the values are not equal.
- If no exception is taken, increments the current GCS pointer register by the size of a GCS exception return record.
- Sets PSTATE.EXLOCK to the value of the Effective value of GCSCR_ELx.EXLOCKEN.

I_{PTYBX} Irrespective of whether address tagging for instruction addresses is enabled or not, all 64 bits of following registers are compared in the compare operation of the `GCSPOPCX` instruction:

- ELR_ELx.
- LR.

R_{YMVVH} Explicit synchronization is not required for direct writes to ELR_ELx and SPSR_ELx be observable to indirect reads made by the `GCSPUSHX` instruction.

R_{SSMHC} Explicit synchronization is not required for direct writes to ELR_ELx and SPSR_ELx be observable to indirect reads made by the `GCSPOPCX` instruction.

R_{HXXCL} If Guarded Control Stacks are GCS Enabled at ELx, where x is 1 or greater, a `GCSPOPX` instruction executed at ELx performs the following operations:

- Load a GCS exception return record from the virtual address defined by the current GCS pointer register.
- If the value loaded from the lowest addressed doubleword of the GCS exception return record is not `0x0000_0000_0000_0009` then a GCS Data Check exception is taken.
- Otherwise, the current GCS pointer register is incremented by the size of a GCS exception return record.

R_{XHXGJ} Execution of `GCSPOPX` at EL0 is UNDEFINED.

I_{CFFNS} When an MSR instruction would write to the relevant ELR_ELx or SPSR_ELx for the current Exception level ELy, the Effective value of GCSCR_ELy.EXLOCKEN and PSTATE.EXLOCK may prevent the write.

I_{DXBZZ} For more information on exception entry and return, see Legal exception returns from AArch64 state, Illegal exception returns from AArch64 state, and Exception entry.

I_{BGLWD} For more information on FEAT_GCS in Debug state, see About Debug state.

R_{YZBWD} The Effective value of GCSCR_ELx.EXLOCKEN is 1 for all purposes other than returning the value of a direct read of GCSCR_ELx, if all of the following are true:

- GCSCR_ELx.EXLOCKEN is 1.
- The PE is executing in Non-debug state.

R_{WRDNF} The Effective value of GCSCR_ELx.EXLOCKEN is 0 for all purposes other than returning the value of a direct read of GCSCR_ELx, if any of the following are true:

- GCSCR_ELx.EXLOCKEN is 0.
- The PE is executing in Debug state.

Note

While executing in Debug state, ELR_ELx and SPSR_ELx registers are accessible at ELx irrespective of the value of PSTATE.EXLOCK.
Execution of the `GCSPUSHX` instruction in Debug state is unaffected by the value of PSTATE.EXLOCK.

D11.4.2 Using the exception return protection features

S_{MQBXD} Arm notes that the use of `ERETAA, ERETAB` for an exception return is unlikely when `GCSPOPCX` is used to validate the exception return state. This is because a prior `GCSPUSHX` instruction will have pushed an exception return address without a pointer authentication code to the Guarded Control Stack, and a `GCSPOPCX` will need to validate this exception return address without a pointer authentication code. Therefore, the authentication would need to be performed before the `GCSPOPCX` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception handling software needs to be aware of the Guarded Control Stack for scenarios where an exception return does not return to the point where an exception was taken from, since this might require modification of the exception return state held on the Guarded Control Stack.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.5 Stage 1 permission model

G_{FJCSY} FEAT_GCS provides EL1 with controls to govern the behavior of EL0 Guarded Control Stacks when operating in the EL1&0 translation regime, with little or no intervention from Exception levels higher than EL1.

G_{FHRKC} FEAT_GCS provides EL2 with controls to govern the behavior of EL0 Guarded Control Stacks when operating in the EL2&0 translation regime, with little or no intervention from Exception levels higher than EL2.

G_{RDKLS} FEAT_GCS provides EL1 with controls to govern the behavior of EL1 Guarded Control Stacks, while allowing EL2 to opt-in to manage some aspects of EL1 Guarded Control Stacks to provide additional security.

G_{PDKVN} FEAT_GCS provides EL2 with controls to govern the behavior of EL2 Guarded Control Stacks.

G_{LJFBN} FEAT_GCS provides EL3 with controls to govern the behavior of EL3 Guarded Control Stacks.

I_{CZTDM} FEAT_S1PIE is required to be implemented if FEAT_GCS is implemented.

D11.5.1 Stage 1 Base Permissions

I_{MDFXL} The permissions encodings that permit Guarded Control Stack data access can only be configured when stage 1 translation uses the Indirect permission scheme.

I_{BVLMP} For translation regimes that apply to EL0 and a higher Exception level and where a Guarded Control Stack is being used for privileged execution, unprivileged data accesses do not write to such a Guarded Control Stack. For more information, see Stage 1 permissions.

I_{ZZMLZ} For translation regimes that apply to EL0 and a higher Exception level and where a Guarded Control Stack is being used for privileged execution but also accessible by any of unprivileged Data read accesses or unprivileged Instruction accesses, such a Guarded Control Stack is accessible by privileged Guarded Control Stack data access irrespective of the value of PSTATE.PAN. For more information, see Stage 1 permissions.

I_{LZXNK} For the purpose of permission checking, the DC IVAC instruction is considered to produce a privileged explicit data write access other than a Guarded Control Stack data access.

If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether the DC IVAC instruction can generate a Permission fault.

I_{FRYGL} When FEAT_CMOW is implemented and SCTLR_EL1.UCI is 1 and SCTLR_EL1.CMOW is 1, for the purpose of permission checking, the following instructions executed at EL0 are considered as producing an unprivileged explicit data write access other than a Guarded Control Stack data access:

- IC IVAU.
- DC CVAC.
- DC CIGVAC.
- DC CIGDVAC.

If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether the above instructions can generate a Permission fault.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the above instructions can generate a Permission fault.

S_{DBWHS} EL0 Guarded Control Stack pages are expected to use:

- UnprivRead as 1, UnprivWrite as 0, UnprivExecute as 0 and UnprivGCS as 1 in S1UnprivBasePerm.
- PrivRead as 1, PrivWrite as 0 or 1, PrivExecute as 0 and PrivGCS as 0 in S1PrivBasePerm.

EL0 Guarded Control Stack pages are generally accessible from EL1, enabling management from EL1. If PrivWrite is 0, then EL1 software should use the `GCSSTTR` instruction to manage the EL0 Guarded Control Stack.

S_{GRGDT} EL1 Guarded Control Stack pages are expected to use:

- UnprivRead as 0, UnprivWrite as 0, UnprivExecute as 0 and UnprivGCS as 0 in S1UnprivBasePerm.
- PrivRead as 1, PrivWrite as 0, PrivExecute as 0 and PrivGCS as 1 in S1PrivBasePerm.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Management of EL1 Guarded Control Stack pages is possible from EL1 by using the GCSSTR or GCSPUSHM instructions.

SYNDKX In a translation regime that applies to EL0 and a higher Exception level, the higher Exception level software cannot use unprivileged store instructions that perform non-Guarded Control Stack data access to manage the Guarded Control Stack pages of EL0. The higher Exception level software should use privileged store instructions or GCSSTTR instead.

SHKJXY In a translation regime that applies to EL0 and a higher Exception level, the higher Exception level software can make use of GCSSTR and GCSPUSHM instructions to manage the Guarded Control Stack pages of the higher Exception level.

In a translation regime that applies to only one Exception level, software can make use of GCSSTR and GCSPUSHM instructions to manage the Guarded Control Stack pages of the current Exception level.

Note: The GCSPUSHM instruction executes as a NOP when the Guarded Control Stack is Disabled at the current Exception level. The GCSSTR instruction is functional even if the Guarded Control Stack is Disabled at the current Exception level.

SMQRCG When PSTATE.PAN is 1, in a translation regime that applies to EL0 and a higher Exception level, the higher Exception level software can use unprivileged load instructions that perform non-Guarded Control Stack data access to manage the Guarded Control Stack pages of EL0.

Example:

```
;At EL0
BL                ;produce a GCS store operation
SVC #0x0          ;exception to EL1 and PSTATE.PAN is set to 0b1
MRS x0,GCSPR_EL0
GCSB DSYNC
LDTR x1,[x0]      ;read GCS contents of EL0
```

D11.5.2 Security states

IFYTRK A Guarded Control Stack data access generated in Secure state that is marked in the first stage of translation as being Non-secure generates a Permission fault.

IFDCRS A Guarded Control Stack data access generated in Secure state is allowed to access Non-secure memory as determined by Memory access control.

IRSDVG When FEAT_RME is implemented, during execution at EL3, a Guarded Control Stack data access to physical memory other than the Root physical address space causes a stage 1 Permission fault.

IHGYTX If execution is using the Realm EL2 or Realm EL2&0 translation regime, a Guarded Control Stack data access to physical memory other than the Realm physical address space causes a stage 1 Permission fault.

ITHLDW If execution is using the Realm EL1&0 translation regime, a Guarded Control Stack data access is allowed to access physical memory that is not from the Realm physical address space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.6 Stage 2 Permission model

I_{RXBNM} Unless otherwise stated, for the purpose of permission checking in stage 2 translations and the purposes of granule protection checks performed by FEAT_RME, Guarded Control Stack data access are considered as explicit data accesses produced by other load/store instructions.

I_{SWMHZ} While the Effective value of HCR_EL2.{E2H, TGE} is {0, 1}, stage 1 translation is disabled and stage 2 does not have any special permissions for the Guarded Control Stack at EL0. The architecture provides no additional protection for the EL0 Guarded Control Stack while the PE is executing in this configuration.

D11.6.1 Hardening Stage 1 translations

D_{DDFTT} VTCR_EL2.GCSH provides an mechanism to enforce the use of stage 1 translations that are hardened by FEAT_THE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.7 Guarded Control Stack switching

G_{XMPQT} FEAT_GCS provides means to switch between multiple Guarded Control Stacks of an Exception level at the same Exception level, without providing direct write access to the current GCS pointer register.

G_{GTMLR} Arm expects that an individual Guarded Control Stack is used by only one PE at a time, to ensure the stack is not being updated by one PE while being used by another.

I_{FZSVZ} When a Guarded Control Stack is not currently in use, the top entry contains a special value which indicates it is capped. The only way that the Guarded Control Stack can become uncapped is by execution of the `GCSSS1 Xn` and `GCSSS2 Xn` instructions, to switch between two Guarded Control Stacks. These instructions check the cap on the incoming Guarded Control Stack and then add a cap to the outgoing Guarded Control Stack. Since there is no direct write access to the current GCS pointer register, the only way the current GCS pointer register can be made to point to the new Guarded Control Stack is using these instructions, and the checks performed by these instructions ensure that the value loaded is always pointing to a capped Guarded Control Stack. This ensures that any arbitrary location in a Guarded Control Stack cannot be switched to.

Figure D11-1 shows the process of how these instructions perform the switching between 2 Guarded Control Stacks.

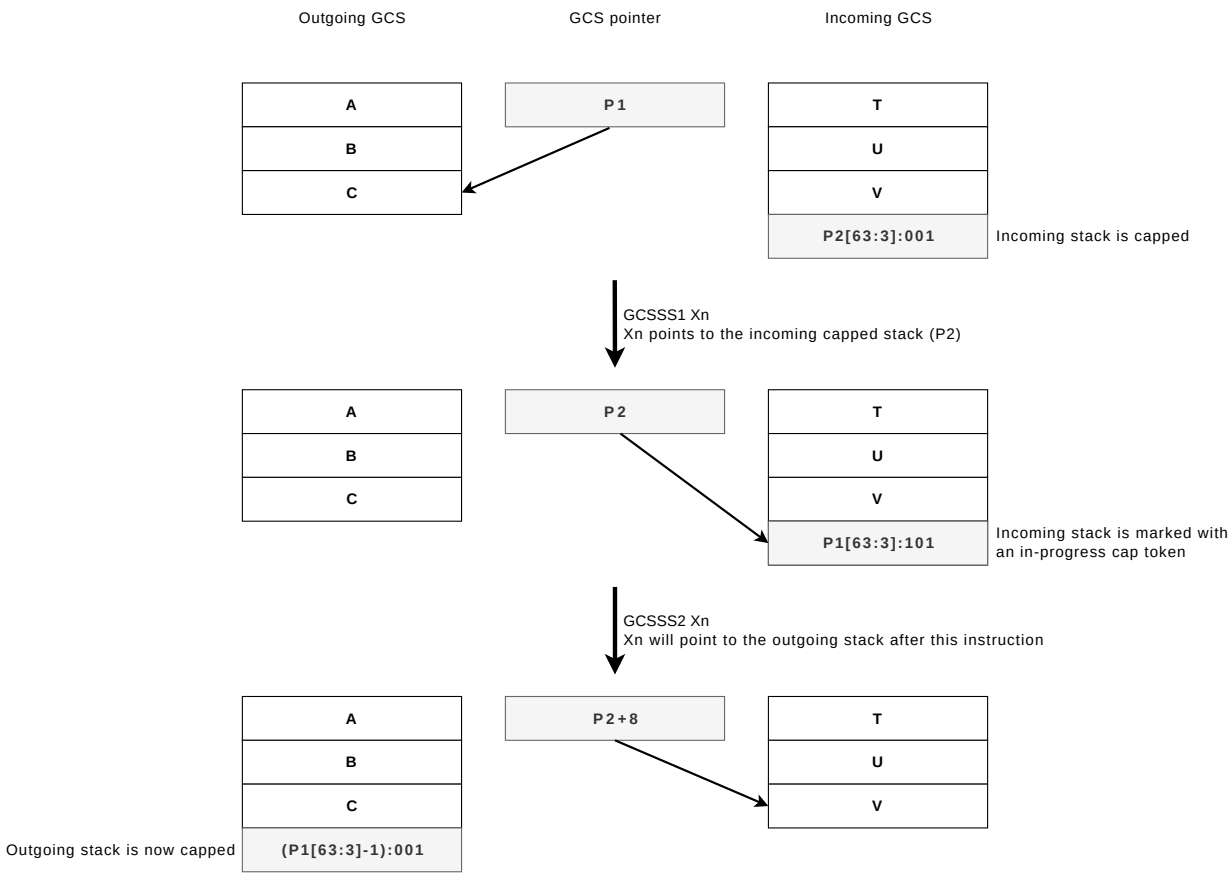

Figure D11-1 Guarded Control Stack switching

I_{GPHKD} The instruction pair `GCSSS1 Xn` and `GCSSS2 Xn` provide means to switch between Guarded Control Stacks of the current Exception level without directly accessing GCSPR_ELx at the same Exception level.

D_{PCHQS} A value of `0b000000000001` in bits [11:0] of a Guarded Control Stack entry is called a Valid cap token.

D_{QJDCV} A GCS cap record is defined as a Valid GCS cap record, if bits [63:12] of the entry value are the same as bits [63:12] of the address where the entry is stored and bits [11:0] contain a Valid cap token.

D_{ZNSQG} A value of `0b101` in bits [2:0] of a Guarded Control Stack entry is called an In-progress cap token.

D_{XGMLV} A GCS cap record is defined as an In-progress GCS cap record, if bits [63:3] of the value are the same as bits [63:3] of the current GCS pointer register value and bits [2:0] contain an In-progress cap token.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{QHSGP} The `GCSSS1 Xn` instruction performs the following functionalities in order:

1. A doubleword is loaded from the Guarded Control Stack that is pointed to by `Xn`.
2. If the loaded value is a Valid GCS cap record:
   - The top entry of the Guarded Control Stack pointed to by `Xn` is overwritten with an In-progress GCS cap record:
     - Bits [63:3] of the In-progress GCS cap record contain bits [63:3] of the current GCS pointer register.
     - Bits [2:0] of the In-progress GCS cap record contain `0b101`.
   - The current GCS pointer register is updated to the following value:
     - Bits [63:3] are set to bits [63:3] of `Xn`.
     - Bits [2:0] are set to `0b000`.
3. If the loaded value is not a Valid GCS cap record, a GCS Data Check exception is generated and no write is required to be performed.
4. All observers in the shareability domain observe the load and store atomically.

R_{PBWVT} The `GCSSS2 Xn` instruction performs the following functionalities in order:

1. A doubleword is loaded from the memory pointed by the current GCS pointer register.
2. If the loaded value contains an In-progress cap token:
   - A Valid GCS cap record is written to the outgoing GCS, with the following parameters:
     - The target address of the write has:
       - Bits [63:3] set to bits [63:3] of the loaded value, minus 1.
       - Bits [2:0] set to `0b000`.
     - The data value written has:
       - Bits [63:12] set to bits [63:12] of the target address.
       - Bits [11:0] set to `0b000000000001`.
   - The current GCS pointer register is incremented by the size of a GCS procedure return record.
   - `Xn` is updated to contain:
     - Bits [63:3] set to bits [63:3] of the loaded value, minus 1.
     - Bits [2:0] are set to `0b000`.
3. If the loaded value does not contain an In-progress cap token, a GCS Data Check exception is generated.

R_{KFMRX} For the purpose of permission checking, and for watchpoints, the `GCSSS1 Xn` instruction is treated as performing both a load and a store, even if the store does not happen.

G_{HMJGM} FEAT_GCS allows software to use address tagging in a Guarded Control Stack Pointer.

I_{MGLTC} While address tagging for data addresses is enabled, if the tag value in the target address register of the `GCSSS1 Xn` instruction is not same as the tag value in the Valid GCS cap record, the comparison as part of the `GCSSS1 Xn` instruction will cause a GCS Data Check exception.

I_{MBHFS} While address tagging for data addresses is enabled, the In-progress GCS cap record stored by the `GCSSS1 Xn` instruction will contain any tag value that is present in the current GCS pointer register.

R_{XWQDB} All restrictions that are applicable to usage of the atomic instructions are also applicable to the usage of the `GCSSS1 Xn` instruction.

I_{DDFMX} For example, the memory types for which it is architecturally guaranteed that the `GCSSS1 Xn` instruction will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

R_{PTCXM} For the purpose of exception prioritization, a GCS Data Check exception from a `GCSSS2 Xn` instruction is considered to be caused by the load that is part of the same `GCSSS2 Xn` instruction.

R_{XTZVS} For the purpose of prioritizing exceptions and debug events, all exceptions and debug events due to load part of a `GCSSS2 Xn` instruction are higher priority than exceptions and debug events due to store part of the same `GCSSS2 Xn` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This is a stricter requirement when compared to most of other instructions that cause multiple single copy atomic accesses where the priority is not architecturally defined. See Prioritization of Synchronous exceptions taken to AArch64 state for more information.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.8 GCS record for software use

I_{FVGLK} A GCS software record value is reserved for software use.

I_{WJZPW} If any of the following instructions attempts to remove the value in a GCS software record then a GCS Data Check exception is generated:

- A procedure return instruction.
- `GCSPOPCX`.
- `GCSPOPM`.
- `GCSPOPX`.
- `GCSSS1`.
- `GCSSS2`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.9 Guarded Control Stack exceptions

D_{WWXWR} FEAT_GCS provides the GCS Data Check exception to report data mismatches in the comparison operations related to the Guarded Control Stack.

I_{QXRVB} For the comparison operations related to the Guarded Control Stack, see R_{PCQXX}, R_{LWRMC}, and R_{QHSGP}.

D_{PRYMV} FEAT_GCS provides the EXLOCK Exception to report any exceptions due to an incompatible value in PSTATE.EXLOCK.

I_{RVKKZ} For a GCS Data Check exception, an EXLOCK Exception, a trap on a `GCSSTR` instruction, or a trap on a `GCSSTTR` instruction, the syndrome information is captured in ESR_ELx.

I_{WLNVC} PMU events EXC_DABORT and EXC_TRAP_DABORT count occurrences of a GCS Data Check exception.

I_{PHRFN} Traps on the `GCSSTR` and `GCSSTTR` instructions are counted by the EXC_UNDEF and EXC_TRAP_OTHER PMU events.

I_{YLMVR} An EXLOCK Exception is counted by the EXC_UNDEF and EXC_TRAP_OTHER PMU events, according to the existing definitions of those PMU events.

I_{MKLKV} For more information on tracing GCS exceptions, see:

- R_{LYGJZ}.
- R_{GZQKS}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.10 Guarded Control Stack data accesses

R_{ZMNSV} A Guarded Control Stack data access that is not aligned to size of a GCS procedure return record generates an Alignment fault.

I_{RYLTG} If a Guarded Control Stack data access causes an Alignment fault, the resulting fault is reported with the ESR_ELx.EC code `0b100100` or `0b100101`.

R_{VDJXL} Unless otherwise specified, a Guarded Control Stack data access is single-copy atomic at 64-bit granularity.

I_{BFHQY} For more information on atomicity, see Requirements for single-copy atomicity.

I_{KNXKG} A Guarded Control Stack data access might clear the state of the local Exclusives monitor if the local monitor is in the Exclusive Access state. For more information on the effects on the local monitor, see Exclusive access instructions and Non-shareable memory locations.

R_{FFJDX} When a Guarded Control Stack data access accesses a memory type that is not one of the following, it is CONSTRAINED UNPREDICTABLE whether:

- A Data Abort for unsupported access, using ESR_ELx.DFSC value `0b110101`, is generated.
  - In this case, it is IMPLEMENTATION DEFINED whether the Data Abort is generated according to the memory type after all stages of translation, or at each stage of translation.
- The access proceeds using the specified memory type.

The memory types are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

S_{JBQRL} If translation or data caches are disabled, then Guarded Control Stack data access use Device or Normal Non-cacheable memory types respectively.

To avoid use of mismatched memory attributes or generation of a Data Abort for an unsupported access, software should use the following sequence when disabling translation or data caches:

1. Disable GCS using GCSCR_ELx and GCSCRE0_EL1.
2. `GCSB`.
3. Disable translation and/or data caches.

R_{KSTSV} For the purpose of Memory Partitioning and Monitoring, a Guarded Control Stack data access is treated in the same way as any other load/store access at that Exception level.

R_{CYLYV} The endianness of a Guarded Control Stack data access is same as the endianness of any other load/store access at that Exception level.

R_{QYMTJ} A Guarded Control Stack data access is checked against watchpoint debug events, similar to any other load/store access.

I_{QJRGN} There are no controls in the watchpoint control registers to select or deselect a Guarded Control Stack data access independently of any other load/store access.

I_{WBHHX} When FEAT_MTE is implemented, a Guarded Control Stack data access is a Tag Unchecked accesses as defined by R_{DRGYL}.

I_{WLNYQ} If a Guarded Control Stack data access causes a synchronous Data Abort exception, the resulting fault is reported as a Data Abort from the current Exception level with the ESR_ELx.EC code `0b100100` or `0b100101`.

I_{JGYZS} If a Guarded Control Stack data access causes a Granule Protection Check exception, the resulting fault is reported as the Granule Protection Check exception. For more information on the ISS encoding, see ISS encoding for a Granule Protection Check exception.

I_{BQPPZ} If a Guarded Control Stack data access causes a watchpoint exception, the resulting fault is reported as a watchpoint exception from the current Exception level with the ESR_ELx.EC code `0b110100` or `0b110101`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{YMCFF} For more information on The Access flag when there is a GCS Data Check exception generated from an instruction, see Hardware management of the Access flag.

I_{KDWGR} When hardware management of the Access flag is enabled and if there is a GCS Data Check exception caused by a `GCSSS2` instruction, it is CONSTRAINED UNPREDICTABLE whether translations for the store part of the `GCSSS2` instruction update the Access flag.

I_{LKVJG} When hardware management of the dirty state mechanism is enabled and if there is a GCS Data Check exception generated from a `GCSSS1` instruction, it is CONSTRAINED UNPREDICTABLE whether translations for Guarded Control Stack data access update the dirty state information.

R_{CQXHQ} When hardware management of the dirty state mechanism is enabled and if there is a GCS Data Check exception generated from a `GCSSS2` instruction, translations for Guarded Control Stack data access do not update the dirty state information.

I_{HGNJQ} For the purposes of generating PMU events, Guarded Control Stack data accesses are:

- Memory-write operations and Memory-read operations.
- Demand accesses.
- Attributable to the Exception level where the instruction that generated the Guarded Control Stack data access was executed.

This means that:

- A Guarded Control Stack data access is counted by events that count Memory-write operations or Memory-read operations. For example, a Guarded Control Stack data access is counted by one or more of MEM_ACCESS, LD_RETIRED, and ST_RETIRED events.
- For the purpose of event filtering, an Overshooting GCS Memory effect is considered as a data access executed at the current Exception level irrespective of the effective privilege of the data access.

I_{VTQSX} If FEAT_SPE is implemented, for information on sampling a Guarded Control Stack data access, see The Statistical Profiling Extension.

I_{NPBWN} An Overshooting GCS Memory effect that is not associated with any instruction is not sampled.

I_{QRDQX} For information on the Events packet in the Statistical Profiling Extension sample record, see Events packet.

D11.10.1 Guarded Control Stack data access behaviors

D_{NFPLW} A GCS Memory Read effect is a Memory Read effect produced by any of the following instructions:

- A procedure return instruction; to read the data addressed by the current GCS pointer plus any applicable offset.
- `GCSPOPM`; to read the data addressed by the current GCS pointer plus any applicable offset.
- `GCSSS2`; to read the data addressed by the current GCS pointer plus any applicable offset.
- `GCSPOPCX`; to read the data addressed by the current GCS pointer plus any applicable offset.
- `GCSPOPX`; to read the data addressed by the current GCS pointer plus any applicable offset.

A GCS Memory Write effect is a Memory Write effect produced by any of the following:

- A branch with link instruction; to write the data addressed by the current GCS pointer plus any applicable offset.
- `GCSSS2`; to write the data addressed by a calculation from the data read by the same `GCSSS2` instruction.
- `GCSPUSHM`; to write the data addressed by the current GCS pointer plus any applicable offset.
- `GCSPUSHX`; to write the data addressed by the current GCS pointer plus any applicable offset.
- `GCSSTR`; to write the data addressed by the Xn|SP operand of the instruction, plus any applicable offset.
- `GCSSTTR`; to write the data addressed by the Xn|SP operand of the instruction, plus any applicable offset.
- An Overshooting GCS Memory effect; to write the data addressed by the current GCS pointer plus any applicable offset.

A GCS Memory effect is one of the following:

- A GCS Memory Read effect.
- A GCS Memory Write effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D_{MYKCM} A GCSSS1 Memory effect is a Memory effect generated by a GCSSS1 instruction, to access the data addressed by the Xt operand. A GCSSS1 Memory effect is one of the following:

- A GCSSS1 Memory Read effect, which occurs when reading the data.
- A GCSSS1 Memory Write effect, which occurs when writing the data.

D_{MVQYZ} A GCSB effect is generated by any of the following instructions:

- GCSB.
- GCSSS2.

A GCSB effect provides the ability to manage the ordering and coherency of GCS Memory effects in relation to other Memory effects.

D_{NHBBP} A Guarded Control Stack data access is one of the following:

- A GCS Memory effect.
- A GCSSS1 Memory effect.

D_{QQLCF} A GCS effect is one of the following:

- GCS Memory effect.
- A GCSSS1 Memory effect.
- A GCSB effect.

D_{CHDJM} A GCSB instruction causes a GCSB effect.

A GCSSS2 instruction causes a GCSB effect.

A GCSB effect provides the ability to manage the ordering and coherency of GCS Memory effects in relation to other Memory effects.

R_{FZRGP} Two effects E1 and E2 to the Same Location follow the same ordering requirements as two Explicit Memory effects to the Same Location if one of the following applies:

- E1 and E2 are both GCS Memory effects.
- E1 is a GCS Memory effect, E2 is an Explicit Memory effect, and there exists a GCSB effect E3 such that E1 appears in program-order before E3, and E3 appears in program order before E2.
- E1 is an Explicit Memory effect, E2 is a GCS Memory effect, and there exists a GCSB effect E3 such that E1 appears in program-ordered before E3, and E3 appears in program order before E2.
- E1 is a GCS Memory effect and E2 is a GCSSS1 Memory effect.
- E1 is a GCSSS1 Memory effect and E2 is a GCS Memory effect.
- E1 and E2 are both GCSSS1 Memory effects.
- E1 is a GCSSS1 Memory effect and E2 is an Explicit Memory effect.
- E1 is an Explicit Memory effect and E2 is a GCSSS1 Memory effect.

R_{TDLVW} A GCS effect E1 is Hardware-required-ordered-before another GCS effect E2 if one of the following applies:

- E1 is a GCSB effect and E2 is a GCS Memory effect. E1 appears in program order before E2.
- E1 is a GCS Memory effect and E2 is a GCSB effect. E1 appears in program order before E2.

R_{ZRZQM} A GCS effect E1 is Barrier-ordered-before an Explicit Memory effect E2, or conversely an Explicit Memory effect E1 is Barrier-ordered-before a GCS effect E2, if the GCS effect is a GCSB or a GCSSS1 Memory effect, E1 appears in program order before E2, and if and only if one of the following applies:

- E1 appears in program order before a DMB FULL effect E3 and E3 appears in program order before E2.
- E1 is the Memory Write effect generated by an atomic instruction with both Acquire and Release semantics.
- E1 is a Memory Read effect (R1) which appears in program order before a DMB LD that appears in program order before E2.
  E1 is a Memory Read effect (R1), except an Implicit Tag Memory Read effect, generated by an instruction with Acquire or AcquirePC semantics.
- E2 is a Memory Write effect (W2) generated by an instruction with Release semantics.
- E1 is a Memory Write effect (W1) which appears in program order before a DMB ST that appears in program order before E2.

R_{YJHMB} GCSSS1 performs a read-modify-write and is therefore subjected to the Atomic-ordered-before ordering requirements and the Atomic axiom.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{BGSJM} GCSSS2 performs a Memory Read effect to access the data addressed by the current GCS pointer, which returns address X, and is Intrinsically-ordered-before a Memory Write effect to X by the same GCSSS2 instruction.

R_{RCNPD} The GCS Memory Write effect performed by a GCSSS2 instruction is Intrinsically-ordered-before the GCSB effect from the same GCSSS2 instruction.

I_{YJSDD} As GCSSS2 causes a GCSB effect, migrating a Guarded Control Stack between PEs does not require an explicit GCSB instruction if the Guarded Control Stack is switched using the GCSSS1 and GCSSS2 instruction pair.

I_{WFDGZ} Entry to Debug state or exit from Debug state is not required to cause a GCSB effect which means that a debugger needs to manually cause such an event should such synchronization be required, for example by executing a GCSB instruction.

G_{CXFHZ} To allow efficient usage of cache lines, FEAT_GCS permits invalidating the dirty cache line contents that are already consumed by the procedure return instructions.

R_{SJPKS} On execution of a GCS Memory effect that is a read access R1 to a Location M and is produced by any of the following instructions, without any exception caused by those instructions, a Memory Write effect is induced to the Location M and the write value is a CONSTRAINED UNPREDICTABLE choice of the following:

- A value that is written to the same Location M, by a GCS Memory effect that is a write access W2 and all of the following are true:
  - W2 is in program order before R1.
  - There is no Memory Write effect W3 to the same Location M, that is not a GCS Memory effect, that exists in Coherence order after W2.
  - There is no write access W4 to the same Location M, and all of the following are true for W4:
    - W4 is a Guarded Control Stack data access.
    - W4 is in program order after W2 and in program order before R1.
    - There is a GCSB effect that appears in program order after W4 and appears in program order before R1.
- A value that is written to the same Location M, by a Memory Write effect W2 that is not a Guarded Control Stack data access and all of the following are true:
  - W2 is in program order before R1.
  - There is no write access W3 to the same Location M, that is not a Guarded Control Stack data access, that exists in Coherence order after W2.
  - There is no write access W4 to the same Location M, and all of the following are true for W4:
    - W4 is a Guarded Control Stack data access.
    - W4 is in program order after W2 and in program order before R1.
    - There is a GCSB effect that appears in program order after W4 and appears in program order before R1.

The induced Memory Write effect is not the same as the one generated by a store instruction or one generated by a Branch with Link instruction. For example, the Memory Write effect:

- Does not cause any exceptions.
- Does not cause the side effects due to the translation.
- Does not affect the state of the local Exclusives monitor.
- Is transparent to translations and watchpoints.

The Memory Write effect is single-copy-atomic at doubleword granularity.

The instructions that can produce the Memory Read effect R1 mentioned above are:

- All procedure return instructions
- GCSPOPM.
- GCSPOPX.
- GCSPOPCX.
- GCSSS2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D11-1

For example, the `LDR` instruction in the following sequence is permitted to read any one of the following values:

- `0x300`, since this is the value of a permitted induced Memory Write effect due to being the value of the most recent Explicit Memory Write effect.
- `0x10C`, since this is the value of a permitted induced Memory Write effect due to being the value of a recent GCS Memory Write effect.
- `0x110`, since this is the value of a permitted induced Memory Write effect due to being the value of a recent GCS Memory Write effect.
- `0x114`, since this is the value of the most recent GCS Memory Write effect.
- `0x000`, which is a permitted Overshooting GCS Memory effect occurring after the final `RET` in the sequence.

```
PC     Instruction
0x0F0   STR             ;store 0x200 to an address 0x1000
0x0F4   GCSB DSYNC
0x0F8   BL              ;store 0xFC to the address 0x1000
0x200   RET             ;load from the address 0x1000
0x0FC   GCSB DSYNC
0x100   STR             ;store 0x300 to the address 0x1000
0x104   GCSB DSYNC      ;make STR visible to subsequent load/stores from
                        ;BL/RET instructions
0x108   BL              ;store 0x10C to the address 0x1000
0x200   RET             ;load from the address 0x1000
0x10C   BL              ;store 0x110 to the address 0x1000
0x200   RET             ;load from the address 0x1000
0x110   BL              ;store 0x114 to the address 0x1000
0x200   RET             ;load from the address 0x1000
0x114   GCSB DSYNC      ;make load/stores from BL/RET instructions visible
                        ;to subsequent LDR instructions
0x118   LDR             ;load from the address 0x1000
```

Note that the values `0x200` and `0xFC` are not permitted in this example, due to the presence of one or more of the `GCSB DSYNC` instructions which prevent these previously stored values being permitted as the value of an induced Memory Write effect that would be visible to the `LDR` instruction. In particular, the `GCSB DSYNC` instruction at PC `0x104` bounds the limits of which values are permitted.

I_{XFLPD} R_{SJPKS} provides implementation flexibility with respect to partial cache line transfer by permitting zeros to be written to entries below the current value of the GCS pointer.

R_{GZNKD} An Overshooting GCS Memory effect is defined as the following:

- As a GCS Memory Write effect.
- With a write value of 0.
- As a write to a location M whose virtual address is lower than the current GCS pointer register.

An Overshooting GCS Memory effect S1 is only permitted when all of the following are true:

- When the Guarded Control Stack is GCS Enabled at the current Exception level.
- An Overshooting GCS Memory effect has occurred to all Locations whose virtual address is greater than the virtual address of Location M and less than the current GCS pointer register.
- If S1 was any other GCS Memory effect it would have updated Location M.
- When hardware management of dirty state mechanism is enabled, S1 updates the dirty state information like any other GCS Memory effect.

R_{QQTVL} Any transition of the local Exclusives monitor to the Open Access state caused by the Overshooting GCS Memory effect must not indefinitely delay forward progress of execution.

S_{YTYHX} It is anticipated that software uses guard pages between the Guarded Control Stacks of any two threads so that the contents of a Guarded Control Stack that is not currently in use are not overwritten by speculative accesses.

S_{QDQQQ} The Overshooting GCS Memory effect can cause potential race conditions when a `GCSSTR` or `GCSSTTR` instruction accesses an address that is less than the address present in the applicable Guarded Control Stack register. Software is expected to maintain appropriate values in the applicable Guarded Control Stack registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{RRBKD} No more Overshooting GCS Memory effects are generated for the Guarded Control Stack of an Exception level if any of the following are true:

- A `GCSB` effect occurs at that Exception level while the Guarded Control Stack is disabled for that Exception level, and the Guarded Control Stack remains disabled while PE is executing at that Exception Level.
- A `GCSB` effect is executed at a different Exception level and PE continues to execute at a different Exception level.

R_{LDSNT} The meaning of completion for a Guarded Control Stack data access is as following:

- A Memory Read effect R1 that is a GCS Memory effect or `GCSSS1` Memory effect to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain, that is not a GCS Memory effect or `GCSSS1` Memory effect, will be Coherence-after R1.
  - Any write W1 to the same Location by an Observer within the shareability domain, that is a GCS Memory effect or `GCSSS1` Memory effect, will be Coherence-after R1 if a `GCSB` effect appears in program order before W1 and R1 is Ordered-before the `GCSB` effect using a DSB instruction.
  - Any translation table walks associated with R1 are complete for that shareability domain.
- A Memory Write effect W1 that is a GCS Memory effect or `GCSSS1` Memory effect to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain, that is not a GCS Memory effect or `GCSSS1` Memory effect, will be Coherence-after W1.
  - Any write W2 to the same Location by an Observer within the shareability domain, that is a GCS Memory effect or `GCSSS1` Memory effect, will be Coherence-after W1 if a `GCSB` effect appears in program order before W2 and W1 is Ordered-before the `GCSB` effect using a DSB instruction.
  - Any read to the same Location by an Observer within the shareability domain, that is not a GCS Memory effect or `GCSSS1` Memory effect, will either Reads-from W1 or Reads-from a Memory Write effect that is Coherence-after W1.
  - Any read R1 to the same Location by an Observer within the shareability domain, that is a GCS Memory effect or `GCSSS1` Memory effect, will either Reads-from W1 or Reads-from a Memory Write effect that is Coherence-after W1 if a `GCSB` effect appears in program order before R1 and W1 is Ordered-before the `GCSB` effect using a DSB instruction.
  - Any translation table walks associated with the write are complete for that shareability domain.

R_{NKPMR} In a TLB maintenance operation, for the purpose of completion requirements of memory accesses, a GCS Memory effect or `GCSSS1` Memory effect is treated in the same way as a Memory effect produced by a regular load/store instruction.

R_{FKQCQ} When a GCS Memory effect or `GCSSS1` Memory effect A1 appears in program order before a data cache maintenance instruction A2, for the purpose of creating ordering relationship between A1 and A2, A1 is treated as a Memory effect produced by a regular load/store instruction.

R_{HLTYS} When a data cache maintenance instruction A1 appears in program order before a GCS Memory effect or `GCSSS1` Memory effect A2, for the purpose of creating ordering relationship between A1 and A2, A2 is treated as a Memory effect produced by a regular load/store instruction.

R_{NLNHL} For the purpose of creating ordering relationship between a data cache maintenance instruction A1 on PE1 and a GCS Memory effect or `GCSSS1` Memory effect A2 on PE2, A2 is treated as a Memory effect produced by a regular load/store instruction if all of the following are true:

- A1 is Ordered-before a `GCSB` effect executed on PE2.
- A `GCSB` effect appears in program order before A2.

There is no ordering relationship defined between A1 and A2 if above conditions are not met.

R_{HZDQD} For the purpose of creating ordering relationship between a GCS Memory effect or `GCSSS1` Memory effect A1 on PE1 and a data cache maintenance instruction A2 on PE2, A1 is treated as a Memory effect produced by a regular load/store instruction if all of the following are true:

- A `GCSB` effect appears in program order after A1.
- A `GCSB` effect is Ordered-before A2.

There is no ordering relationship defined between A1 and A2 if above conditions are not met.

S_{XRJSB} Context switching software is expected to use a `GCSB` effect as appropriate. For example code sequences, see Switching EL0 Guarded Control Stacks from EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.11 Detecting when FEAT_GCS is enabled

G_{LCMSZ} FEAT_GCS provides support for a common software supporting the Guarded Control Stack to run with minimal overheads on a PE that does not implement FEAT_GCS.

G_{VTQRM} FEAT_GCS provides support for a common software supporting the Guarded Control Stack to run on a PE which implements FEAT_GCS when the Guarded Control Stack is disabled for this software. This software runs with minimal additional overheads due to the Guarded Control Stack support.

I_{KZSCH} Software might need to interact with the Guarded Control Stack at various points, in particular to manage the contents including removing entries from the GCS using the `GCSPOPM` instruction. FEAT_CHK provides a low-overhead mechanism to detect whether FEAT_GCS is enabled at the current Exception level, to allow such software to avoid running such code when FEAT_GCS is disabled. For more information about FEAT_CHK, see Check Feature.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D12
The Generic Timer in AArch64 state

This chapter describes the implementation of the Arm Generic Timer. It includes an overview of the AArch64 System register interface to an Arm Generic Timer.

It contains the following sections:

- About the Generic Timer.
- The AArch64 view of the Generic Timer.

The Generic Timer in AArch32 state describes the AArch32 view of the Generic Timer, and System Level Implementation of the Generic Timer describes the system level implementation of the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.1 About the Generic Timer

Figure D12-1 shows an example System on Chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *Arm Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

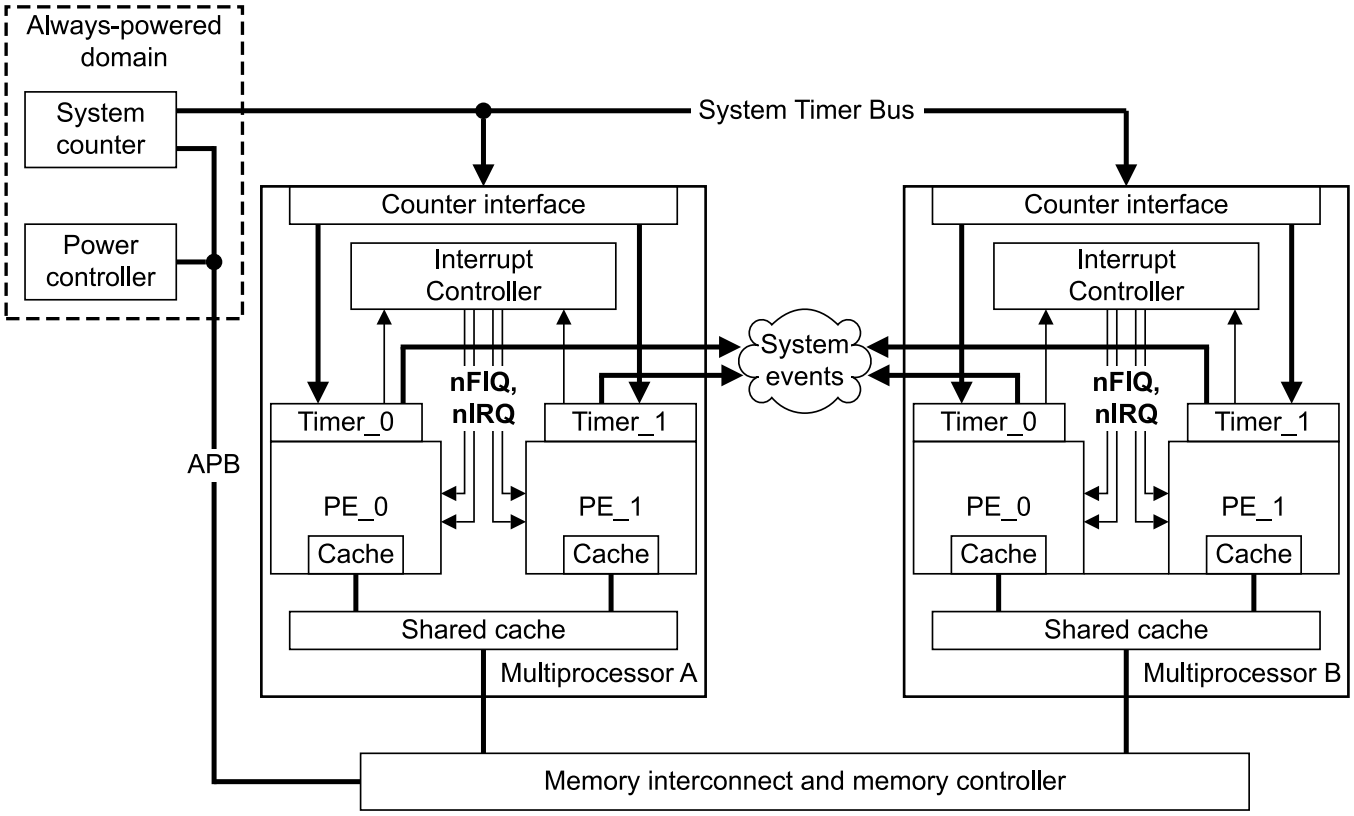

Figure D12-1 Generic Timer example

The Generic Timer:

- Provides a system counter that measures the passing of time in real-time.

  Note

  The Generic Timer can also provide other components at a system level, but Figure D12-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure D12-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch64 state. The Generic Timer in AArch32 state describes access to this component from AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure D12-1.
- Optionally, can provide timer components for use at a system level.

System Level Implementation of the Generic Timer describes this memory-mapped component.

D12.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see The system counter. Because this must be implemented at the system level, it is accessed through The system level memory-mapped implementation of the Generic Timer. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, which gives access to the count value of the system counter. “When FEAT_ECV_POFF is implemented, the CNTPOFF_EL2 register allows offsetting of physical timers and counters.
- A virtual counter, which gives access to virtual time. In AArch64 state, the CNTVOFF_EL2 register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch64 state, the timers that are accessible from AArch64 state are:
  - An EL1 physical timer.
  - A Non-secure EL2 physical timer.
  - An EL3 physical timer.
  - An EL1 virtual timer.
  - A Non-secure EL2 virtual timer.
  - A Secure EL2 virtual timer.
  - A Secure EL2 physical timer.

  The Non-secure EL2 virtual timer is available only when FEAT_VHE is implemented.
  The Secure EL2 timers are available only when FEAT_SEL2 is implemented.

The AArch64 view of the Generic Timer describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into *frames*. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally. provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0 - 7.

System Level Implementation of the Generic Timer describes these components.

D12.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width

From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.

From Armv8.6, must be 64 bits wide.

Effective frequency

This indicates the amount of time each unit of the counter represents. For example:

- An effective frequency of 50MHz means each unit of the counter represents 20ns.
- An effective frequency of 1GHz means each unit of the counter represents 1ns.
- The effective frequency is indicated in the CNTFRQ_EL0 register.

From Armv8.0 to Armv8.5 inclusive, the effective frequency is a fixed value, typically in the range 1-50MHz.

From Armv8.6, the effective frequency is a fixed value of 1GHz.

The counter might not be driven by a clock running at the effective frequency. For example, a counter with an effective frequency of 1GHz might have an actual clock running at 125MHz but the counter increments by 8 on each tick of the clock.

Roll-over

Roll-over time of not less than 40 years.

Counter resolution

The counter resolution is a representation of how frequently the counter is updated.

For example, a counter might have an effective frequency of 1GHz, but the actual clock runs at 125MHz and therefore the counter resolution is 125Mhz.

From Armv8.6, Arm recommends the counter resolution is not less than 50MHz in normal running operation.

The counter resolution might vary over time, to accommodate different operating modes of the system such as lower power modes. Such variation might be invisible to the programmer and be system specific, or might be controllable by the programmer through the frequency modes table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accuracy

Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.

Use of different counter resolutions must not affect the implemented accuracy.

Start-up

Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can operate with a lower counter resolution by incrementing by larger amounts at a lower actual clock frequency. For example, a system counter with an effective frequency of 10MHz might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the actual clock frequency, to reduce power consumption.

In this case, the counter must support transitions between higher resolution operation and lower resolution operation, without any impact on the required accuracy of the counter.

From Armv8.6 the counter operates at a fixed effective frequency of 1GHz.

The CNTFRQ_EL0 register is intended to hold a copy of the effective frequency to allow fast reference to this frequency by software running on the PE. For more information, see Initializing and reading the system effective frequency.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. Gray count scheme for timer distribution scheme gives more information about this possibility.

D12.1.2.1 Initializing and reading the system effective frequency

The CNTFRQ_EL0 register must be programmed to the effective frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the effective frequency to the CNTFRQ_EL0 register. Only software executing at the highest implemented Exception level can write to CNTFRQ_EL0.

Note

The CNTFRQ_EL0 register is UNKNOWN at reset, and therefore the effective frequency must be set as part of the system boot process.

Software can read the CNTFRQ_EL0 register, to determine the effective frequency, in the following states:

- Non-secure, Secure, and Realm EL2.
- Non-secure, Secure, and Realm EL1.
- When CNTKCTL_EL1.{EL0PCTEN, EL0VCTEN} is not {0, 0} and CNTHCTL_EL2.{EL0PCTEN, EL0VCTEN} is not {0, 0}, Non-secure, Secure, and Realm EL0.
- Secure EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Secure EL1.
- When CNTKCTL_EL1.{EL0PCTEN, EL0VCTEN} is not {0, 0} and CNTHCTL_EL2.{EL0PCTEN, EL0VCTEN} is not {0, 0}, Non-secure and Secure EL0.

D12.1.2.2 Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the counter resolution.
- Enabling Halt-on-debug, which a debugger can then use to suspend counting.

For descriptions of these controls, see System Level Implementation of the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2 The AArch64 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch64 state:

- The physical counter.
- The virtual counter.
- Event streams.
- Timers.

D12.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT_EL0 register holds the current physical counter value. This is defined as the pseudocode function `PhysicalCountInt()`. Reads of CNTPCT_EL0 register at EL2 or EL3 return the current value of `PhysicalCountInt()`.

When FEAT_ECV is implemented, the CNTPOFF_EL2 register holds the optional physical offset that can be applied at EL0 and EL1 whether EL0 and EL1 are using AArch64 state or AArch32 state.

For more information, see The physical offset register.

Reads of CNTPCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

D12.2.1.1 The self-synchronized view of the physical counter

When FEAT_ECV is implemented, an alternative way to read the physical counter is supported. The CNTPCTSS_EL0 register is a non-speculative view of the physical counter, as seen from the Exception level that CNTPCTSS_EL0 is read from.

Accesses to the CNTPCTSS_EL0 are subject to the same traps as accesses to the CNTPCT_EL0.

Reads of CNTPCT_EL0 occur in program order relative to reads of CNTPCT_EL0 or CNTPCTSS_EL0.

Reads of CNTPCTSS_EL0 occur in program order relative to reads of CNTPCT_EL0 or CNTPCTSS_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D12-1 Ensuring reads of the physical counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTPCT_EL0 must be read, an `ISB` is used to ensure that the read of CNTPCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                        ; polling for some communication to indicate a requirement to
                            ; read the timer
    LDR X1, [X2]
    CMP X1, #1              ; has had the value 1 written to it
    B.NE loop
    ISB                     ; without this the CNTPCT_EL0 could be read before the memory
                            ; location in [X2]
    MRS X1, CNTPCT_EL0
```

When FEAT_ECV is implemented, an access to CNTPCTSS_EL0 can be used in place of the CNTPCT_EL0 which, because it cannot be accessed speculatively, allows the `ISB` to be removed. This means that the following code sequence can be used:

```
loop                        ; polling for some communication to indicate a requirement to
                            ; read the timer
    LDR X1, [X2]
    CMP X1, #1              ; has had the value 1 written to it
    B.NE loop
    MRS X1, CNTPCTSS_EL0
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
  ...                       ; earlier loads and stores
    DSB                     ; completes the earlier loads and stores
    ISB                     ; without this the CNTPCT_EL0 could be read before the completion
                            ; of the earlier loads and stores
    MRS X1, CNTPCT_EL0
```

Or, if FEAT_ECV is implemented:

```
  ...                       ; earlier loads and stores
    DSB                     ; completes earlier loads and stores
    MRS X1, CNTPCTSS_EL0
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers and memory from the read of the physical counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D12-2 Ensuring reads of the physical counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

either:

```
 DSB
   ISB
   MRS Xn, CNTPCT{SS}_EL0    ; either view of the physical counter has the same effect in
                             ; this example
```

or

```
 DMB
   LDR Xa, [Xd]              ; this could be any memory location, for example the stack pointer
   CBZ Xa, next
next
   ISB                       ; this ISB is not needed if the MRS is accessing CNTPCTSS_EL0
   MRS Xn, CNTPCT{SS}_EL0
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
 MRS Xn, CNTPCT{SS}_EL0    ; either view of the physical counter has the same effect in
                           ; this example
   ISB
   LDR Xa, [Xb]            ; this load will be executed after the timer has been read
```

or

```
 MRS Xn, CNTPCT{SS}_EL0    ; either view of the physical counter has the same effect in
                           ; this example
   EOR Xm, Xn, Xn
   LDR Xa, [Xb, Xm]        ; this load will be executed after the timer has been read
```

D12.2.1.2 The physical offset register

When FEAT_ECV_POFF is implemented, the CNTPOFF_EL2 register allows an offset to be applied to the physical counter, as viewed from EL1 and EL0, and to the EL1 physical timer. The functionality of this 64-bit register is affected by CNTHCTL_EL2.ECV.

When CNTHCTL_EL2.ECV is 1, an MRS to CNTPCT_EL0 or CNTPCTSS_EL0 from either EL0 or EL1 that is not trapped returns the value `(PhysicalCountInt() - CNTPOFF_EL2)`. For information on how the EL1 physical timer interrupt is triggered when CNTHCTL_EL2.ECV is 1, see Operation of the CompareValue views of the timers.

When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is 0, then:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value `PhysicalCountInt()`.
- The physical offset is treated as zero for all timer and counter calculations involving the physical offset.

When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is 0, then the behavior of the counters and timers is as described for Armv8.5 and the optional physical offset is not used.

When SCR_EL3.ECVEn is 0, the value of CNTPOFF_EL2 is treated as 0 for all purposes other than direct reads or writes to CNTPOFF_EL2 from EL3.

D12.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, which indicates virtual time.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing at EL1 or EL0, the virtual offset value relates to the current virtual machine.

The CNTVOFF_EL2 register contains the virtual offset, see The virtual offset register.

The CNTVCT_EL0 register holds the current virtual counter value.

Reads of CNTVCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

D12.2.2.1 The self-synchronized view of the virtual counter

When FEAT_ECV is implemented, an alternative way to read the virtual counter is supported. The CNTVCTSS_EL0 register is a non-speculative view of the virtual counter, as seen from the Exception level that CNTVCTSS_EL0 is read from.

Accesses to the CNTVCTSS_EL0 are subject to the same traps as accesses to the CNTVCT_EL0.

Reads of CNTVCT_EL0 occur in program order relative to reads of CNTVCT_EL0 or CNTVCTSS_EL0.

Reads of CNTVCTSS_EL0 occur in program order relative to reads of CNTVCT_EL0 or CNTVCTSS_EL0.

Example D12-3 Ensuring reads of the virtual counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTVCT_EL0 must be read, an ISB is used to ensure that the read of CNTVCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                        ; polling for some communication to indicate a requirement to
                            ; read the timer
    LDR X1, [X2]
    CMP X1, #1              ; has had the value 1 written to it
    B.NE loop
    ISB                     ; without this the CNTVCT_EL0 could be read before the memory
                            ; location in [X2]
    MRS X1, CNTVCT_EL0
```

When FEAT_ECV is implemented, an access to CNTVCTSS_EL0 can be used in place of the CNTVCT_EL0, which, because it cannot be accessed speculatively, allows the `ISB` to be removed. This means that the following code sequence can be used:

```
loop                        ; polling for some communication to indicate a requirement to
                            ; read the timer
    LDR X1, [X2]
    CMP X1, #1              ; has had the value 1 written to it
    B.NE loop
    MRS X1, CNTVCTSS_EL0
```

Similarly where a read of the virtual counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following two sequences can be used:

```
  ...                       ; earlier loads and stores
    DSB                     ; completes earlier loads and stores
    ISB                     ; without this CNTPCT_EL0 could be read before the completion
                            ; of the earlier loads and stores
    MRS X1, CNTVCT_EL0
```

Or, if FEAT_ECV is implemented:

```
  ...                       ; earlier loads and stores
    DSB                     ; completes earlier loads and stores
    MRS X1, CNTVCTSS_EL0
```

Neither view of the virtual counter ensures that:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers and memory from the read of the virtual counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

Example D12-4 Ensuring reads of the virtual counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

either:

```
 DSB
   ISB
   MRS Xn, CNTVCT{SS}_EL0    ; either view of the virtual counter has the same effect in
                             ; this example
```

or

```
 DMB
   LDR Xa, [Xd]              ; this could be any memory location, for example the stack pointer
   CBZ Xa, next
next
   ISB                       ; this ISB is not needed if the MRS is accessing CNTVCT{SS}_EL0
   MRS Xn, CNTVCT{SS}_EL0
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
 MRS Xn, CNTVCT{SS}_EL0      ; either view of the virtual counter has the same effect in
                             ; this example
   ISB
   LDR Xa, [Xb]              ; this load will be executed after the timer has been read
```

or

```
 MRS Xn, CNTVCT{SS}_EL0      ; either view of the virtual counter has the same effect in
                             ; this example
   EOR Xm, Xn, Xn
   LDR Xa, [Xb, Xm]          ; this load will be executed after the timer has been read
```

D12.2.2.2 The virtual offset register

The virtual counter is a counter that has a *virtual offset* relative to the physical counter as viewed from EL2 and EL3. This virtual offset is held in the register CNTVOFF_EL2. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented, then the virtual counter uses a fixed offset of zero.

D12.2.3 Event streams

An implementation that includes the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wakeup events as part of the mechanism described in Wait for Event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The CNTKCTL_EL1.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the virtual counter.

In all implementations, the CNTHCTL_EL2.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:

- EVNTI selects the counter bit that triggers the event.
- If FEAT_ECV is not implemented, EVNTI selects between bits[0:15].
- If FEAT_ECV is implemented, EVNTIS selects whether ENVTI selects between bits[0:15] or bits[8:23].
- EVNTDIR selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

Note

If the event stream is configured to produce events from the low order bits of the counter when the effective frequency is very high (for example 1GHz), then the practical update rate of the counter might mean that the event stream is not generated as the low order bit might not change. Software can rely on an event stream rate of at least 1MHz in normal operation.

The pseudocode descriptions of the operation of an event stream are `SetEventRegister`, `TestEventCNTV`, and `TestEventCNTP`.

D12.2.4 Timers

In an implementation of the Generic Timer that includes EL3, if EL3 can use AArch64, the following timers are implemented:

- An EL1 physical timer, which:
  - In Non-secure state, can be accessed from EL1 unless those accesses are trapped to EL2.
  - In Secure state, can be accessed from EL1.
  - In Realm state, can be accessed from EL1 unless those accesses are trapped to EL2.

  When this timer can be accessed from EL1, an EL1 control determines whether it can be accessed from EL0.
- A Non-secure EL2 physical timer.
- An EL3 physical timer. SCR_EL3.{ST, EEL2} determines whether this register is accessible from Secure EL1.
- An EL1 virtual timer.
- When FEAT_VHE is implemented, a Non-secure EL2 virtual timer.
- When FEAT_SEL2 is implemented, a Secure EL2 physical timer.
- When FEAT_SEL2 is implemented, a Secure EL2 virtual timer.

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:

- Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.
- Has, in addition, a 32-bit Control register.

Table D12-1 Physical timer registers summary for the Generic Timer

Timer[a] register	EL1 physical timer	EL2 physical timer	Secure EL2 physical timer[b]	EL3 physical timer
CV	CNTP_CVAL_EL0	CNTHP_CVAL_EL2	CNTHPS_CVAL_EL2	CNTPS_CVAL_EL1
TV	CNTP_TVAL_EL0	CNTHP_TVAL_EL2	CNTHPS_TVAL_EL2	CNTPS_TVAL_EL1
Control	CNTP_CTL_EL0	CNTHP_CTL_EL2	CNTHPS_CTL_EL2	CNTPS_CTL_EL1

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Present only when the implementation includes FEAT_SEL2.

Table D12-2 Virtual timer register summary for the Generic Timer

Timer[a] register	EL1 virtual timer	EL2 virtual timer[b]	Secure EL2 virtual timer[c]
CV	CNTV_CVAL_EL0	CNTHV_CVAL_EL2	CNTHVS_CVAL_EL2
TV	CNTV_TVAL_EL0	CNTHV_TVAL_EL2	CNTHVS_TVAL_EL2
Control	CNTV_CTL_EL0	CNTHV_CTL_EL2	CNTHVS_CTL_EL2

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Only when the implementation includes FEAT_VHE.

c. Present only when the implementation includes FEAT_SEL2.

D12.2.4.1 Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register.

When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, CNTV_CTL_EL0, CNTHV_CTL_EL2 or CNTHVS_CTL_EL2. For CNTP_CTL_EL0, the asserted interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch32 register CNTP_CTL.

The operation of this view of a timer is:

```
TimerConditionMet = (((PhysicalCountInt() - Offset)[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`TimerConditionMet` Is TRUE if the timer condition for this counter is met, and FALSE otherwise.

`PhysicalCountInt()` The physical counter value, which can be read from the CNTPCT_EL0 register when read from EL2 or EL3.

`Offset` For the EL1 physical timer, the offset value is the value held in the CNTPOFF_EL2 register when all of the following are true:

- SCR_EL3.{NS, EEL2} is not {0, 0}.
- SCR_EL3.ECVEn is 1,

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTHCTL_EL2.ECV is 1.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise the offset value of the EL1 physical timer is zero.

For the EL1 virtual timer, the offset value is held in the CNTVOFF_EL2 register.

For the EL3 timer, the EL2 Secure physical timer, the EL2 Non-secure physical timer, the EL2 Secure virtual timer, and the EL2 Non-secure virtual timer the offset value is zero.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL_EL0, CNTHP_CVAL_EL2, CNTPS_CVAL_EL1, CNTHPS_CVAL_EL2, CNTV_CVAL_EL0, CNTHV_CVAL_EL2, or CNTHVS_CVAL_EL2.

In this view of a timer, `PhysicalCountInt()`, `Offset`, and `CompareValue` are all 64-bit unsigned values.

Note

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.
Software can observe the counter value by the offset in some situations by reading CNTVCT_EL0.

D12.2.4.2 Operation of the TimerValue views of the timers

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, CNTV_CTL_EL0, CNTHV_CTL_EL2, or CNTHVS_CTL_EL2.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads

```
TimerValue = (CompareValue[63:0] - (PhysicalCountInt()[63:0] - Offset[63:0]))[31:0]
```

Writes

```
CompareValue = ((PhysicalCountInt()[63:0] - Offset[63:0])[63:0] +
    ↪SignExtend{64}(TimerValue[31:0]))
```

Where the arguments other than `TimerValue` have the definitions used in Operation of the CompareValue views of the timers, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL_EL0, CNTHP_TVAL_EL2, CNTHPS_TVAL_EL2, CNTPS_TVAL_EL1, CNTV_TVAL_EL0, CNTHV_TVAL_EL2, or CNTHVS_TVAL_EL2.

In this view of a timer, values are signed in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

Note

- Operation of the CompareValue views of the timers gives a strict definition of `TimerConditionMet`. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x8000_0000`, the TimerValue view can be used as giving an effect equivalent to:

  `TimerConditionMet = (TimerValue ≤ 0)`

- Programming TimerValue to a negative number with magnitude greater than (Counter - Offset) can lead to an arithmetic overflow that causes the CompareValue to be a large positive value. This potentially delays meeting the timer condition for a long period of time.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D13
The Performance Monitors Extension

This chapter describes the implementation of the Arm Performance Monitors, which are an optional non-invasive debug component. It describes version 3 of the *Performance Monitor Unit* (PMU) architecture, FEAT_PMUv3. It contains the following sections:

- About the Performance Monitors.
- Accuracy of the Performance Monitors.
- Behavior on overflow.
- Attributability.
- Controlling the PMU counters.
- Multithreaded implementations.
- Event filtering.
- Event threshold and edge counting.
- PMU snapshots.
- Performance Monitors and Debug state.
- Counter access.
- Performance Monitors Extension registers.

Note

Table K14-2 disambiguates the general register references used in this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.1 About the Performance Monitors

The Performance Monitors Extension is an OPTIONAL feature of an implementation, but Arm strongly recommends that implementations include version 3 of the Performance Monitors Extension, FEAT_PMUv3.

Note

No previous versions of the Performance Monitors Extension can be implemented in architectures from Armv8.0.

The Performance Monitors Extension provides:

- The 64-bit cycle counter, PMCCNTR_EL0. The cycle counter is always implemented.
- The 64-bit instruction counter, PMICNTR_EL0. The instruction counter is an optional feature, FEAT_PMUv3_ICNTR.
- The event counters, PMEVCNTR<n>_EL0. The number of event counters is an IMPLEMENTATION DEFINED parameter, NUM_PMU_COUNTERS, between 0 and 31.

  Note

  The Performance Monitors Extension permits an implementation with zero event counters only if one of the following is true:

  — EL2 is not implemented.

  — FEAT_HPMN0 is implemented.

  This means only the cycle counter is implemented. Optionally, the instruction counter can also be implemented. Arm recommends that at least two event counters are implemented, and that hypervisors provide at least this many event counters to guest operating systems.

  The event counters are either 64 or 32 bit counters:

  — If FEAT_PMUv3p5 is implemented and the highest Exception level is using AArch64, then the event counters are 64 bit.

  — If FEAT_PMUv3p5 is not implemented, the event counters are 32 bit.
- Controls for:
  — Enabling counters. See Enabling PMU counters.
  — Resetting counters. Resetting counters.
  — Flagging overflows. See Behavior on overflow.
  — Enabling interrupts on overflow. See Generating overflow interrupt requests.
  — Freezing counters. See Freezing PMU counters.
  — Prohibiting counting. See Prohibiting counting.
  — Threshold and edge counting. See Event threshold and edge counting.
  — Enabling access to the Performance Monitors from EL0. See EL0 access controls.
  — Partitioning the implemented event counters into ranges. See Event counter ranges.

The PMU architecture uses event numbers to identify an event. It:

- Defines event numbers for common events, for use across many architectures and microarchitectures. See Common event numbers.

  Note

  Implementations that include FEAT_PMUv3 must, as a minimum requirement, implement a subset of the common events. See Common event numbers.
- Reserves a large event number space for IMPLEMENTATION DEFINED events. See IMPLEMENTATION DEFINED event numbers.

The full set of events for an implementation is IMPLEMENTATION DEFINED. Arm recommends that implementations include all of the events that are appropriate to the architecture profile and microarchitecture of the implementation. For more information, see PMU Event Descriptions.

When an implementation includes the Performance Monitors Extension, the architecture defines the following possible interfaces to the Performance Monitors Extension registers:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A System register interface. This interface is mandatory.

  Note

  In AArch32 state, the interface is in the (`coproc` == `0b1111`) encoding space.

- An external debug interface which optionally supports memory-mapped accesses. Implementation of this interface is OPTIONAL. See Recommended External Interface to the Performance Monitors.

An operating system can use the System registers to access the counters.

Also, if required, the operating system can enable application software to access the counters. This enables an application to monitor its own performance with fine-grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

To enable interaction with external monitoring, an implementation might consider additional enhancements, such as providing:

- A set of events, from which a selection can be exported onto a bus for use as external events.
- The ability to count external events. This enhancement requires the implementation to include a set of external event input signals.

The Performance Monitors Extension is common to AArch64 operation and AArch32 operation. This means the architecture defines both AArch64 and AArch32 System registers to access the Performance Monitors. For example, the Performance Monitors Cycle Count Register is accessible as:

- When executing in AArch64 state, PMCCNTR_EL0.
- When executing in AArch32 state, PMCCNTR.

When executing in AArch32 state, if FEAT_PMUv3p5 is implemented, bits [63:32] of the event counters are not accessible. If the implementation does not support AArch64 at any Exception level, 64-bit event counters are not required to be implemented.

When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0 is added, but is not accessible in AArch32 state.

D13.1.1 Event counter ranges

DXWVNH

The event counters are partitioned into three ranges, as follows:

- If *epmn* is less-than NUM_PMU_COUNTERS, then event counters [*epmn*..(NUM_PMU_COUNTERS-1)] are in the *third range*.
  If *epmn* is equal to NUM_PMU_COUNTERS, then no event counters are in the *third range*.
- If *hpmn* is less-than *epmn*, then event counters [*hpmn*..(epmn-1)] are in the *second range*.
  If *hpmn* is equal to *epmn*, then no event counters are in the *second range*.
- If *hpmn* is not zero, then event counters [0..(*hpmn*-1)] are in the *first range*.
  If *hpmn* is zero, then no event counters are in the *first range*.

Where *epmn* and *hpmn* are defined as follows:

epmn

Is the Effective value of PMCCR.EPMN. If FEAT_PMUv3_EXTPMN is not implemented, then the Effective value of PMCCR.EPMN is NUM_PMU_COUNTERS, meaning no counters are in the third range.

hpmn

Is the Effective value of HDCR.HPMN, as described in R_{MFYQC}. If EL2 is not implemented, then *hpmn* has the same value as *epmn*, meaning no counters are in the second range.

This does not depend on whether EL2 is enabled in the current Security state. Each range of event counters has its own global controls.

Counters in the second range are also described as *reserved for EL2*.

Counters in the third range are also described as *reserved for an external agent*.

ISTBND

If any of the following apply, then the behavior is CONSTRAINED UNPREDICTABLE:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_HPMN0 is not implemented and HDCR.HPMN is 0.
- FEAT_PMUv3_EXTPMN is not implemented and HDCR.HPMN is greater-than NUM_PMU_COUNTERS.
- FEAT_PMUv3_EXTPMN is implemented and PMCCR.EPMN is greater-than NUM_PMU_COUNTERS.

See:

- The Performance Monitors Extension when accessing the Performance Monitors Extension in AArch64 state.
- The Performance Monitors Extension when accessing the Performance Monitors Extension in AArch32 state.

I_{SBQGJ} When EL2 is enabled in the current Security state, self-hosted software executing at EL1 and, if enabled by PMUSERENR, EL0 only has access through System registers to event counters in the first range, the cycle counter PMCCNTR, and, if FEAT_PMUv3_ICNTR is implemented, the instruction counter, PMICNTR_EL0. This software is not able to directly determine that event counters in the second range are implemented.

Self-hosted software executing at EL2 and EL3, and other Exception levels when EL2 is disabled in the current Security state, additionally has access through System registers to event counters in the second range.

Self-hosted software executing on the PE has no access through System registers to event counters in the third range. This software is not able to directly determine that event counters in the third range are implemented.

R_{NNBYQ} If all of the following are true, then direct reads of PMCR_EL0.N or PMCR.N return the Effective value of HDCR.HPMN:

- PE is executing at EL0 or EL1.
- The access does not generate an exception.

R_{ZCCDD} If FEAT_PMUv3_EXTPMN is implemented and any of the following are true, then direct reads of PMCR_EL0.N or PMCR.N return the Effective value of PMCCR.EPMN:

- PE is executing at EL2 or EL3.
- PE is executing at EL0 or EL1 and EL2 is either disabled or not implemented in the current Security state.
- The access does not generate an exception.

S_{HVGXL} An external agent reserving event counters should perform one of the following actions:

- Halt the PE immediately after reset and write to PMCCR.EPMN while the PE is in Debug state so that software running on the PE does not observe PMCR.N prior to the configuration of PMCCR.EPMN.
- Use an IMPLEMENTATION DEFINED mechanism such that PMCCR.EPMN is set to an intended value.

An external agent should not modify PMCCR.EPMN after any software starts running on the PE.

Similar guidance applies for EL2 to not modify HDCR.HPMN for a Guest OS after the Guest has been started.

D13.1.2 Interaction with EL3

Software executing at EL3 can trap attempts by lower Exception levels to access the PMU. This means that the Monitor can identify any software which is using the PMU and switch contexts, if required.

Software executing at EL3 can:

- Prohibit counting of events Attributable to Secure state.
- If FEAT_PMUv3p5 is implemented, prohibit the cycle counter from counting cycles in Secure state and at EL3.
- If FEAT_PMUv3p7 is implemented:
  - Prohibit event counters from counting events at EL3.
  - Prohibit the cycle counter from counting cycles at EL3.
- If FEAT_PMUv3_ICNTR is implemented, prohibit the instruction counter from counting architecturally-executed instructions in Secure state and at EL3.

For more information, see Controlling the PMU counters.

In AArch32 state, the Performance Monitors registers are Common registers, see Classification of System registers.

If FEAT_MTPMU is implemented and EL3 is implemented, MDCR_EL3.MTPME and SDCR.MTPME enable and disable the PMEVTYPER<n>.MT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.1.3 Interaction with EL2

IFZHQW Software executing at EL3 or EL2 can program HDCR.HPMN to partition the event counters into a first range and a second range. See Event counter ranges.

IGZYQM Software executing at EL3 or EL2 can:

- Trap an access at EL0 or EL1 to the PMU. This means the hypervisor can identify which Guest OSs are using the PMU and intelligently employ switching of the PMU state. There is a separate trap for the PMCR register, and if FEAT_FGT is implemented and enabled, fine-grained traps are provided.
- If FEAT_PMUv3p1 is implemented, prohibit counting of events Attributable to EL2 by the event counters in the first range.
- If FEAT_PMUv3p5 is implemented, prohibit the cycle counter from counting cycles at EL2.
- If FEAT_PMUv3_ICNTR is implemented, prohibit the instruction counter from counting instructions at EL2.

When EL2 is implemented and enabled in the current Security state, software executing at EL1 and, if enabled by PMUSERENR, EL0:

- Will read the Effective value of HDCR.HPMN for PMCR.N.
- Cannot access the event counters in the second range, or the controls associated with them.

If FEAT_MTPMU is implemented, EL3 is not implemented, and EL2 is implemented, MDCR_EL2.MTPME and HDCR.MTPME enable and disable the PMEVTYPER<n>.MT bit.

RMFYQC If FEAT_PMUv3_EXTPMN is implemented, EL2 is implemented, and MDCR_EL2.HPMN is greater-than the Effective value of PMCCR.EPMN, then all of the following apply:

- Direct reads of MDCR_EL2.HPMN return the Effective value of PMCCR.EPMN.
- The Effective value of MDCR_EL2.HPMN is the Effective value of PMCCR.EPMN. This includes indirect reads of MDCR_EL2.HPMN as a result of direct reads from PMCR_EL0.N or PMCR.N.

If FEAT_PMUv3_EXTPMN is implemented and EL2 is not implemented, then the Effective value of MDCR_EL2.HPMN is the Effective value of PMCCR.EPMN.

IMVYJY For more information, see:

- Enabling PMU counters.
- Counter access.
- Controlling the PMU counters.
- Multithreaded implementations.

D13.1.4 Interaction with trace

When FEAT_ETE is implemented, the trace unit uses the PMU events as External Inputs. See External inputs.

When FEAT_ETMv4 is implemented, it is IMPLEMENTATION DEFINED whether the implementation exports PMU events to the ETM trace unit, and the form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting is controlled by PMCR.X.

For more information about when the trace unit is permitted to observe PMU events, see Controls to prohibit trace at Exception levels and External inputs.

When FEAT_ETE is implemented, the PMU implements a set of events for importing external events from the trace unit and the cross-trigger interface. See Required events. Arm recommends similar functionality is implemented when FEAT_ETMv4 is implemented, and further recommends reusing the events defined for FEAT_ETE. When implemented, this functionality enables the trace unit to pass in events to be counted.

D13.1.5 Export of PMU events

It is IMPLEMENTATION DEFINED whether the implementation exports counter events to any other external monitoring agent to provide triggering information. The form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting is controlled by PMCR.X.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.2 Accuracy of the Performance Monitors

The Performance Monitors:

- Are a non-invasive debug component. See Non-invasive behavior.
- Must provide broadly accurate and statistically useful count information.

However, the Performance Monitors allow for:

- A reasonable degree of inaccuracy in the counts to keep the implementation and validation cost low. See A reasonable degree of inaccuracy.
- IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, that software must configure before using certain PMU events. For example, to configure how the PE generates PMU events for components such as external caches and external memory.
- Other IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to optionally put the PE in an operating state that might do one or both of the following:
  - Change the level of non-invasiveness of the Performance Monitors so that enabling an event counter can impact the performance or behavior of the PE.
  - Allow inaccurate counts. This includes, but is not limited to, cycle counts.

D13.2.1 Non-invasive behavior

The Performance Monitors are a non-invasive debug feature. A non-invasive debug feature permits the observation of data and program flow. Performance Monitors, PC Sample-based Profiling and Trace are non-invasive debug features.

Non-invasive debug components do not guarantee that they do not make any changes to the behavior or performance of the processor. Any changes that do occur must not be severe however, as this will reduce the usefulness of event counters for performance measurement and profiling. This does not include any change to program behavior that results from the same program being instrumented to use the Performance Monitors, or from some other performance monitoring process being run concurrently with the process being profiled in a multitasking operating system. As such, a reasonable variation in performance is permissible.

Note

Power consumption is one measure of performance. Therefore, a reasonable variation in power consumption is permissible.

Arm does not define a *reasonable variation in performance*, but recommends that such a variation is kept within 5% of normal operating performance, when averaged across a suite of code that is representative of the application workload.

Note

For profiles other than A-profile, there is the potential for stronger requirements. Ultimately, performance requirements are determined by end-users, and not set by the architecture.

For some common architectural events, this requirement to be non-invasive can conflict with the requirement to present an accurate value of the count under normal operating conditions. Should an implementation require more performance-invasive techniques to accurately count an event, there are the following options:

- If the event is optional, define an alternative IMPLEMENTATION DEFINED event that accurately counts the event and document the impact on performance of enabling the event.
- Provide an IMPLEMENTATION DEFINED control that disables accurate counting of the event to restore broadly accurate performance, and document the impact on performance of accurate counting.

Note

The IMPLEMENTATION DEFINED invasiveness might include the PE disabling one or more performance optimizations such as multi-issue, out-of-order execution, and pipelining.
A software implication is that the performance invasiveness is proportional to the sampling interval. This can disproportionately affect profiling of rare events, because the population is smaller and therefore the interval will tend to be smaller.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.2.2 A reasonable degree of inaccuracy

The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation, validation, and performance overhead costs low, a reasonable degree of inaccuracy in the counts and the detection of overflow is acceptable. Arm does not define a *reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions, the counters must present an accurate value of the count.
- In exceptional circumstances, such as enabling or disabling the counter, a change in Security state, taking an exception, executing an exception return, or other boundary condition, it is acceptable for the count to be inaccurate, unless otherwise stated.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

Note

An implementation must not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. For example, it is not reasonable for the count of branch behavior to be inaccurate when caused by a systematic error generated by the loop structure producing a dropping in branch count.
However, dropping a single branch count as the result of a rare interaction with an interrupt is acceptable.

The architecture does not permit these behaviors for events that specifically count architectural exceptional circumstances, such as a change in Exception level, for example, EXC_TAKEN.

The permitted inaccuracy limits the possible uses of the Performance Monitors. In particular, the architecture does not define the point in a pipeline where the event counter is incremented, relative to the point where a read of the event counters is made. This means that pipelining effects can cause some imprecision, and can affect which events are counted.

This applies to both architectural and microarchitectural events, and can apply differently to different events. Arm recommends that is applied consistently between counters. For example, any boundary condition effects that apply to an event counter counting the CPU_CYCLES event should apply in the same way to the cycle counter PMCCNTR_EL0. However, this is not required.

Inaccuracies around exceptional events, such as enabling or disabling the counter, should be such that the inaccuracy is reduced by lengthening the sampling period.

Where a direct write to a Performance Monitors control register disables a counter, and is followed by a Context Synchronization event, any subsequent indirect read of the control register by the Performance Monitors to determine whether the counter is enabled will return the updated value. Any subsequent direct read of the counter or counter overflow status flags will return the value at the point the counter was disabled.

Note

The imprecision means that the counter might have counted an event around the time the counter was disabled, but does not allow the event to be observed as counted after the counter was disabled.

A change of Security state can also affect the accuracy of the Performance Monitors, see Interaction with EL3.

In addition to this, entry to and exit from Debug state can disturb the normal running of the PE, causing further inaccuracy in the Performance Monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. An implementation can employ methods to limit this inaccuracy, for example by promptly disabling the counters during the Debug state entry sequence.

An implementation must document any particular scenarios where significant inaccuracies are expected.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.3 Behavior on overflow

R_{QZJKS} The event counters, PMEVCNTR<n> are either 32-bit or 64-bit unsigned counters:

- If FEAT_PMUv3p5 is not implemented, then 32-bit event counters are implemented.
- If FEAT_PMUv3p5 is implemented, then 64-bit event counters are implemented.

The event counters overflow in the following situations:

- If FEAT_PMUv3p5 is not implemented and incrementing PMEVCNTR<n> causes an unsigned overflow of the event counter, then the PE sets PMOVSCLR[*n*] to 1.
- If FEAT_PMUv3p5 is implemented and event counter PMEVCNTR<n> is in the first range, then event counter overflow is configured by PMCR.LP:
  - If the Effective value of PMCR.LP is 0 and incrementing PMEVCNTR<n> causes an unsigned overflow of bits [31:0] of the event counter, then the PE sets PMOVSCLR[*n*] to 1.
  - If the Effective value of PMCR.LP is 1 and incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, then the PE sets PMOVSCLR[*n*] to 1.

  When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of PMCR.LP is 1.
- If FEAT_PMUv3p5 is implemented and event counter PMEVCNTR<n> is in the second range, then event counter overflow is configured by HDCR.HLP:
  - If the Effective value of HDCR.HLP is 0 and incrementing PMEVCNTR<n> causes an unsigned overflow of bits [31:0] of the event counter, then the PE sets PMOVSCLR[*n*] to 1.
  - If the Effective value of HDCR.HLP is 1 and incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, then the PE sets PMOVSCLR[*n*] to 1.

  When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of HDCR.HLP is 1.
- If event counter PMEVCNTR<n> is in the third range and incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, then the PE sets PMOVSCLR[*n*] to 1.

R_{FBGFH} If FEAT_PMUv3_ICNTR is implemented, then when incrementing PMICNTR_EL0 causes an unsigned overflow of PMICNTR_EL0[63:0], the PE sets PMOVSCLR.F0 to 1.

R_{NTZHW} The cycle counter, PMCCNTR, is a 64-bit unsigned counter, that is configured by PMCR.LC:

- When the Effective value of PMCR.LC is 0, if incrementing PMCCNTR causes an unsigned overflow of bits [31:0] of the cycle counter, the PE sets PMOVSCLR[31] to 1.
- When the Effective value of PMCR.LC is 1, if incrementing PMCCNTR causes an unsigned overflow of bits [63:0] of the cycle counter, the PE sets PMOVSCLR[31] to 1.

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of PMCR.LC is 1.

R_{HTPDY} The update of PMOVSCLR occurs synchronously with the update of the counter.

I_{RYCLJ} The behavior of the PMU, and other functions that observe the PMU or are triggered by its state, such as the BRBE freeze events, ignore the values of PMUSERENR_EL0 and PMUACR_EL1. For example, a counter that is not currently accessible to EL0 might overflow and cause an unexpected PMU freeze-on-overflow or BRBE freeze event when executing at EL0.

I_{FYVTD} For all 64-bit counters, incrementing the counter is the same whether an unsigned overflow occurs at [31:0] or [63:0]. If the counter increments for an event, bits [63:0] are always incremented,

I_{HRJFC} A PMU overflow can generate an interrupt request. For more information, see Generating overflow interrupt requests.

I_{YBSFJ} If FEAT_EBEP is implemented, then a PMU overflow can generate a PMU Profiling exception. For more information, see Exception-based event profiling.

I_{ZWQTX} If FEAT_BRBE is implemented, then a PMU overflow can generate a BRBE freeze event. For more information, see Branch record buffer operation.

I_{LMKZN} If FEAT_PMUv3p9 and FEAT_Debugv8p9 are implemented, then a PMU overflow can cause the PE to halt and enter Debug state. For more information, see PMU Overflow external debug request.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IPVBJQ A PMU overflow can be used as an input trigger event to the *Cross Trigger Interface* (CTI) for the PE. For more information, see Performance Monitors overflow trigger event.

IXBXPG If FEAT_PMUv3p7 is implemented, then a PMU overflow can cause event counting to be frozen. For more information, see Freezing PMU counters.

IZNDRG Software executing at EL1 or higher must take care that setting PMCR.LP or HDCR.HLP does not cause software executing at lower Exception levels to malfunction. If legacy software accesses the PMU at lower Exception levels, software at the higher Exception levels should not set the PMCR.LP or HDCR.HLP fields to 1. However, if the legacy software does not use the counter overflow, it is not affected by setting the PMCR.LP or HDCR.HLP to 1.

D13.3.1 Generating overflow interrupt requests

Software can program the Performance Monitors so that an overflow interrupt request is generated when a counter overflows. See PMINTENSET and PMINTENCLR.

Note

- The mechanism by which an interrupt request from the Performance Monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.
- Arm recommends that the overflow interrupt requests:
  - Translate into a **PMUIRQ** signal, so that they are observable to external devices.
  - Connect to inputs on an IMPLEMENTATION DEFINED Generic Interrupt Controller as a *Private Peripheral Interrupt* (PPI) for the originating processor. See the *ARM Generic Interrupt Controller Architecture Specification* for information about PPIs.
  - Connect to a CTI, see The Embedded Cross-Trigger Interface.
- Arm strongly discourages implementations from connecting overflow interrupt requests from multiple PEs to the same *System Peripheral Interrupt* (SPI) identifier.
- From GICv3, the *ARM® Generic Interrupt Controller Architecture Specification* recommends that the *Private Peripheral Interrupt* (PPI) with ID 23 is used for overflow interrupt requests.

Software can write to the counters to control the frequency at which interrupt requests occur. For example, software might set a 32-bit counter to `0xFFFF_0000`, to generate another counter overflow after 65 536 increments, and reset it to this value every time an overflow interrupt occurs.

Note

If an event can occur multiple times in a single clock cycle, then counter overflow can occur without the counter registering a value of zero.

The overflow interrupt request is a level-sensitive request. The PE signals a request for:

- Any implemented PMEVCNTR<n> counter, when the global enable control for the event counter is 1, the value of PMOVSSET.P<n> is 1, and the value of PMINTENSET.P<n> is 1.
- The cycle counter, when the values of PMCR.E, PMOVSSET.C and PMINTENSET.C are all 1.
- The instruction counter, when the values of PMCR.E, PMOVSSET_EL0.F0, and PMINTENSET_EL1.F0, are all 1.

The overflow interrupt request is active in all Security states. In particular, if EL3 and EL2 are both implemented, overflow events from PMEVCNTR<n> where *n* is greater than or equal to the value of HDCR.HPMN can be signaled from all modes and states but only if the value of HDCR.HPME is 1.

The interrupt handler for the counter overflow request must cancel the interrupt request, for example, by writing 1 to PMOVSCLR.P<n> to clear the overflow bit to 0.

D13.3.1.1 Pseudocode description of overflow interrupt requests

See A-profile Architecture Pseudocode for a pseudocode description of overflow interrupt requests. The `CheckForPMUOverflow()` pseudocode function signal PMU overflow interrupt requests to an interrupt controller and PMU overflow trigger events to the cross-trigger interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.3.2 Exception-based event profiling

R_{CXBNL} If FEAT_EBEP is implemented, a *PMU Profiling exception* is supported.

I_{FNPMD} PMU Profiling exceptions are always taken asynchronously, and are precise. See Definition of a precise exception and imprecise exception for the definition of precise.

I_{JMSPM} FEAT_EBEP allows a PMU Profiling exception to be generated when a counter overflow bit is set to 1.

I_{PFDLQ} For backward compatibility, software has to enable the PMU Profiling exception. The exception is enabled for all implemented counters.

Enabling the PMU Profiling exception disables the PMU overflow interrupt request. The CTI Performance Monitors overflow trigger event is unchanged.

R_{GWLVY} A PMU Profiling exception is *enabled* and taken to one of the following:

- EL1 if all of the following are true:
  - PMECR_EL1.PMEE is `0b11`.
  - EL2 is not implemented, EL2 is disabled in the current Security State, or MDCR_EL2.PMEE is `0b01`.
  - EL2 is not implemented, EL2 is disabled in the current Security State, or HCR_EL2.TGE is 0.
  - EL3 is not implemented or MDCR_EL3.PMEE is `0b01`.
- EL2 if all of the following are true:
  - EL2 is implemented and enabled in the current Security State.
  - One of the following is true:
    - MDCR_EL2.PMEE is `0b11`.
    - PMECR_EL1.PMEE is `0b11`, MDCR_EL2.PMEE is `0b01`, and HCR_EL2.TGE is 1.
  - EL3 is not implemented or MDCR_EL3.PMEE is `0b01`.
- EL3 if all of the following are true:
  - EL3 is implemented.
  - MDCR_EL3.PMEE is `0b11`.

The PMU Profiling exception is *disabled* otherwise.

The pseudocode function `PMUExceptionEnabled()` shows this.

R_{SCBDZ} An enabled PMU Profiling exception is *masked* when any of the following are true:

- The PE is executing in Debug state.
- The PE is executing at a higher Exception level than the target Exception level of the PMU Profiling exception.
- The PE is executing at EL2, the target Exception level of the PMU Profiling exception is EL2, and MDCR_EL2.PMEE is not `0b11`.
- The PE is executing at the target Exception level of the PMU Profiling exception, and PSTATE.PM is 1 or PMECR_EL1.KPME is 0.

Otherwise an enabled PMU Profiling exception is *unmasked*.

The pseudocode function `PMUExceptionMasked()` shows this.

I_{PPSLS} Table D13-1 shows the PMU Profiling exception enable and masking for each Exception level. For this table, EL2 is enabled in the current Security state.

Each column in the left side of the table represents the Effective value of a control bit. Each column in the right side of the table represents an Exception level the exception is taken from. In each cell for each of these columns on each row representing a combination of control bits:

IRQ
: Means the exception is disabled and the PMU interrupt request (**PMUIRQ**) is enabled.

Dis
: Means the exception is disabled and the PMU interrupt request is disabled.

Msk
: Means the exception is enabled and masked.

ELx
: Means the exception is enabled and taken to ELx.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D13-1 PMU Profiling exception enable and masking for each Exception level

MDCR_EL3. PMEE	MDCR_EL2. PMEE	HCR_EL2. TGE	PMECR_EL1. PMEE	PMECR_EL1. KPME	PSTATE.PM	EL3	EL2	EL1	EL0
0b00	xx	0	xx	x	x	IRQ	IRQ	IRQ	IRQ
0b00	xx	1	xx	x	x	IRQ	IRQ	n/a	IRQ
0b01	0b00	0	xx	x	x	IRQ	IRQ	IRQ	IRQ
0b01	0b00	1	xx	x	x	IRQ	IRQ	n/a	IRQ
0b01	0b01	0	0b00	x	x	IRQ	IRQ	IRQ	IRQ
0b01	0b01	0	0b10	x	x	Dis	Dis	Dis	Dis
0b01	0b01	0	0b11	0	x	Msk	Msk	Msk	EL1
0b01	0b01	0	0b11	1	0	Msk	Msk	EL1	EL1
0b01	0b01	0	0b11	1	1	Msk	Msk	Msk	EL1
0b01	0b01	1	0b00	x	x	IRQ	IRQ	n/a	IRQ
0b01	0b01	1	0b10	x	x	Dis	Dis	n/a	Dis
0b01	0b01	1	0b11	x	x	Msk	Msk	n/a	EL2
0b01	0b10	0	xx	x	x	Dis	Dis	Dis	Dis
0b01	0b10	1	xx	x	x	Dis	Dis	n/a	Dis
0b01	0b11	0	xx	0	x	Msk	Msk	EL2	EL2
0b01	0b11	0	xx	1	0	Msk	EL2	EL2	EL2
0b01	0b11	0	xx	1	1	Msk	Msk	EL2	EL2
0b01	0b11	1	xx	0	x	Msk	Msk	n/a	EL2
0b01	0b11	1	xx	1	0	Msk	EL2	n/a	EL2
0b01	0b11	1	xx	1	1	Msk	Msk	n/a	EL2
0b10	xx	0	xx	x	x	Dis	Dis	Dis	Dis
0b10	xx	1	xx	x	x	Dis	Dis	n/a	Dis
0b11	xx	0	xx	0	x	Msk	EL3	EL3	EL3
0b11	xx	0	xx	1	0	EL3	EL3	EL3	EL3
0b11	xx	0	xx	1	1	Msk	EL3	EL3	EL3
0b11	xx	1	xx	0	x	Msk	EL3	n/a	EL3
0b11	xx	1	xx	1	0	EL3	EL3	n/a	EL3
0b11	xx	1	xx	1	1	Msk	EL3	n/a	EL3

R_{KBPMJ} If the PMU Profiling exception is enabled and unmasked, and any of the following apply, then a PMU Profiling exception is taken asynchronously:

- An event counter <n> is implemented, the global enable control for the event counter is 1, PMINTENSET_EL1.P<n> is 1, and PMOVSSET_EL0.P<n> is 1.
- PMCR_EL0.E is 1, PMINTENSET_EL1.C is 1, and PMOVSSET_EL0.C is 1.
- FEAT_PMUv3_ICNTR is implemented, PMCR_EL0.E is 1, PMINTENSET_EL1.F0 is 1, and PMOVSSET_EL0.F0 is 1.

I_{GMXLL} The prioritization of the asynchronous PMU Profiling exception over other exceptions is IMPLEMENTATION DEFINED, as stated by R_{BKHSW} in Asynchronous exception types.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{DKCTL} An unmasked and enabled PMU Profiling exception taken asynchronously is taken in finite time.

R_{HFTPJ} When all the following are true for a value <n>, a PMU Profiling exception will be taken before instruction **C** completes execution:

- Instruction **A** does one of the following:
  - Directly reads PMOVSSET_EL0 or PMOVSCLR_EL0 and observes that bit <n> is 1.
  - Directly writes PMOVSSET_EL0 setting bit <n> to 1.
- A Context synchronization event **B** is in program order after **A**.
- **C** is the first instruction in program order after **B**.
- When **C** starts execution, the PMU Profiling exception is enabled and unmasked, and all other conditions for generating a PMU Profiling exception in R_{KBPMJ} relating to bit <n> apply:
  - If <n> is the index of an implemented event counter, then these are the conditions for event counter <n>.
  - If <n> is 31, then these are the conditions for the cycle counter PMCCNTR_EL0.
  - If FEAT_PMUv3_ICNTR is implemented and <n> is 32, then these are the conditions for the instruction counter PMICNTR_EL0.

  Other values of <n> are not applicable in this rule.

I_{KLVCH} A PMU Profiling exception is taken to the exception vector offset for non-interrupt exceptions at the target Exception level. This includes the asynchronous PMU Profiling exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.4 Attributability

An event caused by the PE counting the event is *Attributable*. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

An event is defined as being either Attributable or Unattributable. If the event is Attributable, it is further defined whether it is Attributable to:

- The current Security state of the PE.
- The current Exception level of the PE.
- When the PE is in Debug state, operations issued to the PE by the debugger through the external debug interface.

In a multithreaded implementation, an event might be generated by another PE with the same values for affinity level 1 and higher. This event is further defined as Attributable to:

- The current Security state of that PE.
- The current Exception level of that PE.
- When that PE is in Debug state, operations issued to that PE by the debugger through the external debug interface.

See Multithreaded implementations for information about enabling and restricting counting events in a multithreaded implementation.

Note

An implementation is described as multithreaded when the lowest level of affinity, as reported by MPIDR_EL1{Aff3, Aff2, Aff1, Aff0}, consists of logical PEs that are implemented using a multithreading type approach. In this section, when referring to a multithreaded implementation, *thread* is used to mean processing elements with:

- MPIDR_EL1.MT or MPIDR.MT set to 1,
- Different values for affinity level 0.
- The same values for affinity level 1 and higher.

An event can be defined as the combination of multiple subevents, which can be either Attributable or Unattributable.

All architecturally defined events are Attributable, unless otherwise stated.

Unattributable events might be counted when Attributable events are not counted. See:

- Interaction with EL3.
- Event filtering.
- Performance Monitors and Debug state.
- Counting events from shared components.

These sections are summarized by Table D13-2 for events Attributable to the processor, and Unattributable events. Table D13-2 entries apply when the counter and PMU are enabled and not frozen. Otherwise, events are not counted.

Table D13-2 Counting events

State	Allowed or prohibited	Filtered	Event type		
			If Attributable to:	Then	Else
Non- debug	Allowed	Not filtered	X	Count	Count
		Filtered	Current Exception level	Do not count	IMPLEMENTATION DEFINED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

State	Allowed or prohibited	Filtered	Event type		
			If Attributable to:	Then	Else
	Prohibited	X	Current Security state	Do not count	IMPLEMENTATION DEFINED
Debug	X	X	Debugger operations or raw cycles	Do not count	IMPLEMENTATION DEFINED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.5 Controlling the PMU counters

This section describes the mechanisms available for controlling the PMU event and cycle counters. The following sections describe those mechanisms:

- Enabling PMU counters.
- Freezing PMU counters.
- Prohibiting counting.

D13.5.1 Enabling PMU counters

ICFQVP Counters are enabled by a combination of the *global enable control* for the counter and a counter enable control, PMCNTENSET, as described in the following sections.

D13.5.1.1 Enabling the event counters

RKYBQW The global enable controls for PMU event counters are defined as follows:

- The global enable control for PMU event counters in the third range is PMCCR.EPME.
- The global enable control for PMU event counters in the second range is HDCR.HPME.
- The global enable control for PMU event counters in the first range is PMCR.E.

ISTZYQ PMCNTENSET.P<n> is the counter enable control for event counter *n*.

IHQHYY Table D13-3 shows the combined effect of the counter enable controls.

Table D13-3 Event counter enables

PMCCR.EPME	HDCR.HPME	PMCR.E	PMCNTENSET.P[n] == 0	PMCNTENSET.P[n] == 1		
				First range	Second range	Third range
0b0	0b0	0b0	Disabled	Disabled	Disabled	Disabled
0b0	0b0	0b1	Disabled	Enabled	Disabled	Disabled
0b0	0b1	0b0	Disabled	Disabled	Enabled	Disabled
0b0	0b1	0b1	Disabled	Enabled	Enabled	Disabled
0b1	0b0	0b0	Disabled	Disabled	Disabled	Enabled
0b1	0b0	0b1	Disabled	Enabled	Disabled	Enabled
0b1	0b1	0b0	Disabled	Disabled	Enabled	Enabled
0b1	0b1	0b1	Disabled	Enabled	Enabled	Enabled

IVQXCB The global enable control is used, with other controls, for all of the following:

- Enabling counting of events by PMEVCNTR<n>.
- Generation of the PMU overflow interrupt request, **PMUIRQ**, by PMOVSCLR[n].
- Generation of the CTI overflow interrupt trigger by PMOVSCLR[n].
- When FEAT_Debugv8p9 is implemented, generation of an External Debug Request, by PMOVSCLR[n].
- When FEAT_EBEP is implemented, generation of an asynchronous PMU Profiling exception, by PMOVSCLR_EL0[n].

The global enable controls for the other ranges do not affect whether the event counter PMEVCNTR<n> generates any of these.

IQKDTZ The effect of HDCR.{HPME, HPMN} on the counter enables applies at all Exception levels and in all Security states.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.5.1.2 Enabling the cycle counter

I_{BTGDJ} For the cycle counter, PMCCNTR, the global enable control is PMCR.E.

For the cycle counter, the counter enable control is PMCNTENSET.C.

PMCR.DP controls whether the cycle counter is disabled when event counting is prohibited, as described in Prohibiting counting.

D13.5.1.3 Enabling the instruction counter

I_{WMNXT} When FEAT_PMUv3_ICNTR is implemented, for the instruction counter, PMICNTR_EL0, the global enable control is PMCR.E and the counter enable control is PMCNTENSET_EL0.F0.

D13.5.2 Freezing PMU counters

D13.5.2.1 Freezing on PMU overflow

R_{FGSDP} If FEAT_PMUv3p7 is implemented, then all of the following apply:

- Affected counters do not count when PMCR.FZO is 1 and any of the following are true:
  - For any event counter <m> in the first range, PMOVSCLR.P<m> is 1.
  - FEAT_PMUv3_ICNTR is implemented, and PMOVSCLR_EL0.F0 is 1.
- All of the following counters are affected by this control:
  - Event counters PMEVCNTR<n> in the first range. This applies even when EL2 is disabled in the current Security state.
  - If FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0.
  - If PMCR.DP is 1, the cycle counter PMCCNTR.
- Other counters are not affected by this control. When PMCR.DP is 0, PMCCNTR is not affected by these controls.

R_{BHFSF} If FEAT_PMUv3p7 and EL2 are implemented, then all of the following apply:

- Affected counters do not count when HDCR.HPMFZO is 1 and any of the following are true:
  - For any event counter <m> in the second range, PMOVSCLR.P<m> is 1.
- All of the following counters are affected by this control:
  - Event counters PMEVCNTR<n>_EL0 in the second range. This applies even when EL2 is disabled in the current Security state.
- Other counters are not affected by this control.

I_{LKVKF} For any event counter PMEVCNTR<n>_EL0 in the third range, all of the following apply:

- The following controls do not affect whether the event counter PMEVCNTR<n>_EL0 counts events:
  - PMCR_EL0.FZO, MDCR_EL2.HPMFZO, and PMOVSCLR_EL0[31:0].
  - PMCR_EL0.FZS, MDCR_EL2.HPMFZS, PMBSR_ELx.S, and PMBLIMITR_EL1.{PMFZ, E}.
- PMOVSCLR_EL0[n] does not affect any of the following:
  - Whether any implemented event counter counts events.
  - Whether the cycle counter counts cycles.
  - Whether a BRBE freeze event occurs.
- Overflow of the event counter PMEVCNTR<n>_EL0 never generates the CHAIN, PMU_OVFS, or PMU_HOVFS PMU events.

R_{MZPBW} When the applicable PMCR_EL0.FZO or MDCR_EL2.HPMFZO bit is 1, it is CONSTRAINED UNPREDICTABLE whether any event happening at or about the same time as the event that caused the overflow is counted. This includes other instances of the same event.

I_{PPVQW} The architecture does not define when PMU events are counted relative to the instructions that caused the event. Events caused by an instruction might be counted before or after the instruction becomes architecturally executed, and events might be counted for operations that are not architecturally executed. Events can be counted speculatively, out-of-order, or both with respect to the simple sequential execution of the program. Events might also be counted simultaneously by other event counters when the overflow occurs, including events from different instructions. Multiple instances of an event might occur simultaneously, thus an event counter unsigned overflow can yield a nonzero value in the event counter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm recommends that such counting anomalies are minimized when software uses the freeze on overflow feature.

When the freeze on overflow feature is being used, software cannot assume that the event counter stops counting at zero when an overflow occurs.

I_{TLHSZ} If an event counter *n* overflows, where *n* is even and event counter *n*+1 is configured to count the CHAIN event, it is CONSTRAINED UNPREDICTABLE whether the CHAIN event observes the overflow event when the applicable PMCR_EL0.FZO or MDCR_EL2.HPMFZO bit is 1 and the corresponding PMCR_EL0.LP or MDCR_EL2.HLP bit is 0.

I_{LMSFT} If a direct read of PMOVSCLR_EL0 returns a nonzero value for a subset of the overflow flags, which means an event counter *n* should not count, then a sequence of direct reads of PMEVCNTR<n>_EL0 ordered after the read of PMOVSCLR_EL0 and before the PMOVSCLR_EL0 flags are cleared to zero, will return the same value for each read, because the event counter has stopped counting.

Note

Direct reads of System registers require explicit synchronization for following direct reads of other System registers to be ordered after the first direct read.

D13.5.2.2 Freezing on an SPE management event

R_{HSGZX} When all of the following are true, a PMU counter is frozen and does not count:

- FEAT_SPEv1p2 is implemented.
- PMBLIMITR_EL1.{PMFZ, E} is {1, 1}.
- SPE profiling is stopped. See Enabling profiling.
- One of the following applies:
  - The counter is an event counter PMEVCNTR<n>_EL0 in the first range and PMCR_EL0.FZS is 1.
  - The counter is an event counter PMEVCNTR<n>_EL0 in the second range and MDCR_EL2.HPMFZS is 1. This also applies when EL2 is disabled in the current Security state.
  - FEAT_PMUv3_ICNTR is implemented, the counter is the instruction counter PMICNTR_EL0, and PMCR_EL0.FZS is 1.
  - FEAT_SPE_DPFZS is implemented, the counter is the cycle counter PMCCNTR_EL0 and PMCR_EL0.{FZS, DP} is {1, 1}. If FEAT_SPE_DPFZS is not implemented, then the cycle counter is not affected by PMCR_EL0.FZS.

The pseudocode function `SPEFreezeOnEvent()` shows this.

See also Profiling Buffer management.

I_{HMLGP} If the highest implemented Exception level is using AArch32, then the Effective value of PMBLIMITR_EL1.E is 0 and FEAT_SPEv1p2 does not affect the PMU event counters. Otherwise, the effect of FEAT_SPEv1p2 on PMU event counters applies in AArch32 state.

D13.5.3 Resetting counters

Software can reset all the event counters and/or the cycle counter in a single operation by writing 1 to PMCR.C and/or PMCR.P.

If FEAT_PMUv3p9 is implemented, then counters can be individually reset together in a single operation using PMZR_EL0.

D13.5.4 Prohibiting counting

R_{WVDWZ} If EL3 is not implemented, the current Security state does not affect the counting of events.

R_{JRYNM} **If EL3 is implemented and FEAT_Debugv8p2 is not implemented**

Counting Attributable events is prohibited in Secure state unless any of the following is true:

- EL3 is using AArch32 and the value of SDCR.SPME is 1.
- EL3 is using AArch64 and the value of MDCR_EL3.SPME is 1.
- EL3 or EL1 is using AArch32, the PE is executing at EL0, and the value of SDER32_EL3.SUNIDEN is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Counting is permitted by an IMPLEMENTATION DEFINED authentication interface, and `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

Note

Software can read the Authentication Status register, DBGAUTHSTATUS to determine the state of an IMPLEMENTATION DEFINED authentication interface.

R_{JDWFF} **If EL3 is implemented and FEAT_Debugv8p2 is implemented and FEAT_PMUv3p7 is not implemented**

Counting Attributable events is prohibited in Secure state unless any of the following is true:

- EL3 is using AArch64 and the value of MDCR_EL3.SPME is 1.
- EL3 is using AArch32 and the value of SDCR.SPME is 1.
- EL3 or EL1 is using AArch32, the PE is executing at EL0, and the value of SDER32_EL3.SUNIDEN is 1.

R_{GNGCW} **If EL3 is implemented and FEAT_PMUv3p7 is implemented**

Counting Attributable events is prohibited at EL3 unless any of the following is true:

- EL3 is using AArch64 and the value of MDCR_EL3.{SPME, MPMX} is {1, 0}.
- EL3 is using AArch64, the value of MDCR_EL3.{SPME, MPMX} is {1, 1}, the event is being counted by an event counter <n> in the second range.
- EL3 is using AArch32 and the value of SDCR.SPME is 1.

Counting Attributable events is prohibited at EL2, EL1, and EL0 in Secure state unless any of the following is true:

- EL3 is using AArch64 and the value of MDCR_EL3.{SPME, MPMX} is not {0, 0}.
- EL3 is using AArch32 and the value of SDCR.SPME is 1.
- EL3 or EL1 is using AArch32, the PE is executing at EL0, and the value of SDER32_EL3.SUNIDEN is 1.

R_{MMZLZ} When EL2 is implemented, counting Attributable events is prohibited at EL2 unless any of the following is true:

- FEAT_PMUv3p1 is not implemented.
- HDCR.HPMD is 0.
- The event is being counted by an event counter <n> in the second range.

R_{PYPYY} If FEAT_SEL2 is implemented, counting Attributable events at Secure EL2 is allowed if and only if counting events is allowed in Secure state, and counting events is allowed at EL2.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when counting Attributable events is prohibited.

R_{TDGMF} The accessibility of Performance Monitors registers is unaffected by whether event counting is enabled or prohibited.

R_{CSRMB} All of the following apply to the instruction counter, PMICNTR_EL0:

- Counting is prohibited at EL3 unless any of the following is true:
  - EL3 is using AArch64 and the value of MDCR_EL3.{SPME, MPMX} is {1, 0}.
  - EL3 is using AArch64, the value of MDCR_EL3.{SPME, MPMX} is {1, 1}, and EL2 is implemented.
- Counting is prohibited at EL2 unless MDCR_EL2.HPMD is 0.
- Counting is prohibited at EL2, EL1, and EL0 in Secure state unless EL3 is using AArch64 and the value of MDCR_EL3.{SPME, MPMX} is not {0, 0}.

R_{GKPZH} The cycle counter, PMCCNTR, counts unless any of the following is true:

- The cycle counter is disabled by PMCR_EL0.E or PMCNTENSET_EL0.C.
- PMCR.DP is 1 and any of the following is true:
  - Event counting by event counters in first range is prohibited.
  - Event counting by event counters in first range is frozen by PMCR.FZO.
  - FEAT_SPE_DPFZS is implemented and event counting by event counters in first range is frozen by PMCR.FZS.

Note

When FEAT_HPMN0 is implemented and HDCR.HPMN is 0, the cycle counter is disabled by PMCR.DP under the same conditions that would prohibit or freeze event counting by event counters in the first range when HDCR.HPMN is not 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PE is in Debug state.
- FEAT_PMUv3p5 is implemented, EL3 is implemented, the PE is in Secure state, and SDCR.SCCD is 1.
- FEAT_PMUv3p5 is implemented, EL2 is implemented, the PE is executing at EL2, and HDCR.HCCD is 1.
- FEAT_PMUv3p7 is implemented, the PE is at EL3, EL3 is using AArch64, and MDCR_EL3.{SCCD, MCCD} is not {0, 0}.

R_{KQDQS} The following controls do not affect whether event counting is prohibited for the event counters in the third range:

- MDCR_EL3.SPME.
- MDCR_EL3.MPMX.
- MDCR_EL2.HPMD.

I_{LNLLM} Counters in the third range can observe all events in all Security states, including Secure state, and, when FEAT_RME is implemented, Root and Realm states, but these counters are only accessible to an observer that has read or write access to the Root PAS if FEAT_RME is implemented, or the Secure PAS otherwise.

However, using this feature does extend the attack surface to include to include these observers. For example, if the FEAT_RME is being used by an external agent to collect telemetry data that is passed to a less secure agent, this might be used to leak information about software executing in Root or Realm states.

Software executing on the PE is unaware that the counters are being used in this way.

The software that configures the counters in the third range should use the PMEVTYPER<n>_EL0 registers to configure the counters to not count in certain PE states such as Root or Realm or Secure states based on the trust level of agents that consume the event counter data.

I_{FZPQV} The `CountPMUEvents(n)` function returns TRUE if any of the following apply:

- PMEVCNTR<n> is enabled and allowed to count events at the current Exception level or state.
- n is 31 and the cycle counter is enabled and allowed to count cycles at the current Exception level and state.
- n is 32 and the instruction counter is enabled and allowed to count instructions at the current Exception level and state.
- Otherwise, the function returns FALSE.

However, this function do not completely describe the behavior for Unattributable events.

For more information, see `CountPMUEvents()` in A-profile Architecture Pseudocode.

I_{BTVLD} The Performance Monitors are intended to be broadly accurate and statistically useful, see Accuracy of the Performance Monitors. Some inaccuracy is permitted at the point of changing between a state where counting is prohibited and a state where counting is allowed, however. To avoid the leaking of information, the permitted inaccuracy is that transactions that are not prohibited can be uncounted. Where possible, prohibited transactions must not be counted, but if they are counted, then that counting must not degrade security.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.6 Multithreaded implementations

If an implementation is multithreaded and the Effective value of PMEVTYPER<n>.MT is 1, then, unless otherwise stated, the counter counts the sum of all events across all PEs with the same level 1 Affinity. A pair of PEs have the same level 1 Affinity if they have the same values for all fields in MPIDR_EL1or MPIDR except the Aff0 field.

See also Cycle event counting in multithreaded implementations for additional requirements for events that count cycles.

Events on other PEs are not counted when the Effective value of PMEVTYPER<n>.MT is 0.

If the CPU implements multithreading, and FEAT_MTPMU is not implemented, for Armv8.5 and earlier, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>.MT is implemented as RW or RES0. From Armv8.6, if the optional FEAT_MTPMU feature is not implemented, the Effective value of PMEVTYPER<n>.MT is RES0.

If FEAT_MTPMU is implemented, EL3 is implemented, and MDCR_EL3.MTPME is 0 or SDCR.MTPME is 0, FEAT_MTPMU is disabled and the Effective value of PMEVTYPER<n>.MT is 0.

If FEAT_MTPMU is implemented, EL3 is not implemented, EL2 is implemented, and MDCR_EL2.MTPME is 0 or HDCR.MTPME is 0, FEAT_MTPMU is disabled and the Effective value of PMEVTYPER<n>.MT is 0.

If FEAT_MTPMU is disabled on a Processing Element PEA, it is IMPLEMENTATION DEFINED whether FEAT_MTPMU is disabled on another Processing Element PEB, if all the following are true:

- FEAT_MTPMU is implemented on PEA and PEB.
- PEA and PEB have the same values for Affinity level 1 and higher.
- PEA and PEB both have MPIDR_EL1.MT or MPIDR.MT set to 1.

However, even when the Effective value of PMEVTYPER<n>.MT is 1, PEA does not count an event that is Attributable to Secure state on PEB if counting events Attributable to Secure state is prohibited on PEA. Similarly, PEA does not count an event that is Attributable to EL2 on PEB if counting events Attributable to EL2 is prohibited on PEA. See Counting events from shared components.

Example D13-1 The effect of having PMEVTYPER<*n*>.MT == 1

If the value of MDCR_EL3.SPME is 0, and *n* is less than PMCR.N on PEA, then event counter *n* on PEA does not count events Attributable to Secure state on PEB, even if one or both of the following applies:

- PEA is in Non-secure state.
- MDCR_EL3.SPME==1 on PEB.

Example D13-2 The effect of having PMEVTYPER<*n*>.MT == 1

When MDCR_EL2.HPMN is not 0, if the value of MDCR_EL2.HPMD is 1 and *n* is less than MDCR_EL2.HPMN on PEA, then event counter *n* on PEA does not count events Attributable to EL2 on PEB, even if one of the following applies:

- MDCR_EL2.HPMD==0 on PEB.
- PEA is not executing at EL2.

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to Secure state when PEA is in Secure state, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to Secure state when PEA is in Non-secure state is permitted.
- Counting Unattributable events related to other Secure operations in the system when PEA is in Non-secure state is permitted.

Otherwise, counting events in Non-secure state is permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to EL2 when PEA is at EL2, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to EL2 when PEA is using another Exception level is permitted.
- Counting Unattributable events related to EL2 when PEA is using another Exception level is permitted.

Otherwise, counting events at another Exception level is permitted.

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to a Security state and an Exception level when FEAT_PMUv3p7 is implemented, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to EL2 when PEA is using another Exception level is permitted.
- Counting Unattributable events related to EL2 when PEA is using another Exception level is permitted.

See Time as measured by the Performance Monitors cycle counter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.7 Event filtering

The PMU can filter events by various combinations of Exception level and Security state. This gives software the flexibility to count events across multiple processes.

D13.7.1 Filtering by Exception level and Security state

In AArch64 state:

- For each event counter, PMEVTYPER<n>_EL0 specifies the Exception levels in which the counter counts events Attributable to Exception levels.
- PMCCFILTR_EL0 specifies the Exception levels in which the cycle counter counts.
- If FEAT_PMUv3_ICNTR is implemented, PMICFILTR_EL0 specifies the Exception levels in which the instruction counter counts.

For an event that is Attributable to an Exception level, in a multithreaded implementation:

- When the Effective value of PMEVTYPER<n>_EL0.MT is 1, the specified filtering is evaluated using the current Exception level and Security state of the thread to which the event is Attributable. See Example D13-3.
- When the Effective value of PMEVTYPER<n>_EL0.MT is 0, the event is only counted if it is Attributable to the counting thread, and the filtering is evaluated using the Exception level and Security state of the counting thread.

Example D13-3

Example of the effect of the PMEVTYPER<*n*>_EL0.MT control

In a multithreaded implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1 and the value of PMEVTYPER<n>_EL0.U is 1 on the counting thread, then event counter *n* does not count events Attributable to EL0 on another thread, even if the counting thread is not executing at EL0.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether the filtering applies. In a multithreaded implementation, if the filtering applies to an Unattributable event, then the filtering is evaluated using the Exception level and Security state of the counting thread.

In AArch32 state, the filtering controls are provided by the PMEVTYPER<n> and PMCCFILTR registers.

For more information, see the individual register descriptions and Multithreaded implementations.

D13.7.2 Filtering by SVE Streaming mode

RYWWSV If FEAT_PMUv3_SME is implemented, then all of the following apply:

- For each event counter, PMEVTYPER<n>_EL0.VS specifies whether the counter counts events Attributable to Streaming SVE mode, Non-streaming SVE mode, or either mode.
- PMCCFILTR_EL0.VS specifies whether the cycle counter counts in Streaming SVE mode, Non-streaming SVE mode, or either mode.
- If FEAT_PMUv3_ICNTR is implemented, PMICFILTR_EL0.VS specifies whether the instruction counter counts in Streaming SVE mode, Non-streaming SVE mode, or either mode.

D13.7.3 Accuracy of event filtering

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. The following sections describe the cases where event counts must not be counted in the wrong state:

- Exception-related events.
- Software increment events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.7.3.1 Exception-related events

The PMU must filter events related to exceptions and exception handling according to the Exception level in which the event occurred. These events are:

- EXC_TAKEN, Exception taken.
- EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return.
- CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.
- TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to translation table base.
- EXC_UNDEF, Exception taken, other synchronous.
- EXC_SVC, Exception taken, Supervisor Call.
- EXC_PABORT, Exception taken, Instruction Abort.
- EXC_DABORT, Exception taken, Data Abort or SError.
- EXC_IRQ, Exception taken, IRQ.
- EXC_FIQ, Exception taken, FIQ.
- EXC_SMC, Exception taken, Secure Monitor Call.
- EXC_HVC, Exception taken, Hypervisor Call.
- EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally.
- EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally.
- EXC_TRAP_OTHER, Exception taken, other traps not Taken locally.
- EXC_TRAP_IRQ, Exception taken, IRQ not Taken locally.
- BRB_FILTRATE, Branch record captured.

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR after the return from the exception.

D13.7.3.2 Software increment events

The PMU must filter software increment events according to the Exception level in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally executed software increment event, and must not count any Speculatively executed software increment.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 or PMSWINC register without an intervening Context Synchronization event, then the counter is incremented twice.

For more information, see SW_INCR.

D13.7.4 Pseudocode description of event filtering

See `CountPMUEvents()` in A-profile Architecture Pseudocode for a pseudocode description of event filtering. However, this function does not completely describe the behavior for Unattributable events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.8 Event threshold and edge counting

I_{CJGPM} When FEAT_PMUv3_TH is implemented, software can program an event counter to only count when the event described by PMEVTYPER<n>_EL0 meets a threshold condition. The supported threshold conditions are:

- Less-than.
- Greater-than-or-equal-to.
- Not-equals.
- Equal-to.

I_{NJMGF} FEAT_PMUv3_EDGE is an optional extension to FEAT_PMUv3_TH that adds edge-detection logic to support counting threshold crossing events.

I_{VBVHX} FEAT_PMUv3_TH2 is an optional extension to FEAT_PMUv3_TH and FEAT_PMUv3_EDGE that adds additional controls for event counter linking.

R_{HVGCV} When FEAT_PMUv3_TH is implemented:

- If PMEVTYPER<n>_EL0.TC is not `0b000` or PMEVTYPER<n>_EL0.TH is nonzero, then threshold counting for event counter <n> is enabled.
- If PMEVTYPER<n>_EL0.TC is `0b000` and PMEVTYPER<n>_EL0.TH is zero, then threshold counting for event counter <n> is disabled.

R_{QVYDV} When FEAT_PMUv3_TH is not implemented, the Effective values of PMEVTYPER<n>_EL0.{TC, TH} are {`0b000`, 0}.

R_{GWTCZ} When FEAT_PMUv3_EDGE is implemented:

- If PMEVTYPER<n>_EL0.TE is 1, edge counting for event counter <n> is enabled.
- If PMEVTYPER<n>_EL0.TE is 0, edge counting for event counter <n> is disabled.

R_{SGLNM} When FEAT_PMUv3_EDGE is not implemented, the Effective value of PMEVTYPER<n>_EL0.TE is 0.

R_{HXHZC} When FEAT_PMUv3_TH2 is implemented, for each odd-numbered event counter <n>, PMEVTYPER<n>_EL0.TLC controls linking event counter <n> to event counter <n-1>.

R_{MRPVL} When FEAT_PMUv3_TH2 is not implemented or <n> is even, the Effective value of PMEVTYPER<n>_EL0.TLC is zero.

R_{JJRJQ} The values $V_B[n]$, $C_T[n]$, $V_T[n]$, $C_P[n]$, $C_E[n]$, $V_E[n]$, and $V[n]$ for an event counter <n> are defined as follows:

- $V_B[n]$ is the (base) value the counter increments by when threshold counting is disabled and edge counting is disabled for the event counter <n>.
- $C_T[n]$ is the result of evaluating the threshold condition. $C_T[n]$ is always one (TRUE) or zero (FALSE).
- $V_T[n]$ is the value that results from applying the threshold condition:
  - When FEAT_PMUv3_TH2 is not implemented or *n* is even, $V_T[n]$ is one of $V_B[n]$, one, or zero.
  - When FEAT_PMUv3_TH2 is implemented and *n* is odd, $V_T[n]$ is one of $V[n-1]$, $V_B[n]$, one, or zero.
- $C_P[n]$ is the value $C_T[n]$ took on the previous cycle if counting the event was allowed on the previous cycle, and zero otherwise. $C_P[n]$ is always one (TRUE) or zero (FALSE).
- $C_E[n]$ is the result of evaluating the edge condition. $C_E[n]$ is always one (TRUE) or zero (FALSE).
- $V_E[n]$ is the value that results from applying the edge condition:
  - When FEAT_PMUv3_TH2 is not implemented or *n* is even, $V_E[n]$ is always $C_E[n]$, that is, one (TRUE) or zero (FALSE).
  - When FEAT_PMUv3_TH2 is implemented and *n* is odd, $V_E[n]$ is one of $V[n-1]$, one, or zero.
- $V[n]$ is the value that results from applying both the threshold and edge conditions. $V[n]$ is one of $V_E[n]$, $V_T[n]$, or $V_B[n]$, depending on the configuration of counter <n>.

If an event counter <m> is disabled, then $V_B[m]$, $C_T[m]$, $V_T[m]$, and $V[m]$ are all zero.

I_{LQXJZ} For even-numbered event counters, PMEVTYPER<n>_EL0.TLC is RES0. For odd-numbered event counters, the following combinations of values are reserved:

- PMEVTYPER<n>_EL0.TLC[1:0] is `0b11`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMEVTYPER<n>_EL0.{TC[0], TE, TLC[1:0]} is {1, 0, `0b10`}.
- PMEVTYPER<n>_EL0.{TE, TLC[1:0]} is {1, `0b01`}.

For all event counters, the following combination is reserved:

- PMEVTYPER<n>_EL0.{TC[1:0], TE} is {`0b00`, 1}.

The effect of setting PMEVTYPER<n>_EL0.{TC, TE, TLC} to reserved values is CONSTRAINED UNPREDICTABLE. See Reserved values in System and memory-mapped registers and translation table entries.

RWXVGN

The threshold condition for event counter <n>, denoted by the value $C_T[n]$, is controlled by PMEVTYPER<n>_EL0.{TC, TH}.
$C_T[n]$ is defined as follows, where **TC** and **TH** are the Effective values of PMEVTYPER<n>_EL0.{TC, TH} respectively:

$$\text{threshold condition} = C_T[n] = \begin{cases} V_B[n] \neq \text{TH}, & \text{if TC[2:1]} = \texttt{0b00} \text{ (not equal).} \\ V_B[n] = \text{TH}, & \text{if TC[2:1]} = \texttt{0b01} \text{ (equal).} \\ V_B[n] \geq \text{TH}, & \text{if TC[2:1]} = \texttt{0b10} \text{ (greater-than-or-equal).} \\ V_B[n] < \text{TH}, & \text{if TC[2:1]} = \texttt{0b11} \text{ (less-than).} \end{cases}$$

IHNHNC

Example D13-4 Incrementing event counter *n* by $V_B[n]$ when $V_B[n]$ meets the threshold condition

When all of the following are true, the event counter *n* will increment by four:

- PMEVTYPER<n>_EL0.TC is `0b010`, equals, meaning threshold counting for event counter *n* is enabled.
- PMEVTYPER<n>_EL0.TH is 4.
- PMEVTYPER<n>.evtCount is `0x003F`, STALL_SLOT.
- There are exactly four operation Slots not occupied by an operation Attributable to the PE on the cycle.

IQLJJP

Example D13-5 Incrementing event counter *n* by 1 when $V_B[n]$ meets the threshold condition

When all of the following are true, the event counter *n* will increment by one:

- PMEVTYPER<n>_EL0.TC is `0b101`, greater-than-or-equal, count, meaning threshold counting for event counter *n* is enabled.
- PMEVTYPER<n>_EL0.TH is 2.
- PMEVTYPER<n>.evtCount is `0x80C1`, FP_FIXED_OPS_SPEC.
- At least two non-scalable floating-point operations are issued on the cycle. For example:
  - Two floating-point add instructions.
  - One floating-point multiply-add instruction

RNXPYM

The edge condition for event counter <n>, denoted by the value $C_E[n]$, is controlled by PMEVTYPER<n>_EL0.TC[0].
$C_E[n]$ is defined as follows, where **TC** is the Effective value of PMEVTYPER<n>_EL0.TC:

$$\text{edge condition} = C_E[n] = \begin{cases} C_P[n] \neq C_T[n], & \text{if TC[0]} = \texttt{0b0} \text{ (both edges).} \\ C_P[n] = 0 \wedge C_T[n] \neq 0, & \text{if TC[0]} = \texttt{0b1} \text{ (single edge).} \end{cases}$$

RXFGYS

When <n> is odd, the result of applying the threshold condition for event counter <n>, denoted by the value $V_T[n]$, is controlled by PMEVTYPER<n>_EL0.{TC, TLC}.
$V_T[n]$ for an odd counter <n> is defined as follows, where **TC** and **TLC** are the Effective values of PMEVTYPER<n>_EL0.{TC, TLC} respectively:

$$V_T[n \mid n \text{ is odd}] = \begin{cases} V_B[n], & \text{if TC[0]} = \texttt{0b0},\ \text{TLC[1]} = \texttt{0b0},\ \text{and } C_T[n] \text{ is TRUE.} \\ V[n-1], & \text{if TC[0]} = \texttt{0b0},\ \text{TLC[1:0]} = \texttt{0b10},\ \text{and } C_T[n] \text{ is TRUE.} \\ 1, & \text{if TC[0]} = \texttt{0b1},\ \text{TLC[1]} = \texttt{0b0},\ \text{and } C_T[n] \text{ is TRUE.} \\ V[n-1], & \text{if TLC[1:0]} = \texttt{0b01},\ \text{and } C_T[n] \text{ is FALSE.} \\ 0, & \text{otherwise.} \end{cases}$$

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{FLGYG} When <n> is odd, the result of applying the edge condition for event counter <n>, denoted by the value $V_E[n]$, is controlled by PMEVTYPER<n>_EL0.TLC.

$V_E[n]$ for an odd counter <n> is defined as follows, where **TLC** is the Effective value of PMEVTYPER<n>_EL0.TLC:

$$V_E[n \mid n \text{ is odd}] = \begin{cases} 1, & \text{if TLC[1:0]} = \texttt{0b00}, \text{ and } C_E[n] \text{ is TRUE.} \\ V[n-1], & \text{if TLC[1:0]} = \texttt{0b10}, \text{ and } C_E[n] \text{ is TRUE.} \\ 0, & \text{otherwise.} \end{cases}$$

Note

The input $V[n-1]$ from event counter <n-1> to the functions for event counter <n> in R_{XFGYS} and R_{FLGYG} is the *result* of applying the threshold and edge conditions to event counter <n-1> defined by R_{HBWBB}, and *not* the base value for event counter <n-1>, $V_B[n-1]$.

R_{YKFCB} When <n> is even, the result of applying the threshold condition for event counter <n>, denoted by value $V_T[n]$, is controlled by PMEVTYPER<n>_EL0.TC.

$V_T[n]$ for an even counter <n> is defined as follows, where **TC** is the value of PMEVTYPER<n>_EL0.TC:

$$V_T[n \mid n \text{ is even}] = \begin{cases} V_B[n], & \text{if TC[0]} = \texttt{0b0} \text{ and } C_T[n] \text{ is TRUE.} \\ 1, & \text{if TC[0]} = \texttt{0b1} \text{ and } C_T[n] \text{ is TRUE.} \\ 0, & \text{otherwise.} \end{cases}$$

R_{BGDTJ} When <n> is even, the result of applying the edge condition for event counter <n>, denoted by the value $V_E[n]$, is as defined as follows:

$$V_E[n \mid n \text{ is even}] = \begin{cases} 1, & \text{if } C_E[n] \text{ is TRUE.} \\ 0, & \text{otherwise.} \end{cases}$$

I_{HZBBG} R_{YKFCB} and R_{BGDTJ} are subsets of R_{XFGYS} and R_{FLGYG}, respectively, that apply when the Effective value of PMEVTYPER<n>_EL0.TLC is zero, and therefore also apply when FEAT_PMUv3_TH2 is not implemented, regardless of whether <n> is even or odd.

R_{HBWBB} The value of $V[n]$ for event counter <n> is defined as follows, where **TC**, **TH**, and **TE** are the Effective values of PMEVTYPER<n>_EL0.{TC, TH, TE} respectively:

$$V[n] = \begin{cases} V_B[n], & \text{if TC} = \texttt{0b000} \text{ and TH} = 0 \text{ (disabled).} \\ V_T[n], & \text{if (TC} \neq \texttt{0b000} \text{ or TH} \neq 0) \text{ and TE} = \texttt{0b0} \text{ (threshold).} \\ V_E[n], & \text{if TE} = \texttt{0b1} \text{ (edge).} \end{cases}$$

The pseudocode function `PMUCountValue()` in A-profile Architecture Pseudocode describes this.

I_{GNYJH}

Example D13-6 Using edge counting without threshold counting

To use edge counting without threshold counting, for example for a single-bit event (that is, V_B[n] is either 0 or 1):

- Set PMEVTYPER<n>_EL0.TH to 0.
- Set PMEVTYPER<n>_EL0.TE to 1.
- Set PMEVTYPER<n>_EL0.TC to:
  - `0b001` for a leading edge (zero to nonzero) count.
  - `0b011` for a falling edge (nonzero to zero) count.
  - `0b010` to count both edges.

I_{XWZRM} R_{XFGYS} and R_{FLGYG} describe the following permitted combinations for an event counter <n> when FEAT_PMUv3_TH2 is implemented and <n> is odd:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D13-4 Count value when PMEVTYPER<n>_EL0.TE is 0, FEAT_PMUv3_TH2 is implemented, and <n> is odd

	TLC	TC[0]	If $C_T[n]$ is TRUE, count by	If $C_T[n]$ is FALSE, count by
(A)	`0b00`	0	$V_B[n]$	0
(B)	`0b00`	1	1	0
(C)	`0b01`	0	$V_B[n]$	$V[n-1]$
(D)	`0b01`	1	1	$V[n-1]$
(E)	`0b10`	0	$V[n-1]$	0

Table D13-5 Count value when PMEVTYPER<n>_EL0.TE is 1, FEAT_PMUv3_TH2 is implemented, and <n> is odd

	TLC	TC[0]	If $C_E[n]$ is TRUE, count by	If $C_E[n]$ is FALSE, count by
(F)	`0b00`	1	1	0
(G)	`0b10`	1	$V[n-1]$	0

R_{YKFCB} and R_{BGDTJ} describe the following permitted combinations for an event counter <n> when FEAT_PMUv3_TH2 is not implemented or <n> is even:

Table D13-6 Count value when PMEVTYPER<n>_EL0.TE is 0, and FEAT_PMUv3_TH2 is not implemented or <n> is even

	TC[0]	If $C_T[n]$ is TRUE, count by	If $C_T[n]$ is FALSE, count by
(A)	0	$V_B[n]$	0
(B)	1	1	0

Table D13-7 Count value when PMEVTYPER<n>_EL0.TE is 1, and FEAT_PMUv3_TH2 is not implemented or <n> is even

	TC[0]	If $C_E[n]$ is TRUE, count by	If $C_E[n]$ is FALSE, count by
(F)	1	1	0

I_{GBTTP}

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D13-7 Using linking to count the logical combination of the inputs to two other counters

FEAT_PMUv3_TH2 allows the following linking functions to be defined, when using a threshold value of zero:

	Function	Equivalent function	Use
(1a)	$V_B[n]$ AND $V_B[n-1]$	`if V_B[n] == 0 then V[n-1] or 0`	(E)
(1b)	$V_B[n]$ AND $V_B[n-1]$	`if V_B[n] == 0 then V_B[n] else V[n-1]`	(C)
(2)	$V_B[n]$ OR $V_B[n-1]$	`if V_B[n] != 0 then 1 else V[n-1]`	(D)
(3)	$V_B[n]$ OR $V_B[n-1]$	`if V_B[n] != 0 then V_B[n] else V[n-1]`	(C)
(4)	NOT $V_B[n]$ AND $V_B[n-1]$	`if V_B[n] == 0 then V[n-1] else 0`	(E)
(5)	NOT $V_B[n]$ OR $V_B[n-1]$	`if V_B[n] == 0 then 1 else V[n-1]`	(D)

How event counter <n-1> is configured also affects the output, as the results for (C), (D), and (E) depend on $V[n-1]$, and not $V_B[n-1]$. For example:

- In examples (2) and (5), event counter <n-1> might need to be configured such that its threshold mechanism delivers $V[n-1]$ as zero or one. That is, using configuration (B).
- Event counter <n-1> might be configured so that $V[n-1]$ is zero when $V_B[n-1]$ is nonzero, and one otherwise.
- Threshold counting might be disabled on event counter <n-1> meaning the counter counts the base value for event counter <n-1> is counted when the threshold condition on event counter <n> is not met.

Other threshold values can be used to define richer functions. For example, increment the counter by $V_B[n-1]$ only if $V_B[n] \geq X$.

R_{MRLWN} Two PMU events are coincident if they are synchronous in the implementation, meaning they are counted at the same time. It is IMPLEMENTATION DEFINED which events are coincident.

Note

An implementation typically includes a pipeline for executing instructions. An instruction might generate events at any stage in the pipeline. Unless the implementation explicitly synchronizes these events, which is unlikely in the general case due to the implementation cost, these events would not be coincident despite being generated by the same instruction. Two events generated by different instructions might be coincident due to instruction-level parallelism.

I_{YTVPB} Linked PMU event counters can only be meaningfully used with coincident events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.9 PMU snapshots

IVZCDM The PMU snapshot extension provides a mechanism to take a snapshot of the PMU state at a given cycle on request.

RSCLCQ If all of the following are true, then a Capture event is generated:

- FEAT_PMUv3_SS is implemented.
- Capture events are enabled.
- Any of the following occur:
  - The value 1 is written to PMSSCR_EL1.SS.
  - The assertion of an IMPLEMENTATION DEFINED external snapshot request mechanism. This indirectly writes 1 to PMSSCR_EL1.SS.
- The Core power domain is powered on.

RMFJWS If Capture events are disabled, then PMSSCR_EL1.SS ignores writes.

RYWLFL If any of the following are true, then Capture events are disabled:

- EL3 is implemented and MDCR_EL3.PMSSE is `0b00`.
- All of the following are true:
  - EL3 is not implemented or MDCR_EL3.PMSSE is `0b01`.
  - EL2 is implemented.
  - MDCR_EL2.PMSSE is `0b00`.

  This applies even when EL2 is disabled in the current Security state.
- All of the following are true:
  - EL3 is not implemented or MDCR_EL3.PMSSE is `0b01`.
  - EL2 is not implemented or MDCR_EL2.PMSSE is `0b01`. This applies even when EL2 is disabled in the current Security state.
  - PMECR_EL1.SSE is `0b00`.

Otherwise, Capture events are enabled.

The pseudocode function `PMUCaptureEventEnabled()` shows this.

RTSYTY If all of the following are true, then Capture events are allowed:

- Capture events are enabled.
- The OS Lock is unlocked.
- Any of the following are true:
  - EL3 is implemented and MDCR_EL3.PMSSE is `0b11`.
  - All of the following are true:
    - EL3 is not implemented or MDCR_EL3.PMSSE is `0b01`.
    - EL2 is implemented.
    - MDCR_EL2.PMSSE is `0b11`.

    This applies even when EL2 is disabled in the current Security state.
  - All of the following are true:
    - EL3 is not implemented or MDCR_EL3.PMSSE is `0b01`.
    - EL2 is not implemented or MDCR_EL2.PMSSE is `0b01`. This applies even when EL2 is disabled in the current Security state.
    - PMECR_EL1.SSE is `0b11`.
- Either the PE is in Non-debug state or the PE allows Capture events in Debug state.

Otherwise, Capture events are prohibited.

It is IMPLEMENTATION DEFINED whether the PE allows Capture events in Debug state.

The pseudocode function `PMUCaptureEventAllowed()` shows this.

IMRCSY Table D13-9 summarizes the effect of the controls on PMU Capture events. In this table:

Disabled Means Capture events are disabled.

Prohibited Means Capture events are enabled and prohibited.

Allowed Means Capture events are enabled and allowed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D13-9 PMU Capture events

MDCR_EL3.PMSSE	MDCR_EL2.PMSSE	PMECR_EL1.SSE	Capture Events
0b00	xx	xx	Disabled
0b01	0b00	xx	Disabled
0b01	0b01	0b00	Disabled
0b01	0b01	0b10	Prohibited
0b01	0b01	0b11	Allowed
0b01	0b10	xx	Prohibited
0b01	0b11	xx	Allowed
0b10	xx	xx	Prohibited
0b11	xx	xx	Allowed

Note

When all of the following apply, an EL3 Secure or Root monitor switching to Secure state should set MDCR_EL2.PMSSE to 0b01:

- SCR_EL3.EEL2 is 0, disabling EL2 in Secure state.
- MDCR_EL3.PMSSE is 0b01, enabling use of PMU snapshot by lower Exception levels.
- MDCR_EL3.{MPMX, SPME} is not {0, 0}, meaning Secure state uses the PMU.

I_{WHYSQ}

This ensures that control over PMU snapshot moves to PMECR_EL1. In such a scenario, the EL3 monitor might have to rewrite other MDCR_EL2 fields. For example setting MDCR_EL2.HPMN to PMCR_EL0.N so that Secure EL1 has full control over the PMU counters. Alternatively, if Secure EL1 does not support PMU snapshot, then the EL3 monitor could set MDCR_EL3.PMSSE or MDCR_EL2.PMSSE to 0b00, and avoid also context switching PMECR_EL1.
If Secure state is not using the PMU (MDCR_EL3.SPME is 0), then the EL3 monitor should leave MDCR_EL2.PMSSE and PMECR_EL1 unchanged, so that external snapshot requests can be serviced.

R_{VPZHN}

If a Capture event occurs when allowed, then the PE writes all of the following snapshot registers:

- Event counters PMEVCNTR<n>_EL0 are written to PMEVCNTSVR<n>_EL1.
- Cycle counter PMCCNTR_EL0 is written to PMCCNTSVR_EL1.
- If FEAT_PMUv3_ICNTR is implemented, instruction counter PMICNTR_EL0 is written to PMICNTSVR_EL1.
- If FEAT_PCSRv8p9 is implemented and sampling on PMU snapshot Capture events is enabled by PMPCSCTL.SS, then all of the following:
  - If PC sampling is allowed and the PE is in Non-debug state, then all of the following:
    - The address of a recently-executed instruction and the execution state are written to PMPCSR.
    - If FEAT_PMUv3_EXT32 is implemented, then the Context ID registers CONTEXTIDR_EL2 and CONTEXTIDR_EL1 are written to PMCID2SR and PMCID1SR, respectively, and the Virtual machine identifier (VMID) is written to PMVIDSR.
    - If FEAT_PMUv3_EXT64 is implemented, CONTEXTIDR_EL2 and CONTEXTIDR_EL1 are written to PMCCIDSR, and VMID and CONTEXTIDR_EL1 are written to PMVCIDSR.

    In some circumstances, the address of a Recently-executed instruction is not available or UNKNOWN.
  - Otherwise, PMPCSR[31:0] is set to 0xFFFF_FFFF, and PMPCSR[63:32], PMCID1SR, PMCID2SR, PMVIDSR, PMCCIDSR, and PMVCIDSR are unchanged.

  If sampling on PMU snapshot is disabled then PMPCSR, PMCID1SR, PMCID2SR, PMVIDSR, PMCCIDSR, and PMVCIDSR are unchanged.

After these writes are completed, PMSSCR_EL1.{NC, SS} is set to {0, 0} to indicate that a Capture event has completed successfully.

R_{DBRSV}

PC sampling is allowed is defined by Controlling the PC Sample-based Profiling Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{TYNSW} *Recently-executed* means any instruction executed between the current and previous successful Capture events.

R_{WSVYM} The sampled recently-executed instruction address and context values follow the same rules and constraints described by The PC Sample-based Profiling Extension when PMPCSR is read with PMPCSCTL.SS set to 0. For example:

- When any of the following are true, the sampled instruction address and context might be unknown:
  - The PE is in reset state.
  - No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
  - No branch instruction has retired since the previous Capture event.
  - No branch instruction has retired since PMPCSCTL.SS was set to 1.
- When any of the following are true, it is permitted but not required for bits [31:0] of the sampled instruction address to be recorded as `0xFFFF_FFFF`, and for PMPCSR[63:32], PMCID1SR, PMCID2SR, PMVIDSR, PMCCIDSR, and PMVCIDSR to be unchanged:
  - The PE has not retired any branch instruction since leaving a state where PC Sample-based profiling was suspended.
  - The first Capture event after PMPCSCTL.SS was set to 1.

I_{YSWSD} The recently-executed instruction sampled in PMPCSR is captured before PMSSCR_EL1.{NC, SS} is set to {0, 0}.

I_{MXCMM} For a Capture event triggered by System register write, explicit synchronization is required between observing the Capture event has completed and reading the snapshot registers. That is, for the following sequence, instruction C observes the snapshot registers have been updated by the Capture event, when enabled:

1. An instruction A observes that a Capture event is complete.
2. Software executes a Context synchronization event B in program order after A.
3. An instruction C reads a snapshot register in program order after B.

Example D13-8 Synchronization between Capture event observation and snapshot register reads

The following sequence sets PMSSCR_EL1.SS to 1 to initiate a Capture event, and polls PMSSCR_EL1.SS to detect whether the Capture event has completed:

```
    ; Set PMSSCR_EL1.SS to initiate Capture event request
    MOV X0, #1
    MSR PMSSCR_EL1, X0
loop:
    ; Test PMSSCR_EL1.SS to check whether the Capture event has completed
    MRS X0, PMSSCR_EL1
    ISB
    TBNZ X0, #0, loop
    ; Read snapshot registers
```

R_{PLPCY} When the PE sets PMSSCR_EL1.{NC, SS} to {0, 0} on a successful Capture event:

- If the PE is in Non-debug state then the PMU_SNAPSHOT event is generated.
  This means that the PMU event is generated after any event counter that might be counting the event has been captured.
- If the PE is in Debug state, it is CONSTRAINED UNPREDICTABLE whether the PMU_SNAPSHOT event is generated.

I_{WRNBL} On a successful Capture event, a BRBE freeze event might occur. See R_{LBQZR} in Branch record buffer operation.

I_{HHHDQ} When PMPCSCTL.SS is 1, the architecture does not require any relationship between the PC value captured by PMPCSR and the branches captured in the frozen Branch records.

R_{MPZSB} If a Capture event occurs when prohibited, then:

- PMCCNTSVR_EL1, PMICNTSVR_EL1, and PMEVCNTSVR<n>_EL1 are unchanged.
- The PC sample registers are unchanged.
- PMSSCR_EL1.{NC, SS} is set to {1, 0} to indicate that a Capture event has completed unsuccessfully.
- Generation of Branch records is unaffected.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{KGJMV} Accesses to the snapshot registers are not affected by the OS Lock.

I_{LXYBX} A Capture event can occur when the PE is executing in AArch32 state.

R_{VLKCZ} The number of implemented PMU event counter snapshot registers, <n>, is IMPLEMENTATION DEFINED, and is the same as the number of implemented PMU event counters.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.10 Performance Monitors and Debug state

Events that count cycles are not counted in Debug state.

Events Attributable to the operations issued by the debugger through the external debug interface are not counted in Debug state.

In an implementation that supports multithreading, when the Effective value of PMEVTYPER<n>_EL0.MT is 1, if an event is Attributable to an operation issued by the debugger through the external debug interface to another thread that is in Debug state, then the event is not counted, and it is IMPLEMENTATION DEFINED whether the event is counted when the counting thread is in Debug state.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when the counting PE is in Debug state. If the event might be counted, then the rules in Filtering by Exception level and Security state apply for the current Security state in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.11 Counter access

IVYCGK All implemented event counters are accessible in EL3 and EL2. If EL2 is implemented the hypervisor uses HDCR.HPMN to reserve an event counter, with the effect that if EL2 is enabled in the current Security state, software cannot access that counter and its associated state from EL0 or EL1.

RSRPKH When FEAT_FGT is implemented, if PMSELR.SEL or *n* indicates an unimplemented event counter, access to PMXEVTYPER, PMXEVCNTR, PMEVTYPER<n>, or PMEVCNTR<n> is UNDEFINED.

ISDHCP Whether software can access an event counter at an Exception level does not affect whether the counter counts events at that Exception level. For more information, see Controlling the PMU counters and Enabling PMU counters.

D13.11.1 PMEVCNTR<*n*> event counters

DYBKKV Table D13-10 shows how the counter range affects the behavior of permitted accesses to the event counter registers.

Table D13-10 Result of event counter System register access

Counter range	EL2 enabled in current Security state	Access at Exception level			
		EL3	EL2	EL1	EL0
First range	-	Succeeds	Succeeds	Succeeds	Succeeds
Second range	False	Succeeds	Succeeds	Succeeds	Succeeds
Second range	True	Succeeds	Succeeds	No access	No access
Third range	-	No access	No access	No access	No access
Counter not implemented	-	No access	No access	No access	No access

IMWZHC Where Table D13-10 shows access succeeds for an event counter *n*, the access might be UNDEFINED or generate a trap exception. See the descriptions of PMEVCNTR<n> and PMXEVCNTR for details.

IDXWRB Where Table D13-10 shows no access for an event counter *n*, all of the following apply:

- When PMSELR.SEL is *n*, the PE prevents direct reads and direct writes of PMXEVTYPER or PMXEVCNTR.
- The PE prevents direct reads and direct writes of PMEVTYPER<n> and PMEVCNTR<n>.
- If FEAT_PMUv3_SS is implemented, then the PE prevents direct reads and direct writes of PMEVCNTSVR<n>_EL1.
- Direct reads and direct writes of the following register bits are RAZ/WI:
  - PMOVSCLR[*n*].
  - PMOVSSET[*n*].
  - PMCNTENSET[*n*].
  - PMCNTENCLR[*n*].
  - PMINTENSET[*n*].
  - PMINTENCLR[*n*].
  - PMOVSCLR[*n*].
  - PMOVSSET[*n*].
  - PMCNTENSET[*n*].
  - PMCNTENCLR[*n*].
  - When FEAT_PMUv3p9 is implemented, PMUACR_EL1[*n*].
- Direct writes to PMSWINC[*n*] are ignored.
- When FEAT_PMUv3p9 is implemented, direct writes to PMZR_EL0[*n*] are ignored.
- A direct write of 1 to PMCR.P does not reset PMEVCNTR<n>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.11.2 Cycle and instruction counters

The PMU does not provide any control that a hypervisor can use to reserve the cycle counter for its own use. If FEAT_FGT is implemented, then controls to trap EL1 and EL0 accesses of the cycle counter to EL2 are provided. However, access to the PMU registers are subject to the access permissions described in Configurable instruction controls.

PMICNTR_EL0 counts in both AArch32 and AArch64 states. However, PMICNTR_EL0 and PMICFILTR_EL0 can only be accessed as a System register in AArch64 state. If FEAT_FGT2 is implemented, then controls to trap EL1 and EL0 accesses of the instruction counter to EL2 are provided.

D13.11.3 EL0 access controls

I_{DXSHB} PMUSERENR controls EL0 access to the Performance Monitors. When FEAT_PMUv3p9 is implemented, PMUACR_EL1 provides fine-grained control over EL0 access to the Performance Monitors. PMUSERENR_EL0.UEN enables use of PMUACR_EL1. The behavior of other PMUSERENR_EL0 controls also depend on the value of PMUSERENR_EL0.UEN.

I_{KPPZZ} Table D13-11 summarizes the behavior of accesses to at EL0 to PMEVCNTR<n>, PMCCNTR, PMSWINC, and, if FEAT_PMUv3_ICNTR is implemented and the PE is executing in AArch64 state, PMICNTR_EL0. In this table:

UEN Means PMUSERENR_EL0.UEN. If FEAT_PMUv3p9 is not implemented, or EL1 is using AArch32, then the Effective value of this bit is 0.

EN Means PMUSERENR.EN.

xR/W Means:

- PMUSERENR.ER for PMEVCNTR<n>.
- PMUSERENR.CR for PMCCNTR.
- PMUSERENR_EL0.IR for PMICNTR_EL0.
- PMUSERENR_EL0.SW for PMSWINC.P<n>.

PMUACR_EL1.y Means:

- PMUACR_EL1.P<n> for PMEVCNTR<n>_EL0 and PMSWINC_EL0.
- PMUACR_EL1.C for PMCCNTR_EL0.
- PMUACR_EL1.F0 for PMICNTR_EL0.

This table assumes that all the following apply:

- Event counter <n> is implemented.
- EL2 is not implemented or disabled in the current Security state, or n is less-than HDCR.HPMN.
- FEAT_PMUv3_ICNTR is implemented.

Table D13-11 Summary of counter accesses at EL0

UEN	EN	xR/W	PMUACR_EL1.y	Access to PMEVCNTR<n> or PMCCNTR	Access to PMICNTR_EL0	Access to PMSWINC.P
0b0	0b0	0b0	x	UNDEFINED	UNDEFINED	UNDEFINED
0b0	0b0	0b1	x	RO	UNDEFINED	WO
0b0	0b1	x	x	R/W	UNDEFINED	WO
0b1	x	0b0	0b0	RAZ/WI	RAZ/WI	WI
0b1	x	0b1	0b0	RAZ/WI	RAZ/WI	WO
0b1	x	0b0	0b1	R/W	R/W	WO
0b1	x	0b1	0b1	RO	RO	WO

In Table D13-11, where the column Access to PMEVCNTR<n> or PMCCNTR or the column Access to PMICNTR_EL0 shows access is RO or RAZ/WI, this also includes ignoring writes to PMZR_EL0[y].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D13.12 Performance Monitors Extension registers

Further information on the Performance Monitors Extension Registers can be found in the following sections:

- Table K14-2 lists the Performance Monitors register names for AArch32 and AArch64 states.
- Performance monitors System registers summarizes the Performance Monitors Extension System registers in AArch64 state.
- Performance monitors System registers summarizes the Performance Monitors Extension System registers in AArch32 state.
- Memory-mapped Performance monitors registers summarizes the memory-mapped Performance Monitors Extension registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D14
PMU Event Descriptions

This chapter describes the PMU events that can be counted, their associated event numbers, and the mnemonics for the events. It contains the following sections:

- Definitions.
- The PMU event number space and common events.
- Common event numbers.
- Cycle event counting.
- Meaningful combinations of common events.
- Required events.
- IMPLEMENTATION DEFINED event numbers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.1 Definitions

The following subsections give more information about terms used in the event definitions:

- Definition of terms.
- Levels of caches and TLBs.
- Counting events from shared components.
- Tracking the Guarded Control Stack data accesses.
- Counting exceptions taken locally or not taken locally.

D14.1.1 Definition of terms

This section describes terms used by PMU events as they relate to the implementation of the PMU. For more definitions, see Glossary.

Architectural event

An architectural event is an event which gives the same result for the same program on any implementation of the Arm architecture, subject to the program having the same inputs, including asynchronous events triggered by the system, any IMPLEMENTATION DEFINED or UNPREDICTABLE variation permitted by the event definition, and the reasonable degree of inaccuracy described in A reasonable degree of inaccuracy. INST_RETIRED is an example of an architectural event.

An architectural event should not be confused with an event defined by the architecture, which is referred to as a Common event.

Note: Do not confuse Architectural events and Common events. An Architectural event can be either Common or IMPLEMENTATION DEFINED. See also Common event and IMPLEMENTATION DEFINED event.

At-retirement event

An at-retirement event is generated by an Architecturally executed instruction that does not generate an exception, unless specified otherwise. This includes SVC, HVC, and SMC exceptions.

For cache and TLB events:

- The event counts each architecturally-executed instruction that generates the event.
- The event is only counted once, by the first instruction that causes it. This might not be the first instruction in program order.

  Note

  Consider two loads that execute out of order that access the same cache line granule which is not in the cache. The second instruction in program order causes the miss, and the first instruction fetches data from the same cache miss result. Only the second instruction counts the cache miss event. Also, the second instruction might be speculative and the first instruction retired, meaning the cache miss is not counted at all.

- The population counts for the events are the architecturally-executed instruction counts.

Common event

An event that is defined by the Performance Monitors Extension. For more information, see Common event numbers.

Note: Do not confuse Common events and Architectural events. A Common event can be either Architectural or Microarchitectural. See also Architectural event and Microarchitectural event.

CSIZE

Container size, in bits, that corresponds to the largest non-overlapping SVE or Advanced SIMD vector element size or scalar register size that is encoded in the instruction opcode. This excludes the 64-bit elements of the wide element variants of the SVE bitwise shift and integer compare instructions that overlap the narrower source and destination elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Event in progress

Some events count when another event or condition is *in progress*. This might mean that the event counts the occupancy of a queue or other microarchitectural structure tracking the event. It is usually IMPLEMENTATION DEFINED when an event is in progress.

For example, the MEM_ACCESS_RD_PERCYC event counts when the MEM_ACCESS_RD event is in progress, meaning on each Processor cycle, the counter increments by the number of Memory-read operations that are in progress.

These events can be used to calculate the average number of events in progress. In the case of the MEM_ACCESS_RD_PERCYC event, this is also the average read latency. However, the definition of *in progress* might not include all parts of the operation.

Example D14-1

Stages of an operation not included in the definition of event in progress

In an example implementation, a Memory-read operation generated by a load instruction occupies three pipeline stages in the PE before generating a MEM_ACCESS_RD event when the PE starts to access memory. The event is then considered to be *in progress* until the data is returned to the PE. In the case of a Normal Cacheable access, the PE first looks in the Level 1 data cache, and if the address is cached, returns data in two cycles. Once the data is returned, it is another cycle before the result can be forwarded to any other instruction.

In this example, if all loads hit in the Level 1 cache, the average read latency calculated using the MEM_ACCESS_RD_PERCYC and MEM_ACCESS_RD events might be two cycles. Although this value is correct for the IMPLEMENTATION SPECIFIC definition of this event, it has to be adjusted by a constant four additional cycles to match the more commonly understood definition of Level 1 cache access latency, which for this example would be quoted as six cycles.

Instruction architecturally executed

An instruction that is part of the Execution stream. See also Architecturally executed.

A reasonable degree of inaccuracy allows for counts to be inaccurate in exceptional circumstances. For an event that counts instructions architecturally executed, this allows an implementation to count instructions that do not form part of the Execution stream because of an exceptional event, such as if the instruction generates a synchronous exception or entry to Debug state.

Instructions that have no visible effect on the architectural state of the PE are architecturally executed and counted even if they form part of the Execution stream.

Examples of instructions that have no visible effect are:

- A `NOP`.
- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.

However, for events that count only the execution of instructions that update System registers, such as CID_WRITE_RETIRED, if such an instruction is executed twice without an intervening Context Synchronization event, it is constrained unpredictable whether the first instruction is counted.

Instruction architecturally executed, Condition code check pass

Instruction architecturally executed, Condition code check pass is a class of events that explicitly do not occur for:

- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.
- A Test and Branch on Zero, `TBZ`, instruction that does not branch.
- A Test and Branch on Nonzero, `TBNZ`, instruction that does not branch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A Store-Exclusive instruction that does not write to memory.

Otherwise, the definition of architecturally executed is the same as for Instruction architecturally executed.

A branch that is architecturally executed, with condition code check pass is also described as a branch taken.

Instruction memory access

A PE acquires instructions for execution through instruction fetches. Instruction fetches might be due to:

- Fetching instructions that are architecturally executed.
- The result of the execution of an instruction prefetch instruction, `PLI`.
- Speculation that a particular instruction might be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is IMPLEMENTATION DEFINED. For example, an implementation might fetch many instructions including a non-integer number of instructions in a single instruction memory access.

IMPLEMENTATION DEFINED event

An event that is defined by the implementation. For more information, see IMPLEMENTATION DEFINED event numbers.

Note: Do not confuse IMPLEMENTATION DEFINED events and Microarchitectural events. An IMPLEMENTATION DEFINED event can be either Architectural or Microarchitectural. See also Architectural event and Microarchitectural event.

Memory-read operations

A PE accesses memory through Memory-read operations and Memory-write operations. A Memory-read operation might be due to:

- The result of an architecturally executed Memory-reading instruction.
- The result of a Speculatively executed Memory-reading instruction.
- A translation table walk.
- A GCS data read operation.

  Note

  A GCS data read operation is considered a Memory-read operation regardless of whether L1GCS_CACHE event is implemented.

- An Instruction memory access.
- For some events, the execution of a data preload instruction or hardware speculation that a future instruction might access the memory location.

For levels of cache hierarchy beyond the Level 1 caches for some events, Memory-read operations also include accesses made as part of a refill of another cache closer to the PE. Such refills might be due to:

- Memory-read operations or Memory-write operations that miss in the other cache.
- The execution of a cache maintenance instruction.

  Note

  A cache maintenance instruction is not, in itself, an access to that cache. However, it might generate cache refills which are then treated as memory-read operations beyond that cache.

This list is not exhaustive.

The relationship between memory-read instructions and memory-read operations is IMPLEMENTATION DEFINED. For example, for some implementations an `LDP` instruction that reads two 64-bit registers might generate one memory-read operation if the address is quadword-aligned, but for other addresses it generates two or more memory-read operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Memory-reading instructions

An instruction with explicit Memory read effects, including Tag-read effects. Loads and atomic instructions that return a value to the PE are counted as Memory-reading instructions. `MRS` and `MRRS` instructions transformed to memory reads by FEAT_NV2 are counted as Memory-reading instructions.

Unless otherwise specified, this includes instructions that can have a Memory read effect, even if the instruction does not actually read from memory for some other reason. For example, a conditional load in AArch32 state that fails its condition code, an SVE predicated load with inactive predicates, or a load that generates an MMU fault.

It is IMPLEMENTATION DEFINED whether the prefetch instructions `PRFM`, `PLD`, `PLDW`, and `PLI` count as integer data-processing instructions or Memory-reading instructions. Arm recommends that if the instruction is not implemented as a NOP then it is counted as a Memory-reading instruction.

Memory-writing instructions, other than atomics, that return a result in a PE register other than data from the location being accessed are not counted as Memory-reading instructions. This includes Store-exclusive, `ST64BV`, and `ST64BV0`.

Atomic instructions that do not return a value to the PE are not counted as Memory-reading instructions.

Instructions with only a Tag-Check-read effect are not counted as Memory-reading instructions.

Memory-write operations

Memory-write operations might be due to:

- The result of an architecturally executed Memory-writing instructions.
- The result of a Speculatively executed Memory-writing instructions.
- A hardware update of a translation table entry.
- A GCS data write operation.

  Note

  A GCS data write operation is considered a Memory-write operation regardless of whether L1GCS_CACHE event is implemented.

Note

Speculatively executed Memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a Memory-write operation that is later undone in some IMPLEMENTATION SPECIFIC way.

For levels of cache hierarchy beyond the Level 1 caches for some events, Memory-write operations also include accesses made as part of a write-back from another cache closer to the PE. Such write-backs might be due to:

- Evicting a dirty line from the cache, to allocate a cache line for a cache refill, see Memory-read operations.
- The execution of a cache maintenance instruction.

  Note

  A cache maintenance instruction is not in itself an access to that cache. However, it might generate write-backs which are then treated as Memory-write operations beyond that cache.

- The result of a coherency request from another PE.

This list is not exhaustive.

The relationship between Memory-writing instructions and Memory-write operations is IMPLEMENTATION DEFINED. For example, for some implementations an `STP` instruction that writes two 64-bit registers might generate one Memory-write operation if the address is quadword-aligned, but for other addresses it generates two or more Memory-write operations. In some implementations, the result of two `STR` instructions that write to adjacent memory might be merged into a single Memory-write operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The data written back from a cache that is shared with other PEs might not be data that was written by the PE that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that PE.

Memory-writing instructions

Are instructions with explicit Memory write effects, including Tag-write effects. Stores and atomic instructions are counted as Memory-writing instructions.

Unless otherwise specified, this includes instructions that can have a Memory write effect, even if the instruction does not actually write to memory for some other reason. For example, a conditional store in AArch32 state that fails its condition code, an SVE predicated store with inactive predicates, a `CAS` or Store-exclusive that does not update memory, or a store that generates an MMU fault.

`DC ZVA`, `DC GVA`, and `DC GZVA` are counted as Memory-writing instructions. `MSR` and `MSRR` instructions transformed to memory writes by FEAT_NV2 are counted as Memory-writing instructions.

Microarchitectural event

A microarchitectural event is any event which is not architectural. That is, it will give different results for the same program on two different implementations of the Arm architecture, due to microarchitectural differences in the implementations. L1D_CACHE_REFILL is an example of a microarchitectural event.

A microarchitectural event should not be confused with an event defined by the implementation, which is referred to as an IMPLEMENTATION DEFINED event.

Note: Do not confuse Microarchitectural events and IMPLEMENTATION DEFINED events. A Microarchitectural event can be either Common or IMPLEMENTATION DEFINED. See also Common event and IMPLEMENTATION DEFINED event.

MSIZE

Memory element access size, in bits, that corresponds to a load or store instruction mnemonic suffix, where B=8, H=16, W=32 and D=64. When an instruction mnemonic does not end with B, H, W or D, the memory access size is implied by the scalar transfer register size or SIMD transfer register element size.

Non-SIMD SVE and SME instructions

These instructions are:

- Scalar vector length calculation instructions, ADDPL, ADDSPL, ADDSVL, ADDVL, RDSVL, and RDVL.
- Scalar predicate count instructions that determine a number of elements implied by a named predicate constraint, and use that value to calculate a result written to a general-purpose register. See Predicate count for more information.
- Compare and terminate instructions, CTERMEQ, CTERMNE.
- `MRS` and `MSR` instructions for accessing SVE and SME Special-purpose and System registers, including SMSTART and SMSTOP.

Operation

An event that counts operations counts an IMPLEMENTATION DEFINED choice between:

- Architectural instructions.
- IMPLEMENTATION SPECIFIC microarchitectural operations (micro-ops).

An architectural instruction might create one or more microarchitectural operations (micro-ops) at any point in the execution pipeline. Depending on the event definition, the micro-ops might be counted instead of instructions. The definition of a micro-op is IMPLEMENTATION SPECIFIC. An architecture instruction might create more than one micro-op for each instruction. A micro-op might also be removed or merged with another micro-op in the execution stream, so an architecture instruction might

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

create no micro-ops for an instruction. Any arbitrary translation of instructions to an equivalent sequence of micro-ops is permitted.

Microarchitectural operations might have different characteristics from the architectural instructions they are created from. For events defined with reference to a particular instruction or type of instruction, the PE counts events appropriate to the micro-op.

This means the relationship between a micro-op and an architecturally executed instruction is IMPLEMENTATION DEFINED, and an architecture instruction of one type might be counted as one or more micro-ops of a different type.

However, in order for the profile information to be meaningful to the user of the counter information, Arm recommends that the micro-op is counted as a type appropriate to the instruction when this is appropriate and feasible.

Example D14-2 Counting micro-ops

If an architectural load instruction is split into an address generation micro-op and a load micro-op, and the PMU counts micro-ops, then when counting the operations:

- The instruction is counted as two operations.
- The address generation micro-op is counted as a data-processing operation.
- The load micro-op is counted as a Memory-reading operation.

Example D14-3 Counting SIMD micro-ops

If the PMU counts micro-ops as operations, then a micro-op generated by an SIMD SVE instruction might be counted as an Advanced SIMD operation when all of the following are true:

- The effective vector length for the micro-op is 128 bits.
- The operation is not predicated, and does not have a predicate register as an input or output.
- The operation has an equivalent Advanced SIMD operation.

Operation speculatively executed

An Operation that is Speculatively executed.

There is no architecturally guaranteed relationship between a Speculatively executed micro-op and an architecturally executed instruction.

The results of such an operation can also be discarded, if it transpires that the operation was not required, such as following a mispredicted branch. Therefore, the architecture defines these events as *operations speculatively executed*, where appropriate.

Many PMU events count operations that are speculatively executed. This means that the PMU can observe and record the behavior of operations executed under the effects of speculation. However, the PMU must not count an event that might be used to infer values that cannot be accessed or used by the operation, as defined by Restrictions on the effects of speculation.

For example:

- An address that is not permitted to be used by the operation might be inferred by whether it is present in a cache or TLB.
- A condition code that is not permitted to be used by the operation can be inferred by whether a branch was taken or not taken.
- A predicate value that is not permitted to be used by the operation can be inferred by whether a predicate value is empty or not.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In some events, operation has a more specific meaning described in the event. See SIMD SVE and SME instructions.

Processor cycle

For a non-multithreaded implementation, this means a cycle of the processor. For a multithreaded implementation, processor cycle means each cycle of the multithreaded processor, not just those cycles for which the PE counting the event is active.

SIMD SVE and SME instructions

These instructions are:

- Instructions operating on one-dimensional vectors, reading from or writing to SVE scalable vector registers, SVE scalable predicate registers, or SME ZA array vector slices.
- Instructions operating on two-dimensional tiles, reading from or writing to SME ZA tiles.

Data processing instructions operating on one dimensional scalable vectors are counted by counters counting FP*_SCALE_OPS_SPEC and INT_SCALE_OPS_SPEC events, and increment the counter by a value v such that ($v \times$(VL÷128)) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

Load and store instructions operating on scalable vectors or the ZA array are counted by counters counting (LDST|LD|ST)_SCALE_OPS_SPEC and (LDST|LD|ST)_SCALE_BYTES_SPEC events, and increment the counter by a value v such that:

- For the *_OPS_SPEC events, ($v \times$(VL÷128)) is number of vector elements loaded or stored by the operation or instruction which causes the counter to increment. That is, v is (128÷CSIZE), multiplied by the number of transferred registers.
- For the *_BYTES_SPEC events, ($v \times$(VL÷128)) is number of bytes loaded or stored by the operation or instruction which causes the counter to increment. That is, v is (16÷(CSIZE÷MSIZE)), multiplied by the number of transferred registers.

This does not include load instructions which load a single element and replicate it across a one dimensional scalable vector.

Data processing instructions operating on two dimensional scalable tiles are counted by counters counting *_SCALE2_OPS_SPEC events, and increment the counter by a value v such that ($v \times$(VL÷128)^2) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

Note

An implementation might increment the counter once by v for the whole instruction, or, if the instruction is executed as a sequence of micro-ops multiple times by some fraction of v for each micro-op, such that it increments by v in total.

In addition all of the following apply:

- Events of the form <TYPE>_SCALE{2}_OPS_SPEC will increment a counter by v only if the instruction performs calculations on a structure with elements of type <TYPE>.
- Events of the form <TYPE>_<PRECISION>_SCALE{2}_OPS_SPEC will increment a counter by v only if the largest precision operand of type <TYPE> read or written by the instruction is <PRECISION>.
- Events of the form <TYPE>_<PRECISION>_SCALE{2}_MIN_OPS_SPEC will increment a counter by v only if the smallest precision operand of type <TYPE> read or written by the instruction is <PRECISION>.

Slot

An implementation of a PE might be able to execute multiple micro-ops in a single processor cycle. The maximum number of micro-ops that can be executed might vary at different points in the execution pipeline.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To allow profiling of the utilization of the resource of the PE, an IMPLEMENTATION SPECIFIC point in the execution pipeline is chosen where the maximum number of micro-ops that can be executed is an IMPLEMENTATION DEFINED fixed value.

Each possible micro-op that can be executed at that point in a cycle is called a Slot. The maximum number of micro-ops that can be executed is defined by PMMIR.SLOTS.

Software change of the PC

All of the following are treated as a software change of the PC:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

It is IMPLEMENTATION DEFINED whether any or all of the following are treated as software changes of the PC:

- `BRK` and `BKPT` instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, `SVC`, `HVC`, and `SMC`.
- Context synchronization barrier `ISB` instructions.

Speculatively executed

An instruction or operation that is counted by an event when it might be Speculative.

The architecture does not define the point in a pipeline where an event is counted. For some events, this means the operation or instruction is counted when the operation or instruction is Speculative. The results of such an operation or instruction might later be discarded, if it transpires that the operation was not required, such as following a mispredicted branch, or might be later resolved to be Architecturally executed.

Different groups of events might be counted at different points in the pipeline and so can have different IMPLEMENTATION DEFINED definitions of speculatively executed. Such groups share a common base type, which the event name denotes. Each of the events in the previous example is of the base type, operation speculatively executed.

For groups of events with a common base type, speculatively executed operations are all counted on the same basis, which normally means at the same point in the pipeline. It is possible to compare the counts and make meaningful observations about the program being profiled.

Taken locally

Taken locally is a qualifier that determines which instances of an exception are counted by particular PMU events. See Counting exceptions taken locally or not taken locally.

In this context, an exception that is *Taken locally* means an exception that is one of:

- Taken to the current Exception level.

  Note

  This is not possible when the current Exception level is EL0.

- Taken from EL0 to EL1.
- Taken from EL0 to EL2 because the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

  Note

  An exception taken from EL0 to EL2 because the Effective value of HCR_EL2.{E2H, TGE} is {0, 1} is not *Taken locally*. This includes exceptions taken to EL2 using AArch32 when HCR.TGE is 1.

VL

The current SVE vector length, in bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.1.2 Levels of caches and TLBs

The mapping of levels of cache and TLB to the PMU events is IMPLEMENTATION DEFINED. Although CLIDR_EL1 and CLIDR define the implemented levels of cache, these are not required to correspond with the levels of cache defined for PMU events. The architecture does not provide any way of determining implemented levels of TLB. Also, many implementations include structures that provide some caching at a higher level than the level 1 caches or TLBs. Typically, these structures, that might be called Level 0 caches, or mini caches, or microcaches, are invisible to software. The IMPLEMENTATION SPECIFIC nature of cache and TLB implementations mean that, in general, PMU event counts cannot be used reliably to make direct comparisons between different implementations, and Arm recommends the following implementation guidelines:

- If L3D_CACHE events are implemented, then L2D_CACHE and if applicable L2I_CACHE events should be implemented.
- If L2D_CACHE events are implemented, then L1D_CACHE and if applicable L1I_CACHE events should be implemented.
- If L2I_CACHE events are implemented, then L1I_CACHE events should be implemented.
- Where the Last Level of cache is also the Level 3 or Level 2 unified cache, the LL_CACHE events should be implemented in preference to the L3D_CACHE or L2D_CACHE events as applicable.
- For the Level <n> cache, where <n> = 1 or <n> = 2:
  - If the Level <n> cache is unified, but the cache can disambiguate between Data and Instruction accesses to the cache, then both the L<n>D_CACHE and L<n>I_CACHE events should be implemented.
  - If the cache is unified and the cache cannot disambiguate between Data and Instruction accesses, then only the L<n>D_CACHE should be implemented, counting all accesses.

This final property is IMPLEMENTATION DEFINED.

If an implementation uses separate caching structures to cache the GCS data and translations, the implementation might be capable of separately tracking the Guarded Control Stack data accesses and their translations in the caches and TLBs. See Tracking the Guarded Control Stack data accesses for more information.

N1_*, N2_*, N3_*, and N4_* events are defined for identifying the *non-uniform memory access* (NUMA) distance for data returned directly from caches and memory, with support for:

- Up to four NUMA distance levels.
- Up to three cache types at each level.
- Up to three memory types at each level.
- Cache snoop responses at each level.

The NUMA distance indicates the relative distance that the return data has traveled. The NUMA distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have different levels of hierarchy. The interpretation of NUMA distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system.

The mapping of NUMA distance values to LOCAL and REMOTE events is IMPLEMENTATION DEFINED.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be broadly similar for all devices in the system.

D14.1.3 Counting events from shared components

There is no architectural concept of a *shared* component. However, when a cache, a bus, or any other system component that might generate countable events is implemented, and:

- The extent of the first-order effects due to an event from that component are only applicable to a single PE, then the event is not shared.
- Otherwise, the event is shared.

Second-order effects are not considered when determining if an event is shared.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is `0b0`, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is `0b1`, the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

If an implementation uses shared caching structures for GCS data and translations, the implementation might be capable of separately tracking the Guarded Control Stack data accesses and their translations in the caches and TLBs. See Tracking the Guarded Control Stack data accesses for more information.

Example D14-4 First and second order effects of a cache miss in a multiple-PE implementation

In an implementation that consists of two PEs, each with its own L1 cache, a cache miss by one of the PEs is a first-order effect of an access to its cache. Any snoop that is performed on the L1 cache of the other PE in the implementation as a result of that cache miss is a second order effect.

Note

Shared events are inherently linked to microarchitectures and so the implementer must make an informed decision about how such events are implemented.

D14.1.4 Tracking the Guarded Control Stack data accesses

All of the following PMU events are GCS PMU events:

- L1GCS*.
- GCSTLB*.
- STALL_*_GCS*.

If an implementation is capable of separately tracking the Guarded Control Stack data accesses and their translations in the caches and TLBs, all of the following apply:

- The GCS PMU events are permitted to be implemented and allow software to disambiguate the performance aspects of Guarded Control Stack data accesses from other data accesses.
- L1GCS_CACHE* events and L1GCS_TLB* events are generated for Guarded Control Stack data accesses.
- L1D_CACHE* events and L1D_TLB* events are not generated for Guarded Control Stack data accesses.

If an implementation is not capable of separately tracking the Guarded Control Stack data accesses and their translations in the caches and TLBs, all of the following apply:

- The GCS PMU events are not implemented, and software is not permitted to disambiguate the performance aspects of Guarded Control Stack data accesses from other data accesses.
- L1GCS_CACHE* events and L1GCS_TLB* are not generated for Guarded Control Stack data accesses.
- L1D_CACHE* events and L1D_TLB* events are generated for Guarded Control Stack data accesses.

D14.1.5 Counting exceptions taken locally or not taken locally

Table D14-1 shows the events for exceptions taken to an Exception level using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D14-1 Events for exceptions taken to an Exception level using AArch64

ESR.EC	Description		Event number and classification for exceptions	
			Taken locally	Not Taken locally
0x00	Unknown or uncategorized		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x01	WF* traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x03	AArch32 MCR/MRC traps on (coproc == 0b1111) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x04	AArch32 MCRR/MRRC traps on (coproc == 0b1111) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x05	AArch32 MCR/MRC traps on (coproc == 0b1110) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x06	AArch32 LDC/STC traps on (coproc == 0b1110) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x07	Advanced SIMD or FP traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x08	AArch32 MVFR* and FPSID traps		-	0x008D, EXC_TRAP_OTHER
0x0C	AArch32 MCRR/MRRC traps on (coproc == 0b1110) accesses		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x0E	Illegal instruction set state		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x11	AArch32 SVC		0x0082, EXC_SVC	0x008D, EXC_TRAP_OTHER
0x12	AArch32 HVC that is not disabled		-	0x008A, EXC_HVC
0x13	AArch32 SMC that is not disabled	to EL2	-	0x008D, EXC_TRAP_OTHER
		to EL3	-	0x0088, EXC_SMC
0x15	AArch64 SVC		0x0082, EXC_SVC	0x008D, EXC_TRAP_OTHER
0x16	AArch64 HVC that is not disabled		0x008A, EXC_HVC	0x008A, EXC_HVC
0x17	AArch64 SMC that is not disabled	to EL2	-	0x008D, EXC_TRAP_OTHER
		to EL3	0x0088, EXC_SMC	0x0088, EXC_SMC
0x18	AArch64 MSR, MRS and System instruction traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x19	SVE traps		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x1E	Granule Protection Check exception	Inst	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
		Data	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x1F	IMPLEMENTATION DEFINED exception taken to EL3		IMPLEMENTATION DEFINED[a]	IMPLEMENTATION DEFINED[a]
0x20	Instruction Abort from below		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x21	Instruction Abort from current Exception level		0x0083, EXC_PABORT	-
0x22	PC alignment		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x24	Data Abort from below		0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x25	Data Abort from current Exception level		0x0084, EXC_DABORT	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ESR.EC	Description		Event number and classification for exceptions	
			Taken locally	Not Taken locally
0x26	SP alignment fault exception		0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x28	AArch32 FP exception		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x2C	AArch64 FP exception		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x2D	GCS Exception	GCS Data Check exception	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
		EXLOCK exception	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
		Trap on GCSSTR or GCSSTTR	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x2F	SError exception		0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x30	Breakpoint from below		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x31	Breakpoint from current Exception level		0x0083, EXC_PABORT	-
0x32	Software step from below		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x33	Software step from current Exception level		0x0083, EXC_PABORT	-
0x34	Watchpoint from below		0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x35	Watchpoint from current Exception level		0x0084, EXC_DABORT	-
0x38	AArch32 BKPT instruction		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x3A	AArch32 Vector Catch debug event		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x3C	AArch64 BRK instruction		0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
-	IRQ interrupt		0x0086, EXC_IRQ	0x008E, EXC_TRAP_FIQ
-	FIQ interrupt		0x0087, EXC_FIQ	0x008F, EXC_TRAP_FIQ
All other values	All other exceptions		0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER

a. The exception reported with EC 0x1F is IMPLEMENTATION DEFINED, and therefore it is IMPLEMENTATION DEFINED which event counts the exception, except that the event that counts the exception must correctly indicate whether the exception was Taken locally.

Note

The Glossary defines the term Taken locally, that is used in event definitions in Common microarchitectural events. See also Exception levels for more information.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.2 The PMU event number space and common events

In Armv8.0, the event number space is 10 bits. Armv8.1 extends the event number space, and therefore the PMEVTYPER<n>_EL0.evtCount field, to 16 bits, and is allocated as Table D14-2 shows. For more information about the entries in the Allocation column see the text that follows this table:

Table D14-2 Allocation of the PMU event number space

Event numbers	Allocation
0x0000 - 0x003F	Common architectural and microarchitectural events.
0x0040 - 0x00BF	When FEAT_PMUv3p8 is implemented, common architectural and microarchitectural events. Previously Arm-recommended common architectural and microarchitectural events.
0x00C0 - 0x03FF	IMPLEMENTATION DEFINED events.
When FEAT_PMUv3p1 is implemented:	
0x0400 - 0x3FFF	IMPLEMENTATION DEFINED events.
0x4000 - 0x403F	Common architectural and microarchitectural events.
0x4040 - 0x40BF	When FEAT_PMUv3p8 is implemented, common architectural and microarchitectural events. Previously Arm-recommended common architectural and microarchitectural events.
0x40C0 - 0x7FFF	IMPLEMENTATION DEFINED events.
0x8000 - 0x8FFF	Common architectural and microarchitectural events.
0x9000 - 0xC0BF	Reserved.
0xC0C0 - 0xFFFF	IMPLEMENTATION DEFINED events.

See PMEVTYPER<n>.evtCount for details of the PE behavior when an event number for a reserved or unimplemented PMU event is written to evtCount.

Table D14-3 lists the number, mnemonic, and description of PMU events.

Table D14-3 Event index

Number	Mnemonic	Description
0x0000	SW_INCR	Instruction architecturally executed, Condition code check pass, software increment.
0x0001	L1I_CACHE_REFILL	Level 1 instruction cache refill.
0x0002	L1I_TLB_REFILL	Level 1 instruction TLB refill.
0x0003	L1D_CACHE_REFILL	Level 1 data cache refill.
0x0004	L1D_CACHE	Level 1 data cache access.
0x0005	L1D_TLB_REFILL	Level 1 data TLB refill.
0x0006	LD_RETIRED	Instruction architecturally executed, Condition code check pass, load.
0x0007	ST_RETIRED	Instruction architecturally executed, Condition code check pass, store.
0x0008	INST_RETIRED	Instruction architecturally executed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0009	EXC_TAKEN	Exception taken.
0x000A	EXC_RETURN	Instruction architecturally executed, Condition code check pass, exception return.
0x000B	CID_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.
0x000C	PC_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, Software change of the PC.
0x000D	BR_IMMED_RETIRED	Branch instruction architecturally executed, immediate.
0x000E	BR_RETURN_RETIRED	Branch instruction architecturally executed, procedure return, taken.
0x000F	UNALIGNED_LDST_RETIRED	Instruction architecturally executed, Condition code check pass, unaligned load or store.
0x0010	BR_MIS_PRED	Branch instruction speculatively executed, mispredicted or not predicted.
0x0011	CPU_CYCLES	Cycle.
0x0012	BR_PRED	Predictable branch instruction speculatively executed.
0x0013	MEM_ACCESS	Data memory access.
0x0014	L1I_CACHE	Level 1 instruction cache access.
0x0015	L1D_CACHE_WB	Level 1 data cache write-back.
0x0016	L2D_CACHE	Level 2 data cache access.
0x0017	L2D_CACHE_REFILL	Level 2 data cache refill.
0x0018	L2D_CACHE_WB	Level 2 data cache write-back.
0x0019	BUS_ACCESS	Bus access.
0x001A	MEMORY_ERROR	Local memory error.
0x001B	INST_SPEC	Operation speculatively executed.
0x001C	TTBR_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, write to TTBR.
0x001D	BUS_CYCLES	Bus cycle.
0x001E	CHAIN	Chain a pair of event counters.
0x001F	L1D_CACHE_ALLOCATE	Level 1 data cache allocation without refill.
0x0020	L2D_CACHE_ALLOCATE	Level 2 data cache allocation without refill.
0x0021	BR_RETIRED	Instruction architecturally executed, branch.
0x0022	BR_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted.
0x0023	STALL_FRONTEND	No operation sent for execution due to the frontend.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0024	STALL_BACKEND	No operation sent for execution due to the backend.
0x0025	L1D_TLB	Level 1 data TLB access.
0x0026	L1I_TLB	Level 1 instruction TLB access.
0x0027	L2I_CACHE	Level 2 instruction cache access.
0x0028	L2I_CACHE_REFILL	Level 2 instruction cache refill.
0x0029	L3D_CACHE_ALLOCATE	Level 3 data cache allocation without refill.
0x002A	L3D_CACHE_REFILL	Level 3 data cache refill.
0x002B	L3D_CACHE	Level 3 data cache access.
0x002C	L3D_CACHE_WB	Level 3 data cache write-back.
0x002D	L2D_TLB_REFILL	Level 2 data TLB refill.
0x002E	L2I_TLB_REFILL	Level 2 instruction TLB refill.
0x002F	L2D_TLB	Level 2 data TLB access.
0x0030	L2I_TLB	Level 2 instruction TLB access.
0x0031	REMOTE_ACCESS	Access to a remote device.
0x0032	LL_CACHE	Last level cache access.
0x0033	LL_CACHE_MISS	Last level cache miss.
0x0034	DTLB_WALK	Data TLB access with at least one translation table walk.
0x0035	ITLB_WALK	Instruction TLB access with at least one translation table walk.
0x0036	LL_CACHE_RD	Last level cache access, read.
0x0037	LL_CACHE_MISS_RD	Last level cache miss, read.
0x0038	REMOTE_ACCESS_RD	Access to a remote device, read.
0x0039	L1D_CACHE_LMISS_RD	Level 1 data cache long-latency read miss.
0x003A	OP_RETIRED	Micro-operation architecturally executed.
0x003B	OP_SPEC	Micro-operation speculatively executed.
0x003C	STALL	No operation sent for execution.
0x003D	STALL_SLOT_BACKEND	No operation sent for execution on a Slot due to the backend.
0x003E	STALL_SLOT_FRONTEND	No operation sent for execution on a Slot due to the frontend.
0x003F	STALL_SLOT	No operation sent for execution on a Slot.
0x0040	L1D_CACHE_RD	Level 1 data cache access, read.
0x0041	L1D_CACHE_WR	Level 1 data cache access, write.
0x0042	L1D_CACHE_REFILL_RD	Level 1 data cache refill, read.
0x0043	L1D_CACHE_REFILL_WR	Level 1 data cache refill, write.
0x0044	L1D_CACHE_REFILL_INNER	Level 1 data cache refill, inner.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0045	L1D_CACHE_REFILL_OUTER	Level 1 data cache refill, outer.
0x0046	L1D_CACHE_WB_VICTIM	Level 1 data cache write-back, victim.
0x0047	L1D_CACHE_WB_CLEAN	Level 1 data cache write-back, cleaning and coherency.
0x0048	L1D_CACHE_INVAL	Level 1 data cache invalidate.
0x004C	L1D_TLB_REFILL_RD	Level 1 data TLB refill, read.
0x004D	L1D_TLB_REFILL_WR	Level 1 data TLB refill, write.
0x004E	L1D_TLB_RD	Level 1 data TLB access, read.
0x004F	L1D_TLB_WR	Level 1 data TLB access, write.
0x0050	L2D_CACHE_RD	Level 2 data cache access, read.
0x0051	L2D_CACHE_WR	Level 2 data cache access, write.
0x0052	L2D_CACHE_REFILL_RD	Level 2 data cache refill, read.
0x0053	L2D_CACHE_REFILL_WR	Level 2 data cache refill, write.
0x0056	L2D_CACHE_WB_VICTIM	Level 2 data cache write-back, victim.
0x0057	L2D_CACHE_WB_CLEAN	Level 2 data cache write-back, cleaning and coherency.
0x0058	L2D_CACHE_INVAL	Level 2 data cache invalidate.
0x005C	L2D_TLB_REFILL_RD	Level 2 data TLB refill, read.
0x005D	L2D_TLB_REFILL_WR	Level 2 data TLB refill, write.
0x005E	L2D_TLB_RD	Level 2 data TLB access, read.
0x005F	L2D_TLB_WR	Level 2 data TLB access, write.
0x0060	BUS_ACCESS_RD	Bus access, read.
0x0061	BUS_ACCESS_WR	Bus access, write.
0x0062	BUS_ACCESS_SHARED	Bus access, Normal, Cacheable, Shareable.
0x0063	BUS_ACCESS_NOT_SHARED	Bus access, not Normal, Cacheable, Shareable.
0x0064	BUS_ACCESS_NORMAL	Bus access, normal.
0x0065	BUS_ACCESS_PERIPH	Bus access, peripheral.
0x0066	MEM_ACCESS_RD	Data memory access, read.
0x0067	MEM_ACCESS_WR	Data memory access, write.
0x0068	UNALIGNED_LD_SPEC	Unaligned access, read.
0x0069	UNALIGNED_ST_SPEC	Unaligned access, write.
0x006A	UNALIGNED_LDST_SPEC	Unaligned access.
0x006C	LDREX_SPEC	Exclusive operation speculatively executed, Load-Exclusive.
0x006D	STREX_PASS_SPEC	Exclusive operation speculatively executed, Store-Exclusive pass.
0x006E	STREX_FAIL_SPEC	Exclusive operation speculatively executed, Store-Exclusive fail.
0x006F	STREX_SPEC	Exclusive operation speculatively executed, Store-Exclusive.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0070	LD_SPEC	Operation speculatively executed, load.
0x0071	ST_SPEC	Operation speculatively executed, store.
0x0072	LDST_SPEC	Operation speculatively executed, load or store.
0x0073	DP_SPEC	Operation speculatively executed, integer data processing.
0x0074	ASE_SPEC	Operation speculatively executed, Advanced SIMD data processing.
0x0075	VFP_SPEC	Operation speculatively executed, scalar floating-point data processing.
0x0076	PC_WRITE_SPEC	Operation speculatively executed, Software change of the PC.
0x0077	CRYPTO_SPEC	Operation speculatively executed, cryptographic data processing.
0x0078	BR_IMMED_SPEC	Branch speculatively executed, immediate branch.
0x0079	BR_RETURN_SPEC	Branch speculatively executed, procedure return.
0x007A	BR_INDIRECT_SPEC	Branch speculatively executed, indirect branch.
0x007C	ISB_SPEC	Barrier speculatively executed, ISB.
0x007D	DSB_SPEC	Barrier speculatively executed, DSB.
0x007E	DMB_SPEC	Barrier speculatively executed, DMB.
0x007F	CSDB_SPEC	Barrier speculatively executed, CSDB.
0x0081	EXC_UNDEF	Exception taken, other synchronous.
0x0082	EXC_SVC	Exception taken, Supervisor Call.
0x0083	EXC_PABORT	Exception taken, Instruction Abort.
0x0084	EXC_DABORT	Exception taken, Data Abort or SError.
0x0086	EXC_IRQ	Exception taken, IRQ.
0x0087	EXC_FIQ	Exception taken, FIQ.
0x0088	EXC_SMC	Exception taken, Secure Monitor Call.
0x008A	EXC_HVC	Exception taken, Hypervisor Call.
0x008B	EXC_TRAP_PABORT	Exception taken, Instruction Abort not Taken locally.
0x008C	EXC_TRAP_DABORT	Exception taken, Data Abort or SError not Taken locally.
0x008D	EXC_TRAP_OTHER	Exception taken, other traps not Taken locally.
0x008E	EXC_TRAP_IRQ	Exception taken, IRQ not Taken locally.
0x008F	EXC_TRAP_FIQ	Exception taken, FIQ not Taken locally.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0090	RC_LD_SPEC	Release consistency operation speculatively executed, Load-Acquire.
0x0091	RC_ST_SPEC	Release consistency operation speculatively executed, Store-Release.
0x0092	L3D_CACHE_HIT_WR_FEXT_PROP	Level 3 data cache demand access first hit, write, fetched by external propagating transaction.
0x00A0	L3D_CACHE_RD	Level 3 data cache access, read.
0x00A1	L3D_CACHE_WR	Level 3 data cache access, write.
0x00A2	L3D_CACHE_REFILL_RD	Level 3 data cache refill, read.
0x00A3	L3D_CACHE_REFILL_WR	Level 3 data cache refill, write.
0x00A6	L3D_CACHE_WB_VICTIM	Level 3 data cache write-back, victim.
0x00A7	L3D_CACHE_WB_CLEAN	Level 3 data cache write-back, cleaning and coherency.
0x00A8	L3D_CACHE_INVAL	Level 3 data cache invalidate.
0x00A9	L2D_CACHE_HIT_RD_FEXT_PROP	Level 2 data cache demand access first hit, read, fetched by external propagating transaction.
0x00AA	L2D_CACHE_HIT_WR_FEXT_PROP	Level 2 data cache demand access first hit, write, fetched by external propagating transaction.
0x00AB	L2D_CACHE_HIT_RW_FEXT_PROP	Level 2 data cache demand access first hit, fetched by external propagating transaction.
0x00AC	L3D_CACHE_EXT_PROP	Level 3 data cache allocation due to external propagating transaction.
0x00AD	L3D_CACHE_EXT_PROP_PRFM_IR	Level 3 data cache allocation due to external propagating transaction after prefetch with intent to read.
0x00AE	L3D_CACHE_HIT_RW_FEXT_PROP	Level 3 data cache demand access first hit, fetched by external propagating transaction.
0x00AF	L3D_CACHE_HIT_RD_FEXT_PROP	Level 3 data cache demand access first hit, read, fetched by external propagating transaction.
0x00B0	STSHH_SPEC	Store shared hint instruction speculatively executed.
0x00B1	STSHH_KEEP_SPEC	Store shared hint instruction speculatively executed with KEEP policy.
0x00B2	STSHH_STRM_SPEC	Store shared hint instruction speculatively executed with STRM policy.
0x00B3	PRFM_IR_SPEC	Prefetch with intent to read instruction speculatively executed.
0x00B4	MEM_ACCESS_WR_STSHH	Memory write with store shared hint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x00B5	MEM_ACCESS_WR_STSHH_KEEP	Memory write with store shared hint with keep policy.
0x00B6	MEM_ACCESS_WR_STSHH_STRM	Memory write with store shared hint with stream policy.
0x00B7	MEM_ACCESS_WR_STSHH_KEEP_NEAR	Memory write with store shared hint with keep policy performed near.
0x00B8	MEM_ACCESS_WR_STSHH_KEEP_FAR	Memory write with store shared hint with keep policy performed far.
0x00B9	L1D_CACHE_EXT_PROP	Level 1 data cache allocation due to external propagating transaction.
0x00BA	L1D_CACHE_EXT_PROP_PRFM_IR	Level 1 data cache allocation due to external propagating transaction after prefetch with intent to read.
0x00BB	L1D_CACHE_HIT_RD_FEXT_PROP	Level 1 data cache demand access first hit, read, fetched by external propagating transaction.
0x00BC	L1D_CACHE_HIT_WR_FEXT_PROP	Level 1 data cache demand access first hit, write, fetched by external propagating transaction.
0x00BD	L1D_CACHE_HIT_RW_FEXT_PROP	Level 1 data cache demand access first hit, fetched by external propagating transaction.
0x00BE	L2D_CACHE_EXT_PROP	Level 2 data cache allocation due to external propagating transaction.
0x00BF	L2D_CACHE_EXT_PROP_PRFM_IR	Level 2 data cache allocation due to external propagating transaction after prefetch with intent to read.
0x4000	SAMPLE_POP	Statistical Profiling sample population.
0x4001	SAMPLE_FEED	Statistical Profiling sample taken.
0x4002	SAMPLE_FILTRATE	Statistical Profiling sample taken and not removed by filtering.
0x4003	SAMPLE_COLLISION	Statistical Profiling sample collided with previous sample.
0x4004	CNT_CYCLES	Constant frequency cycles.
0x4005	STALL_BACKEND_MEM	Memory stall cycles.
0x4006	L1I_CACHE_LMISS	Level 1 instruction cache long-latency miss.
0x4009	L2D_CACHE_LMISS_RD	Level 2 data cache long-latency read miss.
0x400A	L2I_CACHE_LMISS	Level 2 instruction cache long-latency miss.
0x400B	L3D_CACHE_LMISS_RD	Level 3 data cache long-latency read miss.
0x400C	TRB_WRAP	Trace buffer current write pointer wrapped.
0x400D	PMU_OVFS	PMU overflow, counters accessible to EL1 and EL0.
0x400E	TRB_TRIG	Trace buffer Trigger Event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x400F	PMU_HOVFS	PMU overflow, counters reserved for use by EL2.
0x4010	TRCEXTOUT0	Trace unit external output 0.
0x4011	TRCEXTOUT1	Trace unit external output 1.
0x4012	TRCEXTOUT2	Trace unit external output 2.
0x4013	TRCEXTOUT3	Trace unit external output 3.
0x4018	CTI_TRIGOUT4	Cross-trigger Interface output trigger 4.
0x4019	CTI_TRIGOUT5	Cross-trigger Interface output trigger 5.
0x401A	CTI_TRIGOUT6	Cross-trigger Interface output trigger 6.
0x401B	CTI_TRIGOUT7	Cross-trigger Interface output trigger 7.
0x4020	LDST_ALIGN_LAT	Access with additional latency from alignment.
0x4021	LD_ALIGN_LAT	Load with additional latency from alignment.
0x4022	ST_ALIGN_LAT	Store with additional latency from alignment.
0x4024	MEM_ACCESS_CHECKED	Checked data memory access.
0x4025	MEM_ACCESS_CHECKED_RD	Checked data memory access, read.
0x4026	MEM_ACCESS_CHECKED_WR	Checked data memory access, write.
0x8000	SIMD_INST_RETIRED	Instruction architecturally executed, SIMD.
0x8001	ASE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD.
0x8002	SVE_INST_RETIRED	Instruction architecturally executed, SVE.
0x8003	ASE_SVE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD or SVE.
0x8004	SIMD_INST_SPEC	Operation speculatively executed, SIMD.
0x8005	ASE_INST_SPEC	Operation speculatively executed, Advanced SIMD.
0x8006	SVE_INST_SPEC	Operation speculatively executed, SVE.
0x8007	ASE_SVE_INST_SPEC	Operation speculatively executed, Advanced SIMD or SVE.
0x8008	UOP_SPEC	Microarchitectural operation speculatively executed.
0x8009	ASE_UOP_SPEC	Microarchitectural operation speculatively executed, Advanced SIMD.
0x800A	SVE_UOP_SPEC	Microarchitectural operation speculatively executed, SVE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x800B	ASE_SVE_UOP_SPEC	Microarchitectural operation speculatively executed, Advanced SIMD or SVE.
0x800C	SIMD_UOP_SPEC	Microarchitectural operation speculatively executed, SIMD.
0x800E	SVE_MATH_SPEC	Operation speculatively executed, SVE math accelerator.
0x8010	FP_SPEC	Floating-point operation speculatively executed, including SIMD.
0x8011	ASE_FP_SPEC	Floating-point operation speculatively executed, Advanced SIMD.
0x8012	SVE_FP_SPEC	Floating-point operation speculatively executed, SVE.
0x8013	ASE_SVE_FP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE.
0x8014	FP_HP_SPEC	Floating-point operation speculatively executed, half-precision.
0x8015	ASE_FP_HP_SPEC	Floating-point operation speculatively executed, Advanced SIMD half-precision, data-processing.
0x8016	SVE_FP_HP_SPEC	Floating-point operation speculatively executed, SVE half-precision, data-processing.
0x8017	ASE_SVE_FP_HP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE half-precision.
0x8018	FP_SP_SPEC	Floating-point operation speculatively executed, single-precision.
0x8019	ASE_FP_SP_SPEC	Floating-point operation speculatively executed, Advanced SIMD single-precision, data-processing.
0x801A	SVE_FP_SP_SPEC	Floating-point operation speculatively executed, SVE single-precision, data-processing.
0x801B	ASE_SVE_FP_SP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE single-precision, data-processing.
0x801C	FP_DP_SPEC	Floating-point operation speculatively executed, double-precision, data-processing.
0x801D	ASE_FP_DP_SPEC	Floating-point operation speculatively executed, Advanced SIMD double-precision, data-processing.
0x801E	SVE_FP_DP_SPEC	Floating-point operation speculatively executed, SVE double-precision, data-processing.
0x801F	ASE_SVE_FP_DP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE double-precision, data-processing.
0x8020	FP_DIV_SPEC	Floating-point operation speculatively executed, divide.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8021	ASE_FP_DIV_SPEC	Floating-point operation speculatively executed, Advanced SIMD divide.
0x8022	SVE_FP_DIV_SPEC	Floating-point operation speculatively executed, SVE divide.
0x8023	ASE_SVE_FP_DIV_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE divide.
0x8024	FP_SQRT_SPEC	Floating-point operation speculatively executed, square root.
0x8025	ASE_FP_SQRT_SPEC	Floating-point operation speculatively executed, Advanced SIMD square root.
0x8026	SVE_FP_SQRT_SPEC	Floating-point operation speculatively executed, SVE square root.
0x8027	ASE_SVE_FP_SQRT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE square-root.
0x8028	FP_FMA_SPEC	Floating-point operation speculatively executed, FMA.
0x8029	ASE_FP_FMA_SPEC	Floating-point operation speculatively executed, Advanced SIMD FMA.
0x802A	SVE_FP_FMA_SPEC	Floating-point operation speculatively executed, SVE FMA.
0x802B	ASE_SVE_FP_FMA_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE FMA.
0x802C	FP_MUL_SPEC	Floating-point operation speculatively executed, multiply.
0x802D	ASE_FP_MUL_SPEC	Floating-point operation speculatively executed, Advanced SIMD multiply.
0x802E	SVE_FP_MUL_SPEC	Floating-point operation speculatively executed, SVE multiply.
0x802F	ASE_SVE_FP_MUL_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE multiply.
0x8030	FP_ADDSUB_SPEC	Floating-point operation speculatively executed, add or subtract.
0x8031	ASE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, Advanced SIMD add or subtract.
0x8032	SVE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, SVE add or subtract.
0x8033	ASE_SVE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract .
0x8034	FP_RECPE_SPEC	Floating-point operation speculatively executed, reciprocal estimate.
0x8035	ASE_FP_RECPE_SPEC	Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8036	SVE_FP_RECPE_SPEC	Floating-point operation speculatively executed, SVE reciprocal estimate.
0x8037	ASE_SVE_FP_RECPE_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate.
0x8038	FP_CVT_SPEC	Floating-point operation speculatively executed, convert.
0x8039	ASE_FP_CVT_SPEC	Floating-point operation speculatively executed, Advanced SIMD convert.
0x803A	SVE_FP_CVT_SPEC	Floating-point operation speculatively executed, SVE convert.
0x803B	ASE_SVE_FP_CVT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE convert.
0x803C	SVE_FP_AREDUCE_SPEC	Floating-point operation speculatively executed, SVE accumulating reduction.
0x803D	ASE_FP_PREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD pairwise add step or pairwise reduce step.
0x803E	SVE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, SVE pairwise or reduction.
0x803F	ASE_SVE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction.
0x8040	INT_SPEC	Integer operation speculatively executed.
0x8041	ASE_INT_SPEC	Integer operation speculatively executed, Advanced SIMD.
0x8042	SVE_INT_SPEC	Integer operation speculatively executed, SVE.
0x8043	ASE_SVE_INT_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE.
0x8044	INT_DIV_SPEC	Integer operation speculatively executed, divide.
0x8045	INT_DIV64_SPEC	Integer operation speculatively executed, 64-bit divide.
0x8046	SVE_INT_DIV_SPEC	Integer operation speculatively executed, SVE divide.
0x8047	SVE_INT_DIV64_SPEC	Integer operation speculatively executed, SVE 64-bit divide.
0x8048	INT_MUL_SPEC	Integer operation speculatively executed, multiply.
0x8049	ASE_INT_MUL_SPEC	Integer operation speculatively executed, Advanced SIMD multiply.
0x804A	SVE_INT_MUL_SPEC	Integer operation speculatively executed, SVE multiply.
0x804B	ASE_SVE_INT_MUL_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE multiply.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x804C	INT_MUL64_SPEC	Integer operation speculatively executed, 64×64 multiply.
0x804D	SVE_INT_MUL64_SPEC	Integer operation speculatively executed, SVE 64×64 multiply.
0x804E	INT_MULH64_SPEC	Integer operation speculatively executed, 64×64 multiply returning high part.
0x804F	SVE_INT_MULH64_SPEC	Integer operation speculatively executed, SVE 64×64 multiply high part.
0x8051	ASE_FP_BF16_SPEC	Floating-point operation speculatively executed, Advanced SIMD BFloat16, data-processing.
0x8052	SVE_FP_BF16_SPEC	Floating-point operation speculatively executed, SVE BFloat16, data-processing.
0x8053	ASE_SVE_FP_BF16_SPEC	Floating-point operation speculatively executed, SVE or Advanced SIMD BFloat16, data-processing.
0x8054	FP_BF16_SPEC	Floating-point operation speculatively executed, BFloat16, data-processing.
0x8056	SVE_SPEC	Operation speculatively executed, SVE data processing.
0x8057	ASE_SVE_SPEC	Operation speculatively executed, Advanced SIMD data processing or SVE data processing.
0x8058	NONFP_SPEC	Non-floating-point operation speculatively executed.
0x8059	ASE_NONFP_SPEC	Non-floating-point operation speculatively executed, Advanced SIMD.
0x805A	SVE_NONFP_SPEC	Non-floating-point operation speculatively executed, SVE.
0x805B	ASE_SVE_NONFP_SPEC	Non-floating-point operation speculatively executed, Advanced SIMD or SVE.
0x805D	ASE_INT_VREDUCE_SPEC	Integer operation speculatively executed, Advanced SIMD reduction.
0x805E	SVE_INT_VREDUCE_SPEC	Integer operation speculatively executed, SVE reduction.
0x805F	ASE_SVE_INT_VREDUCE_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE reduction.
0x8060	SVE_PERM_SPEC	Operation speculatively executed, SVE permute.
0x8061	SVE_PERM_IGRANULE_SPEC	Operation speculatively executed, SVE intra-granule permute.
0x8062	SVE_PERM_XGRANULE_SPEC	Operation speculatively executed, SVE cross-granule permute.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8063	SVE_PERM_VARIABLE_SPEC	Operation speculatively executed, SVE programmable permute.
0x8064	SVE_XPIPE_SPEC	Operation speculatively executed, SVE cross-pipe.
0x8065	SVE_XPIPE_Z2R_SPEC	Operation speculatively executed, SVE vector to scalar cross-pipe.
0x8066	SVE_XPIPE_R2Z_SPEC	Operation speculatively executed, SVE scalar to vector cross-pipe.
0x8067	SVE_PGEN_NVEC_SPEC	Operation speculatively executed, SVE predicate-only.
0x8068	SVE_PGEN_SPEC	Operation speculatively executed, SVE predicate generating.
0x8069	SVE_PGEN_FLG_SPEC	Operation speculatively executed, SVE predicate flag setting.
0x806A	SVE_PGEN_CMP_SPEC	Operation speculatively executed, SVE vector compare.
0x806B	SVE_PGEN_FCM_SPEC	Floating-point operation speculatively executed, SVE vector compare.
0x806C	SVE_PGEN_LOGIC_SPEC	Operation speculatively executed, SVE predicate logical.
0x806D	SVE_PPERM_SPEC	Operation speculatively executed, SVE predicate permute.
0x806E	SVE_PSCAN_SPEC	Operation speculatively executed, SVE predicate scan.
0x806F	SVE_PCNT_SPEC	Operation speculatively executed, SVE predicate count.
0x8070	SVE_PLOOP_WHILE_SPEC	Operation speculatively executed, SVE predicate loop while.
0x8071	SVE_PLOOP_TEST_SPEC	Operation speculatively executed, SVE predicate loop test.
0x8072	SVE_PLOOP_ELTS_SPEC	Operation speculatively executed, SVE predicate loop elements.
0x8073	SVE_PLOOP_TERM_SPEC	Operation speculatively executed, SVE predicate loop termination.
0x8074	SVE_PRED_SPEC	Operation speculatively executed, SIMD predicated.
0x8075	SVE_PRED_EMPTY_SPEC	Operation speculatively executed, SIMD predicated with no active elements.
0x8076	SVE_PRED_FULL_SPEC	Operation speculatively executed, SIMD predicated with all active elements.
0x8077	SVE_PRED_PARTIAL_SPEC	Operation speculatively executed, SIMD predicated with partially active elements.
0x8078	SVE_UNPRED_SPEC	Operation speculatively executed, SVE unpredicated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8079	SVE_PRED_NOT_FULL_SPEC	Operation speculatively executed, SVE predicated with at least one inactive element.
0x807C	SVE_MOVPRFX_SPEC	Operation speculatively executed, SVE MOVPRFX.
0x807D	SVE_MOVPRFX_Z_SPEC	Operation speculatively executed, SVE MOVPRFX zeroing predication.
0x807E	SVE_MOVPRFX_M_SPEC	Operation speculatively executed, SVE MOVPRFX merging predication.
0x807F	SVE_MOVPRFX_U_SPEC	Operation speculatively executed, SVE MOVPRFX unfused.
0x8080	SVE_LDST_SPEC	Operation speculatively executed, SVE load, store, or prefetch.
0x8081	SVE_LD_SPEC	Operation speculatively executed, SVE load.
0x8082	SVE_ST_SPEC	Operation speculatively executed, SVE store.
0x8083	SVE_PRF_SPEC	Operation speculatively executed, SVE prefetch.
0x8084	ASE_SVE_LDST_SPEC	Operation speculatively executed, Advanced SIMD or SVE load or store.
0x8085	ASE_SVE_LD_SPEC	Operation speculatively executed, Advanced SIMD or SVE load.
0x8086	ASE_SVE_ST_SPEC	Operation speculatively executed, Advanced SIMD or SVE store.
0x8087	PRF_SPEC	Operation speculatively executed, prefetch.
0x8088	BASE_LDST_REG_SPEC	Operation speculatively executed, general-purpose register load, store, or prefetch.
0x8089	BASE_LD_REG_SPEC	Operation speculatively executed, general-purpose register load.
0x808A	BASE_ST_REG_SPEC	Operation speculatively executed, general-purpose register store.
0x808B	BASE_PRF_SPEC	Operation speculatively executed, general-purpose register prefetch.
0x808C	FPASE_LDST_REG_SPEC	Operation speculatively executed, SIMD&FP register load or store.
0x808D	FPASE_LD_REG_SPEC	Operation speculatively executed, SIMD&FP register load.
0x808E	FPASE_ST_REG_SPEC	Operation speculatively executed, SIMD&FP register store.
0x8090	SVE_LDST_REG_SPEC	Operation speculatively executed, SVE unpredicated load or store register.
0x8091	SVE_LDR_REG_SPEC	Operation speculatively executed, SVE unpredicated load register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8092	SVE_STR_REG_SPEC	Operation speculatively executed, SVE unpredicated store register.
0x8094	SVE_LDST_PREG_SPEC	Operation speculatively executed, SVE load or store predicate register.
0x8095	SVE_LDR_PREG_SPEC	Operation speculatively executed, SVE load predicate register.
0x8096	SVE_STR_PREG_SPEC	Operation speculatively executed, SVE store predicate register.
0x8098	SVE_LDST_ZREG_SPEC	Operation speculatively executed, SVE load or store vector register.
0x8099	SVE_LDR_ZREG_SPEC	Operation speculatively executed, SVE load vector register.
0x809A	SVE_STR_ZREG_SPEC	Operation speculatively executed, SVE store vector register.
0x809C	SVE_LDST_CONTIG_SPEC	Operation speculatively executed, SVE contiguous load, store, or prefetch element.
0x809D	SVE_LD_CONTIG_SPEC	Operation speculatively executed, SVE single vector contiguous load element.
0x809E	SVE_ST_CONTIG_SPEC	Operation speculatively executed, SVE contiguous store element.
0x809F	SVE_PRF_CONTIG_SPEC	Operation speculatively executed, SVE contiguous prefetch element.
0x80A0	SVE_LDSTNT_CONTIG_SPEC	Operation speculatively executed, SVE non-temporal contiguous load or store element.
0x80A1	SVE_LDNT_CONTIG_SPEC	Operation speculatively executed, SVE non-temporal contiguous load element.
0x80A2	SVE_STNT_CONTIG_SPEC	Operation speculatively executed, SVE non-temporal contiguous store element.
0x80A4	ASE_SVE_LDST_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector.
0x80A5	ASE_SVE_LD_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector.
0x80A6	ASE_SVE_ST_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector.
0x80A8	SVE_LDST_MULTI_SPEC	Operation speculatively executed, SVE contiguous load or store multiple vector.
0x80A9	SVE_LD_MULTI_SPEC	Operation speculatively executed, SVE contiguous load multiple vector.
0x80AA	SVE_ST_MULTI_SPEC	Operation speculatively executed, SVE contiguous store multiple vector.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x80AC	SVE_LDST_NONCONTIG_SPEC	Operation speculatively executed, SVE non-contiguous load, store, or prefetch.
0x80AD	SVE_LD_GATHER_SPEC	Operation speculatively executed, SVE gather-load.
0x80AE	SVE_ST_SCATTER_SPEC	Operation speculatively executed, SVE scatter-store.
0x80AF	SVE_PRF_GATHER_SPEC	Operation speculatively executed, SVE gather-prefetch.
0x80B0	SVE_LDST64_NONCONTIG_SPEC	Operation speculatively executed, SVE 64-bit non-contiguous load, store, or prefetch.
0x80B1	SVE_LD64_GATHER_SPEC	Operation speculatively executed, SVE 64-bit gather-load.
0x80B2	SVE_ST64_SCATTER_SPEC	Operation speculatively executed, SVE 64-bit scatter-store.
0x80B3	SVE_PRF64_GATHER_SPEC	Operation speculatively executed, SVE 64-bit gather-prefetch.
0x80B4	ASE_SVE_UNALIGNED_LDST_SPEC	Advanced SIMD or SVE unaligned access.
0x80B5	ASE_SVE_UNALIGNED_LD_SPEC	Advanced SIMD or SVE unaligned read.
0x80B6	ASE_SVE_UNALIGNED_ST_SPEC	Advanced SIMD or SVE unaligned write.
0x80B8	ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC	Advanced SIMD or SVE unaligned contiguous access.
0x80B9	ASE_SVE_UNALIGNED_CONTIG_LD_SPEC	Advanced SIMD or SVE unaligned contiguous read.
0x80BA	ASE_SVE_UNALIGNED_CONTIG_ST_SPEC	Advanced SIMD or SVE unaligned contiguous write.
0x80BC	SVE_LDFF_SPEC	Operation speculatively executed, SVE first-fault load.
0x80BD	SVE_LDFF_FAULT_SPEC	Operation speculatively executed, SVE first-fault load which set FFR bit to 0.
0x80C0	FP_SCALE_OPS_SPEC	Scalable element arithmetic operations speculatively executed, floating-point.
0x80C1	FP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, floating-point.
0x80C2	FP_HP_SCALE_OPS_SPEC	Scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point.
0x80C3	FP_HP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x80C4	FP_SP_SCALE_OPS_SPEC	Scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point.
0x80C5	FP_SP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point.
0x80C6	FP_DP_SCALE_OPS_SPEC	Scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point.
0x80C7	FP_DP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point.
0x80C8	INT_SCALE_OPS_SPEC	Scalable element arithmetic operations speculatively executed, integer.
0x80C9	INT_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, integer.
0x80CA	LDST_SCALE_OPS_SPEC	Scalable load or store element operations speculatively executed.
0x80CB	LDST_FIXED_OPS_SPEC	Non-scalable load or store element operations speculatively executed.
0x80CC	LD_SCALE_OPS_SPEC	Scalable load element operations speculatively executed.
0x80CD	LD_FIXED_OPS_SPEC	Non-scalable load element operations speculatively executed.
0x80CE	ST_SCALE_OPS_SPEC	Scalable store element operations speculatively executed.
0x80CF	ST_FIXED_OPS_SPEC	Non-scalable store element operations speculatively executed.
0x80D0	FP_SCALE2_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, floating-point.
0x80D2	FP_HP_SCALE2_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, largest type is half-precision floating-point.
0x80D3	FP_BF16_SCALE2_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, largest type is BFloat16 floating-point.
0x80D4	FP_SP_SCALE2_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, largest type is single-precision floating-point.
0x80D6	FP_DP_SCALE2_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, largest type is double-precision floating-point.
0x80D8	INT_SCALE2_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x80DA	LDST_SCALE_BYTES_SPEC	Scalable load and store bytes speculatively executed.
0x80DB	LDST_FIXED_BYTES_SPEC	Non-scalable load and store bytes speculatively executed.
0x80DC	LD_SCALE_BYTES_SPEC	Scalable load bytes speculatively executed.
0x80DD	LD_FIXED_BYTES_SPEC	Non-scalable load bytes speculatively executed.
0x80DE	ST_SCALE_BYTES_SPEC	Scalable store bytes speculatively executed.
0x80DF	ST_FIXED_BYTES_SPEC	Non-scalable store bytes speculatively executed.
0x80E1	ASE_INT8_SPEC	Integer operation speculatively executed, Advanced SIMD 8-bit.
0x80E2	SVE_INT8_SPEC	Integer operation speculatively executed, SVE 8-bit.
0x80E3	ASE_SVE_INT8_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 8-bit.
0x80E5	ASE_INT16_SPEC	Integer operation speculatively executed, Advanced SIMD 16-bit.
0x80E6	SVE_INT16_SPEC	Integer operation speculatively executed, SVE 16-bit.
0x80E7	ASE_SVE_INT16_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 16-bit.
0x80E8	FP_BF16_SCALE_OPS_SPEC	Scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point.
0x80E9	ASE_INT32_SPEC	Integer operation speculatively executed, Advanced SIMD 32-bit.
0x80EA	SVE_INT32_SPEC	Integer operation speculatively executed, SVE 32-bit.
0x80EB	ASE_SVE_INT32_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 32-bit.
0x80ED	ASE_INT64_SPEC	Integer operation speculatively executed, Advanced SIMD 64-bit.
0x80EE	SVE_INT64_SPEC	Integer operation speculatively executed, SVE 64-bit.
0x80EF	ASE_SVE_INT64_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 64-bit.
0x80F1	ASE_FP_DOT_SPEC	Floating-point operation speculatively executed, Advanced SIMD dot-product.
0x80F2	SVE_FP_DOT_SPEC	Floating-point operation speculatively executed, SVE dot-product.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x80F3	ASE_SVE_FP_DOT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product.
0x80F5	ASE_FP_MMLA_SPEC	Floating-point operation speculatively executed, Advanced SIMD matrix multiply.
0x80F6	SVE_FP_MMLA_SPEC	Floating-point operation speculatively executed, SVE matrix multiply.
0x80F7	ASE_SVE_FP_MMLA_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply.
0x80F9	ASE_INT_DOT_SPEC	Operation speculatively executed, Advanced SIMD integer dot-product.
0x80FA	SVE_INT_DOT_SPEC	Integer operation speculatively executed, SVE dot-product.
0x80FB	ASE_SVE_INT_DOT_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE dot-product.
0x80FD	ASE_INT_MMLA_SPEC	Integer operation speculatively executed, Advanced SIMD matrix multiply.
0x80FE	SVE_INT_MMLA_SPEC	Integer operation speculatively executed, SVE matrix multiply.
0x80FF	ASE_SVE_INT_MMLA_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply.
0x8107	BR_SKIP_RETIRED	Branch instruction architecturally executed, not taken.
0x8108	BR_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, immediate, taken.
0x8109	BR_IMMED_SKIP_RETIRED	Branch instruction architecturally executed, immediate, not taken.
0x810A	BR_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect, taken.
0x810B	BR_IND_SKIP_RETIRED	Branch instruction architecturally executed, indirect, not taken.
0x810C	BR_INDNR_TAKEN_RETIRED	Branch instruction architecturally executed, indirect excluding procedure return, taken.
0x810D	BR_INDNR_SKIP_RETIRED	Branch instruction architecturally executed, indirect excluding procedure return, not taken.
0x810E	BR_RETURN_ANY_RETIRED	Branch instruction architecturally executed, procedure return.
0x810F	BR_RETURN_SKIP_RETIRED	Branch instruction architecturally executed, procedure return, not taken.
0x8110	BR_IMMED_PRED_RETIRED	Branch instruction architecturally executed, predicted immediate.
0x8111	BR_IMMED_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted immediate.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8112	BR_IND_PRED_RETIRED	Branch instruction architecturally executed, predicted indirect.
0x8113	BR_IND_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted indirect.
0x8114	BR_RETURN_PRED_RETIRED	Branch instruction architecturally executed, predicted procedure return.
0x8115	BR_RETURN_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted procedure return.
0x8116	BR_INDNR_PRED_RETIRED	Branch instruction architecturally executed, predicted indirect excluding procedure return.
0x8117	BR_INDNR_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted indirect excluding procedure return.
0x8118	BR_TAKEN_PRED_RETIRED	Branch instruction architecturally executed, predicted branch, taken.
0x8119	BR_TAKEN_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted branch, taken.
0x811A	BR_SKIP_PRED_RETIRED	Branch instruction architecturally executed, predicted branch, not taken.
0x811B	BR_SKIP_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted branch, not taken.
0x811C	BR_PRED_RETIRED	Branch instruction architecturally executed, predicted branch.
0x811D	BR_IND_RETIRED	Instruction architecturally executed, indirect branch.
0x811E	BR_INDNR_RETIRED	Branch instruction architecturally executed, indirect excluding procedure return.
0x811F	BRB_FILTRATE	Branch Record captured.
0x8120	INST_FETCH_PERCYC	Instruction fetches in progress.
0x8121	MEM_ACCESS_RD_PERCYC	Data memory reads in progress.
0x8122	SAMPLE_FEED_DS	Statistical Profiling sample taken, selected Data Source.
0x8123	SAMPLE_BUFFER_FULL	Profiling Buffer full.
0x8124	INST_FETCH	Instruction memory access.
0x8125	BUS_REQ_RD_PERCYC	Bus read transactions in progress.
0x8126	BUS_REQ_WR_PERCYC	Bus write transactions in progress.
0x8127	PMU_SNAPSHOT	Successful PMU capture event.
0x8128	DTLB_WALK_PERCYC	Data translation table walks in progress.
0x8129	ITLB_WALK_PERCYC	Instruction translation table walks in progress.
0x812A	SAMPLE_FEED_BR	Statistical Profiling sample taken, branch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x812B	SAMPLE_FEED_LD	Statistical Profiling sample taken, load.
0x812C	SAMPLE_FEED_ST	Statistical Profiling sample taken, store.
0x812D	SAMPLE_FEED_OP	Statistical Profiling sample taken, matching type.
0x812E	SAMPLE_FEED_EVENT	Statistical Profiling sample taken, matching events.
0x812F	SAMPLE_FEED_LAT	Statistical Profiling sample taken, exceeding minimum latency.
0x8130	L1D_TLB_RW	Level data TLB access, demand access.
0x8131	L1I_TLB_RD	Level instruction TLB access, demand access.
0x8132	L1D_TLB_PRFM	Level data TLB access, software prefetch.
0x8133	L1I_TLB_PRFM	Level instruction TLB access, software prefetch.
0x8134	DTLB_HWUPD	Data TLB hardware update of translation table.
0x8135	ITLB_HWUPD	Instruction TLB hardware update of translation table.
0x8136	DTLB_STEP	Data TLB translation table walk, step.
0x8137	ITLB_STEP	Instruction TLB translation table walk, step.
0x8138	DTLB_WALK_LARGE	Data TLB large page translation table walk.
0x8139	ITLB_WALK_LARGE	Instruction TLB large page translation table walk.
0x813A	DTLB_WALK_SMALL	Data TLB small page translation table walk.
0x813B	ITLB_WALK_SMALL	Instruction TLB small page translation table walk.
0x813C	DTLB_WALK_RW	Data TLB demand access, with at least one translation table walk.
0x813D	ITLB_WALK_RD	Instruction TLB demand access, with at least one translation table walk.
0x813E	DTLB_WALK_PRFM	Data TLB software prefetch, with at least one translation table walk.
0x813F	ITLB_WALK_PRFM	Instruction TLB software prefetch, with at least one translation table walk.
0x8140	L1D_CACHE_RW	Level 1 data cache demand access.
0x8141	L1I_CACHE_RD	Level 1 instruction cache demand fetch.
0x8142	L1D_CACHE_PRFM	Level 1 data cache software prefetch.
0x8143	L1I_CACHE_PRFM	Level 1 instruction cache software prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8144	L1D_CACHE_MISS	Level 1 data cache demand access miss.
0x8145	L1I_CACHE_HWPRF	Level 1 instruction cache hardware prefetch.
0x8146	L1D_CACHE_REFILL_PRFM	Level 1 data cache refill, software prefetch.
0x8147	L1I_CACHE_REFILL_PRFM	Level 1 instruction cache refill, software prefetch.
0x8148	L2D_CACHE_RW	Level 2 data cache demand access.
0x8149	L2I_CACHE_RD	Level 2 instruction cache demand fetch.
0x814A	L2D_CACHE_PRFM	Level 2 data cache software prefetch.
0x814B	L2I_CACHE_PRFM	Level 2 instruction cache software prefetch.
0x814C	L2D_CACHE_MISS	Level 2 data cache demand access miss.
0x814D	L2I_CACHE_HWPRF	Level 2 instruction cache hardware prefetch.
0x814E	L2D_CACHE_REFILL_PRFM	Level 2 data cache refill, software prefetch.
0x814F	L2I_CACHE_REFILL_PRFM	Level 2 instruction cache refill, software prefetch.
0x8150	L3D_CACHE_RW	Level 3 data cache demand access.
0x8151	L3D_CACHE_PRFM	Level 3 data cache software prefetch.
0x8152	L3D_CACHE_MISS	Level 3 data cache demand access miss.
0x8153	L3D_CACHE_REFILL_PRFM	Level 3 data cache refill, software prefetch.
0x8154	L1D_CACHE_HWPRF	Level 1 data cache hardware prefetch.
0x8155	L2D_CACHE_HWPRF	Level 2 data cache hardware prefetch.
0x8156	L3D_CACHE_HWPRF	Level 3 data cache hardware prefetch.
0x8157	LL_CACHE_HWPRF	Last level cache hardware prefetch.
0x8158	STALL_FRONTEND_MEMBOUND	Frontend stall cycles, memory bound.
0x8159	STALL_FRONTEND_L1I	Frontend stall cycles, level 1 instruction cache.
0x815A	STALL_FRONTEND_L2I	Frontend stall cycles, level 2 instruction cache.
0x815B	STALL_FRONTEND_MEM	Frontend stall cycles, last level PE cache or memory.
0x815C	STALL_FRONTEND_TLB	Frontend stall cycles, TLB.
0x815D	STALL_BACKEND_BUSY_SMCU	Backend stall cycles, SMCU busy.
0x815E	STALL_BACKEND_BUSY_SMCU_UNALL	Backend stall cycles, awaiting SMCU allocation.
0x8160	STALL_FRONTEND_CPUBOUND	Frontend stall cycles, processor bound.
0x8161	STALL_FRONTEND_FLOW	Frontend stall cycles, flow control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8162	STALL_FRONTEND_FLUSH	Frontend stall cycles, flush recovery.
0x8163	STALL_FRONTEND_RENAME	Frontend stall cycles, rename full.
0x8164	STALL_BACKEND_MEMBOUND	Backend stall cycles, memory bound.
0x8165	STALL_BACKEND_L1D	Backend stall cycles, level 1 data cache.
0x8166	STALL_BACKEND_L2D	Backend stall cycles, level 2 data cache.
0x8167	STALL_BACKEND_TLB	Backend stall cycles, TLB.
0x8168	STALL_BACKEND_ST	Backend stall cycles, store.
0x816A	STALL_BACKEND_CPUBOUND	Backend stall cycles, processor bound.
0x816B	STALL_BACKEND_BUSY	Backend stall cycles, backend busy.
0x816C	STALL_BACKEND_ILOCK	Backend stall cycles, input dependency.
0x816D	STALL_BACKEND_RENAME	Backend stall cycles, rename full.
0x816E	STALL_BACKEND_ATOMIC	Backend stall cycles, atomic operation.
0x816F	STALL_BACKEND_MEMCPYSET	Backend stall cycles, Memory Copy or Set operation.
0x8170	CAS_NEAR_FAIL	Atomic memory Operation speculatively executed, Compare and Swap fail.
0x8171	CAS_NEAR_PASS	Atomic memory Operation speculatively executed, Compare and Swap pass.
0x8172	CAS_NEAR_SPEC	Atomic memory Operation speculatively executed, Compare and Swap near.
0x8173	CAS_FAR_SPEC	Atomic memory Operation speculatively executed, Compare and Swap far.
0x8174	CAS_SPEC	Atomic memory Operation speculatively executed, Compare and Swap.
0x8175	LSE_LD_SPEC	Atomic memory Operation speculatively executed, load.
0x8176	LSE_ST_SPEC	Atomic memory Operation speculatively executed, store.
0x8177	LSE_LDST_SPEC	Atomic memory Operation speculatively executed, load or store.
0x8178	REMOTE_ACCESS_WR	Access to a remote device, write.
0x8179	BRNL_INDNR_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch without link excluding procedure return, taken.
0x817A	BL_TAKEN_RETIRED	Branch instruction architecturally executed, branch with link, taken.
0x817B	BRNL_TAKEN_RETIRED	Branch instruction architecturally executed, branch without link, taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x817C	BL_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch with link, taken.
0x817D	BRNL_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch without link, taken.
0x817E	BL_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, direct branch with link, taken.
0x817F	BRNL_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, direct branch without link, taken.
0x8180	BR_UNCOND_RETIRED	Branch instruction architecturally executed, unconditional branch.
0x8181	BR_COND_RETIRED	Branch instruction architecturally executed, conditional branch.
0x8182	BR_COND_TAKEN_RETIRED	Branch instruction architecturally executed, conditional branch, taken.
0x8183	BR_HINT_COND_RETIRED	Branch instruction architecturally executed, hinted conditional.
0x8184	BR_HINT_COND_PRED_RETIRED	Branch instruction architecturally executed, predicted hinted conditional.
0x8185	BR_HINT_COND_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted hinted conditional.
0x8186	UOP_RETIRED	Micro-operation architecturally executed.
0x8188	DTLB_WALK_BLOCK	Data TLB block translation table walk.
0x8189	ITLB_WALK_BLOCK	Instruction TLB block translation table walk.
0x818A	DTLB_WALK_PAGE	Data TLB page translation table walk.
0x818B	ITLB_WALK_PAGE	Instruction TLB page translation table walk.
0x818D	BUS_REQ_RD	Bus request, read.
0x818E	BUS_REQ_WR	Bus request, write.
0x818F	BUS_REQ	Bus request.
0x8190	ISNP_HIT_RD	Snoop hit, demand instruction fetch.
0x8191	ISNP_HIT_NEAR_RD	Snoop hit in near local cache, demand instruction fetch.
0x8192	ISNP_HIT_FAR_RD	Snoop hit in far local cache, demand instruction fetch.
0x8193	ISNP_HIT_REMOTE_RD	Snoop hit in remote cache, demand instruction fetch.
0x8194	DSNP_HIT_RD	Snoop hit, demand data read.
0x8195	DSNP_HIT_NEAR_RD	Snoop hit in near local cache, demand data read.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8196	DSNP_HIT_FAR_RD	Snoop hit in far local cache, demand data read.
0x8197	DSNP_HIT_REMOTE_RD	Snoop hit in remote cache, demand data read.
0x8198	DSNP_HIT_WR	Snoop hit, demand data write.
0x8199	DSNP_HIT_NEAR_WR	Snoop hit in near local cache, demand data write.
0x819A	DSNP_HIT_FAR_WR	Snoop hit in far local cache, demand data write.
0x819B	DSNP_HIT_REMOTE_WR	Snoop hit in remote cache, demand data write.
0x819C	DSNP_HIT_RW	Snoop hit, demand data access.
0x819D	DSNP_HIT_NEAR_RW	Snoop hit in near local cache, demand data access.
0x819E	DSNP_HIT_FAR_RW	Snoop hit in far local cache, demand data access.
0x819F	DSNP_HIT_REMOTE_RW	Snoop hit in remote cache, demand data access.
0x81A0	DSNP_HIT_PRFM	Snoop hit, software data prefetch.
0x81A1	DSNP_HIT_NEAR_PRFM	Snoop hit in near local cache, software data prefetch.
0x81A2	DSNP_HIT_FAR_PRFM	Snoop hit in far local cache, software data prefetch.
0x81A3	DSNP_HIT_REMOTE_PRFM	Snoop hit in remote cache, software data prefetch.
0x81A4	DSNP_HIT_HWPRF	Snoop hit, hardware data prefetch.
0x81A5	DSNP_HIT_NEAR_HWPRF	Snoop hit in near local cache, hardware data prefetch.
0x81A6	DSNP_HIT_FAR_HWPRF	Snoop hit in far local cache, hardware data prefetch.
0x81A7	DSNP_HIT_REMOTE_HWPRF	Snoop hit in remote cache, hardware data prefetch.
0x81A8	ISNP_HIT_PRFM	Snoop hit, software instruction prefetch.
0x81A9	ISNP_HIT_NEAR_PRFM	Snoop hit in near local cache, software instruction prefetch.
0x81AA	ISNP_HIT_FAR_PRFM	Snoop hit in far local cache, software instruction prefetch.
0x81AB	ISNP_HIT_REMOTE_PRFM	Snoop hit in remote cache, software instruction prefetch.
0x81AC	ISNP_HIT_HWPRF	Snoop hit, hardware instruction prefetch.
0x81AD	ISNP_HIT_NEAR_HWPRF	Snoop hit in near local cache, hardware instruction prefetch.
0x81AE	ISNP_HIT_FAR_HWPRF	Snoop hit in far local cache, hardware instruction prefetch.
0x81AF	ISNP_HIT_REMOTE_HWPRF	Snoop hit in remote cache, hardware instruction prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81B0	ISNP_HIT	Snoop hit, instruction.
0x81B1	ISNP_HIT_NEAR	Snoop hit in near local cache, instruction access.
0x81B2	ISNP_HIT_FAR	Snoop hit in far local cache, instruction access.
0x81B3	ISNP_HIT_REMOTE	Snoop hit in remote cache, instruction access.
0x81B4	DSNP_HIT	Snoop hit, data.
0x81B5	DSNP_HIT_NEAR	Snoop hit in near local cache, data access.
0x81B6	DSNP_HIT_FAR	Snoop hit in far local cache, data access.
0x81B7	DSNP_HIT_REMOTE	Snoop hit in remote cache, data access.
0x81B8	L1I_CACHE_REFILL_HWPRF	Level 1 instruction cache refill, hardware prefetch.
0x81B9	L2I_CACHE_REFILL_HWPRF	Level 2 instruction cache refill, hardware prefetch.
0x81BC	L1D_CACHE_REFILL_HWPRF	Level 1 data cache refill, hardware prefetch.
0x81BD	L2D_CACHE_REFILL_HWPRF	Level 2 data cache refill, hardware prefetch.
0x81BE	L3D_CACHE_REFILL_HWPRF	Level 3 data cache refill, hardware prefetch.
0x81BF	LL_CACHE_REFILL_HWPRF	Last level cache refill, hardware prefetch.
0x81C0	L1I_CACHE_HIT_RD	Level 1 instruction cache demand fetch hit.
0x81C1	L2I_CACHE_HIT_RD	Level 2 instruction cache demand fetch hit.
0x81C4	L1D_CACHE_HIT_RD	Level 1 data cache demand access hit, read.
0x81C5	L2D_CACHE_HIT_RD	Level 2 data cache demand access hit, read.
0x81C6	L3D_CACHE_HIT_RD	Level 3 data cache demand access hit, read.
0x81C7	LL_CACHE_HIT_RD	Last level cache demand access hit, read.
0x81C8	L1D_CACHE_HIT_WR	Level 1 data cache demand access hit, write.
0x81C9	L2D_CACHE_HIT_WR	Level 2 data cache demand access hit, write.
0x81CA	L3D_CACHE_HIT_WR	Level 3 data cache demand access hit, write.
0x81CB	LL_CACHE_HIT_WR	Last level cache demand access hit, write.
0x81CC	L1D_CACHE_HIT_RW	Level 1 data cache demand access hit.
0x81CD	L2D_CACHE_HIT_RW	Level 2 data cache demand access hit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81CE	L3D_CACHE_HIT_RW	Level 3 data cache demand access hit.
0x81CF	LL_CACHE_HIT_RW	Last level cache demand access hit.
0x81D0	L1I_CACHE_HIT_RD_FPRFM	Level 1 instruction cache demand fetch first hit, fetched by software prefetch.
0x81D1	L2I_CACHE_HIT_RD_FPRFM	Level 2 instruction cache demand fetch first hit, fetched by software prefetch.
0x81D4	L1D_CACHE_HIT_RD_FPRFM	Level 1 data cache demand access first hit, read, fetched by software prefetch.
0x81D5	L2D_CACHE_HIT_RD_FPRFM	Level 2 data cache demand access first hit, read, fetched by software prefetch.
0x81D6	L3D_CACHE_HIT_RD_FPRFM	Level 3 data cache demand access first hit, read, fetched by software prefetch.
0x81D7	LL_CACHE_HIT_RD_FPRFM	Last level cache demand access first hit, read, fetched by software prefetch.
0x81D8	L1D_CACHE_HIT_WR_FPRFM	Level 1 data cache demand access first hit, write, fetched by software prefetch.
0x81D9	L2D_CACHE_HIT_WR_FPRFM	Level 2 data cache demand access first hit, write, fetched by software prefetch.
0x81DA	L3D_CACHE_HIT_WR_FPRFM	Level 3 data cache demand access first hit, write, fetched by software prefetch.
0x81DB	LL_CACHE_HIT_WR_FPRFM	Last level cache demand access first hit, write, fetched by software prefetch.
0x81DC	L1D_CACHE_HIT_RW_FPRFM	Level 1 data cache demand access first hit, fetched by software prefetch.
0x81DD	L2D_CACHE_HIT_RW_FPRFM	Level 2 data cache demand access first hit, fetched by software prefetch.
0x81DE	L3D_CACHE_HIT_RW_FPRFM	Level 3 data cache demand access first hit, fetched by software prefetch.
0x81DF	LL_CACHE_HIT_RW_FPRFM	Last level cache demand access first hit, fetched by software prefetch.
0x81E0	L1I_CACHE_HIT_RD_FHWPRF	Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher.
0x81E1	L2I_CACHE_HIT_RD_FHWPRF	Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher.
0x81E4	L1D_CACHE_HIT_RD_FHWPRF	Level 1 data cache demand access first hit, read, fetched by hardware prefetcher.
0x81E5	L2D_CACHE_HIT_RD_FHWPRF	Level 2 data cache demand access first hit, read, fetched by hardware prefetcher.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81E6	L3D_CACHE_HIT_RD_FHWPRF	Level 3 data cache demand access first hit, read, fetched by hardware prefetcher.
0x81E7	LL_CACHE_HIT_RD_FHWPRF	Last level cache demand access first hit, read, fetched by hardware prefetcher.
0x81E8	L1D_CACHE_HIT_WR_FHWPRF	Level 1 data cache demand access first hit, write, fetched by hardware prefetcher.
0x81E9	L2D_CACHE_HIT_WR_FHWPRF	Level 2 data cache demand access first hit, write, fetched by hardware prefetcher.
0x81EA	L3D_CACHE_HIT_WR_FHWPRF	Level 3 data cache demand access first hit, write, fetched by hardware prefetcher.
0x81EB	LL_CACHE_HIT_WR_FHWPRF	Last level cache demand access first hit, write, fetched by hardware prefetcher.
0x81EC	L1D_CACHE_HIT_RW_FHWPRF	Level 1 data cache demand access first hit, fetched by hardware prefetcher.
0x81ED	L2D_CACHE_HIT_RW_FHWPRF	Level 2 data cache demand access first hit, fetched by hardware prefetcher.
0x81EE	L3D_CACHE_HIT_RW_FHWPRF	Level 3 data cache demand access first hit, fetched by hardware prefetcher.
0x81EF	LL_CACHE_HIT_RW_FHWPRF	Last level cache demand access first hit, fetched by hardware prefetcher.
0x81F0	L1I_CACHE_HIT_RD_FPRF	Level 1 instruction cache demand fetch first hit, fetched by prefetch.
0x81F1	L2I_CACHE_HIT_RD_FPRF	Level 2 instruction cache demand fetch first hit, fetched by prefetch.
0x81F4	L1D_CACHE_HIT_RD_FPRF	Level 1 data cache demand access first hit, read, fetched by prefetch.
0x81F5	L2D_CACHE_HIT_RD_FPRF	Level 2 data cache demand access first hit, read, fetched by prefetch.
0x81F6	L3D_CACHE_HIT_RD_FPRF	Level 3 data cache demand access first hit, read, fetched by prefetch.
0x81F7	LL_CACHE_HIT_RD_FPRF	Last level cache demand access first hit, read, fetched by prefetch.
0x81F8	L1D_CACHE_HIT_WR_FPRF	Level 1 data cache demand access first hit, write, fetched by prefetch.
0x81F9	L2D_CACHE_HIT_WR_FPRF	Level 2 data cache demand access first hit, write, fetched by prefetch.
0x81FA	L3D_CACHE_HIT_WR_FPRF	Level 3 data cache demand access first hit, write, fetched by prefetch.
0x81FB	LL_CACHE_HIT_WR_FPRF	Last level cache demand access first hit, write, fetched by prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81FC	L1D_CACHE_HIT_RW_FPRF	Level 1 data cache demand access first hit, fetched by prefetch.
0x81FD	L2D_CACHE_HIT_RW_FPRF	Level 2 data cache demand access first hit, fetched by prefetch.
0x81FE	L3D_CACHE_HIT_RW_FPRF	Level 3 data cache demand access first hit, fetched by prefetch.
0x81FF	LL_CACHE_HIT_RW_FPRF	Last level cache demand access first hit, fetched by prefetch.
0x8200	L1I_CACHE_HIT	Level 1 instruction cache hit.
0x8201	L2I_CACHE_HIT	Level 2 instruction cache hit.
0x8204	L1D_CACHE_HIT	Level 1 data cache hit.
0x8205	L2D_CACHE_HIT	Level 2 data cache hit.
0x8206	L3D_CACHE_HIT	Level 3 data cache hit.
0x8207	LL_CACHE_HIT	Last level cache hit.
0x8208	L1I_CACHE_HIT_PRFM	Level 1 instruction cache software prefetch hit.
0x8209	L2I_CACHE_HIT_PRFM	Level 2 instruction cache software prefetch hit.
0x820C	L1D_CACHE_HIT_PRFM	Level 1 data cache software prefetch hit.
0x820D	L2D_CACHE_HIT_PRFM	Level 2 data cache software prefetch hit.
0x820E	L3D_CACHE_HIT_PRFM	Level 3 data cache software prefetch hit.
0x820F	LL_CACHE_HIT_PRFM	Last level cache software prefetch hit.
0x8214	L1D_CACHE_HITM_RD	Level 1 data cache demand access hit modified, read.
0x8215	L2D_CACHE_HITM_RD	Level 2 data cache demand access hit modified, read.
0x8216	L3D_CACHE_HITM_RD	Level 3 data cache demand access hit modified, read.
0x8217	LL_CACHE_HITM_RD	Last level cache demand access hit modified, read.
0x8218	L1D_CACHE_HITM_WR	Level 1 data cache demand access hit modified, write.
0x8219	L2D_CACHE_HITM_WR	Level 2 data cache demand access hit modified, write.
0x821A	L3D_CACHE_HITM_WR	Level 3 data cache demand access hit modified, write.
0x821B	LL_CACHE_HITM_WR	Last level cache demand access hit modified, write.
0x821C	L1D_CACHE_HITM_RW	Level 1 data cache demand access hit modified.
0x821D	L2D_CACHE_HITM_RW	Level 2 data cache demand access hit modified.
0x821E	L3D_CACHE_HITM_RW	Level 3 data cache demand access hit modified.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x821F	LL_CACHE_HITM_RW	Last level cache demand access hit modified.
0x8224	DSNP_HITM_RD	Snoop hit, demand data read, modified.
0x8225	DSNP_HITM_NEAR_RD	Snoop hit in near local cache, demand data read, modified.
0x8226	DSNP_HITM_FAR_RD	Snoop hit in far local cache, demand data read, modified.
0x8227	DSNP_HITM_REMOTE_RD	Snoop hit in remote cache, demand data read, modified.
0x8228	DSNP_HITM_WR	Snoop hit, demand data write, modified.
0x8229	DSNP_HITM_NEAR_WR	Snoop hit in near local cache, demand data write, modified.
0x822A	DSNP_HITM_FAR_WR	Snoop hit in far local cache, demand data write, modified.
0x822B	DSNP_HITM_REMOTE_WR	Snoop hit in remote cache, demand data write, modified.
0x822C	DSNP_HITM_RW	Snoop hit, demand data access, modified.
0x822D	DSNP_HITM_NEAR_RW	Snoop hit in near local cache, demand data access, modified.
0x822E	DSNP_HITM_FAR_RW	Snoop hit in far local cache, demand data access, modified.
0x822F	DSNP_HITM_REMOTE_RW	Snoop hit in remote cache, demand data access, modified.
0x8230	LOCAL_MEM	Access to memory attached to this device.
0x8231	LOCAL_MEM_RD	Access to memory attached to this device, demand access, read.
0x8232	LOCAL_MEM_WR	Access to memory attached to this device, demand access, write.
0x8233	LOCAL_MEM_RW	Access to memory attached to this device, demand access.
0x8234	LOCAL_MEM_PRFM	Access to memory attached to this device, software prefetch.
0x8235	LOCAL_MEM_LD_RETIRED	Load instruction architecturally executed, access to memory attached to this device.
0x8236	LOCAL_MEM_ST_RETIRED	Store instruction architecturally executed, access to memory attached to this device.
0x8237	LOCAL_MEM_LDST_RETIRED	Load or store instruction architecturally executed, access to memory attached to this device.
0x8238	REMOTE_MEM	Access to memory attached to a remote device.
0x8239	REMOTE_MEM_RD	Access to memory attached to a remote device, demand access, read.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x823A	REMOTE_MEM_WR	Access to memory attached to a remote device, demand access, write.
0x823B	REMOTE_MEM_RW	Access to memory attached to a remote device, demand access.
0x823C	REMOTE_MEM_PRFM	Access to memory attached to a remote device, software prefetch.
0x823D	REMOTE_MEM_LD_RETIRED	Load instruction architecturally executed, access to memory attached to a remote device.
0x823E	REMOTE_MEM_ST_RETIRED	Store instruction architecturally executed, access to memory attached to a remote device.
0x823F	REMOTE_MEM_LDST_RETIRED	Load or store instruction architecturally executed, access to memory attached to a remote device.
0x8240	L1I_LFB_HIT_RD	Level 1 instruction cache demand fetch line-fill buffer hit.
0x8241	L2I_LFB_HIT_RD	Level 2 instruction cache demand fetch line-fill buffer hit.
0x8244	L1D_LFB_HIT_RD	Level 1 data cache demand access line-fill buffer hit, read.
0x8245	L2D_LFB_HIT_RD	Level 2 data cache demand access line-fill buffer hit, read.
0x8246	L3D_LFB_HIT_RD	Level 3 data cache demand access line-fill buffer hit, read.
0x8247	LL_LFB_HIT_RD	Last level cache demand access line-fill buffer hit, read.
0x8248	L1D_LFB_HIT_WR	Level 1 data cache demand access line-fill buffer hit, write.
0x8249	L2D_LFB_HIT_WR	Level 2 data cache demand access line-fill buffer hit, write.
0x824A	L3D_LFB_HIT_WR	Level 3 data cache demand access line-fill buffer hit, write.
0x824B	LL_LFB_HIT_WR	Last level cache demand access line-fill buffer hit, write.
0x824C	L1D_LFB_HIT_RW	Level 1 data cache demand access line-fill buffer hit.
0x824D	L2D_LFB_HIT_RW	Level 2 data cache demand access line-fill buffer hit.
0x824E	L3D_LFB_HIT_RW	Level 3 data cache demand access line-fill buffer hit.
0x824F	LL_LFB_HIT_RW	Last level cache demand access line-fill buffer hit.
0x8250	L1I_LFB_HIT_RD_FPRFM	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch.
0x8251	L2I_LFB_HIT_RD_FPRFM	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8254	L1D_LFB_HIT_RD_FPRFM	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8255	L2D_LFB_HIT_RD_FPRFM	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8256	L3D_LFB_HIT_RD_FPRFM	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8257	LL_LFB_HIT_RD_FPRFM	Last level cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8258	L1D_LFB_HIT_WR_FPRFM	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x8259	L2D_LFB_HIT_WR_FPRFM	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x825A	L3D_LFB_HIT_WR_FPRFM	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x825B	LL_LFB_HIT_WR_FPRFM	Last level cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x825C	L1D_LFB_HIT_RW_FPRFM	Level 1 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x825D	L2D_LFB_HIT_RW_FPRFM	Level 2 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x825E	L3D_LFB_HIT_RW_FPRFM	Level 3 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x825F	LL_LFB_HIT_RW_FPRFM	Last level cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x8260	L1I_LFB_HIT_RD_FHWPRF	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.
0x8261	L2I_LFB_HIT_RD_FHWPRF	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.
0x8264	L1D_LFB_HIT_RD_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8265	L2D_LFB_HIT_RD_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8266	L3D_LFB_HIT_RD_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8267	LL_LFB_HIT_RD_FHWPRF	Last level cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8268	L1D_LFB_HIT_WR_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x8269	L2D_LFB_HIT_WR_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x826A	L3D_LFB_HIT_WR_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x826B	LL_LFB_HIT_WR_FHWPRF	Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x826C	L1D_LFB_HIT_RW_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x826D	L2D_LFB_HIT_RW_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x826E	L3D_LFB_HIT_RW_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x826F	LL_LFB_HIT_RW_FHWPRF	Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x8270	L1I_LFB_HIT_RD_FPRF	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch.
0x8271	L2I_LFB_HIT_RD_FPRF	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch.
0x8274	L1D_LFB_HIT_RD_FPRF	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8275	L2D_LFB_HIT_RD_FPRF	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8276	L3D_LFB_HIT_RD_FPRF	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8277	LL_LFB_HIT_RD_FPRF	Last level cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8278	L1D_LFB_HIT_WR_FPRF	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
0x8279	L2D_LFB_HIT_WR_FPRF	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x827A	L3D_LFB_HIT_WR_FPRF	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
0x827B	LL_LFB_HIT_WR_FPRF	Last level cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
0x827C	L1D_LFB_HIT_RW_FPRF	Level 1 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x827D	L2D_LFB_HIT_RW_FPRF	Level 2 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x827E	L3D_LFB_HIT_RW_FPRF	Level 3 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x827F	LL_LFB_HIT_RW_FPRF	Last level cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x8280	L1I_CACHE_PRF	Level 1 instruction cache, prefetch hit.
0x8281	L2I_CACHE_PRF	Level 2 instruction cache, prefetch hit.
0x8284	L1D_CACHE_PRF	Level 1 data cache, prefetch hit.
0x8285	L2D_CACHE_PRF	Level 2 data cache, prefetch hit.
0x8286	L3D_CACHE_PRF	Level 3 data cache, prefetch hit.
0x8287	LL_CACHE_PRF	Last level cache, prefetch hit.
0x8288	L1I_CACHE_REFILL_PRF	Level 1 instruction cache refill, prefetch hit.
0x8289	L2I_CACHE_REFILL_PRF	Level 2 instruction cache refill, prefetch hit.
0x828C	L1D_CACHE_REFILL_PRF	Level 1 data cache refill, prefetch hit.
0x828D	L2D_CACHE_REFILL_PRF	Level 2 data cache refill, prefetch hit.
0x828E	L3D_CACHE_REFILL_PRF	Level 3 data cache refill, prefetch hit.
0x828F	LL_CACHE_REFILL_PRF	Last level cache refill, prefetch hit.
0x8290	ISNP_HIT_PRF	Snoop hit, instruction prefetch.
0x8291	ISNP_HIT_NEAR_PRF	Snoop hit in near local cache, instruction prefetch.
0x8292	ISNP_HIT_FAR_PRF	Snoop hit in far local cache, instruction prefetch.
0x8293	ISNP_HIT_REMOTE_PRF	Snoop hit in remote cache, instruction prefetch.
0x8294	DSNP_HIT_PRF	Snoop hit, data prefetch.
0x8295	DSNP_HIT_NEAR_PRF	Snoop hit in near local cache, data prefetch.
0x8296	DSNP_HIT_FAR_PRF	Snoop hit in far local cache, data prefetch.
0x8297	DSNP_HIT_REMOTE_PRF	Snoop hit in remote cache, data prefetch.
0x8298	LL_CACHE_RW	Last level cache demand access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8299	LL_CACHE_PRFM	Last level cache software prefetch.
0x829A	LL_CACHE_REFILL	Last level cache refill.
0x829B	LL_CACHE_REFILL_PRFM	Last level cache refill, software prefetch.
0x829C	LL_CACHE_WB	Last level cache write-back.
0x829D	LL_CACHE_WR	Last level cache access, write.
0x829F	LL_CACHE_REFILL_WR	Last level cache refill, write.
0x82A0	MEM_ACCESS_RW	Data memory access, demand access.
0x82A1	INST_FETCH_RD	Instruction memory access, demand fetch.
0x82A2	MEM_ACCESS_PRFM	Data memory access, software prefetch.
0x82A3	INST_FETCH_PRFM	Instruction memory access, software prefetch.
0x82A4	ASE_SVE_RETIRED	Instruction architecturally executed, Advanced SIMD data processing or SVE data processing.
0x82A8	LD_ANY_RETIRED	Instruction architecturally executed, load.
0x82A9	ST_ANY_RETIRED	Instruction architecturally executed, store.
0x82AA	LDST_ANY_RETIRED	Instruction architecturally executed, load or store.
0x82AB	DP_RETIRED	Instruction architecturally executed, integer data processing.
0x82AC	ASE_RETIRED	Instruction architecturally executed, Advanced SIMD data processing.
0x82AD	VFP_RETIRED	Instruction architecturally executed, scalar floating-point data processing.
0x82AE	SVE_RETIRED	Instruction architecturally executed, SVE data processing.
0x82AF	CRYPTO_RETIRED	Instruction architecturally executed, cryptographic data processing.
0x82B0	L1I_CACHE_MISS_RETIRED	Instruction architecturally executed, miss in Level 1 instruction cache.
0x82B1	L2I_CACHE_MISS_RETIRED	Instruction architecturally executed, miss in Level 2 instruction cache.
0x82B3	PRF_RETIRED	Instruction architecturally executed, prefetch.
0x82B4	L1D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 1 data cache.
0x82B5	L2D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 2 data cache.
0x82B6	L3D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 3 data cache.
0x82B7	LL_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Last level cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82B8	L1D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 1 data cache.
0x82B9	L2D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 2 data cache.
0x82BA	L3D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 3 data cache.
0x82BB	LL_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Last level cache.
0x82BC	L1D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 1 data cache.
0x82BD	L2D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 2 data cache.
0x82BE	L3D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 3 data cache.
0x82BF	LL_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Last level cache.
0x82C4	L1D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 1 data cache.
0x82C5	L2D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 2 data cache.
0x82C6	L3D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 3 data cache.
0x82C7	LL_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Last level cache.
0x82C8	L1D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 1 data cache.
0x82C9	L2D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 2 data cache.
0x82CA	L3D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 3 data cache.
0x82CB	LL_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Last level cache.
0x82CC	L1D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 1 data cache.
0x82CD	L2D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 2 data cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82CE	L3D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 3 data cache.
0x82CF	LL_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Last level cache.
0x82D0	L1I_LFB_HIT_RETIRED	Instruction architecturally executed, line-fill buffer hit in Level 1 instruction cache.
0x82D1	L2I_LFB_HIT_RETIRED	Instruction architecturally executed, line-fill buffer hit in Level 2 instruction cache.
0x82D4	L1D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
0x82D5	L2D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
0x82D6	L3D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
0x82D7	LL_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Last level cache.
0x82D8	L1D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
0x82D9	L2D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
0x82DA	L3D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
0x82DB	LL_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Last level cache.
0x82DC	L1D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
0x82DD	L2D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
0x82DE	L3D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
0x82DF	LL_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Last level cache.
0x82E0	L1I_CACHE_HIT_RETIRED	Instruction architecturally executed, hit in Level 1 instruction cache.
0x82E1	L2I_CACHE_HIT_RETIRED	Instruction architecturally executed, hit in Level 2 instruction cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82E4	L1D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 1 data cache.
0x82E5	L2D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 2 data cache.
0x82E6	L3D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 3 data cache.
0x82E7	LL_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Last level cache.
0x82E8	L1D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 1 data cache.
0x82E9	L2D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 2 data cache.
0x82EA	L3D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 3 data cache.
0x82EB	LL_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Last level cache.
0x82EC	L1D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 1 data cache.
0x82ED	L2D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 2 data cache.
0x82EE	L3D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 3 data cache.
0x82EF	LL_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Last level cache.
0x82F0	ITLB_HIT_RETIRED	Instruction architecturally executed, no translation table walk.
0x82F1	DTLB_HIT_LD_RETIRED	Load instruction architecturally executed, no translation table walk.
0x82F2	DTLB_HIT_ST_RETIRED	Store instruction architecturally executed, no translation table walk.
0x82F3	DTLB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, no translation table walk.
0x82F4	ITLB_WALK_RETIRED	Instruction architecturally executed, at least one translation table walk.
0x82F5	DTLB_WALK_LD_RETIRED	Load instruction architecturally executed, at least one translation table walk.
0x82F6	DTLB_WALK_ST_RETIRED	Store instruction architecturally executed, at least one translation table walk.
0x82F7	DTLB_WALK_LDST_RETIRED	Load or store instruction architecturally executed, at least one translation table walk.
0x82F8	DTLB_WALK_PRF	Data TLB prefetch, with at least one translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82F9	ITLB_WALK_PRF	Instruction TLB prefetch, with at least one translation table walk.
0x82FA	DTLB_WALK_HWPRF	Data TLB hardware prefetch, with at least one translation table walk.
0x82FB	ITLB_WALK_HWPRF	Instruction TLB hardware prefetch, with at least one translation table walk.
0x82FC	L1D_TLB_PRF	Level data TLB access, prefetch.
0x82FD	L1I_TLB_PRF	Level instruction TLB access, prefetch.
0x82FE	L1D_TLB_HWPRF	Level data TLB access, hardware prefetch.
0x82FF	L1I_TLB_HWPRF	Level instruction TLB access, hardware prefetch.
0x8304	DSNP_HITM_LD_RETIRED	Load instruction architecturally executed, snoop hit.
0x8305	DSNP_HITM_NEAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in near cache.
0x8306	DSNP_HITM_FAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in far cache.
0x8307	DSNP_HITM_REMOTE_LD_RETIRED	Load instruction architecturally executed, snoop hit in remote cache.
0x8308	DSNP_HITM_ST_RETIRED	Store instruction architecturally executed, snoop hit.
0x8309	DSNP_HITM_NEAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in near cache.
0x830A	DSNP_HITM_FAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in far cache.
0x830B	DSNP_HITM_REMOTE_ST_RETIRED	Store instruction architecturally executed, snoop hit in remote cache.
0x830C	DSNP_HITM_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit.
0x830D	DSNP_HITM_NEAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in near cache.
0x830E	DSNP_HITM_FAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in far cache.
0x830F	DSNP_HITM_REMOTE_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in remote cache.
0x8310	ISNP_HIT_RETIRED	Instruction architecturally executed, instruction fetch snoop hit.
0x8311	ISNP_HIT_NEAR_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in near cache.
0x8312	ISNP_HIT_FAR_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in far cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8313	ISNP_HIT_REMOTE_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in remote cache.
0x8314	DSNP_HIT_LD_RETIRED	Load instruction architecturally executed, snoop hit.
0x8315	DSNP_HIT_NEAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in near cache.
0x8316	DSNP_HIT_FAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in far cache.
0x8317	DSNP_HIT_REMOTE_LD_RETIRED	Load instruction architecturally executed, snoop hit in remote cache.
0x8318	DSNP_HIT_ST_RETIRED	Store instruction architecturally executed, snoop hit.
0x8319	DSNP_HIT_NEAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in near cache.
0x831A	DSNP_HIT_FAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in far cache.
0x831B	DSNP_HIT_REMOTE_ST_RETIRED	Store instruction architecturally executed, snoop hit in remote cache.
0x831C	DSNP_HIT_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit.
0x831D	DSNP_HIT_NEAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in near cache.
0x831E	DSNP_HIT_FAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in far cache.
0x831F	DSNP_HIT_REMOTE_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in remote cache.
0x8320	L1D_CACHE_REFILL_PERCYC	Level 1 data or unified cache refills in progress.
0x8321	L2D_CACHE_REFILL_PERCYC	Level 2 data or unified cache refills in progress.
0x8322	L3D_CACHE_REFILL_PERCYC	Level 3 data or unified cache refills in progress.
0x8324	L1I_CACHE_REFILL_PERCYC	Level 1 instruction or unified cache refills in progress.
0x8325	L2I_CACHE_REFILL_PERCYC	Level 2 instruction or unified cache refills in progress.
0x8328	REMOTE_CACHE_HIT_RD	Remote cache demand access hit, read.
0x8329	REMOTE_CACHE_HITM_RD	Remote cache demand access hit modified, read.
0x832A	REMOTE_LFB_HIT_RD	Remote cache demand access line-fill buffer hit, read.
0x832B	REMOTE_CACHE_HIT_PRFM	Remote cache software prefetch hit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x832C	STALL_BACKEND_ATOMIC_FP	Backend stall cycles, atomic in-memory floating-point load or store.
0x832D	LSE_FP_LD_SPEC	Atomic in-memory floating-point Operation speculatively executed, load.
0x832E	LSE_FP_ST_SPEC	Atomic in-memory floating-point Operation speculatively executed, store.
0x832F	LSE_FP_LDST_SPEC	Atomic in-memory floating-point Operation speculatively executed, load or store.
0x8330	L1GCS_CACHE	Level 1 GCS cache access.
0x8331	L1GCS_CACHE_RW	Level 1 GCS cache demand access.
0x8332	L1GCS_CACHE_HWPRF	Level 1 GCS cache hardware prefetch.
0x8334	L1GCS_CACHE_MISS	Level 1 GCS cache demand access miss.
0x8335	L1GCS_CACHE_MISS_RD	Level 1 GCS cache demand access read miss.
0x8336	L1GCS_CACHE_HIT_RW	Level 1 GCS cache demand access hit.
0x8337	L1GCS_CACHE_HIT_RW_FHWPRF	Level 1 GCS cache demand access first hit, fetched by GCS hardware prefetcher.
0x8338	L1GCS_CACHE_REFILL	Level 1 GCS cache refill.
0x8339	L1GCS_CACHE_REFILL_HWPRF	Level 1 GCS cache refill, hardware prefetch.
0x833A	L1GCS_CACHE_REFILL_PERCYC	Level 1 GCS cache refills in progress.
0x833C	L1GCS_LFB_HIT_RW	Level 1 GCS cache demand access line-fill buffer hit.
0x833D	L1GCS_LFB_HIT_RW_FHWPRF	Level 1 GCS cache demand access line-fill buffer first hit, recently fetched by GCS hardware prefetcher.
0x833E	L1GCS_CACHE_INVAL	Level 1 GCS cache invalidate.
0x8340	L1GCS_TLB	Level 1 GCS TLB access.
0x8341	L1GCS_TLB_RW	Level 1 GCS demand TLB access.
0x8342	L1GCS_TLB_HWPRF	Level 1 GCS demand TLB access, GCS hardware prefetch.
0x8344	GCSTLB_WALK	GCS TLB access with at least one translation table walk.
0x8345	GCSTLB_WALK_RW	GCS TLB demand access with at least one translation table walk.
0x8346	GCSTLB_WALK_PERCYC	Translation table walks in progress.
0x8347	GCSTLB_WALK_STEP	GCS TLB translation table walk, step.
0x8348	SAMPLE_FEED_FP	Statistical Profiling sample taken, floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8349	SAMPLE_FEED_SIMD	Statistical Profiling sample taken, SIMD.
0x8350	STALL_BACKEND_L1GCS	Backend stall cycles, Level 1 GCS cache.
0x8351	STALL_BACKEND_GCSTLB	Backend stall cycles, GCS TLB.
0x8352	SME_FP_SPEC	Operation speculatively executed, SME floating-point, data-processing.
0x8353	SE_FP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point, data-processing.
0x8354	SVE_CMP_IGRANULE_SPEC	Operation speculatively executed, SVE in-granule compare.
0x8355	SVE_CMP_XGRANULE_SPEC	Operation speculatively executed, SVE cross-granule compare.
0x8356	SVE_BITPERM_SPEC	Operation speculatively executed, SVE in-element permute.
0x8358	SME_RETIRED	Instruction architecturally executed, SME data processing.
0x8359	SE_RETIRED	Instruction architecturally executed, Advanced SIMD, SVE, or SME data processing.
0x835A	SME_INST_RETIRED	Instruction architecturally executed, SME.
0x835B	SE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD, SVE, or SME.
0x835C	SME_SPEC	Operation speculatively executed, SME data processing.
0x835D	SE_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME data processing.
0x835E	SME_INST_SPEC	Operation speculatively executed, SME.
0x835F	SE_INST_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME.
0x8360	SME_INT8_SPEC	Operation speculatively executed, SME 8-bit integer, data-processing.
0x8361	SE_INT8_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 8-bit integer, data-processing.
0x8362	SME_FP_BF16_SPEC	Operation speculatively executed, SME BFloat16 floating-point, data-processing.
0x8363	SE_FP_BF16_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD BFloat16 floating-point, data-processing.
0x8364	SME_INT16_SPEC	Operation speculatively executed, SME 16-bit integer, data-processing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8365	SE_INT16_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 16-bit integer, data-processing.
0x8366	SME_FP_HP_SPEC	Operation speculatively executed, SME half-precision floating-point, data-processing.
0x8367	SE_FP_HP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD half-precision floating-point, data-processing.
0x8368	SME_INT32_SPEC	Operation speculatively executed, SME 32-bit integer, data-processing.
0x8369	SE_INT32_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 32-bit integer, data-processing.
0x836A	SME_FP_SP_SPEC	Operation speculatively executed, SME single-precision floating-point, data-processing.
0x836B	SE_FP_SP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD single-precision floating-point, data-processing.
0x836C	SME_INT64_SPEC	Operation speculatively executed, SME 64-bit integer, data-processing.
0x836D	SE_INT64_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 64-bit integer, data-processing.
0x836E	SME_FP_DP_SPEC	Operation speculatively executed, SME double-precision floating-point, data-processing.
0x836F	SE_FP_DP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD double-precision floating-point, data-processing.
0x8370	SME_FP_ADDSUB_SPEC	Operation speculatively executed, SME floating-point addition or subtraction.
0x8371	SE_FP_ADDSUB_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point addition or subtraction.
0x8372	SME_FP_FMA_SPEC	Operation speculatively executed, SME floating-point multiply-add or multiply-subtract.
0x8373	SE_FP_FMA_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point multiply-add or multiply-subtract.
0x8374	SME_FP_DOT_SPEC	Operation speculatively executed, SME floating-point dot product.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8375	SE_FP_DOT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point dot product.
0x8376	SME_FP_MOPA_SPEC	Operation speculatively executed, SME floating-point outer product and accumulate, or outer product and subtract.
0x8378	SME_INT_SPEC	Operation speculatively executed, SME integer, data-processing.
0x8379	SE_INT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer, data-processing.
0x837A	SME_INT_MUL_SPEC	Operation speculatively executed, SME integer multiply or multiply-accumulate.
0x837B	SE_INT_MUL_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer multiply or multiply-accumulate.
0x837C	SME_INT_DOT_SPEC	Operation speculatively executed, SME integer dot product.
0x837D	SE_INT_DOT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer dot product.
0x837E	SME_INT_MOPA_SPEC	Operation speculatively executed, SME integer outer product and accumulate, or outer product and subtract.
0x8380	ZA_ACTIVE	CPU cycles, ZA enabled.
0x8381	SME_PRED2_NOT_FULL_SPEC	Operation speculatively executed, SME 2D predicated with at least one inactive element.
0x8384	SME_PRED2_SPEC	Operation speculatively executed, SME 2D predicated.
0x8385	SME_PRED2_EMPTY_SPEC	Operation speculatively executed, SME 2D predicated with no active elements.
0x8386	SME_PRED2_FULL_SPEC	Operation speculatively executed, SME 2D predicated with all active elements.
0x8387	SME_PRED2_PARTIAL_SPEC	Operation speculatively executed, SME 2D predicated with partially active elements.
0x8388	SME_LDST_ZAREG_SPEC	SME ZA unpredicated load/store.
0x8389	SME_LD_ZAREG_SPEC	SME ZA unpredicated load.
0x838A	SME_ST_ZAREG_SPEC	SME ZA unpredicated store.
0x838C	SME_LDST_ZTREG_SPEC	SME ZT unpredicated load/store.
0x838D	SME_LD_ZTREG_SPEC	SME ZT unpredicated load.
0x838E	SME_ST_ZTREG_SPEC	SME ZT unpredicated store.
0x8390	SME_LDST_REG_SPEC	SME unpredicated load/store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8391	SME_LD_REG_SPEC	SME unpredicated load.
0x8392	SME_ST_REG_SPEC	SME unpredicated store.
0x8394	SME_LDST_TILE_SPEC	SME predicated tile load/store.
0x8395	SME_LD_TILE_SPEC	SME predicated tile load.
0x8396	SME_ST_TILE_SPEC	SME predicated tile store.
0x8398	ASE_LUT_SPEC	Lookup table operation speculatively executed, Advanced SIMD data processing.
0x8399	SVE_LUT_SPEC	Lookup table operation speculatively executed, SVE data processing.
0x839A	SME_LUT_SPEC	Lookup table operation speculatively executed, SME data processing.
0x839B	SE_LUT_SPEC	Lookup table operation speculatively executed, Advanced SIMD, SVE, or SME data processing.
0x83A0	N1_CACHE_HIT_RD	Cache at distance 1, demand access hit, read.
0x83A1	N2_CACHE_HIT_RD	Cache at distance 2, demand access hit, read.
0x83A2	N3_CACHE_HIT_RD	Cache at distance 3, demand access hit, read.
0x83A3	N4_CACHE_HIT_RD	Cache at distance 4, demand access hit, read.
0x83A4	N1_CACHE_HIT_PRFM	Cache at distance 1, software prefetch hit.
0x83A5	N2_CACHE_HIT_PRFM	Cache at distance 2, software prefetch hit.
0x83A6	N3_CACHE_HIT_PRFM	Cache at distance 3, software prefetch hit.
0x83A7	N4_CACHE_HIT_PRFM	Cache at distance 4, software prefetch hit.
0x83A8	N1_CACHE1_HIT_RD	Cache type 1 at distance 1 demand access hit, read.
0x83A9	N2_CACHE1_HIT_RD	Cache type 1 at distance 2 demand access hit, read.
0x83AA	N3_CACHE1_HIT_RD	Cache type 1 at distance 3 demand access hit, read.
0x83AB	N4_CACHE1_HIT_RD	Cache type 1 at distance 4 demand access hit, read.
0x83AC	N1_CACHE1_HIT_PRFM	Cache type 1 at distance 1 software prefetch hit.
0x83AD	N2_CACHE1_HIT_PRFM	Cache type 1 at distance 2 software prefetch hit.
0x83AE	N3_CACHE1_HIT_PRFM	Cache type 1 at distance 3 software prefetch hit.
0x83AF	N4_CACHE1_HIT_PRFM	Cache type 1 at distance 4 software prefetch hit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83B0	N1_CACHE2_HIT_RD	Cache type 2 at distance 1 demand access hit, read.
0x83B1	N2_CACHE2_HIT_RD	Cache type 2 at distance 2 demand access hit, read.
0x83B2	N3_CACHE2_HIT_RD	Cache type 2 at distance 3 demand access hit, read.
0x83B3	N4_CACHE2_HIT_RD	Cache type 2 at distance 4 demand access hit, read.
0x83B4	N1_CACHE2_HIT_PRFM	Cache type 2 at distance 1 software prefetch hit.
0x83B5	N2_CACHE2_HIT_PRFM	Cache type 2 at distance 2 software prefetch hit.
0x83B6	N3_CACHE2_HIT_PRFM	Cache type 2 at distance 3 software prefetch hit.
0x83B7	N4_CACHE2_HIT_PRFM	Cache type 2 at distance 4 software prefetch hit.
0x83B8	N1_CACHE3_HIT_RD	Cache type 3 at distance 1 demand access hit, read.
0x83B9	N2_CACHE3_HIT_RD	Cache type 3 at distance 2 demand access hit, read.
0x83BA	N3_CACHE3_HIT_RD	Cache type 3 at distance 3 demand access hit, read.
0x83BB	N4_CACHE3_HIT_RD	Cache type 3 at distance 4 demand access hit, read.
0x83BC	N1_CACHE3_HIT_PRFM	Cache type 3 at distance 1 software prefetch hit.
0x83BD	N2_CACHE3_HIT_PRFM	Cache type 3 at distance 2 software prefetch hit.
0x83BE	N3_CACHE3_HIT_PRFM	Cache type 3 at distance 3 software prefetch hit.
0x83BF	N4_CACHE3_HIT_PRFM	Cache type 3 at distance 4 software prefetch hit.
0x83C0	N1_CACHE_HITM_RD	Cache at distance 1, demand access hit modified, read.
0x83C1	N2_CACHE_HITM_RD	Cache at distance 2, demand access hit modified, read.
0x83C2	N3_CACHE_HITM_RD	Cache at distance 3, demand access hit modified, read.
0x83C3	N4_CACHE_HITM_RD	Cache at distance 4, demand access hit modified, read.
0x83C4	N1_LFB_HIT_RD	Cache at distance 1 demand access line-fill buffer hit, read.
0x83C5	N2_LFB_HIT_RD	Cache at distance 2 demand access line-fill buffer hit, read.
0x83C6	N3_LFB_HIT_RD	Cache at distance 3 demand access line-fill buffer hit, read.
0x83C7	N4_LFB_HIT_RD	Cache at distance 4 demand access line-fill buffer hit, read.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83C8	N1_CACHE1_HITM_RD	Cache type 1 at distance 1 demand access hit modified, read.
0x83C9	N2_CACHE1_HITM_RD	Cache type 1 at distance 2 demand access hit modified, read.
0x83CA	N3_CACHE1_HITM_RD	Cache type 1 at distance 3 demand access hit modified, read.
0x83CB	N4_CACHE1_HITM_RD	Cache type 1 at distance 4 demand access hit modified, read.
0x83CC	N1_LFB1_HIT_RD	Cache type 1 at distance 1 demand access line-fill buffer hit, read.
0x83CD	N2_LFB1_HIT_RD	Cache type 1 at distance 2 demand access line-fill buffer hit, read.
0x83CE	N3_LFB1_HIT_RD	Cache type 1 at distance 3 demand access line-fill buffer hit, read.
0x83CF	N4_LFB1_HIT_RD	Cache type 1 at distance 4 demand access line-fill buffer hit, read.
0x83D0	N1_CACHE2_HITM_RD	Cache type 2 at distance 1 demand access hit modified, read.
0x83D1	N2_CACHE2_HITM_RD	Cache type 2 at distance 2 demand access hit modified, read.
0x83D2	N3_CACHE2_HITM_RD	Cache type 2 at distance 3 demand access hit modified, read.
0x83D3	N4_CACHE2_HITM_RD	Cache type 2 at distance 4 demand access hit modified, read.
0x83D4	N1_LFB2_HIT_RD	Cache type 2 at distance 1 demand access line-fill buffer hit, read.
0x83D5	N2_LFB2_HIT_RD	Cache type 2 at distance 2 demand access line-fill buffer hit, read.
0x83D6	N3_LFB2_HIT_RD	Cache type 2 at distance 3 demand access line-fill buffer hit, read.
0x83D7	N4_LFB2_HIT_RD	Cache type 2 at distance 4 demand access line-fill buffer hit, read.
0x83D8	N1_CACHE3_HITM_RD	Cache type 3 at distance 1 demand access hit modified, read.
0x83D9	N2_CACHE3_HITM_RD	Cache type 3 at distance 2 demand access hit modified, read.
0x83DA	N3_CACHE3_HITM_RD	Cache type 3 at distance 3 demand access hit modified, read.
0x83DB	N4_CACHE3_HITM_RD	Cache type 3 at distance 4 demand access hit modified, read.
0x83DC	N1_LFB3_HIT_RD	Cache type 3 at distance 1 demand access line-fill buffer hit, read.
0x83DD	N2_LFB3_HIT_RD	Cache type 3 at distance 2 demand access line-fill buffer hit, read.
0x83DE	N3_LFB3_HIT_RD	Cache type 3 at distance 3 demand access line-fill buffer hit, read.
0x83DF	N4_LFB3_HIT_RD	Cache type 3 at distance 4 demand access line-fill buffer hit, read.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83E0	N1_MEM_RD	Access to memory at distance 1, demand access, read.
0x83E1	N2_MEM_RD	Access to memory at distance 2, demand access, read.
0x83E2	N3_MEM_RD	Access to memory at distance 3, demand access, read.
0x83E3	N4_MEM_RD	Access to memory at distance 4, demand access, read.
0x83E4	N1_MEM_PRFM	Access to memory at distance 1, software prefetch.
0x83E5	N2_MEM_PRFM	Access to memory at distance 2, software prefetch.
0x83E6	N3_MEM_PRFM	Access to memory at distance 3, software prefetch.
0x83E7	N4_MEM_PRFM	Access to memory at distance 4, software prefetch.
0x83E8	N1_MEM1_RD	Access to type 1 memory at distance 1, demand access, read.
0x83E9	N2_MEM1_RD	Access to type 1 memory at distance 2, demand access, read.
0x83EA	N3_MEM1_RD	Access to type 1 memory at distance 3, demand access, read.
0x83EB	N4_MEM1_RD	Access to type 1 memory at distance 4, demand access, read.
0x83EC	N1_MEM1_PRFM	Access to type 1 memory at distance 1, software prefetch.
0x83ED	N2_MEM1_PRFM	Access to type 1 memory at distance 2, software prefetch.
0x83EE	N3_MEM1_PRFM	Access to type 1 memory at distance 3, software prefetch.
0x83EF	N4_MEM1_PRFM	Access to type 1 memory at distance 4, software prefetch.
0x83F0	N1_MEM2_RD	Access to type 2 memory at distance 1, demand access, read.
0x83F1	N2_MEM2_RD	Access to type 2 memory at distance 2, demand access, read.
0x83F2	N3_MEM2_RD	Access to type 2 memory at distance 3, demand access, read.
0x83F3	N4_MEM2_RD	Access to type 2 memory at distance 4, demand access, read.
0x83F4	N1_MEM2_PRFM	Access to type 2 memory at distance 1, software prefetch.
0x83F5	N2_MEM2_PRFM	Access to type 2 memory at distance 2, software prefetch.
0x83F6	N3_MEM2_PRFM	Access to type 2 memory at distance 3, software prefetch.
0x83F7	N4_MEM2_PRFM	Access to type 2 memory at distance 4, software prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83F8	N1_MEM3_RD	Access to type 3 memory at distance 1, demand access, read.
0x83F9	N2_MEM3_RD	Access to type 3 memory at distance 2, demand access, read.
0x83FA	N3_MEM3_RD	Access to type 3 memory at distance 3, demand access, read.
0x83FB	N4_MEM3_RD	Access to type 3 memory at distance 4, demand access, read.
0x83FC	N1_MEM3_PRFM	Access to type 3 memory at distance 1, software prefetch.
0x83FD	N2_MEM3_PRFM	Access to type 3 memory at distance 2, software prefetch.
0x83FE	N3_MEM3_PRFM	Access to type 3 memory at distance 3, software prefetch.
0x83FF	N4_MEM3_PRFM	Access to type 3 memory at distance 4, software prefetch.
0x8400	ISNP_HIT_N1_RD	Snoop hit in cache at distance 1, demand instruction fetch.
0x8401	ISNP_HIT_N2_RD	Snoop hit in cache at distance 2, demand instruction fetch.
0x8402	ISNP_HIT_N3_RD	Snoop hit in cache at distance 3, demand instruction fetch.
0x8403	ISNP_HIT_N4_RD	Snoop hit in cache at distance 4, demand instruction fetch.
0x8404	DSNP_HIT_N1_RD	Snoop hit in cache at distance 1, demand data read.
0x8405	DSNP_HIT_N2_RD	Snoop hit in cache at distance 2, demand data read.
0x8406	DSNP_HIT_N3_RD	Snoop hit in cache at distance 3, demand data read.
0x8407	DSNP_HIT_N4_RD	Snoop hit in cache at distance 4, demand data read.
0x8408	DSNP_HIT_N1_WR	Snoop hit in cache at distance 1, demand data write.
0x8409	DSNP_HIT_N2_WR	Snoop hit in cache at distance 2, demand data write.
0x840A	DSNP_HIT_N3_WR	Snoop hit in cache at distance 3, demand data write.
0x840B	DSNP_HIT_N4_WR	Snoop hit in cache at distance 4, demand data write.
0x840C	DSNP_HIT_N1_RW	Snoop hit in cache at distance 1, demand data access.
0x840D	DSNP_HIT_N2_RW	Snoop hit in cache at distance 2, demand data access.
0x840E	DSNP_HIT_N3_RW	Snoop hit in cache at distance 3, demand data access.
0x840F	DSNP_HIT_N4_RW	Snoop hit in cache at distance 4, demand data access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8410	DSNP_HIT_N1_PRFM	Snoop hit in cache at distance 1, software data prefetch.
0x8411	DSNP_HIT_N2_PRFM	Snoop hit in cache at distance 2, software data prefetch.
0x8412	DSNP_HIT_N3_PRFM	Snoop hit in cache at distance 3, software data prefetch.
0x8413	DSNP_HIT_N4_PRFM	Snoop hit in cache at distance 4, software data prefetch.
0x8414	DSNP_HIT_N1_HWPRF	Snoop hit in cache at distance 1, hardware data prefetch.
0x8415	DSNP_HIT_N2_HWPRF	Snoop hit in cache at distance 2, hardware data prefetch.
0x8416	DSNP_HIT_N3_HWPRF	Snoop hit in cache at distance 3, hardware data prefetch.
0x8417	DSNP_HIT_N4_HWPRF	Snoop hit in cache at distance 4, hardware data prefetch.
0x8418	ISNP_HIT_N1_PRFM	Snoop hit in cache at distance 1, software instruction prefetch.
0x8419	ISNP_HIT_N2_PRFM	Snoop hit in cache at distance 2, software instruction prefetch.
0x841A	ISNP_HIT_N3_PRFM	Snoop hit in cache at distance 3, software instruction prefetch.
0x841B	ISNP_HIT_N4_PRFM	Snoop hit in cache at distance 4, software instruction prefetch.
0x841C	ISNP_HIT_N1_HWPRF	Snoop hit in cache at distance 1, hardware instruction prefetch.
0x841D	ISNP_HIT_N2_HWPRF	Snoop hit in cache at distance 2, hardware instruction prefetch.
0x841E	ISNP_HIT_N3_HWPRF	Snoop hit in cache at distance 3, hardware instruction prefetch.
0x841F	ISNP_HIT_N4_HWPRF	Snoop hit in cache at distance 4, hardware instruction prefetch.
0x8420	DSNP_HIT_N1_PRF	Snoop hit in cache at distance 1, data prefetch.
0x8421	DSNP_HIT_N2_PRF	Snoop hit in cache at distance 2, data prefetch.
0x8422	DSNP_HIT_N3_PRF	Snoop hit in cache at distance 3, data prefetch.
0x8423	DSNP_HIT_N4_PRF	Snoop hit in cache at distance 4, data prefetch.
0x8424	ISNP_HIT_N1_PRF	Snoop hit in cache at distance 1, instruction prefetch.
0x8425	ISNP_HIT_N2_PRF	Snoop hit in cache at distance 2, instruction prefetch.
0x8426	ISNP_HIT_N3_PRF	Snoop hit in cache at distance 3, instruction prefetch.
0x8427	ISNP_HIT_N4_PRF	Snoop hit in cache at distance 4, instruction prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8428	ISNP_HIT_N1	Snoop hit in cache at distance 1, instruction access.
0x8429	ISNP_HIT_N2	Snoop hit in cache at distance 2, instruction access.
0x842A	ISNP_HIT_N3	Snoop hit in cache at distance 3, instruction access.
0x842B	ISNP_HIT_N4	Snoop hit in cache at distance 4, instruction access.
0x842C	DSNP_HIT_N1	Snoop hit in cache at distance 1, data access.
0x842D	DSNP_HIT_N2	Snoop hit in cache at distance 2, data access.
0x842E	DSNP_HIT_N3	Snoop hit in cache at distance 3, data access.
0x842F	DSNP_HIT_N4	Snoop hit in cache at distance 4, data access.
0x8431	ASE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD pairwise or reduction.
0x8432	SVE_FP_PREDUCE_SPEC	Floating-point operation speculatively executed, SVE pairwise add step or pairwise reduce step.
0x8434	DSNP_HITM_N1_RD	Snoop hit in cache at distance 1, demand data read, modified.
0x8435	DSNP_HITM_N2_RD	Snoop hit in cache at distance 2, demand data read, modified.
0x8436	DSNP_HITM_N3_RD	Snoop hit in cache at distance 3, demand data read, modified.
0x8437	DSNP_HITM_N4_RD	Snoop hit in cache at distance 4, demand data read, modified.
0x8438	DSNP_HITM_N1_WR	Snoop hit in cache at distance 1, demand data write, modified.
0x8439	DSNP_HITM_N2_WR	Snoop hit in cache at distance 2, demand data write, modified.
0x843A	DSNP_HITM_N3_WR	Snoop hit in cache at distance 3, demand data write, modified.
0x843B	DSNP_HITM_N4_WR	Snoop hit in cache at distance 4, demand data write, modified.
0x843C	DSNP_HITM_N1_RW	Snoop hit in cache at distance 1, demand data access, modified.
0x843D	DSNP_HITM_N2_RW	Snoop hit in cache at distance 2, demand data access, modified.
0x843E	DSNP_HITM_N3_RW	Snoop hit in cache at distance 3, demand data access, modified.
0x843F	DSNP_HITM_N4_RW	Snoop hit in cache at distance 4, demand data access, modified.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8440	ASE_FP_SP_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is single-precision floating-point.
0x8442	ASE_FP_HP_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is half-precision floating-point.
0x8443	ASE_FP_BF16_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x8444	ASE_FP_FP8_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8448	ASE_SVE_FP_SP_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is single-precision floating-point.
0x844A	ASE_SVE_FP_HP_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is half-precision floating-point.
0x844B	ASE_SVE_FP_BF16_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x844C	ASE_SVE_FP_FP8_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8450	SE_FP_SP_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is single-precision floating-point.
0x8452	SE_FP_HP_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is half-precision floating-point.
0x8453	SE_FP_BF16_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x8454	SE_FP_FP8_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8458	SME_FP_SP_MIN_SPEC	SME data processing operation speculatively executed, smallest type is single-precision floating-point.
0x845A	SME_FP_HP_MIN_SPEC	SME data processing operation speculatively executed, smallest type is half-precision floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x845B	SME_FP_BF16_MIN_SPEC	SME data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x845C	SME_FP_FP8_MIN_SPEC	SME data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8460	SVE_FP_SP_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is single-precision floating-point.
0x8462	SVE_FP_HP_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is half-precision floating-point.
0x8463	SVE_FP_BF16_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x8464	SVE_FP_FP8_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x846B	FP_BF16_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point.
0x8470	FP_SP_MIN_SPEC	Floating-point operation speculatively executed, smallest type is single-precision floating-point.
0x8472	FP_HP_MIN_SPEC	Floating-point operation speculatively executed, smallest type is half-precision floating-point.
0x8473	FP_BF16_MIN_SPEC	Floating-point operation speculatively executed, smallest type is BFloat16 floating-point.
0x8474	FP_FP8_MIN_SPEC	Floating-point operation speculatively executed, smallest type is 8-bit floating-point.
0x8480	FP_SP_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point.
0x8482	FP_HP_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point.
0x8483	FP_BF16_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point.
0x8484	FP_FP8_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point.
0x8488	FP_SP_SCALE_MIN_OPS_SPEC	Scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x848A	FP_HP_SCALE_MIN_OPS_SPEC	Scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point.
0x848B	FP_BF16_SCALE_MIN_OPS_SPEC	Scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point.
0x848C	FP_FP8_SCALE_MIN_OPS_SPEC	Scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point.
0x8490	FP_SP_SCALE2_MIN_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, smallest type is single-precision floating-point.
0x8492	FP_HP_SCALE2_MIN_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, smallest type is half-precision floating-point.
0x8493	FP_BF16_SCALE2_MIN_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, smallest type is BFloat16 floating-point.
0x8494	FP_FP8_SCALE2_MIN_OPS_SPEC	Scalable tile arithmetic operations speculatively executed, smallest type is 8-bit floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.3 Common event numbers

The event numbers of the common architectural and microarchitectural events are reserved for the specified events. Each of these event numbers must either:

- Be used for its assigned event.
- Not be used.

However, see Required events.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from an IMPLEMENTATION DEFINED range.

Note

Arm might define other common architectural and microarchitectural event numbers. This is one reason why software must not assume that an event with an assigned common architectural or microarchitectural event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.

It is IMPLEMENTATION DEFINED which events, including Common events, are generated by IMPLEMENTATION DEFINED extensions to the architecture, including accesses to IMPLEMENTATION DEFINED System registers and IMPLEMENTATION DEFINED System instructions. However, the functionality of the IMPLEMENTATION DEFINED extension must be appropriate for the generated events.

The common events are described in the following sections:

- Common architectural events.
- Common microarchitectural events.

The supported common architectural and microarchitectural events in the ranges `0x0000-0x003F` and `0x4000-0x403F` are discoverable to software through:

- The PMCEID0_EL0 and PMCEID1_EL0 registers in AArch64 state.
- The PMCEID0, PMCEID1, PMCEID2, and PMCEID3 registers in AArch32 state.

Arm recommends that the value of 0 is used for the PMCEID0_EL0 or PMCEID1_EL0 bit corresponding to any event that an implementation never generates, even if the implementation is considered to support but never count the event.

Note

- For example, if an implementation never generates the L1D_CACHE_ALLOCATE event, event 31, Arm recommends that PMCEID0_EL0[31] is RAZ.
- In an implementation that supports both Execution states, each bit in the AArch64 PMCEID0_EL0 and PMCEID1_EL0 registers corresponds to a single bit in the AArch32 PMCEID0, PMCEID1, PMCEID2, and PMCEID3 registers, and corresponding bits must have the same behavior.

However, for some implementations, an event in the common events range might be generated by the system, meaning behavior can vary between systems. In such a case, the corresponding PMCEID*n*_EL0 bit might be RAO.

Event numbers that Table D14-2 shows as allocated for common architectural and microarchitectural events that are not described in Common architectural events and Common microarchitectural events are reserved. Future revisions of this Manual, or of the architecture, might assign these reserved values to additional common events. Events that do not require additional features in the PMU can be implemented retrospectively, meaning an implementation of a particular version of the PMU specification might support common events that are first defined in a later version of the PMU specification.

Note

The requirement that an event that is implemented retrospectively *does not require additional features in the PMU* means that it must be possible to represent the event *n* the PMEVTYPER<n>_EL0.evtCount field. This means, for example, that an implementation with a 10-bit PMEVTYPER<n>_EL0.evtCount field can only implement events with event numbers `0x0000-0x03FF`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.3.1 Common architectural events

The Common architectural events are:

0x0000, SW_INCR, Instruction architecturally executed, Condition code check pass, software increment

The counter counts each write to the PMSWINC_EL0 and PMSWINC register, for each implemented event counter <n>:

- In AArch64 state, if PMEVTYPER<n>_EL0.evtCount is 0x0000 then the counter counts each MSR write to PMSWINC_EL0 with bit [n] set to 1.
- In AArch32 state, if PMEVTYPER<n>.evtCount is 0x0000 then the counter counts each MCR write to PMSWINC with bit [n] set to 1.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 or PMSWINC register without an intervening Context synchronization event, then the counter is incremented twice.

Note

In a multithreaded implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts writes by all PEs that have the same affinity at level 1 and above.

PMCEID0_EL0[0] reads as 1 if this event is implemented and 0 otherwise.

0x0006, LD_RETIRED, Instruction architecturally executed, Condition code check pass, load

The counter counts each architecturally-executed memory-reading instruction counted by INST_RETIRED.

It is IMPLEMENTATION DEFINED whether the prefetch instructions counted by PRF_RETIRED are counted as Memory-reading instructions:

- If the prefetch instructions are counted as Memory-reading instructions, then they are counted by LD_RETIRED and LD_ANY_RETIRED.
- Otherwise, if the prefetch instructions are not counted as Memory-reading instructions by LD_RETIRED, then it is further IMPLEMENTATION DEFINED which one of the following applies:
  - They are counted as data-processing instructions.
  - They are counted by LDST_ANY_RETIRED.

Arm recommends that if a prefetch instruction is not implemented as a NOP and the PRF_RETIRED event is not implemented, then the instruction is counted as a Memory-reading instruction.

PMCEID0_EL0[6] reads as 1 if this event is implemented and 0 otherwise.

0x0007, ST_RETIRED, Instruction architecturally executed, Condition code check pass, store

The counter counts each architecturally-executed memory-writing instruction counted by INST_RETIRED.

The counter does not count a Store-Exclusive instruction that fails.

PMCEID0_EL0[7] reads as 1 if this event is implemented and 0 otherwise.

0x0008, INST_RETIRED, Instruction architecturally executed

The counter counts each architecturally-executed instruction.

If FEAT_PMUv3p9 is implemented, then the counter counts architecturally-executed MOVPRFX instructions.

Otherwise, it is IMPLEMENTATION DEFINED whether the counter counts architecturally-executed MOVPRFX instructions.

PMCEID0_EL0[8] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0009, EXC_TAKEN, Exception taken

The counter counts each exception taken.

The counter counts each exception taken.

Note

The counter counts the PE exceptions described in:

- For exceptions taken to an Exception level using AArch64, Exception entry.
- For exceptions taken to an Exception level using AArch32, AArch32 state exception descriptions.

PMCEID0_EL0[9] reads as 1 if this event is implemented and 0 otherwise.

0x000A, EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return

The counter counts each architecturally-executed exception return instruction.

Instructions defined in the following sections are counted as exception return instructions:

- For an exception return from an Exception level using AArch64, Exception return.
- For an exception return from an Exception level using AArch32, Exception return instructions.

It is CONSTRAINED UNPREDICTABLE whether the execution of an exception return instruction is counted if any of the following apply:

- Execution of the instruction is, itself, CONSTRAINED UNPREDICTABLE.
- Execution of the instruction sets PSTATE.IL and does not generate an exception return.

Note

- Examples of when an exception return instruction is CONSTRAINED UNPREDICTABLE are if the instruction is executed in AArch32 state at EL0 or in System mode.
- A particular consequence of this CONSTRAINED UNPREDICTABLE behavior is that an implementation that does not support AArch32 state at EL1 or higher does not have to count AArch32 `MOVS PC, LR`, and related instructions, as exception return instructions.

PMCEID0_EL0[10] reads as 1 if this event is implemented and 0 otherwise.

0x000B, CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR

The counter counts each `MSR` write to CONTEXTIDR_EL1 and `MCR` write to CONTEXTIDR.

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the first write is counted.

The counter counts only writes to these named registers. For example:

- When FEAT_VHE or FEAT_Debugv8p2 is implemented, the counter does not count writes using the register name CONTEXTIDR_EL2.
- When FEAT_VHE is implemented, the counter:
  - Counts each write using the register name CONTEXTIDR_EL1, including when executing at EL2 and the Effective value of HCR_EL2.E2H is 1.
  - Does not count writes using the register name CONTEXTIDR_EL12.
- When FEAT_NV2 is implemented, the counter counts writes using the register name CONTEXTIDR_EL1, including when executing at EL1 and the Effective value of HCR_EL2.{NV2, NV1, NV} is {1, 1, 1}.

PMCEID0_EL0[11] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x000C, PC_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, Software change of the PC

The counter counts each architecturally-executed Software change of the PC that is taken.

This includes all:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

Note

Conditional branches are only counted if the branch is taken.

If FEAT_PMUv3p9 is implemented, then the following instructions are not included as a Software change of the PC:

- A BRK or BKPT instruction.
- An UNDEFINED instruction that generates an exception.
- An exception-generating instruction, SVC, HVC, or SMC, that generates an exception.
- A Context synchronization barrier instruction, ISB.

If FEAT_PMUv3p9 is not implemented, then it is IMPLEMENTATION DEFINED whether any of these instructions are included as a Software change of the PC.

The counter does not increment for exceptions other than those explicitly identified in these lists.

If PC_WRITE_RETIRED and BR_SKIP_RETIRED are both implemented, the PE must treat the following types of instruction in the same way for both events:

- BRK and BKPT instructions.
- UNDEFINED instructions.
- The exception-generating instructions, SVC, HVC, and SMC.
- Context synchronization barrier instructions.

From Armv8.6, if BR_RETIRED is also implemented, the PE must also treat these types of instruction in the same way for the BR_RETIRED, PC_WRITE_RETIRED, and BR_SKIP_RETIRED events.

PMCEID0_EL0[12] reads as 1 if this event is implemented and 0 otherwise.

0x000D, BR_IMMED_RETIRED, Branch instruction architecturally executed, immediate

The counter counts each architecturally-executed immediate branch instruction.

The following instructions are counted as immediate branch instructions:

- For AArch32 state, the following instructions:
  - `B{<c>} <label>`.
  - `BL{<c>} <label>`.
  - `BLX{<c>} <label>`.
  - `CBZ <Rn>, <label>`.
  - `CBNZ <label>`.
- For AArch64 state, the following instructions:
  - `B <label>`.
  - `B.cond <label>`.
  - `BL <label>`.
  - `CBZ <Rn>, <label>`.
  - `CBNZ <Rn>, <label>`.
  - `TBZ <Rn>, <label>`.
  - `TBNZ <Rn>, <label>`.
  - `BC.cond <label>`.
  - `CB<cc> <Rn>, #<imm>, <label>`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `CB<cc> <Rn>, #<Rm>, <label>`.
- `CBB<cc> <Rn>, #<Rm>, <label>`.
- `CBH<cc> <Rn>, #<Rm>, <label>`.

Note

Conditional branches are always counted, regardless of whether the branch is taken or not taken.

If the Context synchronization barrier instruction ISB is counted as a Software change of the PC instruction by PC_WRITE_RETIRED, then it is IMPLEMENTATION DEFINED whether ISB is counted as an immediate branch instruction.

PMCEID0_EL0[13] reads as 1 if this event is implemented and 0 otherwise.

`0x000E`, BR_RETURN_RETIRED, Branch instruction architecturally executed, procedure return, taken

The counter counts each architecturally executed procedure return instruction.

The following instructions are counted as procedure return instructions:

- For AArch32 state, the following instructions:
  - `BX R14`.
  - `MOV PC, LR`.
  - `POP {..., PC}`.
  - `LDR PC, [SP], #offset`.
- For AArch64 state, the following instructions:
  - `RET`.
  - If FEAT_PAuth is implemented, `RETAA` and `RETAB`.
  - If FEAT_PAuth_LR is implemented, `RETAASPPC`, `RETABSPPC`, `RETAASPPCR`, and `RETABSPPCR`.

Note

The counter counts only the listed instructions as procedure returns. For example, it does not count the following AArch32 instructions as procedure return instructions:

- `BX R0`, because Rm != R14.
- `MOV PC, R0`, because Rm != R14.
- `LDM SP, {..., PC}`, because writeback is not specified.
- `LDR PC, [SP, #offset]`, because this specifies the wrong addressing mode.

PMCEID0_EL0[14] reads as 1 if this event is implemented and 0 otherwise.

`0x000F`, UNALIGNED_LDST_RETIRED, Instruction architecturally executed, Condition code check pass, unaligned load or store

The counter counts each memory-reading instruction or memory-writing instruction access that would generate an Alignment fault when Alignment fault checking is enabled.

The counter does not count accesses that would generate an SP alignment fault exception if the applicable stack pointer alignment check is enabled, unless that access would also generate an Alignment fault Data Abort exception if Alignment fault checking is enabled.

It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this event counts accesses that generate an exception, including accesses that do generate Alignment fault Data Abort exceptions.

PMCEID0_EL0[15] reads as 1 if this event is implemented and 0 otherwise.

`0x001C`, TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to TTBR

The counter counts each `MSR` write to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and `MCR` and `MCRR` write to TTBR0 and TTBR1 in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts MSR writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and MCR and MCRR writes to TTBR0 and TTBR1 in AArch32 state.

If the PE executes two writes to the same TTBR, without an intervening Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR, is counted.

The counter counts only writes to these registers by name. For example:

- If EL3 is implemented, the counter does not count writes using the register name TTBR0_EL3. However, when EL3 is using AArch32, the counter counts writes to either banked copy of TTBR0 and TTBR1.
- If EL2 is implemented, the counter does not count writes using the register names TTBR0_EL2, VTTBR_EL2, HTTBR, and VTTBR.
- When FEAT_VHE is implemented, the counter:
  - Counts each write using the register names TTBR0_EL1 and TTBR1_EL1, including when executing at EL2 and the Effective value of HCR_EL2.E2H is 1.
  - Does not count writes using the register names TTBR1_EL2, TTBR0_EL12, and TTBR1_EL12.
- When FEAT_NV2 is implemented, the counter counts each write using the register names TTBR0_EL1 and TTBR1_EL1, including when executing at EL1 and the Effective value of HCR_EL2.{NV2, NV1, NV} is {1, 1, 1}.
- If EL2 is implemented, the counter does not count writes using the register names TTBR0_EL2 and VTTBR_EL2.

PMCEID0_EL0[28] reads as 1 if this event is implemented and 0 otherwise.

0x001E, CHAIN, Chain a pair of event counters

Even-numbered counters never increment as a result of this event.

For an odd-numbered counter <n+1>, the counter increments when an event increments the preceding even-numbered counter <n> on the same PE causing unsigned overflow of bits [31:0] of the event counter <n>, and any of the following are true:

- FEAT_PMUv3p5 is not implemented.
- EL2 is not implemented and PMCR.LP is 0.
- EL2 is implemented, <n> is less than the Effective value of HDCR.HPMN, and PMCR.LP is 0.
- EL2 is implemented, <n> is greater than or equal to the Effective value of HDCR.HPMN, and HDCR.HLP is 0.

This means the CHAIN event can be used to link the odd-numbered counter with the preceding even-numbered counter to provide a 64-bit counter.

If EL2 is implemented and <n+1> is equal to the Effective value of HDCR.HPMN, then it is UNPREDICTABLE whether the counter counts.

If FEAT_PMUv3_EXTPMN is implemented and <n+1> is equal to the Effective value of PMCCR.EPMN, then it is UNPREDICTABLE whether the counter counts.

Note

- When FEAT_PMUv3p5 is not implemented, the CHAIN event allows software to use the N event counters as N 32-bit counters, N÷2 64-bit counters, or a mixture of 32-bit counters and 64-bit counters.
- The CHAIN event only counts overflows from the preceding even-numbered counter on the same PE. This means it ignores the Effective value of PMEVTYPER<n>_EL0.MT.
- The architecture does not provide atomic access to a pair of counters.

To filter the Exception levels and Security states in which the event is counted, software:

- Programs PMEVTYPER<n>_EL0 to count the event in the required conditions.
- Programs PMEVTYPER<n+1>_EL0 to count the CHAIN event in all Exception levels and states.

The PE might ignore the filter settings for the CHAIN event and behave as if they are set to count in all Exception levels and states. If software does not program the event in this way, the count becomes UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The architecture does not define the latency between the low counter overflowing and the high counter incrementing the CHAIN event. There is no requirement for updates to occur synchronously, but software reading or enabling the counter pair by reading the low counter first and the high counter second, with an intervening Context synchronization event, will not observe the low counter incrementing and overflowing for the event and the high counter not incrementing for the resulting CHAIN event. This means that the ISB executed after reading the low counter ensures the completion of the update of the high counter by the CHAIN event.

PMCEID0_EL0[30] reads as 1 if this event is implemented and 0 otherwise.

`0x0021`, BR_RETIRED, Instruction architecturally executed, branch

The counter counts each branch instruction on the architecturally executed path that would incur cost if mispredicted.

If FEAT_PMUv3p9 is implemented:

- Unconditional direct branch instructions are counted.
- Exception return instructions are counted.
- Exception-generating instructions are not counted.
- Context synchronization instructions are not counted.

If FEAT_PMUv3p9 is not implemented, then it is IMPLEMENTATION DEFINED whether the counter increments for any of these instructions.

The counter counts all other branch instructions, memory-reading instructions and data-processing instructions that explicitly write to the PC, at retirement.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

Arm recommends that BR_RETIRED counts Unconditional direct branch instructions and Exception return instructions.

From Armv8.6, if PC_WRITE_RETIRED and BR_RETIRED are both implemented, the PE must treat the following types of instruction in the same way for both events:

- `BRK` and `BKPT` instructions.
- UNDEFINED instructions.
- The exception-generating instructions, `SVC`, `HVC`, and `SMC`.
- Context synchronization barrier instructions.

PMCEID1_EL0[1] reads as 1 if this event is implemented and 0 otherwise.

`0x0081`, EXC_UNDEF, Exception taken, other synchronous

The counter counts each exception counted by EXC_TAKEN that is Taken locally and is not counted as any of the following:

- Exception taken, Supervisor Call (EXC_SVC).
- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort (EXC_PABORT).
- Exception taken, Data Abort or SError (EXC_DABORT).
- Exception taken, IRQ (EXC_IRQ).
- Exception taken, FIQ (EXC_FIQ).

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0082`, EXC_SVC, Exception taken, Supervisor Call

The counter counts each Supervisor Call exception counted by EXC_TAKEN that is Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0083, EXC_PABORT, Exception taken, Instruction Abort

The counter counts each Instruction Abort exception counted by EXC_TAKEN that is Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0084, EXC_DABORT, Exception taken, Data Abort or SError

The counter counts each Guarded control stack Data Check Exception, Data Abort, SError interrupt, or virtual SError interrupt exception counted by EXC_TAKEN that is Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0086, EXC_IRQ, Exception taken, IRQ

The counter counts each IRQ or virtual IRQ exception counted by EXC_TAKEN that is Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0087, EXC_FIQ, Exception taken, FIQ

The counter counts each FIQ or virtual FIQ exception counted by EXC_TAKEN that is Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0088, EXC_SMC, Exception taken, Secure Monitor Call

The counter counts each Secure Monitor Call exception counted by EXC_TAKEN.

The counter does not count SMC instructions that generate other exceptions, including Trap exceptions and Undefined Instruction exceptions.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008A, EXC_HVC, Exception taken, Hypervisor Call

The counter counts each Hypervisor Call exception counted by EXC_TAKEN.

The counter does not count HVC instructions that are UNDEFINED.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008B, EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally

The counter counts each Instruction Abort exception counted by EXC_TAKEN that is not Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008C, EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally

The counter counts each Guarded control stack Data Check Exception, Data Abort, SError interrupt, or virtual SError interrupt exception counted by EXC_TAKEN, that is not Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x008D, EXC_TRAP_OTHER, Exception taken, other traps not Taken locally

The counter counts each exception counted by EXC_TAKEN that is not Taken locally and not counted as any of the following:

- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort not Taken locally (EXC_TRAP_PABORT).
- Exception taken, Data Abort or SError not Taken locally (EXC_TRAP_DABORT).
- Exception taken, IRQ not Taken locally (EXC_TRAP_IRQ).
- Exception taken, FIQ not Taken locally (EXC_TRAP_FIQ).

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008E, EXC_TRAP_IRQ, Exception taken, IRQ not Taken locally

The counter counts each IRQ or virtual IRQ exception counted by EXC_TAKEN that is not Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008F, EXC_TRAP_FIQ, Exception taken, FIQ not Taken locally

The counter counts each FIQ or virtual FIQ exception counted by EXC_TAKEN that is not Taken locally.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x400C, TRB_WRAP, Trace buffer current write pointer wrapped

The counter counts each write made by the Trace Buffer Unit to the trace buffer that causes the trace buffer current write pointer to wrap to the trace buffer base pointer.

PMCEID0_EL0[44] reads as 1 if this event is implemented and 0 otherwise.

0x400D, PMU_OVFS, PMU overflow, counters accessible to EL1 and EL0

The counter counts each unsigned overflow of a PMU counter due to any of the following:

- An event is counted by an event counter <n> and all of the following are true:
  - PMINTENSET_EL1[n] is 1.
  - One of the following is true:
    - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[31:0], and either FEAT_PMUv3p5 is not implemented or PMCR_EL0.LP is 0.
    - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[63:0], FEAT_PMUv3p5 is implemented, and PMCR_EL0.LP is 1.
  - The event counter <n> is in the first range.
- A cycle is counted by PMCCNTR_EL0, PMINTENSET_EL1[31] is 1, and one of the following is true:
  - Counting the cycle causes unsigned overflow of PMCCNTR_EL0[31:0] and PMCR_EL0.LC is 0.
  - Counting the cycle causes unsigned overflow of PMCCNTR_EL0[63:0] and PMCR_EL0.LC is 1.
- FEAT_PMUv3_ICNTR is implemented, an instruction is counted by PMICNTR_EL0, PMINTENSET_EL1.F0 is 1, and counting the instruction causes unsigned overflow of PMICNTR_EL0[63:0].

This event cannot be counted by the PMU. PMCEID0_EL0[45] reads as 0.

0x400E, TRB_TRIG, Trace buffer Trigger Event

The counter counts each trace buffer Trigger Event.

It is IMPLEMENTATION DEFINED whether this event can be counted by the PMU.

PMCEID0_EL0[46] reads as 1 if this event is implemented and can be counted by the PMU, and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x400F, PMU_HOVFS, PMU overflow, counters reserved for use by EL2

The counter counts each unsigned overflow of a PMU counter due to an event counted by an event counter <n> when all of the following are true:

- EL2 is implemented.
- PMINTENSET_EL1[n] is 1.
- One of the following is true:
  - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[31:0], and either FEAT_PMUv3p5 is not implemented or MDCR_EL2.HLP is 0.
  - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[63:0], FEAT_PMUv3p5 is implemented, and MDCR_EL2.HLP is 1.
- The event counter <n> is in the second range.

The event is not transmitted to a trace unit if SelfHostedTraceEnabled() is TRUE and TRFCR_EL2.E2TRE is 0.

Note

This is in addition to the rules for the export of all events to a trace unit. See Controls to prohibit trace at Exception levels.

This event cannot be counted by the PMU. PMCEID0_EL0[47] reads as 0.

0x8000, SIMD_INST_RETIRED, Instruction architecturally executed, SIMD

The counter counts each architecturally executed SIMD instruction.

It is IMPLEMENTATION DEFINED which architecturally executed SIMD instructions are counted in AArch32 state.

When Armv9.5 is not implemented, it is IMPLEMENTATION DEFINED whether scalar loads and stores to SIMD&FP registers other than those listed above are counted.

When Armv9.5 is implemented, other scalar loads and stores to SIMD&FP registers are not counted.

0x8001, ASE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD

The counter counts each architecturally executed instruction counted by ASE_SVE_INST_RETIRED that is classified as an Advanced SIMD instruction.

The following instructions are counted as Advanced SIMD instructions:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - If FEAT_Crypto is implemented, the Advanced SIMD Cryptographic instructions. See The Cryptographic Extension in AArch64 state.
  - Advanced SIMD loads and stores.
- For AArch32 state:
  - Instructions from Advanced SIMD data-processing instructions.
  - If FEAT_Crypto is implemented, the `VMULL` (P64 variant) instruction. See The Cryptographic Extension in AArch32 state.

This includes all instructions that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing instructions.
- Scalar floating-point data-processing instructions.

If FEAT_PMUv3p9 is implemented, Advanced SIMD scalar instructions are not counted as Advanced SIMD instructions. Otherwise, it is IMPLEMENTATION DEFINED whether Advanced SIMD scalar instructions are counted as Advanced SIMD instructions. This includes Advanced SIMD scalar instructions which operate on floating-point values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8002, SVE_INST_RETIRED, Instruction architecturally executed, SVE

The counter counts each architecturally executed instruction counted by ASE_SVE_INST_RETIRED that is classified as an SVE instruction.

An SVE instruction is any instruction that operates on the SVE scalable vector and predicate registers that is not any of the following:

- When FEAT_SME is implemented, an instruction counted by SME_SPEC as an SME instruction.
- If FEAT_PMUv3p9 is implemented, a non-SIMD SVE instruction.

If FEAT_PMUv3p9 is not implemented, it is IMPLEMENTATION DEFINED whether the counter counts non-SIMD instructions.

Instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE instructions.

0x8003, ASE_SVE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD or SVE

The counter counts each architecturally executed instruction counted by SE_INST_RETIRED that is classified as an Advanced SIMD instruction or an SVE instruction.

See ASE_INST_RETIRED and SVE_INST_RETIRED for these classifications.

0x8107, BR_SKIP_RETIRED, Branch instruction architecturally executed, not taken

The counter counts each conditional Software change of the PC instruction that is not taken.

This includes all:

- Conditional branch instructions.
- Conditional memory-reading instructions that explicitly write to the PC.
- Conditional data-processing instructions that explicitly write to the PC.
- Conditional exception return instructions.

These are the same instructions which, if unconditional, or conditional and taken, are counted by the PC_WRITE_RETIRED event.

If FEAT_PMUv3p9 is implemented, then the following instructions are not included as a Software change of the PC:

- A conditional UNDEFINED instruction that does not generate an exception.
- A conditional exception-generating instruction, `SVC`, `HVC`, or `SMC`, that does not generate an exception.
- A conditional Context synchronization barrier instruction, `ISB`, that fails its condition code check.

If FEAT_PMUv3p9 is not implemented, then it is IMPLEMENTATION DEFINED whether these instructions are included as a Software change of the PC.

Note

Many of these instructions can only be conditional in the AArch32 instruction sets.

The counter does not increment for exceptions other than those explicitly identified in these lists.

If PC_WRITE_RETIRED and BR_SKIP_RETIRED are both implemented, the PE must treat the following types of instruction in the same way for both events:

- `BRK` and `BKPT` instructions.
- UNDEFINED instructions.
- The exception-generating instructions, `SVC`, `HVC`, and `SMC`.
- Context synchronization barrier instructions.

From Armv8.6, if BR_RETIRED is also implemented, the PE must also treat these types of instruction in the same way for the BR_RETIRED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8108, BR_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, immediate, taken

The counter counts each instruction counted by both BR_IMMED_RETIRED and PC_WRITE_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is taken.

0x8109, BR_IMMED_SKIP_RETIRED, Branch instruction architecturally executed, immediate, not taken

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_SKIP_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is not taken.

0x810A, BR_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect, taken

The counter counts each instruction counted by both BR_IND_RETIRED and PC_WRITE_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is taken.

0x810B, BR_IND_SKIP_RETIRED, Branch instruction architecturally executed, indirect, not taken

The counter counts each instruction counted by both BR_IND_RETIRED and BR_SKIP_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is not taken.

0x810C, BR_INDNR_TAKEN_RETIRED, Branch instruction architecturally executed, indirect excluding procedure return, taken

The counter counts each instruction counted by both BR_IND_RETIRED and PC_WRITE_RETIRED, that is not counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is taken.

0x810D, BR_INDNR_SKIP_RETIRED, Branch instruction architecturally executed, indirect excluding procedure return, not taken

The counter counts each instruction counted by both BR_INDNR_RETIRED and BR_SKIP_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is not taken.

0x810E, BR_RETURN_ANY_RETIRED, Branch instruction architecturally executed, procedure return

The counter counts each instruction counted by BR_IND_RETIRED where if taken, the branch would be counted by BR_RETURN_RETIRED.

These are all return instructions on the architecturally executed path.

0x810F, BR_RETURN_SKIP_RETIRED, Branch instruction architecturally executed, procedure return, not taken

The counter counts each instruction counted by both BR_RETURN_ANY_RETIRED and BR_SKIP_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is not taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x811D, BR_IND_RETIRED, Instruction architecturally executed, indirect branch

The counter counts each architecturally executed indirect branch instruction.

Indirect branch instructions are Software change of the PC instructions other than exception-generating instructions and immediate branch instructions. Software change of the PC has the same definition as for the PC_WRITE_RETIRED and BR_SKIP_RETIRED events. Immediate branch instructions are defined by the BR_IMMED_RETIRED event.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

0x811E, BR_INDNR_RETIRED, Branch instruction architecturally executed, indirect excluding procedure return

The counter counts each instruction counted by BR_IND_RETIRED that is not counted by BR_RETURN_ANY_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path.

0x811F, BRB_FILTRATE, Branch Record captured

The counter counts each valid Branch record captured in the branch record buffer.

Branch records that are not captured because they are removed by filtering are not counted.

When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event.

It is CONSTRAINED UNPREDICTABLE whether the counter counts Branch records injected by a BRB INJ instruction.

If counting this event causes unsigned overflow of the event counter counting the event and this in turn causes a BRBE freeze event then:

- The Branch record for the operation that generated the event will be generated and captured in the Branch record buffer.
- It is CONSTRAINED UNPREDICTABLE whether the Branch Record Buffer Extension generates Branch records for other operations in program order after the operation that generated the event that would otherwise be generated when generation of Branch records is not Paused.

Arm recommends that implementations minimize capture of additional branches.

0x8179, BRNL_INDNR_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch without link excluding procedure return, taken

The counter counts each branch counted by BRNL_IND_TAKEN_RETIRED that is not counted by BR_RETURN_RETIRED.

These are all indirect branch without link excluding procedure return instructions on the architecturally executed path, where the branch is taken.

0x817A, BL_TAKEN_RETIRED, Branch instruction architecturally executed, branch with link, taken

The counter counts each Software change of the PC counted by PC_WRITE_RETIRED that calls a subroutine and sets LR to return address.

These are all branch with link instructions on the architecturally executed path, where the branch is taken.

0x817B, BRNL_TAKEN_RETIRED, Branch instruction architecturally executed, branch without link, taken

The counter counts each Software change of the PC counted by PC_WRITE_RETIRED that is not counted by BL_TAKEN_RETIRED.

These are all branch without link instructions on the architecturally executed path, where the branch is taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x817C, BL_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch with link, taken

The counter counts each branch counted by both BL_TAKEN_RETIRED and BR_IND_TAKEN_RETIRED.

These are all indirect branch with link instructions on the architecturally executed path, where the branch is taken.

0x817D, BRNL_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch without link, taken

The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IND_TAKEN_RETIRED.

These are all indirect branch without link instructions on the architecturally executed path, where the branch is taken.

0x817E, BL_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, direct branch with link, taken

The counter counts each branch counted by both BL_TAKEN_RETIRED and BR_IMMED_TAKEN_RETIRED.

These are all direct branch with link instructions on the architecturally executed path, where the branch is taken.

0x817F, BRNL_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, direct branch without link, taken

The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IMMED_TAKEN_RETIRED.

These are all direct branch without link instructions on the architecturally executed path, where the branch is taken.

0x8180, BR_UNCOND_RETIRED, Branch instruction architecturally executed, unconditional branch

The counter counts each Software change of the PC counted by BR_RETIRED that is not counted by BR_COND_RETIRED.

These are all unconditional branch instructions on the architecturally executed path.

0x8181, BR_COND_RETIRED, Branch instruction architecturally executed, conditional branch

The counter counts each Software change of the PC counted by BR_RETIRED that is a conditional branch.

These are all conditional branch instructions on the architecturally executed path.

In the A64 instruction set, conditional branches are:

- `B.cond`, `CBNZ`, `CBZ`, `TBNZ`, and `TBZ`.
- If FEAT_HBC is implemented, `BC.cond`.
- If FEAT_CMPBR is implemented, `CB<cc>`, `CBB<cc>`, and `CBH<cc>`.

This includes conditional branch instructions with the `AL` or `NV` condition code.

In the A32 instruction set, conditional branches are Software change of the PC instructions that have bits [31:29] of the instruction opcode not equal to `0b111`.

In the T32 instruction set, conditional branches are `B<c>`, `B<c>.W`, Software change of the PC instructions executed in an IT block, `CBNZ`, and `CBZ`. Branch instruction encodings, including `B{<c>}` and `B{<c>}.W`, where `<c>` is either omitted or `AL` are only considered conditional branches when they are the last instruction in an IT block.

0x8182, BR_COND_TAKEN_RETIRED, Branch instruction architecturally executed, conditional branch, taken

The counter counts each Software change of the PC counted by both BR_COND_RETIRED and PC_WRITE_RETIRED.

These are all conditional branch instructions on the architecturally executed path, where the branch is taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8183, BR_HINT_COND_RETIRED, Branch instruction architecturally executed, hinted conditional

The counter counts each branch counted by BR_COND_RETIRED that is a hinted conditional branch.

These are all hinted conditional branch instructions on the architecturally executed path.

The hinted conditional branch instruction is `BC.cond`.

0x82A4, ASE_SVE_RETIRED, Instruction architecturally executed, Advanced SIMD data processing or SVE data processing

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an Advanced SIMD data-processing instruction or an SVE data-processing instruction.

See ASE_RETIRED and SVE_RETIRED for these classifications.

0x82A8, LD_ANY_RETIRED, Instruction architecturally executed, load

The counter counts each architecturally executed instruction counted by LDST_ANY_RETIRED that is classified as a Memory-reading instruction.

It is IMPLEMENTATION DEFINED whether the prefetch instructions counted by PRF_RETIRED are counted as Memory-reading instructions:

- If the prefetch instructions are counted as Memory-reading instructions, then they are counted by LD_RETIRED and LD_ANY_RETIRED.
- Otherwise, if the prefetch instructions are not counted as Memory-reading instructions by LD_ANY_RETIRED, then it is further IMPLEMENTATION DEFINED which one of the following applies:
  - They are counted as data-processing instructions.
  - They are counted by LDST_ANY_RETIRED.

Arm recommends that if a prefetch instruction is not implemented as a NOP and the PRF_RETIRED event is not implemented, then the instruction is counted as a Memory-reading instruction.

0x82A9, ST_ANY_RETIRED, Instruction architecturally executed, store

The counter counts each architecturally executed instruction counted by LDST_ANY_RETIRED that is classified as a Memory-writing instruction.

0x82AA, LDST_ANY_RETIRED, Instruction architecturally executed, load or store

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as a Memory-reading instruction or a Memory-writing instruction.

See LD_ANY_RETIRED and ST_ANY_RETIRED for these classifications.

If the PRF_RETIRED event is implemented and the prefetch instructions counted by PRF_RETIRED are not counted as Memory-reading instructions by LD_RETIRED and LD_ANY_RETIRED, then it is IMPLEMENTATION DEFINED whether these instructions are counted by LDST_ANY_RETIRED. Arm recommends that these instructions are counted by LDST_ANY_RETIRED.

0x82AB, DP_RETIRED, Instruction architecturally executed, integer data processing

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an integer data-processing instruction.

An integer data-processing instruction is any instruction that is not counted as any of the following:

- A load or store instruction, counted by LD_RETIRED or ST_RETIRED.
- A Software change of PC instruction, counted by PC_WRITE_RETIRED.
- A scalar floating-point data processing instruction, counted by VFP_RETIRED.
- An Advanced SIMD, SVE or SME data processing instruction, counted by SE_RETIRED.
- A cryptographic instruction, counted by CRYPTO_RETIRED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

That is, the following instructions are counted as integer data-processing instructions:

- In AArch64 state:
  - Instructions from Data processing - immediate.
  - Instructions from Data processing - register.
  - Instructions from System register instructions.
  - Instructions from 'Instructions with register argument'.
  - Instructions from System instructions other than Memory-writing instructions.
  - Instructions from Hint instructions.
  - Instructions from Floating-point move (register) that transfer data between a general-purpose register and a SIMD&FP register without conversion: `FMOV` (general).
  - Instructions from SIMD move that transfer data between a general-purpose register and an element or elements in a SIMD&FP register: `DUP` (general), `SMOV`, `UMOV`, and `INS` (general). This includes the aliases `MOV` (from general) and `MOV` (to general).
  - When FEAT_SVE is implemented, non-SIMD SVE instructions.
- In AArch32 state:
  - Instructions from Data-processing instructions.
  - Instructions from PSTATE and banked register access instructions.
  - Instructions from Banked register access instructions.
  - Instructions from Miscellaneous instructions other than `ISB` and prefetches.
  - Instructions from System register access instructions other than `LDC` and `STC` instructions.
  - `VDUP` (general-purpose register).
  - All `VMOV` instructions that transfer data between a general-purpose register and a SIMD&FP register.
  - `VMRS` and `VMSR`.

`0x82AC`, ASE_RETIRED, Instruction architecturally executed, Advanced SIMD data processing

The counter counts each architecturally executed instruction counted by ASE_SVE_RETIRED that is classified as an Advanced SIMD data-processing instruction.

The following instructions are counted as Advanced SIMD data-processing instructions:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - Advanced SIMD scalar instructions, including those which operate on floating-point values.
  - If FEAT_Crypto is implemented, the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions. See The Cryptographic Extension in AArch64 state.
- For AArch32 state:
  - Instructions from Advanced SIMD data-processing instructions.
  - If FEAT_Crypto is implemented, the `VMULL` (P64 variant) instruction. See The Cryptographic Extension in AArch32 state.

This includes all instructions that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing instructions.
- Scalar floating-point data-processing instructions.
- Load or store instructions.
- Cryptographic data-processing instructions counted by CRYPTO_RETIRED. CRYPTO_RETIRED does not count the Cryptographic data-processing instructions included above.

`0x82AD`, VFP_RETIRED, Instruction architecturally executed, scalar floating-point data processing

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as a scalar floating-point instruction.

The following instructions are counted as scalar floating-point data-processing instructions:

- In AArch64 state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - The scalar floating-point instructions from Data processing - SIMD and floating-point. Only the scalar instructions are counted. SIMD floating-point instructions are not counted.
    - Instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.
  - In AArch32 state:
    - Instructions from Floating-point data-processing instructions.
    - Instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `VCVT<mode>` (floating-point), `VCVT`, `VCVTT`, and `VCVTB`.

This includes all scalar instructions that operate on the SIMD&FP registers as floating-point values, except those that are counted as one of the following:

- SIMD scalar instructions.
- Integer data-processing instructions.
- Load or store instructions.

`0x82AE`, SVE_RETIRED, Instruction architecturally executed, SVE data processing

The counter counts each architecturally executed instruction counted by ASE_SVE_RETIRED that is classified as an SVE data-processing instruction.

An SVE data-processing instruction is any instruction that operates on the SVE scalable vector and predicate registers that is not any of the following:

- A load or store instruction. These are counted by SVE_INST_RETIRED.
- If the Cryptographic Extension is implemented, an instruction counted by CRYPTO_RETIRED as a Cryptographic data-processing instruction.
- When FEAT_SME is implemented, an instruction counted by SME_SPEC as an SME data-processing instruction.
- A non-SIMD SVE instruction.

If the Cryptographic Extension and FEAT_SVE2 are implemented, the SVE `PMULLB` and `PMULLT` (Q variants) instructions are not counted by CRYPTO_SPEC and are counted as SVE data-processing instructions.

Data-processing instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE data-processing instructions.

`0x82AF`, CRYPTO_RETIRED, Instruction architecturally executed, cryptographic data processing

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as a cryptographic instruction.

The following instructions are counted as cryptographic data-processing instructions:

- In AArch64 state, instructions from the following sections:
  - The Cryptographic Extension in AArch64 state other than the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions.
  - If FEAT_SVE2 is implemented, SVE2 Crypto Extensions other than the SVE `PMULLB` and `PMULLT` (Q variants) instructions.
- In AArch32 state, instructions from The Cryptographic Extension in AArch32 state other than `VMULL` (P64 variant).

`0x82B3`, PRF_RETIRED, Instruction architecturally executed, prefetch

The counter counts each architecturally executed prefetch instruction counted by INST_RETIRED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8358`, SME_RETIRED, Instruction architecturally executed, SME data processing

The counter counts each architecturally executed instruction counted by SE_RETIRED that is classified as an SME data-processing instruction.

An SME data-processing instruction is any instruction that operates on the ZA or ZT registers that is not any of the following:

- A load or store instruction. These are counted by SME_INST_RETIRED.
- A non-SIMD SME instruction.

Data-processing instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE data-processing instructions.

`0x8359`, SE_RETIRED, Instruction architecturally executed, Advanced SIMD, SVE, or SME data processing

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an Advanced SIMD, SVE, or SME data-processing instruction.

See ASE_SVE_RETIRED and SME_RETIRED for these classifications.

`0x835A`, SME_INST_RETIRED, Instruction architecturally executed, SME

The counter counts each architecturally executed instruction counted by SE_INST_RETIRED that is classified as an SME instruction.

An SME instruction is any instruction that operates on the ZA or ZT registers that is not any of the following:

- A non-SIMD SME instruction.

Instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE instructions.

`0x835B`, SE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD, SVE, or SME

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an Advanced SIMD, SVE, or SME instruction.

See ASE_SVE_INST_RETIRED and SME_INST_RETIRED for these classifications.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.3.2 Common microarchitectural events

The Common microarchitectural events are:

0x0001, L1I_CACHE_REFILL, Level 1 instruction cache refill

The counter counts each access counted by L1I_CACHE that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[1] reads as 1 if this event is implemented and 0 otherwise.

0x0002, L1I_TLB_REFILL, Level 1 instruction TLB refill

The counter counts each Instruction memory access counted by L1I_TLB that causes a TLB refill of the Level 1 instruction TLB.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[2] reads as 1 if this event is implemented and 0 otherwise.

0x0003, L1D_CACHE_REFILL, Level 1 data cache refill

The counter counts each access counted by L1D_CACHE that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 data cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[3] reads as 1 if this event is implemented and 0 otherwise.

`0x0004`, L1D_CACHE, Level 1 data cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 1 data or unified cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as `LDM` or `STM`. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L1D_CACHE_RW event is implemented:

- If the L1D_CACHE_PRFM event is implemented, accesses to the Level 1 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the L1D_CACHE_HWPRF event is implemented, accesses to the Level 1 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the L1D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 1 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

When FEAT_GCS is implemented:

- If the L1GCS_CACHE event is implemented, GCS data read operations and GCS data write operations are not counted.
- Otherwise, GCS data read operations and data write operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[4] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0005, L1D_TLB_REFILL, Level 1 data TLB refill

The counter counts each access counted by L1D_TLB that causes a TLB refill of the Level 1 data or unified TLB.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[5] reads as 1 if this event is implemented and 0 otherwise.

0x0010, BR_MIS_PRED, Branch instruction speculatively executed, mispredicted or not predicted

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

If no program-flow prediction resources are implemented, Arm recommends that the counter counts all branches that are not taken.

PMCEID0_EL0[16] reads as 1 if this event is implemented and 0 otherwise.

0x0011, CPU_CYCLES, Cycle

The counter increments on every cycle.

All counters are subject to changes in clock frequency, including when a `WFI` or `WFE` instruction stops the clock. If FEAT_PMUv3p9 is implemented, then CPU_CYCLES does not increment when the clocks are stopped by `WFI` and `WFE` instructions. Otherwise, it is CONSTRAINED UNPREDICTABLE whether or not CPU_CYCLES continues to increment when the clocks are stopped by `WFI` and `WFE` instructions.

Note

Unlike PMCCNTR or PMCCNTR_EL0, this count is not affected by the cycle counter controls:

- The counter is not incremented in prohibited regions, so is not affected by PMCR.DP or PMCR_EL0.DP.
- The counter increments on every cycle, regardless of the value of PMCR.D or PMCR_EL0.D.
- The counter is reset when event counters are reset by PMCR.P and PMCR_EL0.P, never by PMCR.C or PMCR_EL0.C.
- If FEAT_PMUv3p5 is implemented, the counter ignores SDCR.SCCD, MDCR_EL3.SCCD, HDCR.HCCD, and MDCR_EL2.HCCD.

In a multithreaded implementation:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of PMEVTYPER<n>_EL0.MT for the event counter is 0, then the counter counts each Processor cycle when the counting PE is active.
- If the Effective value of PMEVTYPER<n>_EL0.MT for the event counter is 1, then the counter counts each Processor cycle.

PMCEID0_EL0[17] reads as 1 if this event is implemented and 0 otherwise.

`0x0012`, BR_PRED, Predictable branch instruction speculatively executed

The counter counts each branch or other change in the program flow that the branch prediction resources are capable of predicting.

If all branches are subject to prediction, for example a BTB or BTAC, then all branches are predictable branches.

If branches are decoded before the predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. Arm recommends that all branches are counted.

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by BR_MIS_PRED.

If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that BR_PRED counts all branches.

It is IMPLEMENTATION DEFINED when the branch is counted. Arm recommends that it is counted when the branch is resolved, that is, at the same point in the instruction pipeline as when the BR_MIS_PRED event would be counted if the branch resolves as mispredicted. This means that (BR_PRED - BR_MIS_PRED) is the number of correctly predicted branches and the ratio (BR_MIS_PRED ÷ BR_PRED) can be calculated in a meaningful way.

PMCEID0_EL0[18] reads as 1 if this event is implemented and 0 otherwise.

`0x0013`, MEM_ACCESS, Data memory access

The counter counts each Memory-read operation or Memory-write operation that the PE makes.

The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these.

The number of accesses generated by each instruction is IMPLEMENTATION DEFINED.

The counter does not count:

- Instruction memory accesses.
- Translation table walks.
- Write-back from any cache.
- Refilling of any cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

PMCEID0_EL0[19] reads as 1 if this event is implemented and 0 otherwise.

`0x0014`, L1I_CACHE, Level 1 instruction cache access

The counter counts each Instruction memory access to at least the Level 1 instruction or unified cache.

Each Instruction memory access to other Level 1 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L1I_CACHE_RD event is implemented:

- If the L1I_CACHE_PRFM event is implemented, accesses to the Level 1 instruction cache due to a prefetch instruction are counted. Otherwise, these are not counted.
- If the L1I_CACHE_HWPRF event is implemented, accesses to the Level 1 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L1I_CACHE_RD event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 1 instruction cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[20] reads as 1 if this event is implemented and 0 otherwise.

`0x0015`, L1D_CACHE_WB, Level 1 data cache write-back

The counter counts each write-back of data from the Level 1 data or unified cache to outside of the Level 1 caches.

For example:

- A write-back of a cache line to a Level 2 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 1 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 1 data cache to outside of Level 1 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 caches or memory.
- Writes that write through the Level 1 data cache to outside of the Level 1 caches.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

PMCEID0_EL0[21] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0016, L2D_CACHE, Level 2 data cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 2 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 2 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L2D_CACHE_RW event is implemented:

- If the L2D_CACHE_PRFM event is implemented, accesses to the Level 2 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the L2D_CACHE_HWPRF event is implemented, accesses to the Level 2 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the L2D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[22] reads as 1 if this event is implemented and 0 otherwise.

0x0017, L2D_CACHE_REFILL, Level 2 data cache refill

The counter counts each access counted by L2D_CACHE that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 data cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of and write-backs from any Level 1 data, instruction, or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 data cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 data cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[23] reads as 1 if this event is implemented and 0 otherwise.

`0x0018`, L2D_CACHE_WB, Level 2 data cache write-back

The counter counts each write-back of data from the Level 2 data or unified cache to outside of the Level 1 and Level 2 caches.

For example:

- A write-back of a cache line to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 2 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 2 data cache to outside of Level 1 and Level 2 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 and Level 2 caches or memory.
- Writes that write through the Level 2 data cache to outside of the Level 1 and Level 2 caches.
- Transfers of data from the Level 2 data cache to a Level 1 cache to satisfy a refill of the other cache.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

PMCEID0_EL0[24] reads as 1 if this event is implemented and 0 otherwise.

`0x0019`, BUS_ACCESS, Bus access

The counter counts each Memory-read operation or Memory-write operation that accesses outside of the boundary of the PE and its closely-coupled caches.

Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of accesses across all buses.

A bus access is part of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The BUS_ACCESS event counts each beat of each transaction. That is, each bus cycle counted by BUS_CYCLES for which the bus is active.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An Unattributable bus transaction occurs when a requester outside the PE makes a request that results in a bus access, for example, a coherency request.

If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

PMCEID0_EL0[25] reads as 1 if this event is implemented and 0 otherwise.

`0x001A`, MEMORY_ERROR, Local memory error

The counter counts each occurrence of a memory error signaled by a memory closely coupled to this PE.

The definition of local memories is IMPLEMENTATION DEFINED but includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors as defined in the architecture, such as MMU faults.

PMCEID0_EL0[26] reads as 1 if this event is implemented and 0 otherwise.

`0x001B`, INST_SPEC, Operation speculatively executed

The counter counts each speculatively executed instruction.

This includes instructions that are subsequently not architecturally executed.

PMCEID0_EL0[27] reads as 1 if this event is implemented and 0 otherwise.

`0x001D`, BUS_CYCLES, Bus cycle

The counter increments on every cycle of the interface at the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

Note

If the implementation clocks the external memory interface at the same rate as the processor hardware, then the counter counts every cycle.

PMCEID0_EL0[29] reads as 1 if this event is implemented and 0 otherwise.

`0x001F`, L1D_CACHE_ALLOCATE, Level 1 data cache allocation without refill

The counter counts each Memory-write operation that writes an entire line into the Level 1 data or unified cache without fetching data from outside the Level 1 data or unified cache.

These are allocations of cache lines in the Level 1 data or unified cache that are not refills counted by L1D_CACHE_REFILL. For example:

- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, other PEs in the multithreaded implementation.

PMCEID0_EL0[31] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0020, L2D_CACHE_ALLOCATE, Level 2 data cache allocation without refill

The counter counts each Memory-write operation that writes an entire line into the Level 2 data or unified cache without fetching data from outside the Level 2 data or unified cache.

These are allocations of cache lines in the Level 2 data or unified cache that are not refills counted by L2D_CACHE_REFILL. For example:

- A write-back of an entire cache line from a Level 1 cache to the Level 2 data cache.
- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, other PEs in the multithreaded implementation.

PMCEID1_EL0[0] reads as 1 if this event is implemented and 0 otherwise.

0x0022, BR_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted

The counter counts each branch instruction counted by BR_RETIRED that was not correctly predicted.

If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

PMCEID1_EL0[2] reads as 1 if this event is implemented and 0 otherwise.

0x0023, STALL_FRONTEND, No operation sent for execution due to the frontend

The counter counts each cycle counted by CPU_CYCLES where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend.

The division between frontend and backend is IMPLEMENTATION DEFINED. All STALL events are counted at the same point in the pipeline.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts only if the stall condition is true for all the PEs in the multithreaded processor.

Note

- For a simplified pipeline model of Fetch-Decode-Issue-Execute-Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.
- In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

PMCEID1_EL0[3] reads as 1 if this event is implemented and 0 otherwise.

0x0024, STALL_BACKEND, No operation sent for execution due to the backend

The counter counts each cycle counted by CPU_CYCLES where Attributable instructions or operations for the PE are available to dispatch from the frontend to the backend, but no Attributable instruction or operation is sent for execution because the backend is unable to accept any of the instructions or operations available for the PE.

For example, the backend might be unable to accept operations because of a resource conflict or non-availability.

It is IMPLEMENTATION DEFINED whether the counter also counts each cycle counted by CPU_CYCLES where no Attributable instructions or operations for the PE are available to dispatch from the frontend and the backend is unable to accept any instructions or operations for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events are counted at the same point in the pipeline.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts only if the stall condition is true for all the PEs in the multithreaded processor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

PMCEID1_EL0[4] reads as 1 if this event is implemented and 0 otherwise.

`0x0025`, L1D_TLB, Level 1 data TLB access

The counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 1 data or unified TLB.

Each access to a TLB entry is counted including multiple accesses caused by single instructions such as LDM or STM.

When FEAT_GCS is implemented:

- If the L1GCS_TLB event is implemented, GCS data read operations and GCS data write operations that causes a TLB access to at least a Level 1 TLB are not counted.
- Otherwise, GCS data read operations and data write operations that causes a TLB access to at least a Level 1 TLB are counted.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[5] reads as 1 if this event is implemented and 0 otherwise.

`0x0026`, L1I_TLB, Level 1 instruction TLB access

The counter counts each Instruction memory access that causes a TLB access to at least the Level 1 instruction TLB.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[6] reads as 1 if this event is implemented and 0 otherwise.

`0x0027`, L2I_CACHE, Level 2 instruction cache access

The counter counts each Instruction memory access to at least the Level 2 instruction or unified cache.

Each Instruction memory access to other Level 2 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L2I_CACHE_RD event is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the L2I_CACHE_PRFM event is implemented, accesses to the Level 2 instruction cache due to a prefetch instruction are counted. Otherwise, these are not counted.
- If the L2I_CACHE_HWPRF event is implemented, accesses to the Level 2 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L2I_CACHE_RD event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 instruction cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[7] reads as 1 if this event is implemented and 0 otherwise.

`0x0028`, L2I_CACHE_REFILL, Level 2 instruction cache refill

The counter counts each access counted by L2I_CACHE that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of any Level 1 instruction or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 instruction cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 instruction cache.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[8] reads as 1 if this event is implemented and 0 otherwise.

`0x0029`, L3D_CACHE_ALLOCATE, Level 3 data cache allocation without refill

The counter counts each Memory-write operation that writes an entire line into the Level 3 data or unified cache without fetching data from outside the Level 3 data or unified cache.

These are allocations of cache lines in the Level 3 data or unified cache that are not refills counted by L3D_CACHE_REFILL. For example:

- A write-back of an entire cache line from a Level 2 cache to the Level 3 data cache.
- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, other PEs in the multithreaded implementation.

PMCEID1_EL0[9] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x002A, L3D_CACHE_REFILL, Level 3 data cache refill

The counter counts each access counted by L3D_CACHE that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Level 3 caches, even if the data is ultimately not allocated into the Level 3 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 3 data cache that cause a refill that is satisfied by fetching data from memory, a lower level cache, or a Level 3 cache of another PE.
- Refills of and write-backs from any Level 1 or Level 2 data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Level 3 caches of this PE.
- Accesses to the Level 3 data cache that cause a refill of a Level 2 cache of this PE from outside of the Level 1 to Level 3 caches of this PE, even if there is no refill of the Level 3 data cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[10] reads as 1 if this event is implemented and 0 otherwise.

0x002B, L3D_CACHE, Level 3 data cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 3 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 3 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L3D_CACHE_RW event is implemented:

- If the L3D_CACHE_PRFM event is implemented, accesses to the Level 3 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the L3D_CACHE_HWPRF event is implemented, accesses to the Level 3 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the L3D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 3 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[11] reads as 1 if this event is implemented and 0 otherwise.

`0x002C`, L3D_CACHE_WB, Level 3 data cache write-back

The counter counts each write-back of data from the Level 3 data or unified cache to outside of the Level 1 to Level 3 caches.

For example:

- A write-back of a cache line to a lower level cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 3 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 3 data cache to outside of Level 1 to Level 3 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Level 3 caches or memory.
- Writes that write through the Level 3 data cache to outside of the Level 1 to Level 3 caches.
- Transfers of data from the Level 3 data cache to a Level 1 or Level 2 cache to satisfy a refill of the other cache.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

PMCEID1_EL0[12] reads as 1 if this event is implemented and 0 otherwise.

`0x002D`, L2D_TLB_REFILL, Level 2 data TLB refill

The counter counts each access counted by L2D_TLB that causes a TLB refill of the Level 2 data or unified TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[13] reads as 1 if this event is implemented and 0 otherwise.

`0x002E`, L2I_TLB_REFILL, Level 2 instruction TLB refill

The counter counts each Instruction memory access counted by L2I_TLB that causes a TLB refill of the Level 2 instruction TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[14] reads as 1 if this event is implemented and 0 otherwise.

`0x002F`, L2D_TLB, Level 2 data TLB access

The counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 2 data or unified TLB.

Each access to a TLB entry is counted including refills of Level 1 TLBs.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[15] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0030, L2I_TLB, Level 2 instruction TLB access

The counter counts each Instruction memory access that causes a TLB access to at least the Level 2 instruction TLB.

Each access to a TLB entry is counted including refills of Level 1 TLBs.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[16] reads as 1 if this event is implemented and 0 otherwise.

0x0031, REMOTE_ACCESS, Access to a remote device

The counter counts each Memory-read operation or Memory-write operation that causes an access to a remote device.

That is, a socket that does not contain the PE.

System topology is IMPLEMENTATION DEFINED. This means that it is IMPLEMENTATION DEFINED which systems are defined as multi-socket systems, and, in systems defined as multi-socket systems, which components are defined as being in the same or different sockets. Arm recommends that devices where an access through that device incurs a significant latency penalty compared to other accesses are treated as being in a different socket.

For example, in a system comprising multiple integrated circuits in a multi-chip module, an access to a different integrated circuit in the same module might be treated as an access to another socket, even though the multi-chip module is physically connected to a single socket at the motherboard. However, in another system with many such multi-chip modules, an access to a different integrated system in the same module might be treated as an access to the same socket because an access to an integrated circuit on a different module has much higher latency.

The count includes all accesses to external memory counted by REMOTE_MEM. For more information, see REMOTE_MEM.

It is IMPLEMENTATION DEFINED whether an access that causes a snoop into a different socket but does not return data from or pass data to the remote socket is counted.

PMCEID1_EL0[17] reads as 1 if this event is implemented and 0 otherwise.

0x0032, LL_CACHE, Last level cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Last level cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Last level data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the LL_CACHE_RW event is implemented:

- If the LL_CACHE_PRFM event is implemented, accesses to the Last level cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the LL_CACHE_HWPRF event is implemented, accesses to the Last level cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the LL_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Last level cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[18] reads as 1 if this event is implemented and 0 otherwise.

`0x0033`, LL_CACHE_MISS, Last level cache miss

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is not completed by the Last level cache.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is not completed by the Last level cache.

That is, the access is one of the following:

- A Memory-read operation that does not return data from the Last level cache.
- A Memory-write operation that does not update the Last level cache.

The counter does not count operations that are completed by a cache above the Last level cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[19] reads as 1 if this event is implemented and 0 otherwise.

`0x0034`, DTLB_WALK, Data TLB access with at least one translation table walk

The counter counts each access counted by L1D_TLB that causes a refill of a data or unified TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted. If Armv8.7 is implemented, these accesses are counted.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[20] reads as 1 if this event is implemented and 0 otherwise.

`0x0035`, ITLB_WALK, Instruction TLB access with at least one translation table walk

The counter counts each access counted by L1I_TLB that causes a refill of an instruction TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted. If Armv8.7 is implemented, these accesses are counted.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[21] reads as 1 if this event is implemented and 0 otherwise.

`0x0036`, LL_CACHE_RD, Last level cache access, read

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is a Memory-read operation.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[22] reads as 1 if this event is implemented and 0 otherwise.

`0x0037`, LL_CACHE_MISS_RD, Last level cache miss, read

The counter counts each access counted by both LL_CACHE_MISS and LL_CACHE_RD.

That is, every miss of the Last level cache counted by LL_CACHE_MISS that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[23] reads as 1 if this event is implemented and 0 otherwise.

`0x0038`, REMOTE_ACCESS_RD, Access to a remote device, read

The counter counts each access counted by REMOTE_ACCESS that is a Memory-read operation.

PMCEID1_EL0[24] reads as 1 if this event is implemented and 0 otherwise.

`0x0039`, L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss

The counter counts each Memory-read operation to the Level 1 data or unified cache counted by L1D_CACHE_RD that incurs additional latency because it returns data from outside of the Level 1 data or unified cache of this PE.

The event indicates to software that the access missed in the Level 1 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in an access being made to a Level 2 cache or elsewhere beyond the Level 1 data or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[25] reads as 1 if this event is implemented and 0 otherwise.

`0x003A`, OP_RETIRED, Micro-operation architecturally executed

The counter counts each operation counted by OP_SPEC that would be executed in a Simple sequential execution of the program.

PMCEID1_EL0[26] reads as 1 if this event is implemented and 0 otherwise.

`0x003B`, OP_SPEC, Micro-operation speculatively executed

The counter counts each operation speculatively executed executed by the PE.

This includes operations that would not be executed in a Simple sequential execution of the program.

PMCEID1_EL0[27] reads as 1 if this event is implemented and 0 otherwise.

`0x003C`, STALL, No operation sent for execution

The counter counts each cycle counted by CPU_CYCLES where no operation was sent for execution.

On a multithreaded implementation:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of PMEVTYPER<n>_EL0.MT is 0, then the counter counts cycles when the PE is active and no operation for the PE is sent for execution, even if operations Attributable to other PEs in the multithreaded implementation are sent for execution in that cycle. The counter does not count cycles when the PE is not active.
- If the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts all cycles when no instructions or operations for any PE in the multithreaded implementation were sent for execution.

All STALL events are counted at the same point in the pipeline.

PMCEID1_EL0[28] reads as 1 if this event is implemented and 0 otherwise.

`0x003D`, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend

The counter counts each Slot counted by STALL_SLOT where an Attributable instruction or operation for the PE is available on the Slot to dispatch from the frontend to the backend, but is not sent for execution because the backend is unable to accept the available instruction or operation for the PE.

It is IMPLEMENTATION DEFINED whether the counter also counts each Slot counted by STALL_SLOT where no Attributable instruction or operation for the PE is available on the Slot to dispatch from the frontend and the backend is unable to accept any instruction or operation for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events are counted at the same point in the pipeline. The maximum value by which STALL_SLOT_BACKEND can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. For more information, see STALL_SLOT.

Note

In a single cycle, the sum of values counted by STALL_SLOT_BACKEND and STALL_SLOT_FRONTEND might be greater-than `slots`, if both the backend is unable to accept some operations and there are fewer-than `slots` operations available to dispatch from the frontend.

PMCEID1_EL0[29] reads as 1 if this event is implemented and 0 otherwise.

`0x003E`, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend

The counter counts each Slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend for the Slot.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events are counted at the same point in the pipeline. The maximum value by which STALL_SLOT_FRONTEND can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. For more information, see STALL_SLOT.

Note

In a single cycle, the sum of values counted by STALL_SLOT_BACKEND and STALL_SLOT_FRONTEND might be greater-than `slots`, if both the backend is unable to accept some operations and there are fewer-than `slots` operations available to dispatch from the frontend.

PMCEID1_EL0[30] reads as 1 if this event is implemented and 0 otherwise.

`0x003F`, STALL_SLOT, No operation sent for execution on a Slot

The counter increments by the number of instruction or operation Slots that were not occupied by an instruction or operation Attributable to the PE on each cycle.

All STALL events are counted at the same point in the pipeline. The maximum value by which STALL_SLOT can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. The definition of a Slot is IMPLEMENTATION DEFINED. The formula STALL_SLOT÷(CPU_CYCLES×`slots`) gives the utilization of the Slots of the processor by Attributable instruction or operations of this PE. Each Slot holds at most one instruction or operation each cycle.

On a multithreaded implementation:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of PMEVTYPER<n>_EL0.MT is 0, then the counter counts Slots occupied by an instruction or operation Attributable to other PEs of the multithreaded implementation only when the PE was active in that cycle. The counter does not count Slots on cycles when the PE was not active.
- If the Effective value of PMEVTYPER<n>_EL0.MT is 1, then, for every cycle, the counter counts all Slots not occupied by any instruction or operation for any PE of the multithreaded implementation.

PMCEID1_EL0[31] reads as 1 if this event is implemented and 0 otherwise.

`0x0040`, L1D_CACHE_RD, Level 1 data cache access, read

If the L1D_CACHE_RW event is implemented, the counter counts each access counted by L1D_CACHE_RW that is a Memory-read operation.

If the L1D_CACHE_RW event is not implemented, the counter counts each access counted by L1D_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0041`, L1D_CACHE_WR, Level 1 data cache access, write

If the L1D_CACHE_RW event is implemented, the counter counts each access counted by L1D_CACHE_RW that is a Memory-write operation.

If the L1D_CACHE_RW event is not implemented, the counter counts each access counted by L1D_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0042`, L1D_CACHE_REFILL_RD, Level 1 data cache refill, read

The counter counts each access counted by both L1D_CACHE_RD and L1D_CACHE_REFILL.

That is, every refill of the Level 1 data or unified cache counted by L1D_CACHE_REFILL that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0043`, L1D_CACHE_REFILL_WR, Level 1 data cache refill, write

The counter counts each access counted by both L1D_CACHE_REFILL and L1D_CACHE_WR.

That is, every refill of the Level 1 data or unified cache counted by L1D_CACHE_REFILL that is caused by a Memory-write operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0044`, L1D_CACHE_REFILL_INNER, Level 1 data cache refill, inner

The counter counts each access counted by L1D_CACHE_REFILL that generates a refill satisfied by transfer from another cache inside of the immediate cluster.

Note

The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0045`, L1D_CACHE_REFILL_OUTER, Level 1 data cache refill, outer

The counter counts each access counted by L1D_CACHE_REFILL that generates a refill satisfied by transfer from outside of the immediate cluster.

Note

The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0046`, L1D_CACHE_WB_VICTIM, Level 1 data cache write-back, victim

The counter counts each write-back counted by L1D_CACHE_WB that occurs because of a capacity eviction due to a line being allocated into the cache..

It is IMPLEMENTATION DEFINED whether this includes capacity evictions due to an agent other the PE counting the event. For example, a stashing request from outside of the PE, or an allocation due to another PE that shares the cache.

In all cases, the event is counted only if the eviction is also an L1D_CACHE_WB event. This means that, if the cache is shared, the eviction is counted only if it is attributable to the PE counting the event, as defined by L1D_CACHE_WB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0047, L1D_CACHE_WB_CLEAN, Level 1 data cache write-back, cleaning and coherency

The counter counts each write-back counted by L1D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction..

It is IMPLEMENTATION DEFINED whether the transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is counted when the dirty cache line is not also written back to a Level 2 cache or memory.

In all cases, the event is counted only if the write-back is also an L1D_CACHE_WB event. This means that, if the cache is shared, the write-back is counted only if it is attributable to the PE counting the event, as defined by L1D_CACHE_WB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0048, L1D_CACHE_INVAL, Level 1 data cache invalidate

The counter counts each invalidation of a cache line in the Level 1 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 data or unified cache invalidates a line in the Level 1 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 1 data cache is shared with other PEs. This event does not follow the general rule of Level 1 data cache events of only counting Attributable events.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x004C, L1D_TLB_REFILL_RD, Level 1 data TLB refill, read

The counter counts each access counted by both L1D_TLB_RD and L1D_TLB_REFILL.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x004D, L1D_TLB_REFILL_WR, Level 1 data TLB refill, write

The counter counts each access counted by both L1D_TLB_REFILL and L1D_TLB_WR.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x004E, L1D_TLB_RD, Level 1 data TLB access, read

The counter counts each access counted by L1D_TLB that is a Memory-read operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x004F, L1D_TLB_WR, Level 1 data TLB access, write

The counter counts each access counted by L1D_TLB that is a Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0050, L2D_CACHE_RD, Level 2 data cache access, read

If the L2D_CACHE_RW event is implemented, the counter counts each access counted by L2D_CACHE_RW that is a Memory-read operation.

If the L2D_CACHE_RW event is not implemented, the counter counts each access counted by L2D_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0051, L2D_CACHE_WR, Level 2 data cache access, write

If the L2D_CACHE_RW event is implemented, the counter counts each access counted by L2D_CACHE_RW that is a Memory-write operation.

If the L2D_CACHE_RW event is not implemented, the counter counts each access counted by L2D_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0052, L2D_CACHE_REFILL_RD, Level 2 data cache refill, read

The counter counts each access counted by both L2D_CACHE_RD and L2D_CACHE_REFILL.

That is, every refill of the Level 2 data or unified cache counted by L2D_CACHE_REFILL that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0053, L2D_CACHE_REFILL_WR, Level 2 data cache refill, write

The counter counts each access counted by both L2D_CACHE_REFILL and L2D_CACHE_WR.

That is, every refill of the Level 2 data or unified cache counted by L2D_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0056, L2D_CACHE_WB_VICTIM, Level 2 data cache write-back, victim

The counter counts each write-back counted by L2D_CACHE_WB that occurs because of a capacity eviction due to a line being allocated into the cache..

It is IMPLEMENTATION DEFINED whether this includes capacity evictions due to an agent other the PE counting the event. For example, a stashing request from outside of the PE, or an allocation due to another PE that shares the cache.

In all cases, the event is counted only if the eviction is also an L2D_CACHE_WB event. This means that, if the cache is shared, the eviction is counted only if it is attributable to the PE counting the event, as defined by L2D_CACHE_WB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0057, L2D_CACHE_WB_CLEAN, Level 2 data cache write-back, cleaning and coherency

The counter counts each write-back counted by L2D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction..

It is IMPLEMENTATION DEFINED whether the transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is counted when the dirty cache line is not also written back to a Level 2 cache or memory.

In all cases, the event is counted only if the write-back is also an L2D_CACHE_WB event. This means that, if the cache is shared, the write-back is counted only if it is attributable to the PE counting the event, as defined by L2D_CACHE_WB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0058, L2D_CACHE_INVAL, Level 2 data cache invalidate

The counter counts each invalidation of a cache line in the Level 2 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 2 data or unified cache invalidates a line in the Level 2 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 2 data cache is shared with other PEs. This event does not follow the general rule of Level 2 data cache events of only counting Attributable events.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x005C, L2D_TLB_REFILL_RD, Level 2 data TLB refill, read

The counter counts each access counted by both L2D_TLB_RD and L2D_TLB_REFILL.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x005D, L2D_TLB_REFILL_WR, Level 2 data TLB refill, write

The counter counts each access counted by both L2D_TLB_REFILL and L2D_TLB_WR.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x005E, L2D_TLB_RD, Level 2 data TLB access, read

The counter counts each access counted by L2D_TLB that is a Memory-read operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x005F, L2D_TLB_WR, Level 2 data TLB access, write

The counter counts each access counted by L2D_TLB that is a Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0060, BUS_ACCESS_RD, Bus access, read

The counter counts each access counted by BUS_ACCESS that is a Memory-read operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0061, BUS_ACCESS_WR, Bus access, write

The counter counts each access counted by BUS_ACCESS that is a Memory-write operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0062, BUS_ACCESS_SHARED, Bus access, Normal, Cacheable, Shareable

The counter counts each access counted by BUS_ACCESS that is Normal, Cacheable, Shareable.

Note

It is IMPLEMENTATION DEFINED how the PE translates the attributes from the translation table entry for a region to the attributes on the bus.
In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.
If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0063, BUS_ACCESS_NOT_SHARED, Bus access, not Normal, Cacheable, Shareable

The counter counts each access counted by BUS_ACCESS that is not counted by BUS_ACCESS_SHARED.

For example, the counter counts accesses marked as:

- Normal, Cacheable, Non-shareable.
- Normal, Non-cacheable.
- Device.

Note

It is IMPLEMENTATION DEFINED, how the PE translates the attributes from the translation table entries for a region to the attributes on the bus.
In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.
If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0064, BUS_ACCESS_NORMAL, Bus access, normal

The counter counts each access counted by BUS_ACCESS that is to Normal or bulk memory.

For example, the counter counts Normal, Cacheable and Normal, Non-cacheable accesses but does not count Device accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0065, BUS_ACCESS_PERIPH, Bus access, peripheral

The counter counts each access counted by BUS_ACCESS that is not counted by BUS_ACCESS_NORMAL.

For example, the counter counts accesses to Device memory.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0066, MEM_ACCESS_RD, Data memory access, read

If the MEM_ACCESS_RW event is implemented, the counter counts each access counted by MEM_ACCESS_RW that is a Memory-read operation.

If the MEM_ACCESS_RW event is not implemented, the counter counts each access counted by MEM_ACCESS that is a Memory-read operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x0067, MEM_ACCESS_WR, Data memory access, write

If the MEM_ACCESS_RW event is implemented, the counter counts each access counted by MEM_ACCESS_RW that is a Memory-write operation.

If the MEM_ACCESS_RW event is not implemented, the counter counts each access counted by MEM_ACCESS that is a Memory-write operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0068, UNALIGNED_LD_SPEC, Unaligned access, read

The counter counts each unaligned access counted by UNALIGNED_LDST_SPEC that is a Memory-read operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0069, UNALIGNED_ST_SPEC, Unaligned access, write

The counter counts each unaligned access counted by UNALIGNED_LDST_SPEC that is a Memory-write operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006A, UNALIGNED_LDST_SPEC, Unaligned access

The counter counts each access counted by MEM_ACCESS that is an unaligned Memory-read operation or unaligned Memory-write operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006C, LDREX_SPEC, Exclusive operation speculatively executed, Load-Exclusive

The counter counts each speculatively executed Load-Exclusive instruction.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the STREX_SPEC event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006D, STREX_PASS_SPEC, Exclusive operation speculatively executed, Store-Exclusive pass

The counter counts each speculatively executed Store-Exclusive instruction counted by STREX_SPEC that completed a write.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006E, STREX_FAIL_SPEC, Exclusive operation speculatively executed, Store-Exclusive fail

The counter counts each speculatively executed Store-Exclusive instruction counted by STREX_SPEC that fails to complete a write.

It is within the IMPLEMENTATION DEFINED definition of speculatively executed whether this includes conditional instructions that fail the condition code check.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x006F, STREX_SPEC, Exclusive operation speculatively executed, Store-Exclusive

The counter counts each speculatively executed Store-Exclusive instruction.

The definition of speculatively executed is IMPLEMENTATION DEFINED but it must be the same as for the LDREX_SPEC event.

Arm recommends that this event is implemented if it is not possible to implement the exclusive operation speculatively executed, Store-Exclusive pass, and exclusive operation speculatively executed, Store-Exclusive fail, events with the same degree of speculation as the LDREX_SPEC event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0070, LD_SPEC, Operation speculatively executed, load

The counter counts each operation counted by LDST_SPEC that is a load operation.

Operations due to Memory-reading instructions are counted as load operations.

It is IMPLEMENTATION DEFINED whether operations due to the prefetch instructions counted by PRF_SPEC are counted as load operations:

- If operations due to the prefetch instructions are counted as load operations, then they are counted by LD_SPEC.
- Otherwise, if operations due to the prefetch instructions are not counted as load operations by LD_SPEC, then it is further IMPLEMENTATION DEFINED which one of the following applies:
  - They are counted as data-processing operations.
  - They are counted by LDST_SPEC.

Arm recommends that if a prefetch operation is not implemented as a NOP and the PRF_SPEC event is not implemented, then the operation is counted as a load operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0071, ST_SPEC, Operation speculatively executed, store

The counter counts each operation counted by LDST_SPEC that is a store operation.

Operations due to Memory-writing instructions are counted as store operations.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0072, LDST_SPEC, Operation speculatively executed, load or store

The counter counts each operation counted by INST_SPEC that is a load operation or a store operation.

See LD_SPEC and ST_SPEC for these classifications.

If the PRF_SPEC event is implemented and operations due to the prefetch instructions counted by PRF_SPEC are not counted as load operations by LD_SPEC, then it is IMPLEMENTATION DEFINED whether these operations are counted by LDST_SPEC. Arm recommends that these operations are counted by LDST_SPEC.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0073, DP_SPEC, Operation speculatively executed, integer data processing

The counter counts each operation counted by INST_SPEC that is an integer data-processing operation.

An integer data-processing operation is any operation that is not counted as any of the following:

- A load or store operation, counted by LDST_SPEC.
- A Software change of PC operation, counted by PC_WRITE_SPEC.
- A scalar floating-point data processing operation, counted by VFP_SPEC.
- An Advanced SIMD, SVE or SME data processing operation, counted by SE_SPEC.
- A cryptographic operation, counted by CRYPTO_SPEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

That is, operations due to the following instructions are counted as integer data-processing operations:

- In AArch64 state instructions from the following sections:
  - Data processing - immediate.
  - Data processing - register.
  - System register instructions.
  - 'Instructions with register argument'.
  - System instructions other than Memory-writing instructions.
  - Hint instructions.
  - When FEAT_SVE is implemented and the SVE_SPEC event is implemented, non-SIMD SVE instructions.
- In AArch32 state instructions from the following sections:
  - Data-processing instructions.
  - PSTATE and banked register access instructions.
  - Banked register access instructions.
  - Miscellaneous instructions other than `ISB` and prefetches.
  - System register access instructions other than `LDC` and `STC` instructions.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0074`, ASE_SPEC, Operation speculatively executed, Advanced SIMD data processing

The counter counts each operation counted by ASE_SVE_SPEC that is an Advanced SIMD data-processing operation.

Operations due to the following instructions are counted as Advanced SIMD data-processing operations:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - Advanced SIMD scalar instructions, including those which operate on floating-point values.
  - If FEAT_Crypto is implemented, the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions. See The Cryptographic Extension in AArch64 state.
- For AArch32 state:
  - Instructions from Advanced SIMD data-processing instructions.
  - If FEAT_Crypto is implemented, the `VMULL` (P64 variant) instruction. See The Cryptographic Extension in AArch32 state.

This includes all operations that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing operations.
- Scalar floating-point data-processing operations.
- Load or store operations.
- Cryptographic data-processing operations counted by CRYPTO_SPEC. CRYPTO_SPEC does not count the Cryptographic data-processing operations included above.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0075`, VFP_SPEC, Operation speculatively executed, scalar floating-point data processing

The counter counts each operation counted by INST_SPEC that is a scalar floating-point operation.

Operations due to the following instructions are counted as scalar floating-point data-processing operations:

- In AArch64 state:
  - The scalar floating-point operations from Data processing - SIMD and floating-point. Only the scalar operations are counted. SIMD floating-point operations are not counted.
  - Operations that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.
- In AArch32 state:
  - Instructions from Floating-point data-processing instructions.
  - Operations that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `VCVT<mode>` (floating-point), `VCVT`, `VCVTT`, and `VCVTB`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This includes all scalar operations that operate on the SIMD&FP registers as floating-point values, except those that are counted as one of the following:

- SIMD scalar operations.
- Integer data-processing operations.
- Load or store operations.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0076`, PC_WRITE_SPEC, Operation speculatively executed, Software change of the PC

The counter counts each operation counted by INST_SPEC that is a Software change of the PC.

Software change of the PC operations are defined by the PC_WRITE_RETIRED event.

When FEAT_PMUv3p8 is implemented, the counter counts the operation even if the branch is not taken. Otherwise, it is IMPLEMENTATION DEFINED whether the counter counts the operation when the branch is not taken.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0077`, CRYPTO_SPEC, Operation speculatively executed, cryptographic data processing

The counter counts each operation counted by INST_SPEC that is a cryptographic operation.

Operations due to the following instructions are counted as cryptographic data-processing operations:

- In AArch64 state, instructions from the following sections:
  - The Cryptographic Extension in AArch64 state other than the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions.
  - If FEAT_SVE2 is implemented, SVE2 Crypto Extensions other than the SVE `PMULLB` and `PMULLT` (Q variants) instructions.
- In AArch32 state, instructions from The Cryptographic Extension in AArch32 state other than `VMULL` (P64 variant).

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0078`, BR_IMMED_SPEC, Branch speculatively executed, immediate branch

The counter counts each speculatively executed immediate branch instruction.

Operations due to the following instructions are counted as speculatively executed immediate branch instructions:

- For AArch32 state, the following instructions:
  - `B{<c>} <label>`.
  - `BL{<c>} <label>`.
  - `BLX{<c>} <label>`.
  - `CBZ <Rn>, <label>`.
  - `CBNZ <label>`.
- For AArch64 state, the following instructions:
  - `B <label>`.
  - `B.cond <label>`.
  - `BL <label>`.
  - `CBZ <Rn>, <label>`.
  - `CBNZ <Rn>, <label>`.
  - `TBZ <Rn>, <label>`.
  - `TBNZ <Rn>, <label>`.
  - `BC.cond <label>`.
  - `CB<cc> <Rn>, #<imm>, <label>`.
  - `CB<cc> <Rn>, #<Rm>, <label>`.
  - `CBB<cc> <Rn>, #<Rm>, <label>`.
  - `CBH<cc> <Rn>, #<Rm>, <label>`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Conditional branches are always counted, regardless of whether the branch is taken or not taken.

If the Context synchronization barrier instruction ISB is counted as a Software change of the PC instruction by PC_WRITE_SPEC, then it is IMPLEMENTATION DEFINED whether ISB is counted as a speculatively executed immediate branch instruction.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x0079`, BR_RETURN_SPEC, Branch speculatively executed, procedure return

The counter counts each speculatively executed procedure return instruction.

Operations due to the following instructions are counted as speculatively executed procedure return instructions:

- For AArch32 state, the following instructions:
  - `BX R14`.
  - `MOV PC, LR`.
  - `POP {..., PC}`.
  - `LDR PC, [SP], #offset`.
- For AArch64 state, the following instructions:
  - `RET`.
  - If FEAT_PAuth is implemented, `RETAA` and `RETAB`.
  - If FEAT_PAuth_LR is implemented, `RETAASPPC`, `RETABSPPC`, `RETAASPPCR`, and `RETABSPPCR`.

Note

The counter counts only the listed instructions as procedure returns. For example, it does not count the following AArch32 instructions as procedure return instructions:

- `BX R0`, because Rm != R14.
- `MOV PC, R0`, because Rm != R14.
- `LDM SP, {..., PC}`, because writeback is not specified.
- `LDR PC, [SP, #offset]`, because this specifies the wrong addressing mode.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x007A`, BR_INDIRECT_SPEC, Branch speculatively executed, indirect branch

The counter counts each speculatively executed indirect branch instruction.

Indirect branch instructions are Software change of the PC instructions other than exception-generating instructions and immediate branch instructions. Software change of the PC has the same definition as for the PC_WRITE_SPEC event. Immediate branch instructions are defined by the BR_IMMED_SPEC event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x007C`, ISB_SPEC, Barrier speculatively executed, ISB

The counter counts each speculatively executed Instruction Synchronization Barrier instruction.

This includes the `CP15ISB` instruction.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x007D`, DSB_SPEC, Barrier speculatively executed, DSB

The counter counts each speculatively executed data synchronization and speculative load barrier instruction.

This includes the `CP15DSB`, `SSBB`, and `PSSBB` instructions.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x007E, DMB_SPEC, Barrier speculatively executed, DMB

The counter counts each speculatively executed data memory barrier instruction.

This includes the CP15DSB instruction.

This does not include the implied barrier operations of load/store operations with release consistency semantics.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x007F, CSDB_SPEC, Barrier speculatively executed, CSDB

The counter counts each speculatively executed control speculation barrier instruction.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0090, RC_LD_SPEC, Release consistency operation speculatively executed, Load-Acquire

The counter counts each speculatively executed Memory-read operation with acquire or acquirepc semantics.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0091, RC_ST_SPEC, Release consistency operation speculatively executed, Store-Release

The counter counts each speculatively executed Memory-write operation with release semantics.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0092, L3D_CACHE_HIT_WR_FEXT_PROP, Level 3 data cache demand access first hit, write, fetched by external propagating transaction

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW_FEXT_PROP due to a demand Memory-write operation.

0x00A0, L3D_CACHE_RD, Level 3 data cache access, read

If the L3D_CACHE_RW event is implemented, the counter counts each access counted by L3D_CACHE_RW that is a Memory-read operation.

If the L3D_CACHE_RW event is not implemented, the counter counts each access counted by L3D_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A1, L3D_CACHE_WR, Level 3 data cache access, write

If the L3D_CACHE_RW event is implemented, the counter counts each access counted by L3D_CACHE_RW that is a Memory-write operation.

If the L3D_CACHE_RW event is not implemented, the counter counts each access counted by L3D_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x00A2`, L3D_CACHE_REFILL_RD, Level 3 data cache refill, read

The counter counts each access counted by both L3D_CACHE_RD and L3D_CACHE_REFILL.

That is, every refill of the Level 3 data or unified cache counted by L3D_CACHE_REFILL that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x00A3`, L3D_CACHE_REFILL_WR, Level 3 data cache refill, write

The counter counts each access counted by both L3D_CACHE_REFILL and L3D_CACHE_WR.

That is, every refill of the Level 3 data or unified cache counted by L3D_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x00A6`, L3D_CACHE_WB_VICTIM, Level 3 data cache write-back, victim

The counter counts each write-back counted by L3D_CACHE_WB that occurs because of a capacity eviction due to a line being allocated into the cache..

It is IMPLEMENTATION DEFINED whether this includes capacity evictions due to an agent other the PE counting the event. For example, a stashing request from outside of the PE, or an allocation due to another PE that shares the cache.

In all cases, the event is counted only if the eviction is also an L3D_CACHE_WB event. This means that, if the cache is shared, the eviction is counted only if it is attributable to the PE counting the event, as defined by L3D_CACHE_WB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

`0x00A7`, L3D_CACHE_WB_CLEAN, Level 3 data cache write-back, cleaning and coherency

The counter counts each write-back counted by L3D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction..

It is IMPLEMENTATION DEFINED whether the transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is counted when the dirty cache line is not also written back to a Level 2 cache or memory.

In all cases, the event is counted only if the write-back is also an L3D_CACHE_WB event. This means that, if the cache is shared, the write-back is counted only if it is attributable to the PE counting the event, as defined by L3D_CACHE_WB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x00A8, L3D_CACHE_INVAL, Level 3 data cache invalidate

The counter counts each invalidation of a cache line in the Level 3 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 3 data or unified cache invalidates a line in the Level 3 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 3 data cache is shared with other PEs. This event does not follow the general rule of Level 3 data cache events of only counting Attributable events.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A9, L2D_CACHE_HIT_RD_FEXT_PROP, Level 2 data cache demand access first hit, read, fetched by external propagating transaction

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW_FEXT_PROP due to a demand Memory-read operation.

0x00AA, L2D_CACHE_HIT_WR_FEXT_PROP, Level 2 data cache demand access first hit, write, fetched by external propagating transaction

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW_FEXT_PROP due to a demand Memory-write operation.

0x00AB, L2D_CACHE_HIT_RW_FEXT_PROP, Level 2 data cache demand access first hit, fetched by external propagating transaction

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW where the cache line was fetched in response to an external propagating transaction.

That is, the L2D_CACHE_EXT_PROP event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x00AC, L3D_CACHE_EXT_PROP, Level 3 data cache allocation due to external propagating transaction

The counter counts each operation that writes into the level 3 data cache that is initiated by an external propagating transaction.

These are allocations of cache lines in the Level 3 data cache that are not refills counted by L3D_CACHE_REFILL. This includes allocations triggered by:

- A memory update on another PE with an associated store shared hint.
- A stashing memory update from a device.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x00AD, L3D_CACHE_EXT_PROP_PRFM_IR, Level 3 data cache allocation due to external propagating transaction after prefetch with intent to read

The counter counts each access counted by L3D_CACHE_EXT_PROP that happens after local PE executes a prefetch with intent to read on the location.

0x00AE, L3D_CACHE_HIT_RW_FEXT_PROP, Level 3 data cache demand access first hit, fetched by external propagating transaction

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW where the cache line was fetched in response to an external propagating transaction.

That is, the L3D_CACHE_EXT_PROP event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x00AF, L3D_CACHE_HIT_RD_FEXT_PROP, Level 3 data cache demand access first hit, read, fetched by external propagating transaction

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW_FEXT_PROP due to a demand Memory-read operation.

0x00B0, STSHH_SPEC, Store shared hint instruction speculatively executed

The counter counts each speculatively executed `STSHH` instruction.

0x00B1, STSHH_KEEP_SPEC, Store shared hint instruction speculatively executed with KEEP policy

The counter counts each speculatively executed `STSHH KEEP` instruction counted by STSHH_SPEC.

0x00B2, STSHH_STRM_SPEC, Store shared hint instruction speculatively executed with STRM policy

The counter counts each speculatively executed `STSHH STRM` instruction counted by STSHH_SPEC.

0x00B3, PRFM_IR_SPEC, Prefetch with intent to read instruction speculatively executed

The counter counts each speculatively executed `PRFM IR` instruction counted by BASE_PRF_SPEC.

0x00B4, MEM_ACCESS_WR_STSHH, Memory write with store shared hint

The counter counts each access counted by MEM_ACCESS_WR that has an associated store shared hint.

This hint indicates that the memory update should trigger a transaction that propagates the updated location to other PEs.

0x00B5, MEM_ACCESS_WR_STSHH_KEEP, Memory write with store shared hint with keep policy

The counter counts each access counted by MEM_ACCESS_WR_STSHH that has an associated `KEEP` policy.

0x00B6, MEM_ACCESS_WR_STSHH_STRM, Memory write with store shared hint with stream policy

The counter counts each access counted by MEM_ACCESS_WR_STSHH that has an associated `STRM` policy.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x00B7, MEM_ACCESS_WR_STSHH_KEEP_NEAR, Memory write with store shared hint with keep policy performed near

The counter counts each access counted by MEM_ACCESS_WR_STSHH_KEEP that does not initiate a transaction downstream of the local PE caches.

0x00B8, MEM_ACCESS_WR_STSHH_KEEP_FAR, Memory write with store shared hint with keep policy performed far

The counter counts each access counted by MEM_ACCESS_WR_STSHH_KEEP that initiates a transaction downstream of the local PE caches.

0x00B9, L1D_CACHE_EXT_PROP, Level 1 data cache allocation due to external propagating transaction

The counter counts each operation that writes into the level 1 data cache that is initiated by an external propagating transaction.

These are allocations of cache lines in the Level 1 data cache that are not refills counted by L1D_CACHE_REFILL. This includes allocations triggered by:

- A memory update on another PE with an associated store shared hint.
- A stashing memory update from a device.

0x00BA, L1D_CACHE_EXT_PROP_PRFM_IR, Level 1 data cache allocation due to external propagating transaction after prefetch with intent to read

The counter counts each access counted by L1D_CACHE_EXT_PROP that happens after local PE executes a prefetch with intent to read on the location.

0x00BB, L1D_CACHE_HIT_RD_FEXT_PROP, Level 1 data cache demand access first hit, read, fetched by external propagating transaction

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW_FEXT_PROP due to a demand Memory-read operation.

0x00BC, L1D_CACHE_HIT_WR_FEXT_PROP, Level 1 data cache demand access first hit, write, fetched by external propagating transaction

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW_FEXT_PROP due to a demand Memory-write operation.

0x00BD, L1D_CACHE_HIT_RW_FEXT_PROP, Level 1 data cache demand access first hit, fetched by external propagating transaction

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW where the cache line was fetched in response to an external propagating transaction.

That is, the L1D_CACHE_EXT_PROP event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x00BE, L2D_CACHE_EXT_PROP, Level 2 data cache allocation due to external propagating transaction

The counter counts each operation that writes into the level 2 data cache that is initiated by an external propagating transaction.

These are allocations of cache lines in the Level 2 data cache that are not refills counted by L2D_CACHE_REFILL. This includes allocations triggered by:

- A memory update on another PE with an associated store shared hint.
- A stashing memory update from a device.

0x00BF, L2D_CACHE_EXT_PROP_PRFM_IR, Level 2 data cache allocation due to external propagating transaction after prefetch with intent to read

The counter counts each access counted by L2D_CACHE_EXT_PROP that happens after local PE executes a prefetch with intent to read on the location.

0x4000, SAMPLE_POP, Statistical Profiling sample population

The counter counts each operation that might be sampled, whether or not the operation was sampled.

The counter does not count when profiling is disabled.

PMCEID0_EL0[32] reads as 1 if this event is implemented and 0 otherwise.

0x4001, SAMPLE_FEED, Statistical Profiling sample taken

The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample does not collide with the previous sample.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

PMCEID0_EL0[33] reads as 1 if this event is implemented and 0 otherwise.

0x4002, SAMPLE_FILTRATE, Statistical Profiling sample taken and not removed by filtering

The counter counts each sample counted by SAMPLE_FEED that is not removed by filtering.

Sample records that are not removed by filtering, but are discarded before being written to the Profiling Buffer because of a Profiling Buffer management event or because Discard mode is implemented and enabled, are counted.

PMCEID0_EL0[34] reads as 1 if this event is implemented and 0 otherwise.

0x4003, SAMPLE_COLLISION, Statistical Profiling sample collided with previous sample

The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample collides with a previous sample.

For more information, see 'Sample collisions'.

PMCEID0_EL0[35] reads as 1 if this event is implemented and 0 otherwise.

0x4004, CNT_CYCLES, Constant frequency cycles

The counter increments at a constant frequency equal to the rate of increment of the System Counter, CNTPCT_EL0.

PMCEID0_EL0[36] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x4005, STALL_BACKEND_MEM, Memory stall cycles

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the last level of data or unified cache within the PE clock domain.

If Armv8.7 is implemented, the counter also counts backend stall cycles when a non-cacheable data access is in progress.

If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether the counter counts backend stall cycles when a non-cacheable data access is in progress.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand data miss in the last level of data or unified cache within the PE clock domain for any PE in the multithreaded processor.

PMCEID0_EL0[37] reads as 1 if this event is implemented and 0 otherwise.

0x4006, L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss

If the L1I_CACHE_RD event is implemented, the counter counts each access to the Level 1 instruction or unified cache counted by L1I_CACHE_RD that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.

If the L1I_CACHE_RD event is not implemented, the counter counts each access to the Level 1 instruction or unified cache counted by L1I_CACHE that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.

The event indicates to software that the access missed in the Level 1 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in instructions being returned from a Level 2 cache or elsewhere beyond the Level 1 instruction or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[38] reads as 1 if this event is implemented and 0 otherwise.

0x4009, L2D_CACHE_LMISS_RD, Level 2 data cache long-latency read miss

The counter counts each Memory-read operation to the Level 2 data or unified cache counted by L2D_CACHE_RD that incurs additional latency because it returns data from outside of the Level 1 and Level 2 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 2 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

in all Level 2 caches that the access looks up in and results in an access being made to a Level 3 cache or elsewhere beyond the Level 2 data or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[41] reads as 1 if this event is implemented and 0 otherwise.

`0x400A`, L2I_CACHE_LMISS, Level 2 instruction cache long-latency miss

If the L2I_CACHE_RD event is implemented, the counter counts each access to the Level 2 instruction or unified cache counted by L2I_CACHE_RD that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

If the L2I_CACHE_RD event is not implemented, the counter counts each access to the Level 2 instruction or unified cache counted by L2I_CACHE that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

The event indicates to software that the access missed in the Level 2 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in instructions being returned from a Level 3 cache or elsewhere beyond the Level 2 instruction or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[42] reads as 1 if this event is implemented and 0 otherwise.

`0x400B`, L3D_CACHE_LMISS_RD, Level 3 data cache long-latency read miss

The counter counts each Memory-read operation to the Level 3 data or unified cache counted by L3D_CACHE_RD that incurs additional latency because it returns data from outside of the Level 1 to Level 3 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 3 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 3 data or unified cache.

The counter does not count:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 3 caches that the access looks up in and results in an access being made to a lower level cache or elsewhere beyond the Level 3 data or unified cache. This might be counted as a Level 2 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[43] reads as 1 if this event is implemented and 0 otherwise.

`0x4010`, TRCEXTOUT0, Trace unit external output 0

The counter counts each event signaled by the trace unit on external event 0.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[48] reads as 1 if this event is implemented and 0 otherwise.

`0x4011`, TRCEXTOUT1, Trace unit external output 1

The counter counts each event signaled by the trace unit on external event 1.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[49] reads as 1 if this event is implemented and 0 otherwise.

`0x4012`, TRCEXTOUT2, Trace unit external output 2

The counter counts each event signaled by the trace unit on external event 2.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[50] reads as 1 if this event is implemented and 0 otherwise.

`0x4013`, TRCEXTOUT3, Trace unit external output 3

The counter counts each event signaled by the trace unit on external event 3.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[51] reads as 1 if this event is implemented and 0 otherwise.

`0x4018`, CTI_TRIGOUT4, Cross-trigger Interface output trigger 4

The counter counts each event signaled on CTI output trigger 4.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[56] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x4019`, CTI_TRIGOUT5, Cross-trigger Interface output trigger 5

The counter counts each event signaled on CTI output trigger 5.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[57] reads as 1 if this event is implemented and 0 otherwise.

`0x401A`, CTI_TRIGOUT6, Cross-trigger Interface output trigger 6

The counter counts each event signaled on CTI output trigger 6.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[58] reads as 1 if this event is implemented and 0 otherwise.

`0x401B`, CTI_TRIGOUT7, Cross-trigger Interface output trigger 7

The counter counts each event signaled on CTI output trigger 7.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[59] reads as 1 if this event is implemented and 0 otherwise.

`0x4020`, LDST_ALIGN_LAT, Access with additional latency from alignment

The counter counts each access counted by MEM_ACCESS that incurred additional latency due to the alignment of the address and size of data being accessed.

PMCEID1_EL0[32] reads as 1 if this event is implemented and 0 otherwise.

`0x4021`, LD_ALIGN_LAT, Load with additional latency from alignment

The counter counts each Memory-read operation counted by LDST_ALIGN_LAT.

PMCEID1_EL0[33] reads as 1 if this event is implemented and 0 otherwise.

`0x4022`, ST_ALIGN_LAT, Store with additional latency from alignment

The counter counts each Memory-write operation counted by LDST_ALIGN_LAT.

PMCEID1_EL0[34] reads as 1 if this event is implemented and 0 otherwise.

`0x4024`, MEM_ACCESS_CHECKED, Checked data memory access

The counter counts each memory access counted by MEM_ACCESS that accesses an Allocation Tag due to a Tag Check operation.

It is IMPLEMENTATION DEFINED whether the counter increments on a Tag Checked access made when Tag Check Faults are configured to be ignored by SCTLR_ELx.TCF or SCTLR_ELx.TCF0.

PMCEID1_EL0[36] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x4025, MEM_ACCESS_CHECKED_RD, Checked data memory access, read

The counter counts each Memory-read operation counted by MEM_ACCESS_CHECKED.

It is IMPLEMENTATION DEFINED whether the counter increments on a Tag Checked access made when Tag Check Faults are configured to be ignored by SCTLR_ELx.TCF or SCTLR_ELx.TCF0.

PMCEID1_EL0[37] reads as 1 if this event is implemented and 0 otherwise.

0x4026, MEM_ACCESS_CHECKED_WR, Checked data memory access, write

The counter counts each Memory-write operation counted by MEM_ACCESS_CHECKED.

It is IMPLEMENTATION DEFINED whether the counter increments on a Tag Checked access made when Tag Check Faults are configured to be ignored by SCTLR_ELx.TCF or SCTLR_ELx.TCF0.

PMCEID1_EL0[38] reads as 1 if this event is implemented and 0 otherwise.

0x8004, SIMD_INST_SPEC, Operation speculatively executed, SIMD

The counter counts each speculatively executed operation due to any of:

- An Advanced SIMD, SVE, or SME SIMD data-processing operation. Non-SIMD data-processing operations are not counted.
- An Advanced SIMD or SVE data-processing operation. Non-SIMD data-processing operations are not counted.
- A structure load/store of one or more SIMD&FP registers.
- A load and replicate to one or more SIMD&FP registers.
- A scalar load/store of a SIMD&FP Q register or pair of Q registers.
- An SVE or SME load/store.

It is IMPLEMENTATION DEFINED which Advanced SIMD operations are counted in AArch32 state.

When Armv9.5 is not implemented, it is IMPLEMENTATION DEFINED whether scalar loads and stores to SIMD&FP registers other than those listed above are counted.

When Armv9.5 is implemented, other scalar loads and stores to SIMD&FP registers are not counted.

0x8005, ASE_INST_SPEC, Operation speculatively executed, Advanced SIMD

The counter counts each speculatively executed operation counted by ASE_SVE_INST_SPEC that is classified as an Advanced SIMD operation.

Operations due to the following instructions are counted as Advanced SIMD operations:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - If FEAT_Crypto is implemented, the Advanced SIMD Cryptographic instructions. See The Cryptographic Extension in AArch64 state.
  - Advanced SIMD loads and stores.
- For AArch32 state:
  - Instructions from Advanced SIMD data-processing instructions.
  - If FEAT_Crypto is implemented, the `VMULL` (P64 variant) instruction. See The Cryptographic Extension in AArch32 state.

This includes all operations that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing operations.
- Scalar floating-point data-processing operations.

It is IMPLEMENTATION DEFINED whether Advanced SIMD scalar operations are counted as Advanced SIMD operations. This includes Advanced SIMD scalar instructions which operate on floating-point values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8006, SVE_INST_SPEC, Operation speculatively executed, SVE

The counter counts each speculatively executed operation that is classified as an SVE operation.

An SVE operation is any operation due to an instruction that operates on the SVE scalable vector and predicate registers that is not any of the following:

- When FEAT_SME is implemented, an operation counted by SME_SPEC as an SME operation.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to non-SIMD instructions.

Operations due to instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE operations.

0x8007, ASE_SVE_INST_SPEC, Operation speculatively executed, Advanced SIMD or SVE

The counter counts each speculatively executed operation counted by SE_INST_SPEC that is classified as an Advanced SIMD operation or an SVE operation.

See ASE_INST_SPEC and SVE_INST_SPEC for these classifications.

0x8008, UOP_SPEC, Microarchitectural operation speculatively executed

The counter counts each speculatively executed microarchitectural operation.

0x8009, ASE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD

The counter counts each speculatively executed microarchitectural operation counted by ASE_SVE_UOP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to Advanced SIMD scalar instructions.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

0x800A, SVE_UOP_SPEC, Microarchitectural operation speculatively executed, SVE

The counter counts each speculatively executed microarchitectural operation counted by ASE_SVE_UOP_SPEC due to an SVE instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to non-SIMD SVE instructions.

0x800B, ASE_SVE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD or SVE

The counter counts each speculatively executed microarchitectural operation counted by UOP_SPEC due to an A64 Advanced SIMD or SVE instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitecural operations due to Advanced SIMD scalar and non-SIMD SVE instructions.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

0x800C, SIMD_UOP_SPEC, Microarchitectural operation speculatively executed, SIMD

The counter counts each speculatively executed microarchitectural operation counted by UOP_SPEC due to any of:

- An SVE instruction other than non-SIMD SVE instructions.
- An A64 Advanced SIMD instruction other than an Advanced SIMD scalar instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x800E, SVE_MATH_SPEC, Operation speculatively executed, SVE math accelerator

The counter counts each speculatively executed math function operation due to any of the following instructions:

- SVE: FEXPA, FTMAD, FTSMUL, or FTSSEL.

0x8010, FP_SPEC, Floating-point operation speculatively executed, including SIMD

The counter counts each speculatively executed floating-point data-processing operation due to any of:

- An A64 scalar instruction.
- An A64 Advanced SIMD instruction.
- An SVE instruction.
- An SME instruction.

From Armv9.5, if FEAT_AFP is implemented, then the counter counts any operation due to any of the following instructions:

- Scalar: `FABS`, or `FNEG (scalar)`.
- Advanced SIMD: `FABS`, `FNEG (vector)`, `FRECPE`, or `FRSQRTE`.
- SVE: `FABS`, `FEXPA`, `FNEG`, `FRECPE`, `FRSQRTE`, or `FTSSEL`.

Otherwise, it is IMPLEMENTATION DEFINED whether the counter counts any operation due to the above instructions.

It is IMPLEMENTATION DEFINED whether the counter counts any operations due to the following instructions:

- Data movement instructions which move data without carrying out any operations on the data. This includes identity data-processing operations.
- Conditional select instructions.
- Floating-point conversion instructions.

Any operation counted by INT_SPEC is not counted.

It is IMPLEMENTATION DEFINED which floating-point data-processing operations are counted in AArch32 state.

Arm recommends that if FP_CVT_SPEC is implemented, FP_SPEC counts operations due to convert instructions.

Note

This event differs from the VFP_SPEC event which does not count SIMD operations.

0x8011, ASE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD

The counter counts each speculatively executed floating-point operation counted by ASE_SVE_FP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point operations are counted in AArch32 state.

0x8012, SVE_FP_SPEC, Floating-point operation speculatively executed, SVE

The counter counts each speculatively executed floating-point operation counted by ASE_SVE_FP_SPEC due to an SVE instruction.

0x8013, ASE_SVE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE

The counter counts each speculatively executed floating-point operation counted by SE_FP_SPEC due to an A64 Advanced SIMD or SVE instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8014, FP_HP_SPEC, Floating-point operation speculatively executed, half-precision

The counter counts each speculatively executed floating-point operation counted by FP_SPEC where the largest type is half-precision.

0x8015, ASE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD half-precision, data-processing

The counter counts each speculatively executed half-precision floating-point data-processing operation counted by ASE_SVE_FP_HP_SPEC due to an Advanced SIMD instruction.

0x8016, SVE_FP_HP_SPEC, Floating-point operation speculatively executed, SVE half-precision, data-processing

The counter counts each speculatively executed half-precision floating-point data-processing operation counted by ASE_SVE_FP_HP_SPEC due to an SVE instruction.

0x8017, ASE_SVE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE half-precision

The counter counts each speculatively executed half-precision floating-point operation counted by FP_HP_SPEC due to an Advanced SIMD or SVE instruction.

0x8018, FP_SP_SPEC, Floating-point operation speculatively executed, single-precision

The counter counts each speculatively executed floating-point operation counted by FP_SPEC where the largest type is single-precision.

0x8019, ASE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD single-precision, data-processing

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by ASE_SVE_FP_SP_SPEC due to an Advanced SIMD instruction.

0x801A, SVE_FP_SP_SPEC, Floating-point operation speculatively executed, SVE single-precision, data-processing

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by ASE_SVE_FP_SP_SPEC due to an SVE instruction.

0x801B, ASE_SVE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE single-precision, data-processing

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by FP_SP_SPEC due to an Advanced SIMD or SVE instruction.

0x801C, FP_DP_SPEC, Floating-point operation speculatively executed, double-precision, data-processing

The counter counts each speculatively executed floating-point data-processing operation counted by FP_SPEC where the largest type is double-precision.

0x801D, ASE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD double-precision, data-processing

The counter counts each speculatively executed double-precision floating-point data-processing operation counted by ASE_SVE_FP_DP_SPEC due to an Advanced SIMD instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x801E, SVE_FP_DP_SPEC, Floating-point operation speculatively executed, SVE double-precision, data-processing

The counter counts each speculatively executed double-precision floating-point data-processing operation counted by ASE_SVE_FP_DP_SPEC due to an SVE instruction.

0x801F, ASE_SVE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE double-precision, data-processing

The counter counts each speculatively executed double-precision floating-point data-processing operation due to an Advanced SIMD.

0x8020, FP_DIV_SPEC, Floating-point operation speculatively executed, divide

The counter counts each speculatively executed floating-point divide operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FDIV.
- Advanced SIMD: FDIV.
- SVE: FDIV or FDIVR.

It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

0x8021, ASE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD divide

The counter counts each speculatively executed floating-point divide operation counted by ASE_SVE_FP_DIV_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FDIV.

It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

0x8022, SVE_FP_DIV_SPEC, Floating-point operation speculatively executed, SVE divide

The counter counts each speculatively executed floating-point divide operation counted by ASE_SVE_FP_DIV_SPEC due to any of the following instructions:

- SVE: FDIV or FDIVR.

0x8023, ASE_SVE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE divide

The counter counts each speculatively executed floating-point divide operation counted by FP_DIV_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FDIV.
- SVE: FDIV or FDIVR.

It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

0x8024, FP_SQRT_SPEC, Floating-point operation speculatively executed, square root

The counter counts each speculatively executed floating-point square-root operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FSQRT.
- Advanced SIMD: FSQRT.
- SVE: FSQRT.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8025, ASE_FP_SQRT_SPEC, Floating-point operation speculatively executed, Advanced SIMD square root

The counter counts each speculatively executed floating-point square-root operation counted by ASE_SVE_FP_SQRT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FSQRT.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

0x8026, SVE_FP_SQRT_SPEC, Floating-point operation speculatively executed, SVE square root

The counter counts each speculatively executed floating-point square-root operation counted by ASE_SVE_FP_SQRT_SPEC due to any of the following instructions:

- SVE: FSQRT.

0x8027, ASE_SVE_FP_SQRT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE square-root

The counter counts each speculatively executed floating-point square-root operation counted by FP_SQRT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FSQRT.
- SVE: FSQRT.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

0x8028, FP_FMA_SPEC, Floating-point operation speculatively executed, FMA

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FMADD, FMSUB, FNMADD, or FNMSUB.
- Advanced SIMD: FMLA or FMLS.
- When FEAT_BF16 is implemented, Advanced SIMD: BFMLALB or BFMLALT.
- When FEAT_FCMA is implemented, Advanced SIMD: FCMLA.
- When FEAT_FP8FMA is implemented, Advanced SIMD: FMLALB, FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT, or FMLALT.
- When FEAT_FHM is implemented, Advanced SIMD: FMLAL, FMLAL2, FMLSL, or FMLSL2.
- SVE: BFMLALB, BFMLALT, FCMLA, FMAD, FMLA, FMLS, FMSB, FNMAD, FNMLA, FNMLS, FNMSB, or FTMAD.
- SVE2: BFMLA, BFMLS, BFMLSLB, BFMLSLT, FMLALB, FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT, FMLALT, FMLSLB, or FMLSLT.
- SME2: BFMLA, BFMLAL, BFMLS, BFMLSL, FMLA, FMLAL, FMLALL, FMLS, or FMLSL.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

0x8029, ASE_FP_FMA_SPEC, Floating-point operation speculatively executed, Advanced SIMD FMA

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by ASE_SVE_FP_FMA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMLA or FMLS.
- When FEAT_BF16 is implemented, Advanced SIMD: BFMLALB or BFMLALT.
- When FEAT_FCMA is implemented, Advanced SIMD: FCMLA.
- When FEAT_FP8FMA is implemented, Advanced SIMD: FMLALB, FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT, or FMLALT.
- When FEAT_FHM is implemented, Advanced SIMD: FMLAL, FMLAL2, FMLSL, or FMLSL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

0x802A, SVE_FP_FMA_SPEC, Floating-point operation speculatively executed, SVE FMA

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by ASE_SVE_FP_FMA_SPEC due to any of the following instructions:

- SVE: BFMLALB, BFMLALT, FCMLA, FMAD, FMLA, FMLS, FMSB, FNMAD, FNMLA, FNMLS, FNMSB, or FTMAD.
- SVE2: BFMLA, BFMLS, BFMLSLB, BFMLSLT, FMLALB, FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT, FMLALT, FMLSLB, or FMLSLT.

0x802B, ASE_SVE_FP_FMA_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE FMA

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_FMA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMLA or FMLS.
- When FEAT_BF16 is implemented, Advanced SIMD: BFMLALB or BFMLALT.
- When FEAT_FCMA is implemented, Advanced SIMD: FCMLA.
- When FEAT_FP8FMA is implemented, Advanced SIMD: FMLALB, FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT, or FMLALT.
- When FEAT_FHM is implemented, Advanced SIMD: FMLAL, FMLAL2, FMLSL, or FMLSL2.
- SVE: BFMLALB, BFMLALT, FCMLA, FMAD, FMLA, FMLS, FMSB, FNMAD, FNMLA, FNMLS, FNMSB, or FTMAD.
- SVE2: BFMLA, BFMLS, BFMLSLB, BFMLSLT, FMLALB, FMLALLBB, FMLALLBT, FMLALLTB, FMLALLTT, FMLALT, FMLSLB, or FMLSLT.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

0x802C, FP_MUL_SPEC, Floating-point operation speculatively executed, multiply

The counter counts each speculatively executed floating-point multiply operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FMUL or FMULX.
- Advanced SIMD: FMUL or FMULX.
- SVE: FMUL, FMULX, or FTSMUL.
- SVE2: BFMUL.

It is IMPLEMENTATION DEFINED which floating-point multiply operations are counted in AArch32 state.

0x802D, ASE_FP_MUL_SPEC, Floating-point operation speculatively executed, Advanced SIMD multiply

The counter counts each speculatively executed floating-point multiply operation counted by ASE_SVE_FP_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.

It is IMPLEMENTATION DEFINED which floating-point multiply operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x802E, SVE_FP_MUL_SPEC, Floating-point operation speculatively executed, SVE multiply

The counter counts each speculatively executed floating-point multiply operation counted by ASE_SVE_FP_MUL_SPEC due to any of the following instructions:

- SVE: FMUL, FMULX, or FTSMUL.
- SVE2: BFMUL.

0x802F, ASE_SVE_FP_MUL_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE multiply

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.
- SVE: FMUL, FMULX, or FTSMUL.
- SVE2: BFMUL.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

0x8030, FP_ADDSUB_SPEC, Floating-point operation speculatively executed, add or subtract

The counter counts each speculatively executed floating-point add or subtract operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FADD or FSUB.
- Advanced SIMD: FABD, FADD, or FSUB.
- SVE: FABD, FADD, FSUB, or FSUBR.
- SVE2: BFADD or BFSUB.
- SME2: BFADD, BFSUB, FADD, or FSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

0x8031, ASE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD add or subtract

The counter counts each speculatively executed floating-point add or subtract operation counted by ASE_SVE_FP_ADDSUB_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

0x8032, SVE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, SVE add or subtract

The counter counts each speculatively executed floating-point add or subtract operation counted by ASE_SVE_FP_ADDSUB_SPEC due to any of the following instructions:

- SVE: FABD, FADD, FSUB, or FSUBR.
- SVE2: BFADD or BFSUB.

0x8033, ASE_SVE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract

The counter counts each speculatively executed floating-point add or subtract operation counted by FP_ADDSUB_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SVE: FABD, FADD, FSUB, or FSUBR.
- SVE2: BFADD or BFSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

`0x8034`, FP_RECPE_SPEC, Floating-point operation speculatively executed, reciprocal estimate

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.
- SVE: FRECPE or FRSQRTE.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

`0x8035`, ASE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by ASE_SVE_FP_RECPE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

`0x8036`, SVE_FP_RECPE_SPEC, Floating-point operation speculatively executed, SVE reciprocal estimate

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by ASE_SVE_FP_RECPE_SPEC due to any of the following instructions:

- SVE: FRECPE or FRSQRTE.

`0x8037`, ASE_SVE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by FP_RECPE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.
- SVE: FRECPE or FRSQRTE.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

`0x8038`, FP_CVT_SPEC, Floating-point operation speculatively executed, convert

The counter counts each speculatively executed floating-point convert operation due to any of:

- An A64 scalar floating-point conversion instruction.
- An Advanced SIMD floating-point conversion instruction.
- An SVE register targeting floating-point conversion instruction.

This includes both conversions between floating-point types, and conversions between integer and floating-point types.

It is IMPLEMENTATION DEFINED which floating-point convert operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8039, ASE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD convert

The counter counts each speculatively executed floating-point convert operation counted by ASE_SVE_FP_CVT_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point convert operations are counted in AArch32 state.

0x803A, SVE_FP_CVT_SPEC, Floating-point operation speculatively executed, SVE convert

The counter counts each speculatively executed floating-point convert operation counted by ASE_SVE_FP_CVT_SPEC due to an SVE instruction.

0x803B, ASE_SVE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE convert

The counter counts each speculatively executed floating-point convert operation counted by FP_CVT_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point convert operations are counted in AArch32 state.

0x803C, SVE_FP_AREDUCE_SPEC, Floating-point operation speculatively executed, SVE accumulating reduction

The counter counts each speculatively executed floating-point accumulating reduction operation counted by SVE_FP_SPEC due to any of the following instructions:

- SVE: FADDA.

0x803D, ASE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise add step or pairwise reduce step

The counter counts each speculatively executed floating-point pairwise add step or pairwise reduce step operation counted by ASE_FP_VREDUCE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXP, FMINNMP, or FMINP.

It is IMPLEMENTATION DEFINED which floating-point pairwise add step or pairwise reduce step operations are counted in AArch32 state.

0x803E, SVE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, SVE pairwise or reduction

The counter counts each speculatively executed floating-point treewise reduction or pairwise operations counted by ASE_SVE_FP_VREDUCE_SPEC due to any of the following instructions:

- SVE: FADDV, FMAXNMV, FMAXV, FMINNMV, or FMINV.
- SVE2: FADDP, FADDQV, FMAXNMP, FMAXNMQV, FMAXP, FMAXQV, FMINNMP, FMINNMQV, FMINP, or FMINQV.

Note

Treewise reduction instructions might be executed and counted as multiple pairwise operations.

0x803F, ASE_SVE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction

The counter counts each speculatively executed floating-point reduction operation counted by ASE_SVE_FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXNMV, FMAXP, FMAXV, FMINNMP, FMINNMV, FMINP, or FMINV.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SVE: FADDV, FMAXNMV, FMAXV, FMINNMV, or FMINV.
- SVE2: FADDP, FADDQV, FMAXNMP, FMAXNMQV, FMAXP, FMAXQV, FMINNMP, FMINNMQV, FMINP, or FMINQV.

It is IMPLEMENTATION DEFINED which floating-point reduction operations are counted in AArch32 state.

`0x8040`, INT_SPEC, Integer operation speculatively executed

The counter counts each speculatively executed integer arithmetic operation due to any of:

- An A64 scalar data-processing instruction.
- An Advanced SIMD instruction.
- An SVE data-processing instruction.
- An SME data-processing instruction.

It is IMPLEMENTATION DEFINED whether the counter counts any operation due to the following instructions:

- Scalar: `FABS`, `FNEG (scalar)`.
- Advanced SIMD: `FABS`, `FNEG (vector)`, `FRECPE`, `FRSQRTE`.
- SVE: `FABS`, `FEXPA`, `FNEG`, `FRECPE`, `FRSQRTE`, `FTSSEL`.
- Data movement instructions which move data without carrying out any operations on the data. This includes identity data-processing operations.
- Conditional select instructions.
- Floating-point conversion instructions.

Any operation counted by FP_SPEC is not counted.

It is IMPLEMENTATION DEFINED which integer arithmetic operations are counted in AArch32 state.

`0x8041`, ASE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD

The counter counts each speculatively executed integer operation counted by ASE_SVE_INT_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD integer operations are counted in AArch32 state.

`0x8042`, SVE_INT_SPEC, Integer operation speculatively executed, SVE

The counter counts each speculatively executed integer operation counted by ASE_SVE_INT_SPEC due to an SVE instruction.

`0x8043`, ASE_SVE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE

The counter counts each speculatively executed integer arithmetic operation counted by SE_INT_SPEC due to an A64 Advanced SIMD or SVE instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD integer operations are counted in AArch32 state.

`0x8044`, INT_DIV_SPEC, Integer operation speculatively executed, divide

The counter counts each speculatively executed integer divide operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SDIV or UDIV.
- SVE: SDIV, SDIVR, UDIV, or UDIVR.

It is IMPLEMENTATION DEFINED which integer divide operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8045, INT_DIV64_SPEC, Integer operation speculatively executed, 64-bit divide

The counter counts each speculatively executed integer divide operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SDIV or UDIV.
- SVE: SDIV, SDIVR, UDIV, or UDIVR.

The counter only counts operations with 64-bit operands or vector elements.

0x8046, SVE_INT_DIV_SPEC, Integer operation speculatively executed, SVE divide

The counter counts each speculatively executed integer divide operation counted by INT_DIV_SPEC due to any of the following instructions:

- SVE: SDIV, SDIVR, UDIV, or UDIVR.

0x8047, SVE_INT_DIV64_SPEC, Integer operation speculatively executed, SVE 64-bit divide

The counter counts each speculatively executed integer 64-bit divide operation counted by INT_DIV64_SPEC due to any of the following instructions:

- SVE: SDIV, SDIVR, UDIV, or UDIVR.

The counter only counts operations with 64-bit operands or vector elements.

0x8048, INT_MUL_SPEC, Integer operation speculatively executed, multiply

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, SMADDL, SMULH, UMADDL, or UMULH.
- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.
- When FEAT_RDM is implemented, Advanced SIMD: SQRDMLAH or SQRDMLSH.
- SVE: MAD, MLA (vectors), MLS (vectors), MSB, MUL, SMULH (predicated), or UMULH (predicated).
- SVE2: CMLA, MLA (indexed), MLS (indexed), MUL, PMUL, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH (unpredicated), SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, SQDMULH, SQDMULLB, SQDMULLT, SQRDCMLAH, SQRDMLAH, SQRDMLSH, SQRDMULH, UMLALB, UMLALT, UMLSLB, UMLSLT, UMULH (unpredicated), UMULLB, or UMULLT.
- SME2: SMLAL, SMLALL, SMLSL, SMLSLL, SUMLALL, UMLAL, UMLALL, UMLSL, UMLSLL, or USMLALL.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

0x8049, ASE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD multiply

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by ASE_SVE_INT_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.
- When FEAT_RDM is implemented, Advanced SIMD: SQRDMLAH or SQRDMLSH.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x804A, SVE_INT_MUL_SPEC, Integer operation speculatively executed, SVE multiply

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by ASE_SVE_INT_MUL_SPEC due to any of the following instructions:

- SVE: MAD, MLA (vectors), MLS (vectors), MSB, MUL, SMULH (predicated), or UMULH (predicated).
- SVE2: CMLA, MLA (indexed), MLS (indexed), MUL, PMUL, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH (unpredicated), SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, SQDMULH, SQDMULLB, SQDMULLT, SQRDCMLAH, SQRDMLAH, SQRDMLSH, SQRDMULH, UMLALB, UMLALT, UMLSLB, UMLSLT, UMULH (unpredicated), UMULLB, or UMULLT.

0x804B, ASE_SVE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE multiply

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by INT_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.
- When FEAT_RDM is implemented, Advanced SIMD: SQRDMLAH or SQRDMLSH.
- SVE: MAD, MLA (vectors), MLS (vectors), MSB, MUL, SMULH (predicated), or UMULH (predicated).
- SVE2: CMLA, MLA (indexed), MLS (indexed), MUL, PMUL, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH (unpredicated), SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, SQDMULH, SQDMULLB, SQDMULLT, SQRDCMLAH, SQRDMLAH, SQRDMLSH, SQRDMULH, UMLALB, UMLALT, UMLSLB, UMLSLT, UMULH (unpredicated), UMULLB, or UMULLT.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

0x804C, INT_MUL64_SPEC, Integer operation speculatively executed, 64×64 multiply

The counter counts each speculatively executed 64×64 integer multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, SMULH, or UMULH.
- SVE: MAD, MLA (vectors), MLS (vectors), MSB, MUL (vectors, predicated), SMULH (predicated), or UMULH (predicated).
- SVE2: CMLA (vectors), MLA (indexed), MLS (indexed), MUL, SMULH (unpredicated), SQDMULH, SQRDCMLAH (vectors), SQRDMLAH, SQRDMLSH, SQRDMULH, or UMULH (unpredicated).

The counter only counts operations that perform a 64-bit × 64-bit integer multiply operations.

0x804D, SVE_INT_MUL64_SPEC, Integer operation speculatively executed, SVE 64×64 multiply

The counter counts each speculatively executed 64×64 integer multiply operation counted by INT_MUL64_SPEC due to any of the following instructions:

- SVE: MAD, MLA (vectors), MLS (vectors), MSB, MUL (vectors, predicated), SMULH (predicated), or UMULH (predicated).
- SVE2: CMLA (vectors), MLA (indexed), MLS (indexed), MUL, SMULH (unpredicated), SQDMULH, SQRDCMLAH (vectors), SQRDMLAH, SQRDMLSH, SQRDMULH, or UMULH (unpredicated).

The counter only counts operations that perform a 64-bit × 64-bit integer multiply operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x804E, INT_MULH64_SPEC, Integer operation speculatively executed, 64×64 multiply returning high part

The counter counts each speculatively executed widening 64×64 integer multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SMULH or UMULH.
- SVE: SMULH (predicated) or UMULH (predicated).
- SVE2: SMULH (unpredicated), SQDMULH, SQRDMULH, or UMULH (unpredicated).

These instructions perform 64-bit × 64-bit integer multiply operations.

0x804F, SVE_INT_MULH64_SPEC, Integer operation speculatively executed, SVE 64×64 multiply high part

The counter counts each speculatively executed 64×64 integer multiply returning high part operation counted by INT_MULH64_SPEC due to any of the following instructions:

- SVE: SMULH (predicated) or UMULH (predicated).
- SVE2: SMULH (unpredicated), SQDMULH, SQRDMULH, or UMULH (unpredicated).

0x8051, ASE_FP_BF16_SPEC, Floating-point operation speculatively executed, Advanced SIMD BFloat16, data-processing

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by ASE_SVE_FP_BF16_SPEC due to an Advanced SIMD instruction.

0x8052, SVE_FP_BF16_SPEC, Floating-point operation speculatively executed, SVE BFloat16, data-processing

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by ASE_SVE_FP_BF16_SPEC due to an SVE instruction.

0x8053, ASE_SVE_FP_BF16_SPEC, Floating-point operation speculatively executed, SVE or Advanced SIMD BFloat16, data-processing

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by FP_BF16_SPEC due to an Advanced SIMD or SVE instruction.

0x8054, FP_BF16_SPEC, Floating-point operation speculatively executed, BFloat16, data-processing

The counter counts each speculatively executed floating-point data-processing operation counted by FP_SPEC where the largest type is BFloat16.

0x8056, SVE_SPEC, Operation speculatively executed, SVE data processing

The counter counts each operation counted by ASE_SVE_SPEC that is an SVE data-processing operation.

An SVE data-processing operation is any operation due to an instruction that operates on the SVE scalable vector and predicate registers that is not any of the following:

- A load or store operation. These are counted by SVE_INST_SPEC.
- If the Cryptographic Extension is implemented, an operation counted by CRYPTO_SPEC as a Cryptographic data-processing operation.
- When FEAT_SME is implemented, an operation counted by SME_SPEC as an SME data-processing operation.

If the Cryptographic Extension and FEAT_SVE2 are implemented, operations due to the SVE `PMULLB` and `PMULLT` (Q variants) instructions are not counted by CRYPTO_SPEC and are counted as SVE data-processing operations.

Operations due to data-processing instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE data-processing operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8057, ASE_SVE_SPEC, Operation speculatively executed, Advanced SIMD data processing or SVE data processing

The counter counts each operation counted by INST_SPEC that is an Advanced SIMD data-processing operation or an SVE data-processing operation.

See ASE_SPEC and SVE_SPEC for these classifications.

0x8058, NONFP_SPEC, Non-floating-point operation speculatively executed

The counter counts each speculatively executed data-processing operation due to any of:

- A scalar instruction that would be counted by the DP_SPEC event.
- An A64 Advanced SIMD data processing instruction defined in the section Data processing - SIMD and floating-point that would not be counted by FP_SPEC.
- An SVE instruction with vector source or destination registers that would not be counted by FP_SPEC.
- An SME instruction that would not be counted by FP_SPEC.

It is IMPLEMENTATION DEFINED which non-floating-point data-processing operations are counted in AArch32 state.

0x8059, ASE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD

The counter counts each speculatively executed non-floating-point operation counted by ASE_SVE_NONFP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD non-floating-point data-processing operations are counted in AArch32 state.

0x805A, SVE_NONFP_SPEC, Non-floating-point operation speculatively executed, SVE

The counter counts each speculatively executed non-floating-point operation counted by ASE_SVE_NONFP_SPEC due to an SVE instruction.

0x805B, ASE_SVE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD or SVE

The counter counts each speculatively executed non-floating-point operation counted by NONFP_SPEC due to an A64 Advanced SIMD or SVE instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD non-floating-point data-processing operations are counted in AArch32 state.

0x805D, ASE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD reduction

The counter counts each speculatively executed across-vector and pairwise integer reduction operation counted by ASE_SVE_INT_VREDUCE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: ADDP, ADDV, SADALP, SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADALP, UADDLP, UADDLV, UMAXP, UMAXV, UMINP, or UMINV.

It is IMPLEMENTATION DEFINED which across-vector and pairwise integer reduction operations are counted in AArch32 state.

0x805E, SVE_INT_VREDUCE_SPEC, Integer operation speculatively executed, SVE reduction

The counter counts each speculatively executed across-vector and pairwise integer reduction operation counted by ASE_SVE_INT_VREDUCE_SPEC due to any of the following instructions:

- SVE: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, or UMINV.
- SVE2: ADDP, ADDQV, ANDQV, EORQV, ORQV, SADALP, SMAXP, SMAXQV, SMINP, SMINQV, UADALP, UMAXP, UMAXQV, UMINP, or UMINQV.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x805F, ASE_SVE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE reduction

The counter counts each speculatively executed across-vector and pairwise integer reduction operation counted by INT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: ADDP, ADDV, SADALP, SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADALP, UADDLP, UADDLV, UMAXP, UMAXV, UMINP, or UMINV.
- SVE: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, or UMINV.
- SVE2: ADDP, ADDQV, ANDQV, EORQV, ORQV, SADALP, SMAXP, SMAXQV, SMINP, SMINQV, UADALP, UMAXP, UMAXQV, UMINP, or UMINQV.

It is IMPLEMENTATION DEFINED which across-vector and pairwise integer reduction operations are counted in AArch32 state.

0x8060, SVE_PERM_SPEC, Operation speculatively executed, SVE permute

The counter counts each speculatively executed vector or predicate permute operation due to any of the following instructions:

- SVE: CLASTA, CLASTB, COMPACT, CPY (SIMD&FP scalar), CPY (scalar), DUP (indexed), DUP (scalar), EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV (vector), REVB, REVH, REVW, SPLICE, SUNPKHI, SUNPKLO, TBL, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, or ZIP2.
- SVE2: EXPAND, REVD, TBX, or TBXQ.

0x8061, SVE_PERM_IGRANULE_SPEC, Operation speculatively executed, SVE intra-granule permute

The counter counts each speculatively executed vector or predicate permute operation within a 128-bit vector granule or 16-bit predicate granule due to any of the following instructions:

- SVE: REVB, REVH, REVW, TRN1 (vectors), or TRN2 (vectors).
- SVE2: REVD.

0x8062, SVE_PERM_XGRANULE_SPEC, Operation speculatively executed, SVE cross-granule permute

The counter counts each speculatively executed vector or predicate permute operation that can cross between 128-bit vector granules or 16-bit predicate granules due to any of the following instructions:

- SVE: CLASTA, CLASTB, COMPACT, CPY (SIMD&FP scalar), CPY (scalar), DUP (indexed), DUP (scalar), EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV (vector), SPLICE, SUNPKHI, SUNPKLO, TBL, TRN1 (predicates), TRN2 (predicates), UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, or ZIP2.
- SVE2: EXPAND, TBX, or TBXQ.

0x8063, SVE_PERM_VARIABLE_SPEC, Operation speculatively executed, SVE programmable permute

The counter counts each speculatively executed variable vector permute operation due to any of the following instructions:

- SVE: CLASTA, CLASTB, COMPACT, LASTA, LASTB, SPLICE, or TBL.
- SVE2: EXPAND.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8064, SVE_XPIPE_SPEC, Operation speculatively executed, SVE cross-pipe

The counter counts each speculatively executed cross-pipeline transfer operation due to any of the following instructions:

- SVE: CLASTA (scalar), CLASTB (scalar), CNTP (predicate), CPY (scalar), DECP (scalar), DUP (scalar), INCP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalars), INSR (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), UQINCP (scalar), WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), or WHILELT (predicate).
- SVE2: CNTP (predicate as counter), FIRSTP, LASTP, PSEL, WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, WHILELT, WHILERW, or WHILEWR.

0x8065, SVE_XPIPE_Z2R_SPEC, Operation speculatively executed, SVE vector to scalar cross-pipe

The counter counts each speculatively executed vector to general-purpose scalar cross-pipeline transfer operation counted by SVE_XPIPE_SPEC due to any of the following instructions:

- SVE: CLASTA (scalar), CLASTB (scalar), CNTP (predicate), DECP (scalar), INCP (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), or UQINCP (scalar).
- SVE2: CNTP (predicate as counter), FIRSTP, or LASTP.

0x8066, SVE_XPIPE_R2Z_SPEC, Operation speculatively executed, SVE scalar to vector cross-pipe

The counter counts each speculatively executed general-purpose scalar to vector cross-pipeline transfer operation counted by SVE_XPIPE_SPEC due to any of the following instructions:

- SVE: CPY (scalar), DUP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalars), INSR (scalar), WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), or WHILELT (predicate).
- SVE2: PSEL, WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, WHILELT, WHILERW, or WHILEWR.

0x8067, SVE_PGEN_NVEC_SPEC, Operation speculatively executed, SVE predicate-only

The counter counts each speculatively executed predicate-generating operation that does not read vector registers due to any of the following instructions:

- SVE: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, EOR (predicates), EORS, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE (predicate), PTRUES, PUNPKHI, PUNPKLO, RDFFR, RDFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), ZIP1 (predicates), or ZIP2 (predicates).
- SVE2: PEXT or PTRUE (predicate as counter).

0x8068, SVE_PGEN_SPEC, Operation speculatively executed, SVE predicate generating

The counter counts each speculatively executed predicate-generating operation due to any of the following instructions:

- SVE: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, CMP<cc>, EOR (predicates), EORS, FAC<cc>, FCM<cc>, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE (predicate), PTRUES, PUNPKHI, PUNPKLO, RDFFR, RDFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), WHILELT (predicate), ZIP1 (predicates), or ZIP2 (predicates).
- SVE2: MATCH, NMATCH, PEXT, PMOV (to predicate), PSEL, PTRUE (predicate as counter), WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, WHILELT, WHILERW, or WHILEWR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8069, SVE_PGEN_FLG_SPEC, Operation speculatively executed, SVE predicate flag setting

The counter counts each speculatively executed predicate-generating operation that sets condition flags due to any of the following instructions:

- SVE: ANDS, BICS, BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, CMP<cc>, EORS, NANDS, NORS, ORNS, ORRS, PFIRST, PNEXT, PTRUES, RDFFRS, WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), or WHILELT (predicate).
- SVE2: MATCH, NMATCH, WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, WHILELT, WHILERW, or WHILEWR.

0x806A, SVE_PGEN_CMP_SPEC, Operation speculatively executed, SVE vector compare

The counter counts each speculatively executed vector compare operation due to any of the following instructions:

- SVE: CMP<cc>, FAC<cc>, or FCM<cc>

0x806B, SVE_PGEN_FCM_SPEC, Floating-point operation speculatively executed, SVE vector compare

The counter counts each speculatively executed vector floating-point compare operation due to any of the following instructions:

- SVE: FAC<cc> or FCM<cc>

0x806C, SVE_PGEN_LOGIC_SPEC, Operation speculatively executed, SVE predicate logical

The counter counts each speculatively executed predicate logical operation due to any of the following instructions:

- SVE: AND (predicates), ANDS, BIC (predicates), BICS, EOR (predicates), EORS, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), or ORRS.

0x806D, SVE_PPERM_SPEC, Operation speculatively executed, SVE predicate permute

The counter counts each speculatively executed predicate permute operation due to any of the following instructions:

- SVE: PUNPKHI, PUNPKLO, REV (predicate), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), ZIP1 (predicates), or ZIP2 (predicates).

0x806E, SVE_PSCAN_SPEC, Operation speculatively executed, SVE predicate scan

The counter counts each speculatively executed predicate scanning and generation operation due to any of the following instructions:

- SVE: BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, PFIRST, or PNEXT.

0x806F, SVE_PCNT_SPEC, Operation speculatively executed, SVE predicate count

The counter counts each speculatively executed predicate population count operation due to any of the following instructions:

- SVE: CNTP (predicate), DECP, INCP, SQDECP, SQINCP, UQDECP, or UQINCP.
- SVE2: CNTP (predicate as counter).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8070, SVE_PLOOP_WHILE_SPEC, Operation speculatively executed, SVE predicate loop while

The counter counts each speculatively executed counted predicate generation operation due to any of the following instructions:

- SVE: WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), or WHILELT (predicate).
- SVE2: WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, or WHILELT.

0x8071, SVE_PLOOP_TEST_SPEC, Operation speculatively executed, SVE predicate loop test

The counter counts each speculatively executed loop predicate test operation due to any of the following instructions:

- SVE: BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), or WHILELT (predicate).
- SVE2: WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, or WHILELT.

0x8072, SVE_PLOOP_ELTS_SPEC, Operation speculatively executed, SVE predicate loop elements

The counter counts each speculatively executed loop predicate generation operation due to any of the following instructions:

- SVE: WHILELE (predicate), WHILELO (predicate), WHILELS (predicate), or WHILELT (predicate).
- SVE2: WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, or WHILELT.

The counter increments by (128 ÷ CSIZE).

Note

Multiplying the counter value by (VL ÷ 128) determines the number of vector elements speculatively processed by while loops.

0x8073, SVE_PLOOP_TERM_SPEC, Operation speculatively executed, SVE predicate loop termination

The counter counts each speculatively executed loop-terminating predicate generation operation due to any of:

- An SVE `WHILELE`, `WHILELO`, `WHILELS`, or `WHILELT` instruction which sets PSTATE.N to 0.
- An SVE `BRKAS`, `BRKBS`, `BRKNS`, `BRKPAS`, or `BRKPBS` instruction which sets PSTATE.C to 1.
- An SVE `CTERMEQ` or `CTERMNE` instruction which sets PSTATE.N to 1 and PSTATE.V to 0.
- An SVE2 `WHILEGE`, `WHILEGT`, `WHILEHI`, or `WHILEHS` instruction which sets PSTATE.N to 0.

0x8074, SVE_PRED_SPEC, Operation speculatively executed, SIMD predicated

The counter counts each speculatively executed predicated SIMD operation.

All of the following SVE operations are counted as predicated SIMD operations:

- Data-processing or load operations that write to one or more SVE Z vector destination registers under a Governing predicate using either zeroing or merging predication.
- Predicated stores of one or more SVE Z vector registers.

It is IMPLEMENTATION DEFINED whether data-processing operations due to instructions with a single Governing predicate operand that determines the Active elements which do not write to any SVE Z vector destination registers using either zeroing or merging predication are counted. For example, INCP.

Operations where all predicates are FALSE which result in no effective change in the architectural state are counted.

When FEAT_SPE is implemented, Arm recommends this event is implemented consistently with the PRED field in the SVE data-processing format Operation Type packet.

All speculatively executed SME operations due to instructions with at least one Governing predicate operand that determines the Active elements are counted as predicated SIMD operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

`0x8075`, SVE_PRED_EMPTY_SPEC, Operation speculatively executed, SIMD predicated with no active elements

The counter counts each speculatively executed predicated SIMD operation counted by SVE_PRED_NOT_FULL_SPEC where all elements are Inactive.

That is, all elements in the Governing predicate or predicates are FALSE.

When FEAT_SME is implemented, both SVE and SME operations with at least one Governing predicate operand are counted.

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

`0x8076`, SVE_PRED_FULL_SPEC, Operation speculatively executed, SIMD predicated with all active elements

The counter counts each speculatively executed predicated SIMD operation counted by SVE_PRED_SPEC where all elements are Active.

That is, all elements in the Governing predicate or predicates are all TRUE.

When FEAT_SME is implemented, both SVE and SME operations with at least one Governing predicate operand are counted.

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

`0x8077`, SVE_PRED_PARTIAL_SPEC, Operation speculatively executed, SIMD predicated with partially active elements

The counter counts each speculatively executed predicated SIMD operation counted by SVE_PRED_NOT_FULL_SPEC where neither all elements are Active nor all elements are Inactive.

That is, all elements in the Governing predicate or predicates are neither all TRUE nor all FALSE.

When FEAT_SME is implemented, both SVE and SME operations with at least one Governing predicate operand are counted.

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

`0x8078`, SVE_UNPRED_SPEC, Operation speculatively executed, SVE unpredicated

The counter counts each speculatively executed SIMD operation due to any of:

- An SVE instruction without a Governing predicate operand.
- An SME instruction without any Governing predicate operand.

If FEAT_SME is implemented, both operations due to SVE instructions and operations due to SME instructions with at least one Governing predicate operand that determines the Active elements are counted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8079, SVE_PRED_NOT_FULL_SPEC, Operation speculatively executed, SVE predicated with at least one inactive element

The counter counts each speculatively executed predicated SIMD data-processing, load, or store operation counted by SVE_PRED_SPEC where at least one element is Inactive.

That is, at least one element in the Governing predicate or predicates is FALSE.

When FEAT_SME is implemented, both SVE and SME operations with at least one Governing predicate operand are counted.

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

0x807C, SVE_MOVPRFX_SPEC, Operation speculatively executed, SVE MOVPRFX

The counter counts each speculatively executed operation due to any of the following instructions:

- SVE: MOVPRFX.

The instruction is counted whether or not it is fused with the prefixed instruction.

0x807D, SVE_MOVPRFX_Z_SPEC, Operation speculatively executed, SVE MOVPRFX zeroing predication

The counter counts each speculatively executed operation counted by SVE_MOVPRFX_SPEC where the operation uses zeroing predication.

0x807E, SVE_MOVPRFX_M_SPEC, Operation speculatively executed, SVE MOVPRFX merging predication

The counter counts each speculatively executed operation counted by SVE_MOVPRFX_SPEC where the operation uses merging predication.

0x807F, SVE_MOVPRFX_U_SPEC, Operation speculatively executed, SVE MOVPRFX unfused

The counter counts each speculatively executed operation counted by SVE_MOVPRFX_SPEC where the `MOVPRFX` is not fused with the prefixed instruction.

0x8080, SVE_LDST_SPEC, Operation speculatively executed, SVE load, store, or prefetch

The counter counts each speculatively executed operation that reads from or writes to memory counted by ASE_SVE_LDST_SPEC due to an SVE load, store, or prefetch instruction.

The following are classified as SVE load, store and prefetch instructions:

- Any load which reads from an SVE register.
- Any store which writes to an SVE register.
- Any of the following prefetch instructions which accept a Governing predicate:
  - `PRFB`.
  - `PRFD`.
  - `PRFH`.
  - `PRFW`.

When FEAT_SME is implemented, this includes SME loads and stores of the SVE Z vectors. SME loads and stores of the ZA and ZT registers are not counted.

0x8081, SVE_LD_SPEC, Operation speculatively executed, SVE load

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_SPEC due to an SVE load instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8082, SVE_ST_SPEC, Operation speculatively executed, SVE store

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_SPEC due to an SVE store instruction.

0x8083, SVE_PRF_SPEC, Operation speculatively executed, SVE prefetch

The counter counts each speculatively executed prefetch operation counted by SVE_LDST_SPEC due to any of the following instructions:

- SVE: PRFB, PRFD, PRFH, or PRFW.

0x8084, ASE_SVE_LDST_SPEC, Operation speculatively executed, Advanced SIMD or SVE load or store

The counter counts each speculatively executed operation that reads from or writes to memory due to an A64 Advanced SIMD or SVE load, store, or prefetch instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD load or store operations are counted in AArch32 state.

0x8085, ASE_SVE_LD_SPEC, Operation speculatively executed, Advanced SIMD or SVE load

The counter counts each speculatively executed operation that reads from memory counted by ASE_SVE_LDST_SPEC due to an A64 Advanced SIMD or SVE load instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD load operations are counted in AArch32 state.

0x8086, ASE_SVE_ST_SPEC, Operation speculatively executed, Advanced SIMD or SVE store

The counter counts each speculatively executed operation that writes to memory counted by ASE_SVE_LDST_SPEC due to an A64 Advanced SIMD or SVE store instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD store operations are counted in AArch32 state.

0x8087, PRF_SPEC, Operation speculatively executed, prefetch

The counter counts each speculatively executed prefetch operation due to any of the following A64 instructions:

- Scalar: PRFM, PRFUM, or RPRFM.
- SVE: PRFB, PRFD, PRFH, or PRFW.

It is IMPLEMENTATION DEFINED which prefetch operations are counted in AArch32 state.

0x8088, BASE_LDST_REG_SPEC, Operation speculatively executed, general-purpose register load, store, or prefetch

The counter counts each speculatively executed operation that reads from or writes to memory due to a general-purpose register load, store, or prefetch instruction.

It is IMPLEMENTATION DEFINED which load, store, and prefetch operations are counted in AArch32 state.

0x8089, BASE_LD_REG_SPEC, Operation speculatively executed, general-purpose register load

The counter counts each speculatively executed operation that reads from memory counted by BASE_LDST_REG_SPEC due to a general-purpose register load instruction.

It is IMPLEMENTATION DEFINED which load operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x808A, BASE_ST_REG_SPEC, Operation speculatively executed, general-purpose register store

The counter counts each speculatively executed operation that writes to memory counted by BASE_LDST_REG_SPEC due to a general-purpose register store instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to `DC ZVA` instructions.

It is IMPLEMENTATION DEFINED which store operations are counted in AArch32 state.

0x808B, BASE_PRF_SPEC, Operation speculatively executed, general-purpose register prefetch

The counter counts each speculatively executed prefetch operation counted by BASE_LDST_REG_SPEC due to any of the following A64 instructions:

- Scalar: PRFM, PRFUM, or RPRFM.

It is IMPLEMENTATION DEFINED which prefetch operations are counted in AArch32 state.

0x808C, FPASE_LDST_REG_SPEC, Operation speculatively executed, SIMD&FP register load or store

The counter counts each speculatively executed operation that reads from or writes to memory due to any of the following A64 instructions:

- Scalar: LDP (SIMD&FP), LDR (SIMD&FP), STP (SIMD&FP), or STR (SIMD&FP).
- Advanced SIMD: LD1, LD1R, LDNP, LDP, LDR, LDUR, ST1, STNP, STP, STR, or STUR.
- When FEAT_LRCPC3 is implemented, Advanced SIMD: LDAP1, LDAPUR, STL1, or STLUR.

It is IMPLEMENTATION DEFINED which load and store operations are counted in AArch32 state.

0x808D, FPASE_LD_REG_SPEC, Operation speculatively executed, SIMD&FP register load

The counter counts each speculatively executed operation that reads from memory counted by FPASE_LDST_REG_SPEC due to any of the following A64 instructions:

- Scalar: LDP (SIMD&FP) or LDR (SIMD&FP).
- Advanced SIMD: LD1, LD1R, LDNP, LDP, LDR, or LDUR.
- When FEAT_LRCPC3 is implemented, Advanced SIMD: LDAP1 or LDAPUR.

It is IMPLEMENTATION DEFINED which load operations are counted in AArch32 state.

0x808E, FPASE_ST_REG_SPEC, Operation speculatively executed, SIMD&FP register store

The counter counts each speculatively executed operation that writes to memory counted by FPASE_LDST_REG_SPEC due to any of the following A64 instructions:

- Scalar: STP (SIMD&FP) or STR (SIMD&FP).
- Advanced SIMD: ST1, STNP, STP, STR, or STUR.
- When FEAT_LRCPC3 is implemented, Advanced SIMD: STL1 or STLUR.

It is IMPLEMENTATION DEFINED which store operations are counted in AArch32 state.

0x8090, SVE_LDST_REG_SPEC, Operation speculatively executed, SVE unpredicated load or store register

The counter counts each speculatively executed SVE register load or store operation due to any of the following instructions:

- SVE: LDR or STR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8091, SVE_LDR_REG_SPEC, Operation speculatively executed, SVE unpredicated load register

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_REG_SPEC due to any of the following instructions:

- SVE: LDR.

0x8092, SVE_STR_REG_SPEC, Operation speculatively executed, SVE unpredicated store register

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_REG_SPEC due to any of the following instructions:

- SVE: STR.

0x8094, SVE_LDST_PREG_SPEC, Operation speculatively executed, SVE load or store predicate register

The counter counts each speculatively executed SVE predicate register load or store operation due to any of the following instructions:

- SVE: LDR (predicate) or STR (predicate).

0x8095, SVE_LDR_PREG_SPEC, Operation speculatively executed, SVE load predicate register

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_PREG_SPEC due to any of the following instructions:

- SVE: LDR (predicate).

0x8096, SVE_STR_PREG_SPEC, Operation speculatively executed, SVE store predicate register

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_PREG_SPEC due to any of the following instructions:

- SVE: STR (predicate).

0x8098, SVE_LDST_ZREG_SPEC, Operation speculatively executed, SVE load or store vector register

The counter counts each speculatively executed SVE vector register load or store operation due to any of the following instructions:

- SVE: LDR (vector) or STR (vector).

0x8099, SVE_LDR_ZREG_SPEC, Operation speculatively executed, SVE load vector register

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_ZREG_SPEC due to any of the following instructions:

- SVE: LDR (vector).

0x809A, SVE_STR_ZREG_SPEC, Operation speculatively executed, SVE store vector register

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_ZREG_SPEC due to any of the following instructions:

- SVE: STR (vector).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x809C, SVE_LDST_CONTIG_SPEC, Operation speculatively executed, SVE contiguous load, store, or prefetch element

The counter counts each speculatively executed SVE predictated single vector contiguous element load, store, or prefetch operation due to any of:

- A predicated single vector contiguous load or store instruction operating on the SVE registers.
- An SVE load and replicate `LD1R` or `LD1RQ` instruction.

When FEAT_SME is implemented, this includes the following instructions:

- SME: `LD1B`, `LD1D`, `LD1H`, `LD1Q`, `LD1W`, `LDNT1B`, `LDNT1D`, `LDNT1H`, `LDNT1Q`, `ST1B`, `ST1D`, `ST1H`, `ST1Q`, `ST1W`, `STNT1B`, `STNT1D`, or `STNT1H`.

0x809D, SVE_LD_CONTIG_SPEC, Operation speculatively executed, SVE single vector contiguous load element

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_CONTIG_SPEC due to any of:

- A predicated single vector contiguous load instruction operating on the SVE registers.
- An SVE load and replicate `LD1R` or `LD1RQ` instruction.

When FEAT_SME is implemented, this includes the following instructions:

- SME: `LD1B`, `LD1D`, `LD1H`, `LD1Q`, `LD1W`, `LDNT1B`, `LDNT1D`, or `LDNT1H`.

0x809E, SVE_ST_CONTIG_SPEC, Operation speculatively executed, SVE contiguous store element

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_CONTIG_SPEC due to any of:

- A predicated single vector contiguous store instruction operating on the SVE registers.

When FEAT_SME is implemented, this includes the following instructions:

- SME: `ST1B`, `ST1D`, `ST1H`, `ST1Q`, `ST1W`, `STNT1B`, `STNT1D`, `STNT1H`, and `STNT1Q`.

0x809F, SVE_PRF_CONTIG_SPEC, Operation speculatively executed, SVE contiguous prefetch element

The counter counts each speculatively executed operation counted by SVE_LDST_CONTIG_SPEC due to an SVE predicated single contiguous element prefetch instruction.

0x80A0, SVE_LDSTNT_CONTIG_SPEC, Operation speculatively executed, SVE non-temporal contiguous load or store element

The counter counts each speculatively executed SVE non-temporal contiguous element load or store operation.

0x80A1, SVE_LDNT_CONTIG_SPEC, Operation speculatively executed, SVE non-temporal contiguous load element

The counter counts each speculatively executed operation that reads from memory with a non-temporal hint counted by SVE_LDSTNT_CONTIG_SPEC due to an SVE non-temporal contiguous element load instruction.

When FEAT_SME is implemented, this includes operations due to SME non-temporal contiguous element load instructions operating on the SVE registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80A2, SVE_STNT_CONTIG_SPEC, Operation speculatively executed, SVE non-temporal contiguous store element

The counter counts each speculatively executed operation that writes to memory with a non-temporal hint counted by SVE_LDSTNT_CONTIG_SPEC due to an SVE non-temporal contiguous element store instruction.

When FEAT_SME is implemented, this includes operations due to SME non-temporal contiguous element store instructions operating on the SVE registers.

0x80A4, ASE_SVE_LDST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector

The counter counts each speculatively executed operation due to an A64 Advanced SIMD or SVE multiple vector contiguous structure load or store instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure load or store operations are counted in AArch32 state.

0x80A5, ASE_SVE_LD_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector

The counter counts each speculatively executed operation that reads from memory counted by ASE_SVE_LDST_MULTI_SPEC due to an A64 Advanced SIMD or SVE multiple vector contiguous structure load instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure load operations are counted in AArch32 state.

0x80A6, ASE_SVE_ST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector

The counter counts each speculatively executed operation that writes to memory counted by ASE_SVE_LDST_MULTI_SPEC due to an A64 Advanced SIMD or SVE multiple vector contiguous structure store instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure store operations are counted in AArch32 state.

0x80A8, SVE_LDST_MULTI_SPEC, Operation speculatively executed, SVE contiguous load or store multiple vector

The counter counts each speculatively executed operation that reads from or writes to memory due to an SVE multiple vector contiguous structure load or store instruction.

0x80A9, SVE_LD_MULTI_SPEC, Operation speculatively executed, SVE contiguous load multiple vector

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_MULTI_SPEC due to an SVE multiple vector contiguous structure load instruction.

When FEAT_SME is implemented, this includes operations due to SME multiple vector contiguous load instructions operating on the SVE registers.

0x80AA, SVE_ST_MULTI_SPEC, Operation speculatively executed, SVE contiguous store multiple vector

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_MULTI_SPEC due to an SVE multiple vector contiguous structure store instruction.

When FEAT_SME is implemented, this includes operations due to SME multiple vector contiguous store instructions operating on the SVE registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80AC, SVE_LDST_NONCONTIG_SPEC, Operation speculatively executed, SVE non-contiguous load, store, or prefetch

The counter counts each speculatively executed operation that reads from or writes to memory due to an SVE non-contiguous gather-load, scatter-store, or gather-prefetch instruction.

0x80AD, SVE_LD_GATHER_SPEC, Operation speculatively executed, SVE gather-load

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST_NONCONTIG_SPEC due to an SVE non-contiguous gather-load instruction.

0x80AE, SVE_ST_SCATTER_SPEC, Operation speculatively executed, SVE scatter-store

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST_NONCONTIG_SPEC due to an SVE non-contiguous scatter-store instruction.

0x80AF, SVE_PRF_GATHER_SPEC, Operation speculatively executed, SVE gather-prefetch

The counter counts each speculatively executed prefetch operation counted by SVE_LDST_NONCONTIG_SPEC due to an SVE non-contiguous gather-prefetch instruction.

0x80B0, SVE_LDST64_NONCONTIG_SPEC, Operation speculatively executed, SVE 64-bit non-contiguous load, store, or prefetch

The counter counts each speculatively executed operation that reads from or writes to memory due to an SVE non-continguous gather-load, scatter-store, or gather-prefetch operation with 64-bit vector elements in the address.

0x80B1, SVE_LD64_GATHER_SPEC, Operation speculatively executed, SVE 64-bit gather-load

The counter counts each speculatively executed operation that reads from memory counted by SVE_LDST64_NONCONTIG_SPEC due to an SVE non-contiguous gather-load instruction with 64-bit vector elements in the address.

0x80B2, SVE_ST64_SCATTER_SPEC, Operation speculatively executed, SVE 64-bit scatter-store

The counter counts each speculatively executed operation that writes to memory counted by SVE_LDST64_NONCONTIG_SPEC due to an SVE non-contiguous scatter-store instruction with 64-bit vector elements in the address.

0x80B3, SVE_PRF64_GATHER_SPEC, Operation speculatively executed, SVE 64-bit gather-prefetch

The counter counts each speculatively executed prefetch operation counted by SVE_LDST64_NONCONTIG_SPEC due to an SVE non-contiguous gather-prefetch instruction with 64-bit vector elements in the address.

0x80B4, ASE_SVE_UNALIGNED_LDST_SPEC, Advanced SIMD or SVE unaligned access

The counter counts each speculatively executed A64 Advanced SIMD or SVE load or store instruction where either:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather, scatter, or single element address is not aligned to the memory element access size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD load or store operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80B5, ASE_SVE_UNALIGNED_LD_SPEC, Advanced SIMD or SVE unaligned read

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_LDST_SPEC due to an A64 Advanced SIMD or SVE load instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD load operations are counted in AArch32 state.

0x80B6, ASE_SVE_UNALIGNED_ST_SPEC, Advanced SIMD or SVE unaligned write

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_LDST_SPEC due to an A64 Advanced SIMD or SVE store instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD store operations are counted in AArch32 state.

0x80B8, ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC, Advanced SIMD or SVE unaligned contiguous access

The counter counts each speculatively executed Advanced SIMD or SVE contiguous load or store operation where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD load or store operations are counted in AArch32 state.

0x80B9, ASE_SVE_UNALIGNED_CONTIG_LD_SPEC, Advanced SIMD or SVE unaligned contiguous read

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC due to an A64 Advanced SIMD or SVE load instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD load operations are counted in AArch32 state.

0x80BA, ASE_SVE_UNALIGNED_CONTIG_ST_SPEC, Advanced SIMD or SVE unaligned contiguous write

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC due to a A64 Advanced SIMD or SVE store instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD store operations are counted in AArch32 state.

0x80BC, SVE_LDFF_SPEC, Operation speculatively executed, SVE first-fault load

The counter counts each speculatively executed Memory-read operation counted by SVE_LD_SPEC due to an SVE First-fault or Non-fault load instruction.

0x80BD, SVE_LDFF_FAULT_SPEC, Operation speculatively executed, SVE first-fault load which set FFR bit to 0

The counter counts each speculatively executed Memory-read operation counted by SVE_LDFF_SPEC due to an SVE First-fault or Non-fault load instruction that writes 0 to at least one bit in FFR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80C0, FP_SCALE_OPS_SPEC, Scalable element arithmetic operations speculatively executed, floating-point

The counter increments by *v* for each speculatively executed scalable element arithmetic operation, due to an instruction where the type was floating-point.

Where *v* is a value such that ($v \times (VL \div 128)$) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C1, FP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the type was floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C2, FP_HP_SCALE_OPS_SPEC, Scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point

The counter increments by *v* for each speculatively executed scalable element arithmetic operation, due to an instruction where the largest type was half-precision floating-point.

Where *v* is a value such that ($v \times (VL \div 128)$) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C3, FP_HP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was half-precision floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C4, FP_SP_SCALE_OPS_SPEC, Scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point

The counter increments by *v* for each speculatively executed scalable element arithmetic operation, due to an instruction where the largest type was single-precision floating-point.

Where *v* is a value such that ($v \times (VL \div 128)$) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C5, FP_SP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was single-precision floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80C6, FP_DP_SCALE_OPS_SPEC, Scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point

The counter increments by v for each speculatively executed scalable element arithmetic operation, due to an instruction where the largest type was double-precision floating-point.

Where v is a value such that $(v \times (VL \div 128))$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C7, FP_DP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was double-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C8, INT_SCALE_OPS_SPEC, Scalable element arithmetic operations speculatively executed, integer

The counter increments by v for each speculatively executed scalable element arithmetic operation, due to an instruction where the type was integer.

Where v is a value such that $(v \times (VL \div 128))$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80C9, INT_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, integer

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the type was integer.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80CA, LDST_SCALE_OPS_SPEC, Scalable load or store element operations speculatively executed

The counter counts each speculatively executed Memory-read operation or Memory-write operation due to any of:

- An SVE vector load or store instruction other than a load and replicate instruction.
- An SME vector load or store instruction.

SVE non-vector load or store instructions and SVE vector load and replicate instructions are counted by the LDST_FIXED_OPS_SPEC event.

See SIMD SVE and SME instructions for information on the counter increment for different types of instruction.

The counter does not count tag loads or tag stores.

0x80CB, LDST_FIXED_OPS_SPEC, Non-scalable load or store element operations speculatively executed

The counter counts each speculatively executed Memory-read operation or Memory-write operation due to any of:

- Any load, store, or atomic operation, other than loads and stores of the SVE P and Z registers, and the SME ZA registers.
- Any SVE non-vector load or store operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An SVE load and replicate instruction, including: `LD1R`, `LD1RB`, `LD1RD`, `LD1RH`, `LD1RO`, `LD1ROB`, `LD1ROD`, `LD1ROH`, `LD1ROW`, `LD1RQ`, `LD1RQB`, `LD1RQD`, `LD1RQH`, `LD1RQW`, `LD1RSB`, `LD1RSH`, or `LD1RSW`.

For each instruction, the counter is incremented by the number of operations specified by the instruction. For example, the counter counts operations as follows:

- Load and store of a single register instructions, other than SVE and SME vector loads and stores, increment the counter by 1.
- Load and store of a pair of registers instructions, other than SVE and SME vector loads and stores, increment the counter by 2.
- AArch32 load and store multiple registers instructions increment the counter by the number of registers transferred.
- Atomic store instructions increment the counter by 1. These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by 2. Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.
- Compare and swap of a pair of registers increment the counter by 4.
- SVE and Advanced SIMD `LD1R` instructions increment the counter by 1.
- SVE `LD1RQ` instructions increment the counter by (128 ÷ CSIZE).
- SVE `LD1RO` instructions increment the counter by (256 ÷ CSIZE).
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.
- `DC ZVA` and `DC GZVA` instructions increment by an IMPLEMENTATION DEFINED amount.
- `LDR` (table) and `STR` (table) instructions increment the counter by 1.
- `LD64B` and `ST64B` increment the counter by 8.

Note

When a replicating load instruction loads data of the same size as the vector length, which causes no replication to occur, the instruction is counted.

`0x80CC`, LD_SCALE_OPS_SPEC, Scalable load element operations speculatively executed

The counter counts each speculatively executed Memory-read operation counted by LDST_SCALE_OPS_SPEC due to any of:

- An SVE vector load instruction other than a replicating `LD1R`, `LD1RQ`, or `LD1RO` instruction.
- An SME vector load instruction.

See SIMD SVE and SME instructions for information on the counter increment for different types of instruction.

The counter does not count tag loads.

`0x80CD`, LD_FIXED_OPS_SPEC, Non-scalable load element operations speculatively executed

The counter counts each speculatively executed Memory-read operation counted by LDST_FIXED_OPS_SPEC.

For each instruction, the counter is incremented by the number of operations specified by the instruction. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_OPS_SPEC event counts if the operation is a load atomic, compare and swap, or swap operation.
- The same as for LDST_FIXED_OPS_SPEC if the operation is any other load operation.

`0x80CE`, ST_SCALE_OPS_SPEC, Scalable store element operations speculatively executed

The counter counts each speculatively executed Memory-write operation counted by LDST_SCALE_OPS_SPEC due to any of:

- An SVE predicated vector store instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An SME vector store instruction.

See SIMD SVE and SME instructions for information on the counter increment for different types of instruction.

The counter does not count tag stores.

`0x80CF`, ST_FIXED_OPS_SPEC, Non-scalable store element operations speculatively executed

The counter counts each Memory-write operation counted by LDST_FIXED_OPS_SPEC.

For each instruction, the counter is incremented by the number of operations specified by the instruction. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_OPS_SPEC event counts if the operation is a store atomic, compare and swap, or swap operation.
- The same as for LDST_FIXED_OPS_SPEC if the operation is any other store operation.

`0x80D0`, FP_SCALE2_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the type was floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

`0x80D2`, FP_HP_SCALE2_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, largest type is half-precision floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the largest type was half-precision floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

`0x80D3`, FP_BF16_SCALE2_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, largest type is BFloat16 floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the largest type was BFloat16 floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

`0x80D4`, FP_SP_SCALE2_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, largest type is single-precision floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the largest type was single-precision floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80D6, FP_DP_SCALE2_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, largest type is double-precision floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the largest type was double-precision floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

0x80D8, INT_SCALE2_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, integer

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the type was integer.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

0x80DA, LDST_SCALE_BYTES_SPEC, Scalable load and store bytes speculatively executed

The counter counts each byte speculatively read or written due to any of:

- An SVE vector load or store instruction other than a load and replicate instruction.
- An SME vector load or store instruction.

For each instruction, the counter is incremented by (16 ÷ (CSIZE ÷ MSIZE)), multiplied by the number of transferred vector registers.

SVE non-vector load or store instructions and SVE vector load and replicate instructions are counted by the LDST_FIXED_BYTES_SPEC event.

0x80DB, LDST_FIXED_BYTES_SPEC, Non-scalable load and store bytes speculatively executed

The counter counts each byte speculatively read or written due to any of:

- Any load, store, or atomic operation, other than loads and stores of the SVE P and Z registers, and the SME ZA registers.
- Any SVE non-vector load or store operation.
- An SVE load and replicate instruction, including: `LD1R`, `LD1RB`, `LD1RD`, `LD1RH`, `LD1RO`, `LD1ROB`, `LD1ROD`, `LD1ROH`, `LD1ROW`, `LD1RQ`, `LD1RQB`, `LD1RQD`, `LD1RQH`, `LD1RQW`, `LD1RSB`, `LD1RSH`, or `LD1RSW`.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred multiplied by the number of transfers made per register. For example, the counter counts bytes as follows:

- Load and store of a single register instructions, other than SVE and SME vector loads and stores, increment the counter by (MSIZE ÷ 8).
- Load and store of a pair of registers instructions, other than SVE and SME vector loads and stores, increment the counter by 2 × (MSIZE ÷ 8).
- AArch32 load and store multiple registers instructions increment the counter by the number of registers transferred multiplied by (MSIZE ÷ 8).
- Atomic store instructions increment the counter by (MSIZE ÷ 8). These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by 2 × (MSIZE ÷ 8). Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.
- Compare and swap of a pair of registers increment the counter by 4 × (MSIZE ÷ 8).
- SVE and Advanced SIMD `LD1R` instructions increment the counter by (MSIZE ÷ 8).
- SVE `LD1RQ` instructions increment the counter by 16.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SVE `LD1RO` instructions increment the counter by 32.
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of registers transferred multiplied by the number of bytes being transferred per register.
- `DC ZVA` and `DC GZVA` instructions increment by the counter by $2^{(DCZID_EL0.BS)}$.
- `LDR` (table) and `STR` (table) instructions increment the counter by 64.
- `LD64B` and `ST64B` increment the counter by 64.

`0x80DC`, LD_SCALE_BYTES_SPEC, Scalable load bytes speculatively executed

The counter counts each speculatively read byte due to any of:

- An SVE vector load instruction other than a replicating `LD1R`, `LD1RQ`, or `LD1RO` instruction.
- An SME vector load instruction.

For each instruction, the counter is incremented by (16 ÷ (CSIZE ÷ MSIZE)), multiplied by the number of transferred vector registers.

`0x80DD`, LD_FIXED_BYTES_SPEC, Non-scalable load bytes speculatively executed

The counter counts each speculatively read byte counted by LDST_FIXED_BYTES_SPEC.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_BYTES_SPEC event counts if the operation is a load atomic, compare and swap, or swap operation.
- The same as for LDST_FIXED_BYTES_SPEC if the operation is any other load operation.

`0x80DE`, ST_SCALE_BYTES_SPEC, Scalable store bytes speculatively executed

The counter counts each speculatively written byte due to any of:

- An SVE vector store instruction.
- An SME vector store instruction.

For each instruction, the counter is incremented by (16 ÷ (CSIZE ÷ MSIZE)), multiplied by the number of transferred vector registers.

`0x80DF`, ST_FIXED_BYTES_SPEC, Non-scalable store bytes speculatively executed

The counter counts each speculatively written byte counted by LDST_FIXED_BYTES_SPEC.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_BYTES_SPEC event counts if the operation is a swap, or compare and swap operation.
- The same as for LDST_FIXED_BYTES_SPEC if the operation is any other store operation, including an atomic store operation.

`0x80E1`, ASE_INT8_SPEC, Integer operation speculatively executed, Advanced SIMD 8-bit

The counter counts each operation counted by ASE_SVE_INT8_SPEC that is an Advanced SIMD operation.

`0x80E2`, SVE_INT8_SPEC, Integer operation speculatively executed, SVE 8-bit

The counter counts each operation counted by ASE_SVE_INT8_SPEC that is an SVE operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80E3, ASE_SVE_INT8_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 8-bit

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 8-bit integer.

0x80E5, ASE_INT16_SPEC, Integer operation speculatively executed, Advanced SIMD 16-bit

The counter counts each operation counted by ASE_SVE_INT16_SPEC that is an Advanced SIMD operation.

0x80E6, SVE_INT16_SPEC, Integer operation speculatively executed, SVE 16-bit

The counter counts each operation counted by ASE_SVE_INT16_SPEC that is an SVE operation.

0x80E7, ASE_SVE_INT16_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 16-bit

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 16-bit integer.

0x80E8, FP_BF16_SCALE_OPS_SPEC, Scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point

The counter increments by v for each speculatively executed scalable element arithmetic operation, due to an instruction where the largest type was BFloat16 floating-point.

Where v is a value such that $(v \times (VL \div 128))$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x80E9, ASE_INT32_SPEC, Integer operation speculatively executed, Advanced SIMD 32-bit

The counter counts each operation counted by ASE_SVE_INT32_SPEC that is an Advanced SIMD operation.

0x80EA, SVE_INT32_SPEC, Integer operation speculatively executed, SVE 32-bit

The counter counts each operation counted by ASE_SVE_INT32_SPEC that is an SVE operation.

0x80EB, ASE_SVE_INT32_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 32-bit

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 32-bit integer.

0x80ED, ASE_INT64_SPEC, Integer operation speculatively executed, Advanced SIMD 64-bit

The counter counts each operation counted by ASE_SVE_INT64_SPEC that is an Advanced SIMD operation.

0x80EE, SVE_INT64_SPEC, Integer operation speculatively executed, SVE 64-bit

The counter counts each operation counted by ASE_SVE_INT64_SPEC that is an SVE operation.

0x80EF, ASE_SVE_INT64_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 64-bit

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 64-bit integer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80F1, ASE_FP_DOT_SPEC, Floating-point operation speculatively executed, Advanced SIMD dot-product

The counter counts each dot-product operation counted by ASE_SVE_FP_DOT_SPEC due to any of the following A64 instructions:

- When FEAT_BF16 is implemented, Advanced SIMD: BFDOT.
- When FEAT_FP8DOT2 is implemented, Advanced SIMD: FDOT (2-way, by element) or FDOT (2-way, vector).
- When FEAT_FP8DOT4 is implemented, Advanced SIMD: FDOT (4-way, by element) or FDOT (4-way, vector).

It is IMPLEMENTATION DEFINED which dot-product operations are counted in AArch32 state.

0x80F2, SVE_FP_DOT_SPEC, Floating-point operation speculatively executed, SVE dot-product

The counter counts each dot-product operation counted by ASE_SVE_FP_DOT_SPEC due to any of the following instructions:

- SVE: BFDOT.
- SVE2: FDOT.

0x80F3, ASE_SVE_FP_DOT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product

The counter counts each dot-product operation counted by FP_SPEC due to any of the following A64 instructions:

- When FEAT_BF16 is implemented, Advanced SIMD: BFDOT.
- When FEAT_FP8DOT2 is implemented, Advanced SIMD: FDOT (2-way, by element) or FDOT (2-way, vector).
- When FEAT_FP8DOT4 is implemented, Advanced SIMD: FDOT (4-way, by element) or FDOT (4-way, vector).
- SVE: BFDOT.
- SVE2: FDOT.

It is IMPLEMENTATION DEFINED which dot-product operations are counted in AArch32 state.

0x80F5, ASE_FP_MMLA_SPEC, Floating-point operation speculatively executed, Advanced SIMD matrix multiply

The counter counts each floating-point matrix multiply operation counted by ASE_SVE_FP_MMLA_SPEC due to any of the following A64 instructions:

- When FEAT_BF16 is implemented, Advanced SIMD: BFMMLA.
- When FEAT_F8F16MM is implemented, Advanced SIMD: FMMLA (8-bit floating-point to half-precision) or FMMLA (8-bit floating-point to single-precision).

It is IMPLEMENTATION DEFINED which floating-point matrix multiply operations are counted in AArch32 state.

0x80F6, SVE_FP_MMLA_SPEC, Floating-point operation speculatively executed, SVE matrix multiply

The counter counts each floating-point matrix multiply operation counted by ASE_SVE_FP_MMLA_SPEC due to any of the following instructions:

- SVE: BFMMLA (widening) or FMMLA (non-widening).
- SVE2: FMMLA.

0x80F7, ASE_SVE_FP_MMLA_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply

The counter counts each floating-point matrix multiply operation counted by FP_SPEC due to any of the following A64 instructions:

- When FEAT_BF16 is implemented, Advanced SIMD: BFMMLA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_F8F16MM is implemented, Advanced SIMD: FMMLA (8-bit floating-point to half-precision) or FMMLA (8-bit floating-point to single-precision).
- SVE: BFMMLA (widening) or FMMLA (non-widening).
- SVE2: FMMLA.

It is IMPLEMENTATION DEFINED which floating-point matrix multiply operations are counted in AArch32 state.

`0x80F9`, ASE_INT_DOT_SPEC, Operation speculatively executed, Advanced SIMD integer dot-product

The counter counts each integer dot product operation counted by ASE_SVE_INT_DOT_SPEC due to any of the following A64 instructions:

- When FEAT_DotProd is implemented, Advanced SIMD: SDOT or UDOT.
- When FEAT_I8MM is implemented, Advanced SIMD: SUDOT (by element) or USDOT.

It is IMPLEMENTATION DEFINED which integer dot product operations are counted in AArch32 state.

`0x80FA`, SVE_INT_DOT_SPEC, Integer operation speculatively executed, SVE dot-product

The counter counts each integer dot product operation counted by ASE_SVE_INT_DOT_SPEC due to any of the following instructions:

- SVE: SDOT, SUDOT, UDOT, or USDOT.
- SVE2: CDOT, SDOT, or UDOT.

`0x80FB`, ASE_SVE_INT_DOT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE dot-product

The counter counts each integer dot product operation counted by INT_SPEC due to any of the following A64 instructions:

- When FEAT_DotProd is implemented, Advanced SIMD: SDOT or UDOT.
- When FEAT_I8MM is implemented, Advanced SIMD: SUDOT (by element) or USDOT.
- SVE: SDOT, SUDOT, UDOT, or USDOT.
- SVE2: CDOT, SDOT, or UDOT.

It is IMPLEMENTATION DEFINED which integer dot product operations are counted in AArch32 state.

`0x80FD`, ASE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD matrix multiply

The counter counts each integer matrix multiply operation counted by ASE_SVE_INT_MMLA_SPEC due to any of the following A64 instructions:

- When FEAT_I8MM is implemented, Advanced SIMD: SMMLA (vector), UMMLA (vector), or USMMLA (vector).

It is IMPLEMENTATION DEFINED which integer matrix multiply operations are counted in AArch32 state.

`0x80FE`, SVE_INT_MMLA_SPEC, Integer operation speculatively executed, SVE matrix multiply

The counter counts each integer matrix multiply operation counted by ASE_SVE_INT_MMLA_SPEC due to any of the following instructions:

- SVE: SMMLA, UMMLA, or USMMLA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x80FF, ASE_SVE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply

The counter counts each integer matrix multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- When FEAT_I8MM is implemented, Advanced SIMD: SMMLA (vector), UMMLA (vector), or USMMLA (vector).
- SVE: SMMLA, UMMLA, or USMMLA.

It is IMPLEMENTATION DEFINED which integer matrix multiply operations are counted in AArch32 state.

0x8110, BR_IMMED_PRED_RETIRED, Branch instruction architecturally executed, predicted immediate

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_PRED_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is correctly predicted.

0x8111, BR_IMMED_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted immediate

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_MIS_PRED_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is mispredicted.

0x8112, BR_IND_PRED_RETIRED, Branch instruction architecturally executed, predicted indirect

The counter counts each instruction counted by both BR_IND_RETIRED and BR_PRED_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is correctly predicted.

0x8113, BR_IND_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted indirect

The counter counts each instruction counted by both BR_IND_RETIRED and BR_MIS_PRED_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is mispredicted.

0x8114, BR_RETURN_PRED_RETIRED, Branch instruction architecturally executed, predicted procedure return

The counter counts each instruction counted by BR_IND_PRED_RETIRED where if taken, the branch would be counted by BR_RETURN_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is correctly predicted.

0x8115, BR_RETURN_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted procedure return

The counter counts each instruction counted by BR_IND_MIS_PRED_RETIRED where if taken, the branch would also be counted by BR_RETURN_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is mispredicted.

0x8116, BR_INDNR_PRED_RETIRED, Branch instruction architecturally executed, predicted indirect excluding procedure return

The counter counts each instruction counted by BR_IND_PRED_RETIRED where if taken, the branch would not be counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is correctly predicted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8117, BR_INDNR_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted indirect excluding procedure return

The counter counts each instruction counted by BR_IND_MIS_PRED_RETIRED where if taken, the branch would not be counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is mispredicted.

0x8118, BR_TAKEN_PRED_RETIRED, Branch instruction architecturally executed, predicted branch, taken

The counter counts each instruction counted by both BR_PRED_RETIRED and PC_WRITE_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and taken.

0x8119, BR_TAKEN_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted branch, taken

The counter counts each instruction counted by both BR_MIS_PRED_RETIRED and PC_WRITE_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and taken.

0x811A, BR_SKIP_PRED_RETIRED, Branch instruction architecturally executed, predicted branch, not taken

The counter counts each instruction counted by both BR_PRED_RETIRED and BR_SKIP_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and not taken.

0x811B, BR_SKIP_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted branch, not taken

The counter counts each instruction counted by both BR_MIS_PRED_RETIRED and BR_SKIP_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and not taken.

0x811C, BR_PRED_RETIRED, Branch instruction architecturally executed, predicted branch

The counter counts each instruction counted by BR_RETIRED that is not counted by BR_MIS_PRED_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted.

0x8120, INST_FETCH_PERCYC, Instruction fetches in progress

The counter increments by the number of instruction fetches counted by INST_FETCH in progress on each Processor cycle.

The ratio INST_FETCH_PERCYC ÷ INST_FETCH is the mean duration of instruction fetches in Processor cycles.

0x8121, MEM_ACCESS_RD_PERCYC, Data memory reads in progress

The counter increments by the number of data memory reads counted by MEM_ACCESS_RD in progress on each Processor cycle.

The ratio MEM_ACCESS_RD_PERCYC ÷ MEM_ACCESS_RD is the mean duration of data memory reads in Processor cycles.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8122, SAMPLE_FEED_DS, Statistical Profiling sample taken, selected Data Source

The counter counts each SPE sample that is a load operations where PMSDSFR_EL1[s] is 1 and s is bits [5:0] of the sampled Data Source.

The values of PMSFCR_EL1.FDS are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

0x8123, SAMPLE_BUFFER_FULL, Profiling Buffer full

The counter counts each Profiling Buffer full management event.

0x8124, INST_FETCH, Instruction memory access

The counter counts each Instruction memory access.

The counter increments whether the access results in an access to a Level 1 instruction cache, a Level 2 instruction, data or unified cache, or none of these.

The counter does not increment as a result of:

- Data memory accesses.
- Translation table walks.
- Refilling of any cache.
- Accesses that result from cache maintenance instructions.

0x8125, BUS_REQ_RD_PERCYC, Bus read transactions in progress

The counter increments by the number of bus read transactions counted by BUS_REQ_RD in progress on each Processor cycle.

The ratio BUS_REQ_RD_PERCYC ÷ BUS_REQ_RD is the mean duration of bus read transactions in Processor cycles.

A bus read transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

If the bus is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8126, BUS_REQ_WR_PERCYC, Bus write transactions in progress

The counter increments by the number of bus write transactions counted by BUS_REQ_WR in progress on each Processor cycle.

The ratio BUS_REQ_WR_PERCYC ÷ BUS_REQ_WR is the mean duration of bus write transactions in Processor cycles.

A bus write transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

A mean duration value calculated using this event only measures the latency from the perspective of the bus. This is not necessarily the same as the latency for the write to be Complete, as the write might be completed on the bus before reaching its endpoint.

For some bus implementations, writes are posted, meaning that, from the perspective of the bus, the write completes immediately. There is no response when the write reaches its Completion endpoint. This event is not included for such implementations.

If the bus is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8127`, PMU_SNAPSHOT, Successful PMU capture event

The counter counts each PMU snapshot Capture event that was successful, that is, PMSSCR_EL1.NC is set to 0.

It is CONSTRAINED UNPREDICTABLE whether the counter counts successful Capture events when the PE is in Debug state.

`0x8128`, DTLB_WALK_PERCYC, Data translation table walks in progress

The counter increments by the number of data translation table walks counted by DTLB_WALK in progress on each Processor cycle.

The ratio DTLB_WALK_PERCYC ÷ DTLB_WALK is the mean duration of data translation table walks in Processor cycles.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8129`, ITLB_WALK_PERCYC, Instruction translation table walks in progress

The counter increments by the number of instruction translation table walks counted by ITLB_WALK in progress on each Processor cycle.

The ratio ITLB_WALK_PERCYC ÷ ITLB_WALK is the mean duration of instruction translation table walks in Processor cycles.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x812A`, SAMPLE_FEED_BR, Statistical Profiling sample taken, branch

The counter counts each sample counted by SAMPLE_FEED that is a branch operation.

The values of PMSFCR_EL1.{B,FT} and, if FEAT_SPE_EFT is implemented, PMSFCR_EL1.Bm, are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

`0x812B`, SAMPLE_FEED_LD, Statistical Profiling sample taken, load

The counter counts each sample counted by SAMPLE_FEED that is a load or load atomic operation.

The values of PMSFCR_EL1.{LD,FT} and, if FEAT_SPE_EFT is implemented, PMSFCR_EL1.LDm, are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x812C, SAMPLE_FEED_ST, Statistical Profiling sample taken, store

The counter counts each sample counted by SAMPLE_FEED that is a store or atomic operation, including load atomic operations.

The values of PMSFCR_EL1.{ST,FT} and, if FEAT_SPE_EFT is implemented, PMSFCR_EL1.STm, are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

0x812D, SAMPLE_FEED_OP, Statistical Profiling sample taken, matching type

The counter counts each sample counted by SAMPLE_FEED that meets the sample type filter constraints.

The sample type filter constraints are specified by PMSFCR_EL1.TYPE and, if FEAT_SPE_EFT is implemented, PMSFCR_EL1.TYPEm.

The value of PMSFCR_EL1.FT is ignored when generating this event.

No event is generated if the type filter constraint controls are all zero.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

0x812E, SAMPLE_FEED_EVENT, Statistical Profiling sample taken, matching events

The counter counts each sample counted by SAMPLE_FEED that meets the Events packet filter constraints.

That is, each sample with all the events in the filter sets defined by PMSEVFR_EL1 and, if implemented, PMSNEVFR_EL1 are counted. The values of PMSFCR_EL1.{FnE,FE} are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

0x812F, SAMPLE_FEED_LAT, Statistical Profiling sample taken, exceeding minimum latency

The counter counts each sample counted by SAMPLE_FEED that meets the sample latency filter constraints.

That is, each sample with a total latency greater than or equal to the minimum latency defined by PMSLATFR_EL1.MINLAT are counted. The value of PMSFCR_EL1.FL is ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

0x8130, L1D_TLB_RW, Level data TLB access, demand access

The counter counts each access counted by LD_TLB that is due to a demand Memory-read operation or demand Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1D_TLB_PRFM is implemented.

0x8131, L1I_TLB_RD, Level instruction TLB access, demand access

The counter counts each access counted by LI_TLB that is due to a demand Instruction memory access.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm recommends that this event is implemented if event L1I_TLB_PRFM is implemented.

0x8132, L1D_TLB_PRFM, Level data TLB access, software prefetch

The counter counts each access counted by LD_TLB_PRF that is due to a prefetch instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1D_TLB_RW is implemented.

0x8133, L1I_TLB_PRFM, Level instruction TLB access, software prefetch

The counter counts each access counted by LI_TLB_PRF that is due to a prefetch instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1I_TLB_RD is implemented.

0x8134, DTLB_HWUPD, Data TLB hardware update of translation table

The counter counts each access counted by L1D_TLB that causes a hardware update of a translation table entry.

Each attempted hardware update of a translation table entry is counted once. If the PE requires multiple translation table walk accesses to perform an update, this counts as a single update. If the update fails because it would not be atomic and has to be retried, each retry is counted.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8135, ITLB_HWUPD, Instruction TLB hardware update of translation table

The counter counts each access counted by L1I_TLB that causes a hardware update of a translation table entry.

Each attempted hardware update of a translation table entry is counted once. If the PE requires multiple translation table walk accesses to perform an update, this counts as a single update. If the update fails because it would not be atomic and has to be retried, each retry is counted.

The counter does not count the access if any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8136`, DTLB_STEP, Data TLB translation table walk, step

The counter counts each translation table walk access due to a refill or update of a data or unified TLB.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the translation table walk access if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access causing the refill is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the translation table walk access if the access causing the refill generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8137`, ITLB_STEP, Instruction TLB translation table walk, step

The counter counts each translation table walk access due to a refill or update of an instruction TLB.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the translation table walk access if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the translation table walk access if the access causing the refill generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8138, DTLB_WALK_LARGE, Data TLB large page translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a large page size.

The set of large page sizes is the complement of the set of small page sizes defined by the DTLB_WALK_SMALL event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8139, ITLB_WALK_LARGE, Instruction TLB large page translation table walk

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a large page size.

The set of large page sizes is the complement of the set of small page sizes defined by the ITLB_WALK_SMALL event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x813A, DTLB_WALK_SMALL, Data TLB small page translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a small page size.

The set of small page sizes is the complement of the set of large page sizes defined by the DTLB_WALK_LARGE event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x813B, ITLB_WALK_SMALL, Instruction TLB small page translation table walk

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a small page size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The set of small page sizes is the complement of the set of large page sizes defined by the ITLB_WALK_LARGE event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813C`, DTLB_WALK_RW, Data TLB demand access, with at least one translation table walk

The counter counts each demand access counted by L1D_TLB_RW that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813D`, ITLB_WALK_RD, Instruction TLB demand access, with at least one translation table walk

The counter counts each demand access counted by L1I_TLB_RD that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813E`, DTLB_WALK_PRFM, Data TLB software prefetch, with at least one translation table walk

The counter counts each software prefetch access counted by L1D_TLB_PRFM that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the software prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813F`, ITLB_WALK_PRFM, Instruction TLB software prefetch, with at least one translation table walk

The counter counts each software prefetch access counted by L1I_TLB_PRFM that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the software prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8140`, L1D_CACHE_RW, Level 1 data cache demand access

The counter counts each access counted by L1D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by speculatively executed instructions.

Arm recommends that this event is implemented if any of the following are true:

- Event L1D_CACHE_PRFM is implemented.
- Event L1D_CACHE_HWPRF is implemented.

`0x8141`, L1I_CACHE_RD, Level 1 instruction cache demand fetch

The counter counts each access counted by L1I_CACHE that is due to a demand Instruction memory access.

This includes instruction prefetches made by the PE for speculatively executed instructions.

Arm recommends that this event is implemented if any of the following are true:

- Event L1I_CACHE_PRFM is implemented.
- Event L1I_CACHE_HWPRF is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8142, L1D_CACHE_PRFM, Level 1 data cache software prefetch

The counter counts each access counted by L1D_CACHE_PRF that is due to a prefetch instruction.

Arm recommends that this event is implemented if event L1D_CACHE_RW is implemented.

0x8143, L1I_CACHE_PRFM, Level 1 instruction cache software prefetch

The counter counts each access counted by L1I_CACHE_PRF that is due to a prefetch instruction.

Arm recommends that this event is implemented if event L1I_CACHE_RD is implemented.

0x8144, L1D_CACHE_MISS, Level 1 data cache demand access miss

The counter counts each demand access counted by L1D_CACHE_RW that misses in the Level 1 data or unified cache, causing an access to outside of the Level 1 caches of this PE.

0x8145, L1I_CACHE_HWPRF, Level 1 instruction cache hardware prefetch

The counter counts each access counted by L1I_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 1 instruction or unified cache.

The PE might prefetch instructions as part of instruction pipelining, and might do so for speculatively executed instruction paths. PE instruction prefetching is not counted as hardware prefetching. Unlike PE instruction prefetching, a hardware prefetch only allocates instructions into the cache.

0x8146, L1D_CACHE_REFILL_PRFM, Level 1 data cache refill, software prefetch

The counter counts each access counted by L1D_CACHE_PRFM that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

0x8147, L1I_CACHE_REFILL_PRFM, Level 1 instruction cache refill, software prefetch

The counter counts each access counted by L1I_CACHE_PRFM that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

0x8148, L2D_CACHE_RW, Level 2 data cache demand access

The counter counts each access counted by L2D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:

- Accesses made by speculatively executed instructions.
- Accesses to the Level 2 data or unified cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

Arm recommends that this event is implemented if any of the following are true:

- Event L2D_CACHE_PRFM is implemented.
- Event L2D_CACHE_HWPRF is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8149, L2I_CACHE_RD, Level 2 instruction cache demand fetch

The counter counts each access counted by L2I_CACHE that is due to a demand Instruction memory access.

This includes:

- Instruction prefetches made by the PE for speculatively executed instructions.
- Accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a demand Instruction memory access.

Arm recommends that this event is implemented if any of the following are true:

- Event L2I_CACHE_PRFM is implemented.
- Event L2I_CACHE_HWPRF is implemented.

0x814A, L2D_CACHE_PRFM, Level 2 data cache software prefetch

The counter counts each access counted by L2D_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event L2D_CACHE_RW is implemented.

0x814B, L2I_CACHE_PRFM, Level 2 instruction cache software prefetch

The counter counts each access counted by L2I_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event L2I_CACHE_RD is implemented.

0x814C, L2D_CACHE_MISS, Level 2 data cache demand access miss

The counter counts each demand access counted by L2D_CACHE_RW that misses in the Level 1 and Level 2 data or unified caches, causing an access to outside of the Level 1 and Level 2 caches of this PE.

0x814D, L2I_CACHE_HWPRF, Level 2 instruction cache hardware prefetch

The counter counts each access counted by L2I_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 instruction or unified cache.

0x814E, L2D_CACHE_REFILL_PRFM, Level 2 data cache refill, software prefetch

The counter counts each access counted by L2D_CACHE_PRFM that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x814F, L2I_CACHE_REFILL_PRFM, Level 2 instruction cache refill, software prefetch

The counter counts each access counted by L2I_CACHE_PRFM that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8150, L3D_CACHE_RW, Level 3 data cache demand access

The counter counts each access counted by L3D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:

- Accesses made by speculatively executed instructions.
- Accesses to the Level 3 data or unified cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

Arm recommends that this event is implemented if any of the following are true:

- Event L3D_CACHE_PRFM is implemented.
- Event L3D_CACHE_HWPRF is implemented.

0x8151, L3D_CACHE_PRFM, Level 3 data cache software prefetch

The counter counts each access counted by L3D_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event L3D_CACHE_RW is implemented.

0x8152, L3D_CACHE_MISS, Level 3 data cache demand access miss

The counter counts each demand access counted by L3D_CACHE_RW that misses in the Level 1 to Level 3 data or unified caches, causing an access to outside of the Level 1 to Level 3 caches of this PE.

0x8153, L3D_CACHE_REFILL_PRFM, Level 3 data cache refill, software prefetch

The counter counts each access counted by L3D_CACHE_PRFM that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

0x8154, L1D_CACHE_HWPRF, Level 1 data cache hardware prefetch

The counter counts each access counted by L1D_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 1 data or unified cache.

0x8155, L2D_CACHE_HWPRF, Level 2 data cache hardware prefetch

The counter counts each access counted by L2D_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 data or unified cache.

0x8156, L3D_CACHE_HWPRF, Level 3 data cache hardware prefetch

The counter counts each access counted by L3D_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 3 data or unified cache.

0x8157, LL_CACHE_HWPRF, Last level cache hardware prefetch

The counter counts each access counted by LL_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Last level cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8158, STALL_FRONTEND_MEMBOUND, Frontend stall cycles, memory bound

The counter counts each cycle counted by STALL_FRONTEND when no instructions are delivered from the memory system.

The counter counts each stall that occurs when the frontend interface to memory is busy or stalled.

This includes the stalls counted by STALL_FRONTEND_L1I, STALL_FRONTEND_L2I, STALL_FRONTEND_MEM, and STALL_FRONTEND_TLB, and any other IMPLEMENTATION DEFINED memory stalls.

It does not include stalls that are counted by STALL_FRONTEND_CPUBOUND. However both events will count the same cycle counted by STALL_FRONTEND if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when no instructions are delivered from the memory system for any PE in the multithreaded processor.

0x8159, STALL_FRONTEND_L1I, Frontend stall cycles, level 1 instruction cache

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the first level of instruction cache.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand instruction miss in the first level of instruction cache for any PE in the multithreaded processor.

The counter does not count the cycle if any of the following are true:

- The STALL_FRONTEND_L2I event is implemented and there is a demand instruction miss in the second level of instruction cache, meaning the STALL_FRONTEND_L2I event counts the cycle.
- There is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the STALL_FRONTEND_MEM event counts the cycle.

0x815A, STALL_FRONTEND_L2I, Frontend stall cycles, level 2 instruction cache

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the second level of instruction or unified cache.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand instruction miss in the second level of instruction or unified cache for any PE in the multithreaded processor.

The counter does not count the cycle if there is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the STALL_FRONTEND_MEM event counts the cycle.

0x815B, STALL_FRONTEND_MEM, Frontend stall cycles, last level PE cache or memory

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the last level of instruction or unified cache within the PE clock domain or a non-cacheable instruction fetch in progress.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand instruction miss in the last level of instruction or unified cache within the PE clock domain or a non-cacheable instruction fetch in progress for any PE in the multithreaded processor.

0x815C, STALL_FRONTEND_TLB, Frontend stall cycles, TLB

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the instruction or unified TLB.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand instruction miss in the instruction or unified TLB for any PE in the multithreaded processor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x815D, STALL_BACKEND_BUSY_SMCU, Backend stall cycles, SMCU busy

The counter counts each cycle counted by CPU_CYCLES when the backend is not able to accept an operation because the SMCU is busy, for any reason.

For example:

- The PE is stalled awaiting allocation of processing resources on an SMCU.
- The PE has been allocated processing resources on an SMCU. However, the SMCU cannot process operations at the rate at which they are issued by the PE in Streaming SVE mode.

0x815E, STALL_BACKEND_BUSY_SMCU_UNALL, Backend stall cycles, awaiting SMCU allocation

The counter counts each cycle counted by STALL_BACKEND_BUSY_SMCU when the backend is stalled awaiting allocation of processing resources on an SMCU.

0x8160, STALL_FRONTEND_CPUBOUND, Frontend stall cycles, processor bound

The counter counts each cycle counted by STALL_FRONTEND when the frontend is stalled on a frontend processor resource, not including memory.

The counter counts each stall that occurs when a frontend processor resource is busy.

This includes the stalls counted by STALL_FRONTEND_FLOW, STALL_FRONTEND_FLUSH, and STALL_FRONTEND_RENAME, and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by STALL_FRONTEND_MEMBOUND. However both events will count the same cycle counted by STALL_FRONTEND if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the frontend is stalled on a frontend processor resource, not including memory for any PE in the multithreaded processor.

0x8161, STALL_FRONTEND_FLOW, Frontend stall cycles, flow control

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when the frontend is stalled on unavailability of prediction flow resources.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the frontend is stalled on unavailability of prediction flow resources for any PE in the multithreaded processor.

Note

This event is not counting stalls due to mispredictions, but rather stalls when the frontend is unable to make a prediction.

0x8162, STALL_FRONTEND_FLUSH, Frontend stall cycles, flush recovery

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when the frontend is recovering from a pipeline flush.

The situations where the frontend is flushed are IMPLEMENTATION DEFINED. For example, the frontend might be flushed on a branch misprediction or on a Context synchronization event.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the frontend is recovering from a pipeline flush for any PE in the multithreaded processor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8163, STALL_FRONTEND_RENAME, Frontend stall cycles, rename full

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the STALL_BACKEND_RENAME event counts as zero.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available for any PE in the multithreaded processor.

0x8164, STALL_BACKEND_MEMBOUND, Backend stall cycles, memory bound

The counter counts each cycle counted by STALL_BACKEND when the backend is waiting for a memory access to complete.

The counter counts each stall that occurs when the backend interface to memory is busy or stalled.

This includes the stalls counted by: STALL_BACKEND_ATOMIC, STALL_BACKEND_L1D, STALL_BACKEND_L2D, STALL_BACKEND_MEM, STALL_BACKEND_MEMCPYSET, STALL_BACKEND_ST, STALL_BACKEND_TLB, and any other IMPLEMENTATION DEFINED memory stall events.

It does not include stalls that are counted by STALL_BACKEND_CPUBOUND, although both events might count on the same cycle counted by STALL_BACKEND if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the backend is waiting for a memory access to complete for any PE in the multithreaded processor.

0x8165, STALL_BACKEND_L1D, Backend stall cycles, level 1 data cache

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the first level of data or unified cache.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand data miss in the first level of data or unified cache for any PE in the multithreaded processor.

The counter does not count the cycle if any of the following are true:

- The STALL_BACKEND_L2D event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the STALL_BACKEND_L2D event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

0x8166, STALL_BACKEND_L2D, Backend stall cycles, level 2 data cache

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the second level of data or unified cache.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand data miss in the second level of data or unified cache for any PE in the multithreaded processor.

The counter does not count the cycle if there is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

0x8167, STALL_BACKEND_TLB, Backend stall cycles, TLB

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the data or unified TLB.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand data miss in the data or unified TLB for any PE in the multithreaded processor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8168, STALL_BACKEND_ST, Backend stall cycles, store

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when the backend is stalled waiting for a store.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the backend is stalled waiting for a store for any PE in the multithreaded processor.

0x816A, STALL_BACKEND_CPUBOUND, Backend stall cycles, processor bound

The counter counts each cycle counted by STALL_BACKEND when the backend is stalled on a processor resource, not including memory.

The counter counts each stall that occurs when a backend processor resource is busy. This includes the stalls counted by STALL_BACKEND_RENAME, and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by STALL_BACKEND_MEMBOUND, although both events might count on the same cycle counted by STALL_BACKEND if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the backend is stalled on a processor resource, not including memory for any PE in the multithreaded processor.

0x816B, STALL_BACKEND_BUSY, Backend stall cycles, backend busy

The counter counts each cycle counted by STALL_BACKEND when operations are available from the frontend but the backend is not able to accept an operation because an execution unit is busy.

For example a complex operation unit such as a divider is executing a previous operation and cannot accept a new operation.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when operations are available from the frontend but the backend is not able to accept an operation because an execution unit is busy for any PE in the multithreaded processor.

0x816C, STALL_BACKEND_ILOCK, Backend stall cycles, input dependency

The counter counts each cycle counted by STALL_BACKEND when operations are available from the frontend but at least one is not ready to be sent to the backend because of an input dependency.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when operations are available from the frontend but at least one is not ready to be sent to the backend because of an input dependency for any PE in the multithreaded processor.

0x816D, STALL_BACKEND_RENAME, Backend stall cycles, rename full

The counter counts each cycle counted by STALL_BACKEND_CPUBOUND when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the STALL_FRONTEND_RENAME event counts as zero.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available for any PE in the multithreaded processor.

0x816E, STALL_BACKEND_ATOMIC, Backend stall cycles, atomic operation

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when the backend is processing an Atomic operation.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the backend is processing an Atomic operation for any PE in the multithreaded processor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x816F, STALL_BACKEND_MEMCPYSET, Backend stall cycles, Memory Copy or Set operation

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when the backend is processing an Memory Copy or Set instruction.

The Memory Copy instructions are CPY and CPYF. The Memory Set instructions are SET and SETG.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when the backend is processing an Memory Copy or Set instruction for any PE in the multithreaded processor.

0x8170, CAS_NEAR_FAIL, Atomic memory Operation speculatively executed, Compare and Swap fail

The counter counts each Compare and Swap operation counted by CAS_NEAR_SPEC that does not update the location accessed.

0x8171, CAS_NEAR_PASS, Atomic memory Operation speculatively executed, Compare and Swap pass

The counter counts each Compare and Swap operation counted by CAS_NEAR_SPEC that updates the location accessed.

0x8172, CAS_NEAR_SPEC, Atomic memory Operation speculatively executed, Compare and Swap near

The counter counts each Compare and Swap operation counted by CAS_SPEC that executes locally to the PE.

The definition of locally is IMPLEMENTATION DEFINED. Operations counted by CAS_NEAR_SPEC also generate one of the CAS_NEAR_PASS or CAS_NEAR_FAIL events.

0x8173, CAS_FAR_SPEC, Atomic memory Operation speculatively executed, Compare and Swap far

The counter counts each Compare and Swap operation counted by CAS_SPEC that does not execute locally to the PE.

The definition of locally is IMPLEMENTATION DEFINED. Operations counted by CAS_FAR_SPEC do not generate CAS_NEAR_PASS or CAS_NEAR_FAIL events.

0x8174, CAS_SPEC, Atomic memory Operation speculatively executed, Compare and Swap

The counter counts each Compare and Swap operation.

0x8175, LSE_LD_SPEC, Atomic memory Operation speculatively executed, load

The counter counts each Atomic memory operation counted by LSE_LDST_SPEC that returns a value to the PE, including Swap and Compare-and-Swap operations.

0x8176, LSE_ST_SPEC, Atomic memory Operation speculatively executed, store

The counter counts each Atomic memory operation counted by LSE_LDST_SPEC that does not return a value to the PE.

0x8177, LSE_LDST_SPEC, Atomic memory Operation speculatively executed, load or store

The counter counts each operation counted by LDST_SPEC that is an Atomic memory operation.

0x8178, REMOTE_ACCESS_WR, Access to a remote device, write

The counter counts each access counted by REMOTE_ACCESS that is a Memory-write operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8184, BR_HINT_COND_PRED_RETIRED, Branch instruction architecturally executed, predicted hinted conditional

The counter counts each branch counted by both BR_HINT_COND_RETIRED and BR_PRED_RETIRED.

These are all hinted conditional branch instructions on the architecturally executed path, where the branch is correctly predicted.

0x8185, BR_HINT_COND_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted hinted conditional

The counter counts each branch counted by both BR_HINT_COND_RETIRED and BR_MIS_PRED_RETIRED.

These are all hinted conditional branch instructions on the architecturally executed path, where the branch is mispredicted.

0x8186, UOP_RETIRED, Micro-operation architecturally executed

The counter counts each micro-operation that would be executed in a Simple sequential execution of the program.

Unlike OP_RETIRED, this event is not linked to the definition of OP_SPEC, meaning it counts micro-operations that are created from other operations after those operations are counted by OP_SPEC.

0x8188, DTLB_WALK_BLOCK, Data TLB block translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a Block.

That is, the translation table walk ends at a Block descriptor, at other than the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if both are a Block.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8189, ITLB_WALK_BLOCK, Instruction TLB block translation table walk

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a Block.

That is, the translation table walk ends at a Page descriptor, at other than the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if both are a Block.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x818A, DTLB_WALK_PAGE, Data TLB page translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a Page.

That is, the translation table walk ends at a Block descriptor, at the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if either is a Page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x818B`, ITLB_WALK_PAGE, Instruction TLB page translation table walk

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a Page.

That is, the translation table walk ends at a Page descriptor, at the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if either is a Page.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x818D`, BUS_REQ_RD, Bus request, read

The counter counts each transaction counted by BUS_REQ that is a Memory-read operation.

`0x818E`, BUS_REQ_WR, Bus request, write

The counter counts each transaction counted by BUS_REQ that is a Memory-write operation.

`0x818F`, BUS_REQ, Bus request

The counter counts each request generated by a Memory-read operation or Memory-write operation that accesses outside of the boundary of the PE and its closely-coupled caches.

Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of requests across all buses.

A bus request is the start of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The BUS_REQ event therefore counts the number of transactions. The BUS_ACCESS event counts the occupancy of the transaction on the bus.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED. This must be the same for the BUS_REQ and BUS_ACCESS events.

An Unattributable bus transaction occurs when a requester outside the PE makes a request that results in a bus transaction, for example, a coherency request.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

If the bus is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8190, ISNP_HIT_RD, Snoop hit, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8191, ISNP_HIT_NEAR_RD, Snoop hit in near local cache, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8192, ISNP_HIT_FAR_RD, Snoop hit in far local cache, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8193, ISNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8194, DSNP_HIT_RD, Snoop hit, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8195, DSNP_HIT_NEAR_RD, Snoop hit in near local cache, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8196, DSNP_HIT_FAR_RD, Snoop hit in far local cache, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8197, DSNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8198, DSNP_HIT_WR, Snoop hit, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8199, DSNP_HIT_NEAR_WR, Snoop hit in near local cache, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x819A, DSNP_HIT_FAR_WR, Snoop hit in far local cache, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x819B, DSNP_HIT_REMOTE_WR, Snoop hit in remote cache, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x819C, DSNP_HIT_RW, Snoop hit, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x819D, DSNP_HIT_NEAR_RW, Snoop hit in near local cache, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x819E, DSNP_HIT_FAR_RW, Snoop hit in far local cache, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x819F, DSNP_HIT_REMOTE_RW, Snoop hit in remote cache, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81A0, DSNP_HIT_PRFM, Snoop hit, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A1, DSNP_HIT_NEAR_PRFM, Snoop hit in near local cache, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A2, DSNP_HIT_FAR_PRFM, Snoop hit in far local cache, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A3, DSNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A4, DSNP_HIT_HWPRF, Snoop hit, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A5, DSNP_HIT_NEAR_HWPRF, Snoop hit in near local cache, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A6, DSNP_HIT_FAR_HWPRF, Snoop hit in far local cache, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A7, DSNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81A8, ISNP_HIT_PRFM, Snoop hit, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81A9, ISNP_HIT_NEAR_PRFM, Snoop hit in near local cache, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81AA, ISNP_HIT_FAR_PRFM, Snoop hit in far local cache, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81AB, ISNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81AC, ISNP_HIT_HWPRF, Snoop hit, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81AD, ISNP_HIT_NEAR_HWPRF, Snoop hit in near local cache, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81AE, ISNP_HIT_FAR_HWPRF, Snoop hit in far local cache, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81AF, ISNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B0, ISNP_HIT, Snoop hit, instruction

The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81B1, ISNP_HIT_NEAR, Snoop hit in near local cache, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B2, ISNP_HIT_FAR, Snoop hit in far local cache, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B3, ISNP_HIT_REMOTE, Snoop hit in remote cache, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B4, DSNP_HIT, Snoop hit, data

The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B5, DSNP_HIT_NEAR, Snoop hit in near local cache, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B6, DSNP_HIT_FAR, Snoop hit in far local cache, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B7, DSNP_HIT_REMOTE, Snoop hit in remote cache, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x81B8, L1I_CACHE_REFILL_HWPRF, Level 1 instruction cache refill, hardware prefetch

The counter counts each hardware prefetch counted by L1I_CACHE_HWPRF that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

0x81B9, L2I_CACHE_REFILL_HWPRF, Level 2 instruction cache refill, hardware prefetch

The counter counts each hardware prefetch counted by L2I_CACHE_HWPRF that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x81BC, L1D_CACHE_REFILL_HWPRF, Level 1 data cache refill, hardware prefetch

The counter counts each hardware prefetch counted by L1D_CACHE_HWPRF that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81BD, L2D_CACHE_REFILL_HWPRF, Level 2 data cache refill, hardware prefetch

The counter counts each hardware prefetch counted by L2D_CACHE_HWPRF that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x81BE, L3D_CACHE_REFILL_HWPRF, Level 3 data cache refill, hardware prefetch

The counter counts each hardware prefetch counted by L3D_CACHE_HWPRF that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

0x81BF, LL_CACHE_REFILL_HWPRF, Last level cache refill, hardware prefetch

The counter counts each hardware prefetch counted by LL_CACHE_HWPRF that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

0x81C0, L1I_CACHE_HIT_RD, Level 1 instruction cache demand fetch hit

The counter counts each demand fetch counted by L1I_CACHE_RD that hits in the Level 1 instruction or unified cache.

0x81C1, L2I_CACHE_HIT_RD, Level 2 instruction cache demand fetch hit

The counter counts each demand fetch counted by L2I_CACHE_RD that hits in the Level 2 instruction or unified cache.

0x81C4, L1D_CACHE_HIT_RD, Level 1 data cache demand access hit, read

The counter counts each demand read counted by L1D_CACHE_RD that hits in the Level 1 data or unified cache.

0x81C5, L2D_CACHE_HIT_RD, Level 2 data cache demand access hit, read

The counter counts each demand read counted by L2D_CACHE_RD that hits in the Level 2 data or unified cache.

0x81C6, L3D_CACHE_HIT_RD, Level 3 data cache demand access hit, read

The counter counts each demand read counted by L3D_CACHE_RD that hits in the Level 3 data or unified cache.

0x81C7, LL_CACHE_HIT_RD, Last level cache demand access hit, read

The counter counts each demand read counted by LL_CACHE_RD that hits in the Last level cache.

0x81C8, L1D_CACHE_HIT_WR, Level 1 data cache demand access hit, write

The counter counts each demand write counted by L1D_CACHE_WR that hits in the Level 1 data or unified cache.

0x81C9, L2D_CACHE_HIT_WR, Level 2 data cache demand access hit, write

The counter counts each demand write counted by L2D_CACHE_WR that hits in the Level 2 data or unified cache.

0x81CA, L3D_CACHE_HIT_WR, Level 3 data cache demand access hit, write

The counter counts each demand write counted by L3D_CACHE_WR that hits in the Level 3 data or unified cache.

0x81CB, LL_CACHE_HIT_WR, Last level cache demand access hit, write

The counter counts each demand write counted by LL_CACHE_WR that hits in the Last level cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81CC, L1D_CACHE_HIT_RW, Level 1 data cache demand access hit

The counter counts each demand access counted by L1D_CACHE_RW that hits in the Level 1 data or unified cache.

0x81CD, L2D_CACHE_HIT_RW, Level 2 data cache demand access hit

The counter counts each demand access counted by L2D_CACHE_RW that hits in the Level 2 data or unified cache.

0x81CE, L3D_CACHE_HIT_RW, Level 3 data cache demand access hit

The counter counts each demand access counted by L3D_CACHE_RW that hits in the Level 3 data or unified cache.

0x81CF, LL_CACHE_HIT_RW, Last level cache demand access hit

The counter counts each demand access counted by LL_CACHE_RW that hits in the Last level cache.

0x81D0, L1I_CACHE_HIT_RD_FPRFM, Level 1 instruction cache demand fetch first hit, fetched by software prefetch

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L1I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81D1, L2I_CACHE_HIT_RD_FPRFM, Level 2 instruction cache demand fetch first hit, fetched by software prefetch

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L2I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81D4, L1D_CACHE_HIT_RD_FPRFM, Level 1 data cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

0x81D5, L2D_CACHE_HIT_RD_FPRFM, Level 2 data cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

0x81D6, L3D_CACHE_HIT_RD_FPRFM, Level 3 data cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

0x81D7, LL_CACHE_HIT_RD_FPRFM, Last level cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81D8, L1D_CACHE_HIT_WR_FPRFM, Level 1 data cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x81D9, L2D_CACHE_HIT_WR_FPRFM, Level 2 data cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x81DA, L3D_CACHE_HIT_WR_FPRFM, Level 3 data cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x81DB, LL_CACHE_HIT_WR_FPRFM, Last level cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x81DC, L1D_CACHE_HIT_RW_FPRFM, Level 1 data cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L1D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81DD, L2D_CACHE_HIT_RW_FPRFM, Level 2 data cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L2D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81DE, L3D_CACHE_HIT_RW_FPRFM, Level 3 data cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L3D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81DF, LL_CACHE_HIT_RW_FPRFM, Last level cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the LL_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81E0, L1I_CACHE_HIT_RD_FHWPRF, Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L1I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81E1, L2I_CACHE_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L2I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81E4, L1D_CACHE_HIT_RD_FHWPRF, Level 1 data cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

0x81E5, L2D_CACHE_HIT_RD_FHWPRF, Level 2 data cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

0x81E6, L3D_CACHE_HIT_RD_FHWPRF, Level 3 data cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

0x81E7, LL_CACHE_HIT_RD_FHWPRF, Last level cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by LL_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

0x81E8, L1D_CACHE_HIT_WR_FHWPRF, Level 1 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x81E9, L2D_CACHE_HIT_WR_FHWPRF, Level 2 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x81EA, L3D_CACHE_HIT_WR_FHWPRF, Level 3 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x81EB, LL_CACHE_HIT_WR_FHWPRF, Last level cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by LL_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81EC, L1D_CACHE_HIT_RW_FHWPRF, Level 1 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L1D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81ED, L2D_CACHE_HIT_RW_FHWPRF, Level 2 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L2D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81EE, L3D_CACHE_HIT_RW_FHWPRF, Level 3 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L3D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81EF, LL_CACHE_HIT_RW_FHWPRF, Last level cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the LL_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81F0, L1I_CACHE_HIT_RD_FPRF, Level 1 instruction cache demand fetch first hit, fetched by prefetch

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L1I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81F1, L2I_CACHE_HIT_RD_FPRF, Level 2 instruction cache demand fetch first hit, fetched by prefetch

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L2I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81F4, L1D_CACHE_HIT_RD_FPRF, Level 1 data cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81F5, L2D_CACHE_HIT_RD_FPRF, Level 2 data cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

0x81F6, L3D_CACHE_HIT_RD_FPRF, Level 3 data cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

0x81F7, LL_CACHE_HIT_RD_FPRF, Last level cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

0x81F8, L1D_CACHE_HIT_WR_FPRF, Level 1 data cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x81F9, L2D_CACHE_HIT_WR_FPRF, Level 2 data cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x81FA, L3D_CACHE_HIT_WR_FPRF, Level 3 data cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x81FB, LL_CACHE_HIT_WR_FPRF, Last level cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x81FC, L1D_CACHE_HIT_RW_FPRF, Level 1 data cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L1D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81FD, L2D_CACHE_HIT_RW_FPRF, Level 2 data cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L2D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x81FE, L3D_CACHE_HIT_RW_FPRF, Level 3 data cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L3D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81FF, LL_CACHE_HIT_RW_FPRF, Last level cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the LL_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8200, L1I_CACHE_HIT, Level 1 instruction cache hit

The counter counts each access counted by L1I_CACHE that hits in the Level 1 instruction or unified cache.

0x8201, L2I_CACHE_HIT, Level 2 instruction cache hit

The counter counts each access counted by L2I_CACHE that hits in the Level 2 instruction or unified cache.

0x8204, L1D_CACHE_HIT, Level 1 data cache hit

The counter counts each access counted by L1D_CACHE that hits in the Level 1 data or unified cache.

0x8205, L2D_CACHE_HIT, Level 2 data cache hit

The counter counts each access counted by L2D_CACHE that hits in the Level 2 data or unified cache.

0x8206, L3D_CACHE_HIT, Level 3 data cache hit

The counter counts each access counted by L3D_CACHE that hits in the Level 3 data or unified cache.

0x8207, LL_CACHE_HIT, Last level cache hit

The counter counts each access counted by LL_CACHE that hits in the Last level cache.

0x8208, L1I_CACHE_HIT_PRFM, Level 1 instruction cache software prefetch hit

The counter counts each software prefetch counted by L1I_CACHE_PRFM that hits in the Level 1 instruction or unified cache.

0x8209, L2I_CACHE_HIT_PRFM, Level 2 instruction cache software prefetch hit

The counter counts each software prefetch counted by L2I_CACHE_PRFM that hits in the Level 2 instruction or unified cache.

0x820C, L1D_CACHE_HIT_PRFM, Level 1 data cache software prefetch hit

The counter counts each software prefetch counted by L1D_CACHE_PRFM that hits in the Level 1 data or unified cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x820D, L2D_CACHE_HIT_PRFM, Level 2 data cache software prefetch hit

The counter counts each software prefetch counted by L2D_CACHE_PRFM that hits in the Level 2 data or unified cache.

0x820E, L3D_CACHE_HIT_PRFM, Level 3 data cache software prefetch hit

The counter counts each software prefetch counted by L3D_CACHE_PRFM that hits in the Level 3 data or unified cache.

0x820F, LL_CACHE_HIT_PRFM, Last level cache software prefetch hit

The counter counts each software prefetch counted by LL_CACHE_PRFM that hits in the Last level cache.

0x8214, L1D_CACHE_HITM_RD, Level 1 data cache demand access hit modified, read

The counter counts each cache access counted by L1D_CACHE_HIT_RD that accesses a previously modified cache location.

0x8215, L2D_CACHE_HITM_RD, Level 2 data cache demand access hit modified, read

The counter counts each cache access counted by L2D_CACHE_HIT_RD that accesses a previously modified cache location.

0x8216, L3D_CACHE_HITM_RD, Level 3 data cache demand access hit modified, read

The counter counts each cache access counted by L3D_CACHE_HIT_RD that accesses a previously modified cache location.

0x8217, LL_CACHE_HITM_RD, Last level cache demand access hit modified, read

The counter counts each cache access counted by LL_CACHE_HIT_RD that accesses a previously modified cache location.

0x8218, L1D_CACHE_HITM_WR, Level 1 data cache demand access hit modified, write

The counter counts each cache access counted by L1D_CACHE_HIT_WR that accesses a previously modified cache location.

0x8219, L2D_CACHE_HITM_WR, Level 2 data cache demand access hit modified, write

The counter counts each cache access counted by L2D_CACHE_HIT_WR that accesses a previously modified cache location.

0x821A, L3D_CACHE_HITM_WR, Level 3 data cache demand access hit modified, write

The counter counts each cache access counted by L3D_CACHE_HIT_WR that accesses a previously modified cache location.

0x821B, LL_CACHE_HITM_WR, Last level cache demand access hit modified, write

The counter counts each cache access counted by LL_CACHE_HIT_WR that accesses a previously modified cache location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x821C, L1D_CACHE_HITM_RW, Level 1 data cache demand access hit modified

The counter counts each cache access counted by L1D_CACHE_HIT_RW that accesses a previously modified cache location.

0x821D, L2D_CACHE_HITM_RW, Level 2 data cache demand access hit modified

The counter counts each cache access counted by L2D_CACHE_HIT_RW that accesses a previously modified cache location.

0x821E, L3D_CACHE_HITM_RW, Level 3 data cache demand access hit modified

The counter counts each cache access counted by L3D_CACHE_HIT_RW that accesses a previously modified cache location.

0x821F, LL_CACHE_HITM_RW, Last level cache demand access hit modified

The counter counts each cache access counted by LL_CACHE_HIT_RW that accesses a previously modified cache location.

0x8224, DSNP_HITM_RD, Snoop hit, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that accesses a previously modified cache location.

0x8225, DSNP_HITM_NEAR_RD, Snoop hit in near local cache, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_NEAR_RD that accesses a previously modified cache location.

0x8226, DSNP_HITM_FAR_RD, Snoop hit in far local cache, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_FAR_RD that accesses a previously modified cache location.

0x8227, DSNP_HITM_REMOTE_RD, Snoop hit in remote cache, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_REMOTE_RD that accesses a previously modified cache location.

0x8228, DSNP_HITM_WR, Snoop hit, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that accesses a previously modified cache location.

0x8229, DSNP_HITM_NEAR_WR, Snoop hit in near local cache, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_NEAR_WR that accesses a previously modified cache location.

0x822A, DSNP_HITM_FAR_WR, Snoop hit in far local cache, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_FAR_WR that accesses a previously modified cache location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x822B, DSNP_HITM_REMOTE_WR, Snoop hit in remote cache, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_REMOTE_WR that accesses a previously modified cache location.

0x822C, DSNP_HITM_RW, Snoop hit, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that accesses a previously modified cache location.

0x822D, DSNP_HITM_NEAR_RW, Snoop hit in near local cache, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_NEAR_RW that accesses a previously modified cache location.

0x822E, DSNP_HITM_FAR_RW, Snoop hit in far local cache, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_FAR_RW that accesses a previously modified cache location.

0x822F, DSNP_HITM_REMOTE_RW, Snoop hit in remote cache, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_REMOTE_RW that accesses a previously modified cache location.

0x8230, LOCAL_MEM, Access to memory attached to this device

The counter counts each Memory-read operation or Memory-write operation access to external memory attached to this device.

In a multi-socket system this means attached to the socket that contains the PE. For more information, see REMOTE_ACCESS.

In systems where there multiple types of memory attached to this device with different performance characteristics, it is IMPLEMENTATION DEFINED whether accesses to all external memory types are counted by this event, or are classified as remote accesses and counted by REMOTE_MEM.

For example, in a system with an expansion memory connected to the device that has significantly higher latency than the main system memory, accesses to the main system memory might be counted by this event and accesses to the expansion memory by REMOTE_MEM.

0x8231, LOCAL_MEM_RD, Access to memory attached to this device, demand access, read

The counter counts each access counted by LOCAL_MEM_RW that is a demand Memory-read operation.

0x8232, LOCAL_MEM_WR, Access to memory attached to this device, demand access, write

The counter counts each access counted by LOCAL_MEM_RW that is a demand Memory-write operation.

0x8233, LOCAL_MEM_RW, Access to memory attached to this device, demand access

The counter counts each access counted by LOCAL_MEM that is a demand Memory-read operation or demand Memory-write operation.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8234, LOCAL_MEM_PRFM, Access to memory attached to this device, software prefetch

The counter counts each access counted by LOCAL_MEM that is due to a prefetch instruction.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

0x8235, LOCAL_MEM_LD_RETIRED, Load instruction architecturally executed, access to memory attached to this device

The counter counts each architecturally executed instruction counted by LOCAL_MEM_LDST_RETIRED that is a Memory-reading instruction.

0x8236, LOCAL_MEM_ST_RETIRED, Store instruction architecturally executed, access to memory attached to this device

The counter counts each architecturally executed instruction counted by LOCAL_MEM_LDST_RETIRED that is a Memory-writing instruction.

0x8237, LOCAL_MEM_LDST_RETIRED, Load or store instruction architecturally executed, access to memory attached to this device

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that accessed external memory attached to this device.

0x8238, REMOTE_MEM, Access to memory attached to a remote device

The counter counts each Memory-read operation or Memory-write operation access counted by REMOTE_ACCESS to external memory attached to a remote device.

The counter also counts any accesses to external memory that are not counted by LOCAL_MEM because the implementation classifies them as remote accesses. For example, accesses to expansion memory connected to the device that has significantly higher latency than the main system memory might be classified as remote accesses. For more information, see LOCAL_MEM.

0x8239, REMOTE_MEM_RD, Access to memory attached to a remote device, demand access, read

The counter counts each access counted by REMOTE_MEM_RW that is a demand Memory-read operation.

0x823A, REMOTE_MEM_WR, Access to memory attached to a remote device, demand access, write

The counter counts each access counted by REMOTE_MEM_RW that is a demand Memory-write operation.

0x823B, REMOTE_MEM_RW, Access to memory attached to a remote device, demand access

The counter counts each access counted by REMOTE_MEM that is a demand Memory-read operation or demand Memory-write operation.

0x823C, REMOTE_MEM_PRFM, Access to memory attached to a remote device, software prefetch

The counter counts each access counted by REMOTE_MEM that is due to a prefetch instruction.

0x823D, REMOTE_MEM_LD_RETIRED, Load instruction architecturally executed, access to memory attached to a remote device

The counter counts each architecturally executed instruction counted by REMOTE_MEM_LDST_RETIRED that is a Memory-reading instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x823E, REMOTE_MEM_ST_RETIRED, Store instruction architecturally executed, access to memory attached to a remote device

The counter counts each architecturally executed instruction counted by REMOTE_MEM_LDST_RETIRED that is a Memory-writing instruction.

0x823F, REMOTE_MEM_LDST_RETIRED, Load or store instruction architecturally executed, access to memory attached to a remote device

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that accessed external memory attached to a remote device.

0x8240, L1I_LFB_HIT_RD, Level 1 instruction cache demand fetch line-fill buffer hit

The counter counts each demand access counted by L1I_CACHE_HIT_RD that hits a recently fetched line in the Level 1 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8241, L2I_LFB_HIT_RD, Level 2 instruction cache demand fetch line-fill buffer hit

The counter counts each demand access counted by L2I_CACHE_HIT_RD that hits a recently fetched line in the Level 2 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8244, L1D_LFB_HIT_RD, Level 1 data cache demand access line-fill buffer hit, read

The counter counts each demand access counted by L1D_CACHE_HIT_RD that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8245, L2D_LFB_HIT_RD, Level 2 data cache demand access line-fill buffer hit, read

The counter counts each demand access counted by L2D_CACHE_HIT_RD that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8246, L3D_LFB_HIT_RD, Level 3 data cache demand access line-fill buffer hit, read

The counter counts each demand access counted by L3D_CACHE_HIT_RD that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8247, LL_LFB_HIT_RD, Last level cache demand access line-fill buffer hit, read

The counter counts each demand access counted by LL_CACHE_HIT_RD that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8248, L1D_LFB_HIT_WR, Level 1 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by L1D_CACHE_HIT_WR that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8249, L2D_LFB_HIT_WR, Level 2 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by L2D_CACHE_HIT_WR that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824A, L3D_LFB_HIT_WR, Level 3 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by L3D_CACHE_HIT_WR that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824B, LL_LFB_HIT_WR, Last level cache demand access line-fill buffer hit, write

The counter counts each demand access counted by LL_CACHE_HIT_WR that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824C, L1D_LFB_HIT_RW, Level 1 data cache demand access line-fill buffer hit

The counter counts each demand access counted by L1D_CACHE_HIT_RW that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824D, L2D_LFB_HIT_RW, Level 2 data cache demand access line-fill buffer hit

The counter counts each demand access counted by L2D_CACHE_HIT_RW that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824E, L3D_LFB_HIT_RW, Level 3 data cache demand access line-fill buffer hit

The counter counts each demand access counted by L3D_CACHE_HIT_RW that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x824F, LL_LFB_HIT_RW, Last level cache demand access line-fill buffer hit

The counter counts each demand access counted by LL_CACHE_HIT_RW that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8250, L1I_LFB_HIT_RD_FPRFM, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8251, L2I_LFB_HIT_RD_FPRFM, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8254, L1D_LFB_HIT_RD_FPRFM, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

0x8255, L2D_LFB_HIT_RD_FPRFM, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

0x8256, L3D_LFB_HIT_RD_FPRFM, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

0x8257, LL_LFB_HIT_RD_FPRFM, Last level cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8258, L1D_LFB_HIT_WR_FPRFM, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x8259, L2D_LFB_HIT_WR_FPRFM, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x825A, L3D_LFB_HIT_WR_FPRFM, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x825B, LL_LFB_HIT_WR_FPRFM, Last level cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

0x825C, L1D_LFB_HIT_RW_FPRFM, Level 1 data cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x825D, L2D_LFB_HIT_RW_FPRFM, Level 2 data cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x825E, L3D_LFB_HIT_RW_FPRFM, Level 3 data cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x825F`, LL_LFB_HIT_RW_FPRFM, Last level cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8260`, L1I_LFB_HIT_RD_FHWPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8261`, L2I_LFB_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8264`, L1D_LFB_HIT_RD_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x8265`, L2D_LFB_HIT_RD_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x8266`, L3D_LFB_HIT_RD_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8267, LL_LFB_HIT_RD_FHWPRF, Last level cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

0x8268, L1D_LFB_HIT_WR_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x8269, L2D_LFB_HIT_WR_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x826A, L3D_LFB_HIT_WR_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x826B, LL_LFB_HIT_WR_FHWPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

0x826C, L1D_LFB_HIT_RW_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x826D, L2D_LFB_HIT_RW_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x826E, L3D_LFB_HIT_RW_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x826F, LL_LFB_HIT_RW_FHWPRF, Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8270, L1I_LFB_HIT_RD_FPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8271, L2I_LFB_HIT_RD_FPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8274, L1D_LFB_HIT_RD_FPRF, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8275, L2D_LFB_HIT_RD_FPRF, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

0x8276, L3D_LFB_HIT_RD_FPRF, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

0x8277, LL_LFB_HIT_RD_FPRF, Last level cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

0x8278, L1D_LFB_HIT_WR_FPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x8279, L2D_LFB_HIT_WR_FPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x827A, L3D_LFB_HIT_WR_FPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x827B, LL_LFB_HIT_WR_FPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

0x827C, L1D_LFB_HIT_RW_FPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x827D, L2D_LFB_HIT_RW_FPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x827E, L3D_LFB_HIT_RW_FPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x827F, LL_LFB_HIT_RW_FPRF, Last level cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8280, L1I_CACHE_PRF, Level 1 instruction cache, prefetch hit

The counter counts each access counted by L1I_CACHE that is due to a prefetch instruction, or hardware prefetch.

0x8281, L2I_CACHE_PRF, Level 2 instruction cache, prefetch hit

The counter counts each access counted by L2I_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

0x8284, L1D_CACHE_PRF, Level 1 data cache, prefetch hit

The counter counts each access counted by L1D_CACHE that is due to a prefetch instruction, or hardware prefetch.

0x8285, L2D_CACHE_PRF, Level 2 data cache, prefetch hit

The counter counts each access counted by L2D_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8286, L3D_CACHE_PRF, Level 3 data cache, prefetch hit

The counter counts each access counted by L3D_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

0x8287, LL_CACHE_PRF, Last level cache, prefetch hit

The counter counts each access counted by LL_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Last level cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

0x8288, L1I_CACHE_REFILL_PRF, Level 1 instruction cache refill, prefetch hit

The counter counts each access counted by L1I_CACHE_PRF that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

0x8289, L2I_CACHE_REFILL_PRF, Level 2 instruction cache refill, prefetch hit

The counter counts each access counted by L2I_CACHE_PRF that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x828C, L1D_CACHE_REFILL_PRF, Level 1 data cache refill, prefetch hit

The counter counts each access counted by L1D_CACHE_PRF that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

0x828D, L2D_CACHE_REFILL_PRF, Level 2 data cache refill, prefetch hit

The counter counts each access counted by L2D_CACHE_PRF that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x828E, L3D_CACHE_REFILL_PRF, Level 3 data cache refill, prefetch hit

The counter counts each access counted by L3D_CACHE_PRF that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

0x828F, LL_CACHE_REFILL_PRF, Last level cache refill, prefetch hit

The counter counts each access counted by LL_CACHE_PRF that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

0x8290, ISNP_HIT_PRF, Snoop hit, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8291, ISNP_HIT_NEAR_PRF, Snoop hit in near local cache, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8292, ISNP_HIT_FAR_PRF, Snoop hit in far local cache, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8293, ISNP_HIT_REMOTE_PRF, Snoop hit in remote cache, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8294, DSNP_HIT_PRF, Snoop hit, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8295, DSNP_HIT_NEAR_PRF, Snoop hit in near local cache, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8296, DSNP_HIT_FAR_PRF, Snoop hit in far local cache, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8297, DSNP_HIT_REMOTE_PRF, Snoop hit in remote cache, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

0x8298, LL_CACHE_RW, Last level cache demand access

The counter counts each access counted by LL_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:

- Accesses made by speculatively executed instructions.
- Accesses to the Last level cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

Arm recommends that this event is implemented if any of the following are true:

- Event LL_CACHE_PRFM is implemented.
- Event LL_CACHE_HWPRF is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8299, LL_CACHE_PRFM, Last level cache software prefetch

The counter counts each access counted by LL_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Last level cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event LL_CACHE_RW is implemented.

0x829A, LL_CACHE_REFILL, Last level cache refill

The counter counts each access counted by LL_CACHE that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Last level caches, even if the data is ultimately not allocated into the Last level cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Last level cache that cause a refill that is satisfied by fetching data from memory, or a Last level cache of another PE.
- Refills of and write-backs from any other data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Last level caches of this PE.
- Accesses to the Last level cache that cause a refill of a higher level cache of this PE from outside of the Level 1 to Last level caches of this PE, even if there is no refill of the Last level cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x829B, LL_CACHE_REFILL_PRFM, Last level cache refill, software prefetch

The counter counts each access counted by LL_CACHE_PRFM that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

0x829C, LL_CACHE_WB, Last level cache write-back

The counter counts each write-back of data from the Last level cache to outside of the Level 1 to Last level caches.

For example:

- A write-back of a cache line to memory.
- A write-back of a recently fetched cache line that has not been allocated to the Last level cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Last level cache to outside of Level 1 to Last level caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Last level caches or memory.
- Writes that write through the Last level cache to outside of the Level 1 to Last level caches.
- Transfers of data from the Last level cache to another cache to satisfy a refill of the other cache.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

`0x829D`, LL_CACHE_WR, Last level cache access, write

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is a Memory-write operation.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x829F`, LL_CACHE_REFILL_WR, Last level cache refill, write

The counter counts each access counted by both LL_CACHE_REFILL and LL_CACHE_WR.

That is, every refill of the Last level cache counted by LL_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82A0`, MEM_ACCESS_RW, Data memory access, demand access

The counter counts each access counted by MEM_ACCESS that is a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by speculative instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82A1, INST_FETCH_RD, Instruction memory access, demand fetch

The counter counts each fetch counted by INST_FETCH that is a demand Instruction memory access.

This includes any fetch made for a speculative instruction.

0x82A2, MEM_ACCESS_PRFM, Data memory access, software prefetch

The counter counts each access counted by MEM_ACCESS that is due to a prefetch instruction.

0x82A3, INST_FETCH_PRFM, Instruction memory access, software prefetch

The counter counts each fetch counted by INST_FETCH that is due to a prefetch instruction.

0x82B0, L1I_CACHE_MISS_RETIRED, Instruction architecturally executed, miss in Level 1 instruction cache

The counter counts each instruction architecturally executed counted by INST_RETIRED that missed in Level 1 instruction or unified cache when fetched.

0x82B1, L2I_CACHE_MISS_RETIRED, Instruction architecturally executed, miss in Level 2 instruction cache

The counter counts each instruction architecturally executed counted by INST_RETIRED that missed in Level 2 instruction or unified cache when fetched.

0x82B4, L1D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

0x82B5, L2D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

0x82B6, L3D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

0x82B7, LL_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Last level cache

The counter counts each architecturally executed instruction counted by LL_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

0x82B8, L1D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

0x82B9, L2D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82BA, L3D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

0x82BB, LL_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Last level cache

The counter counts each architecturally executed instruction counted by LL_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

0x82BC, L1D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 1 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Level 1 data or unified cache.

0x82BD, L2D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 2 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Level 2 data or unified cache.

0x82BE, L3D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 3 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Level 3 data or unified cache.

0x82BF, LL_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Last level cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Last level cache.

0x82C4, L1D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

0x82C5, L2D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

0x82C6, L3D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

0x82C7, LL_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Last level cache

The counter counts each architecturally executed instruction counted by LL_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82C8, L1D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

0x82C9, L2D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

0x82CA, L3D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

0x82CB, LL_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Last level cache

The counter counts each architecturally executed instruction counted by LL_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

0x82CC, L1D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 1 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L1D_CACHE_HIT_LDST_RETIRED that hit modified data in Level 1 data or unified cache.

0x82CD, L2D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 2 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L2D_CACHE_HIT_LDST_RETIRED that hit modified data in Level 2 data or unified cache.

0x82CE, L3D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 3 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L3D_CACHE_HIT_LDST_RETIRED that hit modified data in Level 3 data or unified cache.

0x82CF, LL_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Last level cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LL_CACHE_HIT_LDST_RETIRED that hit modified data in Last level cache.

0x82D0, L1I_LFB_HIT_RETIRED, Instruction architecturally executed, line-fill buffer hit in Level 1 instruction cache

The counter counts each instruction architecturally executed counted by L1I_CACHE_HIT_RETIRED that hit a recently fetched line in Level 1 instruction or unified cache when fetched.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82D1, L2I_LFB_HIT_RETIRED, Instruction architecturally executed, line-fill buffer hit in Level 2 instruction cache

The counter counts each instruction architecturally executed counted by L2I_CACHE_HIT_RETIRED that hit a recently fetched line in Level 2 instruction or unified cache when fetched.

0x82D4, L1D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82D5, L2D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82D6, L3D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82D7, LL_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Last level cache

The counter counts each architecturally executed instruction counted by LL_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82D8, L1D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82D9, L2D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82DA, L3D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82DB, LL_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Last level cache

The counter counts each architecturally executed instruction counted by LL_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82DC, L1D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 1 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L1D_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Level 1 data or unified cache.

0x82DD, L2D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 2 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L2D_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Level 2 data or unified cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82DE, L3D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 3 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L3D_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Level 3 data or unified cache.

0x82DF, LL_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Last level cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LL_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Last level cache.

0x82E0, L1I_CACHE_HIT_RETIRED, Instruction architecturally executed, hit in Level 1 instruction cache

The counter counts each instruction architecturally executed counted by INST_RETIRED that hit in Level 1 instruction or unified cache when fetched.

0x82E1, L2I_CACHE_HIT_RETIRED, Instruction architecturally executed, hit in Level 2 instruction cache

The counter counts each instruction architecturally executed counted by INST_RETIRED that hit in Level 2 instruction or unified cache when fetched.

0x82E4, L1D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82E5, L2D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82E6, L3D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82E7, LL_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Last level cache

The counter counts each architecturally executed instruction counted by LL_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x82E8, L1D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 1 data cache

The counter counts each architecturally executed instruction counted by L1D_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82E9, L2D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 2 data cache

The counter counts each architecturally executed instruction counted by L2D_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82EA, L3D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 3 data cache

The counter counts each architecturally executed instruction counted by L3D_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82EB, LL_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Last level cache

The counter counts each architecturally executed instruction counted by LL_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x82EC, L1D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 1 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Level 1 data or unified cache.

0x82ED, L2D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 2 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Level 2 data or unified cache.

0x82EE, L3D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 3 data cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Level 3 data or unified cache.

0x82EF, LL_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Last level cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Last level cache.

0x82F0, ITLB_HIT_RETIRED, Instruction architecturally executed, no translation table walk

The counter counts each instruction which was architecturally executed counted by INST_RETIRED that did not cause a refill of an instruction TLB involving at least one translation table walk access.

The counter does not count the instruction which was architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the instruction which was architecturally executed if the access generates a Translation fault for any other reason.

0x82F1, DTLB_HIT_LD_RETIRED, Load instruction architecturally executed, no translation table walk

The counter counts each Memory-reading instruction architecturally executed counted by DTLB_HIT_LDST_RETIRED that did not cause a refill of a data TLB involving at least one translation table walk access.

0x82F2, DTLB_HIT_ST_RETIRED, Store instruction architecturally executed, no translation table walk

The counter counts each Memory-writing instruction architecturally executed counted by DTLB_HIT_LDST_RETIRED that did not cause a refill of a data TLB involving at least one translation table walk access.

0x82F3, DTLB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, no translation table walk

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that did not cause a refill of a data TLB involving at least one translation table walk access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82F4, ITLB_WALK_RETIRED, Instruction architecturally executed, at least one translation table walk

The counter counts each instruction which was architecturally executed counted by INST_RETIRED that caused a refill of an instruction TLB involving at least one translation table walk access.

The counter does not count the instruction which was architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the instruction which was architecturally executed if the access generates a Translation fault for any other reason.

0x82F5, DTLB_WALK_LD_RETIRED, Load instruction architecturally executed, at least one translation table walk

The counter counts each Memory-reading instruction architecturally executed counted by DTLB_WALK_LDST_RETIRED that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-reading instruction architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-reading instruction architecturally executed if the access generates a Translation fault for any other reason.

0x82F6, DTLB_WALK_ST_RETIRED, Store instruction architecturally executed, at least one translation table walk

The counter counts each Memory-writing instruction architecturally executed counted by DTLB_WALK_LDST_RETIRED that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-writing instruction architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-writing instruction architecturally executed if the access generates a Translation fault for any other reason.

0x82F7, DTLB_WALK_LDST_RETIRED, Load or store instruction architecturally executed, at least one translation table walk

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-reading instruction or Memory-writing instruction architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-reading instruction or Memory-writing instruction architecturally executed if the access generates a Translation fault for any other reason.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82F8, DTLB_WALK_PRF, Data TLB prefetch, with at least one translation table walk

The counter counts each software prefetch or hardware prefetch access counted by L1D_TLB_PRF that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the software prefetch or hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch or hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x82F9, ITLB_WALK_PRF, Instruction TLB prefetch, with at least one translation table walk

The counter counts each software prefetch or hardware prefetch access counted by L1I_TLB_PRF that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the software prefetch or hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch or hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x82FA, DTLB_WALK_HWPRF, Data TLB hardware prefetch, with at least one translation table walk

The counter counts each hardware prefetch access counted by L1D_TLB_HWPRF that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the hardware prefetch access if the access generates a Translation fault for any other reason.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FB`, ITLB_WALK_HWPRF, Instruction TLB hardware prefetch, with at least one translation table walk

The counter counts each hardware prefetch access counted by L1I_TLB_HWPRF that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FC`, L1D_TLB_PRF, Level data TLB access, prefetch

The counter counts each access counted by LD_TLB that is due to a prefetch instruction or hardware prefetch.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FD`, L1I_TLB_PRF, Level instruction TLB access, prefetch

The counter counts each access counted by LI_TLB that is due to a prefetch instruction or hardware prefetch.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FE`, L1D_TLB_HWPRF, Level data TLB access, hardware prefetch

The counter counts each access counted by LD_TLB_PRF that is due to a hardware prefetch.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x82FF, L1I_TLB_HWPRF, Level instruction TLB access, hardware prefetch

The counter counts each access counted by LI_TLB_PRF that is due to a hardware prefetch.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8304, DSNP_HITM_LD_RETIRED, Load instruction architecturally executed, snoop hit

The counter counts each architecturally executed instruction counted by DSNP_HITM_LDST_RETIRED that is a Memory-reading instruction.

0x8305, DSNP_HITM_NEAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in near cache

The counter counts each architecturally executed instruction counted by DSNP_HITM_NEAR_LDST_RETIRED that is a Memory-reading instruction.

0x8306, DSNP_HITM_FAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in far cache

The counter counts each architecturally executed instruction counted by DSNP_HITM_FAR_LDST_RETIRED that is a Memory-reading instruction.

0x8307, DSNP_HITM_REMOTE_LD_RETIRED, Load instruction architecturally executed, snoop hit in remote cache

The counter counts each architecturally executed instruction counted by DSNP_HITM_REMOTE_LDST_RETIRED that is a Memory-reading instruction.

0x8308, DSNP_HITM_ST_RETIRED, Store instruction architecturally executed, snoop hit

The counter counts each architecturally executed instruction counted by DSNP_HITM_LDST_RETIRED that is a Memory-writing instruction.

0x8309, DSNP_HITM_NEAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in near cache

The counter counts each architecturally executed instruction counted by DSNP_HITM_NEAR_LDST_RETIRED that is a Memory-writing instruction.

0x830A, DSNP_HITM_FAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in far cache

The counter counts each architecturally executed instruction counted by DSNP_HITM_FAR_LDST_RETIRED that is a Memory-writing instruction.

0x830B, DSNP_HITM_REMOTE_ST_RETIRED, Store instruction architecturally executed, snoop hit in remote cache

The counter counts each architecturally executed instruction counted by DSNP_HITM_REMOTE_LDST_RETIRED that is a Memory-writing instruction.

0x830C, DSNP_HITM_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit modified data in a data cache outside the cache hierarchy of this PE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x830D, DSNP_HITM_NEAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in near cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HITM_LDST_RETIRED that hit modified data outside of the cache hierarchy of this PE in the local PE cluster.

0x830E, DSNP_HITM_FAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in far cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HITM_LDST_RETIRED that hit modified data outside the local PE cluster on the same device.

0x830F, DSNP_HITM_REMOTE_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in remote cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HITM_LDST_RETIRED that hit modified data on a remote device.

0x8310, ISNP_HIT_RETIRED, Instruction architecturally executed, instruction fetch snoop hit

The counter counts each instruction architecturally executed counted by INST_RETIRED that hit in an instruction cache outside the cache hierarchy of this PE when fetched.

0x8311, ISNP_HIT_NEAR_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in near cache

The counter counts each instruction architecturally executed counted by ISNP_HIT_RETIRED that hit outside of the cache hierarchy of this PE in the local PE cluster when fetched.

0x8312, ISNP_HIT_FAR_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in far cache

The counter counts each instruction architecturally executed counted by ISNP_HIT_RETIRED that hit outside the local PE cluster on the same device when fetched.

0x8313, ISNP_HIT_REMOTE_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in remote cache

The counter counts each instruction architecturally executed counted by ISNP_HIT_RETIRED that hit on a remote device when fetched.

0x8314, DSNP_HIT_LD_RETIRED, Load instruction architecturally executed, snoop hit

The counter counts each architecturally executed instruction counted by DSNP_HIT_LDST_RETIRED that is a Memory-reading instruction.

0x8315, DSNP_HIT_NEAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in near cache

The counter counts each architecturally executed instruction counted by DSNP_HIT_NEAR_LDST_RETIRED that is a Memory-reading instruction.

0x8316, DSNP_HIT_FAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in far cache

The counter counts each architecturally executed instruction counted by DSNP_HIT_FAR_LDST_RETIRED that is a Memory-reading instruction.

0x8317, DSNP_HIT_REMOTE_LD_RETIRED, Load instruction architecturally executed, snoop hit in remote cache

The counter counts each architecturally executed instruction counted by DSNP_HIT_REMOTE_LDST_RETIRED that is a Memory-reading instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8318, DSNP_HIT_ST_RETIRED, Store instruction architecturally executed, snoop hit

The counter counts each architecturally executed instruction counted by DSNP_HIT_LDST_RETIRED that is a Memory-writing instruction.

0x8319, DSNP_HIT_NEAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in near cache

The counter counts each architecturally executed instruction counted by DSNP_HIT_NEAR_LDST_RETIRED that is a Memory-writing instruction.

0x831A, DSNP_HIT_FAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in far cache

The counter counts each architecturally executed instruction counted by DSNP_HIT_FAR_LDST_RETIRED that is a Memory-writing instruction.

0x831B, DSNP_HIT_REMOTE_ST_RETIRED, Store instruction architecturally executed, snoop hit in remote cache

The counter counts each architecturally executed instruction counted by DSNP_HIT_REMOTE_LDST_RETIRED that is a Memory-writing instruction.

0x831C, DSNP_HIT_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in a data cache outside the cache hierarchy of this PE.

0x831D, DSNP_HIT_NEAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in near cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HIT_LDST_RETIRED that hit outside of the cache hierarchy of this PE in the local PE cluster.

0x831E, DSNP_HIT_FAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in far cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HIT_LDST_RETIRED that hit outside the local PE cluster on the same device.

0x831F, DSNP_HIT_REMOTE_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in remote cache

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HIT_LDST_RETIRED that hit on a remote device.

0x8320, L1D_CACHE_REFILL_PERCYC, Level 1 data or unified cache refills in progress

The counter increments by the number of Level 1 data or unified cache refills counted by L1D_CACHE_REFILL in progress on each Processor cycle.

The ratio L1D_CACHE_REFILL_PERCYC ÷ L1D_CACHE_REFILL is the mean duration of Level 1 data or unified cache refills in Processor cycles.

0x8321, L2D_CACHE_REFILL_PERCYC, Level 2 data or unified cache refills in progress

The counter increments by the number of Level 2 data or unified cache refills counted by L2D_CACHE_REFILL in progress on each Processor cycle.

The ratio L2D_CACHE_REFILL_PERCYC ÷ L2D_CACHE_REFILL is the mean duration of Level 2 data or unified cache refills in Processor cycles.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8322, L3D_CACHE_REFILL_PERCYC, Level 3 data or unified cache refills in progress

The counter increments by the number of Level 3 data or unified cache refills counted by L3D_CACHE_REFILL in progress on each Processor cycle.

The ratio L3D_CACHE_REFILL_PERCYC ÷ L3D_CACHE_REFILL is the mean duration of Level 3 data or unified cache refills in Processor cycles.

0x8324, L1I_CACHE_REFILL_PERCYC, Level 1 instruction or unified cache refills in progress

The counter increments by the number of Level 1 instruction or unified cache refills counted by L1I_CACHE_REFILL in progress on each Processor cycle.

The ratio L1I_CACHE_REFILL_PERCYC ÷ L1I_CACHE_REFILL is the mean duration of Level 1 instruction or unified cache refills in Processor cycles.

0x8325, L2I_CACHE_REFILL_PERCYC, Level 2 instruction or unified cache refills in progress

The counter increments by the number of Level 2 instruction or unified cache refills counted by L2I_CACHE_REFILL in progress on each Processor cycle.

The ratio L2I_CACHE_REFILL_PERCYC ÷ L2I_CACHE_REFILL is the mean duration of Level 2 instruction or unified cache refills in Processor cycles.

0x8328, REMOTE_CACHE_HIT_RD, Remote cache demand access hit, read

The counter counts each cache hit on a remote device due to a demand read.

0x8329, REMOTE_CACHE_HITM_RD, Remote cache demand access hit modified, read

The counter counts each cache hit on a remote device counted by REMOTE_CACHE_HIT_RD that accesses a previously modified location.

0x832A, REMOTE_LFB_HIT_RD, Remote cache demand access line-fill buffer hit, read

The counter counts each cache hit on a remote device counted by REMOTE_CACHE_HIT_RD that hits a recently fetched line.

That is, the read hits a cache line that is in the process of being loaded into the cache attached to a remote device, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x832B, REMOTE_CACHE_HIT_PRFM, Remote cache software prefetch hit

The counter counts each cache hit on a remote device due to a software prefetch.

0x832C, STALL_BACKEND_ATOMIC_FP, Backend stall cycles, atomic in-memory floating-point load or store

The counter counts each cycle counted by STALL_BACKEND_ATOMIC when the backend is processing an Atomic in-memory floating-point operation.

0x832D, LSE_FP_LD_SPEC, Atomic in-memory floating-point Operation speculatively executed, load

The counter counts each in-memory floating-point operation counted by LSE_FP_LDST_SPEC that atomically loads a value from memory, operates on that value, stores the result of the operation to memory, and writes the initially loaded value to a destination register.

This includes operations with acquire, release, and neither release nor acquire semantics.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x832E, LSE_FP_ST_SPEC, Atomic in-memory floating-point Operation speculatively executed, store

The counter counts each in-memory floating-point operation counted by LSE_FP_LDST_SPEC that atomically loads a value from memory, operates on that value, stores the result of the operation to memory, and does not write the initially loaded value to a destination register.

This includes operations with acquire, release, and neither release nor acquire semantics.

0x832F, LSE_FP_LDST_SPEC, Atomic in-memory floating-point Operation speculatively executed, load or store

The counter counts each in-memory floating-point operation counted by LSE_LDST_SPEC that atomically loads a value from memory, operates on the value, and stores the result of the operation to memory.

This includes operations with acquire, release, and neither release nor acquire semantics.

0x8330, L1GCS_CACHE, Level 1 GCS cache access

The counter counts each GCS data read operation or GCS data write operation that causes a cache access to at least a Level 1 cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as `GCSPUSHX` or `GCSPOPX`. Each access to other Level 1 memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

Accesses to a Level 1 cache due to a GCS hardware prefetcher as well are counted.

When there are separate GCS caching structures implemented:

- It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.
- Accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

0x8331, L1GCS_CACHE_RW, Level 1 GCS cache demand access

The counter counts each access counted by L1GCS_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by Speculatively executed instructions.

Arm recommends that this event is implemented if event L1GCS_CACHE_HWPRF is implemented.

0x8332, L1GCS_CACHE_HWPRF, Level 1 GCS cache hardware prefetch

The counter counts each access counted by L1GCS_CACHE that is due to a GCS hardware prefetcher.

0x8334, L1GCS_CACHE_MISS, Level 1 GCS cache demand access miss

The counter counts each access counted by L1GCS_CACHE_RW that misses in the Level 1 caches causing an access to outside of the Level 1 caches of this PE.

0x8335, L1GCS_CACHE_MISS_RD, Level 1 GCS cache demand access read miss

The counter counts each access counted by L1GCS_CACHE_MISS that is also a Memory-read operation.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm recommends that this event is implemented if event L1GCS_CACHE is implemented.

0x8336, L1GCS_CACHE_HIT_RW, Level 1 GCS cache demand access hit

The counter counts each access counted by L1GCS_CACHE_RW that hits in a Level 1 cache.

0x8337, L1GCS_CACHE_HIT_RW_FHWPRF, Level 1 GCS cache demand access first hit, fetched by GCS hardware prefetcher

The counter counts each demand access first hit counted by L1GCS_CACHE_HIT_RW where the cache line was fetched by a GCS hardware prefetcher.

That is, the L1GCS_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8338, L1GCS_CACHE_REFILL, Level 1 GCS cache refill

The counter counts each access counted by L1GCS_CACHE that causes a refill of a Level 1 cache from outside of the Level 1 cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8339, L1GCS_CACHE_REFILL_HWPRF, Level 1 GCS cache refill, hardware prefetch

The counter counts each hardware prefetch counted by L1GCS_CACHE_HWPRF that causes a refill of a Level 1 cache from outside of the Level 1 cache.

0x833A, L1GCS_CACHE_REFILL_PERCYC, Level 1 GCS cache refills in progress

The counter increments by the number of Level 1 GCS cache refills counted by L1GCS_CACHE_REFILL in progress on each Processor cycle.

The ratio L1GCS_CACHE_REFILL_PERCYC ÷ L1GCS_CACHE_REFILL is the mean duration of Level 1 GCS cache refills in Processor cycles.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x833C, L1GCS_LFB_HIT_RW, Level 1 GCS cache demand access line-fill buffer hit

The counter counts each demand access counted by L1GCS_CACHE_HIT_RW that hits a recently fetched line in a Level 1 cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x833D, L1GCS_LFB_HIT_RW_FHWPRF, Level 1 GCS cache demand access line-fill buffer first hit, recently fetched by GCS hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L1GCS_LFB_HIT_RW where the cache line was fetched by a GCS hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1GCS_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

0x833E, L1GCS_CACHE_INVAL, Level 1 GCS cache invalidate

The counter counts each invalidation of a cache line in a Level 1 GCS cache.

For example:

- Invalidation of a cache line because of a `GCSB` effect.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 cache invalidates a line in the Level 1 cache.

Note

Software that uses this event must know whether the Level 1 cache is shared with other PEs. This event does not follow the general rule of Level 1 cache events of only counting Attributable events.

0x8340, L1GCS_TLB, Level 1 GCS TLB access

The counter counts each GCS data read operation or GCS data write operation that causes a TLB access to at least a Level 1 TLB.

Each access to a TLB entry is counted including multiple accesses caused by single instructions such as `GCSPUSHX` or `GCSPOPX`.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8341, L1GCS_TLB_RW, Level 1 GCS demand TLB access

The counter counts each access counted by L1GCS_TLB that is due to a demand Memory-read operation or demand Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8342, L1GCS_TLB_HWPRF, Level 1 GCS demand TLB access, GCS hardware prefetch

The counter counts each access counted by L1GCS_TLB that is due to a GCS hardware prefetch.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8344, GCSTLB_WALK, GCS TLB access with at least one translation table walk

The counter counts each access counted by L1GCS_TLB that causes a refill of a TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

Accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is due to a TLB maintenance instruction.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8345, GCSTLB_WALK_RW, GCS TLB demand access with at least one translation table walk

The counter counts each demand access counted by both GCSTLB_WALK and L1GCS_TLB_RW.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x8346, GCSTLB_WALK_PERCYC, Translation table walks in progress

The counter increments by the number of translation table walks counted by GCSTLB_WALK in progress on each Processor cycle.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The ratio GCSTLB_WALK_PERCYC ÷ GCSTLB_WALK is the mean duration of translation table walks in Processor cycles.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8347`, GCSTLB_WALK_STEP, GCS TLB translation table walk, step

The counter counts each translation table walk access due to a refill of a TLB.

The translation table walk must be due to either GCS data read operation or GCS data write operation.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the event if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the event if the access causing the refill generates a Translation fault for any other reason.

Implementation of this optional event requires that FEAT_E0PD is implemented.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8348`, SAMPLE_FEED_FP, Statistical Profiling sample taken, floating-point

The counter counts each sample counted by SAMPLE_FEED that is a floating-point data-processing, load, or store operation.

The values of PMSFCR_EL1.{FP,FT} and, if FEAT_SPE_EFT is implemented, PMSFCR_EL1.FPm, are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

`0x8349`, SAMPLE_FEED_SIMD, Statistical Profiling sample taken, SIMD

The counter counts each sample counted by SAMPLE_FEED that is a SIMD data-processing, load, or store operation.

The values of PMSFCR_EL1.{SIMD,FT} and, if FEAT_SPE_EFT is implemented, PMSFCR_EL1.SIMDm, are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

`0x8350`, STALL_BACKEND_L1GCS, Backend stall cycles, Level 1 GCS cache

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in a first level cache.

The demand data miss must be due to either GCS data read operation or GCS data write operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the cycle if any of the following are true:

- The STALL_BACKEND_L2D event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the STALL_BACKEND_L2D event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

Implementation of this optional event requires that the first level cache is implemented within the PE clock domain and is not the last level cache within the PE clock domain.

Implementation of this optional event requires implementing L1GCS_CACHE event.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand data miss in a first level cache for any PE in the multithreaded processor.

Note

It is possible that both the events STALL_BACKEND_L1GCS and STALL_BACKEND_L1D events may be counted on a same cycle.

`0x8351`, STALL_BACKEND_GCSTLB, Backend stall cycles, GCS TLB

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in a TLB.

The demand data miss must be due to translation of a GCS data read operation or GCS data write operation.

Implementation of this optional event requires implementing L1GCS_TLB event.

For a multithreaded processor implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts when there is a demand data miss in a TLB for any PE in the multithreaded processor.

`0x8352`, SME_FP_SPEC, Operation speculatively executed, SME floating-point, data-processing

The counter counts each speculatively executed floating-point data-processing operation counted by SE_FP_SPEC due to an instruction which reads from or writes to any part of the ZA array.

`0x8353`, SE_FP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point, data-processing

The counter counts each speculatively executed floating-point data-processing operation counted by FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_SPEC, SVE_FP_SPEC, or SME_FP_SPEC.

It is IMPLEMENTATION DEFINED which floating-point data-processing operations are counted in AArch32 state.

`0x8354`, SVE_CMP_IGRANULE_SPEC, Operation speculatively executed, SVE in-granule compare

The counter counts each speculatively executed SVE in-granule comparison operation counted by SVE_SPEC due to an instruction that detects or counts matching elements between segments in vectors.

This includes operations which are due to the following instructions:

- SVE: `MATCH`, `NMATCH`, or `HISTSEG`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8355, SVE_CMP_XGRANULE_SPEC, Operation speculatively executed, SVE cross-granule compare

The counter counts each speculatively executed SVE cross-granule comparison operation counted by SVE_SPEC due to due to an instruction that detects or counts matching elements between vectors.

This includes operations which are due to the following instructions:

- SVE: `HISTCNT`.

0x8356, SVE_BITPERM_SPEC, Operation speculatively executed, SVE in-element permute

The counter counts each speculatively executed SVE permute operation counted by SVE_SPEC due to an instruction that permutes bits within elements of a vector.

This includes operations which are due to the following instructions:

- SVE: `BDEP`, `BEXT`, or `BGRP`.

0x835C, SME_SPEC, Operation speculatively executed, SME data processing

The counter counts each operation counted by SE_SPEC that is an SME data-processing operation.

An SME data-processing operation is any operation due to an instruction that operates on the ZA or ZT registers that is not any of the following:

- A load or store operation. These are counted by SME_INST_SPEC.
- A non-SIMD SME operation.

Operations due to data-processing instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE data-processing operations.

0x835D, SE_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME data processing

The counter counts each operation counted by INST_SPEC that is an Advanced SIMD, SVE, or SME data-processing operation.

See ASE_SVE_SPEC and SME_SPEC for these classifications.

0x835E, SME_INST_SPEC, Operation speculatively executed, SME

The counter counts each speculatively executed operation counted by SE_INST_SPEC that is classified as an SME operation.

An SME operation is any operation due to an instruction that operates on the ZA or ZT registers that is not any of the following:

- A non-SIMD SME operation.

Operations due to instructions defined by FEAT_SME which involve the SVE registers but do not involve any ZA or ZT registers are counted as SVE operations.

0x835F, SE_INST_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME

The counter counts each speculatively executed operation counted by INST_SPEC that is classified as an Advanced SIMD, SVE, or SME operation.

See ASE_SVE_INST_SPEC and SME_INST_SPEC for these classifications.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8360, SME_INT8_SPEC, Operation speculatively executed, SME 8-bit integer, data-processing

The counter counts each speculatively executed 8-bit integer data-processing operation counted by SME_INT_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x8361, SE_INT8_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 8-bit integer, data-processing

The counter counts each speculatively executed 8-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT8_SPEC, SVE_INT8_SPEC, or SME_INT8_SPEC.

It is IMPLEMENTATION DEFINED which 8-bit integer data-processing operations are counted in AArch32 state.

0x8362, SME_FP_BF16_SPEC, Operation speculatively executed, SME BFloat16 floating-point, data-processing

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by SME_FP_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x8363, SE_FP_BF16_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD BFloat16 floating-point, data-processing

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by SVE_FP_BF16_SPEC, or SME_FP_BF16_SPEC.

0x8364, SME_INT16_SPEC, Operation speculatively executed, SME 16-bit integer, data-processing

The counter counts each speculatively executed 16-bit integer data-processing operation counted by SME_INT_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x8365, SE_INT16_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 16-bit integer, data-processing

The counter counts each speculatively executed 16-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT16_SPEC, SVE_INT16_SPEC, or SME_INT16_SPEC.

It is IMPLEMENTATION DEFINED which 16-bit integer data-processing operations are counted in AArch32 state.

0x8366, SME_FP_HP_SPEC, Operation speculatively executed, SME half-precision floating-point, data-processing

The counter counts each speculatively executed half-precision floating-point data-processing operation counted by SME_FP_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8367, SE_FP_HP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD half-precision floating-point, data-processing

The counter counts each speculatively executed half-precision floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_HP_SPEC, SVE_FP_HP_SPEC, or SME_FP_HP_SPEC.

It is IMPLEMENTATION DEFINED which half-precision floating-point data-processing operations are counted in AArch32 state.

0x8368, SME_INT32_SPEC, Operation speculatively executed, SME 32-bit integer, data-processing

The counter counts each speculatively executed 32-bit integer data-processing operation counted by SME_INT_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x8369, SE_INT32_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 32-bit integer, data-processing

The counter counts each speculatively executed 32-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT32_SPEC, SVE_INT32_SPEC, or SME_INT32_SPEC.

It is IMPLEMENTATION DEFINED which 32-bit integer data-processing operations are counted in AArch32 state.

0x836A, SME_FP_SP_SPEC, Operation speculatively executed, SME single-precision floating-point, data-processing

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by SME_FP_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x836B, SE_FP_SP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD single-precision floating-point, data-processing

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_SP_SPEC, SVE_FP_SP_SPEC, or SME_FP_SP_SPEC.

It is IMPLEMENTATION DEFINED which single-precision floating-point data-processing operations are counted in AArch32 state.

0x836C, SME_INT64_SPEC, Operation speculatively executed, SME 64-bit integer, data-processing

The counter counts each speculatively executed 64-bit integer data-processing operation counted by SME_INT_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x836D, SE_INT64_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 64-bit integer, data-processing

The counter counts each speculatively executed 64-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT64_SPEC, SVE_INT64_SPEC, or SME_INT64_SPEC.

It is IMPLEMENTATION DEFINED which 64-bit integer data-processing operations are counted in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x836E, SME_FP_DP_SPEC, Operation speculatively executed, SME double-precision floating-point, data-processing

The counter counts each speculatively executed double-precision floating-point data-processing operation counted by SME_FP_SPEC due to an instruction which reads from or writes to any part of the ZA array.

This event counts based on the largest type read or written by an operation.

0x836F, SE_FP_DP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD double-precision floating-point, data-processing

The counter counts each speculatively executed double-precision floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_DP_SPEC, SVE_FP_DP_SPEC, or SME_FP_DP_SPEC.

It is IMPLEMENTATION DEFINED which double-precision floating-point data-processing operations are counted in AArch32 state.

0x8370, SME_FP_ADDSUB_SPEC, Operation speculatively executed, SME floating-point addition or subtraction

The counter counts each speculatively executed floating-point addition or subtraction operation counted by SE_FP_ADDSUB_SPEC due to any of the following instructions:

- SME2: BFADD, BFSUB, FADD, or FSUB.

0x8371, SE_FP_ADDSUB_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point addition or subtraction

The counter counts each speculatively executed floating-point addition or subtraction operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_ADDSUB_SPEC, SVE_FP_ADDSUB_SPEC, or SME_FP_ADDSUB_SPEC.

0x8372, SME_FP_FMA_SPEC, Operation speculatively executed, SME floating-point multiply-add or multiply-subtract

The counter counts each speculatively executed floating-point multiply-add, multiply-add long, multiply-subtract, or multiply-subtract long operation counted by SE_FP_FMA_SPEC due to any of the following instructions:

- SME2: BFMLA, BFMLAL, BFMLS, BFMLSL, FMLA, FMLAL, FMLALL, FMLS, or FMLSL.

0x8373, SE_FP_FMA_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point multiply-add or multiply-subtract

The counter counts each speculatively executed floating-point multiply-add, multiply-add long, multiply-subtract, or multiply-subtract long operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_FMA_SPEC, SVE_FP_FMA_SPEC, or SME_FP_FMA_SPEC.

0x8374, SME_FP_DOT_SPEC, Operation speculatively executed, SME floating-point dot product

The counter counts each speculatively executed floating-point dot product operation counted by SE_FP_DOT_SPEC due to any of the following instructions:

- SME2: BFDOT, BFVDOT, FDOT, FVDOT, FVDOTB, or FVDOTT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8375, SE_FP_DOT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point dot product

The counter counts each speculatively executed floating-point dot product operation counted by SE_FP_SPEC due to any of the following A64 instructions:

- When FEAT_BF16 is implemented, Advanced SIMD: BFDOT.
- When FEAT_FP8DOT2 is implemented, Advanced SIMD: FDOT (8-bit floating-point to half-precision, by element) or FDOT (8-bit floating-point to half-precision, vector).
- When FEAT_FP8DOT4 is implemented, Advanced SIMD: FDOT (8-bit floating-point to single-precision, by element) or FDOT (8-bit floating-point to single-precision, vector).
- SVE: BFDOT.
- SVE2: FDOT.
- SME2: BFDOT, BFVDOT, FDOT, FVDOT, FVDOTB, or FVDOTT.

That is, each operation counted by ASE_FP_DOT_SPEC, SVE_FP_DOT_SPEC, or SME_FP_DOT_SPEC.

0x8376, SME_FP_MOPA_SPEC, Operation speculatively executed, SME floating-point outer product and accumulate, or outer product and subtract

The counter counts each speculatively executed floating-point outer product and accumulate, or outer product and subtract operation counted by SE_FP_SPEC due to any of the following instructions:

- SME2: BFMOPA, BFMOPS, FMOPA, or FMOPS.

0x8378, SME_INT_SPEC, Operation speculatively executed, SME integer, data-processing

The counter counts each speculatively executed integer data-processing operation counted by SE_INT_SPEC due to an instruction which reads from or writes to any part of the ZA array.

0x8379, SE_INT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer, data-processing

The counter counts each speculatively executed integer data-processing operation counted by INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT_SPEC, SVE_INT_SPEC, or SME_INT_SPEC.

It is IMPLEMENTATION DEFINED which integer data-processing operations are counted in AArch32 state.

0x837A, SME_INT_MUL_SPEC, Operation speculatively executed, SME integer multiply or multiply-accumulate

The counter counts each speculatively executed integer multiply, multiply-add, or multiply-subtract operation counted by SE_INT_MUL_SPEC due to any of the following instructions:

- SME2: SMLAL, SMLALL, SMLSL, SMLSLL, SUMLALL, UMLAL, UMLALL, UMLSL, UMLSLL, or USMLALL.

0x837B, SE_INT_MUL_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer multiply or multiply-accumulate

The counter counts each speculatively executed integer multiply, multiply-add, or multiply-subtract operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT_MUL_SPEC, SVE_INT_MUL_SPEC, or SME_INT_MUL_SPEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x837C, SME_INT_DOT_SPEC, Operation speculatively executed, SME integer dot product

The counter counts each speculatively executed integer dot product operation counted by SE_INT_DOT_SPEC due to any of the following instructions:

- SME2: SDOT, SUDOT, SUVDOT, SVDOT, UDOT, USDOT, USVDOT, or UVDOT.

0x837D, SE_INT_DOT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer dot product

The counter counts each speculatively executed integer dot product operation counted by SE_INT_SPEC due to any of the following A64 instructions:

- When FEAT_DotProd is implemented, Advanced SIMD: SDOT or UDOT.
- When FEAT_I8MM is implemented, Advanced SIMD: SUDOT or USDOT.
- SVE: SDOT, SUDOT, UDOT, or USDOT.
- SVE2: CDOT, SDOT, or UDOT.
- SME2: SDOT, SUDOT, SUVDOT, SVDOT, UDOT, USDOT, USVDOT, or UVDOT.

That is, each operation counted by ASE_INT_DOT_SPEC, SVE_INT_DOT_SPEC, or SME_INT_DOT_SPEC.

0x837E, SME_INT_MOPA_SPEC, Operation speculatively executed, SME integer outer product and accumulate, or outer product and subtract

The counter counts each speculatively executed integer outer product and accumulate, or outer product and subtract operation counted by SE_INT_SPEC due to any of the following instructions:

- SME2: BMOPA, BMOPS, SMOPA, SMOPS, SUMOPA (4-way), SUMOPS, UMOPA, UMOPS, USMOPA (4-way), or USMOPS.

0x8380, ZA_ACTIVE, CPU cycles, ZA enabled

The counter counts each cycle counted by CPU_CYCLES when PSTATE.ZA is 1.

0x8381, SME_PRED2_NOT_FULL_SPEC, Operation speculatively executed, SME 2D predicated with at least one inactive element

The counter counts each speculatively executed predicated 2D SME operation which targets the ZA array counted by SME_PRED2_SPEC where at least one element is Inactive.

That is, at least one element in the Governing predicates is FALSE.

For outer product instructions which are widening, predication is considered with respect to the input element size.

0x8384, SME_PRED2_SPEC, Operation speculatively executed, SME 2D predicated

The counter counts each speculatively executed 2D operation which targets the ZA array counted by SVE_PRED_SPEC due to an SME instruction with a Governing predicate operand that determines the Active elements.

0x8385, SME_PRED2_EMPTY_SPEC, Operation speculatively executed, SME 2D predicated with no active elements

The counter counts each speculatively executed predicated 2D SME operation which targets the ZA array counted by SME_PRED2_NOT_FULL_SPEC where all elements are Inactive.

That is, all elements in the Governing predicates are FALSE.

For outer product instructions which are widening, predication is considered with respect to the input element size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8386, SME_PRED2_FULL_SPEC, Operation speculatively executed, SME 2D predicated with all active elements

The counter counts each speculatively executed predicated 2D SME operation which targets the ZA array counted by SME_PRED2_SPEC where all elements are Active.

That is, all elements in the Governing predicates are TRUE.

For outer product instructions which are widening, predication is considered with respect to the input element size.

0x8387, SME_PRED2_PARTIAL_SPEC, Operation speculatively executed, SME 2D predicated with partially active elements

The counter counts each speculatively executed predicated 2D SME operation which targets the ZA array counted by SME_PRED2_NOT_FULL_SPEC where neither all elements are Active nor all elements are Inactive.

That is, the elements in the Governing predicates are neither all TRUE nor all FALSE.

For outer product instructions which are widening, predication is considered with respect to the input element size.

0x8388, SME_LDST_ZAREG_SPEC, SME ZA unpredicated load/store

The counter counts each speculatively executed operation that reads from or writes to memory counted by SME_LDST_REG_SPEC that was due to an unpredicated instruction targeting the ZA array.

That is, due to any of the following instructions:

- SME2: LDR (array vector) or STR (array vector).

0x8389, SME_LD_ZAREG_SPEC, SME ZA unpredicated load

The counter counts each speculatively executed operation that reads from memory counted by SME_LDST_ZAREG_SPEC that was due to an unpredicated instruction targeting the ZA array.

That is, due to any of the following instructions:

- SME2: LDR (array vector).

0x838A, SME_ST_ZAREG_SPEC, SME ZA unpredicated store

The counter counts each speculatively executed operation that writes to memory counted by SME_LDST_ZAREG_SPEC that was due to an unpredicated instruction targeting the ZA array.

That is, due to any of the following instructions:

- SME2: STR (array vector).

0x838C, SME_LDST_ZTREG_SPEC, SME ZT unpredicated load/store

The counter counts each speculatively executed operation that reads from or writes to memory counted by SME_LDST_REG_SPEC that was due to an unpredicated instruction targeting the ZT register.

That is, due to any of the following instructions:

- SME2: LDR (table) or STR (table).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x838D, SME_LD_ZTREG_SPEC, SME ZT unpredicated load

The counter counts each speculatively executed operation that reads from memory counted by SME_LDST_ZTREG_SPEC that was due to an unpredicated instruction targeting the ZT register.

That is, due to any of the following instructions:

- SME2: LDR (table).

0x838E, SME_ST_ZTREG_SPEC, SME ZT unpredicated store

The counter counts each speculatively executed operation that writes to memory counted by SME_LDST_ZTREG_SPEC that was due to an unpredicated instruction targeting the ZT register.

That is, due to any of the following instructions:

- SME2: STR (table).

0x8390, SME_LDST_REG_SPEC, SME unpredicated load/store

The counter counts each speculatively executed operation that reads from or writes to memory counted by SME_INST_SPEC that was due to an unpredicated instruction targeting the ZA array or ZT register.

That is, due to any of the following instructions:

- SME2: LDR or STR.

0x8391, SME_LD_REG_SPEC, SME unpredicated load

The counter counts each speculatively executed operation that reads from memory counted by SME_LDST_REG_SPEC that was due to an unpredicated instruction targeting the ZA array or ZT register.

That is, due to any of the following instructions:

- SME2: LDR.

0x8392, SME_ST_REG_SPEC, SME unpredicated store

The counter counts each speculatively executed operation that writes to memory counted by SME_LDST_REG_SPEC that was due to an unpredicated instruction targeting the ZA array or ZT register.

That is, due to any of the following instructions:

- SME2: STR.

0x8394, SME_LDST_TILE_SPEC, SME predicated tile load/store

The counter counts each speculatively executed operation that reads from or writes to memory counted by SME_INST_SPEC that was due to an instruction with at least one governing predicate which is targeting the ZA array.

That is, due to any of the following instructions:

- SME2: LD1B (scalar plus scalar, tile slice), LD1D (scalar plus scalar, tile slice), LD1H (scalar plus scalar, tile slice), LD1Q, LD1W (scalar plus scalar, tile slice), ST1B (scalar plus scalar, tile slice), ST1D (scalar plus scalar, tile slice), ST1H (scalar plus scalar, tile slice), ST1Q, or ST1W (scalar plus scalar, tile slice).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8395, SME_LD_TILE_SPEC, SME predicated tile load

The counter counts each speculatively executed operation that reads from memory counted by SME_LDST_TILE_SPEC that was due to an instruction with at least one governing predicate which is targeting the ZA array.

That is, due to any of the following instructions:

- SME2: LD1B (scalar plus scalar, tile slice), LD1D (scalar plus scalar, tile slice), LD1H (scalar plus scalar, tile slice), LD1Q, or LD1W (scalar plus scalar, tile slice).

0x8396, SME_ST_TILE_SPEC, SME predicated tile store

The counter counts each speculatively executed operation that writes to memory counted by SME_LDST_TILE_SPEC that was due to an instruction with at least one governing predicate which is targeting the ZA array.

That is, due to any of the following instructions:

- SME2: ST1B (scalar plus scalar, tile slice), ST1D (scalar plus scalar, tile slice), ST1H (scalar plus scalar, tile slice), ST1Q, or ST1W (scalar plus scalar, tile slice).

0x8398, ASE_LUT_SPEC, Lookup table operation speculatively executed, Advanced SIMD data processing

The counter counts each speculatively executed LUT operation counted by SE_LUT_SPEC that returns a value from a lookup table made up of a V register which is indexed by values from a V register.

0x8399, SVE_LUT_SPEC, Lookup table operation speculatively executed, SVE data processing

The counter counts each speculatively executed LUT operation counted by SE_LUT_SPEC that returns a value from a lookup table made up of a Z register which is indexed by values from a Z register.

That is, due to any of the following instructions:

- SVE2: LUTI2 (8-bit and 16-bit) or LUTI4 (8-bit and 16-bit).

0x839A, SME_LUT_SPEC, Lookup table operation speculatively executed, SME data processing

The counter counts each speculatively executed LUT operation counted by SE_LUT_SPEC that returns a value from a lookup table made up of a ZT register which is indexed by values from a Z register.

That is, due to any of the following instructions:

- SME2: LUTI2 or LUTI4.

0x839B, SE_LUT_SPEC, Lookup table operation speculatively executed, Advanced SIMD, SVE, or SME data processing

The counter counts each speculatively executed LUT operation counted by SE_INST_SPEC that returns a value from a lookup table made up of a ZT register, Z or V register which is indexed by values from a Z or V register.

0x83A0, N1_CACHE_HIT_RD, Cache at distance 1, demand access hit, read

The counter counts each cache hit at distance 1 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83A1`, N2_CACHE_HIT_RD, Cache at distance 2, demand access hit, read

The counter counts each cache hit at distance 2 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 2 represents a further distance of travel than cache distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A2`, N3_CACHE_HIT_RD, Cache at distance 3, demand access hit, read

The counter counts each cache hit at distance 3 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 3 represents a further distance of travel than cache distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A3`, N4_CACHE_HIT_RD, Cache at distance 4, demand access hit, read

The counter counts each cache hit at distance 4 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 4 represents a further distance of travel than cache distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A4`, N1_CACHE_HIT_PRFM, Cache at distance 1, software prefetch hit

The counter counts each cache hit at distance 1 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A5`, N2_CACHE_HIT_PRFM, Cache at distance 2, software prefetch hit

The counter counts each cache hit at distance 2 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 2 represents a further distance of travel than cache distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A6`, N3_CACHE_HIT_PRFM, Cache at distance 3, software prefetch hit

The counter counts each cache hit at distance 3 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 3 represents a further distance of travel than cache distance 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A7`, N4_CACHE_HIT_PRFM, Cache at distance 4, software prefetch hit

The counter counts each cache hit at distance 4 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 4 represents a further distance of travel than cache distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A8`, N1_CACHE1_HIT_RD, Cache type 1 at distance 1 demand access hit, read

The counter counts each demand read counted by N1_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83A9`, N2_CACHE1_HIT_RD, Cache type 1 at distance 2 demand access hit, read

The counter counts each demand read counted by N2_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AA`, N3_CACHE1_HIT_RD, Cache type 1 at distance 3 demand access hit, read

The counter counts each demand read counted by N3_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AB`, N4_CACHE1_HIT_RD, Cache type 1 at distance 4 demand access hit, read

The counter counts each demand read counted by N4_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AC`, N1_CACHE1_HIT_PRFM, Cache type 1 at distance 1 software prefetch hit

The counter counts each software prefetch counted by N1_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AD`, N2_CACHE1_HIT_PRFM, Cache type 1 at distance 2 software prefetch hit

The counter counts each software prefetch counted by N2_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83AE, N3_CACHE1_HIT_PRFM, Cache type 1 at distance 3 software prefetch hit

The counter counts each software prefetch counted by N3_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83AF, N4_CACHE1_HIT_PRFM, Cache type 1 at distance 4 software prefetch hit

The counter counts each software prefetch counted by N4_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B0, N1_CACHE2_HIT_RD, Cache type 2 at distance 1 demand access hit, read

The counter counts each demand read counted by N1_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B1, N2_CACHE2_HIT_RD, Cache type 2 at distance 2 demand access hit, read

The counter counts each demand read counted by N2_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B2, N3_CACHE2_HIT_RD, Cache type 2 at distance 3 demand access hit, read

The counter counts each demand read counted by N3_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B3, N4_CACHE2_HIT_RD, Cache type 2 at distance 4 demand access hit, read

The counter counts each demand read counted by N4_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B4, N1_CACHE2_HIT_PRFM, Cache type 2 at distance 1 software prefetch hit

The counter counts each software prefetch counted by N1_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B5, N2_CACHE2_HIT_PRFM, Cache type 2 at distance 2 software prefetch hit

The counter counts each software prefetch counted by N2_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83B6, N3_CACHE2_HIT_PRFM, Cache type 2 at distance 3 software prefetch hit

The counter counts each software prefetch counted by N3_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B7, N4_CACHE2_HIT_PRFM, Cache type 2 at distance 4 software prefetch hit

The counter counts each software prefetch counted by N4_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B8, N1_CACHE3_HIT_RD, Cache type 3 at distance 1 demand access hit, read

The counter counts each demand read counted by N1_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83B9, N2_CACHE3_HIT_RD, Cache type 3 at distance 2 demand access hit, read

The counter counts each demand read counted by N2_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83BA, N3_CACHE3_HIT_RD, Cache type 3 at distance 3 demand access hit, read

The counter counts each demand read counted by N3_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83BB, N4_CACHE3_HIT_RD, Cache type 3 at distance 4 demand access hit, read

The counter counts each demand read counted by N4_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83BC, N1_CACHE3_HIT_PRFM, Cache type 3 at distance 1 software prefetch hit

The counter counts each software prefetch counted by N1_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83BD, N2_CACHE3_HIT_PRFM, Cache type 3 at distance 2 software prefetch hit

The counter counts each software prefetch counted by N2_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83BE, N3_CACHE3_HIT_PRFM, Cache type 3 at distance 3 software prefetch hit

The counter counts each software prefetch counted by N3_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83BF, N4_CACHE3_HIT_PRFM, Cache type 3 at distance 4 software prefetch hit

The counter counts each software prefetch counted by N4_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83C0, N1_CACHE_HITM_RD, Cache at distance 1, demand access hit modified, read

The counter counts each demand read counted by N1_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 1.

0x83C1, N2_CACHE_HITM_RD, Cache at distance 2, demand access hit modified, read

The counter counts each demand read counted by N2_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 2.

0x83C2, N3_CACHE_HITM_RD, Cache at distance 3, demand access hit modified, read

The counter counts each demand read counted by N3_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 3.

0x83C3, N4_CACHE_HITM_RD, Cache at distance 4, demand access hit modified, read

The counter counts each demand read counted by N4_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 4.

0x83C4, N1_LFB_HIT_RD, Cache at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by N1_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x83C5, N2_LFB_HIT_RD, Cache at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by N2_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x83C6, N3_LFB_HIT_RD, Cache at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by N3_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83C7, N4_LFB_HIT_RD, Cache at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by N4_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x83C8, N1_CACHE1_HITM_RD, Cache type 1 at distance 1 demand access hit modified, read

The counter counts each demand read counted by N1_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83C9, N2_CACHE1_HITM_RD, Cache type 1 at distance 2 demand access hit modified, read

The counter counts each demand read counted by N2_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83CA, N3_CACHE1_HITM_RD, Cache type 1 at distance 3 demand access hit modified, read

The counter counts each demand read counted by N3_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83CB, N4_CACHE1_HITM_RD, Cache type 1 at distance 4 demand access hit modified, read

The counter counts each demand read counted by N4_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83CC, N1_LFB1_HIT_RD, Cache type 1 at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by N1_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83CD, N2_LFB1_HIT_RD, Cache type 1 at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by N2_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83CE, N3_LFB1_HIT_RD, Cache type 1 at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by N3_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83CF, N4_LFB1_HIT_RD, Cache type 1 at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by N4_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D0, N1_CACHE2_HITM_RD, Cache type 2 at distance 1 demand access hit modified, read

The counter counts each demand read counted by N1_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D1, N2_CACHE2_HITM_RD, Cache type 2 at distance 2 demand access hit modified, read

The counter counts each demand read counted by N2_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D2, N3_CACHE2_HITM_RD, Cache type 2 at distance 3 demand access hit modified, read

The counter counts each demand read counted by N3_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D3, N4_CACHE2_HITM_RD, Cache type 2 at distance 4 demand access hit modified, read

The counter counts each demand read counted by N4_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D4, N1_LFB2_HIT_RD, Cache type 2 at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by N1_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83D5, N2_LFB2_HIT_RD, Cache type 2 at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by N2_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D6, N3_LFB2_HIT_RD, Cache type 2 at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by N3_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D7, N4_LFB2_HIT_RD, Cache type 2 at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by N4_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D8, N1_CACHE3_HITM_RD, Cache type 3 at distance 1 demand access hit modified, read

The counter counts each demand read counted by N1_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83D9, N2_CACHE3_HITM_RD, Cache type 3 at distance 2 demand access hit modified, read

The counter counts each demand read counted by N2_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83DA, N3_CACHE3_HITM_RD, Cache type 3 at distance 3 demand access hit modified, read

The counter counts each demand read counted by N3_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83DB, N4_CACHE3_HITM_RD, Cache type 3 at distance 4 demand access hit modified, read

The counter counts each demand read counted by N4_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83DC, N1_LFB3_HIT_RD, Cache type 3 at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by N1_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83DD, N2_LFB3_HIT_RD, Cache type 3 at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by N2_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83DE, N3_LFB3_HIT_RD, Cache type 3 at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by N3_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83DF, N4_LFB3_HIT_RD, Cache type 3 at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by N4_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83E0, N1_MEM_RD, Access to memory at distance 1, demand access, read

The counter counts each demand read to memory at distance 1.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x83E1, N2_MEM_RD, Access to memory at distance 2, demand access, read

The counter counts each demand read to memory at distance 2.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 2 represents a further distance of travel than distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83E2`, N3_MEM_RD, Access to memory at distance 3, demand access, read

The counter counts each demand read to memory at distance 3.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 3 represents a further distance of travel than distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E3`, N4_MEM_RD, Access to memory at distance 4, demand access, read

The counter counts each demand read to memory at distance 4.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 4 represents a further distance of travel than distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E4`, N1_MEM_PRFM, Access to memory at distance 1, software prefetch

The counter counts each software prefetch to memory at distance 1.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E5`, N2_MEM_PRFM, Access to memory at distance 2, software prefetch

The counter counts each software prefetch to memory at distance 2.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 2 represents a further distance of travel than distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E6`, N3_MEM_PRFM, Access to memory at distance 3, software prefetch

The counter counts each software prefetch to memory at distance 3.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 3 represents a further distance of travel than distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83E7, N4_MEM_PRFM, Access to memory at distance 4, software prefetch

The counter counts each software prefetch to memory at distance 4.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 4 represents a further distance of travel than distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x83E8, N1_MEM1_RD, Access to type 1 memory at distance 1, demand access, read

The counter counts each demand read counted by N1_MEM_RD to a first memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

For example, the first memory type might be commodity DRAM connected to a device. However, this must be the same for all devices in the system, such that, for example, N1_MEM1_RD refers to commodity DRAM for a PE at distance 1 from the memory, regardless of which device the PE is located on, and N4_MEM1_RD refers to the same memory for any PE located at distance 4. This means that when a process counting the N1_MEM1_RD event migrates between PEs on different devices, its relative view of the first memory type remains unchanged, even if the distance changes.

0x83E9, N2_MEM1_RD, Access to type 1 memory at distance 2, demand access, read

The counter counts each demand read counted by N2_MEM_RD to a first memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

0x83EA, N3_MEM1_RD, Access to type 1 memory at distance 3, demand access, read

The counter counts each demand read counted by N3_MEM_RD to a first memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

0x83EB, N4_MEM1_RD, Access to type 1 memory at distance 4, demand access, read

The counter counts each demand read counted by N4_MEM_RD to a first memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

0x83EC, N1_MEM1_PRFM, Access to type 1 memory at distance 1, software prefetch

The counter counts each software prefetch counted by N1_MEM_PRFM to a first memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83ED, N2_MEM1_PRFM, Access to type 1 memory at distance 2, software prefetch

The counter counts each software prefetch counted by N2_MEM_PRFM to a first memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83EE, N3_MEM1_PRFM, Access to type 1 memory at distance 3, software prefetch

The counter counts each software prefetch counted by N3_MEM_PRFM to a first memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83EF, N4_MEM1_PRFM, Access to type 1 memory at distance 4, software prefetch

The counter counts each software prefetch counted by N4_MEM_PRFM to a first memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83F0, N1_MEM2_RD, Access to type 2 memory at distance 1, demand access, read

The counter counts each demand read counted by N1_MEM_RD to a second memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

For example, the second memory type might be bulk expansion memory connected to a device. However, this must be the same for all devices in the system, such that, for example, N1_MEM2_RD refers to bulk expansion memory for a PE at distance 1 from the memory, regardless of which device the PE is located on, and N4_MEM2_RD refers to the same memory for any PE located at distance 4. This means that when a process counting the N1_MEM2_RD event migrates between PEs on different devices, its relative view of the second memory type remains unchanged, even if the distance changes.

0x83F1, N2_MEM2_RD, Access to type 2 memory at distance 2, demand access, read

The counter counts each demand read counted by N2_MEM_RD to a second memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

0x83F2, N3_MEM2_RD, Access to type 2 memory at distance 3, demand access, read

The counter counts each demand read counted by N3_MEM_RD to a second memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

0x83F3, N4_MEM2_RD, Access to type 2 memory at distance 4, demand access, read

The counter counts each demand read counted by N4_MEM_RD to a second memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

0x83F4, N1_MEM2_PRFM, Access to type 2 memory at distance 1, software prefetch

The counter counts each software prefetch counted by N1_MEM_PRFM to a second memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83F5, N2_MEM2_PRFM, Access to type 2 memory at distance 2, software prefetch

The counter counts each software prefetch counted by N2_MEM_PRFM to a second memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83F6, N3_MEM2_PRFM, Access to type 2 memory at distance 3, software prefetch

The counter counts each software prefetch counted by N3_MEM_PRFM to a second memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83F7, N4_MEM2_PRFM, Access to type 2 memory at distance 4, software prefetch

The counter counts each software prefetch counted by N4_MEM_PRFM to a second memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83F8, N1_MEM3_RD, Access to type 3 memory at distance 1, demand access, read

The counter counts each demand read counted by N1_MEM_RD to a third memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

For example, the third memory type might be high-bandwidth memory connected to a device. However, this must be the same for all devices in the system, such that, for example, N1_MEM3_RD refers to high-bandwidth memory for a PE at distance 1 from the memory, regardless of which device the PE is located on, and N4_MEM3_RD refers to the same memory for any PE located at distance 4. This means that when a process counting the N1_MEM3_RD event migrates between PEs on different devices, its relative view of the third memory type remains unchanged, even if the distance changes.

0x83F9, N2_MEM3_RD, Access to type 3 memory at distance 2, demand access, read

The counter counts each demand read counted by N2_MEM_RD to a third memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

0x83FA, N3_MEM3_RD, Access to type 3 memory at distance 3, demand access, read

The counter counts each demand read counted by N3_MEM_RD to a third memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

0x83FB, N4_MEM3_RD, Access to type 3 memory at distance 4, demand access, read

The counter counts each demand read counted by N4_MEM_RD to a third memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

0x83FC, N1_MEM3_PRFM, Access to type 3 memory at distance 1, software prefetch

The counter counts each software prefetch counted by N1_MEM_PRFM to a third memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83FD, N2_MEM3_PRFM, Access to type 3 memory at distance 2, software prefetch

The counter counts each software prefetch counted by N2_MEM_PRFM to a third memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x83FE, N3_MEM3_PRFM, Access to type 3 memory at distance 3, software prefetch

The counter counts each software prefetch counted by N3_MEM_PRFM to a third memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x83FF, N4_MEM3_PRFM, Access to type 3 memory at distance 4, software prefetch

The counter counts each software prefetch counted by N4_MEM_PRFM to a third memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

0x8400, ISNP_HIT_N1_RD, Snoop hit in cache at distance 1, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8401, ISNP_HIT_N2_RD, Snoop hit in cache at distance 2, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8402, ISNP_HIT_N3_RD, Snoop hit in cache at distance 3, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8403, ISNP_HIT_N4_RD, Snoop hit in cache at distance 4, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8404, DSNP_HIT_N1_RD, Snoop hit in cache at distance 1, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8405, DSNP_HIT_N2_RD, Snoop hit in cache at distance 2, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8406, DSNP_HIT_N3_RD, Snoop hit in cache at distance 3, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8407, DSNP_HIT_N4_RD, Snoop hit in cache at distance 4, demand data read

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8408, DSNP_HIT_N1_WR, Snoop hit in cache at distance 1, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8409, DSNP_HIT_N2_WR, Snoop hit in cache at distance 2, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x840A, DSNP_HIT_N3_WR, Snoop hit in cache at distance 3, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x840B, DSNP_HIT_N4_WR, Snoop hit in cache at distance 4, demand data write

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x840C, DSNP_HIT_N1_RW, Snoop hit in cache at distance 1, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x840D, DSNP_HIT_N2_RW, Snoop hit in cache at distance 2, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x840E, DSNP_HIT_N3_RW, Snoop hit in cache at distance 3, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x840F, DSNP_HIT_N4_RW, Snoop hit in cache at distance 4, demand data access

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8410, DSNP_HIT_N1_PRFM, Snoop hit in cache at distance 1, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8411, DSNP_HIT_N2_PRFM, Snoop hit in cache at distance 2, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8412, DSNP_HIT_N3_PRFM, Snoop hit in cache at distance 3, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8413, DSNP_HIT_N4_PRFM, Snoop hit in cache at distance 4, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8414, DSNP_HIT_N1_HWPRF, Snoop hit in cache at distance 1, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8415, DSNP_HIT_N2_HWPRF, Snoop hit in cache at distance 2, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8416, DSNP_HIT_N3_HWPRF, Snoop hit in cache at distance 3, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8417, DSNP_HIT_N4_HWPRF, Snoop hit in cache at distance 4, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8418, ISNP_HIT_N1_PRFM, Snoop hit in cache at distance 1, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8419, ISNP_HIT_N2_PRFM, Snoop hit in cache at distance 2, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x841A, ISNP_HIT_N3_PRFM, Snoop hit in cache at distance 3, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x841B, ISNP_HIT_N4_PRFM, Snoop hit in cache at distance 4, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x841C, ISNP_HIT_N1_HWPRF, Snoop hit in cache at distance 1, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x841D, ISNP_HIT_N2_HWPRF, Snoop hit in cache at distance 2, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x841E, ISNP_HIT_N3_HWPRF, Snoop hit in cache at distance 3, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x841F, ISNP_HIT_N4_HWPRF, Snoop hit in cache at distance 4, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8420, DSNP_HIT_N1_PRF, Snoop hit in cache at distance 1, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8421, DSNP_HIT_N2_PRF, Snoop hit in cache at distance 2, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8422, DSNP_HIT_N3_PRF, Snoop hit in cache at distance 3, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8423, DSNP_HIT_N4_PRF, Snoop hit in cache at distance 4, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8424, ISNP_HIT_N1_PRF, Snoop hit in cache at distance 1, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8425, ISNP_HIT_N2_PRF, Snoop hit in cache at distance 2, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8426, ISNP_HIT_N3_PRF, Snoop hit in cache at distance 3, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8427, ISNP_HIT_N4_PRF, Snoop hit in cache at distance 4, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8428, ISNP_HIT_N1, Snoop hit in cache at distance 1, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x8429, ISNP_HIT_N2, Snoop hit in cache at distance 2, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x842A, ISNP_HIT_N3, Snoop hit in cache at distance 3, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x842B, ISNP_HIT_N4, Snoop hit in cache at distance 4, instruction access

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

0x842C, DSNP_HIT_N1, Snoop hit in cache at distance 1, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842D`, DSNP_HIT_N2, Snoop hit in cache at distance 2, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842E`, DSNP_HIT_N3, Snoop hit in cache at distance 3, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842F`, DSNP_HIT_N4, Snoop hit in cache at distance 4, data access

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8431`, ASE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise or reduction

The counter counts each speculatively executed floating-point treewise reduction or pairwise operation counted by ASE_SVE_FP_VREDUCE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXNMV, FMAXP, FMAXV, FMINNMP, FMINNMV, FMINP, or FMINV.

Note

Treewise reduction instructions might be executed and counted as multiple pairwise operations.

`0x8432`, SVE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, SVE pairwise add step or pairwise reduce step

The counter counts each speculatively executed floating-point pairwise add step or pairwise reduce step operation counted by SVE_FP_VREDUCE_SPEC due to any of the following instructions:

- SVE2: FADDP, FMAXNMP, FMAXP, FMINNMP, or FMINP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8434, DSNP_HITM_N1_RD, Snoop hit in cache at distance 1, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N1_RD that accesses a previously modified cache location.

0x8435, DSNP_HITM_N2_RD, Snoop hit in cache at distance 2, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N2_RD that accesses a previously modified cache location.

0x8436, DSNP_HITM_N3_RD, Snoop hit in cache at distance 3, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N3_RD that accesses a previously modified cache location.

0x8437, DSNP_HITM_N4_RD, Snoop hit in cache at distance 4, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N4_RD that accesses a previously modified cache location.

0x8438, DSNP_HITM_N1_WR, Snoop hit in cache at distance 1, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N1_WR that accesses a previously modified cache location.

0x8439, DSNP_HITM_N2_WR, Snoop hit in cache at distance 2, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N2_WR that accesses a previously modified cache location.

0x843A, DSNP_HITM_N3_WR, Snoop hit in cache at distance 3, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N3_WR that accesses a previously modified cache location.

0x843B, DSNP_HITM_N4_WR, Snoop hit in cache at distance 4, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N4_WR that accesses a previously modified cache location.

0x843C, DSNP_HITM_N1_RW, Snoop hit in cache at distance 1, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N1_RW that accesses a previously modified cache location.

0x843D, DSNP_HITM_N2_RW, Snoop hit in cache at distance 2, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N2_RW that accesses a previously modified cache location.

0x843E, DSNP_HITM_N3_RW, Snoop hit in cache at distance 3, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N3_RW that accesses a previously modified cache location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x843F, DSNP_HITM_N4_RW, Snoop hit in cache at distance 4, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N4_RW that accesses a previously modified cache location.

0x8440, ASE_FP_SP_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

0x8442, ASE_FP_HP_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

0x8443, ASE_FP_BF16_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

0x8444, ASE_FP_FP8_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

0x8448, ASE_SVE_FP_SP_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

0x844A, ASE_SVE_FP_HP_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

0x844B, ASE_SVE_FP_BF16_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

0x844C, ASE_SVE_FP_FP8_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8450, SE_FP_SP_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

0x8452, SE_FP_HP_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

0x8453, SE_FP_BF16_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

0x8454, SE_FP_FP8_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

0x8458, SME_FP_SP_MIN_SPEC, SME data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each SME data-processing operation counted by SE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

0x845A, SME_FP_HP_MIN_SPEC, SME data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each SME data-processing operation counted by SE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

0x845B, SME_FP_BF16_MIN_SPEC, SME data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each SME data-processing operation counted by SE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

0x845C, SME_FP_FP8_MIN_SPEC, SME data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each SME data-processing operation counted by SE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

0x8460, SVE_FP_SP_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8462, SVE_FP_HP_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

0x8463, SVE_FP_BF16_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

0x8464, SVE_FP_FP8_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

0x846B, FP_BF16_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was BFloat16 floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x8470, FP_SP_MIN_SPEC, Floating-point operation speculatively executed, smallest type is single-precision floating-point

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was single-precision floating-point.

0x8472, FP_HP_MIN_SPEC, Floating-point operation speculatively executed, smallest type is half-precision floating-point

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was half-precision floating-point.

0x8473, FP_BF16_MIN_SPEC, Floating-point operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

0x8474, FP_FP8_MIN_SPEC, Floating-point operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was 8-bit floating-point.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0x8480, FP_SP_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was single-precision floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x8482, FP_HP_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was half-precision floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x8483, FP_BF16_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was BFloat16 floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x8484, FP_FP8_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point

The counter increments by *v* for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was 8-bit floating-point.

Where *v* is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x8488, FP_SP_SCALE_MIN_OPS_SPEC, Scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point

The counter increments by *v* for each speculatively executed scalable element arithmetic operation, due to an instruction where the smallest type was single-precision floating-point.

Where *v* is a value such that ($v \times (VL \div 128)$) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

0x848A, FP_HP_SCALE_MIN_OPS_SPEC, Scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point

The counter increments by *v* for each speculatively executed scalable element arithmetic operation, due to an instruction where the smallest type was half-precision floating-point.

Where *v* is a value such that ($v \times (VL \div 128)$) is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

`0x848B`, FP_BF16_SCALE_MIN_OPS_SPEC, Scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point

The counter increments by v for each speculatively executed scalable element arithmetic operation, due to an instruction where the smallest type was BFloat16 floating-point.

Where v is a value such that $(v \times (VL \div 128))$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

`0x848C`, FP_FP8_SCALE_MIN_OPS_SPEC, Scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point

The counter increments by v for each speculatively executed scalable element arithmetic operation, due to an instruction where the smallest type was 8-bit floating-point.

Where v is a value such that $(v \times (VL \div 128))$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE2_OPS_SPEC.

`0x8490`, FP_SP_SCALE2_MIN_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, smallest type is single-precision floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the smallest type was single-precision floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

`0x8492`, FP_HP_SCALE2_MIN_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, smallest type is half-precision floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the smallest type was half-precision floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

`0x8493`, FP_BF16_SCALE2_MIN_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, smallest type is BFloat16 floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the smallest type was BFloat16 floating-point.

Where v is a value such that $(v \times (VL \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8494`, FP_FP8_SCALE2_MIN_OPS_SPEC, Scalable tile arithmetic operations speculatively executed, smallest type is 8-bit floating-point

The counter increments by v for each speculatively executed scalable tile arithmetic operation, due to an instruction where the smallest type was 8-bit floating-point.

Where v is a value such that $(v \times (\mathrm{VL} \div 128)^2)$ is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_FIXED_OPS_SPEC or FP_SCALE_OPS_SPEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.4 Cycle event counting

The CPU_CYCLES event and the cycle counter, PMCCNTR, count cycles. The duration of a cycle is subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions.

It is IMPLEMENTATION SPECIFIC whether CPU_CYCLES and PMCCNTR count when the PE is in WFI or WFE state, even if the clocks are not stopped.

In addition, events such as STALL, STALL_FRONTEND and STALL_BACKEND that are defined to only count cycles that are counted by the CPU_CYCLES event have the same limitation.

D14.4.1 Cycle event counting in multithreaded implementations

Multithreaded implementations can have various forms, some examples of these are:

- *Simultaneous Multithreading* (SMT), where every PE thread is active on every Processor cycle.
- *Fine-grained Multithreading* (FGMT), also known as a Barrel processor, where one PE thread is active on each Processor cycle, and this changes regularly.
- *Switch on Event Multithreading* (SoEMT), also known as *Coarse-grained Multithreading* (CGMT), where high latency events cause the processor to switch the active PE thread.

In the above examples, active means that the PE might execute the instructions. A PE can be active but not executing instructions when no instruction is available or because of limited execution resources.

It is IMPLEMENTATION SPECIFIC whether a thread is active when the thread is in WFE or WFI state. This applies for all forms of multithreaded implementation.

When the PMU implementation supports multithreading, the Effective value of PMEVTYPER<n>_EL0.MT bit is 0, and the thread is not in WFE or WFI state, the CPU_CYCLES event only counts Processor cycles when the thread is active. For the example multithreaded implementations, this means that, if the event counter is enabled, event counting is not prohibited, and the thread is not in WFE or WFI state:

- For an SMT implementation, the CPU_CYCLES event counts every Processor cycle.
- For a particular FGMT implementation, that alternates between two threads on each Processor cycle, the CPU_CYCLES event counts every other Processor cycle.
- For a particular SoEMT implementation, that is waiting for a long latency operation, the CPU_CYCLES event does not count Processor cycles, as the PE thread is not active.

It is IMPLEMENTATION SPECIFIC whether CPU_CYCLES event counts Processor cycles when the Effective value of PMEVTYPER<n>_EL0.MT bit is 0, the thread is active, and the thread is in WFE or WFI state.

If the Effective value of PMEVTYPER<n>_EL0.MT bit is 1 and at least one thread is not in WFE or WFI state, then the CPU_CYCLES event counts each Processor cycle, and can only count a maximum of one for each Processor cycle.

If the Effective value of PMEVTYPER<n>_EL0.MT bit is 1, then all of the following apply for events that count PE cycles when a condition applies:

- The event does not count Processor cycles when all threads are in WFE or WFI state.
- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycle when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event. The second condition is any condition in addition to this. For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache. For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The cycle counter, PMCCNTR, is not affected by whether the thread is active or inactive. When enabled, PMCCNTR counts every processor cycle.

For more information about when the Effective value of PMEVTYPER<n>_EL0.MT is 0, see:

- MDCR_EL2.MTPME.
- MDCR_EL3.MTPME.
- HDCR.MTPME.
- SDCR.MTPME.

D14.4.2 Time as measured by the Performance Monitors cycle counter

The Performance Monitors cycle counter, accessed through PMCCNTR_EL0 or PMCCNTR, increments from the hardware processor clock, not PE clock cycles.

The relationship between the count recorded by the Performance Monitors cycle counter and the passage of real time is IMPLEMENTATION DEFINED.

See Controlling the PMU counters for information about when the cycle counter does not increment.

Note

- This means that, in an implementation where PEs are multithreaded, when enabled, the cycle counter continues to increment across all PEs, rather than only counting cycles for which the current PE is active.
- Although the architecture requires that direct reads of PMCCNTR_EL0 or PMCCNTR occur in program order, there is no requirement that the count increments between two such reads. Even when the counter is incrementing on every clock cycle, software might need check that the difference between two reads of the counter is nonzero.

  The architecture requires that an indirect write to the PMCCNTR_EL0 or PMCCNTR is observable to direct reads of the register in finite time. The counter increments from the hardware processor clock are indirect writes to these registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.5 Meaningful combinations of common events

The architecture highlights some meaningful combinations of common events. The following tables list the highlighted combinations:

- Refill events and associated access events.
- Latency events.
- Cache hit events and associated data source events.
- Cache line state tracking events.
- TLB events.
- At-retirement events.
- Speculatively executed operation events.
- Bytes loaded and stored events.
- Overall vector utilization events.
- Vector loop efficiency events.

D14.5.1 Refill events and associated access events

I_{GSMLS} The following are meaningful combinations of refill events and associated access events:

Table D14-4 Meaningful combinations of refill events and associated access events

Numerator	Denominator	Ratio
0x0001 L1I_CACHE_REFILL	0x0014 L1I_CACHE	Attributable Level 1 instruction cache refill rate
0x0002 L1I_TLB_REFILL	0x0026 L1I_TLB	Attributable Level 1 instruction TLB refill rate
0x0003 L1D_CACHE_REFILL	0x0004 L1D_CACHE	Attributable Level 1 data or unified cache refill rate
0x0005 L1D_TLB_REFILL	0x0025 L1D_TLB	Attributable Level 1 data or unified TLB refill rate
0x0017 L2D_CACHE_REFILL	0x0016 L2D_CACHE	Attributable Level 2 data or unified cache refill rate
0x0028 L2I_CACHE_REFILL	0x0027 L2I_CACHE	Attributable Level 2 instruction cache refill rate
0x002A L3D_CACHE_REFILL	0x002B L3D_CACHE	Attributable Level 3 data or unified cache refill rate
0x002D L2D_TLB_REFILL	0x002F L2D_TLB	Attributable Level 2 data or unified TLB refill rate
0x002E L2I_TLB_REFILL	0x0030 L2I_TLB	Attributable Level 2 instruction TLB refill rate
0x0019 BUS_ACCESS	0x001D BUS_CYCLES	Attributable Bus accesses per cycle
0x0033 LL_CACHE_MISS	0x0032 LL_CACHE	Attributable Last Level data or unified cache refill rate
0x0034 DTLB_WALK	0x0025 L1D_TLB	Attributable data TLB miss rate
0x0035 ITLB_WALK	0x0026 L1I_TLB	Attributable instruction TLB miss rate
0x0037 LL_CACHE_MISS_RD	0x0036 LL_CACHE_RD	Attributable memory read operation miss rate
0x0038 REMOTE_ACCESS_RD	0x0031 REMOTE_ACCESS	Attributable read accesses to another socket in a multi-socket system
0x0042 L1D_CACHE_REFILL_RD	0x0040 L1D_CACHE_RD	Attributable Level 1 cache refill rate, read
0x0043 L1D_CACHE_REFILL_WR	0x0041 L1D_CACHE_WR	Attributable Level 1 cache refill rate, write
0x004C L1D_TLB_REFILL_RD	0x004E L1D_TLB_RD	Attributable Level 1 TLB refill rate, read
0x004D L1D_TLB_REFILL_WR	0x004F L1D_TLB_WR	Attributable Level 1 TLB refill rate, write

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Numerator	Denominator	Ratio
0x0052 L2D_CACHE_REFILL_RD	0x0050 L2D_CACHE_RD	Attributable Level 2 data cache refill rate, read
0x0053 L2D_CACHE_REFILL_WR	0x0051 L2D_CACHE_WR	Attributable Level 2 data cache refill rate, write
0x005C L2D_TLB_REFILL_RD	0x005E L2D_TLB_RD	Attributable Level 2 data TLB refill rate, read
0x005D L2D_TLB_REFILL_WR	0x005F L2D_TLB_WR	Attributable Level 2 data TLB refill rate, write
0x00A2 L3D_CACHE_REFILL_RD	0x00A0 L3D_CACHE_RD	Attributable Level 3 data cache refill rate, read
0x00A3 L3D_CACHE_REFILL_WR	0x00A1 L3D_CACHE_WR	Attributable Level 3 data cache refill rate, write

D14.5.2 Latency events

I_{ZHFDC} The following are meaningful combination of latency events:

Table D14-5 Meaningful combinations of latency events

Numerator	Denominator	Ratio
0x8120 INST_FETCH_PERCYC	0x8124 INST_FETCH	Mean duration of instruction fetch events in processor cycles
0x8121 MEM_ACCESS_RD_PERCYC	0x0066 MEM_ACCESS_RD	Mean duration of memory read access events in processor cycles
0x8125, BUS_REQ_RD_PERCYC	0x818D, BUS_REQ_RD	Bus read transaction average latency
0x8128 DTLB_WALK_PERCYC	0x0034 DTLB_WALK	Mean duration of data or unified TLB walk events in processor cycles
0x8129 ITLB_WALK_PERCYC	0x0035 ITLB_WALK	Mean duration of instruction TLB walk events in processor cycles
0x8320, L1D_CACHE_REFILL_PERCYC	0x0003, L1D_CACHE_REFILL	Level 1 data cache refill average duration

D14.5.3 Cache hit events and associated data source events

I_{HMZQY} The following are meaningful combination of cache hit events:

Table D14-6 Meaningful combinations of cache hit events and associated data source events

Numerator	Denominator	Ratio
0x820C, L1D_CACHE_HIT_PRFM [a]	0x8142, L1D_CACHE_PRFM [a]	Level 1 data or unified cache software prefetch hit ratio[a]
0x81CC, L1D_CACHE_HIT_RW [a]	0x82A0, MEM_ACCESS_RW	Level 1 data or unified cache demand cache hit rate[a]
0x81CF, LL_CACHE_HIT_RW	0x82A0, MEM_ACCESS_RW	Last level cache demand cache hit rate
0x8232, LOCAL_MEM_RW	0x82A0, MEM_ACCESS_RW	Local memory demand access rate
0x823B, REMOTE_MEM_RW	0x82A0, MEM_ACCESS_RW	Remote memory demand access rate
0x821D, L2D_CACHE_HITM_RW [a]	0x81CD, L2D_CACHE_HIT_RW [a]	Level 2 data or unified cache demand cache hit rate[a]

a. Similar ratios can be defined for other caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.5.4 Cache line state tracking events

I_{LGBBR} The following are meaningful combination of cache line stake tracking events:

Table D14-7 Meaningful combinations of cache line state tracking events

Ratio	Meaning
1 - (0x81EC, L1D_CACHE_HIT_RW_FHWPRF ÷ 0x8154, L1D_CACHE_HWPRF)[a]	Level 1 data or unified cache data fetched by hardware prefetcher not used[a]
1 - (0x81DC, L1D_CACHE_HIT_RW_FPRFM ÷ 0x8142, L1D_CACHE_PRFM)[a]	Level 1 data or unified cache data fetched by software prefetch not used[a]
0x825C, L1D_LFB_HIT_RW_FPRFM ÷ 0x8142, L1D_CACHE_PRFM [a]	Level 1 data or unified cache data fetched late by software prefetch[a]

a. Similar ratios can be defined for other caches.

D14.5.5 TLB events

I_{PHZQV} The following are meaningful combination of TLB events:

Table D14-8 Meaningful combinations of TLB events

Numerator	Denominator	Ratio
0x8188, DTLB_WALK_BLOCK [a]	0x0034, DTLB_WALK [a]	Data TLB block translation table walk fraction[a]
0x818A, DTLB_WALK_PAGE [a]	0x0034, DTLB_WALK [a]	Data TLB page translation table walk fraction[a]

a. Similar ratios can be defined for instruction TLB accesses.

D14.5.6 At-retirement events

I_{CYQZJ} The following are meaningful combinations of at-retirement events:

Table D14-9 Meaningful combinations of at-retirement events

Numerator	Denominator	Ratio
0x82B0, L1I_CACHE_HIT_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction hit Level 1 instruction cache
0x82E0, L2I_CACHE_MISS_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction missed Level 2 instruction cache
0x82EC, L1D_CACHE_HIT_LDST_RETIRED	0x0008, INST_RETIRED	Architecturally executed load or store instruction hit in Level 1 data cache
0x82C9, L2D_CACHE_HITM_ST_RETIRED	0x0008, INST_RETIRED	Architecturally executed store instruction hit modified data in Level 2 data cache
0x82B7, LL_CACHE_MISS_LD_RETIRED	0x0008, INST_RETIRED	Architecturally executed load instruction missed in Last level cache

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Numerator	Denominator	Ratio
0x82F0, ITLB_HIT_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction which hit in the ITLB
0x82F7, DTLB_WALK_LDST_RETIRED	0x82AA, LDST_ANY_RETIRED	Architecturally executed LDST instructions DTLB miss fraction
0x82AF, CRYPTO_RETIRED	0x0008, INST_RETIRED	Architecturally executed cryptographic data processing fraction

D14.5.7 Speculatively executed operation events

I_{RDHPB} The number of speculatively executed operations performed on individual scalar values, assuming that all SVE vector elements are Active, can be determined from a pair of event counters. Combined multiply-add and multiply-subtract instructions are counted as two operations per element.

Table D14-10 Meaningful combinations of speculatively executed operation events

Ratio	Meaning
0x80C0, FP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C1, FP_FIXED_OPS_SPEC	Total number of individual floating-point operations (any precision) performed
0x80C2, FP_HP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C3, FP_HP_FIXED_OPS_SPEC	Total number of individual half-precision floating-point operations performed
0x80C4, FP_SP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C5, FP_SP_FIXED_OPS_SPEC	Total number of individual single-precision floating-point operations performed
0x80C6, FP_DP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C7, FP_DP_FIXED_OPS_SPEC	Total number of individual double-precision floating-point operations performed
0x80C8, INT_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C9 INT_FIXED_OPS_SPEC	Total number of individual integer operations performed
0x80CA, LDST_SCALE_OPS_SPEC × VL ÷ 128 + 0x80CB, LDST_FIXED_OPS_SPEC	Total number of individual load/store accesses performed
0x80CC, LD_SCALE_OPS_SPEC × VL ÷ 128 + 0x80CD, LD_FIXED_OPS_SPEC	Total number of individual load accesses performed
0x80CE, ST_SCALE_OPS_SPEC × VL ÷ 128 + 0x80CF, ST_FIXED_OPS_SPEC	Total number of individual store accesses performed

D14.5.8 Bytes loaded and stored events

I_{NYJRH} The number of bytes speculatively loaded from memory or stored to memory, assuming that all SVE vector elements are Active, can be determined from a pair of event counters.

Table D14-11 Meaningful combinations of speculative load or store events

Ratio	Meaning
0x80DA, LDST_SCALE_BYTES_SPEC × VL ÷ 128 + 0x80DB, LDST_FIXED_BYTES_SPEC	Total number of bytes loaded from memory and stored to memory

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Ratio	Meaning
0x80DC, LD_SCALE_BYTES_SPEC × VL ÷ 128 + 0x80DD, LD_FIXED_BYTES_SPEC	Total number of bytes loaded from memory
0x80DE, ST_SCALE_BYTES_SPEC × VL ÷ 128 + 0x80DF, ST_FIXED_BYTES_SPEC	Total number of bytes stored to memory

D14.5.9 Overall vector utilization events

I_{MVTZM}

Vector utilization rates for SVE events which ignore the number of Active elements can be estimated by adjusting them using the following ratios:

Table D14-12 Meaningful combinations of speculative predicated events

Numerator	Denominator	Meaning
0x8076, SVE_PRED_FULL_SPEC	0x8074, SVE_PRED_SPEC	All predicates active
0x8077, SVE_PRED_PARTIAL_SPEC	0x8074, SVE_PRED_SPEC	Partial predicates active
0x8075, SVE_PRED_EMPTY_SPEC	0x8074, SVE_PRED_SPEC	No predicates active

D14.5.10 Vector loop efficiency events

I_{MXYQS}

The effectiveness with which sequential or scalar source loops are vectorized can be estimated using ratios of the SVE_PLOOP_*_SPEC predicated loop events, as shown in the following table:

Table D14-13 Meaningful combinations of speculative sequential or scalar source loops events

Numerator	Denominator	Meaning
0x8072, SVE_PLOOP_ELTS_SPEC	0x8073, SVE_PLOOP_TERM_SPEC	Source level iterations per loop
0x8071, SVE_PLOOP_TEST_SPEC	0x8073, SVE_PLOOP_TERM_SPEC	Vectorized iterations per loop
0x8072, SVE_PLOOP_ELTS_SPEC	0x8071, SVE_PLOOP_TEST_SPEC	Parallelism per vector loop

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.6 Required events

When at least one event counter is implemented, the following common events are implemented:

- `0x0000`, SW_INCR, Instruction architecturally executed, Condition code check pass, software increment.
- `0x0011`, CPU_CYCLES, Cycle.
- When the PE includes a Level 1 data or unified cache:
  - `0x0003`, L1D_CACHE_REFILL, Level 1 data cache refill.
  - `0x0004`, L1D_CACHE, Level 1 data cache access.
- When the PE includes program-flow prediction:
  - `0x0010`, BR_MIS_PRED, Mispredicted or not predicted branch Speculatively executed.
  - `0x0012`, BR_PRED, Predictable branch Speculatively executed.

  Note

  When the PE does not include program-flow prediction, Arm strongly recommends that these events are implemented as described in Common microarchitectural events.

- When FEAT_PMUv3_ICNTR is implemented:
  - `0x0008`, INST_RETIRED, Instruction architecturally executed.
- When FEAT_PMUv3_ICNTR is not implemented, at least one of:
  - `0x0008`, INST_RETIRED, Instruction architecturally executed.
  - `0x001B`, INST_SPEC, Operation Speculatively executed.

  Note

  Arm strongly recommends that event `0x0008` is implemented.

- When FEAT_PMUv3p1 is implemented:
  - `0x0023`, STALL_FRONTEND, No operation issued due to the frontend.
  - `0x0024`, STALL_BACKEND, No operation issued due to the backend.
- When FEAT_SVE or FEAT_SME is implemented, at least one of:
  - `0x8002`, SVE_INST_RETIRED, SVE instruction architecturally retired.
  - `0x8006`, SVE_INST_SPEC, SVE operation speculatively executed.
- When FEAT_SPE is implemented:
  - `0x4000`, SAMPLE_POP, Statistical Profiling sample population.
  - `0x4001`, SAMPLE_FEED, Statistical Profiling sample taken.
  - `0x4002`, SAMPLE_FILTRATE, Statistical Profiling sample filtered.
  - `0x4003`, SAMPLE_COLLISION, Statistical Profiling sample collision.
- When FEAT_PMUv3p4 is implemented:
  - `0x003C`, STALL, No operation sent for execution.
  - When the PE includes a Level 1 data or unified cache:
    - `0x0039`, L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss.
    - `0x0040`, L1D_CACHE_RD, Level 1 data cache read.
  - When the PE includes a Level 1 instruction cache:
    - `0x4006`, L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss.
- When FEAT_SPEv1p2 is implemented:
  - `0x812A`, SAMPLE_FEED_BR, Statistical Profiling sample taken, branch.
  - `0x812B`, SAMPLE_FEED_LD, Statistical Profiling sample taken, load.
  - `0x812C`, SAMPLE_FEED_ST, Statistical Profiling sample taken, store.
  - `0x812D`, SAMPLE_FEED_OP, Statistical Profiling sample taken, matching operation type.
  - `0x812E`, SAMPLE_FEED_EVENT, Statistical Profiling sample taken, matching events.
  - `0x812F`, SAMPLE_FEED_LAT, Statistical Profiling sample taken, exceeding minimum latency.
- When FEAT_ETE is implemented:
  - `0x4010`, TRCEXTOUT0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - 0x4011, TRCEXTOUT1.
  - 0x4012, TRCEXTOUT2.
  - 0x4013, TRCEXTOUT3.
  - If TRCIDR5.NUMEXTINSEL > 0, 0x4018, CTI_TRIGOUT4.
  - If TRCIDR5.NUMEXTINSEL > 1, 0x4019, CTI_TRIGOUT5.
  - If TRCIDR5.NUMEXTINSEL > 2, 0x401A, CTI_TRIGOUT6.
  - If TRCIDR5.NUMEXTINSEL > 3, 0x401B, CTI_TRIGOUT7.
- When FEAT_PMUv3p9 is implemented and the PE includes a Level 1 instruction cache:
  - 0x0014, L1I_CACHE, Level 1 instruction cache access.
- When FEAT_PMUv3_SS is implemented:
  - 0x8127, PMU_SNAPSHOT, Successful PMU capture event.
- When FEAT_SPE_FDS is implemented:
  - 0x8122, SAMPLE_FEED_DS, Statistical Profiling sample taken, selected Data Source.
- When FEAT_SPEv1p4 is implemented:
  - 0x8123, SAMPLE_BUFFER_FULL, Profiling Buffer full.
- When FEAT_SPE_EFT is implemented:
  - 0x8348, SAMPLE_FEED_FP, Statistical Profiling sample taken, floating-point.
  - 0x8349, SAMPLE_FEED_SIMD, Statistical Profiling sample taken, SIMD.
- When FEAT_SME is implemented:
  - At least one of the following is implemented:
    - 0x835A, SME_INST_RETIRED, Instruction architecturally executed, SME.
    - 0x835E, SME_INST_SPEC, Operation speculatively executed, SME.
  - At least one of the following is implemented:
    - 0x8358, SME_RETIRED, Instruction architecturally executed, SME data processing.
    - 0x835C, SME_SPEC, Operation speculatively executed, SME data processing.

When any of the following common events are implemented, all three of them are implemented:

- 0x003D, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend.
- 0x003E, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend.
- 0x003F, STALL_SLOT, No operation sent for execution on a Slot.

Arm strongly recommends that the following events are implemented:

- 0x0021, BR_RETIRED.
- 0x0022, BR_MIS_PRED_RETIRED.
- 0x003A, OP_RETIRED.
- 0x003B, OP_SPEC.
- 0x003D, STALL_SLOT_BACKEND.
- 0x003E, STALL_SLOT_FRONTEND.
- 0x003F, STALL_SLOT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D14.7 IMPLEMENTATION DEFINED event numbers

Arm recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common event numbers, applied to all of their implementations. In general, the recommended approach is for standardization across implementations with common features. However, Arm recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

The Arm architecture guarantees not to define any event prefixed with *IMP_* as part of the standard Arm architecture.

Arm strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Separating each of the STALL_FRONTEND and STALL_SLOT_FRONTEND events to count holes in instruction availability.
- Separating each of the STALL_BACKEND and STALL_SLOT_BACKEND events, to count, for example, cumulative duration of stalls, unavailability of execution resources, or missed superscalar issue opportunities.
- Miss rates for additional levels of caches and TLBs.
- Any external events passed to the PE through an IMPLEMENTATION DEFINED mechanism.
- Cumulative duration of a PSTATE.{A, I, F} interrupt mask set to 1.
- Cumulative occupancy for resource queues, such as data access queues, and entry/exit counts, so that average latencies can be determined, separating out counts for key resources that might exist. An implementation might also provide registers in the IMPLEMENTATION DEFINED space to further extend such counts, for example by specifying a minimum latency for an event to be counted.
- Any other microarchitectural features that the implementer considers are valuable to count.

If FEAT_RME is implemented, IMPLEMENTATION DEFINED events are permitted to count all of:

- *Granule Protection Table* (GPT) accesses.
- For address translations subject to *Granule Protection Checks* (GPC):
  - TLB hits. These are called *GPT-related TLB hits*.
  - TLB misses. These are called *GPT-related TLB misses*.

The range of possible IMPLEMENTATION DEFINED event numbers is described in The PMU event number space and common events.

It is not required that implementations document the behavior of all IMPLEMENTATION DEFINED PMU events, including which PMU event numbers are used for these IMPLEMENTATION DEFINED events. However, Arm strongly recommends that implementations do include and document IMPLEMENTATION DEFINED PMU events. In particular events related to microarchitectural characteristics of the implementation not covered by Common events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D15
The System Performance Monitors Extension

This chapter describes the implementation of the Arm System Performance Monitors that are an OPTIONAL non-invasive debug component. It contains the following sections:

- About the System Performance Monitors.
- System PMU configuration.
- Accessing System PMUs.
- Accessing System PMU registers.
- Accessing System PMU counters.
- Generating System PMU overflow interrupt requests.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D15.1 About the System Performance Monitors

R_{XVHSR} A *System PMU* is a PMU other than the PMU implemented by a PE that is accessible to that PE. For more information on the PMU, see The Performance Monitors Extension.

R_{KGSQX} A System PMU might be accessible to other PEs in the system.

I_{XDHVC} The System Performance Monitors Extension is an OPTIONAL feature of an implementation. The architecture does not require that System PMUs are implemented using FEAT_SPMU. Arm recommends that System PMUs are implemented using FEAT_SPMU or *The CoreSight PMU Architecture*. FEAT_SPMU and *The CoreSight PMU Architecture* share a conceptual model for performance monitors.

I_{DDTRR} The System Performance Monitors Extension supports up to 32 System PMUs.

I_{NYCYF} The basic form of a System PMU is:

- A number of 64-bit event counters, SPMEVCNTR<n>_EL0. Each System PMU can support up to 64 event counters.
- It is IMPLEMENTATION DEFINED whether the event counted by each event counter is fixed or programmable. Event configuration registers, SPMEVTYPER<n>_EL0, SPMEVFILTR<n>_EL0, and SPMEVFILT2R<n>_EL0 are provided for this.
- Controls for:
  - Enabling and resetting counters, in the System PMU control register SPMCR_EL0.
  - If FEAT_SPMU2 is implemented, setting individual counters to zero, in SPMZR_EL0.
  - Count enable set and clear registers, SPMCNTENSET_EL0 and SPMCNTENCLR_EL0, respectively.
  - Counter overflow set and clear registers, SPMOVSSET_EL0 and SPMOVSCLR_EL0, respectively.
  - Interrupt enable set and clear registers, SPMINTENSET_EL1 and SPMINTENCLR_EL1, respectively.
  - System PMU access control registers SPMACCESSR_EL3, SPMACCESSR_EL2, and SPMACCESSR_EL1.
  - If Secure EL1 is implemented, an OPTIONAL Secure state observability control register SPMSCR_EL1.
  - If FEAT_RME is implemented, an OPTIONAL Root and Realm state observability control register SPMROOTCR_EL3.
- Read-only identification registers:
  - Implementation and architecture identification registers, SPMIIDR_EL1 and SPMDEVARCH_EL1, respectively.
  - If the System PMU is accessible to other PEs in the system, then affinity register SPMDEVAFF_EL1.
  - Implementation configuration register SPMCFGR_EL1.
  - If the System PMU implements more than one Counter Group, then Counter Group configuration registers SPMCGCR<n>_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D15.2 System PMU configuration

IXCMGX

A System PMU is identified to software by the following registers:

- SPMIIDR_EL1 uniquely identifies the component implementer and provides a part number code specific to that implementation. This register is OPTIONAL, and if the register is not implemented, it is RAZ.
- SPMDEVARCH_EL1 describes a generic programmers' model for the System PMU that might be shared by multiple implementations. This register is OPTIONAL, and if the register is not implemented, it is RAZ.
- SPMDEVAFF_EL1 describes which PEs the System PMU is shared with, if any.
- SPMCFGR_EL1 describes the capabilities of the System PMU, including, but not limited to, the number and size of counters implemented by the System PMU.
- If SPMCFGR_EL1.NCG indicates the System PMU implements more than one Counter Group, then SPMCGCR<n>_EL1 indicates the number of counters in each Counter Group. Otherwise, SPMCGCR<n>_EL1 are RAZ and all the counters are in the first Counter Group.

When these registers are implemented by a System PMU, software can use these registers along with pre-existing knowledge to determine configuration information about the System PMU, as follows:

1. If software recognizes the identity of the System PMU in SPMIIDR_EL1, it can load a configuration for the System PMU.
2. Otherwise, if software recognizes the architecture in SPMDEVARCH_EL1, it can load a generic configuration for the System PMU.
3. Otherwise, software can use the capabilities described by SPMCFGR_EL1 to define a generic configuration for the System PMU.

The configuration contains, for instance, details of how the event configuration registers SPMEVTYPER<n>_EL0, SPMEVFILTR<n>_EL0, and SPMEVFILT2R<n>_EL0, are programmed, including but not limited to the following IMPLEMENTATION DEFINED information:

- Field assignments.
- Whether additional features are implemented, such as threshold and edge detection.
- Whether an interrupt on counter overflow is implemented.
- Assignment of event codes to events.
- Definitions of events.
- Details of any fixed-function counters.
- Useful metrics derived from events.
- Security policies for allowing access to the System PMU from lower Exception levels.

IYKJXW

A System PMU might statically divide its counters into an IMPLEMENTATION DEFINED number of Counter Groups. The number of Counter Groups is identified to software by SPMCFGR_EL1.NCG. At most 15 Counter Groups can be implemented. If SPMCFGR_EL1.NCG is `0b0000`, then only one Counter Group is implemented, and all counters are in the first Counter Group.

Each Counter Group has an IMPLEMENTATION DEFINED number of counters in that group, from zero to the value *max*. The number of counters in Counter Group <n × 8 + m>, is identified to software by SPMCGCR<n>_EL1.N<m>. The first counter in Counter Group <n> is SPMEVCNTR<n × max>_EL0.

The value *max* depends on the number of Counter Groups implemented, and is defined by the following table:

Table D15-1 Value of max when determining the first counter in a Counter Group

Number of Counter Groups	*max*
2	32
3 or 4	16
Between 5 and 8	8
Between 9 and 15	4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D15-1 Determining the number of counters in a System PMU

The number of counters in a System PMU can be determined as described in the following example:

- SPMCFGR_EL1.NCG reads as `0b0001`, indicating 2 Counter Groups.
- SPMCGCR0_EL1 reads as `0x0000_0604`, indicating:
  - Group 0 has four counters: SPMEVCNTR0_EL0 to SPMEVCNTR3_EL0.
  - Group 1 has six counters: SPMEVCNTR32_EL0 to SPMEVCNTR37_EL0.
- SPMCFGR_EL1.N reads as `0x09`, indicating 10 counters are implemented overall.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D15.3 Accessing System PMUs

IVXVXK A System PMU <s> is selected by setting SPMSELR_EL0.SYSPMUSEL to <s>, where $0 \leq s \leq 31$. This determines which System PMU is being accessed by the System PMU registers. If enabled, then the System PMUs are always monitoring.

IWYLYL The largest value of SPMSELR_EL0.SYSPMUSEL is identified by ID_AA64DFR1_EL1.SYSPMUID.

IBWTWP Implemented System PMUs might not be contiguously accessible. That is, if System PMU <s> is not implemented, this does not imply System PMU <s+1> is not implemented. Therefore, the value of ID_AA64DFR1_EL1.SYSPMUID does not necessarily indicate the total number of accessible System PMUs.

RJJNZK Accesses to System PMU <s> are controlled by SPMACCESSR_EL1.P<s>, SPMACCESSR_EL2.P<s>, SPMACCESSR_EL3.P<s>, and if implemented by the System PMU, SPMSCR_EL1 and SPMROOTCR_EL3.

IFKZLR FEAT_SPMU does not provide an equivalent of the fine-grained EL0 controls provided for the PE PMU by PMUSERENR_EL0 and PMUACR_EL1.

D15.3.1 Accessing System PMU registers

RNKSGW A System PMU that is shared by multiple PEs implements multi-copy atomicity for writes to the System PMU registers, with each of those PEs being observers of writes to the registers:

- All writes to the same System PMU register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a System PMU register does not observe a direct write until all observers observe that direct write.

ITZFBD

Example D15-2 Observing a write to System PMU registers

When observing a write to System PMU registers, this means observing all effects of the write. For example:

- The effect writing to SPMEVCNTR<n>_EL0 is to set the value of the counter to the value that was written.
- The effect writing a 1 to SPMOVSCLR_EL0[n] is to clear both SPMOVSCLR_EL0[n] and SPMOVSSET_EL0[n] to 0.

RTVFHH The System PMU is also an observer of writes to the System PMU registers that makes indirect reads and indirect writes of the registers when counting events.

RXFHMP A direct write to a System PMU register that has been synchronized by a Context synchronization event on the PE performing the direct write will be observable to all observers in finite time.

RBXJVP A `DSB` instruction ordered after a direct write to a System PMU register does not complete until all observers observe the direct write. A Context synchronization event is required to create the order between the direct write and the `DSB` instruction.

IYPSBB Because a direct read from a System register in program order following a direct write to the same System register will observe the write to the register, reading back from that register also guarantees that all observers observe the direct write.

D15.3.2 Accessing System PMU counters

IZWVKD A System PMU counter <n> is selected by setting SPMSELR_EL0.BANK to <n DIV 16>, where $0 \leq n \leq 63$.

IRHHZC There are no mechanisms to trap accesses to individual counters, ranges of counters, or unimplemented counters in a System PMU. If not trapped by a higher priority exception, then accesses to unimplemented counters in a System PMU are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D15.3.3 Zeroing System PMU counters

I_{GDKFB} If FEAT_SPMU2 is implemented, then writing 1 to SPMZR_EL0.P<n> sets System PMU counter <n> in System PMU <s> to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D15.4 Generating System PMU overflow interrupt requests

R_{TCYDX} It is IMPLEMENTATION DEFINED whether a System PMU implements a counter overflow flag for each counter SPMEVCNTR<n>_EL0 implemented by the System PMU.

If the System PMU implements a counter overflow flag for SPMEVCNTR<n>_EL0, then all of the following apply:

- When incrementing SPMEVCNTR<n>_EL0 causes an unsigned overflow of the counter, the System PMU sets SPMOVSCLR_EL0[n] to 1.
- SPMOVSCLR_EL0[n] is R/W1C.
- SPMOVSSET_EL0[n] is R/W1S.

Otherwise, SPMOVSCLR_EL0[n] and SPMOVSSET_EL0[n] are RAZ/WI.

Note

If an event can occur multiple times in a single clock cycle, then a counter overflow can occur without the counter registering a value of zero.

R_{NZHDW} The size of each counter implemented by a System PMU is IMPLEMENTATION DEFINED, between 1 and 64 bits. Unimplemented bits in each counter register are RES0.

R_{NBPGV} If a System PMU implements a counter overflow flag for any counter SPMEVCNTR<n>_EL0 implemented by the System PMU, then it is IMPLEMENTATION DEFINED whether the System PMU implements an overflow interrupt request.

I_{NJGBT} Each System PMU that implements an overflow interrupt request has its own interrupt request, and it is IMPLEMENTATION DEFINED whether the interrupts are implemented as a Shared Peripheral Interrupt (SPI) or a Private Peripheral Interrupt (PPI). A System PMU that is not shared with other PEs should use a PPI.

Interrupt IDs are IMPLEMENTATION DEFINED.

R_{PGYZY} When the overflow interrupt request is implemented, software can program SPMINTENSET_EL1 and SPMINTENCLR_EL1 such that an overflow interrupt request is generated when a counter overflow flag is 1.

If the System PMU implements the overflow interrupt request and implements a counter overflow flag for SPMEVCNTR<n>_EL0, then all of the following apply:

- SPMINTENCLR_EL1[n] is R/W1C.
- SPMINTENSET_EL1[n] is R/W1S.

Otherwise, SPMINTENCLR_EL1[n] and SPMINTENSET_EL1[n] are RAZ/WI.

If the overflow interrupt request is implemented as a level-sensitive request, then the System PMU signals an overflow interrupt request when all of the following are true for any implemented counter SPMEVCNTR<n>_EL0:

- SPMCR_EL0.E is 1.
- SPMOVSCLR_EL0[n] is 1.
- SPMINTENCLR_EL1[n] is 1.

For more information, see SPMINTENSET_EL1 and SPMINTENCLR_EL1.

I_{MZGZJ} The architecture does not provide any support for a System PMU interrupt implemented as a message-signaled interrupt (MSI). To support implementing the interrupt as an MSI, additional IMPLEMENTATION DEFINED registers must be defined.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D16
The Activity Monitors Extension

This chapter describes version 1 of the *Activity Monitor Unit* (AMU) architecture, AMUv1, an optional non-invasive component. It contains the following sections:

- About the Activity Monitors Extension.
- Properties and behavior of the activity monitors.
- AMU events and event numbers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D16.1 About the Activity Monitors Extension

The Activity Monitors Extension is an OPTIONAL extension to the Armv8.4 architecture.

The Activity Monitors Extension implements version 1 of the Activity Monitors architecture, AMUv1, and interfaces to the registers defined by AMUv1, the Activity Monitors registers.

Version 1 of the Activity Monitors architecture implements:

- A counter group of four architected 64-bit event counters. The events counted by the architected event counter are fixed and architecturally defined.

  Note

  The Activity Monitors architecture provides space for up to 16 architected event counters. Future versions of the Activity Monitors architecture might use this space to implement additional architected event counters.

- A counter group of up to 16 auxiliary 64-bit event counters. The event counted for each auxiliary event counter can be fixed or programmable, and whether it is fixed or programmable is IMPLEMENTATION DEFINED. When the event counted by an auxiliary event counter is fixed, this event is IMPLEMENTATION DEFINED.
- Controls for enabling and disabling counters.
- When the event counted by an auxiliary event counter is programmable, controls for assigning an event to the counter.
- Controls that determine whether the activity monitor counters continue to count while the PE is halted in Debug state.

The read-only registers AMCFGR and AMCGCR provide information about features supported by the Activity Monitors Extension, the number of counter groups implemented, the total number of counters implemented, the number of counters implemented within each group, and the size of the counters.

The Activity Monitors Extension provides:

- A mandatory System register interface to the Activity Monitors System registers, for both AArch64 and AArch32 states.

  AArch64 Activity Monitors System registers lists the AArch64 Activity Monitors System registers, and AArch32 Activity Monitors System registers lists the AArch32 Activity Monitors System registers. Table K14-3 shows the relationship between the AArch64 and the AArch32 Activity Monitors System registers.
- Controls that allow software to enable or disable access by software running at lower Exception levels to the Activity Monitors registers.
- When FEAT_AMUv1p1 is implemented, and the hypervisor is using AArch64, offset registers that support virtualization of the Activity Monitor event counters.
- An optional external interface providing read-only memory-mapped access to the Activity Monitors registers.

  Alphabetical index of memory-mapped registers lists the Activity Monitors memory-mapped registers. For more information on the recommended external interface, see Recommended External Interface to the Activity Monitors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D16.2 Properties and behavior of the activity monitors

D16.2.1 Basic characteristics of the activity monitor event counters

Every activity monitor event counter is a 64-bit wrapping counter. When an activity monitor event counter wraps, the counter overflows.

Note

The Activity Monitor architecture does not provide support for overflow status indication or interrupts.

The state of the authentication signals do not affect counting.

Any change in clock frequency, including when a WFI and WFE instruction stops the clock, can affect any counter.

If FEAT_AMUv1p1 is implemented, for the architected event counters 0, 2 and 3, and each auxiliary event counter configured to use an offset, there is an offset register which is used to virtualize the count on a read from EL1 or EL0. At EL2, EL3 or from the memory-mapped view, permitted accesses to the counters use the physical view without any offset. See Virtualization

D16.2.2 Counter configuration and controls

For each architected event counter AMEVCNTR0<n>, there is a corresponding event type register AMEVTYPER0<n> which provides information on the event counted by that counter. The event type registers AMEVTYPER0<n> are read-only.

For each auxiliary event counter AMEVCNTR1<n>, there is a corresponding event type register AMEVTYPER1<n> which provides information on the event counted by that counter. When the event counted by an auxiliary event counter is fixed, the corresponding event type register AMEVTYPER1<n> is read-only. When the event counted by an auxiliary event counter is programmable, the corresponding event type register AMEVTYPER1<n> is read/write.

For each counter group, there is a pair of separate controls to enable and disable the counters in that counter group. AMCNTENCLR0 and AMCNTENSET0 are used to disable and enable the architected event counters. AMCNTENCLR1 and AMCNTENSET1 are used to disable and enable the auxiliary event counters.

While the PE is halted in Debug state, AMCR.HDBG controls whether activity monitor counting is halted.

AMUSERENR.EN controls access from EL0 to the Activity Monitor Extension System registers. CPTR_EL2.TAM and HCPTR.TAM control access from EL0 and EL1 to the Activity Monitor Extension System registers. CPTR_EL3.TAM control access from EL0, EL1, and EL2 to the Activity Monitor Extension System registers.

Note

These controls obey the priority order described in Prioritization of Synchronous exceptions taken to AArch64 state and Synchronous exception prioritization for exceptions taken to AArch32 state.

AMUSERENR.EN is configurable at EL1, EL2, and EL3. All other controls, as well as the value of the counters, are configurable only at the highest implemented Exception level.

If FEAT_AMUv1p1 is implemented, AMCG1IDR_EL0 defines which auxiliary counters are implemented, and if virtual offsets are enabled, indicates which of the implemented auxiliary counters have a virtual offset when read from EL0 and EL1.

If FEAT_AMUv1p1 is implemented, AMCR.CG1RZ controls whether the auxiliary event counters read as zero if they are accessed at an Exception level lower than the highest implemented Exception level.

D16.2.3 Power and reset domains

The power domain of the Activity Monitoring Unit is IMPLEMENTATION DEFINED and named the AMU domain.

The reset domain of the Activity Monitoring Unit is IMPLEMENTATION DEFINED and named the AMU reset.

The AMU power domain may be the Core power domain.

When an AMU reset of the AMU power domain occurs, the Activity Monitoring Unit is reset and the counters are reset to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the PE is not in reset, the Activity Monitoring Unit is available.

D16.2.4 Accuracy and non-invasive behavior

The activity monitors are a non-invasive component which must provide broadly accurate and statistically useful count information.

The implementation of an architecturally required event might create a conflict between the requirement to be non-invasive and the requirement to present an accurate value of the count under normal operating conditions. An implementation might provide an IMPLEMENTATION DEFINED control that disables accurate count of the event to restore performance and document the impact on performance of accurate counting. The expectations for non-invasive behavior and the degree of inaccuracy of the activity monitors are otherwise as described for the Performance Monitors architecture.

Note

For information on the expectations for non-invasive behavior and the degree of inaccuracy of the Performance Monitors, see Non-invasive behavior and A reasonable degree of inaccuracy.

D16.2.5 Virtualization

FEAT_AMUv1p1 supports virtualized access to the Activity Monitors event counters at EL1 and EL0.

The fields HCR_EL2.AMVOFFEN and SCR_EL3.AMVOFFEN enable and disable virtualization. When enabled, the architected event counters 0, 2 and 3 have counter offsets. Architected event counter 1 does not have an offset. The register AMCG1IDR_EL0 indicates which of the implemented auxiliary event counters has implemented counter offsets. An implemented event counter that does not have a defined offset has an effective offset of zero. The offset registers can be accessed only at EL2 or EL3, and affect views of the event counters at EL1 and EL0 from the System register interfaces only.

The AMEVCNTVOFF0<n>_EL2 registers hold the offsets for the implemented and enabled architected event counters.

The AMEVCNTVOFF1<n>_EL2 registers hold the offsets for the implemented and enabled auxiliary event counters.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D16.3 AMU events and event numbers

The Activity Monitors architecture uses the event number space defined by the Performance Monitors architecture to identify events.

The Activity Monitors architecture defines additional events and adds them to the event number space defined by the Performance Monitors architecture for common events.

If the event is counting an IMPLEMENTATION DEFINED event, it must use an event number from the IMPLEMENTATION DEFINED event space.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event.

When a common event is available to both the Performance Monitors architecture and the Activity Monitors architecture within one implementation, both architectures use the same event number.

D16.3.1 Architected event counters

Version 1 of the Activity Monitors architecture, AMUv1, requires four events to be counted by the architected activity monitor event counters.

The events required to be counted are:

`0x0011`**, CPU_CYCLES, Processor frequency cycles**

The counter increments on every cycle when the PE is not in WFI or WFE state. When the PE is in WFI or WFE state, this counter does not increment.

This event is counted by AMEVCNTR0<n>, where n is 0.

`0x4004`**, CNT_CYCLES, Constant frequency cycles**

The counter increments at a constant frequency when the PE is not in WFI or WFE state, equal to the rate of increment of the System counter, CNTPCT_EL0. When the PE is in WFI or WFE state, this counter does not increment.

This event is counted by AMEVCNTR0<n>, where n is 1.

`0x0008`**, INST_RETIRED, Instructions retired**

This event is defined identically to INST_RETIRED in the FEAT_PMUv3 architecture.

This event is counted by AMEVCNTR0<n>, where n is 2.

`0x4005`**, STALL_BACKEND_MEM, Memory stall cycles**

The counter counts cycles in which the PE is unable to dispatch instructions from the frontend to the backend of the PE due to a backend stall caused by a demand data miss in the last level of data or unified cache within the PE clock domain or, if Armv8.7 is implemented, a Non-cacheable data access in progress.

If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether the counter counts backend stall cycles when a Non-cacheable data access is in progress.

This event is counted by AMEVCNTR0<n>, where n is 3.

D16.3.2 Auxiliary event counters

Auxiliary event counters can count events defined by the Performance Monitors architecture and IMPLEMENTATION DEFINED events defined specifically for activity monitoring.

Implementations must not reuse an IMPLEMENTATION DEFINED event number for different hardware events across the Performance Monitors architecture and the Activity Monitors architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D17
The Statistical Profiling Extension

This chapter describes the Statistical Profiling Extension. It contains the following sections:

- About the Statistical Profiling Extension.
- Defining the sample population.
- Controlling when an operation is sampled.
- Enabling profiling.
- Filtering sample records.
- The profiling data.
- The Profiling Buffer.
- Profiling Buffer management.
- Synchronization and Statistical Profiling.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.1 About the Statistical Profiling Extension

D_{NLVYX} When the Statistical Profiling Extension is implemented, the PE includes a *Statistical Profiling Unit* (SPU).

I_{WKZFN} When profiling is enabled and not stopped, the SPU does the following:

1. Chooses an operation from a sample population, that can be restricted by Exception level, at a programmable interval that might have some random, or pseudorandom, perturbation.
2. Takes a trace of the sampled operation. This includes the PC, events, timings, and data addresses, related to the sampled operation. This is the profiling operation.
3. If defined, filters out potential sample records generated by the profiling operation by reference to any or all of the following:
   a. The type of operation.
   b. Events.
   c. Latency.
   d. Data source, for loads.
4. Creates a record that contains the traced information. Sample records that meet the criteria of the filter are written to and stored in a memory buffer. These sample records can be processed by software when the memory buffer is full.

R_{JMMZP} After creating the record for the sampled operation, the SPU writes a record to memory sequentially in increasing address order. That is, for each byte in the record, the SPU:

1. If required, translates the current write pointer virtual address to a physical address.
2. If permitted, writes the byte of record data to the write address.
3. If profiling is not stopped, increments the current write pointer.

Profiling Buffer management events are processed as part of this loop. Implementations compliant with the architecture conform with the described behavior of the SPU.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with this specification is the same as a simple sequential model. Although records appear to be written sequentially, the architecture does not require that a record is written sequentially by the SPU.

I_{HXPSC} If FEAT_PMUv3 is implemented, then the SPU also generates PMU events:

- For each operation in the sample population.
- When an operation is chosen for sampling.
- When a sample record is created.

For more information, see PMU Event Descriptions.

D17.1.1 Non-invasive behavior

Statistical Profiling is a non-invasive debug operation:

- While profiling is enabled, the operation and performance of the processing element (PE) must not be significantly impacted between sampled operations, that is, other than for writing out sample records and processing Profiling Buffer management events.
- The performance of the sampled operation and the performance of the PE in general must not be significantly impacted. The sample records are not written to memory until after the sampled operation has finished execution. However, this does not apply if the sample records are physical addresses for data access operations. In this case, the impact is IMPLEMENTATION DEFINED.
- The profiling operation to write sample records must not be excessively impactful on the performance of the sampled operation or the performance of the PE generally.

D17.1.2 Multithreaded implementations

In a multithreaded implementation:

- Statistical Profiling is implemented per-thread.
- The sample interval counter counts only operations for the thread that is being profiled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Latency and other cycle counters count each cycle for the PE for which the thread was active and could issue an operation.

The architecture does not define features for inter-thread profiling and does not support sharing the Profiling Buffer between threads.

Note

An implementation is described as multithreaded when the lowest level of affinity consists of logical processors that are implemented using a multithreading type approach. That is, the performance of processors at the lowest affinity level is interdependent.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.2 Defining the sample population

All samples are taken from a *population of operations*. The population is dynamic rather than static. That is, if a program executes the same operation multiple times (for example, because of loops and subroutines) then that operation appears multiple times in the population.

The operations are one of the following:

- If FEAT_SPE_ArchInst is implemented, architecture instructions.
- If FEAT_SPE_ArchInst is not implemented, IMPLEMENTATION DEFINED microarchitectural operations (micro-ops).

Architecture instruction means a single instruction that is defined by the A-profile instruction set architecture in AArch64 state.

An architecture instruction might create one or more micro-ops at any point in the execution pipeline. The definition of a micro-op is IMPLEMENTATION SPECIFIC. An architecture instruction might create more than one micro-op for each instruction. A micro-op might also be removed or merged with another micro-op in the Execution stream, so an architecture instruction might create no micro-ops for an instruction. Any arbitrary translation of architecture instructions to an equivalent sequence of micro-ops is permitted.

This means that the relationship between a micro-op and an architecturally executed instruction is IMPLEMENTATION DEFINED, and an architectural instruction of one type might be sampled as a micro-op of a different type. See also Information collected for micro-ops. In some implementations, the relationship between architecture instructions and micro-ops might vary over time.

Note

Sampling from architecture instructions does not require that the instruction is architecturally executed.

D17.2.1 Operations that might be excluded from the sample population

It is IMPLEMENTATION DEFINED whether each of the following operations is part of the sample population:

- Operations on misspeculated paths.
- Operations (specifically micro-ops) that do not relate to any architecture instruction.
- Operations that generate non-architectural exceptions.

If the operation is not part of the sample population, the operation does not cause the sample interval counter to decrement, is not counted by the SAMPLE_POP event and therefore is never sampled.

If the operation is part of the sample population, the operation causes the sample interval counter to decrement, is counted by the SAMPLE_POP event, and might be sampled and counted by the SAMPLE_FEED event. However, it is IMPLEMENTATION DEFINED whether the sample record for such a sampled operation is captured in the Profiling Buffer. For more information, see Sample operation records for misspeculated and non-architectural operations and Non-architectural exceptions.

If such a sample record is not captured into the Profiling Buffer, then no packets are output and the sample is not counted by the SAMPLE_FILTRATE event.

Note

If the owning Exception level passes this data to less privileged software for processing, it can set PMSFCR_EL1.FE to 1 and PMSEVFR_EL1[1] to 1 to prevent speculative instructions from being recorded in the Profiling Buffer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.3 Controlling when an operation is sampled

The sample interval counter, PMSICR_EL1.COUNT controls when an operation is selected for sampling. In some implementations, a secondary sample interval counter, PMSICR_EL1.ECOUNT, is also used.

The following sections describe the operation of the sample interval counters.

Details of the random or pseudorandom number generator used when PMSIRR_EL1.RND is set to 1 are IMPLEMENTATION DEFINED. See Generating random numbers for sampling.

D17.3.1 Operation sampling

A sample operation is as follows:

1. Software writes a sampling interval to PMSIRR_EL1.INTERVAL, and sets PMSICR_EL1 to zero. The interval is measured in operations.
2. The sample interval counter is decremented by hardware for each operation when profiling is enabled.
3. When the sample interval counter reaches zero, then:
   a. If random perturbation is enabled, the PE continues to count for a random number of further operations while profiling is enabled.
   b. An operation is chosen for profiling. The choice of operation around the sampling point is IMPLEMENTATION SPECIFIC, but does not introduce sampling bias.
4. The sample interval counter is reloaded and the process loops to step 2. It is IMPLEMENTATION DEFINED whether the sample interval counter is reloaded before step 3.a) or at step 3.b). That is, before or after counting the random number of further operations.
5. The chosen operation is marked as the sampled operation. The PE collects information about the sampled operation as it executes by a profiling operation.
6. The sample record is created when the sampled operation has finished execution.

D17.3.2 Generating random numbers for sampling

The random number generator is IMPLEMENTATION DEFINED. Implementations might use a pseudorandom number. The random number generator must be reset into a useable state. An implementation might include IMPLEMENTATION DEFINED registers to further configure the random number generator.

It is IMPLEMENTATION DEFINED whether the PE adds the random number to the sample interval counter prior to counting down the interval, or after the counter reaches zero and the counter has been reloaded.

D17.3.3 Initializing the sample interval counters

When the PE moves from a state where profiling is disabled to a state where profiling is enabled:

- If PMSICR_EL1 is nonzero, then sampling restarts from the current values in PMSICR_EL1.
- If PMSICR_EL1 is zero, then it is loaded with an initial value. The behavior depends on PMSIRR_EL1.RND and whether FEAT_SPE_ERnd is implemented.
  - If PMSIRR_EL1.RND is 0:
    - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
    - PMSICR_EL1.COUNT[7:0] is set to `0x00`.
  - If PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is not implemented:
    - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
    - PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range `0x00` to `0xFF` inclusive.
  - If PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is implemented:
    - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
    - PMSICR_EL1.COUNT[7:0] is set to `0x00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.3.4 Behavior of the sample interval counter while profiling is enabled

While profiling is enabled, the counters control when an operation is selected for sampling. The behavior depends on PMSIRR_EL1.RND and whether FEAT_SPE_ERnd is implemented.

D17.3.4.1 If PMSIRR_EL1.RND is 0

While nonzero, the sample interval counter decrements by 1 for each member of the sample population. When the counter reaches zero:

- A member of the sampling population is selected for sampling.
- The counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to `0x00`.

Note

Because the counter counts down to zero, when PMSIRR_EL1.RND is 0 the interval between operations being selected for sampling is (*INTERVAL*×256+1).

D17.3.4.2 If PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is not implemented

While nonzero, the sample interval counter decrements by 1 for each member of the sample population. When the counter reaches zero:

- A member of the sampling population is selected for sampling.
- The counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range `0x00` to `0xFF` inclusive.

Note

When PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is not implemented, the mean interval between operations being selected for sampling is (*INTERVAL*×256+128), if the random number generator is uniform.

D17.3.4.3 If PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is implemented

While nonzero, the primary sample interval counter decrements by 1 for each member of the sample population. When the primary counter reaches zero:

- The primary sample interval counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to `0x00`.
- The secondary sample interval counter, PMSICR_EL1.ECOUNT, is set to a random or pseudorandom value in the range `0x00` to `0xFF` inclusive. If this value is zero, then an operation is selected for sampling.

  Otherwise:
  - While the secondary sample interval counter is nonzero, the secondary sample interval counter decrements by 1 for each member of the sample population. The primary sample interval counter also continues to decrement because it is also nonzero.
  - When the secondary sample interval counter reaches zero, an operation is selected for sampling.

Note

When PMSIRR_EL1.RND is set to 1 and FEAT_SPE_ERnd is implemented, the mean interval between operations being selected for sampling is (*INTERVAL*×256+1), if the random number generator is uniform.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.3.5 Behavior of the sample interval counter while profiling is disabled

When profiling is disabled:

- No operations are selected for sampling.
- No sample records are collected.
- The sample interval counters retain their values and do not decrement.

D17.3.6 Where operations are sampled

The exact point in the sampled lifespan of operations at which operations are chosen for profiling is IMPLEMENTATION DEFINED.

Note

Arm recommends that the point at which operations are sampled is linked to the definition of the Performance Monitors Extension (PMU) STALL_FRONTEND and STALL_BACKEND events, so that sampling records information for STALL_BACKEND stalls.

D17.3.7 Sample collisions

R_{QWTQB} The maximum number of sampled operations that a PE can support simultaneously is IMPLEMENTATION DEFINED.

I_{QYCRM} If the maximum number of simultaneous sampled operations has been reached at the point when a new operation must be sampled, the new sample is said to have collided with a previous sampled operation.

The PE records the fact that a sampled operation has collided with another sampled operation. Software can also count the number of collisions and gauge the impact of the collisions.

I_{DBJXM} On a sample collision:

- The PMU event SAMPLE_COLLISION is generated.
- PMBSR_ELx.COLL is set to 1, where ELx is the value in the *Other event* column of Table D17-7. The other fields in PMBSR_ELx and the other PMBSR_ELy registers are unchanged.
- The new operation is not sampled.

I_{RCCQG} Sample collisions are likely to increase with shorter sampling intervals. Samples might not be representative of the overall workload if statistically significant numbers of sample collisions occur.

R_{LTMKQ} When all of the following are true, the direct read **C** of PMBSR_ELx.COLL will return 1:

- An instruction or operation **A** is selected for sampling.
- A sample collision occurs for sampled operation **A**.
  The PE sets PMBSR_ELx.COLL to 1. This is an indirect write to PMBSR_ELx.
- A Context synchronization event **B** is in program order after **A**.
- A direct read **C** of PMBSR_ELx is in program order after **B**.

ELx is the value in the *Other event* column of Table D17-7. The other fields in PMBSR_ELx and the other PMBSR_ELy registers are unchanged. For more information, see Synchronization and Statistical Profiling.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.4 Enabling profiling

R_{TWFRY} If and only if any of the following are true, then profiling is *stopped*:

- PMBSR_EL1.S is 1.
- FEAT_SPE_EXC is implemented and all of the following are true:
  - EL2 is implemented and any of the following applies:
    - The Effective value of SCR_EL3.NS is 1.
    - The Effective value of SCR_EL3.{NS, EEL2} is {0, 1}.
  - EL3 is not implemented or MDCR_EL3.PMSEE is not `0b00`.
  - PMSCR_EL2.EE is either `0b10` or `0b11`.
  - PMBSR_EL2.S is 1.
- FEAT_SPE_EXC is implemented and all of the following are true:
  - EL3 is implemented.
  - MDCR_EL3.PMSEE is either `0b10` or `0b11`.
  - PMBSR_EL3.S is 1

The pseudocode function `SPEProfilingStopped()` shows this.

R_{WXXFL} When all of the following are true, profiling is *enabled*:

- The PE is in AArch64 state.
- PMBLIMITR_EL1.E is 1. The Profiling Buffer is *enabled*.
- Profiling is not stopped.
- The PE is executing at either the Profiling Buffer owning Exception level or any lower Exception level.
- Either the Profiling Buffer owning Exception level is EL2 or the Effective value of HCR_EL2.TGE is 0.
- The PE is executing in the Profiling Buffer owning Security state.
- The PE is in Non-debug state.
- PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} enable profiling at the current Exception level.

Otherwise, profiling is *disabled*.

The pseudocode function `StatisticalProfilingEnabled()` show this.

I_{BRTKK} When the Profiling Buffer is disabled or profiling is stopped:

- Any sample records that have been collected but not written to memory are discarded.
- PMBPTR_EL1 and PMSICR_EL1 are not updated by the SPU.

Note

The owning Security state is controlled by MDCR_EL3.{NSPBE, NSPB} and the owning Exception level is controlled by MDCR_EL2.E2PB. See The owning translation regime.

I_{SBQVZ} If the Profiling Buffer is enabled and the PE is in Non-debug state and using AArch64 state, Table D17-1 defines the valid combinations of the following that define when profiling is enabled:

- The Effective value of SCR_EL3.NSE.
- The Effective value of SCR_EL3.NS.
- The Effective value of SCR_EL3.EEL2.
- The Effective value of MDCR_EL3.NSPBE.
- The Effective value of MDCR_EL3.NSPB.
- The Effective value of MDCR_EL2.E2PB.
- The Effective value of HCR_EL2.TGE.

In Table D17-1:

D	Disabled.
E2SPE	Enabled if PMSCR_EL2.E2SPE == 1, disabled otherwise.
E1SPE	Enabled if PMSCR_EL1.E1SPE == 1, disabled otherwise.
E0HSPE	Enabled if PMSCR_EL2.E0HSPE == 1, disabled otherwise.
E0SPE	Enabled if PMSCR_EL1.E0SPE == 1, disabled otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D17-1 Enabling by Exception level and Security state (for all Exception levels)

Controls							Profiling enabled at			
NSE	NS	NSPBE	NSPB	E2PB	EEL2	TGE	EL3	EL2	EL1	EL0
0	1	1	0b0x	x	x	x	D	D	D	D
		0	0b1x	0b1x	x	0	D	D	E1SPE	E0SPE
				0b1x	x	1	D	D	n/a	D
				0b00	x	0	D	E2SPE	E1SPE	E0SPE
				0b00	x	1	D	E2SPE	n/a	E0HSPE
0	0	0	0b1x	x	x	x	D	D	D	D
			0b0x	x	0	x	D	n/a	E1SPE	E0SPE
				0b1x	1	0	D	D	E1SPE	E0SPE
				0b1x	1	1	D	D	n/a	D
				0b00	1	0	D	E2SPE	E1SPE	E0SPE
				0b00	1	1	D	E2SPE	n/a	E0HSPE
1	1	1	0b0x	x	x	x	D	D	D	D
			0b1x	0b1x	x	0	D	D	E1SPE	E0SPE
				0b1x	x	1	D	D	n/a	D
				0b00	x	0	D	E2SPE	E1SPE	E0SPE
				0b00	x	1	D	E2SPE	n/a	E0HSPE

This is described in the pseudocode function `StatisticalProfilingEnabled()`.

I_{JHPHD} Table D17-2 shows whether profiling is *stopped*. Each column in the left side of the table refers to the Effective value of a field in a System register.

Table D17-2 Profiling stopped

PMBSR_EL1.S	MDCR_EL3.PMSEE	PMBSR_EL3.S	PMSCR_EL2.EE	PMBSR_EL2.S	Profiling stopped
0b0	0b00	X	XX	X	True
0b0	0b01	X	0b0X	X	True
0b0	0b01	X	0b1X	0b0	True
0b0	0b01	X	0b1X	0b1	False
0b0	0b1X	0b0	0b0X	X	True
0b0	0b1X	0b0	0b1X	0b0	True
0b0	0b1X	0b0	0b1X	0b1	False
0b0	0b1X	0b1	XX	X	False
0b1	XX	X	XX	X	False

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.5 Filtering sample records

After the sampled operation has finished execution, the Statistical Profiling Unit might discard the sample records generated by the profiling operation.

PMSFCR_EL1 controls filtering of sample records, as described in the following sections. If multiple filters are enabled, a packet is discarded if it is discarded by any of the enabled filters. These controls combine together as a logical AND.

Example D17-1 Collection of sampled operations

If PMSFCR_EL1.FE is 1, PMSFCR_EL1.FnE is 0, PMSFCR_EL1.FT is 1, and PMSFCR_EL1.FL is 1, then only sampled operations that meet all of the following criteria are recorded and written to the Profiling Buffer:

- The sampled operation is one of the selected operation types.
- The operation has all of the events in the filter set.
- The total latency is equal to or greater than the minimum latency.

This is described in the pseudocode function `SPECollectRecord()`.

D17.5.1 Filtering by operation type

I_{BWNKX} PMSFCR_EL1.FT enables filtering by operation type. When enabled PMSFCR_EL1.{ST, LD, B, FP, SIMD, STm, LDm, Bm, FPm, SIMDm} define the collected types.

I_{ZZTZL} When FEAT_SPE_EFT is implemented, the following values are defined:

- PMSFCR_EL1.{SIMD, FP, ST, LD, B} are the operation type filter controls, `ctrl`.
- PMSFCR_EL1.{SIMDm, FPm, STm, LDm, Bm} are the operation type filter mask, `mask`.
- `flags` is a five bit value {SIMD, FP, B, LD, ST} that defines the type of the sampled operation.

R_{NYZKJ} When FEAT_SPE_EFT is not implemented:

- PMSFCR_EL1.{ST, LD, B} are the operation type filter controls, `ctrl`.
- The operation type filter mask, `mask`, is zero.
- `flags` is a three bit value {ST, LD, B} that defines the type of the sampled operation.

Filtering by SIMD&FP types is not supported.

D_{DSNMB} The operation types are defined as follows:

ST Means a sampled store operation, including all atomic operations. That is, an operation counted by ST_SPEC.

LD Means a sampled load operation, including atomic operations that return a value to a register. That is, an operation counted by LD_SPEC.

B Means a sampled branch operation, including direct and indirect branches and exception returns. That is, an operation counted by PC_WRITE_SPEC, including branches which are not taken, but excluding `BRK`, `SVC`, `HVC`, `SMC`, and UNDEFINED instructions which generate exceptions. It is IMPLEMENTATION DEFINED whether `ISB` is sampled as a branch operation.

FP Means a sampled floating-point operations including all of the following:

- Floating-point data-processing operations, including scalar, Advanced SIMD, SVE, and SME floating-point data-processing operations. That is, operations counted by FP_SPEC.
- Scalar load/store of a SIMD&FP register or pair of SIMD&FP registers, excluding loads/stores of Q registers, load and replicate, and structure loads/stores.

Only supported when FEAT_SPE_EFT is implemented.

SIMD Means a sampled SIMD operation, including all of the following:

- Advanced SIMD, SVE, or SME SIMD data-processing operations.
- Structure load/store of one or more SIMD&FP registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Load and replicate to one or more SIMD&FP registers.
- Scalar load/store of a SIMD&FP Q register or pair of Q registers.
- SVE or SME load/store.

That is, an operation counted by SIMD_INST_SPEC.

Only supported when FEAT_SPE_EFT is implemented.

I_{ZWMZT} An operation might be more than one type, in which case the `flags` field has all applicable bits set. For example:

- A scalar load of a single SIMD&FP register has both the `FP` and `LD` bits set.
- An Atomic operation that returns a value to the PE has both the `LD` and `ST` bits set.

R_{YGCFK} When FEAT_SPE_EFT is implemented and PMSFCR_EL1.FT is 1, all of the following apply:

- The sampled operation is discarded if the following is true:

```
(!IsZero(ctrl AND NOT mask) && IsZero(flags AND (ctrl AND NOT mask)))
```

  That is, the SPU applies a Boolean-OR filter based on each of the operation type filter controls where the corresponding operation type filter mask bit is 0:
  - If all the operation type filter mask bits are 1, then no operations are discarded by this part of the filter.
  - Otherwise, for the operation type bits for which corresponding mask bit is 0:
    - If all these bits are 0, then no operations are discarded by this part of the filter.
    - Otherwise, operations that match any of these bits are selected, and other operations are discarded.
- The sampled operation is discarded if the following is true:

```
(flags AND mask) != (ctrl AND mask)
```

  That is, the SPU applies a Boolean-AND filter based on the remaining operation type filter controls where the corresponding operation type filter mask bit is 1:
  - If all the operation type filter mask bits are 0, no operations are discarded by this part of the filter.
  - Otherwise, only operations that exactly match all of these bits are selected, and other operations are discarded.

R_{LVCZH} When FEAT_SPE_EFT is not implemented, if PMSFCR_EL1.FT is 1 and the following expression is true, then the sampled operation is discarded:

```
IsZero(flags AND ctrl)
```

That is, only operations that match any of the types selected in the operations type filter controls are selected, and other operation types are discarded.

I_{XNXRN} When micro-op sampling is implemented, filtering is based on the micro-op type. See Defining the sample population.

D17.5.2 Filtering by the contents of the Events packet

PMSFCR_EL1.FE enables filtering by events, Streaming mode, and Transactional mode bits that are defined by PMSEVFR_EL1. When PMSFCR_EL1.FE is 1, only sampled operations with all the events in the filter set to 1 are recorded and written to the Profiling Buffer.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE enables filtering by events, Streaming mode, and Transactional mode bits that are defined by PMSNEVFR_EL1. When PMSFCR_EL1.FnE is 1, only sampled operations with all the events in the filter set to 0 are recorded and written to the Profiling Buffer.

D17.5.3 Filtering by latency

PMSFCR_EL1.FL enables filtering by total latency. PMSLATFR_EL1.MINLAT defines the minimum latency. When PMSFCR_EL1.FL is 1, only sampled operations with a total latency greater than or equal to the minimum latency are recorded and written to the Profiling Buffer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.5.4 Filtering by data source

If FEAT_SPE_FDS is implemented, PMSFCR_EL1.FDS enables filtering by a data source mask defined by PMSDSFR_EL1. When PMSFCR_EL1.FDS is 1, sampled load operations with a Data Source packet value `DS` are recorded only if PMSDSFR_EL1[UInt(DS[5:0])] is 1. Load operations without a Data Source and other sampled operations are not affected by PMSFCR_EL1.FDS and PMSDSFR_EL1.

D17.5.5 Discard mode

When Discard mode is implemented and enabled, all sampled operations to be discarded and not written to the Profiling Buffer. Discard mode is enabled when PMBLIMITR_EL1.FM is `0b10`, and has all of the following effects:

- All profiling data is discarded after filtering.
- The PMBLIMITR_EL1.LIMIT and PMBPTR_EL1 fields are ignored. PMBPTR_EL1 does not increment for each sampled operation.
- The restrictions on setting PMBLIMITR_EL1.LIMIT and PMBPTR_EL1 do not apply, see Restrictions on the current write pointer.
- Buffer management events are not generated.

Discard mode is implemented when FEAT_SPEv1p2 is implemented.

Other profiling behaviors are unchanged, including:

- The discarding of profiling data occurs after SAMPLE_FILTRATE and other PMU events are counted.
- Sample collisions will still set PMBSR_ELx.COLL to 1. See Sample collisions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.6 The profiling data

Unless otherwise stated, all sample records that are generated by a profiling operation contain:

- A timestamp, if enabled. This is one of:
  - Physical time.
  - Offset physical time.
  - Virtual time.

  It is IMPLEMENTATION DEFINED how this timestamp relates to the sampled operation. It might be the time when the sampled operation was taken or any later time during the lifetime of the sampled operation, that is, up to the time when the sampled operation finishes execution.

  If the Generic Timer system counter is disabled and timestamps are enabled, then it is IMPLEMENTATION DEFINED whether:
  - The SPU behaves as if timestamps are disabled.
  - The timestamp that is collected in the sample record is UNKNOWN.

  Note

  This behavior describes when CNTCR.EN is 0, the Generic Timer system counter is disabled. This behavior does not apply when the Generic Timer system counter is enabled but not accessible at the current Exception level.

- The context, if enabled, which is one or more of:
  - CONTEXTIDR_EL1.
  - CONTEXTIDR_EL2.
  - The Exception level.
  - The Security state.
- Information about whether the sampled operation generated an exception:
  - The target address for an exception generating operation is not collected.
- Information about whether the sampled operation was Architecturally executed.

If the sampled operation is Architecturally executed and does not generate an exception, the sample record also contains:

- The PC virtual address for the sampled operation.
- Information about whether the sampled operation is a branch, a load, a load atomic, a store, or other.
- Information about whether the sampled operation is conditional, conditional select, or not.
- The total latency, a cycle count from the start of the sampled operation up to the point where the operation has finished execution and is no longer capable of stalling any instruction that consumes its output.
- The issue latency, a count of cycles when the operation is waiting to be issued. A sampled operation might be delayed, for example, because the input operands were not available.

If the sampled operation is not Architecturally executed or generates an exception, it is UNPREDICTABLE whether the record contains all or any of this information and the other information about the operation listed in this section and the following subsections. For information on exceptions being taken in sampled operations, see Exceptions.

The architecture defines a set of additional data that is collected in the sample record for each sampled operation. This is described in the following subsections, and comprises:

- Events, which are required to be implemented consistently with PMU Events. For more information, see Statistical Profiling Extension Sample Record Specification and The Performance Monitors Extension.
- Cycle counters. Cycle count values as described in this architecture, which, for a particular implementation, are fixed with an IMPLEMENTATION DEFINED value, might be omitted from the sample record.
- Addresses.

In addition, the architecture permits IMPLEMENTATION DEFINED events, counters, and addresses to be collected.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.6.1 Asynchronous operation

IPXWXV

If an operation executes asynchronously to the main PE, such that the cycles are not main CPU cycles, then the PE might count the cycles recorded in a Counter packet in one of the following ways:

- The Counter packet counts elapsed cycles in the main CPU clock domain. For example, the asynchronous execution unit signals start and stop events to the CPU clock domain and the cycles are counted in the CPU clock domain.
- When FEAT_SPE_SME is implemented, for operations executed in Streaming SVE mode, the Counter packet counts elapsed cycles in the SMCU clock domain. Counter packet.INDEX and PMSIDR_EL1.ALTCLK indicate that the Streaming Mode Compute Unit clock value is recorded.
  This option is not supported for the Total latency Counter packet.
- The Counter packet counts elapsed cycles in a clock domain other than an architecturally defined clock domain. Counter packet.INDEX indicates an IMPLEMENTATION DEFINED counter value.
  This option is not supported for the Total latency Counter packet.

For example, an operation is executed by a separate Streaming Mode Compute Unit.

D17.6.2 Information collected for micro-ops

Because architectural instructions might create zero, one, or more micro-ops, micro-ops might have different characteristics from the architectural instructions they are created from. The data collected for each micro-op is IMPLEMENTATION DEFINED. The PE collects the subset of data appropriate to the micro-op.

However, in order for the profile information to be meaningful to the user of the profile, Arm recommends that the sampled micro-op is recorded as a type appropriate to the instruction when the recorded properties of the micro-op correlate with the properties of the instruction.

Example D17-2 Sampling of micro-ops

If an architectural load instruction is split into an address generation micro-op and a load micro-op, then when generating the sample record and filtering based on operation type:

- If the address generation micro-op is sampled, the sampled operation is treated as *other*.
- If the load micro-op is sampled, the sampled operation is treated as a load.

D17.6.3 Additional information for each profiled branch or exception return

For an Architecturally executed sampled branch or exception return operation, the profiling operation records:

- The sampled operation type as an unconditional branch or a conditional branch. Sampled exception returns are treated as unconditional branches by the Statistical Profiling Extension.
- If the branch is taken, the target virtual address of the branch. The target virtual address includes the Exception level and Security state of the target. If the sampled operation is an illegal exception return, it is CONSTRAINED UNPREDICTABLE whether the context information recorded in the target virtual address is the actual target context, or the target context that is described by the SPSR.
- If the PE implements branch prediction, whether the branch was correctly predicted or mispredicted.
- Whether the branch was taken or not taken.
- Whether the branch was direct or indirect.
- If the branch is not taken, a target virtual address might be recorded. Software must treat this value if present as UNKNOWN.
- If FEAT_SPE_PBT is implemented, the target address of the most recently executed branch that was taken and retired in program order before the sampled operation.
- If FEAT_SPE_CRR is implemented, whether the branch was a branch with link, a procedure return, or neither of these.

If an operation executes both a Branch and a Guarded Control Stack data access, then all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Operation Type packet for the operation is Branch.
- The information for both the branch and the load or store is collected.
- The operation is treated as both a branch and a load or store for the purposes of filtering. See Filtering sample records.

Note

A sampled operation that generates an exception is not treated as a branch.

It is IMPLEMENTATION DEFINED whether an ISB instruction is treated as a branch to the next instruction.

D17.6.3.1 Previous branch target

FEAT_SPE_PBT adds the capability to record a packet for each event that provides the target address of the previous taken branch. PMSIDR_EL1.PBT describes whether this feature is implemented.

When implemented, the profiling operation records the target address of the most recent branch that was taken and retired in program order before the sampled operation.

It is IMPLEMENTATION DEFINED whether or not the profiling operation records the target address of the most recently taken branch instruction in the following cases:

- The sampled operation is not a sampled retired taken branch operation.
- The most recently taken branch instruction was a Context synchronization operation, exception-generating instruction, or exception return.
- No branch instruction has been retired, prior to the sampled operation, since the most recent Context synchronization operation or taken exception.

The profiling operation does not record the target address of the most recently taken branch instruction in the following cases:

- The most recently taken branch instruction was executed when profiling was disabled or prohibited.
- Either the most recently taken branch instruction or the sampled operation is still speculative.

D17.6.4 Additional information for each profiled memory access operation

For an Architecturally executed sampled load, store, or atomic operation that does not generate an exception, the profiling operation records:

- The data virtual and, if enabled, physical addresses being accessed.
  - If the applicable Top Byte Ignore (TBI) bit is set to one, the virtual address includes any top-byte tag.
  - The physical address is the address the PE accesses in the physical address space, and so includes the Secure address space identifier.
- The sampled operation type, which includes:
  - Whether the sampled operation is a load, store, or atomic.
  - When FEAT_SPEv1p3 and FEAT_MTE are implemented, whether the sampled operation is a Tag load or store.
  - When FEAT_NV2 is implemented, whether the sampled operation is a System register access transformed into a memory access.
  - When FEAT_SPEv1p3 and FEAT_MOPS are implemented, whether the sampled operation is a Memory Copy or Memory Set load or store.
  - Whether the sampled operation is Load-Exclusive, Store-Exclusive or Load-acquire, Store-release.
  - Whether the sampled operation accesses the general-purpose or SIMD&FP registers.
- The translation latency. The count of cycles when at least one part of the operation is waiting for the MMU to complete an address translation. It is IMPLEMENTATION DEFINED whether a cycle is counted if a part of the operation is accessing memory, having completed an address translation for that part of the operation. For more information, see Additional information for each profiled memory access operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Whether the sampled operation accessed the Level 1 data cache and the result.
- Whether the sampled operation accessed the data TLB and the result.
- An optional record of whether the sampled operation accessed Last Level data cache and the result.
- An optional record of whether the sampled operation accessed *another socket* in a multi-socket system.
- An optional, IMPLEMENTATION DEFINED, indicator of the data source for a load. If the sampled operation makes multiple accesses, it is IMPLEMENTATION DEFINED whether this indicator combines information for all parts of the load or applies only for a chosen part of the load.
- If FEAT_SPEv1p1 is implemented, an optional indication that the sampled memory operation is non-optimal for the access size. For more information, see Data Alignment Flag.
- If FEAT_SPEv1p2 is implemented, all of the following:
  - An optional record of whether the sampled operation accessed the Level 2 data cache, and the result.
  - An optional record of whether the sampled operation accessed the modified data in a cache, and whether the sampled operation accessed a recently fetched cache line.
  - An optional record of whether the sampled operation snooped data from a cache outside of the cache hierarchy of this PE.
- If FEAT_GCS is implemented, whether the operation includes a Guarded Control Stack data access.

For each of the cache and another socket indicators, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this information is present for store accesses.

For more information, see Events packet.

Note

A store might be marked as not accessing a cache or another socket because it microarchitecturally finished before doing so. For example, the write was placed into a write buffer. This behavior is IMPLEMENTATION DEFINED and might change from time to time, and such events must be interpreted with care.

It is IMPLEMENTATION DEFINED whether a sampled `DC ZVA`, `DC GVA`, `DC GZVA` operation is recorded as a store operation targeting unspecified registers or a cache maintenance operation. The behavior if the sampled operation is recorded as a data cache maintenance operation is described in Additional information for other operations. Arm recommends that these operations are recorded as store operations.

If the sampled load, store, or atomic operation performs multiple accesses, it is IMPLEMENTATION DEFINED whether the implementation chooses to profile all or a chosen part of the access.

If the implementation profiles part of the access:

- It is IMPLEMENTATION DEFINED how the PE chooses the part of the access, while avoiding introducing systematic bias.

  Note

  For an example of inadvertent systematic bias, consider an implementation where a multiple-register load operation is split into multiple accesses. If the PE systematically chooses the first operation at the lower address for sampling translation latency and data source indicator, and the operation is executed in a loop with an incrementing address, then the first access has better spatial locality with preceding accesses than later accesses and is more likely to both:

  - Hit in the TLB, giving a shorter translation latency.
  - Return data from the Level 1 data cache.

  In this case, or if the PE systematically chooses the last access at the higher address, then sampling would be biased.

- If the accesses are architecturally contiguous, it is further IMPLEMENTATION DEFINED whether the recorded data virtual address is the lowest virtual address that is accessed by the sampled operation or applies to the chosen part of the access.
- If the accesses are not architecturally contiguous, the recorded data virtual address applies for the chosen part of the access.
- It is IMPLEMENTATION DEFINED whether the events and total operation latency apply to the whole operation or the chosen part of the operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The translation latency applies to the chosen part of the operation, and is the count of cycles for which the chosen part of the operation is waiting for the MMU to complete an address translation.

Arm recommends that if the implementation chooses to profile a chosen part of the access, then the recorded addresses, events, and total operation latency apply to the chosen access. That is, the PE behaves as if the chosen part of the access is the sampled operation.

If the sampled load, store, or atomic operation performs a single access, or the implementation chooses to profile all parts of a multiple access:

- If the accesses are architecturally contiguous, the recorded data virtual addresses is the lowest virtual address that is accessed by the sampled operation.
- If the accesses are not architecturally contiguous, the recorded data virtual addresses apply for the chosen part of the access.
- The events and total operation latency apply to the whole operation. For example, when recording whether the sampled operation accessed the Level 1 data cache, the PE records whether any part of the access accessed the Level 1 data cache, and the result, and the total operation latency applies from the issue of the operation to the completion of all parts of the operation.
- The translation latency is an IMPLEMENTATION DEFINED choice between:
  - The count of cycles for which at least one part of the operation is waiting for the MMU to complete an address translation, and no part of the operation is accessing memory.
  - The count of cycles for which at least one part of the operation is waiting for the MMU to complete an address translation.

The sampled data physical address packet is not output if any of the following are true:

- The sampled operation operates on a virtual address and any of the following are true:
  - The PE does not translate the address, for example because it does not perform the access.
  - The sampled data virtual address packet is not output.
- Sampling of physical addresses is prohibited by System register controls.

If `AArch64_ExclusiveMonitorsPass()` or `AArch32_ExclusiveMonitorsPass()` returns FALSE for a Store-Exclusive instruction, it is IMPLEMENTATION DEFINED whether or not the physical address packet is output when permitted by the above rules.

If a sampled virtual address packet is not output, or the PE does not perform the access, then all of the following apply:

- It is IMPLEMENTATION DEFINED whether the Translation latency Counter packet for the load or store is either not recorded, or recorded with a value of zero. If the access does not occur and the address is not translated, the packet is not recorded.
- It is IMPLEMENTATION DEFINED whether the bits corresponding to the access in the Events packet are recorded or always zero. If the access does not occur, these bits are zero.

D17.6.4.1 Additional effects when FEAT_MTE is implemented

If profiling is enabled, each Allocation Tag covers multiple locations in a Tag Granule. It is IMPLEMENTATION DEFINED whether the implementation treats each Allocation Tag access as an access to the data location addressed in the operation, or the whole Tag Granule. That is, whether the data virtual address associated with the sampled access or chosen part of the access is the address of the location being accessed, or the lowest address covered by the same Allocation Tag or Allocation Tags.

An instruction that loads or stores an Allocation Tag is treated as a load or store for the purpose of Statistical Profiling.

When a sampled data access virtual address is from a context using Memory Tagging with Address tagging disabled, the TAG byte in the Address packet payload holds bits[63:56] of the address.

If the operation is an access to an Allocation Tag or multiple Allocation Tags, the sampled data physical address has two possibilities:

- The address generated from translating the sampled data virtual address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The address generated from translating the lowest address covered by the same Allocation Tag or Allocation Tags.

If the two possibilities have different values, it is IMPLEMENTATION DEFINED which one is used as the sampled data physical address. Otherwise, the sampled data physical address is the address generated from translating the sampled data virtual address.

If FEAT_SPEv1p3 is implemented, instructions that load or store Allocation Tags are sampled using a dedicated Operation Type packet subclass, (Tags load/store). See Operation Type packet payload (Load/store, general).

Note

Operations that store both Tags and data are recorded as store operations targeting the general-purpose registers or targeting an unspecified registers subclass.

For more information, see The Memory Tagging Extension.

D17.6.4.2 Additional effects when FEAT_NV2 is implemented

When the sampled operation is a System register access transformed into a memory access by the mechanism described in Enhanced support for nested virtualization, the operation is recorded as an `MRS` or `MSR` operation at EL1 transformed into a load/store. If Statistical Profiling is disabled at EL2, the virtual address for the memory access is not recorded.

D17.6.4.3 Additional effects when FEAT_MOPS is implemented

Memory Copy and Memory Set instructions are sampled as retired if they have started processing. This is true even if the execution is not completed at the time of sampling, for example, due to an interrupt.

Memory Copy operations `CPY` and `CPYF` are sampled using Operation Type packet subclass (Memory Copy load/store), and Memory Set operations `SET` and `SETG` are sampled using Operation Type packet subclass (Memory Set load/store).

If a sampled Memory Copy operation performs both a load and a store, then the following are true:

- The sampled data virtual address is:
  - The corresponding one of load and store addresses if the filter controls are set to sample only one of load and store operations, respectively.
  - An unbiased sampling of one of load and store addresses if the filter controls are set to sample both load and store operations.
- The value of the LDST bit in the Operation Type packet is 0 if the source address is sampled, and 1 if the destination address is sampled.

D17.6.4.4 Additional effects when FEAT_LS64, FEAT_LS64_V, and FEAT_LS64_ACCDATA are implemented

FEAT_LS64, FEAT_LS64_V, and FEAT_LS64_ACCDATA add the `LD64B`, `ST64B`, `ST64BV`, and `ST64BV0` instructions. For SPE:

- `LD64B` is treated as a load.
- `ST64B`, `ST64BV`, and `ST64BV0` are treated as stores. `ST64BV` and `ST64BV0` are store with status result instructions and might have performance properties similar to a Device memory load. Sampled `ST64BV` and `ST64BV0` instructions might have the following additional behaviors normally associated with sampled loads:
  - A Data Source packet is generated. This is optional for these instructions. For example, if the Data Source packet can only describe Normal memory sources, then they are not relevant for these instructions.
  - The total latency counter for the sampled operation preferably includes cycles to the point where the return value is returned to the PE.

Note

These behaviors are optional. The information in these packets might not be available or relevant for all implementations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.6.4.5 Additional effects when FEAT_GCS is implemented

If a sampled operation from `GCSSS2` instruction performs both a load and a store, then the following is true:

- The sampled data virtual address is:
  - The corresponding one of load and store addresses if the filter controls are set to sample only one of load and store operations, respectively.
  - An unbiased sampling of one of load and store addresses if the filter controls are set to sample both load and store operations.
- The value of the LDST bit in the Operation Type packet payload depends on the address sampled:
  - If the source address is sampled, the value is 0.
  - If the destination address is sampled, the value is 1.

For an operation executing both a Branch and the Guarded Control Stack data access, if one of the branch and load or store is filtered out, the sampled operation will continue to record the information for both operations. For example, the Total latency packet or Issue latency packet includes the latencies of both the branch and the load or store operations.

D17.6.4.6 Data Alignment Flag

If FEAT_SPEv1p1 is implemented Events packet.E[11] is set to 1 for a sampled memory operation if the address alignment is non-optimal for the access size.

Address alignment is defined as *non-optimal* if that access incurs an additional performance penalty only because of the address alignment, and is unrelated to whether the access is architecturally misaligned for the access size.

Example D17-3 Data Alignment Flag operation

- A 32-bit word access that is not word aligned is architecturally misaligned, but (if Alignment faults are disabled) might not incur an additional penalty because of this alignment unless the word also happens to span a cache-line boundary.
- A contiguous load operation that loads a vector that is the length of two cache lines is optimally aligned if it has cache-line alignment, even though the operation makes two cache line accesses.
- A non-contiguous SVE load operation that makes a sequence of access is optimal only if all of the access are optimal.

The definition of non-optimal is IMPLEMENTATION DEFINED and support for the Alignment Flag is OPTIONAL.

D17.6.5 Additional information for each profiled conditional instruction

For an Architecturally executed sampled conditional operation that finishes execution, the profiling operation records:

- That the sampled operation was conditional.
- Whether the condition passed or failed.

Only operations where the conditionality is controlled by PSTATE.{N, Z, C, V} are treated as conditional operations. Predicated SVE and SME operations are not treated as conditional operations, because the conditionality of the operation is controlled by a predicate

For each of the following operations, it is IMPLEMENTATION DEFINED whether it is treated as a conditional operation:

- Conditional select, both integer and floating-point.
- Conditional select increment.
- Conditional select negation.
- Conditional select inversion.
- Conditional compare, both integer and floating-point.
- Conditional compare negative.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm recommends that all operations where the conditionality is controlled by PSTATE.{N, Z, C, V} are treated as conditional operations.

Micro-ops might have different characteristics from the architectural instructions they are created from. See Information collected for micro-ops for more information.

Note

If the implementation samples architectural instructions, for some alias instructions, the alias specifies the opposite condition to the machine instruction that they alias. For example, CSET and CSETM. The event as captured by SPE always captures the result of the machine instruction, not the alias.

For conditional branches, see Additional information for each profiled branch or exception return.

D17.6.6 Additional information for each profiled SVE and SME operation

ICKWPL When FEAT_SPEv1p1 is implemented, SVE and SME operations are sampled as described in this section.

DWRNLK This section uses the following terms:

Maximum implemented SVE vector length

- The *Maximum implemented Streaming SVE vector length* means the implemented width of the SVE scalable vector registers in Streaming SVE mode.
- The *Maximum implemented Non-streaming SVE vector length* means the implemented width of the SVE scalable vector registers in Non-streaming SVE mode.

See Maximum implemented SVE vector lengths.

Effective SVE vector length (VL)

Means the current SVE vector length, `SVL` or `NSVL`:

- The *Effective Streaming SVE vector length (SVL)* is less than or equal to the Maximum implemented Streaming SVE vector length.
- The *Effective Non-streaming SVE vector length (NSVL)* is less than or equal to the Maximum implemented Non-streaming SVE vector length.

See Configurable SVE vector lengths.

Effective tile size

Means the tile size of the current instruction, determined by `SVL` and the element size used by the instruction. The Effective tile size is less than or equal to the square of the Maximum implemented Streaming SVE vector length, divided by the element size.

Sampled SVE SIMD operation

Means a sampled operation that reads from or writes to the SVE vector or predicate registers but not the SME ZA storage. This includes instructions with scalar outputs, but excludes non-SIMD instructions or operations. For example, the following instructions are in this category:

- `ADD Z0.S, P7/M, Z0.S, Z1.S.`
- `FMLALB Z2.S, Z3.H, Z4.H.`
- `TBL Z5.H, { Z6.H }, Z7.H.`
- `LASTA W0, P6, Z8.S.`
- `LUTI4 {Z8.H-Z9.H}, ZT0, Z10[2].`
- `FMAX {Z0.H-Z1.H}, {Z2.H-Z3.H}, {Z4.H-Z5.H}.`

Sampled SME vector SIMD operation

Means a sampled operation reading from or writing to SME ZA array vectors or tile slices, which is not reading from or writing to a 2D ZA tile. For example, the following instructions are in this category:

- `ADD ZA.S[W8, 3, VGx2], {Z0.S-Z1.S}, {Z2.S-Z3.S}.`
- `FMLAL ZA.S[W9, 4:5, VGx4], {Z4.H-Z7.H}, {Z8.H-Z11.H}.`
- `MOVA Z11.S, P3/M, ZA0H.S[W12,1].`
- `LD1W {ZA1V.S[W13,0]}, P4/Z, [X1,X2,LSL 2].`

Sampled SME tile SIMD operation

Means a sampled operation reading from or writing to a 2D ZA tile or multiple ZA tiles. For example, the following instructions are in this category:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `SMOPA ZA0.S, P0/M, P1/M, Z2.B, Z3.B`.
- `ZERO {ZA1.H}`.

Sampled effective vector length

Means the length in bits of the vector or vectors operated on by a Sampled SVE SIMD operation or a Sampled SME vector SIMD operation. That is, the number of elements in the vector multiplied by the element size.

In cases where the Sampled effective vector length for the input operands of the sampled operation and the Sampled effective vector length for the output operand of the sampled operation differ, the Sampled effective vector length of the output operand is used.

For a Sampled SVE SIMD operation or a Sampled SME vector SIMD operation which reads or writes only SVE predicate registers, the Sampled effective vector length is:

- The length in bits of the predicate operated on multiplied by 8, when predicate-as-mask encoding is used.
- The VL, when predicate-as-counter encoding is used.

Sampled effective tile size

Means the size in bits of the ZA tile or tiles operated on by a Sampled SME tile SIMD operation. That is, the number of elements in the ZA tile or tiles multiplied by the element size.

Sampled predicated operation

Means a Sampled SVE SIMD operation, Sampled SME vector SIMD operation, or Sampled SME tile SIMD operation that is one of:

- An operation that writes to one or more SVE scalable vector registers or ZA tile slices under a Governing predicate or predicates using either zeroing or merging predication.
- A predicated store of a one or more SVE scalable vector registers or ZA tile slices.

For an implementation that samples micro-operations, an SVE or SME instruction might be split up into one or more micro-operations, some of which are predicated and some of which are not predicated.

Note

Sampled predicated operation excludes operations that do not write a vector register, or do so but not using zeroing or merging predication, and applies to machine instructions rather than aliases. For example, the following instructions are not predicated SVE instructions under this definition:

- `CNTP`, `LASTA`, and `PTRUE` do not write to vector registers.
- `FADDV` and `SMAXV` write scalar values to SIMD&FP registers.
- `COMPACT` and `SEL` (vectors) write to vector registers, and have a predicate operand, but do not use that predicate as a Governing predicate for zeroing or merging predication.
- `MOV` (vector, predicated) appears to be a predicated SVE instruction because it specifies merging predication through the `<PG>/M` operand, but it is actually an alias for the `SEL` (vectors) instruction.

IXYGNN

In some implementations, a SIMD instruction might be split into one or more micro-operations, each operating on only a part of the vector or tile specified by the instruction. When such an operation is sampled by the Statistical Profiling Unit (SPU), the Sampled effective vector length or Sampled effective tile size might be smaller than the Effective vector length or Effective tile size specified by `SVL` or `NSVL`.

Furthermore, some instructions might not affect the whole vector or tile specified by the instruction. For example, a predicated operation or an instruction such as `ZERO` (tiles) that specifies a mask in the instruction. This might affect how an individual micro-operation functions, and, for an implementation that samples operations, how the micro-operation is described by the SPU when sampled.

Example D17-4 Vector length

To support a vector length of 1024 bits, an implementation might split all instructions into four micro-operations on 256-bit vector paths. The implementation must, however, implement 1024-bit wide vector registers.

This behavior might vary based on operation type. For example, an implementation that has a full-width data-path for most operations might choose to break certain complex operations, such as non-contiguous load or stores, into shorter vectors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D17-5 Effective SVE vector length less-than the Maximum implemented

To support an Effective SVE vector length less-than the Maximum implemented SVE vector length, an implementation might choose to execute all operations at the Maximum implemented SVE vector length and discard the results above the Effective SVE vector length. Discarded results, arising from difference between Maximum implemented SVE vector length and Effective vector length, do not form part of the sampled operation and the Sampled effective SVE vector length does not include any discarded portions of the vector.

Results discarded because of predication are part of the sampled operation.

I_{DDCQB} If an implementation samples micro-operations, then, because any arbitrary translation of architecture instructions to an equivalent sequence of micro-ops is permitted, it is IMPLEMENTATION SPECIFIC, and might vary between operation types, whether an operation due to a SIMD SVE, SME, or Advanced SIMD instruction is treated as an equivalent Sampled SVE SIMD data-processing operation, a Sampled SME vector SIMD operation, a sampled Advanced SIMD operation, or a sampled scalar operation. This includes SIMD load and store operations.

For example, when all of the following are true, a sampled micro-op generated by a SIMD SVE instruction might be recorded as a sampled Advanced SIMD operation:

- The Sampled effective vector length is 128 bits.
- The operation is unpredicated, and does not have a predicate register as an input or output.
- The operation has an equivalent Advanced SIMD operation.

This might be because the properties of the instruction are not preserved in the micro-ops after the instruction has been decoded, and so the PE cannot determine the classification of the original instruction at the point where an operation is sampled.

Arm recommends that a SIMD micro-op is sampled as a type appropriate for the SIMD instruction which generated the micro-op. See also Information collected for micro-ops.

R_{LMHHC} The division of instructions into micro-operations must be fixed prior to sampling to guarantee consistently accurate statistical sampling.

R_{MDBYZ} A Sampled SVE SIMD data-processing operation generates a Data processing, SVE vector format Operation Type packet, and:

- If FEAT_SPE_SME is implemented, then the Operation Type packet.EVL field records the upper bound of the Sampled effective vector length multiplied by the number of vectors operated on by the sampled operation. If the result is greater than 2048 bits, then the Operation Type packet.EVL field is set to `0b111`.
- If FEAT_SPE_SME is not implemented, then the Operation Type packet.EVL field records the upper bound of the Sampled effective vector length.

If FEAT_SPE_SME is not implemented, then it is IMPLEMENTATION DEFINED whether any of the following Sampled SVE SIMD data-processing operations generate an Other or a Data processing, SVE vector format Operation Type packet:

- A data-processing operation which involves the ZT registers.
- A data-processing operation with a multi-vector operand which accesses only SVE scalable vector registers.

R_{MZJMH} If FEAT_SPE_SME is implemented, then a Sampled SME tile SIMD operation generates a Data processing, SME array format Operation Type packet, and:

- If the operation operates on the whole ZA array at once, then the Operation Type packet.ETS field is set to `0b1111`.
- Otherwise, the Operation Type packet.ETS field records an upper bound of the Sampled effective tile size for the sampled operation.

If FEAT_SPE_SME is not implemented, then a Sampled SME tile SIMD operation generates an Other format Operation Type packet.

Note

The Data processing, SME array format does not record whether the Sampled SME tile SIMD operation was predicated, or not.

R_{CYBGV} If FEAT_SPE_SME is implemented, then a Sampled SME vector data-processing operation generates a Data processing, SME array format Operation Type packet, and the Operation Type packet.ETS field records an upper bound of the Sampled effective vector length multiplied by the number of vectors operated on by the sampled operation.

If FEAT_SPE_SME is not implemented, then a Sampled SME vector data-processing operation generates an Other format Operation Type packet.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The Data processing, SME array format does not record whether the Sampled SME vector data-processing operation was predicated, or not.

R_{LQXZR} A Sampled SVE vector load/store operation generates a Load/store, SVE or SME format Operation Type packet.

R_{MJQCL} If FEAT_SPE_SME is implemented, then a Sampled SME vector load/store operation generates a Load/store, SVE or SME format Operation Type packet.

If FEAT_SPE_SME is not implemented, then it is IMPLEMENTATION DEFINED whether a Sampled SME vector load/store operation generates a Load/store, SVE or SME format or an Unspecified load/store format Operation Type packet.

R_{JCLMP} If FEAT_SPE_SME is implemented and a Sampled SME vector load/store operation or Sampled SVE SIMD load/store operation generates a Load/store, SVE or SME format Operation Type packet, then the Operation Type packet.EVL field records the upper bound of the Sampled effective vector length multiplied by the number of vectors loaded or stored by the sampled operation. If this product is greater than 2048 bits, then the Operation Type packet.EVL field is set to `0b111`.

If FEAT_SPE_SME is not implemented and a Sampled SME vector load/store operation or Sampled SVE SIMD load/store operation generates a Load/store, SVE or SME format Operation Type packet, then the Operation Type packet.EVL field records the upper bound of the Sampled effective vector length.

R_{BHKSC} For a sampled SVE cache prefetch operation:

- The profiling operation captures an IMPLEMENTATION DEFINED subset of the information captured for an SVE load instruction.
- The profiling operation treats the operation type as Other when generating the sample records and filtering based on operation.
- It is IMPLEMENTATION DEFINED whether the operation is treated as a Sampled SVE SIMD operation:
  - If treated as a Sampled SVE SIMD operation, then the Operation Type packet format is Data processing, SVE vector payload.
  - If not treated as a Sampled SVE SIMD operation, then the Operation Type packet format is Other payload.

R_{MYPFF} For a sampled non-contiguous SVE load or store operation that makes multiple memory accesses, the sampled data virtual address is the address accessed by a random one of the load or store operations chosen from the Sampled predicated operation. If the chosen load or store operation is for an Inactive element, the data virtual address packet is not output.

For more information, see Additional information for each profiled memory access operation.

R_{NHSYR} For a sampled contiguous SVE load or store operation that makes multiple memory accesses, the sampled data virtual address is an IMPLEMENTATION DEFINED choice of:

- The address accessed for the lowest element in the Sampled predicated operation.
- The address used for the access containing the lowest Active element in the Sampled predicated operation.

If the corresponding element is an Inactive element, it is IMPLEMENTATION DEFINED whether the data virtual address packet is output.

D17.6.7 Sample operation records for misspeculated and non-architectural operations

It is IMPLEMENTATION DEFINED whether each of the following operations is part of the sample population:

- Operations on misspeculated paths.
- Operations that do not relate to any architecture instruction.

If the operation is part of the sample population, it is further IMPLEMENTATION DEFINED whether the sample record for the sampled operation is captured in the Profiling Buffer. For more information, see Operations that might be excluded from the sample population.

If such an operation is part of the sample population and the sample record is captured in the Profiling Buffer, then some information for the operation might not be present. However, the Events packet and either the End packet or the

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Timestamp packet is always output. Neither event 0 (generated exception) nor event 1 (architecturally retired) will be set in the Events packet.

The record must not contain or infer values that cannot be accessed or used by the sampled operation, as defined by Restrictions on the effects of speculation.

For example:

- An address that is not permitted to be used by the sampled operation, such as might be recorded in an Address packet.
- A condition code or predicate value that is not permitted to be used by the sampled operation, such as might be inferred from the Mispredicted, Partial or empty predicate, or Empty predicate bits in the Events packet.
- A branch target address predicted in a way that is prohibited, such as might be recorded in an Address packet.

D17.6.8 Additional information for other operations

For a sampled operation that does not use any of the Operation Type packet formats defined in the previous sections, it will use the Other format Operation Type packet. This indicates:

- Whether the instruction was conditional or not. See Additional information for each profiled conditional instruction.
- If FEAT_SPE_FPF is implemented, whether the operation is Advanced SIMD or scalar.
- If FEAT_SPE_FPF is implemented, whether the operation is floating-point or not floating-point.

For cache maintenance operations by virtual address, cache prefetch other than SVE cache prefetch, or address translation instructions, the profiling operation:

- Generates an Other format Operation Type packet.
- Captures an IMPLEMENTATION DEFINED subset of the information captured for a load instruction. See Additional information for each profiled memory access operation.
- Is not treated as a load or a store when filtering based on operation type. See Filtering sample records.

D17.6.9 Controlling the data that is collected

Certain data in sample records is collected only if permitted by one or both of EL1 and EL2. This is to restrict exposure of data to a lower Exception level or to Non-secure state.

CONTEXTIDR_EL1 is collected only if PMSCR_EL1.CX is set to 1, the PE is executing at EL1 or EL0 and any of the following are true when an operation is sampled:

- EL2 is not implemented.
- FEAT_SEL2 is implemented and EL2 is disabled for the current Security state.
- The Effective value of HCR_EL2.TGE is 0.

CONTEXTIDR_EL2 is collected only if the Effective value of PMSCR_EL2.CX is 1 and EL2 is implemented and enabled for the current Security state.

This is described in the pseudocode functions `CollectContextIDR1()` and `CollectContextIDR2()`.

Timestamps are collected only if one of the following is true:

- PMSCR_EL1.TS is set to 1 and the Profiling Buffer is owned by EL1.
- PMSCR_EL2.TS is set to 1 and the Profiling Buffer is owned by EL2.

The timestamp is a choice between:

- Physical time, which is defined by the physical count value returned by `PhysicalCountInt()`.
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of `PhysicalCountInt()` minus a physical offset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Virtual time, which is defined as the value of `PhysicalCountInt()` minus a virtual offset.

Table D17-3 summarizes the choice of value for the Timestamp packet when FEAT_ECV is implemented and `StatisticalProfilingEnabled()` is TRUE. In Table D17-3:

Owning EL This is the Exception level that owns the Profiling Buffer. This is returned by the function `ProfilingBufferOwner()`. If EL2 is disabled in the current Security state, this is always EL1.

EL2 enabled This is TRUE when EL2 is enabled in the current Security state. When EL2 is disabled in the current Security state, this is FALSE.

Virtual offset The virtual offset is treated as zero if a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero. Therefore if any of the following are true, the virtual offset is zero:

- EL2 is not implemented.
- The Effective value of HCR_EL2.E2H is 1, and the sampled operation was executed at EL2.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and the sampled operation was executed at EL0.

Otherwise the virtual offset is the value of CNTVOFF_EL2.

Physical offset If any of the following are true, the physical offset is zero:

- EL2 is not implemented.
- FEAT_ECV_POFF is not implemented.
- CNTHCTL_EL2.ECV is 0.
- EL3 is implemented and SCR_EL3.ECVEn is 0.

Otherwise the physical offset is the value of CNTPOFF_EL2.

Table D17-3 Recorded timestamp when FEAT_ECV is implemented

EL2 enabled	Owning EL	PMSCR_EL2		PMSCR_EL1		Recorded timestamp
		PCT[1:0]	TS	PCT[1:0]	TS	
x	EL1	xx	x	xx	0	None
				`0b00`	1	`PhysicalCountInt()` - virtual offset
FALSE	EL1	xx	x	`0b01`	1	`PhysicalCountInt()`
				`0b11`	1	`PhysicalCountInt()` - physical offset
TRUE	EL1	`0b00`	x	xx	1	`PhysicalCountInt()` - virtual offset
		`0b01`	x	`0b01`	1	`PhysicalCountInt()`
				`0b11`	1	`PhysicalCountInt()` - physical offset
		`0b11`	x	`0b01`	1	`PhysicalCountInt()` - physical offset
				`0b11`	1	`PhysicalCountInt()` - physical offset
	EL2	xx	0	xx	x	None
		`0b00`	1	xx	x	`PhysicalCountInt()` - virtual offset
		`0b01`	1	xx	x	`PhysicalCountInt()`
		`0b11`	1	xx	x	`PhysicalCountInt()` - physical offset

If EL2 is not implemented, see the register descriptions of PMSCR_EL1.PCT and PMSCR_EL2.PCT for details of their behavior. This behavior is described by the pseudocode function `CollectTimeStamp()`.

Physical data addresses are collected only if one of the following is true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Effective value of PMSCR_EL1.PA is 1, the Effective value of PMSCR_EL2.PA is 1, and the Profiling Buffer is owned by EL1.
- The Effective value of PMSCR_EL2.PA is 1 and the Profiling Buffer is owned by EL2.

If EL2 is not implemented or is disabled for the owning Security state, then the Effective value of PMSCR_EL2.PA is 1.

Physical data address collection is described by the pseudocode function `CollectPhysicalAddress()`.

Enabling collection of the physical data addresses has an IMPLEMENTATION DEFINED impact on the sampled operation.

D17.6.10 Exceptions

All sample records written to the Profiling Buffer contain the Events packet and either the End packet or the Timestamp packet.

If the sampled operation generates an exception condition, it is UNPREDICTABLE whether the sample record contains any other information. This includes operations that generate faults or other exception conditions but do not generate exceptions. For example:

- An instruction on a misspeculated path.
- A load operation that is part of a Non-fault load instruction or is not the First active element of a First-fault load instruction that generates an MMU fault or watchpoint.
- An address translation operation or prefetch instruction that generates an MMU fault.

Where a sampled operation generates an exception and the type of exception means that a particular item is not calculated by the sampled operation, that information is not collected by the profiling operation. For more information, see Synchronization and Statistical Profiling.

Example D17-6 Translation Faults

If a sampled operation generates a Translation Fault, the physical address for the sampled operation was not generated by the MMU and cannot be recorded.

D17.6.10.1 Non-architectural exceptions

It is IMPLEMENTATION DEFINED whether operations that generate non-architectural exceptions are part of the sample population. If such an operation is part of the sample population, it is further IMPLEMENTATION DEFINED whether the sample record for a sampled operation that generates a non-architectural exception is captured in the Profiling Buffer. For more information, see Operations that might be excluded from the sample population.

If such an operation is part of the sample population and the sample record is captured in the Profiling Buffer, then the sample might record handling of the non-architectural exception. If the sample record does not record handling of the non-architectural exception, then the sampled operation is not Architecturally executed because of the non-architectural exception and it is recorded using E[1] == 0 (operation is not architecturally executed) in the Events packet. Bit E[0] (operation generated an exception) might be used to indicate the operation did not complete because of the non-architectural exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.7 The Profiling Buffer

I_{LSYJL} When the SPU is not in Discard mode, profile data is collected in a memory *Profiling Buffer*. The Profiling Buffer is defined by:

- PMBPTR_EL1, the current write pointer.
- PMBLIMITR_EL1, the write limit pointer.

R_{KLGPG} The Profiling Buffer starts at the current write pointer and extends to the current limit pointer minus one. The write limit pointer must be aligned to the smallest implemented translation granule size. The alignment of the current write pointer is IMPLEMENTATION DEFINED.

I_{JXFJK} If the Effective value of PMBLIMITR_EL1.nVM is 0, then address translation for the Profiling buffer in the owning translation regime is enabled. See Behavior when address translation is enabled.

If the Effective value of PMBLIMITR_EL1.nVM is 1, then address translation for the Profiling buffer in the owning translation regime is disabled. See Behavior when address translation is disabled.

R_{PSLVX} If FEAT_SPE_nVM is not implemented or all of the following apply, then the Effective value of PMBLIMITR_EL1.nVM is 0:

- EL2 is implemented and enabled in the owning Security state.
- The owning Exception level is EL1.
- The Effective value of PMSCR_EL2.EnVM is 0.

D17.7.1 Behavior when address translation is enabled

R_{LWHSF} If the Effective value of PMBLIMITR_EL1.nVM is 0, then PMBLIMITR_EL1 and PMBPTR_EL1 are virtual addresses in the stage 1 translation regime of the owning translation regime.

R_{VGJHQ} If the Effective value of PMBLIMITR_EL1.nVM is 0, other than as described in this section, then the stage 1 translation process for translating virtual addresses and checking for MMU faults is identical to that for any other virtual address in the owning translation regime. For example, PMBPTR_EL1[63:56] are ignored by address translation if the respective TBI bit is set to 1.

I_{TDKDL} If ELx stage 1 translates an address and stage 1 Indirect permissions are used, then the Statistical Profiling Unit indirectly reads PIR_ELx.

R_{FXNVL} The effect of PSTATE.PAN is not applied to the stage 1 Base permissions.

R_{QJZGB} The effect of FEAT_S1POE stage 1 Overlay permissions and POR_ELx are not applied to the access.

D17.7.2 Behavior when address translation is disabled

R_{RPSLW} If the Effective value of PMBLIMITR_EL1.nVM is 1, then the write limit pointer and current write pointer are:

- Physical address in the owning Security state if the owning translation regime has no stage 2 translation.
- Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.

These addresses are output directly by stage 1 without any address translation. Stage 1 translation is disabled for writes to the Profiling Buffer made by the *Statistical Profiling Unit*.

The pseudocode function `AArch64_S1Enabled()` shows this.

R_{SDWFC} If the Effective value of PMBLIMITR_EL1.nVM is 1, PMBMAR_EL1 defines:

- The memory type.
- And as applicable, the following attributes:
  - Cacheability.
  - Shareability.
  - Device type.

The pseudocode function `AArch64_S1DisabledOutput()` shows this.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{FRCPH} If the Effective value of PMBLIMITR_EL1.nVM is 1, the values of SCTLR_ELx.{C, M} for the owning translation regime are ignored for the purposes of determining the Profiling Buffer Cacheability attributes.

I_{DJRBT} If the Effective value of PMBLIMITR_EL1.nVM is 1, it is possible to generate mismatched attributes for a location from within stage 1 of the owning translation regime using PMBMAR_EL1.

Software must be aware of the consequences of and permitted behaviors when accessing a memory location with mismatched attributes. For more information, see Mismatched memory attributes.

R_{QJNFT} If the Effective value of PMBLIMITR_EL1.nVM is 1, `OAMax` is greater than or equal to `AArch64_PAMax()`, and PMBPTR_EL1[`OAMax:AArch64_PAMax()`] is nonzero, a stage 1 Address Size fault is generated when the *Statistical Profiling Unit* attempts to write to memory, where:

- If FEAT_D128 is implemented, then `OAMax` is 55.
- Otherwise, if FEAT_LPA is implemented or FEAT_LPA2 is implemented , then `OAMax` is 51.
- Otherwise, `OAMax` is 47.

For more information, see Implemented physical address size.

R_{XBFCM} If the Effective value of PMBLIMITR_EL1.nVM is 1 and PMBPTR_EL1[`63:(OAMax+1)`] is nonzero when the *Statistical Profiling Unit* attempts to write to the Profiling Buffer, then one of the following CONSTRAINED UNPREDICTABLE behaviors apply:

- A stage 1 Address Size fault is generated.
- PMBPTR_EL1[`63:(OAMax+1)`] are ignored and treated as zero.

The value of `OAMax` is as defined by R_{QJNFT}.

D17.7.3 Effect of stage 2 translation

R_{DQTTB} If the owning translation regime has stage 2 translations, the stage 2 process of the following is identical to that for any other intermediate physical address generated by the owning translation regime:

- Translating the stage 1 output intermediate physical addresses.
- Translating the stage 1 attributes to a physical address and attributes.
- Checking for MMU faults.

R_{MXRVQ} If stage 2 translates an address and stage 2 Overlay permissions are used, then the effect of FEAT_S2POE stage 2 Overlay permissions apply to the memory access and the Statistical Profiling Unit indirectly reads S2POR_EL1.

Example D17-7

The intermediate physical addresses are translated to physical addresses by stage 2 translation, and checked for stage 2 MMU faults.

- If the Effective value of PMBLIMITR_EL1.nVM is 1, meaning stage 1 translation is disabled, and the resulting IPA is translated by a stage 2 Block or Page descriptor with the AssuredOnly attribute set to 1, then the access translated by that descriptor generates a stage 2 Permission fault. See R_{WWYDL} in Assured translation.
- The attributes from stage 1 are combined with the attributes from the stage 2 translation to generate the physical memory attributes.
- If the Effective value of HCR_EL2.DC in the owning translation regime is 1, then stage 1 translation is disabled and the memory type produced by stage 1 is Normal Non-shareable, Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate, regardless of the values of SCTLR_EL1.C and PMBMAR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.7.4 Restrictions on the current write pointer

This section describes the software rules on setting the current write pointer, PMBPTR_EL1. The rules in this section do not apply when Discard mode is enabled. See Discard mode.

R_{PGGHS} If software violates the restrictions in R_{HXSYK} and R_{HLXGD}, then the behavior is CONSTRAINED UNPREDICTABLE, and the PE might do any of the following at any point after profiling is enabled:

- The value of PMBPTR_EL1 becomes UNKNOWN.
- Write sample records to any address in memory that is writable by a privileged access in the owning translation regime. These addresses are:
  - Virtual addresses in the owning translation regime, if the Effective value of PMBLIMITR_EL1.nVM is 0.
  - Intermediate physical addresses in the owning Security state, if the Effective value of PMBLIMITR_EL1.nVM is 1, and the owning translation regime has stage 2 translations.
  - Physical addresses in the owning Security state, if the Effective value of PMBLIMITR_EL1.nVM is 1, and either the owning translation regime has no stage 2 translation or stage 2 translation is disabled.
- Generate a Profiling Buffer management event, with or without indicating data loss, for one of the following reasons:
  - A Buffer full buffer management event. See Buffer full event.
  - An Access not allowed buffer management event. See Access not allowed.
  - Any MMU Fault. See Faults and watchpoints.
  - An IMPLEMENTATION DEFINED buffer management event. See Implementation defined reason.

  If FEAT_SPE_EXC is implemented, then the management event is recorded as described by R_{HJXHT}.
- Silently discard all profiling data without writing it to memory.
- Behave as if profiling is disabled.
- Generate the SAMPLE_BUFFER_FULL PMU event.

R_{HXSYK} When profiling becomes enabled, all the following must be true:

- The current write pointer must be at least one sample record size below the write limit pointer. That is: `UInt`(PMBPTR_EL1.PTR) <= `UInt`(PMBLIMITR_EL1.LIMIT:`Zeros`(12)) - $2^{PMSIDR_EL1.MaxSize}$.
- PMBPTR_EL1.PTR[63:56] must equal PMBLIMITR_EL1.LIMIT[63:56].

If profiling becomes enabled due to a Context synchronization event, then these conditions apply for the values of these System registers after the Context synchronization event.

R_{HLXGD} When the Profiling Buffer is first configured, PMBPTR_EL1.PTR must be aligned to PMBIDR_EL1.Align. That is, if PMBIDR_EL1.Align is nonzero, PMBPTR_EL1.PTR [`UInt`(PMBIDR_EL1.Align)-1:0] must be all zeros.

However, the current write pointer and write limit pointer can usually be restored to the saved write pointer and saved write limit pointer values that were in the System registers when profiling was disabled, providing a `PSB` and a Context synchronization event were executed before reading the System registers.

Following a management event and executing a `PSB` and a Context synchronization event:

- If PMBSR_ELx.{EA, DL} are not {0, 0}, indicating an External abort and/or data loss occurred, then profiling cannot be restarted using the saved write pointer and saved write limit pointer.
- Otherwise, if PMBSR_ELx.S is 0, indicating no Profiling Buffer management event was signaled, then profiling can be restarted from the saved write pointer and saved write limit pointer.
- Otherwise, if software clears PMBSR_ELx.S to 0 before restarting profiling, then the following apply:
  - If the Profiling Buffer management event was caused by an MMU fault, then profiling can be restarted from the saved write pointer.

    Note

    If the MMU fault has not been corrected, then the SPU will generate a MMU fault Profiling Buffer management event when it next tries to write a sample record.

  - If the Profiling Buffer management event was caused by a buffer full event, then profiling cannot be restarted from the saved write pointer and saved limit pointer, as the saved write pointer will not be at least one sample record below the saved limit pointer. However, software can extend the Profiling Buffer by changing PMBLIMITR_EL1, and then profiling can be restarted from the saved write pointer.
- Otherwise, profiling is restarted with PMBSR_ELx.S set to 1, meaning profiling is stopped and there are no constraints on the values of the current write pointer and write limit pointer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If software restarts profiling with a new current write pointer and new write limit pointer, then the restrictions on the current write pointer when first enabling profiling apply.

ELx is defined by R_{HJXHT}.

For more information, see Synchronization and Statistical Profiling and Profiling Buffer management.

D17.7.5 The owning translation regime

R_{WWFKZ} The owning translation regime is defined by the owning Security state and the owning Exception level.

R_{BWRVP} When the Profiling Buffer is enabled, the owning Security state is:

- Non-secure state if and only if one of the following is true:
  - EL3 is not implemented and the PE executes in Non-secure state.
  - FEAT_RME is not implemented, EL3 is implemented, and MDCR_EL3.NSPB is either `0b10` or `0b11`.
  - FEAT_RME is implemented and MDCR_EL3.{NSPBE, NSPB} is either {`0b0`, `0b10`} or {`0b0`, `0b11`}.
- Secure state if and only if one of the following is true:
  - EL3 is not implemented and the PE executes in Secure state.
  - FEAT_RME is not implemented, EL3 is implemented, and MDCR_EL3.NSPB is either `0b00` or `0b01`.
  - FEAT_RME is implemented, Secure state is implemented, and MDCR_EL3.{NSPBE, NSPB} is either {`0b0`, `0b00`} or {`0b0`, `0b01`}.
- Realm state if and only if all of the following are true:
  - FEAT_RME is implemented.
  - MDCR_EL3.{NSPBE, NSPB} is either {`0b1`, `0b10`} or {`0b1`, `0b11`}.

R_{KFMHR} When the Profiling Buffer is enabled, the owning Exception level is:

- EL1 if and only if at least one of the following is true:
  - EL2 is not implemented in the owning Security state.
  - EL2 is disabled in the owning Security state.
  - MDCR_EL2.E2PB is either `0b10` or `0b11`.
- EL2, if and only if all of the following are true:
  - EL2 is implemented and enabled in the owning Security state.
  - MDCR_EL2.E2PB is `0b00`.

R_{TDHBH} When the Profiling Buffer is enabled and the owning Exception level is EL1, all of the following apply:

- The owning translation regime is EL1&0.
- If the Effective value of PMBLIMITR_EL1.nVM is 0, the Profiling Buffer addresses are virtual addresses in the EL1&0 translation regime using the current ASID from TTBRx_EL1.
- If the Effective value of PMBLIMITR_EL1.nVM is 1, the Profiling Buffer addresses are intermediate physical addresses.
- Intermediate physical addresses (whether from the output of stage 1, or the pointers, as applicable) are subject to stage 2 translation using the current VMID if EL2 is implemented and enabled and HCR_EL2.VM is 1.
- Profiling is disabled at the following Exception levels:
  - EL3.
  - EL2.
  - EL0, if EL2 is implemented and enabled and HCR_EL2.TGE is 1.

R_{KRVDC} When the Profiling Buffer is enabled and the owning Exception level is EL2, all of the following apply:

- If the Effective value of HCR_EL2.E2H is 0, the owning translation regime is EL2.
- If the Effective value of HCR_EL2.E2H is 1, the owning translation regime is EL2&0.
- If the Effective value of HCR_EL2.E2H is 0 and the Effective value of PMBLIMITR_EL1.nVM is 0, the Profiling Buffer addresses are virtual addresses in the EL2 translation regime.
- If the Effective value of HCR_EL2.E2H is 1 and the Effective value of PMBLIMITR_EL1.nVM is 0, the Profiling Buffer addresses are virtual addresses in the EL2&0 translation regime using the current ASID from TTBRx_EL2.
- If the Effective value of PMBLIMITR_EL1.nVM is 1, the Profiling Buffer addresses are physical addresses.
- Profiling is disabled at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IYCDWQ When the Profiling Buffer is enabled, all of the following apply:

- If the owning Security state is Non-secure state, profiling is disabled in Secure and Realm states.
- If the owning Security state is Secure state, profiling is disabled in Non-secure and Realm states.
- If the owning Security state is Realm state, profiling is disabled in Non-secure and Secure states.

D17.7.5.1 Summary of the owning translation regime

IYZHFG Table D17-4 summarizes the owning translation regime. In this table:

E is the value of PMBLIMITR_EL1.E.

NSE is the Effective value of SCR_EL3.NSE.

NS is the Effective value of SCR_EL3.NS.

EEL2 is the Effective value of SCR_EL3.EEL2.

NSPBE is the Effective value of MDCR_EL3.NSPBE.

NSPB is the Effective value of MDCR_EL3.NSPB.

E2PB is the Effective value of MDCR_EL2.E2PB.

E2H is the Effective value of HCR_EL2.E2H.

Table D17-4 Summary of owning translation regime (for all Exception levels using AArch64 state)

Controls								Owning translation regime
E	**NSE**	**NS**	**EEL2**	**NSPBE**	**NSPB**	**E2PB**	**E2H**	
0b0	x	x	x	x	x	x	x	Disabled
0b1	0b0	0b0	0b0	0b0	0b0x	x	x	Secure EL1&0
			0b1	0b0	0b0x	0b1x	x	Secure EL1&0
						0b00	0b0	Secure EL2
							0b1	Secure EL2&0
			x	0b0	0b1x	x	x	Disabled in Secure state
		0b1	x	0b0	0b1x	0b1x	x	Non-secure EL1&0
						0b00	0b0	Non-secure EL2
							0b1	Non-secure EL2&0
					0b0x	x	x	Disabled in Non-secure state
	0b1	0b1	x	0b1	0b1x	0b1x	x	Realm EL1&0
						0b00	0b0	Realm EL2
							0b1	Realm EL2&0
					0b0x	x	x	Disabled in Non-secure state
x	x	x	x	0b1	0b0x	x	x	Reserved

D17.7.6 Memory access types and coherency

Writes to any Device memory type by the SPU occur once.

The memory type and attributes that are used for a write by the SPU to the Profiling Buffer is taken from the translation table entries for the virtual address being written to. That is:

- The writes are treated as coming from an observer that is coherent with all observers in the Shareability domain that is defined by the translation tables.
- There is no requirement to manage coherency for observers in the same Shareability domain but coherency for other observers in the system might require explicit management.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see Synchronization and Statistical Profiling.

If FEAT_MTE2 is implemented, a PE will generate a Tag Unchecked access for each access to the Profiling Buffer as part of writing a sample record.

For more information on FEAT_MTE2, see The Memory Tagging Extension.

Writes to the Profiling Buffer are made as privileged writes within the owning translation regime. However, the value of PSTATE.PAN and the FEAT_S1POE stage 1 Overlay permissions are ignored for these writes and treated as zero, see Faults and watchpoints.

This means that if FEAT_E0PD is implemented, the values of TCR_ELx.E0PDy, where ELx is the owning Exception level, do not apply to accesses to the Profiling Buffer made by the SPU.

D17.7.7 Memory access and crossing page boundaries

A memory access from the SPU that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- Each memory access generated by the SPU uses the memory type and Shareability attribute associated with its own address.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation generated the mismatch, or there is only one stage of translation in the owning translation regime, the resulting Buffer Management event is a stage 1 Data Abort.
  - If only the stage 2 translation generated the mismatch, the resulting Buffer Management event is a stage 2 Data Abort.
  - If both stages of translation generate the mismatch, the resulting Buffer Management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- Some or all of the data is discarded. The write pointer is either updated by the amount of data written not including the discarded data or the amount of data written including the discarded data.

A memory access from the SPU to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the SPU are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the SPU are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation causes the boundary to be crossed, or there is only one stage of translation in the owning translation regime, the resulting Buffer Management event is a stage 1 Data Abort.
  - If only the stage 2 translation causes the boundary to be crossed, the resulting Buffer Management event is a stage 2 Data Abort.
  - If both stages of translation cause the boundary to be crossed, the resulting Buffer Management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- Some or all of the data is discarded. The write pointer is either updated by the amount of data written not including the discarded data or the amount of data written including the discarded data.

Note

The boundary referred to is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

If PMSIDR_EL1.MaxSize indicates the same value as PMBIDR_EL1.Align, then records are a fixed power-of-two size and never cross a page boundary.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.7.8 Cache and TLB operations

TLB maintenance operations that affect the TLB of the PE also affect any TLB caching translations for the SPU of that PE.

Cache maintenance operations that affect the caches of the PE also affect data caching by the SPU of that PE.

This means that the completion of any cache or TLB maintenance instruction includes its completion on all SPUs for PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

See also:

- Completion and endpoint ordering.
- Synchronization and Statistical Profiling.

D17.7.9 Effect on the exclusive monitors

If a Load-exclusive instruction or an operation between Load-exclusive and Store-exclusive instructions is sampled, and the sample record is written to an unrelated address, then to avoid a probe effect, Arm recommends that the Store-exclusive does not systematically fail on account of the sampled operation.

If a Store-exclusive instruction is sampled, and the sample record is written to an unrelated address, then the Store-exclusive must not systematically fail on account of the instruction having been sampled.

D17.7.10 Statistical Profiling and MEC

R_{DFFVT} If FEAT_MEC is implemented, accesses made by the Statistical Profiling Unit to the Profiling Buffer are associated with a MECID that is determined by the owning translation regime, owning Security state, and owning Exception level, as defined in Memory Encryption Contexts extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.8 Profiling Buffer management

R_{FJDLW} A Profiling Buffer management event occurs:

- On a fault, see Faults and watchpoints.
- On an External abort, see External aborts.
- When the Profiling Buffer fills, see Buffer full event.
- When the access is not allowed for other reasons, see Access not allowed.
- For IMPLEMENTATION DEFINED reasons, see Implementation defined reason.

I_{QRJYF} The Profiling Buffer management event behavior differs from that of the FEAT_TRBE trace buffer management event.

R_{NXZXG} On a Profiling Buffer management event:

- The service bit, PMBSR_ELx.S, is set to 1.
- The data loss bit, PMBSR_ELx.DL, is set as described in the event description.
- Additional syndrome for the event is written to PMBSR_ELx.MSS.

The value of ELx depends on the type of event, and is defined by R_{HJXHT}. Unless otherwise stated in the event description, other fields in PMBSR_ELx and the other PMBSR_ELx registers are unchanged.

R_{WVKFM} The following Profiling Buffer management events are reported synchronously:

- Buffer full events.
- If FEAT_SPEv1p5 is not implemented, Alignment fault and MMU fault events.

If FEAT_SPEv1p3 is not implemented, it is IMPLEMENTATION DEFINED whether External aborts are reported to the Statistical Profiling Unit synchronously or asynchronously.

Reported synchronously means that profiling is disabled before the Statistical Profiling Unit samples further operations. The interrupt or Profiling exception resulting from recording the Profiling Buffer management event is an asynchronous exception.

When FEAT_SPEv1p5 is implemented, the Statistical Profiling Unit is permitted to sample further operations after the sampled operation that generates the Alignment fault or MMU fault Profiling Buffer management event. This means that one or more samples might be discarded because of the Alignment fault or MMU fault Profiling Buffer management event.

R_{THGCR} The SPU never writes past the PMBLIMITR_EL1 limit pointer.

I_{QRKCL} Software cannot assume that:

- The PE has not modified memory between the current PMBPTR_EL1 write pointer and the PMBLIMITR_EL1 limit pointer minus one.
- The data between the current PMBPTR_EL1 write pointer and the PMBLIMITR_EL1 limit pointer minus one is valid.
- If PMBSR_ELx.DL is set to 1 on a Profiling Buffer management event, that there is any valid data between the end of the last complete sample record and the current PMBPTR_EL1 write pointer. The value of ELx depends on the type of event. See R_{HJXHT}.
- *Any* valid data has been written to the Profiling Buffer if an External abort is reported asynchronously to the SPU. For more information, see The External abort generates a MMU fault.

I_{VJVKZ} The Statistical Profiling Unit generates PMU events and CTI trigger events when a sampled operation completes.

If writing records to the Profiling Buffer generates an MMU fault, one or more of these events might have been generated for one or more sampled operations that are discarded because of the MMU fault.

R_{HJXHT} If FEAT_SPE_EXC is implemented, then when a Profiling Buffer management event is generated:

- If EL3 is implemented and any of the following are true, then the management event is recorded in PMBSR_EL3:
  - MDCR_EL3.PMSEE is `0b11`.
  - MDCR_EL3.PMSEE is `0b10` and the management event is due to a fault on a write to the Profiling Buffer that would generate a Data Abort exception taken to EL3 if generated by a store instruction executed at the owning Exception level. That is, any of the following faults on a write to the Profiling Buffer:
    - Granule Protection Check (GPC) faults, other than Granule Protection Faults (GPFs). For more information, see Granule Protection Check faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

        - If SCR_EL3.GPF is 1, GPFs.
        - If SCR_EL3.EA is 1, External aborts. See also I_{FDVRZ}.
    - MDCR_EL3.PMSEE is `0b10` and the management event is an IMPLEMENTATION DEFINED buffer management event that the implementation elects to record in PMBSR_EL3.
- Otherwise, if all of the following are true, then the management event is recorded in PMBSR_EL2:
    - EL3 is not implemented or MDCR_EL3.PMSEE is not `0b00`.
    - EL2 is implemented and any of the following are true:
        - The Effective value of SCR_EL3.NS is 1.
        - The Effective value of SCR_EL3.{NS, EEL2} is {0, 1}.

        This applies even when the current Exception level is EL3.
    - One of the following is true:
        - PMSCR_EL2.EE is `0b11`.
        - PMSCR_EL2.EE is `0b10` and the management event is due to any fault on a write to the Profiling Buffer that would generate a Data Abort exception taken to EL2 if generated by a store instruction executed at the owning Exception level.
          If the owning Exception level is EL2, this means any fault on a write to the Profiling Buffer.
          If the owning Exception level is EL1, this means any of the following faults on a write to the Profiling Buffer:
            - Stage 2 faults.
            - If HCR_EL2.GPF is 1, GPFs.
            - If HCR_EL2.TEA is 1, External aborts. See I_{FDVRZ}.
        - PMSCR_EL2.EE is `0b10` and the management event is due to a GPC fault other than a GPF.
        - PMSCR_EL2.EE is `0b10` and the management event is an IMPLEMENTATION DEFINED buffer management event that the implementation elects to record in PMBSR_EL2.
- Otherwise, the management event is recorded in PMBSR_EL1.

If FEAT_SPE_EXC is not implemented, then the management event is recorded in PMBSR_EL1.

The pseudocode function `ReportSPEEvent()` shows this.

I_{FDVRZ} When FEAT_SPEv1p3 is implemented, External aborts are not reported on writes to the trace buffer. However, External aborts on translation table walks, translation table updates, and GPT walks can be reported as MMU faults. See Faults and watchpoints.

I_{JVQRY} Table D17-5, Table D17-6, and Table D17-7 show which PMBSR_ELx register used to record the syndrome for each type of Profiling Buffer management event.

Each column in the left side of the tables refers to the Effective value of a field in a System register.

Each column in the right side of the tables represents a Profiling Buffer management event:

Other event

Means a Profiling Buffer management event recorded with the EC value `0b000000`.

Stage 1 (GPF)

Means a Profiling Buffer management event for a Granule Protection Fault (GPF) reported as a stage 1 abort on a write to the buffer, recorded with the EC value of `0b100100` and an FSC value indicating a GPF.

Stage 1 (External abort)

Means a Profiling Buffer management event for an External abort reported as a stage 1 abort on a write to the buffer, recorded with the EC value of `0b100100` and an FSC value indicating an External abort.

Stage 1 (Other abort)

Means a Profiling Buffer management event for any other stage 1 abort on a write to the buffer, recorded with the EC value `0b100100`.

Stage 2 (GPF)

Means a Profiling Buffer management event for a GPF reported as a stage 2 abort on a write to the buffer, recorded with the EC value of `0b100101` and an FSC value indicating a GPF.

Stage 2 (External abort)

Means a Profiling Buffer management event for an External abort reported as a stage 2 abort on a write to the buffer, recorded with the EC value of `0b100101` and an FSC value indicating an External abort. That is, a synchronous External abort on a stage 2 translation table walk or translation table update.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Stage 2 (Other abort)

Means a Profiling Buffer management event for any other stage 2 abort on a write to the buffer, recorded with the EC value `0b100101`.

GPC fault

Means a Profiling Buffer management event for a Granule Protection Check fault, other than a GPF, on a write to the buffer, recorded with the EC value `0b011110`.

The values in these cells denotes which PMBSR_ELx register is updated for that combination of control register values and Profiling Buffer management event type.

Table D17-5 SPE Profiling Buffer management event reporting (Granule Protection Checks)

MDCR_EL3.PMSEE	SCR_EL3.GPF	PMSCR_EL2.EE	MDCR_EL2.E2PB	HCR_EL2.GPF	GPF		GPC fault
					Stage 1	Stage 2	
0b00	X	XX	XX	X	EL1	EL1	EL1
0b01	X	0b0X	XX	X	EL1	EL1	EL1
0b01	X	0b10	0b00	X	EL2	EL2	EL2
0b01	X	0b10	0b1X	0b0	EL1	EL2	EL2
0b01	X	0b10	0b1X	0b1	EL2	EL2	EL2
0b01	X	0b11	XX	X	EL2	EL2	EL2
0b10	0b0	0b0X	XX	X	EL1	EL1	EL3
0b10	0b0	0b10	0b00	X	EL2	EL2	EL3
0b10	0b0	0b10	0b1X	0b0	EL1	EL2	EL3
0b10	0b0	0b10	0b1X	0b1	EL2	EL2	EL3
0b10	0b0	0b11	XX	X	EL2	EL2	EL3
0b10	0b1	XX	XX	X	EL3	EL3	EL3
0b11	X	XX	XX	X	EL3	EL3	EL3

Table D17-6 SPE Profiling Buffer management event reporting (External aborts)

MDCR_EL3.PMSEE	SCR_EL3.EA	PMSCR_EL2.EE	MDCR_EL2.E2PB	HCR_EL2.TEA	External abort	
					Stage 1	Stage 2
0b00	X	XX	XX	X	EL1	EL1
0b01	X	0b0X	XX	X	EL1	EL1
0b01	X	0b10	0b00	X	EL2	EL2
0b01	X	0b10	0b1X	0b0	EL1	EL2
0b01	X	0b10	0b1X	0b1	EL2	EL2
0b01	X	0b11	XX	X	EL2	EL2
0b10	0b0	0b0X	XX	X	EL1	EL1
0b10	0b0	0b10	0b00	X	EL2	EL2
0b10	0b0	0b10	0b1X	0b0	EL1	EL2
0b10	0b0	0b10	0b1X	0b1	EL2	EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MDCR_EL3.PMSEE	SCR_EL3.EA	PMSCR_EL2.EE	MDCR_EL2.E2PB	HCR_EL2.TEA	External abort	
					Stage 1	Stage 2
0b10	0b0	0b11	XX	X	EL2	EL2
0b10	0b1	XX	XX	X	EL3	EL3
0b11	X	XX	XX	X	EL3	EL3

Table D17-7 SPE Profiling Buffer management event reporting (All other management events)

MDCR_EL3.PMSEE	PMSCR_EL2.EE	MDCR_EL2.E2PB	Other abort		Other event
			Stage 1	Stage 2	
0b00	XX	XX	EL1	EL1	EL1
0b01	0b0X	XX	EL1	EL1	EL1
0b01	0b10	0b00	EL2	EL2	EL1
0b01	0b10	0b1X	EL1	EL2	EL1
0b01	0b11	XX	EL2	EL2	EL2
0b10	0b0X	XX	EL1	EL1	EL1
0b10	0b10	0b00	EL2	EL2	EL1
0b10	0b10	0b1X	EL1	EL2	EL1
0b10	0b11	XX	EL2	EL2	EL2
0b11	XX	XX	EL3	EL3	EL3

R_{MYXZK} When a write to the Profiling Buffer by the Statistical Profiling Unit generates an Alignment fault or MMU fault, the PE sets PMBPTR_EL1 to the faulting address.

The last byte of the last complete sample record, including any padding bytes after the record, is at most $2^{(\text{PMSIDR_EL1.MaxSize})}$ bytes below PMBPTR_EL1.

If PMBPTR_EL1 does not point to the first byte after the last complete record, including any padding after the record, then PMBSR_ELx.DL is set to 1. Otherwise, PMBSR_ELx.DL is set to an IMPLEMENTATION DEFINED choice of 0 and 1.

ELx is defined by R_{HJXHT}. See also Table D17-6.

R_{YZJQX} When FEAT_SPE_EXC is implemented and any of the following are true, then the SPE Profiling exception is *enabled* and pending:

- All of the following are true:
  - EL3 is implemented.
  - MDCR_EL3.PMSEE is `0b10` or `0b11`.
  - PMBSR_EL3.S is 1.

  In this case:
  - The target Exception level is EL3.
  - If all of the following are true, then the SPE Profiling exception is *unmasked*:
    - The PE is executing at EL2, EL1, or EL0.
    - The PE is in Non-debug state.
- All of the following are true:
  - EL3 is not implemented or MDCR_EL3.PMSEE is not `0b00`.
  - EL2 is implemented and enabled in the current Security State.
  - PMSCR_EL2.EE is `0b10` or `0b11`.
  - PMBSR_EL2.S is 1.

  In this case:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The target Exception level is EL2.
  - If the PE is executing in the owning Security state, the PE is in Non-debug state, and any of the following are true, then the SPE Profiling exception is unmasked:
    - The PE is executing at EL1 or EL0.
    - The PE is executing at EL2, PMSCR_EL2.EE is `0b11`, PMSCR_EL2.KE is 1, and PSTATE.PM is 0.
- All of the following are true:
  - EL3 is not implemented or MDCR_EL3.PMSEE is not `0b00`.
  - EL2 is not implemented, or EL2 is disabled in the current Security State, or PMSCR_EL2.EE is not `0b00`.
  - PMSCR_EL1.EE is `0b11`.
  - PMBSR_EL1.S is 1.

  In this case:
  - The target Exception level is EL2 if the Effective value of HCR_EL2.TGE is 1, and EL1 otherwise.
  - If the PE is executing in the owning Security state, the PE is in Non-debug state, and any of the following are true, then the SPE Profiling exception is unmasked:
    - The PE is executing at EL0.
    - The PE is executing at EL1, PMSCR_EL1.KE is 1, and PSTATE.PM is 0.

When none of the above apply, the SPE Profiling exception is *disabled* and not pending.

When the SPE Profiling exception is enabled and unmasked, the SPE Profiling exception will be taken to the *target Exception level*.

When the SPE Profiling exception is enabled and not unmasked, then the SPE Profiling exception is *masked*. The SPE Profiling exception is always masked at EL3.

The pseudocode function `CheckForSPEException()` shows this.

IGMBMC

Table D17-8, Table D17-9, and Table D17-10 show the SPE Profiling exception enable and masking for each Exception level, for the Effective values of each of the PMBSR_EL1.S, PMBSR_EL2.S, and PMBSR_EL3.S bits respectively.

Each column in the left side of the table refers to the Effective value of a field in a System register or PSTATE. If FEAT_SPE_EXC is not implemented, then the Effective values of PMBSR_EL2.S and PMBSR_EL3.S are 0.

In Table D17-8, the IRQ column represents the state of the **PMBIRQ** signal. Table D17-8 does also apply when FEAT_SPE_EXC is disabled. In these cells for each row representing a combination of control bits:

HIGH
: Means the SPE Profiling exception is disabled and the SPE interrupt request is enabled and asserted.

LOW
: Means the SPE interrupt request is not asserted.

The other columns in the right side of the table represent an Exception level the exception is taken from. In each cell for each of these columns for each row representing a combination of control bits:

None
: Means the exception is disabled. For Table D17-9 and Table D17-10 this means that the PMBSR_EL3.S or PMBSR_EL2.S bit (as applicable) is ignored.

B
: Means the exception is enabled and masked by PSTATE.PM. The exception remains pending.

C
: Means the exception is enabled and masked regardless of the value of PSTATE.PM. The exception remains pending.

ELx
: Means the exception is enabled, is not masked, and is taken to ELx.

n/a
: Not applicable. The PE cannot be executing at this Exception level for the specified value of HCR_EL2.TGE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D17-8 SPE Profiling exception and interrupt, taken from EL0, EL1, EL2 or EL3, when the Effective value of PMBSR_EL1.S is 1

MDCR_EL3.PMSEE	PMSCR_EL2.EE	HCR_EL2.TGE	PMSCR_EL1		PSTATE.PM	IRQ	Taken from			
			EE	KE			EL0	EL1	EL2	EL3
0b00	XX	0b0	XX	X	X	HIGH	None	None	None	None
0b00	XX	0b1	XX	X	X	HIGH	None	n/a	None	None
!=0b00	0b00	0b0	XX	X	X	HIGH	None	None	None	None
!=0b00	0b00	0b1	XX	X	X	HIGH	None	n/a	None	None
!=0b00	!=0b00	0b0	0b0X	X	X	HIGH	None	None	None	None
!=0b00	!=0b00	0b0	0b1X	0b0	X	LOW	EL1	C	C	C
!=0b00	!=0b00	0b0	0b1X	0b1	0b0	LOW	EL1	EL1	C	C
!=0b00	!=0b00	0b0	0b1X	0b1	0b1	LOW	EL1	B	C	C
!=0b00	!=0b00	0b1	0b0X	X	X	HIGH	None	n/a	None	None
!=0b00	!=0b00	0b1	0b1X	X	X	LOW	EL2	n/a	C	C

Table D17-9 SPE Profiling exception, taken from EL0, EL1, EL2 or EL3, when the Effective value of PMBSR_EL2.S is 1

MDCR_EL3.PMSEE	PMSCR_EL2		HCR_EL2.TGE	PSTATE.PM	Taken from			
	EE	KE			EL0	EL1	EL2	EL3
0b00	XX	X	0b0	X	None	None	None	None
0b00	XX	X	0b1	X	None	n/a	None	None
!=0b00	0b0X	X	0b0	X	None	None	None	None
!=0b00	0b0X	X	0b1	X	None	n/a	None	None
!=0b00	0b10	X	0b0	X	EL2	EL2	C	C
!=0b00	0b10	X	0b1	X	EL2	n/a	C	C
!=0b00	0b11	0b0	0b0	X	EL2	EL2	C	C
!=0b00	0b11	0b0	0b1	X	EL2	n/a	C	C
!=0b00	0b11	0b1	0b0	0b0	EL2	EL2	EL2	C
!=0b00	0b11	0b1	0b0	0b1	EL2	EL2	B	C
!=0b00	0b11	0b1	0b1	0b0	EL2	n/a	EL2	C
!=0b00	0b11	0b1	0b1	0b1	EL2	n/a	B	C

Table D17-10 SPE Profiling exception, taken from EL0, EL1, EL2 or EL3, when the Effective value of PMBSR_EL3.S is 1

MDCR_EL3.PMSEE	HCR_EL2.TGE	Taken from			
		EL0	EL1	EL2	EL3
0b0X	0b0	None	None	None	None
0b0X	0b1	None	n/a	None	None

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MDCR_EL3.PMSEE	HCR_EL2.TGE	Taken from			
		EL0	EL1	EL2	EL3
0b10	0b0	EL3	EL3	EL3	C
0b10	0b1	EL3	n/a	EL3	C
0b11	0b0	EL3	EL3	EL3	C
0b11	0b1	EL3	n/a	EL3	C

I_{SFJPD} When FEAT_SPE_EXC is implemented, the target Exception level might be the same as the owning Exception level, or a higher Exception level, or a lower Exception level. For example:

- The owning Exception level is EL1, but EL2 triages all SPE Profiling exceptions and so programs PMSCR_EL2.EE to `0b11`.
- The owning Exception level is EL2, because EL2 has mapped the Profiling Buffer in the EL2 translation regime, but EL1 can handle all non-MMU management events, and so EL2 programs PMSCR_EL2.EE to `0b10`.

R_{YMRRB} When FEAT_SPE_EXC is implemented, SPE Profiling exceptions are taken asynchronously and precisely.

R_{NXPMV} When FEAT_SPE_EXC is implemented, if an SPE Profiling exception is enabled and unmasked, and all of the following are true, then the SPE Profiling exception will be taken before an instruction **C** completes execution:

- Instruction **A** is one of the following:
  - A direct read of the corresponding PMBSR_ELx register that observes that the S bit is 1.
  - A direct write of the corresponding PMBSR_ELx register that sets the S bit to 1.
  - A `PSB CSYNC` instruction that synchronizes a profiling operation that causes a Profiling Buffer management event that sets the corresponding PMBSR_ELx.S bit to 1. For more information, see Synchronization and Statistical Profiling.
- **C** follows in program order immediately after a Context synchronization event **B**.
- **B** is in program order after **A**.

Otherwise, the SPE Profiling exception is taken in finite time.

R_{TSJRP} If any of the following are true, then the SPE interrupt request (**PMBIRQ**) is enabled and driven from PMBSR_EL1.S:

- FEAT_SPE_EXC is not implemented.
- EL3 is implemented and MDCR_EL3.PMSEE is `0b00`.
- EL2 is implemented, PMSCR_EL2.EE is `0b00`, and any of the following are true:
  - The Effective value of SCR_EL3.NS is 1.
  - The Effective value of SCR_EL3.{NS, EEL2} is {0, 1}.

  This applies even when the current Exception level is EL3.
- PMSCR_EL1.EE is `0b00`.

Otherwise, if FEAT_SPE_EXC is implemented, then the SPE interrupt request is disabled and driven LOW (inactive). The pseudocode function `SPEInterruptEnabled()` shows this.

I_{GRLRX} When the **PMBIRQ** is enabled:

- A direct write that sets PMBSR_EL1.S to 1 causes the interrupt to be asserted.
- **PMBIRQ** will remain asserted until software clears PMBSR_EL1.S to 0.

I_{SMLLS} If a Generic Interrupt Controller (GIC) is implemented, **PMBIRQ** must be configured as a Private Peripheral Interrupt (PPI) in a multiprocessor system. **PMBIRQ** is signaled by the PE that implements the SPU.

Note

A standard PPI number is allocated by the *Arm® Base System Architecture* (BSA).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.8.1 Prioritization of Profiling Buffer management events

R_{DTGVZ} Where multiple synchronous Profiling Buffer management events occur on writing a sample record, the PE prioritizes them as follows (from highest to lowest priority):

1. Synchronous fault.
2. Synchronous External abort.
3. Buffer full event.

I_{XQRTS} Asynchronous External aborts are not prioritized with respect to other events.

I_{CTRLW} When a Profiling Buffer management event generates an SPE interrupt request, the priority of the SPE interrupt request is separately managed by the interrupt controller. Do not confuse the prioritization of Profiling Buffer management events with the prioritization of SPE interrupt requests by an interrupt controller.

D17.8.2 Buffer full event

If, after writing a sample record, there is not sufficient space in the Profiling Buffer for a sample record of the size indicated by PMSIDR_EL1.MaxSize, and PMBSR_ELx.S is 0, a Profiling Buffer management event is generated:

- PMBSR_ELx.S is set to 1.
- PMBSR_ELx.EC is set to `0b000000`, other buffer management event.
- The BSC field of PMBSR_ELx.MSS is set as follows:
  - PMBSR_ELx.BSC is set to `0b000001`, buffer filled.
- PMBPTR_EL1 is set to the first byte after the last complete sample record, which can include any padding that the implementation uses to force alignment. PMBSR_ELx.DL is unchanged.
- The other fields in PMBSR_ELx and the other PMBSR_ELx registers are unchanged.
- The SAMPLE_BUFFER_FULL PMU event is generated.

ELx is defined by R_{HJXHT}. See also Table D17-7.

That is, the Profiling Buffer management event is generated when the PE writes past the write limit pointer minus $2^{(PMSIDR_EL1.MaxSize)}$. The SPU never writes beyond the write limit pointer.

For more information, see Restrictions on the current write pointer.

D17.8.3 Faults and watchpoints

Table D17-11 lists the faults that might be generated by a write to the Profiling Buffer by the SPU.

Writes to the Profiling Buffer never generate watchpoints.

Table D17-11 Faults

Fault	Conditions
Translation	The translation of a virtual address to a physical address might generate a Translation fault.
Address Size	The translation of a virtual address to a physical address might generate an Address Size fault.
Alignment	If PMBPTR_EL1 is not aligned to an IMPLEMENTATION DEFINED minimum alignment, the behavior is UNPREDICTABLE and a write to the Profiling Buffer by the SPU might generate an Alignment fault. For more information, see Restrictions on the current write pointer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Fault	Conditions
Permission	Writes to the Profiling Buffer are made as privileged writes. If the write does not have write permission, a Permission fault is generated. The value of PSTATE.PAN is treated as zero and the FEAT_S1POE stage 1 Overlay permissions are ignored. If the SPU does not manage the dirty state in translation tables, then accesses ignore the Dirty Bit Modifier bit in Page and Block descriptors and an access might as a result generate a Permission fault. For more information, see Hardware management of the dirty state.[a]
Access flag	If the SPU does not manage the Access flag in translation tables or hardware management of the Access flag state is disabled for the owning translation regime, then any access to a Page or Block with the Access Flag bit set to 0 in a descriptor will generate an Access Flag fault. For more information, see The Access flag.[a]
TLB Conflict fault	IMPLEMENTATION DEFINED.
Unsupported atomic hardware update fault	If hardware update of the translation tables is not guaranteed atomic in regard to other agents that access the memory, the translation of a virtual address to a physical address might generate an Unsupported atomic hardware update fault.
External abort on translation table walk, translation table update, or GPT walk	The translation of a virtual address to a physical address might generate an External abort on the translation table walk, translation table update, or GPT walk, and be treated as a synchronous MMU fault. For more information, see External aborts.
Granule Protection Check	The translation of a virtual address to a physical address might generate a GPC fault.

a. PMBIDR_EL1.F defines whether the SPU manages the Access flag and dirty state in the translation tables.

If a write to the Profiling Buffer generates an Alignment fault or MMU fault, including a Granule Protection Check (GPC) fault, and PMBSR_ELx.S is 0, then all of the following occur:

- A Profiling Buffer management event is generated. This sets PMBSR_ELx.S is set to 1.
- PMBSR_ELx.EC is set to one of:
  - `0b011110`, GPC fault, other than a Granule Protection Fault (GPF), on write to the Profiling Buffer.
  - `0b100100`, stage 1 Data Abort on write to the Profiling Buffer.
  - `0b100101`, stage 2 Data Abort on write to the Profiling Buffer.

  A GPF on translation table walk or update is reported as either a stage 1 or stage 2 Data Abort, as appropriate. Other GPFs are reported as a stage 1 Data Abort.
- The FSC field of PMBSR_ELx.MSS is set as follows:
  - PMBSR_ELx.FSC is set to indicate the type of the fault.
- If FEAT_THE is implemented, then PMBSR_ELx.TopLevel is set as follows:
  - 1, for a stage 2 Permission fault due to a TopLevel check.
  - 0, otherwise.
- If FEAT_THE is implemented, then PMBSR_ELx.AssuredOnly is set as follows:
  - 1, for a stage 2 Permission fault due to the AssuredOnly attribute.
  - 0, otherwise.
- PMBPTR_EL1 is set to the address that generated the fault.
- If PMBPTR_EL1 is not the address of the first byte after the last complete sample record, written by the SPU and including any padding used to force alignment, then PMBSR_ELx.DL is set to 1. Otherwise, PMBSR_ELx.DL is unchanged.

If the MMU fault is an External abort on a translation table walk, translation table update, or GPT walk, then it is IMPLEMENTATION DEFINED whether PMBSR_ELx.EA is set to 1 or unchanged.

The value of ELx depends on the type of event, and is defined by R_{HJXHT}. The other fields in PMBSR_ELx and the other PMBSR_ELx registers are unchanged.

Note

Each of these faults causes a Profiling Buffer management event, not an actual MMU fault exception. The ESR and FAR registers are unchanged.

For more information, see MMU fault-checking sequence.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.8.3.1 Hardware management of dirty state and the Access flag by the Statistical Profiling Extension

If FEAT_SPEv1p3 is implemented, address translations performed by the SPU manage dirty state and the Access flag in Block and Page descriptors. Otherwise, it is IMPLEMENTATION DEFINED whether address translations performed by the SPU manage dirty state and the Access flag. This is discoverable by software using PMBIDR_EL1.F. See Hardware management of the dirty state and Hardware management of the Access flag.

If the SPU manages dirty state and hardware management of dirty state is enabled for the owning translation regime, then the SPU can speculatively update the Translation Table descriptor for any Page or Block in the Statistical Profiling buffer before writing data to it, if the write is otherwise permitted. This includes the case where a buffer management event means the SPU stops writing data before the page or block is written to. For more information, see The Profiling Buffer.

If FEAT_HAFT is implemented and the SPU manages dirty state, for an architecturally executed memory access that is translated by a translation stage with Table descriptor Access flag hardware management enabled, the SPU sets the Access flag to 1 in all Table descriptors accessed during the translation table walk for that access that have the Access flag set to 0.

D17.8.4 External aborts

R_{JFYNW} When a write to the Profiling Buffer generates an External abort, including an External abort on a translation table walk, translation table update, or GPT walk, the permitted IMPLEMENTATION DEFINED behaviors are:

- The External abort is ignored.
- The External abort generates an SError exception.
- For an External abort on a translation table walk, translation table update, or GPT walk only, the External abort generates a MMU fault.
- If FEAT_SPEv1p3 is not implemented, the External abort is reported to the SPU. The SPU generates a Profiling Buffer management event.

I_{KDMQG} PMBIDR_EL1.EA indicates to software the mechanism used by the SPU when a write to memory generates an External abort that does not generate a MMU fault.

Note

The originator of the abort might additionally take other actions. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

D17.8.4.1 The External abort is ignored

I_{VPLMK} If the External abort is ignored, this has the same visible behavior as when a write does not generate any External abort. PMBSR_ELx is not modified.

D17.8.4.2 The External abort generates an SError exception

I_{ZGGXW} If a write to the Profiling Buffer generates an External abort that is taken as an SError exception, the PE takes the SError exception as normal, and PMBSR_ELx fields are unchanged.

D17.8.4.3 The External abort generates a MMU fault

I_{QNXKH} When an External abort on a translation table walk, translation table update, or GPT walk generates a MMU fault, the behavior is as described by Faults and watchpoints.

D17.8.4.4 The External abort is reported to the SPU

I_{TCPLL} The behavior in this section is not permitted if FEAT_SPEv1p3 is implemented.

R_{BPFPN} If a write to the Profiling Buffer generates an External abort that is reported to the SPU:

- The External abort bit, PMBSR_ELx.EA, is set to 1.
- The SPU stops writing sample records to the Profiling Buffer. It is IMPLEMENTATION DEFINED whether an External abort on a write to the Profiling Buffer is reported as synchronous or asynchronous:
  - The External abort is reported as *synchronous* if PMBPTR_EL1 is set to the address that was externally aborted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The External abort is reported as *asynchronous* if PMBPTR_EL1 is not guaranteed to be set to the address that was externally aborted.

- If the External abort is reported as asynchronous or PMBPTR_EL1 is not the address of the first byte of the sample record being written by the SPU, then PMBSR_ELx.DL is set to 1. Otherwise, PMBSR_ELx.DL is unchanged.

Note

Following an External abort reported asynchronously to the SPU, software must not assume that any valid data has been written to the Profiling Buffer.

- If PMBSR_ELx.S is 0, a buffer management event is generated:
  — PMBSR_ELx.S is set to 1.
  — PMBSR_ELx.EC is set to one of:
    — If the External abort is on a GPT fetch, `0b011110`, Granule Protection Check fault on write to Profiling Buffer, other than Granule Protection Fault (GPF).
    — If the External abort is on a stage 2 translation table access, `0b100101`, stage 2 Data Abort on write to buffer.
    — Otherwise, `0b100100`, stage 1 Data Abort on write to buffer.
  — PMBSR_ELx.MSS is set as follows:
    — PMBSR_ELx.FSC is set to indicate a synchronous or asynchronous External abort.

ELx is defined by R_{HJXHT}. See also Table D17-6. Other fields in PMBSR_ELx and the other PMBSR_ELx registers are unchanged.

I_{XPWQC} If External aborts are reported as asynchronous:

- The External abort might not be received until after a first Profiling Buffer management event has set PMBSR_ELx.S to 1.
- Writes to the buffer might generate a second Profiling Buffer management event after the External abort has set PMBSR_ELx.S to 1.

D17.8.5 Access not allowed

An SPU might detect that a write to the Profiling Buffer cannot occur, usually because software has created an architecturally UNPREDICTABLE situation. In these cases, when profiling data is generated, the SPU might discard the data and generate a Profiling Buffer management event:

- If PMBSR_ELx.S is 0, then all of the following occur:
  — PMBSR_ELx.S is set to 1.
  — PMBSR_ELx.DL is set to 1.
  — PMBSR_ELx.EC is set to `0x00`, other buffer management event.
  — PMBSR_ELx.BSC is set to `0b000000`, access not allowed.
- The other fields in PMBSR_ELx and the other PMBSR_ELx registers are unchanged.

ELx is defined by R_{HJXHT}. See also Table D17-7.

D17.8.6 Implementation defined reason

An implementation might include support for generating Profiling Buffer events for IMPLEMENTATION DEFINED reasons. These should normally be disabled unless enabled by IMPLEMENTATION DEFINED controls. The PMBSR_ELx.EC code `0x1F` is reserved for this purpose.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D17.9 Synchronization and Statistical Profiling

I_{RRBSZ} The profiling operation of the SPU:

- Makes indirect reads and indirect writes of System registers.
- Writes to memory.
- Makes further indirect writes to PMBPTR_EL1 as a result of an External abort on a write to memory.

I_{WZSLB} The indirect reads of the PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} controls when determining whether to select an operation for profiling are treated as indirect reads made by the instruction being executed, and subject to the standard requirements for synchronization.

Otherwise, although the profiling operation is generated by a sampled operation, the profiling operation executes independently of the instructions that are executed on the PE.

I_{NRNSL} A `DSB` instruction guarantees that all memory transactions that are made by the PE are observable by writes made by a profiling operation relating to a sampled operation in program order after the `DSB` instruction.

I_{NDWCG} A Context Synchronization event guarantees that a direct write to a System register made by the PE in program order before the Context synchronization event are observable by indirect reads and indirect writes of the same System register made by a profiling operation relating to a sampled operation in program order after the Context synchronization event.

R_{TRDTK} To synchronize previous profiling operations, software must execute a `PSB` Buffer Synchronization instruction.

Note

The `PSB CSYNC` instruction is not defined in the AArch32 instruction set architecture.

I_{MLGHW} Following a Context synchronization event, a `PSB` instruction is guaranteed to synchronize the profiling operations for all instructions that are executed in program order before the Context synchronization event.

Synchronized by the `PSB` instruction means:

- A direct read of a System register in program order following a `PSB` instruction requires explicit synchronization to observe an indirect write to the same System register made by a profiling operation synchronized by the `PSB` instruction.
- An indirect write to a System register made by a profiling operation synchronized by a `PSB` instruction does not affect a direct write to the same System register made in program order following the `PSB` instruction.
- A direct write to a System register in program order following a `PSB` instruction is not allowed to affect an indirect read of the same System register made by a profiling operation synchronized by the `PSB` instruction.
- A `DSB` instruction in program order following a `PSB` instruction does not complete before the writes to the Profiling Buffer of sample records for profiling operations synchronized by the `PSB` instruction have completed. The `DSB` instruction must apply to both loads and stores.

For the indirect write to PMBSR_ELx that is made as a result of an External abort on a write of a sample record to memory, the synchronization rules apply only after the write has completed. Completion of the write includes reporting any Profiling Buffer management event due to an Alignment fault or MMU fault.

Note

If profiling is not disabled when the Context synchronization event occurs, further profiling operations might be generated that are not guaranteed to be synchronized by the `PSB` instruction.
If the PE takes an exception to an Exception level where profiling is disabled, no new operations are selected for sampling. Profiling is always disabled if the owning Exception level is a lower Exception level than the current Exception level.

I_{VKDVF} In the absence of a Context synchronization event, a `PSB` instruction is not required to execute in program order with respect to sampled operations.

D17.9.1 UNPREDICTABLE behavior

R_{PVRNK} In the absence of correct Context synchronization events, it is UNPREDICTABLE whether an indirect read of a System register made by a profiling operation will return the old or the new values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{SXWCP} If the indirect reads mean that `ProfilingBufferEnabled()` returns FALSE when a sample record or records are about to be written to memory, then it is further CONSTRAINED UNPREDICTABLE whether the sample record or records:

- Are written to memory.
- Are silently discarded and not written to memory.
- Are discarded and not written to memory, and an Access Not Allowed Profiling Buffer management event is generated.

R_{VCYWL} If SCR_EL3.{NSE, NS} does not match the owning Security state, then it is CONSTRAINED UNPREDICTABLE whether the sample record or records:

- Are written to memory using an address formed using the identity of the owning translation regime.
- Are written to memory using an address formed using the Effective values of SCR_EL3.{NSE, NS}.
- Are silently discarded and not written to memory.
- Are discarded and not written to memory, and an Access Not Allowed Profiling Buffer management event is generated.

R_{QJWVQ} When FEAT_RME is implemented and MDCR_EL3.{NSPB, NSPBE} selects a reserved value, then it is CONSTRAINED UNPREDICTABLE whether the sample record or records:

- Are written to memory using an address formed using any implemented Security state.
- Are silently discarded and not written to memory.
- Are discarded and not written to memory, and an Access Not Allowed Profiling Buffer management event is generated.

I_{PDDCL} Software must execute a `PSB` instruction to force any sample records to be written to the Profiling Buffer before changing context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D18
Statistical Profiling Extension Sample Record Specification

This chapter describes the sample records generated by the Statistical Profiling Extension. It contains the following sections:

- About the Statistical Profiling Extension sample records.
- Alphabetical list of Statistical Profiling Extension packets.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.1 About the Statistical Profiling Extension sample records

The Statistical Profiling Extension sample record format version is identified by PMSIDR_EL1.Format. The architecture currently defines only version 0.

Note

Armv8.7 defines the SPE sample record format version, allowing future architecture updates to extend or change the record format. PMSIDR_EL1.Format was previously a res0 field in a read-only register. Software that reads and checks PMSIDR_EL1.Format on any implementation prior to Armv8.7 that includes SPE will read a value indicating format version 0 is supported.

The sample record format version 0 is self-describing and extensible. This format allows software to parse profile data even when that profile data contains extended information.

The Statistical Profiling Extension writes a series of sample records to memory, each record consisting of a sequence of packets, and each packet consisting of:

- One or two header bytes.
- Zero, 1, 2, 4, or 8 payload bytes.

D18.1.1 Headers

The first header byte encodes the number of payload bytes:

`0x00` - `0x1F` Single byte header, no payload.

`0x20` - `0x3F` First byte of extended header. Second byte encodes the payload length.

`0x40` - `0x4F`, `0x80` - `0x8F`, `0xC0` - `0xCF`

Header with an 8-bit payload.

`0x50` - `0x5F`, `0x90` - `0x9F`, `0xD0` - `0xDF`

Header with a 16-bit payload.

`0x60` - `0x6F`, `0xA0` - `0xAF`, `0xE0` - `0xEF`

Header with a 32-bit payload.

`0x70`-`0x7F`, `0xB0`-`0xBF`, `0xF0`-`0xFF`

Header with a 64-bit payload.

D18.1.2 Records

A record consists of multiple packets. A record comprises, in ascending address order:

- A sequence of headers, each followed by their payload byte or bytes.
- Either:
  - An End packet header.
  - A Timestamp packet.

Figures in this chapter show each packet as a sequence of bytes. Figure D18-1 shows how bytes are stored in memory in increasing addresses from left to right.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

First byte	1	2	3	4	Last Byte
Header (16-bit data)	Data		Header (8-bit data)	Data	0x01 End Packet
	LSB	MSB			

First Byte	1	2	3	4	5	6	...	12	Last Byte
Header (16-bit data)	Data		Header (8-bit data)	Data	0x71 Timestamp Packet	TS [7:0]	...	TS [55:48]	TS [63:56]
	LSB	MSB							

Figure D18-1 Convention for packet description

In some sections, the figures are split into separate figures for the header byte and payload bytes. For instance, where the number of payload bytes varies according to a field in the header.

The header bytes and payload bytes are described in ascending memory address order. Within a payload value, values are in little-endian byte order.

The size of the access granule for writes to the Profiling Buffer by the Statistical Profiling Unit is IMPLEMENTATION DEFINED, up to a maximum of 2KB. The size of the access granule can vary from time to time.

D18.1.3 Protocol framing packets and forwards compatibility

The padding header, Timestamp packet, and end packet are protocol framing packets that frame the records created by the Statistical Profiling Unit. Only padding headers are permitted between records.

Note

PMBIDR_EL1.Align defines a minimum alignment for records. However, implementations must nevertheless create a valid protocol stream that can be parsed without knowledge of the minimum alignment.

The packet types are described in the following sections. Software must ignore unknown packets, using the size field encoded in the header. This includes packets containing reserved values in fields.

The following sections give an overview of the Statistical Profiling Unit packets output to a memory-mapped Profiling Buffer or Device memory:

- Statistical Profiling Extension protocol packet headers

D18.1.4 Statistical Profiling Extension protocol packet headers

D18.1.4.1 8-bit headers

For Address packets and Counter packets, the 8-bit header format is described as the *Short format*.

Table D18-1 8-bit header encodings

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Description
0	0	0	0	0	0	0	0	Padding
0	0	0	0	0	0	0	1	End packet
0	1	1	1	0	0	0	1	Timestamp packet

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Description
0	1	x	x	0	0	1	0	Events packet
0	1	x	x	0	0	1	1	Data Source packet
0	1	1	0	0	1	x	x	Context packet
0	1	0	0	1	0	x	x	Operation Type packet
1	0	1	1	0	x	x	x	Address packet (Short format)
1	0	0	1	1	x	x	x	Counter packet (Short format)

D18.1.4.2 16-bit headers

For Address packets and Counter packets, the 16-bit header format is described as the *Extended format*.

Table D18-2 16-bit header encodings

Byte 0								Byte 1								Description
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	
0	0	1	0	0	0	x	x	1	0	1	1	0	x	x	x	Address packet
0	0	1	0	0	0	x	x	1	0	0	1	1	x	x	x	Counter packet

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2 Alphabetical list of Statistical Profiling Extension packets

This section lists every SPE packet and their description.

D18.2.1 Address packet

Purpose

Provides an address value for the record. Addresses are always 64 bits.

Attributes

Multi-part packet comprising:

- 8 or 16-bit header.
- 64-bit payload.

Address packet header

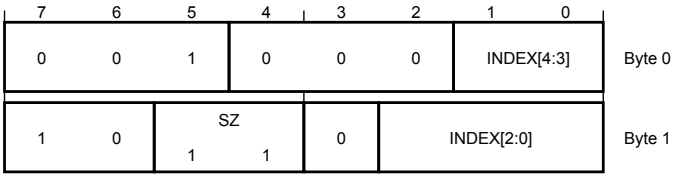

Figure D18-2 Address packet header (Extended format)

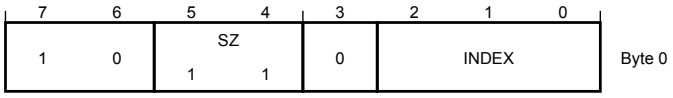

Figure D18-3 Address packet header (Short format)

Byte 1 bits [7:6], when Extended format; byte 0 bits [7:6], when Short format

This field reads as `0b10`.

SZ, byte 1 bits [5:4], when Extended format; byte 0 bits [5:4], when Short format

Payload size.

SZ	Description
`0b11`	Doubleword.

This field reads as `0b11`.

Byte 1 bit [3], when Extended format; byte 0 bit [3], when Short format

This bit reads as `0b0`.

INDEX, byte 0 bits [1:0], byte 1 bits [2:0], when Extended format; byte 0 bits [2:0], when Short format

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INDEX	Description
`0b00000`	Instruction virtual address (PC). The virtual address of the sampled operation.
`0b00001`	Branch target address. The target virtual address of a sampled branch operation.
`0b00010`	Data access virtual address. The virtual address accessed by a sampled data memory accessing operation.
`0b00011`	Data access physical address. The physical address accessed by a sampled data memory accessing operation.
`0b00100`	Previous branch target address. The target virtual address of the most recently taken branch operation in program order before the sampled operation. This value is defined only if FEAT_SPE_PBT is implemented, and is reserved otherwise.
`0b0011x`	IMPLEMENTATION DEFINED address.
`0b1xxxx`	IMPLEMENTATION DEFINED address.

All other values are reserved.

In the Short format header, INDEX[4:3] are zero.

Byte 0 bits [7:5], when Extended format

This field reads as `0b001`.

Byte 0 bits [4:2], when Extended format

This field reads-as-zero.

Address packet payload (instruction virtual address or branch target address)

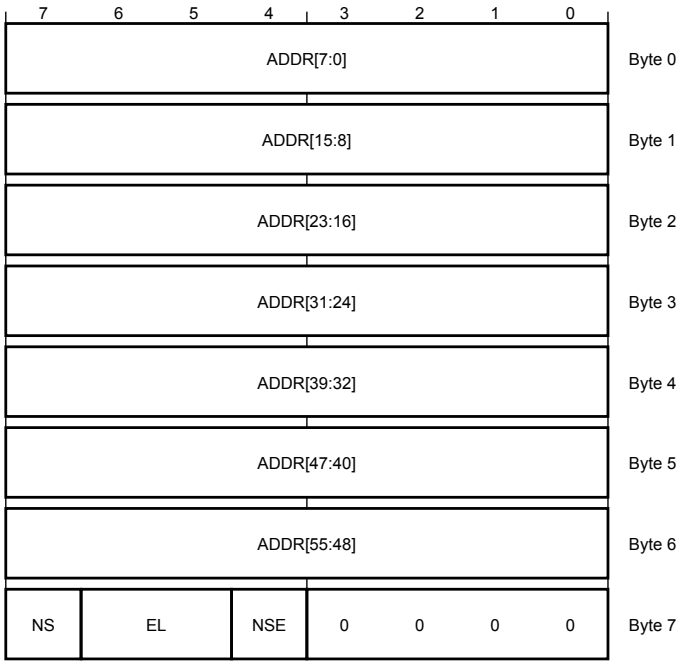

Figure D18-4 Address packet payload

NS, byte 7 bit [7]

Non-secure state.

For an instruction virtual address (PC) this is the Security state the instruction was executed in. For a branch target address, this is the Security state at the target of the branch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For an Exception Return, the Security state at the target of the branch might be different to the Security state the instruction was executed in.

When FEAT_RME is implemented

With the NSE bit, indicates the Security state associated with the address.

NSE	NS	Description
0b0	0b0	Secure state. This value is defined only if Secure state is implemented, and reserved otherwise.
0b0	0b1	Non-secure state.
0b1	0b1	Realm state.

All other values are reserved.

Note

There is no encoding for Root because records are not generated at EL3.

Otherwise

The Security state associated with the address.

NS	Description
0b0	Secure state.
0b1	Non-secure state.

EL, byte 7 bits [6:5]

Exception level. The Exception level associated with the address. For an instruction virtual address (PC) this is the Exception level the instruction was executed in. For a branch target address, this is the Exception level at the target of the branch.

EL	Description
0b00	EL0.
0b01	EL1.
0b10	EL2.
0b11	EL3.

Note

For an Exception Return, the Exception level at the target of the branch might be different to the Exception level the instruction was executed in.

NSE, byte 7 bit [4]

Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented

Together with the NS bit, indicates the Security state associated with the address.

Otherwise

This bit reads-as-zero.

Byte 7 bits [3:0]

This field reads as `0b0000`.

ADDR, bytes <6:0>

Address. Bits [55:0] of the instruction virtual address or branch target address.

Address packet payload (data access virtual address)

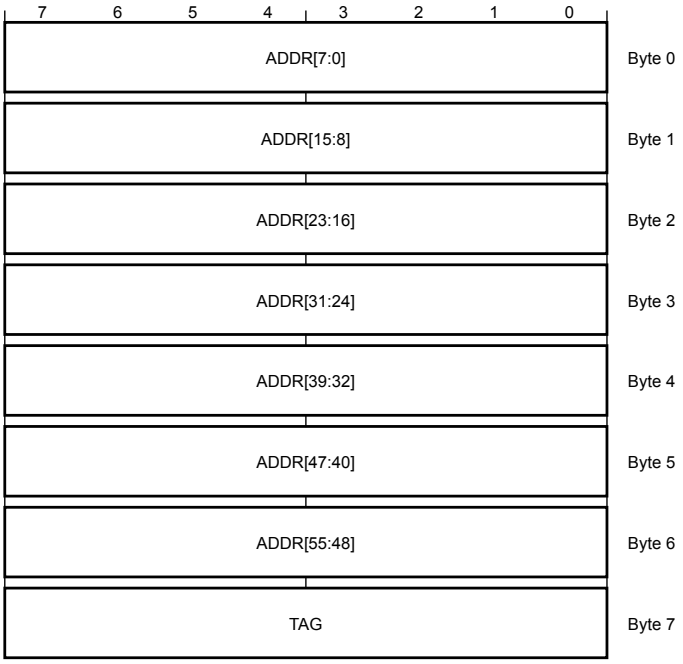

Figure D18-5 Address packet payload

TAG, byte <7>

Top-byte tag.

If the applicable TBI bit is `0b1`, then a data access virtual address includes the top-byte tag. Otherwise this field reads-as-zero.

If the applicable TBI bit is `0b0`, then it is IMPLEMENTATION DEFINED whether this field reads as zero or holds bits[63:56] of the address.

ADDR, bytes <6:0>

Address. Bits [55:0] of the data access virtual address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Address packet payload (data access physical address)

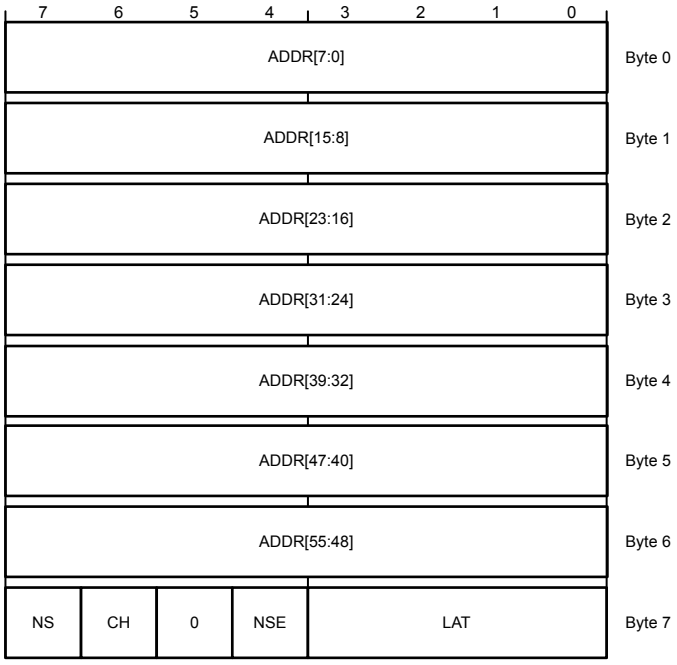

Figure D18-6 Address packet payload

NS, byte 7 bit [7]

Physical address space identifier.

When FEAT_RME is implemented

With the NSE bit, indicates the physical address space of the physical address.

NSE	NS	Description
`0b0`	`0b0`	Secure physical address space. This value is defined only if Secure PAS is implemented, and reserved otherwise.
`0b0`	`0b1`	Non-secure physical address space.
`0b1`	`0b1`	Realm physical address space.

All other values are reserved.

Note

There is no encoding for Root because records are not generated at EL3.

Otherwise

The physical address space of the physical address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Description
`0b0`	Secure physical address space.
`0b1`	Non-secure physical address space.

CH, byte 7 bit [6]

Tag Checked.

When FEAT_MTE2 is implemented

Indicates whether the access was checked against an Allocation Tag in memory.

CH	Description
`0b0`	Access not checked.
`0b1`	Access checked against an Allocation Tag in memory.

If Tag Check Faults are configured to be ignored by SCTLR_EL*x*.TCF or SCTLR_EL*x*.TCF0, then it is IMPLEMENTATION DEFINED whether this bit is `0b1` or `0b0` on a Tag Checked access.

If FEAT_MTE_CANONICAL_TAGS is implemented and a sampled data access physical address is Canonically Tagged, then the value of this bit is `0b0`.

Otherwise

This bit reads-as-zero.

Byte 7 bit [5]

This bit reads as `0b0`.

NSE, byte 7 bit [4]

Physical address space.

When FEAT_RME is implemented

Together with the NS bit, indicates the physical address space associated with the address.

Otherwise

This bit reads-as-zero.

LAT, byte 7 bits [3:0]

Logical Address Tag.

When FEAT_MTE2 is implemented

Logical Address Tag for a checked access. If the access is not checked then this field reads as an IMPLEMENTATION DEFINED choice between `0b0000` and the Logical Address Tag used to perform the access.

Otherwise

This field reads-as-zero.

ADDR, bytes <6:0>

Address. Bits [55:0] of the data access physical address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.2 Context packet

Purpose

Provides context information for the record.

Attributes

Multi-part packet comprising:

- 8-bit header.
- 32-bit payload.

Context packet header

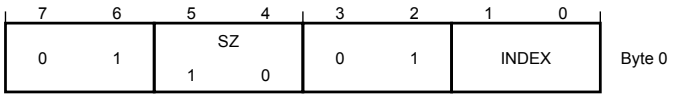

Figure D18-7 Context packet header

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size.

SZ	Description
`0b10`	Word.

This field reads as `0b10`.

Byte 0 bits [3:2]

This field reads as `0b01`.

INDEX, byte 0 bits [1:0]

Identifies the context value.

INDEX	Description
`0b00`	CONTEXTIDR_EL1. Included for all operations if enabled by `CollectContextIDR1`.
`0b01`	CONTEXTIDR_EL2. Included for all operations if enabled by `CollectContextIDR2`.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Context packet payload

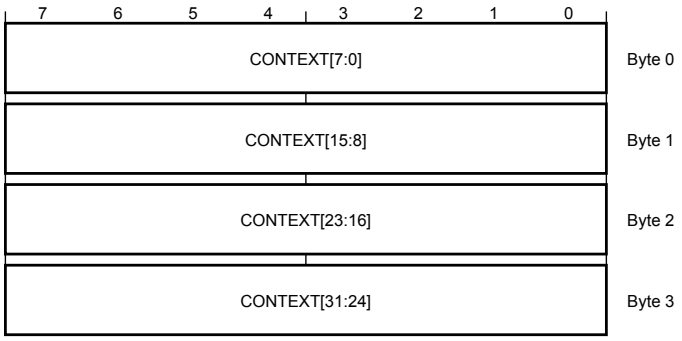

Figure D18-8 Context packet payload

CONTEXT, bytes <3:0>

The context value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.3 Counter packet

Purpose

Count of cycles the operation spent performing all or part of its behavior. The counter value occupies the least significant bits of the payload. The remaining bits are set to zero.

Attributes

Multi-part packet comprising:

- 8 or 16-bit header.
- 16-bit payload.

Counter packet header

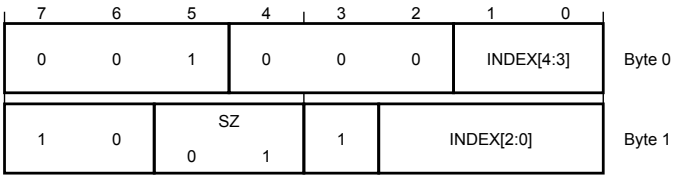

Figure D18-9 Counter packet header (Extended format)

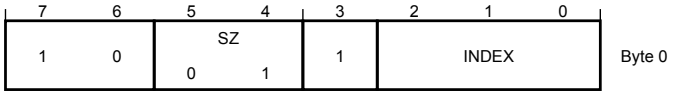

Figure D18-10 Counter packet header (Short format)

Byte 1 bits [7:6], when Extended format; byte 0 bits [7:6], when Short format

This field reads as `0b10`.

SZ, byte 1 bits [5:4], when Extended format; byte 0 bits [5:4], when Short format

Payload size.

SZ	Description
`0b01`	Halfword.

This field reads as `0b01`.

Byte 1 bit [3], when Extended format; byte 0 bit [3], when Short format

This bit reads as `0b1`.

INDEX, byte 0 bits [1:0], byte 1 bits [2:0], when Extended format; byte 0 bits [2:0], when Short format

INDEX	Description
`0b00000`	Total latency. Cycle count from the operation being dispatched for issue to the operation being complete. Included for all operations.
`0b00001`	Issue latency. The count of cycles when the operation is waiting to be issued.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INDEX	Description
`0b00010`	Translation latency. The count of cycles when at least one part of the operation is waiting for the MMU to complete an address translation. It is IMPLEMENTATION DEFINED whether a cycle is counted if a part of the operation is accessing memory, having completed an address translation for that part of the operation. For more information, see Additional information for each profiled memory access operation.
`0b00100`	Alternate clock issue latency. As *issue latency*, but counts cycles in the clock domain specified by PMSIDR_EL1.ALTCLK.
`0b0011x`	IMPLEMENTATION DEFINED counter value.
`0b1xxxx`	IMPLEMENTATION DEFINED counter value.

All other values are reserved.

In the Short format header, INDEX[4:3] are zero.

Dispatched for issue means:

- The operation has been decoded.
- The operation might not be ready to start execution because it is waiting for input values. The operation might be put into a queue.

Issued for execution means the operation is ready to start executing:

- For example, for a memory operation, this should be indicative of the cycle count from memory operation being dispatched for issue to access being initiated (virtual address).

Complete means:

- The operation has completed execution and is no longer capable of stalling any instruction that consumes its output.
- It is IMPLEMENTATION DEFINED whether the operation has committed its results to the architectural state of the PE.
- For example:
  - For an arithmetic, FP or SIMD operation with variable timing, such as divide the results of the operation are available.
  - For load and atomic operations that return data, all data have been returned from memory.
  - For store and atomic operations that do not return data, it is not required that the store has reached its end point for it to be complete.
  - For branch operations, the branch has been resolved as taken or not taken.
  - For barrier operations, the barrier has completed.

For `WFE` and `WFI` operations, it is IMPLEMENTATION DEFINED whether:

- The instruction is complete before the PE enters a low-power state or when the PE wakes from the low-power state.
- Counters count in the low power state.
- Sampling an operation is itself a wakeup event.

Byte 0 bits [7:5], when Extended format

This field reads as `0b001`.

Byte 0 bits [4:2], when Extended format

This field reads-as-zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Counter packet payload

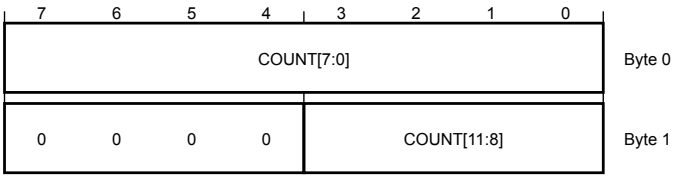

Figure D18-11 Counter packet payload (12-bit counters are implemented)

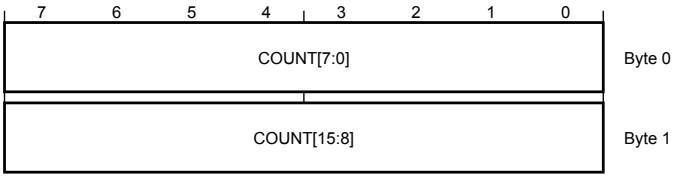

Figure D18-12 Counter packet payload (16-bit counters are implemented)

Byte 1 bits [7:4], when 12-bit counters are implemented

This field reads as `0b0000`.

COUNT, byte 1 bits [3:0], byte <0>, when 12-bit counters are implemented; bytes <1:0>, when 16-bit counters are implemented

The counter value occupies the least significant bits of the payload. The remaining bits are set to zero. The counters are:

- Unsigned numbers.
- 12 or 16 bits.
- Saturating.

PMSIDR_EL1.CountSize indicates the size of counter implemented, and:

- If 12-bit counters are implemented, the value `0xFFF` indicates the count has saturated.
- If 16-bit counters are implemented, the value `0xFFFF` indicates the count has saturated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.4 Data Source packet

Purpose

If the implementation includes support for indicating the loaded data source, the Data Source packet indicates where the data returned for a load operation was sourced. It might also include other information, such as the state of the data at the source. It is IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether this is included for load and atomic operations that generate an External Abort. It is IMPLEMENTATION DEFINED whether this is included for atomic operations that do not return data to a PE register. Included for all other load and atomic operations.

Attributes

Multi-part packet comprising:

- 8-bit header.
- 8 or 16-bit payload.

Data Source packet header

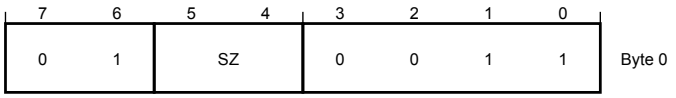

Figure D18-13 Data Source packet header

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size.

SZ	Description
`0b00`	Byte.
`0b01`	Halfword.

Byte 0 bits [3:0]

This field reads as `0b0011`.

Data Source packet payload

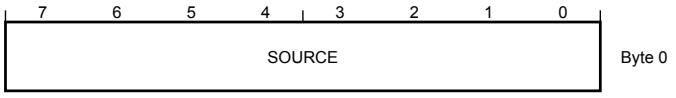

Figure D18-14 Data Source packet payload (SZ == 0b00)

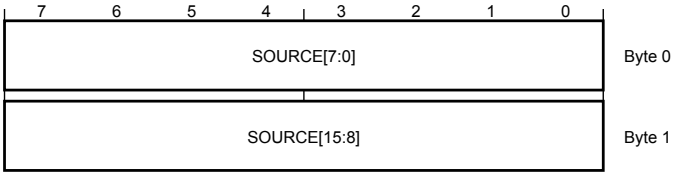

Figure D18-15 Data Source packet payload (SZ == 0b01)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SOURCE, byte <0>, when SZ == `0b00`; bytes <1:0>, when SZ == `0b01`

Because the list of data sources varies from system to system, the definition of this field is IMPLEMENTATION DEFINED. If a sampled operation generated multiple data accesses, it is IMPLEMENTATION DEFINED how the data source information is combined.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.5 End packet

Purpose

Defines the end of a record if a Timestamp packet is not present.

Attributes

8-bit packet.

Field descriptions

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	Byte 0

Figure D18-16 End packet

Byte <0>

This field reads as `0b00000001`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.6 Events packet

Purpose

Indicates events generated by the sampled operation. If the sampled operation generates one of the specified events, then the corresponding field is set to `0b1`. Otherwise, the corresponding field is set to `0b0`. Where applicable, a corresponding PMU event is defined for each event in the packet.

Attributes

Multi-part packet comprising:

- 8-bit header.
- 8, 16, 32, or 64-bit payload.

Events packet header

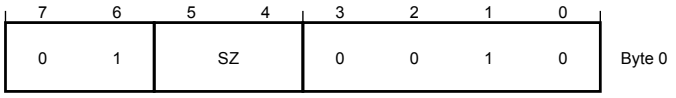

Figure D18-17 Events packet header

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size.

SZ	Description
`0b00`	Byte.
`0b01`	Halfword.
`0b10`	Word.
`0b11`	Doubleword.

Software must treat bits that are not output as zero.

Byte 0 bits [3:0]

This field reads as `0b0010`.

Events packet payload

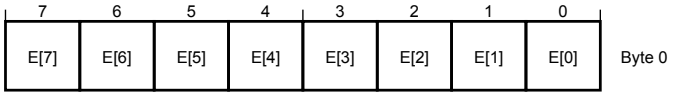

Figure D18-18 Events packet payload (SZ == 0b00)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure D18-19 Events packet payload (SZ == 0b01)

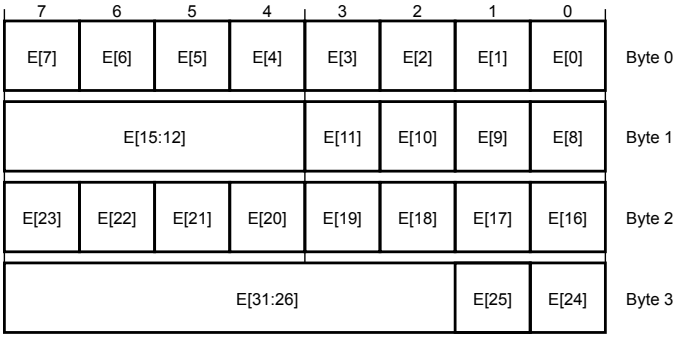

Figure D18-20 Events packet payload (SZ == 0b10)

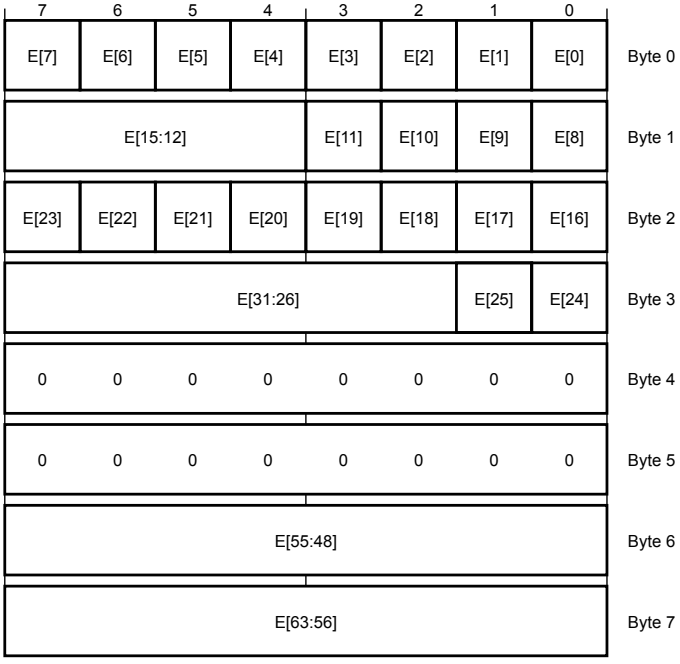

Figure D18-21 Events packet payload (SZ == 0b11)

E[63:48], bytes <7:6>, when SZ == `0b11`

Events 63 to 48. IMPLEMENTATION DEFINED.

Bytes <5:4>, when SZ == `0b11`

This field reads-as-zero.

E[31:26], byte 3 bits [7:1], when SZ == `0b10` or SZ == `0b11`

Events 31 to 26.

When FEAT_SPEv1p4 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field reads-as-zero.

Otherwise

This field reads as an IMPLEMENTATION DEFINED value.

E[25], byte 3 bit [1], when SZ == `0b10` or SZ == `0b11`

Streaming Mode Compute Unit (SMCU) or other shared resource operation.

When (FEAT_SPE_SME or FEAT_SPEv1p5 is implemented), and an SMCU or other shared resource is implemented

E[25]	Description
`0b0`	Operation did not execute on an SMCU or other shared resource.
`0b1`	Operation executed on an SMCU or other shared resource.

A shared resource means a resource that is shared between PEs for the execution of instructions. It is IMPLEMENTATION DEFINED which instructions can be executed on a shared resource.

When FEAT_SPE_SME is implemented or FEAT_SPEv1p4 is implemented

This bit reads-as-zero.

Otherwise

This bit reads as an IMPLEMENTATION DEFINED value.

E[24], byte 3 bit [0], when SZ == `0b10` or SZ == `0b11`

Streaming SVE mode.

When FEAT_SPE_SME is implemented

E[24]	Description
`0b0`	Operation executed when PE is in Non-streaming SVE mode.
`0b1`	Operation executed when PE is in Streaming SVE mode.

When FEAT_SPEv1p4 is implemented

This bit reads-as-zero.

Otherwise

This bit reads as an IMPLEMENTATION DEFINED value.

E[23], byte 2 bit [7], when SZ == `0b10` or SZ == `0b11`

Data snooped.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

When FEAT_SPEv1p4 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[23]	Description
0b0	Operation was not a load, or did not access a cache outside the cache hierarchy of this PE.
0b1	Load operation that snooped data from a cache outside the cache hierarchy of this PE.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with DSNP_HIT_RD.

Otherwise

This bit reads-as-zero.

E[22], byte 2 bit [6], when SZ == 0b10 or SZ == 0b11

Recently fetched.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

When FEAT_SPEv1p4 is implemented

E[22]	Description
0b0	Operation was not a load, or did not hit a recently-fetched cache line.
0b1	Load operation hit a recently-fetched line in a cache.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L1D_LFB_HIT_RD, L2D_LFB_HIT_RD, and LL_LFB_HIT_RD.

Otherwise

This bit reads-as-zero.

E[21], byte 2 bit [5], when SZ == 0b10 or SZ == 0b11

Cache data modified.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

When FEAT_SPEv1p4 is implemented

E[21]	Description
0b0	Operation was not a load, or did not access modified data.
0b1	Load operation accessed modified data in a cache.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with DSNP_HITM_RD, L1D_CACHE_HITM_RD, L2D_CACHE_HITM_RD, and LL_CACHE_HITM_RD.

It is IMPLEMENTATION DEFINED whether this bit is valid for a cache hit (L<*n*>D_CACHE_HITM_RD), a snoop hit (DSNP_HITM_RD), or both.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise

This bit reads-as-zero.

E[20], byte 2 bit [4], when SZ == `0b10` or SZ == `0b11`

Level 2 data cache miss.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

When FEAT_SPEv1p4 is implemented

E[20]	Description
`0b0`	Operation was not a load, or did not access level 2 data cache.
`0b1`	Load operation accessed and missed the level 2 data or unified cache. This excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L2D_CACHE_LMISS_RD.

Otherwise

This bit reads-as-zero.

E[19], byte 2 bit [3], when SZ == `0b10` or SZ == `0b11`

Level 2 data cache access.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

When FEAT_SPEv1p4 is implemented

E[19]	Description
`0b0`	Operation was not a load, or did not access level 2 data cache.
`0b1`	Load operation caused a cache access to at least the level 2 data or unified cache.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L2D_CACHE_RD.

Otherwise

This bit reads-as-zero.

E[18], byte 2 bit [2], when SZ == `0b10` or SZ == `0b11`

Empty predicate.

When FEAT_SPEv1p1 is implemented and (FEAT_SVE is implemented or FEAT_SME is implemented)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[18]	Description
`0b0`	Operation was neither an SVE nor an SME operation, was not predicated, or at least one element in the Governing predicate or predicates was TRUE.
`0b1`	SVE or SME operation and all elements in the Governing predicate or predicates were FALSE.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with SME_PRED2_EMPTY_SPEC and SVE_PRED_EMPTY_SPEC.

For a sampled SVE operation, it is IMPLEMENTATION DEFINED whether this field is valid or reads as zero when the PRED field in the Operation Type field is 0.

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

Otherwise

This bit reads-as-zero.

E[17], byte 2 bit [1], when SZ == `0b10` or SZ == `0b11`

Partial or empty predicate.

When FEAT_SPEv1p1 is implemented and (FEAT_SVE is implemented or FEAT_SME is implemented)

E[17]	Description
`0b0`	Operation was neither an SVE nor an SME operation, was not predicated, or all elements in the Governing predicate or predicates were TRUE.
`0b1`	SVE or SME operation and at least one element in the Governing predicate or predicates was FALSE.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with SME_PRED2_NOT_FULL_SPEC and SVE_PRED_NOT_FULL_SPEC.

For a sampled SVE operation, it is IMPLEMENTATION DEFINED whether this field is valid or reads as zero when the PRED field in the Operation Type field is 0.

Note

For outer product instructions which are widening, predication is considered with respect to the input element size.

Otherwise

This bit reads-as-zero.

E[16], byte 2 bit [0], when SZ == `0b10` or SZ == `0b11`

Transactional.

This bit reads-as-zero.

E[15:12], byte 1 bits [7:4], when SZ == `0b01`, SZ == `0b10`, or SZ == `0b11`

Events 15 to 12. IMPLEMENTATION DEFINED.

E[11], byte 1 bit [3], when SZ == `0b01`, SZ == `0b10`, or SZ == `0b11`

Misalignment.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPEv1p1 is implemented

E[11]	Description
`0b0`	Operation was not a load/store, or a load/store operation that was optimally aligned for the size of data being accessed.
`0b1`	Load/store operation that, due to the alignment of the address and size of data being accessed, incurred additional latency.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with LDST_ALIGN_LAT.

Otherwise

This bit reads-as-zero.

E[10], byte 1 bit [2], when SZ == `0b01`, SZ == `0b10`, or SZ == `0b11`

Remote access.

E[10]	Description
`0b0`	Operation was not a load/store, or did not cause access to another socket.
`0b1`	Load/store operation caused an access to another socket in a multi-socket system. This includes each data memory access that accesses another socket in a multi-socket system, including those that do not return data.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

If this event and FEAT_PMUv3 are implemented, then this event is implemented consistently with REMOTE_ACCESS or REMOTE_ACCESS_RD.

E[9], byte 1 bit [1], when SZ == `0b01`, SZ == `0b10`, or SZ == `0b11`

Last Level cache miss.

E[9]	Description
`0b0`	Operation was not a load/store, or did not miss Last Level cache.
`0b1`	Load/store operation caused an access to at least the Last Level cache but is not completed by the Last Level cache. That is, each: • Load operation that does not return data from the Last Level cache. • Store operation that does not update the Last Level cache. The event is not set for operations that are completed by a cache above the Last Level cache.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

If this event and FEAT_PMUv3 are implemented, then this event is implemented consistently with LL_CACHE_MISS or LL_CACHE_MISS_RD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[8], byte 1 bit [0], when SZ == `0b01`, SZ == `0b10`, or SZ == `0b11`

Last Level cache access.

E[8]	Description
`0b0`	Operation was not a load/store, or did not access Last Level data or unified cache.
`0b1`	Load/store operation caused a cache access to at least the Last Level data or unified cache.

This event is optional. If this event is implemented, then it is further IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero. If this event is not implemented, then this bit reads-as-zero.

If this event and FEAT_PMUv3 are implemented, then this event is implemented consistently with LL_CACHE or LL_CACHE_RD.

Note

The architecture does not define the Last Level cache. The Last Level cache is typically the largest cache on this device shared by all PEs in the inner or outer Shareable domain of this PE. In a multi-socket system, it is IMPLEMENTATION DEFINED whether this includes caches on other sockets.

E[7], byte 0 bit [7]

Mispredicted.

E[7]	Description
`0b0`	Operation was not a branch, or did not cause correction to the predicted program flow.
`0b1`	A branch that caused a correction to the predicted program flow.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with BR_MIS_PRED or BR_MIS_PRED_RETIRED.

E[6], byte 0 bit [6]

Not taken.

E[6]	Description
`0b0`	Operation was not a conditional operation, or did not fail its condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[6]	Description
0b1	A conditional instruction that failed its condition code check. This includes conditional branches, compare-and-branch, conditional select, and conditional compares: • For a conditional branch or compare-and-branch instruction, this means the branch was not taken. • For a conditional select, this means the second operand was written to the result. • For a condition compare, this means the condition flags were set to the immediate value and not the result of the compare. • If the implementation samples micro-operations, the condition result is the result condition of the micro-op. This might not match the condition that was specified in the instruction as the mapping from instructions to operations is IMPLEMENTATION DEFINED. This means that software should be careful interpreting this field, and, without a prior knowledge of this mapping, treat the value as UNKNOWN. • If the implementation samples architectural instructions, then note that for some alias instructions, the operation specifies the opposite condition to the machine instruction that they alias. For example, CSET and CSETM. The event as captured by SPE always captures the result of the underlying instruction, not the alias.

This field is valid only if the OpType defines this as either a Conditional Branch or a Conditional operation, and is RAZ otherwise.

E[5], byte 0 bit [5]

TLB walk.

E[5]	Description
0b0	Operation was not a load/store, or did not generate TLB walk.
0b1	Load/store operation that causes a refill of a data or unified TLB, involving at least one translation table walk access. This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with DTLB_WALK.

E[4], byte 0 bit [4]

TLB access.

E[4]	Description
0b0	Operation was not a load/store that accessed the TLB.
0b1	Load/store operation caused an access to at least the first level of data or unified TLB.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L1D_TLB.

E[3], byte 0 bit [3]

Level 1 data cache refill or miss.

When FEAT_SPEv1p4 is implemented

E[3]	Description
0b0	Operation was not a load/store, or did not miss in the level 1 data cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[3]	Description
0b1	Load/store operation accessed and missed the first level of data or unified cache. This excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

It is IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L1D_CACHE_MISS or L1D_CACHE_MISS_RD.

Otherwise

E[3]	Description
0b0	Operation was not a load/store, or did not cause a level 1 data cache refill.
0b1	Load/store operation caused a refill of at least the first level of data or unified cache. This includes each data memory access that causes a refill from outside the cache. It excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

It is IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L1D_CACHE_REFILL or L1D_CACHE_REFILL_RD.

E[2], byte 0 bit [2]

Level 1 data cache access.

E[2]	Description
0b0	Operation was not a load/store, or did not access the level 1 data cache.
0b1	Load/store operation caused a cache access to at least the first level of data or unified cache.

It is IMPLEMENTATION DEFINED and might be CONSTRAINED UNPREDICTABLE whether a store can finish execution before this event is generated, meaning that, for stores, this event is not recorded and this bit reads-as-zero.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with L1D_CACHE or L1D_CACHE_RD.

E[1], byte 0 bit [1]

Architecturally retired.

When the PE supports sampling of speculative instructions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[1]	Description
`0b0`	Did not retire.
`0b1`	Committed its results to the architectural state of the PE, or completed with a synchronous architectural exception.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with INST_RETIRED.

Note

A conditional instruction can retire even if it fails its condition code check.

Otherwise

This bit reads-as-one.

E[0], byte 0 bit [0]

Generated exception.

E[0]	Description
`0b0`	Did not generate an exception.
`0b1`	Completed with a synchronous exception.

If FEAT_PMUv3 is implemented, then this event is implemented consistently with EXC_TAKEN.

If E[1] in the same Events packet is 0, then the meaning of this bit is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.7 Operation Type packet

Purpose

Defines the type of operation sampled. Included for all operations.

Attributes

Multi-part packet comprising:

- 8-bit header.
- 8-bit payload.

Operation Type packet header

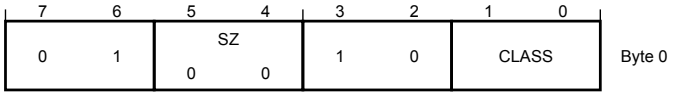

Figure D18-22 Operation Type packet header

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size.

SZ	Description
`0b00`	Byte.

This field reads as `0b00`.

Byte 0 bits [3:2]

This field reads as `0b10`.

CLASS, byte 0 bits [1:0]

Top-level operation class.

CLASS	Description
`0b00`	The payload is one of the following and the SUBCLASS field in the payload encodes which: • Other. • Data processing, SVE vector. • Data processing, SME array.
`0b01`	Load, store, or atomic. The payload is one of the following and the SUBCLASS field in the payload encodes which: • Load/store, general. • Load/store, extended. • Load/store, SVE or SME. • Load/store, Memory Copy. • Load/store, Memory Set. • Load/store, GCS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLASS	Description
`0b10`	Branch or exception return.

All other values are reserved.

Operation Type packet payload (Other)

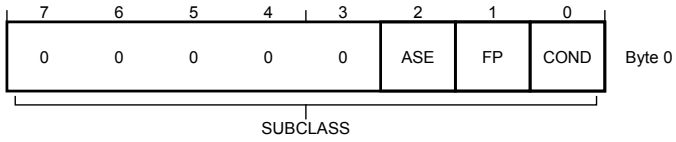

Figure D18-23 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Describes the operation type.

SUBCLASS	Description
`0b00000xxx`	Other operation. Bits [2:0] are further defined as the ASE, FP, and COND fields.

Other values are either defined by other payload types for CLASS==`0b00` or reserved.

This field reads as `0b00000xxx`.

ASE, byte 0 bit [2]

Advanced SIMD.

When FEAT_SPE_FPF is implemented

ASE	Description
`0b0`	Not an Advanced SIMD operation.
`0b1`	Advanced SIMD operation.

Otherwise

This bit reads-as-zero.

FP, byte 0 bit [1]

Floating-point.

When FEAT_SPE_FPF is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FP	Description
0b0	Not a floating-point operation.
0b1	Floating-point operation.

Otherwise

This bit reads-as-zero.

COND, byte 0 bit [0]

Conditional.

COND	Description
0b0	Unconditional operation.
0b1	Conditional select or conditional compare operation.

Operation Type packet payload (Data processing, SVE vector)

Configurations

Defined only if all of the following are true:

- FEAT_SPEv1p1 is implemented.
- FEAT_SVE is implemented.

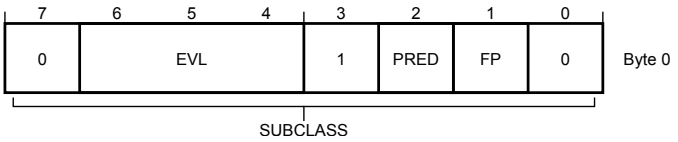

Figure D18-24 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Describes the operation type.

SUBCLASS	Description
0b0xxx1xx0	SVE vector operation. Bits [6:4,2:1] are further defined as the EVL, PRED, and FP fields.

Other values are either defined by other payload types for CLASS==0b00 or reserved.

This field reads as 0b0xxx1xx0.

EVL, byte 0 bits [6:4]

Sampled Effective Vector Length.

Describes the size in bits of the vector or vectors operated on by the sampled SVE SIMD operation. That is:

- For a sampled SVE multi-vector SIMD operation, the number of elements in each vector, multiplied by the number of vectors, multiplied by the element size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For other sampled SVE SIMD operations, the number of elements in the vector multiplied by the element size.

EVL	Description
0b000	32 bits.
0b001	64 bits.
0b010	128 bits.
0b011	256 bits.
0b100	512 bits.
0b101	1024 bits.
0b110	2048 bits.
0b111	More than 2048 bits. This value is defined only if FEAT_SPE_SME is implemented, and reserved otherwise.

If the sampled effective vector length for the input operands of the sampled SVE SIMD operation and the sampled effective vector length for the output operand of the sampled SVE SIMD operation differ, then the sampled effective vector length of the output operand is used.

The Effective SVE vector length is always a power of two. However, the sampled effective vector length can be any size down to the smallest element size.

If the sampled effective vector length is not a power of two, or is less than 32 bits, then the value is rounded up before it is encoded in this field.

PRED, byte 0 bit [2]

Predicated SVE operation.

PRED	Description
0b0	Not predicated.
0b1	Predicated SVE data-processing operation. This includes operations which result in no effective change in architectural state due to all predicates being FALSE.

Predicated SVE data-processing operation means an SVE data-processing operation that writes to one or more SVE Z vector destination registers under a Governing predicate using either zeroing or merging predication.

It is IMPLEMENTATION DEFINED whether this includes data-processing operations due to instructions with a single Governing predicate operand that determines the Active elements, which do not write to any SVE Z vector destination registers using either zeroing or merging predication. For example, INCP and SEL (predicates).

FP, byte 0 bit [1]

Floating-point operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FP	Description
0b0	Not a floating-point operation.
0b1	Floating-point operation.

Operation Type packet payload (Data processing, SME array)

Configurations

Defined only if FEAT_SPE_SME is implemented.

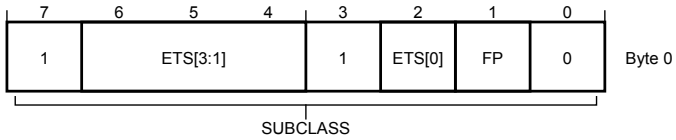

Figure D18-25 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Describes the operation type.

SUBCLASS	Description
0b1xxx1xx0	SME tile operation. Bits [6:4,2:1] are further defined as the ETS and FP fields. This value is defined only if FEAT_SPE_SME is implemented, and reserved otherwise.

Other values are either defined by other payload types for CLASS==0b00 or reserved.

This field reads as 0b1xxx1xx0.

ETS, byte 0 bits [6:4,2]

Sampled Effective vector length or Tile Size.

Describes the size in bits of the portion of the ZA array operated on by the sampled SME vector or tile SIMD operation. That is:

- For a sampled SME tile SIMD operation, the number of elements in the ZA tile or tiles multiplied by the element size.
- For a sampled SME multi-vector SIMD operation, the number of elements in each vector, multiplied by the number of vectors, multiplied by the element size.
- For other sampled SME vector SIMD operations, the number of elements in the vector multiplied by the element size.

ETS	Description
0b0000	128 bits.
0b0001	256 bits.
0b0010	512 bits.
0b0011	1024 bits.
0b0100	2048 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ETS	Description
`0b0101`	4096 bits.
`0b0110`	8192 bits.
`0b0111`	16384 bits.
`0b1000`	32768 bits.
`0b1001`	65536 bits.
`0b1010`	131072 bits.
`0b1011`	262144 bits.
`0b1111`	Whole ZA array.

All other values are reserved.

If the sampled effective vector length or tile size for the input operands of the sampled SME SIMD operation and the sampled effective vector length or tile size for the output operand of the sampled SME SIMD operation differ, then the sampled effective vector length or tile size of the output operand is used.

If the sampled effective vector length or tile size is not a power of two, or is less than 128 bits, then the value is rounded up before it is encoded in this field.

The `0b1111` value is used when the whole ZA array is operated on at once by the sampled instruction or operation. For example the `ZERO` (tiles) instruction writes to the whole ZA array, and might use this encoding even if the instruction specifies a mask that prevents part of the ZA array being updated.

FP, byte 0 bit [1]

Floating-point operation.

FP	Description
`0b0`	Not a floating-point operation.
`0b1`	Floating-point operation.

Operation Type packet payload (Load/store, general)

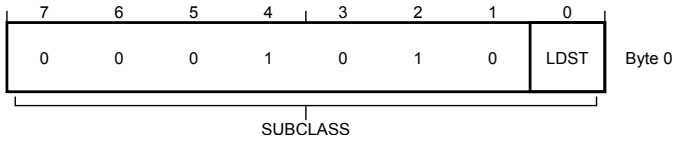

Figure D18-26 Operation Type packet payload (Allocation Tag load/store)

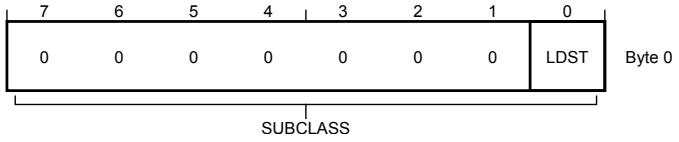

Figure D18-27 Operation Type packet payload (General-purpose load/store)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

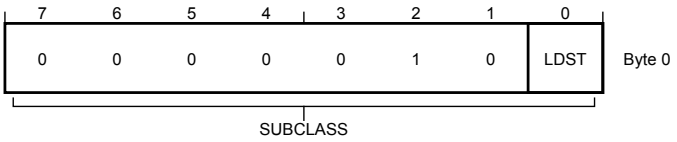

Figure D18-28 Operation Type packet payload (SIMD&FP load/store)

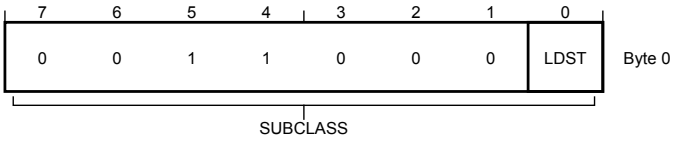

Figure D18-29 Operation Type packet payload (System register access transformed to a load/store)

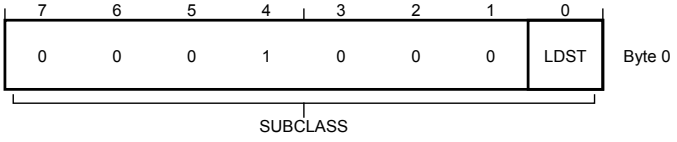

Figure D18-30 Operation Type packet payload (Unspecified load/store)

SUBCLASS, byte <0>

Second-level operation class. Indicates the load/store type.

SUBCLASS	Description
`0b0000000x`	A load/store targeting the general-purpose registers, other than an atomic operation, load-acquire, store-release, or exclusive.
`0b0000010x`	A load/store targeting the SIMD&FP registers, or, if FEAT_SPEv1p1 is not implemented, the SVE registers.
`0b0001000x`	A load/store targeting unspecified registers. This value is defined only if FEAT_SPEv1p1 is implemented, and reserved otherwise.
`0b0001010x`	A load/store of an Allocation Tag or multiple Allocation Tags. This value is defined only if all of the following are true: • FEAT_SPEv1p3 is implemented. • FEAT_MTE is implemented. This value is reserved otherwise.
`0b0011000x`	An `MRS` or `MSR` operation at EL1 transformed to a load/store when the Effective value of HCR_EL2.NV2 is 1. This value is defined only if FEAT_NV2 is implemented, and reserved otherwise.

Other values are either defined by other payload types for CLASS==`0b01` or reserved.

Bit [0] is further defined as the LDST field.

LDST, byte 0 bit [0]

Store not load.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LDST	Description
0b0	Memory-reading instruction. That is, a load, a swap, or an atomic that returns a value to the PE.
0b1	Not a memory-reading instruction. That is, a store or an atomic that does not return a value to the PE.

Memory-reading instruction and Memory-writing instruction are defined in Definitions.

Operation Type packet payload (Load/store, extended)

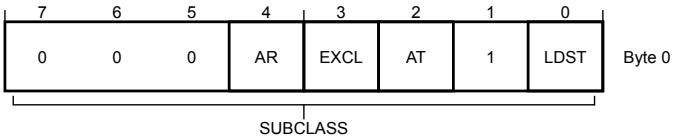

Figure D18-31 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Indicates the load/store type.

SUBCLASS	Description
0b000xxx1x	An atomic operation, load-acquire, store-release, or exclusive. Bits [4:2,0] are further defined as the AT, EXCL, AR, and LDST fields.

Other values are either defined by other payload types for CLASS==0b01 or reserved.

This field reads as 0b000xxx1x.

AR, byte 0 bit [4]

Acquire/Release.

AR	Description
0b0	Load/store/atomic without Acquire or Release semantics.
0b1	Load/store/atomic with Acquire or Release semantics.

EXCL, byte 0 bit [3]

Exclusive.

EXCL	Description
0b0	Load/store/atomic without Exclusive.
0b1	Load/store with Exclusive.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AT, byte 0 bit [2]

Atomic load/store.

AT	Description
`0b0`	Not atomic.
`0b1`	Atomic.

LDST, byte 0 bit [0]

Store not load.

LDST	Description
`0b0`	Memory-reading instruction. That is, a load, a swap, or an atomic that returns a value to the PE.
`0b1`	Not a memory-reading instruction. That is, a store or an atomic that does not return a value to the PE.

Memory-reading instruction and Memory-writing instruction are defined in Definitions.

Operation Type packet payload (Load/store, SVE or SME)

Configurations

Defined only if all of the following are true:

- Any of the following are true:
  - FEAT_SME is implemented.
  - FEAT_SVE is implemented.
- FEAT_SPEv1p1 is implemented.

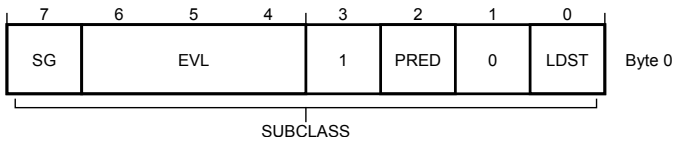

Figure D18-32 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Indicates the load/store type.

SUBCLASS	Description
`0bxxxx1x0x`	A load/store targeting the SVE or SME registers. Bits [7:4,2,0] are further defined as the PRED, EVL, SG, and LDST fields.

Other values are either defined by other payload types for CLASS==`0b01` or reserved.

This field reads as `0bxxxx1x0x`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SG, byte 0 bit [7]

Gather/scatter SVE load/store.

SG	Description
0b0	Neither gather load nor scatter store.
0b1	Gather load or scatter store.

EVL, byte 0 bits [6:4]

Sampled Effective Vector Length.

Describes the size in bits of the vector or vectors operated on by the sampled SVE or SME SIMD load/store operation.

If FEAT_SPE_SME is implemented, then the EVL field records the upper bound of the sampled effective vector length multiplied by the number of vectors loaded or stored by the sampled operation. If the sampled effective vector length for the sampled operation is greater than 2048 bits, then EVL is set to 0b111.

Otherwise, the EVL field records an upper bound of the sampled effective vector length for a single vector loaded or stored by the sampled operation.

EVL	Description
0b000	32 bits.
0b001	64 bits.
0b010	128 bits.
0b011	256 bits.
0b100	512 bits.
0b101	1024 bits.
0b110	2048 bits.
0b111	More than 2048 bits. This value is defined only if FEAT_SPE_SME is implemented, and reserved otherwise.

All other values are reserved.

The Effective SVE vector length is always a power of two. However, the sampled effective vector length can be any size down to the smallest element size.

If the sampled effective vector length is not a power of two, or is less than 32 bits, then the value is rounded up before it is encoded in this field.

PRED, byte 0 bit [2]

Predicated SVE or SME load/store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PRED	Description
0b0	Not predicated.
0b1	The operation is one of the following: • If FEAT_SPEv1p2 is implemented, a predicated SVE or SME load operation that writes to one or more vector destination registers under a Governing predicate using zeroing predication. • A predicated SVE or SME store of one or more vector registers. This includes operations which result in no effective change in architectural state due to all predicates being FALSE.

LDST, byte 0 bit [0]

Store not load.

LDST	Description
0b0	Load.
0b1	Store.

Operation Type packet payload (Load/store, Memory Copy)

Configurations

Defined only if all of the following are true:

- FEAT_SPEv1p3 is implemented.
- FEAT_MOPS is implemented.

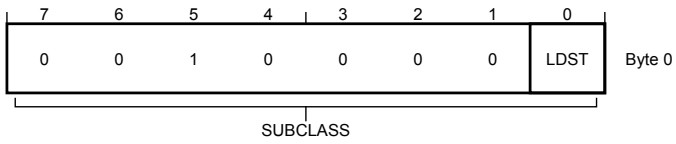

Figure D18-33 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Indicates the load/store type.

SUBCLASS	Description
0b0010000x	A load/store from a Memory Copy operation. Bit [0] is further defined as the LDST field.

Other values are either defined by other payload types for CLASS==0b01 or reserved.

This field reads as 0b0010000x.

LDST, byte 0 bit [0]

Store not load.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LDST	Description
`0b0`	Load. The sampled virtual address is the source address.
`0b1`	Store. The sampled virtual address is the destination address.

Operation Type packet payload (Load/store, Memory Set)

Configurations

Defined only if all of the following are true:

- FEAT_SPEv1p3 is implemented.
- FEAT_MOPS is implemented.

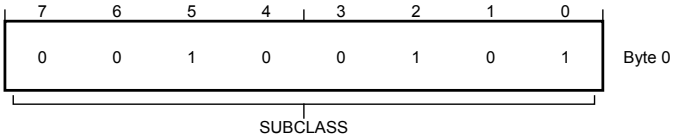

Figure D18-34 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Indicates the load/store type.

SUBCLASS	Description
`0b00100101`	A store from a Memory Set operation.

Other values are either defined by other payload types for CLASS==`0b01` or reserved.

This field reads as `0b00100101`.

Operation Type packet payload (Load/store, GCS)

Configurations

Defined only if all of the following are true:

- FEAT_GCS is implemented.
- FEAT_SPEv1p4 is implemented.

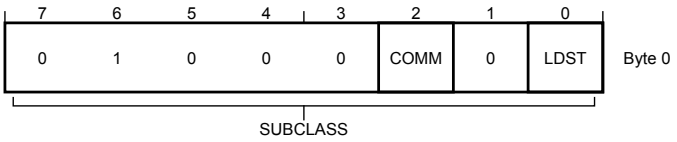

Figure D18-35 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Indicates the load/store type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SUBCLASS	Description
`0b01000x0x`	A load/store from a GCS operation. Bits [2,0] are further defined as the COMM and LDST fields.

Other values are either defined by other payload types for CLASS==`0b01` or reserved.

This field reads as `0b01000x0x`.

COMM, byte 0 bit [2]

Common. Indicates whether or not the sampled load/store operation is from a Branch with link or Procedure return instruction.

COMM	Description
`0b0`	Branch with link or Procedure return.
`0b1`	Neither a Branch with link nor a Procedure return.

LDST, byte 0 bit [0]

Store not load.

LDST	Description
`0b0`	Load or swap that returns a value to the PE.
`0b1`	Store.

Operation Type packet payload (Branch)

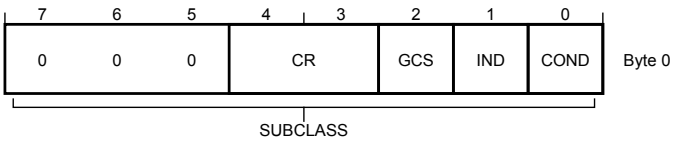

Figure D18-36 Operation Type packet payload

SUBCLASS, byte <0>

Second-level operation class. Describes the operation type.

SUBCLASS	Description
`0b000xxxxx`	Branch. Bits [4:0] are further defined as the CR, GCS, IND, and COND fields.

Other values are reserved.

This field reads as `0b000xxxxx`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CR, byte 0 bits [4:3]

Information about whether the branch was a subroutine call or a subroutine return instruction.

CR	Description
`0b00`	Whether the sampled branch operation is a Branch with link or a Procedure return operation is not recorded.
`0b01`	Branch with link.
`0b10`	Procedure return.
`0b11`	Neither Branch with link nor Procedure return.

If FEAT_SPE_CRR is not implemented, then the only permitted value is `0b00`.

If FEAT_SPE_CRR is implemented, then the value of `0b00` is not permitted.

GCS, byte 0 bit [2]

Guarded control stack data access.

When FEAT_SPEv1p4 is implemented

Indicates whether the sampled branch operation generated a Guarded control stack data access.

GCS	Description
`0b0`	No Guarded control stack data access.
`0b1`	Guarded control stack data access.

If FEAT_GCS is not implemented, then the only permitted value is `0b0`.

Otherwise

This bit reads-as-zero.

IND, byte 0 bit [1]

Indirect.

IND	Description
`0b0`	Direct branch.
`0b1`	Indirect branch.

COND, byte 0 bit [0]

Conditional.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COND	Description
`0b0`	Unconditional branch.
`0b1`	Conditional branch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.8 Padding

Purpose

Allows the PE to create alignment in the protocol buffer.

Attributes

8-bit packet.

Field descriptions

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Byte 0

Figure D18-37 Padding

Byte <0>

This field reads as `0b00000000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D18.2.9 Timestamp packet

Purpose

The 64-bit timestamp value when the operation was sampled. The Timestamp packet must come at the end of the record. If the Timestamp packet is not present, an End packet must come at the end of the record.

Attributes

Multi-part packet comprising:

- 8-bit header.
- 64-bit payload.

Timestamp packet header

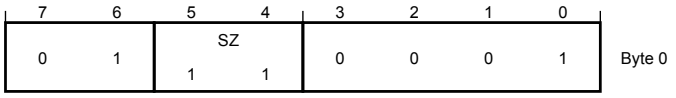

Figure D18-38 Timestamp packet header

Byte 0 bits [7:6]

This field reads as `0b01`.

SZ, byte 0 bits [5:4]

Payload size.

SZ	Description
`0b11`	Doubleword.

This field reads as `0b11`.

Byte 0 bits [3:0]

This field reads as `0b0001`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Timestamp packet payload

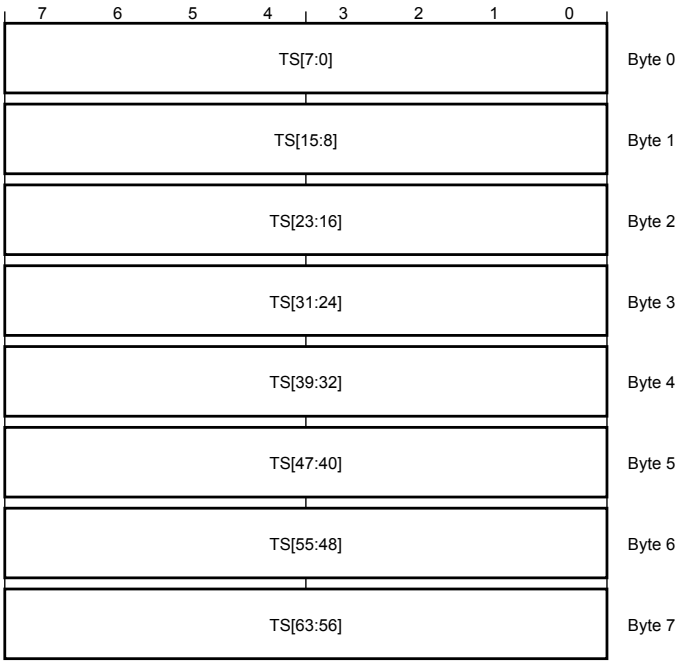

Figure D18-39 Timestamp packet payload

TS, bytes <7:0>

Timestamp value when the operation was sampled. The value depends on the result of `CollectTimeStamp()`:

- If `TimeStamp_Virtual`, the value of `PhysicalCountInt()` - CNTVOFF_EL2.
- If `TimeStamp_Physical`, the value of `PhysicalCountInt()`.
- If `TimeStamp_OffsetPhysical`, the value of `PhysicalCountInt()` - CNTPOFF_EL2. Only supported when FEAT_ECV is implemented.

If `CollectTimeStamp()` returns `TimeStamp_None`, then the Timestamp packet is not included and an End packet must come at the end of the record.

However, if the Generic Timer System counter is disabled and `CollectTimeStamp()` returns a value other than `TimeStamp_None`, then it is IMPLEMENTATION DEFINED which of the following occurs:

- The Statistical Profiling Unit behaves as if `CollectTimeStamp()` returns the value `TimeStamp_None`.
- The value of this field in the record is UNKNOWN.

Note

This set of behaviors is only permitted when the actual System counter is disabled, that is, when CNTEN.EN is `0b0`. It does not apply when the System counter is enabled but not accessible at the current Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D19
The Branch Record Buffer Extension

This chapter describes the *Branch Record Buffer Extension* (BRBE). It contains the following sections:

- About the Branch Record Buffer Extension.
- Branch record filtering.
- Branch record buffer operation.
- Branch record buffer.
- Programmers' model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D19.1 About the Branch Record Buffer Extension

I_{LQHFJ} The Branch Record Buffer Extension provides control path information for compiling and optimizing software. These directed optimizations extract information about hotspots and common control paths in the code.

FEAT_BRBE provides a mechanism for capturing control path history in a low-cost manner.

FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

D19.1.1 Branch records

I_{JBTBH} Each *Branch record* consists of 3 registers:

- BRBINF<n>_EL1.
- BRBSRC<n>_EL1.
- BRBTGT<n>_EL1.

I_{GBCQW} Taken branch instructions, as defined by Branches, Exception generating, and System instructions, generate a Branch record.

I_{LFVJR} Exceptions generate a Branch record.

R_{RPVXK} A *Half-source Branch record* has BRBINF<n>_EL1.VALID set to `0b10`.

R_{LPLYK} A *Half-target Branch record* has BRBINF<n>_EL1.VALID set to `0b01`.

R_{GSMRH} A *Full Branch record* has BRBINF<n>_EL1.VALID set to `0b11`.

R_{XRHTY} BRBINF<n>_EL1.VALID indicates the validity of a Branch record:

- If BRBINF<n>_EL1.VALID is `0b00`, the Branch record is invalid.
- Otherwise, the Branch record is valid.

R_{MLGCF} When a Branch record is generated for any branch or exception which does not transition between a BRBE Prohibited region and a BRBE Non-prohibited region, the Branch record is a *Full Branch record*.

See Branch records for exceptions and Branch records for exception returns for more details on when a Half-source Branch record or a Half-target Branch record is generated.

I_{ZCHRF} When an Instruction Synchronization Barrier instruction causes a Context synchronization event which synchronizes an update to one or more System registers which are indirectly read when generating a Branch record, the synchronization of those register updates occurs before the registers are indirectly read. Such order is generally consistent with indirect reads of System registers performed by events which cause a Context synchronization event.

R_{ZHDCC} When an exception or exception return instruction causes a Context synchronization event which synchronizes an update to one or more System registers which are used to determine whether the source of the Branch record is from a BRBE Prohibited region, indirect reads of those System registers are permitted to occur before the Context synchronization event. Specifically, the registers indirectly read by `BranchRecordAllowed()`. All other indirect reads of System registers used for creation of a Branch record occur after the Context synchronization event, including those for determining whether the Branch record is filtered or not, and those used for determining whether Branch recording is prohibited at the target.

R_{TWYWS} Where an operation such as an exception or exception return causes a Context synchronization event and that synchronizes a System register update which permits a BRBE freeze event to occur, and the same operation also changes the PE state to prevent a BRBE freeze event from occurring, it is IMPLEMENTATION SPECIFIC whether the BRBE freeze event occurs.

R_{CBHRY} The reason for the *Branch record* is captured in BRBINF<n>_EL1.TYPE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D19.1.2 Cycle counting

I_{RRBFF} Each Branch record contains a cycle count value which is representative of the time taken between each Branch record being generated. The cycle count value can be used to determine the relative performance of the program between each Branch record. For large cycle count values, the value stored in each Branch record is encoded to use less storage, with a small loss of precision in the value.

R_{HQXNW} The size of the cycle counter used to generate cycle count values is IMPLEMENTATION DEFINED, from one of the sizes indicated in BRBIDR0_EL1.CC.

R_{KVRBB} Each Branch record contains a cycle count value which indicates the number of PE clock cycles that occurred between the previous Branch record being generated and this Branch record being generated.

R_{SBXCF} In a multithreaded implementation, the cycle counter only counts cycles on which the thread was active.

I_{VPPYW} The Branch record counts cycles in the same way as the CPU_CYCLES PMU event counts cycles when PMEVTYPER<n>_EL0.MT is 0. For more information, see Cycle event counting.

R_{PGXMB} For the purposes of the cycle count, a Branch record is generated only when the corresponding branch instruction or exception is guaranteed to be architecturally executed and the target address has been calculated. Arm recommends that the Branch record is generated as soon after this point as possible.

I_{KBDLR} When a branch target address contains an address tag, the target address captured in the Branch record is the virtual address with the address tag removed.

R_{MJDLC} The cycle count value in a Branch record is Branch cycle count unknown when any of the following are true:

- If EL2 is implemented, BRBCR_EL2.CC is 0.
- BRBCR_EL1.CC is 0.
- This is the first Branch record after the PE exited a BRBE Prohibited region.
- This is the first Branch record after cycle counting has been enabled.
- This is the first Branch record after BRBFCR_EL1.PAUSED is cleared from 1 to 0.
- This is the first Branch record after execution of a BRB IALL instruction.

Note

This applies even when EL2 is disabled in the current Security state.

R_{PBJTJ} When the cycle count value in a Branch record is Branch cycle count unknown:

- BRBINF<n>_EL1.CCU has the value 1.
- BRBINF<n>_EL1.CC contains a value which is all zeros.

The number of cycles indicated by this Branch record is UNKNOWN.

R_{XGSSZ} If the cycle count value in a Branch record would exceed the maximum value of the cycle counter, then:

- BRBINF<n>_EL1.CCU has the value 0.
- BRBINF<n>_EL1.CC contains a value which is all ones.

R_{JZWPG} If the cycle count value in a Branch record is not UNKNOWN and would not exceed the maximum value of the cycle counter, then:

- BRBINF<n>_EL1.CCU has the value 0.
- BRBINF<n>_EL1.CC contains the cycle count value, encoded as defined in BRBINF<n>_EL1.CC.

D19.1.3 Mispredicted branches

I_{QHGFW} Each Branch record generated for a branch instruction contains an indication of whether the branch was correctly or incorrectly predicted by the PE. Branch prediction behavior is IMPLEMENTATION DEFINED and this is an indication of whether such prediction succeeded, or not.

R_{XHWRB} For a Branch record of a branch instruction, one of the following occurs:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL2 is implemented and BRBCR_EL2.MPRED is 0, then BRBINF<n>_EL1.MPRED is set to 0.
- Else if BRBCR_EL1.MPRED is 0, then BRBINF<n>_EL1.MPRED is set to 0.
- Otherwise:
  — BRBINF<n>_EL1.MPRED is 0 for a correctly predicted branch.
  — BRBINF<n>_EL1.MPRED is 1 for an incorrectly predicted branch.

Note: This applies even when EL2 is disabled in the current Security state.

R_{DHNPJ} For a Branch record of an exception, BRBINF<n>_EL1.MPRED has the value 0.

R_{LBRGV} An incorrectly predicted branch is when any of the following is true:

- The direction of a conditional branch was incorrectly predicted at least once during the execution of the instruction.
- The target of a branch was incorrectly predicted at least once during the execution of the instruction.
- The branch was not predicted by a branch predictor.

R_{RDLQF} A correctly predicted branch is one that is not incorrectly predicted.

D19.1.4 BRBE Prohibited regions

I_{NVWPM} An executable program might contain regions of code that are prohibited to generate Branch records, and these regions are called BRBE Prohibited regions. These regions are usually associated with a different Security state or Exception level.

I_{DMPQZ} BRBE Prohibited regions are controlled by the following:

- BRBCR_EL1.E0BRE.
- BRBCR_EL1.E1BRE.
- BRBCR_EL2.E0HBRE.
- BRBCR_EL2.E2BRE.
- MDCR_EL3.SBRBE.

I_{HPZWM} While executing outside a BRBE Prohibited region, Branch records might not be generated because the Branch Record Buffer Extension has a number of filtering functions.

R_{FHGJN} Execution in AArch32 state is a BRBE Prohibited region.

R_{LPYBQ} Execution in Debug state is a BRBE Prohibited region.

R_{JWWFY} When FEAT_BRBEv1p1 and EL3 are implemented:

- When MDCR_EL3.{E3BREC, E3BREW} is {`0b0`, `0b1`} or MDCR_EL3.{E3BREC, E3BREW} is {`0b1`, `0b0`}, self-hosted EL3 branch recording is enabled.
- When MDCR_EL3.{E3BREC, E3BREW} is {`0b0`, `0b0`} or MDCR_EL3.{E3BREC, E3BREW} is {`0b1`, `0b1`}, self-hosted EL3 branch recording is disabled.

R_{GLKGW} Execution at EL3 is a BRBE Prohibited region when any of the following are true:

- FEAT_BRBEv1p1 is not implemented.
- Self-hosted EL3 branch recording is disabled.

R_{SFZQD} Execution at EL2 is a BRBE Prohibited region when any of the following are true:

- BRBCR_EL2.E2BRE is 0.
- MDCR_EL3.SBRBE is `0b00`.
- MDCR_EL3.SBRBE is `0b01` and the PE is in Secure state.

R_{YPHYG} Execution at EL1 is a BRBE Prohibited region when any of the following are true:

- BRBCR_EL1.E1BRE is 0.
- MDCR_EL3.SBRBE is `0b00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MDCR_EL3.SBRBE is `0b01` and the PE is in Secure state.

R_{DBPCP} Execution at EL0 is a BRBE Prohibited region when any of the following are true:

- EL2 is disabled in the current Security state or HCR_EL2.TGE is 0, and BRBCR_EL1.E0BRE is 0.
- EL2 is enabled in the current Security state and HCR_EL2.TGE is 1, and BRBCR_EL2.E0HBRE is 0.
- MDCR_EL3.SBRBE is `0b00`.
- MDCR_EL3.SBRBE is `0b01` and the PE is in Secure state.

R_{YGGSC} While the PE is executing code from a BRBE Prohibited region, no data is captured in Branch records that might provide information about execution in the BRBE Prohibited region.

D19.1.5 Branch records for exceptions

R_{YSKQK} When an exception is taken from a BRBE Prohibited region to a BRBE Prohibited region, no Branch record is generated.

R_{KRJQC} When an exception is taken from a BRBE Non-prohibited region, or an exception is taken to a BRBE Non-prohibited region:

- If the target Exception level is EL1, a Branch record is generated only if BRBCR_EL1.EXCEPTION is 1.
- If the target Exception level is EL2, a Branch record is generated only if BRBCR_EL2.EXCEPTION is 1.
- If the target Exception level is EL3, a Branch record is generated only if FEAT_BRBEv1p1 is implemented and self-hosted EL3 branch recording is enabled.

R_{YBJDJ} When a Branch record is generated for an exception:

- If the exception is taken from a BRBE Prohibited region, then a Half-target Branch record is generated.
- If the exception is taken from a BRBE Non-prohibited region to a BRBE Prohibited region, then a Half-source Branch record is generated.
- If the exception is taken from a BRBE Non-prohibited region to a BRBE Non-prohibited region, then a *Full Branch record* is generated.

R_{LLCTG} When entering Debug state:

- If the entry is from a BRBE Prohibited region, no Branch record is generated.
- If the entry is from a BRBE Non-prohibited region, then a Half-source Branch record is generated.

I_{MZNRY} When a Half-source Branch record or a *Full Branch record* is generated for an Illegal Execution state exception, the source information in the *Branch record* indicates where the exception was taken from, in the same way as all other exceptions.

R_{BZCRW} A Branch record for an exception which contains a valid source address has the source address set to the preferred exception return address for the exception.

R_{FYLTC} A Branch record for an exception which contains a valid target address has the target address set to the address of the exception vector.

D19.1.6 Branch records for exception returns

R_{LMXHS} When an exception return instruction is executed in a BRBE Prohibited region and branches to a BRBE Prohibited region, no Branch record is generated.

R_{ZSHDL} When an exception return instruction is executed in a BRBE Non-prohibited region, or an exception return instruction branches to a BRBE Non-prohibited region:

- If the exception return instruction is executed at EL3, a Branch record is generated only if FEAT_BRBEv1p1 is implemented and self-hosted EL3 branch recording is enabled.
- If the exception return instruction is executed at EL2, a Branch record is generated only if BRBCR_EL2.ERTN is 1.
- If the exception return instruction is executed at EL1, a Branch record is generated only if BRBCR_EL1.ERTN is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{ZTGMW} When a Branch record is generated for an exception return instruction:

- If the exception return instruction is executed in a BRBE Prohibited region, then a Half-target Branch record is generated.
- If the exception return instruction is executed in a BRBE Non-prohibited region and branches to a BRBE Prohibited region, then a Half-source Branch record is generated.
- If the exception return instruction is executed in a BRBE Non-prohibited region and branches to a BRBE Non-prohibited region, then a *Full Branch record* is generated.

R_{RBCFP} When exiting from Debug state:

- If the exit is to a BRBE Prohibited region, no Branch record is generated.
- If the exit is to a BRBE Non-prohibited region, then a Half-target Branch record is generated.

I_{NGWPR} When a Half-target Branch record or a *Full Branch record* is generated for an exception return instruction which is an illegal return or a legal return which sets PSTATE.IL to 1, the target information in the Branch record indicates the target of the branch:

- BRBTGT<n>_EL1.ADDRESS contains the target of the branch.
- BRBINF<n>_EL1.EL contains the value that is loaded in to PSTATE.EL.

I_{JCYHD} When a Half-target Branch record or a *Full Branch record* is generated for an exception return instruction which is an illegal return or a legal return which sets PSTATE.IL to 1, for the purposes of determining whether the target is a BRBE Prohibited region the value that is loaded in to PSTATE.EL is used as the target Exception level.

PSTATE.EL is unchanged on an illegal return, so the current Exception level is the target of the illegal return, regardless of where the return was attempting to return to.

D19.1.7 PE speculation

R_{KXTKS} The Branch records only contain information for a branch, exception, or exception return that is architecturally executed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D19.2 Branch record filtering

INZWBP For Branch records generated outside a BRBE Prohibited region, it is useful to reduce the number of records that are generated to match their use. Table D19-1 lists some different use cases.

Table D19-1 Example use cases for filtering

Use case	Description
Control path	• All branches • Subroutine returns • Exceptions • Exception returns
Call path	• Branch with link instructions • Subroutine returns
Kernel calls	• Exceptions • Exception returns

D19.2.1 Filtering on type

IFSNVG The Branch records can be filtered by independently enabling the generation of the following types:

- Exception.
- Exception return.
- Direct Branch with link.
- Indirect Branch with link.
- Return from subroutine.
- Indirect Branches.
- Conditional Direct Branches.
- Unconditional Direct Branches.

RLYGJZ Control of when Branch records for exceptions are generated is controlled by BRBCR_EL1.EXCEPTION and BRBCR_EL2.EXCEPTION. See Branch records for exceptions for details.

Table D19-2 Exception mapping for exceptions taken to AArch64 state

Reason	Type
Branch Target exception	Inst fault
Breakpoint	Inst debug
Exceptions due to SME functionality	Trap
EXLOCK exception	Inst fault
FIQ	FIQ
GPC exception due to data access	Data fault
GPC exception due to instruction access	Inst fault
Guarded Control Stack Data Check exception	Data fault
HVC	Call

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reason	Type
Halting debug event	Debug halt
IRQ	IRQ
Illegal execution state	Trap
Instruction Abort	Inst fault
Instruction or event trapped by a control bit	Trap
MemCpy or MemSet	Trap
Misaligned PC	Alignment
PAC Fail	Data fault
PMU exception	Inst debug
SError exception	System Error
SMC due to HCR_EL2.TSC	Trap
SMC other than due to HCR_EL2.TSC	Call
SVC due to HFGITR_EL2.SVC_EL0 or HFGITR_EL2.SVC_EL1	Trap
SVC other than due to HFGITR_EL2.SVC_EL0 or HFGITR_EL2.SVC_EL1	Call
Software Breakpoint Instruction	Inst debug
Software Step	Inst debug
Stack Pointer Misalignment	Alignment
Synchronous Data Abort	Data fault
Traps of `GCSSTR` and `GCSSTTR`	Trap
UNDEFINED instruction	Trap
Watchpoint	Data debug

R_{RCGVB} Control of when Branch records for exception return instructions are generated is controlled by BRBCR_EL1.ERTN and BRBCR_EL2.ERTN. See Branch records for exception returns for details.

Table D19-3 A64 return from exception instructions

Instruction	Description
ERET	Return From Exception
ERETAA	Authenticate and Exception return
ERETAB	Authenticate and Exception return

R_{RBDXK} Each of the direct branch with link instructions only generates a Branch record when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.DIRCALL is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.DIRCALL is 0 and BRBFCR_EL1.EnI is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D19-4 A64 direct branch with link instructions

Instruction	Description
BL	Branch with link

R_{VBGTZ} Each of the indirect branch with link instructions only generates a Branch record when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.INDCALL is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.INDCALL is 0 and BRBFCR_EL1.EnI is 1.

Table D19-5 A64 indirect branch with link instructions

Instruction	Description
BLR	Branch with link to register
BLRAA	Authenticate and branch with link
BLRAAZ	Authenticate and branch with link
BLRAB	Authenticate and branch with link
BLRABZ	Authenticate and branch with link

R_{CKNBH} Each of the return from subroutine instructions only generates a Branch record, when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.RTN is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.RTN is 0 and BRBFCR_EL1.EnI is 1.

Table D19-6 A64 return from subroutine instructions

Instruction	Description
RET	Return From subroutine
RETAA	Authenticate and function return
RETAASPPC	Authenticate and function return
RETAASPPCR	Authenticate and function return
RETAB	Authenticate and function return
RETABSPPC	Authenticate and function return
RETABSPPCR	Authenticate and function return

R_{KKLDV} Unless covered by other rules, each of the indirect branch instructions only generates a Branch record when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.INDIRECT is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.INDIRECT is 0 and BRBFCR_EL1.EnI is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D19-7 A64 indirect branch instructions

Instruction	Description
BR	Branch to register
BRAA	Authenticate and branch
BRAAZ	Authenticate and branch
BRAB	Authenticate and branch
BRABZ	Authenticate and branch

R_{BBNSZ} Unless covered by other rules, each of the conditional direct branch instructions only generates a Branch record when the instruction is taken, is executed in a BRBE Non-prohibited region, and if any of the following are true:

- BRBFCR_EL1.CONDDIR is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.CONDDIR is 0 and BRBFCR_EL1.EnI is 1.

Table D19-8 A64 conditional direct branch instructions

Instruction	Description
B.cond	Conditional Branch
BC.cond	Branch Consistent conditionally
CB<cc> (immediate)	Compare register with immediate and branch
CB<cc> (register)	Compare registers and branch
CBB<cc>	Compare bytes and branch
CBH<cc>	Compare halfwords and branch
CBZ or CBNZ	Compare with zero and branch
TBZ or TBNZ	Test and branch

Note: `BC.cond` and `B.cond` instructions with the `AL` or `NV` condition code are considered conditional.

R_{FJYVT} Unless covered by other rules, each of the unconditional direct branch instructions only generates a Branch record when the instructions are executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.DIRECT is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.DIRECT is 0 and BRBFCR_EL1.EnI is 1.

Table D19-9 A64 unconditional direct branch instructions

Instruction	Description
B	Unconditional Branch

R_{FJYDC} It is IMPLEMENTATION DEFINED whether Branch records are generated for each of the following taken unconditional direct branch instructions when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.DIRECT is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.DIRECT is 0 and BRBFCR_EL1.EnI is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D19-10 Optional A64 unconditional direct branch instructions

Instruction	Description
ISB	Instruction Synchronization Barrier

SXZRTW Writing a value of `0b0000_0001` to the filter controls, BRBFCR_EL1<23:16>, ensures Branch records are generated for all branch instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D19.3 Branch record buffer operation

R_{LKWNB} The Branch Record Buffer Extension operation is controlled by the BRBCR_EL1, BRBCR_EL2, and BRBFCR_EL1 registers.

R_{PYBRZ} Generation of Branch records is *paused* when BRBFCR_EL1.PAUSED is 1.

R_{YDZNK} When generation of Branch records is paused, Branch records are not generated.

R_{NXCWF} If EL2 is implemented, a BRBE freeze event occurs when all of the following are true:

- BRBCR_EL1.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSCLR_EL0[(MDCR_EL2.HPMN-1):0] is nonzero.
- The PE is in a BRBE Non-prohibited region.

R_{GXGWY} If EL2 is implemented, a BRBE freeze event occurs when all of the following are true:

- BRBCR_EL2.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSCLR_EL0[(PMCR_EL0.N-1):MDCR_EL2.HPMN] is nonzero.
- The PE is in a BRBE Non-prohibited region.

This applies even when EL2 is disabled in the current Security state.

R_{PKTXQ} If EL2 is not implemented, a BRBE freeze event occurs when all of the following are true:

- BRBCR_EL1.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSCLR_EL0[(PMCR_EL0.N-1):0] is nonzero.
- The PE is in a BRBE Non-prohibited region.

R_{LDMVK} If FEAT_PMUv3_ICNTR is implemented, then a BRBE freeze event occurs when all of the following are true:

- BRBCR_EL1.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSCLR_EL0.F0 is 1.
- The PE is in a BRBE non-Prohibited Region.

R_{LBQZR} If FEAT_PMUv3_SS is implemented, on a successful Capture event, then a BRBE freeze event occurs when all of the following are true:

- FEAT_BRBE is implemented.
- BRBCR_EL1.FZPSS is 1.
- Either EL2 is not implemented or BRBCR_EL2.FZPSS is 1.
  This applies even when EL2 is disabled in the current Security state.
- Generation of Branch records is not paused.
- The PE is in a non-Prohibited Region.

I_{YTVCC} Where a branch instruction, exception, or Debug state entry or exit causes a change to whether execution is in a Prohibited region and this change results in the conditions for a BRBE freeze event to be met, it is IMPLEMENTATION SPECIFIC whether the instruction, exception, Debug state entry or exit generates a Branch record before the BRBE freeze event occurs, or does not generate a Branch record because the BRBE freeze event has already occurred and branch record generation is paused.

This means the Branch record might or might not be captured before the BRBE freeze occurs.

R_{BHYTD} On a BRBE freeze event:

- BRBFCR_EL1.PAUSED is set to 1.
- The current timestamp is captured in BRBTS_EL1.

R_{QKQZL} The source of value of the timestamp captured in BRBTS_EL1 is selected by the combination of programming of BRBCR_EL2.TS and BRBCR_EL1.TS. See Table D19-11 and `BRBETimeStamp()`.

The timestamp is a choice between:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Physical time, which is defined by the physical count value returned by `PhysicalCountInt()`.
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of `PhysicalCountInt()` minus a physical offset.
  If any of the following are true, the physical offset is zero:
  — EL2 is not implemented.
  — FEAT_ECV_POFF is not implemented.
  — CNTHCTL_EL2.ECV is 0.
  — EL3 is implemented and SCR_EL3.ECVEn is 0.
  Otherwise the physical offset is the value of CNTPOFF_EL2.
- Virtual time, which is defined as the value of `PhysicalCountInt()` minus a virtual offset.
  If any of the following are true, the virtual offset is zero:
  — EL2 is not implemented.
  Otherwise, the virtual offset is always CNTVOFF_EL2, including when a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero.

Table D19-11 Captured timestamp

BRBCR_EL2.TS	BRBCR_EL1.TS	Captured timestamp
`0b00` (delegate)	`0b01` (virtual)	`PhysicalCountInt()` - virtual offset
	`0b10` (offset physical)	`PhysicalCountInt()` - physical offset
	`0b11` (physical)	`PhysicalCountInt()`
`0b01` (virtual)	`0bxx`	`PhysicalCountInt()` - virtual offset
`0b10` (offset physical)	`0bxx`	`PhysicalCountInt()` - physical offset
`0b11` (physical)	`0bxx`	`PhysicalCountInt()`

If EL2 is not implemented, the Effective value of BRBCR_EL2.TS is `0b00`.

R_{GWMZV} When a valid Branch record is captured in the Branch record buffer storage, the BRB_FILTRATE event is generated.

R_{GMCHN} When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event. For more information on PMU event filtering, see Exception-related events.

I_{WGZSC} It is CONSTRAINED UNPREDICTABLE whether a BRB_FILTRATE event is generated after a `BRB INJ` causes a Branch record to be injected.

I_{SJDRW} It is expected that the Branch record buffer storage is in a BRBE Prohibited region during software context switches.
During software context switches, if the PMU event counters are not prohibited from counting events, it is expected that event counters set to count the BRB_FILTRATE event are prohibited from counting.

I_{RMGDV} The architecture does not define when PMU events are counted relative to the instructions that caused the event. Events generated by an instruction might be counted before or after the instruction becomes architecturally executed, and events might be counted for operations that do not become architecturally executed. This means that events can be counted speculatively and/or out-of-order regarding the simple sequential execution of the program. Events might also be counted simultaneously by other event counters when the overflow occurs, including events from different instructions. In addition, multiple instances of an event might occur simultaneously, meaning that an event counter unsigned overflow can yield a nonzero value in the event counter.
Furthermore, the Branch records are generated only for architecturally executed operations. See R_{KXTKS}.
These properties mean that, unless otherwise stated, on a BRBE freeze event, it is CONSTRAINED UNPREDICTABLE whether the branches that define the basic block containing the instruction causing that event are captured in the Branch record buffer.
An exception to this relaxation applies for the BRB_FILTRATE event.

S_{JCHLT} If a direct read of PMOVSCLR_EL0 returns a nonzero value for a subset of the overflow flags, such that one of R_{NXCWF}, R_{GXGWY}, or R_{PKTXQ} means that a BRBE freeze event should occur, then a direct read of BRBFCR_EL1 ordered after the direct read of PMOVSCLR_EL0 will return BRBFCR_EL1.PAUSED is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{QNBDV} Direct reads of System registers require explicit synchronization for following direct reads of other System registers to be ordered after the first direct read. For more information, see General behavior of accesses to the AArch64 System registers.

R_{SRJND} If a direct read of BRBFCR_EL1.PAUSED returns 1, then no operations ordered after the direct read will generate further Branch records until BRBFCR_EL1.PAUSED is cleared by software.

Note: The subsequent operations can be ordered by a Context synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D19.4 Branch record buffer

I_{FBHCC} The Branch record buffer can contain:

- 8 Branch records.
- 16 Branch records.
- 32 Branch records.
- 64 Branch records.

This is known as the Branch record buffer storage.

R_{KSLSM} The Branch record buffer storage has a maximum number of Branch records as defined by BRBIDR0_EL1.NUMREC.

I_{PPBZP} The Branch record buffer provides System registers to access the Branch records stored in the Branch record buffer storage. These System registers provide access to up to 32 Branch records without the need for explicit synchronization between each System register read. When more than 32 Branch records are implemented, the Branch record buffer provides a banking mechanism to provide access to multiple banks, each bank containing up to 32 Branch records. BRBFCR_EL1.BANK controls which bank is currently selected, and updates to BRBFCR_EL1.BANK require explicit synchronization before accessing the bank.

R_{WLSWP} Accessing Branch records 0-31 is performed by setting BRBFCR_EL1.BANK to `0b00`.

R_{WRJLW} Accessing Branch records 32-63 is performed by setting BRBFCR_EL1.BANK to `0b01`.

R_{TJGLK} The Branch record with index `0` is the youngest captured branch.

R_{HTRNR} The Branch record with index `n` is younger than *Branch record* with index `n+1`.

R_{DTPDK} On the generation of a new Branch record, if the Branch record buffer storage is full, then the oldest *Branch record* is lost.

R_{SQLCX} When the buffer contains `M` valid Branch records, where `M > 0` and `M` is less than the maximum number of *Branch records*, all of the following are true:

- Branch records with index `0` to `M-1` are all valid.
- All other Branch records are invalid.

R_{PGDLX} The creation of a Branch record is considered an indirect write to BRBTGT<n>_EL1, BRBSRC<n>_EL1 and BRBINF<n>_EL1, and therefore requires explicit synchronization before being read.

I_{SFFNF} The generation of Branch records performs indirect reads and indirect writes of System registers.

I_{KFYTV} Synchronization requirements for AArch64 System registers defines the synchronization requirements for direct reads, direct writes, indirect reads, and indirect writes of System registers made by instructions and external agents.

D19.4.1 Invalidating the record buffer

R_{LLHYN} Execution of `BRB IALL` causes all Branch records to be invalidated.

R_{PFRNW} A Branch record, **R**, is invalidated by the instruction `BRB IALL`, **W**, if all of the following are true:

- **R** is caused by a branch operation or exception, **B**.
- **B** is either:
  - In program order before a *Context synchronization event*, **CSE**.
  - Is the *Context synchronization event*.
- **CSE** is in program order before **W**.

R_{LWPKR} A Branch record **R** is not invalidated by the instruction `BRB IALL`, **W**, if all of the following are true:

- **R** is caused by a branch operation or exception, **B**.
- **B** is in program order after a *Context synchronization event*, **CSE**.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- **CSE** is in program order after **W**.

R_{WMZKF} It is CONSTRAINED UNPREDICTABLE whether a Branch record **R** is invalidated by the instruction `BRB IALL`, **W**, if all of the following are true:

- CSE_1 is in program order before **W**.
- **R** is caused by a branch operation or exception, **B**.
- **B** is in program order after a *Context synchronization event*, CSE_1
- **B** is either:
  - In program order before a *Context synchronization event*, CSE_2.
  - Is the *Context synchronization event*, CSE_2.
- CSE_2 is in program order after **W** and there are no other CSEs between CSE_1 and CSE_2.

If a Branch record is invalidated, all older Branch records are invalidated.

R_{JSCWK} When a Branch record has been invalidated, it remains invalid until it is overwritten by any of the following:

- A new Branch record is created.
- A Branch record is injected using the `BRB INJ` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D19.5 Programmers' model

R_{BNGTH} Reads from an unimplemented Branch record return the value zero.

R_{PKZCF} All Branch records captured while generation of Branch records is not paused, must represent a continuous block of execution for all BRBE Non-prohibited regions.

I_{XSBPR} The captured Branch records might not represent a continuous block if generation of Branch records is paused at any time. To avoid this non-continuous nature, the `BRB IALL` instruction can be used to invalidate all Branch records while generation is paused.

R_{PMRRL} If a Branch record cannot be captured for a branch instruction or exception that is not prohibited and has been selected to generate a record, then all the Branch records must be invalidated. The reasons for a PE being unable to capture a Branch record are IMPLEMENTATION DEFINED and Arm recommends that such reasons are rare.

I_{QJFSV} When a process is migrated to a PE with a smaller number of Branch records implemented then the information from the older Branch records will be lost.

I_{BDKJJ} When FEAT_BRBE is implemented, the following fields are provided in System registers to control access to the Branch record buffer functionality:

- When EL2 is implemented:
  - HDFGRTR_EL2.nBRBIDR.
  - HDFGRTR_EL2.nBRBCTL.
  - HDFGWTR_EL2.nBRBCTL.
  - HDFGRTR_EL2.nBRBDATA.
  - HDFGWTR_EL2.nBRBDATA.
  - HFGITR_EL2.nBRBINJ.
  - HFGITR_EL2.nBRBIALL.
- When EL3 is implemented:
  - MDCR_EL3.SBRBE.

S_{GCYTK} When self-hosted EL3 branch recording is enabled, the following registers must be programmed:

- BRBCR_EL1 and BRBCR_EL2. Software must program these registers to control the following:
  - Recording of exceptions taken to EL1 and EL2.
  - Recording of exception returns from EL1 and EL2.
  - Branch recording at EL0, EL1, and EL2.
  - Occurrence of BRBE freeze events on PMU overflows.
  - Timestamp source.
  - Misprediction information.
  - Cycle count information.
- BRBFCR_EL1. Software must program this register to control the following:
  - Selection of Branch record buffer bank.
  - Selection of branch types to record at all Exception levels.
  - Pausing of Branch recording.

Software should also consider how MDCR_EL3.SBRBE should be programmed to either allow or prohibit access to the captured branch records from lower Exception levels. Even if self-hosted EL3 branch recording is not being used, software should consider whether a `BRB IALL` instruction should be executed before executing software at lower Exception levels.

S_{YLMQQ} Software must invalidate the Branch records after a PE reset to ensure that details of execution before the reset event are not leaked.

I_{FPJPQ} MDCR_EL3.E3BREC resets to 0 on a Cold reset, and MDCR_EL3.E3BREW resets to 0 on a Warm reset. This allows software to program MDCR_EL3.E3BREC and MDCR_EL3.E3BREW such that the BRBE continues recording after Warm reset, or stops recording at Warm reset.

I_{NZMWQ} When FEAT_BRBEv1p1 is implemented, certain fields in the following registers are reset to an architecturally UNKNOWN value on a Cold reset and unchanged on a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BRBCR_EL1.
- BRBCR_EL2.
- BRBFCR_EL1.

D19.5.1 Manual injection of Branch records

I_{DXNLX} The Branch Record Buffer Extension supports the ability to manually create Branch records and inject them in to the Branch record buffer storage. The primary purpose of the injection functionality is to support the restore of the Branch record buffer storage contents, particularly during software context switch events, including migration of software between PEs. The Branch record buffer storage contents are read out using direct reads of BRBSRC<n>_EL1, BRBTGT<n>_EL1, and BRBINF<n>_EL1.

R_{FYXNL} The Branch Record Injection data registers are:

- BRBSRCINJ_EL1.
- BRBTGTINJ_EL1.
- BRBINFINJ_EL1.

I_{FJHMP} Branch record injection consists of creating a single Branch record using direct writes to the Branch Record Injection data registers, then injecting the record into the Branch record buffer storage using `BRB INJ`. This process injects a single Branch record as the youngest entry in the Branch record buffer storage. This process is repeated for each Branch record to be added to the Branch record buffer storage.

I_{BCZRK} When `BRB INJ` is executed outside of a BRBE Prohibited region, it is CONSTRAINED UNPREDICTABLE whether a Branch record is injected and it is CONSTRAINED UNPREDICTABLE whether the cycle count value in the next Branch record to be generated is Branch cycle count unknown.

R_{XVDNN} When `BRB INJ` is executed inside a BRBE Prohibited region, the contents of the Branch Record Injection data registers are used to create a Branch record which is added to the Branch record buffer storage as the youngest entry.

R_{SMBVK} When a `BRB INJ` instruction is executed inside a BRBE Prohibited region and the contents of the Branch Record Injection data registers indicates an invalid record, it is CONSTRAINED UNPREDICTABLE whether a Branch record is injected to the Branch record buffer. An invalid record is one with BRBINFINJ_EL1.VALID set to `0b00`.

R_{HNCSX} For a `BRB INJ` instruction, it is CONSTRAINED UNPREDICTABLE whether a Branch record is injected to the Branch record buffer when all of the following are true:

- The `BRB INJ` instruction is executed inside a BRBE Prohibited region.
- The contents of the Branch Record Injection data registers indicates a valid record.
- The other contents of the Branch Record Injection data registers indicate an incorrectly formatted record.

If a Branch record is injected, then the contents of the Branch record are CONSTRAINED UNPREDICTABLE.

I_{JLVPS} An example of an incorrectly formatted record is one where BRBINFINJ_EL1.VALID is `0b01` and BRBINFINJ_EL1.MPRED is `0b1`.

R_{PWKFJ} Execution of `BRB INJ` does not require explicit synchronization to use the result of direct writes to the Branch Record Injection data registers in program order before `BRB INJ`.

I_{TVDMK} The creation of a Branch record as a result of execution of `BRB INJ` does not use the result of direct writes to the Branch Record Injection data registers in program order after `BRB INJ`. Explicit synchronization is not required to ensure this ordering. For more information, see Synchronization requirements for AArch64 System registers.

R_{LKDTJ} After the execution of `BRB INJ`, the contents of the Branch Record Injection data registers are UNKNOWN.

I_{CPBKF} Changes to the BRB registers are subject to the rules for synchronization for System registers. See Synchronization requirements for AArch64 System registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D20
RAS PE Architecture

This chapter describes the RAS Extension PE Architecture. It contains the following sections:

- About the RAS Extension.
- PE error handling.
- Generating error exceptions.
- Taking error exceptions.
- Error synchronization event.
- Virtual SError exceptions and delegated SError exceptions.
- Error records in the PE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.1 About the RAS Extension

ILBKPL The RAS Extension extends the exception syndrome registers to include fields that allow the PE to report a PE error state when an Error exception is taken.

ITHGHB The RAS Extension defines System registers that are specific to RAS, including to access optional Error records defined by the RAS System Architecture. The format of the error record registers is defined in Error record System register view.

RYWXWL The RAS Extension does not prescribe the level of reliability, availability, and serviceability in the PE. The RAS features that the system and PE include, for example to detect, correct, contain, or defer errors, are IMPLEMENTATION DEFINED. The RAS Extension defines a framework for building RAS features in a PE.

Example D20-1 Minimal implementation of RAS Extension

For systems without a requirement for high reliability and availability, a minimal approach to RAS might include:

- There is no error detection. ERRIDR_EL1.NUM is zero, indicating that no error records are implemented. See Error record System register view.
- Any error signaled to the PE generates an asynchronous SError exception, and on taking the SError exception:
  - The PE error state is recorded as Uncontainable (UC).
    - If FEAT_RASv2 is implemented, then ESR_ELx.VFV is set to 0, meaning FAR_ELx is UNKNOWN and not valid. If FEAT_RASv2 is not implemented, then FAR_ELx is always UNKNOWN and not valid on taking an SError exception.
  - FEAT_PFAR is not implemented, meaning ESR_ELx.PFV is set to 0.

  See Taking error exceptions.
- The `ESB` instruction is implemented as `NOP`.
  See also ESB and virtual SError exceptions and ESB and delegated SError exceptions.

Example D20-2 Minimal implementation of RAS Extension

For systems without a requirement for high reliability and availability, a minimal approach to RAS might include:

- There is no error detection. ERRIDR_EL1.NUM is zero, indicating that no error records are implemented. See Error record System register view.
- Any error signaled to the PE generates an asynchronous SError exception, and on taking the SError exception the PE error state is recorded as Uncontainable (UC).
  See Taking error exceptions.
- The `ESB` instruction is implemented as `NOP`.
  See also ESB and virtual SError exceptions and ESB and delegated SError exceptions.

Example D20-3 Common error handling for RAS features

For a system with a requirement for high reliability and error containment:

- When errors are detected, they are recorded in error records. This includes both correctable and uncorrectable errors. Error correction and detection logic exists on all significant memory structures, and might include other non-memory structures.
- Most uncorrectable errors generate poison values to defer the error.
- On consuming poison from the memory system, a load instruction that would corrupt PE state generates a synchronous External Abort exception, regardless of the memory type. For systems where the performance impact of this approach might be an issue, the controls defined by FEAT_ANERR and FEAT_ADERR are provided.
- On taking error exceptions, the PE error state is Recoverable state (UER).
- FEAT_PFAR is implemented, and the PE makes a best effort attempt to record the faulting physical address in PFAR_ELx or MFAR_ELx, setting ESR_ELx.PFV to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IQGLBZ See also *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100) for more information about the framework for building RAS features in a system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.2 PE error handling

D20.2.1 PE error detection

IKRYQW When a PE accesses memory or other state, an error might be detected in that memory or state, and corrected, deferred, or signaled to the PE as a Detected error with an in-band error response.

Note:

- An error that is deferred might be signaled to the PE with an in-band error response.
- An error might also be signaled to a PE by means other than an in-band error response.

ISLYLK The response from memory or other state is defined by Detecting and consuming errors. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

IDWWQJ When an error is detected by a component on a read or a cache maintenance operation from the PE:

- If the error can be corrected, it is corrected and corrected data is returned.
- If the error cannot be corrected and can be deferred, it is deferred. For example, on a load by poisoning the PE state, if this is supported by the PE implementation.
- If the error cannot be corrected and if implemented and enabled at the component, the Detected error is signaled to the PE as an in-band error response.

IBKVQP When an error is detected by a component consuming a write from the PE:

- If the error can be corrected, it is corrected.
- If the error cannot be corrected and can be deferred, it is deferred to the consumer. For example, by poisoning the location being written.
- If the error cannot be corrected and if implemented and enabled at the component, the Detected error is signaled to the PE as an in-band error response.

IPDDNB The component might record the Detected error and generate a Fault handling interrupt and/or Error recovery interrupt.

IVRYFF If the component implements the RAS System Architecture, its behavior is defined by the RAS System Architecture, and depends on the nature of the error and IMPLEMENTATION DEFINED properties of the component. In each of these cases, the component might be a part of the processor, such as a cache, or might be outside of the processor.

The component might also report the error to a RAS System Architecture node, which records the error and might generate one or more of a Fault handling interrupt, Error recovery interrupt, or Critical Error interrupt depending on the features and configuration of the node.

See also Other errors.

IFLXTY An in-band error response is sometimes referred to as an *External abort*. To avoid confusion with the External abort *exception*, the RAS sections in this Manual use in-band error response to describe the response to the PE for a memory access.

For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

RWLTPV The size of the protection granule for any implemented error detection mechanism in memory is IMPLEMENTATION DEFINED and might vary from moment to moment and instruction to instruction.

For example, the size of the protection granule might depend on one or more of the following:

- The physical address of the Location being accessed.
- Whether a copy of the Location held in a cache is being accessed, and the cache it is held in.
- The type of access being made.
- If FEAT_SME is implemented, whether the PE is in Streaming SVE mode or is not in Streaming SVE mode.

IJJJGW A system might implement multiple error detection mechanisms with differing protection granule sizes.

IYBRSM For a Deferred error, Clearing Deferred errors from poisoned Locations describes mechanisms to remove poison and clear the error. The architecture also permits IMPLEMENTATION DEFINED mechanisms for clearing an error or poison from memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.2.2 PE error propagation

I_{NTXKV} The program-visible architectural state of the PE, referred to as the PE state, includes:

- General-purpose, SIMD&FP, and SVE registers.
- SME and SME2 registers.
- System registers.
- Special-purpose registers.
- PSTATE.

R_{XMBNW} An error is consumed by the PE by any of the following:

- An instruction commits the corruption into the PE state.
- The error is on an instruction fetch and the corrupt instruction is committed for execution.
- The error is on a translation table walk for a committed load, store, or instruction fetch.

I_{HVFKW} For a PE, Error propagation applies to the propagation of Detected error by the PE between the PE state, and any other PE state or memory.

Note: *Memory* includes structures that cache the contents of memory, such as an instruction cache, data cache, or TLB.

R_{VGXBJ} An error is propagated by the PE by one or more of the following occurring that would not have been permitted to occur had the fault not been activated:

- Consumption of the corrupt value by any instruction, propagating the error to the target(s) of the instruction. This includes:
  - A store of a corrupt value.
  - A write of a corrupt value to a System register, Special-purpose register, or PSTATE. Infecting a System register state might mean that the PE generates transactions that would not otherwise be permitted.
- Any operation occurring that should not have occurred, including:
  - A load, translation table walk, or instruction fetch that would not have been permitted, including those from hardware speculation or prefetching.
  - A store to an incorrect address, or a store that would not have been made or not permitted.
  - A direct or indirect write to a Special-purpose or System register that would not have been made or not permitted.
  - Assertion of any signal, such as an interrupt, that would not have been asserted.
- Any operation not occurring that should have occurred.
- Causing the PE to take an imprecise exception, other than an Error exception in response to the error itself. For more information, see Definition of a precise exception and imprecise exception.
- The PE discarding data that it holds in a modified state.
- Any other loss of required uniprocessor semantics, ordering, or coherency.

I_{RMZNK} In R_{VGXBJ}, *not have been permitted to occur* means that the observable behavior of the PE is a deviation from the correct service of the PE, as defined by the architecture. Deviations from the normal behavior of the PE implementation that would otherwise be permitted by the architecture are not deviations from correct service.

Example D20-4

A PE takes an Error exception asynchronously as follows, in program order:

1. A load returns a corrupt value from a first location to a general-purpose register.
2. The PE suppresses a store of the register to a second memory location. In particular, the location is not updated and so retains its previous value.
3. The Error exception is taken.
4. At the point when the Error exception is taken, the ordering constraints imposed by the architecture have not been violated, in particular those relating to the observability of the store at step 2.

Although the error has not been propagated, the PE state is not consistent with the PE having executed all of the instructions up to the point when the Error exception is taken, and so it would be unlikely that software would be able to recover execution.

R_{NQDWB} An error propagated by the PE is silently propagated by the PE only if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. The propagation is not part of the required operation of the PE in taking an Error exception generated by the error.
2. The propagation is not part of the required operation of the PE executing an `ESB` instruction that synchronizes the error.
3. The error is not signaled to the consumer as a Detected error or Deferred error.
4. Any of the following are true:
   - The corrupt value is held in other than the general-purpose, SIMD&FP, SVE, SME, and SME2 registers.
   - The error is propagated by an instruction in program order before either taking an Error exception generated by the error or executing an `ESB` instruction that synchronizes the error, and is propagated to outside of the general-purpose, SIMD&FP, SVE, SME, and SME2 registers.
   - The error is propagated other than by an instruction that consumes the corrupt value as an input operand but otherwise behaves correctly.

IBRBFF In R_{NQDWB}, item 4 means that after taking the Error exception generated by the error, or an `ESB`, propagating an error by, for example, storing the corrupt value to memory, is not considered as silent propagation of the error by the PE.

Example D20-5

A PE takes an Error exception in response to a load that returns a corrupt value to a general-purpose register. The error is not silently propagated to outside of the general-purpose registers before the Error exception is taken.

Neither of the following are considered silent propagation of the error by the PE:

- Taking the Error exception causes the ESR_ELx, ELR_ELx, and SPSR_ELx registers to be updated. This is part of the required operation of the PE.
- After taking the Error exception, software stores the contents of the general-purpose register to memory, and this is not signaled to memory as a Deferred error. This happens in program order after the exception is taken.

The error is not silently propagated by the PE.

Example D20-6

Further to Example D20-4, if either of the following example additional operations occur between 2 and 3, then the PE has silently propagated the error:

- A second store to a third location is performed by the PE, and the architecture requires that the first store is ordered-before the second store. For example, the second store is a store-release operation. In this case, the PE violates the external ordering constraints for the two stores, and the error is silently propagated to any observer of the second store.
- A second load from the second location returns the previous value, and a second store writes that value to a third location. In this case, the PE violates the requirements of the Coherence-before relation because the second load cannot be Coherence-before the first store. The error silently propagates to any observer of the second store.

If instead of the PE suppressing the store at 2, the PE poisons the second memory location, and in the second example propagates that poison through the second load and second store, then the error is not silently propagated.

RDTRFQ The features that a PE includes to prevent silent propagation of an error are IMPLEMENTATION DEFINED.

Example D20-7

An implementation ensures that a corrupt value in a general-purpose, SIMD&FP, SVE, or SME register is not silently propagated, by signalling a Deferred error on any write of data to any memory location, such that the memory location is poisoned.

D20.2.3 Clearing Deferred errors from poisoned Locations

D20.2.3.1 Poison-atomicity of Memory Write Effects

DYQQLJ A Memory Write Effect is *poison-atomic* if the Memory Write Effect removes a Deferred error (poison) at all Locations written to by the Memory Write Effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{XVFDG} For each store instruction with Explicit Memory Write Effects, it is IMPLEMENTATION DEFINED whether the Memory Write Effects are poison-atomic for a region of physical memory. This includes the `DC ZVA` and/or `DC GZVA` instructions.

R_{QSBKL} If a Location in a region of physical memory supports the poison-atomic property for a store instruction, then all Locations in the same naturally-aligned region support this for the same store instruction, where the size of the region is the largest translation granule that the PE supports.

I_{CSJLP} Support for the poison-atomic property for a store instruction is a property of both the PE as well as the system. That is, if the system includes a PE that supports the poison-atomic property for the instruction, then both the PE and the system support this instruction such that it is guaranteed to be poison-atomic.

For example, the system should not split cache line writes of data to be smaller than the poison granule of the downstream component in order to preserve poison-atomicity of a `DC {G}ZVA` instruction.

I_{ZRSSM} There is no architected means to discover the poison-atomic property for `DC ZVA` or `DC GZVA` for any region of physical memory.

I_{NXNTY} When a Memory Write Effect to a Location is not poison-atomic, the Memory Write Effect propagates a Deferred error previously at the Location to the new value being written.

Example D20-8

A region of physical memory uses Error Correction Codes (ECC) to provide error detection and correction. The protection granule for this region of memory is the size of a range covered by a single ECC. The ECC is also used to encode a poison value. This means the poison granule is the same size as the protection granule, for this region of physical memory.

A write that is smaller than the protection granule to this region of physical memory must read the protection granule containing the Location being written in order to calculate a new ECC for the protection granule. If the protection granule contains poison, then a poisoned value is written back, meaning the write is not poison-atomic.

If a write overwrites the entire protection granule, then the system can calculate a new ECC for the protection granule without reading the previous contents, meaning the write can be poison-atomic.

I_{GLFCV} A system might require software to stop using the protection granule until the Deferred error can be removed.

For example, by not using the physical page containing the granule until the system is reset. The architecture does not set any limit on the size of a protection granule, and it might be larger than a translation granule.

I_{HPJFJ} The size of the poison granule is IMPLEMENTATION DEFINED and might vary from moment to moment, instruction to instruction, and by Location. The size of the poison granule might be the same as the size of the protection granule, but this is not guaranteed. See also R_{WLTPV}.

I_{CBCMK} The regions of physical memory that support the poison-atomic property are IMPLEMENTATION DEFINED. Systems might have multiple physical memory types, including multiple physical memory regions used for bulk memory.

D20.2.3.2 Removing poison

I_{VZRKL} For guidance on mechanisms that software can use to remove poison from a Location, see Use of the RAS Extension.

I_{PSLGW} In order to test that poison has been removed from a Location, after the Location has been overwritten by a known-good value, software must be able to read back the Location from physical memory to check that the poison has been removed and the error is not persistent.

To do this, software must ensure that the Location has been cleaned and invalidated from all caches, so that the read returns a value from the Point of Physical Storage for the Location, and not any intermediate cache.

The Point of Coherency might be before a transparent unified cache, meaning that it might be before the Point of Physical Storage for a Location.

For more information on the Point of Physical Storage, see A64 Cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.2.4 Other errors

I_{KRQMR} The RAS Extension deals mostly with errors detected by components outside of the PE, such as memory, and consumed by the PE.

Other errors might be detected from within the processor that implements the PE. If the error is not an error in the PE state then the error might be treated as an error detected by another component.

In the following examples, the *component* reports these errors to a RAS System Architecture node that implements Error records and records the errors, and might generate one or more of a Fault handling interrupt, Error recovery interrupt, or Critical Error interrupt depending on the features and configuration of the node.

Example D20-9

A processor cache can be treated as a component outside the PE.

The cache detects an error in the cache state that cannot be corrected:

- If the error is detected in dirty cache data being evicted from the cache when the PE makes an access, then the error might be deferred by the cache writing poison in the evicted cache data.
- If the PE is performing a partial write that does not completely overwrite the protection granule, then the error might be deferred by the cache writing poisoned to the cache location, and/or evicting the cache line with poison. Deferring the error means the error is not consumed by the PE.

Otherwise, the cache component generates the in-band error response to the PE.

Example D20-10

A processor memory interface can be treated as a component outside the PE.

A processor detects a corrupt or poisoned value being returned on the interface that is not being signaled as an in-band error response and cannot be corrected or deferred. For example, in response to a non-cacheable read or a cache refill.

The memory interface component generates the in-band error response to the PE.

I_{VNTWD} An implementation might include error detection logic within the PE state itself. When the PE detects an error in the PE state, the instruction that uses that state consumes the error, and the PE generates an IMPLEMENTATION DEFINED Error exception, taken as an SError exception. See R_{FNVVJ}.

In this case, the processor that implements the PE includes a RAS System Architecture node that implements Error records that record these errors.

I_{JRQDM} An implementation might support poisoning within the PE state. When the PE consumes a Deferred error, for example a poisoned value, from memory into the PE state, the PE state becomes poisoned. Subsequent operations that read the poisoned value can continue to defer the error by poisoning the result of the operation.

However, if the PE attempts to execute an operation that reads the poisoned value and cannot defer the error further, the PE generates an IMPLEMENTATION DEFINED Error exception, taken as an SError exception. See R_{FNVVJ}.

In this case, the processor that implements the PE includes a RAS System Architecture node that implements Error records that record these errors.

I_{JHQVK} Components outside of the PE might detect errors that are not consumed by the PE. These components might report such errors to a PE using Error recovery interrupts.

R_{XJNNT} For implementations that include the Statistical Profiling Extension, the Statistical Profiling Extension behaves like a separate component.

I_{MJQQZ} Errors from software faults are outside the scope of the RAS Extension. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.3 Generating error exceptions

R_{VJRKF} An Error exception is generated when a detected error is signaled to the PE as an in-band error response to an architecturally-executed memory access or cache maintenance operation. This includes any explicit data access, instruction fetch, translation table walk, or hardware update to the translation tables made by an architecturally-executed instruction.

I_{PJHZS} An Error exception is taken as an asynchronous SError exception, a synchronous External Data Abort exception, or a synchronous External Instruction Abort exception.

R_{MBNBH} It is IMPLEMENTATION DEFINED whether an Error exception can be generated for an error that is consumed by hardware speculation or prefetching by a PE, but that is not committed to the architecturally visible state of the PE.

R_{SHKJB} It is IMPLEMENTATION DEFINED whether an Error exception can be generated for a detected error that is deferred.

R_{GVWJD} It is IMPLEMENTATION DEFINED whether an Error exception can be generated for a detected error that is corrected.

R_{FNVVJ} An Error exception can also be generated for IMPLEMENTATION DEFINED causes. An Error exception generated for an IMPLEMENTATION DEFINED cause is taken as an SError exception.

Example D20-11

An error is detected and neither corrected nor deferred to the PE, and signaled to the PE by means other than an in-band error response, such as a wired SError exception pin. Asserting the SError exception pin causes the PE to generate an SError exception.

Example D20-12

An error is detected by the PE in the PE state, or in the result of a calculation performed by the PE. The detected error generates an SError exception.

I_{ZXHDP} See also:

- Exceptions.
- Taking error exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.4 Taking error exceptions

R_{VXFYS} For an error signaled on a non-speculative instruction fetch or a translation table walk or hardware update of translation tables for an instruction fetch:

- If FEAT_DoubleFault is implemented, then the Error exception is taken as a synchronous Instruction Abort exception.
- Otherwise, it is IMPLEMENTATION DEFINED whether an Error exception on these accesses is taken as a synchronous External Abort exception or as an asynchronous SError exception.

R_{HWVPC} Other than for the IMPLEMENTATION SPECIFIC cases described below, if FEAT_ANERR is implemented and the Effective value of SCTLR2_ELx.EnANERR is `0b0`, an error signaled on a load from Normal memory is handled precisely and synchronously.

It is IMPLEMENTATION SPECIFIC whether SCTLR2_ELx.EnANERR applies to memory reads generated by each of the following:

- If FEAT_LSE is implemented, `LD<OP>`, `SWP`, and `CAS{P}` atomic instructions that return a value to the PE.
- If FEAT_LS64_V is implemented, `LD64B` and `ST64BV{0}` instructions that return a value to the PE.
- If FEAT_SME is implemented:
  - SIMD&FP register loads when the PE is in Streaming SVE mode.
  - SVE register loads when the PE is in Streaming SVE mode.
  - SME register loads.
- If FEAT_THE is implemented, `RCW*` instructions that return a value to the PE.
- If FEAT_NV2 is implemented, a memory read that is generated as a result of HCR_EL2.NV2 transforming an `MRS` or `MRRS` instruction to a load, and the target memory type is not Normal Inner and Outer Write-back cacheable.

This applies to exceptions taken from both AArch64 and AArch32 Execution states.

R_{SYQGN} Other than for the IMPLEMENTATION SPECIFIC cases described below, if FEAT_ADERR is implemented and the Effective value of SCTLR2_ELx.EnADERR is `0b0`, an error signaled on a load from Device memory is handled precisely and synchronously.

It is IMPLEMENTATION SPECIFIC whether SCTLR2_ELx.EnADERR applies to memory reads generated by each of the following:

- SIMD&FP register loads.
- If FEAT_LSE is implemented, `LD<OP>`, `SWP`, and `CAS{P}` atomic instructions that return a value to the PE.
- IF FEAT_SVE is implemented, SVE register loads.
- If FEAT_LS64_V is implemented, `LD64B` and `ST64BV{0}` instructions that return a value to the PE.
- If FEAT_SME is implemented, SME register loads.
- If FEAT_THE is implemented, `RCW*` instructions that return a value to the PE.
- If FEAT_NV2 is implemented, a memory read that is generated as a result of HCR_EL2.NV2 transforming an `MRS` or `MRRS` instruction to a load.

This applies to exceptions taken from both AArch64 and AArch32 Execution states.

I_{YRSKT} For an error signaled on a fetch of GPT information to check an access, the Error exception generates a synchronous External abort on GPT fetch, which is taken as a GPC exception. For more information see GPC faults.

R_{VLFTP} Unless otherwise stated, it is IMPLEMENTATION DEFINED whether an Error exception is handled precisely and synchronously, or handled asynchronously for each non-speculative:

- Explicit memory access made by an instruction.
- Cache maintenance operation.
- If FEAT_MTE is implemented, access to an Allocation Tag in memory made by an instruction.
- Translation table walk or hardware update of translation tables for a memory access made by an instruction.

D_{TSYLW} In this section, *Handled precisely and synchronously* means:

- If an access is due to any Active element of an SVE non-fault vector load instruction, or an Active element that is not the First active element of an SVE First-fault vector load instruction, then the error is reported in the FFR.
- Otherwise, the error generates a precise synchronous Data Abort exception.

D_{NBSFD} In this section, *Handled asynchronously* means:

- If an access is due to any Active element of an SVE non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then the error is not reported in the FFR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The error generates an asynchronous SError exception.

R_{CLFJM} All Error exceptions generated by speculative memory accesses are taken as an asynchronous SError exception.

R_{WFNJG} When an Error exception is taken as an asynchronous SError exception, the exception is taken in finite time.

R_{BCXKN} When any of the following exceptions are taken, the PE records the PE error state in the exception syndrome register:

- A synchronous External abort taken to AArch64 state.
- An SError exception taken to either AArch32 or AArch64 state.

R_{TYVYR} When a synchronous External abort is taken to AArch32 state, the PE does not record the PE error state.

I_{GGCQQ} The exception type and target Execution state determine the set of PE error state values the PE can record.

I_{WSYXB} The recorded PE error state informs software whether software can recover execution and, if so, whether any action by the recovery software to locate and repair the error is necessary first.

I_{TRWXF} Other than as described by I_{WBVYC}, the PE error state recorded in the exception syndrome register describes the recovery of the PE state only. For example, the PE state might be recoverable when the state of system is such that system-level recovery is not possible. See also I_{ZQRGL}.

I_{NFDSM} Software is only able to successfully recover execution and make progress from a restart address for the exception by executing an Exception Return instruction to branch to the instruction at this restart address if all of the following are true:

- The error has not been silently propagated by the PE.
- At the point when the Exception Return instruction is executed, the PE state and memory system state are consistent with the PE having executed all of the instructions up to but not including the instruction at the restart address, and none afterwards. That is, at least one of the following restart conditions is true:
  - The error has been not architecturally consumed by the PE and infected the PE state.
  - Executing the instruction at the restart address will not consume the error and will correct any corrupt state by overwriting it with the correct value or values.

R_{DCKHJ} On taking an Error exception, the PE determines that software is able to recover execution at the point where the exception is taken, with no additional action from software, if and only if all of the following are true:

- The error has not been silently propagated by the PE.
- The restart conditions are met because all of the following are true:
  - Either the error does not remain latent or executing the instruction at the restart address will not consume the error and will correct any corrupt PE state.
  - The restart address is the preferred return address for the exception.
- The PE has not elected to determine that software is not able to recover execution, and has not elected to determine that software is able to recover execution if software takes action to locate and repair the error.

R_{JBHWY} On taking an Error exception, the PE determines that software is able to recover execution if software takes action to locate and repair the error, to get the PE state and memory system state into this consistent state before attempting recovery, if and only if all the following are true:

- The error has not been silently propagated by the PE.
- The restart conditions can be met because the restart address is the preferred return address for the exception and at least one of the following is true:
  - The error remains latent and executing the instruction at the restart address will access the corrupt state. If the error is removed then executing the instruction at the restart address will correct any corrupt PE state and/or corrupt memory state. For example, the instruction at the restart address is a load that will consume the error and corrupts PE state.
  - The error does not remain latent and the PE has elected to determine that software is able to recover execution if software takes action to locate and repair the error.
  - Executing the instruction at the restart address will not consume the error and the PE has elected to determine that software is able to recover execution if software takes action to locate and repair the error.
- The PE has not elected to determine that software is not able to recover execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{GJQWN} On taking an Error exception, the PE determines that software is not able to recover execution if and only if one or more of the following are true:

- The error has been silently propagated by the PE.
- The restart conditions cannot be met even if software takes action to locate and repair the error. This is because at least one of the following is true:
  - The error remains latent and executing the instruction at the restart address will consume the error and corrupt PE state. Either the error cannot be removed or executing the instruction at the restart address will not correct any corrupt PE state.
  - The restart address is not the preferred return address for the exception.
- The PE has elected to determine that software is not able to recover execution.

I_{XMCCR} That the PE determines that software is able to recover execution if software takes action to locate and repair the error does not mean that software can locate and repair the error. For example, the error in memory might be one which cannot be located or cannot be repaired. See I_{ZQRGL}.

Error recovery software might instead make the PE state and memory system state consistent with an *alternative execution* of the program.

Example D20-13

An Error exception is generated by a load from a location in a clean page of memory that is infected by an error. Software might be able to repair the error by:

- Reloading the page from a backing store. This makes the memory system state consistent with the uncorrupted view. Executing the instruction at the restart address will load the uncorrupted value into the PE state.
- Invalidating the clean page and marking it page as inaccessible. Executing the instruction at the restart address will result in a Translation fault being generated when the program tries to access the page. The target of the load will contain an UNKNOWN value, which is permitted by the architecture. The MMU fault handler can then reload the page from the backing store, as it would for a page that has not been previously accessed or has been paged out.

Either approach might result in the virtual address to physical address mapping for the page being changed by software, meaning the memory system state is not consistent with the previously executed instructions. However, the memory system state is consistent with a valid alternative view of the execution of the program that allows software to recover execution.

This recovery is only possible if the error can be isolated to a location.

I_{RHPPV} A PE might include additional IMPLEMENTATION DEFINED mechanisms to aid software locate and repair the error.

If software has to use IMPLEMENTATION DEFINED mechanisms to locate and repair the error, then the PE reports that it has determined that software is not able to recover execution. The PE might use IMPLEMENTATION DEFINED additional syndrome registers to report that software is able to recover execution if software takes action to locate and repair the error using the IMPLEMENTATION DEFINED mechanisms.

I_{RLQVJ} See also:

- PE error state recording in the exception syndrome.
- PE error state classification.

D20.4.1 PE error state recording in the exception syndrome

R_{WPKYM} When an asynchronous SError exception is taken to AArch64 state, the PE records the PE error state in the ESR_ELx exception syndrome register as the applicable one of:

- Uncontainable (UC).
- Unrecoverable state (UEU).
- Recoverable state (UER).
- Restartable state (UEO).
- Corrected (CE).
- Uncategorized error.
- IMPLEMENTATION DEFINED syndrome.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{SDDLL} When an asynchronous SError exception is taken to AArch64 state:

- Uncategorized error is recorded by setting ESR_ELx.ISS to zero. This includes setting ESR_ELx.IDS and ESR_ELx.DFSC to zero.
- IMPLEMENTATION DEFINED syndrome is recorded by setting ESR_ELx.IDS to `0b1`. The remainder of the ESR_ELx.ISS syndrome is IMPLEMENTATION DEFINED.

Other values for the PE error state are recorded in ESR_ELx.AET, by setting ESR_ELx.IDS to `0b0` and ESR_ELx.DFSC to the applicable nonzero fault status code, indicating ESR_ELx.AET is valid.

R_{ZSKKG} If FEAT_RASv2 is implemented and any of the following are true, then an error exception is taken as an SError exception:

- The error has been silently propagated by the PE.
- The PE determines that software is not able to recover execution from the preferred return address of the exception.

R_{FKHHF} When a synchronous External abort exception is taken to AArch64 state, the PE records the PE error state in ESR_ELx.SET as the applicable one of the following:

- If FEAT_RASv2 is not implemented, Uncontainable (UC).
- Recoverable state (UER).
- Restartable state (UEO).

Other values for the PE error state are not supported by synchronous External abort exceptions taken to AArch64 state.

R_{PWKBL} When an asynchronous SError exception is taken to AArch32 state, the PE records the PE error state in DFSR.AET or HSR.AET as appropriate, as the applicable one of:

- Uncontainable (UC).
- Unrecoverable state (UEU).
- Recoverable state (UER).
- Restartable state (UEO).

Other values for the PE error state are not supported by asynchronous SError exceptions taken to AArch32 state.

I_{QVVSM} Table D20-1 summarizes the supported PE error state syndrome values for each type of Error exception.

Table D20-1 Summary of Error exception types and supported PE error state syndrome values

PE error state	External abort to AArch64 state	SError exception to AArch64 state	External abort to AArch32 state	SError exception to AArch32 state
Recorded in:	ESR_ELx.SET	ESR_ELx.AET	No syndrome	DFSR.AET
Uncategorized error	No	Yes (ISS==0)	-	No
IMPLEMENTATION DEFINED syndrome	No	Yes (IDS==1)	-	No
Uncontainable (UC), FEAT_RASv2 implemented	No	Yes (`0b000`)	-	Yes (`0b00`)
Uncontainable (UC), FEAT_RASv2 not implemented	Yes (`0b10`)	Yes (`0b000`)	-	Yes (`0b00`)
Unrecoverable state (UEU)	No	Yes (`0b001`)	-	Yes (`0b01`)
Recoverable state (UER)	Yes (`0b00`)	Yes (`0b011`)	-	Yes (`0b11`)
Restartable state (UEO)	Yes (`0b11`)	Yes (`0b010`)	-	Yes (`0b10`)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PE error state	External abort to AArch64 state	SError exception to AArch64 state	External abort to AArch32 state	SError exception to AArch32 state
Deferred (DE)	No	No	-	No
Corrected (CE)	No	Yes (`0b110`)	-	No

D20.4.2 PE error state classification

I_{CCKWK} The PE determines which PE error state to record based on the following criteria:

- The PE error state syndrome values supported by the type of Error exception being taken. See PE error state recording in the exception syndrome.
- The following IMPLEMENTATION SPECIFIC properties and behaviors of the PE on taking the exception:
  - Whether the error has been silently propagated by the PE.
  - Whether the PE determines that software is able to recover execution at the point where the exception is taken.
  - If the PE determines that software can recover execution, whether software needs locate and repair the error before attempting to recover. If software does not locate and repair the error, then attempting to recover execution might cause the Error exception to be generated again.
  - If the PE determines that software cannot recover execution, whether the error is a synchronizable error.
- Whether the implementation elects to record the PE error state as another state. The PE only does this when the criteria for the other, recorded state are met. The conditions under which the PE elects to record the PE error state as another state are IMPLEMENTATION DEFINED.

The recorded PE error state is defined by the rules in this section.

R_{QKZLB} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Uncontainable (UC):

- One or more of the following are true:
  - The error has been silently propagated by the PE.
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the error is not a synchronizable error.
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the Error exception is taken as a synchronous External abort to AArch64 state. (That is, the type of Error exception does not support reporting the PE error state as Unrecoverable state (UEU).)
  - The implementation has elected to record the PE error state as Uncontainable (UC).
- The Error exception is not taken as a synchronous External abort to AArch32 state.
- Either FEAT_RASv2 is not implemented, or the Error exception is not taken as a synchronous External abort to AArch64 state.
- The implementation has not elected to record the PE error state as IMPLEMENTATION DEFINED syndrome or Uncategorized error, or the type of Error exception does not support reporting the PE error state as IMPLEMENTATION DEFINED syndrome or Uncategorized error.

R_{QGNYD} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Unrecoverable state (UEU):

- The error has not been silently propagated by the PE.
- The Error exception is taken as an SError exception.
- One or more of the following are true:
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the error is a synchronizable error.
  - The implementation has elected to record the PE error state as Unrecoverable state (UEU).
- The implementation has not elected to record the PE error state as Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

I_{FJCZP} Error synchronization event defines synchronizable error.

R_{JHNVT} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Recoverable state (UER):

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The error has not been silently propagated by the PE.
- The Error exception is not taken as a synchronous External abort to AArch32 state.
- The PE determines that software is able to recover execution from the preferred return address of the exception.
- One or more of the following are true:
  - The PE determines that software must take action to locate and repair the error to successfully recover execution. This might be because the exception was taken before the error was architecturally consumed by the PE, at the point when the PE was not be able to make correct progress without either consuming the error or otherwise making the state of the PE unrecoverable.
  - The implementation has elected to record the PE error state as Recoverable state (UER).
- The implementation has not elected to record the PE error state as Unrecoverable state (UEU), Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

R_{MBVCF} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Restartable state (UEO):

- The error has not been silently propagated by the PE.
- The Error exception is not taken as a synchronous External abort to AArch32 state.
- The PE determines that software can recover execution from the preferred return address of the exception without the need for software to take action to locate and repair the error first.
- One or more of the following are true:
  - The error is an uncorrected error. This includes a deferred error.
  - The error is a corrected error and the Error exception is not taken as an SError exception taken to AArch64 state.
  - The implementation has elected to record the PE error state as Restartable state (UEO).
- The implementation has not elected to record the PE error state as any of Recoverable state (UER), Unrecoverable state (UEU), Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

R_{LFXRD} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Corrected (CE):

- The error has been corrected and not silently propagated by the PE.
- The Error exception is taken as an SError exception taken to AArch64 state.
- Software can recover execution from the preferred return address of the exception. Because the error has been corrected, software does not need to take action to locate and repair the error.
- The implementation has not elected to record the PE error state as any other type.

I_{BWPJZ} Arm strongly recommends that a corrected error that is not silently propagated does not generate any Error exception at the PE. The producer of the data that corrected the error might still record the error in a RAS error record, and generate a fault handling interrupt, as described in the *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

R_{NZYRP} If and only if all the following are true, then on taking an Error exception the PE error state is recorded as an Uncategorized error:

- The Error exception is taken as an asynchronous SError exception taken to AArch64 state.
- The implementation has elected to record the PE error state as an Uncategorized error.

R_{VHWHD} If and only if all the following are true, then on taking an Error exception the PE error state is recorded as an IMPLEMENTATION DEFINED syndrome:

- The Error exception is taken as an asynchronous SError exception taken to AArch64 state.
- The implementation has elected to record the PE error state as an IMPLEMENTATION DEFINED syndrome.

I_{SRPJD} The IMPLEMENTATION DEFINED syndrome type might provide additional IMPLEMENTATION DEFINED syndrome recorded in the exception syndrome register. Software might be able to determine the state of the PE from this syndrome, or other IMPLEMENTATION DEFINED syndrome registers.

I_{WLZRP} Uncategorized error and IMPLEMENTATION DEFINED syndrome are defined for backward compatibility with previous versions of the architecture. Arm does not recommend use of these PE error state values in new implementations that include other RAS features.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IVKMZB The PE error states for an Error exception generated when the PE consumes an uncorrected error are summarized by Figure D20-1. Figure D20-1 assumes the type of Error exception supports the resulting PE error state, that the error was neither corrected nor deferred, and the PE never elects to record an error as a different PE error state when permitted, an Uncategorized error syndrome, or an IMPLEMENTATION DEFINED syndrome.

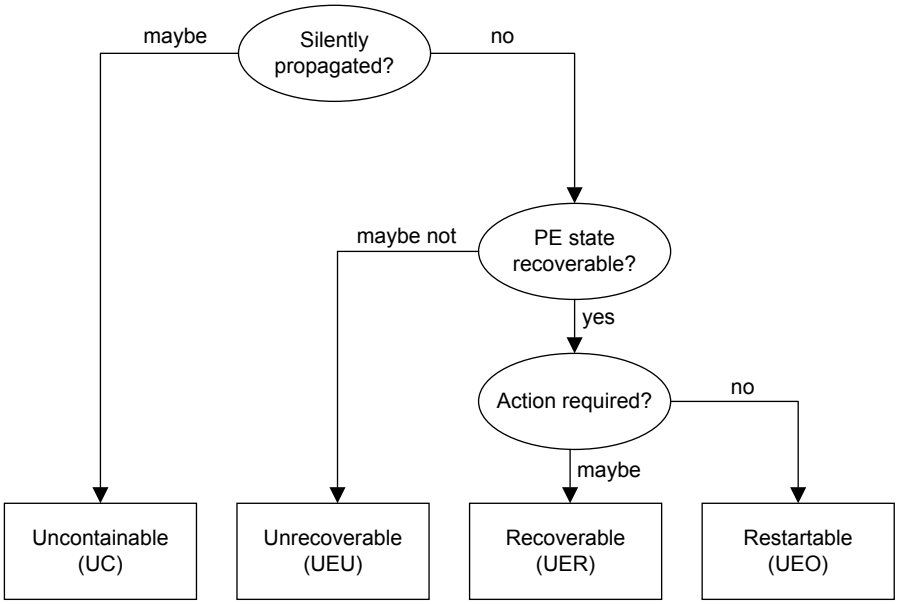

Figure D20-1 PE error states

IZQRGL If the PE error state reports that software can recover execution, or that software isolation might be possible because the error is a synchronizable error, then this does not necessarily mean that the error can be recovered from because the error in the system might be one which does not allow software to recover the operation. Rather, software *might* be able to recover if it can repair the error and continue.

Example D20-14

A component detects an error when accessed by a PE, and records in a RAS System Architecture node that the error is uncontainable at the component, meaning the system has to be shut down to avoid catastrophic failure.

The component signals the error with an in-band error response to a PE, which does not report the severity of the error. The recorded PE error state refers only to the PE, not the system error state.

IWBVYC If the in-band error response can signal the severity of the error to the PE, then the PE might use this information to elect to report the PE error state as the severity of error reported to it, if permitted by the preceding rules.

However, this is not required, and software must not rely on this behavior and should determine from the system whether the error is recoverable at the system level.

Example D20-15

A processor cache detects an uncontainable tag RAM error, and the PE reports the PE error state as Uncontainable (UC), even though the state of the PE itself is Recoverable state (UER). RQKZLB allows this.

ITXFMY An SError exception that is masked when it is generated remains pending until taken later when no longer masked.

In these cases, the PE error state is also unlikely to be *Restartable* or *Recoverable*, and if FEAT_RASv2 is implemented, is reported with ESR_ELx.ELS set to '`0b0`'.

ITGPQZ See also:

- PE error propagation.
- Error synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.4.2.1 Using the PE error state classification

S_{XSKNS} When the PE error state is recorded as Uncontainable (UC):

- The error handling software must assume that either:
  - The error has been silently propagated by the PE.
  - Software is not able to recover execution from the preferred return address of the exception and the error was not a synchronizable error.
- If the error handling software cannot isolate the error to an application or VM, or both, by other means, then the system must be shut down by software to avoid catastrophic failure.

S_{HYWFL} When the PE error state is recorded as Unrecoverable state (UEU):

- The error handling software can assume the error has not been silently propagated by the PE.
- The error handler cannot safely recover execution from the preferred return address of the exception, even if it takes action to locate and repair the error. The state of the affected software, or both, is unrecoverable. However, if the software includes Error synchronization events, the error handler can use the properties of the Error synchronization event to determine which software is affected by the error.
- The affected software cannot continue and must be isolated by the error handling software.

S_{LSFYM} When the PE error state is recorded as Recoverable state (UER):

- The uncorrected error might remain latent in the system.
- If the error handling software takes action to locate and repair the uncorrected error, then the error handler can safely recover execution from the preferred return address of the exception. Otherwise on restart of the affected software the PE might attempt to consume the error again, causing a further Error exception. If the error handler cannot locate and repair the error, then the affected software must be isolated by the error handling software.

S_{GLPZY} When the PE error state is recorded as Restartable state (UEO):

- The error might remain latent in the system.
- The error handling software might take action to locate and repair the error before it is consumed. However, the affected software can be safely restarted by the error handler without software taking any action to locate and repair the error.

For example, the error was signaled when the PE speculatively accessed corrupt data.

S_{GRZQS} When the PE error state is recorded as IMPLEMENTATION DEFINED syndrome or Uncategorized error, if the error handling software is not able to determine the actual state of the PE and memory, it should treat IMPLEMENTATION DEFINED syndrome and Uncategorized error as Uncontainable (UC).

D20.4.2.2 Recording the physical address

D_{LVGRB} When FEAT_PFAR is implemented, the address written to PFAR_ELx.PA or MFAR_EL3.PA on taking a synchronous External Abort or SError exception can be any address within a naturally-aligned fault granule that contains the faulting physical address.

R_{THPSG} The fault granule size is IMPLEMENTATION DEFINED and no larger than the size of the range of values permitted to be recorded in FAR_ELx.

Note

To aid with data recovery, it is recommended that the fault granule is no larger than the error protection granule used in the system.

R_{JZCNK} When FEAT_PFAR is implemented, Execution at EL0 makes PFAR_EL1 become UNKNOWN, and Execution at EL1 or EL0 makes PFAR_EL2 become UNKNOWN.

D20.4.3 Multiple SError exceptions

I_{CPJLW} Multiple physical and/or virtual SError exception conditions might be pending together. The architecture does not define relative priorities for asynchronous exceptions.

R_{DHKQZ} If multiple physical and/or virtual SError exception conditions are pending, then it is IMPLEMENTATION DEFINED whether the multiple pending SError exception conditions are taken as a single SError exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{JBQSC} On taking an SError exception for more than one SError condition:

- If the exception is taken to AArch64 state and one or more pending SError conditions would be reported as IMPLEMENTATION DEFINED syndrome or Uncategorized error, then the syndrome recorded in ESR_ELx.ISS is IMPLEMENTATION DEFINED.
- Otherwise, the recorded PE error state applies recorded by combined effect of the errors.

I_{GNHXJ} Any pending SError conditions that are not taken with other SError exceptions as a single SError exception remains pending after the SError exception is taken.

D20.4.4 Target Exception level for External abort and SError exceptions taken to AArch64 state

D_{FWSVH} This section is included for completeness. It repeats the definitions from this Manual and so is *non-normative*.

These definitions also apply to synchronous External abort exceptions and SError exceptions taken from AArch32 state to AArch64 state.

I_{NRZXZ} The default target Exception level for synchronous External abort exceptions taken to AArch64 state is:

- EL1, if taken from EL0 or EL1.
- EL2, if taken from EL2.
- EL3, if taken from EL3.

The default target Exception level for SError exceptions taken to AArch64 state is EL1.

However:

- If EL3 is implemented and SCR_EL3.EA is `0b1`, then the target Exception level for all SError exceptions and synchronous External abort exceptions is EL3.
- Otherwise, if EL2 is implemented and enabled in the current Security state, then:
  - If HCR_EL2.AMO is `0b1` or HCR_EL2.TGE is `0b1`, then the target Exception level for SError exceptions is EL2.
  - If HCR_EL2.TEA is `0b1` or HCR_EL2.TGE is `0b1`, then the target Exception level for synchronous External abort exceptions taken from EL0 and EL1 is EL2.

When executing in AArch64 state at a higher Exception level than the target Exception level for SError exceptions, SError exceptions are implicitly masked and not taken.

I_{BPGXV} See also The AArch64 System Level Programmers' Model.

D20.4.5 Target Exception level for External abort and SError exceptions taken to AArch32 state

D_{WFQWP} This section is included for completeness. It repeats the definitions from this Manual and so is *non-normative*.

I_{BMBXM} The default target mode for SError and synchronous External abort exceptions taken to AArch32 state is:

- Abort mode, if taken from EL0, EL1 or EL3, including from Monitor mode.
- Hyp mode, if taken from EL2.

However:

- If EL3 is implemented, EL3 is using AArch32, and SCR.EA is `0b1`, then the target mode for SError exceptions and synchronous External abort exceptions is Monitor mode.
- Otherwise, if EL2 is implemented, EL2 is using AArch32, and the PE is in Non-secure state:
  - If HCR.AMO is `0b1` or HCR.TGE is `0b1`, then the target mode for SError exceptions taken from EL0 and EL1 is Hyp mode, using vector offset `0x14`.
  - If HCR.TEA is `0b1` or HCR.TGE is `0b1`, then the target mode for synchronous External abort exceptions taken from EL0 and EL1 is Hyp mode, using vector offset `0x14`.

Unless otherwise stated, vector offset `0x10` is used for SError exceptions and synchronous Data Abort exceptions, and vector offset `0x0C` is used for Prefetch Abort exceptions.

I_{CCPMY} See also The AArch32 System Level Programmers' Model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.5 Error synchronization event

R_{GRJVN} An Error synchronization event is generated by any of the following:

- Executing an ESB instruction.
- When FEAT_IESB is implemented, and one of the following is true, taking an exception in Non-debug state to an Exception level, ELx, using AArch64:
  - The Effective value of the applicable SCTLR_ELx.IESB bit is 0b1.
  - FEAT_DoubleFault is implemented, the Exception level is EL3, and SCR_EL3.NMEA is 0b1.
- When FEAT_IESB is implemented, and one of the following is true, executing an exception return instruction in Non-debug state at an Exception level, ELx, using AArch64:
  - The appropriate SCTLR_ELx.IESB bit is 0b1.
  - FEAT_DoubleFault is implemented, the exception level is EL3, and SCR_EL3.NMEA is 0b1.

R_{RKWPZ} SCTLR_ELx.IESB is ignored in Debug state and treated as zero.

I_{XKTLD} Because SError exceptions are masked in Debug state, and the effects of an implicit Error synchronization barrier could happen at any time regardless of the value of SCTLR_ELx.IESB, an implementation that honors SCTLR_ELx.IESB in Debug state is compliant with R_{RKWPZ}.

I_{PMGVM} In addition to generating an Error synchronization event, the ESB instruction might additionally record and then clear a masked pending asynchronous SError exception. This is also referred to as deferring the pending asynchronous SError exception.

R_{WPPWC} A synchronizable error is one of a subset of errors specific to the implementation that are synchronized by Error synchronization events, generated by an instruction executed on the same PE as the Error synchronization events.

Note: Synchronizable error must not be confused with a synchronous error exception. An Error exception for a synchronizable error might be taken asynchronously.

R_{BZPTV} If the Error exception for a synchronizable error is taken in program order after the Error synchronization event completes, and either physical SError exceptions are unmasked when the Error synchronization event occurs or the Error exception is taken synchronously, then all of the following are true:

- The instruction that generated the synchronizable error is in program order after the Error synchronization event.
- On completion of the Error synchronization event, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the Error synchronization event.

R_{ZRHDD} If the Error exception for a synchronizable error is taken asynchronously as an SError exception, physical SError exceptions are masked when the Error synchronization event occurs, and the SError exception is not pending when the Error synchronization event completes, then all of the following are true:

- The instruction that generated the synchronizable error is in program order after the Error synchronization event.
- On completion of the Error synchronization event, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the Error synchronization event.

The SError exception is not pending when the Error synchronization event completes if a subsequent read of ISR_EL1.A or ISR.A returns 0b0.

R_{YCJTN} If the Error exception for a synchronizable error is taken asynchronously as an SError exception, the Error synchronization event is generated by an ESB instruction executed when physical SError exceptions are masked, and the ESB instruction does not set DISR_EL1.A or DISR.A to 0b1, then all of the following are true:

- The instruction that generated the synchronizable error is in program order after the ESB.
- On completion of the ESB, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the ESB.

R_{NFKMQ} Taken in program order after the Error synchronization event completes means:

- For an Error synchronization event generated by an ESB instruction, the exception is taken in program order after the instruction.
- For an Error synchronization event generated by an exception return instruction when FEAT_IESB implemented, the exception is taken in program order after the instruction.
- For an Error synchronization event generated by an exception entry when FEAT_IESB is implemented, one of the following is true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The exception is taken in program order strictly after the first instruction of the exception handler at the exception vector address.

— The exception is taken from the first instruction of the exception handler at the exception vector address and the ESR_ELx.IESB syndrome bit is recorded as `0b0`.

I_{QZSHG} The definition of synchronizable error means that if the error that is a synchronizable error is generated by an instruction in program order before the Error synchronization event, then either the Error exception is taken before the Error synchronization event, or on executing the Error synchronization event the following apply:

- If physical SError exceptions are unmasked or the Error exception is taken synchronously, then the Error synchronization event ensures that the Error exception is not taken in program order after the Error synchronization event. This allows isolation of the software affected by the error.
- If physical SError exceptions are masked and the Error exception is taken asynchronously, then:
  — If the Error synchronization event was generated by an `ESB`, then the error is recorded in DISR_EL1 or DISR. Software can use the PE error state recorded in DISR_EL1 or DISR to determine what recovery is possible.
  — Otherwise, the Error exception is pending when the Error synchronization event completes.

The SError exception might have been pending before or made pending by the Error synchronization event.

The definition does not mean that if the error is generated by a instruction in program order after the Error synchronization event, then the Error exception will only be taken after the Error synchronization event. The Error exception might be taken before the Error synchronization event, if the PE speculated past the Error synchronization event and speculatively executed the instruction that generated the error. This might cause software to generate a false failure. Error synchronization events are not speculation barriers.

I_{SQCFG} The criteria for the PE error state mean that if the PE reports the PE error state as one of the following, then the error must be either explicitly or implicitly a synchronizable error:

- Unrecoverable state (UEU).
- Recoverable state (UER).
- Restartable state (UEO).

This is because whether the error is a synchronizable error is a criterion for Unrecoverable state (UEU), and the criteria for Recoverable state (UER) and Restartable state (UEO) satisfy the definition of synchronizable error.

For other physical errors:

- An error that has been silently propagated by the PE and is not reported as either IMPLEMENTATION DEFINED syndrome or Uncategorized error must be reported as Uncontainable (UC) and is not containable even if the error is a synchronizable error. Software must assume the error has been silently propagated even if the error is a synchronizable error.
- It is IMPLEMENTATION SPECIFIC whether an error reported with an ESR_ELx.ISS syndrome that is IMPLEMENTATION DEFINED syndrome or Uncategorized error is a synchronizable error.
- The following errors have not been consumed by the PE:
  — A *Deferred error*.
  — A *Corrected error*.
  — An Error exception from a read by hardware speculation that does not corrupt the state of the PE.

  Software can recover execution from these errors regardless of whether the error is a synchronizable error.
- An implementation might have other IMPLEMENTATION DEFINED Error exception and other sources of SError exception, see R_{FNVVJ}. If an IMPLEMENTATION DEFINED SError exception is generated by a level-sensitive interrupt signal, then the SError exception cannot be a synchronizable error.

I_{VFFYW} An Error synchronization event might operate as follows:

1. The PE ensures that any synchronizable error generated by an instruction in program order before the Error synchronization event has caused a physical SError exception to become pending.
2. If a physical SError exception is pending for a synchronizable error and generated by an instruction in program order before the Error synchronization event, and physical SError exceptions are not masked at the current Exception level, then the physical SError exception is taken before completion of the Error synchronization event. The SError exception might have been made pending by the Error synchronization event, or might have been pending before the Error synchronization event.

R_{NPPGJ} If an SError exception for a synchronizable error is pending after completing the Error synchronization event generated by an `ESB` instruction, and physical SError exceptions are masked at the current Exception level, then the `ESB` instruction performs the following steps:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. The pending physical SError exception is recorded in DISR_EL1 or DISR. This includes the PE error state that the pending Error exception would record if taken.
2. The DISR_EL1.A bit or DISR.A bit is set to `0b1`.
3. The pending state of the physical SError exception is cleared.

The SError exception might have been made pending by the Error synchronization event, or might have been pending before the Error synchronization event.

R_{BLRTM} The criteria for `ESB` recording the PE error state in DISR_EL1 or DISR are the same as for that for recording the PE error state in ESR_ELx or DFSR when an SError exception taken to the current execution state.

R_{KNWBN} If an SError exception is taken as part of an Error synchronization event generated by an `ESB` instruction, then the `ESB` instruction address is the *preferred return address* of the exception.

For the definition of the *preferred return address* for an exception, see Preferred return address.

R_{SFHDS} On executing an `ESB` instruction when SError exceptions are masked, any pending SError exception generated by an error that is not a synchronizable error:

- Remains pending after completion of the Error synchronization event.
- Does not update DISR_EL1 or DISR.

I_{CQKXL} The Error recovery interrupt, Fault handling interrupt, and Critical Error interrupt interrupts described by RAS System Architecture are asynchronous interrupts, and so are not synchronizable errors.

I_{BBGXN} If multiple SError exception conditions are pending, then an Error synchronization event synchronizes all errors that are synchronizable errors.

S_{VFHGT} Software must be aware that an SError exception taken at an Error synchronization event or recorded in the DISR_EL1 or DISR register by an `ESB` instruction might have been generated by hardware speculation of an instruction in program order after the Error synchronization event.

D20.5.1 ESB and virtual SError exceptions

R_{LLLVR} If all of the following are true, then an `ESB` instruction executed at EL0 or EL1 synchronizes a pending virtual SError exception:

- EL2 is implemented and enabled in the current Security state.
- Any of the following are true:
  - EL2 is using AArch64, HCR_EL2.TGE is `0b0`, HCR_EL2.VSE is `0b1` and either:
    - HCR_EL2.AMO is `0b1`.
    - FEAT_DoubleFault2 is implemented and the Effective value of HCRX_EL2.TMEA is `0b1`.
  - EL2 is using AArch32, HCR.AMO is `0b1`, HCR.TGE is `0b0`, and HCR.VA is `0b1`.
- The VSESR_EL2 and, if implemented, VDFSR registers are writable.

In these cases, a virtual SError exception is pending, and the following occur when an `ESB` instruction is executed at EL0 or EL1:

- If the virtual SError exception is unmasked at the current Exception level, then the exception is taken before the completion of the `ESB` instruction.
- If the virtual SError exception is masked at the current Exception level, then all the following occur:
  - HCR_EL2.VSE or HCR.VA cleared to `0b0`.
  - The virtual SError exception syndrome from VSESR_EL2 or VDFSR is recorded in VDISR_EL2 or VDISR respectively.
  - VDISR_EL2.A or VDISR.A is set to `0b1` to indicate the virtual SError exception was pending prior to the execution of the `ESB` instruction.

R_{GXHYX} If all of the following are true, then it is IMPLEMENTATION DEFINED whether or not an `ESB` instruction executed at EL0 or EL1 synchronizes a pending virtual SError exception:

- EL2 is implemented and enabled in the current Security state.
- Any of the following are true:
  - EL2 is using AArch64, HCR_EL2.TGE is `0b0`, HCR_EL2.VSE is `0b1` and either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— HCR_EL2.AMO is 0b1.
— FEAT_DoubleFault2 is implemented and the Effective value of HCRX_EL2.TMEA is 0b1.
— EL2 is using AArch32, HCR.AMO is 0b1, HCR.TGE is 0b0, and HCR.VA is 0b1.

- VSESR_EL2 and, if implemented, VDFSR registers are implemented as RAZ/WI.

In these cases, a virtual SError exception is pending. If the IMPLEMENTATION DEFINED choice is that the ESB instruction synchronizes a pending virtual SError exception, then the following occur when an ESB instruction is executed at EL0 or EL1:

- If the virtual SError exception is unmasked at the current Exception level, then the exception is taken before the completion of the ESB instruction.
- If the virtual SError exception is masked at the current Exception level, then all the following occur:
  — HCR_EL2.VSE or HCR.VA cleared to 0b0.
  — The virtual SError exception syndrome in VDISR_EL2 or VDISR is set to zero.
  — VDISR_EL2.A or VDISR.A is set to 0b1 to indicate the virtual SError exception was pending prior to the execution of the ESB instruction.

If the IMPLEMENTATION DEFINED choice is that the ESB instruction does not synchronize a pending virtual SError exception, then an ESB instruction executed at EL0 or EL1 ignores the pending virtual SError exception and the virtual SError exception stays pending.

See also Syndrome registers.

I_{VKQGL} If both a virtual SError exception and a physical SError exception are pending as described in R_{LLLVR} and Error synchronization event, then the physical SError exception is not masked at the current Exception level, because this is not possible when a virtual SError exception is pending. This means that:

- If the virtual SError exception is masked at the current Exception level, then the physical SError exception will be taken before the ESB instruction completes and HCR_EL2.VSE, HCR.VA, VDISR_EL2, and VDISR are unchanged.
- If the virtual SError exception is not masked at the current Exception level, then it is IMPLEMENTATION DEFINED which of the virtual SError or physical SError exceptions is taken before the ESB instruction completes. If the virtual SError exception is taken and, after taking the exception to EL1, the physical SError exception is still pending, then the physical SError exception is still not masked and will be taken.

A virtual SError exception injected using HCR_EL2.VSE has priority over a delegated SError exception injected using SCR_EL3.DSE when both are synchronized by ESB.

D20.5.2 ESB and delegated SError exceptions

R_{KKPVY} If all of the following are true, then an ESB instruction executed at EL0, EL1, or EL2 synchronizes a pending delegated SError exception:

- FEAT_E3DSE is implemented.
- SCR_EL3.{EnDSE, DSE} is {0b1, 0b1}.
- VSESR_EL3 is writable.

In this case, a delegated SError exception is pending, and the following occur when an ESB instruction is executed at EL0, EL1, or EL2:

- If the delegated SError exception is unmasked at the current Exception level, then the exception is taken before the completion of the ESB instruction.
- If the delegated SError exception is masked at the current Exception level, then all the following occur:
  — SCR_EL3.DSE is cleared to 0b0.
  — The delegated SError exception syndrome from VSESR_EL3 is recorded in VDISR_EL3.
  — VDISR_EL3.A is set to 0b1 to indicate the delegated SError exception was pending prior to the execution of the ESB instruction.

R_{GGVCW} If all of the following are true, then it is IMPLEMENTATION DEFINED whether or not an ESB instruction executed at EL0, EL1, or EL2 synchronizes a pending delegated SError exception:

- FEAT_E3DSE is implemented.
- SCR_EL3.{EnDSE, DSE} is {0b1, 0b1}.
- The VSESR_EL3 register is implemented as RAZ/WI.

In this case, a delegated SError exception is pending. If the IMPLEMENTATION DEFINED choice is that the ESB instruction synchronizes a pending delegated SError exception, then the following occur when an ESB instruction is executed at EL0, EL1, or EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the delegated SError exception is unmasked at the current Exception level, then the exception is taken before the completion of the `ESB` instruction.
- If the delegated SError exception is masked at the current Exception level, then all the following occur:
  - SCR_EL3.DSE is cleared to `0b0`.
  - The delegated SError exception syndrome in VDISR_EL3 is set to zero.
  - VDISR_EL3.A is set to `0b1` to indicate the delegated SError exception was pending prior to the execution of the `ESB` instruction.

If the IMPLEMENTATION DEFINED choice is that the `ESB` instruction does not synchronize a pending delegated SError exception, then an `ESB` instruction executed at EL1 or EL2 ignores the pending delegated SError exception and the delegated SError exception stays pending.

See also Syndrome registers.

I_{VYGVB} If both a delegated SError exception and a physical SError exception are pending as described in R_{KKPVY} and Error synchronization event, then the physical SError exception is not masked at the current Exception level, because this is not possible when a delegated SError exception is pending. This means that:

- If the delegated SError exception is masked at the current Exception level, then the physical SError exception will be taken before the `ESB` instruction completes and SCR_EL3.DSE and VDISR_EL3 are unchanged.
- If the delegated SError exception is not masked at the current Exception level, then it is IMPLEMENTATION DEFINED which of the delegated SError or physical SError exceptions is taken before the `ESB` instruction completes. If the delegated SError exception is taken and, after taking the exception to EL1 or EL2, the physical SError exception is still pending, then the physical SError exception is still not masked and will be taken.

A virtual SError exception injected using HCR_EL2.VSE has priority over a delegated SError exception injected using SCR_EL3.DSE when both are synchronized by `ESB`.

D20.5.3 Extension for synchronization at exception entry and return

D_{DPSJR} The rules in this section apply when FEAT_IESB is implemented.

I_{WDSBL} An implicit Error synchronization event has no effect on DISR_EL1 or VDISR_EL2.

R_{KJWNS} When FEAT_DoubleFault is implemented, the PE is in Non-debug state, and the Effective value of SCR_EL3.NMEA is 1, then the Effective value of SCTLR_EL3.IESB is 1.

R_{HLVWK} When FEAT_DoubleFault2 is implemented and the PE is in Non-debug state, all of the following apply:

- If the Effective value of SCTLR2_EL1.NMEA is 1, then the Effective value of SCTLR_EL1.IESB is 1.
- If the Effective value of SCTLR2_EL2.NMEA is 1, then the Effective value of SCTLR_EL2.IESB is 1.

R_{RSFMF} When the PE is in Debug state, the Effective value of SCTLR_ELx.IESB is an IMPLEMENTATION SPECIFIC choice of 0 or 1.

D20.5.3.1 Synchronization on exception entry

R_{RNZWY} For each value of ELx in EL1, EL2, EL3, if all of the following are true, then each exception that is taken to ELx generates an Error synchronization event:

- ELx is using AArch64.
- The Effective value of SCTLR_ELx.IESB is `0b1`.

R_{JQSKQ} If an SError exception is taken to the Exception level ELy as a result of the Error synchronization event generated on exception entry by the FEAT_IESB mechanism, then all the following occur:

- The PE sets the ESR_ELy.IESB bit in the SError exception syndrome to `0b1`.
- The preferred return address for the SError exception is the exception vector address for the original exception.

Note: ELy might be the same Exception level as ELx.

I_{FWZHV} If SError exceptions are masked at ELx, then any SError exception made pending by the Error synchronization event stays pending.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{MMVJW} The prioritization of asynchronous exceptions is IMPLEMENTATION DEFINED. This means that an implementation might choose to behave as if the SError exception made pending by the Error synchronization event was taken before the implicit Error synchronization event, if the SError exception was not masked, taking the SError exception in place of the original exception, unless the original exception is also an SError exception.

In this case, ESR_ELy.IESB is set to `0b0` and the reported PE error state correctly indicates, for instance, whether software can recover execution from the preferred return address for the SError exception in ELR_ELy.

I_{BVBKS} Arm recommends that, when enabled, an implicit Error synchronization event is inserted before taking an exception other than an SError exception. In particular, Arm recommends that:

- When FEAT_DoubleFault is implemented, an implicit Error synchronization event is inserted before taking the exception to EL3.
- When FEAT_DoubleFault2 is implemented, an implicit Error synchronization event is inserted before taking the exception to the target Exception level of an SError exception.

D20.5.3.2 Synchronization on exception return

R_{SKRCR} For each value of ELx in EL1, EL2, EL3, if all of the following are true, then executing an exception return instruction at ELx generates an Error synchronization event:

- The instruction does not generate any exception.
- ELx is using AArch64.
- The Effective value of SCTLR_ELx.IESB is `0b1`.

I_{GPPXQ} On an illegal return event, the exception return instruction sets PSTATE.IL to `0b1`, which causes the next instruction to generate an Illegal State exception. The exception return instruction does not generate the exception.

R_{GXQYD} Any SError exception taken as part of the Error synchronization event terminates execution of the instruction.

R_{LPKVM} If an SError exception is taken to an Exception level, ELy, as a result of the Error synchronization event generated on exception return by the FEAT_IESB mechanism, then all the following occur:

- The PE sets the ESR_ELy.IESB bit in the SError exception syndrome to an IMPLEMENTATION DEFINED choice of `0b0` or `0b1`.
- The preferred return address for the SError exception is the address of the `ERET` instruction.

S_{MHCTH} R_{JQSKQ} and R_{LPKVM} describe how ESR_ELy.IESB records that an SError exception was taken because of an implicit Error synchronization event generated by the FEAT_IESB mechanism.

When ESR_ELy.IESB is 1, software can compare the value in ELR_ELy with the value of VBAR_ELx, where ELx is the Exception level the SError exception was taken from, to confirm whether this is due to the exception entry or the exception return case.

If the error is also reported as a synchronizable error, then:

- In the exception entry case:
  - ESR_ELy.IESB == 1 indicates that the error was generated by an instruction in program order before the exception was taken.
  - ELR_ELy will contain the address of an exception vector. See Exception vectors.
  - Software can determine the Exception level the exception was taken from the value recorded in SPSR_ELx, where ELx is the Exception level the SError was taken from.
- In the exception return case, software can ignore the value of ESR_ELy.IESB:
  - The value 1 indicates that the error was generated by an instruction in program order before the exception return.
  - However, this is indistinguishable from an SError taken before the exception return was executed. Hence the value of ESR_ELy.IESB is IMPLEMENTATION SPECIFIC in this case.

If ID_AA64MMFR4_EL1.EIESB indicates to software that implicit Error synchronization events are inserted before exceptions taken to ELy, then software can ignore the value in ESR_ELy.IESB, as it would only ever be set for the exception return case.

I_{JZHDB} If SError exceptions are masked at ELx, then any SError exception made pending by the Error synchronization event stays pending.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.5.4 Error synchronization barriers in a minimal implementation

I_{GQQCK} Error synchronization events and the ESB instruction can be implemented as no-ops if all of the following apply:

- Either there are no sources of SError exceptions, or all SError exceptions are reported as Uncategorized error and are not synchronizable errors.
- Either EL2 is not implemented, or VSESR_EL2 and VDFSR are implemented as RES0.

This allows for a very low cost implementation of the RAS Extension.

Such an implementation might include FEAT_IESB, meaning the SCTLR_ELx.IESB controls have no effect.

I_{HSHJG} See also Syndrome registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.6 Virtual SError exceptions and delegated SError exceptions

I_{LSSCN} EL2 provides a virtual SError exception.

For more information, see:

- Virtual asynchronous exceptions
- Virtual exceptions when an implementation includes EL2.

I_{YLSLH} When FEAT_E3DSE is implemented, EL3 provides delegated SError exceptions. For more information, see Delegated SErrors.

I_{ZXMPL} FEAT_RAS provides:

- Mechanisms to allow a hypervisor to specify the syndrome value reported to a guest Operating System on taking a virtual SError exception.
- Mechanisms to allow firmware to specify the syndrome value reported to a hypervisor or guest Operating System on taking a delegated SError exception.
- Support for EL0 or EL1 to isolate a virtual SError exception and for EL0, EL1, or EL2 to isolate a delegated SError exception as if it was a physical SError exception. See ESB and virtual SError exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D20.7 Error records in the PE

I_{KMMPK} A component that records detected errors is called a node by the RAS System Architecture. Each node implements one or more Error records.

R_{VNLPC} It is IMPLEMENTATION DEFINED whether the processor that implements a PE implements any nodes.

R_{XKDRX} A PE implementing the RAS Extension might implement the System register interface to nodes.

I_{SCVSB} The System register interface to nodes is not restricted to accessing only PE nodes.

I_{ZRKKQ} When an error is recorded by a PE node, one or more of the following might be generated, according to the configuration of the node:

- A Fault handling interrupt.
- An Error recovery interrupt.
- A Critical Error interrupt.
- An in-band error response.

D20.7.1 Error record System register view

I_{SLVDW} If the System register interface to a node is implemented, then software can access the Error records of the node using Error record System registers.

R_{BYLZQ} The number of Error records that can be accessed using the System registers is IMPLEMENTATION DEFINED, and might be zero. The ERRIDR_EL1 and ERRIDR registers indicate the highest numbered index of the Error records that can be accessed using System registers, plus one.

I_{NWBNQ} The AArch64 Error record System registers are those registers with an ERX*_EL1 mnemonic.
The AArch32 Error record System registers are those registers with an ERX* mnemonic.

I_{BJNDF} The following tables describe the AArch32 and AArch64 Error record System registers.
The descriptions in this section apply whether the Error record is accessed through the indirection mechanism, as described in Table D20-2 and Table D20-3, or as memory-mapped registers. For more information on the RAS memory-mapped registers, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

Table D20-2 Using AArch32 System registers, System register map

Use	To Access	Access	Description
ERXADDR	ERR<n>ADDR[31:0]	RW	Error Record <*n*> Address Register
ERXADDR2	ERR<n>ADDR[63:32]	RW	Error Record <*n*> Address Register
ERXCTLR	ERR<n>CTLR[31:0]	RW	Error Record <*n*> Control Register
ERXCTLR2	ERR<n>CTLR[63:32]	RW	Error Record <*n*> Control Register
ERXFR	ERR<n>FR[31:0]	RO	Error Record <*n*> Feature Register
ERXFR2	ERR<n>FR[63:32]	RO	Error Record <*n*> Feature Register
ERXMISC0	ERR<n>MISC0[31:0]	RW	Error Record <*n*> Miscellaneous Register 0
ERXMISC1	ERR<n>MISC0[63:32]	RW	Error Record <*n*> Miscellaneous Register 0
ERXMISC2	ERR<n>MISC1[31:0]	RW	Error Record <*n*> Miscellaneous Register 1
ERXMISC3	ERR<n>MISC1[63:32]	RW	Error Record <*n*> Miscellaneous Register 1
ERXMISC4[a]	ERR<n>MISC2[31:0]	RW	Error Record <*n*> Miscellaneous Register 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Use	To Access	Access	Description
ERXMISC5[a]	ERR<n>MISC2[63:32]	RW	Error Record <*n*> Miscellaneous Register 2
ERXMISC6[a]	ERR<n>MISC3[31:0]	RW	Error Record <*n*> Miscellaneous Register 3
ERXMISC7[a]	ERR<n>MISC3[63:32]	RW	Error Record <*n*> Miscellaneous Register 3
ERXSTATUS	ERR<n>STATUS	RW	Error Record <*n*> Primary Status Register

a. If FEAT_RASv1p1 is implemented.

Table D20-3 Using AArch64 System registers, System register map

Use	To Access	Access	Description
ERXADDR_EL1	ERR<n>ADDR	RW	Error Record <*n*> Address Register
ERXCTLR_EL1	ERR<n>CTLR	RW	Error Record <*n*> Control Register
ERXFR_EL1	ERR<n>FR	RO	Error Record <*n*> Feature Register
ERXMISC0_EL1	ERR<n>MISC0	RW	Error Record <*n*> Miscellaneous Register 0
ERXMISC1_EL1	ERR<n>MISC1	RW	Error Record <*n*> Miscellaneous Register 1
ERXMISC2_EL1[a]	ERR<n>MISC2	RW	Error Record <*n*> Miscellaneous Register 2
ERXMISC3_EL1[a]	ERR<n>MISC3	RW	Error Record <*n*> Miscellaneous Register 3
ERXPFGCDN_EL1	ERR<n>PFGCDN	RW	Error Record <*n*> Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	ERR<n>PFGCTL	RW	Error Record <*n*> Pseudo-fault Generation Control Register
ERXPFGF_EL1	ERR<n>PFGF	RO	Error Record <*n*> Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	ERR<n>STATUS	RW	Error Record <*n*> Primary Status Register
ERXGSR_EL1[b]	ERRGSR{<n>}	RO	Error Group Status Register

a. If FEAT_RASv1p1 is implemented.
b. If FEAT_RASv2 is implemented.

R_{ZBCFZ} If FEAT_RASv1p1 is implemented, then all Error records accessible through System registers implement RAS System Architecture version 1.1.

R_{MBSFP} If FEAT_RASv2 is implemented, then all Error records accessible through System registers implement RAS System Architecture version 2.

S_{VBBNY} To access an Error record, software:

1. Sets the error selection register, ERRSELR_EL1.SEL or ERRSELR.SEL, to the index of the record being accessed.
2. Accesses the Error record using the ERX*_EL1 or ERX* System registers, as described in I_{BJNDF}.

I_{WKXSB} The Error records accessed through the System registers might be accessible only to the PE associated with those System registers, or they might be shared and therefore accessible to other PEs through either System registers or as a memory-mapped component.

I_{JMVVD} Direct reads of the System registers, including Error record System registers, can occur speculatively and out-of-order relative to other instructions executed on the same PE.

R_{WFPWF} Direct reads and writes of the Error records through the ERX*_EL1 AArch64 System registers, as described in I_{BJNDF}, are indirect reads of ERRSELR_EL1.

R_{FZBLM} Direct reads and writes of the Error records through the ERX* AArch32 System registers, as described in I_{BJNDF}, are indirect reads of ERRSELR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IYMGGY Explicit synchronization is required after reading an ERXGSR_EL1 AArch64 System register and before reading ERXSTATUS_EL1.

Example D20-16 Direct reads of ERXGSR_EL1 cause indirect reads of ERRSELR_EL1

1. An instruction A writes the value <n> to ERRSELR_EL1.SEL. <n> is the index of an implemented System register error record.
2. Software executes a first Context synchronization event B in program order after A.
3. An instruction C in program order after B reads ERXGSR_EL1, and observes bit [n MOD 64] == `0b1`.
4. Software executes a second Context synchronization event D in program order after C.
5. An instruction E in program order after D reads ERXSTATUS_EL1.

If there is no other direct or external write to ERR<n>STATUS that clears ERR<n>STATUS.V between C and E, then instruction E observes ERR<n>STATUS.V == `0b1`.

D20.7.1.1 Syndrome registers

IRGMHN ESR_ELx, HSR, and DFSR are exception syndrome registers. The PE records syndrome information in an exception syndrome register on taking a physical SError exception or synchronous External abort exception. ESR_ELx, HSR, and DFSR are also used by other exceptions.

DISR_EL1 and DISR are the deferred error syndrome registers. The PE records syndrome information in a deferred error syndrome register on deferring a physical SError exception.

The PE also records a virtual syndrome value in ESR_EL1, DFSR, DISR_EL1, or DISR on taking or deferring a virtual SError exception. The virtual syndrome value is provided by software in a corresponding virtual error syndrome register, VSESR_EL2, VDFSR, VDISR_EL2, or VDISR respectively.

The PE also records a delegated syndrome value in ESR_EL2, ESR_EL1, VDISR_EL2, or DISR_EL1 on taking or deferring a delegated SError exception. The delegated syndrome value is provided by software in a corresponding delegated error syndrome register, VSESR_EL3 or VDISR_EL3 respectively.

RSLNMV For a given implementation:

- If `ESB` never synchronizes any errors, then DISR_EL1.A and DISR.A might be RES0.
- The deferred, virtual, and delegated syndrome registers are capable of storing any syndrome value that might be recorded by the PE in an exception syndrome register on taking a physical SError exception or synchronous External abort exception.
- If any of ESR_ELx[24:0], HSR[11:9], and DFSR[15:14,12] is not used and always set to zero by the PE on taking a physical SError exception or synchronous External abort exception, then that bit can be RES0 in that exception syndrome register.
- A bit that is not used and always set to zero or always set to one in ESR_ELx or DISR_EL1 by the PE on taking a physical SError exception is permitted to be RES0 or RES1 respectively in the corresponding deferred, virtual, and delegated error syndrome registers. See Table D20-4.

In Table D20-4, the deferred virtual, or delegated error syndrome register bit described in the left-hand column is permitted to be RES0 or RES1 only if the corresponding bit is always set to zero or always set to one (respectively) on taking a physical SError exception in all of the implemented exception syndrome registers listed in the other columns marked *Yes* on that row. Otherwise, the bit is read/write.

Table D20-4 Permitted relaxations for bits in deferred, virtual, and delegated error syndrome registers

Bit is permitted to be res0 or res1	ESR_ELx[*n*] *n* ∈ [24:0]	HSR[*n*] *n* ∈ [11:9]	DFSR[*n*] *n* ∈ [15:14,12]
VSESR_EL3[*n*]	Yes	-	-
VDISR_EL3[*n*]	Yes	-	-
VSESR_EL2[*n*]	Yes	-	Yes
VDISR_EL2[*n*]	Yes	-	Yes
DISR_EL1[*n*]	Yes	-	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit is permitted to be res0 or res1	ESR_ELx[*n*] *n* ∈ [24:0]	HSR[*n*] *n* ∈ [11:9]	DFSR[*n*] *n* ∈ [15:14,12]
VDFSR[*n*]	-	-	Yes
VDISR[*n*]	-	-	Yes
DISR[*n*]	-	Yes	Yes

I_{FWGLF} R_{SLNMV} means that VSESR_EL3, VSESR_EL2 and VDFSR can be implemented as RAZ/WI when ESR_ELx[24:0], HSR[11:9], and DFSR[15:14,12] are always set to zero by the PE on taking a physical SError exception or synchronous External abort exception. When this is the case, the PE error state is always reported as Uncategorized error when a physical SError exception is taken to AArch64 state.

I_{GQQCK}, in Error synchronization barriers in a minimal implementation then further allows `ESB` to be executed as a NOP, meaning DISR_EL1, DISR, VDISR_EL2, VDISR_EL3, and VDISR can also be implemented as RAZ/WI.

This allows for a very low-cost implementation of the RAS Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D21
MPAM PE Architecture

This chapter describes the MPAM Extension PE Architecture. It contains the following sections:

- About MPAM.
- PARTID spaces.
- Partition Identifiers (PARTIDs).
- Performance Monitoring Groups (PMGs).
- Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on PE Secure state execution.
- Effect of access type on MPAM information.
- Virtualization support.
- MPAM PE-side Maximum bandwidth limit injection control.
- MPAM errors.
- Synchronization of MPAM System register changes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.1 About MPAM

D21.1.1 Introduction

IYYSRL In many systems, multiple software execution environments such as applications and virtual machines (VMs) run concurrently, sharing memory system resources such as memory controllers, caches, and interconnects. With no controls on the sharing of resources, software execution environments might cause interference for each other as they try to access resources. This might result in increased latency or reduced bandwidth, or both, for memory accesses for workloads that are performance sensitive.

Memory system resource Partitioning and Monitoring (MPAM) provides software with control mechanisms to apportion the memory system resources between software execution environments. This is done by:

- Memory system resource partitioning.
- Memory system resource usage monitoring, one of the uses of which is to inform adjustment of the partitioning.

IRKDKZ MPAM provides all of:

- An extension to the A-profile PE architecture:
  - A mechanism for privileged software executing in a PE to attach a Partition Identifier (PARTID) label, and a monitoring property called a Performance Monitoring Group (PMG), to a software execution environment in that PE.
  - If FEAT_MPAM_PE_BW_CTRL is implemented, then the PE implements a mechanism for software to impose a maximum limit on the bandwidth that requests from the PE itself can use. See MPAM PE-side Maximum bandwidth limit injection control.
- For the system:
  - A description of the propagation of PARTIDs, PMGs, and PARTID spaces, through the system.
  - A framework for implementing *MPAM resource controls* in Memory System Components (MSCs). These are the controls that partition the resources of MSCs.
  - An extension to that framework, for implementing *MPAM resource monitors* in MSCs.
  - An extension to that framework, for MPAM MSCs to have performance monitoring that is sensitive to a combination of PARTID and PMG.
  - Some implementation-independent, memory-mapped interfaces to memory system component controls for performance resource controls most likely to be deployed in systems.
  - Some implementation-independent, memory-mapped interfaces to memory system component resource monitoring that might be required to monitor the partitioning of memory system resources.

IPKXWQ When designing an SoC, the designer makes several or all the following SoC components *MPAM-compliant*, meaning that they implement one of the FEAT_MPAM versions:

- Requester component types, for example PEs, DSPs, and DMA controllers.
- Components that provide shared memory system resources that it is possible to partition, for example shared caches, shared interconnects and buffers, Memory Management Units (MMUs), and memory channel controllers.

There is no requirement to make all components in an SoC MPAM-compliant.

INJBFK In this chapter:

- An *MPAM Requester* is any type of MPAM-compliant Requester component. This chapter categorizes MPAM Requesters into:
  - *MPAM PE Requesters*, also called *MPAM PEs*.
  - *MPAM Non-PE Requesters*.
- An *MPAM Memory System Component (MPAM MSC)* is an MPAM-compliant component that provides shared memory system resources.

This chapter describes MPAM PE Requesters, and *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* describes MPAM MSCs. The architecture does not describe MPAM Non-PE Requesters.

IHRMSG The architecture permits an MPAM MSC to be implemented as part of another system component. For example, an MPAM PE can contain caches that are MPAM MSCs.

RSMNGB If an MPAM PE contains MPAM MSCs, it must have memory-mapped registers to identify and configure those MPAM MSCs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.1.2 MPAM information bundles

R_{BDJXQ} MPAM Requesters label every request to the memory system with an *MPAM information bundle*, comprising:

- Two items that together identify a single partition:
  - A number called the Partition Identifier (PARTID).
  - The PARTID space, for example Secure or Non-secure physical PARTID space.

  Each PARTID space defines a set of PARTIDs.
  Each PARTID corresponds to one partition.
- A Performance Monitoring Group (PMG).

D21.1.3 Memory system resource partitioning

R_{DDGSM} Each MPAM MSC implements zero or more *MPAM resource controls*.

I_{GPWRB} Each MPAM resource control has settings for each PARTID.

I_{DTCTY} When an MPAM Requester sends a request, the PARTID space and PARTID in the MPAM information bundle are used as an index into a table in the MPAM MSC, to select the entry that holds the settings for that partition. Those settings configure how the shared partitionable resource is apportioned to that partition.

I_{MHDHR} See also:

- MPAM PE-side Maximum bandwidth limit injection control for MPAM PE-side bandwidth regulation controls.
- For more detail, *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification*.

D21.1.4 Memory system resource usage monitoring

R_{RBCMK} Each MPAM MSC implements zero or more *MPAM resource monitors*.

I_{LDRSM} An MPAM resource monitor is associated with a shared partitionable resource in an MPAM MSC, and is able to measure how much a software execution environment is using that resource, by either:

- For a *memory-bandwidth usage* monitor, counting payload bytes that are passing where the monitor is placed.
- For a *cache-storage usage* monitor, counting bytes of the cache storage used.

I_{HQBLT} An MPAM resource monitor must be configured and enabled before it can be queried for usage of the MPAM MSC resource. Each monitor has a configurable filter register that the monitor uses to filter which bytes are counted. Fields in the filter register are writes, reads, PARTID, PMG, and other criteria. For example, a monitor can be configured to be sensitive to a PARTID, a PMG, or both.

I_{LWKZR} The architecture permits a resource monitor to be implemented as part of an MPAM PE.

R_{BGTCC} If an MPAM PE contains MPAM resource monitors, it must have memory-mapped registers to identify and configure those monitors.

I_{BGGWJ} See also:

- For more detail, *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification*.

D21.1.5 MPAMn_ELx register naming convention

I_{FFCGH} In an MPAM PE, MPAMn_ELx registers contain the PARTIDs and PMGs that the MPAM PE sends as part of MPAM information bundles. The naming convention of these registers is:

- The n in MPAMn_ELx is the Exception level the register holds PARTIDs and PMGs for. That is, the register supplies PARTIDs and PMGs for execution at ELn.
- The x means that ELx and higher can access the register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CSQBF} The following table shows which MPAMn_ELx register supplies PARTIDs and PMGs for which Exception level:

MPAMn_ELx register	Supplies PARTID and PMG when executing at
MPAM0_EL1	EL0 when MPAMHCR_EL2.GSTAPP_PLK is 0.
MPAM1_EL1	EL1, and EL0 when MPAMHCR_EL2.GSTAPP_PLK is 1. See Single PARTID for an EL1 guest OS and all its EL0 applications.
MPAM2_EL2	EL2
MPAM3_EL3	EL3

D21.1.6 The MPAM enable control

R_{XBYGC} The MPAMn_ELx.MPAMEN field at the highest implemented Exception level is the writable *MPAM enable control*. At all lower Exception levels, all other instances of MPAMn_ELx.MPAMEN are read-only copies. For example:

- If EL3 is the highest implemented Exception level, MPAM3_EL3.MPAMEN is the writable MPAM enable control, and MPAM2_EL2.MPAMEN and MPAM1_EL1.MPAMEN are read-only copies.

MPAM0_EL1 does not contain an MPAMEN field.

I_{KRBMV} See the following for the behavior when MPAM is disabled:

- The Default PARTID.
- The Default PMG.
- Interaction of MPAM3_EL3.SDEFLT and the MPAM enable control.

D21.1.7 AArch32 interoperability

I_{RSTCB} MPAM System registers are not accessible from AArch32 state. Therefore, a higher Exception level must set the PARTIDs and PMGs for any Exception level that uses AArch32 state.

D21.1.8 FEAT_MPAM versions

R_{YSNTG} An MPAM PE implements one of these MPAM versions:

Version	Major version reported by ID_AA64PFR0_EL1.MPAM	Minor version reported by ID_AA64PFR1_EL1.MPAM_frac
v0p1	`0b0000`	`0b0001`
v1p0	`0b0001`	`0b0000`
v1p1	`0b0001`	`0b0001`

I_{TGCKZ} v1p0 is the base version. v0p1 and v1p1 include features beyond the base version and are the same except that v0p1 implements the *Force Secure PARTIDs to Non-secure* feature.

R_{KTQFR} The MPAM PE features by MPAM version are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D21-3 MPAM PE features by MPAM version

MPAM PE feature	MPAMIDR_EL1 ID field	v0p1	v1p0	v1p1	MPAM for RME
PARTID virtualization[a]	HAS_HCR	Optional	Optional	Optional	Optional
Force Secure PARTID to Non-secure[b]	HAS_FORCE_NS	Required	Prohibited	Prohibited	Prohibited
Secure Default PARTID[b]	HAS_SDEFLT	Optional	Prohibited	Optional	Optional
TIDR in MPAM2_EL2	HAS_TIDR	Required	Prohibited	Required	Required
Four PARTID spaces	SP4	Prohibited	Prohibited	Prohibited	Required
Alternative PARTID spaces[c]	HAS_ALTSP	Prohibited	Prohibited	Prohibited	Required
FEAT_MPAM_PE_BW_CTRL[d]	HAS_BW_CTRL	Optional	Prohibited	Optional	Optional

a. See PARTID virtualization.
b. See Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on PE Secure state execution.
c. See In an MPAM PE with FEAT_RME, selection of primary or alternative PARTID space.
d. See MPAM PE-side Maximum bandwidth limit injection control.

R_{MQDXN} The system features by MPAM version are:

Table D21-4 MPAM system features by MPAM version

MPAM system feature	v0p1	v1p0	v1p1	MPAM for RME
MPAM_NS signal[a]	Required	Required	Required	Prohibited
MPAM_SP signal[a]	Prohibited	Prohibited	Prohibited	Required

a. See Encoding PARTID spaces onto PARTID space signals.

I_{VQQHN} MPAM for RME supports the Realm Management Extension (RME) in systems, MPAM PEs, and *MPAM Memory System Components* (MPAM MSCs).

R_{ZDLYQ} MPAM for RME is required in a PE that implements both RME and MPAM.

D21.1.8.1 Interoperation of SoC components implementing different MPAM versions

I_{JVGNT} There is no requirement for all MPAM PEs in an SoC to implement the same MPAM version. However, each MPAM PE having a different MPAM version might cause software issues.

I_{JCHFV} There is no required relationship between the MPAM version of an MPAM PE and the MPAM version of an MPAM MSC accessed by that MPAM PE.

I_{KGCTV} For more information, see:

- *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.2 PARTID spaces

R_{FMXKY} The following physical PARTID spaces are supported:

- Non-secure PARTID space.
- If Secure state is implemented, Secure PARTID space.
- If RME is implemented, both of:
  - Realm PARTID space.
  - Root PARTID space.

I_{DTHSD} In an MPAM PE, MPAMIDR_EL1.SP4 reports whether the MPAM PE supports two or four PARTID spaces.

R_{DJRPQ} In an MPAM PE without FEAT_RME, the physical PARTID space sent as part of an MPAM information bundle depends on the Security state and Exception level of the execution producing the request. If the MPAM version implemented is v0p1, the physical PARTID space also depends on the state of the *Force Non-secure PARTID space* control, MPAM3_EL3.FORCE_NS:

Table D21-5 PARTID space sent by an MPAM PE without FEAT_RME

Security state	PARTID space sent for a request produced by execution at:			In MPAM version v0p1, PARTID space sent when MPAM3_EL3.FORCE_NS is 1:
	EL3	EL2	EL1 and EL0	
Non-secure state	-	Non-secure physical PARTID space	Non-secure physical PARTID space	No change to the behavior[a]
Secure state	Secure physical PARTID space	Secure physical PARTID space	Secure physical PARTID space	Non-secure physical PARTID space

a. The MPAM3_EL3.FORCE_NS control does not affect Non-secure state execution.

R_{NTMXM} In an MPAM PE with FEAT_RME, the physical PARTID space sent as part of an MPAM information bundle depends on the Security state and Exception level of the execution producing the request, and the state of the alternative PARTID space controls, that select whether a *primary* or *alternative* PARTID space is sent:

Table D21-6 PARTID space sent by an MPAM PE with FEAT_RME

Security state	If primary selected, the primary PARTID space sent for a request produced by execution at:			If alternative selected, the alternative sent:
	EL3	EL2	EL1 and EL0	
Non-secure state	-	Non-secure physical PARTID space	Non-secure physical PARTID space	No change to the behavior[a]
Secure state	-	Secure physical PARTID space	Secure physical PARTID space	Non-secure physical PARTID space
Realm state	-	Realm physical PARTID space	Realm physical PARTID space	Non-secure physical PARTID space
Root state	Root physical PARTID space	-	-	Selectable Secure or Non-secure physical PARTID space[a]

a. The alternative PARTID space controls do not affect Non-secure state execution.
b. For detail of how Secure or Non-secure is selected, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

R_{FFRDD} The Secure physical PARTID space is not used in systems, or system components, that do not implement Secure state.

I_{VNRRQ} See also:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For detail on how an MPAM PE Requester sends a PARTID space, Encoding PARTID spaces onto PARTID space signals.
- For detail on the alternate PARTID space controls, In an MPAM PE with FEAT_RME, selection of primary or alternative PARTID space.

D21.2.1 PARTID space virtualization

I_{CCVLY} In an MPAM PE, if EL2 is implemented and enabled in the current Security state, and MPAMIDR_EL1.HAS_HCR indicates that the MPAM virtualization option is implemented, execution at EL0 and EL1 might produce a *virtual PARTID space*, that is translated to a *physical PARTID space*. For details, see PARTID virtualization.

D21.2.2 In an MPAM PE with FEAT_RME, selection of primary or alternative PARTID space

R_{SWNPB} For EL3 Root state execution:

- MPAM3_EL3.ALTSP_EL3 selects whether EL3 produces the *primary* or *alternative* PARTID space.
- MPAM3_EL3.RT_ALTSP_NS selects whether the alternative PARTID space is Secure or Non-secure:

Table D21-7 PARTID space that EL3 produces, in an MPAM PE with FEAT_RME

ALTSP_EL3	RT_ALTSP_NS	PARTID space that EL3 produces
0	X	Primary PARTID space. Root physical PARTID space.
1	0	Alternative PARTID space. Secure physical PARTID space.
1	1	Alternative PARTID space. Non-secure physical PARTID space.

R_{DFMYN} For execution below EL3, EL3 is either:

- Forcing primary or alternative PARTID spaces for execution at EL2, EL1, and EL0.
- Permitting EL2 to control its own PARTID space, and the PARTID space for EL1 and EL0.

Forcing primary or alternative PARTID spaces for execution at EL2, EL1, and EL0, is by:

1. MPAM3_EL3.ALTSP_HEN=0 disabling the EL2 *alternative PARTID space controls*.
2. MPAM3_EL3.ALTSP_HFC selecting the primary or alternative PARTID space for execution at EL2, EL1, and EL0.

Permitting EL2 to control its own PARTID space, and the PARTID space for EL1 and EL0, is by:

- MPAM3_EL3.ALTSP_HEN=1 enabling the EL2 alternative PARTID space controls, which means that:
  - EL2 can configure MPAM2_EL2.ALTSP_EL2 to select the primary or alternative PARTID space for EL2.
  - EL2 can configure MPAM2_EL2.ALTSP_HFC to select the primary or alternative PARTID space for EL1 and EL0.

See the following tables for execution at EL2, and at EL1 and EL0, respectively:

Table D21-8 PARTID space that EL2 produces, when EL2 is enabled in the Security state, in an MPAM PE with FEAT_RME

MPAM3_EL3.ALTSP_HEN	MPAM3_EL3.ALTSP_HFC	MPAM2_EL2.ALTSP_EL2	PARTID space that EL2 produces
0	0	X	Primary.
0	1	X	Alternative. Non-secure physical PARTID space.
1	X	0	Primary.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM3_EL3.ALTSP_HEN	MPAM3_EL3.ALTSP_HFC	MPAM2_EL2.ALTSP_EL2	PARTID space that EL2 produces
1	X	1	Alternative. Non-secure physical PARTID space.

Table D21-9 PARTID space that EL1 and EL0 produce, in an MPAM PE with FEAT_RME

MPAM3_EL3.ALTSP_HEN	MPAM3_EL3.ALTSP_HFC	MPAM2_EL2.ALTSP_HFC	PARTID space that EL1 and EL0 produce
0	0	X	Primary.
0	1	X	Alternative. Non-secure physical PARTID space.
1	X	0	Primary.
1	X	1	Alternative. Non-secure physical PARTID space.

R_{KXKGR} When EL2 is disabled in the current Security state, EL1 and EL0 produce the alternative PARTID space when MPAM3_EL3.{ALTSP_HEN, ALTSP_HFC} are {0, 1}, and produce the primary PARTID space otherwise.

R_{BTVYC} When MPAM3_EL3.MPAMEN is 0, so that MPAM is disabled from EL3, MPAM2_EL2.{ALTSP_HFC, ALTSP_EL2} are treated as {0, 0}, and MPAM3_EL3.{ALTSP_HEN, ALTSP_HFC, ALTSP_EL3, RT_ALTSP_NS} are effective.

I_{HDJCV} MPAMn_ELx.ALTSP_FRCD reports whether execution at ELn is being forced to produce the alternative PARTID space:

- MPAM2_EL2.ALTSP_FRCD reports, in all implemented Security states, whether EL3 is forcing the alternative PARTID space for execution at EL2.
- MPAM1_EL1.ALTSP_FRCD reports, in all implemented Security states, whether EL3 or EL2 is forcing the alternative PARTID space for execution at EL1 and EL0.

There is no ALTSP_FRCD field in MPAM0_EL1. There is no mechanism for EL0 to discover whether it is producing the primary or alternative PARTID space. However, EL0 is always producing the same PARTID space as either:

- EL2, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL1 otherwise.

D21.2.3 Encoding PARTID spaces onto PARTID space signals

R_{VKVCD} The physical PARTID space that an MPAM PE Requester sends as part of an MPAM information bundle is sent by encoding it onto:

- An **MPAM_NS** signal if the MPAM PE does not implement RME.
- An **MPAM_SP** signal if the MPAM PE implements RME.

Table D21-10 Encoding of the physical PARTID space sent if the MPAM PE does not implement FEAT_RME

PARTID space the MPAM PE Requester is sending	Encoding on MPAM_NS signal
Secure physical PARTID space	`0b0`
Non-secure physical PARTID space	`0b1`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D21-11 Encoding of the physical PARTID space sent if the MPAM PE implements FEAT_RME

PARTID space the MPAM PE Requester is sending	Encoding on MPAM_SP signal
Secure physical PARTID space	`0b00`
Non-secure physical PARTID space	`0b01`
Root physical PARTID space	`0b10`
Realm physical PARTID space	`0b11`

In MPAM PEs without Secure state, **MPAM_NS** encoding `0b0` or **MPAM_SP** encoding `0b00` is reserved.

I_{MRBXT} SoCs can be constructed comprising Requester components that all support MPAM_NS, that all support MPAM_SP, or some that support MPAM_NS and some that support MPAM_SP.

I_{TWGKV} In MPAMv0p1, if MPAM3_EL3.FORCE_NS is implemented as:

- RAO/WI, Secure PARTIDs are not implemented, and the MPAM PE Requester always sends MPAM_NS as 1 for both the Secure physical PARTID space and Non-secure physical PARTID space.
- RW, then setting MPAM3_EL3.FORCE_NS to 1 causes the MPAM PE Requester to always send MPAM_NS as 1 for both the Secure physical PARTID space and Non-secure physical PARTID space.

See Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on MPAM PE Secure state execution.

I_{NZPRL} See also:

- PARTID spaces, for detail on how an MPAM PE Requester determines which physical PARTID space to send.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.3 Partition Identifiers (PARTIDs)

I_{NHJZF} A *Partition Identifier* (PARTID) is a number that has no inherent meaning and does not imply any ordering. The number solely references a particular partition in a PARTID space.

R_{CZHSY} Each PARTID can exist once in each PARTID space.

R_{RBWRS} The scope of a PARTID is bound by its PARTID space. If a PARTID is used in multiple PARTID spaces, then a reference to that PARTID in one PARTID space does not also reference that PARTID in the other PARTID spaces.

I_{NFCZF} Each *MPAM-compliant* component in an SoC is permitted to implement a maximum PARTID width that is less than or equal to the architecturally defined maximum width of 16-bits.

I_{NPBWK} In an MPAM PE, MPAMIDR_EL1.PARTID_MAX reports the maximum PARTID implemented.

R_{PJTCQ} In an MPAM PE, MPAMn_ELx.{PARTID_D, PARTID_I} are permitted to hold PARTIDs in the range 0 to MPAMIDR_EL1.PARTID_MAX, inclusive.

R_{NYXVG} The PARTID that an MPAM PE Requester sends as part of an MPAM information bundle is one of:

- If any of the conditions for the Default PARTID are met, the Default physical PARTID.
- If the PARTID held is *treated as virtual* and a valid mapping exists, the mapped physical PARTID. Also see both:
  - Behavior when a valid mapping does not exist for a PARTID that is treated as virtual.
  - Behavior when a PARTID is out-of-range.
- If the PARTID held is not treated as virtual, it is determined as follows:
  - For an instruction fetch, MPAMn_ELx.PARTID_I is used.
  - For a data access, MPAMn_ELx.PARTID_D is used. However, if a data access is due to the execution of an SME load or store instruction, or of a Streaming SVE mode load or store instruction, MPAMSM_EL1.PARTID_D is used instead.

  Also see Behavior when a PARTID is out-of-range.

I_{QKRPD} Software can set MPAMn_ELx.{PARTID_D, PARTID_I} to the same or different PARTID values.

I_{QDFWX} Arm recommends that all MPAM-compliant components in an SoC implement the same maximum PARTID, though this is not required.

I_{TLVJP} See also:

- About MPAM, for the definitions of *MPAM-compliant* and *PE Requester*.
- PARTID virtualization, for the definition of *treated as virtual*.

D21.3.1 PARTID virtualization

R_{DNZFT} The behaviors in this section apply if all the following are true:

- EL2 is implemented and enabled in the current Security state.
- Any FEAT_MPAM version is implemented.
- MPAMIDR_EL1.HAS_HCR is 1, meaning that the MPAM virtualization option is implemented.
- If a version other than v1p0 is implemented and execution is in Secure state, then in addition:
  - MPAM3_EL3.SDEFLT is 0, preventing generation of the Default physical PARTID in Secure state. See Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on MPAM PE Secure state execution.

I_{PKKWM} EL2 hypervisor software continuously updates *MPAM virtual PARTID mapping registers*, MPAMVPM0_EL2 to MPAMVPM7_EL2, that map virtual PARTIDs to physical PARTIDs.

I_{TSKGT} Each MPAMVPMn_EL2 register contains four *mapping fields*, and the maximum implemented MPAMVPMn_EL2 is MPAMIDR_EL1.VPMR_MAX. Therefore, the maximum implemented mapping field is (4 x VPMR_MAX) + 3, and virtualized guests are limited to mapping fields in the range 0 to (4 x VPMR_MAX) + 3.

I_{TXFDH}

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D21-1

If MPAMIDR_EL1.VPMR_MAX reports:

- `0b000`, the maximum implemented register is MPAMVPM0_EL2, and the maximum implemented mapping field is (4 x 0) + 3 = 3. There are therefore four virtual PARTIDs available for virtualized guests, numbered 0 to 3, that can be mapped to physical PARTIDs.
- `0b111`, the maximum implemented register is MPAMVPM7_EL2, and the maximum implemented mapping field is (4 x 7) + 3 = 31. There are therefore 32 virtual PARTIDs available for virtualized guests, numbered 0 to 31, that can be mapped to physical PARTIDs.

R_{QMSMK}

When:

- MPAMHCR_EL2.EL0_VPMEN is 1, the PARTIDs in MPAM0_EL1.{PARTID_D, PARTID_I} are *treated as virtual*, and EL0 produces requests with virtual PARTID spaces.
- MPAMHCR_EL2.EL1_VPMEN is 1, the PARTIDs in MPAM1_EL1.{PARTID_D, PARTID_I} are treated as virtual, and EL1 produces requests with virtual PARTID spaces.

When a PARTID is treated as virtual, the MPAM PE attempts to map the PARTID to a physical PARTID using a MPAM virtual PARTID mapping register MPAMVPMn_EL2. The PARTID that is treated as virtual is used as an index into both:

- The vector of physical PARTIDs that the mapping registers hold.
- The array of valid flags that MPAMVPMV_EL2 holds.

I_{XGFNP}

Example D21-2 Example of an MPAM PE mapping a virtual PARTID to a physical PARTID

Assume all the following are true:

- EL2 is implemented and enabled in the current Security state.
- Version v1p1 of FEAT_MPAM is implemented, MPAMIDR_EL1.HAS_HCR is 1, meaning that the MPAM virtualization option is implemented, and MPAMIDR_EL1.HAS_SDEFLT is 0, meaning that MPAM3_EL3.SDEFLT is not implemented.
- MPAMHCR_EL2.EL1_VPMEN is 1, meaning that PARTIDs in MPAM1_EL1.{PARTID_D, PARTID_I} are treated as virtual.
- MPAMIDR_EL1.VPMR_MAX is `0b010`, meaning that the maximum implemented mapping field is (4 x 2) + 3 = 11.
- The value of MPAM1_EL1.PARTID_D is 6.

In this example, 6 is treated as virtual and is the index into the array of valid flags and vector of physical PARTIDs. Behavior is:

1. Valid flag MPAMVPMV_EL2.VPM_V<6> is checked:
   1. A result of 1 means a valid mapping exists for virtual PARTID 6. MPAMVPM1_EL2.PhysPARTID6 contains the physical PARTID that the MPAM PE Requester sends.
   2. A result of 0 means *invalid*.
2. If the result is invalid, the mapping for the Default virtual PARTID, 0, is used. If a valid mapping does not exist for virtual PARTID 0, the MPAM PE Requester sends the Default physical PARTID.

R_{XSJVY}

The virtual PARTID spaces are:

- If Secure state is implemented, Secure virtual PARTID space. This translates to the Secure physical PARTID space.
- Non-secure virtual PARTID space. This translates to the Non-secure physical PARTID space.
- If FEAT_RME is implemented, Realm virtual PARTID space. This translates to the Realm physical PARTID space.

I_{BSWTR}

If a virtual PARTID is greater than (4 x MPAMIDR_EL1.VPMR_MAX) + 3, it is an *out-of-range virtual PARTID*.

R_{QTSFN}

When a PARTID is an out-of-range virtual PARTID, the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The out-of-range virtual PARTID is replaced by any in-range virtual PARTID in the same PARTID space.
- The out-of-range virtual PARTID is replaced by the Default physical PARTID in the same PARTID space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{HWFQY} In each MPAMVPMn_EL2.PhyPARTID<n> mapping field, the architecture requires implementation of sufficient low-order bits to represent MPAMIDR_EL1.PARTID_MAX. It is permitted to implement the bits that are higher-order than MPAMIDR_EL1.PARTID_MAX as RAZ/WI.

D21.3.2 The Default PARTID

R_{QKQCF} The *Default PARTID* number, in each PARTID space, is 0.

R_{CQXMC} In the following scenario, the MPAM PE attempts to map the *Default virtual PARTID*:

- When no valid mapping exists for a PARTID that is *treated as virtual*.
  See PARTID virtualization.

In each of the following scenarios, the MPAM PE Requester sends the *Default physical PARTID* as part of the MPAM information bundle:

- When the MPAM enable control is 0, meaning MPAM is disabled.
- When the MPAM version implemented is any other than v1p0, the request is from Secure state, and MPAM3_EL3.SDEFLT is implemented and is set to 1. See Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on MPAM PE Secure state execution.
- When a PARTID is an *out-of-range virtual PARTID* and the choice of CONSTRAINED UNPREDICTABLE behavior is to replace it with the Default physical PARTID. See Behavior when a PARTID is out-of-range.
- When no valid mapping exists for the Default virtual PARTID. See Behavior when a valid mapping does not exist for a PARTID that is treated as virtual.

Note

I_{XCZMJ} The MPAM architecture permits SoC designers to make non-MPAM Requesters label every request to the memory system with the Default physical PARTID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.4 Performance Monitoring Groups (PMGs)

I_{NQQHK} An MPAM resource monitor measures how much of a shared partitionable resource a software execution environment is using. See Memory system resource usage monitoring. A *Performance Monitoring Group (PMG)* is an identifier used in the filter register of a monitor, MSMON_CFG_MBWU_FLT or MSMON_CFG_CSU_FLT.

Filtering by PMG provides a way to group measurements, and a way to subdivide measurements. For example:

- Multiple monitors, each measuring usage of a different shared resource, can be made sensitive to a single PMG and no PARTIDs, so that they create a group of measurements for usage of the different shared resources by one PMG value.
- Monitors can be made sensitive to both a PARTID and a PMG, so that the PMG provides a subdivision of what is measured for the PARTID.

R_{BVQFD} The PMG that an MPAM PE Requester sends as part of an MPAM information bundle is one of:

- If any of the conditions for the the Default PMG are met, the Default PMG.
- Otherwise, the PMG is determined as follows:
  - For an instruction fetch, MPAMn_ELx.PMG_I is used.
  - For a data access, MPAMn_ELx.PMG_D is used. However, if a data access is due to the execution of an SME load or store instruction, or of a Streaming SVE mode load or store instruction, MPAMSM_EL1.PMG_D is used instead.

  Also see Behavior when a PMG is out-of-range.

I_{JYNXW} Software can set MPAMn_ELx.{PMG_D, PMG_I} to the same or different PMG values.

D21.4.1 PMG virtualization

I_{JNHSB} Performance Monitoring Group (PMG) values are not virtualized. PMGs modify PARTIDs and do not require any further virtualization support.

D21.4.2 The Default PMG

R_{CZSZX} The *Default PMG* number is 0.

R_{PQSLB} In each of the following scenarios, the MPAM PE Requester sends the Default PMG as part of the MPAM information bundle:

- When the MPAM enable control is 0, meaning that MPAM is disabled.
- When a PMG is an *out-of-range* PMG and the choice of CONSTRAINED UNPREDICTABLE behavior is to replace it with the Default PMG. See Behavior when a PMG is out-of-range.
- When a Default PARTID is generated due to a PARTID generation error, and the choice of CONSTRAINED UNPREDICTABLE behavior is to send the Default PMG. See PMG value when a Default PARTID is generated due to an MPAM error.

Note

I_{TJBXP} The MPAM architecture permits SoC designers to make non-MPAM Requesters label every request to the memory system with the Default PMG.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.5 Effect of the MPAM3_EL3.{SDEFLT, FORCE_NS} controls on PE Secure state execution

R_{RFBMQ} The MPAM3_EL3.{SDEFLT, FORCE_NS} controls only affect Secure state execution.

R_{SHYXR} In MPAMv0p1 and from MPAMv1p1, if the *Secure Default* control MPAM3_EL3.SDEFLT is implemented, the PARTID and PMG that Secure state execution produces depends on the value of that control:

MPAM3_EL3.SDEFLT	PARTID and PMG that Secure state execution produces
0b0	MPAMn_ELx.{PARTID_D, PMG_D} or {PARTID_I, PMG_I}. This is compatible with MPAMv1p0.
0b1	The Default PARTID and Default PMG.

R_{XFCWV} In MPAMv0p1, the *Force Non-secure PARTID space* control MPAM3_EL3.FORCE_NS is implemented. If MPAM3_EL3.SDEFLT is also implemented, the PARTID space, PARTID, and PMG that Secure state execution produces are as follows:

MPAM3_EL3 control		PARTID space, PARTID, and PMG that Secure state execution produces
SDEFLT	FORCE_NS	
0b0	0b0	• Secure physical PARTID space. • MPAMn_ELx.{PARTID_D, PMG_D} or {PARTID_I, PMG_I}. This is compatible with MPAMv1p0.
	0b1	• Non-secure physical PARTID space. • MPAMn_ELx.{PARTID_D, PMG_D} or {PARTID_I, PMG_I}.
0b1	0b0	• Secure physical PARTID space. • Default PARTID and Default PMG.
	0b1	• Non-secure physical PARTID space. • Default PARTID and Default PMG.

R_{MSGKT} In MPAMv0p1, it is IMPLEMENTATION DEFINED whether MPAM3_EL3.FORCE_NS is implemented as:

- RAO/WI, and Secure PARTIDs are not implemented.
- RW.

I_{QJPCV} In MPAMv0p1, software can discover whether MPAM3_EL3.FORCE_NS is implemented as RAO/WI or RW by testing whether it is writable to zero.

R_{TRNQL} In MPAM for RME, implementation of MPAM3_EL3.FORCE_NS is not permitted and implementation of the alternative PARTID space controls, that select whether a primary or alternative PARTID space is produced, is required. The effect of MPAM3_EL3.SDEFLT if it is implemented is:

Alternative PARTID space selected?	MPAM3_EL3.SDEFLT	PARTID space, PARTID, and PMG that Secure state execution produces
No	0b0	• Secure physical PARTID space. • MPAMn_ELx.{PARTID_D, PMG_D} or {PARTID_I, PMG_I}. This is compatible with MPAMv1p0.
	0b1	• Secure physical PARTID space. • Default PARTID and Default PMG.
Yes	0b0	• Non-secure physical PARTID space. • MPAMn_ELx.{PARTID_D, PMG_D} or {PARTID_I, PMG_I}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alternative PARTID space selected?	MPAM3_EL3.SDEFLT	PARTID space, PARTID, and PMG that Secure state execution produces
	`0b1`	• Non-secure physical PARTID space. • Default PARTID and Default PMG.

D21.5.1 Interaction of MPAM3_EL3.SDEFLT and the MPAM enable control

R_{CBRSV} In MPAMv0p1 and from MPAMv1p1, if the *Secure Default* control MPAM3_EL3.SDEFLT is implemented:

- When the MPAM enable control, MPAMEN, is 0, the MPAM PE Requester sends the Default physical PARTID and the Default PMG with all requests to the memory system.
- When MPAMEN is 1 and SDEFLT is 0, all requests from all Security states produce the PARTID and PMG from the MPAMn_ELx registers.
- When MPAMEN is 1 and SDEFLT is 1:
  - All requests from Secure state produce the Default PARTID and the Default PMG.
  - All requests from all other Security states produce the PARTID and PMG from the MPAMn_ELx registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.6 Effect of access type on MPAM information

I_{BZNRF} See the following subsections:

- Translation table accesses.
- Granule Protection Table (GPT) accesses.
- Accesses by enhanced support for nested virtualization.
- AT* instructions.
- Unprivileged memory access instructions.
- Accesses by Statistical Profiling Extension.

D21.6.1 Translation table accesses

R_{FYLGZ} Translation table walk accesses, including hardware updates of translation tables, for:

- Instruction fetches, use the same PARTID_I field as their instruction fetch.
- Data accesses, use the same PARTID_D field as their data access.

D21.6.2 Granule Protection Table (GPT) accesses

R_{MFSPP} In MPAM for RME, Granule Protection Table (GPT) accesses for:

- Instruction fetches, use the same PARTID space, PARTID_I, field, and PMG_I field as their instruction fetch.
- Data accesses, use the same PARTID space, PARTID_D field, and PMG_D field as their data access.
- A translation table walk, use the same PARTID space, PARTID, and PMG as the translation table access.

D21.6.3 Accesses by enhanced support for nested virtualization

R_{NFJPN} In an MPAM PE with FEAT_NV2, when the Effective value of HCR_EL2.{NV, NV2} is {1, 1}, `MRS` and `MSR` accesses to certain registers from EL1 are transformed to memory accesses. Those accesses, and translation table walks generated for those accesses, use MPAM2_EL2.{PARTID_D, PMG_D}. See Support for nested virtualization.

D21.6.4 AT* instructions

R_{RSNZD} Accesses to translation tables by `AT*` instructions are given the MPAM information specified for translation table accesses by a data load instruction that is issued from an IMPLEMENTATION DEFINED choice of:

- The Exception level that the `AT*` instruction was executed from.
- The Exception level that configures the translation regime targeted by the `AT*` instruction.

D21.6.5 Unprivileged memory access instructions

R_{GPVVG} For unprivileged memory access instructions executed at ELx, the resulting memory system requests use the same MPAM labels that are used for regular loads and stores at ELx.

D21.6.6 Accesses by Statistical Profiling Extension

R_{CRJBB} An MPAM PE with FEAT_SPE can be configured to record statistically sampled events into a Profiling Buffer in memory. The buffer is accessed through the owning translation regime. MPAM PARTID, PMG, and PARTID space for SPE writes to the Profiling Buffer must use the SPE owning Exception level MPAM data access values. For example, if the owning Exception level is EL2, the Profiling Buffer writes must be performed with MPAM2_EL2.PARTID_D, MPAM2_EL2.PMG_D, with **MPAM_NS** or **MPAM_SP** reflecting the owning Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.6.7 Accesses by Trace Buffer Extension

I_{MNMXW} A PE with FEAT_TRBE can be configured to record trace data into a trace buffer in memory. The MPAM information used for these accesses depends on the operating mode of the Trace Buffer Unit. For more information, see:

- Accesses by Trace Buffer Extension.
- External mode and MPAM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.7 Virtualization support

I_{HRVFB} See the following subsections:

- Single PARTID for an EL1 guest OS and all its EL0 applications.
- Trap controls to virtualize MPAMIDR_EL1.
- Trap controls to trap accesses to MPAM System registers.
- Trap controls priorities.
- Virtualization of PARTIDs.
- Interaction with FEAT_VHE.
- Support for nested virtualization.

D21.7.1 Single PARTID for an EL1 guest OS and all its EL0 applications

R_{FNKGZ} Setting MPAMHCR_EL2.GSTAPP_PLK to 1 causes MPAM1_EL1 to supply PARTIDs and PMGs for both EL1 and EL0, and causes MPAM0_EL1 to be unused, meaning that an EL1 guest OS and all its EL0 applications are in the same partition.

R_{JNWLV} When MPAMHCR_EL2.{GSTAPP_PLK, EL1_VPMEN} is {1, 1}, the PARTID that results from the mapping process is the PARTID that the MPAM PE Requester sends for requests originating from EL0, as well as for those originating from EL1. See PARTID virtualization.

Note

I_{RMPJJ} EL1 software cannot determine the state of MPAMHCR_EL2.GSTAPP_PLK. MPAMHCR_EL2.GSTAPP_PLK==1 does not affect EL1 accesses to MPAM0_EL1.

D21.7.2 Trap controls to virtualize MPAMIDR_EL1

I_{LDNNY} An EL2 hypervisor is able to provide a virtual view of MPAMIDR_EL1 to an EL1 guest OS by using MPAMHCR_EL2.TRAP_MPAMIDR_EL1, or MPAM2_EL2.TIDR if MPAMIDR_EL1.HAS_TIDR is 1, to trap EL1 accesses to MPAMIDR_EL1. The virtual view shows MPAMIDR_EL1.PARTID_MAX as the largest *virtual PARTID* that the EL2 hypervisor is providing for the EL1 guest OS to use.

See also:

- Partition identifiers (PARTIDs).
- PARTID virtualization.

D21.7.3 Trap controls to trap accesses to MPAM System registers

I_{TKRWM} The MPAM architecture provides trap controls for all the following:

- EL1 accesses to MPAM0_EL1, MPAM1_EL1, and MPAMSM_EL1, to trap these to EL2. See the trap controls MPAM2_EL2.{TRAPMPAM0EL1, TRAPMPAM1EL1, EnMPAMSM} respectively.
- EL1 accesses to MPAMBW0_EL1, MPAMBW1_EL1, and MPAMBWSM_EL1, to trap these to EL2. See the trap controls MPAMBW2_EL2.{nTRAP_MPAMBW0_EL1, nTRAP_MPAMBW1_EL1, nTRAP_MPAMBWSM_EL1} respectively.
- EL2 and EL1 accesses to MPAM System registers, to trap these to EL3. See MPAM3_EL3.TRAPLOWER and MPAMBW3_EL3.nTRAPLOWER.

I_{SQQYN} Accesses to all the following are not subject to any traps:

- MPAM1_EL12.
- MPAM1_EL1 when the Effective value of HCR_EL2.E2H is 1.

See Interaction with FEAT_VHE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.7.4 Trap controls priorities

R_{CFZGR} MPAM3_EL3.TRAPLOWER and MPAMBW3_EL3.nTRAPLOWER traps have priority over all traps controlled by MPAM registers with names ending in _EL2.

D21.7.5 Virtualization of PARTIDs

I_{JNSFB} See PARTID virtualization.

D21.7.6 Interaction with FEAT_VHE

I_{QQPKC} When the Effective value of HCR_EL2.E2H is 1, the following are alias registers for accesses from EL2:

- MPAM1_EL12 is an alias for MPAM1_EL1. See R_{PHHPL} in System and Special-purpose register aliasing.
- MPAM1_EL1 is an alias for MPAM2_EL2. See R_{JGGMV} in System and Special-purpose register redirection.
- MPAMBW1_EL12 is an alias for MPAMBW1_EL1. See R_{PHHPL} in System and Special-purpose register aliasing.
- MPAMBW1_EL1 is an alias for MPAMBW2_EL2. See R_{JGGMV} in System and Special-purpose register redirection.

I_{LVGCR} If the Effective value of HCR_EL2.E2H is 1, software executing at EL2 can use all of:

- MPAM1_EL12, to control PARTIDs and PMGs for execution at EL1.
- MPAM1_EL1, to configure the MPAM controls that apply to execution at EL2.
- MPAMBW1_EL12, to configure a maximum limit on the bandwidth that EL1 requests use.
- MPAMBW1_EL1, to configure a maximum limit on the bandwidth that EL2 requests use.

I_{FCMPY} The Effective value of HCR_EL2.E2H does not affect:

- Use of the following registers by EL2 software:
  - MPAM0_EL1, to control PARTIDs and PMGs used by applications running at EL0, or MPAMBW0_EL1, to configure a maximum limit on the bandwidth that EL0 requests use. When running applications at EL0, EL2 software also sets HCR_EL2.TGE to 1 to route exceptions from EL0 to EL2.
  - MPAMHCR_EL2.
  - MPAMVPMV_EL2.
  - MPAMVPMn_ELx.
  - MPAMBWCAP_EL2.
- Use of the following registers by EL1 software:
  - MPAM0_EL1, to control PARTIDs and PMGs used by applications running at EL0, and MPAM1_EL1, to control PARTIDs and PMGs used by EL1 software.
  - MPAMBW0_EL1 and MPAMBW1_EL1, to configure maximum limits on the bandwidth that EL0 requests and EL1 requests use, respectively.

R_{XCQTJ} When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the alternative PARTID space in EL0 is selected only if the alternative PARTID space is selected in EL2. When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the alternative PARTID space in EL0 is selected only if the alternative PARTID space is selected in EL1.

D21.7.7 Support for nested virtualization

R_{WDPPT} In an MPAM PE with FEAT_NV, when the Effective value of HCR_EL2.{NV, NV2} is {1, 0}, EL1 accesses to the following System registers are trapped to a higher Exception level:

- MPAM1_EL12.
- MPAM2_EL2.
- MPAMHCR_EL2.
- MPAMVPMV_EL2.
- The *MPAM virtual PARTID mapping registers*, MPAMVPM0_EL2 to MPAMVPM7_EL2.
- The *MPAM bandwidth control registers* MPAMBW2_EL2, MPAMBWCAP_EL2, and MPAMBW1_EL12.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the MPAM bandwidth control registers MPAMBW2_EL2, MPAMBWCAP_EL2, and MPAMBW1_EL12, the controls that determine which Exception level EL1 accesses are trapped to are MPAM3_EL3.TRAPLOWER and MPAMBW3_EL3.nTRAPLOWER:

Table D21-15 MPAM Bandwidth control register traps

MPAM3_EL3.TRAPLOWER	MPAMBW3_EL3.nTRAPLOWER	Behavior when HCR_EL2.{NV, NV2} is {1, 0}
0	0	EL1 access trapped to EL3
0	1	EL1 access trapped to EL2
1	0	EL1 access trapped to EL3
1	1	EL1 access trapped to EL3

Otherwise, for the other registers, the value of MPAM3_EL3.TRAPLOWER determines which higher Exception level EL1 accesses are trapped to:

- When 0, the EL1 accesses are trapped to EL2.
- When 1, the EL1 accesses are trapped to EL3.

I_{XXYXW} In an MPAM PE with FEAT_NV2, Loads and stores generated by transforming register accesses describes the behavior when the Effective value of HCR_EL2.{NV, NV1, NV2} is {1, 0, 1} and {1, 1, 1}, and Table D8-120 includes the MPAM System registers for which that behavior applies.

R_{JPQVQ} In an MPAM PE with FEAT_NV2, when the Effective value of HCR_EL2.{NV, NV2} is {1, 1}, setting MPAM3_EL3.TRAPLOWER to 1 does not cause EL1 accesses to the MPAM System registers in Table D8-120 to be trapped to EL3.

I_{RXGXW} When the Effective value of HCR_EL2.{NV, NV2} is {1, 1}, EL1 accesses to the MPAM System registers are transformed to memory accesses. See Loads and stores generated by transforming register accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.8 PE-side Maximum bandwidth limit injection control

I_{CBQCG} If FEAT_MPAM_PE_BW_CTRL is implemented, software can use MPAMBWn_ELx registers to impose a maximum limit on the bandwidth that requests from the MPAM PE use.

R_{VQNTY} When MPAMBWn_ELx.ENABLED is 1 and either of the following is true, MPAM PE requests to the memory system are stalled until the bandwidth the MPAM PE is using drops below the limit configured in MPAMBWn_ELx.MAX:

- MPAMBWIDR_EL1.MAX_LIM is `0b00` or `0b10` and MPAMBWn_ELx.HARDLIM is set.
- MPAMBWIDR_EL1.MAX_LIM is `0b00` or `0b01`, MPAMBWn_ELx.HARDLIM is not set, and an IMPLEMENTATION DEFINED mechanism has determined the MPAM PE downstream resources to be saturated.

Otherwise, MPAM PE requests to the memory system are sent unregulated.

R_{RGLDT} MPAMBWSM_EL1.MAX is used instead of MPAMBWn_ELx.MAX when:

- The MPAM PE is in streaming mode, and:
  - Accessing from SVE load/store/prefetch.
  - Accessing from SIMD&FP load/store.
- Accessing from SME load/store.

I_{DPMDJ} If MPAMIDR_EL1.HAS_HCR is 1, meaning that the MPAM virtualization option is implemented, and FEAT_MPAM_PE_BW_CTRL is implemented, EL2 software can use MPAMBWCAP_EL2 to cap a maximum bandwidth limit assigned by EL1&0 software.

R_{CMMQR} When MPAMBWCAP_EL2.ENABLED is 1, EL1&0 requests to the memory system are regulated based on the limit calculated as MIN(MPAMBWCAP_EL2.CAP, MPAMx_BW_CTRL_EL.MAX) and MIN(MPAMBWCAP_EL2.CAP, MPAMBWSM_EL1.MAX).

I_{VVYKT} If {MPAMBWn_ELx, MPAMBWSM_EL1, MPAMBWCAP_EL2}.HW_SCALE_ENABLE is 1, then hardware is permitted to scale down the maximum limit configured in {MPAMBWn_ELx, MPAMBWSM_EL1}.MAX or MPAMBWCAP_EL2.CAP according to the number of PEs in a group that are all issuing the same specific PARTID_D or PARTID_I at the corresponding exception level.

R_{CRGFM} It is IMPLEMENTATION DEFINED how hardware determines the number of PEs that are all issuing the same specific PARTID_D or PARTID_I at any point in time.

I_{FYXGY} Memory bandwidth regulation relies on tracking bandwidth use by PARTIDs. Bandwidth measurements have a dependence on time, and that dependence is referred to as the *accounting window*. The accounting window tracks bandwidth use over a period of time.

R_{FZMPQ} The PE-side maximum bandwidth limit accounting window scheme and its software-discoverability are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.9 MPAM errors

R_{DHSBM} An MPAM error is detected when a memory read, memory write, instruction fetch, or translation table walk results in one of:

- The PARTID is *out-of-range*. Either:
  - The PARTID is a physical PARTID and it exceeds MPAMIDR_EL1.PARTID_MAX.
  - The PARTID is *treated as virtual* and it exceeds (4 x MPAMIDR_EL1.VPMR_MAX) + 3.
- The PARTID is treated as virtual and the corresponding MPAMVPMV_EL2.VPM_V<m> valid flag indicates that a valid mapping does not exist.
- The PMG is out-of-range. The PMG exceeds MPAMIDR_EL1.PMG_MAX.

I_{DWCRN} In some implementations, some MPAM errors might never occur. For example, an implementation with only w bits of PARTID and MPAMIDR_EL1.PARTID_MAX as (2^w - 1), and that truncates PARTID values with non-zero bits higher than w - 1, can never have a physical PARTID out-of-range error.

See also:

- PARTID virtualization, for the definitions of *out-of-range* and *treated as virtual*.

D21.9.1 Behavior when a PARTID is out-of-range

R_{YGRQN} When a physical or virtual PARTID from an MPAMn_ELx.PARTID_D or MPAMn_ELx.PARTID_I field is *out-of-range*, the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The out-of-range PARTID is replaced by any in-range PARTID.
- The out-of-range PARTID is replaced by the Default physical PARTID.

For the replacement PARTID space of a replacement PARTID, see PARTID space on error.

I_{DYBVB} See also:

- PARTID virtualization, for the definition of *out-of-range*.

D21.9.2 Behavior when a valid mapping does not exist for a PARTID that is treated as virtual

R_{NHSJC} If a PARTID from an MPAMn_ELx.PARTID_D or MPAMn_ELx.PARTID_I field, or its replacement if that PARTID was *out-of-range*, is *treated as virtual*, and the corresponding MPAMVPMV_EL2.VPM_V<m> valid flag indicates that a valid mapping does not exist, the mapping for the Default virtual PARTID is used. If a valid mapping does not exist for the Default virtual PARTID, the Default physical PARTID is sent.

I_{FYYSQ} See also:

- PARTID virtualization, for the definitions of *out-of-range* and *treated as virtual*.

D21.9.3 Behavior when a PMG is out-of-range

R_{VPZPD} When a PMG from an MPAMn_ELx.PMG_D or MPAMn_ELx.PMG_I field is *out-of-range*, the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The out-of-range PMG is replaced by any in-range PMG.
- The out-of-range PMG is replaced by the Default PMG.

D21.9.4 PMG value when a Default PARTID is generated due to an MPAM error

R_{YBYMK} When the Default PARTID is generated due to an MPAM error, the PMG value is a CONSTRAINED UNPREDICTABLE choice of:

- MPAMn_ELx.PMG_D or MPAMn_ELx.PMG_I.
- The Default PMG.

D21.9.5 PARTID space on error

R_{VYHDZ} An MPAM error does not change the PARTID space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D21.10 Synchronization of MPAM System register changes

I_{YTTJQ} The rules in Synchronization requirements for AArch64 System registers apply to MPAM registers.

I_{WFCWQ} MPAM configuration is not permitted to be cached in a TLB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D22
The Scalable Matrix Extension

This chapter describes the Scalable Matrix Extension. It contains the following sections:

- Overview.
- SME traps and exceptions.
- Validity of SME and SVE state.
- Streaming execution priority.
- Floating-point behaviors in Streaming SVE mode.
- Floating-point behaviors for instructions that target the SME ZA array.
- Security and power considerations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.1 Overview

IYVCPK The *Scalable Matrix Extension* (SME) defines:

- Architectural state capable of holding two-dimensional matrix tiles.
- A *Streaming SVE processing mode*, which supports execution of SVE2 instructions with a vector length that matches the tile width.
- Instructions that accumulate the outer product of vectors into a tile.
- Load, store, and move instructions that transfer a vector to or from a tile row or column.

The extension also defines System registers and fields that identify the presence and capabilities of SME, and enable and control its behavior at each Exception level.

IPYQMB The *Scalable Matrix Extension version 2* (SME2) extends the SME architecture to increase the number of applications that can benefit from the computational efficiency of SME resources, beyond their initial focus on outer products and matrix-matrix multiplication.

SME2 adds data processing instructions with multi-vector operands and a multi-vector predication mechanism.

These include:

- Multi-vector multiply-accumulate instructions, that read SVE Z vectors and accumulate into the SME *ZA array vectors* to permit reuse of the SME outer product hardware for vector operations, including widening multiplies that accumulate into more vectors than they read.
- Multi-vector instructions that read and write multiple SVE Z vectors to preprocess inputs and post-process outputs of the multi-vector multiply-accumulate and outer product instructions.
- An alternative predication mechanism to the SVE predication mechanism, to control operations performed on multiple vector registers.

SME2 also adds:

- Compressed neural network capability using dedicated lookup table instructions and outer product instructions that support binary neural networks.
- A 512-bit architectural register, ZT0, to support the lookup table feature.

DFZHSK Unless stated otherwise, the acronym *SME* applies to all implemented versions of the Scalable Matrix Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.2 SME traps and exceptions

I_{QZNRN} This section provides additional details for SME-related exceptions. For a list of exception priorities, see R_{ZFGJP}.

R_{JZTCQ} If FEAT_SME is not implemented, then SME and SME2 instructions are UNDEFINED.

R_{GTKQD} If FEAT_SME2 is not implemented, then SME2 instructions are UNDEFINED.

R_{PLYVH} If any of the following is true, an SVE instruction is UNDEFINED:

- FEAT_SME is not implemented and FEAT_SVE is not implemented.
- FEAT_SVE is not implemented and the instruction is illegal when the PE is in *Streaming SVE mode.*

D_{DMBHW} The *SME-related instructions* are:

- SME data-processing instructions.
- `SMSTART` and `SMSTOP` instructions.
- AArch64 `MRS` and `MSR` instructions which directly access any of the SVCR, SMCR_EL1, SMCR_EL2, or SMCR_EL3 registers.

I_{BLTMB} The CPACR_EL1.SMEN, CPTR_EL2.{SMEN, TSM}, CPTR_EL3.ESM trap and enable controls apply to all SME-related instructions.

I_{RGBVJ} The SMCR_ELx.EZT0 enable controls apply to the following SME2 instructions that access the ZT0 register:

- `LDR (table)`.
- `LUTI2`, `LUTI4`.
- `MOVT`.
- `STR (table)`.
- `ZERO (table)`.

I_{XKMKY} FEAT_SME adds an SME exception syndrome in ESR_ELx to identify instructions that are trapped because of any of:

- The SME trap controls in CPACR_EL1, CPTR_EL2, and CPTR_EL3.
- The SME2 trap controls SMCR_ELx.EZT0.
- The PSTATE.SM and PSTATE.ZA modes.

I_{NWNQZ} When the PE is in *Streaming SVE mode* or when FEAT_SVE is not implemented, the CPACR_EL1.SMEN, CPTR_EL2.{SMEN, TSM}, CPTR_EL3.ESM controls configure SVE instructions to trap, and the CPACR_EL1.ZEN, CPTR_EL2.{ZEN, TZ}, CPTR_EL3.EZ controls do not cause any SVE instructions to be trapped.

I_{PKGPR} When the PE is not in *Streaming SVE mode* and FEAT_SVE is implemented, the CPACR_EL1.ZEN, CPTR_EL2.{ZEN, TZ}, CPTR_EL3.EZ controls configure SVE instructions to trap, and the CPACR_EL1.SMEN, CPTR_EL2.{SMEN, TSM}, CPTR_EL3.ESM controls do not cause any SVE instructions to be trapped.

R_{GGLFV} When an SVE `MOVPRFX` instruction predictably prefixes an SVE instruction that is *illegal* due to the current value of PSTATE.SM or PSTATE.ZA, then the execution of either of the instructions generates an exception due to SME functionality, consistent with the behaviors defined by R_{RWVTR}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.3 Validity of SME and SVE state

I_{VBJBR} The Effective value of PSTATE.SM and the value of PSTATE.ZA configure whether SME architectural state is valid and accessible.

I_{WFHKZ} CPACR_EL1.SMEN, CPTR_EL2.{TSM, SMEN}, and CPTR_EL3.ESM configure whether SME-related instructions can be executed or are trapped.

I_{KKFXN} SMCR_ELx.EZT0 configures whether SME2 instructions that access the ZT0 register can be executed or are trapped.

R_{XCCXW} The controls for trapping SME-related instructions and the controls for the validity of SME architectural state are independent.

I_{JGRTR} Because the trap and architectural state validity are controlled independently, the following scenarios are all permissible:

- Instructions trap, state invalid.
  - For example, an OS traps the first usage of SME-related instructions by a process.
- Instructions trap, state valid.
  - For example, a process was running with valid SME architectural state and an OS configures traps to detect when the next usage of SME architectural state occurs.
  - Enabling the trap does not affect or corrupt the SME architectural state.
- Instructions permitted, state invalid.
  - For example, a process is permitted to execute SME-related instructions but is currently not running in *Streaming SVE mode*. SME data-processing instructions which access SVE vector or predicate registers are *illegal* and are trapped, but SVE instructions operate on the Non-streaming SVE register state. The process can execute an `SMSTART` instruction to enter *Streaming SVE mode*.
  - For example, a process is running in *Streaming SVE mode*, but has not enabled access to the ZA storage. SME instructions that access ZA are *illegal* and are trapped, but the process can execute an `SMSTART` instruction to enable access to the ZA storage.
- Instructions permitted, state valid.
  - For example, a process is running in *Streaming SVE mode*, and has enabled access to ZA storage.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.4 Streaming execution priority

D_{DXMSW} Streaming execution refers to the execution of instructions by a PE when that PE is in *Streaming SVE mode*.

I_{CMRVS} Arm expects a variety of implementation styles for SME, including styles where more than one PE shares SME and Streaming SVE compute resources.

I_{PQNJS} Shared SME and Streaming SVE compute resources are called a *Streaming Mode Compute Unit* (SMCU).

I_{MZXWD} For implementations that share an SMCU, this architecture provides per-PE mechanisms that software can use to dynamically prioritize performance characteristics experienced by each PE.

D22.4.1 Streaming execution priority for shared implementations

I_{YYRZQ} Execution of certain instructions by a PE in *Streaming SVE mode* might experience a performance dependency on other PEs in the system that are also executing instructions in *Streaming SVE mode*. For example, this might occur when an SMCU is shared between PEs.

I_{WPQVV} The architecture provides a mechanism to control the streaming execution priority of a PE, in SMPRI_EL1. The streaming execution priority of a PE is relative to the streaming execution priority of other PEs, when a performance dependency exists between PEs executing in *Streaming SVE mode*.

I_{KTYTD} An implementation that does not share SMCUs or has no performance dependency between PEs might not need to limit or prioritize execution of one PE relative to another.

I_{HQXBH} The streaming execution priority mechanism is optional.

D_{DGRTS} All PEs that share a given SMCU form a *Priority domain*.

D_{YQFWM} Different *Priority domains* represent unrelated SMCUs.

R_{WPVQK} All PEs in a *Priority domain* have the same value of SMIDR_EL1.Affinity and SMIDR_EL1.Affinity2.

R_{CVLSF} PEs in different *Priority domains* have different values of SMIDR_EL1.Affinity or SMIDR_EL1.Affinity2.

R_{GGDRC} The streaming execution priority in SMPRI_EL1 affects execution of a PE relative to all other PEs in the same *Priority domain.*

R_{SBCRG} All SMCUs in the system have a consistent interpretation of the streaming execution priority values.

R_{RQXFC} The streaming execution priority mechanism affects the execution of instructions by a shared SMCU when the PE is in *Streaming SVE mode* and does not directly control the execution of other types of instruction.

I_{BLMYK} If system software does not support differentiation of streaming execution priority of threads, it is safe to use a value of 0 for all threads.

I_{YBQNW} The architecture considers *Priority domain* to be non-overlapping sets, meaning that in a shared-SMCU system, a PE is associated with at most one SMCU.

D22.4.1.1 Streaming execution context management

I_{PRNMJ} Arm expects that the SVE and SME instructions used by save, restore, and clear routines for the *Streaming SVE mode* SVE registers, the ZA storage, and the ZT0 register when FEAT_SME2 is implemented, are limited to using the following SME and SVE instructions:

- SME `LDR (array vector)` and `STR (array vector)` instructions.
- SME2 `LDR (table)` and `STR (table)` instructions.
- SVE `LDR (vector)` and `STR (vector)` instructions.
- SVE `LDR (predicate)` and `STR (predicate)` instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SME `ZERO (tiles)` instruction.
- SME2 `ZERO (table)` instruction.
- SVE `DUP (immediate)` instruction with zero immediate.
- SVE `PFALSE` instruction.

For implementations with a shared SMCU, PEs are expected to execute these instructions in a way that experiences a reduced effect of contention for the SMCU from other PEs, compared to other SME and SVE instructions executed in *Streaming SVE mode*.

D22.4.1.2 Streaming execution priority control

I_{RMDCP} The streaming execution priority is controlled by a 4-bit priority value. When the streaming execution priority mechanism is not supported, the priority value is ignored.

I_{FJQRG} A higher priority value corresponds to a higher streaming execution priority. Priority value `15` is the highest priority.

I_{QHKWP} The behavior of any given priority value relative to that of another PE is IMPLEMENTATION DEFINED.

D22.4.1.3 Streaming execution priority virtualization

I_{SQBCZ} The Effective streaming execution priority is either the value configured in SMPRI_EL1 or, if EL2 is implemented and enabled in the current Security state, the value of SMPRI_EL1 mapped into a new value by indexing the fields in SMPRIMAP_EL2. This choice is affected by the current Exception level, and the HCRX_EL2.SMPME configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.5 Floating-point behaviors in Streaming SVE mode

D_{DMPBW} Unless stated otherwise, the behaviors in this section apply when the PE is in Streaming SVE mode, to floating-point instructions that place their results in SIMD&FP registers or SVE scalable vector registers, except for:

- SVE BFloat16 instructions `BFDOT` and `BFMMLA`, for which, see BFloat16 behaviors for instructions that calculate sum-of-products.
- FP8 instructions, for which, see Summary of FP8 instruction behaviors.

The instructions:

R_{PHDZL}
- Honor FPCR.{DN, FZ, RMode, FZ16, AH, FIZ}, unless specified otherwise in Flushing denormalized numbers to zero or Alternate BFloat16 behaviors.

R_{GTYSK}
- Produce the expected IEEE 754 default result and update the FPSR cumulative exception flag bits, unless specified otherwise in Alternate BFloat16 behaviors.

R_{FBFNT}
- Disable trapped floating-point exceptions as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all 0, and treat the NEP element preserve control as if it is 0, if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

See also:

- Streaming SVE mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.6 Floating-point behaviors for instructions that target the SME ZA array

D_{HTZVK} Unless stated otherwise, the behaviors in this section apply to SME and SME2 floating-point instructions that place their results in the ZA array, except the SME BFloat16 instructions `BFMOPA` (widening), `BFMOPS` (widening), `BFDOT`, and `BFVDOT`, for which, see BFloat16 behaviors for instructions that calculate sum-of-products.

The instructions:

R_{WLPHB}
- Honor FPCR.{FZ, RMode, FZ16, AH, FIZ}, unless specified otherwise in Flushing denormalized numbers to zero or Alternate BFloat16 behaviors.

R_{RKHHZ}
- Generate the Default NaN, as if FPCR.DN is 1.

R_{TGSKG}
- Produce the expected IEEE 754 default result but do not update the FPSR cumulative exception flag bits.

R_{FSWYW}
- Disable trapped floating-point exceptions, as if FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are all 0.

R_{BWKKQ} The SME instructions that accumulate dot products of pairs of adjacent half-precision elements in the source vectors into single-precision elements in the ZA array:
- Perform a fused sum-of-products without rounding of the intermediate products, but rounding the single-precision sum before addition to the accumulator tile or multi-vector operand element.
- Generate the default NaN as intermediate sum-of-products when any of the following are true:
  - Any multiplier input is a NaN.
  - Any product is infinity $\times$ 0.0.
  - Both products are infinity of differing signs.
- Generate an intermediate sum-of-products of the same infinity when there are products that are infinity all with the same sign.

R_{RPSLK} If FEAT_AFP is implemented, the instructions honor FPCR.{AH, FIZ}, in which case:
- When FPCR.AH is 1, the sign bit of a generated default NaN result is set to 1 instead of 0.
- When FPCR.AH is 1 and FPCR.FZ is 1, a denormal result, detected after rounding with an unbounded exponent has been applied, is flushed to zero.
- When FPCR.AH is 1, the FPCR.FZ control does not cause denormalized inputs to be flushed to zero.
- When FPCR.FIZ is 1, all denormalized inputs are flushed to zero.

R_{TCLRM}
- Honor FPCR.RMode, supporting all four IEEE 754 rounding modes.

R_{VVVNR}
- Honor FPCR.FZ.

R_{TXKVK}
- Honor FPCR.FZ16.

R_{JRRMJ} Those instructions that multiply single elements from each source vector and accumulate their product into the ZA array perform a fused multiply-add to each accumulator tile or multi-vector operand element without rounding of the intermediate products.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D22.7 Security and power considerations

D22.7.1 Security considerations

I_{DXRGG} All SME load and store instructions adhere to the memory access permissions model in The AArch64 Virtual Memory System Architecture.

I_{MGLWR} SME architectural state can be access-controlled, meaning that higher levels of privilege can trap access to the state from the same or lower levels of privilege.

I_{CYPJJ} System software has controls available to save and restore state between unrelated pieces of software, and must ensure that steps are taken to preserve isolation and privacy.

I_{TDPHC} Operations performed in *Streaming SVE mode* respect the requirements of PSTATE.DIT. FEAT_DIT requires data-independent timing when enabled.

D22.7.2 Power considerations

I_{VGWQW} An implementation might use the activity of PSTATE.{SM, ZA} to influence the choice of power-saving states for both functional units and retention of architected state. A PE might consume less power when PSTATE.{SM, ZA} are {0, 0}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D23
AArch64 System Register Encoding

This chapter describes the AArch64 System register encoding space. It contains the following sections:

- The System register encoding space.
- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D23.1 The System register encoding space

The A64 instruction set includes instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the parts of the System register encoding space that provides access to the System registers described in AArch64 System Register Descriptions.

Note

- See Fixed values in AArch64 instruction and System register descriptions for information about abbreviations used in the System instruction descriptions.
- In AArch32 state, much of this functionality is provided through the System register interface described in The AArch32 System register interface. In AArch64 state, the parameters used to characterize the System register encoding space are {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in Background to the System register interface. There is no particular significance to the naming of these parameters, and no functional distinction between the `opn` parameters and the `CRx` parameters.

Principles of the System instruction class encoding describes some general properties of these encodings. System instruction class encoding overview then describes the top-level encoding of these instructions, identifying that:

- Entries in the encoding space are characterized by the parameter set {`op0`, `op1`, `CRn`, `CRm`, `op2`}.
- `op0` is the most significant parameter for determining allocations in this space.

Much of this encoding space is used for System instructions, as described in The A64 System Instruction Class. This chapter describes only the part of the encoding space that is used for System registers, in the following sections:

- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D23.2 Moves to and from debug and trace System registers

The instructions that move data to and from the debug, execution environment, and trace System registers are encoded with `op0`== `0b10`. This means the encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22 21	20	19 18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 L	1	0 op1	CRn	CRm	op2	Rt
op0								

Note

- The section describes the use of all of the `op0`==`0b10` region of the System register encoding space.
- These encodings access the registers that are equivalent to the AArch32 System registers in the (`coproc` == `0b1110`) encoding space.

The value of `op1` provides the next level of decode of these instructions, as follows:

`op1` == {`0b000`, `0b011`, `0b100`, `0b101`, `0b110`, `0b111`}

Debug, PMU, and System PMU registers.

Note

The standard encoding of debug System registers is `op0` == `0b10`, `op1` == {0, 3, 4}. The registers in the `op0` == `0b11` encoding space that are classified as debug System registers are DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3. See Instructions for accessing non-debug System registers for the encodings of these registers.

`op1` == {`0b001`} Trace and BRBE registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D23.2.1 Instructions for accessing debug System registers

The instructions for accessing debug System registers are:

```
MSR <System_register>, Xt     ; Write to System register
MRS Xt, <System_register>     ; Read from System register
```

Where `<System_register>` is the register name, for example MDCCSR_EL0.

This section includes only the System register access encodings for which both:

- `op0` is `0b10`.
- The value of `op1` is one of {`0b000`, `0b001`, `0b011`, `0b100`, `0b101`, `0b110`, `0b111`}.

Note

These encodings access the registers that are equivalent to the AArch32 System registers in the (`coproc` == `0b1110`) encoding space. For more information, see Mapping of the System registers between the Execution states.

Table D23-1 shows the mapping of the System register encodings for debug System register access.

Table D23-1 Instruction encodings for debug System register access in the (op0==0b10) encoding space

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0000	0b0000	0b010	RW	OSDTRRX_EL1	OSDTRRX_EL1
0b000	0b0000	0b0010	0b000	RW	MDCCINT_EL1	MDCCINT_EL1
0b000	0b0000	0b0010	0b010	RW	MDSCR_EL1	MDSCR_EL1
0b000	0b0000	0b0011	0b010	RW	OSDTRTX_EL1	OSDTRTX_EL1
0b000	0b0000	0b0100	0b010	RW	MDSELR_EL1	MDSELR_EL1
0b000	0b0000	0b0101	0b010	RW	MDSTEPOP_EL1	MDSTEPOP_EL1
0b000	0b0000	0b0110	0b010	RW	OSECCR_EL1	OSECCR_EL1
0b000	0b0000	m[3:0]	0b100	RW	DBGBVR<m>_EL1	DBGBVR<n>_EL1
0b000	0b0000	m[3:0]	0b101	RW	DBGBCR<m>_EL1	DBGBCR<n>_EL1
0b000	0b0000	m[3:0]	0b110	RW	DBGWVR<m>_EL1	DBGWVR<n>_EL1
0b000	0b0000	m[3:0]	0b111	RW	DBGWCR<m>_EL1	DBGWCR<n>_EL1
0b000	0b0001	0b0000	0b000	RO	MDRAR_EL1	MDRAR_EL1
0b000	0b0001	0b0000	0b100	WO	OSLAR_EL1	OSLAR_EL1
0b000	0b0001	0b0001	0b100	RO	OSLSR_EL1	OSLSR_EL1
0b000	0b0001	0b0011	0b100	RW	OSDLR_EL1	OSDLR_EL1
0b000	0b0001	0b0100	0b100	RW	DBGPRCR_EL1	DBGPRCR_EL1
0b000	0b0111	0b1000	0b110	RW	DBGCLAIMSET_EL1	DBGCLAIMSET_EL1
0b000	0b0111	0b1001	0b110	RW	DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1
0b000	0b0111	0b1110	0b110	RO	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1
0b000	0b1001	0b1101	0b00:m[0]	RO	SPMCGCR<m>_EL1	SPMCGCR<n>_EL1
0b000	0b1001	0b1101	0b011	RW	SPMACCESSR_EL1	SPMACCESSR_EL1 SPMACCESSR_EL2
0b000	0b1001	0b1101	0b100	RO	SPMIIDR_EL1	SPMIIDR_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b1001	0b1101	0b101	RO	SPMDEVARCH_EL1	SPMDEVARCH_EL1
0b000	0b1001	0b1101	0b110	RO	SPMDEVAFF_EL1	SPMDEVAFF_EL1
0b000	0b1001	0b1101	0b111	RO	SPMCFGR_EL1	SPMCFGR_EL1
0b000	0b1001	0b1110	0b001	RW	SPMINTENSET_EL1	SPMINTENSET_EL1
0b000	0b1001	0b1110	0b010	RW	SPMINTENCLR_EL1	SPMINTENCLR_EL1
0b000	0b1110	0b1011	0b111	RO	PMCCNTSVR_EL1	PMCCNTSVR_EL1
0b000	0b1110	0b10:m[4:3]	m[2:0]	RO	PMEVCNTSVR<m>_EL1	PMEVCNTSVR<n>_EL1
0b000	0b1110	0b1100	0b000	RO	PMICNTSVR_EL1	PMICNTSVR_EL1
0b001	0b0000	0b0000	0b001	RW	TRCTRACEIDR	TRCTRACEIDR
0b001	0b0000	0b0000	0b010	RW	TRCVICTLR	TRCVICTLR
0b001	0b0000	0b0000	0b110	RO	TRCIDR8	TRCIDR8
0b001	0b0000	0b0000	0b111	RW	TRCIMSPEC0	TRCIMSPEC0
0b001	0b0000	0b0001	0b000	RW	TRCPRGCTLR	TRCPRGCTLR
0b001	0b0000	0b0001	0b001	RW	TRCQCTLR	TRCQCTLR
0b001	0b0000	0b0001	0b010	RW	TRCVIIECTLR	TRCVIIECTLR
0b001	0b0000	0b0001	0b110	RO	TRCIDR9	TRCIDR9
0b001	0b0000	0b0010	0b001	RW	TRCITEEDCR	TRCITEEDCR
0b001	0b0000	0b0010	0b010	RW	TRCVISSCTLR	TRCVISSCTLR
0b001	0b0000	0b0010	0b110	RO	TRCIDR10	TRCIDR10
0b001	0b0000	0b0011	0b000	RO	TRCSTATR	TRCSTATR
0b001	0b0000	0b0011	0b010	RW	TRCVIPCSSCTLR	TRCVIPCSSCTLR
0b001	0b0000	0b0011	0b110	RO	TRCIDR11	TRCIDR11
0b001	0b0000	0b00:m[1:0]	0b100	RW	TRCSEQEVR<m>	TRCSEQEVR<n>
0b001	0b0000	0b00:m[1:0]	0b101	RW	TRCCNTRLDVR<m>	TRCCNTRLDVR<n>
0b001	0b0000	0b0100	0b000	RW	TRCCONFIGR	TRCCONFIGR
0b001	0b0000	0b0100	0b110	RO	TRCIDR12	TRCIDR12
0b001	0b0000	0b0101	0b110	RO	TRCIDR13	TRCIDR13
0b001	0b0000	0b0110	0b000	RW	TRCAUXCTLR	TRCAUXCTLR
0b001	0b0000	0b0110	0b100	RW	TRCSEQRSTEVR	TRCSEQRSTEVR
0b001	0b0000	0b0111	0b100	RW	TRCSEQSTR	TRCSEQSTR
0b001	0b0000	0b01:m[1:0]	0b101	RW	TRCCNTCTLR<m>	TRCCNTCTLR<n>
0b001	0b0000	0b0:m[2:0]	0b111	RW	TRCIMSPEC<m>	TRCIMSPEC<n>
0b001	0b0000	0b1000	0b000	RW	TRCEVENTCTL0R	TRCEVENTCTL0R
0b001	0b0000	0b1000	0b111	RO	TRCIDR0	TRCIDR0
0b001	0b0000	0b1001	0b000	RW	TRCEVENTCTL1R	TRCEVENTCTL1R
0b001	0b0000	0b1001	0b111	RO	TRCIDR1	TRCIDR1
0b001	0b0000	0b1010	0b000	RW	TRCRSR	TRCRSR
0b001	0b0000	0b1010	0b111	RO	TRCIDR2	TRCIDR2
0b001	0b0000	0b1011	0b000	RW	TRCSTALLCTLR	TRCSTALLCTLR
0b001	0b0000	0b1011	0b111	RO	TRCIDR3	TRCIDR3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b001	0b0000	0b10:m[1:0]	0b100	RW	TRCEXTINSELR<m>	TRCEXTINSELR<n>
0b001	0b0000	0b10:m[1:0]	0b101	RW	TRCCNTVR<m>	TRCCNTVR<n>
0b001	0b0000	0b1100	0b000	RW	TRCTSCTLR	TRCTSCTLR
0b001	0b0000	0b1100	0b111	RO	TRCIDR4	TRCIDR4
0b001	0b0000	0b1101	0b000	RW	TRCSYNCPR	TRCSYNCPR
0b001	0b0000	0b1101	0b111	RO	TRCIDR5	TRCIDR5
0b001	0b0000	0b1110	0b000	RW	TRCCCCTLR	TRCCCCTLR
0b001	0b0000	0b1110	0b111	RO	TRCIDR6	TRCIDR6
0b001	0b0000	0b1111	0b000	RW	TRCBBCTLR	TRCBBCTLR
0b001	0b0000	0b1111	0b111	RO	TRCIDR7	TRCIDR7
0b001	0b0001	0b0001	0b100	RO	TRCOSLSR	TRCOSLSR
0b001	0b0001	0b0:m[2:0]	0b010	RW	TRCSSCCR<m>	TRCSSCCR<n>
0b001	0b0001	0b0:m[2:0]	0b011	RW	TRCSSPCICR<m>	TRCSSPCICR<n>
0b001	0b0001	0b1:m[2:0]	0b010	RW	TRCSSCSR<m>	TRCSSCSR<n>
0b001	0b0001	m[3:0]	0b00:m[4]	RW	TRCRSCTLR<m>	TRCRSCTLR<n>
0b001	0b0010	m[2:0]:0b0	0b00:m[3]	RW	TRCACVR<m>	TRCACVR<n>
0b001	0b0010	m[2:0]:0b0	0b01:m[3]	RW	TRCACATR<m>	TRCACATR<n>
0b001	0b0011	0b0000	0b010	RW	TRCCIDCCTLR0	TRCCIDCCTLR0
0b001	0b0011	0b0001	0b010	RW	TRCCIDCCTLR1	TRCCIDCCTLR1
0b001	0b0011	0b0010	0b010	RW	TRCVMIDCCTLR0	TRCVMIDCCTLR0
0b001	0b0011	0b0011	0b010	RW	TRCVMIDCCTLR1	TRCVMIDCCTLR1
0b001	0b0011	m[2:0]:0b0	0b000	RW	TRCCIDCVR<m>	TRCCIDCVR<n>
0b001	0b0011	m[2:0]:0b0	0b001	RW	TRCVMIDCVR<m>	TRCVMIDCVR<n>
0b001	0b0111	0b0010	0b111	RO	TRCDEVID	TRCDEVID
0b001	0b0111	0b1000	0b110	RW	TRCCLAIMSET	TRCCLAIMSET
0b001	0b0111	0b1001	0b110	RW	TRCCLAIMCLR	TRCCLAIMCLR
0b001	0b0111	0b1110	0b110	RO	TRCAUTHSTATUS	TRCAUTHSTATUS
0b001	0b0111	0b1111	0b110	RO	TRCDEVARCH	TRCDEVARCH
0b001	0b1000	m[3:0]	m[4]:0b00	RO	BRBINF<m>_EL1	BRBINF<n>_EL1
0b001	0b1000	m[3:0]	m[4]:0b01	RO	BRBSRC<m>_EL1	BRBSRC<n>_EL1
0b001	0b1000	m[3:0]	m[4]:0b10	RO	BRBTGT<m>_EL1	BRBTGT<n>_EL1
0b001	0b1001	0b0000	0b000	RW	BRBCR_EL1	BRBCR_EL1 BRBCR_EL2
0b001	0b1001	0b0000	0b001	RW	BRBFCR_EL1	BRBFCR_EL1
0b001	0b1001	0b0000	0b010	RW	BRBTS_EL1	BRBTS_EL1
0b001	0b1001	0b0001	0b000	RW	BRBINFINJ_EL1	BRBINFINJ_EL1
0b001	0b1001	0b0001	0b001	RW	BRBSRCINJ_EL1	BRBSRCINJ_EL1
0b001	0b1001	0b0001	0b010	RW	BRBTGTINJ_EL1	BRBTGTINJ_EL1
0b001	0b1001	0b0010	0b000	RO	BRBIDR0_EL1	BRBIDR0_EL1
0b011	0b0000	0b0001	0b000	RO	MDCCSR_EL0	MDCCSR_EL0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b011	0b0000	0b0100	0b000	RW	DBGDTR_EL0	DBGDTR_EL0
0b011	0b0000	0b0101	0b000	RO	DBGDTRRX_EL0	DBGDTRRX_EL0
0b011	0b0000	0b0101	0b000	WO	DBGDTRTX_EL0	DBGDTRTX_EL0
0b011	0b1001	0b1100	0b000	RW	SPMCR_EL0	SPMCR_EL0
0b011	0b1001	0b1100	0b001	RW	SPMCNTENSET_EL0	SPMCNTENSET_EL0
0b011	0b1001	0b1100	0b010	RW	SPMCNTENCLR_EL0	SPMCNTENCLR_EL0
0b011	0b1001	0b1100	0b011	RW	SPMOVSCLR_EL0	SPMOVSCLR_EL0
0b011	0b1001	0b1100	0b100	WO	SPMZR_EL0	SPMZR_EL0
0b011	0b1001	0b1100	0b101	RW	SPMSELR_EL0	SPMSELR_EL0
0b011	0b1001	0b1110	0b011	RW	SPMOVSSET_EL0	SPMOVSSET_EL0
0b011	0b1110	0b000:m[3]	m[2:0]	RW	SPMEVCNTR<m>_EL0	SPMEVCNTR<n>_EL0
0b011	0b1110	0b001:m[3]	m[2:0]	RW	SPMEVTYPER<m>_EL0	SPMEVTYPER<n>_EL0
0b011	0b1110	0b010:m[3]	m[2:0]	RW	SPMEVFILTR<m>_EL0	SPMEVFILTR<n>_EL0
0b011	0b1110	0b011:m[3]	m[2:0]	RW	SPMEVFILT2R<m>_EL0	SPMEVFILT2R<n>_EL0
0b100	0b0000	0b0111	0b000	RW	DBGVCR32_EL2	DBGVCR32_EL2
0b100	0b1001	0b0000	0b000	RW	BRBCR_EL2	BRBCR_EL2
0b100	0b1001	0b1101	0b011	RW	SPMACCESSR_EL2	SPMACCESSR_EL2
0b101	0b1001	0b0000	0b000	RW	BRBCR_EL12	BRBCR_EL1
0b101	0b1001	0b1101	0b011	RW	SPMACCESSR_EL12	SPMACCESSR_EL1
0b110	0b1001	0b1101	0b011	RW	SPMACCESSR_EL3	SPMACCESSR_EL3
0b110	0b1001	0b1110	0b111	RW	SPMROOTCR_EL3	SPMROOTCR_EL3
0b111	0b1001	0b1110	0b111	RW	SPMSCR_EL1	SPMSCR_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D23.3 Moves to and from non-debug System registers, Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with `op0== 0b11`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	CRn	CRm	op2	Rt
				op0					

The value of `CRn` provides the next level of decode of these instructions, as follows:

`CRn` == {`0b0000`, `0b0001`, `0b0010`, `0b0011`, `0b0101`, `0b0110`, `0b0111`, `0b1001`, `0b1010`, `0b1100`, `0b1101`, `0b1110`}

See Instructions for accessing non-debug System registers.

`CRn` == {`0b0100`}

See Instructions for accessing Special-purpose registers.

`CRn`=={`0b1011`, `0b1111`}

See Reserved encodings for IMPLEMENTATION DEFINED registers.

D23.3.1 Instructions for accessing non-debug System registers

The A64 instructions for accessing System registers are:

```
MSR <System_register>, Xt     ; Write to 64-bit System register
MSRR <System_register>, Xt    ; Write to 128-bit System register
MRS Xt, <System_register>     ; Read from 64-bit System register
MRRS Xt, <System_register>    ; Read from 128-bit System register
```

Where `<System_register>` is the register name, for example MIDR_EL1.

This section includes only the System register access encodings for which both:

- `op0` is `0b11`.
- The value of `CRn` is one of {`0b0000`, `0b0001`, `0b0010`, `0b0011`, `0b0100`, `0b0101`, `0b0110`, `0b0111`, `0b1001`, `0b1010`, `0b1100`, `0b1101`, `0b1110`}.

Note

- These encodings access the registers that are equivalent to the AArch32 System registers in the (`coproc` == `0b1111`) encoding space. For more information, see Mapping of the System registers between the Execution states.
- While this group is described as accessing the non-debug System registers, its correct characterization is by the {`op0`, `CRn`} values given in this subsection, and the group includes the debug System registers DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3, which are described in Debug System registers. These registers are exceptions to the standard encoding of debug System registers, which has `op0` == `0b10`. See Instructions for accessing debug System registers for more information.

Direct accesses to some regions of the system register encoding space have a defined behavior even if no register is currently allocated or implemented for the encodings in those regions. The Feature ID space is the System register space in AArch64 with `op0==3, op1={0, 1, 3}, CRn==0, CRm={0-7}, op2={0-7}`. The properties for this space are that:

- For unused encodings in the part of the space that has `op1` == `0` and `CRm` == `{2-7}`, the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If FEAT_FGT is not implemented, it is IMPLEMENTATION DEFINED whether direct reads can be trapped by HCR_EL2.TID3.
— If FEAT_FGT is implemented, direct reads can be trapped by HCR_EL2.TID3.
— If FEAT_IDTE3 is implemented, direct reads can be trapped by SCR_EL3.TID3.
— Direct reads that are not trapped behave as Reserved, RES0 to ensure correct behavior if the encodings are used for ID registers in the future.

- For unused encodings in the part of the space that has either `op1 == 0` and `CRm == 0`, or `op1 == {1, 3}`, the following apply:
  — If FEAT_IDST is not implemented, direct reads are UNDEFINED and therefore generate exceptions reported with EC syndrome value `0x00`.
  — If FEAT_IDST is not implemented, direct reads are UNDEFINED and therefore generate exceptions reported with EC syndrome value `0x00`.
  — If FEAT_IDST is implemented, direct reads generate exceptions reported with EC syndrome value `0x18`.
  — The behavior for direct reads is consistent with the pseudocode function `UnimplementedIDRegister()`.

The instruction encoding for these accesses is:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	CRn	CRm	op2	Rt
				op0					

See text for permitted values of CRn

Table D23-2 shows the encodings of the register access instructions.

See the register descriptions for information about the control that determines whether these accesses are permitted.

Table D23-2 Instruction encodings for non-debug System register access in the (op0==0b11) encoding space

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0000	0b0000	0b000	RO	MIDR_EL1	MIDR_EL1 VPIDR_EL2
0b000	0b0000	0b0000	0b101	RO	MPIDR_EL1	MPIDR_EL1 VMPIDR_EL2
0b000	0b0000	0b0000	0b110	RO	REVIDR_EL1	REVIDR_EL1
0b000	0b0000	0b0001	0b000	RO	ID_PFR0_EL1	ID_PFR0_EL1
0b000	0b0000	0b0001	0b001	RO	ID_PFR1_EL1	ID_PFR1_EL1
0b000	0b0000	0b0001	0b010	RO	ID_DFR0_EL1	ID_DFR0_EL1
0b000	0b0000	0b0001	0b011	RO	ID_AFR0_EL1	ID_AFR0_EL1
0b000	0b0000	0b0001	0b100	RO	ID_MMFR0_EL1	ID_MMFR0_EL1
0b000	0b0000	0b0001	0b101	RO	ID_MMFR1_EL1	ID_MMFR1_EL1
0b000	0b0000	0b0001	0b110	RO	ID_MMFR2_EL1	ID_MMFR2_EL1
0b000	0b0000	0b0001	0b111	RO	ID_MMFR3_EL1	ID_MMFR3_EL1
0b000	0b0000	0b0010	0b000	RO	ID_ISAR0_EL1	ID_ISAR0_EL1
0b000	0b0000	0b0010	0b001	RO	ID_ISAR1_EL1	ID_ISAR1_EL1
0b000	0b0000	0b0010	0b010	RO	ID_ISAR2_EL1	ID_ISAR2_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0000	0b0010	0b011	RO	ID_ISAR3_EL1	ID_ISAR3_EL1
0b000	0b0000	0b0010	0b100	RO	ID_ISAR4_EL1	ID_ISAR4_EL1
0b000	0b0000	0b0010	0b101	RO	ID_ISAR5_EL1	ID_ISAR5_EL1
0b000	0b0000	0b0010	0b110	RO	ID_MMFR4_EL1	ID_MMFR4_EL1
0b000	0b0000	0b0010	0b111	RO	ID_ISAR6_EL1	ID_ISAR6_EL1
0b000	0b0000	0b0011	0b000	RO	MVFR0_EL1	MVFR0_EL1
0b000	0b0000	0b0011	0b001	RO	MVFR1_EL1	MVFR1_EL1
0b000	0b0000	0b0011	0b010	RO	MVFR2_EL1	MVFR2_EL1
0b000	0b0000	0b0011	0b100	RO	ID_PFR2_EL1	ID_PFR2_EL1
0b000	0b0000	0b0011	0b101	RO	ID_DFR1_EL1	ID_DFR1_EL1
0b000	0b0000	0b0011	0b110	RO	ID_MMFR5_EL1	ID_MMFR5_EL1
0b000	0b0000	0b0100	0b000	RO	ID_AA64PFR0_EL1	ID_AA64PFR0_EL1
0b000	0b0000	0b0100	0b001	RO	ID_AA64PFR1_EL1	ID_AA64PFR1_EL1
0b000	0b0000	0b0100	0b010	RO	ID_AA64PFR2_EL1	ID_AA64PFR2_EL1
0b000	0b0000	0b0100	0b100	RO	ID_AA64ZFR0_EL1	ID_AA64ZFR0_EL1
0b000	0b0000	0b0100	0b101	RO	ID_AA64SMFR0_EL1	ID_AA64SMFR0_EL1
0b000	0b0000	0b0100	0b111	RO	ID_AA64FPFR0_EL1	ID_AA64FPFR0_EL1
0b000	0b0000	0b0101	0b000	RO	ID_AA64DFR0_EL1	ID_AA64DFR0_EL1
0b000	0b0000	0b0101	0b001	RO	ID_AA64DFR1_EL1	ID_AA64DFR1_EL1
0b000	0b0000	0b0101	0b010	RO	ID_AA64DFR2_EL1	ID_AA64DFR2_EL1
0b000	0b0000	0b0101	0b100	RO	ID_AA64AFR0_EL1	ID_AA64AFR0_EL1
0b000	0b0000	0b0101	0b101	RO	ID_AA64AFR1_EL1	ID_AA64AFR1_EL1
0b000	0b0000	0b0110	0b000	RO	ID_AA64ISAR0_EL1	ID_AA64ISAR0_EL1
0b000	0b0000	0b0110	0b001	RO	ID_AA64ISAR1_EL1	ID_AA64ISAR1_EL1
0b000	0b0000	0b0110	0b010	RO	ID_AA64ISAR2_EL1	ID_AA64ISAR2_EL1
0b000	0b0000	0b0110	0b011	RO	ID_AA64ISAR3_EL1	ID_AA64ISAR3_EL1
0b000	0b0000	0b0111	0b000	RO	ID_AA64MMFR0_EL1	ID_AA64MMFR0_EL1
0b000	0b0000	0b0111	0b001	RO	ID_AA64MMFR1_EL1	ID_AA64MMFR1_EL1
0b000	0b0000	0b0111	0b010	RO	ID_AA64MMFR2_EL1	ID_AA64MMFR2_EL1
0b000	0b0000	0b0111	0b011	RO	ID_AA64MMFR3_EL1	ID_AA64MMFR3_EL1
0b000	0b0000	0b0111	0b100	RO	ID_AA64MMFR4_EL1	ID_AA64MMFR4_EL1
0b000	0b0001	0b0000	0b000	RW	SCTLR_EL1	SCTLR_EL1 SCTLR_EL2
0b000	0b0001	0b0000	0b001	RW	ACTLR_EL1	ACTLR_EL1 ACTLR_EL2
0b000	0b0001	0b0000	0b010	RW	CPACR_EL1	CPACR_EL1 CPTR_EL2
0b000	0b0001	0b0000	0b011	RW	SCTLR2_EL1	SCTLR2_EL1 SCTLR2_EL2
0b000	0b0001	0b0000	0b101	RW	RGSR_EL1	RGSR_EL1
0b000	0b0001	0b0000	0b110	RW	GCR_EL1	GCR_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0001	0b0010	0b000	RW	ZCR_EL1	ZCR_EL1 ZCR_EL2
0b000	0b0001	0b0010	0b001	RW	TRFCR_EL1	TRFCR_EL1 TRFCR_EL2
0b000	0b0001	0b0010	0b011	RW	TRCITECR_EL1	TRCITECR_EL1 TRCITECR_EL2
0b000	0b0001	0b0010	0b100	RW	SMPRI_EL1	SMPRI_EL1
0b000	0b0001	0b0010	0b110	RW	SMCR_EL1	SMCR_EL1 SMCR_EL2
0b000	0b0001	0b0100	0b000	RW	SCTLRMASK_EL1	SCTLRMASK_EL1 SCTLRMASK_EL2
0b000	0b0001	0b0100	0b001	RW	ACTLRMASK_EL1	ACTLRMASK_EL1 ACTLRMASK_EL2
0b000	0b0001	0b0100	0b010	RW	CPACRMASK_EL1	CPACRMASK_EL1 CPTRMASK_EL2
0b000	0b0001	0b0100	0b011	RW	SCTLR2MASK_EL1	SCTLR2MASK_EL1 SCTLR2MASK_EL2
0b000	0b0001	0b0100	0b100	RW	CPACRALIAS_EL1	CPACR_EL1
0b000	0b0001	0b0100	0b101	RW	ACTLRALIAS_EL1	ACTLR_EL1
0b000	0b0001	0b0100	0b110	RW	SCTLRALIAS_EL1	SCTLR_EL1
0b000	0b0001	0b0100	0b111	RW	SCTLR2ALIAS_EL1	SCTLR2_EL1
0b000	0b0010	0b0000	0b000	RW	TTBR0_EL1	TTBR0_EL1 TTBR0_EL2
0b000	0b0010	0b0000	0b000	RW	TTBR0_EL1	TTBR0_EL1 TTBR0_EL2
0b000	0b0010	0b0000	0b001	RW	TTBR1_EL1	TTBR1_EL1 TTBR1_EL2
0b000	0b0010	0b0000	0b001	RW	TTBR1_EL1	TTBR1_EL1 TTBR1_EL2
0b000	0b0010	0b0000	0b010	RW	TCR_EL1	TCR_EL1 TCR_EL2
0b000	0b0010	0b0000	0b011	RW	TCR2_EL1	TCR2_EL1 TCR2_EL2
0b000	0b0010	0b0001	0b000	RW	APIAKeyLo_EL1	APIAKeyLo_EL1
0b000	0b0010	0b0001	0b001	RW	APIAKeyHi_EL1	APIAKeyHi_EL1
0b000	0b0010	0b0001	0b010	RW	APIBKeyLo_EL1	APIBKeyLo_EL1
0b000	0b0010	0b0001	0b011	RW	APIBKeyHi_EL1	APIBKeyHi_EL1
0b000	0b0010	0b0010	0b000	RW	APDAKeyLo_EL1	APDAKeyLo_EL1
0b000	0b0010	0b0010	0b001	RW	APDAKeyHi_EL1	APDAKeyHi_EL1
0b000	0b0010	0b0010	0b010	RW	APDBKeyLo_EL1	APDBKeyLo_EL1
0b000	0b0010	0b0010	0b011	RW	APDBKeyHi_EL1	APDBKeyHi_EL1
0b000	0b0010	0b0011	0b000	RW	APGAKeyLo_EL1	APGAKeyLo_EL1
0b000	0b0010	0b0011	0b001	RW	APGAKeyHi_EL1	APGAKeyHi_EL1
0b000	0b0010	0b0101	0b000	RW	GCSCR_EL1	GCSCR_EL1 GCSCR_EL2
0b000	0b0010	0b0101	0b001	RW	GCSPR_EL1	GCSPR_EL1 GCSPR_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0010	0b0101	0b010	RW	GCSCRE0_EL1	GCSCRE0_EL1
0b000	0b0010	0b0111	0b010	RW	TCRMASK_EL1	TCRMASK_EL1 TCRMASK_EL2
0b000	0b0010	0b0111	0b011	RW	TCR2MASK_EL1	TCR2MASK_EL1 TCR2MASK_EL2
0b000	0b0010	0b0111	0b110	RW	TCRALIAS_EL1	TCR_EL1
0b000	0b0010	0b0111	0b111	RW	TCR2ALIAS_EL1	TCR2_EL1
0b000	0b0100	0b0000	0b000	RW	SPSR_EL1	SPSR_EL1 SPSR_EL2
0b000	0b0100	0b0000	0b001	RW	ELR_EL1	ELR_EL1 ELR_EL2
0b000	0b0100	0b0001	0b000	RW	SP_EL0	SP_EL0
0b000	0b0100	0b0010	0b000	RW	SPSel	SPSel
0b000	0b0100	0b0010	0b010	RO	CurrentEL	CurrentEL
0b000	0b0100	0b0010	0b011	RW	PAN	PAN
0b000	0b0100	0b0010	0b100	RW	UAO	UAO
0b000	0b0100	0b0011	0b000	RW	ALLINT	ALLINT
0b000	0b0100	0b0011	0b001	RW	PM	PM
0b000	0b0100	0b0110	0b000	RW	ICC_PMR_EL1	ICC_PMR_EL1 ICV_PMR_EL1
0b000	0b0101	0b0001	0b000	RW	AFSR0_EL1	AFSR0_EL1 AFSR0_EL2
0b000	0b0101	0b0001	0b001	RW	AFSR1_EL1	AFSR1_EL1 AFSR1_EL2
0b000	0b0101	0b0010	0b000	RW	ESR_EL1	ESR_EL1 ESR_EL2
0b000	0b0101	0b0011	0b000	RO	ERRIDR_EL1	ERRIDR_EL1
0b000	0b0101	0b0011	0b001	RW	ERRSELR_EL1	ERRSELR_EL1
0b000	0b0101	0b0011	0b010	RO	ERXGSR_EL1	ERXGSR_EL1
0b000	0b0101	0b0100	0b000	RO	ERXFR_EL1	ERXFR_EL1
0b000	0b0101	0b0100	0b001	RW	ERXCTLR_EL1	ERXCTLR_EL1
0b000	0b0101	0b0100	0b010	RW	ERXSTATUS_EL1	ERXSTATUS_EL1
0b000	0b0101	0b0100	0b011	RW	ERXADDR_EL1	ERXADDR_EL1
0b000	0b0101	0b0100	0b100	RO	ERXPFGF_EL1	ERXPFGF_EL1
0b000	0b0101	0b0100	0b101	RW	ERXPFGCTL_EL1	ERXPFGCTL_EL1
0b000	0b0101	0b0100	0b110	RW	ERXPFGCDN_EL1	ERXPFGCDN_EL1
0b000	0b0101	0b0101	0b000	RW	ERXMISC0_EL1	ERXMISC0_EL1
0b000	0b0101	0b0101	0b001	RW	ERXMISC1_EL1	ERXMISC1_EL1
0b000	0b0101	0b0101	0b010	RW	ERXMISC2_EL1	ERXMISC2_EL1
0b000	0b0101	0b0101	0b011	RW	ERXMISC3_EL1	ERXMISC3_EL1
0b000	0b0101	0b0110	0b000	RW	TFSR_EL1	TFSR_EL1 TFSR_EL2
0b000	0b0101	0b0110	0b001	RW	TFSRE0_EL1	TFSRE0_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0110	0b0000	0b000	RW	FAR_EL1	FAR_EL1 FAR_EL2
0b000	0b0110	0b0000	0b101	RW	PFAR_EL1	PFAR_EL1
0b000	0b0111	0b0100	0b000	RW	PAR_EL1	PAR_EL1
0b000	0b0111	0b0100	0b000	RW	PAR_EL1	PAR_EL1
0b000	0b1001	0b1001	0b000	RW	PMSCR_EL1	PMSCR_EL1 PMSCR_EL2
0b000	0b1001	0b1001	0b001	RW	PMSNEVFR_EL1	PMSNEVFR_EL1
0b000	0b1001	0b1001	0b010	RW	PMSICR_EL1	PMSICR_EL1
0b000	0b1001	0b1001	0b011	RW	PMSIRR_EL1	PMSIRR_EL1
0b000	0b1001	0b1001	0b100	RW	PMSFCR_EL1	PMSFCR_EL1
0b000	0b1001	0b1001	0b101	RW	PMSEVFR_EL1	PMSEVFR_EL1
0b000	0b1001	0b1001	0b110	RW	PMSLATFR_EL1	PMSLATFR_EL1
0b000	0b1001	0b1001	0b111	RO	PMSIDR_EL1	PMSIDR_EL1
0b000	0b1001	0b1010	0b000	RW	PMBLIMITR_EL1	PMBLIMITR_EL1
0b000	0b1001	0b1010	0b001	RW	PMBPTR_EL1	PMBPTR_EL1
0b000	0b1001	0b1010	0b011	RW	PMBSR_EL1	PMBSR_EL1 PMBSR_EL2
0b000	0b1001	0b1010	0b100	RW	PMSDSFR_EL1	PMSDSFR_EL1
0b000	0b1001	0b1010	0b101	RW	PMBMAR_EL1	PMBMAR_EL1
0b000	0b1001	0b1010	0b111	RO	PMBIDR_EL1	PMBIDR_EL1
0b000	0b1001	0b1011	0b000	RW	TRBLIMITR_EL1	TRBLIMITR_EL1
0b000	0b1001	0b1011	0b001	RW	TRBPTR_EL1	TRBPTR_EL1
0b000	0b1001	0b1011	0b010	RW	TRBBASER_EL1	TRBBASER_EL1
0b000	0b1001	0b1011	0b011	RW	TRBSR_EL1	TRBSR_EL1 TRBSR_EL2
0b000	0b1001	0b1011	0b100	RW	TRBMAR_EL1	TRBMAR_EL1
0b000	0b1001	0b1011	0b101	RW	TRBMPAM_EL1	TRBMPAM_EL1
0b000	0b1001	0b1011	0b110	RW	TRBTRG_EL1	TRBTRG_EL1
0b000	0b1001	0b1011	0b111	RO	TRBIDR_EL1	TRBIDR_EL1
0b000	0b1001	0b1101	0b011	RW	PMSSCR_EL1	PMSSCR_EL1
0b000	0b1001	0b1110	0b001	RW	PMINTENSET_EL1	PMINTENSET_EL1
0b000	0b1001	0b1110	0b010	RW	PMINTENCLR_EL1	PMINTENCLR_EL1
0b000	0b1001	0b1110	0b100	RW	PMUACR_EL1	PMUACR_EL1
0b000	0b1001	0b1110	0b101	RW	PMECR_EL1	PMECR_EL1
0b000	0b1001	0b1110	0b110	RO	PMMIR_EL1	PMMIR_EL1
0b000	0b1010	0b0010	0b000	RW	MAIR_EL1	MAIR_EL1 MAIR_EL2
0b000	0b1010	0b0010	0b001	RW	MAIR2_EL1	MAIR2_EL1 MAIR2_EL2
0b000	0b1010	0b0010	0b010	RW	PIRE0_EL1	PIRE0_EL1 PIRE0_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b1010	0b0010	0b011	RW	PIR_EL1	PIR_EL1 PIR_EL2
0b000	0b1010	0b0010	0b100	RW	POR_EL1	POR_EL1 POR_EL2
0b000	0b1010	0b0010	0b101	RW	S2POR_EL1	S2POR_EL1
0b000	0b1010	0b0011	0b000	RW	AMAIR_EL1	AMAIR_EL1 AMAIR_EL2
0b000	0b1010	0b0011	0b001	RW	AMAIR2_EL1	AMAIR2_EL1 AMAIR2_EL2
0b000	0b1010	0b0100	0b000	RW	LORSA_EL1	LORSA_EL1
0b000	0b1010	0b0100	0b001	RW	LOREA_EL1	LOREA_EL1
0b000	0b1010	0b0100	0b010	RW	LORN_EL1	LORN_EL1
0b000	0b1010	0b0100	0b011	RW	LORC_EL1	LORC_EL1
0b000	0b1010	0b0100	0b100	RO	MPAMIDR_EL1	MPAMIDR_EL1
0b000	0b1010	0b0100	0b101	RO	MPAMBWIDR_EL1	MPAMBWIDR_EL1
0b000	0b1010	0b0100	0b111	RO	LORID_EL1	LORID_EL1
0b000	0b1010	0b0101	0b000	RW	MPAM1_EL1	MPAM1_EL1 MPAM2_EL2
0b000	0b1010	0b0101	0b001	RW	MPAM0_EL1	MPAM0_EL1
0b000	0b1010	0b0101	0b011	RW	MPAMSM_EL1	MPAMSM_EL1
0b000	0b1010	0b0101	0b100	RW	MPAMBW1_EL1	MPAMBW1_EL1 MPAMBW2_EL2
0b000	0b1010	0b0101	0b101	RW	MPAMBW0_EL1	MPAMBW0_EL1
0b000	0b1010	0b0101	0b111	RW	MPAMBWSM_EL1	MPAMBWSM_EL1
0b000	0b1100	0b0000	0b000	RW	VBAR_EL1	VBAR_EL1 VBAR_EL2
0b000	0b1100	0b0000	0b001	RO	RVBAR_EL1	RVBAR_EL1
0b000	0b1100	0b0000	0b010	RW	RMR_EL1	RMR_EL1
0b000	0b1100	0b0001	0b000	RO	ISR_EL1	ISR_EL1
0b000	0b1100	0b0001	0b001	RW	DISR_EL1	DISR_EL1 VDISR_EL2 VDISR_EL3
0b000	0b1100	0b1000	0b000	RO	ICC_IAR0_EL1	ICC_IAR0_EL1 ICV_IAR0_EL1
0b000	0b1100	0b1000	0b001	WO	ICC_EOIR0_EL1	ICC_EOIR0_EL1 ICV_EOIR0_EL1
0b000	0b1100	0b1000	0b010	RO	ICC_HPPIR0_EL1	ICC_HPPIR0_EL1 ICV_HPPIR0_EL1
0b000	0b1100	0b1000	0b011	RW	ICC_BPR0_EL1	ICC_BPR0_EL1 ICV_BPR0_EL1
0b000	0b1100	0b1000	0b1:m[1:0]	RW	ICC_AP0R<m>_EL1	ICC_AP0R<n>_EL1 ICV_AP0R<n>_EL1
0b000	0b1100	0b1001	0b0:m[1:0]	RW	ICC_AP1R<m>_EL1	ICC_AP1R<n>_EL1 ICV_AP1R<n>_EL1
0b000	0b1100	0b1001	0b101	RO	ICC_NMIAR1_EL1	ICC_NMIAR1_EL1 ICV_NMIAR1_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b1100	0b1011	0b001	WO	ICC_DIR_EL1	ICC_DIR_EL1 ICV_DIR_EL1
0b000	0b1100	0b1011	0b011	RO	ICC_RPR_EL1	ICC_RPR_EL1 ICV_RPR_EL1
0b000	0b1100	0b1011	0b101	WO	ICC_SGI1R_EL1	ICC_SGI1R_EL1
0b000	0b1100	0b1011	0b110	WO	ICC_ASGI1R_EL1	ICC_ASGI1R_EL1
0b000	0b1100	0b1011	0b111	WO	ICC_SGI0R_EL1	ICC_SGI0R_EL1
0b000	0b1100	0b1100	0b000	RO	ICC_IAR1_EL1	ICC_IAR1_EL1 ICV_IAR1_EL1
0b000	0b1100	0b1100	0b001	WO	ICC_EOIR1_EL1	ICC_EOIR1_EL1 ICV_EOIR1_EL1
0b000	0b1100	0b1100	0b010	RO	ICC_HPPIR1_EL1	ICC_HPPIR1_EL1 ICV_HPPIR1_EL1
0b000	0b1100	0b1100	0b011	RW	ICC_BPR1_EL1	ICC_BPR1_EL1 ICV_BPR1_EL1
0b000	0b1100	0b1100	0b100	RW	ICC_CTLR_EL1	ICC_CTLR_EL1 ICV_CTLR_EL1
0b000	0b1100	0b1100	0b101	RW	ICC_SRE_EL1	ICC_SRE_EL1
0b000	0b1100	0b1100	0b110	RW	ICC_IGRPEN0_EL1	ICC_IGRPEN0_EL1 ICV_IGRPEN0_EL1
0b000	0b1100	0b1100	0b111	RW	ICC_IGRPEN1_EL1	ICC_IGRPEN1_EL1 ICV_IGRPEN1_EL1
0b000	0b1101	0b0000	0b001	RW	CONTEXTIDR_EL1	CONTEXTIDR_EL1 CONTEXTIDR_EL2
0b000	0b1101	0b0000	0b011	RW	RCWSMASK_EL1	RCWSMASK_EL1
0b000	0b1101	0b0000	0b011	RW	RCWSMASK_EL1	RCWSMASK_EL1
0b000	0b1101	0b0000	0b100	RW	TPIDR_EL1	TPIDR_EL1
0b000	0b1101	0b0000	0b101	RW	ACCDATA_EL1	ACCDATA_EL1
0b000	0b1101	0b0000	0b110	RW	RCWMASK_EL1	RCWMASK_EL1
0b000	0b1101	0b0000	0b110	RW	RCWMASK_EL1	RCWMASK_EL1
0b000	0b1101	0b0000	0b111	RW	SCXTNUM_EL1	SCXTNUM_EL1 SCXTNUM_EL2
0b000	0b1110	0b0001	0b000	RW	CNTKCTL_EL1	CNTHCTL_EL2 CNTKCTL_EL1
0b001	0b0000	0b0000	0b000	RO	CCSIDR_EL1	CCSIDR_EL1
0b001	0b0000	0b0000	0b001	RO	CLIDR_EL1	CLIDR_EL1
0b001	0b0000	0b0000	0b010	RO	CCSIDR2_EL1	CCSIDR2_EL1
0b001	0b0000	0b0000	0b100	RO	GMID_EL1	GMID_EL1
0b001	0b0000	0b0000	0b110	RO	SMIDR_EL1	SMIDR_EL1
0b001	0b0000	0b0000	0b111	RO	AIDR_EL1	AIDR_EL1
0b010	0b0000	0b0000	0b000	RW	CSSELR_EL1	CSSELR_EL1
0b011	0b0000	0b0000	0b001	RO	CTR_EL0	CTR_EL0
0b011	0b0000	0b0000	0b111	RO	DCZID_EL0	DCZID_EL0
0b011	0b0010	0b0100	0b000	RO	RNDR	RNDR
0b011	0b0010	0b0100	0b001	RO	RNDRRS	RNDRRS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b011	0b0010	0b0101	0b001	RW	GCSPR_EL0	GCSPR_EL0
0b011	0b0100	0b0010	0b000	RW	NZCV	NZCV
0b011	0b0100	0b0010	0b001	RW	DAIF	DAIF
0b011	0b0100	0b0010	0b010	RW	SVCR	SVCR
0b011	0b0100	0b0010	0b101	RW	DIT	DIT
0b011	0b0100	0b0010	0b110	RW	SSBS	SSBS
0b011	0b0100	0b0010	0b111	RW	TCO	TCO
0b011	0b0100	0b0100	0b000	RW	FPCR	FPCR
0b011	0b0100	0b0100	0b001	RW	FPSR	FPSR
0b011	0b0100	0b0100	0b010	RW	FPMR	FPMR
0b011	0b0100	0b0101	0b000	RW	DSPSR_EL0	DSPSR_EL0
0b011	0b0100	0b0101	0b001	RW	DLR_EL0	DLR_EL0
0b011	0b1001	0b0100	0b000	RW	PMICNTR_EL0	PMICNTR_EL0
0b011	0b1001	0b0110	0b000	RW	PMICFILTR_EL0	PMICFILTR_EL0
0b011	0b1001	0b1100	0b000	RW	PMCR_EL0	PMCR_EL0
0b011	0b1001	0b1100	0b001	RW	PMCNTENSET_EL0	PMCNTENSET_EL0
0b011	0b1001	0b1100	0b010	RW	PMCNTENCLR_EL0	PMCNTENCLR_EL0
0b011	0b1001	0b1100	0b011	RW	PMOVSCLR_EL0	PMOVSCLR_EL0
0b011	0b1001	0b1100	0b100	WO	PMSWINC_EL0	PMSWINC_EL0
0b011	0b1001	0b1100	0b101	RW	PMSELR_EL0	PMSELR_EL0
0b011	0b1001	0b1100	0b110	RO	PMCEID0_EL0	PMCEID0_EL0
0b011	0b1001	0b1100	0b111	RO	PMCEID1_EL0	PMCEID1_EL0
0b011	0b1001	0b1101	0b000	RW	PMCCNTR_EL0	PMCCNTR_EL0
0b011	0b1001	0b1101	0b001	RW	PMXEVTYPER_EL0	PMXEVTYPER_EL0
0b011	0b1001	0b1101	0b010	RW	PMXEVCNTR_EL0	PMXEVCNTR_EL0
0b011	0b1001	0b1101	0b100	WO	PMZR_EL0	PMZR_EL0
0b011	0b1001	0b1110	0b000	RW	PMUSERENR_EL0	PMUSERENR_EL0
0b011	0b1001	0b1110	0b011	RW	PMOVSSET_EL0	PMOVSSET_EL0
0b011	0b1010	0b0010	0b100	RW	POR_EL0	POR_EL0
0b011	0b1101	0b0000	0b010	RW	TPIDR_EL0	TPIDR_EL0
0b011	0b1101	0b0000	0b011	RW	TPIDRRO_EL0	TPIDRRO_EL0
0b011	0b1101	0b0000	0b101	RW	TPIDR2_EL0	TPIDR2_EL0
0b011	0b1101	0b0000	0b111	RW	SCXTNUM_EL0	SCXTNUM_EL0
0b011	0b1101	0b0010	0b000	RW	AMCR_EL0	AMCR_EL0
0b011	0b1101	0b0010	0b001	RO	AMCFGR_EL0	AMCFGR_EL0
0b011	0b1101	0b0010	0b010	RO	AMCGCR_EL0	AMCGCR_EL0
0b011	0b1101	0b0010	0b011	RW	AMUSERENR_EL0	AMUSERENR_EL0
0b011	0b1101	0b0010	0b100	RW	AMCNTENCLR0_EL0	AMCNTENCLR0_EL0
0b011	0b1101	0b0010	0b101	RW	AMCNTENSET0_EL0	AMCNTENSET0_EL0
0b011	0b1101	0b0010	0b110	RO	AMCG1IDR_EL0	AMCG1IDR_EL0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b011	0b1101	0b0011	0b000	RW	AMCNTENCLR1_EL0	AMCNTENCLR1_EL0
0b011	0b1101	0b0011	0b001	RW	AMCNTENSET1_EL0	AMCNTENSET1_EL0
0b011	0b1101	0b010:m[3]	m[2:0]	RW	AMEVCNTR0<m>_EL0	AMEVCNTR0<n>_EL0
0b011	0b1101	0b011:m[3]	m[2:0]	RO	AMEVTYPER0<m>_EL0	AMEVTYPER0<n>_EL0
0b011	0b1101	0b110:m[3]	m[2:0]	RW	AMEVCNTR1<m>_EL0	AMEVCNTR1<n>_EL0
0b011	0b1101	0b111:m[3]	m[2:0]	RW	AMEVTYPER1<m>_EL0	AMEVTYPER1<n>_EL0
0b011	0b1110	0b0000	0b000	RW	CNTFRQ_EL0	CNTFRQ_EL0
0b011	0b1110	0b0000	0b001	RO	CNTPCT_EL0	CNTPCT_EL0
0b011	0b1110	0b0000	0b010	RO	CNTVCT_EL0	CNTVCT_EL0
0b011	0b1110	0b0000	0b101	RO	CNTPCTSS_EL0	CNTPCTSS_EL0
0b011	0b1110	0b0000	0b110	RO	CNTVCTSS_EL0	CNTVCTSS_EL0
0b011	0b1110	0b0010	0b000	RW	CNTP_TVAL_EL0	CNTHPS_TVAL_EL2 CNTHP_TVAL_EL2 CNTP_TVAL_EL0
0b011	0b1110	0b0010	0b001	RW	CNTP_CTL_EL0	CNTHPS_CTL_EL2 CNTHP_CTL_EL2 CNTP_CTL_EL0
0b011	0b1110	0b0010	0b010	RW	CNTP_CVAL_EL0	CNTHPS_CVAL_EL2 CNTHP_CVAL_EL2 CNTP_CVAL_EL0
0b011	0b1110	0b0011	0b000	RW	CNTV_TVAL_EL0	CNTHVS_TVAL_EL2 CNTHV_TVAL_EL2 CNTV_TVAL_EL0
0b011	0b1110	0b0011	0b001	RW	CNTV_CTL_EL0	CNTHVS_CTL_EL2 CNTHV_CTL_EL2 CNTV_CTL_EL0
0b011	0b1110	0b0011	0b010	RW	CNTV_CVAL_EL0	CNTHVS_CVAL_EL2 CNTHV_CVAL_EL2 CNTV_CVAL_EL0
0b011	0b1110	0b10:m[4:3]	m[2:0]	RW	PMEVCNTR<m>_EL0	PMEVCNTR<n>_EL0
0b011	0b1110	0b1111	0b111	RW	PMCCFILTR_EL0	PMCCFILTR_EL0
0b011	0b1110	0b11:m[4:3]	m[2:0]	RW	PMEVTYPER<m>_EL0	PMEVTYPER<n>_EL0
0b100	0b0000	0b0000	0b000	RW	VPIDR_EL2	VPIDR_EL2
0b100	0b0000	0b0000	0b101	RW	VMPIDR_EL2	VMPIDR_EL2
0b100	0b0001	0b0000	0b000	RW	SCTLR_EL2	SCTLR_EL2
0b100	0b0001	0b0000	0b001	RW	ACTLR_EL2	ACTLR_EL2
0b100	0b0001	0b0000	0b011	RW	SCTLR2_EL2	SCTLR2_EL2
0b100	0b0001	0b0001	0b000	RW	HCR_EL2	HCR_EL2
0b100	0b0001	0b0001	0b001	RW	MDCR_EL2	MDCR_EL2
0b100	0b0001	0b0001	0b010	RW	CPTR_EL2	CPTR_EL2
0b100	0b0001	0b0001	0b011	RW	HSTR_EL2	HSTR_EL2
0b100	0b0001	0b0001	0b100	RW	HFGRTR_EL2	HFGRTR_EL2
0b100	0b0001	0b0001	0b101	RW	HFGWTR_EL2	HFGWTR_EL2
0b100	0b0001	0b0001	0b110	RW	HFGITR_EL2	HFGITR_EL2
0b100	0b0001	0b0001	0b111	RW	HACR_EL2	HACR_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b0001	0b0010	0b000	RW	ZCR_EL2	ZCR_EL2
0b100	0b0001	0b0010	0b001	RW	TRFCR_EL2	TRFCR_EL2
0b100	0b0001	0b0010	0b010	RW	HCRX_EL2	HCRX_EL2
0b100	0b0001	0b0010	0b011	RW	TRCITECR_EL2	TRCITECR_EL2
0b100	0b0001	0b0010	0b101	RW	SMPRIMAP_EL2	SMPRIMAP_EL2
0b100	0b0001	0b0010	0b110	RW	SMCR_EL2	SMCR_EL2
0b100	0b0001	0b0011	0b001	RW	SDER32_EL2	SDER32_EL2
0b100	0b0001	0b0100	0b000	RW	SCTLRMASK_EL2	SCTLRMASK_EL2
0b100	0b0001	0b0100	0b001	RW	ACTLRMASK_EL2	ACTLRMASK_EL2
0b100	0b0001	0b0100	0b010	RW	CPTRMASK_EL2	CPTRMASK_EL2
0b100	0b0001	0b0100	0b011	RW	SCTLR2MASK_EL2	SCTLR2MASK_EL2
0b100	0b0010	0b0000	0b000	RW	TTBR0_EL2	TTBR0_EL2
0b100	0b0010	0b0000	0b000	RW	TTBR0_EL2	TTBR0_EL2
0b100	0b0010	0b0000	0b001	RW	TTBR1_EL2	TTBR1_EL2
0b100	0b0010	0b0000	0b001	RW	TTBR1_EL2	TTBR1_EL2
0b100	0b0010	0b0000	0b010	RW	TCR_EL2	TCR_EL2
0b100	0b0010	0b0000	0b011	RW	TCR2_EL2	TCR2_EL2
0b100	0b0010	0b0001	0b000	RW	VTTBR_EL2	VTTBR_EL2
0b100	0b0010	0b0001	0b000	RW	VTTBR_EL2	VTTBR_EL2
0b100	0b0010	0b0001	0b010	RW	VTCR_EL2	VTCR_EL2
0b100	0b0010	0b0010	0b000	RW	VNCR_EL2	VNCR_EL2
0b100	0b0010	0b0011	0b010	RW	HDBSSBR_EL2	HDBSSBR_EL2
0b100	0b0010	0b0011	0b011	RW	HDBSSPROD_EL2	HDBSSPROD_EL2
0b100	0b0010	0b0011	0b100	RW	HACDBSBR_EL2	HACDBSBR_EL2
0b100	0b0010	0b0011	0b101	RW	HACDBSCONS_EL2	HACDBSCONS_EL2
0b100	0b0010	0b0101	0b000	RW	GCSCR_EL2	GCSCR_EL2
0b100	0b0010	0b0101	0b001	RW	GCSPR_EL2	GCSPR_EL2
0b100	0b0010	0b0110	0b000	RW	VSTTBR_EL2	VSTTBR_EL2
0b100	0b0010	0b0110	0b010	RW	VSTCR_EL2	VSTCR_EL2
0b100	0b0010	0b0111	0b010	RW	TCRMASK_EL2	TCRMASK_EL2
0b100	0b0010	0b0111	0b011	RW	TCR2MASK_EL2	TCR2MASK_EL2
0b100	0b0011	0b0000	0b000	RW	DACR32_EL2	DACR32_EL2
0b100	0b0011	0b0001	0b000	RW	HDFGRTR2_EL2	HDFGRTR2_EL2
0b100	0b0011	0b0001	0b001	RW	HDFGWTR2_EL2	HDFGWTR2_EL2
0b100	0b0011	0b0001	0b010	RW	HFGRTR2_EL2	HFGRTR2_EL2
0b100	0b0011	0b0001	0b011	RW	HFGWTR2_EL2	HFGWTR2_EL2
0b100	0b0011	0b0001	0b100	RW	HDFGRTR_EL2	HDFGRTR_EL2
0b100	0b0011	0b0001	0b101	RW	HDFGWTR_EL2	HDFGWTR_EL2
0b100	0b0011	0b0001	0b110	RW	HAFGRTR_EL2	HAFGRTR_EL2
0b100	0b0011	0b0001	0b111	RW	HFGITR2_EL2	HFGITR2_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b0100	0b0000	0b000	RW	SPSR_EL2	SPSR_EL1 SPSR_EL2
0b100	0b0100	0b0000	0b001	RW	ELR_EL2	ELR_EL1 ELR_EL2
0b100	0b0100	0b0001	0b000	RW	SP_EL1	SP_EL1
0b100	0b0100	0b0011	0b000	RW	SPSR_irq	SPSR_irq
0b100	0b0100	0b0011	0b001	RW	SPSR_abt	SPSR_abt
0b100	0b0100	0b0011	0b010	RW	SPSR_und	SPSR_und
0b100	0b0100	0b0011	0b011	RW	SPSR_fiq	SPSR_fiq
0b100	0b0101	0b0000	0b001	RW	IFSR32_EL2	IFSR32_EL2
0b100	0b0101	0b0001	0b000	RW	AFSR0_EL2	AFSR0_EL2
0b100	0b0101	0b0001	0b001	RW	AFSR1_EL2	AFSR1_EL2
0b100	0b0101	0b0010	0b000	RW	ESR_EL2	ESR_EL1 ESR_EL2
0b100	0b0101	0b0010	0b011	RW	VSESR_EL2	VSESR_EL2
0b100	0b0101	0b0011	0b000	RW	FPEXC32_EL2	FPEXC32_EL2
0b100	0b0101	0b0110	0b000	RW	TFSR_EL2	TFSR_EL1 TFSR_EL2
0b100	0b0110	0b0000	0b000	RW	FAR_EL2	FAR_EL1 FAR_EL2
0b100	0b0110	0b0000	0b100	RW	HPFAR_EL2	HPFAR_EL2
0b100	0b0110	0b0000	0b101	RW	PFAR_EL2	PFAR_EL2
0b100	0b1001	0b1001	0b000	RW	PMSCR_EL2	PMSCR_EL2
0b100	0b1001	0b1010	0b011	RW	PMBSR_EL2	PMBSR_EL2
0b100	0b1001	0b1011	0b011	RW	TRBSR_EL2	TRBSR_EL2
0b100	0b1010	0b0001	0b001	RW	MAIR2_EL2	MAIR2_EL2
0b100	0b1010	0b0010	0b000	RW	MAIR_EL2	MAIR_EL2
0b100	0b1010	0b0010	0b010	RW	PIRE0_EL2	PIRE0_EL2
0b100	0b1010	0b0010	0b011	RW	PIR_EL2	PIR_EL2
0b100	0b1010	0b0010	0b100	RW	POR_EL2	POR_EL2
0b100	0b1010	0b0010	0b101	RW	S2PIR_EL2	S2PIR_EL2
0b100	0b1010	0b0011	0b000	RW	AMAIR_EL2	AMAIR_EL2
0b100	0b1010	0b0011	0b001	RW	AMAIR2_EL2	AMAIR2_EL2
0b100	0b1010	0b0100	0b000	RW	MPAMHCR_EL2	MPAMHCR_EL2
0b100	0b1010	0b0100	0b001	RW	MPAMVPMV_EL2	MPAMVPMV_EL2
0b100	0b1010	0b0101	0b000	RW	MPAM2_EL2	MPAM2_EL2
0b100	0b1010	0b0101	0b100	RW	MPAMBW2_EL2	MPAMBW2_EL2
0b100	0b1010	0b0101	0b110	RW	MPAMBWCAP_EL2	MPAMBWCAP_EL2
0b100	0b1010	0b0110	0b000	RW	MPAMVPM0_EL2	MPAMVPM0_EL2
0b100	0b1010	0b0110	0b001	RW	MPAMVPM1_EL2	MPAMVPM1_EL2
0b100	0b1010	0b0110	0b010	RW	MPAMVPM2_EL2	MPAMVPM2_EL2
0b100	0b1010	0b0110	0b011	RW	MPAMVPM3_EL2	MPAMVPM3_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b1010	0b0110	0b100	RW	MPAMVPM4_EL2	MPAMVPM4_EL2
0b100	0b1010	0b0110	0b101	RW	MPAMVPM5_EL2	MPAMVPM5_EL2
0b100	0b1010	0b0110	0b110	RW	MPAMVPM6_EL2	MPAMVPM6_EL2
0b100	0b1010	0b0110	0b111	RW	MPAMVPM7_EL2	MPAMVPM7_EL2
0b100	0b1010	0b1000	0b000	RW	MECID_P0_EL2	MECID_P0_EL2
0b100	0b1010	0b1000	0b001	RW	MECID_A0_EL2	MECID_A0_EL2
0b100	0b1010	0b1000	0b010	RW	MECID_P1_EL2	MECID_P1_EL2
0b100	0b1010	0b1000	0b011	RW	MECID_A1_EL2	MECID_A1_EL2
0b100	0b1010	0b1000	0b111	RO	MECIDR_EL2	MECIDR_EL2
0b100	0b1010	0b1001	0b000	RW	VMECID_P_EL2	VMECID_P_EL2
0b100	0b1010	0b1001	0b001	RW	VMECID_A_EL2	VMECID_A_EL2
0b100	0b1100	0b0000	0b000	RW	VBAR_EL2	VBAR_EL2
0b100	0b1100	0b0000	0b001	RO	RVBAR_EL2	RVBAR_EL2
0b100	0b1100	0b0000	0b010	RW	RMR_EL2	RMR_EL2
0b100	0b1100	0b0001	0b001	RW	VDISR_EL2	VDISR_EL2
0b100	0b1100	0b1000	0b0:m[1:0]	RW	ICH_AP0R<m>_EL2	ICH_AP0R<n>_EL2
0b100	0b1100	0b1001	0b0:m[1:0]	RW	ICH_AP1R<m>_EL2	ICH_AP1R<n>_EL2
0b100	0b1100	0b1001	0b101	RW	ICC_SRE_EL2	ICC_SRE_EL2
0b100	0b1100	0b1011	0b000	RW	ICH_HCR_EL2	ICH_HCR_EL2
0b100	0b1100	0b1011	0b001	RO	ICH_VTR_EL2	ICH_VTR_EL2
0b100	0b1100	0b1011	0b010	RO	ICH_MISR_EL2	ICH_MISR_EL2
0b100	0b1100	0b1011	0b011	RO	ICH_EISR_EL2	ICH_EISR_EL2
0b100	0b1100	0b1011	0b101	RO	ICH_ELRSR_EL2	ICH_ELRSR_EL2
0b100	0b1100	0b1011	0b111	RW	ICH_VMCR_EL2	ICH_VMCR_EL2
0b100	0b1100	0b110:m[3]	m[2:0]	RW	ICH_LR<m>_EL2	ICH_LR<n>_EL2
0b100	0b1101	0b0000	0b001	RW	CONTEXTIDR_EL2	CONTEXTIDR_EL2
0b100	0b1101	0b0000	0b010	RW	TPIDR_EL2	TPIDR_EL2
0b100	0b1101	0b0000	0b111	RW	SCXTNUM_EL2	SCXTNUM_EL2
0b100	0b1101	0b100:m[3]	m[2:0]	RW	AMEVCNTVOFF0<m>_EL2	AMEVCNTVOFF0<n>_EL2
0b100	0b1101	0b101:m[3]	m[2:0]	RW	AMEVCNTVOFF1<m>_EL2	AMEVCNTVOFF1<n>_EL2
0b100	0b1110	0b0000	0b011	RW	CNTVOFF_EL2	CNTVOFF_EL2
0b100	0b1110	0b0000	0b110	RW	CNTPOFF_EL2	CNTPOFF_EL2
0b100	0b1110	0b0001	0b000	RW	CNTHCTL_EL2	CNTHCTL_EL2
0b100	0b1110	0b0010	0b000	RW	CNTHP_TVAL_EL2	CNTHP_TVAL_EL2
0b100	0b1110	0b0010	0b001	RW	CNTHP_CTL_EL2	CNTHP_CTL_EL2
0b100	0b1110	0b0010	0b010	RW	CNTHP_CVAL_EL2	CNTHP_CVAL_EL2
0b100	0b1110	0b0011	0b000	RW	CNTHV_TVAL_EL2	CNTHV_TVAL_EL2
0b100	0b1110	0b0011	0b001	RW	CNTHV_CTL_EL2	CNTHV_CTL_EL2
0b100	0b1110	0b0011	0b010	RW	CNTHV_CVAL_EL2	CNTHV_CVAL_EL2
0b100	0b1110	0b0100	0b000	RW	CNTHVS_TVAL_EL2	CNTHVS_TVAL_EL2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b1110	0b0100	0b001	RW	CNTHVS_CTL_EL2	CNTHVS_CTL_EL2
0b100	0b1110	0b0100	0b010	RW	CNTHVS_CVAL_EL2	CNTHVS_CVAL_EL2
0b100	0b1110	0b0101	0b000	RW	CNTHPS_TVAL_EL2	CNTHPS_TVAL_EL2
0b100	0b1110	0b0101	0b001	RW	CNTHPS_CTL_EL2	CNTHPS_CTL_EL2
0b100	0b1110	0b0101	0b010	RW	CNTHPS_CVAL_EL2	CNTHPS_CVAL_EL2
0b101	0b0001	0b0000	0b000	RW	SCTLR_EL12	SCTLR_EL1
0b101	0b0001	0b0000	0b001	RW	ACTLR_EL12	ACTLR_EL1
0b101	0b0001	0b0000	0b010	RW	CPACR_EL12	CPACR_EL1
0b101	0b0001	0b0000	0b011	RW	SCTLR2_EL12	SCTLR2_EL1
0b101	0b0001	0b0010	0b000	RW	ZCR_EL12	ZCR_EL1
0b101	0b0001	0b0010	0b001	RW	TRFCR_EL12	TRFCR_EL1
0b101	0b0001	0b0010	0b011	RW	TRCITECR_EL12	TRCITECR_EL1
0b101	0b0001	0b0010	0b110	RW	SMCR_EL12	SMCR_EL1
0b101	0b0001	0b0100	0b000	RW	SCTLRMASK_EL12	SCTLRMASK_EL1
0b101	0b0001	0b0100	0b001	RW	ACTLRMASK_EL12	ACTLRMASK_EL1
0b101	0b0001	0b0100	0b010	RW	CPACRMASK_EL12	CPACRMASK_EL1
0b101	0b0001	0b0100	0b011	RW	SCTLR2MASK_EL12	SCTLR2MASK_EL1
0b101	0b0010	0b0000	0b000	RW	TTBR0_EL12	TTBR0_EL1
0b101	0b0010	0b0000	0b000	RW	TTBR0_EL12	TTBR0_EL1
0b101	0b0010	0b0000	0b001	RW	TTBR1_EL12	TTBR1_EL1
0b101	0b0010	0b0000	0b001	RW	TTBR1_EL12	TTBR1_EL1
0b101	0b0010	0b0000	0b010	RW	TCR_EL12	TCR_EL1
0b101	0b0010	0b0000	0b011	RW	TCR2_EL12	TCR2_EL1
0b101	0b0010	0b0101	0b000	RW	GCSCR_EL12	GCSCR_EL1
0b101	0b0010	0b0101	0b001	RW	GCSPR_EL12	GCSPR_EL1
0b101	0b0010	0b0111	0b010	RW	TCRMASK_EL12	TCRMASK_EL1
0b101	0b0010	0b0111	0b011	RW	TCR2MASK_EL12	TCR2MASK_EL1
0b101	0b0100	0b0000	0b000	RW	SPSR_EL12	SPSR_EL1
0b101	0b0100	0b0000	0b001	RW	ELR_EL12	ELR_EL1
0b101	0b0101	0b0001	0b000	RW	AFSR0_EL12	AFSR0_EL1
0b101	0b0101	0b0001	0b001	RW	AFSR1_EL12	AFSR1_EL1
0b101	0b0101	0b0010	0b000	RW	ESR_EL12	ESR_EL1
0b101	0b0101	0b0110	0b000	RW	TFSR_EL12	TFSR_EL1
0b101	0b0110	0b0000	0b000	RW	FAR_EL12	FAR_EL1
0b101	0b0110	0b0000	0b101	RW	PFAR_EL12	PFAR_EL1
0b101	0b1001	0b1001	0b000	RW	PMSCR_EL12	PMSCR_EL1
0b101	0b1001	0b1010	0b011	RW	PMBSR_EL12	PMBSR_EL1
0b101	0b1001	0b1011	0b011	RW	TRBSR_EL12	TRBSR_EL1
0b101	0b1010	0b0010	0b000	RW	MAIR_EL12	MAIR_EL1
0b101	0b1010	0b0010	0b001	RW	MAIR2_EL12	MAIR2_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b101	0b1010	0b0010	0b010	RW	PIRE0_EL12	PIRE0_EL1
0b101	0b1010	0b0010	0b011	RW	PIR_EL12	PIR_EL1
0b101	0b1010	0b0010	0b100	RW	POR_EL12	POR_EL1
0b101	0b1010	0b0011	0b000	RW	AMAIR_EL12	AMAIR_EL1
0b101	0b1010	0b0011	0b001	RW	AMAIR2_EL12	AMAIR2_EL1
0b101	0b1010	0b0101	0b000	RW	MPAM1_EL12	MPAM1_EL1
0b101	0b1010	0b0101	0b100	RW	MPAMBW1_EL12	MPAMBW1_EL1
0b101	0b1100	0b0000	0b000	RW	VBAR_EL12	VBAR_EL1
0b101	0b1101	0b0000	0b001	RW	CONTEXTIDR_EL12	CONTEXTIDR_EL1
0b101	0b1101	0b0000	0b111	RW	SCXTNUM_EL12	SCXTNUM_EL1
0b101	0b1110	0b0001	0b000	RW	CNTKCTL_EL12	CNTKCTL_EL1
0b101	0b1110	0b0010	0b000	RW	CNTP_TVAL_EL02	CNTP_TVAL_EL0
0b101	0b1110	0b0010	0b001	RW	CNTP_CTL_EL02	CNTP_CTL_EL0
0b101	0b1110	0b0010	0b010	RW	CNTP_CVAL_EL02	CNTP_CVAL_EL0
0b101	0b1110	0b0011	0b000	RW	CNTV_TVAL_EL02	CNTV_TVAL_EL0
0b101	0b1110	0b0011	0b001	RW	CNTV_CTL_EL02	CNTV_CTL_EL0
0b101	0b1110	0b0011	0b010	RW	CNTV_CVAL_EL02	CNTV_CVAL_EL0
0b110	0b0001	0b0000	0b000	RW	SCTLR_EL3	SCTLR_EL3
0b110	0b0001	0b0000	0b001	RW	ACTLR_EL3	ACTLR_EL3
0b110	0b0001	0b0000	0b011	RW	SCTLR2_EL3	SCTLR2_EL3
0b110	0b0001	0b0001	0b000	RW	SCR_EL3	SCR_EL3
0b110	0b0001	0b0001	0b001	RW	SDER32_EL3	SDER32_EL3
0b110	0b0001	0b0001	0b010	RW	CPTR_EL3	CPTR_EL3
0b110	0b0001	0b0001	0b101	RW	FGWTE3_EL3	FGWTE3_EL3
0b110	0b0001	0b0010	0b000	RW	ZCR_EL3	ZCR_EL3
0b110	0b0001	0b0010	0b110	RW	SMCR_EL3	SMCR_EL3
0b110	0b0001	0b0011	0b001	RW	MDCR_EL3	MDCR_EL3
0b110	0b0010	0b0000	0b000	RW	TTBR0_EL3	TTBR0_EL3
0b110	0b0010	0b0000	0b010	RW	TCR_EL3	TCR_EL3
0b110	0b0010	0b0001	0b100	RW	GPTBR_EL3	GPTBR_EL3
0b110	0b0010	0b0001	0b101	RW	GPCBW_EL3	GPCBW_EL3
0b110	0b0010	0b0001	0b110	RW	GPCCR_EL3	GPCCR_EL3
0b110	0b0010	0b0101	0b000	RW	GCSCR_EL3	GCSCR_EL3
0b110	0b0010	0b0101	0b001	RW	GCSPR_EL3	GCSPR_EL3
0b110	0b0100	0b0000	0b000	RW	SPSR_EL3	SPSR_EL3
0b110	0b0100	0b0000	0b001	RW	ELR_EL3	ELR_EL3
0b110	0b0100	0b0001	0b000	RW	SP_EL2	SP_EL2
0b110	0b0101	0b0001	0b000	RW	AFSR0_EL3	AFSR0_EL3
0b110	0b0101	0b0001	0b001	RW	AFSR1_EL3	AFSR1_EL3
0b110	0b0101	0b0010	0b000	RW	ESR_EL3	ESR_EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b110	0b0101	0b0010	0b011	RW	VSESR_EL3	VSESR_EL3
0b110	0b0101	0b0110	0b000	RW	TFSR_EL3	TFSR_EL3
0b110	0b0110	0b0000	0b000	RW	FAR_EL3	FAR_EL3
0b110	0b0110	0b0000	0b101	RW	MFAR_EL3	MFAR_EL3
0b110	0b1001	0b1010	0b011	RW	PMBSR_EL3	PMBSR_EL3
0b110	0b1001	0b1011	0b011	RW	TRBSR_EL3	TRBSR_EL3
0b110	0b1010	0b0001	0b001	RW	MAIR2_EL3	MAIR2_EL3
0b110	0b1010	0b0010	0b000	RW	MAIR_EL3	MAIR_EL3
0b110	0b1010	0b0010	0b011	RW	PIR_EL3	PIR_EL3
0b110	0b1010	0b0010	0b100	RW	POR_EL3	POR_EL3
0b110	0b1010	0b0011	0b000	RW	AMAIR_EL3	AMAIR_EL3
0b110	0b1010	0b0011	0b001	RW	AMAIR2_EL3	AMAIR2_EL3
0b110	0b1010	0b0101	0b000	RW	MPAM3_EL3	MPAM3_EL3
0b110	0b1010	0b0101	0b100	RW	MPAMBW3_EL3	MPAMBW3_EL3
0b110	0b1010	0b1010	0b001	RW	MECID_RL_A_EL3	MECID_RL_A_EL3
0b110	0b1100	0b0000	0b000	RW	VBAR_EL3	VBAR_EL3
0b110	0b1100	0b0000	0b001	RO	RVBAR_EL3	RVBAR_EL3
0b110	0b1100	0b0000	0b010	RW	RMR_EL3	RMR_EL3
0b110	0b1100	0b0001	0b001	RW	VDISR_EL3	VDISR_EL3
0b110	0b1100	0b1100	0b100	RW	ICC_CTLR_EL3	ICC_CTLR_EL3
0b110	0b1100	0b1100	0b101	RW	ICC_SRE_EL3	ICC_SRE_EL3
0b110	0b1100	0b1100	0b111	RW	ICC_IGRPEN1_EL3	ICC_IGRPEN1_EL3
0b110	0b1101	0b0000	0b010	RW	TPIDR_EL3	TPIDR_EL3
0b110	0b1101	0b0000	0b111	RW	SCXTNUM_EL3	SCXTNUM_EL3
0b111	0b1110	0b0010	0b000	RW	CNTPS_TVAL_EL1	CNTPS_TVAL_EL1
0b111	0b1110	0b0010	0b001	RW	CNTPS_CTL_EL1	CNTPS_CTL_EL1
0b111	0b1110	0b0010	0b010	RW	CNTPS_CVAL_EL1	CNTPS_CVAL_EL1

D23.3.1.1 About the GIC System registers

From version 3 of the GIC architecture, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_ GIC physical CPU interface System registers.

ICH_ GIC virtual interface control System registers.

ICV_ GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

As Table D23-2 shows, the ICV_* registers have the same {op0, op1, CRn, CRm, op2} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

For more information, see the *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

D23.3.2 Reserved encodings for IMPLEMENTATION DEFINED registers

The System register encoding space with op0 == 0b11 reserves the following encodings for IMPLEMENTATION DEFINED registers:

31 30 29 28	27 26 25 24	23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0	L	1 1	op1	1 x 1 1	CRm	op2	Rt
				op0		CRn			

The value of L defines the access type and the use of Rt as follows:

0 Write the value in Rt to the IMPLEMENTATION DEFINED register.

1 Read the value of the IMPLEMENTATION DEFINED register to Rt.

For more information about these encodings, see S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED Registers.

As that section describes, any IMPLEMENTATION DEFINED registers are accessed in a similar way to architecturally-defined System registers, using MRS, MRRS, MSR, and MSRR instructions, see:

- MRS.
- MRRS.
- MSR (immediate).
- MSR (register).
- MSRR.

Accesses to these registers that are not trapped are permitted to be UNDEFINED. Arm recommends that IMPLEMENTATION DEFINED registers that are not UNDEFINED follow the principles in Principles of the System instruction class encoding.

The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_* and postfixed with *_ELx*, where appropriate.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D24
AArch64 System Register Descriptions

This chapter defines the AArch64 System registers. It contains the following sections:

- About the AArch64 System registers.
- General system control registers.
- Debug System registers.
- Trace registers.
- Performance Monitors registers.
- Activity Monitors registers.
- Statistical Profiling Extension registers.
- Branch Record Buffer Extension registers.
- RAS registers.
- Generic Timer registers.
- Guarded Control Stack registers.
- MPAM registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.1 About the AArch64 System registers

The following sections describe common features of the AArch64 registers:

- Reading the System register descriptions.
- General behavior of accesses to the AArch64 System registers.
- Principles of the ID scheme for fields in ID registers.

D24.1.1 Reading the System register descriptions

D24.1.1.1 Registers and register names

The title of each section in AArch64 System Register Descriptions refers to a register.

An AArch64 register is accessed using one of the functions `MRS`, `MSR (register)`, `MSR (immediate)`, `MRRS`, or `MSRR`, using an encoding value.

The encoding value is associated with a mnemonic register name, which may not be the same as the name of the register. Tables listing all encodings, mnemonics and the register they access are found in Table D23-1 and Table D23-2.

Some registers can be accessed using multiple encodings.

For example:

- When virtualization is implemented, see System and Special-purpose register redirection.
- When there is an alias mnemonic, see System and Special-purpose register aliasing.
- When a mnemonic accesses a register in an array of registers, for example PMXEVCNTR_EL0.

D24.1.1.2 Register arrays and names

Arrays of registers are indicated by an `<n>` or `<m>` in the title of the register. These are a shortcut for referencing a number of registers with identical field layout. The index letter is considered to be replaced by a number for each individual register in the array.

There are several different ways for a register in an array to be associated with an encoding:

- The same number of encodings as there are registers in the array. The mnemonic will include an index letter, which may not be the same as the register array index letter, and the encodings will be represented in terms of that index letter. For example, see PMEVCNTR<n>.
- One encoding for all registers in the array. Another register is used to select the register in the array to be accessed. For example, see PMXEVCNTR_EL0.
- A group of encodings, used to access a subset of the register array, with another register used to select which bank of registers is accessed. For example, see SPMEVCNTR<n>_EL0.

D24.1.1.3 Multiple conditions in register fields, resets or accessors

Throughout the register descriptions, conditions are used to represent changes of behavior. If multiple conditions apply to the same item, they are not necessarily mutually exclusive. The conditions are ordered so that the first true condition applies, and no further conditions are considered.

D24.1.1.4 Direct register access

In the pseudocode representing access to a particular encoding, direct access to a register is represented by a function using the same name as the register.

On a direct read, the architecturally visible value includes any field-level access controls or conditional read constraints on the value read from the register using the function.

On a direct write, any field-level access controls and constraints are applied to the value before it is written to the register using the function.

The accessibility pseudocode in the register description represents the traps and redirections for read and write accesses, and in most cases, the function to read or write the register is simple. In a few cases, there are more complex behaviors that would not be possible to represent on the register page, and these are documented in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.1.1.5 Fixed values and reserved values in the System registers

Some fields or bit ranges in registers are described as reserved, or have a fixed value.

For information on the glossary terms RAZ, RES0, RAO, and RES1, see Fixed values in AArch64 instruction and System register descriptions.

For information on the behavior of reserved values, see Reserved values in System and memory-mapped registers and translation table entries.

D24.1.2 General behavior of accesses to the AArch64 System registers

The following subsections give general information about the behavior of accesses to the System registers:

- Reset behavior of AArch64 System registers.
- Synchronization requirements for AArch64 System registers.

D24.1.2.1 Reset behavior of AArch64 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch64, see:

- Reset behavior.
- The appropriate Trace architecture specification, for the Trace System registers.

For a PE reset into an Exception level that is using AArch64, the architecture defines which AArch64 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and Reset behavior summarizes these architectural requirements. Otherwise, RW registers that have a meaningful reset value reset to an architecturally UNKNOWN value.

Note

When the PE resets into an Exception level that is using AArch32, no PE state that relates to execution in AArch64 state is accessible until another reset causes the Execution state to change to AArch64. Therefore, on a reset into AArch32 state, PE state that relates only to execution in AArch64 state cannot have a meaningful reset value.

D24.1.2.1.1 Pseudocode description of resetting System registers

The `AArch64_ResetSystemRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and Reset behavior.

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.1.2.2 Synchronization requirements for AArch64 System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent Context Synchronization event to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous Context Synchronization event. For example, where a memory access is used to communicate a read of such a counter, an `ISB` must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

Unless otherwise stated in its System register definition, a direct write to a System register requires synchronization before software can rely on the effects of that write to affect instructions appearing in program order after the write. This does not apply to Special-purpose registers. For more information, see Instructions for accessing Special-purpose registers. Direct writes to System registers are not allowed to affect any instructions appearing in program order before the direct write. The only exceptions are:

- All direct writes to the same register, that use the same encoding for that register, are guaranteed to occur in program order relative to each other.
- All direct writes to a register occur in program order with respect to all direct reads to the same register using the same encoding.
- Any System register access that an Arm *Architecture Specification* or equivalent specification defines as not requiring synchronization.
- When FEAT_BRBE is implemented, execution of `BRB INJ` does not require explicit synchronization to use the result of direct writes to the Branch record injection data registers in program order before `BRB INJ`.

Explicit synchronization occurs as a result of a Context Synchronization event, which is one of the following events:

- Execution of an `ISB` instruction.
- Exception entry, if FEAT_ExS is not implemented, or if FEAT_ExS is implemented and defines that exception entries to this Exception level are Context Synchronization events.
- Exception return, if FEAT_ExS is not implemented, or if FEAT_ExS is implemented and defines that exception returns from this Exception level are Context Synchronization events.
- Execution of a `DCPS` instruction in Debug state.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

Note

The `ISB` and exception entry events are applicable both in Debug state and in Non-debug state.

The pseudocode for each of these events defines the point at which the explicit synchronization takes effect, as a result of a call to the `SynchronizeContext()` function or `InstructionSynchronizationBarrier()` function. The principle behind the position of the call to `SynchronizeContext()` or `InstructionSynchronizationBarrier()` is that it occurs before indirect reads of System registers that are used in the construction of the target context, but is permitted to occur after indirect reads of System registers that apply to the context in which the instruction is executed or from which the event is taken.

Note

For some instructions, this means that some System registers are indirectly read before the explicit synchronization, and therefore changes to those System registers might need an explicit Context synchronization event before the instruction. For example, changes to some fields of HCR_EL2 at EL2 need an explicit ISB in program order before an ERET instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In addition, any system instructions that cause a write to a System register must be synchronized before the result is guaranteed to be visible to subsequent direct reads of that System register.

Direct and indirect reads and writes to RGSR_EL1, Random Allocation Tag Seed Register, appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Direct reads to any one of the following registers, using the same encoding, occur in program order relative to each other:

- ISR_EL1.
- The Generic Timer registers, that is, CNTPCT_EL0 and CNTVCT_EL0, and the Counter registers CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, and CNTPS_TVAL_EL1.
- DBGCLAIMCLR_EL1.
- The PMU Counters, that is, PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, and PMOVSSET_EL0.
- The System PMU Counters, that is, SPMEVCNTR<n>_EL0, SPMOVSCLR_EL0, and SPMOVSSET_EL0.
- The Debug Communications Channel System registers, that is, DBGDTRRX_EL0, DBGDTR_EL0, and MDCCSR_EL0.

All other direct reads of System registers can occur in any order if synchronization has not been performed.

Table D24-1 describes the synchronization requirements between two successive read or write accesses to the same register, where the ordering of the read or write accesses is:

1. Program order, in the event that both the reads or writes are caused by an instruction executed on this PE, other than one caused by a memory access by this PE.
2. The order of arrival of asynchronous reads and writes at the PE relative to the execution of instructions that cause reads or writes.
3. The order of arrival of asynchronous reads and writes at the PE relative to each other.

Table D24-1 Synchronization requirements

First read/write	Second read/write	Synchronization requirement
Direct read	Direct read	None
	Direct write	None
	Indirect read	None
	Indirect write	None, see Notes
Direct write	Direct read	None
	Direct write	None
	Indirect read	Required
	Indirect write	None, see Notes
Indirect read	Direct read	None
	Direct write	None
	Indirect read	None
	Indirect write	None
Indirect write	Direct read	Required, see Notes
	Direct write	None, see Notes
	Indirect read	Required, see Notes
	Indirect write	None, see Notes

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.1.2.2.1 Notes

The terms Direct read, Direct write, Indirect read, and Indirect write, as used in Table D24-1, are defined as follows:

Direct read Where software uses an MRS or MRRS system register access instruction to read that register into a general purpose register.

Where a direct read of a register has a side-effect that changes the contents of a register, the effect of a direct read on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to have occurred, and be visible to subsequent direct or indirect reads or writes, if synchronization is performed after the direct read.

Direct write Where software uses an MSR (register) or MSRR access instruction to write to that register from a general purpose register.

Where a direct write to a register has an effect on the register that means that the value in the register is not always the last value that is written (as is the case with set and clear registers), the effect of a direct write on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Indirect read Where an instruction uses a System register to establish operating conditions for the instruction, for example, the TTBR_ELx address or whether memory accesses are forced to be Non-cacheable. This includes situations where the contents of one System register selects what value is read or written using a different register. Indirect reads also include reads of the System register by external agents such as debuggers. Instruction execution from MDSTEPOP_EL1 is an indirect read of MDSTEPOP_EL1. Where an indirect read of a register has a side-effect that changes the contents of that register, that is defined to be an indirect write.

Indirect write Where a System register is written as the consequence of some other instruction, exception, operation, or by the asynchronous operation of an external agent, including the passage of time as seen in counters, timers, or performance counters, the assertion of interrupts, or writes from an external debugger.

Note

Registers such as the Exception Syndrome registers that are indirectly written as part of exception entry do not require additional synchronization. This behavior is independent of the value of SCTLR_ELx.EIS, and the list of affected registers is described in the definition of that field.

The value of a system register does not change unless one of the following occurs:

- As the result of a direct write to the register, that is not specified as being trapped, UNDEFINED or IGNORED.
- As the result of one of the specified indirect writes to the register.
- A reset, if the register has a specified reset behavior for that reset.
- The register holds an UNKNOWN value as the result of one of the above.

Where a direct read or write to a register is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two accesses.

Where an indirect write caused by a direct write is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two indirect accesses.

Where a direct read to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated, the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct read and before the subsequent direct or indirect reads or writes.

Where a direct write to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated as a side-effect of that direct write (as opposed to simply being a direct write to the different encoding), the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where indirect writes are caused by the actions of external agents such as debuggers, or by memory-mapped reads or writes by the PE, then an indirect write by that agent and mechanism to a register, followed by an indirect read by that agent and mechanism to the same register using the same address, does not require synchronization.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Note

Even though the 64-bit and 128-bit form of a system register share the same values of `{op0, op1, CRn, CRm, op2}`, they are considered to be distinct encodings. Therefore, explicit synchronization is required to guarantee ordering in all of the following cases:

- MSRR followed by an MRS from the same register.
- MSR (register) followed by an MRRS from the same register.

Where indirect writes to GCSPR_ELx are made by the following instructions, explicit synchronization is not required to be visible to direct reads, indirect reads, and indirect writes in program order after the initial indirect write:

- All Branch with Link instructions.
- All procedure return instructions.
- `GCSPOPM` and `GCSPUSHM`.
- `GCSPOPX`, `GCSPOPCX`, and `GCSPUSHX`.
- `GCSSS1` and `GCSSS2`.

Indirect writes caused by external agents, autonomous asynchronous events, or as a result of memory-mapped writes, to the registers shown in Table D24-2, are required to be observable to:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

Table D24-2 Registers with a guarantee of observability, VMSAv8-64

Registers	Notes
ISR_EL1	Interrupt Status Register
DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1	Debug CLAIM registers
CNTPCT_EL0, CNTVCT_EL0, CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, CNTPS_TVAL_EL1	Generic Timer registers
PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, PMOVSSET_EL0	PMU Counters
SPMEVCNTR<n>_EL0, SPMOVSCLR_EL0, SPMOVSSET_EL0	System PMU Counters
DBGDTRTX_EL0, DBGDTRRX_EL0, DBGDTR_EL0, and the DCC flags in MDCCSR_EL0 and EDSCR	Debug Communication Channel registers
EDSCR.PipeAdv	External Debug Status and Control Register

In addition to the requirements shown in Table D24-2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  - OSLAR_EL1. OSLAR_EL1 is indirectly read to determine whether the subsequent access is permitted.
  - EDLAR, if implemented. EDLAR is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

Note

This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table D24-2 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.

  For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a Context Synchronization event. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.

  Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.
- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See DCC and ITR access in Debug state.

Note

- The provision of explicit synchronization requirements to System registers is provided to allow the direct access to these registers to be implemented in a small number of cycles, and that updates to multiple registers can be performed quickly with the synchronization penalty being paid only when the updates have occurred.
- Since toolkits might use registers such as the thread-local storage registers within compiled code, it is recommended that access to these registers is implemented to take a small number of cycles.
- While no synchronization is required between a direct write and a direct read, or between a direct read and an indirect write, this does not imply that a direct read causes synchronization of a previous direct write. That is, the sequence direct write → direct read → indirect read, with no intervening context synchronization, does not guarantee that the indirect read observes the result of the direct write.
- If FEAT_MTE_ASYNC is implemented, one of the following is required between an indirect write to TFSRE0_EL1, or any TFSR_ELx accessible at ELy, and a direct read or direct write of that register:
  - A `DSB` instruction with the LD qualifier, or neither the LD nor ST qualifier.
  - An exception entry to ELy with SCTLR_ELy.ITFSB set to 1.

D24.1.3 Principles of the ID scheme for fields in ID registers

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields,* Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. ID registers to which this scheme applies identifies the set of ID registers.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

Note

- The ID fields described here are unlike the ones that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors.
- ID fields that do not follow this scheme include the ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, see Alternative ID scheme used for the Performance Monitors Extension version.
- The presence of an ID field for a feature does not imply that the feature is optional.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To provide forward compatibility, software can rely on the features of these fields that are described in this section.

The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, for an unsigned ID field, if the value `0x1` indicates the presence of some instructions, then the value `0x2` will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```
if (value >= number) {
    // do something that relies on the value of the feature
}
```

For ID fields where the value `0x0` defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value `0x0` indicates that the feature is implemented.
- The field value `0xF` indicates that the feature is not implemented.
- Increasingly negative values indicate additional aspects of the architecture that are not implemented as a result of the feature not being implemented.
- Software that depends on the feature can use the test:

```
if value >= 0 {
    // Software features that depend on the presence of the hardware feature
}
```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a *fractional* field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example D24-1 shows the use of such a field.

Example D24-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value `0x1` was defined as indicating that item A is present.
- The value `0x2` was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value `0x1` when A and B only are present. This fractional field is valid only when the original ID field has the value `0x1`.

This approach means that:

- Software that depends on the test if `(value >= 0x2)` can rely on features A, B, and C being present,
- Software that depends on the test `if (value >= 0x1)` can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:

```
if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {
    // Software features that depend on A and B only
}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

D24.1.3.1 ID registers to which this scheme applies

This scheme applies to the following registers:

AArch64 System registers

- The AArch64 views of the AArch32 feature ID registers given by:
  - The AArch32 Auxiliary Feature register ID_AFR0_EL1.
  - The AArch32 Processor Feature registers ID_PFR0_EL1, ID_PFR1_EL1, and ID_PFR2_EL1.
  - The AArch32 Debug Feature register ID_DFR0_EL1 and ID_DFR1_EL1.
  - The AArch32 Memory Model Feature registers ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_MMFR4_EL1, and ID_MMFR5_EL1.
  - The AArch32 Instruction Set Attribute registers ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, and ID_ISAR6_EL1.
  - The AArch32 Media and VFP Feature registers MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
- The AArch64 Auxiliary Feature registers ID_AA64AFR0_EL1 and ID_AA64AFR1_EL1.
- The AArch64 Processor Feature registers ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, and ID_AA64PFR2_EL1.
- The AArch64 Debug Feature registers ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, and ID_AA64DFR2_EL1.
- The AArch64 Memory Model Feature registers ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64MMFR2_EL1, ID_AA64MMFR3_EL1, and ID_AA64MMFR4_EL1.
- The AArch64 Instruction Set Attribute registers ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64ISAR2_EL1, and ID_AA64ISAR3_EL1.
- The AArch64 SVE Feature ID register ID_AA64ZFR0_EL1.

AArch32 System registers

- The AArch32 Auxiliary Feature register ID_AFR0.
- The AArch32 Processor Feature registers ID_PFR0, ID_PFR1, and ID_PFR2.
- The AArch32 Debug Feature register ID_DFR0 and ID_DFR1.
- The AArch32 Memory Model Feature registers ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_MMFR4, and ID_MMFR5.
- The AArch32 Instruction Set Attribute registers ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, and ID_ISAR6.
- The AArch32 Media and FP Feature registers MVFR0, MVFR1, and MVFR2.

Memory-mapped registers

- The External Debug Processor Feature register EDPFR.
- The External Debug Feature register EDDFR, EDDFR1, and EDDFR2.
- The External Debug Auxiliary Processor Feature register EDAA32PFR.

D24.1.3.2 Alternative ID scheme used for the Performance Monitors Extension version

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, that identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0xF` indicates that the Arm-architected Performance Monitors Extension is not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the field value is not `0xF`, the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```
if (value != 0xF and value >= number) {
    // do something that relies on version 'number' of the feature
}
```

For these fields, Arm deprecates use of the value `0xF` in new implementations.

D24.1.3.3 Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes

The ID_AA64MMFR0_EL1.{TGran4_2, TGran64_2, TGran16_2} fields that identify the memory translation stage 2 granule size, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0x0` indicates that support is identified by another field.
- If the field value is not `0x0`, the value indicates the level of support provided.

This means that software should use a test of the form:

```
if (field !=0 and field > value) {
    // do something that relies on the value of the feature
}
```

D24.1.3.4 Alternative ID scheme used for ID_AA64SMFR0_EL1 and ID_AA64FPFR0_EL1

Apart from the ID_AA64SMFR0_EL1.SMEver field, which is a 4-bit unsigned integer conforming to the standard scheme, software must treat the other fields in these registers as follows:

- A 1-bit field value where the bit is `0b0` indicates that the specified feature or instructions described by this field are not implemented.
- A 1-bit field value where the bit is `0b1` indicates that the specified feature or instructions described by this field are implemented.
- In ID_AA64SMFR0_EL1, a 4-bit field indicates whether a group of related SME instructions is implemented, with permitted values defined in the field description. Bits within such a field which only permit the value `0b0000` might be used to identify new instructions in a future version of SME, without changing the meaning of those bits that permit the value `0b0001`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2 General system control registers

This section lists the System registers in AArch64 that are not part of one of the other listed groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.1 ACCDATA_EL1, Accelerator Data

The ACCDATA_EL1 characteristics are:

Purpose

Holds the lower 32 bits of the data that is stored by an ST64BV0, Single-copy atomic 64-byte EL0 store instruction.

Configuration

This register is present only when FEAT_LS64_ACCDATA is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ACCDATA_EL1 are UNDEFINED.

Attributes

ACCDATA_EL1 is a 64-bit register.

Field descriptions

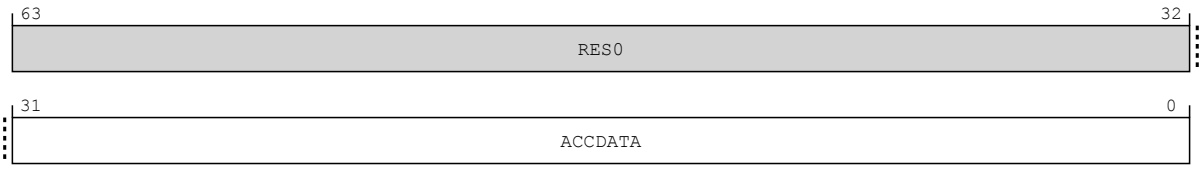

Bits [63:32]

Reserved, RES0.

ACCDATA, bits [31:0]

Accelerator Data field. Holds bits[31:0] of the data that is stored by an ST64BV0 instruction.

Accessing ACCDATA_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ACCDATA_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_LS64_ACCDATA) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ADEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nACCDATA_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().ADEn == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ACCDATA_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ADEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().ADEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ACCDATA_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACCDATA_EL1();
end;
```

```
MSR ACCDATA_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_LS64_ACCDATA) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ADEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nACCDATA_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().ADEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ACCDATA_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ADEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().ADEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ACCDATA_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    ACCDATA_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.2 ACTLR_EL1, Auxiliary Control Register (EL1)

The ACTLR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

Note

Arm recommends the contents of this register have no effect on the PE when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and instead the configuration and control fields are provided by the ACTLR_EL2 register. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

Configuration

AArch64 System register ACTLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ACTLR[31:0].

AArch64 System register ACTLR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ACTLR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ACTLR_EL1 are UNDEFINED.

Attributes

ACTLR_EL1 is a 64-bit register.

Field descriptions

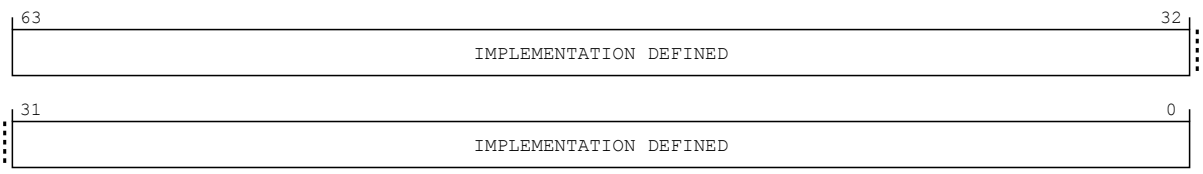

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR_EL1

If the IMPLEMENTATION DEFINED ACTLR_EL12 accessor is implemented, the following behaviors are also implemented:

- MRS/MSR to ACTLR_EL1 from EL2 when the Effective value of HCR_EL2.E2H is 1 accesses ACTLR_EL2.
- MRS/MSR to ACTLR_EL1 from EL1 when the Effective value of HCR_EL2.{NV2, NV1, NV} is {1,0,1} accesses ACTLR_EL1 directly and is not transformed to a memory access.
- MRS/MSR to ACTLRALIAS_EL1 behaves in the same way as ACTLR_EL1.

If FEAT_SRMASK is implemented, MSR to ACTLR_EL1 at EL1 are masked by ACTLRMASK_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ACTLR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TACR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && (!HaveEL(EL3) || SCR_EL3().FGTEn2 == '1')
         ↪&& HFGRTR2_EL2().ACTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior")
         ↪|| EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x118);
    else
        X{64}(t) = ACTLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        X{64}(t) = ACTLR_EL2();
    else
        X{64}(t) = ACTLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLR_EL1();
end;
```

```
MSR ACTLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TACR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && (!HaveEL(EL3) || SCR_EL3().FGTEn2 == '1')
         ↪&& HFGWTR2_EL2().ACTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior")
         ↪|| EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x118) = X{64}(t);
    else
        ACTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        ACTLR_EL2() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        ACTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ACTLR_EL1() = X{64}(t);
end;
```

When an implementation implements ACTLR_ELx accessor behavior and FEAT_VHE is implemented MRS <Xt>, ACTLR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x118);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ACTLR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = ACTLR_EL1();
    else
        Undefined();
    end;
end;
```

When an implementation implements ACTLR_ELx accessor behavior and FEAT_VHE is implemented MSR ACTLR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x118) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            ACTLR_EL1() = X{64}(t);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            ACTLR_EL1() = X{64}(t);
        else
            Undefined();
        end;
    end;
```

When FEAT_SRMASK is implemented MRS <Xt>, ACTLRALIAS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TACR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
        ↪'0') || HFGRTR2_EL2().nACTLRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor
        ↪behavior") || EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x118);
    else
        X{64}(t) = ACTLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        X{64}(t) = ACTLR_EL2();
    else
        X{64}(t) = ACTLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLR_EL1();
end;
```

When FEAT_SRMASK is implemented MSR ACTLRALIAS_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TACR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
        ↪'0') || HFGWTR2_EL2().nACTLRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor
        ↪behavior") || EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x118) = X{64}(t);
    else
        ACTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        ACTLR_EL2() = X{64}(t);
    else
        ACTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ACTLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.3 ACTLR_EL2, Auxiliary Control Register (EL2)

The ACTLR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for EL2.

Note

Arm recommends the contents of this register are updated to apply to EL0 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, gaining configuration and control fields from the ACTLR_EL1. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register ACTLR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HACTLR[31:0].

AArch64 System register ACTLR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HACTLR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ACTLR_EL2 are UNDEFINED.

Attributes

ACTLR_EL2 is a 64-bit register.

Field descriptions

63 … 32	31 … 0
IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR_EL2

If FEAT_SRMASK is implemented, MSR to ACTLR_EL2 at EL2 are masked by ACTLRMASK_EL2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ACTLR_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = ACTLR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLR_EL2();
end;
```

MSR ACTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    ACTLR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    ACTLR_EL2() = X{64}(t);
end;
```

When an implementation implements ACTLR_ELx accessor behavior MRS <Xt>, ACTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TACR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && (!HaveEL(EL3) || SCR_EL3().FGTEn2 ==
         ↪'1') && HFGRTR2_EL2().ACTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor
         ↪behavior") || EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x118);
    else
        X{64}(t) = ACTLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        X{64}(t) = ACTLR_EL2();
    else
        X{64}(t) = ACTLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLR_EL1();
end;
```

When an implementation implements ACTLR_ELx accessor behavior MSR ACTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TACR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && (!HaveEL(EL3) || SCR_EL3().FGTEn2 ==
         ↪'1') && HFGWTR2_EL2().ACTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor
         ↪behavior") || EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x118) = X{64}(t);
    else
        ACTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        ACTLR_EL2() = X{64}(t);
    else
        ACTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ACTLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.4 ACTLR_EL3, Auxiliary Control Register (EL3)

The ACTLR_EL3 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ACTLR_EL3 are UNDEFINED.

Attributes

ACTLR_EL3 is a 64-bit register.

Field descriptions

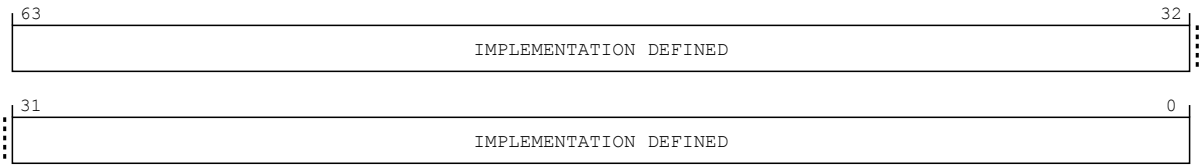

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ACTLR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLR_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ACTLR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().ACTLR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        ACTLR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.5 ACTLRMASK_EL1, Auxiliary Control Masking Register (EL1)

The ACTLRMASK_EL1 characteristics are:

Purpose

Mask register to prevent updates of fields in ACTLR_EL1 on writes to ACTLR_EL1 or ACTLRALIAS_EL1.

Configuration

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ACTLRMASK_EL1 are UNDEFINED.

Attributes

ACTLRMASK_EL1 is a 64-bit register.

Field descriptions

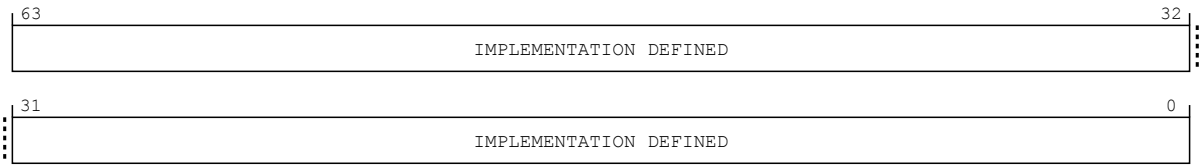

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing ACTLRMASK_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `ACTLRMASK_EL1` or `ACTLRMASK_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ACTLRMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nACTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior")
         ↪|| EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x340);
    else
        X{64}(t) = ACTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        X{64}(t) = ACTLRMASK_EL2();
    else
        X{64}(t) = ACTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLRMASK_EL1();
end;
```

MSR ACTLRMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nACTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior")
         ↪|| EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x340) = X{64}(t);
    elsif !IsZero(ACTLRMASK_EL1()) then
        Undefined();
    else
        ACTLRMASK_EL1() = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        if !IsZero(ACTLRMASK_EL2()) then
            Undefined();
        else
            ACTLRMASK_EL2() = X{64}(t);
        end;
    else
        ACTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ACTLRMASK_EL1() = X{64}(t);
end;
```

When an implementation implements ACTLR_ELx accessor behavior and FEAT_VHE is implemented MRS <Xt>, ACTLRMASK_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x340);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = ACTLRMASK_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = ACTLRMASK_EL1();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
end;
```

When an implementation implements ACTLR_ELx accessor behavior and FEAT_VHE is implemented MSR ACTLRMASK_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x340) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            ACTLRMASK_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        ACTLRMASK_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.6 ACTLRMASK_EL2, Auxiliary Control Masking Register (EL2)

The ACTLRMASK_EL2 characteristics are:

Purpose

Mask register to prevent updates of fields in ACTLR_EL2 on writes.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ACTLRMASK_EL2 are UNDEFINED.

Attributes

ACTLRMASK_EL2 is a 64-bit register.

Field descriptions

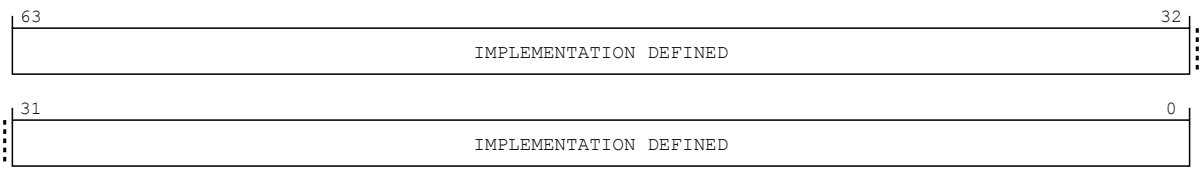

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing ACTLRMASK_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `ACTLRMASK_EL2` or `ACTLRMASK_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ACTLRMASK_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ACTLRMASK_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLRMASK_EL2();
end;
```

MSR ACTLRMASK_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsZero(ACTLRMASK_EL2()) then
        Undefined();
    else
        ACTLRMASK_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ACTLRMASK_EL2() = X{64}(t);
end;
```

MRS <Xt>, ACTLRMASK_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nACTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior")
         ↪|| EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x340);
    else
        X{64}(t) = ACTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        X{64}(t) = ACTLRMASK_EL2();
    else
        X{64}(t) = ACTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ACTLRMASK_EL1();
end;
```

```
MSR ACTLRMASK_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nACTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (!ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior")
     ↪|| EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x340) = X{64}(t);
    elsif !IsZero(ACTLRMASK_EL1()) then
        Undefined();
    else
        ACTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ImpDefBool("IMPLEMENTED_ACTLR_ELx accessor behavior") && ELIsInHost(EL2) then
        if !IsZero(ACTLRMASK_EL2()) then
            Undefined();
        else
            ACTLRMASK_EL2() = X{64}(t);
        end;
    else
        ACTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ACTLRMASK_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.7 AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)

The AFSR0_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

Configuration

AArch64 System register AFSR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ADFSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AFSR0_EL1 are UNDEFINED.

Attributes

AFSR0_EL1 is a 64-bit register.

Field descriptions

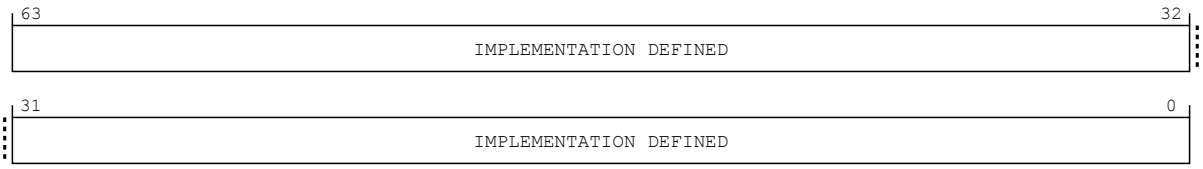

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR0_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `AFSR0_EL1` or `AFSR0_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AFSR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().AFSR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x128);
    else
        X{64}(t) = AFSR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR0_EL2();
    else
        X{64}(t) = AFSR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR0_EL1();
end;
```

MSR AFSR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().AFSR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x128) = X{64}(t);
    else
        AFSR0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AFSR0_EL2() = X{64}(t);
    else
        AFSR0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AFSR0_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AFSR0_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x128);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR0_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR0_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR AFSR0_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x128) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AFSR0_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        AFSR0_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.8 AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)

The AFSR0_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register AFSR0_EL2 bits [31:0] are architecturally mapped to AArch32 System register HADFSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AFSR0_EL2 are UNDEFINED.

Attributes

AFSR0_EL2 is a 64-bit register.

Field descriptions

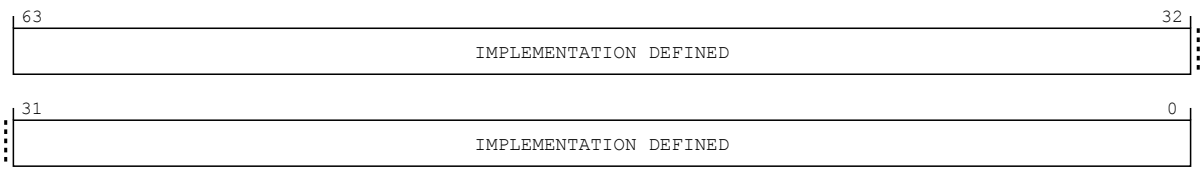

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR0_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `AFSR0_EL2` or `AFSR0_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AFSR0_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = AFSR0_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR0_EL2();
end;
```

MSR AFSR0_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AFSR0_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    AFSR0_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AFSR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().AFSR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x128);
    else
        X{64}(t) = AFSR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR0_EL2();
    else
        X{64}(t) = AFSR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR0_EL1();
end;
```

When FEAT_VHE is implemented MSR AFSR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().AFSR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x128) = X{64}(t);
    else
        AFSR0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AFSR0_EL2() = X{64}(t);
    else
        AFSR0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AFSR0_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.9 AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)

The AFSR0_EL3 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AFSR0_EL3 are UNDEFINED.

Attributes

AFSR0_EL3 is a 64-bit register.

Field descriptions

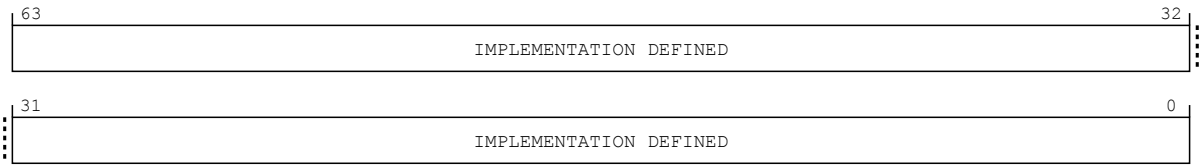

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR0_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AFSR0_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR0_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR AFSR0_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().AFSR0_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        AFSR0_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.10 AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)

The AFSR1_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

Configuration

AArch64 System register AFSR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register AIFSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AFSR1_EL1 are UNDEFINED.

Attributes

AFSR1_EL1 is a 64-bit register.

Field descriptions

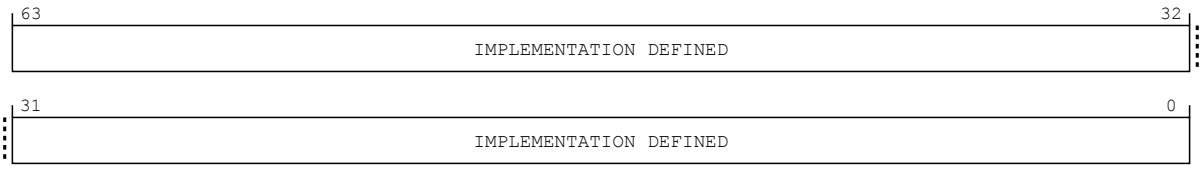

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR1_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `AFSR1_EL1` or `AFSR1_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AFSR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().AFSR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x130);
    else
        X{64}(t) = AFSR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR1_EL2();
    else
        X{64}(t) = AFSR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR1_EL1();
end;
```

MSR AFSR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().AFSR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x130) = X{64}(t);
    else
        AFSR1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AFSR1_EL2() = X{64}(t);
    else
        AFSR1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AFSR1_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AFSR1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x130);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR1_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR1_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR AFSR1_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x130) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AFSR1_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        AFSR1_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.11 AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)

The AFSR1_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register AFSR1_EL2 bits [31:0] are architecturally mapped to AArch32 System register HAIFSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AFSR1_EL2 are UNDEFINED.

Attributes

AFSR1_EL2 is a 64-bit register.

Field descriptions

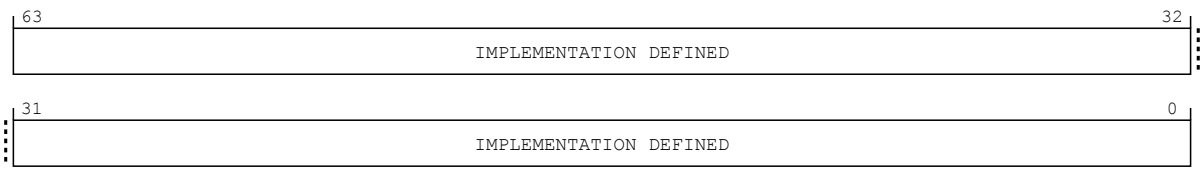

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR1_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `AFSR1_EL2` or `AFSR1_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AFSR1_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = AFSR1_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR1_EL2();
end;
```

MSR AFSR1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AFSR1_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    AFSR1_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AFSR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().AFSR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x130);
    else
        X{64}(t) = AFSR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AFSR1_EL2();
    else
        X{64}(t) = AFSR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR1_EL1();
end;
```

When FEAT_VHE is implemented MSR AFSR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().AFSR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x130) = X{64}(t);
    else
        AFSR1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AFSR1_EL2() = X{64}(t);
    else
        AFSR1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AFSR1_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.12 AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)

The AFSR1_EL3 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AFSR1_EL3 are UNDEFINED.

Attributes

AFSR1_EL3 is a 64-bit register.

Field descriptions

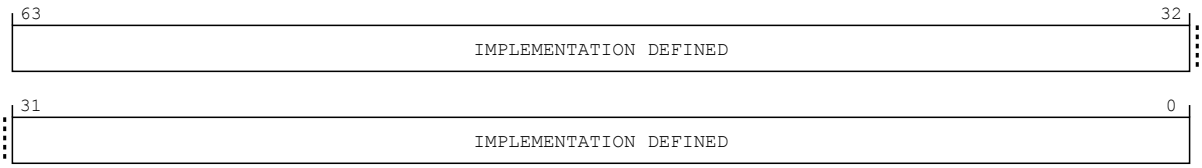

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR1_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AFSR1_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AFSR1_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR AFSR1_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0001	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().AFSR1_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        AFSR1_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.13 AIDR_EL1, Auxiliary ID Register

The AIDR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED identification information.

The value of this register must be interpreted in conjunction with the value of MIDR_EL1.

Configuration

AArch64 System register AIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register AIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AIDR_EL1 are UNDEFINED.

Attributes

AIDR_EL1 is a 64-bit register.

Field descriptions

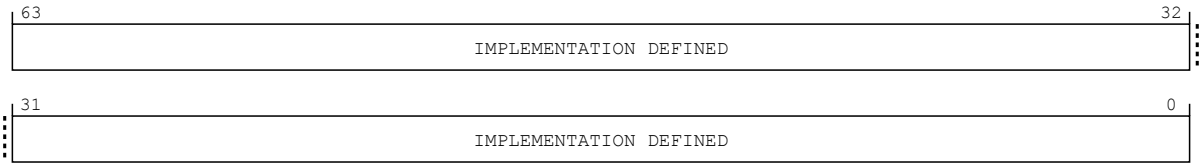

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing AIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && HCR_EL2().TID1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().AIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = AIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = AIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.14 AMAIR2_EL1, Extended Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR2_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR2_EL1.

Configuration

This register is present only when FEAT_AIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AMAIR2_EL1 are UNDEFINED.

Attributes

AMAIR2_EL1 is a 64-bit register.

Field descriptions

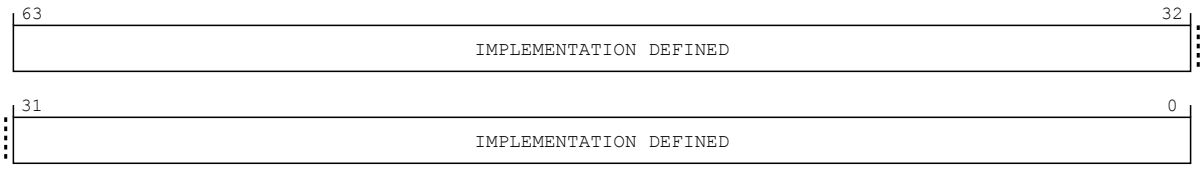

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR2_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `AMAIR2_EL1` or `AMAIR2_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMAIR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nAMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x288);
    else
        X{64}(t) = AMAIR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = AMAIR2_EL2();
    else
        X{64}(t) = AMAIR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR2_EL1();
end;
```

MSR AMAIR2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nAMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x288) = X{64}(t);
    else
        AMAIR2_EL1() = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        AMAIR2_EL2() = X{64}(t);
    else
        AMAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMAIR2_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AMAIR2_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x288);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = AMAIR2_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = AMAIR2_EL1();
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_VHE is implemented MSR AMAIR2_EL12, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x288) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AMAIR2_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        AMAIR2_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.15 AMAIR2_EL2, Extended Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR2_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR2_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_AIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AMAIR2_EL2 are UNDEFINED.

Attributes

AMAIR2_EL2 is a 64-bit register.

Field descriptions

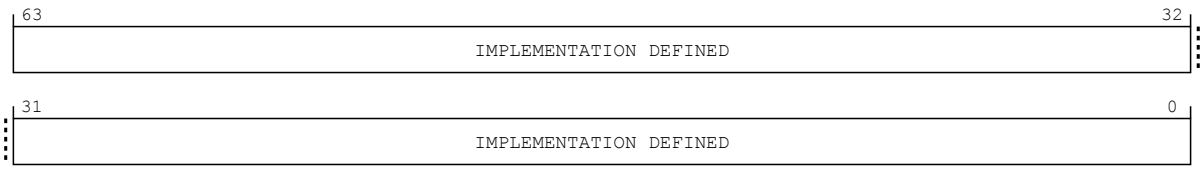

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing AMAIR2_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `AMAIR2_EL2` or `AMAIR2_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMAIR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMAIR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR2_EL2();
end;
```

MSR AMAIR2_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        AMAIR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMAIR2_EL2() = X{64}(t);
end;
```

MRS <Xt>, AMAIR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nAMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x288);
    else
        X{64}(t) = AMAIR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = AMAIR2_EL2();
    else
        X{64}(t) = AMAIR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR2_EL1();
end;
```

MSR AMAIR2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nAMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x288) = X{64}(t);
    else
        AMAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        AMAIR2_EL2() = X{64}(t);
    else
        AMAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMAIR2_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.16 AMAIR2_EL3, Extended Auxiliary Memory Attribute Indirection Register (EL3)

The AMAIR2_EL3 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR2_EL3.

Configuration

This register is present only when FEAT_AIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AMAIR2_EL3 are UNDEFINED.

Attributes

AMAIR2_EL3 is a 64-bit register.

Field descriptions

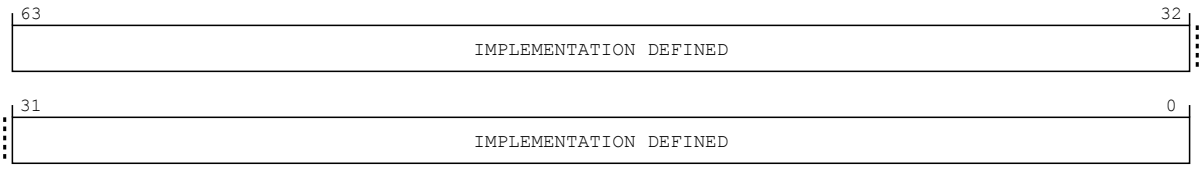

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing AMAIR2_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMAIR2_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR2_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR AMAIR2_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().AMAIR2_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        AMAIR2_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.17 AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL1.

Configuration

AArch64 System register AMAIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register AMAIR0[31:0].

AArch64 System register AMAIR_EL1 bits [63:32] are architecturally mapped to AArch32 System register AMAIR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AMAIR_EL1 are UNDEFINED.

Attributes

AMAIR_EL1 is a 64-bit register.

Field descriptions

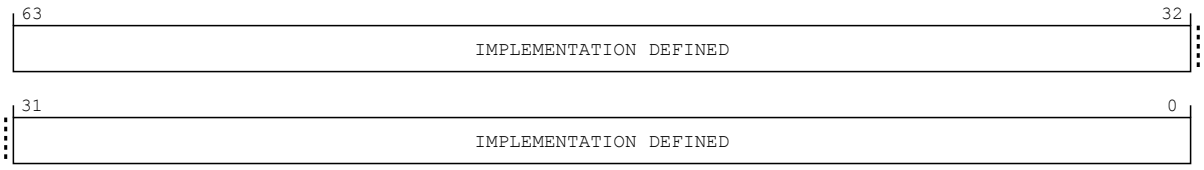

AMAIR_EL1 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `AMAIR_EL1` or `AMAIR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMAIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().AMAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x148);
    else
        X{64}(t) = AMAIR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AMAIR_EL2();
    else
        X{64}(t) = AMAIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR_EL1();
end;
```

MSR AMAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().AMAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x148) = X{64}(t);
    else
        AMAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AMAIR_EL2() = X{64}(t);
    else
        AMAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMAIR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AMAIR_EL12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x148);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AMAIR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = AMAIR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR AMAIR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x148) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AMAIR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        AMAIR_EL1() = X{64}(t);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.18 AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register AMAIR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HAMAIR0[31:0].

AArch64 System register AMAIR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HAMAIR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to AMAIR_EL2 are UNDEFINED.

Attributes

AMAIR_EL2 is a 64-bit register.

Field descriptions

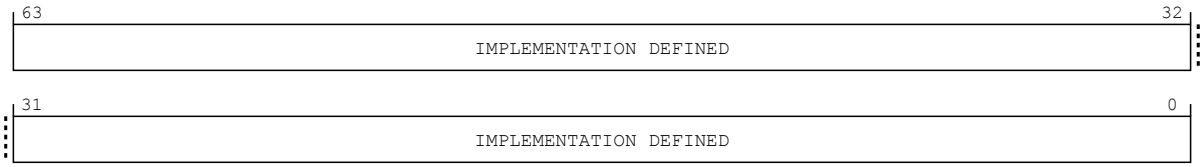

AMAIR_EL2 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `AMAIR_EL2` or `AMAIR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMAIR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = AMAIR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR_EL2();
end;
```

MSR AMAIR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AMAIR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    AMAIR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, AMAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().AMAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x148);
    else
        X{64}(t) = AMAIR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = AMAIR_EL2();
    else
        X{64}(t) = AMAIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR_EL1();
end;
```

When FEAT_VHE is implemented MSR AMAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().AMAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x148) = X{64}(t);
    else
        AMAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        AMAIR_EL2() = X{64}(t);
    else
        AMAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMAIR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.19 AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)

The AMAIR_EL3 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to AMAIR_EL3 are UNDEFINED.

Attributes

AMAIR_EL3 is a 64-bit register.

Field descriptions

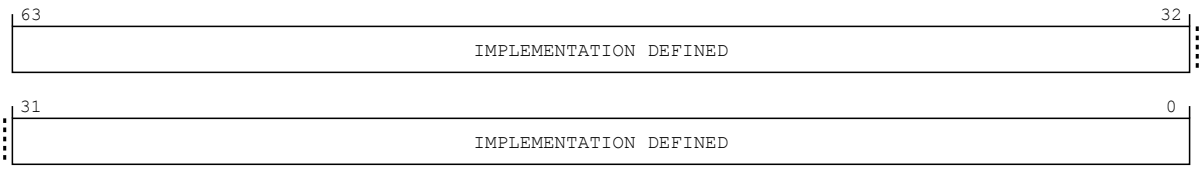

AMAIR_EL3 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMAIR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0011	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMAIR_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR AMAIR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0011	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().AMAIR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        AMAIR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.20 APDAKeyHi_EL1, Pointer Authentication Key A for Data (bits[127:64])

The APDAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key A used for authentication of data pointer values.

Note

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APDAKeyHi_EL1 are UNDEFINED.

Attributes

APDAKeyHi_EL1 is a 64-bit register.

Field descriptions

63 … 32	31 … 0
64 bit value, bits[127:64] of the 128 bit pointer authentication key value	64 bit value, bits[127:64] of the 128 bit pointer authentication key value

APDAKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDAKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APDAKeyHi_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APDAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDAKeyHi_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDAKeyHi_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APDAKeyHi_EL1();
end;
```

```
MSR APDAKeyHi_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APDAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDAKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDAKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APDAKeyHi_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.21 APDAKeyLo_EL1, Pointer Authentication Key A for Data (bits[63:0])

The APDAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key A used for authentication of data pointer values.

Note

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APDAKeyLo_EL1 are UNDEFINED.

Attributes

APDAKeyLo_EL1 is a 64-bit register.

Field descriptions

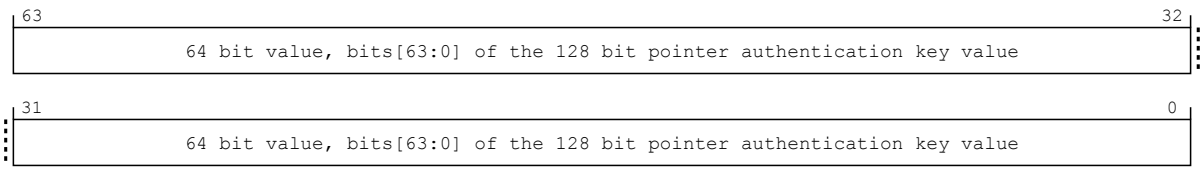

APDAKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APDAKeyLo_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b000

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APDAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDAKeyLo_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDAKeyLo_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APDAKeyLo_EL1();
end;
```

```
MSR APDAKeyLo_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b000

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APDAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDAKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDAKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APDAKeyLo_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.22 APDBKeyHi_EL1, Pointer Authentication Key B for Data (bits[127:64])

The APDBKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key B used for authentication of data pointer values.

Note

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APDBKeyHi_EL1 are UNDEFINED.

Attributes

APDBKeyHi_EL1 is a 64-bit register.

Field descriptions

63 … 32	31 … 0
64 bit value, bits[127:64] of the 128 bit pointer authentication key value	64 bit value, bits[127:64] of the 128 bit pointer authentication key value

APDBKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDBKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APDBKeyHi_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APDBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDBKeyHi_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDBKeyHi_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APDBKeyHi_EL1();
end;
```

```
MSR APDBKeyHi_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APDBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDBKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDBKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APDBKeyHi_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.23 APDBKeyLo_EL1, Pointer Authentication Key B for Data (bits[63:0])

The APDBKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key B used for authentication of data pointer values.

Note

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APDBKeyLo_EL1 are UNDEFINED.

Attributes

APDBKeyLo_EL1 is a 64-bit register.

Field descriptions

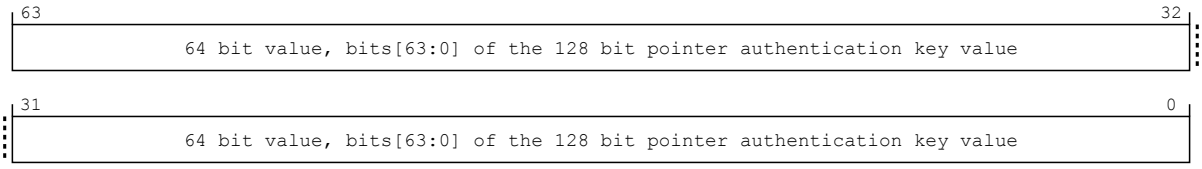

APDBKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDBKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APDBKeyLo_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APDBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDBKeyLo_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APDBKeyLo_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APDBKeyLo_EL1();
end;
```

```
MSR APDBKeyLo_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APDBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDBKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APDBKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APDBKeyLo_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.24 APGAKeyHi_EL1, Pointer Authentication Key A for Code (bits[127:64])

The APGAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APGAKeyHi_EL1 are UNDEFINED.

Attributes

APGAKeyHi_EL1 is a 64-bit register.

Field descriptions

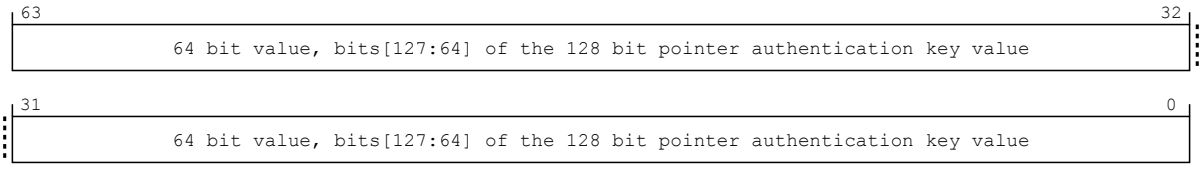

APGAKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APGAKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APGAKeyHi_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APGAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APGAKeyHi_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APGAKeyHi_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APGAKeyHi_EL1();
end;
```

```
MSR APGAKeyHi_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APGAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APGAKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APGAKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APGAKeyHi_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.25 APGAKeyLo_EL1, Pointer Authentication Key A for Code (bits[63:0])

The APGAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APGAKeyLo_EL1 are UNDEFINED.

Attributes

APGAKeyLo_EL1 is a 64-bit register.

Field descriptions

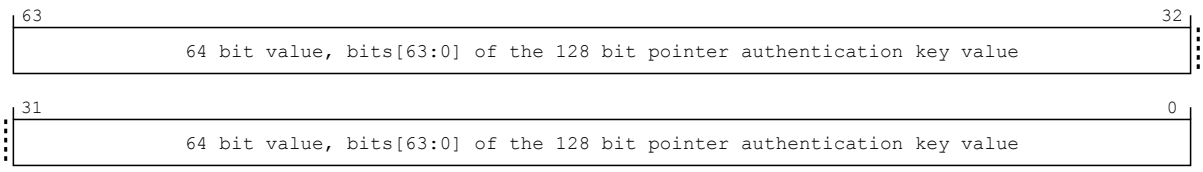

APGAKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APGAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APGAKeyLo_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APGAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APGAKeyLo_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APGAKeyLo_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APGAKeyLo_EL1();
end;
```

```
MSR APGAKeyLo_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APGAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APGAKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APGAKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APGAKeyLo_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.26 APIAKeyHi_EL1, Pointer Authentication Key A for Instruction (bits[127:64])

The APIAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key A used for authentication of instruction pointer values.

Note

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APIAKeyHi_EL1 are UNDEFINED.

Attributes

APIAKeyHi_EL1 is a 64-bit register.

Field descriptions

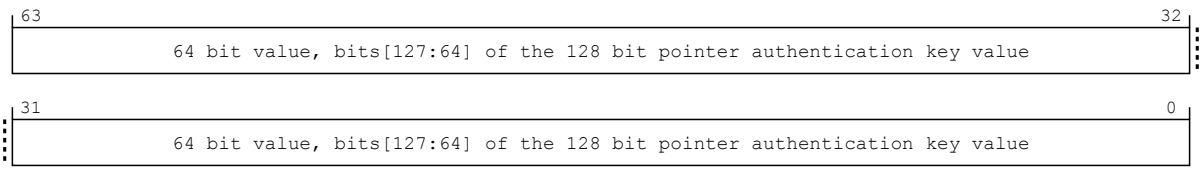

APIAKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIAKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APIAKeyHi_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APIAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIAKeyHi_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIAKeyHi_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APIAKeyHi_EL1();
end;
```

```
MSR APIAKeyHi_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APIAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIAKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIAKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APIAKeyHi_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.27 APIAKeyLo_EL1, Pointer Authentication Key A for Instruction (bits[63:0])

The APIAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key A used for authentication of instruction pointer values.

Note

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APIAKeyLo_EL1 are UNDEFINED.

Attributes

APIAKeyLo_EL1 is a 64-bit register.

Field descriptions

63 … 32	31 … 0
64 bit value, bits[63:0] of the 128 bit pointer authentication key value	64 bit value, bits[63:0] of the 128 bit pointer authentication key value

APIAKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APIAKeyLo_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APIAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIAKeyLo_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIAKeyLo_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APIAKeyLo_EL1();
end;
```

```
MSR APIAKeyLo_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APIAKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIAKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIAKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APIAKeyLo_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.28 APIBKeyHi_EL1, Pointer Authentication Key B for Instruction (bits[127:64])

The APIBKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key B used for authentication of instruction pointer values.

Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APIBKeyHi_EL1 are UNDEFINED.

Attributes

APIBKeyHi_EL1 is a 64-bit register.

Field descriptions

63 … 32	31 … 0
64 bit value, bits[127:64] of the 128 bit pointer authentication key value	64 bit value, bits[127:64] of the 128 bit pointer authentication key value

APIBKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIBKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APIBKeyHi_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APIBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIBKeyHi_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIBKeyHi_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APIBKeyHi_EL1();
end;
```

```
MSR APIBKeyHi_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APIBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIBKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIBKeyHi_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APIBKeyHi_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.29 APIBKeyLo_EL1, Pointer Authentication Key B for Instruction (bits[63:0])

The APIBKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key B used for authentication of instruction pointer values.

Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

Configuration

This register is present only when FEAT_PAuth is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to APIBKeyLo_EL1 are UNDEFINED.

Attributes

APIBKeyLo_EL1 is a 64-bit register.

Field descriptions

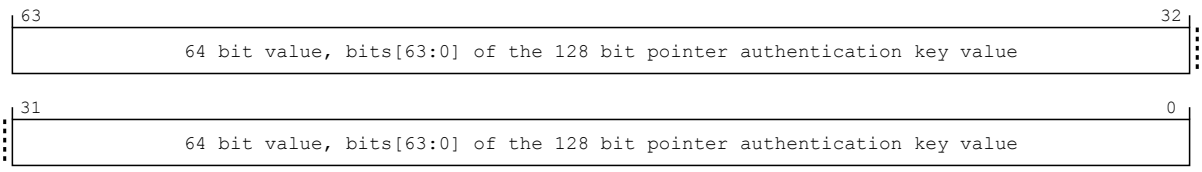

APIBKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIBKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, APIBKeyLo_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().APIBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIBKeyLo_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = APIBKeyLo_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = APIBKeyLo_EL1();
end;
```

```
MSR APIBKeyLo_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().APK == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().APIBKey == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIBKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().APK == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().APK == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        APIBKeyLo_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    APIBKeyLo_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.30 CCSIDR2_EL1, Current Cache Size ID Register 2

The CCSIDR2_EL1 characteristics are:

Purpose

Provides the information about the architecture of the currently selected cache from bits[63:32] of CCSIDR_EL1.

Configuration

In an implementation which does not support AArch32 at EL1, it is IMPLEMENTATION DEFINED whether reading this register gives an UNKNOWN value or is UNDEFINED.

The implementation includes one CCSIDR2_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

AArch64 System register CCSIDR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register CCSIDR2[31:0].

This register is present only when FEAT_CCIDX is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CCSIDR2_EL1 are UNDEFINED.

Attributes

CCSIDR2_EL1 is a 64-bit register.

Field descriptions

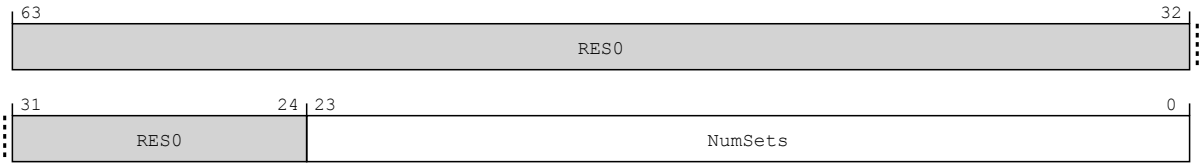

Bits [63:24]

Reserved, RES0.

NumSets, bits [23:0]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Accessing CCSIDR2_EL1

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR2_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2_EL1 read is treated as NOP.
- The CCSIDR2_EL1 read is UNDEFINED. If FEAT_IDST is implemented, this is permitted to be reported with EC syndrome value `0x18`.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CCSIDR2_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_CCIDX) && IsFeatureImplemented(FEAT_AA64)) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CCSIDR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CCSIDR2_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CCSIDR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.31 CCSIDR_EL1, Current Cache Size ID Register

The CCSIDR_EL1 characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

Configuration

The implementation includes one CCSIDR_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

AArch64 System register CCSIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CCSIDR[31:0].

AArch64 System register CCSIDR_EL1 bits [63:32] are architecturally mapped to AArch32 System register CCSIDR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CCSIDR_EL1 are UNDEFINED.

Attributes

CCSIDR_EL1 is a 64-bit register.

Field descriptions

When FEAT_CCIDX is implemented:

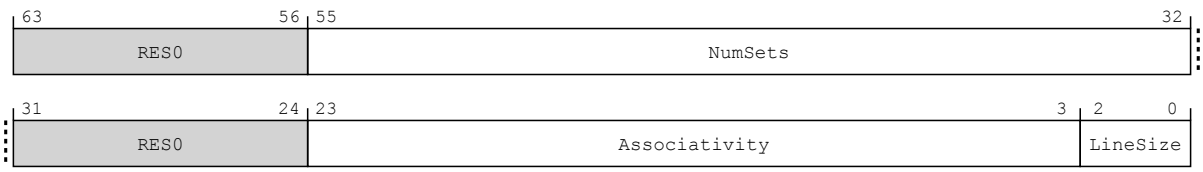

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [63:56]

Reserved, RES0.

NumSets, bits [55:32]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Bits [31:24]

Reserved, RES0.

Associativity, bits [23:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

- For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

When FEAT_MTE2 is implemented, where a cache only holds Allocation tags, this field is RES0.

Otherwise:

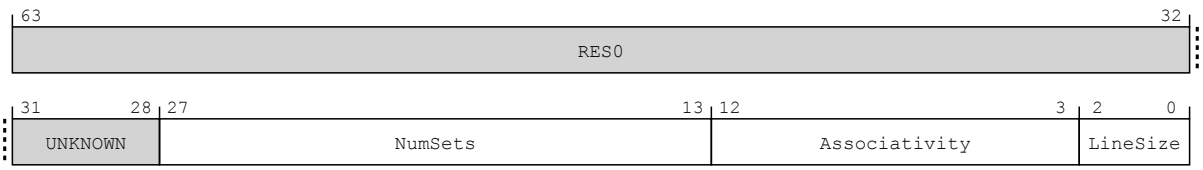

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [63:32]

Reserved, RES0.

Bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

- For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

When FEAT_MTE2 is implemented, where a cache only holds Allocation tags, this field is RES0.

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CCSIDR_EL1

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR_EL1 read is treated as NOP.
- The CCSIDR_EL1 read is UNDEFINED. If FEAT_IDST is implemented, this is permitted to be reported with EC syndrome value `0x18`.
- The CCSIDR_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CCSIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().CCSIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CCSIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CCSIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CCSIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.32 CLIDR_EL1, Cache Level ID Register

The CLIDR_EL1 characteristics are:

Purpose

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

Configuration

AArch64 System register CLIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CLIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CLIDR_EL1 are UNDEFINED.

Attributes

CLIDR_EL1 is a 64-bit register.

Field descriptions

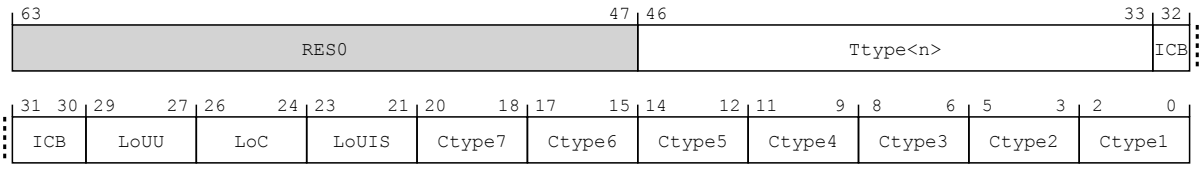

Bits [63:47]

Reserved, RES0.

Ttype<n> , bits [2(n-1)+34:2(n-1)+33], for n = 7 to 1

When FEAT_MTE2 is implemented:

Tag cache type. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy.

Ttype<n>	Meaning
0b00	No Tag Cache.
0b01	Separate Allocation Tag Cache.
0b10	Unified Allocation Tag and Data cache, Allocation Tags and Data in unified lines.
0b11	Unified Allocation Tag and Data cache, Allocation Tags and Data in separate lines.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ICB, bits [32:30]

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ICB	Meaning
0b000	Not disclosed by this mechanism.
0b001	L1 cache is the highest Inner Cacheable level.
0b010	L2 cache is the highest Inner Cacheable level.
0b011	L3 cache is the highest Inner Cacheable level.
0b100	L4 cache is the highest Inner Cacheable level.
0b101	L5 cache is the highest Inner Cacheable level.
0b110	L6 cache is the highest Inner Cacheable level.
0b111	L7 cache is the highest Inner Cacheable level.

Access to this field is RO.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

This field does not describe the requirements for instruction cache invalidation. See CTR_EL0.DIC.

Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoC, bits [26:24]

Level of Coherence for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

This field does not describe the requirements for instruction cache invalidation. See CTR_EL0.DIC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Ctype<n> , bits [3(n-1)+2:3(n-1)], for n = 7 to 1

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

Ctype<n>	Meaning
0b000	No cache.
0b001	Instruction cache only.
0b010	Data cache only.
0b011	Separate instruction and data caches.
0b100	Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 0b000, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 0b000, the values of Ctype4 to Ctype7 must be ignored.

Accessing CLIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CLIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().CLIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CLIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CLIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CLIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.33 CONTEXTIDR_EL1, Context ID Register (EL1)

The CONTEXTIDR_EL1 characteristics are:

Purpose

Identifies the current Process Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configuration

AArch64 System register CONTEXTIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CONTEXTIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CONTEXTIDR_EL1 are UNDEFINED.

Attributes

CONTEXTIDR_EL1 is a 64-bit register.

Field descriptions

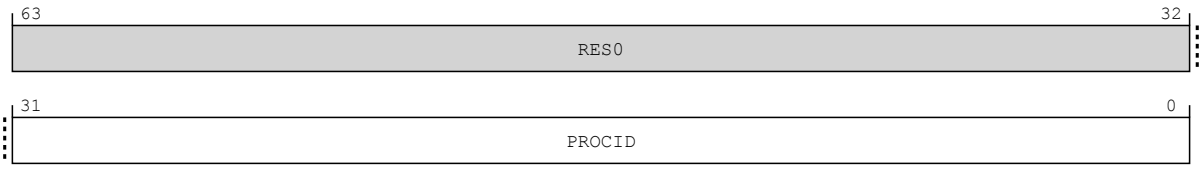

Bits [63:32]

Reserved, RES0.

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

Note

In AArch32 state, when TTBCR.EAE is set to 0, CONTEXTIDR.ASID holds the ASID.
In AArch64 state, CONTEXTIDR_EL1 is independent of the ASID, and for the EL1&0 translation regime either TTBR0_EL1 or TTBR1_EL1 holds the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CONTEXTIDR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CONTEXTIDR_EL1` or `CONTEXTIDR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CONTEXTIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().CONTEXTIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x108);
    else
        X{64}(t) = CONTEXTIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CONTEXTIDR_EL2();
    else
        X{64}(t) = CONTEXTIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CONTEXTIDR_EL1();
end;
```

```
MSR CONTEXTIDR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().CONTEXTIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x108) = X{64}(t);
    else
        CONTEXTIDR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CONTEXTIDR_EL2() = X{64}(t);
    else
        CONTEXTIDR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    CONTEXTIDR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CONTEXTIDR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x108);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CONTEXTIDR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CONTEXTIDR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CONTEXTIDR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x108) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ELIsInHost(EL2) then
        CONTEXTIDR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CONTEXTIDR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.34 CONTEXTIDR_EL2, Context ID Register (EL2)

The CONTEXTIDR_EL2 characteristics are:

Purpose

Identifies the current Process Identifier for EL2.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_Debugv8p1 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CONTEXTIDR_EL2 are UNDEFINED.

Attributes

CONTEXTIDR_EL2 is a 64-bit register.

Field descriptions

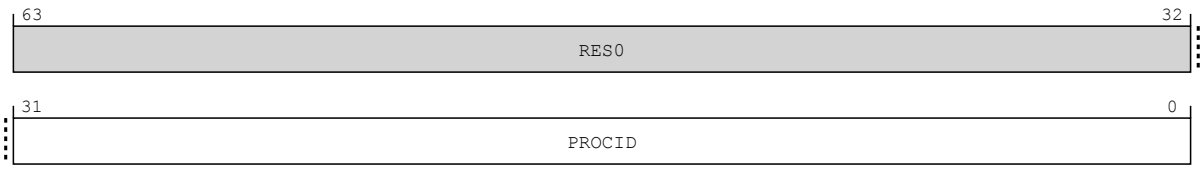

Bits [63:32]

Reserved, RES0.

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CONTEXTIDR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CONTEXTIDR_EL2` or `CONTEXTIDR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CONTEXTIDR_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_Debugv8p1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CONTEXTIDR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CONTEXTIDR_EL2();
end;
```

MSR CONTEXTIDR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_Debugv8p1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CONTEXTIDR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CONTEXTIDR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CONTEXTIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGRTR_EL2().CONTEXTIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x108);
    else
        X{64}(t) = CONTEXTIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CONTEXTIDR_EL2();
    else
        X{64}(t) = CONTEXTIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CONTEXTIDR_EL1();
end;
```

When FEAT_VHE is implemented MSR CONTEXTIDR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGWTR_EL2().CONTEXTIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x108) = X{64}(t);
    else
        CONTEXTIDR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CONTEXTIDR_EL2() = X{64}(t);
    else
        CONTEXTIDR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CONTEXTIDR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.35 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

AArch64 System register CPACR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CPACR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CPACR_EL1 are UNDEFINED.

Attributes

CPACR_EL1 is a 64-bit register.

Field descriptions

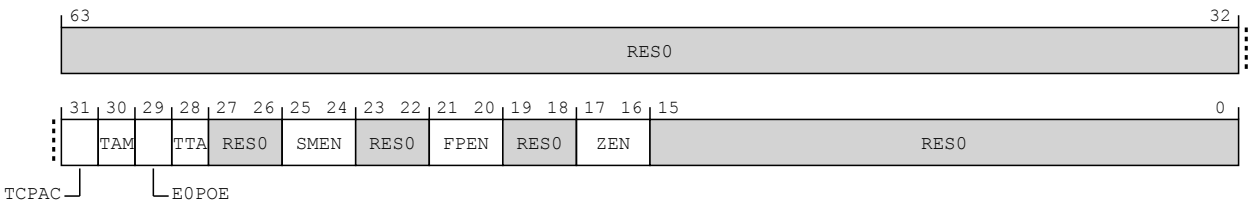

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

When FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.TCPAC == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

TAM, bit [30]

When FEAT_AMUv1 is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.TAM == ‘1’, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

E0POE, bit [29]

When FEAT_S1POE is implemented:

Enable access to POR_EL0.

Traps EL0 accesses to POR_EL0, from AArch64 state only to EL1, or to EL2 when it is implemented and enabled in the current Security state and HCR_EL2.TGE is 1. The exception is reported using EC syndrome value `0x18`.

E0POE	Meaning
`0b0`	This control causes EL0 access to POR_EL0 to be trapped.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.E0POE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TTA, bit [28]

When System register access to the trace unit registers is implemented:

Traps EL0 and EL1 System register accesses to all implemented trace System registers from both Execution states to EL1, or to EL2 when it is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to trace System registers are trapped, reported using EC syndrome value `0x18`.
- In AArch32 state, MRC and MCR accesses to trace System registers are trapped, reported using EC syndrome value `0x05`.
- In AArch32 state, MRRC and MCRR accesses to trace System registers are trapped, reported using EC syndrome value `0x0C`.

TTA	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	This control causes EL0 and EL1 System register accesses to all implemented trace System registers to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- The ETMv4 architecture and ETE do not permit EL0 to access the trace System registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace System registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.TTA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [27:26]

Reserved, RES0.

SMEN, bits [25:24]

When FEAT_SME is implemented:

Traps execution at EL1 and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR or SMCR_EL1 System registers to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

When instructions that directly access the SVCR System register are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

The exception is reported using EC syndrome value `0x1D`, with an ISS code of `0x0000000`.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

SMEN	Meaning
`0b00`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b01`	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
`0b10`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b11`	This control does not cause execution of any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.SMEN == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [23:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps execution at EL1 and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL1, reported using EC syndrome value `0x07`, or to EL2 reported using EC syndrome value `0x00` when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to FPMR, FPCR, FPSR, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers.

Traps execution at EL1 and EL0 of SME and SVE instructions to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using EC syndrome value `0x07`.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

FPEN	Meaning
`0b00`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b01`	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
`0b10`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b11`	This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.FPEN is not `0b11`.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.FPEN == '1', access to this field is RO.

Bits [19:18]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Traps execution at EL1 and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 System register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

The exception is reported using EC syndrome value `0x19`.

A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

ZEN	Meaning
`0b00`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b01`	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
`0b10`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b11`	This control does not cause execution of any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and CPACRMASK_EL1.ZEN == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [15:0]

Reserved, RES0.

Accessing CPACR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CPACR_EL1` or `CPACR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to CPACR_EL1 are masked by CPACRMASK_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CPACR_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b0001`	`0b0000`	`0b010`

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TCPAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().CPACR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x100);
    else
        X{64}(t) = CPACR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = CPTR_EL2();
    else
        X{64}(t) = CPACR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPACR_EL1();
end;
```

```
MSR CPACR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TCPAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().CPACR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x100) = X{64}(t);
    else
        CPACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        CPTR_EL2() = X{64}(t);
    else
        CPACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPACR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CPACR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x100);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
            Undefined();
        elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = CPACR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CPACR_EL1();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CPACR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x100) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
            Undefined();
        elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            CPACR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CPACR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```

When FEAT_SRMASK is implemented MRS <Xt>, CPACRALIAS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TCPAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
         ↪'0') || HFGRTR2_EL2().nCPACRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x100);
    else
        X{64}(t) = CPACR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = CPTR_EL2();
    else
        X{64}(t) = CPACR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPACR_EL1();
end;
```

When FEAT_SRMASK is implemented MSR CPACRALIAS_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TCPAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
         ↪'0') || HFGWTR2_EL2().nCPACRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x100) = X{64}(t);
    else
        CPACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        CPTR_EL2() = X{64}(t);
    else
        CPACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPACR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.36 CPACRMASK_EL1, Architectural Feature Access Control Masking Register

The CPACRMASK_EL1 characteristics are:

Purpose

Mask register to prevent updates of fields in CPACR_EL1 on writes to CPACR_EL1 or CPACRALIAS_EL1.

Configuration

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CPACRMASK_EL1 are UNDEFINED.

Attributes

CPACRMASK_EL1 is a 64-bit register.

Field descriptions

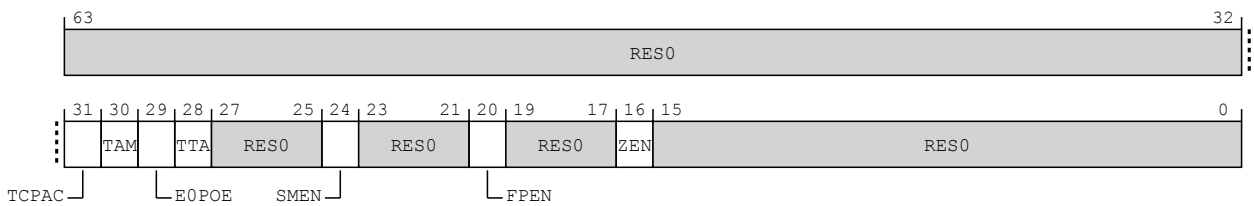

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

When FEAT_NV2p1 is implemented:

Mask bit for TCPAC.

TCPAC	Meaning
0b0	CPACR_EL1.TCPAC is writable.
0b1	CPACR_EL1.TCPAC is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TAM, bit [30]

When FEAT_AMUv1 is implemented and FEAT_NV2p1 is implemented:

Mask bit for TAM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TAM	Meaning
0b0	CPACR_EL1.TAM is writable.
0b1	CPACR_EL1.TAM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0POE, bit [29]

When FEAT_S1POE is implemented:

Mask bit for E0POE.

E0POE	Meaning
0b0	CPACR_EL1.E0POE is writable.
0b1	CPACR_EL1.E0POE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTA, bit [28]

When System register access to the trace unit registers is implemented:

Mask bit for TTA.

TTA	Meaning
0b0	CPACR_EL1.TTA is writable.
0b1	CPACR_EL1.TTA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [27:25]

Reserved, RES0.

SMEN, bit [24]

When FEAT_SME is implemented:

Mask bit for SMEN.

SMEN	Meaning
0b0	CPACR_EL1.SMEN is writable.
0b1	CPACR_EL1.SMEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [23:21]

Reserved, RES0.

FPEN, bit [20]

Mask bit for FPEN.

FPEN	Meaning
0b0	CPACR_EL1.FPEN is writable.
0b1	CPACR_EL1.FPEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [19:17]

Reserved, RES0.

ZEN, bit [16]

When FEAT_SVE is implemented:

Mask bit for ZEN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ZEN	Meaning
0b0	CPACR_EL1.ZEN is writable.
0b1	CPACR_EL1.ZEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [15:0]

Reserved, RES0.

Accessing CPACRMASK_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name CPACRMASK_EL1 or CPACRMASK_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CPACRMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nCPACRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x320);
    else
        X{64}(t) = CPACRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = CPTRMASK_EL2();
    else
        X{64}(t) = CPACRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPACRMASK_EL1();
end;
```

MSR CPACRMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nCPACRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x320) = X{64}(t);
    elsif !IsZero(CPACRMASK_EL1()) then
        Undefined();
    else
        CPACRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(CPTRMASK_EL2()) then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            CPTRMASK_EL2() = X{64}(t);
        end;
    else
        CPACRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPACRMASK_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CPACRMASK_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x320);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = CPACRMASK_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CPACRMASK_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CPACRMASK_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x320) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            CPACRMASK_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CPACRMASK_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.37 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of accesses to CPACR, CPACR_EL1, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register CPTR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HCPTR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CPTR_EL2 are UNDEFINED.

Attributes

CPTR_EL2 is a 64-bit register.

Field descriptions

When EffectiveHCR_EL2_E2H() == '1':

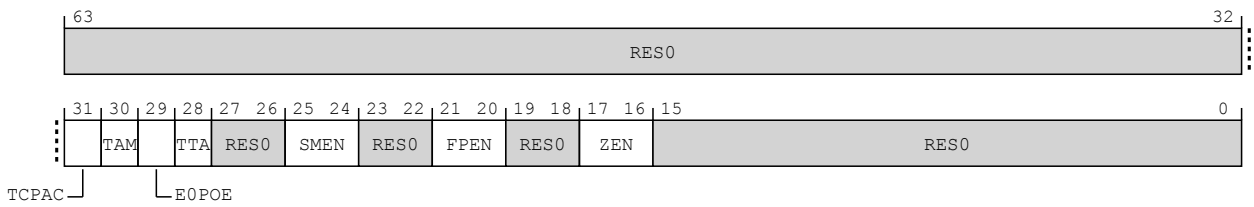

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x18`.

In AArch32 state, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x03`.

TCPAC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2, when EL2 is enabled in the current Security state.

When the Effective value of HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.TCPAC == '1', access to this field is RO.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitors System registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, MRRC or MCRR accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using EC syndrome value `0x04`.

TAM	Meaning
`0b0`	Accesses from EL1 and EL0 to Activity Monitors System registers are not trapped.
`0b1`	Accesses from EL1 and EL0 to Activity Monitors System registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.TAM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

E0POE, bit [29]

When FEAT_S1POE is implemented:

Enable access to POR_EL0.

Traps EL0 accesses to POR_EL0 to EL2, from AArch64 state only. The exception is reported using EC syndrome value `0x18`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E0POE	Meaning
0b0	This control causes EL0 access to POR_EL0 to be trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.E0POE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TTA, bit [28]

When System register access to the trace unit registers is implemented:

Traps System register accesses to all implemented trace System registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace System registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace System registers with cpnum=14, opc1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x05.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt at EL0, EL1 or EL2, to execute a System register access to an implemented trace System register is trapped to EL2, when EL2 is enabled in the current Security state, unless HCR_EL2.TGE is 0 and it is trapped by CPACR.NSTRCDIS or CPACR_EL1.TTA. When HCR_EL2.TGE is 1, any attempt at EL0 or EL2 to execute a System register access to an implemented trace System register is trapped to EL2, when EL2 is enabled in the current Security state.

Note

The ETMv4 architecture and ETE do not permit EL0 to access the trace System registers. If the trace unit implements FEAT_ETMv4 or ETE, EL0 accesses to the trace System registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
EL2 does not provide traps on trace register accesses through the optional Memory-mapped interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.TTA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:26]

Reserved, RES0.

SMEN, bits [25:24]

When FEAT_SME is implemented:

Traps execution at EL2, EL1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

The exception is reported using EC syndrome value `0x1D`, with an ISS code of `0x0000000`.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

SMEN	Meaning
`0b00`	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
`0b01`	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
`0b10`	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
`0b11`	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.SMEN == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [23:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps execution at EL2, EL1, and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x07`.

Traps execution at EL2, EL1, and EL0 of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x07`.

When FEAT_FPMR is implemented, traps execution at EL2, EL1, and EL0 of instructions that access FPMR to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x07`

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

A trap taken as a result of CPTR_EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPEN	Meaning
0b00	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b01	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.FPEN == '1', access to this field is RO.

Bits [19:18]

Reserved, RES0.

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 or ZCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

The exception is reported using EC syndrome value 0x19.

A trap taken as a result of CPTR_EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

ZEN	Meaning
0b00	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b01	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and CPTRMASK_EL2.ZEN == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [15:0]

Reserved, RES0.

Otherwise:

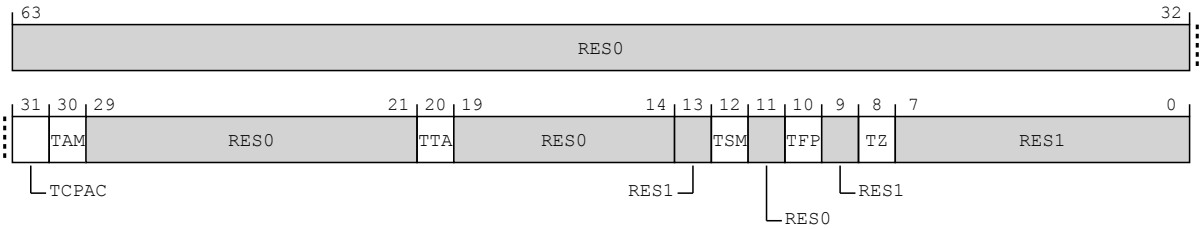

This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x18`.

In AArch32 state, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x03`.

TCPAC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to the following registers are trapped to EL2, when EL2 is enabled in the current Security state: • CPACR_EL1. • CPACR.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

Note

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitors System registers to EL2, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, MCR or MRC accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, MCRR or MRRC accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using EC syndrome value `0x04`.

TAM	Meaning
`0b0`	Accesses from EL1 and EL0 to Activity Monitors System registers are not trapped.
`0b1`	Accesses from EL1 and EL0 to Activity Monitors System registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

When System register access to the trace unit registers is implemented:

Traps System register accesses to all implemented trace System registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace System registers with op0=2, op1=1, and CRn<`0b1000` are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, MRC or MCR accesses to trace System registers with cpnum=14, opc1=1, and CRn<`0b1000` are trapped to EL2, reported using EC syndrome value `0x05`.

TTA	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace System register is trapped to EL2, when EL2 is enabled in the current Security state, unless it is trapped by one of the following controls: • CPACR_EL1.TTA. • CPACR.TRCDIS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- FEAT_ETMv4 and FEAT_ETE do not permit EL0 to access the trace System registers. EL0 accesses to the trace System registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:14]

Reserved, RES0.

Bit [13]

Reserved, RES1.

TSM, bit [12]

When FEAT_SME is implemented:

Traps execution at EL2, EL1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

The exception is reported using EC syndrome value `0x1D`, with an ISS code of `0x0000000`.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.TSM has precedence over a trap taken as a result of CPTR_EL2.TFP.

TSM	Meaning
`0b0`	This control does not cause execution of any instructions to be trapped.
`0b1`	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x07`:
  - FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
  - If FEAT_FPMR is implemented, FPMR.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x07`:
  - MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating-point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

Traps execution at the same Exception levels of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x07`.

A trap taken as a result of CPTR_EL2.TSM has precedence over a trap taken as a result of CPTR_EL2.TFP.

A trap taken as a result of CPTR_EL2.TZ has precedence over a trap taken as a result of CPTR_EL2.TFP.

TFP	Meaning
`0b0`	This control does not cause execution of any instructions to be trapped.
`0b1`	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

Note

FPEXC32_EL2 is not accessible from EL0 using AArch64.
FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES1.

TZ, bit [8]

When FEAT_SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL2 or ZCR_EL1 System registers to EL2, when EL2 is enabled in the current Security state.

The exception is reported using EC syndrome value `0x19`.

A trap taken as a result of CPTR_EL2.TZ has precedence over a trap taken as a result of CPTR_EL2.TFP.

TZ	Meaning
`0b0`	This control does not cause execution of any instructions to be trapped.
`0b1`	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES1.

Bits [7:0]

Reserved, RES1.

Accessing CPTR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CPTR_EL2` or `CPACR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to CPTR_EL2 are masked by CPTRMASK_EL2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CPTR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = CPTR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPTR_EL2();
end;
```

```
MSR CPTR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        CPTR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPTR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TCPAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
     ↪'1') && HFGRTR_EL2().CPACR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x100);
    else
        X{64}(t) = CPACR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = CPTR_EL2();
        else
            X{64}(t) = CPACR_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = CPACR_EL1();
    end;
```

When FEAT_VHE is implemented MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TCPAC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
     ↪'1') && HFGWTR_EL2().CPACR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x100) = X{64}(t);
    else
        CPACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TCPAC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        CPTR_EL2() = X{64}(t);
    else
        CPACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPACR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.38 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

Purpose

Controls trapping to EL3 of accesses to CPACR, CPACR_EL1, HCPTR, CPTR_EL2, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CPTR_EL3 are UNDEFINED.

Attributes

CPTR_EL3 is a 64-bit register.

Field descriptions

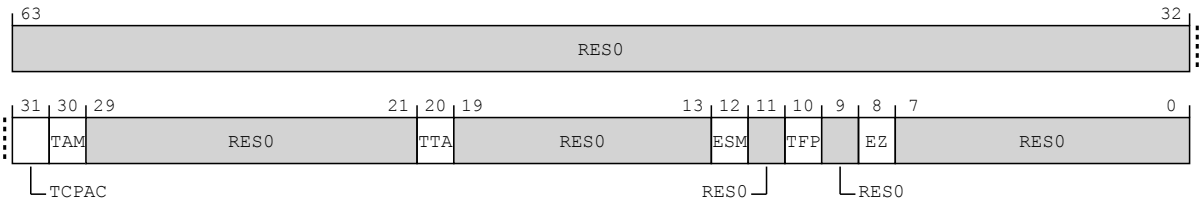

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Traps all of the following to EL3, from both Execution states and any Security state.

- EL2 accesses to CPTR_EL2, reported using EC syndrome value `0x18`, or HCPTR, reported using EC syndrome value `0x03`.
- EL2 and EL1 accesses to CPACR_EL1 reported using EC syndrome value `0x18`, or CPACR reported using EC syndrome value `0x03`.

When CPTR_EL3.TCPAC is:

TCPAC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3, unless they are trapped by CPTR_EL2.TCPAC.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL2, EL1, and EL0 accesses to all Activity Monitors System registers to EL3.

Accesses to the Activity Monitors System registers are trapped as follows:

- In AArch64 state, the following registers are trapped to EL3 and reported using EC syndrome value `0x18`:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, accesses with MRC or MCR to the following registers reported using EC syndrome value `0x03`:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, accesses with MRRC or MCRR to the following registers, reported using EC syndrome value `0x04`:
  - AMEVCNTR0<n>, AMEVCNTR1<n>.

TAM	Meaning
`0b0`	Accesses from EL2, EL1, and EL0 to Activity Monitors System registers are not trapped.
`0b1`	Accesses from EL2, EL1, and EL0 to Activity Monitors System registers are trapped to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

When System register access to the trace unit registers is implemented:

Traps System register accesses. Accesses to the trace System registers, from all Exception levels, any Security state, and both Execution states are trapped to EL3 as follows:

- In AArch64 state, trace System registers with op0=2, op1=1, and CRn<`0b1000` are trapped to EL3 and reported using EC syndrome value `0x18`.
- In AArch32 state, accesses using MCR or MRC to the trace System registers with cpnum=14, opc1=1, and CRn<`0b1000` are reported using EC syndrome value `0x05`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any System register access to the trace System registers is trapped to EL3, unless it is trapped by CPACR.TRCDIS, CPACR_EL1.TTA, or CPTR_EL2.TTA.

Note

The ETMv4 architecture and ETE do not permit EL0 to access the trace System registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace System registers are UNDEFINED, and any resulting exception is higher priority than this trap exception.
EL3 does not provide traps on trace System register accesses through the Memory-mapped interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see 'Configurable instruction controls'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:13]

Reserved, RES0.

ESM, bit [12]

When FEAT_SME is implemented:

Traps execution of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SMCR_EL1, SMCR_EL2, SMCR_EL3, SMPRI_EL1, SMPRIMAP_EL2, or SVCR System registers, from all Exception levels and any Security state, to EL3.

When instructions that directly access the SVCR System register are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

When direct accesses to SMPRI_EL1 and SMPRIMAP_EL2 are trapped, the exception is reported using EC syndrome value `0x18`. Otherwise, the exception is reported using EC syndrome value `0x1D`, with an ISS code of `0x0000000`.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

ESM	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from all Exception levels, any Security state, and both Execution states, to EL3.

This includes the following registers, all reported using EC syndrome value `0x07`:

- FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers.
- If FEAT_FPMR is implemented, FPMR.
- MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers.
- VMSR accesses to FPSID.

Permitted VMSR accesses to FPSID are ignored, but for the purposes of this trap the architecture defines a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

Traps execution at all Exception levels of SME and SVE instructions to EL3 from any Security state. The exception is reported using EC syndrome value `0x07`.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

A trap taken as a result of CPTR_EL3.EZ has precedence over a trap taken as a result of CPTR_EL3.TFP.

TFP	Meaning
`0b0`	This control does not cause execution of any instructions to be trapped.
`0b1`	This control causes execution of these instructions at all Exception levels to be trapped.

Note

FPEXC32_EL2 is not accessible from EL0 using AArch64.
FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

EZ, bit [8]

When FEAT_SVE is implemented:

Traps execution of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL3, ZCR_EL2, or ZCR_EL1 System registers, from all Exception levels and any Security state, to EL3.

The exception is reported using EC syndrome value `0x19`.

A trap taken as a result of CPTR_EL3.EZ has precedence over a trap taken as a result of CPTR_EL3.TFP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EZ	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [7:0]

Reserved, RES0.

Accessing CPTR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CPTR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPTR_EL3();
end;
```

```
MSR CPTR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    CPTR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.39 CPTRMASK_EL2, Architectural Feature Trap Masking Register

The CPTRMASK_EL2 characteristics are:

Purpose

Mask register to prevent updates of fields in CPTR_EL2 on writes.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CPTRMASK_EL2 are UNDEFINED.

Attributes

CPTRMASK_EL2 is a 64-bit register.

Field descriptions

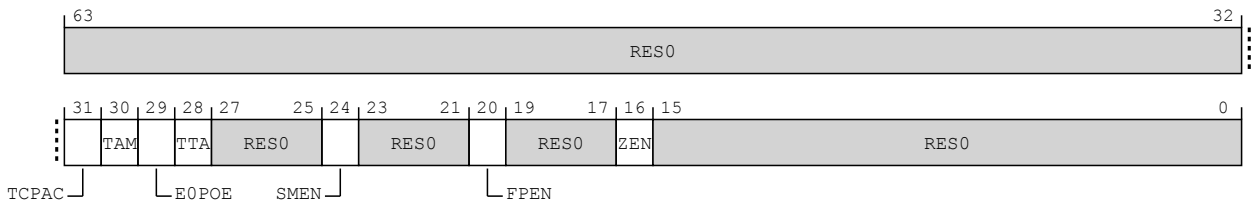

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Mask bit for TCPAC.

TCPAC	Meaning
0b0	CPTR_EL2.TCPAC is writable.
0b1	CPTR_EL2.TCPAC is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Mask bit for TAM.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TAM	Meaning
0b0	CPTR_EL2.TAM is writable.
0b1	CPTR_EL2.TAM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0POE, bit [29]

When FEAT_S1POE is implemented:

Mask bit for E0POE.

E0POE	Meaning
0b0	CPTR_EL2.E0POE is writable.
0b1	CPTR_EL2.E0POE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTA, bit [28]

When System register access to the trace unit registers is implemented:

Mask bit for TTA.

TTA	Meaning
0b0	CPTR_EL2.TTA is writable.
0b1	CPTR_EL2.TTA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:25]

Reserved, RES0.

SMEN, bit [24]

When FEAT_SME is implemented:

Mask bit for SMEN.

SMEN	Meaning
0b0	CPTR_EL2.SMEN is writable.
0b1	CPTR_EL2.SMEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [23:21]

Reserved, RES0.

FPEN, bit [20]

Mask bit for FPEN.

FPEN	Meaning
0b0	CPTR_EL2.FPEN is writable.
0b1	CPTR_EL2.FPEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [19:17]

Reserved, RES0.

ZEN, bit [16]

When FEAT_SVE is implemented:

Mask bit for ZEN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ZEN	Meaning
0b0	CPTR_EL2.ZEN is writable.
0b1	CPTR_EL2.ZEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [15:0]

Reserved, RES0.

Accessing CPTRMASK_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name CPTRMASK_EL2 or CPACRMASK_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CPTRMASK_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = CPTRMASK_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPTRMASK_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR CPTRMASK_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsZero(CPTRMASK_EL2()) then
        Undefined();
    else
        CPTRMASK_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPTRMASK_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, CPACRMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nCPACRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x320);
    else
        X{64}(t) = CPACRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = CPTRMASK_EL2();
    else
        X{64}(t) = CPACRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CPACRMASK_EL1();
end;
```

MSR CPACRMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nCPACRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x320) = X{64}(t);
    elsif !IsZero(CPACRMASK_EL1()) then
        Undefined();
    else
        CPACRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(CPTRMASK_EL2()) then
            Undefined();
        else
            CPTRMASK_EL2() = X{64}(t);
        end;
    else
        CPACRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CPACRMASK_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.40 CSSELR_EL1, Cache Size Selection Register

The CSSELR_EL1 characteristics are:

Purpose

Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type (either instruction or data cache).

Configuration

AArch64 System register CSSELR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CSSELR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CSSELR_EL1 are UNDEFINED.

Attributes

CSSELR_EL1 is a 64-bit register.

Field descriptions

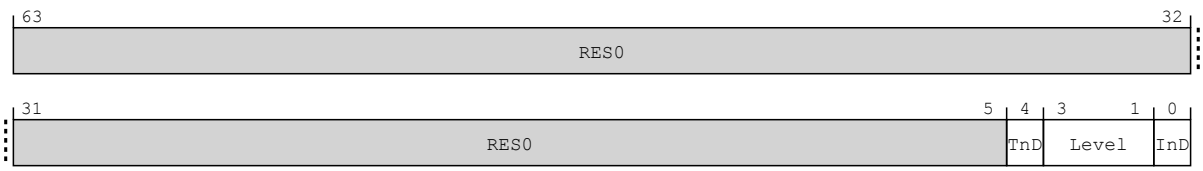

Bits [63:5]

Reserved, RES0.

TnD, bit [4]

When FEAT_MTE2 is implemented:

Allocation Tag not Data bit.

TnD	Meaning
0b0	Data, Instruction or Unified cache.
0b1	Separate Allocation Tag cache.

When CSSELR_EL1.InD == 1, this bit is RES0.

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR_EL1 is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Level, bits [3:1]

Cache level of required cache.

Level	Meaning
0b000	Level 1 cache.
0b001	Level 2 cache.
0b010	Level 3 cache.
0b011	Level 4 cache.
0b100	Level 5 cache.
0b101	Level 6 cache.
0b110	Level 7 cache.

All other values are reserved.

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR_EL1 is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

InD, bit [0]

Instruction not Data bit.

InD	Meaning
0b0	Data or unified cache.
0b1	Instruction cache.

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 is CONSTRAINED UNPREDICTABLE, and returns UNKNOWN values for CSSELR_EL1.{Level, InD}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CSSELR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CSSELR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b010	0b0000	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().CSSELR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CSSELR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CSSELR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CSSELR_EL1();
end;
```

MSR CSSELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b010	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().CSSELR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        CSSELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    CSSELR_EL1() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CSSELR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.41 CTR_EL0, Cache Type Register

The CTR_EL0 characteristics are:

Purpose

Provides information about the architecture of the caches.

Configuration

AArch64 System register CTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register CTR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CTR_EL0 are UNDEFINED.

Attributes

CTR_EL0 is a 64-bit register.

Field descriptions

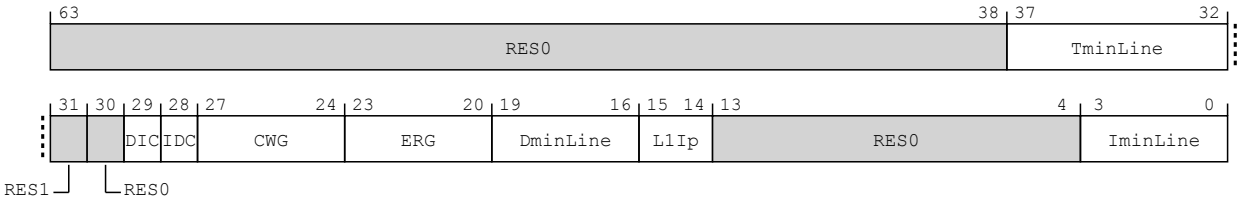

Bits [63:38]

Reserved, RES0.

TminLine, bits [37:32]

When FEAT_MTE2 is implemented:

Tag minimum Line. Log_2 of the number of words covered by Allocation Tags in the smallest cache line of all caches which can contain Allocation tags that are controlled by the PE.

Note

- For an implementation with cache lines containing 64 bytes of data and 4 Allocation Tags, this will be $log_2(64/4) = 4$.
- For an implementation with Allocations Tags in separate cache lines of 128 Allocation Tags per line, this will be $log_2(128*16/4) = 9$.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DIC, bit [29]

Instruction cache invalidation requirements for data to instruction coherence.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DIC	Meaning
0b0	Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.
0b1	Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.

All PEs in the same Inner Shareable shareability domain must have a common value of this field.

Note

If CTR_EL0.DIC is 1, instruction cache maintenance is not required after overwriting instructions, including for different VA aliases of the affected Locations, regardless of the value of CTR_EL0.L1Ip.

Access to this field is RO.

IDC, bit [28]

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

The value of this field is an IMPLEMENTATION DEFINED choice of:

IDC	Meaning
0b0	Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR_EL1.LoC == 0b000 or (CLIDR_EL1.LoUIS == 0b000 and CLIDR_EL1.LoUU == 0b000).
0b1	Data cache clean to the Point of Unification is not required for instruction to data coherence.

If CTR_EL0.DIC is 1 then the value reported in this field must also be 1.

The Effective value of IDC is 1 if any of the following are true:

- CTR_EL0.IDC == 1.
- CLIDR_EL1.LoC == 0b000.
- CLIDR_EL1.LoUIS == 0b000 and CLIDR_EL1.LoUU == 0b000.

All PEs in the same Inner Shareable shareability domain must have a common Effective value of IDC.

Access to this field is RO.

CWG, bits [27:24]

Cache writeback granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of 0b0000 indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm recommends that an implementation that does not support cache write-back implements this field as `0b0001`. This applies, for example, to an implementation that supports only write-through caches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

ERG, bits [23:20]

Exclusives reservation granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value `0b0000` is deprecated.

The value `0b0001` and values greater than `0b1001` are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Ip	Meaning
`0b00`	Reserved.
`0b01`	ASID-tagged Virtual Index, Virtual Tag (AIVIVT).
`0b10`	Virtual Index, Physical Tag (VIPT).
`0b11`	Physical Index, Physical Tag (PIPT).

From Armv8.0, the value `0b01` is reserved.

Access to this field is RO.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CTR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().UCT == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().CTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().UCT == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CTR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID2 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().CTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CTR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CTR_EL0();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CTR_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.42 DACR32_EL2, Domain Access Control Register

The DACR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 DACR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configuration

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

AArch64 System register DACR32_EL2 bits [31:0] are architecturally mapped to AArch32 System register DACR[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DACR32_EL2 are UNDEFINED.

Attributes

DACR32_EL2 is a 64-bit register.

Field descriptions

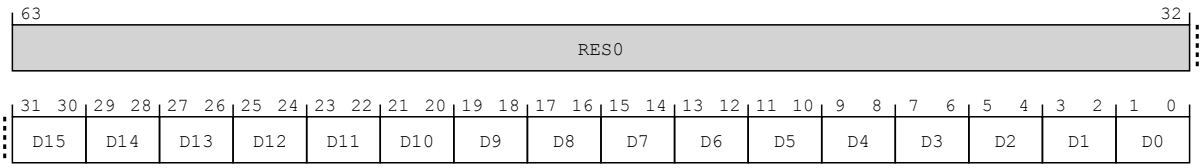

Bits [63:32]

Reserved, RES0.

D<n> , bits [2n+1:2n], for n = 15 to 0

Domain n access permission, where n = 0 to 15. Permitted values are:

D<n>	Meaning
0b00	No access. Any access to the domain generates a Domain fault.
0b01	Client. Accesses are checked against the permission bits in the translation tables.
0b11	Manager. Accesses are not checked against the permission bits in the translation tables.

The value 0b10 is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DACR32_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, DACR32_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0000	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = DACR32_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = DACR32_EL2();
end;
```

```
MSR DACR32_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0000	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    DACR32_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    DACR32_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.43 DCZID_EL0, Data Cache Zero ID Register

The DCZID_EL0 characteristics are:

Purpose

Indicates the block size that is written with byte values of 0 by the DC ZVA (Data Cache Zero by Address) System instruction.

If FEAT_MTE is implemented, this register indicates the granularity at which the DC GVA and DC GZVA instructions write.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DCZID_EL0 are UNDEFINED.

Attributes

DCZID_EL0 is a 64-bit register.

Field descriptions

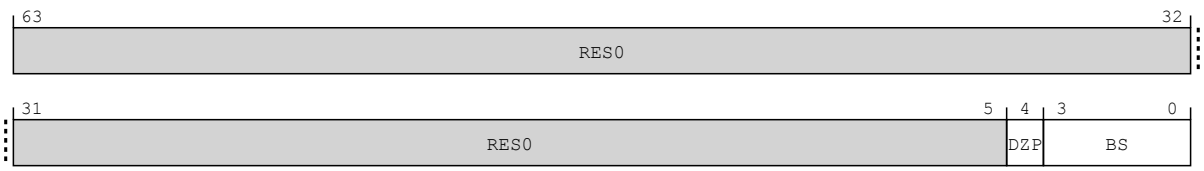

Bits [63:5]

Reserved, RES0.

DZP, bit [4]

Data Zero Prohibited. This field indicates whether use of DC ZVA instructions is permitted or prohibited.

If FEAT_MTE is implemented, this field indicates whether use of the DC GVA and DC GZVA instructions are permitted or prohibited.

DZP	Meaning
0b0	Instructions are permitted.
0b1	Instructions are prohibited.

The value read from this field is governed by the current Exception level and the values of the following fields:

- The Effective value of HCR_EL2.TDZ.
- When the Effective value of HCR_EL2.{E2H, TGE} != ‘11’, SCTLR_EL1.DZE.
- When the Effective value of HCR_EL2.{E2H, TGE} == ‘11’, SCTLR_EL2.DZE.

BS, bits [3:0]

Log_2 of the block size in words written by a DC ZVA instruction. The maximum size supported is 2KB, indicated by value 0b1001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_MTE2 is implemented, then this field determines the block size written by a DC GVA or DC GZVA instruction and the minimum size supported is 16 bytes, indicated by value `0b0010`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing DCZID_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DCZID_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0000	0b0000	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
        ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().DCZID_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = DCZID_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGRTR_EL2().DCZID_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = DCZID_EL0();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = DCZID_EL0();
elsif PSTATE.EL == EL3 then
    X{64}(t) = DCZID_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.44 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL1.

Configuration

AArch64 System register ESR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DFSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ESR_EL1 are UNDEFINED.

Attributes

ESR_EL1 is a 64-bit register.

Field descriptions

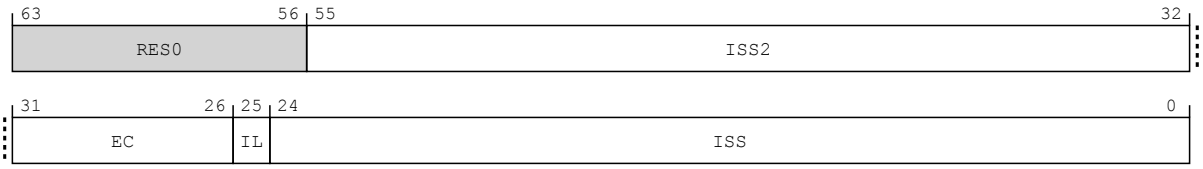

ESR_EL1 is made UNKNOWN as a result of an exception return from EL1.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL1, the value of ESR_EL1 is UNKNOWN. The value written to ESR_EL1 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:56]

Reserved, RES0.

ISS2, bits [55:32]

ISS2 encoding for an exception, the bit assignments are:

ISS2 encoding for an exception from a Data Abort

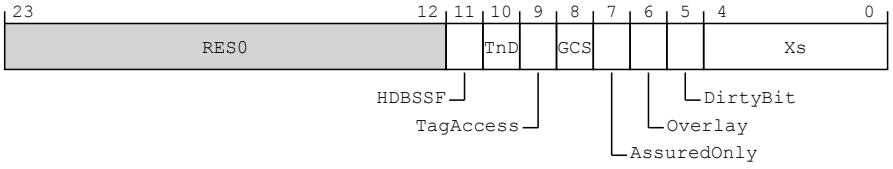

Bits [23:12]

Reserved, RES0.

HDBSSF, bit [11]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_HDBSS is implemented, FEAT_NV is implemented, IsSecondStage(Fault), and GetESR_ELx_DFSC(EL1) IN {'0011xx', '01001x', '0101xx', '10001x', '1001xx'}:

Indicates that the fault was caused by the HDBSS.

When DFSC indicates a stage 2 Permission fault, this field indicates whether that fault was caused because HDBSS cannot have data appended, because the HDBSS is either full or in an error state.

When DFSC indicates a synchronous External abort on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

When DFSC indicates a Granule Protection Fault on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

HDBSSF	Meaning
`0b0`	Fault was not caused by HDBSS.
`0b1`	Fault was caused by HDBSS.

This field only applies for stage 2 Permission faults.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TnD, bit [10]

When FEAT_MTE_CANONICAL_TAGS is implemented:

Tag not Data.

If a memory access generates a Data Abort for a stage 1 Permission fault, this field indicates whether the fault is due to an Allocation Tag access.

TnD	Meaning
`0b0`	Permission fault is not due to a write of an Allocation Tag to Canonically Tagged memory.
`0b1`	Permission fault is due to a write of an Allocation Tag to Canonically Tagged memory.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TagAccess, bit [9]

When FEAT_MTE_PERM is implemented and FEAT_NV is implemented:

NoTagAccess fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If a memory access generates a Data Abort for a Permission fault, this field indicates whether the fault is due to the NoTagAccess memory attribute.

TagAccess	Meaning
0b0	Permission fault is not due to the NoTagAccess memory attribute, and not due to an access using a Tag VA.
0b1	Permission fault is due to the NoTagAccess memory attribute.

For all other Data Aborts this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

GCS, bit [8]

When FEAT_GCS is implemented:

Guarded Control Stack data access.

If a memory access generates a Data Abort, this field indicates whether the fault is due to a Guarded Control Stack data access.

GCS	Meaning
0b0	The Data Abort is not due to a Guarded control stack data access.
0b1	The Data Abort is due to a Guarded control stack data access. The ISV field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented and FEAT_NV is implemented:

AssuredOnly flag.

If EL2 provides information regarding a stage 2 Data Abort to EL1, then this field holds information about the fault.

If this field is 1 and ESR_EL1.GCS is also 1, then the AssuredOnly check might have been the result of VTCR_EL2.GCSH configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AssuredOnly	Meaning
0b0	The Data Abort is not due to AssuredOnly.
0b1	The Data Abort is due to AssuredOnly.

For all other Data Aborts this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When FEAT_S1POE is implemented:

Overlay flag.

If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When FEAT_S1PIE is implemented:

DirtyBit flag.

If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

For any other fault or Access, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Xs, bits [4:0]

When FEAT_LS64 is implemented:

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort exception reported with ISV=1, then this field holds register specifier, Xs.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort exception reported with ISV=1, then this field holds register specifier, Xs.

Otherwise, this field is RES0.

Otherwise:

Reserved, RES0.

ISS2 encoding for an exception from an Instruction Abort

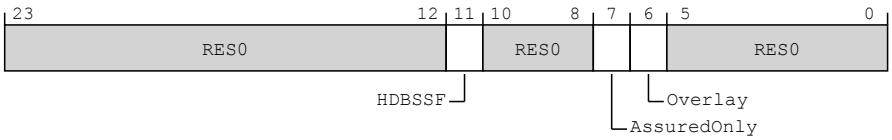

Bits [23:12]

Reserved, RES0.

HDBSSF, bit [11]

When FEAT_HDBSS is implemented and FEAT_NV is implemented:

Indicates that the fault was caused by the HDBSS.

When IFSC indicates a stage 2 Permission fault, this field indicates whether that fault was caused because HDBSS cannot have data appended, because the HDBSS is either full or in an error state.

When IFSC indicates a synchronous External abort on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

When IFSC indicates a Granule Protection Fault on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

HDBSSF	Meaning
0b0	Fault was not caused by HDBSS.
0b1	Fault was caused by HDBSS.

This field only applies for stage 2 Permission faults.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [10:8]

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented and FEAT_NV is implemented:

AssuredOnly flag.

If EL2 provides information regarding a stage 2 Instruction Abort to EL1, then this field holds information about the fault.

AssuredOnly	Meaning
`0b0`	The Instruction Abort is not due to AssuredOnly.
`0b1`	The Instruction Abort is due to AssuredOnly.

For all other Instruction Aborts this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When FEAT_S1POE is implemented:

Overlay flag.

If a memory access generates a Instruction Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
`0b0`	Instruction Abort is not due to Overlay Permissions.
`0b1`	Instruction Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS2 encoding for an exception from a Watchpoint exception

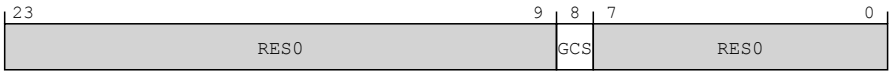

Bits [23:9]

Reserved, RES0.

GCS, bit [8]

When FEAT_GCS is implemented:

Guarded control stack data access.

Indicates that the Watchpoint exception is due to a Guarded control stack data access.

GCS	Meaning
0b0	The Watchpoint exception is not due to a Guarded control stack data access.
0b1	The Watchpoint exception is due to a Guarded control stack data access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [7:0]

Reserved, RES0.

ISS2 encoding for all other exceptions

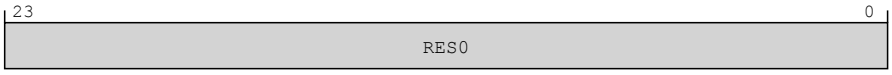

Bits [23:0]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b000000	Unknown reason.	ISS encoding for exceptions with an unknown reason	ISS2 encoding for all other exceptions	
0b000001	Trapped WF* instruction execution. Conditional WF* instructions that fail their condition code check do not cause an exception.	ISS encoding for an exception from a WF* instruction	ISS2 encoding for all other exceptions	
0b000011	Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC value 0b000000.	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b000100	Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC value 0b000000.	ISS encoding for an exception from an MCRR or MRRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b000101	Trapped MCR or MRC access with (coproc==0b1110).	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b000110	Trapped LDC or STC access. The only architected uses of these instruction are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS encoding for an exception from an LDC or STC instruction	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b000111	Access to SME, SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.	ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps	ISS2 encoding for all other exceptions	
0b001001	Trapped use of a Pointer authentication instruction.	ISS encoding for an exception from a trapped Pointer Authentication instruction	ISS2 encoding for all other exceptions	FEAT_PAuth is implemented
0b001010	Trapped execution of any instruction not covered by other EC values.	ISS encoding for an exception from any other instruction	ISS2 encoding for all other exceptions	FEAT_LS64 is implemented
0b001100	Trapped MRRC access with (coproc==0b1110).	ISS encoding for an exception from an MCRR or MRRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b001101	Branch Target Exception.	ISS encoding for a Branch Target Exception	ISS2 encoding for all other exceptions	FEAT_BTI is implemented
0b001110	Illegal Execution state.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b010001	SVC instruction execution in AArch32 state.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b010100	Trapped MSRR, MRRS or System instruction execution in AArch64 state, that is not reported using EC value 0b000000.	ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_SYSREG128 is implemented or FEAT_SYSINSTR128 is implemented
0b010101	SVC instruction execution in AArch64 state.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC values 0b000000, 0b000001, or 0b000111. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview', except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b011001	Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC value 0b000000.	ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ	ISS2 encoding for all other exceptions	FEAT_SVE is implemented
0b011010	Trapped ERET, ERETAA, or ERETAB instruction execution.	ISS encoding for an exception from an ERET, ERETAA, or ERETAB instruction	ISS2 encoding for all other exceptions	FEAT_FGT is implemented or FEAT_NV is implemented
0b011100	Exception from a PAC Fail	ISS encoding for a PAC Fail exception	ISS2 encoding for all other exceptions	FEAT_FPAC is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b011101	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, CPTR_EL3.ESM, or an attempted execution of an instruction that is illegal because of the value of PSTATE.SM or PSTATE.ZA, that is not reported using EC value 0b000000.	ISS encoding for an exception due to SME functionality	ISS2 encoding for all other exceptions	FEAT_SME is implemented
0b100000	Instruction Abort from a lower Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100001	Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100010	PC alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b100100	Data Abort exception from a lower Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100101	Data Abort exception taken without a change in Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100110	SP alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b100111	Memory Operation Exception.	ISS encoding for an exception from the Memory Copy and Memory Set instructions	ISS2 encoding for all other exceptions	FEAT_MOPS is implemented
0b101000	Trapped floating-point exception taken from AArch32 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b101100	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b101101	GCS exception.	ISS encoding for a GCS exception	ISS2 encoding for all other exceptions	FEAT_GCS is implemented
0b101111	SError exception.	ISS encoding for an SError exception	ISS2 encoding for all other exceptions	
0b110000	Breakpoint exception from a lower Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	
0b110001	Breakpoint exception taken without a change in Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	
0b110010	Software Step exception from a lower Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110011	Software Step exception taken without a change in Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110100	Watchpoint exception from a lower Exception level.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b110101	Watchpoint exception taken without a change in Exception level.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	
0b111000	BKPT instruction execution in AArch32 state.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b111100	BRK instruction execution in AArch64 state.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b111101	Profiling exception	ISS encoding for a Profiling exception	ISS2 encoding for all other exceptions	FEAT_EBEP is implemented, or FEAT_SPE_EXC is implemented, or FEAT_TRBE_EXC is implemented

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

IL	Meaning
0b0	16-bit instruction trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IL	Meaning

`0b1`	32-bit instruction trapped. This value is also used when the exception is one of the following:
	• An SError exception.
	• An Instruction Abort exception.
	• A PC alignment fault exception.
	• An SP alignment fault exception.
	• A Data Abort exception for which the value of the ISV bit is 0.
	• An Illegal Execution state exception.
	• Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
	  • ‘0’: 16-bit T32 BKPT instruction.
	  • ‘1’: 32-bit A32 BKPT instruction or A64 BRK instruction.
	• An exception reported using EC value `0b000000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see ‘Mapping of the general-purpose registers between the Execution states’.

If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value `0b11111`.

ISS encoding for exceptions with an unknown reason

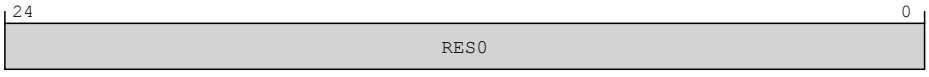

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for exceptions with an unknown reason

When an exception is reported using this EC value, the IL field is set to 1.

This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution of an MSR or MRS instruction using a _EL12 register name when the Effective value of HCR_EL2.E2H is not 1.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, and EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1. If the value of HCR_EL2.TGE is 0, this exception is reported using an ESR_EL1.EC value of `0b000111`.
- If FEAT_UINJ is implemented, an Undefined exception caused by attempting to execute any instruction when PSTATE.UINJ is 1.

ISS encoding for an exception from a WF* instruction

24	23 20	19 10	9 5	4 3	2	1 0
CV	COND	RES0	RN	RES0	RV	TI

CV, bit [24]

Condition code valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:10]

Reserved, RES0.

RN, bits [9:5]

When FEAT_WFxT is implemented:

Register Number. Indicates the register number supplied for a WFET or WFIT instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RV, bit [2]

When FEAT_WFxT is implemented:

Register field Valid.

If TI[1] == 1, then this field indicates whether RN holds a valid register number for the register argument to the trapped `WFET` or `WFIT` instruction.

RV	Meaning
`0b0`	Register field invalid.
`0b1`	Register field valid.

If TI[1] == 0, then this field is RES0.

This field is set to 1 on a trap on `WFET` or `WFIT`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

TI	Meaning	Applies when
`0b00`	WFI trapped.	
`0b01`	WFE trapped.	
`0b10`	WFIT trapped.	FEAT_WFxT is implemented
`0b11`	WFET trapped.	FEAT_WFxT is implemented

When FEAT_WFxT is implemented, this is a two bit field as shown. Otherwise, bit[1] is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a WF* instruction

The following fields describe configuration settings for generating this exception:

- HCR.{TWE, TWI}.
- SCTLR_EL1.{nTWE, nTWI}.
- SCTLR_EL2.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

ISS encoding for an exception from an MCR or MRC access

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

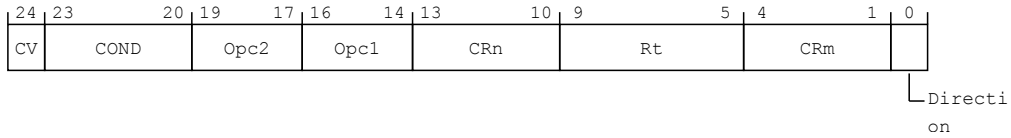

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value 0b000.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

If the Rt value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See ‘Mapping of the general-purpose registers between the Execution states’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write to System register space. MCR instruction.
`0b1`	Read from System register space. MRC or VMRS instruction.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an MCR or MRC access

The following fields describe configuration settings for generating exceptions from an MCR or MRC access using coproc `0b1111`, that are reported using EC value `0b000011`:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, trapped to EL1.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors System registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- HCR.{TRVM, TVM} and HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, trapped to EL2.
- HCR.TTLB and HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, trapped to EL2.
- HCR.{TSW, TPC, TPU} and HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, trapped to EL2.
- HCR.TAC and HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, trapped to EL2.
- HCR.TIDCP and HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, trapped to EL2.
- If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, trapped to EL2.
- HCR.{TID1, TID2, TID3} and HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to error record registers at EL1 using AArch32 state, trapped to EL2.
- HCPTR.TCPAC and CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, trapped to EL2.
- HSTR.T<n> and HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, trapped to EL2.
- CNTHCTL.PL1PCEN and CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HDCR.TTRF, for Non-secure accesses to trace filter control registers from system registers using AArch32 state, trapped to EL2.
- HDCR.{TPM, TPMCR} and MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HCPTR.TAM and CPTR_EL2.TAM, for accesses to Activity Monitors System registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions from an MCR or MRC access using coproc `0b1110`, that are reported using EC value `0b000101`:

- CPACR_EL1.TTA for accesses to trace System registers, trapped to EL2 or EL1.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, trapped to EL2 or EL1.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC System registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC System registers at EL0, EL1, and EL2 trapped to EL3.
- HCR.TID0 and HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, trapped to EL2.
- HCPTR.TTA and CPTR_EL2.TTA, for accesses to trace System registers using AArch32, trapped to EL2.
- HDCR.TDRA and MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DBGDSAR using AArch32, trapped to EL2.
- HDCR.TDOSA and MDCR_EL2.TDOSA, for accesses to powerdown debug System registers, using AArch32 state, trapped to EL2.
- HDCR.TDA and MDCR_EL2.TDA, for accesses to other debug System registers, using AArch32 state, trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace System registers using AArch32, trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug System registers using AArch32, trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug System registers, using AArch32, trapped to EL3.

The following fields describe configuration settings for generating exceptions from a VMSR or VMRS access, that are reported using EC value `0b001000`:

- HCR.TID0 and HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR.TID3 and HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.
- HCPTR.{TCP10, TCP11}, for Non-secure accesses to FPSCR, FPSID, FPEXC, MVFR0, MVFR1, and MVFR2, trapped to EL2.

ISS encoding for an exception from any other instruction

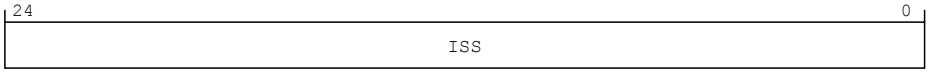

ISS, bits [24:0]

ISS	Meaning	Applies when
`0b0000000000000000000000000`	`ST64BV` instruction trapped.	FEAT_LS64_V is implemented
`0b0000000000000000000000001`	`ST64BV0` instruction trapped.	FEAT_LS64_ACCDATA is implemented
`0b0000000000000000000000010`	`LD64B` or `ST64B` instruction trapped.	FEAT_LS64 is implemented

All other values are reserved.

ISS encoding for an exception from an MCRR or MRRC access

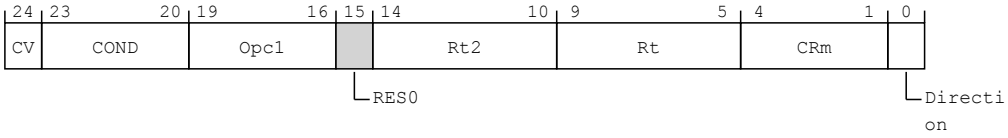

CV, bit [24]

Condition code valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

If the Rt2 value is not 0b1111, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt2 value is 0b1111:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b11111.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See 'Mapping of the general-purpose registers between the Execution states'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

If the Rt value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See 'Mapping of the general-purpose registers between the Execution states'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write to System register space. MCRR instruction.
`0b1`	Read from System register space. MRRC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an MCRR or MRRC access

The following fields describe configuration settings for generating exceptions from an MCRR or MRRC access using coproc `0b1111`, that are reported using EC value `0b000100`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, trapped to EL2 or EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors System registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, trapped to EL2 or EL1.
- HCR.{TRVM, TVM} and HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to error record registers at EL1 using AArch32 state, trapped to EL2.
- HSTR.T<n> and HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, trapped to EL2.
- CNTHCTL.{PL1PCEN, PL1PCTEN} and CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HDCR.TPM and MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HCPTR.TAM and CPTR_EL2.TAM, for accesses to Activity Monitors System registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions from an MCRR or MRRC access using coproc `0b1110`, that are reported using EC value `0b001100`:

- MDSCR_EL1.TDCC, for accesses to the Debug ROM registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, trapped to EL2 or EL1.
- HDCR.TDRA and MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, trapped to EL2.
- MDCR_EL3.TDA, for accesses to debug System registers, using AArch32, trapped to EL3.
- CPACR_EL1.TTA for accesses to trace System registers using AArch32, trapped to EL1 or EL2.
- HCPTR.TTA and CPTR_EL2.TTA, for accesses to trace System registers using AArch32, trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace System registers using AArch32, trapped to EL3.

Note

If FEAT_ETMv4 or FEAT_ETE are implemented, MCRR and MRRC accesses to trace System registers are UNDEFINED and the resulting exception is higher priority than an exception due to these traps.

ISS encoding for an exception from an LDC or STC instruction

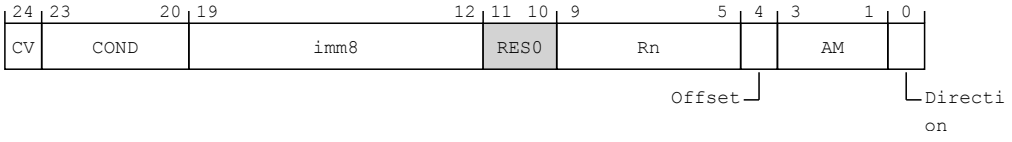

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer.

If the Rn value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rn value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See ‘Mapping of the general-purpose registers between the Execution states’.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Offset	Meaning
`0b0`	Subtract offset.
`0b1`	Add offset.

This bit corresponds to the U bit in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

AM	Meaning
`0b000`	Immediate unindexed.
`0b001`	Immediate post-indexed.
`0b010`	Immediate offset.
`0b011`	Immediate pre-indexed.
`0b100`	For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
`0b110`	For a trapped STC instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in ‘Reserved values in System and memory-mapped registers and translation table entries’.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Direction	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an LDC or STC instruction

The following fields describe the configuration settings from an LDC or STC access for the traps that are reported using EC value 0b000110:

- MDSCR_EL1.TDCC, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL2 or EL1.
- HDCR.TDA and MDCR_EL2.TDA, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL2.
- MDCR_EL3.TDA, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC System registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC System registers at EL0, EL1, and EL2 trapped to EL3.

ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

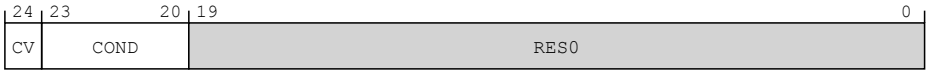

The accesses covered by this trap include:

- Execution of Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.
- Execution of SVE instructions.
- Execution of SME instructions.

For an implementation that does not include either SVE or support for Advanced SIMD and floating-point, the exception is reported using the EC value 0b000000.

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

The following fields describe the configuration settings for the traps that are reported using EC value `0b000111`:

- HCPTR.{TCP10, TCP11}, for Non-secure accesses to Advanced SIMD and floating-point registers and instructions, trapped to EL2.
- HCPTR.TASE, for Non-secure accesses to Advanced SIMD functionality, trapped to EL2.
- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL3.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

24 … 0
RES0

The accesses covered by this trap include:

- Execution of SVE instructions when the PE is not in Streaming SVE mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses to the SVE System registers, ZCR_ELx.

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

The following fields describe the configuration settings for the traps that are reported using EC value `0b011001`:

- CPACR_EL1.ZEN, for execution of SVE instructions and accesses to SVE registers at EL1 or EL0, trapped to EL1.
- CPTR_EL2.ZEN and CPTR_EL2.TZ, for execution of SVE instructions and accesses to SVE registers at EL0, EL1, or EL2, trapped to EL2.
- CPTR_EL3.EZ, for execution of SVE instructions and accesses to SVE registers from all Exception levels, trapped to EL3.

ISS encoding for a Profiling exception

Bits [24:6]

Reserved, RES0.

FSC, bits [5:1]

Indicates why the Profiling exception was generated.

FSC	Meaning	Applies when
`0b00000`	PMU Profiling exception. The exception was generated because at least one PMU counter overflow status flag was 1.	FEAT_EBEP is implemented
`0b00001`	Profiling Buffer management event. The exception was generated because PMBSR_EL1.S was 1.	FEAT_SPE_EXC is implemented
`0b00010`	Trace buffer management event. The exception was generated because TRBSR_EL1.IRQ was 1.	FEAT_TRBE_EXC is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

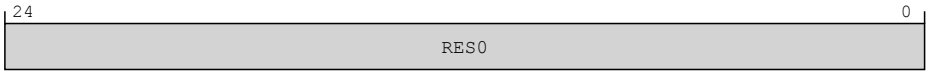

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see 'PC alignment checking'.

'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from the Memory Copy and Memory Set instructions

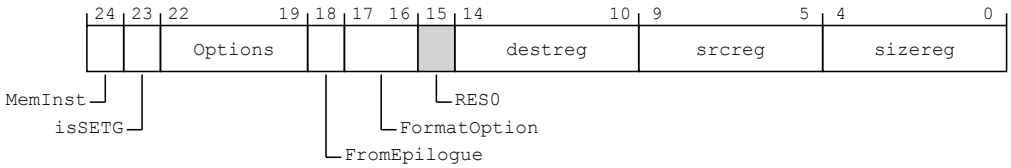

MemInst, bit [24]

Indicates the memory instruction class causing the exception.

MemInst	Meaning
0b0	CPYFE*, CPYFM*, CPYE*, and CPYM* instructions.
0b1	The instruction that caused the exception is in one of the following instruction classes: • SETE*, SETM*. • SETGE*, SETGM*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

isSETG, bit [23]

Indicates whether the instruction belongs to one of the following instruction classes:

- SETGE*, SETGM*.

isSETG	Meaning
0b0	The instruction that caused the exception is not in one of the specified classes.
0b1	The instruction that caused the exception is in one of the specified classes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Options, bits [22:19]

Options : the Options field of the instruction.

For Memory Copy instructions, bits[22:19] forms the Options field, which holds the bits[15:12] of the instruction.

For Memory Set instructions:

- Bits[22:21] are RES0.
- Bits[20:19] form the Options field, which holds the bits[13:12] of the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FromEpilogue, bit [18]

Indicates whether the instruction belongs to one of the following epilogue classes of Memory Copy or Memory Set instructions:

- CPYE*, CPYFE*.
- SETE*, SETGE*.

FromEpilogue	Meaning
`0b0`	The instruction that caused the exception is not in one of the specified classes.
`0b1`	The instruction that caused the exception is in one of the specified classes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FormatOption, bits [17:16]

Reports the Option used to encode the initial Xs, Xd, and Xn register values provided to the instruction that generated the exception.

FormatOption	Meaning
`0b00`	Option B.
`0b01`	Option A.
`0b10`	Option A.
`0b11`	Option B.

For more information, see Memory Copy and Memory Set exceptions.

Note

This field was previously presented as two separate bits, WrongOption, bit[17] and OptionA, bit [16], which were already expected to be used together and not individually.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

destreg, bits [14:10]

The destination register value from the issued instruction, containing the destination address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

srcreg, bits [9:5]

The source register value from the issued instruction, containing either the source address or the source data.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

sizereg, bits [4:0]

The size register value from the issued instruction, containing the number of bytes to be transferred or set.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from HVC or SVC instruction execution

24 … 16	15 … 0
RES0	imm16

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from HVC or SVC instruction execution

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see ‘SVC’ and ‘HVC’.

For A64 instructions, see ‘SVC’ and ‘HVC’.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

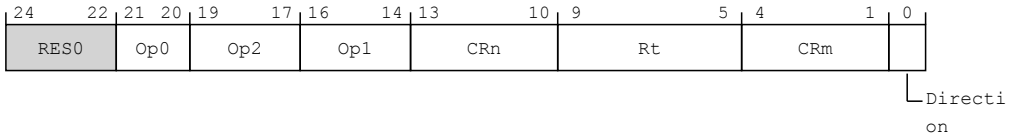

Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

For system instructions which require that the opcode Rt field is set to `0b11111`, but where the trapped instruction has a different value of Rt, an implementation is permitted to return the value `0b11111`, instead of the value of Rt from the trapped instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write access, including MSR instructions.
0b1	Read access, including MRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

For exceptions caused by System instructions, see 'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL1.
- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2 or EL1.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2 or EL1.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- CPACR_EL1.TTA, for accesses to the trace System registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- PMUSERENR_EL0, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors System registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, execution is trapped to EL2.
- If FEAT_EVT is implemented, the following registers control traps for EL1 and EL0 registers and instructions that use this EC value:
  - HCR_EL2.{TICAB, TOCU, TID4}.
  - HCR2.{TICAB, TOCU, TID4}.
- If FEAT_EVT2 is implemented, the following registers control traps for EL1 and EL0 instructions that use this EC value:
  - HCR_EL2.{TTLBOS, TTLBIS}.
- If FEAT_EVT2 is implemented, HCR2.TTLBIS controls traps for EL1 and EL0 instructions using AArch32 state that use this EC value.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- If FEAT_MTE2 is implemented, HCR_EL2.TID5, for accesses to GMID_EL1, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_IDTE3 is implemented, SCR_EL3.TID3, for accesses to ID group 3 registers using AArch64 state, at EL1 and EL2, trapped to EL3.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace System registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug System registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug System registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors System registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the trace filter control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- If FEAT_SPE is implemented:
  - MDCR_EL3.NSPB for accesses to Statistical Profiling and Profiling Buffer control registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
  - MDCR_EL2.TPMS for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 trapped to EL2.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If FEAT_FGT is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR_EL2 for execution of system instructions trapped to EL2.
  - HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  - HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.
- If FEAT_RNG_TRAP is implemented, SCR_EL3.TRNDR for reads of RNDR and RNDRRS using AArch64 state, MRS access trapped to EL3.
- If FEAT_SME is implemented:
  - CPTR_EL3.ESM, for MSR or MRS accesses to SMPRI_EL1 at EL1, EL2, and EL3, trapped to EL3.
  - CPTR_EL3.ESM, for MSR or MRS accesses to SMPRIMAP_EL2 at EL2 and EL3, trapped to EL3.
  - SCTLR_EL1.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2 or EL1.
  - SCTLR_EL2.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2.
  - SCR_EL3.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, EL1, and EL2, trapped to EL3.
- If FEAT_FPMR is implemented:
  - SCTLR_EL1.EnFPM, for accesses to FPMR at EL0, trapped to EL2 or EL1.
  - SCTLR_EL2.EnFPM, for accesses to FPMR at EL0, trapped to EL2.
  - HCRX_EL2.EnFPM, for accesses to FPMR at EL0 and EL1, trapped to EL2.
  - SCR_EL3.EnFPM, for accesses to FPMR at EL0, EL1, and EL2, trapped to EL3.
- If FEAT_NMI is implemented, HCRX_EL2.TALLINT, for MSR writes of ALLINT at EL1, trapped to EL2.
- If FEAT_FGT2 is implemented:
  - SCR_EL3.FGTEn2, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR2_EL2 for reads and HFGWTR2_EL2 for writes of registers, using AArch64 state, using MSR or MRS access at EL1 trapped to EL2.
  - HDFGRTR2_EL2 for reads and HDFGWTR2_EL2 for writes of registers, using AArch64 state, using MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR2_EL2 for execution of system instructions trapped to EL2.
- If FEAT_ITE is implemented, MDCR_EL3.EnITE, for accesses to Instrumentation trace System registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If FEAT_MEC is implemented, SCR_EL3.MECEn, for accesses to MECID registers at EL2, trapped to EL3.
- If FEAT_SPE_FDS is implemented, MDCR_EL3.EnPMS3 for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
- If FEAT_SPE_nVM is implemented, MDCR_EL3.EnPMS4 for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
- If FEAT_RASv2 is implemented, SCR_EL3.TWERR, for accesses to Error Record registers, MSR access at EL1 and EL2 trapped to EL3.
- If FEAT_Debugv8p9 is implemented, MDCR_EL3.EBWE for accesses of MDSELR_EL1, using AArch64 state, MRS or MSR access at EL2 and EL1 trapped to EL3.
- If FEAT_PMUv3p9, FEAT_SPMU, FEAT_EBEP, or FEAT_PMUv3_SS is implemented, MDCR_EL3.EnPM2, for accesses to PMU registers, using AArch64 state, MSR or MRS access at EL2, EL1, and EL0, trapped to EL3.
- If FEAT_PMUv3_SS is implemented, MDCR_EL3.EnPMSS, for accesses to PMU Snapshot registers, using AArch64 state, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_THE is implemented, SCR_EL3.RCWMASKEn for accesses to RCWMASK_EL1 and RCWSMASK_EL1, using AArch64 state, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_AIE is implemented, SCR_EL3.AIEn for accesses to Extended Memory Attribute registers, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_S1PIE, FEAT_S2PIE, FEAT_S1POE, or FEAT_S2POE is implemented, SCR_EL3.PIEn for accesses to Permission Indirection, Overlay registers, MSR or MRS access at EL2, EL1 and EL0 trapped to EL3.
- If FEAT_MPAM_PE_BW_CTRL is implemented, MPAMBW2_EL2.{nTRAP_MPAMBWIDR_EL1, nTRAP_MPAMBW0_EL1, nTRAP_MPAMBW1_EL1} for accesses to MPAMBWn_EL1 registers, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_MPAM_PE_BW_CTRL and FEAT_SME are implemented, MPAMBW2_EL2.nTRAP_MPAMBWSM_EL1, for accesses to MPAMBWSM_EL1, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_MPAM_PE_BW_CTRL is implemented, MPAMBW3_EL3.nTRAPLOWER, for accesses to MPAMBW2_EL2 and MPAMBWn_EL1 registers, using AArch64 state, MRS or MSR access at EL1, trapped to EL3.
- If FEAT_HACDBS is implemented, SCR_EL3.HACDBSEn, for accesses to HACDBSBR_EL2 and HACDBSCONS_EL2, using AArch64 state, MRS or MSR access at EL2, trapped to EL3.
- If FEAT_HDBSS is implemented, SCR_EL3.HDBSSEn, for accesses to HDBSSBR_EL2 and HDBSSPROD_EL2, using AArch64 state, MRS or MSR access at EL2, trapped to EL3.
- If FEAT_SRMASK is implemented, SCR_EL3.SRMASKEn, for MSR or MRS accesses at EL2 or EL1 to the MASK registers using AArch64 state, trapped to EL3.

ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

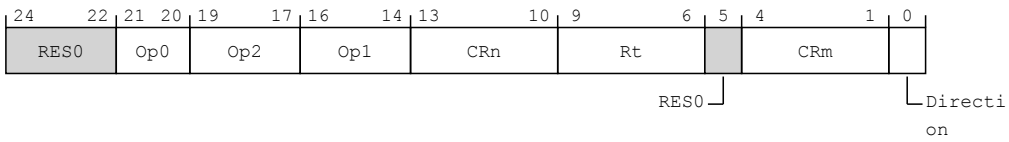

Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:6]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

Note

This value represents register pair of X[Rt:0], X[Rt:1].

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write access, MSRR instructions.
0b1	Read access, MRRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

The following fields describe configuration settings for generating exceptions from an MSRR or MRRS access that are reported using EC value 0b010100:

- If FEAT_FGT is implemented:
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, accesses at EL1 trapped to EL2.
- If FEAT_FGT2 is implemented:
  - HFGRTR2_EL2.nRCWSMASK_EL1 for reads and HFGWTR2_EL2.nRCWSMASK_EL1 for writes of RCWSMASK_EL1, using AArch64 state, accesses at EL1 trapped to EL2.
- If FEAT_SYSREG128 is implemented:
  - SCTLR2_EL1.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL0 trapped to EL1.
  - SCTLR2_EL2.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL0 trapped to EL2.
  - HCRX_EL2.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL1 and EL0 trapped to EL2.
  - SCR_EL3.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL2, EL1, and EL0 trapped to EL3.
- If FEAT_D128 is implemented:
  - HCR_EL2.{TRVM, TVM} for accesses to TTBR0_EL1 and TTBR1_EL1, accesses at EL1 and EL0 trapped to EL2.
  - HCRX_EL2.D128En for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL1 trapped to EL2.
  - HFGITR_EL2 for execution of TLBIP system instructions trapped to EL2.
  - HCR_EL2.{TTLB, TTLBOS, TTLBIS}, for execution of TLBIP instructions using AArch64 state, execution is trapped to EL2.
  - SCR_EL3.D128En for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL2 and EL1 trapped to EL3.
- If FEAT_THE is implemented, SCR_EL3.RCWMASKEn for accesses to RCWMASK_EL1 and RCWSMASK_EL1, using AArch64 state, accesses at EL2 and EL1 trapped to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for an exception from an Instruction Abort

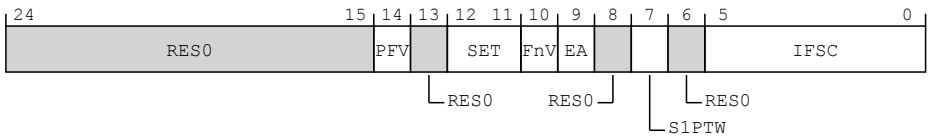

The ISS2 encoding for an exception from an Instruction Abort includes further information about the exception if any of the following are true:

- FEAT_S1POE is implemented and a memory access generates a Instruction Abort due to a Permission fault.
- FEAT_HDBSS is implemented.
- FEAT_THE is implemented.

Bits [24:15]

Reserved, RES0.

PFV, bit [14]

When FEAT_PFAR is implemented and GetESR_ELx_IFSC(EL1) IN {'010000', '01001x', '0101xx'}:

PFAR Valid. Describes whether the PFAR_EL1 register is valid.

PFV	Meaning
0b0	PFAR_EL1 is UNKNOWN.
0b1	PFAR_EL1 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [13]

Reserved, RES0.

SET, bits [12:11]

When FEAT_RAS is implemented and GetESR_ELx_IFSC(EL1) IN {'010000', '01001x', '0101xx'}:

Synchronous Error Type. Describes the PE error state after taking the Instruction Abort exception.

SET	Meaning	Applies when
0b00	Recoverable state (UER).	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET	Meaning	Applies when
0b10	Uncontainable (UC).	FEAT_RASv2 is not implemented
0b11	Restartable state (UEO).	

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception due to SME functionality

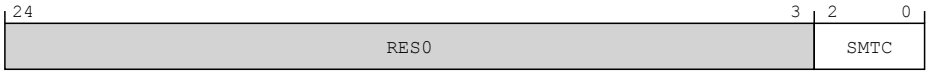

The accesses covered by this trap include:

- Execution of SME instructions.
- Execution of SVE and Advanced SIMD instructions, when the PE is in Streaming SVE mode.
- Direct accesses of SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3.

Bits [24:3]

Reserved, RES0.

SMTC, bits [2:0]

SME Trap Code. Identifies the reason for instruction trapping.

SMTC	Meaning	Applies when
0b000	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC value 0b000000.	
0b001	Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1.	
0b010	SME instruction trapped because PSTATE.SM is 0.	
0b011	SME instruction trapped because PSTATE.ZA is 0.	
0b100	Access to the SME2 ZT0 register trapped as a result of SMCR_EL1.EZT0, SMCR_EL2.EZT0, or SMCR_EL3.EZT0.	FEAT_SME2 is implemented

All other values are reserved.

Additional Information for ISS encoding for an exception due to SME functionality

The following fields describe the configuration settings for the traps that are reported using the EC value 0b011101:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CPACR_EL1.SMEN, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR and SMCR_EL1 System registers at EL1 and EL0, trapped to EL2 or EL1.
- CPTR_EL2.SMEN and CPTR_EL2.TSM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2 at EL2, EL1, and EL0, trapped to EL2.
- CPTR_EL3.ESM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3 from all Exception levels and any Security state, trapped to EL3.
- If FEAT_SME2 is implemented:
  - SMCR_EL1.EZT0, for accesses to ZT0 at EL1 and EL0, trapped to EL2 or EL1.
  - SMCR_EL2.EZT0, for accesses to ZT0 at EL2, EL1, and EL0, trapped to EL2.
  - SMCR_EL3.EZT0, for accesses to ZT0 at any Exception level, trapped to EL3.

ISS encoding for an exception from a Data Abort

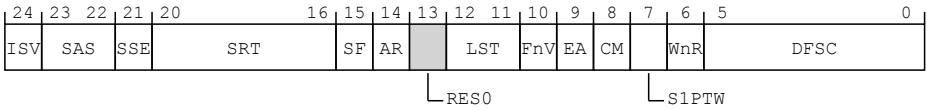

The ISS2 encoding for an exception from a Data Abort includes further information about the exception when any of the following features are implemented:

- FEAT_LS64_V.
- FEAT_LS64_ACCDATA.
- FEAT_S1POE or FEAT_S2POE.
- FEAT_S1PIE or FEAT_S2PIE.
- FEAT_GCS.
- FEAT_MTE_CANONICAL_TAGS.
- FEAT_HDBSS.
- FEAT_MTE_PERM.

ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] represents the instruction syndrome fields {SAS, SSE, SRT, SF, AR}.

ISV	Meaning
0b0	No valid instruction syndrome. ISS[23:14] do not represent an instruction syndrome.
0b1	ISS[23:14] hold a valid instruction syndrome.

The ISV field is only relevant for the following instructions:

- An AArch64 load or store of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).
- An AArch32 instruction LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT, where both of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - It is not performing register writeback.
  - It is not using R15 as a source or destination register.
- If FEAT_LS64 is implemented, an LD64B or ST64B instruction.
- If FEAT_LS64_V is implemented, an ST64BV instruction.
- If FEAT_LS64_ACCDATA is implemented, an ST64BV0 instruction.

The ISV field is 1 if all of the following apply:

- The fault was not a stage 2 fault on a stage 1 translation table walk.
- If FEAT_RAS is implemented, the fault was not any kind of synchronous External abort.
- Any of the following apply:
  - The fault was generated by one of the described AArch64 load or store instructions that transfer a single general-purpose register, and any of the following apply:
    - The fault was a stage 2 fault.
  - The fault was generated by one of the described AArch32 instructions and any of the following apply:
    - The fault was a stage 2 fault.
  - The fault was generated by one of the FEAT_LS64, FEAT_LS64_V or FEAT_LS64_ACCDATA instructions and the fault was a Translation fault, Access flag fault or Permission fault.
- One of the following applies:
  - The address reported in FAR_EL1 is the lowest address accessed by the instruction.
  - The address reported in FAR_EL1 is not the lowest address accessed by the instruction, and the implementation reports ISV as 1 in this case. Arm recommends against this behavior.

Otherwise the value of this field is 0.

Note

The value of this field is 0 if any of the following apply:

- FEAT_MOPS is implemented and the fault was for a memory access generated by a Memory Copy and Memory Set instruction.
- FEAT_MTE is implemented and the fault was for a direct access to Allocation Tags.
- FEAT_GCS is implemented and the fault was for a Guarded control stack data access.
- FEAT_NV2 is implemented and the fault was for a system register access that was transformed to a load or store relative to VNCR_EL2.
- FEAT_LSE is implemented and the fault was for an atomic instruction.

The value of this field is UNKNOWN when the following occurs:

- Stage 2 faults when the exception was generated in Debug state, in memory access mode.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

When ISV == '1':

Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

SAS	Meaning
0b00	Byte
0b01	Halfword
0b10	Word
0b11	Doubleword

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSE, bit [21]

When ISV == '1':

Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

SSE	Meaning
`0b0`	Sign-extension not required.
`0b1`	Data item must be sign-extended.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [20:16]

When ISV == '1':

SRT, bits [20:16]

Syndrome Register Transfer. The register number of the Wt/Xt/Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32, then this is the AArch64 view of the register. See ‘Mapping of the general-purpose registers between the Execution states’.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When ISV == '0', FEAT_RASv2 is implemented, and GetESR_ELx_DFSC(EL1) IN {'010000', '01001x', '0101xx'}:

WU, bits [1:0] of bits [20:16]

Write Update. Describes whether a store instruction that generated an External abort updated the location.

WU	Meaning
0b00	Not a store instruction or translation table update, or the location might have been updated.
0b10	Store instruction or translation table update that did not update the location.
0b11	Store instruction or translation table update that updated the location.

In the description of this field, a store instruction is any memory-writing instruction that explicitly performs a store. This includes instructions that both read and write memory.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

When ISV == '1':

SF, bit [15]

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

SF	Meaning
0b0	Instruction loads/stores a 32-bit general-purpose register.
0b1	Instruction loads/stores a 64-bit general-purpose register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

FnP, bit [15]

FAR not Precise.

FnP	Meaning	Applies when
0b0	The FAR holds the faulting virtual address that generated the Data Abort.	
0b1	The FAR holds any virtual address within the naturally-aligned granule that contains the faulting virtual address that generated a Data Abort due to an SVE contiguous vector load/store instruction, or an SME load/store instruction. For more information about the naturally-aligned fault granule, see FAR_ELx (for example, FAR_EL1).	FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [14]

When ISV == '1':

AR, bit [14]

Acquire/Release.

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register being ZR, it is IMPLEMENTATION SPECIFIC whether this field is reported as 0 or 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_PFAR is implemented and GetESR_ELx_DFSC(EL1) IN {'010000', '01001x', '0101xx'}:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PFV, bit [14]

PFAR Valid. Describes whether the PFAR_EL1 register is valid.

PFV	Meaning
0b0	PFAR_EL1 is UNKNOWN.
0b1	PFAR_EL1 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [13]

Reserved, RES0.

Bits [12:11]

When GetESR_ELx_DFSC(EL1) IN {'00xxxx', '10101x', '0000xx'}:

LST, bits [12:11]

Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort reported with ISV=1.

LST	Meaning	Applies when
0b00	The instruction that generated the Data Abort is not specified by this field.	
0b01	An ST64BV instruction generated the Data Abort.	FEAT_LS64_V is implemented
0b10	An LD64B or ST64B instruction generated the Data Abort.	FEAT_LS64 is implemented
0b11	An ST64BV0 instruction generated the Data Abort.	FEAT_LS64_ACCDATA is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL1) == '010000':

SET, bits [12:11]

Synchronous Error Type. Used when a synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET	Meaning	Applies when
0b00	Recoverable state (UER).	
0b10	Uncontainable (UC).	FEAT_RASv2 is not implemented
0b11	Restartable state (UEO).	

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

CM	Meaning
0b0	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CM	Meaning
0b1	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Abort caused by an instruction reading from a memory location.
0b1	Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- If FEAT_RASv2 is implemented, an External abort on an Atomic access, reported with ESR_EL1.WU set to 0b00.
- A fault reported using a DFSC value of 0b110101, indicating an unsupported Exclusive or atomic access.
- A fault reported using a DFSC value of 0b110001, indicating an unsupported atomic hardware update fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Synchronous Tag Check Fault.	FEAT_MTE2 is implemented
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).	

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a trapped floating-point exception

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

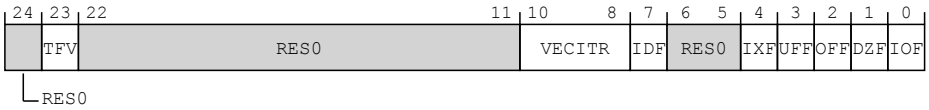

Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

TFV	Meaning
0b0	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
0b1	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see 'Floating-point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDF	Meaning
0b0	Input denormal floating-point exception has not occurred.
0b1	Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IXF	Meaning
0b0	Inexact floating-point exception has not occurred.
0b1	Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

UFF	Meaning
0b0	Underflow floating-point exception has not occurred.
0b1	Underflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

OFF	Meaning
0b0	Overflow floating-point exception has not occurred.
0b1	Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

DZF	Meaning
0b0	Divide by Zero floating-point exception has not occurred.
0b1	Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IOF	Meaning
0b0	Invalid Operation floating-point exception has not occurred.
0b1	Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a trapped floating-point exception

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

ISS encoding for a GCS exception

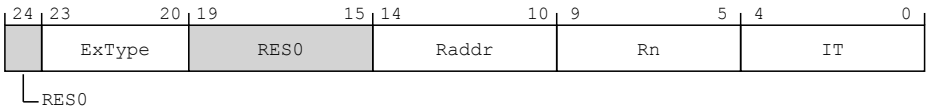

Bit [24]

Reserved, RES0.

ExType, bits [23:20]

The first level classification of GCS exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ExType	Meaning
0b0000	The exception reported is a Guarded Control Stack Data Check Exception.
0b0001	The exception reported is an EXLOCK Exception.
0b0010	The exception reported is a trap exception on GCSSTR or GCSSTTR instruction execution.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:15]

Reserved, RES0.

Raddr, bits [14:10]

When GetESR_ELx_ExType(EL1) == '0010':

Indicates the data address register number supplied in the instruction that has been trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [9:5]

When GetESR_ELx_ExType(EL1) == '0000':

Rn, bits [9:5]

Indicates a register number used by the instruction that caused the Guarded Control Stack Data Check Exception.

For a procedure return instruction reported with ESR_EL1.ISS.IT as 0b00000, contains the register number for the register which contains the target address of the branch.

For a GCSPOPM instruction reported with ESR_EL1.ISS.IT as 0b00001, contains the register number for the register which is the destination register of the instruction.

For a procedure return instruction reported with ESR_EL1.ISS.IT as 0b00010 or 0b00011, contains the value 0b11110, indicating X30.

For a GCSSS1 instruction reported with ESR_EL1.ISS.IT as 0b00100, contains the register number for the register which is the input register of the instruction.

If ESR_EL1.ISS.IT is reported as 0b00101 or 0b01000, this field is UNKNOWN

If ESR_EL1.ISS.IT is reported as 0b01001, this field is 0b11111

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When GetESR_ELx_ExType(EL1) == '0010':

Rvalue, bits [9:5]

Indicates the data value register number supplied in the instruction that has been trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IT, bits [4:0]

When GetESR_ELx_ExType(EL1) == '0000':

Type of the instruction that caused the Guarded Control Stack Data Check Exception.

IT	Meaning
`0b00000`	Guarded Control Stack Data Check Exception is from a procedure return instruction without Pointer authentication.
`0b00001`	Guarded Control Stack Data Check Exception is from a GCSPOPM instruction.
`0b00010`	Guarded Control Stack Data Check Exception is from a procedure return instruction with Pointer authentication that uses key A.
`0b00011`	Guarded Control Stack Data Check Exception is from a procedure return instruction with Pointer authentication that uses key B.
`0b00100`	Guarded Control Stack Data Check Exception is from a GCSSS1 instruction.
`0b00101`	Guarded Control Stack Data Check Exception is from a GCSSS2 instruction.
`0b01000`	Guarded Control Stack Data Check Exception is from a GCSPOPCX instruction.
`0b01001`	Guarded Control Stack Data Check Exception is from a GCSPOPX instruction.

All other values are reserved

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Additional Information for ISS encoding for a GCS exception

The following fields describe the configuration settings for the traps that are reported using EC value `0b101101` and ExType value `0b0010`:

- GCSCRE0_EL1.STREn
- GCSCR_EL1.STREn.
- GCSCR_EL2.STREn.
- GCSCR_EL3.STREn.
- HFGITR_EL2.nGCSSTR_EL1.

ISS encoding for an SError exception

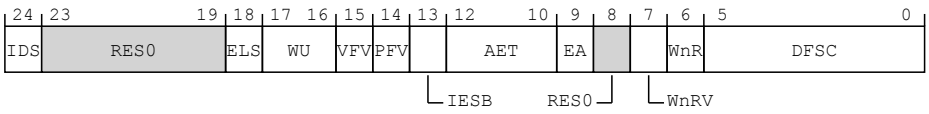

Note

In earlier versions of the architecture, an SError exception is referred to as an SError interrupt or an asynchronous External abort exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDS, bit [24]

IMPLEMENTATION DEFINED syndrome.

IDS	Meaning
0b0	Bits [23:0] of the ISS field holds the fields described in this encoding. **Note** If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.
0b1	Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError exception.

Note

This field was previously called ISV.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:19]

Reserved, RES0.

ELS, bit [18]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL1) == '010001':

Meaning of ELR_ELx.

ELS	Meaning
0b0	Asynchronous. Does not indicate the trigger for the exception.
0b1	Synchronous. The exception was triggered by the instruction at ELR_ELx.

SError exceptions that report this field is 1 are not required to be precise.

The ESR_EL1.AET field describes whether the exception is precise or imprecise.

Corrected, Recoverable or Restartable exceptions are precise. Unrecoverable or Uncontainable exceptions are imprecise.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WU, bits [17:16]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL1) == '010001':

Write Update. Describes whether a store instruction that generated an External abort updated the location.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WU	Meaning
0b00	Not a store instruction or translation table update, or the location might have been updated.
0b10	Store instruction or translation table update that did not update the location.
0b11	Store instruction or translation table update that updated the location.

In the description of this field, a store instruction is any memory-writing instruction that explicitly performs a store. This includes instructions that both read and write memory.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VFV, bit [15]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL1) == '010001':

FAR Valid. Indicates the FAR_EL1 register contains a valid virtual address.

VFV	Meaning
0b0	FAR_EL1 is not valid, and holds an UNKNOWN value.
0b1	FAR_EL1 contains a valid virtual address associated with the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PFV, bit [14]

When FEAT_PFAR is implemented and GetESR_ELx_DFSC(EL1) == '010001':

PFAR Valid. Describes whether the PFAR_EL1 register is valid.

PFV	Meaning
0b0	PFAR_EL1 is UNKNOWN.
0b1	PFAR_EL1 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IESB, bit [13]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_IESB is implemented and GetESR_ELx_DFSC(EL1) == '010001':

Implicit error synchronization event.

IESB	Meaning
0b0	The SError exception was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1	The SError exception was synchronized by the implicit error synchronization event and taken immediately.

This field is set to 1 when the SError exception was synchronized by an implicit error synchronization event and taken immediately on exception entry.

It this field is set to an IMPLEMENTATION SPECIFIC choice of 0 or 1 when the SError exception was synchronized by an implicit error synchronization event and taken immediately on an exception return.

For all other SError exceptions, this field is set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AET, bits [12:10]

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL1) == '010001':

Asynchronous Error Type.

Describes the PE error state after taking the SError exception.

AET	Meaning
0b000	Uncontainable (UC).
0b001	Unrecoverable state (UEU).
0b010	Restartable state (UEO).
0b011	Recoverable state (UER).
0b110	Corrected (CE).

All other values are reserved.

If multiple errors are taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EA, bit [9]

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL1) == '010001':

External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

WnRV, bit [7]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL1) == '010001':

ESR_EL1.WnR valid.

WnRV	Meaning
`0b0`	ESR_EL1.WnR is not valid and has been set to 0.
`0b1`	ESR_EL1.WnR is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WnR, bit [6]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL1) == '010001':

Write-not-Read. When the WnRV field is 1, indicates whether an exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
`0b0`	Exception was caused by an instruction reading from a memory location.
`0b1`	Exception was caused by an instruction writing to a memory location.

Accessing this bit has the following behavior:

- This bit is RES0 if ESR_EL1.WnRV is 0.
- This bit is not valid and reads UNKNOWN if an External abort on a Atomic access, reported with ESR_EL1.WU == `0b00`.
- Otherwise RW.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

DFSC, bits [5:0]

When FEAT_RAS is implemented:

Data Fault Status Code.

DFSC	Meaning
0b000000	Uncategorized error.
0b010001	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

24 … 6	5 … 0
RES0	IFSC

Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

For more information about generating these exceptions:

- For exceptions from AArch64, see ‘Breakpoint exceptions’.
- For exceptions from AArch32, see ‘Breakpoint exceptions’ and ‘Vector Catch exceptions’.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for an exception from a Software Step exception

24	23 … 7	6	5 … 0
ISV	RES0	EX	IFSC

ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

ISV	Meaning
0b0	EX bit is RES0.
0b1	EX bit is valid.

See the EX bit description for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

EX	Meaning
0b0	An instruction other than a Load-Exclusive instruction was stepped.
0b1	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Software Step exception

For more information about generating these exceptions, see 'Software Step exceptions'.

ISS encoding for an exception from a Watchpoint exception

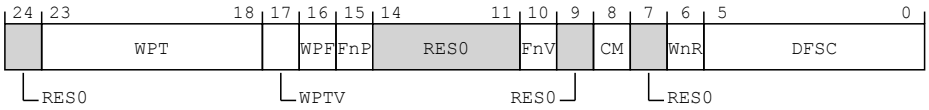

Bit [24]

Reserved, RES0.

WPT, bits [23:18]

When FEAT_Debugv8p2 is implemented:

Watchpoint number.

All other values are reserved.

Otherwise:

Reserved, RES0.

WPTV, bit [17]

When FEAT_Debugv8p2 is implemented:

Watchpoint number Valid.

WPTV	Meaning
0b0	The WPT field is invalid, and holds an UNKNOWN value.
0b1	The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint exception.

If FEAT_Debugv8p9 is implemented, value 0 is not permitted.

When a Watchpoint exception is triggered by a watchpoint match:

- If FEAT_Debugv8p9 is implemented or the PE sets any of FnV, FnP, or WPF to 1, then the PE sets WPTV to 1.
- Otherwise, the PE sets WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

Otherwise:

Reserved, RES0.

WPF, bit [16]

Watchpoint might be false-positive.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WPF	Meaning	Applies when
0b0	The watchpoint matched an address or address range that was accessed by the instruction.	
0b1	The watchpoint matched an address or address range that might not have been accessed by the instruction.	FEAT_SVE is implemented or FEAT_SME is implemented

Arm strongly recommends that this bit is set to 0, other than when one of the following instructions might generate a watchpoint match for an address or address range that the instruction does not access:

- An SVE contiguous vector load/store instruction, when the PE is in Streaming SVE mode.
- An SME load/store instruction.

FnP, bit [15]

FAR not Precise.

This field only has meaning if the FAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0.

FnP	Meaning	Applies when
0b0	If the FnV field is 0, the FAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.	
0b1	The FAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.	FEAT_SVE is implemented or FEAT_SME is implemented

Bits [14:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid.

FnV	Meaning	Applies when
0b0	The FAR is valid, and its value is as described by the FnP field.	
0b1	The FAR is invalid, and holds an UNKNOWN value.	FEAT_SVE is implemented or FEAT_SME is implemented

Bit [9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance instruction:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CM	Meaning
0b0	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Watchpoint exception was generated by the execution of a cache maintenance instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as a cache maintenance instructions, and therefore their execution does not cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Watchpoint exception caused by an instruction reading from a memory location.
0b1	Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Watchpoint exception

For more information about generating these exceptions, see 'Watchpoint exceptions'.

ISS encoding for an exception from execution of a Breakpoint instruction

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

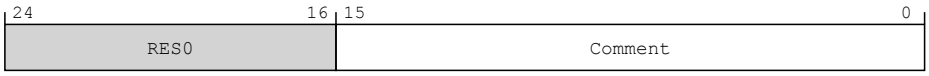

Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from execution of a Breakpoint instruction

For more information about generating these exceptions, see 'Breakpoint instruction exceptions'.

ISS encoding for an exception from an ERET, ERETAA, or ERETAB instruction

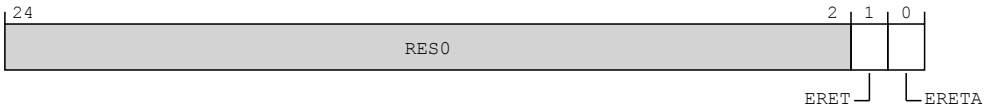

This EC value applies when:

- FEAT_FGT is implemented.
- The Effective value of HCR_EL2.NV is 1.

Bits [24:2]

Reserved, RES0.

ERET, bit [1]

Indicates whether an ERET or ERETA* instruction was trapped to EL1.

ERET	Meaning
0b0	ERET instruction trapped to EL1.
0b1	ERETAA or ERETAB instruction trapped to EL1.

If this bit is 0, the ERETA field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ERETA, bit [0]

Indicates whether an ERETAA or ERETAB instruction was trapped to EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ERETA	Meaning
0b0	ERETAA instruction trapped to EL1.
0b1	ERETAB instruction trapped to EL1.

When the ERET field is 0, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an ERET, ERETAA, or ERETAB instruction

For more information about generating these exceptions, see HCR_EL2.NV.

If FEAT_FGT is implemented, HFGITR_EL2.ERET controls fine-grained trap exceptions from ERET, ERETAA, and ERETAB execution.

ISS encoding for a Branch Target Exception

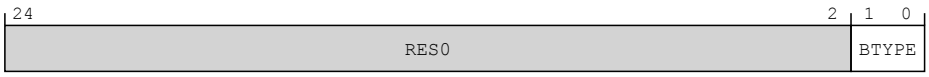

Bits [24:2]

Reserved, RES0.

BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

Additional Information for ISS encoding for a Branch Target Exception

For more information about generating these exceptions, see 'The AArch64 application level programmers model'.

ISS encoding for an exception from a trapped Pointer Authentication instruction

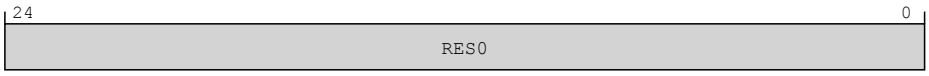

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from a trapped Pointer Authentication instruction

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for a PAC Fail exception

24 … 2	1	0
RES0	DnI	BnA

Bits [24:2]

Reserved, RES0.

DnI, bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

DnI	Meaning
0b0	Instruction Key.
0b1	Data Key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BnA, bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

BnA	Meaning
0b0	A key.
0b1	B key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for a PAC Fail exception

The following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- AUTDA, AUTDZA.
- AUTDB, AUTDZB.
- AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA.
- AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB.
- AUTIASPPC, AUTIASPPCR, AUTIA171615.
- AUTIBSPPC, AUTIBSPPCR, AUTIB171615.

If FEAT_FPACCOMBINE is implemented, the following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- RETAA, RETAB.
- RETAASPPC, RETABSPPC.
- RETAASPPCR, RETABSPPCR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BLRAA, BLRAAZ, BLRAB, BLRABZ.
- BRAA, BRAB, BRAAZ, BRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing ESR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name ESR_EL1 or ESR_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ESR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ESR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x138);
    else
        X{64}(t) = ESR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ESR_EL2();
    else
        X{64}(t) = ESR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ESR_EL1();
end;
```

```
MSR ESR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ESR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x138) = X{64}(t);
    else
        ESR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        ESR_EL2() = X{64}(t);
    else
        ESR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ESR_EL1() = X{64}(t);
end;
```

MRS <Xt>, ESR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = ESR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = ESR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ESR_EL2();
end;
```

MSR ESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        ESR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    ESR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    ESR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, ESR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x138);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ESR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = ESR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR ESR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x138) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        ESR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        ESR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.45 ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register ESR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ESR_EL2 are UNDEFINED.

Attributes

ESR_EL2 is a 64-bit register.

Field descriptions

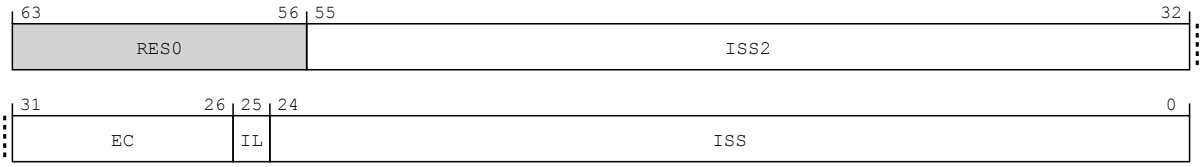

ESR_EL2 is made UNKNOWN as a result of an exception return from EL2.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of ESR_EL2 is UNKNOWN. The value written to ESR_EL2 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:56]

Reserved, RES0.

ISS2, bits [55:32]

ISS2 encoding for an exception, the bit assignments are:

ISS2 encoding for an exception from a Data Abort

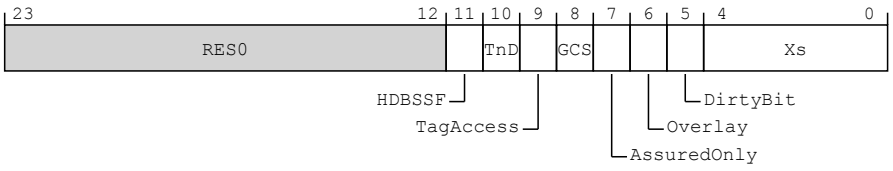

Bits [23:12]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HDBSSF, bit [11]

When FEAT_HDBSS is implemented, IsSecondStage(Fault), and GetESR_ELx_DFSC(EL2) IN {'0011xx', '01001x', '0101xx', '10001x', '1001xx'}:

When DFSC indicates a stage 2 Permission fault, this field indicates whether that fault was caused because HDBSS cannot have data appended, because the HDBSS is either full or in an error state.

When DFSC indicates a synchronous External abort on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

When DFSC indicates a Granule Protection Fault on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

HDBSSF	Meaning
`0b0`	Fault was not caused by HDBSS.
`0b1`	Fault was caused by HDBSS.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TnD, bit [10]

When FEAT_MTE_CANONICAL_TAGS is implemented:

Tag not Data.

If a memory access generates a Data Abort for a stage 1 Permission fault, this field indicates whether the fault is due to an Allocation Tag access.

TnD	Meaning
`0b0`	Permission fault is not due to a write of an Allocation Tag to Canonically Tagged memory.
`0b1`	Permission fault is due to a write of an Allocation Tag to Canonically Tagged memory.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TagAccess, bit [9]

When FEAT_MTE_PERM is implemented:

NoTagAccess fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If a memory access generates a Data Abort for a Permission fault, this field indicates whether the fault is due to the NoTagAccess memory attribute.

TagAccess	Meaning
0b0	Permission fault is not due to the NoTagAccess memory attribute, and not due to an access using a Tag VA.
0b1	Permission fault is due to the NoTagAccess memory attribute.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

GCS, bit [8]

When FEAT_GCS is implemented:

Guarded Control Stack data access.

If a memory access generates a Data Abort, this field indicates whether the fault is due to a Guarded Control Stack data access.

GCS	Meaning
0b0	The Data Abort is not due to a Guarded control stack data access.
0b1	The Data Abort is due to a Guarded control stack data access. The ISV field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented:

AssuredOnly flag.

If a memory access generates a Data Abort for a stage 2 Permission fault, this field holds information about the fault.

If this field is 1 and ESR_EL2.GCS is 1, then the AssuredOnly check might have been the result of VTCR_EL2.GCSH configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AssuredOnly	Meaning
0b0	The Data Abort is not due to AssuredOnly.
0b1	The Data Abort is due to AssuredOnly.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When FEAT_S1POE is implemented or FEAT_S2POE is implemented:

Overlay flag.

If a memory access generates a Data Abort for a Permission fault, this field holds information about the fault.

Overlay	Meaning
0b0	The Data Abort is not due to Overlay Permissions.
0b1	The Data Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When FEAT_S1PIE is implemented or FEAT_S2PIE is implemented:

DirtyBit flag.

If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

For any other fault or Access, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Xs, bits [4:0]

When FEAT_LS64 is implemented:

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort exception reported with ISV=1, then this field holds register specifier, Xs.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort exception reported with ISV=1, then this field holds register specifier, Xs.

Otherwise, this field is RES0.

Otherwise:

Reserved, RES0.

ISS2 encoding for an exception from an Instruction Abort

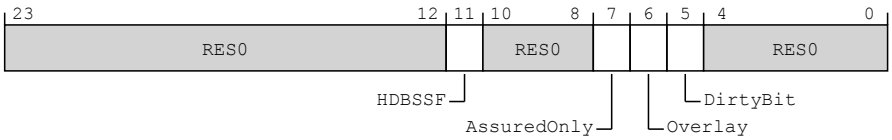

Bits [23:12]

Reserved, RES0.

HDBSSF, bit [11]

When FEAT_HDBSS is implemented:

Indicates that the fault was caused by the HDBSS.

When IFSC indicates a stage 2 Permission fault, this field indicates whether that fault was caused because HDBSS cannot have data appended, because the HDBSS is either full or in an error state.

When IFSC indicates a synchronous External abort on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

When IFSC indicates a Granule Protection Fault on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

HDBSSF	Meaning
0b0	Fault was not caused by HDBSS.
0b1	Fault was caused by HDBSS.

This field only applies for stage 2 Permission faults.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [10:8]

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented:

AssuredOnly flag.

If a memory access generates a Instruction Abort for a Permission fault, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Instruction Abort is not due to AssuredOnly.
0b1	Instruction Abort is due to stage 2 AssuredOnly attribute.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When FEAT_S1POE is implemented or FEAT_S2POE is implemented:

Overlay flag.

If a memory access generates a Instruction Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Instruction Abort is not due to Overlay Permissions.
0b1	Instruction Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When FEAT_S2PIE is implemented:

DirtyBit flag.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If a write access to memory generates an Instruction Abort for a Permission fault using Indirect Permission, this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

For any other fault or Access, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

ISS2 encoding for an exception from a Watchpoint exception

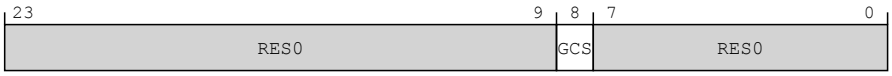

Bits [23:9]

Reserved, RES0.

GCS, bit [8]

When FEAT_GCS is implemented:

Guarded control stack data access.

Indicates that the Watchpoint exception is due to a Guarded control stack data access.

GCS	Meaning
0b0	The Watchpoint exception is not due to a Guarded control stack data access.
0b1	The Watchpoint exception is due to a Guarded control stack data access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [7:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS2 encoding for all other exceptions

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC	Meaning	ISS	ISS2	Applies when
`0b000000`	Unknown reason.	ISS encoding for exceptions with an unknown reason	ISS2 encoding for all other exceptions	
`0b000001`	Trapped WF* instruction execution. Conditional WF* instructions that fail their condition code check do not cause an exception.	ISS encoding for an exception from a WF* instruction	ISS2 encoding for all other exceptions	
`0b000011`	Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC value `0b000000`.	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000100`	Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC value `0b000000`.	ISS encoding for an exception from an MCRR or MRRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000101`	Trapped MCR or MRC access with (coproc==`0b1110`).	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000110`	Trapped LDC or STC access. The only architected uses of these instruction are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS encoding for an exception from an LDC or STC instruction	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b000111	Access to SME, SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.	ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps	ISS2 encoding for all other exceptions	
0b001000	Trapped VMRS access, from ID group trap, that is not reported using EC value 0b000111.	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b001001	Trapped use of a Pointer authentication instruction.	ISS encoding for an exception from a trapped Pointer Authentication instruction	ISS2 encoding for all other exceptions	FEAT_PAuth is implemented
0b001010	Trapped execution of any instruction not covered by other EC values.	ISS encoding for an exception from any other instruction	ISS2 encoding for all other exceptions	FEAT_LS64 is implemented, or FEAT_SPEv1p5 is implemented, or FEAT_TRBEv1p1 is implemented
0b001100	Trapped MRRC access with (coproc==0b1110).	ISS encoding for an exception from an MCRR or MRRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b001101	Branch Target Exception.	ISS encoding for a Branch Target Exception	ISS2 encoding for all other exceptions	FEAT_BTI is implemented
0b001110	Illegal Execution state.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b010001	SVC instruction execution in AArch32 state. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b010010	HVC instruction execution in AArch32 state, when HVC is not disabled.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b010011	SMC instruction execution in AArch32 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS encoding for an exception from SMC instruction execution in AArch32 state	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b010100	Trapped MSRR, MRRS or System instruction execution in AArch64 state, that is not reported using EC value 0b000000.	ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_SYSREG128 is implemented or FEAT_SYSINSTR128 is implemented
0b010101	SVC instruction execution in AArch64 state.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b010110	HVC instruction execution in AArch64 state, when HVC is not disabled.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b010111	SMC instruction execution in AArch64 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS encoding for an exception from SMC instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC values 0b000000, 0b000001 or 0b000111. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview', except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b011001	Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC value 0b000000.	ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ	ISS2 encoding for all other exceptions	FEAT_SVE is implemented
0b011010	Trapped ERET, ERETAA, or ERETAB instruction execution.	ISS encoding for an exception from an ERET, ERETAA, or ERETAB instruction	ISS2 encoding for all other exceptions	FEAT_FGT is implemented or FEAT_NV is implemented
0b011100	Exception from a PAC Fail	ISS encoding for a PAC Fail exception	ISS2 encoding for all other exceptions	FEAT_FPAC is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b011101	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, CPTR_EL3.ESM, or an attempted execution of an instruction that is illegal because of the value of PSTATE.SM or PSTATE.ZA, that is not reported using EC value 0b000000.	ISS encoding for an exception due to SME functionality	ISS2 encoding for all other exceptions	FEAT_SME is implemented
0b100000	Instruction Abort from a lower Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100001	Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100010	PC alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b100100	Data Abort exception from a lower Exception level, excluding Data Abort exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. These Data Abort exceptions might be generated from Exception levels in any Execution state. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b100101	Data Abort exception without a change in Exception level, or Data Abort exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100110	SP alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b100111	Memory Operation Exception.	ISS encoding for an exception from the Memory Copy and Memory Set instructions	ISS2 encoding for all other exceptions	FEAT_MOPS is implemented
0b101000	Trapped floating-point exception taken from AArch32 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b101100	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b101101	GCS exception.	ISS encoding for a GCS exception	ISS2 encoding for all other exceptions	FEAT_GCS is implemented
0b101111	SError exception.	ISS encoding for an SError exception	ISS2 encoding for all other exceptions	
0b110000	Breakpoint exception from a lower Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b110001	Breakpoint exception taken without a change in Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	
0b110010	Software Step exception from a lower Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110011	Software Step exception taken without a change in Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110100	Watchpoint from a lower Exception level, excluding Watchpoint Exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. These Watchpoint Exceptions might be generated from Exception levels using any Execution state.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	
0b110101	Watchpoint exceptions without a change in Exception level, or Watchpoint exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	
0b111000	BKPT instruction execution in AArch32 state.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b111010	Vector Catch exception from AArch32 state. The only case where a Vector Catch exception is taken to an Exception level that is using AArch64 is when the exception is routed to EL2 and EL2 is using AArch64.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b111100	BRK instruction execution in AArch64 state.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b111101	Profiling exception	ISS encoding for a Profiling exception	ISS2 encoding for all other exceptions	FEAT_EBEP is implemented, or FEAT_SPE_EXC is implemented, or FEAT_TRBE_EXC is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other EC values are reserved by Arm, and:

- Unused values in the range `0b000000` - `0b101100` (`0x00` - `0x2C`) are reserved for future use for synchronous exceptions.
- Unused values in the range `0b101101` - `0b111111` (`0x2D` - `0x3F`) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

IL	Meaning
`0b0`	16-bit instruction trapped.

`0b1`	32-bit instruction trapped. This value is also used when the exception is one of the following:
	• An SError exception.
	• An Instruction Abort exception.
	• A PC alignment fault exception.
	• An SP alignment fault exception.
	• A Data Abort exception for which the value of the ISV bit is 0.
	• An Illegal Execution state exception.
	• Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
	  • ‘0’: 16-bit T32 BKPT instruction.
	  • ‘1’: 32-bit A32 BKPT instruction or A64 BRK instruction.
	• An exception reported using EC value `0b000000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see ‘Mapping of the general-purpose registers between the Execution states’.

If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The value `0b11111`.

ISS encoding for exceptions with an unknown reason

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for exceptions with an unknown reason

When an exception is reported using this EC value, the IL field is set to 1.

This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution of an MSR or MRS instruction using a _EL12 register name when the Effective value of HCR_EL2.E2H is not 1.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, and EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1. If the value of HCR_EL2.TGE is 0, this exception is reported using an ESR_EL2.EC value of `0b000111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_UINJ is implemented, an Undefined exception caused by attempting to execute any instruction when PSTATE.UINJ is 1.

ISS encoding for an exception from a WF* instruction

24	23 20	19 10	9 5	4 3	2	1 0
CV	COND	RES0	RN	RES0	RV	TI

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [19:10]

Reserved, RES0.

RN, bits [9:5]

When FEAT_WFxT is implemented:

Register Number. Indicates the register number supplied for a `WFET` or `WFIT` instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.

RV, bit [2]

When FEAT_WFxT is implemented:

Register field Valid.

If TI[1] == 1, then this field indicates whether RN holds a valid register number for the register argument to the trapped `WFET` or `WFIT` instruction.

RV	Meaning
`0b0`	Register field invalid.
`0b1`	Register field valid.

If TI[1] == 0, then this field is RES0.

This field is set to 1 on a trap on `WFET` or `WFIT`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

TI	Meaning	Applies when
`0b00`	WFI trapped.	
`0b01`	WFE trapped.	
`0b10`	WFIT trapped.	FEAT_WFxT is implemented
`0b11`	WFET trapped.	FEAT_WFxT is implemented

When FEAT_WFxT is implemented, this is a two bit field as shown. Otherwise, bit[1] is RES0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a WF* instruction

The following fields describe configuration settings for generating this exception:

- HCR.{TWE, TWI}.
- SCTLR_EL1.{nTWE, nTWI}.
- SCTLR_EL2.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

ISS encoding for an exception from an MCR or MRC access

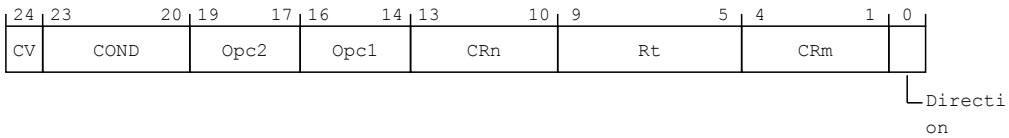

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

If the Rt value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See ‘Mapping of the general-purpose registers between the Execution states’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write to System register space. MCR instruction.
`0b1`	Read from System register space. MRC or VMRS instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an MCR or MRC access

The following fields describe configuration settings for generating exceptions from an MCR or MRC access using coproc `0b1111`, that are reported using EC value `0b000011`:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, trapped to EL1.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors System registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- HCR.{TRVM, TVM} and HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, trapped to EL2.
- HCR.TTLB and HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, trapped to EL2.
- HCR.{TSW, TPC, TPU} and HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, trapped to EL2.
- HCR.TAC and HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, trapped to EL2.
- HCR.TIDCP and HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, trapped to EL2.
- If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, trapped to EL2.
- HCR.{TID1, TID2, TID3} and HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to error record registers at EL1 using AArch32 state, trapped to EL2.
- HCPTR.TCPAC and CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, trapped to EL2.
- HSTR.T<n> and HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, trapped to EL2.
- CNTHCTL.PL1PCEN and CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HDCR.TTRF, for Non-secure accesses to trace filter control registers from system registers using AArch32 state, trapped to EL2.
- HDCR.{TPM, TPMCR} and MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HCPTR.TAM and CPTR_EL2.TAM, for accesses to Activity Monitors System registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CPTR_EL3.TAM, for accesses to Activity Monitors System registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions from an MCR or MRC access using coproc `0b1110`, that are reported using EC value `0b000101`:

- CPACR_EL1.TTA for accesses to trace System registers, trapped to EL2 or EL1.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, trapped to EL2 or EL1.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC System registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC System registers at EL0, EL1, and EL2 trapped to EL3.
- HCR.TID0 and HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, trapped to EL2.
- HCPTR.TTA and CPTR_EL2.TTA, for accesses to trace System registers using AArch32, trapped to EL2.
- HDCR.TDRA and MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, trapped to EL2.
- HDCR.TDOSA and MDCR_EL2.TDOSA, for accesses to powerdown debug System registers, using AArch32 state, trapped to EL2.
- HDCR.TDA and MDCR_EL2.TDA, for accesses to other debug System registers, using AArch32 state, trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace System registers using AArch32, trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug System registers using AArch32, trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug System registers, using AArch32, trapped to EL3.

The following fields describe configuration settings for generating exceptions from a VMSR or VMRS access, that are reported using EC value `0b001000`:

- HCR.TID0 and HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR.TID3 and HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.
- HCPTR.{TCP10, TCP11}, for Non-secure accesses to FPSCR, FPSID, FPEXC, MVFR0, MVFR1, and MVFR2, trapped to EL2.

ISS encoding for an exception from any other instruction

24 … 0
ISS

ISS, bits [24:0]

ISS	Meaning	Applies when
`0b0000000000000000000000000`	`ST64BV` instruction trapped.	FEAT_LS64_V is implemented
`0b0000000000000000000000001`	`ST64BV0` instruction trapped.	FEAT_LS64_ACCDATA is implemented
`0b0000000000000000000000010`	`LD64B` or `ST64B` instruction trapped.	FEAT_LS64 is implemented
`0b0000000000000000000000011`	`PSB CSYNC` instruction trapped.	FEAT_SPEv1p5 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS	Meaning	Applies when
0b00000000000000000000000100	TSB CSYNC instruction trapped.	FEAT_TRBEv1p1 is implemented

All other values are reserved.

ISS encoding for an exception from an MCRR or MRRC access

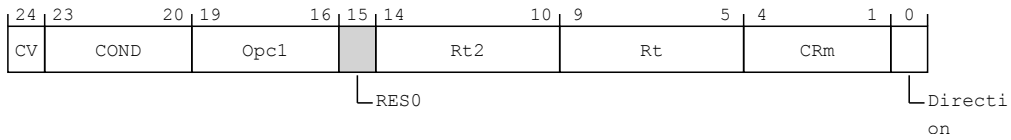

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

If the Rt2 value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt2 value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See ‘Mapping of the general-purpose registers between the Execution states’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

If the Rt value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See ‘Mapping of the general-purpose registers between the Execution states’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to System register space. MCRR instruction.
0b1	Read from System register space. MRRC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an MCRR or MRRC access

The following fields describe configuration settings for generating exceptions from an MCRR or MRRC access using coproc 0b1111, that are reported using EC value 0b000100:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors System registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, trapped to EL2 or EL1.
- HCR.{TRVM, TVM} and HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to error record registers at EL1 using AArch32 state, trapped to EL2.
- HSTR.T<n> and HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, trapped to EL2.
- CNTHCTL.{PL1PCEN, PL1PCTEN} and CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HDCR.TPM and MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HCPTR.TAM and CPTR_EL2.TAM, for accesses to Activity Monitors System registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions from an MCRR or MRRC access using coproc 0b1110, that are reported using EC value 0b001100:

- MDSCR_EL1.TDCC, for accesses to the Debug ROM registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, trapped to EL2 or EL1.
- HDCR.TDRA and MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, trapped to EL2.
- MDCR_EL3.TDA, for accesses to debug System registers, using AArch32, trapped to EL3.
- CPACR_EL1.TTA for accesses to trace System registers using AArch32, trapped to EL1 or EL2.
- HCPTR.TTA and CPTR_EL2.TTA, for accesses to trace System registers using AArch32, trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace System registers using AArch32, trapped to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If FEAT_ETMv4 or FEAT_ETE are implemented, MCRR and MRRC accesses to trace System registers are UNDEFINED and the resulting exception is higher priority than an exception due to these traps.

ISS encoding for an exception from an LDC or STC instruction

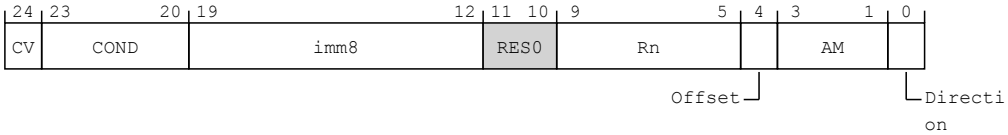

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer.

If the Rn value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rn value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See ‘Mapping of the general-purpose registers between the Execution states’.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Offset	Meaning
`0b0`	Subtract offset.
`0b1`	Add offset.

This bit corresponds to the U bit in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AM	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in 'Reserved values in System and memory-mapped registers and translation table entries'.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an LDC or STC instruction

The following fields describe the configuration settings from an LDC or STC access for the traps that are reported using EC value 0b000110:

- MDSCR_EL1.TDCC, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL2 or EL1.
- HDCR.TDA and MDCR_EL2.TDA, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL2.
- MDCR_EL3.TDA, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC System registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC System registers at EL0, EL1, and EL2 trapped to EL3.

ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

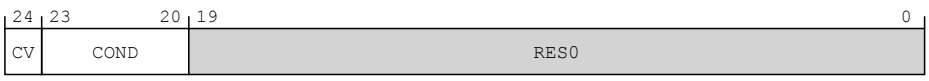

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The accesses covered by this trap include:

- Execution of Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.
- Execution of SVE instructions.
- Execution of SME instructions.

For an implementation that does not include either SVE or support for Advanced SIMD and floating-point, the exception is reported using the EC value `0b000000`.

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional Information for ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

The following fields describe the configuration settings for the traps that are reported using EC value `0b000111`:

- HCPTR.{TCP10, TCP11}, for Non-secure accesses to Advanced SIMD and floating-point registers and instructions, trapped to EL2.
- HCPTR.TASE, for Non-secure accesses to Advanced SIMD functionality, trapped to EL2.
- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL3.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

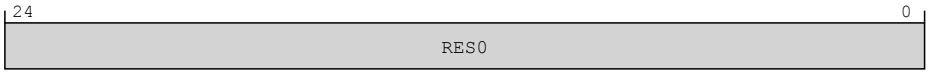

The accesses covered by this trap include:

- Execution of SVE instructions when the PE is not in Streaming SVE mode.
- Accesses to the SVE System registers, ZCR_ELx.

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

The following fields describe the configuration settings for the traps that are reported using EC value `0b011001`:

- CPACR_EL1.ZEN, for execution of SVE instructions and accesses to SVE registers at EL1 or EL0, trapped to EL1.
- CPTR_EL2.ZEN and CPTR_EL2.TZ, for execution of SVE instructions and accesses to SVE registers at EL0, EL1, or EL2, trapped to EL2.
- CPTR_EL3.EZ, for execution of SVE instructions and accesses to SVE registers from all Exception levels, trapped to EL3.

ISS encoding for a Profiling exception

Bits [24:6]

Reserved, RES0.

FSC, bits [5:1]

Indicates why the Profiling exception was generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b00000	PMU Profiling exception. The exception was generated because at least one PMU counter overflow status flag was 1.	FEAT_EBEP is implemented
0b00001	Profiling Buffer management event. The exception was generated because PMBSR_EL2.S was 1.	FEAT_SPE_EXC is implemented
0b00010	Trace buffer management event. The exception was generated because TRBSR_EL2.IRQ was 1.	FEAT_TRBE_EXC is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

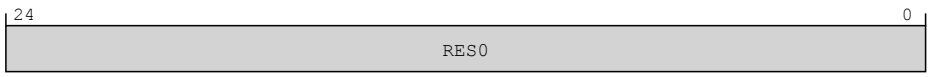

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see 'PC alignment checking'.

'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from the Memory Copy and Memory Set instructions

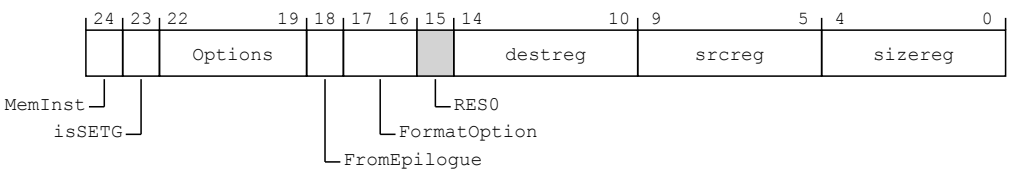

MemInst, bit [24]

Indicates the memory instruction class causing the exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MemInst	Meaning
0b0	CPYFE*, CPYFM*, CPYE*, and CPYM* instructions.
0b1	The instruction that caused the exception is in one of the following instruction classes: • SETE*, SETM*. • SETGE*, SETGM*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

isSETG, bit [23]

Indicates whether the instruction belongs to one of the following instruction classes:

- SETGE*, SETGM*.

isSETG	Meaning
0b0	The instruction that caused the exception is not in one of the specified classes.
0b1	The instruction that caused the exception is in one of the specified classes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Options, bits [22:19]

Options : the Options field of the instruction.

For Memory Copy instructions, bits[22:19] forms the Options field, which holds the bits[15:12] of the instruction.

For Memory Set instructions:

- Bits[22:21] are RES0.
- Bits[20:19] form the Options field, which holds the bits[13:12] of the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FromEpilogue, bit [18]

Indicates whether the instruction belongs to one of the following epilogue classes of Memory Copy or Memory Set instructions:

- CPYE*, CPYFE*.
- SETE*, SETGE*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FromEpilogue	Meaning
0b0	The instruction that caused the exception is not in one of the specified classes.
0b1	The instruction that caused the exception is in one of the specified classes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FormatOption, bits [17:16]

Reports the Option used to encode the initial Xs, Xd, and Xn register values provided to the instruction that generated the exception.

FormatOption	Meaning
0b00	Option B.
0b01	Option A.
0b10	Option A.
0b11	Option B.

For more information, see Memory Copy and Memory Set exceptions.

Note

This field was previously presented as two separate bits, WrongOption, bit[17] and OptionA, bit [16], which were already expected to be used together and not individually.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

destreg, bits [14:10]

The destination register value from the issued instruction, containing the destination address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

srcreg, bits [9:5]

The source register value from the issued instruction, containing either the source address or the source data.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

sizereg, bits [4:0]

The size register value from the issued instruction, containing the number of bytes to be transferred or set.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for an exception from HVC or SVC instruction execution

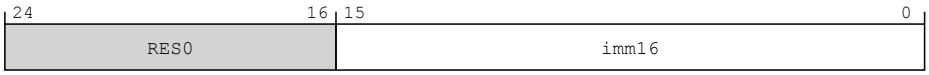

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from HVC or SVC instruction execution

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see 'SVC' and 'HVC'.

For A64 instructions, see 'SVC' and 'HVC'.

If FEAT_FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

ISS encoding for an exception from SMC instruction execution in AArch32 state

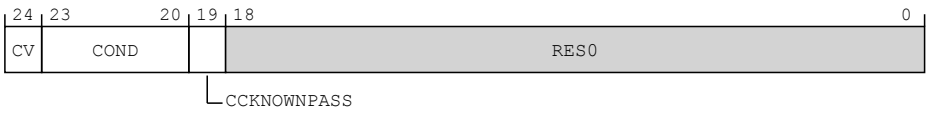

For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

CCKNOWNPASS	Meaning
0b0	The instruction was unconditional, or was conditional and passed its condition code check.
0b1	The instruction was conditional, and might have failed its condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [18:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from SMC instruction execution in AArch32 state

HCR.TSC describes the configuration settings for trapping SMC instructions to EL2.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

ISS encoding for an exception from SMC instruction execution in AArch64 state

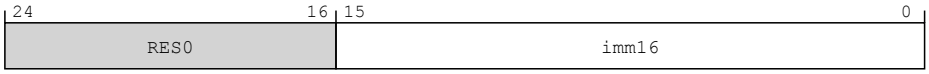

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from SMC instruction execution in AArch64 state

The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

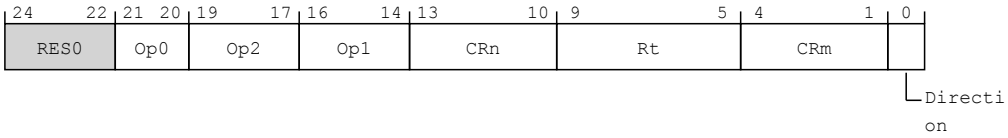

Bits [24:22]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

For system instructions which require that the opcode Rt field is set to `0b11111`, but where the trapped instruction has a different value of Rt, an implementation is permitted to return the value `0b11111`, instead of the value of Rt from the trapped instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write access, including MSR instructions.
`0b1`	Read access, including MRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For exceptions caused by System instructions, see 'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value `0b011000`:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL1.
- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2 or EL1.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2 or EL1.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- CPACR_EL1.TTA, for accesses to the trace System registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- PMUSERENR_EL0, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors System registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, execution is trapped to EL2.
- If FEAT_EVT is implemented, the following registers control traps for EL1 and EL0 registers and instructions that use this EC value:
  - HCR_EL2.{TICAB, TOCU, TID4}.
  - HCR2.{TICAB, TOCU, TID4}.
- If FEAT_EVT2 is implemented, the following registers control traps for EL1 and EL0 instructions that use this EC value:
  - HCR_EL2.{TTLBOS, TTLBIS}.
- If FEAT_EVT2 is implemented, HCR2.TTLBIS controls traps for EL1 and EL0 instructions using AArch32 state that use this EC value.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- If FEAT_MTE2 is implemented, HCR_EL2.TID5, for accesses to GMID_EL1, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_IDTE3 is implemented, SCR_EL3.TID3, for accesses to ID group 3 registers using AArch64 state, at EL1 and EL2, trapped to EL3.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace System registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug System registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug System registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors System registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the trace filter control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- If FEAT_SPE is implemented:
  - MDCR_EL3.NSPB for accesses to Statistical Profiling and Profiling Buffer control registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
  - MDCR_EL2.TPMS for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 trapped to EL2.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If FEAT_FGT is implemented:
  - SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR_EL2 for execution of system instructions trapped to EL2.
  - HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  - HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.
- If FEAT_RNG_TRAP is implemented, SCR_EL3.TRNDR for reads of RNDR and RNDRRS using AArch64 state, MRS access trapped to EL3.
- If FEAT_SME is implemented:
  - CPTR_EL3.ESM, for MSR or MRS accesses to SMPRI_EL1 at EL1, EL2, and EL3, trapped to EL3.
  - CPTR_EL3.ESM, for MSR or MRS accesses to SMPRIMAP_EL2 at EL2 and EL3, trapped to EL3.
  - SCTLR_EL1.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2 or EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SCTLR_EL2.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2.
  - SCR_EL3.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, EL1, and EL2, trapped to EL3.
- If FEAT_FPMR is implemented:
  - SCTLR_EL1.EnFPM, for accesses to FPMR at EL0, trapped to EL2 or EL1.
  - SCTLR_EL2.EnFPM, for accesses to FPMR at EL0, trapped to EL2.
  - HCRX_EL2.EnFPM, for accesses to FPMR at EL0 and EL1, trapped to EL2.
  - SCR_EL3.EnFPM, for accesses to FPMR at EL0, EL1, and EL2, trapped to EL3.
- If FEAT_NMI is implemented, HCRX_EL2.TALLINT, for MSR writes of ALLINT at EL1, trapped to EL2.
- If FEAT_FGT2 is implemented:
  - SCR_EL3.FGTEn2, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR2_EL2 for reads and HFGWTR2_EL2 for writes of registers, using AArch64 state, using MSR or MRS access at EL1 trapped to EL2.
  - HDFGRTR2_EL2 for reads and HDFGWTR2_EL2 for writes of registers, using AArch64 state, using MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR2_EL2 for execution of system instructions trapped to EL2.
- If FEAT_ITE is implemented, MDCR_EL3.EnITE, for accesses to Instrumentation trace System registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If FEAT_MEC is implemented, SCR_EL3.MECEn, for accesses to MECID registers at EL2, trapped to EL3.
- If FEAT_SPE_FDS is implemented, MDCR_EL3.EnPMS3 for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
- If FEAT_SPE_nVM is implemented, MDCR_EL3.EnPMS4 for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
- If FEAT_RASv2 is implemented, SCR_EL3.TWERR, for accesses to Error Record registers, MSR access at EL1 and EL2 trapped to EL3.
- If FEAT_Debugv8p9 is implemented, MDCR_EL3.EBWE for accesses of MDSELR_EL1, using AArch64 state, MRS or MSR access at EL2 and EL1 trapped to EL3.
- If FEAT_PMUv3p9, FEAT_SPMU, FEAT_EBEP, or FEAT_PMUv3_SS is implemented, MDCR_EL3.EnPM2, for accesses to PMU registers, using AArch64 state, MSR or MRS access at EL2, EL1, and EL0, trapped to EL3.
- If FEAT_PMUv3_SS is implemented, MDCR_EL3.EnPMSS, for accesses to PMU Snapshot registers, using AArch64 state, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_THE is implemented, SCR_EL3.RCWMASKEn for accesses to RCWMASK_EL1 and RCWSMASK_EL1, using AArch64 state, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_AIE is implemented, SCR_EL3.AIEn for accesses to Extended Memory Attribute registers, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_S1PIE, FEAT_S2PIE, FEAT_S1POE, or FEAT_S2POE is implemented, SCR_EL3.PIEn for accesses to Permission Indirection, Overlay registers, MSR or MRS access at EL2, EL1 and EL0 trapped to EL3.
- If FEAT_MPAM_PE_BW_CTRL is implemented, MPAMBW2_EL2.{nTRAP_MPAMBWIDR_EL1, nTRAP_MPAMBW0_EL1, nTRAP_MPAMBW1_EL1} for accesses to MPAMBWn_EL1 registers, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_MPAM_PE_BW_CTRL and FEAT_SME are implemented, MPAMBW2_EL2.nTRAP_MPAMBWSM_EL1, for accesses to MPAMBWSM_EL1, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_MPAM_PE_BW_CTRL is implemented, MPAMBW3_EL3.nTRAPLOWER, for accesses to MPAMBW2_EL2 and MPAMBWn_EL1 registers, using AArch64 state, MRS or MSR access at EL1, trapped to EL3.
- If FEAT_HACDBS is implemented, SCR_EL3.HACDBSEn, for accesses to HACDBSBR_EL2 and HACDBSCONS_EL2, using AArch64 state, MRS or MSR access at EL2, trapped to EL3.
- If FEAT_HDBSS is implemented, SCR_EL3.HDBSSEn, for accesses to HDBSSBR_EL2 and HDBSSPROD_EL2, using AArch64 state, MRS or MSR access at EL2, trapped to EL3.
- If FEAT_SRMASK is implemented, SCR_EL3.SRMASKEn, for MSR or MRS accesses at EL2 or EL1 to the MASK registers using AArch64 state, trapped to EL3.

ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

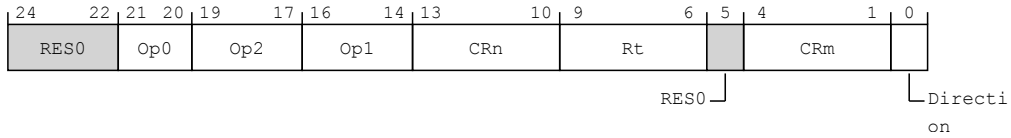

Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:6]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

Note

This value represents register pair of X[Rt:0], X[Rt:1].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write access, MSRR instructions.
`0b1`	Read access, MRRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

The following fields describe configuration settings for generating exceptions from an MSRR or MRRS access that are reported using EC value `0b010100`:

- If FEAT_FGT is implemented:
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, accesses at EL1 trapped to EL2.
- If FEAT_FGT2 is implemented:
  - HFGRTR2_EL2.nRCWSMASK_EL1 for reads and HFGWTR2_EL2.nRCWSMASK_EL1 for writes of RCWSMASK_EL1, using AArch64 state, accesses at EL1 trapped to EL2.
- If FEAT_SYSREG128 is implemented:
  - SCTLR2_EL1.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL0 trapped to EL1.
  - SCTLR2_EL2.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL0 trapped to EL2.
  - HCRX_EL2.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL1 and EL0 trapped to EL2.
  - SCR_EL3.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL2, EL1, and EL0 trapped to EL3.
- If FEAT_D128 is implemented:
  - HCR_EL2.{TRVM, TVM} for accesses to TTBR0_EL1 and TTBR1_EL1, accesses at EL1 and EL0 trapped to EL2.
  - HCRX_EL2.D128En for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL1 trapped to EL2.
  - HFGITR_EL2 for execution of TLBIP system instructions trapped to EL2.
  - HCR_EL2.{TTLB, TTLBOS, TTLBIS}, for execution of TLBIP instructions using AArch64 state, execution is trapped to EL2.
  - SCR_EL3.D128En for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL2 and EL1 trapped to EL3.
- If FEAT_THE is implemented, SCR_EL3.RCWMASKEn for accesses to RCWMASK_EL1 and RCWSMASK_EL1, using AArch64 state, accesses at EL2 and EL1 trapped to EL3.

ISS encoding for an exception from an Instruction Abort

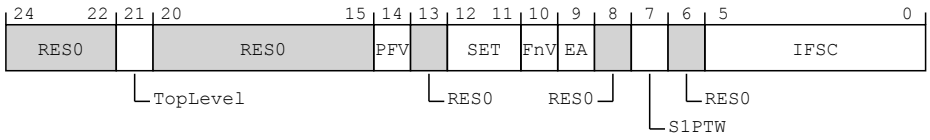

The ISS2 encoding for an exception from an Instruction Abort includes further information about the exception if any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_THE is implemented and memory access generates an Instruction Abort for AssuredOnly.
- FEAT_S1POE or FEAT_S2POE is implemented and a memory access generates an Instruction Abort due to a Permission fault.
- FEAT_HDBSS.
- FEAT_S2PIE.

Bits [24:22]

Reserved, RES0.

TopLevel, bit [21]

When FEAT_THE is implemented:

Indicates if the fault was due to TopLevel.

TopLevel	Meaning
`0b0`	Fault is not due to TopLevel.
`0b1`	Fault is due to TopLevel.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [20:15]

Reserved, RES0.

PFV, bit [14]

When FEAT_PFAR is implemented:

PFAR Valid. Describes whether the PFAR_EL2 register is valid.

PFV	Meaning
`0b0`	PFAR_EL2 is UNKNOWN.
`0b1`	PFAR_EL2 is valid.

This field is valid only if the IFSC code is `0b10000`, `0b01001x`, or `0b0101xx`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [13]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET, bits [12:11]

When FEAT_RAS is implemented and GetESR_ELx_IFSC(EL2) == '010000':

Synchronous Error Type. When IFSC is `0b010000`, describes the PE error state after taking the Instruction Abort exception.

SET	Meaning	Applies when
`0b00`	Recoverable state (UER).	
`0b10`	Uncontainable (UC).	FEAT_RASv2 is not implemented
`0b11`	Restartable state (UEO).	

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

When GetESR_ELx_IFSC(EL2) == '010000':

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
`0b0`	FAR is valid.
`0b1`	FAR is not valid, and holds an UNKNOWN value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception due to SME functionality

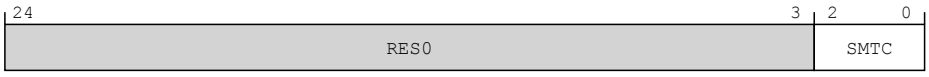

The accesses covered by this trap include:

- Execution of SME instructions.
- Execution of SVE and Advanced SIMD instructions, when the PE is in Streaming SVE mode.
- Direct accesses of SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3.

Bits [24:3]

Reserved, RES0.

SMTC, bits [2:0]

SME Trap Code. Identifies the reason for instruction trapping.

SMTC	Meaning	Applies when
0b000	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC value 0b000000.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SMTC	Meaning	Applies when
0b001	Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1.	
0b010	SME instruction trapped because PSTATE.SM is 0.	
0b011	SME instruction trapped because PSTATE.ZA is 0.	
0b100	Access to the SME2 ZT0 register trapped as a result of SMCR_EL1.EZT0, SMCR_EL2.EZT0, or SMCR_EL3.EZT0.	FEAT_SME2 is implemented

All other values are reserved.

Additional Information for ISS encoding for an exception due to SME functionality

The following fields describe the configuration settings for the traps that are reported using the EC value 0b011101:

- CPACR_EL1.SMEN, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR and SMCR_EL1 System registers at EL1 and EL0, trapped to EL2 or EL1.
- CPTR_EL2.SMEN and CPTR_EL2.TSM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2 at EL2, EL1, and EL0, trapped to EL2.
- CPTR_EL3.ESM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3 from all Exception levels and any Security state, trapped to EL3.
- If FEAT_SME2 is implemented:
  - SMCR_EL1.EZT0, for accesses to ZT0 at EL1 and EL0, trapped to EL2 or EL1.
  - SMCR_EL2.EZT0, for accesses to ZT0 at EL2, EL1, and EL0, trapped to EL2.
  - SMCR_EL3.EZT0, for accesses to ZT0 at any Exception level, trapped to EL3.

ISS encoding for an exception from a Data Abort

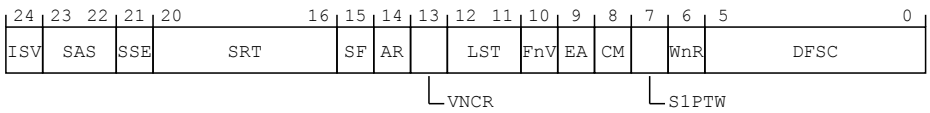

The ISS2 encoding for an exception from a Data Abort includes further information about the exception when any of the following features are implemented:

- FEAT_LS64_V.
- FEAT_LS64_ACCDATA.
- FEAT_THE.
- FEAT_S1POE or FEAT_S2POE.
- FEAT_S1PIE or FEAT_S2PIE.
- FEAT_GCS.
- FEAT_MTE_CANONICAL_TAGS.
- FEAT_MTE_PERM.
- FEAT_HDBSS.

ISV, bit [24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] represents the instruction syndrome fields {SAS, SSE, SRT, SF, AR}.

ISV	Meaning
`0b0`	No valid instruction syndrome. ISS[23:14] do not represent an instruction syndrome.
`0b1`	ISS[23:14] hold a valid instruction syndrome.

The ISV field is only relevant for the following instructions:

- An AArch64 load or store of a single general-purpose register (including the register specified with `0b11111`, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).
- An AArch32 instruction LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT, where both of the following apply:
  - It is not performing register writeback.
  - It is not using R15 as a source or destination register.
- If FEAT_LS64 is implemented, an LD64B or ST64B instruction.
- If FEAT_LS64_V is implemented, an ST64BV instruction.
- If FEAT_LS64_ACCDATA is implemented, an ST64BV0 instruction.

The ISV field is 1 if all of the following apply:

- The fault was not a stage 2 fault on a stage 1 translation table walk.
- If FEAT_RAS is implemented, the fault was not any kind of synchronous External abort.
- Any of the following apply:
  - The fault was generated by one of the described AArch64 load or store instructions that transfer a single general-purpose register, and any of the following apply:
    - The fault was a stage 2 fault.
  - The fault was generated by one of the described AArch32 instructions and any of the following apply:
    - The fault was a stage 2 fault.
  - The fault was generated by one of the FEAT_LS64, FEAT_LS64_V or FEAT_LS64_ACCDATA instructions and the fault was a Translation fault, Access flag fault or Permission fault.
- One of the following applies:
  - The address reported in FAR_EL2 is the lowest address accessed by the instruction.
  - The address reported in FAR_EL2 is not the lowest address accessed by the instruction, and the implementation reports ISV as 1 in this case. Arm recommends against this behavior.

Otherwise the value of this field is 0.

Note

The value of this field is 0 if any of the following apply:

- FEAT_MOPS is implemented and the fault was for a memory access generated by a Memory Copy and Memory Set instruction.
- FEAT_MTE is implemented and the fault was for a direct access to Allocation Tags.
- FEAT_GCS is implemented and the fault was for a Guarded control stack data access.
- FEAT_NV2 is implemented and the fault was for a system register access that was transformed to a load or store relative to VNCR_EL2.
- FEAT_LSE is implemented and the fault was for an atomic instruction.

The value of this field is UNKNOWN when the following occurs:

- Stage 2 faults when the exception was generated in Debug state, in memory access mode.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

When ISV == '1':

Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

SAS	Meaning
`0b00`	Byte
`0b01`	Halfword
`0b10`	Word
`0b11`	Doubleword

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [21]

When ISV == '1':

SSE, bit [21]

Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

SSE	Meaning
`0b0`	Sign-extension not required.
`0b1`	Data item must be sign-extended.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When ISV == '0' and FEAT_THE is implemented:

TopLevel, bit [21]

Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [20:16]

When ISV == '1':

SRT, bits [20:16]

Syndrome Register Transfer. The register number of the Wt/Xt/Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32, then this is the AArch64 view of the register. See 'Mapping of the general-purpose registers between the Execution states'.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When ISV == '0', FEAT_RASv2 is implemented, and GetESR_ELx_DFSC(EL2) IN {'010000', '01001x', '0101xx'}:

WU, bits [1:0] of bits [20:16]

Write Update. Describes whether a store instruction that generated an External abort updated the location.

WU	Meaning
0b00	Not a store instruction or translation table update, or the location might have been updated.
0b10	Store instruction or translation table update that did not update the location.
0b11	Store instruction or translation table update that updated the location.

In the description of this field, a store instruction is any memory-writing instruction that explicitly performs a store. This includes instructions that both read and write memory.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

When ISV == '1':

SF, bit [15]

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

SF	Meaning
0b0	Instruction loads/stores a 32-bit general-purpose register.
0b1	Instruction loads/stores a 64-bit general-purpose register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

FnP, bit [15]

FAR not Precise.

FnP	Meaning	Applies when
0b0	The FAR holds the faulting virtual address that generated the Data Abort.	
0b1	The FAR holds any virtual address within the naturally-aligned granule that contains the faulting virtual address that generated a Data Abort due to an SVE contiguous vector load/store instruction, or an SME load/store instruction. For more information about the naturally-aligned fault granule, see FAR_ELx (for example, FAR_EL1).	FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [14]

When ISV == '1':

AR, bit [14]

Acquire/Release.

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register being ZR, it is IMPLEMENTATION SPECIFIC whether this field is reported as 0 or 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_PFAR is implemented, ISV == '0', and GetESR_ELx_DFSC(EL2) IN {'010000', '01001x', '0101xx'}:

PFV, bit [14]

PFAR Valid. Describes whether the PFAR_EL2 register is valid.

PFV	Meaning
0b0	PFAR_EL2 is UNKNOWN.
0b1	PFAR_EL2 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VNCR, bit [13]

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VNCR	Meaning	Applies when
0b0	The fault was not generated by the use of VNCR_EL2 by EL1 code.	
0b1	The fault was generated by the use of VNCR_EL2 by EL1 code.	FEAT_NV2 is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:11]

When GetESR_ELx_DFSC(EL2) IN {'00xxxx', '10101x', '0000xx'}:

LST, bits [12:11]

Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort reported with ISV=1.

LST	Meaning	Applies when
0b00	The instruction that generated the Data Abort is not specified by this field.	
0b01	An ST64BV instruction generated the Data Abort.	FEAT_LS64_V is implemented
0b10	An LD64B or ST64B instruction generated the Data Abort.	FEAT_LS64 is implemented
0b11	An ST64BV0 instruction generated the Data Abort.	FEAT_LS64_ACCDATA is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL2) == '010000':

SET, bits [12:11]

Synchronous Error Type. Used when a synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

SET	Meaning	Applies when
0b00	Recoverable state (UER).	
0b10	Uncontainable (UC). If FEAT_RASv2 is implemented, this value is reserved.	FEAT_RASv2 is implemented
0b11	Restartable state (UEO).	

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

CM	Meaning
0b0	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Abort caused by an instruction reading from a memory location.
0b1	Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- If FEAT_RASv2 is implemented, an External abort on an Atomic access, reported with ESR_EL2.WU set to 0b00.
- A fault reported using a DFSC value of 0b110101, indicating an unsupported Exclusive or atomic access.
- A fault reported using a DFSC value of 0b110001, indicating an unsupported atomic hardware update fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Synchronous Tag Check Fault.	FEAT_MTE2 is implemented
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).	

All other values are reserved.

For more information about the lookup level associated with a fault, see ‘The lookup level associated with MMU faults’.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a trapped floating-point exception

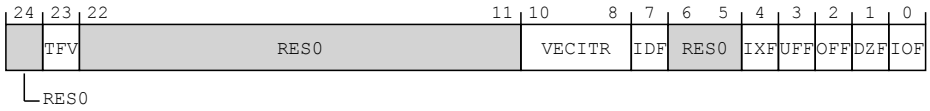

Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TFV	Meaning
0b0	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
0b1	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see 'Floating-point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IDF	Meaning
0b0	Input denormal floating-point exception has not occurred.
0b1	Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IXF	Meaning
0b0	Inexact floating-point exception has not occurred.
0b1	Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

UFF	Meaning
0b0	Underflow floating-point exception has not occurred.
0b1	Underflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

OFF	Meaning
0b0	Overflow floating-point exception has not occurred.
0b1	Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

DZF	Meaning
0b0	Divide by Zero floating-point exception has not occurred.
0b1	Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IOF	Meaning
0b0	Invalid Operation floating-point exception has not occurred.
0b1	Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a trapped floating-point exception

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

ISS encoding for a GCS exception

Bit [24]

Reserved, RES0.

ExType, bits [23:20]

The first level classification of GCS exceptions.

ExType	Meaning
0b0000	The exception reported is a Guarded Control Stack Data Check Exception.
0b0001	The exception reported is an EXLOCK Exception.
0b0010	The exception reported is a trap exception on GCSSTR or GCSSTTR instruction execution.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:15]

Reserved, RES0.

Raddr, bits [14:10]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When GetESR_ELx_ExType(EL2) == '0010':

Indicates the data address register number supplied in the instruction that has been trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [9:5]

When GetESR_ELx_ExType(EL2) == '0000':

Rn, bits [9:5]

Indicates a register number used by the instruction that caused the Guarded Control Stack Data Check Exception.

For a procedure return instruction reported with ESR_EL2.ISS.IT as `0b00000`, contains the register number for the register which contains the target address of the branch.

For a GCSPOPM instruction reported with ESR_EL2.ISS.IT as `0b00001`, contains the register number for the register which is the destination register of the instruction.

For a procedure return instruction reported with ESR_EL2.ISS.IT as `0b00010` or `0b00011`, contains the value `0b11110`, indicating X30.

For a GCSSS1 instruction reported with ESR_EL2.ISS.IT as `0b00100`, contains the register number for the register which is the input register of the instruction.

If ESR_EL2.ISS.IT is reported as `0b00101` or `0b01000`, this field is UNKNOWN

If ESR_EL2.ISS.IT is reported as `0b01001`, this field is `0b11111`

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When GetESR_ELx_ExType(EL2) == '0010':

Rvalue, bits [9:5]

Indicates the data value register number supplied in the instruction that has been trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IT, bits [4:0]

When GetESR_ELx_ExType(EL2) == '0000':

Type of the instruction that caused the Guarded Control Stack Data Check Exception.

IT	Meaning
`0b00000`	Guarded Control Stack Data Check Exception is from a procedure return instruction without Pointer authentication.
`0b00001`	Guarded Control Stack Data Check Exception is from a GCSPOPM instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IT	Meaning
0b00010	Guarded Control Stack Data Check Exception is from a procedure return instruction with Pointer authentication that uses key A.
0b00011	Guarded Control Stack Data Check Exception is from a procedure return instruction with Pointer authentication that uses key B.
0b00100	Guarded Control Stack Data Check Exception is from a GCSSS1 instruction.
0b00101	Guarded Control Stack Data Check Exception is from a GCSSS2 instruction.
0b01000	Guarded Control Stack Data Check Exception is from a GCSPOPCX instruction.
0b01001	Guarded Control Stack Data Check Exception is from a GCSPOPX instruction.

All other values are reserved

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Additional Information for ISS encoding for a GCS exception

The following fields describe the configuration settings for the traps that are reported using EC value 0b101101 and ExType value 0b0010:

- GCSCRE0_EL1.STREn
- GCSCR_EL1.STREn.
- GCSCR_EL2.STREn.
- GCSCR_EL3.STREn.
- HFGITR_EL2.nGCSSTR_EL1.

ISS encoding for an SError exception

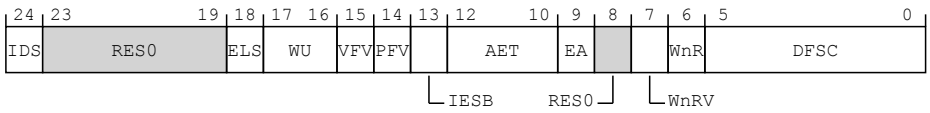

Note

In earlier versions of the architecture, an SError exception is referred to as an SError interrupt or an asynchronous External abort exception.

IDS, bit [24]

IMPLEMENTATION DEFINED syndrome.

IDS	Meaning
0b0	Bits [23:0] of the ISS field holds the fields described in this encoding. **Note** If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDS	Meaning
0b1	Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError exception.

Note

This field was previously called ISV.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:19]

Reserved, RES0.

ELS, bit [18]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL2) == '010001':

Meaning of ELR_ELx.

ELS	Meaning
0b0	Asynchronous. Does not indicate the trigger for the exception.
0b1	Synchronous. The exception was triggered by the instruction at ELR_ELx.

SError exceptions that report this field is 1 are not required to be precise.

The ESR_EL2.AET field describes whether the exception is precise or imprecise.

Corrected, Recoverable or Restartable exceptions are precise. Unrecoverable or Uncontainable exceptions are imprecise.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WU, bits [17:16]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL2) == '010001':

Write Update. Describes whether a store instruction that generated an External abort updated the location.

WU	Meaning
0b00	Not a store instruction or translation table update, or the location might have been updated.
0b10	Store instruction or translation table update that did not update the location.
0b11	Store instruction or translation table update that updated the location.

In the description of this field, a store instruction is any memory-writing instruction that explicitly performs a store. This includes instructions that both read and write memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VFV, bit [15]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL2) == '010001':

FAR Valid. Indicates the FAR_EL2 register contains a valid virtual address.

VFV	Meaning
0b0	FAR_EL2 is not valid, and holds an UNKNOWN value.
0b1	FAR_EL2 contains a valid virtual address associated with the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PFV, bit [14]

When FEAT_PFAR is implemented and GetESR_ELx_DFSC(EL2) == '010001':

PFAR Valid. Describes whether the PFAR_EL2 register is valid.

PFV	Meaning
0b0	PFAR_EL2 is UNKNOWN.
0b1	PFAR_EL2 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IESB, bit [13]

When FEAT_IESB is implemented and GetESR_ELx_DFSC(EL2) == '010001':

Implicit error synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IESB	Meaning
0b0	The SError exception was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1	The SError exception was synchronized by the implicit error synchronization event and taken immediately.

This field is set to 1 when the SError exception was synchronized by an implicit error synchronization event and taken immediately on exception entry.

It this field is set to an IMPLEMENTATION SPECIFIC choice of 0 or 1 when the SError exception was synchronized by an implicit error synchronization event and taken immediately on an exception return.

For all other SError exceptions, this field is set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AET, bits [12:10]

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL2) == '010001':

Asynchronous Error Type.

Describes the PE error state after taking the SError exception.

AET	Meaning
0b000	Uncontainable (UC).
0b001	Unrecoverable state (UEU).
0b010	Restartable state (UEO).
0b011	Recoverable state (UER).
0b110	Corrected (CE).

All other values are reserved.

If multiple errors are taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA, bit [9]

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL2) == '010001':

External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

WnRV, bit [7]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL2) == '010001':

ESR_EL2.WnR valid.

WnRV	Meaning
`0b0`	ESR_EL2.WnR is not valid and has been set to 0.
`0b1`	ESR_EL2.WnR is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WnR, bit [6]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL2) == '010001':

Write-not-Read. When the WnRV field is 1, indicates whether an exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
`0b0`	Exception was caused by an instruction reading from a memory location.
`0b1`	Exception was caused by an instruction writing to a memory location.

Accessing this bit has the following behavior:

- This bit is RES0 if ESR_EL2.WnRV is 0.
- This bit is not valid and reads UNKNOWN if an External abort on a Atomic access, reported with ESR_EL2.WU == `0b00`.
- Otherwise RW.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DFSC, bits [5:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RAS is implemented:

Data Fault Status Code.

DFSC	Meaning
0b000000	Uncategorized error.
0b010001	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

24 … 6	5 … 0
RES0	IFSC

Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

For more information about generating these exceptions:

- For exceptions from AArch64, see ‘Breakpoint exceptions’.
- For exceptions from AArch32, see ‘Breakpoint exceptions’ and ‘Vector Catch exceptions’.

ISS encoding for an exception from a Software Step exception

24	23 … 7	6	5 … 0
ISV	RES0	EX	IFSC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

ISV	Meaning
`0b0`	EX bit is RES0.
`0b1`	EX bit is valid.

See the EX bit description for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

EX	Meaning
`0b0`	An instruction other than a Load-Exclusive instruction was stepped.
`0b1`	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning
`0b100010`	Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Software Step exception

For more information about generating these exceptions, see 'Software Step exceptions'.

ISS encoding for an exception from a Watchpoint exception

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

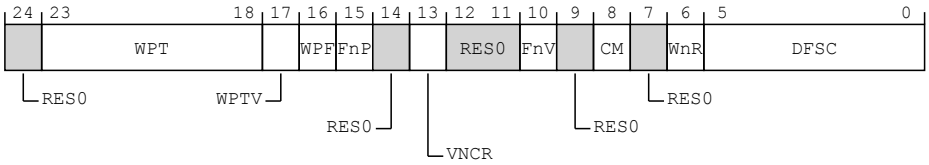

Bit [24]

Reserved, RES0.

WPT, bits [23:18]

When FEAT_Debugv8p2 is implemented:

Watchpoint number.

All other values are reserved.

Otherwise:

Reserved, RES0.

WPTV, bit [17]

When FEAT_Debugv8p2 is implemented:

Watchpoint number Valid.

WPTV	Meaning
0b0	The WPT field is invalid, and holds an UNKNOWN value.
0b1	The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint exception.

If FEAT_Debugv8p9 is implemented, value 0 is not permitted.

When a Watchpoint exception is triggered by a watchpoint match:

- If FEAT_Debugv8p9 is implemented or the PE sets any of FnV, FnP, or WPF to 1, then the PE sets WPTV to 1.
- Otherwise, the PE sets WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

Otherwise:

Reserved, RES0.

WPF, bit [16]

Watchpoint might be false-positive.

WPF	Meaning	Applies when
0b0	The watchpoint matched an address or address range that was accessed by the instruction.	
0b1	The watchpoint matched an address or address range that might not have been accessed by the instruction.	FEAT_SVE is implemented or FEAT_SME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm strongly recommends that this bit is set to 0, other than when one of the following instructions might generate a watchpoint match for an address or address range that the instruction does not access:

- An SVE contiguous vector load/store instruction, when the PE is in Streaming SVE mode.
- An SME load/store instruction.

FnP, bit [15]

FAR not Precise.

This field only has meaning if the FAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0.

FnP	Meaning	Applies when
`0b0`	If the FnV field is 0, the FAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.	
`0b1`	The FAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.	FEAT_SVE is implemented or FEAT_SME is implemented

Bit [14]

Reserved, RES0.

VNCR, bit [13]

Indicates that the watchpoint came from use of VNCR_EL2 register by EL1 code.

VNCR	Meaning	Applies when
`0b0`	The watchpoint was not generated by the use of VNCR_EL2 by EL1 code.	
`0b1`	The watchpoint was generated by the use of VNCR_EL2 by EL1 code.	FEAT_NV2 is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid.

FnV	Meaning	Applies when
`0b0`	The FAR is valid, and its value is as described by the FnP field.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FnV	Meaning	Applies when
0b1	The FAR is invalid, and holds an UNKNOWN value.	FEAT_SVE is implemented or FEAT_SME is implemented

Bit [9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance instruction:

CM	Meaning
0b0	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Watchpoint exception was generated by the execution of a cache maintenance instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as a cache maintenance instructions, and therefore their execution does not cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Watchpoint exception caused by an instruction reading from a memory location.
0b1	Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Watchpoint exception

For more information about generating these exceptions, see 'Watchpoint exceptions'.

ISS encoding for an exception from execution of a Breakpoint instruction

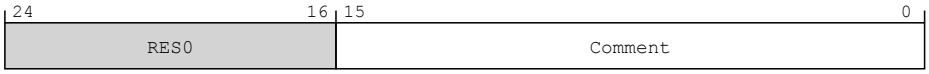

Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from execution of a Breakpoint instruction

For more information about generating these exceptions, see 'Breakpoint instruction exceptions'.

ISS encoding for an exception from an ERET, ERETAA, or ERETAB instruction

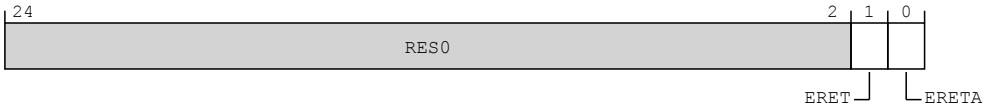

This EC value applies when:

- FEAT_FGT is implemented.
- The Effective value of HCR_EL2.NV is 1.

Bits [24:2]

Reserved, RES0.

ERET, bit [1]

Indicates whether an ERET or ERETA* instruction was trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ERET	Meaning
0b0	ERET instruction trapped to EL2.
0b1	ERETAA or ERETAB instruction trapped to EL2.

If this bit is 0, the ERETA field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ERETA, bit [0]

Indicates whether an ERETAA or ERETAB instruction was trapped to EL2.

ERETA	Meaning
0b0	ERETAA instruction trapped to EL2.
0b1	ERETAB instruction trapped to EL2.

When the ERET field is 0, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an ERET, ERETAA, or ERETAB instruction

For more information about generating these exceptions, see HCR_EL2.NV.

If FEAT_FGT is implemented, HFGITR_EL2.ERET controls fine-grained trap exceptions from ERET, ERETAA, and ERETAB execution.

ISS encoding for a Branch Target Exception

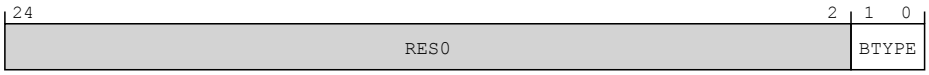

Bits [24:2]

Reserved, RES0.

BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

Additional Information for ISS encoding for a Branch Target Exception

For more information about generating these exceptions, see 'The AArch64 application level programmers model'.

ISS encoding for an exception from a trapped Pointer Authentication instruction

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from a trapped Pointer Authentication instruction

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

ISS encoding for a PAC Fail exception

24 … 2	1	0
RES0	DnI	BnA

Bits [24:2]

Reserved, RES0.

DnI, bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

DnI	Meaning
0b0	Instruction Key.
0b1	Data Key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BnA, bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

BnA	Meaning
0b0	A key.
0b1	B key.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for a PAC Fail exception

The following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- AUTDA, AUTDZA.
- AUTDB, AUTDZB.
- AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA.
- AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB.
- AUTIASPPC, AUTIASPPCR, AUTIA171615.
- AUTIBSPPC, AUTIBSPPCR, AUTIB171615.

If FEAT_FPACCOMBINE is implemented, the following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- RETAA, RETAB.
- RETAASPPC, RETABSPPC.
- RETAASPPCR, RETABSPPCR.
- BLRAA, BLRAAZ, BLRAB, BLRABZ.
- BRAA, BRAB, BRAAZ, BRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing ESR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name ESR_EL2 or ESR_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ESR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = ESR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = ESR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ESR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ESR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        ESR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    ESR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    ESR_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, ESR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ESR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x138);
    else
        X{64}(t) = ESR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = ESR_EL2();
    else
        X{64}(t) = ESR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ESR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ESR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ESR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x138) = X{64}(t);
    else
        ESR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        ESR_EL2() = X{64}(t);
    else
        ESR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ESR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.46 ESR_EL3, Exception Syndrome Register (EL3)

The ESR_EL3 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to ESR_EL3 are UNDEFINED.

Attributes

ESR_EL3 is a 64-bit register.

Field descriptions

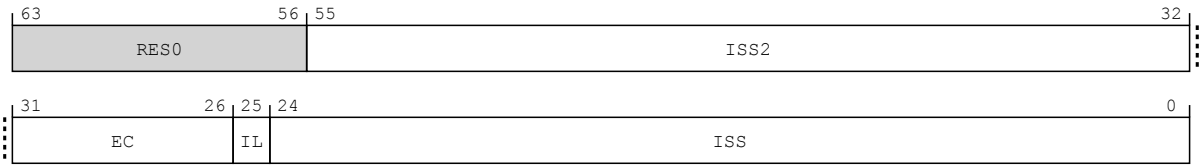

ESR_EL3 is made UNKNOWN as a result of an exception return from EL3.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL3, the value of ESR_EL3 is UNKNOWN. The value written to ESR_EL3 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:56]

Reserved, RES0.

ISS2, bits [55:32]

ISS2 encoding for an exception, the bit assignments are:

ISS2 encoding for an exception from a Data Abort

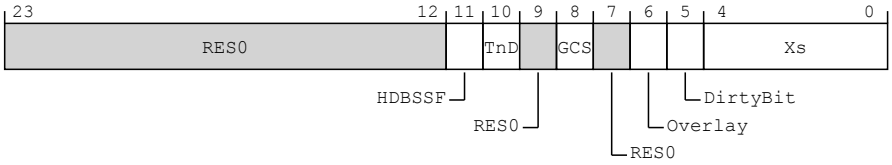

Bits [23:12]

Reserved, RES0.

HDBSSF, bit [11]

When FEAT_HDBSS is implemented, IsSecondStage(Fault), and GetESR_ELx_DFSC(EL3) IN {'0011xx', '01001x', '0101xx', '10001x', '1001xx'}:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates that the fault was caused by the HDBSS.

When DFSC indicates a synchronous External abort on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

HDBSSF	Meaning
0b0	Fault was not caused by HDBSS.
0b1	Fault was caused by HDBSS.

This field only applies for stage 2 Permission faults.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TnD, bit [10]

When FEAT_MTE_CANONICAL_TAGS is implemented:

Tag not Data.

If a memory access generates a Data Abort for a stage 1 Permission fault, this field indicates whether the fault is due to an Allocation Tag access.

TnD	Meaning
0b0	Permission fault is not due to a write of an Allocation Tag to Canonically Tagged memory.
0b1	Permission fault is due to a write of an Allocation Tag to Canonically Tagged memory.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [9]

Reserved, RES0.

GCS, bit [8]

When FEAT_GCS is implemented:

Guarded Control Stack data access.

If a memory access generates a Data Abort, this field indicates whether the fault is due to a Guarded Control Stack data access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GCS	Meaning
0b0	The Data Abort is not due to a Guarded control stack data access.
0b1	The Data Abort is due to a Guarded control stack data access. The ISV field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

Overlay, bit [6]

When FEAT_S1POE is implemented:

Overlay flag.

If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When FEAT_S1PIE is implemented:

DirtyBit flag.

If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For any other fault or Access, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Xs, bits [4:0]

When FEAT_LS64 is implemented:

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort exception reported with ISV=1, then this field holds register specifier, Xs.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort exception reported with ISV=1, then this field holds register specifier, Xs.

Otherwise, this field is RES0.

Otherwise:

Reserved, RES0.

ISS2 encoding for an exception from an Instruction Abort

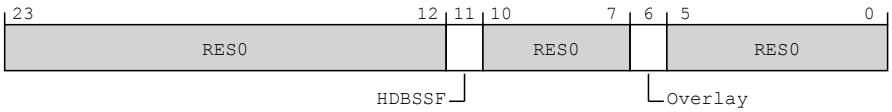

Bits [23:12]

Reserved, RES0.

HDBSSF, bit [11]

When FEAT_HDBSS is implemented:

Indicates that the fault was caused by the HDBSS.

When IFSC indicates a synchronous External abort on translation table walk or hardware update of translation table, this field indicates whether the fault was caused by a write to the HDBSS.

HDBSSF	Meaning
0b0	Fault was not caused by HDBSS.
0b1	Fault was caused by HDBSS.

This field only applies for stage 2 Permission faults.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [10:7]

Reserved, RES0.

Overlay, bit [6]

When FEAT_S1POE is implemented:

Overlay flag.

If a memory access generates a Instruction Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Instruction Abort is not due to Overlay Permissions.
0b1	Instruction Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:0]

Reserved, RES0.

ISS2 encoding for a Granule Protection Check exception

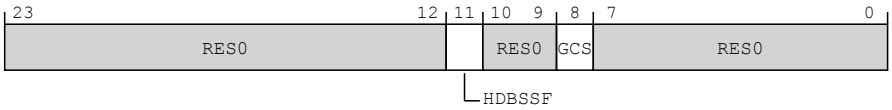

Bits [23:12]

Reserved, RES0.

HDBSSF, bit [11]

When FEAT_HDBSS is implemented and IsSecondStage(Fault):

Indicates that the fault was caused by the HDBSS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HDBSSF	Meaning
0b0	Fault was not caused by HDBSS.
0b1	Fault was caused by HDBSS.

This field only applies for stage 2 Permission faults.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [10:9]

Reserved, RES0.

GCS, bit [8]

When FEAT_GCS is implemented:

Guarded control stack data access.

Indicates that the Granule Protection Check Exception is due to a Guarded control stack data access.

GCS	Meaning
0b0	The Granule Protection Check Exception is not due to a Guarded control stack data access.
0b1	The Granule Protection Check Exception is due to a Guarded control stack data access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [7:0]

Reserved, RES0.

ISS2 encoding for all other exceptions

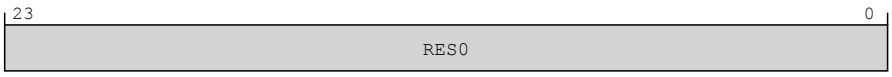

Bits [23:0]

Reserved, RES0.

EC, bits [31:26]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC	Meaning	ISS	ISS2	Applies when
`0b000000`	Unknown reason.	ISS encoding for exceptions with an unknown reason	ISS2 encoding for all other exceptions	
`0b000001`	Trapped WF* instruction execution. Conditional WF* instructions that fail their condition code check do not cause an exception.	ISS encoding for an exception from a WF* instruction	ISS2 encoding for all other exceptions	
`0b000011`	Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC value `0b000000`.	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000100`	Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC value `0b000000`.	ISS encoding for an exception from an MCRR or MRRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000101`	Trapped MCR or MRC access with (coproc==`0b1110`).	ISS encoding for an exception from an MCR or MRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000110`	Trapped LDC or STC access. The only architected uses of these instruction are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS encoding for an exception from an LDC or STC instruction	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
`0b000111`	Access to SME, SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000`.	ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps	ISS2 encoding for all other exceptions	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b001001	Trapped use of a Pointer authentication instruction.	ISS encoding for an exception from a trapped Pointer Authentication instruction	ISS2 encoding for all other exceptions	FEAT_PAuth is implemented
0b001010	Trapped execution of any instruction not covered by other EC values.	ISS encoding for an exception from any other instruction	ISS2 encoding for all other exceptions	FEAT_LS64 is implemented
0b001100	Trapped MRRC access with (coproc==0b1110).	ISS encoding for an exception from an MCRR or MRRC access	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b001101	Branch Target Exception.	ISS encoding for a Branch Target Exception	ISS2 encoding for all other exceptions	FEAT_BTI is implemented
0b001110	Illegal Execution state.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b010011	SMC instruction execution in AArch32 state, when SMC is not disabled.	ISS encoding for an exception from SMC instruction execution in AArch32 state	ISS2 encoding for all other exceptions	FEAT_AA32 is implemented
0b010100	Trapped MSRR, MRRS or System instruction execution in AArch64 state, that is not reported using EC value 0b000000.	ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_SYSREG128 is implemented or FEAT_SYSINSTR128 is implemented
0b010101	SVC instruction execution in AArch64 state.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b010110	HVC instruction execution in AArch64 state, when HVC is not disabled.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b010111	SMC instruction execution in AArch64 state, when SMC is not disabled.	ISS encoding for an exception from SMC instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC values 0b000000, 0b000001 or 0b000111. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview', except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b011001	Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC value 0b000000.	ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ	ISS2 encoding for all other exceptions	FEAT_SVE is implemented
0b011100	Exception from a PAC Fail	ISS encoding for a PAC Fail exception	ISS2 encoding for all other exceptions	FEAT_FPAC is implemented
0b011101	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, CPTR_EL3.ESM, or an attempted execution of an instruction that is illegal because of the value of PSTATE.SM or PSTATE.ZA, that is not reported using EC value 0b000000.	ISS encoding for an exception due to SME functionality	ISS2 encoding for all other exceptions	FEAT_SME is implemented
0b011110	Granule Protection Check exception	ISS encoding for a Granule Protection Check exception	ISS2 encoding for a Granule Protection Check exception	FEAT_RME is implemented
0b011111	IMPLEMENTATION DEFINED exception to EL3.	ISS encoding for an IMPLEMENTATION DEFINED exception to EL3	ISS2 encoding for all other exceptions	
0b100000	Instruction Abort from a lower Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b100001	Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for all other exceptions	
0b100010	PC alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b100100	Data Abort exception from a lower Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100101	Data Abort exception taken without a change in Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100110	SP alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b100111	Memory Operation Exception.	ISS encoding for an exception from the Memory Copy and Memory Set instructions	ISS2 encoding for all other exceptions	FEAT_MOPS is implemented
0b101100	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b101101	GCS exception.	ISS encoding for a GCS exception	ISS2 encoding for all other exceptions	FEAT_GCS is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b101111	SError exception.	ISS encoding for an SError exception	ISS2 encoding for all other exceptions	
0b111100	BRK instruction execution in AArch64 state. This is reported in ESR_EL3 only if a BRK instruction is executed at EL3. This is the only debug exception that can be taken to EL3 when EL3 is using AArch64.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions	FEAT_AA64 is implemented
0b111101	Profiling exception	ISS encoding for a Profiling exception	ISS2 encoding for all other exceptions	FEAT_EBEP is implemented, or FEAT_SPE_EXC is implemented, or FEAT_TRBE_EXC is implemented

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

IL	Meaning
0b0	16-bit instruction trapped.
0b1	32-bit instruction trapped. This value is also used when the exception is one of the following: • An SError exception. • An Instruction Abort exception. • A PC alignment fault exception. • An SP alignment fault exception. • A Data Abort exception for which the value of the ISV bit is 0. • An Illegal Execution state exception. • Any debug exception except for Breakpoint instruction exceptions. • An exception reported using EC value 0b000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see 'Mapping of the general-purpose registers between the Execution states'.

If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value `0b11111`.

ISS encoding for exceptions with an unknown reason

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for exceptions with an unknown reason

When an exception is reported using this EC value, the IL field is set to 1.

This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Attempted execution of an MSR or MRS instruction using a _EL12 register name when the Effective value of HCR_EL2.E2H is not 1.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, and EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1. If the value of HCR_EL2.TGE is 0, this exception is reported using an ESR_EL3.EC value of `0b000111`.
- If FEAT_UINJ is implemented, an Undefined exception caused by attempting to execute any instruction when PSTATE.UINJ is 1.

ISS encoding for an exception from a WF* instruction

24	23 20	19 10	9 5	4 3	2	1 0
CV	COND	RES0	RN	RES0	RV	TI

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:10]

Reserved, RES0.

RN, bits [9:5]

When FEAT_WFxT is implemented:

Register Number. Indicates the register number supplied for a `WFET` or `WFIT` instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.

RV, bit [2]

When FEAT_WFxT is implemented:

Register field Valid.

If TI[1] == 1, then this field indicates whether RN holds a valid register number for the register argument to the trapped `WFET` or `WFIT` instruction.

RV	Meaning
`0b0`	Register field invalid.
`0b1`	Register field valid.

If TI[1] == 0, then this field is RES0.

This field is set to 1 on a trap on `WFET` or `WFIT`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

TI	Meaning	Applies when
0b00	WFI trapped.	
0b01	WFE trapped.	
0b10	WFIT trapped.	FEAT_WFxT is implemented
0b11	WFET trapped.	FEAT_WFxT is implemented

When FEAT_WFxT is implemented, this is a two bit field as shown. Otherwise, bit[1] is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a WF* instruction

The following fields describe configuration settings for generating this exception:

- HCR.{TWE, TWI}.
- SCTLR_EL1.{nTWE, nTWI}.
- SCTLR_EL2.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

ISS encoding for an exception from an MCR or MRC access

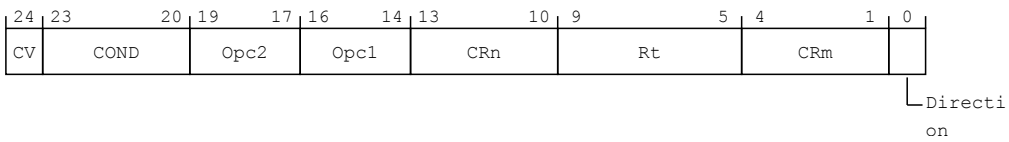

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

If the Rt value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See 'Mapping of the general-purpose registers between the Execution states'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write to System register space. MCR instruction.
`0b1`	Read from System register space. MRC or VMRS instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an MCR or MRC access

The following fields describe configuration settings for generating exceptions from an MCR or MRC access using coproc `0b1111`, that are reported using EC value `0b000011`:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, trapped to EL1.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors System registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- HCR.{TRVM, TVM} and HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, trapped to EL2.
- HCR.TTLB and HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, trapped to EL2.
- HCR.{TSW, TPC, TPU} and HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, trapped to EL2.
- HCR.TAC and HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, trapped to EL2.
- HCR.TIDCP and HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, trapped to EL2.
- If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- HCR.{TID1, TID2, TID3} and HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to error record registers at EL1 using AArch32 state, trapped to EL2.
- HCPTR.TCPAC and CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, trapped to EL2.
- HSTR.T<n> and HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, trapped to EL2.
- CNTHCTL.PL1PCEN and CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HDCR.TTRF, for Non-secure accesses to trace filter control registers from system registers using AArch32 state, trapped to EL2.
- HDCR.{TPM, TPMCR} and MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HCPTR.TAM and CPTR_EL2.TAM, for accesses to Activity Monitors System registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions from an MCR or MRC access using coproc `0b1110`, that are reported using EC value `0b000101`:

- CPACR_EL1.TTA for accesses to trace System registers, trapped to EL2 or EL1.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, trapped to EL2 or EL1.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC System registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC System registers at EL0, EL1, and EL2 trapped to EL3.
- HCR.TID0 and HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, trapped to EL2.
- HCPTR.TTA and CPTR_EL2.TTA, for accesses to trace System registers using AArch32, trapped to EL2.
- HDCR.TDRA and MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, trapped to EL2.
- HDCR.TDOSA and MDCR_EL2.TDOSA, for accesses to powerdown debug System registers, using AArch32 state, trapped to EL2.
- HDCR.TDA and MDCR_EL2.TDA, for accesses to other debug System registers, using AArch32 state, trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace System registers using AArch32, trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug System registers using AArch32, trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug System registers, using AArch32, trapped to EL3.

The following fields describe configuration settings for generating exceptions from a VMSR or VMRS access, that are reported using EC value `0b001000`:

- HCR.TID0 and HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR.TID3 and HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.
- HCPTR.{TCP10, TCP11}, for Non-secure accesses to FPSCR, FPSID, FPEXC, MVFR0, MVFR1, and MVFR2, trapped to EL2.

ISS encoding for an exception from any other instruction

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

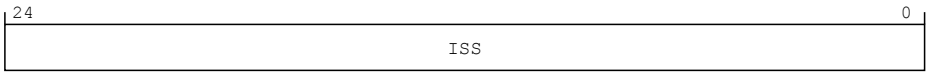

ISS, bits [24:0]

ISS	Meaning	Applies when
`0b0000000000000000000000000`	`ST64BV` instruction trapped.	FEAT_LS64_V is implemented
`0b0000000000000000000000001`	`ST64BV0` instruction trapped.	FEAT_LS64_ACCDATA is implemented
`0b0000000000000000000000010`	`LD64B` or `ST64B` instruction trapped.	FEAT_LS64 is implemented

All other values are reserved.

ISS encoding for an exception from an MCRR or MRRC access

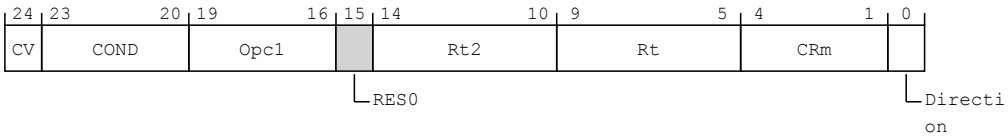

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

If the Rt2 value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt2 value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See ‘Mapping of the general-purpose registers between the Execution states’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

If the Rt value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See 'Mapping of the general-purpose registers between the Execution states'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write to System register space. MCRR instruction.
`0b1`	Read from System register space. MRRC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an MCRR or MRRC access

The following fields describe configuration settings for generating exceptions from an MCRR or MRRC access using coproc `0b1111`, that are reported using EC value `0b000100`:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, trapped to EL2 or EL1.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors System registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, trapped to EL2 or EL1.
- HCR.{TRVM, TVM} and HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to error record registers at EL1 using AArch32 state, trapped to EL2.
- HSTR.T<n> and HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, trapped to EL2.
- CNTHCTL.{PL1PCEN, PL1PCTEN} and CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HDCR.TPM and MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, trapped to EL2.
- HCPTR.TAM and CPTR_EL2.TAM, for accesses to Activity Monitors System registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers from EL0, EL1 and EL2 using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following fields describe configuration settings for generating exceptions from an MCRR or MRRC access using coproc `0b1110`, that are reported using EC value `0b001100`:

- MDSCR_EL1.TDCC, for accesses to the Debug ROM registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, trapped to EL2 or EL1.
- HDCR.TDRA and MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, trapped to EL2.
- MDCR_EL3.TDA, for accesses to debug System registers, using AArch32, trapped to EL3.
- CPACR_EL1.TTA for accesses to trace System registers using AArch32, trapped to EL1 or EL2.
- HCPTR.TTA and CPTR_EL2.TTA, for accesses to trace System registers using AArch32, trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace System registers using AArch32, trapped to EL3.

Note

If FEAT_ETMv4 or FEAT_ETE are implemented, MCRR and MRRC accesses to trace System registers are UNDEFINED and the resulting exception is higher priority than an exception due to these traps.

ISS encoding for an exception from an LDC or STC instruction

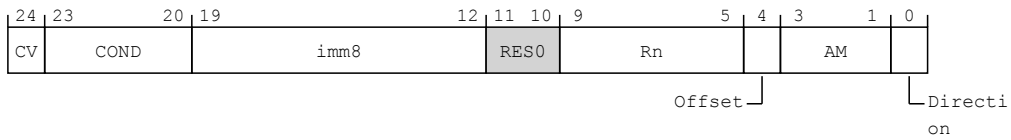

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer.

If the Rn value is not `0b1111`, then the reported value gives the AArch64 view of the register. Otherwise, if the Rn value is `0b1111`:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value `0b11111`.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value `0b11111`.

See 'Mapping of the general-purpose registers between the Execution states'.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Offset	Meaning
`0b0`	Subtract offset.
`0b1`	Add offset.

This bit corresponds to the U bit in the instruction encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

AM	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in 'Reserved values in System and memory-mapped registers and translation table entries'.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from an LDC or STC instruction

The following fields describe the configuration settings from an LDC or STC access for the traps that are reported using EC value 0b000110:

- MDSCR_EL1.TDCC, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL2 or EL1.
- HDCR.TDA and MDCR_EL2.TDA, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL2.
- MDCR_EL3.TDA, for accesses to DBGDTRTXint and DBGDTRRXint, using AArch32 state, trapped to EL3.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC System registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC System registers at EL0, EL1, and EL2 trapped to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

24	23 20	19 0
CV	COND	RES0

The accesses covered by this trap include:

- Execution of Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.
- Execution of SVE instructions.
- Execution of SME instructions.

For an implementation that does not include either SVE or support for Advanced SIMD and floating-point, the exception is reported using the EC value `0b000000`.

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

The following fields describe the configuration settings for the traps that are reported using EC value `0b000111`:

- HCPTR.{TCP10, TCP11}, for Non-secure accesses to Advanced SIMD and floating-point registers and instructions, trapped to EL2.
- HCPTR.TASE, for Non-secure accesses to Advanced SIMD functionality, trapped to EL2.
- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL3.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

24 … 0
RES0

The accesses covered by this trap include:

- Execution of SVE instructions when the PE is not in Streaming SVE mode.
- Accesses to the SVE System registers, ZCR_ELx.

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

The following fields describe the configuration settings for the traps that are reported using EC value `0b011001`:

- CPACR_EL1.ZEN, for execution of SVE instructions and accesses to SVE registers at EL1 or EL0, trapped to EL1.
- CPTR_EL2.ZEN and CPTR_EL2.TZ, for execution of SVE instructions and accesses to SVE registers at EL0, EL1, or EL2, trapped to EL2.
- CPTR_EL3.EZ, for execution of SVE instructions and accesses to SVE registers from all Exception levels, trapped to EL3.

ISS encoding for a Profiling exception

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

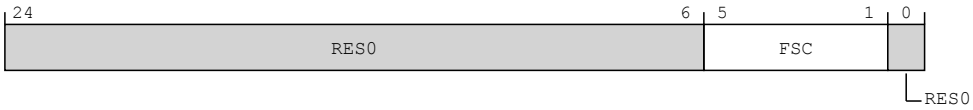

Bits [24:6]

Reserved, RES0.

FSC, bits [5:1]

Indicates why the Profiling exception was generated.

FSC	Meaning	Applies when
0b00000	PMU Profiling exception. The exception was generated because at least one PMU counter overflow status flag was 1.	FEAT_EBEP is implemented
0b00001	Profiling Buffer management event. The exception was generated because PMBSR_EL3.S was 1.	FEAT_SPE_EXC is implemented
0b00010	Trace buffer management event. The exception was generated because TRBSR_EL3.IRQ was 1.	FEAT_TRBE_EXC is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

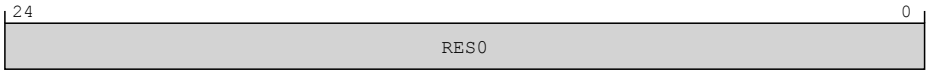

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see 'PC alignment checking'.

'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from the Memory Copy and Memory Set instructions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

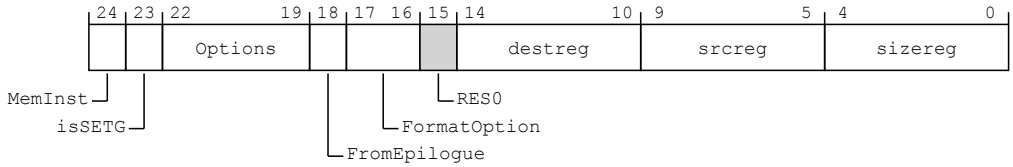

MemInst, bit [24]

Indicates the memory instruction class causing the exception.

MemInst	Meaning
0b0	CPYFE*, CPYFM*, CPYE*, and CPYM* instructions.
0b1	The instruction that caused the exception is in one of the following instruction classes: • SETE*, SETM*. • SETGE*, SETGM*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

isSETG, bit [23]

Indicates whether the instruction belongs to one of the following instruction classes:

- SETGE*, SETGM*.

isSETG	Meaning
0b0	The instruction that caused the exception is not in one of the specified classes.
0b1	The instruction that caused the exception is in one of the specified classes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Options, bits [22:19]

Options : the Options field of the instruction.

For Memory Copy instructions, bits[22:19] forms the Options field, which holds the bits[15:12] of the instruction.

For Memory Set instructions:

- Bits[22:21] are RES0.
- Bits[20:19] form the Options field, which holds the bits[13:12] of the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FromEpilogue, bit [18]

Indicates whether the instruction belongs to one of the following epilogue classes of Memory Copy or Memory Set instructions:

- CPYE*, CPYFE*.
- SETE*, SETGE*.

FromEpilogue	Meaning
`0b0`	The instruction that caused the exception is not in one of the specified classes.
`0b1`	The instruction that caused the exception is in one of the specified classes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FormatOption, bits [17:16]

Reports the Option used to encode the initial Xs, Xd, and Xn register values provided to the instruction that generated the exception.

FormatOption	Meaning
`0b00`	Option B.
`0b01`	Option A.
`0b10`	Option A.
`0b11`	Option B.

For more information, see Memory Copy and Memory Set exceptions.

Note

This field was previously presented as two separate bits, WrongOption, bit[17] and OptionA, bit [16], which were already expected to be used together and not individually.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

destreg, bits [14:10]

The destination register value from the issued instruction, containing the destination address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

srcreg, bits [9:5]

The source register value from the issued instruction, containing either the source address or the source data.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

sizereg, bits [4:0]

The size register value from the issued instruction, containing the number of bytes to be transferred or set.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from HVC or SVC instruction execution

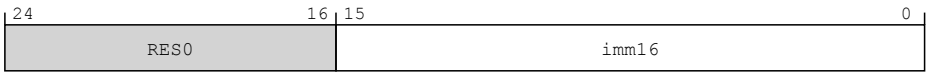

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from HVC or SVC instruction execution

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see 'SVC' and 'HVC'.

For A64 instructions, see 'SVC' and 'HVC'.

If FEAT_FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

ISS encoding for an exception from SMC instruction execution in AArch32 state

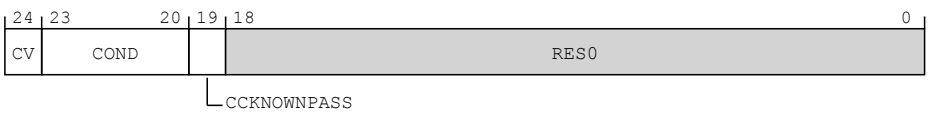

For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCKNOWNPASS	Meaning
0b0	The instruction was unconditional, or was conditional and passed its condition code check.
0b1	The instruction was conditional, and might have failed its condition code check.

Note

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [18:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from SMC instruction execution in AArch32 state

HCR.TSC describes the configuration settings for trapping SMC instructions to EL2.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

ISS encoding for an exception from SMC instruction execution in AArch64 state

24 … 16	15 … 0
RES0	imm16

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from SMC instruction execution in AArch64 state

The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

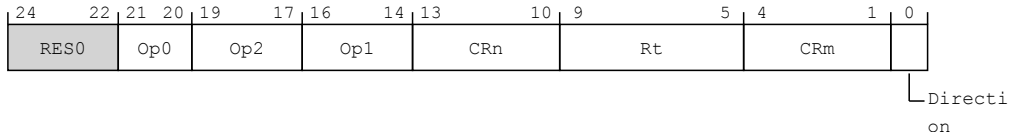

Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

For system instructions which require that the opcode Rt field is set to `0b11111`, but where the trapped instruction has a different value of Rt, an implementation is permitted to return the value `0b11111`, instead of the value of Rt from the trapped instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Direction	Meaning
0b0	Write access, including MSR instructions.
0b1	Read access, including MRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

For exceptions caused by System instructions, see 'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL1.
- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2 or EL1.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2 or EL1.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- CPACR_EL1.TTA, for accesses to the trace System registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- PMUSERENR_EL0, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors System registers using AArch64 state, MSR or MRS access trapped to EL2 or EL1.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, execution is trapped to EL2.
- If FEAT_EVT is implemented, the following registers control traps for EL1 and EL0 registers and instructions that use this EC value:
  - HCR_EL2.{TICAB, TOCU, TID4}.
  - HCR2.{TICAB, TOCU, TID4}.
- If FEAT_EVT2 is implemented, the following registers control traps for EL1 and EL0 instructions that use this EC value:
  - HCR_EL2.{TTLBOS, TTLBIS}.
- If FEAT_EVT2 is implemented, HCR2.TTLBIS controls traps for EL1 and EL0 instructions using AArch32 state that use this EC value.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- If FEAT_MTE2 is implemented, HCR_EL2.TID5, for accesses to GMID_EL1, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_IDTE3 is implemented, SCR_EL3.TID3, for accesses to ID group 3 registers using AArch64 state, at EL1 and EL2, trapped to EL3.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace System registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug System registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug System registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors System registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the trace filter control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug System registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- If FEAT_SPE is implemented:
  - MDCR_EL3.NSPB for accesses to Statistical Profiling and Profiling Buffer control registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
  - MDCR_EL2.TPMS for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 trapped to EL2.
- CPTR_EL3.TAM, for accesses to Activity Monitors System registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If FEAT_FGT is implemented:
  - SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - HFGITR_EL2 for execution of system instructions trapped to EL2.
  - HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  - HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.
- If FEAT_RNG_TRAP is implemented, SCR_EL3.TRNDR for reads of RNDR and RNDRRS using AArch64 state, MRS access trapped to EL3.
- If FEAT_SME is implemented:
  - CPTR_EL3.ESM, for MSR or MRS accesses to SMPRI_EL1 at EL1, EL2, and EL3, trapped to EL3.
  - CPTR_EL3.ESM, for MSR or MRS accesses to SMPRIMAP_EL2 at EL2 and EL3, trapped to EL3.
  - SCTLR_EL1.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2 or EL1.
  - SCTLR_EL2.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2.
  - SCR_EL3.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, EL1, and EL2, trapped to EL3.
- If FEAT_FPMR is implemented:
  - SCTLR_EL1.EnFPM, for accesses to FPMR at EL0, trapped to EL2 or EL1.
  - SCTLR_EL2.EnFPM, for accesses to FPMR at EL0, trapped to EL2.
  - HCRX_EL2.EnFPM, for accesses to FPMR at EL0 and EL1, trapped to EL2.
  - SCR_EL3.EnFPM, for accesses to FPMR at EL0, EL1, and EL2, trapped to EL3.
- If FEAT_NMI is implemented, HCRX_EL2.TALLINT, for MSR writes of ALLINT at EL1, trapped to EL2.
- If FEAT_FGT2 is implemented:
  - SCR_EL3.FGTEn2, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  - HFGRTR2_EL2 for reads and HFGWTR2_EL2 for writes of registers, using AArch64 state, using MSR or MRS access at EL1 trapped to EL2.
  - HDFGRTR2_EL2 for reads and HDFGWTR2_EL2 for writes of registers, using AArch64 state, using MSR or MRS access at EL0 and EL1 trapped to EL2.
  - HFGITR2_EL2 for execution of system instructions trapped to EL2.
- If FEAT_ITE is implemented, MDCR_EL3.EnITE, for accesses to Instrumentation trace System registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If FEAT_MEC is implemented, SCR_EL3.MECEn, for accesses to MECID registers at EL2, trapped to EL3.
- If FEAT_SPE_FDS is implemented, MDCR_EL3.EnPMS3 for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
- If FEAT_SPE_nVM is implemented, MDCR_EL3.EnPMS4 for accesses to SPE registers, using AArch64 state, MSR or MRS access at EL1 and EL2 trapped to EL3.
- If FEAT_RASv2 is implemented, SCR_EL3.TWERR, for accesses to Error Record registers, MSR access at EL1 and EL2 trapped to EL3.
- If FEAT_Debugv8p9 is implemented, MDCR_EL3.EBWE for accesses of MDSELR_EL1, using AArch64 state, MRS or MSR access at EL2 and EL1 trapped to EL3.
- If FEAT_PMUv3p9, FEAT_SPMU, FEAT_EBEP, or FEAT_PMUv3_SS is implemented, MDCR_EL3.EnPM2, for accesses to PMU registers, using AArch64 state, MSR or MRS access at EL2, EL1, and EL0, trapped to EL3.
- If FEAT_PMUv3_SS is implemented, MDCR_EL3.EnPMSS, for accesses to PMU Snapshot registers, using AArch64 state, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_THE is implemented, SCR_EL3.RCWMASKEn for accesses to RCWMASK_EL1 and RCWSMASK_EL1, using AArch64 state, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_AIE is implemented, SCR_EL3.AIEn for accesses to Extended Memory Attribute registers, MSR or MRS access at EL2 and EL1 trapped to EL3.
- If FEAT_S1PIE, FEAT_S2PIE, FEAT_S1POE, or FEAT_S2POE is implemented, SCR_EL3.PIEn for accesses to Permission Indirection, Overlay registers, MSR or MRS access at EL2, EL1 and EL0 trapped to EL3.
- If FEAT_MPAM_PE_BW_CTRL is implemented, MPAMBW2_EL2.{nTRAP_MPAMBWIDR_EL1, nTRAP_MPAMBW0_EL1, nTRAP_MPAMBW1_EL1} for accesses to MPAMBWn_EL1 registers, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_MPAM_PE_BW_CTRL and FEAT_SME are implemented, MPAMBW2_EL2.nTRAP_MPAMBWSM_EL1, for accesses to MPAMBWSM_EL1, using AArch64 state, MRS or MSR access at EL1, trapped to EL2.
- If FEAT_MPAM_PE_BW_CTRL is implemented, MPAMBW3_EL3.nTRAPLOWER, for accesses

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

to MPAMBW2_EL2 and MPAMBWn_EL1 registers, using AArch64 state, MRS or MSR access at EL1, trapped to EL3.
- If FEAT_HACDBS is implemented, SCR_EL3.HACDBSEn, for accesses to HACDBSBR_EL2 and HACDBSCONS_EL2, using AArch64 state, MRS or MSR access at EL2, trapped to EL3.
- If FEAT_HDBSS is implemented, SCR_EL3.HDBSSEn, for accesses to HDBSSBR_EL2 and HDBSSPROD_EL2, using AArch64 state, MRS or MSR access at EL2, trapped to EL3.
- If FEAT_SRMASK is implemented, SCR_EL3.SRMASKEn, for MSR or MRS accesses at EL2 or EL1 to the MASK registers using AArch64 state, trapped to EL3.

ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

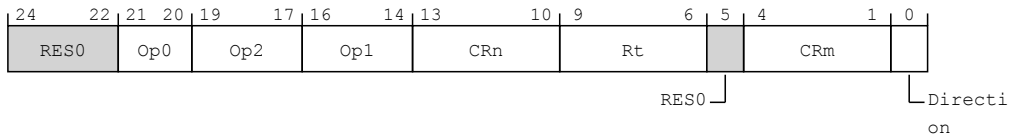

Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:6]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

Note

This value represents register pair of X[Rt:0], X[Rt:1].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [5]

Reserved, RES0.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write access, MSRR instructions.
`0b1`	Read access, MRRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

The following fields describe configuration settings for generating exceptions from an MSRR or MRRS access that are reported using EC value `0b010100`:

- If FEAT_FGT is implemented:
  - HFGRTR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, accesses at EL1 trapped to EL2.
- If FEAT_FGT2 is implemented:
  - HFGRTR2_EL2.nRCWSMASK_EL1 for reads and HFGWTR2_EL2.nRCWSMASK_EL1 for writes of RCWSMASK_EL1, using AArch64 state, accesses at EL1 trapped to EL2.
- If FEAT_SYSREG128 is implemented:
  - SCTLR2_EL1.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL0 trapped to EL1.
  - SCTLR2_EL2.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL0 trapped to EL2.
  - HCRX_EL2.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL1 and EL0 trapped to EL2.
  - SCR_EL3.EnIDCP128 for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL2, EL1, and EL0 trapped to EL3.
- If FEAT_D128 is implemented:
  - HCR_EL2.{TRVM, TVM} for accesses to TTBR0_EL1 and TTBR1_EL1, accesses at EL1 and EL0 trapped to EL2.
  - HCRX_EL2.D128En for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL1 trapped to EL2.
  - HFGITR_EL2 for execution of TLBIP system instructions trapped to EL2.
  - HCR_EL2.{TTLB, TTLBOS, TTLBIS}, for execution of TLBIP instructions using AArch64 state, execution is trapped to EL2.
  - SCR_EL3.D128En for accesses to 128-bit IMPLEMENTATION DEFINED System registers, accesses at EL2 and EL1 trapped to EL3.
- If FEAT_THE is implemented, SCR_EL3.RCWMASKEn for accesses to RCWMASK_EL1 and RCWSMASK_EL1, using AArch64 state, accesses at EL2 and EL1 trapped to EL3.

ISS encoding for an IMPLEMENTATION DEFINED exception to EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

24 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [24:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from an Instruction Abort

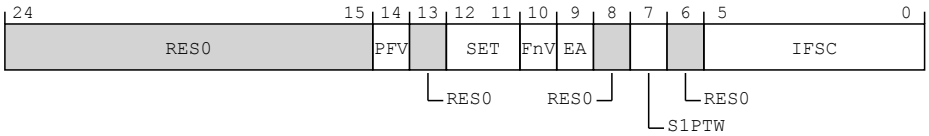

The ISS2 encoding for an exception from an Instruction Abort includes further information about the exception if any of the following are true:

- FEAT_S1POE is implemented and a memory access generates a Instruction Abort due to a Permission fault.
- FEAT_HDBSS is implemented.
- FEAT_THE is implemented.

Bits [24:15]

Reserved, RES0.

PFV, bit [14]

When FEAT_PFAR is implemented and GetESR_ELx_IFSC(EL3) IN {'010000', '01001x', '0101xx'}:

PFAR Valid. Describes whether the MFAR_EL3 register is valid.

PFV	Meaning
0b0	MFAR_EL3 is UNKNOWN.
0b1	MFAR_EL3 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [13]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET, bits [12:11]

When FEAT_RAS is implemented and GetESR_ELx_IFSC(EL3) IN {'010000', '01001x', '0101xx'}:

Synchronous Error Type. Describes the PE error state after taking the Instruction Abort exception.

SET	Meaning	Applies when
`0b00`	Recoverable state (UER).	
`0b10`	Uncontainable (UC).	FEAT_RASv2 is not implemented
`0b11`	Restartable state (UEO).	

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
`0b0`	FAR is valid.
`0b1`	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is `0b010000`. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception due to SME functionality

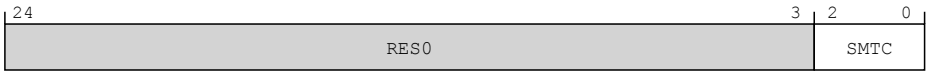

The accesses covered by this trap include:

- Execution of SME instructions.
- Execution of SVE and Advanced SIMD instructions, when the PE is in Streaming SVE mode.
- Direct accesses of SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3.

Bits [24:3]

Reserved, RES0.

SMTC, bits [2:0]

SME Trap Code. Identifies the reason for instruction trapping.

SMTC	Meaning	Applies when
0b000	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC value 0b000000.	
0b001	Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1.	
0b010	SME instruction trapped because PSTATE.SM is 0.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SMTC	Meaning	Applies when
0b011	SME instruction trapped because PSTATE.ZA is 0.	
0b100	Access to the SME2 ZT0 register trapped as a result of SMCR_EL1.EZT0, SMCR_EL2.EZT0, or SMCR_EL3.EZT0.	FEAT_SME2 is implemented

All other values are reserved.

Additional Information for ISS encoding for an exception due to SME functionality

The following fields describe the configuration settings for the traps that are reported using the EC value 0b011101:

- CPACR_EL1.SMEN, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR and SMCR_EL1 System registers at EL1 and EL0, trapped to EL2 or EL1.
- CPTR_EL2.SMEN and CPTR_EL2.TSM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2 at EL2, EL1, and EL0, trapped to EL2.
- CPTR_EL3.ESM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3 from all Exception levels and any Security state, trapped to EL3.
- If FEAT_SME2 is implemented:
  - SMCR_EL1.EZT0, for accesses to ZT0 at EL1 and EL0, trapped to EL2 or EL1.
  - SMCR_EL2.EZT0, for accesses to ZT0 at EL2, EL1, and EL0, trapped to EL2.
  - SMCR_EL3.EZT0, for accesses to ZT0 at any Exception level, trapped to EL3.

ISS encoding for a Granule Protection Check exception

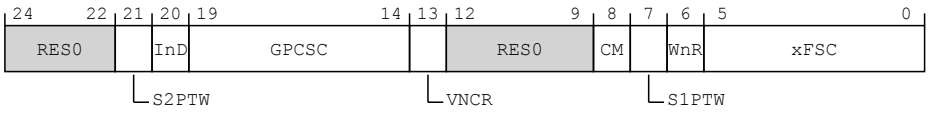

Bits [24:22]

Reserved, RES0.

S2PTW, bit [21]

Indicates whether the Granule Protection Check exception was on an access made for a stage 2 translation table walk.

S2PTW	Meaning
0b0	Fault not on a stage 2 translation table walk.
0b1	Fault on a stage 2 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

InD, bit [20]

Indicates whether the Granule Protection Check exception was on an instruction or data access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

InD	Meaning
0b0	Data access.
0b1	Instruction access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GPCSC, bits [19:14]

Granule Protection Check Status Code.

GPCSC	Meaning
0b000000	GPT address size fault at level 0.
0b000100	GPT walk fault at level 0.
0b000101	GPT walk fault at level 1.
0b001100	Granule protection fault at level 0.
0b001101	Granule protection fault at level 1.
0b010100	Synchronous External abort on GPT fetch at level 0.
0b010101	Synchronous External abort on GPT fetch at level 1.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VNCR, bit [13]

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

VNCR	Meaning	Applies when
0b0	The fault was not generated by the use of VNCR_EL2 by EL1 code.	
0b1	The fault was generated by the use of VNCR_EL2 by EL1 code.	FEAT_NV2 is implemented

When InD is 1, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CM	Meaning
0b0	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

Indicates whether the Granule Protection Check exception was on an access for stage 2 translation for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Abort caused by an instruction reading from a memory location.
0b1	Abort caused by an instruction writing to a memory location.

When InD is 1, this field is RES0.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101, indicating an unsupported Exclusive or atomic access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A fault reported using a DFSC value of `0b110001`, indicating an unsupported atomic hardware update fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

xFSC, bits [5:0]

Instruction or Data Fault Status Code.

xFSC	Meaning	Applies when
`0b100011`	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
`0b100100`	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
`0b100101`	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
`0b100110`	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
`0b100111`	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
`0b101000`	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Data Abort

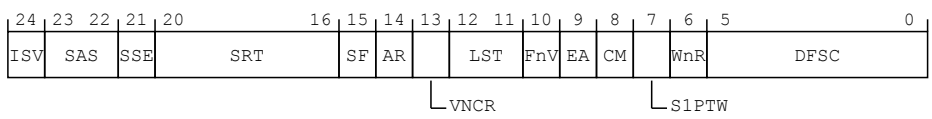

The ISS2 encoding for an exception from a Data Abort includes further information about the exception when any of the following features are implemented:

- FEAT_LS64_V.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_LS64_ACCDATA.
- FEAT_S1POE or FEAT_S2POE.
- FEAT_S1PIE or FEAT_S2PIE.
- FEAT_GCS.
- FEAT_MTE_CANONICAL_TAGS.
- FEAT_HDBSS.
- FEAT_MTE_PERM.

ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] represents the instruction syndrome fields {SAS, SSE, SRT, SF, AR}.

ISV	Meaning
`0b0`	No valid instruction syndrome. ISS[23:14] do not represent an instruction syndrome.
`0b1`	ISS[23:14] hold a valid instruction syndrome.

The ISV field is only relevant for the following instructions:

- An AArch64 load or store of a single general-purpose register (including the register specified with `0b11111`, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).
- An AArch32 instruction LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT, where both of the following apply:
  - It is not performing register writeback.
  - It is not using R15 as a source or destination register.
- If FEAT_LS64 is implemented, an LD64B or ST64B instruction.
- If FEAT_LS64_V is implemented, an ST64BV instruction.
- If FEAT_LS64_ACCDATA is implemented, an ST64BV0 instruction.

The ISV field is 1 if all of the following apply:

- The fault was not a stage 2 fault on a stage 1 translation table walk.
- If FEAT_RAS is implemented, the fault was not any kind of synchronous External abort.
- Any of the following apply:
  - The fault was generated by one of the described AArch64 load or store instructions that transfer a single general-purpose register, and any of the following apply:
    - The fault was a stage 2 fault.
  - The fault was generated by one of the described AArch32 instructions and any of the following apply:
    - The fault was a stage 2 fault.
  - The fault was generated by one of the FEAT_LS64, FEAT_LS64_V or FEAT_LS64_ACCDATA instructions and the fault was a Translation fault, Access flag fault or Permission fault.
- One of the following applies:
  - The address reported in FAR_EL3 is the lowest address accessed by the instruction.
  - The address reported in FAR_EL3 is not the lowest address accessed by the instruction, and the implementation reports ISV as 1 in this case. Arm recommends against this behavior.

Otherwise the value of this field is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The value of this field is 0 if any of the following apply:

- FEAT_MOPS is implemented and the fault was for a memory access generated by a Memory Copy and Memory Set instruction.
- FEAT_MTE is implemented and the fault was for a direct access to Allocation Tags.
- FEAT_GCS is implemented and the fault was for a Guarded control stack data access.
- FEAT_NV2 is implemented and the fault was for a system register access that was transformed to a load or store relative to VNCR_EL2.
- FEAT_LSE is implemented and the fault was for an atomic instruction.

The value of this field is UNKNOWN when the following occurs:

- Stage 2 faults when the exception was generated in Debug state, in memory access mode.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

When ISV == '1':

Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

SAS	Meaning
`0b00`	Byte
`0b01`	Halfword
`0b10`	Word
`0b11`	Doubleword

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is `0b11`.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSE, bit [21]

When ISV == '1':

Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SSE	Meaning
0b0	Sign-extension not required.
0b1	Data item must be sign-extended.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [20:16]

When ISV == '1':

SRT, bits [20:16]

Syndrome Register Transfer. The register number of the Wt/Xt/Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32, then this is the AArch64 view of the register. See 'Mapping of the general-purpose registers between the Execution states'.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When ISV == '0', FEAT_RASv2 is implemented, and GetESR_ELx_DFSC(EL3) IN {'010000', '01001x', '0101xx'}:

WU, bits [1:0] of bits [20:16]

Write Update. Describes whether a store instruction that generated an External abort updated the location.

WU	Meaning
0b00	Not a store instruction or translation table update, or the location might have been updated.
0b10	Store instruction or translation table update that did not update the location.
0b11	Store instruction or translation table update that updated the location.

In the description of this field, a store instruction is any memory-writing instruction that explicitly performs a store. This includes instructions that both read and write memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

When ISV == '1':

SF, bit [15]

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

SF	Meaning
0b0	Instruction loads/stores a 32-bit general-purpose register.
0b1	Instruction loads/stores a 64-bit general-purpose register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

FnP, bit [15]

FAR not Precise.

FnP	Meaning	Applies when
0b0	The FAR holds the faulting virtual address that generated the Data Abort.	
0b1	The FAR holds any virtual address within the naturally-aligned granule that contains the faulting virtual address that generated a Data Abort due to an SVE contiguous vector load/store instruction, or an SME load/store instruction. For more information about the naturally-aligned fault granule, see FAR_ELx (for example, FAR_EL1).	FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [14]

When ISV == '1':

AR, bit [14]

Acquire/Release.

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register being ZR, it is IMPLEMENTATION SPECIFIC whether this field is reported as 0 or 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_PFAR is implemented and GetESR_ELx_DFSC(EL3) IN {'010000', '01001x', '0101xx'}:

PFV, bit [14]

PFAR Valid. Describes whether the MFAR_EL3 register is valid.

PFV	Meaning
0b0	MFAR_EL3 is UNKNOWN.
0b1	MFAR_EL3 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VNCR, bit [13]

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VNCR	Meaning	Applies when
0b0	The fault was not generated by the use of VNCR_EL2 by EL1 code.	
0b1	The fault was generated by the use of VNCR_EL2 by EL1 code.	FEAT_NV2 is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:11]

When GetESR_ELx_DFSC(EL3) IN {'00xxxx', '10101x', '0000xx'}:

LST, bits [12:11]

Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort reported with ISV=1.

LST	Meaning	Applies when
0b00	The instruction that generated the Data Abort is not specified by this field.	
0b01	An ST64BV instruction generated the Data Abort.	FEAT_LS64_V is implemented
0b10	An LD64B or ST64B instruction generated the Data Abort.	FEAT_LS64 is implemented
0b11	An ST64BV0 instruction generated the Data Abort.	FEAT_LS64_ACCDATA is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL3) == '010000':

SET, bits [12:11]

Synchronous Error Type. Used when a synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

SET	Meaning	Applies when
0b00	Recoverable state (UER).	
0b10	Uncontainable (UC).	FEAT_RASv2 is not implemented
0b11	Restartable state (UEO).	

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

CM	Meaning
0b0	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Abort caused by an instruction reading from a memory location.
0b1	Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- If FEAT_RASv2 is implemented, an External abort on an Atomic access, reported with ESR_EL3.WU set to 0b00.
- A fault reported using a DFSC value of 0b110101, indicating an unsupported Exclusive or atomic access.
- A fault reported using a DFSC value of 0b110001, indicating an unsupported atomic hardware update fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Synchronous Tag Check Fault.	FEAT_MTE2 is implemented
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).	

All other values are reserved.

For more information about the lookup level associated with a fault, see ‘The lookup level associated with MMU faults’.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a trapped floating-point exception

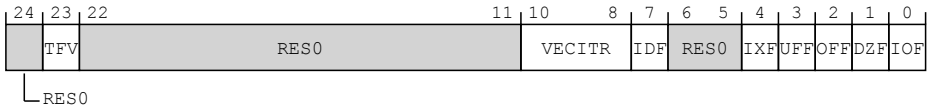

Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TFV	Meaning
0b0	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
0b1	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see 'Floating-point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IDF	Meaning
0b0	Input denormal floating-point exception has not occurred.
0b1	Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IXF	Meaning
0b0	Inexact floating-point exception has not occurred.
0b1	Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

UFF	Meaning
0b0	Underflow floating-point exception has not occurred.
0b1	Underflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

OFF	Meaning
0b0	Overflow floating-point exception has not occurred.
0b1	Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

DZF	Meaning
0b0	Divide by Zero floating-point exception has not occurred.
0b1	Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IOF	Meaning
0b0	Invalid Operation floating-point exception has not occurred.
0b1	Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a trapped floating-point exception

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

ISS encoding for a GCS exception

Bit [24]

Reserved, RES0.

ExType, bits [23:20]

The first level classification of GCS exceptions.

ExType	Meaning
0b0000	The exception reported is a Guarded Control Stack Data Check Exception.
0b0001	The exception reported is an EXLOCK Exception.
0b0010	The exception reported is a trap exception on GCSSTR or GCSSTTR instruction execution.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:15]

Reserved, RES0.

Raddr, bits [14:10]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When GetESR_ELx_ExType(EL3) == '0010':

Indicates the data address register number supplied in the instruction that has been trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [9:5]

When GetESR_ELx_ExType(EL3) == '0000':

Rn, bits [9:5]

Indicates a register number used by the instruction that caused the Guarded Control Stack Data Check Exception.

For a procedure return instruction reported with ESR_EL3.ISS.IT as `0b00000`, contains the register number for the register which contains the target address of the branch.

For a GCSPOPM instruction reported with ESR_EL3.ISS.IT as `0b00001`, contains the register number for the register which is the destination register of the instruction.

For a procedure return instruction reported with ESR_EL3.ISS.IT as `0b00010` or `0b00011`, contains the value `0b11110`, indicating X30.

For a GCSSS1 instruction reported with ESR_EL3.ISS.IT as `0b00100`, contains the register number for the register which is the input register of the instruction.

If ESR_EL3.ISS.IT is reported as `0b00101` or `0b01000`, this field is UNKNOWN

If ESR_EL3.ISS.IT is reported as `0b01001`, this field is `0b11111`

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When GetESR_ELx_ExType(EL3) == '0010':

Rvalue, bits [9:5]

Indicates the data value register number supplied in the instruction that has been trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IT, bits [4:0]

When GetESR_ELx_ExType(EL3) == '0000':

Type of the instruction that caused the Guarded Control Stack Data Check Exception.

IT	Meaning
`0b00000`	Guarded Control Stack Data Check Exception is from a procedure return instruction without Pointer authentication.
`0b00001`	Guarded Control Stack Data Check Exception is from a GCSPOPM instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IT	Meaning
0b00010	Guarded Control Stack Data Check Exception is from a procedure return instruction with Pointer authentication that uses key A.
0b00011	Guarded Control Stack Data Check Exception is from a procedure return instruction with Pointer authentication that uses key B.
0b00100	Guarded Control Stack Data Check Exception is from a GCSSS1 instruction.
0b00101	Guarded Control Stack Data Check Exception is from a GCSSS2 instruction.
0b01000	Guarded Control Stack Data Check Exception is from a GCSPOPCX instruction.
0b01001	Guarded Control Stack Data Check Exception is from a GCSPOPX instruction.

All other values are reserved

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Additional Information for ISS encoding for a GCS exception

The following fields describe the configuration settings for the traps that are reported using EC value 0b101101 and ExType value 0b0010:

- GCSCRE0_EL1.STREn
- GCSCR_EL1.STREn.
- GCSCR_EL2.STREn.
- GCSCR_EL3.STREn.
- HFGITR_EL2.nGCSSTR_EL1.

ISS encoding for an SError exception

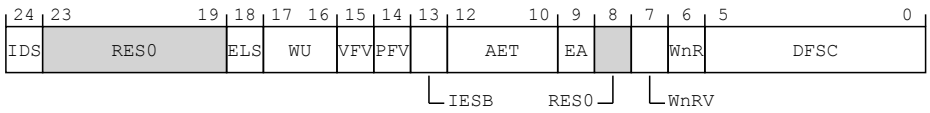

Note

In earlier versions of the architecture, an SError exception is referred to as an SError interrupt or an asynchronous External abort exception.

IDS, bit [24]

IMPLEMENTATION DEFINED syndrome.

IDS	Meaning
0b0	Bits [23:0] of the ISS field holds the fields described in this encoding. **Note** If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDS	Meaning
0b1	Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError exception.

Note

This field was previously called ISV.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:19]

Reserved, RES0.

ELS, bit [18]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL3) == '010001':

Meaning of ELR_ELx.

ELS	Meaning
0b0	Asynchronous. Does not indicate the trigger for the exception.
0b1	Synchronous. The exception was triggered by the instruction at ELR_ELx.

SError exceptions that report this field is 1 are not required to be precise.

The ESR_EL3.AET field describes whether the exception is precise or imprecise.

Corrected, Recoverable or Restartable exceptions are precise. Unrecoverable or Uncontainable exceptions are imprecise.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WU, bits [17:16]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL3) == '010001':

Write Update. Describes whether a store instruction that generated an External abort updated the location.

WU	Meaning
0b00	Not a store instruction or translation table update, or the location might have been updated.
0b10	Store instruction or translation table update that did not update the location.
0b11	Store instruction or translation table update that updated the location.

In the description of this field, a store instruction is any memory-writing instruction that explicitly performs a store. This includes instructions that both read and write memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VFV, bit [15]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL3) == '010001':

FAR Valid. Indicates the FAR_EL3 register contains a valid virtual address.

VFV	Meaning
0b0	FAR_EL3 is not valid, and holds an UNKNOWN value.
0b1	FAR_EL3 contains a valid virtual address associated with the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PFV, bit [14]

When FEAT_PFAR is implemented and GetESR_ELx_DFSC(EL3) == '010001':

PFAR Valid. Describes whether the MFAR_EL3 register is valid.

PFV	Meaning
0b0	MFAR_EL3 is UNKNOWN.
0b1	MFAR_EL3 is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IESB, bit [13]

When FEAT_IESB is implemented and GetESR_ELx_DFSC(EL3) == '010001':

Implicit error synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IESB	Meaning
0b0	The SError exception was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1	The SError exception was synchronized by the implicit error synchronization event and taken immediately.

This field is set to 1 when the SError exception was synchronized by an implicit error synchronization event and taken immediately on exception entry.

It this field is set to an IMPLEMENTATION SPECIFIC choice of 0 or 1 when the SError exception was synchronized by an implicit error synchronization event and taken immediately on an exception return.

For all other SError exceptions, this field is set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AET, bits [12:10]

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL3) == '010001':

Asynchronous Error Type.

Describes the PE error state after taking the SError exception.

AET	Meaning
0b000	Uncontainable (UC).
0b001	Unrecoverable state (UEU).
0b010	Restartable state (UEO).
0b011	Recoverable state (UER).
0b110	Corrected (CE).

All other values are reserved.

If multiple errors are taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA, bit [9]

When FEAT_RAS is implemented and GetESR_ELx_DFSC(EL3) == '010001':

External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

WnRV, bit [7]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL3) == '010001':

ESR_EL3.WnR valid.

WnRV	Meaning
`0b0`	ESR_EL3.WnR is not valid and has been set to 0.
`0b1`	ESR_EL3.WnR is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WnR, bit [6]

When FEAT_RASv2 is implemented and GetESR_ELx_DFSC(EL3) == '010001':

Write-not-Read. When the WnRV field is 1, indicates whether an exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
`0b0`	Exception was caused by an instruction reading from a memory location.
`0b1`	Exception was caused by an instruction writing to a memory location.

Accessing this bit has the following behavior:

- This bit is RES0 if ESR_EL3.WnRV is 0.
- This bit is not valid and reads UNKNOWN if an External abort on a Atomic access, reported with ESR_EL3.WU == `0b00`.
- Otherwise RW.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DFSC, bits [5:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RAS is implemented:

Data Fault Status Code.

DFSC	Meaning
0b000000	Uncategorized error.
0b010001	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ISS encoding for an exception from execution of a Breakpoint instruction

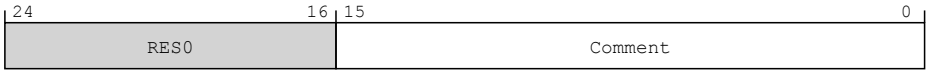

Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from execution of a Breakpoint instruction

For more information about generating these exceptions, see 'Breakpoint instruction exceptions'.

ISS encoding for a Branch Target Exception

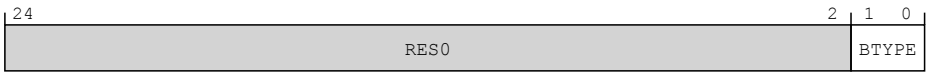

Bits [24:2]

Reserved, RES0.

BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional Information for ISS encoding for a Branch Target Exception

For more information about generating these exceptions, see 'The AArch64 application level programmers model'.

ISS encoding for an exception from a trapped Pointer Authentication instruction

24 … 0
RES0

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from a trapped Pointer Authentication instruction

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

ISS encoding for a PAC Fail exception

24 … 2	1	0
RES0	DnI	BnA

Bits [24:2]

Reserved, RES0.

DnI, bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

DnI	Meaning
0b0	Instruction Key.
0b1	Data Key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BnA, bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BnA	Meaning
0b0	A key.
0b1	B key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for a PAC Fail exception

The following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- AUTDA, AUTDZA.
- AUTDB, AUTDZB.
- AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA.
- AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB.
- AUTIASPPC, AUTIASPPCR, AUTIA171615.
- AUTIBSPPC, AUTIBSPPCR, AUTIB171615.

If FEAT_FPACCOMBINE is implemented, the following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- RETAA, RETAB.
- RETAASPPC, RETABSPPC.
- RETAASPPCR, RETABSPPCR.
- BLRAA, BLRAAZ, BLRAB, BLRABZ.
- BRAA, BRAB, BRAAZ, BRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing ESR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ESR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ESR_EL3();
end;
```

```
MSR ESR_EL3, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    ESR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.47 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL1.

Configuration

AArch64 System register FAR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DFAR[31:0].

AArch64 System register FAR_EL1 bits [63:32] are architecturally mapped to AArch32 System register IFAR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to FAR_EL1 are UNDEFINED.

Attributes

FAR_EL1 is a 64-bit register.

Field descriptions

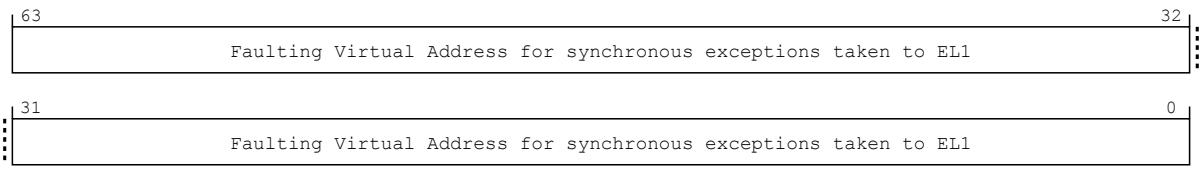

VA, bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), PC alignment faults (EC `0x22`), and Watchpoints (EC `0x34` or `0x35`). ESR_EL1.EC holds the EC syndrome value for the exception.

For a synchronous External abort:

- If the VA that generated the abort was from an address range for which Address tagging is enabled for the translation regime in use when the abort was generated, then bits[63:56] of FAR_EL1 are UNKNOWN.
- If the VA that generated the abort was from an address range for which Address tagging is disabled and Logical Address Tagging is enabled for the translation regime in use when the abort was generated, then bits[59:56] of FAR_EL1 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL1.FnV is 0, and FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL1 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If a memory fault that sets FAR_EL1 is generated from an STZGM instruction, the address held in FAR_EL1 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If a memory fault that sets FAR_EL1, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL1 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are `0x00000001`:

- The faulting address was generated by a load or store instruction that sequentially incremented from address `0xFFFFFFFF`. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL1.{ISV,FnP} to {0,1} on taking a Data Abort exception, the PE sets FAR_EL1 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception.

When the PE sets ESR_EL1.{FnV,FnP} to {0,1} on taking a Watchpoint exception, the PE sets FAR_EL1 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL1.DFSC is `0b010001`, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL1.DFSC is `0b11010x`, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL1 on a synchronous exception from any of the Memory Copy and Memory Set instructions is determined as follows:

- For an MMU fault reported as a Data Abort, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous Data Abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check Fault, the value is any address that caused a Tag Check Fault within the block size of the load or store.
- For a Watchpoint exception, the value is an address range of the size defined by the DCZID_EL0.BS field. This address does not need to be the element with a watchpoint, but can be some earlier element.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see ‘Address tagging’.

For a synchronous Tag Check Fault:

- If FEAT_MTE_TAGGED_FAR is implemented or Address tagging is disabled for the translation regime in use when the abort was generated, all bits of FAR_EL1 are not UNKNOWN.
- If FEAT_MTE_TAGGED_FAR is not implemented and Address tagging is enabled for the translation regime in use when the abort was generated, bits[63:60] of FAR_EL1 are UNKNOWN.

Execution at EL0 makes FAR_EL1 become UNKNOWN.

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or Data Abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all other exceptions taken to EL1, FAR_EL1 is UNKNOWN.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `FAR_EL1` or `FAR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FAR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().FAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x220);
    else
        X{64}(t) = FAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = FAR_EL2();
    else
        X{64}(t) = FAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = FAR_EL1();
end;
```

```
MSR FAR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().FAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x220) = X{64}(t);
    else
        FAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        FAR_EL2() = X{64}(t);
    else
        FAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    FAR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, FAR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x220);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = FAR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = FAR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR FAR_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x220) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        FAR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        FAR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, FAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = FAR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = FAR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = FAR_EL2();
end;
```

When FEAT_VHE is implemented MSR FAR_EL2, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        FAR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    FAR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    FAR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.48 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register FAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HDFAR[31:0].

AArch64 System register FAR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HIFAR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to FAR_EL2 are UNDEFINED.

Attributes

FAR_EL2 is a 64-bit register.

Field descriptions

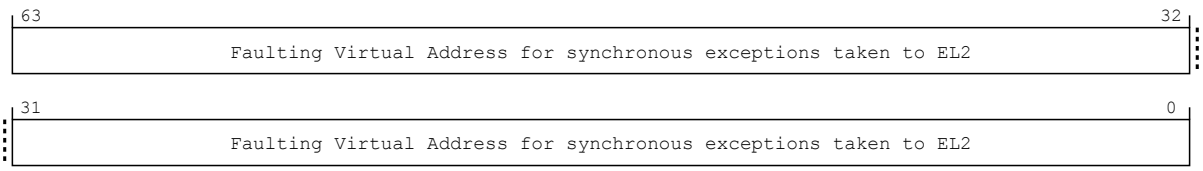

VA, bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), PC alignment faults (EC `0x22`), and Watchpoints (EC `0x34` or `0x35`). ESR_EL2.EC holds the EC syndrome value for the exception.

For a synchronous External abort:

- If the VA that generated the abort was from an address range for which Address tagging is enabled for the translation regime in use when the abort was generated, then bits[63:56] of FAR_EL2 are UNKNOWN.
- If the VA that generated the abort was from an address range for which Address tagging is disabled and Logical Address Tagging is enabled for the translation regime in use when the abort was generated, then bits[59:56] of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL2 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If a memory fault that sets FAR_EL2 is generated from an STZGM instruction, the address held in FAR_EL2 is IMPLEMENTATION DEFINED as one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The lowest address that gave rise to the fault.
- The address specified in the register argument.

If a memory fault that sets FAR_EL2, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL2 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are `0x00000001`:

- The faulting address was generated by a load or store instruction that sequentially incremented from address `0xFFFFFFFF`. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL2.{ISV,FnP} to {0,1} on taking a Data Abort exception, the PE sets FAR_EL2 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception.

When the PE sets ESR_EL2.{FnV,FnP} to {0,1} on taking a Watchpoint exception, the PE sets FAR_EL2 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL2.DFSC is `0b010001`, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL2.DFSC is `0b11010x`, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL2 on a synchronous exception from any of the Memory Copy and Memory Set instructions is determined as follows:

- For an MMU fault reported as a Data Abort, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous Data Abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check Fault, the value is any address that caused a Tag Check Fault within the block size of the load or store.
- For a Watchpoint exception, the value is an address range of the size defined by the DCZID_EL0.BS field. This address does not need to be the element with a watchpoint, but can be some earlier element.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

For a synchronous Tag Check Fault:

- If FEAT_MTE_TAGGED_FAR is implemented or Address tagging is disabled for the translation regime in use when the abort was generated, all bits of FAR_EL2 are not UNKNOWN.
- If FEAT_MTE_TAGGED_FAR is not implemented and Address tagging is enabled for the translation regime in use when the abort was generated, bits[63:60] of FAR_EL2 are UNKNOWN.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or Data Abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL2, FAR_EL2 is UNKNOWN.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `FAR_EL2` or `FAR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FAR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = FAR_EL1();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = FAR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = FAR_EL2();
end;
```

```
MSR FAR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        FAR_EL1() = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    FAR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    FAR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, FAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().FAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x220);
    else
        X{64}(t) = FAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = FAR_EL2();
    else
        X{64}(t) = FAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = FAR_EL1();
end;
```

When FEAT_VHE is implemented MSR FAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().FAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x220) = X{64}(t);
    else
        FAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        FAR_EL2() = X{64}(t);
    else
        FAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    FAR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.49 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions and PC alignment fault exceptions that are taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to FAR_EL3 are UNDEFINED.

Attributes

FAR_EL3 is a 64-bit register.

Field descriptions

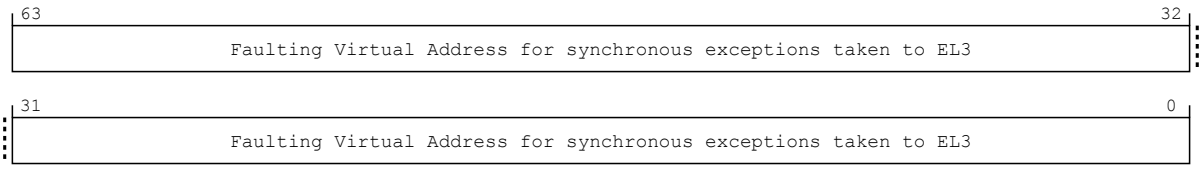

VA, bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL3. Exceptions that set the FAR_EL3 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), and PC alignment faults (EC `0x22`). ESR_EL3.EC holds the EC syndrome value for the exception.

For a synchronous External abort:

- If the VA that generated the abort was from an address range for which Address tagging is enabled for the translation regime in use when the abort was generated, then bits[63:56] of FAR_EL3 are UNKNOWN.
- If the VA that generated the abort was from an address range for which Address tagging is disabled and Logical Address Tagging is enabled for the translation regime in use when the abort was generated, then bits[59:56] of FAR_EL3 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL3.FnV is 0, and FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL3 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If a memory fault that sets FAR_EL3 is generated from an STZGM instruction, the address held in FAR_EL3 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument.

If a memory fault that sets FAR_EL3, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL3 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are `0x00000001`:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The faulting address was generated by a load or store instruction that sequentially incremented from address `0xFFFFFFFF`. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL3.{ISV,FnP} to {0,1} on taking a Data Abort exception, the PE sets FAR_EL3 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception.

The naturally-aligned fault granule is one of:

- When ESR_EL3.DFSC is `0b010001`, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL3.DFSC is `0b11010x`, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL3 on a synchronous exception from any of the Memory Copy and Memory Set instructions is determined as follows:

- For an MMU fault reported as a Data Abort or as a Granule Protection Check Exception, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a GPC fault, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check Fault, the value is any address that caused a Tag Check Fault within the block size of the load or store.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

For a synchronous Tag Check Fault:

- If FEAT_MTE_TAGGED_FAR is implemented or Address tagging is disabled for the translation regime in use when the abort was generated, all bits of FAR_EL3 are not UNKNOWN.
- If FEAT_MTE_TAGGED_FAR is not implemented and Address tagging is enabled for the translation regime in use when the abort was generated, bits[63:60] of FAR_EL3 are UNKNOWN.

Execution at EL2, EL1, and EL0 makes FAR_EL3 become UNKNOWN.

Note

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or Data Abort. It is the lowest address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL3, FAR_EL3 is UNKNOWN.

FAR_EL3 is made UNKNOWN on an exception return from EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, FAR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = FAR_EL3();
end;
```

```
MSR FAR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    FAR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.50 FGWTE3_EL3, Fine-Grained Write Traps EL3

The FGWTE3_EL3 characteristics are:

Purpose

Provides controls for traps of `MSR` and `MSRR` writes to specified EL3 system registers.

Configuration

This register is present only when EL3 is implemented, FEAT_FGWTE3 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to FGWTE3_EL3 are UNDEFINED.

Attributes

FGWTE3_EL3 is a 64-bit register.

Field descriptions

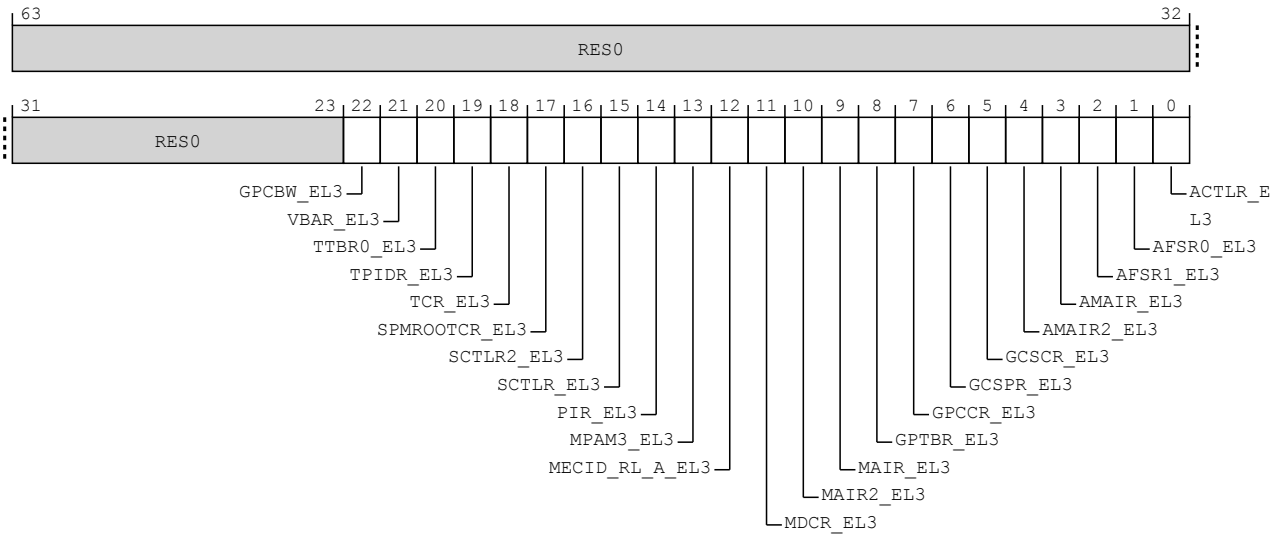

`MSR` accesses are trapped to EL3 and reported with EC syndrome value `0x18`.

`MSRR` accesses are trapped to EL3 and reported with EC syndrome value `0x14`.

The bits in this register are sticky. Writes to these bits have the following properties:

- A write of ‘0’ is ignored.
- A write of ‘1’ updates the bit to 1.

Bits [63:23]

Reserved, RES0.

GPCBW_EL3, bit [22]

When FEAT_RME_GPC3 is implemented:

Traps accesses of GPCBW_EL3 to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPCBW_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

VBAR_EL3, bit [21]

Traps accesses of VBAR_EL3 to EL3.

VBAR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

TTBR0_EL3, bit [20]

Traps accesses of TTBR0_EL3 to EL3.

TTBR0_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

TPIDR_EL3, bit [19]

Traps accesses of TPIDR_EL3 to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPIDR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

TCR_EL3, bit [18]

Traps accesses of TCR_EL3 to EL3.

TCR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

SPMROOTCR_EL3, bit [17]

When FEAT_RME is implemented and FEAT_SPMU is implemented:

Traps accesses of SPMROOTCR_EL3 to EL3.

SPMROOTCR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

SCTLR2_EL3, bit [16]

When FEAT_SCTLR2 is implemented:

Traps accesses of SCTLR2_EL3 to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCTLR2_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

SCTLR_EL3, bit [15]

Traps accesses of SCTLR_EL3 to EL3.

SCTLR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

PIR_EL3, bit [14]

When FEAT_S1PIE is implemented:

Traps accesses of PIR_EL3 to EL3.

PIR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM3_EL3, bit [13]

When FEAT_MPAM is implemented:

Traps accesses of MPAM3_EL3 to EL3.

MPAM3_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

MECID_RL_A_EL3, bit [12]

When FEAT_MEC is implemented:

Traps accesses of MECID_RL_A_EL3 to EL3.

MECID_RL_A_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

MDCR_EL3, bit [11]

Traps accesses of MDCR_EL3 to EL3.

MDCR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

MAIR2_EL3, bit [10]

When FEAT_AIE is implemented:

Traps accesses of MAIR2_EL3 to EL3.

MAIR2_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

MAIR_EL3, bit [9]

Traps accesses of MAIR_EL3 to EL3.

MAIR_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

GPTBR_EL3, bit [8]

When FEAT_RME is implemented:

Traps accesses of GPTBR_EL3 to EL3.

GPTBR_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

GPCCR_EL3, bit [7]

When FEAT_RME is implemented:

Traps accesses of GPCCR_EL3 to EL3.

GPCCR_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

GCSPR_EL3, bit [6]

When FEAT_GCS is implemented:

Traps accesses of GCSPR_EL3 to EL3.

GCSPR_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

GCSCR_EL3, bit [5]

When FEAT_GCS is implemented:

Traps accesses of GCSCR_EL3 to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GCSCR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

AMAIR2_EL3, bit [4]

When FEAT_AIE is implemented:

Traps accesses of AMAIR2_EL3 to EL3.

AMAIR2_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Otherwise:

Reserved, RES0.

AMAIR_EL3, bit [3]

Traps accesses of AMAIR_EL3 to EL3.

AMAIR_EL3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	MSR write accesses to the specified register are trapped to EL3 with EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is W1S.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AFSR1_EL3, bit [2]

Traps accesses of AFSR1_EL3 to EL3.

AFSR1_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

AFSR0_EL3, bit [1]

Traps accesses of AFSR0_EL3 to EL3.

AFSR0_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

ACTLR_EL3, bit [0]

Traps accesses of ACTLR_EL3 to EL3.

ACTLR_EL3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	`MSR` write accesses to the specified register are trapped to EL3 with EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1S.

Accessing FGWTE3_EL3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, FGWTE3_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b101

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_FGWTE3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = FGWTE3_EL3();
end;
```

```
MSR FGWTE3_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b101

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_FGWTE3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    FGWTE3_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.51 FPEXC32_EL2, Floating-Point Exception Control Register

The FPEXC32_EL2 characteristics are:

Purpose

Allows access to the AArch32 register FPEXC from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configuration

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

AArch64 System register FPEXC32_EL2 bits [31:0] are architecturally mapped to AArch32 System register FPEXC[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to FPEXC32_EL2 are UNDEFINED.

Attributes

FPEXC32_EL2 is a 64-bit register.

Field descriptions

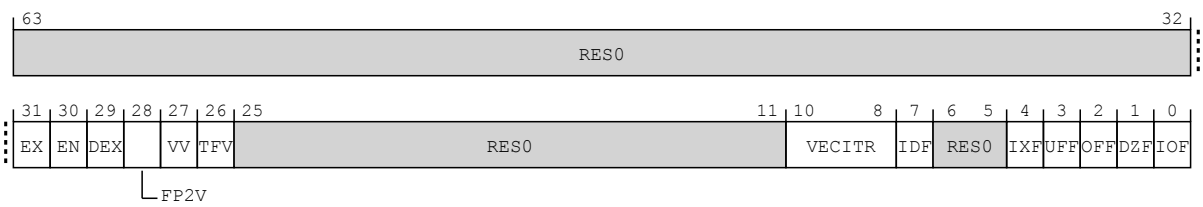

Bits [63:32]

Reserved, RES0.

EX, bit [31]

Exception bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RAZ/WI.

EN, bit [30]

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EN	Meaning
0b0	Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.
0b1	This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSASEDIS.

See the descriptions of the controls for more information.

Note

When executing at EL0 using AArch32:

- If EL1 is using AArch64, then the Effective value of FPEXC.EN is 1.
- If EL2 is using AArch64 and is enabled in the current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1, then the Effective value of FPEXC.EN is 1. However, Arm deprecates using the value of FPEXC32_EL2.EN to determine behavior.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DEX, bit [29]

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function `ExecutingCP10or11Instr()` returning TRUE. This field also indicates whether the FPEXC32_EL2.TFV field is valid.

The meaning of this bit is:

DEX	Meaning
0b0	The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function `ExecutingCP10or11Instr()`. If FPEXC32_EL2.TFV is RW then it is invalid and UNKNOWN. If FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.
0b1	The exception was generated during the execution of an allocated encoding. FPEXC32_EL2.TFV is valid and indicates the cause of the exception.

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the AArch32 FPSCR.{Stride, Len} fields as RAZ, this bit is RES0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FP2V, bit [28]

FPINST2 instruction Valid bit. From Armv8.0, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

VV, bit [27]

VECITR valid bit. From Armv8, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC32_EL2.DEX is 1. When valid, it indicates the cause of the exception and therefore whether FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are valid.

TFV	Meaning
`0b0`	The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR.{Stride, Len} was nonzero. If FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.
`0b1`	FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC32_EL2.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When an implementation does not implement trapping of floating-point exceptions, access to this field is RAZ/WI.
- When an implementation implements FPSCR.{Stride,Len} as RAZ, access to this field is RAO/WI.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector iteration count. From Armv8, this field is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

IDF	Meaning
`0b0`	Input Denormal exception has not occurred.
`0b1`	Input Denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of FPSCR.FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC32_EL2.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

IXF	Meaning
`0b0`	Inexact exception has not occurred.
`0b1`	Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFF, bit [3]

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UFF	Meaning
0b0	Underflow exception has not occurred.
0b1	Underflow exception has occurred.

Underflow trapped exceptions can occur:

- On half-precision data-processing instructions only when FPSCR.FZ16 is 0.
- Otherwise only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC32_EL2.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFF, bit [2]

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

OFF	Meaning
0b0	Overflow exception has not occurred.
0b1	Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZF, bit [1]

Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while FPSCR.DZE was 1:

DZF	Meaning
0b0	Divide by Zero exception has not occurred.
0b1	Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOF, bit [0]

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

IOF	Meaning
0b0	Invalid Operation exception has not occurred.
0b1	Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

Accessing FPEXC32_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPEXC32_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        X{64}(t) = FPEXC32_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        X{64}(t) = FPEXC32_EL2();
    end;
end;
```

MSR FPEXC32_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && CPTR_EL2().FPEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3().TFP == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPEXC32_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TFP == '1' then
        AArch64_SystemAccessTrap(EL3, 0x07);
    else
        FPEXC32_EL2() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.52 GCR_EL1, Tag Control Register.

The GCR_EL1 characteristics are:

Purpose

Tag Control Register.

Configuration

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to GCR_EL1 are UNDEFINED.

Attributes

GCR_EL1 is a 64-bit register.

Field descriptions

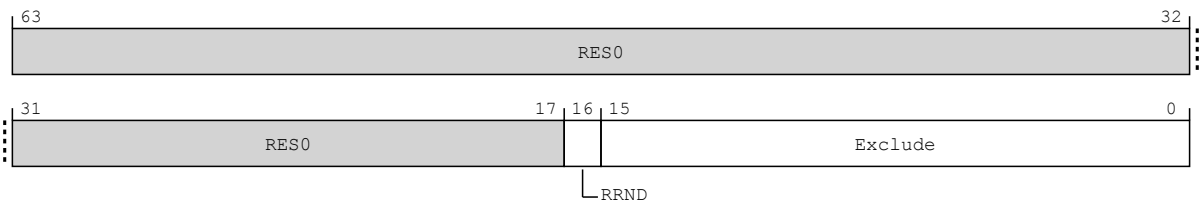

Bits [63:17]

Reserved, RES0.

RRND, bit [16]

Controls generation of tag values by the `IRG` instruction.

RRND	Meaning
`0b0`	`IRG` generates a tag value as defined by `RandomTag()` and `ChooseNonExcludedTag()`. This mode does not provide strong guarantees for randomness and should only be used for debugging purposes.
`0b1`	`IRG` generates an implementation-specific tag value with a distribution of tag values no worse than generated with GCR_EL1.RRND == 0.

Note

Arm recommends that IMPLEMENTATION DEFINED algorithms minimize the risk of a bias by selecting tags from a uniform distribution.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Exclude, bits [15:0]

Allocation Tag values excluded from selection by `ChooseNonExcludedTag()`.

If all bits of GCR_EL1.Exclude are 1, then the Allocation Tag value 0 will be used.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing GCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
        ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCR_EL1();
end;
```

```
MSR GCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b110

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
        ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.53 GMID_EL1, Multiple tag transfer ID Register

The GMID_EL1 characteristics are:

Purpose

Indicates the block size that is accessed by the LDGM and STGM System instructions.

Configuration

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to GMID_EL1 are UNDEFINED.

Attributes

GMID_EL1 is a 64-bit register.

Field descriptions

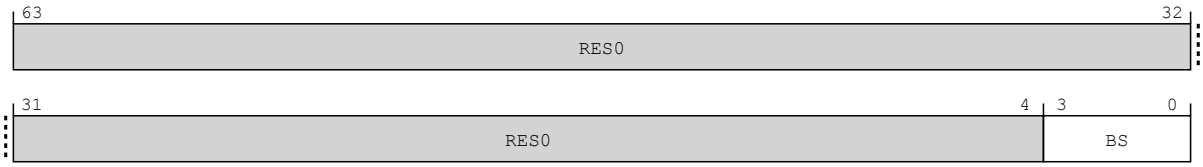

Bits [63:4]

Reserved, RES0.

BS, bits [3:0]

Log_2 of the block size in words. The minimum supported size is 16B (value == 2) and the maximum is 256B (value == 6).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing GMID_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GMID_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_MTE2) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID5 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID5 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID5 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GMID_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID5 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID5 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GMID_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GMID_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.54 GPCBW_EL3, Granule Protection Check Bypass Window Register (EL3)

The GPCBW_EL3 characteristics are:

Purpose

The control register for Granule Protection Check bypass window.

Configuration

This register is present only when FEAT_RME_GPC3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GPCBW_EL3 are UNDEFINED.

Attributes

GPCBW_EL3 is a 64-bit register.

Field descriptions

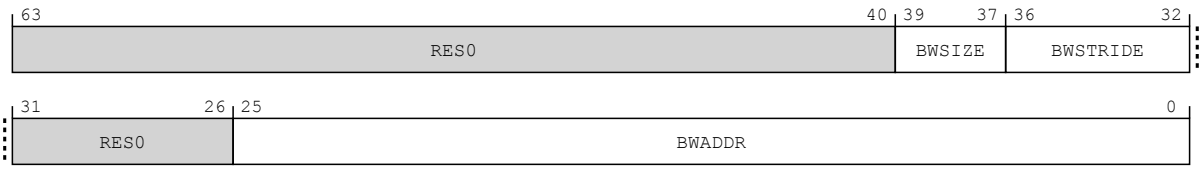

Bits [63:40]

Reserved, RES0.

BWSIZE, bits [39:37]

GPC Bypass Window Size.

BWSIZE defines the size of the GPC bypass memory region.

BWSIZE	Meaning
0b000	30 bits, 1GB GPC bypass window.
0b001	31 bits, 2GB GPC bypass window.
0b010	32 bits, 4GB GPC bypass window.
0b100	34 bits, 16GB GPC bypass window.
0b110	36 bits, 64GB GPC bypass window.

All other values are reserved.

This field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BWSTRIDE, bits [36:32]

GPC Bypass Window Stride.

BWSTRIDE allows creating multiple GPC bypass memory regions in the memory map across a specific stride.

BWSTRIDE	Meaning
0b00000	1TB stride.
0b00010	4TB stride.
0b00100	16TB stride.
0b00110	64TB stride.
0b00111	128TB stride.
0b01000	256TB stride.
0b01001	512TB stride.
0b01010	1PB stride.
0b10000	64PB (No stride).

All other values are reserved.

This field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [31:26]

Reserved, RES0.

BWADDR, bits [25:0]

GPC Bypass window address.

This field represents bits [55:30] of the GPC bypass window base address.

The GPC bypass window is:

- Aligned in memory to the size of the window as specified by GPCBW_EL3.BWSIZE.
- Duplicated in PA space across a stride specified using GPCBW_EL3.BWSTRIDE.

This means that only bits [gpcbwu:gpcbwl] of a PA are compared against bits [gpcbwu:gpcbwl] of the window base address derived from BWADDR when checking if a PA falls within the range of a window, where:

- gpcbwl is derived from GPCBW_EL3.BWSIZE as follows:

BWSIZE	gpcbwl
0b000	30
0b001	31
0b010	32
0b100	34
0b110	36

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- gpcbwu is derived from GPCBW_EL3.BWSTRIDE as follows:

BWSTRIDE	gpcbwu
0b00000	39
0b00010	41
0b00100	43
0b00110	45
0b00111	46
0b01000	47
0b01001	48
0b01010	49
0b10000	55

If the base address derived from BWADDR is not aligned to the size programmed in BWSIZE the configuration is invalid.

If this field is configured to a value that produces a GPC bypass window base address that is greater than or equal to the stride value configured by BWSTRIDE, the configuration is invalid.

An access to a PA falls within a GPC bypass window if the pseudocode function `PAWithinGPCBypassWindow()` returns TRUE.

This field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing GPCBW_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GPCBW_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_RME_GPC3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = GPCBW_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR GPCBW_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_RME_GPC3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().GPCBW_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        GPCBW_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.55 GPCCR_EL3, Granule Protection Check Control Register (EL3)

The GPCCR_EL3 characteristics are:

Purpose

The control register for Granule Protection Checks.

Configuration

This register is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GPCCR_EL3 are UNDEFINED.

Attributes

GPCCR_EL3 is a 64-bit register.

Field descriptions

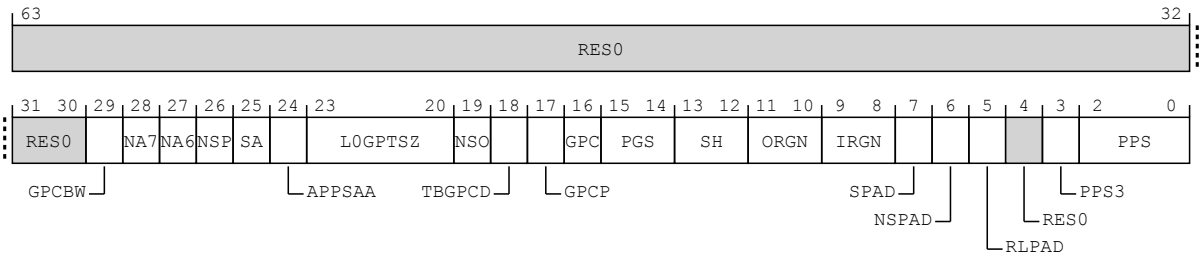

Bits [63:30]

Reserved, RES0.

GPCBW, bit [29]

When FEAT_RME_GPC3 is implemented:

GPC Bypass Window Enable.

This field governs the behavior of the GPC bypass windows.

GPCBW	Meaning
0b0	GPC bypass windows are disabled.
0b1	GPC bypass windows are enabled.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NA7, bit [28]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME_GDI is implemented:

No access 7.

This field governs the behavior of the GPI encoding for NA7.

NA7	Meaning
`0b0`	GPI encoding value of `0b0111` is reserved.
`0b1`	GPI encoding value of `0b0111` is No access permitted from this PE.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NA6, bit [27]

When FEAT_RME_GDI is implemented:

No access 6.

This field governs the behavior of the GPI encoding for NA6.

NA6	Meaning
`0b0`	GPI encoding value of `0b0110` is reserved.
`0b1`	GPI encoding value of `0b0110` is No access permitted from this PE.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSP, bit [26]

When FEAT_RME_GDI is implemented:

Non-secure Protected.

This field governs the behavior of the GPI encoding for NSP.

NSP	Meaning
`0b0`	GPI encoding value of `0b0101` is reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSP	Meaning
0b1	GPI encoding value of 0b0101 is No access permitted from this PE.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SA, bit [25]

When FEAT_RME_GDI is implemented:

System Agent.

This field governs the behavior of the GPI encoding for SA.

SA	Meaning
0b0	GPI encoding value of 0b0100 is reserved.
0b1	GPI encoding value of 0b0100 is No access permitted from this PE.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APPSAA, bit [24]

When FEAT_RME_GPC2 is implemented:

Above PPS All Access. This field governs the behavior of memory accesses to Secure, Realm and Root PA space, for physical addresses above the range configured by GPCCR_EL3.PPS.

APPSAA	Meaning
0b0	Accesses to addresses above the configured PPS must be to Non-secure PA space, otherwise they generate a GPF at level 0.
0b1	Accesses to addresses above the configured PPS, to any PA space, do not generate a GPF because of this control.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

L0GPTSZ, bits [23:20]

Level 0 GPT entry size.

This field advertises the number of least-significant address bits protected by each entry in the level 0 GPT.

L0GPTSZ	Meaning
`0b0000`	30-bits. Each entry covers 1GB of address space.
`0b0100`	34-bits. Each entry covers 16GB of address space.
`0b0110`	36-bits. Each entry covers 64GB of address space.
`0b1001`	39-bits. Each entry covers 512GB of address space.

All other values are reserved.

Access to this field is RO.

NSO, bit [19]

When FEAT_RME_GPC2 is implemented:

Non-secure Only. This field governs the behavior of the GPI encoding for NSO.

NSO	Meaning
`0b0`	GPI encoding value of `0b1101` is reserved.
`0b1`	GPI encoding value of `0b1101` is NSO.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBGPCD, bit [18]

When FEAT_TRBE_EXT is implemented:

Trace Buffer Granule Protection Check Disabled. Controls whether the Trace Buffer Unit accepts or rejects trace when Granule Protection Checks are disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TBGPCD	Meaning
`0b0`	The Trace Buffer Unit rejects trace when GPCCR_EL3.GPC is 0.
`0b1`	The Trace Buffer Unit accepts trace when GPCCR_EL3.GPC is 0.

When the Trace Buffer Unit rejects trace, the trace might remain buffered by the trace unit until the Trace Buffer Unit is able to accept trace. When the Trace Buffer Unit accepts trace, the Trace Buffer Unit writes the trace to memory.

Note

Setting GPCCR_EL3.{TBGPCD, GPC} to {1, 0} means that the Trace Buffer Unit might write to memory without any Granule Protection Checks. The addresses that the Trace Buffer Unit writes to can be programmed by an external agent. The physical address spaces the Trace Buffer Unit can address are restricted by an IMPLEMENTATION DEFINED debug authentication interface.
Setting GPCCR_EL3.{TBGPCD, GPC} to {1, 1} means that GPCCR_EL3.{TBGPCD, GPC} will become {1, 0} on a Warm reset.

This field is ignored by the PE and treated as one when any of the following are true:

- GPCCR_EL3.GPC == 1.
- `ExternalRootInvasiveDebugEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

GPCP, bit [17]

Granule Protection Check Priority.

This control governs behavior of granule protection checks on fetches of stage 2 Table descriptors.

GPCP	Meaning
`0b0`	GPC faults are all reported with a priority that is consistent with the GPC being performed on any access to physical address space.
`0b1`	A GPC fault for the fetch of a Table descriptor for a stage 2 translation table walk might not be generated or reported. All other GPC faults are reported with a priority consistent with the GPC being performed on all accesses to physical address spaces.

This bit is permitted to be cached in a TLB.

An implementation is permitted to treat this field as RES0, with an Effective value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GPC, bit [16]

Granule Protection Check Enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPC	Meaning
0b0	Granule protection checks are disabled. Accesses are not prevented by this mechanism.
0b1	All accesses to physical address spaces are subject to granule protection checks, except for fetches of GPT information and accesses governed by the GPCCR_EL3.GPCP control.

If any stage of translation is enabled, this bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

PGS, bits [15:14]

Physical Granule size.

PGS	Meaning
0b00	4KB.
0b01	64KB.
0b10	16KB.

All other values are reserved.

The value of this field is permitted to be cached in a TLB.

Granule sizes not supported for stage 1 and not supported for stage 2, as defined in ID_AA64MMFR0_EL1, are reserved. For example, if ID_AA64MMFR0_EL1.TGran16 == 0b0000 and ID_AA64MMFR0_EL1.TGran16_2 == 0b0001, then the PGS encoding 0b10 is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [13:12]

GPT fetch Shareability attribute

SH	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

All other values are reserved.

Fetches of GPT information are made with the Shareability attribute that is configured in this field.

If both ORGN and IRGN are configured with Non-cacheable attributes, it is invalid to configure this field to any value other than 0b10.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ORGN, bits [11:10]

GPT fetch Outer cacheability attribute.

ORGN	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

Fetches of GPT information are made with the Outer cacheability attributes configured in this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN, bits [9:8]

GPT fetch Inner cacheability attribute.

IRGN	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

Fetches of GPT information are made with the Inner cacheability attributes configured in this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SPAD, bit [7]

When FEAT_RME_GPC2 is implemented:

Secure PA space Disable. This field controls access to the Secure PA space.

SPAD	Meaning
0b0	This control has no effect on accesses.
0b1	When granule protection checks are enabled, access to the Secure Physical Address space generates a Granule Protection fault.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

NSPAD, bit [6]

When FEAT_RME_GPC2 is implemented:

Non-secure PA space Disable. This field controls access to the Non-secure PA space.

NSPAD	Meaning
0b0	This control has no effect on accesses.
0b1	When granule protection checks are enabled, access to the Non-secure Physical Address space generates a Granule Protection fault.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLPAD, bit [5]

When FEAT_RME_GPC2 is implemented:

Realm PA space Disable. This field controls access to the Realm PA space.

RLPAD	Meaning
0b0	This control has no effect on accesses.
0b1	When granule protection checks are enabled, access to the Realm Physical Address space generates a Granule Protection fault.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [4]

Reserved, RES0.

PPS3, bit [3]

When FEAT_RME_GPC3 is implemented:

This field extends GPCCR_EL3.PPS[2:0], creating a GPCCR_EL3.PPS[3:0] field.

For a description of the values derived by evaluating PPS, see GPCCR_EL3.PPS[2:0].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PPS, bits [2:0]

When FEAT_RME_GPC3 is implemented:

Protected Physical Address Size.

The size of the memory region protected by GPTBR_EL3, in terms of the number of least-significant address bits.

This field is evaluated with PPS3, as {PPS3, PPS} to give PPS[3:0], interpreted as:

PPS[3:0]	Meaning
0b0000	32 bits, 4GB protected address space.
0b0001	36 bits, 64GB protected address space.
0b0010	40 bits, 1TB protected address space.
0b0011	42 bits, 4TB protected address space.
0b0100	44 bits, 16TB protected address space.
0b1000	46 bits, 64TB protected address space.
0b1001	47 bits, 128TB protected address space.
0b0101	48 bits, 256TB protected address space.
0b0110	52 bits, 4PB protected address space.
0b0111	56 bits, 64PB protected address space.

All other values are reserved.

Configuration of this field to a value exceeding the implemented physical address size is invalid.

The value of this field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Protected Physical Address Size.

The size of the memory region protected by GPTBR_EL3, in terms of the number of least-significant address bits.

PPS	Meaning
0b000	32 bits, 4GB protected address space.
0b001	36 bits, 64GB protected address space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PPS	Meaning
0b010	40 bits, 1TB protected address space.
0b011	42 bits, 4TB protected address space.
0b100	44 bits, 16TB protected address space.
0b101	48 bits, 256TB protected address space.
0b110	52 bits, 4PB protected address space.

All other values are reserved.

Configuration of this field to a value exceeding the implemented physical address size is invalid.

The value of this field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing GPCCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GPCCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = GPCCR_EL3();
end;
```

```
MSR GPCCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().GPCCR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        GPCCR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.56 GPTBR_EL3, Granule Protection Table Base Register

The GPTBR_EL3 characteristics are:

Purpose

The control register for Granule Protection Table base address.

Configuration

This register is present only when FEAT_RME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to GPTBR_EL3 are UNDEFINED.

Attributes

GPTBR_EL3 is a 64-bit register.

Field descriptions

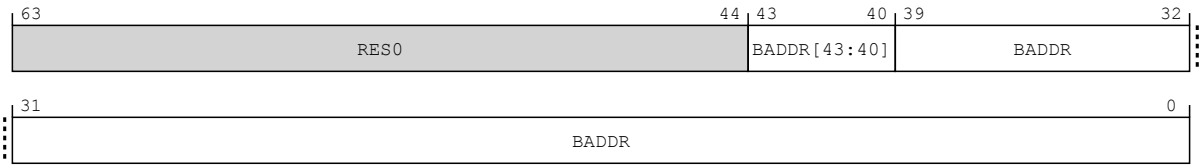

Bits [63:44]

Reserved, RES0.

BADDR[43:40], bits [43:40]

When FEAT_RME_GPC3 is implemented:

Extension to BADDR[39:0]. This field represents bit [55:52] of the level 0 GPT base address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BADDR, bits [39:0]

Base address for the level 0 GPT.

This field represents bits [51:12] of the level 0 GPT base address.

The level 0 GPT is aligned in memory to the greater of:

- The size of the level 0 GPT in bytes.
- 4KB.

Bits [x:0] of the base address are treated as zero, where:

- x = Max(pps - l0gptsz + 2, 11)
- pps is derived from GPCCR_EL3.PPS as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPCCR_EL3.PPS	pps
0b000	32
0b001	36
0b010	40
0b011	42
0b100	44
0b101	48
0b110	52

- l0gptsz is derived from GPCCR_EL3.L0GPTSZ as follows:

GPCCR_EL3.L0GPTSZ	l0gptsz
0b0000	30
0b0100	34
0b0110	36
0b1001	39

If x is greater than 11, then BADDR[x - 12:0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing GPTBR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GPTBR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = GPTBR_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR GPTBR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().GPTBR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        GPTBR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.57 HACDBSBR_EL2, Hardware Accelerator for Cleaning Dirty State Base Register

The HACDBSBR_EL2 characteristics are:

Purpose

Control register for HACDBS structure.

Configuration

This register is present only when FEAT_HACDBS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HACDBSBR_EL2 are UNDEFINED.

Attributes

HACDBSBR_EL2 is a 64-bit register.

Field descriptions

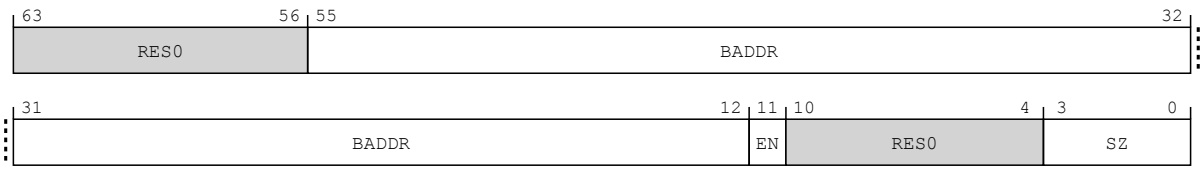

Bits [63:56]

Reserved, RES0.

BADDR, bits [55:12]

HACDBS base address, bits [55:12].

- Bits [55:12] of the base address are the value of this field.
- Bits [11:0] of the base address are 0.

Bits of this field above the implemented physical address size, indicated in ID_AA64MMFR0_EL1.PARange, are RES0.

Based on the value of the SZ field of this register, for encodings of the SZ field greater than 4KB, bits [(SZ+12-1):12] of this field are RES0 such that the base address of the HACDBS is aligned to its size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [11]

Enable use of HACDBS.

EN	Meaning
0b0	Hardware accelerator for cleaning Dirty state is disabled.
0b1	Hardware accelerator for cleaning Dirty state is enabled.

If SCR_EL3.HACDBSEn is set to 0, then this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:4]

Reserved, RES0.

SZ, bits [3:0]

Size of the HACDBS.

SZ	Meaning
0b0000	4KB
0b0001	8KB
0b0010	16KB
0b0011	32KB
0b0100	64KB
0b0101	128KB
0b0110	256KB
0b0111	512KB
0b1000	1MB
0b1001	2MB

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACDBSBR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HACDBSBR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b100

```
if !(IsFeatureImplemented(FEAT_HACDBS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x2F0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HACDBSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HACDBSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HACDBSBR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HACDBSBR_EL2();
end;
```

```
MSR HACDBSBR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b100

```
if !(IsFeatureImplemented(FEAT_HACDBS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x2F0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HACDBSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HACDBSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HACDBSBR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HACDBSBR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.58 HACDBSCONS_EL2, Hardware Accelerator for Cleaning Dirty State Consumer Register

The HACDBSCONS_EL2 characteristics are:

Purpose

Read index for HACDBS structure.

Configuration

This register is present only when FEAT_HACDBS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HACDBSCONS_EL2 are UNDEFINED.

Attributes

HACDBSCONS_EL2 is a 64-bit register.

Field descriptions

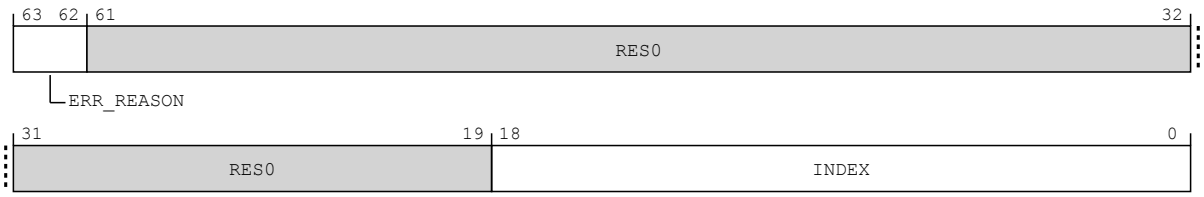

ERR_REASON, bits [63:62]

Reason for HACDBS error.

ERR_REASON	Meaning
0b00	The PE has not experienced an error while processing the HACDBS.
0b01	STRUCTF - A read of an entry from the HACDBS has experienced a fault.
0b10	IPAF - A stage 2 walk of an IPA from a HACDBS entry has experienced an MMU fault.
0b11	IPAHACF - An entry from the HACDBS experienced an error that is not an MMU fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [61:19]

Reserved, RES0.

INDEX, bits [18:0]

This field indicates the index of the HACDBS entry that will be read from next.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing HACDBSCONS_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HACDBSCONS_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_HACDBS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x308);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HACDBSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HACDBSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HACDBSCONS_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HACDBSCONS_EL2();
end;
```

```
MSR HACDBSCONS_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_HACDBS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x308) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HACDBSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HACDBSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HACDBSCONS_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HACDBSCONS_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.59 HACR_EL2, Hypervisor Auxiliary Control Register

The HACR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of IMPLEMENTATION DEFINED aspects of EL1 or EL0 operation.

Note

Arm recommends that the values in this register do not cause unnecessary traps to EL2 when the Effective value of HCR_EL2.{E2H, TGE} == {1, 1}.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register HACR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HACR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to HACR_EL2 are UNDEFINED.

Attributes

HACR_EL2 is a 64-bit register.

Field descriptions

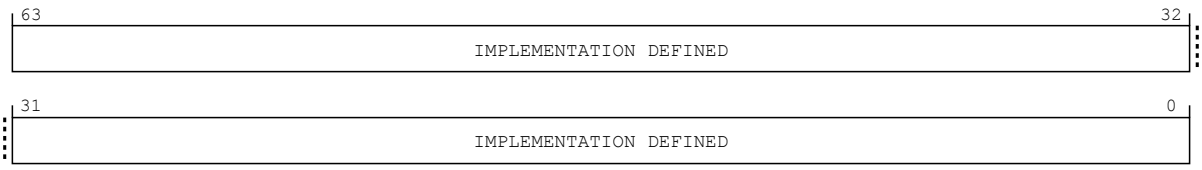

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HACR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = HACR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = HACR_EL2();
end;
```

MSR HACR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HACR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    HACR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.60 HAFGRTR_EL2, Hypervisor Activity Monitors Fine-Grained Read Trap Register

The HAFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of MRS reads of Activity Monitors System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_AMUv1 is implemented, FEAT_FGT is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to HAFGRTR_EL2 are UNDEFINED.

Attributes

HAFGRTR_EL2 is a 64-bit register.

Field descriptions

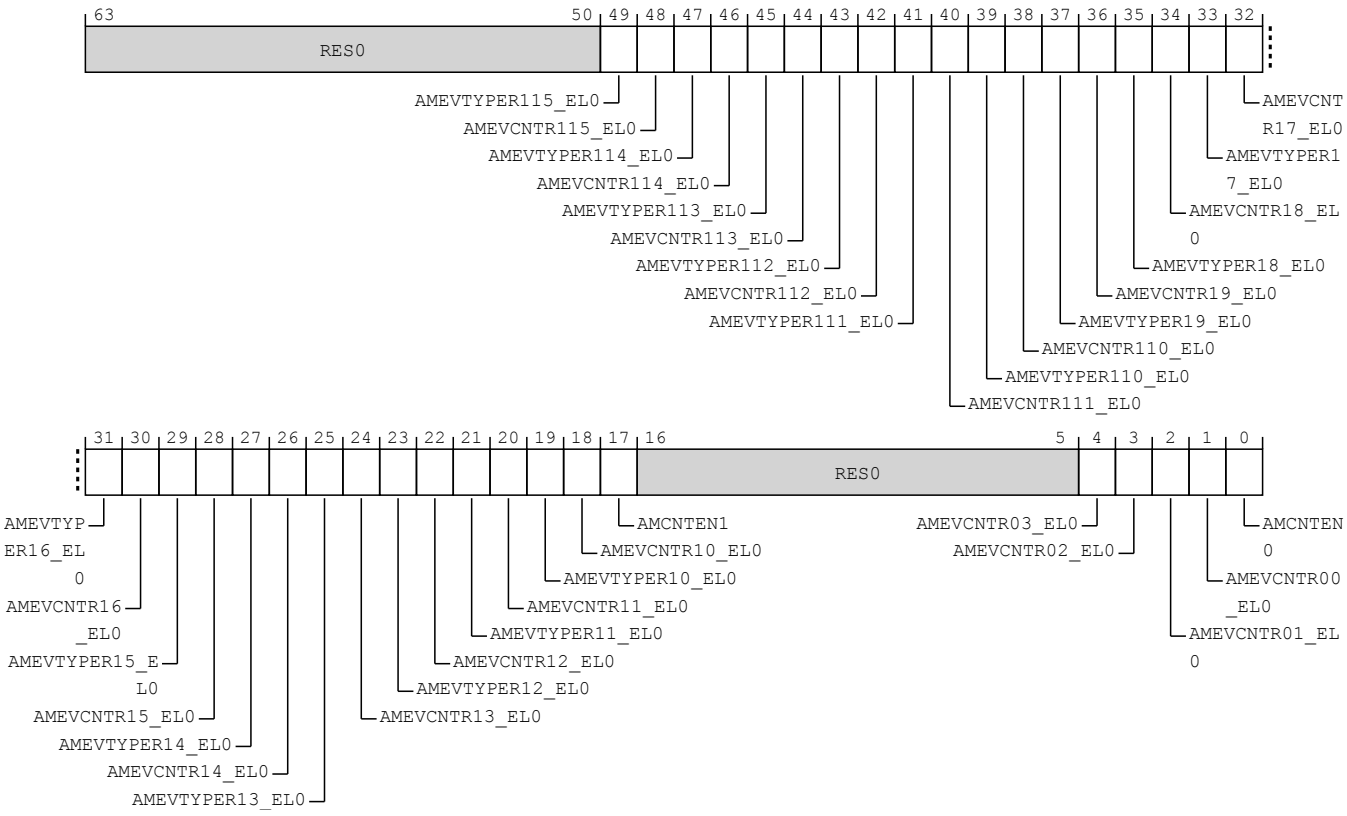

Bits [63:50]

Reserved, RES0.

AMEVTYPER1<x>_EL0 , bits [19+2x], for x = 15 to 0

Trap MRS reads of AMEVTYPER1<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER1<x> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMEVTYPER1<x>_EL0	Meaning
0b0	MRS reads of AMEVTYPER1<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTYPER1<x> at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: • MRS reads of AMEVTYPER1<x>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of AMEVTYPER1<x> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(AMCGCR_EL0.CG1NC), access to this field is RES0.
- When !IsG1ActivityMonitorImplemented(x), access to this field is RES0.

AMEVCNTR1<x>_EL0 , bits [18+2x], for x = 15 to 0

Trap MRS reads of AMEVCNTR1<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR1<x> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

AMEVCNTR1<x>_EL0	Meaning
0b0	MRS reads of AMEVCNTR1<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR1<x> at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: • MRS reads of AMEVCNTR1<x>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of AMEVCNTR1<x> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(AMCGCR_EL0.CG1NC), access to this field is RES0.
- When !IsG1ActivityMonitorImplemented(x), access to this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMCNTEN<x> , bits [17x], for x = 1 to 0

Trap MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MRS reads of AMCNTENCLR<x>_EL0 and AMCNTENSET<x>_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: MRC reads of AMCNTENCLR<x> and AMCNTENSET<x>.

AMCNTEN<x>	Meaning
0b0	The operations listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: • MRS reads at EL1 and EL0 using AArch64 of AMCNTENCLR<x>_EL0 and AMCNTENSET<x>_EL0 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads at EL0 using AArch32 of AMCNTENCLR<x> and AMCNTENSET<x> are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [16:5]

Reserved, RES0.

AMEVCNTR0<x>_EL0 , bits [x+1], for x = 3 to 0

Trap MRS reads of AMEVCNTR0<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR0<x> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

AMEVCNTR0<x>_EL0	Meaning
0b0	MRS reads of AMEVCNTR0<x>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCNTR0<x> at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: • MRS reads of AMEVCNTR0<x>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of AMEVCNTR0<x> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When x >= 4, access to this field is RES0.

Accessing HAFGRTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HAFGRTR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_FGT) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1E8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HAFGRTR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HAFGRTR_EL2();
end;
```

```
MSR HAFGRTR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_FGT) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1E8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HAFGRTR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HAFGRTR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.61 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

Unless otherwise stated, the bits in this register behave as if they are 0 for all purposes other than direct reads of the register if EL2 is not enabled in the current Security state.

AArch64 System register HCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HCR[31:0].

AArch64 System register HCR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HCR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to HCR_EL2 are UNDEFINED.

Attributes

HCR_EL2 is a 64-bit register.

Field descriptions

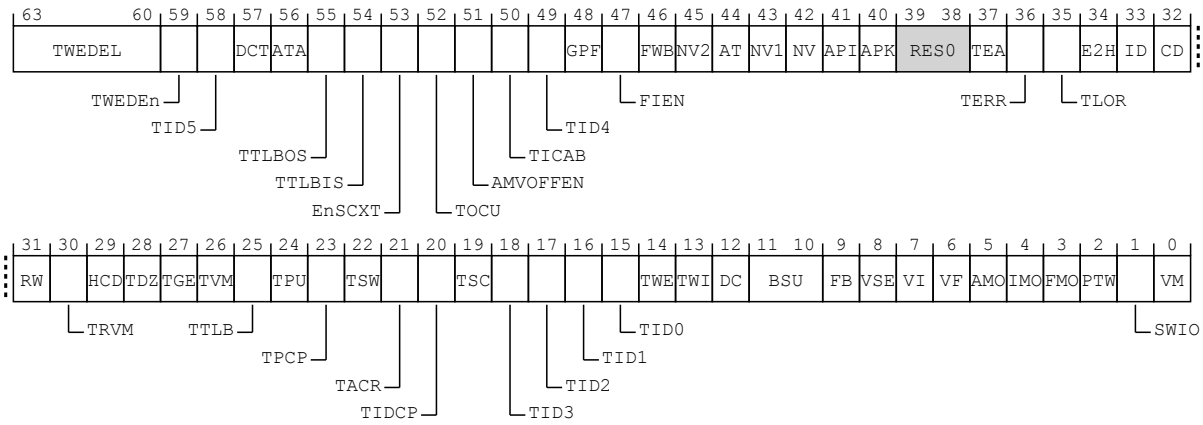

TWEDEL, bits [63:60]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when HCR_EL2.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by HCR_EL2.TWE as $2^{(\mathrm{TWEDEL} + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [59]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by HCR_EL2.TWE.

TWEDEn	Meaning
0b0	The delay for taking the trap is IMPLEMENTATION DEFINED.
0b1	The delay for taking the trap is at least the number of cycles defined in HCR_EL2.TWEDEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID5, bit [58]

When FEAT_MTE2 is implemented:

Trap ID group 5. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

AArch64:

- GMID_EL1.

TID5	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL1 accesses to ID group 5 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DCT, bit [57]

When FEAT_MTE2 is implemented:

Default Cacheability Tagging. When HCR_EL2.DC is in effect, controls whether EL1&0 stage 1 translations have the Tagged attribute.

DCT	Meaning
0b0	Stage 1 translations do not have the Tagged attribute.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Stage 1 translations have the Tagged attribute.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [56]

When FEAT_MTE2 is implemented:

Memory tagging enable override:

- Overrides enabling of Memory tagging at EL1 and EL0.
- If disabled by this control, accesses to the following registers at EL1 are trapped to EL2 and reported using EC syndrome value 0x18:
  - GCR_EL1.
  - RGSR_EL1.
  - TFSRE0_EL1.
  - TFSR_EL1.
  - Accesses with the register name TFSR_EL2 that are not UNDEFINED.

ATA	Meaning
0b0	Disables the use of Memory tagging at EL1 and EL0. The specified registers are trapped to EL2.
0b1	This field has no effect on the use of Memory tagging at EL1 and EL0. The field does not trap the specified registers to EL2.

The Effective value of this field is 1 if any of the following are true:

- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL2 is not implemented or not enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTLBOS, bit [55]

When FEAT_EVT2 is implemented:

Trap TLB maintenance instructions that operate on the Outer Shareable domain. Traps execution of those TLB maintenance instructions at EL1 using AArch64 to EL2, when EL2 is enabled in the current Security state. The following instructions are trapped and reported with EC syndrome value 0x18:

- TLBI VMALLE1OS, TLBI VAE1OS, TLBI ASIDE1OS,TLBI VAAE1OS, TLBI VALE1OS, and TLBI VAALE1OS.
- If FEAT_TLBIRANGE is implemented, TLBI RVAE1OS, TLBI RVAAE1OS, TLBI RVALE1OS, and TLBI RVAALE1OS.
- If FEAT_XS is implemented, then the *OSNXS variants are also trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTLBOS	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Execution of the specified instructions are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTLBIS, bit [54]

When FEAT_EVT2 is implemented:

Trap TLB maintenance instructions that operate on the Inner Shareable domain. Traps execution of those TLB maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, then the following instructions are trapped to EL2 and reported with EC syndrome value 0x18:
  - TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS, TLBI RVAE1IS, TLBI RVAAE1IS, TLBI RVALE1IS, and TLBI RVAALE1IS.
- If EL1 is using AArch32, then the following instructions are trapped to EL2 and reported with EC syndrome value 0x03:
  - TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, and TLBIMVAALIS.
- If FEAT_XS is implemented, then the *ISNXS variants are also trapped.

TTLBIS	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Execution of the specified instructions are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSCXT, bit [53]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enable Access to the SCXTNUM_EL1 and SCXTNUM_EL0 registers. The defined values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnSCXT	Meaning
0b0	When EL2 is enabled in the current Security state, EL1 accesses to SCXTNUM_EL0 and SCXTNUM_EL1 are disabled, causing an exception to EL2, and the value of the registers to be treated as 0. When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1} and EL2 is enabled in the current Security state, EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL2, and the value of the register to be treated as 0.
0b1	This control does not cause accesses to SCXTNUM_EL0 or SCXTNUM_EL1 to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the value of this field is 0, accesses at EL0 are not trapped by this control.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TOCU, bit [52]

When FEAT_EVT is implemented:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at EL0 and EL1 to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL0 is using AArch64, the value of SCTLR_EL1.UCI is 1, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then the following instructions at EL0 are trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IVAU, and DC CVAU.
- If EL1 is using AArch64, then the following instructions at EL1 are trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IVAU, IC IALLU, and DC CVAU.
- If EL1 is using AArch32, then the following instructions are trapped at EL1 to EL2 and reported with EC syndrome value `0x03`:
  - ICIMVAU, ICIALLU, and DCCMVAU.

Note

When SCTLR_EL1.UCI is 0, the trap on execution of instructions at EL0 is higher priority than this control. An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

TOCU	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	For each of the specified instructions, if the execution of the instruction can be trapped, accesses are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

AMVOFFEN, bit [51]

When FEAT_AMUv1p1 is implemented:

Activity Monitors Virtual Offsets Enable.

AMVOFFEN	Meaning
0b0	Virtualization of the Activity Monitors is disabled. Indirect reads of the virtual offset registers are zero.
0b1	Virtualization of the Activity Monitors is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TICAB, bit [50]

When FEAT_EVT is implemented:

Trap `ICIALLUIS` and `IC IALLUIS` cache maintenance instructions. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, then the following instruction is trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IALLUIS.
- If EL1 is using AArch32, then the following instruction is trapped to EL2 and reported with EC syndrome value `0x03`:
  - ICIALLUIS.

TICAB	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	For each of the specified instructions, if the execution of the instruction can be trapped, EL1 access is trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID4, bit [49]

When FEAT_EVT is implemented:

Trap ID group 4. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

AArch64:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL1 reads of CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1.
- EL1 writes to CSSELR_EL1.

AArch32:

- EL1 reads of CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- EL1 writes to CSSELR.

TID4	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 accesses to ID group 4 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

GPF, bit [48]

When FEAT_RME is implemented:

Controls the reporting of Granule protection faults at EL0 and EL1.

GPF	Meaning
`0b0`	This control does not cause exceptions to be routed from EL0 and EL1 to EL2.
`0b1`	Instruction Abort exceptions and Data Abort exceptions due to GPFs from EL0 and EL1 are routed to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FIEN, bit [47]

When FEAT_RASv1p1 is implemented:

Fault Injection Enable. Unless this bit is set to 1, accesses to the ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 registers from EL1 generate a Trap exception to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`.

FIEN	Meaning
`0b0`	Accesses to the specified registers from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FIEN	Meaning
0b1	This control does not cause any instructions to be trapped.

If EL2 is disabled in the current Security state, the Effective value of HCR_EL2.FIEN is 1.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FWB, bit [46]

When FEAT_S2FWB is implemented:

Forced Write-Back. Defines the combined cacheability attributes in a 2 stage translation regime.

FWB	Meaning
0b0	When this bit is 0, then the combination of stage 1 and stage 2 translations on memory type and cacheability attributes are as described in the Armv8.0 architecture. For more information, see 'Combining stage 1 and stage 2 memory type attributes'.
0b1	When this bit is 1, then the encoding of the stage 2 memory type and cacheability attributes in bits[5:2] of the stage 2 Page or Block descriptors are as described in 'Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled'.

In Secure state, this bit applies to both the Secure stage 2 translation and the Non-secure stage 2 translation.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NV2, bit [45]

When FEAT_NV2 is implemented:

Nested Virtualization. Changes the behaviors of HCR_EL2.{NV1, NV} to provide a mechanism for hardware to transform reads and writes from System registers into reads and writes from memory.

NV2	Meaning
0b0	This bit has no effect on the behavior of HCR_EL2.{NV1, NV}. The behavior of HCR_EL2.{NV1, NV} is as defined for FEAT_NV.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NV2	Meaning
0b1	Redefines behavior of HCR_EL2{NV1, NV} to enable: • Transformation of read/writes to registers into read/writes to memory. • Redirection of EL2 registers to EL1 registers. Any exception taken from EL1 and taken to EL1 causes SPSR_EL1.M[3:2] to be set to 0b10 and not 0b01.

When the Effective value of HCR_EL2.NV is 0, the Effective value of this field is 0 and this field is treated as 0 for all purposes other than direct reads and writes of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AT, bit [44]

When FEAT_NV is implemented:

Address Translation. EL1 execution of the following address translation instructions is trapped to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value 0x18:

- AT S1E0R, AT S1E0W, AT S1E1R, AT S1E1W, AT S1E1RP, and AT S1E1WP.
- If FEAT_ATS1A is implemented, AT S1E1A.

AT	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 execution of the specified instructions is trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NV1, bit [43]

When FEAT_NV2 is implemented:

Nested Virtualization.

NV1	Meaning
0b0	If the Effective value of HCR_EL2.{NV2, NV} is {1, 1}, accesses executed from EL1 to implemented EL12, EL02, or EL2 registers are transformed to loads and stores. If the Effective value of HCR_EL2.{NV2, NV} is not {1, 1}, this control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NV1	Meaning
0b1	If the Effective value of HCR_EL2.NV2 is 1, accesses executed from EL1 to implemented EL2 registers are transformed to loads and stores. If the Effective value of HCR_EL2.NV2 is 0, EL1 accesses to VBAR_EL1, ELR_EL1, SPSR_EL1, and, when FEAT_CSV2_2 or FEAT_CSV2_1p2 is implemented, SCXTNUM_EL1, are trapped to EL2, when EL2 is enabled in the current Security state, and are reported using EC syndrome value 0x18.

If the Effective value of HCR_EL2.NV2 is 1, the Effective value of HCR_EL2.NV1 defines which EL1 register accesses are transformed to loads and stores.

The trapping of EL1 registers caused by other control bits has priority over the transformation of these accesses.

If a register is specified that is not implemented by an implementation, then access to that register are UNDEFINED.

For the list of registers affected, see 'Enhanced support for nested virtualization'.

If the Effective value of HCR_EL2.{NV1, NV} is {0, 1}, any exception taken from EL1, and taken to EL1, causes the SPSR_EL1.M[3:2] to be set to 0b10, and not 0b01.

If the Effective value of HCR_EL2.{NV1, NV} is {1, 1}, then:

- The EL1 translation table Block and Page descriptors:
  - Bit[54] holds the PXN instead of the UXN.
  - The Effective value of UXN is 0.
  - Bit[53] is RES0.
  - Bit[6] is treated as 0 regardless of the actual value.
- If Hierarchical Permissions are enabled, the EL1 translation table Table descriptors are as follows:
  - Bit[61] is treated as 0 regardless of the actual value.
  - Bit[60] holds the PXNTable instead of the UXNTable.
  - Bit[59] is RES0.
- When executing at EL1, the Effective value of PSTATE.PAN is 0 for all purposes except reading the value of the bit.
- When executing at EL1, the Effective value of PSTATE.UAO is 1 for all purposes except reading the value of the bit.

If the Effective value of HCR_EL2.{NV1, NV} are {1, 0}, then the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- Behaving as if the Effective value of HCR_EL2.{NV1, NV} is {1, 1} for all purposes other than reading back the value of the HCR_EL2.NV bit.
- Behaving as if the Effective value of HCR_EL2.{NV1, NV} is {0, 0} for all purposes other than reading back the value of the HCR_EL2.NV1 bit.
- Behaving with regard to the Effective value of HCR_EL2.{NV1, NV} behavior as defined in the rest of this description.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_NV is implemented:

Nested Virtualization. EL1 accesses to certain registers are trapped to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NV1	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to VBAR_EL1, ELR_EL1, SPSR_EL1, and, when FEAT_CSV2_2 or FEAT_CSV2_1p2 is implemented, SCXTNUM_EL1, are trapped to EL2, when EL2 is enabled in the current Security state, and are reported using EC syndrome value 0x18.

If the Effective value of HCR_EL2.{NV1, NV} is {0, 1}, then the following effects also apply:

- Any exception taken from EL1, and taken to EL1, causes the SPSR_EL1.M[3:2] to be set to 0b10, and not 0b01.

If the Effective value of HCR_EL2.{NV1, NV} is {1, 1}, then the following effects also apply:

- The EL1 translation table Block and Page descriptors:
  - Bit[54] holds the PXN instead of the UXN.
  - The Effective value of UXN is 0.
  - Bit[53] is RES0.
  - Bit[6] is treated as 0 regardless of the actual value.
- If Hierarchical Permissions are enabled, the EL1 translation table Table descriptors are as follows:
  - Bit[61] is treated as 0 regardless of the actual value.
  - Bit[60] holds the PXNTable instead of the UXNTable.
  - Bit[59] is RES0.
- When executing at EL1, the Effective value of PSTATE.PAN is 0 for all purposes except reading the value of the bit.
- When executing at EL1, the Effective value of PSTATE.UAO is 1 for all purposes except reading the value of the bit.

If the Effective value of HCR_EL2.{NV1, NV} is {1, 0}, then the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- Behaving as if the Effective value of HCR_EL2.{NV1, NV} is {1, 1} for all purposes other than reading back the value of the HCR_EL2.NV bit.
- Behaving as if the Effective value of HCR_EL2.{NV1, NV} is {0, 0} for all purposes other than reading back the value of the HCR_EL2.NV1 bit.
- Behaving with regard to the Effective value of HCR_EL2.{NV1, NV} behavior as defined in the rest of this description.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NV, bit [42]

When FEAT_NV2 is implemented:

Nested Virtualization.

When the Effective value of HCR_EL2.NV2 is 1, redefines register accesses so that:

- Instructions accessing the Special purpose registers SPSR_EL2 and ELR_EL2 instead access SPSR_EL1 and ELR_EL1 respectively.
- Instructions accessing the System registers ESR_EL2 and FAR_EL2 instead access ESR_EL1 and FAR_EL1.

When the Effective value of HCR_EL2.NV2 is 0, traps functionality that is permitted at EL2 and would be UNDEFINED at EL1 if this field was 0, when EL2 is enabled in the current Security state. This applies to the following operations:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL1 accesses to Special-purpose registers that are not UNDEFINED at EL2.
- EL1 accesses to System registers that are not UNDEFINED at EL2.
- Execution of EL2 or EL1 translation regime address translation and TLB maintenance instructions for EL2 and above.

NV	Meaning
0b0	When this bit is set to 0, then the PE behaves as if the Effective value of HCR_EL2.NV2 is 0 for all purposes other than reading this register. This control does not cause any instructions to be trapped. When the Effective value of HCR_EL2.NV2 is 1, no FEAT_NV2 functionality is implemented.
0b1	When the Effective value of HCR_EL2.NV2 is 0, EL1 accesses to the specified registers or the execution of the specified instructions are trapped to EL2, when EL2 is enabled in the current Security state. EL1 read accesses to the CurrentEL register return a value of 0x2. When the Effective value of HCR_EL2.NV2 is 1, this control redefines EL1 register accesses so that instructions accessing SPSR_EL2, ELR_EL2, ESR_EL2, and FAR_EL2 instead access SPSR_EL1, ELR_EL1, ESR_EL1, and FAR_EL1 respectively.

When the Effective value of HCR_EL2.NV2 is 0, then:

- The System or Special-purpose registers for which accesses are trapped and reported using EC syndrome value 0x18 are as follows:
  - Registers accessed using MRS or MSR with a name ending in _EL2, except the following:
    - SP_EL2.
    - If FEAT_MEC is implemented, MECID_A0_EL2, MECID_A1_EL2, MECID_P0_EL2, MECID_P1_EL2, MECIDR_EL2, VMECID_A_EL2, and VMECID_P_EL2.
  - Registers accessed using MRS or MSR with a name ending in _EL12.
  - Registers accessed using MRS or MSR with a name ending in _EL02.
  - Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und, and SPSR_fiq, accessed using MRS or MSR.
  - Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.
- The instructions for which the execution is trapped and reported using EC syndrome value 0x18 are as follows:
  - EL2 translation regime Address Translation instructions and TLB maintenance instructions.
  - EL1 translation regime Address Translation instructions and TLB maintenance instructions that are accessible only from EL2 and EL3.
- The instructions for which the execution is trapped as follows:
  - SMC in an implementation that does not include EL3 and when HCR_EL2.TSC is 1. HCR_EL2.TSC bit is not RES0 in this case. This is reported using EC syndrome value 0x17.
  - The ERET, ERETAA, and ERETAB instructions, reported using EC syndrome value 0x1A.

Note

The priority of this trap is higher than the priority of the HCR_EL2.API trap. If both of these bits are set so that EL1 execution of an ERETAA or ERETAB instruction is trapped to EL2, then the syndrome reported is 0x1A.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_NV is implemented:

Nested Virtualization. Traps functionality that is permitted at EL2 and would be UNDEFINED at EL1 if this field was 0, when EL2 is enabled in the current Security state. This applies to the following operations:

- EL1 accesses to Special-purpose registers that are not UNDEFINED at EL2.
- EL1 accesses to System registers that are not UNDEFINED at EL2.
- Execution of EL2 or EL1 translation regime address translation and TLB maintenance instructions for EL2 and above.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NV	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified registers or the execution of the specified instructions are trapped to EL2, when EL2 is enabled in the current Security state. EL1 read accesses to the CurrentEL register return a value of 0x2.

The System or Special-purpose registers for which accesses are trapped and reported using EC syndrome value 0x18 are as follows:

- Registers accessed using MRS or MSR with a name ending in _EL2, except the following:
  - SP_EL2.
  - If FEAT_MEC is implemented, MECID_A0_EL2, MECID_A1_EL2, MECID_P0_EL2, MECID_P1_EL2, MECIDR_EL2, VMECID_A_EL2, and VMECID_P_EL2.
- Registers accessed using MRS or MSR with a name ending in _EL12.
- Registers accessed using MRS or MSR with a name ending in _EL02.
- Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und, and SPSR_fiq, accessed using MRS or MSR.
- Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.

The instructions for which the execution is trapped and reported using EC syndrome value 0x18 are as follows:

- EL2 translation regime Address Translation instructions and TLB maintenance instructions.
- EL1 translation regime Address Translation instructions and TLB maintenance instructions that are accessible only from EL2 and EL3.

The execution of the ERET, ERETAA, and ERETAB instructions are trapped and reported using EC syndrome value 0x1A.

Note

The priority of this trap is higher than the priority of the HCR_EL2.API trap. If both of these bits are set so that EL1 execution of an ERETAA or ERETAB instruction is trapped to EL2, then the syndrome reported is 0x1A.

The execution of the SMC instructions in an implementation that does not include EL3 and when HCR_EL2.TSC is 1 are trapped and reported using EC syndrome value 0x17. HCR_EL2.TSC bit is not RES0 in this case.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

API, bit [41]

When FEAT_PAuth is implemented:

Controls the use of instructions related to Pointer Authentication:

- PACGA.
- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB.
- PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB.
- RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ERETAA, ERETAB, LDRAA, and LDRAB.
- When FEAT_PAuth_LR is implemented, AUTIASPPC, AUTIASPPCR, AUTIA171615, AUTIBSPPC, AUTIBSPPCR, AUTIB171615, PACIASPPC, PACNBIASPPC, PACIA171615, PACIBSPPC, PACNBIBSPPC, PACIB171615, RETAASPPC, RETAASPPCR, RETABSPPC, RETABSPPCR.

This field is ignored if the instruction is disabled as a result of the SCTLR_ELx.{EnIB, EnIA, EnDA, EnDB} fields.

API	Meaning
`0b0`	The instructions related to Pointer Authentication are trapped to EL2 and reported using EC syndrome value `0x09`, when EL2 is enabled in the current Security state and the instructions are enabled for the EL1&0 translation regime, from: • When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL0. • EL1. If the Effective value of HCR_EL2.NV is 1, the HCR_EL2.NV trap takes precedence over the HCR_EL2.API trap for the `ERETAA` and `ERETAB` instructions. If EL2 is implemented and enabled in the current Security state and the Effective value of HFGITR_EL2.ERET is 1, execution at EL1 using AArch64 of `ERETAA` or `ERETAB` instructions is reported with EC syndrome value `0x1A` with its associated ISS field, as the fine-grained trap has higher priority than the trap enabled by HCR_EL2.API == 0.
`0b1`	This control does not cause any instructions to be trapped.

If FEAT_PAuth is implemented but EL2 is not implemented or is disabled in the current Security state, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APK, bit [40]

When FEAT_PAuth is implemented:

Trap registers holding “key” values for Pointer Authentication. Traps accesses to the following registers from EL1 to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x18`:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1, APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1, APGAKeyLo_EL1, and APGAKeyHi_EL1.

APK	Meaning
`0b0`	Access to the registers holding “key” values for pointer authentication from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.
`0b1`	This control does not cause any instructions to be trapped.

Note

If FEAT_PAuth is implemented but EL2 is not implemented or is disabled in the current Security state, the system behaves as if this bit is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

TEA, bit [37]

When FEAT_RAS is implemented:

Route synchronous External abort exceptions to EL2.

TEA	Meaning
0b0	Synchronous External abort exceptions are unaffected by this mechanism. That is, synchronous External abort exceptions are not taken to EL2 unless routed to EL2 by another control.
0b1	When executing at Exception levels below EL2, and EL2 is enabled in the current Security state, synchronous External abort exceptions are taken to EL2, unless they are routed to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TERR, bit [36]

When FEAT_RAS is implemented:

Trap accesses of Error Record registers. Enables a trap to EL2 on accesses of Error Record registers.

TERR	Meaning
0b0	Accesses of the specified Error Record registers are not trapped by this mechanism.
0b1	Accesses of the specified Error Record registers at EL1 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
- MRS accesses to ERRIDR_EL1 and ERXFR_EL1.
- If FEAT_RASv1p1 is implemented, MRS and MSR accesses to ERXMISC2_EL1 and ERXMISC3_EL1.
- If FEAT_RASv2 is implemented, MRS accesses to ERXGSR_EL1.

In AArch32 state, the instructions affected by this control are:

- MRC and MCR accesses to ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
- MRC accesses to ERRIDR, ERXFR, and ERXFR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_RASv1p1 is implemented, MRC and MCR accesses to ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL2.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Trapped AArch32 instructions are reported using EC syndrome value 0x03.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLOR, bit [35]

When FEAT_LOR is implemented:

Trap LOR registers. Traps Non-secure and Realm EL1 accesses to LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers to EL2.

TLOR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure and Realm EL1 accesses to the LOR registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E2H, bit [34]

When FEAT_VHE is implemented:

EL2 Host. Enables a configuration where a Host Operating System is running at EL2, and the Host Operating System's applications are running at EL0.

E2H	Meaning
0b0	The facilities to support a Host Operating System at EL2 are disabled.
0b1	The facilities to support a Host Operating System at EL2 are enabled.

For information on the behavior of this bit see 'Behavior of HCR_EL2.E2H'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_E2H0 is not implemented, this field is RES1 and behaves as if it is 1 for all purposes other than a direct read.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ID, bit [33]

Stage 2 Instruction access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

ID	Meaning
0b0	This control has no effect on stage 2 of the EL1&0 translation regime.
0b1	Forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable. This applies regardless of the value of HCR_EL2.FWB.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CD, bit [32]

Stage 2 Data access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

CD	Meaning
0b0	This control has no effect on stage 2 of the EL1&0 translation regime for data accesses and translation table walks.
0b1	Forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable. This applies regardless of the value of HCR_EL2.FWB.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RW, bit [31]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AA32EL1 is implemented:

Execution state control for lower Exception levels:

RW	Meaning
0b0	Lower levels are all AArch32.
0b1	The Execution state for EL1 is AArch64. The Execution state for EL0 is determined by the current value of PSTATE.nRW when executing at EL0.

In an implementation that includes EL3, when EL2 is not enabled in Secure state, the PE behaves as if this bit has the same value as the SCR_EL3.RW bit for all purposes other than a direct read or write access of HCR_EL2.

The RW bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAO/WI.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps reads of the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, EL1 accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x18` for `MRS` and `0x14` for `MRRS`:
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, and CONTEXTIDR_EL1.
  - If FEAT_AIE is implemented, MAIR2_EL1 and AMAIR2_EL1.
  - If FEAT_S1PIE is implemented, PIRE0_EL1 and PIR_EL1.
  - If FEAT_S1POE is implemented, POR_EL0 and POR_EL1.
  - If FEAT_S2POE is implemented, S2POR_EL1.
  - If FEAT_TCR2 is implemented, TCR2_EL1.
  - If FEAT_SCTLR2 is implemented, SCTLR2_EL1.
- If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and EL0 is using AArch64, EL0 accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x18` for `MRS`:
  - If FEAT_S1POE is implemented, POR_EL0.
- If EL1 is using AArch32, EL1 accesses using MRC to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`, accesses using MRRC are trapped to EL2 and reported using EC syndrome value `0x04`:
  - SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, and CONTEXTIDR.

TRVM	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Read accesses to the specified Virtual Memory control registers are trapped to EL2, when EL2 is enabled in the current Security state.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the memory system. A hypervisor might use this trap as part of its virtualization of memory management.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HCD, bit [29]

When EL3 is not implemented:

HVC instruction disable. Disables EL1 and EL2 execution of HVC instructions, from both Execution states, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x00`.

HCD	Meaning
`0b0`	HVC instruction execution is enabled at EL2 and EL1.
`0b1`	HVC instructions are UNDEFINED at EL2 and EL1. Any resulting exception is taken to the Exception level at which the HVC instruction is executed.

Note

HVC instructions are always UNDEFINED at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDZ, bit [28]

Traps EL0 and EL1 execution of the following instructions to EL2, when EL2 is enabled in the current Security state, from AArch64 state only, reported using EC syndrome value `0x18`:

- DC ZVA.
- If FEAT_MTE is implemented, DC GVA and DC GZVA.

TDZ	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	In AArch64 state, any attempt to execute an instruction this trap applies to at EL1, or at EL0 when the instruction is not UNDEFINED at EL0, is trapped to EL2 when EL2 is enabled in the current Security state. Reading the DCZID_EL0 returns a value that indicates that the instructions this trap applies to are not supported.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TGE, bit [27]

Trap General Exceptions, from EL0.

TGE	Meaning
`0b0`	This control has no effect on execution at EL0.
`0b1`	When EL2 is not enabled in the current Security state, the Effective value of this field is 0. When EL2 is enabled in the current Security state, in all cases: • All exceptions that would be routed to EL1 are routed to EL2. • If EL1 is using AArch64, the SCTLR_EL1.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1. • If EL1 is using AArch32, the SCTLR.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR. • All virtual interrupts and virtual exceptions are disabled. • Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled. • An exception return to EL1 is treated as an illegal exception return. • The MDCR_EL2.{TDRA, TDOSA, TDA, TDE} fields are treated as being 1 for all purposes other than returning the result of a direct read of MDCR_EL2. In addition, when EL2 is enabled in the current Security state, if: • The Effective value of HCR_EL2.E2H is not 1, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 1. • The Effective value of HCR_EL2.E2H is 1, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 0. For further information on the behavior of this bit when the Effective value of E2H is 1, see 'Behavior of HCR_EL2.E2H'.

HCR_EL2.TGE must not be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TVM, bit [26]

Trap Virtual Memory controls. Traps writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, the following registers are trapped to EL2 and reported using EC syndrome value `0x18` for `MSR` and `0x14` for `MSRR`:
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, and CONTEXTIDR_EL1.
  - If FEAT_AIE is implemented, MAIR2_EL1 and AMAIR2_EL1.
  - If FEAT_S1PIE is implemented, PIRE0_EL1 and PIR_EL1.
  - If FEAT_S1POE is implemented, POR_EL0 and POR_EL1.
  - If FEAT_S2POE is implemented, S2POR_EL1.
  - If FEAT_TCR2 is implemented, TCR2_EL1.
  - If FEAT_SCTLR2 is implemented, SCTLR2_EL1.
- If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and EL0 is using AArch64, EL0 accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x18` for `MSR`:
  - If FEAT_S1POE is implemented, POR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL1 is using AArch32, EL1 accesses using MCR to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`, accesses using MCRR are trapped to EL2 and reported using EC syndrome value `0x04`:
  - SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, and CONTEXTIDR.

TVM	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Write accesses to the specified Virtual Memory control registers are trapped to EL2, when EL2 is enabled in the current Security state.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps execution of TLB maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, then the following instructions are trapped to EL2 and reported using EC syndrome value `0x18`:
  - TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, and TLBI VAALE1.
  - TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, and TLBI VAALE1IS.
  - If FEAT_TLBIOS is implemented, TLBI VMALLE1OS, TLBI VAE1OS, TLBI ASIDE1OS, TLBI VAAE1OS, TLBI VALE1OS, and TLBI VAALE1OS.
  - If FEAT_TLBIRANGE is implemented, TLBI RVAE1, TLBI RVAAE1, TLBI RVALE1, TLBI RVAALE1, TLBI RVAE1IS, TLBI RVAAE1IS, TLBI RVALE1IS, and TLBI RVAALE1IS.
  - If FEAT_TLBIOS and FEAT_TLBIRANGE are implemented, TLBI RVAE1OS, TLBI RVAAE1OS, TLBI RVALE1OS, and TLBI RVAALE1OS.
- If EL1 is using AArch32, then the following instructions are trapped to EL2 and reported using EC syndrome value `0x03`:
  - TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, and TLBIMVAALIS.
  - TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, and TLBIMVAAL
  - ITLBIALL, ITLBIMVA, and ITLBIASID.
  - DTLBIALL, DTLBIMVA, and DTLBIASID.

TTLB	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 execution of the specified TLB maintenance instructions are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

The TLB maintenance instructions are UNDEFINED at EL0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at EL0 and EL1 to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL0 is using AArch64 and the value of SCTLR_EL1.UCI is 1, then the following instructions at EL0 are trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IVAU and DC CVAU.
- If EL1 is using AArch64, then the following instructions at EL1 are trapped to EL2 and reported with EC syndrome value `0x18`:
  - IC IVAU, IC IALLU, IC IALLUIS, and DC CVAU.
- If EL1 is using AArch32, then the following instructions are trapped to EL2 and reported with EC syndrome value `0x03`:
  - ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU.

Note

When SCTLR_EL1.UCI is 0, the trap on execution of instructions at EL0 is higher priority than this control. An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

TPU	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	For each of the specified instructions, if the execution of the instruction can be trapped, access is trapped to EL2, when EL2 is enabled in the current Security state.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TPCP, bit [23]

Trap data or unified cache maintenance instructions that operate to the Point of Coherency, Persistence, or Physical Storage. When EL2 is enabled in the current Security state, traps execution of cache maintenance instructions at EL0 and EL1 to EL2 as follows:

- If EL0 is using AArch64 and the value of SCTLR_EL1.UCI is 1, then the following instructions at EL0 are trapped to EL2 and reported using EC syndrome value `0x18`:
  - DC CIVAC and DC CVAC.
  - If FEAT_MTE is implemented, DC CIGVAC, DC CIGDVAC, DC CGVAC, and DC CGDVAC.
  - If FEAT_DPB is implemented, DC CVAP.
  - If FEAT_DPB and FEAT_MTE are implemented, DC CGVAP and DC CGDVAP.
  - If FEAT_DPB2 is implemented, DC CVADP.
  - If FEAT_DPB2 and FEAT_MTE are implemented, DC CGVADP and DC CGDVADP.
- If FEAT_OCCMO is implemented, DC CIVAOC, DC CIGDVAOC, DC CVAOC and DC CGDVAOC.
- If EL1 is using AArch64, then the following instructions at EL1 are trapped to EL2 and reported using EC syndrome value `0x18`:
  - DC IVAC, DC CIVAC and DC CVAC.
  - If FEAT_MTE is implemented, DC CIGVAC, DC CIGDVAC, DC IGVAC, DC IGDVAC, DC CGVAC, and DC CGDVAC.
  - If FEAT_DPB is implemented, DC CVAP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - If FEAT_DPB and FEAT_MTE are implemented, DC CGVAP and DC CGDVAP.
    - If FEAT_DPB2 is implemented, DC CVADP.
    - If FEAT_DPB2 and FEAT_MTE are implemented, DC CGVADP and DC CGDVADP.
    - If FEAT_PoPS is implemented, DC CIVAPS.
    - If FEAT_PoPS and FEAT_MTE2 are implemented, DC CIGDVAPS.
- If FEAT_OCCMO is implemented, DC CIVAOC, DC CIGDVAOC, DC CVAOC and DC CGDVAOC.
- If EL1 is using AArch32, then the following instructions at EL1 are trapped to EL2 and reported using EC syndrome value `0x03`:
    - DCIMVAC, DCCIMVAC, and DCCMVAC.

Note

This field was previously named TPC.
When SCTLR_EL1.UCI is 0, the trap on execution of instructions at EL0 is higher priority than this control.
An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- AArch64 instructions which invalidate by VA to the Point of Coherency or Physical Storage are always UNDEFINED at EL0 using AArch64.
- DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.

TPCP	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	For each of the specified instructions, if the execution of the instruction can be trapped, it is trapped to EL2, when EL2 is enabled in the current Security state.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, then the following instructions are trapped to EL2 and reported with EC syndrome value `0x18`:
    - DC ISW, DC CSW, and DC CISW.
    - If FEAT_MTE2 is implemented, DC IGSW, DC IGDSW, DC CGSW, DC CGDW, DC CIGSW, and DC CIGDSW.
- If EL1 is using AArch32, then the following instructions are trapped to EL2 and reported with EC syndrome value `0x03`:
    - DCISW, DCCSW, and DCCISW.

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TSW	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TACR, bit [21]

Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, accesses to ACTLR_EL1 to EL2, are trapped to EL2 and reported using EC syndrome value 0x18.
- If EL1 is using AArch32, accesses to ACTLR and, if implemented, ACTLR2 are trapped to EL2 and reported using EC syndrome value 0x03.

TACR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

ACTLR_EL1 is not accessible at EL0.
ACTLR and ACTLR2 are not accessible at EL0.
The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2, when EL2 is enabled in the current Security state as follows:

- In AArch64 state, EL1 access to the encodings in the following reserved encoding spaces are trapped:
  - IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}, and are reported using EC syndrome value 0x18.
  - IMPLEMENTATION DEFINED System instructions, which are accessed using SYSP, with CRn == {11, 15}, and are reported using EC syndrome value 0x14.
  - IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome 0x18.
  - IMPLEMENTATION DEFINED System registers, which are accessed using MRRS and MSRR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome 0x14.
- In AArch32 state, MCR and MRC access to instructions with the following encodings are trapped and reported using EC syndrome value 0x03:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
- All coproc==p15, CRn==c10, opc1 ==={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
- All coproc==p15, CRn==c11, opc1==={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

For accesses to these regions of encoding space by EL0, all of the following apply:

- If FEAT_TIDCP1 is implemented, traps caused by SCTLR_EL1.TIDCP and SCTLR_EL2.TIDCP are reported with higher priority than traps caused by this control.
- If FEAT_TIDCP1 is not implemented, or the access is not trapped by SCTLR_ELx.TIDCP, one of the following applies:
  - If HCR_EL2.TIDCP is 0 then it is IMPLEMENTATION DEFINED, for each encoding, whether the access is performed, is UNDEFINED, or is trapped by an IMPLEMENTATION DEFINED mechanism.
  - If HCR_EL2.TIDCP is 1 then it is IMPLEMENTATION DEFINED, for each encoding, whether the access is trapped to EL2. If it is not trapped it is UNDEFINED.

TIDCP	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2, when EL2 is enabled in the current Security state.

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

Note

The trapping of accesses to these registers from EL1 is higher priority than an exception resulting from the register access being UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TSC, bit [19]

Trap SMC instructions. Traps EL1 execution of SMC instructions to EL2, when EL2 is enabled in the current Security state.

If execution is in AArch64 state, the trap is reported using EC syndrome value `0x17`.

If execution is in AArch32 state, the trap is reported using EC syndrome value `0x13`.

Note

HCR_EL2.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Trap exceptions and SMC exceptions have different preferred return addresses.

TSC	Meaning
`0b0`	This control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TSC	Meaning
0b1	If EL3 is implemented, then any attempt to execute an SMC instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state, regardless of the value of SCR_EL3.SMD. If EL3 is not implemented and the Effective value of HCR_EL2.NV is 1, then any attempt to execute an SMC instruction at EL1 using AArch64 is trapped to EL2. If EL3 is not implemented and the Effective value of HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether: • Any attempt to execute an SMC instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state. • Any attempt to execute an SMC instruction is UNDEFINED.

In AArch32 state, the Armv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

SMC instructions are UNDEFINED at EL0.

If EL3 is not implemented, and the Effective value of HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether this bit is:

- RES0.
- Implemented with the functionality as described in HCR_EL2.TSC.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID3, bit [18]

Trap ID group 3. Traps EL1 reads of group 3 ID registers to EL2, when EL2 is enabled in the current Security state, as follows:

In AArch64 state:

- Reads of the following registers are trapped to EL2, reported using EC syndrome value 0x18:
  - ID_PFR0_EL1, ID_PFR1_EL1, ID_DFR0_EL1, ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
  - ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64AFR0_EL1, and ID_AA64AFR1_EL1.
  - If FEAT_FGT is implemented, reads of the following registers are trapped to EL2. If FEAT_FGT is not implemented, reads of the following registers are trapped to EL2, unless the registers are implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses are trapped to EL2.
    - ID_PFR2_EL1, ID_MMFR4_EL1 and ID_MMFR5_EL1.
    - ID_AA64MMFR3_EL1.
    - ID_AA64MMFR4_EL1.
    - ID_AA64PFR2_EL1.
    - ID_AA64MMFR2_EL1 and ID_ISAR6_EL1.
    - ID_DFR1_EL1.
    - ID_AA64ZFR0_EL1.
    - ID_AA64SMFR0_EL1.
    - ID_AA64ISAR2_EL1.
  - If FEAT_FGT is implemented, this field traps all MRS accesses to registers in the following range that are not already mentioned in this field description: op0 == 3, op1 == 0, CRn == 0, CRm == {2-7}, op2 == {0-7}. If FEAT_FGT is not implemented, it is IMPLEMENTATION DEFINED whether this field traps accesses to registers in the range.

In AArch32 state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- VMRS access to MVFR0, MVFR1, and MVFR2, are trapped to EL2, reported using EC syndrome value `0x08`, unless access is also trapped by HCPTR which takes priority.
- MRC access to the following registers are trapped to EL2, reported using EC syndrome value `0x03`:
  - ID_PFR0, ID_PFR1, ID_PFR2, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
  - If FEAT_FGT is implemented:
    - ID_MMFR4 and ID_MMFR5.
    - ID_ISAR6.
    - ID_DFR1.
    - This field traps all MRC accesses to encodings in the following range that are not already mentioned in this field description: coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.
  - If FEAT_FGT is not implemented:
    - ID_MMFR4 and ID_MMFR5 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 or ID_MMFR5 are trapped.
    - ID_ISAR6 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_ISAR6 are trapped to EL2.
    - ID_DFR1 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1 are trapped to EL2.
    - Otherwise, it is IMPLEMENTATION DEFINED whether this bit traps all MRC accesses to registers in the following range not already mentioned in this field description with coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.

TID3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 read accesses to ID group 3 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID2, bit [17]

Trap ID group 2. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, reads of CTR_EL0, CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL0 is using AArch64 and the value of SCTLR_EL1.UCT is not 0, reads of CTR_EL0 are trapped to EL2, reported using EC syndrome value `0x18`. If the value of SCTLR_EL1.UCT is 0, then EL0 reads of CTR_EL0 are trapped to EL1 and the resulting exception takes precedence over this trap.
- If EL1 is using AArch64, writes to CSSELR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL1 is using AArch32, reads of CTR, CCSIDR, CCSIDR2, CLIDR, and CSSELR are trapped to EL2, reported using EC syndrome value `0x03`.
- If EL1 is using AArch32, writes to CSSELR are trapped to EL2, reported using EC syndrome value `0x03`.

TID2	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 and EL0 accesses to ID group 2 registers are trapped to EL2, when EL2 is enabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID1, bit [16]

Trap ID group 1. Traps EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state as follows:

- In AArch64 state, accesses of REVIDR_EL1 and AIDR_EL1, reported using EC syndrome value `0x18`.
- When FEAT_SME is implemented, accesses of SMIDR_EL1, reported using EC syndrome value `0x18`.
- In AArch32 state, accesses of TCMTR, TLBTR, REVIDR, and AIDR, reported using EC syndrome value `0x03`.

TID1	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 read accesses to ID group 1 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID0, bit [15]

When FEAT_AA32 is implemented:

Trap ID group 0. Traps the following register accesses to EL2:

- EL1 reads of the JIDR, reported using EC syndrome value `0x05`.
- If the JIDR is RAZ from EL0, EL0 reads of the JIDR, reported using EC syndrome value `0x05`.
- EL1 accesses using VMRS of the FPSID, reported using EC syndrome value `0x08`.

Note

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0, then any resulting exception takes precedence over this trap.
- The FPSID is not accessible at EL0 using AArch32.
- Writes to the FPSID are ignored, and not trapped by this control.

TID0	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 read accesses to ID group 0 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

TWE, bit [14]

Traps EL0 and EL1 execution of WFE instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value `0x01`.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

TWE	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt to execute a WFE instruction at EL1 or EL0 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE or SCTLR_EL1.nTWE.

In AArch32 state, the attempted execution of a conditional WFE instruction is trapped only if the instruction passes its condition code check.

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

For more information about when WFE instructions can cause the PE to enter a low-power state, see 'Wait for Event mechanism and Send event'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TWI, bit [13]

Traps EL0 and EL1 execution of WFI instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value `0x01`.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

TWI	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI or SCTLR_EL1.nTWI.

In AArch32 state, the attempted execution of a conditional WFI instruction is trapped only if the instruction passes its condition code check.

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

For more information about when WFI instructions can cause the PE to enter a low-power state, see 'Wait for Interrupt'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DC, bit [12]

Default Cacheability.

DC	Meaning
0b0	This control has no effect on the EL1&0 translation regime.
0b1	In any Security state: • When EL1 is using AArch64, the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1. • When EL1 is using AArch32, the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of SCTLR. • The PE behaves as if the value of the HCR_EL2.VM field is 1 for all purposes other than returning the value of a direct read of HCR_EL2. • The memory type produced by stage 1 of the EL10 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.

This field has no effect on the EL2, EL2&0, and EL3 translation regimes.

This bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from EL1 or EL0:

BSU	Meaning
0b00	No effect.
0b01	Inner Shareable.
0b10	Outer Shareable.
0b11	Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FB, bit [9]

Force broadcast. Causes the following Non-shareable invalidate instructions to be broadcast within the Inner Shareable domain when executed from EL1:

- In AArch32 state: BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALLU, TLBIMVAL, and TLBIMVAAL.
- In AArch64 state:
  - All `TLBI` instructions that are executable at EL1 and do not specify IS or OS.
  - All `TLBIP` instructions that are executable at EL1 and do not specify IS or OS.
  - `IC IALLU`.

For more information, see ‘A64 System instructions for TLB maintenance’.

FB	Meaning
`0b0`	The specified instructions are not affected by this control.
`0b1`	When one of the specified Non-shareable instructions is executed at EL1, the operation is broadcast within the Inner Shareable shareability domain.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VSE, bit [8]

Virtual SError exception.

VSE	Meaning
`0b0`	This mechanism is not making a virtual SError exception pending.
`0b1`	A virtual SError exception is pending because of this mechanism.

The virtual SError exception is enabled only when HCR_EL2.TGE is 0 and either HCR_EL2.AMO is 1 or FEAT_DoubleFault2 is implemented and the Effective value of HCRX_EL2.TMEA is 1.

When FEAT_E3DSE is implemented, virtual SError exceptions pended by this field have priority over delegated SError exceptions pended by SCR_EL3.DSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VI, bit [7]

Virtual IRQ Interrupt.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VI	Meaning
0b0	This mechanism is not making a virtual IRQ pending.
0b1	A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR_EL2.{TGE, IMO} is {0, 1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VF, bit [6]

Virtual FIQ Interrupt.

VF	Meaning
0b0	This mechanism is not making a virtual FIQ pending.
0b1	A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR_EL2.{TGE, FMO} is {0, 1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AMO, bit [5]

Physical SError exception routing.

AMO	Meaning
0b0	When executing at Exception levels below EL3 and the Effective value of HCR_EL2.TGE is 0: • The routing of physical SError exceptions is unaffected by this mechanism. • If FEAT_E3DSE is implemented, then delegated SError exceptions enabled by SCR_EL3.DSE are not taken to EL2. • Virtual SError exceptions are not enabled by this mechanism.
0b1	When executing at Exception levels below EL3, EL2 is enabled in the current Security state, and the Effective value of HCR_EL2.TGE is 0: • Physical SError exceptions are taken to EL2, unless they are routed to EL3. • If FEAT_E3DSE is implemented, then delegated SError exceptions enabled by SCR_EL3.DSE are taken to EL2. • Virtual SError exceptions are enabled.

When executing at EL3, the value of HCR_EL2.AMO has no impact on the behavior of the PE.

When executing at Exception levels below EL3, and EL2 is not enabled in the current Security state, the Effective value of HCR_EL2.AMO is 0.

When the Effective value of HCR_EL2.TGE is 1, regardless of the value of the AMO bit, all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Physical SError exceptions target EL2 unless they are routed to EL3.
- If FEAT_E3DSE is implemented, delegated SError exceptions target EL2.
- Virtual SError exceptions are disabled.

When executing at EL2 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}, it is IMPLEMENTATION DEFINED whether the Effective value of HCR_EL2.AMO is 1 or the value programmed.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMO, bit [4]

Physical IRQ Routing.

IMO	Meaning
`0b0`	When executing at Exception levels below EL3 and the Effective value of HCR_EL2.TGE is 0: • Physical IRQ interrupts are not taken to EL2. • Virtual IRQ interrupts are disabled.
`0b1`	When executing at Exception levels below EL3, EL2 is enabled in the current Security state, and the Effective value of HCR_EL2.TGE is 0: • Physical IRQ exceptions are taken to EL2, unless they are routed to EL3. • Virtual IRQ interrupts are enabled.

When executing at EL3, the Effective value of HCR_EL2.IMO has no impact on the behavior of the PE.

When executing at Exception levels below EL3, and EL2 is not enabled in the current Security state, the Effective value of HCR_EL2.IMO is 0.

When the Effective value of HCR_EL2.TGE is 1, regardless of the value of the IMO bit, all of the following are true:

- Physical IRQ Interrupts target EL2 unless they are routed to EL3.
- Virtual IRQ interrupts are disabled.

When executing at EL2 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}, it is IMPLEMENTATION DEFINED whether the Effective value of HCR_EL2.IMO is 1 or the value programmed.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FMO, bit [3]

Physical FIQ Routing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FMO	Meaning
0b0	When executing at Exception levels below EL3 and the Effective value of HCR_EL2.TGE is 0: • Physical FIQ interrupts are not taken to EL2. • Virtual FIQ interrupts are disabled.
0b1	When executing at Exception levels below EL3, EL2 is enabled in the current Security state, and the Effective value of HCR_EL2.TGE is 0: • Physical FIQ exceptions are taken to EL2, unless they are routed to EL3. • Virtual FIQ interrupts are enabled.

When executing at EL3, the Effective value of HCR_EL2.FMO has no impact on the behavior of the PE.

When executing at Exception levels below EL3, and EL2 is not enabled in the current Security state, the Effective value of HCR_EL2.FMO is 0.

When the Effective value of HCR_EL2.TGE is 1, regardless of the value of the FMO bit, all of the following are true:

- Physical FIQ Interrupts target EL2 unless they are routed to EL3.
- Virtual FIQ interrupts are disabled.

When executing at EL2 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}, it is IMPLEMENTATION DEFINED whether the Effective value of HCR_EL2.FMO is 1 or the value programmed.

For more information, see ‘Asynchronous exception routing’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PTW, bit [2]

Protected Table Walk. In the EL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs, then the value of this bit determines the behavior:

PTW	Meaning
0b0	The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.
0b1	The memory access generates a stage 2 Permission fault.

This bit is permitted to be cached in a TLB.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SWIO, bit [1]

Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SWIO	Meaning
0b0	This control has no effect on the operation of data cache invalidate by set/way instructions.
0b1	Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When the value of this bit is 1:

AArch32: DCISW performs the same invalidation as a DCCISW instruction.

AArch64: DC ISW performs the same invalidation as a DC CISW instruction.

This bit can be implemented as RES1.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the EL1&0 translation regime, when EL2 is enabled in the current Security state.

VM	Meaning
0b0	EL1&0 stage 2 address translation disabled.
0b1	EL1&0 stage 2 address translation enabled.

When the value of this bit is 1, data cache invalidate instructions executed at EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR_EL2.SWIO bit.

This bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x078);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = HCR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = HCR_EL2();
end;
```

```
MSR HCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x078) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HCR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    HCR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.62 HCRX_EL2, Extended Hypervisor Configuration Register

The HCRX_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_HCX is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HCRX_EL2 are UNDEFINED.

Attributes

HCRX_EL2 is a 64-bit register.

Field descriptions

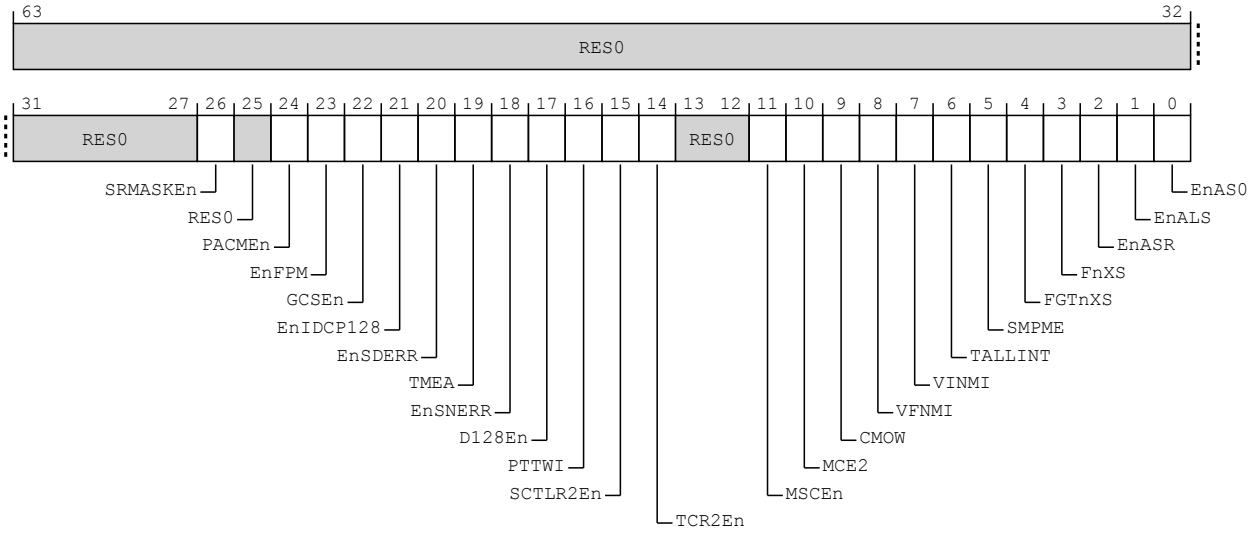

Bits [63:27]

Reserved, RES0.

SRMASKEn, bit [26]

When FEAT_SRMASK is implemented:

If EL1 is using AArch64, accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x18`:

- CPACRMASK_EL1.
- SCTLRMASK_EL1.
- SCTLR2MASK_EL1.
- TCRMASK_EL1.
- TCR2MASK_EL1.
- ACTLRMASK_EL1, if it is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SRMASKEn	Meaning
0b0	EL1 accesses to the specified EL1 registers are trapped to EL2. The values in the registers are treated as 0.
0b1	This control does not cause any instructions to be trapped.

Traps generated by this control have a lower priority than traps generated by the HFGRTR2_EL2 and HFGWTR2_EL2 controls.

When EL2 is not implemented or not enabled in the current Security state, the Effective value of this field is 1.

Otherwise, when the Effective value of SCR_EL3.HXEn is 0, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

PACMEn, bit [24]

When FEAT_PAuth_LR is implemented:

PACM Enable. Controls the effect of a PACM instruction at EL1 and EL0.

PACMEn	Meaning
0b0	The effects of PACM are disabled at EL1 and EL0, regardless of the value in any other controls.
0b1	This control does not disable the effect of PACM at EL1 and EL0.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnFPM, bit [23]

When FEAT_FPMR is implemented:

Enables direct and indirect accesses to FPMR from EL0 and EL1.

When accesses to FPMR are disabled by this control:

- Direct accesses to FPMR from EL0 and EL1 are trapped to EL2 and reported using EC syndrome value `0x18`.
- Execution of FP8 data-processing instructions that indirectly access FPMR is UNDEFINED at EL0 and EL1.

EnFPM	Meaning
`0b0`	Direct and indirect accesses to FPMR are disabled at EL1 and EL0.
`0b1`	This control does not cause any instructions to be trapped.

Traps are not taken if there is a higher priority exception generated by the access.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

GCSEn, bit [22]

When FEAT_GCS is implemented:

Guarded Control Stack enable. Controls Guarded Control Stack behavior at EL1 and EL0.

GCSEn	Meaning
`0b0`	The Guarded Control Stack is disabled at EL1 and EL0.
`0b1`	Guarded Control Stack behavior at EL1 and EL0 is not affected by mechanism.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIDCP128, bit [21]

When FEAT_SYSREG128 is implemented:

Enables access to IMPLEMENTATION DEFINED 128-bit System registers.

EnIDCP128	Meaning
`0b0`	Accesses at EL1, EL0 to IMPLEMENTATION DEFINED 128-bit System registers are trapped to EL2 using EC syndrome value `0x14`, unless the access generates a higher priority exception. Disables the functionality of the 128-bit IMPLEMENTATION DEFINED System registers that are accessible at EL2.
`0b1`	No accesses are trapped by this control.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSDERR, bit [20]

When FEAT_ADERR is implemented:

Enable Synchronous Device Read Error. Override SCTLR2_EL1.EnADERR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnSDERR	Meaning
0b0	This field has no effect on External aborts on Device memory reads in the EL1&0 translation regime.
0b1	External abort on Device memory reads generate synchronous Data Abort exceptions in the EL1&0 translation regime.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 1 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Device memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, HCRX_EL2.EnSDERR is 1, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 1 might have a performance impact for Device memory reads.

This field is ignored by the PE and has an Effective value of 0 when any of the following are true:

- All of the following are true:
  - FEAT_ANERR is implemented.
  - ID_AA64MMFR3_EL1.ADERR reads as `0b0010`.
  - HCRX_EL2.{EnSDERR, EnSNERR} == {1, 0}.
- The Effective value of SCR_EL3.HXEn is 0.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL2 is not implemented or not enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TMEA, bit [19]

When FEAT_DoubleFault2 is implemented:

Trap Masked External Aborts. Controls whether a masked error exception at a lower Exception level is taken to EL2.

TMEA	Meaning
0b0	Synchronous External abort exceptions, physical SError exceptions, and delegated SError exceptions at EL1 and EL0 are unaffected by this mechanism. That is, these exceptions are not taken to EL2 unless routed to EL2 by another control. Virtual SError exceptions are not enabled by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TMEA	Meaning
0b1	When executing at Exception levels below EL2, if EL2 is enabled in the current Security state, then all of the following apply: • When PSTATE.A is 1, synchronous External abort exceptions are taken to EL2, unless they are routed to EL3. • Masked physical SError exceptions are taken to EL2, unless they are routed to EL3. • If FEAT_E3DSE is implemented, then masked delegated SError exceptions enabled by SCR_EL3.DSE are taken to EL2. • If HCR_EL2.TGE is 0, then virtual SError exceptions are enabled.

The Effective value of this field is 0 when any of the following are true:

- The Effective value of SCR_EL3.HXEn is 0.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL2 is not implemented or not enabled in the current Security state.

Virtual SError exceptions are disabled when the Effective value of HCR_EL2.AMO is 0 and the Effective value of HCRX_EL2.TMEA is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSNERR, bit [18]

When FEAT_ANERR is implemented:

Enable Synchronous Normal Read Error. Override SCTLR2_EL1.EnANERR.

EnSNERR	Meaning
0b0	This field has no effect on External aborts on Normal memory reads in the EL1&0 translation regime.
0b1	External abort on Normal memory reads generate synchronous Data Abort exceptions in the EL1&0 translation regime.

Implementation-specific exceptions to applications of this field are described in ‘Taking error exceptions’.

Setting this field to 1 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Normal memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, HCRX_EL2.EnSNERR is 1, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 1 might have a performance impact for Normal memory reads.

This field is ignored by the PE and has an Effective value of 0 when any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following are true:
  - FEAT_ADERR is implemented.
  - ID_AA64MMFR3_EL1.ANERR reads as `0b0010`.
  - HCRX_EL2.{EnSDERR, EnSNERR} == {0, 1}.
- The Effective value of SCR_EL3.HXEn is 0.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL2 is not implemented or not enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D128En, bit [17]

When FEAT_D128 is implemented:

128-bit System Register trap control. Enable access to 128-bit System Registers that relate to VMSAv9-128 via `MRRS`, `MSRR` instructions.

If EL1 is using AArch64, accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x14`:

- TTBR0_EL1.
- TTBR1_EL1.
- If FEAT_THE is implemented, RCWMASK_EL1, RCWSMASK_EL1.
- PAR_EL1.

D128En	Meaning
`0b0`	EL1 accesses to the specified registers are disabled, and trapped to EL2.
`0b1`	This control does not cause any instructions to be trapped.

When EL2 is not implemented or not enabled in the current Security state, the Effective value of this field is 1.

Otherwise, when the Effective value of SCR_EL3.HXEn is 0, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PTTWI, bit [16]

When FEAT_THE is implemented:

Permit Translation Table Walk Incoherence.

Permits RCWS instructions to generate writes that have the Reduced Coherence property.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PTTWI	Meaning
0b0	Write accesses generated by RCWS at EL1 and EL0 do not have the Reduced Coherence property.
0b1	Write accesses generated by RCWS at EL1 and EL0 have the Reduced Coherence property, if enabled by TCR2_EL1.PTTWI.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

This field is permitted to be cached in TLB.

This field is permitted to be implemented as a read-only field with a fixed value of 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SCTLR2En, bit [15]

When FEAT_SCTLR2 is implemented:

SCTLR2_EL1 Enable. In AArch64 state, accesses to SCTLR2_EL1 are trapped to EL2 and reported using EC syndrome value `0x18`.

SCTLR2En	Meaning
0b0	Accesses to SCTLR2_EL1 at EL1 are trapped to EL2, unless the access generates a higher priority exception. The value in SCTLR2_EL1 is treated as 0.
0b1	This control does not cause any instructions to be trapped.

When EL2 is not implemented or not enabled in the current Security state, the Effective value of this field is 1.

Otherwise, when the Effective value of SCR_EL3.HXEn is 0, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCR2En, bit [14]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TCR2 is implemented:

TCR2_EL1 Enable. In AArch64 state, accesses to TCR2_EL1 are trapped to EL2 and reported using EC syndrome value `0x18`.

TCR2En	Meaning
`0b0`	Accesses to TCR2_EL1 at EL1 are trapped to EL2, unless the access generates a higher priority exception. The value in TCR2_EL1 is treated as 0.
`0b1`	This control does not cause any instructions to be trapped.

When EL2 is not implemented or not enabled in the current Security state, the Effective value of this field is 1.

Otherwise, when the Effective value of SCR_EL3.HXEn is 0, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [13:12]

Reserved, RES0.

MSCEn, bit [11]

When FEAT_MOPS is implemented:

Memory Set and Memory Copy instructions Enable. Enables execution of the CPY*, SETG*, SETP*, SETM*, and SETE* instructions at EL1 or EL0.

MSCEn	Meaning
`0b0`	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL1 or EL0.
`0b1`	This control does not cause any instructions to be UNDEFINED.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

MCE2, bit [10]

When FEAT_MOPS is implemented:

Controls Memory Copy and Memory Set exceptions generated as part of attempting to execute the Memory Copy and Memory Set instructions from EL1.

MCE2	Meaning
0b0	Memory Copy and Memory Set exceptions generated from EL1 are taken to EL1.
0b1	Memory Copy and Memory Set exceptions generated from EL1 are taken to EL2.

The Effective value of this field is 0 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CMOW, bit [9]

When FEAT_CMOW is implemented:

Controls the required permissions for the following cache maintenance instructions executed at EL1 or EL0:

- Any DC instruction that operates by VA and performs a clean and invalidate operation.
- Any IC instruction that operates by VA.

CMOW	Meaning
0b0	This control does not cause any instructions to generate a stage 2 Permission fault.
0b1	For these instructions, when executed at EL1 or EL0, the absence of stage 2 write permission generates a stage 2 permission fault.

For more information, see:

- Stage 2 permissions.
- Implications of enabling the dirty state management mechanism.

The Effective value of this field is 0 when any of the following are true:

- The Effective value of SCR_EL3.HXEn is 0.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL2 is not implemented or not enabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VFNMI, bit [8]

When FEAT_NMI is implemented:

Virtual FIQ Interrupt with Superpriority. Enables signaling of virtual FIQ interrupts with Superpriority.

VFNMI	Meaning
`0b0`	When HCR_EL2.VF is 1, a signaled pending virtual FIQ interrupt does not have Superpriority.
`0b1`	When HCR_EL2.VF is 1, a signaled pending virtual FIQ interrupt has Superpriority.

When HCR_EL2.VF is 0, this field has no effect.

The Effective value of this field is 0 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VINMI, bit [7]

When FEAT_NMI is implemented:

Virtual IRQ Interrupt with Superpriority. Enables signaling of virtual IRQ interrupts with Superpriority.

VINMI	Meaning
`0b0`	When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt does not have Superpriority.
`0b1`	When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt has Superpriority.

When HCR_EL2.VI is 0, this field has no effect.

The Effective value of this field is 0 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TALLINT, bit [6]

When FEAT_NMI is implemented:

Traps the following writes at EL1 using AArch64 to EL2, when EL2 is implemented and enabled:

- MSR (register) writes of ALLINT.
- MSR (immediate) writes of ALLINT with a value of 1.

TALLINT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified MSR accesses at EL1 using AArch64 are trapped to EL2.

The Effective value of this field is 0 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SMPME, bit [5]

When FEAT_SME is implemented:

Streaming Mode Priority Mapping Enable.

Controls mapping of the value of SMPRI_EL1.Priority for streaming execution priority at EL0 or EL1.

SMPME	Meaning
`0b0`	The effective priority value is taken from SMPRI_EL1.Priority.
`0b1`	The effective priority value is: • When the current Exception level is EL2 or EL3, the value of SMPRI_EL1.Priority. • When the current Exception level is EL0 or EL1, the value of the SMPRIMAP_EL2 field corresponding to the value of SMPRI_EL1.Priority.

When SMIDR_EL1.SMPS is ‘0’, this field is RES0.

The Effective value of this field is 0 when any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Effective value of SCR_EL3.HXEn is 0.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- EL2 is not implemented or not enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FGTnXS, bit [4]

When FEAT_XS is implemented:

Determines if the fine-grained traps in HFGITR_EL2 that apply to each of the TLBI maintenance instructions that are accessible at EL1 also apply to the corresponding TLBI maintenance instructions with the nXS qualifier.

FGTnXS	Meaning
`0b0`	The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 also applies to the corresponding TLBI instruction with the nXS qualifier at EL1.
`0b1`	The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 does not apply to the corresponding TLBI instruction with the nXS qualifier at EL1.

The Effective value of this field is 0 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnXS, bit [3]

When FEAT_XS is implemented:

Determines the behavior of TLBI instructions affected by the XS attribute.

This control field also determines whether an AArch64 DSB instruction behaves as a DSB instruction with an nXS qualifier when executed at EL0 and EL1.

FnXS	Meaning
`0b0`	This control does not have any effect on the behavior of the TLBI maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FnXS	Meaning
0b1	A TLBI maintenance instruction without the nXS qualifier executed at EL1 behaves in the same way as the corresponding TLBI maintenance instruction with the nXS qualifier. An AArch64 DSB instruction that applies to both loads and stores executed at EL1 or EL0 behaves in the same way as the corresponding DSB instruction with the nXS qualifier executed at EL1 or EL0.

The Effective value of this field is 0 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of SCR_EL3.HXEn is 0.

This field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnASR, bit [2]

When FEAT_LS64_V is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps execution of an ST64BV instruction at EL0 or EL1 to EL2.

EnASR	Meaning
0b0	Execution of an ST64BV instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnASR. Execution of an ST64BV instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using EC syndrome value `0x0A`, with an ISS code of `0x0000000`.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnALS, bit [1]

When FEAT_LS64 is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 or EL1 to EL2.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnALS. Execution of an LD64B or ST64B instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using EC syndrome value `0x0A`, with an ISS code of `0x0000002`.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnAS0, bit [0]

When FEAT_LS64_ACCDATA is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps execution of an ST64BV0 instruction at EL0 or EL1 to EL2.

EnAS0	Meaning
0b0	Execution of an ST64BV0 instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnAS0. Execution of an ST64BV0 instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using EC syndrome value `0x0A`, with an ISS code of `0x0000001`.

The Effective value of this field is 1 when any of the following are true:

- EL2 is not implemented or not enabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Effective value of this field is 0 when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The Effective value of SCR_EL3.HXEn is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HCRX_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HCRX_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_HCX) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x0A0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HXEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HXEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HCRX_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HCRX_EL2();
end;
```

```
MSR HCRX_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_HCX) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x0A0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HXEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HXEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HCRX_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HCRX_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.63 HDBSSBR_EL2, Hardware Dirty State Tracking Structure Base Register

The HDBSSBR_EL2 characteristics are:

Purpose

Control register for HDBSS base address.

Configuration

This register is present only when FEAT_HDBSS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HDBSSBR_EL2 are UNDEFINED.

Attributes

HDBSSBR_EL2 is a 64-bit register.

Field descriptions

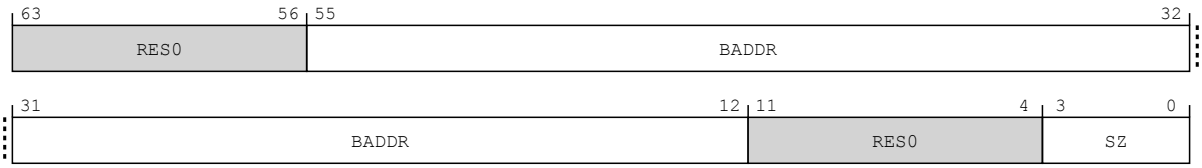

Bits [63:56]

Reserved, RES0.

BADDR, bits [55:12]

HDBSS base address, bits [55:12].

- Bits [55:12] of the base address are the value of this field.
- Bits [11:0] of the base address are 0.

Bits of this field above the implemented physical address size, indicated in ID_AA64MMFR0_EL1.PARange, are RES0.

Based on the value of the SZ field of this register, for encodings of the SZ field greater than 4KB, bits [(SZ+12-1):12] of this field are RES0 such that the base address of the HDBSS is aligned to its size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:4]

Reserved, RES0.

SZ, bits [3:0]

Size of the HDBSS.

SZ	Meaning
0b0000	4KB

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SZ	Meaning
0b0001	8KB
0b0010	16KB
0b0011	32KB
0b0100	64KB
0b0101	128KB
0b0110	256KB
0b0111	512KB
0b1000	1MB
0b1001	2MB

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HDBSSBR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HDBSSBR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_HDBSS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x2E0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HDBSSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HDBSSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HDBSSBR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HDBSSBR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR HDBSSBR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_HDBSS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x2E0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HDBSSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HDBSSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HDBSSBR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HDBSSBR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.64 HDBSSPROD_EL2, Hardware Dirty State Tracking Structure Producer Register

The HDBSSPROD_EL2 characteristics are:

Purpose

Allows producer to update write index for HDBSS.

Configuration

This register is present only when FEAT_HDBSS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HDBSSPROD_EL2 are UNDEFINED.

Attributes

HDBSSPROD_EL2 is a 64-bit register.

Field descriptions

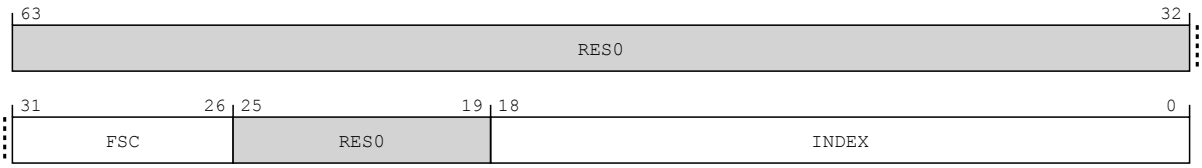

Bits [63:32]

Reserved, RES0.

FSC, bits [31:26]

Fault Status Code.

FSC	Meaning	Applies when
0b000000	The PE has not experienced any error on writes to the HDBSS.	
0b010000	External Abort on write to HDBSS.	
0b101000	Granule Protection Fault on write to HDBSS.	FEAT_RME is implemented

If this field is non-zero then one or more HDBSS writes have experienced a fault since this field was last set to zero, and the associated HDBSS entries have been lost.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [25:19]

Reserved, RES0.

INDEX, bits [18:0]

This field indicates the index of the HDBSS entry that will be written to next.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HDBSSPROD_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HDBSSPROD_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b011

```
if !(IsFeatureImplemented(FEAT_HDBSS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x300);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HDBSSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HDBSSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HDBSSPROD_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HDBSSPROD_EL2();
end;
```

```
MSR HDBSSPROD_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0011	0b011

```
if !(IsFeatureImplemented(FEAT_HDBSS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x300) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().HDBSSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().HDBSSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HDBSSPROD_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HDBSSPROD_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.65 HDFGRTR2_EL2, Hypervisor Debug Fine-Grained Read Trap Register 2

The HDFGRTR2_EL2 characteristics are:

Purpose

Provides controls for traps of `MRS` and `MRC` reads of debug, trace, PMU, and Statistical Profiling System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HDFGRTR2_EL2 are UNDEFINED.

Attributes

HDFGRTR2_EL2 is a 64-bit register.

Field descriptions

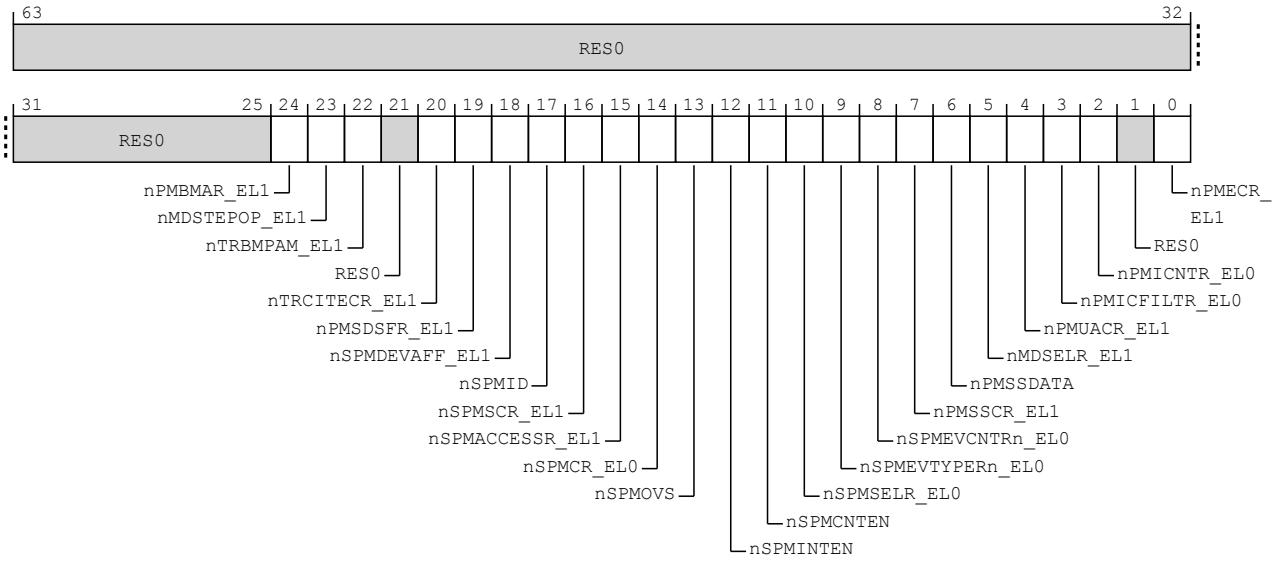

Bits [63:25]

Reserved, RES0.

nPMBMAR_EL1, bit [24]

When FEAT_SPE_nVM is implemented:

Trap `MRS` reads of PMBMAR_EL1 at EL1 using AArch64 to EL2.

nPMBMAR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then `MRS` reads of PMBMAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	MRS reads of PMBMAR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nMDSTEPOP_EL1, bit [23]

When FEAT_STEP2 is implemented:

Trap MRS reads of MDSTEPOP_EL1 at EL1 using AArch64 to EL2.

nMDSTEPOP_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of MDSTEPOP_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of MDSTEPOP_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTRBMPAM_EL1, bit [22]

When FEAT_TRBE_MPAM is implemented:

Trap MRS reads of TRBMPAM_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTRBMPAM_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of TRBMPAM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TRBMPAM_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

nTRCITECR_EL1, bit [20]

When FEAT_ITE is implemented:

Trap MRS reads of TRCITECR_EL1 at EL1 using AArch64 to EL2.

nTRCITECR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of TRCITECR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TRCITECR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMSDSFR_EL1, bit [19]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE_FDS is implemented:

Trap MRS reads of PMSDSFR_EL1 at EL1 using AArch64 to EL2.

nPMSDSFR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of PMSDSFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PMSDSFR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMDEVAFF_EL1, bit [18]

When FEAT_SPMU is implemented:

Trap MRS reads of SPMDEVAFF_EL1 at EL1 using AArch64 to EL2.

nSPMDEVAFF_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SPMDEVAFF_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SPMDEVAFF_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMID, bit [17]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPMU is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMCFGR_EL1.
- SPMCGCR<n>_EL1.
- SPMDEVARCH_EL1.
- SPMIIDR_EL1.

nSPMID	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMSCR_EL1, bit [16]

When FEAT_SPMU is implemented:

Trap MRS reads of SPMSCR_EL1 at EL1 using AArch64 to EL2.

nSPMSCR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SPMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SPMSCR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSPMACCESSR_EL1, bit [15]

When FEAT_SPMU is implemented:

Trap MRS reads of SPMACCESSR_EL1 at EL1 using AArch64 to EL2.

nSPMACCESSR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SPMACCESSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SPMACCESSR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMCR_EL0, bit [14]

When FEAT_SPMU is implemented:

Trap MRS reads of SPMCR_EL0 at EL1 and EL0 using AArch64 to EL2.

nSPMCR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MRS reads of SPMCR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SPMCR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSPMOVS, bit [13]

When FEAT_SPMU is implemented:

Trap MRS reads at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMOVSCLR_EL0.
- SPMOVSSET_EL0.

nSPMOVS	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MRS reads at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMINTEN, bit [12]

When FEAT_SPMU is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMINTENCLR_EL1.
- SPMINTENSET_EL1.

nSPMINTEN	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMCNTEN, bit [11]

When FEAT_SPMU is implemented:

Trap `MRS` reads at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMCNTENCLR_EL0.
- SPMCNTENSET_EL0.

nSPMCNTEN	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then `MRS` reads at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.
`0b1`	`MRS` reads of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMSELR_EL0, bit [10]

When FEAT_SPMU is implemented:

Trap `MRS` reads of SPMSELR_EL0 at EL1 and EL0 using AArch64 to EL2.

nSPMSELR_EL0	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then `MRS` reads of SPMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.
`0b1`	`MRS` reads of SPMSELR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMEVTYPERn_EL0, bit [9]

When FEAT_SPMU is implemented:

Trap MRS reads at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMEVTYPER<n>_EL0.
- SPMEVFILTR<n>_EL0.
- SPMEVFILT2R<n>_EL0.

nSPMEVTYPERn_EL0	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MRS reads at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.
`0b1`	MRS reads of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMEVCNTRn_EL0, bit [8]

When FEAT_SPMU is implemented:

Trap MRS reads of SPMEVCNTR<n>_EL0 at EL1 and EL0 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSPMEVCNTRn_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MRS reads of SPMEVCNTR<n>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SPMEVCNTR<n>_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMSSCR_EL1, bit [7]

When FEAT_PMUv3_SS is implemented:

Trap MRS reads of PMSSCR_EL1 at EL1 using AArch64 to EL2.

nPMSSCR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of PMSSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PMSSCR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMSSDATA, bit [6]

When FEAT_PMUv3_SS is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- PMCCNTSVR_EL1.
- PMEVCNTSVR<n>_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMICNTSVR_EL1, if FEAT_PMUv3_ICNTR is implemented.

nPMSSDATA	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nMDSELR_EL1, bit [5]

When FEAT_Debugv8p9 is implemented:

Trap MRS reads of MDSELR_EL1 at EL1 using AArch64 to EL2.

nMDSELR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of MDSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of MDSELR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMUACR_EL1, bit [4]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3p9 is implemented:

Trap MRS reads of PMUACR_EL1 at EL1 using AArch64 to EL2.

nPMUACR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of PMUACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PMUACR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMICFILTR_EL0, bit [3]

When FEAT_PMUv3_ICNTR is implemented:

Trap MRS reads of PMICFILTR_EL0 at EL1 and EL0 using AArch64 to EL2.

nPMICFILTR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then: • MRS reads of PMICFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception. • PMCNTENCLR_EL0.F0, PMCNTENSET_EL0.F0, PMOVSCLR_EL0.F0, and PMOVSSET_EL0.F0 read as zero at EL1 and EL0. • PMINTENCLR_EL1.F0 and PMINTENSET_EL1.F0 read as zero at EL1.
0b1	MRS reads of PMICFILTR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

nPMICNTR_EL0, bit [2]

When FEAT_PMUv3_ICNTR is implemented:

Trap MRS reads of PMICNTR_EL0 at EL1 and EL0 using AArch64 to EL2.

nPMICNTR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MRS reads of PMICNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PMICNTR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [1]

Reserved, RES0.

nPMECR_EL1, bit [0]

When FEAT_EBEP is implemented or FEAT_PMUv3_SS is implemented:

Trap MRS reads of PMECR_EL1 at EL1 using AArch64 to EL2.

nPMECR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of PMECR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PMECR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HDFGRTR2_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HDFGRTR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1A0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HDFGRTR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HDFGRTR2_EL2();
end;
```

```
MSR HDFGRTR2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        NVMem(0x1A0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HDFGRTR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HDFGRTR2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.66 HDFGRTR_EL2, Hypervisor Debug Fine-Grained Read Trap Register

The HDFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MRS` and `MRC` reads of debug, trace, PMU, and Statistical Profiling System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HDFGRTR_EL2 are UNDEFINED.

Attributes

HDFGRTR_EL2 is a 64-bit register.

Field descriptions

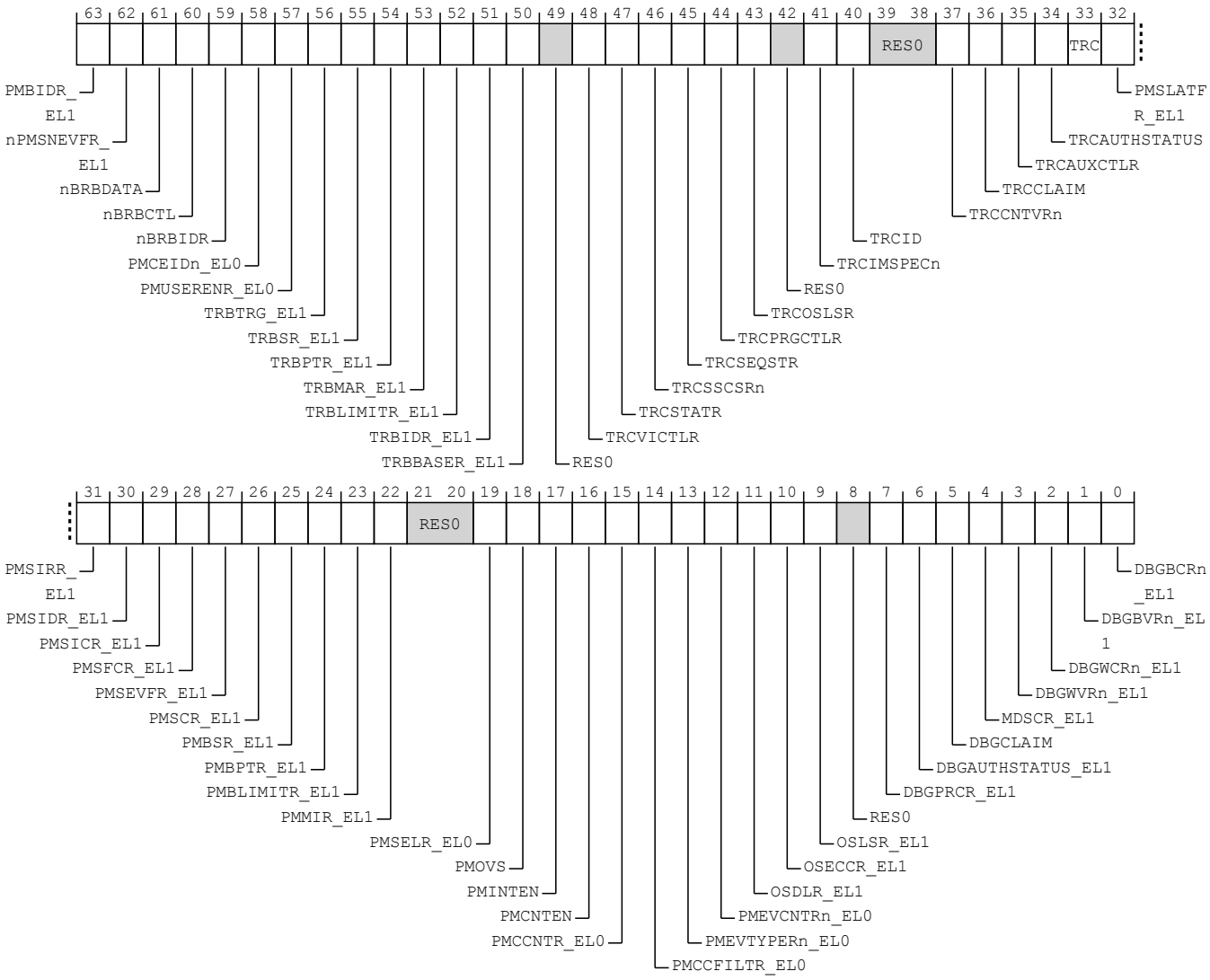

PMBIDR_EL1, bit [63]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE is implemented:

Trap MRS reads of PMBIDR_EL1 at EL1 using AArch64 to EL2.

PMBIDR_EL1	Meaning
0b0	MRS reads of PMBIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMSNEVFR_EL1, bit [62]

When FEAT_SPE_FnE is implemented:

Trap MRS reads of PMSNEVFR_EL1 at EL1 using AArch64 to EL2.

nPMSNEVFR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSNEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PMSNEVFR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBDATA, bit [61]

When FEAT_BRBE is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- BRBINF<n>_EL1.
- BRBINFINJ_EL1.
- BRBSRC<n>_EL1.
- BRBSRCINJ_EL1.
- BRBTGT<n>_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BRBTGTINJ_EL1.
- BRBTS_EL1.

nBRBDATA	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBCTL, bit [60]

When FEAT_BRBE is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- BRBCR_EL1.
- BRBFCR_EL1.

nBRBCTL	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBIDR, bit [59]

When FEAT_BRBE is implemented:

Trap MRS reads of BRBIDR0_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nBRBIDR	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of BRBIDR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of BRBIDR0_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCEIDn_EL0, bit [58]

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCEID<n> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMCEIDn_EL0	Meaning
0b0	MRS reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCEID<n> at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of PMCEID<n> at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMUSERENR_EL0, bit [57]

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMUSERENR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMUSERENR_EL0	Meaning
0b0	MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMUSERENR at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of PMUSERENR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBTRG_EL1, bit [56]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBTRG_EL1 at EL1 using AArch64 to EL2.

TRBTRG_EL1	Meaning
0b0	MRS reads of TRBTRG_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBTRG_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBSR_EL1, bit [55]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBSR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRBSR_EL1	Meaning
0b0	MRS reads of TRBSR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBPTR_EL1, bit [54]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBPTR_EL1 at EL1 using AArch64 to EL2.

TRBPTR_EL1	Meaning
0b0	MRS reads of TRBPTR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBMAR_EL1, bit [53]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBMAR_EL1 at EL1 using AArch64 to EL2.

TRBMAR_EL1	Meaning
0b0	MRS reads of TRBMAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBMAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBLIMITR_EL1, bit [52]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBLIMITR_EL1 at EL1 using AArch64 to EL2.

TRBLIMITR_EL1	Meaning
`0b0`	MRS reads of TRBLIMITR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBIDR_EL1, bit [51]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBIDR_EL1 at EL1 using AArch64 to EL2.

TRBIDR_EL1	Meaning
`0b0`	MRS reads of TRBIDR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBBASER_EL1, bit [50]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TRBE is implemented:

Trap MRS reads of TRBBASER_EL1 at EL1 using AArch64 to EL2.

TRBBASER_EL1	Meaning
0b0	MRS reads of TRBBASER_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBBASER_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [49]

Reserved, RES0.

TRCVICTLR, bit [48]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCVICTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCVICTLR at EL1 using AArch64 to EL2.

TRCVICTLR	Meaning
0b0	MRS reads of TRCVICTLR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCVICTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCSTATR, bit [47]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an Armv9 implementation, trap MRS reads of TRCSTATR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCSTATR at EL1 using AArch64 to EL2.

TRCSTATR	Meaning
0b0	MRS reads of TRCSTATR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCSTATR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCSSCSRn, bit [46]

When IsFeatureImplemented(FEAT_ETE) || ((IsFeatureImplemented(FEAT_ETMv4) && TRCSSCSR<n> are implemented) && IsFeatureImplemented(FEAT_TRC_SR)):

In an Armv9 implementation, trap MRS reads of TRCSSCSR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCSSCSR<n> at EL1 using AArch64 to EL2.

TRCSSCSRn	Meaning
0b0	MRS reads of TRCSSCSR<n> are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCSSCSR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If Single-shot Comparator n is not implementented, a read of TRCSSCSR<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCSEQSTR, bit [45]

When IsFeatureImplemented(FEAT_ETE) || ((IsFeatureImplemented(FEAT_ETMv4) && TRCSEQSTR is implemented) && IsFeatureImplemented(FEAT_TRC_SR)):

In an Armv9 implementation, trap MRS reads of TRCSEQSTR at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an Armv8 implementation, trap MRS reads of ETM TRCSEQSTR at EL1 using AArch64 to EL2.

TRCSEQSTR	Meaning
0b0	MRS reads of TRCSEQSTR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCPRGCTLR, bit [44]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCPRGCTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCPRGCTLR at EL1 using AArch64 to EL2.

TRCPRGCTLR	Meaning
0b0	MRS reads of TRCPRGCTLR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCPRGCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCOSLSR, bit [43]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCOSLSR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCOSLSR at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCOSLSR	Meaning
0b0	MRS reads of TRCOSLSR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCOSLSR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

TRCIMSPECn, bit [41]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCIMSPEC<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCIMSPEC<n> at EL1 using AArch64 to EL2.

TRCIMSPECn	Meaning
0b0	MRS reads of TRCIMSPEC<n> are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCIMSPEC<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

TRCIMSPEC<1-7> are optional. If TRCIMSPEC<n> is not implemented, a read of TRCIMSPEC<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCID, bit [40]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- In an Armv9 implementation:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCDEVARCH.
- TRCDEVID.
- All of the TRCIDR<n> registers.
- In an Armv8 implementation:
  - ETM TRCDEVARCH.
  - ETM TRCDEVID.
  - All of the ETM TRCIDR<n> registers.

TRCID	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

TRCCNTVRn, bit [37]

When IsFeatureImplemented(FEAT_ETE) || ((IsFeatureImplemented(FEAT_ETMv4) && TRCCNTVR<n> are implemented) && IsFeatureImplemented(FEAT_TRC_SR)):

In an Armv9 implementation, trap MRS reads of TRCCNTVR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCCNTVR<n> at EL1 using AArch64 to EL2.

TRCCNTVRn	Meaning
0b0	MRS reads of TRCCNTVR<n> are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCCNTVR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If Counter n is not implemented, a read of TRCCNTVR<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCCLAIM, bit [36]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- In an Armv9 implementation: TRCCLAIMCLR and TRCCLAIMSET.
- In an Armv8 implementation: ETM TRCCLAIMCLR and ETM TRCCLAIMSET.

TRCCLAIM	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCAUXCTLR, bit [35]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCAUXCTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCAUXCTLR at EL1 using AArch64 to EL2.

TRCAUXCTLR	Meaning
0b0	MRS reads of TRCAUXCTLR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCAUXCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCAUTHSTATUS, bit [34]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap `MRS` reads of TRCAUTHSTATUS at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap `MRS` reads of ETM TRCAUTHSTATUS at EL1 using AArch64 to EL2.

TRCAUTHSTATUS	Meaning
`0b0`	`MRS` reads of TRCAUTHSTATUS are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then `MRS` reads of TRCAUTHSTATUS at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRC, bit [33]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

Trap `MRS` reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- In an Armv9 implementation:
  - TRCACATR<n>.
  - TRCACVR<n>.
  - TRCBBCTLR.
  - TRCCCCTLR.
  - TRCCIDCCTLR0.
  - TRCCIDCCTLR1.
  - TRCCIDCVR<n>.
  - TRCCNTCTLR<n>.
  - TRCCNTRLDVR<n>.
  - TRCCONFIGR.
  - TRCEVENTCTL0R.
  - TRCEVENTCTL1R.
  - TRCEXTINSELR<n>.
  - TRCQCTLR.
  - TRCRSCTLR<n>.
  - TRCRSR.
  - TRCSEQEVR<n>.
  - TRCSEQRSTEVR.
  - TRCSSCCR<n>.
  - TRCSSPCICR<n>.
  - TRCSTALLCTLR.
  - TRCSYNCPR.
  - TRCTRACEIDR.
  - TRCTSCTLR.
  - TRCVIIECTLR.
  - TRCVIPCSSCTLR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCVISSCTLR.
- TRCVMIDCCTLR0.
- TRCVMIDCCTLR1.
- TRCVMIDCVR<n>.

- In an Armv8 implementation:
  - ETM TRCACATR<n>.
  - ETM TRCACVR<n>.
  - ETM TRCBBCTLR.
  - ETM TRCCCCTLR.
  - ETM TRCCIDCCTLR0.
  - ETM TRCCIDCCTLR1.
  - ETM TRCCIDCVR<n>.
  - ETM TRCCNTCTLR<n>.
  - ETM TRCCNTRLDVR<n>.
  - ETM TRCCONFIGR.
  - ETM TRCEVENTCTL0R.
  - ETM TRCEVENTCTL1R.
  - ETM TRCEXTINSELR.
  - ETM TRCQCTLR.
  - ETM TRCRSCTLR<n>.
  - ETM TRCSEQEVR<n>.
  - ETM TRCSEQRSTEVR.
  - ETM TRCSSCCR<n>.
  - ETM TRCSSPCICR<n>.
  - ETM TRCSTALLCTLR.
  - ETM TRCSYNCPR.
  - ETM TRCTRACEIDR.
  - ETM TRCTSCTLR.
  - ETM TRCVIIECTLR.
  - ETM TRCVIPCSSCTLR.
  - ETM TRCVISSCTLR.
  - ETM TRCVMIDCCTLR0.
  - ETM TRCVMIDCCTLR1.
  - ETM TRCVMIDCVR<n>.

TRC	Meaning
`0b0`	`MRS` reads of the specified System registers are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then `MRS` reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

A read of an unimplemented register is UNDEFINED.

TRCEXTINSELR<n> and TRCRSR are implemented only if FEAT_ETE is implemented.

TRCEXTINSELR is implemented only if FEAT_ETE is not implemented and FEAT_ETMv4 is implemented.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSLATFR_EL1, bit [32]

When FEAT_SPE is implemented:

Trap MRS reads of PMSLATFR_EL1 at EL1 using AArch64 to EL2.

PMSLATFR_EL1	Meaning
0b0	MRS reads of PMSLATFR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSLATFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSIRR_EL1, bit [31]

When FEAT_SPE is implemented:

Trap MRS reads of PMSIRR_EL1 at EL1 using AArch64 to EL2.

PMSIRR_EL1	Meaning
0b0	MRS reads of PMSIRR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSIRR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSIDR_EL1, bit [30]

When FEAT_SPE is implemented:

Trap MRS reads of PMSIDR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSIDR_EL1	Meaning
0b0	MRS reads of PMSIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSICR_EL1, bit [29]

When FEAT_SPE is implemented:

Trap MRS reads of PMSICR_EL1 at EL1 using AArch64 to EL2.

PMSICR_EL1	Meaning
0b0	MRS reads of PMSICR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSICR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSFCR_EL1, bit [28]

When FEAT_SPE is implemented:

Trap MRS reads of PMSFCR_EL1 at EL1 using AArch64 to EL2.

PMSFCR_EL1	Meaning
0b0	MRS reads of PMSFCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSEVFR_EL1, bit [27]

When FEAT_SPE is implemented:

Trap MRS reads of PMSEVFR_EL1 at EL1 using AArch64 to EL2.

PMSEVFR_EL1	Meaning
`0b0`	MRS reads of PMSEVFR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSCR_EL1, bit [26]

When FEAT_SPE is implemented:

Trap MRS reads of PMSCR_EL1 at EL1 using AArch64 to EL2.

PMSCR_EL1	Meaning
`0b0`	MRS reads of PMSCR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMBSR_EL1, bit [25]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE is implemented:

Trap MRS reads of PMBSR_EL1 at EL1 using AArch64 to EL2.

PMBSR_EL1	Meaning
0b0	MRS reads of PMBSR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMBPTR_EL1, bit [24]

When FEAT_SPE is implemented:

Trap MRS reads of PMBPTR_EL1 at EL1 using AArch64 to EL2.

PMBPTR_EL1	Meaning
0b0	MRS reads of PMBPTR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMBLIMITR_EL1, bit [23]

When FEAT_SPE is implemented:

Trap MRS reads of PMBLIMITR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMBLIMITR_EL1	Meaning
0b0	MRS reads of PMBLIMITR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMMIR_EL1, bit [22]

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMMIR_EL1 at EL1 using AArch64 to EL2.

PMMIR_EL1	Meaning
0b0	MRS reads of PMMIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMMIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [21:20]

Reserved, RES0.

PMSELR_EL0, bit [19]

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMSELR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMSELR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSELR_EL0	Meaning
0b0	MRS reads of PMSELR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMSELR at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of PMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of PMSELR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMOVS, bit [18]

When FEAT_PMUv3 is implemented:

Trap MRS reads and MRC reads of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMOVSCLR_EL0 and PMOVSSET_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMOVSR and PMOVSSET.

PMOVS	Meaning
0b0	MRS reads at EL1 and EL0 using AArch64 and MRC reads at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMINTEN, bit [17]

When FEAT_PMUv3 is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMINTENCLR_EL1.
- PMINTENSET_EL1.

PMINTEN	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCNTEN, bit [16]

When FEAT_PMUv3 is implemented:

Trap MRS reads and MRC reads of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMCNTENCLR_EL0 and PMCNTENSET_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMCNTENCLR and PMCNTENSET.

PMCNTEN	Meaning
0b0	MRS reads at EL1 and EL0 using AArch64 and MRC reads at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCCNTR_EL0, bit [15]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MRC and MRRC reads of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMCCNTR_EL0	Meaning
0b0	MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MRC and MRRC reads of PMCCNTR at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03. • MRRC reads of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x04.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCCFILTR_EL0, bit [14]

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMCCFILTR_EL0	Meaning
0b0	MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCCFILTR at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

PMCCFILTR_EL0 can also be accessed in AArch64 state using PMXEVTYPER_EL0 when PMSELR_EL0.SEL == 31, and PMCCFILTR can also be accessed in AArch32 state using PMXEVTYPER when PMSELR.SEL == 31.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Setting this field to 1 has no effect on accesses to PMXEVTYPER_EL0 and PMXEVTYPER, regardless of the value of PMSELR_EL0.SEL or PMSELR.SEL.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMEVTYPERn_EL0, bit [13]

When FEAT_PMUv3 is implemented:

Trap `MRS` reads and `MRC` reads of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMEVTYPER<n> and PMXEVTYPER.

PMEVTYPERn_EL0	Meaning
`0b0`	`MRS` reads at EL1 and EL0 using AArch64 and `MRC` reads at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • `MRS` reads at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value `0x18`. • `MRC` reads at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value `0x03`.

Regardless of the value of this field, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a read of PMEVTYPER<n>_EL0, or, if n is not 31, a read of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n.
  - In AArch32 state, a read of PMEVTYPER<n>, or, if n is not 31, a read of PMXEVTYPER when PMSELR.SEL == n.
- If event counter n is implemented, n is greater-than-or-equal-to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a read of PMEVTYPER<n>_EL0, or a read of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a read of PMEVTYPER<n>, or a read of PMXEVTYPER when PMSELR.SEL == n, reported with EC syndrome value `0x03`.

See also HDFGRTR_EL2.PMCCFILTR_EL0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

PMEVCNTRn_EL0, bit [12]

When FEAT_PMUv3 is implemented:

Trap MRS reads and MRC reads of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMEVCNTR<n> and PMXEVCNTR.

PMEVCNTRn_EL0	Meaning
0b0	MRS reads at EL1 and EL0 using AArch64 and MRC reads at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value 0x03.

Regardless of the value of this field, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a read of PMEVCNTR<n>_EL0, or a read of PMXEVCNTR_EL0 when PMSELR_EL0.SEL is n.
  - In AArch32 state, a read of PMEVCNTR<n>, or a read of PMXEVCNTR when PMSELR.SEL is n.
- If event counter n is implemented, n is greater than or equal to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a read of PMEVCNTR<n>_EL0, or a read of PMXEVCNTR_EL0 when PMSELR_EL0.SEL is n, reported with EC syndrome value 0x18.
  - In AArch32 state, a read of PMEVCNTR<n>, or a read of PMXEVCNTR when PMSELR.SEL is n, reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

OSDLR_EL1, bit [11]

When FEAT_DoubleLock is implemented:

Trap MRS reads of OSDLR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSDLR_EL1	Meaning
0b0	MRS reads of OSDLR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of OSDLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

OSECCR_EL1, bit [10]

Trap MRS reads of OSECCR_EL1 at EL1 using AArch64 to EL2.

OSECCR_EL1	Meaning
0b0	MRS reads of OSECCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of OSECCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

OSLSR_EL1, bit [9]

Trap MRS reads of OSLSR_EL1 at EL1 using AArch64 to EL2.

OSLSR_EL1	Meaning
0b0	MRS reads of OSLSR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of OSLSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [8]

Reserved, RES0.

DBGPRCR_EL1, bit [7]

Trap MRS reads of DBGPRCR_EL1 at EL1 using AArch64 to EL2.

DBGPRCR_EL1	Meaning
0b0	MRS reads of DBGPRCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGPRCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGAUTHSTATUS_EL1, bit [6]

Trap MRS reads of DBGAUTHSTATUS_EL1 at EL1 using AArch64 to EL2.

DBGAUTHSTATUS_EL1	Meaning
0b0	MRS reads of DBGAUTHSTATUS_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGAUTHSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGCLAIM, bit [5]

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- DBGCLAIMCLR_EL1.
- DBGCLAIMSET_EL1.

DBGCLAIM	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DBGCLAIM	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

MDSCR_EL1, bit [4]

Trap MRS reads of MDSCR_EL1 at EL1 using AArch64 to EL2.

MDSCR_EL1	Meaning
0b0	MRS reads of MDSCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MDSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGWVRn_EL1, bit [3]

Trap MRS reads of DBGWVR<n>_EL1 at EL1 using AArch64 to EL2.

DBGWVRn_EL1	Meaning
0b0	MRS reads of DBGWVR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGWVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If watchpoint n is not implemented, a read of DBGWVR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGWCRn_EL1, bit [2]

Trap MRS reads of DBGWCR<n>_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DBGWCRn_EL1	Meaning
0b0	MRS reads of DBGWCR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGWCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If watchpoint n is not implemented, a read of DBGWCR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGBVRn_EL1, bit [1]

Trap MRS reads of DBGBVR<n>_EL1 at EL1 using AArch64 to EL2.

DBGBVRn_EL1	Meaning
0b0	MRS reads of DBGBVR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGBVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If breakpoint n is not implemented, a read of DBGBVR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGBCRn_EL1, bit [0]

Trap MRS reads of DBGBCR<n>_EL1 at EL1 using AArch64 to EL2.

DBGBCRn_EL1	Meaning
0b0	MRS reads of DBGBCR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGBCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If breakpoint n is not implemented, a read of DBGBCR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing HDFGRTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HDFGRTR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1D0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HDFGRTR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HDFGRTR_EL2();
end;
```

```
MSR HDFGRTR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1D0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HDFGRTR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HDFGRTR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.67 HDFGWTR2_EL2, Hypervisor Debug Fine-Grained Write Trap Register 2

The HDFGWTR2_EL2 characteristics are:

Purpose

Provides controls for traps of MSR and MCR writes of debug, trace, PMU, and Statistical Profiling System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HDFGWTR2_EL2 are UNDEFINED.

Attributes

HDFGWTR2_EL2 is a 64-bit register.

Field descriptions

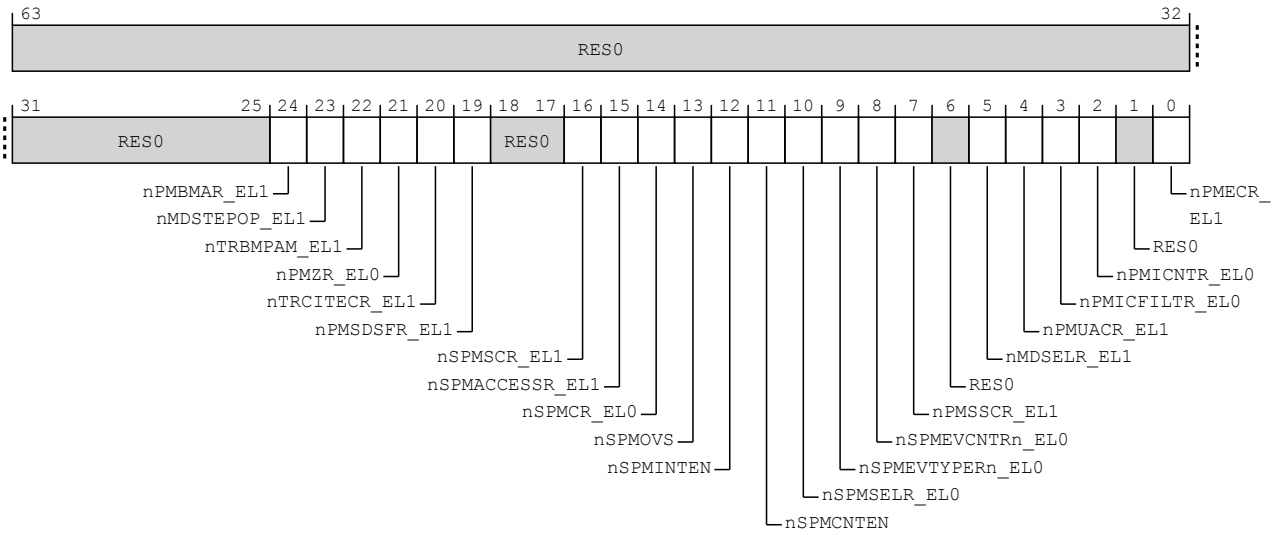

Bits [63:25]

Reserved, RES0.

nPMBMAR_EL1, bit [24]

When FEAT_SPE_nVM is implemented:

Trap MSR writes of PMBMAR_EL1 at EL1 using AArch64 to EL2.

nPMBMAR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of PMBMAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of PMBMAR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nMDSTEPOP_EL1, bit [23]

When FEAT_STEP2 is implemented:

Trap MSR writes of MDSTEPOP_EL1 at EL1 using AArch64 to EL2.

nMDSTEPOP_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of MDSTEPOP_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of MDSTEPOP_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTRBMPAM_EL1, bit [22]

When FEAT_TRBE_MPAM is implemented:

Trap MSR writes of TRBMPAM_EL1 at EL1 using AArch64 to EL2.

nTRBMPAM_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of TRBMPAM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of TRBMPAM_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMZR_EL0, bit [21]

When FEAT_PMUv3p9 is implemented:

Trap MSR writes of PMZR_EL0 at EL1 and EL0 using AArch64 to EL2.

nPMZR_EL0	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes of PMZR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of PMZR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTRCITECR_EL1, bit [20]

When FEAT_ITE is implemented:

Trap MSR writes of TRCITECR_EL1 at EL1 using AArch64 to EL2.

nTRCITECR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of TRCITECR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of TRCITECR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMSDSFR_EL1, bit [19]

When FEAT_SPE_FDS is implemented:

Trap MSR writes of PMSDSFR_EL1 at EL1 using AArch64 to EL2.

nPMSDSFR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of PMSDSFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of PMSDSFR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [18:17]

Reserved, RES0.

nSPMSCR_EL1, bit [16]

When FEAT_SPMU is implemented:

Trap MSR writes of SPMSCR_EL1 at EL1 using AArch64 to EL2.

nSPMSCR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of SPMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	MSR writes of SPMSCR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMACCESSR_EL1, bit [15]

When FEAT_SPMU is implemented:

Trap MSR writes of SPMACCESSR_EL1 at EL1 using AArch64 to EL2.

nSPMACCESSR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of SPMACCESSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SPMACCESSR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMCR_EL0, bit [14]

When FEAT_SPMU is implemented:

Trap MSR writes of SPMCR_EL0 at EL1 and EL0 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSPMCR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes of SPMCR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SPMCR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMOVS, bit [13]

When FEAT_SPMU is implemented:

Trap MSR writes at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMOVSCLR_EL0.
- SPMOVSSET_EL0.

nSPMOVS	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMINTEN, bit [12]

When FEAT_SPMU is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SPMINTENCLR_EL1.
- SPMINTENSET_EL1.

nSPMINTEN	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMCNTEN, bit [11]

When FEAT_SPMU is implemented:

Trap MSR writes at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMCNTENCLR_EL0.
- SPMCNTENSET_EL0.

nSPMCNTEN	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSPMSELR_EL0, bit [10]

When FEAT_SPMU is implemented:

Trap MSR writes of SPMSELR_EL0 at EL1 and EL0 using AArch64 to EL2.

nSPMSELR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes of SPMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SPMSELR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMEVTYPERn_EL0, bit [9]

When FEAT_SPMU is implemented:

Trap MSR writes at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMEVTYPER<n>_EL0.
- SPMEVFILTR<n>_EL0.
- SPMEVFILT2R<n>_EL0.

nSPMEVTYPERn_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSPMEVCNTRn_EL0, bit [8]

When FEAT_SPMU is implemented:

Trap MSR writes at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- SPMEVCNTR<n>_EL0.
- SPMZR_EL0.

nSPMEVCNTRn_EL0	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then MSR writes at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of the specified System registers are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMSSCR_EL1, bit [7]

When FEAT_PMUv3_SS is implemented:

Trap MSR writes of PMSSCR_EL1 at EL1 using AArch64 to EL2.

nPMSSCR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of PMSSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of PMSSCR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [6]

Reserved, RES0.

nMDSELR_EL1, bit [5]

When FEAT_Debugv8p9 is implemented:

Trap MSR writes of MDSELR_EL1 at EL1 using AArch64 to EL2.

nMDSELR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of MDSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of MDSELR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMUACR_EL1, bit [4]

When FEAT_PMUv3p9 is implemented:

Trap MSR writes of PMUACR_EL1 at EL1 using AArch64 to EL2.

nPMUACR_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of PMUACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nPMUACR_EL1	Meaning
0b1	MSR writes of PMUACR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMICFILTR_EL0, bit [3]

When FEAT_PMUv3_ICNTR is implemented:

Trap MSR writes of PMICFILTR_EL0 at EL1 and EL0 using AArch64 to EL2.

nPMICFILTR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then: • MSR writes of PMICFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception. • PMCNTENCLR_EL0.F0, PMCNTENSET_EL0.F0, PMOVSCLR_EL0.F0, and PMOVSSET_EL0.F0 ignore writes at EL1 and EL0. • PMINTENCLR_EL1.F0 and PMINTENSET_EL1.F0 ignore writes at EL1.
0b1	MSR writes of PMICFILTR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPMICNTR_EL0, bit [2]

When FEAT_PMUv3_ICNTR is implemented:

Trap MSR writes of PMICNTR_EL0 at EL1 and EL0 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nPMICNTR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then: • MSR writes of PMICNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception. • PMZR_EL0.F0 ignores writes at EL1 and EL0.
0b1	MSR writes of PMICNTR_EL0 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [1]

Reserved, RES0.

nPMECR_EL1, bit [0]

When FEAT_EBEP is implemented or FEAT_PMUv3_SS is implemented:

Trap MSR writes of PMECR_EL1 at EL1 using AArch64 to EL2.

nPMECR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of PMECR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of PMECR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HDFGWTR2_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, HDFGWTR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1B0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HDFGWTR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HDFGWTR2_EL2();
end;
```

```
MSR HDFGWTR2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1B0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HDFGWTR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HDFGWTR2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.68 HDFGWTR_EL2, Hypervisor Debug Fine-Grained Write Trap Register

The HDFGWTR_EL2 characteristics are:

Purpose

Provides controls for traps of MSR and MCR writes of debug, trace, PMU, and Statistical Profiling System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HDFGWTR_EL2 are UNDEFINED.

Attributes

HDFGWTR_EL2 is a 64-bit register.

Field descriptions

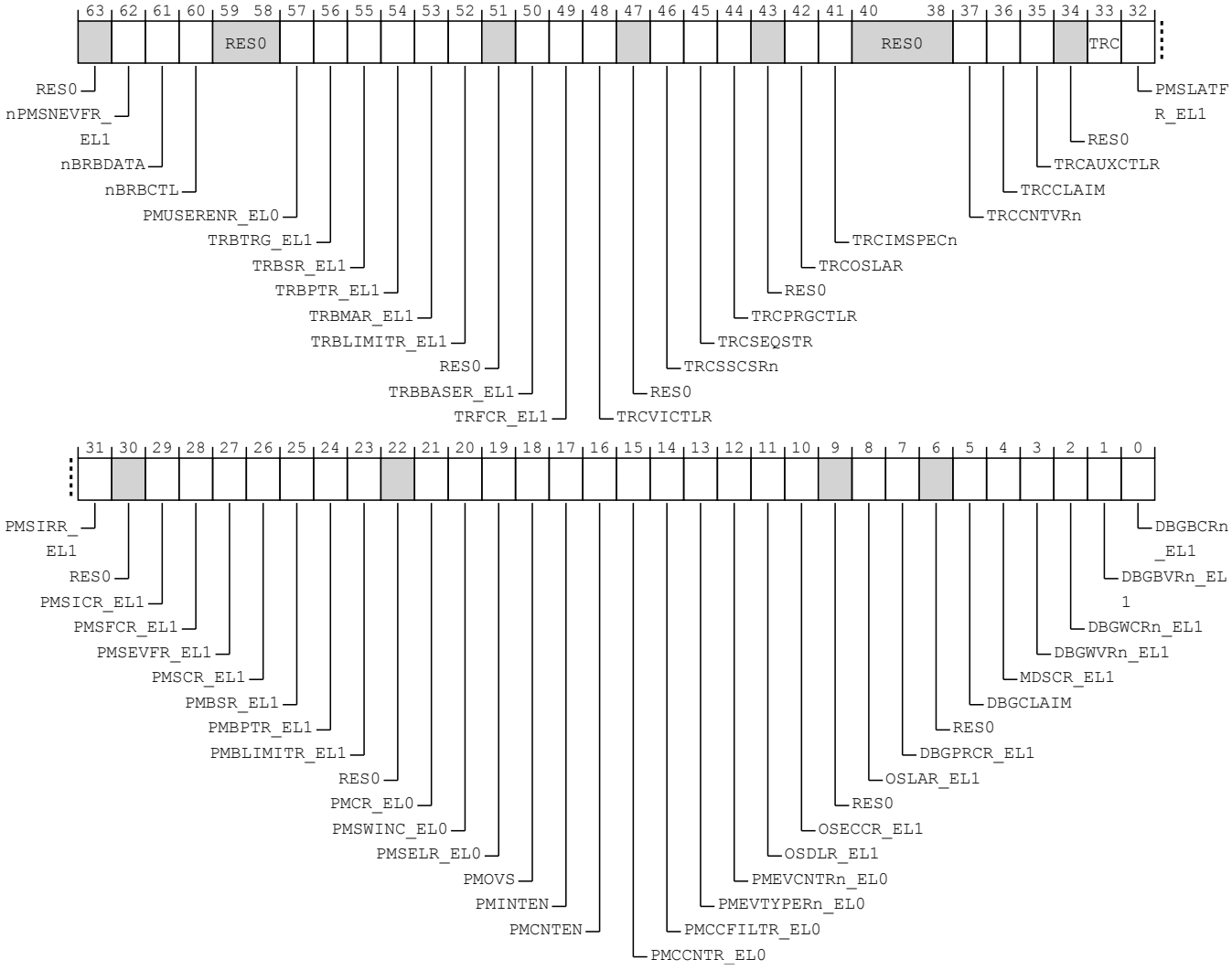

Bit [63]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nPMSNEVFR_EL1, bit [62]

When FEAT_SPE_FnE is implemented:

Trap MSR writes of PMSNEVFR_EL1 at EL1 using AArch64 to EL2.

nPMSNEVFR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSNEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of PMSNEVFR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBDATA, bit [61]

When FEAT_BRBE is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- BRBINFINJ_EL1.
- BRBSRCINJ_EL1.
- BRBTGTINJ_EL1.
- BRBTS_EL1.

nBRBDATA	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBCTL, bit [60]

When FEAT_BRBE is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BRBCR_EL1.
- BRBFCR_EL1.

nBRBCTL	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [59:58]

Reserved, RES0.

PMUSERENR_EL0, bit [57]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMUSERENR_EL0 at EL1 using AArch64 to EL2.

PMUSERENR_EL0	Meaning
0b0	MSR writes of PMUSERENR_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMUSERENR_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBTRG_EL1, bit [56]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBTRG_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRBTRG_EL1	Meaning
0b0	MSR writes of TRBTRG_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBTRG_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBSR_EL1, bit [55]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBSR_EL1 at EL1 using AArch64 to EL2.

TRBSR_EL1	Meaning
0b0	MSR writes of TRBSR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBPTR_EL1, bit [54]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBPTR_EL1 at EL1 using AArch64 to EL2.

TRBPTR_EL1	Meaning
0b0	MSR writes of TRBPTR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBMAR_EL1, bit [53]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBMAR_EL1 at EL1 using AArch64 to EL2.

TRBMAR_EL1	Meaning
`0b0`	MSR writes of TRBMAR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBMAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBLIMITR_EL1, bit [52]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBLIMITR_EL1 at EL1 using AArch64 to EL2.

TRBLIMITR_EL1	Meaning
`0b0`	MSR writes of TRBLIMITR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [51]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRBBASER_EL1, bit [50]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBBASER_EL1 at EL1 using AArch64 to EL2.

TRBBASER_EL1	Meaning
0b0	MSR writes of TRBBASER_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBBASER_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRFCR_EL1, bit [49]

When FEAT_TRF is implemented:

Trap MSR writes of TRFCR_EL1 at EL1 using AArch64 to EL2.

TRFCR_EL1	Meaning
0b0	MSR writes of TRFCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCVICTLR, bit [48]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCVICTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCVICTLR at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCVICTLR	Meaning
0b0	MSR writes of TRCVICTLR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCVICTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [47]

Reserved, RES0.

TRCSSCSRn, bit [46]

When IsFeatureImplemented(FEAT_ETE) || ((IsFeatureImplemented(FEAT_ETMv4) && TRCSSCSR<n> are implemented) && IsFeatureImplemented(FEAT_TRC_SR)):

In an Armv9 implementation, trap MSR writes of TRCSSCSR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCSSCSR<n> at EL1 using AArch64 to EL2.

TRCSSCSRn	Meaning
0b0	MSR writes of TRCSSCSR<n> are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCSSCSR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If Single-shot Comparator n is not implementented, a write of TRCSSCSR<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCSEQSTR, bit [45]

When IsFeatureImplemented(FEAT_ETE) || ((IsFeatureImplemented(FEAT_ETMv4) && TRCSEQSTR is implemented) && IsFeatureImplemented(FEAT_TRC_SR)):

In an Armv9 implementation, trap MSR writes of TRCSEQSTR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCSEQSTR at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCSEQSTR	Meaning
0b0	MSR writes of TRCSEQSTR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCPRGCTLR, bit [44]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCPRGCTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCPRGCTLR at EL1 using AArch64 to EL2.

TRCPRGCTLR	Meaning
0b0	MSR writes of TRCPRGCTLR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCPRGCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [43]

Reserved, RES0.

TRCOSLAR, bit [42]

When FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented:

In an Armv8 implementation, trap MSR writes of ETM TRCOSLAR at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCOSLAR	Meaning
0b0	MSR writes of ETM TRCOSLAR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ETM TRCOSLAR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCIMSPECn, bit [41]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCIMSPEC<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCIMSPEC<n> at EL1 using AArch64 to EL2.

TRCIMSPECn	Meaning
0b0	MSR writes of TRCIMSPEC<n> are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCIMSPEC<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

TRCIMSPEC<1-7> are optional. If TRCIMSPEC<n> is not implemented, a write of TRCIMSPEC<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [40:38]

Reserved, RES0.

TRCCNTVRn, bit [37]

When IsFeatureImplemented(FEAT_ETE) || ((IsFeatureImplemented(FEAT_ETMv4) && TRCCNTVR<n> are implemented) && IsFeatureImplemented(FEAT_TRC_SR)):

In an Armv9 implementation, trap MSR writes of TRCCNTVR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCCNTVR<n> at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCCNTVRn	Meaning
0b0	MSR writes of TRCCNTVR<n> are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCCNTVR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If Counter n is not implemented, a write of TRCCNTVR<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCCLAIM, bit [36]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- In an Armv9 implementation: TRCCLAIMCLR and TRCCLAIMSET.
- In an Armv8 implementation: ETM TRCCLAIMCLR and ETM TRCCLAIMSET.

TRCCLAIM	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCAUXCTLR, bit [35]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCAUXCTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCAUXCTLR at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCAUXCTLR	Meaning
0b0	MSR writes of TRCAUXCTLR are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCAUXCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [34]

Reserved, RES0.

TRC, bit [33]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- In an Armv9 implementation:
  - TRCACATR<n>.
  - TRCACVR<n>.
  - TRCBBCTLR.
  - TRCCCCTLR.
  - TRCCIDCCTLR0.
  - TRCCIDCCTLR1.
  - TRCCIDCVR<n>.
  - TRCCNTCTLR<n>.
  - TRCCNTRLDVR<n>.
  - TRCCONFIGR.
  - TRCEVENTCTL0R.
  - TRCEVENTCTL1R.
  - TRCEXTINSELR<n>.
  - TRCQCTLR.
  - TRCRSCTLR<n>.
  - TRCRSR.
  - TRCSEQEVR<n>.
  - TRCSEQRSTEVR.
  - TRCSSCCR<n>.
  - TRCSSPCICR<n>.
  - TRCSTALLCTLR.
  - TRCSYNCPR.
  - TRCTRACEIDR.
  - TRCTSCTLR.
  - TRCVIIECTLR.
  - TRCVIPCSSCTLR.
  - TRCVISSCTLR.
  - TRCVMIDCCTLR0.
  - TRCVMIDCCTLR1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCVMIDCVR<n>.

- In an Armv8 implementation:

  - ETM TRCACATR<n>.
  - ETM TRCACVR<n>.
  - ETM TRCBBCTLR.
  - ETM TRCCCCTLR.
  - ETM TRCCIDCCTLR0.
  - ETM TRCCIDCCTLR1.
  - ETM TRCCIDCVR<n>.
  - ETM TRCCNTCTLR<n>.
  - ETM TRCCNTRLDVR<n>.
  - ETM TRCCONFIGR.
  - ETM TRCEVENTCTL0R.
  - ETM TRCEVENTCTL1R.
  - ETM TRCEXTINSELR.
  - ETM TRCQCTLR.
  - ETM TRCRSCTLR<n>.
  - ETM TRCSEQEVR<n>.
  - ETM TRCSEQRSTEVR.
  - ETM TRCSSCCR<n>.
  - ETM TRCSSPCICR<n>.
  - ETM TRCSTALLCTLR.
  - ETM TRCSYNCPR.
  - ETM TRCTRACEIDR.
  - ETM TRCTSCTLR.
  - ETM TRCVIIECTLR.
  - ETM TRCVIPCSSCTLR.
  - ETM TRCVISSCTLR.
  - ETM TRCVMIDCCTLR0.
  - ETM TRCVMIDCCTLR1.
  - ETM TRCVMIDCVR<n>.

TRC	Meaning
`0b0`	`MSR` writes of the specified System registers are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then `MSR` writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

A write of an unimplemented register is UNDEFINED.

TRCEXTINSELR<n> and TRCRSR are implemented only if FEAT_ETE is implemented.

TRCEXTINSELR is implemented only if FEAT_ETE is not implemented and FEAT_ETMv4 is implemented.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSLATFR_EL1, bit [32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE is implemented:

Trap MSR writes of PMSLATFR_EL1 at EL1 using AArch64 to EL2.

PMSLATFR_EL1	Meaning
0b0	MSR writes of PMSLATFR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSLATFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSIRR_EL1, bit [31]

When FEAT_SPE is implemented:

Trap MSR writes of PMSIRR_EL1 at EL1 using AArch64 to EL2.

PMSIRR_EL1	Meaning
0b0	MSR writes of PMSIRR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSIRR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [30]

Reserved, RES0.

PMSICR_EL1, bit [29]

When FEAT_SPE is implemented:

Trap MSR writes of PMSICR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSICR_EL1	Meaning
0b0	MSR writes of PMSICR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSICR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSFCR_EL1, bit [28]

When FEAT_SPE is implemented:

Trap MSR writes of PMSFCR_EL1 at EL1 using AArch64 to EL2.

PMSFCR_EL1	Meaning
0b0	MSR writes of PMSFCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSEVFR_EL1, bit [27]

When FEAT_SPE is implemented:

Trap MSR writes of PMSEVFR_EL1 at EL1 using AArch64 to EL2.

PMSEVFR_EL1	Meaning
0b0	MSR writes of PMSEVFR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSCR_EL1, bit [26]

When FEAT_SPE is implemented:

Trap MSR writes of PMSCR_EL1 at EL1 using AArch64 to EL2.

PMSCR_EL1	Meaning
`0b0`	MSR writes of PMSCR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMBSR_EL1, bit [25]

When FEAT_SPE is implemented:

Trap MSR writes of PMBSR_EL1 at EL1 using AArch64 to EL2.

PMBSR_EL1	Meaning
`0b0`	MSR writes of PMBSR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMBPTR_EL1, bit [24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE is implemented:

Trap MSR writes of PMBPTR_EL1 at EL1 using AArch64 to EL2.

PMBPTR_EL1	Meaning
0b0	MSR writes of PMBPTR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMBLIMITR_EL1, bit [23]

When FEAT_SPE is implemented:

Trap MSR writes of PMBLIMITR_EL1 at EL1 using AArch64 to EL2.

PMBLIMITR_EL1	Meaning
0b0	MSR writes of PMBLIMITR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [22]

Reserved, RES0.

PMCR_EL0, bit [21]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMCR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMCR_EL0	Meaning
0b0	MSR writes of PMCR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCR at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of PMCR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes of PMCR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSWINC_EL0, bit [20]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSWINC at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMSWINC_EL0	Meaning
0b0	MSR writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSWINC at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes of PMSWINC at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSELR_EL0, bit [19]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSELR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMSELR_EL0	Meaning
0b0	MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSELR at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes of PMSELR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMOVS, bit [18]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMOVSCLR_EL0 and PMOVSSET_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMOVSR and PMOVSSET.

PMOVS	Meaning
0b0	MSR writes at EL1 and EL0 using AArch64 and MCR writes at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

PMINTEN, bit [17]

When FEAT_PMUv3 is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- PMINTENCLR_EL1.
- PMINTENSET_EL1.

PMINTEN	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCNTEN, bit [16]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMCNTENCLR_EL0 and PMCNTENSET_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMCNTENCLR and PMCNTENSET.

PMCNTEN	Meaning
0b0	MSR writes at EL1 and EL0 using AArch64 and MCR writes at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCCNTR_EL0, bit [15]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MCR and MCRR writes of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMCCNTR_EL0	Meaning
`0b0`	MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MCR and MCRR writes of PMCCNTR at EL0 using AArch32 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`. • MCR writes of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value `0x03`. • MCRR writes of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value `0x04`.

PMCCNTR_EL0 can also be indirectly set to zero by a write of 1 to PMCR_EL0.C or PMZR_EL0.C in AArch64 state, or a write of 1 to PMCR.C in AArch32 state. Setting this field to 1 has no effect on indirect writes to PMCCNTR_EL0 using PMCR_EL0, PMZR_EL0, or PMCR.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMCCFILTR_EL0, bit [14]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

PMCCFILTR_EL0	Meaning
`0b0`	MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCCFILTR at EL0 using AArch32 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMCCFILTR_EL0	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

PMCCFILTR_EL0 can also be accessed in AArch64 state using PMXEVTYPER_EL0 when PMSELR_EL0.SEL == 31, and PMCCFILTR can also be accessed in AArch32 state using PMXEVTYPER when PMSELR.SEL == 31.

Setting this field to 1 has no effect on accesses to PMXEVTYPER_EL0 and PMXEVTYPER, regardless of the value of PMSELR_EL0.SEL or PMSELR.SEL.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMEVTYPERn_EL0, bit [13]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMEVTYPER<n> and PMXEVTYPER.

PMEVTYPERn_EL0	Meaning
0b0	MSR writes at EL1 and EL0 using AArch64 and MCR writes at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value 0x18. • MCR writes at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value 0x03.

Regardless of the value of this field, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a write of PMEVTYPER<n>_EL0, or, if n is not 31, a write of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In AArch32 state, a write of PMEVTYPER<n>, or, if n is not 31, a write of PMXEVTYPER when PMSELR.SEL == n.
- If event counter n is implemented, n is greater-than-or-equal-to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a write of PMEVTYPER<n>_EL0, or a write of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a write of PMEVTYPER<n>, or a write of PMXEVTYPER when PMSELR.SEL == n, reported with EC syndrome value `0x03`.

See also HDFGWTR_EL2.PMCCFILTR_EL0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMEVCNTRn_EL0, bit [12]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of any of the following System registers to EL2:

- At EL1 and EL0 using AArch64: PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: PMEVCNTR<n> and PMXEVCNTR.

PMEVCNTRn_EL0	Meaning
`0b0`	MSR writes at EL1 and EL0 using AArch64 and MCR writes at EL0 using AArch32 of the specified System registers are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes at EL1 and EL0 using AArch64 of any of the specified AArch64 System registers are trapped to EL2 and reported with EC syndrome value `0x18`. • MCR writes at EL0 using AArch32 when EL1 is using AArch64 of any of the specified AArch32 System registers are trapped to EL2 and reported with EC syndrome value `0x03`.

Regardless of the value of this field, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a write of PMEVCNTR<n>_EL0, or a write of PMXEVCNTR_EL0 when PMSELR_EL0.SEL is n.
  - In AArch32 state, a write of PMEVCNTR<n>, or a write of PMXEVCNTR when PMSELR.SEL is n.
- If event counter n is implemented, n is greater than or equal to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  - In AArch64 state, a write of PMEVCNTR<n>_EL0, or a write of PMXEVCNTR_EL0 when PMSELR_EL0.SEL is n, reported with EC syndrome value `0x18`.
  - In AArch32 state, a write of PMEVCNTR<n>, or a write of PMXEVCNTR when PMSELR.SEL is n, reported with EC syndrome value `0x03`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For values of n less than MDCR_EL2.HPMN, PMEVCNTR<n>_EL0 can also be indirectly set to zero by a write of 1 to PMCR_EL0.P or PMZR_EL0.P<n> in AArch64 state, or a write of 1 to PMCR.P in AArch32 state. Setting this field to 1 has no effect on indirect writes to PMEVCNTR<n>_EL0 using PMCR_EL0, PMZR_EL0, or PMCR.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

OSDLR_EL1, bit [11]

When FEAT_DoubleLock is implemented:

Trap MSR writes of OSDLR_EL1 at EL1 using AArch64 to EL2.

OSDLR_EL1	Meaning
`0b0`	MSR writes of OSDLR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of OSDLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

OSECCR_EL1, bit [10]

Trap MSR writes of OSECCR_EL1 at EL1 using AArch64 to EL2.

OSECCR_EL1	Meaning
`0b0`	MSR writes of OSECCR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of OSECCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [9]

Reserved, RES0.

OSLAR_EL1, bit [8]

Trap MSR writes of OSLAR_EL1 at EL1 using AArch64 to EL2.

OSLAR_EL1	Meaning
0b0	MSR writes of OSLAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of OSLAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGPRCR_EL1, bit [7]

Trap MSR writes of DBGPRCR_EL1 at EL1 using AArch64 to EL2.

DBGPRCR_EL1	Meaning
0b0	MSR writes of DBGPRCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGPRCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

DBGCLAIM, bit [5]

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- DBGCLAIMCLR_EL1.
- DBGCLAIMSET_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DBGCLAIM	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

MDSCR_EL1, bit [4]

Trap MSR writes of MDSCR_EL1 at EL1 using AArch64 to EL2.

MDSCR_EL1	Meaning
0b0	MSR writes of MDSCR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of MDSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGWVRn_EL1, bit [3]

Trap MSR writes of DBGWVR<n>_EL1 at EL1 using AArch64 to EL2.

DBGWVRn_EL1	Meaning
0b0	MSR writes of DBGWVR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGWVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If watchpoint n is not implemented, a write of DBGWVR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGWCRn_EL1, bit [2]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trap MSR writes of DBGWCR<n>_EL1 at EL1 using AArch64 to EL2.

DBGWCRn_EL1	Meaning
0b0	MSR writes of DBGWCR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGWCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If watchpoint n is not implemented, a write of DBGWCR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGBVRn_EL1, bit [1]

Trap MSR writes of DBGBVR<n>_EL1 at EL1 using AArch64 to EL2.

DBGBVRn_EL1	Meaning
0b0	MSR writes of DBGBVR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGBVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If breakpoint n is not implemented, a write of DBGBVR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DBGBCRn_EL1, bit [0]

Trap MSR writes of DBGBCR<n>_EL1 at EL1 using AArch64 to EL2.

DBGBCRn_EL1	Meaning
0b0	MSR writes of DBGBCR<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGBCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If breakpoint n is not implemented, a write of DBGBCR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HDFGWTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HDFGWTR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1D8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HDFGWTR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HDFGWTR_EL2();
end;
```

```
MSR HDFGWTR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1D8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HDFGWTR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HDFGWTR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.69 HFGITR2_EL2, Hypervisor Fine-Grained Instruction Trap Register 2

The HFGITR2_EL2 characteristics are:

Purpose

Provides instruction trap controls.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HFGITR2_EL2 are UNDEFINED.

Attributes

HFGITR2_EL2 is a 64-bit register.

Field descriptions

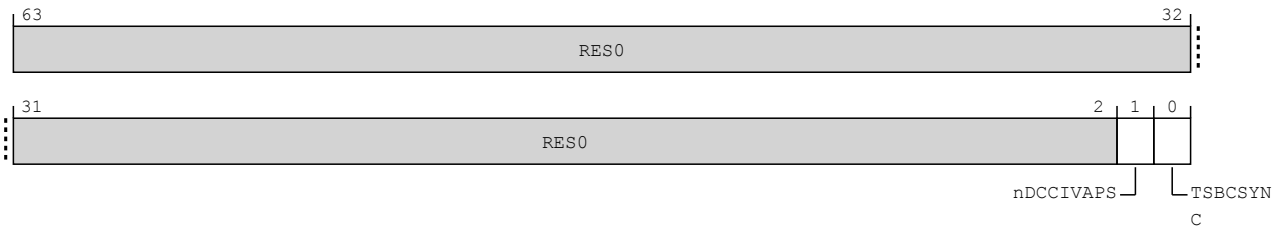

Bits [63:2]

Reserved, RES0.

nDCCIVAPS, bit [1]

When FEAT_PoPS is implemented:

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- DC CIVAPS.
- DC CIGDVAPS, if FEAT_MTE2 is implemented.

If the Point of Physical Storage is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

nDCCIVAPS	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
0b1	Execution of the specified instructions is not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TSBCSYNC, bit [0]

When FEAT_TRBEv1p1 is implemented:

Trap execution of TSB CSYNC at EL1 and EL0 using AArch64 to EL2.

TSBCSYNC	Meaning
`0b0`	Execution of TSB CSYNC is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then execution of TSB CSYNC at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x0A`, unless the instruction generates a higher priority exception.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HFGITR2_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HFGITR2_EL2
```

op0	op1	CRn	CRm	op2
`0b11`	`0b100`	`0b0011`	`0b0001`	`0b111`

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x310);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HFGITR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HFGITR2_EL2();
end;
```

MSR HFGITR2_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b111

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x310) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HFGITR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HFGITR2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.70 HFGITR_EL2, Hypervisor Fine-Grained Instruction Trap Register

The HFGITR_EL2 characteristics are:

Purpose

Provides instruction trap controls.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HFGITR_EL2 are UNDEFINED.

Attributes

HFGITR_EL2 is a 64-bit register.

Field descriptions

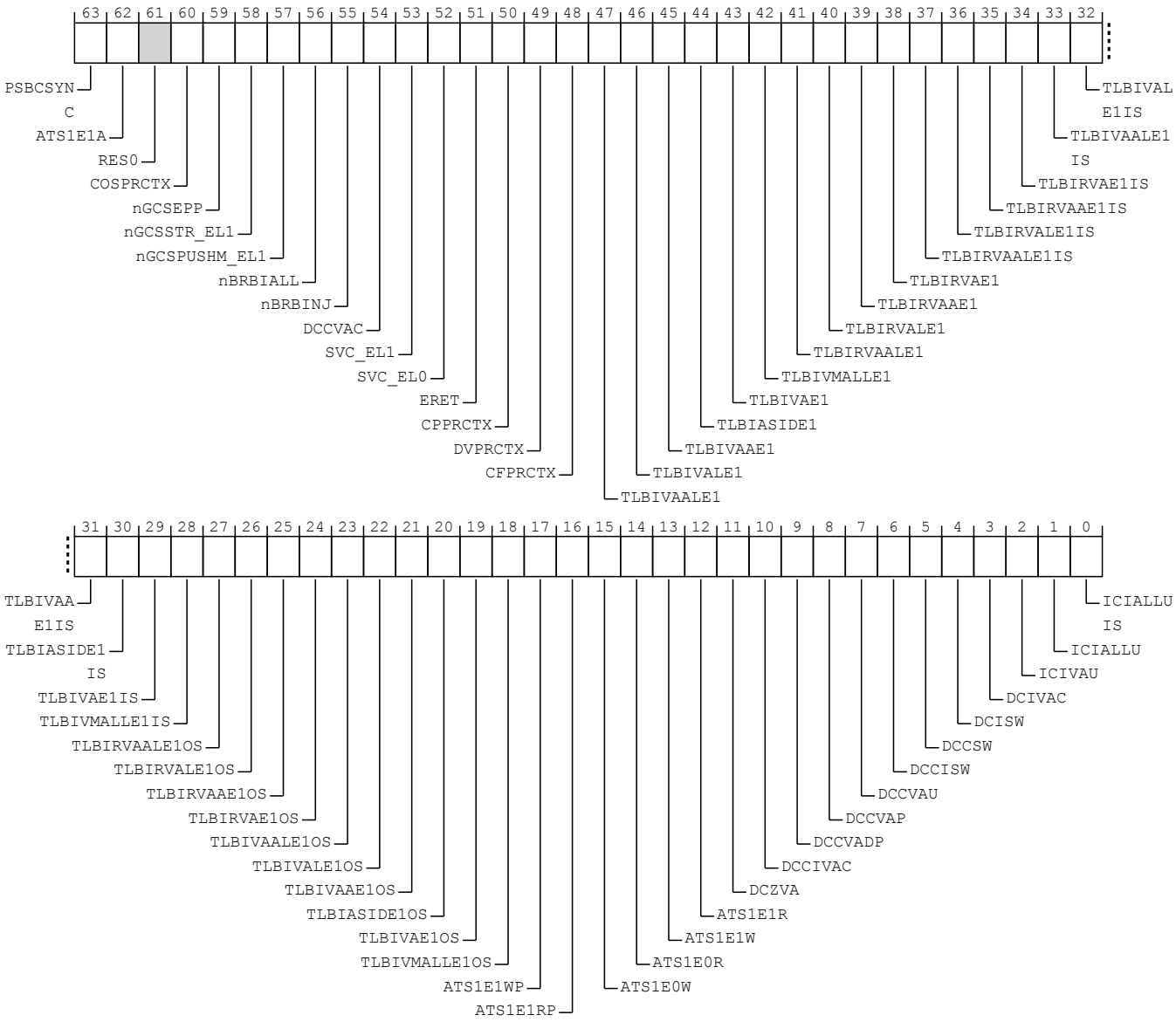

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSBCSYNC, bit [63]

When FEAT_SPEv1p5 is implemented:

Trap execution of PSB CSYNC at EL1 and EL0 using AArch64 to EL2.

PSBCSYNC	Meaning
`0b0`	Execution of PSB CSYNC is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of PSB CSYNC at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x0A`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATS1E1A, bit [62]

When FEAT_ATS1A is implemented:

Trap execution of AT S1E1A at EL1 using AArch64 to EL2.

ATS1E1A	Meaning
`0b0`	Execution of AT S1E1A is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E1A at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [61]

Reserved, RES0.

COSPRCTX, bit [60]

When FEAT_SPECRES2 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trap execution of COSP RCTX at EL1 and EL0 using AArch64 and execution of COSPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

COSPRCTX	Meaning
0b0	Execution of COSP RCTX at EL1 and EL0 using AArch64 and execution of COSPRCTX at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the instruction generates a higher priority exception: • Execution of COSP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18. • Execution of COSPRCTX at EL0 using AArch32 when EL1 is using AArch64 is trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCSEPP, bit [59]

When FEAT_GCS is implemented:

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- GCSPUSHX.
- GCSPOPCX.

nGCSEPP	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
0b1	Execution of the specified instructions is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCSSTR_EL1, bit [58]

When FEAT_GCS is implemented:

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- GCSSTR.
- GCSSTTR when PSTATE.UAO is 1.
- GCSSTTR when the Effective value of HCR_EL2.{NV, NV1} is {1, 1}.

nGCSSTR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the specified instructions generates a GCS exception with EC syndrome value 0x2D, unless the instruction generates a higher priority exception.
0b1	Execution of the specified instructions is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCSPUSHM_EL1, bit [57]

When FEAT_GCS is implemented:

Trap execution of GCSPUSHM at EL1 using AArch64 to EL2.

nGCSPUSHM_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of GCSPUSHM at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
0b1	Execution of GCSPUSHM is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBIALL, bit [56]

When FEAT_BRBE is implemented:

Trap execution of BRB IALL at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nBRBIALL	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of BRB IALL at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
0b1	Execution of BRB IALL is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nBRBINJ, bit [55]

When FEAT_BRBE is implemented:

Trap execution of BRB INJ at EL1 using AArch64 to EL2.

nBRBINJ	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of BRB INJ at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
0b1	Execution of BRB INJ is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DCCVAC, bit [54]

Trap execution of any of the following AArch64 instructions at EL1 and EL0 to EL2:

- DC CVAC.
- DC CGVAC, if FEAT_MTE is implemented.
- DC CGDVAC, if FEAT_MTE is implemented.
- DC CVAOC, if FEAT_OCCMO is implemented.
- DC CGDVAOC, if FEAT_OCCMO is implemented.

DCCVAC	Meaning
0b0	Execution of the specified instructions is not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DCCVAC	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution at EL1 and EL0 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SVC_EL1, bit [53]

Trap execution of SVC at EL1 using AArch64 to EL2.

SVC_EL1	Meaning
0b0	Execution of SVC is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of SVC at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x15, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SVC_EL0, bit [52]

Trap execution of SVC at EL0 using AArch64 and execution of SVC at EL0 using AArch32 when EL1 is using AArch64 to EL2.

SVC_EL0	Meaning
0b0	Execution of SVC at EL0 using AArch64 and execution of SVC at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the instruction generates a higher priority exception: • Execution of SVC at EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x15. • Execution of SVC at EL0 using AArch32 when EL1 is using AArch64 is trapped to EL2 and reported with EC syndrome value 0x11.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

ERET, bit [51]

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- `ERET`.
- `ERETAA`, if FEAT_PAuth is implemented.
- `ERETAB`, if FEAT_PAuth is implemented.

ERET	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x1A`, unless the instruction generates a higher priority exception.

If EL2 is implemented and enabled in the current Security state, HCR_EL2.API == 0, and this field enables a fine-grained trap on the instruction, then execution at EL1 using AArch64 of `ERETAA` or `ERETAB` instructions is trapped to EL2 and reported with EC syndrome value `0x1A` with its associated ISS field, as the fine-grained trap has higher priority than the trap enabled by HCR_EL2.API == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CPPRCTX, bit [50]

When FEAT_SPECRES is implemented:

Trap execution of CPP RCTX at EL1 and EL0 using AArch64 and execution of CPPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

CPPRCTX	Meaning
`0b0`	Execution of CPP RCTX at EL1 and EL0 using AArch64 and execution of CPPRCTX at EL0 using AArch32 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the instruction generates a higher priority exception: • Execution of CPP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`. • Execution of CPPRCTX at EL0 using AArch32 when EL1 is using AArch64 is trapped to EL2 and reported with EC syndrome value `0x03`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DVPRCTX, bit [49]

When FEAT_SPECRES is implemented:

Trap execution of DVP RCTX at EL1 and EL0 using AArch64 and execution of DVPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

DVPRCTX	Meaning
`0b0`	Execution of DVP RCTX at EL1 and EL0 using AArch64 and execution of DVPRCTX at EL0 using AArch32 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the instruction generates a higher priority exception: • Execution of DVP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`. • Execution of DVPRCTX at EL0 using AArch32 when EL1 is using AArch64 is trapped to EL2 and reported with EC syndrome value `0x03`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CFPRCTX, bit [48]

When FEAT_SPECRES is implemented:

Trap execution of CFP RCTX at EL1 and EL0 using AArch64 and execution of CFPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

CFPRCTX	Meaning
`0b0`	Execution of CFP RCTX at EL1 and EL0 using AArch64 and execution of CFPRCTX at EL0 using AArch32 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the instruction generates a higher priority exception: • Execution of CFP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`. • Execution of CFPRCTX at EL0 using AArch32 when EL1 is using AArch64 is trapped to EL2 and reported with EC syndrome value `0x03`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

TLBIVAALE1, bit [47]

Trap execution of TLBI VAALE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAALE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAALE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAALE1NXS.

TLBIVAALE1	Meaning
`0b0`	Execution of TLBI VAALE1 is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVALE1, bit [46]

Trap execution of TLBI VALE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VALE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VALE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VALE1NXS.

TLBIVALE1	Meaning
`0b0`	Execution of TLBI VALE1 is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVAAE1, bit [45]

Trap execution of TLBI VAAE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAAE1NXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAAE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAAE1NXS.

TLBIVAAE1	Meaning
0b0	Execution of TLBI VAAE1 is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIASIDE1, bit [44]

Trap execution of TLBI ASIDE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI ASIDE1NXS.

TLBIASIDE1	Meaning
0b0	Execution of TLBI ASIDE1 is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI ASIDE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVAE1, bit [43]

Trap execution of TLBI VAE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAE1NXS.

TLBIVAE1	Meaning
0b0	Execution of TLBI VAE1 is not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIVAE1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVMALLE1, bit [42]

Trap execution of TLBI VMALLE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VMALLE1NXS.

TLBIVMALLE1	Meaning
0b0	Execution of TLBI VMALLE1 is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VMALLE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIRVAALE1, bit [41]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVAALE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAALE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAALE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAALE1NXS.

TLBIRVAALE1	Meaning
0b0	Execution of TLBI RVAALE1 is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVALE1, bit [40]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVALE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVALE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVALE1NXS.

TLBIRVALE1	Meaning
`0b0`	Execution of TLBI RVALE1 is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVALE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAAE1, bit [39]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVAAE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAAE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAAE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAAE1NXS.

TLBIRVAAE1	Meaning
`0b0`	Execution of TLBI RVAAE1 is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAE1, bit [38]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVAE1 at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1NXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAE1.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAE1NXS.

TLBIRVAE1	Meaning
`0b0`	Execution of TLBI RVAE1 is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1 at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAALE1IS, bit [37]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVAALE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAALE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAALE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAALE1ISNXS.

TLBIRVAALE1IS	Meaning
`0b0`	Execution of TLBI RVAALE1IS is not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIRVAALE1IS	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVALE1IS, bit [36]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVALE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVALE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVALE1ISNXS.

TLBIRVALE1IS	Meaning
0b0	Execution of TLBI RVALE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAAE1IS, bit [35]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVAAE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAAE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAAE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAAE1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIRVAAE1IS	Meaning
0b0	Execution of TLBI RVAAE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAE1IS, bit [34]

When FEAT_TLBIRANGE is implemented:

Trap execution of TLBI RVAE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAE1ISNXS.

TLBIRVAE1IS	Meaning
0b0	Execution of TLBI RVAE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIVAALE1IS, bit [33]

Trap execution of TLBI VAALE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAALE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAALE1IS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAALE1ISNXS.

TLBIVAALE1IS	Meaning
0b0	Execution of TLBI VAALE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVALE1IS, bit [32]

Trap execution of TLBI VALE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VALE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VALE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VALE1ISNXS.

TLBIVALE1IS	Meaning
0b0	Execution of TLBI VALE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VALE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVAAE1IS, bit [31]

Trap execution of TLBI VAAE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAAE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAAE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAAE1ISNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIVAAE1IS	Meaning
0b0	Execution of TLBI VAAE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIASIDE1IS, bit [30]

Trap execution of TLBI ASIDE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI ASIDE1ISNXS.

TLBIASIDE1IS	Meaning
0b0	Execution of TLBI ASIDE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI ASIDE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVAE1IS, bit [29]

Trap execution of TLBI VAE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAE1ISNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAE1IS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAE1ISNXS.

TLBIVAE1IS	Meaning
0b0	Execution of TLBI VAE1IS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIVMALLE1IS, bit [28]

Trap execution of TLBI VMALLE1IS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VMALLE1ISNXS.

TLBIVMALLE1IS	Meaning
`0b0`	Execution of TLBI VMALLE1IS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VMALLE1IS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TLBIRVAALE1OS, bit [27]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVAALE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAALE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAALE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAALE1OSNXS.

TLBIRVAALE1OS	Meaning
`0b0`	Execution of TLBI RVAALE1OS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIRVALE1OS, bit [26]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVALE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVALE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVALE1OSNXS.

TLBIRVALE1OS	Meaning
`0b0`	Execution of TLBI RVALE1OS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAAE1OS, bit [25]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVAAE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAAE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAAE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAAE1OSNXS.

TLBIRVAAE1OS	Meaning
`0b0`	Execution of TLBI RVAAE1OS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIRVAE1OS, bit [24]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVAE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP RVAE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP RVAE1OSNXS.

TLBIRVAE1OS	Meaning
`0b0`	Execution of TLBI RVAE1OS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIVAALE1OS, bit [23]

When FEAT_TLBIOS is implemented:

Trap execution of TLBI VAALE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAALE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAALE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAALE1OSNXS.

TLBIVAALE1OS	Meaning
`0b0`	Execution of TLBI VAALE1OS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIVALE1OS, bit [22]

When FEAT_TLBIOS is implemented:

Trap execution of TLBI VALE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VALE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VALE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VALE1OSNXS.

TLBIVALE1OS	Meaning
`0b0`	Execution of TLBI VALE1OS is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VALE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIVAAE1OS, bit [21]

When FEAT_TLBIOS is implemented:

Trap execution of TLBI VAAE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAAE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAAE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAAE1OSNXS.

TLBIVAAE1OS	Meaning
`0b0`	Execution of TLBI VAAE1OS is not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIVAAE1OS	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIASIDE1OS, bit [20]

When FEAT_TLBIOS is implemented:

Trap execution of TLBI ASIDE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI ASIDE1OSNXS.

TLBIASIDE1OS	Meaning
0b0	Execution of TLBI ASIDE1OS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI ASIDE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIVAE1OS, bit [19]

When FEAT_TLBIOS is implemented:

Trap execution of TLBI VAE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAE1OSNXS.

If FEAT_D128 is implemented, then this field also traps execution of TLBIP VAE1OS.

If FEAT_XS and FEAT_D128 are implemented, then this field also traps execution of TLBIP VAE1OSNXS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBIVAE1OS	Meaning
0b0	Execution of TLBI VAE1OS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLBIVMALLE1OS, bit [18]

When FEAT_TLBIOS is implemented:

Trap execution of TLBI VMALLE1OS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VMALLE1OSNXS.

TLBIVMALLE1OS	Meaning
0b0	Execution of TLBI VMALLE1OS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VMALLE1OS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATS1E1WP, bit [17]

When FEAT_PAN2 is implemented:

Trap execution of AT S1E1WP at EL1 using AArch64 to EL2.

ATS1E1WP	Meaning
0b0	Execution of AT S1E1WP is not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ATS1E1WP	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E1WP at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATS1E1RP, bit [16]

When FEAT_PAN2 is implemented:

Trap execution of AT S1E1RP at EL1 using AArch64 to EL2.

ATS1E1RP	Meaning
0b0	Execution of AT S1E1RP is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E1RP at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATS1E0W, bit [15]

Trap execution of AT S1E0W at EL1 using AArch64 to EL2.

ATS1E0W	Meaning
0b0	Execution of AT S1E0W is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E0W at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

ATS1E0R, bit [14]

Trap execution of AT S1E0R at EL1 using AArch64 to EL2.

ATS1E0R	Meaning
`0b0`	Execution of AT S1E0R is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E0R at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ATS1E1W, bit [13]

Trap execution of AT S1E1W at EL1 using AArch64 to EL2.

ATS1E1W	Meaning
`0b0`	Execution of AT S1E1W is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E1W at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ATS1E1R, bit [12]

Trap execution of AT S1E1R at EL1 using AArch64 to EL2.

ATS1E1R	Meaning
`0b0`	Execution of AT S1E1R is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of AT S1E1R at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCZVA, bit [11]

Trap execution of any of the following AArch64 instructions at EL1 and EL0 to EL2:

- DC ZVA.
- DC GVA, if FEAT_MTE is implemented.
- DC GZVA, if FEAT_MTE is implemented.

Note

Unlike HCR_EL2.TDZ, this field has no effect on DCZID_EL0.DZP.

DCZVA	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 and EL0 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCCIVAC, bit [10]

Trap execution of any of the following AArch64 instructions at EL1 and EL0 to EL2:

- DC CIVAC.
- DC CIGVAC, if FEAT_MTE is implemented.
- DC CIGDVAC, if FEAT_MTE is implemented.
- DC CIVAOC, if FEAT_OCCMO is implemented.
- DC CIGDVAOC, if FEAT_OCCMO is implemented.

DCCIVAC	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution at EL1 and EL0 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DCCVADP, bit [9]

When FEAT_DPB2 is implemented:

Trap execution of any of the following AArch64 instructions at EL1 and EL0 to EL2:

- DC CVADP.
- DC CGVADP, if FEAT_MTE is implemented.
- DC CGDVADP, if FEAT_MTE is implemented.

DCCVADP	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution at EL1 and EL0 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DCCVAP, bit [8]

Trap execution of any of the following AArch64 instructions at EL1 and EL0 to EL2:

- DC CVAP.
- DC CGVAP, if FEAT_MTE is implemented.
- DC CGDVAP, if FEAT_MTE is implemented.

DCCVAP	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution at EL1 and EL0 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCCVAU, bit [7]

Trap execution of DC CVAU at EL1 and EL0 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DCCVAU	Meaning
0b0	Execution of DC CVAU is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution of DC CVAU at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCCISW, bit [6]

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- DC CISW.
- DC CIGSW, if FEAT_MTE2 is implemented.
- DC CIGDSW, if FEAT_MTE2 is implemented.

DCCISW	Meaning
0b0	Execution of the specified instructions is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCCSW, bit [5]

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- DC CSW.
- DC CGSW, if FEAT_MTE2 is implemented.
- DC CGDSW, if FEAT_MTE2 is implemented.

DCCSW	Meaning
0b0	Execution of the specified instructions is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCISW, bit [4]

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- DC ISW.
- DC IGSW, if FEAT_MTE2 is implemented.
- DC IGDSW, if FEAT_MTE2 is implemented.

DCISW	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCIVAC, bit [3]

Trap execution of any of the following AArch64 instructions at EL1 to EL2:

- DC IVAC.
- DC IGVAC, if FEAT_MTE2 is implemented.
- DC IGDVAC, if FEAT_MTE2 is implemented.

DCIVAC	Meaning
`0b0`	Execution of the specified instructions is not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution at EL1 using AArch64 of any of the specified instructions is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ICIVAU, bit [2]

Trap execution of IC IVAU at EL1 and EL0 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ICIVAU	Meaning
0b0	Execution of IC IVAU is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution of IC IVAU at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ICIALLU, bit [1]

Trap execution of IC IALLU at EL1 using AArch64 to EL2.

ICIALLU	Meaning
0b0	Execution of IC IALLU is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution of IC IALLU at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ICIALLUIS, bit [0]

Trap execution of IC IALLUIS at EL1 using AArch64 to EL2.

ICIALLUIS	Meaning
0b0	Execution of IC IALLUIS is not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then if the execution can be trapped, execution of IC IALLUIS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HFGITR_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, HFGITR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1C8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HFGITR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HFGITR_EL2();
end;
```

```
MSR HFGITR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b110

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1C8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HFGITR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HFGITR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.71 HFGRTR2_EL2, Hypervisor Fine-Grained Read Trap Register 2

The HFGRTR2_EL2 characteristics are:

Purpose

Provides controls for traps of `MRRS`, `MRS` and `MRC` reads of System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HFGRTR2_EL2 are UNDEFINED.

Attributes

HFGRTR2_EL2 is a 64-bit register.

Field descriptions

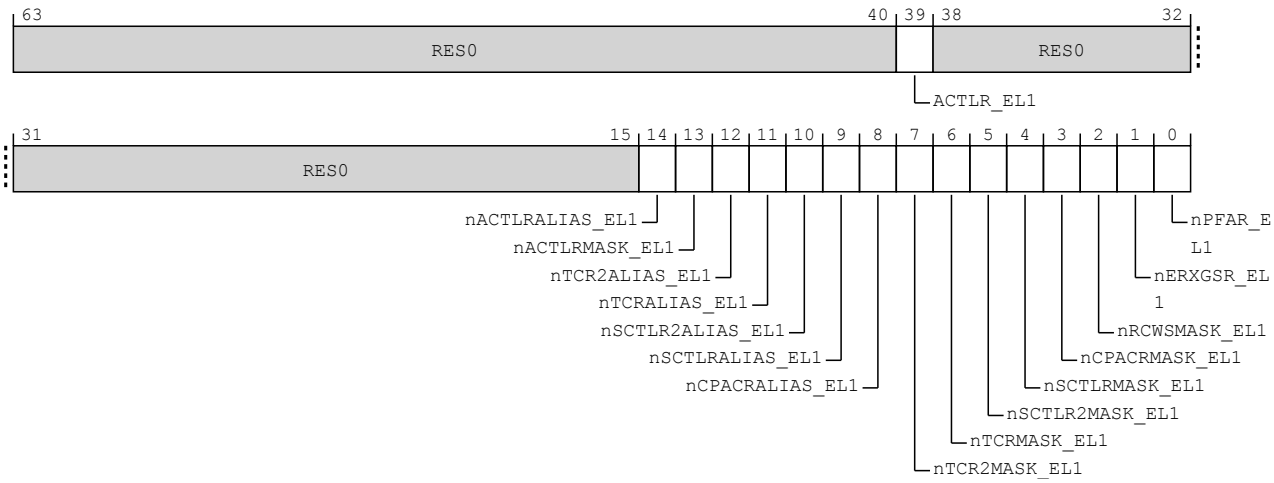

Bits [63:40]

Reserved, RES0.

ACTLR_EL1, bit [39]

When FEAT_SRMASK is implemented:

Trap `MRS` reads of ACTLR_EL1 at EL1 using AArch64 to EL2.

ACTLR_EL1	Meaning
`0b0`	`MRS` reads of ACTLR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, then `MRS` reads of ACTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

This field is ignored by the PE and treated as zero when all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [38:15]

Reserved, RES0.

nACTLRALIAS_EL1, bit [14]

When FEAT_SRMASK is implemented:

Trap MRS reads of ACTLRALIAS_EL1 at EL1 using AArch64 to EL2.

nACTLRALIAS_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MRS reads of ACTLRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.
`0b1`	MRS reads of ACTLRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nACTLRMASK_EL1, bit [13]

When FEAT_SRMASK is implemented:

Trap MRS reads of ACTLRMASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nACTLRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of ACTLRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of ACTLRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCR2ALIAS_EL1, bit [12]

When FEAT_SRMASK is implemented:

Trap MRS reads of TCR2ALIAS_EL1 at EL1 using AArch64 to EL2.

nTCR2ALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of TCR2ALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TCR2ALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCRALIAS_EL1, bit [11]

When FEAT_SRMASK is implemented:

Trap MRS reads of TCRALIAS_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTCRALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of TCRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TCRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLR2ALIAS_EL1, bit [10]

When FEAT_SRMASK is implemented:

Trap MRS reads of SCTLR2ALIAS_EL1 at EL1 using AArch64 to EL2.

nSCTLR2ALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SCTLR2ALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SCTLR2ALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLRALIAS_EL1, bit [9]

When FEAT_SRMASK is implemented:

Trap MRS reads of SCTLRALIAS_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSCTLRALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SCTLRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SCTLRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nCPACRALIAS_EL1, bit [8]

When FEAT_SRMASK is implemented:

Trap MRS reads of CPACRALIAS_EL1 at EL1 using AArch64 to EL2.

nCPACRALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of CPACRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of CPACRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCR2MASK_EL1, bit [7]

When FEAT_SRMASK is implemented:

Trap MRS reads of TCR2MASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTCR2MASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of TCR2MASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TCR2MASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCRMASK_EL1, bit [6]

When FEAT_SRMASK is implemented:

Trap MRS reads of TCRMASK_EL1 at EL1 using AArch64 to EL2.

nTCRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of TCRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TCRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLR2MASK_EL1, bit [5]

When FEAT_SRMASK is implemented:

Trap MRS reads of SCTLR2MASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSCTLR2MASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SCTLR2MASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SCTLR2MASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLRMASK_EL1, bit [4]

When FEAT_SRMASK is implemented:

Trap MRS reads of SCTLRMASK_EL1 at EL1 using AArch64 to EL2.

nSCTLRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of SCTLRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SCTLRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nCPACRMASK_EL1, bit [3]

When FEAT_SRMASK is implemented:

Trap MRS reads of CPACRMASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nCPACRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of CPACRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of CPACRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nRCWSMASK_EL1, bit [2]

When FEAT_THE is implemented:

Trap MRS or MRRS reads of RCWSMASK_EL1 at EL1 using AArch64 to EL2.

nRCWSMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then unless the read generates a higher priority exception: • MRS reads of RCWSMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRRS reads of RCWSMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.
0b1	MRS and MRRS reads of RCWSMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nERXGSR_EL1, bit [1]

When FEAT_RASv2 is implemented:

Trap MRS reads of ERXGSR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nERXGSR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of ERXGSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of ERXGSR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPFAR_EL1, bit [0]

When FEAT_PFAR is implemented:

Trap MRS reads of PFAR_EL1 at EL1 using AArch64 to EL2.

nPFAR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MRS reads of PFAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PFAR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HFGRTR2_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, HFGRTR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x2C0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HFGRTR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HFGRTR2_EL2();
end;
```

```
MSR HFGRTR2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x2C0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HFGRTR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HFGRTR2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.72 HFGRTR_EL2, Hypervisor Fine-Grained Read Trap Register

The HFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of `MRRS`, `MRS` and `MRC` reads of System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HFGRTR_EL2 are UNDEFINED.

Attributes

HFGRTR_EL2 is a 64-bit register.

Field descriptions

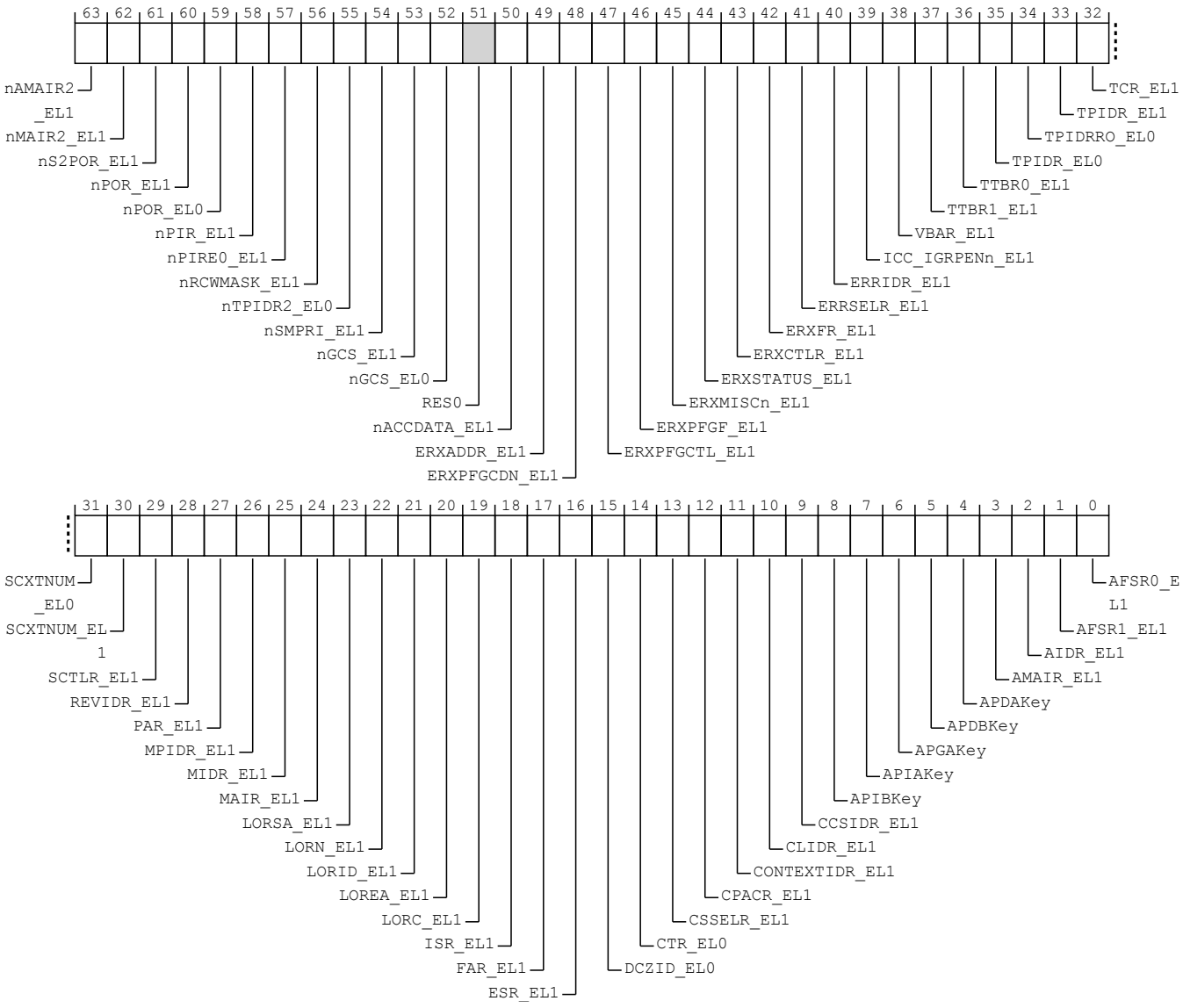

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nAMAIR2_EL1, bit [63]

When FEAT_AIE is implemented:

Trap MRS reads of AMAIR2_EL1 at EL1 using AArch64 to EL2.

nAMAIR2_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AMAIR2_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of AMAIR2_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nMAIR2_EL1, bit [62]

When FEAT_AIE is implemented:

Trap MRS reads of MAIR2_EL1 at EL1 using AArch64 to EL2.

nMAIR2_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MAIR2_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of MAIR2_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nS2POR_EL1, bit [61]

When FEAT_S2POE is implemented:

Trap MRS reads of S2POR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nS2POR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of S2POR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of S2POR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPOR_EL1, bit [60]

When FEAT_S1POE is implemented:

Trap MRS reads of POR_EL1 at EL1 using AArch64 to EL2.

nPOR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of POR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of POR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPOR_EL0, bit [59]

When FEAT_S1POE is implemented:

Trap MRS reads of POR_EL0 at EL1 and EL0 using AArch64 to EL2.

nPOR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of POR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nPOR_EL0	Meaning
0b1	MRS reads of POR_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPIR_EL1, bit [58]

When FEAT_S1PIE is implemented:

Trap MRS reads of PIR_EL1 at EL1 using AArch64 to EL2.

nPIR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PIR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPIRE0_EL1, bit [57]

When FEAT_S1PIE is implemented:

Trap MRS reads of PIRE0_EL1 at EL1 using AArch64 to EL2.

nPIRE0_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PIRE0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of PIRE0_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nRCWMASK_EL1, bit [56]

When FEAT_THE is implemented:

Trap `MRS` or `MRRS` reads of RCWMASK_EL1 at EL1 using AArch64 to EL2.

nRCWMASK_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • `MRS` reads of RCWMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`. • `MRRS` reads of RCWMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x14`.
`0b1`	`MRS` and `MRRS` reads of RCWMASK_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTPIDR2_EL0, bit [55]

When FEAT_SME is implemented:

Trap `MRS` reads of TPIDR2_EL0 at EL1 and EL0 using AArch64 to EL2.

nTPIDR2_EL0	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then `MRS` reads of TPIDR2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.
`0b1`	`MRS` reads of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

nSMPRI_EL1, bit [54]

When FEAT_SME is implemented:

Trap MRS reads of SMPRI_EL1 at EL1 using AArch64 to EL2.

nSMPRI_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCS_EL1, bit [53]

When FEAT_GCS is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- GCSCR_EL1.
- GCSPR_EL1.

nGCS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCS_EL0, bit [52]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_GCS is implemented:

Trap MRS reads at EL1 and EL0 using AArch64 of any of the following AArch64 System registers to EL2:

- GCSCRE0_EL1, at EL1 only.
- GCSPR_EL0.

nGCS_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 and EL0 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [51]

Reserved, RES0.

nACCDATA_EL1, bit [50]

When FEAT_LS64_ACCDATA is implemented:

Trap MRS reads of ACCDATA_EL1 at EL1 using AArch64 to EL2.

nACCDATA_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of ACCDATA_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXADDR_EL1, bit [49]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RAS is implemented:

Trap MRS reads of ERXADDR_EL1 at EL1 using AArch64 to EL2.

ERXADDR_EL1	Meaning
0b0	MRS reads of ERXADDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

ERXPFGCDN_EL1	Meaning
0b0	MRS reads of ERXPFGCDN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ERXPFGCTL_EL1, bit [47]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 to EL2.

ERXPFGCTL_EL1	Meaning
0b0	MRS reads of ERXPFGCTL_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXPFGF_EL1, bit [46]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGF_EL1 at EL1 using AArch64 to EL2.

ERXPFGF_EL1	Meaning
0b0	MRS reads of ERXPFGF_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGF_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- ERXMISC0_EL1.
- ERXMISC1_EL1.
- ERXMISC2_EL1.
- ERXMISC3_EL1.

ERXMISCn_EL1	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is implemented:

Trap MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

ERXSTATUS_EL1	Meaning
0b0	MRS reads of ERXSTATUS_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
- ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXCTLR_EL1, bit [43]

When FEAT_RAS is implemented:

Trap MRS reads of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

ERXCTLR_EL1	Meaning
`0b0`	MRS reads of ERXCTLR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXFR_EL1, bit [42]

When FEAT_RAS is implemented:

Trap MRS reads of ERXFR_EL1 at EL1 using AArch64 to EL2.

ERXFR_EL1	Meaning
`0b0`	MRS reads of ERXFR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:

Trap MRS reads of ERRSELR_EL1 at EL1 using AArch64 to EL2.

ERRSELR_EL1	Meaning
`0b0`	MRS reads of ERRSELR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERRIDR_EL1, bit [40]

When FEAT_RAS is implemented:

Trap MRS reads of ERRIDR_EL1 at EL1 using AArch64 to EL2.

ERRIDR_EL1	Meaning
`0b0`	MRS reads of ERRIDR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ICC_IGRPENn_EL1, bit [39]

When GICv3 is implemented:

Trap MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.

ICC_IGRPENn_EL1	Meaning
`0b0`	MRS reads of ICC_IGRPEN<n>_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VBAR_EL1, bit [38]

Trap MRS reads of VBAR_EL1 at EL1 using AArch64 to EL2.

VBAR_EL1	Meaning
`0b0`	MRS reads of VBAR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTBR1_EL1, bit [37]

Trap MRS or MRRS reads of TTBR1_EL1 at EL1 using AArch64 to EL2.

TTBR1_EL1	Meaning
0b0	MRS and MRRS reads of TTBR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRRS reads of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TTBR0_EL1, bit [36]

Trap MRS or MRRS reads of TTBR0_EL1 at EL1 using AArch64 to EL2.

TTBR0_EL1	Meaning
0b0	MRS and MRRS reads of TTBR0_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRRS reads of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPIDR_EL0, bit [35]

Trap MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPIDR_EL0	Meaning
0b0	MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPIDRRO_EL0, bit [34]

Trap MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 when EL1 is using AArch64 to EL2.

TPIDRRO_EL0	Meaning
0b0	MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of TPIDRURO at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPIDR_EL1, bit [33]

Trap MRS reads of TPIDR_EL1 at EL1 using AArch64 to EL2.

TPIDR_EL1	Meaning
0b0	MRS reads of TPIDR_EL1 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TCR_EL1, bit [32]

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TCR_EL1.
- TCR2_EL1, if FEAT_TCR2 is implemented.

TCR_EL1	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SCXTNUM_EL0, bit [31]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

SCXTNUM_EL0	Meaning
0b0	MRS reads of SCXTNUM_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

SCXTNUM_EL1	Meaning
0b0	MRS reads of SCXTNUM_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SCTLR_EL1, bit [29]

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- SCTLR_EL1.
- SCTLR2_EL1, if FEAT_SCTLR2 is implemented.

SCTLR_EL1	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

REVIDR_EL1, bit [28]

Trap MRS reads of REVIDR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

REVIDR_EL1	Meaning
0b0	MRS reads of REVIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of REVIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

PAR_EL1, bit [27]

Trap MRS or MRRS reads of PAR_EL1 at EL1 using AArch64 to EL2.

PAR_EL1	Meaning
0b0	MRS and MRRS reads of PAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the read generates a higher priority exception: • MRS reads of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRRS reads of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

MPIDR_EL1, bit [26]

Trap MRS reads of MPIDR_EL1 at EL1 using AArch64 to EL2.

MPIDR_EL1	Meaning
0b0	MRS reads of MPIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MIDR_EL1, bit [25]

Trap MRS reads of MIDR_EL1 at EL1 using AArch64 to EL2.

MIDR_EL1	Meaning
0b0	MRS reads of MIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

MAIR_EL1, bit [24]

Trap MRS reads of MAIR_EL1 at EL1 using AArch64 to EL2.

MAIR_EL1	Meaning
0b0	MRS reads of MAIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:

Trap MRS reads of LORSA_EL1 at EL1 using AArch64 to EL2.

LORSA_EL1	Meaning
0b0	MRS reads of LORSA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LORN_EL1, bit [22]

When FEAT_LOR is implemented:

Trap MRS reads of LORN_EL1 at EL1 using AArch64 to EL2.

LORN_EL1	Meaning
0b0	MRS reads of LORN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LORID_EL1, bit [21]

When FEAT_LOR is implemented:

Trap MRS reads of LORID_EL1 at EL1 using AArch64 to EL2.

LORID_EL1	Meaning
0b0	MRS reads of LORID_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORID_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LOREA_EL1, bit [20]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LOR is implemented:

Trap MRS reads of LOREA_EL1 at EL1 using AArch64 to EL2.

LOREA_EL1	Meaning
0b0	MRS reads of LOREA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap MRS reads of LORC_EL1 at EL1 using AArch64 to EL2.

LORC_EL1	Meaning
0b0	MRS reads of LORC_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ISR_EL1, bit [18]

Trap MRS reads of ISR_EL1 at EL1 using AArch64 to EL2.

ISR_EL1	Meaning
0b0	MRS reads of ISR_EL1 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ISR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

FAR_EL1, bit [17]

Trap MRS reads of FAR_EL1 at EL1 using AArch64 to EL2.

FAR_EL1	Meaning
0b0	MRS reads of FAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ESR_EL1, bit [16]

Trap MRS reads of ESR_EL1 at EL1 using AArch64 to EL2.

ESR_EL1	Meaning
0b0	MRS reads of ESR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DCZID_EL0, bit [15]

Trap MRS reads of DCZID_EL0 at EL1 and EL0 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DCZID_EL0	Meaning
0b0	MRS reads of DCZID_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DCZID_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CTR_EL0, bit [14]

Trap MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 to EL2.

CTR_EL0	Meaning
0b0	MRS reads of CTR_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CSSELR_EL1, bit [13]

Trap MRS reads of CSSELR_EL1 at EL1 using AArch64 to EL2.

CSSELR_EL1	Meaning
0b0	MRS reads of CSSELR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CPACR_EL1, bit [12]

Trap MRS reads of CPACR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CPACR_EL1	Meaning
0b0	MRS reads of CPACR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CONTEXTIDR_EL1, bit [11]

Trap MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

CONTEXTIDR_EL1	Meaning
0b0	MRS reads of CONTEXTIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CLIDR_EL1, bit [10]

Trap MRS reads of CLIDR_EL1 at EL1 using AArch64 to EL2.

CLIDR_EL1	Meaning
0b0	MRS reads of CLIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CLIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCSIDR_EL1, bit [9]

Trap MRS reads of CCSIDR_EL1 at EL1 using AArch64 to EL2.

CCSIDR_EL1	Meaning
0b0	MRS reads of CCSIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CCSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

APIBKey, bit [8]

When FEAT_PAuth is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

APIBKey	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

APIAKey	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

APGAKey	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

APDBKey	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APDAKey, bit [4]

When FEAT_PAuth is implemented:

Trap MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

APDAKey	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMAIR_EL1, bit [3]

Trap MRS reads of AMAIR_EL1 at EL1 using AArch64 to EL2.

AMAIR_EL1	Meaning
0b0	MRS reads of AMAIR_EL1 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMAIR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AIDR_EL1, bit [2]

Trap MRS reads of AIDR_EL1 at EL1 using AArch64 to EL2.

AIDR_EL1	Meaning
0b0	MRS reads of AIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AFSR1_EL1, bit [1]

Trap MRS reads of AFSR1_EL1 at EL1 using AArch64 to EL2.

AFSR1_EL1	Meaning
0b0	MRS reads of AFSR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AFSR0_EL1, bit [0]

Trap MRS reads of AFSR0_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AFSR0_EL1	Meaning
0b0	MRS reads of AFSR0_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HFGRTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HFGRTR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1B8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HFGRTR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HFGRTR_EL2();
end;
```

```
MSR HFGRTR_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1B8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HFGRTR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HFGRTR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.73 HFGWTR2_EL2, Hypervisor Fine-Grained Write Trap Register 2

The HFGWTR2_EL2 characteristics are:

Purpose

Provides controls for traps of MSRR, MSR and MCR writes of System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HFGWTR2_EL2 are UNDEFINED.

Attributes

HFGWTR2_EL2 is a 64-bit register.

Field descriptions

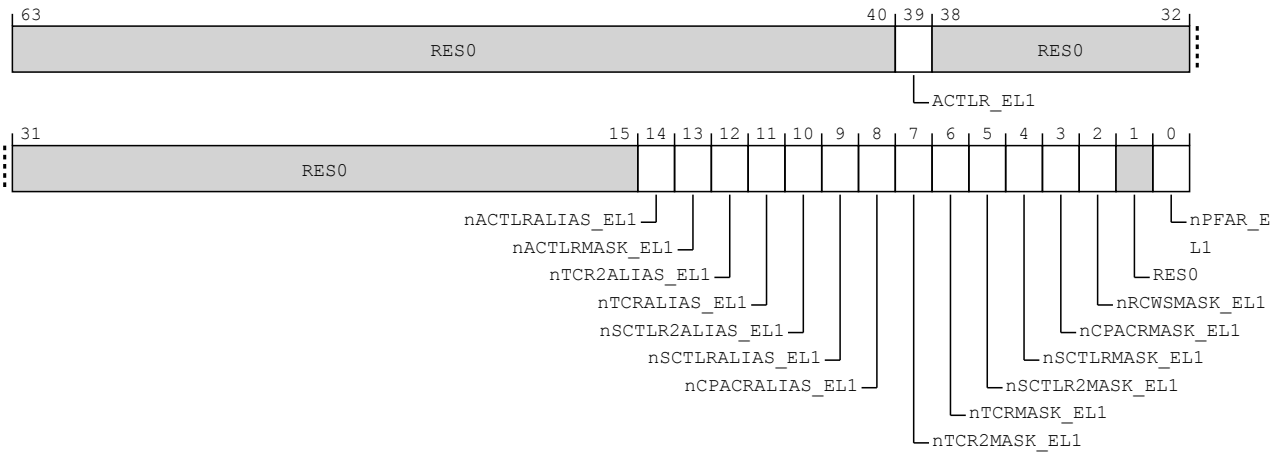

Bits [63:40]

Reserved, RES0.

ACTLR_EL1, bit [39]

When FEAT_SRMASK is implemented:

Trap MSR writes of ACTLR_EL1 at EL1 using AArch64 to EL2.

ACTLR_EL1	Meaning
0b0	MSR writes of ACTLR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, then MSR writes of ACTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [38:15]

Reserved, RES0.

nACTLRALIAS_EL1, bit [14]

When FEAT_SRMASK is implemented:

Trap MSR writes of ACTLRALIAS_EL1 at EL1 using AArch64 to EL2.

nACTLRALIAS_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of ACTLRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	MSR writes of ACTLRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nACTLRMASK_EL1, bit [13]

When FEAT_SRMASK is implemented:

Trap MSR writes of ACTLRMASK_EL1 at EL1 using AArch64 to EL2.

nACTLRMASK_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, then MSR writes of ACTLRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	MSR writes of ACTLRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCR2ALIAS_EL1, bit [12]

When FEAT_SRMASK is implemented:

Trap MSR writes of TCR2ALIAS_EL1 at EL1 using AArch64 to EL2.

nTCR2ALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of TCR2ALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of TCR2ALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCRALIAS_EL1, bit [11]

When FEAT_SRMASK is implemented:

Trap MSR writes of TCRALIAS_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTCRALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of TCRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of TCRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLR2ALIAS_EL1, bit [10]

When FEAT_SRMASK is implemented:

Trap MSR writes of SCTLR2ALIAS_EL1 at EL1 using AArch64 to EL2.

nSCTLR2ALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of SCTLR2ALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SCTLR2ALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLRALIAS_EL1, bit [9]

When FEAT_SRMASK is implemented:

Trap MSR writes of SCTLRALIAS_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSCTLRALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of SCTLRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SCTLRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nCPACRALIAS_EL1, bit [8]

When FEAT_SRMASK is implemented:

Trap MSR writes of CPACRALIAS_EL1 at EL1 using AArch64 to EL2.

nCPACRALIAS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of CPACRALIAS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of CPACRALIAS_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCR2MASK_EL1, bit [7]

When FEAT_SRMASK is implemented:

Trap MSR writes of TCR2MASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTCR2MASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of TCR2MASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of TCR2MASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTCRMASK_EL1, bit [6]

When FEAT_SRMASK is implemented:

Trap MSR writes of TCRMASK_EL1 at EL1 using AArch64 to EL2.

nTCRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of TCRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of TCRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLR2MASK_EL1, bit [5]

When FEAT_SRMASK is implemented:

Trap MSR writes of SCTLR2MASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nSCTLR2MASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of SCTLR2MASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SCTLR2MASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nSCTLRMASK_EL1, bit [4]

When FEAT_SRMASK is implemented:

Trap MSR writes of SCTLRMASK_EL1 at EL1 using AArch64 to EL2.

nSCTLRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of SCTLRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SCTLRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nCPACRMASK_EL1, bit [3]

When FEAT_SRMASK is implemented:

Trap MSR writes of CPACRMASK_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nCPACRMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of CPACRMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of CPACRMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nRCWSMASK_EL1, bit [2]

When FEAT_THE is implemented:

Trap MSR or MSRR writes of RCWSMASK_EL1 at EL1 using AArch64 to EL2.

nRCWSMASK_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then unless the write generates a higher priority exception: • MSR writes of RCWSMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MSRR writes of RCWSMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.
0b1	MSR and MSRR writes of RCWSMASK_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [1]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nPFAR_EL1, bit [0]

When FEAT_PFAR is implemented:

Trap MSR writes of PFAR_EL1 at EL1 using AArch64 to EL2.

nPFAR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, then MSR writes of PFAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of PFAR_EL1 are not trapped by this mechanism.

This field is ignored by the PE and treated as zero when all of the following are true:

- EL3 is implemented.
- SCR_EL3.FGTEn2 == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HFGWTR2_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HFGWTR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x2C8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = HFGWTR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = HFGWTR2_EL2();
end;
```

```
MSR HFGWTR2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0011	0b0001	0b011

```
if !(IsFeatureImplemented(FEAT_FGT2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x2C8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HFGWTR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HFGWTR2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.74 HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register

The HFGWTR_EL2 characteristics are:

Purpose

Provides controls for traps of MSRR, MSR and MCR writes of System registers.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_FGT is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to HFGWTR_EL2 are UNDEFINED.

Attributes

HFGWTR_EL2 is a 64-bit register.

Field descriptions

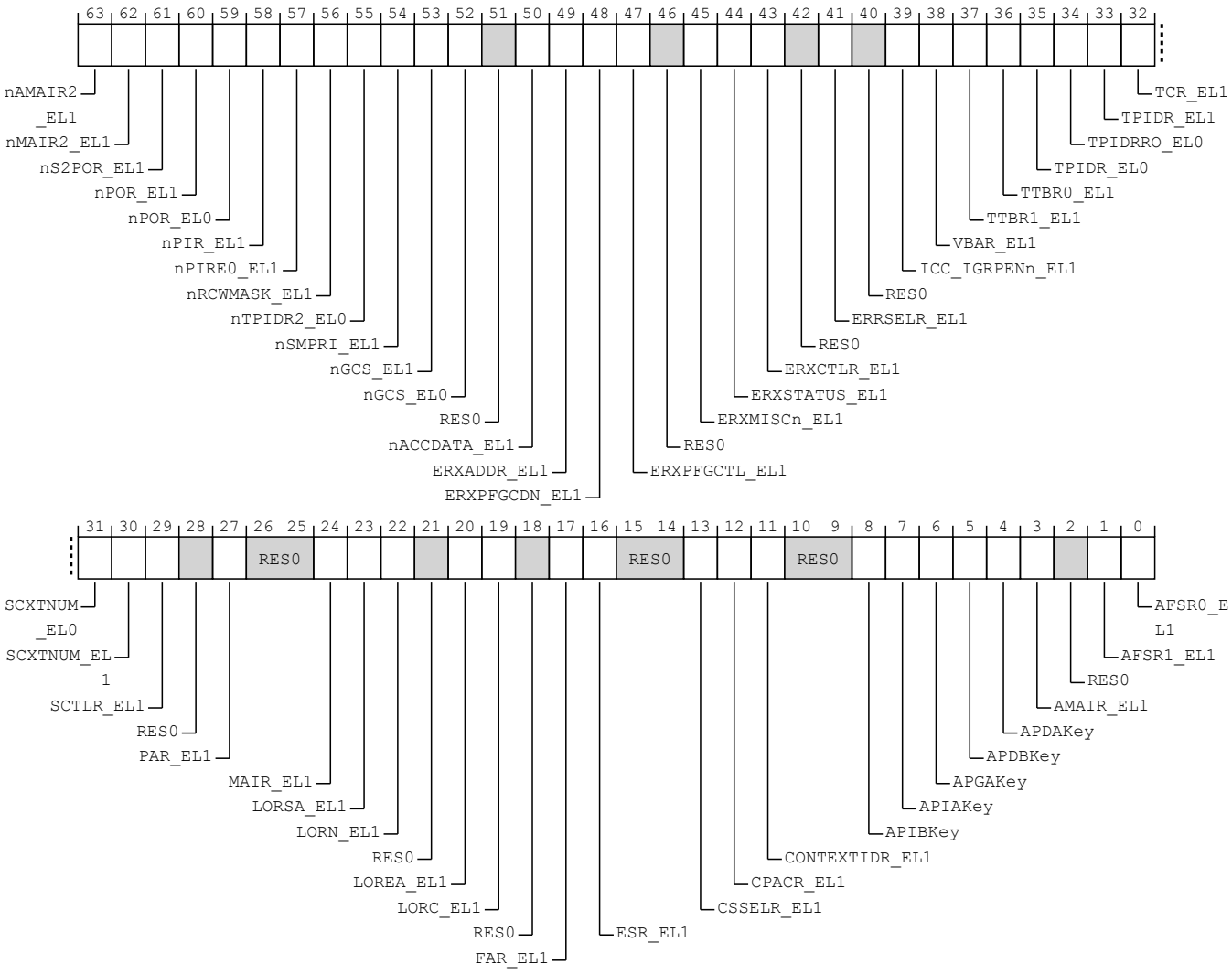

nAMAIR2_EL1, bit [63]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AIE is implemented:

Trap MSR writes of AMAIR2_EL1 at EL1 using AArch64 to EL2.

nAMAIR2_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AMAIR2_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of AMAIR2_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nMAIR2_EL1, bit [62]

When FEAT_AIE is implemented:

Trap MSR writes of MAIR2_EL1 at EL1 using AArch64 to EL2.

nMAIR2_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of MAIR2_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of MAIR2_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nS2POR_EL1, bit [61]

When FEAT_S2POE is implemented:

Trap MSR writes of S2POR_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nS2POR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of S2POR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of S2POR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPOR_EL1, bit [60]

When FEAT_S1POE is implemented:

Trap MSR writes of POR_EL1 at EL1 using AArch64 to EL2.

nPOR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of POR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of POR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPOR_EL0, bit [59]

When FEAT_S1POE is implemented:

Trap MSR writes of POR_EL0 at EL1 and EL0 using AArch64 to EL2.

nPOR_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of POR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nPOR_EL0	Meaning
0b1	MSR writes of POR_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPIR_EL1, bit [58]

When FEAT_S1PIE is implemented:

Trap MSR writes of PIR_EL1 at EL1 using AArch64 to EL2.

nPIR_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of PIR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nPIRE0_EL1, bit [57]

When FEAT_S1PIE is implemented:

Trap MSR writes of PIRE0_EL1 at EL1 using AArch64 to EL2.

nPIRE0_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PIRE0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of PIRE0_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nRCWMASK_EL1, bit [56]

When FEAT_THE is implemented:

Trap `MSR` or `MSRR` writes of RCWMASK_EL1 at EL1 using AArch64 to EL2.

nRCWMASK_EL1	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • `MSR` writes of RCWMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`. • `MSRR` writes of RCWMASK_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x14`.
`0b1`	`MSR` and `MSRR` writes of RCWMASK_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTPIDR2_EL0, bit [55]

When FEAT_SME is implemented:

Trap `MSR` writes of TPIDR2_EL0 at EL1 and EL0 using AArch64 to EL2.

nTPIDR2_EL0	Meaning
`0b0`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then `MSR` writes of TPIDR2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.
`0b1`	`MSR` writes of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

nSMPRI_EL1, bit [54]

When FEAT_SME is implemented:

Trap MSR writes of SMPRI_EL1 at EL1 using AArch64 to EL2.

nSMPRI_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCS_EL1, bit [53]

When FEAT_GCS is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- GCSCR_EL1.
- GCSPR_EL1.

nGCS_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nGCS_EL0, bit [52]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_GCS is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- GCSCRE0_EL1.
- GCSPR_EL0.

nGCS_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of the specified System registers are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [51]

Reserved, RES0.

nACCDATA_EL1, bit [50]

When FEAT_LS64_ACCDATA is implemented:

Trap MSR writes of ACCDATA_EL1 at EL1 using AArch64 to EL2.

nACCDATA_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of ACCDATA_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXADDR_EL1, bit [49]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RAS is implemented:

Trap MSR writes of ERXADDR_EL1 at EL1 using AArch64 to EL2.

ERXADDR_EL1	Meaning
0b0	MSR writes of ERXADDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

ERXPFGCDN_EL1	Meaning
0b0	MSR writes of ERXPFGCDN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ERXPFGCTL_EL1, bit [47]

When FEAT_RASv1p1 is implemented:

Trap MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 to EL2.

ERXPFGCTL_EL1	Meaning
0b0	MSR writes of ERXPFGCTL_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [46]

Reserved, RES0.

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- ERXMISC0_EL1.
- ERXMISC1_EL1.
- ERXMISC2_EL1.
- ERXMISC3_EL1.

ERXMISCn_EL1	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is implemented:

Trap MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

ERXSTATUS_EL1	Meaning
`0b0`	MSR writes of ERXSTATUS_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ERXCTLR_EL1, bit [43]

When FEAT_RAS is implemented:

Trap MSR writes of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

ERXCTLR_EL1	Meaning
`0b0`	MSR writes of ERXCTLR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:

Trap MSR writes of ERRSELR_EL1 at EL1 using AArch64 to EL2.

ERRSELR_EL1	Meaning
`0b0`	MSR writes of ERRSELR_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [40]

Reserved, RES0.

ICC_IGRPENn_EL1, bit [39]

When GICv3 is implemented:

Trap MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ICC_IGRPENn_EL1	Meaning
0b0	MSR writes of ICC_IGRPEN<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VBAR_EL1, bit [38]

Trap MSR writes of VBAR_EL1 at EL1 using AArch64 to EL2.

VBAR_EL1	Meaning
0b0	MSR writes of VBAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TTBR1_EL1, bit [37]

Trap MSR or MSRR writes of TTBR1_EL1 at EL1 using AArch64 to EL2.

TTBR1_EL1	Meaning
0b0	MSR and MSRR writes of TTBR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MSRR writes of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TTBR0_EL1, bit [36]

Trap `MSR` or `MSRR` writes of TTBR0_EL1 at EL1 using AArch64 to EL2.

TTBR0_EL1	Meaning
`0b0`	`MSR` and `MSRR` writes of TTBR0_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • `MSR` writes of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`. • `MSRR` writes of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x14`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPIDR_EL0, bit [35]

Trap `MSR` writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and `MCR` writes of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

TPIDR_EL0	Meaning
`0b0`	`MSR` writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and `MCR` writes of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • `MSR` writes of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`. • `MCR` writes of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 are trapped to EL2 and reported with EC syndrome value `0x03`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPIDRRO_EL0, bit [34]

Trap `MSR` writes of TPIDRRO_EL0 at EL1 using AArch64 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPIDRRO_EL0	Meaning
0b0	MSR writes of TPIDRRO_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDRRO_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPIDR_EL1, bit [33]

Trap MSR writes of TPIDR_EL1 at EL1 using AArch64 to EL2.

TPIDR_EL1	Meaning
0b0	MSR writes of TPIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TCR_EL1, bit [32]

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- TCR_EL1.
- TCR2_EL1, if FEAT_TCR2 is implemented.

TCR_EL1	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SCXTNUM_EL0, bit [31]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

SCXTNUM_EL0	Meaning
0b0	MSR writes of SCXTNUM_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MSR writes of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

SCXTNUM_EL1	Meaning
0b0	MSR writes of SCXTNUM_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SCTLR_EL1, bit [29]

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- SCTLR_EL1.
- SCTLR2_EL1, if FEAT_SCTLR2 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCTLR_EL1	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [28]

Reserved, RES0.

PAR_EL1, bit [27]

Trap MSR or MSRR writes of PAR_EL1 at EL1 using AArch64 to EL2.

PAR_EL1	Meaning
0b0	MSR and MSRR writes of PAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then unless the write generates a higher priority exception: • MSR writes of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MSRR writes of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x14.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [26:25]

Reserved, RES0.

MAIR_EL1, bit [24]

Trap MSR writes of MAIR_EL1 at EL1 using AArch64 to EL2.

MAIR_EL1	Meaning
0b0	MSR writes of MAIR_EL1 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MAIR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:

Trap MSR writes of LORSA_EL1 at EL1 using AArch64 to EL2.

LORSA_EL1	Meaning
0b0	MSR writes of LORSA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LORN_EL1, bit [22]

When FEAT_LOR is implemented:

Trap MSR writes of LORN_EL1 at EL1 using AArch64 to EL2.

LORN_EL1	Meaning
0b0	MSR writes of LORN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

LOREA_EL1, bit [20]

When FEAT_LOR is implemented:

Trap MSR writes of LOREA_EL1 at EL1 using AArch64 to EL2.

LOREA_EL1	Meaning
0b0	MSR writes of LOREA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap MSR writes of LORC_EL1 at EL1 using AArch64 to EL2.

LORC_EL1	Meaning
0b0	MSR writes of LORC_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [18]

Reserved, RES0.

FAR_EL1, bit [17]

Trap MSR writes of FAR_EL1 at EL1 using AArch64 to EL2.

FAR_EL1	Meaning
0b0	MSR writes of FAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

ESR_EL1, bit [16]

Trap MSR writes of ESR_EL1 at EL1 using AArch64 to EL2.

ESR_EL1	Meaning
0b0	MSR writes of ESR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [15:14]

Reserved, RES0.

CSSELR_EL1, bit [13]

Trap MSR writes of CSSELR_EL1 at EL1 using AArch64 to EL2.

CSSELR_EL1	Meaning
0b0	MSR writes of CSSELR_EL1 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSSELR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CPACR_EL1, bit [12]

Trap MSR writes of CPACR_EL1 at EL1 using AArch64 to EL2.

CPACR_EL1	Meaning
0b0	MSR writes of CPACR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CONTEXTIDR_EL1, bit [11]

Trap MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

CONTEXTIDR_EL1	Meaning
0b0	MSR writes of CONTEXTIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [10:9]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

APIBKey, bit [8]

When FEAT_PAuth is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

APIBKey	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

APIAKey	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

APGAKey	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

APDBKey	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APDAKey, bit [4]

When FEAT_PAuth is implemented:

Trap MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

APDAKey	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMAIR_EL1, bit [3]

Trap MSR writes of AMAIR_EL1 at EL1 using AArch64 to EL2.

AMAIR_EL1	Meaning
0b0	MSR writes of AMAIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

AFSR1_EL1, bit [1]

Trap MSR writes of AFSR1_EL1 at EL1 using AArch64 to EL2.

AFSR1_EL1	Meaning
0b0	MSR writes of AFSR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AFSR0_EL1, bit [0]

Trap `MSR` writes of AFSR0_EL1 at EL1 using AArch64 to EL2.

AFSR0_EL1	Meaning
`0b0`	`MSR` writes of AFSR0_EL1 are not trapped by this mechanism.
`0b1`	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then `MSR` writes of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HFGWTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HFGWTR_EL2
```

op0	op1	CRn	CRm	op2
`0b11`	`0b100`	`0b0001`	`0b0001`	`0b101`

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1C0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = HFGWTR_EL2();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = HFGWTR_EL2();
end;
```

```
MSR HFGWTR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b101

```
if !(IsFeatureImplemented(FEAT_FGT) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1C0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FGTEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FGTEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        HFGWTR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    HFGWTR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.75 HPFAR_EL2, Hypervisor IPA Fault Address Register

The HPFAR_EL2 characteristics are:

Purpose

Holds the faulting IPA for some aborts on a stage 2 translation taken to EL2 and GPC exceptions due to a fault on an access for a stage 2 translation.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

The HPFAR_EL2 is written for:

- A Translation fault, Access Flag fault, or Address Size fault on a stage 2 translation not on a stage 1 translation table walk.
- A Translation fault, Access Flag fault, Address Size fault, or Permission fault on stage 2 translation of an address accessed in a stage 1 translation table walk.
- If FEAT_RME is implemented, a Granule Protection Check fault in the second stage of translation.

For all other exceptions taken to EL2, this register is UNKNOWN.

Note

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the Instruction Abort exception or Data Abort exception. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

AArch64 System register HPFAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HPFAR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to HPFAR_EL2 are UNDEFINED.

Attributes

HPFAR_EL2 is a 64-bit register.

Field descriptions

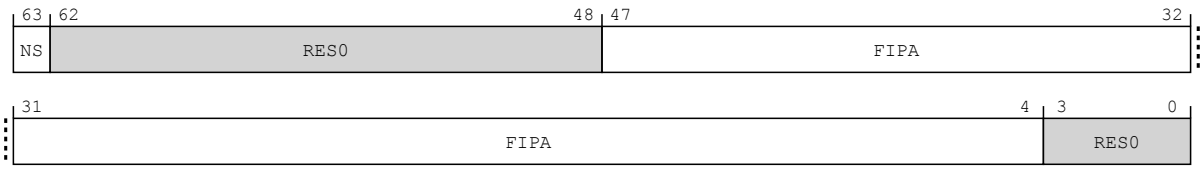

Execution at EL1 or EL0 makes HPFAR_EL2 become UNKNOWN.

NS, bit [63]

When FEAT_SEL2 is implemented:

Faulting IPA address space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	Faulting IPA is from the Secure IPA space.
0b1	Faulting IPA is from the Non-secure IPA space.

For Data Abort exceptions or Instruction Abort exceptions taken to Non-secure EL2:

- This field is RES0.
- The address is from the Non-secure IPA space.

If FEAT_RME is implemented, for Data Abort exceptions or Instruction Abort exceptions taken to Realm EL2:

- This field is RES0.
- The address is from the Realm IPA space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

FIPA, bits [47:4]

When FEAT_D128 is implemented:

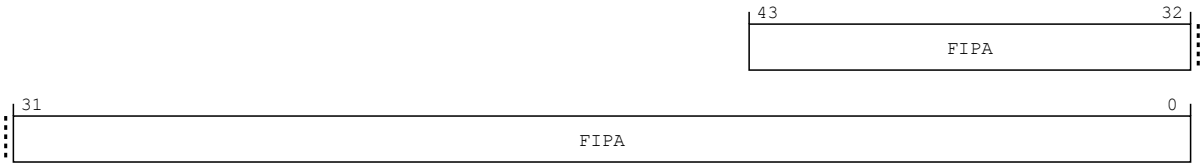

FIPA, bits [43:0]

Bits [55:12] of the Faulting Intermediate Physical Address.

For implementations with fewer than 55 physical address bits, the corresponding upper bits in this field are RES0.

When FEAT_MOPS is implemented, the value presented in FIPA on a synchronous exception that set the HPFAR_EL2 from any of the Memory Copy and Memory Set instructions is within the address range of the current stage 2 translation granule, aligned to the size of the current stage 2 translation granule, of the address that generated the Data abort. In this case, bits[(n-1):0] of the value are UNKNOWN, where 2^n is the current stage 2 translation granule size in bytes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LPA is implemented and FEAT_D128 is not implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

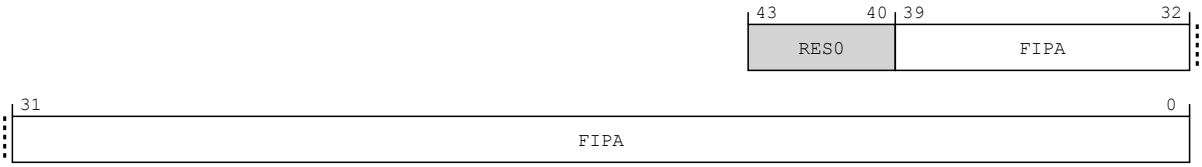

Bits [43:40]

Reserved, RES0.

FIPA, bits [39:0]

Bits [51:12] of the Faulting Intermediate Physical Address.

For implementations with fewer than 52 physical address bits, the corresponding upper bits in this field are RES0.

When FEAT_MOPS is implemented, the value presented in FIPA on a synchronous exception that set the HPFAR_EL2 from any of the Memory Copy and Memory Set instructions is within the address range of the current stage 2 translation granule, aligned to the size of the current stage 2 translation granule, of the address that generated the Data abort. In this case, bits[(n-1):0] of the value are UNKNOWN, where 2^n is the current stage 2 translation granule size in bytes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LPA is not implemented:

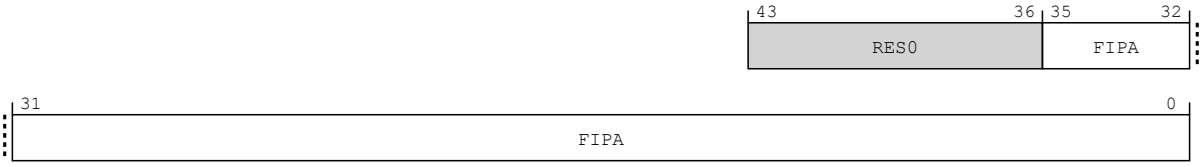

Bits [43:36]

Reserved, RES0.

FIPA, bits [35:0]

Bits[47:12] Faulting Intermediate Physical Address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

When FEAT_MOPS is implemented, the value presented in FIPA on a synchronous exception that set the HPFAR_EL2 from any of the Memory Copy and Memory Set instructions is within the address range of the current stage 2 translation granule, aligned to the size of the current stage 2 translation granule, of the address that generated the Data abort. In this case, bits[(n-1):0] of the value are UNKNOWN, where 2^n is the current stage 2 translation granule size in bytes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing HPFAR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HPFAR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = HPFAR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = HPFAR_EL2();
end;
```

```
MSR HPFAR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HPFAR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    HPFAR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.76 HSTR_EL2, Hypervisor System Trap Register

The HSTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of EL1 or lower AArch32 accesses to the System register in the coproc == `0b1111` encoding space, by the CRn value used to access the register using MCR or MRC instruction. When the register is accessible using an MCRR or MRRC instruction, this is the CRm value used to access the register.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register HSTR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HSTR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to HSTR_EL2 are UNDEFINED.

Attributes

HSTR_EL2 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

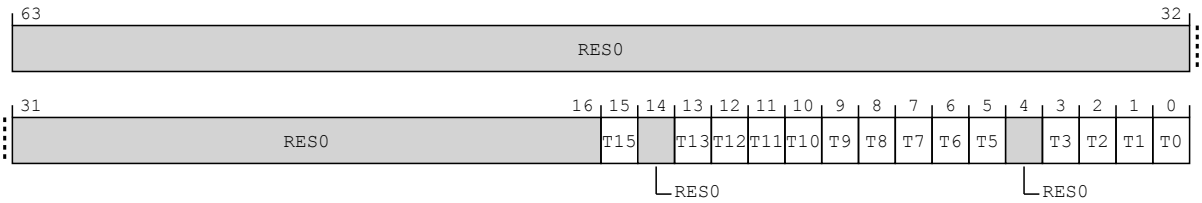

Bits [63:16, 14, 4]

Reserved, RES0.

T<n> , bits [n], for n = 15 to 15, 13 to 5, 3 to 0

The remaining fields control whether EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the System registers in the coproc == `0b1111` encoding space, are trapped to EL2 as follows:

- MCR or MRC accesses to these registers that are trapped to EL2 are reported using EC syndrome value `0x03`, unless the access is UNDEFINED.
- MCRR or MRRC accesses to these registers that are trapped to EL2 are reported using EC syndrome value `0x04`, unless the access is UNDEFINED.

T<n>	Meaning
`0b0`	This control has no effect on EL0 or EL1 accesses to System registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T<n>	Meaning
0b1	System registers in the coproc == 0b1111 encoding space and CRn == <n> or CRm == <n> where T<n> is the name of this field, are trapped as follows: • An EL1 MCR or MRC access is trapped to EL2. • An EL0 MCR or MRC access is trapped to EL2, if the access is not UNDEFINED when the value of this field is 0. • An EL1 MCRR or MRRC access is trapped to EL2. • An EL0 MCRR or MRRC access is trapped to EL2, if the access is not UNDEFINED when the value of this field is 0. It is IMPLEMENTATION DEFINED whether an EL0 access using AArch32 is trapped to EL2, or is UNDEFINED. If the access is UNDEFINED, and generates an exception that is taken to EL1 or EL2 using AArch64, this is reported with EC syndrome value 0x00. **Note** Arm expects that trapping to EL2 of EL0 accesses to these registers is unusual and used only when the hypervisor must virtualize EL0 operation. Arm recommends that, whenever possible, EL0 accesses to these registers behave as they would if the implementation did not include EL2. This means that, if the architecture does not support the EL0 access, then the register access instruction is treated as UNDEFINED and generates an exception that is taken to EL1.

For example, when HSTR_EL2.T7 is 1, for instructions executed at EL1:

- An MCR or MRC instruction with coproc set to 0b1111 and <CRn> set to c7 is trapped to EL2.
- An MCRR or MRRC instruction with coproc set to 0b1111 and <CRm> set to c7 is trapped to EL2.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

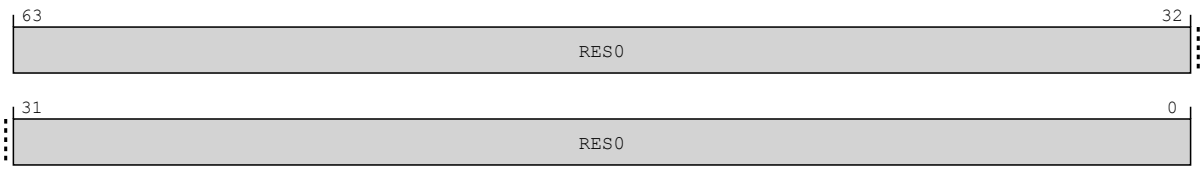

Bits [63:0]

Reserved, RES0.

Accessing HSTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, HSTR_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x080);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = HSTR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = HSTR_EL2();
end;
```

MSR HSTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x080) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSTR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    HSTR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.77 ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0

The ID_AA64AFR0_EL1 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64AFR0_EL1 are UNDEFINED.

Attributes

ID_AA64AFR0_EL1 is a 64-bit register.

Field descriptions

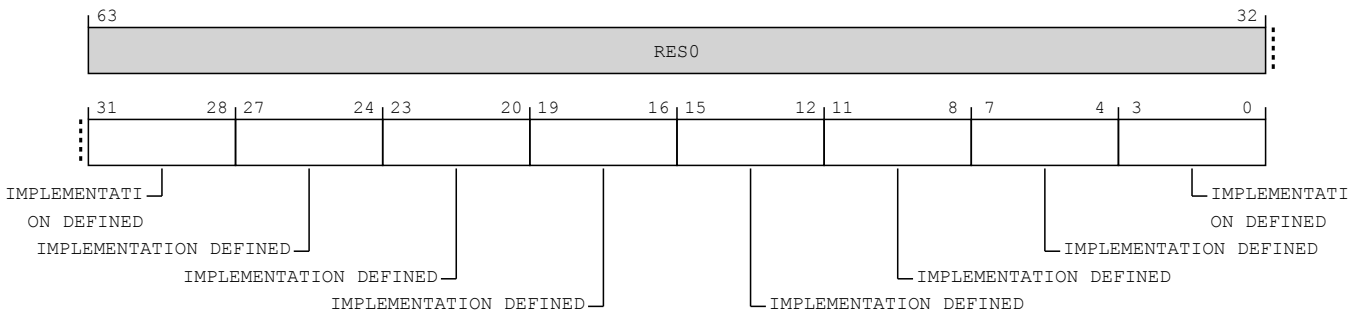

Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:28]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [27:24]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [23:20]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [19:16]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing ID_AA64AFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64AFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64AFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64AFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64AFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.78 ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1

The ID_AA64AFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64AFR1_EL1 are UNDEFINED.

Attributes

ID_AA64AFR1_EL1 is a 64-bit register.

Field descriptions

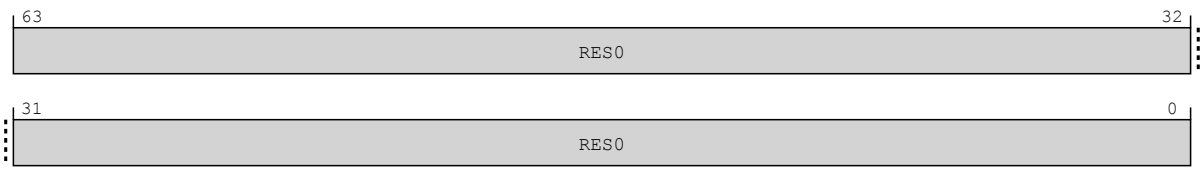

Bits [63:0]

Reserved, RES0.

Accessing ID_AA64AFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64AFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64AFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64AFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64AFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.79 ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0

The ID_AA64DFR0_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

The external register EDDFR gives information from this register.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64DFR0_EL1 are UNDEFINED.

Attributes

ID_AA64DFR0_EL1 is a 64-bit register.

Field descriptions

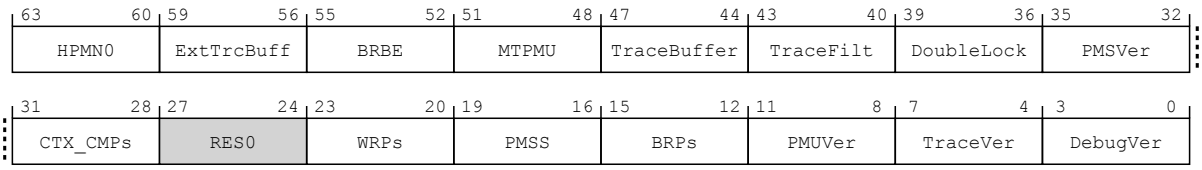

HPMN0, bits [63:60]

Zero PMU event counters for a Guest operating system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HPMN0	Meaning
0b0000	Setting MDCR_EL2.HPMN to zero has CONSTRAINED UNPREDICTABLE behavior.
0b0001	Setting MDCR_EL2.HPMN to zero has defined behavior.

All other values are reserved.

FEAT_HPMN0 implements the functionality identified by the value 0b0001.

From Armv8.8, in an implementation that includes FEAT_PMUv3, FEAT_FGT, and EL2, the value 0b0000 is not permitted.

Access to this field is RO.

ExtTrcBuff, bits [59:56]

Trace Buffer External Mode Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ExtTrcBuff	Meaning
0b0000	Trace Buffer External Mode not implemented.
0b0001	Trace Buffer External Mode implemented.

All other values are reserved.

FEAT_TRBE_EXT implements the functionality identified by the value 0b0001.

Access to this field is RO.

BRBE, bits [55:52]

Branch Record Buffer Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRBE	Meaning
0b0000	Branch Record Buffer Extension not implemented.
0b0001	Branch Record Buffer Extension implemented.
0b0010	As 0b0001, and adds support for branch recording at EL3.

All other values are reserved.

FEAT_BRBE implements the functionality identified by the value 0b0001.

FEAT_BRBEv1p1 implements the functionality identified by the value 0b0010.

From Armv9.3, if FEAT_BRBE is implemented, the value 0b0001 is not permitted.

Access to this field is RO.

MTPMU, bits [51:48]

Multi-threaded PMU extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTPMU	Meaning
0b0000	FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>_EL0.MT and PMEVTYPER<n>.MT are read/write or RES0.
0b0001	FEAT_MTPMU and FEAT_PMUv3 implemented. PMEVTYPER<n>_EL0.MT and PMEVTYPER<n>.MT are read/write. When FEAT_MTPMU is disabled, the Effective values of PMEVTYPER<n>_EL0.MT and PMEVTYPER<n>.MT are 0.
0b1111	FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, PMEVTYPER<n>_EL0.MT and PMEVTYPER<n>.MT are RES0.

All other values are reserved.

FEAT_MTPMU implements the functionality identified by the value 0b0001.

From Armv8.6, in an implementation that includes FEAT_PMUv3, the value 0b0000 is not permitted.

In an implementation that does not include FEAT_PMUv3, the value 0b0001 is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

TraceBuffer, bits [47:44]

Trace Buffer Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceBuffer	Meaning
0b0000	Trace Buffer Extension not implemented.
0b0001	Trace Buffer Extension implemented.
0b0010	As 0b0001, and adds: • If EL2 and FEAT_FGT are implemented, a fine-grained trap on the TSB CSYNC instruction. • If EL2 is implemented, an EL2 control to override TRBLIMITR_EL1.nVM. • The TRBE Profiling exception extension, FEAT_TRBE_EXC.

All other values are reserved.

FEAT_TRBE implements the functionality identified by the value 0b0001.

FEAT_TRBEv1p1 implements the functionality identified by the value 0b0010.

In any Armv9 implementation, if FEAT_ETE is implemented, the value 0b0000 is not permitted.

From Armv9.6, if FEAT_TRBE is implemented, the value 0b0001 is not permitted.

Access to this field is RO.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceFilt	Meaning
0b0000	Armv8.4 Self-hosted Trace Extension not implemented.
0b0001	Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_ETMv4 is implemented, the value 0b0000 is not permitted.

If FEAT_ETE is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

DoubleLock, bits [39:36]

OS Double Lock implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DoubleLock	Meaning
0b0000	OS Double Lock implemented. OSDLR_EL1 is RW.
0b1111	OS Double Lock not implemented. OSDLR_EL1 is RAZ/WI.

All other values are reserved.

FEAT_DoubleLock implements the functionality identified by the value 0b0000.

In Armv8.0, the only permitted value is 0b0000.

If FEAT_Debugv8p2 is implemented and FEAT_DoPD is not implemented, the permitted values are 0b0000 and 0b1111.

If FEAT_DoPD is implemented, the only permitted value is 0b1111.

Access to this field is RO.

PMSVer, bits [35:32]

Statistical Profiling Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMSVer	Meaning
0b0000	Statistical Profiling Extension not implemented.
0b0001	Statistical Profiling Extension implemented.
0b0010	As 0b0001, and adds: • Support for the Events packet Alignment flag. • If FEAT_SVE is implemented, support for the Scalable Vector extensions to Statistical Profiling.
0b0011	As 0b0010, and adds: • Discard mode. • Extended event filtering, including the PMSNEVFR_EL1 System register. • Support for the OPTIONAL previous branch target Address packet. • If FEAT_PMUv3 is implemented, controls to freeze the PMU event counters after an SPE buffer management event occurs. • If FEAT_PMUv3 is implemented, the SAMPLE_FEED_BR, SAMPLE_FEED_EVENT, SAMPLE_FEED_LAT, SAMPLE_FEED_LD, SAMPLE_FEED_OP, and SAMPLE_FEED_ST PMU events.
0b0100	As 0b0011, and adds: • If FEAT_MOPS is implemented, Operation Type packet encodings for Memory Copy and Set operations. • If FEAT_MTE is implemented, Operation Type packet encodings for loads and stores of Allocation Tags.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSVer	Meaning
0b0101	As 0b0100, and adds: • Support for the Events packet Level 2 Data cache access, Level 2 Data cache miss, Cached data modified, Recently fetched cache line, and Cache snoop flags. • Support for Data Source filtering.
0b0110	As 0b0101, and adds: • If EL2 and FEAT_FGT are implemented, a fine-grained trap on the PSB CSYNC instruction. • The SPE Profiling exception extension, FEAT_SPE_EXC. • The Statistical Profiling physical addressing mode extension, FEAT_SPE_nVM.

All other values are reserved.

FEAT_SPE implements the functionality identified by the value 0b0001.

FEAT_SPEv1p1 implements the functionality identified by the value 0b0010.

FEAT_SPEv1p2 implements the functionality identified by the value 0b0011.

FEAT_SPEv1p3 implements the functionality identified by the value 0b0100.

FEAT_SPEv1p4 implements the functionality identified by the value 0b0101.

FEAT_SPEv1p5 implements the functionality identified by the value 0b0110.

From Armv8.5, if FEAT_SPE is implemented, the value 0b0001 is not permitted.

From Armv8.7, if FEAT_SPE is implemented, the value 0b0010 is not permitted.

From Armv8.8, if FEAT_SPE is implemented, the value 0b0011 is not permitted.

From Armv8.9, if FEAT_SPE is implemented, the value 0b0100 is not permitted.

From Armv9.6, if FEAT_SPE is implemented, the value 0b0101 is not permitted.

Access to this field is RO.

CTX_CMPs, bits [31:28]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
0b0000..0b1110	The number of context-aware breakpoints, minus 1.
0b1111	If ID_AA64DFR1_EL1.CTX_CMPs is zero, then 16 context-aware breakpoints are implemented. Otherwise, 16 or more context-aware breakpoints are implemented and ID_AA64DFR1_EL1.CTX_CMPs indicates how many.

Values greater than ID_AA64DFR0_EL1.BRPs are not permitted.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 breakpoints.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
`0b0001..0b1111`	The number of watchpoints, minus 1.

If FEAT_Debugv8p9 is implemented and 16 or more watchpoints are implemented, then this field reads as `0b1111` and ID_AA64DFR1_EL1.WRPs indicates the number of watchpoints.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 watchpoints.

The value `0b0000` is reserved.

Access to this field is RO.

PMSS, bits [19:16]

PMU Snapshot extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMSS	Meaning
`0b0000`	PMU snapshot extension not implemented.
`0b0001`	PMU snapshot extension implemented.

All other values are reserved.

FEAT_PMUv3_SS implements the functionality identified by the value `0b0001`.

Access to this field is RO.

BRPs, bits [15:12]

Index of the highest numbered context-aware breakpoint. Identifies which of the implemented breakpoints are context-aware breakpoints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
`0b0001..0b1110`	Index of the highest numbered context-aware breakpoint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BRPs	Meaning
0b1111	If ID_AA64DFR0_EL1.CTX_CMPs and ID_AA64DFR1_EL1.CTX_CMPs indicate that 16 or fewer breakpoints are context-aware, then the index of the highest context-aware breakpoint is 15. Otherwise, the context-aware breakpoints are the lowest numbered breakpoints.

If ID_AA64DFR1_EL1.BRPs is zero, then this is also the number of implemented breakpoints, minus 1, meaning the context-aware breakpoints are the highest numbered breakpoints.

The value of this field is always less than the number of implemented breakpoints.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 breakpoints.

The value 0b0000 is reserved.

Access to this field is RO.

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMUVer	Meaning
0b0000	Performance Monitors Extension not implemented.
0b0001	Performance Monitors Extension, PMUv3 implemented.
0b0100	PMUv3 for Armv8.1. As 0b0001, and adds support for: • Extended 16-bit PMEVTYPER<n>_EL0.evtCount field. • If EL2 is implemented, the MDCR_EL2.HPMD control.
0b0101	PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR_EL1 register.
0b0110	PMUv3 for Armv8.5. As 0b0101, and adds support for: • 64-bit event counters. • If EL2 is implemented, the MDCR_EL2.HCCD control. • If EL3 is implemented, the MDCR_EL3.SCCD control.
0b0111	PMUv3 for Armv8.7. As 0b0110, and adds support for: • The PMCR_EL0.FZO and, if EL2 is implemented, MDCR_EL2.HPMFZO controls. • If EL3 is implemented, the MDCR_EL3.{MPMX,MCCD} controls.
0b1000	PMUv3 for Armv8.8. As 0b0111, and: • Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF. • Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMUVer	Meaning
0b1001	PMUv3 for Armv8.9. As 0b1000, and: • Updates the definitions of existing PMU events. • Adds support for the PMUSERENR_EL0.UEN control and the PMUACR_EL1 register. • Adds support for the EDECR.PME control.
0b1111	IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0001.

FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.

FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.

FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.

FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.

FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

FEAT_PMUv3p9 implements the functionality identified by the value 0b1001.

From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0001 is not permitted.

From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.

From Armv8.5, if FEAT_PMUv3 is implemented, the value 0b0101 is not permitted.

From Armv8.7, if FEAT_PMUv3 is implemented, the value 0b0110 is not permitted.

From Armv8.8, if FEAT_PMUv3 is implemented, the value 0b0111 is not permitted.

From Armv8.9, if FEAT_PMUv3 is implemented, the value 0b1000 is not permitted.

Access to this field is RO.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace unit is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceVer	Meaning
0b0000	Trace unit System registers not implemented.
0b0001	Trace unit System registers implemented.

All other values are reserved.

When trace unit System registers are implemented, see TRCIDR1 for tracing capabilities of the trace unit.

Access to this field is RO.

DebugVer, bits [3:0]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DebugVer	Meaning
0b0110	Armv8.0 debug architecture.
0b0111	Armv8.1 debug architecture, FEAT_Debugv8p1.
0b1000	Armv8.2 debug architecture, FEAT_Debugv8p2.
0b1001	Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010	Armv8.8 debug architecture, FEAT_Debugv8p8.
0b1011	Armv8.9 debug architecture, FEAT_Debugv8p9.

All other values are reserved.

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

FEAT_Debugv8p8 implements the functionality identified by the value 0b1010.

FEAT_Debugv8p9 implements the functionality identified by the value 0b1011.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

From Armv8.8, the value 0b1001 is not permitted.

From Armv8.9, the value 0b1010 is not permitted.

Access to this field is RO.

Accessing ID_AA64DFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64DFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64DFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64DFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64DFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.80 ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1

The ID_AA64DFR1_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64DFR1_EL1 are UNDEFINED.

Attributes

ID_AA64DFR1_EL1 is a 64-bit register.

Field descriptions

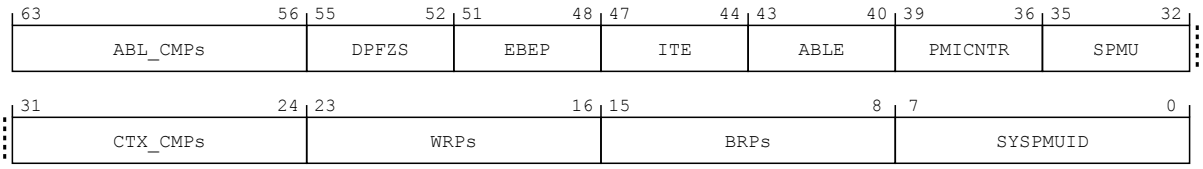

ABL_CMPs, bits [63:56]

When FEAT_ABLE is implemented:

Number of breakpoints that support address linking, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ABL_CMPs	Meaning
0x00..0x3F	Number of breakpoints that support address linking minus 1.

All other values are reserved.

The number of breakpoints that support address linking is never more than either the number of breakpoints or the number of watchpoints.

Access to this field is RO.

Otherwise:

Reserved, RES0.

DPFZS, bits [55:52]

Behavior of the cycle counter when event counting is frozen by a Statistical Profiling management event.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DPFZS	Meaning
0b0000	The cycle counter PMCCNTR_EL0 is never affected by PMCR_EL0.FZS.
0b0001	The cycle counter PMCCNTR_EL0 does not count when PMCR_EL0.DP is 1 and counting by event counters accessible to EL1 is frozen by the PMCR_EL0.FZS mechanism.

FEAT_SPE_DPFZS implements the functionality identified by the value 0b0001.

Access to this field is RO.

EBEP, bits [51:48]

Exception-based event profiling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EBEP	Meaning
0b0000	Exception-based event profiling not implemented.
0b0001	Exception-based event profiling implemented.

All other values are reserved.

FEAT_EBEP implements the functionality identified by the value 0b0001.

Access to this field is RO.

ITE, bits [47:44]

Instrumentation Trace Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ITE	Meaning
0b0000	Instrumentation Trace Extension not implemented.
0b0001	Instrumentation Trace Extension implemented.

All other values are reserved.

FEAT_ITE implements the functionality identified by the value 0b0001.

Access to this field is RO.

ABLE, bits [43:40]

Address Breakpoint Linking Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ABLE	Meaning
0b0000	Address Breakpoint Linking Extension not implemented.
0b0001	Address Breakpoint Linking Extension implemented.

All other values are reserved.

FEAT_BWE implements the address range breakpoints and mismatch breakpoints part of the functionality identified by the value 0b0001.

FEAT_ABLE implements the functionality identified by the value 0b0001.

Access to this field is RO.

PMICNTR, bits [39:36]

PMU fixed-function instruction counter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMICNTR	Meaning
0b0000	PMU fixed-function instruction counter not implemented.
0b0001	PMU fixed-function instruction counter implemented.

All other values are reserved.

FEAT_PMUv3_ICNTR implements the functionality identified by the value 0b0001.

If FEAT_PMUv3 is not implemented, then the only permitted value is 0b0000.

Access to this field is RO.

SPMU, bits [35:32]

System PMU extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SPMU	Meaning
0b0000	System PMU extension not implemented.
0b0001	System PMU extension implemented.
0b0010	As 0b0001, and adds support for SPMZR_EL0.

All other values are reserved.

FEAT_SPMU implements the functionality identified by the value 0b0001.

FEAT_SPMU2 implements the functionality identified by the value 0b0010.

From Armv9.5, the value 0b0001 is not permitted.

Access to this field is RO.

CTX_CMPs, bits [31:24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Context-aware breakpoints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
`0x00`	ID_AA64DFR0_EL1.CTX_CMPs is the number of context-aware breakpoints, minus 1.
`0x01..0x3F`	Number of context-aware breakpoints minus 1.

All other values are reserved.

If 16 or fewer context-aware breakpoints are implemented, then the value of this field is either `0x00` or the same as ID_AA64DFR0_EL1.CTX_CMPs. If more than 16 context-aware breakpoints are implemented, then the value `0x00` is not permitted.

Values greater than ID_AA64DFR1_EL1.BRPs are not permitted.

Access to this field is RO.

WRPs, bits [23:16]

Watchpoints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
`0x00`	ID_AA64DFR0_EL1.WRPs is the number of watchpoints, minus 1.
`0x01..0x3F`	Number of watchpoints minus 1.

All other values are reserved.

Access to this field is RO.

BRPs, bits [15:8]

Breakpoints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
`0x00`	ID_AA64DFR0_EL1.BRPs is the number of breakpoints, minus 1.
`0x01..0x3F`	Number of breakpoints minus 1.

All other values are reserved.

If more than 16 breakpoints are implemented, or the index of the highest numbered context-aware breakpoint is not the number of breakpoints minus 1, then the value `0x00` is not permitted.

Nonzero values less than ID_AA64DFR0_EL1.BRPs are not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SYSPMUID, bits [7:0]

When FEAT_SPMU is implemented:

System PMU ID. Indicates the largest value that can be written to SPMSELR_EL0.SYSPMUSEL.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYSPMUID	Meaning
0x00..0x1F	The largest supported value that can be written to SPMSELR_EL0.SYSPMUSEL.

All other values are reserved.

Since System PMUs might not be contiguously accessible, this field does not necessarily indicate the total number of accessible System PMUs.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing ID_AA64DFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64DFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64DFR1_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64DFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64DFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.81 ID_AA64DFR2_EL1, AArch64 Debug Feature Register 2

The ID_AA64DFR2_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64DFR2_EL1 are UNDEFINED.

Attributes

ID_AA64DFR2_EL1 is a 64-bit register.

Field descriptions

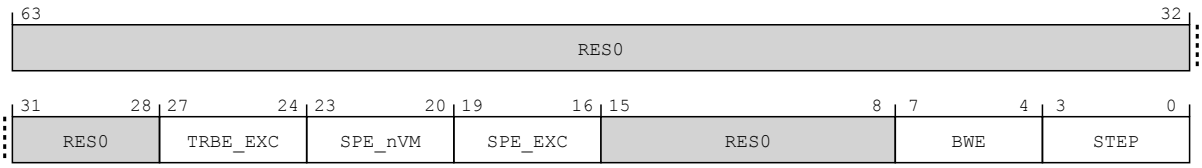

Bits [63:28]

Reserved, RES0.

TRBE_EXC, bits [27:24]

TRBE Profiling exception extension. Describes support for reporting trace buffer management events as TRBE Profiling exceptions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRBE_EXC	Meaning
0b0000	TRBE Profiling exception not implemented.
0b0001	TRBE Profiling exception implemented.

All other values are reserved.

FEAT_TRBE_EXC implements the functionality identified by the value 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SPE_nVM, bits [23:20]

Profiling Buffer physical address mode supported. Describes support for defining the Profiling Buffer using physical addresses or intermediate physical addresses.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SPE_nVM	Meaning
0b0000	Profiling Buffer physical address mode not implemented.
0b0001	Profiling Buffer physical address mode implemented.

All other values are reserved.

FEAT_SPE_nVM implements the functionality identified by the value 0b0001.

Access to this field is RO.

SPE_EXC, bits [19:16]

SPE Profiling exception extension. Describes support for reporting Profiling Buffer management events as SPE Profiling exceptions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SPE_EXC	Meaning
0b0000	SPE Profiling exception not implemented.
0b0001	SPE Profiling exception implemented.

All other values are reserved.

FEAT_SPE_EXC implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [15:8]

Reserved, RES0.

BWE, bits [7:4]

Breakpoints and watchpoint enhancements.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BWE	Meaning
0b0000	This field does not indicate whether DBGBCR<n>_EL1.MASK and address mismatch breakpoints are implemented.
0b0001	DBGBCR<n>_EL1.MASK and address mismatch breakpoints are implemented.
0b0010	As 0b0001, and address mismatch watchpoints are implemented.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_BWE implements the functionality identified by the value `0b0001`.

FEAT_BWE2 implements the functionality identified by the value `0b0010`.

When this field is `0b0000`, ID_AA64DFR1_EL1.ABLE might indicate the presence of support for DBGBCR<n>_EL1.MASK and address mismatch breakpoints.

From Armv9.5, the value `0b0001` is not permitted.

Access to this field is RO.

STEP, bits [3:0]

Enhanced Software Step Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

STEP	Meaning
`0b0000`	Execution from MDSTEPOP_EL1 is not supported for Software Step.
`0b0001`	Execution from MDSTEPOP_EL1 is supported for Software Step.

All other values are reserved.

FEAT_STEP2 implements the functionality identified by the value `0b0001`.

Access to this field is RO.

Accessing ID_AA64DFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64DFR2_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b0000`	`0b0101`	`0b010`

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64DFR2_EL1()) ||
     ↪ImpDefBool("ID_AA64DFR2_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64DFR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64DFR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64DFR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.82 ID_AA64FPFR0_EL1, AArch64 Floating-point Feature Register 0

The ID_AA64FPFR0_EL1 characteristics are:

Purpose

Provides information about floating-point formats and instructions implemented in AArch64 state.

The fields in this register do not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64SMFR0_EL1 and ID_AA64FPFR0_EL1.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64FPFR0_EL1 are UNDEFINED.

Attributes

ID_AA64FPFR0_EL1 is a 64-bit register.

Field descriptions

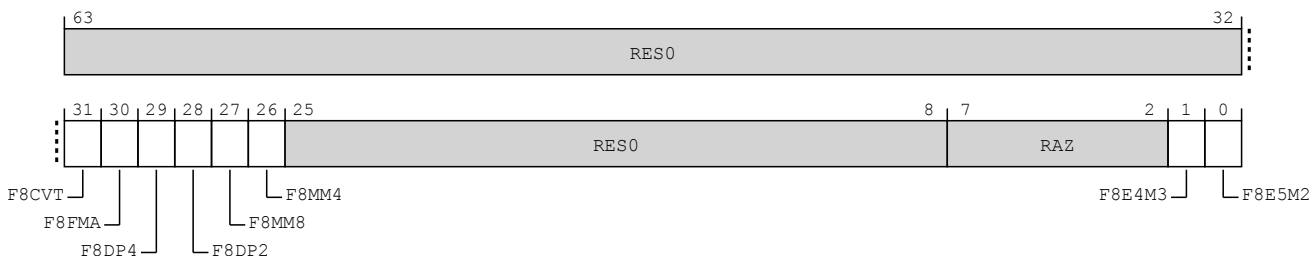

Bits [63:32]

Reserved, RES0.

F8CVT, bit [31]

Indicates support for the following instructions:

- The Advanced SIMD floating-point scaling instruction `FSCALE`.
- The Advanced SIMD FP8 convert instructions `BF1CVTL`, `BF1CVTL2`, `BF2CVTL`, `BF2CVTL2`, `F1CVTL`, `F1CVTL2`, `F2CVTL`, `F2CVTL2`, `FCVTN`, and `FCVTN2`.
- When FEAT_SVE2 or FEAT_SME2 is implemented, the SVE2 FP8 convert instructions `BF1CVT`, `BF1CVTLT`, `BF2CVT`, `BF2CVTLT`, `F1CVT`, `F1CVTLT`, `F2CVT`, `F2CVTLT`, `BFCVTN`, `FCVTN`, `FCVTNB`, and `FCVTNT`.
- When FEAT_SME2 is implemented, the SME2 multi-vector floating-point scaling instruction `FSCALE` and the SME2 FP8 convert instructions `BF1CVT`, `BF1CVTL`, `BF2CVT`, `BF2CVTL`, `F1CVT`, `F1CVTL`, `F2CVT`, `F2CVTL`, `BFCVT`, `FCVT`, and `FCVTN`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F8CVT	Meaning
0b0	The specified instructions are not implemented.
0b1	The specified instructions are implemented.

FEAT_FP8 implements the functionality identified by the value 1.

Access to this field is RO.

F8FMA, bit [30]

Indicates support for the following instructions:

- The Advanced SIMD FP8 to single-precision and half-precision multiply-accumulate instructions FMLALB, FMLALT, FMLALLBB, FMLALLBT, FMLALLTB, and FMLALLTT.
- When FEAT_SVE2 is implemented and the PE is not in Streaming SVE mode, the SVE2 FP8 to single-precision and half-precision multiply-accumulate instructions FMLALB, FMLALT, FMLALLBB, FMLALLBT, FMLALLTB, and FMLALLTT.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8FMA	Meaning
0b0	The specified instructions are not implemented.
0b1	The specified instructions are implemented.

FEAT_FP8FMA implements the functionality identified by the value 1.

Access to this field is RO.

F8DP4, bit [29]

Indicates support for the following instructions:

- Advanced SIMD FP8 to single-precision 4-way dot product FDOT (4-way) instructions.
- When FEAT_SVE2 is implemented and the PE is not in Streaming SVE mode, SVE FP8 to single-precision 4-way dot product FDOT (4-way) instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8DP4	Meaning
0b0	The specified instructions are not implemented by this feature.
0b1	The specified instructions are implemented.

Note

Other features may implement some of the specified instructions.

All other values are reserved.

FEAT_FP8DOT4 implements the functionality identified by the value 1.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F8DP2, bit [28]

Indicates support for the following instructions:

- Advanced SIMD FP8 to half-precision 2-way dot product `FDOT` (2-way) instructions.
- When FEAT_SVE2 is implemented and the PE is not in Streaming SVE mode, SVE FP8 to half-precision 2-way dot product `FDOT` (2-way) instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8DP2	Meaning
`0b0`	The specified instructions are not implemented by this feature.
`0b1`	The specified instructions are implemented.

Note

Other features may implement some of the specified instructions.

FEAT_FP8DOT2 implements the functionality identified by the value 1.

Access to this field is RO.

F8MM8, bit [27]

Indicates support for the following instructions:

- Advanced SIMD FP8 to single-precision matrix multiply `FMMLA` (8-way, FP8 to FP32) instruction.
- If FEAT_SVE2 is implemented, SVE FP8 to single-precision matrix multiply `FMMLA` (widening, FP8 to FP32) instruction is implemented when the PE is not in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8MM8	Meaning
`0b0`	Advanced SIMD and SVE FP8 to single-precision matrix multiply instructions are not implemented by this feature.
`0b1`	The specified Advanced SIMD and SVE FP8 to single-precision matrix multiply instructions are implemented.

FEAT_F8F32MM implements the functionality identified by the value `0b0001`

Access to this field is RO.

F8MM4, bit [26]

Indicates support for the following instructions:

- Advanced SIMD FP8 to half-precision matrix multiply `FMMLA` (4-way, FP8 to FP16) instruction.
- If FEAT_SVE2 is implemented, SVE FP8 to half-precision matrix multiply `FMMLA` (widening, FP8 to FP16) instruction is implemented when the PE is not in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F8MM4	Meaning
0b0	Advanced SIMD and SVE FP8 to half-precision matrix multiply instructions are not implemented by this feature.
0b1	The specified Advanced SIMD and SVE FP8 to half-precision matrix multiply instructions are implemented.

FEAT_F8F16MM implements the functionality identified by the value 0b0001

Access to this field is RO.

Bits [25:8]

Reserved, RES0.

Bits [7:2]

Reserved for data formats 2 to 7.

Reserved, RAZ.

F8E4M3, bit [1]

Indicates support for OFP8 E4M3 format and behavior for FP8 instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8E4M3	Meaning
0b0	OFP8 E4M3 format is not supported.
0b1	OFP8 E4M3 format is supported.

If FEAT_FP8 is implemented, the only permitted value is 1.

Otherwise, the only permitted value is 0.

For more information on OFP8 formats, see the Open Compute Project, OCP 8-bit Floating Point Specification (OFP8).

Access to this field is RO.

F8E5M2, bit [0]

Indicates support for OFP8 E5M2 format and behavior for FP8 instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8E5M2	Meaning
0b0	OFP8 E5M2 format is not supported.
0b1	OFP8 E5M2 format is supported.

If FEAT_FP8 is implemented, the only permitted value is 1.

Otherwise, the only permitted value is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information on OFP8 formats, see the Open Compute Project, OCP 8-bit Floating Point Specification (OFP8).

Access to this field is RO.

Accessing ID_AA64FPFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64FPFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64FPFR0_EL1()) ||
     ↪ImpDefBool("ID_AA64FPFR0_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64FPFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64FPFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64FPFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.83 ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0

The ID_AA64ISAR0_EL1 characteristics are:

Purpose

Provides information about the instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64ISAR0_EL1 are UNDEFINED.

Attributes

ID_AA64ISAR0_EL1 is a 64-bit register.

Field descriptions

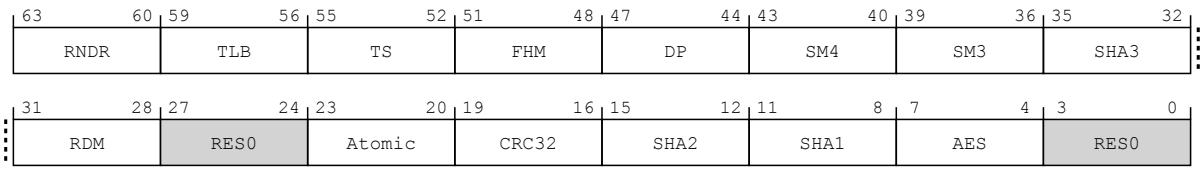

RNDR, bits [63:60]

Indicates support for Random Number instructions in AArch64 state.

When FEAT_RNG_TRAP is implemented, the value returned by a direct read of ID_AA64ISAR0_EL1.RNDR is further controlled by the value of SCR_EL3.TRNDR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RNDR	Meaning
0b0000	No Random Number instructions are implemented.
0b0001	RNDR and RNDRRS registers are implemented.

All other values are reserved.

FEAT_RNG implements the functionality identified by the value 0b0001.

Access to this field is RO.

TLB, bits [59:56]

Indicates support for Outer Shareable and TLB range maintenance instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLB	Meaning
0b0000	Outer Shareable and TLB range maintenance instructions are not implemented.
0b0001	Outer Shareable TLB maintenance instructions are implemented.
0b0010	Outer Shareable and TLB range maintenance instructions are implemented.

All other values are reserved.

FEAT_TLBIOS implements the functionality identified by the values 0b0001 and 0b0010.

FEAT_TLBIRANGE implements the functionality identified by the value 0b0010.

From Armv8.4, the values 0b0000 and 0b0001 are not permitted.

Access to this field is RO.

TS, bits [55:52]

Indicates support for flag manipulation instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TS	Meaning
0b0000	No flag manipulation instructions are implemented.
0b0001	CFINV, RMIF, SETF16, and SETF8 instructions are implemented.
0b0010	CFINV, RMIF, SETF16, SETF8, AXFLAG, and XAFLAG instructions are implemented.

All other values are reserved.

FEAT_FlagM implements the functionality identified by the value 0b0001.

FEAT_FlagM2 implements the functionality identified by the value 0b0010.

In Armv8.4, the value 0b0000 is not permitted.

From Armv8.5, the value 0b0001 is not permitted.

Access to this field is RO.

FHM, bits [51:48]

Indicates support for the Advanced SIMD half-precision to single-precision multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FHM	Meaning
0b0000	The Advanced SIMD half-precision to single-precision multiply instructions are not implemented.
0b0001	FMLAL and FMLSL instructions are implemented.

All other values are reserved.

FEAT_FHM implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_FP16 is implemented, the value 0b0000 is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

DP, bits [47:44]

Indicates support for Dot Product instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DP	Meaning
0b0000	No Dot Product instructions implemented.
0b0001	UDOT and SDOT instructions implemented.

All other values are reserved.

FEAT_DotProd implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_AdvSIMD is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

SM4, bits [43:40]

Indicates support for SM4 instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SM4	Meaning
0b0000	No SM4 instructions implemented.
0b0001	SM4E and SM4EKEY instructions implemented.

All other values are reserved.

If FEAT_SM4 is not implemented, the value 0b0001 is reserved.

FEAT_SM4 implements the functionality identified by the value 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM3.

Access to this field is RO.

SM3, bits [39:36]

Indicates support for the following SM3 instructions SM3SS1, SM3TT1A, SM3TT1B, SM3TT2A, SM3TT2B, SM3PARTW1, and SM3PARTW2 in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SM3	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SM3 is not implemented, the value `0b0001` is reserved.

FEAT_SM3 implements the functionality identified by the value `0b0001`.

This field must have the same value as ID_AA64ISAR0_EL1.SM4.

Access to this field is RO.

SHA3, bits [35:32]

Indicates support for the following SHA3 instructions `EOR3`, `RAX1`, `XAR`, and `BCAX` in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA3	Meaning
`0b0000`	The specified instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

If FEAT_SHA3 is not implemented, the value `0b0001` is reserved.

FEAT_SHA3 implements the functionality identified by the value `0b0001`.

If the value of ID_AA64ISAR0_EL1.SHA1 is `0b0000`, this field must have the value `0b0000`.

If the value of this field is `0b0001`, ID_AA64ISAR0_EL1.SHA2 must have the value `0b0010`.

Access to this field is RO.

RDM, bits [31:28]

Indicates support for `SQRDMLAH` and `SQRDMLSH` instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RDM	Meaning
`0b0000`	No `RDMA` instructions implemented.
`0b0001`	`SQRDMLAH` and `SQRDMLSH` instructions implemented.

All other values are reserved.

FEAT_RDM implements the functionality identified by the value `0b0001`.

From Armv8.1, the value `0b0000` is not permitted.

Access to this field is RO.

Bits [27:24]

Reserved, RES0.

Atomic, bits [23:20]

Indicates support for Atomic instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Atomic	Meaning
0b0000	No Atomic instructions implemented.
0b0010	LDADD, LDCLR, LDEOR, LDSET, LDSMAX, LDSMIN, LDUMAX, LDUMIN, CAS, CASP, and SWP instructions implemented.
0b0011	As for 0b0010, plus 128-bit instructions LDCLRP, LDSETP, and SWPP.

All other values are reserved.

FEAT_LSE implements the functionality identified by the value 0b0010.

FEAT_LSE128 implements the functionality identified by the value 0b0011.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

CRC32, bits [19:16]

Indicates support for the following CRC32 instructions CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CRC32	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_CRC32 implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

SHA2, bits [15:12]

Indicates support for SHA2 instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA2	Meaning
0b0000	No SHA2 instructions implemented.
0b0001	Implements instructions: SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
0b0010	Implements instructions: • SHA256H, SHA256H2, SHA256SU0, and SHA256SU1. • SHA512H, SHA512H2, SHA512SU0, and SHA512SU1.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SHA256 implements the functionality identified by the value `0b0001`.

FEAT_SHA512 implements the functionality identified by the value `0b0010`.

If the value of ID_AA64ISAR0_EL1.SHA1 is `0b0000`, this field must have the value `0b0000`.

If the value of this field is `0b0010`, ID_AA64ISAR0_EL1.SHA3 must have the value `0b0001`.

Access to this field is RO.

SHA1, bits [11:8]

Indicates support for the following SHA1 instructions `SHA1C`, `SHA1P`, `SHA1M`, `SHA1H`, `SHA1SU0`, and `SHA1SU1` in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA1	Meaning
`0b0000`	The specified instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_SHA1 implements the functionality identified by the value `0b0001`.

If the value of ID_AA64ISAR0_EL1.SHA2 is `0b0000`, this field must have the value `0b0000`.

Access to this field is RO.

AES, bits [7:4]

Indicates support for AES instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AES	Meaning
`0b0000`	No AES instructions implemented.
`0b0001`	`AESE`, `AESD`, `AESMC`, and `AESIMC` instructions implemented.
`0b0010`	As for `0b0001`, plus `PMULL` and `PMULL2` instructions operating on 64-bit source elements.

FEAT_AES implements the functionality identified by the value `0b0001`.

FEAT_PMULL implements the functionality identified by the value `0b0010`.

All other values are reserved.

Access to this field is RO.

Bits [3:0]

Reserved, RES0.

Accessing ID_AA64ISAR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ID_AA64ISAR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64ISAR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.84 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64ISAR1_EL1 are UNDEFINED.

Attributes

ID_AA64ISAR1_EL1 is a 64-bit register.

Field descriptions

63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
LS64	XS	I8MM	DGH	BF16	SPECRES	SB	FRINTTS

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
GPI	GPA	LRCPC	FCMA	JSCVT	API	APA	DPB

LS64, bits [63:60]

Indicates support for `LD64B` and `ST64B*` instructions, and the ACCDATA_EL1 register.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LS64	Meaning
`0b0000`	No `LD64B` or `ST64B` instructions are supported.
`0b0001`	The `LD64B` and `ST64B` instructions are supported.
`0b0010`	As `0b0001`, and adds the `ST64BV` instruction and traps.
`0b0011`	As `0b0010`, and adds the `ST64BV0` instruction, ACCDATA_EL1 register, and traps.
`0b0100`	As `0b0011`, and adds support for atomic accesses to Write-back Cacheable, Shareable memory using one of the following: • `LD64B` and `ST64B` instructions. • SIMD&FP instructions that load or store a pair of 128-bit registers by generating 32-byte single-copy atomic accesses.

All other values are reserved.

FEAT_LS64 implements the functionality identified by `0b0001`.

FEAT_LS64_V implements the functionality identified by `0b0010`.

FEAT_LS64_ACCDATA implements the functionality identified by `0b0011`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LS64WB implements the functionality identified by `0b0100`.

Access to this field is RO.

XS, bits [59:56]

Indicates support for the XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

XS	Meaning
`0b0000`	The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are not supported.
`0b0001`	The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are supported.

All other values are reserved.

FEAT_XS implements the functionality identified by `0b0001`.

From Armv8.7, the value `0b0000` is not permitted.

Access to this field is RO.

I8MM, bits [55:52]

Indicates support for the following Advanced SIMD Int8 matrix multiplication instructions `SMMLA`, `SUDOT`, `UMMLA`, `USMMLA`, and `USDOT` in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

I8MM	Meaning
`0b0000`	The specified instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_I8MM implements the functionality identified by `0b0001`.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ZFR0_EL1.I8MM.

From Armv8.6, the value `0b0000` is not permitted.

Access to this field is RO.

DGH, bits [51:48]

Indicates support for the Data Gathering Hint instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DGH	Meaning
0b0000	Data Gathering Hint is not implemented.
0b0001	Data Gathering Hint is implemented.

All other values are reserved.

FEAT_DGH implements the functionality identified by 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.

If the DGH instruction has no effect in preventing the merging of memory accesses, the value of this field is 0b0000.

Access to this field is RO.

BF16, bits [47:44]

Indicates support for Advanced SIMD and floating-point BFloat16 instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BF16	Meaning
0b0000	BFloat16 instructions are not implemented.
0b0001	BFCVT, BFCVTN, BFCVTN2, BFDOT, BFMLALB, BFMLALT, and BFMMLA instructions are implemented.
0b0010	As 0b0001, but the FPCR.EBF field is also supported.

All other values are reserved.

FEAT_BF16 implements the functionality identified by 0b0001.

FEAT_EBF16 implements the functionality identified by 0b0010.

When FEAT_SVE or FEAT_SME is implemented, this field must return the same value as ID_AA64ZFR0_EL1.BF16.

From Armv8.6 and Armv9.1, the value 0b0000 is not permitted.

Access to this field is RO.

SPECRES, bits [43:40]

Indicates support for prediction invalidation instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SPECRES	Meaning
0b0000	Prediction invalidation instructions are not implemented.
0b0001	CFP RCTX, DVP RCTX and CPP RCTX instructions are implemented.
0b0010	As 0b0001, and COSP RCTX instruction is implemented.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_SPECRES implements the functionality identified by `0b0001`.

FEAT_SPECRES2 implements the functionality identified by `0b0010`.

From Armv8.5, the value `0b0000` is not permitted.

From Armv8.9, the value `0b0001` is not permitted.

Access to this field is RO.

SB, bits [39:36]

Indicates support for `SB` instruction in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SB	Meaning
`0b0000`	`SB` instruction is not implemented.
`0b0001`	`SB` instruction is implemented.

All other values are reserved.

FEAT_SB implements the functionality identified by `0b0001`.

From Armv8.5, the value `0b0000` is not permitted.

Access to this field is RO.

FRINTTS, bits [35:32]

Indicates support for `FRINT32Z`, `FRINT32X`, `FRINT64Z`, and `FRINT64X` instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FRINTTS	Meaning
`0b0000`	The specified instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_FRINTTS implements the functionality identified by `0b0001`.

From Armv8.5, the value `0b0000` is not permitted.

Access to this field is RO.

GPI, bits [31:28]

Indicates support for an IMPLEMENTATION DEFINED algorithm is implemented in the PE for generic code authentication in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPI	Meaning
0b0000	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
0b0001	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

When this field is nonzero, FEAT_PACIMP is implemented.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPA is nonzero, or the value of ID_AA64ISAR2_EL1.GPA3 is nonzero, this field must have the value 0b0000.

Access to this field is RO.

GPA, bits [27:24]

Indicates whether the QARMA5 algorithm is implemented in the PE for generic code authentication in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GPA	Meaning
0b0000	Generic Authentication using the QARMA5 algorithm is not implemented.
0b0001	Generic Authentication using the QARMA5 algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACQARMA5 implements the functionality identified by 0b0001.

When this field is nonzero, FEAT_PACQARMA5 is implemented.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is nonzero, or the value of ID_AA64ISAR2_EL1.GPA3 is nonzero, this field must have the value 0b0000.

Access to this field is RO.

LRCPC, bits [23:20]

Indicates support for weaker release consistency, RCpc, based model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LRCPC	Meaning
0b0000	RCpc instructions are not implemented.
0b0001	The no offset LDAPR, LDAPRB, and LDAPRH instructions are implemented.
0b0010	As 0b0001, and the LDAPUR (unscaled), LDAPURB (unscaled), LDAPURSB (unscaled), LDAPURH (unscaled), LDAPURSH (unscaled), LDAPURSW (unscaled), STLUR (unscaled), STLURB (unscaled) and STLURH (unscaled) instructions are implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LRCPC	Meaning
0b0011	As 0b0010, and the post-index LDAPR, LDIAPP, STILP, and pre-index STLR instructions are implemented. If Advanced SIMD and floating-point is implemented, then the LDAPUR (SIMD&FP), LDAP1 (SIMD&FP), STLUR (SIMD&FP), and STL1 (SIMD&FP) instructions are implemented in Advanced SIMD and floating-point.

All other values are reserved.

FEAT_LRCPC implements the functionality identified by the value 0b0001.

FEAT_LRCPC2 implements the functionality identified by the value 0b0010.

FEAT_LRCPC3 implements the functionality identified by the value 0b0011.

From Armv8.3, the value 0b0000 is not permitted.

From Armv8.4, the value 0b0001 is not permitted.

Access to this field is RO.

FCMA, bits [19:16]

Indicates support for complex number addition and multiplication, where numbers are stored in vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FCMA	Meaning
0b0000	The FCMLA and FCADD instructions are not implemented.
0b0001	The FCMLA and FCADD instructions are implemented.

All other values are reserved.

FEAT_FCMA implements the functionality identified by the value 0b0001.

From Armv8.3, if Advanced SIMD or floating-point is implemented, the value 0b0000 is not permitted.

From Armv8.3, if Advanced SIMD or floating-point is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

JSCVT, bits [15:12]

Indicates support for JavaScript conversion from double-precision floating-point values to integers in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

JSCVT	Meaning
0b0000	The FJCVTZS instruction is not implemented.
0b0001	The FJCVTZS instruction is implemented.

All other values are reserved.

FEAT_JSCVT implements the functionality identified by 0b0001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv8.3, if Advanced SIMD or floating-point is implemented, the value `0b0000` is not permitted.

From Armv8.3, if Advanced SIMD or floating-point is not implemented, the only permitted value is `0b0000`.

Access to this field is RO.

API, bits [11:8]

Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the `PACGA` instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

API	Meaning
`0b0000`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
`0b0001`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, FEAT_EPAC and FEAT_PAuth2 are not implemented.
`0b0010`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, FEAT_EPAC is implemented, and FEAT_PAuth2 is not implemented.
`0b0011`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, FEAT_EPAC is not implemented, and FEAT_PAuth2 is implemented.
`0b0100`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, FEAT_EPAC is not implemented, FEAT_PAuth2 and FEAT_FPAC are implemented.
`0b0101`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, FEAT_EPAC is not implemented, FEAT_PAuth2, FEAT_FPAC, and FEAT_FPACCOMBINE are implemented.
`0b0110`	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, including instructions that allow signing of LR using SP and PC as diversifiers, FEAT_EPAC is not implemented, FEAT_PAuth2, FEAT_FPAC, FEAT_FPACCOMBINE, and FEAT_PAuth_LR are implemented.

All other values are reserved.

FEAT_PAuth implements the functionality identified by `0b0001`.

FEAT_EPAC implements the functionality identified by `0b0010`.

FEAT_PAuth2 implements the functionality identified by `0b0011`.

FEAT_FPAC implements the functionality identified by `0b0100`.

FEAT_FPACCOMBINE implements the functionality identified by `0b0101`.

FEAT_PAuth_LR implements the functionality identified by `0b0110`.

When this field is nonzero, FEAT_PACIMP is implemented.

In Armv8.3, the permitted values are `0b0000`, `0b0001`, `0b0010`, `0b0011`, `0b0100`, and `0b0101`.

From Armv8.6, the permitted values are `0b0000`, `0b0011`, `0b0100`, and `0b0101`.

From Armv9.5, the permitted values are `0b0000`, `0b0011`, `0b0100`, `0b0101`, and `0b0110`.

If the value of ID_AA64ISAR1_EL1.APA is nonzero, or the value of ID_AA64ISAR2_EL1.APA3 is nonzero, this field must have the value `0b0000`.

Access to this field is RO.

APA, bits [7:4]

Indicates whether the QARMA5 algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the `PACGA` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

APA	Meaning
0b0000	Address Authentication using the QARMA5 algorithm is not implemented.
0b0001	Address Authentication using the QARMA5 algorithm is implemented, FEAT_EPAC and FEAT_PAuth2 are not implemented.
0b0010	Address Authentication using the QARMA5 algorithm is implemented, FEAT_EPAC is implemented, and FEAT_PAuth2 is not implemented.
0b0011	Address Authentication using the QARMA5 algorithm is implemented, FEAT_EPAC is not implemented, and FEAT_PAuth2 is implemented.
0b0100	Address Authentication using the QARMA5 algorithm is implemented, FEAT_EPAC is not implemented, FEAT_PAuth2 and FEAT_FPAC are implemented.
0b0101	Address Authentication using the QARMA5 algorithm is implemented, FEAT_EPAC is not implemented, FEAT_PAuth2, FEAT_FPAC, and FEAT_FPACCOMBINE are implemented.
0b0110	Address Authentication using the QARMA5 algorithm is implemented, including instructions that allow signing of LR using SP and PC as diversifiers, FEAT_EPAC is not implemented, FEAT_PAuth2, FEAT_FPAC, FEAT_FPACCOMBINE, and FEAT_PAuth_LR are implemented.

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.

FEAT_EPAC implements the functionality identified by 0b0010.

FEAT_PAuth2 implements the functionality identified by 0b0011.

FEAT_FPAC implements the functionality identified by 0b0100.

FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

FEAT_PAuth_LR implements the functionality identified by 0b0110.

When this field is nonzero, FEAT_PACQARMA5 is implemented.

In Armv8.3, the permitted values are 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From Armv8.6, the permitted values are 0b0000, 0b0011, 0b0100, and 0b0101.

From Armv9.5, the permitted values are 0b0000, 0b0011, 0b0100, 0b0101, and 0b0110.

If the value of ID_AA64ISAR1_EL1.API is nonzero, or the value of ID_AA64ISAR2_EL1.APA3 is nonzero, this field must have the value 0b0000.

Access to this field is RO.

DPB, bits [3:0]

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DPB	Meaning
0b0000	DC CVAP not supported.
0b0001	DC CVAP supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DPB	Meaning
0b0010	DC CVAP and DC CVADP supported.

All other values are reserved.

FEAT_DPB implements the functionality identified by the value 0b0001.

FEAT_DPB2 implements the functionality identified by the value 0b0010.

From Armv8.2, the value 0b0000 is not permitted.

From Armv8.5, the value 0b0001 is not permitted.

Access to this field is RO.

Accessing ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64ISAR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64ISAR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.85 ID_AA64ISAR2_EL1, AArch64 Instruction Set Attribute Register 2

The ID_AA64ISAR2_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64ISAR2_EL1 are UNDEFINED.

Attributes

ID_AA64ISAR2_EL1 is a 64-bit register.

Field descriptions

63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
ATS1A	LUT	CSSC	RPRFM	PCDPHINT	PRFMSLC	SYSINSTR_128	SYSREG_128

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
CLRBHB	PAC_frac	BC	MOPS	APA3	GPA3	RPRES	WFxT

ATS1A, bits [63:60]

Indicates support for address translation instructions, which perform stage 1 address translation for the given virtual address without checking for stage 1 permissions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ATS1A	Meaning
0b0000	Address Translate Stage 1 instructions without Permissions Checks are not implemented
0b0001	Address Translate Stage 1 instructions without Permissions Checks are implemented.

All other values are reserved.

Access to this field is RO.

LUT, bits [59:56]

Indicates support for:

- Advanced SIMD lookup table instructions with 2-bit and 4-bit indices.
- If FEAT_SVE2 or FEAT_SME2 is implemented, SVE lookup table instructions with 2-bit and 4-bit indices.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

LUT	Meaning
0b0000	The specified instructions are not implemented.
0b0001	Lookup table instructions with 2-bit indices LUTI2 and 4-bit indices LUTI4 are implemented.

All other values are reserved.

FEAT_LUT implements the functionality identified by the value 0b0001.

From Armv9.5, if FEAT_AdvSIMD is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

CSSC, bits [55:52]

Indicates support for common short sequence compression instructions.

If FEAT_CMPBR is implemented, indicates support for compare and branch instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSSC	Meaning
0b0000	Common short sequence compression instructions are not implemented.
0b0001	The following common short sequence compression instructions are implemented: • 32-bit and 64-bit ABS, CNT, CTZ • 32-bit SMAX, UMAX, SMIN, UMIN (immediate). • 64-bit SMAX, UMAX, SMIN, UMIN (immediate). • 32-bit SMAX, UMAX, SMIN, UMIN (register). • 64-bit SMAX, UMAX, SMIN, UMIN (register).
0b0010	As 0b0001, and adds compare and branch instructions: • CBB<cc>. • CB<cc> (immediate), CB<cc> (register). • CBH<cc>.

All other values are reserved.

FEAT_CSSC implements the functionality identified by the value 0b0001.

FEAT_CMPBR implements the functionality identified by the value 0b0010.

From Armv9.4, the value 0b0000 is not permitted.

From Armv9.6, the value 0b0001 is not permitted.

Access to this field is RO.

RPRFM, bits [51:48]

RPRFM hint instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

RPRFM	Meaning
0b0000	RPRFM hint instruction is not implemented and is treated as a NOP.
0b0001	RPRFM hint instruction is implemented.

All other values are reserved.

FEAT_RPRFM implements the functionality identified by the value 0b0001.

Access to this field is RO.

PCDPHINT, bits [47:44]

Indicates support for producer-consumer data placement hints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCDPHINT	Meaning
0b0000	The STSHH and PRFM IR hint instructions are not implemented.
0b0001	The STSHH and PRFM IR hint instructions are implemented.

FEAT_PCDPHINT implements the functionality identified by the value 0b0001.

Access to this field is RO.

PRFMSLC, bits [43:40]

Indicates whether the PRFM and PRFUM instructions support a system level cache option.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PRFMSLC	Meaning
0b0000	The PRFM and PRFUM instructions do not support the SLC target.
0b0001	The PRFM and PRFUM instructions support the SLC target.

All other values are reserved.

FEAT_PRFMSLC implements the functionality identified by the value 0b0001.

Access to this field is RO.

SYSINSTR_128, bits [39:36]

SYSINSTR_128. Indicates support for System instructions that can take 128-bit inputs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SYSINSTR_128	Meaning
0b0000	System instructions that can take 128-bit inputs are not supported.
0b0001	System instructions that can take 128-bit inputs are supported.

All other values are reserved.

FEAT_SYSINSTR128 implements the functionality identified by the value 0b0001.

Access to this field is RO.

SYSREG_128, bits [35:32]

SYSREG_128. Indicates support for instructions to access 128-bit System Registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYSREG_128	Meaning
0b0000	Instructions to access 128-bit System Registers are not supported.
0b0001	Instructions to access 128-bit System Registers are supported.

All other values are reserved.

FEAT_SYSREG128 implements the functionality identified by the value 0b0001.

Access to this field is RO.

CLRBHB, bits [31:28]

Indicates support for the CLRBHB instruction in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CLRBHB	Meaning
0b0000	CLRBHB instruction is not implemented.
0b0001	CLRBHB instruction is implemented.

All other values are reserved.

FEAT_CLRBHB implements the functionality identified by the value 0b0001.

From Armv8.9, the value 0b0000 is not permitted.

Access to this field is RO.

PAC_frac, bits [27:24]

Indicates which address bit is used to determine the size of the PAC field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAC_frac	Meaning
0b0000	The address bit which is used to define the size of the PAC field is dependent on whether address tagging is used.
0b0001	The address bit which is used to define the size of the PAC field is fixed.

All other values are reserved.

FEAT_CONSTPACFIELD implements the functionality identified by the value 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

BC, bits [23:20]

Indicates support for the BC instruction in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BC	Meaning
0b0000	BC instruction is not implemented.
0b0001	BC instruction is implemented.

All other values are reserved.

FEAT_HBC implements the functionality identified by the value 0b0001.

From Armv8.8, the value 0b0000 is not permitted.

Access to this field is RO.

MOPS, bits [19:16]

Indicates support for the Memory Copy and Memory Set instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MOPS	Meaning
0b0000	The Memory Copy and Memory Set instructions are not implemented in AArch64 state.
0b0001	The Memory Copy and Memory Set instructions are implemented in AArch64 state with the following exception. If FEAT_MTE is implemented, then SETGP*, SETGM* and SETGE* instructions are also supported.

All other values are reserved.

FEAT_MOPS implements the functionality identified by the value 0b0001.

From Armv8.8, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

APA3, bits [15:12]

Indicates whether the QARMA3 algorithm is implemented in the PE for address authentication in AArch64 state. This applies to all Pointer Authentication instructions other than the `PACGA` instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

APA3	Meaning
`0b0000`	Address Authentication using the QARMA3 algorithm is not implemented.
`0b0001`	Address Authentication using the QARMA3 algorithm is implemented, FEAT_EPAC and FEAT_PAuth2 are not implemented.
`0b0010`	Address Authentication using the QARMA3 algorithm is implemented, FEAT_EPAC is implemented, and FEAT_PAuth2 is not implemented.
`0b0011`	Address Authentication using the QARMA3 algorithm is implemented, FEAT_EPAC is not implemented, and FEAT_PAuth2 is implemented.
`0b0100`	Address Authentication using the QARMA3 algorithm is implemented, FEAT_EPAC is not implemented, FEAT_PAuth2 and FEAT_FPAC are implemented.
`0b0101`	Address Authentication using the QARMA3 algorithm is implemented, FEAT_EPAC is not implemented, FEAT_PAuth2, FEAT_FPAC, and FEAT_FPACCOMBINE are implemented.
`0b0110`	Address Authentication using the QARMA3 algorithm is implemented, including instructions that allow signing of LR using SP and PC as diversifiers, FEAT_EPAC is not implemented, FEAT_PAuth2, FEAT_FPAC, FEAT_FPACCOMBINE, and FEAT_PAuth_LR are implemented.

All other values are reserved.

FEAT_PAuth implements the functionality identified by the value `0b0001`.

FEAT_EPAC implements the functionality identified by the value `0b0010`.

FEAT_PAuth2 implements the functionality identified by the value `0b0011`.

FEAT_FPAC implements the functionality identified by the value `0b0100`.

FEAT_FPACCOMBINE implements the functionality identified by the value `0b0101`.

FEAT_PAuth_LR implements the functionality identified by the value `0b0110`.

When this field is nonzero, FEAT_PACQARMA3 is implemented.

In Armv8.3, the permitted values are `0b0000`, `0b0001`, `0b0010`, `0b0011`, `0b0100`, and `0b0101`.

From Armv8.6, the permitted values are `0b0000`, `0b0011`, `0b0100`, and `0b0101`.

From Armv9.5, the permitted values are `0b0000`, `0b0011`, `0b0100`, `0b0101`, and `0b0110`.

If the value of ID_AA64ISAR1_EL1.API is nonzero, or the value of ID_AA64ISAR1_EL1.APA is nonzero, this field must have the value `0b0000`.

Access to this field is RO.

GPA3, bits [11:8]

Indicates whether the QARMA3 algorithm is implemented in the PE for generic code authentication in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPA3	Meaning
0b0000	Generic Authentication using the QARMA3 algorithm is not implemented.
0b0001	Generic Authentication using the QARMA3 algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACQARMA3 implements the functionality identified by the value 0b0001.

When this field is nonzero, FEAT_PACQARMA3 is implemented.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is nonzero, or the value of ID_AA64ISAR1_EL1.GPA is nonzero, this field must have the value 0b0000.

Access to this field is RO.

RPRES, bits [7:4]

Indicates support for 12 bits of mantissa in single-precision reciprocal and reciprocal square root instructions in AArch64 state, when FPCR.AH is 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RPRES	Meaning
0b0000	Single-precision reciprocal and reciprocal square root estimates give 8 bits of mantissa, when FPCR.AH is 1.
0b0001	Single-precision reciprocal and reciprocal square root estimates give 12 bits of mantissa, when FPCR.AH is 1.

All other values are reserved.

FEAT_RPRES implements the functionality identified by the value 0b0001.

When FPCR.AH is 0, the floating-point reciprocal estimate and reciprocal square root estimate instructions give 8 bits of mantissa.

Access to this field is RO.

WFxT, bits [3:0]

Indicates support for the WFET and WFIT instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WFxT	Meaning
0b0000	WFET and WFIT are not supported.
0b0010	WFET and WFIT are supported, and the register number is reported in the ESR_ELx on exceptions.

All other values are reserved.

FEAT_WFxT implements the functionality identified by the value 0b0010.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv8.7, the only permitted value is `0b0010`.

Access to this field is RO.

Accessing ID_AA64ISAR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64ISAR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64ISAR2_EL1()) ||
     ↪ImpDefBool("ID_AA64ISAR2_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64ISAR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.86 ID_AA64ISAR3_EL1, AArch64 Instruction Set Attribute Register 3

The ID_AA64ISAR3_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64ISAR3_EL1 are UNDEFINED.

Attributes

ID_AA64ISAR3_EL1 is a 64-bit register.

Field descriptions

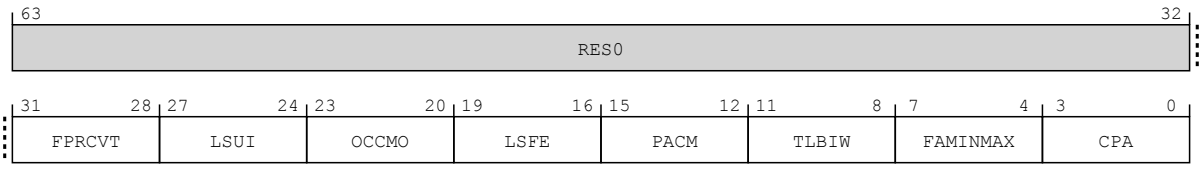

Bits [63:32]

Reserved, RES0.

FPRCVT, bits [31:28]

Indicates support for the following conversion between floating-point and integer instructions with only scalar SIMD&FP register operands and results, and with different input and output register sizes, when the PE is not in Streaming SVE mode:

- FCVTAS (scalar SIMD&FP), FCVTAU (scalar SIMD&FP).
- FCVTMS (scalar SIMD&FP), FCVTMU (scalar SIMD&FP).
- FCVTNS (scalar SIMD&FP), FCVTNU (scalar SIMD&FP).
- FCVTPS (scalar SIMD&FP), FCVTPU (scalar SIMD&FP).
- FCVTZS (scalar SIMD&FP), FCVTZU (scalar SIMD&FP).
- SCVTF (scalar SIMD&FP), UCVTF (scalar SIMD&FP).

If FEAT_SME is implemented, indicates support for the above instructions, as well as the following conversion between floating-point and integer instructions with only scalar SIMD&FP register operands and results, and with the same input and output register sizes, when the PE is in Streaming SVE mode:

- FCVTAS (vector), FCVTAU (vector).
- FCVTMS (vector), FCVTMU (vector).
- FCVTNS (vector), FCVTNU (vector).
- FCVTPS (vector), FCVTPU (vector).
- FCVTZS (vector), FCVTZU (vector).
- SCVTF (vector), UCVTF (vector).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPRCVT	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

FEAT_FPRCVT implements the functionality identified by the value 0b0001.

From Armv9.6, if FEAT_FP is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

LSUI, bits [27:24]

Support for the following load and store unprivileged instructions:

- LDTXR, STTXR.
- LDATXR, STLTXR.
- CAST, CASAT, CASALT, CASLT.
- CASPT, CASPAT, CASPALT, CASPLT.
- LDTP, STTP.
- LDTP (SIMD&FP), STTP (SIMD&FP).
- LDTNP, STTNP.
- LDTNP (SIMD&FP), STTNP (SIMD&FP).
- SWPT, SWPTA, SWPTL, SWPTAL.
- LDTADD, LDTADDA, LDTADDAL, LDTADDL.
- LDTSET, LDTSETA, LDTSETAL, LDTSETL.
- LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL.
- STTADD, STTADDL.
- STTSET, STTSETL.
- STTCLR, STTCLRL.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LSUI	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

FEAT_LSUI implements the functionality identified by the value 0b0001.

Access to this field is RO.

OCCMO, bits [23:20]

Indicates support for the following cache maintenance operation to the Outer cache level instructions:

- DC CIVAOC.
- DC CVAOC.

If FEAT_MTE is implemented:

- DC CIGDVAOC.
- DC CGDVAOC.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OCCMO	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

FEAT_OCCMO implements the functionality identified by the value 0b0001.

From Armv9.6, the value 0b0000 is not permitted.

Access to this field is RO.

LSFE, bits [19:16]

Indicates support for the following A64 Base atomic floating-point in-memory instructions:

- Floating-point atomic add in memory: LDFADD, LDFADDA, LDFADDAL, and LDFADDL.
- BFloat16 atomic add in memory: LDBFADD, LDBFADDA, LDBFADDAL, and LDBFADDL.
- Floating-point atomic maximum in memory: LDFMAX, LDFMAXA, LDFMAXAL, and LDFMAXL.
- BFloat16 atomic maximum in memory: LDBFMAX, LDBFMAXA, LDBFMAXAL, and LDBFMAXL.
- Floating-point atomic maximum number in memory: LDFMAXNM, LDFMAXNMA, LDFMAXNMAL, and LDFMAXNML.
- BFloat16 atomic maximum number in memory: LDBFMAXNM, LDBFMAXNMA, LDBFMAXNMAL, and LDBFMAXNML.
- Floating-point atomic minimum in memory: LDFMIN, LDFMINA, LDFMINAL, and LDFMINL.
- BFloat16 atomic minimum in memory: LDBFMIN, LDBFMINA, LDBFMINAL, and LDBFMINL.
- Floating-point atomic minimum number in memory: LDFMINNM, LDFMINNMA, LDFMINNMAL, and LDFMINNML.
- BFloat16 atomic minimum number in memory: LDBFMINNM, LDBFMINNMA, LDBFMINNMAL, and LDBFMINNML.
- Floating-point atomic add in memory, without return: STFADD and STFADDL.
- BFloat16 atomic add in memory, without return: STBFADD and STBFADDL.
- Floating-point atomic maximum in memory, without return: STFMAX and STFMAXL.
- BFloat16 atomic maximum in memory, without return: STBFMAX and STBFMAXL.
- Floating-point atomic maximum number in memory, without return: STFMAXNM and STFMAXNML.
- BFloat16 atomic maximum number in memory, without return: STBFMAXNM and STBFMAXNML.
- Floating-point atomic minimum in memory, without return: STFMIN and STFMINL.
- BFloat16 atomic minimum in memory, without return: STBFMIN and STBFMINL.
- Floating-point atomic minimum number in memory, without return: STFMINNM and STFMINNML.
- BFloat16 atomic minimum number in memory, without return: STBFMINNM and STBFMINNML.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LSFE	Meaning
0b0000	Atomic floating-point in-memory instructions are not implemented.
0b0001	The specified Atomic floating-point in-memory instructions are implemented.

FEAT_LSFE implements the functionality identified by the value 0b0001

Access to this field is RO.

PACM, bits [15:12]

Indicates the implementation of PSTATE.PACM.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PACM	Meaning
0b0000	PSTATE.PACM is not implemented.
0b0001	Trivial implementation of PSTATE.PACM.
0b0010	Full implementation of PSTATE.PACM.

All other values are reserved.

FEAT_PAuth_LR implements the functionality identified by the values 0b0001 and 0b0010.

If FEAT_PAuth_LR is implemented, the value 0b0000 is not permitted.

If ID_AA64ISAR1_EL1.API is 0b0000, the value 0b0001 is not permitted.

If one of FEAT_PACQARMA3 or FEAT_PACQARMA5 are implemented, the value 0b0001 is not permitted.

Access to this field is RO.

TLBIW, bits [11:8]

Support for TLBI VMALL for Dirty state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TLBIW	Meaning
0b0000	TLBI VMALL for Dirty state is not supported.
0b0001	TLBI VMALL for Dirty state is supported.

All other values are reserved.

FEAT_TLBIW implements the functionality identified by the value 0b0001.

Access to this field is RO.

FAMINMAX, bits [7:4]

Indicates support for the following Advanced SIMD, SVE2, and SME2 instructions that calculate maximum and minimum absolute value:

- When Advanced SIMD is implemented, the Advanced SIMD instructions FAMAX and FAMIN.
- When FEAT_SVE2 or FEAT_SME2 is implemented, the SVE2 instructions FAMAX and FAMIN.
- When FEAT_SME2 is implemented, the SME2 instructions FAMAX (multiple and single vectors), FAMIN (multiple and single vectors), FAMAX (multiple vectors), and FAMIN (multiple vectors).

The value of this field is an IMPLEMENTATION DEFINED choice of:

FAMINMAX	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FAMINMAX implements the functionality identified by the value `0b0001`.

Access to this field is RO.

CPA, bits [3:0]

Indicates support for Checked Pointer Arithmetic instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CPA	Meaning
`0b0000`	Checked Pointer Arithmetic instructions are not implemented.
`0b0001`	Checked Pointer Arithmetic instructions are implemented.
`0b0010`	Checked Pointer Arithmetic instructions are implemented, and Checked Pointer Arithmetic can be enabled.

All other values are reserved.

FEAT_CPA implements the functionality identified by the value `0b0001`.

FEAT_CPA2 implements the functionality identified by the value `0b0010`.

From Armv9.5, the value `0b0000` is not permitted.

Access to this field is RO.

Accessing ID_AA64ISAR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64ISAR3_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b0000`	`0b0110`	`0b011`

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64ISAR3_EL1()) ||
        ↪ImpDefBool("ID_AA64ISAR3_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR3_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ISAR3_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64ISAR3_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.87 ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0

The ID_AA64MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64MMFR0_EL1 are UNDEFINED.

Attributes

ID_AA64MMFR0_EL1 is a 64-bit register.

Field descriptions

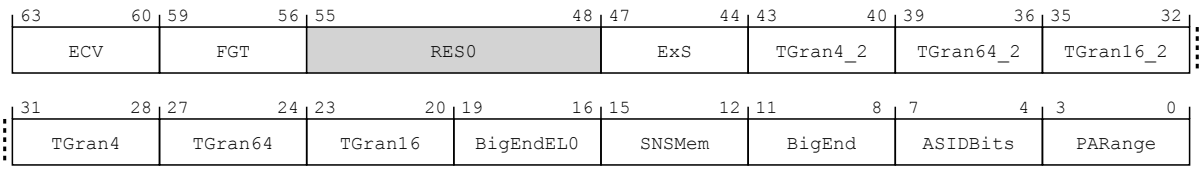

ECV, bits [63:60]

Indicates presence of Enhanced Counter Virtualization.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ECV	Meaning
0b0000	Enhanced Counter Virtualization is not implemented.
0b0001	Enhanced Counter Virtualization is implemented. Supports CNTHCTL_EL2.{EL1TVT, EL1TVCT, EL1NVPCT, EL1NVVCT, EVNTIS}, CNTKCTL_EL1.EVNTIS, CNTPCTSS_EL0 counter views, and CNTVCTSS_EL0 counter views. Extends the PMSCR_EL1.PCT, PMSCR_EL2.PCT, TRFCR_EL1.TS, and TRFCR_EL2.TS fields.
0b0010	As 0b0001, and the CNTPOFF_EL2 register and the CNTHCTL_EL2.ECV and SCR_EL3.ECVEn fields are implemented.

All other values are reserved.

FEAT_ECV implements the functionality identified by the value 0b0001.

FEAT_ECV_POFF implements the functionality identified by the value 0b0010.

From Armv8.6, the value 0b0000 is not permitted.

Access to this field is RO.

FGT, bits [59:56]

Indicates presence of the Fine-Grained Trap controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

FGT	Meaning
0b0000	Fine-grained trap controls are not implemented.
0b0001	Fine-grained trap controls are implemented. Supports: • If EL2 is implemented, the HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers, and their associated traps. • If EL2 is implemented, MDCR_EL2.TDCC. • If EL3 is implemented, MDCR_EL3.TDCC. • If both EL2 and EL3 are implemented, SCR_EL3.FGTEn.
0b0010	As 0b0001, and also includes support for: • If EL2 is implemented, the HDFGRTR2_EL2, HDFGWTR2_EL2, HFGITR2_EL2, HFGRTR2_EL2, and HFGWTR2_EL2 registers, and their associated traps. • If both EL2 and EL3 are implemented, SCR_EL3.FGTEn2.

All other values are reserved.

FEAT_FGT implements the functionality identified by the value 0b0001.

FEAT_FGT2 implements the functionality identified by the value 0b0010.

From Armv8.6, the value 0b0000 is not permitted.

From Armv8.9, the value 0b0001 is not permitted.

Access to this field is RO.

Bits [55:48]

Reserved, RES0.

ExS, bits [47:44]

Indicates support for disabling context synchronizing exception entry and exit.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ExS	Meaning
0b0000	All exception entries and exits are context synchronization events.
0b0001	Non-context synchronizing exception entry and exit are supported.

All other values are reserved.

FEAT_ExS implements the functionality identified by the value 0b0001.

Access to this field is RO.

TGran4_2, bits [43:40]

Indicates support for 4KB memory granule size at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TGran4_2	Meaning	Applies when
0b0000	Support for 4KB granule at stage 2 is identified in the ID_AA64MMFR0_EL1.TGran4 field.	
0b0001	4KB granule not supported at stage 2.	
0b0010	4KB granule supported at stage 2.	
0b0011	4KB granule at stage 2 supports 52-bit input addresses and can describe 52-bit output addresses.	FEAT_LPA2 is implemented

All other values are reserved.

If EL2 is not implemented, this field is 0b0000. Otherwise, the value 0b0000 is deprecated.

Note

This field does not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes for more information.

Access to this field is RO.

TGran64_2, bits [39:36]

Indicates support for 64KB memory granule size at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TGran64_2	Meaning
0b0000	Support for 64KB granule at stage 2 is identified in the ID_AA64MMFR0_EL1.TGran64 field.
0b0001	64KB granule not supported at stage 2.
0b0010	64KB granule supported at stage 2.

All other values are reserved.

If EL2 is not implemented, this field is 0b0000. Otherwise, the value 0b0000 is deprecated.

Note

This field does not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes for more information.

Access to this field is RO.

TGran16_2, bits [35:32]

Indicates support for 16KB memory granule size at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TGran16_2	Meaning	Applies when
0b0000	Support for 16KB granule at stage 2 is identified in the ID_AA64MMFR0_EL1.TGran16 field.	
0b0001	16KB granule not supported at stage 2.	
0b0010	16KB granule supported at stage 2.	
0b0011	16KB granule at stage 2 supports 52-bit input addresses and can describe 52-bit output addresses.	FEAT_LPA2 is implemented

All other values are reserved.

If EL2 is not implemented, this field is 0b0000. Otherwise, the value 0b0000 is deprecated.

Note

This field does not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes for more information.

Access to this field is RO.

TGran4, bits [31:28]

Indicates support for 4KB memory translation granule size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TGran4	Meaning	Applies when
0b0000	4KB granule supported.	
0b0001	4KB granule supports 52-bit input addresses and can describe 52-bit output addresses.	FEAT_LPA2 is implemented
0b1111	4KB granule not supported.	

All other values are reserved.

Access to this field is RO.

TGran64, bits [27:24]

Indicates support for 64KB memory translation granule size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TGran64	Meaning
0b0000	64KB granule supported.
0b1111	64KB granule not supported.

All other values are reserved.

Access to this field is RO.

TGran16, bits [23:20]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates support for 16KB memory translation granule size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TGran16	Meaning	Applies when
0b0000	16KB granule not supported.	
0b0001	16KB granule supported.	
0b0010	16KB granule supports 52-bit input addresses and can describe 52-bit output addresses.	FEAT_LPA2 is implemented

All other values are reserved.

Access to this field is RO.

BigEndEL0, bits [19:16]

Indicates support for mixed-endian at EL0 only.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BigEndEL0	Meaning
0b0000	No mixed-endian support at EL0. The SCTLR_EL1.E0E bit has a fixed value.
0b0001	Mixed-endian support at EL0. The SCTLR_EL1.E0E bit can be configured.

FEAT_MixedEndEL0 implements the functionality identified by the value 0b0001.

All other values are reserved.

This field is invalid and is RES0 if ID_AA64MMFR0_EL1.BigEnd is not 0b0000.

Access to this field is RO.

SNSMem, bits [15:12]

Indicates support for a distinction between Secure and Non-secure Memory.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SNSMem	Meaning
0b0000	Does not support a distinction between Secure and Non-secure Memory.
0b0001	Does support a distinction between Secure and Non-secure Memory.

Note

If EL3 is implemented, the value 0b0000 is not permitted.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BigEnd, bits [11:8]

Indicates support for mixed-endian configuration.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BigEnd	Meaning
0b0000	No mixed-endian support. The SCTLR_ELx.EE bits have a fixed value. See the BigEndEL0 field, bits[19:16], for whether EL0 supports mixed-endian.
0b0001	Mixed-endian support. The SCTLR_ELx.EE and SCTLR_EL1.E0E bits can be configured.

FEAT_MixedEnd implements the functionality identified by the value 0b0001.

If this field is 0b0001, FEAT_MixedEndEL0 is also implemented.

All other values are reserved.

Access to this field is RO.

ASIDBits, bits [7:4]

Number of ASID bits.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ASIDBits	Meaning
0b0000	8 bits.
0b0010	16 bits.

All other values are reserved.

Access to this field is RO.

PARange, bits [3:0]

Physical Address range supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PARange	Meaning	Applies when
0b0000	32 bits, 4GB.	
0b0001	36 bits, 64GB.	
0b0010	40 bits, 1TB.	
0b0011	42 bits, 4TB.	
0b0100	44 bits, 16TB.	
0b0101	48 bits, 256TB.	
0b0110	52 bits, 4PB.	FEAT_LPA is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0111	56 bits, 64PB.	FEAT_D128 is implemented

All other values are reserved.

Access to this field is RO.

Accessing ID_AA64MMFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64MMFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64MMFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.88 ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1

The ID_AA64MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64MMFR1_EL1 are UNDEFINED.

Attributes

ID_AA64MMFR1_EL1 is a 64-bit register.

Field descriptions

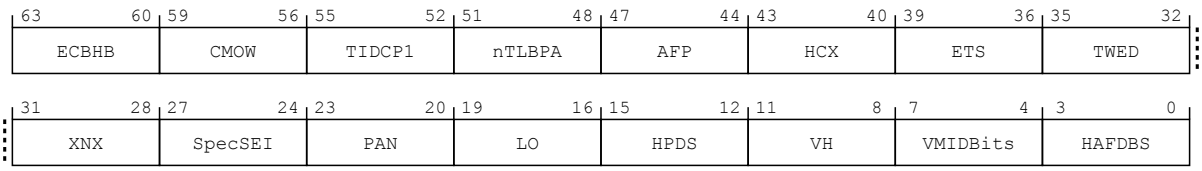

ECBHB, bits [63:60]

Indicates support for restrictions on branch history speculation around exceptions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ECBHB	Meaning
0b0000	The implementation does not disclose whether restrictions are imposed on branch history speculation around exceptions.
0b0001	The implementation imposes restrictions on branch history speculation around exceptions.

All other values are reserved.

FEAT_ECBHB implements the functionality identified by the value 0b0001.

From Armv8.9, the value 0b0000 is not permitted.

Access to this field is RO.

CMOW, bits [59:56]

Indicates support for cache maintenance instruction permission.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CMOW	Meaning
0b0000	SCTLR_EL1.CMOW, SCTLR_EL2.CMOW, and HCRX_EL2.CMOW bits are not implemented.
0b0001	SCTLR_EL1.CMOW is implemented. If EL2 is implemented, SCTLR_EL2.CMOW and HCRX_EL2.CMOW bits are implemented.

All other values are reserved.

FEAT_CMOW implements the functionality identified by the value 0b0001.

From Armv8.8, the value 0b0000 is not permitted.

Access to this field is RO.

TIDCP1, bits [55:52]

Indicates whether SCTLR_EL1.TIDCP and SCTLR_EL2.TIDCP are implemented in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TIDCP1	Meaning
0b0000	SCTLR_EL1.TIDCP and SCTLR_EL2.TIDCP bits are not implemented and are RES0.
0b0001	SCTLR_EL1.TIDCP bit is implemented. If EL2 is implemented, SCTLR_EL2.TIDCP bit is implemented.

All other values are reserved.

FEAT_TIDCP1 implements the functionality identified by the value 0b0001.

From Armv8.8, the value 0b0000 is not permitted.

Access to this field is RO.

nTLBPA, bits [51:48]

Indicates support for intermediate caching of translation table walks.

The value of this field is an IMPLEMENTATION DEFINED choice of:

nTLBPA	Meaning
0b0000	The intermediate caching of translation table walks might include non-coherent physical translation caches.
0b0001	The intermediate caching of translation table walks does not include non-coherent physical translation caches.

Non-coherent physical translation caches are non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE, where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

FEAT_nTLBPA implements the functionality identified by the value 0b0001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

AFP, bits [47:44]

Indicates support for FPCR.{AH, FIZ, NEP}.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AFP	Meaning
0b0000	The FPCR.{AH, FIZ, NEP} fields are not supported.
0b0001	The FPCR.{AH, FIZ, NEP} fields are supported.

All other values are reserved.

FEAT_AFP implements the functionality identified by the value 0b0001.

From Armv8.7, if Advanced SIMD and floating-point is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

HCX, bits [43:40]

Indicates support for HCRX_EL2 and its associated EL3 trap.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HCX	Meaning
0b0000	HCRX_EL2 and its associated EL3 trap are not supported.
0b0001	HCRX_EL2 and its associated EL3 trap are supported.

All other values are reserved.

FEAT_HCX implements the functionality identified by the value 0b0001.

From Armv8.7, if EL2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

ETS, bits [39:36]

Indicates support for Enhanced Translation Synchronization.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ETS	Meaning
0b0000	Enhanced Translation Synchronization is not supported.
0b0001	Enhanced Translation Synchronization is not supported.
0b0010	FEAT_ETS2 is implemented.
0b0011	FEAT_ETS3 is implemented.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ETS2 implements the functionality identified by the value `0b0010`.

FEAT_ETS3 implements the functionality identified by the value `0b0011`.

From Armv8.8, the values `0b0000` and `0b0001` are not permitted.

From Armv9.5, the value `0b0010` is not permitted.

Access to this field is RO.

TWED, bits [35:32]

Indicates support for the configurable delayed trapping of WFE and WFET.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TWED	Meaning
`0b0000`	Configurable delayed trapping of WFE and WFET are not supported.
`0b0001`	Configurable delayed trapping of WFE and WFET are supported.

All other values are reserved.

FEAT_TWED implements the functionality identified by the value `0b0001`.

Access to this field is RO.

XNX, bits [31:28]

Indicates support for execute-never control distinction by Exception level at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

XNX	Meaning
`0b0000`	Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
`0b0001`	Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

FEAT_XNX implements the functionality identified by the value `0b0001`.

From Armv8.2, the value `0b0000` is not permitted.

Access to this field is RO.

SpecSEI, bits [27:24]

When FEAT_RAS is implemented:

Describes whether the PE can generate SError exceptions from speculative reads of memory, including speculative instruction fetches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SpecSEI	Meaning
0b0000	The PE never generates an SError exception due to an External abort on a speculative read.
0b0001	The PE might generate an SError exception due to an External abort on a speculative read.

All other values are reserved.

FEAT_SpecSEI implements the functionality identified by the value 0b0001.

Access to this field is RO.

Otherwise:

Reserved, RES0.

PAN, bits [23:20]

Privileged Access Never. Indicates support for the PAN bit in PSTATE, SPSR_EL1, SPSR_EL2, SPSR_EL3, and DSPSR_EL0.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PAN	Meaning
0b0000	PAN not supported.
0b0001	PAN supported.
0b0010	PAN supported and AT S1E1RP and AT S1E1WP instructions supported.
0b0011	PAN supported, AT S1E1RP and AT S1E1WP instructions supported, and SCTLR_EL1.EPAN and SCTLR_EL2.EPAN bits supported.

All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

FEAT_PAN3 implements the functionality added by the value 0b0011.

From Armv8.1, the value 0b0000 is not permitted.

From Armv8.2, the value 0b0001 is not permitted.

From Armv8.7, the value 0b0010 is not permitted.

Access to this field is RO.

LO, bits [19:16]

LORegions. Indicates support for LORegions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LO	Meaning
0b0000	LORegions not supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LO	Meaning
0b0001	LORegions supported.

All other values are reserved.

FEAT_LOR implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

HPDS, bits [15:12]

Hierarchical Permission Disables. Indicates support for disabling hierarchical controls in translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HPDS	Meaning
0b0000	Disabling of hierarchical controls not supported.
0b0001	Disabling of hierarchical controls supported with the TCR_EL1.{HPD1, HPD0}, TCR_EL2.HPD or TCR_EL2.{HPD1, HPD0}, and TCR_EL3.HPD bits.
0b0010	As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

FEAT_HPDS implements the functionality identified by the value 0b0001.

FEAT_HPDS2 implements the functionality identified by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

VH, bits [11:8]

Virtualization Host Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VH	Meaning
0b0000	Virtualization Host Extensions not supported.
0b0001	Virtualization Host Extensions supported.

All other values are reserved.

FEAT_VHE implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

VMIDBits, bits [7:4]

Number of VMID bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDBits	Meaning
0b0000	8 bits
0b0010	16 bits

All other values are reserved.

FEAT_VMID16 implements the functionality identified by the value 0b0010.

Access to this field is RO.

HAFDBS, bits [3:0]

Hardware updates to Access flag and Dirty state in translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HAFDBS	Meaning
0b0000	Hardware update of the Access flag and dirty state are not supported.
0b0001	Support for hardware update of the Access flag for Block and Page descriptors.
0b0010	As 0b0001, and adds support for hardware update of dirty state.
0b0011	As 0b0010, and adds support for hardware update of the Access flag for Table descriptors.
0b0100	As 0b0011, and adds support for hardware tracking of Dirty state Structure.

All other values are reserved.

FEAT_HAF implements the functionality identified by the value 0b0001.

FEAT_HAFDBS implements the functionality identified by the value 0b0010.

FEAT_HAFT implements the functionality identified by the value 0b0011.

FEAT_HDBSS implements the functionality identified by the value 0b0100.

Access to this field is RO.

Accessing ID_AA64MMFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64MMFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64MMFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.89 ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2

The ID_AA64MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64MMFR2_EL1 are UNDEFINED.

Attributes

ID_AA64MMFR2_EL1 is a 64-bit register.

Field descriptions

63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
E0PD	EVT	BBM	TTL	RES0	FWB	IDS	AT

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
ST	NV	CCIDX	VARange	IESB	LSM	UAO	CnP

E0PD, bits [63:60]

Indicates support for the E0PD mechanism.

The value of this field is an IMPLEMENTATION DEFINED choice of:

E0PD	Meaning
0b0000	E0PDx mechanism is not implemented.
0b0001	E0PDx mechanism is implemented.

All other values are reserved.

FEAT_E0PD implements the functionality identified by the value 0b0001.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVT, bits [59:56]

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EVT	Meaning
0b0000	HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps are not supported.
0b0001	HCR_EL2.{TOCU, TICAB, TID4} traps are supported.
0b0010	As 0b0001, and HCR_EL2.{TTLBOS, TTLBIS} traps are supported.

All other values are reserved.

FEAT_EVT implements the functionality identified by the value 0b0001.

FEAT_EVT2 implements the functionality identified by the value 0b0010.

If EL2 is not implemented, the only permitted value is 0b0000.

From Armv8.5, if EL2 is implemented, the value 0b0001 is not permitted.

Access to this field is RO.

BBM, bits [55:52]

Allows identification of the requirements of the hardware to have break-before-make sequences when changing block or table size for a translation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BBM	Meaning
0b0000	Break-before-make sequence must be used.
0b0001	Level 1 support for changing block size is supported.
0b0010	Level 2 support for changing block size is supported.

All other values are reserved.

FEAT_BBML1 implements the functionality identified by the value 0b0001.

FEAT_BBML2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

TTL, bits [51:48]

Indicates support for TTL field in address operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b0000	TLB maintenance instructions by address have bits[47:44] as RES0.
0b0001	TLB maintenance instructions by address have bits[47:44] holding the TTL field.

All other values are reserved.

FEAT_TTL implements the functionality identified by the value 0b0001.

This field affects TLBI IPAS2E1, TLBI IPAS2E1IS, TLBI IPAS2E1OS, TLBI IPAS2LE1, TLBI IPAS2LE1IS, TLBI IPAS2LE1OS, TLBI VAAE1, TLBI VAAE1IS, TLBI VAAE1OS, TLBI VAALE1, TLBI VAALE1IS, TLBI VAALE1OS, TLBI VAE1, TLBI VAE1IS, TLBI VAE1OS, TLBI VAE2, TLBI VAE2IS, TLBI VAE2OS, TLBI VAE3, TLBI VAE3IS, TLBI VAE3OS,TLBI VALE1, TLBI VALE1IS, TLBI VALE1OS, TLBI VALE2, TLBI VALE2IS, TLBI VALE2OS, TLBI VALE3, TLBI VALE3IS, TLBI VALE3OS.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

Bits [47:44]

Reserved, RES0.

FWB, bits [43:40]

Indicates support for HCR_EL2.FWB.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FWB	Meaning
0b0000	HCR_EL2.FWB bit is not supported.
0b0001	HCR_EL2.FWB is supported.

All other values are reserved.

FEAT_S2FWB implements the functionality identified by the value 0b0001.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

IDS, bits [39:36]

Indicates the EC syndrome value in ESR_ELx.EC that is reported if an exception is generated by a read access to the feature ID space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IDS	Meaning
0b0000	An exception which is generated by a read access to the feature ID space, other than a trap caused by HCR_EL2.TIDx, SCTLR_EL1.UCT, or SCTLR_EL2.UCT is reported by ESR_ELx.EC == 0x0.
0b0001	All exceptions generated by an AArch64 read access to the feature ID space are reported by ESR_ELx.EC == 0x18.
0b0010	As 0b0001 and introduces support for trapping ID register accesses to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

The Feature ID space is defined as the System register space in AArch64 with op0==3, op1=={0, 1, 3}, CRn==0, CRm=={0-7}, op2=={0-7}.

FEAT_IDST implements the functionality identified by the value `0b0001`.

FEAT_IDTE3 implements the functionality identified by `0b0010`.

From Armv8.4, the value `0b0000` is not permitted.

From Armv9.6, if EL3 is implemented, the value `0b0001` is not permitted.

Access to this field is RO.

AT, bits [35:32]

Identifies support for unaligned single-copy atomicity and atomic functions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AT	Meaning
`0b0000`	Unaligned single-copy atomicity and atomic functions are not supported.
`0b0001`	Unaligned single-copy atomicity and atomic functions with a 16-byte address range aligned to 16-bytes are supported.

All other values are reserved.

FEAT_LSE2 implements the functionality identified by the value `0b0001`.

From Armv8.4, the value `0b0000` is not permitted.

Access to this field is RO.

ST, bits [31:28]

Identifies support for small translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ST	Meaning
`0b0000`	The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 39.
`0b0001`	The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 48 for 4KB and 16KB granules, and 47 for 64KB granules.

All other values are reserved.

FEAT_TTST implements the functionality identified by the value `0b0001`.

When FEAT_SEL2 is implemented, the value `0b0000` is not permitted.

Access to this field is RO.

NV, bits [27:24]

Nested Virtualization. If EL2 is implemented, indicates support for the use of nested virtualization.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NV	Meaning
0b0000	If ID_AA64MMFR4_EL1.NV_frac != 0b0000, support for nested virtualization is described in ID_AA64MMFR4_EL1.NV_frac. Otherwise, nested virtualization is not supported.
0b0001	The HCR_EL2.{AT, NV1, NV} bits are implemented.
0b0010	The VNCR_EL2 register and the HCR_EL2.{NV2, AT, NV1, NV} bits are implemented.

All other values are reserved.

If EL2 is not implemented, the only permitted value is 0b0000.

FEAT_NV implements the functionality identified by the value 0b0001.

FEAT_NV2 implements the functionality identified by the value 0b0010.

In Armv8.3, if EL2 is implemented, the permitted values are 0b0000 and 0b0001.

From Armv8.4, if EL2 is implemented, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

CCIDX, bits [23:20]

Support for the use of revised CCSIDR_EL1 register format.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCIDX	Meaning
0b0000	32-bit format implemented for all levels of the CCSIDR_EL1.
0b0001	64-bit format implemented for all levels of the CCSIDR_EL1.

All other values are reserved.

FEAT_CCIDX implements the functionality identified by the value 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

VARange, bits [19:16]

Indicates support for a larger virtual address.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VARange	Meaning	Applies when
0b0000	VMSAv8-64 supports 48-bit VAs.	
0b0001	VMSAv8-64 supports 52-bit VAs when using the 64KB translation granule. The size for other translation granules is not defined by this field.	
0b0010	VMSAv9-128 supports 56-bit VAs.	FEAT_D128 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_LVA implements the functionality identified by the value `0b0001`.

FEAT_LVA3 implements the functionality identified by the value `0b0010`.

Access to this field is RO.

IESB, bits [15:12]

Indicates support for the IESB fields in the SCTLR_ELx and ESR_ELx registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IESB	Meaning
`0b0000`	IESB fields in the SCTLR_ELx and ESR_ELx registers is not supported.
`0b0001`	IESB fields in the SCTLR_ELx and ESR_ELx registers is supported. If an SError exception is taken ELx as a result of the error synchronization event generated on exception entry by the FEAT_IESB mechanism, then the PE sets ESR_ELx.IESB bit in the SError exception syndrome to `0b1`. If an SError exception is taken to ELx as a result of the error synchronization event generated on exception return by the FEAT_IESB mechanism, then the PE sets the ESR_ELx.IESB bit in the SError exception syndrome to an IMPLEMENTATION DEFINED choice of `0b0` or `0b1`.

All other values are reserved.

FEAT_IESB implements the functionality identified by the value `0b0001`.

Access to this field is RO.

LSM, bits [11:8]

Indicates support for LSMAOE and nTLSMD bits in SCTLR_EL1 and SCTLR_EL2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LSM	Meaning
`0b0000`	LSMAOE and nTLSMD bits not supported.
`0b0001`	LSMAOE and nTLSMD bits supported.

All other values are reserved.

FEAT_LSMAOC implements the functionality identified by the value `0b0001`.

Access to this field is RO.

UAO, bits [7:4]

User Access Override.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UAO	Meaning
0b0000	UAO not supported.
0b0001	UAO supported.

All other values are reserved.

FEAT_UAO implements the functionality identified by the value 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

Access to this field is RO.

CnP, bits [3:0]

Indicates support for Common not Private translations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CnP	Meaning
0b0000	Common not Private translations not supported.
0b0001	Common not Private translations supported.

All other values are reserved.

FEAT_TTCNP implements the functionality identified by the value 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

Access to this field is RO.

Accessing ID_AA64MMFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64MMFR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64MMFR2_EL1()) ||
         ↪ImpDefBool("ID_AA64MMFR2_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
         ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64MMFR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.90 ID_AA64MMFR3_EL1, AArch64 Memory Model Feature Register 3

The ID_AA64MMFR3_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64MMFR3_EL1 are UNDEFINED.

Attributes

ID_AA64MMFR3_EL1 is a 64-bit register.

Field descriptions

63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
Spec_FPACC	ADERR	SDERR	RES0	ANERR	SNERR	D128_2	D128

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
MEC	AIE	S2POE	S1POE	S2PIE	S1PIE	SCTLRX	TCRX

Spec_FPACC, bits [63:60]

When FEAT_FPACCOMBINE is implemented:

Speculative behavior in the event of a PAC authentication failure in an implementation that includes FEAT_FPACCOMBINE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Spec_FPACC	Meaning
0b0000	The implementation does not disclose whether the speculative use of pointers processed by a PAC Authentication is materially different in terms of the impact on cached microarchitectural state between passing and failing of the PAC Authentication.
0b0001	The speculative use of pointers processed by a PAC Authentication is not materially different in terms of the impact on cached microarchitectural state between passing and failing of the PAC Authentication.

All other values are reserved.

For the purpose of this definition, cached microarchitecture state is the state of caching agents such as instruction caches, data caches and TLBs which can be altered as a result of speculation caused by a mispredicted execution, but is not restored to the state prior to the speculation when the misprediction is corrected.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

ADERR, bits [59:56]

Asynchronous Device error exceptions. With ID_AA64MMFR3_EL1.SDERR, describes the PE behavior for External aborts signaled on Device memory loads.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ADERR	Meaning
0b0000	If FEAT_RASv2 is not implemented and ID_AA64MMFR3_EL1.SDERR is 0b0000, then the behavior is not described. Otherwise, the behavior is described by ID_AA64MMFR3_EL1.SDERR.
0b0001	All External aborts on Device memory loads are handled asynchronously.
0b0010	FEAT_ADERR is implemented. SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR are implemented. If FEAT_ANERR is also implemented, then software should ensure that all the following apply: • SCTLR2_ELx.EnADERR is set to the value of SCTLR2_ELx.EnANERR. • HCRX_EL2.EnSDERR is set to the value of HCRX_EL2.EnSNERR.
0b0011	FEAT_ADERR is implemented. SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR are implemented. If FEAT_ANERR is also implemented, then SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR operate independently of SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR.

All other values are reserved.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

When FEAT_RASv2 is implemented and ID_AA64MMFR3_EL1.SDERR is 0b0000, the value of this field is 0b0001.

When ID_AA64MMFR3_EL1.SDERR is 0b0001, the value of this field is 0b0000.

When ID_AA64MMFR3_EL1.SDERR is 0b0010, the value of this field is 0b0010.

When ID_AA64MMFR3_EL1.SDERR is 0b0011, the value of this field is 0b0011.

FEAT_ADERR implements the functionality described by the value 0b0010.

Access to this field is RO.

SDERR, bits [55:52]

Synchronous Device error exceptions. With ID_AA64MMFR3_EL1.ADERR, describes the PE behavior for External aborts signaled on Device memory loads.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SDERR	Meaning
0b0000	If FEAT_RASv2 is not implemented and ID_AA64MMFR3_EL1.ADERR is 0b0000, then the behavior is not described. Otherwise, the behavior is described by ID_AA64MMFR3_EL1.ADERR.
0b0001	All External aborts on Device memory loads are handled synchronously.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SDERR	Meaning
0b0010	FEAT_ADERR is implemented. SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR are implemented. If FEAT_ANERR is also implemented, then software should ensure that all the following apply: • SCTLR2_ELx.EnADERR is set to the value of SCTLR2_ELx.EnANERR. • HCRX_EL2.EnSDERR is set to the value of HCRX_EL2.EnSNERR.
0b0011	FEAT_ADERR is implemented. SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR are implemented. If FEAT_ANERR is also implemented, then SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR operate independently of SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR.

All other values are reserved.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

When FEAT_RASv2 is implemented and ID_AA64MMFR3_EL1.ADERR is 0b0000, the value of this field is 0b0001.

When ID_AA64MMFR3_EL1.ADERR is 0b0001, the value of this field is 0b0000.

When ID_AA64MMFR3_EL1.ADERR is 0b0010, the value of this field is 0b0010.

When ID_AA64MMFR3_EL1.ADERR is 0b0011, the value of this field is 0b0011.

FEAT_ADERR implements the functionality described by the value 0b0010.

Access to this field is RO.

Bits [51:48]

Reserved, RES0.

ANERR, bits [47:44]

Asynchronous Normal error exceptions. With ID_AA64MMFR3_EL1.SNERR, describes the PE behavior for External aborts signaled on Normal memory loads.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ANERR	Meaning
0b0000	If FEAT_RASv2 is not implemented and ID_AA64MMFR3_EL1.SNERR is 0b0000, then the behavior is not described. Otherwise, the behavior is described by ID_AA64MMFR3_EL1.SNERR.
0b0001	All External aborts on Normal memory loads are handled asynchronously.
0b0010	FEAT_ANERR is implemented. SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR are implemented. If FEAT_ADERR is also implemented, then software should ensure that all the following apply: • SCTLR2_ELx.EnANERR is set to the value of SCTLR2_ELx.EnADERR. • HCRX_EL2.EnSNERR is set to the value of HCRX_EL2.EnSDERR.
0b0011	FEAT_ANERR is implemented. SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR are implemented. If FEAT_ADERR is also implemented, then SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR operate independently of SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

When FEAT_RASv2 is implemented and ID_AA64MMFR3_EL1.SNERR is `0b0000`, the value of this field is `0b0001`.

When ID_AA64MMFR3_EL1.SNERR is `0b0001`, the value of this field is `0b0000`.

When ID_AA64MMFR3_EL1.SNERR is `0b0010`, the value of this field is `0b0010`.

When ID_AA64MMFR3_EL1.SNERR is `0b0011`, the value of this field is `0b0011`.

FEAT_ANERR implements the functionality described by the value `0b0010`.

Access to this field is RO.

SNERR, bits [43:40]

Synchronous Normal error exceptions. With ID_AA64MMFR3_EL1.ANERR, describes the PE behavior for External aborts signaled on Normal memory loads.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SNERR	Meaning
`0b0000`	If FEAT_RASv2 is not implemented and ID_AA64MMFR3_EL1.ANERR is `0b0000`, then the behavior is not described. Otherwise, the behavior is described by ID_AA64MMFR3_EL1.ANERR.
`0b0001`	All External aborts on Normal memory loads are handled synchronously.
`0b0010`	FEAT_ANERR is implemented. SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR are implemented. If FEAT_ADERR is also implemented, then software should ensure that all the following apply: • SCTLR2_ELx.EnANERR is set to the value of SCTLR2_ELx.EnADERR. • HCRX_EL2.EnSNERR is set to the value of HCRX_EL2.EnSDERR.
`0b0011`	FEAT_ANERR is implemented. SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR are implemented. If FEAT_ADERR is also implemented, then SCTLR2_ELx.EnANERR and HCRX_EL2.EnSNERR operate independently of SCTLR2_ELx.EnADERR and HCRX_EL2.EnSDERR.

All other values are reserved.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

When FEAT_RASv2 is implemented and ID_AA64MMFR3_EL1.ANERR is `0b0000`, the value of this field is `0b0001`.

When ID_AA64MMFR3_EL1.ANERR is `0b0001`, the value of this field is `0b0000`.

When ID_AA64MMFR3_EL1.ANERR is `0b0010`, the value of this field is `0b0010`.

When ID_AA64MMFR3_EL1.ANERR is `0b0011`, the value of this field is `0b0011`.

FEAT_ANERR implements the functionality described by the value `0b0010`.

Access to this field is RO.

D128_2, bits [39:36]

128-bit translation table descriptor at stage 2. Indicates support for 128-bit translation table descriptor at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D128_2	Meaning
0b0000	128-bit translation table descriptor Extension at stage 2 is not supported.
0b0001	128-bit translation table descriptor Extension at stage 2 is supported.

All other values are reserved.

Access to this field is RO.

D128, bits [35:32]

128-bit translation table descriptor. Indicates support for 128-bit translation table descriptor.

The value of this field is an IMPLEMENTATION DEFINED choice of:

D128	Meaning
0b0000	128-bit translation table descriptor Extension is not supported.
0b0001	128-bit translation table descriptor Extension is supported.

All other values are reserved.

Access to this field is RO.

MEC, bits [31:28]

Indicates support for Memory Encryption Contexts.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MEC	Meaning
0b0000	Memory Encryption Contexts is not supported.
0b0001	Memory Encryption Contexts is supported, with multiple contexts in the Realm physical address space.

All other values are reserved.

FEAT_MEC implements the functionality identified by the value 0b0001.

Access to this field is RO.

AIE, bits [27:24]

Attribute Indexing. Indicates support for the Attribute Index Enhancement.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AIE	Meaning
0b0000	The Attribute Index Enhancement is not supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0001	The Attribute Index Enhancement at stage 1 is supported.

All other values are reserved.

FEAT_AIE implements the functionality identified by the value 0b0001.

Access to this field is RO.

S2POE, bits [23:20]

Stage 2 Permission Overlay. Indicates support for Permission Overlay at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

S2POE	Meaning
0b0000	Permission Overlay at stage 2 is not supported.
0b0001	Permission Overlay at stage 2 is supported.

All other values are reserved.

FEAT_S2POE implements the functionality identified by the value 0b0001.

Access to this field is RO.

S1POE, bits [19:16]

Stage 1 Permission Overlay. Indicates support for Permission Overlay at stage 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

S1POE	Meaning
0b0000	Permission Overlay at stage 1 is not supported.
0b0001	Permission Overlay at stage 1 is supported.

All other values are reserved.

FEAT_S1POE implements the functionality identified by the value 0b0001.

Access to this field is RO.

S2PIE, bits [15:12]

Stage 2 Permission Indirection. Indicates support for Permission Indirection at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

S2PIE	Meaning
0b0000	Permission Indirection at stage 2 is not supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0001	Permission Indirection at stage 2 is supported.

All other values are reserved.

FEAT_S2PIE implements the functionality identified by the value 0b0001.

Access to this field is RO.

S1PIE, bits [11:8]

Stage 1 Permission Indirection. Indicates support for Permission Indirection at stage 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

S1PIE	Meaning
0b0000	Permission Indirection at stage 1 is not supported.
0b0001	Permission Indirection at stage 1 is supported.

All other values are reserved.

FEAT_S1PIE implements the functionality identified by the value 0b0001.

Access to this field is RO.

SCTLRX, bits [7:4]

SCTLR Extension. Indicates support for extension of SCTLR_ELx.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SCTLRX	Meaning
0b0000	SCTLR2_EL1, SCTLR2_EL2, SCTLR2_EL3 registers, and their associated trap controls are not implemented.
0b0001	SCTLR2_EL1, SCTLR2_EL2, SCTLR2_EL3 resisters, and their associated trap controls are implemented.

All other values are reserved.

From Armv8.9, the value 0b0000 is not permitted.

FEAT_SCTLR2 implements the functionality described by the value 0b0001.

Access to this field is RO.

TCRX, bits [3:0]

TCR Extension. Indicates support for extension of TCR_ELx.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCRX	Meaning
0b0000	TCR2_EL1, TCR2_EL2, and their associated trap controls are not implemented.
0b0001	TCR2_EL1, TCR2_EL2, and their associated trap controls are implemented.

All other values are reserved.

From Armv8.9, the value 0b0000 is not permitted.

FEAT_TCR2 implements the functionality described by the value 0b0001.

Access to this field is RO.

Accessing ID_AA64MMFR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64MMFR3_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64MMFR3_EL1()) ||
     ↪ImpDefBool("ID_AA64MMFR3_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR3_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = ID_AA64MMFR3_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64MMFR3_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.91 ID_AA64MMFR4_EL1, AArch64 Memory Model Feature Register 4

The ID_AA64MMFR4_EL1 characteristics are:

Purpose

Provides additional information about implemented memory model and memory management support in AArch64.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64MMFR4_EL1 are UNDEFINED.

Attributes

ID_AA64MMFR4_EL1 is a 64-bit register.

Field descriptions

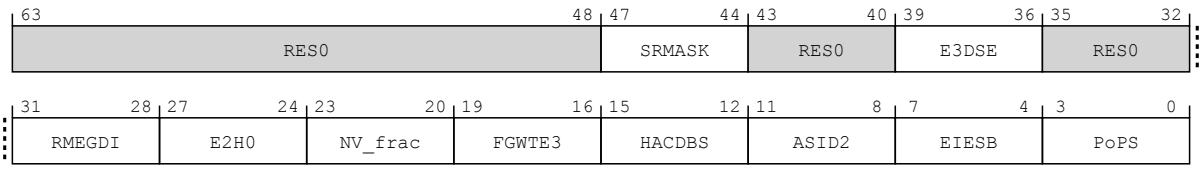

Bits [63:48]

Reserved, RES0.

SRMASK, bits [47:44]

Indicates support for bitwise write masks for the following registers:

- ACTLR_EL1.
- CPACR_EL1.
- SCTLR_EL1.
- SCTLR2_EL1.
- TCR_EL1.
- TCR2_EL1.
- ACTLR_EL2.
- CPTR_EL2.
- SCTLR_EL2.
- SCTLR2_EL2.
- TCR_EL2.
- TCR2_EL2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SRMASK	Meaning
0b0000	Bitwise write masks for the specified registers are not supported.
0b0001	Bitwise write masks for the specified registers are supported.

FEAT_SRMASK implements the functionality identified by the value 0b0001.

From Armv9.6, the value 0b0000 is not permitted.

Access to this field is RO.

Bits [43:40]

Reserved, RES0.

E3DSE, bits [39:36]

Delegated SError exceptions from EL3. Describes support for delegated SError injection from EL3.

The value of this field is an IMPLEMENTATION DEFINED choice of:

E3DSE	Meaning
0b0000	FEAT_E3DSE is not implemented.
0b0001	FEAT_E3DSE is implemented. The following are implemented: • Register fields SCR_EL3.DSE and SCR_EL3.EnDSE. • Registers VSESR_EL3 and VDISR_EL3.

All other values are reserved.

FEAT_E3DSE implements the functionality described by the value 0b0001.

Access to this field is RO.

Bits [35:32]

Reserved, RES0.

RMEGDI, bits [31:28]

RME Granular Data Isolation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RMEGDI	Meaning
0b0000	The Granular Data isolation GPI encodings are reserved.
0b0001	The Granular Data Isolation GPI encodings can be configured to be either reserved or No Access from this PE.

All other values are reserved.

FEAT_RME_GDI implements the functionality described by the value 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2H0, bits [27:24]

When FEAT_AA64EL2 is implemented:

Indicates support for programming HCR_EL2.E2H.

The value of this field is an IMPLEMENTATION DEFINED choice of:

E2H0	Meaning
0b0000	FEAT_E2H0 is implemented.
0b1110	FEAT_E2H0 is not implemented. HCR_EL2.NV1 is RES0.
0b1111	FEAT_E2H0 is not implemented.

All other values are reserved.

If FEAT_NV is not implemented, then the value 0b1110 is not permitted.

If FEAT_E2H0 is implemented and FEAT_VHE is not implemented, then HCR_EL2.E2H is RES0.

If FEAT_E2H0 is implemented and FEAT_VHE is implemented, then HCR_EL2.E2H can be programmed to 0 or 1.

If FEAT_E2H0 is not implemented then:

- FEAT_VHE is implemented.
- HCR_EL2.E2H is RES1 and behaves as though it is 1.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NV_frac, bits [23:20]

Indicates support for a subset of FEAT_NV and FEAT_NV2 behaviors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NV_frac	Meaning
0b0000	Support for FEAT_NV and FEAT_NV2 is described in ID_AA64MMFR2_EL1.NV.
0b0001	FEAT_NV and FEAT_NV2 are implemented, but all of the following apply: • ID_AA64MMFR2_EL1.NV is 0b0000. • Programming HCR_EL2.{NV, NV2} to {1, 0} behaves as {1, 1}.
0b0010	As 0b0001 and adds stateful bits in specified _EL1 registers for software usage under nested virtualization.

FEAT_NV2p1 implements the functionality indicated by the value 0b0010.

Access to this field is RO.

FGWTE3, bits [19:16]

Indicates support for Fine Grained Write Trap EL3 feature.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

FGWTE3	Meaning
0b0000	Fine Grained Write Trap EL3 is not supported.
0b0001	Fine Grained Write Trap EL3 is supported.

All other values are reserved.

FEAT_FGWTE3 implements the functionality identified by the value 0b0001.

Access to this field is RO.

HACDBS, bits [15:12]

Support for Hardware accelerator for cleaning Dirty state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HACDBS	Meaning
0b0000	Hardware accelerator for cleaning Dirty state is not supported.
0b0001	Hardware accelerator for cleaning Dirty state is supported.

All other values are reserved.

FEAT_HACDBS implements the functionality identified by the value 0b0001.

Access to this field is RO.

ASID2, bits [11:8]

Indicates support for concurrent use of two ASIDs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ASID2	Meaning
0b0000	FEAT_ASID2 is not implemented.
0b0001	FEAT_ASID2 is implemented.

All other values are reserved.

From Armv9.5, the value 0b0000 is not permitted.

Access to this field is RO.

EIESB, bits [7:4]

When FEAT_IESB is implemented:

Early Implicit Error Synchronization event. Indicates whether the implicit Error synchronization event inserted on taking an exception to ELx when SCTLR_ELx.IESB is 1 is inserted before or after the exception is taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

EIESB	Meaning
0b1111	When FEAT_IESB is implemented and enabled, SError exceptions are routed to EL3, and either FEAT_DoubleFault is not implemented or the Effective value of SCR_EL3.NMEA is 1, an implicit Error synchronization event might be inserted after an exception is taken to EL3, unless that exception is also an SError exception.
0b0000	FEAT_IESB is not implemented, or behavior is not described.
0b0001	When FEAT_IESB is implemented and enabled, SError exceptions are routed to EL3, and either FEAT_DoubleFault is not implemented or the Effective value of SCR_EL3.NMEA is 1, an implicit Error synchronization event is inserted before an exception taken to EL3, unless that exception is also an SError exception.
0b0010	As 0b0001, and also: • When FEAT_IESB is implemented and enabled, SError exceptions are routed to EL1, and either FEAT_DoubleFault2 is not implemented or the Effective value of SCTLR2_EL1.NMEA is 1, an implicit Error synchronization event is inserted before an exception taken to EL1, unless that exception is also an SError exception. • When FEAT_IESB is implemented and enabled, SError exceptions are routed to EL2, and either FEAT_DoubleFault2 is not implemented or the Effective value of SCTLR2_EL2.NMEA is 1, an implicit Error synchronization event is inserted before an exception taken to EL2, unless that exception is also an SError exception.
0b0011	When FEAT_IESB is implemented and enabled, an implicit Error synchronization event is inserted before taking an exception other than an SError exception, regardless of the Effective values of SCR_EL3.NMEA and SCTLR2_ELx.NMEA.

All other values are reserved.

This field describes the PE behavior on taking an exception to ELx when SCTLR_ELx.IESB is 1. This field does not describe the behavior when SCTLR_ELx.IESB is 0.

The behavior described by this field only applies for the conditions described above. For example, if ID_AA64MMFR4_EL1.EIESB reads as 0b0001, then it does not describe the behavior when SError exceptions are not routed to EL3, or when FEAT_DoubleFault is implemented and the Effective value of SCR_EL3.NMEA is 0.

Inserting the event before the exception is taken means that if the Error synchronization event causes an SError exception to become pending, and SError exceptions are not masked and not disabled, then the SError exception is taken in place of the original exception.

When FEAT_IESB is not implemented, the only permitted value of this field is 0b0000.

Access to this field is RO.

Otherwise:

Reserved, RES0.

PoPS, bits [3:0]

Support for the clean and invalidate to the Point of Physical Storage instructions:

- DC CIVAPS.
- If FEAT_MTE2 is implemented, DC CIGDVAPS.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PoPS	Meaning
0b0000	The System instructions to clean and invalidate to the Point of Physical Storage are not implemented.
0b0001	The specified System instructions to clean and invalidate to the Point of Physical Storage are implemented.

FEAT_PoPS implements the functionality described by the value 0b0001.

All other values are reserved.

Access to this field is RO.

Accessing ID_AA64MMFR4_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64MMFR4_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64MMFR4_EL1()) ||
     ↪ImpDefBool("ID_AA64MMFR4_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64MMFR4_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = ID_AA64MMFR4_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64MMFR4_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.92 ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0

The ID_AA64PFR0_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

The external register EDPFR gives information from this register.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64PFR0_EL1 are UNDEFINED.

Attributes

ID_AA64PFR0_EL1 is a 64-bit register.

Field descriptions

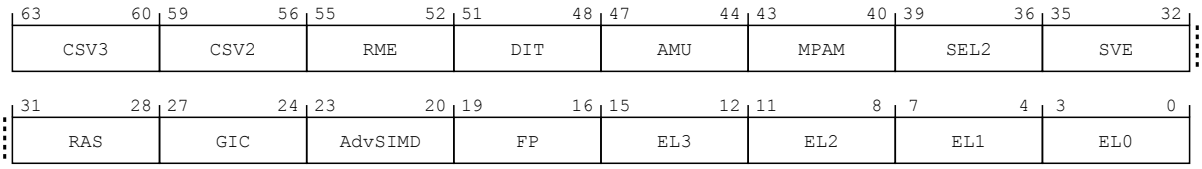

CSV3, bits [63:60]

Speculative use of faulting data.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV3	Meaning
0b0000	This PE does not disclose whether data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, can be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.
0b0001	Data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, cannot be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.

All other values are reserved.

FEAT_CSV3 implements the functionality identified by the value 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

From Armv8.5, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSV2, bits [59:56]

Speculative use of out of context prediction resources.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV2	Meaning
0b0000	The implementation does not disclose whether FEAT_CSV2 is implemented.
0b0001	FEAT_CSV2 is implemented, but FEAT_CSV2_2 and FEAT_CSV2_3 are not implemented. ID_AA64PFR1_EL1.CSV2_frac determines whether either or both of FEAT_CSV2_1p1 or FEAT_CSV2_1p2 are implemented.
0b0010	FEAT_CSV2_2 is implemented, but FEAT_CSV2_3 is not implemented.
0b0011	FEAT_CSV2_3 is implemented.

All other values are reserved.

FEAT_CSV2 implements the functionality identified by the value 0b0001.

FEAT_CSV2_2 implements the functionality identified by the value 0b0010.

FEAT_CSV2_3 implements the functionality identified by the feature 0b0011.

From Armv8.5, the value 0b0000 is not permitted.

Access to this field is RO.

RME, bits [55:52]

Realm Management Extension (RME).

The value of this field is an IMPLEMENTATION DEFINED choice of:

RME	Meaning
0b0000	Realm Management Extension not implemented.
0b0001	RMEv1 is implemented.
0b0010	As 0b0001, and adds support for the GPC2 Extension.
0b0011	As 0b0010, and adds support for the GPC3 Extension.

All other values are reserved.

FEAT_RME implements the functionality identified by the value 0b0001.

FEAT_RME_GPC2 implements the functionality identified by the value 0b0010.

FEAT_RME_GPC3 implements the functionality identified by the value 0b0011.

Access to this field is RO.

DIT, bits [51:48]

Data Independent Timing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DIT	Meaning
0b0000	AArch64 does not guarantee constant execution time of any instructions.
0b0001	AArch64 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

FEAT_DIT implements the functionality identified by the value 0b0001.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

AMU, bits [47:44]

Indicates support for Activity Monitors Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AMU	Meaning
0b0000	Activity Monitors Extension is not implemented.
0b0001	FEAT_AMUv1 is implemented.
0b0010	FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

FEAT_AMUv1 implements the functionality identified by the value 0b0001.

FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0000.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

MPAM, bits [43:40]

Indicates the major version number of support for the MPAM Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MPAM	Meaning
0b0000	The major version number of the MPAM extension is 0.
0b0001	The major version number of the MPAM extension is 1.

All other values are reserved.

When combined with the minor version number from ID_AA64PFR1_EL1.MPAM_frac, the “major.minor” version is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM Extension version	MPAM	MPAM_frac
Not implemented.	0b0000	0b0000
v0.1 is implemented.	0b0000	0b0001
v1.0 is implemented.	0b0001	0b0000
v1.1 is implemented.	0b0001	0b0001

For more information, see 'The Memory Partitioning and Monitoring (MPAM) Extension'.

Access to this field is RO.

SEL2, bits [39:36]

Secure EL2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SEL2	Meaning
0b0000	Secure EL2 is not implemented.
0b0001	Secure EL2 is implemented.

All other values are reserved.

FEAT_SEL2 implements the functionality identified by the value 0b0001.

From Armv8.4, if Secure state and EL2 are implemented, the value 0b0000 is not permitted.

From Armv8.4, if Secure state is not implemented, or if EL2 is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

SVE, bits [35:32]

Scalable Vector Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SVE	Meaning
0b0000	SVE architectural state and programmers' model are not implemented.
0b0001	SVE architectural state and programmers' model are implemented.

All other values are reserved.

FEAT_SVE implements the functionality identified by the value 0b0001.

If implemented, refer to ID_AA64ZFR0_EL1 for information about which SVE instructions are available.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RAS, bits [31:28]

RAS Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS	Meaning
`0b0000`	No RAS Extension.
`0b0001`	Support for the Reliability, Availability, and Serviceability Extension is implemented. The ESB instruction and the Error synchronization event are supported.
`0b0010`	As `0b0001`, and adds support for: • If EL3 is implemented, FEAT_DoubleFault. • Additional ERXMISC<m>_EL1 System registers. • Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.
`0b0011`	As `0b0010` and adds support for: • The error group status register, ERXGSR_EL1. • The SCR_EL3.TWERR write trap control for error record System registers. • Additional fields in ESR_ELx.ISS for error exceptions. Error records accessed through System registers conform to either RAS System Architecture v1.1 or RAS System Architecture v2.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value `0b0001`.

FEAT_DoubleFault implements the functionality identified by the value `0b0010`.

When ERRIDR_EL1.NUM is zero:

- If FEAT_DoubleFault is not implemented, then this field is permitted to have the value `0b0001`, regardless of whether FEAT_RASv1p1 or FEAT_RASv2 are implemented.
- If FEAT_DoubleFault is implemented, then this field is permitted to have the value `0b0010`, regardless of whether FEAT_RASv2 is implemented.

When ERRIDR_EL1.NUM is nonzero:

- FEAT_RASv1p1 implements the functionality identified by the value `0b0010`.
- FEAT_RASv2 implements the functionality identified by the value `0b0011`.

From Armv8.2, the value `0b0000` is not permitted.

From Armv8.4, if FEAT_DoubleFault is implemented or ERRIDR_EL1.NUM is nonzero, the value `0b0001` is not permitted.

Note

When the value of this field is `0b0001`, ID_AA64PFR1_EL1.RAS_frac indicates whether FEAT_RASv1p1 is implemented.

From Armv8.9, if ERRIDR_EL1.NUM is nonzero, the value `0b0010` is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

GIC, bits [27:24]

System register GIC CPU interface.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface system registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

Access to this field is RO.

AdvSIMD, bits [23:20]

Advanced SIMD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AdvSIMD	Meaning
0b0000	Advanced SIMD is implemented, including support for the following SISD and SIMD operations: • Integer byte, halfword, word and doubleword element operations. • Single-precision and double-precision floating-point arithmetic. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- 0b0000 in an implementation with Advanced SIMD support that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with Advanced SIMD support that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without Advanced SIMD support.

Access to this field is RO.

FP, bits [19:16]

Floating-point.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FP	Meaning
0b0000	Floating-point is implemented, and includes support for: • Single-precision and double-precision floating-point types. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- 0b0000 in an implementation with floating-point support that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with floating-point support that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without floating-point support.

Access to this field is RO.

EL3, bits [15:12]

EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented.
0b0001	EL3 can be executed in AArch64 state only.
0b0010	EL3 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

The value 0b0010 is not permitted in Armv9-A implementations.

Access to this field is RO.

EL2, bits [11:8]

EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented.
0b0001	EL2 can be executed in AArch64 state only.
0b0010	EL2 can be executed in either AArch64 or AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

The value `0b0010` is not permitted in Armv9-A implementations.

Access to this field is RO.

EL1, bits [7:4]

EL1 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL1	Meaning
`0b0001`	EL1 can be executed in AArch64 state only.
`0b0010`	EL1 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

The value `0b0010` is not permitted in Armv9-A implementations.

Access to this field is RO.

EL0, bits [3:0]

EL0 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL0	Meaning
`0b0001`	EL0 can be executed in AArch64 state only.
`0b0010`	EL0 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

Access to this field is RO.

Accessing ID_AA64PFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64PFR0_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b0000`	`0b0100`	`0b000`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64PFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64PFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64PFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.93 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64PFR1_EL1 are UNDEFINED.

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions

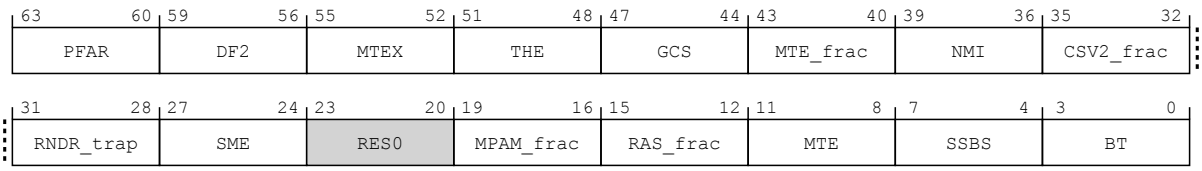

PFAR, bits [63:60]

Support for physical fault address registers, FEAT_PFAR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PFAR	Meaning
0b0000	FEAT_PFAR is not implemented.
0b0001	FEAT_PFAR is implemented. Includes support for the PFAR_ELx and, if EL3 is implemented, MFAR_EL3 registers.

All other values are reserved.

FEAT_PFAR implements the functionality identified by the value 0b0001.

Access to this field is RO.

DF2, bits [59:56]

Support for error exception routing extensions, FEAT_DoubleFault2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DF2	Meaning
0b0000	FEAT_DoubleFault2 is not implemented. **Note** This does not mean that FEAT_DoubleFault, as identified by ID_AA64PFR0_EL1.RAS >= 0b0010, is not implemented.
0b0001	FEAT_DoubleFault2 is implemented. As ID_AA64PFR0_EL1.RAS == 0b0010, and also includes support for routing error exceptions: • Traps for masked error exceptions, HCRX_EL2.TMEA and SCR_EL3.TMEA. • Additional controls for masking SError exceptions, SCTLR2_EL1.NMEA, and SCTLR2_EL2.NMEA. • Additional controls for taking external aborts to the SError exception vector, SCTLR2_EL1.EASE and SCTLR2_EL2.EASE.

All other values are reserved.

FEAT_DoubleFault2 implements the functionality identified by the value 0b0001.

Access to this field is RO.

MTEX, bits [55:52]

Support for additional MTE tag checking modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTEX	Meaning
0b0000	Canonical Tag checking and Memory tagging with Address tagging disabled are not supported.
0b0001	The following additional tag checking modes for MTE are supported: • Canonical Tag checking. • Memory tagging with Address tagging disabled.

All other values are reserved.

This field is valid only if ID_AA64PFR1_EL1.MTE >= 0b0010.

FEAT_MTE_NO_ADDRESS_TAGS and FEAT_MTE_CANONICAL_TAGS implement the functionality identified by the value 0b0001.

If FEAT_MTE2 is not implemented, the value 0b0001 is not permitted.

From Armv8.9, if FEAT_MTE2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

THE, bits [51:48]

Support for Translation Hardening Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

THE	Meaning
0b0000	Translation Hardening Extension is not implemented.
0b0001	The RCW and RCWS instructions, their associated registers and traps are supported. If EL2 is implemented, the AssuredOnly check, TopLevel check, and their associated controls are implemented. If EL2 and FEAT_GCS are implemented, VTCR_EL2.GCSH is implemented.

All other values are reserved.

FEAT_THE implements the functionality identified by the value 0b0001.

Access to this field is RO.

GCS, bits [47:44]

Support for Guarded Control Stack.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GCS	Meaning
0b0000	Guarded Control Stack is not implemented.
0b0001	Guarded Control Stack is implemented.

All other values are reserved.

FEAT_GCS implements the functionality identified by the value 0b0001.

Access to this field is RO.

MTE_frac, bits [43:40]

Support for Asynchronous reporting of a Tag Check Fault.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTE_frac	Meaning
0b0000	Asynchronous reporting of a Tag Check Fault is supported.
0b1111	Asynchronous reporting of a Tag Check Fault is not supported.

All other values are reserved.

This field is valid only if ID_AA64PFR1_EL1.MTE >= 0b0010.

FEAT_MTE_ASYNC implements the functionality identified by the value 0b0000.

If FEAT_MTE_ASYM_FAULT is implemented this field must be 0b0000.

Access to this field is RO.

NMI, bits [39:36]

Non-maskable Interrupt. Indicates support for Non-maskable interrupts.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NMI	Meaning
0b0000	SCTLR_ELx.{SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are not supported.
0b0001	SCTLR_ELx.{SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are supported.

All other values are reserved.

FEAT_NMI implements the functionality identified by the value 0b0001.

From Armv8.8, the value 0b0000 is not permitted.

Access to this field is RO.

CSV2_frac, bits [35:32]

CSV2 fractional field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV2_frac	Meaning
0b0000	Either ID_AA64PFR0_EL1.CSV2 is not 0b0001, or the implementation does not disclose whether FEAT_CSV2_1p1 is implemented. FEAT_CSV2_1p2 is not implemented.
0b0001	FEAT_CSV2_1p1 is implemented, but FEAT_CSV2_1p2 is not implemented.
0b0010	FEAT_CSV2_1p2 is implemented.

All other values are reserved.

FEAT_CSV2_1p1 implements the functionality identified by the value 0b0001.

FEAT_CSV2_1p2 implements the functionality identified by the value 0b0010.

From Armv8.0, the permitted values are 0b0000, 0b0001, and 0b0010.

The values 0b0001 and 0b0010 are permitted only when ID_AA64PFR0_EL1.CSV2 is 0b0001.

Access to this field is RO.

RNDR_trap, bits [31:28]

Random Number trap to EL3 field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RNDR_trap	Meaning
0b0000	Trapping of RNDR and RNDRRS to EL3 is not supported.
0b0001	Trapping of RNDR and RNDRRS to EL3 is supported. SCR_EL3.TRNDR is present.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RNG_TRAP implements the functionality identified by the value `0b0001`.

Access to this field is RO.

SME, bits [27:24]

Scalable Matrix Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
`0b0000`	SME architectural state and programmers' model are not implemented.
`0b0001`	SME architectural state and programmers' model are implemented.
`0b0010`	As `0b0001`, plus the SME2 ZT0 register.

All other values are reserved.

FEAT_SME implements the functionality identified by the value `0b0001`.

FEAT_SME2 implements the functionality identified by the value `0b0010`.

From Armv9.2, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

If implemented, refer to ID_AA64SMFR0_EL1 and ID_AA64ZFR0_EL1 for information about which SME and SVE instructions are available.

Access to this field is RO.

Bits [23:20]

Reserved, RES0.

MPAM_frac, bits [19:16]

Indicates the minor version number of support for the MPAM Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MPAM_frac	Meaning
`0b0000`	The minor version number of the MPAM extension is 0.
`0b0001`	The minor version number of the MPAM extension is 1.

All other values are reserved.

When combined with the major version number from ID_AA64PFR0_EL1.MPAM, The combined "major.minor" version is:

MPAM Extension version	MPAM	MPAM_frac
Not implemented.	`0b0000`	`0b0000`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM Extension version	MPAM	MPAM_frac
v0.1 is implemented.	0b0000	0b0001
v1.0 is implemented.	0b0001	0b0000
v1.1 is implemented.	0b0001	0b0001

For more information, see 'The Memory Partitioning and Monitoring (MPAM) Extension'.

Access to this field is RO.

RAS_frac, bits [15:12]

RAS Extension fractional field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS_frac	Meaning
0b0000	If ID_AA64PFR0_EL1.RAS == 0b0001, support for the Reliability, Availability, and Serviceability Extension is implemented.
0b0001	If ID_AA64PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for: • Additional ERXMISC<m>_EL1 System registers. • Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS, and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0000.

When ERRIDR_EL1.NUM is zero, this field is permitted to have the value 0b0000, regardless of whether FEAT_RASv1p1 is implemented.

When ERRIDR_EL1.NUM is nonzero, FEAT_RASv1p1 implements the functionality identified by the value 0b0001.

This field is valid only if ID_AA64PFR0_EL1.RAS == 0b0001.

Access to this field is RO.

MTE, bits [11:8]

Support for the Memory Tagging Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTE	Meaning
0b0000	Memory Tagging Extension is not implemented.
0b0001	Instruction-only Memory Tagging Extension is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTE	Meaning
0b0010	As 0b0001, and adds: • Support for in-memory Allocation tags. • Support for synchronous tag checking. • Optional support for Asynchronous reporting of a Tag Check Fault, identified as FEAT_MTE_ASYNC. Support for FEAT_MTE_ASYNC is indicated by ID_AA64PFR1_EL1.MTE_frac.
0b0011	As 0b0010, except that support for FEAT_MTE_ASYNC is mandatory, and adds support for Asymmetric Tag Check Fault handling, identified as FEAT_MTE_ASYM_FAULT.

All other values are reserved.

FEAT_MTE implements the functionality identified by the value 0b0001.

FEAT_MTE2 implements the functionality identified by the value 0b0010.

FEAT_MTE3 implements the functionality identified by the value 0b0011.

From Armv8.7, when the value of this field is >= 0b0010, ID_AA64PFR2_EL1.MTEPERM indicates support for FEAT_MTE_PERM.

From Armv8.7, when the value of this field is >= 0b0010, the following fields indicate support for FEAT_MTE4:

- ID_AA64PFR1_EL1.MTEX
- ID_AA64PFR2_EL1.MTEFAR
- ID_AA64PFR2_EL1.MTESTOREONLY

Access to this field is RO.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SSBS	Meaning
0b0000	AArch64 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001	AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.
0b0010	As 0b0001, and adds the MSR and MRS instructions to directly read and write the PSTATE.SSBS field.

All other values are reserved.

FEAT_SSBS implements the functionality identified by the value 0b0001.

FEAT_SSBS2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

BT, bits [3:0]

Branch Target Identification mechanism support in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning
0b0000	The Branch Target Identification mechanism is not implemented.
0b0001	The Branch Target Identification mechanism is implemented.

All other values are reserved.

FEAT_BTI implements the functionality identified by the value 0b0001.

From Armv8.5, the value 0b0000 is not permitted.

Access to this field is RO.

Accessing ID_AA64PFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64PFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64PFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64PFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64PFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.94 ID_AA64PFR2_EL1, AArch64 Processor Feature Register 2

The ID_AA64PFR2_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AA64PFR2_EL1 are UNDEFINED.

Attributes

ID_AA64PFR2_EL1 is a 64-bit register.

Field descriptions

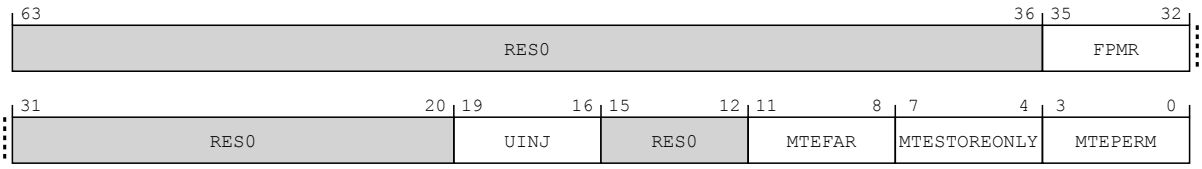

Bits [63:36]

Reserved, RES0.

FPMR, bits [35:32]

Indicates support for FPMR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPMR	Meaning
0b0000	FPMR is not implemented.
0b0001	FPMR is implemented.

All other values are reserved.

FEAT_FPMR implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [31:20]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UINJ, bits [19:16]

Support for software injection of Undefined Instruction exceptions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

UINJ	Meaning
0b0000	Software injection of Undefined Instruction exceptions is not implemented.
0b0001	Software injection of Undefined Instruction exceptions is implemented.

FEAT_UINJ implements the functionality identified by the value 0b0001.

From Armv9.6, the value 0b0000 is not permitted.

Access to this field is RO.

Bits [15:12]

Reserved, RES0.

MTEFAR, bits [11:8]

Indicates whether FAR_ELx[63:60] are UNKNOWN on a synchronous exception due to a Tag Check Fault.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTEFAR	Meaning
0b0000	On a synchronous exception due to a Tag Check Fault, FAR_ELx[63:60] are UNKNOWN.
0b0001	On a synchronous exception due to a Tag Check Fault, FAR_ELx[63:60] are not UNKNOWN.

All other values are reserved.

FEAT_MTE_TAGGED_FAR implements the functionality identified by the value 0b0001.

If FEAT_MTE2 is not implemented, the value 0b0001 is not permitted.

From Armv8.9, if FEAT_MTE2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

MTESTOREONLY, bits [7:4]

Support for Store-only Tag checking.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTESTOREONLY	Meaning
0b0000	Store-only Tag checking is not supported.
0b0001	Store-only Tag checking is supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_MTE_STORE_ONLY implements the functionality identified by the value `0b0001`.

If FEAT_MTE2 is not implemented, the value `0b0001` is not permitted.

From Armv8.9, if FEAT_MTE2 is implemented, the value `0b0000` is not permitted.

Access to this field is RO.

MTEPERM, bits [3:0]

Support for Allocation tag access permissions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTEPERM	Meaning
`0b0000`	Allocation tag access permissions are not supported.
`0b0001`	Allocation tag access permissions are supported. **Note** NoTagAccess is supported at stage 2 of translation only.

All other values are reserved.

FEAT_MTE_PERM implements the functionality identified by the value `0b0001`

If FEAT_MTE2 is not implemented, the value `0b0001` is not permitted.

From Armv8.9, if FEAT_MTE2 is implemented, the value `0b0000` is not permitted.

Access to this field is RO.

Accessing ID_AA64PFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64PFR2_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b0000`	`0b0100`	`0b010`

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64PFR2_EL1()) ||
         ↪ImpDefBool("ID_AA64PFR2_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64PFR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
         ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64PFR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64PFR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.95 ID_AA64SMFR0_EL1, SME Feature ID Register 0

The ID_AA64SMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented features of the AArch64 Scalable Matrix Extension.

The fields in this register do not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64SMFR0_EL1 and ID_AA64FPFR0_EL1.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_AA64SMFR0_EL1 is a 64-bit register.

Field descriptions

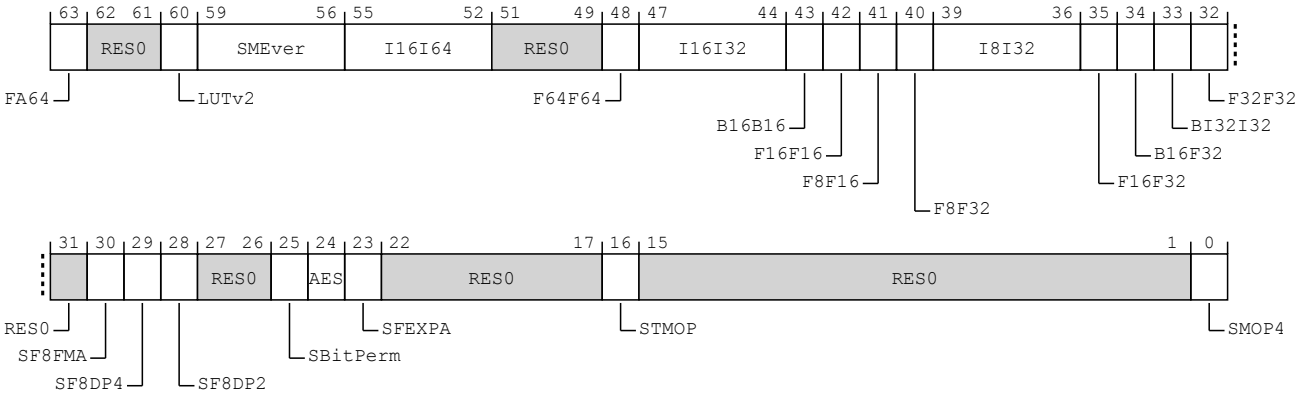

FA64, bit [63]

Indicates support at each Exception Level for execution of the full AArch64 Advanced SIMD and SVE instruction sets when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FA64	Meaning
0b0	Only those AArch64 instructions defined as being legal can be executed in streaming SVE mode.
0b1	All implemented AArch64 instructions are legal for execution in Streaming SVE mode, when enabled by SMCR_EL1.FA64, SMCR_EL2.FA64, and SMCR_EL3.FA64.

FEAT_SME_FA64 implements the functionality identified by the value 1.

Access to this field is RO.

Bits [62:61]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LUTv2, bit [60]

Indicates support for the following additional variants of SME2 lookup table `LUTI4` and `MOVT` instructions:

- A `LUTI4` instruction with 8-bit result elements, two consecutively numbered source vectors, and four consecutively numbered destination vectors.
- If FEAT_SME2p1 is implemented, a `LUTI4` instruction with 8-bit result elements, two consecutively numbered source vectors, and four destination vectors with strided register numbering.
- A `MOVT` instruction that copies a single Z source vector to ZT0.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LUTv2	Meaning
`0b0`	The specified instructions are not implemented by this control.
`0b1`	The specified instructions are implemented.

All other values are reserved.

FEAT_SME_LUTv2 implements the functionality identified by the value 1.

Access to this field is RO.

SMEver, bits [59:56]

When FEAT_SME is implemented:

Indicates support for SME instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SMEver	Meaning
`0b0000`	The mandatory SME instructions are implemented.
`0b0001`	As `0b0000`, and adds the mandatory SME2 instructions.
`0b0010`	As `0b0001`, and adds the mandatory SME2.1 instructions.
`0b0011`	As `0b0010`, and adds the mandatory SME2.2 instructions.

All other values are reserved.

FEAT_SME2 implements the functionality identified by the value `0b0001`.

FEAT_SME2p1 implements the functionality identified by the value `0b0010`.

From Armv9.4, the value `0b0001` is not permitted.

FEAT_SME2p2 implements the functionality identified by `0b0011`.

From Armv9.6, the values `0b0000` and `0b0010` are not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

I16I64, bits [55:52]

Indicates support for the following SME instructions that accumulate into 64-bit integer elements in the ZA array:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The variants of the ADDHA, ADDVA, SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate into 64-bit integer tiles.
- When FEAT_SME2 is implemented, the variants of the ADD, ADDA, SDOT, SMLALL, SMLSLL, SUB, SUBA, SVDOT, UDOT, UMLALL, UMLSLL, and UVDOT instructions that accumulate into 64-bit integer elements in ZA array vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

I16I64	Meaning
0b0000	The specified instructions are not implemented by this control.
0b1111	The specified instructions are implemented.

All other values are reserved.

FEAT_SME_I16I64 implements the functionality identified by the value 0b1111.

Access to this field is RO.

Bits [51:49]

Reserved, RES0.

F64F64, bit [48]

Indicates support for the following SME instructions that accumulate into double-precision floating-point elements in the ZA array:

- The variants of the FMOPA and FMOPS instructions that accumulate into double-precision tiles.
- When FEAT_SME2 is implemented, the variants of the FADD, FMLA, FMLS, and FSUB instructions that accumulate into double-precision elements in ZA array vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F64F64	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented .

FEAT_SME_F64F64 implements the functionality identified by the value 1.

Access to this field is RO.

I16I32, bits [47:44]

When FEAT_SME2 is implemented:

Indicates support for SME2 SMOPA (2-way), SMOPS (2-way), UMOPA (2-way), and UMOPS (2-way) instructions that accumulate 16-bit outer products into 32-bit integer tiles.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I16I32	Meaning
0b0000	The specified instructions are not implemented by this control.
0b0101	The specified instructions are implemented.

All other values are reserved.

If FEAT_SME2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

B16B16, bit [43]

Indicates support for the SME ZA-targeting non-widening BFloat16 BFADD, BFMLA, BFMLS, BFMOPA, BFMOPS, and BFSUB instructions with BFloat16 operands and results.

The value of this field is an IMPLEMENTATION DEFINED choice of:

B16B16	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

FEAT_SME_B16B16 implements the functionality identified by the value 1.

Access to this field is RO.

F16F16, bit [42]

Indicates support for the following SME2 half-precision floating-point instructions:

- FMOPA and FMOPS instructions that accumulate half-precision outer-products into half-precision tiles.
- Multi-vector FADD, FMLA, FMLS, and FSUB instructions with half-precision operands and results.
- Multi-vector FCVT and FCVTL instructions that convert half-precision inputs to single-precision results.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F16F16	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

FEAT_SME_F16F16 implements the functionality identified by the value 1.

Access to this field is RO.

F8F16, bit [41]

Indicates support for the following SME2 instructions:

- The ZA-targeting FP8 instructions FDOT (2-way), FMLAL, FMOPA (2-way), and FVDOT that accumulate into half-precision floating-point elements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ZA-targeting non-widening half-precision FADD and FSUB instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8F16	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

FEAT_SME_F8F16 implements the functionality identified by the value 1.

Access to this field is RO.

F8F32, bit [40]

Indicates support for the SME2 ZA-targeting FP8 FDOT (4-way), FMLALL, FMOPA (4-way), FVDOTB, and FVDOTT instructions that accumulate into single-precision floating-point elements.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F8F32	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

FEAT_SME_F8F32 implements the functionality identified by the value 1.

Access to this field is RO.

I8I32, bits [39:36]

When FEAT_SME is implemented:

Indicates support for SME SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate 8-bit outer products into 32-bit tiles

The value of this field is an IMPLEMENTATION DEFINED choice of:

I8I32	Meaning
0b0000	The specified instructions are not implemented by this control.
0b1111	The specified instructions are implemented.

All other values are reserved.

If FEAT_SME is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

F16F32, bit [35]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SME is implemented:

Indicates support for SME FMOPA and FMOPS instructions that accumulate half-precision floating-point outer products into single-precision floating-point tiles.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F16F32	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

If FEAT_SME is implemented, the value 0 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

B16F32, bit [34]

When FEAT_SME is implemented:

Indicates support for SME BFMOPA and BFMOPS instructions that accumulate BFloat16 outer products into single-precision floating-point tiles.

The value of this field is an IMPLEMENTATION DEFINED choice of:

B16F32	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

If FEAT_SME is implemented, the value 0 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

BI32I32, bit [33]

When FEAT_SME2 is implemented:

Indicates support for SME BMOPA and BMOPS instructions that accumulate thirty-two 1-bit binary outer products into 32-bit integer tiles.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BI32I32	Meaning
0b0	The specified instructions are not implemented by this control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	The specified instructions are implemented.

If FEAT_SME2 is implemented, the value 0 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

F32F32, bit [32]

When FEAT_SME is implemented:

Indicates support for SME FMOPA and FMOPS instructions that accumulate single-precision floating-point outer products into single-precision floating-point tiles.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F32F32	Meaning
0b0	The specified instructions are not implemented by this control.
0b1	The specified instructions are implemented.

If FEAT_SME is implemented, the value 0 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bit [31]

Reserved, RES0.

SF8FMA, bit [30]

Indicates support for the SVE2 FP8 to single-precision and half-precision multiply-accumulate FMLALB, FMLALT, FMLALLBB, FMLALLBT, FMLALLTB, and FMLALLTT instructions when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SF8FMA	Meaning
0b0	This control has no effect on Streaming SVE mode behavior.
0b1	The specified instructions are supported in Streaming SVE mode.

Note

Other features may support some of the specified instructions in Non-streaming SVE mode.

If FEAT_SME2 and FEAT_FP8FMA are implemented, the value 0 is not permitted.

FEAT_SSVE_FP8FMA implements the functionality identified by the value 1.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SF8DP4, bit [29]

Indicates support for the SVE2 FP8 to single-precision 4-way dot product `FDOT` (4-way) instructions when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SF8DP4	Meaning
`0b0`	This control has no effect on Streaming SVE mode behavior.
`0b1`	The specified instructions are supported in Streaming SVE mode.

Note

Other features may support some of the specified instructions in Non-streaming SVE mode.

If FEAT_SME2 and FEAT_FP8DOT4 are implemented, the value 0 is not permitted.

FEAT_SSVE_FP8DOT4 implements the functionality identified by the value 1.

Access to this field is RO.

SF8DP2, bit [28]

Indicates support for the SVE2 FP8 to half-precision 2-way dot product `FDOT` (2-way) instructions when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SF8DP2	Meaning
`0b0`	This control has no effect on Streaming SVE mode behavior.
`0b1`	The specified instructions are supported in Streaming SVE mode.

Note

Other features may support some of the specified instructions in Non-streaming SVE mode.

If FEAT_SME2 and FEAT_FP8DOT2 are implemented, the value 0 is not permitted.

FEAT_SSVE_FP8DOT2 implements the functionality identified by the value 1.

Access to this field is RO.

Bits [27:26]

Reserved, RES0.

SBitPerm, bit [25]

Indicates support for the SVE bit permute instructions identified as implemented by ID_AA64ZFR0_EL1.BitPerm, when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SBitPerm	Meaning
0b0	This control has no effect on Streaming SVE mode behavior.
0b1	The specified instructions are implemented when the PE is in Streaming SVE mode.

Note

Other features may support some of the specified instructions in Non-streaming SVE mode.

FEAT_SSVE_BitPerm implements the functionality identified by the value 1.

Access to this field is RO.

AES, bit [24]

Indicates support for the SVE AES and 128-bit polynomial multiply long instructions identified as implemented by ID_AA64ZFR0_EL1.AES, when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AES	Meaning
0b0	This control has no effect on Streaming SVE mode behavior.
0b1	The specified instructions are supported when the PE is in Streaming SVE mode.

Note

Other features may support some of the specified instructions in Non-streaming SVE mode.

FEAT_SSVE_AES implements the functionality identified by the value 1.

Access to this field is RO.

SFEXPA, bit [23]

Indicates support for the SVE FEXPA instruction when the PE is in Streaming SVE mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SFEXPA	Meaning
0b0	This control has no effect on Streaming SVE mode behavior.
0b1	The specified instruction is supported when the PE is in Streaming SVE mode.

Note

Other features may support some of the specified instructions in Non-streaming SVE mode.

FEAT_SSVE_FEXPA implements the functionality identified by the value 1.

Access to this field is RO.

Bits [22:17]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STMOP, bit [16]

Indicates support for the following SME Structured sparsity outer product instructions:

- `BFTMOPA` (non-widening), if FEAT_SME_B16B16 is implemented.
- `BFTMOPA` (widening, BF16 to FP32).
- `FTMOPA` (non-widening, FP16), if FEAT_SME_F16F16 is implemented.
- `FTMOPA` (non-widening, FP32).
- `FTMOPA` (widening, 2-way, FP16 to FP32).
- `FTMOPA` (widening, 2-way, FP8 to FP16), if FEAT_SME_F8F16 is implemented.
- `FTMOPA` (widening, 4-way, FP8 to FP32) if FEAT_SME_F8F32 is implemented.
- `STMOPA`, `SUTMOPA`, `USTMOPA`, `UTMOPA` (4-way, Int8 to Int32). `STMOPA`, `UTMOPA` (2-way, Int16 to Int32).

The value of this field is an IMPLEMENTATION DEFINED choice of:

STMOP	Meaning
`0b0`	The specified instructions are not implemented by this control.
`0b1`	The specified instructions are implemented.

FEAT_SME_TMOP implements the functionality identified by the value 1.

Access to this field is RO.

Bits [15:1]

Reserved, RES0.

SMOP4, bit [0]

Indicates support for the following SME Quarter-tile outer product instructions:

- `BFMOP4A`, `BFMOP4S` (non-widening, BF16), if FEAT_SME_B16B16 is implemented.
- `BFMOP4A`, `BFMOP4S` (widening, 2-way, BF16 to FP32).
- `FMOP4A`, `FMOP4S` (non-widening, FP16), if FEAT_SME_F16F16 is implemented.
- `FMOP4A`, `FMOP4S` (non-widening, FP32).
- `FMOP4A`, `FMOP4S` (non-widening, FP64), if FEAT_SME_F64F64 is implemented.
- `FMOP4A`, `FMOP4S` (widening, 2-way, FP16 to FP32).
- `FMOP4A`(widening, 2-way, FP8 to FP16), if FEAT_SME_F8F16 is implemented.
- `FMOP4A`(widening, 4-way, FP8 to FP32) instruction, if FEAT_SME_F8F32 is implemented.
- `SMOP4A`, `SMOP4S`, `SUMOP4A`, `SUMOP4S`, `UMOP4A`, `UMOP4S`, `USMOP4A`, `USMOP4S` (4-way, Int8 to Int32).
- `SMOP4A`, `SMOP4S`, `SUMOP4A`, `SUMOP4S`, `UMOP4A`, `UMOP4S`, `USMOP4A`, `USMOP4S` (4-way, Int16 to Int64), if FEAT_SME_I16I64 is implemented.
- `SMOP4A`, `SMOP4S`, `UMOP4A`, `UMOP4S` (2-way, Int16 to Int32).

The value of this field is an IMPLEMENTATION DEFINED choice of:

SMOP4	Meaning
`0b0`	The specified instructions are not implemented by this control.
`0b1`	The specified instructions are implemented.

FEAT_SME_MOP4 implements the functionality identified by the value 1.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ID_AA64SMFR0_EL1

This register is read-only and can be accessed from EL1 and higher.

This register is only accessible from the AArch64 state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64SMFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64SMFR0_EL1()) ||
     ↪ImpDefBool("ID_AA64SMFR0_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64SMFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64SMFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64SMFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.96 ID_AA64ZFR0_EL1, SVE Feature ID Register 0

The ID_AA64ZFR0_EL1 characteristics are:

Purpose

Provides additional information about the implemented features of the AArch64 Scalable Vector Extension instruction set, when FEAT_SVE or FEAT_SME is implemented.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.
If FEAT_SME is implemented and FEAT_SVE is not implemented, then SVE instructions can only be executed when the PE is in Streaming SVE mode and the instructions are legal for execution in Streaming SVE mode.

Attributes

ID_AA64ZFR0_EL1 is a 64-bit register.

Field descriptions

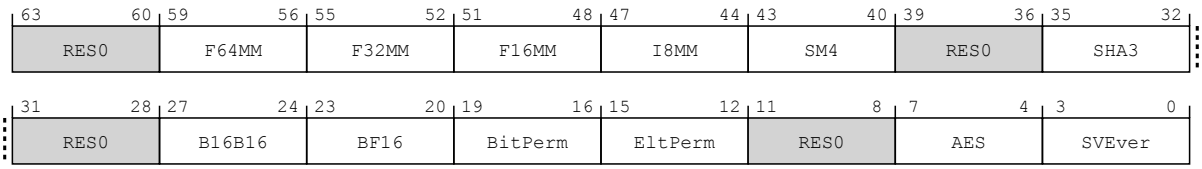

Bits [63:60]

Reserved, RES0.

F64MM, bits [59:56]

Indicates support for the SVE double-precision floating-point matrix multiply-accumulate instruction `FMMLA`, the `LD1RO*` instructions, the 128-bit element variants of the SVE `TRN1`, `TRN2`, `UZP1`, `UZP2`, `ZIP1`, and `ZIP2` instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F64MM	Meaning
`0b0000`	This field does not indicate support for the specified instructions.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_F64MM implements the functionality identified by `0b0001`.

The instructions described by nonzero values of this field might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F32MM, bits [55:52]

Indicates support for the SVE single-precision floating-point matrix multiply-accumulate instruction `FMMLA`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F32MM	Meaning
`0b0000`	This field does not indicate support for the specified instruction.
`0b0001`	The specified instruction is implemented.

All other values are reserved.

FEAT_F32MM implements the functionality identified by `0b0001`.

The instructions described by nonzero values of this field might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

F16MM, bits [51:48]

Indicates support for the SVE half-precision floating-point matrix multiply-accumulate to single-precision instruction `FMMLA` (widening, FP16 to FP32).

The value of this field is an IMPLEMENTATION DEFINED choice of:

F16MM	Meaning
`0b0000`	This field does not indicate support for the specified instruction.
`0b0001`	The specified instruction is implemented.

FEAT_SVE_F16F32MM implements the functionality identified by `0b0001`.

The instructions described by nonzero values of this field might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

I8MM, bits [47:44]

Indicates support for the following SVE 8-bit integer sum-of-products and accumulate to 32-bit integer instructions `SMMLA`, `SUDOT`, `UMMLA`, `USMMLA`, and `USDOT`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

I8MM	Meaning
`0b0000`	This field does not indicate support for the specified instructions.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_I8MM implements the functionality identified by `0b0001`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ISAR1_EL1.I8MM.

From Armv8.6, if SVE is implemented, the value `0b0000` is not permitted.

SVE `SMMLA`, `UMMLA`, and `USMMLA` instructions might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

SM4, bits [43:40]

Indicates support for SVE SM4 instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SM4	Meaning
`0b0000`	This field does not indicate support for the specified instructions.
`0b0001`	SVE `SM4E` and `SM4EKEY` instructions are implemented.

All other values are reserved.

FEAT_SVE_SM4 implements the functionality identified by `0b0001`.

The instructions described by nonzero values of this field might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

Bits [39:36]

Reserved, RES0.

SHA3, bits [35:32]

Indicates support for the SVE SHA3 instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA3	Meaning
`0b0000`	This field does not indicate support for the specified instructions.
`0b0001`	SVE `RAX1` instruction is implemented.

All other values are reserved.

FEAT_SVE_SHA3 implements the functionality identified by `0b0001`.

If FEAT_SME2p1 is not implemented, then the instructions described by nonzero values of this field might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

Bits [31:28]

Reserved, RES0.

B16B16, bits [27:24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates support for SVE non-widening BFloat16 instructions and SME multi-vector Z-targeting non-widening BFloat16 instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

B16B16	Meaning
0b0000	This field does not indicate support for the specified instructions.
0b0001	The following non-widening BFloat16 instructions are implemented: • SVE instructions: BFADD, BFCLAMP, BFMAX, BFMAXNM, BFMIN, BFMINNM, BFMLA, BFMLS, BFMUL, and BFSUB. • If FEAT_SME2 is implemented, SME multi-vector Z-targeting instructions: BFCLAMP, BFMAX, BFMAXNM, BFMIN, and BFMINNM.
0b0010	As 0b0001, and adds the following non-widening BFloat16 instructions: • SVE instruction BFSCALE. • If FEAT_SME2 is implemented, SME multi-vector Z-targeting instructions BFMUL and BFSCALE.

FEAT_SVE_B16B16 implements the functionality identified by 0b0001.

FEAT_SVE_BFSCALE implements the functionality identified by 0b0010.

Access to this field is RO.

BF16, bits [23:20]

Indicates support for SVE BFloat16 instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BF16	Meaning
0b0000	SVE BFloat16 instructions are not implemented.
0b0001	SVE BFCVT, BFCVTNT, BFDOT, BFMLALB, BFMLALT, and BFMMLA instructions are implemented.
0b0010	As 0b0001, but the FPCR.EBF field is also supported.

All other values are reserved.

FEAT_BF16 implements the functionality identified by 0b0001.

FEAT_EBF16 implements the functionality identified by 0b0010.

This field must return the same value as ID_AA64ISAR1_EL1.BF16.

SVE BFMMLA instructions might not be legal when the PE is in Streaming SVE mode.

From Armv8.6 and Armv9.1, the value 0b0000 is not permitted.

Access to this field is RO.

BitPerm, bits [19:16]

Indicates support for the SVE bit permute instructions BDEP, BEXT, and BGRP.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BitPerm	Meaning
0b0000	This field does not indicate support for the specified instructions.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_SVE_BitPerm implements the functionality identified by 0b0001 when the PE is not in Streaming SVE mode.

FEAT_SSVE_BitPerm which is identified in ID_AA64SMFR0_EL1.SBitPerm, indicates whether the FEAT_SVE_BitPerm functionality is implemented when the PE is in Streaming SVE mode.

Access to this field is RO.

EltPerm, bits [15:12]

When FEAT_SVE2p2 is implemented or FEAT_SME2p2 is implemented:

If FEAT_SVE2p2 is implemented, the following SVE instructions are implemented when the PE is not in Streaming SVE mode:

- 8-bit and 16-bit element COMPACT.
- 8-bit, 16-bit, 32-bit, and 64-bit element EXPAND.

If FEAT_SME2p2 is implemented, the following SVE instructions are implemented when the PE is in Streaming SVE mode:

- 8-bit, 16-bit, 32-bit, and 64-bit element COMPACT and EXPAND.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EltPerm	Meaning
0b0000	This field does not indicate support for the specified instructions.
0b0001	The specified instructions are implemented.

If FEAT_SVE2p2 or FEAT_SME2p2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [11:8]

Reserved, RES0.

AES, bits [7:4]

Indicates support for SVE Advanced Encryption Standard instructions and 128-bit polynomial multiply long instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AES	Meaning
0b0000	This field does not indicate support for the specified instructions.
0b0001	The following instructions are implemented: • SVE single-vector AESD, AESE, AESIMC, and AESMC instructions. • The 128-bit destination element variant of the SVE single-vector PMULLB and PMULLT instructions.
0b0010	As 0b0001.
0b0011	As 0b0010, and adds the following SVE instructions: • Multi-vector AESD, AESDIMC, AESE and AESEMC instructions. • Multi-vector 128-bit destination element PMULL and PMLAL instructions.

All other values are reserved.

FEAT_SVE_AES implements the functionality identified by the value 0b0001.

FEAT_SVE_PMULL128 implements the functionality identified by the value 0b0010.

FEAT_SVE_AES2 implements the functionality identified by 0b0011.

If FEAT_SSVE_AES is not implemented, then the instructions described by nonzero values of this field might not be legal when the PE is in Streaming SVE mode.

Access to this field is RO.

SVEver, bits [3:0]

When FEAT_SVE is implemented or FEAT_SME is implemented:

Indicates support for SVE instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SVEver	Meaning
0b0000	The SVE instructions are implemented.
0b0001	As 0b0000, and adds the mandatory SVE2 instructions.
0b0010	As 0b0001, and adds the mandatory SVE2.1 instructions.
0b0011	As 0b0010, and adds the mandatory SVE2.2 instructions.

All other values are reserved.

FEAT_SVE2 implements the functionality identified by 0b0001 when the PE is not in Streaming SVE mode.

FEAT_SME implements the functionality identified by 0b0001 when the PE is in Streaming SVE mode.

FEAT_SME2p1 implements the functionality identified by 0b0010 when the PE is in Streaming SVE mode.

FEAT_SVE2p1 implements the functionality identified by 0b0010 when the PE is not in Streaming SVE mode.

FEAT_SVE2p2 implements the functionality identified by 0b0011 when the PE is not in Streaming SVE mode.

FEAT_SME2p2 implements the functionality identified by 0b0011 when the PE is in Streaming SVE mode.

From Armv9, the value 0b0000 is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv9.4, the value `0b0001` is not permitted.

From Armv9.6, the value `0b0010` is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing ID_AA64ZFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64ZFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64ZFR0_EL1()) ||
     ↪ImpDefBool("ID_AA64ZFR0_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ZFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AA64ZFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AA64ZFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.97 ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0

The ID_AFR0_EL1 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_AFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_AFR0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_AFR0_EL1 are UNDEFINED.

Attributes

ID_AFR0_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

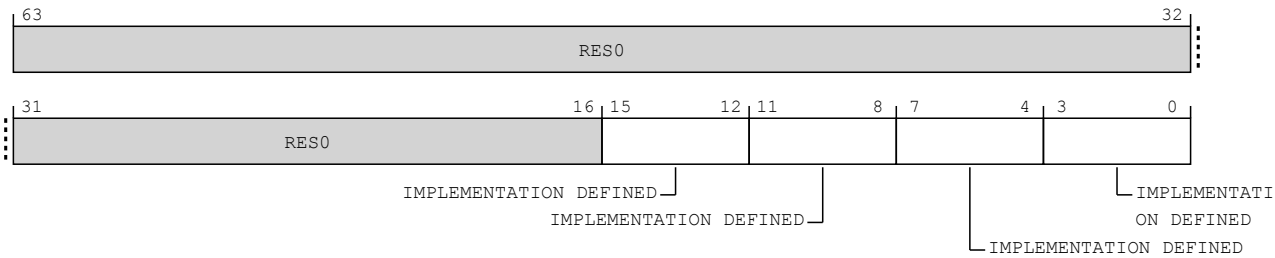

Bits [63:16]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

63 … 32	31 … 0
UNKNOWN	UNKNOWN

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_AFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_AFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = ID_AFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_AFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.98 ID_DFR0_EL1, AArch32 Debug Feature Register 0

The ID_DFR0_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_DFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_DFR0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_DFR0_EL1 are UNDEFINED.

Attributes

ID_DFR0_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

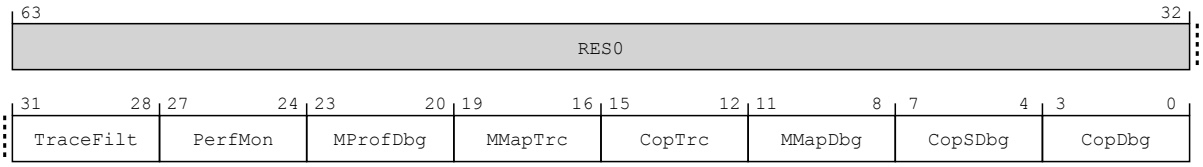

Bits [63:32]

Reserved, RES0.

TraceFilt, bits [31:28]

Armv8.4 Self-hosted Trace Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceFilt	Meaning
0b0000	Armv8.4 Self-hosted Trace Extension not implemented.
0b0001	Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_ETMv4 is implemented, the value 0b0000 is not permitted.

If FEAT_ETE is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PerfMon, bits [27:24]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'

The value of this field is an IMPLEMENTATION DEFINED choice of:

PerfMon	Meaning
0b0000	Performance Monitors Extension not implemented.
0b0001	Performance Monitors Extension, PMUv1 implemented.
0b0010	Performance Monitors Extension, PMUv2 implemented.
0b0011	Performance Monitors Extension, PMUv3 implemented.
0b0100	PMUv3 for Armv8.1. As 0b0011, and adds support for: • Extended 16-bit PMEVTYPER<n>.evtCount field. • If EL2 is implemented, the HDCR.HPMD control.
0b0101	PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR register.
0b0110	PMUv3 for Armv8.5. As 0b0101, and adds support for: • 64-bit event counters. • If EL2 is implemented, the HDCR.HCCD control. • If EL3 is implemented, the MDCR_EL3.SCCD control.
0b0111	PMUv3 for Armv8.7. As 0b0110, and adds support for: • The PMCR.FZO and, if EL2 is implemented, HDCR.HPMFZO controls. • If EL3 is implemented, the MDCR_EL3.{MPMX,MCCD} controls.
0b1000	PMUv3 for Armv8.8. As 0b0111, and: • Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF. • Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.
0b1001	PMUv3 for Armv8.9. As 0b1000, and: • Updates the definitions of existing PMU events. • Adds support for the EDECR.PME control.
0b1111	IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0011.

FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.

FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.

FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.

FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.

FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

FEAT_PMUv3p9 implements the functionality identified by the value 0b1001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In any Armv8 implementation, the values `0b0001` and `0b0010` are not permitted.

From Armv8.1, if FEAT_PMUv3 is implemented, the value `0b0011` is not permitted.

From Armv8.4, if FEAT_PMUv3 is implemented, the value `0b0100` is not permitted.

From Armv8.5, if FEAT_PMUv3 is implemented, the value `0b0101` is not permitted.

From Armv8.7, if FEAT_PMUv3 is implemented, the value `0b0110` is not permitted.

From Armv8.8, if FEAT_PMUv3 is implemented, the value `0b0111` is not permitted.

From Armv8.9, if FEAT_PMUv3 is implemented, the value `0b1000` is not permitted.

Access to this field is RO.

MProfDbg, bits [23:20]

M-profile Debug. Support for memory-mapped debug model for M-profile processors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MProfDbg	Meaning
`0b0000`	Not supported.
`0b0001`	Support for M-profile Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Access to this field is RO.

MMapTrc, bits [19:16]

Memory-mapped Trace. Support for memory-mapped trace model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MMapTrc	Meaning
`0b0000`	Not supported.
`0b0001`	Support for Arm trace architecture, with memory-mapped access.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

For more information, see the Arm® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

Access to this field is RO.

CopTrc, bits [15:12]

Support for System registers-based trace model, using registers in the coproc == `0b1110` encoding space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CopTrc	Meaning
0b0000	Not supported.
0b0001	Support for Arm trace architecture, with System registers access.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

For more information, see the Arm® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

Access to this field is RO.

MMapDbg, bits [11:8]

Memory-mapped Debug. Support for Armv7 memory-mapped debug model for A and R-profile processors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MMapDbg	Meaning
0b0000	Not supported.
0b0100	Support for Armv7, v7 Debug architecture, with memory-mapped access.
0b0101	Support for Armv7, v7.1 Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

The optional memory map defined by Armv8 is not compatible with Armv7.

Access to this field is RO.

CopSDbg, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = 0b1110 encoding space, for an A-profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CopDbg, bits [3:0]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CopDbg	Meaning
0b0000	Not supported.
0b0010	Armv6, v6 Debug architecture, with System registers access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CopDbg	Meaning
0b0011	Armv6, v6.1 Debug architecture, with System registers access.
0b0100	Armv7, v7 Debug architecture, with System registers access.
0b0101	Armv7, v7.1 Debug architecture, with System registers access.
0b0110	Armv8 debug architecture.
0b0111	Armv8.1 debug architecture, FEAT_Debugv8p1.
0b1000	Armv8.2 debug architecture, FEAT_Debugv8p2.
0b1001	Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010	Armv8.8 debug architecture, FEAT_Debugv8p8.
0b1011	Armv8.9 debug architecture, FEAT_Debugv8p9.

All other values are reserved.

The values 0b0000, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted in Armv8.

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

FEAT_Debugv8p8 implements the functionality identified by the value 0b1010.

FEAT_Debugv8p9 implements the functionality identified by the value 0b1011.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

From Armv8.8, the value 0b1001 is not permitted.

From Armv8.9, the value 0b1010 is not permitted.

Access to this field is RO.

Otherwise:

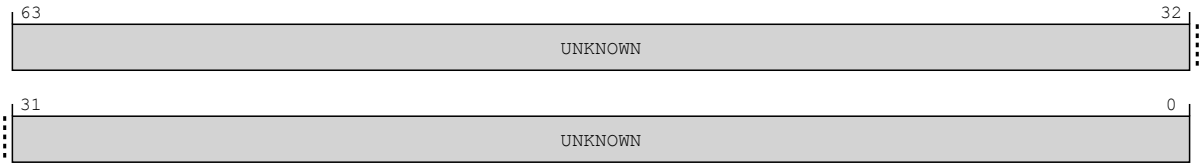

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_DFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_DFR0_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_DFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_DFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_DFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.99 ID_DFR1_EL1, AArch32 Debug Feature Register 1

The ID_DFR1_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch64 System register ID_DFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_DFR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_DFR1_EL1 are UNDEFINED.

Attributes

ID_DFR1_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

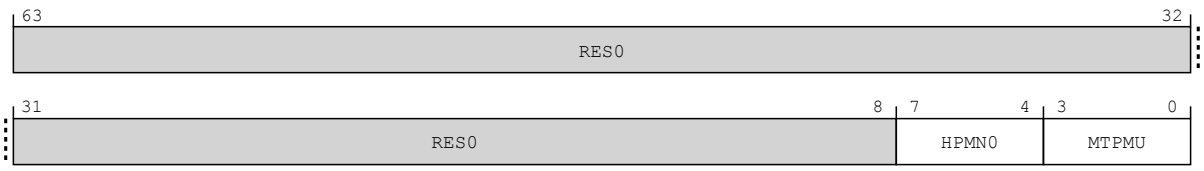

Bits [63:8]

Reserved, RES0.

HPMN0, bits [7:4]

Zero PMU event counters for a Guest operating system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HPMN0	Meaning
0b0000	Setting HDCR.HPMN to zero has CONSTRAINED UNPREDICTABLE behavior.
0b0001	Setting HDCR.HPMN to zero has defined behavior.

All other values are reserved.

If FEAT_PMUv3 is not implemented, FEAT_FGT is not implemented, or EL2 is not implemented, the only permitted value is 0b0000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_HPMN0 implements the functionality identified by the value `0b0001`.

From Armv8.8, in an implementation that includes FEAT_PMUv3, FEAT_FGT, and EL2, the value `0b0000` is not permitted.

Access to this field is RO.

MTPMU, bits [3:0]

Multi-threaded PMU extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTPMU	Meaning
`0b0000`	FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>.MT are read/write or RES0.
`0b0001`	FEAT_MTPMU and FEAT_PMUv3 implemented. PMEVTYPER<n>.MT are read/write. When FEAT_MTPMU is disabled, the Effective values of PMEVTYPER<n>.MT are 0.
`0b1111`	FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, PMEVTYPER<n>.MT are RES0.

All other values are reserved.

FEAT_MTPMU implements the functionality identified by the value `0b0001`.

From Armv8.6, in an implementation that includes FEAT_PMUv3, the value `0b0000` is not permitted.

In an implementation that does not include FEAT_PMUv3, the value `0b0001` is not permitted.

Access to this field is RO.

Otherwise:

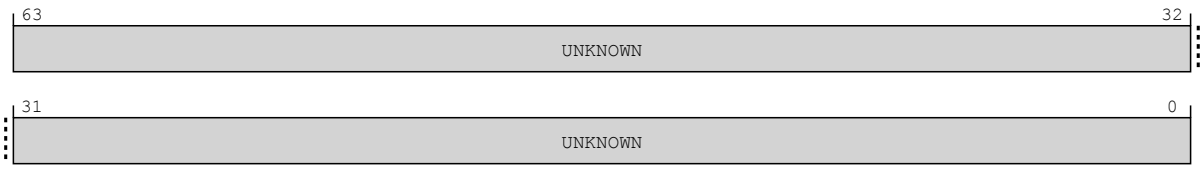

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_DFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_DFR1_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b0000`	`0b0011`	`0b101`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_DFR1_EL1()) ||
        ↪ImpDefBool("ID_DFR1_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_DFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_DFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_DFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.100 ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0

The ID_ISAR0_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_ISAR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR0_EL1 are UNDEFINED.

Attributes

ID_ISAR0_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

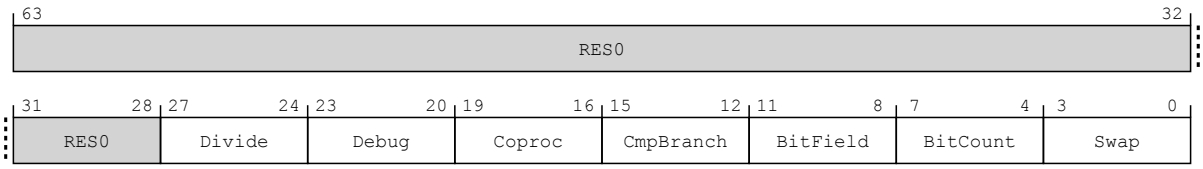

Bits [63:28]

Reserved, RES0.

Divide, bits [27:24]

Indicates the implemented Divide instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Divide	Meaning
0b0000	None implemented.
0b0001	Adds SDIV and UDIV in the T32 instruction set.
0b0010	As for 0b0001, and adds SDIV and UDIVin the A32 instruction set.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug, bits [23:20]

Indicates the implemented Debug instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Debug	Meaning
0b0000	None implemented.
0b0001	Adds BKPT.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Coproc, bits [19:16]

Indicates the implemented System register access instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Coproc	Meaning
0b0000	None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.
0b0001	Adds generic CDP, LDC, MCR, MRC, and STC.
0b0010	As for 0b0001, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.
0b0011	As for 0b0010, and adds generic MCRR and MRRC.
0b0100	As for 0b0011, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

CmpBranch, bits [15:12]

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CmpBranch	Meaning
0b0000	None implemented.
0b0001	Adds CBNZ and CBZ.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BitField, bits [11:8]

Indicates support for the following BitField instructions `BFC`, `BFI`, `SBFX`, and `UBFX`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BitField	Meaning
`0b0000`	The specified instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

BitCount, bits [7:4]

Indicates the implemented Bit Counting instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BitCount	Meaning
`0b0000`	None implemented.
`0b0001`	Adds `CLZ`.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

Swap, bits [3:0]

Indicates the implemented Swap instructions in the A32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Swap	Meaning
`0b0000`	None implemented.
`0b0001`	Adds `SWP` and `SWPB`.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0000`.

Access to this field is RO.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

63 … 32	31 … 0
UNKNOWN	UNKNOWN

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_ISAR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = ID_ISAR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.101 ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1

The ID_ISAR1_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_ISAR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR1_EL1 are UNDEFINED.

Attributes

ID_ISAR1_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

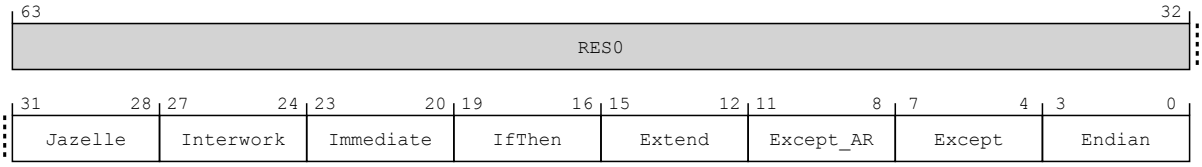

Bits [63:32]

Reserved, RES0.

Jazelle, bits [31:28]

Indicates the implemented Jazelle extension instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Jazelle	Meaning
0b0000	No support for Jazelle.
0b0001	Adds the BXJ instruction and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Interwork, bits [27:24]

Indicates the implemented Interworking instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Interwork	Meaning
0b0000	None implemented.
0b0001	Adds the BX instruction, and the T bit in the PSR.
0b0010	As for 0b0001, and adds the BLX instruction. PC loads have BX-like behavior.
0b0011	As for 0b0010, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0011.

Access to this field is RO.

Immediate, bits [23:20]

Indicates support for the following data-processing instructions with long immediates:

- The MOVT instruction.
- The MOV instruction encodings with zero-extended 16-bit immediates.
- The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Immediate	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

IfThen, bits [19:16]

Indicates the implemented If-Then instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IfThen	Meaning
0b0000	None implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0001	Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Extend, bits [15:12]

Indicates the implemented Extend instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Extend	Meaning
0b0000	No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.
0b0001	Adds the SXTB, SXTH, UXTB, and UXTH instructions.
0b0010	As for 0b0001, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

Access to this field is RO.

Except_AR, bits [11:8]

Indicates the implemented A and R-profile exception-handling instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Except_AR	Meaning
0b0000	None implemented.
0b0001	Adds the SRS and RFE instructions, and the A and R-profile forms of the CPS instruction.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Except, bits [7:4]

Indicates the implemented exception-handling instructions in the A32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Except	Meaning
0b0000	Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.
0b0001	Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Endian, bits [3:0]

Indicates the implemented Endian instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Endian	Meaning
0b0000	None implemented.
0b0001	Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Otherwise:

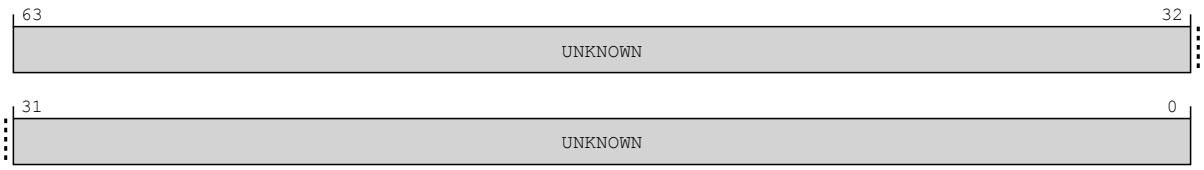

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_ISAR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.102 ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2

The ID_ISAR2_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_ISAR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR2_EL1 are UNDEFINED.

Attributes

ID_ISAR2_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

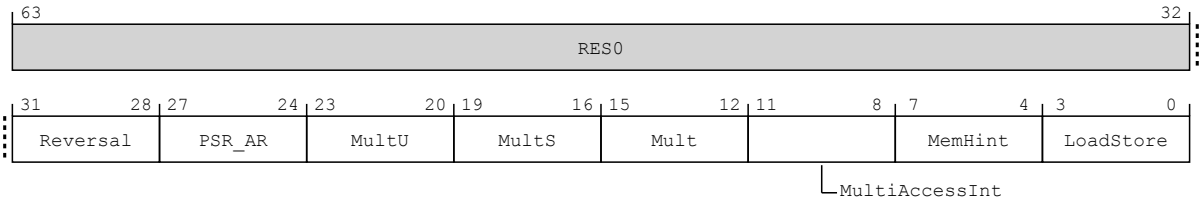

Bits [63:32]

Reserved, RES0.

Reversal, bits [31:28]

Indicates the implemented Reversal instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Reversal	Meaning
0b0000	None implemented.
0b0001	Adds the REV, REV16, and REVSH instructions.
0b0010	As for 0b0001, and adds the RBIT instruction.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSR_AR, bits [27:24]

Indicates the implemented A and R-profile instructions to manipulate the PSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PSR_AR	Meaning
0b0000	None implemented.
0b0001	Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC,LR,#N instruction.

Access to this field is RO.

MultU, bits [23:20]

Indicates the implemented advanced unsigned Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultU	Meaning
0b0000	None implemented.
0b0001	Adds the UMULL and UMLAL instructions.
0b0010	As for 0b0001, and adds the UMAAL instruction.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

Access to this field is RO.

MultS, bits [19:16]

Indicates the implemented advanced signed Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultS	Meaning
0b0000	None implemented.
0b0001	Adds the SMULL and SMLAL instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MultS	Meaning
0b0010	As for 0b0001, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.
0b0011	As for 0b0010, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0011.

Access to this field is RO.

Mult, bits [15:12]

Indicates the implemented additional Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Mult	Meaning
0b0000	No additional instructions implemented. This means only MUL is implemented.
0b0001	Adds the MLA instruction.
0b0010	As for 0b0001, and adds the MLS instruction.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

Access to this field is RO.

MultiAccessInt, bits [11:8]

Indicates the support for interruptible multi-access instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultiAccessInt	Meaning
0b0000	No support. This means the LDM and STM instructions are not interruptible.
0b0001	LDM and STM instructions are restartable.
0b0010	LDM and STM instructions are continuable.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

MemHint, bits [7:4]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates the implemented Memory Hint instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MemHint	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the PLD instruction.
`0b0010`	Adds the PLD instruction. (`0b0001` and `0b0010` have identical effects.)
`0b0011`	As for `0b0001` (or `0b0010`), and adds the PLI instruction.
`0b0100`	As for `0b0011`, and adds the PLDW instruction.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0100`.

Access to this field is RO.

LoadStore, bits [3:0]

Indicates the implemented additional load/store instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LoadStore	Meaning
`0b0000`	No additional load/store instructions implemented.
`0b0001`	Adds the LDRD and STRD instructions.
`0b0010`	As for `0b0001`, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Access to this field is RO.

Otherwise:

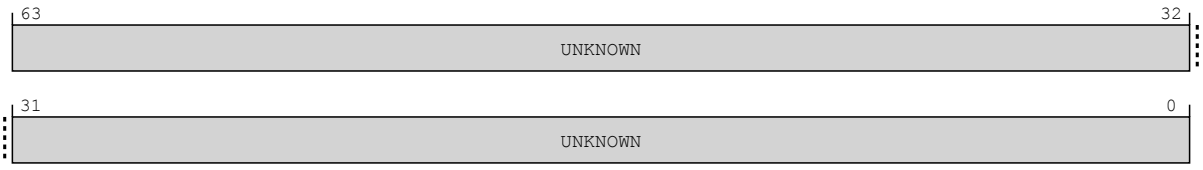

Bits [63:0]

Reserved, UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ID_ISAR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_ISAR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.103 ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3

The ID_ISAR3_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

AArch64 System register ID_ISAR3_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR3[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR3_EL1 are UNDEFINED.

Attributes

ID_ISAR3_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

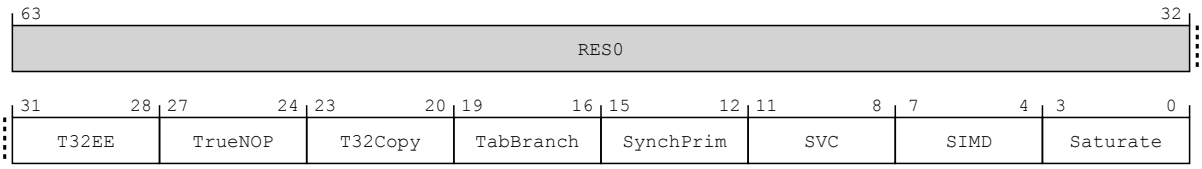

Bits [63:32]

Reserved, RES0.

T32EE, bits [31:28]

Indicates the implemented T32EE instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

T32EE	Meaning
0b0000	None implemented.
0b0001	Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TrueNOP, bits [27:24]

Indicates the implemented true NOP instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TrueNOP	Meaning
`0b0000`	None implemented. This means there are no NOP instructions that do not have any register dependencies.
`0b0001`	Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

T32Copy, bits [23:20]

Indicates the support for T32 non flag-setting MOV instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

T32Copy	Meaning
`0b0000`	Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.
`0b0001`	Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

TabBranch, bits [19:16]

Indicates the implemented Table Branch instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TabBranch	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the TBB and TBH instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SynchPrim, bits [15:12]

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SynchPrim	Meaning
`0b0000`	If SynchPrim_frac == `0b0000`, no Synchronization Primitives implemented.
`0b0001`	If SynchPrim_frac == `0b0000`, adds the LDREX and STREX instructions. If SynchPrim_frac == `0b0011`, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.
`0b0010`	If SynchPrim_frac == `0b0000`, as for [`0b0001`, `0b0011`] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Access to this field is RO.

SVC, bits [11:8]

Indicates the implemented SVC instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SVC	Meaning
`0b0000`	Not implemented.
`0b0001`	Adds the SVC instruction.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

SIMD, bits [7:4]

Indicates the implemented SIMD instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMD	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the SSAT and USAT instructions, and the Q bit in the PSRs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIMD	Meaning
0b0011	As for 0b0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0011.

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports Advanced SIMD and floating-point instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

Access to this field is RO.

Saturate, bits [3:0]

Indicates the implemented Saturate instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Saturate	Meaning
0b0000	None implemented. This means no non-Advanced SIMD saturate instructions are implemented.
0b0001	Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Otherwise:

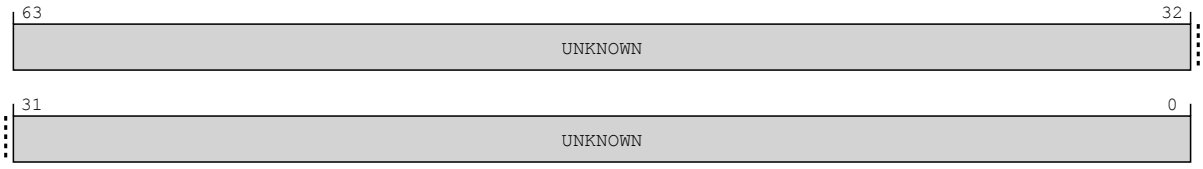

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ID_ISAR3_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR3_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR3_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR3_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.104 ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4

The ID_ISAR4_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_ISAR4_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR4[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR4_EL1 are UNDEFINED.

Attributes

ID_ISAR4_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

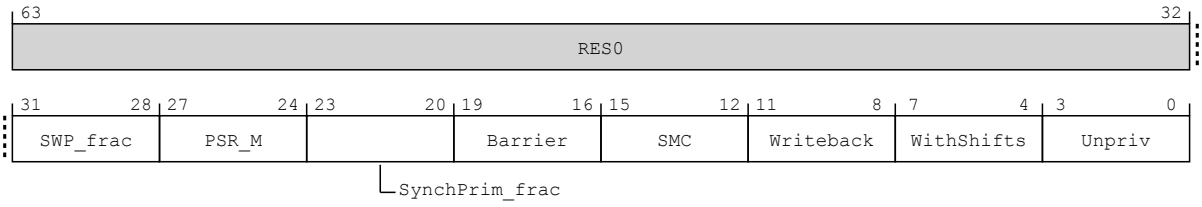

Bits [63:32]

Reserved, RES0.

SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for SWP or SWPB instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SWP_frac	Meaning
0b0000	SWP or SWPB instructions not implemented.
0b0001	SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other Requesters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is 0b0000.

In Armv8-A, the only permitted value is 0b0000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

PSR_M, bits [27:24]

Indicates the implemented M-profile instructions to modify the PSRs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PSR_M	Meaning
0b0000	None implemented.
0b0001	Adds the M-profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

SynchPrim_frac, bits [23:20]

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SynchPrim_frac	Meaning
0b0000	If SynchPrim == 0b0000, no Synchronization Primitives implemented. If SynchPrim == 0b0001, adds the LDREX and STREX instructions. If SynchPrim == 0b0010, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.
0b0011	If SynchPrim == 0b0001, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

Barrier, bits [19:16]

Indicates the implemented Barrier instructions in the T32 and A32 instruction sets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Barrier	Meaning
0b0000	None implemented. Barrier operations are provided only as System instructions in the (coproc==0b1111) encoding space.
0b0001	Adds the DMB, DSB, and ISB barrier instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

SMC, bits [15:12]

Indicates the implemented SMC instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SMC	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the SMC instruction.

All other values are reserved.

In Armv8-A, the permitted values are:

- If EL3 is implemented and EL1 can use AArch32, the only permitted value is `0b0001`.
- If neither EL3 nor EL2 is implemented, the only permitted value is `0b0000`.

If EL1 cannot use AArch32, this field has the value `0b0000`.

Access to this field is RO.

Writeback, bits [11:8]

Indicates the support for Writeback addressing modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Writeback	Meaning
`0b0000`	Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.
`0b0001`	Adds support for all of the writeback addressing modes.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

Access to this field is RO.

WithShifts, bits [7:4]

Indicates the support for instructions with shifts.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WithShifts	Meaning
`0b0000`	Nonzero shifts supported only in MOV and shift instructions.
`0b0001`	Adds support for shifts of loads and stores over the range LSL 0-3.
`0b0011`	As for `0b0001`, and adds support for other constant shift options, both on load/store and other instructions.
`0b0100`	As for `0b0011`, and adds support for register-controlled shift options.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0100`.

Access to this field is RO.

Unpriv, bits [3:0]

Indicates the implemented unprivileged instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Unpriv	Meaning
`0b0000`	None implemented. No T variant instructions are implemented.
`0b0001`	Adds the LDRBT, LDRT, STRBT, and STRT instructions.
`0b0010`	As for `0b0001`, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Access to this field is RO.

Otherwise:

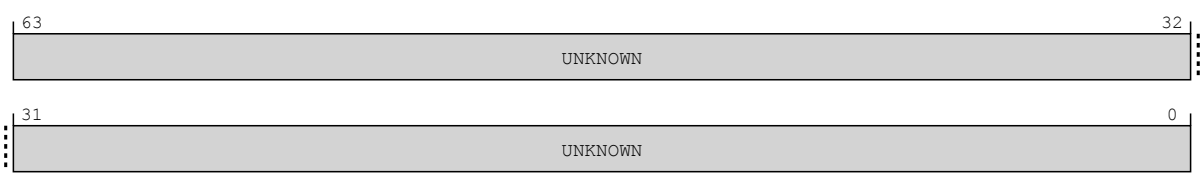

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR4_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_ISAR4_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR4_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR4_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR4_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.105 ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5

The ID_ISAR5_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR4_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_ISAR5_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR5[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR5_EL1 are UNDEFINED.

Attributes

ID_ISAR5_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

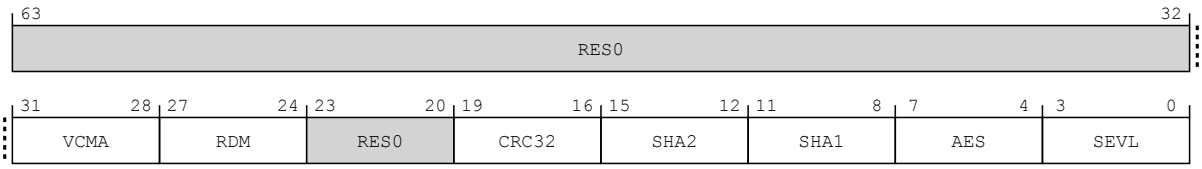

Bits [63:32]

Reserved, RES0.

VCMA, bits [31:28]

Indicates AArch32 support for complex number addition and multiplication where numbers are stored in vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VCMA	Meaning
0b0000	The VCMLA and VCADD instructions are not implemented in AArch32.
0b0001	The VCMLA and VCADD instructions are implemented in AArch32.

All other values are reserved.

FEAT_FCMA implements the functionality identified by 0b0001.

From Armv8.3, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RDM, bits [27:24]

Indicates whether the `VQRDMLAH` and `VQRDMLSH` instructions are implemented in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RDM	Meaning
`0b0000`	No `VQRDMLAH` and `VQRDMLSH` instructions implemented.
`0b0001`	`VQRDMLAH` and `VQRDMLSH` instructions implemented.

All other values are reserved.

FEAT_RDM implements the functionality identified by the value `0b0001`.

From Armv8.1, the value `0b0000` is not permitted.

Access to this field is RO.

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates whether the CRC32 instructions `CRC32B`, `CRC32H`, `CRC32W`, `CRC32CB`, `CRC32CH`, and `CRC32CW` are implemented in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CRC32	Meaning
`0b0000`	CRC32 instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_CRC32 implements the functionality identified by the value `0b0001`.

In Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.1, the value `0b0000` is not permitted.

Access to this field is RO.

SHA2, bits [15:12]

Indicates whether the SHA2 instructions `SHA256H`, `SHA256H2`, `SHA256SU0`, and `SHA256SU1` are implemented in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA2	Meaning
`0b0000`	No SHA2 instructions implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SHA2	Meaning
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SHA1, bits [11:8]

Indicates whether the SHA1 instructions SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 are implemented in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA1	Meaning
0b0000	No SHA1 instructions implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

AES, bits [7:4]

Indicates whether the AES instructions are implemented in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AES	Meaning
0b0000	No AES instructions implemented.
0b0001	AESE, AESD, AESMC, and AESIMC implemented.
0b0010	As for 0b0001, plus VMULL (polynomial) instructions operating on 64-bit data quantities.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0010.

Access to this field is RO.

SEVL, bits [3:0]

Indicates whether the SEVL instruction is implemented in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SEVL	Meaning
0b0000	SEVL is implemented as a NOP.
0b0001	SEVL is implemented as Send Event Local.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Otherwise:

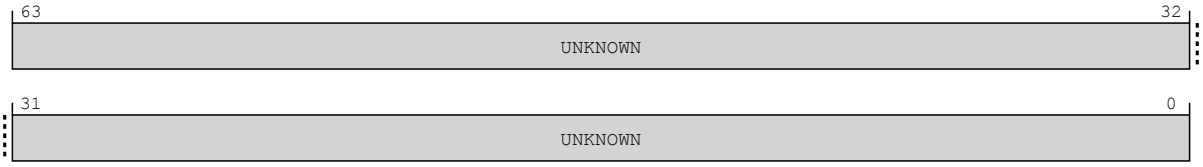

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR5_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_ISAR5_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR5_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR5_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR5_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.106 ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6

The ID_ISAR6_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1 and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch64 System register ID_ISAR6_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR6[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_ISAR6_EL1 are UNDEFINED.

Attributes

ID_ISAR6_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

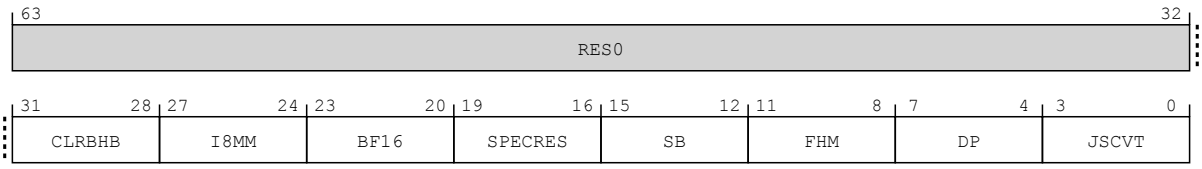

Bits [63:32]

Reserved, RES0.

CLRBHB, bits [31:28]

Indicates support for the CLRBHB instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CLRBHB	Meaning
0b0000	CLRBHB instruction is not implemented.
0b0001	CLRBHB instruction is implemented.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_CLRBHB implements the functionality identified by `0b0001`.

From Armv8.9, the value `0b0000` is not permitted.

Access to this field is RO.

I8MM, bits [27:24]

Indicates support for the following Advanced SIMD Int8 matrix multiplication instructions `VSMMLA`, `VSUDOT`, `VUMMLA`, `VUSMMLA`, and `VUSDOT` in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

I8MM	Meaning
`0b0000`	Int8 matrix multiplication instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_AA32I8MM implements the functionality identified by `0b0001`.

Access to this field is RO.

BF16, bits [23:20]

Indicates support for the following Advanced SIMD and floating-point BFloat16 instructions `VCVT`, `VCVTB`, `VCVTT`, `VDOT`, `VFMAB`, `VFMAT`, and `VMMLA` instructions with BF16 operand or result types in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BF16	Meaning
`0b0000`	BFloat16 instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_AA32BF16 implements the functionality identified by `0b0001`.

Access to this field is RO.

SPECRES, bits [19:16]

Indicates support for prediction invalidation instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SPECRES	Meaning
`0b0000`	Prediction invalidation instructions are not implemented.
`0b0001`	CFPRCTX, DVPRCTX, and CPPRCTX instructions are implemented.
`0b0010`	As `0b0001`, and COSPRCTX instruction is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_SPECRES implements the functionality identified by `0b0001`.

FEAT_SPECRES2 implements the functionality identified by `0b0010`.

From Armv8.5, the value `0b0000` is not permitted.

From Armv8.9, the value `0b0001` is not permitted.

Access to this field is RO.

SB, bits [15:12]

Indicates support for the `SB` instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SB	Meaning
`0b0000`	`SB` instruction is not implemented.
`0b0001`	`SB` instruction is implemented.

All other values are reserved.

FEAT_SB implements the functionality identified by `0b0001`.

From Armv8.5, value `0b0000` is not permitted.

Access to this field is RO.

FHM, bits [11:8]

Indicates support for the following Advanced SIMD and floating-point `VFMAL` and `VFMSL` instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FHM	Meaning
`0b0000`	The specified instructions are not implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

FEAT_FHM implements the functionality identified by `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

DP, bits [7:4]

Indicates support for dot product instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DP	Meaning
0b0000	Dot product instructions are not implemented.
0b0001	VUDOT and VSDOT instructions are implemented.

All other values are reserved.

FEAT_DotProd implements the functionality identified by 0b0001.

In Armv8.2, the permitted values are 0b0000 and 0b0001.

From Armv8.4, the only permitted value is 0b0001.

Access to this field is RO.

JSCVT, bits [3:0]

Indicates support for the VJCVT instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

JSCVT	Meaning
0b0000	The VJCVT instruction is not implemented.
0b0001	The VJCVT instruction is implemented.

All other values are reserved.

FEAT_JSCVT implements the functionality identified by 0b0001.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the value 0b0000 is not implemented.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

Otherwise:

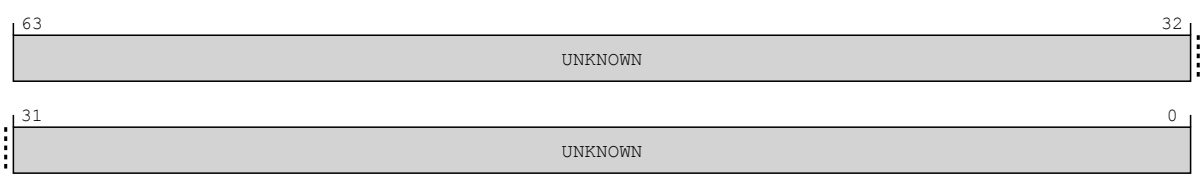

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR6_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ID_ISAR6_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_ISAR6_EL1()) ||
     ↪ImpDefBool("ID_ISAR6_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR6_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_ISAR6_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_ISAR6_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.107 ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0

The ID_MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_MMFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_MMFR0_EL1 are UNDEFINED.

Attributes

ID_MMFR0_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

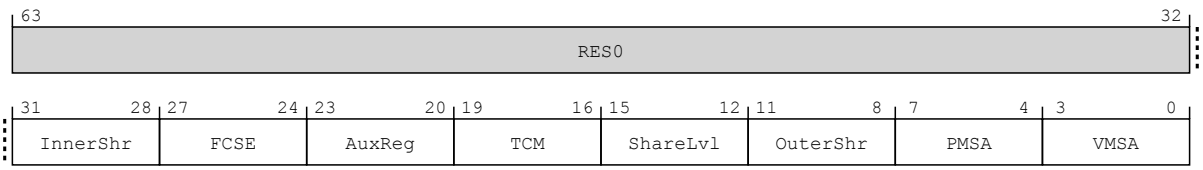

Bits [63:32]

Reserved, RES0.

InnerShr, bits [31:28]

Innermost Shareability. Indicates the innermost shareability domain implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

InnerShr	Meaning
0b0000	Implemented as Non-cacheable.
0b0001	Implemented with hardware coherency support.
0b1111	Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are 0b0000, 0b0001, and 0b1111.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0_EL1.ShareLvl having the value 0b0001.

When ID_MMFR0_EL1.ShareLvl is zero, this field is UNKNOWN.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FCSE, bits [27:24]

Indicates whether the implementation includes the FCSE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FCSE	Meaning
0b0000	Not supported.
0b0001	Support for FCSE.

All other values are reserved.

From Armv8 the only permitted value is 0b0000.

Access to this field is RO.

AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxReg	Meaning
0b0000	None supported.
0b0001	Support for Auxiliary Control Register only.
0b0010	Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

From Armv8 the only permitted value is 0b0010.

Note

Accesses to unimplemented Auxiliary registers are UNDEFINED.

Access to this field is RO.

TCM, bits [19:16]

Indicates support for TCMs and associated DMAs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TCM	Meaning
0b0000	Not supported.
0b0001	Support is IMPLEMENTATION DEFINED.
0b0010	Support for TCM only, Armv6 implementation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0011	Support for TCM and DMA, Armv6 implementation.

All other values are reserved.

In Armv8-A the only permitted value is 0b0000.

Access to this field is RO.

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of shareability levels implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ShareLvl	Meaning
0b0000	One level of shareability implemented.
0b0001	Two levels of shareability implemented.

All other values are reserved.

From Armv8 the only permitted value is 0b0001.

Access to this field is RO.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost shareability domain implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OuterShr	Meaning
0b0000	Implemented as Non-cacheable.
0b0001	Implemented with hardware coherency support.
0b1111	Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are 0b0000, 0b0001, and 0b1111.

Access to this field is RO.

PMSA, bits [7:4]

Indicates support for a PMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSA	Meaning
0b0000	Not supported.
0b0001	Support for IMPLEMENTATION DEFINED PMSA.
0b0010	Support for PMSAv6, with a Cache Type Register implemented.
0b0011	Support for PMSAv7, with support for memory subsections. Armv7-R profile.

All other values are reserved.

In Armv8-A the only permitted value is 0b0000.

Access to this field is RO.

VMSA, bits [3:0]

Indicates support for a VMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMSA	Meaning
0b0000	Not supported.
0b0001	Support for IMPLEMENTATION DEFINED VMSA.
0b0010	Support for VMSAv6, with Cache and TLB Type Registers implemented.
0b0011	Support for VMSAv7, with support for remapping and the Access flag. Armv7-A profile.
0b0100	As for 0b0011, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.
0b0101	As for 0b0100, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In Armv8-A the only permitted value is 0b0101.

Access to this field is RO.

Otherwise:

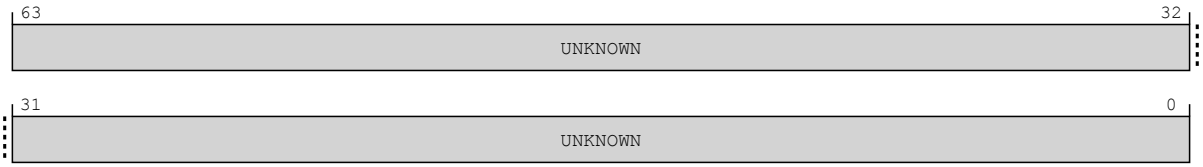

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ID_MMFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_MMFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.108 ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1

The ID_MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_MMFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_MMFR1_EL1 are UNDEFINED.

Attributes

ID_MMFR1_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

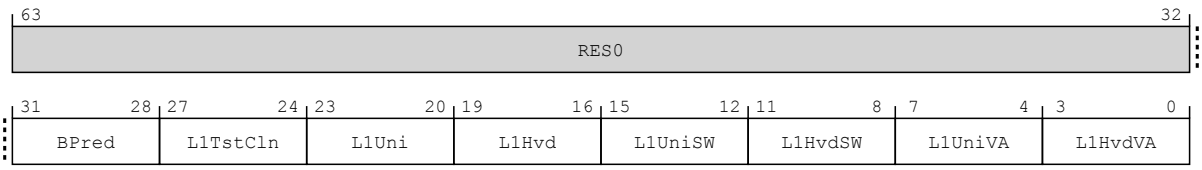

Bits [63:32]

Reserved, RES0.

BPred, bits [31:28]

Branch Predictor. Indicates branch predictor management requirements.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPred	Meaning
0b0000	No branch predictor, or no MMU present. Implies a fixed MPU configuration.
0b0001	Branch predictor requires flushing on: • Enabling or disabling a stage of address translation. • Writing new data to instruction locations. • Writing new mappings to the translation tables. • Changes to the TTBR0, TTBR1, or TTBCR registers. • Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BPred	Meaning
0b0010	Branch predictor requires flushing on: • Enabling or disabling a stage of address translation. • Writing new data to instruction locations. • Writing new mappings to the translation tables. • Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.
0b0011	Branch predictor requires flushing only on writing new data to instruction locations.
0b0100	For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In Armv8-A, the permitted values are 0b0010, 0b0011, and 0b0100. For values other than 0b0000 and 0b0100 the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

Access to this field is RO.

L1TstCln, bits [27:24]

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1TstCln	Meaning
0b0000	None supported.
0b0001	Supported Level 1 data cache test and clean operations are: • Test and clean data cache.
0b0010	As for 0b0001, and adds: • Test, clean, and invalidate data cache.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

L1Uni, bits [23:20]

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Uni	Meaning
0b0000	None supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L1Uni	Meaning
0b0001	Supported entire Level 1 cache operations are: • Invalidate cache, including branch predictor if appropriate. • Invalidate branch predictor, if appropriate.
0b0010	As for 0b0001, and adds: • Clean cache, using a recursive model that uses the cache dirty status bit. • Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

L1Hvd, bits [19:16]

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Hvd	Meaning
0b0000	None supported.
0b0001	Supported entire Level 1 cache operations are: • Invalidate instruction cache, including branch predictor if appropriate. • Invalidate branch predictor, if appropriate.
0b0010	As for 0b0001, and adds: • Invalidate data cache. • Invalidate data cache and instruction cache, including branch predictor if appropriate.
0b0011	As for 0b0010, and adds: • Clean data cache, using a recursive model that uses the cache dirty status bit. • Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

L1UniSW, bits [15:12]

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L1UniSW	Meaning
0b0000	None supported.
0b0001	Supported Level 1 unified cache line maintenance operations by set/way are: • Clean cache line by set/way.
0b0010	As for 0b0001, and adds: • Clean and invalidate cache line by set/way.
0b0011	As for 0b0010, and adds: • Invalidate cache line by set/way.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdSW, bits [11:8]

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdSW	Meaning
0b0000	None supported.
0b0001	Supported Level 1 Harvard cache line maintenance operations by set/way are: • Clean data cache line by set/way. • Clean and invalidate data cache line by set/way.
0b0010	As for 0b0001, and adds: • Invalidate data cache line by set/way.
0b0011	As for 0b0010, and adds: • Invalidate instruction cache line by set/way.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

L1UniVA, bits [7:4]

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L1UniVA	Meaning
0b0000	None supported.
0b0001	Supported Level 1 unified cache line maintenance operations by VA are: • Clean cache line by VA. • Invalidate cache line by VA. • Clean and invalidate cache line by VA.
0b0010	As for 0b0001, and adds: • Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdVA, bits [3:0]

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdVA	Meaning
0b0000	None supported.
0b0001	Supported Level 1 Harvard cache line maintenance operations by VA are: • Clean data cache line by VA. • Invalidate data cache line by VA. • Clean and invalidate data cache line by VA. • Clean instruction cache line by VA.
0b0010	As for 0b0001, and adds: • Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

Otherwise:

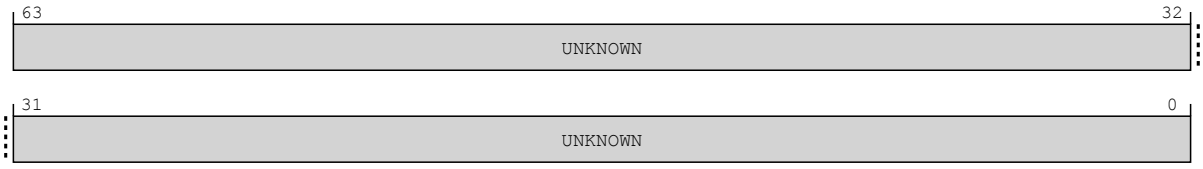

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_MMFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.109 ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2

The ID_MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_MMFR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_MMFR2_EL1 are UNDEFINED.

Attributes

ID_MMFR2_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

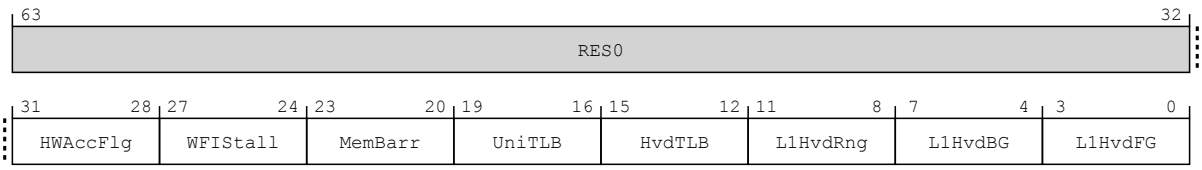

Bits [63:32]

Reserved, RES0.

HWAccFlg, bits [31:28]

Hardware Access Flag. In earlier versions of the Arm Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HWAccFlg	Meaning
0b0000	Not supported.
0b0001	Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

From Armv8.0, 0b0001 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WFIStall, bits [27:24]

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WFIStall	Meaning
`0b0000`	Not supported.
`0b0001`	Support for WFI stalling.

All other values are reserved.

Access to this field is RO.

MemBarr, bits [23:20]

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc==`0b1111`) encoding space:

The value of this field is an IMPLEMENTATION DEFINED choice of:

MemBarr	Meaning
`0b0000`	None supported.
`0b0001`	Supported memory barrier System instructions are: • Data Synchronization Barrier (DSB).
`0b0010`	As for `0b0001`, and adds: • Instruction Synchronization Barrier (ISB). • Data Memory Barrier (DMB).

All other values are reserved.

From Armv8.0, the values `0b000` and `0b0001` are not permitted.

Arm deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

Access to this field is RO.

UniTLB, bits [19:16]

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

UniTLB	Meaning
`0b0000`	Not supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UniTLB	Meaning
0b0001	Supported unified TLB maintenance operations are: • Invalidate all entries in the TLB. • Invalidate TLB entry by VA.
0b0010	As for 0b0001, and adds: • Invalidate TLB entries by ASID match.
0b0011	As for 0b0010, and adds: • Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation.
0b0100	As for 0b0011, and adds: • Invalidate Hyp mode unified TLB entry by VA. • Invalidate entire Non-secure PL1&0 unified TLB. • Invalidate entire Hyp mode unified TLB.
0b0101	As for 0b0100, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL, TLBIMVALH.
0b0110	As for 0b0101, and adds the following operations: TLBIIPAS2IS, TLBIIPAS2LIS, TLBIIPAS2, TLBIIPAS2L.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0110.

Access to this field is RO.

HvdTLB, bits [15:12]

If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1HvdRng, bits [11:8]

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdRng	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache maintenance range operations are: • Invalidate data cache range by VA. • Invalidate instruction cache range by VA. • Clean data cache range by VA. • Clean and invalidate data cache range by VA.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv8.0, the value `0b0001` is not permitted.

Access to this field is RO.

L1HvdBG, bits [7:4]

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdBG	Meaning
`0b0000`	Not supported.
`0b0001`	Supported Level 1 Harvard cache background fetch operations are: • Fetch instruction cache range by VA. • Fetch data cache range by VA.

All other values are reserved.

From Armv8.0, the value `0b0001` is not permitted.

Access to this field is RO.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdFG	Meaning
`0b0000`	Not supported.
`0b0001`	Supported Level 1 Harvard cache foreground fetch operations are: • Fetch instruction cache range by VA. • Fetch data cache range by VA.

All other values are reserved.

From Armv8.0, the value `0b0001` is not permitted.

Access to this field is RO.

Otherwise:

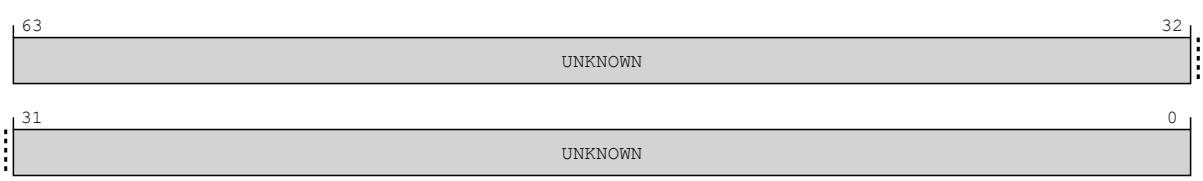

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_MMFR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.110 ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3

The ID_MMFR3_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_MMFR3_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR3[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_MMFR3_EL1 are UNDEFINED.

Attributes

ID_MMFR3_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

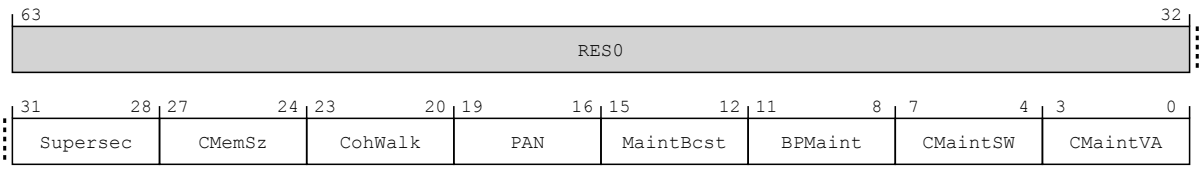

Bits [63:32]

Reserved, RES0.

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Supersec	Meaning
0b0000	Supersections supported.
0b1111	Supersections not supported.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b1111.

Access to this field is RO.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

CMemSz	Meaning
0b0000	4GB, corresponding to a 32-bit physical address range.
0b0001	64GB, corresponding to a 36-bit physical address range.
0b0010	1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification.:

The value of this field is an IMPLEMENTATION DEFINED choice of:

CohWalk	Meaning
0b0000	Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.
0b0001	Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

PAN, bits [19:16]

Privileged Access Never. Indicates support for the PAN bit in CPSR, SPSR, and DSPSR in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PAN	Meaning
0b0000	PAN not supported.
0b0001	PAN supported.
0b0010	PAN supported and ATS1CPRP and ATS1CPWP instructions supported.

All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv8.2, the value `0b0001` is not permitted.

Access to this field is RO.

MaintBcst, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MaintBcst	Meaning
`0b0000`	Cache, TLB, and branch predictor operations only affect local structures.
`0b0001`	Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.
`0b0010`	Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Access to this field is RO.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPMaint	Meaning
`0b0000`	None supported.
`0b0001`	Supported branch predictor maintenance operations are: • Invalidate all branch predictors.
`0b0010`	As for `0b0001`, and adds: • Invalidate branch predictors by VA.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0010`.

Access to this field is RO.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CMaintSW	Meaning
`0b0000`	None supported.
`0b0001`	Supported hierarchical cache maintenance instructions by set/way are: • Invalidate data cache by set/way. • Clean data cache by set/way. • Clean and invalidate data cache by set/way.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

Access to this field is RO.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CMaintVA	Meaning
`0b0000`	None supported.
`0b0001`	Supported hierarchical cache maintenance operations by VA are: • Invalidate data cache by VA. • Clean data cache by VA. • Clean and invalidate data cache by VA. • Invalidate instruction cache by VA. • Invalidate all instruction cache entries.

All other values are reserved.

In Armv8-A, the only permitted value is `0b0001`.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Access to this field is RO.

Otherwise:

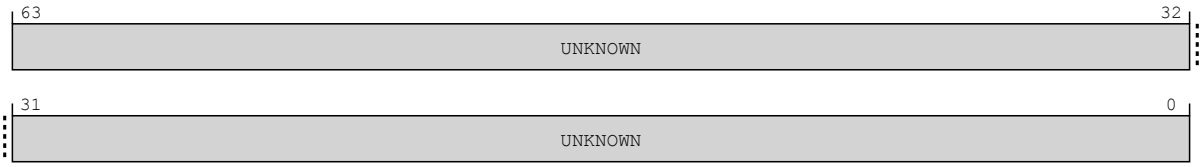

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR3_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b111

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR3_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR3_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_MMFR3_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.111 ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4

The ID_MMFR4_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch64 System register ID_MMFR4_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR4[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_MMFR4_EL1 are UNDEFINED.

Attributes

ID_MMFR4_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

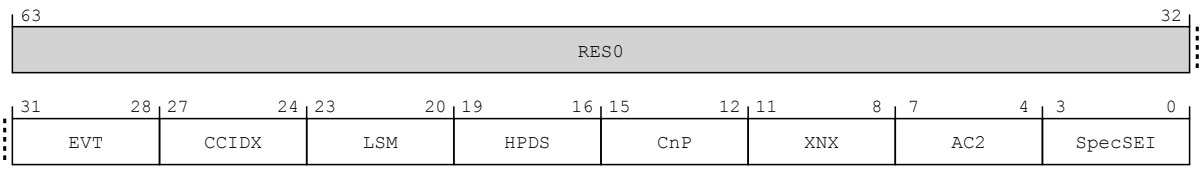

Bits [63:32]

Reserved, RES0.

EVT, bits [31:28]

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR2.{TTLBIS, TOCU, TICAB, TID4} traps.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EVT	Meaning
0b0000	HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are not supported.
0b0001	HCR2.{TOCU, TICAB, TID4} traps are supported.
0b0010	As 0b0001, and HCR2.TTLBIS trap is supported.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_EVT implements the functionality identified by the value `0b0001`.

FEAT_EVT2 implements the functionality identified by the value `0b0010`.

If EL2 is not implemented or does not support AArch32, the only permitted value is `0b0000`.

From Armv8.5, if EL2 is supported and supports AArch32, the value `0b0001` is not permitted.

Access to this field is RO.

CCIDX, bits [27:24]

Support for use of the revised CCSIDR format and the presence of the CCSIDR2 is indicated.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCIDX	Meaning
`0b0000`	32-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is not implemented.
`0b0001`	64-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is implemented.

All other values are reserved.

FEAT_CCIDX implements the functionality identified by `0b0001`.

From Armv8.3, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

LSM, bits [23:20]

Indicates support for LSMAOE and nTLSMD bits in HSCTLR and SCTLR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LSM	Meaning
`0b0000`	LSMAOE and nTLSMD bits not supported.
`0b0001`	LSMAOE and nTLSMD bits supported.

All other values are reserved.

FEAT_LSMAOC implements the functionality identified by the value `0b0001`.

From Armv8.2, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

HPDS, bits [19:16]

Hierarchical permission disables bits in translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPDS	Meaning
0b0000	Disabling of hierarchical controls not supported.
0b0001	Supports disabling of hierarchical controls using the TTBCR2.HPD0, TTBCR2.HPD1, and HTCR.HPD bits.
0b0010	As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

FEAT_AA32HPD implements the functionality identified by the value 0b0001.

FEAT_HPDS2 implements the functionality added by the value 0b0010.

Note

The value 0b0000 implies that the encoding for TTBCR2 is UNDEFINED.

Access to this field is RO.

CnP, bits [15:12]

Common not Private translations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CnP	Meaning
0b0000	Common not Private translations not supported.
0b0001	Common not Private translations supported.

All other values are reserved.

FEAT_TTCNP implements the functionality identified by the value 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

Access to this field is RO.

XNX, bits [11:8]

Support for execute-never control distinction by Exception level at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

XNX	Meaning
0b0000	Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
0b0001	Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

FEAT_XNX implements the functionality identified by the value 0b0001.

When FEAT_XNX is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If all of the following conditions are true, it is IMPLEMENTATION DEFINED whether the value of ID_MMFR4_EL1.XNX is `0b0000` or `0b0001`:
  - ID_AA64MMFR1_EL1.XNX ==1.
  - EL2 cannot use AArch32.
  - EL1 can use AArch32.
- If EL2 can use AArch32 then the value `0b0000` is not permitted.

Access to this field is RO.

AC2, bits [7:4]

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AC2	Meaning
`0b0000`	ACTLR2 and HACTLR2 are not implemented.
`0b0001`	ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In Armv8.0 and Armv8.1 the permitted values are `0b0000` and `0b0001`.

From Armv8.2, the only permitted value is `0b0001`.

Access to this field is RO.

SpecSEI, bits [3:0]

When FEAT_RAS is implemented:

Describes whether the PE can generate SError exceptions from speculative reads of memory, including speculative instruction fetches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SpecSEI	Meaning
`0b0000`	The PE never generates an SError exception due to an External abort on a speculative read.
`0b0001`	The PE might generate an SError exception due to an External abort on a speculative read.

All other values are reserved.

FEAT_SpecSEI implements the functionality identified by the value `0b0001`.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

63 … 32	31 … 0
UNKNOWN	UNKNOWN

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR4_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR4_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR4_EL1()) ||
     ↪ImpDefBool("ID_MMFR4_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR4_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = ID_MMFR4_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_MMFR4_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.112 ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5

The ID_MMFR5_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch64 System register ID_MMFR5_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR5[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_MMFR5_EL1 are UNDEFINED.

Attributes

ID_MMFR5_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

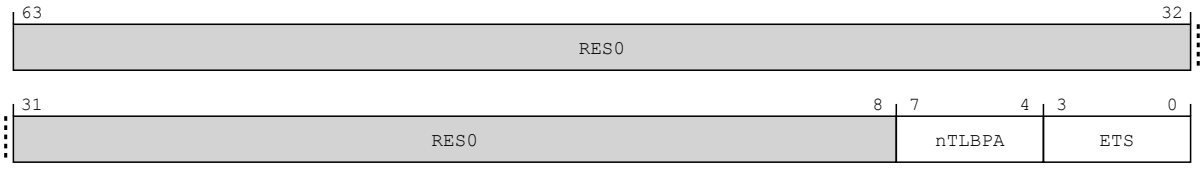

Bits [63:8]

Reserved, RES0.

nTLBPA, bits [7:4]

Indicates support for intermediate caching of translation table walks.

The value of this field is an IMPLEMENTATION DEFINED choice of:

nTLBPA	Meaning
0b0000	The intermediate caching of translation table walks might include non-coherent physical translation caches.
0b0001	The intermediate caching of translation table walks does not include non-coherent physical translation caches.

Non-coherent physical translation caches are non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE, where either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

FEAT_nTLBPA implements the functionality identified by the value `0b0001`.

From Armv8.0, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

ETS, bits [3:0]

Indicates support for Enhanced Translation Synchronization.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ETS	Meaning
`0b0000`	Enhanced Translation Synchronization is not supported.
`0b0001`	Enhanced Translation Synchronization is not supported.
`0b0010`	FEAT_ETS2 is implemented.
`0b0011`	FEAT_ETS3 is implemented.

All other values are reserved.

FEAT_ETS2 implements the functionality identified by the value `0b0010`.

FEAT_ETS3 implements the functionality identified by the value `0b0011`.

From Armv8.8, the values `0b0000` and `0b0001` are not permitted.

From Armv9.5, the value `0b0010` is not permitted.

Access to this field is RO.

Otherwise:

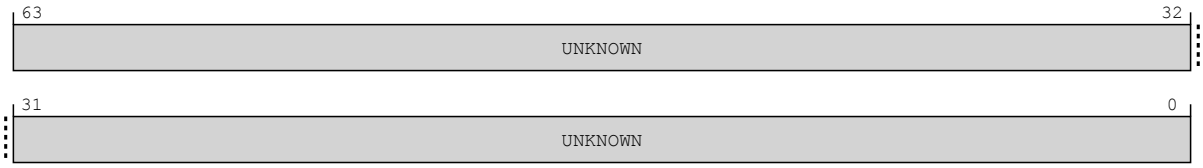

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR5_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR5_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR5_EL1()) ||
     ↪ImpDefBool("ID_MMFR5_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR5_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_MMFR5_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_MMFR5_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.113 ID_PFR0_EL1, AArch32 Processor Feature Register 0

The ID_PFR0_EL1 characteristics are:

Purpose

Gives top-level information about the instruction sets supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1_EL1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_PFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_PFR0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_PFR0_EL1 are UNDEFINED.

Attributes

ID_PFR0_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

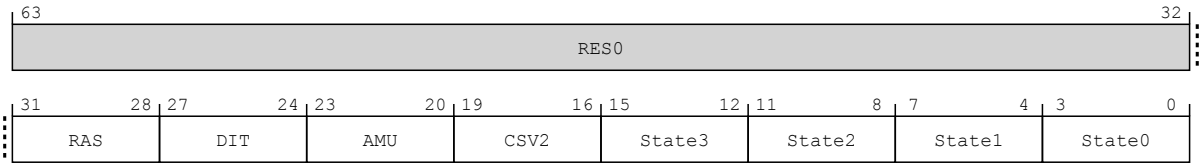

Bits [63:32]

Reserved, RES0.

RAS, bits [31:28]

RAS Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS	Meaning
0b0000	No RAS Extension.
0b0001	Support for the Reliability, Availability, and Serviceability Extension is implemented. The ESB instruction and the Error synchronization event are supported.
0b0010	As 0b0001, and adds support for additional ERXMISC<m> System registers. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.
0b0011	As 0b0010, and requires that error records accessed through System registers conform to either RAS System Architecture v1.1 or RAS System Architecture v2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_RAS implements the functionality identified by the value `0b0001`.

FEAT_DoubleFault implements the functionality identified by the value `0b0010`.

When ERRIDR_EL1.NUM is zero:

- If FEAT_DoubleFault is not implemented, then this field is permitted to have the value `0b0001`, regardless of whether FEAT_RASv1p1 or FEAT_RASv2 are implemented.
- If FEAT_DoubleFault is implemented, then this field is permitted to have the value `0b0010`, regardless of whether FEAT_RASv2 is implemented.

When ERRIDR_EL1.NUM is nonzero:

- FEAT_RASv1p1 implements the functionality identified by the value `0b0010`.
- FEAT_RASv2 implements the functionality identified by the value `0b0011`.

From Armv8.2, the value `0b0000` is not permitted.

From Armv8.4, if FEAT_DoubleFault is implemented or ERRIDR_EL1.NUM is nonzero, the value `0b0001` is not permitted.

Note

When the value of this field is `0b0001`, ID_AA64PFR1_EL1.RAS_frac indicates whether FEAT_RASv1p1 is implemented.

From Armv8.9, if ERRIDR_EL1.NUM is nonzero, the value `0b0010` is not permitted.

Access to this field is RO.

DIT, bits [27:24]

Data Independent Timing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DIT	Meaning
`0b0000`	AArch32 does not guarantee constant execution time of any instructions.
`0b0001`	AArch32 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

FEAT_DIT implements the functionality identified by the value `0b0001`.

From Armv8.4, the only permitted value is `0b0001`.

Access to this field is RO.

AMU, bits [23:20]

Indicates support for Activity Monitors Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMU	Meaning
0b0000	Activity Monitors Extension is not implemented.
0b0001	FEAT_AMUv1 is implemented.
0b0010	FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

FEAT_AMUv1 implements the functionality identified by the value 0b0001.

FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0000.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

CSV2, bits [19:16]

Speculative use of out of context branch targets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV2	Meaning
0b0000	The implementation does not disclose whether FEAT_CSV2 is implemented.
0b0001	FEAT_CSV2 is implemented, but FEAT_CSV2_1p1 is not implemented.
0b0010	FEAT_CSV2_1p1 is implemented.

All other values are reserved.

FEAT_CSV2 implements the functionality identified by the value 0b0001.

FEAT_CSV2_1p1 implements the functionality identified by the value 0b0010.

From Armv8.5, the permitted values are 0b0001 and 0b0010.

Access to this field is RO.

State3, bits [15:12]

T32EE instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State3	Meaning
0b0000	Not implemented.
0b0001	T32EE instruction set implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

State2, bits [11:8]

Jazelle extension support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State2	Meaning
0b0000	Not implemented.
0b0001	Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.
0b0010	Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

State1, bits [7:4]

T32 instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State1	Meaning
0b0000	T32 instruction set not implemented.
0b0001	T32 encodings before the introduction of Thumb-2 technology implemented: • All instructions are 16-bit. • A BL or BLX is a pair of 16-bit instructions. • 32-bit instructions other than BL and BLX cannot be encoded.
0b0011	T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0011.

Access to this field is RO.

State0, bits [3:0]

A32 instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State0	Meaning
0b0000	A32 instruction set not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

State0	Meaning
0b0001	A32 instruction set implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Access to this field is RO.

Otherwise:

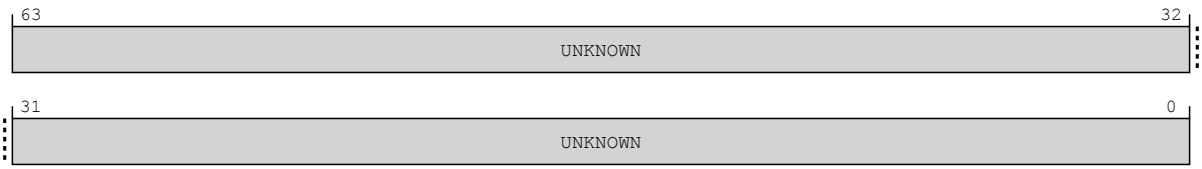

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_PFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = ID_PFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_PFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_PFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.114 ID_PFR1_EL1, AArch32 Processor Feature Register 1

The ID_PFR1_EL1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch64 System register ID_PFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_PFR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_PFR1_EL1 are UNDEFINED.

Attributes

ID_PFR1_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

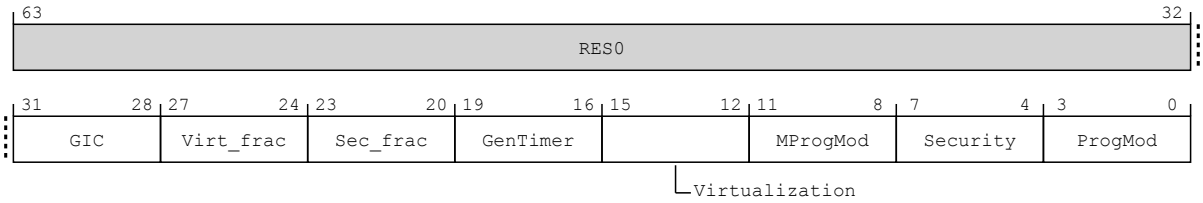

Bits [63:32]

Reserved, RES0.

GIC, bits [31:28]

System register GIC CPU interface.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface system registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Virt_frac, bits [27:24]

Virtualization fractional field. When the Virtualization field is `0b0000`, determines the support for Virtualization Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Virt_frac	Meaning
`0b0000`	No Virtualization Extensions are implemented.
`0b0001`	The following Virtualization Extensions are implemented: • The SCR.SIF bit, if EL3 is implemented. • The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented. • The MSR (banked register) and MRS (banked register) instructions. • The ERET instruction.

All other values are reserved.

In Armv8-A, the permitted values are:

- `0b0000` when EL2 is implemented.
- `0b0001` when EL2 is not implemented.

This field is valid only when the value of ID_PFR1_EL1.Virtualization is 0, otherwise it holds the value `0b0000`.

Note

The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

Access to this field is RO.

Sec_frac, bits [23:20]

Security fractional field. When the Security field is `0b0000`, determines the support for Security Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Sec_frac	Meaning
`0b0000`	No Security Extensions are implemented.
`0b0001`	The following Security Extensions are implemented: • The VBAR register. • The TTBCR.PD0 and TTBCR.PD1 bits.
`0b0010`	As for `0b0001`, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8-A, the permitted values are:

- `0b0000` when EL3 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `0b0001` or `0b0010` when EL3 is not implemented.

This field is valid only when the value of ID_PFR1_EL1.Security is 0, otherwise it holds the value `0b0000`.

Access to this field is RO.

GenTimer, bits [19:16]

Generic Timer support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GenTimer	Meaning
`0b0000`	Generic Timer is not implemented.
`0b0001`	Generic Timer is implemented.
`0b0010`	Generic Timer is implemented, and also includes support for CNTHCTL.EVNTIS and CNTKCTL.EVNTIS fields, and CNTPCTSS and CNTVCTSS counter views.

All other values are reserved.

FEAT_ECV implements the functionality identified by the value `0b0010`.

In Armv8.0, the only permitted value is `0b0001`.

From Armv8.6, the only permitted value is `0b0010`.

Access to this field is RO.

Virtualization, bits [15:12]

Virtualization support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Virtualization	Meaning
`0b0000`	EL2, Hyp mode, and the HVC instruction not implemented.
`0b0001`	EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == `0b0001` implemented.

All other values are reserved.

In Armv8-A, the permitted values are:

- `0b0000` when EL2 is not implemented.
- `0b0001` when EL2 is implemented.

In an implementation that includes EL2, if EL2 cannot use AArch32 but EL1 can use AArch32, then this field has the value `0b0001`.

If EL1 cannot use AArch32, then this field has the value `0b0000`.

Note

The ID_ISARs do not identify whether the HVC instruction is implemented.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MProgMod, bits [11:8]

M-profile programmers' model support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MProgMod	Meaning
0b0000	Not supported.
0b0010	Support for two-stack programmers' model.

All other values are reserved.

In Armv8-A the only permitted value is 0b0000.

Access to this field is RO.

Security, bits [7:4]

Security support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Security	Meaning
0b0000	EL3, Monitor mode, and the SMC instruction not implemented.
0b0001	EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0b0001 implemented.
0b0010	As for 0b0001, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 when EL3 is not implemented.
- 0b0001 when EL3 is implemented.

In an implementation that includes EL3, if EL3 cannot use AArch32 but EL1 can use AArch32, then this field has the value 0b0001.

If EL1 cannot use AArch32, then this field has the value 0b0000.

Access to this field is RO.

ProgMod, bits [3:0]

Support for the standard programmers' model for Armv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ProgMod	Meaning
0b0000	Not supported.
0b0001	Supported.

All other values are reserved.

In Armv8-A, the permitted values are 0b0001 and 0b0000.

If EL1 cannot use AArch32, then this field has the value 0b0000.

Access to this field is RO.

Otherwise:

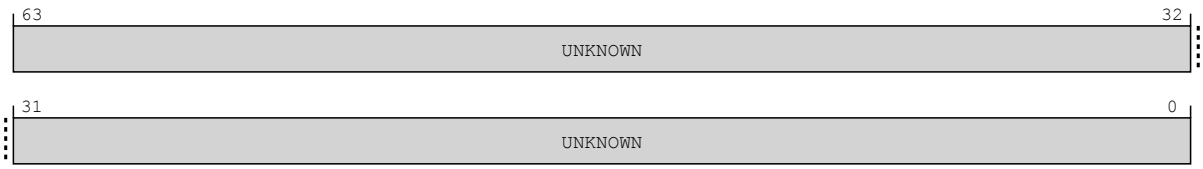

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_PFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_PFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_PFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_PFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.115 ID_PFR2_EL1, AArch32 Processor Feature Register 2

The ID_PFR2_EL1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1 and ID_PFR1_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch64 System register ID_PFR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_PFR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ID_PFR2_EL1 are UNDEFINED.

Attributes

ID_PFR2_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

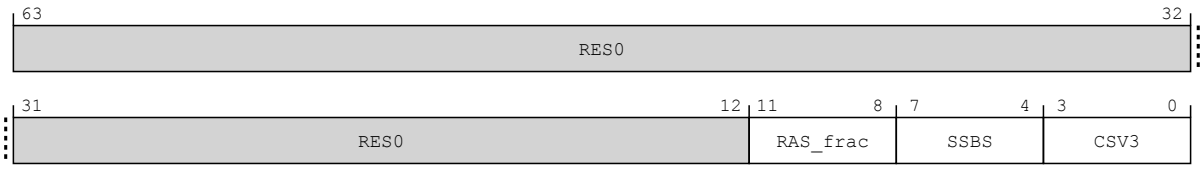

Bits [63:12]

Reserved, RES0.

RAS_frac, bits [11:8]

RAS Extension fractional field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS_frac	Meaning
0b0000	If ID_PFR0_EL1.RAS == 0b0001, Support for the Reliability, Availability, and Serviceability Extension is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RAS_frac	Meaning
0b0001	If ID_PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for additional ERXMISC<m> System registers. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0000.

When ERRIDR_EL1.NUM is zero, this field is permitted to have the value 0b0000, regardless of whether FEAT_RASv1p1 is implemented.

When ERRIDR_EL1.NUM is nonzero, FEAT_RASv1p1 implements the functionality identified by the value 0b0001.

This field is valid only if ID_PFR0_EL1.RAS == 0b0001.

Access to this field is RO.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SSBS	Meaning
0b0000	AArch32 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001	AArch32 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

All other values are reserved.

Access to this field is RO.

CSV3, bits [3:0]

Speculative use of faulting data.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV3	Meaning
0b0000	This PE does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by other instructions in the speculative sequence.
0b0001	Data loaded under speculation with a permission or domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_CSV3 implements the functionality identified by the value `0b0001`.

In Armv8.0, the permitted values are `0b0000` and `0b0001`.

From Armv8.5, the only permitted value is `0b0001`.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

Access to this field is RO.

Otherwise:

63 … 32	31 … 0
UNKNOWN	UNKNOWN

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_PFR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_PFR2_EL1()) ||
     ↪ImpDefBool("ID_PFR2_EL1 trapped by HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = ID_PFR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ID_PFR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ID_PFR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.116 IFSR32_EL2, Instruction Fault Status Register (EL2)

The IFSR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 IFSR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configuration

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

AArch64 System register IFSR32_EL2 bits [31:0] are architecturally mapped to AArch32 System register IFSR[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to IFSR32_EL2 are UNDEFINED.

Attributes

IFSR32_EL2 is a 64-bit register.

Field descriptions

When TTBCR.EAE == '0':

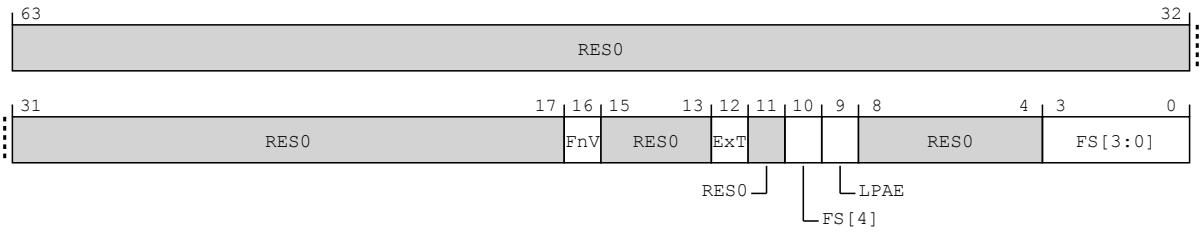

Bits [63:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	IFAR is valid.
0b1	IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]

Fault Status bits. Bits [10] and [3:0] are interpreted together.

FS	Meaning	Applies when
0b00001	PC alignment fault.	
0b00010	Debug exception.	
0b00011	Access flag fault, level 1.	
0b00101	Translation fault, level 1.	
0b00110	Access flag fault, level 2.	
0b00111	Translation fault, level 2.	
0b01000	Synchronous External abort, not on translation table walk.	
0b01001	Domain fault, level 1.	
0b01011	Domain fault, level 2.	
0b01100	Synchronous External abort, on translation table walk, level 1.	
0b01101	Permission fault, level 1.	
0b01110	Synchronous External abort, on translation table walk, level 2.	
0b01111	Permission fault, level 2.	
0b10000	TLB conflict abort.	
0b10100	IMPLEMENTATION DEFINED fault (Lockdown fault).	
0b11001	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b11100	Synchronous parity or ECC error on translation table walk, level 1.	FEAT_RAS is not implemented
0b11110	Synchronous parity or ECC error on translation table walk, level 2.	FEAT_RAS is not implemented

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Short-descriptor translation table lookup'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

LPAE	Meaning
0b0	Using the Short-descriptor translation table formats.
0b1	Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

When TTBCR.EAE == '1':

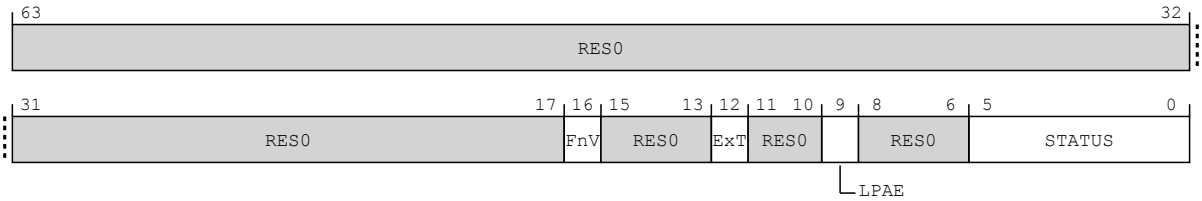

Bits [63:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	IFAR is valid.
0b1	IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

LPAE	Meaning
`0b0`	Using the Short-descriptor translation table formats.
`0b1`	Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

STATUS	Meaning	Applies when
`0b000000`	Address size fault in translation table base register.	
`0b000001`	Address size fault, level 1.	
`0b000010`	Address size fault, level 2.	
`0b000011`	Address size fault, level 3.	
`0b000101`	Translation fault, level 1.	
`0b000110`	Translation fault, level 2.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STATUS	Meaning	Applies when
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk.	
0b010101	Synchronous External abort on translation table walk, level 1.	
0b010110	Synchronous External abort on translation table walk, level 2.	
0b010111	Synchronous External abort on translation table walk, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3.	FEAT_RAS is not implemented
0b100001	PC alignment fault.	
0b100010	Debug exception.	
0b110000	TLB conflict abort.	

All other values are reserved.

When FEAT_RAS is implemented, 0b011000, 0b011101, 0b011110, and 0b011111 are reserved.

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Long-descriptor translation table lookup'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing IFSR32_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, IFSR32_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = IFSR32_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = IFSR32_EL2();
end;
```

```
MSR IFSR32_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    IFSR32_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    IFSR32_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.117 ISR_EL1, Interrupt Status Register

The ISR_EL1 characteristics are:

Purpose

Shows the pending status of IRQ and FIQ interrupts and SError exceptions.

When FEAT_NMI is implemented, also shows whether a pending IRQ or FIQ interrupt has Superpriority.

Configuration

AArch64 System register ISR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ISR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to ISR_EL1 are UNDEFINED.

Attributes

ISR_EL1 is a 64-bit register.

Field descriptions

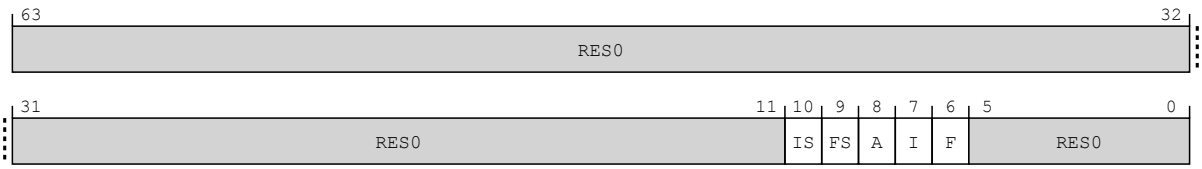

Bits [63:11]

Reserved, RES0.

IS, bit [10]

When FEAT_NMI is implemented:

IRQ with Superpriority pending bit. Indicates whether an IRQ interrupt with Superpriority is pending.

IS	Meaning
0b0	No pending IRQ with Superpriority.
0b1	An IRQ interrupt with Superpriority is pending.

If all of the following apply, then this field shows the pending status of virtual IRQ interrupts with Superpriority:

- EL2 is implemented and enabled in the current Security state.
- HCR_EL2.IMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical IRQ interrupts with Superpriority.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FS, bit [9]

When FEAT_NMI is implemented:

FIQ with Superpriority pending bit. Indicates whether an FIQ interrupt with Superpriority is pending.

FS	Meaning
0b0	No pending FIQ with Superpriority.
0b1	An FIQ interrupt with Superpriority is pending.

If all of the following apply, then this field shows the pending status of virtual FIQ interrupts with Superpriority:

- EL2 is implemented and enabled in the current Security state.
- HCR_EL2.FMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical FIQ interrupts with Superpriority.

Otherwise:

Reserved, RES0.

A, bit [8]

SError exception pending bit. Indicates whether an SError exception is pending.

A	Meaning
0b0	No pending SError.
0b1	An SError exception is pending.

If all of the following apply, then this field shows the pending status of virtual SError exceptions:

- EL2 is implemented and enabled in the current Security state.
- Any of the following apply:
  - HCR_EL2.AMO is 1.
  - FEAT_DoubleFault2 is implemented and the Effective value of HCRX_EL2.TMEA is 1.
- The PE is executing at EL1.

Otherwise, if all of the following apply, then this field shows the pending status of delegated SError exceptions:

- FEAT_E3DSE is implemented.
- SCR_EL3.EnDSE is 1.
- The PE is executing at EL2 or EL1.

Otherwise, this field shows the pending status of physical SError exceptions.

If the physical SError exception is edge-triggered, this field is cleared to zero when the physical SError exception is taken.

I, bit [7]

IRQ pending bit. Indicates whether an IRQ interrupt is pending.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I	Meaning
0b0	No pending IRQ.
0b1	An IRQ interrupt is pending.

If all of the following apply, then this field shows the pending status of virtual IRQ interrupts:

- EL2 is implemented and enabled in the current Security state.
- HCR_EL2.IMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical IRQ interrupts.

Note

This bit indicates the presence of a pending IRQ interrupt regardless of whether the interrupt has Superpriority.

F, bit [6]

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

F	Meaning
0b0	No pending FIQ.
0b1	An FIQ interrupt is pending.

If all of the following apply, then this field shows the pending status of virtual FIQ interrupts:

- EL2 is implemented and enabled in the current Security state.
- HCR_EL2.FMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical FIQ interrupts.

Note

This bit indicates the presence of a pending FIQ interrupt regardless of whether the interrupt has Superpriority.

Bits [5:0]

Reserved, RES0.

Accessing ISR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ISR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ISR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = ISR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = ISR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = ISR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.118 LORC_EL1, LORegion Control (EL1)

The LORC_EL1 characteristics are:

Purpose

Enables and disables LORegions, and selects the current LORegion descriptor.

Configuration

If no LORegion descriptors are supported by the PE, then this register is RES0.

This register is present only when FEAT_LOR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to LORC_EL1 are UNDEFINED.

Attributes

LORC_EL1 is a 64-bit register.

Field descriptions

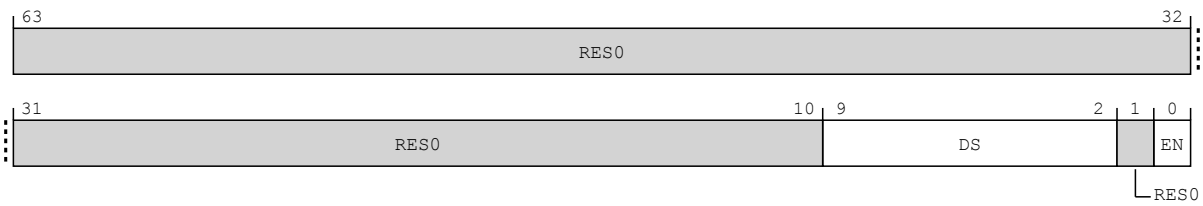

Bits [63:10]

Reserved, RES0.

DS, bits [9:2]

Descriptor Select. Selects the current LORegion descriptor accessed by LORSA_EL1, LOREA_EL1, and LORN_EL1.

If this field points to an LORegion descriptor that is not supported by an implementation, then the registers LORN_EL1, LOREA_EL1, and LORSA_EL1 are RES0.

The number of LORegion descriptors in IMPLEMENTATION DEFINED. The maximum number of LORegion descriptors supported is 256. If the number is less than 256, then bits[63:M+2] are RES0, where M is Log_2(Number of LORegion descriptors supported by the implementation).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [1]

Reserved, RES0.

EN, bit [0]

Enable. Indicates whether LORegions are enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EN	Meaning
0b0	Disabled. Memory accesses do not match any LORegions.
0b1	Enabled. Memory accesses may match a LORegion.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing LORC_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, LORC_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().LORC_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORC_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORC_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        X{64}(t) = LORC_EL1();
    end;
end;
```

MSR LORC_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().LORC_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LORC_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LORC_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        LORC_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.119 LOREA_EL1, LORegion End Address (EL1)

The LOREA_EL1 characteristics are:

Purpose

Holds the physical address of the end of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Configuration

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

This register is present only when FEAT_LOR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to LOREA_EL1 are UNDEFINED.

Attributes

LOREA_EL1 is a 64-bit register.

Field descriptions

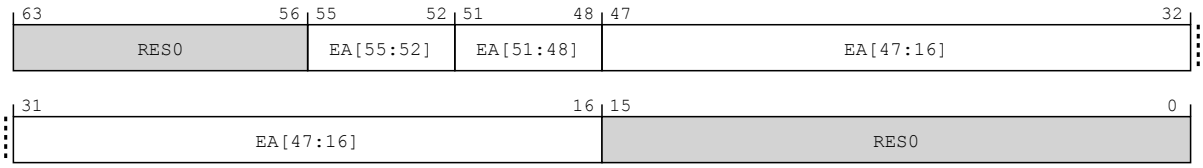

Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:56]

Reserved, RES0.

EA[55:52], bits [55:52]

When FEAT_D128 is implemented:

Extension to EA[47:16]. For more information, see EA[47:16].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to EA[47:16]. For more information, see EA[47:16].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EA[47:16], bits [47:16]

Bits [47:16] of the end physical address of an LORegion described in the current LORegion descriptor selected by LORC_EL1.DS. Bits[15:0] of this address are `0xFFFF`. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When FEAT_LPA is implemented and 52-bit addresses are in use, EA[51:48] form bits [51:48] of the end physical address of the LORegion. Otherwise, when 52-bit addresses are not in use, EA[51:48] is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:0]

Reserved, RES0.

Accessing LOREA_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, LOREA_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().LOREA_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LOREA_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LOREA_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        X{64}(t) = LOREA_EL1();
    end;
end;
```

```
MSR LOREA_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().LOREA_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LOREA_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LOREA_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        LOREA_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.120 LORID_EL1, LORegionID (EL1)

The LORID_EL1 characteristics are:

Purpose

Indicates the number of LORegions and LORegion descriptors supported by the PE.

Configuration

If no LORegion descriptors are implemented, then the registers LORC_EL1, LORN_EL1, LOREA_EL1, and LORSA_EL1 are RES0.

This register is present only when FEAT_LOR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to LORID_EL1 are UNDEFINED.

Attributes

LORID_EL1 is a 64-bit register.

Field descriptions

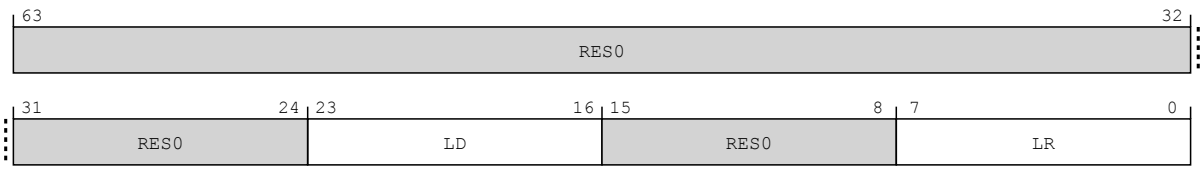

Bits [63:24]

Reserved, RES0.

LD, bits [23:16]

Number of LORegion descriptors supported by the PE. This is an 8-bit binary number.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LD	Meaning
0x00..0xFF	The number of LORegions descriptors supported.

Access to this field is RO.

Bits [15:8]

Reserved, RES0.

LR, bits [7:0]

Number of LORegions supported by the PE. This is an 8-bit binary number.

Note

If LORID_EL1 indicates that no LORegions are implemented, then LoadLOAcquire and StoreLORelease will behave as LoadAcquire and StoreRelease.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

LR	Meaning
0x00..0xFF	The number of LORegions supported.

Access to this field is RO.

Accessing LORID_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, LORID_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().LORID_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORID_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORID_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = LORID_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.121 LORN_EL1, LORegion Number (EL1)

The LORN_EL1 characteristics are:

Purpose

Holds the number of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Configuration

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

This register is present only when FEAT_LOR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to LORN_EL1 are UNDEFINED.

Attributes

LORN_EL1 is a 64-bit register.

Field descriptions

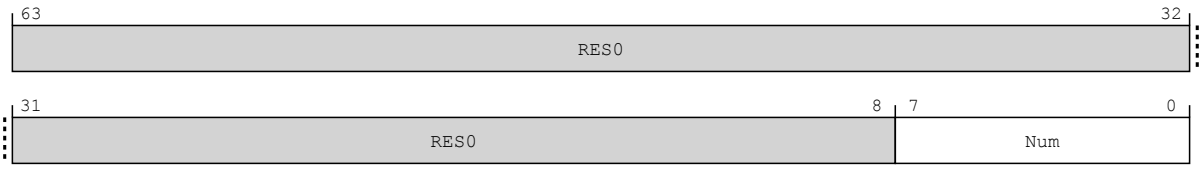

Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:8]

Reserved, RES0.

Num, bits [7:0]

Number of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

The maximum number of LORegions supported by the PE is 256. If the maximum number is less than 256, then bits[8:N] are RES0, where N is (Log_2(Number of LORegions supported by the PE)).

If this field points to a LORegion that is not supported by the PE, then the current LORegion descriptor does not match any LORegion.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing LORN_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, LORN_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().LORN_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORN_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORN_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        X{64}(t) = LORN_EL1();
    end;
end;
```

MSR LORN_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().LORN_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LORN_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LORN_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        LORN_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.122 LORSA_EL1, LORegion Start Address (EL1)

The LORSA_EL1 characteristics are:

Purpose

Indicates whether the current LORegion descriptor selected by LORC_EL1.DS is enabled, and holds the physical address of the start of the LORegion.

Configuration

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

This register is present only when FEAT_LOR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to LORSA_EL1 are UNDEFINED.

Attributes

LORSA_EL1 is a 64-bit register.

Field descriptions

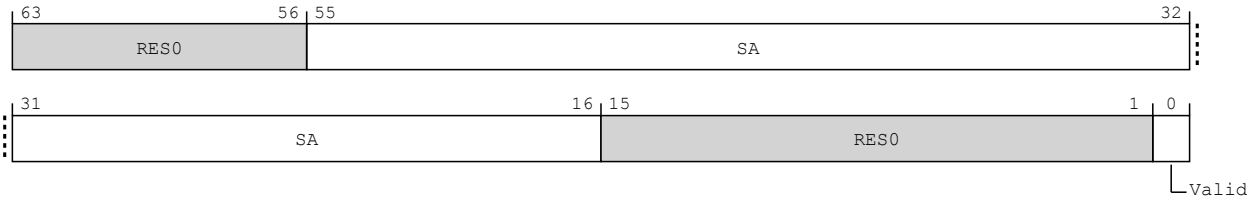

Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:56]

Reserved, RES0.

SA, bits [55:16]

When FEAT_D128 is implemented:

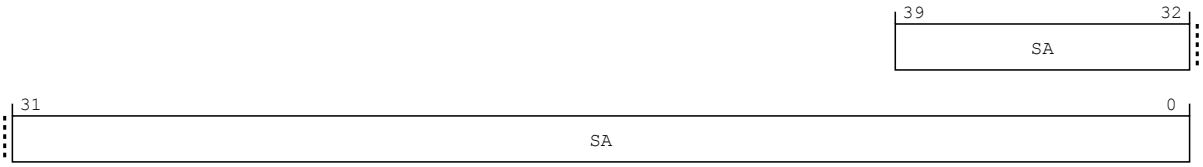

SA, bits [39:0]

Bits [55:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Bits[15:0] of this address are `0x0000`.

For implementations with fewer than 56 physical address bits, the corresponding upper bits of this field are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LPA is implemented and FEAT_D128 is not implemented:

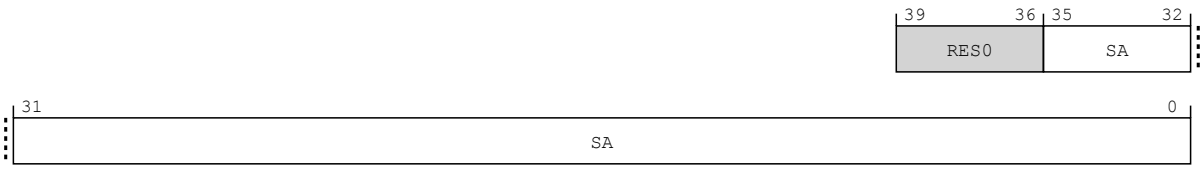

Bits [39:36]

Reserved, RES0.

SA, bits [35:0]

Bits [51:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Bits[15:0] of this address are `0x0000`.

For implementations with fewer than 52 physical address bits, the corresponding upper bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LPA is not implemented:

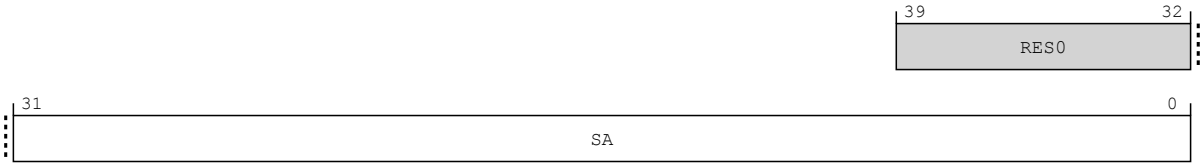

Bits [39:32]

Reserved, RES0.

SA, bits [31:0]

Bits [47:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Bits[15:0] of this address are `0x0000`.

For implementations with fewer than 48 physical address bits, the corresponding upper bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:1]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Valid, bit [0]

Indicates whether the current LORegion descriptor is enabled.

Valid	Meaning
0b0	LORegion descriptor is disabled.
0b1	LORegion descriptor is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing LORSA_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, LORSA_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().LORSA_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = LORSA_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = LORSA_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        X{64}(t) = LORSA_EL1();
    end;
end;
```

```
MSR LORSA_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_LOR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TLOR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().LORSA_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LORSA_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TLOR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TLOR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        LORSA_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().NS == '0' then
        Undefined();
    else
        LORSA_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.123 MAIR2_EL1, Extended Memory Attribute Indirection Register (EL1)

The MAIR2_EL1 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a VMSAv8-64 or VMSAv9-128 translation table entry for stage 1 translations at EL1.

Configuration

This register is present only when FEAT_AIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MAIR2_EL1 are UNDEFINED.

Attributes

MAIR2_EL1 is a 64-bit register.

Field descriptions

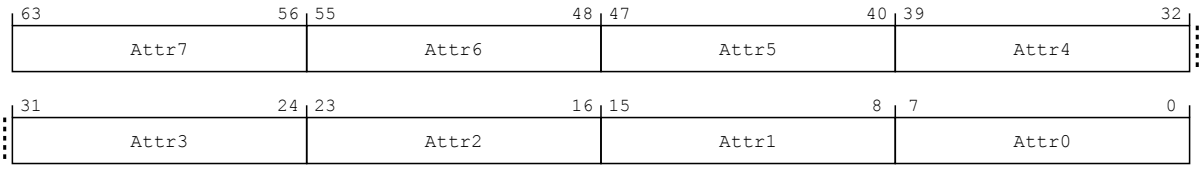

Attr<n> , bits [8n+7:8n], for n = 7 to 0

Memory Attribute encoding.

When stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a VMSAv8-64 or VMSAv9-128 translation table entry is 1, AttrIndx[2:0] gives the value of <n> in Attr<n>.

When stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a VMSAv8-64 or VMSAv9-128 translation table entry is 0, see MAIR_EL1.Attr

Attr is encoded as follows:

Attr		Meaning
0b0000dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
0b0000dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
0b0000dd1x		UNPREDICTABLE.
0booooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
0b01000000		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b10100000		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR2_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name MAIR2_EL1 or MAIR2_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MAIR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x280);
    else
        X{64}(t) = MAIR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = MAIR2_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = MAIR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR2_EL1();
end;
```

MSR MAIR2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGWTR_EL2().nMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x280) = X{64}(t);
    else
        MAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        MAIR2_EL2() = X{64}(t);
    else
        MAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MAIR2_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, MAIR2_EL12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x280);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = MAIR2_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = MAIR2_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR MAIR2_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x280) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            MAIR2_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        MAIR2_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.124 MAIR2_EL2, Extended Memory Attribute Indirection Register (EL2)

The MAIR2_EL2 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a VMSAv8-64 or VMSAv9-128 translation table entry for stage 1 translations at EL1.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_AIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MAIR2_EL2 are UNDEFINED.

Attributes

MAIR2_EL2 is a 64-bit register.

Field descriptions

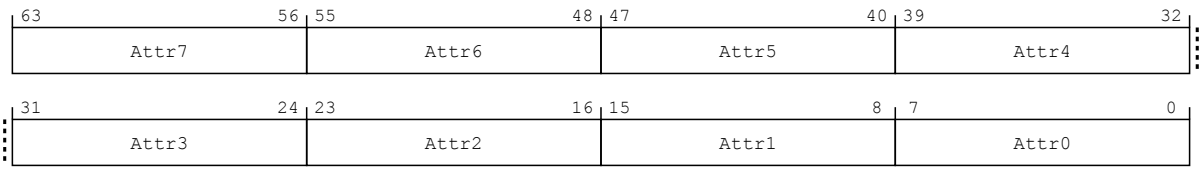

Attr<n> , bits [8n+7:8n], for n = 7 to 0

Memory Attribute encoding.

When stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a VMSAv8-64 or VMSAv9-128 translation table entry is 1, AttrIndx[2:0] gives the value of <n> in Attr<n>.

When stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a VMSAv8-64 or VMSAv9-128 translation table entry is 0, see MAIR_EL2.Attr

Attr is encoded as follows:

Attr		Meaning
`0b0000`dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
`0b0000`dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
`0b0000`dd1x		UNPREDICTABLE.
0boooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
`0b01000000`		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
`0b10100000`		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR2_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name MAIR2_EL2 or MAIR2_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MAIR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MAIR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR2_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MAIR2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MAIR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MAIR2_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, MAIR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x280);
    else
        X{64}(t) = MAIR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = MAIR2_EL2();
    else
        X{64}(t) = MAIR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR2_EL1();
end;
```

MSR MAIR2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nMAIR2_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x280) = X{64}(t);
    else
        MAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        MAIR2_EL2() = X{64}(t);
    else
        MAIR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MAIR2_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.125 MAIR2_EL3, Extended Memory Attribute Indirection Register (EL3)

The MAIR2_EL3 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a VMSAv8-64 or VMSAv9-128 translation table entry for stage 1 translations at EL1.

Configuration

This register is present only when FEAT_AIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MAIR2_EL3 are UNDEFINED.

Attributes

MAIR2_EL3 is a 64-bit register.

Field descriptions

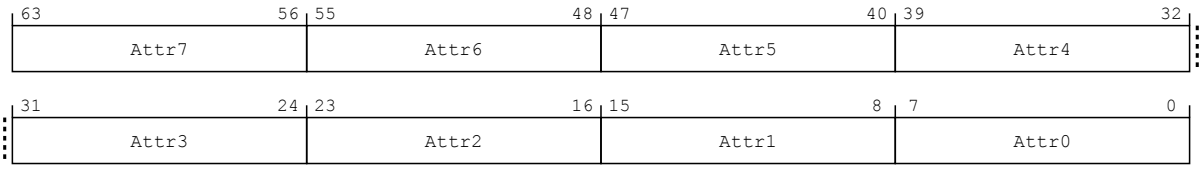

Attr<n> , bits [8n+7:8n], for n = 7 to 0

Memory Attribute encoding.

When stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a VMSAv8-64 or VMSAv9-128 translation table entry is 1, AttrIndx[2:0] gives the value of <n> in Attr<n>.

When stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a VMSAv8-64 or VMSAv9-128 translation table entry is 0, see MAIR_EL3.Attr

Attr is encoded as follows:

Attr		Meaning
0b0000dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
0b0000dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
0b0000dd1x		UNPREDICTABLE.
0booooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
0b01000000		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b10100000		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR2_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MAIR2_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR2_EL3();
end;
```

```
MSR MAIR2_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_AIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().MAIR2_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        MAIR2_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.126 MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

Configuration

When TTBCR.EAE == ‘0’, AArch64 System register MAIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PRRR[31:0].

When TTBCR.EAE == ‘1’, AArch64 System register MAIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MAIR0[31:0].

When TTBCR.EAE == ‘0’, AArch64 System register MAIR_EL1 bits [63:32] are architecturally mapped to AArch32 System register NMRR[31:0].

When TTBCR.EAE == ‘1’, AArch64 System register MAIR_EL1 bits [63:32] are architecturally mapped to AArch32 System register MAIR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MAIR_EL1 are UNDEFINED.

Attributes

MAIR_EL1 is a 64-bit register.

Field descriptions

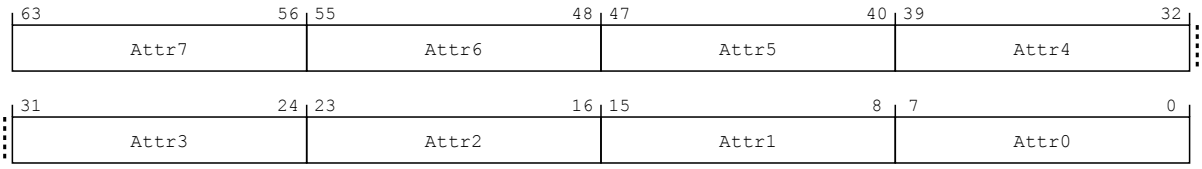

MAIR_EL1 is permitted to be cached in a TLB.

Attr<n> , bits [8n+7:8n], for n = 7 to 0

Memory Attribute encoding.

When FEAT_AIE is implemented and stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a Long descriptor format translation table entry is 0, or when FEAT_AIE is not implemented, AttrIndx[2:0] gives the value of <n> in Attr<n>.

When FEAT_AIE is implemented and stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a Long descriptor format translation table entry is 1, see MAIR2_EL1.Attr

Attr is encoded as follows:

Attr	Meaning
0b0000dd00	Device memory. See encoding of ‘dd’ for the type of Device memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0b0000dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
0b0000dd1x		UNPREDICTABLE.
0booooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
0b01000000		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b10100000		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

‘dd’	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

‘oooo’		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name MAIR_EL1 or MAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MAIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().MAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x140);
    else
        X{64}(t) = MAIR_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = MAIR_EL2();
    else
        X{64}(t) = MAIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR_EL1();
end;
```

MSR MAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().MAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x140) = X{64}(t);
    else
        MAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        MAIR_EL2() = X{64}(t);
    else
        MAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MAIR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, MAIR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x140);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            X{64}(t) = MAIR_EL1();
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = MAIR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR MAIR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x140) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        MAIR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        MAIR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.127 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register MAIR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HMAIR0[31:0].

AArch64 System register MAIR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HMAIR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MAIR_EL2 are UNDEFINED.

Attributes

MAIR_EL2 is a 64-bit register.

Field descriptions

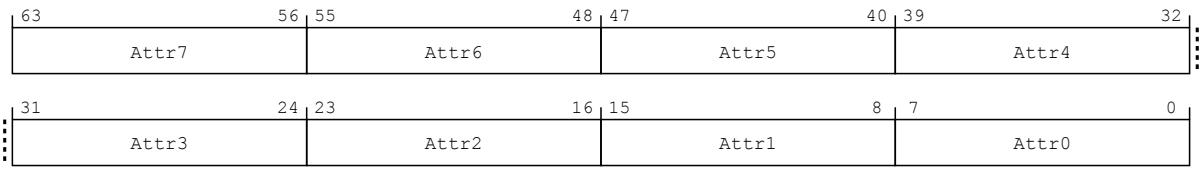

MAIR_EL2 is permitted to be cached in a TLB.

Attr<n> , bits [8n+7:8n], for n = 7 to 0

Memory Attribute encoding.

When FEAT_AIE is implemented and stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a Long descriptor format translation table entry is 0, or when FEAT_AIE is not implemented, AttrIndx[2:0] gives the value of <n> in Attr<n>.

When FEAT_AIE is implemented and stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a Long descriptor format translation table entry is 1, see MAIR2_EL2.Attr

Attr is encoded as follows:

Attr	Meaning
`0b0000`dd00	Device memory. See encoding of ‘dd’ for the type of Device memory.
`0b0000`dd01	If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
`0b0000`dd1x	UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0boooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
0b01000000		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b10100000		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

‘dd’	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

‘oooo’		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

‘iiii’		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'iiii'		Meaning
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name MAIR_EL2 or MAIR_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MAIR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = MAIR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MAIR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    MAIR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    MAIR_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, MAIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().MAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x140);
    else
        X{64}(t) = MAIR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = MAIR_EL2();
    else
        X{64}(t) = MAIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MAIR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().MAIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x140) = X{64}(t);
    else
        MAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        MAIR_EL2() = X{64}(t);
    else
        MAIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MAIR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.128 MAIR_EL3, Memory Attribute Indirection Register (EL3)

The MAIR_EL3 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MAIR_EL3 are UNDEFINED.

Attributes

MAIR_EL3 is a 64-bit register.

Field descriptions

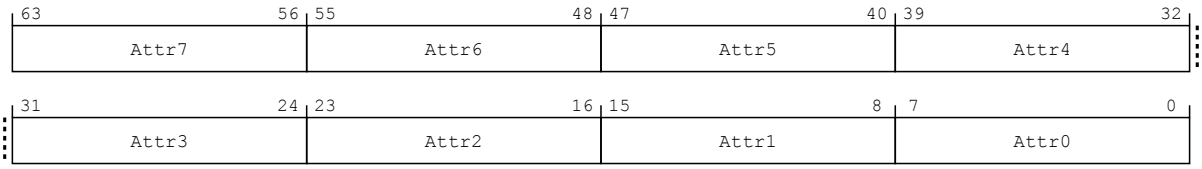

MAIR_EL3 is permitted to be cached in a TLB.

Attr<n> , bits [8n+7:8n], for n = 7 to 0

Memory Attribute encoding.

When FEAT_AIE is implemented and stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a Long descriptor format translation table entry is 0, or when FEAT_AIE is not implemented, AttrIndx[2:0] gives the value of <n> in Attr<n>.

When FEAT_AIE is implemented and stage 1 Attributes Index Extension is enabled and AttrIndx[3] in a Long descriptor format translation table entry is 1, see MAIR2_EL3.Attr

Attr is encoded as follows:

Attr		Meaning
`0b0000`dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
`0b0000`dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
`0b0000`dd1x		UNPREDICTABLE.
0boooo iiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
`0b01000000`		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
`0b10100000`		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MAIR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MAIR_EL3();
end;
```

```
MSR MAIR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().MAIR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        MAIR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.129 MECID_A0_EL2, Alternate MECID for EL2 and EL2&0 translation regimes

The MECID_A0_EL2 characteristics are:

Purpose

Alternate MECID for EL2 and EL2&0 accesses translated by TTBR0_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MECID_A0_EL2 are UNDEFINED.

Attributes

MECID_A0_EL2 is a 64-bit register.

Field descriptions

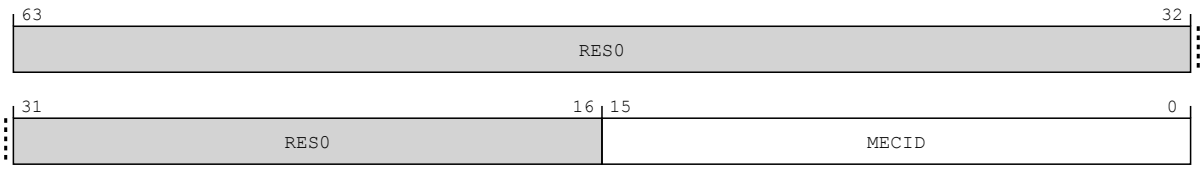

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MECID_A0_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MECID_A0_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MECID_A0_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MECID_A0_EL2();
end;
```

MSR MECID_A0_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b001

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MECID_A0_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MECID_A0_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.130 MECID_A1_EL2, Alternate MECID for EL2&0 translation regimes.

The MECID_A1_EL2 characteristics are:

Purpose

Alternate MECID for EL2&0 accesses translated by TTBR1_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MECID_A1_EL2 are UNDEFINED.

Attributes

MECID_A1_EL2 is a 64-bit register.

Field descriptions

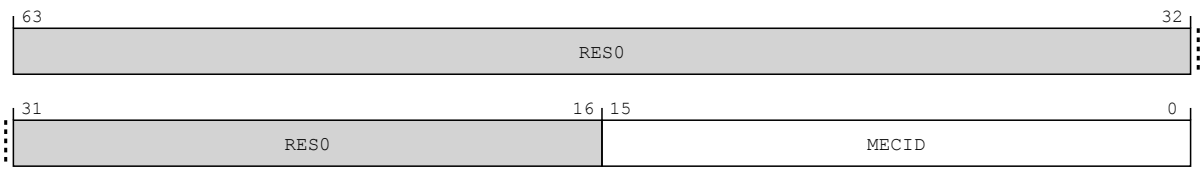

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MECID_A1_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MECID_A1_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MECID_A1_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MECID_A1_EL2();
end;
```

MSR MECID_A1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b011

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MECID_A1_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MECID_A1_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.131 MECID_P0_EL2, Primary MECID for EL2 and EL2&0 translation regimes

The MECID_P0_EL2 characteristics are:

Purpose

Primary MECID for EL2 and EL2&0 accesses translated by TTBR0_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MECID_P0_EL2 are UNDEFINED.

Attributes

MECID_P0_EL2 is a 64-bit register.

Field descriptions

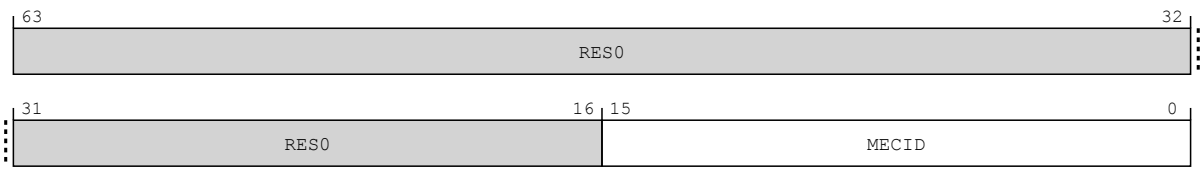

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MECID_P0_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MECID_P0_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MECID_P0_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MECID_P0_EL2();
end;
```

MSR MECID_P0_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b000

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MECID_P0_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MECID_P0_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.132 MECID_P1_EL2, Primary MECID for EL2&0 translation regimes

The MECID_P1_EL2 characteristics are:

Purpose

Primary MECID for EL2&0 accesses translated by TTBR1_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MECID_P1_EL2 are UNDEFINED.

Attributes

MECID_P1_EL2 is a 64-bit register.

Field descriptions

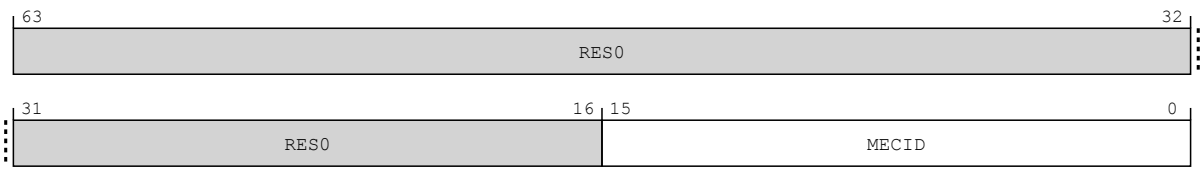

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MECID_P1_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MECID_P1_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MECID_P1_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MECID_P1_EL2();
end;
```

MSR MECID_P1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b010

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MECID_P1_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MECID_P1_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.133 MECID_RL_A_EL3, Realm PA space Alternate MECID for EL3 stage 1 translation regime

The MECID_RL_A_EL3 characteristics are:

Purpose

Alternate MECID for EL3 accesses to the Realm physical address space, translated by TTBR0_EL3.

Configuration

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MECID_RL_A_EL3 are UNDEFINED.

Attributes

MECID_RL_A_EL3 is a 64-bit register.

Field descriptions

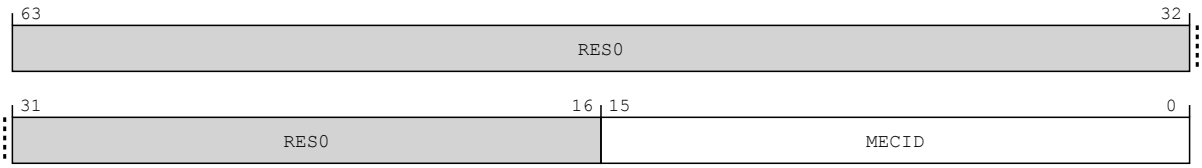

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MECID_RL_A_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MECID_RL_A_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b1010	0b001

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MECID_RL_A_EL3();
end;
```

```
MSR MECID_RL_A_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b1010	0b001

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().MECID_RL_A_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        MECID_RL_A_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.134 MECIDR_EL2, MEC Identification Register

The MECIDR_EL2 characteristics are:

Purpose

MEC identification register.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MECIDR_EL2 are UNDEFINED.

Attributes

MECIDR_EL2 is a 64-bit register.

Field descriptions

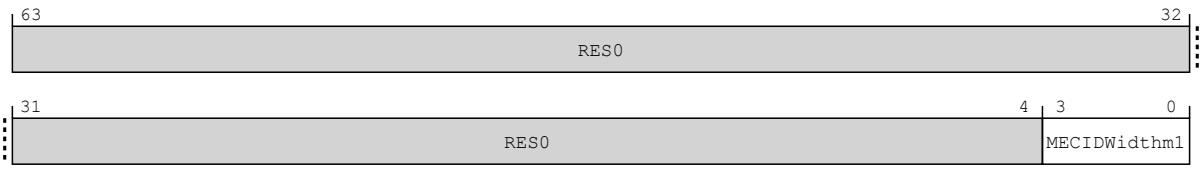

Bits [63:4]

Reserved, RES0.

MECIDWidthm1, bits [3:0]

MECID width minus 1.

The value of this field plus 1 is the MECID width in bits, that this PE supports.

This field has an IMPLEMENTATION DEFINED value.

For example, the value `0b1111` indicates that this PE supports a MECID width of 16 bits, and provides 2^{16} possible MECID values.

MECIDWidth is defined as MECIDR_EL2.MECIDWidthm1 + 1.

Access to this field is RO.

Accessing MECIDR_EL2

For accesses from EL2 and EL3, this register is RO.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MECIDR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1000	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    X{64}(t) = MECIDR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MECIDR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.135 MFAR_EL3, Physical Fault Address Register (EL3)

The MFAR_EL3 characteristics are:

Purpose

Records the faulting physical address for a Granule Protection Check, synchronous External abort, or SError exception taken to EL3.

Configuration

This register is present only when (FEAT_PFAR is implemented or FEAT_RME is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to MFAR_EL3 are UNDEFINED.

Attributes

MFAR_EL3 is a 64-bit register.

Field descriptions

When FEAT_RME is implemented and the exception is a GPC exception:

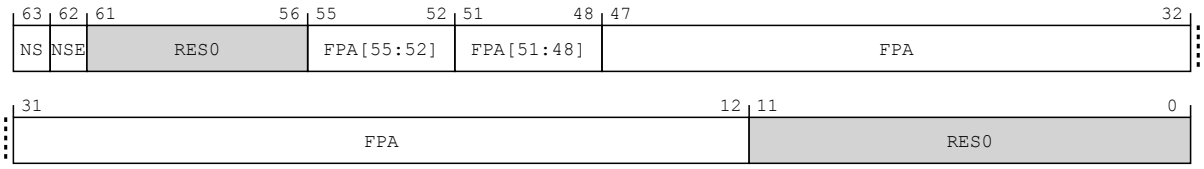

NS, bit [63]

Together with MFAR_EL3.NSE, reports the physical address space of the access that triggered the exception.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSE, bit [62]

Together with MFAR_EL3.NS, reports the physical address space of the access that triggered the exception.

For a description of the values derived by evaluating NS and NSE together, see MFAR_EL3.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [61:56]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPA[55:52], bits [55:52]

When FEAT_D128 is implemented:

When FEAT_D128 is implemented, extension to MFAR_EL3.FPA[47:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FPA[51:48], bits [51:48]

When FEAT_LPA is implemented:

When FEAT_LPA is implemented, extension to MFAR_EL3.FPA[47:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FPA, bits [47:12]

Bits [47:12] of the Faulting Physical Address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

When FEAT_PFAR is implemented and the exception is a synchronous External abort or SError exception:

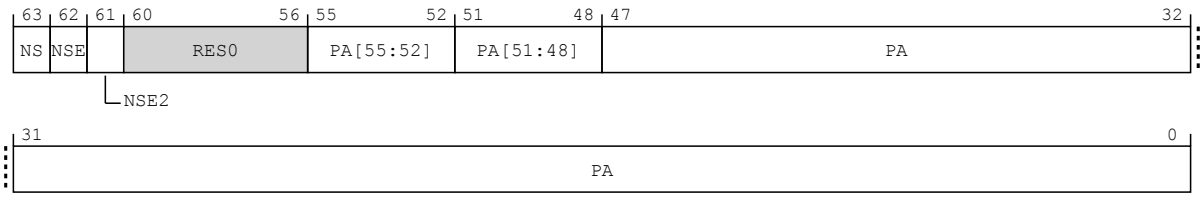

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with MFAR_EL3.NSE and MFAR_EL3.NSE2, reports the physical address space of the access that triggered the exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b0	0b1	Non-secure.
0b0	0b1	0b0	Root.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RME is implemented:

Together with MFAR_EL3.NSE, reports the physical address space of the access that triggered the exception.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Non-secure. Reports the physical address space of the access that triggered the exception.

NS	Meaning
0b0	Secure physical address space.
0b1	Non-secure physical address space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSE, bit [62]

When FEAT_RME is implemented:

Together with MFAR_EL3.NS, reports the physical address space of the access that triggered the exception.

For a description of the values derived by evaluating NS and NSE together, see MFAR_EL3.NS.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSE2, bit [61]

When FEAT_RME_GDI is implemented:

Together with MFAR_EL3.NS and MFAR_EL3.NSE, reports the physical address space of the access that triggered the exception.

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see MFAR_EL3.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

When FEAT_D128 is implemented, extension to MFAR_EL3.PA[47:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:

When FEAT_LPA is implemented, extension to MFAR_EL3.PA[47:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA, bits [47:0]

Physical Address. Bits [47:0] of the aborting physical address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The recorded address can be any address within the same naturally-aligned fault granule as the faulting physical address, where the size of the fault granule is IMPLEMENTATION DEFINED and no larger than the larger than:

- The size of the range of values permitted to be recorded in FAR_EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing MFAR_EL3

MFAR_EL3 is not valid and reads UNKNOWN if ESR_EL3.EC is recorded indicating an Abort or SError exception and ESR_EL3.PFV is recorded as 0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MFAR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b101

```
if !((IsFeatureImplemented(FEAT_PFAR) || IsFeatureImplemented(FEAT_RME)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MFAR_EL3();
end;
```

```
MSR MFAR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b101

```
if !((IsFeatureImplemented(FEAT_PFAR) || IsFeatureImplemented(FEAT_RME)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    MFAR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.136 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

AArch64 System register MIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MIDR[31:0].

AArch64 System register MIDR_EL1 bits [31:0] are architecturally mapped to External register MIDR_EL1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MIDR_EL1 are UNDEFINED.

Attributes

MIDR_EL1 is a 64-bit register.

Field descriptions

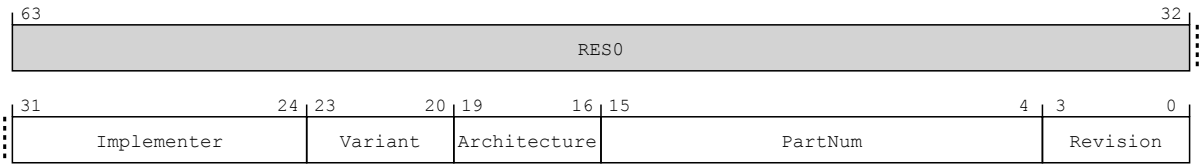

Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Implementer	Meaning
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGRTR_EL2().MIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() then
        X{64}(t) = VPIDR_EL2();
    else
        X{64}(t) = MIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = MIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.137 MPIDR_EL1, Multiprocessor Affinity Register

The MPIDR_EL1 characteristics are:

Purpose

In a multiprocessor system, provides an additional PE identification mechanism.

Configuration

In a uniprocessor system, Arm recommends that each Aff<n> field of this register returns a value of 0.

AArch64 System register MPIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MPIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MPIDR_EL1 are UNDEFINED.

Attributes

MPIDR_EL1 is a 64-bit register.

Field descriptions

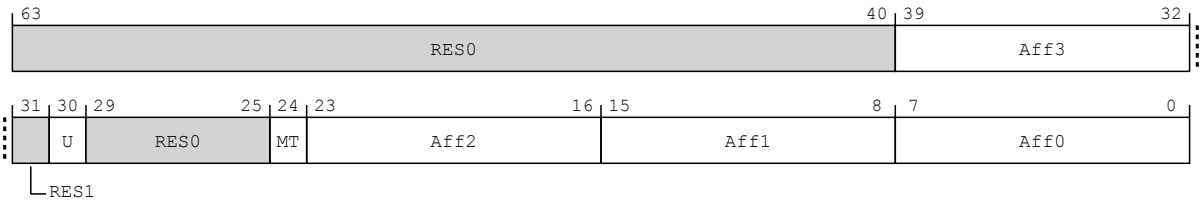

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

Aff3 is not supported in AArch32 state.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, MPIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGRTR_EL2().MPIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() then
        X{64}(t) = VMPIDR_EL2();
    else
        X{64}(t) = MPIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = MPIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.138 MVFR0_EL1, AArch32 Media and VFP Feature Register 0

The MVFR0_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR1_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

AArch64 System register MVFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register MVFR0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MVFR0_EL1 are UNDEFINED.

Attributes

MVFR0_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

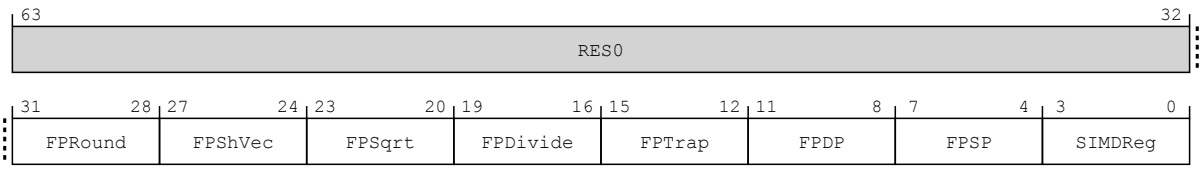

Bits [63:32]

Reserved, RES0.

FPRound, bits [31:28]

Floating-Point Rounding modes. Indicates whether the floating-point implementation provides support for rounding modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPRound	Meaning
0b0000	Not implemented, or only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.
0b0001	All rounding modes supported.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In Armv8-A the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

FPShVec, bits [27:24]

Short Vectors. Indicates whether the floating-point implementation provides support for the use of short vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPShVec	Meaning
`0b0000`	Short vectors not supported.
`0b0001`	Short vector operation supported.

All other values are reserved.

In Armv8-A the only permitted value is `0b0000`.

Access to this field is RO.

FPSqrt, bits [23:20]

Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPSqrt	Meaning
`0b0000`	Not supported in hardware.
`0b0001`	Supported.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

Access to this field is RO.

FPDivide, bits [19:16]

Indicates whether the floating-point implementation provides support for VFP divide operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDivide	Meaning
`0b0000`	Not supported in hardware.
`0b0001`	Supported.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In Armv8-A the permitted values are `0b0000` and `0b0001`.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

Access to this field is RO.

FPTrap, bits [15:12]

Floating-Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPTrap	Meaning
`0b0000`	Not supported.
`0b0001`	Supported.

All other values are reserved.

A value of `0b0001` indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

Access to this field is RO.

FPDP, bits [11:8]

Floating-Point Double-Precision. Indicates whether the floating-point implementation provides support for double-precision operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDP	Meaning
`0b0000`	Not supported in hardware.
`0b0001`	Supported, VFPv2.
`0b0010`	Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are `0b0000` and `0b0010`.

A value of `0b0001` or `0b0010` indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is `0b0001`.
- VDIV.F64 is only available if the Divide field is `0b0001`.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

Access to this field is RO.

FPSP, bits [7:4]

Floating-Point Single-Precision. Indicates whether the floating-point implementation provides support for single-precision operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPSP	Meaning
0b0000	Not supported in hardware.
0b0001	Supported, VFPv2.
0b0010	Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are 0b0000 and 0b0010.

A value of 0b0001 or 0b0010 indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is 0b0001.
- VDIV.F32 is only available if the Divide field is 0b0001.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

Access to this field is RO.

SIMDReg, bits [3:0]

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDReg	Meaning
0b0000	The implementation has no Advanced SIMD and floating-point support.
0b0001	The implementation includes floating-point support with 16 x 64-bit registers.
0b0010	The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8-A the permitted values are 0b0000 and 0b0010.

Access to this field is RO.

Otherwise:

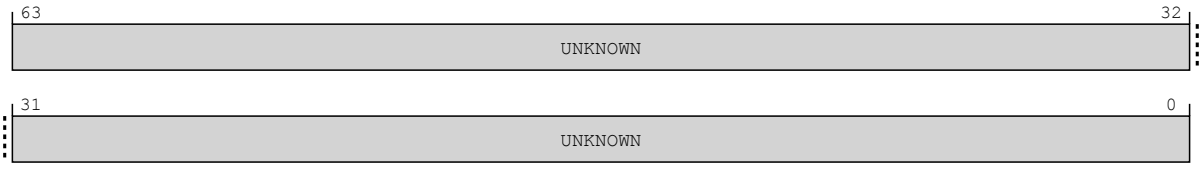

Bits [63:0]

Reserved, UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing MVFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MVFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MVFR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MVFR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MVFR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.139 MVFR1_EL1, AArch32 Media and VFP Feature Register 1

The MVFR1_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

AArch64 System register MVFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register MVFR1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MVFR1_EL1 are UNDEFINED.

Attributes

MVFR1_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

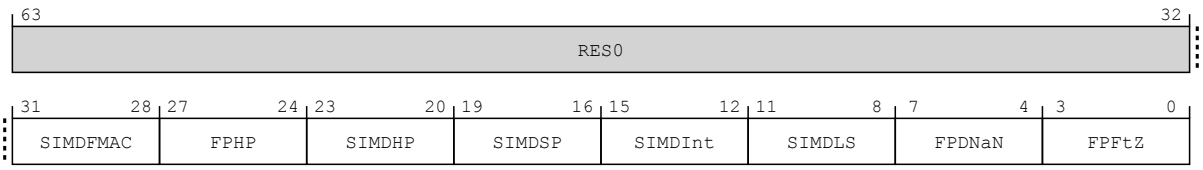

Bits [63:32]

Reserved, RES0.

SIMDFMAC, bits [31:28]

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the Advanced SIMD implementation provides fused multiply accumulate instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDFMAC	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.

Access to this field is RO.

FPHP, bits [27:24]

Floating-Point Half-Precision. Indicates the level of half-precision floating-point support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPHP	Meaning
`0b0000`	Not supported.
`0b0001`	Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.
`0b0010`	As for `0b0001`, and adds instructions for conversion between double-precision and half-precision.
`0b0011`	As for `0b0010`, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8, the permitted values are:

- `0b0000` in an implementation without floating-point support.
- `0b0010` in an implementation with floating-point support that does not include the FEAT_FP16 extension.
- `0b0011` in an implementation with floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

Half-Precision instructions supported	FPHP	SIMDHP
No support	`0b0000`	`0b0000`
Conversions only	`0b0010`	`0b0001`
Conversions and arithmetic	`0b0011`	`0b0010`

Access to this field is RO.

SIMDHP, bits [23:20]

Advanced SIMD Half-Precision. Indicates the level of half-precision floating-point support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDHP	Meaning
`0b0000`	Not supported.
`0b0001`	SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0010	As for 0b0001, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8, the permitted values are:

- 0b0000 in an implementation without SIMD floating-point support.
- 0b0001 in an implementation with SIMD floating-point support that does not include the FEAT_FP16 extension.
- 0b0010 in an implementation with SIMD floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

Half-Precision instructions supported	FPHP	SIMDHP
No support	0b0000	0b0000
Conversions only	0b0010	0b0001
Conversions and arithmetic	0b0011	0b0010

Access to this field is RO.

SIMDSP, bits [19:16]

Advanced SIMD Single-Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDSP	Meaning
0b0000	Not implemented.
0b0001	Implemented. This value is permitted only if the SIMDInt field is 0b0001.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SIMDInt, bits [15:12]

Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDInt	Meaning
0b0000	Not implemented.
0b0001	Implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIMDInt	Meaning

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

SIMDLS, bits [11:8]

Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDLS	Meaning
`0b0000`	Not implemented.
`0b0001`	Implemented.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

FPDNaN, bits [7:4]

Default NaN mode. Indicates whether the floating-point implementation provides support only for the Default NaN mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDNaN	Meaning
`0b0000`	Not implemented, or hardware supports only the Default NaN mode.
`0b0001`	Hardware supports propagation of NaN values.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0001`.

Access to this field is RO.

FPFtZ, bits [3:0]

Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPFtZ	Meaning
0b0000	Not implemented, or hardware supports only the Flush-to-Zero mode of operation.
0b0001	Hardware supports full denormalized number arithmetic.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Otherwise:

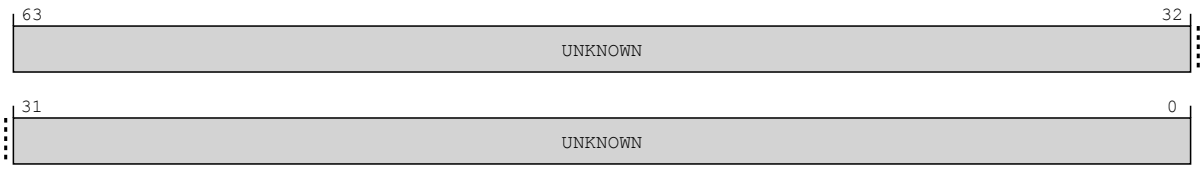

Bits [63:0]

Reserved, UNKNOWN.

Accessing MVFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MVFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MVFR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MVFR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MVFR1_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.140 MVFR2_EL1, AArch32 Media and VFP Feature Register 2

The MVFR2_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR1_EL1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

AArch64 System register MVFR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register MVFR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MVFR2_EL1 are UNDEFINED.

Attributes

MVFR2_EL1 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented:

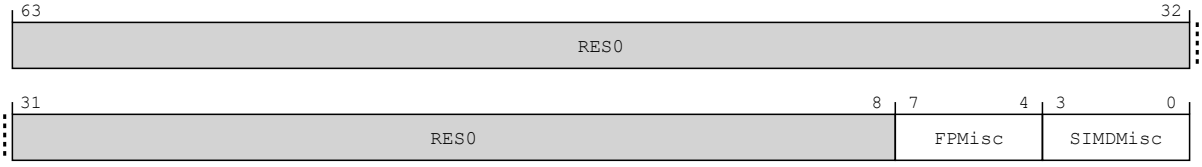

Bits [63:8]

Reserved, RES0.

FPMisc, bits [7:4]

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPMisc	Meaning
0b0000	Not implemented, or no support for miscellaneous features.
0b0001	Support for Floating-point selection.
0b0010	As 0b0001, and Floating-point Conversion to Integer with Directed Rounding modes.
0b0011	As 0b0010, and Floating-point Round to Integer Floating-point.
0b0100	As 0b0011, and Floating-point MaxNum and MinNum.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In Armv8-A, the permitted values are `0b0000` and `0b0100`.

Access to this field is RO.

SIMDMisc, bits [3:0]

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDMisc	Meaning
`0b0000`	Not implemented, or no support for miscellaneous features.
`0b0001`	Floating-point Conversion to Integer with Directed Rounding modes.
`0b0010`	As `0b0001`, and Floating-point Round to Integer Floating-point.
`0b0011`	As `0b0010`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8-A, the permitted values are `0b0000` and `0b0011`.

Access to this field is RO.

Otherwise:

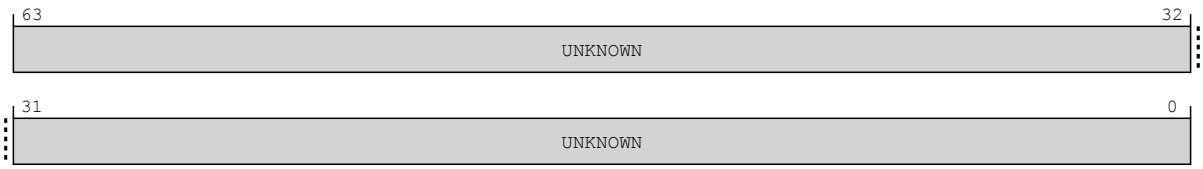

Bits [63:0]

Reserved, UNKNOWN.

Accessing MVFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MVFR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TID3 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MVFR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_IDTE3) && SCR_EL3().TID3 == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MVFR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MVFR2_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.141 PAR_EL1, Physical Address Register

The PAR_EL1 characteristics are:

Purpose

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Configuration

AArch64 System register PAR_EL1 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

Single stage AT Instructions (ATS1*) report their result using the 128-bit format of PAR_EL1 if the translation system that they target uses VMSAv9-128.

ATS12* Instructions report their result using the 128-bit format PAR_EL1 if either of the following is true:

- if stage 2 translations are enabled and the stage 2 translation system uses VMSAv9-128.
- if stage 2 translations are disabled and the stage 1 translation system uses VMSAv9-128.

Otherwise, 64-bit format of PAR_EL1 is used.

AArch64 System register PAR_EL1 bits [63:0] are architecturally mapped to AArch32 System register PAR[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to PAR_EL1 are UNDEFINED.

Attributes

PAR_EL1 is a:

- 128-bit register when FEAT_D128 is implemented, GetPAR_EL1_D128() == ‘1’, and GetPAR_EL1_F() == ‘0’.
- 128-bit register when FEAT_D128 is implemented, GetPAR_EL1_D128() == ‘1’, and GetPAR_EL1_F() == ‘1’.
- 128-bit register when FEAT_D128 is implemented, GetPAR_EL1_D128() == ‘0’, and GetPAR_EL1_F() == ‘0’.
- 128-bit register when FEAT_D128 is implemented, GetPAR_EL1_D128() == ‘0’, and GetPAR_EL1_F() == ‘1’.
- 64-bit register when FEAT_D128 is not implemented and GetPAR_EL1_F() == ‘0’.
- 64-bit register when FEAT_D128 is not implemented and GetPAR_EL1_F() == ‘1’.

Field descriptions

When FEAT_D128 is implemented, GetPAR_EL1_D128() == '1', and GetPAR_EL1_F() == '0':

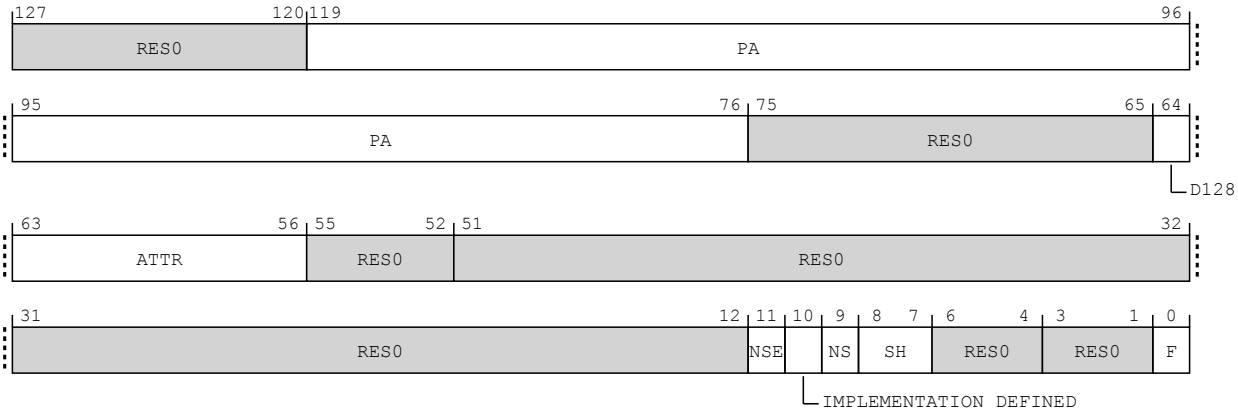

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- The PAR_EL1.{ATTR, SH} fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors.
- See the PAR_EL1.NS bit description for constraints on the value it returns.

Bits [127:120]

Reserved, RES0.

PA, bits [119:76]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[55:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [75:65]

Reserved, RES0.

D128, bit [64]

Indicates if the PAR_EL1 uses the 128-bit format.

D128	Meaning
`0b1`	PAR_EL1 uses the 128-bit format. PAR_EL1[127:0] holds valid data.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR_EL1, MAIR_EL2, and MAIR_EL3.

If FEAT_MTE_PERM is implemented and the instruction performed a stage 2 translation, the following additional encoding is defined:

ATTR	Meaning
`0b11100000`	Tagged NoTagAccess Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory.

Note

This encoding in MAIR_ELx is Reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value returned in this field can be the resulting attribute that is actually implemented by the implementation, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

Note

The attributes presented are consistent with the stages of translation applied in the address translation instruction. If the instruction performed a stage 1 translation only, the attributes are from the stage 1 translation. If the instruction performed a stage 1 and stage 2 translation, the attributes are from the combined stage 1 and stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [55:52, 6:4]

Reserved, RES0.

Bits [51:12]

Reserved, RES0.

NSE, bit [11]

When FEAT_RME is implemented:

Reports the NSE attribute for a translation table descriptor from the EL3 translation regime.

For a description of the values derived by evaluating NS and NSE together, see PAR_EL1.NS.

For a result from a Secure, Non-secure, or Realm translation regime, this bit is unknown.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

When FEAT_RME is implemented:

Non-secure. The NS attribute for a translation table entry from a Secure translation regime, a Realm translation regime, and the EL3 translation regime.

For a result from an EL3 translation regime, NS and NSE are evaluated together to report the physical address space:

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b1	0b1	Realm.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

For a result from an S1E1 or S1E0 operation on the Realm EL1&0 translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address.

SH	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

The value 0b01 is reserved.

Note

This field returns the value 0b10 for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:1]

Reserved, RES0.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b0	Address translation completed successfully.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is implemented, GetPAR_EL1_D128() == '1', and GetPAR_EL1_F() == '1':

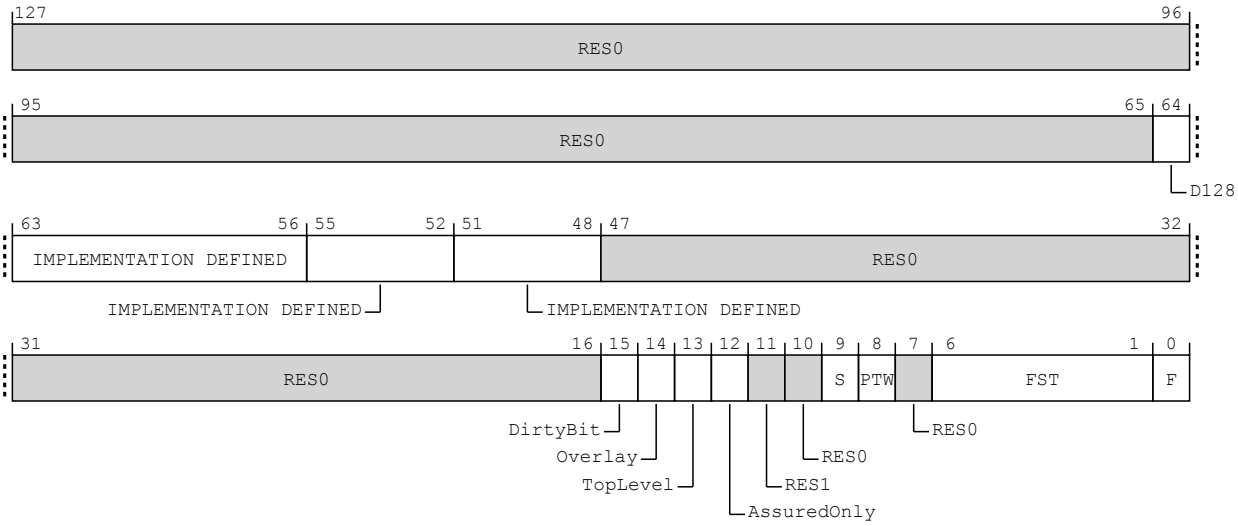

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

Bits [127:65]

Reserved, RES0.

D128, bit [64]

Indicates if the PAR_EL1 uses the 128-bit format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D128	Meaning
0b1	PAR_EL1 uses the 128-bit format. PAR_EL1[127:0] holds valid data.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:16]

Reserved, RES0.

DirtyBit, bit [15]

When FEAT_S1PIE is implemented or FEAT_S2PIE is implemented:

DirtyBit flag.

If PAR_EL1.FST indicates a Permission fault for a stage of translation that is using Indirect Permissions, and dirty state is managed by software, then this field holds information about the fault.

DirtyBit	Meaning
0b0	The Permission Fault is not due to dirty state.
0b1	The Permission Fault is due to dirty state.

For any other fault or Access, this field is RES0.

Note

At stage 1, dirty state is indicated by the nDirty bit in Block and Page descriptors. At stage 2, dirty state is indicated by the Dirty bit in Block and Page descriptors.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Overlay, bit [14]

When FEAT_S1POE is implemented or FEAT_S2POE is implemented:

Overlay flag.

If PAR_EL1.FST indicates a Permission fault for a stage of translation, then this field holds information about the fault.

Overlay	Meaning
`0b0`	The Data Abort is not due to Overlay Permissions.
`0b1`	The Data Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TopLevel, bit [13]

When FEAT_THE is implemented:

Fault due to TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
`0b0`	Fault is not due to TopLevel.
`0b1`	Fault is due to TopLevel.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [12]

When FEAT_THE is implemented:

AssuredOnly flag.

If PAR_EL1.S indicates a stage 2 fault, then this field holds information about the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AssuredOnly	Meaning
0b0	The Data Abort is not due to AssuredOnly.
0b1	The Data Abort is due to AssuredOnly.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [11]

Reserved, RES1.

Bit [10]

Reserved, RES0.

S, bit [9]

Indicates the translation stage at which the translation aborted:

S	Meaning
0b0	Translation aborted because of a fault in the stage 1 translation.
0b1	Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PTW, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status code, as shown in the Data Abort ESR encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented
0b111101	Section Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.	FEAT_AA32EL1 is implemented
0b111110	Page Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.	FEAT_AA32EL1 is implemented

Note

The encodings for FST do not include Synchronous External abort on translation table walk or hardware update of translation table, because these MMU faults are reported as a Data Abort exception instead of being recorded in PAR_EL1. See Exceptions to reporting the fault in PAR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is implemented, GetPAR_EL1_D128() == '0', and GetPAR_EL1_F() == '0':

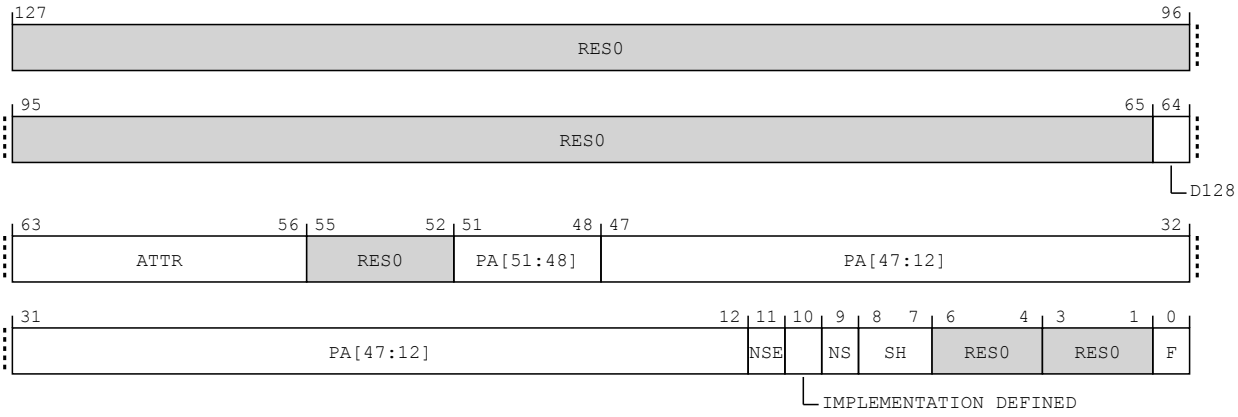

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- The PAR_EL1.{ATTR, SH} fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors.
- See the PAR_EL1.NS bit description for constraints on the value it returns.

Bits [127:65]

Reserved, RES0.

D128, bit [64]

Indicates if the PAR_EL1 uses the 128-bit format.

D128	Meaning
0b0	PAR_EL1 uses the 64-bit format. PAR_EL1[63:0] holds valid data.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR_EL1, MAIR_EL2, and MAIR_EL3.

If FEAT_MTE_PERM is implemented and the instruction performed a stage 2 translation, the following additional encoding is defined:

ATTR	Meaning
0b11100000	Tagged NoTagAccess Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory.

Note

This encoding in MAIR_ELx is Reserved.

The value returned in this field can be the resulting attribute that is actually implemented by the implementation, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

Note

The attributes presented are consistent with the stages of translation applied in the address translation instruction. If the instruction performed a stage 1 translation only, the attributes are from the stage 1 translation. If the instruction performed a stage 1 and stage 2 translation, the attributes are from the combined stage 1 and stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [55:52, 6:4]

Reserved, RES0.

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to PA[47:12]. For more information, see PA[47:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA[47:12], bits [47:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[47:12].

When FEAT_LPA is implemented and 52-bit addresses are in use, PA[51:48] forms the upper part of the address value. Otherwise, when 52-bit addresses are not in use, PA[51:48] is RES0.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSE, bit [11]

When FEAT_RME is implemented:

Reports the NSE attribute for a translation table entry from the EL3 translation regime.

For a description of the values derived by evaluating NS and NSE together, see PAR_EL1.NS.

For a result from a Secure, Non-secure, or Realm translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

When FEAT_RME is implemented:

Non-secure. The NS attribute for a translation table entry from a Secure translation regime, a Realm translation regime, and the EL3 translation regime.

For a result from an EL3 translation regime, NS and NSE are evaluated together to report the physical address space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

For a result from an S1E1 or S1E0 operation on the Realm EL1&0 translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address.

SH	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

The value 0b01 is reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This field returns the value `0b10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:1]

Reserved, RES0.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
`0b0`	Address translation completed successfully.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is implemented, GetPAR_EL1_D128() == '0', and GetPAR_EL1_F() == '1':

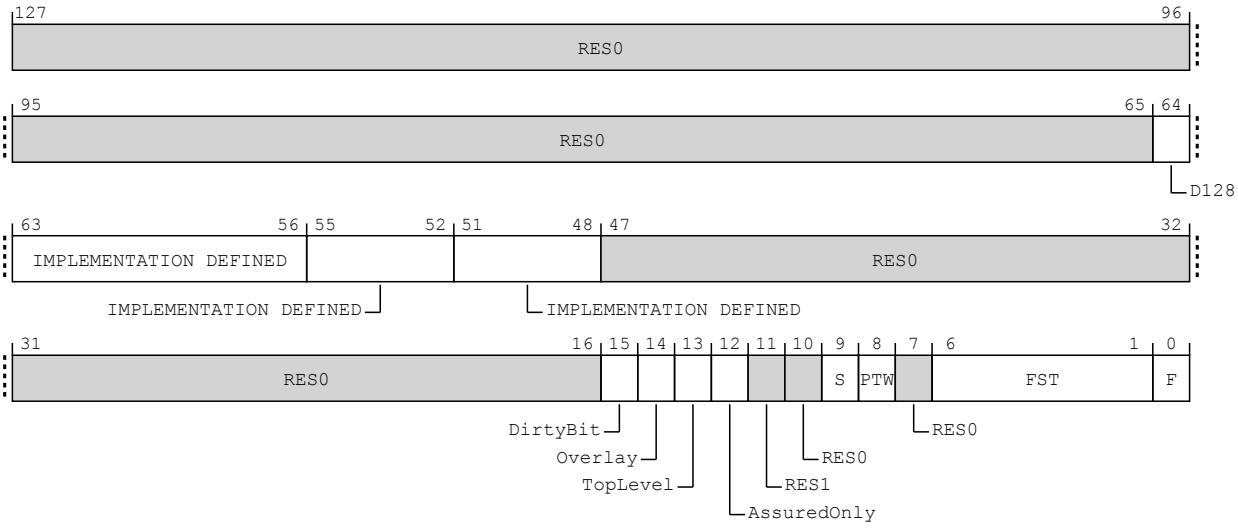

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

Bits [127:65]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D128, bit [64]

Indicates if the PAR_EL1 uses the 128-bit format.

D128	Meaning
0b0	PAR_EL1 uses the 64-bit format. PAR_EL1[63:0] holds valid data.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:16]

Reserved, RES0.

DirtyBit, bit [15]

When FEAT_S1PIE is implemented or FEAT_S2PIE is implemented:

DirtyBit flag.

If PAR_EL1.FST indicates a Permission fault for a stage of translation that is using Indirect Permissions, and dirty state is managed by software, then this field holds information about the fault.

DirtyBit	Meaning
0b0	The Permission Fault is not due to nDirty State or Dirty State.
0b1	The Permission Fault is due to nDirty State or Dirty State.

For any other fault or Access, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Overlay, bit [14]

When FEAT_S1POE is implemented or FEAT_S2POE is implemented:

Overlay flag.

If PAR_EL1.FST indicates a Permission fault for a stage of translation, then this field holds information about the fault.

Overlay	Meaning
`0b0`	The Data Abort is not due to Overlay Permissions.
`0b1`	The Data Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TopLevel, bit [13]

When FEAT_THE is implemented:

Fault due to TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
`0b0`	Fault is not due to TopLevel.
`0b1`	Fault is due to TopLevel.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [12]

When FEAT_THE is implemented:

AssuredOnly flag.

If PAR_EL1.S indicates a stage 2 fault, then this field holds information about the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AssuredOnly	Meaning
0b0	The Data Abort is not due to AssuredOnly.
0b1	The Data Abort is due to AssuredOnly.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [11]

Reserved, RES1.

Bit [10]

Reserved, RES0.

S, bit [9]

Indicates the translation stage at which the translation aborted:

S	Meaning
0b0	Translation aborted because of a fault in the stage 1 translation.
0b1	Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PTW, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status code, as shown in the Data Abort exception ESR encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented
0b111101	Section Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.	FEAT_AA32EL1 is implemented
0b111110	Page Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.	FEAT_AA32EL1 is implemented

Note

The encodings for FST do not include Synchronous External abort on translation table walk or hardware update of translation table, because these MMU faults are reported as a Data Abort exception instead of being recorded in PAR_EL1. See Exceptions to reporting the fault in PAR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is not implemented and GetPAR_EL1_F() == '0':

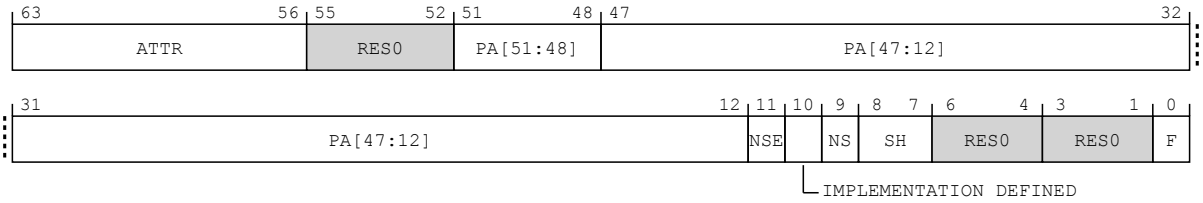

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PAR_EL1.{ATTR, SH} fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors.
- See the PAR_EL1.NS bit description for constraints on the value it returns.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR_EL1, MAIR_EL2, and MAIR_EL3.

If FEAT_MTE_PERM is implemented and the instruction performed a stage 2 translation, the following additional encoding is defined:

ATTR	Meaning
`0b11100000`	Tagged NoTagAccess Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory.

Note

This encoding in MAIR_ELx is Reserved.

The value returned in this field can be the resulting attribute that is actually implemented by the implementation, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

Note

The attributes presented are consistent with the stages of translation applied in the address translation instruction. If the instruction performed a stage 1 translation only, the attributes are from the stage 1 translation. If the instruction performed a stage 1 and stage 2 translation, the attributes are from the combined stage 1 and stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [55:52, 6:4]

Reserved, RES0.

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to PA[47:12]. For more information, see PA[47:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA[47:12], bits [47:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[47:12].

When FEAT_LPA is implemented and 52-bit addresses are in use, PA[51:48] forms the upper part of the address value. Otherwise, when 52-bit addresses are not in use, PA[51:48] is RES0.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSE, bit [11]

When FEAT_RME is implemented:

Reports the NSE attribute for a translation table entry from the EL3 translation regime.

For a description of the values derived by evaluating NS and NSE together, see PAR_EL1.NS.

For a result from a Secure, Non-secure, or Realm translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

When FEAT_RME is implemented:

Non-secure. The NS attribute for a translation table entry from a Secure translation regime, a Realm translation regime, and the EL3 translation regime.

For a result from an EL3 translation regime, NS and NSE are evaluated together to report the physical address space:

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

For a result from an S1E1 or S1E0 operation on the Realm EL1&0 translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address.

SH	Meaning
`0b00`	Non-shareable.
`0b10`	Outer Shareable.
`0b11`	Inner Shareable.

The value `0b01` is reserved.

Note

This field returns the value `0b10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:1]

Reserved, RES0.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
`0b0`	Address translation completed successfully.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is not implemented and GetPAR_EL1_F() == '1':

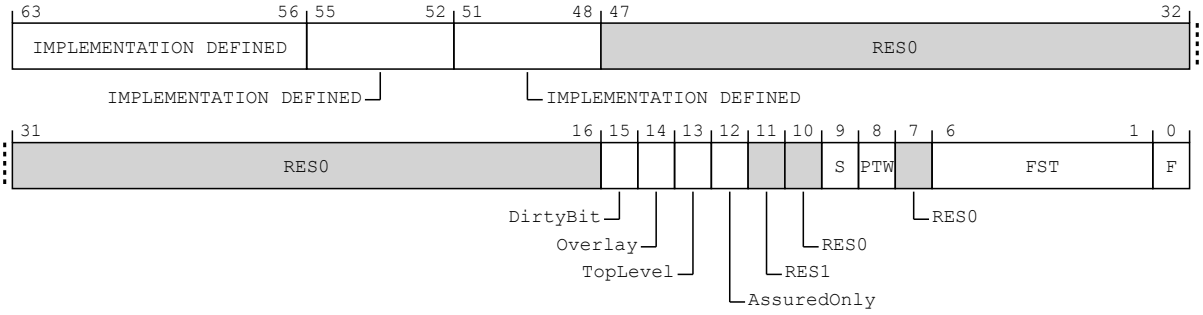

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:16]

Reserved, RES0.

DirtyBit, bit [15]

When FEAT_S1PIE is implemented or FEAT_S2PIE is implemented:

DirtyBit flag.

If PAR_EL1.FST indicates a Permission fault for a stage of translation that is using Indirect Permissions, and dirty state is managed by software, then this field holds information about the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DirtyBit	Meaning
0b0	The Permission Fault is not due to nDirty State or Dirty State.
0b1	The Permission Fault is due to nDirty State or Dirty State.

For any other fault or Access, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Overlay, bit [14]

When FEAT_S1POE is implemented or FEAT_S2POE is implemented:

Overlay flag.

If PAR_EL1.FST indicates a Permission fault for a stage of translation, then this field holds information about the fault.

Overlay	Meaning
0b0	The Data Abort is not due to Overlay Permissions.
0b1	The Data Abort is due to Overlay Permissions.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TopLevel, bit [13]

When FEAT_THE is implemented:

Fault due to TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

AssuredOnly, bit [12]

When FEAT_THE is implemented:

AssuredOnly flag.

If PAR_EL1.S indicates a stage 2 fault, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	The Data Abort is not due to AssuredOnly.
0b1	The Data Abort is due to AssuredOnly.

For any other fault, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [11]

Reserved, RES1.

Bit [10]

Reserved, RES0.

S, bit [9]

Indicates the translation stage at which the translation aborted:

S	Meaning
0b0	Translation aborted because of a fault in the stage 1 translation.
0b1	Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PTW, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST, bits [6:1]

Fault status code, as shown in the Data Abort exception ESR encoding.

FST	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented
0b111101	Section Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.	FEAT_AA32EL1 is implemented
0b111110	Page Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.	FEAT_AA32EL1 is implemented

Note

The encodings for FST do not include Synchronous External abort on translation table walk or hardware update of translation table, because these MMU faults are reported as a Data Abort exception instead of being recorded in PAR_EL1. See Exceptions to reporting the fault in PAR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PAR_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().PAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = PAR_EL1()[63:0];
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = PAR_EL1()[63:0];
elsif PSTATE.EL == EL3 then
    X{64}(t) = PAR_EL1()[63:0];
end;
```

MSR PAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().PAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        PAR_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    PAR_EL1()[63:0] = X{64}(t);
elsif PSTATE.EL == EL3 then
    PAR_EL1()[63:0] = X{64}(t);
end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, PAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGRTR_EL2().PAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = PAR_EL1()[127:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = PAR_EL1()[127:0];
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = PAR_EL1()[127:0];
end;
```

When FEAT_D128 is implemented MSRR PAR_EL1, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGWTR_EL2().PAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        PAR_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        PAR_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    PAR_EL1()[127:0] = X{128}(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.142 PFAR_EL1, Physical Fault Address Register (EL1)

The PFAR_EL1 characteristics are:

Purpose

Records the faulting physical address for a synchronous External abort, or SError exception taken to EL1.

Configuration

This register is present only when FEAT_PFAR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PFAR_EL1 are UNDEFINED.

Attributes

PFAR_EL1 is a 64-bit register.

Field descriptions

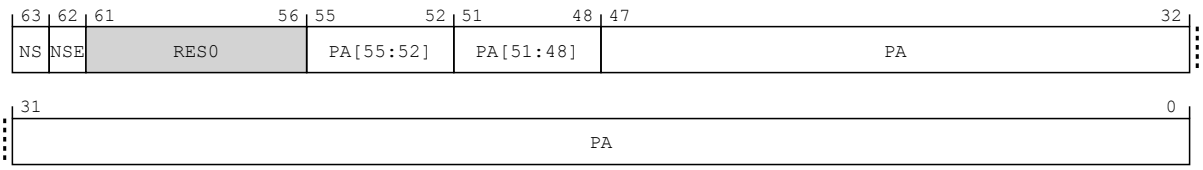

NS, bit [63]

When FEAT_RME is implemented:

Together with PFAR_EL1.NSE, reports the physical address space of the access that triggered the exception.

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Reserved.
0b1	0b1	Realm.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When EL3 is implemented or FEAT_Secure is implemented:

Non-secure. Reports the physical address space of the access that triggered the exception.

NS	Meaning
0b0	Secure physical address space.
0b1	Non-secure physical address space.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSE, bit [62]

When FEAT_RME is implemented:

Together with PFAR_EL1.NS, reports the physical address space of the access that triggered the exception.

For a description of the values derived by evaluating NS and NSE together, see MFAR_EL3.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [61:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

When FEAT_D128 is implemented, extension to PFAR_EL1.PA[47:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:

When FEAT_LPA is implemented, extension to PFAR_EL1.PA[47:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA, bits [47:0]

Physical Address. Bits [47:0] of the aborting physical address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The recorded address can be any address within the same naturally-aligned fault granule as the faulting physical address, where the size of the fault granule is IMPLEMENTATION DEFINED and no larger than the larger than:

- The size of the range of values permitted to be recorded in FAR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PFAR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `PFAR_EL1` or `PFAR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

PFAR_EL1 is not valid and reads UNKNOWN if ESR_EL1.PFV is recorded as 0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PFAR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_PFAR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nPFAR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x2D0);
    else
        X{64}(t) = PFAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PFAR_EL2();
    else
        X{64}(t) = PFAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PFAR_EL1();
end;
```

```
MSR PFAR_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_PFAR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nPFAR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x2D0) = X{64}(t);
    else
        PFAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PFAR_EL2() = X{64}(t);
    else
        PFAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PFAR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, PFAR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_PFAR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x2D0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
                Undefined();
            elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                X{64}(t) = PFAR_EL1();
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = PFAR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR PFAR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_PFAR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x2D0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            PFAR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PFAR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.143 PFAR_EL2, Physical Fault Address Register (EL2)

The PFAR_EL2 characteristics are:

Purpose

Records the faulting physical address for a synchronous External abort, or SError exception taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_PFAR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PFAR_EL2 are UNDEFINED.

Attributes

PFAR_EL2 is a 64-bit register.

Field descriptions

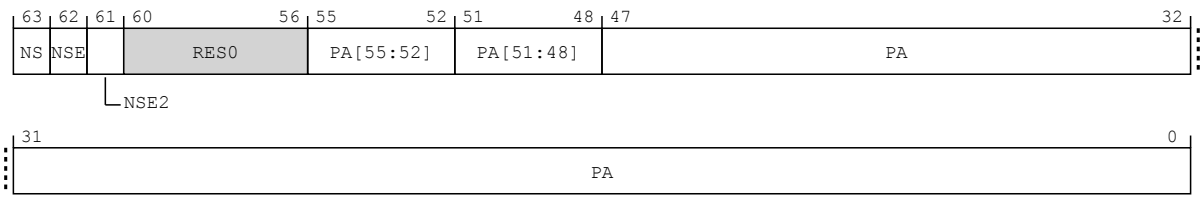

NS, bit [63]

When FEAT_RME_GDI is implemented:

Together with PFAR_EL2.NSE and PFAR_EL2.NSE2, reports the physical address space of the access that triggered the exception.

NSE2	NSE	NS	Meaning
0b0	0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b0	0b1	Non-secure.
0b0	0b1	0b0	Reserved.
0b0	0b1	0b1	Realm.
0b1	0b0	0b0	System Agent.
0b1	0b0	0b1	NS Protected.
0b1	0b1	0b0	Reserved.
0b1	0b1	0b1	Reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RME is implemented:

Together with PFAR_EL2.NSE, reports the physical address space of the access that triggered the exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Reserved.
0b1	0b1	Realm.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When EL3 is implemented or FEAT_Secure is implemented:

Non-secure. Reports the physical address space of the access that triggered the exception.

NS	Meaning
0b0	Secure physical address space.
0b1	Non-secure physical address space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSE, bit [62]

When FEAT_RME is implemented:

Together with PFAR_EL2.NS, reports the physical address space of the access that triggered the exception.

For a description of the values derived by evaluating NS and NSE together, see MFAR_EL3.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSE2, bit [61]

When FEAT_RME_GDI is implemented and FEAT_MEC is implemented:

Together with PFAR_EL2.NS and PFAR_EL2.NSE, reports the physical address space of the access that triggered the exception.

For a description of the values derived by evaluating NS, NSE, and NSE2 together, see PFAR_EL2.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [60:56]

Reserved, RES0.

PA[55:52], bits [55:52]

When FEAT_D128 is implemented:

When FEAT_D128 is implemented, extension to PFAR_EL2.PA[47:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:

When FEAT_LPA is implemented, extension to PFAR_EL2.PA[47:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA, bits [47:0]

Physical Address. Bits [47:0] of the aborting physical address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The recorded address can be any address within the same naturally-aligned fault granule as the faulting physical address, where the size of the fault granule is IMPLEMENTATION DEFINED and no larger than the larger than:

- The size of the range of values permitted to be recorded in FAR_EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PFAR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `PFAR_EL2` or `PFAR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

PFAR_EL2 is not valid and reads UNKNOWN if ESR_EL2.PFV is recorded as 0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PFAR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_PFAR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PFAR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PFAR_EL2();
end;
```

```
MSR PFAR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_PFAR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PFAREn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PFAREn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PFAR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PFAR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.144 PIR_EL1, Permission Indirection Register 1 (EL1)

The PIR_EL1 characteristics are:

Purpose

Stage 1 Permission Indirection Register for privileged access of the EL1&0 translation regime.

Configuration

This register is present only when FEAT_S1PIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PIR_EL1 are UNDEFINED.

Attributes

PIR_EL1 is a 64-bit register.

Field descriptions

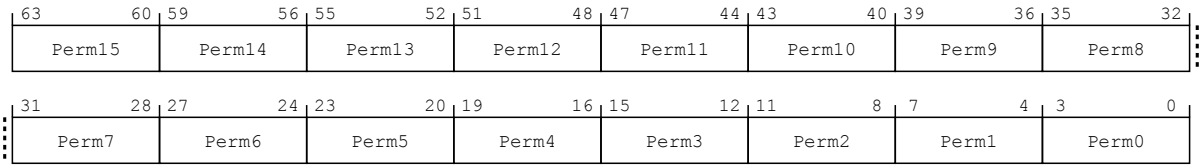

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Represents stage 1 Base Permissions.

Perm<m>	Meaning
0b0000	No access. Overlay applied.
0b0001	Read. Overlay applied.
0b0010	Execute. Overlay applied.
0b0011	Read and Execute. Overlay applied.
0b0100	Reserved - treated as No access. Overlay applied.
0b0101	Read and Write. Overlay applied.
0b0110	Read, Write, and Execute. Overlay applied. WXN control applied.
0b0111	Read, Write, and Execute. Overlay applied.
0b1000	Read. Overlay not applied.
0b1001	Read, GCS Read, and GCS Write. Overlay not applied.
0b1010	Read and Execute. Overlay not applied.
0b1011	Reserved - treated as No access. Overlay not applied.
0b1100	Read and Write. Overlay not applied.
0b1101	Reserved - treated as No access. Overlay not applied.
0b1110	Read, Write, and Execute. Overlay not applied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1111	Reserved - treated as No access. Overlay not applied.

This field is permitted to be cached in a TLB.

When stage 1 Indirect Permission mechanism is disabled, this register is ignored.

Unless otherwise specified, the WXN control is not applied.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PIR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name PIR_EL1 or PIR_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPIR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x2A0);
    else
        X{64}(t) = PIR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PIR_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = PIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIR_EL1();
end;
```

MSR PIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGWTR_EL2().nPIR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x2A0) = X{64}(t);
    else
        PIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PIR_EL2() = X{64}(t);
    else
        PIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PIR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, PIR_EL12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x2A0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PIR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = PIR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR PIR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x2A0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            PIR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        PIR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.145 PIR_EL2, Permission Indirection Register 2 (EL2)

The PIR_EL2 characteristics are:

Purpose

Stage 1 Permission Indirection Register for privileged access of the EL2 or EL2&0 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_S1PIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PIR_EL2 are UNDEFINED.

Attributes

PIR_EL2 is a 64-bit register.

Field descriptions

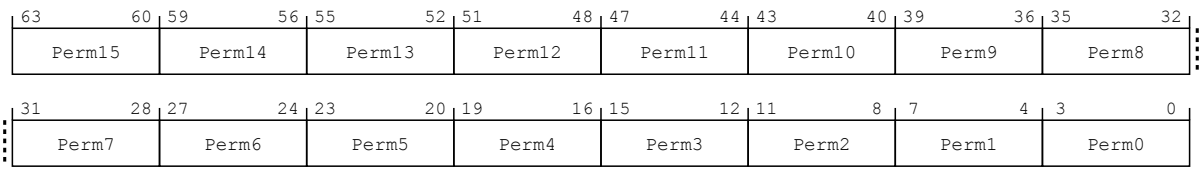

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Represents stage 1 Base Permissions.

Perm<m>	Meaning
0b0000	No access. Overlay applied.
0b0001	Read. Overlay applied.
0b0010	Execute. Overlay applied.
0b0011	Read and Execute. Overlay applied.
0b0100	Reserved - treated as No access. Overlay applied.
0b0101	Read and Write. Overlay applied.
0b0110	Read, Write, and Execute. Overlay applied. WXN control applied.
0b0111	Read, Write, and Execute. Overlay applied.
0b1000	Read. Overlay not applied.
0b1001	Read, GCS Read, and GCS Write. Overlay not applied.
0b1010	Read and Execute. Overlay not applied.
0b1011	Reserved - treated as No access. Overlay not applied.
0b1100	Read and Write. Overlay not applied.
0b1101	Reserved - treated as No access. Overlay not applied.
0b1110	Read, Write, and Execute. Overlay not applied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Perm<m>	Meaning
0b1111	Reserved - treated as No access. Overlay not applied.

This field is permitted to be cached in a TLB.

When stage 1 Indirect Permission mechanism is disabled, this register is ignored.

Unless otherwise specified, the WXN control is not applied.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PIR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name PIR_EL2 or PIR_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PIR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PIR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIR_EL2();
end;
```

```
MSR PIR_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PIR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PIR_EL2() = X{64}(t);
end;
```

MRS <Xt>, PIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPIR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x2A0);
    else
        X{64}(t) = PIR_EL1();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PIR_EL2();
    else
        X{64}(t) = PIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIR_EL1();
end;
```

```
MSR PIR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nPIR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x2A0) = X{64}(t);
    else
        PIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PIR_EL2() = X{64}(t);
    else
        PIR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PIR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.146 PIR_EL3, Permission Indirection Register 3 (EL3)

The PIR_EL3 characteristics are:

Purpose

Stage 1 Permission Indirection Register for privileged access of the EL3 translation regime.

Configuration

This register is present only when FEAT_S1PIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PIR_EL3 are UNDEFINED.

Attributes

PIR_EL3 is a 64-bit register.

Field descriptions

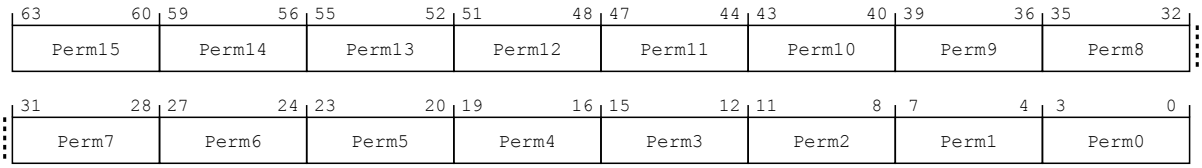

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Represents stage 1 Base Permissions.

Perm<m>	Meaning
0b0000	No access. Overlay applied.
0b0001	Read. Overlay applied.
0b0010	Execute. Overlay applied.
0b0011	Read and Execute. Overlay applied.
0b0100	Reserved - treated as No access. Overlay applied.
0b0101	Read and Write. Overlay applied.
0b0110	Read, Write, and Execute. Overlay applied. WXN control applied.
0b0111	Read, Write, and Execute. Overlay applied.
0b1000	Read. Overlay not applied.
0b1001	Read, GCS Read, and GCS Write. Overlay not applied.
0b1010	Read and Execute. Overlay not applied.
0b1011	Reserved - treated as No access. Overlay not applied.
0b1100	Read and Write. Overlay not applied.
0b1101	Reserved - treated as No access. Overlay not applied.
0b1110	Read, Write, and Execute. Overlay not applied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1111	Reserved - treated as No access. Overlay not applied.

This field is permitted to be cached in a TLB.

When stage 1 Indirect Permission mechanism is disabled, this register is ignored.

Unless otherwise specified, the WXN control is not applied.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PIR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PIR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIR_EL3();
end;
```

```
MSR PIR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().PIR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        PIR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.147 PIRE0_EL1, Permission Indirection Register 0 (EL1)

The PIRE0_EL1 characteristics are:

Purpose

Stage 1 Permission Indirection Register for unprivileged access of the EL1&0 translation regime.

Configuration

This register is present only when FEAT_S1PIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PIRE0_EL1 are UNDEFINED.

Attributes

PIRE0_EL1 is a 64-bit register.

Field descriptions

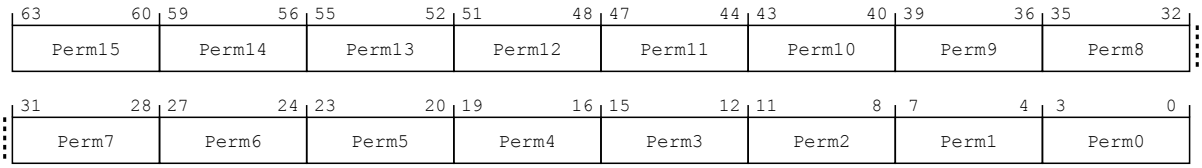

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Represents stage 1 Base Permissions.

Perm<m>	Meaning
0b0000	No access. Overlay applied.
0b0001	Read. Overlay applied.
0b0010	Execute. Overlay applied.
0b0011	Read and Execute. Overlay applied.
0b0100	Reserved - treated as No access. Overlay applied.
0b0101	Read and Write. Overlay applied.
0b0110	Read, Write, and Execute. Overlay applied. WXN control applied.
0b0111	Read, Write, and Execute. Overlay applied.
0b1000	Read. Overlay not applied.
0b1001	Read, GCS Read, and GCS Write. Overlay not applied.
0b1010	Read and Execute. Overlay not applied.
0b1011	Reserved - treated as No access. Overlay not applied.
0b1100	Read and Write. Overlay not applied.
0b1101	Reserved - treated as No access. Overlay not applied.
0b1110	Read, Write, and Execute. Overlay not applied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1111	Reserved - treated as No access. Overlay not applied.

This field is permitted to be cached in a TLB.

When stage 1 Indirect Permission mechanism is disabled, this register is ignored.

Unless otherwise specified, the WXN control is not applied.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PIRE0_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name PIRE0_EL1 or PIRE0_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PIRE0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPIRE0_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x290);
    else
        X{64}(t) = PIRE0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PIRE0_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = PIRE0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIRE0_EL1();
end;
```

MSR PIRE0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nPIRE0_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x290) = X{64}(t);
    else
        PIRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PIRE0_EL2() = X{64}(t);
    else
        PIRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PIRE0_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, PIRE0_EL12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x290);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PIRE0_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = PIRE0_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR PIRE0_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x290) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            PIRE0_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        PIRE0_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.148 PIRE0_EL2, Permission Indirection Register 0 (EL2)

The PIRE0_EL2 characteristics are:

Purpose

Stage 1 Permission Indirection Register for unprivileged access of the EL2&0 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_S1PIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PIRE0_EL2 are UNDEFINED.

Attributes

PIRE0_EL2 is a 64-bit register.

Field descriptions

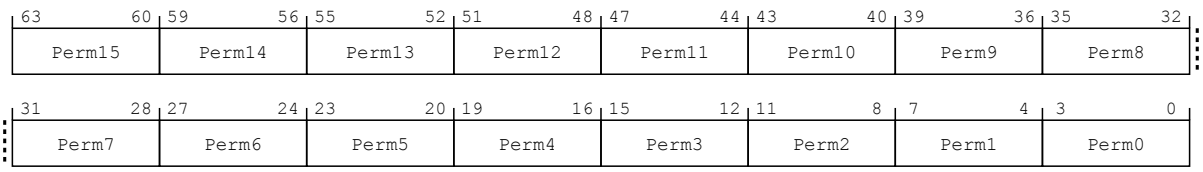

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Represents stage 1 Base Permissions.

Perm<m>	Meaning
0b0000	No access. Overlay applied.
0b0001	Read. Overlay applied.
0b0010	Execute. Overlay applied.
0b0011	Read and Execute. Overlay applied.
0b0100	Reserved - treated as No access. Overlay applied.
0b0101	Read and Write. Overlay applied.
0b0110	Read, Write, and Execute. Overlay applied. WXN control applied.
0b0111	Read, Write, and Execute. Overlay applied.
0b1000	Read. Overlay not applied.
0b1001	Read, GCS Read, and GCS Write. Overlay not applied.
0b1010	Read and Execute. Overlay not applied.
0b1011	Reserved - treated as No access. Overlay not applied.
0b1100	Read and Write. Overlay not applied.
0b1101	Reserved - treated as No access. Overlay not applied.
0b1110	Read, Write, and Execute. Overlay not applied.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Perm<m>	Meaning
0b1111	Reserved - treated as No access. Overlay not applied.

This field is permitted to be cached in a TLB.

When stage 1 Indirect Permission mechanism is disabled, this register is ignored.

Unless otherwise specified, the WXN control is not applied.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PIRE0_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name PIRE0_EL2 or PIRE0_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PIRE0_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PIRE0_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIRE0_EL2();
end;
```

```
MSR PIRE0_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PIRE0_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PIRE0_EL2() = X{64}(t);
end;
```

MRS <Xt>, PIRE0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPIRE0_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x290);
    else
        X{64}(t) = PIRE0_EL1();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PIRE0_EL2();
    else
        X{64}(t) = PIRE0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PIRE0_EL1();
end;
```

MSR PIRE0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_S1PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nPIRE0_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x290) = X{64}(t);
    else
        PIRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PIRE0_EL2() = X{64}(t);
    else
        PIRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PIRE0_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.149 POR_EL0, Permission Overlay Register 0 (EL0)

The POR_EL0 characteristics are:

Purpose

Stage 1 Permission Overlay Register for unprivileged access of EL1&0 or EL2&0 translation regime.

Configuration

This register is present only when FEAT_S1POE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to POR_EL0 are UNDEFINED.

Attributes

POR_EL0 is a 64-bit register.

Field descriptions

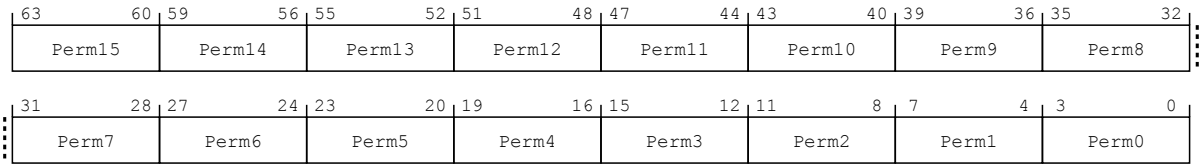

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Perm Represents stage 1 Overlay Permissions.

Perm<m>	Meaning
0b0000	No access.
0b0001	Read.
0b0010	Execute.
0b0011	Read, Execute.
0b0100	Write.
0b0101	Write, Read.
0b0110	Write, Execute.
0b0111	Read, Write, Execute.
0b1xxx	Reserved - treated as No access

If FEAT_D128 is implemented and VMSAv9-128 is in use, then fields Perm[8] to Perm[15] are used for POIndex values 8 to 15.

Otherwise, the fields Perm[8] to Perm[15] are RES0.

This field is not permitted to be cached in a TLB.

When the stage 1 Overlay mechanism is disabled, this field is IGNORED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing POR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, POR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().E0POE == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().nPOR_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CPTR_EL2().E0POE == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = POR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPOR_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = POR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = POR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = POR_EL0();
end;
```

```
MSR POR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && CPACR_EL1().E0POE == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGWTR_EL2().nPOR_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CPTR_EL2().E0POE == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        POR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nPOR_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        POR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        POR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    POR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.150 POR_EL1, Permission Overlay Register 1 (EL1)

The POR_EL1 characteristics are:

Purpose

Stage 1 Permission Overlay Register for privileged access of the EL1&0 translation regime.

Configuration

This register is present only when FEAT_S1POE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to POR_EL1 are UNDEFINED.

Attributes

POR_EL1 is a 64-bit register.

Field descriptions

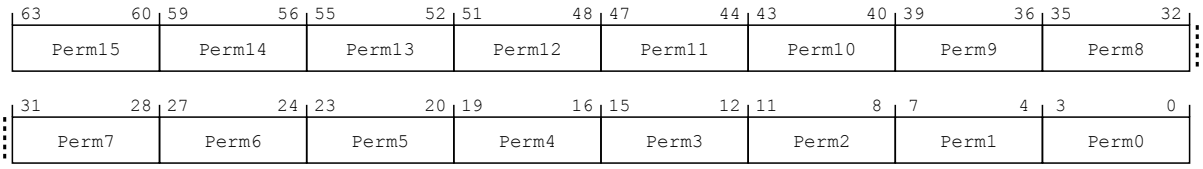

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Perm Represents stage 1 Overlay Permissions.

Perm<m>	Meaning
0b0000	No access.
0b0001	Read.
0b0010	Execute.
0b0011	Read, Execute.
0b0100	Write.
0b0101	Write, Read.
0b0110	Write, Execute.
0b0111	Read, Write, Execute.
0b1xxx	Reserved - treated as No access

If FEAT_D128 is implemented and VMSAv9-128 is in use, then fields Perm[8] to Perm[15] are used for POIndex values 8 to 15.

Otherwise, the fields Perm[8] to Perm[15] are RES0.

This field is not permitted to be cached in a TLB.

When the stage 1 Overlay mechanism is disabled, this field is IGNORED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing POR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `POR_EL1` or `POR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, POR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPOR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x2A8);
    else
        X{64}(t) = POR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = POR_EL2();
    else
        X{64}(t) = POR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = POR_EL1();
end;
```

```
MSR POR_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nPOR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x2A8) = X{64}(t);
    else
        POR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        POR_EL2() = X{64}(t);
    else
        POR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    POR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, POR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x2A8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
                Undefined();
            elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                X{64}(t) = POR_EL1();
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = POR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR POR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x2A8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            POR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        POR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.151 POR_EL2, Permission Overlay Register 2 (EL2)

The POR_EL2 characteristics are:

Purpose

Stage 1 Permission Overlay Register for privileged access of the EL2 or EL2&0 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_S1POE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to POR_EL2 are UNDEFINED.

Attributes

POR_EL2 is a 64-bit register.

Field descriptions

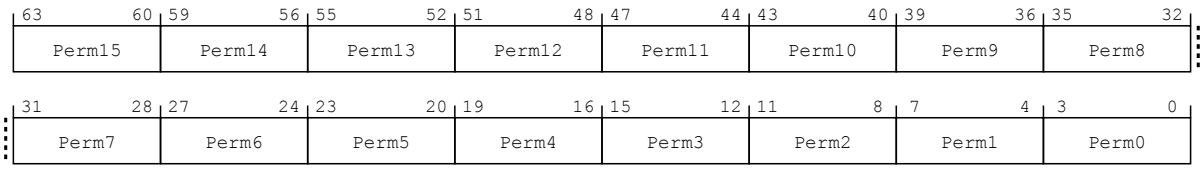

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Perm Represents stage 1 Overlay Permissions.

Perm<m>	Meaning
0b0000	No access.
0b0001	Read.
0b0010	Execute.
0b0011	Read, Execute.
0b0100	Write.
0b0101	Write, Read.
0b0110	Write, Execute.
0b0111	Read, Write, Execute.
0b1xxx	Reserved - treated as No access

If FEAT_D128 is implemented and VMSAv9-128 is in use, then fields Perm[8] to Perm[15] are used for POIndex values 8 to 15.

Otherwise, the fields Perm[8] to Perm[15] are RES0.

This field is not permitted to be cached in a TLB.

When the stage 1 Overlay mechanism is disabled, this field is IGNORED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing POR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `POR_EL2` or `POR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, POR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = POR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = POR_EL2();
end;
```

```
MSR POR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        POR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    POR_EL2() = X{64}(t);
end;
```

MRS <Xt>, POR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nPOR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x2A8);
    else
        X{64}(t) = POR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = POR_EL2();
    else
        X{64}(t) = POR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = POR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR POR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nPOR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x2A8) = X{64}(t);
    else
        POR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        POR_EL2() = X{64}(t);
    else
        POR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    POR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.152 POR_EL3, Permission Overlay Register 3 (EL3)

The POR_EL3 characteristics are:

Purpose

Stage 1 Permission Overlay Register for privileged access of the EL3 translation regime.

Configuration

This register is present only when FEAT_S1POE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to POR_EL3 are UNDEFINED.

Attributes

POR_EL3 is a 64-bit register.

Field descriptions

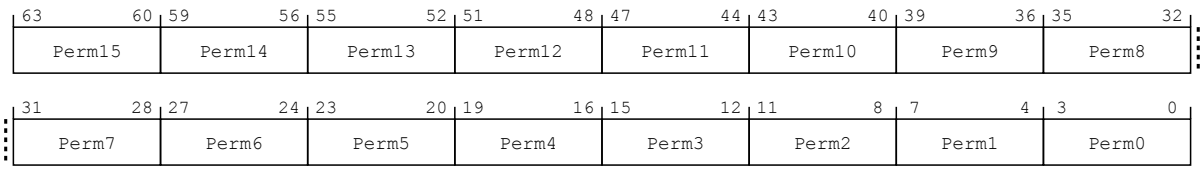

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Perm Represents stage 1 Overlay Permissions.

Perm<m>	Meaning
0b0000	No access.
0b0001	Read.
0b0010	Execute.
0b0011	Read, Execute.
0b0100	Write.
0b0101	Write, Read.
0b0110	Write, Execute.
0b0111	Read, Write, Execute.
0b1xxx	Reserved - treated as No access

If FEAT_D128 is implemented and VMSAv9-128 is in use, then fields Perm[8] to Perm[15] are used for POIndex values 8 to 15.

Otherwise, the fields Perm[8] to Perm[15] are RES0.

This field is not permitted to be cached in a TLB.

When stage 1 Overlay mechanism is disabled, this register is ignored.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing POR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, POR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = POR_EL3();
end;
```

```
MSR POR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_S1POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    POR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.153 RCWMASK_EL1, Read Check Write Instruction Mask (EL1)

The RCWMASK_EL1 characteristics are:

Purpose

Contains the mask used by RCW instructions.

Configuration

RCWMASK_EL1 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

This register is present only when FEAT_THE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to RCWMASK_EL1 are UNDEFINED.

Attributes

RCWMASK_EL1 is a:

- 128-bit register when FEAT_D128 is implemented.
- 64-bit register otherwise.

Field descriptions

When FEAT_D128 is implemented:

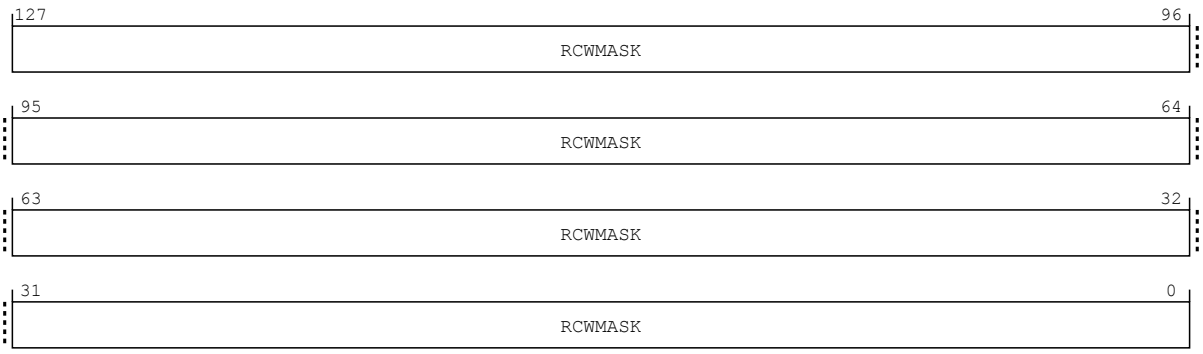

RCWMASK, bits [127:0]

Mask used to decide which bit-fields are writable to the 128-bit Descriptor by RCW or RCWS instructions.

If RCWMASK_EL1 is indirectly read by 128-bit variants of RCW or RCWS instructions:

- The Effective value of RCWMASK[n] is the same as RCWMASK_EL1[n], except as follows:
  - If n >= 17, and n <= 55, the Effective value of RCWMASK[n] is the same as RCWMASK_EL1[16].
  - If n is in {126:125, 120:119, 114, 107:101, 90:56, 1:0}, the Effective value of RCWMASK[n] is 0.
  - If n >= 121, n <= 124, and FEAT_S1POE is not implemented, the Effective value of RCWMASK[n] is 0.
  - If FEAT_MEC is not implemented, the Effective value of RCWMASK[108] is 0.

If RCWMASK_EL1 is indirectly read by 64-bit variants of RCW or RCWS instructions:

- The Effective value of RCWMASK[n] is the same as RCWMASK_EL1[n], except as follows:
  - If n >= 18, and n <= 49, the Effective value of RCWMASK[n] is the same as RCWMASK_EL1[17].

RCWMASK_EL1 register bits {126:125, 120:119, 114, 107:101, 90:64, 52, 49:18, 0} are RES0.

If FEAT_S1POE is not implemented, RCWMASK_EL1 register bits {124:121} are RES0.

If FEAT_MEC is not implemented, RCWMASK_EL1[108] is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

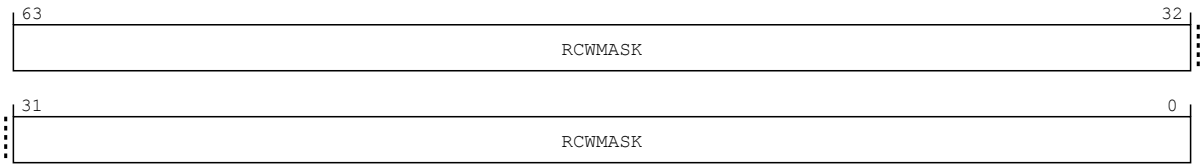

RCWMASK, bits [63:0]

Mask used to decide which bit-fields are writable to the 64-bit Descriptor by RCW or RCWS Instructions.

The Effective value of RCWMASK[n] is the same as RCWMASK_EL1[n], except as follows:

- If n >= 18, and n <= 49, the Effective value of RCWMASK[n] is the same as RCWMASK_EL1[17].

RCWMASK_EL1 register bits {52, 49:18, 0} are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing RCWMASK_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RCWMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nRCWMASK_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = RCWMASK_EL1()[63:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = RCWMASK_EL1()[63:0];
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = RCWMASK_EL1()[63:0];
end;
```

MSR RCWMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nRCWMASK_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        RCWMASK_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        RCWMASK_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    RCWMASK_EL1()[63:0] = X{64}(t);
end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, RCWMASK_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGRTR_EL2().nRCWMASK_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = RCWMASK_EL1()[127:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = RCWMASK_EL1()[127:0];
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = RCWMASK_EL1()[127:0];
end;
```

When FEAT_D128 is implemented MSRR RCWMASK_EL1, <Xt>, <Xt+1>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().nRCWMASK_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        RCWMASK_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        RCWMASK_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    RCWMASK_EL1()[127:0] = X{128}(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.154 RCWSMASK_EL1, Software Read Check Write Instruction Mask (EL1)

The RCWSMASK_EL1 characteristics are:

Purpose

Contains the software mask used by RCWS instructions.

Configuration

RCWSMASK_EL1 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

This register is present only when FEAT_THE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to RCWSMASK_EL1 are UNDEFINED.

Attributes

RCWSMASK_EL1 is a:

- 128-bit register when FEAT_D128 is implemented.
- 64-bit register otherwise.

Field descriptions

When FEAT_D128 is implemented:

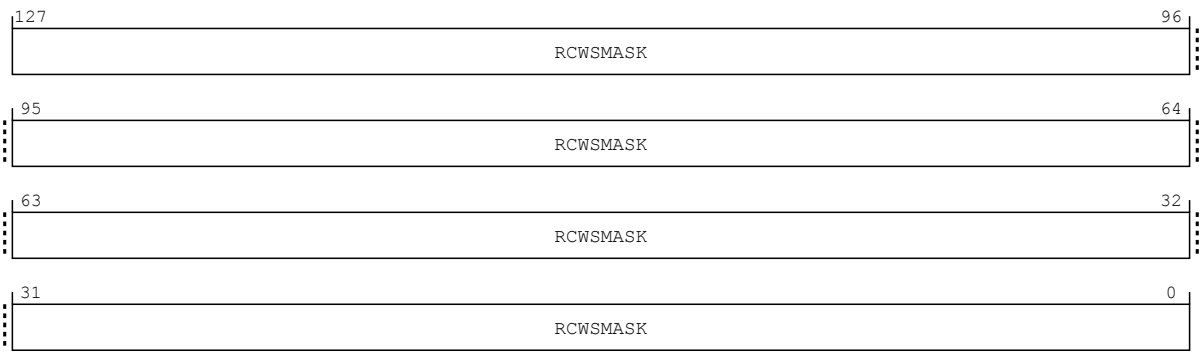

RCWSMASK, bits [127:0]

Software Mask used to decide which bit-fields are writable to the 128-bit Descriptor by RCWS instructions.

If RCWSMASK_EL1 is indirectly read by 128-bit variants of RCWS instructions:

- The Effective value of RCWSMASK[n] is the same as RCWSMASK_EL1[n], except as follows:
  - If n >= 17, and n <= 55, the Effective value of RCWSMASK[n] is the same as RCWSMASK_EL1[16].
  - If n is in {126:125, 120:119, 114, 107:101, 90:56, 1:0}, the Effective value of RCWSMASK[n] is 0.
  - If n >= 121, n <= 124, and FEAT_S1POE is not implemented, the Effective value of RCWSMASK[n] is 0.
  - If FEAT_MEC is not implemented, the Effective value of RCWSMASK bit RCWSMASK_EL1[108] is 0.

If RCWSMASK_EL1 is indirectly read by 64-bit variants of RCWS instructions:

- The Effective value of RCWSMASK[n] is the same as RCWSMASK_EL1[n], except as follows:
  - If n >= 18, and n <= 49, the Effective value of RCWSMASK[n] is the same as RCWSMASK_EL1[17].
  - If n == 52 and Protection is enabled, the Effective value of RCWSMASK[52] is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RCWSMASK_EL1 register bits {126:125, 120:119, 114, 107:101, 90:64, 49:18, 0} are RES0.

If FEAT_S1POE is not implemented, RCWSMASK_EL1 register bits {124:121} are RES0.

If FEAT_MEC is not implemented, RCWSMASK_EL1[108] is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

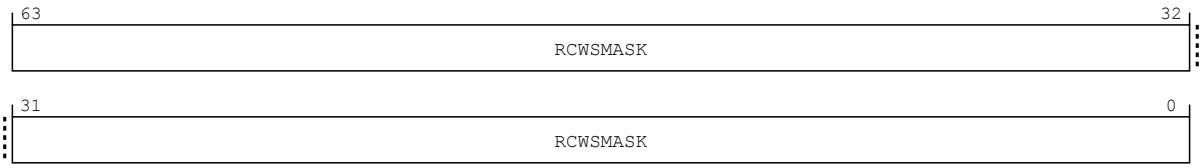

RCWSMASK, bits [63:0]

Software Mask used to decide which bit-fields are writable to the 64-bit Descriptor by RCWS Instruction.

The Effective value of RCWSMASK[n] is the same as RCWSMASK_EL1[n], except as follows

- If n >= 18, and n <= 49, the Effective value of RCWSMASK[n] is the same as RCWSMASK_EL1[17].
- If n == 52 and Protection is enabled, the Effective value of RCWSMASK[52] is 0.

RCWSMASK_EL1 register bits {49:18, 0} are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing RCWSMASK_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RCWSMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nRCWSMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = RCWSMASK_EL1()[63:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = RCWSMASK_EL1()[63:0];
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = RCWSMASK_EL1()[63:0];
end;
```

MSR RCWSMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nRCWSMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        RCWSMASK_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        RCWSMASK_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    RCWSMASK_EL1()[63:0] = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, RCWSMASK_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
        ↪'0') || HFGRTR2_EL2().nRCWSMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = RCWSMASK_EL1()[127:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = RCWSMASK_EL1()[127:0];
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = RCWSMASK_EL1()[127:0];
end;
```

When FEAT_D128 is implemented MSRR RCWSMASK_EL1, <Xt>, <Xt+1>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_THE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
         ↪'0') || HFGWTR2_EL2().nRCWSMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        RCWSMASK_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().RCWMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().RCWMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        RCWSMASK_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    RCWSMASK_EL1()[127:0] = X{128}(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.155 REVIDR_EL1, Revision ID Register

The REVIDR_EL1 characteristics are:

Purpose

Provides implementation-specific minor revision information.

Configuration

If REVIDR_EL1 has the same value as MIDR_EL1, then its contents have no significance.

AArch64 System register REVIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register REVIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to REVIDR_EL1 are UNDEFINED.

Attributes

REVIDR_EL1 is a 64-bit register.

Field descriptions

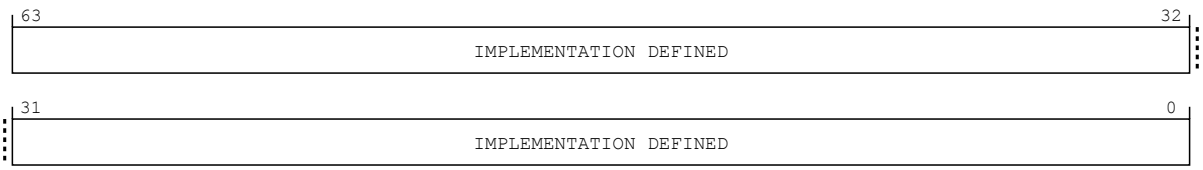

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing REVIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, REVIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && HCR_EL2().TID1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().REVIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = REVIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = REVIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = REVIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.156 RGSR_EL1, Random Allocation Tag Seed Register.

The RGSR_EL1 characteristics are:

Purpose

Random Allocation Tag Seed Register.

Configuration

When GCR_EL1.RRND is 1, updates to RGSR_EL1 are implementation-specific.

Direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to RGSR_EL1 are UNDEFINED.

Attributes

RGSR_EL1 is a 64-bit register.

Field descriptions

When GCR_EL1.RRND == '0':

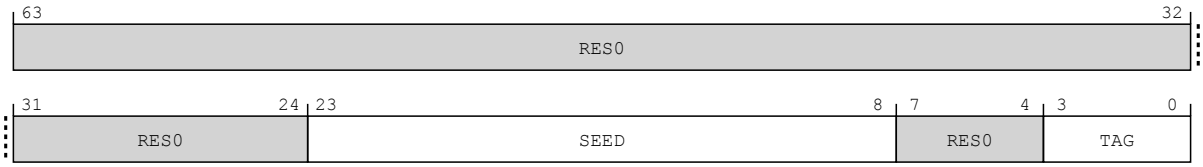

Bits [63:24]

Reserved, RES0.

SEED, bits [23:8]

Seed register used for generating values returned by `RandomTag()`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

TAG, bits [3:0]

Tag generated by the most recent IRG instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

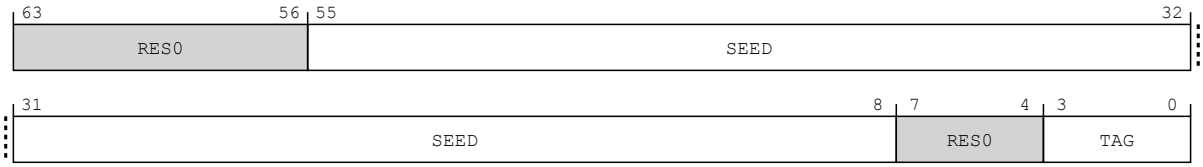

Bits [63:56]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [55:8]

IMPLEMENTATION DEFINED.

Note

Software is recommended to avoid writing SEED[15:0] with a value of zero, unless this has been generated by the PE in response to an earlier value with SEED being nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

TAG, bits [3:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing RGSR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RGSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
        ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = RGSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
     ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = RGSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = RGSR_EL1();
end;
```

```
MSR RGSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_MTE2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
     ↪'1') then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
     ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        RGSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
     ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        RGSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    RGSR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.157 RMR_EL1, Reset Management Register (EL1)

The RMR_EL1 characteristics are:

Purpose

When this register is implemented:

- A write to the register at EL1 can request a Warm reset.
- If EL1 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configuration

When EL1 is the highest implemented Exception level:

- If EL1 can use all Execution states then this register must be implemented.
- If EL1 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When the highest implemented Exception level is EL1, AArch64 System register RMR_EL1 bits [31:0] are architecturally mapped to AArch32 System register RMR[31:0].

This register is present only when the highest implemented Exception level is EL1 and FEAT_AA64 is implemented. Otherwise, direct accesses to RMR_EL1 are UNDEFINED.

Attributes

RMR_EL1 is a 64-bit register.

Field descriptions

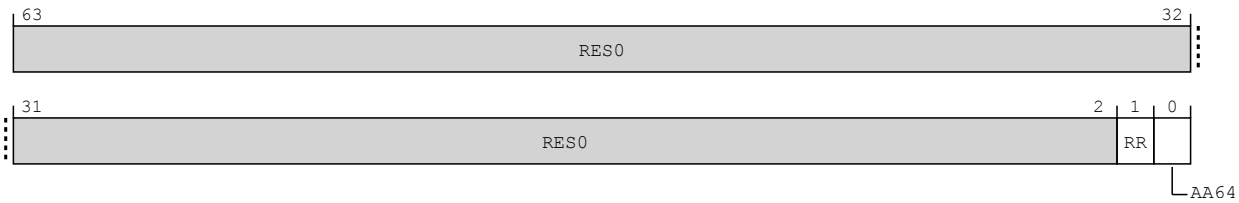

Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

AA64, bit [0]

When Implementation can reset into AArch32 state:

Determines which Execution state the PE boots into after a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AA64	Meaning
0b0	AArch32.
0b1	AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

The reset behavior of this field is:

- On a Cold reset, this field resets to '1'.

Otherwise:

Reserved, RAO/WI.

Accessing RMR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RMR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b010

```
if !(IsHighestEL(EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL1 && IsHighestEL(EL1) then
    X{64}(t) = RMR_EL1();
else
    Undefined();
end;
```

```
MSR RMR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b010

```
if !(IsHighestEL(EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL1 && IsHighestEL(EL1) then
    RMR_EL1() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.158 RMR_EL2, Reset Management Register (EL2)

The RMR_EL2 characteristics are:

Purpose

When this register is implemented:

- A write to the register at EL2 can request a Warm reset.
- If EL2 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configuration

When EL2 is the highest implemented Exception level:

- If EL2 can use all Execution states then this register must be implemented.
- If EL2 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

When the highest implemented Exception level is EL2, AArch64 System register RMR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HRMR[31:0].

This register is present only when the highest implemented Exception level is EL2 and FEAT_AA64 is implemented. Otherwise, direct accesses to RMR_EL2 are UNDEFINED.

Attributes

RMR_EL2 is a 64-bit register.

Field descriptions

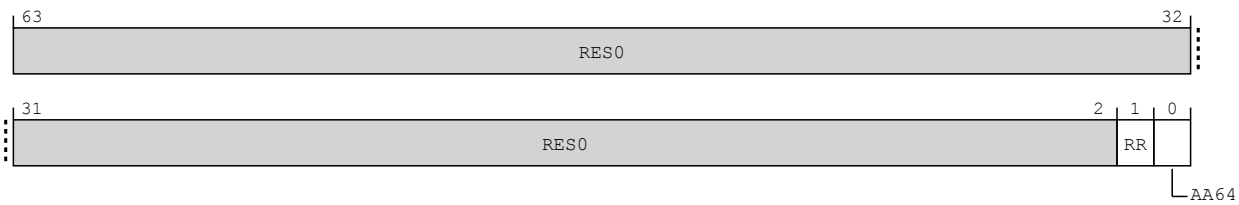

Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

AA64, bit [0]

When Implementation can reset into AArch32 state:

Determines which Execution state the PE boots into after a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AA64	Meaning
0b0	AArch32.
0b1	AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

The reset behavior of this field is:

- On a Cold reset, this field resets to '1'.

Otherwise:

Reserved, RAO/WI.

Accessing RMR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RMR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b010

```
if !(IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL1 && IsHighestEL(EL2) && EffectiveHCR_EL2_NVx() IN {'xx1'} then
    AArch64_SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    X{64}(t) = RMR_EL2();
else
    Undefined();
end;
```

```
MSR RMR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b010

```
if !(IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL1 && IsHighestEL(EL2) && EffectiveHCR_EL2_NVx() IN {'xx1'} then
    AArch64_SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    RMR_EL2() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.159 RMR_EL3, Reset Management Register (EL3)

The RMR_EL3 characteristics are:

Purpose

If EL3 is implemented and this register is implemented:

- A write to the register at EL3 can request a Warm reset.
- If EL3 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configuration

When EL3 is implemented:

- If EL3 can use all Execution states then this register must be implemented.
- If EL3 cannot use AArch32, then it is IMPLEMENTATION DEFINED whether the register is implemented.

Otherwise, direct accesses to RMR_EL3 are UNDEFINED.

When EL3 is implemented, AArch64 System register RMR_EL3 bits [31:0] are architecturally mapped to AArch32 System register RMR[31:0].

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to RMR_EL3 are UNDEFINED.

Attributes

RMR_EL3 is a 64-bit register.

Field descriptions

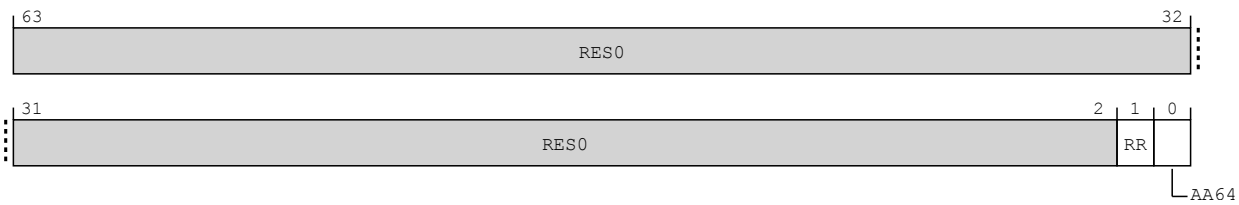

Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

AA64, bit [0]

When Implementation can reset into AArch32 state:

Determines which Execution state the PE boots into after a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AA64	Meaning
0b0	AArch32.
0b1	AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

The reset behavior of this field is:

- On a Cold reset, this field resets to '1'.

Otherwise:

Reserved, RAO/WI.

Accessing RMR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RMR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL3 && IsHighestEL(EL3) then
    X{64}(t) = RMR_EL3();
else
    Undefined();
end;
```

```
MSR RMR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL3 && IsHighestEL(EL3) then
    RMR_EL3() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.160 RNDR, Random Number

The RNDR characteristics are:

Purpose

Random Number. Returns a 64-bit random number from an approved Random Bit Generator, where the Deterministic Random Bit Generator within the Random Bit Generator is reseeded from an approved entropy source at an IMPLEMENTATION DEFINED rate. See 'Properties of the generated random number'.

If the hardware returns a genuine random number, PSTATE.NZCV is set to `0b0000`.

If the instruction cannot return a genuine random number in a reasonable period of time, PSTATE.NZCV is set to `0b0100` and the data value returned is 0.

Configuration

This register is present only when (FEAT_RNG is implemented or FEAT_RNG_TRAP is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to RNDR are UNDEFINED.

Attributes

RNDR is a 64-bit register.

Field descriptions

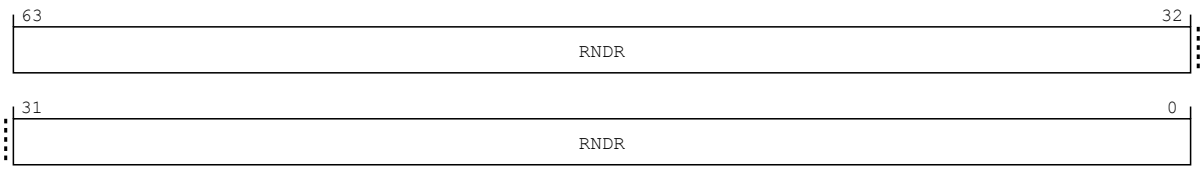

RNDR, bits [63:0]

Random Number. Returns a 64-bit Random Number from an approved Random Bit Generator, where the Deterministic Random Bit Generator within the Random Bit Generator is reseeded from an approved entropy source at an IMPLEMENTATION DEFINED rate. See 'Properties of the generated random number'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing RNDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RNDR
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0010	0b0100	0b000

```
if !((IsFeatureImplemented(FEAT_RNG) || IsFeatureImplemented(FEAT_RNG_TRAP)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDR();
    end;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDR();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDR();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDR();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.161 RNDRRS, Random Number Full Entropy

The RNDRRS characteristics are:

Purpose

Random Number with fresh full entropy. Returns a 64-bit random number from an approved Random Bit Generator, using either a Non-deterministic Random Bit Generator or one where the Deterministic Random Bit Generator is reseeded, where possible, from an approved entropy source before the return of the random number. See 'Properties of the generated random number'.

If the hardware returns a genuine random number, PSTATE.NZCV is set to `0b0000`.

If the instruction cannot return a genuine random number in a reasonable period of time, PSTATE.NZCV is set to `0b0100` and the data value returned is 0.

When FEAT_RNG_TRAP is implemented and SCR_EL3.TRNDR is 1, reads of this register are trapped to EL3.

Configuration

This register is present only when (FEAT_RNG is implemented or FEAT_RNG_TRAP is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to RNDRRS are UNDEFINED.

Attributes

RNDRRS is a 64-bit register.

Field descriptions

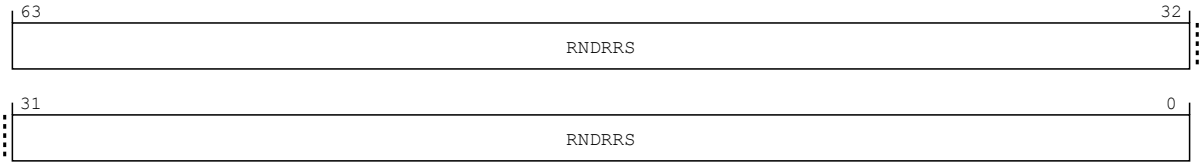

RNDRRS, bits [63:0]

Random Number with fresh full entropy. Returns a 64-bit random number from an approved Random Bit Generator, using either a Non-deterministic Random Bit Generator or one where the Deterministic Random Bit Generator is reseeded, where possible, from an approved entropy source before the return of the random number. See 'Properties of the generated random number'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing RNDRRS

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RNDRRS
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0010	0b0100	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !((IsFeatureImplemented(FEAT_RNG) || IsFeatureImplemented(FEAT_RNG_TRAP)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDRRS();
    end;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDRRS();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDRRS();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3().TRNDR == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        Undefined();
    else
        X{64}(t) = RNDRRS();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.162 RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)

The RVBAR_EL1 characteristics are:

Purpose

If EL1 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configuration

This register is present only when the highest implemented Exception level is EL1 and FEAT_AA64 is implemented. Otherwise, direct accesses to RVBAR_EL1 are UNDEFINED.

Attributes

RVBAR_EL1 is a 64-bit register.

Field descriptions

ResetAddress, bits [63:0]

The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing RVBAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RVBAR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b001

```
if !(IsHighestEL(EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL1 && IsHighestEL(EL1) then
    X{64}(t) = RVBAR_EL1();
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.163 RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)

The RVBAR_EL2 characteristics are:

Purpose

If EL2 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when the highest implemented Exception level is EL2 and FEAT_AA64 is implemented. Otherwise, direct accesses to RVBAR_EL2 are UNDEFINED.

Attributes

RVBAR_EL2 is a 64-bit register.

Field descriptions

ResetAddress, bits [63:0]

The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing RVBAR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RVBAR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b001

```
if !(IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL1 && IsHighestEL(EL2) && EffectiveHCR_EL2_NVx() IN {'xx1'} then
    AArch64_SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    X{64}(t) = RVBAR_EL2();
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.164 RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)

The RVBAR_EL3 characteristics are:

Purpose

If EL3 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configuration

Only implemented if the highest Exception level implemented is EL3.

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to RVBAR_EL3 are UNDEFINED.

Attributes

RVBAR_EL3 is a 64-bit register.

Field descriptions

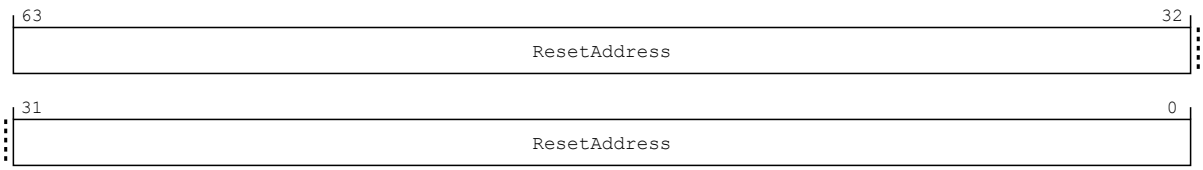

ResetAddress, bits [63:0]

The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing RVBAR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RVBAR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL3 && IsHighestEL(EL3) then
    X{64}(t) = RVBAR_EL3();
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.165 S2PIR_EL2, Stage 2 Permission Indirection Register (EL2)

The S2PIR_EL2 characteristics are:

Purpose

Stage 2 Permission Indirection Register for EL1&0 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_S2PIE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to S2PIR_EL2 are UNDEFINED.

Attributes

S2PIR_EL2 is a 64-bit register.

Field descriptions

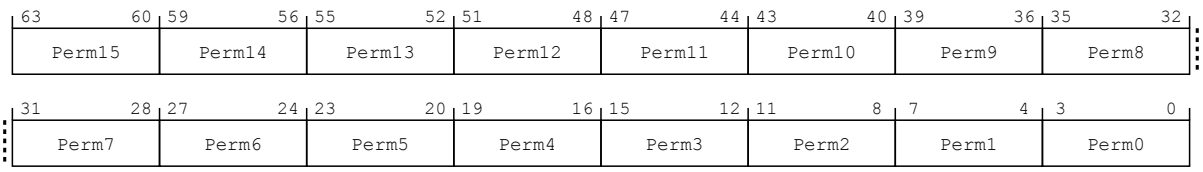

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Represents stage 2 Base Permissions.

Perm<m>	Meaning
0b0000	No Access.
0b0001	Reserved - treated as No Access.
0b0010	MRO.
0b0011	MRO-TL1.
0b0100	WO.
0b0101	Reserved - treated as No Access.
0b0110	MRO-TL0.
0b0111	MRO-TL01.
0b1000	RO.
0b1001	RO+uX.
0b1010	RO+pX.
0b1011	RO+puX.
0b1100	RW.
0b1101	RW+uX.
0b1110	RW+pX.
0b1111	RW+puX.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Perm<m>	Meaning

This field is permitted to be cached in a TLB.

When stage 2 Indirect Permission mechanism is disabled, the contents of this register are ignored.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing S2PIR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, S2PIR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_S2PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x2B0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = S2PIR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = S2PIR_EL2();
end;
```

```
MSR S2PIR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_S2PIE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x2B0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        S2PIR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    S2PIR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.166 S2POR_EL1, Stage 2 Permission Overlay Register (EL1)

The S2POR_EL1 characteristics are:

Purpose

Stage 2 Permission Overlay Register for EL1&0 translation regime.

Configuration

This register is present only when FEAT_S2POE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to S2POR_EL1 are UNDEFINED.

Attributes

S2POR_EL1 is a 64-bit register.

Field descriptions

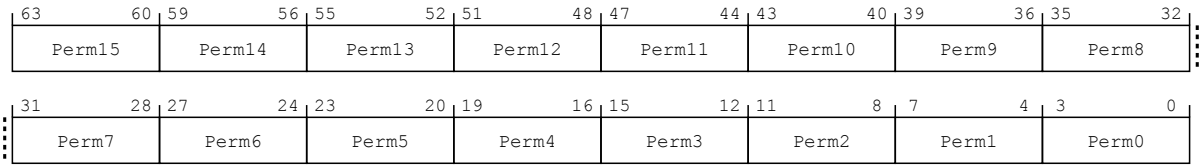

Perm<m> , bits [4m+3:4m], for m = 15 to 0

Configures stage 2 Overlay Permissions.

Perm<m>	Meaning
0b0000	No Access.
0b0001	Reserved - treated as No Access.
0b0010	MRO.
0b0011	MRO-TL1.
0b0100	WO.
0b0101	Reserved - treated as No Access.
0b0110	MRO-TL0.
0b0111	MRO-TL01.
0b1000	RO.
0b1001	RO+uX.
0b1010	RO+pX.
0b1011	RO+puX.
0b1100	RW.
0b1101	RW+uX.
0b1110	RW+pX.
0b1111	RW+puX.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is not permitted to be cached in a TLB.

When stage 2 Permission Overlay mechanism is disabled, this register is ignored.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing S2POR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, S2POR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_S2POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nS2POR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x2B8);
    else
        X{64}(t) = S2POR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = S2POR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = S2POR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR S2POR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_S2POE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nS2POR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x2B8) = X{64}(t);
    else
        S2POR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().PIEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().PIEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        S2POR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    S2POR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.167 S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED Registers

The S3_<op1><*Cn*><Cm>_<op2> characteristics are:

Purpose

This area of the instruction set space is reserved for IMPLEMENTATION DEFINED registers.

Configuration

When FEAT_SYSREG128 is implemented, each register in this space is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to S3_<op1>_<Cn>_<Cm>_<op2> are UNDEFINED.

Attributes

S3_<op1><*Cn*><Cm>_<op2> is a:

- 128-bit register when FEAT_SYSREG128 is implemented.
- 64-bit register otherwise.

Field descriptions

When FEAT_SYSREG128 is implemented:

127 … 96	95 … 64	63 … 32	31 … 0
IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [127:0]

IMPLEMENTATION DEFINED.

Otherwise:

63 … 32	31 … 0
IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing S3_<op1>_<Cn>_<Cm>_<op2>

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, S3_<op1>_C<Cn>_C<Cm>_<op2>
```

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL3 then
    AArch64_ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
end;
```

```
MSR S3_<op1>_C<Cn>_C<Cm>_<op2>, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        AArch64_ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
    end;
elsif PSTATE.EL == EL2 then
    AArch64_ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL3 then
    AArch64_ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
end;
```

When FEAT_SYSREG128 is implemented MRRS <Xt>, <Xt+1>, S3_<op1>_C<Cn>_C<Cm>_<op2>

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnIDCP128 == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        else
            AArch64_SystemAccessTrap(EL1, 0x14);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif !ELIsInHost(EL0) && (!IsSCTLR2EL1Enabled() || SCTLR2_EL1().EnIDCP128 == '0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        else
            AArch64_SystemAccessTrap(EL1, 0x14);
        end;
    elsif ELIsInHost(EL0) && (!IsSCTLR2EL2Enabled() || SCTLR2_EL2().EnIDCP128 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && !ELIsInHost(EL0) && (!IsHCRXEL2Enabled() || HCRX_EL2().EnIDCP128 == '0')
         ↪then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().EnIDCP128 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        AArch64_ImpDefSysRegRead128(op0, op1, CRn, CRm, op2, t, t2);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnIDCP128 == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().EnIDCP128 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().EnIDCP128 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        else
            AArch64_ImpDefSysRegRead128(op0, op1, CRn, CRm, op2, t, t2);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnIDCP128 == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnIDCP128 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        else
            AArch64_ImpDefSysRegRead128(op0, op1, CRn, CRm, op2, t, t2);
        end;
    elsif PSTATE.EL == EL3 then
        AArch64_ImpDefSysRegRead128(op0, op1, CRn, CRm, op2, t, t2);
    end;
```

When FEAT_SYSREG128 is implemented MSRR S3_<op1>_C<Cn>_C<Cm>_<op2>, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnIDCP128 == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().TIDCP == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        else
            AArch64_SystemAccessTrap(EL1, 0x14);
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif !ELIsInHost(EL0) && (!IsSCTLR2EL1Enabled() || SCTLR2_EL1().EnIDCP128 == '0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        else
            AArch64_SystemAccessTrap(EL1, 0x14);
        end;
    elsif ELIsInHost(EL0) && (!IsSCTLR2EL2Enabled() || SCTLR2_EL2().EnIDCP128 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && !ELIsInHost(EL0) && (!IsHCRXEL2Enabled() || HCRX_EL2().EnIDCP128 == '0')
         ↪then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().EnIDCP128 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        AArch64_ImpDefSysRegWrite128(op0, op1, CRn, CRm, op2, t, t2);
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnIDCP128 == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TIDCP == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().EnIDCP128 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().EnIDCP128 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        AArch64_ImpDefSysRegWrite128(op0, op1, CRn, CRm, op2, t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnIDCP128 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnIDCP128 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        AArch64_ImpDefSysRegWrite128(op0, op1, CRn, CRm, op2, t, t2);
    end;
elsif PSTATE.EL == EL3 then
    AArch64_ImpDefSysRegWrite128(op0, op1, CRn, CRm, op2, t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.168 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

Purpose

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0, EL1, and EL2. The Security state is Secure, Non-secure, or Realm.
- The Execution state at lower Exception levels.
- Whether IRQ, FIQ, SError exceptions, and External abort exceptions are taken to EL3.
- Whether various operations are trapped to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCR_EL3 are UNDEFINED.

Attributes

SCR_EL3 is a 64-bit register.

Field descriptions

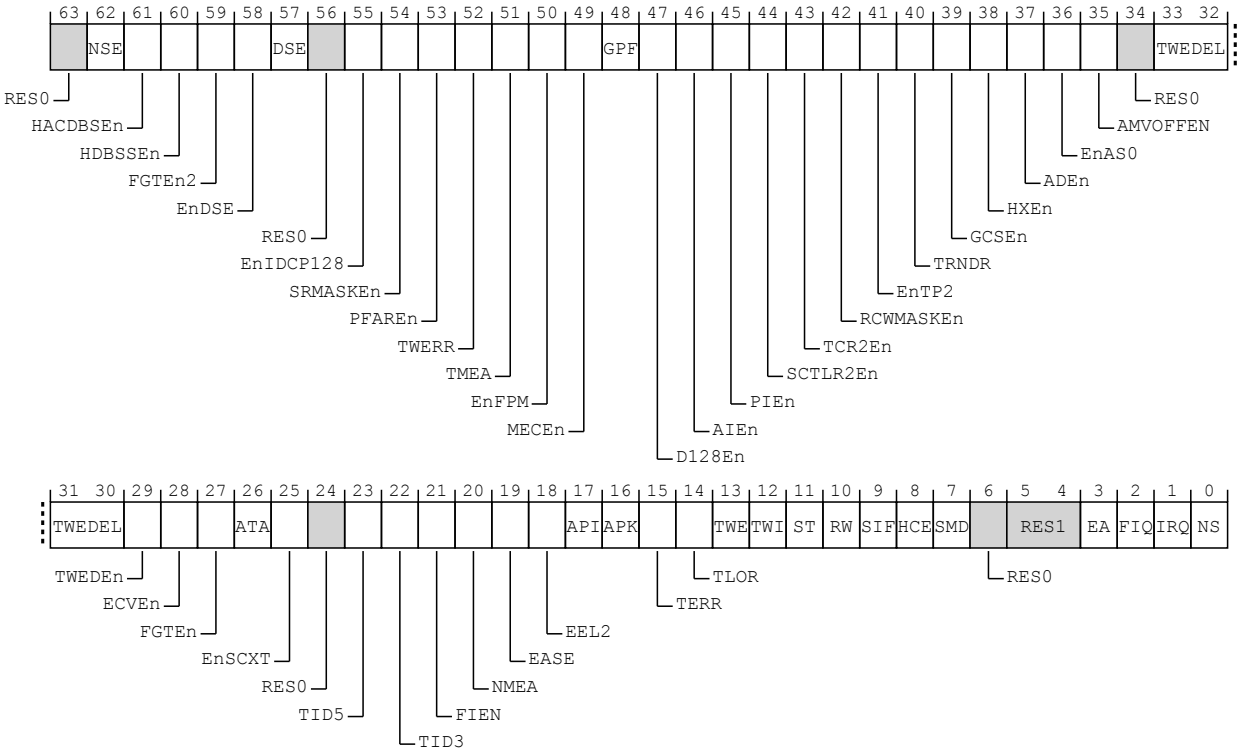

Bit [63]

Reserved, RES0.

NSE, bit [62]

When FEAT_RME is implemented:

This field, evaluated with SCR_EL3.NS, selects the Security state of EL2 and lower Exception levels.

For a description of the values derived by evaluating NS and NSE together, see SCR_EL3.NS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

The Effective value of this bit is 0.

Access to this field is RES0.

HACDBSEn, bit [61]

When FEAT_HACDBS is implemented:

Enables access to the HACDBSBR_EL2 and HACDBSCONS_EL2 registers at EL2.

HACDBSEn	Meaning
`0b0`	EL2 accesses to the specified registers are trapped to EL3.
`0b1`	This control does not cause any instructions to be trapped.

Traps are reported using EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HDBSSEn, bit [60]

When FEAT_HDBSS is implemented:

Enables access to HDBSSBR_EL2 and HDBSSPROD_EL2 registers at EL2.

HDBSSEn	Meaning
`0b0`	EL2 accesses to the specified registers are trapped to EL3.
`0b1`	This control does not cause any instructions to be trapped.

Traps are reported using EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FGTEn2, bit [59]

When FEAT_FGT2 is implemented:

Fine-Grained Traps Enable 2.

When EL2 is implemented, enables the traps to EL2 controlled by HDFGRTR2_EL2, HDFGWTR2_EL2, HFGITR2_EL2, HFGRTR2_EL2, and HFGWTR2_EL2, and controls access to those registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FGTEn2	Meaning
0b0	EL2 accesses to the specified registers are trapped to EL3. The values in these registers are treated as 0.
0b1	EL2 accesses to the specified registers are not trapped to EL3 by this mechanism.

Traps caused by accesses to the fine-grained trap registers are reported using EC syndrome value 0x18 and its associated ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnDSE, bit [58]

When FEAT_E3DSE is implemented:

Enable for delegated SError exceptions pended by SCR_EL3.DSE.

EnDSE	Meaning
0b0	Delegated SError exceptions pended by SCR_EL3.DSE are disabled.
0b1	Delegated SError exceptions pended by SCR_EL3.DSE are enabled.

This field is ignored by the PE and treated as zero if SCR_EL3.EA is 0 and the Effective value of SCR_EL3.TMEA is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DSE, bit [57]

When FEAT_E3DSE is implemented:

Delegated SError exception for EL2, EL1, and EL0.

DSE	Meaning
0b0	This mechanism is not making a delegated SError exception pending.
0b1	A delegated SError exception for EL2, EL1, and EL0 is pending because of this mechanism.

When EL2 is implemented and enabled in the current Security state, delegated SError exceptions pended by this field are affected by HCR_EL2.AMO and HCRX_EL2.TMEA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Virtual SError exceptions pended by HCR_EL2.VSE have priority over delegated SError exceptions pended by this field.

This field is ignored by the PE and treated as zero when SCR_EL3.EnDSE == 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [56]

Reserved, RES0.

EnIDCP128, bit [55]

When FEAT_SYSREG128 is implemented:

Enables access to IMPLEMENTATION DEFINED 128-bit System registers.

EnIDCP128	Meaning
`0b0`	Accesses at EL2, EL1, EL0 to IMPLEMENTATION DEFINED 128-bit System registers are trapped to EL3 using EC syndrome value `0x14`, unless the access generates a higher priority exception. Disables the functionality of the 128-bit IMPLEMENTATION DEFINED System registers that are accessible at EL3.
`0b1`	No accesses are trapped by this control.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SRMASKEn, bit [54]

When FEAT_SRMASK is implemented:

Enables access to the following MASK registers:

- CPACRMASK_EL1, and CPACRMASK_EL12.
- SCTLRMASK_EL1, and SCTLRMASK_EL12.
- SCTLR2MASK_EL1, and SCTLR2MASK_EL12.
- TCRMASK_EL1, and TCRMASK_EL12.
- TCR2MASK_EL1, and TCR2MASK_EL12.
- ACTLRMASK_EL1 and ACTLRMASK_EL12, if they are implemented.
- CPTRMASK_EL2.
- SCTLRMASK_EL2.
- SCTLR2MASK_EL2.
- TCRMASK_EL2.
- TCR2MASK_EL2.
- ACTLRMASK_EL2, if it is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SRMASKEn	Meaning
0b0	EL2 accesses to the specified registers are trapped to EL3. EL1 accesses to the specified EL1 registers are trapped to EL3. The values in the registers are treated as 0.
0b1	No accesses are trapped by this control.

Traps are reported using EC syndrome value 0x18.

Traps generated by this control have a lower priority than traps generated by the HCRX_EL2.SRMASKEn, HFGRTR2_EL2 and HFGWTR2_EL2 controls.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PFAREn, bit [53]

When FEAT_PFAR is implemented:

Enable access to Physical Fault Address Registers. When disabled, accesses to Physical Fault Address Registers generate a trap to EL3.

PFAREn	Meaning
0b0	Accesses of the specified Physical Fault Address Registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
0b1	This control does not cause any instructions to be trapped.

In AArch64 state, the instructions affected by this control are: MRS and MSR accesses to PFAR_EL1, PFAR_EL2, and PFAR_EL12.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3.

Trapped instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWERR, bit [52]

When FEAT_RASv2 is implemented:

Trap writes of Error Record registers. Enables a trap to EL3 on writes of Error Record registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TWERR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Writes of the specified Error Record registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are: MSR accesses to ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, ERXMISC2_EL1, ERXMISC3_EL1, and ERXSTATUS_EL1.

In AArch32 state, the instructions affected by this control are: MCR accesses to ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7, and ERXSTATUS.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Trapped AArch32 instructions are reported using EC syndrome value 0x03.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TMEA, bit [51]

When FEAT_DoubleFault2 is implemented:

Trap Masked External Aborts. Controls whether a masked error exception at a lower Exception level is taken to EL3.

TMEA	Meaning
0b0	Synchronous External abort exceptions and SError exceptions at EL2, EL1, and EL0 are unaffected by this mechanism. That is, these exceptions are not taken to EL3 unless routed to EL3 by another control.
0b1	When executing at Exception levels below EL3, all of the following apply: • When PSTATE.A is 1, synchronous External abort exceptions are taken to EL3, unless they are taken from EL1 or EL0 and routed to EL2 by another control. • Masked physical SError exceptions are taken to EL3, unless they are taken from EL1 or EL0 and routed to EL2 by another control.

This field has no effect on the routing of virtual or delegated SError exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnFPM, bit [50]

When FEAT_FPMR is implemented:

Enables direct and indirect accesses to FPMR from EL2, EL1, and EL0.

When accesses to FPMR are disabled by this control:

- Direct accesses to FPMR from EL2, EL1, and EL0 are trapped to EL3 and reported with EC syndrome value `0x18`.
- Execution of FP8 data-processing instructions that indirectly access FPMR is UNDEFINED at EL2, EL1 and EL0.

EnFPM	Meaning
`0b0`	Direct and indirect accesses to FPMR are disabled at EL2, EL1 and EL0.
`0b1`	This control does not cause any instructions to be trapped.

Traps are not taken if there is a higher priority exception generated by the access.

If EL3 is not implemented, the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MECEn, bit [49]

When FEAT_MEC is implemented:

Enables access to the following EL2 MECID registers, from EL2:

- MECID_P0_EL2.
- MECID_A0_EL2
- MECID_P1_EL2
- MECID_A1_EL2
- VMECID_P_EL2
- VMECID_A_EL2

Accesses to these registers are trapped and reported using EC syndrome value `0x18`.

MECEn	Meaning
`0b0`	EL2 accesses to any of the specified registers are trapped to EL3. The values of the specified registers are treated as 0 for all purposes other than direct reads or writes to the register from EL3.
`0b1`	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPF, bit [48]

When FEAT_RME is implemented:

Controls the reporting of Granule protection faults at EL0, EL1 and EL2.

GPF	Meaning
0b0	This control does not cause exceptions to be routed from EL0, EL1 or EL2 to EL3.
0b1	GPFs at EL0, EL1 and EL2 are routed to EL3 and reported as Granule Protection Check exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D128En, bit [47]

When FEAT_D128 is implemented:

128-bit System Register trap control. Enables access to 128-bit System Registers via MRRS, MSRR instructions.

MRRS and MSRR accesses from EL1 and EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value 0x14:

- TTBR0_EL1.
- TTBR1_EL1.
- RCWMASK_EL1, RCWSMASK_EL1.
- PAR_EL1.

MRRS and MSRR accesses from EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value 0x14:

- TTBR1_EL2 and accesses using the register name TTBR1_EL12.
- TTBR0_EL2 and accesses using the register name TTBR0_EL12.
- VTTBR_EL2.

D128En	Meaning
0b0	EL1 and EL2 accesses to the specified registers are disabled, and trapped to EL3.
0b1	This control does not cause any instructions to be trapped.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AIEn, bit [46]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AIE is implemented:

MAIR2_ELx, AMAIR2_ELx Register access trap control.

Accesses from EL1 and EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value `0x18`:

- AMAIR2_EL1.
- MAIR2_EL1.

Accesses from EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value `0x18`:

- AMAIR2_EL2 and accesses using the register name AMAIR2_EL12.
- MAIR2_EL2 and accesses using the register name MAIR2_EL12.

AIEn	Meaning
`0b0`	EL1 and EL2 accesses to the specified registers are disabled, and trapped to EL3. The values in these registers are treated as 0.
`0b1`	This control does not cause any instructions to be trapped.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PIEn, bit [45]

When FEAT_S1PIE is implemented, or FEAT_S2PIE is implemented, or FEAT_S1POE is implemented, or FEAT_S2POE is implemented:

Permission Indirection, Overlay Register access trap control. Enables access to Permission Indirection and Overlay registers.

Accesses from EL0, EL1 and EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value `0x18`:

- POR_EL0.

Accesses from EL1 and EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value `0x18`:

- PIRE0_EL1.
- PIR_EL1.
- POR_EL1.
- S2POR_EL1.

Accesses from EL2 using AArch64 to the following registers are trapped and reported using EC syndrome value `0x18`:

- PIRE0_EL2 and accesses using the register name PIRE0_EL12.
- PIR_EL2 and accesses using the register name PIR_EL12.
- POR_EL2 and accesses using the register name POR_EL12.
- S2PIR_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PIEn	Meaning
0b0	EL0, EL1 and EL2 accesses to the specified registers are disabled, and trapped to EL3.
0b1	This control does not cause any instructions to be trapped.

If this field is 0, it is IMPLEMENTATION SPECIFIC whether the values of the named registers are treated as zero.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SCTLR2En, bit [44]

When FEAT_SCTLR2 is implemented:

SCTLR2_ELx register trap control. Enables access to SCTLR2_EL1 and SCTLR2_EL2 registers.

SCTLR2En	Meaning
0b0	EL1 and EL2 accesses to SCTLR2_EL1 and SCTLR2_EL2 registers are disabled, and trapped to EL3. The values in these registers are treated as 0.
0b1	This control does not cause any instructions to be trapped.

Traps are reported using EC syndrome value 0x18.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCR2En, bit [43]

When FEAT_TCR2 is implemented:

TCR2_ELx register trap control. Enables access to TCR2_EL1 and TCR2_EL2 registers.

TCR2En	Meaning
0b0	EL1 and EL2 accesses to TCR2_EL1 and TCR2_EL2 registers are disabled, and trapped to EL3. The values in these registers are treated as 0.
0b1	This control does not cause any instructions to be trapped.

Traps are reported using EC syndrome value 0x18.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RCWMASKEn, bit [42]

When FEAT_THE is implemented:

RCW and RCWS Mask register trap control. Enables access to RCWMASK_EL1, RCWSMASK_EL1.

RCWMASKEn	Meaning
0b0	EL1 and EL2 accesses to RCWMASK_EL1 and RCWSMASK_EL1 registers are disabled, and trapped to EL3.
0b1	This control does not cause any instructions to be trapped.

Traps for MRS, MSR access are reported using EC syndrome value 0x18.

Traps for MRRS, MSRR acceess are reported using EC syndrome value 0x14.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnTP2, bit [41]

When FEAT_SME is implemented:

Traps instructions executed at EL2, EL1, and EL0 that access TPIDR2_EL0 to EL3. The exception is reported using EC syndrome value 0x18.

EnTP2	Meaning
0b0	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRNDR, bit [40]

When FEAT_RNG_TRAP is implemented:

Controls trapping of reads of RNDR and RNDRRS. The exception is reported using EC syndrome value 0x18.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRNDR	Meaning
0b0	This control does not cause RNDR and RNDRRS to be trapped. When FEAT_RNG is implemented: • ID_AA64ISAR0_EL1.RNDR returns the value 0b0001. When FEAT_RNG is not implemented: • ID_AA64ISAR0_EL1.RNDR returns the value 0b0000. • MRS reads of RNDR and RNDRRS are UNDEFINED.
0b1	ID_AA64ISAR0_EL1.RNDR returns the value 0b0001. Any attempt to read RNDR or RNDRRS is trapped to EL3.

When FEAT_RNG is not implemented, Arm recommends that SCR_EL3.TRNDR is initialized before entering Exception levels below EL3 and not subsequently changed.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

GCSEn, bit [39]

When FEAT_GCS is implemented:

Guarded Control Stack enable. Controls access to the Guarded Control Stack registers from EL2, EL1, and EL0, and controls whether the Guarded Control Stack is enabled.

The Guarded Control Stack registers trapped by this mechanism are:

- GCSCRE0_EL1.
- GCSCR_EL1.
- GCSCR_EL2.
- GCSCR_EL12.
- GCSPR_EL0.
- GCSPR_EL1.
- GCSPR_EL2.
- GCSPR_EL12.

GCSEn	Meaning
0b0	Trap read and write accesses to all Guarded Control Stack registers to EL3. All Guarded Control Stack behavior is disabled at EL2, EL1, and EL0.
0b1	This control does not cause any instructions to be trapped, and does not disable Guarded Control Stack behavior at EL2, EL1, or EL0.

Traps are reported using EC syndrome value 0x18.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

HXEn, bit [38]

When FEAT_HCX is implemented:

Enables access to the HCRX_EL2 register at EL2 from EL3.

HXEn	Meaning
0b0	Accesses at EL2 to HCRX_EL2 are trapped to EL3.
0b1	This control does not cause any instructions to be trapped.

When EL3 is not implemented, the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ADEn, bit [37]

When FEAT_LS64_ACCDATA is implemented:

Enables access to the ACCDATA_EL1 register at EL1 and EL2.

ADEn	Meaning
0b0	Accesses to ACCDATA_EL1 at EL1 and EL2 are trapped to EL3, unless the accesses are trapped to EL2 by the EL2 fine-grained trap.
0b1	This control does not cause accesses to ACCDATA_EL1 to be trapped.

If the HFGWTR_EL2.nACCDATA_EL1 or HFGRTR_EL2.nACCDATA_EL1 traps are enabled, they take priority over this trap.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnAS0, bit [36]

When FEAT_LS64_ACCDATA is implemented:

Traps execution of an ST64BV0 instruction at EL0, EL1, or EL2 to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnAS0	Meaning
0b0	EL0 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL1 by SCTLR_EL1.EnAS0, or to EL2 by either HCRX_EL2.EnAS0 or SCTLR_EL2.EnAS0. EL1 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL2 by HCRX_EL2.EnAS0. EL2 execution of an ST64BV0 instruction is trapped to EL3.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using EC syndrome value 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMVOFFEN, bit [35]

When FEAT_AMUv1p1 is implemented:

Activity Monitors Virtual Offsets Enable.

AMVOFFEN	Meaning
0b0	Accesses to AMEVCNTVOFF0<n>_EL2 and AMEVCNTVOFF1<n>_EL2 at EL2 are trapped to EL3. Indirect reads of the virtual offset registers are zero.
0b1	Accesses to AMEVCNTVOFF0<n>_EL2 and AMEVCNTVOFF1<n>_EL2 are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [34]

Reserved, RES0.

TWEDEL, bits [33:30]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCR_EL3.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCR_EL3.TWE as $2^{(\overline{\text{TWEDEL}} + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [29]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCR_EL3.TWE.

Traps are reported using EC syndrome value `0x01`.

TWEDEn	Meaning
`0b0`	The delay for taking the trap is IMPLEMENTATION DEFINED.
`0b1`	The delay for taking the trap is at least the number of cycles defined in SCR_EL3.TWEDEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ECVEn, bit [28]

When FEAT_ECV_POFF is implemented:

ECV Enable. Enables access to the CNTPOFF_EL2 register.

ECVEn	Meaning
`0b0`	EL2 accesses to CNTPOFF_EL2 are trapped to EL3, and the value of CNTPOFF_EL2 is treated as 0 for all purposes other than direct reads or writes to the register from EL3.
`0b1`	EL2 accesses to CNTPOFF_EL2 are not trapped to EL3 by this mechanism.

When FEAT_ECV_POFF is not implemented, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FGTEn, bit [27]

When FEAT_FGT is implemented:

Fine-Grained Traps Enable. When EL2 is implemented, enables the traps to EL2 controlled by HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2, and HFGWTR_EL2, and controls access to those registers.

Note

If EL2 is not implemented but EL3 is implemented, FEAT_FGT implements the MDCR_EL3.TDCC traps.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FGTEn	Meaning
0b0	EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are trapped to EL3, and the traps to EL2 controlled by those registers are disabled.
0b1	EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are not trapped to EL3 by this mechanism.

Traps caused by accesses to the fine-grained trap registers are reported using EC syndrome value 0x18 and its associated ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [26]

When FEAT_MTE2 is implemented:

Memory tagging enable override:

- Overrides enabling of Memory tagging at EL2, EL1 and EL0.
- If disabled by this control and not trapped to EL2, accesses to the following registers at EL1 and EL2 are trapped to EL3 and reported using EC syndrome value 0x18:
  - GCR_EL1.
  - RGSR_EL1.
  - TFSRE0_EL1.
  - TFSR_EL1.
  - TFSR_EL2.
  - Accesses with the register name TFSR_EL12 that are not UNDEFINED.

ATA	Meaning
0b0	Disables the use of Memory tagging at EL2, EL1 and EL0. The specified registers are trapped to EL3 unless trapped to a lower Exception level.
0b1	This field does not disable the use of Memory tagging at EL2, EL1 and EL0. The field does not trap the specified registers to EL3.

When EL3 is not implemented, the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSCXT, bit [25]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enables access to the SCXTNUM_EL2, SCXTNUM_EL1, and SCXTNUM_EL0 registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnSCXT	Meaning
0b0	Accesses at EL0, EL1 and EL2 to SCXTNUM_EL0, SCXTNUM_EL1, or SCXTNUM_EL2 registers are trapped to EL3 if they are not trapped by a higher priority exception, and the values of these registers are treated as 0.
0b1	This control does not cause any accesses to be trapped, or register values to be treated as 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [24]

Reserved, RES0.

TID5, bit [23]

When FEAT_IDTE3 is implemented and FEAT_MTE2 is implemented:

Trap ID group 5. EL2 and EL1 reads of the group 5 ID register GMID_EL1 are trapped to EL3, reported using EC syndrome value 0x18, unless the instruction generates a higher priority exception.

TID5	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL2 accesses to ID group 5 registers are trapped to EL3. When EL2 is not enabled in the current Security state, or when HCR_EL2.TID5 is 0, EL1 read accesses to the specified registers are trapped to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID3, bit [22]

When FEAT_IDTE3 is implemented:

Trap ID group 3. EL2 and EL1 reads of the following group 3 registers are trapped to EL3, reported using EC syndrome value 0x18, unless the instruction generates a higher priority exception:

- ID_PFR0_EL1, ID_PFR1_EL1, ID_DFR0_EL1, ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
- ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64AFR0_EL1, and ID_AA64AFR1_EL1.
- ID_PFR2_EL1, and ID_MMFR4_EL1.
- ID_MMFR5_EL1.
- ID_AA64MMFR3_EL1.
- ID_AA64MMFR4_EL1.
- ID_AA64PFR2_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR2_EL1 and ID_ISAR6_EL1.
- ID_DFR1_EL1.
- ID_AA64DFR2_EL1.
- ID_AA64ZFR0_EL1.
- ID_AA64SMFR0_EL1.
- ID_AA64FPFR0_EL1.
- ID_AA64ISAR2_EL1.
- ID_AA64ISAR3_EL1.
- This field traps all MRS accesses to registers in the following range that are not already mentioned in this field description: op0 == 3, op1 == 0, CRn == 0, CRm == {2-7}, op2 == {0-7}.

TID3	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL2 read accesses to ID group 3 registers are trapped to EL3. When EL2 is not enabled in the current Security state, or when HCR_EL2.TID3 is 0, EL1 read accesses to the specified registers are trapped to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FIEN, bit [21]

When FEAT_RASv1p1 is implemented:

Fault Injection enable. Trap accesses to the registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 from EL1 and EL2 to EL3, reported using EC syndrome value `0x18`.

FIEN	Meaning
`0b0`	Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.
`0b1`	This control does not cause any instructions to be trapped.

If EL3 is not implemented, the Effective value of SCR_EL3.FIEN is 1.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMEA, bit [20]

When FEAT_DoubleFault is implemented:

Non-maskable External Aborts. Controls whether PSTATE.A masks SError exceptions at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NMEA	Meaning
`0b0`	SError exceptions are not taken at EL3 if PSTATE.A == 1.
`0b1`	SError exceptions are taken at EL3 regardless of the value of PSTATE.A.

This field is ignored by the PE and treated as zero when all of the following are true:

- FEAT_DoubleFault2 is not implemented.
- SCR_EL3.EA is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EASE, bit [19]

When FEAT_DoubleFault is implemented:

External aborts to SError exception vector.

EASE	Meaning
`0b0`	Synchronous External abort exceptions taken to EL3 are taken to the appropriate synchronous exception vector offset from VBAR_EL3.
`0b1`	Synchronous External abort exceptions taken to EL3 are taken to the appropriate SError exception vector offset from VBAR_EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EEL2, bit [18]

When FEAT_SEL2 is implemented:

Secure EL2 Enable.

EEL2	Meaning
`0b0`	All behaviors associated with Secure EL2 are disabled. All registers, including timer registers, defined by FEAT_SEL2 are UNDEFINED, and those timers are disabled.
`0b1`	All behaviors associated with Secure EL2 are enabled.

When the value of this bit is 1, then:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When SCR_EL3.NS == 0, the SCR_EL3.RW bit is treated as 1 for all purposes other than reading or writing the register.
- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2, using EC syndrome value `0x03` :
  - A read or write of the SCR.
  - A read or write of the NSACR.
  - A read or write of the MVBAR.
  - A read or write of the SDCR.
  - Execution of an ATS12NSO** instruction.
- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2 using EC syndrome value `0x00` :
  - Execution of an SRS instruction that uses R13_mon.
  - Execution of an MRS (Banked register) or MSR (Banked register) instruction that would access SPSR_mon, R13_mon, or R14_mon.

Note

If the Effective value of SCR_EL3.EEL2 is 0, then these operations executed in Secure EL1 using AArch32 are trapped to EL3.

A Secure only implementation that does not implement EL3 but implements EL2, behaves as if SCR_EL3.EEL2 == 1.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

API, bit [17]

When FEAT_PAuth is implemented:

Controls the use of the following instructions related to Pointer Authentication.

- PACGA.
- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB, when any of the following are true:
  - In EL0, when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and the associated SCTLR_EL1.En<N><M> == 1.
  - In EL0, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and the associated SCTLR_EL2.En<N><M> == 1.
  - In EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  - In EL2, when the associated SCTLR_EL2.En<N><M> == 1.
- When FEAT_PAuth_LR is implemented, AUTIASPPC, AUTIASPPCR, AUTIA171615, AUTIBSPPC, AUTIBSPPCR, AUTIB171615, PACIASPPC, PACNBIASPPC, PACIA171615, PACIBSPPC, PACNBIBSPPC, PACIB171615, RETAASPPC, RETAASPPCR, RETABSPPC, RETABSPPCR, when any of the following are true:
  - In EL0, when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and the associated SCTLR_EL1.En<N><M> == 1.
  - In EL0, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and the associated SCTLR_EL2.En<N><M> == 1.
  - In EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  - In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

API	Meaning
0b0	The specified instructions are trapped to EL3, when the instructions are enabled, unless they are trapped to EL2 as a result of the higher priority HCR_EL2.API trap.
0b1	This control does not cause any instructions to be trapped.

Traps are reported using EC syndrome value 0x09.

An instruction is trapped only if Pointer Authentication is enabled for that instruction, for more information, see 'PAC generation and verification keys'.

Note

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APK, bit [16]

When FEAT_PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers, using EC syndrome value 0x18, from EL1 or EL2 to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1.
- APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1.
- APGAKeyLo_EL1, and APGAKeyHi_EL1.

APK	Meaning
0b0	Access to the registers holding "key" values for pointer authentication from EL1 or EL2 are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps.
0b1	This control does not cause any instructions to be trapped.

For more information, see 'PAC generation and verification keys'.

Note

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TERR, bit [15]

When FEAT_RAS is implemented:

Trap accesses of Error Record registers. Enables a trap to EL3 on accesses of Error Record registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TERR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Accesses of the specified Error Record registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
- MRS accesses to ERRIDR_EL1 and ERXFR_EL1.
- If FEAT_RASv1p1 is implemented, MRS and MSR accesses to ERXMISC2_EL1 and ERXMISC3_EL1.
- If FEAT_RASv2 is implemented, MRS accesses to ERXGSR_EL1.

In AArch32 state, the instructions affected by this control are:

- MRC and MCR accesses to ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
- MRC accesses to ERRIDR, ERXFR, and ERXFR2.
- If FEAT_RASv1p1 is implemented, MRC and MCR accesses to ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Trapped AArch32 instructions are reported using EC syndrome value 0x03.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLOR, bit [14]

When FEAT_LOR is implemented:

Trap LOR registers. Traps Non-secure and Realm accesses to the LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers from EL1 and EL2 to EL3, unless the access has been trapped to EL2.

TLOR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure and Realm EL1 and EL2 accesses to the LOR registers that are not UNDEFINED are trapped to EL3, unless it is trapped by HCR_EL2.TLOR.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWE, bit [13]

Traps EL2, EL1, and EL0 execution of WFE instructions to EL3, from any Security state and both Execution states, reported using EC syndrome value `0x01`.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

TWE	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE, HCR.TWE, SCTLR_EL1.nTWE, SCTLR_EL2.nTWE, or HCR_EL2.TWE.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFE instructions can cause the PE to enter a low-power state, see 'Wait for Event mechanism and Send event'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TWI, bit [12]

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from any Security state and both Execution states, reported using EC syndrome value `0x01`.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

TWI	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI, HCR.TWI, SCTLR_EL1.nTWI, SCTLR_EL2.nTWI, or HCR_EL2.TWI.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about when WFI instructions can cause the PE to enter a low-power state, see 'Wait for Interrupt'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ST, bit [11]

Traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3, from AArch64 state only, reported using EC syndrome value `0x18`.

ST	Meaning
`0b0`	Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are trapped to EL3 when Secure EL2 is disabled. If Secure EL2 is enabled, the behavior is as if the value of this field was 1.
`0b1`	This control does not cause any instructions to be trapped.

Note

Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3. These registers are not accessible at EL0.

When FEAT_RME is implemented and Secure state is not implemented, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RW, bit [10]

When FEAT_AA32EL1 is implemented:

Execution state control for lower Exception levels.

RW	Meaning
`0b0`	All lower Exception levels are using AArch32.
`0b1`	The next lower Exception level is using AArch64: • If EL2 is implemented and enabled in the Security state determined by SCR_EL3.{NSE,NS}, then EL2 is using AArch64 and EL2 controls the Execution state for EL1. • If EL2 is not implemented or EL2 is disabled in the Security state determined by SCR_EL3.{NSE,NS}, then EL1 is using AArch64. When executing at EL0, if the next higher Exception level is using AArch64, then the Execution state is determined by PSTATE.nRW. Otherwise, EL0 uses AArch32.

If any of the following apply, then the Effective value of this bit is 1:

- EL2 is implemented and does not support AArch32, and SCR_EL3.NS is 1.
- The Effective value of SCR_EL3.{EEL2,NS} is {1,0}.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAO/WI.

SIF, bit [9]

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction execution from memory marked in the first stage of translation as being Non-secure.

SIF	Meaning
0b0	Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is permitted.
0b1	Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is not permitted.

When FEAT_RME is implemented and Secure state is not implemented, this bit is RES0.

When FEAT_PAN3 is implemented, it is IMPLEMENTATION DEFINED whether SCR_EL3.SIF is also used to determine instruction access permission for the purpose of PAN.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HCE, bit [8]

Hypervisor Call instruction enable. Enables HVC instructions at EL3 and, if EL2 is enabled in the current Security state, at EL2 and EL1, in both Execution states, reported using EC syndrome value 0x00.

HCE	Meaning
0b0	HVC instructions are UNDEFINED.
0b1	HVC instructions are enabled at EL3, EL2, and EL1.

Note

HVC instructions are always UNDEFINED at EL0 and, if Secure EL2 is disabled, at Secure EL1. Any resulting exception is taken from the current Exception level to the current Exception level.

If EL2 is not implemented, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SMD, bit [7]

Secure Monitor Call disable. Disables SMC instructions at EL1 and above, from any Security state and both Execution states, reported using EC syndrome value 0x00.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SMD	Meaning
0b0	SMC instructions are enabled at EL3, EL2 and EL1.
0b1	SMC instructions are UNDEFINED.

Note

SMC instructions are always UNDEFINED at EL0. Any resulting exception is taken from the current Exception level to the current Exception level.
If HCR_EL2.TSC or HCR.TSC traps attempted EL1 execution of SMC instructions to EL2, that trap has priority over this disable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError exception routing.

EA	Meaning
0b0	When executing at Exception levels below EL3, External aborts and SError exceptions are not taken to EL3. In addition, when executing at EL3: • SError exceptions are not taken. • External aborts are taken to EL3.
0b1	When executing at any Exception level, External aborts and SError exceptions are taken to EL3.

This field has no effect on the routing of virtual or delegated SError exceptions.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FIQ, bit [2]

Physical FIQ Routing.

FIQ	Meaning
0b0	When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3. When executing at EL3, physical FIQ interrupts are not taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	When executing at any Exception level, physical FIQ interrupts are taken to EL3.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRQ, bit [1]

Physical IRQ Routing.

IRQ	Meaning
0b0	When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3. When executing at EL3, physical IRQ interrupts are not taken.
0b1	When executing at any Exception level, physical IRQ interrupts are taken to EL3.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [0]

When FEAT_RME is implemented:

Non-secure bit. This field is used in combination with SCR_EL3.NSE to select the Security state of EL2 and lower Exception levels.

NSE	NS	Meaning
0b0	0b0	Secure.
0b0	0b1	Non-secure.
0b1	0b0	Reserved.
0b1	0b1	Realm.

When Secure state is not implemented, SCR_EL3.NS is RES1 and its Effective value is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Non-secure bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	Indicates that EL0 and EL1 are in Secure state. When FEAT_SEL2 is implemented and SCR_EL3.EEL2 == 1, then EL2 is using AArch64 and in Secure state.
0b1	Indicates that Exception levels lower than EL3 are in Non-secure state, so memory accesses from those Exception levels cannot access Secure memory.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCR_EL3();
end;
```

```
MSR SCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    SCR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.169 SCTLR2_EL1, System Control Register (EL1)

The SCTLR2_EL1 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL1 and EL0.

Configuration

This register is present only when FEAT_SCTLR2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR2_EL1 are UNDEFINED.

Attributes

SCTLR2_EL1 is a 64-bit register.

Field descriptions

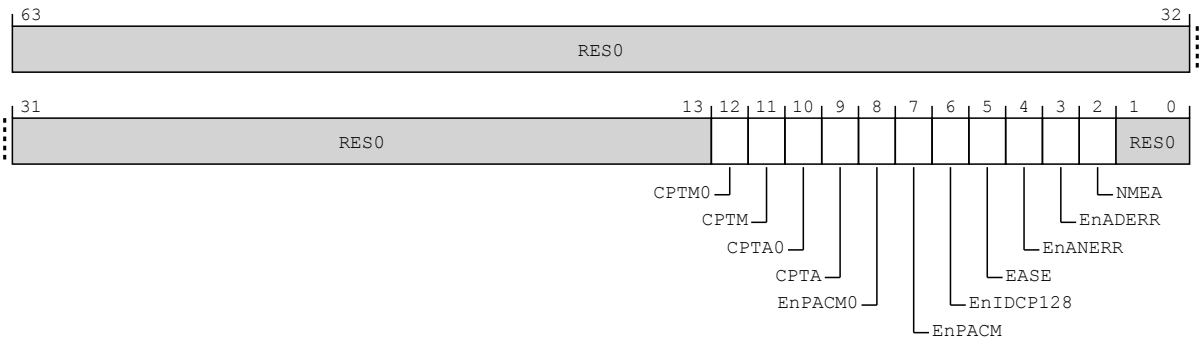

Bits [63:13]

Reserved, RES0.

CPTM0, bit [12]

When FEAT_CPA2 is implemented:

This field controls unprivileged Checked Pointer Arithmetic for Multiplication.

CPTM0	Meaning
0b0	Pointer Arithmetic for Multiplication is not checked.
0b1	Pointer Arithmetic for Multiplication is checked.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

If the Effective value of SCTLR2_EL1.CPTA0 is 0, then the Effective value of this field is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.CPTM0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CPTM, bit [11]

When FEAT_CPA2 is implemented:

This field controls Checked Pointer Arithmetic for Multiplication at EL1.

CPTM	Meaning
`0b0`	Pointer Arithmetic for Multiplication is not checked.
`0b1`	Pointer Arithmetic for Multiplication is checked.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

If the Effective value of SCTLR2_EL1.CPTA is 0, then the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.CPTM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CPTA0, bit [10]

When FEAT_CPA2 is implemented:

This field controls unprivileged Checked Pointer Arithmetic for Addition.

CPTA0	Meaning
`0b0`	Pointer Arithmetic for Addition is not checked.
`0b1`	Pointer Arithmetic for Addition is checked.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution.

This field is ignored by the PE and treated as zero when any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.CPTA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CPTA, bit [9]

When FEAT_CPA2 is implemented:

This field controls Checked Pointer Arithmetic for Addition at EL1.

CPTA	Meaning
`0b0`	Pointer Arithmetic for Addition is not checked.
`0b1`	Pointer Arithmetic for Addition is checked.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.CPTA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnPACM0, bit [8]

When FEAT_PAuth_LR is implemented:

PACM Enable at EL0. Controls the effect of a PACM instruction at EL0.

EnPACM0	Meaning
`0b0`	The effects of PACM are disabled at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	A PACM instruction at EL0 causes PSTATE.PACM to be set to 1.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.PACMEn == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.EnPACM0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnPACM, bit [7]

When FEAT_PAuth_LR is implemented:

PACM Enable at EL1. Controls the effect of a PACM instruction at EL1.

EnPACM	Meaning
0b0	The effects of PACM are disabled at EL1.
0b1	A PACM instruction at EL1 causes PSTATE.PACM to be set to 1.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.PACMEn == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.EnPACM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnIDCP128, bit [6]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SYSREG128 is implemented:

Enables access to IMPLEMENTATION DEFINED 128-bit System registers.

EnIDCP128	Meaning
`0b0`	Accesses at EL0 to IMPLEMENTATION DEFINED 128-bit System registers are trapped to EL1 using an ESR_EL1.EC value of `0x14`, unless the access generates a higher priority exception. Disables the functionality of the 128-bit IMPLEMENTATION DEFINED System registers that are accessible at EL1.
`0b1`	No accesses are trapped by this control.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.EnIDCP128 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EASE, bit [5]

When FEAT_DoubleFault2 is implemented:

External Aborts to SError exception vector.

EASE	Meaning
`0b0`	Synchronous External abort exceptions taken to EL1 are taken to the appropriate synchronous exception vector offset from VBAR_EL1.
`0b1`	Synchronous External abort exceptions taken to EL1 are taken to the appropriate SError exception vector offset from VBAR_EL1.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.EASE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnANERR, bit [4]

When FEAT_ANERR is implemented:

Enable Asynchronous Normal Read Error.

EnANERR	Meaning
`0b0`	External aborts on Normal memory reads generate synchronous Data Abort exceptions in the EL1&0 translation regime.
`0b1`	External aborts on Normal memory reads generate synchronous Data Abort or asynchronous SError exceptions in the EL1&0 translation regime.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 0 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Normal memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, SCTLR2_EL1.EnANERR is 0, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 0 might have a performance impact for Normal memory reads.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.EnSNERR is 1.

Otherwise, this field is ignored by the PE and treated as one when all of the following are true:

- FEAT_ADERR is implemented.
- ID_AA64MMFR3_EL1.ANERR reads as `0b0010`.
- SCTLR2_EL1.EnADERR is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.EnANERR == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnADERR, bit [3]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_ADERR is implemented:

Enable Asynchronous Device Read Error.

EnADERR	Meaning
`0b0`	External aborts on Device memory reads generate synchronous Data Abort exceptions in the EL1&0 translation regime.
`0b1`	External aborts on Device memory reads generate synchronous Data Abort or asynchronous SError exceptions in the EL1&0 translation regime.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 0 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Device memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, SCTLR2_EL1.EnADERR is 0, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 0 might have a performance impact for Device memory reads.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.EnSDERR is 1.

Otherwise, this field is ignored by the PE and treated as one when all of the following are true:

- FEAT_ANERR is implemented.
- ID_AA64MMFR3_EL1.ADERR reads as `0b0010`.
- SCTLR2_EL1.EnANERR is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.EnADERR == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

NMEA, bit [2]

When FEAT_DoubleFault2 is implemented:

Non-maskable External Aborts. Controls whether PSTATE.A masks physical SError exceptions at EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NMEA	Meaning
0b0	Physical SError exceptions are not taken at EL1 if PSTATE.A == 1, unless routed to a higher Exception level.
0b1	Physical SError exceptions are taken at EL1 regardless of the value of PSTATE.A, unless routed to a higher Exception level.

This field has no effect on the masking of virtual and delegated SError interrupts.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and SCR_EL3.SCTLR2En == 0.
- EL2 is implemented and enabled in the current Security state and the Effective value of HCRX_EL2.SCTLR2En is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLR2MASK_EL1.NMEA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [1:0]

Reserved, RES0.

Accessing SCTLR2_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name SCTLR2_EL1 or SCTLR2_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to SCTLR2_EL1 are masked by SCTLR2MASK_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x278);
        else
            X{64}(t) = SCTLR2_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = SCTLR2_EL2();
        else
            X{64}(t) = SCTLR2_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = SCTLR2_EL1();
    end;
```

```
MSR SCTLR2_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x278) = X{64}(t);
    else
        SCTLR2_EL1() = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SCTLR2_EL2() = X{64}(t);
    else
        SCTLR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SCTLR2_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x278);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCTLR2_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR2_EL1();
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_VHE is implemented MSR SCTLR2_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x278) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SCTLR2_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SCTLR2_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```

When FEAT_SRMASK is implemented MRS <Xt>, SCTLR2ALIAS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
            ↪'0') || HFGRTR2_EL2().nSCTLR2ALIAS_EL1 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x278);
        else
            X{64}(t) = SCTLR2_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = SCTLR2_EL2();
        else
            X{64}(t) = SCTLR2_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = SCTLR2_EL1();
    end;
```

When FEAT_SRMASK is implemented MSR SCTLR2ALIAS_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b111

```
    if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
            Undefined();
        elsif EL2Enabled() && HCR_EL2().TVM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
            ↪'0') || HFGWTR2_EL2().nSCTLR2ALIAS_EL1 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        NVMem(0x278) = X{64}(t);
    else
        SCTLR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SCTLR2_EL2() = X{64}(t);
    else
        SCTLR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.170 SCTLR2_EL2, System Control Register (EL2)

The SCTLR2_EL2 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL2.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, these controls also apply to execution at EL0.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SCTLR2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR2_EL2 are UNDEFINED.

Attributes

SCTLR2_EL2 is a 64-bit register.

Field descriptions

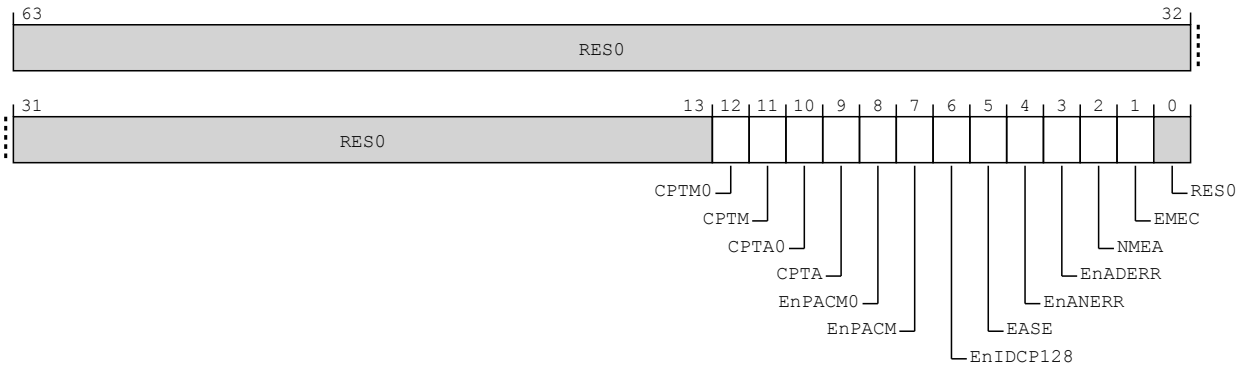

Bits [63:13]

Reserved, RES0.

CPTM0, bit [12]

When FEAT_CPA2 is implemented and EffectiveHCR_EL2_E2H() == '1':

This field controls unprivileged Checked Pointer Arithmetic for Multiplication.

CPTM0	Meaning
0b0	Pointer Arithmetic for Multiplication is not checked.
0b1	Pointer Arithmetic for Multiplication is checked.

If HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

If the Effective value of SCTLR2_EL2.CPTA0 is 0, then the Effective value of this field is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.CPTM0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CPTM, bit [11]

When FEAT_CPA2 is implemented:

This field controls Checked Pointer Arithmetic for Multiplication at EL2.

CPTM	Meaning
`0b0`	Pointer Arithmetic for Multiplication is not checked.
`0b1`	Pointer Arithmetic for Multiplication is checked.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

If the Effective value of SCTLR2_EL2.CPTA is 0, then the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.CPTM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CPTA0, bit [10]

When FEAT_CPA2 is implemented and EffectiveHCR_EL2_E2H() == '1':

This field controls unprivileged Checked Pointer Arithmetic for Addition.

CPTA0	Meaning
`0b0`	Pointer Arithmetic for Addition is not checked.
`0b1`	Pointer Arithmetic for Addition is checked.

If HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.CPTA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CPTA, bit [9]

When FEAT_CPA2 is implemented:

This field controls Checked Pointer Arithmetic for Addition at EL2.

CPTA	Meaning
`0b0`	Pointer Arithmetic for Addition is not checked.
`0b1`	Pointer Arithmetic for Addition is checked.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.CPTA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnPACM0, bit [8]

When FEAT_PAuth_LR is implemented and EffectiveHCR_EL2_E2H() == '1':

PACM Enable at EL0. Controls the effect of a PACM instruction at EL0.

EnPACM0	Meaning
`0b0`	The effects of PACM are disabled at EL0.
`0b1`	A PACM instruction at EL0 causes PSTATE.PACM to be set to 1.

If HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EnPACM0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnPACM, bit [7]

When FEAT_PAuth_LR is implemented:

PACM Enable at EL2. Controls the effect of a PACM instruction at EL2.

EnPACM	Meaning
`0b0`	The effects of PACM are disabled at EL2.
`0b1`	A PACM instruction at EL2 causes PSTATE.PACM to be set to 1.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EnPACM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnIDCP128, bit [6]

When FEAT_SYSREG128 is implemented:

Enables access to IMPLEMENTATION DEFINED 128-bit System registers.

EnIDCP128	Meaning
`0b0`	Accesses at EL0 to IMPLEMENTATION DEFINED 128-bit System registers are trapped to EL2 using an ESR_EL2.EC value of `0x14`, unless the access generates a higher priority exception. Disables the functionality of the 128-bit IMPLEMENTATION DEFINED System registers that are accessible at EL2.
`0b1`	No accesses are trapped by this control.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EnIDCP128 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EASE, bit [5]

When FEAT_DoubleFault2 is implemented:

External Aborts to SError exception vector.

EASE	Meaning
`0b0`	Synchronous External abort exceptions taken to EL2 are taken to the appropriate synchronous exception vector offset from VBAR_EL2.
`0b1`	Synchronous External abort exceptions taken to EL2 are taken to the appropriate SError exception vector offset from VBAR_EL2.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EASE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnANERR, bit [4]

When FEAT_ANERR is implemented:

Enable Asynchronous Normal Read Error.

EnANERR	Meaning
`0b0`	External aborts on Normal memory reads generate synchronous Data Abort exceptions in the EL2 and EL2&0 translation regimes.
`0b1`	External aborts on Normal memory reads generate synchronous Data Abort or asynchronous SError exceptions in the EL2 and EL2&0 translation regimes.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 0 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Normal memory read in every case. There might be implementation-specific

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, SCTLR2_EL2.EnANERR is 0, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 0 might have a performance impact for Normal memory reads.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

Otherwise, this field is ignored by the PE and treated as one when all of the following are true:

- FEAT_ADERR is implemented.
- ID_AA64MMFR3_EL1.ANERR reads as `0b0010`.
- SCTLR2_EL2.EnADERR is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EnANERR == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnADERR, bit [3]

When FEAT_ADERR is implemented:

Enable Asynchronous Device Read Error.

EnADERR	Meaning
`0b0`	External aborts on Device memory reads generate synchronous Data Abort exceptions in the EL2 and EL2&0 translation regimes.
`0b1`	External aborts on Device memory reads generate synchronous Data Abort or asynchronous SError exceptions in the EL2 and EL2&0 translation regimes.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 0 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Device memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, SCTLR2_EL2.EnADERR is 0, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 0 might have a performance impact for Device memory reads.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

Otherwise, this field is ignored by the PE and treated as one when all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_ANERR is implemented.
- ID_AA64MMFR3_EL1.ADERR reads as `0b0010`.
- SCTLR2_EL2.EnANERR is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EnADERR == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

NMEA, bit [2]

When FEAT_DoubleFault2 is implemented:

Non-maskable External Aborts. Controls whether physical SError exceptions are masked at EL2 when PSTATE.A is 1 or HCR_EL2.{AMO, TGE} is {0, 0}.

NMEA	Meaning
`0b0`	The masking of physical SError exceptions at EL2 is unaffected by this mechanism.
`0b1`	Physical SError exceptions are taken at EL2 regardless of whether they are masked by any mechanism, unless they are routed to EL3.

This field has no effect on the masking of delegated SError interrupts.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.NMEA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EMEC, bit [1]

When FEAT_MEC is implemented:

Enables MEC. When enabled, memory accesses to the Realm physical address space are associated with a MECID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EMEC	Meaning
0b0	MEC is not enabled for the Realm physical address space.
0b1	MEC is enabled for the Realm physical address space.

This bit is permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.SCTLR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLR2MASK_EL2.EMEC == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bit [0]

Reserved, RES0.

Accessing SCTLR2_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name SCTLR2_EL2 or SCTLR2_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to SCTLR2_EL2 are masked by SCTLR2MASK_EL2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLR2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = SCTLR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR2_EL2();
end;
```

MSR SCTLR2_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SCTLR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2_EL2() = X{64}(t);
end;
```

MRS <Xt>, SCTLR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x278);
    else
        X{64}(t) = SCTLR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SCTLR2_EL2();
    else
        X{64}(t) = SCTLR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR2_EL1();
end;
```

MSR SCTLR2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        NVMem(0x278) = X{64}(t);
    else
        SCTLR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SCTLR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SCTLR2_EL2() = X{64}(t);
    else
        SCTLR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.171 SCTLR2_EL3, System Control Register (EL3)

The SCTLR2_EL3 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL3.

Configuration

This register is present only when FEAT_SCTLR2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR2_EL3 are UNDEFINED.

Attributes

SCTLR2_EL3 is a 64-bit register.

Field descriptions

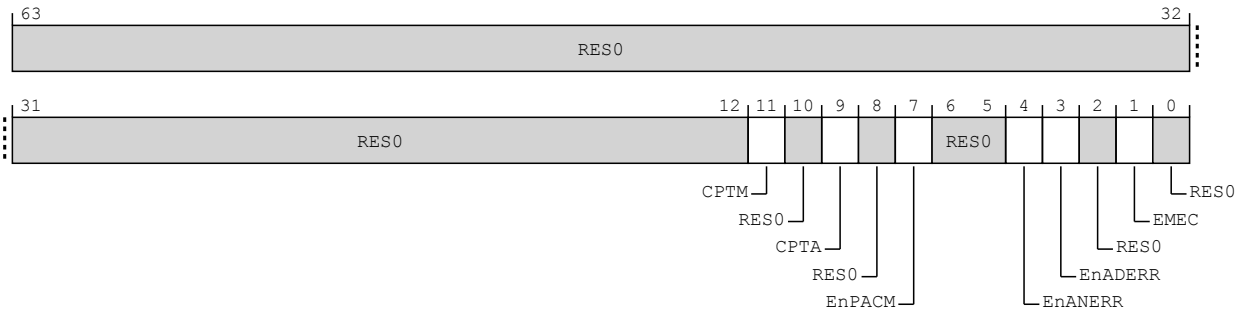

Bits [63:12]

Reserved, RES0.

CPTM, bit [11]

When FEAT_CPA2 is implemented:

This field controls Checked Pointer Arithmetic for Multiplication at EL3.

CPTM	Meaning
0b0	Pointer Arithmetic for Multiplication is not checked.
0b1	Pointer Arithmetic for Multiplication is checked.

If the Effective value of SCTLR2_EL3.CPTA is 0, then the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [10]

Reserved, RES0.

CPTA, bit [9]

When FEAT_CPA2 is implemented:

This field controls Checked Pointer Arithmetic for Addition at EL3.

CPTA	Meaning
`0b0`	Pointer Arithmetic for Addition is not checked.
`0b1`	Pointer Arithmetic for Addition is checked.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

EnPACM, bit [7]

When FEAT_PAuth_LR is implemented:

PACM Enable at EL3. Controls the effect of a PACM instruction at EL3.

EnPACM	Meaning
`0b0`	The effects of PACM are disabled at EL3.
`0b1`	A PACM instruction at EL3 causes PSTATE.PACM to be set to 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

EnANERR, bit [4]

When FEAT_ANERR is implemented:

Enable Asynchronous Normal Read Error.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnANERR	Meaning
0b0	External aborts on Normal memory reads generate synchronous Data Abort exceptions in the EL3 translation regime.
0b1	External aborts on Normal memory reads generate synchronous Data Abort or asynchronous SError exceptions in the EL3 translation regime.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 0 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Normal memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, SCTLR2_EL3.EnANERR is 0, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 0 might have a performance impact for Normal memory reads.

This field is ignored by the PE and treated as one when all of the following are true:

- FEAT_ADERR is implemented.
- ID_AA64MMFR3_EL1.ANERR reads as `0b0010`.
- SCTLR2_EL3.EnADERR is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EnADERR, bit [3]

When FEAT_ADERR is implemented:

Enable Asynchronous Device Read Error.

EnADERR	Meaning
0b0	External aborts on Device memory reads generate synchronous Data Abort exceptions in the EL3 translation regime.
0b1	External aborts on Device memory reads generate synchronous Data Abort or asynchronous SError exceptions in the EL3 translation regime.

Implementation-specific exceptions to applications of this field are described in 'Taking error exceptions'.

Setting this field to 0 does not guarantee that the PE is able to take a synchronous Data Abort exception for an External abort on a Device memory read in every case. There might be implementation-specific circumstances when an error on a load cannot be taken synchronously. These circumstances should be rare enough that treating such occurrences as fatal does not cause a significant increase in failure rate.

If FEAT_SVE is implemented, SCTLR2_EL3.EnADERR is 0, and the access generating the External abort is due to any Active element of an SVE Non-fault vector load instruction or an Active element that is not the

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

First active element of an SVE First-fault vector load instruction, then no exception is generated and the External abort is reported in the FFR.

Setting this field to 0 might have a performance impact for Device memory reads.

This field is ignored by the PE and treated as one when all of the following are true:

- FEAT_ANERR is implemented.
- ID_AA64MMFR3_EL1.ADERR reads as `0b0010`.
- SCTLR2_EL3.EnANERR is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [2]

Reserved, RES0.

EMEC, bit [1]

When FEAT_MEC is implemented:

Enables MEC. When enabled, memory accesses to the Realm physical address space are associated with MECID_RL_A_EL3.

EMEC	Meaning
`0b0`	MEC is not enabled for the Realm physical address space.
`0b1`	MEC is enabled for the Realm physical address space.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [0]

Reserved, RES0.

Accessing SCTLR2_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLR2_EL3
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR2_EL3();
end;
```

```
MSR SCTLR2_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_SCTLR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().SCTLR2_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        SCTLR2_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.172 SCTLR2MASK_EL1, Extended System Control Masking Register (EL1)

The SCTLR2MASK_EL1 characteristics are:

Purpose

Mask register to prevent updates of fields in SCTLR2_EL1 on writes to SCTLR2_EL1 or SCTLR2ALIAS_EL1.

Configuration

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR2MASK_EL1 are UNDEFINED.

Attributes

SCTLR2MASK_EL1 is a 64-bit register.

Field descriptions

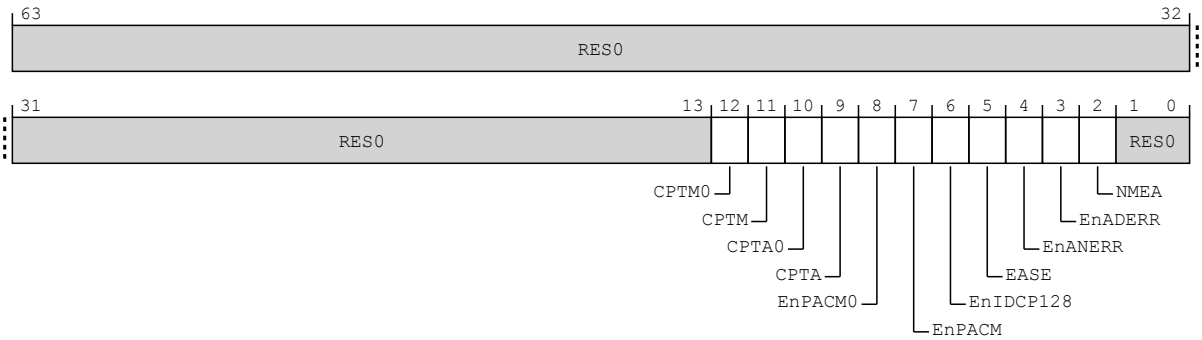

Bits [63:13]

Reserved, RES0.

CPTM0, bit [12]

When FEAT_CPA2 is implemented:

Mask bit for CPTM0.

CPTM0	Meaning
0b0	SCTLR2_EL1.CPTM0 is writable.
0b1	SCTLR2_EL1.CPTM0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CPTM, bit [11]

When FEAT_CPA2 is implemented:

Mask bit for CPTM.

CPTM	Meaning
0b0	SCTLR2_EL1.CPTM is writable.
0b1	SCTLR2_EL1.CPTM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CPTA0, bit [10]

When FEAT_CPA2 is implemented:

Mask bit for CPTA0.

CPTA0	Meaning
0b0	SCTLR2_EL1.CPTA0 is writable.
0b1	SCTLR2_EL1.CPTA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CPTA, bit [9]

When FEAT_CPA2 is implemented:

Mask bit for CPTA.

CPTA	Meaning
0b0	SCTLR2_EL1.CPTA is writable.
0b1	SCTLR2_EL1.CPTA is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnPACM0, bit [8]

When FEAT_PAuth_LR is implemented:

Mask bit for EnPACM0.

EnPACM0	Meaning
`0b0`	SCTLR2_EL1.EnPACM0 is writable.
`0b1`	SCTLR2_EL1.EnPACM0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnPACM, bit [7]

When FEAT_PAuth_LR is implemented:

Mask bit for EnPACM.

EnPACM	Meaning
`0b0`	SCTLR2_EL1.EnPACM is writable.
`0b1`	SCTLR2_EL1.EnPACM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIDCP128, bit [6]

When FEAT_SYSREG128 is implemented:

Mask bit for EnIDCP128.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnIDCP128	Meaning
0b0	SCTLR2_EL1.EnIDCP128 is writable.
0b1	SCTLR2_EL1.EnIDCP128 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EASE, bit [5]

When FEAT_DoubleFault2 is implemented:

Mask bit for EASE.

EASE	Meaning
0b0	SCTLR2_EL1.EASE is writable.
0b1	SCTLR2_EL1.EASE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnANERR, bit [4]

When FEAT_ANERR is implemented:

Mask bit for EnANERR.

EnANERR	Meaning
0b0	SCTLR2_EL1.EnANERR is writable.
0b1	SCTLR2_EL1.EnANERR is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnADERR, bit [3]

When FEAT_ADERR is implemented:

Mask bit for EnADERR.

EnADERR	Meaning
0b0	SCTLR2_EL1.EnADERR is writable.
0b1	SCTLR2_EL1.EnADERR is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMEA, bit [2]

When FEAT_DoubleFault2 is implemented:

Mask bit for NMEA.

NMEA	Meaning
0b0	SCTLR2_EL1.NMEA is writable.
0b1	SCTLR2_EL1.NMEA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [1:0]

Reserved, RES0.

Accessing SCTLR2MASK_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name SCTLR2MASK_EL1 or SCTLR2MASK_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SCTLR2MASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nSCTLR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x328);
    else
        X{64}(t) = SCTLR2MASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SCTLR2MASK_EL2();
    else
        X{64}(t) = SCTLR2MASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR2MASK_EL1();
end;
```

```
MSR SCTLR2MASK_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nSCTLR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x328) = X{64}(t);
    elsif !IsZero(SCTLR2MASK_EL1()) then
        Undefined();
    else
        SCTLR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(SCTLR2MASK_EL2()) then
            Undefined();
        else
            SCTLR2MASK_EL2() = X{64}(t);
        end;
    else
        SCTLR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2MASK_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SCTLR2MASK_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x328);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCTLR2MASK_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR2MASK_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR SCTLR2MASK_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x328) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SCTLR2MASK_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SCTLR2MASK_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.173 SCTLR2MASK_EL2, Extended System Control Masking Register (EL2)

The SCTLR2MASK_EL2 characteristics are:

Purpose

Mask register to prevent updates of fields in SCTLR2_EL2 on writes.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR2MASK_EL2 are UNDEFINED.

Attributes

SCTLR2MASK_EL2 is a 64-bit register.

Field descriptions

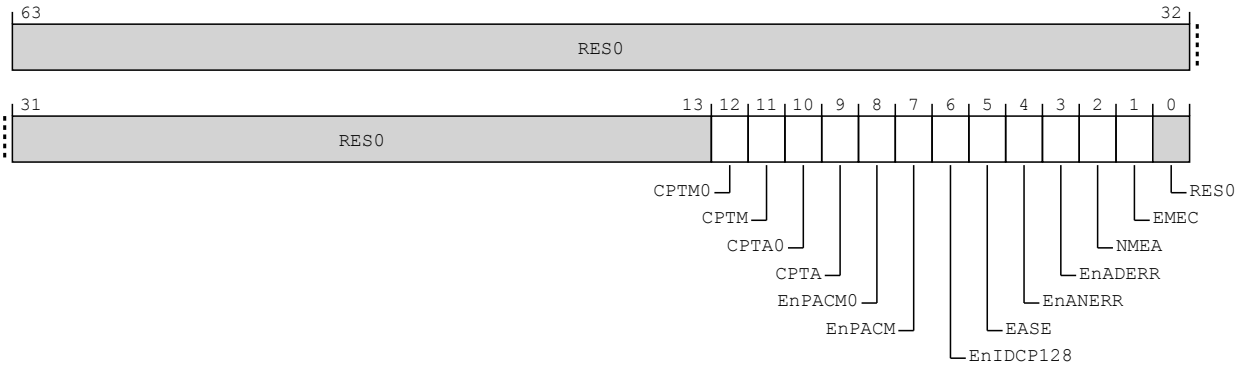

Bits [63:13]

Reserved, RES0.

CPTM0, bit [12]

When FEAT_CPA2 is implemented:

Mask bit for CPTM0.

CPTM0	Meaning
0b0	SCTLR2_EL2.CPTM0 is writable.
0b1	SCTLR2_EL2.CPTM0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

CPTM, bit [11]

When FEAT_CPA2 is implemented:

Mask bit for CPTM.

CPTM	Meaning
0b0	SCTLR2_EL2.CPTM is writable.
0b1	SCTLR2_EL2.CPTM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CPTA0, bit [10]

When FEAT_CPA2 is implemented:

Mask bit for CPTA0.

CPTA0	Meaning
0b0	SCTLR2_EL2.CPTA0 is writable.
0b1	SCTLR2_EL2.CPTA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CPTA, bit [9]

When FEAT_CPA2 is implemented:

Mask bit for CPTA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CPTA	Meaning
0b0	SCTLR2_EL2.CPTA is writable.
0b1	SCTLR2_EL2.CPTA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnPACM0, bit [8]

When FEAT_PAuth_LR is implemented:

Mask bit for EnPACM0.

EnPACM0	Meaning
0b0	SCTLR2_EL2.EnPACM0 is writable.
0b1	SCTLR2_EL2.EnPACM0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnPACM, bit [7]

When FEAT_PAuth_LR is implemented:

Mask bit for EnPACM.

EnPACM	Meaning
0b0	SCTLR2_EL2.EnPACM is writable.
0b1	SCTLR2_EL2.EnPACM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnIDCP128, bit [6]

When FEAT_SYSREG128 is implemented:

Mask bit for EnIDCP128.

EnIDCP128	Meaning
0b0	SCTLR2_EL2.EnIDCP128 is writable.
0b1	SCTLR2_EL2.EnIDCP128 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EASE, bit [5]

When FEAT_DoubleFault2 is implemented:

Mask bit for EASE.

EASE	Meaning
0b0	SCTLR2_EL2.EASE is writable.
0b1	SCTLR2_EL2.EASE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnANERR, bit [4]

When FEAT_ANERR is implemented:

Mask bit for EnANERR.

EnANERR	Meaning
0b0	SCTLR2_EL2.EnANERR is writable.
0b1	SCTLR2_EL2.EnANERR is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnADERR, bit [3]

When FEAT_ADERR is implemented:

Mask bit for EnADERR.

EnADERR	Meaning
`0b0`	SCTLR2_EL2.EnADERR is writable.
`0b1`	SCTLR2_EL2.EnADERR is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMEA, bit [2]

When FEAT_DoubleFault2 is implemented:

Mask bit for NMEA.

NMEA	Meaning
`0b0`	SCTLR2_EL2.NMEA is writable.
`0b1`	SCTLR2_EL2.NMEA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EMEC, bit [1]

When FEAT_MEC is implemented:

Mask bit for EMEC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EMEC	Meaning
0b0	SCTLR2_EL2.EMEC is writable.
0b1	SCTLR2_EL2.EMEC is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [0]

Reserved, RES0.

Accessing SCTLR2MASK_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name SCTLR2MASK_EL2 or SCTLR2MASK_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLR2MASK_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SCTLR2MASK_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR2MASK_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SCTLR2MASK_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsZero(SCTLR2MASK_EL2()) then
        Undefined();
    else
        SCTLR2MASK_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2MASK_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, SCTLR2MASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nSCTLR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x328);
    else
        X{64}(t) = SCTLR2MASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SCTLR2MASK_EL2();
    else
        X{64}(t) = SCTLR2MASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR2MASK_EL1();
end;
```

MSR SCTLR2MASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nSCTLR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x328) = X{64}(t);
    elsif !IsZero(SCTLR2MASK_EL1()) then
        Undefined();
    else
        SCTLR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(SCTLR2MASK_EL2()) then
            Undefined();
        else
            SCTLR2MASK_EL2() = X{64}(t);
        end;
    else
        SCTLR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR2MASK_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.174 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL1 and EL0.

Configuration

AArch64 System register SCTLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register SCTLR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR_EL1 are UNDEFINED.

Attributes

SCTLR_EL1 is a 64-bit register.

Field descriptions

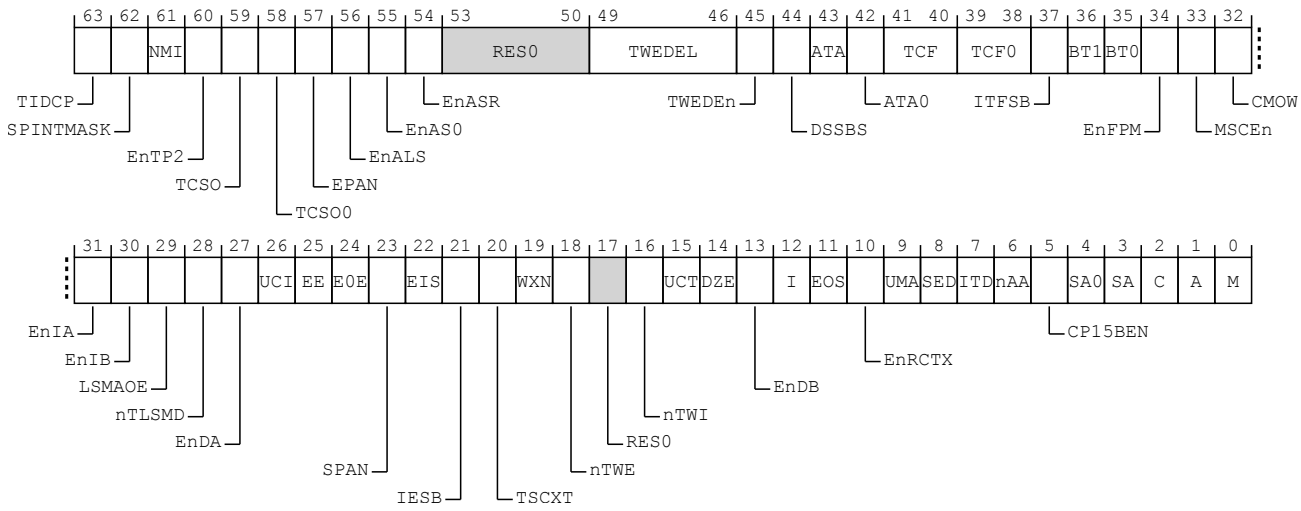

TIDCP, bit [63]

When FEAT_TIDCP1 is implemented:

Trap IMPLEMENTATION DEFINED functionality. When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL1.

TIDCP	Meaning
0b0	No instructions accessing the System register or System instruction spaces are trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TIDCP	Meaning

0b1	Instructions accessing the following System register or System instruction spaces are trapped to EL1 by this mechanism:
	• In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped:
	  • IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}, and are reported using EC syndrome value 0x18.
	  • IMPLEMENTATION DEFINED System instructions, which are accessed using SYSP, with CRn == {11, 15}, and are reported using EC syndrome value 0x14.
	  • IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome value 0x18.
	  • IMPLEMENTATION DEFINED System registers, which are accessed using MRRS and MSRR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome value 0x14.
	• In AArch32 state, EL0 MCR and MRC access to the following encodings are trapped and reported using EC syndrome value 0x03:
	  • All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
	  • All coproc==p15, CRn==c10, opc1 ==={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
	  • All coproc==p15, CRn==c11, opc1==={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TIDCP == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL1.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL1.

SPINTMASK	Meaning
0b0	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL1.
0b1	When PSTATE.SP is 1 and execution is at EL1, an IRQ or FIQ interrupt that is targeted to EL1 is masked regardless of any indication of Superpriority. PSTATE.ALLINT is set to 0 on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.SPINTMASK == ‘1’, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

NMI	Meaning
0b0	This control does not affect interrupt masking behavior.
0b1	This control enables all of the following: • The use of the PSTATE.ALLINT interrupt mask. • IRQ and FIQ interrupts to have Superpriority as an additional attribute. • PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.NMI == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnTP2, bit [60]

When FEAT_SME is implemented:

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using EC syndrome value 0x18.

EnTP2	Meaning
0b0	This control causes execution of these instructions at EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnTP2 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCSO, bit [59]

When FEAT_MTE_STORE_ONLY is implemented:

Tag Checking Store Only.

TCSO	Meaning
0b0	This field has no effect on Tag checking.
0b1	Explicit Memory Read Effects generated by instructions executed in EL1 are Tag Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TCSO == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCSO0, bit [58]

When FEAT_MTE_STORE_ONLY is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, Tag Checking Store Only in EL0.

TCSO0	Meaning
0b0	This field has no effect on Tag checking.
0b1	Explicit Memory Read Effects generated by instructions executed in EL0 are Tag Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TCSO0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EPAN, bit [57]

When FEAT_PAN3 is implemented:

Enhanced Privileged Access Never. When PSTATE.PAN is 1, determines whether an EL1 data access to a page with stage 1 EL0 instruction access permission generates a Permission fault as a result of the Privileged Access Never mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPAN	Meaning
0b0	No additional Permission faults are generated by this mechanism.
0b1	An EL1 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault. Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN = 1 if the accesses were not speculative, will not cause an allocation into a cache. When executing at EL1, this does not prevent unprivileged speculative accesses generated from the EL0 hardware-defined context from causing allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EPAN == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnALS, bit [56]

When FEAT_LS64 is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 to EL1.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using EC syndrome value 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnALS == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnAS0, bit [55]

When FEAT_LS64_ACCDATA is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps execution of an ST64BV0 instruction at EL0 to EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnAS0	Meaning
0b0	Execution of an ST64BV0 instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using EC syndrome value 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnAS0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnASR, bit [54]

When FEAT_LS64_V is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps execution of an ST64BV instruction at EL0 to EL1.

EnASR	Meaning
0b0	Execution of an ST64BV instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using EC syndrome value 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnASR == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [53:50]

Reserved, RES0.

TWEDEL, bits [49:46]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL1.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCTLR_EL1.nTWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TWEDEL == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCTLR_EL1.nTWE.

TWEDEn	Meaning
0b0	The delay for taking the trap is IMPLEMENTATION DEFINED.
0b1	The delay for taking the trap is at least the number of cycles defined in SCTLR_EL1.TWEDEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TWEDEn == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL1.
0b1	PSTATE.SSBS is set to 1 on an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.DSSBS == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access at EL1.

Controls use of Memory tagging at EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ATA	Meaning
0b0	Use of Memory tagging is disabled at EL1.
0b1	Use of Memory tagging is enabled at EL1.

The Effective value of this field is 0 if any of the following are true:

- The Effective value of SCR_EL3.ATA is 0.
- The Effective value of HCR_EL2.ATA is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.ATA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

ATA0, bit [42]

When FEAT_MTE2 is implemented:

Allocation Tag Access at EL0.

Controls access to Allocation Tags and Tag Check operations at EL0 when HCR_EL2,{E2H,TGE} is not {1,1}.

ATA0	Meaning
0b0	Use of Memory tagging is disabled at EL0.
0b1	Use of Memory tagging is enabled at EL0.

The Effective value of this field is 0 if any of the following are true:

- The Effective value of SCR_EL3.ATA is 0.
- The Effective value of HCR_EL2.ATA is 0.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.ATA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCF, bits [41:40]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_MTE2 is implemented:

Tag Check Fault at EL1. Controls the effect of Tag Check Faults due to Loads and Stores at EL1.

TCF	Meaning	Applies when
`0b00`	Tag Check Faults have no effect on the PE.	
`0b01`	Tag Check Faults cause a synchronous exception.	
`0b10`	Tag Check Faults are asynchronously accumulated.	
`0b11`	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	FEAT_MTE_ASYM_FAULT is implemented

If FEAT_MTE3 is not implemented, the value `0b11` is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TCF == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCF0, bits [39:38]

When FEAT_MTE2 is implemented:

Tag Check Fault at EL0. When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, controls the effect of Tag Check Faults due to Loads and Stores at EL0.

TCF0	Meaning	Applies when
`0b00`	Tag Check Faults have no effect on the PE.	
`0b01`	Tag Check Faults cause a synchronous exception.	
`0b10`	Tag Check Faults are asynchronously accumulated.	
`0b11`	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	FEAT_MTE_ASYM_FAULT is implemented

If FEAT_MTE3 is not implemented, the value `0b11` is reserved.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TCF0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When FEAT_MTE_ASYNC is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL1, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0_EL1 and TFSR_EL1 registers.

ITFSB	Meaning
0b0	Tag Check Faults are not synchronized on entry to EL1.
0b1	Tag Check Faults are synchronized on entry to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.ITFSB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

BT1, bit [36]

When FEAT_BTI is implemented:

Configures the Branch Type compatibility of the implicit BTI behavior for the following instructions at EL1:

- PACIASP.
- PACIBSP.
- If FEAT_PAuth_LR is implemented, PACIASPPC.
- If FEAT_PAuth_LR is implemented, PACIBSPPC.

BT1	Meaning
0b0	When the PE is executing at EL1, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.jc.
0b1	When the PE is executing at EL1, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.c.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.BT1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT0, bit [35]

When FEAT_BTI is implemented:

Configures the Branch Type compatibility of the implicit BTI behavior for the following instructions at EL0:

- `PACIASP`.
- `PACIBSP`.
- If FEAT_PAuth_LR is implemented, `PACIASPPC`.
- If FEAT_PAuth_LR is implemented, `PACIBSPPC`.

BT0	Meaning
`0b0`	When the PE is executing at EL0, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.jc.
`0b1`	When the PE is executing at EL0, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.c.

When the value of the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the value of SCTLR_EL1.BT0 has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.BT0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnFPM, bit [34]

When FEAT_FPMR is implemented:

Enables direct and indirect accesses to FPMR from EL0.

When accesses to FPMR are disabled by this control:

- Direct accesses to FPMR from EL0 are trapped to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1. These exceptions are reported using EC syndrome value `0x18`.
- Execution of FP8 data-processing instructions that indirectly access FPMR is UNDEFINED at EL0.

EnFPM	Meaning
`0b0`	Direct and indirect accesses to FPMR are disabled at EL0.
`0b1`	This control does not cause any instructions to be trapped.

Traps are not taken if there is a higher priority exception generated by the access.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0, and the SCTLR_EL2.EnFPM control is used for this purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnFPM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

MSCEn, bit [33]

When FEAT_MOPS is implemented:

Memory Copy and Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions at EL0.

MSCEn	Meaning
0b0	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.MSCEn == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CMOW, bit [32]

When FEAT_CMOW is implemented:

Controls the required permissions for the following cache maintenance instructions executed at EL0:

- Any DC instruction that operates by VA and performs a clean and invalidate operation.
- Any IC instruction that operates by VA.

CMOW	Meaning
0b0	This control does not cause any instruction to generate a stage 1 Permission fault.
0b1	For these instructions, when executed at EL0, the absence of stage 1 unprivileged write permission generates a stage 1 permission fault.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

For more information, see:

- Stage 1 permissions.
- Implications of enabling the dirty state management mechanism.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.CMOW == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIAKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnIA	Meaning
0b0	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIAKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIAKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACIA()` and `AuthIA()` pseudocode functions. When pointer authentication is enabled, `AddPACIA()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthIA()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnIA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIBKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnIB	Meaning
0b0	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIBKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIBKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACIB()` and `AuthIB()` pseudocode functions. When pointer authentication is enabled, `AddPACIB()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthIB()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnIB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
0b0	For all memory accesses at EL0, T32 and A32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
0b1	The ordering and interrupt behavior of T32 and A32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.LSMAOE == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
0b0	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
0b1	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.nTLSMD == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDAKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnDA	Meaning
`0b0`	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDAKey_EL1 key, is not enabled.
`0b1`	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDAKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACDA()` and `AuthDA()` pseudocode functions. When pointer authentication is enabled, `AddPACDA()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthDA()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnDA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when it is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`, as follows:

- DC CVAU, DC CIVAC, DC CVAC, and IC IVAU.
- If FEAT_MTE is implemented, DC CIGVAC, DC CIGDVAC, DC CGVAC, and DC CGDVAC.
- If FEAT_DPB is implemented, DC CVAP.
- If FEAT_DPB and FEAT_MTE are implemented, DC CGVAP and DC CGDVAP.
- If FEAT_DPB2 is implemented, DC CVADP.
- If FEAT_DPB2 and FEAT_MTE are implemented, DC CGVADP and DC CGDVADP.
- If FEAT_OCCMO is implemented, DC CIVAOC, DC CIGDVAOC, DC CVAOC and DC CGDVAOC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UCI	Meaning
0b0	For each of the specified instructions, if the execution of the instruction can be trapped, access at EL0 using AArch64 is trapped.
0b1	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.UCI == '1', access to this field is RO.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

EE	Meaning
0b0	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.
0b1	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

The EE bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EE == '1', access to this field is RO.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

Reserved, RES0.

E0E, bit [24]

When FEAT_MixedEndEL0 is implemented:

Endianness of data accesses at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E0E	Meaning
0b0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

This bit has no effect on the endianness of Explicit Memory Effects generated by unprivileged memory access instructions executed at EL1.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.E0E == '1', access to this field is RO.

When FEAT_BigEndEL0 is implemented:

Explicit data accesses at EL0 are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL0 are little-endian.

Reserved, RES0.

SPAN, bit [23]

When FEAT_PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1.

SPAN	Meaning
0b0	PSTATE.PAN is set to 1 on taking an exception to EL1.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.SPAN == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

EIS, bit [22]

When FEAT_ExS is implemented:

Exception Entry is a context synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EIS	Meaning
0b0	The taking of an exception to EL1 is not a context synchronization event.
0b1	The taking of an exception to EL1 is a context synchronization event.

If SCTLR_EL1.EIS is 0:

- Indirect writes to ESR_EL1, FAR_EL1, SPSR_EL1, ELR_EL1 are synchronized on exception entry to EL1, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.
- Some exception entries reported are not considered IFBEs per the memory model.

The following are not affected by the value of SCTLR_EL1.EIS:

- Changes to the PSTATE information on entry to EL1.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- The memory model requirement for exception entry to generate an Instruction Fetch Barrier Effect for some exception entries. See Basic definitions for the list of exception entries.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EIS == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable. Possible values are:

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL1. • Before the operational pseudocode of each Exception Return instruction executed at EL1.

If FEAT_DoubleFault2 is implemented, the PE is in Non-debug state, and the Effective value of SCTLR2_EL1.NMEA is 1, then SCTLR_EL1.IESB is ignored and the PE behaves as if SCTLR_EL1.IESB is 1 for all purposes other than direct read of the register.

The value of this field is ignored in Debug state and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.IESB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

TSCXT	Meaning
0b0	EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
0b1	EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The value of SCXTNUM_EL0 is treated as 0.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.TSCXT == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can restrict execute permissions on writable pages.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	In the EL1&0 translation regime, any region of memory that is writable at EL1 is XN at EL1, and any region of memory that is writable at EL0 is XN at EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

This field is RES0 if TCR2_EL1.PIE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.WXN == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using EC syndrome value `0x01`.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

nTWE	Meaning
`0b0`	Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
`0b1`	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.nTWE == '1', access to this field is RO.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using EC syndrome value `0x01`.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

nTWI	Meaning
`0b0`	Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
`0b1`	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.nTWI == '1', access to this field is RO.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`.

UCT	Meaning
`0b0`	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.UCT == '1', access to this field is RO.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`.

If FEAT_MTE is implemented, this trap also applies to DC GVA and DC GZVA.

DZE	Meaning
`0b0`	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions this trap applies to are not supported.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.DZE == '1', access to this field is RO.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDBKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnDB	Meaning
0b0	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDBKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDBKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACDB()` and `AuthDB()` pseudocode functions. When pointer authentication is enabled, `AddPACDB()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthDB()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnDB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

I, bit [12]

Stage 1 instruction access Cacheability control, for accesses at EL0 and EL1:

I	Meaning
0b0	All instruction access to Stage 1 Normal memory from EL0 and EL1 are Stage 1 Non-cacheable. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Stage 1 Cacheability of instruction access to Stage 1 Normal memory from EL0 and EL1. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.I == '1', access to this field is RO.

EOS, bit [11]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_ExS is implemented:

Exception Exit is a context synchronization event.

EOS	Meaning
0b0	An exception return from EL1 is not a context synchronization event
0b1	An exception return from EL1 is a context synchronization event

If SCTLR_EL1.EOS is set to 0:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL1 and ELR_EL1 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EOS == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Enable EL0 access to the following System instructions:

- CFPRCTX, DVPRCTX and CPPRCTX instructions.
- If FEAT_SPECRES2 is implemented, COSPRCTX.
- CFP RCTX, DVP RCTX and CPP RCTX instructions.
- If FEAT_SPECRES2 is implemented, COSP RCTX.

EnRCTX	Meaning
0b0	EL0 access to these instructions is disabled, and these instructions are trapped to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.
0b1	EL0 access to these instructions is enabled.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.EnRCTX == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

UMA, bit [9]

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value `0x18`.

UMA	Meaning
`0b0`	Any attempt at EL0 using AArch64 to execute an `MRS`, `MSR(register)`, or `MSR(immediate)` instruction that accesses the DAIF is trapped.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the value of this bit is treated as 0 for all purposes other than reading the value of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.UMA == '1', access to this field is RO.

SED, bit [8]

When FEAT_AA32EL0 is implemented:

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

SED	Meaning
`0b0`	SETEND instruction execution is enabled at EL0 using AArch32.
`0b1`	SETEND instructions are UNDEFINED at EL0 using AArch32 and any attempt at EL0 to access a SETEND instruction generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, reported using EC syndrome value `0x00`.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.SED == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

ITD, bit [7]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AA32EL0 is implemented:

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

ITD	Meaning
`0b0`	All IT instruction functionality is enabled at EL0 using AArch32.

`0b1`	Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates an exception, reported using EC syndrome value `0x00`, to EL1 or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1:
	• All encodings of the IT instruction with hw1[3:0]!=1000.
	• All encodings of the subsequent instruction with the following values for hw1:
	  • `0b11xxxxxxxxxxxxxx`: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
	  • `0b1011xxxxxxxxxxxx`: All instructions in 'Miscellaneous 16-bit instructions'.
	  • `0b10100xxxxxxxxxxx`: ADD Rd, PC, #imm
	  • `0b01001xxxxxxxxxxx`: LDR Rd, [PC, #imm]
	  • `0b0100x1xxx1111xxx`: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
	  • `0b010001xx1xxxx111`: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.
	These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.
	It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:
	• A 16-bit instruction, that can only be followed by another 16-bit instruction.
	• The first half of a 32-bit instruction.
	This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.
	An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1, then behavior is CONSTRAINED UNPREDICTABLE. For more information, see 'Changes to an ITD control by an instruction in an IT block'.

ITD is optional, but if it is implemented in the SCTLR_EL1, then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When an implementation does not implement ITD, access to this field is RAZ/WI.
- Access to this field is RO if all of the following are true:
  - FEAT_SRMASK is implemented
  - EL1 is the current Exception level
  - IsSystemRegisterMaskingEnabled(EL1)
  - SCTLRMASK_EL1.ITD == '1'

Otherwise:

Reserved, RES1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL1 and EL0 under certain conditions.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single naturally aligned 16-byte quantity, for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.
- If FEAT_LRCPC3 is implemented, the post index versions of LDAPR and the pre index versions of STLR.
- If FEAT_LRCPC3 and Advanced SIMD and floating-point instructions are implemented, LDAPUR (SIMD&FP), LDAP1 (SIMD&FP), STLUR (SIMD&FP), and STL1 (SIMD&FP). If FEAT_LRCPC3 is implemented, the following instructions generate an Alignment fault if all bytes being accessed for a single register are not within a single naturally aligned 16-byte quantity, for access:
- LDIAPP, STILP.

nAA	Meaning
`0b0`	Unaligned accesses by the specified instructions generate an Alignment fault.
`0b1`	This control does not generate Alignment faults.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register, or registers, being ZR, it is IMPLEMENTATION SPECIFIC whether this field behaves as 1 or the programmed value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.nAA == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

CP15BEN, bit [5]

When FEAT_AA32EL0 is implemented:

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from EL0:

CP15BEN	Meaning
`0b0`	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED and generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using EC syndrome value `0x00`.
`0b1`	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CP15BEN is optional, but if it is implemented in the SCTLR_EL1, then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When an implementation does not implement CP15BEN, access to this field is RAO/WI.
- Access to this field is RO if all of the following are true:
  - FEAT_SRMASK is implemented
  - EL1 is the current Exception level
  - IsSystemRegisterMaskingEnabled(EL1)
  - SCTLRMASK_EL1.CP15BEN == '1'

Otherwise:

Reserved, RES0.

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.SA0 == '1', access to this field is RO.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.SA == '1', access to this field is RO.

C, bit [2]

Stage 1 Cacheability control, for data accesses.

C	Meaning
0b0	All data access to Stage 1 Normal memory from EL0 and EL1, and all Normal memory accesses from unified cache to the EL1&0 Stage 1 translation tables, are treated as Stage 1 Non-cacheable.
0b1	This control has no effect on the Stage 1 Cacheability of: • Data access to Normal memory from EL0 and EL1. • Normal memory accesses to the EL1&0 Stage 1 translation tables.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the Effective value of HCR_EL2.DC bit in the current Security state is 1, the PE ignores SCTLR_EL1.C. This means that EL0 and EL1 data accesses to Normal memory are Cacheable.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.C == '1', access to this field is RO.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0.

A	Meaning
`0b0`	Alignment fault checking is disabled when executing at EL1 or EL0. Alignment checks on some instructions are not disabled by this control. For more information, see 'Alignment of data accesses'.
`0b1`	Alignment fault checking is enabled when executing at EL1 or EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the value of this bit is treated as 0 for all purposes other than reading the value of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.A == '1', access to this field is RO.

M, bit [0]

MMU enable for EL1&0 stage 1 address translation.

M	Meaning
`0b0`	EL1&0 stage 1 address translation disabled. See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.
`0b1`	EL1&0 stage 1 address translation enabled.

If the Effective value of HCR_EL2.{DC, TGE} in the current Security state is not {0, 0}, then the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and SCTLRMASK_EL1.M == '1', access to this field is RO.

Accessing SCTLR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `SCTLR_EL1` or `SCTLR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to SCTLR_EL1 are masked by SCTLRMASK_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x110);
    else
        X{64}(t) = SCTLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR_EL2();
    else
        X{64}(t) = SCTLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR_EL1();
end;
```

```
MSR SCTLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x110) = X{64}(t);
    else
        SCTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        SCTLR_EL2() = X{64}(t);
    else
        SCTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SCTLR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x110);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR SCTLR_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x110) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        SCTLR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SCTLR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```

When FEAT_SRMASK is implemented MRS <Xt>, SCTLRALIAS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
        ↪'0') || HFGRTR2_EL2().nSCTLRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x110);
    else
        X{64}(t) = SCTLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR_EL2();
    else
        X{64}(t) = SCTLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_SRMASK is implemented MSR SCTLRALIAS_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
        ↪'0') || HFGWTR2_EL2().nSCTLRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x110) = X{64}(t);
    else
        SCTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        SCTLR_EL2() = X{64}(t);
    else
        SCTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.175 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL2.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, these controls apply also to execution at EL0.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register SCTLR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HSCTLR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR_EL2 are UNDEFINED.

Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions

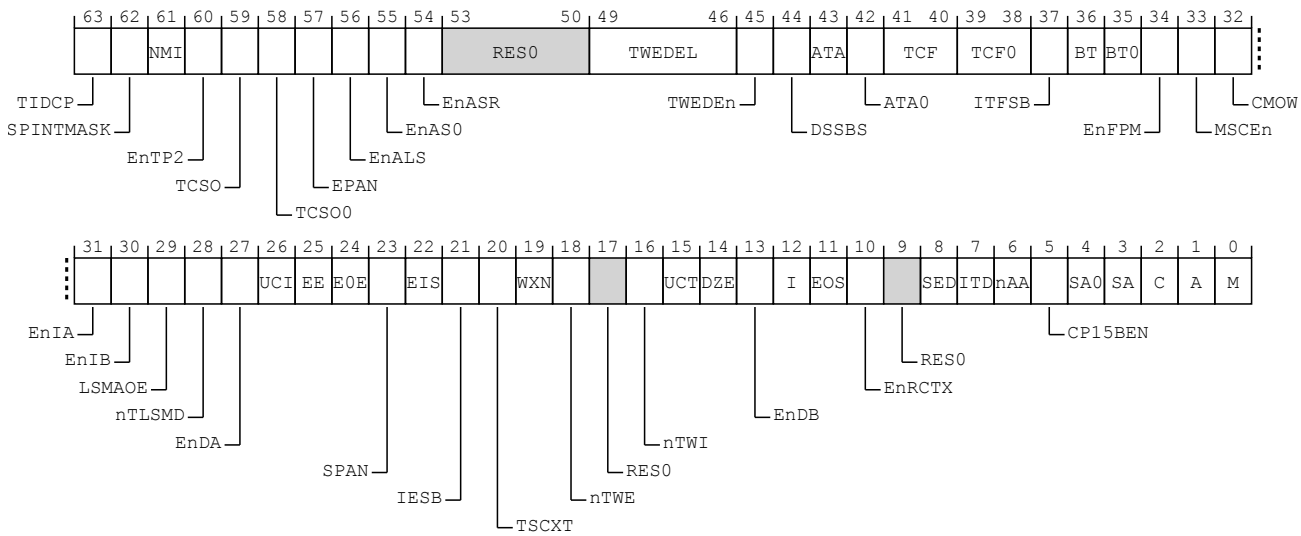

TIDCP, bit [63]

When FEAT_TIDCP1 is implemented and EffectiveHCR_EL2_E2H() == '1':

Trap IMPLEMENTATION DEFINED functionality. Traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2.

TIDCP	Meaning
0b0	No instructions accessing the System register or System instruction spaces are trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TIDCP	Meaning

0b1	If the Effective value of HCR_EL2.TGE is 0, no instructions accessing the System register or System instruction spaces are trapped by this mechanism.
	If the Effective value of HCR_EL2.TGE is 1, instructions accessing the following System register or System instruction spaces are trapped to EL2 by this mechanism:
	• In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped:
	  • IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}, and are reported using EC syndrome value 0x18.
	  • IMPLEMENTATION DEFINED System instructions, which are accessed using SYSP, with CRn == {11, 15}, and are reported using EC syndrome value 0x14.
	  • IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome value 0x18.
	  • IMPLEMENTATION DEFINED System registers, which are accessed using MRRS and MSRR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome value 0x14.
	• In AArch32 state, EL0 MCR and MRC accesses to the following encodings are trapped and reported using EC syndrome value 0x03:
	  • All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
	  • All coproc==p15, CRn==c10, opc1 ==={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
	  • All coproc==p15, CRn==c11, opc1==={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TIDCP == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL2.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL2.

SPINTMASK	Meaning
0b0	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL2.
0b1	When PSTATE.SP is 1 and execution is at EL2, an IRQ or FIQ interrupt that is targeted to EL2 is masked regardless of any indication of Superpriority. PSTATE.ALLINT is set to 0 on taking an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.SPINTMASK == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

NMI	Meaning
0b0	This control does not affect interrupt masking behavior.
0b1	This control enables all of the following: • The use of the PSTATE.ALLINT interrupt mask. • IRQ and FIQ interrupts to have Superpriority as an additional attribute. • PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.NMI == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnTP2, bit [60]

When FEAT_SME is implemented and EffectiveHCR_EL2_E2H() == '1':

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL2 when EL2 is implemented and enabled for the current Security state. The exception is reported using EC syndrome value 0x18.

EnTP2	Meaning
0b0	This control causes execution of these instructions at EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnTP2 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

TCSO, bit [59]

When FEAT_MTE_STORE_ONLY is implemented:

Tag Checking Store Only.

TCSO	Meaning
0b0	This field has no effect on Tag checking.
0b1	Explicit Memory Read Effects generated by instructions executed in EL2 are Tag Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TCSO == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCSO0, bit [58]

When FEAT_MTE_STORE_ONLY is implemented and EffectiveHCR_EL2_E2H() == '1':

Tag Checking Store Only in EL0.

TCSO0	Meaning
0b0	This field has no effect on Tag checking.
0b1	Explicit Memory Read Effects generated by instructions executed in EL0 are Tag Unchecked.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TCSO0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EPAN, bit [57]

When FEAT_PAN3 is implemented and EffectiveHCR_EL2_E2H() == '1':

Enhanced Privileged Access Never. When PSTATE.PAN is 1, determines whether an EL2 data access to a page with EL0 instruction access permission generates a Permission fault as a result of the Privileged Access Never mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPAN	Meaning
0b0	No additional Permission faults are generated by this mechanism.
0b1	An EL2 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault. Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN = 1 if the accesses were not speculative, will not cause an allocation into a cache. When executing at EL2, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this does not prevent unprivileged speculative accesses generated from the EL0 hardware-defined context from causing allocation into a cache.

Note

The value of HCR_EL2.TGE does not change the effect of this field on privileged accesses.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EPAN == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnALS, bit [56]

When FEAT_LS64 is implemented and EffectiveHCR_EL2_E2H() == '1':

Traps execution of an LD64B or ST64B instruction at EL0 to EL2.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

A trap of an LD64B or ST64B instruction is reported using EC syndrome value 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnALS == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnAS0, bit [55]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LS64_ACCDATA is implemented and EffectiveHCR_EL2_E2H() == '1':

Traps execution of an ST64BV0 instruction at EL0 to EL2.

EnAS0	Meaning
`0b0`	Execution of an ST64BV0 instruction at EL0 is trapped to EL2.
`0b1`	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

A trap of an ST64BV0 instruction is reported using EC syndrome value `0x0A`, with an ISS code of `0x0000001`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnAS0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnASR, bit [54]

When FEAT_LS64_V is implemented and EffectiveHCR_EL2_E2H() == '1':

Traps execution of an ST64BV instruction at EL0 to EL2.

EnASR	Meaning
`0b0`	Execution of an ST64BV instruction at EL0 is trapped to EL2.
`0b1`	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

A trap of an ST64BV instruction is reported using EC syndrome value `0x0A`, with an ISS code of `0x0000000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnASR == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [53:50]

Reserved, RES0.

TWEDEL, bits [49:46]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TWED is implemented and EffectiveHCR_EL2_E2H() == '1':

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL2.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCTLR_EL2.nTWE as $2^{(TWEDEL + 8)}$ cycles.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TWEDEL == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

When FEAT_TWED is implemented and EffectiveHCR_EL2_E2H() == '1':

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCTLR_EL2.nTWE.

TWEDEn	Meaning
0b0	The delay for taking a WFE* trap is IMPLEMENTATION DEFINED.
0b1	The delay for taking a WFE* trap is at least the number of cycles defined in SCTLR_EL2.TWEDEL.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TWEDEn == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on exception entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL2.
0b1	PSTATE.SSBS is set to 1 on an exception to EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.DSSBS == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access at EL2.

Controls use of Memory tagging at EL2.

ATA	Meaning
0b0	Use of Memory tagging is disabled at EL2.
0b1	Use of Memory tagging is enabled at EL2.

If the Effective value of SCR_EL3.ATA is 0, then the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.ATA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

ATA0, bit [42]

When FEAT_MTE2 is implemented and EffectiveHCR_EL2_E2H() == '1':

Allocation Tag Access at EL0.

Controls use of Memory Tagging at EL0.

ATA0	Meaning
0b0	Use of Memory tagging is disabled at EL0.
0b1	Use of Memory tagging is enabled at EL2.

If the Effective value of SCR_EL3.ATA is 0, then the Effective value of this field is 0.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.ATA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCF, bits [41:40]

When FEAT_MTE2 is implemented:

Tag Check Fault at EL2. Controls the effect of Tag Check Faults due to Loads and Stores at EL2.

TCF	Meaning	Applies when
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	FEAT_MTE_ASYM_FAULT is implemented

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TCF == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCF0, bits [39:38]

When FEAT_MTE2 is implemented and EffectiveHCR_EL2_E2H() == '1':

Tag Check Fault at EL0. Controls the effect of Tag Check Faults due to Loads and Stores at EL0.

TCF0	Meaning	Applies when
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	FEAT_MTE_ASYM_FAULT is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

If FEAT_MTE3 is not implemented, the value `0b11` is reserved.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TCF0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When FEAT_MTE_ASYNC is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL2, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0_EL1, TFSR_EL1, and TFSR_EL2 registers.

ITFSB	Meaning
`0b0`	Tag Check Faults are not synchronized on entry to EL2.
`0b1`	Tag Check Faults are synchronized on entry to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.ITFSB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

BT, bit [36]

When FEAT_BTI is implemented:

Indicates the Branch Type compatibility of the implicit BTI behavior for the following instructions at EL2:

- `PACIASP`.
- `PACIBSP`.
- If FEAT_PAuth_LR is implemented, `PACIASPPC`.
- If FEAT_PAuth_LR is implemented, `PACIBSPPC`.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit is named BT1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning
0b0	When the PE is executing at EL2, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.jc.
0b1	When the PE is executing at EL2, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.c.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.BT == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

BT0, bit [35]

When FEAT_BTI is implemented and EffectiveHCR_EL2_E2H() == '1':

Indicates the Branch Type compatibility of the implicit BTI behavior for the following instructions at EL0:

- PACIASP.
- PACIBSP.
- If FEAT_PAuth_LR is implemented, PACIASPPC.
- If FEAT_PAuth_LR is implemented, PACIBSPPC.

BT0	Meaning
0b0	When the PE is executing at EL0, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.jc.
0b1	When the PE is executing at EL0, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.c.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.BT0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnFPM, bit [34]

When FEAT_FPMR is implemented and EffectiveHCR_EL2_E2H() == '1':

Enables direct and indirect accesses to FPMR from EL0.

When accesses to FPMR are disabled by this control:

- Direct accesses to FPMR from EL0 are trapped to EL2 and reported using EC syndrome value 0x18.
- Execution of FP8 data-processing instructions that indirectly access FPMR is UNDEFINED at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnFPM	Meaning
0b0	Direct and indirect accesses to FPMR are disabled at EL0.
0b1	This control does not cause any instructions to be trapped.

Traps are not taken if there is a higher priority exception generated by the access.

If EL2 is not implemented or is disabled in the current Security state, the Effective value of this field is 1.

If the Effective value of HCR_EL2.TGE is 0, this field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnFPM == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

MSCEn, bit [33]

When FEAT_MOPS is implemented and EffectiveHCR_EL2_E2H() == '1':

Memory Copy and Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions at EL0.

MSCEn	Meaning
0b0	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

When FEAT_MOPS is implemented and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the Effective value of this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.MSCEn == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

CMOW, bit [32]

When FEAT_CMOW is implemented and EffectiveHCR_EL2_E2H() == '1':

Controls the required permissions for the following cache maintenance instructions executed at EL0:

- Any DC instruction that operates by VA and performs a clean and invalidate operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Any IC instruction that operates by VA.

CMOW	Meaning
0b0	This control does not cause any instruction to generate a stage 1 Permission fault.
0b1	For these instructions, when executed at EL0, the absence of stage 1 unprivileged write permission generates a stage 1 permission fault.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

For more information, see:

- Stage 1 permissions.
- Implications of enabling the dirty state management mechanism.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.CMOW == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIAKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnIA	Meaning
0b0	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIAKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIAKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACIA()` and `AuthIA()` pseudocode functions. When pointer authentication is enabled, `AddPACIA()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthIA()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnIA == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIBKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnIB	Meaning
0b0	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIBKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIBKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACIB()` and `AuthIB()` pseudocode functions. When pointer authentication is enabled, `AddPACIB()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthIB()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnIB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented and EffectiveHCR_EL2_E2H() == '1':

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
0b0	For all memory accesses at EL0, T32 and A32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
0b1	The ordering and interrupt behavior of T32 and A32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.LSMAOE == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented and EffectiveHCR_EL2_E2H() == '1':

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
0b0	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
0b1	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.nTLSMD == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDAKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnDA	Meaning
0b0	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDAKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDAKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACDA()` and `AuthDA()` pseudocode functions. When pointer authentication is enabled, `AddPACDA()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthDA()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnDA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

UCI, bit [26]

When EffectiveHCR_EL2_E2H() == '1':

Traps execution of cache maintenance instructions at EL0 to EL2, from AArch64 state only, reported using EC syndrome value `0x18`, as follows:

- DC CVAU, DC CIVAC, DC CVAC, and IC IVAU.
- If FEAT_MTE is implemented, DC CIGVAC, DC CIGDVAC, DC CGVAC, and DC CGDVAC.
- If FEAT_DPB is implemented, DC CVAP.
- If FEAT_DPB and FEAT_MTE are implemented, DC CGVAP and DC CGDVAP.
- If FEAT_DPB2 is implemented, DC CVADP.
- If FEAT_DPB2 and FEAT_MTE are implemented, DC CGVADP and DC CGDVADP.
- If FEAT_OCCMO is implemented, DC CIVAOC, DC CIGDVAOC, DC CVAOC and DC CGDVAOC.

UCI	Meaning
`0b0`	For each of the specified instructions, if the execution of the instruction can be trapped, access at EL0 using AArch64 is trapped to EL2.
`0b1`	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.UCI == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime.

EE	Meaning
`0b0`	Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EE	Meaning
0b1	Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EE == ‘1’, access to this field is RO.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

Reserved, RES0.

E0E, bit [24]

When FEAT_MixedEndEL0 is implemented and EffectiveHCR_EL2_E2H() == '1':

Endianness of data accesses at EL0.

E0E	Meaning
0b0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

This bit has no effect on the endianness of Explicit Memory Effects generated by unprivileged memory access instructions executed at EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.E0E == ‘1’, access to this field is RO.

When FEAT_BigEndEL0 is implemented and EffectiveHCR_EL2_E2H() == '1':

Explicit data accesses at EL0 are big-endian.

Reserved, RES1.

When FEAT_LittleEndEL0 is implemented and EffectiveHCR_EL2_E2H() == '1':

Explicit data accesses at EL0 are little-endian.

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

SPAN, bit [23]

When EffectiveHCR_EL2_E2H() == '1':

Set Privileged Access Never, on taking an exception to EL2.

SPAN	Meaning
0b0	PSTATE.PAN is set to 1 on taking an exception to EL2.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.SPAN == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

EIS, bit [22]

When FEAT_ExS is implemented:

Exception entry is a context synchronization event.

EIS	Meaning
0b0	The taking of an exception to EL2 is not a context synchronization event.
0b1	The taking of an exception to EL2 is a context synchronization event.

If SCTLR_EL2.EIS is set to 0:

- Indirect writes to ESR_EL2, FAR_EL2, SPSR_EL2, ELR_EL2, and HPFAR_EL2 are synchronized on exception entry to EL2, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.
- Some exception entries reported are not considered IFBEs per the memory model.

The following are not affected by the value of SCTLR_EL2.EIS:

- Changes to the PSTATE information on entry to EL2.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- The memory model requirement for exception entry to generate an Instruction Fetch Barrier Effect for some exception entries. See Basic definitions for the list of exception entries.
- Exit from Debug state.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EIS == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable.

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL2. • Before the operational pseudocode of each Exception Return instruction executed at EL2.

If FEAT_DoubleFault2 is implemented, the PE is in Non-debug state, and the Effective value of SCTLR2_EL2.NMEA is 1, then SCTLR_EL2.IESB is ignored and the PE behaves as if SCTLR_EL2.IESB is 1 for all purposes other than direct read of the register.

The value of this field is ignored in Debug state and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.IESB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented) and EffectiveHCR_EL2_E2H() == '1':

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

TSCXT	Meaning
0b0	EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
0b1	EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL2, and the SCXTNUM_EL0 value is treated as 0.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.TSCXT == ‘1’, access to this field is RO.

When FEAT_CSV2_2 is not implemented, FEAT_CSV2_1p2 is not implemented, and EffectiveHCR_EL2_E2H() == '1':

Reserved, RES1.

Otherwise:

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can restrict execute permissions on writable pages.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	In the EL2 or EL2&0 translation regime, any region of memory that is writable at EL2 is XN at EL2. In the EL2&0 translation regime, any region of memory that is writable at EL0 is XN at EL0.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

This field is RES0 if TCR2_EL2.PIE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.WXN == ‘1’, access to this field is RO.

nTWE, bit [18]

When EffectiveHCR_EL2_E2H() == '1':

Traps execution of WFE instructions at EL0 to EL2, from both Execution states.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

nTWE	Meaning
0b0	Any attempt to execute a WFE instruction at EL0 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.nTWE == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

When EffectiveHCR_EL2_E2H() == '1':

Traps execution of WFI instructions at EL0 to EL2, from both Execution states.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

nTWI	Meaning
0b0	Any attempt to execute a WFI instruction at EL0 is trapped EL2, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.nTWI == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

UCT, bit [15]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EffectiveHCR_EL2_E2H() == '1':

Traps EL0 accesses to the CTR_EL0 to EL2, from AArch64 state only.

UCT	Meaning
0b0	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.UCT == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DZE, bit [14]

When EffectiveHCR_EL2_E2H() == '1':

Traps execution of DC ZVA instructions at EL0 to EL2, from AArch64 state only.

If FEAT_MTE is implemented, this trap also applies to DC GVA and DC GZVA.

DZE	Meaning
0b0	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions that this trap applies to are not supported.
0b1	This control does not cause any instructions to be trapped.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.DZE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDBKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnDB	Meaning
0b0	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDBKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDBKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACDB()` and `AuthDB()` pseudocode functions. When pointer authentication is enabled, `AddPACDB()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthDB()` returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnDB == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2 and, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL0.

I	Meaning
0b0	All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, all instruction accesses to Normal memory from EL0 are Non-cacheable for all levels of instruction and unified cache. If SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Cacheability of instruction access to Normal memory from EL2 and, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, instruction access to Normal memory from EL0. If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL3 translation regime.

When EL2 is disabled in the current Security state or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.I == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EOS, bit [11]

When FEAT_ExS is implemented:

Exception exit is a context synchronization event.

EOS	Meaning
0b0	An exception return from EL2 is not a context synchronization event.
0b1	An exception return from EL2 is a context synchronization event.

If SCTLR_EL2.EOS is set to 0:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL2 and ELR_EL2 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EOS == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented and EffectiveHCR_EL2_E2H() == '1':

Enable EL0 access to the following System instructions:

- CFPRCTX, DVPRCTX and CPPRCTX instructions.
- If FEAT_SPECRES2 is implemented, COSPRCTX.
- CFP RCTX, DVP RCTX and CPP RCTX instructions.
- If FEAT_SPECRES2 is implemented, COSP RCTX.

EnRCTX	Meaning
0b0	EL0 access to these instructions is disabled, and these instructions are trapped to EL2.
0b1	EL0 access to these instructions is enabled.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.EnRCTX == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bit [9]

Reserved, RES0.

SED, bit [8]

When FEAT_AA32EL0 is implemented and EffectiveHCR_EL2_E2H() == '1':

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

SED	Meaning
0b0	SETEND instruction execution is enabled at EL0 using AArch32.
0b1	SETEND instructions are UNDEFINED at EL0 using AArch32.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.SED == '1', access to this field is RO.

When FEAT_AA32EL0 is not implemented and EffectiveHCR_EL2_E2H() == '1':

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

Access to this field is RES1.

Otherwise:

Reserved, RES0.

ITD, bit [7]

When FEAT_AA32EL0 is implemented and EffectiveHCR_EL2_E2H() == '1':

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

ITD	Meaning
0b0	All IT instruction functionality is enabled at EL0 using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITD	Meaning

`0b1`	Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED:
	• All encodings of the IT instruction with hw1[3:0]!=1000.
	• All encodings of the subsequent instruction with the following values for hw1:
	  • `0b11xxxxxxxxxxxxxx`: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
	  • `0b1011xxxxxxxxxxxx`: All instructions in 'Miscellaneous 16-bit instructions'.
	  • `0b10100xxxxxxxxxxx`: ADD Rd, PC, #imm
	  • `0b01001xxxxxxxxxxx`: LDR Rd, [PC, #imm]
	  • `0b0100x1xxx1111xxx`: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
	  • `0b010001xx1xxxx111`: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.
	These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.
	It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:
	• A 16-bit instruction, that can only be followed by another 16-bit instruction.
	• The first half of a 32-bit instruction.
	This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.
	An implementation might vary dynamically whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

If an instruction in an active IT block that would be disabled by this field sets this field to 1, then behavior is CONSTRAINED UNPREDICTABLE. For more information see 'Changes to an ITD control by an instruction in an IT block'.

ITD is optional, but if it is implemented in the SCTLR_EL2, then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When an implementation does not implement ITD, access to this field is RAZ/WI.
- Access to this field is RO if all of the following are true:
  - FEAT_SRMASK is implemented
  - EL2 is the current Exception level
  - IsSystemRegisterMaskingEnabled(EL2)
  - SCTLRMASK_EL2.ITD == '1'

When FEAT_AA32EL0 is not implemented and EffectiveHCR_EL2_E2H() == '1':

Reserved, RES1.

Otherwise:

Reserved, RES0.

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults under certain conditions at EL2, and, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following instructions generate an Alignment fault if all bytes being accessed are not within a single naturally aligned 16-byte quantity, for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.
- If FEAT_LRCPC3 is implemented, the post index versions of LDAPR and the pre index versions of STLR.
- If FEAT_LRCPC3 and Advanced SIMD and floating-point instructions are implemented, LDAPUR (SIMD&FP), LDAP1 (SIMD&FP), STLUR (SIMD&FP), and STL1 (SIMD&FP).

If FEAT_LRCPC3 is implemented, the following instructions generate an Alignment fault if all bytes being accessed for a single register are not within a single naturally aligned 16-byte quantity, for access:

- LDIAPP, STILP.

nAA	Meaning
`0b0`	Unaligned accesses by the specified instructions generate an Alignment fault.
`0b1`	Unaligned accesses by the specified instructions do not generate an Alignment fault.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register, or registers, being ZR, it is IMPLEMENTATION SPECIFIC whether this field behaves as 1 or the programmed value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.nAA == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

CP15BEN, bit [5]

When FEAT_AA32EL0 is implemented and EffectiveHCR_EL2_E2H() == '1':

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from EL0:

CP15BEN	Meaning
`0b0`	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
`0b1`	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

CP15BEN is optional, but if it is implemented in the SCTLR_EL2, then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.CP15BEN == '1', access to this field is RO.

When FEAT_AA32EL0 is not implemented and EffectiveHCR_EL2_E2H() == '1':

Access to this field is RES0.

Otherwise:

Reserved, RES1.

SA0, bit [4]

When EffectiveHCR_EL2_E2H() == '1':

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

If the Effective value of HCR_EL2.TGE is 0, the field is IGNORED for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.SA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.SA == '1', access to this field is RO.

C, bit [2]

Data access Cacheability control, for accesses at EL2 and, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning

0b0	The following are Non-cacheable for all levels of data and unified cache:
	• Data accesses to Normal memory from EL2.
	• When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, Normal memory accesses to the EL2 translation tables.
	• When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}:
	  • Data accesses to Normal memory from EL0.
	  • Normal memory accesses to the EL2&0 translation tables.

0b1	This control has no effect on the Cacheability of:
	• Data access to Normal memory from EL2.
	• When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, Normal memory accesses to the EL2 translation tables.
	• When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}:
	  • Data accesses to Normal memory from EL0.
	  • Normal memory accesses to the EL2&0 translation tables.

This bit has no effect on the EL3 translation regime.

When EL2 is disabled in the current Security state or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.C == '1', access to this field is RO.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2 and, when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL0.

A	Meaning
0b0	Alignment fault checking is disabled when executing at EL2. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, alignment fault checking disabled when executing at EL0. Alignment checks on some instructions are not disabled by this control. For more information, see 'Alignment of data accesses'.
0b1	Alignment fault checking is enabled when executing at EL2. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, alignment fault checking enabled when executing at EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.A == '1', access to this field is RO.

M, bit [0]

MMU enable for EL2 or EL2&0 stage 1 address translation.

M	Meaning
0b0	When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL2 stage 1 address translation disabled. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL2&0 stage 1 address translation disabled. See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.
0b1	When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL2 stage 1 address translation enabled. When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL2&0 stage 1 address translation enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and SCTLRMASK_EL2.M == '1', access to this field is RO.

Accessing SCTLR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `SCTLR_EL2` or `SCTLR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to SCTLR_EL2 are masked by SCTLRMASK_EL2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SCTLR_EL2();
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(t) = SCTLR_EL2();
end;
```

MSR SCTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    SCTLR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    SCTLR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x110);
    else
        X{64}(t) = SCTLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLR_EL2();
    else
        X{64}(t) = SCTLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_VHE is implemented MSR SCTLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().SCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x110) = X{64}(t);
    else
        SCTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        SCTLR_EL2() = X{64}(t);
    else
        SCTLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.176 SCTLR_EL3, System Control Register (EL3)

The SCTLR_EL3 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLR_EL3 are UNDEFINED.

Attributes

SCTLR_EL3 is a 64-bit register.

Field descriptions

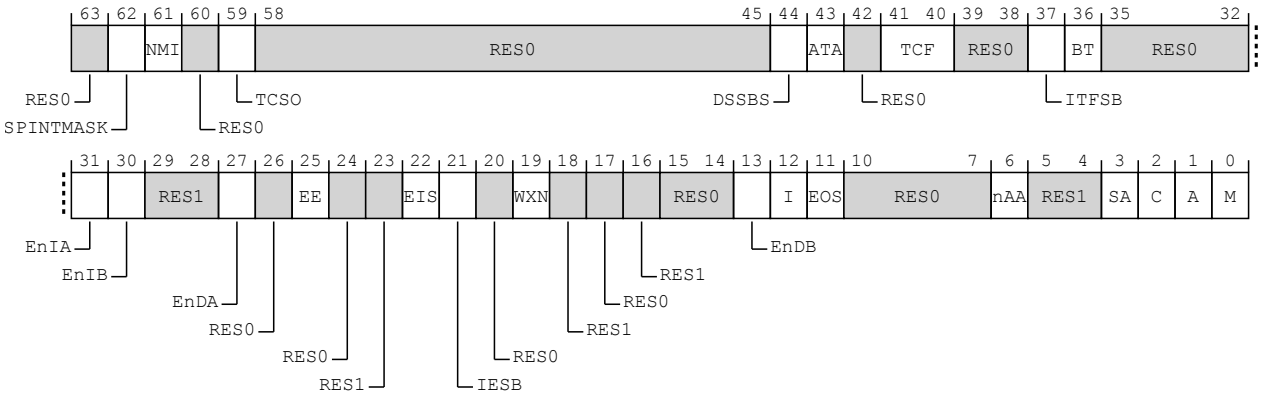

Bit [63]

Reserved, RES0.

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL3.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL3.

SPINTMASK	Meaning
0b0	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL3.
0b1	When PSTATE.SP is 1 and execution is at EL3, an IRQ or FIQ interrupt that is targeted to EL3 is masked regardless of any indication of Superpriority. PSTATE.ALLINT is set to 0 on taking an exception to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

NMI	Meaning
0b0	This control does not affect interrupt masking behavior.
0b1	This control enables all of the following: • The use of the PSTATE.ALLINT interrupt mask. • IRQ and FIQ interrupts to have Superpriority as an additional attribute. • PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bit [60]

Reserved, RES0.

TCSO, bit [59]

When FEAT_MTE_STORE_ONLY is implemented:

Tag Checking Store Only.

TCSO	Meaning
0b0	This field has no effect on Tag checking.
0b1	Explicit Memory Read Effects generated by instructions executed at EL3 are Tag Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [58:45]

Reserved, RES0.

DSSBS, bit [44]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL3.
0b1	PSTATE.SSBS is set to 1 on an exception to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access at EL3.

Controls use of Memory tagging at EL3.

ATA	Meaning
0b0	Use of Memory tagging is disabled at EL3.
0b1	Use of Memory tagging is enabled at EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

TCF, bits [41:40]

When FEAT_MTE2 is implemented:

Tag Check Fault at EL3. Controls the effect of Tag Check Faults due to Loads and Stores at EL3.

TCF	Meaning	Applies when
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCF	Meaning	Applies when
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	FEAT_MTE_ASYM_FAULT is implemented

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

ITFSB, bit [37]

When FEAT_MTE_ASYNC is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL3, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0_EL1 and TFSR_ELx registers.

ITFSB	Meaning
0b0	Tag Check Faults are not synchronized on entry to EL3.
0b1	Tag Check Faults are synchronized on entry to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT, bit [36]

When FEAT_BTI is implemented:

Indicates the Branch Type compatibility of the implicit BTI behavior for the following instructions at EL3:

- PACIASP.
- PACIBSP.
- If FEAT_PAuth_LR is implemented, PACIASPPC.
- If FEAT_PAuth_LR is implemented, PACIBSPPC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning
0b0	When the PE is executing at EL3, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.jc.
0b1	When the PE is executing at EL3, when the specified instructions have an implicit BTI behavior, they are compatible with the same BTYPE values as BTI.c.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIAKey_EL1 key, in the EL3 translation regime.

Possible values of this bit are:

EnIA	Meaning
0b0	Pointer authentication of instruction addresses, using the APIAKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses, using the APIAKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACIA()` and `AuthIA()` pseudocode functions. Specifically, when the field is 1, `AddPACIA()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthIA()` returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIBKey_EL1 key, in the EL3 translation regime.

Possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnIB	Meaning
0b0	Pointer authentication of instruction addresses, using the APIBKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses, using the APIBKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACIB()` and `AuthIB()` pseudocode functions. Specifically, when the field is 1, `AddPACIB()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthIB()` returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APDAKey_EL1 key, in the EL3 translation regime.

EnDA	Meaning
0b0	Pointer authentication of data addresses, using the APDAKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses, using the APDAKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACDA()` and `AuthDA()` pseudocode functions. Specifically, when the field is 1, `AddPACDA()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthDA()` returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Endianness of data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EE	Meaning
0b0	Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are little-endian.
0b1	Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are big-endian.

The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

Reserved, RES0.

Bit [24]

Reserved, RES0.

Bit [23]

Reserved, RES1.

EIS, bit [22]

When FEAT_ExS is implemented:

Exception Entry is a context synchronization event.

EIS	Meaning
0b0	The taking of an exception to EL3 is not a context synchronization event.
0b1	The taking of an exception to EL3 is a context synchronization event.

If SCTLR_EL3.EIS is set to 0:

- Indirect writes to ESR_EL3, FAR_EL3, SPSR_EL3, ELR_EL3 are synchronized on exception entry to EL3, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.
- Some exception entries reported are not considered IFBEs per the memory model.

The following are not affected by the value of SCTLR_EL3.EIS:

- Changes to the PSTATE information on entry to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- The memory model requirement for exception entry to generate an Instruction Fetch Barrier Effect for some exception entries. See Basic definitions for the list of exception entries.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable.

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL3. • Before the operational pseudocode of each Exception Return instruction executed at EL3.

The value of this field is ignored in Debug state and treated as zero.

When FEAT_DoubleFault is implemented, the PE is in Non-debug state, and the Effective value of SCR_EL3.NMEA is 1, this field is ignored and its Effective value is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL3 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable in the EL3 translation regime is forced to XN for accesses from software executing at EL3.

This bit applies only when SCTLR_EL3.M bit is set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The WXN bit is permitted to be cached in a TLB.

This field is RES0 if TCR_EL3.PIE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Bits [15:14]

Reserved, RES0.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APDBKey_EL1 key, in the EL3 translation regime.

EnDB	Meaning
`0b0`	Pointer authentication of data addresses, using the APDBKey_EL1 key, is not enabled.
`0b1`	Pointer authentication of data addresses, using the APDBKey_EL1 key, is enabled.

Note

This field controls the behavior of the `AddPACDB()` and `AuthDB()` pseudocode functions. Specifically, when the field is 1, `AddPACDB()` returns a copy of a pointer to which a pointer authentication code has been added, and `AuthDB()` returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL3:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I	Meaning
0b0	All instruction access to Normal memory from EL3 are Non-cacheable for all levels of instruction and unified cache. If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Cacheability of instruction access to Normal memory from EL3. If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0, EL2, or EL2&0 translation regimes.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

EOS, bit [11]

When FEAT_ExS is implemented:

Exception Exit is a context synchronization event.

EOS	Meaning
0b0	An exception return from EL3 is not a context synchronization event
0b1	An exception return from EL3 is a context synchronization event

If SCTLR_EL3.EOS is set to 0:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL3.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL3 and ELR_EL3 on exception return is synchronized.
- If the PE enters Debug state before the first instruction after an Exception return from EL3 to Non-secure state, any pending Halting debug event completes execution.
- The GIC behavior that allocates interrupts to FIQ or IRQ changes simultaneously with leaving the EL3 Exception level.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bits [10:7]

Reserved, RES0.

nAA, bit [6]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL3 under certain conditions.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single naturally aligned 16-byte quantity, for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.
- If FEAT_LRCPC3 is implemented, the post index versions of LDAPR and the pre index versions of STLR.
- If FEAT_LRCPC3 and Advanced SIMD and floating-point instructions are implemented, LDAPUR (SIMD&FP), LDAP1 (SIMD&FP), STLUR (SIMD&FP), and STL1 (SIMD&FP).

If FEAT_LRCPC3 is implemented, the following instructions generate an Alignment fault if all bytes being accessed for a single register are not within a single naturally aligned 16-byte quantity, for access:

- LDIAPP, STILP.

nAA	Meaning
`0b0`	Unaligned accesses by the specified instructions generate an Alignment fault.
`0b1`	Unaligned accesses by the specified instructions do not generate an Alignment fault.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register, or registers, being ZR, it is IMPLEMENTATION SPECIFIC whether this field behaves as 1 or the programmed value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL3 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	All data access to Normal memory from EL3, and all Normal memory accesses to the EL3 translation tables, are Non-cacheable for all levels of data and unified cache.
0b1	This control has no effect on the Cacheability of: • Data access to Normal memory from EL3. • Normal memory accesses to the EL3 translation tables.

This bit has no effect on the EL1&0, EL2, or EL2&0 translation regimes.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL3.

A	Meaning
0b0	Alignment fault checking is disabled when executing at EL3. Alignment checks on some instructions are not disabled by this control. For more information, see 'Alignment of data accesses'.
0b1	Alignment fault checking is enabled when executing at EL3. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL3 stage 1 address translation. Possible values of this bit are:

M	Meaning
0b0	EL3 stage 1 address translation disabled. See the SCTLR_EL3.I field for the behavior of instruction accesses to Normal memory.
0b1	EL3 stage 1 address translation enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

Accessing SCTLR_EL3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SCTLR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLR_EL3();
end;
```

```
MSR SCTLR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().SCTLR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        SCTLR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.177 SCTLRMASK_EL1, System Control Masking Register (EL1)

The SCTLRMASK_EL1 characteristics are:

Purpose

Mask register to prevent updates of fields in SCTLR_EL1 on writes to SCTLR_EL1 or SCTLRALIAS_EL1.

Configuration

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLRMASK_EL1 are UNDEFINED.

Attributes

SCTLRMASK_EL1 is a 64-bit register.

Field descriptions

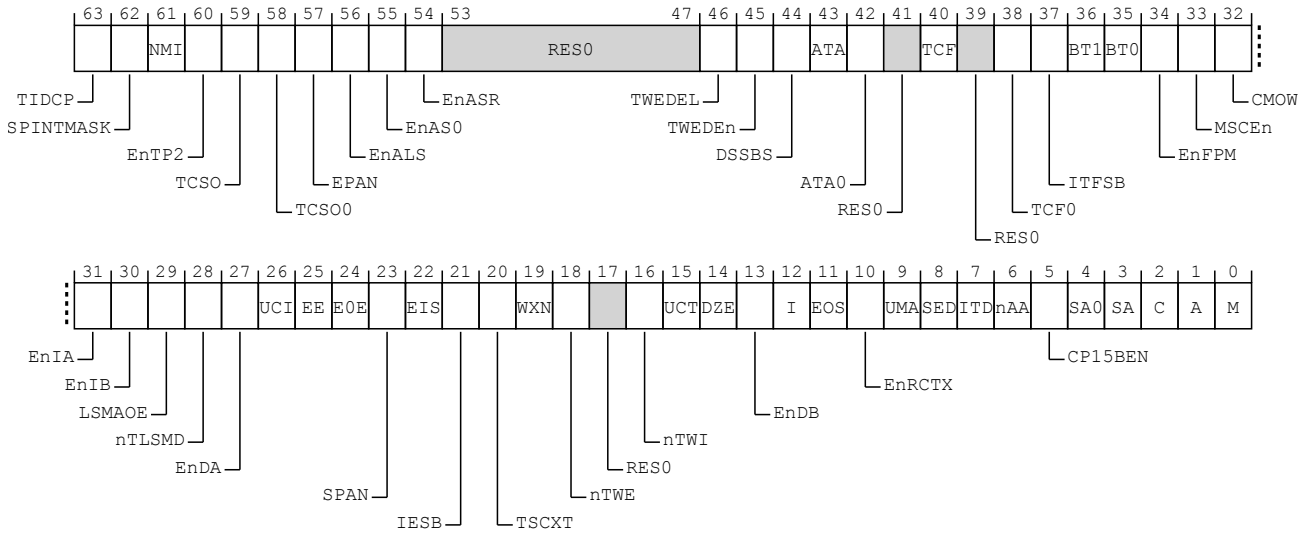

TIDCP, bit [63]

When FEAT_TIDCP1 is implemented:

Mask bit for TIDCP.

TIDCP	Meaning
0b0	SCTLR_EL1.TIDCP is writable.
0b1	SCTLR_EL1.TIDCP is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

Mask bit for SPINTMASK.

SPINTMASK	Meaning
0b0	SCTLR_EL1.SPINTMASK is writable.
0b1	SCTLR_EL1.SPINTMASK is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMI, bit [61]

When FEAT_NMI is implemented:

Mask bit for NMI.

NMI	Meaning
0b0	SCTLR_EL1.NMI is writable.
0b1	SCTLR_EL1.NMI is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnTP2, bit [60]

When FEAT_SME is implemented:

Mask bit for EnTP2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnTP2	Meaning
0b0	SCTLR_EL1.EnTP2 is writable.
0b1	SCTLR_EL1.EnTP2 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCSO, bit [59]

When FEAT_MTE_STORE_ONLY is implemented:

Mask bit for TCSO.

TCSO	Meaning
0b0	SCTLR_EL1.TCSO is writable.
0b1	SCTLR_EL1.TCSO is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCSO0, bit [58]

When FEAT_MTE_STORE_ONLY is implemented:

Mask bit for TCSO0.

TCSO0	Meaning
0b0	SCTLR_EL1.TCSO0 is writable.
0b1	SCTLR_EL1.TCSO0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPAN, bit [57]

When FEAT_PAN3 is implemented:

Mask bit for EPAN.

EPAN	Meaning
0b0	SCTLR_EL1.EPAN is writable.
0b1	SCTLR_EL1.EPAN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnALS, bit [56]

When FEAT_LS64 is implemented:

Mask bit for EnALS.

EnALS	Meaning
0b0	SCTLR_EL1.EnALS is writable.
0b1	SCTLR_EL1.EnALS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnAS0, bit [55]

When FEAT_LS64_ACCDATA is implemented:

Mask bit for EnAS0.

EnAS0	Meaning
0b0	SCTLR_EL1.EnAS0 is writable.
0b1	SCTLR_EL1.EnAS0 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnASR, bit [54]

When FEAT_LS64_V is implemented:

Mask bit for EnASR.

EnASR	Meaning
`0b0`	SCTLR_EL1.EnASR is writable.
`0b1`	SCTLR_EL1.EnASR is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [53:47]

Reserved, RES0.

TWEDEL, bit [46]

When FEAT_TWED is implemented:

Mask bit for TWEDEL.

TWEDEL	Meaning
`0b0`	SCTLR_EL1.TWEDEL is writable.
`0b1`	SCTLR_EL1.TWEDEL is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TWED is implemented:

Mask bit for TWEDEn.

TWEDEn	Meaning
0b0	SCTLR_EL1.TWEDEn is writable.
0b1	SCTLR_EL1.TWEDEn is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Mask bit for DSSBS.

DSSBS	Meaning
0b0	SCTLR_EL1.DSSBS is writable.
0b1	SCTLR_EL1.DSSBS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When FEAT_MTE2 is implemented:

Mask bit for ATA.

ATA	Meaning
0b0	SCTLR_EL1.ATA is writable.
0b1	SCTLR_EL1.ATA is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA0, bit [42]

When FEAT_MTE2 is implemented:

Mask bit for ATA0.

ATA0	Meaning
`0b0`	SCTLR_EL1.ATA0 is writable.
`0b1`	SCTLR_EL1.ATA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [41]

Reserved, RES0.

TCF, bit [40]

When FEAT_MTE2 is implemented:

Mask bit for TCF.

TCF	Meaning
`0b0`	SCTLR_EL1.TCF is writable.
`0b1`	SCTLR_EL1.TCF is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [39]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCF0, bit [38]

When FEAT_MTE2 is implemented:

Mask bit for TCF0.

TCF0	Meaning
0b0	SCTLR_EL1.TCF0 is writable.
0b1	SCTLR_EL1.TCF0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When FEAT_MTE_ASYNC is implemented:

Mask bit for ITFSB.

ITFSB	Meaning
0b0	SCTLR_EL1.ITFSB is writable.
0b1	SCTLR_EL1.ITFSB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT1, bit [36]

When FEAT_BTI is implemented:

Mask bit for BT1.

BT1	Meaning
0b0	SCTLR_EL1.BT1 is writable.
0b1	SCTLR_EL1.BT1 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT0, bit [35]

When FEAT_BTI is implemented:

Mask bit for BT0.

BT0	Meaning
`0b0`	SCTLR_EL1.BT0 is writable.
`0b1`	SCTLR_EL1.BT0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnFPM, bit [34]

When FEAT_FPMR is implemented:

Mask bit for EnFPM.

EnFPM	Meaning
`0b0`	SCTLR_EL1.EnFPM is writable.
`0b1`	SCTLR_EL1.EnFPM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MSCEn, bit [33]

When FEAT_MOPS is implemented:

Mask bit for MSCEn.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSCEn	Meaning
0b0	SCTLR_EL1.MSCEn is writable.
0b1	SCTLR_EL1.MSCEn is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CMOW, bit [32]

When FEAT_CMOW is implemented:

Mask bit for CMOW.

CMOW	Meaning
0b0	SCTLR_EL1.CMOW is writable.
0b1	SCTLR_EL1.CMOW is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIA, bit [31]

When FEAT_PAuth is implemented:

Mask bit for EnIA.

EnIA	Meaning
0b0	SCTLR_EL1.EnIA is writable.
0b1	SCTLR_EL1.EnIA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnIB, bit [30]

When FEAT_PAuth is implemented:

Mask bit for EnIB.

EnIB	Meaning
0b0	SCTLR_EL1.EnIB is writable.
0b1	SCTLR_EL1.EnIB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented:

Mask bit for LSMAOE.

LSMAOE	Meaning
0b0	SCTLR_EL1.LSMAOE is writable.
0b1	SCTLR_EL1.LSMAOE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented:

Mask bit for nTLSMD.

nTLSMD	Meaning
0b0	SCTLR_EL1.nTLSMD is writable.
0b1	SCTLR_EL1.nTLSMD is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnDA, bit [27]

When FEAT_PAuth is implemented:

Mask bit for EnDA.

EnDA	Meaning
`0b0`	SCTLR_EL1.EnDA is writable.
`0b1`	SCTLR_EL1.EnDA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Mask bit for UCI.

UCI	Meaning
`0b0`	SCTLR_EL1.UCI is writable.
`0b1`	SCTLR_EL1.UCI is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Mask bit for EE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EE	Meaning
0b0	SCTLR_EL1.EE is writable.
0b1	SCTLR_EL1.EE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0E, bit [24]

When FEAT_MixedEndEL0 is implemented:

Mask bit for E0E.

E0E	Meaning
0b0	SCTLR_EL1.E0E is writable.
0b1	SCTLR_EL1.E0E is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SPAN, bit [23]

When FEAT_PAN is implemented:

Mask bit for SPAN.

SPAN	Meaning
0b0	SCTLR_EL1.SPAN is writable.
0b1	SCTLR_EL1.SPAN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EIS, bit [22]

When FEAT_ExS is implemented:

Mask bit for EIS.

EIS	Meaning
0b0	SCTLR_EL1.EIS is writable.
0b1	SCTLR_EL1.EIS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IESB, bit [21]

When FEAT_IESB is implemented:

Mask bit for IESB.

IESB	Meaning
0b0	SCTLR_EL1.IESB is writable.
0b1	SCTLR_EL1.IESB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Mask bit for TSCXT.

TSCXT	Meaning
0b0	SCTLR_EL1.TSCXT is writable.
0b1	SCTLR_EL1.TSCXT is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WXN, bit [19]

Mask bit for WXN.

WXN	Meaning
`0b0`	SCTLR_EL1.WXN is writable.
`0b1`	SCTLR_EL1.WXN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Mask bit for nTWE.

nTWE	Meaning
`0b0`	SCTLR_EL1.nTWE is writable.
`0b1`	SCTLR_EL1.nTWE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Mask bit for nTWI.

nTWI	Meaning
`0b0`	SCTLR_EL1.nTWI is writable.
`0b1`	SCTLR_EL1.nTWI is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Mask bit for UCT.

UCT	Meaning
`0b0`	SCTLR_EL1.UCT is writable.
`0b1`	SCTLR_EL1.UCT is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Mask bit for DZE.

DZE	Meaning
`0b0`	SCTLR_EL1.DZE is writable.
`0b1`	SCTLR_EL1.DZE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Mask bit for EnDB.

EnDB	Meaning
`0b0`	SCTLR_EL1.EnDB is writable.
`0b1`	SCTLR_EL1.EnDB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

I, bit [12]

Mask bit for I.

I	Meaning
0b0	SCTLR_EL1.I is writable.
0b1	SCTLR_EL1.I is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EOS, bit [11]

When FEAT_ExS is implemented:

Mask bit for EOS.

EOS	Meaning
0b0	SCTLR_EL1.EOS is writable.
0b1	SCTLR_EL1.EOS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Mask bit for EnRCTX.

EnRCTX	Meaning
0b0	SCTLR_EL1.EnRCTX is writable.
0b1	SCTLR_EL1.EnRCTX is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UMA, bit [9]

Mask bit for UMA.

UMA	Meaning
`0b0`	SCTLR_EL1.UMA is writable.
`0b1`	SCTLR_EL1.UMA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SED, bit [8]

When FEAT_AA32EL0 is implemented:

Mask bit for SED.

SED	Meaning
`0b0`	SCTLR_EL1.SED is writable.
`0b1`	SCTLR_EL1.SED is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITD, bit [7]

When FEAT_AA32EL0 is implemented:

Mask bit for ITD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITD	Meaning
0b0	SCTLR_EL1.ITD is writable.
0b1	SCTLR_EL1.ITD is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nAA, bit [6]

When FEAT_LSE2:

Mask bit for nAA.

nAA	Meaning
0b0	SCTLR_EL1.nAA is writable.
0b1	SCTLR_EL1.nAA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CP15BEN, bit [5]

When FEAT_AA32EL0 is implemented:

Mask bit for CP15BEN.

CP15BEN	Meaning
0b0	SCTLR_EL1.CP15BEN is writable.
0b1	SCTLR_EL1.CP15BEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SA0, bit [4]

Mask bit for SA0.

SA0	Meaning
0b0	SCTLR_EL1.SA0 is writable.
0b1	SCTLR_EL1.SA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

Mask bit for SA.

SA	Meaning
0b0	SCTLR_EL1.SA is writable.
0b1	SCTLR_EL1.SA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Mask bit for C.

C	Meaning
0b0	SCTLR_EL1.C is writable.
0b1	SCTLR_EL1.C is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

A, bit [1]

Mask bit for A.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A	Meaning
0b0	SCTLR_EL1.A is writable.
0b1	SCTLR_EL1.A is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

M, bit [0]

Mask bit for M.

M	Meaning
0b0	SCTLR_EL1.M is writable.
0b1	SCTLR_EL1.M is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing SCTLRMASK_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name SCTLRMASK_EL1 or SCTLRMASK_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLRMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nSCTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x318);
    else
        X{64}(t) = SCTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SCTLRMASK_EL2();
    else
        X{64}(t) = SCTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLRMASK_EL1();
end;
```

MSR SCTLRMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nSCTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x318) = X{64}(t);
    elsif !IsZero(SCTLRMASK_EL1()) then
        Undefined();
    else
        SCTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(SCTLRMASK_EL2()) then
            Undefined();
        else
            SCTLRMASK_EL2() = X{64}(t);
        end;
    else
        SCTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLRMASK_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SCTLRMASK_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x318);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCTLRMASK_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = SCTLRMASK_EL1();
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_VHE is implemented MSR SCTLRMASK_EL12, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x318) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SCTLRMASK_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SCTLRMASK_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.178 SCTLRMASK_EL2, System Control Masking Register (EL2)

The SCTLRMASK_EL2 characteristics are:

Purpose

Mask register to prevent updates of fields in SCTLR_EL2 on writes.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SCTLRMASK_EL2 are UNDEFINED.

Attributes

SCTLRMASK_EL2 is a 64-bit register.

Field descriptions

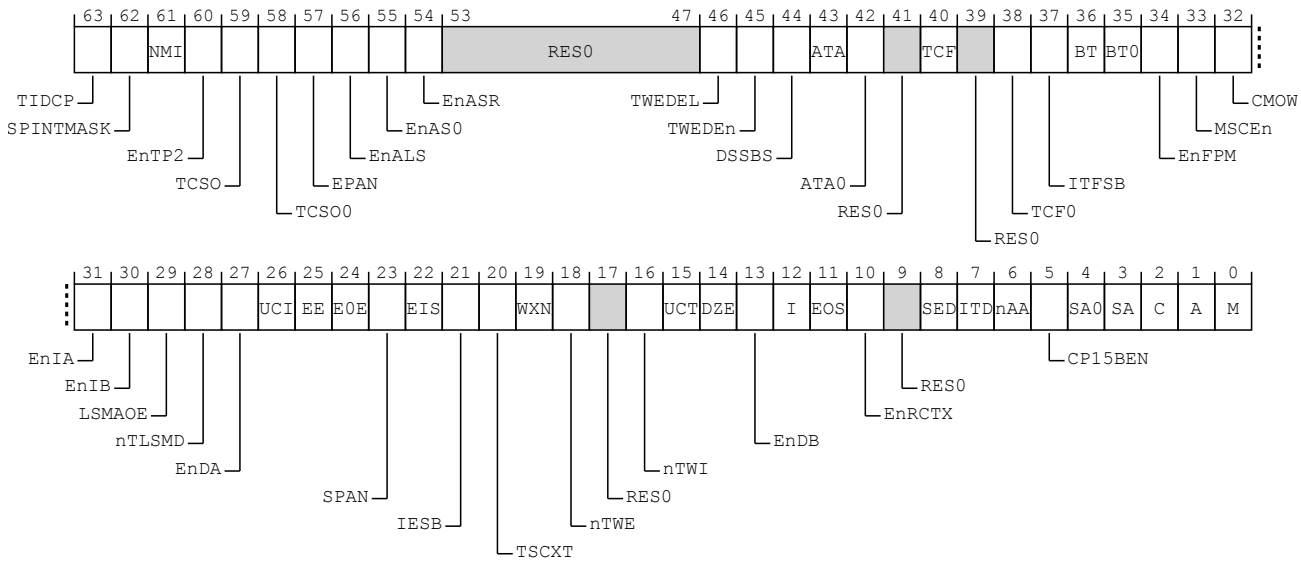

TIDCP, bit [63]

When FEAT_TIDCP1 is implemented:

Mask bit for TIDCP.

TIDCP	Meaning
0b0	SCTLR_EL2.TIDCP is writable.
0b1	SCTLR_EL2.TIDCP is not writable.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL2, this field resets to `'0'`.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

Mask bit for SPINTMASK.

SPINTMASK	Meaning
`0b0`	SCTLR_EL2.SPINTMASK is writable.
`0b1`	SCTLR_EL2.SPINTMASK is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMI, bit [61]

When FEAT_NMI is implemented:

Mask bit for NMI.

NMI	Meaning
`0b0`	SCTLR_EL2.NMI is writable.
`0b1`	SCTLR_EL2.NMI is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnTP2, bit [60]

When FEAT_SME is implemented:

Mask bit for EnTP2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnTP2	Meaning
0b0	SCTLR_EL2.EnTP2 is writable.
0b1	SCTLR_EL2.EnTP2 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCSO, bit [59]

When FEAT_MTE_STORE_ONLY is implemented:

Mask bit for TCSO.

TCSO	Meaning
0b0	SCTLR_EL2.TCSO is writable.
0b1	SCTLR_EL2.TCSO is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCSO0, bit [58]

When FEAT_MTE_STORE_ONLY is implemented:

Mask bit for TCSO0.

TCSO0	Meaning
0b0	SCTLR_EL2.TCSO0 is writable.
0b1	SCTLR_EL2.TCSO0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPAN, bit [57]

When FEAT_PAN3 is implemented:

Mask bit for EPAN.

EPAN	Meaning
0b0	SCTLR_EL2.EPAN is writable.
0b1	SCTLR_EL2.EPAN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnALS, bit [56]

When FEAT_LS64 is implemented:

Mask bit for EnALS.

EnALS	Meaning
0b0	SCTLR_EL2.EnALS is writable.
0b1	SCTLR_EL2.EnALS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnAS0, bit [55]

When FEAT_LS64_ACCDATA is implemented:

Mask bit for EnAS0.

EnAS0	Meaning
0b0	SCTLR_EL2.EnAS0 is writable.
0b1	SCTLR_EL2.EnAS0 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnASR, bit [54]

When FEAT_LS64_V is implemented:

Mask bit for EnASR.

EnASR	Meaning
`0b0`	SCTLR_EL2.EnASR is writable.
`0b1`	SCTLR_EL2.EnASR is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [53:47]

Reserved, RES0.

TWEDEL, bit [46]

When FEAT_TWED is implemented:

Mask bit for TWEDEL.

TWEDEL	Meaning
`0b0`	SCTLR_EL2.TWEDEL is writable.
`0b1`	SCTLR_EL2.TWEDEL is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TWED is implemented:

Mask bit for TWEDEn.

TWEDEn	Meaning
0b0	SCTLR_EL2.TWEDEn is writable.
0b1	SCTLR_EL2.TWEDEn is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Mask bit for DSSBS.

DSSBS	Meaning
0b0	SCTLR_EL2.DSSBS is writable.
0b1	SCTLR_EL2.DSSBS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [43]

When FEAT_MTE2 is implemented:

Mask bit for ATA.

ATA	Meaning
0b0	SCTLR_EL2.ATA is writable.
0b1	SCTLR_EL2.ATA is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA0, bit [42]

When FEAT_MTE2 is implemented:

Mask bit for ATA0.

ATA0	Meaning
`0b0`	SCTLR_EL2.ATA0 is writable.
`0b1`	SCTLR_EL2.ATA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [41]

Reserved, RES0.

TCF, bit [40]

When FEAT_MTE2 is implemented:

Mask bit for TCF.

TCF	Meaning
`0b0`	SCTLR_EL2.TCF is writable.
`0b1`	SCTLR_EL2.TCF is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [39]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCF0, bit [38]

When FEAT_MTE2 is implemented:

Mask bit for TCF0.

TCF0	Meaning
0b0	SCTLR_EL2.TCF0 is writable.
0b1	SCTLR_EL2.TCF0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When FEAT_MTE_ASYNC is implemented:

Mask bit for ITFSB.

ITFSB	Meaning
0b0	SCTLR_EL2.ITFSB is writable.
0b1	SCTLR_EL2.ITFSB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT, bit [36]

When FEAT_BTI is implemented:

Mask bit for BT.

BT	Meaning
0b0	SCTLR_EL2.BT is writable.
0b1	SCTLR_EL2.BT is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT0, bit [35]

When FEAT_BTI is implemented:

Mask bit for BT0.

BT0	Meaning
`0b0`	SCTLR_EL2.BT0 is writable.
`0b1`	SCTLR_EL2.BT0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnFPM, bit [34]

When FEAT_FPMR is implemented:

Mask bit for EnFPM.

EnFPM	Meaning
`0b0`	SCTLR_EL2.EnFPM is writable.
`0b1`	SCTLR_EL2.EnFPM is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MSCEn, bit [33]

When FEAT_MOPS is implemented:

Mask bit for MSCEn.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSCEn	Meaning
0b0	SCTLR_EL2.MSCEn is writable.
0b1	SCTLR_EL2.MSCEn is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CMOW, bit [32]

When FEAT_CMOW is implemented:

Mask bit for CMOW.

CMOW	Meaning
0b0	SCTLR_EL2.CMOW is writable.
0b1	SCTLR_EL2.CMOW is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIA, bit [31]

When FEAT_PAuth is implemented:

Mask bit for EnIA.

EnIA	Meaning
0b0	SCTLR_EL2.EnIA is writable.
0b1	SCTLR_EL2.EnIA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnIB, bit [30]

When FEAT_PAuth is implemented:

Mask bit for EnIB.

EnIB	Meaning
0b0	SCTLR_EL2.EnIB is writable.
0b1	SCTLR_EL2.EnIB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented:

Mask bit for LSMAOE.

LSMAOE	Meaning
0b0	SCTLR_EL2.LSMAOE is writable.
0b1	SCTLR_EL2.LSMAOE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented:

Mask bit for nTLSMD.

nTLSMD	Meaning
0b0	SCTLR_EL2.nTLSMD is writable.
0b1	SCTLR_EL2.nTLSMD is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnDA, bit [27]

When FEAT_PAuth is implemented:

Mask bit for EnDA.

EnDA	Meaning
`0b0`	SCTLR_EL2.EnDA is writable.
`0b1`	SCTLR_EL2.EnDA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Mask bit for UCI.

UCI	Meaning
`0b0`	SCTLR_EL2.UCI is writable.
`0b1`	SCTLR_EL2.UCI is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Mask bit for EE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EE	Meaning
0b0	SCTLR_EL2.EE is writable.
0b1	SCTLR_EL2.EE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0E, bit [24]

When FEAT_MixedEndEL0 is implemented:

Mask bit for E0E.

E0E	Meaning
0b0	SCTLR_EL2.E0E is writable.
0b1	SCTLR_EL2.E0E is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SPAN, bit [23]

Mask bit for SPAN.

SPAN	Meaning
0b0	SCTLR_EL2.SPAN is writable.
0b1	SCTLR_EL2.SPAN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EIS, bit [22]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_ExS is implemented:

Mask bit for EIS.

EIS	Meaning
0b0	SCTLR_EL2.EIS is writable.
0b1	SCTLR_EL2.EIS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IESB, bit [21]

When FEAT_IESB is implemented:

Mask bit for IESB.

IESB	Meaning
0b0	SCTLR_EL2.IESB is writable.
0b1	SCTLR_EL2.IESB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Mask bit for TSCXT.

TSCXT	Meaning
0b0	SCTLR_EL2.TSCXT is writable.
0b1	SCTLR_EL2.TSCXT is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WXN, bit [19]

Mask bit for WXN.

WXN	Meaning
`0b0`	SCTLR_EL2.WXN is writable.
`0b1`	SCTLR_EL2.WXN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Mask bit for nTWE.

nTWE	Meaning
`0b0`	SCTLR_EL2.nTWE is writable.
`0b1`	SCTLR_EL2.nTWE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Mask bit for nTWI.

nTWI	Meaning
`0b0`	SCTLR_EL2.nTWI is writable.
`0b1`	SCTLR_EL2.nTWI is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Mask bit for UCT.

UCT	Meaning
`0b0`	SCTLR_EL2.UCT is writable.
`0b1`	SCTLR_EL2.UCT is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Mask bit for DZE.

DZE	Meaning
`0b0`	SCTLR_EL2.DZE is writable.
`0b1`	SCTLR_EL2.DZE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Mask bit for EnDB.

EnDB	Meaning
`0b0`	SCTLR_EL2.EnDB is writable.
`0b1`	SCTLR_EL2.EnDB is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

I, bit [12]

Mask bit for I.

I	Meaning
0b0	SCTLR_EL2.I is writable.
0b1	SCTLR_EL2.I is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EOS, bit [11]

When FEAT_ExS is implemented:

Mask bit for EOS.

EOS	Meaning
0b0	SCTLR_EL2.EOS is writable.
0b1	SCTLR_EL2.EOS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Mask bit for EnRCTX.

EnRCTX	Meaning
0b0	SCTLR_EL2.EnRCTX is writable.
0b1	SCTLR_EL2.EnRCTX is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [9]

Reserved, RES0.

SED, bit [8]

When FEAT_AA32EL0 is implemented:

Mask bit for SED.

SED	Meaning
`0b0`	SCTLR_EL2.SED is writable.
`0b1`	SCTLR_EL2.SED is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITD, bit [7]

When FEAT_AA32EL0 is implemented:

Mask bit for ITD.

ITD	Meaning
`0b0`	SCTLR_EL2.ITD is writable.
`0b1`	SCTLR_EL2.ITD is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

nAA, bit [6]

Mask bit for nAA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nAA	Meaning
0b0	SCTLR_EL2.nAA is writable.
0b1	SCTLR_EL2.nAA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CP15BEN, bit [5]

When FEAT_AA32EL0 is implemented:

Mask bit for CP15BEN.

CP15BEN	Meaning
0b0	SCTLR_EL2.CP15BEN is writable.
0b1	SCTLR_EL2.CP15BEN is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SA0, bit [4]

Mask bit for SA0.

SA0	Meaning
0b0	SCTLR_EL2.SA0 is writable.
0b1	SCTLR_EL2.SA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

Mask bit for SA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SA	Meaning
0b0	SCTLR_EL2.SA is writable.
0b1	SCTLR_EL2.SA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Mask bit for C.

C	Meaning
0b0	SCTLR_EL2.C is writable.
0b1	SCTLR_EL2.C is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

A, bit [1]

Mask bit for A.

A	Meaning
0b0	SCTLR_EL2.A is writable.
0b1	SCTLR_EL2.A is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

M, bit [0]

Mask bit for M.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Meaning
0b0	SCTLR_EL2.M is writable.
0b1	SCTLR_EL2.M is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing SCTLRMASK_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name SCTLRMASK_EL2 or SCTLRMASK_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCTLRMASK_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SCTLRMASK_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLRMASK_EL2();
end;
```

```
MSR SCTLRMASK_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsZero(SCTLRMASK_EL2()) then
        Undefined();
    else
        SCTLRMASK_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLRMASK_EL2() = X{64}(t);
end;
```

MRS <Xt>, SCTLRMASK_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nSCTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x318);
    else
        X{64}(t) = SCTLRMASK_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SCTLRMASK_EL2();
    else
        X{64}(t) = SCTLRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCTLRMASK_EL1();
end;
```

MSR SCTLRMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nSCTLRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x318) = X{64}(t);
    elsif !IsZero(SCTLRMASK_EL1()) then
        Undefined();
    else
        SCTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(SCTLRMASK_EL2()) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            SCTLRMASK_EL2() = X{64}(t);
        end;
    else
        SCTLRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCTLRMASK_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.179 SCXTNUM_EL0, EL0 Read/Write Software Context Number

The SCXTNUM_EL0 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL0 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configuration

This register is present only when (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to SCXTNUM_EL0 are UNDEFINED.

Attributes

SCXTNUM_EL0 is a 64-bit register.

Field descriptions

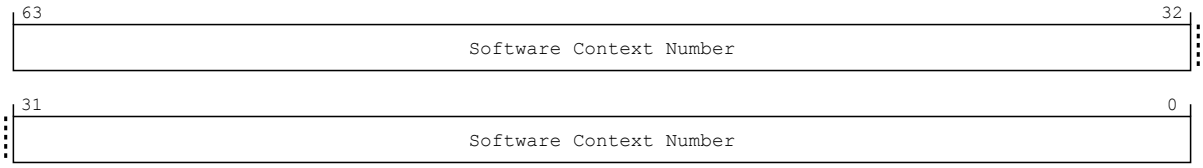

SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL0 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCXTNUM_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().TSCXT == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().EnSCXT == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
             ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().SCXTNUM_EL0 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && SCTLR_EL2().TSCXT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCXTNUM_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
            Undefined();
        elsif EL2Enabled() && HCR_EL2().EnSCXT == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HFGRTR_EL2().SCXTNUM_EL0 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCXTNUM_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCXTNUM_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = SCXTNUM_EL0();
    end;
```

MSR SCXTNUM_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().TSCXT == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().EnSCXT == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGWTR_EL2().SCXTNUM_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && SCTLR_EL2().TSCXT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SCXTNUM_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().EnSCXT == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().SCXTNUM_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SCXTNUM_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SCXTNUM_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.180 SCXTNUM_EL1, EL1 Read/Write Software Context Number

The SCXTNUM_EL1 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL1 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configuration

This register is present only when (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to SCXTNUM_EL1 are UNDEFINED.

Attributes

SCXTNUM_EL1 is a 64-bit register.

Field descriptions

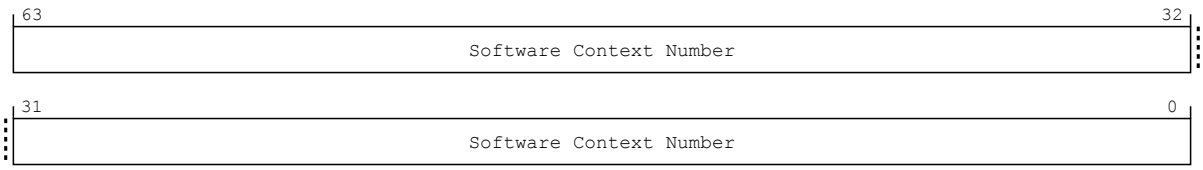

SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL1 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `SCXTNUM_EL1` or `SCXTNUM_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCXTNUM_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && HCR_EL2().EnSCXT == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HFGRTR_EL2().SCXTNUM_EL1 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x188);
        else
            X{64}(t) = SCXTNUM_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = SCXTNUM_EL2();
        else
            X{64}(t) = SCXTNUM_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = SCXTNUM_EL1();
    end;
```

MSR SCXTNUM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().EnSCXT == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().SCXTNUM_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x188) = X{64}(t);
    else
        SCXTNUM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SCXTNUM_EL2() = X{64}(t);
    else
        SCXTNUM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SCXTNUM_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x188);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SCXTNUM_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = SCXTNUM_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR SCXTNUM_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x188) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SCXTNUM_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SCXTNUM_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.181 SCXTNUM_EL2, EL2 Read/Write Software Context Number

The SCXTNUM_EL2 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL2 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to SCXTNUM_EL2 are UNDEFINED.

Attributes

SCXTNUM_EL2 is a 64-bit register.

Field descriptions

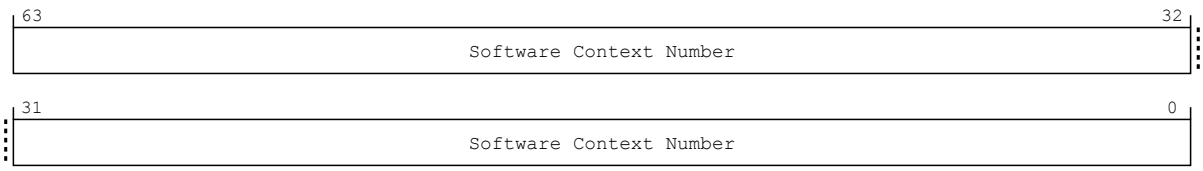

SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL2 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `SCXTNUM_EL2` or `SCXTNUM_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCXTNUM_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SCXTNUM_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCXTNUM_EL2();
end;
```

```
MSR SCXTNUM_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SCXTNUM_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, SCXTNUM_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().EnSCXT == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().SCXTNUM_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x188);
    else
        X{64}(t) = SCXTNUM_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SCXTNUM_EL2();
    else
        X{64}(t) = SCXTNUM_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCXTNUM_EL1();
end;
```

```
MSR SCXTNUM_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !((IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2().EnSCXT == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGWTR_EL2().SCXTNUM_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x188) = X{64}(t);
    else
        SCXTNUM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnSCXT == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnSCXT == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SCXTNUM_EL2() = X{64}(t);
    else
        SCXTNUM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.182 SCXTNUM_EL3, EL3 Read/Write Software Context Number

The SCXTNUM_EL3 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL3 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configuration

This register is present only when (HaveEL(EL3) && (IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to SCXTNUM_EL3 are UNDEFINED.

Attributes

SCXTNUM_EL3 is a 64-bit register.

Field descriptions

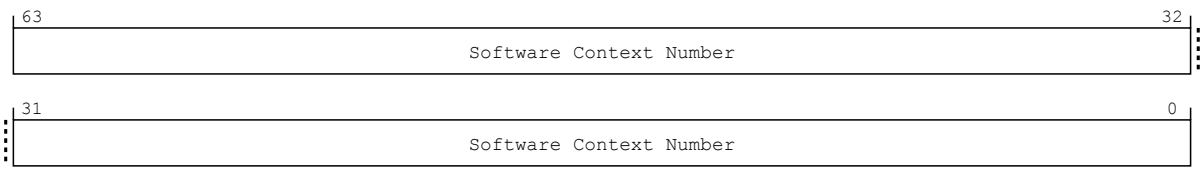

SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL3 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SCXTNUM_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b111

```
if !(HaveEL(EL3) && (IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SCXTNUM_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SCXTNUM_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b111

```
if !(HaveEL(EL3) && (IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    SCXTNUM_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.183 SMCR_EL1, SME Control Register (EL1)

The SMCR_EL1 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at Exception levels EL1 and EL0.

Configuration

This register has no effect if the PE is not in Streaming SVE mode.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this register has no effect on execution at EL0.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL1 are UNDEFINED.

Attributes

SMCR_EL1 is a 64-bit register.

Field descriptions

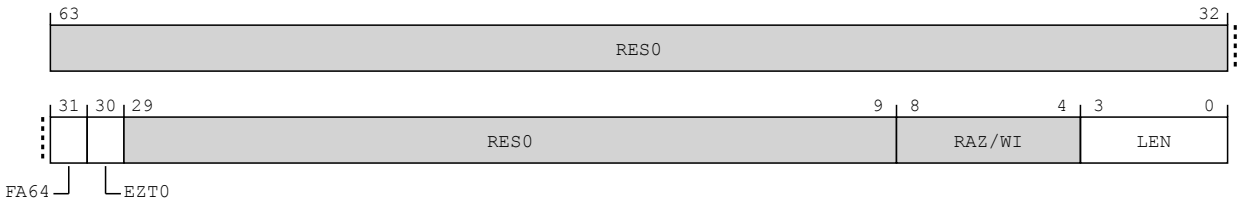

Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction at EL1 is considered legal when executed in Streaming SVE mode.

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, controls whether execution of an A64 instruction at EL0 is considered legal when executed in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal when executed in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal when executed in Streaming SVE mode at EL1 and EL0, if they are treated as legal at more privileged Exception levels in the current Security state.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at EL1 and EL0 of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

The exception is reported using ESR_EL1.EC or ESR_EL2.EC value `0x1D`, with an ISS code of `0x0000004`, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
`0b0`	This control causes execution of these instructions at EL1 and EL0 to be trapped.
`0b1`	This control does not cause execution of any instruction to be trapped.

Changes to this field only affect whether instructions that access ZT0 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTATE.ZA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL1 of (LEN+1)*128 bits. This field also defines the Effective Streaming SVE vector length at EL0 when EL2 is not implemented, or EL2 is not enabled in the current Security state, or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL1.

For all purposes other than returning the result of a direct read of SMCR_EL1, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. If EL2 is implemented and enabled in the current Security state, and the requested length is greater than the Effective length at EL2, then the Effective length at EL2 is used.
3. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL1.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `SMCR_EL1` or `SMCR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SMCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif CPACR_EL1().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x1F0);
    else
        X{64}(t) = SMCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = SMCR_EL2();
    else
        X{64}(t) = SMCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        X{64}(t) = SMCR_EL1();
    end;
end;
```

```
MSR SMCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif CPACR_EL1().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x1F0) = X{64}(t);
    else
        SMCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif ELIsInHost(EL2) then
        SMCR_EL2() = X{64}(t);
    else
        SMCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL1() = X{64}(t);
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, SMCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x1F0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
            Undefined();
        elsif CPTR_EL2().SMEN IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x1D);
            end;
        else
            X{64}(t) = SMCR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        if CPTR_EL3().ESM == '0' then
            AArch64_SystemAccessTrap(EL3, 0x1D);
        else
            X{64}(t) = SMCR_EL1();
        end;
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR SMCR_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x1F0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
            Undefined();
        elsif CPTR_EL2().SMEN IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x1D);
            end;
        else
            SMCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        if CPTR_EL3().ESM == '0' then
            AArch64_SystemAccessTrap(EL3, 0x1D);
        else
            SMCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.184 SMCR_EL2, SME Control Register (EL2)

The SMCR_EL2 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at Exception levels EL2, EL1, and EL0.

Configuration

This register has no effect if the PE is not in Streaming SVE mode, or if EL2 is not enabled in the current Security state.

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL2 are UNDEFINED.

Attributes

SMCR_EL2 is a 64-bit register.

Field descriptions

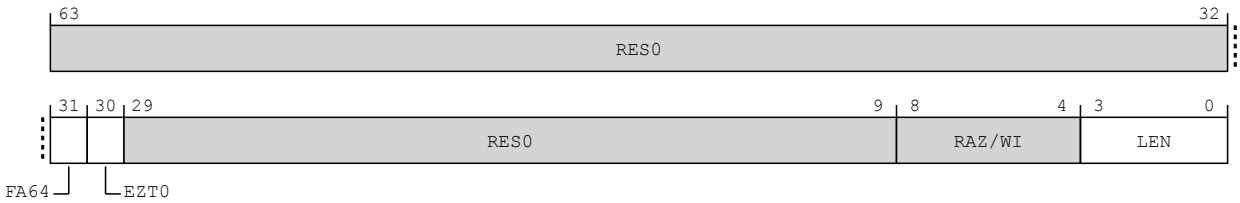

Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction at EL2, EL1, and EL0 when EL2 is implemented and enabled in the current Security state is considered legal when executed in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal when executed in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal when executed in Streaming SVE mode at EL2, EL1, and EL0, if they are treated as legal at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at EL2, EL1, and EL0 of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL2, when EL2 is enabled in the current Security state.

The exception is reported using ESR_EL2.EC value `0x1D`, with an ISS code of `0x0000004`, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
`0b0`	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
`0b1`	This control does not cause execution of any instruction to be trapped.

Changes to this field only affect whether instructions that access ZT0 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTATE.ZA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL2 of (LEN+1)*128 bits. This field also defines the Effective Streaming SVE vector length at EL0 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL2.

For all purposes other than returning the result of a direct read of SMCR_EL2, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
3. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL2.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name SMCR_EL2 or SMCR_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SMCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        X{64}(t) = SMCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        X{64}(t) = SMCR_EL2();
    end;
end;
```

```
MSR SMCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    else
        SMCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL2() = X{64}(t);
    end;
end;
```

MRS <Xt>, SMCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif CPACR_EL1().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x1F0);
    else
        X{64}(t) = SMCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SMCR_EL2();
    else
        X{64}(t) = SMCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        X{64}(t) = SMCR_EL1();
    end;
end;
```

```
MSR SMCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SME) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif CPACR_EL1().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x1F0) = X{64}(t);
    else
        SMCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TSM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif ELIsInHost(EL2) && CPTR_EL2().SMEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x1D);
        end;
    elsif ELIsInHost(EL2) then
        SMCR_EL2() = X{64}(t);
    else
        SMCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.185 SMCR_EL3, SME Control Register (EL3)

The SMCR_EL3 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at all Exception levels.

Configuration

This register has no effect if the PE is not in Streaming SVE mode.

This register is present only when FEAT_SME is implemented and EL3 is implemented. Otherwise, direct accesses to SMCR_EL3 are UNDEFINED.

Attributes

SMCR_EL3 is a 64-bit register.

Field descriptions

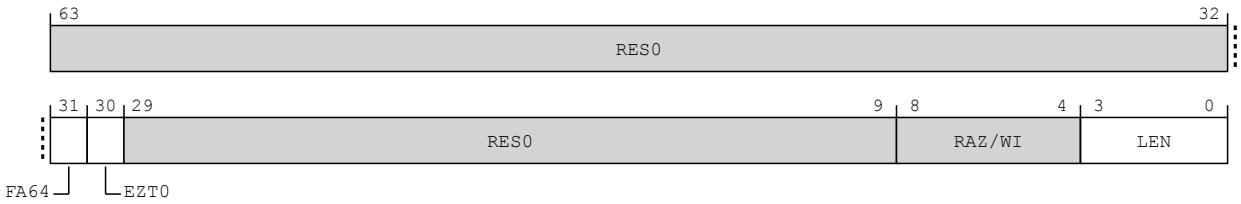

Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction is considered legal when executed in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal when executed in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal when executed in Streaming SVE mode .

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at all Exception levels of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL3.

The exception is reported using ESR_EL3.EC value `0x1D`, with an ISS code of `0x0000004`, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
`0b0`	This control causes execution of these instructions at all Exception levels to be trapped.
`0b1`	This control does not cause execution of any instruction to be trapped.

Changes to this field only affect whether instructions that access ZT0 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTATE.ZA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL3 of (LEN+1)*128 bits.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL3.

For all purposes other than returning the result of a direct read of SMCR_EL3, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SMCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b110

```
if !(IsFeatureImplemented(FEAT_SME) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        X{64}(t) = SMCR_EL3();
    end;
end;
```

```
MSR SMCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b110

```
if !(IsFeatureImplemented(FEAT_SME) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.186 SMIDR_EL1, Streaming Mode Identification Register

The SMIDR_EL1 characteristics are:

Purpose

Provides additional identification mechanisms for scheduling purposes, for a PE that supports Streaming SVE mode.

Configuration

This register is present only when FEAT_SME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SMIDR_EL1 are UNDEFINED.

Attributes

SMIDR_EL1 is a 64-bit register.

Field descriptions

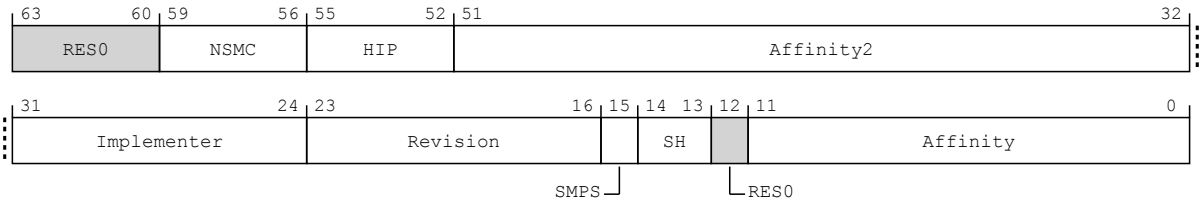

Bits [63:60]

Reserved, RES0.

NSMC, bits [59:56]

If SMIDR_EL1.{SH,Affinity} indicates that the implementation of Streaming SVE mode is shared, then this field identifies the number of SMCUs, minus 1, associated with the concatenated SMIDR_EL1.{Affinity2,Affinity} 32-bit value.

If SMIDR_EL1.{SH,Affinity} indicates that the implementation of Streaming SVE mode is not shared, then this field is zero and should be ignored by software.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NSMC	Meaning
0b0000	The implementation of Streaming SVE mode associated with this PE is not shared or is a single SMCU.
0b0001..0b1110	The number of SMCUs in the group of SMCUs providing the implementation of Streaming SVE mode for this PE, minus 1.
0b1111	Reserved.

Access to this field is RO.

HIP, bits [55:52]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SME2p2 is implemented and SMIDR_EL1.SMPS == '1':

Highest Implemented Priority. If Streaming SVE mode execution priority is supported, this field indicates the range of priority levels implemented by the PE, and the Highest Implemented Priority value.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HIP	Meaning
`0b0000`	All streaming execution priorities from 0 to 15 are implemented. The Highest Implemented Priority value is 15.
`0b0001..0b1111`	All streaming execution priorities less than or equal to this value are implemented. The Highest Implemented Priority value is the value of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Affinity2, bits [51:32]

The most significant 20 bits of the SMCU affinity for this PE, to be used in conjunction with SMIDR_EL1.Affinity.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
`0x00`	Reserved for software use.
`0x41`	Arm Limited.
`0x42`	Broadcom Corporation.
`0x43`	Cavium Inc.
`0x44`	Digital Equipment Corporation.
`0x46`	Fujitsu Ltd.
`0x49`	Infineon Technologies AG.
`0x4D`	Motorola or Freescale Semiconductor Inc.
`0x4E`	NVIDIA Corporation.
`0x50`	Applied Micro Circuits Corporation.
`0x51`	Qualcomm Inc.
`0x56`	Marvell International Ltd.
`0x69`	Intel Corporation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Implementer	Meaning
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

It is not required that this value is the same as the value of MIDR_EL1.Implementer.

Access to this field is RO.

Revision, bits [23:16]

Revision number for the Streaming Mode Compute Unit (SMCU).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SMPS, bit [15]

Indicates support for Streaming SVE mode execution priority.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SMPS	Meaning
0b0	Priority control not supported.
0b1	Priority control supported.

Access to this field is RO.

SH, bits [14:13]

Indicates whether the implementation of Streaming SVE mode in this PE is shared with other PEs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SH	Meaning
0b00	Refer to SMIDR_EL1.Affinity.
0b01	Reserved.
0b10	The implementation of Streaming SVE mode is not shared with other PEs.
0b11	The implementation of Streaming SVE mode is shared with other PEs.

Access to this field is RO.

Bit [12]

Reserved, RES0.

Affinity, bits [11:0]

The least significant 12 bits of the SMCU affinity for this PE.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the implementation of Streaming SVE mode is shared, then the concatenated SMIDR_EL1.{Affinity2,Affinity} 32-bit value identifies which shared SMCUs are associated with this PE. Every PE that shares the same SMCUs has the same 32-bit affinity value. The 32-bit affinity value is unique within the system as a whole.

The SMIDR_EL1.SH field indicates whether the implementation of Streaming SVE mode is shared with other PEs. However, if SMIDR_EL1.SH is zero, then SMIDR_EL1.Affinity is used to indicate whether the implementation of Streaming SVE is shared, as follows:

- If SMIDR_EL1.Affinity is zero, then the implementation of Streaming SVE mode is not shared with other PEs.
- If SMIDR_EL1.Affinity is not zero, then the implementation of Streaming SVE mode is shared with other PEs.

Access to this field is RO.

Accessing SMIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SMIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TID1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = SMIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = SMIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SMIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.187 SMPRI_EL1, Streaming Mode Priority Register

The SMPRI_EL1 characteristics are:

Purpose

Configures the streaming execution priority for instructions executed on a shared Streaming Mode Compute Unit (SMCU) when the PE is in Streaming SVE mode at any Exception level.

Configuration

When SMIDR_EL1.SMPS is ‘0’, this register is RES0.

This register is present only when FEAT_SME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SMPRI_EL1 are UNDEFINED.

Attributes

SMPRI_EL1 is a 64-bit register.

Field descriptions

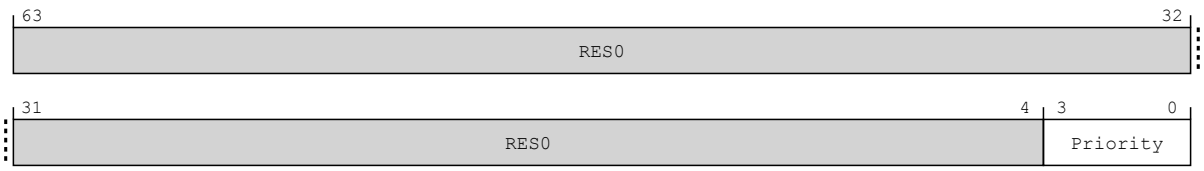

Bits [63:4]

Reserved, RES0.

Priority, bits [3:0]

Streaming execution priority value.

Either this value is used directly, or it is mapped into an effective priority value using SMPRIMAP_EL2.

This value is used directly when any of the following are true:

- The current Exception level is EL3 or EL2.
- The current Exception level is EL1 or EL0, if EL2 is implemented and enabled in the current Security state and HCRX_EL2.SMPME is ‘0’.
- The current Exception level is EL1 or EL0, if EL2 is either not implemented or not enabled in the current Security state.

The precise meaning and behavior of each streaming execution priority value is IMPLEMENTATION DEFINED.

In an implementation that shares execution resources between PEs, higher priority values are allocated more processing resource than other PEs configured with lower priority values in the same Priority domain.

If FEAT_SME2p2 is implemented, and this field is greater than the Highest Implemented Priority value indicated by SMIDR_EL1.HIP, then the PE uses the Highest Implemented Priority value as the priority.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMPRI_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SMPRI_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nSMPRI_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SMPRI_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SMPRI_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{64}(t) = SMPRI_EL1();
    end;
end;
```

```
MSR SMPRI_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nSMPRI_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SMPRI_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SMPRI_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        SMPRI_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.188 SMPRIMAP_EL2, Streaming Mode Priority Mapping Register

The SMPRIMAP_EL2 characteristics are:

Purpose

Maps the value in SMPRI_EL1 to a streaming execution priority value for instructions executed at EL1 and EL0 in the same Security states as EL2.

Configuration

When SMIDR_EL1.SMPS is ‘0’, this register is RES0.

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_SME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SMPRIMAP_EL2 are UNDEFINED.

Attributes

SMPRIMAP_EL2 is a 64-bit register.

Field descriptions

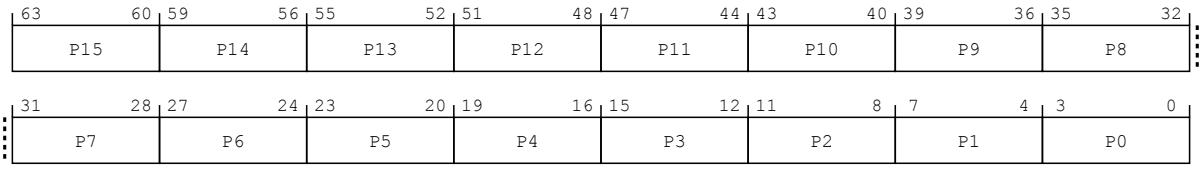

When all of the following are true, the value in SMPRI_EL1 is mapped to a streaming execution priority using this register:

- The current Exception level is EL1 or EL0.
- EL2 is implemented and enabled in the current Security state.
- HCRX_EL2.SMPME is ‘1’.

Otherwise, SMPRI_EL1 holds the streaming execution priority value.

P15, bits [63:60]

Priority Mapping Entry 15. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘15’.

This value is the highest streaming execution priority.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P14, bits [59:56]

Priority Mapping Entry 14. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘14’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P13, bits [55:52]

Priority Mapping Entry 13. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘13’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P12, bits [51:48]

Priority Mapping Entry 12. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘12’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P11, bits [47:44]

Priority Mapping Entry 11. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘11’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P10, bits [43:40]

Priority Mapping Entry 10. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘10’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P9, bits [39:36]

Priority Mapping Entry 9. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘9’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P8, bits [35:32]

Priority Mapping Entry 8. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘8’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P7, bits [31:28]

Priority Mapping Entry 7. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘7’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P6, bits [27:24]

Priority Mapping Entry 6. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘6’.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P5, bits [23:20]

Priority Mapping Entry 5. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '5'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P4, bits [19:16]

Priority Mapping Entry 4. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '4'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P3, bits [15:12]

Priority Mapping Entry 3. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '3'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P2, bits [11:8]

Priority Mapping Entry 2. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '2'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P1, bits [7:4]

Priority Mapping Entry 1. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '1'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P0, bits [3:0]

Priority Mapping Entry 0. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '0'.

This value is the lowest streaming execution priority.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMPRIMAP_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SMPRIMAP_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1F8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SMPRIMAP_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{64}(t) = SMPRIMAP_EL2();
    end;
end;
```

MSR SMPRIMAP_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1F8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().ESM == '0' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().ESM == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SMPRIMAP_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().ESM == '0' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        SMPRIMAP_EL2() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.189 TCR2_EL1, Extended Translation Control Register (EL1)

The TCR2_EL1 characteristics are:

Purpose

The control register for stage 1 of the EL1&0 translation regime.

Configuration

This register is present only when FEAT_TCR2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCR2_EL1 are UNDEFINED.

Attributes

TCR2_EL1 is a 64-bit register.

Field descriptions

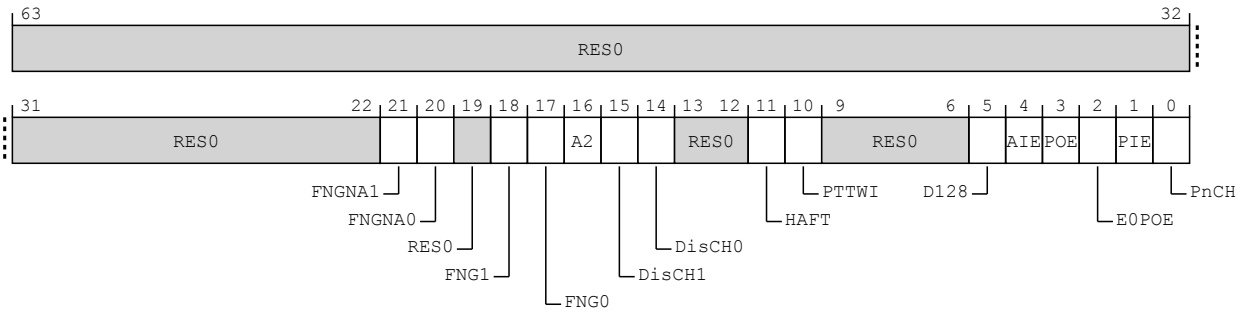

Unless stated otherwise, all the bits in TCR2_EL1 are permitted to be cached in a TLB.

Bits [63:22]

Reserved, RES0.

FNGNA1, bit [21]

When FEAT_THE is implemented:

Force non-global for unassured translations using TTBR1_EL1.

FNGNA1	Meaning
0b0	This bit has no effect on the interpretation of the nG bit.
0b1	Translations using TTBR1_EL1 are treated as non-global regardless of the value of the nG bit if all of the following is true: • The translation is for the EL1&0 translation regime. • Stage 1 and stage 2 translation are enabled. • Protection is enabled. • The final stage 1 translation using the descriptor does not have the Assured Translation property.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This bit is permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.FNGNA1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

FNGNA0, bit [20]

When FEAT_THE is implemented:

Force non-global for unassured translations using TTBR0_EL1.

FNGNA0	Meaning
0b0	This bit has no effect on the interpretation of the nG bit.
0b1	Translations using TTBR0_EL1 are treated as non-global regardless of the value of the nG bit if all of the following is true: • The translation is for the EL1&0 translation regime. • Stage 1 and stage 2 translation are enabled. • Protection is enabled. • The final stage 1 translation using the descriptor does not have the Assured Translation property.

This bit is permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.FNGNA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bit [19]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FNG1, bit [18]

When FEAT_ASID2 is implemented:

Force non-global translations for TTBR1_EL1.

FNG1	Meaning
0b0	This bit has no effect on the interpretation of the nG bit.
0b1	Translations using TTBR1_EL1 are treated as non-global regardless of the value of the nG bit.

This bit is permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.FNG1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

FNG0, bit [17]

When FEAT_ASID2 is implemented:

Force non-global translations for TTBR0_EL1.

FNG0	Meaning
0b0	This bit has no effect on the interpretation of the nG bit.
0b1	Translations using TTBR0_EL1 are treated as non-global regardless of the value of the nG bit.

This bit is permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.FNG0 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

A2, bit [16]

When FEAT_ASID2 is implemented:

Enable use of two ASIDs.

A2	Meaning
0b0	Use of two ASIDs is disabled.
0b1	Use of two ASIDs is enabled.

This bit is permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.A2 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DisCH1, bit [15]

When FEAT_D128 is implemented and TCR2_EL1.D128 == '1':

Disable the Contiguous bit for the Start Table for TTBR1_EL1.

DisCH1	Meaning
0b0	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR1_EL1 is not affected by this field.
0b1	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR1_EL1 is treated as 0.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.DisCH1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DisCH0, bit [14]

When FEAT_D128 is implemented and TCR2_EL1.D128 == '1':

Disable the Contiguous bit for the Start Table for TTBR0_EL1.

DisCH0	Meaning
0b0	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR0_EL1 is not affected by this field.
0b1	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR0_EL1 is treated as 0.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.DisCH0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [13:12]

Reserved, RES0.

HAFT, bit [11]

When FEAT_HAFT is implemented:

Hardware managed Access Flag for Table descriptors.

Enables the Hardware managed Access Flag for Table descriptors.

HAFT	Meaning
0b0	Hardware managed Access Flag for Table descriptors is disabled.
0b1	Hardware managed Access Flag for Table descriptors is enabled.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.HAFT == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

PTTWI, bit [10]

When FEAT_THE is implemented:

Permit Translation table walk Incoherence.

Permits RCWS instructions to generate writes that have the Reduced Coherence property.

PTTWI	Meaning
`0b0`	Write accesses generated by RCWS at EL1&0 do not have the Reduced Coherence property.
`0b1`	Write accesses generated by RCWS at EL1&0 have the Reduced Coherence property if HCRX_EL2.PTTWI is 1.

This bit is permitted to be implemented as a read-only bit with a fixed value of 0.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.PTTWI == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [9:6]

Reserved, RES0.

D128, bit [5]

When FEAT_D128 is implemented:

Enables VMSAv9-128 translation system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D128	Meaning
0b0	Translation system follows VMSAv8-64 translation process.
0b1	Translation system follows VMSAv9-128 translation process.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.D128 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

AIE, bit [4]

When FEAT_AIE is implemented:

Enable Attribute Indexing Extension.

AIE	Meaning
0b0	Attribute Indexing Extension Disabled.
0b1	Attribute Indexing Extension Enabled.

This field is RES1 when TCR2_EL1.D128 is 1.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.AIE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

POE, bit [3]

When FEAT_S1POE is implemented:

Enables Permission Overlays for privileged accesses from EL1&0 translation regime.

POE	Meaning
0b0	Permission overlay disabled for EL1 access in stage 1 of EL1&0 translation regime.
0b1	Permission overlay enabled for EL1 access in stage 1 of EL1&0 translation regime.

This bit is not permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.POE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

E0POE, bit [2]

When FEAT_S1POE is implemented:

Enables Permission Overlays for unprivileged accesses from EL1&0 translation regime.

E0POE	Meaning
0b0	Permission overlay disabled for EL0 access in stage 1 of EL1&0 translation regime.
0b1	Permission overlay enabled for EL0 access in stage 1 of EL1&0 translation regime.

This bit is not permitted to be cached in a TLB.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.E0POE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

PIE, bit [1]

When FEAT_S1PIE is implemented:

Enables usage of Indirect Permission Scheme.

PIE	Meaning
0b0	Direct permission model.
0b1	Indirect permission model.

This field is RES1 when TCR2_EL1.D128 is 1.

This field is ignored by the PE and treated as zero when any of the following are true:

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.PIE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

PnCH, bit [0]

When FEAT_THE is implemented:

Protected attribute enable. Enables use of bit[52] of the stage 1 translation table entries as the Protected bit, for translations using TTBRn_EL1.

PnCH	Meaning
0b0	For translations using TTBRn_EL1, bit[52] of each stage 1 translation table entry is not the Protected bit.
0b1	For translations using TTBRn_EL1, bit[52] of each stage 1 translation table entry is the Protected bit.

If bit[52] is used as the Protected bit, it is not used as the Contiguous bit.

This field is RES0 when TCR2_EL1.D128 is 1.

This field is ignored by the PE and treated as zero when any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All of the following are true:
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCRX_EL2.TCR2En is 0.
- EL3 is implemented and SCR_EL3.TCR2En == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCR2MASK_EL1.PnCH == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TCR2_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TCR2_EL1` or `TCR2_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to TCR2_EL1 are masked by TCR2MASK_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x270);
    else
        X{64}(t) = TCR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TCR2_EL2();
    else
        X{64}(t) = TCR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR2_EL1();
end;
```

```
MSR TCR2_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x270) = X{64}(t);
    else
        TCR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TCR2_EL2() = X{64}(t);
    else
        TCR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    TCR2_EL1() = X{64}(t);
end;
```

MRS <Xt>, TCR2_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x270);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TCR2_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCR2_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR TCR2_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EffectiveHCR_EL2_NVx() == '101' then
            NVMem(0x270) = X{64}(t);
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
                Undefined();
            elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                TCR2_EL1() = X{64}(t);
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            TCR2_EL1() = X{64}(t);
        else
            Undefined();
        end;
    end;
```

When FEAT_SRMASK is implemented MRS <Xt>, TCR2ALIAS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b111

```
    if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
            Undefined();
        elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
             ↪'0') || HFGRTR2_EL2().nTCR2ALIAS_EL1 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x270);
        else
            X{64}(t) = TCR2_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = TCR2_EL2();
        else
            X{64}(t) = TCR2_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = TCR2_EL1();
    end;
```

When FEAT_SRMASK is implemented MSR TCR2ALIAS_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b111

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
         ↪'0') || HFGWTR2_EL2().nTCR2ALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x270) = X{64}(t);
    else
        TCR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TCR2_EL2() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        TCR2_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR2_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.190 TCR2_EL2, Extended Translation Control Register (EL2)

The TCR2_EL2 characteristics are:

Purpose

The control register for stage 1 of the EL2&0 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_TCR2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCR2_EL2 are UNDEFINED.

Attributes

TCR2_EL2 is a 64-bit register.

Field descriptions

When EffectiveHCR_EL2_E2H() == '0':

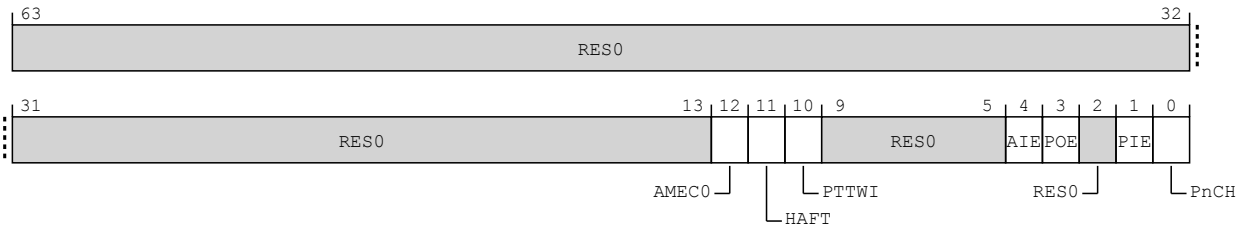

Unless stated otherwise, all the bits in TCR2_EL2 are permitted to be cached in a TLB.

Bits [63:13]

Reserved, RES0.

AMEC0, bit [12]

When FEAT_MEC is implemented:

This field controls the enabling of the Alternate MECID translations for the EL2 translation regime.

TCR2_EL2.AMEC0 is provided to enable the safe update of MECID_A0_EL2, by disabling access and speculation to AMEC==1 Block or Page descriptors during the update.

AMEC0	Meaning
0b0	Use of a Block or Page descriptor containing AMEC == 1 generates a Translation fault.
0b1	Accesses translated by a Block or Page descriptor containing AMEC == 1 are associated with the MECID configured in MECID_A0_EL2.

This bit is permitted to be cached in a TLB only if it is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

When SCTLR2_EL2.EMEC is 0, this field is ignored by the PE and the bit position of AMEC is RES0 in Block and Page descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When IsCurrentSecurityState(SS_Secure), access to this field is RES0.
- When IsCurrentSecurityState(SS_NonSecure), access to this field is RES0.

Otherwise:

Reserved, RES0.

HAFT, bit [11]

When FEAT_HAFT is implemented:

Hardware managed Access Flag for Table descriptors.

Enables the Hardware managed Access Flag for Table descriptors.

HAFT	Meaning
`0b0`	Hardware managed Access Flag for Table descriptors is disabled.
`0b1`	Hardware managed Access Flag for Table descriptors is enabled.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PTTWI, bit [10]

When FEAT_THE is implemented:

Permit Translation table walk Incoherence.

Permits RCWS instructions to generate writes that have the Reduced Coherence property.

PTTWI	Meaning
`0b0`	Write accesses generated by RCWS at EL2 or EL2&0 do not have the Reduced Coherence property.
`0b1`	Write accesses generated by RCWS at EL2 or EL2&0 have the Reduced Coherence property.

This bit is permitted to be implemented as a read-only bit with a fixed value of 0.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [9:5]

Reserved, RES0.

AIE, bit [4]

When FEAT_AIE is implemented:

Enable Attribute Indexing Extension.

AIE	Meaning
`0b0`	Attribute Indexing Extension Disabled.
`0b1`	Attribute Indexing Extension Enabled.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

POE, bit [3]

When FEAT_S1POE is implemented:

Enables Permission Overlay for EL2 accesses.

POE	Meaning
`0b0`	Permission overlay disabled for EL2 access in stage 1 of EL2 translation regime.
`0b1`	Permission overlay enabled for EL2 access in stage 1 of EL2 translation regime.

This bit is not permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [2]

Reserved, RES0.

PIE, bit [1]

When FEAT_S1PIE is implemented:

Enables usage of Indirect Permission Scheme.

PIE	Meaning
0b0	Direct permission model.
0b1	Indirect permission model.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PnCH, bit [0]

When FEAT_THE is implemented:

Protected attribute enable. Enables use of bit[52] of stage 1 translation table entries as the Protected bit, for translations using TTBR0_EL2 when HCR_EL2.E2H is 0.

PnCH	Meaning
0b0	For translations using TTBRn_EL2, bit[52] of each stage 1 translation table entry is not the Protected bit.
0b1	For translations using TTBR0_EL2, bit[52] of each stage 1 translation table entry is the Protected bit.

If bit[52] is used as the Protected bit, it is not used as the Contiguous bit.

This field is RES0 when TCR2_EL2.D128 is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EffectiveHCR_EL2_E2H() == '1':

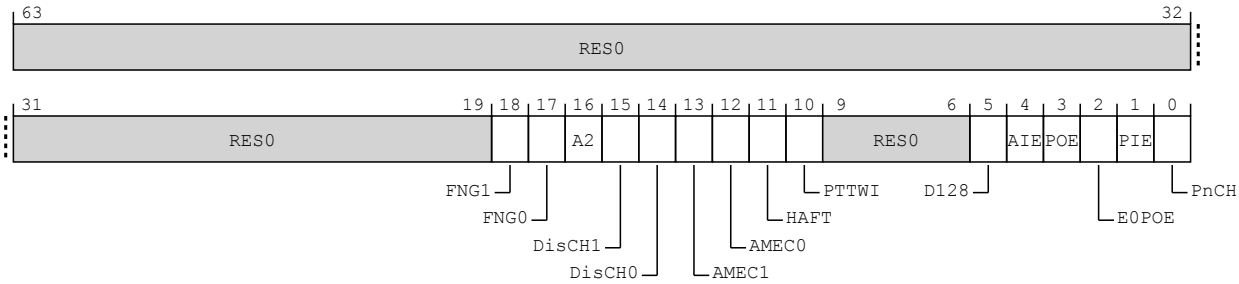

Unless stated otherwise, all the bits in TCR2_EL2 are permitted to be cached in a TLB.

Bits [63:19]

Reserved, RES0.

FNG1, bit [18]

When FEAT_ASID2 is implemented:

Force non-global translations for TTBR1_EL2.

FNG1	Meaning
0b0	This bit has no effect on the interpretation of the nG bit.
0b1	Translations are treated as non-global regardless of the value of the nG bit.

This bit is permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.FNG1 == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

FNG0, bit [17]

When FEAT_ASID2 is implemented:

Force non-global translations for TTBR0_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FNG0	Meaning
0b0	This bit has no effect on the interpretation of the nG bit.
0b1	Translations are treated as non-global regardless of the value of the nG bit.

This bit is permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.FNG0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

A2, bit [16]

When FEAT_ASID2 is implemented:

Enable use of two ASIDs.

A2	Meaning
0b0	Use of two ASIDs is disabled.
0b1	Use of two ASIDs is enabled.

This bit is permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.A2 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DisCH1, bit [15]

When FEAT_D128 is implemented and TCR2_EL2.D128 == '1':

Disable the Contiguous bit for the Start Table for TTBR1_EL2.

DisCH1	Meaning
0b0	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR1_EL2 is not affected by this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DisCH1	Meaning
0b1	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR1_EL2 is treated as 0.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.DisCH1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DisCH0, bit [14]

When FEAT_D128 is implemented and TCR2_EL2.D128 == '1':

Disable the Contiguous bit for the Start Table for TTBR0_EL2.

DisCH0	Meaning
0b0	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR0_EL2 is not affected by this field.
0b1	The Contiguous bit of Block or Page descriptors of the Start Table for TTBR0_EL2 is treated as 0.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.DisCH0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

AMEC1, bit [13]

When FEAT_MEC is implemented:

This field controls the enabling of the Alternate MECID translations for accesses in the TTBR1_EL2 half of the VA range, for the EL2&0 translation regime.

TCR2_EL2.AMEC1 is provided to enable the safe update of MECID_A1_EL2, by disabling access and speculation to AMEC == 1 Block or Page descriptors during the update.

AMEC1	Meaning
0b0	Use of a Block or Page descriptor containing AMEC == 1 generates a Translation fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMEC1	Meaning
0b1	Accesses translated by a Block or Page descriptor containing AMEC == 1 are associated with the MECID configured in MECID_A1_EL2.

This bit is permitted to be cached in a TLB only if it is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

When SCTLR2_EL2.EMEC is 0, this field is ignored by the PE and the bit position of AMEC is RES0 in Block and Page descriptors.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When IsCurrentSecurityState(SS_Secure), access to this field is RES0.
- When IsCurrentSecurityState(SS_NonSecure), access to this field is RES0.
- Access to this field is RO if all of the following are true:
  - FEAT_SRMASK is implemented
  - EL2 is the current Exception level
  - IsSystemRegisterMaskingEnabled(EL2)
  - TCR2MASK_EL2.AMEC1 == '1'

Otherwise:

Reserved, RES0.

AMEC0, bit [12]

When FEAT_MEC is implemented:

This field controls the enabling of the Alternate MECID translations for accesses in the TTBR0_EL2 half of the VA range, for the EL2&0 translation regime.

TCR2_EL2.AMEC0 is provided to enable the safe update of MECID_A0_EL2, by disabling access and speculation to AMEC==1 Block or Page descriptors during the update.

AMEC0	Meaning
0b0	Use of a Block or Page descriptor containing AMEC == 1 generates a Translation fault.
0b1	Accesses translated by a Block or Page descriptor containing AMEC == 1 are associated with the MECID configured in MECID_A0_EL2.

This bit is permitted to be cached in a TLB only if it is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

When SCTLR2_EL2.EMEC is 0, this field is ignored by the PE and the bit position of AMEC is RES0 in Block and Page descriptors.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When IsCurrentSecurityState(SS_Secure), access to this field is RES0.
- When IsCurrentSecurityState(SS_NonSecure), access to this field is RES0.
- Access to this field is RO if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_SRMASK is implemented
- EL2 is the current Exception level
- IsSystemRegisterMaskingEnabled(EL2)
- TCR2MASK_EL2.AMEC0 == '1'

Otherwise:

Reserved, RES0.

HAFT, bit [11]

When FEAT_HAFT is implemented:

Hardware managed Access Flag for Table descriptors.

Enables the Hardware managed Access Flag for Table descriptors.

HAFT	Meaning
0b0	Hardware managed Access Flag for Table descriptors is disabled.
0b1	Hardware managed Access Flag for Table descriptors is enabled.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.HAFT == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

PTTWI, bit [10]

When FEAT_THE is implemented:

Permit Translation table walk Incoherence.

Permits RCWS instructions to generate writes that have the Reduced Coherence property.

PTTWI	Meaning
0b0	Write accesses generated by RCWS do not have the Reduced Coherence property.
0b1	Write accesses generated by RCWS have the Reduced Coherence property.

This bit is permitted to be implemented as a read-only bit with a fixed value of 0.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.PTTWI == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [9:6]

Reserved, RES0.

D128, bit [5]

When FEAT_D128 is implemented:

Enables VMSAv9-128 translation system.

D128	Meaning
0b0	Translation system follows VMSAv8-64 translation process.
0b1	Translation system follows VMSAv9-128 translation process.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.D128 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

AIE, bit [4]

When FEAT_AIE is implemented:

Enable Attribute Indexing Extension.

AIE	Meaning
0b0	Attribute Indexing Extension Disabled.
0b1	Attribute Indexing Extension Enabled.

This field is RES1 when TCR2_EL2.D128 is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.AIE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

POE, bit [3]

When FEAT_S1POE is implemented:

Enables Permission Overlay for privileged accesses from EL2&0 translation regime.

POE	Meaning
0b0	Permission overlay disabled for EL2 access in stage 1 of EL2&0 translation regime.
0b1	Permission overlay enabled for EL2 access in stage 1 of EL2&0 translation regime.

This bit is not permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.POE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

E0POE, bit [2]

When FEAT_S1POE is implemented:

Enables Permission Overlay for unprivileged accesses from EL2&0 translation regime.

E0POE	Meaning
0b0	Permission overlay disabled for EL0 access in stage 1 of EL2&0 translation regime.
0b1	Permission overlay enabled for EL0 access in stage 1 of EL2&0 translation regime.

This bit is not permitted to be cached in a TLB.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.E0POE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

PIE, bit [1]

When FEAT_S1PIE is implemented:

Enables usage of Indirect Permission Scheme.

PIE	Meaning
0b0	Direct permission model.
0b1	Indirect permission model.

This field is RES1 when TCR2_EL2.D128 is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.PIE == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

PnCH, bit [0]

When FEAT_THE is implemented:

Protected attribute enable. Enables use of bit[52] of stage 1 translation table entries as the Protected bit, for translations using TTBRn_EL2 when HCR_EL2.E2H is 1.

PnCH	Meaning
0b0	For translations using TTBRn_EL2, bit[52] of each stage 1 translation table entry is not the Protected bit.
0b1	For translations using TTBR0_EL2, bit[52] of each stage 1 translation table entry is the Protected bit.

If bit[52] is used as the Protected bit, it is not used as the Contiguous bit.

This field is RES0 when TCR2_EL2.D128 is 1.

When EL3 is implemented and SCR_EL3.TCR2En == 0, this field is ignored by the PE and treated as zero.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCR2MASK_EL2.PnCH == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TCR2_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TCR2_EL2` or `TCR2_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to TCR2_EL2 are masked by TCR2MASK_EL2.

Accesses to this register use the following encodings in the System register encoding space:

```
MSR TCR2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TCR2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR2_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, TCR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x270);
    else
        X{64}(t) = TCR2_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TCR2_EL2();
    else
        X{64}(t) = TCR2_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR2_EL1();
end;
```

```
MSR TCR2_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TCR_EL1 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            NVMem(0x270) = X{64}(t);
        else
            TCR2_EL1() = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            TCR2_EL2() = X{64}(t);
        else
            TCR2_EL1() = X{64}(t);
        end;
    elsif PSTATE.EL == EL3 then
        TCR2_EL1() = X{64}(t);
    end;
```

When FEAT_VHE is implemented MRS <Xt>, TCR2_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_TCR2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TCR2En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TCR2_EL2();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(t) = TCR2_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.191 TCR2MASK_EL1, Extended Translation Control Masking Register (EL1)

The TCR2MASK_EL1 characteristics are:

Purpose

Mask register to prevent updates of fields in TCR2_EL1 on writes to TCR2_EL1 or TCR2ALIAS_EL1.

Configuration

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCR2MASK_EL1 are UNDEFINED.

Attributes

TCR2MASK_EL1 is a 64-bit register.

Field descriptions

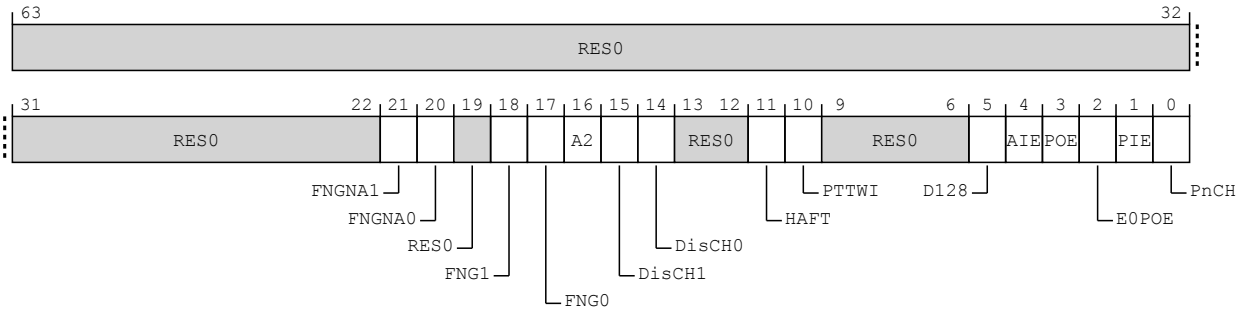

Bits [63:22]

Reserved, RES0.

FNGNA1, bit [21]

When FEAT_THE is implemented:

Mask bit for FNGNA1.

FNGNA1	Meaning
0b0	TCR2_EL1.FNGNA1 is writable.
0b1	TCR2_EL1.FNGNA1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FNGNA0, bit [20]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_THE is implemented:

Mask bit for FNGNA0.

FNGNA0	Meaning
0b0	TCR2_EL1.FNGNA0 is writable.
0b1	TCR2_EL1.FNGNA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [19]

Reserved, RES0.

FNG1, bit [18]

When FEAT_ASID2 is implemented:

Mask bit for FNG1.

FNG1	Meaning
0b0	TCR2_EL1.FNG1 is writable.
0b1	TCR2_EL1.FNG1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FNG0, bit [17]

When FEAT_ASID2 is implemented:

Mask bit for FNG0.

FNG0	Meaning
0b0	TCR2_EL1.FNG0 is writable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FNG0	Meaning
0b1	TCR2_EL1.FNG0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

A2, bit [16]

When FEAT_ASID2 is implemented:

Mask bit for A2.

A2	Meaning
0b0	TCR2_EL1.A2 is writable.
0b1	TCR2_EL1.A2 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DisCH1, bit [15]

When FEAT_D128 is implemented:

Mask bit for DisCH1.

DisCH1	Meaning
0b0	TCR2_EL1.DisCH1 is writable.
0b1	TCR2_EL1.DisCH1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DisCH0, bit [14]

When FEAT_D128 is implemented:

Mask bit for DisCH0.

DisCH0	Meaning
`0b0`	TCR2_EL1.DisCH0 is writable.
`0b1`	TCR2_EL1.DisCH0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [13:12]

Reserved, RES0.

HAFT, bit [11]

When FEAT_HAFT is implemented:

Mask bit for HAFT.

HAFT	Meaning
`0b0`	TCR2_EL1.HAFT is writable.
`0b1`	TCR2_EL1.HAFT is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PTTWI, bit [10]

When FEAT_THE is implemented:

Mask bit for PTTWI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PTTWI	Meaning
0b0	TCR2_EL1.PTTWI is writable.
0b1	TCR2_EL1.PTTWI is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [9:6]

Reserved, RES0.

D128, bit [5]

When FEAT_D128 is implemented:

Mask bit for D128.

D128	Meaning
0b0	TCR2_EL1.D128 is writable.
0b1	TCR2_EL1.D128 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AIE, bit [4]

When FEAT_AIE is implemented:

Mask bit for AIE.

AIE	Meaning
0b0	TCR2_EL1.AIE is writable.
0b1	TCR2_EL1.AIE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

POE, bit [3]

When FEAT_S1POE is implemented:

Mask bit for POE.

POE	Meaning
0b0	TCR2_EL1.POE is writable.
0b1	TCR2_EL1.POE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0POE, bit [2]

When FEAT_S1POE is implemented:

Mask bit for E0POE.

E0POE	Meaning
0b0	TCR2_EL1.E0POE is writable.
0b1	TCR2_EL1.E0POE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PIE, bit [1]

When FEAT_S1PIE is implemented:

Mask bit for PIE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PIE	Meaning
0b0	TCR2_EL1.PIE is writable.
0b1	TCR2_EL1.PIE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PnCH, bit [0]

When FEAT_THE is implemented:

Mask bit for PnCH.

PnCH	Meaning
0b0	TCR2_EL1.PnCH is writable.
0b1	TCR2_EL1.PnCH is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TCR2MASK_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name TCR2MASK_EL1 or TCR2MASK_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCR2MASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HFGRTR2_EL2().nTCR2MASK_EL1 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x338);
        else
            X{64}(t) = TCR2MASK_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = TCR2MASK_EL2();
        else
            X{64}(t) = TCR2MASK_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = TCR2MASK_EL1();
    end;
```

MSR TCR2MASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nTCR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x338) = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif !IsZero(TCR2MASK_EL1()) then
        Undefined();
    else
        TCR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(TCR2MASK_EL2()) then
            Undefined();
        else
            TCR2MASK_EL2() = X{64}(t);
        end;
    else
        TCR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR2MASK_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TCR2MASK_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x338);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TCR2MASK_EL1();
        end;
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCR2MASK_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR TCR2MASK_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x338) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            TCR2MASK_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TCR2MASK_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.192 TCR2MASK_EL2, Extended Translation Control Masking Register (EL2)

The TCR2MASK_EL2 characteristics are:

Purpose

Mask register to prevent updates of fields in TCR2_EL2 on writes.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCR2MASK_EL2 are UNDEFINED.

Attributes

TCR2MASK_EL2 is a 64-bit register.

Field descriptions

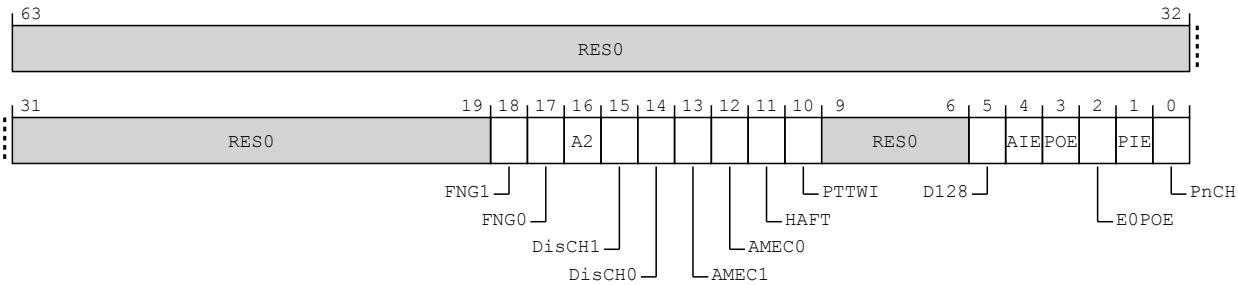

Bits [63:19]

Reserved, RES0.

FNG1, bit [18]

When FEAT_ASID2 is implemented:

Mask bit for FNG1.

FNG1	Meaning
0b0	TCR2_EL2.FNG1 is writable.
0b1	TCR2_EL2.FNG1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FNG0, bit [17]

When FEAT_ASID2 is implemented:

Mask bit for FNG0.

FNG0	Meaning
0b0	TCR2_EL2.FNG0 is writable.
0b1	TCR2_EL2.FNG0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

A2, bit [16]

When FEAT_ASID2 is implemented:

Mask bit for A2.

A2	Meaning
0b0	TCR2_EL2.A2 is writable.
0b1	TCR2_EL2.A2 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DisCH1, bit [15]

When FEAT_D128 is implemented:

Mask bit for DisCH1.

DisCH1	Meaning
0b0	TCR2_EL2.DisCH1 is writable.
0b1	TCR2_EL2.DisCH1 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DisCH0, bit [14]

When FEAT_D128 is implemented:

Mask bit for DisCH0.

DisCH0	Meaning
`0b0`	TCR2_EL2.DisCH0 is writable.
`0b1`	TCR2_EL2.DisCH0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMEC1, bit [13]

When FEAT_MEC is implemented:

Mask bit for AMEC1.

AMEC1	Meaning
`0b0`	TCR2_EL2.AMEC1 is writable.
`0b1`	TCR2_EL2.AMEC1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMEC0, bit [12]

When FEAT_MEC is implemented:

Mask bit for AMEC0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMEC0	Meaning
0b0	TCR2_EL2.AMEC0 is writable.
0b1	TCR2_EL2.AMEC0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HAFT, bit [11]

When FEAT_HAFT is implemented:

Mask bit for HAFT.

HAFT	Meaning
0b0	TCR2_EL2.HAFT is writable.
0b1	TCR2_EL2.HAFT is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PTTWI, bit [10]

When FEAT_THE is implemented:

Mask bit for PTTWI.

PTTWI	Meaning
0b0	TCR2_EL2.PTTWI is writable.
0b1	TCR2_EL2.PTTWI is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [9:6]

Reserved, RES0.

D128, bit [5]

When FEAT_D128 is implemented:

Mask bit for D128.

D128	Meaning
0b0	TCR2_EL2.D128 is writable.
0b1	TCR2_EL2.D128 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AIE, bit [4]

When FEAT_AIE is implemented:

Mask bit for AIE.

AIE	Meaning
0b0	TCR2_EL2.AIE is writable.
0b1	TCR2_EL2.AIE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

POE, bit [3]

When FEAT_S1POE is implemented:

Mask bit for POE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

POE	Meaning
0b0	TCR2_EL2.POE is writable.
0b1	TCR2_EL2.POE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0POE, bit [2]

When FEAT_S1POE is implemented:

Mask bit for E0POE.

E0POE	Meaning
0b0	TCR2_EL2.E0POE is writable.
0b1	TCR2_EL2.E0POE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PIE, bit [1]

When FEAT_S1PIE is implemented:

Mask bit for PIE.

PIE	Meaning
0b0	TCR2_EL2.PIE is writable.
0b1	TCR2_EL2.PIE is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PnCH, bit [0]

When FEAT_THE is implemented:

Mask bit for PnCH.

PnCH	Meaning
0b0	TCR2_EL2.PnCH is writable.
0b1	TCR2_EL2.PnCH is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TCR2MASK_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name TCR2MASK_EL2 or TCR2MASK_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCR2MASK_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TCR2MASK_EL2();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(t) = TCR2MASK_EL2();
end;
```

MSR TCR2MASK_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsZero(TCR2MASK_EL2()) then
        Undefined();
    else
        TCR2MASK_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR2MASK_EL2() = X{64}(t);
end;
```

MRS <Xt>, TCR2MASK_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nTCR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x338);
    else
        X{64}(t) = TCR2MASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TCR2MASK_EL2();
    else
        X{64}(t) = TCR2MASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR2MASK_EL1();
end;
```

MSR TCR2MASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b011

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nTCR2MASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x338) = X{64}(t);
    elsif !IsZero(TCR2MASK_EL1()) then
        Undefined();
    else
        TCR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(TCR2MASK_EL2()) then
            Undefined();
        else
            TCR2MASK_EL2() = X{64}(t);
        end;
    else
        TCR2MASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR2MASK_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.193 TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

The control register for stage 1 of the EL1&0 translation regime.

Configuration

AArch64 System register TCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TTBCR[31:0].

AArch64 System register TCR_EL1 bits [63:32] are architecturally mapped to AArch32 System register TTBCR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TCR_EL1 are UNDEFINED.

Attributes

TCR_EL1 is a 64-bit register.

Field descriptions

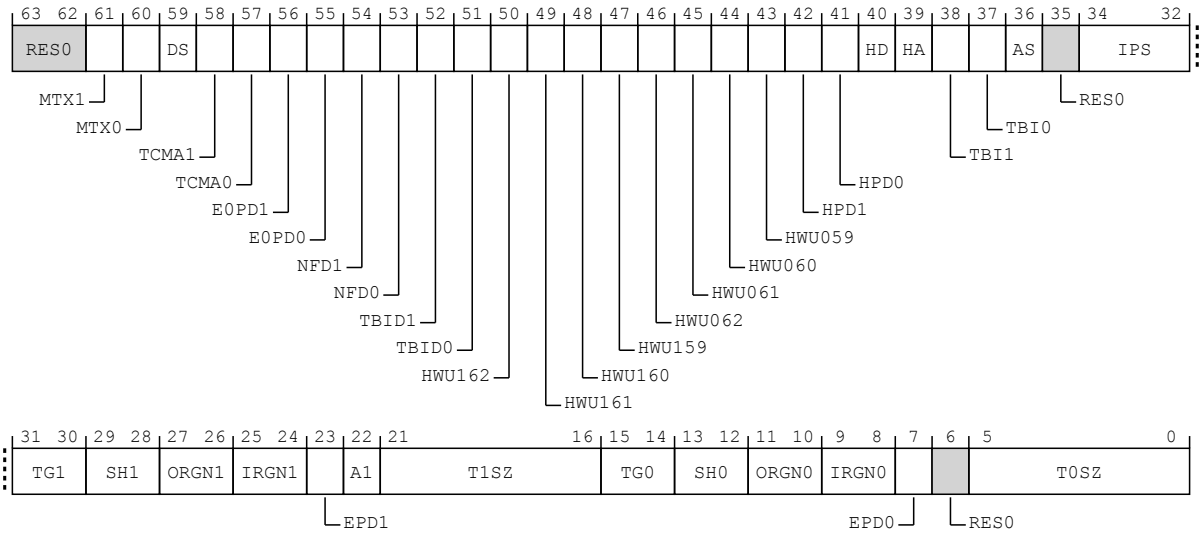

Any of the bits in TCR_EL1, other than the EPDx bits when they have the value 1, and the A1 bit are permitted to be cached in a TLB.

Bits [63:62]

Reserved, RES0.

MTX1, bit [61]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Extended memory tag checking.

This field controls address generation and Canonical tagging when EL0 and EL1 are using AArch64 where the data address would be translated by tables pointed to by TTBR1_EL1.

This control has an effect regardless of whether stage 1 of the EL1&0 translation regime is enabled or not.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTX1	Meaning
0b0	Canonical tagging is disabled. This control has no effect on address generation.
0b1	Canonical tagging is enabled. Bits[59:56] of a 64-bit VA hold a Logical Address Tag, and all of the following apply: • Bits[59:56] are treated as 0b1111 when checking if the address is out of range. • If FEAT_PAuth is implemented, bits[59:56] are not part of the PAC field. • A Canonical Tag Check operation is performed on Tag Checked memory accesses to a Canonically Tagged memory location.

For more information see Logical address tagging and Memory region tagging types.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.MTX1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

MTX0, bit [60]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Extended memory tag checking.

This field controls address generation and Canonical tagging when EL0 and EL1 are using AArch64 where the data address would be translated by tables pointed to by TTBR0_EL1.

This control has an effect regardless of whether stage 1 of the EL1&0 translation regime is enabled or not.

MTX0	Meaning
0b0	Canonical tagging is disabled. This control has no effect on address generation.
0b1	Canonical tagging is enabled. Bits[59:56] of a 64-bit VA hold a Logical Address Tag, and all of the following apply: • Bits[59:56] are treated as 0b0000 when checking if the address is out of range. • If FEAT_PAuth is implemented, bits[59:56] are not part of the PAC field. • A Canonical Tag Check operation is performed on Tag Checked memory accesses to a Canonically Tagged memory location.

For more information see Logical address tagging and Memory region tagging types.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.MTX0 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

DS, bit [59]

When FEAT_LPA2 is implemented and (FEAT_D128 is not implemented or TCR2_EL1.D128 == '0'):

This field affects:

- Whether a 52-bit output address can be described by the translation tables of the 4KB or 16KB translation granules.
- The minimum value of TCR_EL1.{T0SZ,T1SZ}.
- How and where shareability for Block and Page descriptors are encoded.

DS	Meaning
`0b0`	Bits[49:48] of translation descriptors are RES0. Bits[9:8] in Block and Page descriptors encode shareability information in the SH[1:0] field. Bits[9:8] in Table descriptors are ignored by hardware. The minimum value of the TCR_EL1.{T0SZ, T1SZ} fields is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. Output address[51:48] is `0b0000`.
`0b1`	Bits[49:48] of translation descriptors hold output address[49:48]. Bits[9:8] of Translation table descriptors hold output address[51:50]. The shareability information of Block and Page descriptors for cacheable locations is determined by: • TCR_EL1.SH0 if the VA is translated using tables pointed to by TTBR0_EL1. • TCR_EL1.SH1 if the VA is translated using tables pointed to by TTBR1_EL1. The minimum value of the TCR_EL1.{T0SZ, T1SZ} fields is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. All calculations of the stage 1 base address are modified for tables of fewer than 8 entries so that the table is aligned to 64 bytes. Bits[5:2] of TTBR0_EL1 or TTBR1_EL1 are used to hold bits[51:48] of the output address in all cases. **Note** As FEAT_LVA must be implemented if TCR_EL1.DS == 1, the minimum value of the TCR_EL1.{T0SZ, T1SZ} fields is 12, as determined by that extension. For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as `0b0000`. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as `0b00`. **Note** This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.DS == '1', access to this field is RO.

Otherwise:

The Effective value of this bit is 0.

Access to this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCMA1, bit [58]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL1, and at EL0 if the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, when address[59:55] = `0b11111`.

TCMA1	Meaning
`0b0`	This control has no effect on the generation of Unchecked accesses at EL1 or EL0.
`0b1`	All accesses at EL1 and EL0 are Unchecked.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TCMA1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCMA0, bit [57]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL1, and at EL0 if the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, when address[59:55] = `0b00000`.

TCMA0	Meaning
`0b0`	This control has no effect on the generation of Unchecked accesses at EL1 or EL0.
`0b1`	All accesses at EL1 and EL0 are Unchecked.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TCMA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_E0PD is implemented:

Faulting control for unprivileged access to any address translated by TTBR1_EL1.

E0PD1	Meaning
0b0	Unprivileged access to any address translated by TTBR1_EL1 will not generate a fault by this mechanism.
0b1	Unprivileged access to any address translated by TTBR1_EL1 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.E0PD1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When FEAT_E0PD is implemented:

Faulting control for unprivileged access to any address translated by TTBR0_EL1.

E0PD0	Meaning
0b0	Unprivileged access to any address translated by TTBR0_EL1 will not generate a fault by this mechanism.
0b1	Unprivileged access to any address translated by TTBR0_EL1 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.E0PD0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When FEAT_SVE is implemented:

Non-Fault translation timing Disable when using TTBR1_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Controls how a TLB miss is reported in response to a non-fault unprivileged access for a virtual address that is translated using TTBR1_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

NFD1	Meaning
0b0	Does not affect the handling of a TLB miss on accesses translated using TTBR1_EL1.
0b1	A TLB miss on a virtual address that is translated using TTBR1_EL1 due to the specified access types causes the access to fail without taking an exception. The amount of time that the failure takes to be handled should not predictively leak whether it was caused by a TLB miss or a Permission fault, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.NFD1 == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

NFD0, bit [53]

When FEAT_SVE is implemented:

Non-Fault translation timing Disable when using TTBR0_EL1.

Controls how a TLB miss is reported in response to a non-fault unprivileged access for a virtual address that is translated using TTBR0_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

NFD0	Meaning
0b0	Does not affect the handling of a TLB miss on accesses translated using TTBR0_EL1.
0b1	A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. The amount of time that the failure takes to be handled should not predictively leak whether it was caused by a TLB miss or a Permission fault, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.NFD0 == ‘1’, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

TBID1, bit [52]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID1	Meaning
0b0	TCR_EL1.TBI1 applies to Instruction and Data accesses.
0b1	TCR_EL1.TBI1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TBID1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID0	Meaning
0b0	TCR_EL1.TBI0 applies to Instruction and Data accesses.
0b1	TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TBID0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU162, bit [50]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU162	Meaning
0b0	For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU162 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU161	Meaning
0b0	For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU161 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU160	Meaning
0b0	For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU160 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU159	Meaning
0b0	For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU159 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When FEAT_HPDS2 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU062	Meaning
0b0	For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU062 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU061, bit [45]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU061	Meaning
0b0	For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU061 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU060, bit [44]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU060	Meaning
0b0	For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU060 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU059, bit [43]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU059	Meaning
0b0	For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HWU059 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPD1	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HPD1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL1.

HPD0	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HPD0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HD, bit [40]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL0 and EL1.

HD	Meaning
0b0	Stage 1 hardware management of dirty state disabled.
0b1	Stage 1 hardware management of dirty state enabled.

When the Effective value of TCR_EL1.HA is 0, this field behaves as 0 for all purposes other than a direct read of the value of this field.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HD == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HA, bit [39]

When FEAT_HAF is implemented:

Hardware Access flag update in stage 1 translations from EL0 and EL1.

HA	Meaning
0b0	Stage 1 Access flag update disabled.
0b1	Stage 1 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.HA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the TTBR1_EL1 region, or ignored and used for tagged addresses.

TBI1	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated at EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL1.TBID1 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL1 or EL0.
- An exception taken to EL1.
- An exception return to EL1 or EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TBI1 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TBI0, bit [37]

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL1 region, or ignored and used for tagged addresses.

TBI0	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated at EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL1 or EL0.
- An exception taken to EL1.
- An exception return to EL1 or EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TBI0 == '1', access to this field is RO.

AS, bit [36]

ASID Size.

AS	Meaning
0b0	8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.
0b1	16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.AS == '1', access to this field is RO.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

Intermediate Physical Address Size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IPS	Meaning
0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.
0b111	56 bits, 64PB.

The values 0b110 and 0b111 represent different output address sizes depending on implementation choices and translation configuration.

The following table captures the output address size represented by the value 0b110:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1]	Represented OA size
Any	0b0101	Any	Any	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	0	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	1	52 bits, 4PB
VMSAv8-64	0b011x	64KB	N/A	52 bits, 4PB
VMSAv9-128	0b011x	Any	N/A	52 bits, 4PB

[1] This column represents the value of TCR_EL1.DS.

The following table captures the output address size represented by the value 0b111:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	Represented OA size
Any	0b0110	Any	OA size represented by 0b110
VMSAv8-64	0b0111	Any	OA size represented by 0b110
VMSAv9-128	0b0111	Any	56 bits, 64PB

If 52-bit PA is supported, and the translation table descriptors cannot express an OA larger than 48-bits, then bits[51:48] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL1.

If 56-bit PA is supported, and the translation table descriptors cannot express an OA larger than 52-bits, then bits[55:52] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL1.

If the output address size represented by this field is larger than the supported PA size expressed in ID_AA64MMFR0_EL1.PARange, then the output address size is treated as being the same as the supported PA size. Arm strongly recommends that software avoids configuring this field to a value representing an output address size larger than the supported PA size.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.IPS == '1', access to this field is RO.

TG1, bits [31:30]

Granule size for the TTBR1_EL1.

TG1	Meaning
0b01	16KB.
0b10	4KB.
0b11	64KB.

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TG1 == '1', access to this field is RO.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using TTBR1_EL1.

SH1	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.SH1 == '1', access to this field is RO.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ORGN1	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.ORGN1 == '1', access to this field is RO.

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

IRGN1	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.IRGN1 == '1', access to this field is RO.

EPD1, bit [23]

Translation table walk disable for translations using TTBR1_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL1. The encoding of this bit is:

EPD1	Meaning
0b0	Perform translation table walks using TTBR1_EL1.
0b1	A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.EPD1 == '1', access to this field is RO.

A1, bit [22]

Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1	Meaning
0b0	TTBR0_EL1.ASID defines the ASID.
0b1	TTBR1_EL1.ASID defines the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.A1 == '1', access to this field is RO.

T1SZ, bits [21:16]

The size offset of the memory region addressed by TTBR1_EL1. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, TCR_EL1.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, TCR_EL1.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.T1SZ == '1', access to this field is RO.

TG0, bits [15:14]

Granule size for the TTBR0_EL1.

TG0	Meaning
0b00	4KB
0b01	64KB
0b10	16KB

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.TG0 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL1.

SH0	Meaning
0b00	Non-shareable
0b10	Outer Shareable
0b11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.SH0 == '1', access to this field is RO.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.ORGN0 == '1', access to this field is RO.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.IRGN0 == '1', access to this field is RO.

EPD0, bit [7]

Translation table walk disable for translations using TTBR0_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0_EL1. The encoding of this bit is:

EPD0	Meaning
`0b0`	Perform translation table walks using TTBR0_EL1.
`0b1`	A TLB miss on an address that is translated using TTBR0_EL1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.EPD0 == '1', access to this field is RO.

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL1. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, TCR_EL1.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, TCR_EL1.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL1 is the current Exception level, IsSystemRegisterMaskingEnabled(EL1), and TCRMASK_EL1.T0SZ == '1', access to this field is RO.

Accessing TCR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TCR_EL1` or `TCR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to TCR_EL1 are masked by TCRMASK_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x120);
    else
        X{64}(t) = TCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCR_EL2();
    else
        X{64}(t) = TCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR_EL1();
end;
```

```
MSR TCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x120) = X{64}(t);
    else
        TCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TCR_EL2() = X{64}(t);
    else
        TCR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    TCR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x120);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR TCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x120) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if ELIsInHost(EL2) then
            TCR_EL1() = X{64}(t);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            TCR_EL1() = X{64}(t);
        else
            Undefined();
        end;
    end;
```

When FEAT_SRMASK is implemented MRS <Xt>, TCRALIAS_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b110

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2().TRVM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
            ↪'0') || HFGRTR2_EL2().nTCRALIAS_EL1 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x120);
        else
            X{64}(t) = TCR_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            X{64}(t) = TCR_EL2();
        else
            X{64}(t) = TCR_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = TCR_EL1();
    end;
```

When FEAT_SRMASK is implemented MSR TCRALIAS_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b110

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2().TVM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 ==
        ↪'0') || HFGWTR2_EL2().nTCRALIAS_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x120) = X{64}(t);
    else
        TCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TCR_EL2() = X{64}(t);
    else
        TCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.194 TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

Purpose

The control register for stage 1 of the EL2, or EL2&0, translation regime:

- When the Effective value of HCR_EL2.E2H is not 1, this register controls stage 1 of the EL2 translation regime, that supports a single VA range, translated using TTBR0_EL2.
- When the Effective value of HCR_EL2.E2H is 1, this register controls stage 1 of the EL2&0 translation regime, that supports both:
  - A lower VA range, translated using TTBR0_EL2.
  - A higher VA range, translated using TTBR1_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register TCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HTCR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TCR_EL2 are UNDEFINED.

Attributes

TCR_EL2 is a 64-bit register.

Field descriptions

When EffectiveHCR_EL2_E2H() == '0':

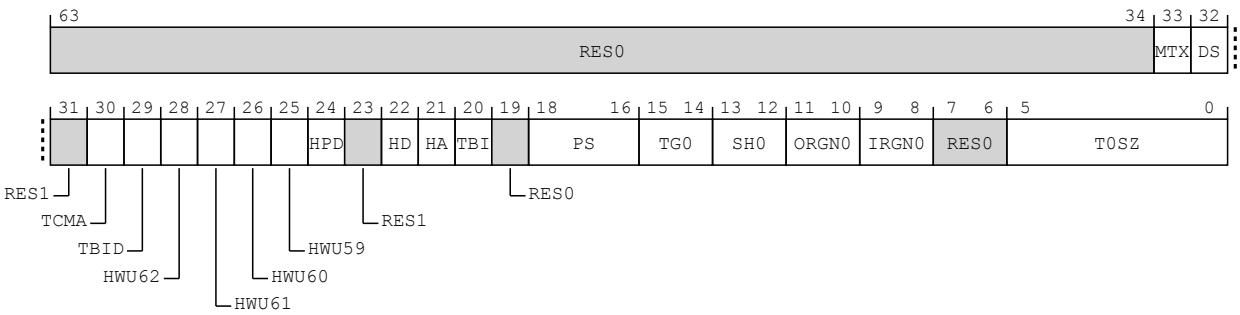

Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

Bits [63:34]

Reserved, RES0.

MTX, bit [33]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Extended memory tag checking.

This field controls address generation and Canonical tagging when EL2 is using AArch64 where the data address would be translated by tables pointed to by TTBR0_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This control has an effect regardless of whether stage 1 of the EL2 translation regime is enabled or not.

MTX	Meaning
`0b0`	Canonical tagging is disabled. This control has no effect on address generation.
`0b1`	Canonical tagging is enabled. Bits[59:56] of a 64-bit VA hold a Logical Address Tag, and all of the following apply: • Bits[59:56] are treated as `0b0000` when checking if the address is out of range. • If FEAT_PAuth is implemented, bits[59:56] are not part of the PAC field. • A Canonical Tag Check operation is performed on Tag Checked memory accesses to a Canonically Tagged memory location.

For more information see Logical address tagging and Memory region tagging types.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DS, bit [32]

When FEAT_LPA2 is implemented:

This field affects:

- Whether a 52-bit output address can be described by the translation tables of the 4KB or 16KB translation granules.
- The minimum value of TCR_EL2.T0SZ.
- How and where shareability for Block and Page descriptors are encoded.

DS	Meaning
`0b0`	Bits[49:48] of translation descriptors are RES0. Bits[9:8] in Block and Page descriptors encode shareability information in the SH[1:0] field. Bits[9:8] in table descriptors are ignored by hardware. The minimum value of TCR_EL2.T0SZ is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. Output address[51:48] is `0b0000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DS	Meaning
0b1	Bits[49:48] of translation descriptors hold output address[49:48]. Bits[9:8] of Translation table descriptors hold output address[51:50]. The shareability information of Block and Page descriptors for cacheable locations is determined by TCR_EL2.SH0. The minimum value of TCR_EL2.T0SZ is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. All calculations of the stage 1 base address are modified for tables of fewer than 8 entries so that the table is aligned to 64 bytes. Bits[5:2] of TTBR0_EL2 are used to hold bits[51:48] of the output address in all cases. **Note** As FEAT_LVA must be implemented if TCR_EL2.DS == 1, the minimum value of the TCR_EL2.T0SZ field is 12, as determined by that extension. For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 0b00. **Note** This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

The Effective value of this bit is 0.

Access to this field is RES0.

Bit [31]

Reserved, RES1.

TCMA, bit [30]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL2 when address [59:56] = 0b0000.

TCMA	Meaning
0b0	This control has no effect on the generation of Unchecked accesses.
0b1	All accesses are Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID, bit [29]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID	Meaning
0b0	TCR_EL2.TBI applies to Instruction and Data accesses.
0b1	TCR_EL2.TBI applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

HWU62	Meaning
0b0	Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU61	Meaning
0b0	Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

HWU60	Meaning
0b0	Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

HWU59	Meaning
0b0	Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD, bit [24]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL2.

HPD	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled. **Note** In this case, bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE and are no longer reserved, allowing them to be used by software.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL2.

HD	Meaning
0b0	Stage 1 hardware management of dirty state disabled.
0b1	Stage 1 hardware management of dirty state enabled.

When the Effective value of TCR_EL2.HA is 0, this field behaves as 0 for all purposes other than a direct read of the value of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HA, bit [21]

When FEAT_HAF is implemented:

Hardware Access flag update in stage 1 translations from EL2.

HA	Meaning
0b0	Stage 1 Access flag update disabled.
0b1	Stage 1 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI, bit [20]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

For more information, see 'Address tagging'.

TBI	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated at EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID is 1, then this field only applies to Data accesses.

If the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL2.
- An exception taken to EL2.
- An exception return to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PS	Meaning
0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.

The following table captures the output address size represented by the value 0b110:

ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1] | Represented OA size |	
0b0101	Any	Any	48 bits, 256TB
0b011x	4KB or 16KB	0	48 bits, 256TB
0b011x	4KB or 16KB	1	52 bits, 4PB
0b011x	64KB	N/A	52 bits, 4PB

[1] This column represents the value of TCR_EL2.DS.

The output address size represented by the value 0b111 is the OA size represented by 0b110.

If 52-bit PA is supported, and the translation table descriptors cannot express an OA larger than 48-bits, then bits[51:48] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL2.

If 56-bit PA is supported, and the translation table descriptors cannot express an OA larger than 52-bits, then bits[55:52] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL2.

If the output address size represented by this field is larger than the supported PA size expressed in ID_AA64MMFR0_EL1.PARange, then the output address size is treated as being the same as the supported PA size. Arm strongly recommends that software avoids configuring this field to a value representing an output address size larger than the supported PA size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the TTBR0_EL2.

TG0	Meaning
0b00	4KB.
0b01	64KB.
0b10	16KB.

Other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

SH0	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRGN0	Meaning
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, TCR_EL2.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, TCR_EL2.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When EffectiveHCR_EL2_E2H() == '1':

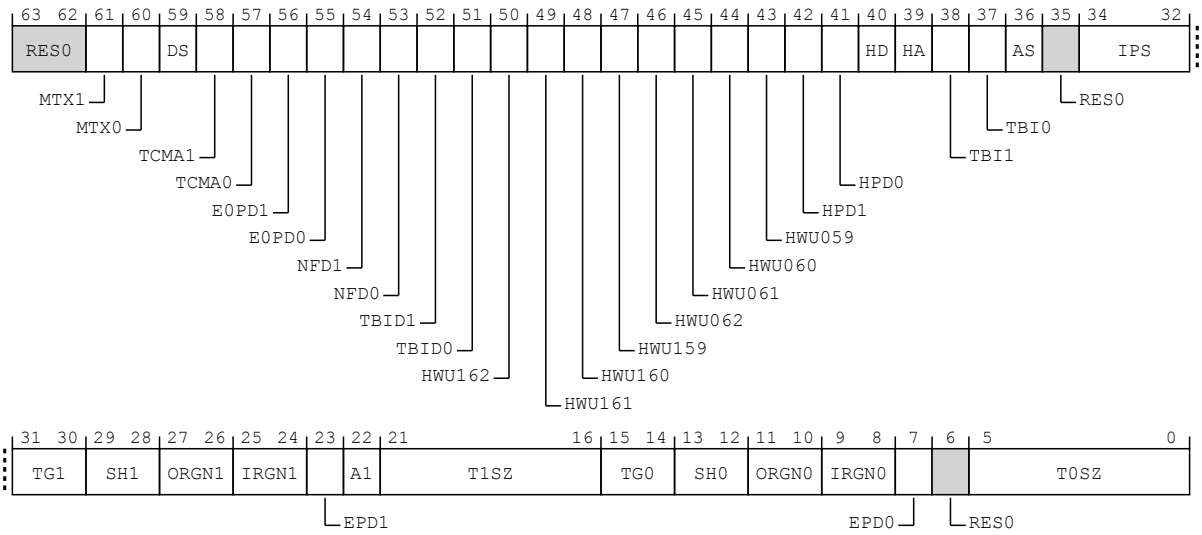

Any of the bits in TCR_EL2, other than the EPDx bits when they have the value 1, and the A1 bit are permitted to be cached in a TLB.

Bits [63:62]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTX1, bit [61]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Extended memory tag checking.

This field controls address generation and Canonical tagging when EL0 and EL2 are using AArch64 where the data address would be translated by tables pointed to by TTBR1_EL2.

This control has an effect regardless of whether stage 1 of the EL2&0 translation regime is enabled or not.

MTX1	Meaning
`0b0`	Canonical tagging is disabled. This control has no effect on address generation.
`0b1`	Canonical tagging is enabled. Bits[59:56] of a 64-bit VA hold a Logical Address Tag, and all of the following apply: • Bits[59:56] are treated as `0b1111` when checking if the address is out of range. • If FEAT_PAuth is implemented, bits[59:56] are not part of the PAC field. • A Canonical Tag Check operation is performed on Tag Checked memory accesses to a Canonically Tagged memory location.

For more information see Logical address tagging and Memory region tagging types.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.MTX1 == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

MTX0, bit [60]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Extended memory tag checking.

This field controls address generation and Canonical tagging when EL0 and EL2 are using AArch64 where the data address would be translated by tables pointed to by TTBR0_EL2.

This control has an effect regardless of whether stage 1 of the EL2&0 translation regime is enabled or not.

MTX0	Meaning
`0b0`	Canonical tagging is disabled. This control has no effect on address generation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTX0	Meaning
`0b1`	Canonical tagging is enabled. Bits[59:56] of a 64-bit VA hold a Logical Address Tag, and all of the following apply: • Bits[59:56] are treated as `0b0000` when checking if the address is out of range. • If FEAT_PAuth is implemented, bits[59:56] are not part of the PAC field. • A Canonical Tag Check operation is performed on Tag Checked memory accesses to a Canonically Tagged memory location.

For more information see Logical address tagging and Memory region tagging types.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.MTX0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

DS, bit [59]

When FEAT_LPA2 is implemented and (FEAT_D128 is not implemented or TCR2_EL2.D128 == '0'):

This field affects:

- Whether a 52-bit output address can be described by the translation tables of the 4KB or 16KB translation granules.
- The minimum value of TCR_EL2.{T0SZ,T1SZ}.
- How and where shareability for Block and Page descriptors are encoded.

DS	Meaning
`0b0`	Bits[49:48] of translation descriptors are RES0. Bits[9:8] in Block and Page descriptors encode shareability information in the SH[1:0] field. Bits[9:8] in table descriptors are ignored by hardware. The minimum value of the TCR_EL2.{T0SZ, T1SZ} fields is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. Output address[51:48] is `0b0000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DS	Meaning
0b1	Bits[49:48] of translation descriptors hold output address[49:48]. Bits[9:8] of Translation table descriptors hold output address[51:50]. The shareability information of Block and Page descriptors for cacheable locations is determined by: • TCR_EL2.SH0 if the VA is an address that is translated using tables pointed to by TTBR0_EL2. • TCR_EL2.SH1 if the VA is an address that is translated using tables pointed to by TTBR1_EL2. The minimum value of the TCR_EL2.{T0SZ, T1SZ} fields is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. All calculations of the stage 1 base address are modified for tables of fewer than 16 entries so that the table is aligned to 64 bytes. Bits[5:2] of TTBR0_EL2 or TTBR1_EL2 are used to hold bits[51:48] of the output address in all cases. **Note** As FEAT_LVA must be implemented if TCR_EL2.DS == 1, the minimum value of the TCR_EL2.{T0SZ, T1SZ} fields is 12, as determined by that extension. For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 0b00. **Note** This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.DS == '1', access to this field is RO.

Otherwise:

Reserved, RES0, and the Effective value of this bit is 0.

TCMA1, bit [58]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL2, and at EL0 if HCR_EL2.TGE=1, when address[59:55] = 0b11111.

TCMA1	Meaning
0b0	This control has no effect on the generation of Unchecked accesses at EL2 or EL0.
0b1	All accesses are Unchecked.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TCMA1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TCMA0, bit [57]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL2, and at EL0 if HCR_EL2.TGE=1, when address[59:55] = `0b00000`.

TCMA0	Meaning
`0b0`	This control has no effect on the generation of Unchecked accesses at EL2 or EL0.
`0b1`	All accesses are Unchecked.

Note

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TCMA0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

When FEAT_E0PD is implemented:

Faulting control for unprivileged access to any address translated by TTBR1_EL2.

E0PD1	Meaning
`0b0`	Unprivileged access to any address translated by TTBR1_EL2 will not generate a fault by this mechanism.
`0b1`	Unprivileged access to any address translated by TTBR1_EL2 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.E0PD1 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When FEAT_E0PD is implemented:

Faulting control for unprivileged access to any address translated by TTBR0_EL2.

E0PD0	Meaning
0b0	Unprivileged access to any address translated by TTBR0_EL2 will not generate a fault by this mechanism.
0b1	Unprivileged access to any address translated by TTBR0_EL2 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.E0PD0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When FEAT_SVE is implemented:

Non-Fault translation timing Disable when using TTBR1_EL2.

Controls how a TLB miss is reported in response to a non-fault unprivileged access for a virtual address that is translated using TTBR1_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

NFD1	Meaning
0b0	Does not affect the handling of a TLB miss on accesses translated using TTBR1_EL2.
0b1	A TLB miss on a virtual address that is translated using TTBR1_EL2 due to the specified access types causes the access to fail without taking an exception. The amount of time that the failure takes to be handled should not predictively leak whether it was caused by a TLB miss or a Permission fault, to mitigate attacks that use fault timing.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.NFD1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

NFD0, bit [53]

When FEAT_SVE is implemented:

Non-Fault translation timing Disable when using TTBR0_EL2.

Controls how a TLB miss is reported in response to a non-fault unprivileged access for a virtual address that is translated using TTBR0_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

NFD0	Meaning
0b0	Does not affect the handling of a TLB miss on accesses translated using TTBR0_EL2.
0b1	A TLB miss on a virtual address that is translated using TTBR0_EL2 due to the specified access types causes the access to fail without taking an exception. The amount of time that the failure takes to be handled should not predictively leak whether it was caused by a TLB miss or a Permission fault, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.NFD0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TBID1, bit [52]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID1	Meaning
0b0	TCR_EL2.TBI1 applies to Instruction and Data accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	TCR_EL2.TBI1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TBID1 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For more information, see 'Address tagging'.

TBID0	Meaning
0b0	TCR_EL2.TBI0 applies to Instruction and Data accesses.
0b1	TCR_EL2.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TBID0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

HWU162	Meaning
0b0	For translations using TTBR1_EL2, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL2, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU162 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

HWU161	Meaning
0b0	For translations using TTBR1_EL2, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL2, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU161 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

HWU160	Meaning
0b0	For translations using TTBR1_EL2, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL2, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU160 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

HWU159	Meaning
0b0	For translations using TTBR1_EL2, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL2, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU159 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL2.

HWU062	Meaning
0b0	For translations using TTBR0_EL2, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL2, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU062 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

HWU061, bit [45]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL2.

HWU061	Meaning
0b0	For translations using TTBR0_EL2, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL2, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU061 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HWU060, bit [44]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL2.

HWU060	Meaning
0b0	For translations using TTBR0_EL2, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL2, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU060 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU059, bit [43]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL2.

HWU059	Meaning
0b0	For translations using TTBR0_EL2, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL2, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HWU059 == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL2.

HPD1	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HPD1 == ‘1’, access to this field is RO.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPD0	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HPD0 == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HD, bit [40]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL2.

HD	Meaning
0b0	Stage 1 hardware management of dirty state disabled.
0b1	Stage 1 hardware management of dirty state enabled.

When the Effective value of TCR_EL2.HA is 0, this field behaves as 0 for all purposes other than a direct read of the value of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HD == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

HA, bit [39]

When FEAT_HAF is implemented:

Hardware Access flag update in stage 1 translations from EL2.

HA	Meaning
0b0	Stage 1 Access flag update disabled.
0b1	Stage 1 Access flag update enabled.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.HA == '1', access to this field is RO.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR1_EL2 region, or ignored and used for tagged addresses.

For more information, see 'Address tagging'.

TBI1	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated at EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL2. It has an effect whether the EL2&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID1 is 1, then this field only applies to Data accesses.

If the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL2
- If the Effective value of HCR_EL2.{E2H, TGE} is {1,1}, a branch or procedure return within EL0.
- An exception taken to EL2.
- An exception return to EL2.
- If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then an exception return to EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TBI1 == '1', access to this field is RO.

TBI0, bit [37]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

For more information, see 'Address tagging'.

TBI0	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated at EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID0 is 1, then this field only applies to Data accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL2
- If the Effective value of HCR_EL2.{E2H, TGE} is {1,1}, a branch or procedure return within EL0.
- An exception taken to EL2.
- An exception return to EL2.
- If the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, then an exception return to EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TBI0 == '1', access to this field is RO.

AS, bit [36]

ASID Size.

AS	Meaning
0b0	8 bit - the upper 8 bits of TTBR0_EL2 and TTBR1_EL2 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.
0b1	16 bit - the upper 16 bits of TTBR0_EL2 and TTBR1_EL2 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.AS == '1', access to this field is RO.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

Intermediate Physical Address Size.

IPS	Meaning
0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IPS	Meaning
0b111	56 bits, 64PB.

The values 0b110 and 0b111 represent different output address sizes depending on implementation choices and translation configuration.

The following table captures the output address size represented by the value 0b110:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1]	Represented OA size
Any	0b0101	Any	Any	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	0	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	1	52 bits, 4PB
VMSAv8-64	0b011x	64KB	N/A	52 bits, 4PB
VMSAv9-128	0b011x	Any	N/A	52 bits, 4PB

[1] This column represents the value of TCR_EL2.DS.

The following table captures the output address size represented by the value 0b111:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	Represented OA size
Any	0b0110	Any	OA size represented by 0b110
VMSAv8-64	0b0111	Any	OA size represented by 0b110
VMSAv9-128	0b0111	Any	56 bits, 64PB

If 52-bit PA is supported, and the translation table descriptors cannot express an OA larger than 48-bits, then bits[51:48] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL2.

If 56-bit PA is supported, and the translation table descriptors cannot express an OA larger than 52-bits, then bits[55:52] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL2.

If the output address size represented by this field is larger than the supported PA size expressed in ID_AA64MMFR0_EL1.PARange, then the output address size is treated as being the same as the supported PA size. Arm strongly recommends that software avoids configuring this field to a value representing an output address size larger than the supported PA size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.IPS == ‘1’, access to this field is RO.

TG1, bits [31:30]

Granule size for the TTBR1_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG1	Meaning
0b01	16KB.
0b10	4KB.
0b11	64KB.

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TG1 == '1', access to this field is RO.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using TTBR1_EL2.

SH1	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.SH1 == '1', access to this field is RO.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL2.

ORGN1	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.ORGN1 == '1', access to this field is RO.

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL2.

IRGN1	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.IRGN1 == '1', access to this field is RO.

EPD1, bit [23]

Translation table walk disable for translations using TTBR1_EL2. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL2. The encoding of this bit is:

EPD1	Meaning
0b0	Perform translation table walks using TTBR1_EL2.
0b1	A TLB miss on an address that is translated using TTBR1_EL2 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.EPD1 == '1', access to this field is RO.

A1, bit [22]

Selects whether TTBR0_EL2 or TTBR1_EL2 defines the ASID. The encoding of this bit is:

A1	Meaning
0b0	TTBR0_EL2.ASID defines the ASID.
0b1	TTBR1_EL2.ASID defines the ASID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.A1 == '1', access to this field is RO.

T1SZ, bits [21:16]

The size offset of the memory region addressed by TTBR1_EL2. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, TCR_EL2.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, TCR_EL2.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.T1SZ == '1', access to this field is RO.

TG0, bits [15:14]

Granule size for the TTBR0_EL2.

TG0	Meaning
0b00	4KB.
0b01	64KB.
0b10	16KB.

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.TG0 == '1', access to this field is RO.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH0	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.SH0 == '1', access to this field is RO.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.ORGN0 == '1', access to this field is RO.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.IRGN0 == '1', access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPD0, bit [7]

Translation table walk disable for translations using TTBR0_EL2. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0_EL2. The encoding of this bit is:

EPD0	Meaning
0b0	Perform translation table walks using TTBR0_EL2.
0b1	A TLB miss on an address that is translated using TTBR0_EL2 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.EPD0 == '1', access to this field is RO.

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, TCR_EL2.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, TCR_EL2.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SRMASK is implemented, EL2 is the current Exception level, IsSystemRegisterMaskingEnabled(EL2), and TCRMASK_EL2.T0SZ == '1', access to this field is RO.

Accessing TCR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TCR_EL2` or `TCR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

If FEAT_SRMASK is implemented, accesses to TCR_EL2 are masked by TCRMASK_EL2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TCR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR_EL2();
end;
```

MSR TCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    TCR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    TCR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = NVMem(0x120);
    else
        X{64}(t) = TCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCR_EL2();
    else
        X{64}(t) = TCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR_EL1();
end;
```

When FEAT_VHE is implemented MSR TCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().TCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x120) = X{64}(t);
    else
        TCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TCR_EL2() = X{64}(t);
    else
        TCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.195 TCR_EL3, Translation Control Register (EL3)

The TCR_EL3 characteristics are:

Purpose

The control register for stage 1 of the EL3 translation regime.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCR_EL3 are UNDEFINED.

Attributes

TCR_EL3 is a 64-bit register.

Field descriptions

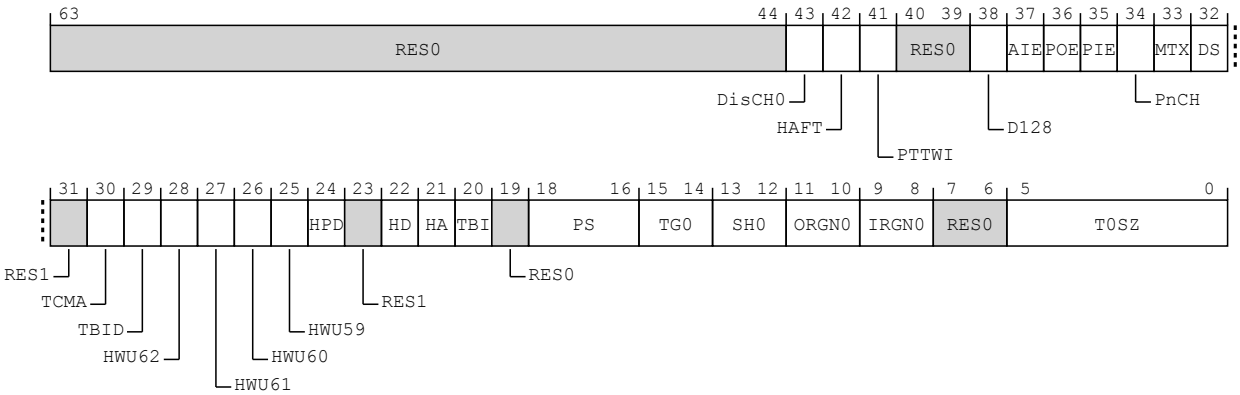

Unless stated otherwise, any of the bits in TCR_EL3 are permitted to be cached in a TLB.

Bits [63:44]

Reserved, RES0.

DisCH0, bit [43]

When FEAT_D128 is implemented and TCR_EL3.D128 == '1':

Disable the Contiguous bit for the Start Table.

DisCH0	Meaning
0b0	The Contiguous bit of Block or Page descriptors of the Start Table is not affected by this field.
0b1	The Contiguous bit of Block or Page descriptors of the Start Table is treated as 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

HAFT, bit [42]

When FEAT_HAFT is implemented:

Hardware managed Access Flag for Table descriptors.

Enables the Hardware managed Access Flag for Table descriptors.

HAFT	Meaning
0b0	Hardware managed Access Flag for Table descriptors is disabled.
0b1	Hardware managed Access Flag for Table descriptors is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PTTWI, bit [41]

When FEAT_THE is implemented:

Permit Translation table walk Incoherence.

Permits RCWS instructions to generate writes that have the Reduced Coherence property.

PTTWI	Meaning
0b0	Write accesses generated by RCWS at EL3 do not have the Reduced Coherence property.
0b1	Write accesses generated by RCWS at EL3 have the Reduced Coherence property.

This bit is permitted to be implemented as a read-only bit with a fixed value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [40:39]

Reserved, RES0.

D128, bit [38]

When FEAT_D128 is implemented:

Enables VMSAv9-128 translation system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D128	Meaning
0b0	Translation system follows VMSAv8-64 translation process.
0b1	Translation system follows VMSAv9-128 translation process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AIE, bit [37]

When FEAT_AIE is implemented:

Enable Attribute Indexing Extension.

AIE	Meaning
0b0	Attribute Indexing Extension Disabled.
0b1	Attribute Indexing Extension Enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

POE, bit [36]

When FEAT_S1POE is implemented:

Enables Permission Overlay for EL3 accesses.

POE	Meaning
0b0	Permission overlay disabled for EL3 access in stage 1 of EL3 translation regime.
0b1	Permission overlay enabled for EL3 access in stage 1 of EL3 translation regime.

This bit is not permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PIE, bit [35]

When FEAT_S1PIE is implemented:

Enables usage of Indirect Permission Scheme.

PIE	Meaning
0b0	Direct permission model.
0b1	Indirect permission model.

This field is RES1 when TCR_EL3.D128 is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PnCH, bit [34]

When FEAT_THE is implemented:

Protected attribute enable. Enables use of bit[52] of the stage 1 translation table entries as the Protected bit, for translations using TTBR0_EL3.

PnCH	Meaning
0b0	For translations using TTBR0_EL3, bit[52] of each stage 1 translation table entry is not the Protected bit.
0b1	For translations using TTBR0_EL3, bit[52] of each stage 1 translation table entry is the Protected bit.

If bit[52] is used as the Protected bit, it is not used as the Contiguous bit.

This field is RES0 when TCR_EL3.D128 is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MTX, bit [33]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Extended memory tag checking.

This field controls address generation and Canonical tagging when EL3 is using AArch64 where the data address would be translated by tables pointed to by TTBR0_EL3.

This control has an effect regardless of whether stage 1 of the EL3 translation regime is enabled or not.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTX	Meaning
`0b0`	Canonical tagging is disabled. This control has no effect on address generation.
`0b1`	Canonical tagging is enabled. Bits[59:56] of a 64-bit VA hold a Logical Address Tag, and all of the following apply: • Bits[59:56] are treated as `0b0000` when checking if the address is out of range. • If FEAT_PAuth is implemented, bits[59:56] are not part of the PAC field. • A Canonical Tag Check operation is performed on Tag Checked memory accesses to a Canonically Tagged memory location.

For more information see Logical address tagging and Memory region tagging types.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DS, bit [32]

When FEAT_LPA2 is implemented and (FEAT_D128 is not implemented or TCR_EL3.D128 == '0'):

This field affects:

- Whether a 52-bit output address can be described by the translation tables of the 4KB or 16KB translation granules.
- The minimum value of TCR_EL3.T0SZ.
- How and where shareability for Block and Page descriptors are encoded.

DS	Meaning
`0b0`	Bits[49:48] of translation descriptors are RES0. Bits[9:8] in Block and Page descriptors encode shareability information in the SH[1:0] field. Bits[9:8] in Table descriptors are ignored by hardware. The minimum value of TCR_EL3.T0SZ is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. Output address[51:48] is `0b0000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DS	Meaning
0b1	Bits[49:48] of translation descriptors hold output address[49:48]. Bits[9:8] of table translation descriptors hold output address[51:50]. The shareability information of Block and Page descriptors for cacheable locations is determined by TCR_EL3.SH0. The minimum value of TCR_EL3.T0SZ is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. All calculations of the stage 1 base address are modified for tables of fewer than 8 entries so that the table is aligned to 64 bytes. Bits[5:2] of TTBR0_EL3 are used to hold bits[51:48] of the output address in all cases. **Note** As FEAT_LVA must be implemented if TCR_EL3.DS == 1, the minimum value of the TCR_EL3.T0SZ field is 12, as determined by that extension. For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 0b00. **Note** This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

The Effective value of this bit is 0.

Access to this field is RES0.

Bit [31]

Reserved, RES1.

TCMA, bit [30]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL3 when address [59:56] = 0b0000.

TCMA	Meaning
0b0	This control has no effect on the generation of Unchecked accesses.
0b1	All accesses are Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID, bit [29]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

TBID	Meaning
0b0	TCR_EL3.TBI applies to Instruction and Data accesses.
0b1	TCR_EL3.TBI applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL3.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

HWU62	Meaning
0b0	Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU61	Meaning
0b0	Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

HWU60	Meaning
0b0	Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

HWU59	Meaning
0b0	Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD, bit [24]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL3.

HPD	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled. **Note** In this case, bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE, and are no longer reserved, allowing them to be used by software.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL3.

HD	Meaning
0b0	Stage 1 hardware management of dirty state disabled.
0b1	Stage 1 hardware management of dirty state enabled.

When the Effective value of TCR_EL3.HA is 0, this field behaves as 0 for all purposes other than a direct read of the value of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HA, bit [21]

When FEAT_HAF is implemented:

Hardware Access flag update in stage 1 translations from EL3.

HA	Meaning
0b0	Stage 1 Access flag update disabled.
0b1	Stage 1 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI, bit [20]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL3 region, or ignored and used for tagged addresses.

TBI	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated at EL3 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL3. It has an effect whether the EL3 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL3.TBID is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL3.
- A exception taken to EL3.
- An exception return to EL3.

For more information, see ‘Address tagging’.

Note

This control detrmines the scope of address tagging. It never causes an exception to be generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PS	Meaning
0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.
0b111	56 bits, 64PB.

The values 0b110 and 0b111 represent different output address sizes depending on implementation choices and translation configuration.

The following table captures the output address size represented by the value 0b110:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1]	Represented OA size
Any	0b0101	Any	Any	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	0	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	1	52 bits, 4PB
VMSAv8-64	0b011x	64KB	N/A	52 bits, 4PB
VMSAv9-128	0b011x	Any	N/A	52 bits, 4PB

[1] This column represents the value of TCR_EL3.DS.

The following table captures the output address size represented by the value 0b111:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	Represented OA size
Any	0b0110	Any	OA size represented by 0b110
VMSAv8-64	0b0111	Any	OA size represented by 0b110
VMSAv9-128	0b0111	Any	56 bits, 64PB

If 52-bit PA is supported, and the translation table descriptors cannot express an OA larger than 48-bits, then bits[51:48] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL3.

If 56-bit PA is supported, and the translation table descriptors cannot express an OA larger than 52-bits, then bits[55:52] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL3.

If the output address size represented by this field is larger than the supported PA size expressed in ID_AA64MMFR0_EL1.PARange, then the output address size is treated as being the same as the supported PA size. Arm strongly recommends that software avoids configuring this field to a value representing an output address size larger than the supported PA size.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the TTBR0_EL3.

TG0	Meaning
0b00	4KB.
0b01	64KB.
0b10	16KB.

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL3.

SH0	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ORGN0	Meaning
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL3. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, TCR_EL3.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, TCR_EL3.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCR_EL3();
end;
```

```
MSR TCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().TCR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        TCR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.196 TCRMASK_EL1, Translation Control Masking Register (EL1)

The TCRMASK_EL1 characteristics are:

Purpose

Mask register to prevent updates of fields in TCR_EL1 on writes to TCR_EL1 or TCRALIAS_EL1.

Configuration

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCRMASK_EL1 are UNDEFINED.

Attributes

TCRMASK_EL1 is a 64-bit register.

Field descriptions

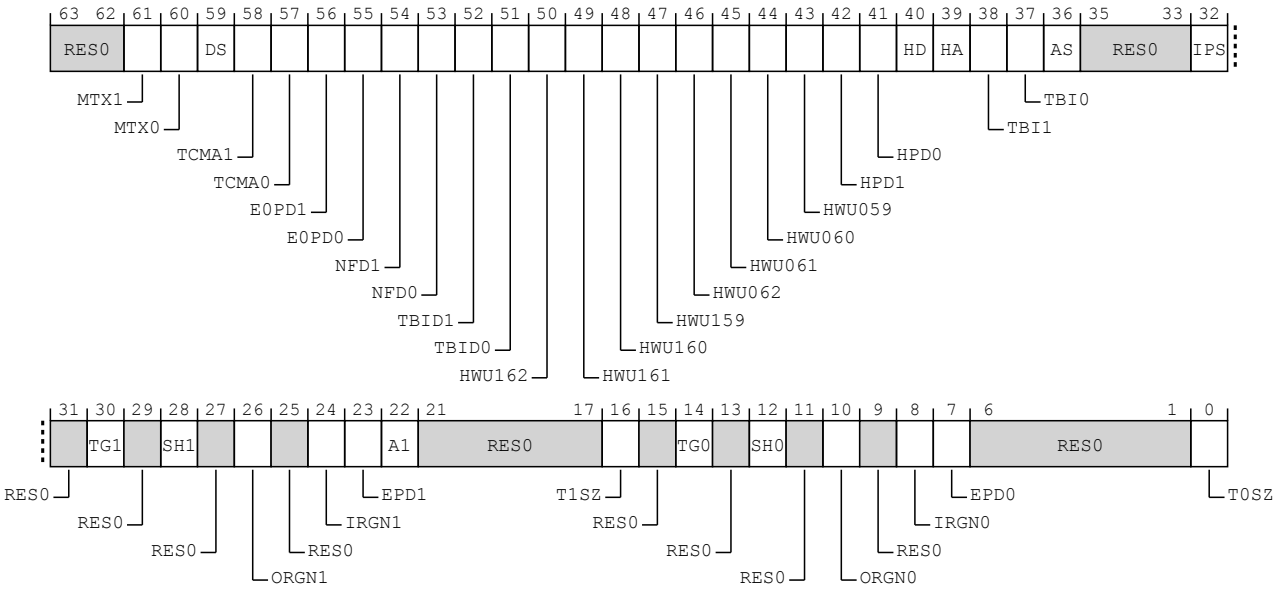

Bits [63:62]

Reserved, RES0.

MTX1, bit [61]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Mask bit for MTX1.

MTX1	Meaning
0b0	TCR_EL1.MTX1 is writable.
0b1	TCR_EL1.MTX1 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MTX0, bit [60]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Mask bit for MTX0.

MTX0	Meaning
`0b0`	TCR_EL1.MTX0 is writable.
`0b1`	TCR_EL1.MTX0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DS, bit [59]

When FEAT_LPA2 is implemented:

Mask bit for DS.

DS	Meaning
`0b0`	TCR_EL1.DS is writable.
`0b1`	TCR_EL1.DS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCMA1, bit [58]

When FEAT_MTE2 is implemented:

Mask bit for TCMA1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCMA1	Meaning
0b0	TCR_EL1.TCMA1 is writable.
0b1	TCR_EL1.TCMA1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCMA0, bit [57]

When FEAT_MTE2 is implemented:

Mask bit for TCMA0.

TCMA0	Meaning
0b0	TCR_EL1.TCMA0 is writable.
0b1	TCR_EL1.TCMA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

When FEAT_E0PD is implemented:

Mask bit for E0PD1.

E0PD1	Meaning
0b0	TCR_EL1.E0PD1 is writable.
0b1	TCR_EL1.E0PD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When FEAT_E0PD is implemented:

Mask bit for E0PD0.

E0PD0	Meaning
0b0	TCR_EL1.E0PD0 is writable.
0b1	TCR_EL1.E0PD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When FEAT_SVE is implemented:

Mask bit for NFD1.

NFD1	Meaning
0b0	TCR_EL1.NFD1 is writable.
0b1	TCR_EL1.NFD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD0, bit [53]

When FEAT_SVE is implemented:

Mask bit for NFD0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NFD0	Meaning
0b0	TCR_EL1.NFD0 is writable.
0b1	TCR_EL1.NFD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID1, bit [52]

When FEAT_PAuth is implemented:

Mask bit for TBID1.

TBID1	Meaning
0b0	TCR_EL1.TBID1 is writable.
0b1	TCR_EL1.TBID1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When FEAT_PAuth is implemented:

Mask bit for TBID0.

TBID0	Meaning
0b0	TCR_EL1.TBID0 is writable.
0b1	TCR_EL1.TBID0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU162, bit [50]

When FEAT_HPDS2 is implemented:

Mask bit for HWU162.

HWU162	Meaning
0b0	TCR_EL1.HWU162 is writable.
0b1	TCR_EL1.HWU162 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When FEAT_HPDS2 is implemented:

Mask bit for HWU161.

HWU161	Meaning
0b0	TCR_EL1.HWU161 is writable.
0b1	TCR_EL1.HWU161 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When FEAT_HPDS2 is implemented:

Mask bit for HWU160.

HWU160	Meaning
0b0	TCR_EL1.HWU160 is writable.
0b1	TCR_EL1.HWU160 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When FEAT_HPDS2 is implemented:

Mask bit for HWU159.

HWU159	Meaning
`0b0`	TCR_EL1.HWU159 is writable.
`0b1`	TCR_EL1.HWU159 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When FEAT_HPDS2 is implemented:

Mask bit for HWU062.

HWU062	Meaning
`0b0`	TCR_EL1.HWU062 is writable.
`0b1`	TCR_EL1.HWU062 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [45]

When FEAT_HPDS2 is implemented:

Mask bit for HWU061.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU061	Meaning
0b0	TCR_EL1.HWU061 is writable.
0b1	TCR_EL1.HWU061 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU060, bit [44]

When FEAT_HPDS2 is implemented:

Mask bit for HWU060.

HWU060	Meaning
0b0	TCR_EL1.HWU060 is writable.
0b1	TCR_EL1.HWU060 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU059, bit [43]

When FEAT_HPDS2 is implemented:

Mask bit for HWU059.

HWU059	Meaning
0b0	TCR_EL1.HWU059 is writable.
0b1	TCR_EL1.HWU059 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPD1, bit [42]

When FEAT_HPDS is implemented:

Mask bit for HPD1.

HPD1	Meaning
0b0	TCR_EL1.HPD1 is writable.
0b1	TCR_EL1.HPD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When FEAT_HPDS is implemented:

Mask bit for HPD0.

HPD0	Meaning
0b0	TCR_EL1.HPD0 is writable.
0b1	TCR_EL1.HPD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HD, bit [40]

When FEAT_HAFDBS is implemented:

Mask bit for HD.

HD	Meaning
0b0	TCR_EL1.HD is writable.
0b1	TCR_EL1.HD is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [39]

When FEAT_HAF is implemented:

Mask bit for HA.

HA	Meaning
`0b0`	TCR_EL1.HA is writable.
`0b1`	TCR_EL1.HA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

Mask bit for TBI1.

TBI1	Meaning
`0b0`	TCR_EL1.TBI1 is writable.
`0b1`	TCR_EL1.TBI1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TBI0, bit [37]

Mask bit for TBI0.

TBI0	Meaning
`0b0`	TCR_EL1.TBI0 is writable.
`0b1`	TCR_EL1.TBI0 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AS, bit [36]

Mask bit for AS.

AS	Meaning
`0b0`	TCR_EL1.AS is writable.
`0b1`	TCR_EL1.AS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [35:33]

Reserved, RES0.

IPS, bit [32]

Mask bit for IPS.

IPS	Meaning
`0b0`	TCR_EL1.IPS is writable.
`0b1`	TCR_EL1.IPS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RES0.

TG1, bit [30]

Mask bit for TG1.

TG1	Meaning
`0b0`	TCR_EL1.TG1 is writable.
`0b1`	TCR_EL1.TG1 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

SH1, bit [28]

Mask bit for SH1.

SH1	Meaning
`0b0`	TCR_EL1.SH1 is writable.
`0b1`	TCR_EL1.SH1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [27]

Reserved, RES0.

ORGN1, bit [26]

Mask bit for ORGN1.

ORGN1	Meaning
`0b0`	TCR_EL1.ORGN1 is writable.
`0b1`	TCR_EL1.ORGN1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [25]

Reserved, RES0.

IRGN1, bit [24]

Mask bit for IRGN1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRGN1	Meaning
0b0	TCR_EL1.IRGN1 is writable.
0b1	TCR_EL1.IRGN1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EPD1, bit [23]

Mask bit for EPD1.

EPD1	Meaning
0b0	TCR_EL1.EPD1 is writable.
0b1	TCR_EL1.EPD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

A1, bit [22]

Mask bit for A1.

A1	Meaning
0b0	TCR_EL1.A1 is writable.
0b1	TCR_EL1.A1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [21:17]

Reserved, RES0.

T1SZ, bit [16]

Mask bit for T1SZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1SZ	Meaning
0b0	TCR_EL1.T1SZ is writable.
0b1	TCR_EL1.T1SZ is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

TG0, bit [14]

Mask bit for TG0.

TG0	Meaning
0b0	TCR_EL1.TG0 is writable.
0b1	TCR_EL1.TG0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

SH0, bit [12]

Mask bit for SH0.

SH0	Meaning
0b0	TCR_EL1.SH0 is writable.
0b1	TCR_EL1.SH0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ORGN0, bit [10]

Mask bit for ORGN0.

ORGN0	Meaning
0b0	TCR_EL1.ORGN0 is writable.
0b1	TCR_EL1.ORGN0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

IRGN0, bit [8]

Mask bit for IRGN0.

IRGN0	Meaning
0b0	TCR_EL1.IRGN0 is writable.
0b1	TCR_EL1.IRGN0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EPD0, bit [7]

Mask bit for EPD0.

EPD0	Meaning
0b0	TCR_EL1.EPD0 is writable.
0b1	TCR_EL1.EPD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [6:1]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T0SZ, bit [0]

Mask bit for T0SZ.

T0SZ	Meaning
0b0	TCR_EL1.T0SZ is writable.
0b1	TCR_EL1.T0SZ is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing TCRMASK_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TCRMASK_EL1` or `TCRMASK_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCRMASK_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nTCRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x330);
    else
        X{64}(t) = TCRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TCRMASK_EL2();
    else
        X{64}(t) = TCRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCRMASK_EL1();
end;
```

```
MSR TCRMASK_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nTCRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x330) = X{64}(t);
    elsif !IsZero(TCRMASK_EL1()) then
        Undefined();
    else
        TCRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(TCRMASK_EL2()) then
            Undefined();
        else
            TCRMASK_EL2() = X{64}(t);
        end;
    else
        TCRMASK_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    TCRMASK_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TCRMASK_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x330);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TCRMASK_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = TCRMASK_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR TCRMASK_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x330) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            TCRMASK_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TCRMASK_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.197 TCRMASK_EL2, Translation Control Masking Register (EL2)

The TCRMASK_EL2 characteristics are:

Purpose

Mask register to prevent updates of fields in TCR_EL2 on writes.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SRMASK is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TCRMASK_EL2 are UNDEFINED.

Attributes

TCRMASK_EL2 is a 64-bit register.

Field descriptions

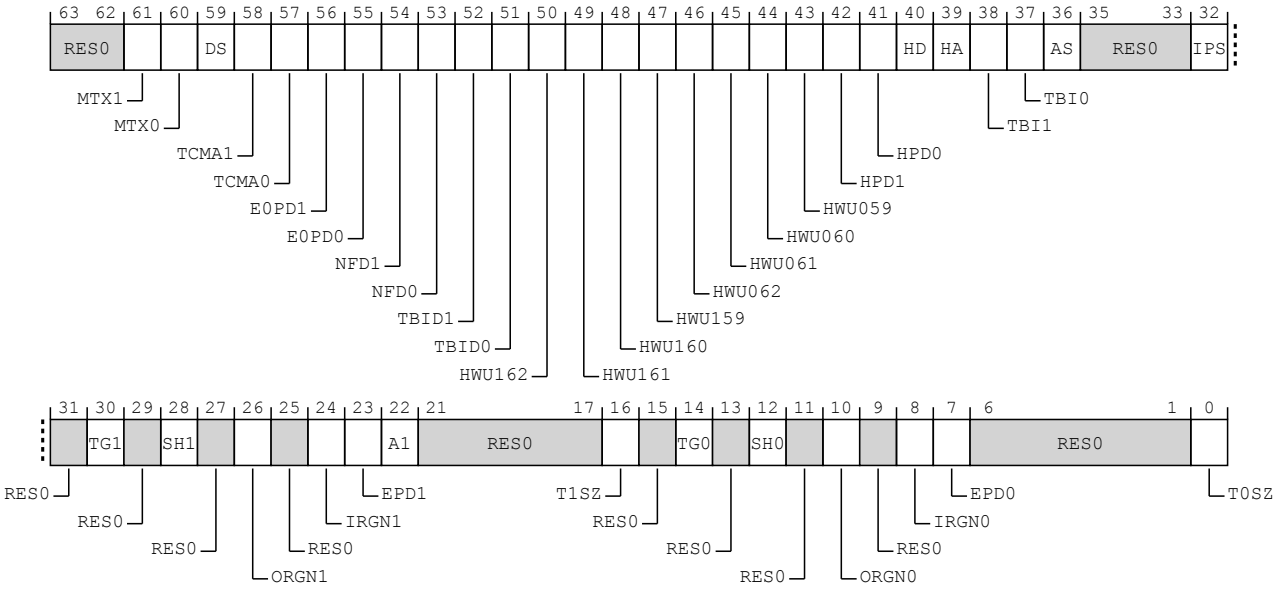

Bits [63:62]

Reserved, RES0.

MTX1, bit [61]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Mask bit for MTX1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTX1	Meaning
0b0	TCR_EL2.MTX1 is writable.
0b1	TCR_EL2.MTX1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MTX0, bit [60]

When FEAT_MTE_NO_ADDRESS_TAGS is implemented or FEAT_MTE_CANONICAL_TAGS is implemented:

Mask bit for MTX0.

MTX0	Meaning
0b0	TCR_EL2.MTX0 is writable.
0b1	TCR_EL2.MTX0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DS, bit [59]

When FEAT_LPA2 is implemented:

Mask bit for DS.

DS	Meaning
0b0	TCR_EL2.DS is writable.
0b1	TCR_EL2.DS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

TCMA1, bit [58]

When FEAT_MTE2 is implemented:

Mask bit for TCMA1.

TCMA1	Meaning
0b0	TCR_EL2.TCMA1 is writable.
0b1	TCR_EL2.TCMA1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCMA0, bit [57]

When FEAT_MTE2 is implemented:

Mask bit for TCMA0.

TCMA0	Meaning
0b0	TCR_EL2.TCMA0 is writable.
0b1	TCR_EL2.TCMA0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

When FEAT_E0PD is implemented:

Mask bit for E0PD1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E0PD1	Meaning
0b0	TCR_EL2.E0PD1 is writable.
0b1	TCR_EL2.E0PD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When FEAT_E0PD is implemented:

Mask bit for E0PD0.

E0PD0	Meaning
0b0	TCR_EL2.E0PD0 is writable.
0b1	TCR_EL2.E0PD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When FEAT_SVE is implemented:

Mask bit for NFD1.

NFD1	Meaning
0b0	TCR_EL2.NFD1 is writable.
0b1	TCR_EL2.NFD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NFD0, bit [53]

When FEAT_SVE is implemented:

Mask bit for NFD0.

NFD0	Meaning
0b0	TCR_EL2.NFD0 is writable.
0b1	TCR_EL2.NFD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID1, bit [52]

When FEAT_PAuth is implemented:

Mask bit for TBID1.

TBID1	Meaning
0b0	TCR_EL2.TBID1 is writable.
0b1	TCR_EL2.TBID1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When FEAT_PAuth is implemented:

Mask bit for TBID0.

TBID0	Meaning
0b0	TCR_EL2.TBID0 is writable.
0b1	TCR_EL2.TBID0 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When FEAT_HPDS2 is implemented:

Mask bit for HWU162.

HWU162	Meaning
`0b0`	TCR_EL2.HWU162 is writable.
`0b1`	TCR_EL2.HWU162 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When FEAT_HPDS2 is implemented:

Mask bit for HWU161.

HWU161	Meaning
`0b0`	TCR_EL2.HWU161 is writable.
`0b1`	TCR_EL2.HWU161 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When FEAT_HPDS2 is implemented:

Mask bit for HWU160.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU160	Meaning
0b0	TCR_EL2.HWU160 is writable.
0b1	TCR_EL2.HWU160 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When FEAT_HPDS2 is implemented:

Mask bit for HWU159.

HWU159	Meaning
0b0	TCR_EL2.HWU159 is writable.
0b1	TCR_EL2.HWU159 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When FEAT_HPDS2 is implemented:

Mask bit for HWU062.

HWU062	Meaning
0b0	TCR_EL2.HWU062 is writable.
0b1	TCR_EL2.HWU062 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU061, bit [45]

When FEAT_HPDS2 is implemented:

Mask bit for HWU061.

HWU061	Meaning
0b0	TCR_EL2.HWU061 is writable.
0b1	TCR_EL2.HWU061 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU060, bit [44]

When FEAT_HPDS2 is implemented:

Mask bit for HWU060.

HWU060	Meaning
0b0	TCR_EL2.HWU060 is writable.
0b1	TCR_EL2.HWU060 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU059, bit [43]

When FEAT_HPDS2 is implemented:

Mask bit for HWU059.

HWU059	Meaning
0b0	TCR_EL2.HWU059 is writable.
0b1	TCR_EL2.HWU059 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When FEAT_HPDS is implemented:

Mask bit for HPD1.

HPD1	Meaning
`0b0`	TCR_EL2.HPD1 is writable.
`0b1`	TCR_EL2.HPD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When FEAT_HPDS is implemented:

Mask bit for HPD0.

HPD0	Meaning
`0b0`	TCR_EL2.HPD0 is writable.
`0b1`	TCR_EL2.HPD0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HD, bit [40]

When FEAT_HAFDBS is implemented:

Mask bit for HD.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HD	Meaning
0b0	TCR_EL2.HD is writable.
0b1	TCR_EL2.HD is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [39]

When FEAT_HAF is implemented:

Mask bit for HA.

HA	Meaning
0b0	TCR_EL2.HA is writable.
0b1	TCR_EL2.HA is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

Mask bit for TBI1.

TBI1	Meaning
0b0	TCR_EL2.TBI1 is writable.
0b1	TCR_EL2.TBI1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TBI0, bit [37]

Mask bit for TBI0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TBI0	Meaning
0b0	TCR_EL2.TBI0 is writable.
0b1	TCR_EL2.TBI0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AS, bit [36]

Mask bit for AS.

AS	Meaning
0b0	TCR_EL2.AS is writable.
0b1	TCR_EL2.AS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [35:33]

Reserved, RES0.

IPS, bit [32]

Mask bit for IPS.

IPS	Meaning
0b0	TCR_EL2.IPS is writable.
0b1	TCR_EL2.IPS is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG1, bit [30]

Mask bit for TG1.

TG1	Meaning
0b0	TCR_EL2.TG1 is writable.
0b1	TCR_EL2.TG1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

SH1, bit [28]

Mask bit for SH1.

SH1	Meaning
0b0	TCR_EL2.SH1 is writable.
0b1	TCR_EL2.SH1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [27]

Reserved, RES0.

ORGN1, bit [26]

Mask bit for ORGN1.

ORGN1	Meaning
0b0	TCR_EL2.ORGN1 is writable.
0b1	TCR_EL2.ORGN1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [25]

Reserved, RES0.

IRGN1, bit [24]

Mask bit for IRGN1.

IRGN1	Meaning
0b0	TCR_EL2.IRGN1 is writable.
0b1	TCR_EL2.IRGN1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EPD1, bit [23]

Mask bit for EPD1.

EPD1	Meaning
0b0	TCR_EL2.EPD1 is writable.
0b1	TCR_EL2.EPD1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

A1, bit [22]

Mask bit for A1.

A1	Meaning
0b0	TCR_EL2.A1 is writable.
0b1	TCR_EL2.A1 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [21:17]

Reserved, RES0.

T1SZ, bit [16]

Mask bit for T1SZ.

T1SZ	Meaning
0b0	TCR_EL2.T1SZ is writable.
0b1	TCR_EL2.T1SZ is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

TG0, bit [14]

Mask bit for TG0.

TG0	Meaning
0b0	TCR_EL2.TG0 is writable.
0b1	TCR_EL2.TG0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

SH0, bit [12]

Mask bit for SH0.

SH0	Meaning
0b0	TCR_EL2.SH0 is writable.
0b1	TCR_EL2.SH0 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

ORGN0, bit [10]

Mask bit for ORGN0.

ORGN0	Meaning
`0b0`	TCR_EL2.ORGN0 is writable.
`0b1`	TCR_EL2.ORGN0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

IRGN0, bit [8]

Mask bit for IRGN0.

IRGN0	Meaning
`0b0`	TCR_EL2.IRGN0 is writable.
`0b1`	TCR_EL2.IRGN0 is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EPD0, bit [7]

Mask bit for EPD0.

EPD0	Meaning
`0b0`	TCR_EL2.EPD0 is writable.
`0b1`	TCR_EL2.EPD0 is not writable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [6:1]

Reserved, RES0.

T0SZ, bit [0]

Mask bit for T0SZ.

T0SZ	Meaning
`0b0`	TCR_EL2.T0SZ is writable.
`0b1`	TCR_EL2.T0SZ is not writable.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing TCRMASK_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TCRMASK_EL2` or `TCRMASK_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCRMASK_EL2
```

op0	op1	CRn	CRm	op2
`0b11`	`0b100`	`0b0010`	`0b0111`	`0b010`

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = TCRMASK_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCRMASK_EL2();
end;
```

MSR TCRMASK_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsZero(TCRMASK_EL2()) then
        Undefined();
    else
        TCRMASK_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCRMASK_EL2() = X{64}(t);
end;
```

MRS <Xt>, TCRMASK_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nTCRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x330);
    else
        X{64}(t) = TCRMASK_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TCRMASK_EL2();
    else
        X{64}(t) = TCRMASK_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TCRMASK_EL1();
end;
```

MSR TCRMASK_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0111	0b010

```
if !(IsFeatureImplemented(FEAT_SRMASK) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGWTR2_EL2().nTCRMASK_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SRMASKEn == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x330) = X{64}(t);
    elsif !IsZero(TCRMASK_EL1()) then
        Undefined();
    else
        TCRMASK_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().SRMASKEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        if !IsZero(TCRMASK_EL2()) then
            Undefined();
        else
            TCRMASK_EL2() = X{64}(t);
        end;
    else
        TCRMASK_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TCRMASK_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.198 TFSR_EL1, Tag Fault Status Register (EL1)

The TFSR_EL1 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring at EL1 that are not taken precisely.

Configuration

This register is present only when FEAT_MTE_ASYNC is implemented. Otherwise, direct accesses to TFSR_EL1 are UNDEFINED.

Attributes

TFSR_EL1 is a 64-bit register.

Field descriptions

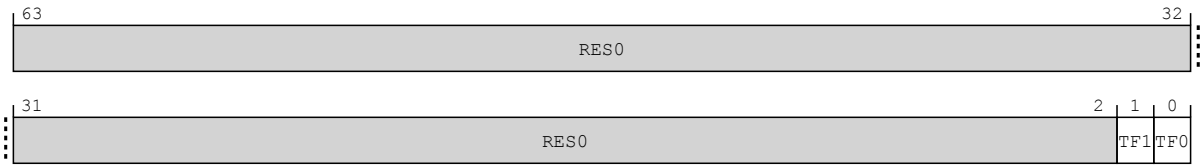

Bits [63:2]

Reserved, RES0.

TF1, bit [1]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '1' occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TF0, bit [0]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '0' occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TFSR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TFSR_EL1` or `TFSR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TFSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
        ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x190);
    else
        X{64}(t) = TFSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TFSR_EL2();
    else
        X{64}(t) = TFSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TFSR_EL1();
end;
```

```
MSR TFSR_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
        ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x190) = X{64}(t);
    else
        TFSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TFSR_EL2() = X{64}(t);
    else
        TFSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TFSR_EL1() = X{64}(t);
end;
```

MRS <Xt>, TFSR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
                ↪'1') then
                Undefined();
            elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA ==
                ↪'1') then
                AArch64_SystemAccessTrap(EL2, 0x18);
            elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                X{64}(t) = TFSR_EL1();
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
            ↪'1') then
            Undefined();
        elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TFSR_EL2();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = TFSR_EL2();
    end;
```

```
MSR TFSR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
            ↪'1') then
            Undefined();
        elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA ==
            ↪'1') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            TFSR_EL1() = X{64}(t);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
     ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TFSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TFSR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TFSR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x190);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) &&
         ↪SCR_EL3().ATA == '1') then
            Undefined();
        elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TFSR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if ELIsInHost(EL2) then
            X{64}(t) = TFSR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR TFSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x190) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) &&
            ↪SCR_EL3().ATA == '1') then
            Undefined();
        elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            TFSR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TFSR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.199 TFSR_EL2, Tag Fault Status Register (EL2)

The TFSR_EL2 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring at EL2 that are not taken precisely.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MTE_ASYNC is implemented. Otherwise, direct accesses to TFSR_EL2 are UNDEFINED.

Attributes

TFSR_EL2 is a 64-bit register.

Field descriptions

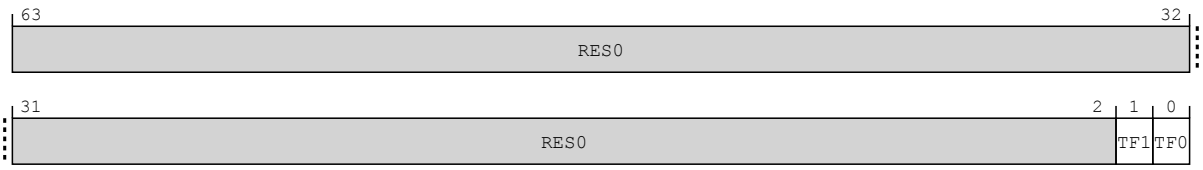

Bits [63:2]

Reserved, RES0.

TF1, bit [1]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '1' occurs.

When the Effective value of HCR_EL2.E2H is not 1, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TF0, bit [0]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '0' occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TFSR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TFSR_EL2` or `TFSR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TFSR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
            ↪'1') then
            Undefined();
        elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA ==
            ↪'1') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TFSR_EL1();
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TFSR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TFSR_EL2();
end;
```

```
MSR TFSR_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
            ↪'1') then
            Undefined();
        elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA ==
            ↪'1') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            TFSR_EL1() = X{64}(t);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TFSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TFSR_EL2() = X{64}(t);
end;
```

MRS <Xt>, TFSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
         ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x190);
    else
        X{64}(t) = TFSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
         ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TFSR_EL2();
    else
        X{64}(t) = TFSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TFSR_EL1();
end;
```

MSR TFSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
         ↪'1') then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
         ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x190) = X{64}(t);
    else
        TFSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
     ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TFSR_EL2() = X{64}(t);
    else
        TFSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TFSR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.200 TFSR_EL3, Tag Fault Status Register (EL3)

The TFSR_EL3 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring at EL3 that are not taken precisely.

Configuration

This register is present only when FEAT_MTE_ASYNC is implemented. Otherwise, direct accesses to TFSR_EL3 are UNDEFINED.

Attributes

TFSR_EL3 is a 64-bit register.

Field descriptions

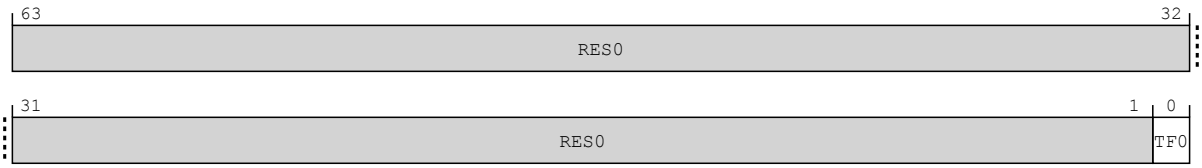

Bits [63:1]

Reserved, RES0.

TF0, bit [0]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '0' occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TFSR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TFSR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0110	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TFSR_EL3();
end;
```

```
MSR TFSR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    TFSR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.201 TFSRE0_EL1, Tag Fault Status Register (EL0).

The TFSRE0_EL1 characteristics are:

Purpose

Holds accumulated Tag Check Faults occurring at EL0 that are not taken precisely.

Configuration

This register is present only when FEAT_MTE_ASYNC is implemented. Otherwise, direct accesses to TFSRE0_EL1 are UNDEFINED.

Attributes

TFSRE0_EL1 is a 64-bit register.

Field descriptions

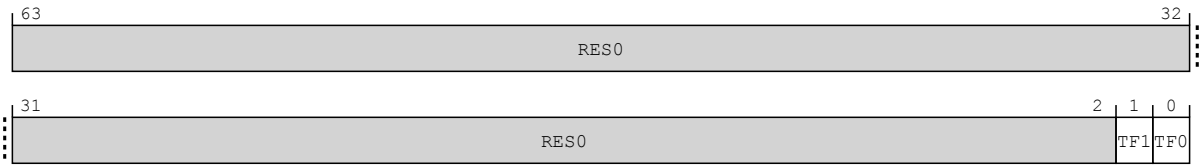

Bits [63:2]

Reserved, RES0.

TF1, bit [1]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '1' occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TF0, bit [0]

When FEAT_MTE_ASYNC is implemented:

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == '0' occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TFSRE0_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TFSRE0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
        ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TFSRE0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TFSRE0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TFSRE0_EL1();
end;
```

```
MSR TFSRE0_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_MTE_ASYNC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
        ↪'1') then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && !(IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().ATA == '1')
         ↪then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TFSRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA ==
         ↪'1') then
        Undefined();
    elsif HaveEL(EL3) && !(IsFeatureImplemented(FEAT_MTE2) && SCR_EL3().ATA == '1') then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TFSRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TFSRE0_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.202 TPIDR2_EL0, EL0 Read/Write Software Thread ID Register 2

The TPIDR2_EL0 characteristics are:

Purpose

Provides a location where SME-aware software executing at EL0 can store thread identifying information, for context management purposes.

The PE makes no use of this register.

Configuration

This register is present only when FEAT_SME is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TPIDR2_EL0 are UNDEFINED.

Attributes

TPIDR2_EL0 is a 64-bit register.

Field descriptions

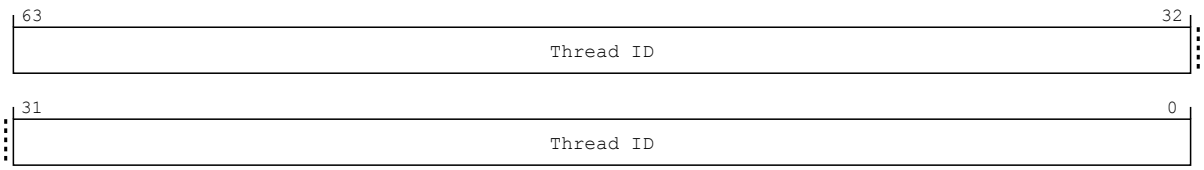

ThreadID, bits [63:0]

Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR2_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TPIDR2_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnTP2 == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().EnTP2 == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif ELIsInHost(EL0) && SCTLR_EL2().EnTP2 == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
             ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().nTPIDR2_EL0 == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().EnTP2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TPIDR2_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnTP2 == '0' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HFGRTR_EL2().nTPIDR2_EL0 == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && SCR_EL3().EnTP2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TPIDR2_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnTP2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().EnTP2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TPIDR2_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = TPIDR2_EL0();
    end;
```

MSR TPIDR2_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnTP2 == '0' then
        Undefined();
    elsif !ELIsInHost(EL0) && SCTLR_EL1().EnTP2 == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnTP2 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGWTR_EL2().nTPIDR2_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnTP2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TPIDR2_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnTP2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nTPIDR2_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().EnTP2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TPIDR2_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().EnTP2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().EnTP2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TPIDR2_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TPIDR2_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.203 TPIDR_EL0, EL0 Read/Write Software Thread ID Register

The TPIDR_EL0 characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configuration

AArch64 System register TPIDR_EL0 bits [31:0] are architecturally mapped to AArch32 System register TPIDRURW[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TPIDR_EL0 are UNDEFINED.

Attributes

TPIDR_EL0 is a 64-bit register.

Field descriptions

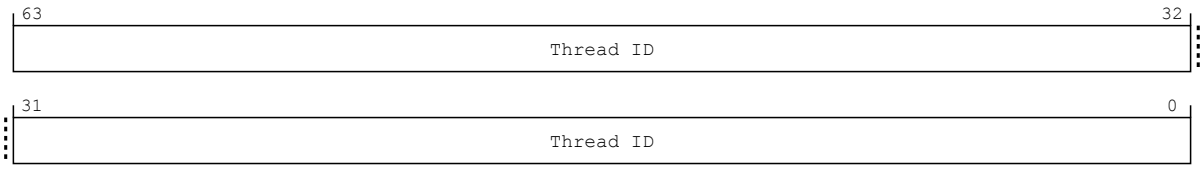

ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TPIDR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
        ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().TPIDR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = TPIDR_EL0();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGRTR_EL2().TPIDR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = TPIDR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TPIDR_EL0();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TPIDR_EL0();
end;
```

MSR TPIDR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
        ↪SCR_EL3().FGTEn == '1') && HFGWTR_EL2().TPIDR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        TPIDR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGWTR_EL2().TPIDR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        TPIDR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    TPIDR_EL0() = X{64}(t);
elsif PSTATE.EL == EL3 then
    TPIDR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.204 TPIDR_EL1, EL1 Software Thread ID Register

The TPIDR_EL1 characteristics are:

Purpose

Provides a location where software executing at EL1 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configuration

AArch64 System register TPIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TPIDRPRW[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TPIDR_EL1 are UNDEFINED.

Attributes

TPIDR_EL1 is a 64-bit register.

Field descriptions

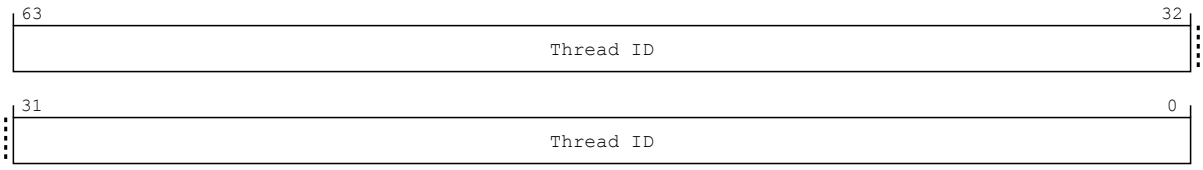

ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TPIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGRTR_EL2().TPIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = TPIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TPIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TPIDR_EL1();
end;
```

```
MSR TPIDR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGWTR_EL2().TPIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        TPIDR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    TPIDR_EL1() = X{64}(t);
elsif PSTATE.EL == EL3 then
    TPIDR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.205 TPIDR_EL2, EL2 Software Thread ID Register

The TPIDR_EL2 characteristics are:

Purpose

Provides a location where software executing at EL2 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register TPIDR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HTPIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TPIDR_EL2 are UNDEFINED.

Attributes

TPIDR_EL2 is a 64-bit register.

Field descriptions

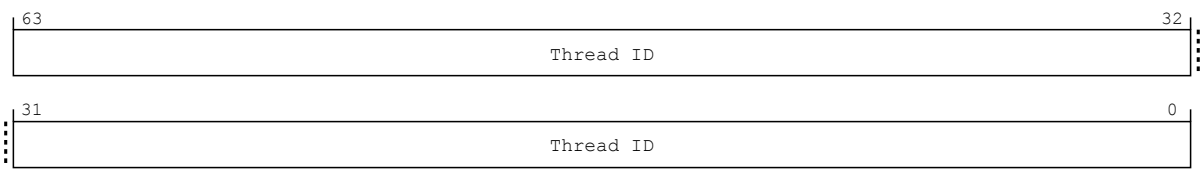

ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TPIDR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x090);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TPIDR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TPIDR_EL2();
end;
```

```
MSR TPIDR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x090) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    TPIDR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    TPIDR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.206 TPIDR_EL3, EL3 Software Thread ID Register

The TPIDR_EL3 characteristics are:

Purpose

Provides a location where software executing at EL3 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TPIDR_EL3 are UNDEFINED.

Attributes

TPIDR_EL3 is a 64-bit register.

Field descriptions

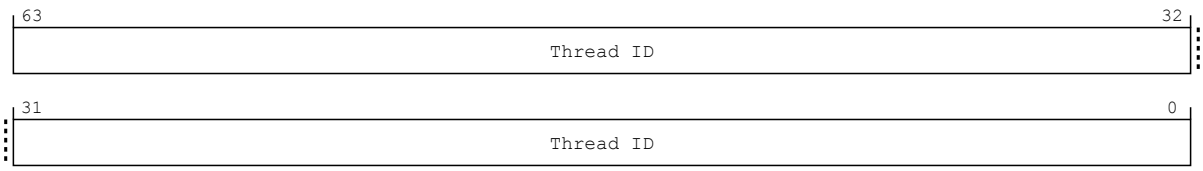

ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TPIDR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TPIDR_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TPIDR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().TPIDR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        TPIDR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.207 TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register

The TPIDRRO_EL0 characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configuration

AArch64 System register TPIDRRO_EL0 bits [31:0] are architecturally mapped to AArch32 System register TPIDRURO[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TPIDRRO_EL0 are UNDEFINED.

Attributes

TPIDRRO_EL0 is a 64-bit register.

Field descriptions

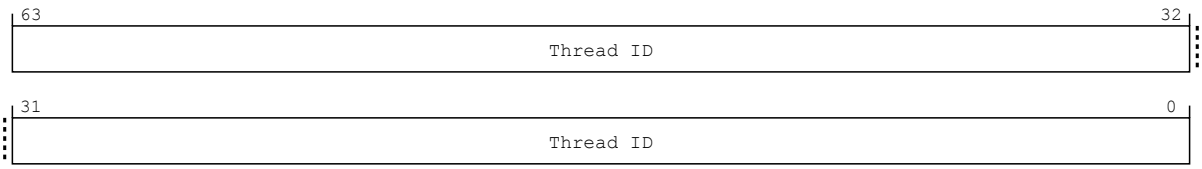

ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

Accessing TPIDRRO_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TPIDRRO_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
        ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().TPIDRRO_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = TPIDRRO_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGRTR_EL2().TPIDRRO_EL0 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = TPIDRRO_EL0();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TPIDRRO_EL0();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TPIDRRO_EL0();
end;
```

```
MSR TPIDRRO_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HFGWTR_EL2().TPIDRRO_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        TPIDRRO_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    TPIDRRO_EL0() = X{64}(t);
elsif PSTATE.EL == EL3 then
    TPIDRRO_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.208 TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL1&0 translation regime, and other information for this translation regime.

Configuration

TTBR0_EL1 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

AArch64 System register TTBR0_EL1 bits [63:0] are architecturally mapped to AArch32 System register TTBR0[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TTBR0_EL1 are UNDEFINED.

Attributes

TTBR0_EL1 is a:

- 128-bit register when FEAT_D128 is implemented and TCR2_EL1.D128 == ‘1’.
- 64-bit register when FEAT_D128 is not implemented or TCR2_EL1.D128 == ‘0’.

Field descriptions

When FEAT_D128 is implemented and TCR2_EL1.D128 == '1':

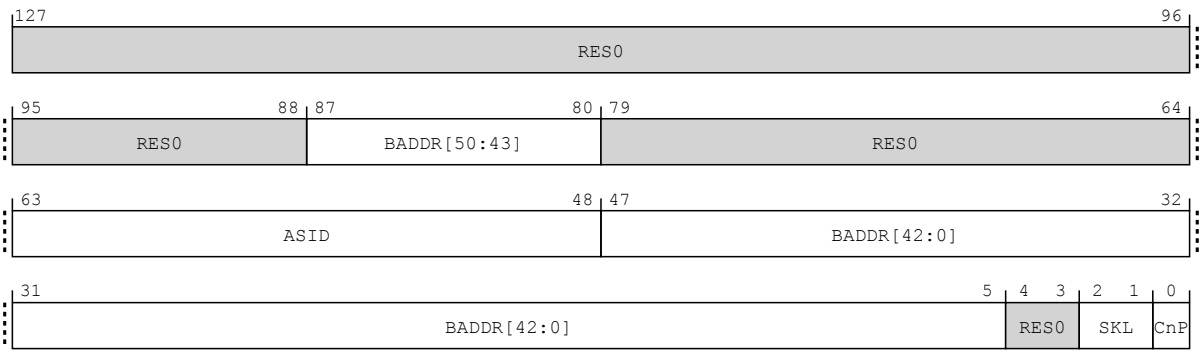

Bits [127:88]

Reserved, RES0.

BADDR, bits [87:80, 47:5]

Translation table base address:

- Bits A[55:x] of the stage 1 translation table base address bits are in register bits[87:80, 47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [79:64]

Reserved, RES0.

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level associated with translation table walks using TTBR0_EL1.

This determines the number of levels to be skipped from the regular start level of the stage 1 EL1&0 translation table walks using TTBR0_EL1.

SKL	Meaning
0b00	Skip 0 level from the regular start level.
0b01	Skip 1 level from the regular start level.
0b10	Skip 2 levels from the regular start level.
0b11	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1.

CnP	Meaning
0b0	The translation table entries pointed to by TTBR0_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR0_EL1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR0_EL1.CnP on those other PEs. • The value of the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b1	The translation table entries pointed to by TTBR0_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR0_EL1. • The translation tables relate to the same translation regime. • The ASID is the same as the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR0_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When FEAT_D128 is not implemented or TCR2_EL1.D128 == '0':

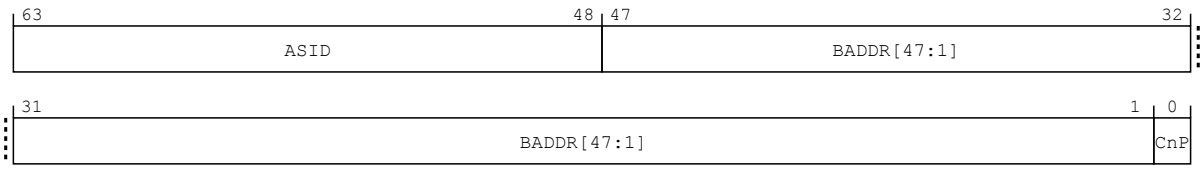

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BADDR[47:1], bits [47:1]

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the translation stage, and the translation granule size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The BADDR field represents a 52-bit address if any of the following apply:

- The value of TCR_EL1.IPS represents an OA size of 52 bits.
- The Effective value of TCR_EL1.DS is 1.

When TTBR0_EL1.BADDR represents a 52-bit addresses, all of the following apply:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL1.IPS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits A[55:52] of the stage 1 translation table base address are zero.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR0_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by TTBR0_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR0_EL1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR0_EL1.CnP on those other PEs. • The value of the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b1	The translation table entries pointed to by TTBR0_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR0_EL1. • The translation tables relate to the same translation regime. • The ASID is the same as the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR0_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR0_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TTBR0_EL1` or `TTBR0_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TTBR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TTBR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x200);
    else
        X{64}(t) = TTBR0_EL1()[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TTBR0_EL2()[63:0];
    else
        X{64}(t) = TTBR0_EL1()[63:0];
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR0_EL1()[63:0];
end;
```

MSR TTBR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TTBR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x200) = X{64}(t);
    else
        TTBR0_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TTBR0_EL2()[63:0] = X{64}(t);
    else
        TTBR0_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TTBR0_EL1()[63:0] = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TTBR0_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x200);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            X{64}(t) = TTBR0_EL1()[63:0];
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = TTBR0_EL1()[63:0];
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR TTBR0_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b000

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EffectiveHCR_EL2_NVx() == '101' then
            NVMem(0x200) = X{64}(t);
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            TTBR0_EL1()[63:0] = X{64}(t);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            TTBR0_EL1()[63:0] = X{64}(t);
        else
            Undefined();
        end;
    end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGRTR_EL2().TTBR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{128}(t, t2) = NVMem128(0x200);
    else
        X{128}(t, t2) = TTBR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif ELIsInHost(EL2) then
        X{128}(t, t2) = TTBR0_EL2();
    else
        X{128}(t, t2) = TTBR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = TTBR0_EL1();
end;
```

When FEAT_D128 is implemented MSRR TTBR0_EL1, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGWTR_EL2().TTBR0_EL1 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            NVMem128(0x200) = X{128}(t, t2);
        else
            TTBR0_EL1()[127:0] = X{128}(t, t2);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif ELIsInHost(EL2) then
            TTBR0_EL2()[127:0] = X{128}(t, t2);
        else
            TTBR0_EL1()[127:0] = X{128}(t, t2);
        end;
    elsif PSTATE.EL == EL3 then
        TTBR0_EL1()[127:0] = X{128}(t, t2);
    end;
```

When FEAT_D128 is implemented and FEAT_VHE is implemented MRRS <Xt>, <Xt+1>, TTBR0_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b000

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EffectiveHCR_EL2_NVx() == '101' then
            X{128}(t, t2) = NVMem128(0x200);
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x14);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
                Undefined();
            elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x14);
                end;
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            X{128}(t, t2) = TTBR0_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{128}(t, t2) = TTBR0_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_D128 is implemented and FEAT_VHE is implemented MSRR TTBR0_EL12, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem128(0x200) = X{128}(t, t2);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        else
            TTBR0_EL1()[127:0] = X{128}(t, t2);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TTBR0_EL1()[127:0] = X{128}(t, t2);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.209 TTBR0_EL2, Translation Table Base Register 0 (EL2)

The TTBR0_EL2 characteristics are:

Purpose

When the Effective value of HCR_EL2.E2H is not 1, holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL2 translation regime, and other information for this translation regime.

When the Effective value of HCR_EL2.E2H is 1, holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL2&0 translation regime, and other information for this translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

TTBR0_EL2 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

AArch64 System register TTBR0_EL2 bits [47:0] are architecturally mapped to AArch32 System register HTTBR[47:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TTBR0_EL2 are UNDEFINED.

Attributes

TTBR0_EL2 is a:

- 128-bit register when FEAT_D128 is implemented, TCR2_EL2.D128 == ‘1’, and EffectiveHCR_EL2_E2H() == ‘1’.
- 64-bit register when FEAT_D128 is not implemented or TCR2_EL2.D128 == ‘0’.

Field descriptions

When FEAT_D128 is implemented, TCR2_EL2.D128 == '1', and EffectiveHCR_EL2_E2H() == '1':

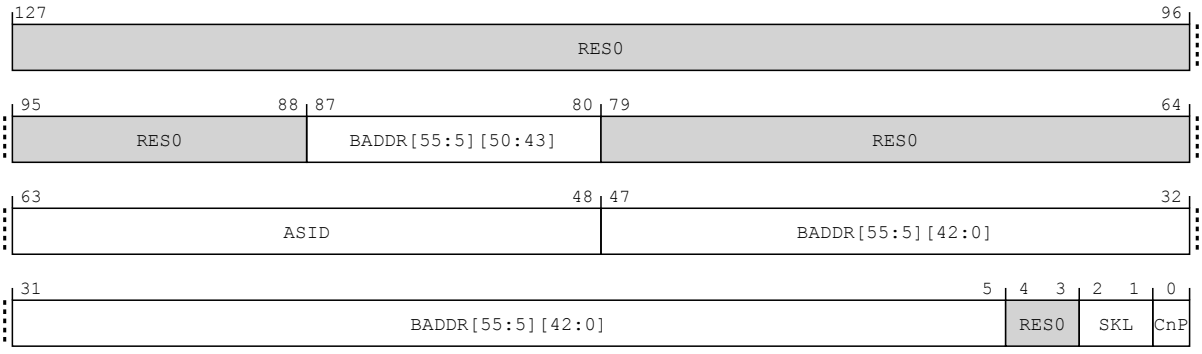

Bits [127:88]

Reserved, RES0.

BADDR[55:5], bits [87:80, 47:5]

Translation table base address:

- Bits A[55:x] of the stage 1 translation table base address bits are in register bits[87:80, 47:x].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [79:64]

Reserved, RES0.

ASID, bits [63:48]

When FEAT_VHE is implemented:

When the Effective value of HCR_EL2.E2H is 1, it holds an ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level associated with translation table walks using TTBR0_EL2.

This determines the number of levels to be skipped from the regular start level of the stage 1 EL2&0 translation table walks using TTBR0_EL2.

SKL	Meaning
0b00	Skip 0 level from the regular start level.
0b01	Skip 1 level from the regular start level.
0b10	Skip 2 levels from the regular start level.
0b11	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b0	The translation table entries pointed to by TTBR0_EL2 for the current translation regime, and ASID if applicable, are permitted to differ from corresponding entries for TTBR0_EL2 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR0_EL2.CnP on those other PEs. • When the current translation regime is the EL2&0 regime, the value of the current ASID.
0b1	The translation table entries pointed to by TTBR0_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR0_EL2. • The translation tables relate to the same translation regime. • If that translation regime is the EL2&0 regime, the ASID is the same as the current ASID.

This bit is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When FEAT_D128 is not implemented or TCR2_EL2.D128 == '0':

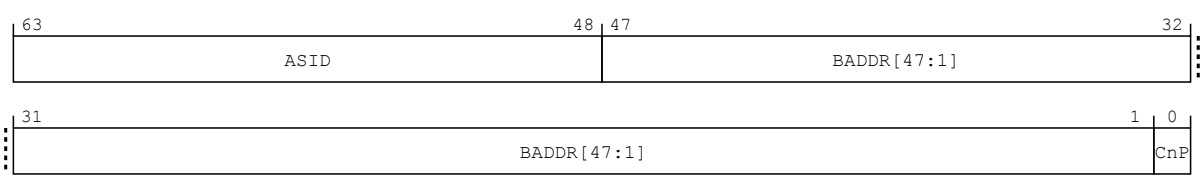

ASID, bits [63:48]

When FEAT_VHE is implemented:

When the Effective value of HCR_EL2.E2H is not 1, this field is RES0.

When the Effective value of HCR_EL2.E2H is 1, it holds an ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BADDR[47:1], bits [47:1]

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL2.T0SZ, the translation stage, and the translation granule size.

The BADDR field represents a 52-bit address if any of the following apply:

- The value of TCR_EL2.{I}PS represents an OA size of 52 bits.
- The Effective value of TCR_EL2.DS is 1.

When TTBR0_EL2.BADDR represents a 52-bit addresses, all of the following apply:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
The OA size specified by TCR_EL2.{I}PS is determined as follows:

- The value of TCR_EL2.PS when the Effective value of HCR_EL2.E2H is not 1.
- The value of TCR_EL2.IPS when the Effective value of HCR_EL2.E2H is 1.

For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL2.{I}PS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits A[55:52] of the stage 1 translation table base address are zero.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR0_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b0	The translation table entries pointed to by TTBR0_EL2 for the current translation regime, and ASID if applicable, are permitted to differ from corresponding entries for TTBR0_EL2 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR0_EL2.CnP on those other PEs. • When the current translation regime is the EL2&0 regime, the value of the current ASID.
0b1	The translation table entries pointed to by TTBR0_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR0_EL2. • The translation tables relate to the same translation regime. • If that translation regime is the EL2&0 regime, the ASID is the same as the current ASID.

This bit is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR0_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TTBR0_EL2` or `TTBR0_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TTBR0_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TTBR0_EL2()[63:0];
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR0_EL2()[63:0];
end;
```

MSR TTBR0_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    TTBR0_EL2()[63:0] = X{64}(t);
elsif PSTATE.EL == EL3 then
    TTBR0_EL2()[63:0] = X{64}(t);
end;
```

MRS <Xt>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TTBR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x200);
    else
        X{64}(t) = TTBR0_EL1()[63:0];
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TTBR0_EL2()[63:0];
    else
        X{64}(t) = TTBR0_EL1()[63:0];
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR0_EL1()[63:0];
end;
```

MSR TTBR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TTBR0_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x200) = X{64}(t);
    else
        TTBR0_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TTBR0_EL2()[63:0] = X{64}(t);
    else
        TTBR0_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TTBR0_EL1()[63:0] = X{64}(t);
end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, TTBR0_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        else
            X{128}(t, t2) = TTBR0_EL2();
        end;
    elsif PSTATE.EL == EL3 then
        X{128}(t, t2) = TTBR0_EL2();
    end;
```

When FEAT_D128 is implemented MSRR TTBR0_EL2, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        TTBR0_EL2()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    TTBR0_EL2()[127:0] = X{128}(t, t2);
end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HFGRTR_EL2().TTBR0_EL1 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{128}(t, t2) = NVMem128(0x200);
        else
            X{128}(t, t2) = TTBR0_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif ELIsInHost(EL2) then
            X{128}(t, t2) = TTBR0_EL2();
        else
            X{128}(t, t2) = TTBR0_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{128}(t, t2) = TTBR0_EL1();
    end;
```

When FEAT_D128 is implemented MSRR TTBR0_EL1, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif EL2Enabled() && HCR_EL2().TVM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HFGWTR_EL2().TTBR0_EL1 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem128(0x200) = X{128}(t, t2);
    else
        TTBR0_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif ELIsInHost(EL2) then
        TTBR0_EL2()[127:0] = X{128}(t, t2);
    else
        TTBR0_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    TTBR0_EL1()[127:0] = X{128}(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.210 TTBR0_EL3, Translation Table Base Register 0 (EL3)

The TTBR0_EL3 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL3 translation regime, and other information for this translation regime.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TTBR0_EL3 are UNDEFINED.

Attributes

TTBR0_EL3 is a 64-bit register.

Field descriptions

When FEAT_D128 is implemented and TCR_EL3.D128 == '1':

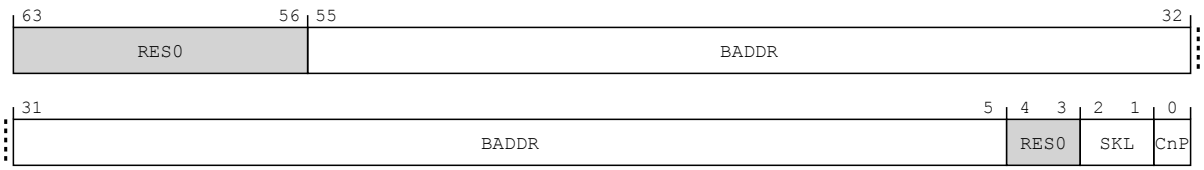

Bits [63:56]

Reserved, RES0.

BADDR, bits [55:5]

- Bits A[55:x] of the stage 1 translation table base address bits are in register bits[55:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level associated with translation table walks using TTBR0_EL3.

This determines the number of levels to be skipped from the regular start level of the stage 1 EL3 translation table walks using TTBR0_EL3.

SKL	Meaning
0b00	Skip 0 level from the regular start level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SKL	Meaning
0b01	Skip 1 level from the regular start level.
0b10	Skip 2 levels from the regular start level.
0b11	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL3 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1.

CnP	Meaning
0b0	The translation table entries pointed to by TTBR0_EL3, for the current translation regime, are permitted to differ from corresponding entries for TTBR0_EL3 for other PEs in the Inner Shareable domain. This is not affected by the value of TTBR0_EL3.CnP on those other PEs.
0b1	The translation table entries pointed to by TTBR0_EL3 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1 and the translation table entries are pointed to by TTBR0_EL3.

This bit is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL3.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL3s do not point to the same translation table entries the results of translations using TTBR0_EL3 are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is not implemented or TCR_EL3.D128 == '0':

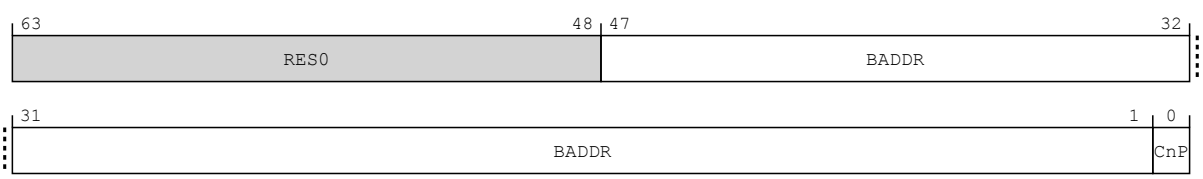

Bits [63:48]

Reserved, RES0.

BADDR, bits [47:1]

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL3.T0SZ, the translation stage, and the translation granule size.

The BADDR field represents a 52-bit address if any of the following apply:

- The value of TCR_EL3.PS represents an OA size of 52 bits.
- The Effective value of TCR_EL3.DS is 1.

When TTBR0_EL3.BADDR represents a 52-bit addresses, all of the following apply:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL3.PS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits A[55:52] of the stage 1 translation table base address are zero.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR0_EL3, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL3 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b0	The translation table entries pointed to by TTBR0_EL3, for the current translation regime, are permitted to differ from corresponding entries for TTBR0_EL3 for other PEs in the Inner Shareable domain. This is not affected by the value of TTBR0_EL3.CnP on those other PEs.
0b1	The translation table entries pointed to by TTBR0_EL3 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1 and the translation table entries are pointed to by TTBR0_EL3.

This bit is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL3.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL3s do not point to the same translation table entries the results of translations using TTBR0_EL3 are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR0_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TTBR0_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR0_EL3();
end;
```

```
MSR TTBR0_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().TTBR0_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        TTBR0_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.211 TTBR1_EL1, Translation Table Base Register 1 (EL1)

The TTBR1_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL1&0 stage 1 translation regime, and other information for this translation regime.

Configuration

TTBR1_EL1 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

AArch64 System register TTBR1_EL1 bits [63:0] are architecturally mapped to AArch32 System register TTBR1[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to TTBR1_EL1 are UNDEFINED.

Attributes

TTBR1_EL1 is a:

- 128-bit register when FEAT_D128 is implemented and TCR2_EL1.D128 == ‘1’.
- 64-bit register when FEAT_D128 is not implemented or TCR2_EL1.D128 == ‘0’.

Field descriptions

When FEAT_D128 is implemented and TCR2_EL1.D128 == '1':

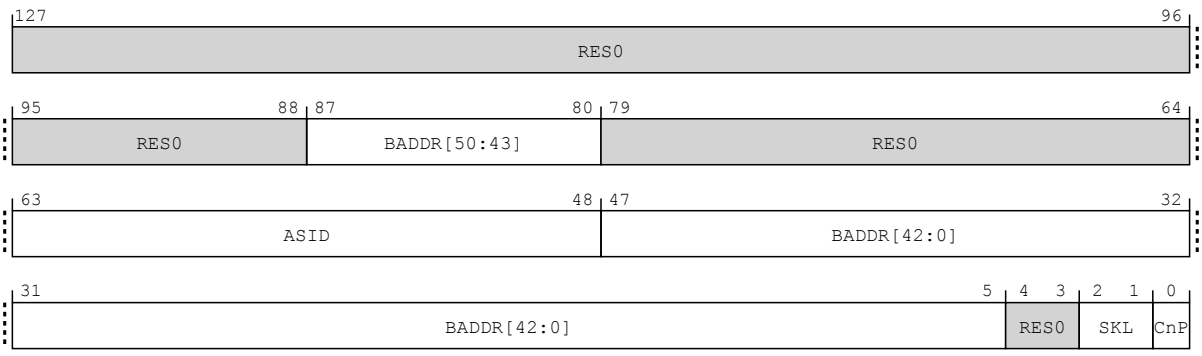

Bits [127:88]

Reserved, RES0.

BADDR, bits [87:80, 47:5]

Translation table base address:

- Bits A[55:x] of the stage 1 translation table base address bits are in register bits[87:80, 47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [79:64]

Reserved, RES0.

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level associated with translation table walks using TTBR1_EL1.

This determines the number of levels to be skipped from the regular start level of the stage 1 EL1&0 translation table walks using TTBR1_EL1.

SKL	Meaning
0b00	Skip 0 level from the regular start level.
0b01	Skip 1 level from the regular start level.
0b10	Skip 2 levels from the regular start level.
0b11	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1.

CnP	Meaning
0b0	The translation table entries pointed to by TTBR1_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR1_EL1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR1_EL1.CnP on those other PEs. • The value of the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b1	The translation table entries pointed to by TTBR1_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR1_EL1. • The translation tables relate to the same translation regime. • The ASID is the same as the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR1_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When FEAT_D128 is not implemented or TCR2_EL1.D128 == '0':

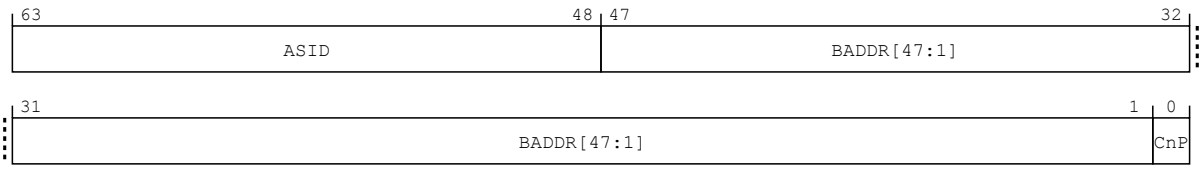

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BADDR[47:1], bits [47:1]

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T1SZ, the translation stage, and the translation granule size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The BADDR field represents a 52-bit address if any of the following apply:

- The value of TCR_EL1.IPS represents an OA size of 52 bits.
- The Effective value of TCR_EL1.DS is 1.

When TTBR1_EL1.BADDR represents a 52-bit addresses, all of the following apply:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL1.IPS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits A[55:52] of the stage 1 translation table base address are zero.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR1_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by TTBR1_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR1_EL1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR1_EL1.CnP on those other PEs. • The value of the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b1	The translation table entries pointed to by TTBR1_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR1_EL1. • The translation tables relate to the same translation regime. • The ASID is the same as the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR1_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR1_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TTBR1_EL1` or `TTBR1_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TTBR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TTBR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x210);
    else
        X{64}(t) = TTBR1_EL1()[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TTBR1_EL2()[63:0];
    else
        X{64}(t) = TTBR1_EL1()[63:0];
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR1_EL1()[63:0];
end;
```

MSR TTBR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TTBR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x210) = X{64}(t);
    else
        TTBR1_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TTBR1_EL2()[63:0] = X{64}(t);
    else
        TTBR1_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TTBR1_EL1()[63:0] = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TTBR1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x210);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            X{64}(t) = TTBR1_EL1()[63:0];
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = TTBR1_EL1()[63:0];
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR TTBR1_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x210) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TTBR1_EL1()[63:0] = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TTBR1_EL1()[63:0] = X{64}(t);
    else
        Undefined();
    end;
end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, TTBR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGRTR_EL2().TTBR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{128}(t, t2) = NVMem128(0x210);
    else
        X{128}(t, t2) = TTBR1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif ELIsInHost(EL2) then
        X{128}(t, t2) = TTBR1_EL2();
    else
        X{128}(t, t2) = TTBR1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = TTBR1_EL1();
end;
```

When FEAT_D128 is implemented MSRR TTBR1_EL1, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGWTR_EL2().TTBR1_EL1 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            NVMem128(0x210) = X{128}(t, t2);
        else
            TTBR1_EL1()[127:0] = X{128}(t, t2);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif ELIsInHost(EL2) then
            TTBR1_EL2()[127:0] = X{128}(t, t2);
        else
            TTBR1_EL1()[127:0] = X{128}(t, t2);
        end;
    elsif PSTATE.EL == EL3 then
        TTBR1_EL1()[127:0] = X{128}(t, t2);
    end;
```

When FEAT_D128 is implemented and FEAT_VHE is implemented MRRS <Xt>, <Xt+1>, TTBR1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{128}(t, t2) = NVMem128(0x210);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            X{128}(t, t2) = TTBR1_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{128}(t, t2) = TTBR1_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_D128 is implemented and FEAT_VHE is implemented MSRR TTBR1_EL12, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem128(0x210) = X{128}(t, t2);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        else
            TTBR1_EL1()[127:0] = X{128}(t, t2);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TTBR1_EL1()[127:0] = X{128}(t, t2);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.212 TTBR1_EL2, Translation Table Base Register 1 (EL2)

The TTBR1_EL2 characteristics are:

Purpose

When the Effective value of HCR_EL2.E2H is 1, holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL2&0 translation regime, and other information for this translation regime.

Note

When the Effective value of HCR_EL2.E2H is not 1, the contents of this register are ignored by the PE, except for a direct read or write of the register.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

TTBR1_EL2 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

This register is present only when FEAT_VHE is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TTBR1_EL2 are UNDEFINED.

Attributes

TTBR1_EL2 is a:

- 128-bit register when FEAT_D128 is implemented, TCR2_EL2.D128 == ‘1’, and EffectiveHCR_EL2_E2H() == ‘1’.
- 64-bit register when FEAT_D128 is not implemented or TCR2_EL2.D128 == ‘0’.

Field descriptions

When FEAT_D128 is implemented, TCR2_EL2.D128 == '1', and EffectiveHCR_EL2_E2H() == '1':

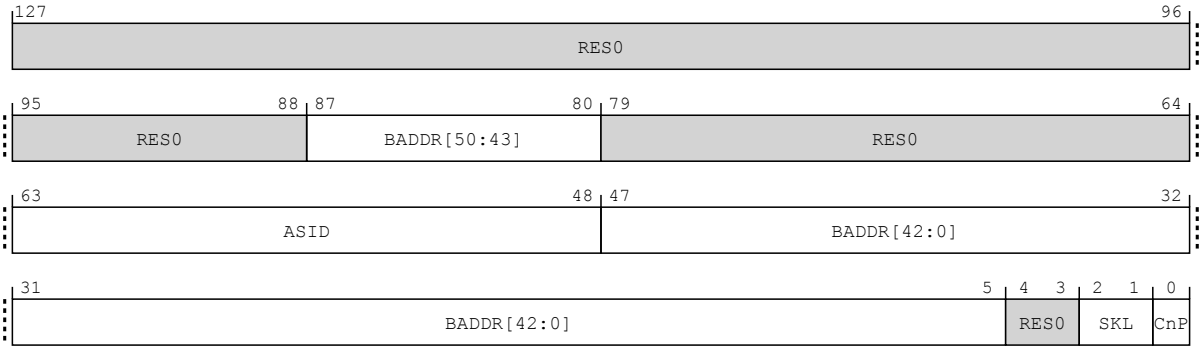

Bits [127:88]

Reserved, RES0.

BADDR, bits [87:80, 47:5]

Translation table base address:

- Bits A[55:x] of the stage 1 translation table base address bits are in register bits[87:80, 47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [79:64]

Reserved, RES0.

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level associated with translation table walks using TTBR1_EL2.

This determines the number of levels to be skipped from the regular start level of the stage 1 EL2&0 translation table walks using TTBR1_EL2.

SKL	Meaning
`0b00`	Skip 0 level from the regular start level.
`0b01`	Skip 1 level from the regular start level.
`0b10`	Skip 2 levels from the regular start level.
`0b11`	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b0	The translation table entries pointed to by TTBR1_EL2 for the current ASID are permitted to differ from corresponding entries for TTBR1_EL2 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR1_EL2.CnP on those other PEs. • The value of the current ASID.
0b1	The translation table entries pointed to by TTBR1_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR1_EL2. • The ASID is the same as the current ASID.

This bit is permitted to be cached in a TLB.

Note

- TTBR1_EL2 is accessible only when the Effective value of HCR_EL2.E2H is 1, meaning the current translation regime is the EL2&0 regime.
- If the value of the TTBR1_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When FEAT_D128 is not implemented or TCR2_EL2.D128 == '0':

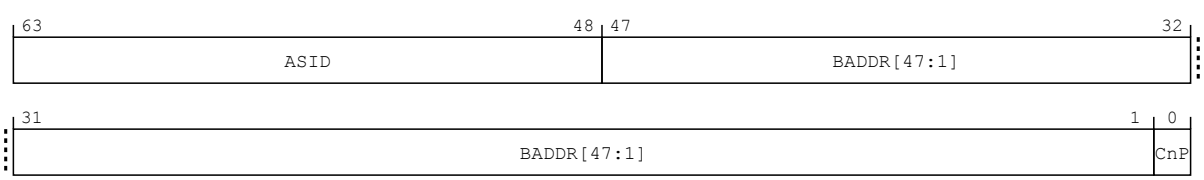

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BADDR[47:1], bits [47:1]

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL2.T1SZ, the translation stage, and the translation granule size.

The BADDR field represents a 52-bit address if any of the following apply:

- The value of TCR_EL2.{I}PS represents an OA size of 52 bits.
- The Effective value of TCR_EL2.DS is 1.

When TTBR1_EL2.BADDR represents a 52-bit addresses, all of the following apply:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
The OA size specified by TCR_EL2.{I}PS is determined as follows:

- The value of TCR_EL2.PS when the Effective value of HCR_EL2.E2H is not 1.
- The value of TCR_EL2.IPS when the Effective value of HCR_EL2.E2H is 1.

For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL2.{I}PS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits A[55:52] of the stage 1 translation table base address are zero.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR1_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b0	The translation table entries pointed to by TTBR1_EL2 for the current ASID are permitted to differ from corresponding entries for TTBR1_EL2 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR1_EL2.CnP on those other PEs. • The value of the current ASID.
0b1	The translation table entries pointed to by TTBR1_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR1_EL2. • The ASID is the same as the current ASID.

This bit is permitted to be cached in a TLB.

Note

- TTBR1_EL2 is accessible only when the Effective value of HCR_EL2.E2H is 1, meaning the current translation regime is the EL2&0 regime.
- If the value of the TTBR1_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR1_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TTBR1_EL2` or `TTBR1_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TTBR1_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_VHE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = TTBR1_EL2()[63:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR1_EL2()[63:0];
end;
```

MSR TTBR1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_VHE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    TTBR1_EL2()[63:0] = X{64}(t);
elsif PSTATE.EL == EL3 then
    TTBR1_EL2()[63:0] = X{64}(t);
end;
```

MRS <Xt>, TTBR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().TTBR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x210);
    else
        X{64}(t) = TTBR1_EL1()[63:0];
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = TTBR1_EL2()[63:0];
    else
        X{64}(t) = TTBR1_EL1()[63:0];
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TTBR1_EL1()[63:0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TTBR1_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2().TVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().TTBR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x210) = X{64}(t);
    else
        TTBR1_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        TTBR1_EL2()[63:0] = X{64}(t);
    else
        TTBR1_EL1()[63:0] = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TTBR1_EL1()[63:0] = X{64}(t);
end;
```

```
When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, TTBR1_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_VHE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{128}(t, t2) = TTBR1_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = TTBR1_EL2();
end;
```

When FEAT_D128 is implemented MSRR TTBR1_EL2, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_VHE) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        TTBR1_EL2()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    TTBR1_EL2()[127:0] = X{128}(t, t2);
end;
```

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, TTBR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TRVM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().TTBR1_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x14);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{128}(t, t2) = NVMem128(0x210);
        else
            X{128}(t, t2) = TTBR1_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif ELIsInHost(EL2) then
            X{128}(t, t2) = TTBR1_EL2();
        else
            X{128}(t, t2) = TTBR1_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{128}(t, t2) = TTBR1_EL1();
    end;
```

When FEAT_D128 is implemented MSRR TTBR1_EL1, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
            Undefined();
        elsif EL2Enabled() && HCR_EL2().TVM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HFGWTR_EL2().TTBR1_EL1 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().D128En == '0') then
            AArch64_SystemAccessTrap(EL2, 0x14);
        elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x14);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            NVMem128(0x210) = X{128}(t, t2);
        else
            TTBR1_EL1()[127:0] = X{128}(t, t2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    elsif ELIsInHost(EL2) then
        TTBR1_EL2()[127:0] = X{128}(t, t2);
    else
        TTBR1_EL1()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    TTBR1_EL1()[127:0] = X{128}(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.213 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL1.

Configuration

AArch64 System register VBAR_EL1 bits [31:0] are architecturally mapped to AArch32 System register VBAR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to VBAR_EL1 are UNDEFINED.

Attributes

VBAR_EL1 is a 64-bit register.

Field descriptions

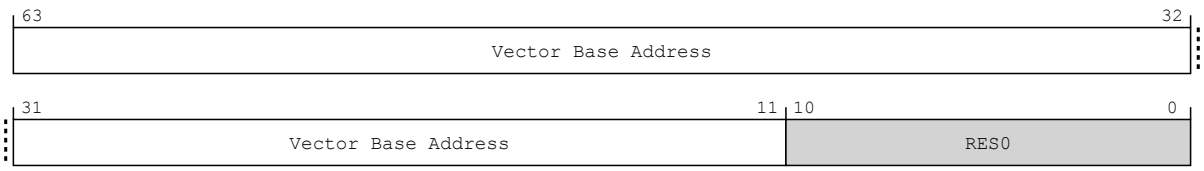

VBA, bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

Note

If the implementation supports FEAT_LVA3, then:

- If tagged addresses are not being used, bits [63:56] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

Otherwise:
If the implementation supports FEAT_LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation does not support FEAT_LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing VBAR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `VBAR_EL1` or `VBAR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VBAR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().VBAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x250);
    else
        X{64}(t) = VBAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = VBAR_EL2();
    else
        X{64}(t) = VBAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = VBAR_EL1();
end;
```

```
MSR VBAR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().VBAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x250) = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        VBAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        VBAR_EL2() = X{64}(t);
    else
        VBAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    VBAR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, VBAR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x250);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = VBAR_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = VBAR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR VBAR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x250) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        VBAR_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        VBAR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.214 VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VBAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HVBAR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to VBAR_EL2 are UNDEFINED.

Attributes

VBAR_EL2 is a 64-bit register.

Field descriptions

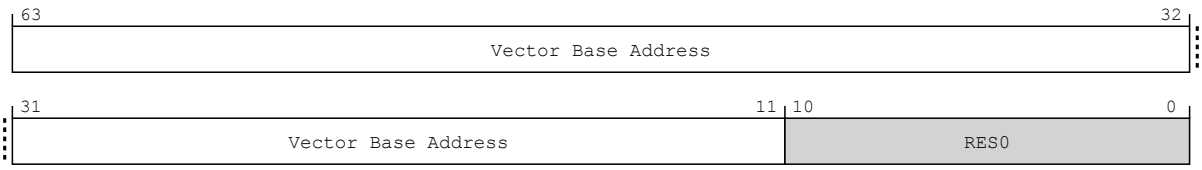

VBA, bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If FEAT_LVA3 is implemented:

- If the Effective value of HCR_EL2.E2H is 1:
  - If tagged addresses are not being used, bits [63:56] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If the Effective value of HCR_EL2.E2H is not 1:
  - If tagged addresses are not being used, bits [63:56] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

Otherwise :
If FEAT_LVA is implemented:

- If the Effective value of HCR_EL2.E2H is 1:
  - If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If the Effective value of HCR_EL2.E2H is not 1:
  - If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

If FEAT_LVA is not implemented:

- If the Effective value of HCR_EL2.E2H is 1:
  - If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If the Effective value of HCR_EL2.E2H is not 1:
  - If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing VBAR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `VBAR_EL2` or `VBAR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VBAR_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VBAR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VBAR_EL2();
end;
```

MSR VBAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VBAR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    VBAR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, VBAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGRTR_EL2().VBAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x250);
    else
        X{64}(t) = VBAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = VBAR_EL2();
    else
        X{64}(t) = VBAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = VBAR_EL1();
end;
```

When FEAT_VHE is implemented MSR VBAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '011' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGWTR_EL2().VBAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x250) = X{64}(t);
    else
        VBAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        VBAR_EL2() = X{64}(t);
    else
        VBAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    VBAR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.215 VBAR_EL3, Vector Base Address Register (EL3)

The VBAR_EL3 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL3.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to VBAR_EL3 are UNDEFINED.

Attributes

VBAR_EL3 is a 64-bit register.

Field descriptions

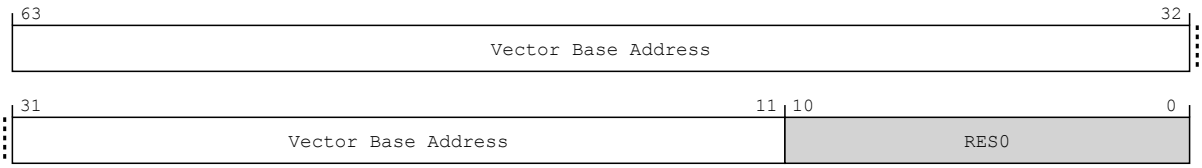

VBA, bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL3.

Note

If the implementation supports FEAT_LVA3, then:

- If tagged addresses are not being used, bits [63:56] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

Otherwise:
If the implementation supports FEAT_LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

If the implementation does not support FEAT_LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing VBAR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VBAR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VBAR_EL3();
end;
```

```
MSR VBAR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().VBAR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.216 VMECID_A_EL2, Alternate MECID for EL1&0 stage 2 translation regime

The VMECID_A_EL2 characteristics are:

Purpose

Alternate MECID for EL1&0 stage 2 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to VMECID_A_EL2 are UNDEFINED.

Attributes

VMECID_A_EL2 is a 64-bit register.

Field descriptions

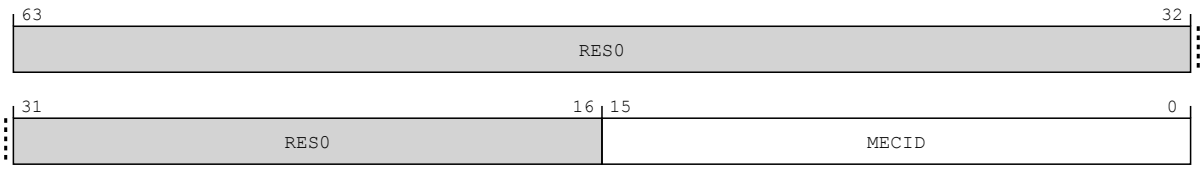

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VMECID_A_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VMECID_A_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1001	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = VMECID_A_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = VMECID_A_EL2();
end;
```

MSR VMECID_A_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1001	0b001

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        VMECID_A_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    VMECID_A_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.217 VMECID_P_EL2, Primary MECID for EL1&0 stage 2 translation regime

The VMECID_P_EL2 characteristics are:

Purpose

Primary MECID for EL1&0 stage 2 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_MEC is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to VMECID_P_EL2 are UNDEFINED.

Attributes

VMECID_P_EL2 is a 64-bit register.

Field descriptions

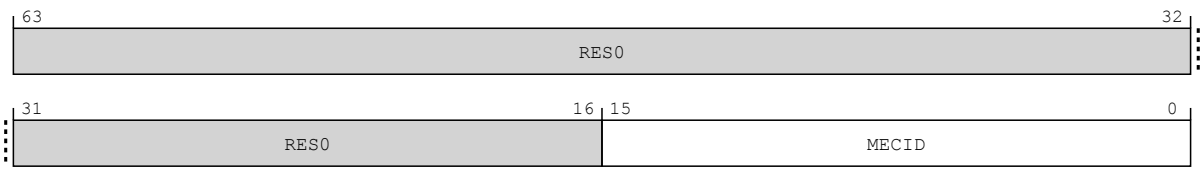

Bits [63:16]

Reserved, RES0.

MECID, bits [15:0]

If MECIDWidth is less than 16 bits, bits[15:MECIDWidth] are RES0.

Note

MECIDWidth is defined in MECIDR_EL2.MECIDWidthm1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VMECID_P_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VMECID_P_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = VMECID_P_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = VMECID_P_EL2();
end;
```

MSR VMECID_P_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b1001	0b000

```
if !(IsFeatureImplemented(FEAT_MEC) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Realm) then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().MECEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().MECEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        VMECID_P_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    VMECID_P_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.218 VMPIDR_EL2, Virtualization Multiprocessor ID Register

The VMPIDR_EL2 characteristics are:

Purpose

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by EL1 reads of MPIDR_EL1, which in a multiprocessor system, provides an additional PE identification system.

Configuration

If EL2 is not implemented, reads of this register return the value of the MPIDR_EL1 and writes to the register are ignored.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VMPIDR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VMPIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to VMPIDR_EL2 are UNDEFINED.

Attributes

VMPIDR_EL2 is a 64-bit register.

Field descriptions

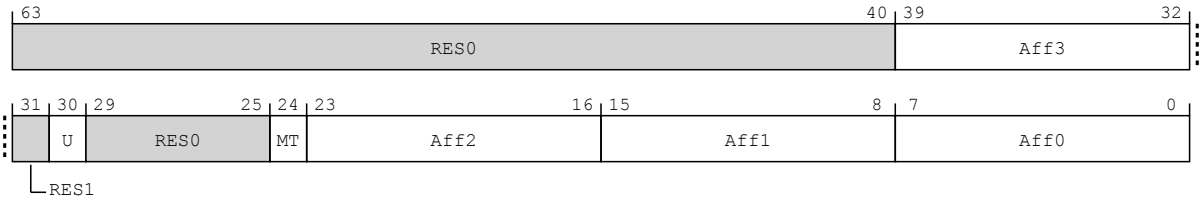

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of VMPIDR_EL2.Aff0 for more information.

Aff3 is not supported in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of VMPIDR_EL2.Aff0 for more information about affinity levels.

MT	Meaning
0b0	Performance of PEs at the lowest affinity level is largely independent.
0b1	Performance of PEs at the lowest affinity level is very interdependent.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Aff2, bits [23:16]

Affinity level 2. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Aff1, bits [15:8]

Affinity level 1. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Aff0, bits [7:0]

Affinity level 0.

The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VMPIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, VMPIDR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x050);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VMPIDR_EL2();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        X{64}(t) = MPIDR_EL1();
    else
        X{64}(t) = VMPIDR_EL2();
    end;
end;
```

```
MSR VMPIDR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x050) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VMPIDR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return;
    else
        VMPIDR_EL2() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, MPIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGRTR_EL2().MPIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() then
        X{64}(t) = VMPIDR_EL2();
    else
        X{64}(t) = MPIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = MPIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.219 VNCR_EL2, Virtual Nested Control Register

The VNCR_EL2 characteristics are:

Purpose

When FEAT_NV2 is implemented, holds the base address that is used to define the memory location that is accessed by transformed reads and writes of System registers.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_NV2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to VNCR_EL2 are UNDEFINED.

Attributes

VNCR_EL2 is a 64-bit register.

Field descriptions

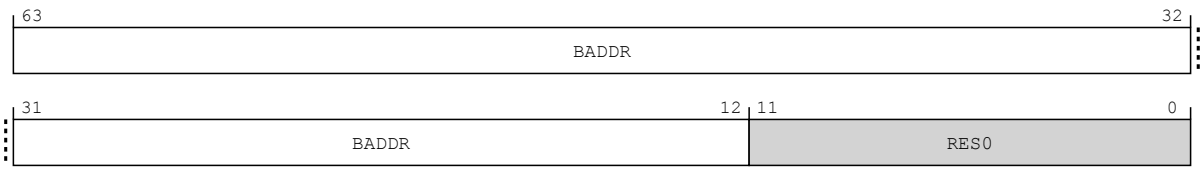

BADDR, bits [63:12]

Base Address. When a register read/write is transformed to be a Load or Store, the address of the load/store is to VNCR_EL2.BADDR:Offset<11:0>.

Bits[63:n] are RESS where n is one of the following:

- If FEAT_LVA3 is implemented, n is 57.
- Otherwise, if FEAT_LVA is implemented, n is 53.
- If FEAT_LVA is not implemented, n is 49.

If the bits marked as RESS do not all have the same value as bit[n-1], then there is a CONSTRAINED UNPREDICTABLE choice between:

- Generating a Translation fault when translating the transformed register read/write.
- The bits behave as the same value as bit[n-1] for all purposes other than reading back the register.
- The bits are treated as the same value as bit[n-1] for all purposes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

Accessing VNCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, VNCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0010	0b000

```
if !(IsFeatureImplemented(FEAT_NV2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x0B0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VNCR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VNCR_EL2();
end;
```

```
MSR VNCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0010	0b000

```
if !(IsFeatureImplemented(FEAT_NV2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x0B0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VNCR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    VNCR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.220 VPIDR_EL2, Virtualization Processor ID Register

The VPIDR_EL2 characteristics are:

Purpose

Holds the value of the Virtualization Processor ID. This is the value returned by EL1 reads of MIDR_EL1.

Configuration

If EL2 is not implemented, reads of this register return the value of the MIDR_EL1 and writes to the register are ignored.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VPIDR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VPIDR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to VPIDR_EL2 are UNDEFINED.

Attributes

VPIDR_EL2 is a 64-bit register.

Field descriptions

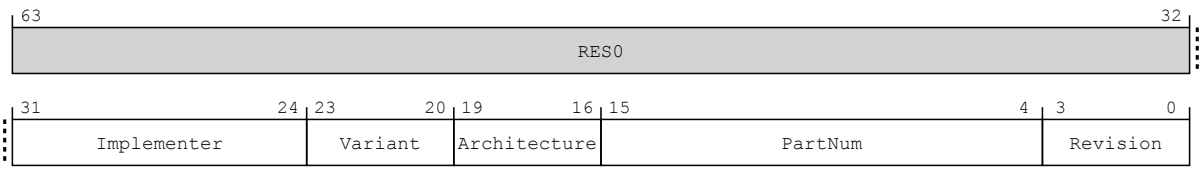

Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Implementer	Meaning
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Architecture, bits [19:16]

Architecture version.

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VPIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VPIDR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x088);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VPIDR_EL2();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        X{64}(t) = MIDR_EL1();
    else
        X{64}(t) = VPIDR_EL2();
    end;
end;
```

```
MSR VPIDR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x088) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    VPIDR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return;
    else
        VPIDR_EL2() = X{64}(t);
    end;
end;
```

MRS <Xt>, MIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HFGRTR_EL2().MIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() then
        X{64}(t) = VPIDR_EL2();
    else
        X{64}(t) = MIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = MIDR_EL1();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.221 VSTCR_EL2, Virtualization Secure Translation Control Register

The VSTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the Secure EL1&0 translation regime.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SEL2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to VSTCR_EL2 are UNDEFINED.

Attributes

VSTCR_EL2 is a 64-bit register.

Field descriptions

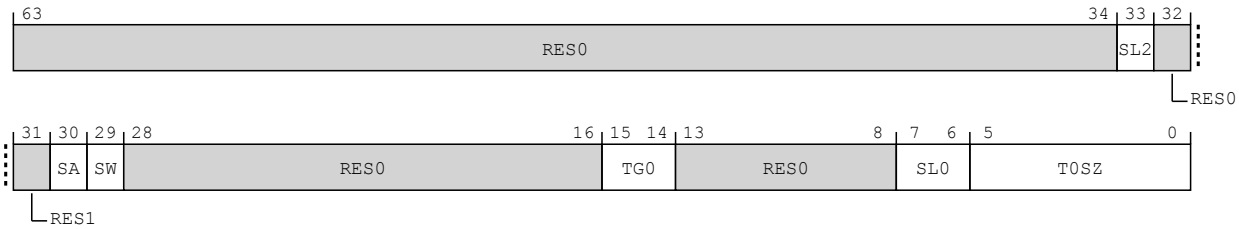

Any of the bits in VSTCR_EL2 are permitted to be cached in a TLB.

Bits [63:34]

Reserved, RES0.

SL2, bit [33]

When FEAT_LPA2 is implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

Starting level of the Secure stage 2 translation lookup controlled by VSTCR_EL2.

If VTCR_EL2.DS == 1, then VSTCR_EL2.SL2, in combination with VSTCR_EL2.SL0, gives encodings for the Secure stage 2 translation table walk initial lookup level.

If VTCR_EL2.DS == 0, then VSTCR_EL2.SL2 is RES0.

If the translation granule size is not 4KB, then this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SA, bit [30]

Secure stage 2 translation output address space.

SA	Meaning
0b0	All stage 2 translations for the Secure IPA space access the Secure PA space.
0b1	All stage 2 translations for the Secure IPA space access the Non-secure PA space.

When the value of VSTCR_EL2.SW is 1, this bit behaves as 1 for all purposes other than reading back the value of the bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SW, bit [29]

Secure stage 2 translation address space.

SW	Meaning
0b0	All stage 2 translation table walks for the Secure IPA space are to the Secure PA space.
0b1	All stage 2 translation table walks for the Secure IPA space are to the Non-secure PA space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [28:16]

Reserved, RES0.

TG0, bits [15:14]

Secure stage 2 granule size for VSTTBR_EL2.

TG0	Meaning
0b00	4KB.
0b01	64KB.
0b10	16KB.

Other values are reserved.

If FEAT_GTG is implemented, ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} indicate which granule sizes are supported for stage 2 translation.

If FEAT_GTG is not implemented, ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} indicate which granule sizes are supported.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [13:8]

Reserved, RES0.

SL0, bits [7:6]

When FEAT_TTST is implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

Starting level of the Secure stage 2 translation lookup, controlled by VSTCR_EL2. The meaning of this field depends on the value of VSTCR_EL2.TG0.

SL0	Meaning
`0b00`	If VSTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 2. • If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0, start at level 2. • If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 1, start at level -1. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.
`0b01`	If VSTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 1. • If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0, start at level 1. • If FEAT_LPA2 is implemented, the combination of VSTCR_EL2.SL0 == 01 and VSTCR_EL2.SL2 == 1 is reserved. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.
`0b10`	If VSTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 0. • If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0, start at level 0. • If FEAT_LPA2 is implemented, the combination of VSTCR_EL2.SL0 == 10 and VSTCR_EL2.SL2 == 1 is reserved. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.
`0b11`	If VSTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 3. • If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0, start at level 3. • If FEAT_LPA2 is implemented, the combination of VSTCR_EL2.SL0 == 11 and VSTCR_EL2.SL2 == 1 is reserved. If VSTCR_EL2.TG0 is `0b10` (16KB granule) and FEAT_LPA2 is implemented, start at level 0.

If this field is programmed to a value that is not consistent with the programming of VSTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TTST is not implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

Starting level of the Secure stage 2 translation lookup, controlled by VSTCR_EL2. The meaning of this field depends on the value of VSTCR_EL2.TG0.

SL0	Meaning
`0b00`	If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 2. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.
`0b01`	If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 1. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.
`0b10`	If VSTCR_EL2.TG0 is `0b00` (4KB granule), start at level 0. If VSTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VSTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by VSTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for this field depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

If this field is programmed to a value that is not consistent with the programming of SL0, then a stage 2 level 0 Translation fault is generated.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSTCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VSTCR_EL2
```

op0	op1	CRn	CRm	op2
`0b11`	`0b100`	`0b0010`	`0b0110`	`0b010`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x048);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        X{64}(t) = VSTCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        X{64}(t) = VSTCR_EL2();
    end;
end;
```

```
MSR VSTCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x048) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        VSTCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        VSTCR_EL2() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.222 VSTTBR_EL2, Virtualization Secure Translation Table Base Register

The VSTTBR_EL2 characteristics are:

Purpose

The base register for stage 2 translation tables to translate Secure IPAs in the Secure EL1&0 translation regime. Holds the base address of the translation table for the initial lookup for stage 2 of an address translation for a Secure IPA in the Secure EL1&0 translation regime, and other information for this translation stage.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_SEL2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to VSTTBR_EL2 are UNDEFINED.

Attributes

VSTTBR_EL2 is a 64-bit register.

Field descriptions

When FEAT_D128 is implemented and VTCR_EL2.D128 == '1':

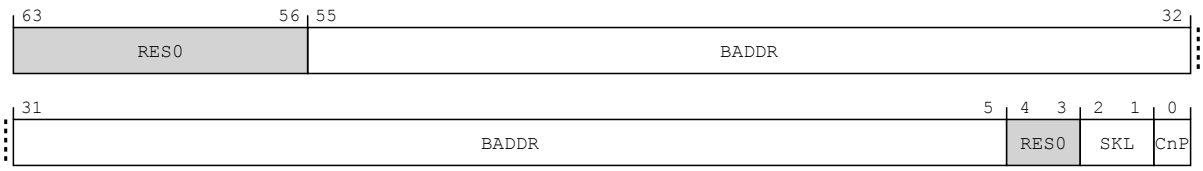

Bits [63:56]

Reserved, RES0.

BADDR, bits [55:5]

- Bits A[55:x] of the stage 2 translation table base address bits are in register bits[55:x].
- Bits A[(x-1):0] of the stage 2 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level. Skip Level determines the number of levels to be skipped from the regular start level of the Secure stage 2 translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SKL	Meaning
0b00	Skip 0 level from the regular start level.
0b01	Skip 1 level from the regular start level.
0b10	Skip 2 levels from the regular start level.
0b11	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

Common not Private, for stage 2 of the Secure EL1&0 translation regime. In an implementation that includes FEAT_TTCNP, indicates whether each entry that is pointed to by VSTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1.

CnP	Meaning
0b0	The translation table entries pointed to by VSTTBR_EL2 are permitted to differ from the entries for VSTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.
0b1	The translation table entries pointed to by VSTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This bit is permitted to be cached in a TLB.

Note

If the value of VSTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VSTTBR_EL2s do not point to the same translation table entries when using the current VMID, then the results of translations using VSTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_D128 is not implemented or VTCR_EL2.D128 == '0':

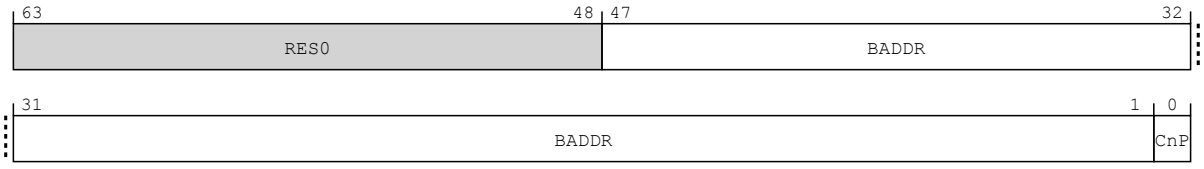

Bits [63:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BADDR, bits [47:1]

The BADDR field represents a 52-bit address if any of the following apply:

- The value of VTCR_EL2.PS represents an OA size of 52 bits.
- The Effective value of VTCR_EL2.DS is 1.

When VSTTBR_EL2.BADDR represents a 52-bit addresses, all of the following apply:

- Register bits[47:x] hold bits[47:x] of the stage 2 translation table base address, where x is determined by the size of the translation table at the start level.
- The smallest permitted value of x is 6.
- Register bits[5:2] hold bits[51:48] of the stage 2 translation table base address.
- Bits[x:0] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.

Otherwise, all of the following apply:

- Register bits[47:x] hold bits[47:x] of the stage 2 translation table base address.
- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits[51:48] of the stage 2 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of VTCR_EL2.PS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits [55:52] of the stage 2 translation table base address are zero.

If any VSTTBR_EL2[47:1] bit that is defined as RES0 has the value 1 when a translation table walk is performed using VSTTBR_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of VSTCR_EL2.T0SZ, the stage of translation, and the translation granule size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

Common not Private, for stage 2 of the Secure EL1&0 translation regime. In an implementation that includes FEAT_TTCNP, indicates whether each entry that is pointed to by VSTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b0	The translation table entries pointed to by VSTTBR_EL2 are permitted to differ from the entries for VSTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.
0b1	The translation table entries pointed to by VSTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This bit is permitted to be cached in a TLB.

Note

If the value of VSTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VSTTBR_EL2s do not point to the same translation table entries when using the current VMID, then the results of translations using VSTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSTTBR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VSTTBR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b000

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x030);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        X{64}(t) = VSTTBR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = VSTTBR_EL2();
    end;
end;
```

```
MSR VSTTBR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b000

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x030) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        VSTTBR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        VSTTBR_EL2() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.223 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the EL1&0 translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VTCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VTCR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to VTCR_EL2 are UNDEFINED.

Attributes

VTCR_EL2 is a 64-bit register.

Field descriptions

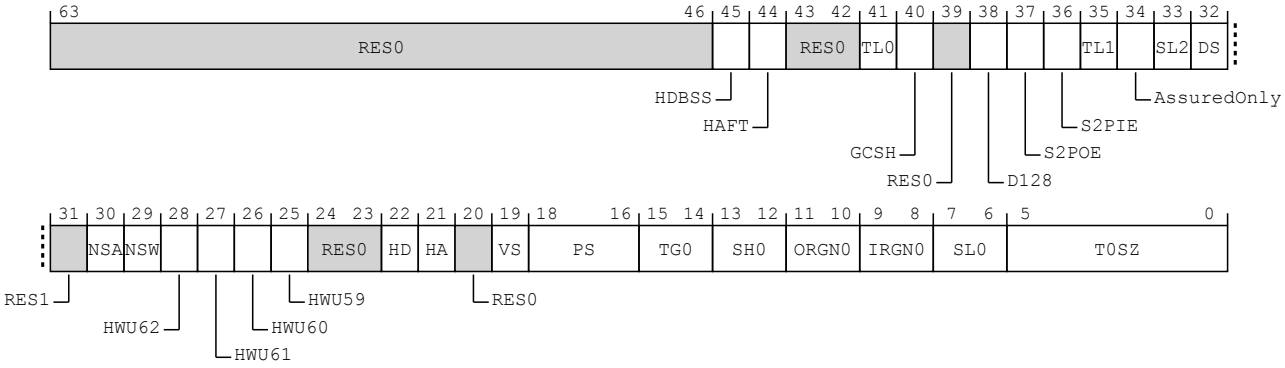

Unless stated otherwise, any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Bits [63:46]

Reserved, RES0.

HDBSS, bit [45]

When FEAT_HDBSS is implemented:

Enable use of HDBSS.

HDBSS	Meaning
0b0	Hardware tracking of Dirty state Structure is disabled.
0b1	Hardware tracking of Dirty state Structure is enabled.

If VTCR_EL2.{HA, HD} is not {1, 1}, the Effective value of this field is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If SCR_EL3.HDBSSEn is 0, then this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HAFT, bit [44]

When FEAT_HAFT is implemented:

Hardware managed Access Flag for Table descriptors.

Enables the Hardware managed Access Flag for Table descriptors.

HAFT	Meaning
0b0	Hardware managed Access Flag for Table descriptors is disabled.
0b1	Hardware managed Access Flag for Table descriptors is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [43:42]

Reserved, RES0.

TL0, bit [41]

When FEAT_THE is implemented:

Control bit to check for presence of MMU TopLevel0 permission attribute.

TL0	Meaning
0b0	This bit does not have any effect on stage 2 translations.
0b1	Enables MMU TopLevel0 permission attribute check for TTBR0_EL1 and TTBR1_EL1 translations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

GCSH, bit [40]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_THE is implemented and FEAT_GCS is implemented:

Assured stage 1 translations for Guarded Control Stacks. Enforces use of the AssuredOnly attribute in stage 2 for the memory accessed by privileged Guarded Control Stack data accesses.

GCSH	Meaning
0b0	For the memory accessed by privileged Guarded Control Stack data accesses, the AssuredOnly attribute in stage 2 is not required to be set.
0b1	For the memory accessed by privileged Guarded Control Stack data accesses, the AssuredOnly attribute in stage 2 is required to be set.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [39]

Reserved, RES0.

D128, bit [38]

When FEAT_D128 is implemented:

Enables VMSAv9-128 translation system for stage 2 translation.

D128	Meaning
0b0	Translation system follows VMSAv8-64 translation process.
0b1	Translation system follows VMSAv9-128 translation process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S2POE, bit [37]

When FEAT_S2POE is implemented:

Enable Permission Overlay. Enables permission overlay in stage 2 Permission model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S2POE	Meaning
0b0	Overlay disabled.
0b1	Overlay enabled.

This bit is not permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S2PIE, bit [36]

When FEAT_S2PIE is implemented:

Select Permission Model. Enables usage of permission indirection in stage 2 Permission model.

S2PIE	Meaning
0b0	Direct permission model.
0b1	Indirect permission model.

This field is RES1 when VTCR_EL2.D128 is set.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TL1, bit [35]

When FEAT_THE is implemented:

Control bit to check for presence of MMU TopLevel1 permission attribute.

TL1	Meaning
0b0	This bit does not have any effect on stage 2 translations.
0b1	Enables MMU TopLevel1 permission attribute check for TTBR0_EL1 and TTBR1_EL1 translations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [34]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_THE is implemented:

AssuredOnly attribute enable for VMSAv8-64. Configures use of bit[58] of the stage 2 translation table Block or Page descriptor.

AssuredOnly	Meaning
`0b0`	Bit[58] of each stage 2 translation Block or Page descriptor does not indicate AssuredOnly attribute.
`0b1`	Bit[58] of each stage 2 translation Block or Page descriptor indicates AssuredOnly attribute.

This field is RES0 when VTCR_EL2.D128 is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SL2, bit [33]

When FEAT_LPA2 is implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

Starting level of the stage 2 translation lookup controlled by VTCR_EL2.

If VTCR_EL2.DS == 1, then VTCR_EL2.SL2, in combination with VTCR_EL2.SL0, gives encodings for the stage 2 translation table walk initial lookup level.

If VTCR_EL2.DS == 0, then VTCR_EL2.SL2 is RES0.

If the translation granule size is not 4KB, then this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DS, bit [32]

When FEAT_LPA2 is implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

This field affects:

- Whether a 52-bit output address can be described by the translation tables of the 4KB or 16KB translation granules.
- The minimum value of VTCR_EL2.T0SZ and VSTCR_EL2.T0SZ.
- How and where shareability for Block and Page descriptors are encoded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DS	Meaning
0b0	Bits[49:48] of translation descriptors are RES0. Bits[9:8] in Block and Page descriptors encode shareability information in the SH[1:0] field. Bits[9:8] in Table descriptors are ignored by hardware. The minimum value of VTCR_EL2.T0SZ is 16. Any memory access using a smaller value generates a stage 2 level 0 translation table fault. The minimum value of VSTCR_EL2.T0SZ is 16. Any memory access using a smaller value generates a stage 2 level 0 translation table fault. Output address[51:48] is 0000.
0b1	Bits[49:48] of translation descriptors hold output address[49:48]. Bits[9:8] in translation descriptors hold output address[51:50]. The shareability information of Block and Page descriptors for cacheable locations is determined by VTCR_EL2.SH0. The minimum value of VTCR_EL2.T0SZ is 12. Any memory access using a smaller value generates a stage 2 level 0 translation table fault. The minimum value of VSTCR_EL2.T0SZ is 12. Any memory access using a smaller value generates a stage 2 level 0 translation table fault. **Note** As FEAT_LPA must be implemented if VTCR_EL2.DS == 1, the minimum values of VTCR_EL2.T0SZ and VSTCR_EL2.T0SZ are 12, as determined by that extension. For the TLBI range instructions affecting IPA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 00. **Note** This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [31]

Reserved, RES1.

NSA, bit [30]

When FEAT_SEL2 is implemented:

Non-secure stage 2 translation output address space for the Secure EL1&0 translation regime.

NSA	Meaning
0b0	All stage 2 translations for the Non-secure IPA space of the Secure EL1&0 translation regime access the Secure PA space.
0b1	All stage 2 translations for the Non-secure IPA space of the Secure EL1&0 translation regime access the Non-secure PA space.

This bit behaves as 1 for all purposes other than reading back the value of the bit when one of the following is true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The value of VTCR_EL2.NSW is 1.
- The value of VSTCR_EL2.SA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSW, bit [29]

When FEAT_SEL2 is implemented:

Non-secure stage 2 translation table address space for the Secure EL1&0 translation regime.

NSW	Meaning
0b0	All stage 2 translation table walks for the Non-secure IPA space of the Secure EL1&0 translation regime are to the Secure PA space.
0b1	All stage 2 translation table walks for the Non-secure IPA space of the Secure EL1&0 translation regime are to the Non-secure PA space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 2 translation table Block or Page entry.

HWU62	Meaning
0b0	Bit[62] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[62] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 2 translation table Block or Page entry.

HWU61	Meaning
0b0	Bit[61] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[61] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 2 translation table Block or Page entry.

HWU60	Meaning
0b0	Bit[60] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[60] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 2 translation table Block or Page entry.

HWU59	Meaning
0b0	Bit[59] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU59	Meaning
0b1	Bit[59] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:23]

Reserved, RES0.

HD, bit [22]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 2 translations when EL2 is enabled in the current Security state.

HD	Meaning
0b0	Stage 2 hardware management of dirty state disabled.
0b1	Stage 2 hardware management of dirty state enabled.

When the Effective value of VTCR_EL2.HA is 0, this field behaves as 0 for all purposes other than a direct read of the value of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [21]

When FEAT_HAF is implemented:

Hardware Access flag update in stage 2 translations when EL2 is enabled in the current Security state.

HA	Meaning
0b0	Stage 2 Access flag update disabled.
0b1	Stage 2 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [20]

Reserved, RES0.

VS, bit [19]

When FEAT_VMID16 is implemented:

VMID Size.

VS	Meaning
0b0	8-bit VMID. The upper 8 bits of VTTBR_EL2 are ignored by the hardware, and treated as if they are all zeros, for every purpose except when reading back the register.
0b1	16-bit VMID. The upper 8 bits of VTTBR_EL2 are used for allocation and matching in the TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PS, bits [18:16]

Physical address Size for the second stage of translation.

PS	Meaning
0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.
0b111	56 bits, 64PB.

The values 0b110 and 0b111 represent different output address sizes depending on implementation choices and translation configuration.

The following table captures the output address size represented by the value 0b110:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1]	Represented OA size
Any	0b0101	Any	Any	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	0	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	1	52 bits, 4PB

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1]	Represented OA size
VMSAv8-64	`0b011x`	64KB	N/A	52 bits, 4PB
VMSAv9-128	`0b011x`	Any	N/A	52 bits, 4PB

[1] This column represents the value of VTCR_EL2.DS.

The following table captures the output address size represented by the value `0b111`:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	Represented OA size
Any	`0b0110`	Any	OA size represented by `0b110`
VMSAv8-64	`0b0111`	Any	OA size represented by `0b110`
VMSAv9-128	`0b0111`	Any	56 bits, 64PB

If 52-bit PA is supported, and the translation table descriptors cannot express an OA larger than 48-bits, then bits[51:48] of every translation table base address are treated as `0b0000` for the stage of translation controlled by VTCR_EL2.

If 56-bit PA is supported, and the translation table descriptors cannot express an OA larger than 52-bits, then bits[55:52] of every translation table base address are treated as `0b0000` for the stage of translation controlled by VTCR_EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the VTTBR_EL2.

TG0	Meaning
`0b00`	4KB.
`0b01`	64KB.
`0b10`	16KB.

Other values are reserved.

If FEAT_GTG is implemented, ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} indicate which granule sizes are supported for stage 2 translation.

If FEAT_GTG is not implemented, ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} indicate which granule sizes are supported.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

SH0	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SL0, bits [7:6]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TTST is implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

Starting level of the stage 2 translation lookup, controlled by VTCR_EL2. The meaning of this field depends on the value of VTCR_EL2.TG0.

SL0	Meaning
`0b00`	If VTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 2. • If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0, start at level 2. • If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 1, start at level -1. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 3.
`0b01`	If VTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 1. • If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0, start at level 1. • If FEAT_LPA2 is implemented, the combination of VTCR_EL2.SL0 == 01 and VTCR_EL2.SL2 == 1 is reserved. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 2.
`0b10`	If VTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 0. • If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0, start at level 0. • If FEAT_LPA2 is implemented, the combination of VTCR_EL2.SL0 == 10 and VTCR_EL2.SL2 == 1 is reserved. If VTCR_EL2.TG0 is `0b10` (16KB granule) or `0b01` (64KB granule), start at level 1.
`0b11`	If VTCR_EL2.TG0 is `0b00` (4KB granule): • If FEAT_LPA2 is not implemented, start at level 3. • If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0, start at level 3. • If FEAT_LPA2 is implemented, the combination of VTCR_EL2.SL0 == 11 and VTCR_EL2.SL2 == 1 is reserved. If VTCR_EL2.TG0 is `0b10` (16KB granule) and FEAT_LPA2 is implemented, start at level 0.

If this field is programmed to a value that is not consistent with the programming of VTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TTST is not implemented and (FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'):

Starting level of the stage 2 translation lookup, controlled by VTCR_EL2. The meaning of this field depends on the value of VTCR_EL2.TG0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SL0	Meaning
0b00	If VTCR_EL2.TG0 is 0b00 (4KB granule), start at level 2. If VTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 3.
0b01	If VTCR_EL2.TG0 is 0b00 (4KB granule), start at level 1. If VTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 2.
0b10	If VTCR_EL2.TG0 is 0b00 (4KB granule), start at level 0. If VTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 1.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by VTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in 'The AArch64 Virtual Memory System Architecture'.

If this field is programmed to a value that is not consistent with the programming of SL0, then a stage 2 level 0 Translation fault is generated.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented, VTCR_EL2.DS is 1, and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented, VTCR_EL2.DS is 1, and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VTCR_EL2

Unless stated otherwise, any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VTCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x040);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VTCR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VTCR_EL2();
end;
```

MSR VTCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x040) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VTCR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    VTCR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.224 VTTBR_EL2, Virtualization Translation Table Base Register

The VTTBR_EL2 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the EL1&0 translation regime, and other information for this translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

VTTBR_EL2 is a 128-bit register that can also be accessed as a 64-bit value. If it is accessed as a 64-bit register, accesses read and write bits [63:0] and do not modify bits [127:64].

AArch64 System register VTTBR_EL2 bits [63:0] are architecturally mapped to AArch32 System register VTTBR[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to VTTBR_EL2 are UNDEFINED.

Attributes

VTTBR_EL2 is a:

- 128-bit register when FEAT_D128 is implemented and VTCR_EL2.D128 == '1'.
- 64-bit register when FEAT_D128 is not implemented or VTCR_EL2.D128 == '0'.

Field descriptions

When FEAT_D128 is implemented and VTCR_EL2.D128 == '1':

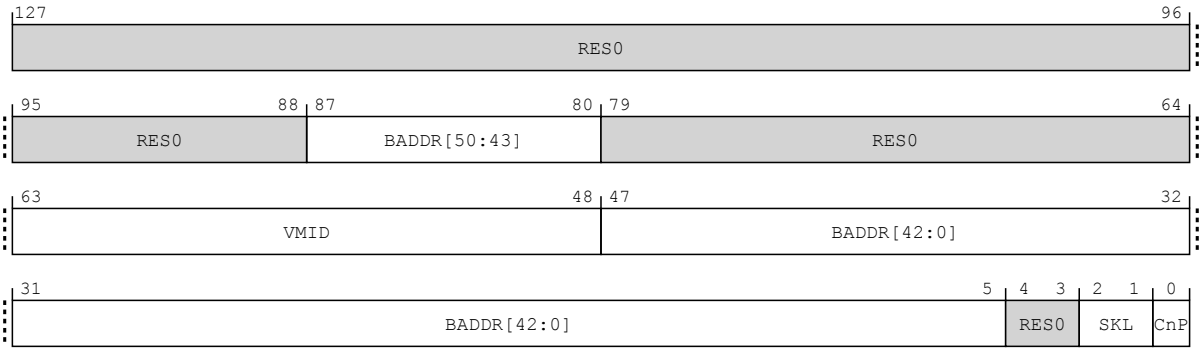

Bits [127:88]

Reserved, RES0.

BADDR, bits [87:80, 47:5]

Translation table base address:

- Bits A[55:x] of the stage 2 translation table base address bits are in register bits[87:80, 47:x].
- Bits A[(x-1):0] of the stage 2 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. x is calculated based on LOG2(StartTableSize), as described in VMSAv9-128. The smallest permitted value of x is 5.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [79:64]

Reserved, RES0.

VMID, bits [63:48]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

VMID, bits [15:0]

The VMID for the translation table.

If the implementation has an 8-bit VMID, bits [15:8] of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_VMID16 is not implemented or VTCR_EL2.VS == '0':

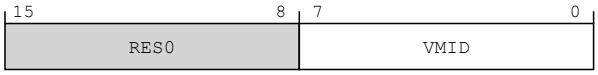

Bits [15:8]

Reserved, RES0.

VMID, bits [7:0]

The VMID for the translation table.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

SKL, bits [2:1]

Skip Level. Skip Level determines the number of levels to be skipped from the regular start level of the Non-Secure stage 2 translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SKL	Meaning
0b00	Skip 0 level from the regular start level.
0b01	Skip 1 level from the regular start level.
0b10	Skip 2 levels from the regular start level.
0b11	Skip 3 levels from the regular start level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1.

CnP	Meaning
0b0	The translation table entries pointed to by VTTBR_EL2 are permitted to differ from the entries for VTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.
0b1	The translation table entries pointed to by VTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This bit is permitted to be cached in a TLB.

Note

If the value of VTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBR_EL2s do not point to the same translation table entries when using the current VMID, then the results of translations using VTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When FEAT_D128 is not implemented or VTCR_EL2.D128 == '0':

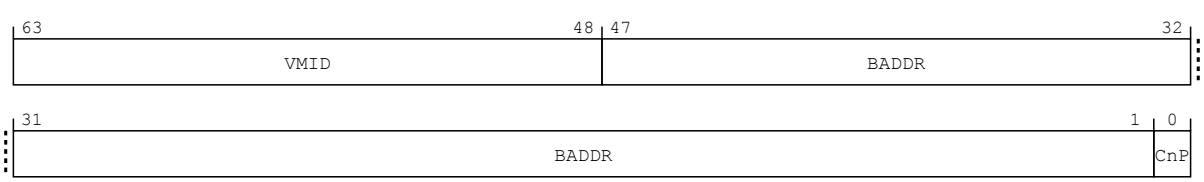

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID, bits [63:48]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

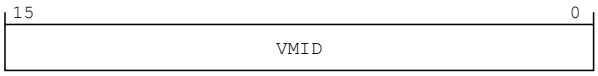

VMID, bits [15:0]

The VMID for the translation table.

If the implementation has an 8-bit VMID, bits [15:8] of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_VMID16 is not implemented or VTCR_EL2.VS == '0':

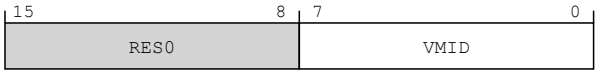

Bits [15:8]

Reserved, RES0.

VMID, bits [7:0]

The VMID for the translation table.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, A[47:x] or A[51:x], bits[47:1].

The BADDR field represents a 52-bit address if any of the following apply:

- The value of VTCR_EL2.PS represents an OA size of 52 bits.
- The Effective value of VTCR_EL2.DS is 1.

When VTTBR_EL2.BADDR represents a 52-bit addresses, all of the following apply:

- Register bits[47:x] hold bits[47:x] of the stage 2 translation table base address, where x is determined by the size of the translation table at the start level.
- The smallest permitted value of x is 6.
- Register bits[5:2] hold bits[51:48] of the stage 2 translation table base address.
- Bits[x:0] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.

Otherwise, all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Register bits[47:x] hold bits[47:x] of the stage 2 translation table base address.
- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits[51:48] of the stage 2 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of VTCR_EL2.PS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation supports a 56 bit PA size, bits [55:52] of the stage 2 translation table base address are zero.

If any VTTBR_EL2[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using VTTBR_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of VTCR_EL2.T0SZ, the stage of translation, and the translation granule size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by VTTBR_EL2 are permitted to differ from the entries for VTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.
`0b1`	The translation table entries pointed to by VTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This bit is permitted to be cached in a TLB.

Note

If the value of VTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBR_EL2s do not point to the same translation table entries when using the current VMID, then the results of translations using VTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing VTTBR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VTTBR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x020);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VTTBR_EL2()[63:0];
elsif PSTATE.EL == EL3 then
    X{64}(t) = VTTBR_EL2()[63:0];
end;
```

```
MSR VTTBR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x020) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VTTBR_EL2()[63:0] = X{64}(t);
elsif PSTATE.EL == EL3 then
    VTTBR_EL2()[63:0] = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_D128 is implemented MRRS <Xt>, <Xt+1>, VTTBR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{128}(t, t2) = NVMem128(0x020);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        X{128}(t, t2) = VTTBR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{128}(t, t2) = VTTBR_EL2();
end;
```

When FEAT_D128 is implemented MSRR VTTBR_EL2, <Xt>, <Xt+1>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem128(0x020) = X{128}(t, t2);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x14);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().D128En == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().D128En == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x14);
        end;
    else
        VTTBR_EL2()[127:0] = X{128}(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    VTTBR_EL2()[127:0] = X{128}(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.225 ZCR_EL1, SVE Control Register (EL1)

The ZCR_EL1 characteristics are:

Purpose

This register controls aspects of SVE visible at Exception levels EL1 and EL0.

Configuration

This register has no effect when FEAT_SME is implemented and the PE is in Streaming SVE mode.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this register has no effect on execution at EL0.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to ZCR_EL1 are UNDEFINED.

Attributes

ZCR_EL1 is a 64-bit register.

Field descriptions

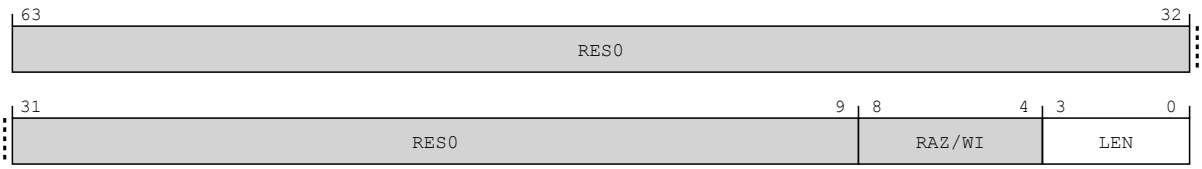

Bits [63:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Non-streaming SVE vector length at EL1 of (LEN+1)*128 bits. This field also defines the Effective Non-streaming SVE vector length at EL0 when EL2 is not implemented, or EL2 is not enabled in the current Security state, or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

The Non-streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support a subset of the architecturally permitted lengths. An implementation is required to support all lengths that are powers of two, from 128 bits up to its maximum implemented Non-streaming SVE vector length.

When FEAT_SME is not implemented, or the PE is not in Streaming SVE mode, the Effective SVE vector length (VL) is equal to the Effective Non-streaming SVE vector length.

When FEAT_SME is implemented and the PE is in Streaming SVE mode, VL is equal to the Effective Streaming SVE vector length. See SMCR_EL1.

For all purposes other than returning the result of a direct read of ZCR_EL1, the PE selects the Effective Non-streaming SVE vector length by performing checks in the following order:

1. If EL2 is implemented and enabled in the current Security state, and the requested length is greater than the Effective length at EL2, then the Effective length at EL2 is used.
2. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3. Otherwise, the Effective length is the highest supported Non-streaming SVE vector length that is less than or equal to the requested length.

An indirect read of ZCR_EL1.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ZCR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `ZCR_EL1` or `ZCR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ZCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif CPACR_EL1().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x19);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x1E0);
    else
        X{64}(t) = ZCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = ZCR_EL2();
    else
        X{64}(t) = ZCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        X{64}(t) = ZCR_EL1();
    end;
end;
```

```
MSR ZCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif CPACR_EL1().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x19);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x1E0) = X{64}(t);
    else
        ZCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif ELIsInHost(EL2) then
        ZCR_EL2() = X{64}(t);
    else
        ZCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        ZCR_EL1() = X{64}(t);
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, ZCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x1E0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
            Undefined();
        elsif CPTR_EL2().ZEN IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x19);
        elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x19);
            end;
        else
            X{64}(t) = ZCR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        if CPTR_EL3().EZ == '0' then
            AArch64_SystemAccessTrap(EL3, 0x19);
        else
            X{64}(t) = ZCR_EL1();
        end;
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR ZCR_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x1E0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
            Undefined();
        elsif CPTR_EL2().ZEN IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x19);
        elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x19);
            end;
        else
            ZCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        if CPTR_EL3().EZ == '0' then
            AArch64_SystemAccessTrap(EL3, 0x19);
        else
            ZCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.226 ZCR_EL2, SVE Control Register (EL2)

The ZCR_EL2 characteristics are:

Purpose

This register controls aspects of SVE visible at Exception levels EL2, EL1, and EL0.

Configuration

This register has no effect when EL2 is not enabled in the current Security state, or when FEAT_SME is implemented and the PE is in Streaming SVE mode.

This register is present only when FEAT_SVE is implemented. Otherwise, direct accesses to ZCR_EL2 are UNDEFINED.

Attributes

ZCR_EL2 is a 64-bit register.

Field descriptions

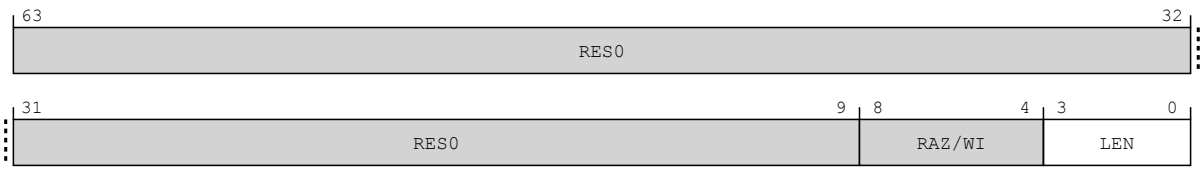

Bits [63:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Non-streaming SVE vector length at EL2 of (LEN+1)*128 bits. This field also defines the Effective Non-streaming SVE vector length at EL0 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The Non-streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support a subset of the architecturally permitted lengths. An implementation is required to support all lengths that are powers of two, from 128 bits up to its maximum implemented Non-streaming SVE vector length.

When FEAT_SME is not implemented, or the PE is not in Streaming SVE mode, the Effective SVE vector length (VL) is equal to the Effective Non-streaming SVE vector length.

When FEAT_SME is implemented and the PE is in Streaming SVE mode, VL is equal to the Effective Streaming SVE vector length. See SMCR_EL2.

For all purposes other than returning the result of a direct read of ZCR_EL2, the PE selects the Effective Non-streaming SVE vector length by performing checks in the following order:

1. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
2. Otherwise, the Effective length is the highest supported Non-streaming SVE vector length that is less than or equal to the requested length.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An indirect read of ZCR_EL2.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ZCR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `ZCR_EL2` or `ZCR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ZCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    else
        X{64}(t) = ZCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        X{64}(t) = ZCR_EL2();
    end;
end;
```

```
MSR ZCR_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    else
        ZCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        ZCR_EL2() = X{64}(t);
    end;
end;
```

MRS <Xt>, ZCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif CPACR_EL1().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x19);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x1E0);
    else
        X{64}(t) = ZCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = ZCR_EL2();
    else
        X{64}(t) = ZCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        X{64}(t) = ZCR_EL1();
    end;
end;
```

```
MSR ZCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif CPACR_EL1().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL1, 0x19);
    elsif EL2Enabled() && !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x1E0) = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        ZCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().EZ == '0' then
        Undefined();
    elsif !ELIsInHost(EL2) && CPTR_EL2().TZ == '1' then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif ELIsInHost(EL2) && CPTR_EL2().ZEN IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3().EZ == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x19);
        end;
    elsif ELIsInHost(EL2) then
        ZCR_EL2() = X{64}(t);
    else
        ZCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        ZCR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.2.227 ZCR_EL3, SVE Control Register (EL3)

The ZCR_EL3 characteristics are:

Purpose

This register controls aspects of SVE visible at all Exception levels.

Configuration

This register has no effect when FEAT_SME is implemented and the PE is in Streaming SVE mode.

This register is present only when FEAT_SVE is implemented. Otherwise, direct accesses to ZCR_EL3 are UNDEFINED.

Attributes

ZCR_EL3 is a 64-bit register.

Field descriptions

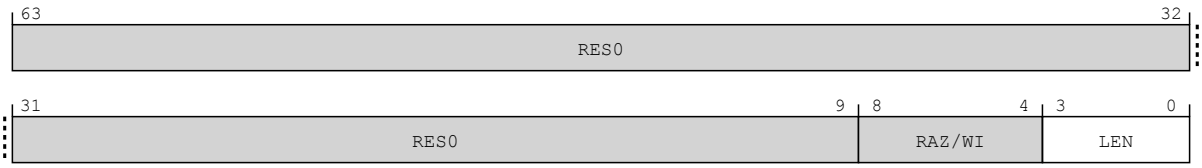

Bits [63:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Non-streaming SVE vector length at EL3 of (LEN+1)*128 bits.

The Non-streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support a subset of the architecturally permitted lengths. An implementation is required to support all lengths that are powers of two, from 128 bits up to its maximum implemented Non-streaming SVE vector length.

When FEAT_SME is not implemented, or the PE is not in Streaming SVE mode, the Effective SVE vector length (VL) is equal to the Effective Non-streaming SVE vector length.

When FEAT_SME is implemented and the PE is in Streaming SVE mode, VL is equal to the Effective Streaming SVE vector length. See SMCR_EL3.

For all purposes other than returning the result of a direct read of ZCR_EL3, the PE selects the highest supported Non-streaming SVE vector length that is less than or equal to the requested length.

An indirect read of ZCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ZCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ZCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        X{64}(t) = ZCR_EL3();
    end;
end;
```

```
MSR ZCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_SVE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().EZ == '0' then
        AArch64_SystemAccessTrap(EL3, 0x19);
    else
        ZCR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3 Debug System registers

This section lists the debug System registers in AArch64 state, in alphabetic order:

- The principal encoding space for debug System registers is `op0` == `0b10`, `op1` == {0, 3, 4}. Instructions for accessing debug System registers summarizes the registers in this encoding space and lists them in order of their encodings.
- In addition, the following registers in the `op0` == `0b11` encoding space are classified as debug System registers:
  - DLR_EL0.
  - DSPSR_EL0.
  - MDCR_EL2.
  - MDCR_EL3.
  - SDER32_EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.1 DBGAUTHSTATUS_EL1, Debug Authentication Status Register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configuration

AArch64 System register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGAUTHSTATUS[31:0].

AArch64 System register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGAUTHSTATUS_EL1 are UNDEFINED.

Attributes

DBGAUTHSTATUS_EL1 is a 64-bit register.

Field descriptions

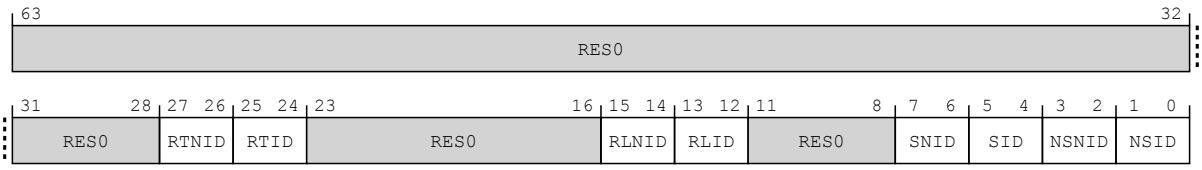

Bits [63:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
`0b00`	Not implemented.
`0b10`	Implemented and disabled. `ExternalRootInvasiveDebugEnabled()` == FALSE.
`0b11`	Implemented and enabled. `ExternalRootInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is `0b00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLID.

RLID, bits [13:12]

Realm invasive debug.

RLID	Meaning
`0b00`	Not implemented.
`0b10`	Implemented and disabled. `ExternalRealmInvasiveDebugEnabled()` == FALSE.
`0b11`	Implemented and enabled. `ExternalRealmInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is `0b00`.

Bits [11:8]

Reserved, RES0.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:

Secure non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise:

Secure non-invasive debug.

SNID	Meaning
`0b00`	Secure state is not implemented.
`0b10`	Implemented and disabled. `ExternalSecureNoninvasiveDebugEnabled()` == FALSE.
`0b11`	Implemented and enabled. `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. `ExternalSecureInvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalSecureInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

NSNID, bits [3:2]

When FEAT_Debugv8p4 is implemented:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b11	Implemented and enabled. EL3 is implemented or the Effective value of SCR_EL3.NS is 1.

All other values are reserved.

Otherwise:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. `ExternalNoninvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. `ExternalInvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

Accessing DBGAUTHSTATUS_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, DBGAUTHSTATUS_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1110	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGAUTHSTATUS_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGAUTHSTATUS_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGAUTHSTATUS_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DBGAUTHSTATUS_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configuration

If breakpoint n is not implemented, accesses to this register are UNDEFINED.

AArch64 System register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBCR<n>[31:0].

AArch64 System register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to External register DBGBCR<n>_EL1[31:0].

When FEAT_Debugv8p9 is implemented, AArch64 System register DBGBCR<n>_EL1 bits [63:32] are architecturally mapped to External register DBGBCR<n>_EL1[63:32].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGBCR<n>_EL1 are UNDEFINED.

Attributes

DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions

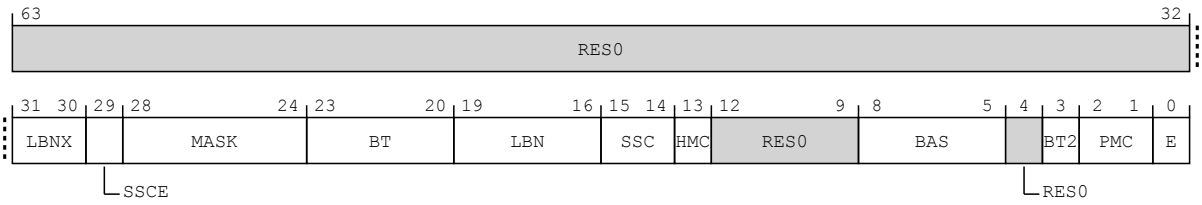

Bits [63:32]

Reserved, RES0.

LBNX, bits [31:30]

When FEAT_Debugv8p9 is implemented:

Linked Breakpoint Number.

For Linked address matching breakpoints, with DBGBCR<n>_EL1.LBN, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

This field extends DBGBCR<n>_EL1.LBN to support up to 64 implemented breakpoints.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSCE, bit [29]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented:

Security State Control Extended.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MASK, bits [28:24]

When FEAT_BWE is implemented:

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
`0b00000`	No mask.
`0b00011..0b11111`	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

If any of the following apply, then the behavior of the breakpoint is CONSTRAINED UNPREDICTABLE:

- DBGBCR<n>_EL1.MASK is a reserved value.
- DBGBCR<n>_EL1.MASK is zero, DBGBCR<n>_EL1.{BT2, BT} is {0, `0b010x`} (address mismatch breakpoint without linking enabled), and AArch32 is not supported at EL1.
- DBGBCR<n>_EL1.MASK is a valid nonzero value and any of the following apply:
  - DBGBCR<n>_EL1.BAS is not `0b1111`, and AArch32 is supported at EL0.
  - DBGBVR<n>_EL1[(MASK-1):0] is nonzero.
  - DBGBCR<n>_EL1.{BT2, BT} is {0, `0b0x1x`} or {0, `0b1xxx`} (Context matching breakpoint).

When any of these conditions apply, the breakpoint behaves as if one of the following:

- DBGBCR<n>_EL1.MASK has been programmed with a defined value, which might be `0b00000` (no mask), other than for a direct read of DBGBCR<n>_EL1.
- The breakpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type.

With DBGBCR<n>_EL1.BT2 when implemented, specifies breakpoint type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning	Applies when
0b0000	Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.	
0b0001	Linked instruction address match. As 0b0000, but linked to a breakpoint that has linking enabled.	
0b0010	Unlinked Context ID match. If the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL2. Otherwise, DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1.	breakpoint n is context-aware
0b0011	As 0b0010, with linking enabled.	breakpoint n is context-aware
0b0100	Unlinked instruction address mismatch. DBGBVR<n>_EL1 is the address of an instruction.	FEAT_BWE is implemented
0b0101	Linked instruction address mismatch. As 0b0100, but linked to a breakpoint that has linking enabled.	FEAT_BWE is implemented
0b0110	Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
0b0111	As 0b0110, with linking enabled.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
0b1000	Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	EL2 is implemented and breakpoint n is context-aware
0b1001	As 0b1000, with linking enabled.	EL2 is implemented and breakpoint n is context-aware
0b1010	Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	EL2 is implemented and breakpoint n is context-aware
0b1011	As 0b1010, with linking enabled.	EL2 is implemented and breakpoint n is context-aware
0b1100	Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
0b1101	As 0b1100, with linking enabled.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
0b1110	Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
0b1111	As 0b1110, with linking enabled.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked address matching breakpoints, with DBGBCR<n>_EL1.LBNX when implemented, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

For more information on the effect of programming the fields to a reserved set of values, see 'Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

For more information, see DBGBCR<n>_EL1.SSC.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

When FEAT_AA32 is implemented:

Byte address select. Defines which halfwords an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0011	DBGBVR<n>_EL1	Use for T32 instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS	Match instruction at	Constraint for debuggers
0b1100	DBGBVR<n>_EL1 + 2	Use for T32 instructions.
0b1111	DBGBVR<n>_EL1	Use for A64 and A32 instructions.

All other values are reserved. For more information, see 'Reserved DBGBCR<n>_EL1.BAS values'.

For more information on using the BAS field in address match breakpoints, see 'Using the BAS field in Address Match breakpoints'.

For Context matching breakpoints, this field is RES1 and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bit [4]

Reserved, RES0.

BT2, bit [3]

When FEAT_ABLE is implemented and breakpoint n supports address breakpoint linking:

Breakpoint Type 2. With DBGBCR<n>_EL1.BT, specifies breakpoint type.

BT2	Meaning
0b0	As DBGBCR<n>_EL1.BT.
0b1	As DBGBCR<n>_EL1.BT, but with linking enabled. This value is only defined for the following DBGBCR<n>_EL1.BT values: 0b0000, 0b0001, 0b0100, and 0b0101. All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

For more information, see DBGBCR<n>_EL1.SSC.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E, bit [0]

Enable breakpoint n.

E	Meaning
0b0	Breakpoint n disabled.
0b1	Breakpoint n enabled.

This field is ignored by the PE and treated as zero when all of the following are true:

- Any of the following are true:
  - `HaltOnBreakpointOrWatchpoint()` is FALSE and the Effective value of MDSCR_EL1.EMBWE is 0.
  - `HaltOnBreakpointOrWatchpoint()` is TRUE and the Effective value of EDSCR2.EHBWE is 0.
- FEAT_Debugv8p9 is implemented.
- n >= 16.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>_EL1

When FEAT_Debugv8p9 is implemented, a PE is permitted to support up to 64 implemented breakpoints.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGBCR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b101

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_BREAKPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16) >=
     ↪NUM_BREAKPOINTS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGBCRn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            X{64}(t) = DBGBCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
        else
            X{64}(t) = DBGBCR_EL1(m);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            X{64}(t) = DBGBCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
        else
            X{64}(t) = DBGBCR_EL1(m);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            X{64}(t) = DBGBCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
        else
            X{64}(t) = DBGBCR_EL1(m);
        end;
    end;
end;
```

```
MSR DBGBCR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b101

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_BREAKPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16) >=
     ↪NUM_BREAKPOINTS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGBCRn_EL1 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGBCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGBCR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGBCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGBCR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGBCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGBCR_EL1(m) = X{64}(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 63

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

Configuration

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

- When DBGBCR<n>_EL1.BT is `0b000x`, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is `0b001x`, `0b011x`, or `0b110x`, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is `0b100x`, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is `0b101x`, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is `0b111x`, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

AArch64 System register DBGBVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBVR<n>[31:0].

AArch64 System register DBGBVR<n>_EL1 bits [63:32] are architecturally mapped to AArch32 System register DBGBXVR<n>[31:0].

AArch64 System register DBGBVR<n>_EL1 bits [63:0] are architecturally mapped to External register DBGBVR<n>_EL1[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGBVR<n>_EL1 are UNDEFINED.

Attributes

DBGBVR<n>_EL1 is a 64-bit register.

Field descriptions

When DBGBCR<n>_EL1.BT IN {'000x'}:

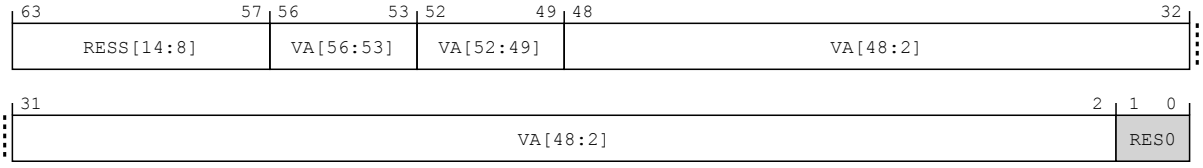

RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must set all bits in this field to the same value as the most significant bit of the VA field. If all bits in this field are not the same value as the most significant bit of the VA field, then all of the following apply:

- It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing an address.
- If the breakpoint is not context-aware, it is IMPLEMENTATION DEFINED whether the value read back in each bit of this field is a copy of the most significant bit of the VA field or the value written.

Bits [56:53]

When FEAT_LVA3 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VA[56:53], bits [56:53]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA3 is implemented, (VA[56:53]:VA[52:49]) forms the upper part of the address value. If FEAT_LVA3 is not implemented, bits VA[56:53] are part of the RESS field.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'001x'}:

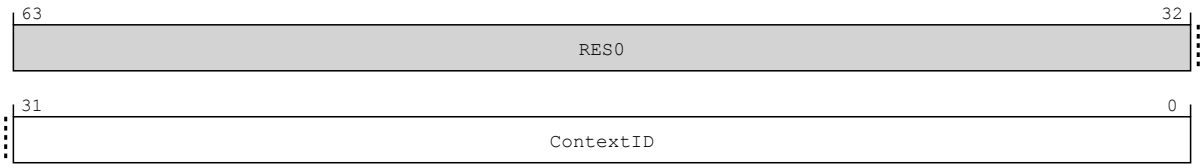

Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when the Effective value of HCR_EL2.E2H is 1, and either:

- The PE is executing at EL2.
- HCR_EL2.TGE is 1, the PE is executing at EL0, and EL2 is enabled in the current Security state.

Otherwise, the value is compared against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'011x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

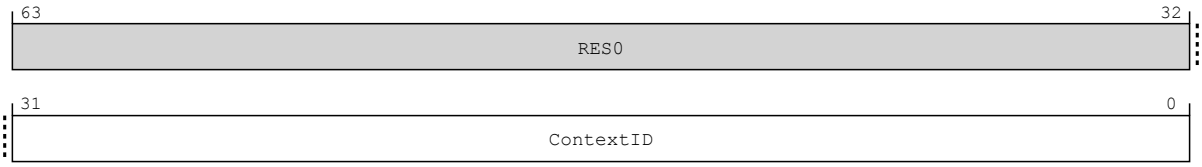

Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'100x'} and EL2 is implemented:

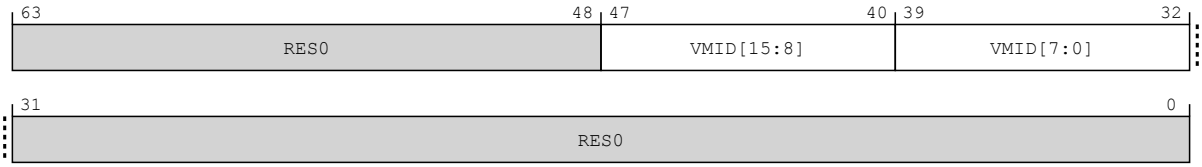

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented, VTCR_EL2.VS == '1', and EL2 is using AArch64:

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'101x'} and EL2 is implemented:

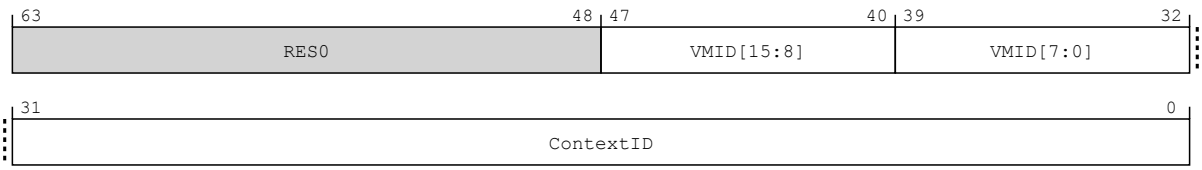

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented, VTCR_EL2.VS == '1', and EL2 is using AArch64:

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'110x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

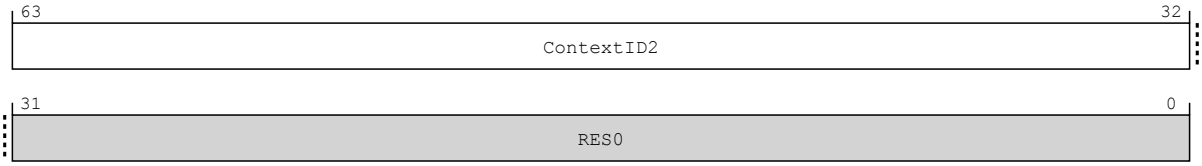

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'111x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

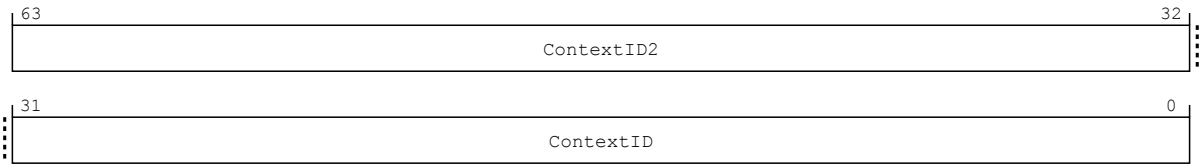

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<n>_EL1

When FEAT_Debugv8p9 is implemented, a PE is permitted to support up to 64 implemented breakpoints.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGBVR<m>_EL1 ; Where m = 0-15
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b100

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_BREAKPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16) >=
     ↪NUM_BREAKPOINTS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGBVRn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            X{64}(t) = DBGBVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
        else
            X{64}(t) = DBGBVR_EL1(m);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            X{64}(t) = DBGBVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
        else
            X{64}(t) = DBGBVR_EL1(m);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            X{64}(t) = DBGBVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
        else
            X{64}(t) = DBGBVR_EL1(m);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

```
MSR DBGBVR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b100

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_BREAKPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16) >=
     ↪NUM_BREAKPOINTS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGBVRn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGBVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGBVR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGBVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGBVR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGBVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGBVR_EL1(m) = X{64}(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear Register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET_EL1 register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch64 System register DBGCLAIMCLR_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[63:0].

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGCLAIMCLR_EL1 are UNDEFINED.

Attributes

DBGCLAIMCLR_EL1 is a 64-bit register.

Field descriptions

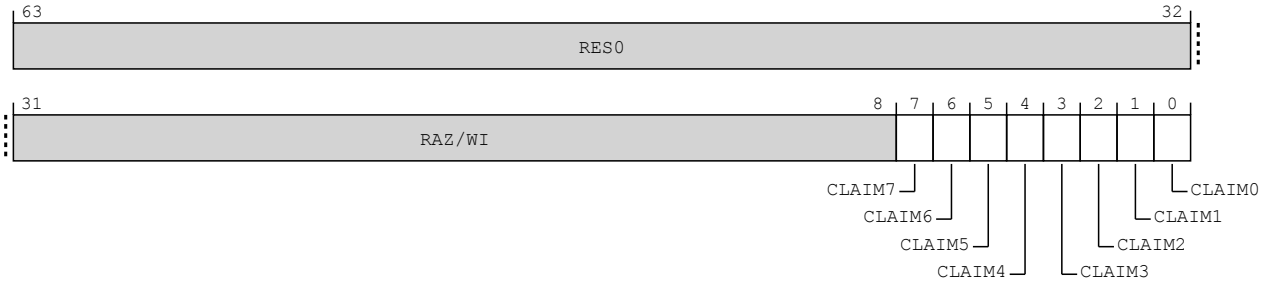

Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is W1C.

Accessing DBGCLAIMCLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGCLAIMCLR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1001	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGCLAIMCLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGCLAIMCLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DBGCLAIMCLR_EL1();
end;
```

MSR DBGCLAIMCLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1001	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        DBGCLAIMCLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        DBGCLAIMCLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGCLAIMCLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set Register

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch64 System register DBGCLAIMSET_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[63:0].

AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGCLAIMSET_EL1 are UNDEFINED.

Attributes

DBGCLAIMSET_EL1 is a 64-bit register.

Field descriptions

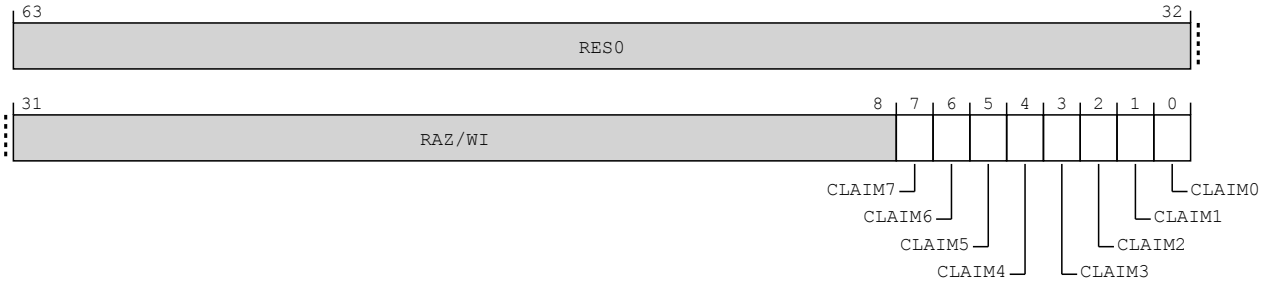

Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a write: Ignored.
0b1	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/W1S.

Accessing DBGCLAIMSET_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGCLAIMSET_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1000	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGCLAIMSET_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGCLAIMSET_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DBGCLAIMSET_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR DBGCLAIMSET_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1000	0b110

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        DBGCLAIMSET_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        DBGCLAIMSET_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGCLAIMSET_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.6 DBGDTR_EL0, Debug Data Transfer Register, half-duplex

The DBGDTR_EL0 characteristics are:

Purpose

Transfers 64 bits of data between the PE and an external debugger. Can transfer both ways using only a single register.

Configuration

When written, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGDTR_EL0 are UNDEFINED.

Attributes

DBGDTR_EL0 is a 64-bit register.

Field descriptions

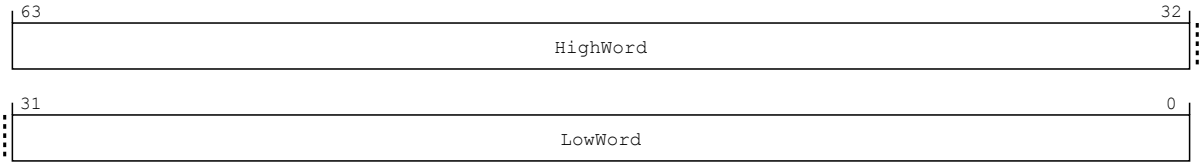

HighWord, bits [63:32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If TXfull is 1, then writes to this register set DTRRX to an UNKNOWN value. Otherwise, writes to this register set DTRRX to the value in this field. Writes to this register do not change RXfull.

Reads of this register:

- If RXfull is 1, return the last value written to DTRTX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

LowWord, bits [31:0]

If TXfull is 1, then writes to this register set DTRTX to an UNKNOWN value. Otherwise, writes to this register set DTRTX to the value in this field and set TXfull to 1.

Reads of this register:

- If RXfull is 1, return the last value written to DTRRX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

Accessing DBGDTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGDTR_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif Halted() then
    X{64}(t) = Read_DBGDTR_EL0{64}();
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{64}(t) = Read_DBGDTR_EL0{64}();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{64}(t) = Read_DBGDTR_EL0{64}();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{64}(t) = Read_DBGDTR_EL0{64}();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = Read_DBGDTR_EL0{64}();
end;
```

MSR DBGDTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif Halted() then
    Write_DBGDTR_EL0{64}(X{64}(t));
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        Write_DBGDTR_EL0{64}(X{64}(t));
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        Write_DBGDTR_EL0{64}(X{64}(t));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Write_DBGDTR_EL0{64}(X{64}(t));
    end;
elsif PSTATE.EL == EL3 then
    Write_DBGDTR_EL0{64}(X{64}(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.7 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configuration

AArch64 System register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

AArch64 System register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGDTRRX_EL0 are UNDEFINED.

Attributes

DBGDTRRX_EL0 is a 64-bit register.

Field descriptions

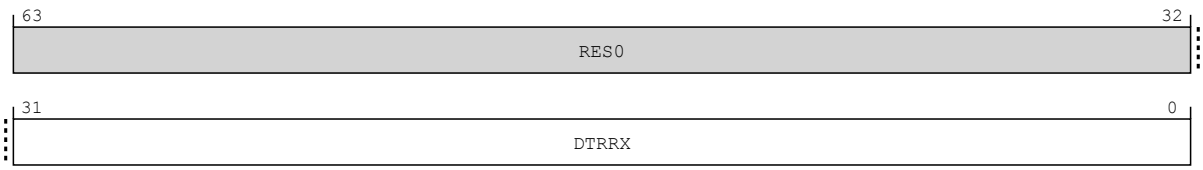

Bits [63:32]

Reserved, RES0.

DTRRX, bits [31:0]

DTRRX. Reads of this register:

- If RXfull is 1, return the last value written to DTRRX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRX_EL0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, DBGDTRRX_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif Halted() then
    X{32}(t) = Read_DBGDTR_EL0{32}();
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{32}(t) = Read_DBGDTR_EL0{32}();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{32}(t) = Read_DBGDTR_EL0{32}();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{32}(t) = Read_DBGDTR_EL0{32}();
    end;
elsif PSTATE.EL == EL3 then
    X{32}(t) = Read_DBGDTR_EL0{32}();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.8 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configuration

AArch64 System register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

AArch64 System register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGDTRTX_EL0 are UNDEFINED.

Attributes

DBGDTRTX_EL0 is a 64-bit register.

Field descriptions

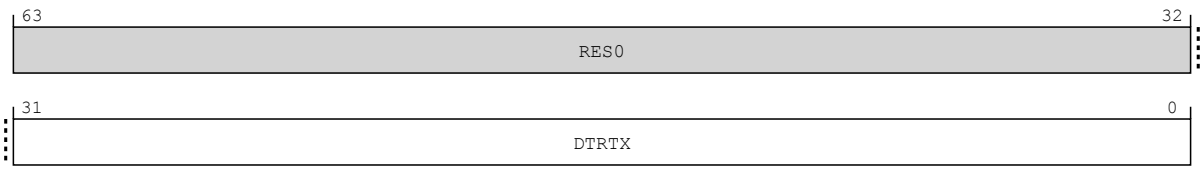

Bits [63:32]

Reserved, RES0.

DTRTX, bits [31:0]

DTRTX. Writes to this register:

- If TXfull is 1, DTRTX is set to an UNKNOWN value.
- If TXfull is 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see ‘The Debug Communication Channel and Instruction Transfer Register’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTX_EL0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR DBGDTRTX_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif Halted() then
    Write_DBGDTR_EL0{32}(X{32}(t));
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        Write_DBGDTR_EL0{32}(X{32}(t));
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        Write_DBGDTR_EL0{32}(X{32}(t));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        Write_DBGDTR_EL0{32}(X{32}(t));
    end;
elsif PSTATE.EL == EL3 then
    Write_DBGDTR_EL0{32}(X{32}(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.9 DBGPRCR_EL1, Debug Power Control Register

The DBGPRCR_EL1 characteristics are:

Purpose

Controls behavior of the PE on powerdown request.

Configuration

AArch64 System register DBGPRCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGPRCR[31:0].

AArch64 System register DBGPRCR_EL1 bit [0] is architecturally mapped to External register EDPRCR[0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGPRCR_EL1 are UNDEFINED.

Attributes

DBGPRCR_EL1 is a 64-bit register.

Field descriptions

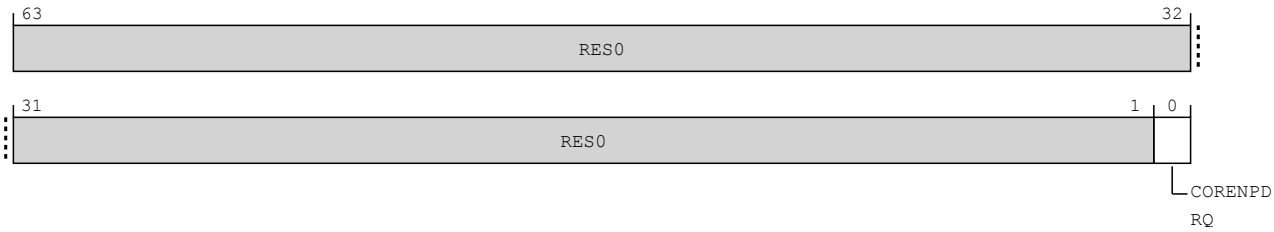

Bits [63:1]

Reserved, RES0.

CORENPDRQ, bit [0]

When FEAT_DoPD is implemented:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether this bit is reset to its Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is set to an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.

Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.

Accessing DBGPRCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGPRCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0100	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGPRCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGPRCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGPRCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DBGPRCR_EL1();
end;
```

MSR DBGPRCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGPRCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        DBGPRCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        DBGPRCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGPRCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.10 DBGVCR32_EL2, Debug Vector Catch Register

The DBGVCR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 register DBGVCR from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configuration

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

AArch64 System register DBGVCR32_EL2 bits [31:0] are architecturally mapped to AArch32 System register DBGVCR[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to DBGVCR32_EL2 are UNDEFINED.

Attributes

DBGVCR32_EL2 is a 64-bit register.

Field descriptions

When EL3 is implemented:

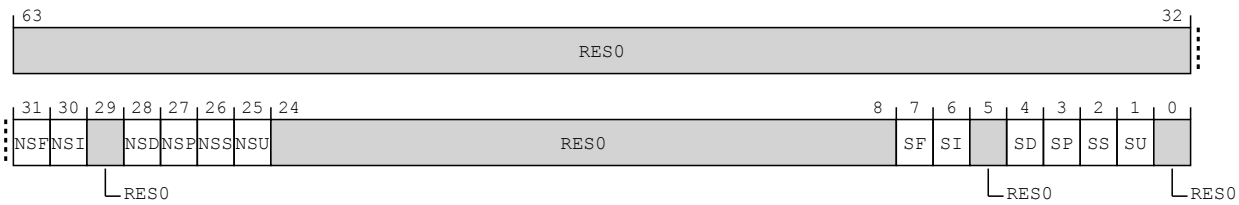

Bits [63:32]

Reserved, RES0.

NSF, bit [31]

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSI, bit [30]

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSD, bit [28]

Data Abort exception vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSP, bit [27]

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSS, bit [26]

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSU, bit [25]

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort exception vector catch enable in Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

When EL3 is not implemented:

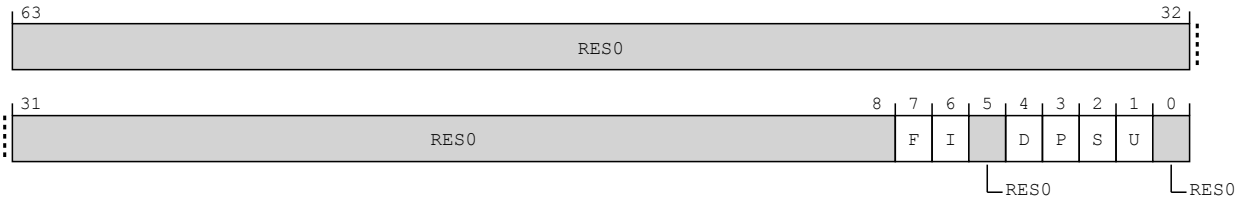

Bits [63:8]

Reserved, RES0.

F, bit [7]

FIQ vector catch enable.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I, bit [6]

IRQ vector catch enable.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

D, bit [4]

Data Abort exception vector catch enable.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P, bit [3]

Prefetch Abort vector catch enable.

The exception vector offset `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [2]

Supervisor Call (SVC) vector catch enable.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]

Undefined Instruction vector catch enable.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing DBGVCR32_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGVCR32_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b100	0b0000	0b0111	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = DBGVCR32_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DBGVCR32_EL2();
end;
```

```
MSR DBGVCR32_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b100	0b0000	0b0111	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        DBGVCR32_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGVCR32_EL2() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.11 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 63

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configuration

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch64 System register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

AArch64 System register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to External register DBGWCR<n>_EL1[31:0].

When FEAT_Debugv8p9 is implemented, AArch64 System register DBGWCR<n>_EL1 bits [63:32] are architecturally mapped to External register DBGWCR<n>_EL1[63:32].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGWCR<n>_EL1 are UNDEFINED.

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions

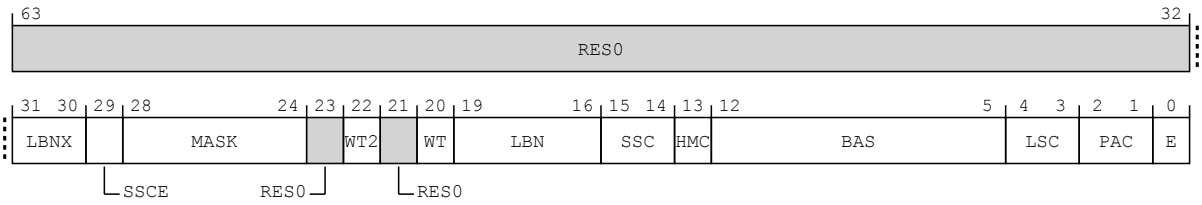

Bits [63:32]

Reserved, RES0.

LBNX, bits [31:30]

When FEAT_Debugv8p9 is implemented:

Linked Breakpoint Number.

For Linked data address watchpoints, with DBGWCR<n>_EL1.LBN, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

This field extends DBGWCR<n>_EL1.LBN to support up to 64 implemented breakpoints.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSCE, bit [29]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented:

Security State Control Extended.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
`0b00000`	No mask.
`0b00011..0b11111`	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

If programmed with a reserved value, the watchpoint behaves as if one of the following:

- DBGWCR<n>_EL1.MASK has been programmed with a defined value, which might be `0b00000` (no mask), other than for a direct read of DBGWCR<n>_EL1.
- The watchpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [23]

Reserved, RES0.

WT2, bit [22]

When FEAT_BWE2 is implemented:

Watchpoint Type 2. With DBGWCR<n>_EL1.WT, specifies watchpoint type.

WT2	Meaning
`0b0`	Watchpoint n is an address match watchpoint.
`0b1`	Watchpoint n is an address mismatch watchpoint.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked watchpoint.
0b1	Linked watchpoint.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked data address watchpoints, with DBGWCR<n>_EL1.LBNX when implemented, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see ‘Execution conditions for which a watchpoint generates Watchpoint exceptions’.

For more information on the effect of programming the fields to a reserved value, see ’Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see ‘Execution conditions for which a watchpoint generates Watchpoint exceptions’.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or doubleword addressed by DBGWVR<n>_EL1 is being watched.

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

In cases where DBGWVR<n>_EL1 addresses a doubleword:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are nonzero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See 'Reserved DBGWCR<n>_EL1.BAS values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
`0b01`	Match instructions that load from a watchpointed address.
`0b10`	Match instructions that store to a watchpointed address.
`0b11`	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n.

E	Meaning
0b0	Watchpoint n disabled.
0b1	Watchpoint n enabled.

This field is ignored by the PE and treated as zero when all of the following are true:

- Any of the following are true:
  - `HaltOnBreakpointOrWatchpoint()` is FALSE and the Effective value of MDSCR_EL1.EMBWE is 0.
  - `HaltOnBreakpointOrWatchpoint()` is TRUE and the Effective value of EDSCR2.EHBWE is 0.
- FEAT_Debugv8p9 is implemented.
- n >= 16.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>_EL1

When FEAT_Debugv8p9 is implemented, a PE is permitted to support up to 64 implemented watchpoints.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGWCR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b111

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_WATCHPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(MDSELR_EL1().BANK) * 16) >= NUM_WATCHPOINTS)
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().DBGWCRn_EL1 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            if IsFeatureImplemented(FEAT_Debugv8p9) then
                X{64}(t) = DBGWCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
            else
                X{64}(t) = DBGWCR_EL1(m);
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            if IsFeatureImplemented(FEAT_Debugv8p9) then
                X{64}(t) = DBGWCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
            else
                X{64}(t) = DBGWCR_EL1(m);
            end;
        end;
    elsif PSTATE.EL == EL3 then
        if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            if IsFeatureImplemented(FEAT_Debugv8p9) then
                X{64}(t) = DBGWCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
            else
                X{64}(t) = DBGWCR_EL1(m);
            end;
        end;
    end;
```

```
MSR DBGWCR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_WATCHPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(MDSELR_EL1().BANK) * 16) >= NUM_WATCHPOINTS)
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGWCRn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGWCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGWCR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGWCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGWCR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGWCR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGWCR_EL1(m) = X{64}(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.12 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 63

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configuration

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch64 System register DBGWVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWVR<n>[31:0].

AArch64 System register DBGWVR<n>_EL1 bits [63:0] are architecturally mapped to External register DBGWVR<n>_EL1[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DBGWVR<n>_EL1 are UNDEFINED.

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions

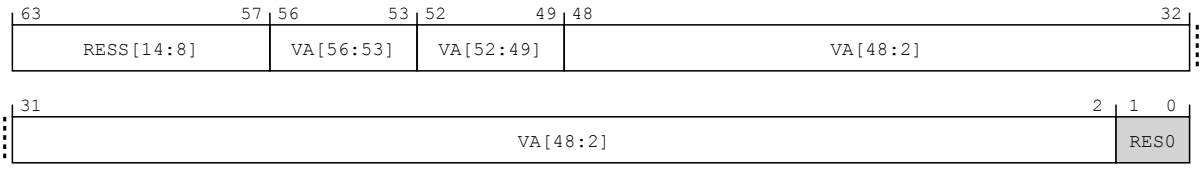

RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must set all bits in this field to the same value as the most significant bit of the VA field. If all bits in this field are not the same value as the most significant bit of the VA field, then all of the following apply:

- It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing an address.
- It is IMPLEMENTATION DEFINED whether the value read back in each bit of this field is a copy of the most significant bit of the VA field or the value written.

Bits [56:53]

When FEAT_LVA3 is implemented:

VA[56:53], bits [56:53]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA3 is implemented, (VA[56:53]:VA[52:49]) forms the upper part of the address value. If FEAT_LVA3 is not implemented, bits VA[56:53] are part of the RESS field.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>_EL1

When FEAT_Debugv8p9 is implemented, a PE is permitted to support up to 64 implemented watchpoints.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGWVR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b110

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_WATCHPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(MDSELR_EL1().BANK) * 16) >= NUM_WATCHPOINTS)
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().DBGWVRn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            if IsFeatureImplemented(FEAT_Debugv8p9) then
                X{64}(t) = DBGWVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
            else
                X{64}(t) = DBGWVR_EL1(m);
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            if IsFeatureImplemented(FEAT_Debugv8p9) then
                X{64}(t) = DBGWVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
            else
                X{64}(t) = DBGWVR_EL1(m);
            end;
        end;
    elsif PSTATE.EL == EL3 then
        if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            if IsFeatureImplemented(FEAT_Debugv8p9) then
                X{64}(t) = DBGWVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16));
            else
                X{64}(t) = DBGWVR_EL1(m);
            end;
        end;
    end;
```

```
MSR DBGWVR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b110

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif (!IsFeatureImplemented(FEAT_Debugv8p9) && m >= NUM_WATCHPOINTS) ||
     ↪(IsFeatureImplemented(FEAT_Debugv8p9) && m + (UInt(MDSELR_EL1().BANK) * 16) >= NUM_WATCHPOINTS)
     ↪then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().DBGWVRn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGWVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGWVR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGWVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGWVR_EL1(m) = X{64}(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DBGWVR_EL1(m + (UInt(EffectiveMDSELR_EL1_BANK()) * 16)) = X{64}(t);
        else
            DBGWVR_EL1(m) = X{64}(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.13 DLR_EL0, Debug Link Register

The DLR_EL0 characteristics are:

Purpose

In Debug state, holds the address to restart from.

Configuration

AArch64 System register DLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DLR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DLR_EL0 are UNDEFINED.

Attributes

DLR_EL0 is a 64-bit register.

Field descriptions

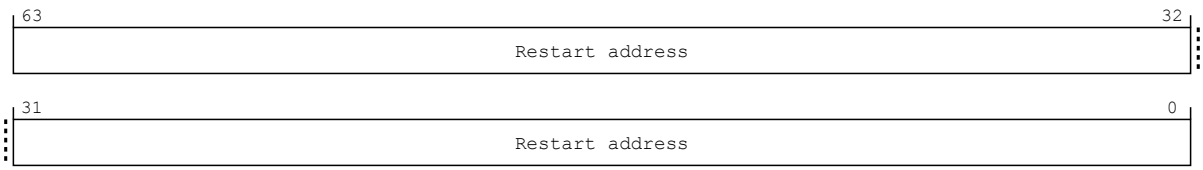

ADDR, bits [63:0]

Restart address.

Accessing DLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DLR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    X{64}(t) = DLR_EL0();
end;
```

```
MSR DLR_EL0, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    DLR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.14 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configuration

AArch64 System register DSPSR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DSPSR[31:0].

When FEAT_Debugv8p9 is implemented, AArch64 System register DSPSR_EL0 bits [63:32] are architecturally mapped to AArch32 System register DSPSR2[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to DSPSR_EL0 are UNDEFINED.

Attributes

DSPSR_EL0 is a 64-bit register.

Field descriptions

When FEAT_AA32 is implemented and exiting Debug state to AArch32 state:

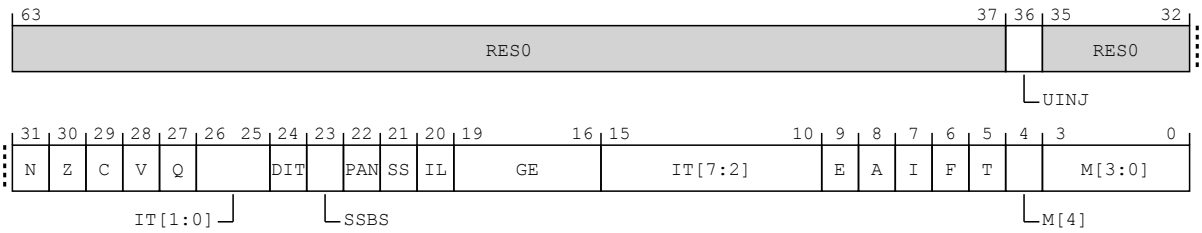

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Copied to PSTATE.UINJ on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Copied to PSTATE.N on exiting Debug state.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Copied to PSTATE.Z on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Copied to PSTATE.C on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Copied to PSTATE.V on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Copied to PSTATE.Q on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Copied to PSTATE.IT on exiting Debug state.

On exiting Debug state, DSPSR_EL0.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Copied to PSTATE.DIT on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Copied to PSTATE.SSBS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Copied to PSTATE.PAN on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Copied to PSTATE.SS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Copied to PSTATE.IL on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Copied to PSTATE.GE on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR_EL0.E is RES0. If the implementation does not support little-endian operation, DSPSR_EL0.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FIQ interrupt mask. Copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Copied to PSTATE.T on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Copied to PSTATE.nRW on exiting Debug state.

M[4]	Meaning
0b1	AArch32 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Copied to PSTATE.M[3:0] on exiting Debug state.

M[3:0]	Meaning	Applies when
0b0000	User.	
0b0001	FIQ.	
0b0010	IRQ.	
0b0011	Supervisor.	
0b0110	Monitor.	FEAT_EL3 is implemented
0b0111	Abort.	
0b1010	Hyp.	
0b1011	Undefined.	
0b1111	System.	

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_AA64 is implemented and entering or exiting Debug state from or to AArch64 state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

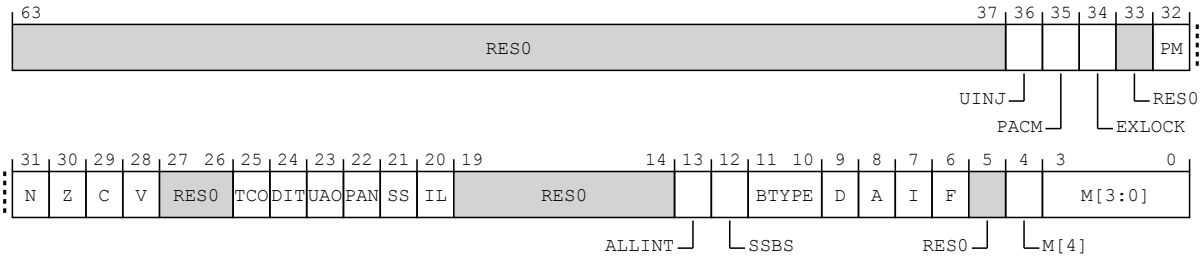

Bits [63:37]

Reserved, RES0.

UINJ, bit [36]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to the value of PSTATE.UINJ on entering Debug state, and copied to PSTATE.UINJ on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PACM, bit [35]

When FEAT_PAuth_LR is implemented:

PACM. Set to the value of PSTATE.PACM on entering Debug state, and copied to PSTATE.PACM on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLOCK, bit [34]

When FEAT_GCS is implemented:

Exception return state lock. Set to the value of PSTATE.EXLOCK on entering Debug state, and copied to PSTATE.EXLOCK on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [33]

Reserved, RES0.

PM, bit [32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_EBEP is implemented:

Profiling exception mask bit. Set to the value of PSTATE.PM on entering Debug state, and copied to PSTATE.PM on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on entering Debug state, and copied to PSTATE.TCO on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on entering Debug state, and copied to PSTATE.UAO on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

ALLINT, bit [13]

When FEAT_NMI is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on entering Debug state, and copied to PSTATE.ALLINT on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on entering Debug state, and copied to PSTATE.BTYPE on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on entering Debug state, and copied to PSTATE.D on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0, the value of PSTATE.nRW, on entering Debug state from AArch64 state, and copied to PSTATE.nRW on exiting Debug state.

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning	Applies when
0b0000	EL0.	
0b0100	EL1 with SP_EL0 (EL1t).	
0b0101	EL1 with SP_EL1 (EL1h).	
0b1000	EL2 with SP_EL0 (EL2t).	
0b1001	EL2 with SP_EL2 (EL2h).	
0b1100	EL3 with SP_EL0 (EL3t).	FEAT_EL3 is implemented
0b1101	EL3 with SP_EL3 (EL3h).	FEAT_EL3 is implemented

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on entering Debug state and copied to PSTATE.EL on exiting Debug state.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on entering Debug state and copied to PSTATE.SP on exiting Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DSPSR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DSPSR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    X{64}(t) = DSPSR_EL0();
end;
```

```
MSR DSPSR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    DSPSR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.15 MDCCINT_EL1, Monitor DCC Interrupt Enable Register

The MDCCINT_EL1 characteristics are:

Purpose

Enables interrupt requests to be signaled based on the DCC status flags.

Configuration

AArch64 System register MDCCINT_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDCCINT[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MDCCINT_EL1 are UNDEFINED.

Attributes

MDCCINT_EL1 is a 64-bit register.

Field descriptions

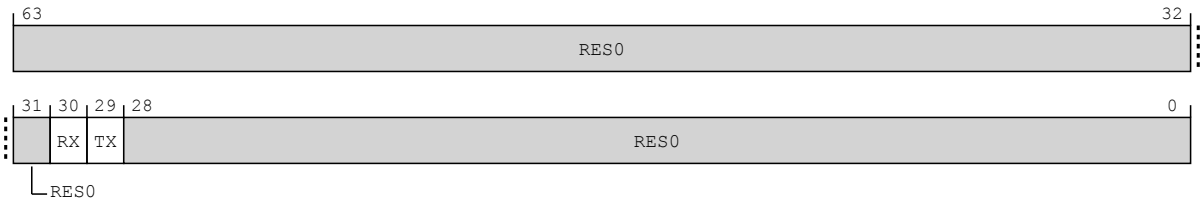

Bits [63:31]

Reserved, RES0.

RX, bit [30]

DCC interrupt request enable control for DTRRX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

RX	Meaning
0b0	No interrupt request generated by DTRRX.
0b1	Interrupt request will be generated on RXfull == 1.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TX, bit [29]

DCC interrupt request enable control for DTRTX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TX	Meaning
0b0	No interrupt request generated by DTRTX.
0b1	Interrupt request will be generated on TXfull == 0.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bits [28:0]

Reserved, RES0.

Accessing MDCCINT_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDCCINT_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X{64}(t) = MDCCINT_EL1();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
         ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDCCINT_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
         ↪then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDCCINT_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDCCINT_EL1();
end;
```

```
MSR MDCCINT_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    MDCCINT_EL1() = X{64}(t);
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDCCINT_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDCCINT_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MDCCINT_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.16 MDCCSR_EL0, Monitor DCC Status Register

The MDCCSR_EL0 characteristics are:

Purpose

Read-only register containing control status flags for the DCC.

Configuration

AArch64 System register MDCCSR_EL0 bits [30:29] are architecturally mapped to External register EDSCR[30:29].

AArch64 System register MDCCSR_EL0 bits [30:29] are architecturally mapped to AArch32 System register DBGDSCRint[30:29].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MDCCSR_EL0 are UNDEFINED.

Attributes

MDCCSR_EL0 is a 64-bit register.

Field descriptions

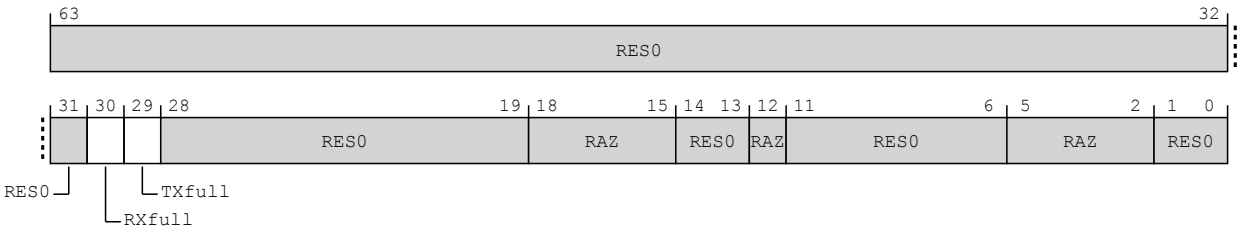

Bits [63:31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

TXfull, bit [29]

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Bits [28:19]

Reserved, RES0.

Bits [18:15]

Reserved, RAZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [14:13]

Reserved, RES0.

Bit [12]

Reserved, RAZ.

Bits [11:6]

Reserved, RES0.

Bits [5:2]

Reserved, RAZ.

Bits [1:0]

Reserved, RES0.

Accessing MDCCSR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDCCSR_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X{64}(t) = MDCCSR_EL0();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = MDCCSR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDCCSR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
     ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDCCSR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDCCSR_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.17 MDCR_EL2, Monitor Debug Configuration Register (EL2)

The MDCR_EL2 characteristics are:

Purpose

Provides EL2 configuration options for self-hosted debug and the Performance Monitors Extension.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register MDCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HDCR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MDCR_EL2 are UNDEFINED.

Attributes

MDCR_EL2 is a 64-bit register.

Field descriptions

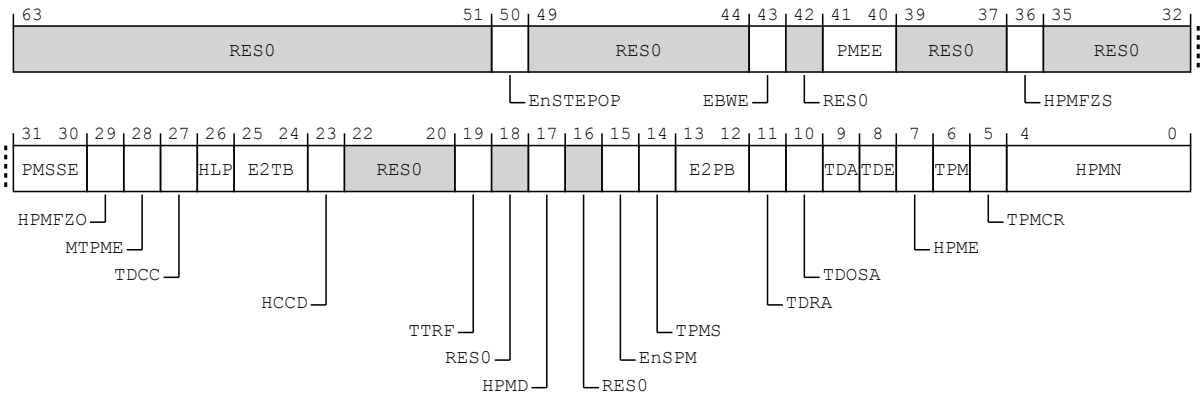

Bits [63:51]

Reserved, RES0.

EnSTEPOP, bit [50]

When FEAT_STEP2 is implemented:

EnSTEPOP	Meaning
0b0	Execution from MDSTEPOP_EL1 is disabled.
0b1	Execution from MDSTEPOP_EL1 is not disabled by this control.

If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of this field is 1, other than for a direct read of the register.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [49:44]

Reserved, RES0.

EBWE, bit [43]

When FEAT_Debugv8p9 is implemented:

Extended Breakpoint and Watchpoint Enable. Enables use of additional breakpoints or watchpoints.

EBWE	Meaning
0b0	The Effective value of MDSCR_EL1.EMBWE is 0. The Effective value of MDSELR_EL1.BANK is zero at EL2.
0b1	The Effective values of MDSCR_EL1.EMBWE and MDSELR_EL1.BANK are not affected by this field.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of this field is 1, other than for a direct read of the register.

This field is ignored by the PE and treated as 0 when EL3 is implemented and MDCR_EL3.EBWE is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [42]

Reserved, RES0.

PMEE, bits [41:40]

When FEAT_EBEP is implemented:

Performance Monitors Exception Enable. Controls the generation of the **PMUIRQ** signal and the PMU Profiling exception at EL0, EL1, and EL2.

PMEE	Meaning
0b00	The **PMUIRQ** signal is asserted on a PMU overflow, and the PMU Profiling exception is disabled.
0b01	The **PMUIRQ** signal and the PMU Profiling exception are both controlled by PMECR_EL1.PMEE.
0b11	The **PMUIRQ** signal is deasserted, and the PMU Profiling exception is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of this field is `0b01`, other than for a direct read of the register.

This field is ignored by the PE when all of the following are true:

- EL3 is implemented.
- MDCR_EL3.PMEE != `0b01`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:37]

Reserved, RES0.

HPMFZS, bit [36]

When FEAT_SPEv1p2 is implemented:

Hyp Performance Monitors Freeze-on-SPE event. Stop counters when PMBLIMITR_EL1.{PMFZ,E} is {1,1} and profiling is stopped.

HPMFZS	Meaning
`0b0`	Do not freeze on a Statistical Profiling Buffer Management event.
`0b1`	Affected counters do not count following a Statistical Profiling Buffer Management event.

The pseudocode function `SPEProfilingStopped()` describes when profiling is stopped.

The counters affected by this field are the event counters in the second range. This applies even when EL2 is disabled in the current Security state.

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

If MDCR_EL2.HPMN is equal to `GetNumEventCountersSelfHosted()`, then this field has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

PMSSE, bits [31:30]

When FEAT_PMUv3_SS is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Performance Monitors Snapshot Enable. Controls the generation of Capture events.

PMSSE	Meaning
0b00	Capture events are disabled.
0b01	Capture events are controlled by PMECR_EL1.SSE.
0b10	Capture events are enabled and prohibited.
0b11	Capture events are enabled and allowed.

If EL2 is not implemented, then the Effective value of this field is 0b01.

The reset behavior of this field is:

- On a Cold reset:
  - When the highest implemented Exception level is EL2, this field resets to '00'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPMFZO, bit [29]

When FEAT_PMUv3p7 is implemented:

Hyp Performance Monitors Freeze-on-overflow. Stop event counters on overflow.

HPMFZO	Meaning
0b0	Do not freeze on overflow.
0b1	Affected counters do not count when PMOVSCLR_EL0[m] is 1 for any event counter PMEVCNTR<m>_EL0 in the second range.

The counters affected by this field are the event counters in the second range. This applies even when EL2 is disabled in the current Security state.

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

If MDCR_EL2.HPMN is equal to GetNumEventCountersSelfHosted(), then this field has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MTPME, bit [28]

When FEAT_MTPMU is implemented and EL3 is not implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>_EL0.MT bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTPME	Meaning
0b0	FEAT_MTPMU is disabled. The Effective value of PMEVTYPER<n>_EL0.MT is 0.
0b1	PMEVTYPER<n>_EL0.MT bits not affected by this field.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this field is 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'1'`.

Otherwise:

Reserved, RES0.

TDCC, bit [27]

When FEAT_FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL1 and EL0 to EL2.

TDCC	Meaning
0b0	This control does not cause any register accesses to be trapped.
0b1	If EL2 is implemented and enabled in the current Security state, accesses to the DCC System registers at EL1 and EL0 generate a Trap exception to EL2, unless the access also generates a higher priority exception. Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC System registers trapped by this control are:

AArch64: OSDTRRX_EL1, OSDTRTX_EL1, MDCCSR_EL0, MDCCINT_EL1, and, when the PE is in Non-debug state, DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- `0x05` for trapped AArch32 `MRC` and `MCR` accesses with `coproc` == `0b1110`.
- `0x06` for trapped AArch32 `LDC` to DBGDTRTXint and `STC` from DBGDTRRXint.
- `0x18` for trapped AArch64 `MRS` and `MSR` accesses.

When the PE is in Debug state, MDCR_EL2.TDCC does not trap any accesses to:

AArch64: DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HLP, bit [26]

When FEAT_PMUv3p5 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Hypervisor Long Event Counter Enable. Determines which event counter bit generates an overflow recorded by PMOVSR[n].

HLP	Meaning
0b0	Affected counters overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].
0b1	Affected counters overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

The counters affected by this field are the event counters in the second range. This applies even when EL2 is disabled in the current Security state.

The following are not affected by this field:

- Other event counters.
- PMCCNTR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0.

If MDCR_EL2.HPMN is equal to `GetNumEventCountersSelfHosted()`, then this field has no effect.

For more information see the description of MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E2TB, bits [25:24]

When FEAT_TRBE is implemented:

EL2 Trace Buffer.

If EL2 is implemented and enabled in the trace buffer owning Security state, then this field controls the trace buffer owning translation regime.

If EL2 is implemented and enabled in the current Security state, then this field controls access to trace buffer System registers from EL1.

E2TB	Meaning
0b00	If EL2 is implemented and enabled in the trace buffer owning Security state, then the trace buffer owning Exception level is EL2. Otherwise, the trace buffer owning Exception level is EL1 and, if `TraceBufferEnabled()` == TRUE, tracing is prohibited at EL2. If EL2 is implemented and enabled in the current Security state, then accesses to trace buffer System registers at EL1 are trapped to EL2, unless the access generates a higher priority exception.
0b10	Trace buffer owning Exception level is EL1. If `TraceBufferEnabled()` == TRUE, then tracing is prohibited at EL2. If EL2 is implemented and enabled in the current Security state, then accesses to trace buffer System registers at EL1 are trapped to EL2, unless the access generates a higher priority exception.
0b11	Trace buffer owning Exception level is EL1. If `TraceBufferEnabled()` == TRUE, then tracing is prohibited at EL2. Accesses to trace buffer System registers at EL1 are not trapped by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to TRBBASER_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBPTR_EL1, TRBSR_EL1, and TRBTRG_EL1.
- If FEAT_TRBE_MPAM is implemented, `MRS` and `MSR` accesses to TRBMPAM_EL1.
- If FEAT_TRBE_EXC is implemented, `MRS` and `MSR` accesses to TRBSR_EL2 and TRBSR_EL12.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL2, reported using EC syndrome value `0x18`.

When `TraceBufferEnabled()`==FALSE, this field has no effect on whether tracing is prohibited.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HCCD, bit [23]

When FEAT_PMUv3p5 is implemented:

Hypervisor Cycle Counter Disable. Prohibits PMCCNTR_EL0 from counting at EL2.

HCCD	Meaning
`0b0`	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR_EL0 is prohibited at EL2.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bits [22:20]

Reserved, RES0.

TTRF, bit [19]

When FEAT_TRF is implemented:

Traps use of the Trace Filter Control registers at EL1 to EL2, as follows:

- Access to TRFCR_EL1 is trapped to EL2, reported using EC syndrome value `0x18`.
- Access to TRFCR is trapped to EL2, reported using EC syndrome value `0x03`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTRF	Meaning
0b0	Accesses to the specified registers at EL1 are not affected by this control.
0b1	Accesses to the specified registers at EL1 generate a trap exception to EL2 when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

When FEAT_PMUv3p1 is implemented and FEAT_Debugv8p2 is implemented:

Guest Performance Monitors Disable. Controls PMU operation at EL2.

HPMD	Meaning
0b0	Counters are not affected by this mechanism.
0b1	Affected counters are prohibited from counting at EL2. If PMCR_EL0.DP is 1, then PMCCNTR_EL0 is disabled at EL2. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.

The counters affected by this field are:

- Event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When FEAT_PMUv3p1 is implemented:

Guest Performance Monitors Disable. Controls PMU operation at EL2 when `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.

HPMD	Meaning
0b0	Counters are not affected by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPMD	Meaning
0b1	If ExternalSecureNoninvasiveDebugEnabled() is FALSE, then all the following apply: • Affected event counters are prohibited from counting at EL2. • If PMCR_EL0.DP is 1, then PMCCNTR_EL0 is disabled at EL2. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.

If ExternalSecureNoninvasiveDebugEnabled() is TRUE, then the event counters and PMCCNTR_EL0 are not affected by this field.

Otherwise, the counters affected by this field are:

- Event counters in the first range.
- If PMCR_EL0.DP is 1, the cycle counter, PMCCNTR_EL0.

Other event counters are not affected by this field. When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bit [16]

Reserved, RES0.

EnSPM, bit [15]

When FEAT_SPMU is implemented:

Enable access to System PMU registers. When disabled, accesses to System PMU registers generate a trap to EL2.

EnSPM	Meaning
0b0	Accesses of the specified System PMU registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.
0b1	Accesses of the specified System PMU registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: MRS and MSR accesses to SPMACCESSR_EL1, SPMCFGR_EL1, SPMCGCR<n>_EL1, SPMCNTENCLR_EL0, SPMCNTENSET_EL0, SPMCR_EL0, SPMDEVAFF_EL1, SPMDEVARCH_EL1, SPMEVCNTR<n>_EL0, SPMEVFILT2R<n>_EL0, SPMEVFILTR<n>_EL0, SPMEVTYPER<n>_EL0, SPMIIDR_EL1, SPMINTENCLR_EL1, SPMINTENSET_EL1, SPMOVSCLR_EL0, SPMOVSSET_EL0, SPMSCR_EL1, and SPMSELR_EL0.

Trapped instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPMS, bit [14]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE is implemented:

Trap Performance Monitor Sampling. Enables a trap to EL2 on accesses of SPE registers.

TPMS	Meaning
0b0	Accesses of the specified SPE registers are not trapped by this mechanism.
0b1	Accesses of the specified SPE registers at EL1 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to PMSCR_EL1, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIRR_EL1, and PMSLATFR_EL1.
- MRS accesses to PMSIDR_EL1.
- If FEAT_SPE_FnE is implemented, MRS and MSR accesses to PMSNEVFR_EL1.
- If FEAT_SPE_FDS is implemented, MRS and MSR accesses to PMSDSFR_EL1.

Trapped instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E2PB, bits [13:12]

When FEAT_SPE is implemented:

EL2 Profiling Buffer.

If EL2 is implemented and enabled in the Profiling Buffer owning Security state, then this field controls the Profiling Buffer owning translation regime.

If EL2 is implemented and enabled in the current Security state, then this field controls access to Profiling Buffer System registers from EL1.

E2PB	Meaning
0b00	If EL2 is implemented and enabled in the Profiling Buffer owning Security state, then the Profiling Buffer owning Exception level is EL2. Otherwise, the Profiling Buffer owning Exception level is EL1 and, Profiling is prohibited at EL2. If EL2 is implemented and enabled in the current Security state, then accesses to Profiling Buffer System registers at EL1 are trapped to EL2, unless the access generates a higher priority exception.
0b10	Profiling Buffer owning Exception level is EL1. Profiling is prohibited at EL2. If EL2 is implemented and enabled in the current Security state, then accesses to Profiling Buffer System registers at EL1 are trapped to EL2, unless the access generates a higher priority exception.
0b11	Profiling Buffer owning Exception level is EL1. Profiling is prohibited at EL2. Accesses to Profiling Buffer System registers at EL1 are not trapped by this mechanism.

All other values are reserved.

In AArch64 state, the instructions affected by this control are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MRS and MSR accesses to PMBLIMITR_EL1, PMBPTR_EL1, and PMBSR_EL1.
- If FEAT_SPE_nVM is implemented, MRS and MSR accesses to PMBMAR_EL1.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL2, reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps System register accesses to the Debug ROM registers to EL2 when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64, accesses to MDRAR_EL1 are trapped to EL2, reported using EC syndrome value 0x18.
- If EL0 or EL1 is using AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x05 and MRRC or MCRR accesses are trapped to EL2, reported using EC syndrome value 0x0C:
  - DBGDRAR, DBGDSAR.

TDRA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by the following: • DBGDSCRext.UDCCdis. • MDSCR_EL1.TDCC.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- The Effective value of MDCR_EL2.TDE is 1.
- The Effective value of HCR_EL2.TGE is 1.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug System registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDOSA, bit [10]

When FEAT_DoubleLock is implemented:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug System registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, and DBGPRCR_EL1.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x05`:
  - DBGOSLSR, DBGOSLAR, DBGOSDLR, and DBGPRCR.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

TDOSA	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 System register accesses to the powerdown debug System registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- The Effective value of MDCR_EL2.TDE is 1.
- The Effective value of HCR_EL2.TGE is 1.

System register accesses to the debug System registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug System registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x05`:
  - DBGOSLSR, DBGOSLAR, and DBGPRCR.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 are trapped.

It is IMPLEMENTATION DEFINED whether accesses to DBGOSDLR are trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 System register accesses to the powerdown debug System registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- The Effective value of MDCR_EL2.TDE is 1.
- The Effective value of HCR_EL2.TGE is 1.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug System registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap accesses of debug System registers. Enables a trap to EL2 on accesses of debug System registers.

TDA	Meaning
0b0	Accesses of the specified debug System registers are not trapped by this mechanism.
0b1	Accesses of the specified debug System registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to DBGAUTHSTATUS_EL1, DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1, MDCCINT_EL1, MDCCSR_EL0, MDSCR_EL1, OSDTRRX_EL1, OSDTRTX_EL1, and OSECCR_EL1.
- If FEAT_Debugv8p9 is implemented, MRS and MSR accesses to MDSELR_EL1.
- If FEAT_STEP2 is implemented, MRS and MSR accesses to MDSTEPOP_EL1.
- In Non-debug state, MRS accesses to DBGDTRRX_EL0 and DBGDTR_EL0 and MSR accesses to DBGDTRTX_EL0 and DBGDTR_EL0.

In AArch32 state, the instructions affected by this control are:

- MRC and MCR accesses to DBGAUTHSTATUS, DBGBCR<n>, DBGBVR<n>, DBGBXVR<n>, DBGCLAIMCLR, DBGCLAIMSET, DBGDCCINT, DBGDEVID, DBGDEVID1, DBGDEVID2, DBGDIDR, DBGDSCRext, DBGDSCRint, DBGDTRRXext, DBGDTRTXext, DBGOSECCR, DBGVCR, DBGWCR<n>, DBGWFAR, and DBGWVR<n>.
- STC accesses to DBGDTRRXint and LDC accesses to DBGDTRTXint.
- In Non-debug state, MRC accesses to DBGDTRRXint and MCR accesses to DBGDTRTXint.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trapped AArch32 instructions are reported using EC syndrome value `0x05` for `MRC` and `MCR` accesses, and `0x06` for `LDC` and `STC` accesses.

The following instructions are not trapped in Debug state:

- AArch64 `MRS` accesses to DBGDTRRX_EL0 and DBGDTR_EL0 and `MSR` accesses to DBGDTRTX_EL0 and DBGDTR_EL0.
- AArch32 `MRC` accesses to DBGDTRRXint and `MCR` accesses to DBGDTRTXint.

If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI, then it is IMPLEMENTATION DEFINED whether AArch64 accesses to MDSELR_EL1 are trapped to EL2 when MDCR_EL2.TDA is 1.

This field is ignored by the PE and treated as one when any of the following are true:

- The Effective value of MDCR_EL2.TDE is 1.
- The Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDE, bit [8]

Trap Debug Exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL_D.

TDE	Meaning
`0b0`	The debug target Exception level is EL1.
`0b1`	If EL2 is enabled for the current Effective value of SCR_EL3.NS, the debug target Exception level is EL2, otherwise the debug target Exception level is EL1. The MDCR_EL2.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

For more information, see 'Routing debug exceptions'.

This field is treated as being 1 for all purposes other than a direct read when the Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HPME, bit [7]

When FEAT_PMUv3 is implemented:

Hyp Enable.

HPME	Meaning
`0b0`	Affected counters are disabled and do not count.
`0b1`	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are the event counters in the second range. This applies even when EL2 is disabled in the current Security state.

The following counters are not affected by this field:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Other event counters.
- PMCCNTR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0.

If MDCR_EL2.HPMN is equal to `GetNumEventCountersSelfHosted()`, then this field has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When FEAT_PMUv3 is implemented:

Trap accesses of PMU registers. Enables a trap to EL2 on accesses of PMU registers.

TPM	Meaning
`0b0`	Accesses of the specified PMU registers are not trapped by this mechanism.
`0b1`	Accesses of the specified PMU registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMINTENCLR_EL1, PMINTENSET_EL1, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMSWINC_EL0, PMUSERENR_EL0, PMXEVCNTR_EL0, and PMXEVTYPER_EL0.
- `MRS` accesses to PMCEID0_EL0 and PMCEID1_EL0.
- If FEAT_PMUv3p4 is implemented, `MRS` accesses to PMMIR_EL1.
- If FEAT_PMUv3p9 is implemented, `MSR` accesses to PMZR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, `MRS` accesses to PMICFILTR_EL0 and PMICNTR_EL0.
- If FEAT_EBEP is implemented or FEAT_PMUv3_SS is implemented, `MRS` and `MSR` accesses to PMECR_EL1.

In AArch32 state, the instructions affected by this control are:

- `MRC` and `MCR` accesses to PMCCFILTR, PMCCNTR, PMCNTENCLR, PMCNTENSET, PMCR, PMEVCNTR<n>, PMEVTYPER<n>, PMINTENCLR, PMINTENSET, PMOVSR, PMOVSSET, PMSELR, PMSWINC, PMUSERENR, PMXEVCNTR, and PMXEVTYPER.
- `MRC` accesses to PMCEID0 and PMCEID1.
- `MRRC` and `MCRR` accesses to PMCCNTR.
- If FEAT_PMUv3p1 is implemented, `MRC` accesses to PMCEID2 and PMCEID3.
- If FEAT_PMUv3p4 is implemented, `MRC` accesses to PMMIR.

Trapped AArch64 instructions are reported using EC syndrome value `0x18`.

Trapped AArch32 instructions are reported using EC syndrome value `0x03` for `MRC` and `MCR` accesses, and `0x04` for `MRRC` and `MCRR` accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPMCR, bit [5]

When FEAT_PMUv3 is implemented:

Trap PMCR_EL0 or PMCR accesses. Traps EL0 and EL1 accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to PMCR_EL0 are trapped to EL2, reported using EC syndrome value `0x18`.
- In AArch32 state, accesses to PMCR are trapped to EL2, reported using EC syndrome value `0x03`.

TPMCR	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 and EL1 accesses to the specified registers are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by the following: • PMUSERENR.EN. • PMUSERENR_EL0.EN.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPMN, bits [4:0]

When FEAT_PMUv3 is implemented:

Defines the number of event counters PMEVCNTR<n>_EL0 and, if FEAT_PMUv3_SS is implemented, event counter snapshot registers PMEVCNTSVR<n>_EL1, that are accessible from EL1 and, if permitted, from EL0.

MDCR_EL2.HPMN divides the event counters accessible from self-hosted software into a first range and a second range.

When FEAT_PMUv3_EXTPMN is implemented, all of the following apply:

- PMCCR.EPMN divides the `NUM_PMU_COUNTERS` event counters implemented by the PE into the event counters that MDCR_EL2.HPMN divides into the first and second ranges, and a third range that is inaccessible from self-hosted software.
- If MDCR_EL2.HPMN is not 0 and is less than the Effective value of PMCCR.EPMN, then event counters [0..(MDCR_EL2.HPMN-1)] are in the first range, and the remaining event counters [MDCR_EL2.HPMN..(PMCCR.EPMN-1)] are in the second range.
- The pseudocode function `GetNumEventCountersSelfHosted()` returns the Effective value of PMCCR.EPMN.

When FEAT_PMUv3_EXTPMN is not implemented, all of the following apply:

- All of the `NUM_PMU_COUNTERS` event counters are accessible to self-hosted software and no counters are in the third range.
- If MDCR_EL2.HPMN is not 0 and is less than `NUM_PMU_COUNTERS`, then event counters [0..(MDCR_EL2.HPMN-1)] are in the first range, and the remaining event counters [MDCR_EL2.HPMN..(`NUM_PMU_COUNTERS`-1)] are in the second range.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The pseudocode function `GetNumEventCountersSelfHosted()` returns `NUM_PMU_COUNTERS`.

If FEAT_HPMN0 is implemented and MDCR_EL2.HPMN is 0, then all of the following apply:

- No event counters are in the first range.
- If FEAT_PMUv3_EXTPMN is implemented, then event counters [0..(PMCCR.EPMN-1)] are in the second range.
- If FEAT_PMUv3_EXTPMN is not implemented, then all event counters are in the second range.

If MDCR_EL2.HPMN is equal to `GetNumEventCountersSelfHosted()`, or EL2 is not implemented, then all of the following apply:

- If FEAT_PMUv3_EXTPMN is implemented, then event counters [0..(PMCCR.EPMN-1)] are in the first range.
- If FEAT_PMUv3_EXTPMN is not implemented, then all event counters are in the first range.
- No event counters are in the second range.

All of the following apply for an event counter PMEVCNTR<n>_EL0 in the first range:

- The counter is accessible from EL1, EL2, and EL3.
- The counter is accessible from EL0 if permitted by PMUSERENR_EL0 and PMUACR_EL1, or by PMUSERENR.
- If FEAT_PMUv3p5 is implemented, then PMCR_EL0.LP or PMCR.LP determines whether the counter overflow flag PMOVSSET_EL0[n] is set on unsigned overflow of PMEVCNTR<n>_EL0[31:0] or PMEVCNTR<n>_EL0[63:0].
- PMCR_EL0.E and PMCNTENSET_EL0[n] enable the operation of the event counter.

All of the following apply for an event counter PMEVCNTR<n>_EL0 in the second range:

- The counter is accessible from EL2 and EL3.
- If EL2 is disabled in the current Security state, then the event counter is accessible from EL1, and from EL0 if permitted by PMUSERENR_EL0 and PMUACR_EL1, or by PMUSERENR.
- If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP determines whether the counter overflow flag PMOVSSET_EL0[n] is set on unsigned overflow of PMEVCNTR<n>_EL0[31:0] or PMEVCNTR<n>_EL0[63:0].
- MDCR_EL2.HPME and PMCNTENSET_EL0[n] enable the operation of the event counter.

If FEAT_PMUv3_SS is implemented, then all of the following apply:

- For an event counter snapshot register PMEVCNTSVR<n>_EL1 in the first range, the register is accessible from EL1, EL2, and EL3.

- For an event counter snapshot register PMEVCNTSVR<n>_EL1 in the second range, the register is accessible from EL2 and EL3. If EL2 is disabled in the current Security state, the event counter is also accessible from EL1.

For information about counters in the third range, see the description of PMCCR.EPMN.

Values greater than `GetNumEventCountersSelfHosted()` are reserved. If FEAT_HPMN0 is not implemented, then the value 0 is reserved.

When FEAT_PMUv3_EXTPMN is not implemented and this field is set to a reserved value, the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- The number of event counters in each of the first and second ranges is UNKNOWN. That is, either:
  - The PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN nonzero value less than or equal to `NUM_PMU_COUNTERS`.
  - All counters are in the second range and none are in the first range.

When FEAT_PMUv3_EXTPMN is implemented and this field is set to a reserved value, the following behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is the Effective value of PMCCR.EPMN.
- No event counters are in the second range.
- The value returned by an indirect read of MDCR_EL2.HPMN as a result of direct reads of PMCR_EL0.N or PMCR.N is the Effective value of PMCCR.EPMN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to the expression `NUM_PMU_COUNTERS`.

Otherwise:

Reserved, RES0.

Accessing MDCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDCR_EL2();
end;
```

```
MSR MDCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MDCR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.18 MDCR_EL3, Monitor Debug Configuration Register (EL3)

The MDCR_EL3 characteristics are:

Purpose

Provides EL3 configuration options for self-hosted debug and the Performance Monitors Extension.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MDCR_EL3 are UNDEFINED.

Attributes

MDCR_EL3 is a 64-bit register.

Field descriptions

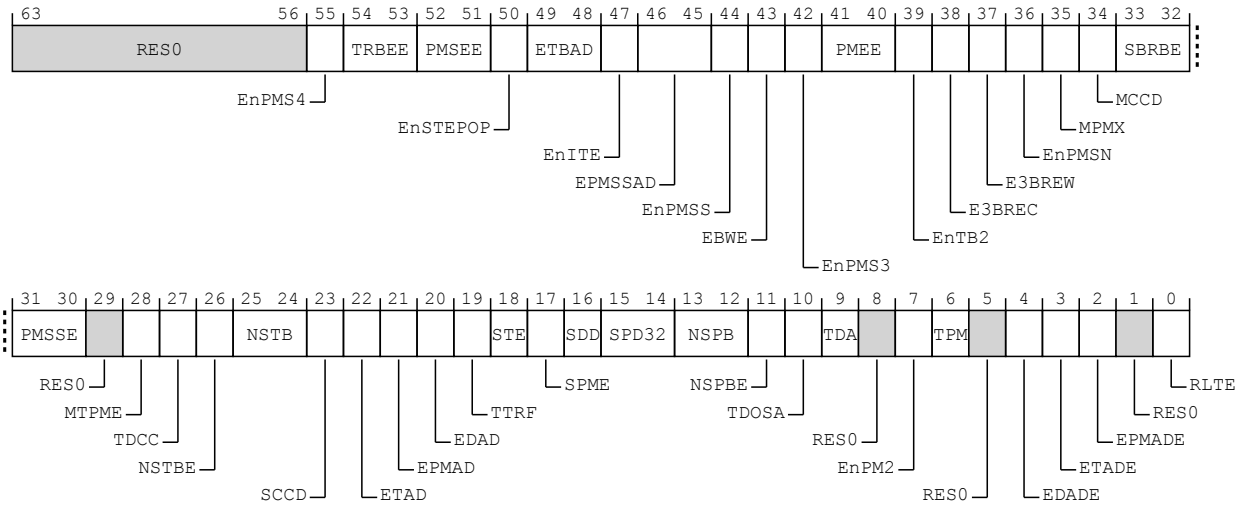

Bits [63:56]

Reserved, RES0.

EnPMS4, bit [55]

When FEAT_SPE_nVM is implemented:

Enable access to SPE registers. When disabled, accesses to SPE registers generate a trap to EL3.

EnPMS4	Meaning
0b0	Accesses of the specified SPE registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
0b1	Accesses of the specified SPE registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: MRS and MSR accesses to PMBMAR_EL1.

Trapped instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRBEE, bits [54:53]

When FEAT_TRBE_EXC is implemented:

Trace buffer management event Exception Enable.

TRBEE	Meaning
0b00	Disabled. TRBE Profiling exceptions for all Exception levels are disabled. All of the following apply: • No trace buffer management events are recorded in TRBSR_EL3. • TRBE Profiling exceptions are not generated. • TRBSR_EL1.IRQ drives the interrupt request signal **TRBIRQ**. • Accesses to TRBSR_EL2 at EL2 are trapped to EL3. • Accesses to TRBSR_EL1 at EL1 ignore the value of HCR_EL2.NV1 and accesses to TRBSR_EL1 at EL2 ignore the value of HCR_EL2.E2H.
0b01	Delegated. TRBE Profiling exceptions for EL3 are disabled, but might be enabled for EL2 or EL1 by TRFCR_EL2.EE or TRFCR_EL1.EE. All of the following apply: • No trace buffer management events are recorded in TRBSR_EL3. • TRBSR_EL3.IRQ is ignored and TRBE Profiling exceptions are not taken to EL3.

0b10	Enabled. TRBE Profiling exceptions for EL3 are enabled for trace buffer management events targeting EL3, as follows:
	• Trace buffer management events due to a fault on a write to the trace buffer that would generate a Data Abort exception taken to EL3 if generated by a store instruction executed at the owning Exception level are recorded in TRBSR_EL3. That is, any of the following faults:
	  • Granule Protection Check faults other than Granule Protection Faults (GPFs).
	  • If SCR_EL3.GPF is 1, GPFs.
	  • If SCR_EL3.EA is 1, External aborts.
	• TRBE Profiling exceptions are generated and taken to EL3 when unmasked and TRBSR_EL3.IRQ is 1.

0b11	Trap all. TRBE Profiling exceptions for EL3 are enabled for all trace buffer management events, as follows: • All trace buffer management events are recorded in TRBSR_EL3. • TRBE Profiling exceptions are generated and taken to EL3 when unmasked and TRBSR_EL3.IRQ is 1.

If EL3 is not implemented, then the Effective value of MDCR_EL3.TRBEE is `0b01`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSEE, bits [52:51]

When FEAT_SPE_EXC is implemented:

Profiling Buffer management event Exception Enable.

PMSEE	Meaning
`0b00`	Disabled. SPE Profiling exceptions for all Exception levels are disabled. All of the following apply: • No Profiling Buffer management events are recorded in PMBSR_EL3. • SPE Profiling exceptions are not generated. • PMBSR_EL1.S drives the interrupt request signal **PMBIRQ**. • Accesses to PMBSR_EL2 at EL2 are trapped to EL3. • Accesses to PMBSR_EL1 at EL1 ignore the value of HCR_EL2.NV1 and accesses to PMBSR_EL1 at EL2 ignore the value of HCR_EL2.E2H.
`0b01`	Delegated. SPE Profiling exceptions for EL3 are disabled, but might be enabled for EL2 or EL1 by PMSCR_EL2.EE or PMSCR_EL1.EE. All of the following apply: • No Profiling Buffer management events are recorded in PMBSR_EL3. • PMBSR_EL3.S is ignored and SPE Profiling exceptions are not taken to EL3.

`0b10`	Enabled. SPE Profiling exceptions for EL3 are enabled for Profiling Buffer management events targeting EL3, as follows:
	• Profiling Buffer management events due to a fault on a write to the Profiling Buffer that would generate a Data Abort exception taken to EL3 if generated by a store instruction executed at the owning Exception level are recorded in PMBSR_EL3. That is, any of the following faults:
	  • Granule Protection Check faults other than Granule Protection Faults (GPFs).
	  • If SCR_EL3.GPF is 1, GPFs.
	  • If SCR_EL3.EA is 1, External aborts.
	• SPE Profiling exceptions are generated and taken to EL3 when unmasked and PMBSR_EL3.S is 1.

`0b11`	Trap all. SPE Profiling exceptions for EL3 are enabled for all Profiling Buffer management events, as follows: • All Profiling Buffer management events are recorded in PMBSR_EL3. • SPE Profiling exceptions are generated and taken to EL3 when unmasked and PMBSR_EL3.S is 1.

If EL3 is not implemented, then the Effective value of MDCR_EL3.PMSEE is `0b01`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSTEPOP, bit [50]

When FEAT_STEP2 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnSTEPOP	Meaning
0b0	Execution from MDSTEPOP_EL1 is disabled. EL2, EL1 and EL0 System register accesses to MDSTEPOP_EL1 are trapped to EL3, unless the access generates a higher priority exception.
0b1	Execution from MDSTEPOP_EL1 is not disabled by this control. System register accesses to MDSTEPOP_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ETBAD, bits [49:48]

When FEAT_TRBE_EXT is implemented:

External Trace Buffer Access Disable. Controls access to the Trace Buffer registers from an external debugger.

ETBAD	Meaning	Applies when
0b00	Non-secure accesses from an external debugger to Trace Buffer registers are prohibited. If FEAT_RME is implemented, Secure and Realm accesses from an external debugger to Trace Buffer registers are prohibited and Root accesses to Trace Buffer registers are allowed. If FEAT_RME is not implemented, Secure accesses to Trace Buffer registers are allowed.	
0b01	Secure and Non-secure accesses from an external debugger to Trace Buffer registers are prohibited. Root and Realm accesses to Trace Buffer registers are allowed.	FEAT_RME is implemented
0b10	Realm and Non-secure accesses from an external debugger to Trace Buffer registers are prohibited. Root and Secure accesses to Trace Buffer registers are allowed.	FEAT_RME is implemented
0b11	All accesses from an external debugger to Trace Buffer registers are allowed.	

If EL3 is not implemented, then the Effective value of this field is 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

Otherwise:

Reserved, RES0.

EnITE, bit [47]

When FEAT_ITE is implemented:

Enable access to Instrumentation trace System registers. When disabled, accesses to Instrumentation trace System registers generate a trap to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnITE	Meaning
0b0	Accesses of the specified Instrumentation trace System registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
0b1	Accesses of the specified Instrumentation trace System registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: MRS and MSR accesses to TRCITECR_EL1, TRCITECR_EL2, and TRCITECR_EL12.

Trapped instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EPMSSAD, bits [46:45]

When FEAT_PMUv3_SS is implemented:

External PMU Snapshot Access Disable. Controls access to the PMU Snapshot registers from an external debugger.

External accesses of the following registers are affected by this control:

- PMCCNTSVR_EL1, PMEVCNTSVR<n>_EL1, and PMSSCR_EL1.
- If FEAT_PMUv3_ICNTR is implemented, PMICNTSVR_EL1.

EPMSSAD	Meaning	Applies when
0b00	Non-secure accesses from an external debugger to the affected PMU Snapshot registers are prohibited. If FEAT_RME is implemented, Secure and Realm accesses from an external debugger to the affected PMU Snapshot registers are prohibited. Other accesses from an external debugger to the specified registers are not affected by this control.	
0b01	Secure and Non-secure accesses from an external debugger to the affected PMU Snapshot registers are prohibited. Other accesses from an external debugger to the specified registers are not affected by this control.	FEAT_RME is implemented
0b10	Realm and Non-secure accesses from an external debugger to the affected PMU Snapshot registers are prohibited. Other accesses from an external debugger to the specified registers are not affected by this control.	FEAT_RME is implemented
0b11	No accesses from an external debugger to the affected PMU Snapshot registers are prohibited by this control.	

If EL3 is not implemented, then the Effective value of this field is 0b11.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to `'00'`.

Otherwise:

Reserved, RES0.

EnPMSS, bit [44]

When FEAT_PMUv3_SS is implemented:

Enable access to PMU Snapshot registers. When disabled, accesses to PMU Snapshot registers generate a trap to EL3.

EnPMSS	Meaning
`0b0`	Accesses of the specified PMU Snapshot registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
`0b1`	Accesses of the specified PMU Snapshot registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to PMCCNTSVR_EL1, PMEVCNTSVR<n>_EL1, and PMSSCR_EL1.
- If FEAT_PMUv3_ICNTR is implemented, `MRS` and `MSR` accesses to PMICNTSVR_EL1.

Trapped instructions are reported using EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EBWE, bit [43]

When FEAT_Debugv8p9 is implemented:

Extended Breakpoint and Watchpoint Enable. Enables use of additional breakpoints or watchpoints, and enables a trap to EL3 on accesses to debug System registers.

EBWE	Meaning
`0b0`	The Effective values of MDSCR_EL1.EMBWE and MDCR_EL2.EBWE are 0. The Effective value of MDSELR_EL1.BANK is zero at EL3. Accesses of MDSELR_EL1 at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
`0b1`	The Effective values of MDSCR_EL1.EMBWE, MDCR_EL2.EBWE, and MDSELR_EL1.BANK are not affected by this field. Accesses of MDSELR_EL1 are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: `MRS` and `MSR` accesses to MDSELR_EL1.

Trapped instructions are reported using EC syndrome value `0x18`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

If EL3 is not implemented, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EnPMS3, bit [42]

When FEAT_SPE_FDS is implemented:

Enable access to SPE registers. When disabled, accesses to SPE registers generate a trap to EL3.

EnPMS3	Meaning
`0b0`	Accesses of the specified SPE registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
`0b1`	Accesses of the specified SPE registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: `MRS` and `MSR` accesses to PMSDSFR_EL1.

Trapped instructions are reported using EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMEE, bits [41:40]

When FEAT_EBEP is implemented:

Performance Monitors Exception Enable. Controls the generation of the **PMUIRQ** signal and the PMU Profiling exception at all Exception levels.

PMEE	Meaning
`0b00`	The **PMUIRQ** signal is asserted on a PMU overflow, and the PMU Profiling exception is disabled.
`0b01`	The **PMUIRQ** signal and the PMU Profiling exception are both controlled by MDCR_EL2.PMEE.
`0b11`	The **PMUIRQ** signal is deasserted, and the PMU Profiling exception is enabled.

All other values are reserved.

If EL3 is not implemented, then the Effective value of this field is `0b01`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'00'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnTB2, bit [39]

When FEAT_TRBE_MPAM is implemented:

Enable access to Trace Buffer registers. When disabled, accesses to Trace Buffer registers generate a trap to EL3.

EnTB2	Meaning
0b0	Accesses of the specified Trace Buffer registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.
0b1	Accesses of the specified Trace Buffer registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: MRS and MSR accesses to TRBMPAM_EL1.

Trapped instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3BREC, bit [38]

When FEAT_BRBEv1p1 is implemented:

Branch Record Buffer EL3 Cold Reset Enable. With MDCR_EL3.E3BREW, controls branch recording at EL3.

E3BREC	Meaning
0b0	When MDCR_EL3.E3BREW == 0: Branch recording at EL3 is disabled. When MDCR_EL3.E3BREW == 1: Branch recording at EL3 is enabled.
0b1	When MDCR_EL3.E3BREW == 0: Branch recording at EL3 is enabled. When MDCR_EL3.E3BREW == 1: Branch recording at EL3 is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

E3BREW, bit [37]

When FEAT_BRBEv1p1 is implemented:

Branch Record Buffer EL3 Warm Reset Enable. With MDCR_EL3.E3BREC, controls branch recording at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a description of the values derived by evaluating MDCR_EL3.E3BREC and MDCR_EL3.E3BREW together, see MDCR_EL3.E3BREC.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EnPMSN, bit [36]

When FEAT_SPE_FnE is implemented:

Trap accesses to PMSNEVFR_EL1. Controls access to Statistical Profiling PMSNEVFR_EL1 System register from EL2 and EL1.

EnPMSN	Meaning
`0b0`	Accesses to PMSNEVFR_EL1 at EL2 and EL1 generate a Trap exception to EL3.
`0b1`	Do not trap PMSNEVFR_EL1 to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MPMX, bit [35]

When FEAT_PMUv3p7 is implemented:

Monitor Performance Monitors Extended control. With MDCR_EL3.SPME, controls PMU operation at EL3.

MPMX	Meaning
`0b0`	Counters are not affected by this mechanism.
`0b1`	Affected counters are prohibited from counting at EL3. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled at EL3. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.

The counters affected by this field are:

- If EL2 is implemented and MDCR_EL3.SPME is 1, event counters PMEVCNTR<n>_EL0 in the first range.
- If EL2 is not implemented or MDCR_EL3.SPME is 0, event counters in the first and second ranges.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter, PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter, PMCCNTR_EL0.

Other event counters are not affected by this field. When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

MCCD, bit [34]

When FEAT_PMUv3p7 is implemented:

Monitor Cycle Counter Disable. Prohibits the Cycle Counter, PMCCNTR_EL0, from counting at EL3.

MCCD	Meaning
`0b0`	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR_EL0 is prohibited at EL3.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SBRBE, bits [33:32]

When FEAT_BRBE is implemented:

Secure Branch Record Buffer Enable. Controls branch recording by the BRBE, and access to BRBE registers and instructions at EL2 and EL1.

SBRBE	Meaning
`0b00`	Direct accesses to BRBE registers and instructions, except when at EL3, generate a Trap exception to EL3. EL0, EL1, and EL2 are prohibited regions.
`0b01`	Direct accesses to BRBE registers and instructions in Secure state, except when at EL3, generate a Trap exception to EL3. EL0, EL1, and EL2 in Secure state are prohibited regions. This control does not cause any direct accesses to BRBE registers when not in Secure state to be trapped, and does not cause any Exception levels when not in Secure state to be a prohibited region.
`0b10`	Direct accesses to BRBE registers and instructions, except when at EL3, generate a Trap exception to EL3. This control does not cause any Exception levels to be prohibited regions.
`0b11`	This control does not cause any direct accesses to BRBE registers or instruction to be trapped, and does not cause any Exception levels to be a prohibited region.

The Branch Record Buffer registers trapped by this control are: BRBCR_EL1, BRBCR_EL2, BRBCR_EL12, BRBFCR_EL1, BRBIDR0_EL1, BRBINF<n>_EL1, BRBINFINJ_EL1, BRBSRC<n>_EL1, BRBSRCINJ_EL1, BRBTGT<n>_EL1, BRBTGTINJ_EL1, and BRBTS_EL1.

The Branch Record Buffer instructions trapped by this control are:

- BRB IALL.
- BRB INJ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If FEAT_BRBEv1p1 is not implemented, EL3 is a prohibited region.

If EL3 is not implemented, then the Effective value of this field is `0b11`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMSSE, bits [31:30]

When FEAT_PMUv3_SS is implemented:

Performance Monitors Snapshot Enable. Controls the generation of Capture events.

PMSSE	Meaning
`0b00`	Capture events are disabled.
`0b01`	Capture events are controlled by MDCR_EL2.PMSSE.
`0b10`	Capture events are enabled and prohibited.
`0b11`	Capture events are enabled and allowed.

If EL3 is not implemented, then the Effective value of this field is `0b01`.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

Otherwise:

Reserved, RES0.

Bit [29]

Reserved, RES0.

MTPME, bit [28]

When FEAT_MTPMU is implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>_EL0.MT bits.

MTPME	Meaning
`0b0`	FEAT_MTPMU is disabled. The Effective value of PMEVTYPER<n>_EL0.MT is 0.
`0b1`	PMEVTYPER<n>_EL0.MT bits not affected by this field.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this field is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'1'`.

Otherwise:

Reserved, RES0.

TDCC, bit [27]

When FEAT_FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL2, EL1, and EL0 to EL3.

TDCC	Meaning
`0b0`	This control does not cause any register accesses to be trapped.
`0b1`	Accesses to the DCC System registers at EL2, EL1, and EL0 generate a Trap exception to EL3, unless the access also generates a higher priority exception. Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC System registers trapped by this control are:

AArch64: OSDTRRX_EL1, OSDTRTX_EL1, MDCCSR_EL0, MDCCINT_EL1, and, when the PE is in Non-debug state, DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- `0x05` for trapped AArch32 `MRC` and `MCR` accesses with `coproc` == `0b1110`.
- `0x06` for trapped AArch32 `LDC` to DBGDTRTXint and `STC` from DBGDTRRXint.
- `0x18` for trapped AArch64 `MRS` and `MSR` accesses.

When the PE is in Debug state, MDCR_EL3.TDCC does not trap any accesses to:

AArch64: DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.

AArch32: DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSTBE, bit [26]

When FEAT_TRBE is implemented and FEAT_RME is implemented:

Non-secure Trace Buffer Extended. Together with MDCR_EL3.NSTB, controls the trace buffer owning Security state and accesses to trace buffer System registers from EL2 and EL1.

For a description of the values derived by evaluating NSTB and NSTBE together, see MDCR_EL3.NSTB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSTB, bits [25:24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TRBE is implemented and FEAT_RME is implemented:

Non-secure Trace Buffer. Together with MDCR_EL3.NSTBE, controls the trace buffer owning Security state and accesses to trace buffer System registers from EL2 and EL1.

NSTBE	NSTB	Meaning
0b0	0b00	Trace buffer owning Security state is Secure state. When TraceBufferEnabled()==TRUE, tracing is prohibited in Realm and Non-secure states. Accesses to trace buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception. When Secure state is not implemented, this encoding is reserved.
0b0	0b01	Trace buffer owning Security state is Secure state. When TraceBufferEnabled()==TRUE, tracing is prohibited in Realm and Non-secure states. Accesses to trace buffer System registers at Realm and Non-secure EL2, and Realm and Non-secure EL1, are trapped to EL3, unless the access generates a higher priority exception. When Secure state is not implemented, this encoding is reserved.
0b0	0b10	Trace buffer owning Security state is Non-secure state. When TraceBufferEnabled()==TRUE, tracing is prohibited in Secure and Realm states. Accesses to trace buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception.
0b0	0b11	Trace buffer owning Security state is Non-secure state. When TraceBufferEnabled()==TRUE, tracing is prohibited in Secure and Realm states. Accesses to trace buffer System registers at Secure and Realm EL2, and Secure and Realm EL1, are trapped to EL3, unless the access generates a higher priority exception.
0b1	0b10	Trace buffer owning Security state is Realm state. | When TraceBufferEnabled()==TRUE, tracing is prohibited in Secure and Non-secure states. | Accesses to trace buffer System registers at EL2 and EL1 are | trapped to EL3, unless the access generates a higher priority exception.
0b1	0b11	Trace buffer owning Security state is Realm state. When TraceBufferEnabled()==TRUE, tracing is prohibited in Secure and Non-secure states. Accesses to trace buffer System registers at Secure and Non-secure EL2, and Secure and Non-secure EL1, are trapped to EL3, unless the access generates a higher priority exception.

All other combinations of MDCR_EL3.NSTBE and MDCR_EL3.NSTB are reserved.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to TRBBASER_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBPTR_EL1, TRBSR_EL1, and TRBTRG_EL1.
- If FEAT_TRBE_MPAM is implemented, `MRS` and `MSR` accesses to TRBMPAM_EL1.
- If FEAT_TRBE_EXC is implemented, `MRS` and `MSR` accesses to TRBSR_EL2 and TRBSR_EL12.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3, reported using EC syndrome value `0x18`.

When `TraceBufferEnabled()`==FALSE, these controls have no effect on whether tracing is prohibited.

If the Trace Buffer Unit is enabled and using Self-hosted mode, and MDCR_EL3.{NSTB, NSTBE} selects a reserved value, then the behavior is CONSTRAINED UNPREDICTABLE, and the Trace Buffer Unit does one of:

- Behaves as if the Trace Buffer Unit is disabled.
- Selects an implemented Security state as the owning Security state.
- When trace data is received from the trace unit, it is not written to memory and the Trace Buffer Unit generates a trace buffer management event, as follows:
  - TRBSR_ELx.IRQ is set to 1.
  - If TRBSR_ELx.S is 0, then all of the following occur:
    - TRBSR_ELx.S is set to 1, Collection is stopped.
    - TRBSR_ELx.EC is set to `0x00`, Other buffer management event.
    - TRBSR_ELx.BSC is set to `0b000000`, Access not allowed.
  - The other fields in TRBSR_ELx are unchanged.

If the Trace Buffer Unit is enabled and using Self-hosted mode, and MDCR_EL3.{NSTB, NSTBE} selects a

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

reserved value, then it is CONSTRAINED UNPREDICTABLE whether or not accesses to trace buffer System registers at EL2 and EL1 generate Trap exceptions to EL3.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 1, then the Effective value of this field is `0b11`.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is `0b01`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TRBE is implemented:

Non-secure Trace Buffer. Controls the trace buffer owning Security state and accesses to trace buffer System registers from EL2 and EL1.

NSTB	Meaning
`0b00`	Trace buffer owning Security state is Secure state. When `TraceBufferEnabled()`==TRUE, tracing is prohibited in Non-secure state. Accesses to trace buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception.
`0b01`	Trace buffer owning Security state is Secure state. When `TraceBufferEnabled()`==TRUE, tracing is prohibited in Non-secure state. Accesses to trace buffer System registers at EL2 and EL1 in Non-secure state are trapped to EL3, unless the access generates a higher priority exception.
`0b10`	Trace buffer owning Security state is Non-secure state. When `TraceBufferEnabled()`==TRUE, tracing is prohibited in Secure state. Accesses to trace buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception.
`0b11`	Trace buffer owning Security state is Non-secure state. When `TraceBufferEnabled()`==TRUE, tracing is prohibited in Secure state. Accesses to trace buffer System registers at EL2 and EL1 in Secure state are trapped to EL3, unless the access generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to TRBBASER_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBPTR_EL1, TRBSR_EL1, and TRBTRG_EL1.
- If FEAT_TRBE_MPAM is implemented, `MRS` and `MSR` accesses to TRBMPAM_EL1.
- If FEAT_TRBE_EXC is implemented, `MRS` and `MSR` accesses to TRBSR_EL2 and TRBSR_EL12.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3, reported using EC syndrome value `0x18`.

When `TraceBufferEnabled()`==FALSE, this control has no effect on whether tracing is prohibited.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 1, then the Effective value of this field is `0b11`.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is `0b01`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCCD, bit [23]

When FEAT_PMUv3p5 is implemented:

Secure Cycle Counter Disable. Prohibits PMCCNTR_EL0 from counting in Secure state and EL3.

SCCD	Meaning
0b0	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
0b1	Cycle counting by PMCCNTR_EL0 is prohibited in Secure state and EL3.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

ETAD, bit [22]

When FEAT_RME is implemented, FEAT_TRC_EXT is implemented, and FEAT_TRBE is implemented:

External Trace Access Disable. Together with MDCR_EL3.ETADE, controls access to trace unit registers by an external debugger.

ETADE	ETAD	Meaning
0b0	0b0	No accesses from an external debugger to trace unit registers are prohibited by this control.
0b0	0b1	Realm and Non-secure accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected by this control.
0b1	0b0	Secure and Non-secure accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected by this control.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected by this control.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

When FEAT_TRC_EXT is implemented and FEAT_TRBE is implemented:

External Trace Access Disable. Controls Non-secure access to trace unit registers by an external debugger.

ETAD	Meaning
0b0	No accesses from an external debugger to the trace unit registers are prohibited by this control.
0b1	Non-secure accesses from an external debugger to some trace unit registers are prohibited. See individual registers for the effect of this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EPMAD, bit [21]

When FEAT_RME is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable. Together with MDCR_EL3.EPMADE, controls access to Performance Monitor registers by an external debugger.

External accesses of the following Performance Monitor registers are affected by this control:

- PMCCFILTR_EL0, PMCCNTR_EL0, PMCFGR, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMINTENCLR_EL1, PMINTENSET_EL1, PMOVSCLR_EL0, and PMOVSSET_EL0.
- If PMEVFILT2R<n> is implemented, PMEVFILT2R<n>.
- If implemented, PMIIDR.
- If PMSWINC_EL0 is implemented in the external view, PMSWINC_EL0.
- If FEAT_PMUv3p4 is implemented, PMMIR.
- If FEAT_PMUv3_EXT64 is implemented, PMCNTEN, PMINTEN, and PMOVS.
- If FEAT_PMUv3p9 is implemented, PMZR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, PMCGCR0, PMICFILTR_EL0, and PMICNTR_EL0.
- If FEAT_PCSRv8p9 is implemented, PMPCSCTL.

EPMADE	EPMAD	Meaning
0b0	0b0	No accesses from an external debugger to affected Performance Monitor registers are prohibited by this control.
0b0	0b1	Realm and Non-secure accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected by this control.
0b1	0b0	Secure and Non-secure accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected by this control.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected by this control.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable. Controls Non-secure access to Performance Monitor registers by an external debugger.

External accesses of the following Performance Monitor registers are affected by this control:

- PMCCFILTR_EL0, PMCCNTR_EL0, PMCFGR, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMINTENCLR_EL1, PMINTENSET_EL1, PMOVSCLR_EL0, and PMOVSSET_EL0.
- If PMEVFILT2R<n> is implemented, PMEVFILT2R<n>.
- If implemented, PMIIDR.
- If PMSWINC_EL0 is implemented in the external view, PMSWINC_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_PMUv3p4 is implemented, PMMIR.
- If FEAT_PMUv3_EXT64 is implemented, PMCNTEN, PMINTEN, and PMOVS.
- If FEAT_PMUv3p9 is implemented, PMZR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, PMCGCR0, PMICFILTR_EL0, and PMICNTR_EL0.
- If FEAT_PCSRv8p9 is implemented, PMPCSCTL.

EPMAD	Meaning
`0b0`	No accesses from an external debugger to the Performance Monitor registers are prohibited by this control.
`0b1`	Non-secure accesses from an external debugger to the affected Performance Monitor registers are prohibited.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable. Controls access to Performance Monitor registers by an external debugger.

External accesses of the following Performance Monitor registers are affected by this control:

- PMCCFILTR_EL0, PMCCNTR_EL0, PMCFGR, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMINTENCLR_EL1, PMINTENSET_EL1, PMOVSCLR_EL0, and PMOVSSET_EL0.
- If PMEVFILT2R<n> is implemented, PMEVFILT2R<n>.
- If implemented, PMIIDR.
- If PMSWINC_EL0 is implemented in the external view, PMSWINC_EL0.
- If FEAT_PMUv3p4 is implemented, PMMIR.

EPMAD	Meaning
`0b0`	No accesses from an external debugger to the Performance Monitor registers are prohibited by this control.
`0b1`	If the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()` returns FALSE, then accesses from an external debugger to the affected Performance Monitor registers are prohibited.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EDAD, bit [20]

When FEAT_RME is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

External Debug Access Disable. Together with MDCR_EL3.EDADE, controls access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger.

External accesses of the following debug registers are affected by this control:

- DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1, and OSLAR_EL1.

EDADE	EDAD	Meaning
0b0	0b0	No accesses from an external debugger to affected debug registers are prohibited by this control.
0b0	0b1	Realm and Non-secure accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected by this control.
0b1	0b0	Secure and Non-secure accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected by this control.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected by this control.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When FEAT_Debugv8p4 is implemented:

External Debug Access Disable. Controls Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger.

External accesses of the following debug registers are affected by this control:

- DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1, and OSLAR_EL1.

EDAD	Meaning
0b0	No accesses from an external debugger to the debug registers are prohibited by this control.
0b1	Non-secure accesses from an external debugger to the affected debug registers are prohibited.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When FEAT_Debugv8p2 is implemented:

External Debug Access Disable. Controls access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger.

External accesses of the following debug registers are affected by this control:

- DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1, and OSLAR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDAD	Meaning
0b0	No accesses from an external debugger to the debug registers are prohibited by this control.
0b1	If the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()` returns FALSE, then accesses from an external debugger to the affected debug registers are prohibited.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

External Debug Access Disable. Controls access to breakpoint registers, watchpoint registers, and optionally OSLAR_EL1 by an external debugger.

External accesses of the following debug registers are affected by this control:

- DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, and DBGWVR<n>_EL1.
- It is IMPLEMENTATION DEFINED whether this control affects OSLAR_EL1.

EDAD	Meaning
0b0	No accesses from an external debugger to the debug registers are prohibited by this control.
0b1	If the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()` returns FALSE, then accesses from an external debugger to the affected debug registers are prohibited.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

TTRF, bit [19]

When FEAT_TRF is implemented:

Trap Trace Filter controls. Traps use of the Trace Filter control registers at EL2 and EL1 to EL3.

The Trace Filter registers trapped by this control are:

- TRFCR_EL2, TRFCR_EL12, TRFCR_EL1, reported using EC syndrome value `0x18`.
- HTRFCR and TRFCR, reported using EC syndrome value `0x03`.

TTRF	Meaning
0b0	Accesses to Trace Filter registers at EL2 and EL1 are not affected by this field.
0b1	Accesses to Trace Filter registers at EL2 and EL1 generate a Trap exception to EL3, unless the access generates a higher priority exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

STE, bit [18]

When FEAT_TRF is implemented and HaveSecureState():

Secure Trace enable. Enables tracing in Secure state.

STE	Meaning
0b0	Trace prohibited in Secure state unless overridden by the IMPLEMENTATION DEFINED authentication interface.
0b1	Trace in Secure state is not affected by this field.

This field also controls the level of authentication required by an external debugger to enable external tracing. See 'Register controls to enable self-hosted trace'.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

SPME, bit [17]

When FEAT_PMUv3 is implemented and FEAT_PMUv3p7 is implemented:

Secure Performance Monitors Enable. Controls PMU operation in Secure state and at EL3 when MDCR_EL3.MPMX is 0.

SPME	Meaning
0b0	Affected counters are prohibited from counting in Secure state and at EL3. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled in Secure state and at EL3. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.
0b1	Counters are not affected by this mechanism.

When MDCR_EL3.MPMX is 0, the counters affected by this field are:

- Event counters in the first and second ranges. For more information about event counter ranges, see MDCR_EL2.HPMN.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter, PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter, PMCCNTR_EL0.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

When MDCR_EL3.MPMX is 1, this field controls which event counters are affected by MDCR_EL3.MPMX at EL3. See MDCR_EL3.MPMX for more information.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

When FEAT_PMUv3 is implemented and FEAT_Debugv8p2 is implemented:

Secure Performance Monitors Enable. Controls PMU operation in Secure state.

SPME	Meaning
`0b0`	Event counting is prohibited in Secure state. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled in Secure state. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Event counting and PMCCNTR_EL0 are not affected by this mechanism.

The counters affected by this field are:

- All event counters.
- If PMCR_EL0.DP is 1, the cycle counter, PMCCNTR_EL0.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When FEAT_PMUv3 is implemented:

Secure Performance Monitors Enable. Controls PMU operation in Secure state.

SPME	Meaning
`0b0`	If `ExternalSecureNoninvasiveDebugEnabled()` is FALSE, then all the following apply: • Event counting is prohibited in Secure state. • If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled in Secure state. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Event counting and PMCCNTR_EL0 are not affected by this mechanism.

If `ExternalSecureNoninvasiveDebugEnabled()` is TRUE, then the event counters and PMCCNTR_EL0 are not affected by this field.

Otherwise, the counters affected by this field are:

- All event counters.
- If PMCR_EL0.DP is 1, the cycle counter, PMCCNTR_EL0.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SDD, bit [16]

When HaveSecureState():

AArch64 Secure Self-hosted invasive debug disable. Disables Software debug exceptions in Secure state, other than Breakpoint Instruction exceptions.

SDD	Meaning
0b0	Debug exceptions in Secure state are not affected by this field.
0b1	Debug exceptions, other than Breakpoint Instruction exceptions, are disabled from all Exception levels in Secure state.

The SDD bit is ignored unless both of the following are true:

- The PE is in Secure state.
- The Effective value of SCR_EL3.RW is 0.

If Secure EL2 is implemented and enabled, and Secure EL1 is using AArch32, then:

- If debug exceptions from Secure EL1 are enabled, debug exceptions from Secure EL0 are also enabled.
- Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER32_EL3.SUIDEN is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SPD32, bits [15:14]

When FEAT_AA32EL1 is implemented:

AArch32 Secure self-hosted privileged debug. Enables or disables debug exceptions from Secure EL1 using AArch32, other than Breakpoint Instruction exceptions.

SPD32	Meaning
0b00	Legacy mode. Debug exceptions from Secure EL1 are enabled by the IMPLEMENTATION DEFINED authentication interface.
0b10	Secure privileged debug disabled. Debug exceptions from Secure EL1 are disabled.
0b11	Secure privileged debug enabled. Debug exceptions from Secure EL1 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as 0b00. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Breakpoint Instruction exceptions. These are always enabled.

This field is ignored unless both of the following are true:

- The PE is in Secure state.
- The Effective value of SCR_EL3.RW is 0.

If Secure EL1 is using AArch32, then:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If debug exceptions from Secure EL1 are enabled, then debug exceptions from Secure EL0 are also enabled.
- Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER32_EL3.SUIDEN is 1.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is `0b11`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSPB, bits [13:12]

When FEAT_SPE is implemented and FEAT_RME is implemented:

Non-secure Profiling Buffer. Together with MDCR_EL3.NSPBE, controls the Profiling Buffer owning Security state and accesses to Statistical Profiling and Profiling Buffer System registers from EL2 and EL1.

NSPBE	NSPB	Meaning
`0b0`	`0b00`	Profiling Buffer owning Security state is Secure state. Profiling is disabled in Realm and Non-secure states. Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception. When Secure state is not implemented, this encoding is reserved.
`0b0`	`0b01`	Profiling Buffer owning Security state is Secure state. Profiling is disabled in Realm and Non-secure states. Accesses to Statistical Profiling and Profiling Buffer System registers at Realm and Non-secure EL2, and Realm and Non-secure EL1, are trapped to EL3, unless the access generates a higher priority exception. When Secure state is not implemented, this encoding is reserved.
`0b0`	`0b10`	Profiling Buffer owning Security state is Non-secure state. Profiling is disabled in Secure and Realm states. Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception.
`0b0`	`0b11`	Profiling Buffer owning Security state is Non-secure state. Profiling is disabled in Secure and Realm states. Accesses to Statistical Profiling and Profiling Buffer System registers at Secure and Realm EL2, and Secure and Realm EL1, are trapped to EL3, unless the access generates a higher priority exception.
`0b1`	`0b10`	Profiling Buffer owning Security state is Realm state. | Profiling is disabled in Secure and Non-secure states. | Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 are | trapped to EL3, unless the access generates a higher priority exception.
`0b1`	`0b11`	Profiling Buffer owning Security state is Realm state. Profiling is disabled in Secure and Non-secure states. Accesses to Statistical Profiling and Profiling Buffer System registers at Secure and Non-secure EL2, and Secure and Non-secure EL1, are trapped to EL3, unless the access generates a higher priority exception.

All other combinations of MDCR_EL3.NSPBE and MDCR_EL3.NSPB are reserved.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to PMBLIMITR_EL1, PMBPTR_EL1, PMBSR_EL1, PMSCR_EL1, PMSCR_EL2, PMSCR_EL12, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIRR_EL1, and PMSLATFR_EL1.
- `MRS` accesses to PMSIDR_EL1.
- If FEAT_SPE_FnE is implemented, `MRS` and `MSR` accesses to PMSNEVFR_EL1.
- If FEAT_SPE_FDS is implemented, `MRS` and `MSR` accesses to PMSDSFR_EL1.
- If FEAT_SPE_EXC is implemented, `MRS` and `MSR` accesses to PMBSR_EL2 and PMBSR_EL12.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_SPE_nVM is implemented, MRS and MSR accesses to PMBMAR_EL1.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3, reported using EC syndrome value 0x18.

If profiling is enabled and MDCR_EL3.{NSPB, NSPBE} selects a reserved value, then the behavior is CONSTRAINED UNPREDICTABLE, and the Statistical Profiling Unit does one of:

- Behaves as if profiling is disabled.
- Selects an implemented Security state as the owning Security state.
- When profiling data is generated, it is not written to memory and the Statistical Profiling Unit generates a Profiling Buffer management event, as follows:
  - If PMBSR_ELx.S is 0, then all of the following occur:
    - PMBSR_ELx.S is set to 1.
    - PMBSR_ELx.DL is set to 1.
    - PMBSR_ELx.EC is set to 0b000000, Other buffer management event.
    - PMBSR_ELx.BSC is set to 0b000000, Access not allowed.
  - The other fields in PMBSR_ELx are unchanged.

If profiling is enabled and MDCR_EL3.{NSPB, NSPBE} selects a reserved value, then it is CONSTRAINED UNPREDICTABLE whether or not accesses to the Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 generate Trap exceptions to EL3.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 1, then the Effective value of this field is 0b11.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 0b01.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPE is implemented:

Non-secure Profiling Buffer. Controls the Profiling Buffer owning Security state and accesses to Statistical Profiling and Profiling Buffer System registers from EL2 and EL1.

NSPB	Meaning
0b00	Profiling Buffer owning Security state is Secure state. Profiling is disabled in Non-secure state. Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception.
0b01	Profiling Buffer owning Security state is Secure state. Profiling is disabled in Non-secure state. Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 in Non-secure state are trapped to EL3, unless the access generates a higher priority exception.
0b10	Profiling Buffer owning Security state is Non-secure state. Profiling is disabled in Secure state. Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 are trapped to EL3, unless the access generates a higher priority exception.
0b11	Profiling Buffer owning Security state is Non-secure state. Profiling is disabled in Secure state. Accesses to Statistical Profiling and Profiling Buffer System registers at EL2 and EL1 in Secure state are trapped to EL3, unless the access generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to PMBLIMITR_EL1, PMBPTR_EL1, PMBSR_EL1, PMSCR_EL1, PMSCR_EL2, PMSCR_EL12, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIRR_EL1, and PMSLATFR_EL1.
- MRS accesses to PMSIDR_EL1.
- If FEAT_SPE_FnE is implemented, MRS and MSR accesses to PMSNEVFR_EL1.
- If FEAT_SPE_FDS is implemented, MRS and MSR accesses to PMSDSFR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_SPE_EXC is implemented, MRS and MSR accesses to PMBSR_EL2 and PMBSR_EL12.
- If FEAT_SPE_nVM is implemented, MRS and MSR accesses to PMBMAR_EL1.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3, reported using EC syndrome value 0x18.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 1, then the Effective value of this field is 0b11.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 0b01.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSPBE, bit [11]

When FEAT_SPE is implemented and FEAT_RME is implemented:

Non-secure Profiling Buffer Extended. Together with MDCR_EL3.NSPB, controls the Profiling Buffer owning Security state and accesses to Statistical Profiling and Profiling Buffer System registers from EL2 and EL1.

For a description of the values derived by evaluating NSPB and NSPBE together, see MDCR_EL3.NSPB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDOSA, bit [10]

When FEAT_DoubleLock is implemented:

Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug System registers to EL3.

Accesses to the registers are trapped as follows:

- Accesses from AArch64 state, OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, DBGPRCR_EL1, and any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this field, are trapped to EL3 and reported using EC syndrome value 0x18.
- Accesses using MCR or MRC to DBGOSLAR, DBGOSLSR, DBGOSDLR, and DBGPRCR, are trapped to EL3 and reported using EC syndrome value 0x05.
- Accesses to any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this field.

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL2 and EL1 System register accesses to the powerdown debug System registers are trapped to EL3, unless it is trapped by HDCR.TDOSA or MDCR_EL2.TDOSA.

Note

The powerdown debug System registers are not accessible at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug System registers to EL3.

The following registers are affected by this trap:

- AArch64: OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
- AArch32: DBGOSLAR, DBGOSLSR, and DBGPRCR.
- AArch64 and AArch32: Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this field.
- It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 and DBGOSDLR are trapped.

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL2 and EL1 System register accesses to the powerdown debug System registers are trapped to EL3, unless it is trapped by HDCR.TDOSA or MDCR_EL2.TDOSA.

Note

The powerdown debug System registers are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap accesses of debug System registers. Enables a trap to EL3 on accesses of debug System registers.

TDA	Meaning
0b0	Accesses of the specified debug System registers are not trapped by this mechanism.
0b1	Accesses of the specified debug System registers at EL2, EL1, and EL0 are trapped to EL3, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to DBGAUTHSTATUS_EL1, DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1, DBGVCR32_EL2, DBGWCR<n>_EL1, DBGWVR<n>_EL1, MDCCINT_EL1, MDCCSR_EL0, MDCR_EL2, MDRAR_EL1, MDSCR_EL1, OSDTRRX_EL1, OSDTRTX_EL1, and OSECCR_EL1.
- If FEAT_Debugv8p9 is implemented, MRS and MSR accesses to MDSELR_EL1.
- If FEAT_STEP2 is implemented, MRS and MSR accesses to MDSTEPOP_EL1.
- In Non-debug state, MRS accesses to DBGDTRRX_EL0 and DBGDTR_EL0 and MSR accesses to DBGDTRTX_EL0 and DBGDTR_EL0.

In AArch32 state, the instructions affected by this control are:

- MRC and MCR accesses to DBGAUTHSTATUS, DBGBCR<n>, DBGBVR<n>, DBGBXVR<n>, DBGCLAIMCLR, DBGCLAIMSET, DBGDCCINT, DBGDEVID, DBGDEVID1, DBGDEVID2, DBGDIDR, DBGDRAR, DBGDSAR, DBGDSCRext, DBGDSCRint, DBGDTRRXext, DBGDTRTXext, DBGOSECCR, DBGVCR, DBGWCR<n>, DBGWFAR, DBGWVR<n>, HDCR, and SDER.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MRRC accesses to DBGDRAR and DBGDSAR.
- STC accesses to DBGDTRRXint and LDC accesses to DBGDTRTXint.
- In Non-debug state, MRC accesses to DBGDTRRXint and MCR accesses to DBGDTRTXint.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Trapped AArch32 instructions are reported using EC syndrome value 0x03 for MRC and MCR accesses with coproc == 0b1111, 0x05 for MCR and MCR accesses with coproc == 0b1110, 0x06 for LDC and STC accesses, and 0x0C for MRRC accesses.

The following instructions are not trapped in Debug state:

- AArch64 MRS accesses to DBGDTRRX_EL0 and DBGDTR_EL0 and MSR accesses to DBGDTRTX_EL0 and DBGDTR_EL0.
- AArch32 MRC accesses to DBGDTRRXint and MCR accesses to DBGDTRTXint.

If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI, then it is IMPLEMENTATION DEFINED whether AArch64 accesses to MDSELR_EL1 are trapped to EL3 when MDCR_EL3.TDA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

EnPM2, bit [7]

When FEAT_PMUv3p9 is implemented, or FEAT_SPMU is implemented, or FEAT_EBEP is implemented, or FEAT_PMUv3_SS is implemented, or FEAT_SPMU2 is implemented:

Enable access to PMU registers. When disabled, accesses to PMU registers generate a trap to EL3.

EnPM2	Meaning
0b0	Accesses of the specified PMU registers at EL2, EL1, and EL0 are trapped to EL3, unless the instruction generates a higher priority exception. If FEAT_PMUv3_ICNTR is implemented, then: • PMCNTENCLR_EL0.F0, PMCNTENSET_EL0.F0, PMOVSCLR_EL0.F0, PMOVSSET_EL0.F0, and PMZR_EL0.F0 read-as-zero and ignore writes at EL2, EL1, and EL0. • PMINTENCLR_EL1.F0 and PMINTENSET_EL1.F0 read-as-zero and ignore writes at EL2 and EL1.
0b1	Accesses of the specified PMU registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are:

- If FEAT_EBEP is implemented or FEAT_PMUv3_SS is implemented, MRS and MSR accesses to PMECR_EL1.
- If FEAT_PMUv3_ICNTR is implemented, MRS and MSR accesses to PMICFILTR_EL0 and PMICNTR_EL0.
- If FEAT_PMUv3p9 is implemented, MRS and MSR accesses to PMUACR_EL1.
- If FEAT_SPMU is implemented, MRS and MSR accesses to SPMACCESSR_EL1, SPMACCESSR_EL2, SPMACCESSR_EL12, SPMCFGR_EL1, SPMCGCR<n>_EL1, SPMCNTENCLR_EL0, SPMCNTENSET_EL0, SPMCR_EL0, SPMDEVAFF_EL1, SPMDEVARCH_EL1, SPMEVCNTR<n>_EL0, SPMEVFILT2R<n>_EL0, SPMEVFILTR<n>_EL0, SPMEVTYPER<n>_EL0, SPMIIDR_EL1, SPMINTENCLR_EL1, SPMINTENSET_EL1, SPMOVSCLR_EL0, SPMOVSSET_EL0, SPMSCR_EL1, and SPMSELR_EL0.
- If FEAT_SPMU2 is implemented, MSR writes to SPMZR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trapped instructions are reported using EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When FEAT_PMUv3 is implemented:

Trap accesses of PMU registers. Enables a trap to EL3 on accesses of PMU registers.

TPM	Meaning
`0b0`	Accesses of the specified PMU registers are not trapped by this mechanism.
`0b1`	Accesses of the specified PMU registers at EL2, EL1, and EL0 are trapped to EL3, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMINTENCLR_EL1, PMINTENSET_EL1, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMSWINC_EL0, PMUSERENR_EL0, PMXEVCNTR_EL0, and PMXEVTYPER_EL0.
- `MRS` accesses to PMCEID0_EL0 and PMCEID1_EL0.
- If FEAT_PMUv3p4 is implemented, `MRS` accesses to PMMIR_EL1.
- If FEAT_PMUv3p9 is implemented, `MSR` accesses to PMZR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, `MRS` and `MSR` accesses to PMICFILTR_EL0 and PMICNTR_EL0.
- If FEAT_EBEP is implemented or FEAT_PMUv3_SS is implemented, `MRS` and `MSR` accesses to PMECR_EL1.

In AArch32 state, the instructions affected by this control are:

- `MRC` and `MCR` accesses to PMCCFILTR, PMCCNTR, PMCEID0, PMCEID1, PMCNTENCLR, PMCNTENSET, PMCR, PMEVCNTR<n>, PMEVTYPER<n>, PMINTENCLR, PMINTENSET, PMOVSR, PMOVSSET, PMSELR, PMSWINC, PMUSERENR, PMXEVCNTR, and PMXEVTYPER.
- `MRRC` and `MCRR` accesses to PMCCNTR.
- If FEAT_PMUv3p1 is implemented, `MRC` accesses to PMCEID2 and PMCEID3.
- If FEAT_PMUv3p4 is implemented, `MRC` accesses to PMMIR.

Trapped AArch64 instructions are reported using EC syndrome value `0x18`.

Trapped AArch32 instructions are reported using EC syndrome value `0x03` for `MRC` and `MCR` accesses, and `0x04` for `MRRC` and `MCRR` accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDADE, bit [4]

When FEAT_RME is implemented:

External Debug Access Disable Extended. Together with MDCR_EL3.EDAD, controls access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger.

For a description of the values derived by evaluating MDCR_EL3.EDAD and MDCR_EL3.EDADE together, see MDCR_EL3.EDAD.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

ETADE, bit [3]

When FEAT_RME is implemented, FEAT_TRC_EXT is implemented, and FEAT_TRBE is implemented:

External Trace Access Disable Extended. Together with MDCR_EL3.ETAD, controls access to trace unit registers by an external debugger.

For a description of the values derived by evaluating MDCR_EL3.ETAD and MDCR_EL3.ETADE together, see MDCR_EL3.ETAD.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EPMADE, bit [2]

When FEAT_RME is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable Extended. Together with MDCR_EL3.EPMAD, controls access to Performance Monitor registers by an external debugger.

For a description of the values derived by evaluating MDCR_EL3.EPMAD and MDCR_EL3.EPMADE together, see MDCR_EL3.EPMAD.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [1]

Reserved, RES0.

RLTE, bit [0]

When FEAT_RME is implemented and FEAT_TRF is implemented:

Realm Trace enable. Enables tracing in Realm state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLTE	Meaning
0b0	Trace prohibited in Realm state, unless overridden by the IMPLEMENTATION DEFINED authentication interface.
0b1	Trace in Realm state is not affected by this field.

This field also controls the level of authentication that is required by an external debugger to enable external tracing.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Accessing MDCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0011	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDCR_EL3();
end;
```

```
MSR MDCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0011	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().MDCR_EL3 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        MDCR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.19 MDRAR_EL1, Monitor Debug ROM Address Register

The MDRAR_EL1 characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Use of this register is deprecated.

Configuration

AArch64 System register MDRAR_EL1 bits [63:0] are architecturally mapped to AArch32 System register DBGDRAR[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MDRAR_EL1 are UNDEFINED.

Attributes

MDRAR_EL1 is a 64-bit register.

Field descriptions

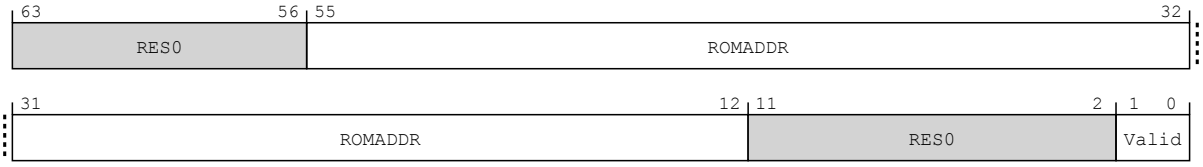

Bits [63:56]

Reserved, RES0.

ROMADDR, bits [55:12]

When FEAT_D128 is implemented and MDRAR_EL1.Valid != '00':

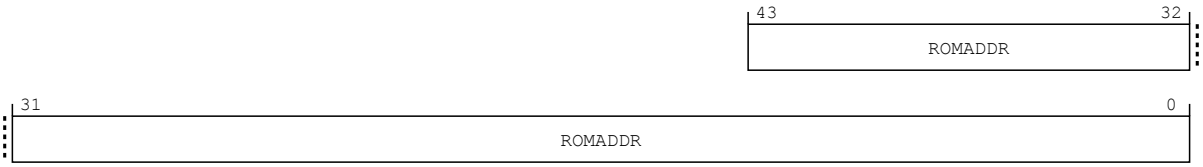

ROMADDR, bits [43:0]

Bits [55:12] of the ROM table physical address.

Bits [11:0] of the ROM table physical address are zero.

This field has an IMPLEMENTATION DEFINED value.

For implementations with fewer than 56 physical address bits, the corresponding upper bits of this field are RES0

In an implementation that includes EL3, ROMADDR is an address in Non-secure PA space. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure PA space. If FEAT_RME is

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

implemented, it is IMPLEMENTATION DEFINED whether the ROM table is also accessible in the Root or Realm PA spaces.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

Access to this field is RO.

When FEAT_D128 is not implemented, FEAT_LPA is implemented, and MDRAR_EL1.Valid != '00':

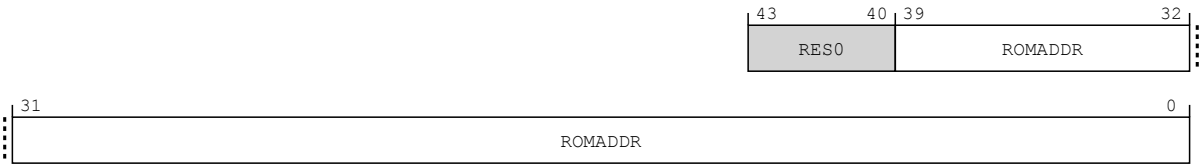

Bits [43:40]

Reserved, RES0.

ROMADDR, bits [39:0]

Bits [51:12] of the ROM table physical address.

Bits [11:0] of the ROM table physical address are zero.

This field has an IMPLEMENTATION DEFINED value.

For implementations with fewer than 52 physical address bits, the corresponding upper bits of this field are RES0

In an implementation that includes EL3, ROMADDR is an address in Non-secure PA space. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure PA space. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether the ROM table is also accessible in the Root or Realm PA spaces.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

Access to this field is RO.

When FEAT_D128 is not implemented, FEAT_LPA is not implemented, and MDRAR_EL1.Valid != '00':

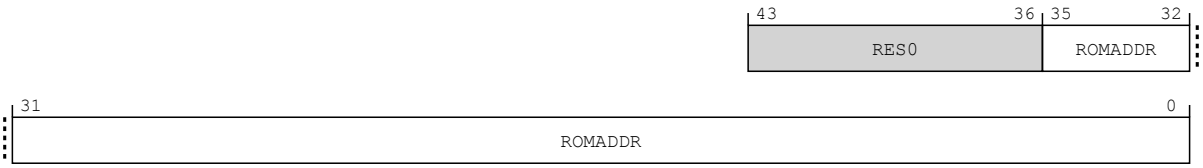

Bits [43:36]

Reserved, RES0.

ROMADDR, bits [35:0]

Bits [39:12] of the ROM table physical address.

Bits [11:0] of the ROM table physical address are zero.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For implementations with fewer than 48 physical address bits, the corresponding upper bits of this field are RES0

In an implementation that includes EL3, ROMADDR is an address in Non-secure PA space. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure PA space. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Root or Realm PA spaces.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

Access to this field is RO.

When MDRAR_EL1.Valid == '00':

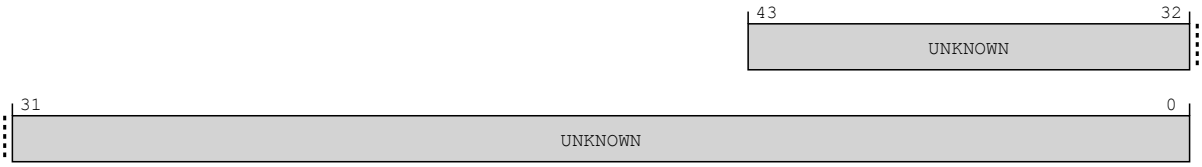

Bits [43:0]

Reserved, UNKNOWN.

Bits [11:2]

Reserved, RES0.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Valid	Meaning
`0b00`	ROM Table address is not valid. Software must ignore ROMADDR.
`0b11`	ROM Table address is valid.

Other values are reserved.

Arm recommends implementations set this field to zero.

Access to this field is RO.

Accessing MDRAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDRAR_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDRA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDRAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDRAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDRAR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.20 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

AArch64 System register MDSCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDSCRext[31:0].

AArch64 System register MDSCR_EL1 bit [15] is architecturally mapped to AArch32 System register DBGDSCRint[15].

AArch64 System register MDSCR_EL1 bit [12] is architecturally mapped to AArch32 System register DBGDSCRint[12].

AArch64 System register MDSCR_EL1 bits [5:2] are architecturally mapped to AArch32 System register DBGDSCRint[5:2].

AArch64 System register MDSCR_EL1 bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to External register EDSCR[31:29, 27:26, 23:21, 19, 14, 6].

AArch64 System register MDSCR_EL1 bits [35, 33] are architecturally mapped to External register EDSCR2[3, 1].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to MDSCR_EL1 are UNDEFINED.

Attributes

MDSCR_EL1 is a 64-bit register.

Field descriptions

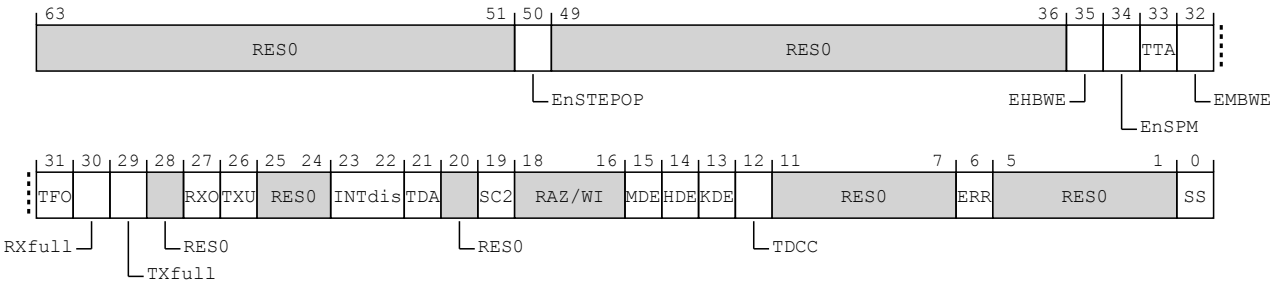

Bits [63:51]

Reserved, RES0.

EnSTEPOP, bit [50]

When FEAT_STEP2 is implemented:

Software step control bit. Enable execution from MDSTEPOP_EL1. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnSTEPOP	Meaning
0b0	Execution from MDSTEPOP_EL1 is disabled.
0b1	Execution from MDSTEPOP_EL1 is not disabled by this control.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [49:36]

Reserved, RES0.

EHBWE, bit [35]

When FEAT_Debugv8p9 is implemented:

Extended Halting Breakpoint and Watchpoint Enable. Used for save/restore of EDSCR2.EHBWE.

When OSLSR_EL1.OSLK is 0, software must treat this field as UNK/SBZP.

When OSLSR_EL1.OSLK is 1, this field holds the value of EDSCR2.EHBWE. Reads and writes of this field are indirect accesses to EDSCR2.EHBWE.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

EnSPM, bit [34]

When FEAT_SPMU is implemented:

Enable access to System PMU registers. When disabled, accesses to System PMU registers generate a trap to EL1.

EnSPM	Meaning
0b0	Accesses of the specified System PMU registers at EL0 are trapped to EL1, unless the instruction generates a higher priority exception.
0b1	Accesses of the specified System PMU registers are not trapped by this mechanism.

In AArch64 state, the instructions affected by this control are: MRS and MSR accesses to SPMCNTENCLR_EL0, SPMCNTENSET_EL0, SPMCR_EL0, SPMEVCNTR<n>_EL0, SPMEVFILT2R<n>_EL0, SPMEVFILTR<n>_EL0, SPMEVTYPER<n>_EL0, SPMOVSCLR_EL0, SPMOVSSET_EL0, and SPMSELR_EL0.

Unless the instruction generates a higher priority exception:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL2 is implemented and enabled in the current Security state, and HCR_EL2.TGE is 1, then trapped instructions generate an exception to EL2.
- Otherwise, trapped instructions generate an exception to EL1.

Trapped instructions are reported using EC syndrome value `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TTA, bit [33]

When FEAT_TRBE_EXT is implemented or FEAT_ETEv1p3 is implemented:

Trap Trace Accesses. Used for save/restore of EDSCR2.TTA.

When OSLSR_EL1.OSLK is 0, software must treat this field as UNK/SBZP.

When OSLSR_EL1.OSLK is 1, this field holds the value of EDSCR2.TTA. Reads and writes of this field are indirect accesses to EDSCR2.TTA.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

EMBWE, bit [32]

When FEAT_Debugv8p9 is implemented:

Extended Monitor Breakpoint and Watchpoint Enable. Enables use of additional breakpoints or watchpoints.

EMBWE	Meaning
`0b0`	Breakpoint and Watchpoint exceptions are disabled for each breakpoint <n> and watchpoint <n>, where n is greater than or equal to 16. The Effective value of MDSELR_EL1.BANK is zero at EL1.
`0b1`	Breakpoint and Watchpoint exceptions are not affected by this mechanism. The Effective value of MDSELR_EL1.BANK is not affected by this field.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

This field is ignored by the PE and treated as zero when any of the following are true:

- EL3 is implemented and MDCR_EL3.EBWE is 0.
- EL2 is implemented and enabled in the current Security state, and MDCR_EL2.EBWE is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter override. Used for save/restore of EDSCR.TFO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

Used for save/restore of EDSCR.RXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

TXfull, bit [29]

Used for save/restore of EDSCR.TXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Bit [28]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

When OSLSR_EL1.OSLK == 1, if bits [27,6] of the value written to MDSCR_EL1 are {1,0}, that is, the RXO bit is 1 and the ERR bit is 0, the PE sets EDSCR.{RXO,ERR} to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

When OSLSR_EL1.OSLK == 1, if bits [26,6] of the value written to MDSCR_EL1 are {1,0}, that is, the TXU bit is 1 and the ERR bit is 0, the PE sets EDSCR.{TXU,ERR} to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of EDSCR.INTdis.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

TDA, bit [21]

Used for save/restore of EDSCR.TDA.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Bit [20]

Reserved, RES0.

SC2, bit [19]

When FEAT_PCSRv8 is implemented, FEAT_VHE is implemented, and FEAT_PCSRv8p2 is not implemented:

Used for save/restore of EDSCR.SC2.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.SC2. Reads and writes of this bit are indirect accesses to EDSCR.SC2.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [18:16]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

MDE, bit [15]

Monitor debug events. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

MDE	Meaning
`0b0`	Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
`0b1`	Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

KDE, bit [13]

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D. Permitted values are:

KDE	Meaning
`0b0`	Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.
`0b1`	All debug exceptions enabled within EL_D.

RES0 if EL_D is using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDCC, bit [12]

Traps EL0 accesses to the Debug Communication Channel (DCC) registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, as follows:

- In AArch64 state, MRS or MSR accesses to the following DCC System registers are trapped, reported using EC syndrome value `0x18`:
  - MDCCSR_EL0.
  - If not in Debug state, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped, reported using EC syndrome value `0x05`.
  - DBGDSCRint, DBGDIDR, DBGDSAR, DBGDRAR.
  - If not in Debug state, DBGDTRRXint, and DBGDTRTXint.
- In AArch32 state, LDC access to DBGDTRRXint and STC access to DBGDTRTXint are trapped, reported using EC syndrome value `0x06`.
- In AArch32 state, MRRC accesses to DBGDSAR and DBGDRAR are trapped, reported using EC syndrome value `0x0C`.

TDCC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 using AArch64: EL0 accesses to the AArch64 DCC System registers are trapped. EL0 using AArch32: EL0 accesses to the AArch32 DCC System registers are trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.ERR. Reads and writes of this bit are indirect accesses to EDSCR.ERR.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Bits [5:1]

Reserved, RES0.

SS, bit [0]

Software step control bit. If EL_D is using AArch64, enable Software step. Permitted values are:

SS	Meaning
0b0	Software step disabled
0b1	Software step enabled.

RES0 if EL_D is using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MDSCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDSCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().MDSCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x158);
    else
        X{64}(t) = MDSCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDSCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDSCR_EL1();
end;
```

MSR MDSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().MDSCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x158) = X{64}(t);
    else
        MDSCR_EL1() = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MDSCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.21 MDSELR_EL1, Breakpoint and Watchpoint Selection Register

The MDSELR_EL1 characteristics are:

Purpose

Selects the current breakpoints or watchpoints accessed by System register instructions.

Configuration

This register is present only when FEAT_Debugv8p9 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MDSELR_EL1 are UNDEFINED.

Attributes

MDSELR_EL1 is a 64-bit register.

Field descriptions

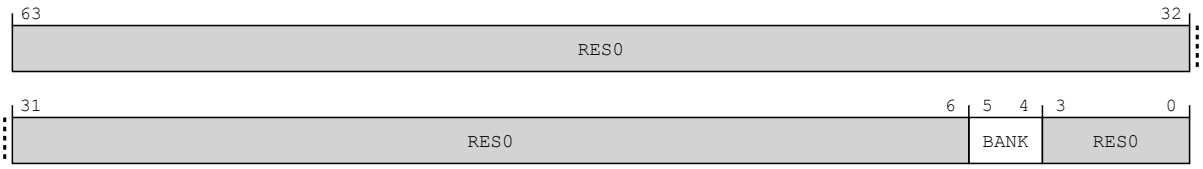

Bits [63:6]

Reserved, RES0.

BANK, bits [5:4]

Breakpoint and watchpoint bank select.

BANK	Meaning	Applies when
0b00	Select 0 to 15.	
0b01	Select 16 to 31.	NUM_BREAKPOINTS > 16 or NUM_WATCHPOINTS > 16
0b10	Select 32 to 47.	NUM_BREAKPOINTS > 32 or NUM_WATCHPOINTS > 32
0b11	Select 48 to 63.	NUM_BREAKPOINTS > 48 or NUM_WATCHPOINTS > 48

Each of the following register names accesses a register for breakpoint or watchpoint <n>, where `n = UInt(MDSELR_EL1.BANK:m[3:0])`:

- DBGBCR<m>_EL1.
- DBGBVR<m>_EL1.
- DBGWCR<m>_EL1.
- DBGWVR<m>_EL1.

This field is ignored by the PE and treated as zeros when any of the following are true:

- Executing at EL3 and MDCR_EL3.EBWE is 0.
- Executing at EL2 and the Effective value of MDCR_EL2.EBWE is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Executing at EL1 and the Effective value of MDSCR_EL1.EMBWE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if all of the following are true:
  - NUM_BREAKPOINTS <= 16
  - NUM_WATCHPOINTS <= 16
- Otherwise, access to this field is RW.

Bits [3:0]

Reserved, RES0.

Accessing MDSELR_EL1

When 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI, it is IMPLEMENTATION DEFINED whether these trap controls have any effect on accesses to MDSELR_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDSELR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_Debugv8p9) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EBWE == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nMDSELR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EBWE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDSELR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EBWE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EBWE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDSELR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDSELR_EL1();
end;
```

```
MSR MDSELR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_Debugv8p9) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EBWE == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
     ↪|| HDFGWTR2_EL2().nMDSELR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EBWE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDSELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EBWE == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EBWE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDSELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MDSELR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.22 MDSTEPOP_EL1, Monitor Debug Step Opcode Register

The MDSTEPOP_EL1 characteristics are:

Purpose

Used to execute instructions while Software Step is in active-not-pending state.

Configuration

This register is present only when FEAT_STEP2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to MDSTEPOP_EL1 are UNDEFINED.

Attributes

MDSTEPOP_EL1 is a 64-bit register.

Field descriptions

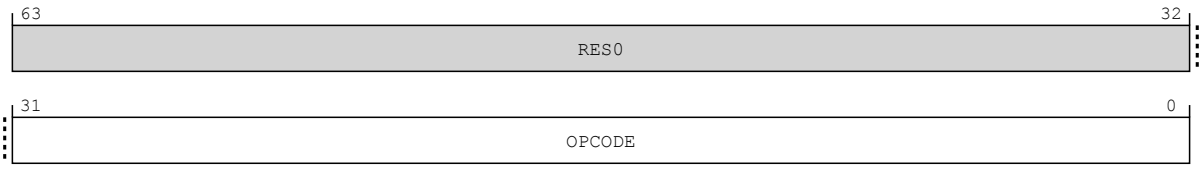

Bits [63:32]

Reserved, RES0.

OPCODE, bits [31:0]

A64 instruction to be executed on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MDSTEPOP_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDSTEPOP_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_STEP2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnSTEPOP == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nMDSTEPOP_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnSTEPOP == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDSTEPOP_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnSTEPOP == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnSTEPOP == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MDSTEPOP_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MDSTEPOP_EL1();
end;
```

```
MSR MDSTEPOP_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_STEP2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnSTEPOP == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nMDSTEPOP_EL1 == '0') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnSTEPOP == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDSTEPOP_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnSTEPOP == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnSTEPOP == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MDSTEPOP_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MDSTEPOP_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.23 OSDLR_EL1, OS Double Lock Register

The OSDLR_EL1 characteristics are:

Purpose

Used to control the OS Double Lock.

Configuration

AArch64 System register OSDLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGOSDLR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to OSDLR_EL1 are UNDEFINED.

Attributes

OSDLR_EL1 is a 64-bit register.

Field descriptions

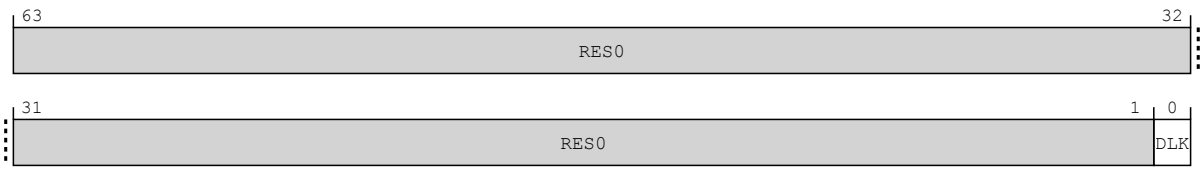

Bits [63:1]

Reserved, RES0.

DLK, bit [0]

When FEAT_DoubleLock is implemented:

OS Double Lock control bit.

DLK	Meaning
0b0	OS Double Lock unlocked.
0b1	OS Double Lock locked, if DBGPRCR_EL1.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

Accessing OSDLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, OSDLR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' &&
         ↪(IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_DoubleLock) && HDFGRTR_EL2().OSDLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL2.TDOSA")) then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSDLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' &&
         ↪(IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSDLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = OSDLR_EL1();
end;
```

```
MSR OSDLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' &&
         ↪(IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪IsFeatureImplemented(FEAT_DoubleLock) && HDFGWTR_EL2().OSDLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL2.TDOSA")) then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSDLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' &&
         ↪(IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSDLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    OSDLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.24 OSDTRRX_EL1, OS Lock Data Transfer Register, Receive

The OSDTRRX_EL1 characteristics are:

Purpose

Used for save and restore of DBGDTRRX_EL0. It is a component of the Debug Communications Channel.

Configuration

AArch64 System register OSDTRRX_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXext[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to OSDTRRX_EL1 are UNDEFINED.

Attributes

OSDTRRX_EL1 is a 64-bit register.

Field descriptions

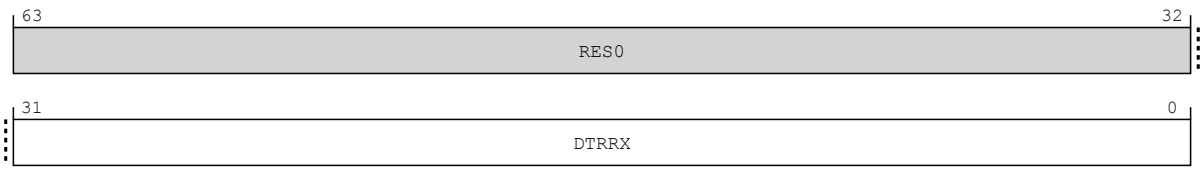

Bits [63:32]

Reserved, RES0.

DTRRX, bits [31:0]

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

Accessing OSDTRRX_EL1

Arm deprecates reads and writes of OSDTRRX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, OSDTRRX_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X{64}(t) = OSDTRRX_EL1();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSDTRRX_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSDTRRX_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = OSDTRRX_EL1();
end;
```

```
MSR OSDTRRX_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    OSDTRRX_EL1() = X{64}(t);
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSDTRRX_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSDTRRX_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    OSDTRRX_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.25 OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit

The OSDTRTX_EL1 characteristics are:

Purpose

Used for save/restore of DBGDTRTX_EL0. It is a component of the Debug Communications Channel.

Configuration

AArch64 System register OSDTRTX_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXext[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to OSDTRTX_EL1 are UNDEFINED.

Attributes

OSDTRTX_EL1 is a 64-bit register.

Field descriptions

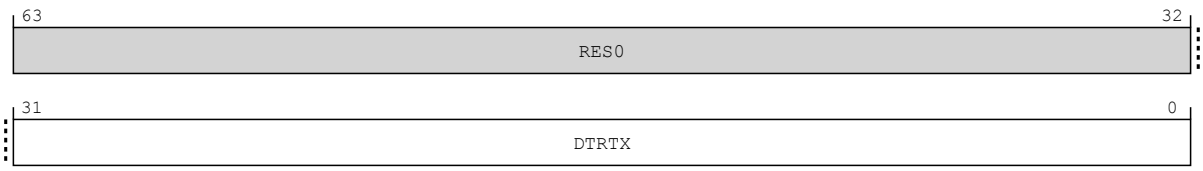

Bits [63:32]

Reserved, RES0.

DTRTX, bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

Accessing OSDTRTX_EL1

Arm deprecates reads and writes of OSDTRTX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, OSDTRTX_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0011	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X{64}(t) = OSDTRTX_EL1();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSDTRTX_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSDTRTX_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = OSDTRTX_EL1();
end;
```

```
MSR OSDTRTX_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0011	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    OSDTRTX_EL1() = X{64}(t);
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSDTRTX_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1'
        ↪then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSDTRTX_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    OSDTRTX_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.26 OSECCR_EL1, OS Lock Exception Catch Control Register

The OSECCR_EL1 characteristics are:

Purpose

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

Configuration

If OSLSR_EL1.OSLK == 0, then OSECCR_EL1 returns an UNKNOWN value on reads and ignores writes.

AArch64 System register OSECCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGOSECCR[31:0].

AArch64 System register OSECCR_EL1 bits [31:0] are architecturally mapped to External register EDECCR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to OSECCR_EL1 are UNDEFINED.

Attributes

OSECCR_EL1 is a 64-bit register.

Field descriptions

When OSLSR_EL1.OSLK == '1':

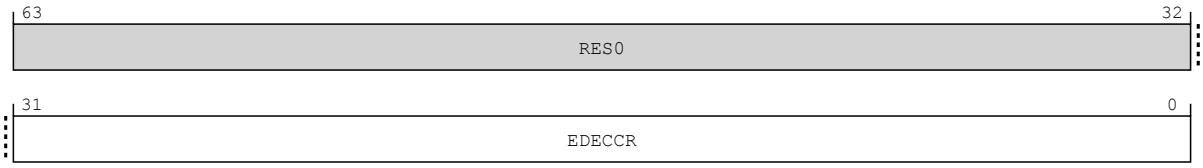

Bits [63:32]

Reserved, RES0.

EDECCR, bits [31:0]

Used for save/restore to EDECCR over powerdown.

Reads or writes to this field are indirect accesses to EDECCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to the expression `0x00`.

Accessing OSECCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, OSECCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0110	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().OSECCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = OSECCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = OSECCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = OSECCR_EL1();
    end;
end;
```

```
MSR OSECCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().OSECCR_EL1 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' then
        return;
    else
        OSECCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif OSLSR_EL1().OSLK == '0' then
        return;
    else
        OSECCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if OSLSR_EL1().OSLK == '0' then
        return;
    else
        OSECCR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.27 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

Configuration

The OS Lock can also be locked or unlocked using DBGOSLAR.

AArch64 System register OSLAR_EL1 bits [31:0] are architecturally mapped to External register OSLAR_EL1[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to OSLAR_EL1 are UNDEFINED.

Attributes

OSLAR_EL1 is a 64-bit register.

Field descriptions

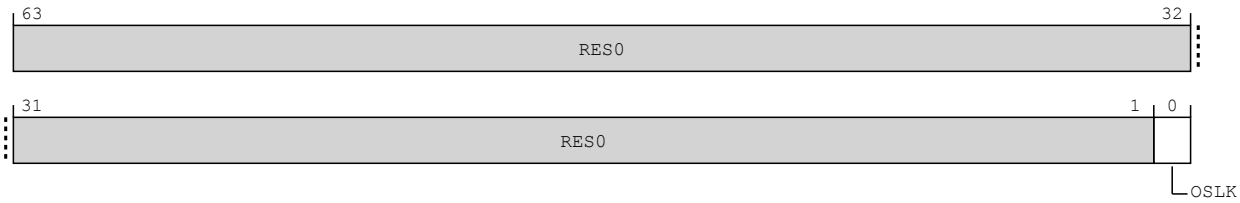

Bits [63:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use OSLSR_EL1.OSLK to check the current status of the lock.

Accessing OSLAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MSR OSLAR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().OSLAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSLAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        OSLAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    OSLAR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.28 OSLSR_EL1, OS Lock Status Register

The OSLSR_EL1 characteristics are:

Purpose

Provides the status of the OS Lock.

Configuration

AArch64 System register OSLSR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGOSLSR[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to OSLSR_EL1 are UNDEFINED.

Attributes

OSLSR_EL1 is a 64-bit register.

Field descriptions

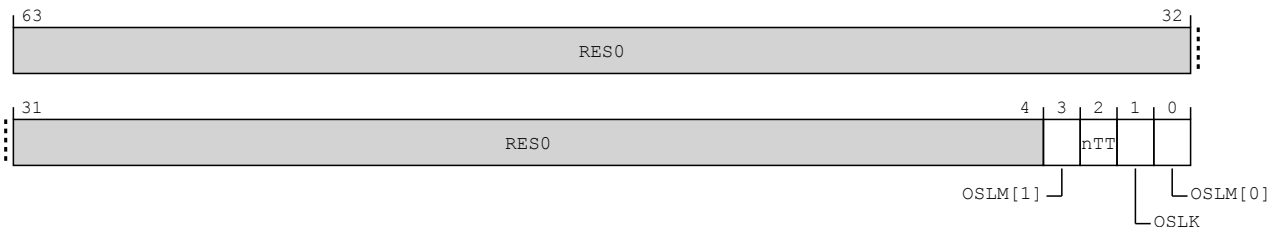

Bits [63:4]

Reserved, RES0.

OSLM, bits [3, 0]

OS Lock model implemented. Identifies the form of OS save and restore mechanism implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b00	OS Lock not implemented.
0b10	OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value 0b00 is not permitted.

Access to this field is RO.

nTT, bit [2]

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLK, bit [1]

OS Lock Status.

OSLK	Meaning
0b0	OS Lock unlocked.
0b1	OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'1'`.

Accessing OSLSR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, OSLSR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().OSLSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSLSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDOSA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = OSLSR_EL1();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = OSLSR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.29 SDER32_EL2, AArch32 Secure Debug Enable Register

The SDER32_EL2 characteristics are:

Purpose

Allows access to the AArch32 register SDER from Secure EL2 and EL3 only.

Configuration

When EL3 is implemented, AArch64 System register SDER32_EL2 bits [63:0] are architecturally mapped to AArch64 System register SDER32_EL3[63:0].

AArch64 System register SDER32_EL2 bits [31:0] are architecturally mapped to AArch32 System register SDER[31:0].

This register is present only when EL2 is implemented, FEAT_SEL2 is implemented, FEAT_AA32EL1 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to SDER32_EL2 are UNDEFINED.

Attributes

SDER32_EL2 is a 64-bit register.

Field descriptions

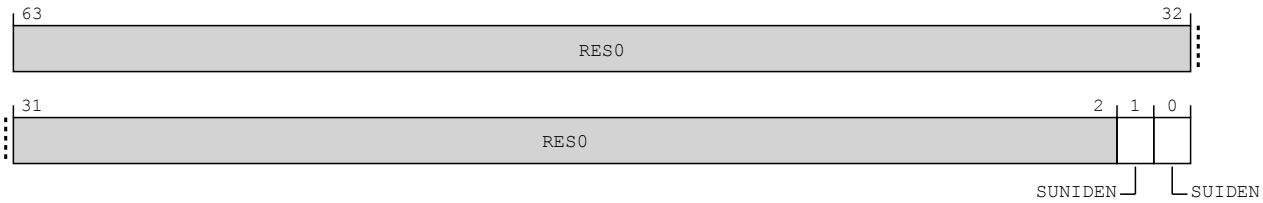

Bits [63:2]

Reserved, RES0.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug Enable.

SUNIDEN	Meaning
0b0	This bit has no effect on non-invasive debug.
0b1	Non-invasive debug is allowed in Secure EL0 using AArch32.

When Secure EL1 is using AArch32, the forms of non-invasive debug affected by this control are:

- The PC Sample-based Profiling Extension. See About the PC Sample-based Profiling Extension.
- When `SelfHostedTraceEnabled()` == FALSE, processor trace.
- When EL3 is implemented, Performance Monitors.

When Secure EL1 is using AArch64, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SUIDEN, bit [0]

When EL3 is implemented:

Secure User Invasive Debug Enable.

SUIDEN	Meaning
0b0	This bit does not affect the generation of debug exceptions at Secure EL0.
0b1	If EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing SDER32_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SDER32_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0011	0b001

```
if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA32EL1) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        X{64}(t) = SDER32_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        X{64}(t) = SDER32_EL2();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SDER32_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0011	0b001

```
if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA32EL1) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TDA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        SDER32_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        SDER32_EL2() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.3.30 SDER32_EL3, AArch32 Secure Debug Enable Register

The SDER32_EL3 characteristics are:

Purpose

Allows access to the AArch32 register SDER from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configuration

When EL2 is implemented and FEAT_SEL2 is implemented, AArch64 System register SDER32_EL3 bits [63:0] are architecturally mapped to AArch64 System register SDER32_EL2[63:0].

AArch64 System register SDER32_EL3 bits [31:0] are architecturally mapped to AArch32 System register SDER[31:0].

This register is present only when EL3 is implemented, FEAT_AA32EL1 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to SDER32_EL3 are UNDEFINED.

Attributes

SDER32_EL3 is a 64-bit register.

Field descriptions

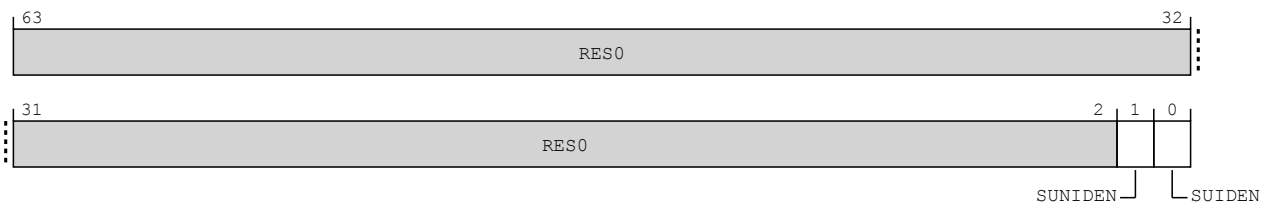

Bits [63:2]

Reserved, RES0.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug Enable.

SUNIDEN	Meaning
0b0	This bit has no effect on non-invasive debug.
0b1	Non-invasive debug is allowed in Secure EL0 using AArch32.

When Secure EL1 is using AArch32, the forms of non-invasive debug affected by this control are:

- The PC Sample-based Profiling Extension. See About the PC Sample-based Profiling Extension.
- When `SelfHostedTraceEnabled()` == FALSE, processor trace.
- Performance Monitors.

When Secure EL1 is using AArch64, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SUIDEN, bit [0]

Secure User Invasive Debug Enable.

SUIDEN	Meaning
0b0	This bit does not affect the generation of debug exceptions at Secure EL0.
0b1	If EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SDER32_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SDER32_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SDER32_EL3();
end;
```

```
MSR SDER32_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    SDER32_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4 Trace registers

This section lists the Trace registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.1 TRBBASER_EL1, Trace Buffer Base Address Register

The TRBBASER_EL1 characteristics are:

Purpose

Defines the base address for the trace buffer.

Configuration

When FEAT_TRBE_EXT is implemented, AArch64 System register TRBBASER_EL1 bits [63:0] are architecturally mapped to External register TRBBASER_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBBASER_EL1 are UNDEFINED.

Attributes

TRBBASER_EL1 is a 64-bit register.

Field descriptions

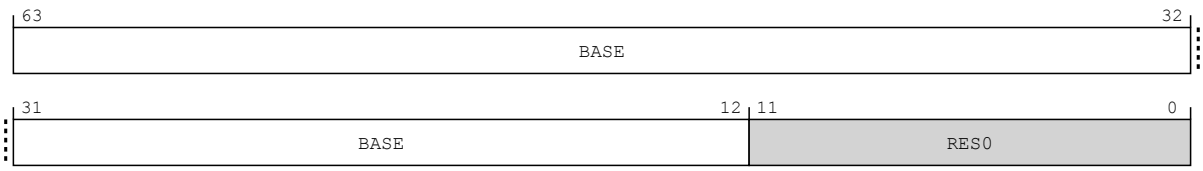

BASE, bits [63:12]

Trace Buffer Base pointer address. (TRBBASER_EL1.BASE « 12) is the address of the first byte in the trace buffer. Bits [11:0] of the Base pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBBASER_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

Accessing TRBBASER_EL1

The PE might ignore a write to TRBBASER_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBBASER_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b010

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBBASER_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBBASER_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBBASER_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBBASER_EL1();
    end;
end;
```

```
MSR TRBBASER_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRBBASER_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBBASER_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBBASER_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBBASER_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.2 TRBIDR_EL1, Trace Buffer ID Register

The TRBIDR_EL1 characteristics are:

Purpose

Describes constraints on using the Trace Buffer Unit to software, including whether the Trace Buffer Unit can be programmed at the current Exception level.

Configuration

When FEAT_TRBE_EXT is implemented, AArch64 System register TRBIDR_EL1 bits [63:0] are architecturally mapped to External register TRBIDR_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBIDR_EL1 are UNDEFINED.

Attributes

TRBIDR_EL1 is a 64-bit register.

Field descriptions

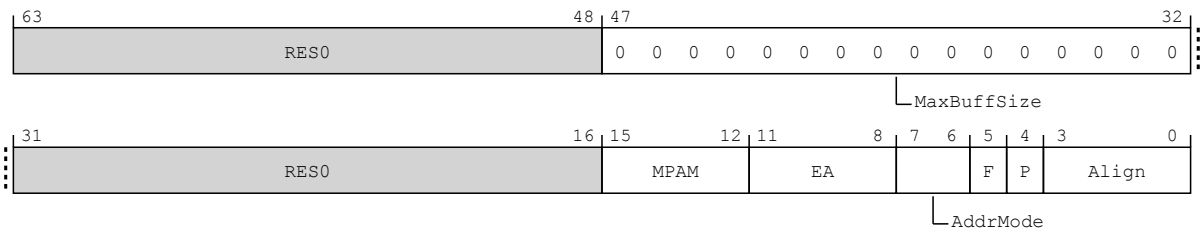

Bits [63:48]

Reserved, RES0.

MaxBuffSize, bits [47:32]

Maximum supported trace buffer size. Reserved for software use.

Reads as `0x0000`

The only permitted value is `0x0000`, indicating there is no limit to the maximum buffer size.

Note

Permitted values relate to the values an implementation is permitted to set this field to. A hypervisor might trap accesses to this register and use other values to describe limitations of its virtualization support to a guest operating system, as follows:

- MaxBuffSize bits[8:0] encodes a mantissa value, `M`.
- MaxBuffSize bits[13:9] encodes an exponent value, `E`.
- MaxBuffSize bits[15:14] are reserved.

The maximum buffer size, in bytes, is expressed using the following function:

```
if IsZero(E), then UInt(M:Zeros(12)) else UInt('1':M:Zeros(UInt(E)+11))
```

For example:

- A value of `0x0001` means a maximum buffer size of 4KB.
- A value of `0x3FFF` means a maximum buffer size of 4092TB.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:16]

Reserved, RES0.

MPAM, bits [15:12]

When FEAT_TRBE_EXT is implemented:

FEAT_MPAMv0p1 or FEAT_MPAMv1p0 extensions. Indicates Memory Partitioning and Monitoring (MPAM) support in the Trace Buffer Unit when using External mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MPAM	Meaning
0b0000	FEAT_MPAMv0p1 and FEAT_MPAMv1p0 are not implemented by the PE.
0b0001	FEAT_MPAMv0p1 or FEAT_MPAMv1p0 is implemented by the Trace Buffer Unit, using Default physical PARTID and PMG values in External mode.
0b0010	Trace Buffer MPAM extensions implemented, using FEAT_MPAMv0p1 or FEAT_MPAMv1p0.

When FEAT_MPAMv0p1 and FEAT_MPAMv1p0 are not implemented by the PE, the only permitted value is 0b0000.

When either FEAT_MPAMv0p1 or FEAT_MPAMv1p0 is implemented by the PE, the value 0b0000 is not permitted.

FEAT_TRBE_MPAM implements the functionality identified by the value 0b0010.

Access to this field is RO.

Otherwise:

Reserved, RES0.

EA, bits [11:8]

External Abort handling. Describes how the PE manages External aborts on writes made by the Trace Buffer Unit to the trace buffer.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EA	Meaning
0b0000	Not described.
0b0001	The PE ignores External aborts on writes made by the Trace Buffer Unit.
0b0010	An External abort on a write made by the Trace Buffer Unit generates an asynchronous SError exception at the PE.

All other values are reserved.

From Armv9.3, the value 0b0000 is not permitted.

The behavior described by this field does not apply for External aborts on a translation table walk, translation table update, or GPT walk made by the Trace Buffer Unit that are reported as MMU faults using TRBSR_ELx. For more information, see ‘External aborts’.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AddrMode, bits [7:6]

Address Modes. Describes the addressing modes available for the trace buffer.

AddrMode	Meaning
`0b00`	Virtual and physical address modes are supported.
`0b01`	Only virtual address mode is supported.
`0b10`	Reserved for software use under virtualization, to show that only physical address mode is supported.

Other values are reserved.

If the Effective value of TRFCR_EL2.DnVM is 1 and the value returned for TRBIDR_EL1.P is 0, then this field reads as `0b01`. Otherwise, this field reads as `0b00`.

Note

A hypervisor might trap accesses to this register to describe limitations of its virtualization support to a guest operating system.

F, bit [5]

Flag updates. Describes how address translations performed by the Trace Buffer Unit manage the Access flag and dirty state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F	Meaning
`0b0`	Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is always disabled for all translation stages.
`0b1`	Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is controlled in the same way as explicit memory accesses in the trace buffer owning translation regime.

Note

If hardware management of the Access flag is disabled for a stage of translation, an access to a Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.
If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor and might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

From Armv9.3, the value 0 is not permitted.

Access to this field is RO.

P, bit [4]

Programming not allowed. When read at EL3, this field reads as zero. Otherwise, indicates that the trace buffer owning Exception level is a higher Exception level or the trace buffer owning Security state is not the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P	Meaning
0b0	Programming is allowed.
0b1	Programming not allowed.

The value read from this field depends on the current Exception level and the Effective values of MDCR_EL3.NSTB, MDCR_EL3.NSTBE, and MDCR_EL2.E2TB:

- If EL3 is implemented, MDCR_EL3.NSTB is 0b0x, and either FEAT_RME is not implemented, or Secure state is implemented and MDCR_EL3.NSTBE is 0, then this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
  - If Secure EL2 is implemented and enabled, and MDCR_EL2.E2TB is 0b00, Secure EL1.
- If EL3 is implemented, MDCR_EL3.NSTB is 0b1x and either FEAT_RME is not implemented or MDCR_EL3.NSTBE is 0, then this field reads as one from:
  - If Secure state is implemented, Secure EL1.
  - If Secure EL2 is implemented, Secure EL2.
  - If EL2 is implemented and MDCR_EL2.E2TB is 0b00, Non-secure EL1.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
- If FEAT_RME is implemented, and MDCR_EL3.{NSTB, NSTBE} is {0b1x, 1}, then this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - If Secure state is implemented, Secure EL1 and Secure EL2.
  - If MDCR_EL2.E2TB is 0b00, Realm EL1.
- If EL3 is not implemented, EL2 is implemented, and MDCR_EL2.E2TB is 0b00, then this field reads as one from EL1.

Otherwise, this field reads as zero.

Align, bits [3:0]

Defines the minimum alignment constraint for writes to TRBPTR_EL1 and TRBTRG_EL1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Align	Meaning
0b0000	Byte.
0b0001	Halfword.
0b0010	Word.
0b0011	Doubleword.
0b0100	16 bytes.
0b0101	32 bytes.
0b0110	64 bytes.
0b0111	128 bytes.
0b1000	256 bytes.
0b1001	512 bytes.
0b1010	1KB.
0b1011	2KB.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b111

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBIDR_EL1();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.3 TRBLIMITR_EL1, Trace Buffer Limit Address Register

The TRBLIMITR_EL1 characteristics are:

Purpose

Defines the top address for the trace buffer, and controls the trace buffer modes and enable.

Configuration

When FEAT_TRBE_EXT is implemented, AArch64 System register TRBLIMITR_EL1 bits [63:0] are architecturally mapped to External register TRBLIMITR_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBLIMITR_EL1 are UNDEFINED.

Attributes

TRBLIMITR_EL1 is a 64-bit register.

Field descriptions

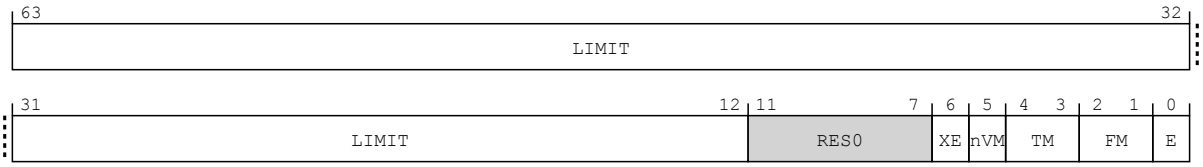

LIMIT, bits [63:12]

Trace Buffer Limit pointer address. (TRBLIMITR_EL1.LIMIT « 12) is the address of the last byte in the trace buffer plus one. Bits [11:0] of the Limit pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBLIMITR_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

XE, bit [6]

When FEAT_TRBE_EXT is implemented:

Trace Buffer Unit External mode enable. Used for save/restore of TRBLIMITR_EL1.XE.

XE	Meaning
0b0	Trace Buffer Unit is not enabled by this control.
0b1	If SelfHostedTraceEnabled() is FALSE, the Trace Buffer Unit is enabled.

Software must treat this field as UNK/SBZP when the OS Lock is unlocked.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When !OSLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

nVM, bit [5]

Address mode.

nVM	Meaning
`0b0`	The trace buffer pointers are virtual addresses.
`0b1`	The trace buffer pointers are: • Physical address in the owning security state if the owning translation regime has no stage 2 translation. • Intermediate physical addresses in the owning security state if the owning translation regime has stage 2 translations.

If FEAT_TRBE_EXT is implemented and `SelfHostedTraceEnabled()` == FALSE, then the PE ignores the value of this field and the trace buffer pointers are always physical addresses.

If FEAT_TRBEv1p1 is implemented, `SelfHostedTraceEnabled()` == TRUE, and the Effective value of TRFCR_EL2.DnVM is 1, then the PE ignores the value of this field, and the trace buffer pointers are always virtual addresses.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES1 if all of the following are true:
  - FEAT_TRBE_EXT is implemented
  - !SelfHostedTraceEnabled()
- Otherwise, access to this field is RW.

TM, bits [4:3]

Trigger mode.

TM	Meaning
`0b00`	Stop on trigger. Flush then stop collection and raise maintenance interrupt on Trigger Event.
`0b01`	IRQ on trigger. Continue collection and raise maintenance interrupt on Trigger Event.
`0b11`	Ignore trigger. Continue collection and do not raise maintenance interrupt on Trigger Event.

All other values are reserved.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

FM, bits [2:1]

Trace buffer mode.

FM	Meaning
0b00	Fill mode. Stop collection and raise maintenance interrupt on current write pointer wrap.
0b01	Wrap mode. Continue collection and raise maintenance interrupt on current write pointer wrap.
0b11	Circular Buffer mode. Continue collection and do not raise maintenance interrupt on current write pointer wrap.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Trace Buffer Unit enable. Controls whether the Trace Buffer Unit is enabled when `SelfHostedTraceEnabled()` == TRUE.

E	Meaning
0b0	Trace Buffer Unit is not enabled by this control.
0b1	If `SelfHostedTraceEnabled()` is TRUE, the Trace Buffer Unit is enabled.

If FEAT_TRBE_EXT is implemented and `SelfHostedTraceEnabled()` == FALSE, then TRBLIMITR_EL1.XE controls whether the Trace Buffer Unit is enabled.

If FEAT_TRBE_EXT is not implemented, then the Trace Buffer Unit is disabled when `SelfHostedTraceEnabled()` == FALSE.

All output is discarded by the Trace Buffer Unit when the Trace Buffer Unit is disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRBLIMITR_EL1

The PE might ignore a write to TRBLIMITR_EL1 if all the following are true:

- TRBLIMITR_EL1.E is 1.
- Either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- The write does not set TRBLIMITR_EL1.E to 0.

If FEAT_TRBE_EXT is implemented, the PE might ignore a write to TRBLIMITR_EL1 if all the following are true:

- TRBLIMITR_EL1.XE is 1.
- The Trace Buffer Unit is using External mode.
- The write does not set TRBLIMITR_EL1.XE to 0.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TRBLIMITR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b000

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBLIMITR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBLIMITR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBLIMITR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBLIMITR_EL1();
    end;
end;
```

```
MSR TRBLIMITR_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b000

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRBLIMITR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBLIMITR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBLIMITR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBLIMITR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.4 TRBMAR_EL1, Trace Buffer Memory Attribute Register

The TRBMAR_EL1 characteristics are:

Purpose

Controls Trace Buffer Unit accesses to memory.

Configures some of the memory attributes of writes performed by the Trace Buffer Unit.

Configuration

When FEAT_TRBE_EXT is implemented, AArch64 System register TRBMAR_EL1 bits [63:0] are architecturally mapped to External register TRBMAR_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBMAR_EL1 are UNDEFINED.

Attributes

TRBMAR_EL1 is a 64-bit register.

Field descriptions

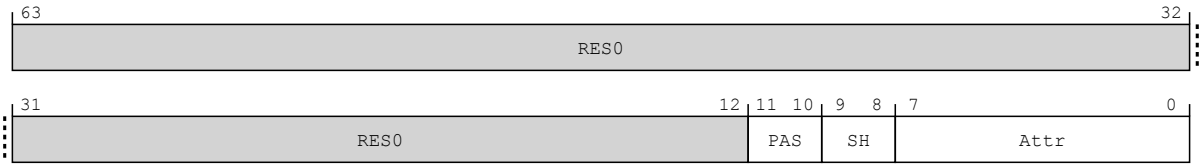

Bits [63:12]

Reserved, RES0.

PAS, bits [11:10]

When FEAT_TRBE_EXT is implemented:

Physical address specifier. Defines the PAS attribute for memory addressed by the buffer in External mode.

PAS	Meaning	Applies when
0b00	Secure.	FEAT_Secure is implemented
0b01	Non-secure.	
0b10	Root.	FEAT_RME is implemented
0b11	Realm.	FEAT_RME is implemented

All other values are reserved.

If the Trace Buffer Unit is using external mode and either TRBMAR_EL1.PAS is set to a reserved value, or the IMPLEMENTATION DEFINED authentication interface prohibits invasive debug of the Security state corresponding to the physical address space selected by TRBMAR_EL1.PAS, then when the Trace Buffer Unit receives trace data from the trace unit, it does not write the trace data to memory and generates a trace buffer management event. That is, if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBMAR_EL1.PAS is `0b00`, and either `ExternalSecureInvasiveDebugEnabled()` == FALSE or Secure state is not implemented.
- TRBMAR_EL1.PAS is `0b10`, and either `ExternalRootInvasiveDebugEnabled()` == FALSE or FEAT_RME is not implemented.
- TRBMAR_EL1.PAS is `0b11`, and either `ExternalRealmInvasiveDebugEnabled()` == FALSE or FEAT_RME is not implemented.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bits [9:8]

Trace buffer stage 1 shareability domain. Defines the shareability domain for Normal memory used by the trace buffer.

SH	Meaning
`0b00`	Non-shareable.
`0b10`	Outer Shareable.
`0b11`	Inner Shareable.

All other values are reserved.

If the trace buffer pointers specify virtual addresses, the memory attributes are defined by the translation tables and this field is ignored.

This field is ignored when TRBMAR_EL1.Attr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_TRBE_EXT is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_TRBE_EXT is not implemented, this field resets to an architecturally UNKNOWN value.

Attr, bits [7:0]

Trace buffer stage 1 memory type and attributes. Defines the memory type and, for Normal memory, the cacheability attributes, for memory addressed by the trace buffer.

The encoding of this field is the same as that of a MAIR_ELx.Attr<n> field, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0b0000dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
0b0000dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
0b0000dd1x		UNPREDICTABLE.
0boooooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
0b01000000		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b10100000		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

‘dd’	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

‘oooo’		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

If the trace buffer pointers specify virtual addresses, the memory attributes are defined by the translation tables and this field is ignored.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_TRBE_EXT is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_TRBE_EXT is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing TRBMAR_EL1

The PE might ignore a write to TRBMAR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBMAR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBMAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBMAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBMAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBMAR_EL1();
    end;
end;
```

MSR TRBMAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b100

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRBMAR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBMAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBMAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBMAR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.5 TRBMPAM_EL1, Trace Buffer MPAM Configuration Register

The TRBMPAM_EL1 characteristics are:

Purpose

Defines the PARTID, PMG, and MPAM_SP values used by the trace buffer unit in external mode.

Configuration

AArch64 System register TRBMPAM_EL1 bits [63:0] are architecturally mapped to External register TRBMPAM_EL1[63:0].

This register is present only when FEAT_TRBE_MPAM is implemented. Otherwise, direct accesses to TRBMPAM_EL1 are UNDEFINED.

Attributes

TRBMPAM_EL1 is a 64-bit register.

Field descriptions

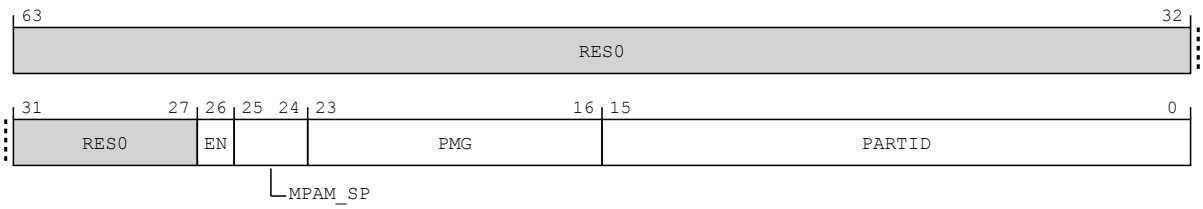

Bits [63:27]

Reserved, RES0.

EN, bit [26]

Enable. Enables use of non-Default MPAM values.

EN	Meaning
0b0	Use Default physical PARTID and Default physical PMG values, and the PARTID space corresponding to the trace physical address space defined by TRBMAR_EL1.PAS.
0b1	Use TRBMPAM_EL1.{PARTID, PMG, MPAM_SP}.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

MPAM_SP, bits [25:24]

Partition Identifier space. Selects the PARTID space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM_SP	Meaning	Applies when
`0b00`	PARTID is in the Secure PARTID space.	FEAT_Secure is implemented
`0b01`	PARTID is in the Non-secure PARTID space.	
`0b10`	PARTID is in the Root PARTID space.	FEAT_RME is implemented
`0b11`	PARTID is in the Realm PARTID space.	FEAT_RME is implemented

If the Trace Buffer Unit is using external mode and either TRBMPAM_EL1.MPAM_SP is set to reserved value, or the IMPLEMENTATION DEFINED authentication interface prohibits invasive debug of the Security state corresponding to the Partition Identifier space selected by TRBMPAM_EL1.MPAM_SP, then when the Trace Buffer Unit receives trace data from the trace unit, it does not write the trace data to memory and generates a trace buffer management event.

The interface prohibits invasive debug of the Security state if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure state is implemented, `ExternalSecureInvasiveDebugEnabled()` == FALSE and TRBMPAM_EL1.MPAM_SP is `0b00`.
- FEAT_RME is implemented, `ExternalRootInvasiveDebugEnabled()` == FALSE, and TRBMPAM_EL1.MPAM_SP is `0b10`.
- FEAT_RME is implemented, `ExternalRealmInvasiveDebugEnabled()` == FALSE, and TRBMPAM_EL1.MPAM_SP is `0b11`.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PMG, bits [23:16]

Performance Monitoring Group. Selects the PMG.

Only sufficient low-order bits are required to represent the TRBDEVID1.PMG_MAX. Higher-order bits are RES0.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PARTID, bits [15:0]

Partition Identifier. Selects the PARTID.

Only sufficient low-order bits are required to represent the TRBDEVID1.PARTID_MAX. Higher-order bits are RES0.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBMPAM_EL1

The PE might ignore a write to TRBMPAM_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TRBMPAM_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b101

```
if !IsFeatureImplemented(FEAT_TRBE_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnTB2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nTRBMPAM_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnTB2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBMPAM_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnTB2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnTB2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBMPAM_EL1();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
        ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBMPAM_EL1();
    end;
end;
```

```
MSR TRBMPAM_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b101

```
if !IsFeatureImplemented(FEAT_TRBE_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnTB2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
        ↪|| HDFGWTR2_EL2().nTRBMPAM_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnTB2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
        ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBMPAM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnTB2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnTB2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBMPAM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBMPAM_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.6 TRBPTR_EL1, Trace Buffer Write Pointer Register

The TRBPTR_EL1 characteristics are:

Purpose

Defines the current write pointer for the trace buffer.

Configuration

When FEAT_TRBE_EXT is implemented, AArch64 System register TRBPTR_EL1 bits [63:0] are architecturally mapped to External register TRBPTR_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBPTR_EL1 are UNDEFINED.

Attributes

TRBPTR_EL1 is a 64-bit register.

Field descriptions

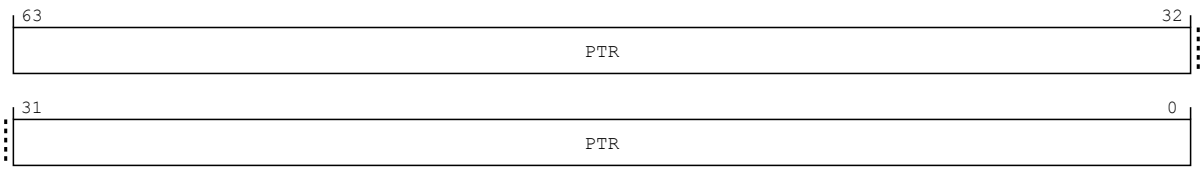

PTR, bits [63:0]

Trace Buffer current write pointer address.

Defines the virtual address of the next entry to be written to the trace buffer.

If TRBIDR_EL1.Align is not zero, then it is IMPLEMENTATION DEFINED whether bits [M-1:0] are RES0 or read/write, where M is an integer between 1 and TRBIDR_EL1.Align inclusive.

The architecture places restrictions on the values that software can write to the pointer. For more information, see Restrictions on Programming the Trace Buffer Unit.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBPTR_EL1

The PE might ignore a write to TRBPTR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBPTR_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b001

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBPTR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBPTR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBPTR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBPTR_EL1();
    end;
end;
```

```
MSR TRBPTR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRBPTR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBPTR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBPTR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBPTR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.7 TRBSR_EL1, Trace Buffer Status/syndrome Register (EL1)

The TRBSR_EL1 characteristics are:

Purpose

Provides syndrome information to software for a trace buffer management event.

Configuration

AArch64 System register TRBSR_EL1 bits [63:0] are architecturally mapped to External register TRBSR_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBSR_EL1 are UNDEFINED.

Attributes

TRBSR_EL1 is a 64-bit register.

Field descriptions

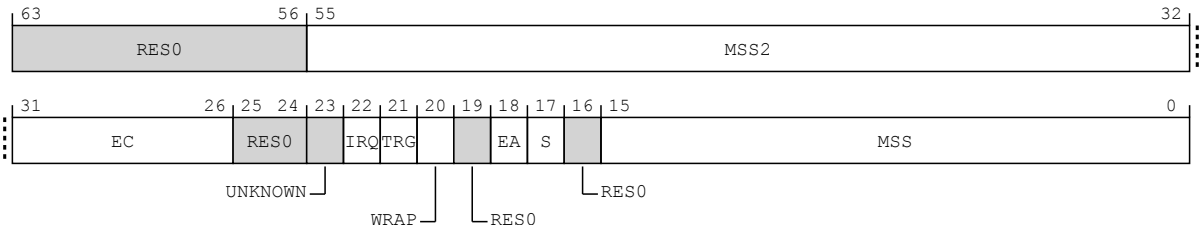

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other trace buffer management events

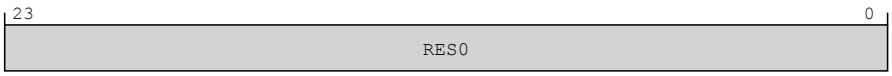

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

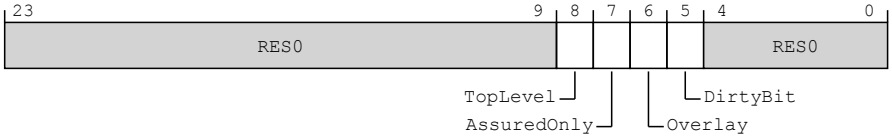

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, TRBSR_EL1.EC == '100101', and GetTRBSR_EL1_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetTRBSR_EL1_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetTRBSR_EL1_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to trace buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the trace buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other trace buffer management event. All trace buffer management events other than those described by the other defined Event class codes.	MSS encoding for other trace buffer management events	MSS2 encoding for other trace buffer management events	
0b011110	Granule Protection Check fault on write to trace buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to trace buffer	MSS2 encoding for Granule Protection Check faults on write to trace buffer	FEAT_RME is implemented
0b011111	Trace buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	
0b100101	Stage 2 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [25:24]

Reserved, RES0.

Bit [23]

When FEAT_TRBE_EXT is implemented:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRQ, bit [22]

Maintenance status. Indicates that a trace buffer management event has been recorded.

IRQ	Meaning
0b0	No trace buffer management event for EL1 has been recorded.
0b1	A trace buffer management event for EL1 has been recorded.

When FEAT_TRBE_EXC is implemented, this field indicates a management event for EL1.

If FEAT_TRBE_EXC is implemented and the TRBE Profiling exception for EL1 is enabled, then when this field is 1, a TRBE Profiling exception for EL1 is pending

If FEAT_TRBE_EXC is not implemented or the TRBE Profiling exception for EL1 is disabled, then this field drives the **TRBIRQ** trace buffer interrupt request signal.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

TRG, bit [21]

Triggered.

TRG	Meaning
0b0	No Detected Trigger has been observed since this field was last cleared to zero.
0b1	A Detected Trigger has been observed since this field was last cleared to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

WRAP, bit [20]

Wrapped.

WRAP	Meaning
0b0	The current write pointer has not wrapped since this field was last cleared to zero.
0b1	The current write pointer has wrapped since this field was last cleared to zero.

For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in the current write pointer, if the current write pointer is equal to the Limit pointer minus one, the current write pointer is wrapped by setting it to the Base pointer, and this field is set to 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External Abort.

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Trace Buffer Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Trace Buffer Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. From Armv9.3, this field is not set to 1 by the PE for any other trace buffer management event.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Stopped.

S	Meaning
0b0	Collection has not been stopped.
0b1	Collection is stopped.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [16]

Reserved, RES0.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the trace buffer management event.

The syndrome contents for each trace buffer management event are described in the following sections.

MSS encoding for other trace buffer management events

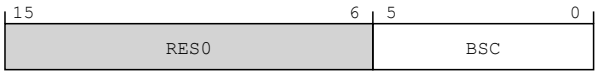

Bits [15:6]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BSC, bits [5:0]

Trace buffer status code

BSC	Meaning	Applies when
0b000000	Collection not stopped, or access not allowed.	
0b000001	Trace buffer filled. Collection stopped because the current write pointer wrapped to the base pointer and the trace buffer mode is Fill mode.	
0b000010	Trigger Event. Collection stopped because of a Trigger Event. See TRBTRG_EL1 for more information.	
0b000011	Manual Stop. Collection stopped because of a Manual Stop event. See TRBCR.ManStop for more information.	FEAT_TRBE_EXT is implemented
0b000100	Buffer size. The requested trace buffer size was too large.	

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

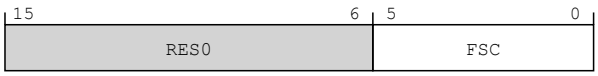

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to trace buffer

15 … 0
RES0

Bits [15:0]

Reserved, RES0.

MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

15 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Accessing TRBSR_EL1

The PE might ignore a write to TRBSR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `TRBSR_EL1` or `TRBSR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TRBSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b011

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} && (EffectiveTRFCR_EL2_EE() != '00' && TRFCR_EL1().EE !=
         ↪'00') then
        X{64}(t) = NVMem(0x860);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    elsif EffectiveTRFCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        X{64}(t) = TRBSR_EL2();
    else
        X{64}(t) = TRBSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBSR_EL1();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TRBSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b011

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} && (EffectiveTRFCR_EL2_EE() != '00' && TRFCR_EL1().EE !=
         ↪'00') then
        NVMem(0x860) = X{64}(t);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    elsif EffectiveTRFCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        TRBSR_EL2() = X{64}(t);
    else
        TRBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBSR_EL1() = X{64}(t);
    end;
end;
```

When FEAT_TRBE_EXC is implemented and FEAT_VHE is implemented MRS <Xt>, TRBSR_EL12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1011	0b011

```
if !(IsFeatureImplemented(FEAT_TRBE_EXC) && IsFeatureImplemented(FEAT_VHE)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x860);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRBSR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRBSR_EL1();
        end;
    else
        Undefined();
    end;
end;
```

When FEAT_TRBE_EXC is implemented and FEAT_VHE is implemented MSR TRBSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1011	0b011

```
if !(IsFeatureImplemented(FEAT_TRBE_EXC) && IsFeatureImplemented(FEAT_VHE)) then
    Undefined();
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x860) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            TRBSR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            TRBSR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.8 TRBSR_EL2, Trace Buffer Syndrome Register (EL2)

The TRBSR_EL2 characteristics are:

Purpose

Provides syndrome information to software for a trace buffer management event.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_TRBE_EXC is implemented. Otherwise, direct accesses to TRBSR_EL2 are UNDEFINED.

Attributes

TRBSR_EL2 is a 64-bit register.

Field descriptions

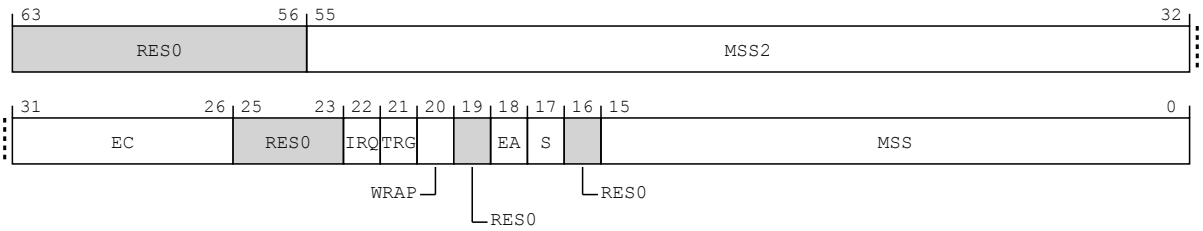

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other trace buffer management events

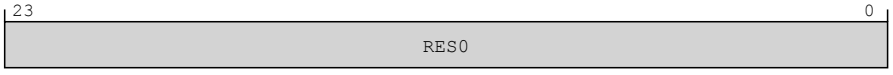

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

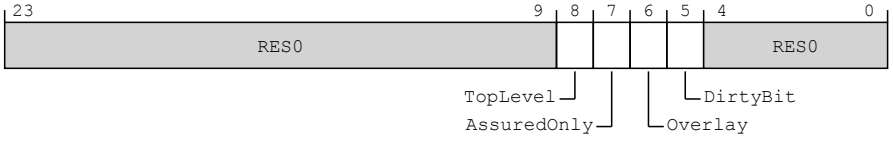

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, TRBSR_EL2.EC == '100101', and GetTRBSR_EL2_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetTRBSR_EL2_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetTRBSR_EL2_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
`0b0`	Permission Fault is not due to dirty state.
`0b1`	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to trace buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the trace buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other trace buffer management event. All trace buffer management events other than those described by the other defined Event class codes.	MSS encoding for other trace buffer management events	MSS2 encoding for other trace buffer management events	
0b011110	Granule Protection Check fault on write to trace buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to trace buffer	MSS2 encoding for Granule Protection Check faults on write to trace buffer	FEAT_RME is implemented
0b011111	Trace buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	
0b100101	Stage 2 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [25:23]

Reserved, RES0.

IRQ, bit [22]

Maintenance status. Indicates that a trace buffer management event has been recorded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRQ	Meaning
0b0	No trace buffer management event for EL2 has been recorded.
0b1	A trace buffer management event for EL2 has been recorded.

When FEAT_TRBE_EXC is implemented, this field indicates a management event for EL2.

If the TRBE Profiling exception for EL2 is enabled, then when this field is 1, a TRBE Profiling exception for EL2 is pending

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

TRG, bit [21]

Triggered.

TRG	Meaning
0b0	No Detected Trigger has been observed since this field was last cleared to zero.
0b1	A Detected Trigger has been observed since this field was last cleared to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

WRAP, bit [20]

Wrapped.

WRAP	Meaning
0b0	The current write pointer has not wrapped since this field was last cleared to zero.
0b1	The current write pointer has wrapped since this field was last cleared to zero.

For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in the current write pointer, if the current write pointer is equal to the Limit pointer minus one, the current write pointer is wrapped by setting it to the Base pointer, and this field is set to 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External Abort.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Trace Buffer Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Trace Buffer Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. From Armv9.3, this field is not set to 1 by the PE for any other trace buffer management event.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Stopped.

S	Meaning
0b0	Collection has not been stopped.
0b1	Collection is stopped.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [16]

Reserved, RES0.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the trace buffer management event.

The syndrome contents for each trace buffer management event are described in the following sections.

MSS encoding for other trace buffer management events

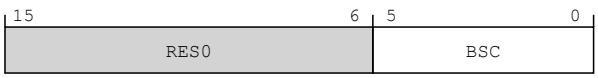

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Trace buffer status code

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BSC	Meaning	Applies when
0b000000	Collection not stopped, or access not allowed.	
0b000001	Trace buffer filled. Collection stopped because the current write pointer wrapped to the base pointer and the trace buffer mode is Fill mode.	
0b000010	Trigger Event. Collection stopped because of a Trigger Event. See TRBTRG_EL1 for more information.	
0b000011	Manual Stop. Collection stopped because of a Manual Stop event. See TRBCR.ManStop for more information.	FEAT_TRBE_EXT is implemented
0b000100	Buffer size. The requested trace buffer size was too large.	

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

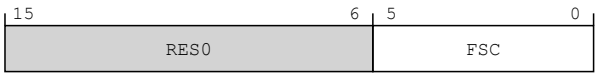

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to trace buffer

15	0
RES0	

Bits [15:0]

Reserved, RES0.

MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

15	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Accessing TRBSR_EL2

The PE might ignore a write to TRBSR_EL2 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TRBSR_EL2` or `TRBSR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBSR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1011	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TRBE_EXC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TRBEE == '00' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TRBEE == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TRBSR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TRBSR_EL2();
end;
```

MSR TRBSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1011	0b011

```
if !IsFeatureImplemented(FEAT_TRBE_EXC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TRBEE == '00' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TRBEE == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TRBSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TRBSR_EL2() = X{64}(t);
end;
```

MRS <Xt>, TRBSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b011

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} && (EffectiveTRFCR_EL2_EE() != '00' && TRFCR_EL1().EE !=
         ↪'00') then
        X{64}(t) = NVMem(0x860);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    elsif EffectiveTRFCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        X{64}(t) = TRBSR_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = TRBSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
        ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBSR_EL1();
    end;
end;
```

```
MSR TRBSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b011

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HDFGWTR_EL2().TRBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} && (EffectiveTRFCR_EL2_EE() != '00' && TRFCR_EL1().EE !=
        ↪'00') then
        NVMem(0x860) = X{64}(t);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
        ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
        ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    elsif EffectiveTRFCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        TRBSR_EL2() = X{64}(t);
    else
        TRBSR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBSR_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.9 TRBSR_EL3, Trace Buffer Syndrome Register (EL3)

The TRBSR_EL3 characteristics are:

Purpose

Provides syndrome information to software for a trace buffer management event.

Configuration

This register is present only when FEAT_TRBE_EXC is implemented and EL3 is implemented. Otherwise, direct accesses to TRBSR_EL3 are UNDEFINED.

Attributes

TRBSR_EL3 is a 64-bit register.

Field descriptions

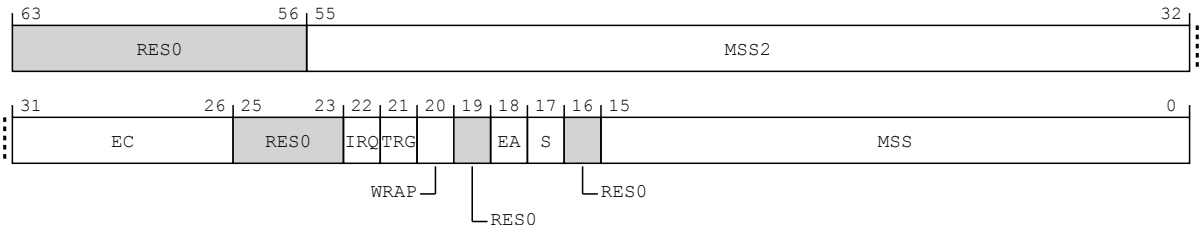

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other trace buffer management events

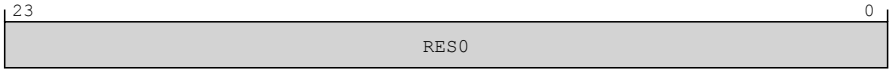

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

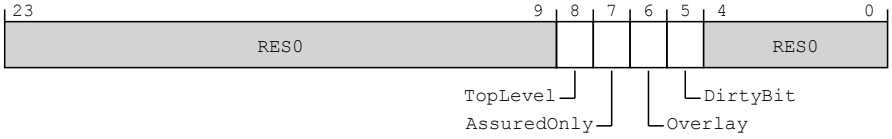

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, TRBSR_EL3.EC == '100101', and GetTRBSR_EL3_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetTRBSR_EL3_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetTRBSR_EL3_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to trace buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the trace buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other trace buffer management event. All trace buffer management events other than those described by the other defined Event class codes.	MSS encoding for other trace buffer management events	MSS2 encoding for other trace buffer management events	
0b011110	Granule Protection Check fault on write to trace buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to trace buffer	MSS2 encoding for Granule Protection Check faults on write to trace buffer	FEAT_RME is implemented
0b011111	Trace buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	
0b100101	Stage 2 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [25:23]

Reserved, RES0.

IRQ, bit [22]

Maintenance status. Indicates that a trace buffer management event has been recorded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRQ	Meaning
0b0	No trace buffer management event for EL3 has been recorded.
0b1	A trace buffer management event for EL3 has been recorded.

When FEAT_TRBE_EXC is implemented, this field indicates a management event for EL3.

If the TRBE Profiling exception for EL3 is enabled, then when this field is 1, a TRBE Profiling exception for EL3 is pending

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

TRG, bit [21]

Triggered.

TRG	Meaning
0b0	No Detected Trigger has been observed since this field was last cleared to zero.
0b1	A Detected Trigger has been observed since this field was last cleared to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

WRAP, bit [20]

Wrapped.

WRAP	Meaning
0b0	The current write pointer has not wrapped since this field was last cleared to zero.
0b1	The current write pointer has wrapped since this field was last cleared to zero.

For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in the current write pointer, if the current write pointer is equal to the Limit pointer minus one, the current write pointer is wrapped by setting it to the Base pointer, and this field is set to 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External Abort.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Trace Buffer Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Trace Buffer Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. From Armv9.3, this field is not set to 1 by the PE for any other trace buffer management event.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Stopped.

S	Meaning
0b0	Collection has not been stopped.
0b1	Collection is stopped.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [16]

Reserved, RES0.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the trace buffer management event.

The syndrome contents for each trace buffer management event are described in the following sections.

MSS encoding for other trace buffer management events

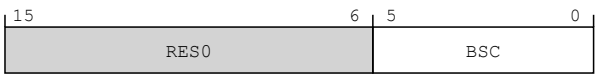

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Trace buffer status code

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BSC	Meaning	Applies when
0b000000	Collection not stopped, or access not allowed.	
0b000001	Trace buffer filled. Collection stopped because the current write pointer wrapped to the base pointer and the trace buffer mode is Fill mode.	
0b000010	Trigger Event. Collection stopped because of a Trigger Event. See TRBTRG_EL1 for more information.	
0b000011	Manual Stop. Collection stopped because of a Manual Stop event. See TRBCR.ManStop for more information.	FEAT_TRBE_EXT is implemented
0b000100	Buffer size. The requested trace buffer size was too large.	

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

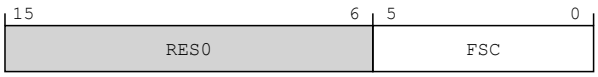

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to trace buffer

15	0
RES0	

Bits [15:0]

Reserved, RES0.

MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

15	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Accessing TRBSR_EL3

The PE might ignore a write to TRBSR_EL3 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBSR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1001	0b1011	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TRBE_EXC) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = TRBSR_EL3();
end;
```

```
MSR TRBSR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1001	0b1011	0b011

```
if !(IsFeatureImplemented(FEAT_TRBE_EXC) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    TRBSR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.10 TRBTRG_EL1, Trace Buffer Trigger Counter Register

The TRBTRG_EL1 characteristics are:

Purpose

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

Configuration

When FEAT_TRBE_EXT is implemented, AArch64 System register TRBTRG_EL1 bits [63:0] are architecturally mapped to External register TRBTRG_EL1[63:0].

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBTRG_EL1 are UNDEFINED.

Attributes

TRBTRG_EL1 is a 64-bit register.

Field descriptions

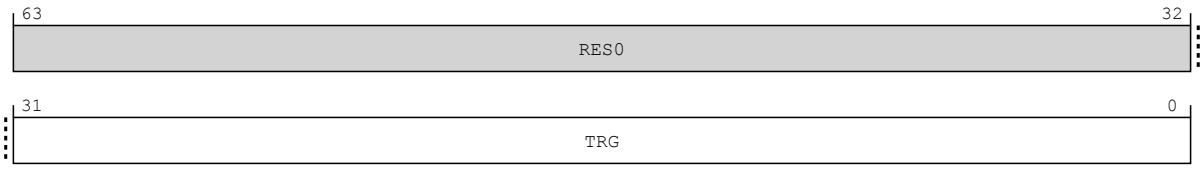

Bits [63:32]

Reserved, RES0.

TRG, bits [31:0]

Trigger count.

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

TRBTRG_EL1 decrements by 1 for every byte of trace written to the trace buffer when all of the following are true:

- TRBTRG_EL1 is nonzero.
- TRBSR_EL1.TRG is 1.

The architecture places restrictions on the values that software can write to the counter.

Note

As a result of the restrictions an implementation might treat some of TRG[M:0] as RES0, where M is defined by TRBIDR_EL1.Align.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBTRG_EL1

The PE might ignore a write to TRBTRG_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBLIMITR_EL1.XE is 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRBTRG_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b110

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRBTRG_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBTRG_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBTRG_EL1();
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRBTRG_EL1();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TRBTRG_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1011	0b110

```
if !IsFeatureImplemented(FEAT_TRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRBTRG_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2TB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBTRG_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSTBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSTBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBTRG_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRBTRG_EL1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.11 TRCACATR<n>, Trace Address Comparator Access Type Register <n>, n = 0 - 15

The TRCACATR<n> characteristics are:

Purpose

Defines the type of access for the corresponding TRCACVR<n> Register. This register configures the context type, Exception levels, alignment, masking that is applied by the Address Comparator, and how the Address Comparator behaves when it is one half of an Address Range Comparator.

Configuration

AArch64 System register TRCACATR<n> bits [63:0] are architecturally mapped to External register TRCACATR<n>[63:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and (UInt(TRCIDR4.NUMACPAIRS) * 2) > n. Otherwise, direct accesses to TRCACATR<n> are UNDEFINED.

Attributes

TRCACATR<n> is a 64-bit register.

Field descriptions

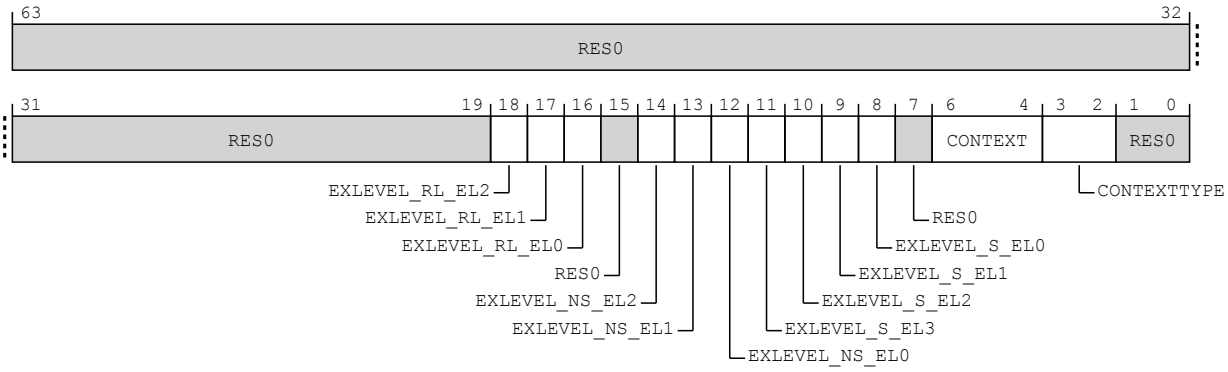

Bits [63:19]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [18]

When FEAT_RME is implemented:

Realm EL2 address comparison control. Controls whether a comparison can occur at EL2 in Realm state.

EXLEVEL_RL_EL2	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator performs comparisons in Realm EL2. When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator does not perform comparisons in Realm EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_RL_EL2	Meaning
0b1	When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator does not perform comparisons in Realm EL2. When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator performs comparisons in Realm EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL1, bit [17]

When FEAT_RME is implemented:

Realm EL1 address comparison control. Controls whether a comparison can occur at EL1 in Realm state.

EXLEVEL_RL_EL1	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator performs comparisons in Realm EL1. When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator does not perform comparisons in Realm EL1.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator does not perform comparisons in Realm EL1. When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator performs comparisons in Realm EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [16]

When FEAT_RME is implemented:

Realm EL0 address comparison control. Controls whether a comparison can occur at EL0 in Realm state.

EXLEVEL_RL_EL0	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator performs comparisons in Realm EL0. When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator does not perform comparisons in Realm EL0.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator does not perform comparisons in Realm EL0. When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator performs comparisons in Realm EL0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [14]

When Non-secure EL2 is implemented:

Non-secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Non-secure state.

EXLEVEL_NS_EL2	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL2.
0b1	The Address Comparator does not perform comparisons in Non-secure EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [13]

When Non-secure EL1 is implemented:

Non-secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Non-secure state.

EXLEVEL_NS_EL1	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL1.
0b1	The Address Comparator does not perform comparisons in Non-secure EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL0, bit [12]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Non-secure EL0 is implemented:

Non-secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Non-secure state.

EXLEVEL_NS_EL0	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL0.
0b1	The Address Comparator does not perform comparisons in Non-secure EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL3, bit [11]

When EL3 is implemented:

EL3 address comparison control. Controls whether a comparison can occur at EL3.

EXLEVEL_S_EL3	Meaning
0b0	The Address Comparator performs comparisons at EL3.
0b1	The Address Comparator does not perform comparisons at EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL2, bit [10]

When Secure EL2 is implemented:

Secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Secure state.

EXLEVEL_S_EL2	Meaning
0b0	The Address Comparator performs comparisons in Secure EL2.
0b1	The Address Comparator does not perform comparisons in Secure EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL1, bit [9]

When Secure EL1 is implemented:

Secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Secure state.

EXLEVEL_S_EL1	Meaning
0b0	The Address Comparator performs comparisons in Secure EL1.
0b1	The Address Comparator does not perform comparisons in Secure EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL0, bit [8]

When Secure EL0 is implemented:

Secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Secure state.

EXLEVEL_S_EL0	Meaning
0b0	The Address Comparator performs comparisons in Secure EL0.
0b1	The Address Comparator does not perform comparisons in Secure EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

CONTEXT, bits [6:4]

When TRCIDR4.NUMCIDC != '0000' or TRCIDR4.NUMVMIDC != '0000':

Selects a Context Identifier Comparator or Virtual Context Identifier Comparator:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXT	Meaning	Applies when
0b000	Comparator 0.	
0b001	Comparator 1.	UInt(TRCIDR4.NUMCIDC) > 1 or UInt(TRCIDR4.NUMVMIDC) > 1
0b010	Comparator 2.	UInt(TRCIDR4.NUMCIDC) > 2 or UInt(TRCIDR4.NUMVMIDC) > 2
0b011	Comparator 3.	UInt(TRCIDR4.NUMCIDC) > 3 or UInt(TRCIDR4.NUMVMIDC) > 3
0b100	Comparator 4.	UInt(TRCIDR4.NUMCIDC) > 4 or UInt(TRCIDR4.NUMVMIDC) > 4
0b101	Comparator 5.	UInt(TRCIDR4.NUMCIDC) > 5 or UInt(TRCIDR4.NUMVMIDC) > 5
0b110	Comparator 6.	UInt(TRCIDR4.NUMCIDC) > 6 or UInt(TRCIDR4.NUMVMIDC) > 6
0b111	Comparator 7.	UInt(TRCIDR4.NUMCIDC) > 7 or UInt(TRCIDR4.NUMVMIDC) > 7

The width of this field is dependent on the maximum number of Context Identifier Comparators or Virtual Context Identifier Comparators implemented. Unimplemented bits are RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CONTEXTTYPE, bits [3:2]

When TRCIDR4.NUMCIDC != '0000' or TRCIDR4.NUMVMIDC != '0000':

Controls whether the Address Comparator is dependent on a Context Identifier Comparator, a Virtual Context Identifier Comparator, or both comparisons.

CONTEXTTYPE	Meaning	Applies when
0b00	The Address Comparator is not dependent on the Context Identifier Comparators or Virtual Context Identifier Comparators.	
0b01	The Address Comparator is dependent on the Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Context Identifier Comparator and the address comparison match.	TRCIDR4.NUMCIDC != ‘0000’
0b10	The Address Comparator is dependent on the Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Virtual Context Identifier Comparator and the address comparison match.	TRCIDR4.NUMVMIDC != ‘0000’

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXTTYPE	Meaning	Applies when
0b11	The Address Comparator is dependent on the Context Identifier Comparator and Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if the Context Identifier Comparator, the Virtual Context Identifier Comparator, and address comparison all match.	TRCIDR4.NUMCIDC != '0000' and TRCIDR4.NUMVMIDC != '0000'

If TRCIDR4.NUMCIDC == 0b0000, then bit [2] is RES0.

If TRCIDR4.NUMVMIDC == 0b0000, then bit [3] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [1:0]

Reserved, RES0.

Accessing TRCACATR<n>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<a>.GROUP == 0b0100 and TRCRSCTLR<a>.SAC[n] == 1.
- TRCRSCTLR<a>.GROUP == 0b0101 and TRCRSCTLR<a>.ARC[n/2] == 1.
- TRCVIIECTLR.EXCLUDE[n/2] == 1.
- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
- TRCVISSCTLR.STOP[n] == 1.
- TRCSSCCR<>.ARC[n/2] == 1.
- TRCSSCCR<>.SAC[n] == 1.
- TRCQCTLR.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCACATR<m> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0010	m[2:0]:0b0	0b01:m[3]

```
let m:integer = UInt(op2[0] :: CRm[3:1]);

if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL1, 0x18);
        elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().TRC == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCACATR(m);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
            Undefined();
        elsif CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCACATR(m);
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TTA == '1' then
            AArch64_SystemAccessTrap(EL3, 0x18);
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCACATR(m);
        end;
    end;
```

```
MSR TRCACATR<m>, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0010	m[2:0]:0b0	0b01:m[3]

```
let m:integer = UInt(op2[0] :: CRm[3:1]);

if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCACATR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCACATR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCACATR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.12 TRCACVR<n>, Trace Address Comparator Value Register <n>, n = 0 - 15

The TRCACVR<n> characteristics are:

Purpose

Contains the address value.

Configuration

AArch64 System register TRCACVR<n> bits [63:0] are architecturally mapped to External register TRCACVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and (UInt(TRCIDR4.NUMACPAIRS) * 2) > n. Otherwise, direct accesses to TRCACVR<n> are UNDEFINED.

Attributes

TRCACVR<n> is a 64-bit register.

Field descriptions

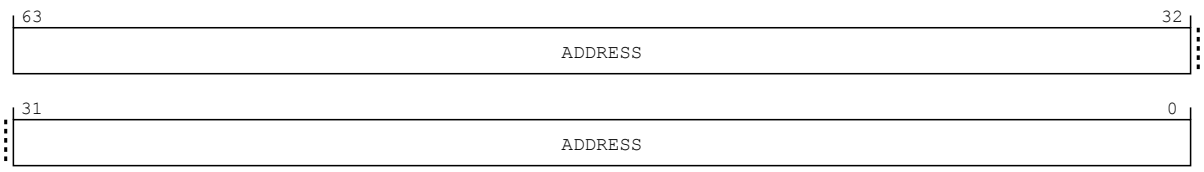

ADDRESS, bits [63:0]

Address Value.

The Address Comparators can support implementations that use multiple address widths. When the trace unit compares the ADDRESS field with an address that has a width less than this field, then the address must be zero-extended to the ADDRESS field width. The trace unit then compares all implemented bits. For example, in a system that supports both 32-bit and 64-bit addresses, when the PE is in AArch32 state the comparator must zero-extend the 32-bit address and compare against the full 64 bits that are stored in TRCACVR<n>.ADDRESS. This requires that the trace analyzer always programs all implemented bits of TRCACVR<n>.ADDRESS.

The result of writing a value other than all zeros or all ones to ADDRESS at bits[63:P] is an UNKNOWN value, where P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The result of writing a value of all zeros or all ones to ADDRESS at bits[63:P] is the written value, and a read of the register returns the written value.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCACVR<n>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<a>.GROUP == `0b0100` and TRCRSCTLR<a>.SAC[n] == 1.
- TRCRSCTLR<a>.GROUP == `0b0101` and TRCRSCTLR<a>.ARC[n/2] == 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCVIIECTLR.EXCLUDE[n/2] == 1.
- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
- TRCVISSCTLR.STOP[n] == 1.
- TRCSSCCR<>.ARC[n/2] == 1.
- TRCSSCCR<>.SAC[n] == 1.
- TRCQCTLR.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCACVR<m> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0010	m[2:0]:0b0	0b00:m[3]

```
let m:integer = UInt(op2[0] :: CRm[3:1]);

if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCACVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCACVR(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCACVR(m);
    end;
end;
```

```
MSR TRCACVR<m>, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0010	m[2:0]:0b0	0b00:m[3]

```
let m:integer = UInt(op2[0] :: CRm[3:1]);

if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCACVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        TRCACVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCACVR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.13 TRCAUTHSTATUS, Trace Authentication Status Register

The TRCAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

For additional information, see the CoreSight Architecture Specification.

Configuration

AArch64 System register TRCAUTHSTATUS bits [31:0] are architecturally mapped to External register TRCAUTHSTATUS[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCAUTHSTATUS are UNDEFINED.

Attributes

TRCAUTHSTATUS is a 64-bit register.

Field descriptions

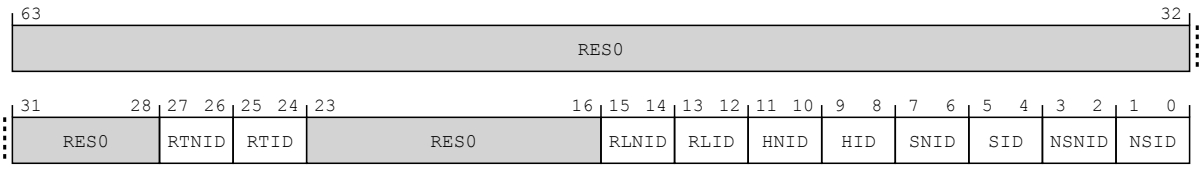

Bits [63:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTNID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
0b00	Not implemented.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLNID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLID, bits [13:12]

Realm invasive debug.

RLID	Meaning
0b00	Not implemented.

HNID, bits [11:10]

Hyp Non-invasive Debug. Indicates whether a separate enable control for EL2 non-invasive debug features is implemented and enabled.

HNID	Meaning
0b00	Separate Hyp non-invasive debug enable not implemented, or EL2 non-invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

HID, bits [9:8]

Hyp Invasive Debug. Indicates whether a separate enable control for EL2 invasive debug features is implemented and enabled.

HID	Meaning
0b00	Separate Hyp invasive debug enable not implemented, or EL2 invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

SNID, bits [7:6]

Secure Non-invasive Debug. Indicates whether Secure non-invasive debug features are implemented and enabled.

SNID	Meaning
0b00	Secure non-invasive debug features not implemented.
0b10	Implemented and disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SNID	Meaning
0b11	Implemented and enabled.

All other values are reserved.

When Secure state is implemented, this field reads as 0b10 or 0b11 depending whether Secure non-invasive debug is enabled.

When Secure state is not implemented, this field reads as 0b00.

SID, bits [5:4]

Secure Invasive Debug. Indicates whether Secure invasive debug features are implemented and enabled.

SID	Meaning
0b00	Secure invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

NSNID, bits [3:2]

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug features are implemented and enabled.

NSNID	Meaning
0b00	Non-secure non-invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

When Non-secure state is implemented, this field reads as 0b11.

When Non-secure state is not implemented, this field reads as 0b00.

NSID, bits [1:0]

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug features are implemented and enabled.

NSID	Meaning
0b00	Non-secure invasive debug features not implemented.
0b10	Implemented and disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSID	Meaning
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

Accessing TRCAUTHSTATUS

For implementations that support multiple access mechanisms, different access mechanisms can return different values for reads of TRCAUTHSTATUS if the authentication signals have changed and that change has not yet been synchronized by a Context synchronization event. This scenario can happen if, for example, the external debugger view is implemented separately from the system instruction view to allow for separate power domains, and so observes changes on the signals differently.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCAUTHSTATUS
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b1110	0b110

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCAUTHSTATUS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCAUTHSTATUS();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCAUTHSTATUS();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCAUTHSTATUS();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.14 TRCAUXCTLR, Trace Auxiliary Control Register

The TRCAUXCTLR characteristics are:

Purpose

The function of this register is IMPLEMENTATION DEFINED.

Configuration

AArch64 System register TRCAUXCTLR bits [31:0] are architecturally mapped to External register TRCAUXCTLR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCAUXCTLR are UNDEFINED.

Attributes

TRCAUXCTLR is a 64-bit register.

Field descriptions

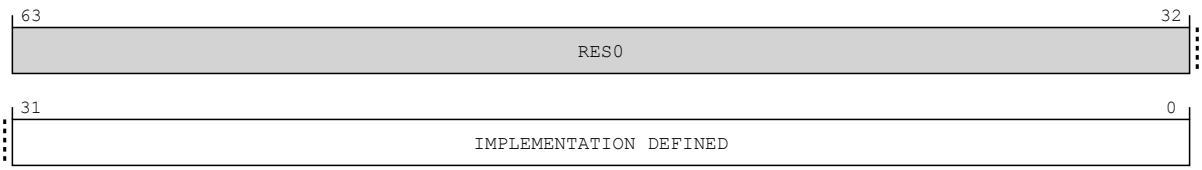

Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to the expression `0x00000000`.

Accessing TRCAUXCTLR

If this register is nonzero then it might cause the behavior of a trace unit to contradict this architecture specification. See the documentation of the specific implementation for information about the IMPLEMENTATION DEFINED support for this register.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCAUXCTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0110	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCAUXCTLR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCAUXCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCAUXCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCAUXCTLR();
    end;
end;
```

```
MSR TRCAUXCTLR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0110	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCAUXCTLR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCAUXCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCAUXCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCAUXCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.15 TRCBBCTLR, Trace Branch Broadcast Control Register

The TRCBBCTLR characteristics are:

Purpose

Controls the regions in the memory map where branch broadcasting is active.

Configuration

AArch64 System register TRCBBCTLR bits [31:0] are architecturally mapped to External register TRCBBCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, TRCIDR0.TRCBB == '1', and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCBBCTLR are UNDEFINED.

Attributes

TRCBBCTLR is a 64-bit register.

Field descriptions

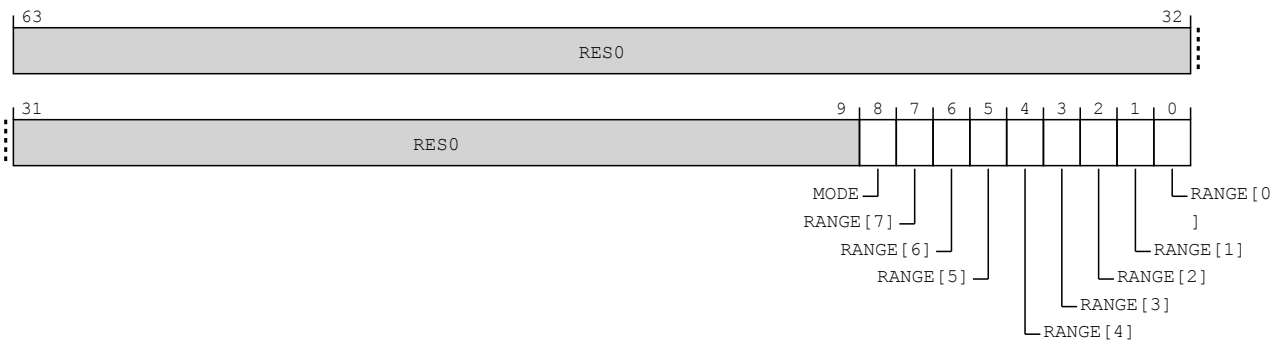

Bits [63:9]

Reserved, RES0.

MODE, bit [8]

Mode.

MODE	Meaning
0b0	Exclude Mode. Branch broadcasting is not active for instructions in the address ranges defined by TRCBBCTLR.RANGE. If TRCBBCTLR.RANGE == 0x00 then branch broadcasting is active for all instructions.
0b1	Include Mode. Branch broadcasting is active for instructions in the address ranges defined by TRCBBCTLR.RANGE. If TRCBBCTLR.RANGE == 0x00 then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE. That is, the trace unit might or might not consider any instructions to be in a branch broadcasting region.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RANGE[<m>] , bits [m], for m = 7 to 0

Selects whether Address Range Comparator <m> is used with branch broadcasting.

RANGE[<m>]	Meaning
0b0	The address range that Address Range Comparator <m> defines, is not selected.
0b1	The address range that Address Range Comparator <m> defines, is selected.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCBBCTLR

Must be programmed if TRCCONFIGR.BB == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCBBCTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1111	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().TRCBB == '1' &&
     ↪UInt(TRCIDR4().NUMACPAIRS) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCBBCTLR();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
            Undefined();
        elsif CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCBBCTLR();
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TTA == '1' then
            AArch64_SystemAccessTrap(EL3, 0x18);
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCBBCTLR();
        end;
    end;
```

MSR TRCBBCTLR, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1111	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().TRCBB == '1' &&
     ↪UInt(TRCIDR4().NUMACPAIRS) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCBBCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCBBCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCBBCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.16 TRCCCCTLR, Trace Cycle Count Control Register

The TRCCCCTLR characteristics are:

Purpose

Set the threshold value for cycle counting.

Configuration

AArch64 System register TRCCCCTLR bits [31:0] are architecturally mapped to External register TRCCCCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR0.TRCCCI == '1'. Otherwise, direct accesses to TRCCCCTLR are UNDEFINED.

Attributes

TRCCCCTLR is a 64-bit register.

Field descriptions

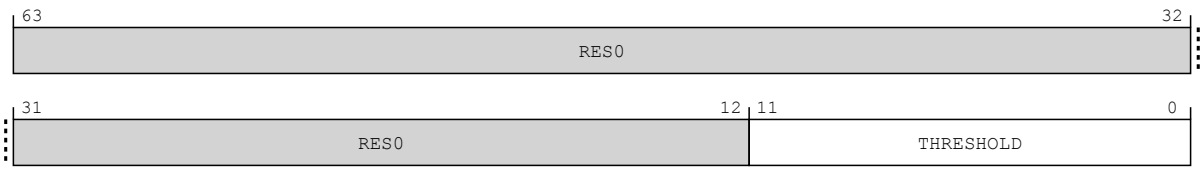

Bits [63:12]

Reserved, RES0.

THRESHOLD, bits [11:0]

Sets the threshold value for instruction trace cycle counting.

The minimum threshold value that can be programmed into THRESHOLD is given in TRCIDR3.CCITMIN. If the THRESHOLD value is smaller than the value in TRCIDR3.CCITMIN then the behavior is CONSTRAINED UNPREDICTABLE. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

Writing a value of zero when TRCCONFIGR.CCI enables instruction trace cycle counting results in CONSTRAINED UNPREDICTABLE behavior. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCCCTLR

Must be programmed if TRCCONFIGR.CCI == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCCCTLR
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1110	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().TRCCCI == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCCCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCCCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCCCTLR();
    end;
end;
```

```
MSR TRCCCCTLR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1110	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().TRCCCI == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCCCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCCCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCCCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.17 TRCCIDCCTLR0, Trace Context Identifier Comparator Control Register 0

The TRCCIDCCTLR0 characteristics are:

Purpose

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 0 to 3.

Configuration

AArch64 System register TRCCIDCCTLR0 bits [31:0] are architecturally mapped to External register TRCCIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, UInt(TRCIDR4.NUMCIDC) > `0x0`, and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR0 are UNDEFINED.

Attributes

TRCCIDCCTLR0 is a 64-bit register.

Field descriptions

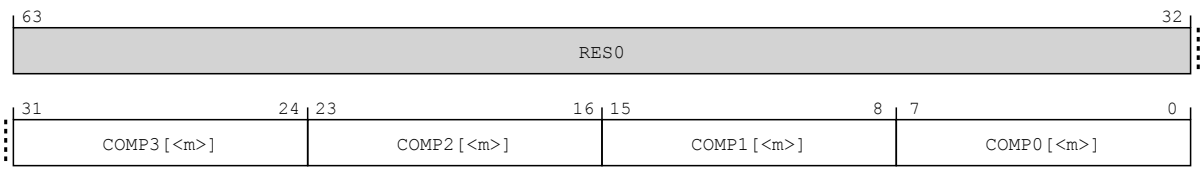

Bits [63:32]

Reserved, RES0.

COMP3[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 3:

TRCCIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR3. Each bit in this field corresponds to a byte in TRCCIDCVR3.

COMP3[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

COMP2[<m>] , bits [m+16], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 2:

TRCCIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR2. Each bit in this field corresponds to a byte in TRCCIDCVR2.

COMP2[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP1[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 1:

TRCCIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR1. Each bit in this field corresponds to a byte in TRCCIDCVR1.

COMP1[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP0[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 0:

TRCCIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR0. Each bit in this field corresponds to a byte in TRCCIDCVR0.

COMP0[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCCIDCCTLR0

If software uses the TRCCIDCVR<n> registers, for n = 0 to 3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCIDCCTLR0
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMCIDC) >
     ↪0x0 && UInt(TRCIDR2().CIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().TRC == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCCIDCCTLR0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
            Undefined();
        elsif CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCCIDCCTLR0();
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TTA == '1' then
            AArch64_SystemAccessTrap(EL3, 0x18);
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCCIDCCTLR0();
        end;
    end;
```

```
MSR TRCCIDCCTLR0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMCIDC) >
     ↪0x0 && UInt(TRCIDR2().CIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCCTLR0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCCTLR0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCCTLR0() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.18 TRCCIDCCTLR1, Trace Context Identifier Comparator Control Register 1

The TRCCIDCCTLR1 characteristics are:

Purpose

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 4 to 7.

Configuration

AArch64 System register TRCCIDCCTLR1 bits [31:0] are architecturally mapped to External register TRCCIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, UInt(TRCIDR4.NUMCIDC) > `0x4`, and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR1 are UNDEFINED.

Attributes

TRCCIDCCTLR1 is a 64-bit register.

Field descriptions

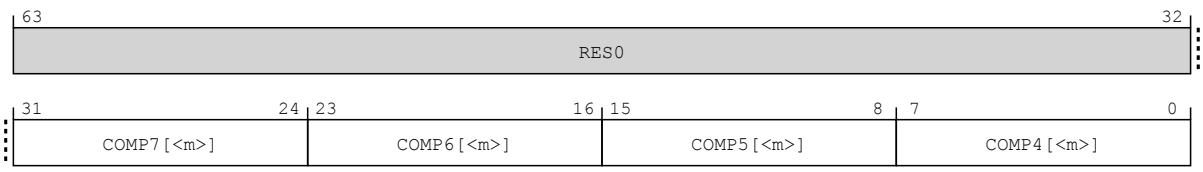

Bits [63:32]

Reserved, RES0.

COMP7[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 7:

TRCCIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR7. Each bit in this field corresponds to a byte in TRCCIDCVR7.

COMP7[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

COMP6[<m>] , bits [m+16], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 6:

TRCCIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR6. Each bit in this field corresponds to a byte in TRCCIDCVR6.

COMP6[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP5[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 5:

TRCCIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR5. Each bit in this field corresponds to a byte in TRCCIDCVR5.

COMP5[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP4[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 4:

TRCCIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR4. Each bit in this field corresponds to a byte in TRCCIDCVR4.

COMP4[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCCIDCCTLR1

If software uses the TRCCIDCVR<n> registers, for n = 4 to 7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCIDCCTLR1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMCIDC) >
     ↪0x4 && UInt(TRCIDR2().CIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().TRC == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCCIDCCTLR1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
            Undefined();
        elsif CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCCIDCCTLR1();
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TTA == '1' then
            AArch64_SystemAccessTrap(EL3, 0x18);
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCCIDCCTLR1();
        end;
    end;
```

```
MSR TRCCIDCCTLR1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMCIDC) >
     ↪0x4 && UInt(TRCIDR2().CIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCCTLR1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCCTLR1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCCTLR1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.19 TRCCIDCVR<n>, Trace Context Identifier Comparator Value Registers <n>, n = 0 - 7

The TRCCIDCVR<n> characteristics are:

Purpose

Contains a Context identifier value.

Configuration

AArch64 System register TRCCIDCVR<n> bits [63:0] are architecturally mapped to External register TRCCIDCVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMCIDC) > n. Otherwise, direct accesses to TRCCIDCVR<n> are UNDEFINED.

Attributes

TRCCIDCVR<n> is a 64-bit register.

Field descriptions

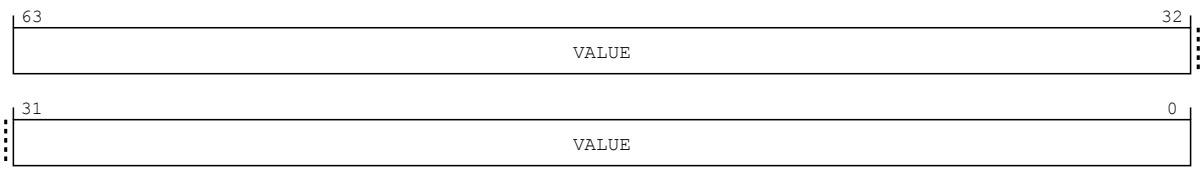

VALUE, bits [63:0]

Context identifier value. The width of this field is indicated by TRCIDR2.CIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Context identifier is zero until the PE updates the Context identifier.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCIDCVR<n>

Must be programmed if any of the following are true:

- TRCRSCTLR<a>.GROUP == `0b0110` and TRCRSCTLR<a>.CID[n] == 1.
- TRCACATR<a>.CONTEXTTYPE == `0b01` or `0b11` and TRCACATR<a>.CONTEXT == n.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCIDCVR<m> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
`0b10`	`0b001`	`0b0011`	m[2:0]:`0b0`	`0b000`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:1]);

if m >= NUM_TRACE_CONTEXT_IDENTIFIER_COMPARATORS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCIDCVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCIDCVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCIDCVR(m);
    end;
end;
```

```
MSR TRCCIDCVR<m>, <Xt> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	m[2:0]:0b0	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:1]);

if m >= NUM_TRACE_CONTEXT_IDENTIFIER_COMPARATORS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCIDCVR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.20 TRCCLAIMCLR, Trace Claim Tag Clear Register

The TRCCLAIMCLR characteristics are:

Purpose

In conjunction with TRCCLAIMSET, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configuration

AArch64 System register TRCCLAIMCLR bits [63:0] are architecturally mapped to AArch64 System register TRCCLAIMSET[63:0].

AArch64 System register TRCCLAIMCLR bits [31:0] are architecturally mapped to External register TRCCLAIMCLR[31:0].

AArch64 System register TRCCLAIMCLR bits [31:0] are architecturally mapped to External register TRCCLAIMSET[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCCLAIMCLR are UNDEFINED.

Attributes

TRCCLAIMCLR is a 64-bit register.

Field descriptions

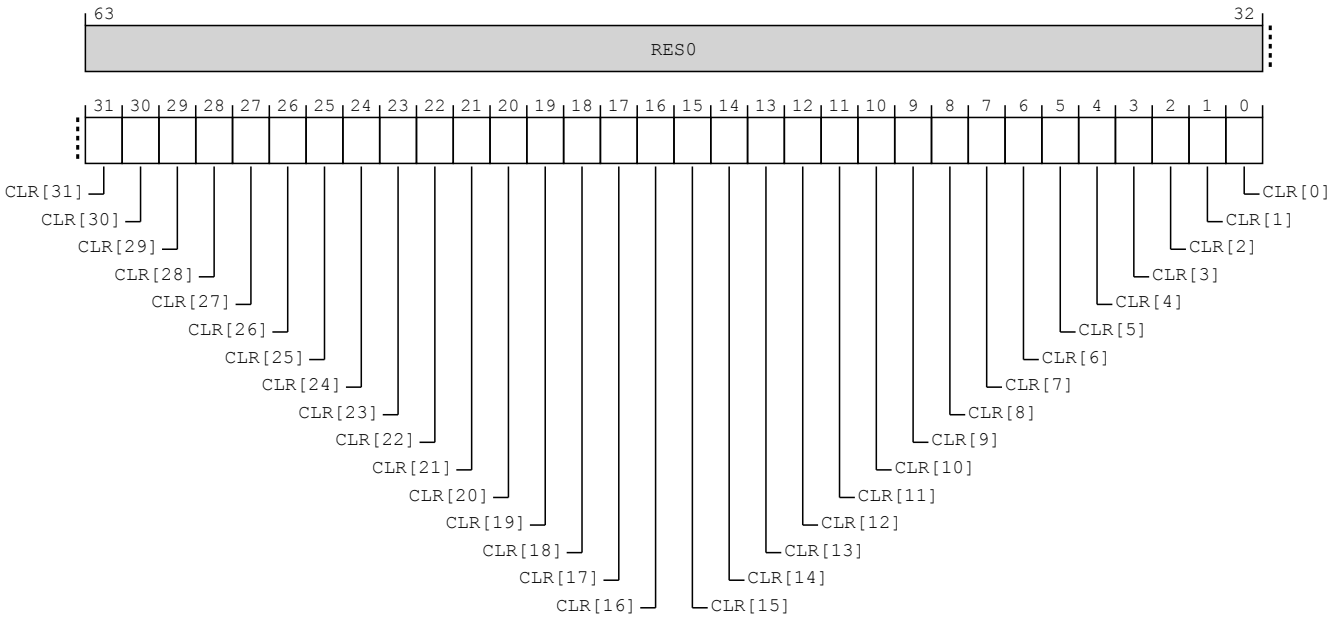

Bits [63:32]

Reserved, RES0.

CLR[<m>] , bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLR[<m>]	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in TRCCLAIMSET.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Accessing this field has the following behavior:

- When m >= NUM_TRC_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing TRCCLAIMCLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCLAIMCLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b1001	0b110

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCLAIMCLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCLAIMCLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCLAIMCLR();
    end;
end;
```

MSR TRCCLAIMCLR, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b1001	0b110

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCLAIMCLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCLAIMCLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCLAIMCLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.21 TRCCLAIMSET, Trace Claim Tag Set Register

The TRCCLAIMSET characteristics are:

Purpose

In conjunction with TRCCLAIMCLR, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configuration

The number of claim tag bits implemented is IMPLEMENTATION DEFINED. Arm recommends that implementations support a minimum of four claim tag bits, that is, SET[3:0] reads as `0b1111`.

AArch64 System register TRCCLAIMSET bits [63:0] are architecturally mapped to AArch64 System register TRCCLAIMCLR[63:0].

AArch64 System register TRCCLAIMSET bits [31:0] are architecturally mapped to External register TRCCLAIMSET[31:0].

AArch64 System register TRCCLAIMSET bits [31:0] are architecturally mapped to External register TRCCLAIMCLR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCCLAIMSET are UNDEFINED.

Attributes

TRCCLAIMSET is a 64-bit register.

Field descriptions

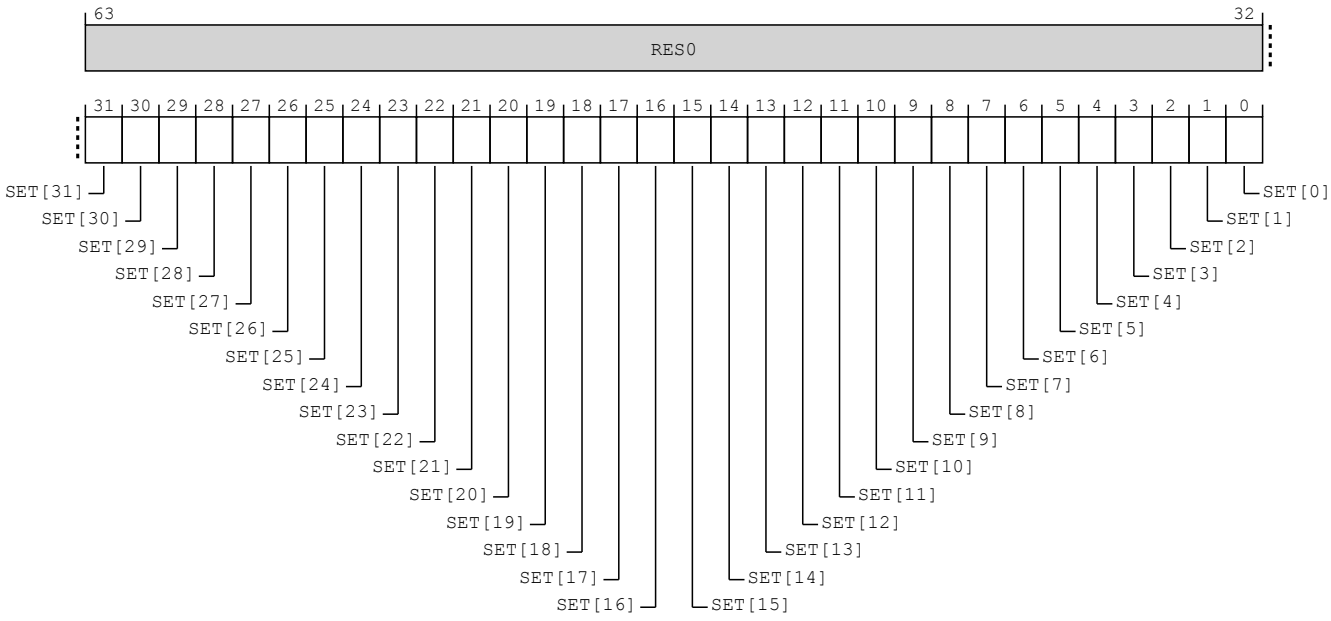

Bits [63:32]

Reserved, RES0.

SET[<m>] , bits [m], for m = 31 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

SET[<m>]	Meaning
0b0	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

Accessing this field has the following behavior:

- When m >= NUM_TRC_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/W1S.

Accessing TRCCLAIMSET

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCLAIMSET
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b1000	0b110

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCLAIMSET();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCLAIMSET();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCLAIMSET();
    end;
end;
```

```
MSR TRCCLAIMSET, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b1000	0b110

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCCLAIM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCLAIMSET() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCLAIMSET() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCLAIMSET() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.22 TRCCNTCTLR<n>, Trace Counter Control Register <n>, n = 0 - 3

The TRCCNTCTLR<n> characteristics are:

Purpose

Controls the operation of Counter <n>.

Configuration

AArch64 System register TRCCNTCTLR<n> bits [31:0] are architecturally mapped to External register TRCCNTCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTCTLR<n> are UNDEFINED.

Attributes

TRCCNTCTLR<n> is a 64-bit register.

Field descriptions

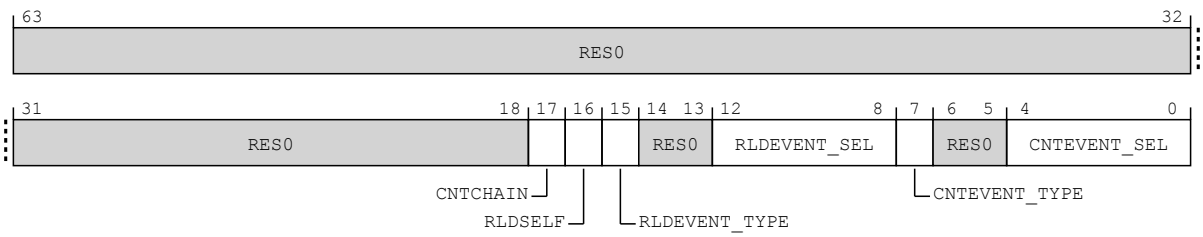

Bits [63:18]

Reserved, RES0.

CNTCHAIN, bit [17]

When (n MOD 2) != 0:

For TRCCNTCTLR3 and TRCCNTCTLR1, this field controls whether the Counter decrements when a reload event occurs for Counter <n-1>.

CNTCHAIN	Meaning
0b0	The Counter does not decrement when a reload event for Counter <n-1> occurs.
0b1	Counter <n> decrements when a reload event for Counter <n-1> occurs. This concatenates Counter <n> and Counter <n-1>, to provide a larger count value.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLDSELF, bit [16]

Controls whether a reload event occurs for the Counter, when the Counter reaches zero.

RLDSELF	Meaning
`0b0`	Normal mode. The Counter is in Normal mode.
`0b1`	Self-reload mode. The Counter is in Self-reload mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RLDEVENT_TYPE, bit [15]

Selects an event, that when it occurs causes a reload event for Counter <n>

Chooses the type of Resource Selector.

RLDEVENT_TYPE	Meaning
`0b0`	A single Resource Selector. TRCCNTCTLR<n>.RLDEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCCNTCTLR<n>.RLDEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR<n>.RLDEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

RLDEVENT_SEL, bits [12:8]

Selects an event, that when it occurs causes a reload event for Counter <n>

Defines the selected Resource Selector or pair of Resource Selectors. TRCCNTCTLR<n>.RLDEVENT.TYPE controls whether TRCCNTCTLR<n>.RLDEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTEVENT_TYPE, bit [7]

Selects an event, that when it occurs causes Counter <n> to decrement.

Chooses the type of Resource Selector.

CNTEVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCCNTCTLR<n>.CNTEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCCNTCTLR<n>.CNTEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR<n>.CNTEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

CNTEVENT_SEL, bits [4:0]

Selects an event, that when it occurs causes Counter <n> to decrement.

Defines the selected Resource Selector or pair of Resource Selectors. TRCCNTCTLR<n>.CNTEVENT.TYPE controls whether TRCCNTCTLR<n>.CNTEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTCTLR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == 0b0010 and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCNTCTLR<m> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b01:m[1:0]	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_COUNTERS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTCTLR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTCTLR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTCTLR(m);
    end;
end;
```

```
MSR TRCCNTCTLR<m>, <Xt> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b01:m[1:0]	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_COUNTERS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTCTLR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTCTLR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTCTLR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.23 TRCCNTRLDVR<n>, Trace Counter Reload Value Register <n>, n = 0 - 3

The TRCCNTRLDVR<n> characteristics are:

Purpose

This sets or returns the reload count value for Counter <n>.

Configuration

AArch64 System register TRCCNTRLDVR<n> bits [31:0] are architecturally mapped to External register TRCCNTRLDVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTRLDVR<n> are UNDEFINED.

Attributes

TRCCNTRLDVR<n> is a 64-bit register.

Field descriptions

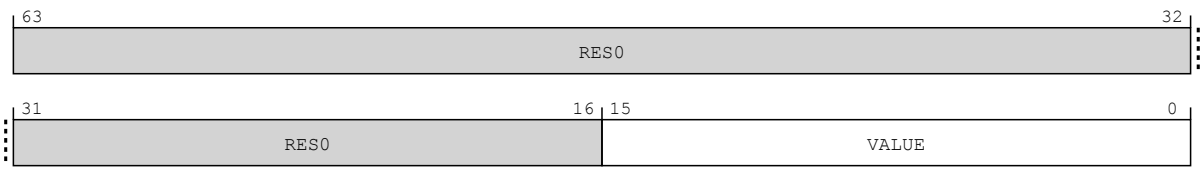

Bits [63:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the reload value for Counter <n>. When a reload event occurs for Counter <n> then the trace unit copies the VALUE<n> field into Counter <n>.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTRLDVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCNTRLDVR<m> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b00:m[1:0]	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_COUNTERS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTRLDVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTRLDVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTRLDVR(m);
    end;
end;
```

```
MSR TRCCNTRLDVR<m>, <Xt> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b00:m[1:0]	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_COUNTERS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTRLDVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTRLDVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTRLDVR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.24 TRCCNTVR<n>, Trace Counter Value Register <n>, n = 0 - 3

The TRCCNTVR<n> characteristics are:

Purpose

This sets or returns the value of Counter <n>.

Configuration

AArch64 System register TRCCNTVR<n> bits [31:0] are architecturally mapped to External register TRCCNTVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTVR<n> are UNDEFINED.

Attributes

TRCCNTVR<n> is a 64-bit register.

Field descriptions

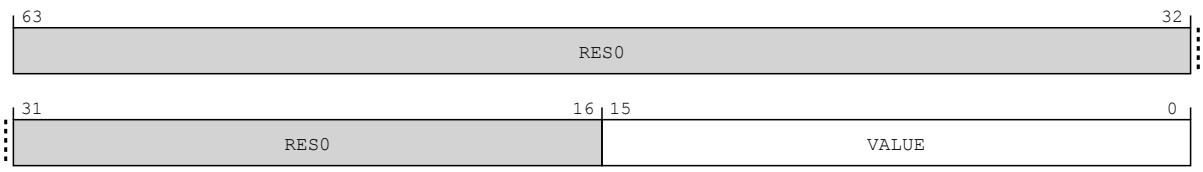

Bits [63:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the count value of Counter.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCNTVR<m> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b10:m[1:0]	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_COUNTERS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCCNTVRn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCNTVR(m);
    end;
end;
```

```
MSR TRCCNTVR<m>, <Xt> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b10:m[1:0]	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_COUNTERS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCCNTVRn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCNTVR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.25 TRCCONFIGR, Trace Configuration Register

The TRCCONFIGR characteristics are:

Purpose

Controls the tracing options.

Configuration

AArch64 System register TRCCONFIGR bits [31:0] are architecturally mapped to External register TRCCONFIGR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCCONFIGR are UNDEFINED.

Attributes

TRCCONFIGR is a 64-bit register.

Field descriptions

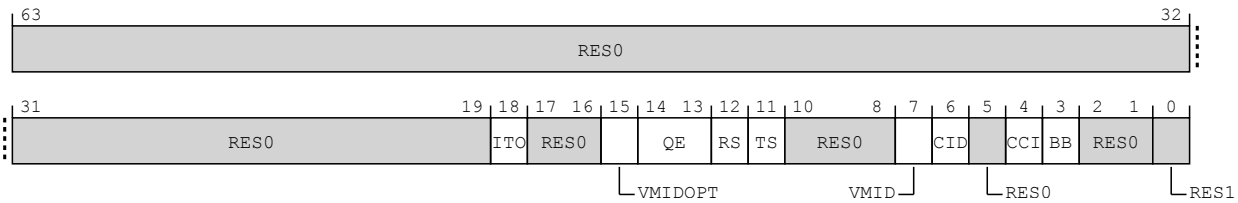

Bits [63:19]

Reserved, RES0.

ITO, bit [18]

When TRCIDR0.ITE == '1':

Instrumentation Trace Override.

ITO	Meaning
0b0	Instrumentation Trace Override disabled.
0b1	Instrumentation Trace Override enabled.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [17:16]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMIDOPT, bit [15]

When TRCIDR2.VMIDOPT == '01':

Virtual context identifier selection control.

VMIDOPT	Meaning
0b0	VTTBR_EL2.VMID is used as the Virtual context identifier.
0b1	CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

When TRCIDR2.VMIDOPT == '00':

Reserved, RES0.

Virtual context identifier selection control.

VTTBR_EL2.VMID is used as the Virtual context identifier.

When TRCIDR2.VMIDOPT == '10':

Reserved, RES1.

Virtual context identifier selection control.

CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

Otherwise:

Reserved, RES0.

QE, bits [14:13]

When TRCIDR0.QSUPP == '01':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b01	Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.

All other values are reserved.

When TRCIDR0.QSUPP == '10':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b11	Q elements with instruction counts are enabled. Q elements without instruction counts are enabled.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR0.QSUPP == '11':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b01	Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.
0b11	Q elements with instruction counts are enabled. Q elements without instruction counts are enabled.

All other values are reserved.

Otherwise:

Reserved, RES0.

RS, bit [12]

When TRCIDR0.RETSTACK == '1':

Return stack control.

RS	Meaning
0b0	Return stack is disabled.
0b1	Return stack is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TS, bit [11]

When TRCIDR0.TSSIZE != '00000':

Global timestamp tracing control.

TS	Meaning
0b0	Global timestamp tracing is disabled.
0b1	Global timestamp tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [10:8]

Reserved, RES0.

VMID, bit [7]

When TRCIDR2.VMIDSIZE != '00000':

Virtual context identifier tracing control.

VMID	Meaning
0b0	Virtual context identifier tracing is disabled.
0b1	Virtual context identifier tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CID, bit [6]

When TRCIDR2.CIDSIZE != '00000':

Context identifier tracing control.

CID	Meaning
0b0	Context identifier tracing is disabled.
0b1	Context identifier tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

CCI, bit [4]

When TRCIDR0.TRCCCI == '1':

Cycle counting instruction tracing control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCI	Meaning
0b0	Cycle counting instruction tracing is disabled.
0b1	Cycle counting instruction tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BB, bit [3]

When TRCIDR0.TRCBB == '1':

Branch broadcasting control.

BB	Meaning
0b0	Branch broadcasting is disabled.
0b1	Branch broadcasting is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [2:1]

Reserved, RES0.

Bit [0]

Reserved, RES1.

Accessing TRCCONFIGR

Must always be programmed.

TRCCONFIGR.QE must be set to 0b00 if TRCCONFIGR.BB is not 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCCONFIGR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0100	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCONFIGR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCONFIGR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCCONFIGR();
    end;
end;
```

```
MSR TRCCONFIGR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0100	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCONFIGR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCONFIGR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCCONFIGR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.26 TRCDEVARCH, Trace Device Architecture Register

The TRCDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

AArch64 System register TRCDEVARCH bits [31:0] are architecturally mapped to External register TRCDEVARCH[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCDEVARCH are UNDEFINED.

Attributes

TRCDEVARCH is a 64-bit register.

Field descriptions

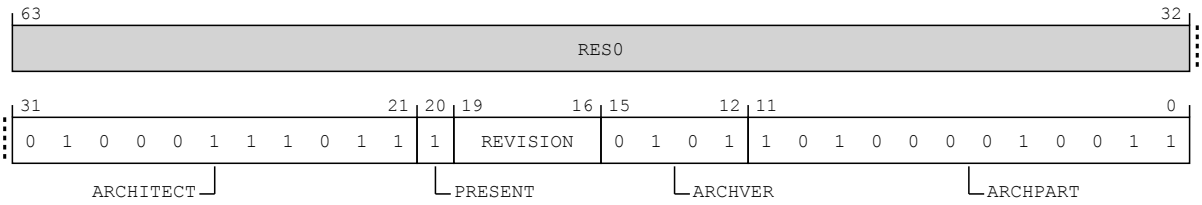

Bits [63:32]

Reserved, RES0.

ARCHITECT, bits [31:21]

Defines the architect of the component. For Trace, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the TRCDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION	Meaning
0b0000	ETEv1.0, FEAT_ETE.
0b0001	ETEv1.1, FEAT_ETEv1p1.
0b0010	ETEv1.2, FEAT_ETEv1p2.
0b0011	ETEv1.3, FEAT_ETEv1p3.

All other values are reserved.

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0101	ETEv1.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0x5.

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA13	Arm PE trace architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA13.

Access to this field is RO.

Accessing TRCDEVARCH

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCDEVARCH
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b1111	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCDEVARCH();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCDEVARCH();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCDEVARCH();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.27 TRCDEVID, Trace Device Configuration Register

The TRCDEVID characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

AArch64 System register TRCDEVID bits [31:0] are architecturally mapped to External register TRCDEVID[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCDEVID are UNDEFINED.

Attributes

TRCDEVID is a 64-bit register.

Field descriptions

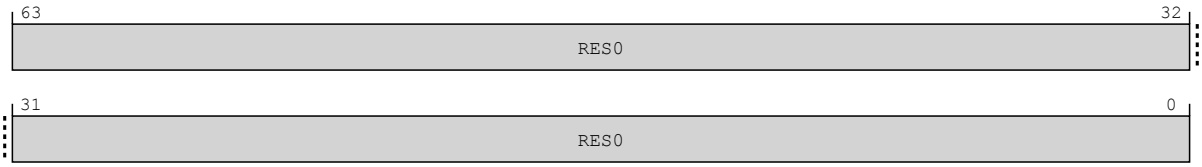

Bits [63:0]

Reserved, RES0.

Accessing TRCDEVID

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCDEVID
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0111	0b0010	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCDEVID();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCDEVID();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCDEVID();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.28 TRCEVENTCTL0R, Trace Event Control 0 Register

The TRCEVENTCTL0R characteristics are:

Purpose

Controls the generation of ETEEvents.

Configuration

AArch64 System register TRCEVENTCTL0R bits [31:0] are architecturally mapped to External register TRCEVENTCTL0R[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR4.NUMRSPAIR != ‘0000’. Otherwise, direct accesses to TRCEVENTCTL0R are UNDEFINED.

Attributes

TRCEVENTCTL0R is a 64-bit register.

Field descriptions

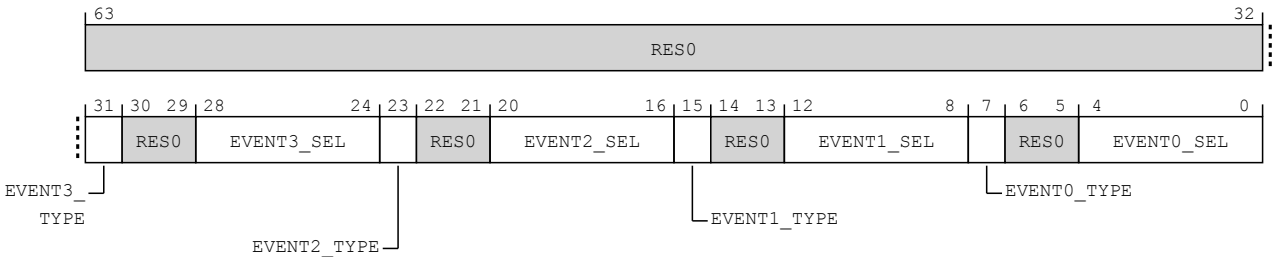

Bits [63:32]

Reserved, RES0.

EVENT3_TYPE, bit [31]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 3:

Chooses the type of Resource Selector.

EVENT3_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT3.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT3.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [30:29]

Reserved, RES0.

EVENT3_SEL, bits [28:24]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 3:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[3] == 1, then Event element 3 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.TYPE controls whether TRCEVENTCTL0R.EVENT3.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT2_TYPE, bit [23]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 2:

Chooses the type of Resource Selector.

EVENT2_TYPE	Meaning
`0b0`	A single Resource Selector. TRCEVENTCTL0R.EVENT2.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT2.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [22:21]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVENT2_SEL, bits [20:16]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 2:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[2] == 1, then Event element 2 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.TYPE controls whether TRCEVENTCTL0R.EVENT2.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT1_TYPE, bit [15]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 1:

Chooses the type of Resource Selector.

EVENT1_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT1.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT1.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT1.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [14:13]

Reserved, RES0.

EVENT1_SEL, bits [12:8]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 1:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[1] == 1, then Event element 1 is generated in the instruction trace element stream.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT1.TYPE controls whether TRCEVENTCTL0R.EVENT1.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT0_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT0_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT0.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT0.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT0.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

EVENT0_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[0] == 1, then Event element 0 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT0.TYPE controls whether TRCEVENTCTL0R.EVENT0.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRCEVENTCTL0R

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCEVENTCTL0R
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1000	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR4().NUMRSPAIR !=
     ↪'0000') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEVENTCTL0R();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEVENTCTL0R();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEVENTCTL0R();
    end;
end;
```

```
MSR TRCEVENTCTL0R, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1000	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR4().NUMRSPAIR !=
     ↪'0000') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEVENTCTL0R() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEVENTCTL0R() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEVENTCTL0R() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.29 TRCEVENTCTL1R, Trace Event Control 1 Register

The TRCEVENTCTL1R characteristics are:

Purpose

Controls the behavior of the ETEEvents that TRCEVENTCTL0R selects.

Configuration

AArch64 System register TRCEVENTCTL1R bits [31:0] are architecturally mapped to External register TRCEVENTCTL1R[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCEVENTCTL1R are UNDEFINED.

Attributes

TRCEVENTCTL1R is a 64-bit register.

Field descriptions

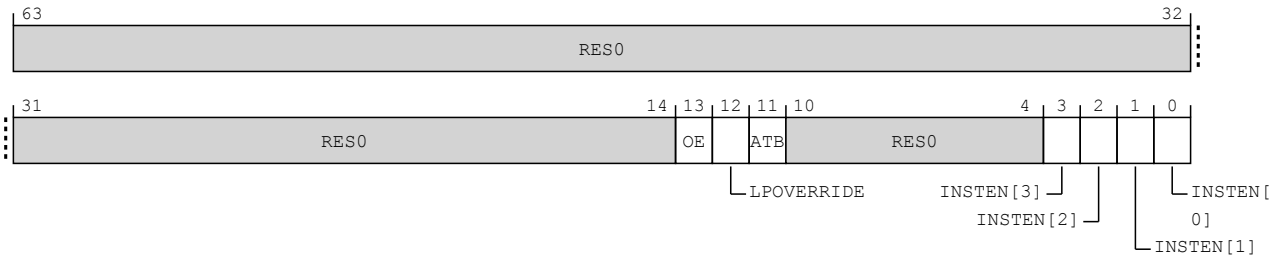

Bits [63:14]

Reserved, RES0.

OE, bit [13]

When TRCIDR5.OE == '1':

ETE Trace Output Enable control.

OE	Meaning
0b0	Trace output to any IMPLEMENTATION DEFINED trace output interface is disabled.
0b1	Trace output to any IMPLEMENTATION DEFINED trace output interface is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

LPOVERRIDE, bit [12]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR5.LPOVERRIDE == '1':

Low-power Override Mode select.

LPOVERRIDE	Meaning
`0b0`	Trace unit Low-power Override Mode is not enabled. That is, the trace unit is permitted to enter low-power state.
`0b1`	Trace unit Low-power Override Mode is enabled. That is, entry to a low-power state does not affect the trace unit resources or trace generation.

Otherwise:

Reserved, RES0.

ATB, bit [11]

When TRCIDR5.ATBTRIG == '1':

AMBA Trace Bus (ATB) trigger enable.

If a CoreSight ATB interface is implemented then when ETEEvent 0 occurs the trace unit sets:

- ATID == `0x7D`.
- ATDATA to the value of TRCTRACEIDR.

If the width of ATDATA is greater than the width of TRCTRACEIDR.TRACEID then the trace unit zeros the upper ATDATA bits.

If ETEEvent 0 is programmed to occur based on program execution, such as an Address Comparator, the ATB trigger might not be inserted into the ATB stream at the same time as any trace generated by that program execution is output by the trace unit. Typically, the generated trace might be buffered in a trace unit which means that the ATB trigger would be output before the associated trace is output.

If ETEEvent 0 is asserted multiple times in close succession, the trace unit is required to generate an ATB trigger for the first assertion, but might ignore one or more of the subsequent assertions. Arm recommends that the window in which ETEEvent 0 is ignored is limited only by the time taken to output an ATB trigger.

ATB	Meaning
`0b0`	ATB trigger is disabled.
`0b1`	ATB trigger is enabled.

Otherwise:

Reserved, RES0.

Bits [10:4]

Reserved, RES0.

INSTEN[<m>] , bits [m], for m = 3 to 0

Event element control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INSTEN[<m>]	Meaning
0b0	The trace unit does not generate an Event element <m>.
0b1	The trace unit generates an Event element <m> when ETEEvent <m> occurs.

Accessing this field has the following behavior:

- When TRCIDR4.NUMRSPAIR == ‘0000’, access to this field is RES0.
- Access to this field is RES0 if all of the following are true:
  - TRCIDR4.NUMRSPAIR != ‘0000’
  - m > UInt(TRCIDR0.NUMEVENT)
- Otherwise, access to this field is RW.

Accessing TRCEVENTCTL1R

Must be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCEVENTCTL1R
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1001	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEVENTCTL1R();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEVENTCTL1R();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEVENTCTL1R();
    end;
end;
```

```
MSR TRCEVENTCTL1R, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1001	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEVENTCTL1R() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEVENTCTL1R() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEVENTCTL1R() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.30 TRCEXTINSELR<n>, Trace External Input Select Register <n>, n = 0 - 3

The TRCEXTINSELR<n> characteristics are:

Purpose

Use this to set, or read, which External Inputs are resources to the trace unit.

The name TRCEXTINSELR is an alias of TRCEXTINSELR0.

Configuration

AArch64 System register TRCEXTINSELR<n> bits [31:0] are architecturally mapped to External register TRCEXTINSELR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR5.NUMEXTINSEL) > n. Otherwise, direct accesses to TRCEXTINSELR<n> are UNDEFINED.

Attributes

TRCEXTINSELR<n> is a 64-bit register.

Field descriptions

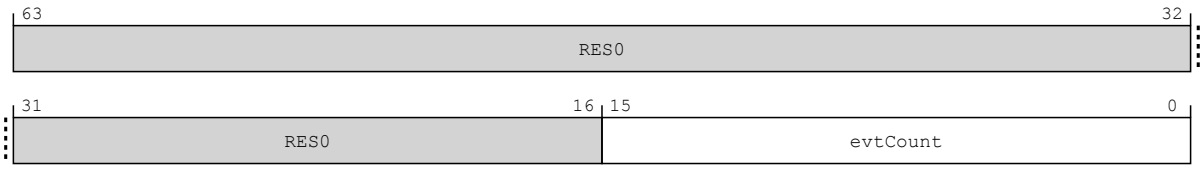

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

PMU event to select.

The event number as defined by the Arm ARM.

Software must program this field with a PMU event that is supported by the PE being programmed.

If evtCount is programmed to a PMU event that is reserved or not supported by the PE, the behavior depends on the PMU event type:

- For the range `0x0000` to `0x003F`, then the PMU event is not active, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED PMU events, it is UNPREDICTABLE what PMU event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

UNPREDICTABLE means the PMU event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include a PMU event from a set of common IMPLEMENTATION DEFINED PMU events, then no PMU event is counted and the value read back on evtCount is the value written.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCEXTINSELR<n>

Must be programmed if any of the following is true: TRCRSCTLR<a>.GROUP == `0b0000` and TRCRSCTLR<a>.EXTIN[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCEXTINSELR<m> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b10:m[1:0]	0b100

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_EXTERNAL_INPUT_SELECTOR_RESOURCES then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEXTINSELR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEXTINSELR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCEXTINSELR(m);
    end;
end;
```

```
MSR TRCEXTINSELR<m>, <Xt> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b10:m[1:0]	0b100

```
let m:integer = UInt(CRm[1:0]);

if m >= NUM_TRACE_EXTERNAL_INPUT_SELECTOR_RESOURCES then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEXTINSELR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEXTINSELR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCEXTINSELR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.31 TRCIDR0, Trace ID Register 0

The TRCIDR0 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR0 bits [31:0] are architecturally mapped to External register TRCIDR0[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR0 are UNDEFINED.

Attributes

TRCIDR0 is a 64-bit register.

Field descriptions

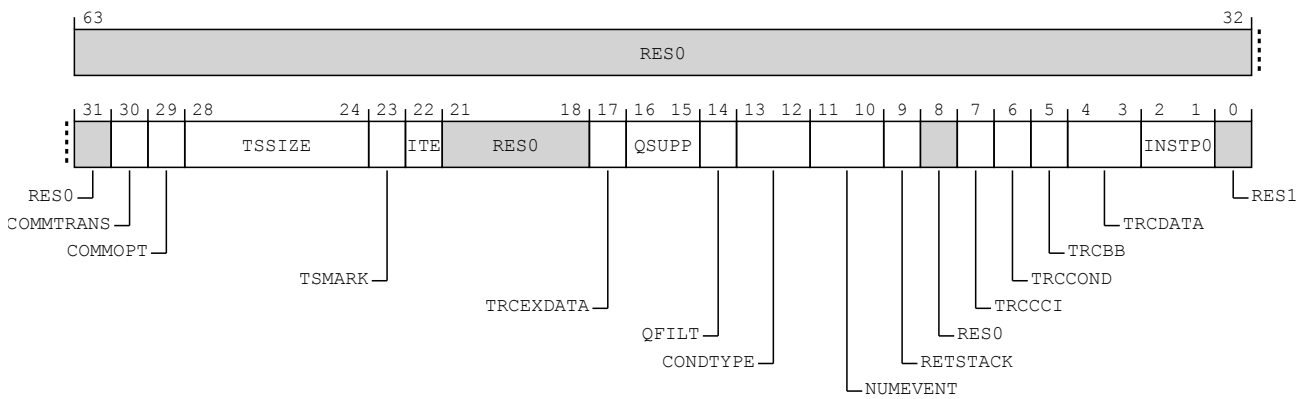

Bits [63:31]

Reserved, RES0.

COMMTRANS, bit [30]

Transaction Start element behavior.

The value of this field is an IMPLEMENTATION DEFINED choice of:

COMMTRANS	Meaning
0b0	Transaction Start elements are P0 elements.
0b1	Transaction Start elements are not P0 elements.

Access to this field is RO.

COMMOPT, bit [29]

Indicates the contents and encodings of Cycle count packets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMMOPT	Meaning
0b0	Commit mode 0.
0b1	Commit mode 1.

The Commit mode defines the contents and encodings of Cycle Count packets, in particular how Commit elements are indicated by these packets. See the descriptions of these packets for more details.

Accessing this field has the following behavior:

- Access to this field is RAO/WI if all of the following are true:
  - TRCIDR0.TRCCCI == '1'
  - UInt(TRCIDR8.MAXSPEC) == 0x0
- When TRCIDR0.TRCCCI == '0', access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

TSSIZE, bits [28:24]

Indicates that the trace unit implements Global timestamping and the size of the timestamp value.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TSSIZE	Meaning
0b00000	Global timestamping not implemented.
0b01000	Global timestamping implemented with a 64-bit timestamp value.

All other values are reserved.

This field reads as 0b01000.

Access to this field is RO.

TSMARK, bit [23]

Indicates whether Timestamp Marker elements are generated.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TSMARK	Meaning
0b0	Timestamp Marker elements are not generated.
0b1	Timestamp Marker elements are generated.

Access to this field is RO.

ITE, bit [22]

Indicates whether Instrumentation Trace is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITE	Meaning
0b0	Instrumentation Trace not implemented.
0b1	Instrumentation Trace implemented.

This field has the value 1 if FEAT_ITE is implemented.

Access to this field is RO.

Bits [21:18]

Reserved, RES0.

TRCEXDATA, bit [17]

When TRCIDR0.TRCDATA != '00':

Indicates if the trace unit implements tracing of data transfers for exceptions and exception returns. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCEXDATA	Meaning
0b0	Tracing of data transfers for exceptions and exception returns not implemented.
0b1	Tracing of data transfers for exceptions and exception returns implemented.

Access to this field is RO.

Otherwise:

Reserved, RES0.

QSUPP, bits [16:15]

Indicates that the trace unit implements Q element support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

QSUPP	Meaning
0b00	Q element support is not implemented.
0b01	Q element support is implemented, and only supports Q elements with instruction counts.
0b10	Q element support is implemented, and only supports Q elements without instruction counts.
0b11	Q element support is implemented, and supports: • Q elements with instruction counts. • Q elements without instruction counts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

QFILT, bit [14]

Indicates if the trace unit implements Q element filtering.

The value of this field is an IMPLEMENTATION DEFINED choice of:

QFILT	Meaning
0b0	Q element filtering is not implemented.
0b1	Q element filtering is implemented.

If TRCIDR0.QSUPP == 0b00 then this field is 0.

Access to this field is RO.

CONDTYPE, bits [13:12]

When TRCIDR0.TRCCOND == '1':

Indicates how conditional instructions are traced. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CONDTYPE	Meaning
0b00	Conditional instructions are traced with an indication of whether they pass or fail their condition code check.
0b01	Conditional instructions are traced with an indication of the APSR condition flags.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NUMEVENT, bits [11:10]

When TRCIDR4.NUMRSPAIR == '0000':

Indicates the number of ETEEvents implemented.

NUMEVENT	Meaning
0b00	The trace unit supports 0 ETEEvents.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR4.NUMRSPAIR != '0000':

Indicates the number of ETEEvents implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMEVENT	Meaning
`0b00`	The trace unit supports 1 ETEEvent.
`0b01`	The trace unit supports 2 ETEEvents.
`0b10`	The trace unit supports 3 ETEEvents.
`0b11`	The trace unit supports 4 ETEEvents.

Access to this field is RO.

Otherwise:

Reserved, RES0.

RETSTACK, bit [9]

Indicates if the trace unit supports the return stack.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RETSTACK	Meaning
`0b0`	Return stack not implemented.
`0b1`	Return stack implemented.

Access to this field is RO.

Bit [8]

Reserved, RES0.

TRCCCI, bit [7]

Indicates if the trace unit implements cycle counting.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCCCI	Meaning
`0b0`	Cycle counting not implemented.
`0b1`	Cycle counting implemented.

This field reads as 1.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCCOND, bit [6]

Indicates if the trace unit implements conditional instruction tracing. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCCOND	Meaning
`0b0`	Conditional instruction tracing not implemented.
`0b1`	Conditional instruction tracing implemented.

This field reads as 0.

Access to this field is RO.

TRCBB, bit [5]

Indicates if the trace unit implements branch broadcasting.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCBB	Meaning
`0b0`	Branch broadcasting not implemented.
`0b1`	Branch broadcasting implemented.

This field reads as 1.

Access to this field is RO.

TRCDATA, bits [4:3]

Indicates if the trace unit implements data tracing. Data tracing is not implemented in ETE and this field is reserved for other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCDATA	Meaning
`0b00`	Data tracing not implemented.
`0b11`	Data tracing implemented.

All other values are reserved.

This field reads as `0b00`.

Access to this field is RO.

INSTP0, bits [2:1]

Indicates if load and store instructions are P0 instructions. Load and store instructions as P0 instructions is not implemented in ETE and this field is reserved for other trace architectures.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

INSTP0	Meaning
0b00	Load and store instructions are not P0 instructions.
0b11	Load and store instructions are P0 instructions.

All other values are reserved.

When FEAT_ETE is implemented, the only permitted value is 0b00.

Access to this field is RO.

Bit [0]

Reserved, RES1.

Accessing TRCIDR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR0
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1000	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR0();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR0();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.32 TRCIDR1, Trace ID Register 1

The TRCIDR1 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR1 bits [31:0] are architecturally mapped to External register TRCIDR1[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR1 are UNDEFINED.

Attributes

TRCIDR1 is a 64-bit register.

Field descriptions

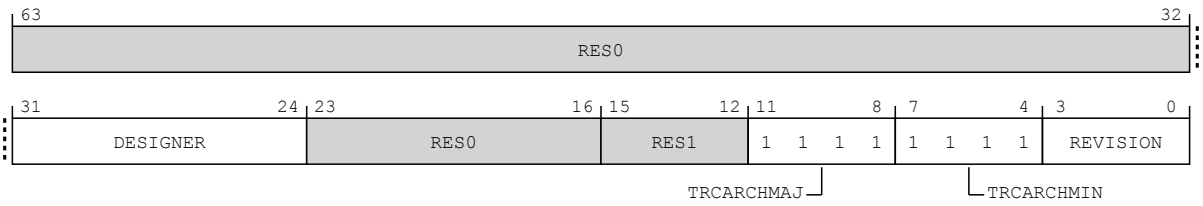

Bits [63:32]

Reserved, RES0.

DESIGNER, bits [31:24]

Indicates which company designed the trace unit. The permitted values of this field are the same as MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RES1.

TRCARCHMAJ, bits [11:8]

Major architecture version.

TRCARCHMAJ	Meaning
`0b1111`	If both TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN == `0xF` then refer to TRCDEVARCH.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

This field reads as `0b1111`.

Access to this field is RO.

TRCARCHMIN, bits [7:4]

Minor architecture version.

TRCARCHMIN	Meaning
`0b1111`	If both TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN == `0xF` then refer to TRCDEVARCH.

All other values are reserved.

This field reads as `0b1111`.

Access to this field is RO.

REVISION, bits [3:0]

Implementation revision.

Returns an IMPLEMENTATION DEFINED value that identifies the revision of the trace unit.

Arm deprecates any use of this field and recommends that implementations set this field to zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCIDR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR1
```

op0	op1	CRn	CRm	op2
`0b10`	`0b001`	`0b0000`	`0b1001`	`0b111`

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR1();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.33 TRCIDR10, Trace ID Register 10

The TRCIDR10 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR10 bits [31:0] are architecturally mapped to External register TRCIDR10[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR10 are UNDEFINED.

Attributes

TRCIDR10 is a 64-bit register.

Field descriptions

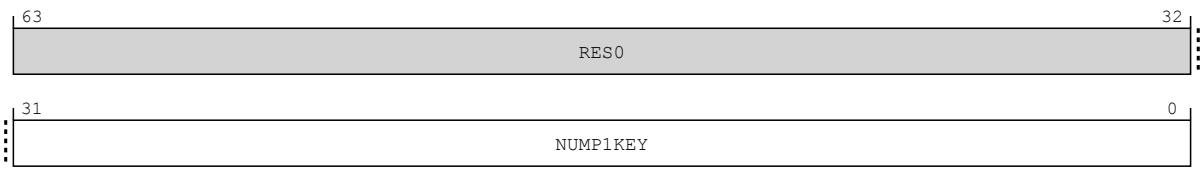

Bits [63:32]

Reserved, RES0.

NUMP1KEY, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR10

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR10
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0010	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR10();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR10();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR10();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.34 TRCIDR11, Trace ID Register 11

The TRCIDR11 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR11 bits [31:0] are architecturally mapped to External register TRCIDR11[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR11 are UNDEFINED.

Attributes

TRCIDR11 is a 64-bit register.

Field descriptions

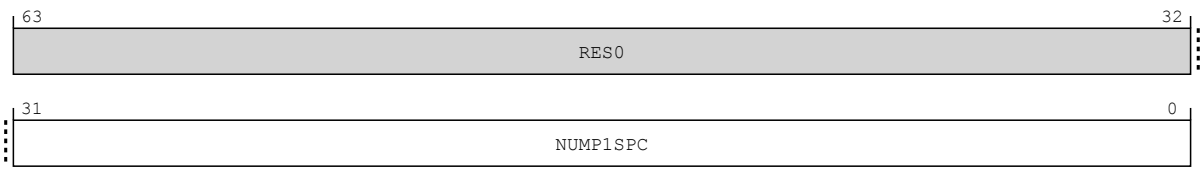

Bits [63:32]

Reserved, RES0.

NUMP1SPC, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of special P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR11

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR11
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0011	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR11();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR11();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR11();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.35 TRCIDR12, Trace ID Register 12

The TRCIDR12 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR12 bits [31:0] are architecturally mapped to External register TRCIDR12[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR12 are UNDEFINED.

Attributes

TRCIDR12 is a 64-bit register.

Field descriptions

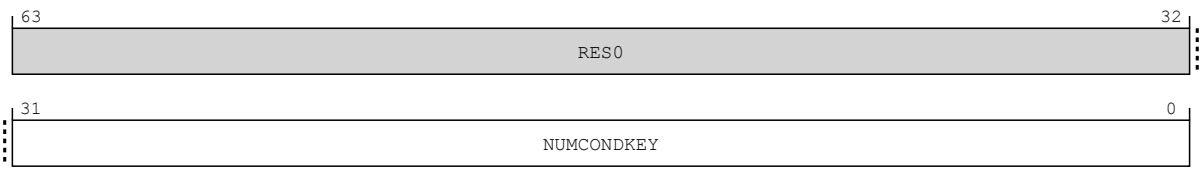

Bits [63:32]

Reserved, RES0.

NUMCONDKEY, bits [31:0]

When TRCIDR0.TRCCOND == '1':

Indicates the number of conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR12

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR12
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0100	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR12();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR12();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR12();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.36 TRCIDR13, Trace ID Register 13

The TRCIDR13 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR13 bits [31:0] are architecturally mapped to External register TRCIDR13[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR13 are UNDEFINED.

Attributes

TRCIDR13 is a 64-bit register.

Field descriptions

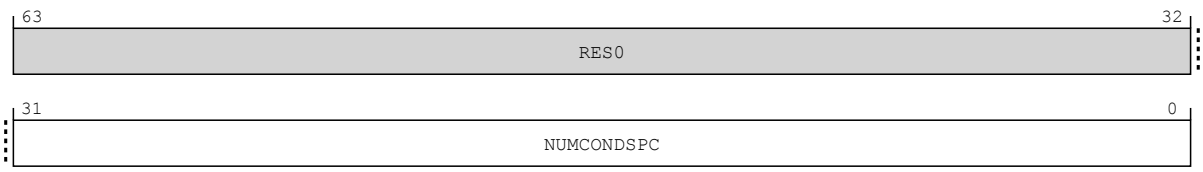

Bits [63:32]

Reserved, RES0.

NUMCONDSPC, bits [31:0]

When TRCIDR0.TRCCOND == '1':

Indicates the number of special conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR13

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR13
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0101	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR13();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR13();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR13();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.37 TRCIDR2, Trace ID Register 2

The TRCIDR2 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR2 bits [31:0] are architecturally mapped to External register TRCIDR2[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR2 are UNDEFINED.

Attributes

TRCIDR2 is a 64-bit register.

Field descriptions

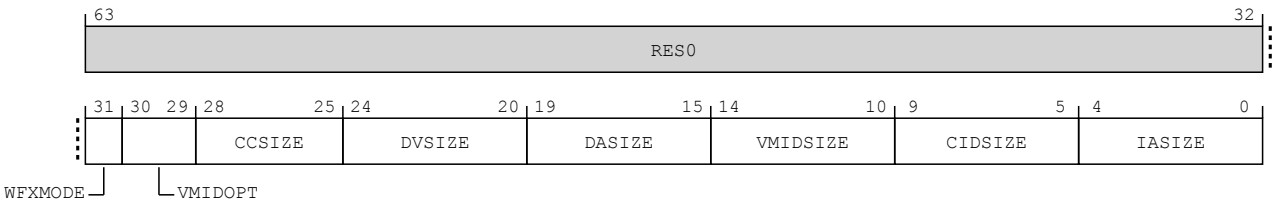

Bits [63:32]

Reserved, RES0.

WFXMODE, bit [31]

Indicates whether `WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions:

The value of this field is an IMPLEMENTATION DEFINED choice of:

WFXMODE	Meaning
0b0	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are not classified as P0 instructions.
0b1	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions.

Access to this field is RO.

VMIDOPT, bits [30:29]

Indicates the options for Virtual context identifier selection.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMIDOPT	Meaning
0b00	Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES0.
0b01	Virtual context identifier selection supported. TRCCONFIGR.VMIDOPT is implemented.
0b10	Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES1.

All other values are reserved.

If TRCIDR2.VMIDSIZE == 0b00000 then this field is 0b00.

If TRCIDR2.VMIDSIZE != 0b00000 then this field is 0b10.

Access to this field is RO.

CCSIZE, bits [28:25]

When TRCIDR0.TRCCCI == '1':

Indicates the size of the cycle counter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCSIZE	Meaning
0b0000	The cycle counter is 12 bits in length.
0b0001	The cycle counter is 13 bits in length.
0b0010	The cycle counter is 14 bits in length.
0b0011	The cycle counter is 15 bits in length.
0b0100	The cycle counter is 16 bits in length.
0b0101	The cycle counter is 17 bits in length.
0b0110	The cycle counter is 18 bits in length.
0b0111	The cycle counter is 19 bits in length.
0b1000	The cycle counter is 20 bits in length.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

DVSIZE, bits [24:20]

When TRCIDR0.TRCDATA != '00':

Indicates the data value size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DVSIZE	Meaning
0b00000	Data value tracing not implemented.
0b00100	Data value tracing has a maximum of 32-bit data values.
0b01000	Data value tracing has a maximum of 64-bit data values.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

DASIZE, bits [19:15]

When TRCIDR0.TRCDATA != '00':

Indicates the data address size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DASIZE	Meaning
0b00000	Data address tracing not implemented.
0b00100	Data address tracing has a maximum of 32-bit data addresses.
0b01000	Data address tracing has a maximum of 64-bit data addresses.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

VMIDSIZE, bits [14:10]

Indicates the trace unit Virtual context identifier size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDSIZE	Meaning
0b00000	Virtual context identifier tracing is not supported.
0b00001	8-bit Virtual context identifier size.
0b00010	16-bit Virtual context identifier size.
0b00100	32-bit Virtual context identifier size.

All other values are reserved.

If the PE does not implement EL2 then this field is 0b00000.

If the PE implements EL2 then this field is 0b00100.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

CIDSIZE, bits [9:5]

Indicates the Context identifier size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CIDSIZE	Meaning
0b00000	Context identifier tracing is not supported.
0b00100	32-bit Context identifier size.

All other values are reserved.

This field reads as 0b00100.

Access to this field is RO.

IASIZE, bits [4:0]

Virtual instruction address size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IASIZE	Meaning
0b00100	Maximum of 32-bit instruction address size.
0b01000	Maximum of 64-bit instruction address size.

All other values are reserved.

This field reads as 0b01000.

Access to this field is RO.

Accessing TRCIDR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR2
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1010	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR2();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR2();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR2();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.38 TRCIDR3, Trace ID Register 3

The TRCIDR3 characteristics are:

Purpose

Returns the base architecture of the trace unit.

Configuration

AArch64 System register TRCIDR3 bits [31:0] are architecturally mapped to External register TRCIDR3[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR3 are UNDEFINED.

Attributes

TRCIDR3 is a 64-bit register.

Field descriptions

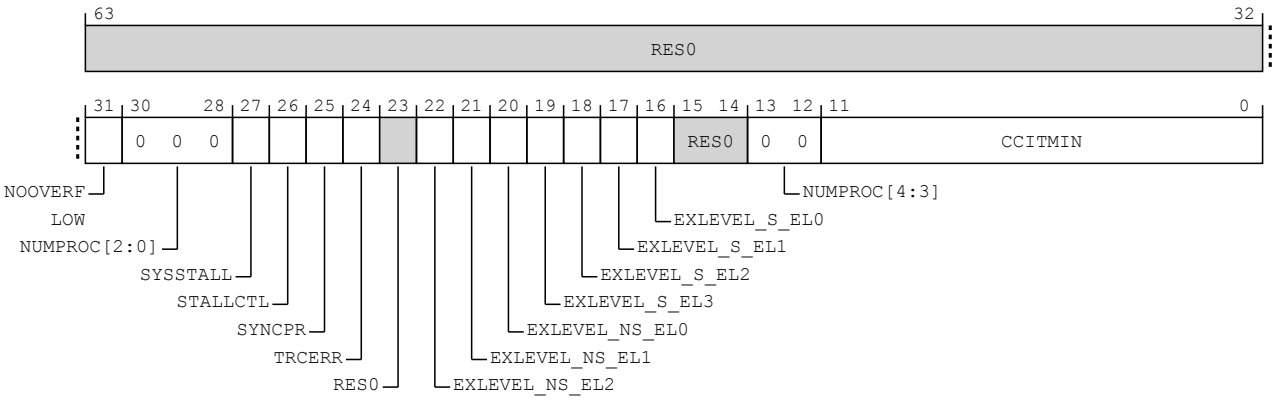

Bits [63:32]

Reserved, RES0.

NOOVERFLOW, bit [31]

Indicates if overflow prevention is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NOOVERFLOW	Meaning
0b0	Overflow prevention is not implemented.
0b1	Overflow prevention is implemented.

If TRCIDR3.STALLCTL == 0 then this field is 0.

Access to this field is RO.

NUMPROC, bits [13:12, 30:28]

Indicates the number of PEs available for tracing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMPROC	Meaning
0b00000	The trace unit can trace one PE.

This field reads as 0b00000.

Access to this field is RO.

SYSSTALL, bit [27]

Indicates if stalling of the PE is permitted.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYSSTALL	Meaning
0b0	Stalling of the PE is not permitted.
0b1	Stalling of the PE is permitted.

The value of this field might be dynamic and change based on system conditions.

If TRCIDR3.STALLCTL == 0 then this field is 0.

Access to this field is RO.

STALLCTL, bit [26]

Indicates if trace unit implements stalling of the PE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

STALLCTL	Meaning
0b0	Stalling of the PE is not implemented.
0b1	Stalling of the PE is implemented.

Access to this field is RO.

SYNCPR, bit [25]

Indicates if an implementation has a fixed synchronization period.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYNCPR	Meaning
0b0	TRCSYNCPR is read/write so software can change the synchronization period.
0b1	TRCSYNCPR is read-only so the synchronization period is fixed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field reads as 0.

Access to this field is RO.

TRCERR, bit [24]

Indicates forced tracing of System Error exceptions is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCERR	Meaning
0b0	Forced tracing of System Error exceptions is not implemented.
0b1	Forced tracing of System Error exceptions is implemented.

This field reads as 1.

Access to this field is RO.

Bit [23]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]

Indicates if Non-secure EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL2	Meaning
0b0	Non-secure EL2 is not implemented.
0b1	Non-secure EL2 is implemented.

Access to this field is RO.

EXLEVEL_NS_EL1, bit [21]

Indicates if Non-secure EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL1	Meaning
0b0	Non-secure EL1 is not implemented.
0b1	Non-secure EL1 is implemented.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL0, bit [20]

Indicates if Non-secure EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL0	Meaning
0b0	Non-secure EL0 is not implemented.
0b1	Non-secure EL0 is implemented.

Access to this field is RO.

EXLEVEL_S_EL3, bit [19]

Indicates if EL3 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL3	Meaning
0b0	EL3 is not implemented.
0b1	EL3 is implemented.

Access to this field is RO.

EXLEVEL_S_EL2, bit [18]

Indicates if Secure EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL2	Meaning
0b0	Secure EL2 is not implemented.
0b1	Secure EL2 is implemented.

Access to this field is RO.

EXLEVEL_S_EL1, bit [17]

Indicates if Secure EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_S_EL1	Meaning
0b0	Secure EL1 is not implemented.
0b1	Secure EL1 is implemented.

Access to this field is RO.

EXLEVEL_S_EL0, bit [16]

Indicates if Secure EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL0	Meaning
0b0	Secure EL0 is not implemented.
0b1	Secure EL0 is implemented.

Access to this field is RO.

Bits [15:14]

Reserved, RES0.

CCITMIN, bits [11:0]

When TRCIDR0.TRCCCI == '0':

Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

Reads as 0x000

Access to this field is RO.

When TRCIDR0.TRCCCI == '1':

Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCITMIN	Meaning
0x001..0xFFF	The minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

The minimum value of this field is 0x001.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TRCIDR3
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1011	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR3();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR3();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR3();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.39 TRCIDR4, Trace ID Register 4

The TRCIDR4 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR4 bits [31:0] are architecturally mapped to External register TRCIDR4[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR4 are UNDEFINED.

Attributes

TRCIDR4 is a 64-bit register.

Field descriptions

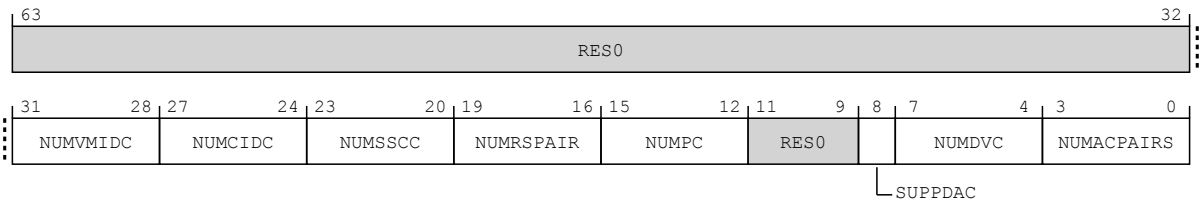

Bits [63:32]

Reserved, RES0.

NUMVMIDC, bits [31:28]

Indicates the number of Virtual Context Identifier Comparators that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMVMIDC	Meaning
0b0000..0b1000	The number of Virtual Context Identifier Comparators in this implementation.

All other values are reserved.

If the PE does not implement EL2 then this field is 0b0000.

Access to this field is RO.

NUMCIDC, bits [27:24]

Indicates the number of Context Identifier Comparators that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMCIDC	Meaning
0b0000..0b1000	The number of Context Identifier Comparators in this implementation.

All other values are reserved.

Access to this field is RO.

NUMSSCC, bits [23:20]

Indicates the number of Single-shot Comparator Controls that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMSSCC	Meaning
0b0000..0b1000	The number of Single-shot Comparator Controls in this implementation.

All other values are reserved.

Access to this field is RO.

NUMRSPAIR, bits [19:16]

Indicates the number of resource selector pairs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMRSPAIR	Meaning
0b0000	This implementation has zero resource selector pairs.
0b0001..0b1111	The number of resource selector pairs in this implementation, minus one.

All other values are reserved.

Access to this field is RO.

NUMPC, bits [15:12]

Indicates the number of PE Comparator Inputs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMPC	Meaning
0b0000..0b1000	The number of PE Comparator Inputs in this implementation.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [11:9]

Reserved, RES0.

SUPPDAC, bit [8]

When TRCIDR4.NUMACPAIRS != '0000':

Indicates whether data address comparisons are implemented. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SUPPDAC	Meaning
`0b0`	Data address comparisons not implemented.
`0b1`	Data address comparisons implemented.

This field reads as 0.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NUMDVC, bits [7:4]

Indicates the number of data value comparators. Data value comparators are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMDVC	Meaning
`0b0000..0b1000`	The number of data value comparators in this implementation.

All other values are reserved.

This field reads as `0b0000`.

Access to this field is RO.

NUMACPAIRS, bits [3:0]

Indicates the number of Address Comparator pairs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMACPAIRS	Meaning
`0b0000..0b1000`	The number of Address Comparator pairs in this implementation.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCIDR4

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR4
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1100	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR4();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR4();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR4();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.40 TRCIDR5, Trace ID Register 5

The TRCIDR5 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR5 bits [31:0] are architecturally mapped to External register TRCIDR5[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR5 are UNDEFINED.

Attributes

TRCIDR5 is a 64-bit register.

Field descriptions

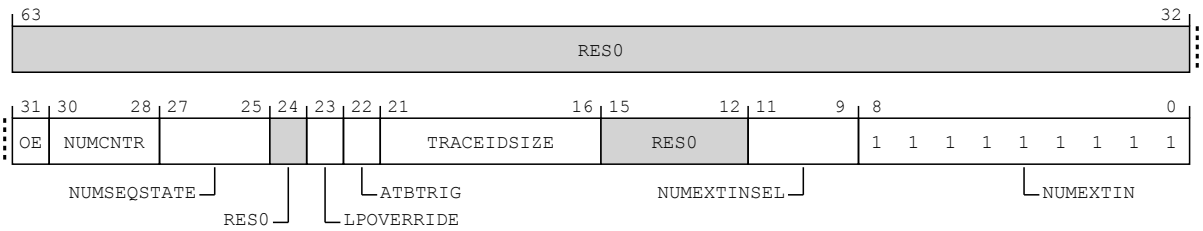

Bits [63:32]

Reserved, RES0.

OE, bit [31]

Indicates support for the ETE Trace Output Enable.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OE	Meaning
0b0	ETE Trace Output Enable is not implemented.
0b1	ETE Trace Output Enable is implemented.

When FEAT_ETEv1p3 is implemented and when any IMPLEMENTATION DEFINED trace output interface is implemented, this field is 1.

Access to this field is RO.

NUMCNTR, bits [30:28]

Indicates the number of Counters that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMCNTR	Meaning
0b000..0b100	The number of Counters implemented.

All other values are reserved.

If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0b000.

Access to this field is RO.

NUMSEQSTATE, bits [27:25]

Indicates if the Sequencer is implemented and the number of Sequencer states that are implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMSEQSTATE	Meaning
0b000	The Sequencer is not implemented.
0b100	Four Sequencer states are implemented.

All other values are reserved.

If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0b000.

Access to this field is RO.

Bit [24]

Reserved, RES0.

LPOVERRIDE, bit [23]

Indicates support for Low-power Override Mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LPOVERRIDE	Meaning
0b0	The trace unit does not support Low-power Override Mode.
0b1	The trace unit supports Low-power Override Mode.

Access to this field is RO.

ATBTRIG, bit [22]

Indicates if the implementation can support ATB triggers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ATBTRIG	Meaning
0b0	The implementation does not support ATB triggers.
0b1	The implementation supports ATB triggers.

If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0.

Access to this field is RO.

TRACEIDSIZE, bits [21:16]

Indicates the trace ID width.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRACEIDSIZE	Meaning
0b000000	The external trace interface is not implemented.
0b000111	The implementation supports a 7-bit trace ID.

All other values are reserved.

Note

AMBA ATB requires a 7-bit trace ID width.

Access to this field is RO.

Bits [15:12]

Reserved, RES0.

NUMEXTINSEL, bits [11:9]

Indicates how many External Input Selector resources are implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMEXTINSEL	Meaning
0b000..0b100	The number of External Input Selector resources implemented.

All other values are reserved.

Access to this field is RO.

NUMEXTIN, bits [8:0]

Indicates how many External Inputs are implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMEXTIN	Meaning
0b111111111	Unified PMU event selection.

All other values are reserved.

Access to this field is RO.

Accessing TRCIDR5

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR5
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1101	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR5();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR5();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR5();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.41 TRCIDR6, Trace ID Register 6

The TRCIDR6 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR6 bits [31:0] are architecturally mapped to External register TRCIDR6[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR6 are UNDEFINED.

Attributes

TRCIDR6 is a 64-bit register.

Field descriptions

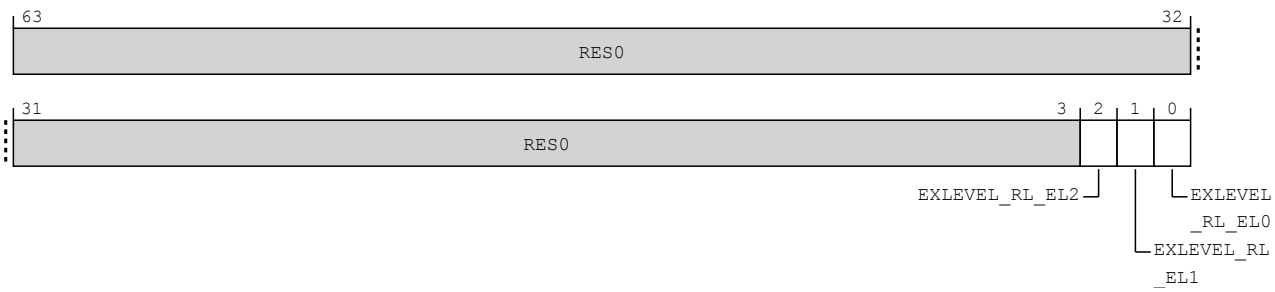

Bits [63:3]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [2]

Indicates if Realm EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_RL_EL2	Meaning
0b0	Realm EL2 is not implemented.
0b1	Realm EL2 is implemented.

Access to this field is RO.

EXLEVEL_RL_EL1, bit [1]

Indicates if Realm EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_RL_EL1	Meaning
0b0	Realm EL1 is not implemented.
0b1	Realm EL1 is implemented.

Access to this field is RO.

EXLEVEL_RL_EL0, bit [0]

Indicates if Realm EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_RL_EL0	Meaning
0b0	Realm EL0 is not implemented.
0b1	Realm EL0 is implemented.

Access to this field is RO.

Accessing TRCIDR6

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR6
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1110	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR6();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR6();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR6();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.42 TRCIDR7, Trace ID Register 7

The TRCIDR7 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR7 bits [31:0] are architecturally mapped to External register TRCIDR7[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR7 are UNDEFINED.

Attributes

TRCIDR7 is a 64-bit register.

Field descriptions

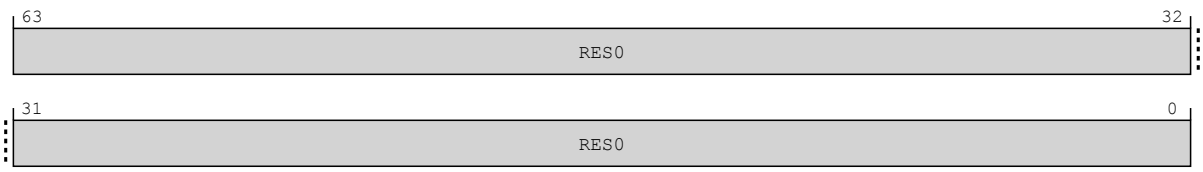

Bits [63:0]

Reserved, RES0.

Accessing TRCIDR7

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1111	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR7();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR7();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR7();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.43 TRCIDR8, Trace ID Register 8

The TRCIDR8 characteristics are:

Purpose

Returns the maximum speculation depth of the instruction trace element stream.

Configuration

AArch64 System register TRCIDR8 bits [31:0] are architecturally mapped to External register TRCIDR8[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR8 are UNDEFINED.

Attributes

TRCIDR8 is a 64-bit register.

Field descriptions

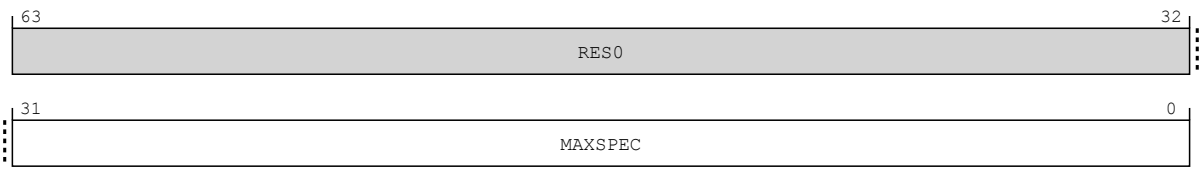

Bits [63:32]

Reserved, RES0.

MAXSPEC, bits [31:0]

Indicates the maximum speculation depth of the instruction trace element stream. This is the maximum number of P0 elements in the trace element stream that can be speculative at any time.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCIDR8

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR8
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR8();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR8();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR8();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.44 TRCIDR9, Trace ID Register 9

The TRCIDR9 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

AArch64 System register TRCIDR9 bits [31:0] are architecturally mapped to External register TRCIDR9[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIDR9 are UNDEFINED.

Attributes

TRCIDR9 is a 64-bit register.

Field descriptions

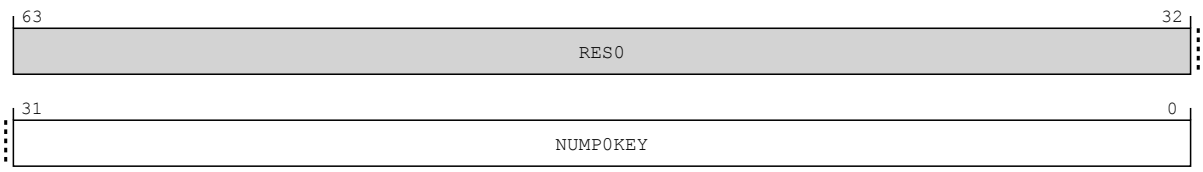

Bits [63:32]

Reserved, RES0.

NUMP0KEY, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of P0 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR9

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR9
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCID == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR9();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR9();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIDR9();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.45 TRCIMSPEC0, Trace IMP DEF Register 0

The TRCIMSPEC0 characteristics are:

Purpose

TRCIMSPEC0 shows the presence of any IMPLEMENTATION DEFINED features, and provides an interface to enable the features that are provided.

Configuration

AArch64 System register TRCIMSPEC0 bits [31:0] are architecturally mapped to External register TRCIMSPEC0[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIMSPEC0 are UNDEFINED.

Attributes

TRCIMSPEC0 is a 64-bit register.

Field descriptions

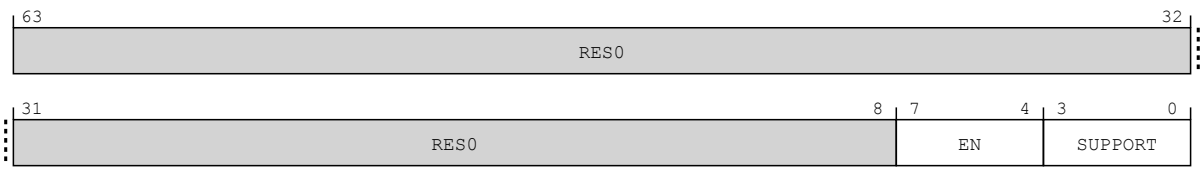

Bits [63:8]

Reserved, RES0.

EN, bits [7:4]

When TRCIMSPEC0.SUPPORT != '0000':

Enable. Controls whether the IMPLEMENTATION DEFINED features are enabled.

EN	Meaning
0b0000	The IMPLEMENTATION DEFINED features are not enabled. The trace unit must behave as if the IMPLEMENTATION DEFINED features are not supported.
0b0001	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0010	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0011	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0100	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0101	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0110	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0111	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1000	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1001	The trace unit behavior is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EN	Meaning
0b1010	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1011	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1100	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1101	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1110	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1111	The trace unit behavior is IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0000'.

Otherwise:

Reserved, RES0.

SUPPORT, bits [3:0]

Indicates whether the implementation supports IMPLEMENTATION DEFINED features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SUPPORT	Meaning
0b0000	No IMPLEMENTATION DEFINED features are supported.
0b0001	IMPLEMENTATION DEFINED features are supported.
0b0010	IMPLEMENTATION DEFINED features are supported.
0b0011	IMPLEMENTATION DEFINED features are supported.
0b0100	IMPLEMENTATION DEFINED features are supported.
0b0101	IMPLEMENTATION DEFINED features are supported.
0b0110	IMPLEMENTATION DEFINED features are supported.
0b0111	IMPLEMENTATION DEFINED features are supported.
0b1000	IMPLEMENTATION DEFINED features are supported.
0b1001	IMPLEMENTATION DEFINED features are supported.
0b1010	IMPLEMENTATION DEFINED features are supported.
0b1011	IMPLEMENTATION DEFINED features are supported.
0b1100	IMPLEMENTATION DEFINED features are supported.
0b1101	IMPLEMENTATION DEFINED features are supported.
0b1110	IMPLEMENTATION DEFINED features are supported.
0b1111	IMPLEMENTATION DEFINED features are supported.

Use of nonzero values requires written permission from Arm.

Access to this field is RO.

Accessing TRCIMSPEC0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TRCIMSPEC0
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCIMSPECn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIMSPEC0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIMSPEC0();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIMSPEC0();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TRCIMSPEC0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCIMSPECn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCIMSPEC0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCIMSPEC0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCIMSPEC0() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.46 TRCIMSPEC<n>, Trace IMP DEF Register <n>, n = 1 - 7

The TRCIMSPEC<n> characteristics are:

Purpose

These registers might return information that is specific to an implementation, or enable features specific to an implementation to be programmed. The product Technical Reference Manual describes these registers.

Configuration

AArch64 System register TRCIMSPEC<n> bits [31:0] are architecturally mapped to External register TRCIMSPEC<n>[31:0].

This register is present only when an implementation implements TRCIMSPEC<n>, FEAT_ETE is implemented, and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCIMSPEC<n> are UNDEFINED.

Attributes

TRCIMSPEC<n> is a 64-bit register.

Field descriptions

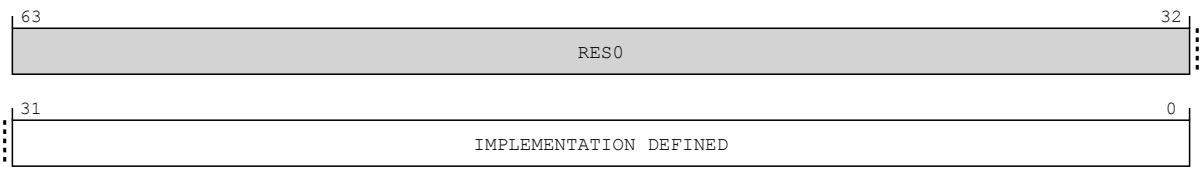

Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

Accessing TRCIMSPEC<n>

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIMSPEC<m> ; Where m = 1-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0:m[2:0]	0b111

```
let m:integer = UInt(CRm[2:0]);

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCIMSPECn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIMSPEC(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIMSPEC(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCIMSPEC(m);
    end;
end;
```

```
MSR TRCIMSPEC<m>, <Xt> ; Where m = 1-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0:m[2:0]	0b111

```
let m:integer = UInt(CRm[2:0]);

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCIMSPECn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCIMSPEC(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCIMSPEC(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCIMSPEC(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.47 TRCITECR_EL1, Instrumentation Trace Control Register (EL1)

The TRCITECR_EL1 characteristics are:

Purpose

Provides EL1 controls for Trace Instrumentation.

Configuration

This register is present only when FEAT_ITE is implemented, System register access to the trace unit registers is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TRCITECR_EL1 are UNDEFINED.

Attributes

TRCITECR_EL1 is a 64-bit register.

Field descriptions

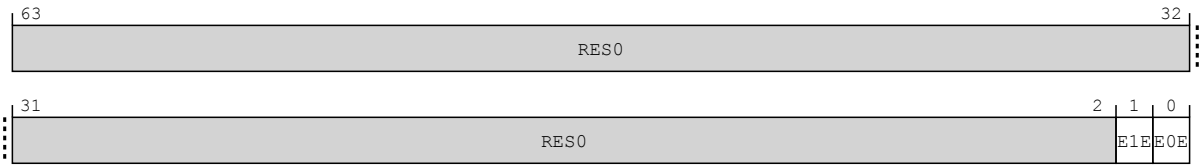

Bits [63:2]

Reserved, RES0.

E1E, bit [1]

EL1 Instrumentation Trace Enable.

E1E	Meaning
0b0	Instrumentation trace prohibited at EL1.
0b1	Instrumentation trace not prohibited at EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

E0E, bit [0]

EL0 Instrumentation Trace Enable.

E0E	Meaning
0b0	Instrumentation trace prohibited at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Instrumentation trace not prohibited at EL0.

This field is ignored by the PE when EL2 is implemented and enabled in the current Security state and the Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing TRCITECR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name TRCITECR_EL1 or TRCITECR_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCITECR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nTRCITECR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x888);
    else
        X{64}(t) = TRCITECR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TRCITECR_EL2();
    else
        X{64}(t) = TRCITECR_EL1();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(t) = TRCITECR_EL1();
end;
```

MSR TRCITECR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nTRCITECR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x888) = X{64}(t);
    else
        TRCITECR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TRCITECR_EL2() = X{64}(t);
    else
        TRCITECR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TRCITECR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TRCITECR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x888);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TRCITECR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = TRCITECR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR TRCITECR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x888) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            TRCITECR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TRCITECR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.48 TRCITECR_EL2, Instrumentation Trace Control Register (EL2)

The TRCITECR_EL2 characteristics are:

Purpose

Provides EL2 controls for Trace Instrumentation.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_ITE is implemented, System register access to the trace unit registers is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to TRCITECR_EL2 are UNDEFINED.

Attributes

TRCITECR_EL2 is a 64-bit register.

Field descriptions

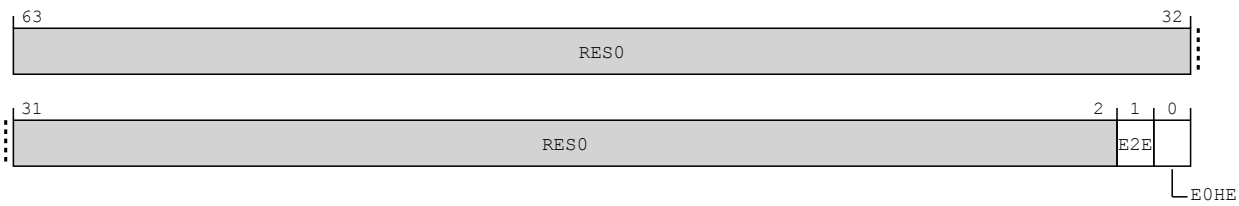

Bits [63:2]

Reserved, RES0.

E2E, bit [1]

EL2 Instrumentation Trace Enable.

E2E	Meaning
0b0	Instrumentation trace prohibited at EL2.
0b1	Instrumentation trace not prohibited at EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

E0HE, bit [0]

EL0 Instrumentation Trace Enable.

E0HE	Meaning
0b0	Instrumentation trace prohibited at EL0 when the Effective value of HCR_EL2.TGE is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E0HE	Meaning
0b1	Instrumentation trace not prohibited at EL0 when the Effective value of HCR_EL2.TGE is 1.

This field is ignored by the PE when any of the following are true:

- The Effective value of HCR_EL2.TGE is 0.
- EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRCITECR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `TRCITECR_EL2` or `TRCITECR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCITECR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TRCITECR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TRCITECR_EL2();
end;
```

```
MSR TRCITECR_EL2, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TRCITECR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TRCITECR_EL2() = X{64}(t);
end;
```

MRS <Xt>, TRCITECR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nTRCITECR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x888);
    else
        X{64}(t) = TRCITECR_EL1();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TRCITECR_EL2();
    else
        X{64}(t) = TRCITECR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TRCITECR_EL1();
end;
```

MSR TRCITECR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_ITE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nTRCITECR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x888) = X{64}(t);
    else
        TRCITECR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnITE == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnITE == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TRCITECR_EL2() = X{64}(t);
    else
        TRCITECR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TRCITECR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.49 TRCITEEDCR, Instrumentation Trace Extension External Debug Control Register

The TRCITEEDCR characteristics are:

Purpose

Controls instrumentation trace filtering.

Configuration

AArch64 System register TRCITEEDCR bits [31:0] are architecturally mapped to External register TRCITEEDCR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and FEAT_ITE is implemented. Otherwise, direct accesses to TRCITEEDCR are UNDEFINED.

Attributes

TRCITEEDCR is a 64-bit register.

Field descriptions

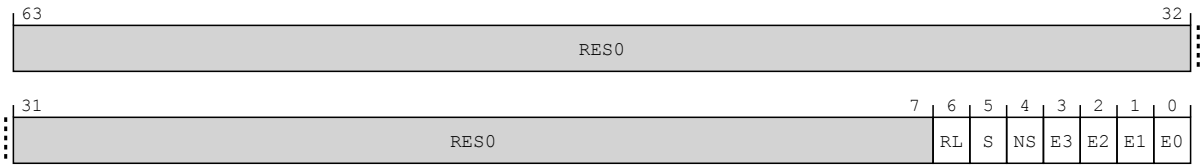

Bits [63:7]

Reserved, RES0.

RL, bit [6]

When FEAT_RME is implemented:

Instrumentation Trace in Realm state.

RL	Meaning
0b0	Instrumentation trace prohibited in Realm state.
0b1	Instrumentation trace permitted in Realm state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When HaveSecureState():

Instrumentation Trace in Secure state.

S	Meaning
`0b0`	Instrumentation trace prohibited in Secure state.
`0b1`	Instrumentation trace permitted in Secure state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

When FEAT_RME is not implemented, this field is used in conjunction with TRCCONFIGR.ITO, TRCITEEDCR.E3, and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Secure state.

When FEAT_RME is implemented, this field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NS, bit [4]

When EL3 is implemented or NonSecureOnlyImplementation():

Instrumentation Trace in Non-secure state.

NS	Meaning
`0b0`	Instrumentation trace prohibited in Non-secure state.
`0b1`	Instrumentation trace permitted in Non-secure state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3, bit [3]

When EL3 is implemented:

Instrumentation Trace Enable at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3	Meaning
0b0	Instrumentation trace prohibited at EL3.
0b1	Instrumentation trace permitted at EL3.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

When FEAT_RME is not implemented, TRCITEEDCR.E3 is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.S to control whether Instrumentation trace is permitted or prohibited at EL3.

When FEAT_RME is implemented, TRCITEEDCR.E3 is used in conjunction with TRCCONFIGR.ITO to control whether Instrumentation trace is permitted or prohibited at EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E<m> , bits [m], for m = 2 to 0

Instrumentation Trace Enable at EL<m>.

E<m>	Meaning
0b0	Instrumentation trace prohibited at EL<m>.
0b1	Instrumentation trace permitted at EL<m>.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This bit is used in conjunction with TRCCONFIGR.ITO, TRCITEEDCR.NS, TRCITEEDCR.S, and TRCITEEDCR.RL to control whether Instrumentation trace is permitted or prohibited at EL<m> in the specified Security states.

TRCITEEDCR.E<2> is RES0 if EL2 is not implemented in any Security states.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCITEEDCR

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCITEEDCR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_ITE)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCITEEDCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCITEEDCR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCITEEDCR();
    end;
end;
```

```
MSR TRCITEEDCR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) &&
     ↪IsFeatureImplemented(FEAT_ITE)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCITEEDCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCITEEDCR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCITEEDCR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.50 TRCOSLSR, Trace OS Lock Status Register

The TRCOSLSR characteristics are:

Purpose

Returns the status of the Trace OS Lock.

Configuration

AArch64 System register TRCOSLSR bits [31:0] are architecturally mapped to External register TRCOSLSR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCOSLSR are UNDEFINED.

Attributes

TRCOSLSR is a 64-bit register.

Field descriptions

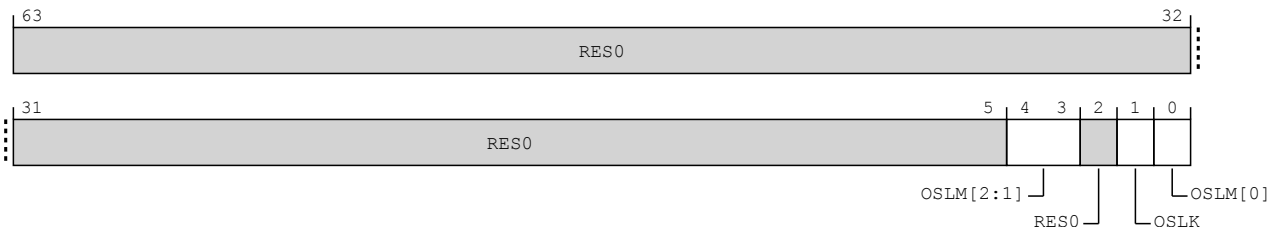

Bits [63:5]

Reserved, RES0.

OSLM, bits [4:3, 0]

OS Lock model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b000	Trace OS Lock is not implemented.
0b010	Trace OS Lock is implemented.
0b100	Trace OS Lock is not implemented, and the trace unit is controlled by the PE OS Lock.

All other values are reserved.

When FEAT_ETE is implemented, the values 0b000 and 0b010 are not permitted.

Access to this field is RO.

Bit [2]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLK, bit [1]

OS Lock status.

OSLK	Meaning
0b0	The OS Lock is unlocked.
0b1	The OS Lock is locked.

When FEAT_ETE is implemented, this field indicates the state of the PE OS Lock.

When FEAT_ETMv4 is implemented, this field indicates the state of the Trace OS Lock.

Accessing TRCOSLSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCOSLSR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b0001	0b100

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCOSLSR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCOSLSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCOSLSR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCOSLSR();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.51 TRCPRGCTLR, Trace Programming Control Register

The TRCPRGCTLR characteristics are:

Purpose

Enables the trace unit.

Configuration

AArch64 System register TRCPRGCTLR bits [31:0] are architecturally mapped to External register TRCPRGCTLR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCPRGCTLR are UNDEFINED.

Attributes

TRCPRGCTLR is a 64-bit register.

Field descriptions

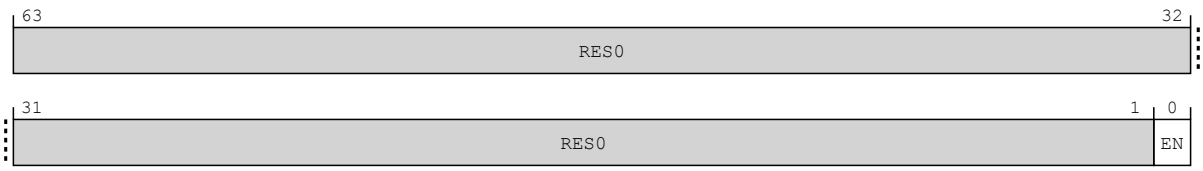

Bits [63:1]

Reserved, RES0.

EN, bit [0]

Trace unit enable.

EN	Meaning
0b0	The trace unit is disabled.
0b1	The trace unit is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Accessing TRCPRGCTLR

Must be programmed.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCPRGCTLR
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCPRGCTLR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCPRGCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCPRGCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCPRGCTLR();
    end;
end;
```

```
MSR TRCPRGCTLR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCPRGCTLR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCPRGCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCPRGCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCPRGCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.52 TRCQCTLR, Trace Q Element Control Register

The TRCQCTLR characteristics are:

Purpose

Controls when Q elements are enabled.

Configuration

AArch64 System register TRCQCTLR bits [31:0] are architecturally mapped to External register TRCQCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR0.QFILT == '1'. Otherwise, direct accesses to TRCQCTLR are UNDEFINED.

Attributes

TRCQCTLR is a 64-bit register.

Field descriptions

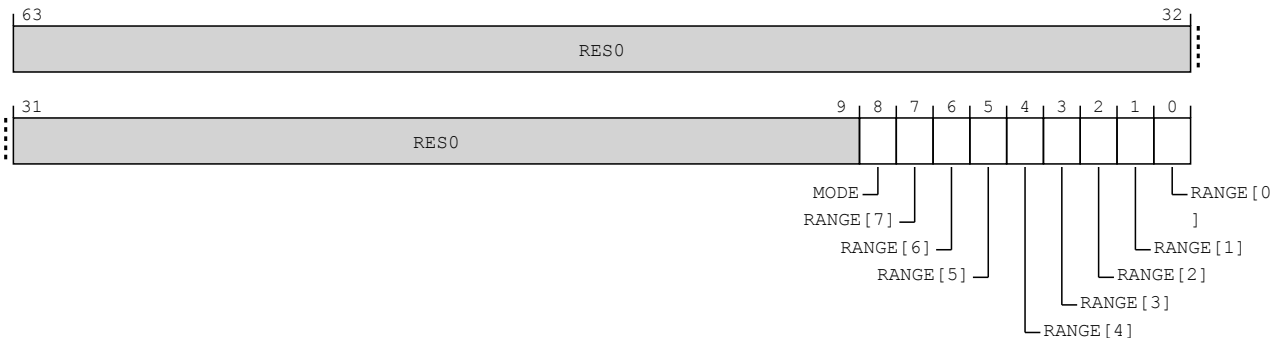

Bits [63:9]

Reserved, RES0.

MODE, bit [8]

Selects whether the Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit is permitted to generate Q elements or address ranges where the trace unit is not permitted to generate Q elements:

MODE	Meaning
0b0	Exclude mode. The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit must not generate Q elements. If no ranges are selected, Q elements are permitted across the entire memory map.
0b1	Include Mode. The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit can generate Q elements. If all the implemented bits in RANGE are set to 0 then Q elements are disabled.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RANGE[<m>] , bits [m], for m = 7 to 0

Specifies whether Address Range Comparator <m> controls Q elements.

RANGE[<m>]	Meaning
0b0	The address range that Address Range Comparator <m> defines is not selected.
0b1	The address range that Address Range Comparator <m> defines is selected.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCQCTLR

Must be programmed if TRCCONFIGR.QE != 0b00.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCQCTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().QFILT == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCQCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCQCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCQCTLR();
    end;
end;
```

```
MSR TRCQCTLR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().QFILT == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        TRCQCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCQCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCQCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.53 TRCRSCTLR<n>, Trace Resource Selection Control Register <n>, n = 2 - 31

The TRCRSCTLR<n> characteristics are:

Purpose

Controls the selection of the resources in the trace unit.

Configuration

Resource selector 0 always returns FALSE.

Resource selector 1 always returns TRUE.

Resource selectors are implemented in pairs. Each odd numbered resource selector is part of a pair with the even numbered resource selector that is numbered as one less than it. For example, resource selectors 2 and 3 form a pair.

AArch64 System register TRCRSCTLR<n> bits [31:0] are architecturally mapped to External register TRCRSCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and ((UInt(TRCIDR4.NUMRSPAIR) + 1) * 2) > n. Otherwise, direct accesses to TRCRSCTLR<n> are UNDEFINED.

Attributes

TRCRSCTLR<n> is a 64-bit register.

Field descriptions

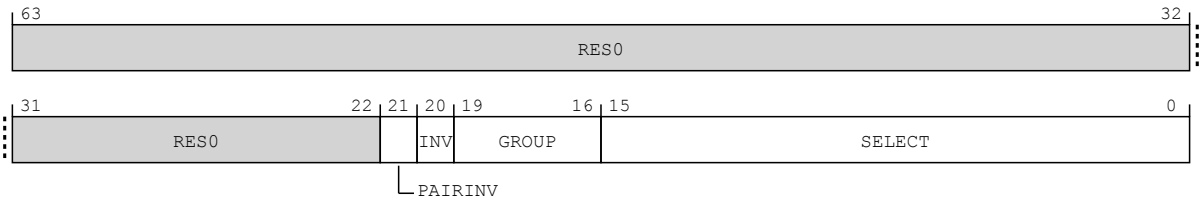

Bits [63:22]

Reserved, RES0.

PAIRINV, bit [21]

When (n MOD 2) == 0:

Controls whether the combined result from a resource selector pair is inverted.

PAIRINV	Meaning
0b0	Do not invert the combined output of the 2 resource selectors.
0b1	Invert the combined output of the 2 resource selectors.

If:

- A is the register TRCRSCTLR<n>.
- B is the register TRCRSCTLR<n+1>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Then the combined output of the 2 resource selectors A and B depends on the value of (A.PAIRINV, A.INV, B.INV) as follows:

- `0b000` -> A and B.
- `0b001` -> Reserved.
- `0b010` -> not(A) and B.
- `0b011` -> not(A) and not(B).
- `0b100` -> not(A) or not(B).
- `0b101` -> not(A) or B.
- `0b110` -> Reserved.
- `0b111` -> A or B.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

INV, bit [20]

Controls whether the resource, that TRCRSCTLR<n>.GROUP and TRCRSCTLR<n>.SELECT selects, is inverted.

INV	Meaning
`0b0`	Do not invert the output of this selector.
`0b1`	Invert the output of this selector.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

GROUP, bits [19:16]

Selects a group of resources.

GROUP	Meaning	SELECT
`0b0000`	External Input Selectors.	SELECT encoding for External Input Selectors
`0b0001`	PE Comparator Inputs.	SELECT encoding for PE Comparator Inputs
`0b0010`	Counters and Sequencer.	SELECT encoding for Counters and Sequencer
`0b0011`	Single-shot Comparator Controls.	SELECT encoding for Single-shot Comparator Controls
`0b0100`	Single Address Comparators.	SELECT encoding for Single Address Comparators
`0b0101`	Address Range Comparators.	SELECT encoding for Address Range Comparators
`0b0110`	Context Identifier Comparators.	SELECT encoding for Context Identifier Comparators
`0b0111`	Virtual Context Identifier Comparators.	SELECT encoding for Virtual Context Identifier Comparators

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT, bits [15:0]

Resource Specific Controls. Contains the controls specific to the resource group selected by GROUP, described in the following sections.

SELECT encoding for External Input Selectors

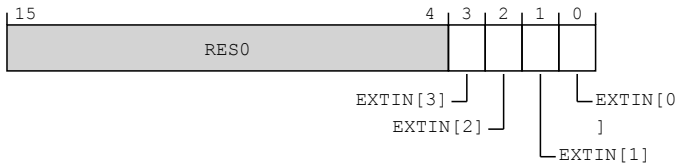

Bits [15:4]

Reserved, RES0.

EXTIN[<m>] , bits [m], for m = 3 to 0

Selects one or more External Inputs.

EXTIN[<m>]	Meaning
0b0	Ignore EXTIN <m>.
0b1	Select EXTIN <m>.

This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for PE Comparator Inputs

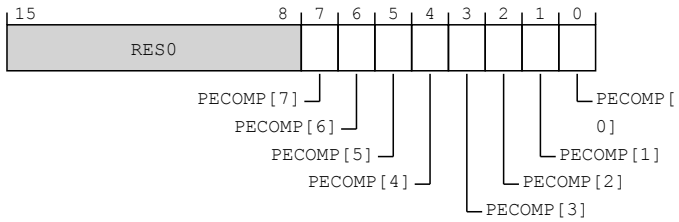

Bits [15:8]

Reserved, RES0.

PECOMP[<m>] , bits [m], for m = 7 to 0

Selects one or more PE Comparator Inputs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PECOMP[<m>]	Meaning
0b0	Ignore PE Comparator Input <m>.
0b1	Select PE Comparator Input <m>.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Counters and Sequencer

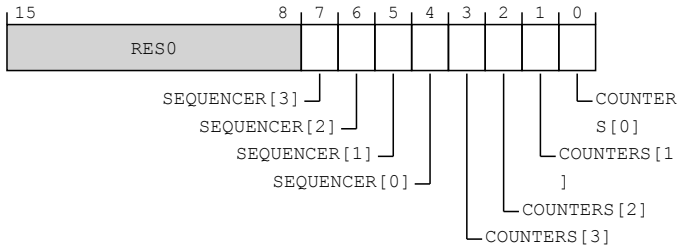

Bits [15:8]

Reserved, RES0.

SEQUENCER[<m>] , bits [m+4], for m = 3 to 0

Sequencer states.

SEQUENCER[<m>]	Meaning
0b0	Ignore Sequencer state <m>.
0b1	Select Sequencer state <m>.

This bit is RES0 if m >= TRCIDR5.NUMSEQSTATE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COUNTERS[<m>] , bits [m], for m = 3 to 0

Counters resources at zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COUNTERS[<m>]	Meaning
0b0	Ignore Counter <m>.
0b1	Select Counter <m> is zero.

This bit is RES0 if m >= TRCIDR5.NUMCNTR.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Single-shot Comparator Controls

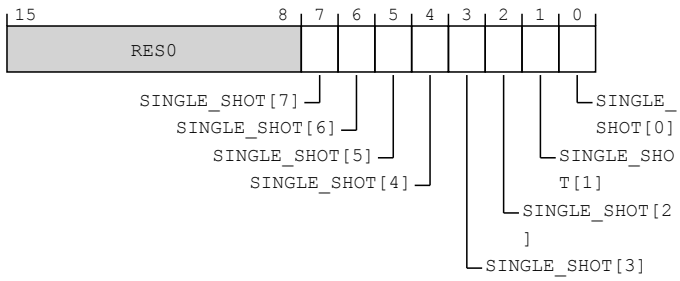

Bits [15:8]

Reserved, RES0.

SINGLE_SHOT[<m>] , bits [m], for m = 7 to 0

Selects one or more Single-shot Comparator Controls.

SINGLE_SHOT[<m>]	Meaning
0b0	Ignore Single-shot Comparator Control <m>.
0b1	Select Single-shot Comparator Control <m>.

This bit is RES0 if m >= TRCIDR4.NUMSSCC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Single Address Comparators

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

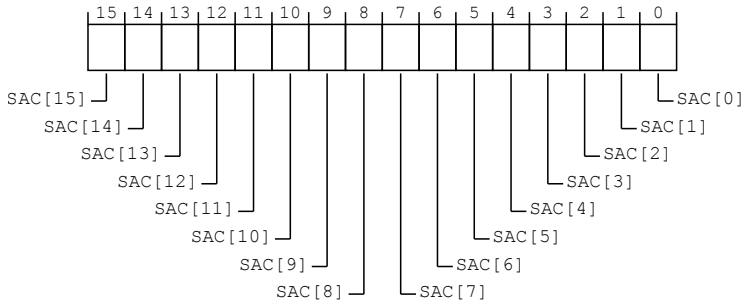

SAC[<m>] , bits [m], for m = 15 to 0

Selects one or more Single Address Comparators.

SAC[<m>]	Meaning
0b0	Ignore Single Address Comparator <m>.
0b1	Select Single Address Comparator <m>.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Address Range Comparators

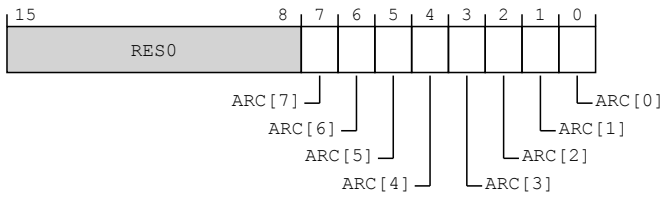

Bits [15:8]

Reserved, RES0.

ARC[<m>] , bits [m], for m = 7 to 0

Selects one or more Address Range Comparators.

ARC[<m>]	Meaning
0b0	Ignore Address Range Comparator <m>.
0b1	Select Address Range Comparator <m>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Context Identifier Comparators

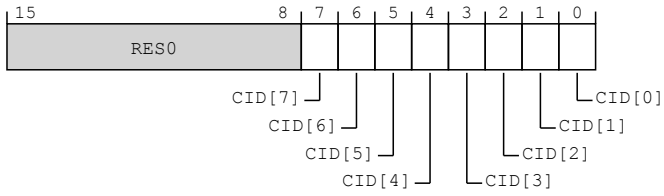

Bits [15:8]

Reserved, RES0.

CID[<m>] , bits [m], for m = 7 to 0

Selects one or more Context Identifier Comparators.

CID[<m>]	Meaning
0b0	Ignore Context Identifier Comparator <m>.
0b1	Select Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMCIDC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Virtual Context Identifier Comparators

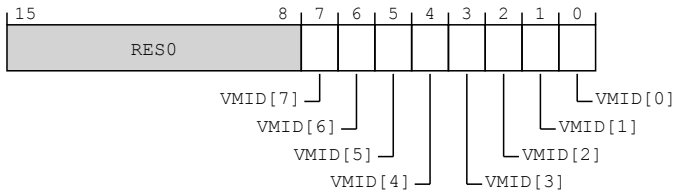

Bits [15:8]

Reserved, RES0.

VMID[<m>] , bits [m], for m = 7 to 0

Selects one or more Virtual Context Identifier Comparators.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID[<m>]	Meaning
0b0	Ignore Virtual Context Identifier Comparator <m>.
0b1	Select Virtual Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMVMIDC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCRSCTLR<n>

Must be programmed if any of the following are true:

- TRCCNTCTLR<a>.RLDEVENT_TYPE == 0 and TRCCNTCTLR<a>.RLDEVENT_SEL == n.
- TRCCNTCTLR<a>.RLDEVENT_TYPE == 1 and TRCCNTCTLR<a>.RLDEVENT_SEL == n/2.
- TRCCNTCTLR<a>.CNTEVENT_TYPE == 0 and TRCCNTCTLR<a>.CNTEVENT_SEL == n.
- TRCCNTCTLR<a>.CNTEVENT_TYPE == 1 and TRCCNTCTLR<a>.CNTEVENT_SEL == n/2.
- TRCEVENTCTL0R.EVENT0_TYPE == 0 and TRCEVENTCTL0R.EVENT0_SEL == n.
- TRCEVENTCTL0R.EVENT0_TYPE == 1 and TRCEVENTCTL0R.EVENT0_SEL == n/2.
- TRCEVENTCTL0R.EVENT1_TYPE == 0 and TRCEVENTCTL0R.EVENT1_SEL == n.
- TRCEVENTCTL0R.EVENT1_TYPE == 1 and TRCEVENTCTL0R.EVENT1_SEL == n/2.
- TRCEVENTCTL0R.EVENT2_TYPE == 0 and TRCEVENTCTL0R.EVENT2_SEL == n.
- TRCEVENTCTL0R.EVENT2_TYPE == 1 and TRCEVENTCTL0R.EVENT2_SEL == n/2.
- TRCEVENTCTL0R.EVENT3_TYPE == 0 and TRCEVENTCTL0R.EVENT3_SEL == n.
- TRCEVENTCTL0R.EVENT3_TYPE == 1 and TRCEVENTCTL0R.EVENT3_SEL == n/2.
- TRCSEQEVR<a>.B_TYPE == 0 and TRCSEQEVR<a>.B_SEL = n.
- TRCSEQEVR<a>.B_TYPE == 1 and TRCSEQEVR<a>.B_SEL = n/2.
- TRCSEQEVR<a>.F_TYPE == 0 and TRCSEQEVR<a>.F_SEL = n.
- TRCSEQEVR<a>.F_TYPE == 1 and TRCSEQEVR<a>.F_SEL = n/2.
- TRCSEQRSTEVR.RST_TYPE == 0 and TRCSEQRSTEVR.RST_SEL == n.
- TRCSEQRSTEVR.RST_TYPE == 1 and TRCSEQRSTEVR.RST_SEL == n/2.
- TRCTSCTLR.EVENT_TYPE == 0 and TRCTSCTLR.EVENT_SEL == n.
- TRCTSCTLR.EVENT_TYPE == 1 and TRCTSCTLR.EVENT_SEL == n/2.
- TRCVICTLR.EVENT_TYPE == 0 and TRCVICTLR.EVENT_SEL == n.
- TRCVICTLR.EVENT_TYPE == 1 and TRCVICTLR.EVENT_SEL == n/2.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCRSCTLR<m> ; Where m = 2-31
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	m[3:0]	0b00:m[4]

```
let m:integer = UInt(op2[0] :: CRm[3:0]);

if m >= NUM_TRACE_RESOURCE_SELECTOR_PAIRS * 2 then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif CPACR_EL1().TTA == '1' then
            AArch64_SystemAccessTrap(EL1, 0x18);
        elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().TRC == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCRSCTLR(m);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
            Undefined();
        elsif CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCRSCTLR(m);
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TTA == '1' then
            AArch64_SystemAccessTrap(EL3, 0x18);
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCRSCTLR(m);
        end;
    end;
```

```
MSR TRCRSCTLR<m>, <Xt> ; Where m = 2-31
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	m[3:0]	0b00:m[4]

```
let m:integer = UInt(op2[0] :: CRm[3:0]);

if m >= NUM_TRACE_RESOURCE_SELECTOR_PAIRS * 2 then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCRSCTLR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCRSCTLR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCRSCTLR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.54 TRCRSR, Trace Resources Status Register

The TRCRSR characteristics are:

Purpose

Use this to set, or read, the status of the resources.

Configuration

AArch64 System register TRCRSR bits [31:0] are architecturally mapped to External register TRCRSR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCRSR are UNDEFINED.

Attributes

TRCRSR is a 64-bit register.

Field descriptions

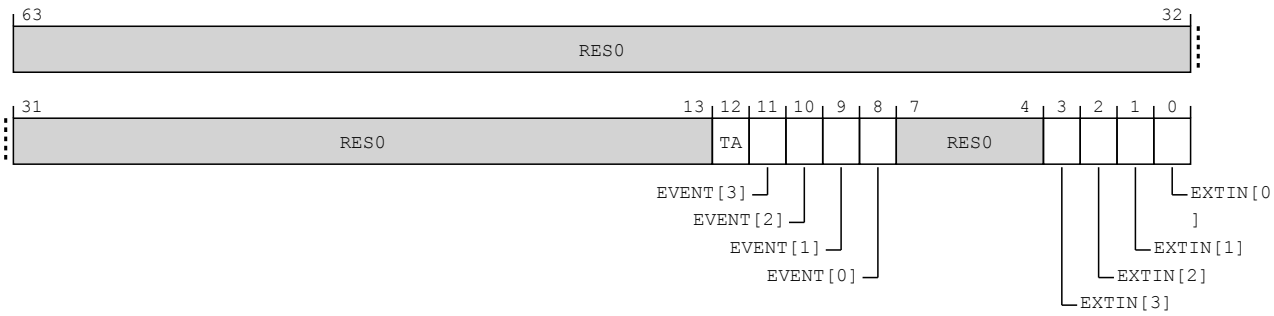

Bits [63:13]

Reserved, RES0.

TA, bit [12]

Tracing active.

TA	Meaning
0b0	Tracing is not active.
0b1	Tracing is active.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

EVENT[<m>] , bits [m+8], for m = 3 to 0

Untraced status of ETEEvents.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVENT[<m>]	Meaning
0b0	An ETEEvent <m> has not occurred.
0b1	An ETEEvent <m> has occurred while the resources were in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When TRCIDR4.NUMRSPAIR == ‘0000’, access to this field is RES0.
- Access to this field is RES0 if all of the following are true:
  - TRCIDR4.NUMRSPAIR != ‘0000’
  - m > UInt(TRCIDR0.NUMEVENT)
- Otherwise, access to this field is RW.

Bits [7:4]

Reserved, RES0.

EXTIN[<m>] , bits [m], for m = 3 to 0

The sticky status of the External Input Selectors.

EXTIN[<m>]	Meaning
0b0	An event selected by External Input Selector <m> has not occurred.
0b1	At least one event selected by External Input Selector <m> has occurred while the resources were in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR5.NUMEXTINSEL), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCRSR

Must always be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCRSR
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1010	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCRSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCRSR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCRSR();
    end;
end;
```

```
MSR TRCRSR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1010	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCRSR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCRSR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCRSR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.55 TRCSEQEVR<n>, Trace Sequencer State Transition Control Register <n>, n = 0 - 2

The TRCSEQEVR<n> characteristics are:

Purpose

Moves the Sequencer state:

- Backwards, from state n+1 to state n when a programmed resource event occurs.
- Forwards, from state n to state n+1 when a programmed resource event occurs.

Configuration

AArch64 System register TRCSEQEVR<n> bits [31:0] are architecturally mapped to External register TRCSEQEVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR5.NUMSEQSTATE != ‘000’. Otherwise, direct accesses to TRCSEQEVR<n> are UNDEFINED.

Attributes

TRCSEQEVR<n> is a 64-bit register.

Field descriptions

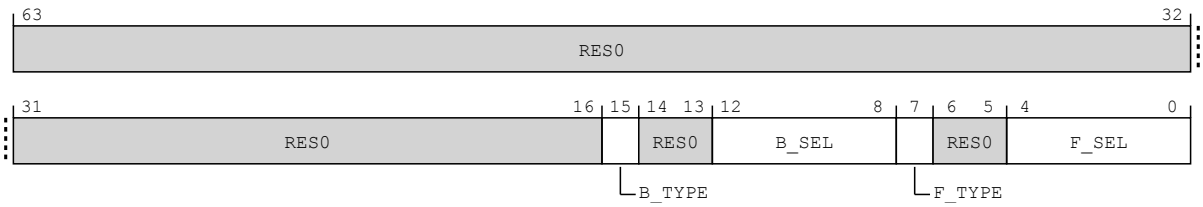

Bits [63:16]

Reserved, RES0.

B_TYPE, bit [15]

Backward field. Selects an event that causes the Sequencer to move from state n+1 to state n. For example, if TRCSEQEVR2.B_SEL == `0x14`, then when event `0x14` occurs, the Sequencer moves from state 3 to state 2.

Chooses the type of Resource Selector.

B_TYPE	Meaning
`0b0`	A single Resource Selector. TRCSEQEVR<n>.B.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCSEQEVR<n>.B.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR<n>.B.SEL[4] is RES0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

B_SEL, bits [12:8]

Backward field. Selects an event that causes the Sequencer to move from state n+1 to state n. For example, if TRCSEQEVR2.B_SEL == `0x14`, then when event `0x14` occurs, the Sequencer moves from state 3 to state 2.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.B.TYPE controls whether TRCSEQEVR<n>.B.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

F_TYPE, bit [7]

Forward field. Selects an event that causes the Sequencer to move from state n to state n+1. For example, if TRCSEQEVR1.F_SEL == `0x12`, then when event `0x12` occurs, the Sequencer moves from state 1 to state 2.

Chooses the type of Resource Selector.

F_TYPE	Meaning
`0b0`	A single Resource Selector. TRCSEQEVR<n>.F.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCSEQEVR<n>.F.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR<n>.F.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

F_SEL, bits [4:0]

Forward field. Selects an event that causes the Sequencer to move from state n to state n+1. For example, if TRCSEQEVR1.F_SEL == `0x12`, then when event `0x12` occurs, the Sequencer moves from state 1 to state 2.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.F.TYPE controls whether TRCSEQEVR<n>.F.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQEVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.SEQUENCER != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSEQEVR<m> ; Where m = 0-2
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b00:m[1:0]	0b100

```
let m:integer = UInt(CRm[1:0]);

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQEVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQEVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQEVR(m);
    end;
end;
```

MSR TRCSEQEVR<m>, <Xt> ; Where m = 0-2

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b00:m[1:0]	0b100

```
let m:integer = UInt(CRm[1:0]);

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQEVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        TRCSEQEVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQEVR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.56 TRCSEQRSTEVR, Trace Sequencer Reset Control Register

The TRCSEQRSTEVR characteristics are:

Purpose

Moves the Sequencer to state 0 when a programmed resource event occurs.

Configuration

AArch64 System register TRCSEQRSTEVR bits [31:0] are architecturally mapped to External register TRCSEQRSTEVR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR5.NUMSEQSTATE != ‘000’. Otherwise, direct accesses to TRCSEQRSTEVR are UNDEFINED.

Attributes

TRCSEQRSTEVR is a 64-bit register.

Field descriptions

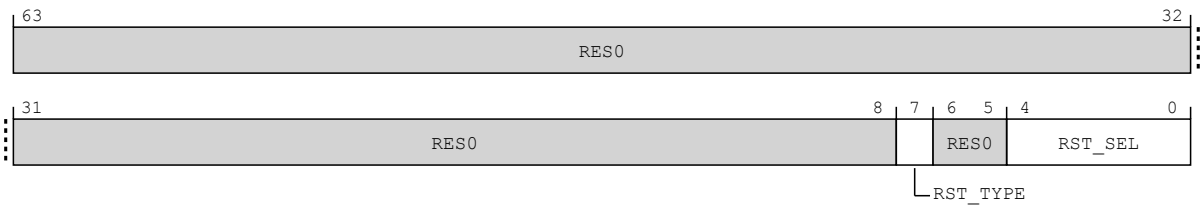

Bits [63:8]

Reserved, RES0.

RST_TYPE, bit [7]

Reset field. Selects an event that causes the Sequencer to move to state 0.

Chooses the type of Resource Selector.

RST_TYPE	Meaning
0b0	A single Resource Selector. TRCSEQRSTEVR.RST.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCSEQRSTEVR.RST.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQRSTEVR.RST.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RST_SEL, bits [4:0]

Reset field. Selects an event that causes the Sequencer to move to state 0.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQRSTEVR.RST.TYPE controls whether TRCSEQRSTEVR.RST.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQRSTEVR

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.SEQUENCER != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSEQRSTEVR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0110	0b100

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR5().NUMSEQSTATE !=
     ↪'000') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQRSTEVR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQRSTEVR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQRSTEVR();
    end;
end;
```

```
MSR TRCSEQRSTEVR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0110	0b100

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR5().NUMSEQSTATE !=
     ↪'000') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQRSTEVR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQRSTEVR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQRSTEVR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.57 TRCSEQSTR, Trace Sequencer State Register

The TRCSEQSTR characteristics are:

Purpose

Use this to set, or read, the Sequencer state.

Configuration

AArch64 System register TRCSEQSTR bits [31:0] are architecturally mapped to External register TRCSEQSTR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR5.NUMSEQSTATE != ‘000’. Otherwise, direct accesses to TRCSEQSTR are UNDEFINED.

Attributes

TRCSEQSTR is a 64-bit register.

Field descriptions

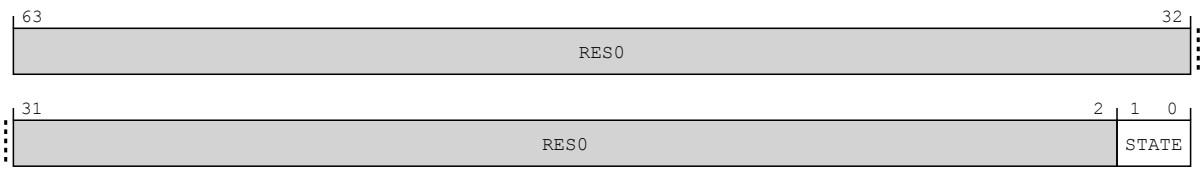

Bits [63:2]

Reserved, RES0.

STATE, bits [1:0]

Set or returns the state of the Sequencer.

STATE	Meaning
0b00	State 0.
0b01	State 1.
0b10	State 2.
0b11	State 3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQSTR

Must be programmed if TRCRSCTLR<a>.GROUP == 0b0010 and TRCRSCTLR<a>.SEQUENCER != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, TRCSEQSTR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0111	0b100

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR5().NUMSEQSTATE !=
     ↪'000') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCSEQSTR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQSTR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQSTR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSEQSTR();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TRCSEQSTR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0111	0b100

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR5().NUMSEQSTATE !=
     ↪'000') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCSEQSTR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQSTR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQSTR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSEQSTR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.58 TRCSSCCR<n>, Trace Single-shot Comparator Control Register <n>, n = 0 - 7

The TRCSSCCR<n> characteristics are:

Purpose

Controls the corresponding Single-shot Comparator Control resource.

Configuration

AArch64 System register TRCSSCCR<n> bits [31:0] are architecturally mapped to External register TRCSSCCR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCCR<n> are UNDEFINED.

Attributes

TRCSSCCR<n> is a 64-bit register.

Field descriptions

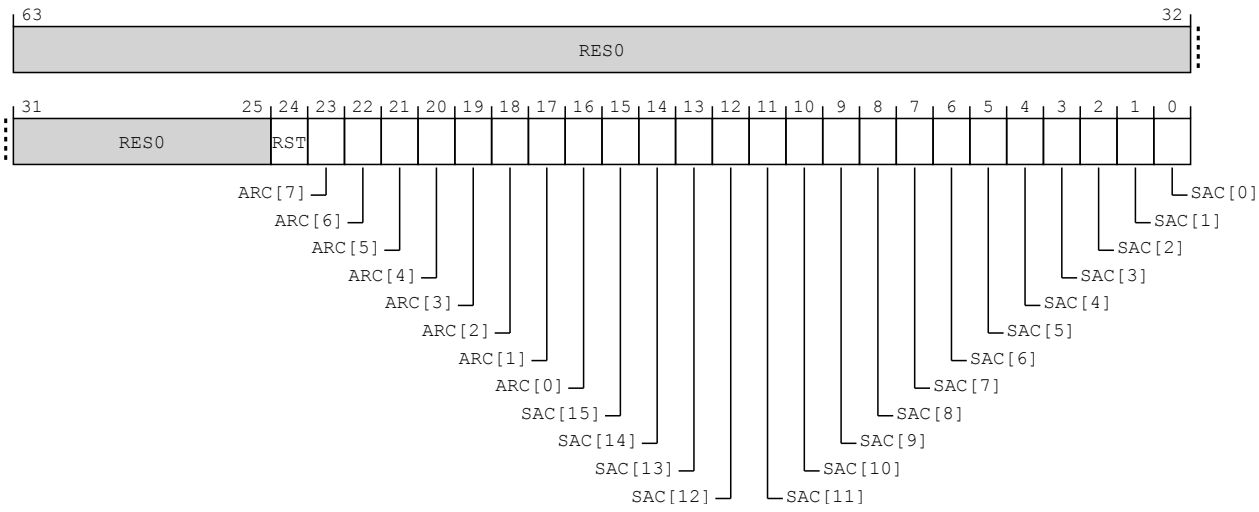

Bits [63:25]

Reserved, RES0.

RST, bit [24]

Selects the Single-shot Comparator Control mode.

RST	Meaning
0b0	The Single-shot Comparator Control is in single-shot mode.
0b1	The Single-shot Comparator Control is in multi-shot mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARC[<m>] , bits [m+16], for m = 7 to 0

Selects one or more Address Range Comparators for Single-shot control.

ARC[<m>]	Meaning
0b0	The Address Range Comparator <m>, is not selected for Single-shot control.
0b1	The Address Range Comparator <m>, is selected for Single-shot control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

SAC[<m>] , bits [m], for m = 15 to 0

Selects one or more Single Address Comparators for Single-shot control.

SAC[<m>]	Meaning
0b0	The Single Address Comparator <m>, is not selected for Single-shot control.
0b1	The Single Address Comparator <m>, is selected for Single-shot control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCSSCCR<n>

Must be programmed if any TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSSCCR<m> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b0:m[2:0]	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[2:0]);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSCCR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSCCR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSCCR(m);
    end;
end;
```

```
MSR TRCSSCCR<m>, <Xt> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b0:m[2:0]	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[2:0]);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSCCR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSCCR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSCCR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.59 TRCSSCSR<n>, Trace Single-shot Comparator Control Status Register <n>, n = 0 - 7

The TRCSSCSR<n> characteristics are:

Purpose

Returns the status of the corresponding Single-shot Comparator Control.

Configuration

AArch64 System register TRCSSCSR<n> bits [31:0] are architecturally mapped to External register TRCSSCSR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCSR<n> are UNDEFINED.

Attributes

TRCSSCSR<n> is a 64-bit register.

Field descriptions

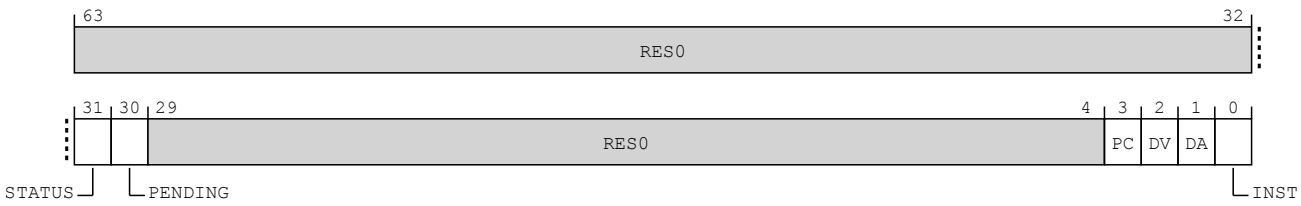

Bits [63:32]

Reserved, RES0.

STATUS, bit [31]

Single-shot Comparator Control status. Indicates if any of the comparators selected by this Single-shot Comparator control have matched. The selected comparators are defined by TRCSSCCR<n>.ARC, TRCSSCCR<n>.SAC, and TRCSSPCICR<n>.PC.

STATUS	Meaning
0b0	No match has occurred. When the first match occurs, this field takes a value of 1. It remains at 1 until explicitly modified by a write to this register.
0b1	One or more matches has occurred. If TRCSSCCR<n>.RST == 0 then: • There is only one match and no more matches are possible. • Software must reset this field to 0 to re-enable the Single-shot Comparator Control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

PENDING, bit [30]

Single-shot pending status. The Single-shot Comparator Control fired while the resources were in the Paused state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PENDING	Meaning
0b0	No match has occurred.
0b1	One or more matches has occurred.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [29:4]

Reserved, RES0.

PC, bit [3]

PE Comparator Input support. Indicates if the Single-shot Comparator Control supports PE Comparator Inputs.

PC	Meaning
0b0	This Single-shot Comparator Control does not support PE Comparator Inputs. Selecting any PE Comparator Inputs using the associated TRCSSPCICR<n> results in CONSTRAINED UNPREDICTABLE behavior of the Single-shot Comparator Control resource. The Single-shot Comparator Control might match unexpectedly or might not match.
0b1	This Single-shot Comparator Control supports PE Comparator Inputs.

Access to this field is RO.

DV, bit [2]

Data value comparator support. Data value comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

DV	Meaning
0b0	This Single-shot Comparator Control does not support data value comparisons.
0b1	This Single-shot Comparator Control supports data value comparisons.

This field reads as 0.

Access to this field is RO.

DA, bit [1]

Data Address Comparator support. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DA	Meaning
0b0	This Single-shot Comparator Control does not support data address comparisons.
0b1	This Single-shot Comparator Control supports data address comparisons.

This field reads as 0.

Access to this field is RO.

INST, bit [0]

Instruction Address Comparator support. Indicates if the Single-shot Comparator Control supports instruction address comparisons.

INST	Meaning
0b0	This Single-shot Comparator Control does not support instruction address comparisons.
0b1	This Single-shot Comparator Control supports instruction address comparisons.

This field reads as 1.

Access to this field is RO.

Accessing TRCSSCSR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSSCSR<m> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b1:m[2:0]	0b010

```
let m:integer = UInt(CRm[2:0]);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HDFGRTR_EL2().TRCSSCSRn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSCSR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSCSR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSCSR(m);
    end;
end;
```

```
MSR TRCSSCSR<m>, <Xt> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b1:m[2:0]	0b010

```
let m:integer = UInt(CRm[2:0]);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCSSCSRn == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSCSR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSCSR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSCSR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.60 TRCSSPCICR<n>, Trace Single-shot Processing Element Comparator Input Control Register <n>, n = 0 - 7

The TRCSSPCICR<n> characteristics are:

Purpose

Returns the status of the corresponding Single-shot Comparator Control.

Configuration

AArch64 System register TRCSSPCICR<n> bits [31:0] are architecturally mapped to External register TRCSSPCICR<n>[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, UInt(TRCIDR4.NUMSSCC) > n, UInt(TRCIDR4.NUMPC) > 0, and TRCSSCSR<n>.PC == '1'. Otherwise, direct accesses to TRCSSPCICR<n> are UNDEFINED.

Attributes

TRCSSPCICR<n> is a 64-bit register.

Field descriptions

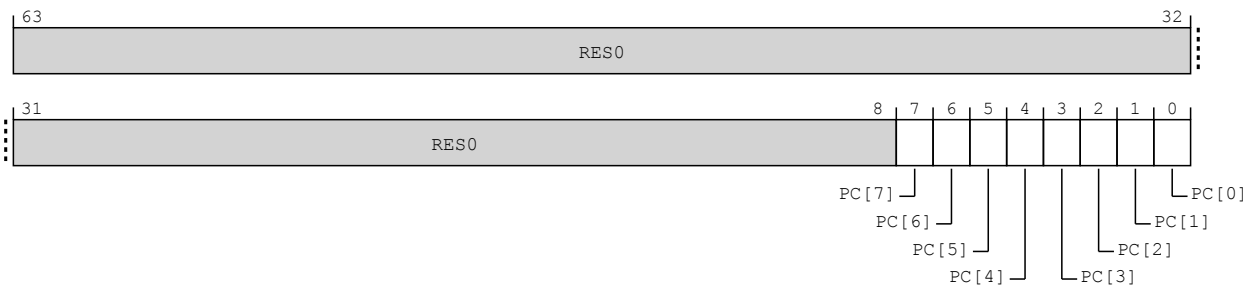

Bits [63:8]

Reserved, RES0.

PC[<m>] , bits [m], for m = 7 to 0

Selects one or more PE Comparator Inputs for Single-shot control.

PC[<m>]	Meaning
0b0	The single PE Comparator Input <m>, is not selected as for Single-shot control.
0b1	The single PE Comparator Input <m>, is selected as for Single-shot control.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCSSPCICR<n>

Must be programmed if implemented and any TRCRSCTLR<a>.GROUP == `0b0011` and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSSPCICR<m> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b0:m[2:0]	0b011

```
let m:integer = UInt(CRm[2:0]);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSPCICR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSPCICR(m);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSSPCICR(m);
    end;
end;
```

```
MSR TRCSSPCICR<m>, <Xt> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0001	0b0:m[2:0]	0b011

```
let m:integer = UInt(CRm[2:0]);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSPCICR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSPCICR(m) = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSSPCICR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.61 TRCSTALLCTLR, Trace Stall Control Register

The TRCSTALLCTLR characteristics are:

Purpose

Enables trace unit functionality that prevents trace unit buffer overflows.

Configuration

AArch64 System register TRCSTALLCTLR bits [31:0] are architecturally mapped to External register TRCSTALLCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR3.STALLCTL == '1'. Otherwise, direct accesses to TRCSTALLCTLR are UNDEFINED.

Attributes

TRCSTALLCTLR is a 64-bit register.

Field descriptions

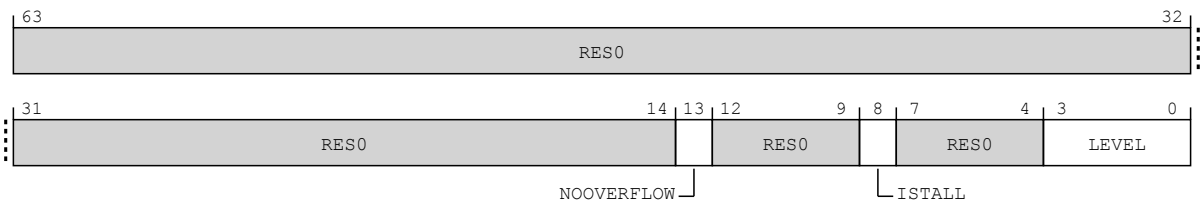

Bits [63:14]

Reserved, RES0.

NOOVERFLOW, bit [13]

When TRCIDR3.NOOVERFLOW == '1':

Trace overflow prevention.

NOOVERFLOW	Meaning
0b0	Trace unit buffer overflow prevention is disabled.
0b1	Trace unit buffer overflow prevention is enabled.

Note

Enabling this feature might cause a significant performance impact.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

ISTALL, bit [8]

Instruction stall control. Controls if a trace unit can stall the PE when the trace buffer space is less than LEVEL.

ISTALL	Meaning
`0b0`	The trace unit must not stall the PE.
`0b1`	The trace unit can stall the PE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

LEVEL, bits [3:0]

Threshold level field. The field can support 16 monotonic levels from `0b0000` to `0b1111`.

The value `0b0000` defines the Minimal invasion level. This setting has a greater risk of a trace unit buffer overflow.

The value `0b1111` defines the Maximum invasion level. This setting has a reduced risk of a trace unit buffer overflow.

Note

For some implementations, invasion might occur at the minimal invasion level.

One or more of the least significant bits of LEVEL are permitted to be RES0. Arm recommends that LEVEL[3:2] are fully implemented. Arm strongly recommends that LEVEL[3] is always implemented. If one or more bits are RES0 and are written with a nonzero value, the effective value of LEVEL is rounded down to the nearest power of 2 value which has the RES0 bits as zero. For example, if LEVEL[1:0] are RES0 and a value of `0b1110` is written to LEVEL, the effective value of LEVEL is `0b1100`.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSTALLCTLR

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSTALLCTLR
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1011	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR3().STALLCTL == '1')
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSTALLCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSTALLCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSTALLCTLR();
    end;
end;
```

```
MSR TRCSTALLCTLR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1011	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR3().STALLCTL == '1')
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSTALLCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSTALLCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSTALLCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.62 TRCSTATR, Trace Status Register

The TRCSTATR characteristics are:

Purpose

Returns the trace unit status.

Configuration

AArch64 System register TRCSTATR bits [31:0] are architecturally mapped to External register TRCSTATR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCSTATR are UNDEFINED.

Attributes

TRCSTATR is a 64-bit register.

Field descriptions

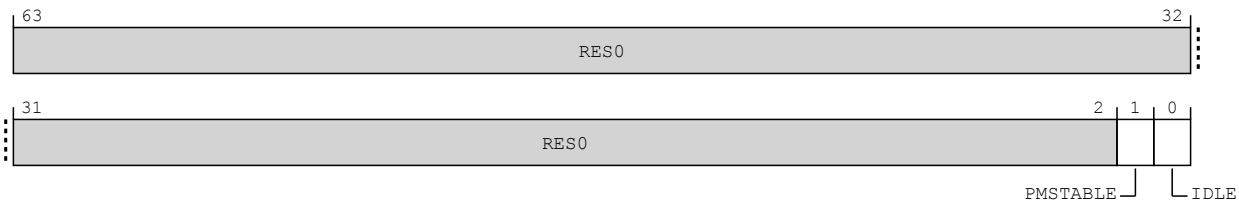

Bits [63:2]

Reserved, RES0.

PMSTABLE, bit [1]

Programmers' model stable.

PMSTABLE	Meaning
0b0	The programmers' model is not stable.
0b1	The programmers' model is stable.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - TRCPRGCTLR.EN == '1'
  - !OSLockStatus()
- Otherwise, access to this field is RO.

IDLE, bit [0]

Idle status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDLE	Meaning
0b0	The trace unit is not idle.
0b1	The trace unit is idle.

Accessing TRCSTATR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSTATR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCSTATR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSTATR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSTATR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSTATR();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.63 TRCSYNCPR, Trace Synchronization Period Register

The TRCSYNCPR characteristics are:

Purpose

Controls how often trace protocol synchronization requests occur.

Configuration

AArch64 System register TRCSYNCPR bits [31:0] are architecturally mapped to External register TRCSYNCPR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCSYNCPR are UNDEFINED.

Attributes

TRCSYNCPR is a 64-bit register.

Field descriptions

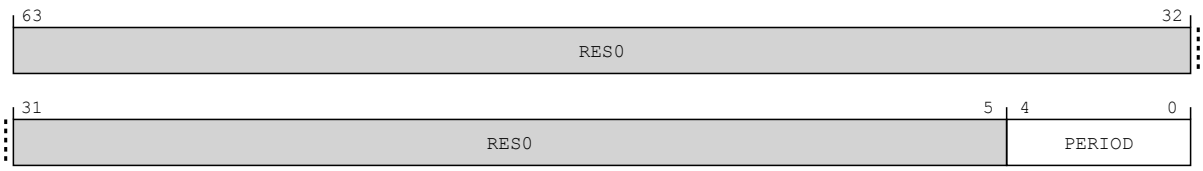

Bits [63:5]

Reserved, RES0.

PERIOD, bits [4:0]

Defines the number of bytes of trace between each periodic trace protocol synchronization request.

PERIOD	Meaning
0b00000	Trace protocol synchronization is disabled.
0b01000	Trace protocol synchronization request occurs after 2^8 bytes of trace.
0b01001	Trace protocol synchronization request occurs after 2^9 bytes of trace.
0b01010	Trace protocol synchronization request occurs after 2^{10} bytes of trace.
0b01011	Trace protocol synchronization request occurs after 2^{11} bytes of trace.
0b01100	Trace protocol synchronization request occurs after 2^{12} bytes of trace.
0b01101	Trace protocol synchronization request occurs after 2^{13} bytes of trace.
0b01110	Trace protocol synchronization request occurs after 2^{14} bytes of trace.
0b01111	Trace protocol synchronization request occurs after 2^{15} bytes of trace.
0b10000	Trace protocol synchronization request occurs after 2^{16} bytes of trace.
0b10001	Trace protocol synchronization request occurs after 2^{17} bytes of trace.
0b10010	Trace protocol synchronization request occurs after 2^{18} bytes of trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PERIOD	Meaning
0b10011	Trace protocol synchronization request occurs after 2^{19} bytes of trace.
0b10100	Trace protocol synchronization request occurs after 2^{20} bytes of trace.

Other values are reserved. If a reserved value is programmed into PERIOD, then the behavior of the synchronization period counter is CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:

- No trace protocol synchronization requests are generated by this counter.
- Trace protocol synchronization requests occur at the specified period.
- Trace protocol synchronization requests occur at some other UNKNOWN period which can vary.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSYNCPR

Must be programmed if TRCIDR3.SYNCPR == 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCSYNCPR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1101	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSYNCPR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSYNCPR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCSYNCPR();
    end;
end;
```

```
MSR TRCSYNCPR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1101	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSYNCPR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSYNCPR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCSYNCPR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.64 TRCTRACEIDR, Trace ID Register

The TRCTRACEIDR characteristics are:

Purpose

Sets the trace ID for instruction trace.

Configuration

AArch64 System register TRCTRACEIDR bits [31:0] are architecturally mapped to External register TRCTRACEIDR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCTRACEIDR are UNDEFINED.

Attributes

TRCTRACEIDR is a 64-bit register.

Field descriptions

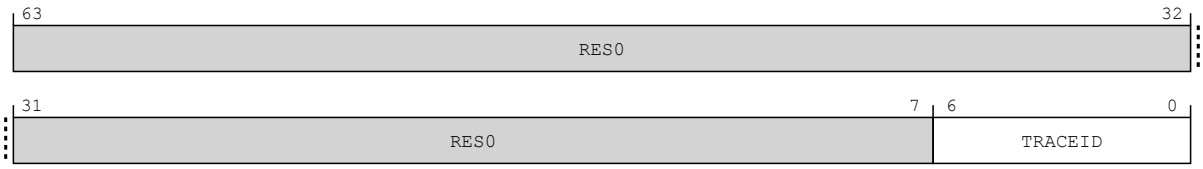

Bits [63:7]

Reserved, RES0.

TRACEID, bits [6:0]

Trace ID field. Sets the trace ID value for instruction trace. The width of the field is indicated by the value of TRCIDR5.TRACEIDSIZE. Unimplemented bits are RES0.

If an implementation supports AMBA ATB, then:

- The width of the field is 7 bits.
- Writing a reserved trace ID value does not affect behavior of the trace unit but it might cause UNPREDICTABLE behavior of the trace capture infrastructure.

See the AMBA ATB Protocol Specification for information about which ATID values are reserved.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCTRACEIDR

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCTRACEIDR
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCTRACEIDR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCTRACEIDR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCTRACEIDR();
    end;
end;
```

```
MSR TRCTRACEIDR, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b001

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCTRACEIDR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCTRACEIDR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCTRACEIDR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.65 TRCTSCTLR, Trace Timestamp Control Register

The TRCTSCTLR characteristics are:

Purpose

Controls the insertion of global timestamps in the trace stream.

Configuration

AArch64 System register TRCTSCTLR bits [31:0] are architecturally mapped to External register TRCTSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and TRCIDR0.TSSIZE != '00000'. Otherwise, direct accesses to TRCTSCTLR are UNDEFINED.

Attributes

TRCTSCTLR is a 64-bit register.

Field descriptions

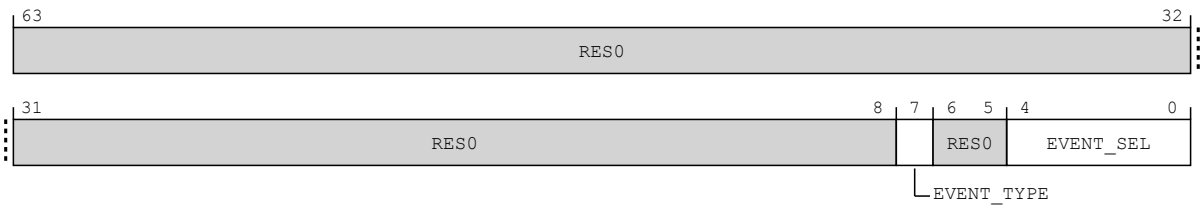

Bits [63:8]

Reserved, RES0.

EVENT_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCTSCTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCTSCTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCTSCTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [6:5]

Reserved, RES0.

EVENT_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

Defines the selected Resource Selector or pair of Resource Selectors. TRCTSCTLR.EVENT.TYPE controls whether TRCTSCTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRCTSCTLR

Must be programmed if TRCCONFIGR.TS == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCTSCTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().TSSIZE != '00000')
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCTSCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCTSCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCTSCTLR();
    end;
end;
```

MSR TRCTSCTLR, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && TRCIDR0().TSSIZE != '00000')
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCTSCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCTSCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCTSCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.66 TRCVICTLR, Trace ViewInst Main Control Register

The TRCVICTLR characteristics are:

Purpose

Controls instruction trace filtering.

Configuration

AArch64 System register TRCVICTLR bits [31:0] are architecturally mapped to External register TRCVICTLR[31:0].

This register is present only when FEAT_ETE is implemented and System register access to the trace unit registers is implemented. Otherwise, direct accesses to TRCVICTLR are UNDEFINED.

Attributes

TRCVICTLR is a 64-bit register.

Field descriptions

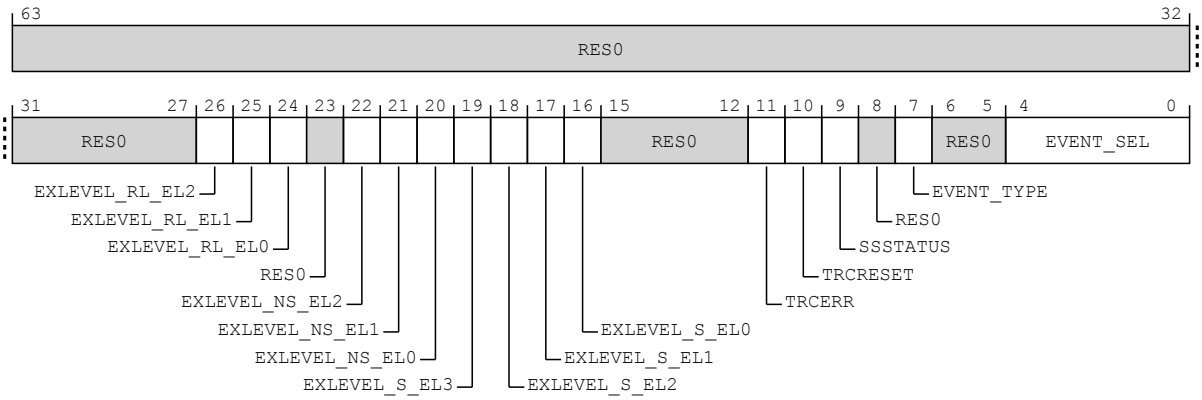

Bits [63:27]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [26]

When FEAT_RME is implemented:

Filter instruction trace for EL2 in Realm state.

EXLEVEL_RL_EL2	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit generates instruction trace for EL2 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit does not generate instruction trace for EL2 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit does not generate instruction trace for EL2 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit generates instruction trace for EL2 in Realm state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL1, bit [25]

When FEAT_RME is implemented:

Filter instruction trace for EL1 in Realm state.

EXLEVEL_RL_EL1	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit generates instruction trace for EL1 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit does not generate instruction trace for EL1 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit does not generate instruction trace for EL1 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit generates instruction trace for EL1 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [24]

When FEAT_RME is implemented:

Filter instruction trace for EL0 in Realm state.

EXLEVEL_RL_EL0	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit generates instruction trace for EL0 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit does not generate instruction trace for EL0 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit does not generate instruction trace for EL0 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit generates instruction trace for EL0 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [23]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]

When Non-secure EL2 is implemented:

Filter instruction trace for EL2 in Non-secure state.

EXLEVEL_NS_EL2	Meaning
0b0	The trace unit generates instruction trace for EL2 in Non-secure state.
0b1	The trace unit does not generate instruction trace for EL2 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [21]

When Non-secure EL1 is implemented:

Filter instruction trace for EL1 in Non-secure state.

EXLEVEL_NS_EL1	Meaning
0b0	The trace unit generates instruction trace for EL1 in Non-secure state.
0b1	The trace unit does not generate instruction trace for EL1 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL0, bit [20]

When Non-secure EL0 is implemented:

Filter instruction trace for EL0 in Non-secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL0	Meaning
0b0	The trace unit generates instruction trace for EL0 in Non-secure state.
0b1	The trace unit does not generate instruction trace for EL0 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL3, bit [19]

When EL3 is implemented:

Filter instruction trace for EL3.

EXLEVEL_S_EL3	Meaning
0b0	The trace unit generates instruction trace for EL3.
0b1	The trace unit does not generate instruction trace for EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL2, bit [18]

When Secure EL2 is implemented:

Filter instruction trace for EL2 in Secure state.

EXLEVEL_S_EL2	Meaning
0b0	The trace unit generates instruction trace for EL2 in Secure state.
0b1	The trace unit does not generate instruction trace for EL2 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_S_EL1, bit [17]

When Secure EL1 is implemented:

Filter instruction trace for EL1 in Secure state.

EXLEVEL_S_EL1	Meaning
0b0	The trace unit generates instruction trace for EL1 in Secure state.
0b1	The trace unit does not generate instruction trace for EL1 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL0, bit [16]

When Secure EL0 is implemented:

Filter instruction trace for EL0 in Secure state.

EXLEVEL_S_EL0	Meaning
0b0	The trace unit generates instruction trace for EL0 in Secure state.
0b1	The trace unit does not generate instruction trace for EL0 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [15:12]

Reserved, RES0.

TRCERR, bit [11]

When TRCIDR3.TRCERR == '1':

Controls the forced tracing of System Error exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCERR	Meaning
0b0	Forced tracing of System Error exceptions is disabled.
0b1	Forced tracing of System Error exceptions is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCRESET, bit [10]

Controls the forced tracing of PE Resets.

TRCRESET	Meaning
0b0	Forced tracing of PE Resets is disabled.
0b1	Forced tracing of PE Resets is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SSSTATUS, bit [9]

ViewInst start/stop function status.

SSSTATUS	Meaning
0b0	Stopped State. The ViewInst start/stop function is in the stopped state.
0b1	Started State. The ViewInst start/stop function is in the started state.

Before software enables the trace unit, it must write to this field to set the initial state of the ViewInst start/stop function. If the ViewInst start/stop function is not used then set this field to 1. Arm recommends that the value of this field is set before each trace session begins.

If the trace unit becomes disabled while a start point or stop point is still speculative, then the value of TRCVICTLR.SSSTATUS is UNKNOWN and might represent the result of a speculative start point or stop point.

If software which is running on the PE being traced disables the trace unit, either by clearing TRCPRGCTLR.EN or locking the OS Lock, Arm recommends that a DSB and an ISB instruction are executed before disabling the trace unit to prevent any start points or stop points being speculative at the point of disabling the trace unit. This procedure assumes that all start points or stop points occur before the barrier instructions are executed. The procedure does not guarantee that there are no speculative start points or stop points when disabling, although it helps minimize the probability.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access to this field is RES1 if all of the following are true:
  - TRCIDR4.NUMACPAIRS == ‘0000’
  - TRCIDR4.NUMPC == ‘0000’
- Otherwise, access to this field is RW.

Bit [8]

Reserved, RES0.

EVENT_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCVICTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCVICTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCVICTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

Bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

EVENT_SEL, bits [4:0]

Defines the selected Resource Selector or pair of Resource Selectors. TRCVICTLR.EVENT.TYPE controls whether TRCVICTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

When TRCIDR4.NUMRSPAIR == '0000':

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, bits [4:0]

This field is reserved:

- Bits [4:1] are RES0.
- Bit [0] is RES1.

Otherwise:

Reserved, RES0.

Accessing TRCVICTLR

Must be programmed.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVICTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRCVICTLR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVICTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVICTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVICTLR();
    end;
end;
```

```
MSR TRCVICTLR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0000	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRCVICTLR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVICTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVICTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVICTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.67 TRCVIIECTLR, Trace ViewInst Include/Exclude Control Register

The TRCVIIECTLR characteristics are:

Purpose

Use this to select, or read, the Address Range Comparators for the ViewInst include/exclude function.

Configuration

AArch64 System register TRCVIIECTLR bits [31:0] are architecturally mapped to External register TRCVIIECTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVIIECTLR are UNDEFINED.

Attributes

TRCVIIECTLR is a 64-bit register.

Field descriptions

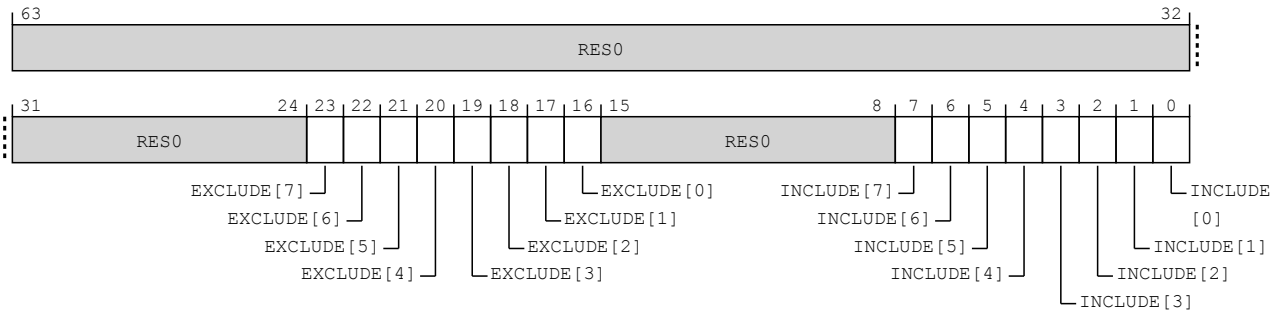

Bits [63:24]

Reserved, RES0.

EXCLUDE[<m>] , bits [m+16], for m = 7 to 0

Exclude Address Range Comparator <m>. Selects whether Address Range Comparator <m> is in use with the ViewInst exclude function.

EXCLUDE[<m>]	Meaning
0b0	The address range that Address Range Comparator <m> defines, is not selected for the ViewInst exclude function.
0b1	The address range that Address Range Comparator <m> defines, is selected for the ViewInst exclude function.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, access to this field is RW.

Bits [15:8]

Reserved, RES0.

INCLUDE[<m>] , bits [m], for m = 7 to 0

Include Address Range Comparator <m>.

Selects whether Address Range Comparator <m> is in use with the ViewInst include function.

Selecting no comparators for the ViewInst include function indicates that all instructions are included by default.

The ViewInst exclude function then indicates which ranges are excluded.

INCLUDE[<m>]	Meaning
0b0	The address range that Address Range Comparator <m> defines, is not selected for the ViewInst include function.
0b1	The address range that Address Range Comparator <m> defines, is selected for the ViewInst include function.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVIIECTLR

Must be programmed if TRCIDR4.NUMACPAIRS > 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVIIECTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMACPAIRS) >
     ↪0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().TRC == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCVIIECTLR();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
            Undefined();
        elsif CPTR_EL2().TTA == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCVIIECTLR();
        end;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3().TTA == '1' then
            AArch64_SystemAccessTrap(EL3, 0x18);
        elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
             ↪EDSCR2().TTA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            X{64}(t) = TRCVIIECTLR();
        end;
    end;
```

MSR TRCVIIECTLR, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0001	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMACPAIRS) >
     ↪0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVIIECTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVIIECTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVIIECTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.68 TRCVIPCSSCTLR, Trace ViewInst Start/Stop PE Comparator Control Register

The TRCVIPCSSCTLR characteristics are:

Purpose

Use this to select, or read, which PE Comparator Inputs can control the ViewInst start/stop function.

Configuration

AArch64 System register TRCVIPCSSCTLR bits [31:0] are architecturally mapped to External register TRCVIPCSSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMPC) > 0. Otherwise, direct accesses to TRCVIPCSSCTLR are UNDEFINED.

Attributes

TRCVIPCSSCTLR is a 64-bit register.

Field descriptions

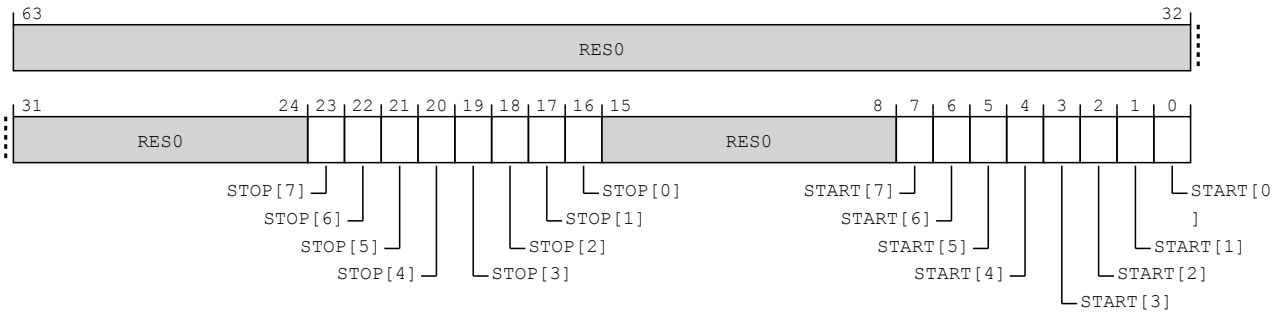

Bits [63:24]

Reserved, RES0.

STOP[<m>] , bits [m+16], for m = 7 to 0

Selects whether PE Comparator Input <m> is in use with the ViewInst start/stop function for the purpose of stopping trace.

STOP[<m>]	Meaning
0b0	The PE Comparator Input <m> is not selected as a stop resource.
0b1	The PE Comparator Input <m> is selected as a stop resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMPC), access to this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, access to this field is RW.

Bits [15:8]

Reserved, RES0.

START[<m>] , bits [m], for m = 7 to 0

Selects whether PE Comparator Input <m> is in use with the ViewInst start/stop function for the purpose of starting trace.

START[<m>]	Meaning
0b0	The PE Comparator Input <m> is not selected as a start resource.
0b1	The PE Comparator Input <m> is selected as a start resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMPC), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVIPCSSCTLR

Must be programmed if TRCIDR4.NUMPC != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVIPCSSCTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMPC) > 0)
    then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVIPCSSCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVIPCSSCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVIPCSSCTLR();
    end;
end;
```

```
MSR TRCVIPCSSCTLR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMPC) > 0)
     ↪then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVIPCSSCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVIPCSSCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVIPCSSCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.69 TRCVISSCTLR, Trace ViewInst Start/Stop Control Register

The TRCVISSCTLR characteristics are:

Purpose

Use this to select, or read, the Single Address Comparators for the ViewInst start/stop function.

Configuration

AArch64 System register TRCVISSCTLR bits [31:0] are architecturally mapped to External register TRCVISSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVISSCTLR are UNDEFINED.

Attributes

TRCVISSCTLR is a 64-bit register.

Field descriptions

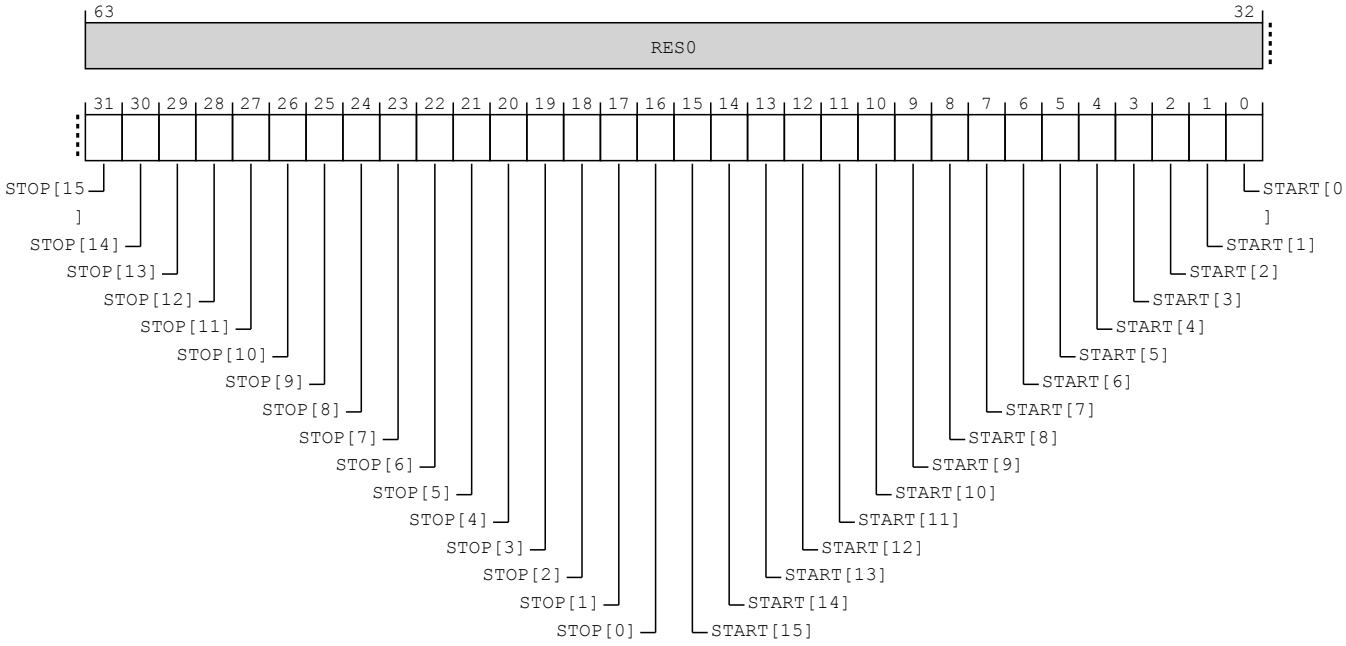

Bits [63:32]

Reserved, RES0.

STOP[<m>] , bits [m+16], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function for the purpose of stopping trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STOP[<m>]	Meaning
0b0	The Single Address Comparator <m> is not selected as a stop resource.
0b1	The Single Address Comparator <m> is selected as a stop resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

START[<m>] , bits [m], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function for the purpose of starting trace.

START[<m>]	Meaning
0b0	The Single Address Comparator <m> is not selected as a start resource.
0b1	The Single Address Comparator <m> is selected as a start resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVISSCTLR

Must be programmed if TRCIDR4.NUMACPAIRS > 0b0000.

For any 2 comparators selected for the ViewInst start/stop function, the comparator containing the lower address must be a lower numbered comparator.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVISSCTLR
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0010	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMACPAIRS) >
     ↪0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVISSCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVISSCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVISSCTLR();
    end;
end;
```

```
MSR TRCVISSCTLR, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0000	0b0010	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMACPAIRS) >
     ↪0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVISSCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVISSCTLR() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVISSCTLR() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.70 TRCVMIDCCTLR0, Trace Virtual Context Identifier Comparator Control Register 0

The TRCVMIDCCTLR0 characteristics are:

Purpose

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=0-3.

Configuration

AArch64 System register TRCVMIDCCTLR0 bits [31:0] are architecturally mapped to External register TRCVMIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, UInt(TRCIDR4.NUMVMIDC) > `0x0`, and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR0 are UNDEFINED.

Attributes

TRCVMIDCCTLR0 is a 64-bit register.

Field descriptions

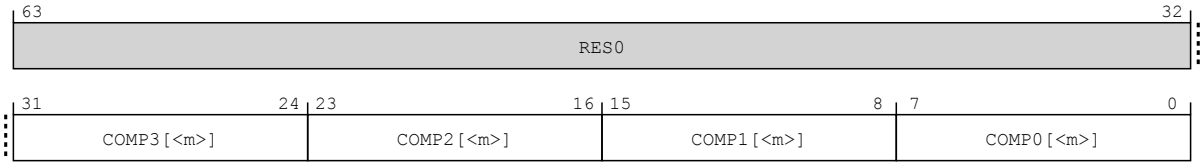

Bits [63:32]

Reserved, RES0.

COMP3[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 3:

TRCVMIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR3. Each bit in this field corresponds to a byte in TRCVMIDCVR3.

COMP3[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

COMP2[<m>] , bits [m+16], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 2:

TRCVMIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR2. Each bit in this field corresponds to a byte in TRCVMIDCVR2.

COMP2[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP1[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 1:

TRCVMIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR1. Each bit in this field corresponds to a byte in TRCVMIDCVR1.

COMP1[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP0[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 0:

TRCVMIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR0. Each bit in this field corresponds to a byte in TRCVMIDCVR0.

COMP0[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCVMIDCCTLR0

If software uses the TRCVMIDCVR<n> registers, where n=0-3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVMIDCCTLR0
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMVMIDC) >
     ↪0x0 && UInt(TRCIDR2().VMIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCCTLR0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCCTLR0();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCCTLR0();
    end;
end;
```

```
MSR TRCVMIDCCTLR0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMVMIDC) >
     ↪0x0 && UInt(TRCIDR2().VMIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCCTLR0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCCTLR0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCCTLR0() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.71 TRCVMIDCCTLR1, Trace Virtual Context Identifier Comparator Control Register 1

The TRCVMIDCCTLR1 characteristics are:

Purpose

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=4-7.

Configuration

AArch64 System register TRCVMIDCCTLR1 bits [31:0] are architecturally mapped to External register TRCVMIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, UInt(TRCIDR4.NUMVMIDC) > `0x4`, and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR1 are UNDEFINED.

Attributes

TRCVMIDCCTLR1 is a 64-bit register.

Field descriptions

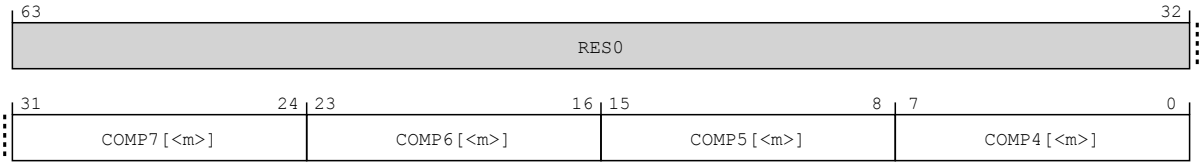

Bits [63:32]

Reserved, RES0.

COMP7[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 7:

TRCVMIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR7. Each bit in this field corresponds to a byte in TRCVMIDCVR7.

COMP7[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

COMP6[<m>] , bits [m+16], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 6:

TRCVMIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR6. Each bit in this field corresponds to a byte in TRCVMIDCVR6.

COMP6[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP5[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 5:

TRCVMIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR5. Each bit in this field corresponds to a byte in TRCVMIDCVR5.

COMP5[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP4[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 4:

TRCVMIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR4. Each bit in this field corresponds to a byte in TRCVMIDCVR4.

COMP4[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCVMIDCCTLR1

If software uses the TRCVMIDCVR<n> registers, where n=4-7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVMIDCCTLR1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMVMIDC) >
     ↪0x4 && UInt(TRCIDR2().VMIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCCTLR1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCCTLR1();
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCCTLR1();
    end;
end;
```

```
MSR TRCVMIDCCTLR1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_ETE) && IsFeatureImplemented(FEAT_TRC_SR) && UInt(TRCIDR4().NUMVMIDC) >
     ↪0x4 && UInt(TRCIDR2().VMIDSIZE) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCCTLR1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCCTLR1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCCTLR1() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.72 TRCVMIDCVR<n>, Trace Virtual Context Identifier Comparator Value Register <n>, n = 0 - 7

The TRCVMIDCVR<n> characteristics are:

Purpose

Contains the Virtual Context Identifier Comparator value.

Configuration

AArch64 System register TRCVMIDCVR<n> bits [63:0] are architecturally mapped to External register TRCVMIDCVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, System register access to the trace unit registers is implemented, and UInt(TRCIDR4.NUMVMIDC) > n. Otherwise, direct accesses to TRCVMIDCVR<n> are UNDEFINED.

Attributes

TRCVMIDCVR<n> is a 64-bit register.

Field descriptions

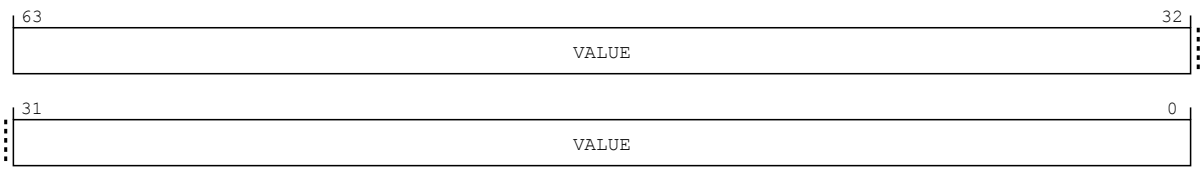

VALUE, bits [63:0]

Virtual context identifier value. The width of this field is indicated by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Virtual context identifier is zero until the PE updates the Virtual context identifier .

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCVMIDCVR<n>

Must be programmed if any of the following are true:

- TRCRSCTLR<a>.GROUP == `0b0111` and TRCRSCTLR<a>.VMID[n] == 1.
- TRCACATR<a>.CONTEXTTYPE == `0b10` or `0b11` and TRCACATR<a>.CONTEXT == n.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCVMIDCVR<m> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
`0b10`	`0b001`	`0b0011`	m[2:0]:`0b0`	`0b001`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:1]);

if m >= NUM_TRACE_VIRTUAL_CONTEXT_IDENTIFIER_COMPARATORS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X{64}(t) = TRCVMIDCVR(m);
    end;
end;
```

```
MSR TRCVMIDCVR<m>, <Xt> ; Where m = 0-7
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b0011	m[2:0]:0b0	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:1]);

if m >= NUM_TRACE_VIRTUAL_CONTEXT_IDENTIFIER_COMPARATORS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPACR_EL1().TTA == '1' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRC == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TTA == '1' then
        Undefined();
    elsif CPTR_EL2().TTA == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TTA == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCVR(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3().TTA == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) && OSLSR_EL1().OSLK == '0' && HaltingAllowed() &&
         ↪EDSCR2().TTA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        TRCVMIDCVR(m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.73 TRFCR_EL1, Trace Filter Control Register (EL1)

The TRFCR_EL1 characteristics are:

Purpose

Provides EL1 controls for Trace.

Configuration

AArch64 System register TRFCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TRFCR[31:0].

This register is present only when FEAT_TRF is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TRFCR_EL1 are UNDEFINED.

Attributes

TRFCR_EL1 is a 64-bit register.

Field descriptions

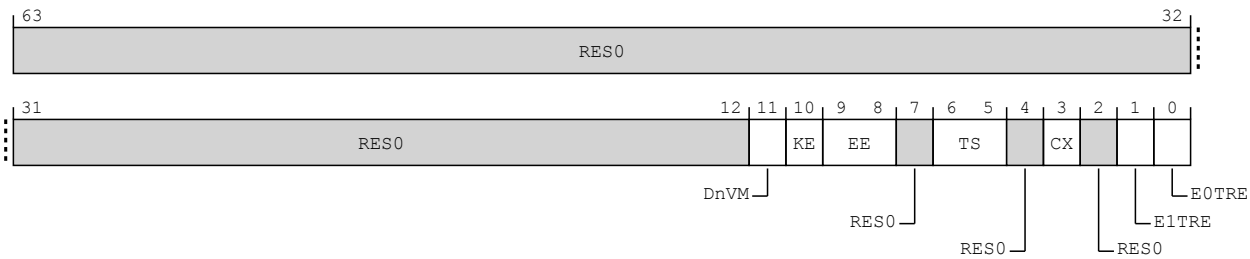

Bits [63:12]

Reserved, RES0.

DnVM, bit [11]

When FEAT_TRBEv1p1 is implemented and FEAT_NV is implemented:

Reserved for software use in nested virtualization. See also TRFCR_EL2.DnVM.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

KE, bit [10]

When FEAT_TRBE_EXC is implemented:

Kernel exception enable for TRBE Profiling exceptions taken to EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

KE	Meaning
0b0	TRBE Profiling exceptions taken to EL1 are always masked at EL1.
0b1	Enabled TRBE Profiling exceptions taken to EL1 are masked at EL1 when PSTATE.PM is 1 and unmasked when PSTATE.PM is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EE, bits [9:8]

When FEAT_TRBE_EXC is implemented:

Exception Enable.

EE	Meaning	Applies when
0b00	Disabled. TRBE Profiling exceptions for EL1 are disabled. All of the following apply: • Unless enabled by a higher Exception level, TRBE Profiling exceptions are not generated. • TRBSR_EL1.IRQ drives the interrupt request signal **TRBIRQ**. • Accesses to TRBSR_EL1 at EL1 ignore the value of HCR_EL2.NV1.	
0b01	Reserved for software use in nested virtualization. Behaves as 0b00 for the purpose of controlling the TRBE Profiling exception and interrupt request signal **TRBIRQ**, and as 0b11 for the purpose of accesses to TRBSR_EL1.	FEAT_NV is implemented
0b10	Reserved for software use in nested virtualization. Behaves as 0b11 for the purposes of controlling the TRBE Profiling exception and interrupt request signal **TRBIRQ**, and accesses to TRBSR_EL1.	FEAT_NV is implemented
0b11	Enabled. TRBE Profiling exceptions for EL1 are enabled, as follows: • All trace buffer management events are recorded in TRBSR_EL1, unless they are configured to be recorded in TRBSR_EL3 by MDCR_EL3.TRBEE or TRBSR_EL2 by TRFCR_EL2.EE. • TRBE Profiling exceptions are generated and taken to EL1 when unmasked and TRBSR_EL1.IRQ is 1, unless the Effective value of HCR_EL2.TGE is 1, in which case the exception is taken to EL2. • The interrupt request signal **TRBIRQ** is not asserted.	

For more information on the values reserved for software use in nested virtualization, see TRFCR_EL2.EE.

If the Effective value of TRFCR_EL2.EE is 0b00, then the Effective value of TRFCR_EL1.EE is 0b00.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

TS	Meaning	Applies when
`0b00`	Reserved for software use in nested virtualization. Behaves as `0b01`. See also TRFCR_EL2.TS,	FEAT_NV2p1 is implemented
`0b01`	Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF_EL2.	
`0b10`	Guest physical timestamp. The traced timestamp is the physical counter value minus a physical offset. If any of the following are true, then the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • SCR_EL3.ECVEn == 0. • CNTHCTL_EL2.ECV == 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented
`0b11`	Physical timestamp. The traced timestamp is the physical counter value.	

All other values are reserved.

If any of the following are true, then this field is ignored by the PE:

- EL2 is implemented and TRFCR_EL2.TS is not `0b00`.
- `SelfHostedTraceEnabled()` == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [4]

Reserved, RES0.

CX, bit [3]

When FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization. See also TRFCR_EL2.CX.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [2]

Reserved, RES0.

E1TRE, bit [1]

EL1 Trace Enable.

E1TRE	Meaning
0b0	Trace is prohibited at EL1.
0b1	Trace is allowed at EL1.

If any of the following are true, then this field is ignored by the PE:

- `SelfHostedTraceEnabled()` == FALSE.
- The Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

E0TRE, bit [0]

EL0 Trace Enable.

E0TRE	Meaning
0b0	Trace is prohibited at EL0.
0b1	Trace is allowed at EL0.

If any of the following are true, then this field is ignored by the PE:

- `SelfHostedTraceEnabled()` == FALSE.
- The Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRFCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif EL2Enabled() && MDCR_EL2().TTRF == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x880);
    else
        X{64}(t) = TRFCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = TRFCR_EL2();
    else
        X{64}(t) = TRFCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TRFCR_EL1();
end;
```

```
MSR TRFCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().TRFCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TTRF == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x880) = X{64}(t);
    else
        TRFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TRFCR_EL2() = X{64}(t);
    else
        TRFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TRFCR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TRFCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x880);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = TRFCR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = TRFCR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR TRFCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x880) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            TRFCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        TRFCR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.4.74 TRFCR_EL2, Trace Filter Control Register (EL2)

The TRFCR_EL2 characteristics are:

Purpose

Provides EL2 controls for Trace.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register TRFCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HTRFCR[31:0].

This register is present only when FEAT_TRF is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to TRFCR_EL2 are UNDEFINED.

Attributes

TRFCR_EL2 is a 64-bit register.

Field descriptions

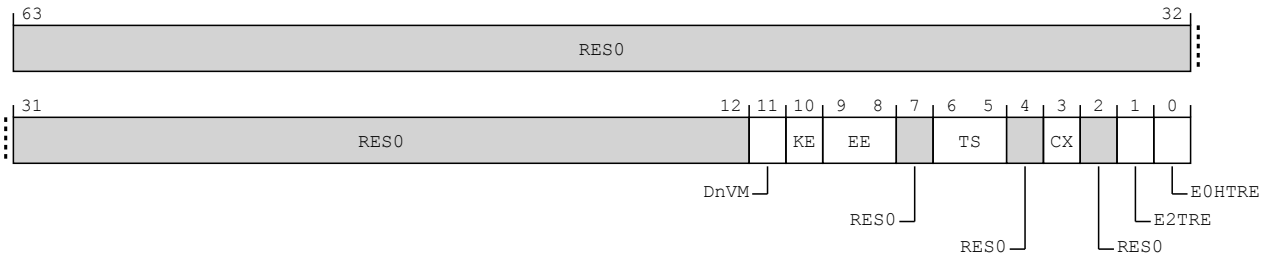

Bits [63:12]

Reserved, RES0.

DnVM, bit [11]

When FEAT_TRBEv1p1 is implemented:

Disable use of physical address trace buffer pointers.

DnVM	Meaning
0b0	Use of physical address trace buffer pointers is permitted.
0b1	Use of physical address trace buffer pointers is disabled. The PE behaves as if TRBLIMITR_EL1.nVM is 0.

If EL2 is disabled in the owning Security state, or the trace buffer owning Exception level is EL2, then the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

KE, bit [10]

When FEAT_TRBE_EXC is implemented:

Kernel exception enable for TRBE Profiling exceptions taken to EL2.

KE	Meaning
0b0	TRBE Profiling exceptions taken to EL2 are always masked at EL2.
0b1	Enabled TRBE Profiling exceptions taken to EL2 are masked at EL2 when PSTATE.PM is 1 and unmasked when PSTATE.PM is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EE, bits [9:8]

When FEAT_TRBE_EXC is implemented:

Exception Enable.

EE	Meaning
0b00	Disabled. TRBE Profiling exceptions for EL2 and EL1 are disabled. All of the following apply: • No trace buffer management events are recorded in TRBSR_EL2. • Unless enabled by a higher Exception level, TRBE Profiling exceptions are not generated. • TRBSR_EL1.IRQ drives the interrupt request signal **TRBIRQ**. • Accesses to TRBSR_EL1 at EL1 ignore the value of HCR_EL2.NV1 and accesses to TRBSR_EL1 at EL2 ignore the value of HCR_EL2.E2H.
0b01	Delegated. TRBE Profiling exceptions for EL2 are disabled, but might be enabled for EL1 by TRFCR_EL1.EE. All of the following apply: • No trace buffer management events are recorded in TRBSR_EL2. • TRBSR_EL2.IRQ is ignored and TRBE Profiling exceptions are not taken to EL2, other than for the case when the Effective value of HCR_EL2.TGE is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EE	Meaning

0b10	Enabled. TRBE Profiling exceptions for EL2 are enabled for trace buffer management events targeting EL2, as follows:
	• Trace buffer management events due to a fault on a write to the trace buffer that would generate a Data Abort exception taken to EL2 if generated by a store instruction executed at the owning Exception level are recorded in TRBSR_EL2, unless they are configured to be recorded in TRBSR_EL3 by MDCR_EL3.TRBEE.
	If the trace buffer owning Exception level is EL2, then this means any fault on a write to the trace buffer.
	If the trace buffer owning Exception level is EL1, then this means any of the following faults on a write to the trace buffer:
	  • Stage 2 faults.
	  • If HCR_EL2.TEA is 1, synchronous External aborts.
	  • If HCR_EL2.GPF is 1, Granule Protection Faults (GPFs).
	• Trace buffer management events due to Granule Protection Check faults other than GPFs on a write to the trace buffer are recorded in TRBSR_EL2, unless they are configured to be recorded in TRBSR_EL3 by MDCR_EL3.TRBEE.
	• TRBE Profiling exceptions are generated and taken to EL2 when unmasked and TRBSR_EL2.IRQ is 1.

0b11	Trap all. TRBE Profiling exceptions for EL2 are enabled for all trace buffer management events, as follows: • All trace buffer management events are recorded in TRBSR_EL2, unless they are configured to be recorded in TRBSR_EL3 by MDCR_EL3.TRBEE. • TRBE Profiling exceptions are generated and taken to EL2 when unmasked and TRBSR_EL2.IRQ is 1.

If the Effective value of MDCR_EL3.TRBEE is `0b00`, then the Effective value of TRFCR_EL2.EE is `0b00`. Otherwise, if EL2 is not implemented or the Effective value of SCR_EL3.{NS, EEL2} is {0, 0}, then the Effective value of TRFCR_EL2.EE is `0b01`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

TS	Meaning	Applies when
0b00	Timestamp controlled by TRFCR_EL1.TS or TRFCR.TS.	
0b01	Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF_EL2.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TS	Meaning	Applies when
0b10	Guest physical timestamp. The traced timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • SCR_EL3.ECVEn == 0. • CNTHCTL_EL2.ECV == 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented
0b11	Physical timestamp. The traced timestamp is the physical counter value.	

If `SelfHostedTraceEnabled()` == FALSE, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'00'`.

Bit [4]

Reserved, RES0.

CX, bit [3]

CONTEXTIDR_EL2 and VMID trace enable.

CX	Meaning
0b0	CONTEXTIDR_EL2 and VMID trace prohibited.
0b1	CONTEXTIDR_EL2 and VMID trace allowed.

If `SelfHostedTraceEnabled()` == FALSE, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bit [2]

Reserved, RES0.

E2TRE, bit [1]

EL2 Trace Enable.

E2TRE	Meaning
0b0	Trace is prohibited at EL2.
0b1	Trace is allowed at EL2.

If `SelfHostedTraceEnabled()` == FALSE, then this field is ignored by the PE.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

E0HTRE, bit [0]

EL0 Trace Enable.

E0HTRE	Meaning
0b0	Trace is prohibited at EL0.
0b1	Trace is allowed at EL0.

If any of the following are true, then this field is ignored by the PE:

- `SelfHostedTraceEnabled`() == FALSE.
- The Effective value of HCR_EL2.TGE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRFCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRFCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = TRFCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = TRFCR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSR TRFCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        TRFCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    TRFCR_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, TRFCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif EL2Enabled() && MDCR_EL2().TTRF == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x880);
    else
        X{64}(t) = TRFCR_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif ELIsInHost(EL2) then
            X{64}(t) = TRFCR_EL2();
        else
            X{64}(t) = TRFCR_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = TRFCR_EL1();
    end;
```

When FEAT_VHE is implemented MSR TRFCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_TRF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HDFGWTR_EL2().TRFCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TTRF == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x880) = X{64}(t);
    else
        TRFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        TRFCR_EL2() = X{64}(t);
    else
        TRFCR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    TRFCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5 Performance Monitors registers

This section lists the Performance Monitoring registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.1 PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configuration

AArch64 System register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCCFILTR[31:0].

When FEAT_PMUv3_TH is implemented, or FEAT_PMUv3p8 is implemented, or FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_SME is implemented, AArch64 System register PMCCFILTR_EL0 bits [63:32] are architecturally mapped to External register PMCCFILTR_EL0[63:32].

AArch64 System register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to External register PMCCFILTR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are UNDEFINED.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions

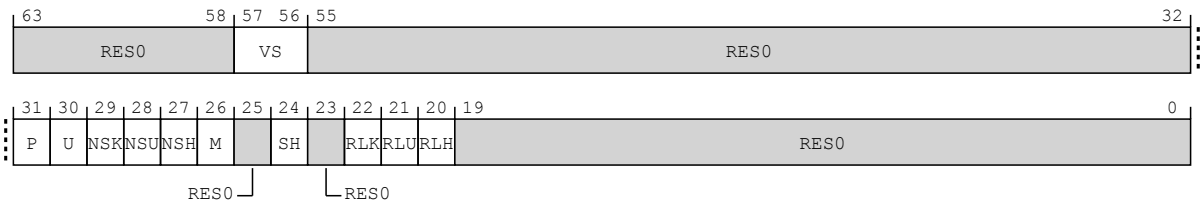

Bits [63:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting cycles in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of cycles.
0b01	The PE does not count cycles in Streaming SVE mode.
0b10	The PE does not count cycles in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting cycles in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR_EL0.NSK.

If FEAT_RME is implemented, then counting cycles in Realm EL1 is further controlled by PMCCFILTR_EL0.RLK.

If EL3 is implemented, then counting cycles at EL3 is further controlled by PMCCFILTR_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting cycles in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR_EL0.NSU.

If FEAT_RME is implemented, then counting cycles in Realm EL0 is further controlled by PMCCFILTR_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting cycles in Non-secure EL1. If PMCCFILTR_EL0.NSK is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL1.

NSK	Meaning
0b0	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in Non-secure EL1.
0b1	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in Non-secure EL1. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting cycles in Non-secure EL0. If PMCCFILTR_EL0.NSU is not equal to PMCCFILTR_EL0.U, then the PE does not count cycles in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL0.

NSU	Meaning
0b0	When PMCCFILTR_EL0.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.U == 1, the PE does not count cycles in Non-secure EL0.
0b1	When PMCCFILTR_EL0.U == 0, the PE does not count cycles in Non-secure EL0. When PMCCFILTR_EL0.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
0b0	The PE does not count cycles in EL2.
0b1	This mechanism has no effect on filtering of cycles.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting cycles in Secure EL2 is further controlled by PMCCFILTR_EL0.SH.

If FEAT_RME is implemented, then counting cycles in Realm EL2 is further controlled by PMCCFILTR_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

EL3 filtering. Controls counting cycles in EL3. If PMCCFILTR_EL0.M is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in EL3. Otherwise, this mechanism has no effect on filtering of cycles in EL3.

M	Meaning
0b0	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in EL3.
0b1	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in EL3. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting cycles in Secure EL2. If PMCCFILTR_EL0.SH is equal to PMCCFILTR_EL0.NSH, then the PE does not count cycles in Secure EL2. Otherwise, this mechanism has no effect on filtering of cycles in Secure EL2.

SH	Meaning
0b0	When PMCCFILTR_EL0.NSH == 0, the PE does not count cycles in Secure EL2. When PMCCFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of cycles.
0b1	When PMCCFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.NSH == 1, the PE does not count cycles in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting cycles in Realm EL1. If PMCCFILTR_EL0.RLK is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in Realm EL1. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL1.

RLK	Meaning
0b0	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in Realm EL1.
0b1	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in Realm EL1. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting cycles in Realm EL0. If PMCCFILTR_EL0.RLU is not equal to PMCCFILTR_EL0.U, then the PE does not count cycles in Realm EL0. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL0.

RLU	Meaning
0b0	When PMCCFILTR_EL0.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.U == 1, the PE does not count cycles in Realm EL0.
0b1	When PMCCFILTR_EL0.U == 0, the PE does not count cycles in Realm EL0. When PMCCFILTR_EL0.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting cycles in Realm EL2. If PMCCFILTR_EL0.RLH is equal to PMCCFILTR_EL0.NSH, then the PE does not count cycles in Realm EL2. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL2.

RLH	Meaning
0b0	When PMCCFILTR_EL0.NSH == 0, the PE does not count cycles in Realm EL2. When PMCCFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of cycles.
0b1	When PMCCFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.NSH == 1, the PE does not count cycles in Realm EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

PMCCFILTR_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to `0b11111`.

Permitted reads and writes of PMCCFILTR_EL0 are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.C is 0.

Permitted writes of PMCCFILTR_EL0 are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.{UEN,CR} is {1,1}.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCCFILTR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMCCFILTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().C == '0' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = PMCCFILTR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMCCFILTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCCFILTR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCCFILTR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCCFILTR_EL0();
end;
```

MSR PMCCFILTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMCCFILTR_EL0 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().C == '0' ||
     ↪PMUSERENR_EL0().CR == '1') then
        return;
    else
        PMCCFILTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HDFGWTR_EL2().PMCCFILTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCCFILTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCCFILTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCCFILTR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.2 PMCCNTR_EL0, Performance Monitors Cycle Count Register

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See 'Time as measured by the Performance Monitors cycle counter' for more information.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configuration

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR_EL0 continues to increment when clocks are stopped by WFI and WFE instructions.

AArch64 System register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch32 System register PMCCNTR[63:0].

AArch64 System register PMCCNTR_EL0 bits [63:0] are architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are UNDEFINED.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions

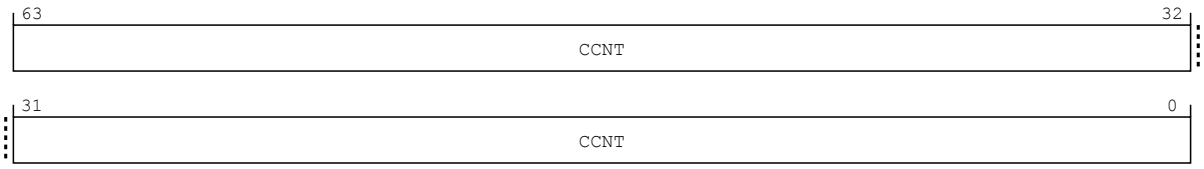

CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR_EL0

Permitted reads and writes of PMCCNTR_EL0 are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.C is 0.

Permitted writes of PMCCNTR_EL0 are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.{UEN,CR} is {1,1}.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCCNTR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,CR,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[CR,EN] == '00') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().C == '0' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = PMCCNTR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCCNTR_EL0();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCCNTR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCCNTR_EL0();
end;
```

```
MSR PMCCNTR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().C == '0' ||
         ↪PMUSERENR_EL0().CR == '1') then
        return;
    else
        PMCCNTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCCNTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCCNTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCCNTR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.3 PMCCNTSVR_EL1, Performance Monitors Cycle Count Saved Value Register

The PMCCNTSVR_EL1 characteristics are:

Purpose

Captures the PMU Cycle counter, PMCCNTR_EL0.

Configuration

AArch64 System register PMCCNTSVR_EL1 bits [63:0] are architecturally mapped to External register PMCCNTSVR_EL1[63:0].

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCCNTSVR_EL1 are UNDEFINED.

Attributes

PMCCNTSVR_EL1 is a 64-bit register.

Field descriptions

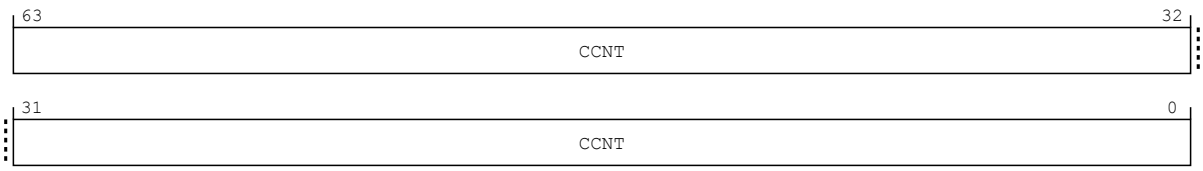

CCNT, bits [63:0]

Sampled Cycle Count. The value of PMCCNTR_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTSVR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCCNTSVR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1110	0b1011	0b111

```
if !(IsFeatureImplemented(FEAT_PMUv3_SS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMSSDATA == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCCNTSVR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCCNTSVR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCCNTSVR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.4 PMCEID0_EL0, Performance Monitors Common Event Identification Register 0

The PMCEID0_EL0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the ranges `0x0000` to `0x001F` and `0x4000` to `0x401F`.

For more information about the Common events and the use of the PMCEID<n>_EL0 registers see 'The PMU event number space and common events'.

Configuration

AArch64 System register PMCEID0_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCEID0[31:0].

AArch64 System register PMCEID0_EL0 bits [63:32] are architecturally mapped to AArch32 System register PMCEID2[31:0].

AArch64 System register PMCEID0_EL0 bits [31:0] are architecturally mapped to External register PMCEID0[31:0].

AArch64 System register PMCEID0_EL0 bits [63:32] are architecturally mapped to External register PMCEID2[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCEID0_EL0 are UNDEFINED.

Attributes

PMCEID0_EL0 is a 64-bit register.

Field descriptions

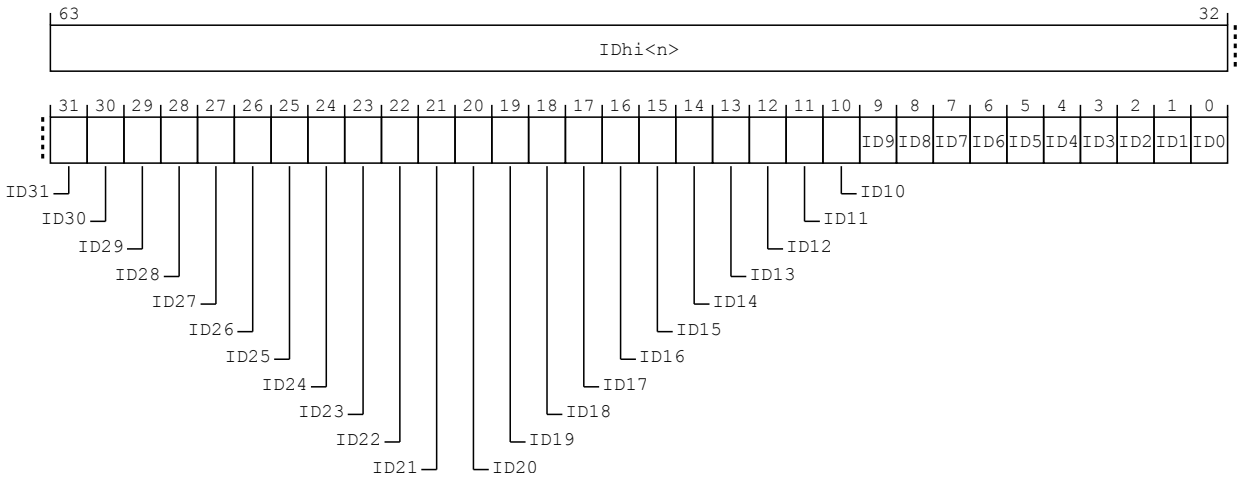

IDhi<n> , bits [n+32], for n = 31 to 0

When FEAT_PMUv3p1 is implemented:

IDhi[n] corresponds to Common event (`0x4000` + n).

For each bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDhi<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Otherwise:

Reserved, RES0.

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Accessing PMCEID0_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCEID0_EL0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b110

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().TID == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCEID0_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCEID0_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCEID0_EL0();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCEID0_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.5 PMCEID1_EL0, Performance Monitors Common Event Identification Register 1

The PMCEID1_EL0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the ranges `0x0020` to `0x003F` and `0x4020` to `0x403F`.

For more information about the Common events and the use of the PMCEID<n>_EL0 registers see 'The PMU event number space and common events'.

Configuration

AArch64 System register PMCEID1_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCEID1[31:0].

AArch64 System register PMCEID1_EL0 bits [63:32] are architecturally mapped to AArch32 System register PMCEID3[31:0].

AArch64 System register PMCEID1_EL0 bits [31:0] are architecturally mapped to External register PMCEID1[31:0].

AArch64 System register PMCEID1_EL0 bits [63:32] are architecturally mapped to External register PMCEID3[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCEID1_EL0 are UNDEFINED.

Attributes

PMCEID1_EL0 is a 64-bit register.

Field descriptions

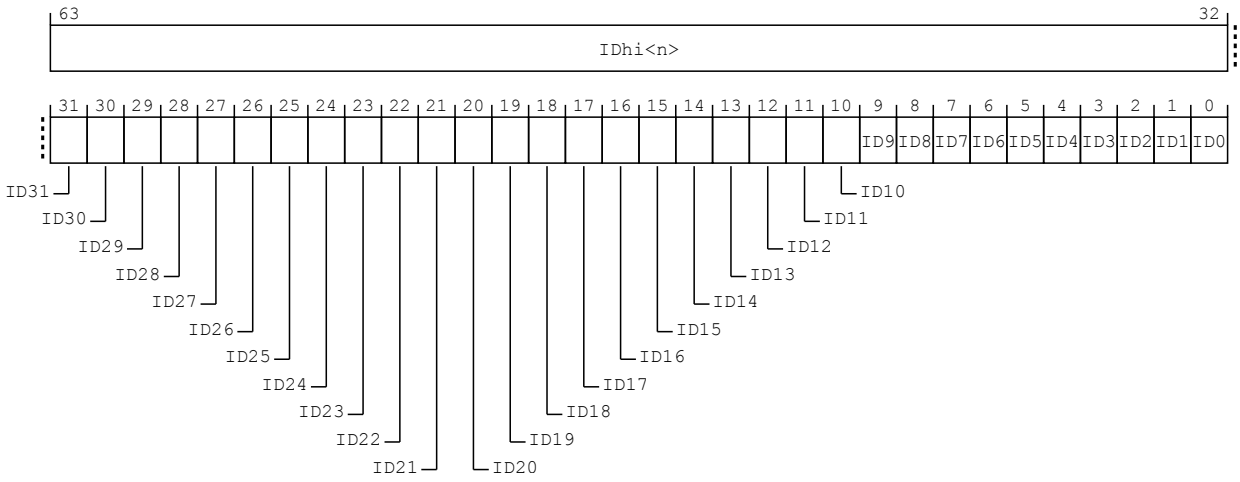

IDhi<n> , bits [n+32], for n = 31 to 0

When FEAT_PMUv3p1 is implemented:

IDhi[n] corresponds to Common event (`0x4020` + n).

For each bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDhi<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Otherwise:

Reserved, RES0.

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (0x0020 + n).

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Accessing PMCEID1_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCEID1_EL0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b111

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().TID == '1' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCEID1_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCEID1_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCEID1_EL0();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCEID1_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.6 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which counters are enabled.

Configuration

AArch64 System register PMCNTENCLR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[63:0].

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMCNTENCLR_EL0 bits [63:32] are architecturally mapped to External register PMCNTENCLR_EL0[63:32].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMCNTENCLR_EL0 bits [63:32] are architecturally mapped to External register PMCNTENSET_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are UNDEFINED.

Attributes

PMCNTENCLR_EL0 is a 64-bit register.

Field descriptions

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 disable. On writes, allows software to disable PMICNTR_EL0. On reads, returns the PMICNTR_EL0 enable status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F0	Meaning
`0b0`	PMICNTR_EL0 disabled.
`0b1`	PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - EL0 is the current Exception level
  - Any of the following are true:
    - PMUSERENR_EL0.UEN == ‘0’
    - PMUACR_EL1.F0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - All of the following are true:
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
      - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL0 is the current Exception level
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
  - PMUSERENR_EL0.IR == ‘1’
- Access to this field is WO/RAZ if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PSTATE.EL IN {EL1, EL0}
- [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
- Any of the following are true:
  - All of the following are true:
    - EL3 is implemented
    - SCR_EL3.FGTEn2 == ‘0’
  - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - EL0 is the current Exception level
  - PMUSERENR_EL0.IR == ‘1’
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.C == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == ‘11’
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.P<m> == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == '11'
- Otherwise, access to this field is W1C.

Accessing PMCNTENCLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCNTENCLR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b010

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCNTENCLR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCNTENCLR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCNTENCLR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCNTENCLR_EL0();
end;
```

```
MSR PMCNTENCLR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b010

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCNTENCLR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCNTENCLR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCNTENCLR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCNTENCLR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.7 PMCNTENSET_EL0, Performance Monitors Count Enable Set Register

The PMCNTENSET_EL0 characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which counters are enabled.

Configuration

AArch64 System register PMCNTENSET_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[63:0].

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMCNTENSET_EL0 bits [63:32] are architecturally mapped to External register PMCNTENSET_EL0[63:32].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMCNTENSET_EL0 bits [63:32] are architecturally mapped to External register PMCNTENCLR_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are UNDEFINED.

Attributes

PMCNTENSET_EL0 is a 64-bit register.

Field descriptions

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 enable. On writes, allows software to enable PMICNTR_EL0. On reads, returns the PMICNTR_EL0 enable status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - EL0 is the current Exception level
  - Any of the following are true:
    - PMUSERENR_EL0.UEN == ‘0’
    - PMUACR_EL1.F0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - All of the following are true:
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
      - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL0 is the current Exception level
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
  - PMUSERENR_EL0.IR == ‘1’
- Access to this field is WO/RAZ if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PSTATE.EL IN {EL1, EL0}
- [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
- Any of the following are true:
  - All of the following are true:
    - EL3 is implemented
    - SCR_EL3.FGTEn2 == ‘0’
  - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - EL0 is the current Exception level
  - PMUSERENR_EL0.IR == ‘1’
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.C == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == ‘11’
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.P<m> == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == ‘11’
- Otherwise, access to this field is W1S.

Accessing PMCNTENSET_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCNTENSET_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCNTENSET_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCNTENSET_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCNTENSET_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCNTENSET_EL0();
end;
```

MSR PMCNTENSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCNTENSET_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMCNTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCNTENSET_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCNTENSET_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCNTENSET_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.8 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configuration

AArch64 System register PMCR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCR[31:0].

AArch64 System register PMCR_EL0 bits [31:0] are architecturally mapped to External register PMCR_EL0[31:0].

When FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMCR_EL0 bits [63:32] are architecturally mapped to External register PMCR_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMCR_EL0 are UNDEFINED.

Attributes

PMCR_EL0 is a 64-bit register.

Field descriptions

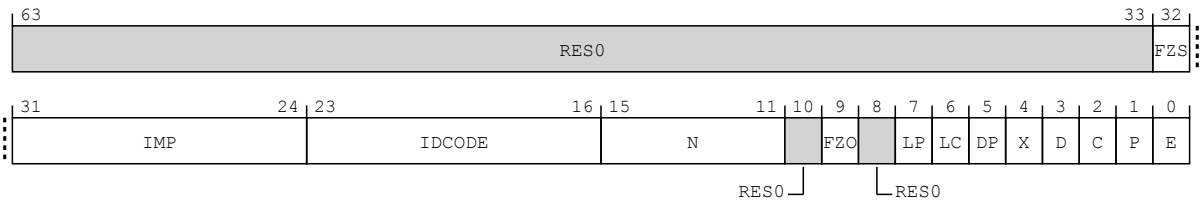

Bits [63:33]

Reserved, RES0.

FZS, bit [32]

When FEAT_SPEv1p2 is implemented:

Freeze-on-SPE event. Stop counters when PMBLIMITR_EL1.{PMFZ,E} is {1,1} and profiling is stopped.

FZS	Meaning
0b0	Do not freeze on a Statistical Profiling Buffer Management event.
0b1	Affected counters do not count following a Statistical Profiling Buffer Management event.

The pseudocode function `SPEProfilingStopped()` describes when profiling is stopped.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_SPE_DPFZS is implemented and PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When FEAT_SPE_DPFZS is not implemented or PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA32 is implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IMP, bits [31:24]

When FEAT_PMUv3p7 is not implemented:

Implementer code.

If this field is zero, then PMCR_EL0.IDCODE is RES0 and software must use MIDR_EL1 to identify the PE.

Otherwise, this field and PMCR_EL0.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A nonzero value has the same interpretation as MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

IDCODE, bits [23:16]

When PMCR_EL0.IMP != '00000000':

Identification code. Arm deprecates use of this field.

Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of `0b00000`-`0b11111`. If the value is `0b00000`, then only PMCCNTR_EL0 is implemented. If the value is `0b11111`, then PMCCNTR_EL0 and 31 event counters are implemented.

When EL2 is implemented and enabled for the current Security state, reads of this field from EL1 and EL0 return the Effective value of MDCR_EL2.HPMN.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

FZO	Meaning
0b0	Do not freeze on overflow.
0b1	Affected counters do not count when any of the following applies: • For any event counter PMEVCNTR<m>_EL0 in the first range, PMOVSCLR_EL0[m] is 1. • FEAT_PMUv3_ICNTR is implemented and PMOVSCLR_EL0.F0 is 1.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

LP, bit [7]

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.P[n].

LP	Meaning
0b0	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].
0b1	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

The counters affected by this field are the event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

When FEAT_AA32 is implemented:

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

Arm deprecates use of PMCR_EL0.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

When ((HaveEL(EL3) || (IsFeatureImplemented(FEAT_PMUv3p1) && HaveEL(EL2))) || IsFeatureImplemented(FEAT_PMUv3p7)) || IsFeatureImplemented(FEAT_SPE_DPFZS):

Disable cycle counter when event counting is prohibited.

DP	Meaning
0b0	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DP	Meaning
0b1	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2. • If FEAT_SPE_DPFZS is implemented and event counting is frozen by PMCR_EL0.FZS, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented, EL3 is implemented, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3. • If EL3 is implemented, MDCR_EL3.SPME is 0, and either FEAT_PMUv3p7 is not implemented or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited or frozen. For more information about event counter ranges, see MDCR_EL2.HPMN.

If FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented, meaning PMCR_EL0.FZS is implemented, and FEAT_SPE_DPFZS is not implemented, then cycle counting by PMCCNTR_EL0 is not affected by PMCR_EL0.FZS.

For more information, see ‘Prohibiting event and cycle counting’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETE is implemented, this field does not affect the use of PMU events as an External Input by the trace unit.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When FEAT_AA32 is implemented:

Clock divider.

D	Meaning
0b0	When enabled, PMCCNTR_EL0 counts every clock cycle.
0b1	When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If the Effective value of PMCR_EL0.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
0b0	No action.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P	Meaning
0b1	Reset all affected event counters PMEVCNTR<n>_EL0 to zero.

The event counters affected by this field are:

- All event counters in the first range.
- If any of the following are true, all event counters in the second range:
  - EL2 is disabled or not implemented in the current Security state.
  - The PE is executing at EL2 or EL3.

Writes to this field do not affect other event counters, the cycle counter PMCCNTR_EL0, or the instruction counter PMICNTR_EL0.

For more information about event counter ranges, see MDCR_EL2.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing PMCR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' || (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN ==
         ↪'1') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMCR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMCR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMCR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMCR_EL0();
end;
```

```
MSR PMCR_EL0, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' || (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN ==
         ↪'1') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMCR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMCR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMCR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMCR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMCR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCR_EL0() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.9 PMECR_EL1, Performance Monitors Extended Control Register (EL1)

The PMECR_EL1 characteristics are:

Purpose

Provides EL1 configuration options for the Performance Monitors.

Configuration

This register is present only when (FEAT_EBEP is implemented or FEAT_PMUv3_SS is implemented) and FEAT_AA64 is implemented. Otherwise, direct accesses to PMECR_EL1 are UNDEFINED.

Attributes

PMECR_EL1 is a 64-bit register.

Field descriptions

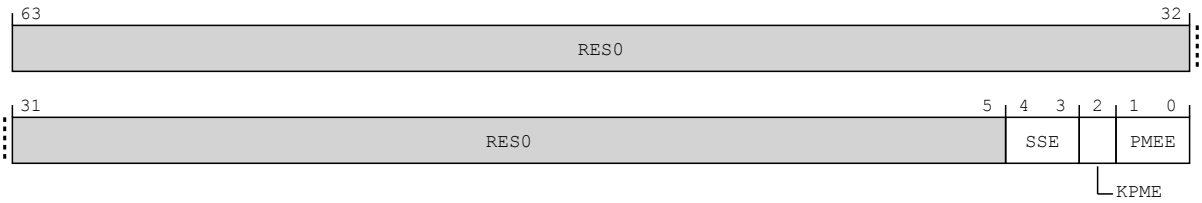

Bits [63:5]

Reserved, RES0.

SSE, bits [4:3]

When FEAT_PMUv3_SS is implemented:

Snapshot Enable. Controls the generation of Capture events.

SSE	Meaning
0b00	Capture events are disabled.
0b10	Capture events are enabled and prohibited.
0b11	Capture events are enabled and allowed.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When the highest implemented Exception level is EL1, this field resets to '00'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

KPME, bit [2]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_EBEP is implemented:

Local (Kernel) PMU Exception Enable. Enables PMU Profiling exceptions taken to the current Exception level.

KPME	Meaning
0b0	PMU Profiling exceptions taken to the current Exception level are disabled.
0b1	PMU Profiling exceptions taken to the current Exception level are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMEE, bits [1:0]

When FEAT_EBEP is implemented:

Performance Monitors Exception Enable. Controls the generation of the **PMUIRQ** signal and the PMU Profiling exception at EL0 and EL1.

PMEE	Meaning
0b00	The **PMUIRQ** signal is asserted on a PMU overflow, and the PMU Profiling exception is disabled.
0b11	The **PMUIRQ** signal is deasserted, and the PMU Profiling exception is enabled.

All other values are reserved.

This field is ignored by the PE when any of the following are true:

- All of the following are true:
  - EL3 is implemented.
  - MDCR_EL3.PMEE != 0b01.
- All of the following are true:
  - EL2 is implemented and enabled in the current Security State.
  - MDCR_EL2.PMEE != 0b01.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '00'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing PMECR_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PMECR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b101

```
if !((IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_PMUv3_SS)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMECR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMECR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMECR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMECR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR PMECR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b101

```
if !((IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_PMUv3_SS)) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nPMECR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMECR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMECR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMECR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.10 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configuration

AArch64 System register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVCNTR<n>[31:0].

AArch64 System register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to External register PMEVCNTR<n>_EL0[31:0].

When FEAT_PMUv3p5 is implemented, AArch64 System register PMEVCNTR<n>_EL0 bits [63:32] are architecturally mapped to External register PMEVCNTR<n>_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are UNDEFINED.

Attributes

PMEVCNTR<n>_EL0 is a 64-bit register.

Field descriptions

When FEAT_PMUv3p5 is implemented:

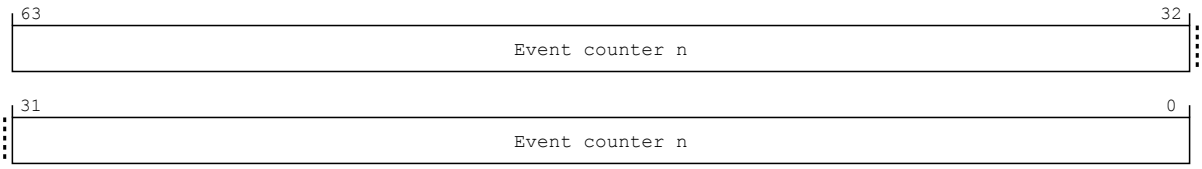

EVCNT, bits [63:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

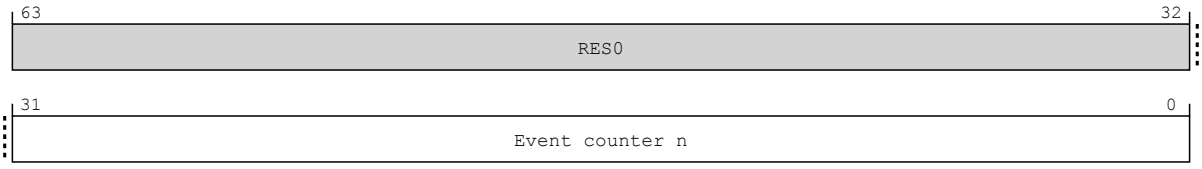

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:32]

Reserved, RES0.

EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>_EL0

PMEVCNTR<n>_EL0 can also be accessed by using PMXEVCNTR_EL0 with PMSELR_EL0.SEL set to the value of <n>.

If FEAT_FGT is implemented and <n> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVCNTR<n>_EL0 is as follows:

- If <n> is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVCNTR<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMEVCNTR<n>_EL0 are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.P<n> is 0.

Permitted writes of PMEVCNTR<n>_EL0 are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.{UEN,ER} is {1,1}.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{UEN,ER,EN}.
If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMEVCNTR<m>_EL0 ; Where m = 0-30
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b10:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMEVCNTRn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1()[m] == '0' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = PMEVCNTR_EL0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMEVCNTRn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = PMEVCNTR_EL0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMEVCNTR_EL0(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMEVCNTR_EL0(m);
end;
```

```
MSR PMEVCNTR<m>_EL0, <Xt> ; Where m = 0-30
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b10:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMEVCNTRn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1()[m] == '0' ||
         ↪PMUSERENR_EL0().ER == '1') then
        return;
    else
        PMEVCNTR_EL0(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMEVCNTRn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMEVCNTR_EL0(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMEVCNTR_EL0(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMEVCNTR_EL0(m) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.11 PMEVCNTSVR<n>_EL1, Performance Monitors Event Count Saved Value Registers, n = 0 - 30

The PMEVCNTSVR<n>_EL1 characteristics are:

Purpose

Captures the PMU Event counter <n>, PMEVCNTR<n>_EL0.

Configuration

AArch64 System register PMEVCNTSVR<n>_EL1 bits [63:0] are architecturally mapped to External register PMEVCNTSVR<n>_EL1[63:0].

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMEVCNTSVR<n>_EL1 are UNDEFINED.

Attributes

PMEVCNTSVR<n>_EL1 is a 64-bit register.

Field descriptions

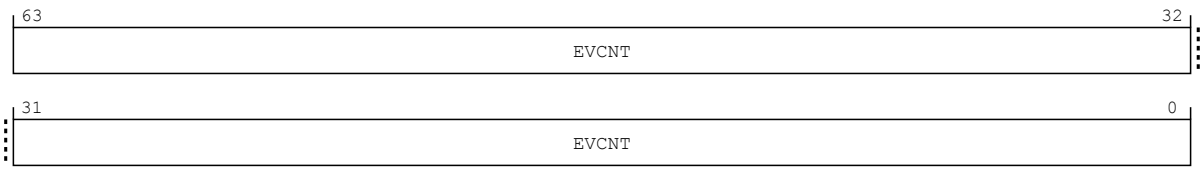

EVCNT, bits [63:0]

Sampled Event Count. The value of PMEVCNTR<n>_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTSVR<n>_EL1

If <n> is greater-than-or-equal-to the Effective value of PMCCR.EPMN, then direct reads of PMEVCNTSVR<n>_EL1 are UNDEFINED.

Otherwise, direct reads of PMEVCNTSVR<n>_EL1 generate a Trap exception to EL2 when all of the following are true:

- <n> is greater-than-or-equal-to the number of snapshot registers accessible at the current Exception level.
- EL2 is implemented and enabled in the current Security state.
- The access is from EL1.

Note

If EL2 is implemented and enabled in the current Security state, MDCR_EL2.HPMN identifies the number of accessible snapshot registers at EL1. Otherwise, the number of accessible snapshot registers is the number of implemented event counters. See MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PMEVCNTSVR<m>_EL1 ; Where m = 0-30
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1110	0b10:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_PMUv3_SS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMSSDATA == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMEVCNTSVR_EL1(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMEVCNTSVR_EL1(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMEVCNTSVR_EL1(m);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.12 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

AArch64 System register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVTYPER<n>[31:0].

AArch64 System register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

When FEAT_PMUv3_TH is implemented, or FEAT_PMUv3p8 is implemented, or FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_SME is implemented, AArch64 System register PMEVTYPER<n>_EL0 bits [63:32] are architecturally mapped to External register PMEVTYPER<n>_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are UNDEFINED.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions

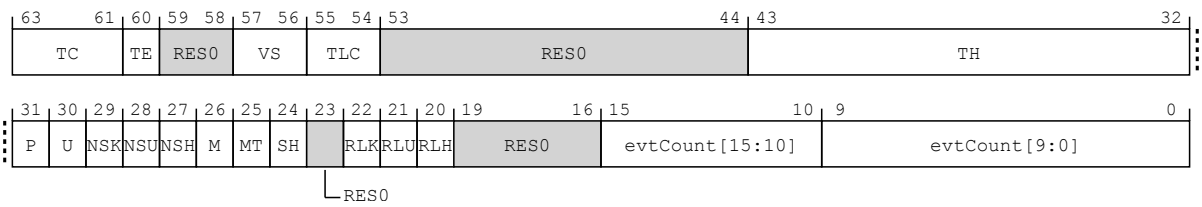

TC, bits [63:61]

When (IsFeatureImplemented(FEAT_PMUv3_TH) && ((!IsFeatureImplemented(FEAT_PMUv3_EDGE)) || (PMEVTYPER<n>_EL0.TE == '0'))) && (((!IsFeatureImplemented(FEAT_PMUv3_TH2)) || ((n MOD 2) == 0)) || (PMEVTYPER<n>_EL0.TLC IN {'0x'})):

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- For odd values of n, V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero. V[n-1] is not defined for even values of n.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
0b000	Not-equal. The counter increments by V_B[n] on each processor cycle when V_B[n] is not equal to TH[n].
0b001	Not-equal, count. The counter increments by 1 on each processor cycle when V_B[n] is not equal to TH[n].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TC	Meaning
0b010	Equals. The counter increments by V_B[n] on each processor cycle when V_B[n] is equal to TH[n].
0b011	Equals, count. The counter increments by 1 on each processor cycle when V_B[n] is equal to TH[n].
0b100	Greater-than-or-equal. The counter increments by V_B[n] on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b101	Greater-than-or-equal, count. The counter increments by 1 on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b110	Less-than. The counter increments by V_B[n] on each processor cycle when V_B[n] is less than TH[n].
0b111	Less-than, count. The counter increments by 1 on each processor cycle when V_B[n] is less than TH[n].

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC[2:1] is true:

- If PMEVTYPER<n>_EL0.TC[0] is 0, then the counter increments by V_B[n].
- If PMEVTYPER<n>_EL0.TC[0] is 1, then the counter increments by 1.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC[2:1] is false:

- If FEAT_PMUv3_TH2 is implemented, n is odd, and PMEVTYPER<n>_EL0.TLC is `0b01`, then the counter increments by V[n-1].
- Otherwise, the counter does not increment.

If PMEVTYPER<n>_EL0.{TC, TLC, TH} are zero, then the threshold function is disabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3_TH2 is implemented, PMEVTYPER<n>_EL0.TE == '0', (n MOD 2) == 1, and PMEVTYPER<n>_EL0.TLC == '10':

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
0b000	Not-equal. The counter increments by V[n-1] on each processor cycle when V_B[n] is not equal to TH[n].
0b010	Equals. The counter increments by V[n-1] on each processor cycle when V_B[n] is equal to TH[n].
0b100	Greater-than-or-equal. The counter increments by V[n-1] on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b110	Less-than. The counter increments by V[n-1] on each processor cycle when V_B[n] is less than TH[n].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is true, the counter increments by V[n-1].

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is false, the counter does not increment.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3_EDGE is implemented and PMEVTYPER<n>_EL0.TE == '1':

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- For odd values of n, V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero. V[n-1] is not defined for even values of n.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
`0b001`	Equal to not-equal. The counter increments on each processor cycle when V_B[n] is not equal to TH[n] and V_B[n] was equal to TH[n] on the previous processor cycle.
`0b010`	Equal to/from not-equal. The counter increments on each processor cycle when either: • V_B[n] is not equal to TH[n] and V_B[n] was equal to TH[n] on the previous processor cycle. • V_B[n] is equal to TH[n] and V_B[n] was not equal to TH[n] on the previous processor cycle.
`0b011`	Not-equal to equal. The counter increments on each processor cycle when V_B[n] is equal to TH[n] and V_B[n] was not equal to TH[n] on the previous processor cycle.
`0b101`	Less-than to greater-than-or-equal. The counter increments on each processor cycle when V_B[n] is greater than or equal to TH[n] and V_B[n] was less than TH[n] on the previous processor cycle.
`0b110`	Less-than to/from greater-than-or-equal. The counter increments on each processor cycle when either: • V_B[n] is greater than or equal to TH[n] and V_B[n] was less than TH[n] on the previous processor cycle. • V_B[n] is less than TH[n] and V_B[n] was greater than or equal to TH[n] on the previous processor cycle.
`0b111`	Greater-than-or-equal to less-than. The counter increments on each processor cycle when V_B[n] is less than TH[n] and V_B[n] was greater than or equal to TH[n] on the previous processor cycle.

All other values are reserved.

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is true:

- If FEAT_PMUv3_TH2 is implemented, n is odd, and PMEVTYPER<n>_EL0.TLC is `0b10`, then the counter increments by V[n-1].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, the counter increments by 1.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is false, the counter does not increment.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TE, bit [60]

When FEAT_PMUv3_EDGE is implemented:

Threshold Edge. Enables the edge condition. When PMEVTYPER<n>_EL0.TE is 1, the event counter increments on cycles when the result of the threshold condition changes. See PMEVTYPER<n>_EL0.TC for more information.

TE	Meaning
`0b0`	Threshold edge condition disabled.
`0b1`	Threshold edge condition enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [59:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting events in Streaming and Non-streaming SVE modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VS	Meaning
0b00	This mechanism has no effect on the filtering of events.
0b01	The PE does not count events in Streaming SVE mode.
0b10	The PE does not count events in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLC, bits [55:54]

When FEAT_PMUv3_TH2 is implemented and (n MOD 2) == 1:

Threshold Linking Control. Extends PMEVTYPER<n>_EL0.TC with additional controls for event linking. See PMEVTYPER<n>_EL0.TC.

TLC	Meaning
0b00	Threshold linking disabled.
0b01	Threshold linking enabled. If the threshold condition described by PMEVTYPER<n>_EL0.TC is false, the counter increments by V[n-1]. Otherwise, the counter increments as described by PMEVTYPER<n>_EL0.TC.
0b10	Threshold linking enabled. If the threshold condition described by PMEVTYPER<n>_EL0.TC is true, the counter increments by V[n-1]. Otherwise, the counter does not increment.

All other values are reserved.

See PMEVTYPER<n>_EL0.TC for more information

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [53:44]

Reserved, RES0.

TH, bits [43:32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_TH is implemented:

Threshold value. Provides the unsigned value for the threshold function defined by PMEVTYPER<n>_EL0.TC.

If PMEVTYPER<n>_EL0.{TC, TH} are both zero and either FEAT_PMUv3_TH2 is not implemented or PMEVTYPER<n>_EL0.TLC is also zero, then the threshold function is disabled.

If PMMIR_EL1.THWIDTH is less than 12, then bits PMEVTYPER<n>_EL0.TH[11:UInt(PMMIR_EL1.THWIDTH)] are RES0. This accounts for the behavior when writing a value greater-than-or-equal-to $2^{\text{UInt(PMMIR_EL1.THWIDTH)}}$.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to the expression `0x000`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting events in EL1.

P	Meaning
`0b0`	This mechanism has no effect on filtering of events.
`0b1`	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>_EL0.NSK.

If FEAT_RME is implemented, then counting events in Realm EL1 is further controlled by PMEVTYPER<n>_EL0.RLK.

If EL3 is implemented, then counting events at EL3 is further controlled by PMEVTYPER<n>_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting events in EL0.

U	Meaning
`0b0`	This mechanism has no effect on filtering of events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>_EL0.NSU.

If FEAT_RME is implemented, then counting events in Realm EL0 is further controlled by PMEVTYPER<n>_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting events in Non-secure EL1. If PMEVTYPER<n>_EL0.NSK is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL1.

NSK	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in Non-secure EL1.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in Non-secure EL1. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting events in Non-secure EL0. If PMEVTYPER<n>_EL0.NSU is not equal to PMEVTYPER<n>_EL0.U, then the PE does not count events in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSU	Meaning
0b0	When PMEVTYPER<n>_EL0.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.U == 1, the PE does not count events in Non-secure EL0.
0b1	When PMEVTYPER<n>_EL0.U == 0, the PE does not count events in Non-secure EL0. When PMEVTYPER<n>_EL0.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting events in Secure EL2 is further controlled by PMEVTYPER<n>_EL0.SH.

If FEAT_RME is implemented, then counting events in Realm EL2 is further controlled by PMEVTYPER<n>_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

EL3 filtering. Controls counting events in EL3. If PMEVTYPER<n>_EL0.M is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in EL3. Otherwise, this mechanism has no effect on filtering of events in EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in EL3.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in EL3. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MT, bit [25]

When FEAT_MTPMU is implemented or an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See ‘Multithreaded implementations’ and ‘Cycle event counting in multithreaded implementations’.

From Armv8.6, the IMPLEMENTATION DEFINED multi-threaded PMU extension is not permitted, meaning if FEAT_MTPMU is not implemented, this field is RES0. See ID_AA64DFR0_EL1.MTPMU.

This field is ignored by the PE and treated as zero when FEAT_MTPMU is implemented and disabled.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting events in Secure EL2. If PMEVTYPER<n>_EL0.SH is equal to PMEVTYPER<n>_EL0.NSH, then the PE does not count events in Secure EL2. Otherwise, this mechanism has no effect on filtering of events in Secure EL2.

SH	Meaning
`0b0`	When PMEVTYPER<n>_EL0.NSH == 0, the PE does not count events in Secure EL2. When PMEVTYPER<n>_EL0.NSH == 1, this mechanism has no effect on filtering of events.
`0b1`	When PMEVTYPER<n>_EL0.NSH == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.NSH == 1, the PE does not count events in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting events in Realm EL1. If PMEVTYPER<n>_EL0.RLK is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in Realm EL1. Otherwise, this mechanism has no effect on filtering of events in Realm EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLK	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in Realm EL1.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in Realm EL1. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting events in Realm EL0. If PMEVTYPER<n>_EL0.RLU is not equal to PMEVTYPER<n>_EL0.U, then the PE does not count events in Realm EL0. Otherwise, this mechanism has no effect on filtering of events in Realm EL0.

RLU	Meaning
0b0	When PMEVTYPER<n>_EL0.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.U == 1, the PE does not count events in Realm EL0.
0b1	When PMEVTYPER<n>_EL0.U == 0, the PE does not count events in Realm EL0. When PMEVTYPER<n>_EL0.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting events in Realm EL2. If PMEVTYPER<n>_EL0.RLH is equal to PMEVTYPER<n>_EL0.NSH, then the PE does not count events in Realm EL2. Otherwise, this mechanism has no effect on filtering of events in Realm EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLH	Meaning
`0b0`	When PMEVTYPER<n>_EL0.NSH == 0, the PE does not count events in Realm EL2. When PMEVTYPER<n>_EL0.NSH == 1, this mechanism has no effect on filtering of events.
`0b1`	When PMEVTYPER<n>_EL0.NSH == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.NSH == 1, the PE does not count events in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When FEAT_PMUv3p1 is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

The ranges of event numbers allocated to each type of event are shown in ‘Allocation of the PMU event number space’.

If FEAT_PMUv3p8 is implemented and PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.

Note

Arm recommends this behavior for all implementations of FEAT_PMUv3.

Otherwise, if PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_PMUv3p1 is implemented, for the range `0x4000` to `0x403F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>_EL0

PMEVTYPER<n>_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to n.

If FEAT_FGT is implemented and <n> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVTYPER<n>_EL0 is as follows:

- If <n> is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMEVTYPER<n>_EL0 are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.P<n> is 0.

Permitted writes of PMEVTYPER<n>_EL0 are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.{UEN,ER} is {1,1}.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{UEN,EN}.
If EL2 is implemented and enabled in the current Security state, at EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PMEVTYPER<m>_EL0 ; Where m = 0-30
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1()[m] == '0' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = PMEVTYPER_EL0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMEVTYPER_EL0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMEVTYPER_EL0(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMEVTYPER_EL0(m);
end;
```

```
MSR PMEVTYPER<m>_EL0, <Xt> ; Where m = 0-30
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1()[m] == '0' ||
         ↪PMUSERENR_EL0().ER == '1') then
        return;
    else
        PMEVTYPER_EL0(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMEVTYPER_EL0(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMEVTYPER_EL0(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMEVTYPER_EL0(m) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.13 PMICFILTR_EL0, Performance Monitors Instruction Counter Filter Register

The PMICFILTR_EL0 characteristics are:

Purpose

Configures the Instruction Counter.

Configuration

AArch64 System register PMICFILTR_EL0 bits [63:0] are architecturally mapped to External register PMICFILTR_EL0[63:0].

This register is present only when FEAT_PMUv3_ICNTR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMICFILTR_EL0 are UNDEFINED.

Attributes

PMICFILTR_EL0 is a 64-bit register.

Field descriptions

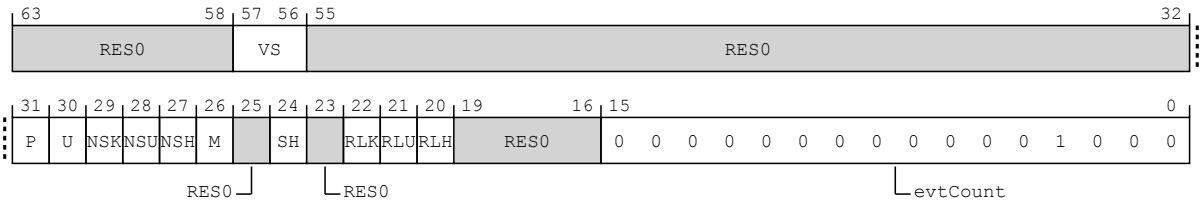

Bits [63:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting instructions in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of instructions.
0b01	The PE does not count instructions in Streaming SVE mode.
0b10	The PE does not count instructions in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting instructions in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of instructions.
0b1	The PE does not count instructions in EL1.

If Secure and Non-secure states are implemented, then counting instructions in Non-secure EL1 is further controlled by PMICFILTR_EL0.NSK.

If FEAT_RME is implemented, then counting instructions in Realm EL1 is further controlled by PMICFILTR_EL0.RLK.

If EL3 is implemented, then counting instructions at EL3 is further controlled by PMICFILTR_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting instructions in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of instructions.
0b1	The PE does not count instructions in EL0.

If Secure and Non-secure states are implemented, then counting instructions in Non-secure EL0 is further controlled by PMICFILTR_EL0.NSU.

If FEAT_RME is implemented, then counting instructions in Realm EL0 is further controlled by PMICFILTR_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting instructions in Non-secure EL1. If PMICFILTR_EL0.NSK is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of instructions in Non-secure EL1.

NSK	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in Non-secure EL1.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in Non-secure EL1. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting instructions in Non-secure EL0. If PMICFILTR_EL0.NSU is not equal to PMICFILTR_EL0.U, then the PE does not count instructions in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of instructions in Non-secure EL0.

NSU	Meaning
0b0	When PMICFILTR_EL0.U == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.U == 1, the PE does not count instructions in Non-secure EL0.
0b1	When PMICFILTR_EL0.U == 0, the PE does not count instructions in Non-secure EL0. When PMICFILTR_EL0.U == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL2 is implemented:

EL2 filtering. Controls counting instructions in EL2.

NSH	Meaning
0b0	The PE does not count instructions in EL2.
0b1	This mechanism has no effect on filtering of instructions.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting instructions in Secure EL2 is further controlled by PMICFILTR_EL0.SH.

If FEAT_RME is implemented, then counting instructions in Realm EL2 is further controlled by PMICFILTR_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

EL3 filtering. Controls counting instructions in EL3. If PMICFILTR_EL0.M is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in EL3. Otherwise, this mechanism has no effect on filtering of instructions in EL3.

M	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in EL3.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in EL3. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [25]

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting instructions in Secure EL2. If PMICFILTR_EL0.SH is equal to PMICFILTR_EL0.NSH, then the PE does not count instructions in Secure EL2. Otherwise, this mechanism has no effect on filtering of instructions in Secure EL2.

SH	Meaning
0b0	When PMICFILTR_EL0.NSH == 0, the PE does not count instructions in Secure EL2. When PMICFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of instructions.
0b1	When PMICFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.NSH == 1, the PE does not count instructions in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting instructions in Realm EL1. If PMICFILTR_EL0.RLK is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in Realm EL1. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL1.

RLK	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in Realm EL1.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in Realm EL1. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting instructions in Realm EL0. If PMICFILTR_EL0.RLU is not equal to PMICFILTR_EL0.U, then the PE does not count instructions in Realm EL0. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL0.

RLU	Meaning
`0b0`	When PMICFILTR_EL0.U == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.U == 1, the PE does not count instructions in Realm EL0.
`0b1`	When PMICFILTR_EL0.U == 0, the PE does not count instructions in Realm EL0. When PMICFILTR_EL0.U == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting instructions in Realm EL2. If PMICFILTR_EL0.RLH is equal to PMICFILTR_EL0.NSH, then the PE does not count instructions in Realm EL2. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL2.

RLH	Meaning
`0b0`	When PMICFILTR_EL0.NSH == 0, the PE does not count instructions in Realm EL2. When PMICFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of instructions.
`0b1`	When PMICFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.NSH == 1, the PE does not count instructions in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count.

Reads as `0x0008`

Access to this field is RO.

Accessing PMICFILTR_EL0

Permitted reads and writes of PMICFILTR_EL0 are RAZ/WI if all of the following are true:

- EL0 is the current Exception level.
- PMUACR_EL1.F0 is 0.

Permitted writes of PMICFILTR_EL0 are ignored if all of the following are true:

- EL0 is the current Exception level.
- PMUSERENR_EL0.IR is 1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMICFILTR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b0110	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3_ICNTR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().UEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nPMICFILTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().F0 == '0' then
            X{64}(t) = Zeros{64};
        else
            X{64}(t) = PMICFILTR_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGRTR2_EL2().nPMICFILTR_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMICFILTR_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMICFILTR_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = PMICFILTR_EL0();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR PMICFILTR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b0110	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3_ICNTR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().UEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
     ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nPMICFILTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().F0 == '0' ||
     ↪PMUSERENR_EL0().IR == '1') then
        return;
    else
        PMICFILTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
     ↪|| HDFGWTR2_EL2().nPMICFILTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMICFILTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMICFILTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMICFILTR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.14 PMICNTR_EL0, Performance Monitors Instruction Counter Register

The PMICNTR_EL0 characteristics are:

Purpose

If event counting is not prohibited and the instruction counter is enabled, the counter increments for each architecturally-executed instruction, according to the configuration specified by PMICFILTR_EL0.

Configuration

AArch64 System register PMICNTR_EL0 bits [63:0] are architecturally mapped to External register PMICNTR_EL0[63:0].

This register is present only when FEAT_PMUv3_ICNTR is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMICNTR_EL0 are UNDEFINED.

Attributes

PMICNTR_EL0 is a 64-bit register.

Field descriptions

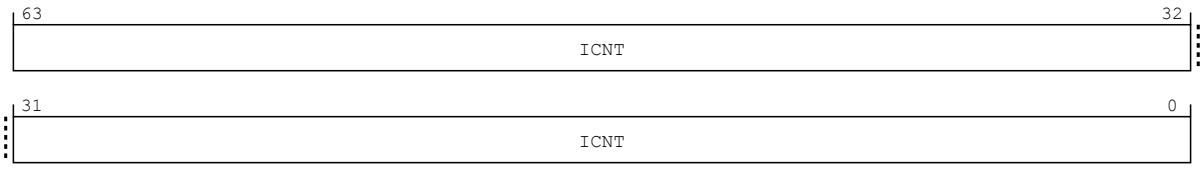

ICNT, bits [63:0]

Instruction Counter.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMICNTR_EL0

PMICNTR_EL0 treats permitted reads-as-zero and ignores permitted writes if all of the following are true:

- EL0 is the current Exception level.
- PMUACR_EL1.F0 is 0.

PMICNTR_EL0 ignores permitted writes if all of the following are true:

- EL0 is the current Exception level.
- PMUSERENR_EL0.IR is 1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PMICNTR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3_ICNTR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().UEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
     ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nPMICNTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().F0 == '0' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = PMICNTR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
     ↪|| HDFGRTR2_EL2().nPMICNTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        X{64}(t) = PMICNTR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMICNTR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMICNTR_EL0();
end;
```

```
MSR PMICNTR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3_ICNTR) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().UEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nPMICNTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().F0 == '0' ||
     ↪PMUSERENR_EL0().IR == '1') then
        return;
    else
        PMICNTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
     ↪|| HDFGWTR2_EL2().nPMICNTR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMICNTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMICNTR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMICNTR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.15 PMICNTSVR_EL1, Performance Monitors Instruction Count Saved Value Register

The PMICNTSVR_EL1 characteristics are:

Purpose

Captures the PMU Instruction counter, PMICNTR_EL0.

Configuration

AArch64 System register PMICNTSVR_EL1 bits [63:0] are architecturally mapped to External register PMICNTSVR_EL1[63:0].

This register is present only when FEAT_PMUv3_ICNTR is implemented, FEAT_PMUv3_SS is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to PMICNTSVR_EL1 are UNDEFINED.

Attributes

PMICNTSVR_EL1 is a 64-bit register.

Field descriptions

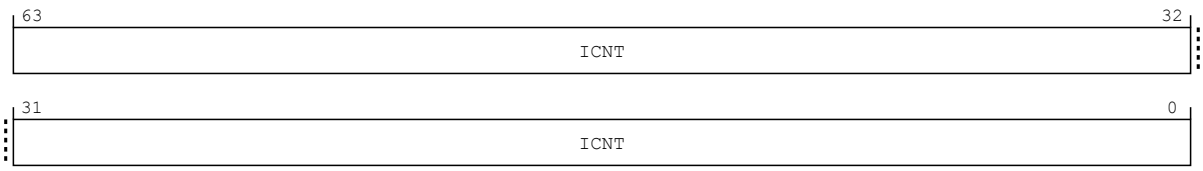

ICNT, bits [63:0]

Sampled Instruction Count. The value of PMICNTR_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMICNTSVR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMICNTSVR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1110	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3_ICNTR) && IsFeatureImplemented(FEAT_PMUv3_SS) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMSSDATA == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMICNTSVR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMICNTSVR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMICNTSVR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.16 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register

The PMINTENCLR_EL1 characteristics are:

Purpose

Allows software to disable the generation of interrupt requests or, when FEAT_EBEP is implemented, PMU Profiling exceptions on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which overflow interrupt requests or PMU Profiling exceptions are enabled.

Configuration

AArch64 System register PMINTENCLR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[63:0].

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMINTENCLR_EL1 bits [63:32] are architecturally mapped to External register PMINTENCLR_EL1[63:32].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMINTENCLR_EL1 bits [63:32] are architecturally mapped to External register PMINTENSET_EL1[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are UNDEFINED.

Attributes

PMINTENCLR_EL1 is a 64-bit register.

Field descriptions

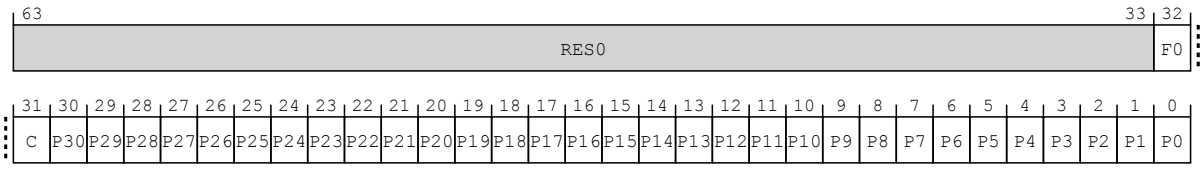

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMICNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable status.

F0	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL1 is the current Exception level
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - All of the following are true:
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
      - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is WO/RAZ if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL1 is the current Exception level
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL1 is the current Exception level
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMINTENCLR_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b010

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMINTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMINTENCLR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMINTENCLR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMINTENCLR_EL1();
end;
```

```
MSR PMINTENCLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b010

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMINTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMINTENCLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMINTENCLR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMINTENCLR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.17 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register

The PMINTENSET_EL1 characteristics are:

Purpose

Allows software to enable the generation of interrupt requests or, when FEAT_EBEP is implemented, PMU Profiling exceptions on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which overflow interrupt requests or PMU Profiling exceptions are enabled.

Configuration

AArch64 System register PMINTENSET_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[63:0].

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMINTENSET_EL1 bits [63:32] are architecturally mapped to External register PMINTENSET_EL1[63:32].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMINTENSET_EL1 bits [63:32] are architecturally mapped to External register PMINTENCLR_EL1[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are UNDEFINED.

Attributes

PMINTENSET_EL1 is a 64-bit register.

Field descriptions

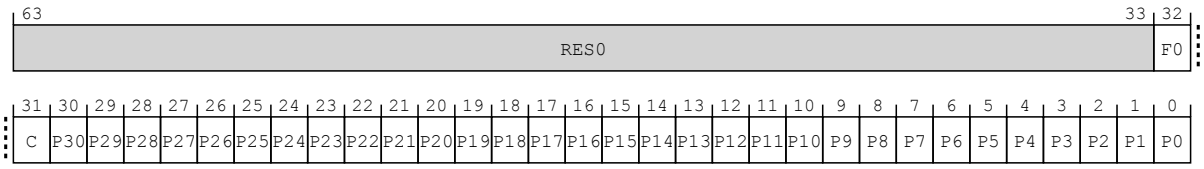

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMICNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable status.

F0	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL1 is the current Exception level
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - All of the following are true:
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
      - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is WO/RAZ if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL1 is the current Exception level
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL1 is the current Exception level
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing PMINTENSET_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMINTENSET_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMINTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMINTENSET_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMINTENSET_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMINTENSET_EL1();
end;
```

```
MSR PMINTENSET_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMINTEN == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMINTENSET_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMINTENSET_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMINTENSET_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.18 PMMIR_EL1, Performance Monitors Machine Identification Register

The PMMIR_EL1 characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation to software.

Configuration

AArch64 System register PMMIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMMIR[31:0].

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented, AArch64 System register PMMIR_EL1 bits [31:0] are architecturally mapped to External register PMMIR[31:0].

When (IsFeatureImplemented(FEAT_PMUv3_EXT) && IsFeatureImplemented(FEAT_PMUv3p4)) && (IsFeatureImplemented(FEAT_PMUv3_EXT64) || IsFeatureImplemented(FEAT_PMUv3p9)), AArch64 System register PMMIR_EL1 bits [63:32] are architecturally mapped to External register PMMIR[63:32].

This register is present only when FEAT_PMUv3p4 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMMIR_EL1 are UNDEFINED.

Attributes

PMMIR_EL1 is a 64-bit register.

Field descriptions

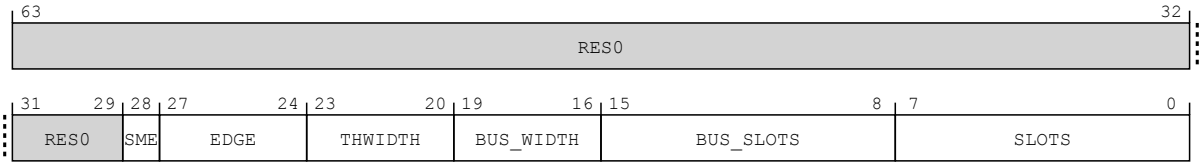

Bits [63:29]

Reserved, RES0.

SME, bit [28]

PMUv3 for SME. Adds support for the Streaming SVE mode filter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
0b0	Streaming SVE mode filter not implemented.
0b1	Adds support for the Streaming SVE mode filter.

All other values are reserved.

FEAT_PMUv3_SME implements the functionality identified by the value 1.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDGE, bits [27:24]

PMU event edge detection. With PMMIR_EL1.THWIDTH, indicates implementation of event counter thresholding features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDGE	Meaning
0b0000	FEAT_PMUv3_EDGE is not implemented.
0b0001	FEAT_PMUv3_EDGE is implemented.
0b0010	As 0b0001, and adds support for threshold value linking between a pair of counters.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, the only permitted value is 0b0000.

FEAT_PMUv3_EDGE implements the functionality identified by the value 0b0001.

FEAT_PMUv3_TH2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

THWIDTH, bits [23:20]

PMEVTYPER<n>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<n>_EL0.TH field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

THWIDTH	Meaning
0b0000	FEAT_PMUv3_TH is not implemented.
0b0001	1 bit. PMEVTYPER<n>_EL0.TH[11:1] are RES0.
0b0010	2 bits. PMEVTYPER<n>_EL0.TH[11:2] are RES0.
0b0011	3 bits. PMEVTYPER<n>_EL0.TH[11:3] are RES0.
0b0100	4 bits. PMEVTYPER<n>_EL0.TH[11:4] are RES0.
0b0101	5 bits. PMEVTYPER<n>_EL0.TH[11:5] are RES0.
0b0110	6 bits. PMEVTYPER<n>_EL0.TH[11:6] are RES0.
0b0111	7 bits. PMEVTYPER<n>_EL0.TH[11:7] are RES0.
0b1000	8 bits. PMEVTYPER<n>_EL0.TH[11:8] are RES0.
0b1001	9 bits. PMEVTYPER<n>_EL0.TH[11:9] are RES0.
0b1010	10 bits. PMEVTYPER<n>_EL0.TH[11:10] are RES0.
0b1011	11 bits. PMEVTYPER<n>_EL0.TH[11] is RES0.
0b1100	12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<n>_EL0.TH is $2^{(\text{PMMIR_EL1.THWIDTH})}$ minus one.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
`0b0000`	The information is not available.
`0b0011`	Four bytes.
`0b0100`	8 bytes.
`0b0101`	16 bytes.
`0b0110`	32 bytes.
`0b0111`	64 bytes.
`0b1000`	128 bytes.
`0b1001`	256 bytes.
`0b1010`	512 bytes.
`0b1011`	1024 bytes.
`0b1100`	2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, PMMIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b110

```
if !(IsFeatureImplemented(FEAT_PMUv3p4) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMMIR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMMIR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMMIR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMMIR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.19 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register

The PMOVSCLR_EL0 characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch64 System register PMOVSCLR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[63:0].

AArch64 System register PMOVSCLR_EL0 bits [63:0] are architecturally mapped to External register PMOVS[63:0].

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSSET[31:0].

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMOVSCLR_EL0 bits [63:32] are architecturally mapped to External register PMOVSCLR_EL0[63:32].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMOVSCLR_EL0 bits [63:32] are architecturally mapped to External register PMOVSSET_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are UNDEFINED.

Attributes

PMOVSCLR_EL0 is a 64-bit register.

Field descriptions

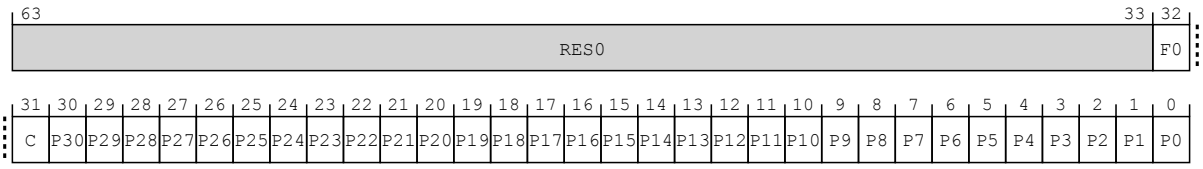

Bits [63:33]

Reserved, RES0.

F0, bit [32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_ICNTR is implemented:

Unsigned overflow flag for PMICNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMICNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMICNTR_EL0 overflow status.

F0	Meaning
`0b0`	PMICNTR_EL0 has not overflowed.
`0b1`	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - EL0 is the current Exception level
  - Any of the following are true:
    - PMUSERENR_EL0.UEN == ‘0’
    - PMUACR_EL1.F0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - All of the following are true:
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
      - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL0 is the current Exception level
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
  - PMUSERENR_EL0.IR == ‘1’
- Access to this field is WO/RAZ if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

        - SCR_EL3.FGTEn2 == ‘0’
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
    - Access to this field is RO if all of the following are true:
      - FEAT_FGT2 is implemented
      - EL2Enabled()
      - PSTATE.EL IN {EL1, EL0}
      - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
      - Any of the following are true:
        - All of the following are true:
          - EL3 is implemented
          - SCR_EL3.FGTEn2 == ‘0’
        - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
    - Access to this field is RO if all of the following are true:
      - EL0 is the current Exception level
      - PMUSERENR_EL0.IR == ‘1’
    - Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.C == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == ‘11’
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.P<m> == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == ‘11’
- Otherwise, access to this field is W1C.

Accessing PMOVSCLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMOVSCLR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b011

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMOVS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMOVSCLR_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().PMOVS == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMOVSCLR_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMOVSCLR_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = PMOVSCLR_EL0();
    end;
```

```
MSR PMOVSCLR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b011

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMOVS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMOVSCLR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMOVS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMOVSCLR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMOVSCLR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMOVSCLR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.20 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register

The PMOVSSET_EL0 characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch64 System register PMOVSSET_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[63:0].

AArch64 System register PMOVSSET_EL0 bits [63:0] are architecturally mapped to External register PMOVS[63:0].

AArch64 System register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSSET[31:0].

AArch64 System register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

AArch64 System register PMOVSSET_EL0 bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

AArch64 System register PMOVSSET_EL0 bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMOVSSET_EL0 bits [63:32] are architecturally mapped to External register PMOVSCLR_EL0[63:32].

When FEAT_PMUv3p9 is implemented or FEAT_PMUv3_EXT64 is implemented, AArch64 System register PMOVSSET_EL0 bits [63:32] are architecturally mapped to External register PMOVSSET_EL0[63:32].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are UNDEFINED.

Attributes

PMOVSSET_EL0 is a 64-bit register.

Field descriptions

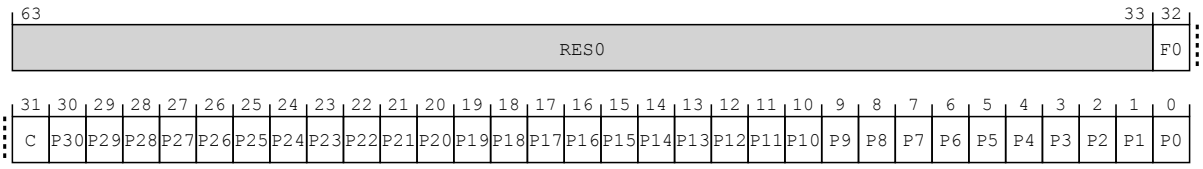

Bits [63:33]

Reserved, RES0.

F0, bit [32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_ICNTR is implemented:

Unsigned overflow flag for PMICNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMICNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMICNTR_EL0 overflow status.

F0	Meaning
`0b0`	PMICNTR_EL0 has not overflowed.
`0b1`	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - EL0 is the current Exception level
  - Any of the following are true:
    - PMUSERENR_EL0.UEN == ‘0’
    - PMUACR_EL1.F0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - All of the following are true:
      - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
      - HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - EL0 is the current Exception level
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    - HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
  - PMUSERENR_EL0.IR == ‘1’
- Access to this field is WO/RAZ if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    - All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— HDFGRTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_FGT2 is implemented
  - EL2Enabled()
  - PSTATE.EL IN {EL1, EL0}
  - [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
  - Any of the following are true:
    — All of the following are true:
      - EL3 is implemented
      - SCR_EL3.FGTEn2 == ‘0’
    — HDFGWTR2_EL2.nPMICFILTR_EL0 == ‘0’
- Access to this field is RO if all of the following are true:
  - EL0 is the current Exception level
  - PMUSERENR_EL0.IR == ‘1’
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.C == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == ‘11’
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.P<m> == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == '11'
- Otherwise, access to this field is W1S.

Accessing PMOVSSET_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMOVSSET_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b011

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMOVS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMOVSSET_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().PMOVS == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMOVSSET_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMOVSSET_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = PMOVSSET_EL0();
    end;
```

```
MSR PMOVSSET_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b011

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMOVS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMOVSSET_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMOVS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMOVSSET_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMOVSSET_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMOVSSET_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.21 PMSELR_EL0, Performance Monitors Event Counter Selection Register

The PMSELR_EL0 characteristics are:

Purpose

Selects the current event counter PMEVCNTR<n>_EL0 or the cycle counter PMCCNTR_EL0.

Used in conjunction with PMXEVTYPER_EL0 to determine the event that increments a selected counter, and the modes and states in which the selected counter increments.

Used in conjunction with PMXEVCNTR_EL0 to determine the value of a selected counter.

Configuration

AArch64 System register PMSELR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMSELR[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMSELR_EL0 are UNDEFINED.

Attributes

PMSELR_EL0 is a 64-bit register.

Field descriptions

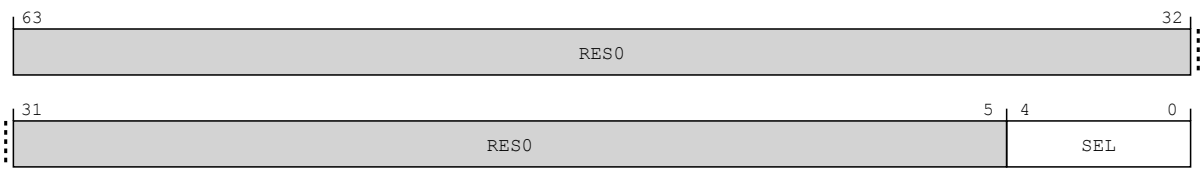

Bits [63:5]

Reserved, RES0.

SEL, bits [4:0]

Event counter select. Selects the counter accessed by subsequent accesses to PMXEVTYPER_EL0 and PMXEVCNTR_EL0.

SEL	Meaning
0b00000..0b11110	Select event counter PMEVCNTR<n>_EL0, where n is the value of this field: • MRS and MSR of PMXEVTYPER_EL0 access PMEVTYPER<n>_EL0. • MRS and MSR of PMXEVCNTR_EL0 access PMEVCNTR<n>_EL0.
0b11111	Select the cycle counter, PMCCNTR_EL0: • MRS and MSR of PMXEVTYPER_EL0 access PMCCFILTR_EL0. • MRS and MSR of PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE. For more information, see PMXEVCNTR_EL0.

If FEAT_FGT is not implemented and this field is set to a value greater than or equal to the number of implemented counters, but not equal to 31, then direct reads of this field return an UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the results of accesses to the event counters, including when PMSELR_EL0.SEL is set to the index of an unimplemented or inaccessible event counter, see PMXEVTYPER_EL0 and PMXEVCNTR_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSELR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSELR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b101

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMSELR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSELR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSELR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSELR_EL0();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSELR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSELR_EL0();
end;
```

MSR PMSELR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b101

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMSELR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSELR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSELR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        PMSELR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSELR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSELR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.22 PMSSCR_EL1, Performance Monitors Snapshot Status and Capture Register

The PMSSCR_EL1 characteristics are:

Purpose

Holds status information about the captured counters and provides a mechanism for software to initiate a sample.

Configuration

AArch64 System register PMSSCR_EL1 bits [63:0] are architecturally mapped to External register PMSSCR_EL1[63:0].

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMSSCR_EL1 are UNDEFINED.

Attributes

PMSSCR_EL1 is a 64-bit register.

Field descriptions

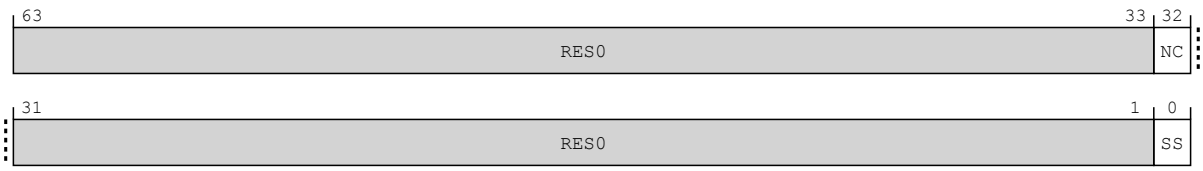

Bits [63:33]

Reserved, RES0.

NC, bit [32]

No Capture. Indicates whether the PMU counters have been captured.

NC	Meaning
0b0	PMU counters captured.
0b1	PMU counters not captured.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [31:1]

Reserved, RES0.

SS, bit [0]

Snapshot Capture and Status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS	Meaning
0b0	On a read, the Capture event has completed.
0b1	On a read, the Capture event has not completed. On a write, request a Capture event.

A write of 0 to this field is ignored.

It is CONSTRAINED UNPREDICTABLE whether a Capture event has completed if this field is modified when the Capture event is ongoing.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When PMU capture events are disabled, access to this field is RO.
- Otherwise, access to this field is RW.

Accessing PMSSCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSSCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_PMUv3_SS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMSSCR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSSCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSSCR_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSSCR_EL1();
end;
```

```
MSR PMSSCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_PMUv3_SS) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nPMSSCR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSS == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSSCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.23 PMSWINC_EL0, Performance Monitors Software Increment Register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see 'SW_INCR'.

Configuration

AArch64 System register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMSWINC[31:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, AArch64 System register PMSWINC_EL0 bits [31:0] are architecturally mapped to External register PMSWINC_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMSWINC_EL0 are UNDEFINED.

Attributes

PMSWINC_EL0 is a 64-bit register.

Field descriptions

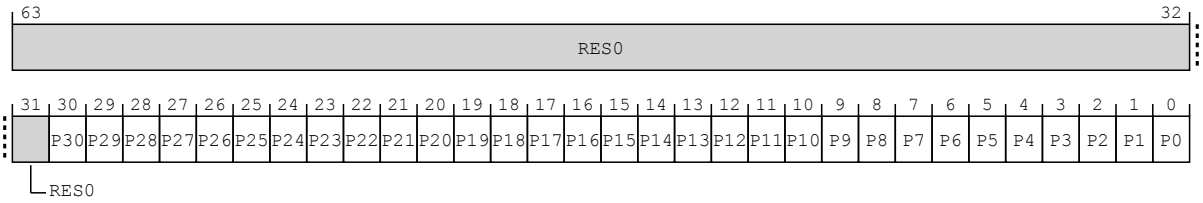

Bits [63:31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Software increment.

P<m>	Meaning
`0b0`	Write is ignored.
`0b1`	Increment PMEVCNTR<m>_EL0, if PMEVCNTR<m>_EL0 is configured to count software increment events.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.SW] == '10'
  - PMUACR_EL1.P<m> == '0'
- Otherwise, access to this field is WO/RAZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMSWINC_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MSR PMSWINC_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b100

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,SW,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[SW,EN] == '00') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMSWINC_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSWINC_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSWINC_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSWINC_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        PMSWINC_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSWINC_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.24 PMUACR_EL1, Performance Monitors User Access Control Register

The PMUACR_EL1 characteristics are:

Purpose

Enables or disables EL0 access to specfic Performance Monitors.

Configuration

This register is present only when FEAT_PMUv3p9 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMUACR_EL1 are UNDEFINED.

Attributes

PMUACR_EL1 is a 64-bit register.

Field descriptions

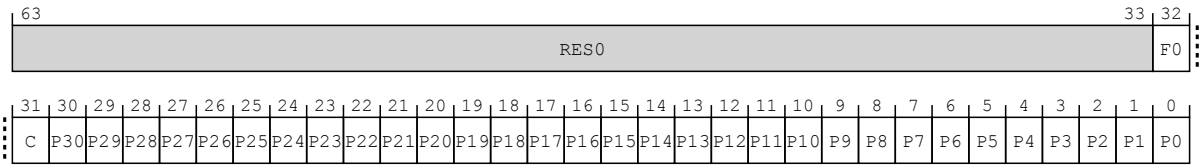

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

EL0 accesses to PMICNTR_EL0 enable. With PMUSERENR_EL0.UEN and PMUSERENR_EL0.IR, controls EL0 accesses to PMICNTR_EL0.

F0	Meaning
0b0	If the Effective value of PMUSERENR_EL0.UEN is 1, then EL0 accesses to PMICNTR_EL0 and associated controls are RAZ/WI, and permitted EL0 writes to PMZR_EL0.F0 are ignored.
0b1	If the Effective value of PMUSERENR_EL0.UEN is 1, then EL0 accesses to PMICNTR_EL0 and associated controls are either read-only or read/write, depending on the value of PMUSERENR_EL0.IR.

The controls associated with PMICNTR_EL0 that are accessible at EL0 are PMCNTENSET_EL0.F0, PMCNTENCLR_EL0.F0, PMOVSSET_EL0.F0, and PMOVSCLR_EL0.F0.

This field is ignored by the PE when any of the following are true:

- EL1 is using AArch32.
- PMUSERENR_EL0.UEN is 0.
- The access generates an exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

C, bit [31]

EL0 accesses to PMCCNTR_EL0 enable. With PMUSERENR_EL0.EN, PMUSERENR_EL0.UEN, and PMUSERENR_EL0.CR, controls EL0 accesses to PMCCNTR_EL0.

C	Meaning
0b0	If the Effective value of PMUSERENR_EL0.UEN is 1, then EL0 accesses to PMCCNTR_EL0 and associated controls are RAZ/WI, and permitted EL0 writes to PMZR_EL0.C are ignored.
0b1	If the Effective value of PMUSERENR_EL0.UEN is 1, then EL0 accesses to PMCCNTR_EL0 and associated controls are either read-only or read/write, depending on the value of PMUSERENR_EL0.CR.

The controls associated with PMCCNTR_EL0 that are accessible at EL0 are PMCNTENSET_EL0.C, PMCNTENCLR_EL0.C, PMOVSSET_EL0.C, and PMOVSCLR_EL0.C.

This field is ignored by the PE when any of the following are true:

- EL1 is using AArch32.
- PMUSERENR_EL0.UEN is 0.
- The access generates an exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

EL0 accesses to PMEVCNTR<m>_EL0 enable. With PMUSERENR_EL0.EN, PMUSERENR_EL0.UEN, and PMUSERENR_EL0.ER, controls EL0 accesses to PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	If the Effective value of PMUSERENR_EL0.UEN is 1, then EL0 accesses to PMEVCNTR<m>_EL0 and associated controls are RAZ/WI, and permitted EL0 writes to PMZR_EL0.P<m> are ignored.
0b1	If the Effective value of PMUSERENR_EL0.UEN is 1, then EL0 accesses to PMEVCNTR<m>_EL0 and associated controls are either read-only or read/write, depending on the value of PMUSERENR_EL0.ER.

The controls associated with PMEVCNTR<m>_EL0 that are accessible at EL0 are PMCNTENSET_EL0.P<m>, PMCNTENCLR_EL0.P<m>, PMOVSSET_EL0.P<m>, and PMOVSCLR_EL0.P<m>.

This field is ignored by the PE when any of the following are true:

- EL1 is using AArch32.
- PMUSERENR_EL0.UEN is 0.
- The access generates an exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMUACR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMUACR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b100

```
if !(IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMUACR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMUACR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMUACR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMUACR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR PMUACR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b100

```
if !(IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nPMUACR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMUACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMUACR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMUACR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.25 PMUSERENR_EL0, Performance Monitors User Enable Register

The PMUSERENR_EL0 characteristics are:

Purpose

Enables or disables EL0 access to the Performance Monitors.

Configuration

AArch64 System register PMUSERENR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMUSERENR[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMUSERENR_EL0 are UNDEFINED.

Attributes

PMUSERENR_EL0 is a 64-bit register.

Field descriptions

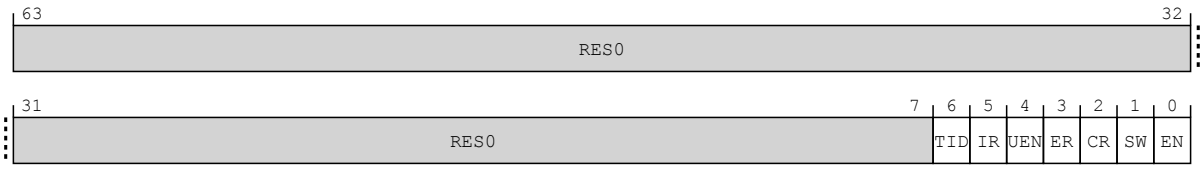

Bits [63:7]

Reserved, RES0.

TID, bit [6]

When FEAT_PMUv3p9 is implemented:

Trap ID registers. Traps EL0 read access to common event identification registers.

TID	Meaning
0b0	Accesses to PMCEID<n>_EL0 and PMCEID<n> are not trapped by this mechanism.
0b1	EL0 read accesses to PMCEID<n>_EL0 and PMCEID<n> are trapped.

In AArch64 state, the register accesses affected by this control are:

- MRS reads of PMCEID0_EL0 and PMCEID1_EL0.

In AArch32 state, the register accesses affected by this control are:

- MRC reads of PMCEID0, PMCEID1, PMCEID2, and PMCEID3.

When trapped, reads generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS reads are reported using EC syndrome value 0x18.
- AArch32 MRC reads are reported using EC syndrome value 0x03.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IR, bit [5]

When FEAT_PMUv3_ICNTR is implemented:

Instruction counter Read-only. When PMUSERENR_EL0.UEN is 1, controls whether EL0 writes to instruction counter are ignored.

IR	Meaning
0b0	EL0 accesses are not affected by this mechanism.
0b1	If the Effective value of PMUSERENR_EL0.UEN is 1, then all of the following apply at EL0: • Writes to PMICNTR_EL0 are ignored. • The controls associated with instruction counter are RAZ/WI. • Writes to PMZR_EL0.F0 are ignored.

The controls associated with PMICNTR_EL0 that are accessible at EL0 are PMCNTENSET_EL0.F0, PMCNTENCLR_EL0.F0, PMOVSSET_EL0.F0, and PMOVSCLR_EL0.F0.

Ignored writes are not trapped and do not generate an exception.

This field is ignored by the PE when any of the following are true:

- PMUSERENR_EL0.UEN is 0.
- The access generates an exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UEN, bit [4]

When FEAT_PMUv3p9 is implemented:

User Enable, with access controlled by PMUACR_EL1. Enables EL0 read/write access to PMU registers, other than PMCR_EL0.

UEN	Meaning
0b0	If FEAT_PMUv3_ICNTR is implemented, then EL0 accesses to PMICFILTR_EL0 and PMICNTR_EL0 are trapped. EL0 accesses to the other specified PMU registers, PMCR_EL0, and PMCR are trapped, unless enabled by PMUSERENR_EL0.{ER,CR,SW,EN}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UEN	Meaning
0b1	EL0 accesses to the specified PMU registers are enabled, unless trapped by another control. The behavior of permitted accesses is controlled by PMUSERENR_EL0.{ IR, ER,CR,SW} and PMUACR_EL1. EL0 accesses to PMCR_EL0 and PMCR are trapped.

In AArch64 state, the register accesses affected by this control are:

- MRS or MSR accesses to the following registers:
  - PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMXEVCNTR_EL0, and PMXEVTYPER_EL0.
  - If FEAT_PMUv3_ICNTR is implemented, PMICFILTR_EL0 and PMICNTR_EL0.
- MRS reads of PMCEID0_EL0 and PMCEID1_EL0.
- MSR writes to PMSWINC_EL0 and PMZR_EL0.

In AArch32 state, the register accesses affected by this control are:

- MRC or MCR accesses to PMCCFILTR, PMCCNTR, PMCNTENCLR, PMCNTENSET, PMEVCNTR<n>, PMEVTYPER<n>, PMOVSR, PMOVSSET, PMSELR, PMXEVCNTR, and PMXEVTYPER.
- MRC reads of PMCEID0, PMCEID1, PMCEID2, and PMCEID3.
- MCR writes to PMSWINC.
- MRRC or MCRR accesses to PMCCNTR.

When trapped, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS and MSR accesses are reported using EC syndrome value 0x18.
- AArch32 MRC and MCR accesses are reported using EC syndrome value 0x03.
- AArch32 MRRC and MCRR accesses are reported using EC syndrome value 0x04.

This field is ignored by the PE and treated as zero when EL1 is using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ER, bit [3]

When FEAT_PMUv3p9 is implemented:

Event counters Read enable or Read-only.

When PMUSERENR_EL0.{UEN,EN} is {0,0}, PMUSERENR_EL0.ER enables EL0 reads of the event counters and EL0 reads and writes of the select register.

When PMUSERENR_EL0.UEN is 1, EL0 reads of the event counters and EL0 writes to PMZR_EL0 are enabled by PMUSERENR_EL0.UEN, unless trapped by another control, and PMUSERENR_EL0.ER controls the behavior of EL0 writes to the event counters and PMZR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ER	Meaning
0b0	When PMUSERENR_EL0.UEN == 0, EL0 reads of the event counters and EL0 reads and writes of the select register are disabled, unless enabled by PMUSERENR_EL0.EN. When PMUSERENR_EL0.UEN == 1, permitted EL0 writes are not affected by this mechanism.
0b1	When PMUSERENR_EL0.UEN == 0, EL0 reads of the event counters and EL0 reads and writes of the select register are enabled, unless trapped by another control. When PMUSERENR_EL0.UEN == 1, permitted EL0 writes to the event counters and PMZR_EL0.P[30:0] are ignored.

In AArch64 state, the register accesses affected by this control are:

- When PMUSERENR_EL0.{UEN,EN} is {0,0}:
  - MRS reads of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
  - MRS and MSR accesses to PMSELR_EL0.
- When PMUSERENR_EL0.UEN is 1, MSR writes to PMZR_EL0.P<n>, PMEVCNTR<n>_EL0, and PMXEVCNTR_EL0.

In AArch32 state, the register accesses affected by this control are:

- When PMUSERENR_EL0.{UEN,EN} is {0,0}:
  - MRC reads of PMEVCNTR<n> and PMXEVCNTR.
  - MRC and MCR accesses to PMSELR.
- When PMUSERENR_EL0.UEN is 1, MCR writes to PMEVCNTR<n> and PMXEVCNTR.

When disabled, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS and MSR accesses are reported using EC syndrome value 0x18.
- AArch32 MRC and MCR accesses are reported using EC syndrome value 0x03.

Ignored writes are not trapped and do not generate an exception.

This field is ignored by the PE when PMUSERENR_EL0.{UEN,EN} == {0,1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Event counters Read enable.

When PMUSERENR_EL0.EN is 0, PMUSERENR_EL0.ER enables EL0 reads of the event counters and EL0 reads and writes of the select register.

ER	Meaning
0b0	EL0 reads of the event counters and EL0 reads and writes of the select register are disabled, unless enabled by PMUSERENR_EL0.EN.
0b1	EL0 reads of the event counters and EL0 reads and writes of the select register are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MRS reads of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
- MRS and MSR accesses to PMSELR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In AArch32 state, the register accesses affected by this control are:

- `MRC` reads of PMEVCNTR<n> and PMXEVCNTR.
- `MRC` and `MCR` accesses to PMSELR.

When disabled, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 `MRS` and `MSR` accesses are reported using EC syndrome value `0x18`.
- AArch32 `MRC` and `MCR` accesses are reported using EC syndrome value `0x03`.

This field is ignored by the PE when PMUSERENR_EL0.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CR, bit [2]

When FEAT_PMUv3p9 is implemented:

Cycle counter Read enable or Read-only.

When PMUSERENR_EL0.{UEN,EN} is {0,0}, PMUSERENR_EL0.CR enables EL0 reads of the cycle counter.

When PMUSERENR_EL0.UEN is 1, EL0 reads of the cycle counter and EL0 writes to PMZR_EL0 are enabled by PMUSERENR_EL0.UEN, unless trapped by another control, and PMUSERENR_EL0.CR controls the behavior of EL0 writes to the cycle counter and PMZR_EL0.

CR	Meaning
`0b0`	When PMUSERENR_EL0.UEN == 0, EL0 reads of the cycle counter are disabled, unless enabled by PMUSERENR_EL0.EN. When PMUSERENR_EL0.UEN == 1, permitted EL0 writes are not affected by this mechanism.
`0b1`	When PMUSERENR_EL0.UEN == 0, EL0 reads of the cycle counter are enabled, unless trapped by another control. When PMUSERENR_EL0.UEN == 1, permitted EL0 writes to the cycle counter and PMZR_EL0.C are ignored.

In AArch64 state, the register accesses affected by this control are:

- When PMUSERENR_EL0.{UEN,EN} is {0,0}, `MRS` reads of PMCCNTR_EL0.
- When PMUSERENR_EL0.UEN is 1, `MSR` writes to PMZR_EL0.C and PMCCNTR_EL0.

In AArch32 state, the register accesses affected by this control are:

- When PMUSERENR_EL0.{UEN,EN} is {0,0}:
  - `MRC` reads of PMCCNTR.
  - `MRRC` reads of PMCCNTR.
- When PMUSERENR_EL0.UEN is 1:
  - `MCR` writes to PMCCNTR.
  - `MCRR` writes to PMCCNTR.

When disabled, reads generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 `MRS` reads are reported using EC syndrome value `0x18`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32 `MRC` reads are reported using EC syndrome value `0x03`.
- AArch32 `MRRC` reads are reported using EC syndrome value `0x04`.

Ignored writes are not trapped and do not generate an exception.

This field is ignored by the PE when PMUSERENR_EL0.{UEN,EN} == {0,1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Cycle counter Read enable.

When PMUSERENR_EL0.EN is 0, PMUSERENR_EL0.CR enables EL0 reads of the cycle counter.

CR	Meaning
`0b0`	EL0 reads of the cycle counter are disabled, unless enabled by PMUSERENR_EL0.EN.
`0b1`	EL0 reads of the cycle counter are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- `MRS` reads of PMCCNTR_EL0.

In AArch32 state, the register accesses affected by this control are:

- `MRC` reads of PMCCNTR.
- `MRRC` reads of PMCCNTR.

When disabled, reads generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 `MRS` reads are reported using EC syndrome value `0x18`.
- AArch32 `MRC` reads are reported using EC syndrome value `0x03`.
- AArch32 `MRRC` reads are reported using EC syndrome value `0x04`.

This field is ignored by the PE when PMUSERENR_EL0.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SW, bit [1]

When FEAT_PMUv3p9 is implemented:

Software increment register Write enable.

When PMUSERENR_EL0.UEN is 0, PMUSERENR_EL0.SW enables EL0 writes to the Software increment register.

When PMUSERENR_EL0.UEN is 1, EL0 writes to the Software increment register are enabled by PMUSERENR_EL0.UEN, unless trapped by another control, and PMUSERENR_EL0.SW controls the behavior of EL0 writes to the Software increment register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SW	Meaning
0b0	When PMUSERENR_EL0.UEN == 0, EL0 writes to the Software increment register are disabled, unless enabled by PMUSERENR_EL0.EN. When PMUSERENR_EL0.UEN == 1, permitted EL0 writes are not affected by this mechanism.
0b1	When PMUSERENR_EL0.UEN == 0, EL0 writes to the Software increment register are enabled, unless trapped by another control. When PMUSERENR_EL0.UEN == 1, permitted EL0 writes to the Software increment register ignore the value of PMUACR_EL1.

In AArch64 state, the register accesses affected by this control are:

- MSR writes to PMSWINC_EL0.

In AArch32 state, the register accesses affected by this control are:

- MCR writes to PMSWINC.

When disabled, writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MSR writes are reported using EC syndrome value 0x18.
- AArch32 MCR writes are reported using EC syndrome value 0x03.

This field is ignored by the PE when PMUSERENR_EL0.{UEN,EN} == {0,1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Software increment register Write enable.

When PMUSERENR_EL0.EN is 0, PMUSERENR_EL0.SW enables EL0 writes to the Software increment register.

SW	Meaning
0b0	EL0 writes to the Software increment register are disabled, unless enabled by PMUSERENR_EL0.EN.
0b1	EL0 writes to the Software increment register are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MSR writes to PMSWINC_EL0.

In AArch32 state, the register accesses affected by this control are:

- MCR writes to PMSWINC.

When disabled, writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MSR writes are reported using EC syndrome value 0x18.
- AArch32 MCR writes are reported using EC syndrome value 0x03.

This field is ignored by the PE when PMUSERENR_EL0.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EN, bit [0]

Enable.

Enables EL0 read/write access to PMU registers, other than the instruction counter.

EN	Meaning
0b0	EL0 accesses to the specified PMU System registers are trapped, unless enabled by PMUSERENR_EL0.{UEN,ER,CR,SW}.
0b1	EL0 accesses to the specified PMU System registers are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MRS or MSR accesses to PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMXEVCNTR_EL0, and PMXEVTYPER_EL0.
- MRS reads of PMCEID0_EL0 and PMCEID1_EL0.
- MSR writes to the following registers:
  - PMSWINC_EL0.
  - If FEAT_PMUv3p9 is implemented, PMZR_EL0.

Note

When FEAT_PMUv3_ICNTR is implemented, this field does not affect MRS and MSR accesses to PMICNTR_EL0 and PMICFILTR_EL0.

In AArch32 state, the register accesses affected by this control are:

- MRC or MCR accesses to PMCCFILTR, PMCCNTR, PMCNTENCLR, PMCNTENSET, PMCR, PMEVCNTR<n>, PMEVTYPER<n>, PMOVSR, PMOVSSET, PMSELR, PMXEVCNTR, and PMXEVTYPER.
- MRC reads of the following registers:
  - PMCEID0 and PMCEID1.
  - If FEAT_PMUv3p1 is implemented, PMCEID2 and PMCEID3.
- MCR writes to PMSWINC.
- MRRC or MCRR accesses to PMCCNTR.

When trapped, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS and MSR accesses are reported using EC syndrome value 0x18.
- AArch32 MRC and MCR accesses are reported using EC syndrome value 0x03.
- AArch32 MRRC and MCRR accesses are reported using EC syndrome value 0x04.

This field is ignored by the PE when FEAT_PMUv3p9 is implemented and PMUSERENR_EL0.UEN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMUSERENR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMUSERENR_EL0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMUSERENR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMUSERENR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMUSERENR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMUSERENR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMUSERENR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMUSERENR_EL0();
end;
```

```
MSR PMUSERENR_EL0, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b000

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMUSERENR_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMUSERENR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMUSERENR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMUSERENR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.26 PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register

The PMXEVCNTR_EL0 characteristics are:

Purpose

Reads or writes the value of the selected event counter, PMEVCNTR<n>_EL0. PMSELR_EL0.SEL determines which event counter is selected.

Configuration

AArch64 System register PMXEVCNTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMXEVCNTR[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMXEVCNTR_EL0 are UNDEFINED.

Attributes

PMXEVCNTR_EL0 is a 64-bit register.

Field descriptions

When FEAT_PMUv3p5 is implemented:

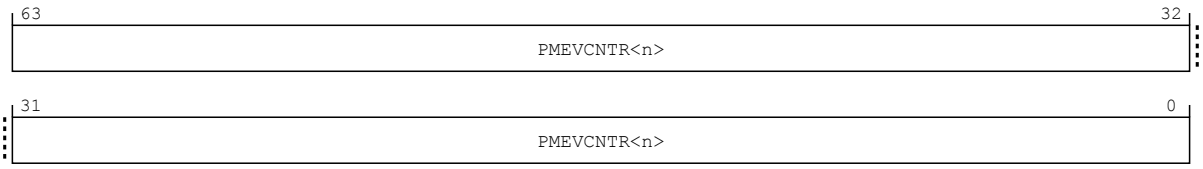

PMEVCNTR<n>, bits [63:0]

Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

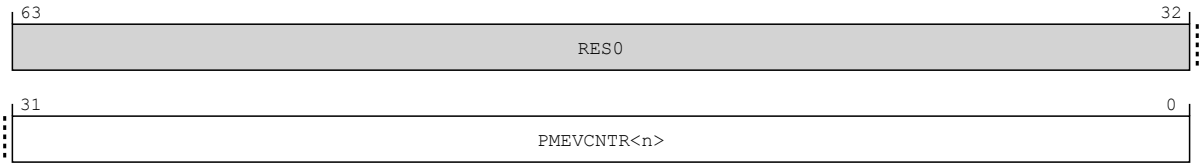

Bits [63:32]

Reserved, RES0.

PMEVCNTR<n>, bits [31:0]

Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMXEVCNTR_EL0

If FEAT_FGT is implemented and PMSELR_EL0.SEL is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVCNTR_EL0 is as follows:

- If PMSELR_EL0.SEL is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and PMSELR_EL0.SEL is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMXEVCNTR_EL0 are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.P<UInt(PMSELR_EL0.SEL)> is 0.

Permitted writes of PMXEVCNTR_EL0 are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.{UEN,ER} is {1,1}.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{UEN,ER,EN}.
If EL2 is implemented and enabled in the current Security state, at EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters.
Otherwise, the number of accessible event counters is determined by the Effective value of PMCCR.EPMN. For more information, see MDCR_EL2.HPMN and PMCCR.EPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMXEVCNTR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b010

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif UInt(PMSELR_EL0().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        elsif (IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
             ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00') then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
             ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMEVCNTRn_EL0 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            else
                AArch64_SystemAccessTrap(EL2, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' &&
             ↪PMUACR_EL1()[UInt(PMSELR_EL0().SEL)] == '0' then
            X{64}(t) = Zeros{64};
        else
            X{64}(t) = PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL));
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().PMEVCNTRn_EL0 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            else
                AArch64_SystemAccessTrap(EL2, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL));
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL));
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL));
end;
```

```
MSR PMXEVCNTR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b010

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif UInt(PMSELR_EL0().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMEVCNTRn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' &&
         ↪(PMUACR_EL1()[UInt(PMSELR_EL0().SEL)] == '0' || PMUSERENR_EL0().ER == '1') then
        return;
    else
        PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMEVCNTRn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMEVCNTR_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.27 PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register

The PMXEVTYPER_EL0 characteristics are:

Purpose

When PMSELR_EL0.SEL selects an event counter, this accesses a PMEVTYPER<n>_EL0 register. When PMSELR_EL0.SEL selects the cycle counter, this accesses PMCCFILTR_EL0.

Configuration

AArch64 System register PMXEVTYPER_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMXEVTYPER[31:0].

This register is present only when FEAT_PMUv3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMXEVTYPER_EL0 are UNDEFINED.

Attributes

PMXEVTYPER_EL0 is a 64-bit register.

Field descriptions

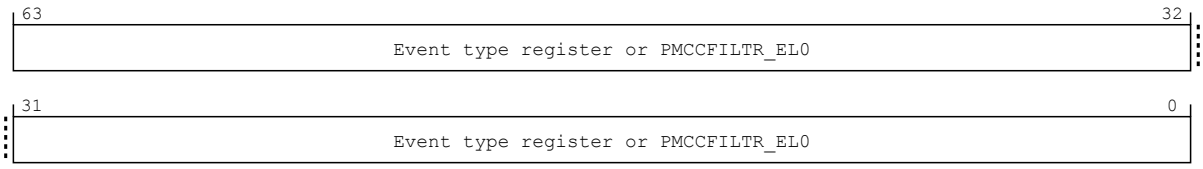

EVTYPERn, bits [63:0]

When PMSELR_EL0.SEL == 31, this register accesses PMCCFILTR_EL0.

Otherwise, this register accesses PMEVTYPER<n>_EL0 where n is the value in PMSELR_EL0.SEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMXEVTYPER_EL0

If FEAT_FGT is implemented, and PMSELR_EL0.SEL is not 31 and is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVTYPER_EL0 is as follows:

- If PMSELR_EL0.SEL is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented, and PMSELR_EL0.SEL is not 31 and is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVTYPER_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR_EL0.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, PMSELR_EL0 is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Permitted reads and writes of PMXEVTYPER_EL0 are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.UEN is 1.
- Any of the following are true:
  - PMSELR_EL0.SEL is not 31 and PMUACR_EL1.P<UInt(PMSELR_EL0.SEL)> is 0.
  - PMSELR_EL0.SEL is 31 and PMUACR_EL1.C is 0.

Permitted writes of PMXEVTYPER_EL0 are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- PMUSERENR_EL0.UEN is 1.
- Any of the following are true:
  - PMSELR_EL0.SEL is not 31 and PMUSERENR_EL0.ER is 1.
  - PMSELR_EL0.SEL is 31 and PMUSERENR_EL0.CR is 1.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{UEN,EN}.
If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters.
Otherwise, the number of accessible event counters is determined by the Effective value of PMCCR.EPMN. For more information, see MDCR_EL2.HPMN and PMCCR.EPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMXEVTYPER_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b001

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif UInt(PMSELR_EL0().SEL) != 31 && UInt(PMSELR_EL0().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGRTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) != 31 && UInt(PMSELR_EL0().SEL) >=
         ↪GetNumEventCountersAccessible() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            else
                AArch64_SystemAccessTrap(EL2, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && ((UInt(PMSELR_EL0().SEL)
             ↪!= 31 && PMUACR_EL1()[UInt(PMSELR_EL0().SEL)] == '0') || (UInt(PMSELR_EL0().SEL) == 31 &&
             ↪PMUACR_EL1().C == '0')) then
            X{64}(t) = Zeros{64};
        elsif UInt(PMSELR_EL0().SEL) == 31 then
            X{64}(t) = PMCCFILTR_EL0();
        else
            X{64}(t) = PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL));
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().PMEVTYPERn_EL0 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) != 31 && UInt(PMSELR_EL0().SEL) >=
             ↪GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            else
                AArch64_SystemAccessTrap(EL2, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif UInt(PMSELR_EL0().SEL) == 31 then
            X{64}(t) = PMCCFILTR_EL0();
        else
            X{64}(t) = PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL));
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif UInt(PMSELR_EL0().SEL) == 31 then
            X{64}(t) = PMCCFILTR_EL0();
        else
            X{64}(t) = PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL));
        end;
    elsif PSTATE.EL == EL3 then
        if UInt(PMSELR_EL0().SEL) == 31 then
            X{64}(t) = PMCCFILTR_EL0();
        else
            X{64}(t) = PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL));
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

```
MSR PMXEVTYPER_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b001

```
if !(IsFeatureImplemented(FEAT_PMUv3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif UInt(PMSELR_EL0().SEL) != 31 && UInt(PMSELR_EL0().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HDFGWTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) != 31 && UInt(PMSELR_EL0().SEL) >=
         ↪GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1' && ((UInt(PMSELR_EL0().SEL)
         ↪!= 31 && (PMUACR_EL1()[UInt(PMSELR_EL0().SEL)] == '0' || PMUSERENR_EL0().ER == '1')) ||
         ↪(UInt(PMSELR_EL0().SEL) == 31 && (PMUACR_EL1().C == '0' || PMUSERENR_EL0().CR == '1'))) then
        return;
    elsif UInt(PMSELR_EL0().SEL) == 31 then
        PMCCFILTR_EL0() = X{64}(t);
    else
        PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && UInt(PMSELR_EL0().SEL) != 31 && UInt(PMSELR_EL0().SEL) >=
         ↪GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif UInt(PMSELR_EL0().SEL) == 31 then
        PMCCFILTR_EL0() = X{64}(t);
    else
        PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif UInt(PMSELR_EL0().SEL) == 31 then
        PMCCFILTR_EL0() = X{64}(t);
    else
        PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if UInt(PMSELR_EL0().SEL) == 31 then
        PMCCFILTR_EL0() = X{64}(t);
    else
        PMEVTYPER_EL0(UInt(PMSELR_EL0().SEL)) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.28 PMZR_EL0, Performance Monitors Zero with Mask

The PMZR_EL0 characteristics are:

Purpose

Zero the set of counters specified by the mask written to PMZR_EL0.

Configuration

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is implemented, AArch64 System register PMZR_EL0 bits [63:0] are architecturally mapped to External register PMZR_EL0[63:0].

This register is present only when FEAT_PMUv3p9 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to PMZR_EL0 are UNDEFINED.

Attributes

PMZR_EL0 is a 64-bit register.

Field descriptions

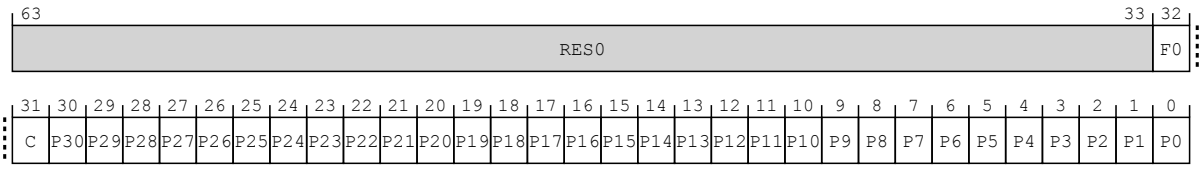

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Zero PMICNTR_EL0.

F0	Meaning
0b0	Write is ignored.
0b1	Set PMICNTR_EL0 to zero.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - PSTATE.EL != EL3
  - MDCR_EL3.EnPM2 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - EL0 is the current Exception level
  - Any of the following are true:
    - PMUSERENR_EL0.UEN == ‘0’
    - PMUACR_EL1.F0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_FGT2 is implemented
- EL2Enabled()
- PSTATE.EL IN {EL1, EL0}
- [HCR_EL2.E2H, HCR_EL2.TGE] != ‘11’
- Any of the following are true:
  - All of the following are true:
    - EL3 is implemented
    - SCR_EL3.FGTEn2 == ‘0’
  - HDFGWTR2_EL2.nPMICNTR_EL0 == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - EL0 is the current Exception level
  - PMUSERENR_EL0.IR == ‘1’
- Otherwise, access to this field is WO/RAZ.

Otherwise:

Reserved, RES0.

C, bit [31]

Zero PMCCNTR_EL0.

C	Meaning
0b0	Write is ignored.
0b1	Set PMCCNTR_EL0 to zero.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.C == ‘0’
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == ‘11’
- Otherwise, access to this field is WO/RAZ.

P<m> , bits [m], for m = 30 to 0

Zero PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	Write is ignored.
0b1	Set PMEVCNTR<m>_EL0 to zero.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL0 is the current Exception level
- PMUSERENR_EL0.UEN == ‘1’
- PMUACR_EL1.P<m> == ‘0’

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == ‘11’
- Otherwise, access to this field is WO/RAZ.

Accessing PMZR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MSR PMZR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b100

```
if !(IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif PMUSERENR_EL0().EN == '0' && (!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN ==
         ↪'0') then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nPMZR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ZeroPMUCounters(X{64}(t));
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nPMZR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ZeroPMUCounters(X{64}(t));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().TPM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ZeroPMUCounters(X{64}(t));
    end;
elsif PSTATE.EL == EL3 then
    ZeroPMUCounters(X{64}(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.29 SPMACCESSR_EL1, System Performance Monitors Access Register (EL1)

The SPMACCESSR_EL1 characteristics are:

Purpose

Controls access to System PMUs from EL0.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMACCESSR_EL1 are UNDEFINED.

Attributes

SPMACCESSR_EL1 is a 64-bit register.

Field descriptions

63 62	61 60	59 58	57 56	55 54	53 52	51 50	49 48	47 46	45 44	43 42	41 40	39 38	37 36	35 34	33 32
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [2m+1:2m], for m = 31 to 0

System PMU <m> access. Controls access to System PMU <m>.

P<m>	Meaning
0b00	MRS read and MSR write System register accesses to System PMU <m> at EL0 are trapped to EL1, unless the instruction generates a higher priority exception.
0b01	MSR write System register accesses to System PMU <m> at EL0 are trapped to EL1, unless the instruction generates a higher priority exception.
0b11	This control does not cause any instructions to be trapped.

All other values are reserved.

The registers trapped by this control are:

AArch64: SPMCNTENCLR_EL0, SPMCNTENSET_EL0, SPMCR_EL0, SPMEVCNTR<n>_EL0, SPMEVFILT2R<n>_EL0, SPMEVFILTR<n>_EL0, SPMEVTYPER<n>_EL0, SPMOVSCLR_EL0, and SPMOVSSET_EL0.

This field is ignored by the PE when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The Effective value of HCR_EL2.{E2H,TGE} is {1,1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m > UInt(ID_AA64DFR1_EL1.SYSPMUID), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SPMACCESSR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `SPMACCESSR_EL1` or `SPMACCESSR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMACCESSR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMACCESSR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8E8);
    else
        X{64}(t) = SPMACCESSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SPMACCESSR_EL2();
    else
        X{64}(t) = SPMACCESSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMACCESSR_EL1();
end;
```

```
MSR SPMACCESSR_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMACCESSR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8E8) = X{64}(t);
    else
        SPMACCESSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SPMACCESSR_EL2() = X{64}(t);
    else
        SPMACCESSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMACCESSR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, SPMACCESSR_EL12

op0	op1	CRn	CRm	op2
0b10	0b101	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x8E8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
                Undefined();
            elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                X{64}(t) = SPMACCESSR_EL1();
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = SPMACCESSR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR SPMACCESSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b101	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x8E8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMACCESSR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        SPMACCESSR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.30 SPMACCESSR_EL2, System Performance Monitors Access Register (EL2)

The SPMACCESSR_EL2 characteristics are:

Purpose

Controls access to System PMUs from EL1 and EL0.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMACCESSR_EL2 are UNDEFINED.

Attributes

SPMACCESSR_EL2 is a 64-bit register.

Field descriptions

63 62	61 60	59 58	57 56	55 54	53 52	51 50	49 48	47 46	45 44	43 42	41 40	39 38	37 36	35 34	33 32
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [2m+1:2m], for m = 31 to 0

System PMU <m> access. Controls access to System PMU <m>.

P<m>	Meaning
0b00	MRS read and MSR write System register accesses to System PMU <m> at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.
0b01	MSR write System register accesses to System PMU <m> at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.
0b11	This control does not cause any instructions to be trapped.

All other values are reserved.

The registers trapped by this control are:

AArch64: SPMCFGR_EL1, SPMCGCR<n>_EL1, SPMCNTENCLR_EL0, SPMCNTENSET_EL0, SPMCR_EL0, SPMDEVAFF_EL1, SPMDEVARCH_EL1, SPMEVCNTR<n>_EL0, SPMEVFILT2R<n>_EL0, SPMEVFILTR<n>_EL0, SPMEVTYPER<n>_EL0, SPMIIDR_EL1, SPMINTENCLR_EL1, SPMINTENSET_EL1, SPMOVSCLR_EL0, SPMOVSSET_EL0, and SPMSCR_EL1.

This field is ignored by the PE when EL2 is not implemented or disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m > UInt(ID_AA64DFR1_EL1.SYSPMUID), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SPMACCESSR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `SPMACCESSR_EL2` or `SPMACCESSR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMACCESSR_EL2
```

op0	op1	CRn	CRm	op2
0b10	0b100	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMACCESSR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMACCESSR_EL2();
end;
```

```
MSR SPMACCESSR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b100	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMACCESSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMACCESSR_EL2() = X{64}(t);
end;
```

MRS <Xt>, SPMACCESSR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMACCESSR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8E8);
    else
        X{64}(t) = SPMACCESSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = SPMACCESSR_EL2();
    else
        X{64}(t) = SPMACCESSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMACCESSR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SPMACCESSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMACCESSR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8E8) = X{64}(t);
    else
        SPMACCESSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        SPMACCESSR_EL2() = X{64}(t);
    else
        SPMACCESSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMACCESSR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.31 SPMACCESSR_EL3, System Performance Monitors Access Register (EL3)

The SPMACCESSR_EL3 characteristics are:

Purpose

Controls access to System PMUs from EL2, EL1 and EL0.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMACCESSR_EL3 are UNDEFINED.

Attributes

SPMACCESSR_EL3 is a 64-bit register.

Field descriptions

63 62	61 60	59 58	57 56	55 54	53 52	51 50	49 48	47 46	45 44	43 42	41 40	39 38	37 36	35 34	33 32
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [2m+1:2m], for m = 31 to 0

System PMU <m> access. Controls access to System PMU <m>.

P<m>	Meaning
0b00	MRS read and MSR write System register accesses to System PMU <m> at EL2, EL1, and EL0 are trapped to EL3, unless the instruction generates a higher priority exception.
0b01	MSR write System register accesses to System PMU <m> at EL2, EL1, and EL0 are trapped to EL3, unless the instruction generates a higher priority exception.
0b11	This control does not cause any instructions to be trapped.

All other values are reserved.

The registers trapped by this control are:

AArch64: SPMCFGR_EL1, SPMCGCR<n>_EL1, SPMCNTENCLR_EL0, SPMCNTENSET_EL0, SPMCR_EL0, SPMDEVAFF_EL1, SPMDEVARCH_EL1, SPMEVCNTR<n>_EL0, SPMEVFILT2R<n>_EL0, SPMEVFILTR<n>_EL0, SPMEVTYPER<n>_EL0, SPMIIDR_EL1, SPMINTENCLR_EL1, SPMINTENSET_EL1, SPMOVSCLR_EL0, SPMOVSSET_EL0, and SPMSCR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m > UInt(ID_AA64DFR1_EL1.SYSPMUID), access to this field is RES0.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SPMACCESSR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMACCESSR_EL3
```

op0	op1	CRn	CRm	op2
0b10	0b110	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMACCESSR_EL3();
end;
```

```
MSR SPMACCESSR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b110	0b1001	0b1101	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    SPMACCESSR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.32 SPMCFGR_EL1, System Performance Monitors Configuration Register

The SPMCFGR_EL1 characteristics are:

Purpose

Describes the capabilities of System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMCFGR_EL1 are UNDEFINED.

Attributes

SPMCFGR_EL1 is a 64-bit register.

Field descriptions

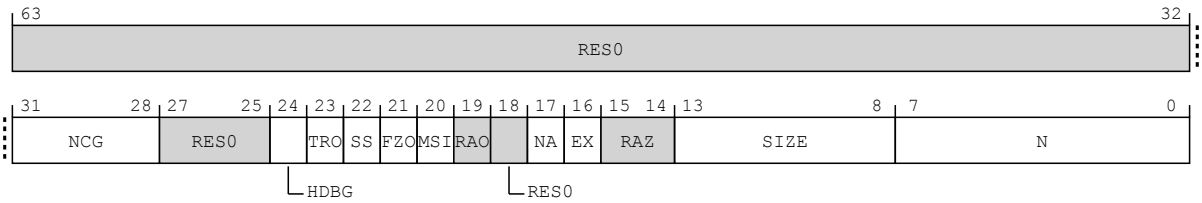

Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Counter Groups.

Defines the number of counter groups implemented by System PMU <s>, minus one.

If this field is zero, then one counter group is implemented and SPMCGCR<n>_EL1 read-as-zero.

Otherwise, for each counter group <m>, SPMCGCR<m DIV 8>_EL1.N<m MOD 8> defines the number of counters in the group.

Locating the first counter in each group depends on the number of implemented groups. Each counter group starts with counter:

- SPMEVTYPER<m×32>_EL0, meaning there are at most 32 counters per group, if there are 2 counter groups.
- SPMEVTYPER<m×16>_EL0, meaning there are at most 16 counters per group, if there are 3 or 4 counter groups.
- SPMEVTYPER<m×8>_EL0, meaning there are at most 8 counters per group, if there are between 5 and 8 counter groups.
- SPMEVTYPER<m×4>_EL0, meaning there are at most 4 counters per group, if there are more than 8 counter groups.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [27:25]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HDBG, bit [24]

Halt-on-debug supported. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

TRO, bit [23]

Trace output supported. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SS, bit [22]

Snapshot supported. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

FZO, bit [21]

Freeze-on-overflow supported. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

MSI, bit [20]

Message-signaled interrupts supported. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [19]

Reserved, RAO.

Bit [18]

Reserved, RES0.

NA, bit [17]

No write access when running. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

EX, bit [16]

Export supported. For more information on this field, see ‘CoreSight PMU Architecture’.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [15:14]

Reserved, RAZ.

SIZE, bits [13:8]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Counter size. The size of the largest counter implemented by System PMU <s>.

SIZE	Meaning
0b000111	8-bit counters.
0b001001	10-bit counters.
0b001011	12-bit counters.
0b001111	16-bit counters.
0b010011	20-bit counters.
0b010111	24-bit counters.
0b011111	32-bit counters.
0b100011	36-bit counters.
0b100111	40-bit counters.
0b101011	44-bit counters.
0b101111	48-bit counters.
0b110011	52-bit counters.
0b110111	56-bit counters.
0b111111	64-bit counters.

All other values are reserved.

Not all counters must be this size. For example, a System PMU might include a mix of 32-bit and 64-bit counters.

N, bits [7:0]

Number of event counters implemented by System PMU <s>, minus 1.

N	Meaning
0x00..0x3F	Number of event counters implemented by System PMU <s>, minus 1.

All other values are reserved.

Accessing SPMCFGR_EL1

To access SPMCFGR_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMCFGR_EL1 reads-as-zero if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMCFGR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMID == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCFGR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCFGR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMCFGR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.33 SPMCGCR<n>_EL1, System PMU Counter Group Configuration Registers, n = 0 - 1

The SPMCGCR<n>_EL1 characteristics are:

Purpose

Describes the configuration of counter groups in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMCGCR<n>_EL1 are UNDEFINED.

Attributes

SPMCGCR<n>_EL1 is a 64-bit register.

Field descriptions

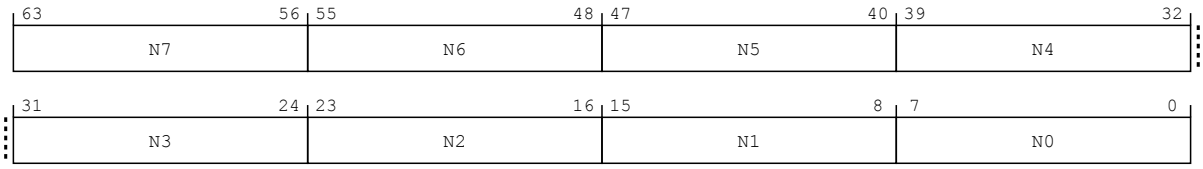

N<m> , bits [8m+7:8m], for m = 7 to 0

Number of counters in counter group 8n+m.

The maximum size of each counter group depends on the number of implemented groups and the largest implemented counter size. For more information, see SPMCFGR_EL1.NCG.

Accessing SPMCGCR<n>_EL1

To access SPMCGCR<n>_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMCGCR<n>_EL1 reads-as-zero if any of the following are true:

- System PMU <s> implements one counter group (SPMCFGR_EL1.NCG is zero).
- System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMCGCR<m>_EL1 ; Where m = 0-1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b00:m[0]

```
let m:integer = UInt(op2[0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMID == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCGCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL), m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCGCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL), m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMCGCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL), m);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.34 SPMCNTENCLR_EL0, System Performance Monitors Count Enable Clear Register

The SPMCNTENCLR_EL0 characteristics are:

Purpose

Disable event counters in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMCNTENCLR_EL0 are UNDEFINED.

Attributes

SPMCNTENCLR_EL0 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Event counter <m> disable.

P<m>	Meaning
0b0	Event counter <m> in System PMU <s> is disabled.
0b1	Event counter <m> in System PMU <s> is enabled.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When event counter <m> is not implemented by System PMU <s>, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing SPMCNTENCLR_EL0

To access SPMCNTENCLR_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMCNTENCLR_EL0 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMCNTENCLR_EL0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b010

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMCNTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMCNTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪== '00' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
```

MSR SPMCNTENCLR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b010

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMCNTEN == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMCNTEN == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMCNTENCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.35 SPMCNTENSET_EL0, System Performance Monitors Count Enable Set Register

The SPMCNTENSET_EL0 characteristics are:

Purpose

Enables event counters in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMCNTENSET_EL0 are UNDEFINED.

Attributes

SPMCNTENSET_EL0 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Event counter <m> enable.

P<m>	Meaning
0b0	Event counter <m> in System PMU <s> is disabled.
0b1	Event counter <m> in System PMU <s> is enabled.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When event counter <m> is not implemented by System PMU <s>, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing SPMCNTENSET_EL0

To access SPMCNTENSET_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMCNTENSET_EL0 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMCNTENSET_EL0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMCNTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMCNTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪== '00' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
```

MSR SPMCNTENSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMCNTEN == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMCNTEN == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMCNTENSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.36 SPMCR_EL0, System Performance Monitor Control Register

The SPMCR_EL0 characteristics are:

Purpose

Main control register for System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMCR_EL0 are UNDEFINED.

Attributes

SPMCR_EL0 is a 64-bit register.

Field descriptions

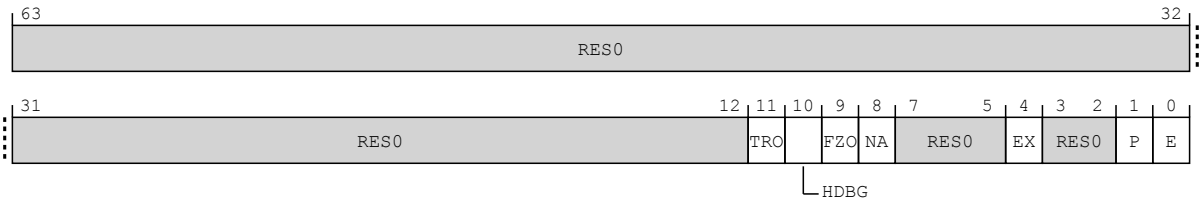

Bits [63:12]

Reserved, RES0.

TRO, bit [11]

When SPMCFGR_EL1.TRO == '1':

Trace enable. For more information on this field, see ‘CoreSight PMU Architecture’.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When EL0 is the current Exception level, access to this field is RO.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

HDBG, bit [10]

When SPMCFGR_EL1.HDBG == '1':

Halt-on-debug. For more information on this field, see ‘CoreSight PMU Architecture’.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When EL0 is the current Exception level, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

FZO, bit [9]

When SPMCFGR_EL1.FZO == '1':

Freeze-on-overflow. For more information on this field, see ‘CoreSight PMU Architecture’.

Note

If implemented by a System PMU, then freeze-on-overflow affects only the counters of System PMU <s>, not other System PMUs nor the PE PMU.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NA, bit [8]

When SPMCFGR_EL1.NA == '1':

Not accessible. For more information on this field, see ‘CoreSight PMU Architecture’.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [7:5]

Reserved, RES0.

EX, bit [4]

When SPMCFGR_EL1.EX == '1':

Export enable. For more information on this field, see ‘CoreSight PMU Architecture’.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [3:2]

Reserved, RES0.

P, bit [1]

Event counter reset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P	Meaning
0b0	Write is ignored.
0b1	Reset all event counters in System PMU <s> to zero.

Note

Resetting the event counters does not affect any overflow flags.

Access to this field is WO/RAZ.

E, bit [0]

Count enable. This field controls System PMU <s>.

E	Meaning
0b0	Monitor is disabled.
0b1	Monitor is enabled.

Performance monitor overflow IRQs are only signaled by System PMU <s> when this field is 1.

The reset behavior of this field is:

- On a System PMU reset, this field resets to `'0'`.

Accessing SPMCR_EL0

To access SPMCR_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMCR_EL0 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMCR_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMCR_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪== '00' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGRTR2_EL2().nSPMCR_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪== '00' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```

```
MSR SPMCR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMCR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMCR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMCR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.37 SPMDEVAFF_EL1, System Performance Monitors Device Affinity Register

The SPMDEVAFF_EL1 characteristics are:

Purpose

For additional information, see the CoreSight Architecture Specification.

For a System PMU that has affinity with a single PE or a group of PEs, SPMDEVAFF_EL1 is a copy of MPIDR_EL1 or part of MPIDR_EL1:

- If the System PMU has affinity with a single PE, then the affinity level is 0 and SPMDEVAFF_EL1 reads the same value as MPIDR_EL1, and SPMDEVAFF_EL1.F0V reads-as-one to indicate affinity level 0.
- If the System PMU has affinity with a group of PEs, then the affinity level is 1, 2, or 3, parts of SPMDEVAFF_EL1 reads the same value as parts of MPIDR_EL1, and the rest of SPMDEVAFF_EL1 indicates the level.

For example, if the group of PEs is a subset of the PEs at affinity level 1, then all of the following are true:

- All the PEs in the group have the same values in MPIDR_EL1.{Aff3,Aff2}, and these values are equal to SPMDEVAFF_EL1.{Aff3,Aff2}.
- SPMDEVAFF_EL1.Aff1 is nonzero and not `0x80`, and SPMDEVAFF_EL1.{Aff0,F0V} read-as-zero, to indicate at least affinity level 1. The subset of PEs at level 1 that the System PMU has affinity with is indicated by the least-significant set bit in SPMDEVAFF_EL1.Aff1. In this example, if SPMDEVAFF_EL1.Aff1[2:0] is `0b100`, then the System PMU has affinity with the up-to 8 PEs that have MPIDR_EL1.Aff1[7:3] == SPMDEVAFF_EL1.Aff1[7:3].

Depending on the IMPLEMENTATION DEFINED nature of the system, it might be possible that SPMDEVAFF_EL1 is read before system firmware has configured the System PMU and/or the PE or group of PEs that the System PMU has affinity with. When this is the case, SPMDEVAFF_EL1 reads-as-zero.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMDEVAFF_EL1 are UNDEFINED.

Attributes

SPMDEVAFF_EL1 is a 64-bit register.

Field descriptions

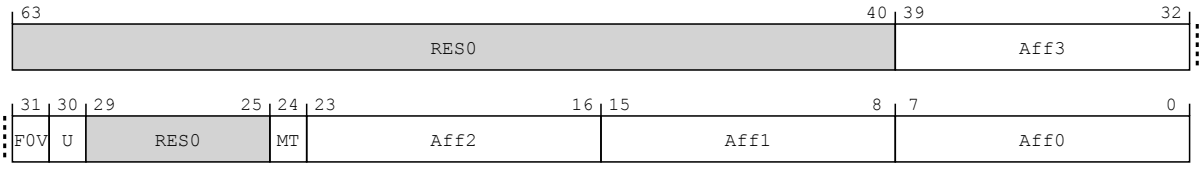

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

PE affinity level 3. The MPIDR_EL1.Aff3 field, viewed from the highest Exception level of the associated PE or PEs.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F0V, bit [31]

Indicates that the SPMDEVAFF_EL1.Aff0 field is valid.

The value of this field is an IMPLEMENTATION DEFINED choice of:

F0V	Meaning
0b0	SPMDEVAFF_EL1.Aff0 is not valid, and the PE affinity is above level 0 or a subset of level 0.
0b1	SPMDEVAFF_EL1.Aff0 is valid, and the PE affinity is at level 0.

Access to this field is RO.

U, bit [30]

When SPMDEVAFF_EL1.F0V == '1':

Uniprocessor. The MPIDR_EL1.U field, viewed from the highest Exception level of the associated PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, UNKNOWN.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

When SPMDEVAFF_EL1.F0V == '1':

Multithreaded. The MPIDR_EL1.MT field, viewed from the highest Exception level of the associated PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, UNKNOWN.

Aff2, bits [23:16]

When affine with a PE or PEs at affinity level 2 or below:

PE affinity level 2. The MPIDR_EL1.Aff2 field, viewed from the highest Exception level of the associated PE or PEs.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

When affine with a sub-set of PEs at affinity level 2:

PE affinity level 2. Defines part of the MPIDR_EL1.Aff2 field, viewed from the highest Exception level of the associated PEs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Aff2	Meaning
0bxxxxxxx1	SPMDEVAFF_EL1.Aff2[7:1] is the value of MPIDR_EL1.Aff2[7:1], viewed from the highest Exception level of the associated PEs.
0bxxxxxx10	SPMDEVAFF_EL1.Aff2[7:2] is the value of MPIDR_EL1.Aff2[7:2], viewed from the highest Exception level of the associated PEs.
0bxxxxx100	SPMDEVAFF_EL1.Aff2[7:3] is the value of MPIDR_EL1.Aff2[7:3], viewed from the highest Exception level of the associated PEs.
0bxxxx1000	SPMDEVAFF_EL1.Aff2[7:4] is the value of MPIDR_EL1.Aff2[7:4], viewed from the highest Exception level of the associated PEs.
0bxxx10000	SPMDEVAFF_EL1.Aff2[7:5] is the value of MPIDR_EL1.Aff2[7:5], viewed from the highest Exception level of the associated PEs.
0bxx100000	SPMDEVAFF_EL1.Aff2[7:6] is the value of MPIDR_EL1.Aff2[7:6], viewed from the highest Exception level of the associated PEs.
0bx1000000	SPMDEVAFF_EL1.Aff2[7] is the value of MPIDR_EL1.Aff2[7], viewed from the highest Exception level of the associated PEs.

Access to this field is RO.

Otherwise:

PE affinity level NOT DEFINED. Indicates whether the PE affinity is at level 3.

Aff2	Meaning
0x80	PE affinity is at level 3.

All other values are reserved.

Access to this field is RO.

Aff1, bits [15:8]

When affine with a PE or PEs at affinity level 1 or below:

PE affinity level 1. The MPIDR_EL1.Aff1 field, viewed from the highest Exception level of the associated PE or PEs.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

When affine with a sub-set of PEs at affinity level 1:

PE affinity level 1. Defines part of the MPIDR_EL1.Aff1 field, viewed from the highest Exception level of the associated PEs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Aff1	Meaning
0bxxxxxxx1	SPMDEVAFF_EL1.Aff1[7:1] is the value of MPIDR_EL1.Aff1[7:1], viewed from the highest Exception level of the associated PEs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Aff1	Meaning
0bxxxxxx10	SPMDEVAFF_EL1.Aff1[7:2] is the value of MPIDR_EL1.Aff1[7:2], viewed from the highest Exception level of the associated PEs.
0bxxxxx100	SPMDEVAFF_EL1.Aff1[7:3] is the value of MPIDR_EL1.Aff1[7:3], viewed from the highest Exception level of the associated PEs.
0bxxxx1000	SPMDEVAFF_EL1.Aff1[7:4] is the value of MPIDR_EL1.Aff1[7:4], viewed from the highest Exception level of the associated PEs.
0bxxx10000	SPMDEVAFF_EL1.Aff1[7:5] is the value of MPIDR_EL1.Aff1[7:5], viewed from the highest Exception level of the associated PEs.
0bxx100000	SPMDEVAFF_EL1.Aff1[7:6] is the value of MPIDR_EL1.Aff1[7:6], viewed from the highest Exception level of the associated PEs.
0bx1000000	SPMDEVAFF_EL1.Aff1[7] is the value of MPIDR_EL1.Aff1[7], viewed from the highest Exception level of the associated PEs.

Access to this field is RO.

Otherwise:

PE affinity level 1. Indicates whether the PE affinity is at level 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Aff1	Meaning
0x00	PE affinity is above level 2 or a subset of level 2.
0x80	PE affinity is at level 2.

Access to this field is RO.

Aff0, bits [7:0]

When affine with a PE at affinity level 0:

PE affinity level 0. The MPIDR_EL1.Aff0 field, viewed from the highest Exception level of the associated PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

When affine with a sub-set of PEs at affinity level 0:

PE affinity level 0. Defines part of the MPIDR_EL1.Aff0 field, viewed from the highest Exception level of the associated PEs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Aff0	Meaning
0bxxxxxxx1	SPMDEVAFF_EL1.Aff0[7:1] is the value of MPIDR_EL1.Aff0[7:1], viewed from the highest Exception level of the associated PEs.
0bxxxxxx10	SPMDEVAFF_EL1.Aff0[7:2] is the value of MPIDR_EL1.Aff0[7:2], viewed from the highest Exception level of the associated PEs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Aff0	Meaning
0bxxxxx100	SPMDEVAFF_EL1.Aff0[7:3] is the value of MPIDR_EL1.Aff0[7:3], viewed from the highest Exception level of the associated PEs.
0bxxxx1000	SPMDEVAFF_EL1.Aff0[7:4] is the value of MPIDR_EL1.Aff0[7:4], viewed from the highest Exception level of the associated PEs.
0bxxx10000	SPMDEVAFF_EL1.Aff0[7:5] is the value of MPIDR_EL1.Aff0[7:5], viewed from the highest Exception level of the associated PEs.
0bxx100000	SPMDEVAFF_EL1.Aff0[7:6] is the value of MPIDR_EL1.Aff0[7:6], viewed from the highest Exception level of the associated PEs.
0bx1000000	SPMDEVAFF_EL1.Aff0[7] is the value of MPIDR_EL1.Aff0[7], viewed from the highest Exception level of the associated PEs.

Access to this field is RO.

Otherwise:

PE affinity level 0. Indicates whether the PE affinity is at level 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Aff0	Meaning
0x00	PE affinity is above level 1 or a subset of level 1.
0x80	PE affinity is at level 1.

Access to this field is RO.

Accessing SPMDEVAFF_EL1

Reads of SPMDEVAFF_EL1 are not affected by the value of VMPIDR_EL2 at any Exception level.

If System PMU <s> has affinity only with this PE, then it is IMPLEMENTATION DEFINED whether SPMDEVAFF_EL1 reads-as-zero or reads the same value as MPIDR_EL1.

To access SPMDEVAFF_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMDEVAFF_EL1 reads-as-zero if any of the following are true:

- System PMU <s> is not implemented.
- System PMU <s> has no affinity with the PE or cluster of PEs.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMDEVAFF_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMDEVAFF_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMDEVAFF_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMDEVAFF_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMDEVAFF_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.38 SPMDEVARCH_EL1, System Performance Monitors Device Architecture Register

The SPMDEVARCH_EL1 characteristics are:

Purpose

Provides discovery information for System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMDEVARCH_EL1 are UNDEFINED.

Attributes

SPMDEVARCH_EL1 is a 64-bit register.

Field descriptions

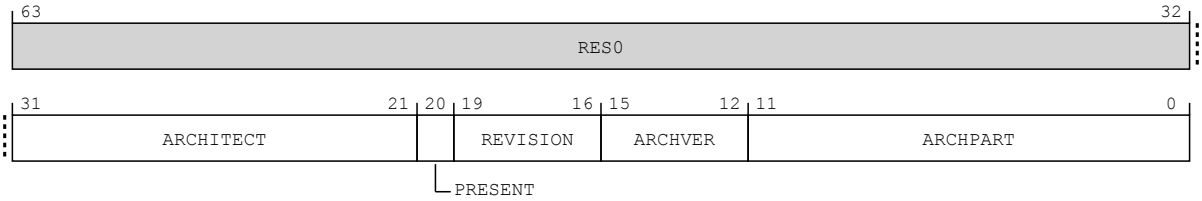

Bits [63:32]

Reserved, RES0.

ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Defines that SPMDEVARCH_EL1 register is present.

PRESENT	Meaning
0b0	Device Architecture information not present.
0b1	Device Architecture information present.

If SPMDEVARCH_EL1 is not present, the register is RES0.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

This field has an IMPLEMENTATION DEFINED value.

SPMDEVARCH_EL1.ARCHVER and SPMDEVARCH_EL1.ARCHPART are also defined as a single field, SPMDEVARCH_EL1.ARCHID, so that SPMDEVARCH_EL1.ARCHVER is SPMDEVARCH_EL1.ARCHID[15:12].

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

This field has an IMPLEMENTATION DEFINED value.

SPMDEVARCH_EL1.ARCHVER and SPMDEVARCH_EL1.ARCHPART are also defined as a single field, SPMDEVARCH_EL1.ARCHID, so that SPMDEVARCH_EL1.ARCHPART is SPMDEVARCH_EL1.ARCHID[11:0].

Access to this field is RO.

Accessing SPMDEVARCH_EL1

To access SPMDEVARCH_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMDEVARCH_EL1 reads-as-zero if any of the following are true:

- System PMU <s> is not implemented.
- System PMU <s> does not implement SPMDEVARCH_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMDEVARCH_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b101

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMID == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMDEVARCH_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMDEVARCH_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMDEVARCH_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.39 SPMEVCNTR<n>_EL0, System Performance Monitors Event Count Register, n = 0 - 63

The SPMEVCNTR<n>_EL0 characteristics are:

Purpose

Event counter <n> in System PMU <s>, where n is 0 to 63.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMEVCNTR<n>_EL0 are UNDEFINED.

Attributes

SPMEVCNTR<n>_EL0 is a 64-bit register.

Field descriptions

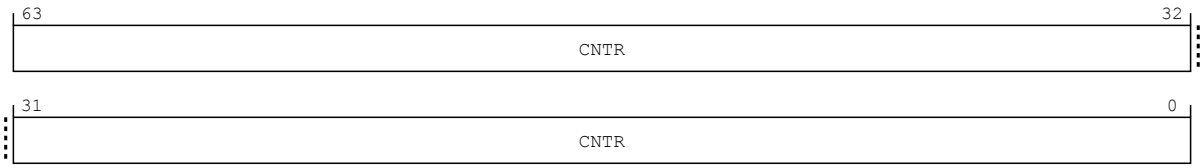

CNTR, bits [63:0]

Event counter n.

The number of implemented bits for SPMEVCNTR<n>_EL0 is IMPLEMENTATION DEFINED. Unimplemented bits are RES0.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing SPMEVCNTR<n>_EL0

To access SPMEVCNTR<n>_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s and SPMSELR_EL0.BANK to n[5:4].

SPMEVCNTR<n>_EL0 reads-as-zero and ignores writes if any of the following are true:

- Event counter <n> is not implemented by System PMU <s>.
- System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMEVCNTR<m>_EL0 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b000:m[3]	m[2:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMEVCNTRn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMEVCNTRn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
end;
```

```
MSR SPMEVCNTR<m>_EL0, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b000:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif MDSCR_EL1().EnSPM == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMEVCNTRn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
            SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMEVCNTRn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        return;
    else
        SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        return;
    else
        SPMEVCNTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.40 SPMEVFILT2R<n>_EL0, System Performance Monitors Event Filter Control Register 2, n = 0 - 63

The SPMEVFILT2R<n>_EL0 characteristics are:

Purpose

With SPMEVTYPER<n>_EL0 and SPMEVFILTR<n>_EL0, configures when event counter SPMEVCNTR<n>_EL0 in System PMU <s> increments.

The contents of this register are IMPLEMENTATION DEFINED. For more information, see SPMEVTYPER<n>_EL0.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMEVFILT2R<n>_EL0 are UNDEFINED.

Attributes

SPMEVFILT2R<n>_EL0 is a 64-bit register.

Field descriptions

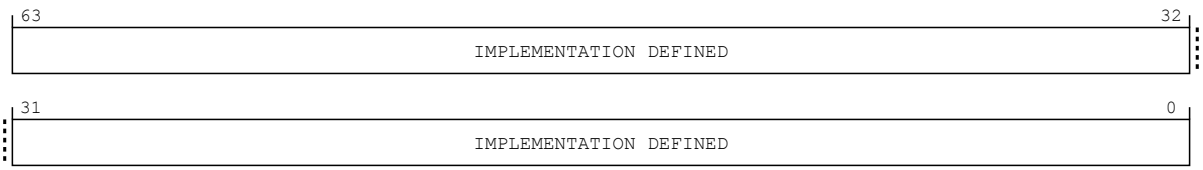

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an IMPLEMENTATION DEFINED value.

Accessing SPMEVFILT2R<n>_EL0

To access SPMEVFILT2R<n>_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s and SPMSELR_EL0.BANK to n[5:4].

SPMEVFILT2R<n>_EL0 reads-as-zero and ignores writes if any of the following are true:

- Event counter <n> is not implemented by System PMU <s>.
- System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMEVFILT2R<m>_EL0 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b011:m[3]	m[2:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
end;
```

```
MSR SPMEVFILT2R<m>_EL0, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b011:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif MDSCR_EL1().EnSPM == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
            SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        return;
    else
        SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        return;
    else
        SPMEVFILT2R_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.41 SPMEVFILTR<n>_EL0, System Performance Monitors Event Filter Control Register, n = 0 - 63

The SPMEVFILTR<n>_EL0 characteristics are:

Purpose

With SPMEVTYPER<n>_EL0 and SPMEVFILT2R<n>_EL0, configures when event counter SPMEVCNTR<n>_EL0 in System PMU <s> increments.

The contents of this register are IMPLEMENTATION DEFINED. For more information, see SPMEVTYPER<n>_EL0.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMEVFILTR<n>_EL0 are UNDEFINED.

Attributes

SPMEVFILTR<n>_EL0 is a 64-bit register.

Field descriptions

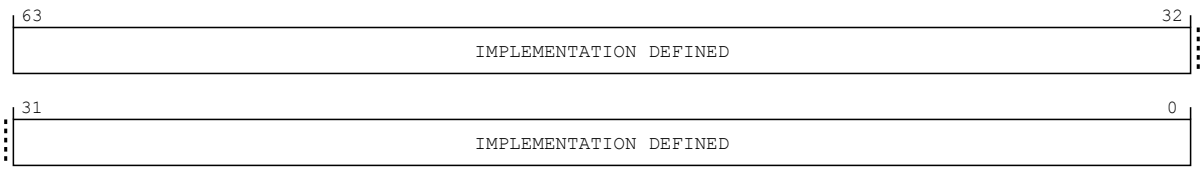

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an IMPLEMENTATION DEFINED value.

Accessing SPMEVFILTR<n>_EL0

To access SPMEVFILTR<n>_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s and SPMSELR_EL0.BANK to n[5:4].

SPMEVFILTR<n>_EL0 reads-as-zero and ignores writes if any of the following are true:

- Event counter <n> is not implemented by System PMU <s>.
- System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMEVFILTR<m>_EL0 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b010:m[3]	m[2:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
     ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
     ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
     ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
end;
```

```
MSR SPMEVFILTR<m>_EL0, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b010:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
     ↪!= '11' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif MDSCR_EL1().EnSPM == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
            SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        return;
    else
        SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        return;
    else
        SPMEVFILTR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.42 SPMEVTYPER<n>_EL0, System Performance Monitors Event Type Register, n = 0 - 63

The SPMEVTYPER<n>_EL0 characteristics are:

Purpose

With SPMEVFILTR<n>_EL0 and SPMEVFILT2R<n>_EL0, configures when event counter SPMEVCNTR<n>_EL0 in System PMU <s> increments.

The contents of this register are IMPLEMENTATION DEFINED. An Event Type Select Register typically contains:

- A field defining the event that the counter is responsive to, in the least-significant bits.
- Controls for per-counter filtering, such as by mode or state.
- Additional controls, such as for a per-counter state machine.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMEVTYPER<n>_EL0 are UNDEFINED.

Attributes

SPMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions

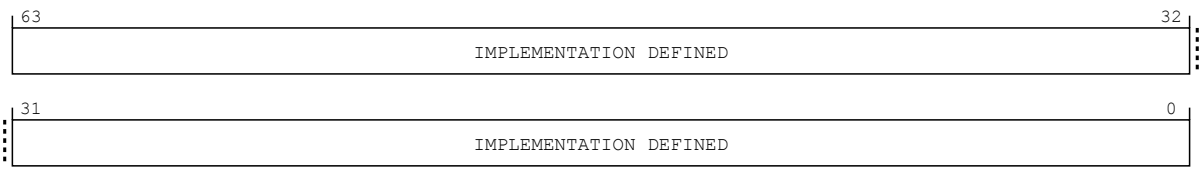

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an IMPLEMENTATION DEFINED value.

Accessing SPMEVTYPER<n>_EL0

To access SPMEVTYPER<n>_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s and SPMSELR_EL0.BANK to n[5:4].

SPMEVTYPER<n>_EL0 reads-as-zero and ignores writes if any of the following are true:

- Event counter <n> is not implemented by System PMU <s>.
- System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMEVTYPER<m>_EL0 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b001:m[3]	m[2:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m);
    end;
end;
```

```
MSR SPMEVTYPER<m>_EL0, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1110	0b001:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif MDSCR_EL1().EnSPM == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
            SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMEVTYPERn_EL0 == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
             ↪then
            return;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m)
         ↪then
        return;
    else
        SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if !IsSPMUCounterImplemented(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) then
        return;
    else
        SPMEVTYPER_EL0(UInt(SPMSELR_EL0().SYSPMUSEL), (UInt(SPMSELR_EL0().BANK) * 16) + m) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.43 SPMIIDR_EL1, System PMU Implementation Identification Register

The SPMIIDR_EL1 characteristics are:

Purpose

Provides discovery information for System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMIIDR_EL1 are UNDEFINED.

Attributes

SPMIIDR_EL1 is a 64-bit register.

Field descriptions

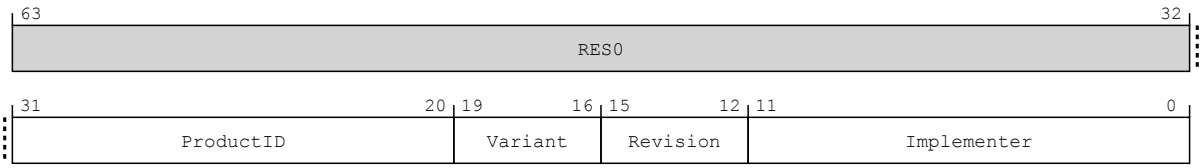

Bits [63:32]

Reserved, RES0.

ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by SPMIIDR_EL1.ProductID, or the major revision of the component.

When defining a major revision, SPMIIDR_EL1.Variant and SPMIIDR_EL1.Revision together form the revision number of the component, with SPMIIDR_EL1.Variant being the most significant part and SPMIIDR_EL1.Revision the least significant part. When a component is changed, SPMIIDR_EL1.Variant or SPMIIDR_EL1.Revision is increased to ensure that software can differentiate the different revisions of the component. If SPMIIDR_EL1.Variant is increased then SPMIIDR_EL1.Revision should be set to `0b0000`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [15:12]

Component minor revision.

When a component is changed:

- If SPMIIDR_EL1.Variant and SPMIIDR_EL1.Revision together form the revision number of the component then:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SPMIIDR_EL1.Variant or SPMIIDR_EL1.Revision is increased to ensure that software can differentiate the different revisions of the component.
- If Variant is increased then Revision should be set to `0b0000`.
- Otherwise, SPMIIDR_EL1.Revision is increased to ensure that software can differentiate the different revisions of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the System PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

Access to this field is RO.

Accessing SPMIIDR_EL1

To access SPMIIDR_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMIIDR_EL1 reads-as-zero if any of the following are true:

- System PMU <s> is not implemented.
- System PMU <s> does not implement SPMIIDR_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMIIDR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1101	0b100

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMID == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMIIDR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMIIDR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMIIDR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.44 SPMINTENCLR_EL1, System Performance Monitors Interrupt Enable Clear Register

The SPMINTENCLR_EL1 characteristics are:

Purpose

Disables the generation of interrupt requests on overflows from event counters in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMINTENCLR_EL1 are UNDEFINED.

Attributes

SPMINTENCLR_EL1 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Event counter <m> overflow interrupt request disable.

P<m>	Meaning
0b0	Event counter <m> in System PMU <s> interrupt request is disabled.
0b1	Event counter <m> in System PMU <s> interrupt request is enabled.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - event counter <m> is not implemented by System PMU <s>
  - event counter <m> does not implement an overflow flag
  - System PMU <s> does not implement an overflow interrupt request
- Otherwise, access to this field is W1C.

Accessing SPMINTENCLR_EL1

To access SPMINTENCLR_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMINTENCLR_EL1 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPMINTENCLR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1110	0b010

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMINTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMINTENCLR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMINTENCLR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMINTENCLR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SPMINTENCLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1110	0b010

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMINTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMINTENCLR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMINTENCLR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMINTENCLR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.45 SPMINTENSET_EL1, System Performance Monitors Interrupt Enable Set Register

The SPMINTENSET_EL1 characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from event counters in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMINTENSET_EL1 are UNDEFINED.

Attributes

SPMINTENSET_EL1 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Event counter <m> overflow interrupt request enable.

P<m>	Meaning
0b0	Event counter <m> in System PMU <s> interrupt request is disabled.
0b1	Event counter <m> in System PMU <s> interrupt request is enabled.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - event counter <m> is not implemented by System PMU <s>
  - event counter <m> does not implement an overflow flag
  - System PMU <s> does not implement an overflow interrupt request
- Otherwise, access to this field is W1S.

Accessing SPMINTENSET_EL1

To access SPMINTENSET_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMINTENSET_EL1 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPMINTENSET_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMINTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMINTENSET_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMINTENSET_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMINTENSET_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SPMINTENSET_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b1001	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMINTEN == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMINTENSET_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMINTENSET_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMINTENSET_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.46 SPMOVSCLR_EL0, System Performance Monitors Overflow Flag Status Clear Register

The SPMOVSCLR_EL0 characteristics are:

Purpose

Clears the state of overflow bits for event counters in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMOVSCLR_EL0 are UNDEFINED.

Attributes

SPMOVSCLR_EL0 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Event counter <m> unsigned overflow bit clear.

P<m>	Meaning
0b0	Event counter <m> in System PMU <s> has not overflowed.
0b1	Event counter <m> in System PMU <s> has overflowed.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - event counter <m> is not implemented by System PMU <s>
  - event counter <m> does not implement an overflow flag
- Otherwise, access to this field is W1C.

Accessing SPMOVSCLR_EL0

To access SPMOVSCLR_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMOVSCLR_EL0 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPMOVSCLR_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMOVS == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMOVS == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```

MSR SPMOVSCLR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMOVS == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMOVS == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMOVSCLR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.47 SPMOVSSET_EL0, System Performance Monitors Overflow Flag Status Set Register

The SPMOVSSET_EL0 characteristics are:

Purpose

Sets the state of overflow bits for event counters in System PMU <s>.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMOVSSET_EL0 are UNDEFINED.

Attributes

SPMOVSSET_EL0 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Event counter <m> unsigned overflow bit set.

P<m>	Meaning
0b0	Event counter <m> in System PMU <s> has not overflowed.
0b1	Event counter <m> in System PMU <s> has overflowed.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - event counter <m> is not implemented by System PMU <s>
  - event counter <m> does not implement an overflow flag
- Otherwise, access to this field is W1S.

Accessing SPMOVSSET_EL0

To access SPMOVSSET_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMOVSSET_EL0 reads-as-zero and ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPMOVSSET_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1110	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMOVS == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMOVS == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```

MSR SPMOVSSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1110	0b011

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
             ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMOVS == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
             ↪|| HDFGWTR2_EL2().nSPMOVS == '0') then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
             ↪!= '11' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMOVSSET_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.48 SPMROOTCR_EL3, System Performance Monitors Root and Realm Control Register

The SPMROOTCR_EL3 characteristics are:

Purpose

Controls observability of Root and Realm events by System PMU <s>.

Configuration

This register is present only when FEAT_RME is implemented, FEAT_SPMU is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMROOTCR_EL3 are UNDEFINED.

Attributes

SPMROOTCR_EL3 is a 64-bit register.

Field descriptions

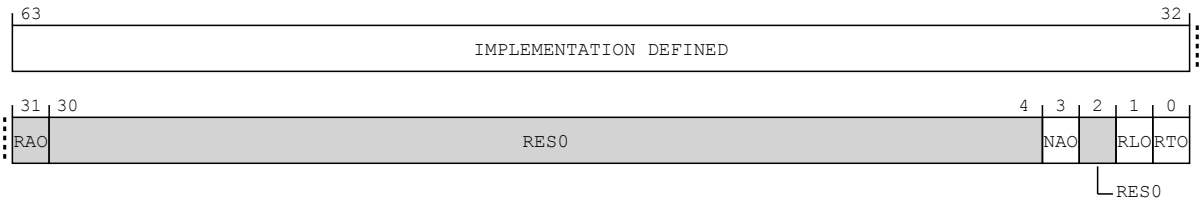

IMPLEMENTATION DEFINED, bits [63:32]

IMPLEMENTATION DEFINED observation controls. Additional IMPLEMENTATION DEFINED bits to control certain types of filter or events by System PMU <s>.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an IMPLEMENTATION DEFINED value.

Bit [31]

Reserved, RAO.

Indicates SPMROOTCR_EL3 is implemented by System PMU <s>.

Bits [30:4]

Reserved, RES0.

NAO, bit [3]

When System PMU <s> can count or monitor non-attributable events:

Non-attributable Observation. Controls whether events or monitorable characteristics not attributable with any source can be monitored by System PMU <s>.

NAO	Meaning
0b0	Events not attributable with any event source are not counted by System PMU <s>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Counting non-attributable events by System PMU <s> is not prevented by this mechanism.

When both SPMROOTCR_EL3 and SPMSCR_EL1 are implemented, non-attributable events are counted only if both SPMROOTCR_EL3.NAO is 1 and SPMSCR_EL1.{NAO, SO} is nonzero.

SPMROOTCR_EL3.NAO has the opposite reset polarity to SPMSCR_EL1.NAO.

The reset behavior of this field is:

- On a System PMU reset, this field resets to `'1'`.

Otherwise:

Reserved, RES0.

Bit [2]

Reserved, RES0.

RLO, bit [1]

Realm Observation. Controls whether events or monitorable characteristics attributable to a Realm event source can be monitored by System PMU <s>.

RLO	Meaning
0b0	Events attributable to a Realm event source are not counted by System PMU <s>.
0b1	Counting events by System PMU <s> that are attributable to a Realm event source is not prevented by this mechanism.

The reset behavior of this field is:

- On a System PMU reset, this field resets to `'0'`.

RTO, bit [0]

Root Observation. Controls whether events or monitorable characteristics attributable to a Root event source can be monitored by System PMU <s>.

RTO	Meaning
0b0	Events attributable to a Root event source are not counted by System PMU <s>.
0b1	Counting events by System PMU <s> that are attributable to a Root event source is not prevented by this mechanism.

The reset behavior of this field is:

- On a System PMU reset, this field resets to `'0'`.

Accessing SPMROOTCR_EL3

To access SPMROOTCR_EL3 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMROOTCR_EL3 reads-as-zero and ignores writes if any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- System PMU <s> is not implemented.
- System PMU <s> does not implement SPMROOTCR_EL3.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMROOTCR_EL3
```

op0	op1	CRn	CRm	op2
0b10	0b110	0b1001	0b1110	0b111

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_SPMU) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMROOTCR_EL3(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```

```
MSR SPMROOTCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b110	0b1001	0b1110	0b111

```
if !(IsFeatureImplemented(FEAT_RME) && IsFeatureImplemented(FEAT_SPMU) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().SPMROOTCR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        SPMROOTCR_EL3(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.49 SPMSCR_EL1, System Performance Monitors Secure Control Register

The SPMSCR_EL1 characteristics are:

Purpose

Controls observability of Secure events by System PMU <s>, and optionally controls Secure attributes for message signaled interrupts and Non-secure access to the performance monitor registers.

Configuration

This register is present only when Secure EL1 is implemented, FEAT_SPMU is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMSCR_EL1 are UNDEFINED.

Attributes

SPMSCR_EL1 is a 64-bit register.

Field descriptions

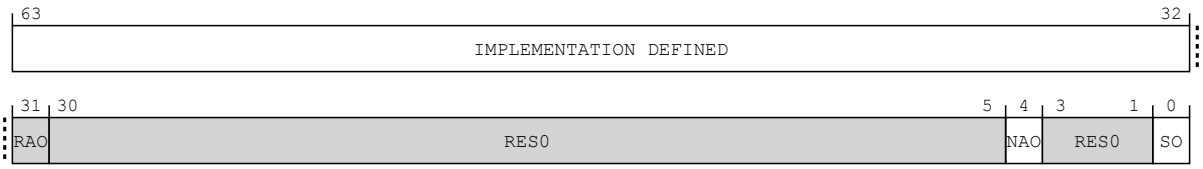

IMPLEMENTATION DEFINED, bits [63:32]

IMPLEMENTATION DEFINED observation controls. Additional IMPLEMENTATION DEFINED bits to control certain types of filter or events by System PMU <s>.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an IMPLEMENTATION DEFINED value.

Bit [31]

Reserved, RAO.

Indicates SPMSCR_EL1 is implemented by System PMU <s>.

This field reads-as-one.

Bits [30:5]

Reserved, RES0.

NAO, bit [4]

When System PMU <s> can count or monitor non-attributable events:

Non-attributable Observation. Controls whether events or monitorable characteristics not attributable with any source can be monitored by System PMU <s>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NAO	Meaning
0b0	Events not attributable with any event source are not counted by System PMU <s>, unless overridden by SPMSCR_EL1.SO.
0b1	Counting non-attributable events by System PMU <s> is not prevented by this mechanism.

When both SPMROOTCR_EL3 and SPMSCR_EL1 are implemented, non-attributable events are counted only if both SPMROOTCR_EL3.NAO is 1 and SPMSCR_EL1.{NAO, SO} is nonzero.

SPMSCR_EL1.NAO has the opposite reset polarity to SPMROOTCR_EL3.NAO.

This field is optional if Root and Realm states are not implemented. When this field is not implemented, System PMU <s> behaves as if SPMSCR_EL1.NAO is 0, and whether events or monitorable characteristics not attributable with any source can be monitored is controlled by SPMSCR_EL1.SO.

The reset behavior of this field is:

- On a System PMU reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bits [3:1]

Reserved, RES0.

SO, bit [0]

Secure Observation. Controls whether events or monitorable characteristics attributable to a Secure event source can be monitored by System PMU <s>.

SO	Meaning
0b0	Events attributable to a Secure event source are not counted by System PMU <s>.
0b1	Counting events by System PMU <s> that are attributable to a Secure event source is not prevented by this mechanism.

Also controls whether events or monitorable characteristics not attributable with any source can be monitored by System PMU <s>. See SPMSCR_EL1.NAO.

The reset behavior of this field is:

- On a System PMU reset, this field resets to `'0'`.

Accessing SPMSCR_EL1

To access SPMSCR_EL1 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMSCR_EL1 reads-as-zero and ignores writes if any of the following are true:

- System PMU <s> is not implemented.
- System PMU <s> does not implement SPMSCR_EL1.

SPMSCR_EL1 is UNDEFINED if accessed in Non-secure or Realm state.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, SPMSCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b111	0b1001	0b1110	0b111

```
if !(HaveELUsingSecurityState(EL1, TRUE) && IsFeatureImplemented(FEAT_SPMU) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif IsCurrentSecurityState(SS_NonSecure) || (IsFeatureImplemented(FEAT_RME) &&
     ↪IsCurrentSecurityState(SS_Realm)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nSPMSCR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMSCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪== '00' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMSCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMSCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL));
end;
```

```
MSR SPMSCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b111	0b1001	0b1110	0b111

```
if !(HaveELUsingSecurityState(EL1, TRUE) && IsFeatureImplemented(FEAT_SPMU) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif IsCurrentSecurityState(SS_NonSecure) || (IsFeatureImplemented(FEAT_RME) &&
     ↪IsCurrentSecurityState(SS_Realm)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMSCR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMSCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMSCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMSCR_EL1(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.50 SPMSELR_EL0, System Performance Monitors Select Register

The SPMSELR_EL0 characteristics are:

Purpose

Selects the System PMU and event counter registers to access.

Configuration

This register is present only when FEAT_SPMU is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMSELR_EL0 are UNDEFINED.

Attributes

SPMSELR_EL0 is a 64-bit register.

Field descriptions

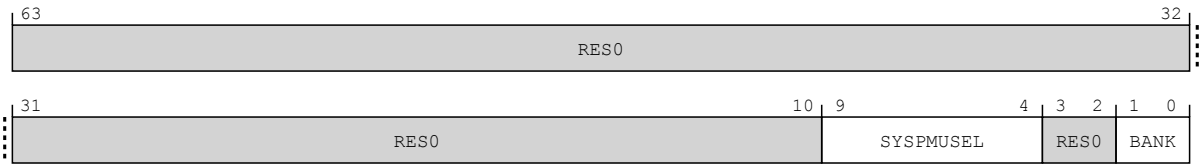

Bits [63:10]

Reserved, RES0.

SYSPMUSEL, bits [9:4]

System PMU Select. Selects a System PMU <s> to access.

Values `0x20` to `0x3F` are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:2]

Reserved, RES0.

BANK, bits [1:0]

System PMU bank access control. Selects a bank of 16 System PMU event counters and related controls to access.

BANK	Meaning
`0b00`	Select event counters 0 to 15.
`0b01`	Select event counters 16 to 31.
`0b10`	Select event counters 32 to 47.
`0b11`	Select event counters 48 to 63.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SPMSELR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPMSELR_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b101

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
     ↪SCR_EL3().FGTEn2 == '0') || HDFGRTR2_EL2().nSPMSELR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMSELR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
     ↪|| HDFGRTR2_EL2().nSPMSELR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = SPMSELR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = SPMSELR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = SPMSELR_EL0();
end;
```

```
MSR SPMSELR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b101

```
if !(IsFeatureImplemented(FEAT_SPMU) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
     ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMSELR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMSELR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
     ↪|| HDFGWTR2_EL2().nSPMSELR_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMSELR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMSELR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMSELR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.5.51 SPMZR_EL0, System Performance Monitors Zero with Mask

The SPMZR_EL0 characteristics are:

Purpose

Zero the set of System PMU event counters specified by the mask written to SPMZR_EL0.

Configuration

This register is present only when FEAT_SPMU2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to SPMZR_EL0 are UNDEFINED.

Attributes

SPMZR_EL0 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
P63	P62	P61	P60	P59	P58	P57	P56	P55	P54	P53	P52	P51	P50	P49	P48	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P31	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

P<m> , bits [m], for m = 63 to 0

Zero event counter <m>.

P<m>	Meaning
0b0	Write is ignored.
0b1	Set event counter<m> in System PMU <s> to zero.

The reset behavior of this field is:

- On a System PMU reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When event counter <m> is not implemented by System PMU <s>, access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing SPMZR_EL0

To access SPMZR_EL0 for System PMU <s>, set SPMSELR_EL0.SYSPMUSEL to s.

SPMZR_EL0 ignores writes if System PMU <s> is not implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MSR SPMZR_EL0, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b011	0b1001	0b1100	0b100

```
if !(IsFeatureImplemented(FEAT_SPMU2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif MDSCR_EL1().EnSPM == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif !ELIsInHost(EL0) && SPMACCESSR_EL1()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) &&
         ↪SCR_EL3().FGTEn2 == '0') || HDFGWTR2_EL2().nSPMEVCNTRn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMZR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nSPMEVCNTRn_EL0 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().EnSPM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && SPMACCESSR_EL2()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMZR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPM2 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2]
         ↪!= '11' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().EnPM2 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SPMACCESSR_EL3()[UInt(SPMSELR_EL0().SYSPMUSEL) * 2+:2] != '11' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        SPMZR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    SPMZR_EL0(UInt(SPMSELR_EL0().SYSPMUSEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6 Activity Monitors registers

This section lists the Activity Monitors registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.1 AMCFGR_EL0, Activity Monitors Configuration Register

The AMCFGR_EL0 characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configuration

AArch64 System register AMCFGR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCFGR[31:0].

When FEAT_AMU_EXT32 is implemented, AArch64 System register AMCFGR_EL0 bits [31:0] are architecturally mapped to External register AMCFGR[31:0].

When FEAT_AMU_EXT64 is implemented, AArch64 System register AMCFGR_EL0 bits [63:0] are architecturally mapped to External register AMCFGR[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCFGR_EL0 are UNDEFINED.

Attributes

AMCFGR_EL0 is a 64-bit register.

Field descriptions

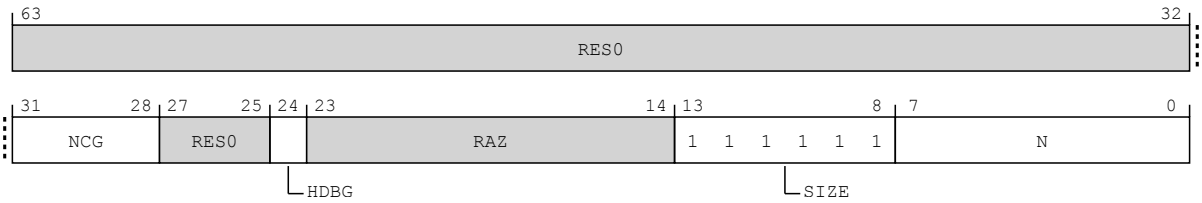

Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

This feature must be supported, and so this bit is 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HDBG	Meaning
0b0	AMCR_EL0.HDBG is RES0.
0b1	AMCR_EL0.HDBG is read/write.

Access to this field is RO.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of the activity monitor event counters, minus one.

The counters are 64-bit, so the value of this field is 0b111111.

This field is used by software to determine the spacing of the counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as 0b111111

Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters implemented in all groups. One less than the sum of AMCGCR_EL0.CG1NC and AMCGCR_EL0.CG0NC.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMCFGR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCFGR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCFGR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCFGR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCFGR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCFGR_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.2 AMCG1IDR_EL0, Activity Monitors Counter Group 1 Identification Register

The AMCG1IDR_EL0 characteristics are:

Purpose

Defines which auxiliary counters are implemented, and which of them have a corresponding virtual offset register, AMEVCNTVOFF1<n>_EL2 implemented.

Configuration

This register is present only when FEAT_AMUv1p1 is implemented. Otherwise, direct accesses to AMCG1IDR_EL0 are UNDEFINED.

Attributes

AMCG1IDR_EL0 is a 64-bit register.

Field descriptions

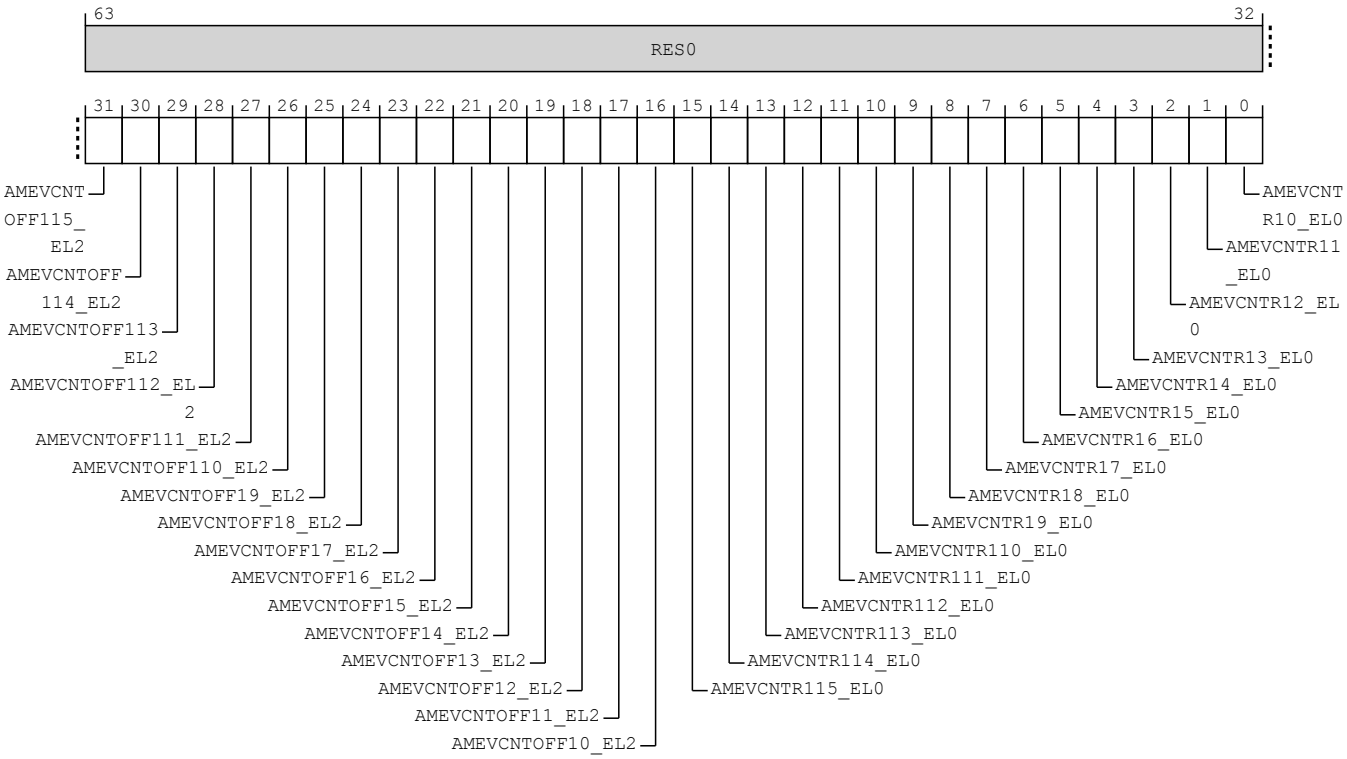

Bits [63:32]

Reserved, RES0.

AMEVCNTOFF1<n>_EL2 , bits [n+16], for n = 15 to 0

Indicates which implemented auxiliary counters have a corresponding virtual offset register, AMEVCNTVOFF1<n>_EL2 implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AMEVCNTOFF1<n>_EL2	Meaning
0b0	AMEVCNTR1<n>_EL0 does not have an offset, or is not implemented.
0b1	The offset AMEVCNTVOFF1<n>_EL2 is implemented for AMEVCNTR1<n>_EL0.

AMEVCNTR1<n>_EL0 , bits [n], for n = 15 to 0

Indicates which auxiliary counters AMEVCNTR1<n>_EL0 are implemented.

AMEVCNTR1<n>_EL0	Meaning
0b0	AMEVCNTR1<n>_EL0 is not implemented.
0b1	AMEVCNTR1<n>_EL0 is implemented.

Accessing AMCG1IDR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCG1IDR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_AMUv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCG1IDR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCG1IDR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCG1IDR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCG1IDR_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.3 AMCGCR_EL0, Activity Monitors Counter Group Configuration Register

The AMCGCR_EL0 characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configuration

AArch64 System register AMCGCR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCGCR[31:0].

When FEAT_AMU_EXT32 is implemented, AArch64 System register AMCGCR_EL0 bits [31:0] are architecturally mapped to External register AMCGCR[31:0].

When FEAT_AMU_EXT64 is implemented, AArch64 System register AMCGCR_EL0 bits [63:0] are architecturally mapped to External register AMCGCR[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCGCR_EL0 are UNDEFINED.

Attributes

AMCGCR_EL0 is a 64-bit register.

Field descriptions

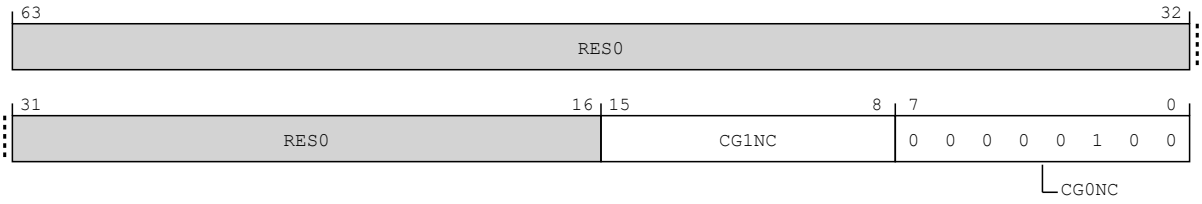

Bits [63:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. Highest index of the implemented counters in the auxiliary counter group, plus one. Zero indicates that there are no counters in the auxiliary counter group.

If FEAT_AMUv1p1 is implemented, then AMCG1IDR_EL0 further defines which auxiliary counters are implemented, and the counters in the auxiliary counter group might be indexed non-contiguously.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CG1NC	Meaning
0x00..0x10	The number of counters.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as `0x04`

Access to this field is RO.

Accessing AMCGCR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCGCR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCGCR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCGCR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = AMCGCR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCGCR_EL0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.4 AMCNTENCLR0_EL0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0_EL0 characteristics are:

Purpose

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>_EL0.

Configuration

AArch64 System register AMCNTENCLR0_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

AArch64 System register AMCNTENCLR0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR0[31:0].

AArch64 System register AMCNTENCLR0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR[31:0].

AArch64 System register AMCNTENCLR0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET0[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR0_EL0 are UNDEFINED.

Attributes

AMCNTENCLR0_EL0 is a 64-bit register.

Field descriptions

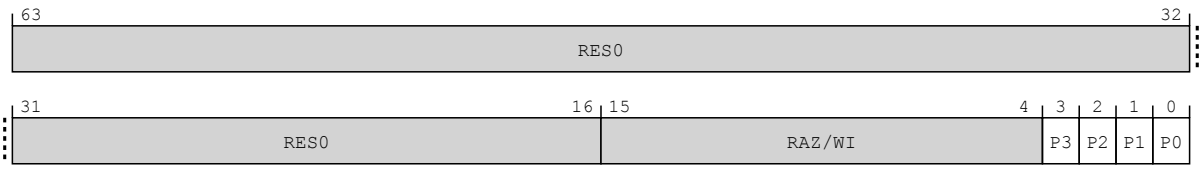

Bits [63:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>_EL0.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<n>	Meaning
0b0	When read, means that AMEVCNTR0<n>_EL0 is disabled. When written, has no effect.
0b1	When read, means that AMEVCNTR0<n>_EL0 is enabled. When written, disables AMEVCNTR0<n>_EL0.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR0_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCNTENCLR0_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b100

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2().AMCNTEN0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENCLR0_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HAFGRTR_EL2().AMCNTEN0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENCLR0_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENCLR0_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCNTENCLR0_EL0();
end;
```

MSR AMCNTENCLR0_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b100

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENCLR0_EL0() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.5 AMCNTENCLR1_EL0, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1_EL0 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>_EL0.

Configuration

AArch64 System register AMCNTENCLR1_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

AArch64 System register AMCNTENCLR1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

AArch64 System register AMCNTENCLR1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

AArch64 System register AMCNTENCLR1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR[63:32].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1_EL0 are UNDEFINED.

Attributes

AMCNTENCLR1_EL0 is a 64-bit register.

Field descriptions

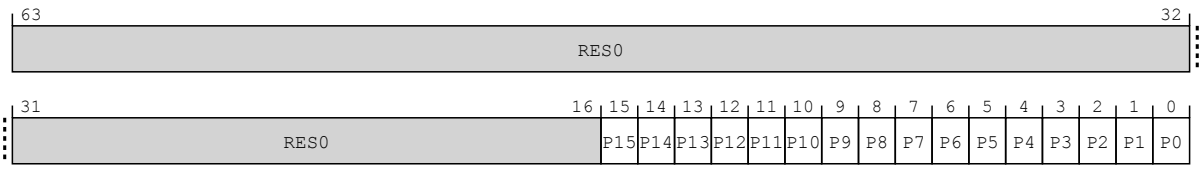

Bits [63:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>_EL0.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n>_EL0 is disabled. When written, has no effect.
0b1	When read, means that AMEVCNTR1<n>_EL0 is enabled. When written, disables AMEVCNTR1<n>_EL0.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression 0x00000000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When n >= UInt(AMCGCR_EL0.CG1NC), access to this field is RAZ/WI.
- When !IsG1ActivityMonitorImplemented(n), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR1_EL0

If there are no auxiliary monitor event counters implemented, reads and writes of AMCNTENCLR1_EL0 are UNDEFINED.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR_EL0.NCG == `0b0000`.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCNTENCLR1_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2().AMCNTEN1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENCLR1_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HAFGRTR_EL2().AMCNTEN1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = AMCNTENCLR1_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENCLR1_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCNTENCLR1_EL0();
end;
```

```
MSR AMCNTENCLR1_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENCLR1_EL0() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.6 AMCNTENSET0_EL0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0_EL0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>_EL0.

Configuration

AArch64 System register AMCNTENSET0_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET0[31:0].

AArch64 System register AMCNTENSET0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET0[31:0].

AArch64 System register AMCNTENSET0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET[31:0].

AArch64 System register AMCNTENSET0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR0[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET0_EL0 are UNDEFINED.

Attributes

AMCNTENSET0_EL0 is a 64-bit register.

Field descriptions

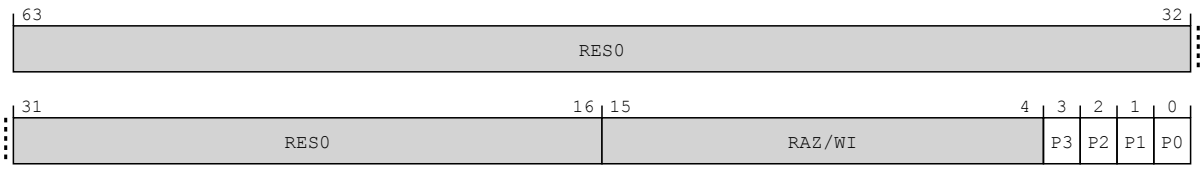

Bits [63:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter enable bit for AMEVCNTR0<n>_EL0.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<n>	Meaning
0b0	When read, means that AMEVCNTR0<n>_EL0 is disabled. When written, has no effect.
0b1	When read, means that AMEVCNTR0<n>_EL0 is enabled. When written, enables AMEVCNTR0<n>_EL0.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET0_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCNTENSET0_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2().AMCNTEN0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENSET0_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HAFGRTR_EL2().AMCNTEN0 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENSET0_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENSET0_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCNTENSET0_EL0();
end;
```

MSR AMCNTENSET0_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENSET0_EL0() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.7 AMCNTENSET1_EL0, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1_EL0 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>_EL0.

Configuration

AArch64 System register AMCNTENSET1_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

AArch64 System register AMCNTENSET1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

AArch64 System register AMCNTENSET1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

AArch64 System register AMCNTENSET1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET[63:32].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1_EL0 are UNDEFINED.

Attributes

AMCNTENSET1_EL0 is a 64-bit register.

Field descriptions

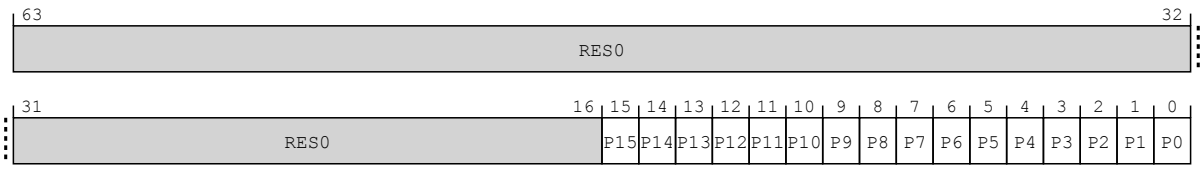

Bits [63:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<n>_EL0.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n>_EL0 is disabled. When written, has no effect.
0b1	When read, means that AMEVCNTR1<n>_EL0 is enabled. When written, enables AMEVCNTR1<n>_EL0.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression 0x00000000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When n >= UInt(AMCGCR_EL0.CG1NC), access to this field is RAZ/WI.
- When !IsG1ActivityMonitorImplemented(n), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET1_EL0

If there are no auxiliary monitor event counters implemented, reads and writes of AMCNTENSET1_EL0 are UNDEFINED.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR_EL0.NCG == `0b0000`.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCNTENSET1_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2().AMCNTEN1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENSET1_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HAFGRTR_EL2().AMCNTEN1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = AMCNTENSET1_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCNTENSET1_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCNTENSET1_EL0();
end;
```

```
MSR AMCNTENSET1_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENSET1_EL0() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.8 AMCR_EL0, Activity Monitors Control Register

The AMCR_EL0 characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configuration

AArch64 System register AMCR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCR[31:0].

When FEAT_AMU_EXT32 is implemented, AArch64 System register AMCR_EL0 bits [31:0] are architecturally mapped to External register AMCR[31:0].

When FEAT_AMU_EXT64 is implemented, AArch64 System register AMCR_EL0 bits [63:0] are architecturally mapped to External register AMCR[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCR_EL0 are UNDEFINED.

Attributes

AMCR_EL0 is a 64-bit register.

Field descriptions

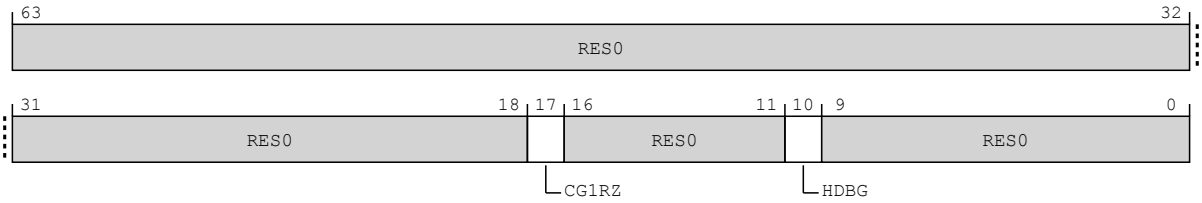

Bits [63:18]

Reserved, RES0.

CG1RZ, bit [17]

When FEAT_AMUv1p1 is implemented:

Counter Group 1 Read Zero.

CG1RZ	Meaning
0b0	System register reads of AMEVCNTR1<n>_EL0 return the event count at all implemented and enabled Exception levels.
0b1	If the current Exception level is the highest implemented Exception level, system register reads of AMEVCNTR1<n>_EL0 return the event count. Otherwise, reads of AMEVCNTR1<n>_EL0 return a zero value.

Note

Reads from the memory-mapped view are unaffected by this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

HDBG	Meaning
0b0	Activity monitors do not halt counting when the PE is halted in Debug state.
0b1	Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing AMCR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMCR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = AMCR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMCR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMCR_EL0();
end;
```

MSR AMCR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMCR_EL0() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.9 AMEVCNTR0<n>_EL0, Activity Monitors Event Counter Registers 0, n = 0 - 3

The AMEVCNTR0<n>_EL0 characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configuration

AArch64 System register AMEVCNTR0<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMEVCNTR0<n>[31:0].

AArch64 System register AMEVCNTR0<n>_EL0 bits [63:0] are architecturally mapped to External register AMEVCNTR0<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR0<n>_EL0 are UNDEFINED.

Attributes

AMEVCNTR0<n>_EL0 is a 64-bit register.

Field descriptions

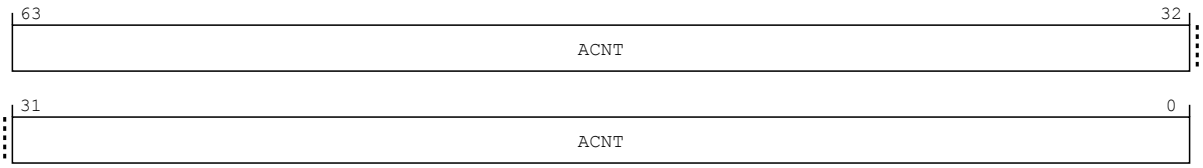

ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 3.

If all of the following are true, reads of the AMEVCNTR0<n>_EL0 registers from EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF0<n>_EL2<63:0>), where PCount is the physical count returned when AMEVCNTR0<n>_EL0 is read from EL2 or EL3:

- FEAT_AMUv1p1 is implemented.
- HCR_EL2.AMVOFFEN and SCR_EL3.AMVOFFEN are 1.
- EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x0000000000000000`.

Accessing AMEVCNTR0<n>_EL0

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n>_EL0 are UNDEFINED.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, AMEVCNTR0<m>_EL0 ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b010:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2()[m + 1] == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVCNTR0_EL0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HAFGRTR_EL2()[m + 1] == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVCNTR0_EL0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = AMEVCNTR0_EL0(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMEVCNTR0_EL0(m);
end;
```

```
MSR AMEVCNTR0<m>_EL0, <Xt> ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b010:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR0_EL0(m) = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.10 AMEVCNTR1<n>_EL0, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n>_EL0 characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configuration

AArch64 System register AMEVCNTR1<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMEVCNTR1<n>[31:0].

AArch64 System register AMEVCNTR1<n>_EL0 bits [63:0] are architecturally mapped to External register AMEVCNTR1<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<n>_EL0 are UNDEFINED.

Attributes

AMEVCNTR1<n>_EL0 is a 64-bit register.

Field descriptions

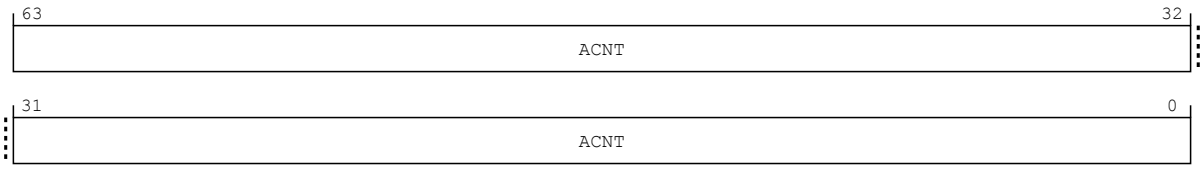

ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If all of the following are true, reads of the AMEVCNTR1<n>_EL0 registers from EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF1<n>_EL2<63:0>), where PCount is the physical count returned when AMEVCNTR1<n>_EL0 is read from EL2 or EL3:

- FEAT_AMUv1p1 is implemented.
- HCR_EL2.AMVOFFEN and SCR_EL3.AMVOFFEN are 1.
- AMCR_EL0.CG1RZ is 0.
- EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x0000000000000000`.

Accessing AMEVCNTR1<n>_EL0

If <n> is greater than or equal to AMCGCR_EL0 or FEAT_AMUv1p1 is implemented and auxiliary counter <n> is not implemented, then reads and writes of AMEVCNTR1<n>_EL0 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, AMEVCNTR1<m>_EL0 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b110:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
     ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2()[(2 * m) + 18] == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif AMCR_EL0().CG1RZ == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = AMEVCNTR1_EL0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HAFGRTR_EL2()[(2 * m) + 18] == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsHighestEL(PSTATE.EL) && AMCR_EL0().CG1RZ == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = AMEVCNTR1_EL0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif !IsHighestEL(PSTATE.EL) && AMCR_EL0().CG1RZ == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = AMEVCNTR1_EL0(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMEVCNTR1_EL0(m);
end;
```

```
MSR AMEVCNTR1<m>_EL0, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b110:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR1_EL0(m) = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.11 AMEVCNTVOFF0<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 0, n = 0 - 15

The AMEVCNTVOFF0<n>_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset for architected activity monitor events.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_AMUv1p1 is implemented. Otherwise, direct accesses to AMEVCNTVOFF0<n>_EL2 are UNDEFINED.

Attributes

AMEVCNTVOFF0<n>_EL2 is a 64-bit register.

Field descriptions

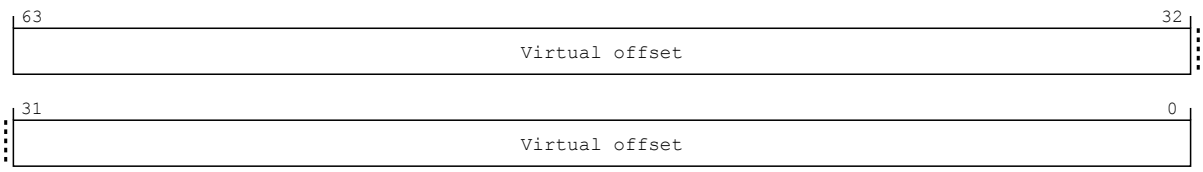

VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMEVCNTVOFF0<n>_EL2

If <n> is not 0, 2 or 3, reads and writes of AMEVCNTVOFF0<n>_EL2 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMEVCNTVOFF0<m>_EL2 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b100:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1p1) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif !(m IN {0, 2, 3}) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0xA00 + (8 * m));
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AMVOFFEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AMVOFFEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVCNTVOFF0_EL2(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMEVCNTVOFF0_EL2(m);
end;
```

```
MSR AMEVCNTVOFF0<m>_EL2, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b100:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1p1) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif !(m IN {0, 2, 3}) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0xA00 + (8 * m)) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AMVOFFEN == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && SCR_EL3().AMVOFFEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        AMEVCNTVOFF0_EL2(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMEVCNTVOFF0_EL2(m) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.12 AMEVCNTVOFF1<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 1, n = 0 - 15

The AMEVCNTVOFF1<n>_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset for auxiliary activity monitor events.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_AMUv1p1 is implemented. Otherwise, direct accesses to AMEVCNTVOFF1<n>_EL2 are UNDEFINED.

Attributes

AMEVCNTVOFF1<n>_EL2 is a 64-bit register.

Field descriptions

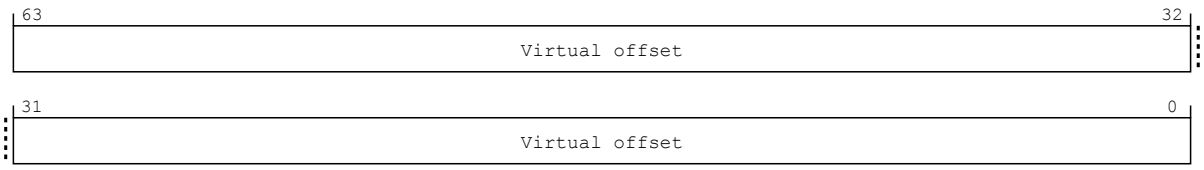

VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMEVCNTVOFF1<n>_EL2

If auxiliary counter <n> is not implemented, then reads and writes of AMEVCNTVOFF1<n>_EL2 are UNDEFINED.

Note

AMCG1IDR_EL0 identifies which auxiliary activity monitor event counters have a corresponding virtual offset implemented.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMEVCNTVOFF1<m>_EL2 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b101:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1p1) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif !IsG1ActivityMonitorOffsetImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0xA80 + (8 * m));
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AMVOFFEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AMVOFFEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVCNTVOFF1_EL2(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMEVCNTVOFF1_EL2(m);
end;
```

```
MSR AMEVCNTVOFF1<m>_EL2, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b101:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1p1) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorOffsetImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0xA80 + (8 * m)) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().AMVOFFEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().AMVOFFEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        AMEVCNTVOFF1_EL2(m) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMEVCNTVOFF1_EL2(m) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.13 AMEVTYPER0<n>_EL0, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n>_EL0 characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n>_EL0 counts.

Configuration

AArch64 System register AMEVTYPER0<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER0<n>[31:0].

AArch64 System register AMEVTYPER0<n>_EL0 bits [31:0] are architecturally mapped to External register AMEVTYPER0<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n>_EL0 are UNDEFINED.

Attributes

AMEVTYPER0<n>_EL0 is a 64-bit register.

Field descriptions

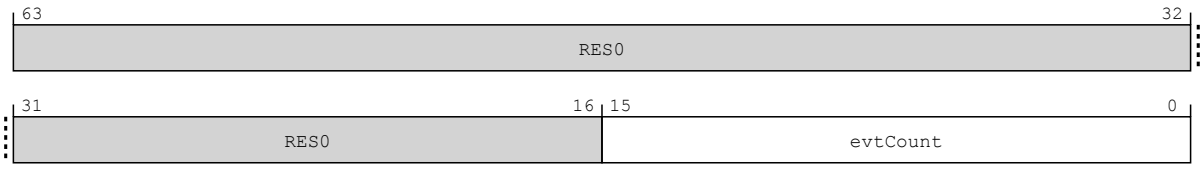

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>_EL0. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

evtCount	Meaning	Applies when
0x0011	Processor frequency cycles.	n == 0
0x4004	Constant frequency cycles.	n == 1
0x0008	Instructions retired.	n == 2
0x4005	Memory stall cycles.	n == 3

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMEVTYPER0<n>_EL0

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n>_EL0 are UNDEFINED.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMEVTYPER0<m>_EL0 ; Where m = 0-3
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b011:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVTYPER0_EL0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVTYPER0_EL0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVTYPER0_EL0(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMEVTYPER0_EL0(m);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.14 AMEVTYPER1<n>_EL0, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n>_EL0 characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n>_EL0 counts.

Configuration

AArch64 System register AMEVTYPER1<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER1<n>[31:0].

AArch64 System register AMEVTYPER1<n>_EL0 bits [31:0] are architecturally mapped to External register AMEVTYPER1<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER1<n>_EL0 are UNDEFINED.

Attributes

AMEVTYPER1<n>_EL0 is a 64-bit register.

Field descriptions

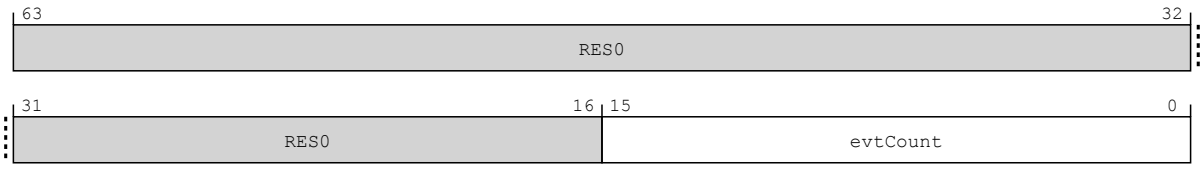

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>_EL0.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1<n>_EL0, then:

- It is UNPREDICTABLE which event will be counted.
- The value read back is UNKNOWN.

The event counted by AMEVCNTR1<n>_EL0 might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED.

If the corresponding counter AMEVCNTR1<n>_EL0 is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMEVTYPER1<n>_EL0

If <n> is greater than or equal to AMCGCR_EL0.CG1NC or FEAT_AMUv1p1 is implemented and auxiliary counter <n> is not implemented, then reads and writes of AMEVTYPER1<n>_EL0 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMEVTYPER1<m>_EL0 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b111:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HAFGRTR_EL2()[(2 * m) + 19] == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVTYPER1_EL0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HAFGRTR_EL2()[(2 * m) + 19] == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVTYPER1_EL0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMEVTYPER1_EL0(m);
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMEVTYPER1_EL0(m);
end;
```

```
MSR AMEVTYPER1<m>_EL0, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b111:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: op2[2:0]);

if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) && !ImpDefBool("AMEVCNTR1_EL0()[m] is fixed") then
    AMEVTYPER1_EL0(m) = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.6.15 AMUSERENR_EL0, Activity Monitors User Enable Register

The AMUSERENR_EL0 characteristics are:

Purpose

Global user enable register for the activity monitors. Enables or disables EL0 access to the activity monitors. AMUSERENR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configuration

AArch64 System register AMUSERENR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMUSERENR[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMUSERENR_EL0 are UNDEFINED.

Attributes

AMUSERENR_EL0 is a 64-bit register.

Field descriptions

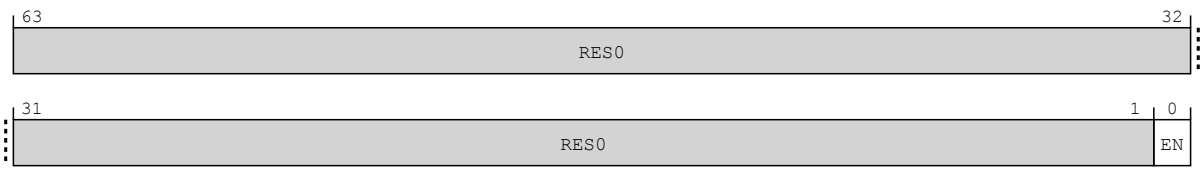

Bits [63:1]

Reserved, RES0.

EN, bit [0]

Traps EL0 accesses to the activity monitors registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped, reported using EC syndrome value `0x18`:
  - AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`:
  - AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVCNTR0<n>, AMEVCNTR1<n>, AMEVTYPER0<n>, and AMEVTYPER1<n>.

EN	Meaning
`0b0`	EL0 accesses to the activity monitors registers are trapped.
`0b1`	This control does not cause any instructions to be trapped. Software can access all activity monitor registers at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- AMUSERENR_EL0 can always be read at EL0 and is not governed by this bit.

Note

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMUSERENR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, AMUSERENR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMUSERENR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = AMUSERENR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = AMUSERENR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = AMUSERENR_EL0();
end;
```

MSR AMUSERENR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_AMUv1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && CPTR_EL2().TAM == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        AMUSERENR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && CPTR_EL3().TAM == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        AMUSERENR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    AMUSERENR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7 Statistical Profiling Extension registers

This section lists the Statistical Profiling Extension registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.1 PMBIDR_EL1, Profiling Buffer ID Register

The PMBIDR_EL1 characteristics are:

Purpose

Provides information to software as to whether the buffer can be programmed at the current Exception level.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBIDR_EL1 are UNDEFINED.

Attributes

PMBIDR_EL1 is a 64-bit register.

Field descriptions

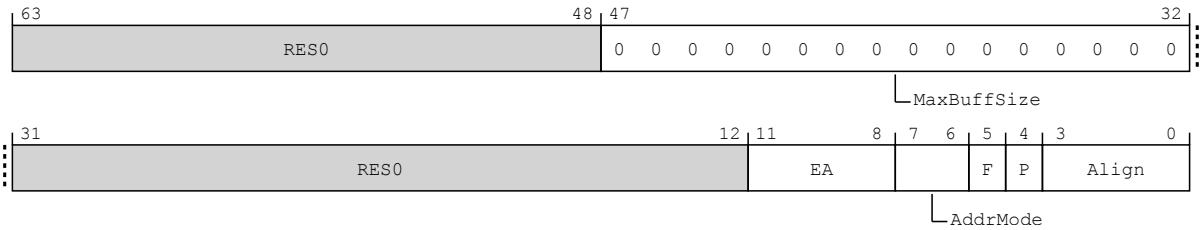

Bits [63:48]

Reserved, RES0.

MaxBuffSize, bits [47:32]

Maximum supported Profiling Buffer size. Reserved for software use.

Reads as `0x0000`

The only permitted value is `0x0000`, indicating there is no limit to the maximum buffer size.

Note

Permitted values relate to the values an implementation is permitted to set this field to. A hypervisor might trap accesses to this register and use other values to describe limitations of its virtualization support to a guest operating system, as follows:

- MaxBuffSize bits[8:0] encodes a mantissa value, `M`.
- MaxBuffSize bits[13:9] encodes an exponent value, `E`.
- MaxBuffSize bits[15:14] are reserved.

The maximum buffer size, in bytes, is expressed using the following function:

```
if IsZero(E), then UInt(M:Zeros(12)) else UInt('1':M:Zeros(UInt(E)+11))
```

For example:

- A value of `0x0001` means a maximum buffer size of 4KB.
- A value of `0x3FFF` means a maximum buffer size of 4092TB.

Access to this field is RO.

Bits [31:12]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EA, bits [11:8]

External Abort handling. Describes how the PE manages External aborts on writes made by the Statistical Profiling Unit to the Profiling Buffer.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EA	Meaning
0b0000	Not described.
0b0001	The PE ignores External aborts on writes made by the Statistical Profiling Unit.
0b0010	An External abort on a write made by the Statistical Profiling Unit generates an asynchronous SError exception at the PE.

All other values are reserved.

From Armv8.8, the value 0b0000 is not permitted.

The behavior described by this field does not apply for External aborts on a translation table walk, translation table update, or GPT walk made by the Statistical Profiling Unit that are reported as MMU faults using PMBSR_ELx. For more information, see 'External aborts'.

Access to this field is RO.

AddrMode, bits [7:6]

When FEAT_SPE_nVM is implemented:

Address Modes. Describes the addressing modes available for the Profiling Buffer.

AddrMode	Meaning
0b00	Only virtual address mode is supported.
0b01	Virtual and physical address modes are supported.
0b11	Reserved for software use under virtualization, to show that only physical address mode is supported.

Other values are reserved.

If the Effective value of PMSCR_EL2.EnVM is 1 and the value returned for PMBIDR_EL1.P is 0, then this field reads as 0b01. Otherwise, this field reads as 0b00.

Note

A hypervisor might trap accesses to this register to describe limitations of its virtualization support to a guest operating system.

Otherwise:

Reserved, RES0.

F, bit [5]

Flag updates. Describes how address translations performed by the Statistical Profiling Unit manage the Access flag and dirty state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F	Meaning
0b0	Hardware management of the Access flag and dirty state for accesses made by the Statistical Profiling Unit is always disabled for all translation stages.
0b1	Hardware management of the Access flag and dirty state for accesses made by the Statistical Profiling Unit is controlled in the same way as explicit memory accesses in the Profiling Buffer owning translation regime.

Note

If hardware management of the Access flag is disabled for a stage of translation, an access to a Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.
If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor and might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

From Armv8.8, the value 0 is not permitted.

Access to this field is RO.

P, bit [4]

Programming not allowed. When read at EL3, this field reads as zero. Otherwise, indicates that the Profiling Buffer owning Exception level is a higher Exception level or the Profiling Buffer owning Security state is not the current Security state.

P	Meaning
0b0	Programming is allowed.
0b1	Programming not allowed.

The value read from this field depends on the current Exception level and the Effective values of MDCR_EL3.NSPB, MDCR_EL3.NSPBE, and MDCR_EL2.E2PB:

- If EL3 is implemented, MDCR_EL3.NSPB is 0b0x, and either FEAT_RME is not implemented, or Secure state is implemented and MDCR_EL3.NSPBE is 0, then this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
  - If Secure EL2 is implemented and enabled, and MDCR_EL2.E2PB is 0b00, Secure EL1.
- If EL3 is implemented, MDCR_EL3.NSPB is 0b1x and either FEAT_RME is not implemented or MDCR_EL3.NSPBE is 0, then this field reads as one from:
  - If Secure state is implemented, Secure EL1.
  - If Secure EL2 is implemented, Secure EL2.
  - If EL2 is implemented and MDCR_EL2.E2PB is 0b00, Non-secure EL1.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
- If FEAT_RME is implemented, and MDCR_EL3.{NSPB, NSPBE} is {0b1x, 1}, then this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - If Secure state is implemented, Secure EL1 and Secure EL2.
  - If MDCR_EL2.E2PB is 0b00, Realm EL1.
- If EL3 is not implemented, EL2 is implemented, and MDCR_EL2.E2PB is 0b00, then this field reads as one from EL1.

Otherwise, this field reads as zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Align, bits [3:0]

Defines the minimum alignment constraint for writes to PMBPTR_EL1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Align	Meaning
0b0000	Byte.
0b0001	Halfword.
0b0010	Word.
0b0011	Doubleword.
0b0100	16 bytes.
0b0101	32 bytes.
0b0110	64 bytes.
0b0111	128 bytes.
0b1000	256 bytes.
0b1001	512 bytes.
0b1010	1KB.
0b1011	2KB.

All other values are reserved.

For more information, see Restrictions on the current write pointer.

If this field is nonzero, then every record is a multiple of this size.

Access to this field is RO.

Accessing PMBIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b111

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        ↪HDFGRTR_EL2().PMBIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = PMBIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = PMBIDR_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.2 PMBLIMITR_EL1, Profiling Buffer Limit Address Register

The PMBLIMITR_EL1 characteristics are:

Purpose

Defines the upper limit for the profiling buffer, and enables the profiling buffer

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBLIMITR_EL1 are UNDEFINED.

Attributes

PMBLIMITR_EL1 is a 64-bit register.

Field descriptions

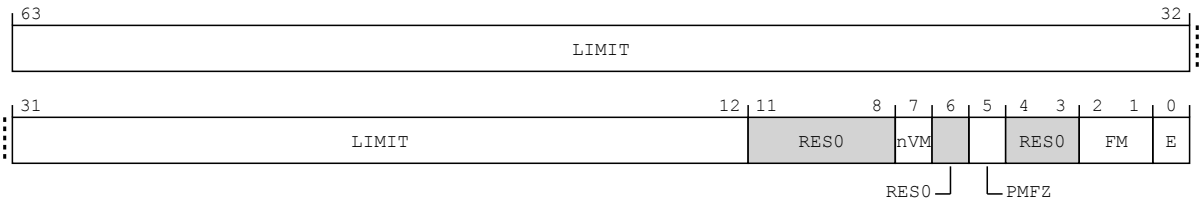

LIMIT, bits [63:12]

Limit address. PMBLIMITR_EL1.LIMIT:Zeros(12) is the address of the first byte in memory after the last byte in the profiling buffer. If the smallest implemented translation granule is not 4KB, then bits[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value, Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:8]

Reserved, RES0.

nVM, bit [7]

When FEAT_SPE_nVM is implemented:

Address mode.

nVM	Meaning
0b0	The Profiling Buffer pointers are virtual addresses.
0b1	The Profiling Buffer pointers are: • Physical address in the owning security state if the owning translation regime has no stage 2 translation. • Intermediate physical addresses in the owning security state if the owning translation regime has stage 2 translations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the Effective value of PMSCR_EL2.EnVM is 0, then the PE ignores the value of this field, and the Profiling Buffer pointers are always virtual addresses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RW.

Otherwise:

Reserved, RES0.

Bit [6]

Reserved, RES0.

PMFZ, bit [5]

When FEAT_SPEv1p2 is implemented:

Freeze PMU on SPE event. Stop PMU event counters when PMBSR_EL1.S == 1.

PMFZ	Meaning
0b0	Do not freeze PMU event counters on Statistical Profiling Buffer Management event.
0b1	Freeze PMU event counters on Statistical Profiling Buffer Management event.

The PMU event counters affected by this control is controlled by PMCR_EL0.FZS and, if EL2 is implemented, MDCR_EL2.HPMFZS. See the descriptions of these control bits for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.

FM, bits [2:1]

Fill mode.

FM	Meaning	Applies when
0b00	Fill mode. Stop collection and raise maintenance interrupt on buffer fill.	
0b10	Discard mode. All output is discarded.	FEAT_SPEv1p2 is implemented

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E, bit [0]

Profiling Buffer enable

E	Meaning
0b0	All output is discarded.
0b1	Profiling buffer enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PMBLIMITR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBLIMITR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMBLIMITR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x800);
    else
        X{64}(t) = PMBLIMITR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMBLIMITR_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBLIMITR_EL1();
end;
```

```
MSR PMBLIMITR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMBLIMITR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x800) = X{64}(t);
    else
        PMBLIMITR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMBLIMITR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMBLIMITR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.3 PMBMAR_EL1, Profiling Buffer Memory Attribute Register

The PMBMAR_EL1 characteristics are:

Purpose

Controls Statistical Profiling Unit accesses to memory.

Configures some of the memory attributes of writes performed by the Statistical Profiling Unit.

Configuration

This register is present only when FEAT_SPE_nVM is implemented. Otherwise, direct accesses to PMBMAR_EL1 are UNDEFINED.

Attributes

PMBMAR_EL1 is a 64-bit register.

Field descriptions

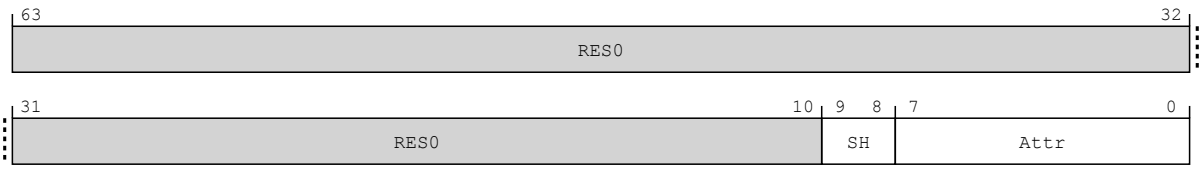

Bits [63:10]

Reserved, RES0.

SH, bits [9:8]

Profiling Buffer stage 1 shareability domain. Defines the shareability domain for Normal memory used by the Profiling Buffer.

SH	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

All other values are reserved.

If the Profiling Buffer pointers specify virtual addresses, the memory attributes are defined by the translation tables and this field is ignored.

This field is ignored when PMBMAR_EL1.Attr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Attr, bits [7:0]

Profiling Buffer stage 1 memory type and attributes. Defines the memory type and, for Normal memory, the cacheability attributes, for memory addressed by the Profiling Buffer.

The encoding of this field is the same as that of a MAIR_ELx.Attr<n> field, as follows:

Attr		Meaning
`0b0000`dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
`0b0000`dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
`0b0000`dd1x		UNPREDICTABLE.
0boooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
`0b01000000`		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
`0b10100000`		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
`0b11110000`		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
`0b`xxxx`0000`	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

`dd` is encoded as follows:

‘dd’	Meaning
`0b00`	Device-nGnRnE memory.
`0b01`	Device-nGnRE memory.
`0b10`	Device-nGRE memory.
`0b11`	Device-GRE memory.

`oooo` is encoded as follows:

‘oooo’		Meaning
`0b0000`		See encoding of Attr.
`0b00`RW	where RW != 00	Normal memory, Outer Write-Through Transient.
`0b0100`		Normal memory, Outer Non-cacheable.
`0b01`RW	where RW != 00	Normal memory, Outer Write-Back Transient.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'oooo'	Meaning
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

If the Profiling Buffer pointers specify virtual addresses, the memory attributes are defined by the translation tables and this field is ignored.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMBMAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBMAR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SPE_nVM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS4 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMBMAR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS4 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMBMAR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS4 == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS4 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMBMAR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBMAR_EL1();
end;
```

```
MSR PMBMAR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b101

```
if !IsFeatureImplemented(FEAT_SPE_nVM) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS4 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nPMBMAR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS4 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMBMAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS4 == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS4 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMBMAR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMBMAR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.4 PMBPTR_EL1, Profiling Buffer Write Pointer Register

The PMBPTR_EL1 characteristics are:

Purpose

Defines the current write pointer for the profiling buffer.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBPTR_EL1 are UNDEFINED.

Attributes

PMBPTR_EL1 is a 64-bit register.

Field descriptions

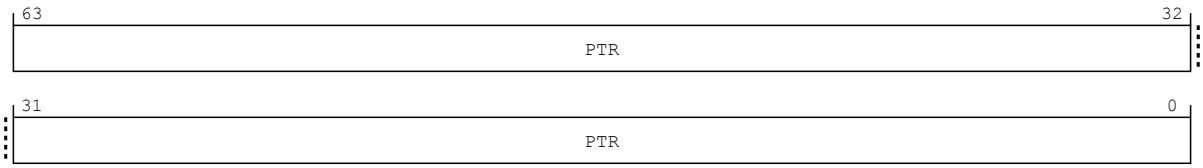

PTR, bits [63:0]

Current write address. Defines the virtual address of the next entry to be written to the buffer.

If PMBIDR_EL1.Align is not zero, then it is IMPLEMENTATION DEFINED whether bits [M-1:0] are RES0 or read/write, where M is an integer between 1 and PMBIDR_EL1.Align inclusive.

The architecture places restrictions on the values software can write to the pointer when the SPU is not in Discard mode. For more information see Restrictions on the current write pointer.

On a management interrupt, PMBPTR_EL1 is frozen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMBPTR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBPTR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b001

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
             ↪HDFGRTR_EL2().PMBPTR_EL1 == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
            X{64}(t) = NVMem(0x810);
        else
            X{64}(t) = PMBPTR_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMBPTR_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = PMBPTR_EL1();
    end;
```

MSR PMBPTR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b001

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMBPTR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x810) = X{64}(t);
    else
        PMBPTR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMBPTR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMBPTR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.5 PMBSR_EL1, Profiling Buffer Status/syndrome Register (EL1)

The PMBSR_EL1 characteristics are:

Purpose

Provides syndrome information to software for a Profiling Buffer management event.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBSR_EL1 are UNDEFINED.

Attributes

PMBSR_EL1 is a 64-bit register.

Field descriptions

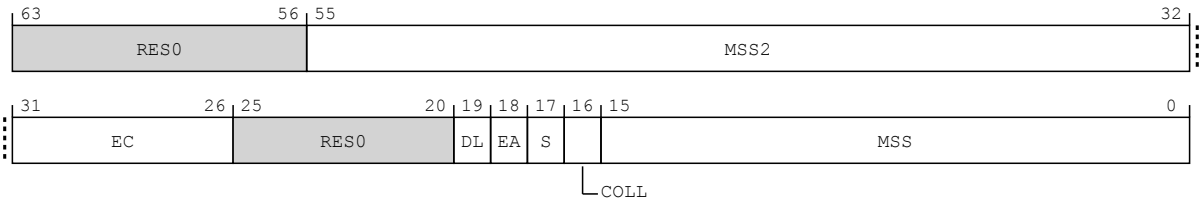

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other Profiling Buffer management events

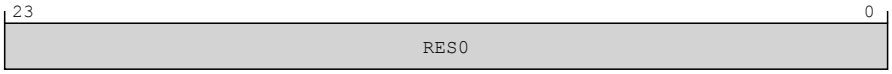

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer

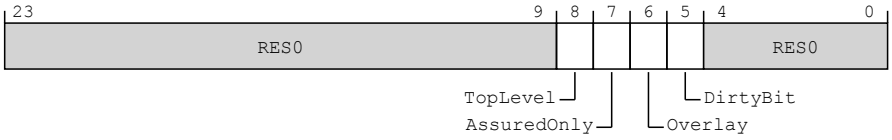

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, PMBSR_EL1.EC == '100101', and GetPMBSR_EL1_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetPMBSR_EL1_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetPMBSR_EL1_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to Profiling Buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the Profiling Buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other Profiling Buffer management event. All Profiling Buffer management events other than those described by the other defined Event class codes.	MSS encoding for other Profiling Buffer management events	MSS2 encoding for other Profiling Buffer management events	
0b011110	Granule Protection Check fault on write to Profiling Buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to Profiling Buffer	MSS2 encoding for Granule Protection Check faults on write to Profiling Buffer	FEAT_RME is implemented
0b011111	Profiling Buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to Profiling Buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	
0b100101	Stage 2 Data Abort on write to Profiling Buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [25:20]

Reserved, RES0.

DL, bit [19]

Partial record lost. Following a buffer management event, indicates whether the last record written to the Profiling Buffer is complete. PMBSR_EL1.DL is also set to 1 when an asynchronous External abort is reported to the SPU.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DL	Meaning
0b0	PMBPTR_EL1 points to the first byte after the last complete record written to the Profiling Buffer by the SPU, and no asynchronous External abort has been reported to the SPU.
0b1	Part of a record was lost because of a Profiling Buffer management event, or an asynchronous External abort has been reported to the SPU. PMBPTR_EL1 might not be the address of the first byte after the last complete sample record written to the Profiling Buffer by the SPU.

When the SPU sets this bit to 1, software must not assume that there is any valid data between the end of the last complete record and PMBPTR_EL1. If software restarts profiling from the saved PMBPTR_EL1 value, then this might result in buffer contents that software cannot parse. If PMBSR_EL1.EA is also set to 1 by the SPU, then software must not assume that any valid data has been written to the Profiling Buffer.

This bit is RES0 if there are no circumstances when the PE would ever set this bit to 1 as a result of a Profiling Buffer management event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External abort.

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Statistical Profiling Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Statistical Profiling Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. When FEAT_SPEv1p3 is implemented, this field is not set to 1 by the PE for any other Profiling Buffer management event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Service. Indicates that a Profiling Buffer management event has been recorded.

S	Meaning
0b0	No Profiling Buffer management event for EL1 has been recorded.
0b1	A Profiling Buffer management event for EL1 has been recorded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE_EXC is implemented, this field indicates a management event for EL1.

If FEAT_SPE_EXC is implemented and the SPE Profiling exception for EL1 is enabled, then when this field is 1, an SPE Profiling exception for EL1 is pending

If FEAT_SPE_EXC is not implemented or the SPE Profiling exception for EL1 is disabled, then this field drives the **PMBIRQ** Profiling Buffer interrupt request signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COLL, bit [16]

Collision detected.

COLL	Meaning
0b0	No collision events detected.
0b1	At least one collision event was recorded.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the Profiling Buffer management event.

The syndrome contents for each Profiling Buffer management event are described in the following sections.

MSS encoding for other Profiling Buffer management events

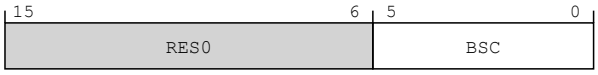

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Profiling Buffer status code

BSC	Meaning
0b000000	Collection not stopped, or access not allowed.
0b000001	Profiling Buffer filled.
0b000100	Buffer size. The requested Profiling Buffer size was too large.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer

15 … 6	5 … 0
RES0	FSC

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

It is IMPLEMENTATION DEFINED whether each of the Access Flag fault, asynchronous External abort and synchronous External abort, Alignment fault, and TLB Conflict abort values can be generated by the PE. For more information see ‘Faults and Watchpoints’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to Profiling Buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15	0
RES0	

Bits [15:0]

Reserved, RES0.

MSS encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason

15	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Accessing PMBSR_EL1

When the Effective value of HCR_EL2.E2H is 1 and FEAT_SPE_EXC is implemented, without explicit synchronization, accesses from EL3 using the accessor name `PMBSR_EL1` or `PMBSR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (EffectivePMSCR_EL2_EE() == '00' || PMSCR_EL1().EE ==
         ↪'00' || EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x820);
    else
        X{64}(t) = PMBSR_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectivePMSCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        X{64}(t) = PMBSR_EL2();
    else
        X{64}(t) = PMBSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBSR_EL1();
end;
```

MSR PMBSR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (EffectivePMSCR_EL2_EE() == '00' || PMSCR_EL1().EE ==
         ↪'00' || EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x820) = X{64}(t);
    else
        PMBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectivePMSCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        PMBSR_EL2() = X{64}(t);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PMBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMBSR_EL1() = X{64}(t);
end;
```

When FEAT_SPE_EXC is implemented and FEAT_VHE is implemented MRS <Xt>, PMBSR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x820);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMBSR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = PMBSR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_SPE_EXC is implemented and FEAT_VHE is implemented MSR PMBSR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x820) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            PMBSR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        PMBSR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.6 PMBSR_EL2, Profiling Buffer Syndrome Register (EL2)

The PMBSR_EL2 characteristics are:

Purpose

Provides syndrome information to software for a Profiling Buffer management event.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_SPE_EXC is implemented. Otherwise, direct accesses to PMBSR_EL2 are UNDEFINED.

Attributes

PMBSR_EL2 is a 64-bit register.

Field descriptions

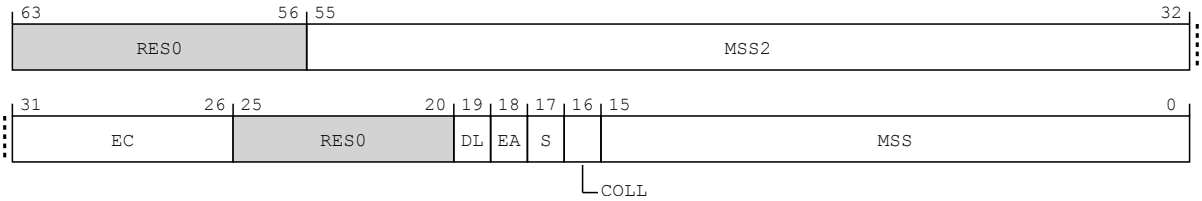

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other Profiling Buffer management events

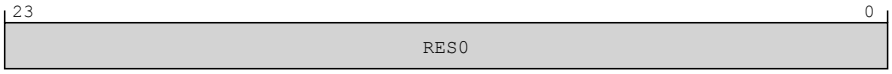

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer

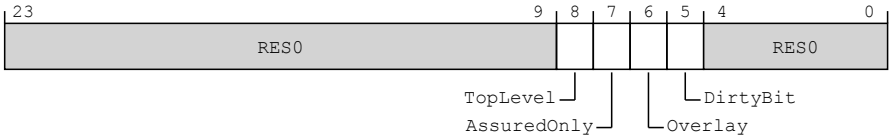

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, PMBSR_EL2.EC == '100101', and GetPMBSR_EL2_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetPMBSR_EL2_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetPMBSR_EL2_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to Profiling Buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the Profiling Buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other Profiling Buffer management event. All Profiling Buffer management events other than those described by the other defined Event class codes.	MSS encoding for other Profiling Buffer management events	MSS2 encoding for other Profiling Buffer management events	
0b011110	Granule Protection Check fault on write to Profiling Buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to Profiling Buffer	MSS2 encoding for Granule Protection Check faults on write to Profiling Buffer	FEAT_RME is implemented
0b011111	Profiling Buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to Profiling Buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	
0b100101	Stage 2 Data Abort on write to Profiling Buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [25:20]

Reserved, RES0.

DL, bit [19]

Partial record lost. Following a buffer management event, indicates whether the last record written to the Profiling Buffer is complete. PMBSR_EL2.DL is also set to 1 when an asynchronous External abort is reported to the SPU.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DL	Meaning
0b0	PMBPTR_EL1 points to the first byte after the last complete record written to the Profiling Buffer by the SPU, and no asynchronous External abort has been reported to the SPU.
0b1	Part of a record was lost because of a Profiling Buffer management event, or an asynchronous External abort has been reported to the SPU. PMBPTR_EL1 might not be the address of the first byte after the last complete sample record written to the Profiling Buffer by the SPU.

When the SPU sets this bit to 1, software must not assume that there is any valid data between the end of the last complete record and PMBPTR_EL1. If software restarts profiling from the saved PMBPTR_EL1 value, then this might result in buffer contents that software cannot parse. If PMBSR_EL2.EA is also set to 1 by the SPU, then software must not assume that any valid data has been written to the Profiling Buffer.

This bit is RES0 if there are no circumstances when the PE would ever set this bit to 1 as a result of a Profiling Buffer management event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External abort.

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Statistical Profiling Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Statistical Profiling Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. When FEAT_SPEv1p3 is implemented, this field is not set to 1 by the PE for any other Profiling Buffer management event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Service. Indicates that a Profiling Buffer management event has been recorded.

S	Meaning
0b0	No Profiling Buffer management event for EL2 has been recorded.
0b1	A Profiling Buffer management event for EL2 has been recorded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE_EXC is implemented, this field indicates a management event for EL2.

If the SPE Profiling exception for EL2 is enabled, then when this field is 1, an SPE Profiling exception for EL2 is pending

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COLL, bit [16]

Collision detected.

COLL	Meaning
0b0	No collision events detected.
0b1	At least one collision event was recorded.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the Profiling Buffer management event.

The syndrome contents for each Profiling Buffer management event are described in the following sections.

MSS encoding for other Profiling Buffer management events

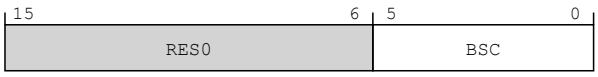

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Profiling Buffer status code

BSC	Meaning
0b000000	Collection not stopped, or access not allowed.
0b000001	Profiling Buffer filled.
0b000100	Buffer size. The requested Profiling Buffer size was too large.

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

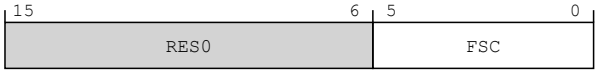

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

It is IMPLEMENTATION DEFINED whether each of the Access Flag fault, asynchronous External abort and synchronous External abort, Alignment fault, and TLB Conflict abort values can be generated by the PE. For more information see ‘Faults and Watchpoints’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to Profiling Buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15	0
RES0	

Bits [15:0]

Reserved, RES0.

MSS encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason

15	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Accessing PMBSR_EL2

When the Effective value of HCR_EL2.E2H is 1 and FEAT_SPE_EXC is implemented, without explicit synchronization, accesses from EL2 using the accessor name `PMBSR_EL2` or `PMBSR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBSR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE_EXC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().PMSEE == '00' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().PMSEE == '00' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMBSR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBSR_EL2();
end;
```

MSR PMBSR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE_EXC) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().PMSEE == '00' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().PMSEE == '00' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMBSR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMBSR_EL2() = X{64}(t);
end;
```

MRS <Xt>, PMBSR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (EffectivePMSCR_EL2_EE() == '00' || PMSCR_EL1().EE ==
         ↪'00' || EffectiveHCR_EL2_NVx() == '111') then
        X{64}(t) = NVMem(0x820);
    else
        X{64}(t) = PMBSR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectivePMSCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        X{64}(t) = PMBSR_EL2();
    else
        X{64}(t) = PMBSR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBSR_EL1();
end;
```

```
MSR PMBSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMBSR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().E2PB IN {'x0'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} && (EffectivePMSCR_EL2_EE() == '00' || PMSCR_EL1().EE ==
     ↪'00' || EffectiveHCR_EL2_NVx() == '111') then
        NVMem(0x820) = X{64}(t);
    else
        PMBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectivePMSCR_EL2_EE() != '00' && ELIsInHost(EL2) then
        PMBSR_EL2() = X{64}(t);
    else
        PMBSR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMBSR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.7 PMBSR_EL3, Profiling Buffer Syndrome Register (EL3)

The PMBSR_EL3 characteristics are:

Purpose

Provides syndrome information to software for a Profiling Buffer management event.

Configuration

This register is present only when FEAT_SPE_EXC is implemented and EL3 is implemented. Otherwise, direct accesses to PMBSR_EL3 are UNDEFINED.

Attributes

PMBSR_EL3 is a 64-bit register.

Field descriptions

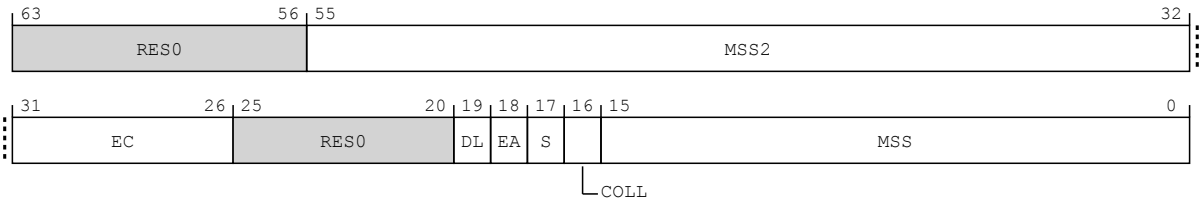

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other Profiling Buffer management events

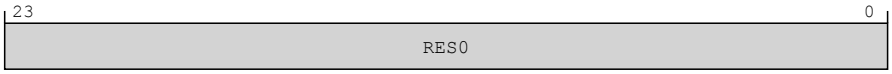

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer

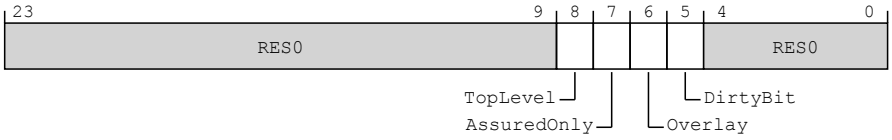

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, PMBSR_EL3.EC == '100101', and GetPMBSR_EL3_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetPMBSR_EL3_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetPMBSR_EL3_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to Profiling Buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the Profiling Buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other Profiling Buffer management event. All Profiling Buffer management events other than those described by the other defined Event class codes.	MSS encoding for other Profiling Buffer management events	MSS2 encoding for other Profiling Buffer management events	
0b011110	Granule Protection Check fault on write to Profiling Buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to Profiling Buffer	MSS2 encoding for Granule Protection Check faults on write to Profiling Buffer	FEAT_RME is implemented
0b011111	Profiling Buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to Profiling Buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	
0b100101	Stage 2 Data Abort on write to Profiling Buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [25:20]

Reserved, RES0.

DL, bit [19]

Partial record lost. Following a buffer management event, indicates whether the last record written to the Profiling Buffer is complete. PMBSR_EL3.DL is also set to 1 when an asynchronous External abort is reported to the SPU.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DL	Meaning
0b0	PMBPTR_EL1 points to the first byte after the last complete record written to the Profiling Buffer by the SPU, and no asynchronous External abort has been reported to the SPU.
0b1	Part of a record was lost because of a Profiling Buffer management event, or an asynchronous External abort has been reported to the SPU. PMBPTR_EL1 might not be the address of the first byte after the last complete sample record written to the Profiling Buffer by the SPU.

When the SPU sets this bit to 1, software must not assume that there is any valid data between the end of the last complete record and PMBPTR_EL1. If software restarts profiling from the saved PMBPTR_EL1 value, then this might result in buffer contents that software cannot parse. If PMBSR_EL3.EA is also set to 1 by the SPU, then software must not assume that any valid data has been written to the Profiling Buffer.

This bit is RES0 if there are no circumstances when the PE would ever set this bit to 1 as a result of a Profiling Buffer management event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External abort.

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Statistical Profiling Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Statistical Profiling Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. When FEAT_SPEv1p3 is implemented, this field is not set to 1 by the PE for any other Profiling Buffer management event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Service. Indicates that a Profiling Buffer management event has been recorded.

S	Meaning
0b0	No Profiling Buffer management event for EL3 has been recorded.
0b1	A Profiling Buffer management event for EL3 has been recorded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE_EXC is implemented, this field indicates a management event for EL3.

If the SPE Profiling exception for EL3 is enabled, then when this field is 1, an SPE Profiling exception for EL3 is pending

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COLL, bit [16]

Collision detected.

COLL	Meaning
0b0	No collision events detected.
0b1	At least one collision event was recorded.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the Profiling Buffer management event.

The syndrome contents for each Profiling Buffer management event are described in the following sections.

MSS encoding for other Profiling Buffer management events

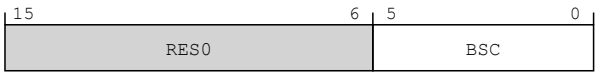

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Profiling Buffer status code

BSC	Meaning
0b000000	Collection not stopped, or access not allowed.
0b000001	Profiling Buffer filled.
0b000100	Buffer size. The requested Profiling Buffer size was too large.

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to Profiling Buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

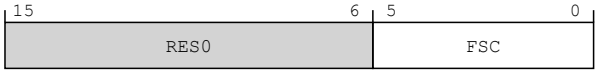

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

It is IMPLEMENTATION DEFINED whether each of the Access Flag fault, asynchronous External abort and synchronous External abort, Alignment fault, and TLB Conflict abort values can be generated by the PE. For more information see ‘Faults and Watchpoints’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to Profiling Buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15	0
RES0	

Bits [15:0]

Reserved, RES0.

MSS encoding for Profiling Buffer management event for an IMPLEMENTATION DEFINED reason

15	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Accessing PMBSR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMBSR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1001	0b1010	0b011

```
if !(IsFeatureImplemented(FEAT_SPE_EXC) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMBSR_EL3();
end;
```

```
MSR PMBSR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1001	0b1010	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_SPE_EXC) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    PMBSR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.8 PMSCR_EL1, Statistical Profiling Control Register (EL1)

The PMSCR_EL1 characteristics are:

Purpose

Provides EL1 controls for Statistical Profiling.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSCR_EL1 are UNDEFINED.

Attributes

PMSCR_EL1 is a 64-bit register.

Field descriptions

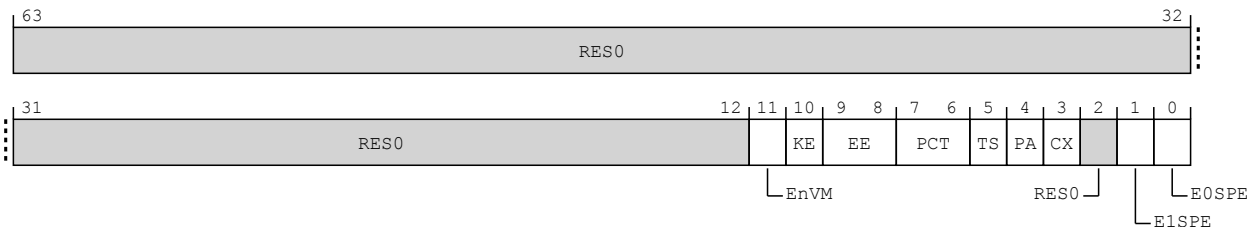

Bits [63:12]

Reserved, RES0.

EnVM, bit [11]

When FEAT_SPE_nVM is implemented and FEAT_NV is implemented:

Reserved for software use in nested virtualization. See also PMSCR_EL2.EnVM.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

KE, bit [10]

When FEAT_SPE_EXC is implemented:

Kernel exception enable for SPE Profiling exceptions taken to EL1.

KE	Meaning
0b0	SPE Profiling exceptions taken to EL1 are always masked at EL1.
0b1	Enabled SPE Profiling exceptions taken to EL1 are masked at EL1 when PSTATE.PM is 1 and unmasked when PSTATE.PM is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EE, bits [9:8]

When FEAT_SPE_EXC is implemented:

Exception Enable.

EE	Meaning	Applies when
`0b00`	Disabled. SPE Profiling exceptions for EL1 are disabled. All of the following apply: • Unless enabled by a higher Exception level, SPE Profiling exceptions are not generated. • PMBSR_EL1.S drives the interrupt request signal **PMBIRQ**. • Accesses to PMBSR_EL1 at EL1 ignore the value of HCR_EL2.NV1.	
`0b01`	Reserved for software use in nested virtualization. Behaves as `0b00` for the purpose of controlling the SPE Profiling exception and interrupt request signal **PMBIRQ**, and as `0b11` for the purpose of accesses to PMBSR_EL1.	FEAT_NV is implemented
`0b10`	Reserved for software use in nested virtualization. Behaves as `0b11` for the purposes of controlling the SPE Profiling exception and interrupt request signal **PMBIRQ**, and accesses to PMBSR_EL1.	FEAT_NV is implemented
`0b11`	Enabled. SPE Profiling exceptions for EL1 are enabled, as follows: • All Profiling Buffer management events are recorded in PMBSR_EL1, unless they are configured to be recorded in PMBSR_EL3 by MDCR_EL3.PMSEE or PMBSR_EL2 by PMSCR_EL2.EE. • SPE Profiling exceptions are generated and taken to EL1 when unmasked and PMBSR_EL1.S is 1, unless the Effective value of HCR_EL2.TGE is 1, in which case the exception is taken to EL2. • The interrupt request signal **PMBIRQ** is not asserted.	

For more information on the values reserved for software use in nested virtualization, see PMSCR_EL2.EE.

If the Effective value of PMSCR_EL2.EE is `0b00`, then the Effective value of PMSCR_EL1.EE is `0b00`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PCT, bits [7:6]

When EL2 is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Physical Timestamp. If timestamp sampling is enabled and the Profiling Buffer owning Exception level is EL1, requests which timestamp counter value is collected.

If FEAT_ECV is implemented, this is a two-bit field as shown. Otherwise, bit[7] is RES0.

PCT	Meaning	Applies when
0b00	Virtual timestamp. The collected timestamp is the physical counter minus the value of CNTVOFF_EL2.	
0b01	Physical timestamp. The collected timestamp is the physical counter.	
0b11	Guest physical timestamp. The collected timestamp is the physical counter minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • SCR_EL3.ECVEn == 0. • CNTHCTL_EL2.ECV == 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented

When EL2 is implemented, all of the following apply:

- If the Profiling Buffer owning Exception level is EL1 and EL2 is enabled in the owning Security state, then the value of PMSCR_EL2.PCT might override or modify the meaning of this field.
- If the Profiling Buffer owning Exception level is EL2, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Physical Timestamp. Reserved. This field reads as 0b01 and ignores writes. Software should treat this field as UNK/SBZP.

When EL2 is not implemented, the Effective values of CNTVOFF_EL2 and CNTPOFF_EL2 are zero, meaning the virtual counter and physical counter have the same value.

TS, bit [5]

Timestamp sample enable. Enables recording of a Timestamp packet when the owning Exception level is EL1.

TS	Meaning
0b0	Timestamp packet recording disabled.
0b1	Timestamp packet recording enabled.

If the Profiling Buffer owning Exception level is EL2, then this field is ignored by the PE. For more information, see Controlling the data that is collected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PA, bit [4]

Physical Address sample enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PA	Meaning
0b0	Physical addresses are not collected.
0b1	Physical addresses are collected.

When EL2 is implemented, all of the following apply:

- If the Profiling Buffer owning Exception level is EL1, then this field is combined with the Effective value PMSCR_EL2.PA to determine whether Physical addresses are collected.
- If the Profiling Buffer owning Exception level is EL2, then this field is ignored by the PE.

For more information, see Controlling the data that is collected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CX, bit [3]

CONTEXTIDR_EL1 sample enable. Enables recording of a Context packet containing the value of CONTEXTIDR_EL1.

CX	Meaning
0b0	CONTEXTIDR_EL1 recording disabled.
0b1	CONTEXTIDR_EL1 recording enabled.

The PE ignores the value of this field and CONTEXTIDR_EL1 is not recorded when any of the following apply:

- The sampled operation executes at EL2.
- The sampled operation executes at EL0 and the Effective value of HCR_EL2.TGE is 1.

For more information, see Controlling the data that is collected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E1SPE, bit [1]

EL1 Statistical Profiling Enable.

E1SPE	Meaning
0b0	Sampling disabled at EL1.
0b1	Sampling enabled at EL1.

If the Effective value of HCR_EL2.TGE is 1, then this field is ignored by the PE.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E0SPE, bit [0]

EL0 Statistical Profiling Enable. Controls sampling at EL0 when the Effective value of HCR_EL2.TGE is 0 or if EL2 is disabled or not implemented.

E0SPE	Meaning
0b0	Sampling disabled at EL0.
0b1	Sampling enabled at EL0.

If the Effective value of HCR_EL2.TGE is 1, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x828);
    else
        X{64}(t) = PMSCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PMSCR_EL2();
    else
        X{64}(t) = PMSCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSCR_EL1();
end;
```

```
MSR PMSCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x828) = X{64}(t);
    else
        PMSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PMSCR_EL2() = X{64}(t);
    else
        PMSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_VHE is implemented MRS <Xt>, PMSCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x828);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = PMSCR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = PMSCR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR PMSCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x828) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
            Undefined();
        elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            PMSCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        PMSCR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.9 PMSCR_EL2, Statistical Profiling Control Register (EL2)

The PMSCR_EL2 characteristics are:

Purpose

Provides EL2 controls for Statistical Profiling.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSCR_EL2 are UNDEFINED.

Attributes

PMSCR_EL2 is a 64-bit register.

Field descriptions

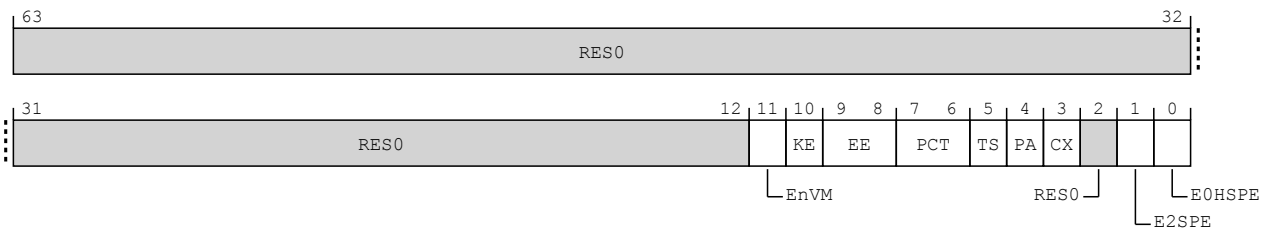

Bits [63:12]

Reserved, RES0.

EnVM, bit [11]

When FEAT_SPE_nVM is implemented:

Enable use of physical address Profiling Buffer pointers.

EnVM	Meaning
0b0	Use of physical address Profiling Buffer pointers is disabled. The PE behaves as if PMBLIMITR_EL1.nVM is 0.
0b1	Use of physical address Profiling Buffer pointers is permitted.

If EL2 is disabled in the owning Security state, or the Profiling Buffer owning Exception level is EL2, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

KE, bit [10]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_SPE_EXC is implemented:

Kernel exception enable for SPE Profiling exceptions taken to EL2.

KE	Meaning
0b0	SPE Profiling exceptions taken to EL2 are always masked at EL2.
0b1	Enabled SPE Profiling exceptions taken to EL2 are masked at EL2 when PSTATE.PM is 1 and unmasked when PSTATE.PM is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EE, bits [9:8]

When FEAT_SPE_EXC is implemented:

Exception Enable.

EE	Meaning
0b00	Disabled. SPE Profiling exceptions for EL2 and EL1 are disabled. All of the following apply: • No Profiling Buffer management events are recorded in PMBSR_EL2. • Unless enabled by a higher Exception level, SPE Profiling exceptions are not generated. • PMBSR_EL1.S drives the interrupt request signal **PMBIRQ**. • Accesses to PMBSR_EL1 at EL1 ignore the value of HCR_EL2.NV1 and accesses to PMBSR_EL1 at EL2 ignore the value of HCR_EL2.E2H.
0b01	Delegated. SPE Profiling exceptions for EL2 are disabled, but might be enabled for EL1 by PMSCR_EL1.EE. All of the following apply: • No Profiling Buffer management events are recorded in PMBSR_EL2. • PMBSR_EL2.S is ignored and SPE Profiling exceptions are not taken to EL2, other than for the case when the Effective value of HCR_EL2.TGE is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EE	Meaning
0b10	Enabled. SPE Profiling exceptions for EL2 are enabled for Profiling Buffer management events targeting EL2, as follows: • Profiling Buffer management events due to a fault on a write to the Profiling Buffer that would generate a Data Abort exception taken to EL2 if generated by a store instruction executed at the owning Exception level are recorded in PMBSR_EL2, unless they are configured to be recorded in PMBSR_EL3 by MDCR_EL3.PMSEE. If the Profiling Buffer owning Exception level is EL2, then this means any fault on a write to the Profiling Buffer. If the Profiling Buffer owning Exception level is EL1, then this means any of the following faults on a write to the Profiling Buffer: • Stage 2 faults. • If HCR_EL2.TEA is 1, synchronous External aborts. • If HCR_EL2.GPF is 1, Granule Protection Faults (GPFs). • Profiling Buffer management events due to Granule Protection Check faults other than GPFs on a write to the Profiling Buffer are recorded in PMBSR_EL2, unless they are configured to be recorded in PMBSR_EL3 by MDCR_EL3.PMSEE. • SPE Profiling exceptions are generated and taken to EL2 when unmasked and PMBSR_EL2.S is 1.
0b11	Trap all. SPE Profiling exceptions for EL2 are enabled for all Profiling Buffer management events, as follows: • All Profiling Buffer management events are recorded in PMBSR_EL2, unless they are configured to be recorded in PMBSR_EL3 by MDCR_EL3.PMSEE. • SPE Profiling exceptions are generated and taken to EL2 when unmasked and PMBSR_EL2.S is 1.

If the Effective value of MDCR_EL3.PMSEE is `0b00`, then the Effective value of PMSCR_EL2.EE is `0b00`. Otherwise, if EL2 is not implemented or the Effective value of SCR_EL3.{NS, EEL2} is {0, 0}, then the Effective value of PMSCR_EL2.EE is `0b01`.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PCT, bits [7:6]

Physical Timestamp. If timestamp sampling is enabled, determines which counter is collected. The behavior depends on the Profiling Buffer owning Exception level.

If FEAT_ECV is implemented, this is a two-bit field as shown. Otherwise, bit[7] is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCT	Meaning	Applies when
0b00	Virtual timestamp. The collected timestamp is the physical counter minus a virtual offset. If the Profiling Buffer owning Exception level is EL2 and any of the following are true, then the virtual offset is zero: • The sampled operation executed at EL2 and the Effective value of HCR_EL2.E2H is 1. • The sampled operation executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}. Otherwise, the virtual offset is the value of CNTVOFF_EL2.	
0b01	If the Profiling Buffer owning Exception level is EL1, then the timestamp value is selected by PMSCR_EL1.PCT. Otherwise, physical timestamp. The collected timestamp is the physical counter.	
0b11	If the Profiling Buffer owning Exception level is EL1 and PMSCR_EL1.PCT is 0b00, then guest virtual timestamp. The collected timestamp is the physical counter minus the value of CNTVOFF_EL2. Otherwise, guest physical timestamp. The collected timestamp is the physical counter minus a physical offset. If any of the following are true, then the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • SCR_EL3.ECVEn is 0. • CNTHCTL_EL2.ECV is 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented

All other values are reserved.

If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of this field is 0b01, other than for a direct read of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TS, bit [5]

Timestamp sample enable. Enables recording of a Timestamp packet when the owning Exception level is EL2.

TS	Meaning
0b0	Timestamp packet recording disabled.
0b1	Timestamp packet recording enabled.

If the Profiling Buffer owning Exception level is EL1, then this field is ignored by the PE. For more information, see Controlling the data that is collected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PA, bit [4]

Physical Address Sample Enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PA	Meaning
0b0	Physical addresses are not collected.
0b1	Physical addresses are collected.

If the Profiling Buffer owning Exception level is EL2, then this field determines whether physical addresses are collected.

If the Profiling Buffer owning Exception level is EL1, and EL2 is enabled in the owning Security state, then this field is combined with PMSCR_EL1.PA to determine whether physical addresses are collected.

If EL2 is not implemented or EL2 is disabled in the owning Security state, then the Effective value of this field is 1.

For more information, see Controlling the data that is collected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CX, bit [3]

CONTEXTIDR_EL2 sample enable. Enables recording of a Context packet containing the value of CONTEXTIDR_EL2.

CX	Meaning
0b0	CONTEXTIDR_EL2 recording disabled.
0b1	CONTEXTIDR_EL2 recording enabled.

The PE ignores the value of this field and CONTEXTIDR_EL2 is not recorded when EL2 is not implemented or EL2 is disabled in the current Security state.

For more information, see Controlling the data that is collected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E2SPE, bit [1]

EL2 Statistical Profiling Enable.

E2SPE	Meaning
0b0	Sampling disabled at EL2.
0b1	Sampling enabled at EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MDCR_EL2.E2PB != ‘00’, access to this field is RES0.
- Otherwise, access to this field is RW.

E0HSPE, bit [0]

EL0 Statistical Profiling Enable.

E0HSPE	Meaning
0b0	Sampling disabled at EL0.
0b1	Sampling enabled at EL0.

If the Effective value of HCR_EL2.TGE is 0, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MDCR_EL2.E2PB != ‘00’, access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing PMSCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSCR_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSCR_EL2();
end;
```

MSR PMSCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSCR_EL2() = X{64}(t);
end;
```

MRS <Xt>, PMSCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x828);
    else
        X{64}(t) = PMSCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = PMSCR_EL2();
    else
        X{64}(t) = PMSCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSCR_EL1();
end;
```

MSR PMSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x828) = X{64}(t);
    else
        PMSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        PMSCR_EL2() = X{64}(t);
    else
        PMSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.10 PMSDSFR_EL1, Sampling Data Source Filter Register

The PMSDSFR_EL1 characteristics are:

Purpose

Controls sample filtering by Data Source.

Configuration

This register is present only when FEAT_SPE_FDS is implemented. Otherwise, direct accesses to PMSDSFR_EL1 are UNDEFINED.

Attributes

PMSDSFR_EL1 is a 64-bit register.

Field descriptions

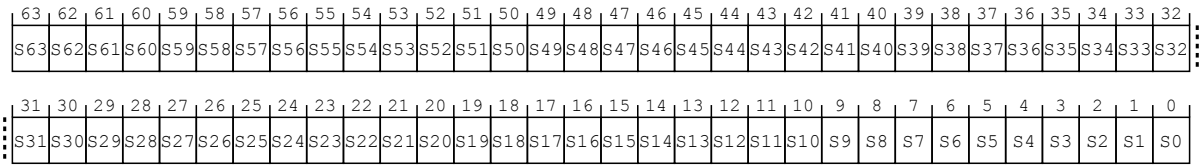

S<m> , bits [m], for m = 63 to 0

S[<m>] is the Data Source filter for IMPLEMENTATION DEFINED Data Source <m>.

S<m>	Meaning
0b0	If PMSFCR_EL1.FDS is 1, do not record load operations that have bits [5:0] of the Data Source packet set to <m>.
0b1	Load operations with Data Source <m> are unaffected by PMSFCR_EL1.FDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When filtering on Data Source <m> is not supported, access to this field is RAZ/WI.

Accessing PMSDSFR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSDSFR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SPE_FDS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS3 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGRTR2_EL2().nPMSDSFR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS3 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x858);
    else
        X{64}(t) = PMSDSFR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS3 == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS3 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSDSFR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSDSFR_EL1();
end;
```

```
MSR PMSDSFR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SPE_FDS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS3 == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HDFGWTR2_EL2().nPMSDSFR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS3 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x858) = X{64}(t);
    else
        PMSDSFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMS3 == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMS3 == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSDSFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSDSFR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.11 PMSEVFR_EL1, Sampling Event Filter Register

The PMSEVFR_EL1 characteristics are:

Purpose

Controls sample filtering by events. The overall filter is the logical AND of these filters. For example, if PMSEVFR_EL1.E[3] and PMSEVFR_EL1.E[5] are both set to 1, only samples that have both event 3 (Level 1 unified or data cache refill) and event 5 (TLB walk) set to 1 are recorded.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSEVFR_EL1 are UNDEFINED.

Attributes

PMSEVFR_EL1 is a 64-bit register.

Field descriptions

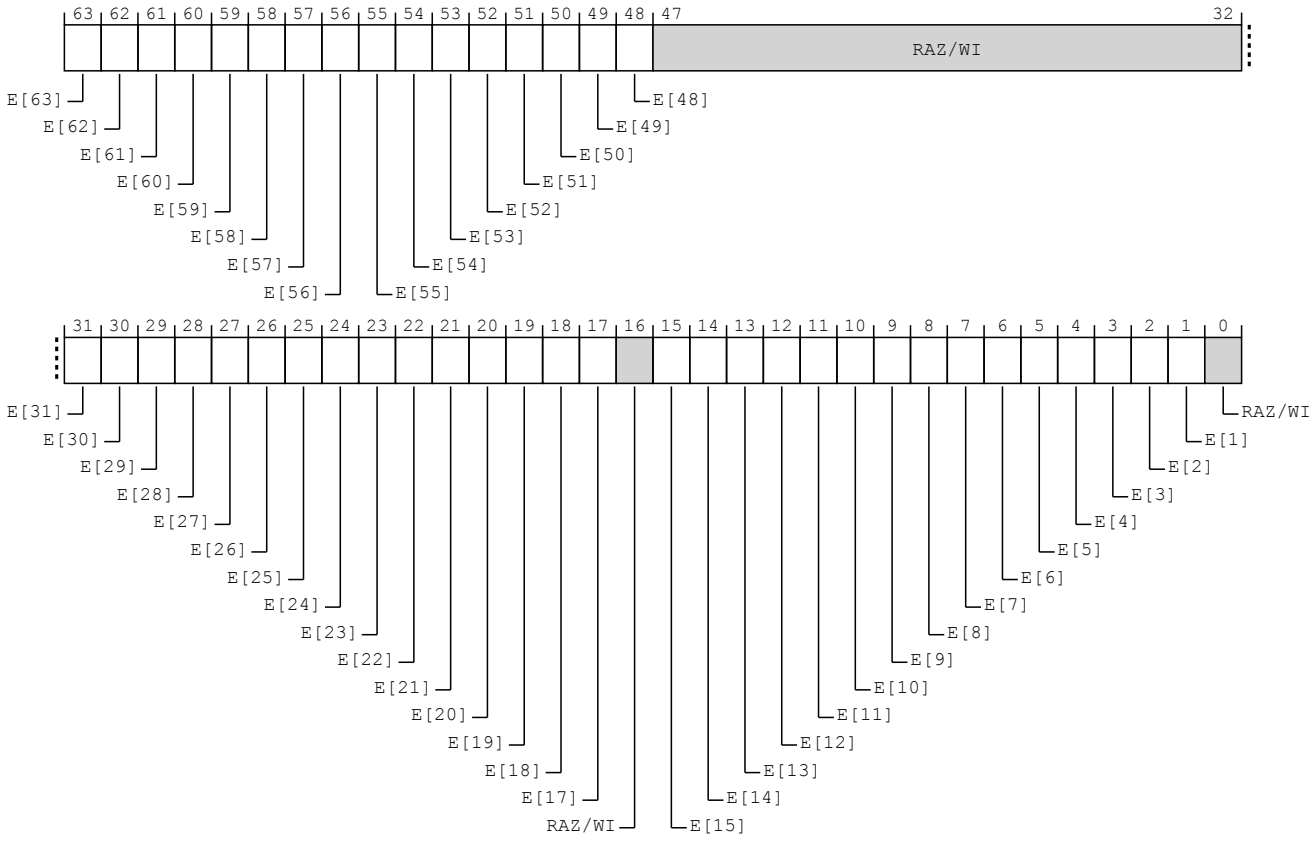

E[63], bit [63]

When event 63 is implemented and filtering on event 63 is supported:

Filter on IMPLEMENTATION DEFINED event 63.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[63]	Meaning
0b0	Event 63 is ignored.
0b1	Do not record samples that have event 63 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[63] is 1, and PMSNEVFR_EL1.E[63] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[62], bit [62]

When event 62 is implemented and filtering on event 62 is supported:

Filter on IMPLEMENTATION DEFINED event 62.

E[62]	Meaning
0b0	Event 62 is ignored.
0b1	Do not record samples that have event 62 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[62] is 1, and PMSNEVFR_EL1.E[62] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[61], bit [61]

When event 61 is implemented and filtering on event 61 is supported:

Filter on IMPLEMENTATION DEFINED event 61.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[61]	Meaning
0b0	Event 61 is ignored.
0b1	Do not record samples that have event 61 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[61] is 1, and PMSNEVFR_EL1.E[61] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[60], bit [60]

When event 60 is implemented and filtering on event 60 is supported:

Filter on IMPLEMENTATION DEFINED event 60.

E[60]	Meaning
0b0	Event 60 is ignored.
0b1	Do not record samples that have event 60 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[60] is 1, and PMSNEVFR_EL1.E[60] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[59], bit [59]

When event 59 is implemented and filtering on event 59 is supported:

Filter on IMPLEMENTATION DEFINED event 59.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[59]	Meaning
0b0	Event 59 is ignored.
0b1	Do not record samples that have event 59 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[59] is 1, and PMSNEVFR_EL1.E[59] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[58], bit [58]

When event 58 is implemented and filtering on event 58 is supported:

Filter on IMPLEMENTATION DEFINED event 58.

E[58]	Meaning
0b0	Event 58 is ignored.
0b1	Do not record samples that have event 58 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[58] is 1, and PMSNEVFR_EL1.E[58] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[57], bit [57]

When event 57 is implemented and filtering on event 57 is supported:

Filter on IMPLEMENTATION DEFINED event 57.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[57]	Meaning
0b0	Event 57 is ignored.
0b1	Do not record samples that have event 57 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[57] is 1, and PMSNEVFR_EL1.E[57] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[56], bit [56]

When event 56 is implemented and filtering on event 56 is supported:

Filter on IMPLEMENTATION DEFINED event 56.

E[56]	Meaning
0b0	Event 56 is ignored.
0b1	Do not record samples that have event 56 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[56] is 1, and PMSNEVFR_EL1.E[56] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[55], bit [55]

When event 55 is implemented and filtering on event 55 is supported:

Filter on IMPLEMENTATION DEFINED event 55.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[55]	Meaning
0b0	Event 55 is ignored.
0b1	Do not record samples that have event 55 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[55] is 1, and PMSNEVFR_EL1.E[55] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[54], bit [54]

When event 54 is implemented and filtering on event 54 is supported:

Filter on IMPLEMENTATION DEFINED event 54.

E[54]	Meaning
0b0	Event 54 is ignored.
0b1	Do not record samples that have event 54 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[54] is 1, and PMSNEVFR_EL1.E[54] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[53], bit [53]

When event 53 is implemented and filtering on event 53 is supported:

Filter on IMPLEMENTATION DEFINED event 53.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[53]	Meaning
0b0	Event 53 is ignored.
0b1	Do not record samples that have event 53 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[53] is 1, and PMSNEVFR_EL1.E[53] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[52], bit [52]

When event 52 is implemented and filtering on event 52 is supported:

Filter on IMPLEMENTATION DEFINED event 52.

E[52]	Meaning
0b0	Event 52 is ignored.
0b1	Do not record samples that have event 52 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[52] is 1, and PMSNEVFR_EL1.E[52] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[51], bit [51]

When event 51 is implemented and filtering on event 51 is supported:

Filter on IMPLEMENTATION DEFINED event 51.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[51]	Meaning
0b0	Event 51 is ignored.
0b1	Do not record samples that have event 51 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[51] is 1, and PMSNEVFR_EL1.E[51] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[50], bit [50]

When event 50 is implemented and filtering on event 50 is supported:

Filter on IMPLEMENTATION DEFINED event 50.

E[50]	Meaning
0b0	Event 50 is ignored.
0b1	Do not record samples that have event 50 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[50] is 1, and PMSNEVFR_EL1.E[50] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[49], bit [49]

When event 49 is implemented and filtering on event 49 is supported:

Filter on IMPLEMENTATION DEFINED event 49.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[49]	Meaning
0b0	Event 49 is ignored.
0b1	Do not record samples that have event 49 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[49] is 1, and PMSNEVFR_EL1.E[49] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[48], bit [48]

When event 48 is implemented and filtering on event 48 is supported:

Filter on IMPLEMENTATION DEFINED event 48.

E[48]	Meaning
0b0	Event 48 is ignored.
0b1	Do not record samples that have event 48 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[48] is 1, and PMSNEVFR_EL1.E[48] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bits [47:32]

Reserved, RAZ/WI.

E[31], bit [31]

When FEAT_SPEv1p4 is not implemented, event 31 is implemented, and filtering on event 31 is supported:

Filter on IMPLEMENTATION DEFINED event 31.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[31]	Meaning
0b0	Event 31 is ignored.
0b1	Do not record samples that have event 31 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[31] is 1, and PMSNEVFR_EL1.E[31] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[30], bit [30]

When FEAT_SPEv1p4 is not implemented, event 30 is implemented, and filtering on event 30 is supported:

Filter on IMPLEMENTATION DEFINED event 30.

E[30]	Meaning
0b0	Event 30 is ignored.
0b1	Do not record samples that have event 30 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[30] is 1, and PMSNEVFR_EL1.E[30] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[29], bit [29]

When FEAT_SPEv1p4 is not implemented, event 29 is implemented, and filtering on event 29 is supported:

Filter on IMPLEMENTATION DEFINED event 29.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[29]	Meaning
0b0	Event 29 is ignored.
0b1	Do not record samples that have event 29 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[29] is 1, and PMSNEVFR_EL1.E[29] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[28], bit [28]

When FEAT_SPEv1p4 is not implemented, event 28 is implemented, and filtering on event 28 is supported:

Filter on IMPLEMENTATION DEFINED event 28.

E[28]	Meaning
0b0	Event 28 is ignored.
0b1	Do not record samples that have event 28 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[28] is 1, and PMSNEVFR_EL1.E[28] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[27], bit [27]

When FEAT_SPEv1p4 is not implemented, event 27 is implemented, and filtering on event 27 is supported:

Filter on IMPLEMENTATION DEFINED event 27.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[27]	Meaning
0b0	Event 27 is ignored.
0b1	Do not record samples that have event 27 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[27] is 1, and PMSNEVFR_EL1.E[27] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[26], bit [26]

When FEAT_SPEv1p4 is not implemented, event 26 is implemented, and filtering on event 26 is supported:

Filter on IMPLEMENTATION DEFINED event 26.

E[26]	Meaning
0b0	Event 26 is ignored.
0b1	Do not record samples that have event 26 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[26] is 1, and PMSNEVFR_EL1.E[26] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[25], bit [25]

When (FEAT_SPE_SME is implemented or FEAT_SPEv1p5 is implemented) and event 25 is implemented:

Filter on SMCU or other shared resource operation event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[25]	Meaning
0b0	SMCU or other shared resource operation event is ignored.
0b1	Do not record samples that have the SMCU or other shared resource operation event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[25] is 1, and PMSNEVFR_EL1.E[25] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPEv1p4 is not implemented, event 25 is implemented, and filtering on event 25 is supported:

Filter on IMPLEMENTATION DEFINED event 25.

E[25]	Meaning
0b0	Event 25 is ignored.
0b1	Do not record samples that have event 25 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[25] is 1, and PMSNEVFR_EL1.E[25] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[24], bit [24]

When FEAT_SPE_SME is implemented:

Filter on Streaming SVE mode event.

E[24]	Meaning
0b0	Streaming SVE mode event is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[24]	Meaning
0b1	Do not record samples that have the Streaming SVE mode event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[24] is 1, and PMSNEVFR_EL1.E[24] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPEv1p4 is not implemented, event 24 is implemented, and filtering on event 24 is supported:

Filter on IMPLEMENTATION DEFINED event 24.

E[24]	Meaning
0b0	Event 24 is ignored.
0b1	Do not record samples that have event 24 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[24] is 1, and PMSNEVFR_EL1.E[24] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[23], bit [23]

When FEAT_SPEv1p4 is implemented and event 23 is implemented:

Filter on Data snooped event.

E[23]	Meaning
0b0	Data snooped event is ignored.
0b1	Do not record samples that have the Data snooped event == 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[23] is 1, and PMSNEVFR_EL1.E[23] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[22], bit [22]

When FEAT_SPEv1p4 is implemented and event 22 is implemented:

Filter on Recently fetched event.

E[22]	Meaning
`0b0`	Recently fetched event is ignored.
`0b1`	Do not record samples that have the Recently fetched event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[22] is 1, and PMSNEVFR_EL1.E[22] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[21], bit [21]

When FEAT_SPEv1p4 is implemented and event 21 is implemented:

Filter on Cache data modified event.

E[21]	Meaning
`0b0`	Cache data modified event is ignored.
`0b1`	Do not record samples that have the Cache data modified event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[21] is 1, and PMSNEVFR_EL1.E[21] is 1, then, for each sampled operation, it is

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[20], bit [20]

When FEAT_SPEv1p4 is implemented and event 20 is implemented:

Filter on Level 2 data cache miss event.

E[20]	Meaning
0b0	Level 2 data cache miss event is ignored.
0b1	Do not record samples that have the Level 2 data cache miss event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[20] is 1, and PMSNEVFR_EL1.E[20] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[19], bit [19]

When FEAT_SPEv1p4 is implemented and event 19 is implemented:

Filter on Level 2 data cache access event.

E[19]	Meaning
0b0	Level 2 data cache access event is ignored.
0b1	Do not record samples that have the Level 2 data cache access event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[19] is 1, and PMSNEVFR_EL1.E[19] is 1, then, for each sampled operation, it is

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[18], bit [18]

When FEAT_SPEv1p1 is implemented and (FEAT_SVE is implemented or FEAT_SME is implemented):

Filter on Empty predicate event.

E[18]	Meaning
0b0	Empty predicate event is ignored.
0b1	Do not record samples that have the Empty predicate event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[18] is 1, and PMSNEVFR_EL1.E[18] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[17], bit [17]

When FEAT_SPEv1p1 is implemented and (FEAT_SVE is implemented or FEAT_SME is implemented):

Filter on Partial or empty predicate event.

E[17]	Meaning
0b0	Partial or empty predicate event is ignored.
0b1	Do not record samples that have the Partial or empty predicate event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[17] is 1, and PMSNEVFR_EL1.E[17] is 1, then, for each sampled operation, it is

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [16]

Reserved, RAZ/WI.

E[15], bit [15]

When event 15 is implemented and filtering on event 15 is supported:

Filter on IMPLEMENTATION DEFINED event 15.

E[15]	Meaning
`0b0`	Event 15 is ignored.
`0b1`	Do not record samples that have event 15 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[15] is 1, and PMSNEVFR_EL1.E[15] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[14], bit [14]

When event 14 is implemented and filtering on event 14 is supported:

Filter on IMPLEMENTATION DEFINED event 14.

E[14]	Meaning
`0b0`	Event 14 is ignored.
`0b1`	Do not record samples that have event 14 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[14] is 1, and PMSNEVFR_EL1.E[14] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[13], bit [13]

When event 13 is implemented and filtering on event 13 is supported:

Filter on IMPLEMENTATION DEFINED event 13.

E[13]	Meaning
0b0	Event 13 is ignored.
0b1	Do not record samples that have event 13 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[13] is 1, and PMSNEVFR_EL1.E[13] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[12], bit [12]

When event 12 is implemented and filtering on event 12 is supported:

Filter on IMPLEMENTATION DEFINED event 12.

E[12]	Meaning
0b0	Event 12 is ignored.
0b1	Do not record samples that have event 12 == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[12] is 1, and PMSNEVFR_EL1.E[12] is 1, then, for each sampled operation, it is

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[11], bit [11]

When FEAT_SPEv1p1 is implemented:

Filter on Misalignment event.

E[11]	Meaning
0b0	Misalignment event is ignored.
0b1	Do not record samples that have the Misalignment event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[11] is 1, and PMSNEVFR_EL1.E[11] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[10], bit [10]

When (FEAT_SPEv1p4 is implemented or filtering on event 10 is optionally supported) and event 10 is implemented:

Filter on Remote access event.

E[10]	Meaning
0b0	Remote access event is ignored.
0b1	Do not record samples that have the Remote access event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[10] is 1, and PMSNEVFR_EL1.E[10] is 1, then, for each sampled operation, it is

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[9], bit [9]

When (FEAT_SPEv1p4 is implemented or filtering on event 9 is optionally supported) and event 9 is implemented:

Filter on Last Level cache miss event.

E[9]	Meaning
0b0	Last Level cache miss event is ignored.
0b1	Do not record samples that have the Last Level cache miss event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[9] is 1, and PMSNEVFR_EL1.E[9] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[8], bit [8]

When (FEAT_SPEv1p4 is implemented or filtering on event 8 is optionally supported) and event 8 is implemented:

Filter on Last Level cache access event.

E[8]	Meaning
0b0	Last Level cache access event is ignored.
0b1	Do not record samples that have the Last Level cache access event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[8] is 1, and PMSNEVFR_EL1.E[8] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[7], bit [7]

Filter on Mispredicted event.

E[7]	Meaning
0b0	Mispredicted event is ignored.
0b1	Do not record samples that have the Mispredicted event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[7] is 1, and PMSNEVFR_EL1.E[7] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[6], bit [6]

When FEAT_SPE_FnE is implemented:

Filter on Not taken event.

E[6]	Meaning
0b0	Not taken event is ignored.
0b1	Do not record samples that have the Not taken event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[6] is 1, and PMSNEVFR_EL1.E[6] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[5], bit [5]

Filter on TLB walk event.

E[5]	Meaning
0b0	TLB walk event is ignored.
0b1	Do not record samples that have the TLB walk event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[5] is 1, and PMSNEVFR_EL1.E[5] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[4], bit [4]

When FEAT_SPEv1p4 is implemented or filtering on event 4 is optionally supported:

Filter on TLB access event.

E[4]	Meaning
0b0	TLB access event is ignored.
0b1	Do not record samples that have the TLB access event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[4] is 1, and PMSNEVFR_EL1.E[4] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[3], bit [3]

Filter on Level 1 data cache refill or miss event.

E[3]	Meaning
0b0	Level 1 data cache refill or miss event is ignored.
0b1	Do not record samples that have the Level 1 data cache refill or miss event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[3] is 1, and PMSNEVFR_EL1.E[3] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[2], bit [2]

When FEAT_SPEv1p4 is implemented or filtering on event 2 is optionally supported:

Filter on Level 1 data cache access event.

E[2]	Meaning
0b0	Level 1 data cache access event is ignored.
0b1	Do not record samples that have the Level 1 data cache access event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[2] is 1, and PMSNEVFR_EL1.E[2] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[1], bit [1]

When the PE supports sampling of speculative instructions:

Filter on Architecturally retired event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[1]	Meaning
0b0	Architecturally retired event is ignored.
0b1	Do not record samples that have the Architecturally retired event == 0.

If FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[1] is 1, and PMSNEVFR_EL1.E[1] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FE is 0.

If the PE does not support the sampling of speculative instructions, or always discards the sample record for speculative instructions, this bit reads as an UNKNOWN value and the PE ignores its value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, UNKNOWN.

Bit [0]

Reserved, RAZ/WI.

Accessing PMSEVFR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSEVFR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b101

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSEVFR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x830);
    else
        X{64}(t) = PMSEVFR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSEVFR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSEVFR_EL1();
end;
```

MSR PMSEVFR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b101

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSEVFR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x830) = X{64}(t);
    else
        PMSEVFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSEVFR_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    PMSEVFR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.12 PMSFCR_EL1, Sampling Filter Control Register

The PMSFCR_EL1 characteristics are:

Purpose

Controls sample filtering. The filter is the logical AND of the filters controlled by FDS, FnE, FL, FT, and FE bits. For example, if PMSFCR_EL1.FE is 1 and PMSFCR_EL1.FT is 1, then only samples including the selected types and the selected events will be recorded.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSFCR_EL1 are UNDEFINED.

Attributes

PMSFCR_EL1 is a 64-bit register.

Field descriptions

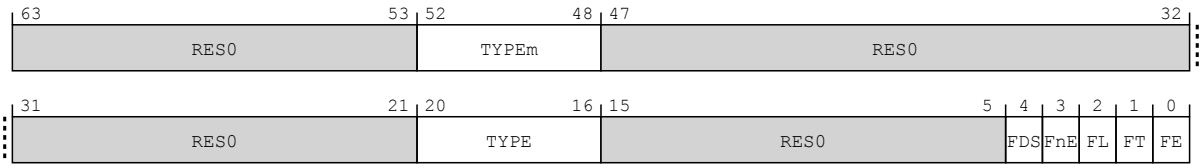

Bits [63:53]

Reserved, RES0.

TYPEm, bits [52:48]

When FEAT_SPE_EFT is implemented:

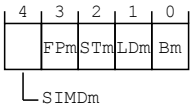

SIMDm, bit [4]

SIMD filter mask.

SIMDm	Meaning
0b0	PMSFCR_EL1.SIMD controls whether SIMD operations are recorded as part of a Boolean-OR filter with other masked operation type filter controls.
0b1	PMSFCR_EL1.SIMD controls whether SIMD operations are recorded as part of a Boolean-AND filter with other unmasked operation type filter controls.

This field is ignored by the PE when PMSFCR_EL1.FT is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FPm, bit [3]

Floating-point filter mask.

FPm	Meaning
0b0	PMSFCR_EL1.FP controls whether floating-point operations are recorded as part of a Boolean-OR filter with other masked operation type filter controls.
0b1	PMSFCR_EL1.FP controls whether floating-point operations are recorded as part of a Boolean-AND filter with other unmasked operation type filter controls.

This field is ignored by the PE when PMSFCR_EL1.FT is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

STm, bit [2]

Store filter mask.

STm	Meaning
0b0	PMSFCR_EL1.ST controls whether store operations are recorded as part of a Boolean-OR filter with other masked operation type filter controls.
0b1	PMSFCR_EL1.ST controls whether store operations are recorded as part of a Boolean-AND filter with other unmasked operation type filter controls.

This field is ignored by the PE when PMSFCR_EL1.FT is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LDm, bit [1]

Load filter mask.

LDm	Meaning
0b0	PMSFCR_EL1.LD controls whether load operations are recorded as part of a Boolean-OR filter with other masked operation type filter controls.
0b1	PMSFCR_EL1.LD controls whether load operations are recorded as part of a Boolean-AND filter with other unmasked operation type filter controls.

This field is ignored by the PE when PMSFCR_EL1.FT is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bm, bit [0]

Branch filter mask.

Bm	Meaning
0b0	PMSFCR_EL1.B controls whether branch operations are recorded as part of a Boolean-OR filter with other masked operation type filter controls.
0b1	PMSFCR_EL1.B controls whether branch operations are recorded as part of a Boolean-AND filter with other unmasked operation type filter controls.

This field is ignored by the PE when PMSFCR_EL1.FT is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPE_EFT is not implemented:

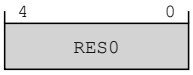

Bits [4:0]

Reserved, RES0.

Bits [47:21]

Reserved, RES0.

TYPE, bits [20:16]

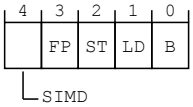

SIMD, bit [4]

When FEAT_SPE_EFT is implemented:

SIMD filter enable.

SIMD	Meaning
0b0	If PMSFCR_EL1.SIMDm is 1, then record only operations that are not SIMD operations. Otherwise, do not record SIMD operations, unless allowed by another operation type filter.
0b1	If PMSFCR_EL1.SIMDm is 1, then record only operations that are SIMD operations. Otherwise, record all SIMD operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is ignored by the PE and no records are removed by this filter when any of the following apply:

- PMSFCR_EL1.FT is 0.
- PMSFCR_EL1.SIMDm is 0 and the values of the PMSFCR_EL1.{SIMD, FP, ST, LD, B} bits for which the corresponding PMSFCR_EL1.{SIMDm, FPm, STm, LDm, Bm} bit is zero are all zero.

For filtering purposes, SIMD operations means all Advanced SIMD, SVE, and SME SIMD operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FP, bit [3]

When FEAT_SPE_EFT is implemented:

Floating-point filter enable.

FP	Meaning
0b0	If PMSFCR_EL1.FPm is 1, then record only operations that are not floating-point operations. Otherwise, do not record floating-point operations, unless allowed by another operation type filter.
0b1	If PMSFCR_EL1.FPm is 1, then record only operations that are floating-point operations. Otherwise, record all floating-point operations.

This field is ignored by the PE and no records are removed by this filter when any of the following apply:

- PMSFCR_EL1.FT is 0.
- PMSFCR_EL1.FPm is 0 and the values of the PMSFCR_EL1.{SIMD, FP, ST, LD, B} bits for which the corresponding PMSFCR_EL1.{SIMDm, FPm, STm, LDm, Bm} bit is zero are all zero.

For filtering purposes, floating-point operations means all scalar, Advanced SIMD, SVE, and SME floating-point operations, as defined by the FP_SPEC event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ST, bit [2]

Store filter enable.

ST	Meaning
0b0	If FEAT_SPE_EFT is implemented and PMSFCR_EL1.STm is 1, then record only operations that are not store operations. Otherwise, do not record store operations, unless allowed by another operation type filter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ST	Meaning
0b1	If FEAT_SPE_EFT is implemented and PMSFCR_EL1.STm is 1, then record only operations that are store operations. Otherwise, record all store operations.

This field is ignored by the PE and no records are removed by this filter when any of the following apply:

- PMSFCR_EL1.FT is 0.
- FEAT_SPE_EFT is implemented, PMSFCR_EL1.STm is 0, and the values of the PMSFCR_EL1.{SIMD, FP, ST, LD, B} bits for which the corresponding PMSFCR_EL1.{SIMDm, FPm, STm, LDm, Bm} bit is zero are all zero.

For filtering purposes, store operations includes vector stores and all atomic operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LD, bit [1]

Load filter enable.

LD	Meaning
0b0	If FEAT_SPE_EFT is implemented and PMSFCR_EL1.LDm is 1, then record only operations that are not load operations. Otherwise, do not record load operations, unless allowed by another operation type filter.
0b1	If FEAT_SPE_EFT is implemented and PMSFCR_EL1.LDm is 1, then record only operations that are load operations. Otherwise, record all load operations.

This field is ignored by the PE and no records are removed by this filter when any of the following apply:

- PMSFCR_EL1.FT is 0.
- FEAT_SPE_EFT is implemented, PMSFCR_EL1.LDm is 0, and the values of the PMSFCR_EL1.{SIMD, FP, ST, LD, B} bits for which the corresponding PMSFCR_EL1.{SIMDm, FPm, STm, LDm, Bm} bit is zero are all zero.

For filtering purposes, load operations includes vector loads and atomic operations that return a value to the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

B, bit [0]

Branch filter enable.

B	Meaning
0b0	If FEAT_SPE_EFT is implemented and PMSFCR_EL1.Bm is 1, then record only operations that are not branch operations. Otherwise, do not record branch operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B	Meaning
0b1	If FEAT_SPE_EFT is implemented and PMSFCR_EL1.Bm is 1, then record only operations that are branch operations. Otherwise, record all branch operations.

This field is ignored by the PE and no records are removed by this filter when any of the following apply:

- PMSFCR_EL1.FT is 0.
- FEAT_SPE_EFT is implemented, PMSFCR_EL1.Bm is 0, and the values of the PMSFCR_EL1.{SIMD, FP, ST, LD, B} bits for which the corresponding PMSFCR_EL1.{SIMDm, FPm, STm, LDm, Bm} bit is zero are all zero.

For filtering purposes, branch operations includes exception returns.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:5]

Reserved, RES0.

FDS, bit [4]

When FEAT_SPE_FDS is implemented:

Filter by Data Source.

FDS	Meaning
0b0	Data Source filtering disabled.
0b1	Data Source filtering enabled. Samples of load instructions reporting a Data Source not selected by PMSDSFR_EL1 will not be recorded.

If PMSFCR_EL1.FDS is 1 and PMSDSFR_EL1 is zero, then no load operations with a Data Source will be recorded.

Load operations without a Data Source and other sampled operations are unaffected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnE, bit [3]

When FEAT_SPE_FnE is implemented:

Filter by event, inverted.

FnE	Meaning
0b0	Inverted event filtering disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FnE	Meaning
0b1	Inverted event filtering enabled. Samples including the events selected by PMSNEVFR_EL1 will not be recorded.

If any of the following are true, then for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded or the PE behaves as if PMSFCR_EL1.FnE is 0:

- PMSFCR_EL1.FnE is 1 and PMSNEVFR_EL1 is zero.
- PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, and there exists a value x such that PMSEVFR_EL1.E[x] is 1 and PMSNEVFR_EL1.E[x] is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FL, bit [2]

Filter by latency.

FL	Meaning
0b0	Latency filtering disabled.
0b1	Latency filtering enabled. Samples with a total latency less than PMSLATFR_EL1.MINLAT will not be recorded.

If PMSFCR_EL1.FL is 1 and PMSLATFR_EL1.MINLAT is zero, then for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded or the PE behaves as if PMSFCR_EL1.FL is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FT, bit [1]

Filter by type. The filter is controlled by the TYPE and, if FEAT_SPE_EFT is implemented, TYPEm fields.

FT	Meaning
0b0	Type filtering disabled.
0b1	Type filtering enabled. Samples not one of the selected types will not be recorded.

Type filtering filters according to the type of the sampled operation.

If FEAT_SPE_EFT is not implemented, PMSFCR_EL1.FT is 1, and the PMSFCR_EL1.{ST, LD, B} bits are all zero, then for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded or the PE behaves as if PMSFCR_EL1.FT is 0.

For more information, see Filtering by operation type.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FE, bit [0]

Filter by event.

FE	Meaning
0b0	Event filtering disabled.
0b1	Event filtering enabled. Samples not including the events selected by PMSEVFR_EL1 will not be recorded.

If any of the following are true, then for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded or the PE behaves as if PMSFCR_EL1.FE is 0:

- PMSFCR_EL1.FE is 1 and PMSEVFR_EL1 is zero.
- FEAT_SPE_FnE is implemented, PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, and there exists a value x such that PMSEVFR_EL1.E[x] is 1 and PMSNEVFR_EL1.E[x] is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSFCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b100

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSFCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSFCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSFCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSFCR_EL1();
end;
```

```
MSR PMSFCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b100

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSFCR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSFCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.13 PMSICR_EL1, Sampling Interval Counter Register

The PMSICR_EL1 characteristics are:

Purpose

Software must write zero to PMSICR_EL1 before enabling sample profiling for a sampling session. Software must then treat PMSICR_EL1 as an opaque, 64-bit, read/write register used for context switches only.

Configuration

The value of PMSICR_EL1 does not change whilst profiling is disabled.

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSICR_EL1 are UNDEFINED.

Attributes

PMSICR_EL1 is a 64-bit register.

Field descriptions

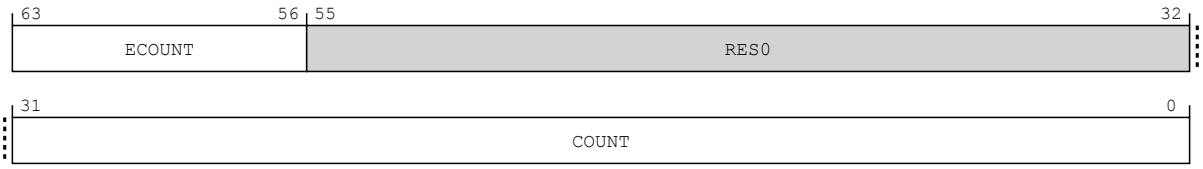

ECOUNT, bits [63:56]

When FEAT_SPE_ERnd is implemented:

Secondary sample interval counter. Provides the secondary counter used after the primary counter reaches zero.

While the secondary counter is nonzero and profiling is enabled, the secondary counter decrements by 1 for each member of the sample population. The primary counter also continues to decrement since it is also nonzero. When the secondary counter reaches zero, a member of the sampling population is selected for sampling.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

COUNT, bits [31:0]

Primary sample interval counter. Provides the primary counter used for sampling.

When the PE moves from a state or Exception level where profiling is disabled to a state or Exception level where profiling is enabled, if the value of this register is zero, then the primary counter is loaded from PMSIRR_EL1.

While the primary counter is nonzero and sampling is enabled, the primary counter decrements by 1 for each member of the sample population.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The sample interval counter counts either a number of operations or a number of instructions, depending on the IMPLEMENTATION DEFINED value of PMSIDR_EL1.ArchInst.

When the counter reaches zero, the behavior depends on the value of PMSIRR_EL1.RND and whether FEAT_SPE_ERnd is implemented:

- If PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is implemented, then the secondary counter is set to a random or pseudorandom value in the range `0x00` to `0xFF`.
- Otherwise, a member of the sampling population is selected for sampling.
- The primary counter is loaded from PMSIRR_EL1.

The primary counter is loaded from PMSIRR_EL1 means:

- PMSICR_EL1.COUNT[31:8] is set to the value of PMSIRR_EL1.INTERVAL.
- If PMSIRR_EL1.RND is 1 and FEAT_SPE_ERnd is not implemented, then PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range `0x00` to `0xFF`.
- Otherwise, PMSICR_EL1.COUNT[7:0] is set to `0x00`.

For more information, see Initializing the sample interval counters and Behavior of the sample interval counter while profiling is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSICR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSICR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b010

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSICR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x838);
    else
        X{64}(t) = PMSICR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSICR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSICR_EL1();
end;
```

```
MSR PMSICR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b010

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSICR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x838) = X{64}(t);
    else
        PMSICR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSICR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSICR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.14 PMSIDR_EL1, Sampling Profiling ID Register

The PMSIDR_EL1 characteristics are:

Purpose

Describes the Statistical Profiling implementation to software.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSIDR_EL1 are UNDEFINED.

Attributes

PMSIDR_EL1 is a 64-bit register.

Field descriptions

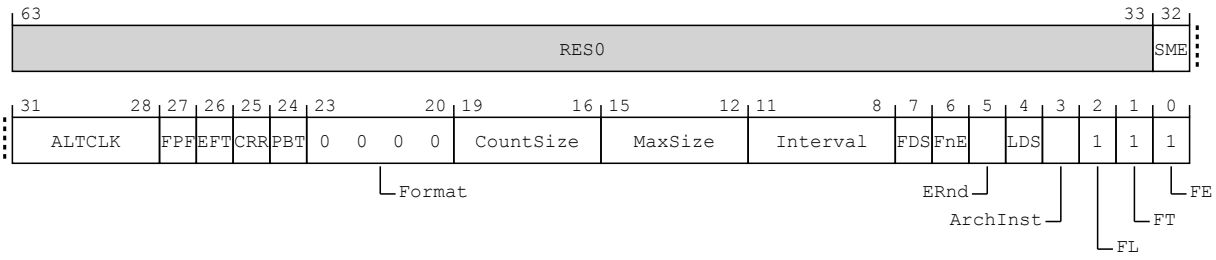

Bits [63:33]

Reserved, RES0.

SME, bit [32]

SPE for SME. Describes support for the Scalable Matrix Extensions to Statistical Profiling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
0b0	The SME extensions to the Statistical Profiling Extension are not implemented.
0b1	Adds support for Statistical Profiling of the Scalable Matrix Extensions.

FEAT_SPE_SME implements the functionality identified by the value 1.

Access to this field is RO.

ALTCLK, bits [31:28]

Alternate clock domain.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ALTCLK	Meaning
0b0000	Alternate clock domain not implemented, or CPU clock domain.
0b0001	The external Streaming Mode Compute Unit provides the alternate clock domain.
0b1111	IMPLEMENTATION DEFINED clock domain.

All other values are reserved.

FEAT_SPE_ALTCLK implements the functionality identified by a nonzero value.

Access to this field is RO.

FPF, bit [27]

Floating-point flag. Describes whether Operation Type packets for sampled scalar and SIMD operations contain floating-point and Advanced SIMD indications.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPF	Meaning
0b0	Operation Type packets for scalar and Advanced SIMD operations do not contain floating-point or Advanced SIMD information.
0b1	Operation Type packets for scalar and Advanced SIMD operations contain floating-point or Advanced SIMD information.

FEAT_SPE_FPF implements the functionality identified by the value 1.

Access to this field is RO.

EFT, bit [26]

Extended filtering by type. Describes whether Extended filtering by operation type is supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EFT	Meaning
0b0	PMSFCR_EL1.{SIMDm, FPm, STm, Bm, LDm, SIMD, FP} are RES0.
0b1	PMSFCR_EL1.{SIMDm, FPm, STm, Bm, LDm, SIMD, FP} are implemented.

FEAT_SPE_EFT implements the functionality identified by the value 1.

Access to this field is RO.

CRR, bit [25]

Call Return branch records. Describes whether the Operation Type packets for sampled branch operations contain Call Return information.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRR	Meaning
0b0	Operation Type packets for branch operations do not contain Call Return information.
0b1	Operation Type packets for branch operations contain Call Return information.

FEAT_SPE_CRR implements the functionality identified by the value 1.

Access to this field is RO.

PBT, bit [24]

Previous branch target Address packet. Describes whether the Previous branch target Address packet is supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PBT	Meaning
0b0	Previous branch target Address packet not implemented.
0b1	Previous branch target Address packet implemented.

FEAT_SPE_PBT implements the OPTIONAL functionality identified by the value 1.

Access to this field is RO.

Format, bits [23:20]

Defines the format of the sample records.

Format	Meaning
0b0000	Format 0.

All other values are reserved.

Access to this field is RO.

CountSize, bits [19:16]

Defines the size of the counters.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CountSize	Meaning
0b0010	12-bit saturating counters.
0b0011	16-bit saturating counters.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MaxSize, bits [15:12]

Defines the largest size for a single record, rounded up to a power-of-two. If this is the same as the minimum alignment (PMBIDR_EL1.Align), then each record is exactly this size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MaxSize	Meaning
`0b0100`	16 bytes.
`0b0101`	32 bytes.
`0b0110`	64 bytes.
`0b0111`	128 bytes.
`0b1000`	256 bytes.
`0b1001`	512 bytes.
`0b1010`	1KB.
`0b1011`	2KB.

All other values are reserved.

The values `0b0100` and `0b0101` are not permitted for an implementation.

Access to this field is RO.

Interval, bits [11:8]

Minimum sampling interval. This provides guidance from the implementer to the smallest sampling interval.

The interval is defined as a number of either operations or instructions, depending on the IMPLEMENTATION DEFINED value of PMSIDR_EL1.ArchInst.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Interval	Meaning
`0b0000`	256 operations or instructions, or no minimum specified.
`0b0010`	512 operations or instructions.
`0b0011`	768 operations or instructions.
`0b0100`	1,024 operations or instructions.
`0b0101`	1,536 operations or instructions.
`0b0110`	2,048 operations or instructions.
`0b0111`	3,072 operations or instructions.
`0b1000`	4,096 operations or instructions.

All other values are reserved.

If this field reads as a nonzero value, then setting the sample interval to a value less than the indicated value is likely to cause a statistically significant number of sample collisions. For example, the smallest interval might indicate the typical size of the speculation window for the implementation. That is, the number of operations or instructions that will be executing in parallel, when the processor implementation is not heavily loaded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Setting the sample interval to a value greater than or equal to the indicated value does not guarantee a statistically insignificant number of sample collisions, as this is typically dependent on the program being profiled. For example, if the program has a large number of long latency loads due to cache misses, then this might cause more collisions when these operations or instructions are sampled.

For more information, see 'Sample collisions'.

Access to this field is RO.

FDS, bit [7]

When FEAT_SPEv1p4 is implemented:

Filtering by data source.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FDS	Meaning
0b0	PMSDSFR_EL1 is not implemented and PMSFCR_EL1.FDS is RES0.
0b1	PMSDSFR_EL1 and PMSFCR_EL1.FDS are implemented.

FEAT_SPE_FDS implements the functionality identified by the value 1.

From Armv8.9, if FEAT_SPE_LDS is implemented, the value 0 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

FnE, bit [6]

When FEAT_SPE_FnE is implemented:

Filtering by events, inverted.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FnE	Meaning
0b0	PMSNEVFR_EL1 is not implemented and PMSFCR_EL1.FnE is RES0.
0b1	PMSNEVFR_EL1 and PMSFCR_EL1.FnE are implemented.

FEAT_SPE_FnE implements the functionality identified by the value 1.

From Armv8.7, if FEAT_SPE is implemented, the value 0 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

ERnd, bit [5]

Defines how the random number generator is used in determining the interval between samples, when enabled by PMSIRR_EL1.RND.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

ERnd	Meaning
0b0	The random number is added at the start of the interval, and the sample is taken and a new interval started when the combined interval expires.
0b1	The random number is added and the new interval started after the interval programmed in PMSIRR_EL1.INTERVAL expires, and the sample is taken when the random interval expires.

FEAT_SPE_ERnd implements the functionality identified by the value 1.

Access to this field is RO.

LDS, bit [4]

Data source indicator for sampled load instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LDS	Meaning
0b0	Loaded data source not implemented.
0b1	Loaded data source implemented.

FEAT_SPE_LDS implements the functionality identified by the value 1.

Access to this field is RO.

ArchInst, bit [3]

Architectural instruction profiling. Defines the subject of the profiling population.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ArchInst	Meaning
0b0	Micro-op sampling implemented.
0b1	Architecture instruction sampling implemented.

FEAT_SPE_ArchInst implements the functionality identified by the value 1.

Access to this field is RO.

FL, bit [2]

Filtering by latency.

Reads as 0b1

Access to this field is RO.

FT, bit [1]

Filtering by operation type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reads as `0b1`

Access to this field is RO.

FE, bit [0]

Filtering by events.

Reads as `0b1`

Access to this field is RO.

Accessing PMSIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b111

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.15 PMSIRR_EL1, Sampling Interval Reload Register

The PMSIRR_EL1 characteristics are:

Purpose

Defines the interval between samples.

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSIRR_EL1 are UNDEFINED.

Attributes

PMSIRR_EL1 is a 64-bit register.

Field descriptions

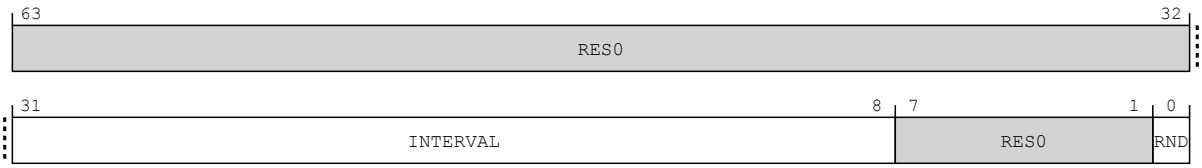

Bits [63:32]

Reserved, RES0.

INTERVAL, bits [31:8]

Bits [31:8] of the PMSICR_EL1 interval counter reload value. Software must set this to a nonzero value.

If this field is zero, then an UNKNOWN sampling interval is used.

If PMSIDR_EL1.Interval is nonzero, then it provides guidance from the implementer to the smallest sampling interval that should be used.

Setting this field to a nonzero value smaller than the indicated value is likely to cause a statistically significant number of sample collisions. See Sample collisions. Setting the sample interval to a value greater than or equal to the indicated value does not guarantee a statistically insignificant number of sample collisions, as this is typically dependent on the program being profiled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:1]

Reserved, RES0.

RND, bit [0]

Controls randomization of the sampling interval.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RND	Meaning
0b0	Disable randomization of sampling interval.
0b1	Add (pseudo-)random jitter to sampling interval.

The random number generator is not architected.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSIRR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSIRR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSIRR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x840);
    else
        X{64}(t) = PMSIRR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSIRR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSIRR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR PMSIRR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b011

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSIRR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x840) = X{64}(t);
    else
        PMSIRR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSIRR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSIRR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.16 PMSLATFR_EL1, Sampling Latency Filter Register

The PMSLATFR_EL1 characteristics are:

Purpose

Controls sample filtering by latency

Configuration

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSLATFR_EL1 are UNDEFINED.

Attributes

PMSLATFR_EL1 is a 64-bit register.

Field descriptions

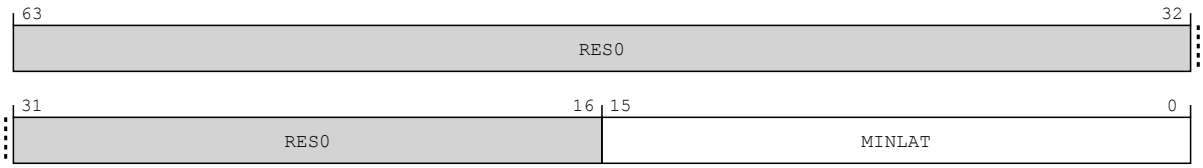

Bits [63:16]

Reserved, RES0.

MINLAT, bits [15:0]

Minimum latency.

When PMSFCR_EL1.FL is 1, defines the value used for filtering by latency. Samples with a Total latency less than PMSLATFR_EL1.MINLAT are not recorded.

If PMSIDR_EL1.CountSize is `0b0010`, PMSLATFR_EL1.MINLAT[15:12] is RES0.

This field is ignored by the PE when PMSFCR_EL1.FL is 0.

For more information, see Filtering by latency.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSLATFR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSLATFR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().PMSLATFR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x848);
    else
        X{64}(t) = PMSLATFR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSLATFR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSLATFR_EL1();
end;
```

```
MSR PMSLATFR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b110

```
if !IsFeatureImplemented(FEAT_SPE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().PMSLATFR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x848) = X{64}(t);
    else
        PMSLATFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSLATFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSLATFR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.7.17 PMSNEVFR_EL1, Sampling Inverted Event Filter Register

The PMSNEVFR_EL1 characteristics are:

Purpose

Controls sample filtering by events. The overall inverted filter is the logical OR of these filters. For example, if PMSNEVFR_EL1.E[3] and PMSNEVFR_EL1.E[5] are both set to 1, samples that have either event 3 (Level 1 unified or data cache refill) or event 5 (TLB walk) set to 1 are not recorded.

Configuration

This register is present only when FEAT_SPE_FnE is implemented. Otherwise, direct accesses to PMSNEVFR_EL1 are UNDEFINED.

Attributes

PMSNEVFR_EL1 is a 64-bit register.

Field descriptions

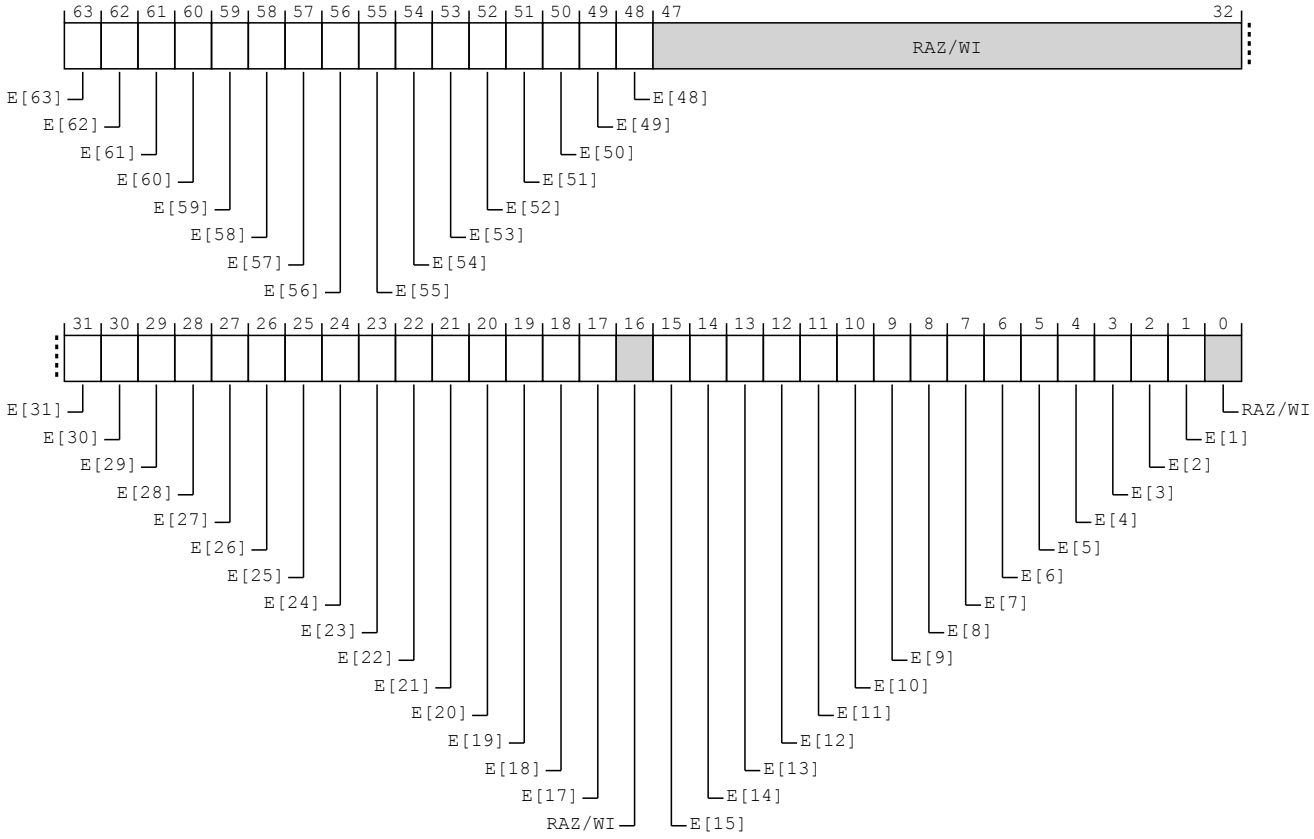

E[63], bit [63]

When event 63 is implemented and filtering on event 63 is supported:

Filter on IMPLEMENTATION DEFINED event 63.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[63]	Meaning
0b0	Event 63 is ignored.
0b1	Do not record samples that have event 63 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[63] is 1, and PMSNEVFR_EL1.E[63] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[62], bit [62]

When event 62 is implemented and filtering on event 62 is supported:

Filter on IMPLEMENTATION DEFINED event 62.

E[62]	Meaning
0b0	Event 62 is ignored.
0b1	Do not record samples that have event 62 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[62] is 1, and PMSNEVFR_EL1.E[62] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[61], bit [61]

When event 61 is implemented and filtering on event 61 is supported:

Filter on IMPLEMENTATION DEFINED event 61.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[61]	Meaning
0b0	Event 61 is ignored.
0b1	Do not record samples that have event 61 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[61] is 1, and PMSNEVFR_EL1.E[61] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[60], bit [60]

When event 60 is implemented and filtering on event 60 is supported:

Filter on IMPLEMENTATION DEFINED event 60.

E[60]	Meaning
0b0	Event 60 is ignored.
0b1	Do not record samples that have event 60 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[60] is 1, and PMSNEVFR_EL1.E[60] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[59], bit [59]

When event 59 is implemented and filtering on event 59 is supported:

Filter on IMPLEMENTATION DEFINED event 59.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[59]	Meaning
0b0	Event 59 is ignored.
0b1	Do not record samples that have event 59 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[59] is 1, and PMSNEVFR_EL1.E[59] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[58], bit [58]

When event 58 is implemented and filtering on event 58 is supported:

Filter on IMPLEMENTATION DEFINED event 58.

E[58]	Meaning
0b0	Event 58 is ignored.
0b1	Do not record samples that have event 58 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[58] is 1, and PMSNEVFR_EL1.E[58] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[57], bit [57]

When event 57 is implemented and filtering on event 57 is supported:

Filter on IMPLEMENTATION DEFINED event 57.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[57]	Meaning
0b0	Event 57 is ignored.
0b1	Do not record samples that have event 57 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[57] is 1, and PMSNEVFR_EL1.E[57] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[56], bit [56]

When event 56 is implemented and filtering on event 56 is supported:

Filter on IMPLEMENTATION DEFINED event 56.

E[56]	Meaning
0b0	Event 56 is ignored.
0b1	Do not record samples that have event 56 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[56] is 1, and PMSNEVFR_EL1.E[56] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[55], bit [55]

When event 55 is implemented and filtering on event 55 is supported:

Filter on IMPLEMENTATION DEFINED event 55.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[55]	Meaning
0b0	Event 55 is ignored.
0b1	Do not record samples that have event 55 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[55] is 1, and PMSNEVFR_EL1.E[55] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[54], bit [54]

When event 54 is implemented and filtering on event 54 is supported:

Filter on IMPLEMENTATION DEFINED event 54.

E[54]	Meaning
0b0	Event 54 is ignored.
0b1	Do not record samples that have event 54 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[54] is 1, and PMSNEVFR_EL1.E[54] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[53], bit [53]

When event 53 is implemented and filtering on event 53 is supported:

Filter on IMPLEMENTATION DEFINED event 53.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[53]	Meaning
0b0	Event 53 is ignored.
0b1	Do not record samples that have event 53 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[53] is 1, and PMSNEVFR_EL1.E[53] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[52], bit [52]

When event 52 is implemented and filtering on event 52 is supported:

Filter on IMPLEMENTATION DEFINED event 52.

E[52]	Meaning
0b0	Event 52 is ignored.
0b1	Do not record samples that have event 52 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[52] is 1, and PMSNEVFR_EL1.E[52] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[51], bit [51]

When event 51 is implemented and filtering on event 51 is supported:

Filter on IMPLEMENTATION DEFINED event 51.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[51]	Meaning
0b0	Event 51 is ignored.
0b1	Do not record samples that have event 51 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[51] is 1, and PMSNEVFR_EL1.E[51] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[50], bit [50]

When event 50 is implemented and filtering on event 50 is supported:

Filter on IMPLEMENTATION DEFINED event 50.

E[50]	Meaning
0b0	Event 50 is ignored.
0b1	Do not record samples that have event 50 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[50] is 1, and PMSNEVFR_EL1.E[50] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[49], bit [49]

When event 49 is implemented and filtering on event 49 is supported:

Filter on IMPLEMENTATION DEFINED event 49.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[49]	Meaning
0b0	Event 49 is ignored.
0b1	Do not record samples that have event 49 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[49] is 1, and PMSNEVFR_EL1.E[49] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[48], bit [48]

When event 48 is implemented and filtering on event 48 is supported:

Filter on IMPLEMENTATION DEFINED event 48.

E[48]	Meaning
0b0	Event 48 is ignored.
0b1	Do not record samples that have event 48 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[48] is 1, and PMSNEVFR_EL1.E[48] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bits [47:32]

Reserved, RAZ/WI.

E[31], bit [31]

When FEAT_SPEv1p4 is not implemented, event 31 is implemented, and filtering on event 31 is supported:

Filter on IMPLEMENTATION DEFINED event 31.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[31]	Meaning
0b0	Event 31 is ignored.
0b1	Do not record samples that have event 31 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[31] is 1, and PMSNEVFR_EL1.E[31] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[30], bit [30]

When FEAT_SPEv1p4 is not implemented, event 30 is implemented, and filtering on event 30 is supported:

Filter on IMPLEMENTATION DEFINED event 30.

E[30]	Meaning
0b0	Event 30 is ignored.
0b1	Do not record samples that have event 30 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[30] is 1, and PMSNEVFR_EL1.E[30] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[29], bit [29]

When FEAT_SPEv1p4 is not implemented, event 29 is implemented, and filtering on event 29 is supported:

Filter on IMPLEMENTATION DEFINED event 29.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[29]	Meaning
0b0	Event 29 is ignored.
0b1	Do not record samples that have event 29 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[29] is 1, and PMSNEVFR_EL1.E[29] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[28], bit [28]

When FEAT_SPEv1p4 is not implemented, event 28 is implemented, and filtering on event 28 is supported:

Filter on IMPLEMENTATION DEFINED event 28.

E[28]	Meaning
0b0	Event 28 is ignored.
0b1	Do not record samples that have event 28 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[28] is 1, and PMSNEVFR_EL1.E[28] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[27], bit [27]

When FEAT_SPEv1p4 is not implemented, event 27 is implemented, and filtering on event 27 is supported:

Filter on IMPLEMENTATION DEFINED event 27.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[27]	Meaning
0b0	Event 27 is ignored.
0b1	Do not record samples that have event 27 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[27] is 1, and PMSNEVFR_EL1.E[27] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[26], bit [26]

When FEAT_SPEv1p4 is not implemented, event 26 is implemented, and filtering on event 26 is supported:

Filter on IMPLEMENTATION DEFINED event 26.

E[26]	Meaning
0b0	Event 26 is ignored.
0b1	Do not record samples that have event 26 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[26] is 1, and PMSNEVFR_EL1.E[26] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[25], bit [25]

When (FEAT_SPE_SME is implemented or FEAT_SPEv1p5 is implemented) and event 25 is implemented:

Filter on Not SMCU or other shared resource operation event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[25]	Meaning
0b0	SMCU or other shared resource operation event is ignored.
0b1	Do not record samples that have the SMCU or other shared resource operation event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[25] is 1, and PMSNEVFR_EL1.E[25] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPEv1p4 is not implemented, event 25 is implemented, and filtering on event 25 is supported:

Filter on IMPLEMENTATION DEFINED event 25.

E[25]	Meaning
0b0	Event 25 is ignored.
0b1	Do not record samples that have event 25 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[25] is 1, and PMSNEVFR_EL1.E[25] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[24], bit [24]

When FEAT_SPE_SME is implemented:

Filter on Non-streaming SVE mode event.

E[24]	Meaning
0b0	Streaming SVE mode event is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[24]	Meaning
0b1	Do not record samples that have the Streaming SVE mode event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[24] is 1, and PMSNEVFR_EL1.E[24] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPEv1p4 is not implemented, event 24 is implemented, and filtering on event 24 is supported:

Filter on IMPLEMENTATION DEFINED event 24.

E[24]	Meaning
0b0	Event 24 is ignored.
0b1	Do not record samples that have event 24 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[24] is 1, and PMSNEVFR_EL1.E[24] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[23], bit [23]

When FEAT_SPEv1p4 is implemented and event 23 is implemented:

Filter on Data not snooped event.

E[23]	Meaning
0b0	Data snooped event is ignored.
0b1	Do not record samples that have the Data snooped event == 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[23] is 1, and PMSNEVFR_EL1.E[23] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[22], bit [22]

When FEAT_SPEv1p4 is implemented and event 22 is implemented:

Filter on Not recently fetched event.

E[22]	Meaning
`0b0`	Recently fetched event is ignored.
`0b1`	Do not record samples that have the Recently fetched event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[22] is 1, and PMSNEVFR_EL1.E[22] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[21], bit [21]

When FEAT_SPEv1p4 is implemented and event 21 is implemented:

Filter on Cache data not modified event.

E[21]	Meaning
`0b0`	Cache data modified event is ignored.
`0b1`	Do not record samples that have the Cache data modified event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[21] is 1, and PMSNEVFR_EL1.E[21] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[20], bit [20]

When FEAT_SPEv1p4 is implemented and event 20 is implemented:

Filter on Level 2 data cache hit event.

E[20]	Meaning
0b0	Level 2 data cache miss event is ignored.
0b1	Do not record samples that have the Level 2 data cache miss event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[20] is 1, and PMSNEVFR_EL1.E[20] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[19], bit [19]

When FEAT_SPEv1p4 is implemented and event 19 is implemented:

Filter on No level 2 data cache access event.

E[19]	Meaning
0b0	Level 2 data cache access event is ignored.
0b1	Do not record samples that have the Level 2 data cache access event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[19] is 1, and PMSNEVFR_EL1.E[19] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[18], bit [18]

When FEAT_SPEv1p1 is implemented and (FEAT_SVE is implemented or FEAT_SME is implemented):

Filter on Not empty predicate event.

E[18]	Meaning
0b0	Empty predicate event is ignored.
0b1	Do not record samples that have the Empty predicate event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[18] is 1, and PMSNEVFR_EL1.E[18] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[17], bit [17]

When FEAT_SPEv1p1 is implemented and (FEAT_SVE is implemented or FEAT_SME is implemented):

Filter on Not partial predicate event.

E[17]	Meaning
0b0	Partial or empty predicate event is ignored.
0b1	Do not record samples that have the Partial or empty predicate event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[17] is 1, and PMSNEVFR_EL1.E[17] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [16]

Reserved, RAZ/WI.

E[15], bit [15]

When event 15 is implemented and filtering on event 15 is supported:

Filter on IMPLEMENTATION DEFINED event 15.

E[15]	Meaning
`0b0`	Event 15 is ignored.
`0b1`	Do not record samples that have event 15 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[15] is 1, and PMSNEVFR_EL1.E[15] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[14], bit [14]

When event 14 is implemented and filtering on event 14 is supported:

Filter on IMPLEMENTATION DEFINED event 14.

E[14]	Meaning
`0b0`	Event 14 is ignored.
`0b1`	Do not record samples that have event 14 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[14] is 1, and PMSNEVFR_EL1.E[14] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[13], bit [13]

When event 13 is implemented and filtering on event 13 is supported:

Filter on IMPLEMENTATION DEFINED event 13.

E[13]	Meaning
0b0	Event 13 is ignored.
0b1	Do not record samples that have event 13 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[13] is 1, and PMSNEVFR_EL1.E[13] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[12], bit [12]

When event 12 is implemented and filtering on event 12 is supported:

Filter on IMPLEMENTATION DEFINED event 12.

E[12]	Meaning
0b0	Event 12 is ignored.
0b1	Do not record samples that have event 12 == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[12] is 1, and PMSNEVFR_EL1.E[12] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[11], bit [11]

When FEAT_SPEv1p1 is implemented:

Filter on Aligned event.

E[11]	Meaning
0b0	Misalignment event is ignored.
0b1	Do not record samples that have the Misalignment event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[11] is 1, and PMSNEVFR_EL1.E[11] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[10], bit [10]

When (FEAT_SPEv1p4 is implemented or filtering on event 10 is optionally supported) and event 10 is implemented:

Filter on No remote access event.

E[10]	Meaning
0b0	Remote access event is ignored.
0b1	Do not record samples that have the Remote access event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[10] is 1, and PMSNEVFR_EL1.E[10] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[9], bit [9]

When (FEAT_SPEv1p4 is implemented or filtering on event 9 is optionally supported) and event 9 is implemented:

Filter on Last Level cache hit event.

E[9]	Meaning
0b0	Last Level cache miss event is ignored.
0b1	Do not record samples that have the Last Level cache miss event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[9] is 1, and PMSNEVFR_EL1.E[9] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[8], bit [8]

When (FEAT_SPEv1p4 is implemented or filtering on event 8 is optionally supported) and event 8 is implemented:

Filter on No Last Level cache access event.

E[8]	Meaning
0b0	Last Level cache access event is ignored.
0b1	Do not record samples that have the Last Level cache access event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[8] is 1, and PMSNEVFR_EL1.E[8] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[7], bit [7]

Filter on Correctly predicted event.

E[7]	Meaning
0b0	Mispredicted event is ignored.
0b1	Do not record samples that have the Mispredicted event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[7] is 1, and PMSNEVFR_EL1.E[7] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[6], bit [6]

When FEAT_SPE_FnE is implemented:

Filter on Taken event.

E[6]	Meaning
0b0	Not taken event is ignored.
0b1	Do not record samples that have the Not taken event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[6] is 1, and PMSNEVFR_EL1.E[6] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[5], bit [5]

Filter on TLB hit event.

E[5]	Meaning
0b0	TLB walk event is ignored.
0b1	Do not record samples that have the TLB walk event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[5] is 1, and PMSNEVFR_EL1.E[5] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[4], bit [4]

When FEAT_SPEv1p4 is implemented or filtering on event 4 is optionally supported:

Filter on No TLB access event.

E[4]	Meaning
0b0	TLB access event is ignored.
0b1	Do not record samples that have the TLB access event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[4] is 1, and PMSNEVFR_EL1.E[4] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[3], bit [3]

Filter on Level 1 data cache hit event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[3]	Meaning
0b0	Level 1 data cache refill or miss event is ignored.
0b1	Do not record samples that have the Level 1 data cache refill or miss event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[3] is 1, and PMSNEVFR_EL1.E[3] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[2], bit [2]

When FEAT_SPEv1p4 is implemented or filtering on event 2 is optionally supported:

Filter on No Level 1 data cache access event.

E[2]	Meaning
0b0	Level 1 data cache access event is ignored.
0b1	Do not record samples that have the Level 1 data cache access event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[2] is 1, and PMSNEVFR_EL1.E[2] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

E[1], bit [1]

When the PE supports sampling of speculative instructions:

Filter on Speculative event.

E[1]	Meaning
0b0	Architecturally retired event is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E[1]	Meaning
0b1	Do not record samples that have the Architecturally retired event == 1.

If PMSFCR_EL1.FnE is 1, PMSFCR_EL1.FE is 1, PMSEVFR_EL1.E[1] is 1, and PMSNEVFR_EL1.E[1] is 1, then, for each sampled operation, it is CONSTRAINED UNPREDICTABLE whether the sample is discarded, and it is CONSTRAINED UNPREDICTABLE whether the SAMPLE_FEED_EVENT PMU event is generated.

This field is ignored by the PE when PMSFCR_EL1.FnE is 0.

If the PE does not support the sampling of speculative instructions, or always discards the sample record for speculative instructions, this bit reads as an UNKNOWN value and the PE ignores its value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [0]

Reserved, RAZ/WI.

Accessing PMSNEVFR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMSNEVFR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b001

```
if !IsFeatureImplemented(FEAT_SPE_FnE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSN == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
     ↪HDFGRTR_EL2().nPMSNEVFR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x850);
    else
        X{64}(t) = PMSNEVFR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSN == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = PMSNEVFR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = PMSNEVFR_EL1();
end;
```

```
MSR PMSNEVFR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1001	0b001

```
if !IsFeatureImplemented(FEAT_SPE_FnE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSN == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nPMSNEVFR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2().TPMS == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x850) = X{64}(t);
    else
        PMSNEVFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && CheckMDCR_EL3_NSPBTrap() then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().EnPMSN == '0' then
        Undefined();
    elsif HaveEL(EL3) && CheckMDCR_EL3_NSPBTrap() then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().EnPMSN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        PMSNEVFR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSNEVFR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8 Branch Record Buffer Extension registers

This section lists the Branch Record Buffer registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.1 BRBCR_EL1, Branch Record Buffer Control Register (EL1)

The BRBCR_EL1 characteristics are:

Purpose

Controls the Branch Record Buffer.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBCR_EL1 are UNDEFINED.

Attributes

BRBCR_EL1 is a 64-bit register.

Field descriptions

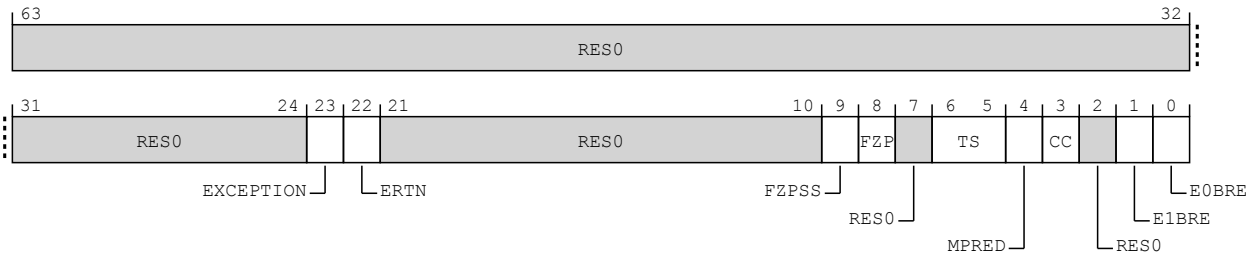

Bits [63:24]

Reserved, RES0.

EXCEPTION, bit [23]

Enable the recording of entry to EL1 via an exception.

EXCEPTION	Meaning
0b0	Disable the recording of Branch records for exceptions when taken to EL1.
0b1	Enable the recording of Branch records for exceptions when taken to EL1.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

ERTN, bit [22]

Allow the recording Branch records for exception return instructions from EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ERTN	Meaning
0b0	Disable the recording Branch records for exception return instructions from EL1.
0b1	Enable the recording Branch records for exception return instructions from EL1.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bits [21:10]

Reserved, RES0.

FZPSS, bit [9]

When FEAT_PMUv3_SS is implemented:

Freeze BRBE on PMU Snapshot.

FZPSS	Meaning
0b0	Branch recording is not affected by this control.
0b1	If either EL2 is not implemented or BRBCR_EL2.FZPSS is 1, then a BRBE freeze event occurs when a successful Capture event occurs.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FZP, bit [8]

When FEAT_PMUv3 is implemented:

Freeze BRBE on PMU overflow.

FZP	Meaning
0b0	Branch recording is not affected by this control.
0b1	A BRBE freeze event occurs when the PE is in a Non-prohibited region, BRBFCR_EL1.PAUSED is 0, and any of the following applies: • For any event counter PMEVCNTR<m>_EL0 in the first range, PMOVSCLR_EL0[m] is 1. • FEAT_PMUv3_ICNTR is implemented and PMOVSCLR_EL0.F0 is 1. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control.

TS	Meaning	Applies when
0b01	Virtual timestamp. The BRBE recorded timestamp is the physical counter value, minus the value of CNTVOFF_EL2.	
0b10	Guest physical timestamp. The BRBE recorded timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • EL3 is implemented and SCR_EL3.ECVEn == 0. • EL2 is implemented and CNTHCTL_EL2.ECV == 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented
0b11	Physical timestamp. The BRBE recorded timestamp is the physical counter value.	

All other values are reserved.

This field is ignored by the PE when EL2 is implemented and BRBCR_EL2.TS != 0b00.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

MPRED, bit [4]

Mask the recording of mispredicts.

MPRED	Meaning
0b0	Disable the recording of mispredict information.
0b1	Allow the recording of mispredict information.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

CC, bit [3]

Enable the recording of cycle count information.

CC	Meaning
0b0	Disable the recording of cycle count information.
0b1	Allow the recording of cycle count information.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E1BRE, bit [1]

EL1 Branch recording enable.

E1BRE	Meaning
0b0	Branch recording prohibited at EL1.
0b1	Branch recording enabled at EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

E0BRE, bit [0]

EL0 Branch recording enable.

E0BRE	Meaning
0b0	Branch recording prohibited at EL0.
0b1	Branch recording enabled at EL0.

This field is ignored by the PE when all of the following are true:

- The Effective value of HCR_EL2.TGE is 1.
- EL2 is implemented and enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing BRBCR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `BRBCR_EL1` or `BRBCR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, BRBCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBCTL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8E0);
    else
        X{64}(t) = BRBCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = BRBCR_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = BRBCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBCR_EL1();
end;
```

```
MSR BRBCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBCTL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8E0) = X{64}(t);
    else
        BRBCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        BRBCR_EL2() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        BRBCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBCR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, BRBCR_EL12

op0	op1	CRn	CRm	op2
0b10	0b101	0b1001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x8E0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0'
            ↪then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS ==
            ↪'1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = BRBCR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = BRBCR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR BRBCR_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b101	0b1001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x8E0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0'
            ↪then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS ==
            ↪'1' then
            Undefined();
        elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            BRBCR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        BRBCR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.2 BRBCR_EL2, Branch Record Buffer Control Register (EL2)

The BRBCR_EL2 characteristics are:

Purpose

Controls the Branch Record Buffer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBCR_EL2 are UNDEFINED.

Attributes

BRBCR_EL2 is a 64-bit register.

Field descriptions

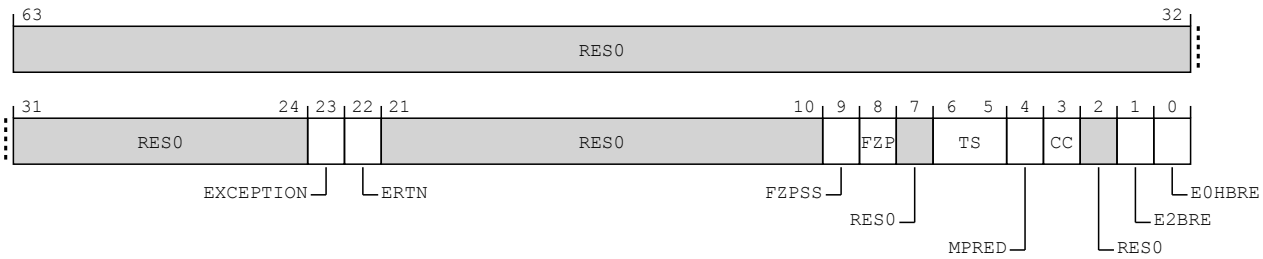

Bits [63:24]

Reserved, RES0.

EXCEPTION, bit [23]

Enable the recording of entry to EL2 via an exception.

EXCEPTION	Meaning
0b0	Disable the recording of Branch records for exceptions when taken to EL2.
0b1	Enable the recording of Branch records for exceptions when taken to EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

ERTN, bit [22]

Allow the recording Branch records for exception return instructions from EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ERTN	Meaning
0b0	Disable the recording Branch records for exception return instructions from EL2.
0b1	Enable the recording Branch records for exception return instructions from EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bits [21:10]

Reserved, RES0.

FZPSS, bit [9]

When FEAT_PMUv3_SS is implemented:

Freeze BRBE on PMU Snapshot.

FZPSS	Meaning
0b0	Branch recording is not affected by this control.
0b1	If BRBCR_EL1.FZPSS is 1, then a BRBE freeze event occurs when a PMU snapshot occurs.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FZP, bit [8]

When FEAT_PMUv3 is implemented:

Freeze BRBE on PMU overflow.

FZP	Meaning
0b0	Branch recording is not affected by this control.
0b1	A BRBE freeze event occurs when the PE is in a Non-prohibited region, BRBFCR_EL1.PAUSED is 0, and, for any event counter PMEVCNTR<m>_EL0 in the second range, PMOVSCLR_EL0[m] is 1. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control.

TS	Meaning	Applies when
0b00	Timestamp controlled by BRBCR_EL1.TS.	
0b01	Virtual timestamp. The BRBE recorded timestamp is the physical counter value, minus the value of CNTVOFF_EL2.	
0b10	Guest physical timestamp. The BRBE recorded timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • EL3 is implemented and SCR_EL3.ECVEn == 0. • EL2 is implemented and CNTHCTL_EL2.ECV == 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented
0b11	Physical timestamp. The BRBE recorded timestamp is the physical counter value.	

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

MPRED, bit [4]

Mask the recording of mispredicts.

MPRED	Meaning
0b0	Disable the recording of mispredict information.
0b1	Allow the recording of mispredict information.

If EL2 is not implemented, then the Effective value of this field is 1, other than for a direct read of the register.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

CC, bit [3]

Enable the recording of cycle count information.

CC	Meaning
0b0	Disable the recording of cycle count information.
0b1	Allow the recording of cycle count information.

If EL2 is not implemented, then the Effective value of this field is 1, other than for a direct read of the register.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E2BRE, bit [1]

EL2 Branch recording enable.

E2BRE	Meaning
0b0	Branch recording prohibited at EL2.
0b1	Branch recording enabled at EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

E0HBRE, bit [0]

EL0 Branch recording enable.

E0HBRE	Meaning
0b0	Branch recording prohibited at EL0 when the Effective value of HCR_EL2.TGE is 1.
0b1	Branch recording enabled at EL0 when the Effective value of HCR_EL2.TGE is 1.

This field is ignored by the PE when any of the following are true:

- The Effective value of HCR_EL2.TGE is 0.
- EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing BRBCR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `BRBCR_EL2` or `BRBCR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, BRBCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBCTL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8E0);
    else
        X{64}(t) = BRBCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = BRBCR_EL2();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = BRBCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBCR_EL1();
end;
```

MRS <Xt>, BRBCR_EL2

op0	op1	CRn	CRm	op2
0b10	0b100	0b1001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBCR_EL2();
end;
```

MSR BRBCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBCTL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8E0) = X{64}(t);
    else
        BRBCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        BRBCR_EL2() = X{64}(t);
    else
        BRBCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBCR_EL1() = X{64}(t);
end;
```

```
MSR BRBCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b100	0b1001	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBCR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.3 BRBFCR_EL1, Branch Record Buffer Function Control Register

The BRBFCR_EL1 characteristics are:

Purpose

Functional controls for the Branch Record Buffer.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBFCR_EL1 are UNDEFINED.

Attributes

BRBFCR_EL1 is a 64-bit register.

Field descriptions

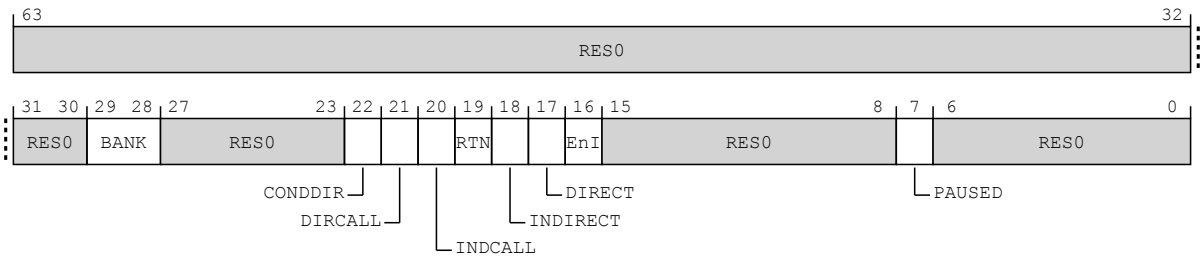

Bits [63:30]

Reserved, RES0.

BANK, bits [29:28]

Branch record buffer bank access control.

BANK	Meaning
0b00	Select branch records 0 to 31.
0b01	Select branch records 32 to 63.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:23]

Reserved, RES0.

CONDDIR, bit [22]

Match on conditional direct branch instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONDDIR	Meaning
0b0	Do not match on conditional direct branch instructions.
0b1	Match on conditional direct branch instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

DIRCALL, bit [21]

Match on direct branch with link instructions.

DIRCALL	Meaning
0b0	Do not match on direct branch with link instructions.
0b1	Match on direct branch with link instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

INDCALL, bit [20]

Match on indirect branch with link instructions.

INDCALL	Meaning
0b0	Do not match on indirect branch with link instructions.
0b1	Match on indirect branch with link instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

RTN, bit [19]

Match on function return instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RTN	Meaning
0b0	Do not match on function return instructions.
0b1	Match on function return instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

INDIRECT, bit [18]

Match on indirect branch instructions.

INDIRECT	Meaning
0b0	Do not match on indirect branch instructions.
0b1	Match on indirect branch instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

DIRECT, bit [17]

Match on unconditional direct branch instructions.

DIRECT	Meaning
0b0	Do not match on unconditional direct branch instructions.
0b1	Match on unconditional direct branch instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

EnI, bit [16]

Include or exclude matches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnI	Meaning
0b0	Include records for matches, and exclude records for non-matches.
0b1	Exclude records for matches, and include records for non-matches.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bits [15:8]

Reserved, RES0.

PAUSED, bit [7]

Branch recording Paused status.

PAUSED	Meaning
0b0	Branch recording is not Paused.
0b1	Branch recording is Paused.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bits [6:0]

Reserved, RES0.

Accessing BRBFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, BRBFCR_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBCTL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBFCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBFCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBFCR_EL1();
end;
```

```
MSR BRBFCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBCTL == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBFCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBFCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.4 BRBIDR0_EL1, Branch Record Buffer ID0 Register

The BRBIDR0_EL1 characteristics are:

Purpose

Indicates the features of the branch buffer unit.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBIDR0_EL1 are UNDEFINED.

Attributes

BRBIDR0_EL1 is a 64-bit register.

Field descriptions

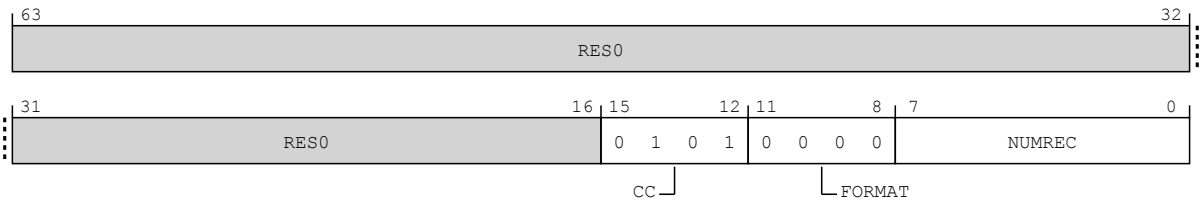

Bits [63:16]

Reserved, RES0.

CC, bits [15:12]

Cycle counter support. Defined values are:

CC	Meaning
0b0101	20-bit cycle counter implemented.

All other values are reserved.

Access to this field is RO.

FORMAT, bits [11:8]

Data format of records of the Branch record buffer. Defined values are:

FORMAT	Meaning
0b0000	Format 0.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMREC, bits [7:0]

Number of records supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMREC	Meaning
0x08	8 branch records implemented.
0x10	16 branch records implemented.
0x20	32 branch records implemented.
0x40	64 branch records implemented.

All other values are reserved.

Access to this field is RO.

Accessing BRBIDR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, BRBIDR0_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBIDR == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBIDR0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBIDR0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBIDR0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.5 BRBINFINJ_EL1, Branch Record Buffer Information Injection Register

The BRBINFINJ_EL1 characteristics are:

Purpose

The information of a Branch record for injection.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBINFINJ_EL1 are UNDEFINED.

Attributes

BRBINFINJ_EL1 is a 64-bit register.

Field descriptions

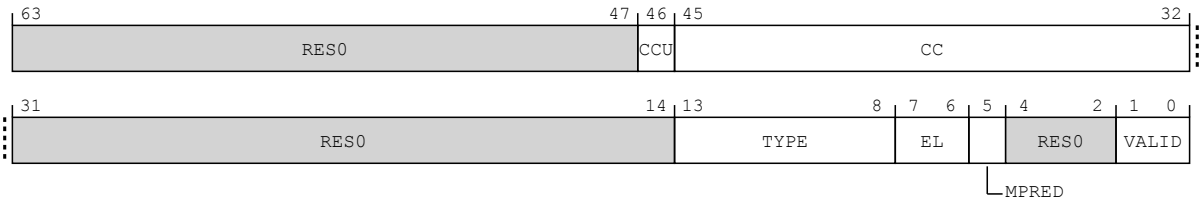

Bits [63:47]

Reserved, RES0.

CCU, bit [46]

The number of PE clock cycles since the last Branch record entry is UNKNOWN.

CCU	Meaning
0b0	Indicates that the number of PE clock cycles since the last Branch record is indicated by BRBINFINJ_EL1.CC.
0b1	Indicates that the number of PE clock cycles since the last Branch record is UNKNOWN.

The value in this field is only valid when BRBINFINJ_EL1.VALID != 0b00.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When BRBINFINJ_EL1.VALID == ‘00’, access to this field is RES0.
- Otherwise, access to this field is RW.

CC, bits [45:32]

The number of PE clock cycles since the last Branch record entry.

The format of this field uses a mantissa and exponent to express the cycle count value, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CC bits[7:0] indicate the mantissa M.
- CC bits[13:8] indicate the exponent E.

The cycle count is expressed using the following function:

```
if IsZero(E), then UInt(M) else UInt('1':M:Zeros(UInt(E)-1))
```

If required, the cycle count is rounded to a multiple of $2^{(E-1)}$ towards zero before being encoded.

A value of all ones in both the mantissa and exponent indicates the cycle count value exceeded the size of the cycle counter.

The value in this field is only valid when BRBINFINJ_EL1.VALID != `0b00`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINFINJ_EL1.CCU == '1'
  - BRBINFINJ_EL1.VALID == '00'
- Otherwise, access to this field is RW.

Bits [31:14]

Reserved, RES0.

TYPE, bits [13:8]

Branch type.

TYPE	Meaning
`0b000000`	Unconditional direct branch, excluding Branch with link.
`0b000001`	Indirect branch, excluding Branch with link, Return from subroutine, and Exception return.
`0b000010`	Direct Branch with link.
`0b000011`	Indirect Branch with link.
`0b000101`	Return from subroutine.
`0b000111`	Exception return.
`0b001000`	Conditional direct branch.
`0b100001`	Debug halt.
`0b100010`	Call.
`0b100011`	Trap.
`0b100100`	SError.
`0b100110`	Instruction debug.
`0b100111`	Data debug.
`0b101010`	Alignment.
`0b101011`	Inst Fault.
`0b101100`	Data Fault.
`0b101110`	IRQ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Meaning
0b101111	FIQ.
0b110000	IMPLEMENTATION DEFINED exception to EL3.
0b111001	Debug State Exit.

All other values are reserved.

The value in this field is only valid when BRBINFINJ_EL1.VALID != 0b00.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When BRBINFINJ_EL1.VALID == '00', access to this field is RES0.
- Otherwise, access to this field is RW.

EL, bits [7:6]

The Exception level at the target address.

EL	Meaning	Applies when
0b00	EL0.	
0b01	EL1.	
0b10	EL2.	
0b11	EL3.	FEAT_BRBEv1p1 is implemented

The value in this field is only valid when BRBINFINJ_EL1.VALID == 0b11 or BRBINFINJ_EL1.VALID == 0b01.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == '00'
  - BRBINFINJ_EL1.VALID == '10'
- Otherwise, access to this field is RW.

MPRED, bit [5]

Branch mispredict.

MPRED	Meaning
0b0	Branch was correctly predicted or the result of the prediction was not captured.
0b1	Branch was incorrectly predicted.

The value in this field is only valid when BRBINFINJ_EL1.VALID == 0b11 or BRBINFINJ_EL1.VALID == 0b10.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == '00'
  - BRBINFINJ_EL1.VALID == '01'
  - BRBINFINJ_EL1.TYPE[5] == '1'
- Otherwise, access to this field is RW.

Bits [4:2]

Reserved, RES0.

VALID, bits [1:0]

The Branch record is valid.

VALID	Meaning
0b00	This Branch record is not valid. The values of following fields are not valid: • BRBTGTINJ_EL1.ADDRESS. • BRBSRCINJ_EL1.ADDRESS. • BRBINFINJ_EL1.MPRED. • BRBINFINJ_EL1.LASTFAILED. • BRBINFINJ_EL1.T. • BRBINFINJ_EL1.EL. • BRBINFINJ_EL1.TYPE. • BRBINFINJ_EL1.CC. • BRBINFINJ_EL1.CCU.
0b01	This Branch record is valid. The values of following fields are not valid: • BRBSRCINJ_EL1.ADDRESS. • BRBINFINJ_EL1.T. • BRBINFINJ_EL1.MPRED.
0b10	This Branch record is valid. The values of following fields are not valid: • BRBTGTINJ_EL1.ADDRESS. • BRBINFINJ_EL1.EL.
0b11	This Branch record is valid.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing BRBINFINJ_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, BRBINFINJ_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBINFINJ_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBINFINJ_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBINFINJ_EL1();
end;
```

```
MSR BRBINFINJ_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBINFINJ_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBINFINJ_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBINFINJ_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.6 BRBINF<n>_EL1, Branch Record Buffer Information Register <n>, n = 0 - 31

The BRBINF<n>_EL1 characteristics are:

Purpose

The information for Branch record n + (BRBFCR_EL1.BANK × 32).

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBINF<n>_EL1 are UNDEFINED.

Attributes

BRBINF<n>_EL1 is a 64-bit register.

Field descriptions

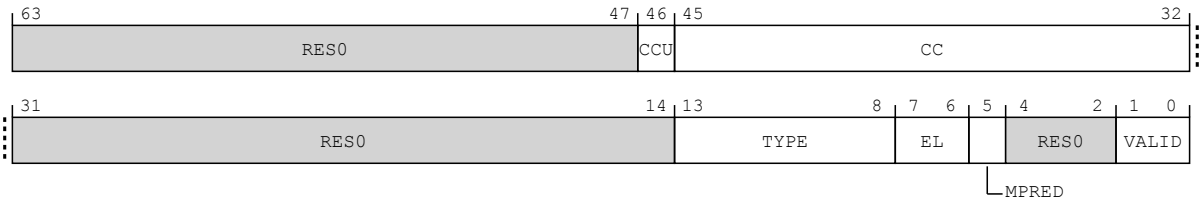

Bits [63:47]

Reserved, RES0.

CCU, bit [46]

The number of PE clock cycles since the last Branch record entry is UNKNOWN.

CCU	Meaning
0b0	Indicates that the number of PE clock cycles since the last Branch record is indicated by BRBINF<n>_EL1.CC.
0b1	Indicates that the number of PE clock cycles since the last Branch record is UNKNOWN.

The value in this field is only valid when BRBINF<n>_EL1.VALID != 0b00.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When BRBINF<n>_EL1.VALID == ‘00’, access to this field is RES0.
- Otherwise, access to this field is RO.

CC, bits [45:32]

The number of PE clock cycles since the last Branch record entry.

The format of this field uses a mantissa and exponent to express the cycle count value, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CC bits[7:0] indicate the mantissa M.
- CC bits[13:8] indicate the exponent E.

The cycle count is expressed using the following function:

```
if IsZero(E), then UInt(M) else UInt('1':M:Zeros(UInt(E)-1))
```

If required, the cycle count is rounded to a multiple of $2^{(E-1)}$ towards zero before being encoded.

A value of all ones in both the mantissa and exponent indicates the cycle count value exceeded the size of the cycle counter.

The value in this field is only valid when BRBINF<n>_EL1.VALID != `0b00`.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINF<n>_EL1.CCU == '1'
  - BRBINF<n>_EL1.VALID == '00'
- Otherwise, access to this field is RO.

Bits [31:14]

Reserved, RES0.

TYPE, bits [13:8]

Branch type.

TYPE	Meaning
`0b000000`	Unconditional direct branch, excluding Branch with link.
`0b000001`	Indirect branch, excluding Branch with link, Return from subroutine, and Exception return.
`0b000010`	Direct Branch with link.
`0b000011`	Indirect Branch with link.
`0b000101`	Return from subroutine.
`0b000111`	Exception return.
`0b001000`	Conditional direct branch.
`0b100001`	Debug halt.
`0b100010`	Call.
`0b100011`	Trap.
`0b100100`	SError.
`0b100110`	Instruction debug.
`0b100111`	Data debug.
`0b101010`	Alignment.
`0b101011`	Inst Fault.
`0b101100`	Data Fault.
`0b101110`	IRQ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TYPE	Meaning
0b101111	FIQ.
0b110000	IMPLEMENTATION DEFINED exception to EL3.
0b111001	Debug State Exit.

All other values are reserved.

The value in this field is only valid when BRBINF<n>_EL1.VALID != 0b00.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When BRBINF<n>_EL1.VALID == '00', access to this field is RES0.
- Otherwise, access to this field is RO.

EL, bits [7:6]

The Exception level at the target address.

EL	Meaning	Applies when
0b00	EL0.	
0b01	EL1.	
0b10	EL2.	
0b11	EL3.	FEAT_BRBEv1p1 is implemented

The value in this field is only valid when BRBINF<n>_EL1.VALID == 0b11 or BRBINF<n>_EL1.VALID == 0b01.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINF<n>_EL1.VALID == '00'
  - BRBINF<n>_EL1.VALID == '10'
- Otherwise, access to this field is RO.

MPRED, bit [5]

Branch mispredict.

MPRED	Meaning
0b0	Branch was correctly predicted or the result of the prediction was not captured.
0b1	Branch was incorrectly predicted.

The value in this field is only valid when BRBINF<n>_EL1.VALID == 0b11 or BRBINF<n>_EL1.VALID == 0b10.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINF<n>_EL1.VALID == ‘00’
  - BRBINF<n>_EL1.VALID == ‘01’
  - BRBINF<n>_EL1.TYPE[5] == ‘1’
- Otherwise, access to this field is RO.

Bits [4:2]

Reserved, RES0.

VALID, bits [1:0]

The Branch record is valid.

VALID	Meaning
0b00	This Branch record is not valid. The values of following fields are not valid: • BRBTGT<n>_EL1.ADDRESS. • BRBSRC<n>_EL1.ADDRESS. • BRBINF<n>_EL1.MPRED. • BRBINF<n>_EL1.LASTFAILED. • BRBINF<n>_EL1.T. • BRBINF<n>_EL1.EL. • BRBINF<n>_EL1.TYPE. • BRBINF<n>_EL1.CC. • BRBINF<n>_EL1.CCU.
0b01	This Branch record is valid. The values of following fields are not valid: • BRBSRC<n>_EL1.ADDRESS. • BRBINF<n>_EL1.T. • BRBINF<n>_EL1.MPRED.
0b10	This Branch record is valid. The values of following fields are not valid: • BRBTGT<n>_EL1.ADDRESS. • BRBINF<n>_EL1.EL.
0b11	This Branch record is valid.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing BRBINF<n>_EL1

BRBINF<n>_EL1 reads-as-zero if n + (BRBFCR_EL1.BANK × 32) >= BRBIDR0_EL1.NUMREC.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, BRBINF<m>_EL1 ; Where m = 0-31
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1000	m[3:0]	m[4]:0b00

```
let m:integer = UInt(op2[2] :: CRm[3:0]);

if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = BRBINF_EL1(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = BRBINF_EL1(m);
    end;
elsif PSTATE.EL == EL3 then
    if m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = BRBINF_EL1(m);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.7 BRBSRCINJ_EL1, Branch Record Buffer Source Address Injection Register

The BRBSRCINJ_EL1 characteristics are:

Purpose

The source address of a Branch record for injection.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBSRCINJ_EL1 are UNDEFINED.

Attributes

BRBSRCINJ_EL1 is a 64-bit register.

Field descriptions

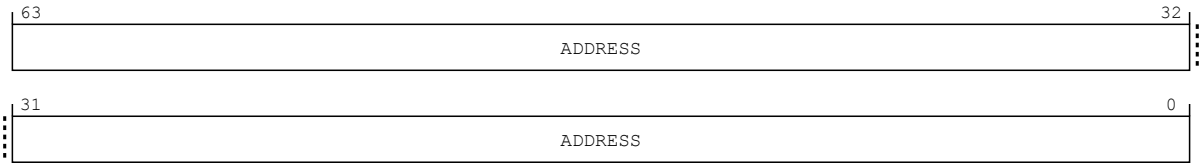

ADDRESS, bits [63:0]

Source virtual address of the Branch record.

When a direct write occurs with a value with ADDRESS bits [63:P] indicating an invalid address, an UNKNOWN value which indicates an invalid address is written to bits [63:P].

An invalid address is:

- For an address in a translation regime with 2 VA ranges, with bits [63:P] not all zeroes or all ones.
- For an address in a translation regime with a single VA range, with bits [63:P] not all zeroes.

P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The value in bits [P-1:0] is the value written.

When a direct write occurs with a value with ADDRESS bits [63:P] indicating a valid address, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == ‘00’
  - BRBINFINJ_EL1.VALID == ‘01’
- Otherwise, access to this field is RW.

Accessing BRBSRCINJ_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, BRBSRCINJ_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBSRCINJ_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBSRCINJ_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBSRCINJ_EL1();
end;
```

```
MSR BRBSRCINJ_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBSRCINJ_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBSRCINJ_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBSRCINJ_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.8 BRBSRC<n>_EL1, Branch Record Buffer Source Address Register <n>, n = 0 - 31

The BRBSRC<n>_EL1 characteristics are:

Purpose

The source address of Branch record n + (BRBFCR_EL1.BANK × 32).

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBSRC<n>_EL1 are UNDEFINED.

Attributes

BRBSRC<n>_EL1 is a 64-bit register.

Field descriptions

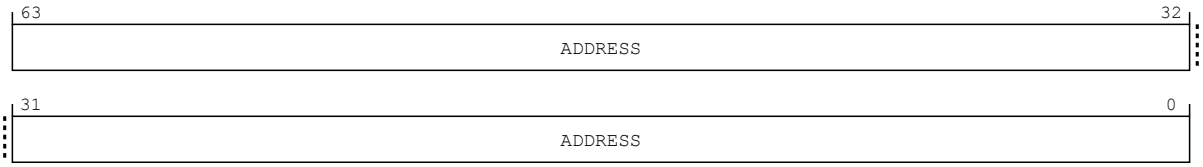

ADDRESS, bits [63:0]

Source virtual address of the Branch record.

When an indirect write occurs with a value with ADDRESS bits [63:P] indicating an invalid address, an UNKNOWN value which indicates an invalid address is written to bits [63:P].

An invalid address is:

- For an address in a translation regime with 2 VA ranges, with bits [63:P] not all zeroes or all ones.
- For an address in a translation regime with a single VA range, with bits [63:P] not all zeroes.

P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The value in bits [P-1:0] is the value written.

When an indirect write occurs with a value with ADDRESS bits [63:P] indicating a valid address, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINF<n>_EL1.VALID == ‘00’
  - BRBINF<n>_EL1.VALID == ‘01’
- Otherwise, access to this field is RO.

Accessing BRBSRC<n>_EL1

BRBSRC<n>_EL1 is RES0 if n + (BRBFCR_EL1.BANK × 32) >= BRBIDR0_EL1.NUMREC.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, BRBSRC<m>_EL1 ; Where m = 0-31
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1000	m[3:0]	m[4]:0b01

```
let m:integer = UInt(op2[2] :: CRm[3:0]);

if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = BRBSRC_EL1(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = BRBSRC_EL1(m);
    end;
elsif PSTATE.EL == EL3 then
    if m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = BRBSRC_EL1(m);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.9 BRBTGTINJ_EL1, Branch Record Buffer Target Address Injection Register

The BRBTGTINJ_EL1 characteristics are:

Purpose

The target address of a Branch record for injection.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBTGTINJ_EL1 are UNDEFINED.

Attributes

BRBTGTINJ_EL1 is a 64-bit register.

Field descriptions

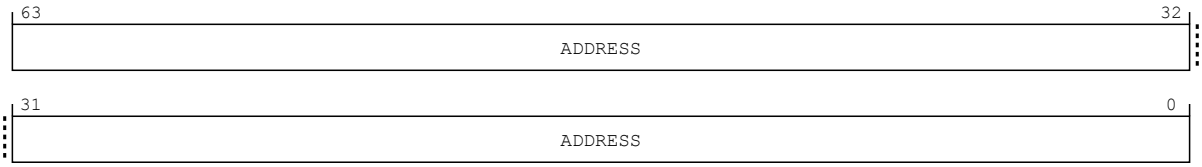

ADDRESS, bits [63:0]

Target virtual address of the Branch record.

When a direct write occurs with a value with ADDRESS bits [63:P] indicating an invalid address, an UNKNOWN value which indicates an invalid address is written to bits [63:P].

An invalid address is:

- For an address in a translation regime with 2 VA ranges, with bits [63:P] not all zeroes or all ones.
- For an address in a translation regime with a single VA range, with bits [63:P] not all zeroes.

P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The value in bits [P-1:0] is the value written.

When a direct write occurs with a value with ADDRESS bits [63:P] indicating a valid address, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == ‘00’
  - BRBINFINJ_EL1.VALID == ‘10’
- Otherwise, access to this field is RW.

Accessing BRBTGTINJ_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, BRBTGTINJ_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBTGTINJ_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBTGTINJ_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBTGTINJ_EL1();
end;
```

```
MSR BRBTGTINJ_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBTGTINJ_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBTGTINJ_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBTGTINJ_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.10 BRBTGT<n>_EL1, Branch Record Buffer Target Address Register <n>, n = 0 - 31

The BRBTGT<n>_EL1 characteristics are:

Purpose

The target address of Branch record n + (BRBFCR_EL1.BANK × 32).

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBTGT<n>_EL1 are UNDEFINED.

Attributes

BRBTGT<n>_EL1 is a 64-bit register.

Field descriptions

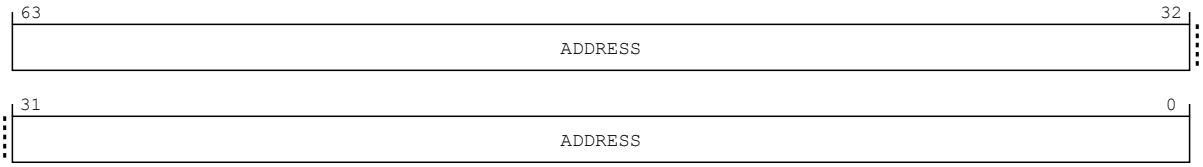

ADDRESS, bits [63:0]

Target virtual address of the Branch record.

When an indirect write occurs with a value with ADDRESS bits [63:P] indicating an invalid address, an UNKNOWN value which indicates an invalid address is written to bits [63:P].

An invalid address is:

- For an address in a translation regime with 2 VA ranges, with bits [63:P] not all zeroes or all ones.
- For an address in a translation regime with a single VA range, with bits [63:P] not all zeroes.

P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The value in bits [P-1:0] is the value written.

When an indirect write occurs with a value with ADDRESS bits [63:P] indicating a valid address, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES0 if any of the following are true:
  - BRBINF<n>_EL1.VALID == ‘00’
  - BRBINF<n>_EL1.VALID == ‘10’
- Otherwise, access to this field is RO.

Accessing BRBTGT<n>_EL1

BRBTGT<n>_EL1 is RES0 if n + (BRBFCR_EL1.BANK × 32) >= BRBIDR0_EL1.NUMREC.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, BRBTGT<m>_EL1 ; Where m = 0-31
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1000	m[3:0]	m[4]:0b10

```
let m:integer = UInt(op2[2] :: CRm[3:0]);

if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = BRBTGT_EL1(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = BRBTGT_EL1(m);
    end;
elsif PSTATE.EL == EL3 then
    if m + (UInt(BRBFCR_EL1().BANK) * 32) >= NUM_BRBE_RECORDS then
        X{64}(t) = Zeros{64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X{64}(t) = BRBTGT_EL1(m);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.8.11 BRBTS_EL1, Branch Record Buffer Timestamp Register

The BRBTS_EL1 characteristics are:

Purpose

Captures the Timestamp value on a BRBE freeze event.

Configuration

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBTS_EL1 are UNDEFINED.

Attributes

BRBTS_EL1 is a 64-bit register.

Field descriptions

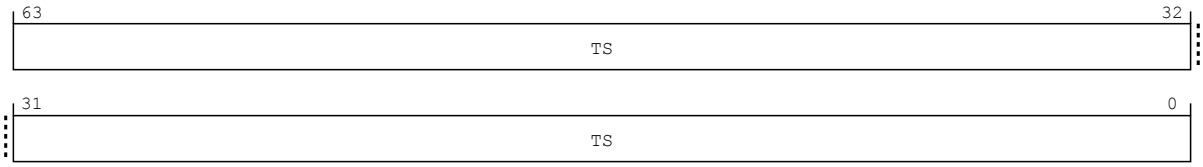

TS, bits [63:0]

Timestamp value at the time of a BRBE freeze event.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_BRBEv1p1 is not implemented, this field resets to the expression `0x0000`.

When FEAT_BRBEv1p1 is implemented, Arm recommends that this field is preserved on a Warm reset, but it is IMPLEMENTATION DEFINED whether this field resets to 0 or is preserved.

When FEAT_BRBEv1p1 is implemented, on a Cold reset it is IMPLEMENTATION DEFINED whether this field resets to an architecturally UNKNOWN value.

Accessing BRBTS_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, BRBTS_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGRTR_EL2().nBRBDATA == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBTS_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = BRBTS_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = BRBTS_EL1();
end;
```

```
MSR BRBTS_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b001	0b1001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_BRBE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HDFGWTR_EL2().nBRBDATA == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBTS_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        Undefined();
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE != '11' && SCR_EL3().NS == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MDCR_EL3().SBRBE IN {'x0'} && SCR_EL3().NS == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        BRBTS_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    BRBTS_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9 RAS registers

This section lists RAS registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.1 DISR_EL1, Deferred Interrupt Status Register

The DISR_EL1 characteristics are:

Purpose

Records that an SError exception has been consumed by an `ESB` instruction.

Configuration

AArch64 System register DISR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DISR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to DISR_EL1 are UNDEFINED.

Attributes

DISR_EL1 is a 64-bit register.

Field descriptions

When DISR_EL1.IDS == '0':

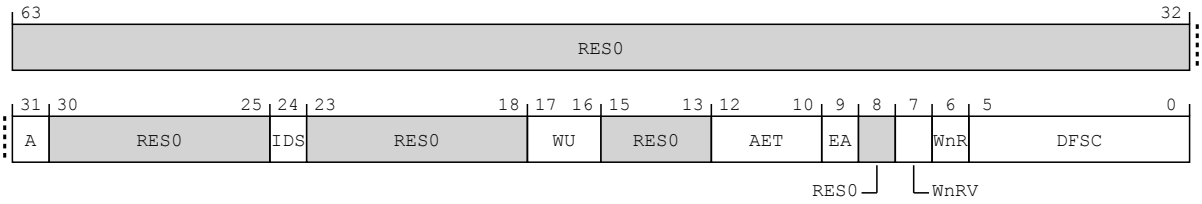

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an `ESB` instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

Indicates the deferred SError exception type.

IDS	Meaning
`0b0`	Deferred error uses architecturally-defined format.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:18]

Reserved, RES0.

WU, bits [17:16]

When FEAT_RASv2 is implemented:

Write update. See the description of ESR_ELx.WU for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [15:13]

Reserved, RES0.

AET, bits [12:10]

Asynchronous Error Type. See the description of ESR_ELx.AET for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort Type. See the description of ESR_ELx.EA for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

WnRV, bit [7]

When FEAT_RASv2 is implemented:

Write-not-Read Valid. See the description of ESR_ELx.WnRV for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WnR, bit [6]

When FEAT_RASv2 is implemented:

Write-not-Read. See the description of ESR_ELx.WnR for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC, bits [5:0]

Fault Status Code. See the description of ESR_ELx.DFSC for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When DISR_EL1.IDS == '1':

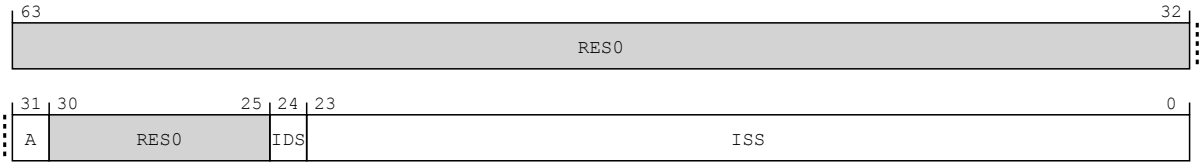

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

Indicates the deferred SError exception type.

IDS	Meaning
0b1	Deferred error uses IMPLEMENTATION DEFINED format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED syndrome. See the description of ESR_ELx[23:0] for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DISR_EL1

An indirect write to DISR_EL1 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR_EL1 occurring in program order after the ESB instruction.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, DISR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().AMO == '1' || (IsFeatureImplemented(FEAT_DoubleFault2) &&
        ↪IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) then
        X{64}(t) = VDISR_EL2();
    elsif HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        X{64}(t) = VDISR_EL3();
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = DISR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        X{64}(t) = VDISR_EL3();
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = DISR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DISR_EL1();
end;
```

```
MSR DISR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().AMO == '1' || (IsFeatureImplemented(FEAT_DoubleFault2) &&
        ↪IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) then
        VDISR_EL2() = X{64}(t);
    elsif HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        VDISR_EL3() = X{64}(t);
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        return;
    else
        DISR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        VDISR_EL3() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        return;
    else
        DISR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    DISR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.2 ERRIDR_EL1, Error Record ID Register

The ERRIDR_EL1 characteristics are:

Purpose

Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

Configuration

AArch64 System register ERRIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERRIDR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRIDR_EL1 are UNDEFINED.

Attributes

ERRIDR_EL1 is a 64-bit register.

Field descriptions

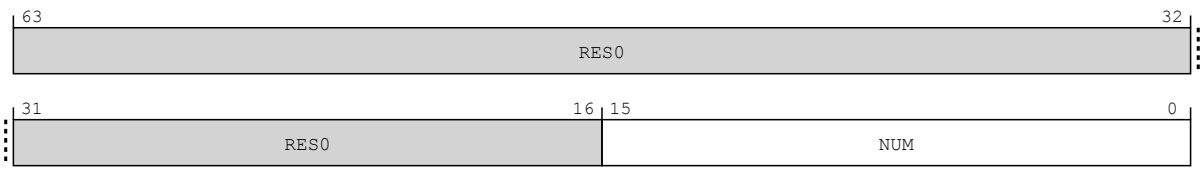

Bits [63:16]

Reserved, RES0.

NUM, bits [15:0]

Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates no records can be accessed through the Error Record System registers.

Each implemented record is owned by a node. A node might own multiple records.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing ERRIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERRIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERRIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.3 ERRSELR_EL1, Error Record Select Register

The ERRSELR_EL1 characteristics are:

Purpose

Selects an error record to be accessed through the Error Record System registers.

Configuration

If ERRIDR_EL1 indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR_EL1 is UNDEFINED or RES0.

AArch64 System register ERRSELR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERRSELR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRSELR_EL1 are UNDEFINED.

Attributes

ERRSELR_EL1 is a 64-bit register.

Field descriptions

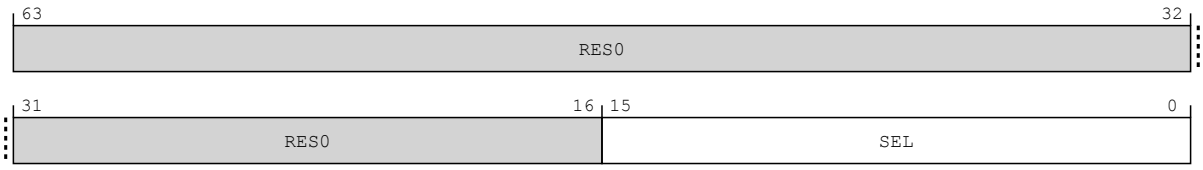

Bits [63:16]

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR_EL1.SEL is `0x0004`, then direct reads and writes of ERXSTATUS_EL1 access ERR4STATUS.

If ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then all of the following apply:

- The value read back from ERRSELR_EL1.SEL is UNKNOWN.
- Unless the access to an ERX*_EL1 register generates a higher priority exception, one of the following occurs:
  - An UNKNOWN error record is selected.
  - The ERX*_EL1 registers are RAZ/WI.
  - ERX*_EL1 register reads and writes are NOPs.
  - ERX*_EL1 register reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ERRSELR_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ERRSELR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERRSELR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRSELR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRSELR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRSELR_EL1();
end;
```

```
MSR ERRSELR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERRSELR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRSELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRSELR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRSELR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.4 ERXADDR_EL1, Selected Error Record Address Register

The ERXADDR_EL1 characteristics are:

Purpose

Accesses ERR<n>ADDR for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXADDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXADDR[31:0].

AArch64 System register ERXADDR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXADDR2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXADDR_EL1 are UNDEFINED.

Attributes

ERXADDR_EL1 is a 64-bit register.

Field descriptions

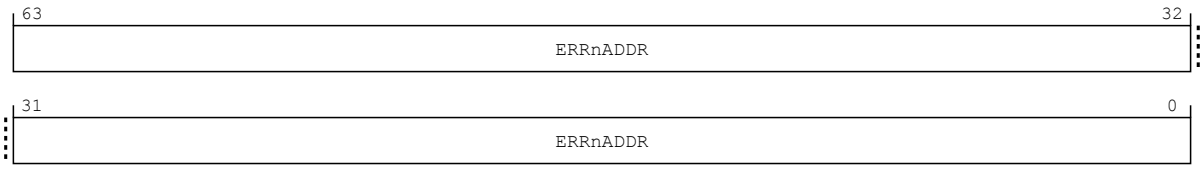

ERRnADDR, bits [63:0]

ERXADDR_EL1 accesses ERR<n>ADDR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXADDR_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR_EL1 is RAZ/WI.
- Direct reads and writes of ERXADDR_EL1 are NOPs.
- Direct reads and writes of ERXADDR_EL1 are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXADDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXADDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRADDR(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRADDR(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRADDR(UInt(ERRSELR_EL1().SEL));
end;
```

```
MSR ERXADDR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b011

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXADDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRADDR(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRADDR(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRADDR(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.5 ERXCTLR_EL1, Selected Error Record Control Register

The ERXCTLR_EL1 characteristics are:

Purpose

Accesses ERR<n>CTLR for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXCTLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXCTLR[31:0].

AArch64 System register ERXCTLR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXCTLR2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXCTLR_EL1 are UNDEFINED.

Attributes

ERXCTLR_EL1 is a 64-bit register.

Field descriptions

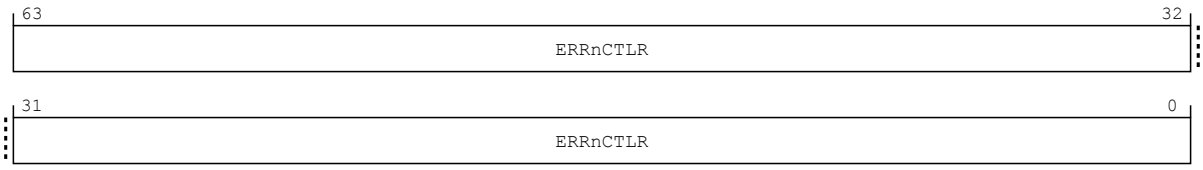

ERRnCTLR, bits [63:0]

ERXCTLR_EL1 accesses ERR<n>CTLR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXCTLR_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR_EL1 is RAZ/WI.
- Direct reads and writes of ERXCTLR_EL1 are NOPs.
- Direct reads and writes of ERXCTLR_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR is not present, meaning reads and writes of ERXCTLR_EL1 are RES0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXCTLR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRCTLR(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRCTLR(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRCTLR(UInt(ERRSELR_EL1().SEL));
end;
```

MSR ERXCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXCTLR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRCTLR(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRCTLR(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRCTLR(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.6 ERXFR_EL1, Selected Error Record Feature Register

The ERXFR_EL1 characteristics are:

Purpose

Accesses ERR<n>FR for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXFR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXFR[31:0].

AArch64 System register ERXFR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXFR2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXFR_EL1 are UNDEFINED.

Attributes

ERXFR_EL1 is a 64-bit register.

Field descriptions

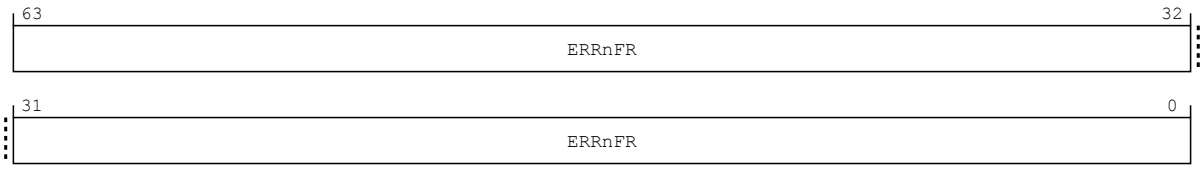

ERRnFR, bits [63:0]

ERXFR_EL1 accesses ERR<n>FR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXFR_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR_EL1 is RAZ.
- Direct reads of ERXFR_EL1 are NOPs.
- Direct reads of ERXFR_EL1 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXFR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXFR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRFR(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRFR(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRFR(UInt(ERRSELR_EL1().SEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.7 ERXGSR_EL1, Selected Error Record Group Status Register

The ERXGSR_EL1 characteristics are:

Purpose

Shows the status for the records in a group of error records.

Accesses ERRGSR for the group of error records <n> selected by ERRSELR_EL1.SEL[15:6].

Configuration

This register is present only when FEAT_RASv2 is implemented. Otherwise, direct accesses to ERXGSR_EL1 are UNDEFINED.

Attributes

ERXGSR_EL1 is a 64-bit register.

Field descriptions

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
S63	S62	S61	S60	S59	S58	S57	S56	S55	S54	S53	S52	S51	S50	S49	S48	S47	S46	S45	S44	S43	S42	S41	S40	S39	S38	S37	S36	S35	S34	S33	S32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S31	S30	S29	S28	S27	S26	S25	S24	S23	S22	S21	S20	S19	S18	S17	S16	S15	S14	S13	S12	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1	S0

S<q> , bits [q], for q = 63 to 0

The status for error record <m>, where m = q + (UInt(ERRSELR_EL1.SEL[15:6])×64). A read-only copy of ERR<m>STATUS.V.

S<q>	Meaning
0b0	No error.
0b1	One or more errors.

When !(IsErrorRecordImplemented(m) && error record m supports this type of reporting), access to this field is RES0.

Accessing ERXGSR_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN group of error records are selected.
- ERXGSR_EL1 is RAZ.
- Direct reads of ERXGSR_EL1 are NOPs.
- Direct reads of ERXGSR_EL1 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ERXGSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_RASv2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT2) && ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0')
         ↪|| HFGRTR2_EL2().nERXGSR_EL1 == '0') then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRGSR(UInt(ERRSELR_EL1().SEL[15:6]));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRGSR(UInt(ERRSELR_EL1().SEL[15:6]));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRGSR(UInt(ERRSELR_EL1().SEL[15:6]));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.8 ERXMISC0_EL1, Selected Error Record Miscellaneous Register 0

The ERXMISC0_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC0 for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXMISC0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC0[31:0].

AArch64 System register ERXMISC0_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC0_EL1 are UNDEFINED.

Attributes

ERXMISC0_EL1 is a 64-bit register.

Field descriptions

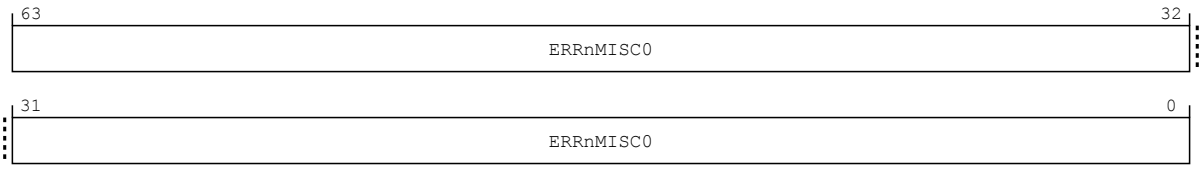

ERRnMISC0, bits [63:0]

ERXMISC0_EL1 accesses ERR<n>MISC0, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC0_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC0_EL1 are NOPs.
- Direct reads and writes of ERXMISC0_EL1 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXMISC0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL));
end;
```

MSR ERXMISC0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRMISC0(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.9 ERXMISC1_EL1, Selected Error Record Miscellaneous Register 1

The ERXMISC1_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC1 for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXMISC1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC2[31:0].

AArch64 System register ERXMISC1_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC3[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC1_EL1 are UNDEFINED.

Attributes

ERXMISC1_EL1 is a 64-bit register.

Field descriptions

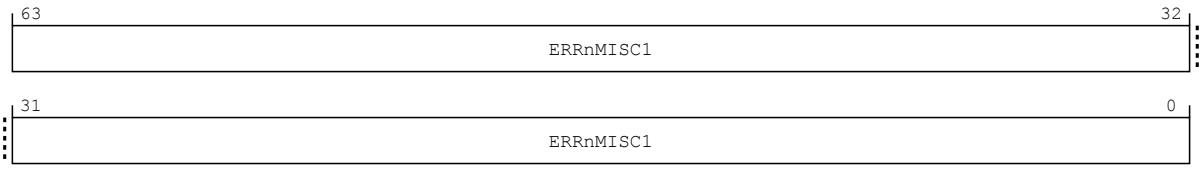

ERRnMISC1, bits [63:0]

ERXMISC1_EL1 accesses ERR<n>MISC1, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC1_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC1_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC1_EL1 are NOPs.
- Direct reads and writes of ERXMISC1_EL1 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC1_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXMISC1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC1(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC1(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRMISC1(UInt(ERRSELR_EL1().SEL));
end;
```

MSR ERXMISC1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC1(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC1(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRMISC1(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.10 ERXMISC2_EL1, Selected Error Record Miscellaneous Register 2

The ERXMISC2_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC2 for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXMISC2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC4[31:0].

AArch64 System register ERXMISC2_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC5[31:0].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC2_EL1 are UNDEFINED.

Attributes

ERXMISC2_EL1 is a 64-bit register.

Field descriptions

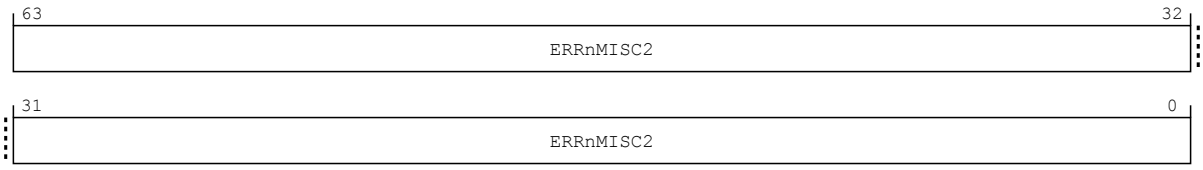

ERRnMISC2, bits [63:0]

ERXMISC2_EL1 accesses ERR<n>MISC2, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC2_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC2_EL1 are NOPs.
- Direct reads and writes of ERXMISC2_EL1 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC2_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXMISC2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC2(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC2(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRMISC2(UInt(ERRSELR_EL1().SEL));
end;
```

MSR ERXMISC2_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC2(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC2(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRMISC2(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.11 ERXMISC3_EL1, Selected Error Record Miscellaneous Register 3

The ERXMISC3_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC3 for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXMISC3_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC6[31:0].

AArch64 System register ERXMISC3_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC7[31:0].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC3_EL1 are UNDEFINED.

Attributes

ERXMISC3_EL1 is a 64-bit register.

Field descriptions

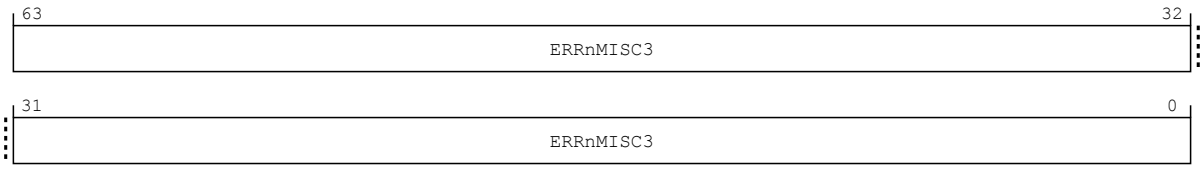

ERRnMISC3, bits [63:0]

ERXMISC3_EL1 accesses ERR<n>MISC3, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC3_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC3_EL1 are NOPs.
- Direct reads and writes of ERXMISC3_EL1 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC3_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXMISC3_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC3(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRMISC3(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRMISC3(UInt(ERRSELR_EL1().SEL));
end;
```

MSR ERXMISC3_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b011

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXMISCn_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC3(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRMISC3(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRMISC3(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.12 ERXPFGCDN_EL1, Selected Pseudo-fault Generation Countdown Register

The ERXPFGCDN_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGCDN for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGCDN_EL1 are UNDEFINED.

Attributes

ERXPFGCDN_EL1 is a 64-bit register.

Field descriptions

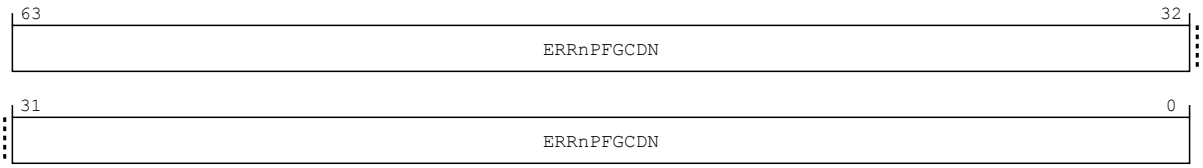

ERRnPFGCDN, bits [63:0]

ERXPFGCDN_EL1 accesses ERR<n>PFGCDN, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXPFGCDN_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<q>FR.INJ reads as `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGCDN is not present, meaning reads and writes of ERXPFGCDN_EL1 are RES0.

ERR<n>PFGCDN describes additional constraints that also apply when ERR<n>PFGCDN is accessed through ERXPFGCDN_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ERXPFGCDN_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().FIEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXPFGCDN_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRPFGCDN(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRPFGCDN(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRPFGCDN(UInt(ERRSELR_EL1().SEL));
end;
```

```
MSR ERXPFGCDN_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().FIEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXPFGCDN_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRPFGCDN(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRPFGCDN(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRPFGCDN(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.13 ERXPFGCTL_EL1, Selected Pseudo-fault Generation Control Register

The ERXPFGCTL_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGCTL for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGCTL_EL1 are UNDEFINED.

Attributes

ERXPFGCTL_EL1 is a 64-bit register.

Field descriptions

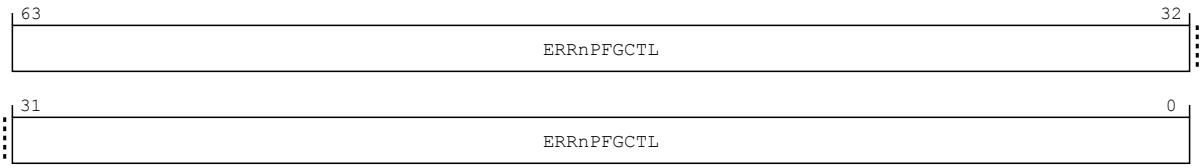

ERRnPFGCTL, bits [63:0]

ERXPFGCTL_EL1 accesses ERR<n>PFGCTL, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXPFGCTL_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<q>FR.INJ reads as `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGCTL is not present, meaning reads and writes of ERXPFGCTL_EL1 are RES0.

ERR<n>PFGCTL describes additional constraints that also apply when ERR<n>PFGCTL is accessed through ERXPFGCTL_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ERXPFGCTL_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().FIEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXPFGCTL_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRPFGCTL(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRPFGCTL(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRPFGCTL(UInt(ERRSELR_EL1().SEL));
end;
```

```
MSR ERXPFGCTL_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().FIEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXPFGCTL_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRPFGCTL(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRPFGCTL(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRPFGCTL(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.14 ERXPFGF_EL1, Selected Pseudo-fault Generation Feature Register

The ERXPFGF_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGF for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGF_EL1 are UNDEFINED.

Attributes

ERXPFGF_EL1 is a 64-bit register.

Field descriptions

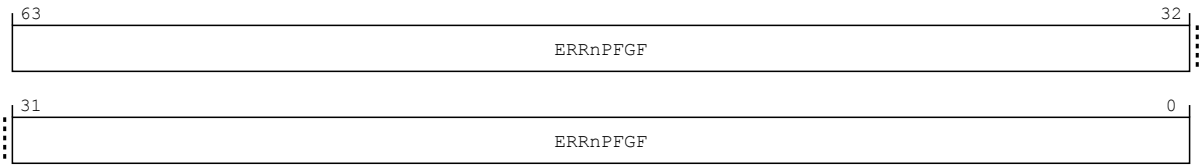

ERRnPFGF, bits [63:0]

ERXPFGF_EL1 accesses ERR<n>PFGF, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXPFGF_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<q>FR.INJ reads as `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGF is not present, meaning reads of ERXPFGF_EL1 are RES0.

ERR<n>PFGF describes additional constraints that also apply when ERR<n>PFGF is accessed through ERXPFGF_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ERXPFGF_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().FIEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXPFGF_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRPFGF(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().FIEN == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().FIEN == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRPFGF(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRPFGF(UInt(ERRSELR_EL1().SEL));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.15 ERXSTATUS_EL1, Selected Error Record Primary Status Register

The ERXSTATUS_EL1 characteristics are:

Purpose

Accesses ERR<n>STATUS for the error record <n> selected by ERRSELR_EL1.SEL.

Configuration

AArch64 System register ERXSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXSTATUS[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXSTATUS_EL1 are UNDEFINED.

Attributes

ERXSTATUS_EL1 is a 64-bit register.

Field descriptions

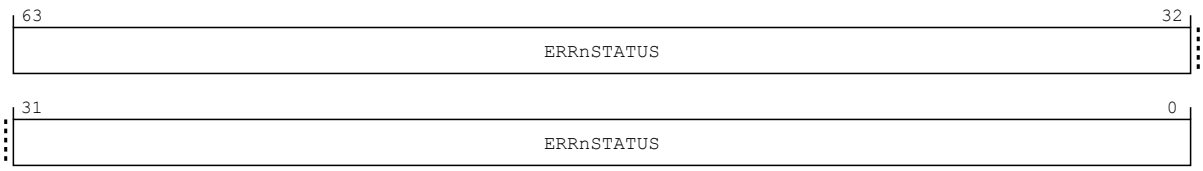

ERRnSTATUS, bits [63:0]

ERXSTATUS_EL1 accesses ERR<n>STATUS, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXSTATUS_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXSTATUS_EL1 is RAZ/WI.
- Direct reads and writes of ERXSTATUS_EL1 are NOPs.
- Direct reads and writes of ERXSTATUS_EL1 are UNDEFINED.

ERR<n>STATUS describes additional constraints that also apply when ERR<n>STATUS is accessed through ERXSTATUS_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXSTATUS_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().ERXSTATUS_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRSTATUS(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = ERRSTATUS(UInt(ERRSELR_EL1().SEL));
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = ERRSTATUS(UInt(ERRSELR_EL1().SEL));
end;
```

```
MSR ERXSTATUS_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b010

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif EL2Enabled() && HCR_EL2().TERR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().ERXSTATUS_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRSTATUS(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && SCR_EL3().TWERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        ERRSTATUS(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    ERRSTATUS(UInt(ERRSELR_EL1().SEL)) = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.16 VDISR_EL2, Virtual Deferred Interrupt Status Register (EL2)

The VDISR_EL2 characteristics are:

Purpose

Records that a virtual SError exception has been consumed by an `ESB` instruction executed at EL1 or EL0.

An indirect write to VDISR_EL2 made by an `ESB` instruction does not require an explicit synchronization operation for the value written to be observed by a direct read of one of the following registers occurring in program order after the `ESB` instruction:

- DISR_EL1.
- DISR.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VDISR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VDISR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VDISR_EL2 are UNDEFINED.

Attributes

VDISR_EL2 is a 64-bit register.

Field descriptions

When EL1 is using AArch64:

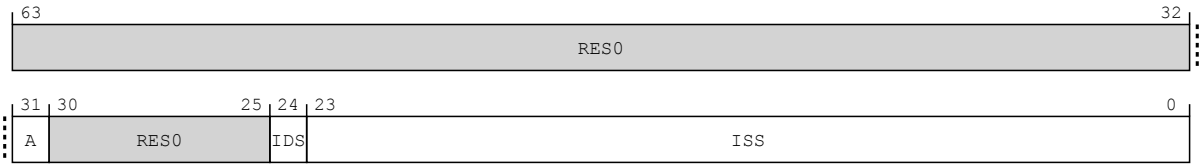

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

The value copied from VSESR_EL2.IDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS, bits [23:0]

The value copied from VSESR_EL2.ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When EL1 is using AArch32 and VDISR_EL2.LPAE == '0':

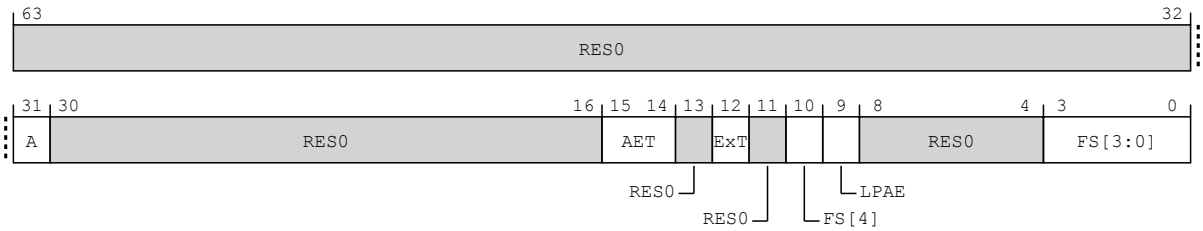

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VSESR_EL2.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VSESR_EL2.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]

Fault status code. Set to `0b10110` when an `ESB` instruction defers a virtual SError exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FS	Meaning
0b10110	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an ESB instruction defers a virtual SError exception.

LPAE	Meaning
0b0	Using the Short-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

When EL1 is using AArch32 and VDISR_EL2.LPAE == '1':

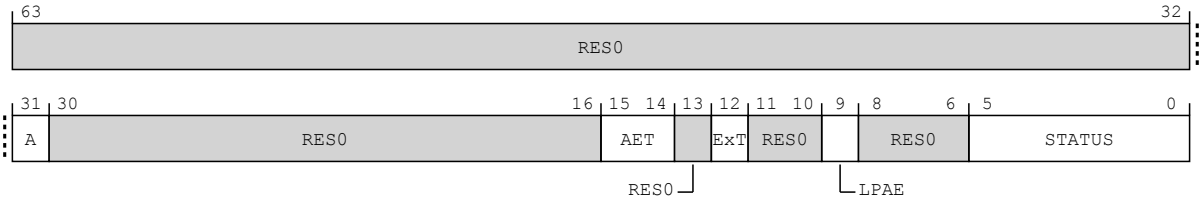

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers a virtual SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VSESR_EL2.AET.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VSESR_EL2.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an ESB instruction defers a virtual SError exception.

LPAE	Meaning
0b1	Using the Long-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status code. Set to 0b010001 when an ESB instruction defers a virtual SError exception.

STATUS	Meaning
0b010001	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VDISR_EL2

An indirect write to VDISR_EL2 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of one of the following registers occurring in program order after the ESB instruction:

- DISR_EL1.
- DISR.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, VDISR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x500);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VDISR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VDISR_EL2();
end;
```

```
MSR VDISR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x500) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VDISR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    VDISR_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, DISR_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().AMO == '1' || (IsFeatureImplemented(FEAT_DoubleFault2) &&
         ↪IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) then
        X{64}(t) = VDISR_EL2();
    elsif HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        X{64}(t) = VDISR_EL3();
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = DISR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        X{64}(t) = VDISR_EL3();
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = DISR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DISR_EL1();
end;
```

MSR DISR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().AMO == '1' || (IsFeatureImplemented(FEAT_DoubleFault2) &&
         ↪IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) then
        VDISR_EL2() = X{64}(t);
    elsif HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        VDISR_EL3() = X{64}(t);
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        return;
    else
        DISR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        VDISR_EL3() = X{64}(t);
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        return;
    else
        DISR_EL1() = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    DISR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.17 VDISR_EL3, Virtual Deferred Interrupt Status Register (EL3)

The VDISR_EL3 characteristics are:

Purpose

Records that a delegated SError exception has been consumed by an `ESB` instruction executed at EL2, EL1, or EL0 when the Effective value of SCR_EL3.DSE is 1.

An indirect write to VDISR_EL3 made by an `ESB` instruction does not require an explicit synchronization operation for the value written to be observed by a direct read of DISR_EL1 occurring in program order after the `ESB` instruction.

Configuration

Note

The encoding for this register is subject to change.

This register is present only when FEAT_E3DSE is implemented. Otherwise, direct accesses to VDISR_EL3 are UNDEFINED.

Attributes

VDISR_EL3 is a 64-bit register.

Field descriptions

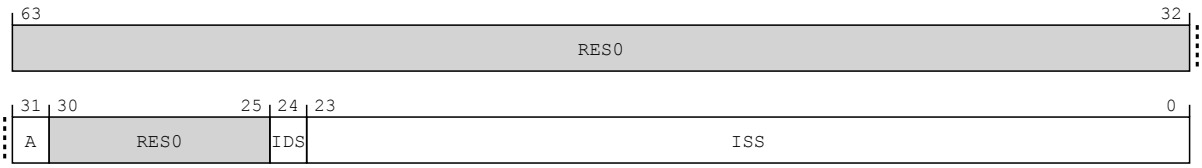

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an `ESB` instruction defers a delegated SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

The value copied from VSESR_EL3.IDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

The value copied from VSESR_EL3.ISS.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VDISR_EL3

An indirect write to VDISR_EL3 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR_EL1 occurring in program order after the ESB instruction.

Note

The accessibility pseudocode for DISR_EL1 has not been updated to show the effect of VDISR_EL3.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VDISR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_E3DSE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VDISR_EL3();
end;
```

```
MSR VDISR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_E3DSE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    VDISR_EL3() = X{64}(t);
end;
```

```
MRS <Xt>, DISR_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().AMO == '1' || (IsFeatureImplemented(FEAT_DoubleFault2) &&
        ↪IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) then
        X{64}(t) = VDISR_EL2();
    elsif HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        X{64}(t) = VDISR_EL3();
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = DISR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        X{64}(t) = VDISR_EL3();
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        X{64}(t) = Zeros{64};
    else
        X{64}(t) = DISR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = DISR_EL1();
end;
```

MSR DISR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (HCR_EL2().AMO == '1' || (IsFeatureImplemented(FEAT_DoubleFault2) &&
        ↪IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) then
        VDISR_EL2() = X{64}(t);
    elsif HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        VDISR_EL3() = X{64}(t);
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        return;
    else
        DISR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EffectiveSCR_EL3_EnDSE() == '1' then
        VDISR_EL3() = X{64}(t);
    elsif HaveEL(EL3) && !Halted() && SCR_EL3().EA == '1' then
        return;
    else
        DISR_EL1() = X{64}(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    DISR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.18 VSESR_EL2, Virtual SError Exception Syndrome Register (EL2)

The VSESR_EL2 characteristics are:

Purpose

Provides the syndrome value reported to software on taking a virtual SError exception to EL1, or on executing an `ESB` instruction at EL1.

When the virtual SError exception injected using HCR_EL2.VSE is taken to EL1 using AArch64, then the syndrome value is reported in ESR_EL1.

When the virtual SError exception injected using HCR_EL2.VSE is taken to EL1 using AArch32, then the syndrome value is reported in DFSR.{AET, ExT} and the remainder of DFSR is set as defined by VMSAv8-32. For more information, see The AArch32 Virtual Memory System Architecture.

When the virtual SError exception injected using HCR_EL2.VSE is deferred by an `ESB` instruction, then the syndrome value is written to VDISR_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VSESR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VDFSR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VSESR_EL2 are UNDEFINED.

Attributes

VSESR_EL2 is a 64-bit register.

Field descriptions

When EL1 is using AArch32:

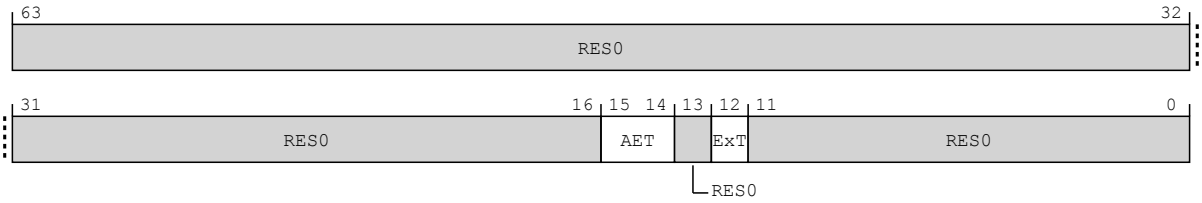

Bits [63:16]

Reserved, RES0.

AET, bits [15:14]

When a virtual SError exception is taken to EL1 using AArch32, DFSR[15:14] is set to VSESR_EL2.AET.

When a virtual SError exception is deferred by an `ESB` instruction, VDISR_EL2[15:14] is set to VSESR_EL2.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ExT, bit [12]

When a virtual SError exception is taken to EL1 using AArch32, DFSR[12] is set to VSESR_EL2.ExT.

When a virtual SError exception is deferred by an ESB instruction, VDISR_EL2[12] is set to VSESR_EL2.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

When EL1 is using AArch64:

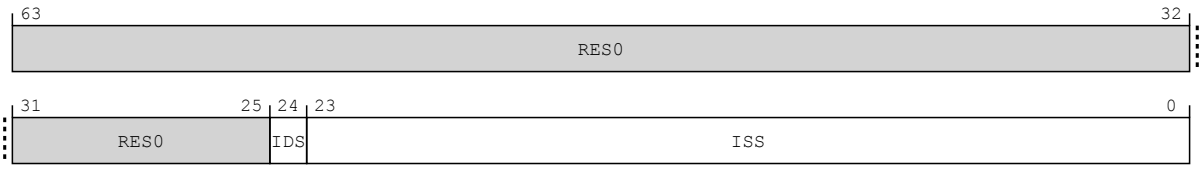

Bits [63:25]

Reserved, RES0.

IDS, bit [24]

When a virtual SError exception is taken to EL1 using AArch64, ESR_EL1[24] is set to VSESR_EL2.IDS.

When a virtual SError exception is deferred by an ESB instruction, VDISR_EL2[24] is set to VSESR_EL2.IDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

When a virtual SError exception is taken to EL1 using AArch64, ESR_EL1[23:0] is set to VSESR_EL2.ISS.

When a virtual SError exception is deferred by an ESB instruction, VDISR_EL2[23:0] is set to VSESR_EL2.ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSESR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VSESR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x508);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = VSESR_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VSESR_EL2();
end;
```

MSR VSESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_RAS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x508) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VSESR_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    VSESR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.9.19 VSESR_EL3, Virtual SError Exception Syndrome Register (EL3)

The VSESR_EL3 characteristics are:

Purpose

Provides the syndrome value reported to software when the Effective value of SCR_EL3.DSE is 1 on taking a delegated SError exception to EL2 or EL1, or on executing an `ESB` instruction at EL2 or EL1.

When the delegated SError exception injected using SCR_EL3.DSE is taken to EL2 using AArch64, then the syndrome value is reported in ESR_EL2.

When the delegated SError exception injected using SCR_EL3.DSE is taken to EL1 using AArch64, then the syndrome value is reported in ESR_EL1.

When the delegated SError exception injected using SCR_EL3.DSE is deferred by an `ESB` instruction, then the syndrome value is written to VDISR_EL3.

Configuration

This register is present only when FEAT_E3DSE is implemented. Otherwise, direct accesses to VSESR_EL3 are UNDEFINED.

Attributes

VSESR_EL3 is a 64-bit register.

Field descriptions

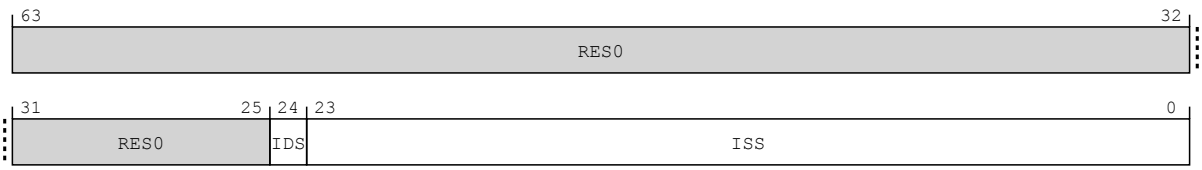

Bits [63:25]

Reserved, RES0.

IDS, bit [24]

When a delegated SError exception triggered by SCR_EL3.DSE is taken to EL2 or EL1 using AArch64, ESR_ELx[24] is set to VSESR_EL3.IDS.

When a delegated SError exception triggered by SCR_EL3.DSE is deferred by an `ESB` instruction, VDISR_EL3[24] is set to VSESR_EL3.IDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

When a delegated SError exception triggered by SCR_EL3.DSE is taken to EL2 or EL1 using AArch64, ESR_ELx[23:0] is set to VSESR_EL3.ISS.

When a delegated SError exception triggered by SCR_EL3.DSE is deferred by an `ESB` instruction, VDISR_EL3[23:0] is set to VSESR_EL3.ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing VSESR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VSESR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_E3DSE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = VSESR_EL3();
end;
```

```
MSR VSESR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_E3DSE) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    VSESR_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10 Generic Timer registers

This section lists the Generic Timer registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.1 CNTFRQ_EL0, Counter-timer Frequency Register

The CNTFRQ_EL0 characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configuration

AArch64 System register CNTFRQ_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTFRQ[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTFRQ_EL0 are UNDEFINED.

Attributes

CNTFRQ_EL0 is a 64-bit register.

Field descriptions

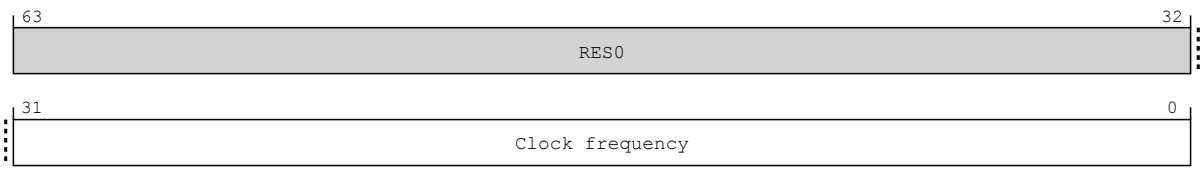

Bits [63:32]

Reserved, RES0.

ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTFRQ_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().[EL0PCTEN,EL0VCTEN] == '00' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().[EL0PCTEN,EL0VCTEN] == '00' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = CNTFRQ_EL0();
    end;
elsif PSTATE.EL == EL1 then
    X{64}(t) = CNTFRQ_EL0();
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTFRQ_EL0();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTFRQ_EL0();
end;
```

```
MSR CNTFRQ_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    CNTFRQ_EL0() = X{64}(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.2 CNTHCTL_EL2, Counter-timer Hypervisor Control Register

The CNTHCTL_EL2 characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from EL1 to the physical counter and the EL1 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

Fields that control the generation of events from the event stream have an effect regardless of the current Exception level and whether EL2 is enabled in the current Security state.

All other fields have no effect if EL2 is not enabled in the current Security state.

AArch64 System register CNTHCTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHCTL[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHCTL_EL2 are UNDEFINED.

Attributes

CNTHCTL_EL2 is a 64-bit register.

Field descriptions

When EffectiveHCR_EL2_E2H() == '1':

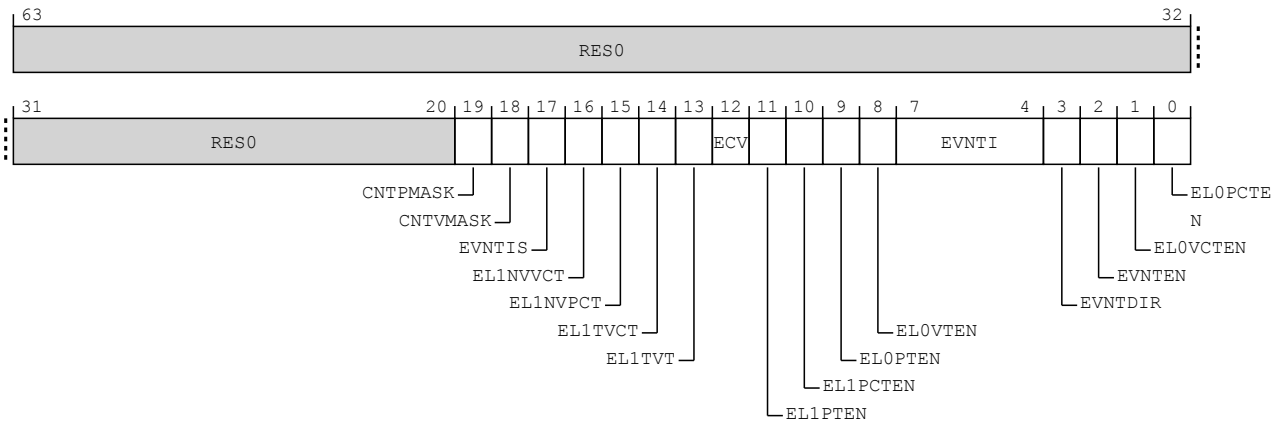

Bits [63:20]

Reserved, RES0.

CNTPMASK, bit [19]

When FEAT_RME is implemented:

CNTPMASK	Meaning
0b0	This control has no effect on CNTP_CTL_EL0.IMASK.
0b1	CNTP_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTPMASK	Meaning

This bit is RES0 in Non-secure and Secure state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CNTVMASK, bit [18]

When FEAT_RME is implemented:

CNTVMASK	Meaning
0b0	This control has no effect on CNTV_CTL_EL0.IMASK.
0b1	CNTV_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field.

This bit is RES0 in Non-secure and Secure state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVNTIS, bit [17]

When FEAT_ECV is implemented:

Controls the scale of the generation of the event stream.

EVNTIS	Meaning
0b0	The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[15:0].
0b1	The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVVCT, bit [16]

When FEAT_ECV is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When HCR_EL2.TGE is 0 and the Effective value of HCR_EL2.{NV2, NV1, NV} is {1, 0, 1}, traps EL1 accesses to the specified EL1 virtual timer registers using the EL02 descriptors to EL2 as follows:

Accesses to CNTV_CTL_EL02 and CNTV_CVAL_EL02 are trapped to EL2 and reported with EC syndrome value `0x18`.

EL1NVVCT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to the specified registers are trapped to EL2.

If HCR_EL2.TGE is 1 or the Effective value of HCR_EL2.{NV2, NV1, NV} is not {1, 0, 1}, this control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVPCT, bit [15]

When FEAT_ECV is implemented:

When HCR_EL2.TGE is 0 and the Effective value of HCR_EL2.{NV2, NV1, NV} is {1, 0, 1}, traps EL1 accesses to the specified EL1 physical timer registers using the EL02 descriptors to EL2 as follows:

Accesses to CNTP_CTL_EL02 and CNTP_CVAL_EL02 are trapped to EL2 and reported with EC syndrome value `0x18`.

EL1NVPCT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to the specified registers are trapped to EL2.

If HCR_EL2.TGE is 1 or the Effective value of HCR_EL2.{NV2, NV1, NV} is not {1, 0, 1}, this control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVCT, bit [14]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_ECV is implemented:

Traps EL0 and EL1 accesses to the EL1 virtual counter registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTVCT_EL0 and CNTVCTSS_EL0 are trapped to EL2 and reported using EC syndrome value `0x18`, unless they are trapped by CNTKCTL_EL1.EL0VCTEN.
- In AArch32 state, accesses to CNTVCT are trapped to EL2 and reported with EC syndrome value `0x04`, unless they are trapped by CNTKCTL_EL1.EL0VCTEN or CNTKCTL.PL0VCTEN.

EL1TVCT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 and EL1 accesses to the specified registers are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

If HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVT, bit [13]

When FEAT_ECV is implemented:

When the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, traps EL0 and EL1 accesses to the EL1 virtual timer registers to EL2, when EL2 is enabled for the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless they are trapped by CNTKCTL_EL1.EL0VTEN:
  - CNTV_CTL_EL0.
  - CNTV_CVAL_EL0.
  - CNTV_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped to EL2 and reported using EC syndrome value `0x03`, and MCRR and MRRC accesses are trapped to EL2 and reported using EC syndrome value `0x04`, unless they are trapped by CNTKCTL_EL1.EL0VTEN or CNTKCTL.PL0VTEN:
  - CNTV_CTL.
  - CNTV_CVAL.
  - CNTV_TVAL.

EL1TVT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 and EL1 accesses to the specified registers are trapped to EL2.

If HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ECV, bit [12]

When FEAT_ECV_POFF is implemented:

Enables the Enhanced Counter Virtualization functionality registers.

When the Enhanced Counter Virtualization is enabled, the behavior is as follows:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value (`PCount`<63:0> - CNTPOFF_EL2<63:0>).
- The EL1 physical timer interrupt is triggered when ((`PCount`<63:0> - CNTPOFF_EL2<63:0>) - `PCVal`<63:0>) is greater than or equal to 0.

`PCount`<63:0> is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3.

`PCVal`<63:0> is the EL1 physical timer compare value for this timer.

ECV	Meaning
`0b0`	Enhanced Counter Virtualization functionality is disabled.
`0b1`	Enhanced Counter Virtualization functionality is enabled.

When HCR_EL2.TGE is 1 or SCR_EL3.{NS, EEL2} is {0, 0}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1PTEN, bit [11]

When HCR_EL2.TGE is 0, traps EL0 and EL1 accesses to the EL1 physical timer registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless they are trapped by CNTKCTL_EL1.EL0PTEN:
  - CNTP_CTL_EL0.
  - CNTP_CVAL_EL0.
  - CNTP_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03` and MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`, unless they are trapped by CNTKCTL_EL1.EL0PTEN or CNTKCTL.PL0PTEN:
  - CNTP_CTL.
  - CNTP_CVAL.
  - CNTP_TVAL.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL1PTEN	Meaning
0b0	EL0 and EL1 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL1PCTEN, bit [10]

When HCR_EL2.TGE is 0, traps EL0 and EL1 accesses to the EL1 physical counter registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTPCT_EL0 and CNTPCTSS_EL0 are trapped to EL2 and reported with EC syndrome value 0x18, unless they are trapped by CNTKCTL_EL1.EL0PCTEN.
- In AArch32 state, MRRC or MCRR accesses to CNTPCT are trapped to EL2 and reported with EC syndrome value 0x04, unless they are trapped by CNTKCTL_EL1.EL0PCTEN or CNTKCTL.PL0PCTEN.

EL1PCTEN	Meaning
0b0	EL0 and EL1 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0PTEN, bit [9]

When HCR_EL2.TGE is 1, traps EL0 accesses to the physical timer registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value 0x18:
  - CNTP_CTL_EL0.
  - CNTP_CVAL_EL0.
  - CNTP_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value 0x03 and MRRC and MCRR accesses are trapped and reported using EC syndrome value 0x04:
  - CNTP_CTL.
  - CNTP_CVAL.
  - CNTP_TVAL.

EL0PTEN	Meaning
0b0	EL0 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0PTEN	Meaning

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

When HCR_EL2.TGE is 1, traps EL0 accesses to the virtual timer registers to EL2 as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value `0x18`:
  - CNTV_CTL_EL0.
  - CNTV_CVAL_EL0.
  - CNTV_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03` and MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`:
  - CNTV_CTL.
  - CNTV_CVAL.
  - CNTV_TVAL.

EL0VTEN	Meaning
`0b0`	EL0 accesses to the specified registers are trapped to EL2.
`0b1`	This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit of CNTPCT_EL0, as seen from EL2, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTHCTL_EL2.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTPCT_EL0.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTPCT_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTPCT_EL0 trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from CNTPCT_EL0 as seen from EL2.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VCTEN, bit [1]

When HCR_EL2.TGE is 1, traps EL0 accesses to the frequency register and virtual counter registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value 0x18:
  - CNTVCT_EL0.
  - CNTVCTSS_EL0.
  - CNTFRQ_EL0 if CNTHCTL_EL2.EL0PCTEN is 0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value 0x03 and MRRC and MCRR accesses are trapped and reported using EC syndrome value 0x04:
  - CNTVCT and if CNTHCTL.EL0PCTEN is 0, CNTFRQ.

EL0VCTEN	Meaning
0b0	EL0 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If HCR_EL2.TGE is 0, the field is ignored for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Traps EL0 accesses to the frequency register and physical counter registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value 0x18:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - CNTPCT_EL0.
  - CNTPCTSS_EL0.
  - CNTFRQ_EL0 if CNTHCTL_EL2.EL0VCTEN is 0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03` and MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`:
  - CNTPCT and if CNTHCTL_EL2.EL0VCTEN is 0, CNTFRQ.

EL0PCTEN	Meaning
`0b0`	From AArch64 state: EL0 accesses to the specified registers are trapped to EL2.
`0b1`	This control does not cause any instructions to be trapped.

If HCR_EL2.TGE is 0, the control does not cause any instructions to be trapped for all purposes other than direct reads and writes of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

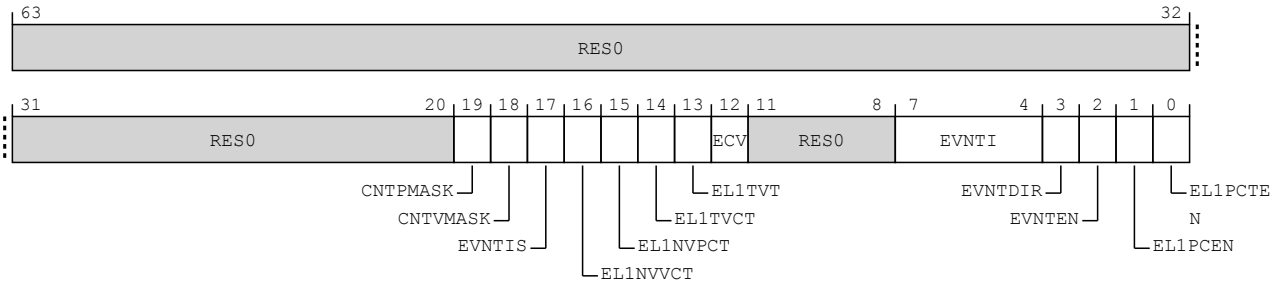

The following field descriptions apply in all Armv8.0 implementations.

The descriptions also explain the behavior when EL3 is implemented and EL2 is not implemented.

Bits [63:20]

Reserved, RES0.

CNTPMASK, bit [19]

When FEAT_RME is implemented:

CNTPMASK	Meaning
`0b0`	This control has no effect on CNTP_CTL_EL0.IMASK.
`0b1`	CNTP_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field.

This bit is RES0 in Non-secure and Secure state.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CNTVMASK, bit [18]

When FEAT_RME is implemented:

CNTVMASK	Meaning
0b0	This control has no effect on CNTV_CTL_EL0.IMASK.
0b1	CNTV_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field.

This bit is RES0 in Non-secure and Secure state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVNTIS, bit [17]

When FEAT_ECV is implemented:

Controls the scale of the generation of the event stream.

EVNTIS	Meaning
0b0	The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[15:0].
0b1	The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVVCT, bit [16]

When FEAT_ECV is implemented:

When the Effective value of HCR_EL2.{NV2, NV1, NV} is {1, 0, 1}, traps EL1 accesses to the specified EL1 virtual timer registers using the EL02 descriptors to EL2 as follows:

Accesses to CNTV_CTL_EL02 and CNTV_CVAL_EL02 are trapped to EL2 and reported with EC syndrome value 0x18.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL1NVVCT	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified registers are trapped to EL2.

If the Effective value of HCR_EL2.{NV2, NV1, NV} is not {1, 0, 1}, this control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVPCT, bit [15]

When FEAT_ECV is implemented:

When the Effective value of HCR_EL2.{NV2, NV1, NV} is {1, 0, 1}, traps EL1 accesses to the specified EL1 physical timer registers using the EL02 descriptors to EL2 as follows:

Accesses to CNTP_CTL_EL02 and CNTP_CVAL_EL02 are trapped to EL2 and reported with EC syndrome value 0x18.

EL1NVPCT	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified registers are trapped to EL2.

If the Effective value of HCR_EL2.{NV2, NV1, NV} is not {1, 0, 1}, this control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVCT, bit [14]

When FEAT_ECV is implemented:

Traps EL0 and EL1 accesses to the EL1 virtual counter registers to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTVCT_EL0 and CNTVCTSS_EL0 are trapped to EL2 and reported using EC syndrome value 0x18, unless they are trapped by CNTKCTL_EL1.EL0VCTEN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In AArch32 state, accesses to CNTVCT are trapped to EL2 and reported using EC syndrome value `0x04`, unless they are trapped by CNTKCTL_EL1.EL0VCTEN or CNTKCTL.PL0VCTEN.

EL1TVCT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 and EL1 accesses to the specified registers are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVT, bit [13]

When FEAT_ECV is implemented:

If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, then traps EL0 and EL1 accesses to the EL1 virtual timer registers to EL2, when EL2 is enabled for the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 and reported with EC syndrome value `0x18`, unless they are trapped by CNTKCTL_EL1.EL0VTEN:
  - CNTV_CTL_EL0.
  - CNTV_CVAL_EL0.
  - CNTV_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03` and MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`, unless they are trapped by CNTKCTL_EL1.EL0VTEN or CNTKCTL.PL0VTEN:
  - CNTV_CTL.
  - CNTV_CVAL.
  - CNTV_TVAL.

EL1TVT	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 and EL1 accesses to the specified registers are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ECV, bit [12]

When FEAT_ECV_POFF is implemented:

Enables the Enhanced Counter Virtualization functionality registers.

When the Enhanced Counter Virtualization is enabled, the behavior is as follows:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value (`PCount`<63:0> - CNTPOFF_EL2<63:0>).
- The EL1 physical timer interrupt is triggered when ((`PCount`<63:0> - CNTPOFF_EL2<63:0>) - `PCVal`<63:0>) is greater than or equal to 0.

`PCount` is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3.

`PCVal`<63:0> is the EL1 physical timer compare value for this timer.

ECV	Meaning
`0b0`	Enhanced Counter Virtualization functionality is disabled.
`0b1`	Enhanced Counter Virtualization functionality is enabled.

When SCR_EL3.{NS, EEL2} is {0, 0} or if FEAT_ECV_POFF is not implemented, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [11:8]

Reserved, RES0.

EVNTI, bits [7:4]

Selects which bit of CNTPCT_EL0, as seen from EL2,is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTHCTL_EL2.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTPCT_EL0.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTPCT_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTPCT_EL0 trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
`0b0`	A 0 to 1 transition of the trigger bit triggers an event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from CNTPCT_EL0 as seen from EL2.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL1PCEN, bit [1]

Traps EL0 and EL1 accesses to the EL1 physical timer registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTP_CTL_EL0, CNTP_CVAL_EL0, CNTP_TVAL_EL0 are trapped to EL2, reported using EC syndrome value 0x18, unless they are trapped by CNTKCTL_EL1.EL0PTEN.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using EC syndrome value 0x03, and MRRC and MCRR accesses are trapped to EL2, reported using EC syndrome value 0x04, unless they are trapped by CNTKCTL_EL1.EL0PTEN or CNTKCTL.PL0PTEN:
  - CNTP_CTL, CNTP_CVAL, CNTP_TVAL.

EL1PCEN	Meaning
0b0	EL0 and EL1 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL1PCTEN, bit [0]

Traps EL0 and EL1 accesses to the EL1 physical counter registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTPCT_EL0 and CNTPCTSS_EL0 are trapped to EL2 and reported using EC syndrome value 0x18, unless they are trapped by CNTKCTL_EL1.EL0PCTEN.
- In AArch32 state, MRRC or MCRR accesses to CNTPCT are trapped to EL2 and reported using EC syndrome value 0x04, unless they are trapped by CNTKCTL_EL1.EL0PCTEN or CNTKCTL.PL0PCTEN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL1PCTEN	Meaning
0b0	EL0 and EL1 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHCTL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name CNTHCTL_EL2 or CNTKCTL_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHCTL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTHCTL_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTHCTL_EL2();
end;
```

```
MSR CNTHCTL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0001	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CNTHCTL_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CNTHCTL_EL2() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTKCTL_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = CNTKCTL_EL1();
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTHCTL_EL2_VHE(CNTHCTL_EL2());
    else
        X{64}(t) = CNTKCTL_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTKCTL_EL1();
end;
```

When FEAT_VHE is implemented MSR CNTKCTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1() = X{64}(t);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTHCTL_EL2() = CNTHCTL_EL2_VHE(X{64}(t));
    else
        CNTKCTL_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    CNTKCTL_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.3 CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control Register

The CNTHP_CTL_EL2 characteristics are:

Purpose

Control register for the EL2 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHP_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHP_CTL[31:0].

This register is present only when (HaveEL(EL3) || (((!HaveEL(EL3)) && HaveEL(EL2)) && (!IsFeatureImplemented(FEAT_SEL2)))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to CNTHP_CTL_EL2 are UNDEFINED.

Attributes

CNTHP_CTL_EL2 is a 64-bit register.

Field descriptions

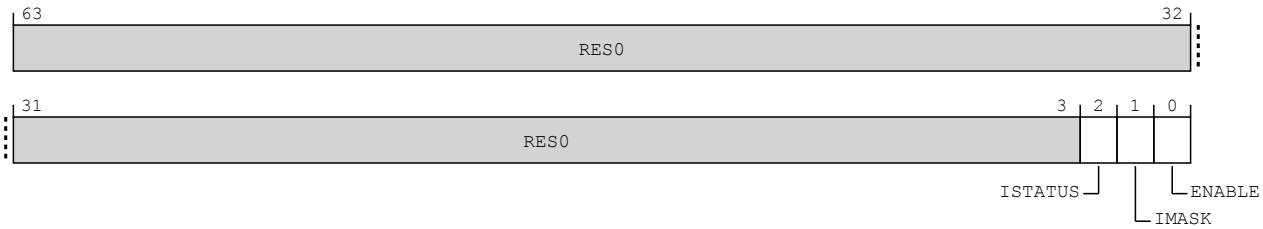

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CTL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CNTHP_CTL_EL2` or `CNTP_CTL_EL0` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHP_CTL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !((HaveEL(EL3) || (!HaveEL(EL3) && HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTHP_CTL_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTHP_CTL_EL2();
end;
```

MSR CNTHP_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b001

```
if !((HaveEL(EL3) || (!HaveEL(EL3) && HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CNTHP_CTL_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CNTHP_CTL_EL2() = X{64}(t);
end;
```

MRS <Xt>, CNTP_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            X{64}(t) = CNTHPS_CTL_EL2();
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            X{64}(t) = CNTHP_CTL_EL2();
        else
            X{64}(t) = CNTP_CTL_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x180);
        else
            X{64}(t) = CNTP_CTL_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            X{64}(t) = CNTHPS_CTL_EL2();
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            X{64}(t) = CNTHP_CTL_EL2();
        else
            X{64}(t) = CNTP_CTL_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = CNTP_CTL_EL0();
    end;
```

MSR CNTP_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            CNTHPS_CTL_EL2() = X{64}(t);
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            CNTHP_CTL_EL2() = X{64}(t);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CNTP_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x180) = X{64}(t);
    else
        CNTP_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CTL_EL2() = X{64}(t);
    else
        CNTP_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CTL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.4 CNTHP_CVAL_EL2, Counter-timer Physical Timer CompareValue Register (EL2)

The CNTHP_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the EL2 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHP_CVAL_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTHP_CVAL[63:0].

This register is present only when (HaveEL(EL3) || (((!HaveEL(EL3)) && HaveEL(EL2)) && (!IsFeatureImplemented(FEAT_SEL2)))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to CNTHP_CVAL_EL2 are UNDEFINED.

Attributes

CNTHP_CVAL_EL2 is a 64-bit register.

Field descriptions

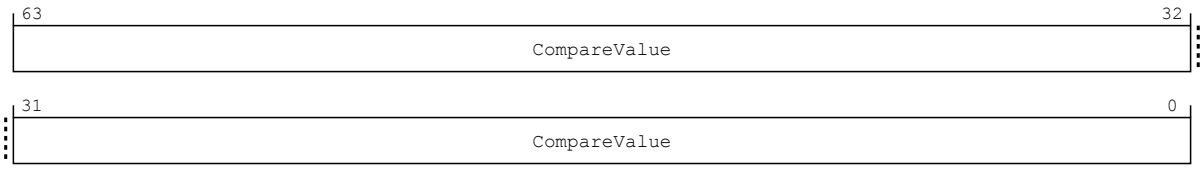

CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHP_CTL_EL2.ENABLE is 1, and TimerConditionMet is TRUE for the EL2 physical timer, the timer condition is met and all of the following are true:

- CNTHP_CTL_EL2.ISTATUS is set to 1.
- If CNTHP_CTL_EL2.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTHP_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CVAL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CNTHP_CVAL_EL2` or `CNTP_CVAL_EL0` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTHP_CVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b010

```
if !((HaveEL(EL3) || (!HaveEL(EL3) && HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTHP_CVAL_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTHP_CVAL_EL2();
end;
```

```
MSR CNTHP_CVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b010

```
if !((HaveEL(EL3) || (!HaveEL(EL3) && HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CNTHP_CVAL_EL2() = X{64}(t);
end;
```

```
MRS <Xt>, CNTP_CVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CVAL_EL2();
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x178);
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CVAL_EL2();
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTP_CVAL_EL0();
end;
```

MSR CNTP_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x178) = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.5 CNTHP_TVAL_EL2, Counter-timer Physical Timer TimerValue Register (EL2)

The CNTHP_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the EL2 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHP_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHP_TVAL[31:0].

This register is present only when (HaveEL(EL3) || (((!HaveEL(EL3)) && HaveEL(EL2)) && (!IsFeatureImplemented(FEAT_SEL2)))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to CNTHP_TVAL_EL2 are UNDEFINED.

Attributes

CNTHP_TVAL_EL2 is a 64-bit register.

Field descriptions

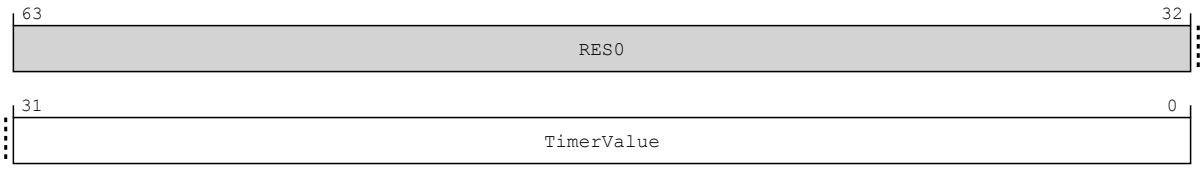

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHP_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHP_CTL_EL2.ENABLE is 1, the value returned is (CNTHP_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHP_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHP_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHP_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHP_CTL_EL2.ISTATUS is set to 1.
- If CNTHP_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHP_CTL_EL2.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTHP_TVAL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CNTHP_TVAL_EL2` or `CNTP_TVAL_EL0` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHP_TVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b000

```
if !((HaveEL(EL3) || (!HaveEL(EL3) && HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if CNTHP_CTL_EL2().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
    end;
elsif PSTATE.EL == EL3 then
    if CNTHP_CTL_EL2().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
    end;
end;
```

```
MSR CNTHP_TVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0010	0b000

```
if !((HaveEL(EL3) || (!HaveEL(EL3) && HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
end;
```

```
MRS <Xt>, CNTP_TVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
     ↪'1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
     ↪'1') && CNTHCTL_EL2().ECV == '1' then
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if CNTP_CTL_EL0().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
    end;
end;
```

MSR CNTP_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
         ↪'1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
        CNTP_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTPOFF_EL2();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
         ↪'1') && CNTHCTL_EL2().ECV == '1' then
        CNTP_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTPOFF_EL2();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.6 CNTHPS_CTL_EL2, Counter-timer Secure Physical Timer Control Register (EL2)

The CNTHPS_CTL_EL2 characteristics are:

Purpose

Control register for the Secure EL2 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHPS_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHPS_CTL[31:0].

This register is present only when FEAT_SEL2 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHPS_CTL_EL2 are UNDEFINED.

Attributes

CNTHPS_CTL_EL2 is a 64-bit register.

Field descriptions

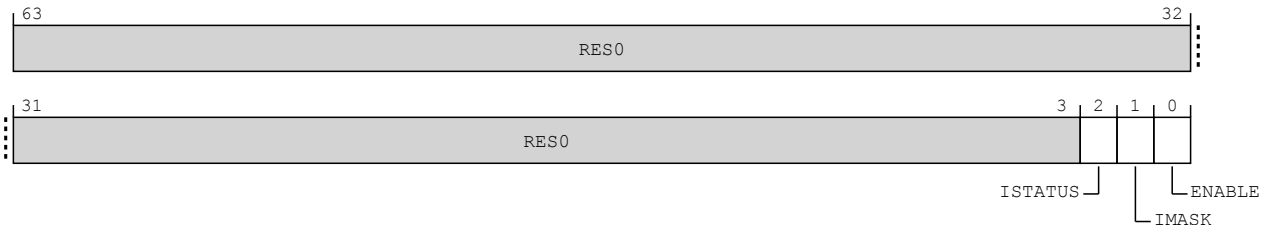

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHPS_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_CTL_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHPS_CTL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        X{64}(t) = CNTHPS_CTL_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        X{64}(t) = CNTHPS_CTL_EL2();
    end;
end;
```

MSR CNTHPS_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        CNTHPS_CTL_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        CNTHPS_CTL_EL2() = X{64}(t);
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTP_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2)
     ↪then
        X{64}(t) = CNTHPS_CTL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CTL_EL2();
    else
        X{64}(t) = CNTP_CTL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x180);
    else
        X{64}(t) = CNTP_CTL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CTL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CTL_EL2();
    else
        X{64}(t) = CNTP_CTL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTP_CTL_EL0();
end;
```

When FEAT_VHE is implemented MSR CNTP_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2)
         ↪then
        CNTHPS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CTL_EL2() = X{64}(t);
    else
        CNTP_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x180) = X{64}(t);
    else
        CNTP_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CTL_EL2() = X{64}(t);
    else
        CNTP_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CTL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.7 CNTHPS_CVAL_EL2, Counter-timer Secure Physical Timer CompareValue Register (EL2)

The CNTHPS_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the Secure EL2 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHPS_CVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHPS_CVAL[31:0].

This register is present only when EL2 is implemented, FEAT_SEL2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHPS_CVAL_EL2 are UNDEFINED.

Attributes

CNTHPS_CVAL_EL2 is a 64-bit register.

Field descriptions

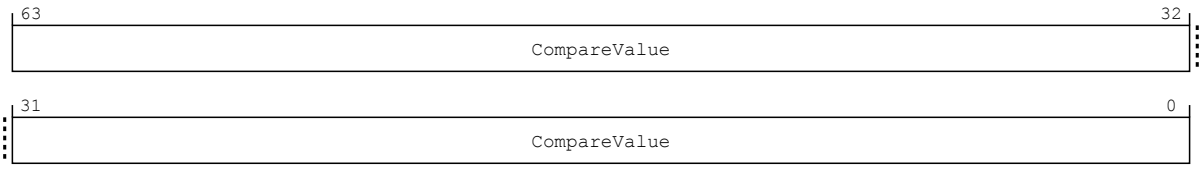

CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL_EL2.ENABLE is 1, and TimerConditionMet is TRUE for the Secure EL2 physical timer, the timer condition is met and all of the following are true:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_CVAL_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTHPS_CVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b010

```
if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        X{64}(t) = CNTHPS_CVAL_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        X{64}(t) = CNTHPS_CVAL_EL2();
    end;
end;
```

```
MSR CNTHPS_CVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b010

```
if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        CNTHPS_CVAL_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        CNTHPS_CVAL_EL2() = X{64}(t);
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTP_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2)
     ↪then
        X{64}(t) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CVAL_EL2();
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x178);
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CVAL_EL2();
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTP_CVAL_EL0();
end;
```

When FEAT_VHE is implemented MSR CNTP_CVAL_EL0, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2)
         ↪then
        CNTHPS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x178) = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.8 CNTHPS_TVAL_EL2, Counter-timer Secure Physical Timer TimerValue Register (EL2)

The CNTHPS_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the Secure EL2 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHPS_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHPS_TVAL[31:0].

This register is present only when EL2 is implemented, FEAT_SEL2 is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHPS_TVAL_EL2 are UNDEFINED.

Attributes

CNTHPS_TVAL_EL2 is a 64-bit register.

Field descriptions

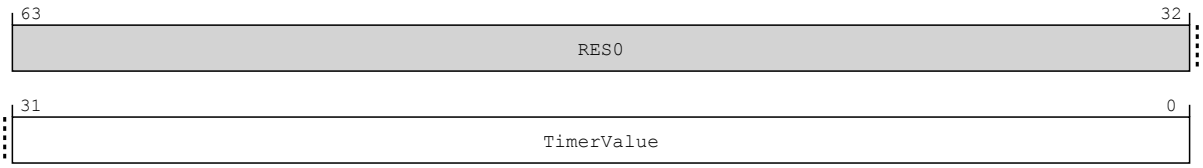

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL_EL2.ENABLE is 1, the value returned is (CNTHPS_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHPS_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHPS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_TVAL_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTHPS_TVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b000

```
if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    end;
end;
```

```
MSR CNTHPS_TVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b000

```
if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTP_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2)
         ↪then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn
         ↪== '1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - (PhysicalCountInt() -
                 ↪CNTPOFF_EL2()))[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn
             ↪== '1') && CNTHCTL_EL2().ECV == '1' then
            if CNTP_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - (PhysicalCountInt() -
                     ↪CNTPOFF_EL2()))[31:0]);
            end;
        else
            if CNTP_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            if CNTHPS_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            if CNTHP_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        else
            if CNTP_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL3 then
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
```

When FEAT_VHE is implemented MSR CNTP_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2)
         ↪then
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn
         ↪== '1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
        CNTP_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTPOFF_EL2();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn
         ↪== '1') && CNTHCTL_EL2().ECV == '1' then
        CNTP_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTPOFF_EL2();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.9 CNTHV_CTL_EL2, Counter-timer Virtual Timer Control Register (EL2)

The CNTHV_CTL_EL2 characteristics are:

Purpose

Control register for the EL2 virtual timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHV_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHV_CTL[31:0].

This register is present only when (IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || ((!HaveEL(EL3)) && (!IsFeatureImplemented(FEAT_SEL2))))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to CNTHV_CTL_EL2 are UNDEFINED.

Attributes

CNTHV_CTL_EL2 is a 64-bit register.

Field descriptions

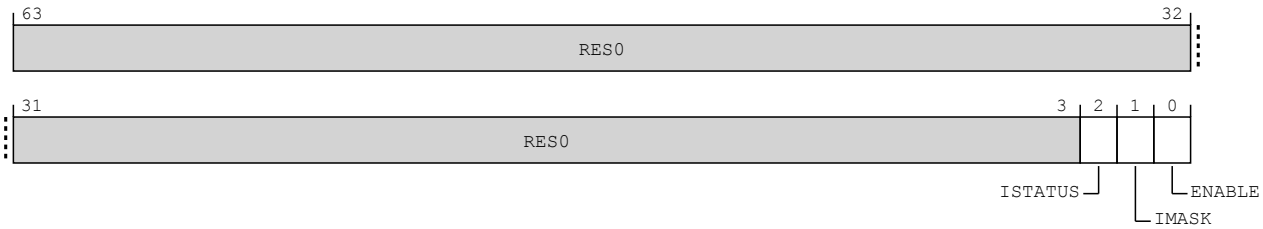

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHV_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHV_CTL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CNTHV_CTL_EL2` or `CNTV_CTL_EL0` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHV_CTL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || (!HaveEL(EL3) &&
     ↪!IsFeatureImplemented(FEAT_SEL2))) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTHV_CTL_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTHV_CTL_EL2();
end;
```

MSR CNTHV_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || (!HaveEL(EL3) &&
     ↪!IsFeatureImplemented(FEAT_SEL2))) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CNTHV_CTL_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CNTHV_CTL_EL2() = X{64}(t);
end;
```

MRS <Xt>, CNTV_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
             ↪'1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            X{64}(t) = CNTHVS_CTL_EL2();
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            X{64}(t) = CNTHV_CTL_EL2();
        else
            X{64}(t) = CNTV_CTL_EL0();
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            X{64}(t) = NVMem(0x170);
        else
            X{64}(t) = CNTV_CTL_EL0();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            X{64}(t) = CNTHVS_CTL_EL2();
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            X{64}(t) = CNTHV_CTL_EL2();
        else
            X{64}(t) = CNTV_CTL_EL0();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = CNTV_CTL_EL0();
    end;
```

MSR CNTV_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
             ↪'1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            CNTHVS_CTL_EL2() = X{64}(t);
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            CNTHV_CTL_EL2() = X{64}(t);
        else
            CNTV_CTL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x170) = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2() = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTV_CTL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.10 CNTHV_CVAL_EL2, Counter-timer Virtual Timer CompareValue Register (EL2)

The CNTHV_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the EL2 virtual timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHV_CVAL_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTHV_CVAL[63:0].

This register is present only when (IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || ((!HaveEL(EL3)) && (!IsFeatureImplemented(FEAT_SEL2))))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to CNTHV_CVAL_EL2 are UNDEFINED.

Attributes

CNTHV_CVAL_EL2 is a 64-bit register.

Field descriptions

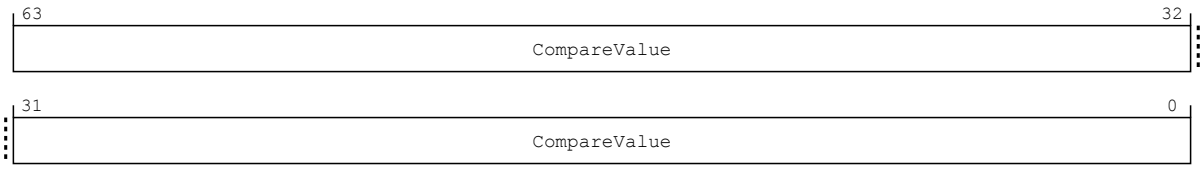

CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHV_CTL_EL2.ENABLE is 1, and TimerConditionMet is TRUE for the EL2 virtual timer, the timer condition is met and all of the following are true:

- CNTHV_CTL_EL2.ISTATUS is set to 1.
- If CNTHV_CTL_EL2.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTHV_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHV_CVAL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CNTHV_CVAL_EL2` or `CNTV_CVAL_EL0` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MRS <Xt>, CNTHV_CVAL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || (!HaveEL(EL3) &&
     ↪!IsFeatureImplemented(FEAT_SEL2))) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTHV_CVAL_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTHV_CVAL_EL2();
end;
```

MSR CNTHV_CVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b010

```
if !(IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || (!HaveEL(EL3) &&
     ↪!IsFeatureImplemented(FEAT_SEL2))) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CNTHV_CVAL_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CNTHV_CVAL_EL2() = X{64}(t);
end;
```

MRS <Xt>, CNTV_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
         ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CVAL_EL2();
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x168);
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CVAL_EL2();
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTV_CVAL_EL0();
end;
```

```
MSR CNTV_CVAL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
         ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = X{64}(t);
    else
        CNTV_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x168) = X{64}(t);
    else
        CNTV_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = X{64}(t);
    else
        CNTV_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTV_CVAL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.11 CNTHV_TVAL_EL2, Counter-timer Virtual Timer TimerValue Register (EL2)

The CNTHV_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the EL2 virtual timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHV_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHV_TVAL[31:0].

This register is present only when (IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || ((!HaveEL(EL3)) && (!IsFeatureImplemented(FEAT_SEL2))))) && IsFeatureImplemented(FEAT_AA64). Otherwise, direct accesses to CNTHV_TVAL_EL2 are UNDEFINED.

Attributes

CNTHV_TVAL_EL2 is a 64-bit register.

Field descriptions

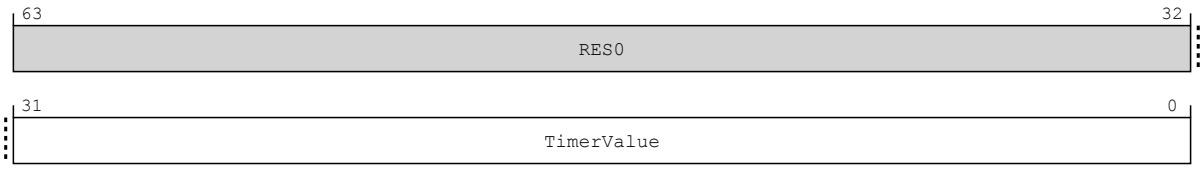

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHV_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHV_CTL_EL2.ENABLE is 1, the value returned is (CNTHV_CVAL_EL2 - CNTVCT_EL0).

On a write of this register, CNTHV_CVAL_EL2 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHV_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTHV_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHV_CTL_EL2.ISTATUS is set to 1.
- If CNTHV_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHV_CTL_EL2.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTHV_TVAL_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `CNTHV_TVAL_EL2` or `CNTV_TVAL_EL0` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHV_TVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || (!HaveEL(EL3) &&
     ↪!IsFeatureImplemented(FEAT_SEL2))) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if CNTHV_CTL_EL2().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
    end;
elsif PSTATE.EL == EL3 then
    if CNTHV_CTL_EL2().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
    end;
end;
```

```
MSR CNTHV_TVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || (!HaveEL(EL3) &&
     ↪!IsFeatureImplemented(FEAT_SEL2))) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
end;
```

MRS <Xt>, CNTV_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHV_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif HaveEL(EL2) && !ELIsInHost(EL0) then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        end;
    else
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        end;
    else
        if CNTV_CTL_EL0().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            if CNTHVS_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            if CNTHV_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif !ELIsInHost(EL2) then
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
            end;
        else
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL3 then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF()))[31:0]);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
```

MSR CNTV_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
        ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif HaveEL(EL2) && !ELIsInHost(EL0) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif !ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.12 CNTHVS_CTL_EL2, Counter-timer Secure Virtual Timer Control Register (EL2)

The CNTHVS_CTL_EL2 characteristics are:

Purpose

Control register for the Secure EL2 virtual timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHVS_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHVS_CTL[31:0].

This register is present only when FEAT_SEL2 is implemented, FEAT_VHE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHVS_CTL_EL2 are UNDEFINED.

Attributes

CNTHVS_CTL_EL2 is a 64-bit register.

Field descriptions

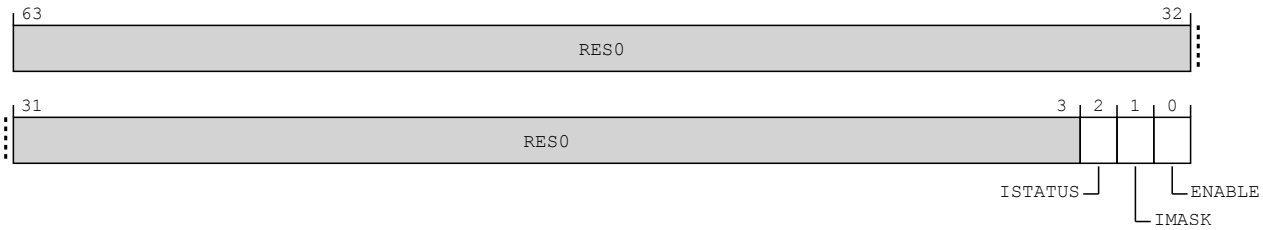

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the CNTHVS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the CNTHVS_CTL_EL2.ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHVS_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHVS_CTL_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHVS_CTL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        X{64}(t) = CNTHVS_CTL_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        X{64}(t) = CNTHVS_CTL_EL2();
    end;
end;
```

MSR CNTHVS_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        CNTHVS_CTL_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        CNTHVS_CTL_EL2() = X{64}(t);
    end;
end;
```

MRS <Xt>, CNTV_CTL_EL0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CTL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CTL_EL2();
    else
        X{64}(t) = CNTV_CTL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x170);
    else
        X{64}(t) = CNTV_CTL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CTL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CTL_EL2();
    else
        X{64}(t) = CNTV_CTL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTV_CTL_EL0();
end;
```

MSR CNTV_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2() = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x170) = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2() = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTV_CTL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.13 CNTHVS_CVAL_EL2, Counter-timer Secure Virtual Timer CompareValue Register (EL2)

The CNTHVS_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the Secure EL2 virtual timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHVS_CVAL_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTHVS_CVAL[63:0].

This register is present only when FEAT_SEL2 is implemented, FEAT_VHE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHVS_CVAL_EL2 are UNDEFINED.

Attributes

CNTHVS_CVAL_EL2 is a 64-bit register.

Field descriptions

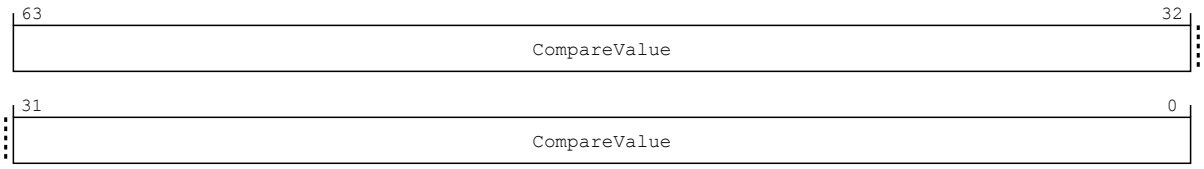

CompareValue, bits [63:0]

Holds the Secure EL2 virtual timer CompareValue.

When CNTHVS_CTL_EL2.ENABLE is 1, and TimerConditionMet is TRUE for the Secure EL2 virtual timer, the timer condition is met and all of the following are true:

- CNTHVS_CTL_EL2.ISTATUS is set to 1.
- If CNTHVS_CTL_EL2.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by ‘Operation of the CompareValue views of the timers’.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTHVS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHVS_CVAL_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTHVS_CVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        X{64}(t) = CNTHVS_CVAL_EL2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        X{64}(t) = CNTHVS_CVAL_EL2();
    end;
end;
```

```
MSR CNTHVS_CVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        CNTHVS_CVAL_EL2() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        CNTHVS_CVAL_EL2() = X{64}(t);
    end;
end;
```

MRS <Xt>, CNTV_CVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CVAL_EL2();
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x168);
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CVAL_EL2();
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTV_CVAL_EL0();
end;
```

MSR CNTV_CVAL_EL0, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = X{64}(t);
    else
        CNTV_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x168) = X{64}(t);
    else
        CNTV_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = X{64}(t);
    else
        CNTV_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTV_CVAL_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.14 CNTHVS_TVAL_EL2, Counter-timer Secure Virtual Timer TimerValue Register (EL2)

The CNTHVS_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the Secure EL2 virtual timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch64 System register CNTHVS_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHVS_TVAL[31:0].

This register is present only when FEAT_SEL2 is implemented, FEAT_VHE is implemented, and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTHVS_TVAL_EL2 are UNDEFINED.

Attributes

CNTHVS_TVAL_EL2 is a 64-bit register.

Field descriptions

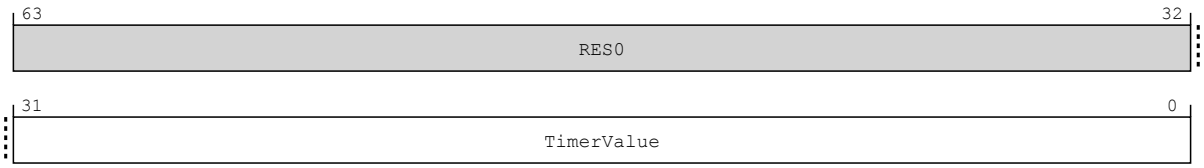

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHVS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHVS_CTL_EL2.ENABLE is 1, the value returned is (CNTHVS_CVAL_EL2 - CNTVCT_EL0).

On a write of this register, CNTHVS_CVAL_EL2 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHVS_CTL_EL2.ENABLE is 1, the timer condition is met when ((CNTVCT_EL0 - CNTHVS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHVS_CTL_EL2.ISTATUS is set to 1.
- If CNTHVS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL_EL2.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHVS_TVAL_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTHVS_TVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    end;
end;
```

```
MSR CNTHVS_TVAL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE) &&
     ↪IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    else
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if SCR_EL3().EEL2 == '0' then
        Undefined();
    else
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
end;
```

MRS <Xt>, CNTV_TVAL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHV_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif HaveEL(EL2) && !ELIsInHost(EL0) then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        end;
    else
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL2) then
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
            end;
        else
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            if CNTHVS_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            if CNTHV_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif !ELIsInHost(EL2) then
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
            end;
        else
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL3 then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF()))[31:0]);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
```

```
MSR CNTV_TVAL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif HaveEL(EL2) && !ELIsInHost(EL0) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif !ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.15 CNTKCTL_EL1, Counter-timer Kernel Control Register

The CNTKCTL_EL1 characteristics are:

Purpose

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this register does not cause any event stream from the virtual counter to be generated, and does not control access to the counters and timers. The access to counters and timers at EL0 is controlled by CNTHCTL_EL2.

When FEAT_VHE is not implemented, or when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, this register controls the generation of an event stream from the virtual counter, and access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

Configuration

AArch64 System register CNTKCTL_EL1 bits [31:0] are architecturally mapped to AArch32 System register CNTKCTL[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTKCTL_EL1 are UNDEFINED.

Attributes

CNTKCTL_EL1 is a 64-bit register.

Field descriptions

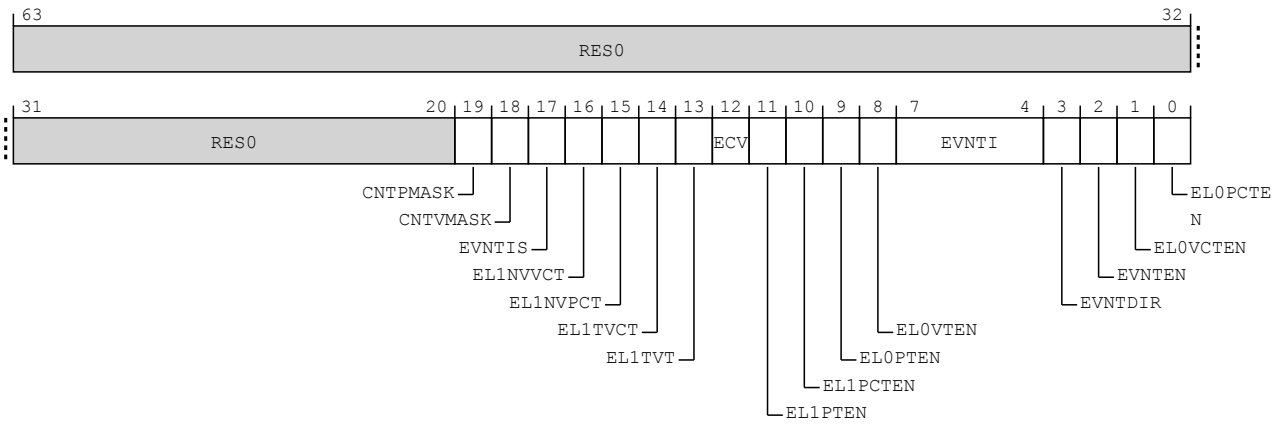

Bits [63:20]

Reserved, RES0.

CNTPMASK, bit [19]

When FEAT_RME is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CNTVMASK, bit [18]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVNTIS, bit [17]

When FEAT_ECV is implemented:

Controls the scale of the generation of the event stream.

EVNTIS	Meaning
0b0	The CNTKCTL_EL1.EVNTI field applies to CNTVCT_EL0[15:0].
0b1	The CNTKCTL_EL1.EVNTI field applies to CNTVCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVVCT, bit [16]

When FEAT_ECV is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1NVPCT, bit [15]

When FEAT_ECV is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVCT, bit [14]

When FEAT_ECV is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1TVT, bit [13]

When FEAT_ECV is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ECV, bit [12]

When FEAT_ECV is implemented and FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1PTEN, bit [11]

When FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL1PCTEN, bit [10]

When FEAT_NV2p1 is implemented:

Reserved for software use in nested virtualization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EL0PTEN, bit [9]

Traps EL0 accesses to the physical timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, the following registers are trapped and reported using EC syndrome value `0x18`:
  - CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped and reported using EC syndrome value `0x04`:
  - CNTP_CTL, CNTP_CVAL, CNTP_TVAL.

EL0PTEN	Meaning
`0b0`	EL0 accesses to the physical timer registers are trapped to EL1.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Traps EL0 accesses to the virtual timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped and reported using EC syndrome value `0x18`:
  - CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value `0x03`, MRRC and MCRR accesses are trapped using EC syndrome value `0x04`:
  - CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.

EL0VTEN	Meaning
`0b0`	EL0 accesses to the virtual timer registers are trapped.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit of CNTVCT_EL0, as seen from EL1, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTKCTL_EL1.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTVCT_EL0.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTVCT_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTVCT_EL0 trigger bit, as seen from EL1 and defined by EVNTI, generates an event when the event stream is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

When FEAT_VHE is not implemented, or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, enables the generation of an event stream from CNTVCT_EL0 as seen from EL1.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this control does not enable the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VCTEN, bit [1]

Traps EL0 accesses to the frequency register and virtual counter registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped and reported using EC syndrome value 0x18:
  - CNTVCT_EL0.
  - CNTVCTSS_EL0.
  - If CNTKCTL_EL1.EL0PCTEN is 0, CNTFRQ_EL0.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value 0x03, MRRC and MCRR accesses are trapped and reported using EC syndrome value 0x04:
  - CNTVCT and if CNTKCTL_EL1.EL0PCTEN is 0, CNTFRQ.

EL0VCTEN	Meaning
0b0	EL0 accesses to the frequency register and virtual counter registers are trapped.
0b1	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0PCTEN, bit [0]

Traps EL0 accesses to the frequency register and physical counter register to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, the following registers are trapped and reported using EC syndrome value `0x18`:
  - CNTPCT_EL0
  - CNTPCTSS_EL0
  - If CNTKCTL_EL1.EL0VCTEN is 0, CNTFRQ_EL0.
- In AArch32 state, MCR or MRC accesses the following registers are trapped and reported using EC syndrome value `0x03`, MCRR or MRRC accesses are trapped and reported using EC syndrome value `0x04`:
  - CNTPCT and if CNTKCTL_EL1.EL0VCTEN is 0, CNTFRQ.

EL0PCTEN	Meaning
`0b0`	EL0 accesses to the frequency register and physical counter register are trapped.
`0b1`	This control does not cause any instructions to be trapped.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTKCTL_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTKCTL_EL1` or `CNTKCTL_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTKCTL_EL1
```

op0	op1	CRn	CRm	op2
`0b11`	`0b000`	`0b1110`	`0b0001`	`0b000`

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    X{64}(t) = CNTKCTL_EL1();
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTHCTL_EL2_VHE(CNTHCTL_EL2());
    else
        X{64}(t) = CNTKCTL_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTKCTL_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSR CNTKCTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1() = X{64}(t);
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTHCTL_EL2() = CNTHCTL_EL2_VHE(X{64}(t));
    else
        CNTKCTL_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTKCTL_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTKCTL_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTKCTL_EL1();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTKCTL_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CNTKCTL_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTKCTL_EL1() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTKCTL_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.16 CNTP_CTL_EL0, Counter-timer Physical Timer Control Register

The CNTP_CTL_EL0 characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

AArch64 System register CNTP_CTL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTP_CTL[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTP_CTL_EL0 are UNDEFINED.

Attributes

CNTP_CTL_EL0 is a 64-bit register.

Field descriptions

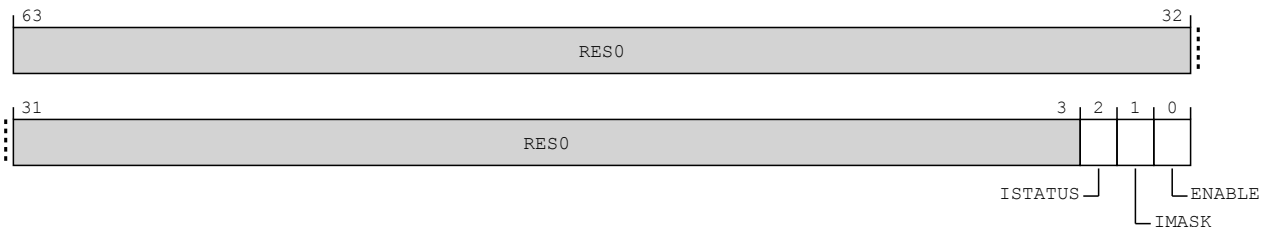

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL_EL0 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CTL_EL0

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTP_CTL_EL0` or `CNTP_CTL_EL02` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTP_CTL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CTL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CTL_EL2();
    else
        X{64}(t) = CNTP_CTL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x180);
    else
        X{64}(t) = CNTP_CTL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CTL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CTL_EL2();
    else
        X{64}(t) = CNTP_CTL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTP_CTL_EL0();
end;
```

```
MSR CNTP_CTL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            CNTHPS_CTL_EL2() = X{64}(t);
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            CNTHP_CTL_EL2() = X{64}(t);
        else
            CNTP_CTL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            NVMem(0x180) = X{64}(t);
        else
            CNTP_CTL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            CNTHPS_CTL_EL2() = X{64}(t);
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            CNTHP_CTL_EL2() = X{64}(t);
        else
            CNTP_CTL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL3 then
        CNTP_CTL_EL0() = X{64}(t);
    end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTP_CTL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVPCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            X{64}(t) = NVMem(0x180);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTP_CTL_EL0();
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTP_CTL_EL0();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CNTP_CTL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVPCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            NVMem(0x180) = X{64}(t);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTP_CTL_EL0() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTP_CTL_EL0() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.17 CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue Register

The CNTP_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configuration

AArch64 System register CNTP_CVAL_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTP_CVAL[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTP_CVAL_EL0 are UNDEFINED.

Attributes

CNTP_CVAL_EL0 is a 64-bit register.

Field descriptions

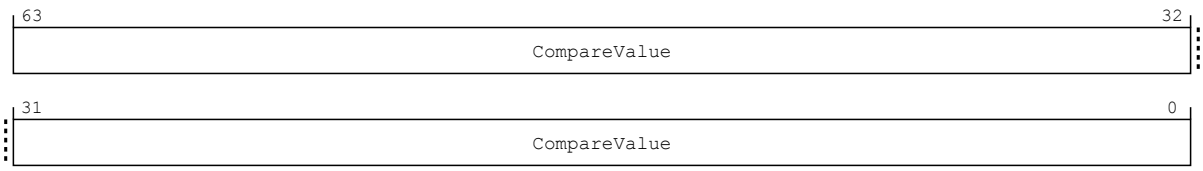

CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL_EL0.ENABLE is 1, and TimerConditionMet is TRUE for the EL1 physical timer, the timer condition is met and all of the following are true:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTP_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL_EL0

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTP_CVAL_EL0` or `CNTP_CVAL_EL02` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTP_CVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CVAL_EL2();
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x178);
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHP_CVAL_EL2();
    else
        X{64}(t) = CNTP_CVAL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTP_CVAL_EL0();
end;
```

```
MSR CNTP_CVAL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x178) = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = X{64}(t);
    else
        CNTP_CVAL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTP_CVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVPCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            X{64}(t) = NVMem(0x178);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTP_CVAL_EL0();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTP_CVAL_EL0();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CNTP_CVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVPCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            NVMem(0x178) = X{64}(t);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTP_CVAL_EL0() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTP_CVAL_EL0() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.18 CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue Register

The CNTP_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

AArch64 System register CNTP_TVAL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTP_TVAL[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTP_TVAL_EL0 are UNDEFINED.

Attributes

CNTP_TVAL_EL0 is a 64-bit register.

Field descriptions

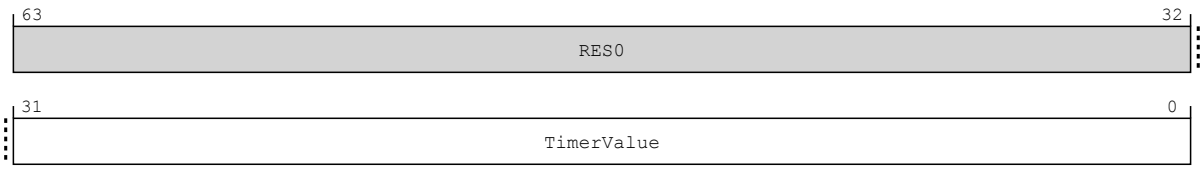

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL_EL0.ENABLE is 1, the value returned is (CNTP_CVAL_EL0 - CNTPCT_EL0).

On a write of this register, CNTP_CVAL_EL0 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTP_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTP_CTL_EL0.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

Note

The value of CNTPCT_EL0 used in these calculations is the value seen at the Exception level that the CNTPCT_EL0 register is being read or written from.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTP_TVAL_EL0

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTP_TVAL_EL0` or `CNTP_TVAL_EL02` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTP_TVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
     ↪'1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
     ↪'1') && CNTHCTL_EL2().ECV == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    else
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if CNTP_CTL_EL0().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
    end;
end;
```

```
MSR CNTP_TVAL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
         ↪'1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
        CNTP_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTPOFF_EL2();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
         ↪'1') && CNTHCTL_EL2().ECV == '1' then
        CNTP_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTPOFF_EL2();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    else
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTP_TVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if CNTP_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTP_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CNTP_TVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTP_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.19 CNTPCT_EL0, Counter-timer Physical Count Register

The CNTPCT_EL0 characteristics are:

Purpose

Reads of CNTPCT_EL0 return the 64-bit physical count value minus a physical offset.

Configuration

All reads to the CNTPCT_EL0 occur in program order relative to reads to CNTPCTSS_EL0 or CNTPCT_EL0.

AArch64 System register CNTPCT_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTPCT[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTPCT_EL0 are UNDEFINED.

Attributes

CNTPCT_EL0 is a 64-bit register.

Field descriptions

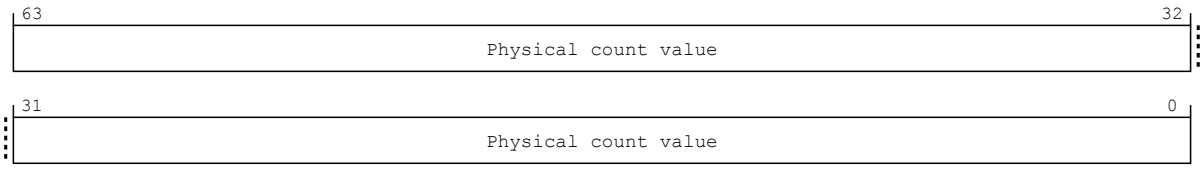

PhysicalCount, bits [63:0]

Physical count value.

If the access is not trapped and all of the following are true, then reads of CNTPCT_EL0 from EL0 or EL1 return (PhysicalCountInt<63:0> - CNTPOFF_EL2<63:0>):

- EL2 is implemented and enabled in the current Security state.
- CNTHCTL_EL2.ECV is 1.
- SCR_EL3.ECVEn is 1.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise, reads of CNTPCT_EL0 return PhysicalCountInt<63:0>.

PhysicalCountInt is defined by 'The Physical Counter'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTPCT_EL0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PCTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
            ↪'1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
            X{64}(t) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
            ↪'1') && CNTHCTL_EL2().ECV == '1' then
            X{64}(t) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    X{64}(t) = PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.20 CNTPCTSS_EL0, Counter-timer Self-Synchronized Physical Count Register

The CNTPCTSS_EL0 characteristics are:

Purpose

Holds the self-synchronized view of the 64-bit physical count value.

Configuration

All reads to the CNTPCTSS_EL0 occur in program order relative to reads to CNTPCT_EL0 or CNTPCTSS_EL0.

This register is a view of the CNTPCT_EL0 register for which reads appear to occur in program order relative to other instructions, without the need for any explicit synchronization. Reads of this register return a value consistent with the counter not being read until the read instruction is known to be non-speculative.

AArch64 System register CNTPCTSS_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTPCTSS[63:0].

This register is present only when FEAT_ECV is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTPCTSS_EL0 are UNDEFINED.

Attributes

CNTPCTSS_EL0 is a 64-bit register.

Field descriptions

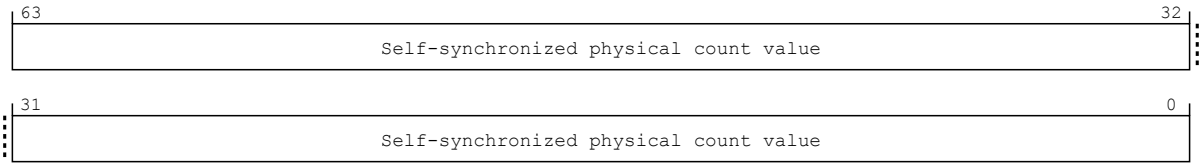

Physical count value.

If the access is not trapped and all of the following are true, then reads of CNTPCTSS_EL0 from EL0 or EL1 return (PhysicalCountInt<63:0> - CNTPOFF_EL2<63:0>):

- EL2 is implemented and enabled in the current Security state.
- CNTHCTL_EL2.ECV is 1.
- SCR_EL3.ECVEn is 1.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise, reads of CNTPCTSS_EL0 return PhysicalCountInt<63:0>.

PhysicalCountInt is defined by 'The Physical Counter'.

SSPhysicalCount, bits [63:0]

Self-synchronized physical count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCTSS_EL0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTPCTSS_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b101

```
if !(IsFeatureImplemented(FEAT_ECV) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0PCTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif EL2Enabled() && !ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) && HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
            ↪'1') && CNTHCTL_EL2().ECV == '1' && !ELIsInHost(EL0) then
            X{64}(t) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().ECVEn ==
            ↪'1') && CNTHCTL_EL2().ECV == '1' then
            X{64}(t) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    X{64}(t) = PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.21 CNTPOFF_EL2, Counter-timer Physical Offset Register

The CNTPOFF_EL2 characteristics are:

Purpose

Holds the 64-bit physical offset. This is the offset for the AArch64 EL1 physical timer and counter when Enhanced Counter Virtualization is enabled.

Configuration

The physical offset applies to:

- Direct reads of CNTPCT_EL0 from EL0 or EL1.
- Indirect reads of the physical counter by the EL1 physical timer. See 'The Generic Timer in AArch64 state'.
- Indirect reads of the physical counter for timestamps generated by profiling logic. See 'The Statistical Profiling Extension' and 'AArch64 Self-hosted Trace'.

The physical offset only applies under conditions described by the relevant sections.

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_ECV_POFF is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTPOFF_EL2 are UNDEFINED.

Attributes

CNTPOFF_EL2 is a 64-bit register.

Field descriptions

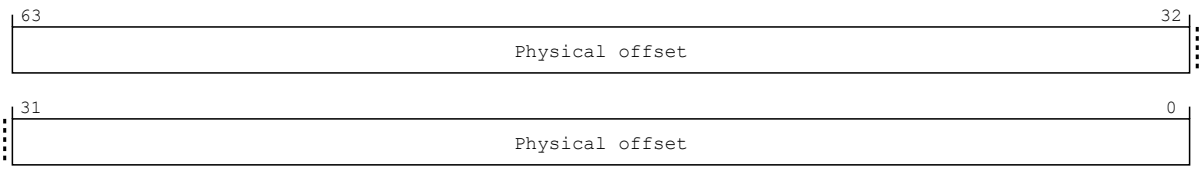

PO, bits [63:0]

Physical offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPOFF_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTPOFF_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_ECV_POFF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x1A8);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ECVEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().ECVEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = CNTPOFF_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTPOFF_EL2();
end;
```

```
MSR CNTPOFF_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_ECV_POFF) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x1A8) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ECVEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().ECVEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        CNTPOFF_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTPOFF_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.22 CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control Register

The CNTPS_CTL_EL1 characteristics are:

Purpose

Control register for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTPS_CTL_EL1 are UNDEFINED.

Attributes

CNTPS_CTL_EL1 is a 64-bit register.

Field descriptions

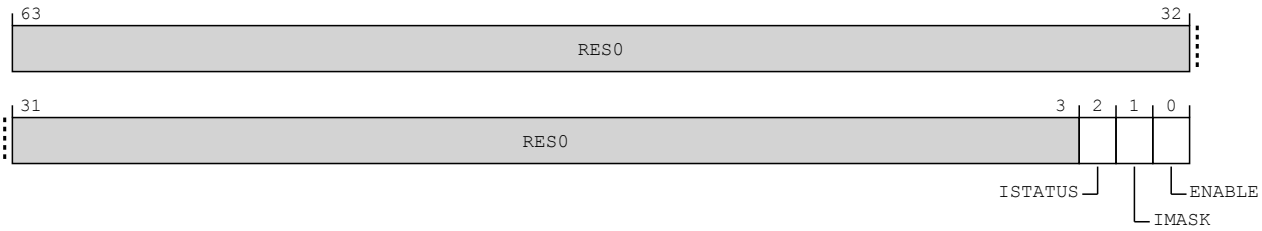

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTPS_TVAL_EL1 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPS_CTL_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTPS_CTL_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ST == '0' then
            Undefined();
        elsif SCR_EL3().EEL2 == '1' then
            Undefined();
        elsif SCR_EL3().ST == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = CNTPS_CTL_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTPS_CTL_EL1();
end;
```

```
MSR CNTPS_CTL_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b001

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ST == '0' then
            Undefined();
        elsif SCR_EL3().EEL2 == '1' then
            Undefined();
        elsif SCR_EL3().ST == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            CNTPS_CTL_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    CNTPS_CTL_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.23 CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue Register

The CNTPS_CVAL_EL1 characteristics are:

Purpose

Holds the compare value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTPS_CVAL_EL1 are UNDEFINED.

Attributes

CNTPS_CVAL_EL1 is a 64-bit register.

Field descriptions

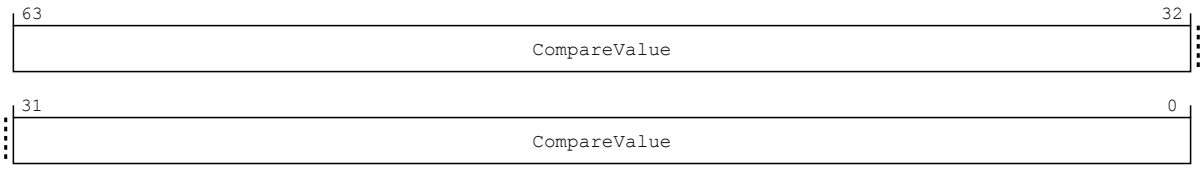

CompareValue, bits [63:0]

Holds the secure physical timer CompareValue.

When CNTPS_CTL_EL1.ENABLE is 1, and TimerConditionMet is TRUE for the EL1 secure physical timer, the timer condition is met and all of the following are true:

- CNTPS_CTL_EL1.ISTATUS is set to 1.
- If CNTPS_CTL_EL1.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTPS_CTL_EL1.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPS_CVAL_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTPS_CVAL_EL1
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ST == '0' then
            Undefined();
        elsif SCR_EL3().EEL2 == '1' then
            Undefined();
        elsif SCR_EL3().ST == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = CNTPS_CVAL_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTPS_CVAL_EL1();
end;
```

MSR CNTPS_CVAL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b010

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ST == '0' then
            Undefined();
        elsif SCR_EL3().EEL2 == '1' then
            Undefined();
        elsif SCR_EL3().ST == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            CNTPS_CVAL_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    CNTPS_CVAL_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.24 CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue Register

The CNTPS_TVAL_EL1 characteristics are:

Purpose

Holds the timer value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

Configuration

This register is present only when EL3 is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTPS_TVAL_EL1 are UNDEFINED.

Attributes

CNTPS_TVAL_EL1 is a 64-bit register.

Field descriptions

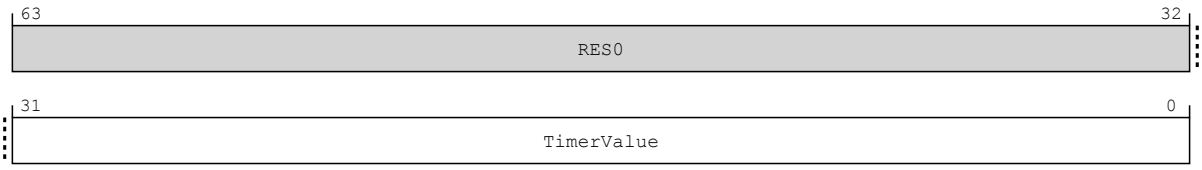

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the secure physical timer.

On a read of this register:

- If CNTPS_CTL_EL1.ENABLE is 0, the value returned is UNKNOWN.
- If CNTPS_CTL_EL1.ENABLE is 1, the value returned is (CNTPS_CVAL_EL1 - CNTPCT_EL0).

On a write of this register, CNTPS_CVAL_EL1 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTPS_CTL_EL1.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTPS_CVAL_EL1) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTPS_CTL_EL1.ISTATUS is set to 1.
- If CNTPS_CTL_EL1.IMASK is 0, an interrupt is generated.

When CNTPS_CTL_EL1.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPS_TVAL_EL1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTPS_TVAL_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ST == '0' then
            Undefined();
        elsif SCR_EL3().EEL2 == '1' then
            Undefined();
        elsif SCR_EL3().ST == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            if CNTPS_CTL_EL1().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTPS_CVAL_EL1() - PhysicalCountInt())[31:0]);
            end;
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CNTPS_CTL_EL1().ENABLE == '0' then
        X{64}(t) = ARBITRARY:bits(64);
    else
        X{64}(t) = ZeroExtend{64}((CNTPS_CVAL_EL1() - PhysicalCountInt())[31:0]);
    end;
end;
```

```
MSR CNTPS_TVAL_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b111	0b1110	0b0010	0b000

```
if !(HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && SCR_EL3().NS == '0' then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().ST == '0' then
            Undefined();
        elsif SCR_EL3().EEL2 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        elsif SCR_EL3().ST == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            CNTPS_CVAL_EL1() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    CNTPS_CVAL_EL1() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.25 CNTV_CTL_EL0, Counter-timer Virtual Timer Control Register

The CNTV_CTL_EL0 characteristics are:

Purpose

Control register for the virtual timer.

Configuration

AArch64 System register CNTV_CTL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTV_CTL[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTV_CTL_EL0 are UNDEFINED.

Attributes

CNTV_CTL_EL0 is a 64-bit register.

Field descriptions

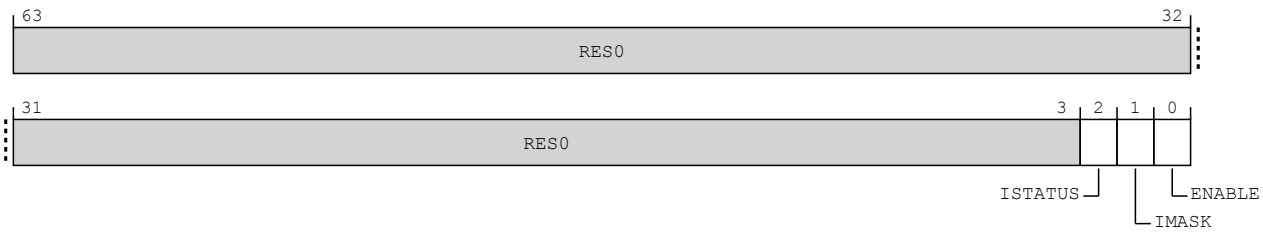

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL_EL0 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CTL_EL0

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTV_CTL_EL0` or `CNTV_CTL_EL02` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTV_CTL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
         ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CTL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CTL_EL2();
    else
        X{64}(t) = CNTV_CTL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x170);
    else
        X{64}(t) = CNTV_CTL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CTL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CTL_EL2();
    else
        X{64}(t) = CNTV_CTL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTV_CTL_EL0();
end;
```

```
MSR CNTV_CTL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
         ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2() = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x170) = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = X{64}(t);
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2() = X{64}(t);
    else
        CNTV_CTL_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    CNTV_CTL_EL0() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTV_CTL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVVCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            X{64}(t) = NVMem(0x170);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTV_CTL_EL0();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTV_CTL_EL0();
    else
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

When FEAT_VHE is implemented MSR CNTV_CTL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVVCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            NVMem(0x170) = X{64}(t);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTV_CTL_EL0() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTV_CTL_EL0() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.26 CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue Register

The CNTV_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the virtual timer.

Configuration

AArch64 System register CNTV_CVAL_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTV_CVAL[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTV_CVAL_EL0 are UNDEFINED.

Attributes

CNTV_CVAL_EL0 is a 64-bit register.

Field descriptions

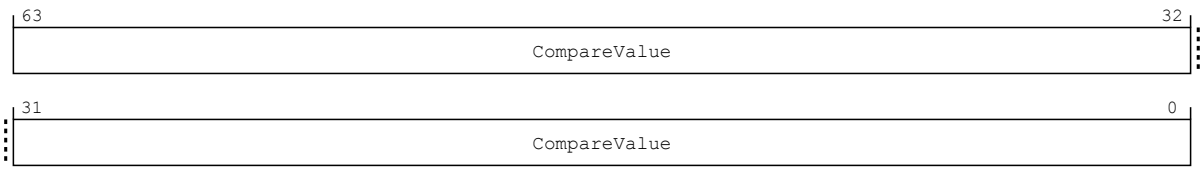

CompareValue, bits [63:0]

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL_EL0.ENABLE is 1, and TimerConditionMet is TRUE for the EL1 virtual timer, the timer condition is met and all of the following are true:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTV_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL_EL0

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTV_CVAL_EL0` or `CNTV_CVAL_EL02` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTV_CVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CVAL_EL2();
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x168);
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        X{64}(t) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        X{64}(t) = CNTHV_CVAL_EL2();
    else
        X{64}(t) = CNTV_CVAL_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTV_CVAL_EL0();
end;
```

```
MSR CNTV_CVAL_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
             ↪'1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            CNTHVS_CVAL_EL2() = X{64}(t);
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            CNTHV_CVAL_EL2() = X{64}(t);
        else
            CNTV_CVAL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        elsif EffectiveHCR_EL2_NVx() IN {'111'} then
            NVMem(0x168) = X{64}(t);
        else
            CNTV_CVAL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            CNTHVS_CVAL_EL2() = X{64}(t);
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            CNTHV_CVAL_EL2() = X{64}(t);
        else
            CNTV_CVAL_EL0() = X{64}(t);
        end;
    elsif PSTATE.EL == EL3 then
        CNTV_CVAL_EL0() = X{64}(t);
    end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTV_CVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b010

```
    if !IsFeatureImplemented(FEAT_AA64) then
        Undefined();
    elsif PSTATE.EL == EL0 then
        Undefined();
    elsif PSTATE.EL == EL1 then
        if EffectiveHCR_EL2_NVx() == '101' then
            if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVVCT == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                X{64}(t) = NVMem(0x168);
            end;
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X{64}(t) = CNTV_CVAL_EL0();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = CNTV_CVAL_EL0();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR CNTV_CVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        if EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1NVVCT == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            NVMem(0x168) = X{64}(t);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = X{64}(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.27 CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue Register

The CNTV_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the EL1 virtual timer.

Configuration

AArch64 System register CNTV_TVAL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTV_TVAL[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTV_TVAL_EL0 are UNDEFINED.

Attributes

CNTV_TVAL_EL0 is a 64-bit register.

Field descriptions

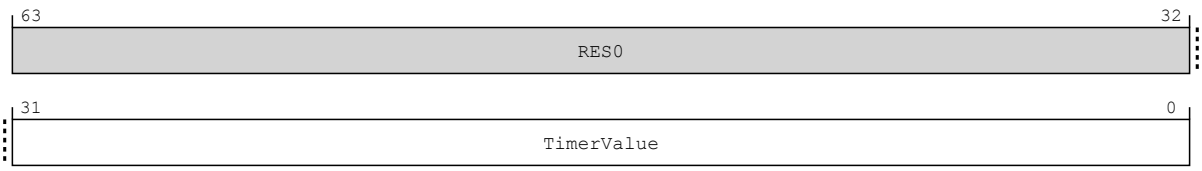

Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL1 virtual timer.

On a read of this register:

- If CNTV_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL_EL0.ENABLE is 1, the value returned is (CNTV_CVAL_EL0 - CNTVCT_EL0).

On a write of this register, CNTV_CVAL_EL0 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTV_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTV_CTL_EL0.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTV_TVAL_EL0

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `CNTV_TVAL_EL0` or `CNTV_TVAL_EL02` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTV_TVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
     ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHV_CTL_EL2().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
        end;
    elsif HaveEL(EL2) && !ELIsInHost(EL0) then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        end;
    else
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            if CNTHVS_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
            if CNTHV_CTL_EL2().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0]);
            end;
        elsif !ELIsInHost(EL2) then
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
            end;
        else
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
            end;
        end;
    elsif PSTATE.EL == EL3 then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0]);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() - CNTVOFF()))[31:0]);
        else
            X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - PhysicalCountInt())[31:0]);
        end;
    end;
```

MSR CNTV_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT ==
        ↪'1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif HaveEL(EL2) && !ELIsInHost(EL0) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif ELIsInHost(EL2) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    elsif !ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF();
    else
        CNTV_CVAL_EL0() = SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt();
    end;
end;
```

When FEAT_VHE is implemented MRS <Xt>, CNTV_TVAL_EL02

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if CNTV_CTL_EL0().ENABLE == '0' then
            X{64}(t) = ARBITRARY:bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() -
                    ↪CNTVOFF_EL2()))[31:0]);
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            if CNTV_CTL_EL0().ENABLE == '0' then
                X{64}(t) = ARBITRARY:bits(64);
            else
                X{64}(t) = ZeroExtend{64}((CNTV_CVAL_EL0() - (PhysicalCountInt() -
                    ↪CNTVOFF_EL2()))[31:0]);
            end;
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR CNTV_TVAL_EL02, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        CNTV_CVAL_EL0() = (SignExtend{64}(X{64}(t)[31:0]) + PhysicalCountInt()) - CNTVOFF_EL2();
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.28 CNTVCT_EL0, Counter-timer Virtual Count Register

The CNTVCT_EL0 characteristics are:

Purpose

Reads of CNTVCT_EL0 return the 64-bit physical count value minus a virtual offset.

Configuration

All reads to the CNTVCT_EL0 occur in program order relative to reads to CNTVCTSS_EL0 or CNTVCT_EL0.

AArch64 System register CNTVCT_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTVCT[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTVCT_EL0 are UNDEFINED.

Attributes

CNTVCT_EL0 is a 64-bit register.

Field descriptions

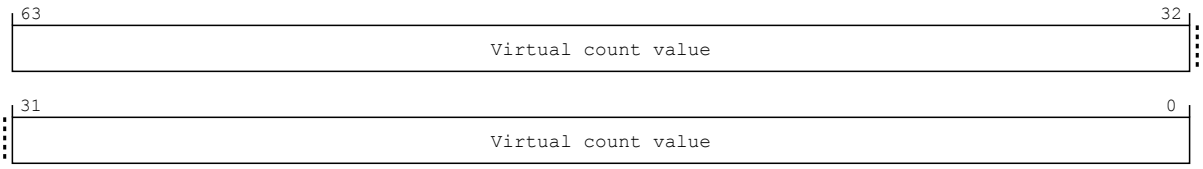

VirtualCount, bits [63:0]

Virtual count value.

When EL2 is implemented, if the access is not trapped, and any of the following are true, then reads of CNTVCT_EL0 from EL0 and EL1 return (PhysicalCountInt<63:0> - CNTVOFF_EL2<63:0>):

- All of the following are true:
  - The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
  - CNTVCT_EL0 is read from EL0.
- All of the following are true:
  - The Effective value of HCR_EL2.E2H is 0.
  - CNTVCT_EL0 is read from EL2.
- CNTVCT_EL0 is read from EL1 or EL3.

Otherwise reads of CNTVCT_EL0 return PhysicalCountInt<63:0>.

PhysicalCountInt is defined by 'The Physical Counter'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT_EL0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTVCT_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VCTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if HaveEL(EL2) && (!EL2Enabled() || !ELIsInHost(EL0)) then
            X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && !ELUsingAArch32(EL2) &&
             ↪(!EL2Enabled() || !ELIsInHost(EL0)) then
            X{64}(t) = PhysicalCountInt() - CNTVOFF();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if HaveEL(EL2) then
            X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL2) && !ELIsInHost(EL2) then
        X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
    else
        X{64}(t) = PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        X{64}(t) = PhysicalCountInt() - CNTVOFF();
    else
        X{64}(t) = PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.29 CNTVCTSS_EL0, Counter-timer Self-Synchronized Virtual Count Register

The CNTVCTSS_EL0 characteristics are:

Purpose

Reads of CNTVCTSS_EL0 return the 64-bit physical count value minus a virtual offset.

Configuration

All reads to the CNTVCTSS_EL0 occur in program order relative to reads to CNTVCT_EL0 or CNTVCTSS_EL0.

This register is a view of the CNTVCT_EL0 register for which reads appear to occur in program order relative to other instructions, without the need for any explicit synchronization. Reads of this register return a value consistent with the counter not being read until the read instruction is known to be non-speculative.

AArch64 System register CNTVCTSS_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTVCTSS[63:0].

This register is present only when FEAT_ECV is implemented and FEAT_AA64 is implemented. Otherwise, direct accesses to CNTVCTSS_EL0 are UNDEFINED.

Attributes

CNTVCTSS_EL0 is a 64-bit register.

Field descriptions

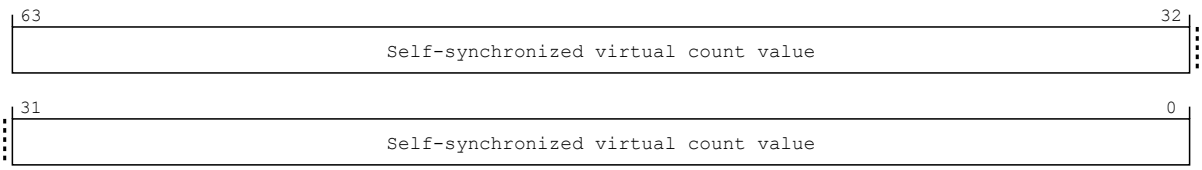

SSVirtualCount, bits [63:0]

Self-synchronized virtual count value.

When EL2 is implemented, if the access is not trapped, and any of the following are true, then reads of CNTVCTSS_EL0 from EL0 and EL1 return (PhysicalCountInt<63:0> - CNTVOFF_EL2<63:0>):

- All of the following are true:
  - The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
  - CNTVCTSS_EL0 is read from EL0.
- All of the following are true:
  - The Effective value of HCR_EL2.E2H is 0.
  - CNTVCTSS_EL0 is read from EL2.
- CNTVCTSS_EL0 is read from EL1 or EL3.

Otherwise reads of CNTVCTSS_EL0 return PhysicalCountInt<63:0>.

PhysicalCountInt is defined by 'The Physical Counter'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCTSS_EL0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CNTVCTSS_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b110

```
if !(IsFeatureImplemented(FEAT_ECV) && IsFeatureImplemented(FEAT_AA64)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1().EL0VCTEN == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            AArch64_SystemAccessTrap(EL1, 0x18);
        end;
    elsif ELIsInHost(EL0) && CNTHCTL_EL2().EL0VCTEN == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && !ELIsInHost(EL0) && CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if HaveEL(EL2) && (!EL2Enabled() || !ELIsInHost(EL0)) then
            X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && !ELUsingAArch32(EL2) &&
             ↪(!EL2Enabled() || !ELIsInHost(EL0)) then
            X{64}(t) = PhysicalCountInt() - CNTVOFF();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        if HaveEL(EL2) then
            X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
        else
            X{64}(t) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL2) && !ELIsInHost(EL2) then
        X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
    else
        X{64}(t) = PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        X{64}(t) = PhysicalCountInt() - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        X{64}(t) = PhysicalCountInt() - CNTVOFF();
    else
        X{64}(t) = PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.10.30 CNTVOFF_EL2, Counter-timer Virtual Offset Register

The CNTVOFF_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset for the AArch64 virtual timers and counters.

Configuration

The virtual offset applies to:

- Direct reads of CNTVCT_EL0.
- Indirect reads of the virtual counter by the EL1 virtual timer. See 'The Generic Timer in AArch64 state'.
- Indirect reads of the virtual counter for timestamps generated by profiling logic. See 'The Statistical Profiling Extension' and 'AArch64 Self-hosted Trace'.

The virtual offset only applies under conditions described by the relevant sections.

AArch64 System register CNTVOFF_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTVOFF[63:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct accesses to CNTVOFF_EL2 are UNDEFINED.

Attributes

CNTVOFF_EL2 is a 64-bit register.

Field descriptions

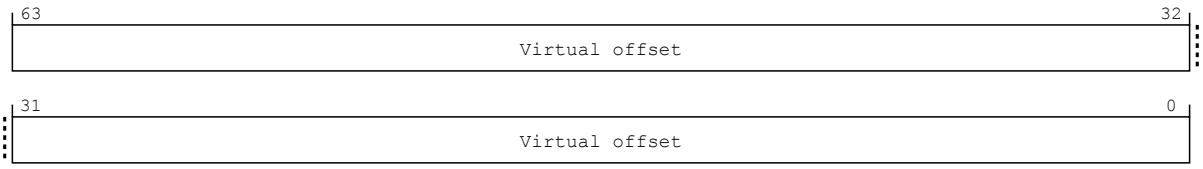

VOffset, bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTVOFF_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x060);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    X{64}(t) = CNTVOFF_EL2();
elsif PSTATE.EL == EL3 then
    X{64}(t) = CNTVOFF_EL2();
end;
```

MSR CNTVOFF_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA64) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x060) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    CNTVOFF_EL2() = X{64}(t);
elsif PSTATE.EL == EL3 then
    CNTVOFF_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11 Guarded Control Stack registers

This section lists the Guarded Control Stack registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.1 GCSCR_EL1, Guarded Control Stack Control Register (EL1)

The GCSCR_EL1 characteristics are:

Purpose

Controls the Guarded Control Stack at EL1.

Configuration

This register is present only when FEAT_GCS is implemented. Otherwise, direct accesses to GCSCR_EL1 are UNDEFINED.

Attributes

GCSCR_EL1 is a 64-bit register.

Field descriptions

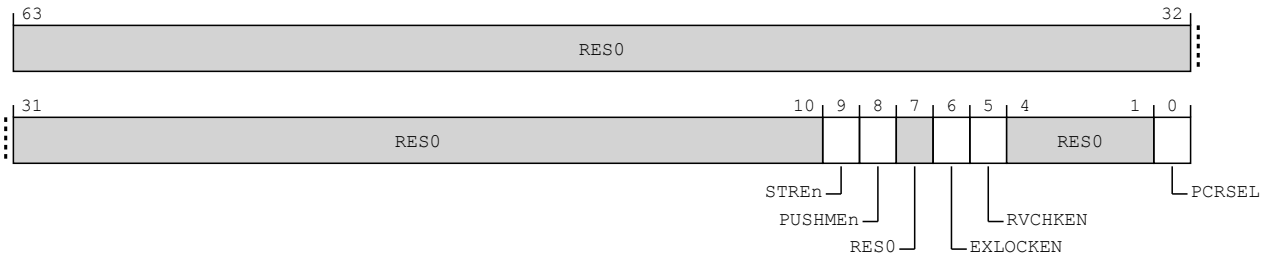

Bits [63:10]

Reserved, RES0.

STREn, bit [9]

Execution of the following instructions are trapped:

- GCSSTR.
- GCSSTTR if any of the following are true.
  - PSTATE.UAO is 1.
  - If EL2 is implemented and enabled in the current Security state and the Effective value of HCR_EL2.{NV,NV1} is {1,1}.

STREn	Meaning
0b0	Execution of any of the specified instructions at EL1 causes a GCS exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

PUSHMEn, bit [8]

Trap GCSPUSHM instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PUSHMEn	Meaning
0b0	Execution of a GCSPUSHM instruction at EL1 causes a Trap exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [7]

Reserved, RES0.

EXLOCKEN, bit [6]

Exception state lock.

Prevents MSR instructions from writing to ELR_EL1 or SPSR_EL1.

EXLOCKEN	Meaning
0b0	EL1 exception state locking disabled.
0b1	EL1 exception state locking enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

RVCHKEN, bit [5]

Return value check enable.

RVCHKEN	Meaning
0b0	Return value checking disabled at EL1.
0b1	Return value checking enabled at EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:1]

Reserved, RES0.

PCRSEL, bit [0]

Guarded Control Stack procedure call return enable selection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCRSEL	Meaning
0b0	Guarded Control Stack at EL1 is not PCR Selected.
0b1	Guarded Control Stack at EL1 is PCR Selected.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing GCSCR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name GCSCR_EL1 or GCSCR_EL12 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8D0);
    else
        X{64}(t) = GCSCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = GCSCR_EL2();
    else
        X{64}(t) = GCSCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSCR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR GCSCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8D0) = X{64}(t);
    else
        GCSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        GCSCR_EL2() = X{64}(t);
    else
        GCSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSCR_EL1() = X{64}(t);
end;
```

```
When FEAT_VHE is implemented MRS <Xt>, GCSCR_EL12
```

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            X{64}(t) = NVMem(0x8D0);
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
                Undefined();
            elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                X{64}(t) = GCSCR_EL1();
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = GCSCR_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR GCSCR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x8D0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            GCSCR_EL1() = X{64}(t);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        GCSCR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.2 GCSCR_EL2, Guarded Control Stack Control Register (EL2)

The GCSCR_EL2 characteristics are:

Purpose

Controls the Guarded Control Stack at EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_GCS is implemented. Otherwise, direct accesses to GCSCR_EL2 are UNDEFINED.

Attributes

GCSCR_EL2 is a 64-bit register.

Field descriptions

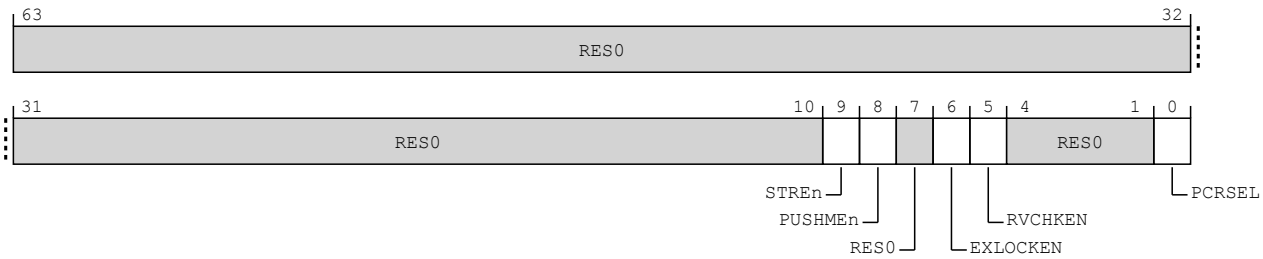

Bits [63:10]

Reserved, RES0.

STREn, bit [9]

Execution of the following instructions are trapped:

- GCSSTR.
- GCSSTTR if any of the following are true.
  - The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
  - The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and PSTATE.UAO is 1.

STREn	Meaning
0b0	Execution of any of the specified instructions at EL2 cause a GCS exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

PUSHMEn, bit [8]

Trap GCSPUSHM instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PUSHMEn	Meaning
0b0	Execution of a GCSPUSHM instruction at EL2 causes a Trap exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [7]

Reserved, RES0.

EXLOCKEN, bit [6]

Exception state lock.

Prevents MSR instructions from writing to ELR_EL2 or SPSR_EL2.

EXLOCKEN	Meaning
0b0	EL2 exception state locking disabled.
0b1	EL2 exception state locking enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

RVCHKEN, bit [5]

Return value check enable.

RVCHKEN	Meaning
0b0	Return value checking disabled at EL2.
0b1	Return value checking enabled at EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:1]

Reserved, RES0.

PCRSEL, bit [0]

Guarded Control Stack procedure call return enable selection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCRSEL	Meaning
0b0	Guarded Control Stack at EL2 is not PCR Selected.
0b1	Guarded Control Stack at EL2 is PCR Selected.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing GCSCR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name GCSCR_EL2 or GCSCR_EL1 are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSCR_EL2();
end;
```

```
MSR GCSCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0101	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCSCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSCR_EL2() = X{64}(t);
end;
```

MRS <Xt>, GCSCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8D0);
    else
        X{64}(t) = GCSCR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = GCSCR_EL2();
    else
        X{64}(t) = GCSCR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSCR_EL1();
end;
```

MSR GCSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8D0) = X{64}(t);
    else
        GCSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        GCSCR_EL2() = X{64}(t);
    else
        GCSCR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSCR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.3 GCSCR_EL3, Guarded Control Stack Control Register (EL3)

The GCSCR_EL3 characteristics are:

Purpose

Controls the Guarded Control Stack at EL3.

Configuration

This register is present only when FEAT_GCS is implemented and EL3 is implemented. Otherwise, direct accesses to GCSCR_EL3 are UNDEFINED.

Attributes

GCSCR_EL3 is a 64-bit register.

Field descriptions

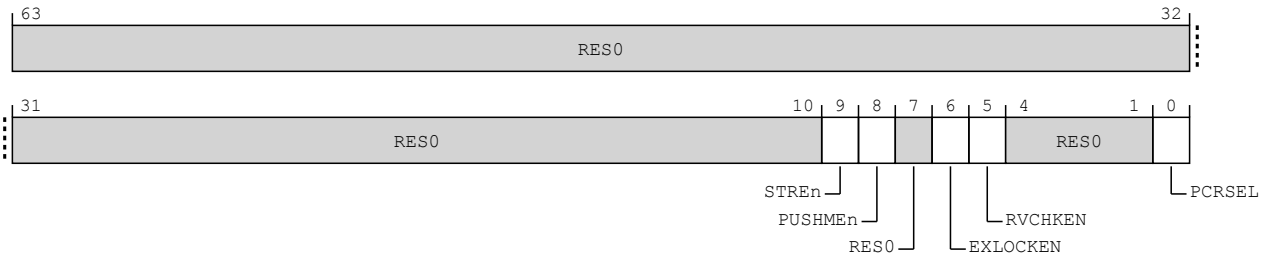

Bits [63:10]

Reserved, RES0.

STREn, bit [9]

Execution of the following instructions are trapped:

- GCSSTR.
- GCSSTTR.

STREn	Meaning
0b0	Execution of any of the specified instructions at EL3 cause a GCS exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

PUSHMEn, bit [8]

Trap GCSPUSHM instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PUSHMEn	Meaning
0b0	Execution of a GCSPUSHM instruction at EL3 causes a Trap exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [7]

Reserved, RES0.

EXLOCKEN, bit [6]

Exception state lock.

Prevents MSR instructions from writing to ELR_EL3 or SPSR_EL3.

EXLOCKEN	Meaning
0b0	EL3 exception state locking disabled.
0b1	EL3 exception state locking enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

RVCHKEN, bit [5]

Return value check enable.

RVCHKEN	Meaning
0b0	Return value checking disabled at EL3.
0b1	Return value checking enabled at EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:1]

Reserved, RES0.

PCRSEL, bit [0]

Guarded Control Stack procedure call return enable selection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCRSEL	Meaning
0b0	Guarded Control Stack at EL3 is not PCR Selected.
0b1	Guarded Control Stack at EL3 is PCR Selected.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing GCSCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSCR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0101	0b000

```
if !(IsFeatureImplemented(FEAT_GCS) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSCR_EL3();
end;
```

```
MSR GCSCR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0101	0b000

```
if !(IsFeatureImplemented(FEAT_GCS) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().GCSCR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        GCSCR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.4 GCSCRE0_EL1, Guarded Control Stack Control Register (EL0)

The GCSCRE0_EL1 characteristics are:

Purpose

Controls the Guarded Control Stack at EL0.

Configuration

This register is present only when FEAT_GCS is implemented. Otherwise, direct accesses to GCSCRE0_EL1 are UNDEFINED.

Attributes

GCSCRE0_EL1 is a 64-bit register.

Field descriptions

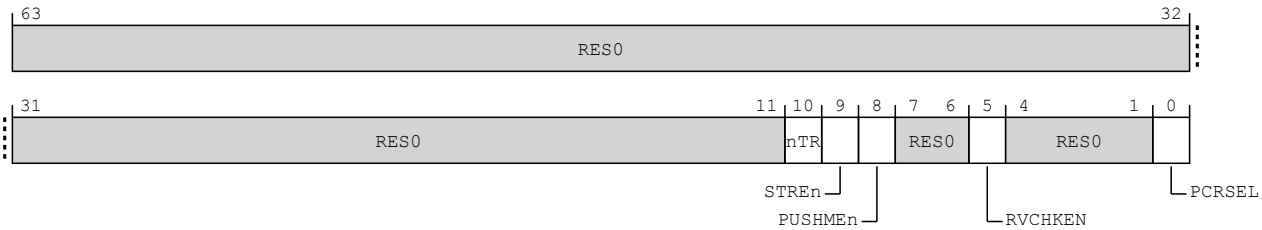

Bits [63:11]

Reserved, RES0.

nTR, bit [10]

Trap GCS register accesses from EL0.

nTR	Meaning
0b0	Read accesses to GCSPR_EL0 at EL0 cause a Trap exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

STREn, bit [9]

Execution of the following instructions are trapped:

- GCSSTR.
- GCSSTTR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STREn	Meaning
0b0	Execution of any of the specified instructions at EL0 cause a GCS exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

PUSHMEn, bit [8]

Trap GCSPUSHM instruction.

PUSHMEn	Meaning
0b0	Execution of a GCSPUSHM instruction at EL0 causes a Trap exception.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [7:6]

Reserved, RES0.

RVCHKEN, bit [5]

Return value check enable.

RVCHKEN	Meaning
0b0	Return value checking disabled at EL0.
0b1	Return value checking enabled at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:1]

Reserved, RES0.

PCRSEL, bit [0]

Guarded Control Stack procedure call return enable selection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCRSEL	Meaning
0b0	Guarded Control Stack at EL0 is not PCR Selected.
0b1	Guarded Control Stack at EL0 is PCR Selected.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing GCSCRE0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSCRE0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nGCS_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSCRE0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSCRE0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSCRE0_EL1();
end;
```

```
MSR GCSCRE0_EL1, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nGCS_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCSCRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCSCRE0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSCRE0_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.5 GCSPR_EL0, Guarded Control Stack Pointer Register (EL0)

The GCSPR_EL0 characteristics are:

Purpose

Contains the Guarded Control Stack Pointer at EL0.

Configuration

This register is present only when FEAT_GCS is implemented. Otherwise, direct accesses to GCSPR_EL0 are UNDEFINED.

Attributes

GCSPR_EL0 is a 64-bit register.

Field descriptions

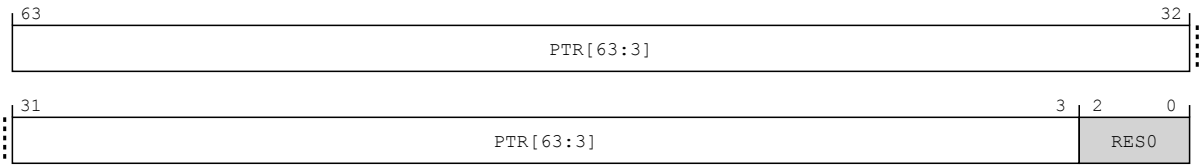

PTR[63:3], bits [63:3]

EL0 Guarded Control Stack Pointer bits [63:3].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [2:0]

Reserved, RES0.

Accessing GCSPR_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSPR_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif (!EL2Enabled() || HCR_EL2().TGE != '1') && GCSCRE0_EL1().nTR == '0' then
        AArch64_SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2().TGE == '1' && GCSCRE0_EL1().nTR == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && !ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) ||
         ↪SCR_EL3().FGTEn == '1') && HFGRTR_EL2().nGCS_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSPR_EL0();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nGCS_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSPR_EL0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSPR_EL0();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSPR_EL0();
end;
```

```
MSR GCSPR_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nGCS_EL0 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCSPR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCSPR_EL0() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSPR_EL0() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.6 GCSPR_EL1, Guarded Control Stack Pointer Register (EL1)

The GCSPR_EL1 characteristics are:

Purpose

Contains the Guarded Control Stack Pointer at EL1.

Configuration

This register is present only when FEAT_GCS is implemented. Otherwise, direct accesses to GCSPR_EL1 are UNDEFINED.

Attributes

GCSPR_EL1 is a 64-bit register.

Field descriptions

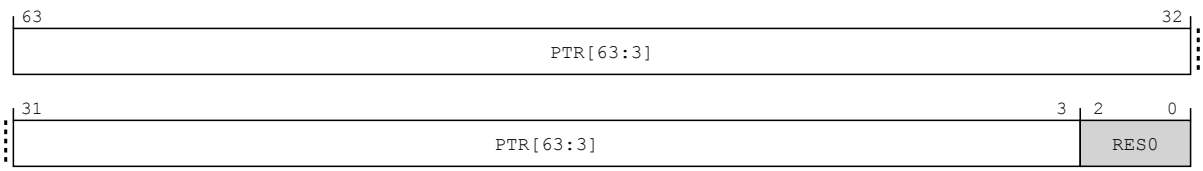

PTR[63:3], bits [63:3]

EL1 Guarded Control Stack Pointer bits [63:3].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [2:0]

Reserved, RES0.

Accessing GCSPR_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `GCSPR_EL1` or `GCSPR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSPR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8C0);
    else
        X{64}(t) = GCSPR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = GCSPR_EL2();
    else
        X{64}(t) = GCSPR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSPR_EL1();
end;
```

MSR GCSPR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8C0) = X{64}(t);
    else
        GCSPR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        GCSPR_EL2() = X{64}(t);
    else
        GCSPR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSPR_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, GCSPR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x8C0);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = GCSPR_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = GCSPR_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR GCSPR_EL12, <Xt>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b101	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x8C0) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
            Undefined();
        elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            GCSPR_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        GCSPR_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.7 GCSPR_EL2, Guarded Control Stack Pointer Register (EL2)

The GCSPR_EL2 characteristics are:

Purpose

Contains the Guarded Control Stack Pointer at EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register is present only when FEAT_GCS is implemented. Otherwise, direct accesses to GCSPR_EL2 are UNDEFINED.

Attributes

GCSPR_EL2 is a 64-bit register.

Field descriptions

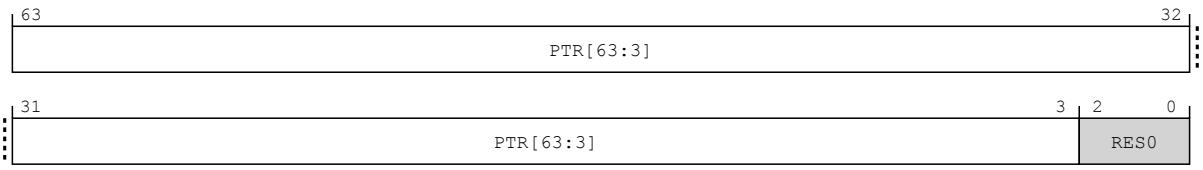

PTR[63:3], bits [63:3]

EL2 Guarded Control Stack Pointer bits [63:3].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [2:0]

Reserved, RES0.

Accessing GCSPR_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `GCSPR_EL2` or `GCSPR_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSPR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0101	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = GCSPR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSPR_EL2();
end;
```

MSR GCSPR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        GCSPR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSPR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, GCSPR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGRTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x8C0);
    else
        X{64}(t) = GCSPR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = GCSPR_EL2();
    else
        X{64}(t) = GCSPR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSPR_EL1();
end;
```

```
MSR GCSPR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_GCS) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
         ↪HFGWTR_EL2().nGCS_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x8C0) = X{64}(t);
    else
        GCSPR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && SCR_EL3().GCSEn == '0' then
        Undefined();
    elsif HaveEL(EL3) && SCR_EL3().GCSEn == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        GCSPR_EL2() = X{64}(t);
    else
        GCSPR_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    GCSPR_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.11.8 GCSPR_EL3, Guarded Control Stack Pointer Register (EL3)

The GCSPR_EL3 characteristics are:

Purpose

Contains the Guarded Control Stack Pointer at EL3.

Configuration

This register is present only when FEAT_GCS is implemented and EL3 is implemented. Otherwise, direct accesses to GCSPR_EL3 are UNDEFINED.

Attributes

GCSPR_EL3 is a 64-bit register.

Field descriptions

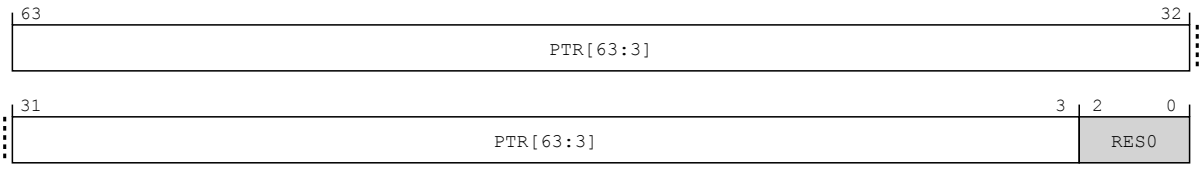

PTR[63:3], bits [63:3]

EL3 Guarded Control Stack Pointer bits [63:3].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [2:0]

Reserved, RES0.

Accessing GCSPR_EL3

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, GCSPR_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_GCS) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = GCSPR_EL3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR GCSPR_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0010	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_GCS) && HaveEL(EL3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().GCSPR_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        GCSPR_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12 MPAM registers

This section lists the MPAM registers in AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.1 MPAM0_EL1, MPAM0 Register (EL1)

The MPAM0_EL1 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests when executing at EL0.

Configuration

If FEAT_MPAM is implemented, EL2 is implemented and enabled in the current Security state, and the MPAM virtualization option is present, then:

- If MPAMHCR_EL2.GSTAPP_PLK == 1 and the Effective value of HCR_EL2.TGE is 0, MPAM1_EL1 is used instead of MPAM0_EL1 to generate MPAM information to label memory requests.
- If the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1} and MPAMHCR_EL2.EL0_VPMEN == 1, then MPAM PARTIDs in MPAM0_EL1 are virtual and mapped into physical PARTIDs for the current Security state.

This register is present only when FEAT_MPAM is implemented. Otherwise, direct accesses to MPAM0_EL1 are UNDEFINED.

Attributes

MPAM0_EL1 is a 64-bit register.

Field descriptions

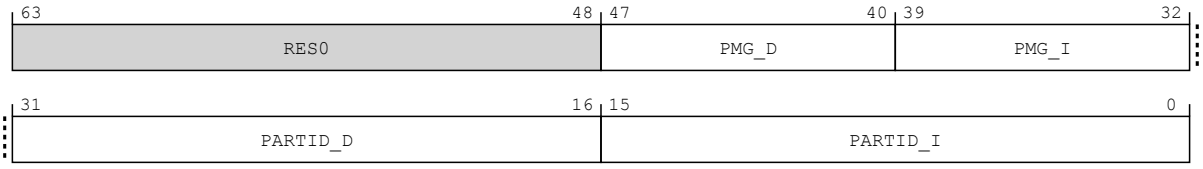

Bits [63:48]

Reserved, RES0.

PMG_D, bits [47:40]

Performance monitoring group property for PARTID_D.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PMG_I, bits [39:32]

Performance monitoring group property for PARTID_I.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_D, bits [31:16]

Partition ID for data accesses, including load and store accesses, made from EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PARTID_I, bits [15:0]

Partition ID for instruction accesses made from EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAM0_EL1

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAM0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAM2_EL2().TRAPMPAM0EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = MPAM0_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAM0_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAM0_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAM0_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
        ↪&& MPAM2_EL2().TRAPMPAM0EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        MPAM0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAM0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAM0_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.2 MPAM1_EL1, MPAM1 Register (EL1)

The MPAM1_EL1 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests when executing at EL1.

Configuration

If FEAT_MPAMv1p0 or FEAT_MPAMv0p1 is implemented, EL2 is implemented and enabled in the current Security state, and the MPAM virtualization option is present, then:

- If MPAMHCR_EL2.GSTAPP_PLK == 1 and the Effective value of HCR_EL2.TGE is 0, MPAM1_EL1 is used instead of MPAM0_EL1 to generate MPAM labels for memory requests when executing at EL0.
- If MPAMHCR_EL2.EL1_VPMEN == 1, MPAM PARTIDs in MPAM1_EL1 are virtual and mapped into physical PARTIDs for the current Security state. This mapping of MPAM1_EL1 virtual PARTIDs to physical PARTIDs when EL1_VPMEN is 1 also applies when MPAM1_EL1 is used at EL0 due to MPAMHCR_EL2.GSTAPP_PLK.

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented) and EL3 is implemented, AArch64 System register MPAM1_EL1 bit [63] is architecturally mapped to AArch64 System register MPAM3_EL3[63].

When ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (!HaveEL(EL3))) && HaveEL(EL2), AArch64 System register MPAM1_EL1 bit [63] is architecturally mapped to AArch64 System register MPAM2_EL2[63].

This register is present only when FEAT_MPAM is implemented. Otherwise, direct accesses to MPAM1_EL1 are UNDEFINED.

Attributes

MPAM1_EL1 is a 64-bit register.

Field descriptions

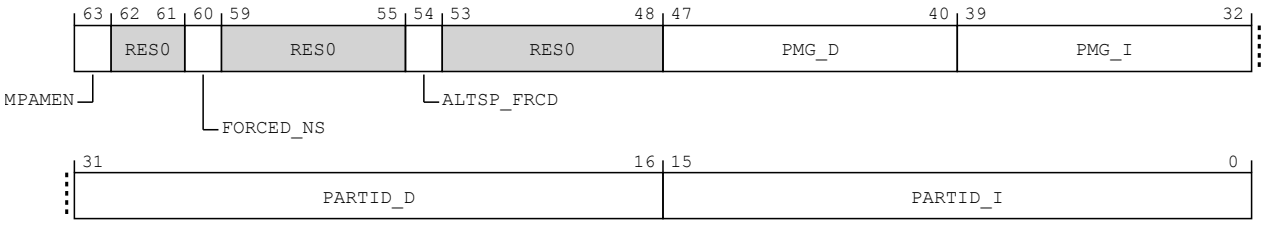

MPAMEN, bit [63]

MPAM Enable. MPAM is enabled when MPAMEN == 1. When disabled, all PARTIDs and PMGs are output as their default value in the corresponding ID space.

MPAMEN	Meaning
0b0	The default PARTID and default PMG are output in MPAM information.
0b1	MPAM information is output based on the MPAMn_ELx register for ELn according the MPAM configuration.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - MPAM3_EL3.MPAMEN == ‘0’
- Access to this field is RAO/WI if all of the following are true:
  - EL3 is implemented
  - MPAM3_EL3.MPAMEN == ‘1’
- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is not implemented
  - EL2 is implemented
  - MPAM2_EL2.MPAMEN == ‘0’
- Access to this field is RAO/WI if all of the following are true:
  - EL3 is not implemented
  - EL2 is implemented
  - MPAM2_EL2.MPAMEN == ‘1’

Bits [62:61]

Reserved, RES0.

FORCED_NS, bit [60]

When FEAT_MPAMv0p1 is implemented:

In the Secure state, FORCED_NS indicates the state of MPAM3_EL3.FORCE_NS.

FORCED_NS	Meaning
`0b0`	In the Non-secure state, always reads as 0. In the Secure state, indicates that MPAM3_EL3.FORCE_NS == 0.
`0b1`	In the Secure state, indicates that MPAM3_EL3.FORCE_NS == 1.

Always reads as 0 in the Non-secure state.

Writes are ignored.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [59:55]

Reserved, RES0.

ALTSP_FRCD, bit [54]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Alternative PARTID forced for PARTIDs in this register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ALTSP_FRCD	Meaning
0b0	The PARTIDs in MPAM1_EL1 and MPAM0_EL1 are using the primary PARTID space.
0b1	The PARTIDs in MPAM1_EL1 and MPAM0_EL1 are using the alternative PARTID space.

This bit indicates that a higher Exception level has forced the PARTIDs in this register to use the alternative PARTID space defined for the current Security state.

In MPAM1_EL1, it also indicates that MPAM0_EL1 is forced to use alternative PARTID space.

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

Accessing this field has the following behavior:

- When !UsePrimarySpaceEL10(), access to this field is RAO/WI.
- Otherwise, access to this field is RAZ/WI.

Otherwise:

Reserved, RES0.

Bits [53:48]

Reserved, RES0.

PMG_D, bits [47:40]

Performance monitoring group property for PARTID_D.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PMG_I, bits [39:32]

Performance monitoring group property for PARTID_I.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_D, bits [31:16]

Partition ID for data accesses, including load and store accesses, made from EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_I, bits [15:0]

Partition ID for instruction accesses made from EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAM1_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `MPAM1_EL1` or `MPAM1_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, MPAM1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
        ↪&& MPAM2_EL2().TRAPMPAM1EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x900);
    else
        X{64}(t) = MPAM1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = MPAM2_EL2();
    else
        X{64}(t) = MPAM1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAM1_EL1();
end;
```

```
MSR MPAM1_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAM2_EL2().TRAPMPAM1EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x900) = X{64}(t);
    else
        MPAM1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        MPAM2_EL2() = X{64}(t);
    else
        MPAM1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAM1_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, MPAM1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x900);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            Undefined();
        elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = MPAM1_EL1();
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        X{64}(t) = MPAM1_EL1();
    else
        Undefined();
    end;
end;
```

When FEAT_VHE is implemented MSR MPAM1_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x900) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            Undefined();
        elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            MPAM1_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        MPAM1_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.3 MPAM2_EL2, MPAM2 Register (EL2)

The MPAM2_EL2 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests when executing at EL2.

Configuration

This register has no effect on execution at EL1 and EL0, if EL2 is not enabled in the current Security state.

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented) and EL3 is implemented, AArch64 System register MPAM2_EL2 bit [63] is architecturally mapped to AArch64 System register MPAM3_EL3[63].

When FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented, AArch64 System register MPAM2_EL2 bit [63] is architecturally mapped to AArch64 System register MPAM1_EL1[63].

This register is present only when FEAT_MPAM is implemented. Otherwise, direct accesses to MPAM2_EL2 are UNDEFINED.

Attributes

MPAM2_EL2 is a 64-bit register.

Field descriptions

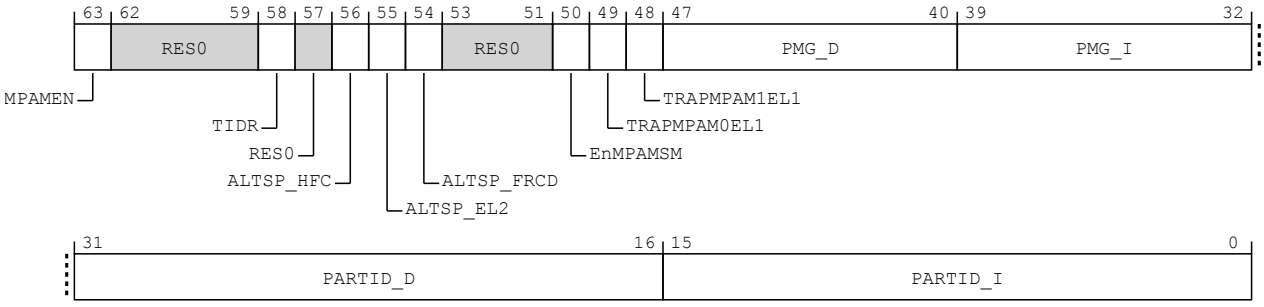

MPAMEN, bit [63]

MPAM Enable. MPAM is enabled when MPAMEN == 1. When disabled, all PARTIDs and PMGs are output as their default value in the corresponding ID space.

MPAMEN	Meaning
0b0	The default PARTID and default PMG are output in MPAM information from all Exception levels.
0b1	MPAM information is output based on the MPAMn_ELx register for ELn according to the MPAM configuration.

If EL3 is implemented, this field is read-only and reads the current value of the read/write MPAM3_EL3.MPAMEN bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - EL3 is implemented
  - MPAM3_EL3.MPAMEN == '0'
- Access to this field is RAO/WI if all of the following are true:
  - EL3 is implemented
  - MPAM3_EL3.MPAMEN == '1'
- When EL3 is not implemented, access to this field is RW.

Bits [62:59]

Reserved, RES0.

TIDR, bit [58]

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p1 is implemented) and MPAMIDR_EL1.HAS_TIDR == '1':

TIDR traps accesses to MPAMIDR_EL1 from EL1 to EL2.

TIDR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Trap accesses to MPAMIDR_EL1 from EL1 to EL2.

MPAMHCR_EL2.TRAP_MPAMIDR_EL1 == 1 also traps MPAMIDR_EL1 accesses from EL1 to EL2. If either TIDR or TRAP_MPAMIDR_EL1 are 1, accesses are trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [57]

Reserved, RES0.

ALTSP_HFC, bit [56]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Hierarchical force of alternative PARTID space controls. When MPAM3_EL3.ALTSP_HEN is 0, ALTSP controls in MPAM2_EL2 have no effect. When MPAM3_EL3.ALTSP_HEN is 1, this bit selects whether the PARTIDs in MPAM1_EL1 and MPAM0_EL1 are in the primary (0) or alternative (1) PARTID space for the security state.

ALTSP_HFC	Meaning
0b0	When MPAM3_EL3.ALTSP_HEN is 1, the PARTID space of MPAM1_EL1.PARTID_I, MPAM1_EL1.PARTID_D, MPAM0_EL1.PARTID_I, and MPAM0_EL1.PARTID_D are in the primary PARTID space for the Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ALTSP_HFC	Meaning
0b1	When MPAM3_EL3.ALTSP_HEN is 1, the PARTID space of MPAM1_EL1.PARTID_I, MPAM1_EL1.PARTID_D, MPAM0_EL1.PARTID_I, and MPAM0_EL1.PARTID_D are in the alternative PARTID space for the Security state.

This control has no effect when MPAM3_EL3.ALTSP_HEN is 0.

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ALTSP_EL2, bit [55]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Select alternative PARTID space for PARTIDs in MPAM2_EL2 when MPAM3_EL3.ALTSP_HEN is 1.

ALTSP_EL2	Meaning
0b0	When MPAM3_EL3.ALTSP_HEN is 1, selects the primary PARTID space for MPAM2_EL2.PARTID_I and MPAM2_EL2.PARTID_D.
0b1	When MPAM3_EL3.ALTSP_HEN is 1, selects the alternative PARTID space for MPAM2_EL2.PARTID_I and MPAM2_EL2.PARTID_D.

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ALTSP_FRCD, bit [54]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Alternative PARTID forced for PARTIDs in this register.

ALTSP_FRCD	Meaning
0b0	The PARTIDs in this register are using the primary PARTID space.
0b1	The PARTIDs in this register are using the alternative PARTID space.

This bit indicates that a higher Exception level has forced the PARTIDs in this register to use the alternative PARTID space defined for the current Security state. In EL2, it is also 1 when MPAM2_EL2.ALTSP_EL2 is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When !UsePrimarySpaceEL2(), access to this field is RAO/WI.
- Otherwise, access to this field is RAZ/WI.

Otherwise:

Reserved, RES0.

Bits [53:51]

Reserved, RES0.

EnMPAMSM, bit [50]

When FEAT_SME is implemented:

Traps execution at EL1 of instructions that directly access the MPAMSM_EL1 register to EL2. The exception is reported using ESR_ELx.EC syndrome value `0x18`.

EnMPAMSM	Meaning
`0b0`	This control causes execution of these instructions at EL1 to be trapped.
`0b1`	This control does not cause execution of any instructions to be trapped.

This field has no effect on accesses to MPAMSM_EL1 from EL2 or EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRAPMPAM0EL1, bit [49]

Trap accesses from EL1 to the MPAM0_EL1 register trap to EL2.

TRAPMPAM0EL1	Meaning
`0b0`	Accesses to MPAM0_EL1 from EL1 are not trapped.
`0b1`	Accesses to MPAM0_EL1 from EL1 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to `'1'`.
  - When EL3 is implemented, this field resets to an architecturally UNKNOWN value.

TRAPMPAM1EL1, bit [48]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trap accesses from EL1 to the MPAM1_EL1 register trap to EL2.

TRAPMPAM1EL1	Meaning
`0b0`	Accesses to MPAM1_EL1 from EL1 are not trapped.
`0b1`	Accesses to MPAM1_EL1 from EL1 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to `'1'`.
  - When EL3 is implemented, this field resets to an architecturally UNKNOWN value.

PMG_D, bits [47:40]

Performance monitoring group for data accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PMG_I, bits [39:32]

Performance monitoring group for instruction accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_D, bits [31:16]

Partition ID for data accesses, including load and store accesses, made from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_I, bits [15:0]

Partition ID for instruction accesses made from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAM2_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `MPAM2_EL2` or `MPAM1_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAM2_EL2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAM2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAM2_EL2();
end;
```

MSR MPAM2_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAM2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAM2_EL2() = X{64}(t);
end;
```

MRS <Xt>, MPAM1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAM2_EL2().TRAPMPAM1EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x900);
    else
        X{64}(t) = MPAM1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = MPAM2_EL2();
    else
        X{64}(t) = MPAM1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAM1_EL1();
end;
```

```
MSR MPAM1_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
        ↪&& MPAM2_EL2().TRAPMPAM1EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x900) = X{64}(t);
    else
        MPAM1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        MPAM2_EL2() = X{64}(t);
    else
        MPAM1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAM1_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.4 MPAM3_EL3, MPAM3 Register (EL3)

The MPAM3_EL3 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests when executing at EL3.

Configuration

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented) and EL2 is implemented, AArch64 System register MPAM3_EL3 bit [63] is architecturally mapped to AArch64 System register MPAM2_EL2[63].

When FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented, AArch64 System register MPAM3_EL3 bit [63] is architecturally mapped to AArch64 System register MPAM1_EL1[63].

This register is present only when FEAT_MPAM is implemented. Otherwise, direct accesses to MPAM3_EL3 are UNDEFINED.

Attributes

MPAM3_EL3 is a 64-bit register.

Field descriptions

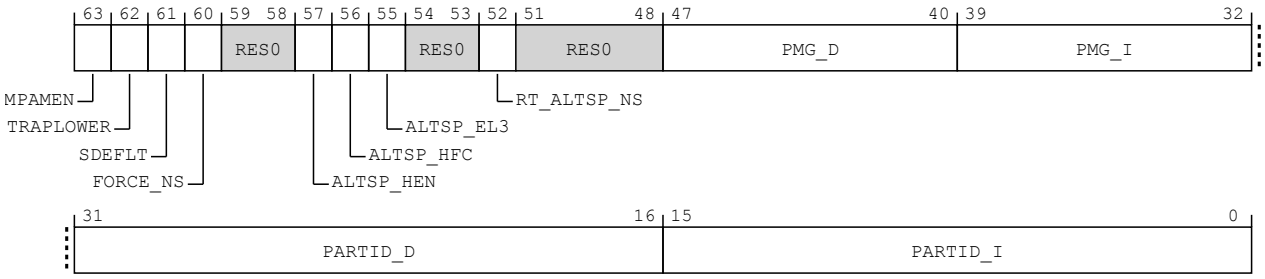

MPAMEN, bit [63]

MPAM Enable. MPAM is enabled when MPAMEN == 1. When disabled, all PARTIDs and PMGs are output as their default value in the corresponding ID space.

Values of this field are:

MPAMEN	Meaning
0b0	The default PARTID and default PMG are output in MPAM information when executing at any ELn.
0b1	MPAM information is output based on the MPAMn_ELx register for ELn according the MPAM configuration.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRAPLOWER, bit [62]

Trap direct accesses to MPAM System registers that are not UNDEFINED from all ELn lower than EL3.

TRAPLOWER	Meaning
`0b0`	Do not force trapping of direct accesses of MPAM System registers to EL3.
`0b1`	Force direct accesses of MPAM System registers to trap to EL3.

Note

This trap is higher priority than any of the traps controlled by MPAM _EL2 registers.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

SDEFLT, bit [61]

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p1 is implemented) and MPAMIDR_EL1.HAS_SDEFLT == '1':

SDEFLT overrides the PARTID and PMG with the default PARTID and default PMG when executing in the Secure state.

SDEFLT	Meaning
`0b0`	The PARTID and PMG are determined normally in the Secure state.
`0b1`	When executing in the Secure state, the PARTID is always PARTID 0, and the PMG is always PMG 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FORCE_NS, bit [60]

When FEAT_MPAMv0p1 is implemented and MPAMIDR_EL1.HAS_FORCE_NS == '1':

FORCE_NS forces MPAM_NS to always be 1 in the Secure state.

FORCE_NS	Meaning
`0b0`	MPAM_NS is 0 when executing in the Secure state.
`0b1`	MPAM_NS is 1 when executing in the Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An implementation is permitted to have this field as RAO if the implementation does not support generating MPAM_NS as 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [59:58]

Reserved, RES0.

ALTSP_HEN, bit [57]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Hierarchical enable for alternative PARTID space controls. Alternative PARTID space controls in MPAM2_EL2 have no effect when this field is zero.

ALTSP_HEN	Meaning
0b0	Disable alternative PARTID space controls in MPAM2_EL2. The PARTID space for PARTIDs in MPAM2_EL2, MPAM1_EL1, and MPAM0_EL1 is selected by MPAM3_EL3.ALTSP_HFC.
0b1	Enable alternative PARTID space controls in MPAM2_EL2 to control the PARTID space used for PARTIDs in MPAM2_EL2, MPAM1_EL1, and MPAM0_EL1.

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ALTSP_HFC, bit [56]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Hierarchical force of alternative PARTID space controls. When MPAM3_EL3.ALTSP_HEN is 0, the PARTID space for PARTIDs in MPAM2_EL2, MPAM1_EL1, and MPAM0_EL1 is selected by the value of this bit.

ALTSP_HFC	Meaning
0b0	When MPAM3_EL3.ALTSP_HEN is 0, the PARTID space of MPAM2_EL2.PARTID, MPAM1_EL1.PARTID and MPAM0_EL1.PARTID are the primary PARTID space for the security state.
0b1	When MPAM3_EL3.ALTSP_HEN is 0, the PARTID space of MPAM2_EL2.PARTID and MPAM1_EL1.PARTID and MPAM0_EL1.PARTID are the alternative PARTID space for the security state.

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ALTSP_EL3, bit [55]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Select alternative PARTID space for PARTIDs in MPAM3_EL3.

ALTSP_EL3	Meaning
0b0	Selects the primary PARTID space of MPAM3_EL3.PARTID_I and MPAM3_EL3.PARTID_D.
0b1	Selects the alternative PARTID space of MPAM3_EL3.PARTID_I and MPAM3_EL3.PARTID_D.

For more information, see In a PE with FEAT_RME, selection of primary or alternative PARTID space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Bits [54:53]

Reserved, RES0.

RT_ALTSP_NS, bit [52]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == '1':

Selects whether the alternative PARTID space for the Root security state is the Secure PARTID space or the Non-secure PARTID space. MPAM3_EL3.RT_ALTSP_NS selects the alternative PARTID space for the Root Security state when MPAM3_EL3.ALTSP_EL3 == 1.

RT_ALTSP_NS	Meaning
0b0	The alternative PARTID space in the Root security state is the Secure PARTID space.
0b1	The alternative PARTID space in the Root security state is the Non-secure PARTID space.

This field has no effect except in the Root security state (EL3).

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Bits [51:48]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMG_D, bits [47:40]

Performance monitoring group for data accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PMG_I, bits [39:32]

Performance monitoring group for instruction accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_D, bits [31:16]

Partition ID for data accesses, including load and store accesses, made from EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_I, bits [15:0]

Partition ID for instruction accesses made from EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAM3_EL3

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAM3_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAM3_EL3();
end;
```

```
MSR MPAM3_EL3, <Xt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_FGWTE3) && FGWTE3_EL3().MPAM3_EL3 == '1' then
        AArch64_SystemAccessTrap(EL3, 0x18);
    else
        MPAM3_EL3() = X{64}(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.5 MPAMBW0_EL1, MPAM PE-side Maximum Bandwidth Control Register (EL0)

The MPAMBW0_EL1 characteristics are:

Purpose

Enables software to configure a maximum fraction of memory bandwidth that the PE is permitted to use when executing at EL0 with its current PARTID.

Configuration

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented. Otherwise, direct accesses to MPAMBW0_EL1 are UNDEFINED.

Attributes

MPAMBW0_EL1 is a 64-bit register.

Field descriptions

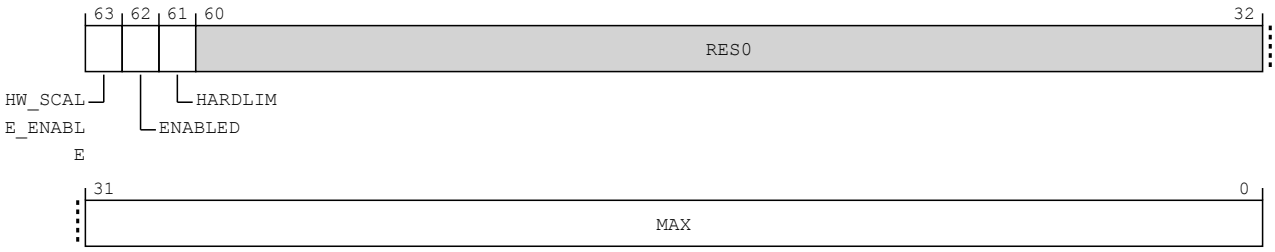

HW_SCALE_ENABLE, bit [63]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1':

Enables hardware bandwidth scaling of the MPAMBW0_EL1.MAX value.

HW_SCALE_ENABLE	Meaning
0b0	PE-side memory bandwidth control hardware scaling at EL0 is disabled.
0b1	PE-side memory bandwidth control hardware scaling at EL0 is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ENABLED, bit [62]

Enables the PE-side memory bandwidth control when at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ENABLED	Meaning
0b0	The PE-side memory bandwidth control at EL0 is disabled.
0b1	The PE-side memory bandwidth control at EL0 is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HARDLIM, bit [61]

PE-side Maximum Bandwidth Limit Behavior Selection.

HARDLIM	Meaning
0b0	Soft limit: when MPAMBW0_EL1.MAX bandwidth is exceeded, the PE is unregulated unless the downstream memory path is saturated. It is IMPLEMENTATION DEFINED how hardware determines when the downstream memory path is saturated.
0b1	Hard limit: when MPAMBW0_EL1.MAX bandwidth is exceeded, the PE does not use any more bandwidth until the memory bandwidth for the PE falls below MPAMBW0_EL1.MAX.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MPAMBWIDR_EL1.MAX_LIM == ‘00’, access to this field is RW.
- When MPAMBWIDR_EL1.MAX_LIM == ‘01’, access to this field is RAZ/WI.
- When MPAMBWIDR_EL1.MAX_LIM == ‘10’, access to this field is RAO/WI.

Bits [60:32]

Reserved, RES0.

MAX, bits [31:0]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1' and MPAMBW0_EL1.HW_SCALE_ENABLE == '1':

31	0
MAX	

MAX, bits [31:0]

Maximum memory bandwidth allocated to the PE when executing at EL0 with its current PARTID.

The value is represented as a multiplier of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [31:16] represent the integer part of the value.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When MPAMBWIDR_EL1.HAS_HW_SCALE == '0' or MPAMBW0_EL1.HW_SCALE_ENABLE == '0':

31 … 16	15 … 0
RES0	MAX

Bits [31:16]

Reserved, RES0.

MAX, bits [15:0]

Maximum memory bandwidth allocated to the PE when executing at EL0 with its current PARTID.

The value is represented as a fraction of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMBW0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMBW0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b101

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBW0_EL1 == '0' then
            AArch64_SystemAccessTrap(EL2, 0x18);
        else
            X{64}(t) = MPAMBW0_EL1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
            Undefined();
        elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
             ↪MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            X{64}(t) = MPAMBW0_EL1();
        end;
    elsif PSTATE.EL == EL3 then
        X{64}(t) = MPAMBW0_EL1();
    end;
```

MSR MPAMBW0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b101

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBW0_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        MPAMBW0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMBW0_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMBW0_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.6 MPAMBW1_EL1, MPAM PE-side Maximum Bandwidth Control Register (EL1)

The MPAMBW1_EL1 characteristics are:

Purpose

Enables software to configure a maximum fraction of memory bandwidth that the PE is permitted to use when executing at EL1 with its current PARTID.

Configuration

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented. Otherwise, direct accesses to MPAMBW1_EL1 are UNDEFINED.

Attributes

MPAMBW1_EL1 is a 64-bit register.

Field descriptions

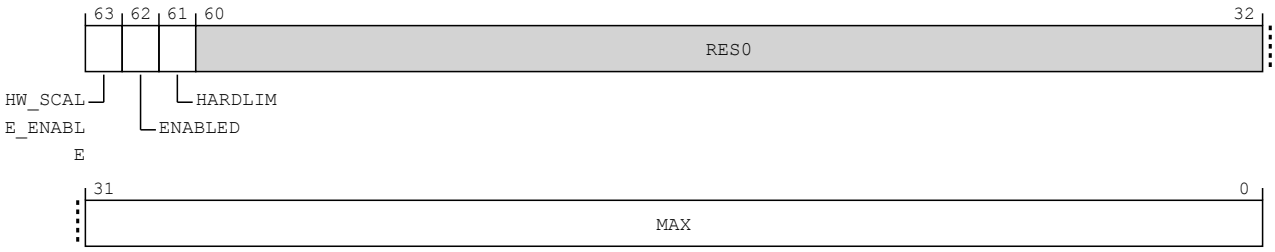

HW_SCALE_ENABLE, bit [63]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1':

Enables hardware bandwidth scaling of the MPAMBW1_EL1.MAX value.

HW_SCALE_ENABLE	Meaning
0b0	PE-side memory bandwidth control hardware scaling at EL1 is disabled.
0b1	PE-side memory bandwidth control hardware scaling at EL1 is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ENABLED, bit [62]

Enables the PE-side memory bandwidth control when at EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ENABLED	Meaning
0b0	The PE-side memory bandwidth control at EL1 is disabled.
0b1	The PE-side memory bandwidth control at EL1 is enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

HARDLIM, bit [61]

PE-side Maximum Bandwidth Limit Behavior Selection.

HARDLIM	Meaning
0b0	Soft limit: when MPAMBW1_EL1.MAX bandwidth is exceeded, the PE is unregulated unless the downstream memory path is saturated. It is IMPLEMENTATION DEFINED how hardware determines when the downstream memory path is saturated.
0b1	Hard limit: when MPAMBW1_EL1.MAX bandwidth is exceeded, the PE does not use any more bandwidth until the memory bandwidth for the PE falls below MPAMBW1_EL1.MAX.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MPAMBWIDR_EL1.MAX_LIM == ‘00’, access to this field is RW.
- When MPAMBWIDR_EL1.MAX_LIM == ‘01’, access to this field is RAZ/WI.
- When MPAMBWIDR_EL1.MAX_LIM == ‘10’, access to this field is RAO/WI.

Bits [60:32]

Reserved, RES0.

MAX, bits [31:0]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1' and MPAMBW1_EL1.HW_SCALE_ENABLE == '1':

31	0
MAX	

MAX, bits [31:0]

Maximum memory bandwidth allocated to the PE when executing at EL1 with its current PARTID.

The value is represented as a multiplier of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [31:16] represent the integer part of the value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When MPAMBWIDR_EL1.HAS_HW_SCALE == '0' or MPAMBW1_EL1.HW_SCALE_ENABLE == '0':

31 … 16	15 … 0
RES0	MAX

Bits [31:16]

Reserved, RES0.

MAX, bits [15:0]

Maximum memory bandwidth allocated to the PE when executing at EL1 with its current PARTID.

The value is represented as a fraction of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMBW1_EL1

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL3 using the accessor name `MPAMBW1_EL1` or `MPAMBW1_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMBW1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBW1_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x908);
    else
        X{64}(t) = MPAMBW1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = MPAMBW2_EL2();
    else
        X{64}(t) = MPAMBW1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMBW1_EL1();
end;
```

MSR MPAMBW1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBW1_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x908) = X{64}(t);
    else
        MPAMBW1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        MPAMBW2_EL2() = X{64}(t);
    else
        MPAMBW1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMBW1_EL1() = X{64}(t);
end;
```

When FEAT_VHE is implemented MRS <Xt>, MPAMBW1_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        X{64}(t) = NVMem(0x908);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
            if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
                 ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                AArch64_SystemAccessTrap(EL2, 0x18);
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        if ELIsInHost(EL2) then
            if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
                 ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
                Undefined();
            elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
                Undefined();
            elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
                 ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
                if EL3SDDUndef() then
                    Undefined();
                else
                    AArch64_SystemAccessTrap(EL3, 0x18);
                end;
            else
                X{64}(t) = MPAMBW1_EL1();
            end;
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL3 then
        if ELIsInHost(EL2) then
            X{64}(t) = MPAMBW1_EL1();
        else
            Undefined();
        end;
    end;
```

When FEAT_VHE is implemented MSR MPAMBW1_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b1010	0b0101	0b100

```
    if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() == '101' then
        NVMem(0x908) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if ELIsInHost(EL2) then
        if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
            Undefined();
        elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
             ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            MPAMBW1_EL1() = X{64}(t);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    if ELIsInHost(EL2) then
        MPAMBW1_EL1() = X{64}(t);
    else
        Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.7 MPAMBW2_EL2, MPAM PE-side Maximum Bandwidth Control Register (EL2)

The MPAMBW2_EL2 characteristics are:

Purpose

Enables software to configure a maximum fraction of memory bandwidth that the PE is permitted to use when executing at EL2 with its current PARTID.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented. Otherwise, direct accesses to MPAMBW2_EL2 are UNDEFINED.

Attributes

MPAMBW2_EL2 is a 64-bit register.

Field descriptions

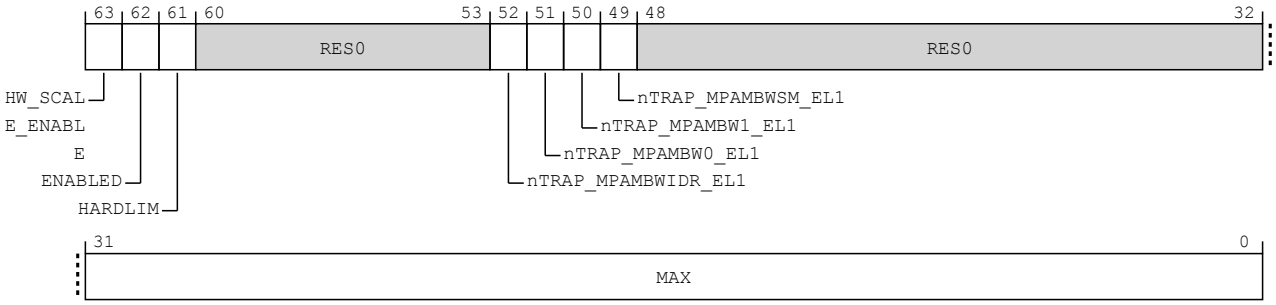

HW_SCALE_ENABLE, bit [63]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1':

Enables hardware bandwidth scaling of the MPAMBW2_EL2.MAX value.

HW_SCALE_ENABLE	Meaning
0b0	PE-side memory bandwidth control hardware scaling at EL2 is disabled.
0b1	PE-side memory bandwidth control hardware scaling at EL2 is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ENABLED, bit [62]

Enables the PE-side memory bandwidth control when at EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ENABLED	Meaning
0b0	The PE-side memory bandwidth control at EL2 is disabled.
0b1	The PE-side memory bandwidth control at EL2 is enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

HARDLIM, bit [61]

PE-side Maximum Bandwidth Limit Behavior Selection.

HARDLIM	Meaning
0b0	Soft limit: when MPAMBW2_EL2.MAX bandwidth is exceeded, the PE is unregulated unless the downstream memory path is saturated. It is IMPLEMENTATION DEFINED how hardware determines when the downstream memory path is saturated.
0b1	Hard limit: when MPAMBW2_EL2.MAX bandwidth is exceeded, the PE does not use any more bandwidth until the memory bandwidth for the PE falls below MPAMBW2_EL2.MAX.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MPAMBWIDR_EL1.MAX_LIM == ‘00’, access to this field is RW.
- When MPAMBWIDR_EL1.MAX_LIM == ‘01’, access to this field is RAZ/WI.
- When MPAMBWIDR_EL1.MAX_LIM == ‘10’, access to this field is RAO/WI.

Bits [60:53]

Reserved, RES0.

nTRAP_MPAMBWIDR_EL1, bit [52]

Traps accesses to MPAMBWIDR_EL1 from EL1 to EL2.

nTRAP_MPAMBWIDR_EL1	Meaning
0b0	Accesses to MPAMBWIDR_EL1 from EL1 are trapped to EL2 with EC syndrome value 0x18.
0b1	Accesses to MPAMBWIDR_EL1 from EL1 are not trapped by this mechanism.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

nTRAP_MPAMBW0_EL1, bit [51]

Traps accesses to MPAMBW0_EL1 from EL1 to EL2.

nTRAP_MPAMBW0_EL1	Meaning
`0b0`	Accesses to MPAMBW0_EL1 from EL1 are trapped to EL2 with EC syndrome value `0x18`.
`0b1`	Accesses to MPAMBW0_EL1 from EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

nTRAP_MPAMBW1_EL1, bit [50]

Traps accesses to MPAMBW1_EL1 from EL1 to EL2.

nTRAP_MPAMBW1_EL1	Meaning
`0b0`	Accesses to MPAMBW1_EL1 from EL1 are trapped to EL2 with EC syndrome value `0x18`.
`0b1`	Accesses to MPAMBW1_EL1 from EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

nTRAP_MPAMBWSM_EL1, bit [49]

When FEAT_SME is implemented:

Traps accesses to MPAMBWSM_EL1 from EL1 to EL2.

nTRAP_MPAMBWSM_EL1	Meaning
`0b0`	Accesses to MPAMBWSM_EL1 from EL1 are trapped to EL2 with EC syndrome value `0x18`.
`0b1`	Accesses to MPAMBWSM_EL1 from EL1 are not trapped by this mechanism.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [48:32]

Reserved, RES0.

MAX, bits [31:0]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1' and MPAMBW2_EL2.HW_SCALE_ENABLE == '1':

31 … 0
MAX

MAX, bits [31:0]

Maximum memory bandwidth allocated to the PE when executing at EL2 with its current PARTID.

The value is represented as a multiplier of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [31:16] represent the integer part of the value.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When MPAMBWIDR_EL1.HAS_HW_SCALE == '0' or MPAMBW2_EL2.HW_SCALE_ENABLE == '0':

31 … 16	15 … 0
RES0	MAX

Bits [31:16]

Reserved, RES0.

MAX, bits [15:0]

Maximum memory bandwidth allocated to the PE when executing at EL2 with its current PARTID.

The value is represented as a fraction of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing MPAMBW2_EL2

When the Effective value of HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the accessor name `MPAMBW2_EL2` or `MPAMBW1_EL1` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMBW2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
            ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMBW2_EL2();
    end;
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X{64}(t) = MPAMBW2_EL2();
end;
```

```
MSR MPAMBW2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
            ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMBW2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMBW2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
When FEAT_VHE is implemented MRS <Xt>, MPAMBW1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBW1_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        X{64}(t) = NVMem(0x908);
    else
        X{64}(t) = MPAMBW1_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        X{64}(t) = MPAMBW2_EL2();
    else
        X{64}(t) = MPAMBW1_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMBW1_EL1();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

When FEAT_VHE is implemented MSR MPAMBW1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBW1_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EffectiveHCR_EL2_NVx() IN {'111'} then
        NVMem(0x908) = X{64}(t);
    else
        MPAMBW1_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif ELIsInHost(EL2) then
        MPAMBW2_EL2() = X{64}(t);
    else
        MPAMBW1_EL1() = X{64}(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL3 then
    MPAMBW1_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.8 MPAMBW3_EL3, MPAM PE-side Maximum Bandwidth Control Register (EL3)

The MPAMBW3_EL3 characteristics are:

Purpose

Enables software to configure a maximum fraction of memory bandwidth that the PE is permitted to use when executing at EL3 with its current PARTID.

Configuration

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented. Otherwise, direct accesses to MPAMBW3_EL3 are UNDEFINED.

Attributes

MPAMBW3_EL3 is a 64-bit register.

Field descriptions

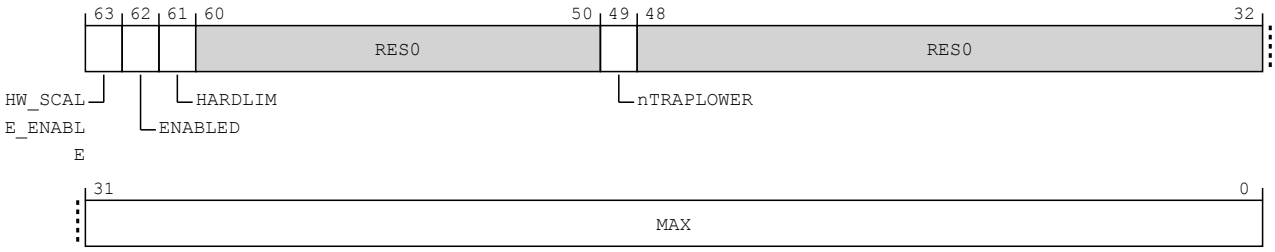

HW_SCALE_ENABLE, bit [63]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1':

Enables hardware bandwidth scaling of the MPAMBW3_EL3.MAX value.

HW_SCALE_ENABLE	Meaning
0b0	PE-side memory bandwidth control hardware scaling at EL3 is disabled.
0b1	PE-side memory bandwidth control hardware scaling at EL3 is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ENABLED, bit [62]

Enables the PE-side memory bandwidth control when at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ENABLED	Meaning
0b0	The PE-side memory bandwidth control at EL3 is disabled.
0b1	The PE-side memory bandwidth control at EL3 is enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

HARDLIM, bit [61]

PE-side Maximum Bandwidth Limit Behavior Selection.

HARDLIM	Meaning
0b0	Soft limit: when MPAMBW3_EL3.MAX bandwidth is exceeded, the PE is unregulated unless the downstream memory path is saturated. It is IMPLEMENTATION DEFINED how hardware determines when the downstream memory path is saturated.
0b1	Hard limit: when MPAMBW3_EL3.MAX bandwidth is exceeded, the PE does not use any more bandwidth until the memory bandwidth for the PE falls below MPAMBW3_EL3.MAX.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MPAMBWIDR_EL1.MAX_LIM == ‘00’, access to this field is RW.
- When MPAMBWIDR_EL1.MAX_LIM == ‘01’, access to this field is RAZ/WI.
- When MPAMBWIDR_EL1.MAX_LIM == ‘10’, access to this field is RAO/WI.

Bits [60:50]

Reserved, RES0.

nTRAPLOWER, bit [49]

Traps accesses to MPAMBW2_EL2, MPAMBWCAP_EL2, MPAMBW1_EL1, MPAMBW0_EL1, MPAMBWSM_EL1, MPAMBWIDR_EL1 from any lower EL to EL3.

nTRAPLOWER	Meaning
0b0	Accesses to MPAMBW2_EL2, MPAMBWCAP_EL2, MPAMBW1_EL1, MPAMBW0_EL1, MPAMBWSM_EL1, MPAMBWIDR_EL1 from EL1 are trapped to EL3 with EC syndrome value 0x18.
0b1	Accesses to MPAMBW2_EL2, MPAMBWCAP_EL2, MPAMBW1_EL1, MPAMBW0_EL1, MPAMBWSM_EL1, MPAMBWIDR_EL1 from EL1 are not trapped by this mechanism.

Note

This trap is higher priority than any of the traps controlled by MPAM _EL2 registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Bits [48:32]

Reserved, RES0.

MAX, bits [31:0]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1' and MPAMBW3_EL3.HW_SCALE_ENABLE == '1':

31 … 0
MAX

MAX, bits [31:0]

Maximum memory bandwidth allocated to the PE when executing at EL3 with its current PARTID.

The value is represented as a multiplier of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [31:16] represent the integer part of the value.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When MPAMBWIDR_EL1.HAS_HW_SCALE == '0' or MPAMBW3_EL3.HW_SCALE_ENABLE == '0':

31 … 16	15 … 0
RES0	MAX

Bits [31:16]

Reserved, RES0.

MAX, bits [15:0]

Maximum memory bandwidth allocated to the PE when executing at EL3 with its current PARTID.

The value is represented as a fraction of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMBW3_EL3

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, MPAMBW3_EL3
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMBW3_EL3();
end;
```

```
MSR MPAMBW3_EL3, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b1010	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    MPAMBW3_EL3() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.9 MPAMBWCAP_EL2, MPAM PE-side Maximum Bandwidth Limit Virtualization Register

The MPAMBWCAP_EL2 characteristics are:

Purpose

Allows software executing at EL2 to provide MPAMBWCAP_EL2.CAP as an upper bound to MPAMBW1_EL1.MAX.

If FEAT_SME is implemented, the upper bound also applies to MPAMBWSM_EL1.MAX when executing at EL1: the maximum bandwidth allowed for the PE is MIN(MPAMBWSM_EL1.MAX, MPAMBWCAP_EL2.CAP).

If the Effective value of HCR_EL2.{E2H,TGE} is not {1,1}:

- The upper bound also applies to MPAMBW0_EL1.MAX: the maximum bandwidth allowed for the PE is MIN(MPAMBW0_EL1.MAX, MPAMBWCAP_EL2.CAP).
- If FEAT_SME is implemented, the upper bound also applies to MPAMBWSM_EL1.MAX when executing at EL0: the maximum bandwidth allowed for the PE is MIN(MPAMBWSM_EL1.MAX, MPAMBWCAP_EL2.CAP).

If MPAMBWCAP_EL2.ENABLED is 1, a PARTID that has used more than min(CAP,MAX) is given no access to additional bandwidth.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented and MPAMIDR_EL1.HAS_HCR == '1'. Otherwise, direct accesses to MPAMBWCAP_EL2 are UNDEFINED.

Attributes

MPAMBWCAP_EL2 is a 64-bit register.

Field descriptions

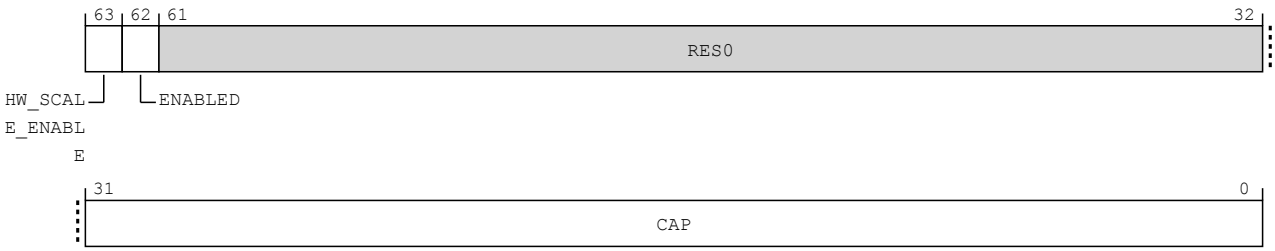

HW_SCALE_ENABLE, bit [63]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1':

Enables hardware bandwidth scaling of the MPAMBWCAP_EL2.CAP value.

HW_SCALE_ENABLE	Meaning
0b0	PE-side memory bandwidth control hardware scaling for EL2 capping is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HW_SCALE_ENABLE	Meaning
0b1	PE-side memory bandwidth control hardware scaling for EL2 capping is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ENABLED, bit [62]

Enables the PE-side memory bandwidth control capping by EL2.

ENABLED	Meaning
0b0	The PE-side memory bandwidth control capping by EL2 is disabled.
0b1	The PE-side memory bandwidth control capping by EL2 is enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [61:32]

Reserved, RES0.

CAP, bits [31:0]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1' and MPAMBWCAP_EL2.HW_SCALE_ENABLE == '1':

31	0
CAP	

CAP, bits [31:0]

Upper bound to the maximum memory bandwidth allocated to the current PARTID in MPAMBW1_EL1.MAX, MPAMBW0_EL1.MAX and MPAMBWSM_EL1.MAX.

The value is represented as a multiplier of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [31:16] represent the integer part of the value.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The value set in the MAX field must be less than or equal to this upper bound.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When MPAMBWIDR_EL1.HAS_HW_SCALE == '0' or MPAMBWCAP_EL2.HW_SCALE_ENABLE == '0':

31 … 16	15 … 0
RES0	CAP

Bits [31:16]

Reserved, RES0.

CAP, bits [15:0]

Upper bound to the maximum memory bandwidth allocated to the current PARTID in MPAMBW1_EL1.MAX, MPAMBW0_EL1.MAX and MPAMBWSM_EL1.MAX.

The value is represented as a fraction of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The value set in the MAX field must be less than or equal to this upper bound.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMBWCAP_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMBWCAP_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b110

```
if !(IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) && MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x910);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
            ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMBWCAP_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMBWCAP_EL2();
end;
```

MSR MPAMBWCAP_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b110

```
if !(IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) && MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x910) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
            if EL3SDDUndef() then
                Undefined();
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMBWCAP_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMBWCAP_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.10 MPAMBWIDR_EL1, MPAM PE-side Bandwidth Controls ID Register

The MPAMBWIDR_EL1 characteristics are:

Purpose

Indicates the supported PE-side memory bandwidth parameter values.

Configuration

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented. Otherwise, direct accesses to MPAMBWIDR_EL1 are UNDEFINED.

Attributes

MPAMBWIDR_EL1 is a 64-bit register.

Field descriptions

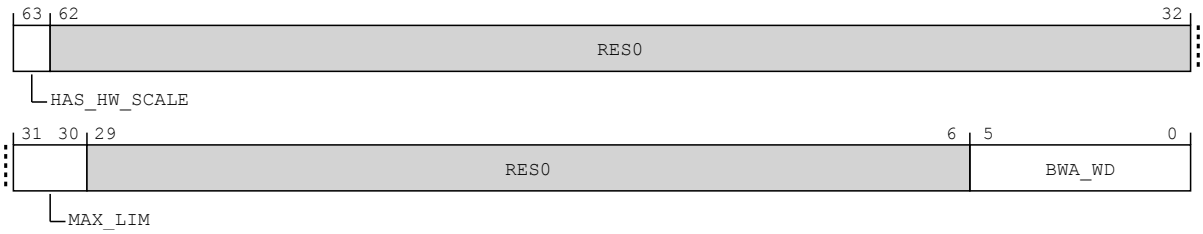

HAS_HW_SCALE, bit [63]

Indicates whether hardware support for auto-scaling of MPAMBWn_ELx.MAX, MPAMBWSM_EL1.MAX and MPAMBWCAP_EL2.CAP limits is available.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HAS_HW_SCALE	Meaning
0b0	Hardware support for auto-scaling is not implemented.
0b1	Hardware support for auto-scaling is implemented.

Access to this field is RO.

Bits [62:32]

Reserved, RES0.

MAX_LIM, bits [31:30]

Indicates the implemented maximum bandwidth limit partitioning behaviors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MAX_LIM	Meaning
0b00	Both soft limit and hard limit behaviors are implemented.
0b01	Soft limit behavior is implemented.
0b10	Hard limit behavior is implemented.
0b11	Reserved.

Access to this field is RO.

Bits [29:6]

Reserved, RES0.

BWA_WD, bits [5:0]

Indicates the number of implemented bits in the bandwidth allocation fields MPAMBWn_ELx.MAX, MPAMBWSM_EL1.MAX and MPAMBWCAP_EL2.CAP.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BWA_WD	Meaning
0b000001..0b010000	Number of implemented bits in the bandwidth allocation fields.

Access to this field is RO.

Accessing MPAMBWIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMBWIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBWIDR_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = MPAMBWIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMBWIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMBWIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.11 MPAMBWSM_EL1, MPAM Streaming Mode Bandwidth Control Register (EL1)

The MPAMBWSM_EL1 characteristics are:

Purpose

Enables software to configure a maximum fraction of memory bandwidth that the PE is permitted for SME memory accesses labeled with values from MPAMSM_EL1.

Configuration

This register is present only when FEAT_MPAM_PE_BW_CTRL is implemented and FEAT_SME is implemented. Otherwise, direct accesses to MPAMBWSM_EL1 are UNDEFINED.

Attributes

MPAMBWSM_EL1 is a 64-bit register.

Field descriptions

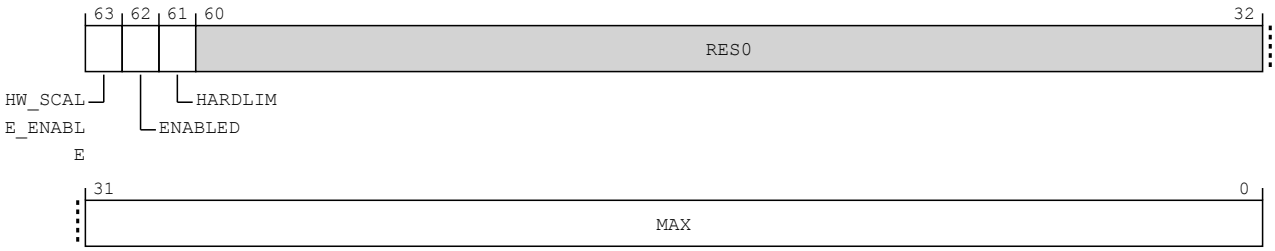

HW_SCALE_ENABLE, bit [63]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1':

Enables hardware bandwidth scaling of the MPAMBWSM_EL1.MAX value.

HW_SCALE_ENABLE	Meaning
0b0	PE-side memory bandwidth control hardware scaling for streaming PARTIDs is disabled.
0b1	PE-side memory bandwidth control hardware scaling for streaming PARTIDs is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ENABLED, bit [62]

Enables the PE-side memory bandwidth control for streaming PARTIDs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ENABLED	Meaning
0b0	PE-side memory bandwidth control for streaming PARTIDs is disabled.
0b1	PE-side memory bandwidth control for streaming PARTIDs is enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

HARDLIM, bit [61]

PE-side maximum bandwidth limit behavior selection.

HARDLIM	Meaning
0b0	Soft limit: when MPAMBWSM_EL1.MAX bandwidth is exceeded, the PE is unregulated unless the downstream memory path is saturated. It is IMPLEMENTATION DEFINED how hardware determines when the downstream memory path is saturated.
0b1	Hard limit: when MPAMBWSM_EL1.MAX bandwidth is exceeded, the PE does not use any more bandwidth until the memory bandwidth for the PE falls below MPAMBWSM_EL1.MAX.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When MPAMBWIDR_EL1.MAX_LIM == ‘00’, access to this field is RW.
- When MPAMBWIDR_EL1.MAX_LIM == ‘01’, access to this field is RAZ/WI.
- When MPAMBWIDR_EL1.MAX_LIM == ‘10’, access to this field is RAO/WI.

Bits [60:32]

Reserved, RES0.

MAX, bits [31:0]

When MPAMBWIDR_EL1.HAS_HW_SCALE == '1' and MPAMBWSM_EL1.HW_SCALE_ENABLE == '1':

31	0
MAX	

MAX, bits [31:0]

Maximum memory bandwidth allocated to the partition selected by MPAMSM_EL1.PARTID_D.

The value is represented as a multiplier of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [31:16] represent the integer part of the value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When MPAMBWIDR_EL1.HAS_HW_SCALE == '0' or MPAMBWSM_EL1.HW_SCALE_ENABLE == '0':

31 … 16	15 … 0
RES0	MAX

Bits [31:16]

Reserved, RES0.

MAX, bits [15:0]

Maximum memory bandwidth allocated to the partition selected by MPAMSM_EL1.PARTID_D.

The value is represented as a fraction of the available bandwidth for the PE. The value is represented in base-2 fixed-point format.

Bits [15:(16 - MPAMBWIDR_EL1.BWA_WD)] represent the fractional part of the value. When MPAMBWIDR_EL1.BWA_WD indicates a width less than 16 bits, bits [(15 - MPAMBWIDR_EL1.BWA_WD):0] are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMBWSM_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMBWSM_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) && IsFeatureImplemented(FEAT_SME)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
        ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
        ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBWSM_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = MPAMBWSM_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMBWSM_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMBWSM_EL1();
end;
```

MSR MPAMBWSM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_MPAM_PE_BW_CTRL) && IsFeatureImplemented(FEAT_SME)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && MPAMBW2_EL2().nTRAP_MPAMBWSM_EL1 == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        MPAMBWSM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && MPAMBW3_EL3().nTRAPLOWER == '0' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif HaveEL(EL3) && MPAMBW3_EL3().nTRAPLOWER == '0' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMBWSM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMBWSM_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.12 MPAMHCR_EL2, MPAM Hypervisor Control Register (EL2)

The MPAMHCR_EL2 characteristics are:

Purpose

Controls the PARTID virtualization features of MPAM.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when (FEAT_MPAMv1p0 is implemented or FEAT_MPAMv0p1 is implemented) and MPAMIDR_EL1.HAS_HCR == '1'. Otherwise, direct accesses to MPAMHCR_EL2 are UNDEFINED.

Attributes

MPAMHCR_EL2 is a 64-bit register.

Field descriptions

When FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented:

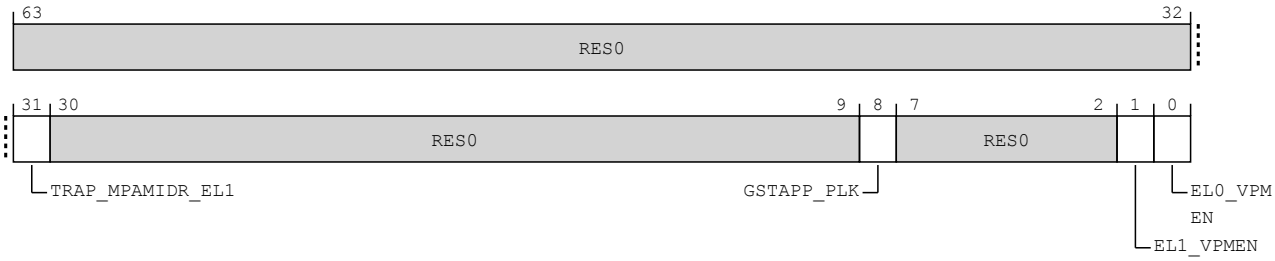

Bits [63:32]

Reserved, RES0.

TRAP_MPAMIDR_EL1, bit [31]

Trap accesses from EL1 to MPAMIDR_EL1 to EL2.

TRAP_MPAMIDR_EL1	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Direct accesses to MPAMIDR_EL1 from EL1 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to '1'.
  - When EL3 is implemented, this field resets to an architecturally UNKNOWN value.

Bits [30:9]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GSTAPP_PLK, bit [8]

Make the PARTIDs at EL0 the same as the PARTIDs at EL1. When executing at EL0, EL2 is enabled, HCR_EL2.TGE == 0 and GSTAPP_PLK = 1, MPAM1_EL1 is used instead of MPAM0_EL1 to generate MPAM labels for memory requests.

GSTAPP_PLK	Meaning
0b0	MPAM0_EL1 is used to generate MPAM labels when executing at EL0.
0b1	MPAM1_EL1 is used to generate MPAM labels when executing at EL0 with EL2 enabled and HCR_EL2.TGE == 0. Otherwise MPAM0_EL1 is used.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:2]

Reserved, RES0.

EL1_VPMEN, bit [1]

Enable virtualized identifiers in MPAM1_EL1 when executing at EL1. This bit also enables virtualized identifiers when MPAM1_EL1 is used to generate MPAM identifiers for memory requests at EL0 due to GSTAPP_PLK == 1.

EL1_VPMEN	Meaning
0b0	MPAM1_EL1.PARTID_I and MPAM1_EL1.PARTID_D are physical PARTIDs used to label memory system requests.
0b1	MPAM1_EL1.PARTID_I and MPAM1_EL1.PARTID_D are virtual PARTIDs used to index the MPAMVPMn_EL2.PhyPARTID fields to map the virtual PARTID into a physical PARTID to label memory system requests.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0_VPMEN, bit [0]

Enable virtualized identifiers in MPAM0_EL1 unless the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, EL0_VPMEN is ignored and MPAM0_EL1 identifiers are not virtualized.

When MPAMHCR_EL2.GSTAPP_PLK == 1 and HCR_EL2.TGE == 0, MPAM1_EL1 is used as the source of identifiers and the virtualization of identifiers in MPAM1_EL1 is controlled by MPAMHCR_EL2.EL1_VPMEN.

EL0_VPMEN	Meaning
0b0	MPAM0_EL1.PARTID_I and MPAM0_EL1.PARTID_D are physical PARTIDs used to label memory system requests.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0_VPMEN	Meaning
0b1	MPAM0_EL1.PARTID_I and MPAM0_EL1.PARTID_D are virtual PARTIDs used to index the MPAMVPMn_EL2.PhyPARTID fields to map the virtual PARTID into a physical PARTID to label memory system requests.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMHCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMHCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0100	0b000

```
if !((IsFeatureImplemented(FEAT_MPAMv1p0) || IsFeatureImplemented(FEAT_MPAMv0p1)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x930);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMHCR_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMHCR_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMHCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0100	0b000

```
if !((IsFeatureImplemented(FEAT_MPAMv1p0) || IsFeatureImplemented(FEAT_MPAMv0p1)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x930) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMHCR_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMHCR_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.13 MPAMIDR_EL1, MPAM ID Register (EL1)

The MPAMIDR_EL1 characteristics are:

Purpose

Indicates the maximum PARTID and PMG values supported in the implementation and the support for other optional features.

Configuration

This register is present only when FEAT_MPAM is implemented. Otherwise, direct accesses to MPAMIDR_EL1 are UNDEFINED.

Attributes

MPAMIDR_EL1 is a 64-bit register.

Field descriptions

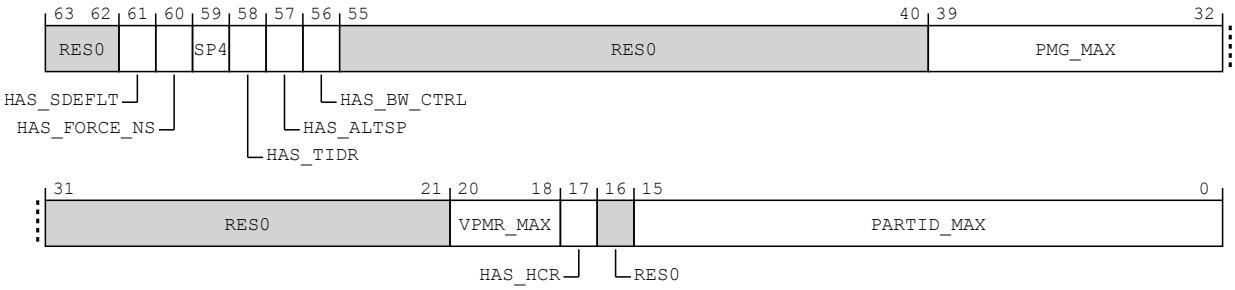

Bits [63:62]

Reserved, RES0.

HAS_SDEFLT, bit [61]

HAS_SDEFLT indicates support for MPAM3_EL3.SDEFLT bit.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HAS_SDEFLT	Meaning
0b0	The SDEFLT bit is not implemented in MPAM3_EL3.
0b1	The SDEFLT bit is implemented in MPAM3_EL3.

When MPAM3_EL3.SDEFLT == 1, accesses from the Secure Execution state use the default PARTID, PARTID == 0.

Access to this field is RO.

HAS_FORCE_NS, bit [60]

HAS_FORCE_NS indicates support for MPAM3_EL3.FORCE_NS bit.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HAS_FORCE_NS	Meaning
0b0	The FORCE_NS bit is not implemented in MPAM3_EL3.
0b1	The FORCE_NS bit is implemented in MPAM3_EL3.

When MPAM3_EL3.FORCE_NS == 1, accesses from the Secure Execution state have MPAM_NS == 1.

Access to this field is RO.

SP4, bit [59]

Supports 4 MPAM PARTID spaces.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SP4	Meaning
0b0	MPAM supports 2 PARTID spaces.
0b1	MPAM supports 4 PARTID spaces.

Access to this field is RO.

HAS_TIDR, bit [58]

HAS_TIDR indicates support for MPAM2_EL2.TIDR bit.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HAS_TIDR	Meaning
0b0	The TIDR bit is not implemented in MPAM2_EL2.
0b1	The TIDR bit is implemented in MPAM2_EL2.

Note

Arm recommends that when the MPAM version is MPAM v0.1 or MPAM v1.1, MPAMIDR_EL1.HAS_TIDR is 1 and that the MPAM2_EL2.TIDR field is implemented.

Access to this field is RO.

HAS_ALTSP, bit [57]

HAS_ALTSP indicates support for alternative PARTID spaces.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HAS_ALTSP	Meaning
0b0	Alternative PARTID spaces are not implemented.
0b1	Alternative PARTID spaces are implemented with control bits in MPAM3_EL3 and MPAM2_EL2.

Access to this field is RO.

HAS_BW_CTRL, bit [56]

HAS_BW_CTRL indicates support for PE-side bandwidth controls.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HAS_BW_CTRL	Meaning
0b0	PE-side bandwidth controls are not implemented.
0b1	PE-side bandwidth controls are implemented.

FEAT_MPAM_PE_BW_CTRL implements the functionality identified by the value 1.

Access to this field is RO.

Bits [55:40]

Reserved, RES0.

PMG_MAX, bits [39:32]

The largest value of PMG that the implementation can generate. The PMG_I and PMG_D fields of every MPAMn_ELx must implement at least enough bits to represent PMG_MAX.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [31:21]

Reserved, RES0.

VPMR_MAX, bits [20:18]

When MPAMIDR_EL1.HAS_HCR == '1':

Indicates the maximum register index n for the MPAMVPM<n>_EL2 registers.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

HAS_HCR, bit [17]

HAS_HCR indicates that the PE implementation supports MPAM virtualization, including MPAMHCR_EL2, MPAMVPMV_EL2, and MPAMVPM<n>_EL2 with n in the range 0 to VPMR_MAX. Must be 0 if EL2 is not implemented in either Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

HAS_HCR	Meaning
0b0	MPAM virtualization is not supported.
0b1	MPAM virtualization is supported.

Access to this field is RO.

Bit [16]

Reserved, RES0.

PARTID_MAX, bits [15:0]

The largest value of PARTID that the implementation can generate. The PARTID_I and PARTID_D fields of every MPAMn_ELx must implement at least enough bits to represent PARTID_MAX.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPAMIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_MPAM) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAMIDR_EL1().HAS_HCR == '1' && MPAMHCR_EL2().TRAP_MPAMIDR_EL1 == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAMIDR_EL1().HAS_TIDR == '1' && MPAM2_EL2().TIDR == '1' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = MPAMIDR_EL1();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMIDR_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMIDR_EL1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.14 MPAMSM_EL1, MPAM Streaming Mode Register

The MPAMSM_EL1 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests that are:

- Issued due to the execution of SME load and store instructions.
- Issued when the PE is in Streaming SVE mode due to the execution of SVE and SIMD&FP load and store instructions and SVE prefetch instructions.

If an implementation uses a shared SMCU, then the MPAM labels in this register have precedence over the labels in MPAM0_EL1, MPAM1_EL1, MPAM2_EL2, and MPAM3_EL3.

If an implementation includes an SMCU that is not shared with other PEs, then it is IMPLEMENTATION DEFINED whether the MPAM labels in this register have precedence over the labels in MPAM0_EL1, MPAM1_EL1, MPAM2_EL2, and MPAM3_EL3.

When FEAT_MPAMv1p0 or FEAT_MPAMv0p1 is implemented, the MPAM labels in this register are used if MPAM1_EL1.MPAMEN is 1.

For memory requests issued from EL0, the MPAM PARTID in this register is virtual and mapped into a physical PARTID when all of the following are true:

- EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- The MPAM virtualization option is implemented and MPAMHCR_EL2.EL0_VPMEN is 1.

For memory requests issued from EL1, the MPAM PARTID in this register is virtual and mapped into a physical PARTID when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The MPAM virtualization option is implemented and MPAMHCR_EL2.EL1_VPMEN is 1.

Configuration

This register is present only when FEAT_MPAM is implemented and FEAT_SME is implemented. Otherwise, direct accesses to MPAMSM_EL1 are UNDEFINED.

Attributes

MPAMSM_EL1 is a 64-bit register.

Field descriptions

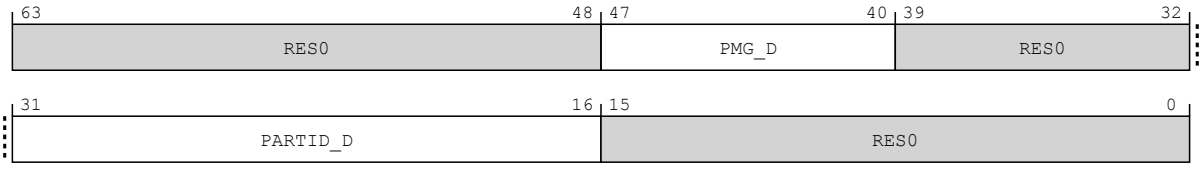

Bits [63:48]

Reserved, RES0.

PMG_D, bits [47:40]

Performance monitoring group property for PARTID_D.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [39:32]

Reserved, RES0.

PARTID_D, bits [31:16]

Partition ID for requests issued due to the execution at any Exception level of SME load and store instructions and, when the PE is in Streaming SVE mode, SVE and SIMD&FP load and store instructions and SVE prefetch instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:0]

Reserved, RES0.

Accessing MPAMSM_EL1

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMSM_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_MPAM) && IsFeatureImplemented(FEAT_SME)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAM2_EL2().EnMPAMSM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        X{64}(t) = MPAMSM_EL1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMSM_EL1();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMSM_EL1();
end;
```

MSR MPAMSM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_MPAM) && IsFeatureImplemented(FEAT_SME)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
         ↪&& MPAM2_EL2().EnMPAMSM == '0' then
        AArch64_SystemAccessTrap(EL2, 0x18);
    else
        MPAMSM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMSM_EL1() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMSM_EL1() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.15 MPAMVPM0_EL2, MPAM Virtual PARTID Mapping Register 0

The MPAMVPM0_EL2 characteristics are:

Purpose

MPAMVPM0_EL2 provides mappings from virtual PARTIDs 0 - 3 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM<n>_EL2 register. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented) and MPAMIDR_EL1.HAS_HCR == '1'. Otherwise, direct accesses to MPAMVPM0_EL2 are UNDEFINED.

Attributes

MPAMVPM0_EL2 is a 64-bit register.

Field descriptions

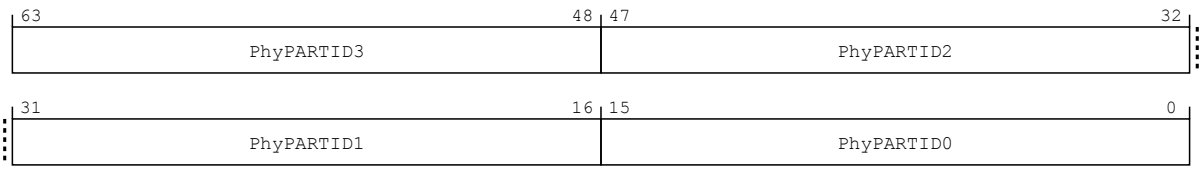

PhyPARTID3, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 3. PhyPARTID3 gives the mapping of virtual PARTID 3 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID2, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 2. PhyPARTID2 gives the mapping of virtual PARTID 2 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID1, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 1. PhyPARTID1 gives the mapping of virtual PARTID 1 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PhyPARTID0, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 0. PhyPARTID0 gives the mapping of virtual PARTID 0 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM0_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM0_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b000

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x940);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM0_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM0_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM0_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b000

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x940) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM0_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM0_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.16 MPAMVPM1_EL2, MPAM Virtual PARTID Mapping Register 1

The MPAMVPM1_EL2 characteristics are:

Purpose

MPAMVPM1_EL2 provides mappings from virtual PARTIDs 4 - 7 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM0_EL2 to MPAMVPM7_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) > 0). Otherwise, direct accesses to MPAMVPM1_EL2 are UNDEFINED.

Attributes

MPAMVPM1_EL2 is a 64-bit register.

Field descriptions

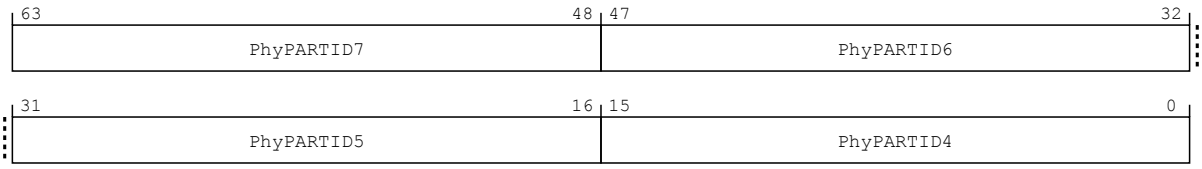

PhyPARTID7, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 7. PhyPARTID7 gives the mapping of virtual PARTID 7 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID6, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 6. PhyPARTID6 gives the mapping of virtual PARTID 6 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID5, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 5. PhyPARTID5 gives the mapping of virtual PARTID 5 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID4, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 4. PhyPARTID4 gives the mapping of virtual PARTID 4 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM1_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM1_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b001

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x948);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM1_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM1_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM1_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b001

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 0) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x948) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM1_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM1_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.17 MPAMVPM2_EL2, MPAM Virtual PARTID Mapping Register 2

The MPAMVPM2_EL2 characteristics are:

Purpose

MPAMVPM2_EL2 provides mappings from virtual PARTIDs 8 - 11 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM0_EL2 to MPAMVPM7_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) > 1). Otherwise, direct accesses to MPAMVPM2_EL2 are UNDEFINED.

Attributes

MPAMVPM2_EL2 is a 64-bit register.

Field descriptions

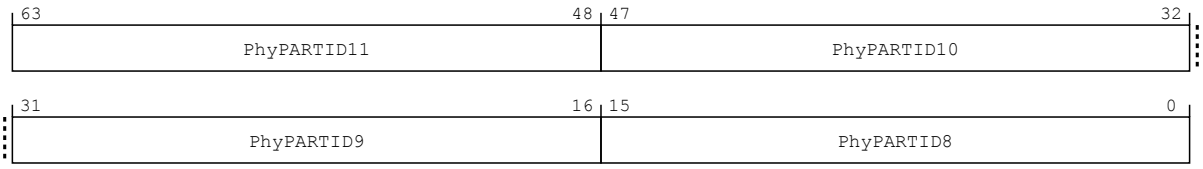

PhyPARTID11, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 11. PhyPARTID11 gives the mapping of virtual PARTID 11 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID10, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 10. PhyPARTID10 gives the mapping of virtual PARTID 10 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID9, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 9. PhyPARTID9 gives the mapping of virtual PARTID 9 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID8, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 8. PhyPARTID8 gives the mapping of virtual PARTID 8 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM2_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM2_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b010

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x950);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM2_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM2_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM2_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b010

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x950) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM2_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM2_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.18 MPAMVPM3_EL2, MPAM Virtual PARTID Mapping Register 3

The MPAMVPM3_EL2 characteristics are:

Purpose

MPAMVPM3_EL2 provides mappings from virtual PARTIDs 12 - 15 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM<n>_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) > 2). Otherwise, direct accesses to MPAMVPM3_EL2 are UNDEFINED.

Attributes

MPAMVPM3_EL2 is a 64-bit register.

Field descriptions

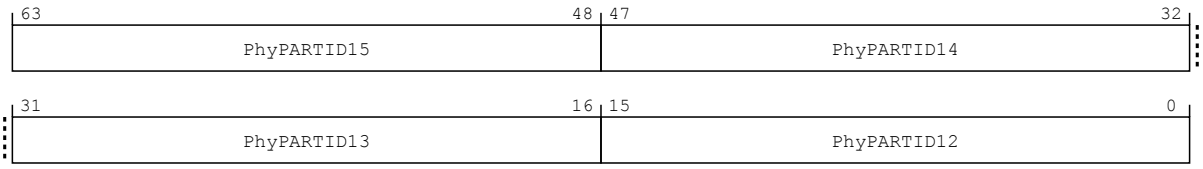

PhyPARTID15, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 15. PhyPARTID15 gives the mapping of virtual PARTID 15 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID14, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 14. PhyPARTID14 gives the mapping of virtual PARTID 14 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID13, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 13. PhyPARTID13 gives the mapping of virtual PARTID 13 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID12, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 12. PhyPARTID12 gives the mapping of virtual PARTID 12 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM3_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM3_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b011

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x958);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM3_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM3_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM3_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b011

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x958) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM3_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM3_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.19 MPAMVPM4_EL2, MPAM Virtual PARTID Mapping Register 4

The MPAMVPM4_EL2 characteristics are:

Purpose

MPAMVPM4_EL2 provides mappings from virtual PARTIDs 16 - 19 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM<n>_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) > 3). Otherwise, direct accesses to MPAMVPM4_EL2 are UNDEFINED.

Attributes

MPAMVPM4_EL2 is a 64-bit register.

Field descriptions

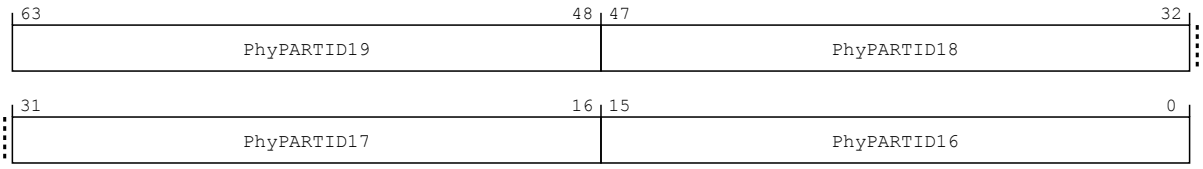

PhyPARTID19, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 19. PhyPARTID19 gives the mapping of virtual PARTID 19 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID18, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 18. PhyPARTID18 gives the mapping of virtual PARTID 18 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID17, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 17. PhyPARTID17 gives the mapping of virtual PARTID 17 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID16, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 16. PhyPARTID16 gives the mapping of virtual PARTID 16 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM4_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM4_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b100

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x960);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM4_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM4_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM4_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b100

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x960) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM4_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM4_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.20 MPAMVPM5_EL2, MPAM Virtual PARTID Mapping Register 5

The MPAMVPM5_EL2 characteristics are:

Purpose

MPAMVPM5_EL2 provides mappings from virtual PARTIDs 20 - 23 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM<n>_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) > 4). Otherwise, direct accesses to MPAMVPM5_EL2 are UNDEFINED.

Attributes

MPAMVPM5_EL2 is a 64-bit register.

Field descriptions

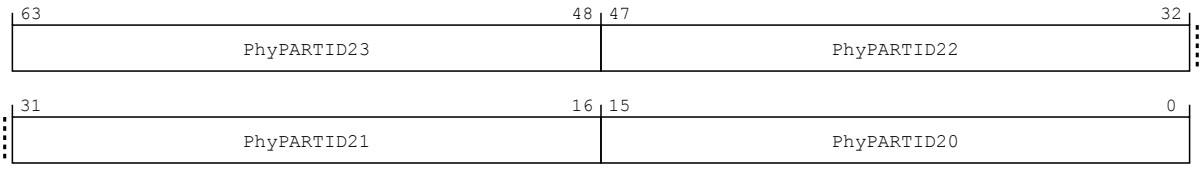

PhyPARTID23, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 23. PhyPARTID23 gives the mapping of virtual PARTID 23 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID22, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 22. PhyPARTID22 gives the mapping of virtual PARTID 22 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID21, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 21. PhyPARTID21 gives the mapping of virtual PARTID 21 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID20, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 20. PhyPARTID20 gives the mapping of virtual PARTID 20 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM5_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM5_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b101

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 4) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x968);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM5_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM5_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM5_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b101

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 4) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x968) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM5_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM5_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.21 MPAMVPM6_EL2, MPAM Virtual PARTID Mapping Register 6

The MPAMVPM6_EL2 characteristics are:

Purpose

MPAMVPM6_EL2 provides mappings from virtual PARTIDs 24 - 27 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM<n>_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for the PARTID in MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) > 5). Otherwise, direct accesses to MPAMVPM6_EL2 are UNDEFINED.

Attributes

MPAMVPM6_EL2 is a 64-bit register.

Field descriptions

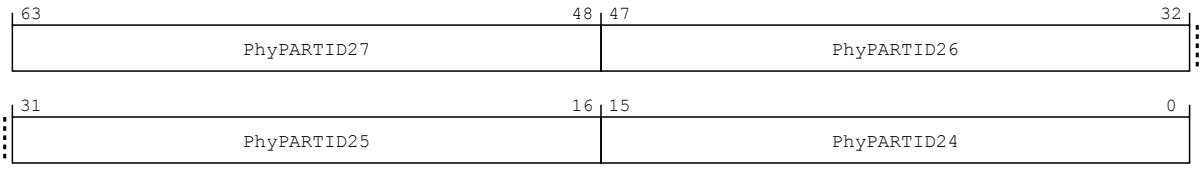

PhyPARTID27, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 27. PhyPARTID27 gives the mapping of virtual PARTID 27 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID26, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 26. PhyPARTID26 gives the mapping of virtual PARTID 26 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID25, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 25. PhyPARTID25 gives the mapping of virtual PARTID 25 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID24, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 24. PhyPARTID24 gives the mapping of virtual PARTID 24 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM6_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM6_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b110

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 5) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x970);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM6_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM6_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM6_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b110

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) > 5) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x970) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM6_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM6_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.22 MPAMVPM7_EL2, MPAM Virtual PARTID Mapping Register 7

The MPAMVPM7_EL2 characteristics are:

Purpose

MPAMVPM7_EL2 provides mappings from virtual PARTIDs 28 - 31 to physical PARTIDs.

MPAMIDR_EL1.VPMR_MAX field gives the index of the highest implemented MPAMVPM<n>_EL2 registers. VPMR_MAX can be as large as 7 (8 registers) or 32 virtual PARTIDs. If MPAMIDR_EL1.VPMR_MAX == 0, there is only a single MPAMVPM<n>_EL2 register, MPAMVPM0_EL2.

Virtual PARTID mapping is enabled by MPAMHCR_EL2.EL1_VPMEN for the PARTID in MPAM1_EL1 and by MPAMHCR_EL2.EL0_VPMEN for MPAM0_EL1.

A virtual-to-physical PARTID mapping entry, PhyPARTID<n>, is valid only when the MPAMVPMV_EL2.VPM_V bit in bit position n is set to 1.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) && (MPAMIDR_EL1.HAS_HCR == '1')) && (UInt(MPAMIDR_EL1.VPMR_MAX) == 7). Otherwise, direct accesses to MPAMVPM7_EL2 are UNDEFINED.

Attributes

MPAMVPM7_EL2 is a 64-bit register.

Field descriptions

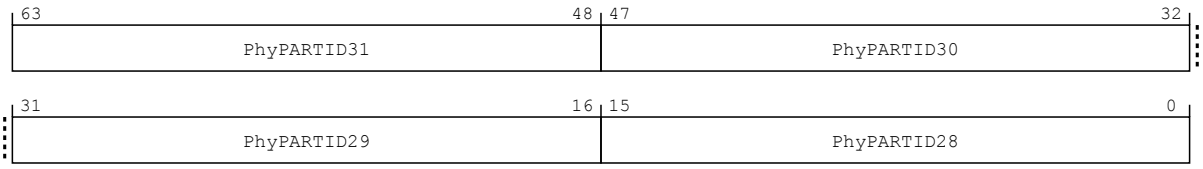

PhyPARTID31, bits [63:48]

Virtual PARTID Mapping Entry for virtual PARTID 31. PhyPARTID31 gives the mapping of virtual PARTID 31 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID30, bits [47:32]

Virtual PARTID Mapping Entry for virtual PARTID 30. PhyPARTID30 gives the mapping of virtual PARTID 30 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID29, bits [31:16]

Virtual PARTID Mapping Entry for virtual PARTID 29. PhyPARTID29 gives the mapping of virtual PARTID 29 to a physical PARTID.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PhyPARTID28, bits [15:0]

Virtual PARTID Mapping Entry for virtual PARTID 28. PhyPARTID28 gives the mapping of virtual PARTID 28 to a physical PARTID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAMVPM7_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPAMVPM7_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b111

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) == 7) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x978);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPM7_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPM7_EL2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MPAMVPM7_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0110	0b111

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1' && UInt(MPAMIDR_EL1().VPMR_MAX) == 7) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x978) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPM7_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPM7_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D24.12.23 MPAMVPMV_EL2, MPAM Virtual Partition Mapping Valid Register

The MPAMVPMV_EL2 characteristics are:

Purpose

Valid bits for virtual PARTID mapping entries. Each bit m corresponds to virtual PARTID mapping entry m in the MPAMVPM<n>_EL2 registers where n = m » 2.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p0 is implemented) and MPAMIDR_EL1.HAS_HCR == '1'. Otherwise, direct accesses to MPAMVPMV_EL2 are UNDEFINED.

Attributes

MPAMVPMV_EL2 is a 64-bit register.

Field descriptions

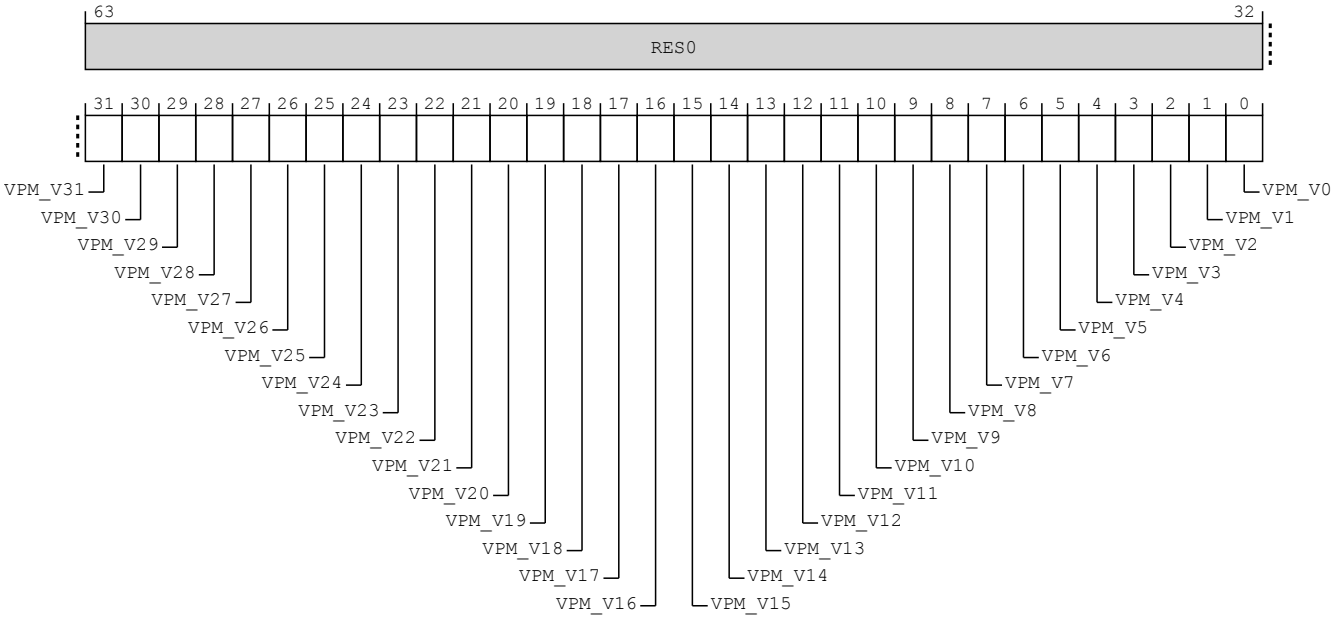

Bits [63:32]

Reserved, RES0.

VPM_V<m> , bits [m], for m = 31 to 0

Contains valid bit for virtual PARTID mapping entry corresponding to virtual PARTID<m>.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When m >= ((UInt(MPAMIDR_EL1.VPMR_MAX) + 1) * 4), access to this field is RES0.

Accessing MPAMVPMV_EL2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, MPAMVPMV_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0100	0b001

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        X{64}(t) = NVMem(0x938);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        X{64}(t) = MPAMVPMV_EL2();
    end;
elsif PSTATE.EL == EL3 then
    X{64}(t) = MPAMVPMV_EL2();
end;
```

```
MSR MPAMVPMV_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0100	0b001

```
if !((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
     ↪MPAMIDR_EL1().HAS_HCR == '1') then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EffectiveHCR_EL2_NVx() IN {'1x1'} then
        NVMem(0x938) = X{64}(t);
    elsif EffectiveHCR_EL2_NVx() IN {'xx1'} then
        if HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0))
             ↪&& MPAM3_EL3().TRAPLOWER == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_SystemAccessTrap(EL3, 0x18);
            end;
        else
            AArch64_SystemAccessTrap(EL2, 0x18);
        end;
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && (IsFeatureImplemented(FEAT_MPAMv0p1) ||
         ↪IsFeatureImplemented(FEAT_MPAMv1p0)) && MPAM3_EL3().TRAPLOWER == '1' then
        Undefined();
    elsif HaveEL(EL3) && (IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p0)) &&
         ↪MPAM3_EL3().TRAPLOWER == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_SystemAccessTrap(EL3, 0x18);
        end;
    else
        MPAMVPMV_EL2() = X{64}(t);
    end;
elsif PSTATE.EL == EL3 then
    MPAMVPMV_EL2() = X{64}(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part E
The AArch32 Application Level Architecture

Chapter E1
The AArch32 Application Level Programmers' Model

This chapter gives an Application level description of the programmers' model for software executing in AArch32 state. This means it describes execution in EL0 when EL0 is using AArch32. It contains the following sections:

- About the Application level programmers' model.
- The Application level programmers' model in AArch32 state.
- Advanced SIMD and floating-point instructions.
- About the AArch32 System register interface.
- Exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for the development of applications that execute in AArch32 state.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of that system information is needed to put the Application level programmers' model into context.

Depending on the implementation, the architecture supports multiple levels of execution privilege. These privilege levels are indicated by different *Exception levels* that number upwards from EL0, where EL0 corresponds to the lowest privilege level and is often described as *unprivileged*. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see Arm architectural concepts.

System software determines the Exception level, and therefore the level of privilege, at which application software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged. This has the following effects:

- It means that the operating system can allocate system resources to an application in a unique or shared manner.
- It provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some System level understanding is helpful, and if appropriate it gives a reference to the System level description.

Application level software is generally unaware of its Security state, and of any virtualization. For more information, see The Armv8-A security model and The effect of implementing EL2 on the Exception model.

Note

- When an implementation includes EL3, application and operating system software normally executes in Non-secure state.
- Older documentation, describing implementations or architecture versions that support only two privilege levels, often refers to execution at EL1 as *privileged* execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.2 The Application level programmers' model in AArch32 state

The following sections give more information about the application level programmers' model in AArch32 state:

- Instruction sets, arithmetic operations, and register files.
- Core data types and arithmetic in AArch32 state.
- The general-purpose registers, and the PC, in AArch32 state.
- Process state, PSTATE.
- Jazelle support.

E1.2.1 Instruction sets, arithmetic operations, and register files

The T32 and A32 instruction sets both provide a wide range of integer arithmetic and logical operations, that operate on a register file of sixteen 32-bit registers, that are comprised of the AArch32 general-purpose registers and the PC. As described in The general-purpose registers, and the PC, in AArch32 state, these registers include the registers SP (R13) and LR (R14), which have specialized uses. Core data types and arithmetic in AArch32 state gives more information about these operations.

In addition, an implementation that implements the T32 and A32 instruction sets includes both:

- Scalar floating-point instructions.
- The Advanced SIMD vector instructions.

Floating-point and vector instructions operate on a separate common register file, described in The SIMD and floating-point register file. Advanced SIMD and floating-point instructions gives more information about these instructions.

E1.2.2 Core data types and arithmetic in AArch32 state

When executing in AArch32 state, a PE supports the following data types in memory:

Byte 8 bits.

Halfword 16 bits.

Word 32 bits.

Doubleword 64 bits.

PE registers are 32 bits in size. The instruction sets provide instructions that use the following data types for data held in registers:

- 32-bit pointers.
- Unsigned or signed 32-bit integers.
- Unsigned 16-bit or 8-bit integers, held in zero-extended form.
- Signed 16-bit or 8-bit integers, held in sign-extended form.
- Two 16-bit integers packed into a register.
- Four 8-bit integers packed into a register.
- Unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. Software can load and store doublewords using these instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For information about the atomicity of memory accesses, see Atomicity in the Arm architecture.

When any of the data types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to 2^N-1, using normal binary format.

When any of these types is described as *signed*, the N-bit data value represents an integer in the range $-2^{(N-1)}$ to $+2^{(N-1)}-1$, using two's complement format.

The instructions that operate on packed halfwords or bytes include some multiply instructions that use only one of two halfwords, and SIMD instructions that perform parallel addition or subtraction on all of the halfwords or bytes.

Note

These SIMD instructions operate on values held in the general-purpose registers, and must not be confused with the Advanced SIMD instructions that operate on a separate register file that provides registers of up to 128 bits.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

E1.2.2.1 Integer arithmetic

The instruction set provides a wide range of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and divisions. The pseudocode defines these operations either using ASL stdlib code or using ASL functions described in the *Arm® Architecture Specification Language Reference (ARM DDI 0626)*.

E1.2.2.1.1 Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

Logical Shift Left

The `LSL()` stdlib function moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Logical Shift Right

The `LSR()` stdlib function moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Arithmetic Shift Right

The `ASR()` stdlib function moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Rotate Right The `ROR()` stdlib function moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the right end of the bitstring can be produced as a carry output.

Rotate Right with Extend

The `RRX()` pseudocode function moves each bit of a bitstring right by one bit. A carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.

E1.2.2.1.2 Pseudocode description of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand or operands otherwise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. When necessary, multi-word additions and subtractions can be synthesized from this status information. In pseudocode, the `AddWithCarry()` function provides an addition with a carry input and a set of output Condition flags including carry output and overflow:

An important property of the `AddWithCarry()` function is that if:

```
(result, nzcv) = AddWithCarry{N}(x, NOT( y), carry_in)
```

Then:

- If `carry_in == '1'`, then `result == x-y` with:
  - `nzcv[0] == '1'` if signed overflow occurred during the subtraction.
  - `nzcv[1] == '1'` if unsigned borrow did not occur during the subtraction, that is, if `x` $\geq$ `y`.
- If `carry_in == '0'`, then `result == x-y-1` with:
  - `nzcv[0] == '1'` if signed overflow occurred during the subtraction.
  - `nzcv[1] == '1'` if unsigned borrow did not occur during the subtraction, that is, if `x` $\geq$ `y`.

Taken together, this means that the `carry_in` and `nzcv[1]` output in `AddWithCarry{N}()` calls can act as NOT borrow flags for subtractions as well as carry flags for additions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.2.2.1.3 Pseudocode description of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N. This is supported in pseudocode by:

- The `SignedSatQ{N}()` and `UnsignedSatQ{N}()` functions when an operation requires, in addition to the saturated result, a Boolean argument that indicates whether saturation occurred.
- The `SignedSat{N}()` and `UnsignedSat{N}()` functions when only the saturated result is required.

`SatQ{N}(i, unsigned)` returns either `UnsignedSatQ{N}(i)` or `SignedSatQ{N}(i)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat{N}(i)` or `SignedSat{N}(i)` depending on the value of its third argument.

E1.2.3 The general-purpose registers, and the PC, in AArch32 state

In the AArch32 Application level view, a PE has:

- Fifteen general-purpose 32-bit registers, R0 to R14, of which R13 and R14 have alternative names reflecting how they are, or can be, used:
  - R13 is usually identified as SP.
  - R14 is usually identified as LR.
- The PC (*Program Counter*), which can be described as R15.

The specialized uses of the SP (R13), LR (R14), and PC (R15) are:

SP, the stack pointer

The PE uses SP as a pointer to the active stack.

In the T32 instruction set, some instructions cannot access SP. Instructions that can access SP can use SP as a general-purpose register.

The A32 instruction set provides more general access to SP, and it can be used as a general-purpose register.

Note

Using SP for any purpose other than as a stack pointer might break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.

Software can refer to SP as R13.

LR, the link register

The link register can be used to hold return link information, and some cases described in this manual require this use of the LR. When software does not require the LR for linking, it can use it for other purposes. Software can refer to LR as R14.

PC, the Program Counter

- When executing an A32 instruction, PC reads as the address of the current instruction plus 8.
- When executing a T32 instruction, PC reads as the address of the current instruction plus 4.
- Writing an address to PC causes a branch to that address.

Most T32 instructions cannot access PC.

The A32 instruction set provides more general access to the PC, and many A32 instructions can use the PC as a general-purpose register. However, Arm deprecates the use of PC for any purpose other than as the Program Counter. See Writing to the PC for more information.

Software can refer to PC as R15.

See AArch32 general-purpose registers, the PC, and the Special-purpose registers for the system level view of these registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In general, Arm strongly recommends using the names SP, LR, and PC instead of R13, R14, and R15. However, sometimes it is simpler to use the R13-R15 names when referring to a group of registers. For example, it is simpler to refer to *registers R8 to R15*, rather than to *registers R8 to R12, the SP, LR, and PC*. These two descriptions of the group of registers have the same meaning.

E1.2.3.1 Writing to the PC

In the T32 and A32 instruction sets, many data-processing instructions can write to the PC. Writes to the PC are handled as follows:

- In T32 state, the following 16-bit T32 instruction encodings branch to the value written to the PC:
  - Encoding T2 of ADD, ADDS (register).
  - Encoding T1 of MOV, MOVS (register).

  The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.
- The `B`, `BL`, `CBNZ`, `CBZ`, `TBB`, and `TBH` instructions remain in the same instruction set state and branch to the value written to the PC.

  The definition of each of these instructions ensures that the value written to the PC is correctly aligned for the current instruction set state.
- The `BLX` (immediate) instruction switches between T32 and A32 states and branches to the value written to the PC. Its definition ensures that the value written to the PC is correctly aligned for the new instruction set state.
- The following instructions write a value to the PC, treating that value as an interworking address to branch to, with low-order bits that determine the new instruction set state:
  - `BLX` (register), `BX`, and `BXJ`.
  - `LDR` instructions with `<Rt>` equal to the PC.
  - `POP` and all forms of `LDM` except `LDM` (exception return), when the register list includes the PC.
  - In A32 state only, `ADC`, `ADD`, `ADR`, `AND`, `ASR` (immediate), `BIC`, `EOR`, `LSL` (immediate), `LSR` (immediate), `MOV`, `MVN`, `ORR`, `ROR` (immediate), `RRX`, `RSB`, `RSC`, `SBC`, and `SUB` instructions with `<Rd>` equal to the PC and without flag-setting specified.

  For details of how an interworking address specifies the new instruction set state and instruction address, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

  Note

  The register-shifted register instructions, which are available only in the A32 instruction set and are summarized in Data-processing register (register shift), are CONSTRAINED UNPREDICTABLE if they attempt to write to the PC, see Using R15 by instruction.

- Some instructions are treated as exception return instructions, and write both the PC and the CPSR. For more information, including which instructions are exception return instructions, see Exception return to an Exception level using AArch32.
- Some instructions cause an exception, and the exception handler address is written to the PC as part of the exception entry.

E1.2.3.2 Pseudocode description of operations on the AArch32 general-purpose registers and the PC

In pseudocode, the uses of the `R()` accessor, with an index argument `n`, are:

- Reading or writing R0-R12, SP, and LR, using n = 0-12, 13, and 14 respectively.
- Reading the PC, using n = 15.

Pseudocode description of general-purpose register and PC operations describes accesses to these registers.

Descriptions of A32 store instructions that store the PC value use the `PCStoreValue()` pseudocode function to specify the PC value stored by the instruction.

Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that also selects the instruction set to execute after the branch. A simple branch is performed by the `BranchWritePC()` function.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An interworking branch is performed by the `BXWritePC()` function.

The `LoadWritePC()` and `ALUWritePC()` functions are used for two cases where the behavior was systematically modified between architecture versions.

E1.2.4 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

Note

In this chapter, references to PSTATE link to the more appropriate of:

- The Application-level view of PSTATE given in this section.
- The System-level description in Process state, PSTATE.

The following PSTATE information is accessible at EL0:

The condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C, and V Condition flags, combining them with the *Condition code* for the instruction, to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see Conditional execution.

The overflow or saturation flag

Q Some instructions can set this. For those instructions that can, the PE:

- Sets it to 1 if the instruction indicates overflow or saturation.
- Leaves it unchanged otherwise.

For more information, see Pseudocode description of saturation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The greater than or equal flags

GE[3:0] The instructions described in Parallel addition and subtraction instructions update these to indicate the results from individual bytes or halfwords of the operation. These flags can control a later `SEL` instruction. For more information, see SEL.

PSTATE also contains *PE state controls*. There is no direct access to these from application level instructions, but they can be changed by side-effects of application level instructions. They are:

Instruction set state

J, T The current instruction set state, as shown in Table E1-1. The J bit is RES0, see the Note in this section.

Table E1-1 PSTATE.{J, T} encoding

J	T	Instruction set state
0	0	A32
0	1	T32

A32 The PE is executing the A32 instruction set, summarized in A32 Instruction Set Encoding.

T32 The PE is executing the T32 instruction set, summarized in T32 Instruction Set Encoding.

Note

Encoding with J==1 before Armv8, Jazelle, and T32EE states

In previous versions of the Arm architecture, the encoding {1, 0} selected Jazelle state, and encoding {1, 1} selected T32EE state. From the introduction of Armv8, the architecture does not support either of these states, and these are encodings for unimplemented instruction set states, see Unimplemented instruction sets.

AArch32 state requires a Trivial Jazelle implementation, see Trivial implementation of the Jazelle extension.

The IT block state

IT[7:0] The If-Then controls for the T32 `IT` instruction, which applies to the *IT block* of instructions that immediately follow the `IT` instruction. See IT for a description of the `IT` instruction and its associated `IT` block.

For more information about the use of PSTATE.IT, see Use of PSTATE.IT.

Endianness mapping

E For data accesses, controls the endianness:

0 Little-endian.

1 Big-endian.

If an implementation does not provide:

- Big-endian support for data accesses, this bit is RES0.
- Little-endian support for data accesses, this bit is RES1.

Instruction fetches are always little-endian, and ignore PSTATE.E.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Timing control bits

DIT *Data Independent Timing* (DIT) bit. For more information, see About the DIT bit.

This bit is implemented only when FEAT_DIT is implemented.

On a reset to AArch32 state, this bit is set to 0.

E1.2.4.1 Accessing PSTATE fields at EL0

The following sections describe which PSTATE fields can be directly accessed at EL0, and how they can be accessed:

- The Application Program Status Register, APSR.
- The SETEND instruction.

E1.2.4.1.1 The Application Program Status Register, APSR

At EL0, some PSTATE fields can be accessed using the Special-purpose *Application Program Status Register* (APSR). The APSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

The APSR bit assignments are:

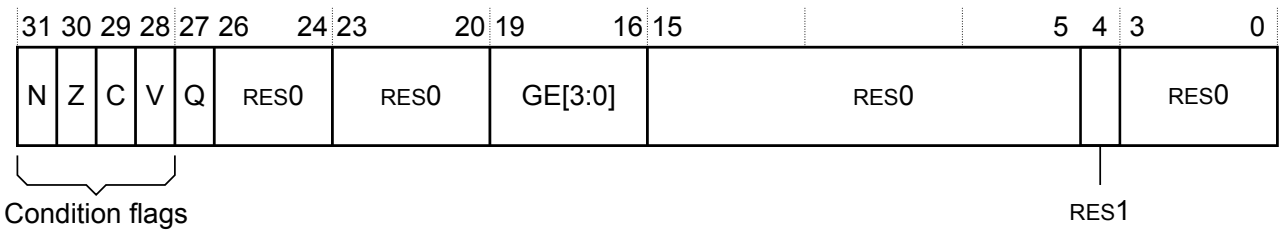

N, Z, C, V, bits [31:28]

The PSTATE Condition flags.

Q, bit [27] The PSTATE overflow or saturation flag.

Bits[26:24] Reserved, RES0. Software can use MSR instructions that write the top byte of the APSR without using a read-modify-write sequence. If it does this, it must write zeros to bits[26:24].

Bits[23:20, 15:0]

Reserved bits that are allocated to system features, or are available for future expansion. Unprivileged execution ignores writes to fields that are accessible only at EL1 or higher. However, application level software that writes to the APSR must treat reserved bits as *Do-Not-Modify* (DNM) bits. For more information about the reserved bits, see The Current Program Status Register, CPSR.

These bits are UNKNOWN on a Read, and it is permitted that, on a read of APSR:

- Bit[22] returns the value of PSTATE.PAN.
- Bit[9] returns the value of PSTATE.E.
- Bits[8:6] return the value of PSTATE.{A, I, F}, the mask bits.
- Bits[4:0] return the value of PSTATE.M[4:0]. Bit[4] is RES1 indicating that the PE is in AArch32 state.

Note

This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GE[3:0], bits [19:16]

The PSTATE greater than or equal flags.

The other PSTATE fields cannot be accessed by using the APSR.

The system level alias for the APSR is the CPSR. The CPSR is a superset of the APSR. See The Current Program Status Register, CPSR.

Writes to the PSTATE fields have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the APSR write. This means they are guaranteed:

- Not to be visible to earlier instructions in the Execution stream.
- To be visible to later instructions in the Execution stream.

E1.2.4.1.2 The SETEND instruction

The T32 and A32 instruction sets both include an instruction to manipulate PSTATE.E:

`SETEND BE` Sets PSTATE.E to 1, for big-endian operation.

`SETEND LE` Sets PSTATE.E to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see SETEND. Arm deprecates use of the `SETEND` instruction.

E1.2.4.2 Use of PSTATE.IT

PSTATE.IT provides the If-Then controls for the T32 `IT` instruction, which applies to the *IT block* of instructions that immediately follow the `IT` instruction.

PSTATE.IT divides into two subfields:

IT[7:5] Holds the *base condition* for the current IT block. The base condition is the top three bits of the Condition code specified by the `<firstcond>` field of the `IT` instruction.

IT[4:0] Encodes:

- Implicitly, the size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is indicated by the position of the least significant 1 in this field, as shown in Table E1-2.
- For each instruction in the IT block, the least significant bit of the Condition code. This is encoded in the IT block entries that Table E1-2 shows as N*x*.

Note

Changing the least significant bit of a Condition code from 0 to 1 has the effect of inverting the Condition code.

Both subfields are all zeros when no IT block is active.

When an `IT` instruction is executed, PSTATE.IT is set according to the `<firstcond>` field of the instruction and the *Then* and *Else* (T and E) parameters in the instruction, see IT. This means that, on executing an IT instruction, the initial state of PSTATE.IT depends on the number of instructions in the IT block, as Table E1-2 shows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table E1-2 Initial state of PSTATE.IT on executing an IT instruction

Number of instructions in IT block	PSTATE.IT bits[a]						Notes
	[7:5]	[4]	[3]	[2]	[1]	[0]	
4	cond_base	N1	N2	N3	N4	1	-
3	cond_base	N1	N2	N3	1	0	-
2	cond_base	N1	N2	1	0	0	-
1	cond_base	N1	1	0	0	0	-
Not executing an `IT` instruction	000	0	0	0	0	0	No IT block is active

a. Combinations of the IT bits not shown in this table are reserved.

In Table E1-2, N1 refers to the first instruction in the IT block, and N2, N3, and N4 refer to the second, third, and fourth instructions in the IT block if they are present,

When permitted, an instruction in an IT block is conditional, see Conditional instructions and Conditional execution. The Condition code used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, PSTATE.IT[4:0] is left-shifted by one bit, so that PSTATE[4] always relates to the next instruction to be executed.

Table E1-3 shows how PSTATE.IT during the execution of an `IT` instruction with four instructions in the IT block.

Table E1-3 Updates to PSTATE.IT when executing an IT instruction with a four-instruction IT block

IT block instruction being executed	PSTATE.IT bits[a]						Notes
	[7:5]	[4]	[3]	[2]	[1]	[0]	
First	cond_base	N1	N2	N3	N4	1	-
Second	cond_base	N2	N3	N4	1	0	-
Third	cond_base	N3	N4	1	0	0	-
Fourth	cond_base	N4	1	0	0	0	-
Not executing an `IT` instruction	000	0	0	0	0	0	No IT block is active

a. Combinations of the IT bits not shown in this table are reserved.

A few instructions, for example `BKPT`, cannot be conditional and therefore are always executed ignoring the current value of PSTATE.IT.

For details of what happens if an instruction in an IT block takes an exception, see Overview of exception entry.

An instruction that might complete its normal execution by branching is only permitted in an IT block as the last instruction in the block. This means that normal execution of the instruction always results in PSTATE.IT advancing to execution where no IT block is active.

Implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD, to disable use of IT for some instructions, making them UNDEFINED. When an implementation includes ITD control fields, Changes to an ITD control by an instruction in an IT block describes the permitted CONSTRAINED UNPREDICTABLE behaviors if an instruction in an IT block changes the value of an ITD control to disable the use of the `IT` instruction.

On a branch or an exception return, if PSTATE.IT is set to a value that is not consistent with the instruction stream being branched to or returned to, then instruction execution is CONSTRAINED UNPREDICTABLE.

PSTATE.IT affects instruction execution only in T32 state. In A32 state, PSTATE.IT must be `0b00000000`, otherwise the behavior is CONSTRAINED UNPREDICTABLE.

For more information, see CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.2.4.2.1 Changes to an ITD control by an instruction in an IT block

In an implementation that includes SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD controls, if an instruction in an IT block changes an ITD control so that the `IT` instruction using the IT block would be disabled, then one of the following behaviors applies:

- The change to the ITD field, once synchronized, has no effect on the execution of instructions in the current IT block, but applies only to any subsequent execution of an `IT` instruction to which the control applies.
- Synchronizing the change to the ITD field guarantees that all bits of PSTATE.IT are cleared to 0.

In addition, after the change to the ITD field has been synchronized, any remaining instructions in the IT block that would be made UNDEFINED by the new value of ITD are either:

- Executed normally.
- Treated as UNDEFINED.

The choice between the options described in this section is determined by the implementation, and any choice can vary between different changes to an ITD control by an instruction in an IT block.

E1.2.4.3 Pseudocode description of PSTATE PE state fields

The pseudocode function `CurrentInstrSet()` returns the current instruction set. The pseudocode function `SelectInstrSet()` selects a new instruction set.

PSTATE.IT advances after normal execution of an IT block instruction. This is described by the `AArch32_ITAdvance()` pseudocode function.

The pseudocode function `InITBlock()` tests whether the current instruction is in an IT block. The pseudocode function `LastInITBlock()` tests whether the current instruction is the last instruction in an IT block.

The `BigEndian()` pseudocode function tests whether big-endian data memory accesses are currently selected.

E1.2.5 About the DIT bit

A *data-independent-time sequence of code* is the sequence of instructions executed from the first instruction executed after a change of CPSR.DIT from 0 to 1 through the last instruction executed before a change of CPSR.DIT from 1 to 0, inclusive. Instructions speculatively executed between the change of CPSR.DIT from 0 to 1 and the change of CPSR.DIT from 1 to 0 are considered part of the data-independent-time sequence of code.

A *data-independent-time resource* is any of:

- a general-purpose register other than the zero register (ZR)
- Stack Pointer register
- SIMD&FP register
- the NZCV flags

Note

The collection of NZCV is considered a single data-independent-time resource.

A *data-independent-time instruction* is an instruction that is required to honor the data-independent timing behaviors described in this section.

A *data-independent-time* value is any of:

- Any given value (V) read from memory by the Explicit Memory Read Effect of a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C), where V is not consumed by any other instruction (I2) for which any of the following apply:
  - I2 is not a data-independent-time instruction.
  - I2 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Any given value (V) written to a data-independent-time resource by a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C), where V is not consumed by any other instruction (I2) for which any of the following apply:
  - I2 is not a data-independent-time instruction.
  - I2 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
- Any given value (V) read from a data-independent-time resource by a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C) and none of the following apply:
  - V was produced by another instruction (I0) and any of the following apply:
    - I0 is not a data-independent-time instruction.
    - I0 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect produced by I1.
  - V is not consumed by any other instruction (I2) for which any of the following apply:
    - I2 is not a data-independent-time instruction.
    - I2 is not executed in C.
    - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
- Any given value (V) written to memory by the Explicit Memory Write Effect of a data-independent-time instruction (I1) that is executed within a data-independent-time sequence of code (C) and none of the following apply:
  - V was produced by another instruction (I0) and any of the following apply:
    - I0 is not a data-independent-time instruction.
    - I0 is not executed in C.
  - V is used to form the address or Logical Address Tag of a Memory Effect produced by I1.
  - V was sourced from the zero register.
  - V is not consumed by any other instruction (I2) for which any of the following apply:
    - I2 is not a data-independent-time instruction.
    - I2 is not executed in C.
    - V is used to form the address or Logical Address Tag of a Memory Effect generated by I2.
- Any given value (V) speculatively read from memory at a Location (L) as the result of executing a data-independent-time sequence of code, where L is not explicitly read by the data-independent-time sequence of code. Examples of this include:
  - Reading additional bytes from memory while reading the bytes required by an Explicit Memory Read Effect.
  - Hardware-generated prefetches.
  - Prefetch memory instructions.

The execution time of a given data-independent-time sequence of code must be independent of all data-independent-time values.

The execution time of a data-independent-time instruction (I1) executed within a data-independent-time sequence of code must be independent of all data-independent-time values consumed or produced by I1.

The time to respond to an asynchronous exception taken from within a data-independent-time sequence of code must be independent of all data-independent-time values.

The execution of a data-independent-time sequence of code (C1) must not cause an irreversible change in microarchitectural state of the PE that would permit a sequence of code (C2) that is executed after C1 to determine any data-independent-time value accessed during the execution of C1. For the purposes of this requirement, a data-independent-time value written by an Explicit Memory Write Effect that is held in caches is not considered an irreversible change in microarchitectural state.

The execution of a data-independent-time sequence of code (C1) must not alter the state of PMU counters in a way that allows a sequence of code (C2) that is executed after C1 to determine any data-independent-time value accessed during C1.

Note

In cases where an instruction is architecturally required to only read or write a subset of a data-independent-time resource, the entire data-independent-time resource is considered to have been read or written for the sake of these requirements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The architecture does not guarantee that the execution time of a data-independent-time sequence of code is independent of data-independent-time values if those values are read from or written to a peripheral.

Note

The classification of a value being a data-independent-time value due to having been read from a memory location (L) by an Explicit Memory Read Effect or written to L by an Explicit Memory Write Effect is based on the particular Effect generated by a data-independent-time instruction executed from within a data-independent-time sequence of code. The value held in L is not itself considered a data-independent-time value. As a result, it is possible for the execution time of a data-independent-time sequence of code to vary based on the value held in L if L is accessed via any of the following:

- Outside the data-independent-time sequence of code; either architecturally or speculatively.
- Within a data-independent-time sequence of code but not as a data-independent-time value.

Note

When the value of CPSR.DIT is 0, the architecture makes no statement about the timing properties of any instructions.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

When an exception is taken from AArch32 state to AArch32 state, CPSR.DIT is copied to SPSR.DIT.

When an exception is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

When an exception returns to AArch32 state from AArch32 state, SPSR.DIT is copied to CPSR.DIT.

When an exception returns to AArch32 state from AArch64 state, SPSR_ELx.DIT is copied to CPSR.DIT.

CPSR.DIT bit can be written using an `MSR` instruction at any Exception Level in AArch32 state, and read using an `MRS` instruction at any Exception Level.

E1.2.6 Jazelle support

The architecture requires AArch32 state to include a trivial implementation of the Jazelle Extension, as described in Trivial implementation of the Jazelle extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3 Advanced SIMD and floating-point instructions

In general, the architecture requires implementation of Advanced SIMD and floating-point instructions in the T32 and A32 instruction sets, but see Implications of not including Advanced SIMD and floating-point support.

The Advanced SIMD instructions perform packed *Single Instruction Multiple Data* (SIMD) operations, on either integer or single-precision floating-point operands. The floating-point instructions perform single-precision or double-precision scalar floating-point operations. When FEAT_FP16 is implemented, half-precision floating-point can also be used for data processing.

These instructions permit *floating-point exceptions*, such as Overflow or Divide by Zero, to be handled without trapping. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. The architecture also optionally supports the trapping of floating-point exceptions. For more information about floating-point exceptions, see Floating-point exceptions and exception traps.

The Advanced SIMD and floating-point instructions also provide the following conversion functions:

- Between half-precision floating-point and single-precision floating point, in both directions.
- From double-precision, floating-point to single-precision floating point or integer.
- When FEAT_AA32BF16 is implemented, between single-precision floating-point and BFloat16 floating-point.

Some Advanced SIMD instructions support polynomial arithmetic over {0, 1}, as described in Polynomial arithmetic over {0, 1}.

For system level information about the Advanced SIMD and floating-point implementation, see Advanced SIMD and floating-point support.

The following sections give more information about the Advanced SIMD and floating-point instructions:

- Floating-point standards, and terminology.
- The SIMD and floating-point register file.
- Data types supported by the Advanced SIMD implementation.
- Advanced SIMD and floating-point System registers.
- Floating-point data types and arithmetic.
- Flushing denormalized numbers to zero.
- Floating-point exceptions and exception traps.
- Controls of Advanced SIMD operation that do not apply to floating-point operation.
- Implications of not including Advanced SIMD and floating-point support.
- Pseudocode description of floating-point operations.

E1.3.1 The SIMD and floating-point register file

The Advanced SIMD and floating-point instructions use the same register file, which comprises 32 registers. This is distinct from the register file that holds the general-purpose registers and the PC.

The Advanced SIMD and floating-point views of the register file are different. The following sections describe these different views. Figure E1-1 shows the views of the register file, and the way the word, doubleword, and quadword registers overlap.

E1.3.1.1 Advanced SIMD views of the register file

Advanced SIMD can view this register file as:

- Sixteen 128-bit quadword registers, `Q0-Q15`.
- Thirty-two 64-bit doubleword registers, `D0-D31`.

These views can be used simultaneously. For example, a program might hold 64-bit vectors in `D0` and `D1` and a 128-bit vector in `Q1`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.1.2 Floating-point views of the register file

The Advanced SIMD and floating-point register file consists of thirty-two doubleword registers, which can be viewed as:

- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available to Advanced SIMD instructions.
- Thirty-two 32-bit single word registers, `S0-S31`. Only half of the set is accessible in this view.

Note

In AArch32 state, half-precision floating point values are always represented using the bottom 16 bits of a single word register, `S0-S31`. When a half-precision value is written to a single word register, the top 16 bits of that register are set to 0.

The two views can be used simultaneously.

E1.3.1.3 SIMD and Floating-point register file mapping onto registers

Figure E1-1 shows the different views of the SIMD and floating-point register file, and the relationship between them.

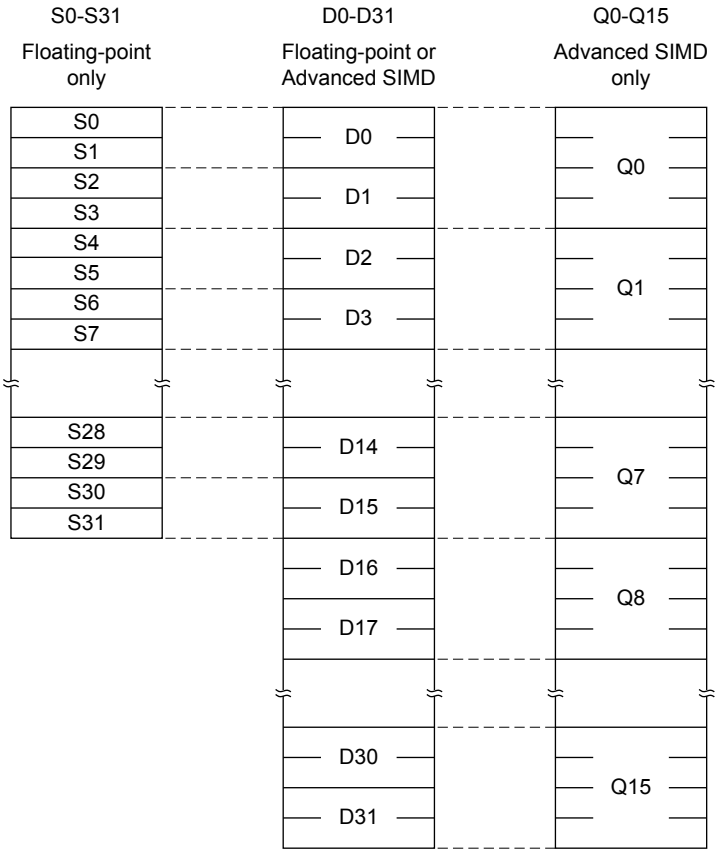

Figure E1-1 SIMD and floating-point register file, AArch32 operation

The mapping between the registers is as follows:

- `S<2n>` maps to the least significant half of `D<n>`.
- `S<2n+1>` maps to the most significant half of `D<n>`.
- `D<2n>` maps to the least significant half of `Q<n>`.
- `D<2n+1>` maps to the most significant half of `Q<n>`.

For example, software can access the least significant half of the elements of a vector in `Q6` by referring to `D12`, and the most significant half of the elements by referring to `D13`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.1.4 Pseudocode description of the SIMD and Floating-point register file

The array `_Dclone`, accessors `S()`, and `D()` provide the `S0-S31`, `D0-D31`, and `Q0-Q15` views of the Advanced SIMD and floating-point registers. The array `_Dclone` is used to hold a copy of the Advanced SIMD and VFP registers.

The `Din()` function returns a doubleword register from the `_Dclone` copy of the SIMD and Floating-point register file, and the `Qin()` function returns a quadword register from that register file.

Note

The `CheckAdvSIMDEnabled()` function copies the `D()` register file to `_Dclone`, see Pseudocode description of enabling SIMD and floating-point functionality.

E1.3.2 Data types supported by the Advanced SIMD implementation

Advanced SIMD instructions can operate on integer and floating-point data, and the implementation defines a set of data types that support the required data formats. Advanced SIMD vector formats in AArch32 state describes these formats.

E1.3.2.1 Advanced SIMD vectors

In an implementation that includes support for Advanced SIMD operation, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements - 1). An index of 0 refers to the least significant end of the vector. In Advanced SIMD vector formats in AArch32 state, Figure A1-3 shows the Advanced SIMD vector formats.

E1.3.3 Advanced SIMD and floating-point System registers

The Advanced SIMD and floating-point instructions have a shared register space for System registers. The only register in this space that is accessible at the Application level is the FPSCR.

Writes to the FPSCR can have side-effects on various aspects of PE operation. All of these side-effects are synchronous to the FPSCR write. This means they are guaranteed not to be visible to earlier instructions in the Execution stream, and they are guaranteed to be visible to later instructions in the Execution stream.

See Advanced SIMD and floating-point System registers for the system level view of the registers.

These registers can be described as the *SIMD and floating-point System registers*.

E1.3.4 Floating-point data types and arithmetic

The T32 and A32 floating-point instructions support single-precision (32-bit) and double-precision (64-bit) data types and arithmetic as defined by the IEEE 754 floating-point standard. They also support the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types. When FEAT_FP16 is implemented, it also supports the half-precision floating-point data type for data processing operations. When FEAT_AA32BF16 is implemented, it also supports the BFloat16 floating-point storage format.

Arm standard floating-point arithmetic means IEEE 754 floating-point arithmetic with the restrictions described in Floating-point support, including supporting only the input and output values described in Arm standard floating-point input and output values.

The AArch32 Advanced SIMD instructions support single-precision and, when FEAT_FP16 is implemented, half-precision Arm standard floating-point arithmetic.

The following sections describe the Advanced SIMD and floating-point formats:

- Half-precision floating-point formats.
- Single-precision floating-point format.
- Double-precision floating-point format.
- BFloat16 floating-point format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following sections describe features of Advanced SIMD and floating-point processing:

- Flushing denormalized numbers to zero.
- NaN handling and the Default NaN.

E1.3.5 Flushing denormalized numbers to zero

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and Intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization. If a number satisfies the condition `0 < Abs(result) < MinNorm`, it is treated as a denormalized number.

`MinNorm` is defined as follows:

- For half-precision numbers, `MinNorm` is 2^{-14}.
- For single-precision and BFloat16 numbers, `MinNorm` is 2^{-126}.
- For double-precision numbers, `MinNorm` is 2^{-1022}.

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.
- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

E1.3.5.1 Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has an input denormalized number that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For Advanced SIMD and floating-point instructions, other than `FABS` and `FNEG`, that process half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If FPSCR.FZ16 is 0, denormalized half-precision inputs are not flushed to zero.
- If FPSCR.FZ16 is 1, flushing denormalized inputs to zero occurs as follows:
  - If an instruction does not convert a half-precision input to a higher precision output, all input denormalized numbers are flushed to zero.
  - If an instruction converts a half-precision input to a higher precision output, input denormalized numbers are not flushed to zero.

For Advanced SIMD and scalar floating-point instructions, other than `FABS` and `FNEG`, that process single-precision, or double-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If FPSCR.FZ is 0, flushing denormalized inputs to zero occurs as follows:
  - For Advanced SIMD floating-point instructions, all single-precision and double-precision inputs that are denormalized numbers are flushed to zero.
  - For scalar floating-point instructions, single-precision and double-precision inputs that are denormalized numbers are not flushed to zero.
- If FPSCR.FZ is 1, for all T32 and A32 instructions, single-precision and double-precision inputs that are denormalized numbers are flushed to zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_AA32BF16 is implemented, for Advanced SIMD and scalar floating-point instructions, other than `FABS` and `FNEG`, that process BFloat16 inputs, flushing denormalized inputs to zero is treated as follows:

- Instructions that convert from single-precision floating-point values to BFloat16 format flush denormalized inputs to zero.
- For any value of FPSCR.FZ, VDOT (vector), VDOT (by element), and VMMLA instructions flush all BFloat16 inputs that are denormalized numbers to zero.

E1.3.5.2 Flushing to zero of denormalized numbers as Intermediate results of some BFloat16 instructions

BFloat16 arithmetic instructions VDOT (by element), VDOT (vector), and VMMLA, convert BFloat16 input values to IEEE single-precision format, and calculate N-way dot-products, accumulating the products in single-precision accumulators.

If FEAT_AA32BF16 is implemented, for Advanced SIMD and floating-point instructions, if a BFloat16 arithmetic instruction processes an Intermediate result that is a single-precision denormalized number, the Intermediate result is unconditionally flushed to zero.

E1.3.5.3 Flushing denormalized outputs to zero

If flushing denormalized outputs to zero is enabled for an instruction and a data type, and an output from that instruction is a denormalized number of that data type, the output operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has an output denormalized number that is flushed to zero, for all purposes within the instruction other than calculating floating-point exceptions, all outputs that are denormalized numbers are treated as though they were zero with the same sign as the output.

For Advanced SIMD and floating-point instructions, other than `FABS` and `FNEG`, that generate half-precision outputs, flushing denormalized outputs to zero can be controlled as follows:

- If FPSCR.FZ16 is 0, denormalized half-precision outputs are not flushed to zero.
- If FPSCR.FZ16 is 1, flushing denormalized outputs to zero occurs as follows:
  - If the instruction does not convert a half-precision input to a higher precision output, all output denormalized numbers are flushed to zero.
  - If the instruction converts a half-precision input to a higher precision output, output denormalized numbers are not flushed to zero.

For Advanced SIMD and scalar floating-point instructions, other than `FABS` and `FNEG`, that process single-precision, or double-precision inputs, flushing denormalized outputs to zero can be controlled as follows:

- If FPSCR.FZ is 0, flushing denormalized outputs to zero occurs as follows:
  - For Advanced SIMD floating-point instructions, all single-precision and double-precision outputs that are denormalized numbers are flushed to zero.
  - For scalar floating-point instructions, single-precision and double-precision outputs that are denormalized numbers are not flushed to zero.
- If FPSCR.FZ is 1, for all T32 and A32 instructions, single-precision and double-precision outputs that are denormalized numbers are flushed to zero.

If FEAT_AA32BF16 is implemented, for Advanced SIMD and scalar floating-point instructions, other than `FABS` and `FNEG`, that generate BFloat16 outputs, flushing denormalized outputs to zero can be controlled as follows:

- BFloat16 arithmetic instructions flush denormalized outputs to zero.
- If FPSCR.FZ is 1, instructions that convert from single-precision floating-point values to BFloat16 format flush denormalized outputs to zero.
- VDOT (vector), VDOT (by element), and VMMLA instructions flush all BFloat16 outputs that are denormalized numbers to zero regardless of the value of FPSCR.FZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.6 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the mantissa, and specifies that the sign bit of a NaN has no significance.

The Arm architecture additionally defines:

- A BFloat16 NaN, that follows the encoding in BFloat16 floating-point format.
- A *Default NaN*, compliant with the IEEE 754 standard, that follows the encoding in Table E1-4.

For a Quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a speculative result of a floating-point calculation not involving NaN inputs.

E1.3.6.1 The Default NaN

The Default NaN is encoded as described in Table E1-4.

Table E1-4 Default NaN encoding

	Half-precision, IEEE format	Single-precision	Double-precision	BFloat16
Sign bit	0	0	0	0
Exponent	`0x1F`	`0xFF`	`0x7FF`	`0xFF`
Fraction	Bit[9] == 1, bits[8:0] == 0	Bit[22] == 1, bits[21:0] == 0	Bit[51] == 1, bits[50:0] == 0	Bit[6] == 1, bits[5:0] == 0

If FPSCR.DN is 1, for Advanced SIMD and floating-point instructions other than `FABS`, `FMAX*`, `FMIN*` and `FNEG`, if any input to a floating-point operation performed by the instruction is a NaN, the output of the floating-point operation is the Default NaN.

For `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, Default NaN behavior is explained in the instruction description.

If FPSCR.DN is 0, for floating-point processing the Default NaN is not used for NaN propagation.

If VDOT (vector), VDOT (by element), and VMMLA instructions generate a NaN, the NaN is the default NaN. regardless of the setting of FPSCR.DN.

If a floating-point instruction performs a floating-point operation, and that instruction generates an untrapped Invalid Operation floating-point exception for a reason other than one of the inputs being a signaling NaN, the output is the Default NaN.

E1.3.6.2 NaN handling

The IEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived Quiet NaN output might have both a different sign and a different number of fraction bits from its source. See instruction descriptions for details.

E1.3.6.3 NaN propagation

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the instruction and the number of operands.

If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

- If the input NaN is a Quiet NaN, the output NaN is the same as the input NaN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the input NaN is a signaling NaN, the output NaN is derived as follows:
  - If the handling of a signaling NaN by the instruction generates an Invalid Operation exception, the output NaN is the quieted version of the input NaN.
  - If the handling of a signaling NaN by the instruction does not generate an Invalid Operation exception, the output NaN is the same as the input NaN. This case applies for FABS, FNEG, and FTSSEL instructions.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

- If the input NaN is a Quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

- If the input NaN is a Quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, when an operand is described as *first* this relates to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If FPSCR.DN is 0, for Advanced SIMD, floating-point, or BFloat16 instructions that perform a floating-point operation, other than `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, NaN outputs that derive from NaN inputs are derived as follows:

- If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:
  - At least one operand is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, an instruction outputs a Quiet NaN derived from the first NaN operand:
  - At least one operand is a NaN, but none of the operands is a signaling NaN.
  - The instruction is not trapped.

If an output NaN is derived from an input NaN, the pseudocode functions `FPAbs()`, and `FPNeg()` can change the sign of the NaN,

E1.3.7 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPSCR.RMode or by the instruction.

The rounding mode control field FPSCR.RMode can select the following rounding modes:

- Round to Nearest (RN) mode.
- Round towards Plus Infinity (RP) mode.
- Round towards Minus Infinity (RM) mode.
- Round towards Zero (RZ) mode.

The following two additional rounding modes are not selected by FPSCR.RMode, but are used by some instructions:

- Round to Odd mode.
- Round to Nearest with ties to away mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.7.1 Round to Nearest mode

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

E1.3.7.2 Round towards Plus Infinity mode

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

E1.3.7.3 Round towards Minus Infinity mode

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

E1.3.7.4 Round towards Zero mode

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

E1.3.7.5 Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounding mode is used by the VCVTA (Advanced SIMD), VCVTA (floating-point), VRINTA (Advanced SIMD), and VRINTA (floating-point) instructions.

Round to Nearest with Ties to Away rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity, the sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the larger number.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

E1.3.7.6 Round to Odd mode

Round to Odd mode is not defined by IEEE 754.

For BFloat16 instructions, if an intermediate format has at least two more bits of precision than the result format, Round to Odd mode is used and operates as follows:

- If the rounded value is inexact, the least significant bit of the fraction is set to 1.
- If the value is too large to represent in the single-precision format, the rounded value is a single-precision Infinity, the sign of the rounded value is the same as the sign of the value before rounding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.8 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*.

Note

An Advanced SIMD instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an *operation*, rather than on an *instruction*.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated, and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

If an implementation supports synchronous exception generation for a floating-point exception, then the registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception.

On return from a synchronous floating-point exception, software might not restore the cumulative exception flags.

Trapped floating-point exceptions are taken to the following levels:

- If a trapped floating-point exception occurs at EL0, the exception level it is taken to is as follows:
  - If EL2 is using AArch32 and HCR.TGE is 1, the exception is taken to EL2.
  - If EL2 is using AArch64 and HCR_EL2.TGE is 1, the exception is taken to EL2
  - Otherwise, the exception is taken to EL1
- If a trapped floating-point exception occurs at EL1, it is taken to EL1.
- If a trapped floating-point exception occurs at EL2, it is taken to EL2.
- If a trapped floating-point exception occurs at EL3, it is taken to EL3.

If a trapped floating-point exception is taken to an Exception level that is using AArch64, then it is reported in the ELR_ELx for the target Exception level, as described in Exception entry.

If the exception is taken to an Exception level that is using AArch32, then it is taken as an Undefined Instruction exception, see Undefined Instruction exception. The FPEXC identifies the floating-point exceptions that occurred since the corresponding status bits in that register were last set to 0.

E1.3.8.1 Input Denormal exceptions

The cumulative floating-point exception bit FPSCR.IDC, and the trap enable bit FPSCR.IDE both relate to Input Denormal exceptions.

If a single-precision or double-precision floating-point input is flushed to zero, an Input Denormal exception is generated.

If a half-precision floating-point value is flushed to zero, an Input Denormal exception is not generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.8.2 Inexact exceptions

The cumulative floating-point exception bit FPSCR.IXC, and the trap enable bit FPSCR.IXE both relate to Inexact exceptions.

If a denormalized output is flushed to zero, an Inexact exception is not generated.

If a result is not flushed to zero, and the result does not equal the result calculated with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

E1.3.8.3 Underflow exceptions

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPSCR.UFE both relate to Underflow exceptions.

For the purpose of Underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then the Underflow exception is generated as follows:

- If the result is exact, an Underflow floating-point exception is generated only if FPSCR.UFE is 1.
- If the result is inexact, an Underflow floating-point exception is always generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is generated. The Underflow exception is not trapped regardless of the value of FPSCR.UFE.

E1.3.8.4 Overflow exceptions

The cumulative floating-point exception bit FPSCR.OFC, and the trap enable bit FPSCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an Overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the rounding mode and the sign of the result before rounding as follows:

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision's largest finite number with the sign of the result before rounding.

E1.3.8.5 Divide by Zero exceptions

The cumulative floating-point exception bit FPSCR.DZC, and the trap enable bit FPSCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite nonzero number by zero, a Divide by Zero exception is generated.

For the purpose of Divide by Zero exception generation, testing for zero occurs after flushing of denormalized numbers to zero.

A denormalized dividend that is flushed to zero is treated as zero and prevents Divide by Zero from occurring.

If the dividend is a finite nonzero, normalized number, and the divisor is a denormalized number, the divisor is treated as zero and causes Divide by Zero to occur.

For the reciprocal and reciprocal square root estimate functions, the dividend is assumed to be +1.0. This means that a zero or denormalized operand to these functions causes generation of a Divide by Zero floating-point exception.

If a floating-point operation divides a finite nonzero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.8.6 Invalid Operation exceptions

The cumulative floating-point exception bit FPSCR.IOC, and the trap enable bit FPSCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN, and the instruction is not `FABS` or `FNEG`.
- Subtracting two equal-magnitude infinities that are of the same sign.
- Multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Taking the square root of an operand that is less than zero.

For the purpose of Invalid Operation Exception generation, testing for zero occurs after flushing of denormalized numbers to zero. So a denormalized input that is flushed to zero is treated as zero.

If the input is one of: a Quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation exception.

For the signaling compare instructions FCMPE and FCCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

E1.3.8.7 Floating-point exception traps

For Advanced SIMD instructions, and for floating-point instructions when floating-point exception trapping is not supported, these are non-trapping exceptions and the data-processing instructions do not generate any trapped exceptions.

For floating-point instructions when floating-point exception trapping is supported:

- The floating-point exceptions can be trapped, by setting trap enable bits in the FPSCR, see Floating-point exceptions and exception traps, and:
  - When a trap is not enabled the corresponding floating-point exception updates the corresponding FPSCR cumulative bit, but does not generate an exception.
  - When a trap is enabled the corresponding floating-point exception does not update the FPSCR, but generates an exception. In this case, bits in the FPEXC indicate which floating-point exceptions have occurred.
- The definition of the Underflow floating-point exception is different in the trapped and cumulative exception cases. In the trapped case, the definition is:
  - The trapped Underflow floating-point exception occurs if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, regardless of whether the rounded result is inexact.
- As with cumulative exceptions, higher priority trapped exceptions can prevent lower priority exceptions from occurring, as described in Combinations of floating-point exceptions.
- For Invalid Operation floating-point exceptions, for details of which Quiet NaN is produced as the default result, see NaN handling and the Default NaN.
- For Overflow floating-point exceptions, the sign bit of the default result is determined normally for the overflowing operation.
- For Divide by Zero floating-point exceptions, the sign bit of the default result is determined normally for a division. This means it is the exclusive-OR of the sign bits of the two operands.

Table E1-5 shows the results of untrapped floating-point exceptions. That table uses the following abbreviations:

MaxNorm The maximum normalized number of the destination precision.

RM Round towards Minus Infinity mode, as defined in the IEEE 754 standard.

RN Round to Nearest mode, as defined in the IEEE 754 standard.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RP Round towards Plus Infinity mode, as defined in the IEEE 754 standard.

RZ Round towards Zero mode, as defined in the IEEE 754 standard.

For more information about the IEEE 754 descriptions of the rounding modes, see Floating-point standards, and terminology.

Table E1-5 Results of untrapped floating-point exceptions

Exception type	Default result for positive sign		Default result for negative sign	
IOC, Invalid Operation	Quiet NaN		Quiet NaN	
DZC, Divide by Zero	+infinity		-infinity	
OFC, Overflow	RN, RP: RM, RZ:	+infinity +MaxNorm	RN, RM: RP, RZ:	-infinity -MaxNorm
UFC, Underflow	Normal rounded result		Normal rounded result	
IXC, Inexact	Normal rounded result		Normal rounded result	
IDC, Input Denormal	Normal rounded result		Normal rounded result	

E1.3.8.8 Combinations of floating-point exceptions

Many pseudocode functions perform *floating-point operations*, including `FixedToFP()`, `FPAdd()`, `FPCompare()`, `FPCompareEQ()`, `FPCompareGE()`, `FPCompareGT()`, `FPDiv()`, `FPMax()`, `FPMin()`, `FPMul()`, `FPMulAdd()`, `FPRecipEstimate()`, `FPRecipStep()`, `FPRSqrtEstimate()`, `FPRSqrtStep()`, `FPSqrt()`, `FPSub()`, and `FPToFixed()`. All of these operations can generate floating-point exceptions.

Note

`FPAbs()` and `FPNeg()` are not classified as *floating-point operations* because:

- They cannot generate floating-point exceptions.
- The floating-point operation behavior described in the following sections does not apply to them:
  - Flushing denormalized numbers to zero.
  - NaN handling and the Default NaN.

More than one exception might occur on the same operation. The architecture permits only the following combinations of floating-point exceptions:

- Overflow with Inexact.
- Underflow with Inexact.
- Input Denormal with one or more of the other floating-point exception types.

The priority order of these floating-point exceptions is that the Inexact exception is treated as lowest priority, and the Input Denormal exception is treated as highest priority.

When none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative bit in the FPSCR to be set.

When one or more floating-point exceptions caused by an operation is trapped, the behavior of the instruction depends on the priority of the exceptions:

- If the higher priority floating-point exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether any information about the lower priority floating-point exception is provided.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Information about the lower priority floating-point exception might be provided in:

— The FPEXC, if the exception generated by the trap is taken to an Exception level that is using AArch32.

— The ESR_ELx.ISS field, if the exception generated by the trap is taken to an Exception level that is using AArch64.

However, information might be provided in another IMPLEMENTATION DEFINED way, for example using an IMPLEMENTATION DEFINED register.

Apart from this, the lower priority floating-point exception is ignored in this case.

- If the higher priority floating-point exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the lower priority floating-point exception is handled normally, using this default result.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, a floating-point exception on one operation is treated as higher priority than a floating-point exception on another operation if the occurrence of the second floating-point exception depends on the result of the first operation. Otherwise, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority.

For example, a `VMLA.F32` instruction specifies a floating-point multiplication followed by a floating-point addition. The addition can generate Overflow, Underflow and Inexact floating-point exceptions, all of which depend on both operands to the addition and so are treated as lower priority than any floating-point exception on the multiplication. The same applies to Invalid Operation floating-point exceptions on the addition caused by adding opposite-signed infinities. The addition can also generate an Input Denormal floating-point exception, caused by the addend being a denormalized number while in Flush-to-zero mode. It is CONSTRAINED UNPREDICTABLE which of an Input Denormal floating-point exception on the addition and a floating-point exception on the multiplication is treated as higher priority, because the occurrence of the Input Denormal floating-point exception does not depend on the result of the multiplication. The same applies to an Invalid Operation floating-point exception on the addition caused by the addend being a signaling NaN.

Note

The `VFMA` instruction performs a vector addition and a vector multiplication as a single operation. The `VFMS` instruction performs a vector subtraction and a vector multiplication as a single operation.

E1.3.9 Controls of Advanced SIMD operation that do not apply to floating-point operation

Earlier architectures permitted implementation of either, both, or neither of the Advanced SIMD and floating-point additions to the base instruction set, and provided some controls that applied to the Advanced SIMD functionality but not to the floating-point functionality. From the introduction of Armv8, Advanced SIMD functionality cannot be separated from floating-point functionality, but in AArch32 state these controls function as they did in earlier architectures. This means they apply only to the following instructions and instruction encodings:

- All instructions with encodings defined in:
  — Advanced SIMD data-processing, for the T32 instruction set.
  — Advanced SIMD data-processing, for the A32 instruction set.
- All instructions with encodings defined in:
  — Advanced SIMD element or structure load/store, for the T32 instruction set.
  — Advanced SIMD element or structure load/store, for the A32 instruction set.
- The form of the `VDUP` instruction described in VDUP (general-purpose register).
- The byte and halfword forms of the VMOV instructions described in each of:
  — VMOV (general-purpose register to scalar).
  — VMOV (scalar to general-purpose register).

The controls of this functionality are:

- The CPACR.ASEDIS field.
- The HCPTR.TASE field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation that supports Advanced SIMD functionality, support for each of these controls is optional:

- If the CPACR.ASEDIS control is not supported then the CPACR.ASEDIS field is RAZ/WI. This is equivalent to the control permitting the execution of Advanced SIMD instructions at EL1 and EL0.
- If the HCPTR.TASE control is not supported then the HCPTR.TASE field is RAZ/WI. This means the HCPTR does not provide a control that can trap Non-secure execution of Advanced SIMD instructions to Hyp mode.

E1.3.10 Implications of not including Advanced SIMD and floating-point support

In general, the architecture requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementation targeting specialized markets, Arm might produce or license an Armv8-A implementation that does not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- Each of the CPACR.{cp10, cp11} fields is RES0.
- The CPACR.ASEDIS bit is RES0.
- Each of the HCPTR.{TASE, TCP10, TCP11} fields is RES1.
- Each of the NSACR.{NSASEDIS, cp10, cp11} fields is RES0.
- The FPEXC register is UNDEFINED.

E1.3.11 Pseudocode description of floating-point operations

The following subsections contain pseudocode definitions of the floating-point functionality supported by the architecture:

- Generation of specific floating-point values.
- Floating-point negation and absolute value.
- Floating-point value unpacking.
- Floating-point exception and NaN handling.
- Floating-point rounding.
- Selection of Arm standard floating-point arithmetic.
- Floating-point comparisons.
- Floating-point maximum and minimum.
- Floating-point addition and subtraction.
- Floating-point multiplication and division.
- Floating-point fused multiply-add.
- Floating-point reciprocal estimate and step.
- Floating-point square root.
- Floating-point reciprocal square root estimate and step.
- Floating-point conversions.

E1.3.11.1 Generation of specific floating-point values

The following pseudocode functions generate specific floating-point values. The `sign` argument is `'0'` for the positive version and `'1'` for the negative version:

- `FPInfinity()`.
- `FPMaxNormal()`.
- `FPZero()`.
- `FPTwo()`.
- `FPThree()`.
- `FPDefaultNaN()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.11.2 Floating-point negation and absolute value

The floating-point negation and absolute value operations affect only the sign bit. They do not treat NaN operands specially, nor denormalized number operands when flush-to-zero is selected.

The floating-point negation operation is described by the pseudocode function `FPNeg()`. The floating-point absolute value operation is described by the pseudocode function `FPAbs()`.

E1.3.11.3 Floating-point value unpacking

The `FPUnpack()` function determines the type of a floating-point number, defined by `FPType{}`, and its numerical value. It also does flush-to-zero processing on input operands.

E1.3.11.4 Floating-point exception and NaN handling

The `FPProcessException()` procedure checks whether a floating-point exception is trapped, and handles it accordingly. The floating-point exception types are defined by `FPExc{}`.

The `FPProcessNaN()` function processes a NaN operand, producing the correct result value and generating an Invalid Operation floating-point exception if necessary. The `FPProcessNaNs()` function performs the standard NaN processing for a two-operand operation. The `FPProcessNaNs3()` function performs the standard NaN processing for a three-operand operation.

E1.3.11.5 Floating-point rounding

The `FPRound()` function rounds and encodes a floating-point result to a specified destination format. This includes processing Overflow, Underflow and Inexact floating-point exceptions and performing flush-to-zero processing on result values.

E1.3.11.6 Selection of Arm standard floating-point arithmetic

The `StandardFPCR()` function returns the FPSCR value that selects Arm standard floating-point arithmetic. Most of the arithmetic functions have a Boolean `fpscr_controlled` argument that is `TRUE` for Floating-point operations and `FALSE` for Advanced SIMD operations, and that selects between using the real FPSCR value and this value.

E1.3.11.7 Floating-point comparisons

The `FPCompare()` function compares two floating-point numbers, producing a {N, Z, C, V} Condition flags result as shown in Table E1-6:

Table E1-6 Effect of a Floating-point comparison on the Condition flags

Comparison result	N	Z	C	V
Equal	0	1	1	0
Less than	1	0	0	0
Greater than	0	0	1	0
Unordered	0	0	1	1

This result defines the operation of the `VCMP` floating-point instruction. The `VCMP` instruction writes these flag values in the FPSCR. After using a `VMRS` instruction to transfer them to the APSR, they can control conditional execution as shown in Table F1-1.

The `FPCompareEQ()`, `FPCompareGE()`, and `FPCompareGT()` functions describe the operation of Advanced SIMD instructions that perform floating-point comparisons.

E1.3.11.8 Floating-point maximum and minimum

The `FPMax()` function returns the maximum of two floating-point numbers. The `FPMin()` function returns the minimum of two floating-point numbers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.11.9 Floating-point addition and subtraction

The `FPAdd()` function adds two floating-point numbers. The `FPSub()` function subtracts one floating-point number from another floating-point number.

E1.3.11.10 Floating-point multiplication and division

The `FPMul()` function multiplies two floating-point numbers. The `FPDiv()` function divides one floating-point number by another floating-point number.

E1.3.11.11 Floating-point fused multiply-add

The `FPMulAdd()` function performs a floating-point fused multiply-add.

E1.3.11.12 Floating-point reciprocal estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of a number.

The `VRECPE` instruction produces the initial estimate of the reciprocal. It uses the pseudocode functions:

- `FPRecipEstimate()`.
- `UnsignedRecipEstimate()`.

Table E1-7 shows the results where input values are out of range.

Table E1-7 VRECPE results for out of range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	<= `0x7FFF_FFFF`	`0xFFFF_FFFF`
Floating-point	NaN	Default NaN
Floating-point	±0 or denormalized number	±infinity[a]
Floating-point	±infinity	±0
Floating-point	Absolute value >= 2^{126}	±0

a. FPSCR.DZC is set to 1

The Newton-Raphson iteration:

$x_{n+1} = x_n(2-dx_n)$

converges to $(1/d)$ if x_0 is the result of `VRECPE` applied to d.

The `VRECPS` instruction performs a (2 - op1 × op2) calculation and can be used with a multiplication to perform a step of this iteration. The functionality of this instruction is defined by the `FPRecipStep()` pseudocode function.

Table E1-8 shows the results where input values are out of range.

Table E1-8 VRECPS results for out of range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Input Vn[i]	Input Vm[i]	Result Vd[i]
±0.0 or denormalized number	±infinity	2.0
±infinity	±0.0 or denormalized number	2.0

E1.3.11.13 Floating-point square root

The `FPSqrt()` function returns the square root of a floating-point number.

E1.3.11.14 Floating-point reciprocal square root estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of the square root of a number.

The `VRSQRTE` instruction produces the initial estimate of the reciprocal of the square root. It uses the pseudocode functions:

- `FPRSqrtEstimate()`.
- `UnsignedRSqrtEstimate()`.

Table E1-9 shows the results where input values are out of range.

Table E1-9 VRSQRTE results for out of range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	<= `0x3FFF_FFFF`	`0xFFFF_FFFF`
Floating-point	NaN, -(normalized number), -infinity	Default NaN
Floating-point	-0 or -(denormalized number)	- infinity[a]
Floating-point	+0 or +(denormalized number)	+infinity[a]
Floating-point	+infinity	+0

a. FPSCR.DZC is set to 1.

The Newton-Raphson iteration:

$x_{n+1} = x_n(3-dx_n^2)/2$

converges to $(1/\sqrt{d})$ if x_0 is the result of `VRSQRTE` applied to d.

The `VRSQRTS` instruction performs a (3 - op1×op2)/2 calculation and can be used with two multiplications to perform a step of this iteration. The functionality of this instruction is defined by the `FPRSqrtStep()` pseudocode function.

Table E1-10 shows the results where input values are out of range.

Table E1-10 VRSQRTS results for out of range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
±0.0 or denormalized number	±infinity	1.5
±infinity	±0.0 or denormalized number	1.5

`FPRSqrtStep()` calls the `FPHalvedSub()` pseudocode function.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3.11.15 Floating-point conversions

The `FPConvert()` pseudocode function performs conversions between half-precision, single-precision, and double-precision floating-point numbers.

The `FPToFixed()` and `FixedToFP()` functions perform conversions between floating-point numbers and integers or fixed-point numbers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.4 About the AArch32 System register interface

AArch32 state provides a System register encoding space, which is indexed by the parameter set {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}, and a set of System register access instructions. This encoding space is used for:

- System registers.
- System instructions, for:
  - Cache and branch predictor maintenance.
  - Address translation.
  - TLB maintenance.

This encoding space uses only the `coproc` values `0b111x`.

Note

The encoding space with `coproc` values `0b101x` is redefined to provide Advanced SIMD and floating-point functionality.

The `coproc` encodings provide access to System register encoding space as follows:

- The (`coproc` ==`0b1111`) encodings provide system control functionality, by providing access to System registers and System instructions. This includes architecture and feature identification, as well as control, status information and configuration support.
  The following sections give a general description of these encodings:
  - About the System registers for VMSAv8-32.
  - Organization of registers in the (coproc == `0b1111`) encoding space.
  - Functional grouping of VMSAv8-32 System registers.

  These encodings also provide:
  - The Performance Monitor System registers. For more information, see The Performance Monitors Extension.
    The Activity Monitors System registers. For more information, see The Activity Monitors Extension.
- The (`coproc` ==`0b1110`) encodings provide access to additional registers, which support:
  - Debug, see AArch32 Self-hosted Debug.
  - The Jazelle identification registers, see Jazelle support.

UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses gives information more information about permitted accesses to the System registers in AArch32 state.

Most functionality in the (`coproc` ==`0b111x`) encoding space cannot be accessed by software executing at EL0. This manual clearly identifies those functions that can be accessed at EL0.

For more information:

- About this encoding space, including the naming of the parameters that index the space, see The AArch32 System register interface.
- About the System interface access instructions, see System register access instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.5 Exceptions

The Arm architecture uses the following terms to describe various types of exceptional condition:

Exceptions

In the Arm architecture, an *exception* causes entry to EL1, EL2, or EL3. If the Exception level that is entered is using AArch32, it also causes entry to the PE mode in which the exception must be taken. A software handler for the exception is then executed.

Note

The term *floating-point exception* does not use this meaning of *exception*. This term is described later in this list.

Exceptions include:

- Reset.
- Interrupts.
- Memory system aborts.
- Undefined instructions.
- Supervisor calls (SVCs), Secure Monitor calls (SMCs), and Hypervisor calls (HVCs).
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in Handling exceptions that are taken to an Exception level using AArch32. In an implementation that includes all the Exception levels, aspects that are visible to application level software are:

- The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a call to the operating system, or other system component that is accessible only at EL1.
- The `SMC` instruction causes a Secure Monitor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Secure Monitor Call exception by methods defined by the operating system, or by another component of the software system that executes at EL1 or higher.
- The `HVC` instruction causes a Hypervisor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Hypervisor Call exception by methods defined by the hypervisor, or by another component of the software system that executes at EL1 or higher.
- The `BKPT` instruction causes a Breakpoint Instruction exception, which is taken as a Prefetch Abort exception. This provides a mechanism for a debugger to insert breakpoints into unprivileged software, or for unprivileged software to make a call into a debugger that is accessible at EL1.
- The `WFI` (Wait for Interrupt) instruction provides a hint that nothing needs to be done until an interrupt or another WFI wakeup event occurs, see Wait For Interrupt. This means the hardware might enter a low-power state until the wakeup event occurs.
- The `WFE` (Wait for Event) instruction provides a hint that nothing needs to be done until either an `SEV` instruction generates an event, or another WFE wakeup event occurs, see Wait For Event and Send Event. This means the hardware might enter a low-power state until the wakeup event occurs.

Floating-point exceptions

These relate to exceptional conditions encountered during floating-point arithmetic, such as Divide by Zero or Overflow. For more information, see:

- Floating-point exceptions and exception traps.
- The FPEXC and FPSCR register descriptions.
- ANSI/IEEE Std. 754, *IEEE Standard for Binary Floating-Point Arithmetic*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E2
The AArch32 Application Level Memory Model

This chapter gives an application level description of the memory model for software executing in AArch32 state. This means it describes the memory model for execution in EL0 when EL0 is using AArch32 in the following sections:

- About the Arm memory model.
- Atomicity in the Arm architecture.
- Definition of the memory model.
- Ordering of translation table walks.
- Caches and memory hierarchy.
- Alignment support.
- Endian support.
- Memory types and attributes.
- Mismatched memory attributes.
- Synchronization and semaphores

Note

In this chapter, System register names usually link to the description of the register in AArch32 System Register Descriptions, for example SCTLR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be reordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.

This section contains:

- Address space.
- Memory type overview.

E2.1.1 Address space

Address calculations are performed using 32-bit registers. Supervisory software determines the valid address range.

Attempting to access an address that is not valid generates an MMU fault.

Address calculations are performed modulo 2^{32}.

The result of an address calculation is UNKNOWN if it overflows or underflows the 32-bit address range A[31:0].

Memory accesses use the `MemA{size}()`, `MemO{size}()`, `MemU{size}()`, and `MemU_unpriv{size}()` pseudocode accessors:

- The `MemA{size}()` accessor makes an aligned access of the required type.
- The `MemO{size}()` accessor makes an ordered access of the required type.
- The `MemU{size}()` accessor makes an unaligned access of the required type
- The `MemU_unpriv{size}()` accessor makes an unaligned, unprivileged access of the required type.

Each of these accessors calls the `Mem_with_type{size}()` accessor, which specifies the required access. This calls `AArch32_MemSingle{size}()`, which performs an atomic, little-endian read.

The `AccessDescriptor` type defines the different access types and attributes.

Note

- The AArch32 System Level Memory Model and The AArch32 Virtual Memory System Architecture include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also reference pseudocode descriptions of these operations.
- For references to the pseudocode that relates to memory accesses, see Basic memory access, Unaligned memory access, and Aligned memory access.

E2.1.2 Memory type overview

The architecture provides the following mutually-exclusive memory types:

Normal This is generally used for the majority of memory operations, both read/write and read-only operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Device The Arm architecture forbids speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See Gathering.
- They preserve the access order and synchronization requirements, both for accesses to a single peripheral and where there is a synchronization requirement on the observability of one or more memory write and read accesses. See Reordering
- They indicate whether a write can be acknowledged other than at the end point. See Early Write Acknowledgement.

For more information on Normal memory and Device memory, see Memory types and attributes.

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A Note in Device memory describes how these memory types map onto the memory types used from the introduction of Armv8.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. The atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- Requirements for single-copy atomicity.
- Properties of single-copy atomic accesses.
- Multi-copy atomicity.
- Requirements for multi-copy atomicity.
- Concurrent modification and execution of instructions.

For more information about the memory types, see Memory type overview.

E2.2.1 Requirements for single-copy atomicity

In AArch32 state, the single-copy atomic PE accesses are:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.
- Memory accesses caused by `LDREXD` and `STREXD` instructions to doubleword-aligned locations.

`LDM`, `LDC`, `LDRD`, `STM`, `STC`, `STRD`, `PUSH`, `POP`, `RFE`, `SRS`, `VLDM`, `VLDR`, `VSTM`, and `VSTR` instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. The architecture does not require subsequences of two or more word accesses from the sequence to be single-copy atomic.

`LDRD` and `STRD` accesses to 64-bit aligned locations are 64-bit single-copy atomic as seen by translation table walks and accesses to translation tables.

Note

This requirement has been added to avoid the need for complex measures to avoid atomicity issues when changing translation table entries, without creating a requirement that all locations in the memory system are 64-bit single-copy atomic. This addition means:

- The system designer must ensure that all writable memory locations that might be used to hold translations, such as bulk SDRAM, can be accessed with 64-bit single-copy atomicity.
- Software must ensure that translation tables are not held in memory locations that cannot meet this atomicity requirement, such as peripherals that are typically accessed using a narrow bus.

This requirement places no burden on read-only memory locations for which reads have no side effects, since it is impossible to detect the size of memory accesses to such locations.

Advanced SIMD element and structure loads and stores are executed as a sequence of accesses of the element or structure size. The architecture requires the element accesses to be single-copy atomic if and only if both:

- The element size is 32 bits, or smaller.
- The elements are naturally-aligned.

Accesses to 64-bit elements or structures that are 32-bit aligned are executed as a sequence of 32-bit accesses, each of which is single-copy atomic. The architecture does not require subsequences of two or more 32-bit accesses from the sequence to be single-copy atomic.

When an access is not single-copy atomic by the rules described in this section, it is executed as a sequence of one or more accesses that aggregate to the size of the original access. Each of the accesses in this sequence is single-copy atomic, at least at the byte level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In this section, the terms *before the write operation* and *after the write operation* mean before or after the write operation has had its effect on the coherence order of the bytes of the memory location accessed by the write operation.

If, according to these rules, an instruction is executed as a sequence of accesses, a synchronous Data Abort exception or Debug state entry can be taken during that sequence. This causes execution of the instruction to be abandoned. See Data Abort exception and, when FEAT_LSMAOC is implemented, Taking an interrupt or other exception during a multiple-register load or store.

If the synchronous Data Abort exception is returned from using the preferred return address, the instruction that generated the sequence of accesses is re-executed and so any access that was performed before the exception was taken is repeated. This also applies to an exit from Debug state.

Note

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:

- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic:
  - At 32-bit granularity in A32 state.
  - At 16-bit granularity in T32 state.

  Concurrent modification and execution of instructions describes additional constraints on the behavior of instruction fetches.
- Translation table walks are performed using accesses that are single-copy atomic:
  - At 32-bit granularity when using Short-descriptor format translation tables.
  - At 64-bit granularity when using Long-descriptor format translation tables.

E2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2, if one of the overlapping reads generated by L_1 Reads-from-memory one of the overlapping writes generated by S_2, then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1.

For more information, see Definition of the memory model.

E2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The memory model is Other-multi-copy-atomic. For more information, see External visibility requirement.

E2.2.5 Concurrent modification and execution of instructions

The architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where the instruction before modification and the instruction after modification are:

- When executing the A32 instruction set, a `B`, `BKPT`, `BL`, `HVC`, `ISB`, `NOP`, `SMC`, or `SVC` instruction.
- When executing the T32 instruction set. a 16-bit `B`, `BKPT`, `BLX`, `BX`, `NOP`, or `SVC` instruction.

In addition, for the 32-bit T32 instructions, for which Instruction encodings describes the meaning of {`hw1`, `hw2`}:

- `hw1` of a 32-bit `BL` (immediate) instruction can be concurrently modified to `hw1` of another `BL` (immediate) instruction:
  - This means that some of the most significant bits of the immediate value can be modified.
- `hw1` of a 32-bit `BLX` (immediate) instruction can be concurrently modified to `hw1` of another `BLX` immediate instruction:
  - This means that some of the most significant bits of the immediate value can be modified.
- `hw1` of a 32-bit `BL` (immediate) or `BLX` (immediate) instruction can be concurrently modified to a T32 16-bit `B`, `BX`, `BLX`, `BKPT`, or `SVC` instruction. This modification also works in reverse.
- `hw2` of a 32-bit `BL` (immediate) instruction can be concurrently modified to `hw2` of another `BL` (immediate) instruction with a different immediate:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.
- `hw2` of a 32-bit `BLX` (immediate) instruction can be concurrently modified to `hw2` of another `BLX` (immediate) instruction with a different immediate:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.
- `hw2` of a 32-bit `B` (immediate) instruction with a condition field can be concurrently modified to `hw2` of another 32-bit `B` (immediate) instruction with a condition field with a different immediate:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.
- `hw2` of a 32-bit `B` (immediate) instruction without a condition field can be concurrently modified to `hw2` of another 32-bit `B` (immediate) instruction without a condition field:
  - This means that some bits of the immediate value, including the least significant bits, can be modified.

Note

- In the T32 instruction set:
  - The only encodings of `BKPT` and `SVC` are 16-bit.
  - The only encoding of `BL` is 32-bit.
- The `ISB` instruction can be concurrently modified and executed in the A32 and A64 instruction sets, but not in the T32 instruction set.

For the instructions explicitly identified in this section, the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The instructions to which this applies are the `B`, `BL`, `NOP`, `BKPT`, `SVC`, `HVC`, and `SMC` instructions.

For both instruction sets, if one thread of execution changes a conditional branch instruction to another conditional branch instruction, and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- The old condition being associated with the new target address.
- The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the *always* condition.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.
2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for self-modifying code
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first
; line instead of STR for a 16-bit instruction.
STR <Rt>, [Rn]
DCCMVAU Rn                      ; Clean data cache by MVA to point of unification (PoU)
DSB                         ; Ensure visibility of the data stored
ICIMVAU Rn                      ; Invalidate instruction cache by VA to PoU
BPIMVA Rn                       ; Invalidate branch predictor by MVA to PoU
DSB
```

Note

- The `DCCMVAU` operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.

3. In a multiprocessor system, the `ICIMVAU` and `BPIMVA` are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, once the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
        ISB                             ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see Synchronization and coherency issues between data and instruction accesses.

For information about the Memory Effects generated by instruction fetches, see External visibility requirement.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.3 Definition of the memory model

This section describes observation and ordering in the memory model. It contains the following subsections:

- Basic and dependency definitions.
- Ordering relations and constraints.
- Ordering of instruction fetches.
- Restrictions on the effects of speculation.
- Memory barriers.

For more information on endpoint ordering of memory accesses, see Reordering.

In the memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see Memory types and attributes.

The architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- The AArch32 System Level Memory Model.
- The AArch32 Virtual Memory System Architecture.

See also Mismatched memory attributes.

E2.3.1 Basic and dependency definitions

The memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory. See Ordering requirements defined by the formal concurrency model for more information.

E2.3.2 Ordering relations and constraints

See Ordering requirements defined by the formal concurrency model for more information.

E2.3.3 Ordering of instruction fetches

For two memory locations A and B, if A has been written to with an updated value and been made coherent with the instruction fetches of the shareability domain before B has been written to with an updated value by an observer in the same shareability domain, then where, for an observer in the shareability domain, an instruction read from B appears in program order before an instruction fetched from A, if the instruction read from B contains the updated value of B then the instruction read from A appearing later in program order will contain the updated value of A.

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR.{DIC, IDC} == {0, 0} The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {1, 1} The write is complete for the shareability domain.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When CTR.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the ICIMVAU instructions of a different core.
  - When CTR.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

E2.3.4 AArch32 restrictions on the effects of speculation

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context Synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context Synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context Synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context Synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a Translation fault, it cannot be used to form an address or generate condition codes to be used by other instructions in the speculative sequence.
- When data is loaded under speculation from a location that does not have a valid translation for the translation regime being speculated in, the data cannot be used to form an address or generate condition codes to be used by other instructions in the speculative sequence.
- When data is loaded as a result of speculative accesses made after `TLBI` + `DSB` + `ERET` using a translation that was invalidated by the TLBI, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the microarchitectural state.
- Changes to Special-purpose registers can occur speculatively.
- Execute-never controls apply to speculative instruction fetching. See Access permissions for instruction execution.
- When writing new instructions to memory, there is no requirement for an `SB` instruction to prevent speculative execution of the old code. See Instruction cache maintenance instructions.

Note

The prohibition of using data loaded under speculation with faults to form addresses, condition codes or SVE predicate values does not prohibit the use of value predicted data from such locations for such purposes, so long as the training of the data value prediction was from the hardware defined context that is using the prediction. A consequence of this is that training of value prediction cannot be based on data loaded under speculation with a translation or Permission fault.

E2.3.4.1 AArch32 Speculative Store Bypass Safe

When FEAT_SSBS is implemented, CPSR.SSBS is a control that can be set by software to indicate whether hardware is permitted to use, in a manner that is potentially speculatively exploitable, a speculative value in a register that has been loaded from memory using a load instruction that speculatively read an entry for the location being loaded from, where

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

When the value of CPSR.SSBS is 0, hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

When the value of CPSR.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Note

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.
- Before the introduction of FEAT_SSBS, the control bit SPSR.SSBS was RES0, and therefore software written without awareness of FEAT_SSBS is expected to program it to 0. This means that CPSR.SSBS will not be set, so the PE will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

E2.3.4.2 Definition of exploitative control of speculative execution

The execution of some code (code1) can exploitatively control speculative execution of some other code (code2) if and only if all of the following apply:

- The actions of code1 can influence, in a manner that is not hard-to-determine, the prediction of multi-bit values that determine speculative execution of code2 to cause an irreversible change to the microarchitectural state of the PE that is indicative of some architectural state accessible to the execution context of code2.
- code1 has control in determining the choice of the architectural state that the irreversible change to the microarchitectural state is indicative of.
- The irreversible changes to the microarchitectural state of the PE can be measured by code executing in an execution context other than that of code2 to allow the retrieval of the architectural state in a computationally feasible manner that is not hard-to-determine.

E2.3.4.3 Definition of predictive leakage to speculative execution

The execution of some code (code1) can predictively leak to some other code (code2) if and only if all of the following apply:

- The execution of code1 influences, in a manner that is not hard-to-determine, the predictive microarchitectural structures of the implementation that predict multi-bit values, not binary choices, to behave in a way that is indicative of some architectural state accessible to the execution context of code1.
- The predictive microarchitectural structures of the implementation impact the timing of the speculative execution of code2 in a way that enables code2 to recover the architectural state in a manner that is not hard-to-determine.
- code1 and code2 are not collaborating to communicate using the mechanisms in the previous two bullets.

Note

Mechanisms to prevent the influence and the state recovery being “not hard-to-determine” are left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.

E2.3.4.4 Further restrictions on the effects of speculation from Armv8.5

Further restrictions on speculation are introduced by some additional architectural features as described here.

FEAT_CSV3 introduces these restrictions:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An attempt to access data under speculation, where the data is inaccessible given the current state of the PE cannot be used to alter the architectural or microarchitectural state in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed through means such as measuring the timing of code. This restriction applies to any of the following cases which make the data inaccessible:
  - reading from a Location with a Permission or Domain fault.
  - directly reading from a system register that is not architecturally readable.
  - reading from a SIMD&FP register that is not accessible due to traps configured by a more privileged Exception level.
  - indirectly reading from a system register containing a Pointer Authentication Key for which the corresponding Pointer Authentication instructions are disabled.

  Note

  The current state of the PE is defined as the EL, values of all special-purpose registers, and values of all system registers under which the hardware attempts to access the data, which may be on the speculative execution path. If on the speculative execution path, the hardware must respect any changes to the PE state that occur on the speculative path, respecting the same rules that would be required if executing on the non-speculative path.

  Note

  The following is a list of examples in which use of the inaccessible data could lead to recovery of the value of the inaccessible data:

  - to form an address or generate condition codes to be used in the speculative sequence.
  - to form the register value used for the comparison of a compare-and-branch, test-and-branch, or compare-and-swap instruction in the speculative sequence. Note that this is to cover conditional instructions that do not use condition codes to determine their outcomes.
  - by prediction mechanisms such as Cache Prefetch or Data Value predictions.
  - as an input to an instruction in the speculative sequence where the execution timing of the instruction is dependent on the data value.

  Note

  It is permissible for a value of zero to be produced by the load or register read and consumed by an instruction newer than the load in the speculative sequence even if the value of zero results in different execution timing compared to non-zero values.

  Note

  As the effects of speculation are not architecturally visible, this restriction level requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the microarchitectural state.

  Note

  Changes to Special-purpose registers can occur speculatively.

FEAT_CSV2 and FEAT_CSV2_1p1 introduce a range of additional restrictions:

If FEAT_CSV2 is implemented:

- Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2), as a result of the behavior of any of the following resources:
  - Branch target prediction based on the branch targets used in context1.
    - This applies to both direct and indirect branches, including return instructions, but excludes the prediction of the direction of a conditional branch.
  - Data Value predictions based on data value from execution in context1.

    Note

    The prediction of the PSTATE.{N, Z, C, V} values is not considered a data value for this purpose.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Virtual address-based cache prefetch predictions generated as a result of execution in context1.
- Any other prediction mechanisms, other than Branch, Data Value, or Cache Prefetch predictions.
  In this definition, the hardware-defined context is determined by:
  - The Exception level.
  - The Security state.
  - When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
  - When executing at EL0 and using the EL1&0 translation regime, the ASID and, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL2&0 translation regime, the ASID.

If FEAT_CSV2_1p1 is implemented:

- Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2) as a result of the behavior of branch target prediction based on the branch history used in context1.
- Branch or data values trained from one instruction address cannot exploitatively control, or predictively leak to, the speculative execution of code from a different address.

E2.3.5 Memory barriers

The Arm architecture is a weakly ordered memory architecture that supports out of order completion. *Memory barrier* is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The following subsections describe the memory barrier instructions:

- Instruction Synchronization Barrier.
- Data Memory Barrier.
- Data Synchronization Barrier.
- Speculation Barrier.
- Consumption of Speculative Data Barrier.
- Speculative Store Bypass Barrier.
- Physical Speculative Store Bypass Barrier.
- Trace Synchronization Barrier.
- Load-Acquire, Store-Release.

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

The `DMB` and `DSB` memory barriers affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

AArch32 state also supports the legacy barrier instructions CP15DMB, CP15DSB, and CP15ISB. These instructions are executed as `MCR`s using the appropriate encoding, and are accessible from EL0. However, for performance reasons Arm deprecates any use of these operations, and strongly recommends that software uses the `DMB`, `DSB`, and `ISB` instructions described in this section instead. Optionally, an implementation can support a CP15BEN control that supervisory

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

software can use to disable use of these instructions, meaning the corresponding MCR encodings are UNDEFINED. When the CP15BEN control is supported, setting one of the following CP15BEN fields to 0 makes execution of CP15DMB, CP15DSB, and CP15ISB UNDEFINED:

- SCTLR_EL1.CP15BEN, for execution of these instructions at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.CP15BEN, for execution of these instructions at EL0 or EL1 when EL1 is using AArch32.
- HSCTLR.CP15BEN, for execution of these instructions at EL2 when EL2 is using AArch32.

E2.3.5.1 Instruction Synchronization Barrier

An ISB instruction ensures that all instructions that come after the ISB instruction in program order are fetched from the cache or memory after the ISB instruction has completed. Using an ISB ensures that the effects of context-changing operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context-changing operations that require the insertion of an ISB instruction to ensure the effects of the operation are visible to instructions fetched after the ISB instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the ISB instruction take effect only after the ISB has been executed.

The pseudocode function for the operation of an ISB is `InstructionSynchronizationBarrier()`.

See also Memory barriers.

E2.3.5.2 Data Memory Barrier

The DMB instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The DMB instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the DMB instruction is covered formally in the Definition of the memory model and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been Observed-by the PE before the DMB instruction is executed, are Observed-by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are Observed-by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The pseudocode function for the operation of a DMB instruction is `DataMemoryBarrier()`.

E2.3.5.3 Data Synchronization Barrier

A DSB instruction is a memory barrier that ensures that memory accesses that occur before the DSB instruction have completed before the completion of the DSB instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a DSB at EL2 ensures that any memory accesses caused by speculative translation table walks from the Non-secure PL1&0 translation regime have been observed.

For more information, see Use of out-of-context translation regimes.

A DSB executed by a PE, PEe, completes when all of the following apply:

- All explicit memory effects of the required access types appearing in program order before the DSB are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, the following instructions issued by PEe before the DSB are complete for the required shareability domain:
  - All cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - All AArch32 TLB maintenance instructions.
  - All PSB instructions.
- If the required access types of the DSB is reads and writes, completion of a DSB instruction executed by PEe ensures that:
  - All previous TLB maintenance operations generated by AArch32 TLB maintenance instructions executed at EL1 by PEe when HCRX_EL2.FnXS is 1 are finished for all PEs in the shareability domain of the DSB instruction.
  - All previous TLB maintenance operations generated by AArch32 TLB maintenance instructions are finished for all PEs in the shareability domain of the DSB instruction.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes, other than:

- Being fetched from memory and decoded, or any change to architectural or microarchitectural state resulting from being fetched or decoded.
- Indirectly or directly reading any of the following, so long as the reading of these resources does not cause a change in architectural state:
  - General-purpose registers
  - SIMD&FP registers
  - The Stack Pointer register
  - The Program Counter
  - Special-purpose registers
- If FEAT_ETS2 is not implemented, having any virtual addresses of loads and stores translated.

The pseudocode function for the operation of a DSB is `DataSynchronizationBarrier()`.

See also Memory barrier instructions and Memory barriers.

E2.3.5.3.1 Maintenance scope of the data synchronization barrier operations

A DSB instruction that applies to both reads and writes takes an argument that specifies its maintenance scope. The maintenance scope indicates the subset of TLB invalidate (TLBI) instructions, TLB invalidate pair (TLBIP) instructions, and Instruction cache maintenance (IC) instructions that are covered by this DSB, for one of:

- Outer Shareable domain.
- Inner Shareable domain.
- Non-shareable domain.

When a DSB instruction that applies to both reads and writes is used to ensure completion of TLB maintenance instructions or Instruction cache maintenance (IC) instructions, all of the following apply:

- A DSB NSH is sufficient to ensure completion of TLB maintenance instructions or Instruction cache maintenance (IC) instructions that apply to a single PE.
- A DSB ISH is sufficient to ensure completion of TLB maintenance instructions or Instruction cache maintenance (IC) instructions that apply to PEs in the same Inner Shareable domain.
- A DSB OSH is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Outer Shareable domain.

E2.3.5.4 Speculation Barrier

An SB instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation. An example is speculative allocation into any caching structure where the allocation of that entry could indicate data value present in memory or in the registers.

The speculative execution of an SB instruction cannot be as a result of any of the following:

- Control flow speculation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Data value speculation.

An SB instruction can complete when all of the following apply:

- It is known that it is not speculative, or it is speculative only as a result of either:
  - Speculating that an instruction that could generate an exception does not generate an exception.
  - Speculating past the point in the Execution stream where a precise asynchronous exception is taken.
- All the predicted data values generated by instructions appearing in program order before the SB instruction are architecturally resolved, and so are not speculative.

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction.

E2.3.5.5 Consumption of Speculative Data Barrier

The CSDB instruction is a memory barrier instruction that controls speculative execution arising from data value prediction.

Any instruction, other than a branch instruction, that appears in program order after the CSDB cannot be speculatively executed using the results of any of the following predictions if those predictions come from instructions that appear in program order before the CSDB and have not been architecturally resolved:

- Data value predictions of any instructions.
- PSTATE.{N, Z, C, V} predictions of any instructions other than conditional branch instructions or conditional instructions that write to the PC.

Note

For purposes of the definition of CSDB, PSTATE.{N, Z, C, V} are not considered data values. This definition permits:

- Control flow speculation before and after the CSDB instruction.
- Speculative execution of conditional data processing instructions after the CSDB instruction, unless they use the results of data value or PSTATE.{N, Z, C, V} predictions of instructions appearing in program order before the CSDB instruction that have not been architecturally resolved.

E2.3.5.6 Speculative Store Bypass Barrier

The SSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB instruction.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.3.5.7 Physical Speculative Store Bypass Barrier

The `PSSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `PSSBB`, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the `PSSBB` instruction.
- When a load to a location appears in program order before the `PSSBB`, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the `PSSBB` instruction.

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

E2.3.5.8 Trace Synchronization Barrier

The `TSB` instruction is a barrier instruction that preserves the relative order of accesses to System registers due to trace operations and other accesses to the same registers.

A trace operation is an operation of the trace unit generating trace for an instruction when FEAT_TRF is implemented and enabled.

A `TSB` is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that `TSB` is executed in the necessary order.

If trace is generated between a Context synchronization event and a `TSB` operation, these trace operations may be reordered with respect to the `TSB` operation, and therefore may not be synchronized.

The following situations are synchronized using a `TSB`:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a Context synchronization event C.
  - C is in program order before a `TSB` operation T.
  - B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a Context synchronization event C1.
  - C1 is in program order before `TSB` operation T.
  - T is executed in program order before a second Context synchronization event C2.
  - B is executed in program order after C2.

A `TSB` is not needed when a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a Context synchronization event C.
- B is executed in program order before C.

The pseudocode function for the operation of a `TSB` is `TraceSynchronizationBarrier()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.3.5.9 Load-Acquire, Store-Release

The architecture provides a set of instructions with Acquire semantics for loads, and Release semantics for stores.

The full definition of the Load-Acquire instruction is covered formally in the Definition of the memory model and this introduction to the Load-Acquire instruction is not intended to contradict that section.

The basic principle of a Load-Acquire instruction is to introduce order between the memory access generated by the Load-Acquire instruction and the memory accesses appearing in program order after the Load-Acquire instruction, such that the memory access generated by the Load-Acquire instruction is Observed-by each PE, to the extent that PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire instruction are Observed-by that PE, to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the Definition of the memory model and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the memory accesses generated by the PEe executing the Store-Release instruction, together with those which originate from a different PE, to the extent that the PEe is required to observe them coherently, Observed-by the PEe before executing the Store-release.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

In addition, the use of a Load-Acquire or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral arrival order.

Load-Acquire and Store-Release, other than `LDAEXD` and `STLEXD`, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

`LDAEXD` and `STLEXD` access two data elements. The address supplied to the instructions must be doubleword-aligned, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit `DMB` memory barrier instruction.

Table E2-1 summarizes the Load-Acquire/Store-release instructions.

Table E2-1 Load-Acquire/Store-Release instructions

Data type	Load-Acquire	Store-Release	Load-Acquire Exclusive	Store-Release Exclusive
32-bit word	`LDA`	`STL`	`LDAEX`	`STLEX`
16-bit halfword	`LDAH`	`STLH`	`LDAEXH`	`STLEXH`
8-bit byte	`LDAB`	`STLB`	`LDAEXB`	`STLEXB`
64-bit doubleword	-	-	`LDAEXD`	`STLEXD`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4 Ordering of translation table walks

If FEAT_ETS2 is implemented, then an Effect E_1 is Ordered-before an Effect E_2 when all of the following apply:

- E_1 is not generated by an Advanced SIMD or floating-point instruction.
- E_2 is not generated for an instruction fetch.
- E_1 is an Explicit Memory Effect.
- E_1 appears in program order before E_3.
- E_3 is a TLBUncacheable Fault Effect.
- E_2 is Translation-intrinsically-before E_3.
- E_2 is an Implicit TTD Memory Read Effect.

The full definition of the ordering implications of FEAT_ETS2 is covered formally in Ordering requirements defined by the formal concurrency model, and the rules in this section are not intended to contradict that section.

For more information about ordering requirements to FEAT_ETS2, see DSB-ordered-before and ETS-ordered-before.

If FEAT_ETS3 is implemented, then an Effect E_1 is Ordered-before an Effect E_2 when all of the following apply:

- E_1 is not generated by an Advanced SIMD or floating-point instruction.
- E_2 is not generated for an instruction fetch.
- E_1 is an Explicit Memory Effect.
- E_1 appears in program order before E_3.
- E_3 is an MMU Fault Effect.
- E_2 is Translation-intrinsically-before E_3.
- E_2 is an Implicit TTD Memory Read Effect.

The full definition of the ordering implications of FEAT_ETS3 is covered formally in Ordering requirements defined by the formal concurrency model, and the rules in this section are not intended to contradict that section.

For more information about ordering requirements to FEAT_ETS3, see DSB-ordered-before and ETS-ordered-before.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.5 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- Introduction to caches.
- Memory hierarchy.
- Implication of caches for the application programmer.
- Prefetching into cache.

E2.5.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access and take account of two principles of locality:

Spatial locality

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

E2.5.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, a memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure E2-1 shows an example of such a system in a system that supports virtual addressing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

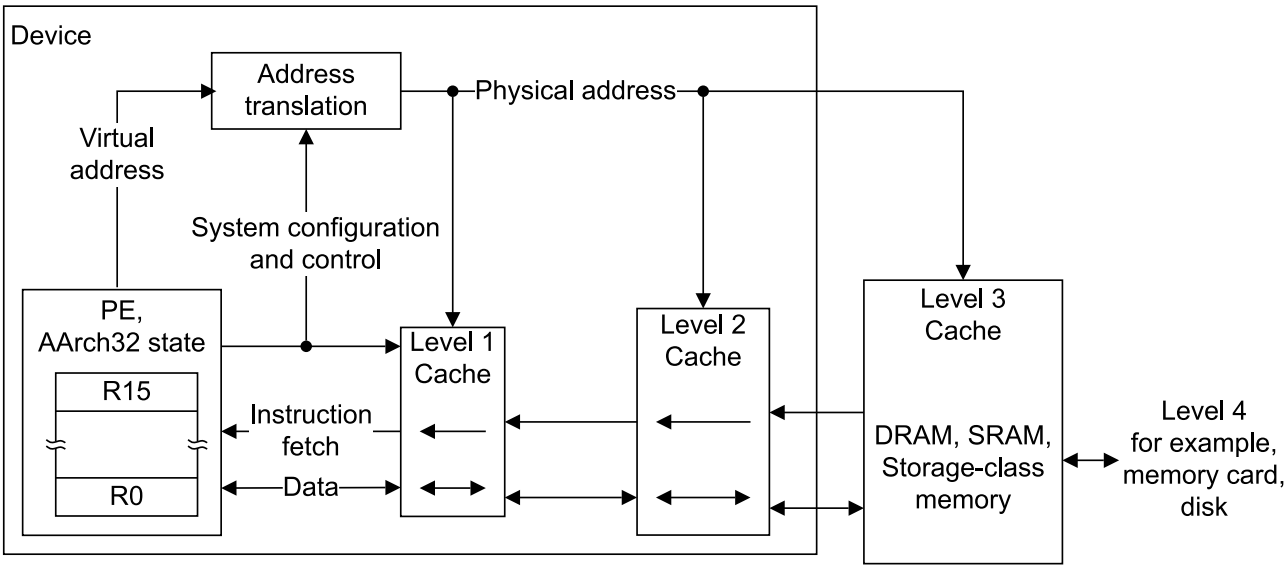

Figure E2-1 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the PE, as shown in Figure E2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU) and Point of Coherency (PoC). For more information, including the definitions of PoU and PoC, see About cache maintenance in AArch32 state.

Note

FEAT_DPB adds architectural support for an additional conceptual point, Point of Persistence, but this support is provided only in AArch64 state. For more information, see About cache maintenance in AArch64 state.

E2.5.2.1 The Cacheability and Shareability memory attributes

Cacheability and Shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This term defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This term defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about the Cacheability and Shareability attributes, see Memory types and attributes.

E2.5.3 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

E2.5.3.1 Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using system calls to functions using cache maintenance instructions that execute at a higher Exception level.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see Non-shareable Normal memory and Shareable, Inner Shareable, and Outer Shareable Normal memory.

Note

The performance of these hardware coherency mechanisms is highly IMPLEMENTATION SPECIFIC. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

E2.5.3.2 Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last Context Synchronization event on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being re-fetched from memory.

The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context Synchronization events and cache maintenance instructions. These can be accessed only from an Exception level that is higher than EL0, and therefore require a system call, see Exception-generating and exception-handling instructions. The following code sequence can be used for this purpose:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
        STR Rt, [Rn]
        DCCMVAU Rn          ; Clean data cache by MVA to point of unification (PoU)
        DSB                 ; Ensure visibility of the data cleaned from cache
        ICIMVAU Rn          ; Invalidate instruction cache by MVA to PoU
        BPIMVA  Rn          ; Invalidate branch predictor by MVA to PoU
        DSB                 ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR.{DIC, IDC} == {0, 0} The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CTR.{DIC, IDC} == {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} == {1, 1} The write is complete for the shareability domain.

Note

- For accesses that are Non-cacheable or Write-Through, the clean data cache instruction is not required. For accesses that are Non-cacheable, the invalidate instruction cache is not required, because in AArch32 state these accesses are not permitted to be held in an instruction cache.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions.

E2.5.4 Prefetching into cache

The Arm architecture provides the memory system hints `PLD` (Preload Data), `PLDW`(Preload Data With Intent To Write) and `PLI` (Preload Instruction) that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring data or instruction locations into caches.

The prefetch instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is reported using an SError exception and taken using an asynchronous Data Abort exception. For more information, see Data Abort exception.

A `PLD`, `PLDW`, or `PLI` instruction can only cause allocation to software-visible caching structures such caches or TLBs for memory locations that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

A `PLD`, `PLDW`, or `PLI` instruction can access any memory location in Normal memory that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

Note

In each case, the entire list applies to each of `PLD`, `PLDW`, and `PLI`.

A `PLD`, `PLDW`, or `PLI` instruction is guaranteed not to access any type of Device memory.

A `PLI` instruction must not perform any access that cannot be performed by a speculative instruction fetch by the processor. Therefore in a VMSA implementation, if all associated MMUs are disabled, a `PLI` instruction cannot access any memory location that cannot be accessed by instruction fetches.

The pseudocode enumeration `PrefetchHint` defines the prefetch hint types.

The `Hint_Prefetch()` pseudocode function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as prefetching the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information on `PLD`, `PLI`, and `PLDW`, see:

- PLD, PLDW (immediate).
- PLD (literal).
- PLD, PLDW (register).
- PLI (immediate, literal).
- PLI (register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.6 Alignment support

This section describes alignment support. It contains the following subsections:

- Instruction alignment.
- Unaligned data access.
- Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE.
- Unaligned data access restrictions.
- Generation of Alignment faults by load/store multiple accesses to Device memory.

For more information about Alignment faults, see Alignment faults.

E2.6.1 Instruction alignment

A32 instructions are word-aligned.

T32 instructions are halfword-aligned.

E2.6.2 Unaligned data access

An A-profile implementation must support unaligned data accesses to Normal memory by some load and store instructions. As Table E2-2 shows, software can control whether a misaligned access to Normal memory by one of these instructions causes an Alignment fault Data Abort exception:

- By setting SCTLR.A, for unaligned accesses from any mode other than Hyp mode.
- By setting HSCTLR.A, for unaligned accesses from Hyp mode.

Table E2-2 Alignment requirements of load/store instructions

Instructions	Alignment check	Result if check fails when: SCTLR.A or HSCTLR.A is 0	SCTLR.A or HSCTLR.A is 1
`LDRB`, `LDREXB`, `LDRBT`, `LDRSB`, `LDRSBT`, `STRB`, `STREXB`, `STRBT`, `TBB`	None	-	-
`LDRH`, `LDRHT`, `LDRSH`, `LDRSHT`, `STRH`, `STRHT`, `TBH`	Halfword	Unaligned access	Alignment fault
`LDREXH`, `STREXH`, `LDAH`, `STLH`, `LDAEXH`, `STLEXH`	Halfword	Alignment fault	Alignment fault
`LDR`, `LDRT`, `STR`, `STRT` `PUSH`, encodings T3 and A2 only `POP`, encodings T3 and A2 only	Word	Unaligned access	Alignment fault
`LDREX`, `STREX`, `LDA`, `STL`, `LDAEX`, `STLEX`	Word	Alignment fault	Alignment fault
`LDREXD`, `STREXD`, `LDAEXD`, `STLEXD`	Doubleword	Alignment fault	Alignment fault
All forms of `LDM` and `STM`, `LDRD`, `RFE`, `SRS`, `STRD`	Word	Alignment fault	Alignment fault
`LDC`, `STC`	Word	Alignment fault	Alignment fault
`VLDM`, `VPOP`, `VPUSH`, `VSTM`	Word	Alignment fault	Alignment fault
`VLDR`, `VSTR` - single-precision scalar and double-precision scalar	Word	Alignment fault	Alignment fault
`VLDR`, `VSTR` - half-precision scalar	Halfword	Alignment fault	Alignment fault
`VLD1`, `VLD2`, `VLD3`, `VLD4`, `VST1`, `VST2`, `VST3`, `VST4`, all with standard alignment	Element size	Unaligned access	Alignment fault

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instructions	Alignment check	Result if check fails when: SCTLR.A or HSCTLR.A is 0	SCTLR.A or HSCTLR.A is 1
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with :<align> specified[a]	As specified by :<align>	Alignment fault	Alignment fault

a. Previous versions of this manual used @<align> to specify alignment. Both forms are supported, see T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions for more information.

Note

Any unaligned access to any type of Device memory generates an Alignment fault, see Alignment faults.

E2.6.3 Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE

Any load instruction that is not faulted by the alignment restrictions shown in Table E2-2 and that loads the PC has CONSTRAINED UNPREDICTABLE behavior if the address it loads from is not word-aligned, see Loads and Stores to unaligned locations. This overrules any permitted load/store behavior shown in Table E2-2.

E2.6.4 Unaligned data access restrictions

The following points apply to unaligned data accesses:

- Accesses are not guaranteed to be single-copy atomic except at the byte access level, see Atomicity in the Arm architecture.
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally-aligned access.
- An operation that performs an unaligned access can abort on any memory access that it makes, and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the boundary.

E2.6.5 Generation of Alignment faults by load/store multiple accesses to Device memory

When FEAT_LSMAOC is implemented and the value of the applicable nTLSMD field is 0, any memory access by an AArch32 Load Multiple or Store Multiple instruction to an address that the stage 1 translation assigns as Device-nGRE, Device-nGnRE, or Device-nGnRnE generates an Alignment fault.

The applicable nTLSMD field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the LDM or STM instruction is executed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.7 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- Instruction endianness.
- Data endianness.
- Endianness of memory-mapped peripherals.

E2.7.1 General description of endianness in the Arm architecture

This section only describes memory addressing and the effects of endianness for data elements up to doubleword of 64 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure E2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The doubleword at address A.
- The words at addresses A and A+4.
- The halfwords at addresses A, A+2, A+4, and A+6.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, and A+7.

The terms in Figure E2-2 have the following definitions:

MSByte Most significant byte.

LSByte Least significant byte.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

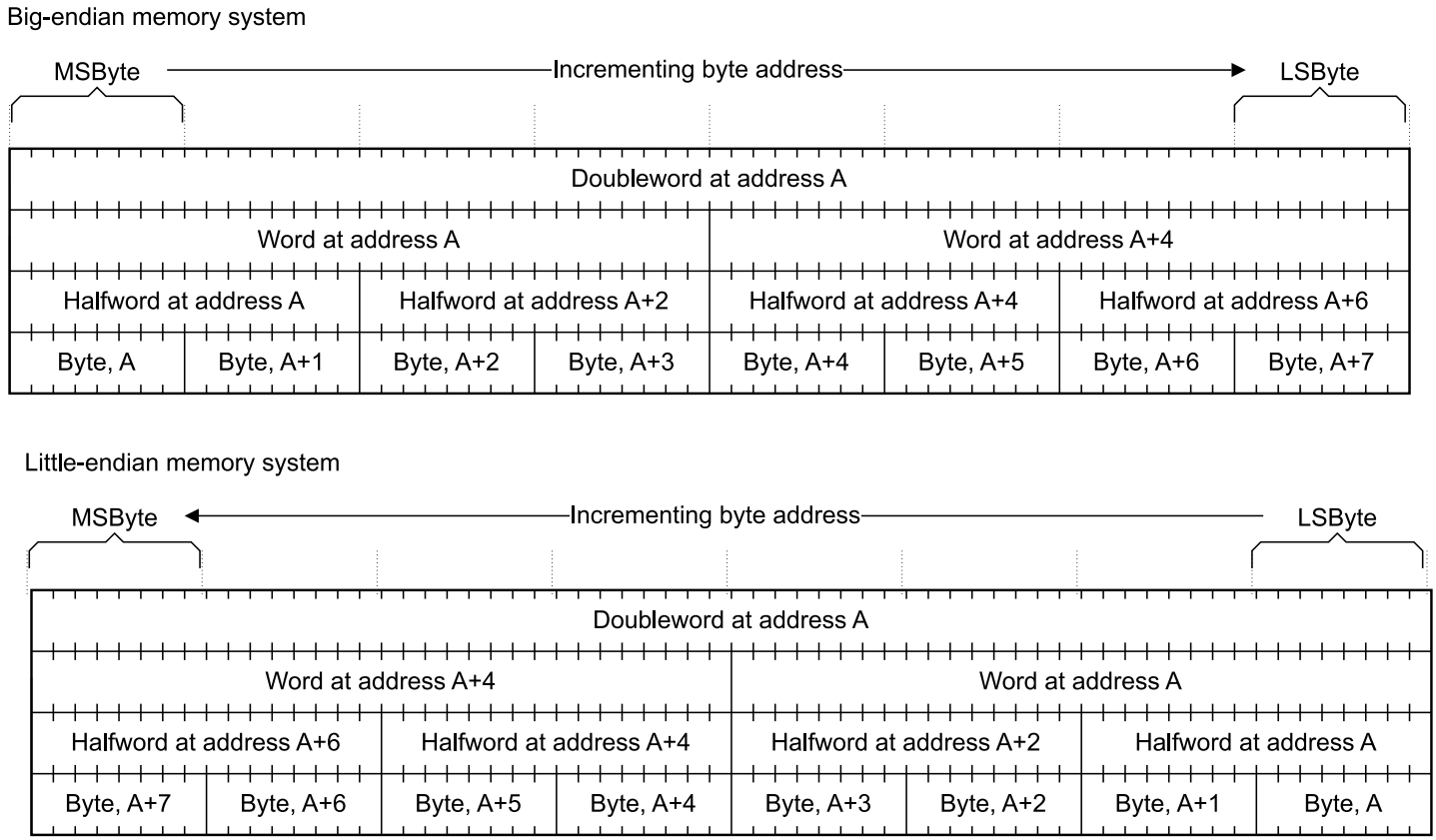

In this figure, *Byte, A+1* is an abbreviation for *Byte at address A+1*

Figure E2-2 Endianness relationships in AArch32 state

E2.7.2 Instruction endianness

The mapping of instruction memory is always little-endian.

E2.7.3 Data endianness

The size of the data value that is loaded or stored is the size that is used for the purpose of endian conversion for floating-point, Advanced SIMD, and general-purpose register loads and stores.

Table E2-3 shows the element sizes of all the load/store instructions, for all instruction sets.

Table E2-3 Element size of load/store instructions

Instructions	Element size
`LDRB`, `LDREXB`, `LDRBT`, `LDRSB`, `LDRSBT`, `STRB`, `STREXB`, `STRBT`, `TBB`	Byte
`LDRH`, `LDREXH`, `LDRHT`, `LDRSH`, `LDRSHT`, `STRH`, `STREXH`, `STRHT`, `TBH`	Halfword
`LDR`, `LDRT`, `LDREX`, `STR`, `STRT`, `STREX`	Word
`LDRD`, `LDREXD`, `STRD`, `STREXD`	Word
All forms of `LDM`, `PUSH`, `POP`, `RFE`, `SRS`, all forms of `STM`,	Word
`LDC`, `STC`	Word

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instructions	Element size
Forms of `VLDM`, `VLDR`, `VPOP`, `VPUSH`, `VSTM`, `VSTR` that transfer 32-bit S registers	Word
Forms of `VLDM`, `VLDR`, `VPOP`, `VPUSH`, `VSTM`, `VSTR` that transfer 64-bit D registers	Doubleword
`VLD1`, `VLD2`, `VLD3`, `VLD4`, `VST1`, `VST2`, `VST3`, `VST4`	Element size of the Advanced SIMD access

CPSR.E determines the data endianness.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for Advanced SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for Advanced SIMD element and data structure loads and stores. For more information, see Endianness in Advanced SIMD.

E2.7.3.1 Instructions to reverse bytes in registers

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table E2-4 shows the instructions that provide this functionality in the T32 and A32 instruction sets.

Table E2-4 Byte reversal instructions

Function	T32/A32 instruction	Notes
Reverse bytes in whole register	`REV`	For use with general purpose registers
Reverse bytes in 16-bit halfwords	`REV16`	For use with general purpose registers
Reverse bytes in halfword and sign-extend	`REVSH`	For use with general purpose registers
Reverse elements in doublewords, vector	`VREV64`	For use with registers in the SIMD and floating-point register file
Reverse elements in words, vector	`VREV32`	For use with registers in the SIMD and floating-point register file
Reverse elements in halfwords, vector	`VREV16`	For use with registers in the SIMD and floating-point register file

E2.7.3.2 Endianness in Advanced SIMD

Advanced SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used by the PE to load and store data correctly in both big-endian and little-endian systems.

Consider, for example, the A32 or T32 instruction:

```
VLD1.16 {D0}, [R1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure E2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

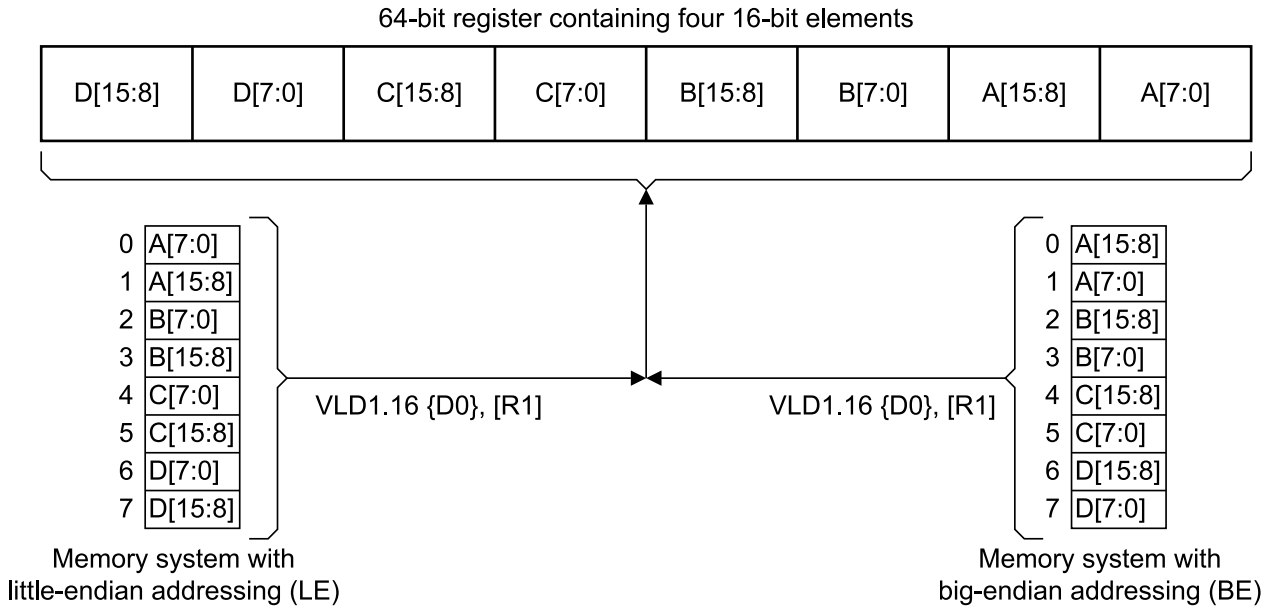

Figure E2-3 Advanced SIMD byte order example for AArch32 state

For information about the alignment of Advanced SIMD instructions, see Alignment support.

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in A-profile Architecture Pseudocode.

E2.7.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a cross-trigger interface, see About the External Debug Registers.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see System Level Implementation of the Generic Timer.
- A memory-mapped register interface to the Performance Monitors, see Recommended External Interface to the Performance Monitors.
- A memory-mapped register interface to the Activity Monitors, see Recommended External Interface to the Activity Monitors.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.*

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.8 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- Normal memory.
- Device memory.
- Memory access restrictions.

E2.8.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform Speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- An Explicit Memory Write Effect to a memory location with the Normal attribute completes in finite time.
- Explicit Memory Write Effects to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer Cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes. For the purposes of this requirement, the endpoint for a location in Conventional memory is the PoC.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Normal memory.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The following sections describe the other attributes for Normal memory:

- Shareable Normal memory.
- Non-shareable Normal memory.
- Cacheability attributes for Normal memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also:

- Multi-register loads and stores that access Normal memory.
- Atomicity in the Arm architecture.
- Memory barriers. For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.
- Concurrent modification and execution of instructions.

E2.8.1.1 Shareable Normal memory

A Normal memory location has a Shareability attribute that is defined as one of:

- Inner Shareable.
- Outer Shareable.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see Synchronization and coherency issues between data and instruction accesses.

Note

- System designers can use the Shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the Shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different PEs.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

E2.8.1.1.1 Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

The details of the use of the Shareability attributes are system-specific. Example E2-1 shows how they might be used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example E2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different Shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same Shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable Shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable Shareability domain.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

E2.8.1.2 Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

E2.8.1.3 Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see Cacheability, cache allocation hints, and cache transient hints.

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint this does not affect the behavior of accesses to that location.
- For other cases, see Mismatched memory attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the Shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. For this reason, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations and constraints.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner Cacheability attributes can lie outside a cache that is controlled by the Outer Cacheability attributes.
- An implementation might not have any outer cache.

Example E2-2, Example E2-3, and Example E2-4 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see Memory hierarchy.
- When managing coherency, system designs must consider both the Inner and Outer Cacheability attributes, as well as the Shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example E2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner Cacheability attribute applied to L1 and L2 cache.
- The Outer Cacheability attribute applied to L3 cache.

Example E2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner Cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer Cacheability attribute.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example E2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner Cacheability attribute applied to L1 cache.
- The Outer Cacheability attribute applied to L2 and L3 cache.

E2.8.1.4 Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more registers from the SIMD and floating-point register file from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

E2.8.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Device memory are:

Gathering Identified as G or nG, see Gathering.

Reordering Identified as R or nR, see Reordering.

Early Write Acknowledgement

Identified as E or nE, see Early Write Acknowledgement.

The Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early Write Acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 added this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

The Device-nGRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Armv8 added this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden.

The Device-GRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.
- Earlier versions of the Arm architecture defined the following memory types:
  - Strongly-ordered memory. This is the equivalent of the Device-nGnRnE memory type.
  - Device memory. This is the equivalent of the Device-nGnRE memory type.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  An exception to this applies:
  - Reads generated by the Advanced SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  Note

  - An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.
  - Write speculation that is visible to other observers is prohibited for all memory types.
- An Explicit Memory Write Effect to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit memory write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Explicit Memory Write Effects to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two Explicit Memory Write Effects of Device memory type to the same location might be merged before they reach the endpoint, unless both Explicit Memory Write Effects have the non-Gathering attribute or there is an ordered-before relationship between the two Explicit Memory Write Effects.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note

In the Non-secure PL1&0 translation regime in systems where HCR.TGE == 1 and HCR.DC == 0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes the value of HSR.ISS.[24] to be 0.

See also Memory access restrictions.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR. For the Non-secure EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

E2.8.2.1 Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All access occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Device memory.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a DMB or DSB barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

E2.8.2.2 Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.

An implementation is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.

An additional relaxation is that an implementation is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

E2.8.2.3 Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nE Indicates that the location has the No Early Write Acknowledgement attribute.

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgment of a write comes from the endpoint, it is guaranteed that:
  - Only the endpoint of the write access returns a write acknowledgment of the access.
  - No earlier point in the memory system returns a write acknowledgment.
- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgment of a write comes from the endpoint, the acknowledgment of a write is not required to come from the endpoint.

Note

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgment of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute affects only where the endpoint acknowledgment is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
- The areas of the physical memory map for which write acknowledgment from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgment from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgment from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgment from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgment from the endpoint.

E2.8.2.4 Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load and store instructions.

For all instructions that load or store one or more registers from the SIMD and floating-point register file there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load and store instructions.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address, even if the resulting accesses are to any type of Device memory.

For an `LDRD`, `STRD`, or `LDM` instruction with a register list that includes the PC, or an `STM` instruction with a register list that includes the PC, the order in which the registers are accessed is not defined by the architecture.

For a load or store of an Advanced SIMD element or structure, the order in which the registers are accessed is not defined by the architecture.

For a `VLDM` and `VSTM` instruction with a register list that does not include the PC, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.

For a `LDM` or `STM` instruction with a register list that does not include the PC:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_LSMAOC is not implemented, and when FEAT_LSMAOC is implemented and the value of the applicable LSMAOE field is 1, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.
- When FEAT_LSMAOC is implemented and the value of the applicable LSMAOE field is 0, no memory accesses are required to be ordered.
- When FEAT_LSMAOC is implemented and the value of the applicable nTLSMD field is 0, any memory access to an address that the stage 1 translation assigns as Device-nGRE, Device-nGnRE, or Device-nGnRnE generates an Alignment fault.

The applicable LSMAOE or nTLSMD field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the `LDM` or `STM` instruction is executed.

Armv8.2 deprecates software relying on accesses to Device memory made by a single `LDM` or `STM` instruction not being reordered.

E2.8.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit memory read effects to any two adjacent bytes in memory, *p* and *p+1*, generated by the same instruction, and for two explicit memory write effects to any two adjacent bytes in memory, *p* and *p+1*, that are generated by the same instruction:
  - The bytes *p* and *p+1* must have the same memory type and Shareability attributes. otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LDC`, `LDM`, `LDRD STC`, `STM or STRD` instruction, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes *p* and *p+1*.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see Crossing a page boundary with different memory types or Shareability attributes.
- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross a 4KB address boundary then behavior is CONSTRAINED UNPREDICTABLE and Crossing a 4KB boundary with a Device access describes the permitted behaviors.

  Note

  — The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

  — This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses a 4KB address boundary.

  — Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.9 Mismatched memory attributes

Mismatched memory attributes are controlled by privileged software. For more information, see The AArch32 Virtual Memory System Architecture.

Physical memory Locations are accessed with *mismatched attributes* if all accesses to the Location do not use a common definition of all of the following attributes of that Location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.
     - All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.
2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Normal Inner Non-cacheable, Outer Non-cacheable memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:

- Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
- Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory Location all assign the same Shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a Shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a `DMB` barrier instruction between any accesses to the same cacheable Location that use different attributes.

   Note

   In AArch32 state, cache maintenance instructions can be accessed only from an Exception level that is higher than EL0, and therefore require a system call. For information on system calls, see Exception-generating and exception-handling instructions. For information about the AArch32 cache maintenance instructions, see AArch32 cache and branch predictor support.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see Ordering of cache and branch predictor maintenance instructions.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:
   - Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
   - A `DMB` barrier is placed between any accesses to the same memory Location that use different attributes.

   Note

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Note

As described in Non-cacheable accesses and instruction caches, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.10 Synchronization and semaphores

The architecture provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal and Device memory.

Note

Use of the synchronization primitives scales for multiprocessing system designs.

Table E2-5 shows the synchronization primitives and the associated `CLREX` instruction.

Table E2-5 Synchronization primitives and associated instruction, T32 and A32 instruction sets

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	`LDREXB`	`STREXB`	-
	Load-Acquire/Store-Release	`LDAEXB`	`STLEXB`	-
Halfword	-	`LDREXH`	`STREXH`	-
	Load-Acquire/Store-Release	`LDAEXH`	`STLEXH`	-
Word	-	`LDREX`	`STREX`	-
	Load-Acquire/Store-Release	`LDAEX`	`STLEX`	-
Doubleword	-	`LDREXD`	`STREXD`	-
	Load-Acquire/Store-Release	`LDAEXD`	`STLEXD`	-
None	Clear-Exclusive	-	-	`CLREX`

a. Instruction in the T32 and A32 instruction sets.

Except for the row showing the `CLREX` instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see Marking and the size of the marked memory block. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- Exclusive access instructions and Non-shareable memory locations.
- Exclusive access instructions and shareable memory locations.
- Marking and the size of the marked memory block.
- Context switch support.
- Load-Exclusive and Store-Exclusive instruction usage restrictions.
- Use of WFE and SEV instructions by spin-locks.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.10.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the Shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is *set*.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is *clear*.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure E2-4 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

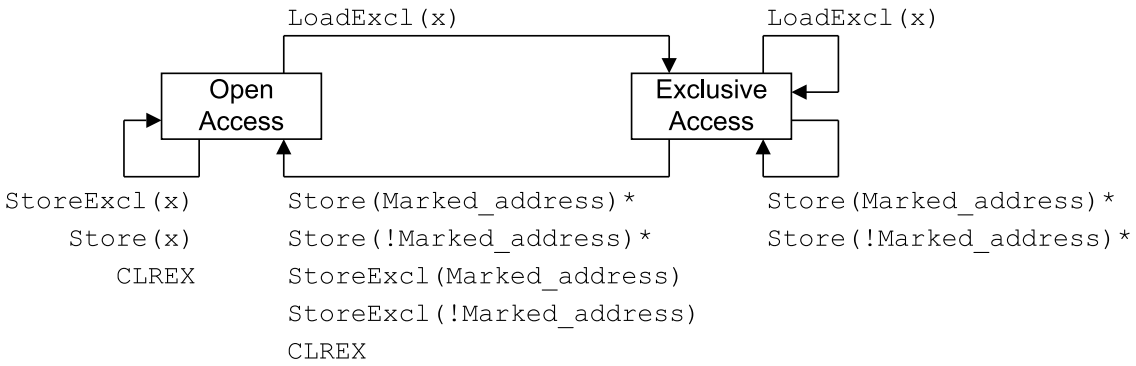

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address `x` used for the operation.

Figure E2-4 Local monitor state machine diagram

For more information about marking, see Marking and the size of the marked memory block.

Note

For the local monitor state machine, as shown in Figure E2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction, by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

E2.10.1.1 Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure E2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure E2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure E2-4 must not indefinitely delay forward progress of execution.

E2.10.2 Exclusive access instructions and shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in Exclusive access instructions and Non-shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

memory locations, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:

— Updating memory.
— Returning a status value of 0.

The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the Shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:

  — Can hold at least one marked block.
  — Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is CONSTRAINED UNPREDICTABLE, see Load-Exclusive and Store-Exclusive instruction usage restrictions.

Note

The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Fault status code of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— DFSR.STATUS = `0b110101` when using the Long-descriptor translation table format. The fault can also be reported in the HSR.ISS[5:0] field for exceptions to Hyp mode.

— DFSR.FS = `0b10101` when using the Short-descriptor translation table format.

If the IMPLEMENTATION DEFINED MMU fault is generated for the Non-secure PL1&0 translation regime then:

— If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.

— Otherwise, the fault is a second stage fault and the exception is taken to EL2.

The priority of this fault is IMPLEMENTATION DEFINED.

- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by an Arm PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

E2.10.2.1 Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  — A status value of 0 is returned to a register to acknowledge the successful store.
  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  — If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  — A status value of 1 is returned to a register to indicate that the store failed.
  — The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  — If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  — If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means it responds to:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses generated by PE(n).
- Accesses generated by the other observers in the Shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

E2.10.2.1.1 Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see Wait For Event and Send Event.

Figure E2-5 shows the state machine for PE(n) in a global monitor.

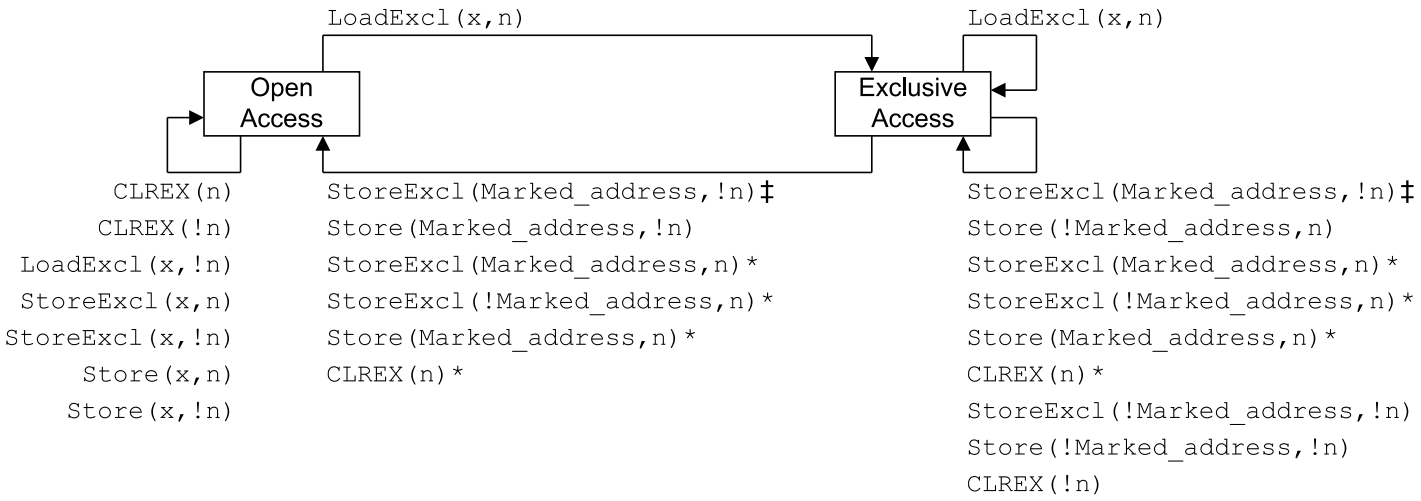

‡`StoreExcl`(Marked_address,!n) clears the monitor only if the `StoreExcl` updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure E2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see Marking and the size of the marked memory block.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For the global monitor state machine, as shown in Figure E2-5:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure E2-5 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can update only the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

E2.10.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4 - 512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where *a* is 4.
- 512 words in an implementation where *a* is 11.

For example, in an implementation where *a* is 4, a successful `LDREXB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

E2.10.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

E2.10.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDREX/STREX` pair or a `LDREXB/STREXB` pair, executed as part of a loop that contains only a single Load-Exclusive and

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Store-Exclusive pair. To support different implementations of these functions, software must follow the notes and restrictions given in this subsection.

The following notes describe use of a Load-Exclusive/ Store-Exclusive instruction pair, `LoadExcl`/`StoreExcl`, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table E2-5. In this context, a `LoadExcl`/`StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses.

  - The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes.

  - The block of data of the size of the larger of the transaction sizes used by the `LoadExcl`/`StoreExcl` pair at the address accessed by the `LoadExcl`/`StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.
- If FEAT_LSE is not implemented, `LoadExcl`/`StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, the software meets all of the following conditions. If FEAT_LSE is implemented, software can maximize the likelihood that `LoadExcl`/`StoreExcl` loops make forward progress, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, by meeting all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, software prefetches, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, `ISB` barriers, indirect branches, or Branch with Link instructions.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:
  - There are no stores or `PLDW` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive. This includes to VA aliases to those addresses.
  - There are no loads or prefetches to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.
  - There are no other Store-Exclusive instructions to any other address.
  - There are no direct or indirect System register writes, other than changes to the flag fields in the CPSR or FPSCR, caused by data processing or comparison instructions.
  - There are no direct or indirect address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, indirect branches, or Branch with Link instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

— Performing stores to a PA covered by the Exclusives monitor.

— Prefetching with intent to write to a PA covered by the Exclusives monitor.

— Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.

— Executing instruction cache invalidate all instructions.

— Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.
- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.
- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.
- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:

  — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

  Note

  This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.

  — The data at the address accessed by the `StoreExcl` is UNKNOWN.

  Note

  Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `StoreExcl` instruction, and the change is performed using a break-before-make sequence as described in Using break-before-make when updating translation table entries, if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

  Note

  Arm expects that, in many implementations, either:

  — The TLB invalidation will clear either the local or global monitor.

  — The PA will be checked between the `LoadExcl` and `StoreExcl`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PLDW` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PLDW` accesses to a memory location might impede the forward progress of another PE.

Note

In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

E2.10.6 Use of WFE and SEV instructions by spin-locks

The architecture provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in Wait For Event and Send Event. However, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an `SEVL` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part F

The AArch32 Instruction Sets

Chapter F1
About the T32 and A32 Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- Format of instruction descriptions.
- Standard assembler syntax fields.
- Conditional execution.
- Shifts applied to a register.
- Memory accesses.
- Encoding of lists of general-purpose registers and the PC.
- General information about the T32 and A32 instruction descriptions.
- Additional pseudocode support for instruction descriptions.
- Additional information about Advanced SIMD and floating-point instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.1 Format of instruction descriptions

The instruction descriptions in T32 and A32 Base Instruction Set Instruction Descriptions and T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions normally use the following format:

- Instruction section title.
- Introduction to the instruction.
- A description of each encoding of the instruction.
- Assembler syntax.
- Pseudocode describing how the instruction operates.
- Notes, if applicable.

Each of these items is described in more detail in the following subsections.

F1.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instruction or instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

F1.1.2 Introduction to the instruction

The introduction to the instruction briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

F1.1.3 Instruction encodings

This is a list of one or more instruction encodings. Each instruction encoding is labeled as:

- A1, A2, A3 . . . for the first, second, third, and any additional A32 encodings.
- T1, T2, T3 . . . for the first, second, third, and any additional T32 encodings.

Each instruction encoding description consists of:

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. Sometimes, multiple syntax variants are given. These are written in a `typewriter` font using the conventions described in Assembler syntax prototype line conventions. The correct one to use can be indicated by:
  - — A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading *Flag setting, rotate right with extend variant* in the description of the A1 encoding of the `ADC`, `ADCS` (register) instructions in A1.
  - — An annotation to the syntax, such as *Inside IT block* or *Outside IT block*. See, for example, the syntax descriptions of the T1 encoding of the `ADC`, `ADCS` (register) instructions in T1.

  In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants, and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax `AND R0, R0, R8` ensures selection of a 32-bit encoding but `AND R0, R0, R1` selects a 16-bit encoding.

  For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

  - — If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture *prefers* the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding. Software can use the `.W` and `.N` qualifiers to specify the required encoding width, see Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If multiple encodings of the same length can encode the instruction, the *Assembler syntax* subsection says which encoding is preferred, and how software can, instead, select the other encodings.
Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

Note

In some cases, an instruction is available in one instruction set but not in another. The *Assembler syntax* subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == `0b1111` described in Branch, branch with link, and block data transfer, System register access, Advanced SIMD, floating-point, and Supervisor call, and Unconditional instructions cannot have a Condition code, but the equivalent T32 instructions often can, and this usually appears in the *Assembler syntax* subsection as a statement that the A32 instruction cannot be conditional.
However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might use simpler syntaxes when they are suitable for the operand combination, to produce more readable disassembled code.

- An encoding diagram, where:
  — For a 32-bit A32 encoding diagram, the bits are numbered from 31-0.
  — For a 16-bit T32 encoding diagram, the bits are numbered from 15-0.
  This halfword can be described as `hw1` of the instruction.
  — For a 32-bit T32 encoding diagram, the bits are numbered from 15-0 for each halfword, as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word.
  In this case, the left-hand halfword in the diagram is identified as `hw1`, and the right-hand halfword is identified as `hw2`.

  Where instructions are stored using the standard little-endian instruction endianness:
  — The encoding diagram for an A32 instruction at address A shows, from left to right, the bytes at addresses A+3, A+2, A+1, A.
  — The encoding diagram for a 32-bit T32 instruction shows bytes in the order A+1, A for `hw1`, followed by bytes A+3, A+2 for `hw2`.
- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Arm Pseudocode Definition.

F1.1.4 Assembler symbols

The *Assembly symbols* describe the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- Descriptions of all variable or optional fields of the syntax.
  Some syntax fields are standardized across all or most instructions. Standard assembler syntax fields describes these fields.
  By default, syntax fields that specify registers, such as `<Rd>`, `<Rn>`, or `<Rt>`, can be any of R0-R12 or LR in T32 instructions, and any of R0-R12, SP, or LR in A32 instructions. These require that the encoding-specific pseudocode set the corresponding integer variable (such as `d`, `n`, or `t`) to the corresponding register number, using 0-12 for R0-R12, 13 for SP, or 14 for LR:
  — Normally, software can do this by setting the corresponding field in the instruction, typically named Rd, Rn, Rt, to the binary encoding of that number.
  — In the case of 16-bit T32 encodings, the field is normally of length 3, and so the encoding is available only when the assembler syntax specifies one of R0-R7. Such encodings often use a register field name like Rdn. This indicates that the encoding is available only if `<Rd>` and `<Rn>` specify the same register, and that the register number of that register is encoded in the field if they do.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or documents other differences from the default rules for such fields. Examples of extensions are permitting the use of the SP in a T32 instruction, or permitting the use of the PC, identified using register number 15.

- Where appropriate, text that briefly describes changes from the pre-UAL assembler syntax. Where present, this usually consists of an alternative pre-UAL form of the assembler mnemonic. The pre-UAL assembler syntax does not conflict with UAL. Arm recommends that it is supported, as an optional extension to UAL, so that pre-UAL assembler source files can be assembled.

F1.1.4.1 Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

`< >`

Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence requires only the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an instruction contains an item `<Rn>` and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as `add = TRUE`. The assembler must use only encodings that produce that output.

`{ }`

Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

`#`

In the assembler syntax, numeric constants are normally preceded by a #. Some UAL instruction syntax descriptions explicitly show this # as optional. Any UAL assembler:

- Must treat the # as optional where an instruction syntax description shows it as optional.
- Can treat the # either as mandatory or as optional where an instruction syntax description does not show it as optional.

Note

Arm recommends that UAL assemblers treat all uses of # shown in this manual as optional.

spaces

Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

`+/-`

This indicates an optional + or - sign. If neither is coded, + is assumed.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. In a few places, the { and } characters must be encoded as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

F1.1.5 Pseudocode describing how the instruction operates

The *Operation for all classes* subsection contains encoding-independent pseudocode that describes the main operation of the instruction.

F1.1.5.1 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses, see External visibility requirement.
- Pseudocode does not describe the exact rules when an instruction that generates any of the following fails its condition code check:
  - UNDEFINED instruction.
  - Hyp trap.
  - Monitor trap.
  - Trap to AArch64 exception.

  In such cases, the `Undefined()` pseudocode statement or call to the applicable trap function lies inside the `if ConditionPassed() then ...` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. For the exact rules, see:
  - Conditional execution of undefined instructions.
  - EL2 configurable controls.
  - EL3 configurable controls.
  - Configurable instruction controls.
- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. Combinations of floating-point exceptions describes the exact rules.

  Note

  There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the exceptions in Handling exceptions that are taken to an Exception level using AArch32.

F1.1.5.2 Instruction encoding diagrams and instruction pseudocode

This is a list of one or more instruction encodings. Each instruction encoding is labeled as:

- A1, A2, A3 . . . for the first, second, third, and any additional A32 encodings.
- T1, T2, T3 . . . for the first, second, third, and any additional T32 encodings.

Each instruction encoding description consists of:

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. Sometimes, multiple syntax variants are given. These are written in a `typewriter` font using the conventions described in Assembler syntax prototype line conventions. The correct one to use can be indicated by:
  - A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading *Flag setting, rotate right with extend variant* in the description of the A1 encoding of the `ADC`, `ADCS` (register) instructions in A1.
  - An annotation to the syntax, such as *Inside IT block* or *Outside IT block*. See, for example, the syntax descriptions of the T1 encoding of the `ADC`, `ADCS` (register) instructions in T1.

  In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants, and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax `AND R0, R0, R8` ensures selection of a 32-bit encoding but `AND R0, R0, R1` selects a 16-bit encoding.

  For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:
  - If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture *prefers* the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding. Software can use the `.W` and `.N` qualifiers to specify the required encoding width, see Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If multiple encodings of the same length can encode the instruction, the *Assembler syntax* subsection says which encoding is preferred, and how software can, instead, select the other encodings.
Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

Note

In some cases, an instruction is available in one instruction set but not in another. The *Assembler syntax* subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == `0b1111` described in Branch, branch with link, and block data transfer, System register access, Advanced SIMD, floating-point, and Supervisor call, and Unconditional instructions cannot have a Condition code, but the equivalent T32 instructions often can, and this usually appears in the *Assembler syntax* subsection as a statement that the A32 instruction cannot be conditional.
However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might use simpler syntaxes when they are suitable for the operand combination, to produce more readable disassembled code.

- An encoding diagram, where:
  — For a 32-bit A32 encoding diagram, the bits are numbered from 31-0.
  — For a 16-bit T32 encoding diagram, the bits are numbered from 15-0.
  This halfword can be described as `hw1` of the instruction.
  — For a 32-bit T32 encoding diagram, the bits are numbered from 15-0 for each halfword, as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word.
  In this case, the left-hand halfword in the diagram is identified as `hw1`, and the right-hand halfword is identified as `hw2`.

  Where instructions are stored using the standard little-endian instruction endianness:
  — The encoding diagram for an A32 instruction at address A shows, from left to right, the bytes at addresses A+3, A+2, A+1, A.
  — The encoding diagram for a 32-bit T32 instruction shows bytes in the order A+1, A for `hw1`, followed by bytes A+3, A+2 for `hw2`.
- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Instruction encoding diagrams and instruction pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

`<c>`
Is an optional field. It specifies the condition under which the instruction is executed. See Conditional execution for the range of available conditions and their encoding. If `<c>` is omitted, it defaults to *always* (`AL`).

`<q>`
Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

`.N`
Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

`.W`
Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither `.W` nor `.N` is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description. The assembler syntax includes a mandatory `.W` qualifier, along with a note describing the cases in which it applies, where this qualifier is required to select a particular encoding for an instruction. Additional assembler syntax will describe the syntax when the conditions are not met.

Note

When assembling to the A32 instruction set, the `.N` qualifier produces an assembler error and the `.W` qualifier has no effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.3 Conditional execution

Most T32 and A32 instructions can be executed conditionally, based on the values of the APSR Condition flags. Table F1-1 lists the available conditions.

Table F1-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	`EQ`	Equal	Equal	Z == 1
0001	`NE`	Not equal	Not equal, or unordered	Z == 0
0010	`CS`[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	`CC`[c]	Carry clear	Less than	C == 0
0100	`MI`	Minus, negative	Less than	N == 1
0101	`PL`	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	`VS`	Overflow	Unordered	V == 1
0111	`VC`	No overflow	Not unordered	V == 0
1000	`HI`	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	`LS`	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	`GE`	Signed greater than or equal	Greater than or equal	N == V
1011	`LT`	Signed less than	Less than, or unordered	N != V
1100	`GT`	Signed greater than	Greater than	Z == 0 and N == V
1101	`LE`	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (`AL`)[d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.

b. `HS` (unsigned higher or same) is a synonym for `CS`.

c. `LO` (unsigned lower) is a synonym for `CC`.

d. `AL` is an optional mnemonic extension for always, except in `IT` instructions. For details, see IT.

In T32 instructions, the condition, if it is not `AL`, is normally encoded in a preceding `IT` instruction. For more information, see Conditional instructions and IT. Some conditional branch instructions do not require a preceding `IT` instruction, because they include a Condition code in their encoding.

Implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD, to disable use of IT for some instructions, making them UNDEFINED.

In A32 instructions, bits[31:28] of the instruction contain either:

- The Condition code, see The Condition code field in A32 instruction encodings.
- `0b1111` for some A32 instructions that can only be executed unconditionally.

F1.3.1 The Condition code field in A32 instruction encodings

Every conditional A32 instruction contains a 4-bit Condition code field, the `cond` field, in bits 31-28:

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field contains one of the values `0b0000-0b1110`, as shown in Table F1-1. Most instruction mnemonics can be extended with the letters defined in the Mnemonic extension column of that table.

If the *always* (`AL`) condition is specified, the instruction is executed irrespective of the value of the Condition flags. The absence of a Condition code on an instruction mnemonic implies the `AL` Condition code.

F1.3.2 Pseudocode description of conditional execution

The `CurrentCond()` function returns a 4-bit condition specifier as follows:

- For A32 instructions, it returns bits[31:28] of the instruction.
- For the T1 and T3 encodings of the Branch instruction (see B), it returns the 4-bit `cond` field of the encoding.
- For all other T32 instructions:
  - If `PSTATE.IT<3:0> != '0000'` it returns `PSTATE.IT<7:4>`.
  - If `PSTATE.IT<7:0> == '00000000'` it returns `'1110'`.
  - Otherwise, execution of the instruction is CONSTRAINED UNPREDICTABLE.

  For more information, see Process state, PSTATE.

The `ConditionPassed()` function uses this condition specifier and the Condition flags to determine whether the instruction must be executed, by calling the `ConditionHolds()` function.

A-profile Architecture Pseudocode includes the definitions of these functions.

Undefined Instruction exception describes the handling of conditional instructions that are UNDEFINED, UNPREDICTABLE, or CONSTRAINED UNPREDICTABLE. The pseudocode in the manual, as a sequential description of the instructions, has limitations in this respect. For more information, see Limitations of the instruction pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.4 Shifts applied to a register

A32 register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both T32 and A32 data-processing instructions can apply the same range of different constant shifts to the second operand register. For details, see Constant shifts.

A32 data-processing instructions can apply a register-controlled shift to the second operand register.

F1.4.1 Constant shifts

These are the same in T32 and A32 instructions, except that the input bits come from different positions.

`<shift>` is an optional shift to be applied to `<Rm>`. It can be any one of:

(omitted) No shift.

`LSL #<n>` Logical shift left `<n>` bits. 1 <= `<n>` <= 31.

`LSR #<n>` Logical shift right `<n>` bits. 1 <= `<n>` <= 32.

`ASR #<n>` Arithmetic shift right `<n>` bits. 1 <= `<n>` <= 32.

`ROR #<n>` Rotate right `<n>` bits. 1 <= `<n>` <= 31.

`RRX` Rotate right one bit, with extend. Bit[0] is written to `shifter_carry_out`, bits[31:1] are shifted right one bit, and the Carry flag is shifted into bit[31].

Note

Assemblers can permit the use of some or all of `ASR #0`, `LSL #0`, `LSR #0`, and `ROR #0` to specify that no shift is to be performed. This is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must omit the shift specifier when the instruction specifies no shift.
Similarly, assemblers can permit the use of `#0` in the immediate forms of `ASR`, `LSL`, `LSR`, and `ROR` instructions to specify that no shift is to be performed, that is, that a `MOV` (register) instruction is wanted. Again, this is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must use the `MOV` (register) syntax when the instruction specifies no shift.

F1.4.1.1 Encoding

The assembler encodes `<shift>` into two type bits and five immediate bits, as follows:

(omitted) type = `0b00`, immediate = 0.

`LSL #<n>` type = `0b00`, immediate = `<n>`.

`LSR #<n>` type = `0b01`.

If `<n>` < 32, immediate = `<n>`.

If `<n>` == 32, immediate = 0.

`ASR #<n>` type = `0b10`.

If `<n>` < 32, immediate = `<n>`.

If `<n>` == 32, immediate = 0.

`ROR #<n>` type = `0b11`, immediate = `<n>`.

`RRX` type = `0b11`, immediate = 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.4.2 Register controlled shifts

These are available only in A32 instructions.

`<type>` is the type of shift to apply to the value read from `<Rm>`. It must be one of:

`ASR`	Arithmetic shift right, encoded as type = `0b10`.
`LSL`	Logical shift left, encoded as type = `0b00`.
`LSR`	Logical shift right, encoded as type = `0b01`.
`ROR`	Rotate right, encoded as type = `0b11`.

The bottom byte of `<Rs>` contains the shift amount.

F1.4.3 Pseudocode description of instruction-specified shifts and rotates

The pseudocode enumeration `SRType {}` defines the shift types. Shift and rotate instruction decode is described by the pseudocode function:

- `DecodeImmShift()` for a constant shift.
- `DecodeRegShift()` for a register controlled shift.

Shift and rotate operations are made by the pseudocode function `Shift()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

```
[<Rn>, <offset>]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>, <offset>]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register

The assembly language syntax for this mode is:

```
[<Rn>], <offset>
```

In each case, `<Rn>` is the base register. `<offset>` can be:

- An immediate constant, such as `<imm8>` or `<imm12>`.
- An index register, `<Rm>`.
- A shifted index register, such as `<Rm>, LSL #<shift>`.

For information about unaligned access, endianness, and exclusive access, see:

- Alignment support.
- Endian support.
- Synchronization and semaphores.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.6 Encoding of lists of general-purpose registers and the PC

A number of instructions operate on lists of general-purpose registers. For some load instructions, the list of registers to be loaded can include the PC. For these instructions, the assembler syntax includes a `<registers>` field, which provides a list of the registers to be operated on, with list entries separated by commas.

The registers list is encoded in the instruction encoding. Most often, this is done using an 8-bit, 13-bit, or 16-bit `register_list` field. This section gives more information about these and other possible register list encodings.

In a `register_list` field, each bit corresponds to a single register, and if the `<registers>` field of the assembler instruction includes Rt then `register_list<t>` is set to 1, otherwise it is set to 0.

The full rules for the encoding of lists of general-purpose registers, and possibly the PC, are:

- Except for the cases listed here, 16-bit T32 encodings use an 8-bit register list, and can access only registers R0-R7.

  The exceptions to this rule are:

  — The T1 encoding of `POP` uses an 8-bit register list, and an additional bit, `P`, that corresponds to the PC. This means it can access any of R0-R7 and the PC.

  — The T1 encoding of `PUSH` uses an 8-bit register list, and an additional bit, `M`, that corresponds to the LR. This means it can access any of R0-R7 and the LR.
- 32-bit T32 encodings of load operations use a 13-bit register list, and two additional bits, `M`, corresponding to the LR, and `P`, corresponding to the PC. This means these instructions can access any of R0-R12 and the LR and PC.
- 32-bit T32 encodings of store operations use a 13-bit register list, and one additional bit, `M`, corresponding to the LR. This means these instructions can access any of R0-R12 and the LR.
- Except for the case listed here, A32 encodings use a 16-bit register list. This means these instructions can access any of R0-R12 and the SP, LR, and PC.

  The exception to this rule is:

  — The System instructions `LDM` (exception return) and `LDM` (User registers) use a 15-bit register list. This means these instructions can access any of R0-R12 and the SP and LR.
- The T3 and A2 encodings of `POP`, and the T3 and A2 encodings of `PUSH`, access a single register from the set of registers {R0-R12, LR, PC} and encode the register number in the `Rt` field.

Note

`POP` is a load operation, and `PUSH` is a store operation.

In every case, the encoding-specific pseudocode converts the register list into a 32-bit variable, `registers`, with a bit corresponding to each of the registers R0-R12, SP, LR, and PC.

Note

Some Advanced SIMD and floating-point instructions operate on lists of SIMD and floating-point registers. The assembler syntax of these instructions includes a `<list>` field that specifies the registers to be operated on, and the description of the instruction in Alphabetical list of T32 and A32 base instruction set instructions defines the use and encoding of this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7 General information about the T32 and A32 instruction descriptions

T32 Instruction Set Encoding describes the T32 instruction encodings, and A32 Instruction Set Encoding describes the A32 instruction encodings. The following subsections give more information about the descriptions of these instructions and their encodings:

- Execution of instructions in debug state.
- Fixed values in AArch32 instruction and System register descriptions.
- UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space.
- T32 and A32 Advanced SIMD and floating-point instruction encodings.
- The PC and the use of `0b1111` as a register specifier in T32 and A32 instructions.
- The SP and the use of `0b1101` as a register specifier in T32 and A32 instructions.
- Modified immediate constants in T32 and A32 instructions.

F1.7.1 Execution of instructions in debug state

In general, except for the instructions described in Debug state, the T32 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see Executing instructions in Debug state.

Note

- A32 instructions cannot be executed in Debug state.
- For many T32 instructions, execution is unchanged in Debug state. Executing instructions in Debug state identifies these instructions.

F1.7.2 Fixed values in AArch32 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See Read-As-Zero (RAZ).

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, Should-Be-Zero (SBZ) or RES0.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. For more information, see the Glossary definition of RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

The `(0)` and RES0 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

RAO Read-As-One. See Read-As-One (RAO).

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, Should-Be-One (SBO) or RES1.

In instruction encoding diagrams, and sometimes in other descriptions, `(1)` indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(1)(1)(1)` or as `(111)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. For more information, see the Glossary definition of *RES1*.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields.

The `(1)` and RES1 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

Note

In register diagrams, (0) is a synonym for RES0, and (1) is a synonym for RES1, where RES0 and RES1 are defined in the Glossary. However, when used in an instruction encoding diagram, (0) and (1) have the narrower definition that behavior is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE if either:

- A bit marked as (0) has the value 1.
- A bit marked as (1) has the value 0.

F1.7.3 UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE.

  From the introduction of Armv8-A, the architecturally UNPREDICTABLE behavior in AArch32 state is greatly reduced. Most cases that earlier versions of the architecture describe as UNPREDICTABLE become either:

  — CONSTRAINED UNPREDICTABLE, meaning the architecture defines a limited range of permitted behaviors.

  — Fully predictable.

  For more information, see Architectural Constraints on UNPREDICTABLE Behaviors.
- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in T32 Instruction Set Encoding or A32 Instruction Set Encoding.

An instruction is UNPREDICTABLE only if:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- It is declared as UNPREDICTABLE in an instruction description or in T32 Instruction Set Encoding or A32 Instruction Set Encoding, and Architectural Constraints on UNPREDICTABLE Behaviors does not redefine the behavior as CONSTRAINED UNPREDICTABLE.
- The pseudocode for that encoding does not indicate that a different special case applies, and a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively. In most cases, Armv8 makes these cases CONSTRAINED UNPREDICTABLE, as described in SBZ or SBO fields T32 and A32 in instructions.

Unless otherwise specified, T32 and A32 instructions provided as part of an architectural extension, or by an optional feature of the architecture, are UNDEFINED in an implementation that does not include that extension or feature.

Note

Examples of where this rule applies are:

- The instructions provided by the Cryptographic Extension.
- The System instructions that provide access to the System registers of the OPTIONAL Performance Monitors Extension.
- The Advanced SIMD and floating-point instructions.

For more information about UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction behavior, see Undefined Instruction exception.

F1.7.4 T32 and A32 Advanced SIMD and floating-point instruction encodings

The T32 and A32 encodings of Advanced SIMD and floating-point instructions that are described in T32 Instruction Set Encoding and in A32 Instruction Set Encoding are common to the T32 and A32 instruction sets. This means:

- The instruction groups, and the set of instructions in each group, are identical for T32 and A32.
- For each instruction:
  - Each T32 encoding is exactly equivalent to an A32 encoding.
  - There is no T32 encoding without an equivalent A32 encoding, and no A32 encoding without an equivalent T32 encoding.

Note

- In the T32 instruction sets, the Advanced SIMD and floating-point instructions have 32-bit encodings.
- In the base instruction sets, some instructions are common to the T32 and A32 instruction sets, whereas other instructions have equivalent but not identical functionality in the two instruction sets.

32-bit T32 encodings are described as two contiguous halfwords, `{hw1:hw2}`, as described in Instruction encodings. In general:

- `hw1` of a T32 encoding maps onto bits[31:16] of an equivalent A32 encoding.
- `hw2` of a T32 encoding maps onto bits[15:0] of an equivalent A32 encoding.

However, the different structures of the T32 instruction encoding space and the A32 instruction encoding space mean that:

- For a given Advanced SIMD and floating-point instruction group:
  - The positions of the fields that identify the instruction, or instruction encoding, within the instruction group might differ between the T32 encodings and the A32 encodings.
  - However, the field values that identify the instruction of instruction encoding are identical for the T32 encoding and the A32 encoding.

The remainder of this section describes the equivalence of the T32 and A32 encodings for each of the Advanced SIMD and floating-point instruction groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7.4.1 Advanced SIMD data-processing

The T32 encoding of the Advanced SIMD data-processing group is:

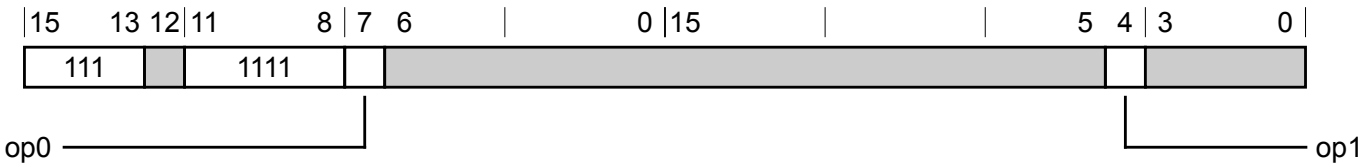

The A32 encoding of the Advanced SIMD data-processing group is:

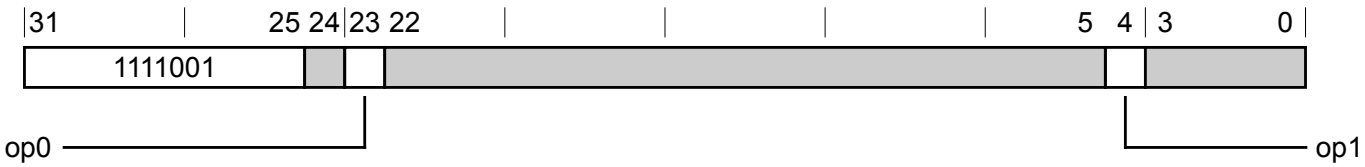

The encodings in this group are identified by:

- `hw1`[15:13] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and:
  - Has the value `0b111` in the T32 encoding.
  - Has the value `0b001` in the A32 encoding.
- `hw1`[11:8] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and has the value `0b111`.

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

T32 encoding	A32 encoding	Field size
`op0:op1`	`op0`	2 bits
`op2`	`op1`	15 bits
`op3`	`op2`	1 bit
`op4`	`op3`	1 bit

F1.7.4.2 Advanced SIMD element or structure load/store

The T32 encoding of the Advanced SIMD element or structure load/store group is:

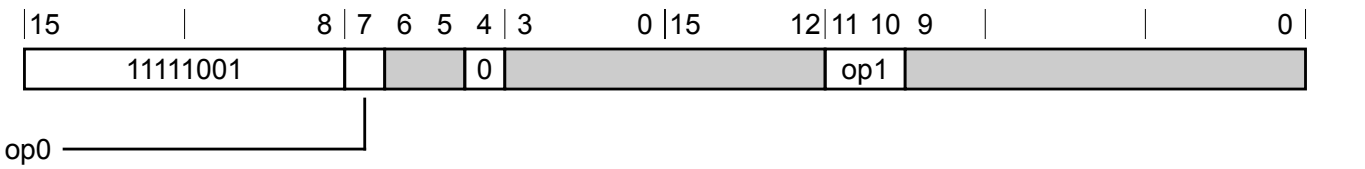

The A32 encoding of the Advanced SIMD element or structure load/store group is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

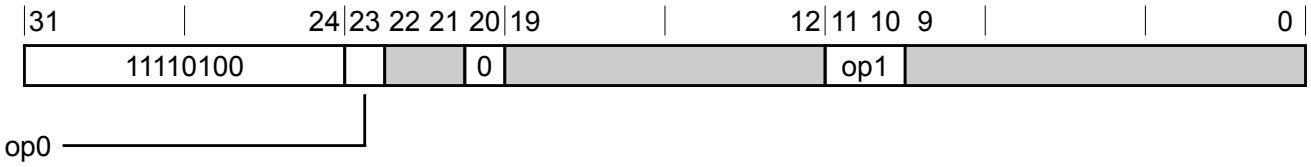

The encodings in this group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and has the value `0b1111`.
- `hw1`[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and:
  - Has the value `0b1001` in the T32 encoding.
  - Has the value `0b0100` in the A32 encoding.
- `hw1`[4] of the T32 encoding is equivalent to bit[20] of the A32 encoding, and has the value `0b0`.

`op0`, `op1`, and `op2` are the fields that identify the instructions, or instruction encodings, within this group, and they are in equivalent positions in the T32 and A32 encodings.

F1.7.4.3 Advanced SIMD and floating-point load/store and 64-bit register moves

The T32 encoding of the Advanced SIMD and floating-point load/store and 64-bit register moves group is:

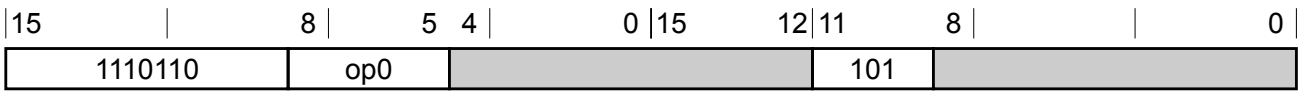

The A32 encoding of the Advanced SIMD and floating-point load/store and 64-bit register moves group is:

The encodings in the group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - Has the value `0b1110` in the T32 encoding.
  - Can have any value other than `0b1111` in the A32 encoding.
    This range of values is required because A32 instructions in this group can be executed conditionally, see Conditional execution.
- `hw1`[11:9] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and has the value `0b110`.
- `hw2`[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.

`op0` is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7.4.4 Advanced SIMD and floating-point 32-bit register moves

The T32 encoding of the Advanced SIMD and floating-point 32-bit register moves group is:

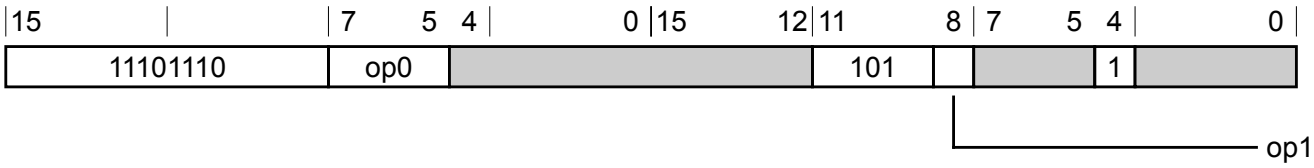

The A32 encoding of the Advanced SIMD 32-bit register moves group is:

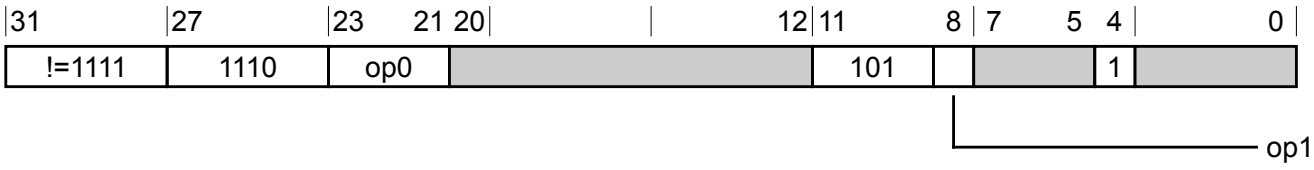

The encodings in this group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - Has the value `0b1110` in the T32 encoding.
  - Can have any value other than `0b1111` in the A32 encoding.
    This range of values is required because A32 instructions in this group can be executed conditionally, see Conditional execution.
- `hw1`[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value `0b1110`.
- `hw2`[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.
- `hw2`[4] of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value `0b1`.

`op0` is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

F1.7.4.5 Floating-point data-processing

The T32 encoding of the Floating-point data-processing group is:

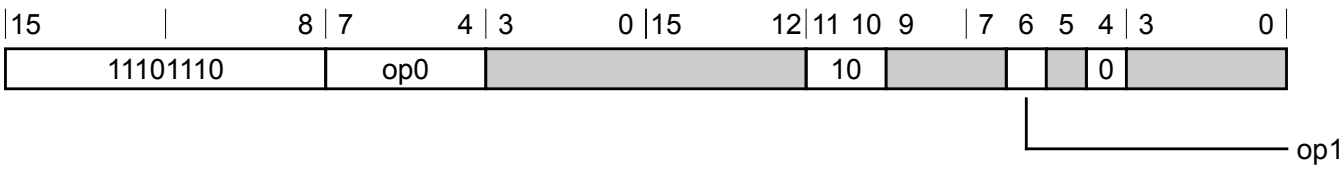

The A32 encoding of the Floating-point data-processing group is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encodings in this group are identified by:

- `hw1`[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - In the T32 encoding, `hw1`[15:13] has the value `0b111`, and `hw1`[12] is the `op0` parameter used in identifying instruction encodings within this group.
  - In the A32 encoding, is the `cond` field and also implies the value of bit[28] of some A32 instruction encodings within this group, as the following table shows:

cond	Significance of bit[28] in A32 encodings
!= `0b1111`	Part of the `cond` field.
`0b1111`	Has fixed value of 1.

The range of `cond` values other than `0b1111` is required because A32 instructions in this group can be executed conditionally, see Conditional execution.

- `hw1`[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value `0b1110`.
- `hw2`[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.
- `hw2`[4] of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value `0b0`.

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

T32 encoding	A32 encoding
`op0`	Bit[28] of the instruction encoding is 1 when `cond` is `0b1111`.
`op1`	`op0`
`op2`	`op1`
`op3`	`op2`

F1.7.5 The PC and the use of `0b1111` as a register specifier in T32 and A32 instructions

Restrictions on the use of PC or `0b1111` as a register specifier differ between the T32 and the A32 instruction sets, as described in:

- T32 restrictions on the use of the PC, and use of `0b1111` as a register specifier.
- A32 restrictions on the use of PC or `0b1111` as a register specifier.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7.5.1 T32 restrictions on the use of the PC, and use of `0b1111` as a register specifier

The use of `0b1111` as a register specifier is not normally permitted in T32 instructions. When a value of `0b1111` is permitted, various meanings are possible. For register reads, these meanings include:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions `TBB` and `TBH` can be the PC. This means branch tables can be placed in memory immediately after the instruction.

  Note

  Arm deprecates use of the PC as the base register in the `STC` instruction.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits[1:0] forced to zero. The base register of `LDC`, `LDR`, `LDRB`, `LDRD` (pre-indexed, no writeback), `LDRH`, `LDRSB`, and `LDRSH` instructions can be the word-aligned PC. This provides PC-relative data addressing. In addition, some encodings of the `ADD` and `SUB` instructions permit their source registers to be `0b1111` for the same purpose.
- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings include:

- The PC can be specified as the destination register of an `LDR` instruction. This is done by encoding Rt as `0b1111`. The loaded value is treated as an address, and the effect of execution is a branch to that address. Bit[0] of the loaded value selects whether to execute A32 or T32 instructions after the branch.
- Some other instructions write the PC in similar ways. An instruction can specify that the PC is written:
  - Implicitly, for example, branch instructions.
  - Explicitly by a register specifier of `0b1111`, for example 16-bit `MOV` (register) instructions.
  - Explicitly by using a register mask, for example `LDM` instructions.

  The address to branch to can be:
  - A loaded value, for example, `RFE`.
  - A register value, for example, `BX`.
  - The result of a calculation, for example, `TBB` or `TBH`.

  The method of choosing the instruction set used after the branch can be:
  - Similar to the `LDR` case, for example, `LDM` or `BX`.
  - A fixed instruction set other than the one currently being used, for example, the immediate form of `BLX`.
  - Unchanged, for example, branch instructions or 16-bit `MOV` (register) instructions.
  - Set from the SPSR.T bit, for `RFE` and `SUBS PC, LR, #imm8`.
- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.
- If the destination register specifier of an `LDRB`, `LDRH`, `LDRSB`, or `LDRSH` instruction is `0b1111`, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an `MRC` instruction is `0b1111`, bits[31:28] of the value transferred from the System register are written to the N, Z, C, and V condition flags in the APSR, and bits[27:0] are discarded.

F1.7.5.2 A32 restrictions on the use of PC or `0b1111` as a register specifier

In A32 instructions, the use of `0b1111` as a register specifier specifies the PC.

Many instructions are CONSTRAINED UNPREDICTABLE if they use `0b1111` as a register specifier. This is specified by pseudocode in the instruction description. Armv8-A constrains the resulting CONSTRAINED UNPREDICTABLE behavior, see Using R15 by instruction.

Note

Arm deprecates use of the PC as the base register in any store instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7.6 The SP and the use of `0b1101` as a register specifier in T32 and A32 instructions

In the T32 and A32 instruction sets, Arm recommends that the use of `0b1101` as a register specifier specifies the SP.

Note

- The recommendation that the register specifier `0b1101` is used only to specify the SP applies to both the T32 and the A32 instruction sets.
- Despite this recommendation, T32 instructions that can access R13, or the SP, behave predictably from the introduction of Armv8.

F1.7.7 Modified immediate constants in T32 and A32 instructions

The following sections describe the encoding of modified immediate constants:

- Modified immediate constants in T32 instructions.
- Modified immediate constants in A32 instructions.
- Modified immediate constants in T32 and A32 Advanced SIMD instructions.
- Modified immediate constants in T32 and A32 floating-point instructions.

F1.7.7.1 Modified immediate constants in T32 instructions

The encoding of a modified immediate constant in a 32-bit T32 instruction is:

15 14 13 12 11	10	9 8 7 6 5 4 3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
	i			imm3		a b c d	e f g h

Table F1-5 shows the range of modified immediate constants available in T32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, h, and i bits, and the imm3 field, in the instruction.

Table F1-5 Encoding of modified immediates in T32 data-processing instructions

i:imm3:a	<const>[a]
0000x	`00000000 00000000 00000000 abcdefgh`
0001x	`00000000 abcdefgh 00000000 abcdefgh`[b]
0010x	`abcdefgh 00000000 abcdefgh 00000000`[b]
0011x	`abcdefgh abcdefgh abcdefgh abcdefgh`[b]
01000	`1bcdefgh 00000000 00000000 00000000`
01001	`01bcdefg h0000000 00000000 00000000`[c]
01010	`001bcdef gh000000 00000000 00000000`
01011	`0001bcde fgh00000 00000000 00000000`[c]
.	. 8-bit values shifted to other positions
.	.
.	.
11101	`00000000 00000000 000001bc defgh000`[c]
11110	`00000000 00000000 0000001b cdefgh00`
11111	`00000000 00000000 00000001 bcdefgh0`[c]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

a. This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

b. Arm deprecates using a modified immediate with abcdefgh == 00000000, and these cases are CONSTRAINED UNPREDICTABLE, see UNPREDICTABLE cases in immediate constants in T32 data-processing instructions.

c. Not available in A32 instructions if `h` == 1.

Note

As the footnotes to Table F1-5 show, the range of values available in T32 modified immediate constants is slightly different from the range of values available in A32 instructions. See Modified immediate constants in A32 instructions for the A32 values.

F1.7.7.1.1 Carry out

A logical instruction with i::imm3::a == '00xxx' does not affect the Carry flag. Otherwise, a logical flag-setting instruction sets the Carry flag to the value of bit[31] of the modified immediate constant.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7.7.1.2 Operation of modified immediate constants, T32 instructions

For a T32 data-processing instruction, the `T32ExpandImm()` pseudocode function returns the value of the 32-bit immediate constant, calling `T32ExpandImm_C()` to evaluate the constant.

F1.7.7.2 Modified immediate constants in A32 instructions

The encoding of a modified immediate constant in an A32 instruction is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																				rotation				a	b	c	d	e	f	g	h

Table F1-6 shows the range of modified immediate constants available in A32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, and h bits and the rotation field in the instruction.

Table F1-6 Encoding of modified immediates in A32 processing instructions

rotation	<const>[a]
0000	`00000000 00000000 00000000 abcdefgh`
0001	`gh000000 00000000 00000000 00abcdef`
0010	`efgh0000 00000000 00000000 0000abcd`
0011	`cdefgh00 00000000 00000000 000000ab`
0100	`abcdefgh 00000000 00000000 00000000`
. . .	. . . 8-bit values shifted to other even-numbered positions
1001	`00000000 00abcdef gh000000 00000000`
. . .	. . . 8-bit values shifted to other even-numbered positions
1110	`00000000 00000000 0000abcd efgh0000`
1111	`00000000 00000000 000000ab cdefgh00`

a. This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

Note

The range of values available in A32 modified immediate constants is slightly different from the range of values available in 32-bit T32 instructions. See Modified immediate constants in T32 instructions.

F1.7.7.2.1 Carry out

A logical instruction with the rotation field set to `0b0000` does not affect APSR.C. Otherwise, a logical flag-setting instruction sets APSR.C to the value of bit[31] of the modified immediate constant.

F1.7.7.2.2 Constants with multiple encodings

Some constant values have multiple possible encodings. In this case, a UAL assembler must select the encoding with the lowest unsigned value of the rotation field. This is the encoding that appears first in Table F1-6. For example, the constant `#3` must be encoded with (rotation, abcdefgh) == (`0b0000`, `0b00000011`), not (`0b0001`, `0b00001100`), (`0b0010`, `0b00110000`), or (`0b0011`, `0b11000000`).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In particular, this means that all constants in the range 0-255 are encoded with rotation == `0b0000`, and permitted constants outside that range are encoded with rotation != `0b0000`. A flag-setting logical instruction with a modified immediate constant therefore leaves APSR.C unchanged if the constant is in the range 0-255 and sets it to the most significant bit of the constant otherwise. This matches the behavior of T32 modified immediate constants for all constants that are permitted in both the T32 and A32 instruction sets.

An alternative syntax is available for a modified immediate constant that permits the programmer to specify the encoding directly. In this syntax, `#<const>` is instead written as `#<byte>, #<rot>`, where:

`<byte>` Is the numeric value of abcdefgh, in the range 0-255.

`<rot>` Is twice the numeric value of rotation, an even number in the range 0-30.

This syntax permits all A32 data-processing instructions with modified immediate constants to be disassembled to assembler syntax that assembles to the original instruction.

This syntax also makes it possible to write variants of some flag-setting logical instructions that have different effects on APSR.C to those obtained with the normal `#<const>` syntax. For example, `ANDS R1, R2, #12, #2` has the same behavior as `ANDS R1, R2, #3` except that it sets APSR.C to 0 instead of leaving it unchanged. Such variants of flag-setting logical instructions do not have equivalents in the T32 instruction set, and Arm deprecates their use.

F1.7.7.2.3 Operation of modified immediate constants, A32 instructions

For an A32 data-processing instruction, the `A32ExpandImm()` pseudocode function returns the value of the 32-bit immediate constant, calling `A32ExpandImm_C()` to evaluate the constant.

F1.7.7.3 Modified immediate constants in T32 and A32 Advanced SIMD instructions

Table F1-7 shows the modified immediate constants available with Advanced SIMD instructions, and how they are encoded.

Table F1-7 Modified immediate values for Advanced SIMD instructions

op	cmode	Constant[a]	<dt>[b]	Notes
-	000x	00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh	I32	[c]
	001x	00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000	I32	[c,d]
	010x	00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000	I32	[c,d]
	011x	abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000	I32	[c,d]
	100x	00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh	I16	[c]
	101x	abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000	I16	[c,d]
	1100	00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh 11111111	I32	[d,e]
	1101	00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111 11111111	I32	[d,e]
0	1110	abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh	I8	[f]
	1111	aBbbbbbc defgh000 00000000 00000000 aBbbbbbc defgh000 00000000 00000000	F32	[f,g]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	cmode	Constant[a]	<dt>[b]	Notes
1	1110	`aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh`	`I64`	[f]
	1111	UNDEFINED	-	-

a. In this table, the immediate value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembler syntax, the constant is specified by a data type and a value of that type. That value is specified in the normal way (a decimal number by default) and is replicated enough times to fill the 64-bit immediate. For example, a data type of `I32` and a value of 10 specify the 64-bit constant `0x0000000A0000000A`.

b. This specifies the data type used when the instruction is disassembled. On assembly, the data type must be matched in the table if possible. Other data types are permitted as pseudo-instructions when a program is assembled, provided the 64-bit constant specified by the data type and value is available for the instruction. If a constant is available in more than one way, the first entry in this table that can produce it is used. For example, `VMOV.I64 D0, #0x8000000080000000` does not specify a 64-bit constant that is available from the I64 line of the table, but does specify one that is available from the fourth I32 line or the F32 line. It is assembled to the first of these, and therefore is disassembled as `VMOV.I32 D0, #0x80000000`.

c. This constant is available for the `VBIC`, `VMOV`, `VMVN`, and `VORR` instructions.

d. CONSTRAINED UNPREDICTABLE if `abcdefgh` == `0b00000000`, see UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions. The required behavior is that these encodings produce an immediate constant of zero.

e. This constant is available for the `VMOV` and `VMVN` instructions only.

f. This constant is available for the `VMOV` instruction only.

g. In this entry, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.7.7.3.1 Operation of modified immediate constants, Advanced SIMD instructions

For a T32 or A32 Advanced SIMD instruction that uses a modified immediate constant, the operation described by the `AdvSIMDExpandImm()` pseudocode function returns the value of the 64-bit immediate constant.

F1.7.7.4 Modified immediate constants in T32 and A32 floating-point instructions

Table F1-8 shows the immediate constants available in the `VMOV` (immediate) floating-point instruction, and Table F1-9 shows the resulting floating-point values.

Table F1-8 Floating-point modified immediate constants

Data type	imm4H	imm4L	Constant [a]
`F16`	abcd	efgh	`aBbbcdef gh000000`
`F32`	abcd	efgh	`aBbbbbbc defgh000 00000000 00000000`
`F64`	abcd	efgh	`aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000`

a. In this column, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

Table F1-9 Floating-point constant values

efgh	bcd							
	`000`	`001`	`010`	`011`	`100`	`101`	`110`	`111`
`0000`	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
`0001`	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
`0010`	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
`0011`	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
`0100`	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
`0101`	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
`0110`	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
`0111`	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
`1000`	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
`1001`	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
`1010`	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
`1011`	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
`1100`	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
`1101`	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
`1110`	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
`1111`	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

F1.7.7.4.1 Operation of modified immediate constants, floating-point instructions

For a T32 or A32 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.8 Additional pseudocode support for instruction descriptions

Earlier sections of this chapter include pseudocode that describes features of the execution of T32 and A32 instructions, see:

- Pseudocode description of conditional execution.
- Pseudocode description of instruction-specified shifts and rotates

The following subsection gives additional pseudocode support functions for some of the instructions described in Alphabetical list of T32 and A32 base instruction set instructions. See also Pseudocode support for the banked register transfer instructions.

F1.8.1 Pseudocode description of operations for System register access instructions

The `AArch32_SysRegRead()` function obtains the word for an `MRC` instruction from the System register.

The `AArch32_SysRegRead64()` function obtains the two words for an `MRRC` instruction from the System register.

Note

The relative significance of the two words returned is IMPLEMENTATION DEFINED, but all uses within this manual present the two words in the order (most significant, least significant).

The `AArch32_SysRegWrite()` procedure sends the word for an `MCR` instruction to the System register.

The `AArch32_SysRegWrite64()` procedure sends the two words for an `MCRR` instruction to the System register.

Note

The relative significance of `word2` and `word1` is IMPLEMENTATION DEFINED, but all uses within this manual treat `word2` as more significant than `word1`.

F1.8.2 Pseudocode details of system calls

The `AArch32_CallSupervisor()` pseudocode function generates a Supervisor Call exception. Valid execution of the `SVC` instruction calls this function.

The `AArch32_CallHypervisor()` pseudocode function generates an HVC exception. Valid execution of the `HVC` instruction calls this function.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.9 Additional information about Advanced SIMD and floating-point instructions

The following subsections give additional information about the Advanced SIMD and floating-point instructions:

- Advanced SIMD and floating-point instruction syntax.
- The Advanced SIMD addressing mode.
- Advanced SIMD instruction modifiers.
- Advanced SIMD operand shapes.
- Data type specifiers.
- Register specifiers.
- Register lists.
- Register encoding.
- Advanced SIMD scalars.

Note

The Advanced SIMD architecture, its associated implementations, and supporting software, are commonly referred to as NEON™ technology.

F1.9.1 Advanced SIMD and floating-point instruction syntax

Advanced SIMD and floating-point instructions use the general conventions of the T32 and A32 instruction sets.

Advanced SIMD and floating-point data-processing instructions use the following general format:

```
V{<modifier>}<operation>{<shape>}{<c>}{<q>}{.<dt>}  {<dest>,} <src1>, <src2>
```

All Advanced SIMD and floating-point instructions begin with a `V`. This distinguishes Advanced SIMD vector and floating-point instructions from scalar instructions.

The main operation is specified in the `<operation>` field. It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.

The `<c>` and `<q>` fields are standard assembler syntax fields. For details, see Standard assembler syntax fields.

F1.9.2 The Advanced SIMD addressing mode

All the element and structure load/store instructions use this addressing mode. There is a choice of three formats:

`[<Rn>{:<align>}]` The address is contained in general-purpose register Rn.

Rn is not updated by this instruction.

Encoded as Rm = `0b1111`.

If Rn is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

`[<Rn>{:<align>}]!` The address is contained in general-purpose register Rn.

Rn is updated by this instruction: `Rn = Rn + transfer_size`

Encoded as Rm = `0b1101`.

`transfer_size` is the number of bytes transferred by the instruction. This means that, after the instruction is executed, Rn points to the address in memory immediately following the last address loaded from or stored to.

If Rn is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

This addressing mode can also be written as:

`[<Rn>{:align}], #<transfer_size>`

However, disassembly produces the `[<Rn>{:align}]!` form.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`[<Rn>{:<align>}], <Rm>`

The address is contained in general-purpose register `<Rn>`.

Rn is updated by this instruction: `Rn = Rn + Rm`

Encoded as Rm = Rm. Rm must not be encoded as `0b1111` or `0b1101`, the PC or the SP.

If Rn is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

The CONSTRAINED UNPREDICTABLE behavior of encodings where Rn is `0b1111` is described in the section: Using R15 by instruction.

In all cases, `<align>` specifies an alignment, as specified by the individual instruction descriptions.

Previous versions of the manual used the `@` character for alignment. So, for example, the first format in this section was shown as `[<Rn>{@<align>}]`. Both `@` and `:` are supported. However, to ensure portability of code to assemblers that treat `@` as a comment character, `:` is preferred.

F1.9.3 Advanced SIMD instruction modifiers

The `<modifier>` field provides additional variants of some instructions. Table F1-10 provides definitions of the modifiers. Modifiers are not available for every instruction.

Table F1-10 Advanced SIMD instruction modifiers

<modifier>	Meaning
Q	The operation uses saturating arithmetic.
R	The operation performs rounding.
D	The operation doubles the result (before accumulation, if any).
H	The operation halves the result.

F1.9.4 Advanced SIMD operand shapes

The `<shape>` field provides additional variants of some instructions. Table F1-11 provides definitions of the shapes. Operand shapes are not available for every instruction.

Table F1-11 Advanced SIMD operand shapes

<shape>	Meaning	Typical register shape	
(none)	The operands and result are all the same width.	Dd, Dn, Dm	Qd, Qn, Qm
L	Long operation - result is twice the width of both operands	Qd, Dn, Dm	
N	Narrow operation - result is half the width of both operands	Dd, Qn, Qm	
W	Wide operation - result and first operand are twice the width of the second operand	Qd, Qn, Dm	

Note

- Some assemblers support a Q shape specifier, which requires all operands to be Q registers. An example of using this specifier is `VADDQ.S32 q0, q1, q2`. This is not standard UAL, and Arm recommends that programmers do not use a Q shape specifier.
- A disassembler must not generate any shape specifier not shown in Table F1-11.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.9.5 Data type specifiers

The `<dt>` field normally contains one data type specifier. Unless the assembler syntax description for the instruction indicates otherwise, this indicates the data type contained in:

- The second operand, if any.
- The operand, if there is no second operand.
- The result, if there are no operand registers.

The data types of the other operand and result are implied by the `<dt>` field combined with the instruction shape. For information about data type formats, see Data types supported by the Advanced SIMD implementation.

In the instruction syntax descriptions in About the T32 and A32 Instruction Descriptions, the `<dt>` field is usually specified as a single field. However, where more convenient, it is sometimes specified as a concatenation of two fields, `<type><size>`.

F1.9.5.1 Syntax flexibility

There is some flexibility in the data type specifier syntax:

- Software can specify three data types, specifying the result and both operand data types. For example:
  `VSUBW.I16.I16.S8 Q3, Q5, D0` instead of `VSUBW.S8 Q3, Q5, D0`
- Software can specify two data types, specifying the data types of the two operands. The data type of the result is implied by the instruction shape. For example:
  `VSUBW.I16.S8 Q3, Q5, D0` instead of `VSUBW.S8 Q3, Q5, D0`
- Software can specify two data types, specifying the data types of the single operand and the result. For example:
  `VMOVN.I16.I32 D0, Q1` instead of `VMOVN.I32 D0, Q1`
- Where an instruction requires a less specific data type, software can instead specify a more specific type, as shown in Table F1-12.
- Where an instruction does not require a data type, software can provide one.
- The `F32` data type can be abbreviated to `F`.
- The `F64` data type can be abbreviated to `D`.

In all cases, if software provides additional information, the additional information must match the instruction shape. Disassembly does not regenerate this additional information.

Table F1-12 Data type specification flexibility

Specified data type	Permitted more specific data types				
None	Any				
`.I<size>`	-	`.S<size>`	`.U<size>`	-	-
`.8`	`.I8`	`.S8`	`.U8`	`.P8`	-
`.16`	`.I16`	`.S16`	`.U16`	`.P16`	`.F16`
`.32`	`.I32`	`.S32`	`.U32`	-	`.F32` or `.F`
`.64`	`.I64`	`.S64`	`.U64`	-	`.F64` or `.D`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.9.6 Register specifiers

The `<dest>`, `<src1>`, and `<src2>` fields contain register specifiers, or in some cases scalar specifiers or register lists. Table F1-13 shows the register and scalar specifier formats that appear in the instruction descriptions.

If `<dest>` is omitted, it is the same as `<src1>`.

Table F1-13 Advanced SIMD and floating-point register specifier formats

<specifier>	Usual meaning[a]	Used in
`<Qd>`	A quadword destination register for the result vector.	Advanced SIMD
`<Qn>`	A quadword source register for the first operand vector.	Advanced SIMD
`<Qm>`	A quadword source register for the second operand vector.	Advanced SIMD
`<Dd>`	A doubleword destination register for the result vector.	Both
`<Dn>`	A doubleword source register for the first operand vector.	Both
`<Dm>`	A doubleword source register for the second operand vector.	Both
`<Sd>`	A singleword destination register for the result vector.	Floating-point
`<Sn>`	A singleword source register for the first operand vector.	Floating-point
`<Sm>`	A singleword source register for the second operand vector.	Floating-point
`<Dd[x]>`	A destination scalar for the result. Element x of vector `<Dd>`.	Advanced SIMD
`<Dn[x]>`	A source scalar for the first operand. Element x of vector `<Dn>`.	Both[b]
`<Dm[x]>`	A source scalar for the second operand. Element x of vector `<Dm>`.	Advanced SIMD
`<Rt>`	A general-purpose register, used for a source or destination address.	Both
`<Rt2>`	A general-purpose register, used for a source or destination address.	Both
`<Rn>`	A general-purpose register, used as a load or store base address.	Both
`<Rm>`	A general-purpose register, used as a post-indexed address source.	Both

a. In some instructions the roles of registers are different.

b. In the floating-point instructions, `<Dn[x]>` is used only in `VMOV` (scalar to general-purpose register), see VMOV (scalar to general-purpose register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.9.7 Register lists

A register list is a list of register specifiers separated by commas and enclosed in brackets { and }. There are restrictions on what registers can appear in a register list. These restrictions are described in the individual instruction descriptions. Table F1-14 shows some register list formats, with examples of actual register lists corresponding to those formats.

Note

Register lists must not wrap around the end of the register bank.

F1.9.7.1 Syntax flexibility

There is some flexibility in the register list syntax:

- Where a register list contains consecutive registers, they can be specified as a range, instead of listing every register, for example `{D0-D3}` instead of `{D0, D1, D2, D3}`.
- Where a register list contains an even number of consecutive doubleword registers starting with an even-numbered register, it can be written as a list of quadword registers instead, for example `{Q1, Q2}` instead of `{D2-D5}`.
- Where a register list contains only one register, the enclosing braces can be omitted, for example `VLD1.8 D0, [R0]` instead of `VLD1.8 {D0}, [R0]`.

Table F1-14 Example register lists

Format	Example	Alternative
`{<Dd>}`	`{D3}`	`D3`
`{<Dd>, <Dd+1>, <Dd+2>}`	`{D3, D4, D5}`	`{D3-D5}`
`{<Dd[x]>, <Dd+2[x]}`	`{D0[3], D2[3]}`	-
`{<Dd[]>}`	`{D7[]}`	`D7[]`

F1.9.8 Register encoding

An Advanced SIMD register is either:

- *Quadword*, meaning it is 128 bits wide.
- *Doubleword*, meaning it is 64 bits wide.

Some instructions have options for either doubleword or quadword registers. This is normally encoded in Q, bit[6], as Q = 0 for doubleword operations, or Q = 1 for quadword operations.

A floating-point register is either:

- Double-precision, meaning it is 64 bits wide.
- Single-precision, meaning it is 32 bits wide.

This is encoded in the `sz` field, bit[8], as sz = 1 for double-precision operations, or sz = 0 for single-precision operations.

The T32 instruction encoding of Advanced SIMD or floating-point registers is:

15 14 13 12 11 10 9 8 7	6	5 4	3 2 1 0	15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
	D		Vn	Vd		sz	N	Q	M		Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The A32 instruction encoding of Advanced SIMD or floating-point registers is:

31 30 29 28 27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
	D		Vn	Vd		sz	N	Q	M		Vm

Some instructions use only one or two registers, and use the unused register fields as additional opcode bits.

Table F1-15 shows the encodings for the registers.

Table F1-15 Encoding of register numbers

Register mnemonic	Usual usage	Register number encoded in[a]	Notes[a]	Used in
`<Qd>`	Destination (quadword)	D, Vd (bits[22, 15:13])	bit[12] == 0[b]	Advanced SIMD
`<Qn>`	First operand (quadword)	N, Vn (bits[7, 19:17])	bit[16] == 0[b]	Advanced SIMD
`<Qm>`	Second operand (quadword)	M, Vm (bits[5, 3:1])	bit[0] == 0[b]	Advanced SIMD
`<Dd>`	Destination (doubleword)	D, Vd (bits[22, 15:12])	-	Both
`<Dn>`	First operand (doubleword)	N, Vn (bits[7, 19:16])	-	Both
`<Dm>`	Second operand (doubleword)	M, Vm (bits[5, 3:0])	-	Both
`<Sd>`	Destination (single-precision)	Vd, D (bits[15:12, 22])	-	Floating-point
`<Sn>`	First operand (single-precision)	Vn, N (bits[19:16, 7])	-	Floating-point
`<Sm>`	Second operand (single-precision)	Vm, M (bits[3:0, 5])	-	Floating-point

a. Bit numbers given for the A32 instruction encoding. See the figures in this section for the equivalent bits in the T32 encoding.

b. If this bit is 1, the instruction is UNDEFINED.

F1.9.9 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table F1-16 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to Floating-point single-precision registers. That is, `Dm[x]` in a 32-bit context (0 <= `m` <= 15, 0 <= `x` <=1) is equivalent to `S[2m + x]`.

Table F1-16 Encoding of scalars in multiply instructions

Scalar mnemonic	Usual usage	Scalar size	Register specifier	Index specifier	Accessible registers
`<Dm[x]>`	Second operand	16-bit	Vm[2:0]	M, Vm[3]	D0-D7
		32-bit	Vm[3:0]	M	D0-D15

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F2
The AArch32 Instruction Sets Overview

This chapter describes the T32 and A32 instruction sets. It contains the following sections:

- Support for instructions in different versions of the Arm architecture.
- Unified Assembler Language.
- Branch instructions.
- Data-processing instructions.
- PSTATE and banked register access instructions.
- Load/store instructions.
- Load/store multiple instructions.
- Miscellaneous instructions.
- Exception-generating and exception-handling instructions.
- System register access instructions.
- Advanced SIMD and floating-point load/store instructions.
- Advanced SIMD and floating-point register transfer instructions.
- Advanced SIMD data-processing instructions.
- Floating-point data-processing instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.1 Support for instructions in different versions of the Arm architecture

This manual describes the T32 and A32 instruction sets for the architecture. Therefore, it indicates how any options or extensions in the architecture affect the available instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.2 Unified Assembler Language

This manual uses the Arm *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 and A32 instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Most earlier Arm assembly language mnemonics are still supported as synonyms, as described in the instruction details.

Note

Most earlier T32 assembly language mnemonics are *not* supported.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an `ADD R0, R1, R2` instruction. The most common instruction selection rule is that when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

F2.2.1 Conditional instructions

For maximum portability of UAL assembly language between the T32 and A32 instruction sets, Arm recommends that:

- `IT` instructions are written before conditional instructions in the correct way for the T32 instruction set.
- When assembling to the A32 instruction set, assemblers check that any `IT` instructions are correct, but do not generate any code for them.

Although other T32 instructions are unconditional, all instructions that are made conditional by an `IT` instruction must be written with a condition. These conditions must match the conditions imposed by the `IT` instruction. For example, an `ITTEE EQ` instruction imposes the `EQ` condition on the first two following instructions, and the `NE` condition on the next two. Those four instructions must be written with `EQ`, `EQ`, `NE` and `NE` conditions respectively.

Some instructions cannot be made conditional by an `IT` instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise.

The branch instruction encodings that include a condition code field cannot be made conditional by an `IT` instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding `IT` instruction, it is assembled using a branch instruction encoding that does not include a condition code field.

F2.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC, 4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a T32 instruction, or plus 8 for an A32 instruction. The `Align(PC, 4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC, 4)` values for an A32 instruction, but there can be for a T32 instruction.
2. Calculate the offset from the `PC` or `Align(PC, 4)` value of the instruction to the address of the labeled instruction or literal data item.
3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC, 4)` value and adds the calculated offset to form the required address.

   Note

   For instructions that can encode a subtraction operation, if the instruction cannot encode the calculated offset but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The syntax of the following instructions includes a label:

- `B`, `BL`, and `BLX` (immediate). The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch.
- `CBNZ` and `CBZ`. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch. They do not support backward branches.
- `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLDW`, `PLI`, and `VLDR`. The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

  When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the `Align(PC, 4)` value of the instruction. Encodings that subtract 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by `[PC, #+/-<imm>]`, where:

  `+/-` Is + or omitted to specify that the immediate offset is to be added to the `Align(PC, 4)` value, or – if it is to be subtracted.

  `<imm>` Is the immediate offset.

  This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC, 4)` value, and to disassemble them to a syntax that can be re-assembled correctly.
- `ADR`. The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

  When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC, 4)` value of the instruction. The encoding that subtracts 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions `ADD <Rd>, PC, #<imm>` or subtractions `SUB <Rd>, PC, #<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the `Align(PC, 4)` value, and to disassemble it to a syntax that can be re-assembled correctly.

Note

Arm recommends that where possible, software avoids using:

- The alternative syntax for the `ADR`, `LDC`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLI`, `PLDW`, and `VLDR` instructions.
- The encodings of these instructions that subtract 0 from the `Align(PC, 4)` value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.3 Branch instructions

Table F2-1 summarizes the branch instructions in the T32 and A32 instruction sets. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

Table F2-1 Branch instructions

Instruction	See	Range, T32	Range, A32
Branch to target address	B	±16MB	±32MB
Compare and Branch on Nonzero Compare and Branch on Zero	CBNZ, CBZ	0-126 bytes	-[a]
Call a subroutine Call a subroutine, change instruction set[b]	BL, BLX (immediate)	±16MB ±16MB	±32MB ±32MB
Call a subroutine, optionally change instruction set	BLX (register)	Any	Any
Branch to target address, change instruction set	BX	Any	Any
Change to Jazelle state	BXJ	-	-
Table Branch (byte offsets) Table Branch (halfword offsets)	TBB, TBH	0-510 bytes 0-131070 bytes	-[a]

a. These instructions do not exist in the A32 instruction set.

b. The range is determined by the instruction set of the BLX instruction, not of the instruction it branches to.

Branches to loaded and calculated addresses can be performed by `LDR`, `LDM` and data-processing instructions. For details, see Load/store instructions, Load/store multiple instructions, Standard data-processing instructions, and Shift instructions.

In addition to the branch instructions shown in Table F2-1:

- In the A32 instruction set, a data-processing instruction that targets the PC behaves as a branch instruction. For more information, see Data-processing instructions.
- In the T32 and A32 instruction sets, a load instruction that targets the PC behaves as a branch instruction. For more information, see Load/store instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- Standard data-processing instructions.
- Shift instructions.
- Multiply instructions.
- Saturating instructions.
- Saturating addition and subtraction instructions.
- Packing and unpacking instructions.
- Parallel addition and subtraction instructions.
- Divide instructions.
- Miscellaneous data-processing instructions.

For related Advanced SIMD and floating-point instructions see Advanced SIMD data-processing instructions and Floating-point data-processing instructions.

F2.4.1 Standard data-processing instructions

These instructions perform basic data-processing operations, and share a common format with some variations.

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be another register Rm, or an immediate constant.

If the second operand is an immediate constant, it can be:

- Encoded directly in the instruction.
- A *modified immediate constant* that uses 12 bits of the instruction to encode a range of constants. T32 and A32 instructions have slightly different ranges of modified immediate constants. For more information, see Modified immediate constants in T32 instructions and Modified immediate constants in A32 instructions.

If the second operand is another register, it can optionally be shifted in any of the following ways:

`LSL` Logical Shift Left by 1-31 bits.

`LSR` Logical Shift Right by 1-32 bits.

`ASR` Arithmetic Shift Right by 1-32 bits.

`ROR` Rotate Right by 1-31 bits.

`RRX` Rotate Right with Extend. For details, see Shift and rotate operations.

In T32 code, the amount to shift by is always a constant encoded in the instruction. In A32 code, the amount to shift by is either a constant encoded in the instruction, or the value of a register, Rs.

For instructions other than `CMN`, `CMP`, `TEQ`, and `TST`, the result of the data-processing operation is placed in the destination register. In the A32 instruction set, the destination register can be the PC, causing the result to be treated as a branch address. In the T32 instruction set, this is only permitted for some 16-bit forms of the `ADD` and `MOV` instructions.

These instructions can optionally set the Condition flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table F2-2 summarizes the main data-processing instructions in the T32 and A32 instruction sets. Generally, each of these instructions is described in three sections in About the T32 and A32 Instruction Descriptions, one section for each of the following:

- `INSTRUCTION` (immediate) where the second operand is a modified immediate constant.
- `INSTRUCTION` (register) where the second operand is a register, or a register shifted by a constant.
- `INSTRUCTION` (register-shifted register) where the second operand is a register shifted by a value obtained from another register. These are only available in the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F2-2 Standard data-processing instructions

Instruction	Mnemonic	Notes
Add with Carry	`ADC`	-
Add	`ADD`	T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Form PC-relative Address	`ADR`	First operand is the PC. Second operand is an immediate constant. T32 instruction set uses a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
Bitwise AND	`AND`	-
Bitwise Bit Clear	`BIC`	-
Compare Negative	`CMN`	Sets flags. Like `ADD` but with no destination register.
Compare	`CMP`	Sets flags. Like `SUB` but with no destination register.
Bitwise Exclusive OR	`EOR`	-
Copy operand to destination	`MOV`	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an `LSL`, `LSR`, `ASR`, or `ROR` instruction instead. For details, see Shift instructions. The T32 and A32 instruction sets permit use of a modified immediate constant or a zero-extended 16-bit immediate constant.
Bitwise NOT	`MVN`	Has only one operand, with the same options as the second operand in most of these instructions.
Bitwise OR NOT	`ORN`	Not available in the A32 instruction set.
Bitwise OR	`ORR`	-
Reverse Subtract	`RSB`	Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.
Reverse Subtract with Carry	`RSC`	Not available in the T32 instruction set.
Subtract with Carry	`SBC`	-
Subtract	`SUB`	T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Test Equivalence	`TEQ`	Sets flags. Like `EOR` but with no destination register.
Test	`TST`	Sets flags. Like `AND` but with no destination register.

F2.4.2 Shift instructions

Table F2-3 lists the shift instructions in the T32 and A32 instruction sets.

Table F2-3 Shift instructions

Instruction	See
Arithmetic Shift Right	ASR (immediate) ASR (register) ASRS (immediate) ASRS (register)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See
Logical Shift Left	LSL (immediate) LSL (register) LSLS (immediate) LSLS (register)
Logical Shift Right	LSR (immediate) LSR (register) LSRS (immediate) LSRS (register)
Rotate Right	ROR (immediate) ROR (register) RORS (immediate) RORS (register)
Rotate Right with Extend	RRX RRXS

In the A32 instruction set only, the destination register of these instructions can be the PC, causing the result to be treated as an address to branch to.

F2.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are the same whether the operands are signed or unsigned.

- Table F2-4 summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.
  The least significant 32 bits of the result are used. More significant bits are discarded.
- Table F2-5 summarizes the signed multiply instructions.
- Table F2-6 summarizes the unsigned multiply instructions.

Table F2-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	MLA, MLAS	$32 = 32 + 32 \times 32$
Multiply and Subtract	MLS	$32 = 32 - 32 \times 32$
Multiply	MUL, MULS	$32 = 32 \times 32$

Table F2-5 Signed multiply instructions

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate (halfwords)	SMLABB, SMLABT, SMLATB, SMLATT	$32 = 32 + 16 \times 16$
Signed Multiply Accumulate Dual	SMLAD, SMLADX	$32 = 32 + 16 \times 16 + 16 \times 16$
Signed Multiply Accumulate Long	SMLAL, SMLALS	$64 = 64 + 32 \times 32$

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate Long (halfwords)	SMLALBB, SMLALBT, SMLALTB, SMLALTT	64 = 64 + 16 × 16
Signed Multiply Accumulate Long Dual	SMLALD, SMLALDX	64 = 64 + 16 × 16 + 16 × 16
Signed Multiply Accumulate (word by halfword)	SMLAWB, SMLAWT	32 = 32 + 32 × 16[a]
Signed Multiply Subtract Dual	SMLSD, SMLSDX	32 = 32 + 16 × 16 - 16 × 16
Signed Multiply Subtract Long Dual	SMLSLD, SMLSLDX	64 = 64 + 16 × 16 - 16 × 16
Signed Most Significant Word Multiply Accumulate	SMMLA, SMMLAR	32 = 32 + 32 × 32[b]
Signed Most Significant Word Multiply Subtract	SMMLS, SMMLSR	32 = 32 - 32 × 32[b]
Signed Most Significant Word Multiply	SMMUL, SMMULR	32 = 32 × 32[b]
Signed Dual Multiply Add	SMUAD, SMUADX	32 = 16 × 16 + 16 × 16
Signed Multiply (halfwords)	SMULBB, SMULBT, SMULTB, SMULTT	32 = 16 × 16
Signed Multiply Long	SMULL, SMULLS	64 = 32 × 32
Signed Multiply (word by halfword)	SMULWB, SMULWT	32 = 32 × 16[a]
Signed Dual Multiply Subtract	SMUSD, SMUSDX	32 = 16 × 16 - 16 × 16

a. The most significant 32 bits of the 48-bit product are used. Less significant bits are discarded.

b. The most significant 32 bits of the 64-bit product are used. Less significant bits are discarded.

Table F2-6 Unsigned multiply instructions

Instruction	See	Operation (number of bits)
Unsigned Multiply Accumulate Accumulate Long	UMAAL	64 = 32 + 32 + 32 × 32
Unsigned Multiply Accumulate Long	UMLAL, UMLALS	64 = 64 + 32 × 32
Unsigned Multiply Long	UMULL, UMULLS	64 = 32 × 32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.4.4 Saturating instructions

Table F2-7 lists the saturating instructions in the T32 and A32 instruction sets. For more information, see Pseudocode description of saturation.

Table F2-7 Saturating instructions

Instruction	See	Operation
Signed Saturate	SSAT	Saturates optionally shifted 32-bit value to selected range
Signed Saturate 16	SSAT16	Saturates two 16-bit values to selected range
Unsigned Saturate	USAT	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate 16	USAT16	Saturates two 16-bit values to selected range

F2.4.5 Saturating addition and subtraction instructions

Table F2-8 lists the saturating addition and subtraction instructions in the T32 and A32 instruction sets. For more information, see Pseudocode description of saturation.

Table F2-8 Saturating addition and subtraction instructions

Instruction	See	Operation
Saturating Add	QADD	Add, saturating result to the 32-bit signed integer range
Saturating Subtract	QSUB	Subtract, saturating result to the 32-bit signed integer range
Saturating Double and Add	QADD	Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range
Saturating Double and Subtract	QDSUB	Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range

F2.4.6 Packing and unpacking instructions

Table F2-9 lists the packing and unpacking instructions in the T32 and A32 instruction sets.

Table F2-9 Packing and unpacking instructions

Instruction	See	Operation
Pack Halfword	PKHBT, PKHTB	Combine halfwords
Signed Extend and Add Byte	SXTAB	Extend 8 bits to 32 and add
Signed Extend and Add Byte 16	SXTAB16	Dual extend 8 bits to 16 and add
Signed Extend and Add Halfword	SXTAH	Extend 16 bits to 32 and add
Signed Extend Byte	SXTB	Extend 8 bits to 32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See	Operation
Signed Extend Byte 16	SXTB16	Dual extend 8 bits to 16
Signed Extend Halfword	SXTH	Extend 16 bits to 32
Unsigned Extend and Add Byte	UXTAB	Extend 8 bits to 32 and add
Unsigned Extend and Add Byte 16	UXTAB16	Dual extend 8 bits to 16 and add
Unsigned Extend and Add Halfword	UXTAH	Extend 16 bits to 32 and add
Unsigned Extend Byte	UXTB	Extend 8 bits to 32
Unsigned Extend Byte 16	UXTB16	Dual extend 8 bits to 16
Unsigned Extend Halfword	UXTH	Extend 16 bits to 32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.4.7 Parallel addition and subtraction instructions

These instructions perform additions and subtractions on the values of two registers and write the result to a destination register, treating the register values as sets of two halfwords or four bytes. That is, they perform SIMD additions or subtractions on the general-purpose registers.

These instructions consist of a prefix followed by a main instruction mnemonic. The prefixes are as follows:

`S` Signed arithmetic modulo 2^8 or 2^{16}.

`Q` Signed saturating arithmetic.

`SH` Signed arithmetic, halving the results.

`U` Unsigned arithmetic modulo 2^8 or 2^{16}.

`UQ` Unsigned saturating arithmetic.

`UH` Unsigned arithmetic, halving the results.

The main instruction mnemonics are as follows:

`ADD16` Adds the top halfwords of two operands to form the top halfword of the result, and the bottom halfwords of the same two operands to form the bottom halfword of the result.

`ASX` Exchanges halfwords of the second operand, and then adds top halfwords and subtracts bottom halfwords.

`SAX` Exchanges halfwords of the second operand, and then subtracts top halfwords and adds bottom halfwords.

`SUB16` Subtracts each halfword of the second operand from the corresponding halfword of the first operand to form the corresponding halfword of the result.

`ADD8` Adds each byte of the second operand to the corresponding byte of the first operand to form the corresponding byte of the result.

`SUB8` Subtracts each byte of the second operand from the corresponding byte of the first operand to form the corresponding byte of the result.

The instruction set permits all 36 combinations of prefix and main instruction operand, as Table F2-10 shows.

See also Advanced SIMD parallel addition and subtraction.

Table F2-10 Parallel addition and subtraction instructions

Main instruction	Signed	Saturating	Signed halving	Unsigned	Unsigned saturating	Unsigned halving
`ADD16`, add, two halfwords	`SADD16`	`QADD16`	`SHADD16`	`UADD16`	`UQADD16`	`UHADD16`
`ASX`, add and subtract with exchange	`SASX`	`QASX`	`SHASX`	`UASX`	`UQASX`	`UHASX`
`SAX`, subtract and add with exchange	`SSAX`	`QSAX`	`SHSAX`	`USAX`	`UQSAX`	`UHSAX`
`SUB16`, subtract, two halfwords	`SSUB16`	`QSUB16`	`SHSUB16`	`USUB16`	`UQSUB16`	`UHSUB16`
`ADD8`, add, four bytes	`SADD8`	`QADD8`	`SHADD8`	`UADD8`	`UQADD8`	`UHADD8`
`SUB8`, subtract, four bytes	`SSUB8`	`QSUB8`	`SHSUB8`	`USUB8`	`UQSUB8`	`UHSUB8`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.4.8 Divide instructions

Signed and unsigned integer divide instructions are included in both the T32 instruction set and the A32 instruction set.

For descriptions of the instructions see:

- SDIV.
- UDIV.

For the `SDIV` and `UDIV`

instructions, division by zero always returns a zero result.

The ID_ISAR0.Divide_instrs field indicates the level of support for these instructions. The field value of `0b0010` indicates they are implemented in both the T32 and A32 instruction sets.

F2.4.9 Miscellaneous data-processing instructions

Table F2-11 lists the miscellaneous data-processing instructions in the T32 and A32 instruction sets. Immediate values in these instructions are simple binary numbers.

Table F2-11 Miscellaneous data-processing instructions

Instruction	See	Notes
BitField Clear	BFC	-
BitField Insert	BFI	-
Count Leading Zeros	CLZ	-
Move Top	MOVT	Moves 16-bit immediate value to top halfword. Bottom halfword unchanged.
Reverse Bits	RBIT	-
Byte-Reverse Word	REV	-
Byte-Reverse Packed Halfword	REV16	-
Byte-Reverse Signed Halfword	REVSH	-
Signed BitField Extract	SBFX	-
Select Bytes using GE flags	SEL	-
Unsigned BitField Extract	UBFX	-
Unsigned Sum of Absolute Differences	USAD8	-
Unsigned Sum of Absolute Differences and Accumulate	USADA8	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.5 PSTATE and banked register access instructions

These instructions transfer PE state information to or from a general-purpose register.

F2.5.1 PSTATE access instructions

PSTATE holds process state information, see Process state, PSTATE. In AArch32 state:

- At EL1 or higher, PSTATE is accessible using the *Current Program Status Register* (CPSR).
- At EL0, a subset of the CPSR is accessible as the *Application Program Status Register* (APSR).
- On taking an exception, the contents of the CPSR are copied to the *Saved Program Status Register* (SPSR) of the mode from which the exception is taken.

The `MRS` and `MSR` instructions move the contents of the CPSR, APSR, or the SPSR of the current mode to or from a general-purpose register, see:

- MRS.
- MSR (immediate).
- MSR (register).

When executed at EL0, `MRS` and `MSR` instructions can only access the APSR.

The PSTATE Condition flags, PSTATE.{N, Z, C, V} are set by the execution of data-processing instructions, and can control the execution of conditional instructions. However, software can set the Condition flags explicitly using the `MSR` instruction, and can read the current state of the Condition flags explicitly using the `MRS` instruction.

In addition, at EL1 or higher, software can use the `CPS` instruction to change the PSTATE.M field and the PSTATE.{A, I, F} interrupt mask bits, see CPS, CPSID, CPSIE.

F2.5.2 Banked register access instructions

At EL1 or higher, the `MRS` (banked register) and `MSR` (banked register) instructions move the contents of a banked general-purpose register, the SPSR, or the ELR_hyp, to or from a general-purpose register. See:

- MRS (Banked register).
- MSR (Banked register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.6 Load/store instructions

Table F2-12 summarizes the general-purpose register load/store instructions in the T32 and A32 instruction sets. Some of these instructions can also operate on the PC. See also:

- Load/store multiple instructions.
- Synchronization and semaphores, for more information about the Load-Exclusive and Store-Exclusive instructions.
- Load-Acquire, Store-Release, for more information about the Load-Acquire/Store-Release and Load-Acquire Exclusive/Store-Release Exclusive instructions.
- Advanced SIMD and floating-point load/store instructions.

Load/store instructions have several options for addressing memory. For more information, see Addressing modes.

Table F2-12 Load/store instructions

	Load	Store	Unprivileged		Exclusive		Load-Acquire	Store-Release	Exclusive Load-Acquire	Store-Release
Data type			Load	Store	Load	Store				
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX	LDA	STL	LDAEX	STLEX
16-bit halfword	-	STRH	-	STRHT	-	STREXH	LDAH	STLH	LDAEXH	STLEXH
16-bit unsigned halfword	LDRH	-	LDRHT	–	LDREXH	-	-	-	-	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-	-	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB	LDAB	STLB	LDAEXB	STLEXB
8-bit unsigned byte	LDRB	–	LDRBT	-	LDREXB	-	-	-	-	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-	-	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD	-	-	LDAEXD	STLEXD

F2.6.1 Loads to the PC

The LDR instruction can load a value into the PC. The value loaded is treated as an interworking address, as described by the LoadWritePC() pseudocode function in Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

F2.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.

F2.6.3 Load unprivileged and Store unprivileged

When executing at EL0, a Load unprivileged or Store unprivileged instruction operates in the same way as the corresponding ordinary load or store instruction. For example, an LDRT instruction executes in exactly the same way as the equivalent LDR instruction. When executed at PL1, Load unprivileged and Store unprivileged instructions behave as they would if they were executed at EL0. For example, an LDRT instruction executes in exactly the way that the equivalent LDR instruction would execute at EL0. In particular, the instructions make unprivileged memory accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

As described in Security state, Exception levels, and AArch32 execution privilege, execution at PL1 describes all of the following:

- Execution at Non-secure EL1 using AArch32.
- Execution at Secure EL1 using AArch32 when EL3 is not implemented.
- Execution at Secure EL1 using AArch32 when EL3 is implemented and is using AArch64.
- Execution at Secure EL3 when EL3 is implemented and is using AArch32.

The Load unprivileged and Store unprivileged instructions are CONSTRAINED UNPREDICTABLE if executed at EL2.

For more information about execution privilege, see About access permissions.

F2.6.4 Load-Exclusive and Store-Exclusive

Load-Exclusive and Store-Exclusive instructions provide shared memory synchronization. For more information, see Synchronization and semaphores.

F2.6.5 Load-Acquire and Store-Release

Load-Acquire and Store-Release instructions provide memory barriers. Load-Acquire Exclusive and Store-Release Exclusive instructions provide memory barriers with shared memory synchronization. For more information, see Load-Acquire, Store-Release.

F2.6.6 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers R0-R12, SP, or LR.

For loads, the base register can be the PC. This provides PC-relative addressing for position-independent code. Instructions marked (literal) in their title in About the T32 and A32 Instruction Descriptions are PC-relative loads.

The offset takes one of three formats:

Immediate The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets, and input/output registers.

Register The offset is a value from a general-purpose register. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

Scaled register The offset is a general-purpose register, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element.

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

Offset The offset is added to or subtracted from the base register to form the memory address.

Pre-indexed The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

Post-indexed The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. The result is stored back in the base register, to permit automatic indexing through an array or memory block.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See About the T32 and A32 Instruction Descriptions for full details for each instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.7 Load/store multiple instructions

Load Multiple instructions load from memory a subset, or possibly all, of the general-purpose registers and the PC.

Store Multiple instructions store to memory a subset, or possibly all, of the general-purpose registers.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

Table F2-13 summarizes the load/store multiple instructions in the T32 and A32 instruction sets.

Table F2-13 Load/store multiple instructions

Instruction	See
Load Multiple, Increment After or Full Descending	LDM, LDMIA, LDMFD
Load Multiple, Decrement After or Full Ascending [a]	LDMDA, LDMFA
Load Multiple, Decrement Before or Empty Ascending	LDMDB, LDMEA
Load Multiple, Increment Before or Empty Descending [a]	LDMIB, LDMED
Pop multiple registers off the stack[b]	POP
Push multiple registers onto the stack[c]	PUSH
Store Multiple, Increment After or Empty Ascending	STM, STMIA, STMEA
Store Multiple, Decrement After or Empty Descending [a]	STMDA, STMED
Store Multiple, Decrement Before or Full Descending	STMDB, STMFD
Store Multiple, Increment Before or Full Ascending [a]	STMIB, STMFA

a. Not available in the T32 instruction set.

b. This instruction is equivalent to an `LDM` instruction with the SP as base register, and base register updating.

c. This instruction is equivalent to an `STMDB` instruction with the SP as base register, and base register updating.

When executing at EL1, variants of the `LDM` and `STM` instructions load and store User mode registers. Another system level variant of the `LDM` instruction performs an exception return.

F2.7.1 Loads to the PC

The `LDM`, `LDMDA`, `LDMDB`, `LDMIB`, and `POP` instructions can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.8 Miscellaneous instructions

Table F2-14 summarizes the miscellaneous instructions in the T32 and A32 instruction sets.

Table F2-14 Miscellaneous instructions

Instruction	See
Clear Branch History	CLRBHB
Clear-Exclusive	CLREX
Data Memory Barrier	DMB
Data Synchronization Barrier	DSB
Error Synchronization Barrier	ESB
Instruction Synchronization Barrier	ISB
If-Then	IT
No Operation	NOP
Preload Data	PLD, PLDW (immediate) PLD (literal) PLD, PLDW (register)
Preload Instruction	PLI (immediate, literal) PLI (register)
Speculation Barrier	SB
Set Endianness	SETEND[a]
Set Privileged Access Never	SETPAN
Send Event	SEV
Send Event Local	SEVL
Wait For Event	WFE
Wait For Interrupt	WFI
Yield	YIELD[b]

a. Arm deprecates any use of the `SETEND` instruction.

b. See also The Yield instruction.

Note

Previous versions of the architecture defined the `DBG` instruction, that could provide a hint to the debug system, in this group. From the introduction of Armv8, this instruction executes as a `NOP`. Arm deprecates any use of the `DBG` instruction.

F2.8.1 The Yield instruction

In a *Symmetric Multithreading* (SMT) design, a thread can use the `YIELD` instruction to give a hint to the PE that it is running on. The `YIELD` hint indicates that whatever the thread is currently doing is of low importance, and so could yield. For example, the thread might be sitting in a spin-lock. A similar use might be in modifying the arbitration priority of the snoop bus in a multiprocessor (MP) system. Defining such an instruction permits binary compatibility between SMT and SMP systems.

AArch32 state defines a `YIELD` instruction as a specific NOP (No Operation) hint instruction.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use on migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a `NOP` if there is no implementation benefit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.9 Exception-generating and exception-handling instructions

The following instructions are intended specifically to cause a synchronous exception to occur:

- The `SVC` instruction generates a Supervisor Call exception. For more information, see Supervisor Call (SVC) exception.
- The Breakpoint instruction `BKPT` provides software breakpoints. For more information, see Breakpoint Instruction exceptions.
- In an implementation that includes EL3 the `SMC` instruction generates a Secure Monitor Call exception. For more information, see Secure Monitor Call (SMC) exception.
- In an implementation that includes EL2 the `HVC` instruction generates a Hypervisor Call exception. For more information, see Hypervisor Call (HVC) exception.

Debug state summarizes the Debug state instructions.

For an exception taken to an EL1 mode:

- The system level variants of the `SUBS` and `LDM` instructions can perform a return from an exception.

  Note

  The variants of `SUBS` include `MOVS`. See the references to Subtract (exception return), Move (exception return), and Load Multiple (exception return) in Table F2-15 for more information.

- The `SRS` instruction can be used near the start of the handler, to store return information. The `RFE` instruction can then perform a return from the exception using the stored return information.

In an implementation that includes EL2, the `ERET` instruction performs a return from an exception taken to Hyp mode.

For more information, see Exception return to an Exception level using AArch32.

Table F2-15 summarizes the instructions, in the T32 and A32 instruction sets, for generating or handling an exception. Except for `BKPT` and `SVC`, these are system level instructions.

Table F2-15 Exception-generating and exception-handling instructions

Instruction	See
Supervisor Call	SVC
Breakpoint	BKPT
Secure Monitor Call	SMC
Return From Exception	RFE, RFEDA, RFEDB, RFEIA, RFEIB
Subtract (exception return)[a]	SUB, SUBS (immediate)[a]
Move (exception return)[a]	MOV, MOVS (register)[a]
Hypervisor Call	HVC
Exception Return	ERET
Load Multiple (exception return)	LDM (exception return)
Store Return State	SRS, SRSDA, SRSDB, SRSIA, SRSIB

a. The A32 instruction set includes other instruction forms that can be used for an exception return, that have previously been described as variants of `SUBS PC, LR`. Arm deprecates any use of these instruction forms.

F2.9.1 Debug state

Table F2-16 shows the Debug state instructions that are implemented in the T32 instruction set:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F2-16 T32 Debug state instructions

Mnemonic	Instruction	See	Note
`DCPSn`	Debug switch to EL*n*	DCPS1 DCPS2 DCPS3	-
`ERET`	Debug restore PE state (DRPS)	ERET	When executed in Debug state, the T1 encoding of `ERET` performs the DRPS operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.10 System register access instructions

The System register encoding space is indexed using the parameters {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}, see The AArch32 System register interface. This encoding space provides System registers and System instructions. The only permitted values of `coproc` are `0b1110` and `0b1111`, and the following instructions give access to this encoding space:

- Instructions that transfer data between general-purpose registers and System registers. See:
  - MCR.
  - MCRR.
  - MRC.
  - MRRC.
- Instructions that load or store from memory to a System register. See:
  - LDC (immediate).
  - LDC (literal).
  - STC.

Note

The System register encoding space with `coproc`== `0b101x` is redefined to provide some of the Advanced SIMD and floating-point functionality. That is, to:

- Initiate a floating-point data-processing operation, see Floating-point data-processing instructions.
- Transfer data between general-purpose registers and the Advanced SIMD and floating-point registers, see Advanced SIMD and floating-point register transfer instructions.
- Load or store data to the Advanced SIMD and floating-point registers, see Advanced SIMD and floating-point load/store instructions.

System register access instructions are part of the instruction stream executed by the PE, and therefore any System register access instruction that cannot be executed by the implementation causes an Undefined Instruction exception. In the A-profile and the R-profile architectures, the instruction encodings in the System register access instruction encoding space are unallocated, and generate Undefined Instruction exceptions, except for:

- The instructions summarized in this section that access the `coproc` == `0b111x` encoding space.
- The instructions in the `coproc` == `0b101x` encoding space that are redefined to provide Advanced SIMD and floating-point functionality, as summarized in the Note in this section.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.11 Advanced SIMD and floating-point load/store instructions

Table F2-17 summarizes the SIMD and floating-point register file load/store instructions in the Advanced SIMD and floating-point instruction sets.

Advanced SIMD also provides instructions for loading and storing multiple elements, or structures of elements, see Element and structure load/store instructions.

Table F2-17 SIMD and floating-point register file load/store instructions

Instruction	See	Operation
Vector Load Multiple	VLDM, VLDMDB, VLDMIA	Load 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point. Load 1-32 consecutive 32-bit registers, floating-point only.
Vector Load Register	VLDR (immediate) VLDR (literal)	Load one 64-bit register, Advanced SIMD and floating-point. Load one 32-bit register, floating-point only.
Vector Store Multiple	VSTM, VSTMDB, VSTMIA	Store 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point. Store 1-32 consecutive 32-bit registers, floating-point only.
Vector Store Register	VSTR	Store one 64-bit register, Advanced SIMD and floating-point. Store one 32-bit register, floating-point only.

F2.11.1 Element and structure load/store instructions

Table F2-18 shows the element and structure load/store instructions available in the Advanced SIMD instruction set. Loading and storing structures of more than one element automatically de-interleaves or interleaves the elements, see Figure F2-1 for an example of de-interleaving. Interleaving is the inverse process.

Table F2-18 Element and structure load/store instructions

Instruction	See
Load single element	
Multiple elements	VLD1 (multiple single elements)
To one lane	VLD1 (single element to one lane)
To all lanes	VLD1 (single element to all lanes)
Load 2-element structure	
Multiple structures	VLD2 (multiple 2-element structures)
To one lane	VLD2 (single 2-element structure to one lane)
To all lanes	VLD2 (single 2-element structure to all lanes)
Load 3-element structure	
Multiple structures	VLD3 (multiple 3-element structures)
To one lane	VLD3 (single 3-element structure to one lane)
To all lanes	VLD3 (single 3-element structure to all lanes)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See
Load 4-element structure	
Multiple structures	VLD4 (multiple 4-element structures)
To one lane	VLD4 (single 4-element structure to one lane)
To all lanes	VLD4 (single 4-element structure to all lanes)
Store single element	
Multiple elements	VST1 (multiple single elements)
From one lane	VST1 (single element from one lane)
Store 2-element structure	
Multiple structures	VST2 (multiple 2-element structures)
From one lane	VST2 (single 2-element structure from one lane)
Store 3-element structure	
Multiple structures	VST3 (multiple 3-element structures)
From one lane	VST3 (single 3-element structure from one lane)
Store 4-element structure	
Multiple structures	VST4 (multiple 4-element structures)
From one lane	VST4 (single 4-element structure from one lane)

Figure F2-1 shows the de-interleaving of a `VLD3.16` (multiple 3-element structures) instruction:

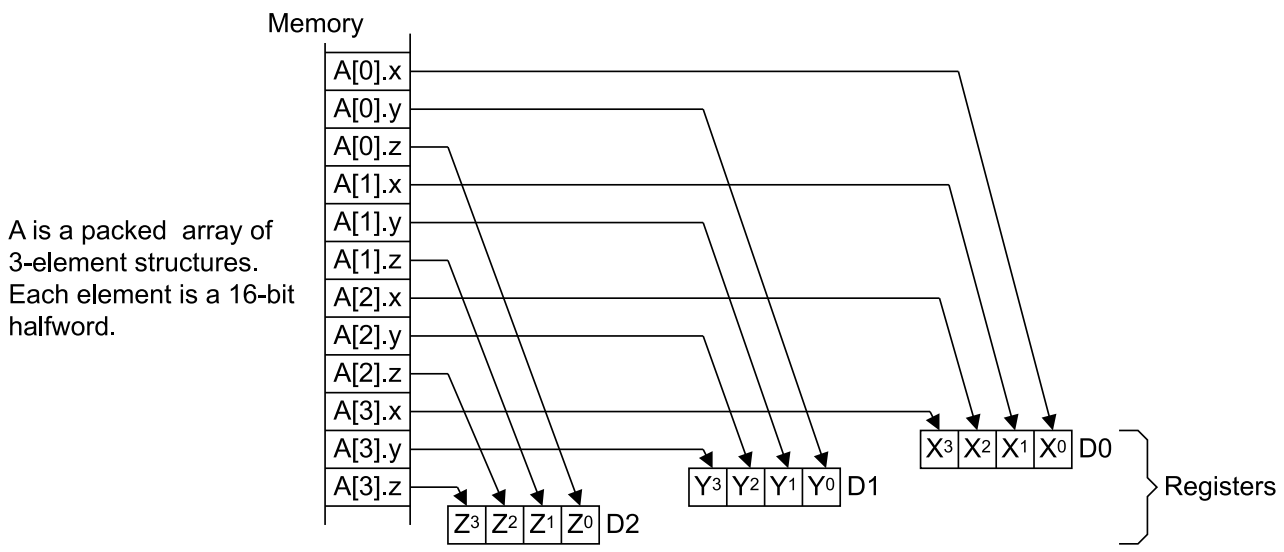

Figure F2-1 De-interleaving an array of 3-element structures

Figure F2-1 shows the `VLD3.16` instruction operating to three 64-bit registers that comprise four 16-bit elements:

- Different instructions in this group would produce similar figures, but operate on different numbers of registers. For example, `VLD4` and `VST4` instructions operate on four registers.
- Different element sizes would produce similar figures but with 8-bit or 32-bit elements.
- These instructions operate only on doubleword (64-bit) registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.12 Advanced SIMD and floating-point register transfer instructions

Table F2-19 summarizes the SIMD and floating-point register file transfer instructions in the Advanced SIMD and floating-point instruction sets. These instructions transfer data between the general-purpose registers and the registers in the SIMD and floating-point register file.

Advanced SIMD vectors, and single-precision and double-precision floating-point registers, are all views of the same register file. For details, see The SIMD and floating-point register file.

Table F2-19 SIMD and floating-point register file transfer instructions

Instruction	See
Copy element from general-purpose register to every element of an Advanced SIMD vector	VDUP (general-purpose register)
Copy byte, halfword, or word from general-purpose register to a register in the SIMD and floating-point register file	VMOV (general-purpose register to scalar)
Copy byte, halfword, or word from a register in the SIMD and floating-point register file to a general-purpose register	VMOV (scalar to general-purpose register)
Copy from half-precision floating-point register to general-purpose register, or from general-purpose register to half-precision floating-point register Only supported if FEAT_FP16 is implemented	VMOV (between general-purpose register and half-precision)
Copy from single-precision floating-point register to general-purpose register, or from general-purpose register to single-precision floating-point register	VMOV (between general-purpose register and single-precision)
Copy two words from general-purpose registers to consecutive single-precision floating-point registers, or from consecutive single-precision floating-point registers to general-purpose registers	VMOV (between two general-purpose registers and two single-precision registers)
Copy two words from general-purpose registers to a doubleword register in the SIMD and floating-point register file, or from a doubleword register in the SIMD and floating-point register file to general-purpose registers	VMOV (between two general-purpose registers and a doubleword floating-point register)
Copy from an Advanced SIMD and floating-point System Register to a general-purpose register	VMRS
Copy from a general-purpose register to an Advanced SIMD and floating-point System Register	VMSR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.13 Advanced SIMD data-processing instructions

Advanced SIMD data-processing instructions process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel.

Instructions operate on vectors held in 64-bit or 128-bit registers. Figure F2-2 shows an operation on two 64-bit operand vectors, generating a 64-bit vector result.

Note

Figure F2-2 and other similar figures show 64-bit vectors that consist of four 16-bit elements, and 128-bit vectors that consist of four 32-bit elements. Other element sizes produce similar figures, but with 1, 2, 8, or 16 operations performed in parallel instead of 4.

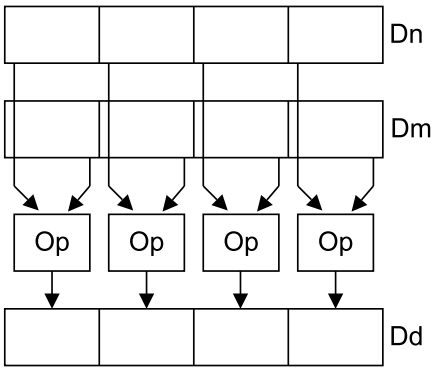

Figure F2-2 Advanced SIMD instruction operating on 64-bit registers

Many Advanced SIMD instructions have variants that produce vectors of elements double the size of the inputs. In this case, the number of elements in the result vector is the same as the number of elements in the operand vectors, but each element, and the whole vector, is double the size.

Figure F2-3 shows an example of an Advanced SIMD instruction operating on 64-bit registers, and generating a 128-bit result.

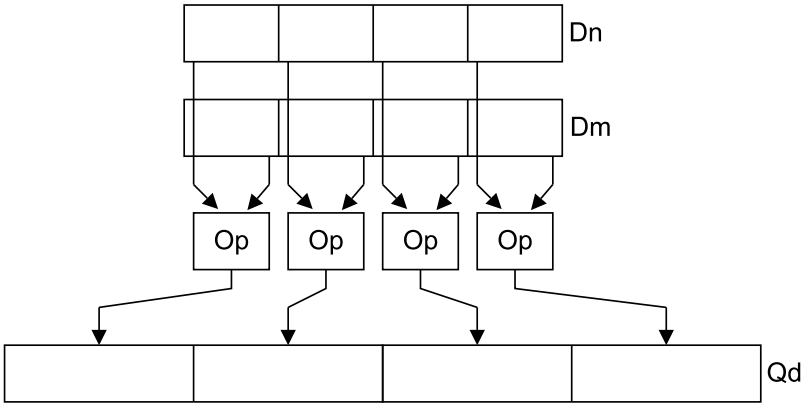

Figure F2-3 Advanced SIMD instruction producing wider result

There are also Advanced SIMD instructions that have variants that produce vectors containing elements half the size of the inputs. Figure F2-4 shows an example of an Advanced SIMD instruction operating on one 128-bit register, and generating a 64-bit result.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

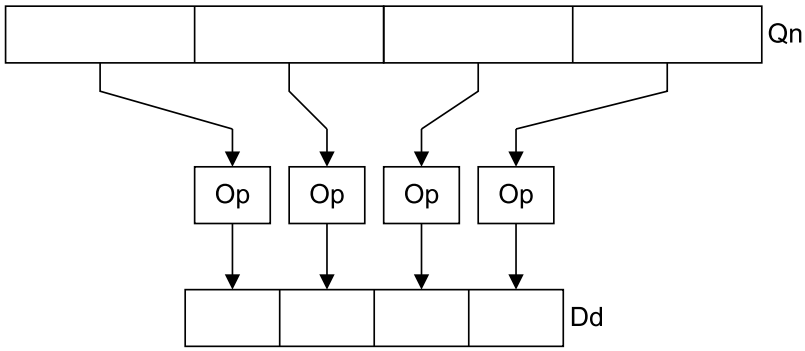

Figure F2-4 Advanced SIMD instruction producing narrower result

Some Advanced SIMD instructions do not conform to these standard patterns. Their operation patterns are described in the individual instruction descriptions.

Advanced SIMD instructions that perform floating-point arithmetic use the Arm standard floating-point arithmetic defined in Floating-point support.

The following sections summarize the Advanced SIMD data-processing instructions:

- Advanced SIMD parallel addition and subtraction.
- Bitwise Advanced SIMD data-processing instructions.
- Advanced SIMD comparison instructions.
- Advanced SIMD shift instructions.
- Advanced SIMD multiply instructions.
- Advanced SIMD dot product instructions.
- Miscellaneous Advanced SIMD data-processing instructions.
- Advanced SIMD BFloat16 instructions.
- Advanced SIMD matrix multiply instructions.
- The Cryptographic Extension in AArch32 state.

F2.13.1 Advanced SIMD parallel addition and subtraction

Table F2-20 shows the Advanced SIMD parallel add and subtract instructions.

Table F2-20 Advanced SIMD parallel add and subtract instructions

Instruction	See
Vector Add	VADD (integer) VADD (floating-point)
Vector Add and Narrow, returning High Half	VADDHN
Vector Add Long	VADDL
Vector Add Wide	VADDW
Vector Halving Add	VHADD
Vector Halving Subtract	VHSUB
Vector Pairwise Add and Accumulate Long	VPADAL
Vector Pairwise Add	VPADD (integer) VPADD (floating-point)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See
Vector Pairwise Add Long	VPADDL
Vector Rounding Add and Narrow, returning High Half	VRADDHN
Vector Rounding Halving Add	VRHADD
Vector Rounding Subtract and Narrow, returning High Half	VRSUBHN
Vector Saturating Add	VQADD
Vector Saturating Subtract	VQSUB
Vector Subtract	VSUB (integer) VSUB (floating-point)
Vector Subtract and Narrow, returning High Half	VSUBHN
Vector Subtract Long	VSUBL
Vector Subtract Wide	VSUBW

F2.13.2 Bitwise Advanced SIMD data-processing instructions

Table F2-21 shows bitwise Advanced SIMD data-processing instructions. These operate on the doubleword (64-bit) or quadword (128-bit) registers in the SIMD and floating-point register file, and there is no division into vector elements.

Table F2-21 Bitwise Advanced SIMD data-processing instructions

Instruction	See
Vector Bitwise AND	VAND (register)
Vector Bitwise Bit Clear (AND complement)	VBIC (immediate) VBIC (register)
Vector Bitwise Exclusive OR	VEOR
Vector Bitwise Insert if False	VBIF
Vector Bitwise Insert if True	VBIT
Vector Bitwise Move	VMOV (immediate) VMOV (register)
Vector Bitwise NOT	VMVN (immediate) VMVN (register)
Vector Bitwise OR	VORR (immediate) VORR (register)
Vector Bitwise OR NOT	VORN (register)
Vector Bitwise Select	VBSL

F2.13.3 Advanced SIMD comparison instructions

Table F2-22 shows Advanced SIMD comparison instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F2-22 Advanced SIMD comparison instructions

Instruction	See
Vector Absolute Compare Greater Than or Equal	VACGE
Vector Absolute Compare Greater Than	VACGT
Vector Compare Equal	VCEQ (register)
Vector Compare Equal to Zero	VCEQ (immediate #0)
Vector Compare Greater Than or Equal	VCGE (register)
Vector Compare Greater Than or Equal to Zero	VCGE (immediate #0)
Vector Compare Greater Than	VCGT (register)
Vector Compare Greater Than Zero	VCGT (immediate #0)
Vector Compare Less Than or Equal to Zero	VCLE (immediate #0)
Vector Compare Less Than Zero	VCLT (immediate #0)
Vector Test Bits	VTST

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.13.4 Advanced SIMD shift instructions

Table F2-23 lists the shift instructions in the Advanced SIMD instruction set.

Table F2-23 Advanced SIMD shift instructions

Instruction	See
Vector Saturating Rounding Shift Left	VQRSHL
Vector Saturating Rounding Shift Right and Narrow	VQRSHRN, VQRSHRUN
Vector Saturating Shift Left	VQSHL (register) VQSHL, VQSHLU (immediate)
Vector Saturating Shift Right and Narrow	VQSHRN, VQSHRUN
Vector Rounding Shift Left	VRSHL
Vector Rounding Shift Right	VRSHR
Vector Rounding Shift Right and Accumulate	VRSRA
Vector Rounding Shift Right and Narrow	VRSHRN
Vector Shift Left	VSHL (immediate) VSHL (register)
Vector Shift Left Long	VSHLL
Vector Shift Right	VSHR
Vector Shift Right and Narrow	VSHRN
Vector Shift Left and Insert	VSLI
Vector Shift Right and Accumulate	VSRA
Vector Shift Right and Insert	VSRI

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.13.5 Advanced SIMD multiply instructions

Table F2-24 shows the Advanced SIMD multiply instructions.

Table F2-24 Advanced SIMD multiply instructions

Instruction	See
Vector Multiply Accumulate	VMLA (integer) VMLA (floating-point) VMLA (by scalar)
Vector Multiply Accumulate Long	VMLAL (integer) VMLAL (by scalar)
Vector Multiply Subtract	VMLS (integer) VMLS (floating-point) VMLS (by scalar)
Vector Multiply Subtract Long	VMLSL (integer) VMLSL (by scalar)
Vector Multiply	VMUL (integer and polynomial) VMUL (floating-point) VMUL (by scalar)
Vector Multiply Long	VMULL (integer and polynomial) VMULL (by scalar)
Vector Fused Multiply Accumulate	VFMA
Vector Floating-Point Multiply-Add Long	VFMAL (vector) VFMAL (by scalar)
Vector Fused Multiply Subtract	VFMS
Vector Floating-Point Multiply-Subtract Long	VFMSL (vector) VFMSL (by scalar)
Vector Saturating Doubling Multiply Accumulate Long	VQDMLAL
Vector Saturating Doubling Multiply Subtract Long	VQDMLSL
Vector Saturating Doubling Multiply Returning High Half	VQDMULH
Vector Saturating Doubling Multiply Long	VQDMULL
Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half	VQRDMLAH
Vector Saturating Rounding Doubling Multiply Subtract Returning High Half	VQRDMLSH
Vector Saturating Rounding Doubling Multiply Returning High Half	VQRDMULH

Advanced SIMD multiply instructions can operate on vectors of:

- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit polynomials over {0, 1}. `VMUL` and `VMULL` are the only instructions that operate on polynomials. `VMULL` produces a 16-bit polynomial over {0, 1}.
- Single-precision (32-bit) or half-precision (16-bit) floating-point numbers.

They can also act on one vector and one scalar.

Long instructions have doubleword (64-bit) operands, and produce quadword (128-bit) results. Other Advanced SIMD multiply instructions can have either doubleword or quadword operands, and produce results of the same size.

Floating-point multiply instructions can operate on:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Half-precision (16-bit) floating-point numbers.
- Single-precision (32-bit) floating-point numbers.
- Double-precision (64-bit) floating-point numbers.

F2.13.6 Advanced SIMD dot product instructions

FEAT_DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second vector.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note

That is, a single element from the second vector is used, and the dot product is calculated between each element of the first vector and this single element from the second vector.

Table F2-25 shows the Advanced SIMD dot product instructions.

Table F2-25 Advanced SIMD dot product instructions

Mnemonic	Instruction	See
VSDOT	Signed dot product (vector form)	VSDOT (vector)
VUDOT	Unsigned dot product (vector form)	VUDOT (vector)
VSDOT	Signed dot product (indexed form)	VSDOT (by element)
VSUDOT	Mixed sign integer dot product by indexed quadruplet[a]	VSUDOT (by element)
VUDOT	Unsigned dot product (indexed form)	VUDOT (by element)
VUSDOT	Mixed sign integer dot product (vector format)[a]	VUSDOT (vector)
	Mixed sign integer dot product by indexed quadruplet[a]	VUSDOT (by element)

a. This instruction is only supported when FEAT_AA32I8MM is implemented.

F2.13.7 AArch32 Advanced SIMD complex number arithmetic instructions

FEAT_FCMA provides AArch32 Advanced SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide single-precision versions. If FEAT_FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Table F2-26 shows the FEAT_FCMA AArch32 Advanced SIMD instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F2-26 Advanced SIMD complex number arithmetic instructions

Mnemonic	Instruction	See
VCADD	Floating-point complex add	VCADD
VCMLA	Floating-point complex multiply accumulate (vector form)	VCMLA
VCMLA	Floating-point complex multiply accumulate (indexed form)	VCMLA (by element)

A pair of VCMLA instructions can be used to perform a complex number multiplication. In Complex multiplication, this is demonstrated for the similar AArch64 instruction FCMLA. The usage of VCMLA in this manner is identical.

F2.13.8 Advanced SIMD BFloat16 instructions

When FEAT_AA32BF16 is implemented, BFloat16 instructions are available in AArch32 state.

Table F2-27 shows the Advanced SIMD BFloat16 instructions.

Table F2-27 BFloat16 Advanced SIMD instructions

Mnemonic	Instruction	See
VDOT	BFloat16 floating-point vector dot product (vector and by scalar formats)	VDOT (vector) VDOT (by element)
VMMLA	BFloat16 floating-point matrix multiply-accumulate	VMMLA
VFMAB, VFMAT	BFloat16 floating-point widening multiply-add long (vector and by scalar formats)	VFMAB, VFMAT (BFloat16, vector) VFMAB, VFMAT (BFloat16, by scalar)
VCVT	BFloat16 convert from single-precision to BFloat16 format	VCVT (from single-precision to BFloat16, Advanced SIMD)

F2.13.9 Advanced SIMD matrix multiply instructions

When FEAT_AA32I8MM is implemented, these instructions are available in AArch32 state. They include integer and mixed sign dot product instructions.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector matrix is organized in row-by-row order.
- The second source vector matrix elements are organized in a column-by-column order.
- The destination vector matrix is organized in row-by-row order.

One matrix multiplication is performed per segment.

Table F2-28 shows the Advanced SIMD matrix multiply instructions.

Table F2-28 Matrix multiply Advanced SIMD instructions

Mnemonic	Instruction	See
VSMMLA	Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix	VSMMLA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
VUMMLA	Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix	VUMMLA
VUSMMLA	Widening 8-bit mixed sign integer matrix multiply-accumulate into 2x2 matrix	VUSMMLA

F2.13.10 Miscellaneous Advanced SIMD data-processing instructions

Table F2-29 shows miscellaneous Advanced SIMD data-processing instructions.

Table F2-29 Miscellaneous Advanced SIMD data-processing instructions

Instruction	See
Vector Absolute Difference and Accumulate	VABA
Vector Absolute Difference and Accumulate Long	VABAL
Vector Absolute Difference	VABD (integer) VABD (floating-point)
Vector Absolute Difference Long	VABDL (integer)
Vector Absolute	VABS
Vector Convert between floating-point and fixed-point	VCVT (between floating-point and fixed-point, Advanced SIMD)
Vector Convert between floating-point and integer	VCVT (between floating-point and integer, Advanced SIMD)
Vector Convert between half-precision and single-precision	VCVT (between half-precision and single-precision, Advanced SIMD)
Vector Count Leading Sign Bits	VCLS
Vector Count Leading Zeros	VCLZ
Vector Count Set Bits	VCNT
Vector Duplicate scalar	VDUP (scalar)
Vector Extract	VEXT (byte elements)
Vector move Insertion	VINS
Vector Move and Narrow	VMOVN
Vector Move Long	VMOVL
Vector Move extraction	VMOVX
Vector Maximum	VMAX (integer) VMAX (floating-point)
Vector Minimum	VMIN (integer) VMIN (floating-point)
Vector Negate	VNEG
Vector Pairwise Maximum	VPMAX (integer) VPMAX (floating-point)
Vector Pairwise Minimum	VPMIN (integer) VPMIN (floating-point)
Vector Reciprocal Estimate	VRECPE
Vector Reciprocal Step	VRECPS
Vector Reciprocal Square Root Estimate	VRSQRTE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See
Vector Reciprocal Square Root Step	VRSQRTS
Vector Reverse in halfwords	VREV16
Vector Reverse in words	VREV32
Vector Reverse in doublewords	VREV64
Vector Saturating Absolute	VQABS
Vector Saturating Move and Narrow	VQMOVN, VQMOVUN
Vector Saturating Negate	VQNEG
Vector Swap	VSWP
Vector Table Lookup	VTBL, VTBX
Vector Transpose	VTRN
Vector Unzip	VUZP
Vector Zip	VZIP

F2.13.11 The Cryptographic Extension in AArch32 state

The instructions provided by the optional Cryptographic Extension use the Advanced SIMD and floating-point register file. For more information about the functions they provide see:

- *Advanced Encryption Standard (AES)*.
- *The Galois/Counter Mode of Operation*.
- *Secure Hash Algorithm (SHA)*.

Table F2-30 shows the AArch32 Cryptographic Extension instructions.

Table F2-30 AArch32 Cryptographic Extension instructions

Mnemonic	Instruction	See
`AESD`	AES single round decryption	AESD
`AESE`	AES single round encryption	AESE
`AESIMC`	AES inverse mix columns	AESIMC
`AESMC`	AES mix columns	AESMC
`VMULL`	Polynomial multiply long	VMULL (integer and polynomial) [a]
`SHA1C`	SHA1 hash update (choose)	SHA1C
`SHA1H`	SHA1 fixed rotate	SHA1H
`SHA1M`	SHA1 hash update (majority)	SHA1M
`SHA1P`	SHA1 hash update (parity)	SHA1P
`SHA1SU0`	SHA1 schedule update 0	SHA1SU0
`SHA1SU1`	SHA1 schedule update 1	SHA1SU1
`SHA256H`	SHA256 hash update (part 1)	SHA256H
`SHA256H2`	SHA256 hash update (part 2)	SHA256H2
`SHA256SU0`	SHA256 schedule update 0	SHA256SU0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SHA256SU1	SHA256 schedule update 1	SHA256SU1

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.14 Floating-point data-processing instructions

Table F2-31 summarizes the data-processing instructions in the floating-point instruction set. In this table, *floating-point register* means a register in the SIMD and floating-point register file. The BFloat16 floating-point instructions are provided by FEAT_AA32BF16.

For details of the floating-point arithmetic used by floating-point instructions, see Floating-point support.

Table F2-31 Floating-point data-processing instructions

Instruction	See
BFloat16 convert from single-precision to BFloat16 format writing to bottom half of single-precision register	VCVTB (BFloat16)
BFloat16 convert from single-precision to BFloat16 format writing to top half of single-precision register	VCVTT (BFloat16)
Convert between double-precision and single-precision	VCVT (between double-precision and single-precision)
Convert between floating-point and fixed-point	VCVT (between floating-point and fixed-point, floating-point)
Convert between half-precision and single-precision, writing to bottom half of single-precision register	VCVTB
Convert between half-precision and single-precision, writing to top half of single-precision register	VCVTT
Convert from floating-point to integer	VCVT (floating-point to integer, floating-point)
Convert from floating-point to integer using FPSCR rounding mode	VCVTR
Convert from integer to floating-point	VCVT (integer to floating-point, floating-point)
Floating-point JavaScript convert to signed fixed-point, rounding toward zero	VJCVT
Copy from one floating-point register to another	VMOV (register)
Divide	VDIV
Move immediate value to a floating-point register	VMOV (immediate)
Square Root	VSQRT
Vector Absolute value	VABS
Vector Add	VADD (floating-point)
Vector Compare with exceptions disabled	VCMPE
Vector Compare with exceptions enabled	VCMP
Vector Fused Multiply Accumulate	VFMA
Vector Fused Multiply Subtract	VFMS
Vector Fused Negate Multiply Accumulate	VFNMA
Vector Fused Negate Multiply Subtract	VFNMS
Vector Multiply	VMUL (floating-point)
Vector Multiply Accumulate	VMLA (floating-point)
Vector Multiply Subtract	VMLS (floating-point)
Vector Negate Multiply	VNMUL
Vector Negate Multiply Accumulate	VNMLA
Vector Negate Multiply Subtract	VNMLS

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	See
Vector Negate, by inverting the sign bit	VNEG
Vector Subtract	VSUB (floating-point)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F3
T32 Instruction Set Encoding

This chapter describes the encoding of the T32 instruction set. It contains the following sections:

- T32 instruction set encoding.
- About the T32 Advanced SIMD and floating-point instructions and their encoding.

In this chapter:

- In the decode tables, an entry of `x` for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1 T32 instruction set encoding

The T32 instruction stream is a sequence of halfword-aligned halfwords. Each T32 instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If the value of bits[15:11] of the halfword being decoded is one of the following, the halfword is the first halfword of a 32-bit instruction:

- `0b11101`.
- `0b11110`.
- `0b11111`.

Otherwise, the halfword is a 16-bit instruction.

The T32 instruction encoding is:

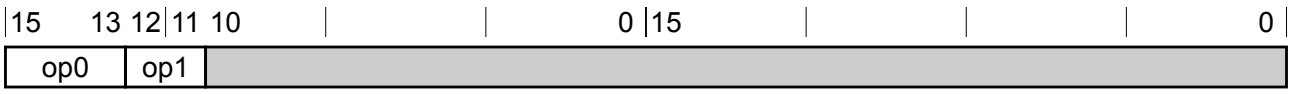

Table F3-1 Main encoding table for the T32 instruction set

Decode fields		Decode group or instruction page
op0	**op1**	
`!= 111`	`xx`	16-bit
`111`	`00`	B - T2 variant
`111`	`!= 00`	32-bit

F3.1.1 16-bit

This section describes the encoding of the 16-bit group. The encodings in this section are decoded from T32 instruction set encoding.

This decode also imposes the constraint:

- `op0<5:3> != 111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-2 Encoding table for the 16-bit group

Decode fields op0	Decode group or instruction page
00xxxx	Shift (immediate), add, subtract, move, and compare
010000	Data-processing (two low registers)
010001	Special data instructions and branch and exchange
01001x	LDR (literal) - T1 variant
0101xx	Load/store (register offset)
011xxx	Load/store word/byte (immediate offset)
1000xx	Load/store halfword (immediate offset)
1001xx	Load/store (SP-relative)
1010xx	Add PC/SP (immediate)
1011xx	Miscellaneous 16-bit instructions
1100xx	Load/store multiple
1101xx	Conditional branch, and Supervisor Call

F3.1.1.1 Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers) instruction class. The encodings in this section are decoded from 16-bit.

15	14	13	12	11	10	9 … 6	5 … 3	2 … 0
0	1	0	0	0	0	op	Rs	Rd

Table F3-3

Decode fields op	Instruction page
0000	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) - Logical shift left variant
0011	MOV, MOVS (register-shifted register) - Logical shift right variant
0100	MOV, MOVS (register-shifted register) - Arithmetic shift right variant
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) - Rotate right variant
1000	TST (register)
1001	RSB, RSBS (immediate)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields	Instruction page
op	
1010	CMP (register)
1011	CMN (register)
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

F3.1.1.2 Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. The encodings in this section are decoded from 16-bit.

15 14 13 12	11	10	9	8 6	5 3	2 0
0 1 0 1	L	B	H	Rm	Rn	Rt

Table F3-4

Decode fields			Instruction page
L	B	H	
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

F3.1.1.3 Load/store word/byte (immediate offset)

This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. The encodings in this section are decoded from 16-bit.

15 14 13 12	11	10 6	5 3	2 0
0 1 1 B	L	imm5	Rn	Rt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-5

Decode fields		Instruction page
B	**L**	
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

F3.1.1.4 Load/store halfword (immediate offset)

This section describes the encoding of the Load/store halfword (immediate offset) instruction class. The encodings in this section are decoded from 16-bit.

15 14 13 12	11	10 … 6	5 … 3	2 … 0
1 0 0 0	L	imm5	Rn	Rt

Table F3-6

Decode fields	Instruction page
L	
0	STRH (immediate)
1	LDRH (immediate)

F3.1.1.5 Load/store (SP-relative)

This section describes the encoding of the Load/store (SP-relative) instruction class. The encodings in this section are decoded from 16-bit.

15 14 13 12	11	10 … 8	7 … 0
1 0 0 1	L	Rt	imm8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-7

Decode fields L	Instruction page
0	STR (immediate)
1	LDR (immediate)

F3.1.1.6 Add PC/SP (immediate)

This section describes the encoding of the Add PC/SP (immediate) instruction class. The encodings in this section are decoded from 16-bit.

15 14 13 12	11	10 8	7 0
1 0 1 0	SP	Rd	imm8

Table F3-8

Decode fields SP	Instruction page
0	ADR
1	ADD, ADDS (SP plus immediate)

F3.1.1.7 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from 16-bit.

15 14 13 12	11	10 8	7 0
1 1 0 0	L	Rn	register_list

Table F3-9

Decode fields L	Instruction page
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1.2 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. The encodings in this section are decoded from 16-bit.

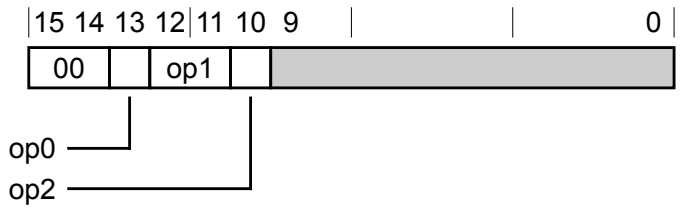

Table F3-10 Encoding table for the Shift (immediate), add, subtract, move, and compare group

Decode fields			Decode group or instruction page
op0	op1	op2	
0	11	0	Add, subtract (three low registers)
0	11	1	Add, subtract (two low registers and immediate)
0	!= 11	x	MOV, MOVS (register) - T2 variant
1	xx	x	Add, subtract, compare, move (one low register and immediate)

F3.1.2.1 Add, subtract (three low registers)

This section describes the encoding of the Add, subtract (three low registers) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare.

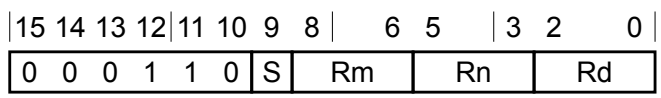

Table F3-11

Decode fields	Instruction page
S	
0	ADD, ADDS (register)
1	SUB, SUBS (register)

F3.1.2.2 Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10	9	8 6	5 3	2 0
0 0 0 1	1 1	S	imm3	Rn	Rd

Table F3-12

Decode fields S	Instruction page
0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

F3.1.2.3 Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare.

15 14 13 12	11 10 8	7 0
0 0 1 op	Rd	imm8

Table F3-13

Decode fields op	Instruction page
00	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

F3.1.3 Special data instructions and branch and exchange

This section describes the encoding of the Special data instructions and branch and exchange group. The encodings in this section are decoded from 16-bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15	10	9	8	7	0
010001		op0			

Table F3-14 Encoding table for the Special data instructions and branch and exchange group

Decode fields op0	Decode group or instruction page
`11`	Branch and exchange
`!= 11`	Add, subtract, compare, move (two high registers)

F3.1.3.1 Branch and exchange

This section describes the encoding of the Branch and exchange instruction class. The encodings in this section are decoded from Special data instructions and branch and exchange.

15	14	13	12	11	10	9	8	7	6	3	2	1	0
0	1	0	0	0	1	1	1	L	Rm		(0)	(0)	(0)

Table F3-15

Decode fields L	Instruction page
`0`	BX
`1`	BLX (register)

F3.1.3.2 Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. The encodings in this section are decoded from Special data instructions and branch and exchange.

15	14	13	12	11	10	9	8	7	6	3	2	0
0	1	0	0	0	1	!=11		D	Rs		Rd	
						op						

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-16

Decode fields			Instruction page
op	D:Rd	Rs	
00	!= 1101	!= 1101	ADD, ADDS (register)
00	xxxx	1101	ADD, ADDS (SP plus register) - T1
00	1101	!= 1101	ADD, ADDS (SP plus register) - T2
01	xxxx	xxxx	CMP (register)
10	xxxx	xxxx	MOV, MOVS (register)

F3.1.4 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions group. The encodings in this section are decoded from 16-bit.

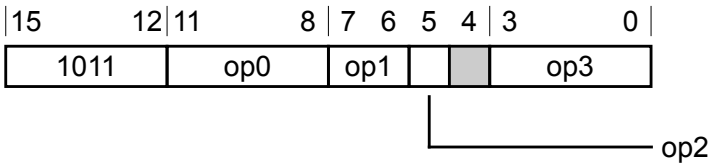

Table F3-17 Encoding table for the Miscellaneous 16-bit instructions group

Decode fields				Decode group or instruction page	Feature
op0	op1	op2	op3		
0000	xx	x	xxxx	Adjust SP (immediate)	-
0010	xx	x	xxxx	Extend	-
0110	00	0	xxxx	SETPAN	FEAT_PAN
0110	00	1	xxxx	Unallocated.	-
0110	01	x	xxxx	Change Processor State	-
0110	1x	x	xxxx	Unallocated.	-
0111	xx	x	xxxx	Unallocated.	-
1000	xx	x	xxxx	Unallocated.	-
1010	10	x	xxxx	HLT	-
1010	!= 10	x	xxxx	Reverse bytes	-
1110	xx	x	xxxx	BKPT	-
1111	xx	x	0000	Hints	-
1111	xx	x	!= 0000	IT	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Decode group or instruction page	Feature
op0	op1	op2	op3		
x0x1	xx	x	xxxx	CBNZ, CBZ	-
x10x	xx	x	xxxx	Push and Pop	-

F3.1.4.1 Adjust SP (immediate)

This section describes the encoding of the Adjust SP (immediate) instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

15	14	13	12	11	10	9	8	7	6		0
1	0	1	1	0	0	0	0	S	imm7		

Table F3-18

Decode fields	Instruction page
S	
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

F3.1.4.2 Extend

This section describes the encoding of the Extend instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

15	14	13	12	11	10	9	8	7	6	5	3	2	0
1	0	1	1	0	0	1	0	U	B	Rm		Rd	

Table F3-19

Decode fields		Instruction page
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
1	1	UXTB

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1.4.3 Change Processor State

This section describes the encoding of the Change Processor State instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

15	14	13	12	11	10	9	8	7	6	5	4	0
1	0	1	1	0	1	1	0	0	1	op	flags	

Table F3-20

Decode fields		Instruction page
op	flags	
0	xxxxx	SETEND
1	0xxxx	CPS, CPSID, CPSIE - Interrupt enable variant
1	1xxxx	CPS, CPSID, CPSIE - Interrupt disable variant

F3.1.4.4 Reverse bytes

This section describes the encoding of the Reverse bytes instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

15	14	13	12	11	10	9	8	7	6	5	3	2	0
1	0	1	1	1	0	1	0	!=10		Rm		Rd	
								op					

Table F3-21

Decode fields	Instruction page
op	
00	REV
01	REV16
11	REVSH

F3.1.4.5 Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15	14	13	12	11	10	9	8	7 – 4	3	2	1	0
1	0	1	1	1	1	1	1	hint	0	0	0	0

Table F3-22

Decode fields hint	Instruction page
0000	NOP
0001	YIELD
0010	WFE
0011	WFI
0100	SEV
0101	SEVL
011x	Reserved hint, behaves as NOP.
1xxx	Reserved hint, behaves as NOP.

F3.1.4.6 Push and Pop

This section describes the encoding of the Push and Pop instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

15	14	13	12	11	10	9	8	7 – 0
1	0	1	1	L	1	0	P	register_list

Table F3-23

Decode fields L	Instruction page
0	PUSH
1	POP

F3.1.5 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call group. The encodings in this section are decoded from 16-bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

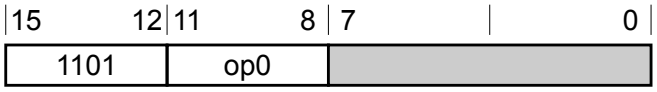

Table F3-24 Encoding table for the Conditional branch, and Supervisor Call group

Decode fields op0	Decode group or instruction page
`111x`	Exception generation
`!= 111x`	B - T1 variant

F3.1.5.1 Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from Conditional branch, and Supervisor Call.

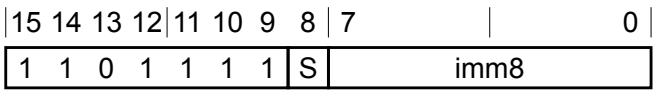

Table F3-25

Decode fields S	Instruction page
`0`	UDF
`1`	SVC

F3.1.6 32-bit

This section describes the encoding of the 32-bit group. The encodings in this section are decoded from T32 instruction set encoding.

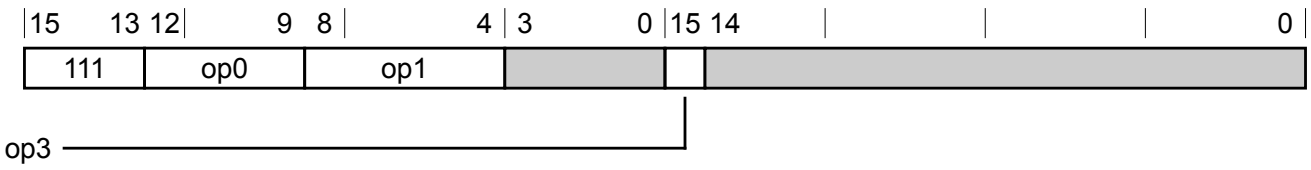

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This decode also imposes the constraint:

- `op0<3:2> != 00`.

Table F3-26 Encoding table for the 32-bit group

Decode fields			Decode group or instruction page
op0	op1	op3	
x11x	xxxxx	x	System register access, Advanced SIMD, and floating-point
0100	xx0xx	x	Load/store multiple
0100	xx1xx	x	Load/store dual, load/store exclusive, load-acquire/store-release, and table branch
0101	xxxxx	x	Data-processing (shifted register)
10xx	xxxxx	1	Branches and miscellaneous control
10x0	xxxxx	0	Data-processing (modified immediate)
10x1	xxxx0	0	Data-processing (plain binary immediate)
10x1	xxxx1	0	Unallocated.
1100	1xxx0	x	Advanced SIMD element or structure load/store
1100	!= 1xxx0	x	Load/store single
1101	0xxxx	x	Data-processing (register)
1101	10xxx	x	Multiply, multiply accumulate, and absolute difference
1101	11xxx	x	Long multiply and divide

F3.1.6.1 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from 32-bit.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13	0
1 1 1 0 1 0 0	opc	0 W L	Rn	P M	register_list

Table F3-27

Decode fields		Instruction page
opc	L	
00	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - T1
00	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - T1
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page
opc	L	
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - T2
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - T2

F3.1.6.2 Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. The encodings in this section are decoded from 32-bit.

15 14 13 12	11 10 9 8	5 4	3 0	15 14 12	11 8	7 6 5 4	3 0
1 1 1 0 1 0 1	op1	S	Rn	(0) imm3	Rd	imm2 stype	Rm

Table F3-28

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
0000	0	xxxx	!= 0000011	xxxx	AND, ANDS (register) - AND, shift or rotate by value variant
0000	0	xxxx	0000011	xxxx	AND, ANDS (register) - AND, rotate right with extend variant
0000	1	xxxx	!= 0000011	!= 1111	AND, ANDS (register) - ANDS, shift or rotate by value variant
0000	1	xxxx	!= 0000011	1111	TST (register) - Shift or rotate by value variant
0000	1	xxxx	0000011	!= 1111	AND, ANDS (register) - ANDS, rotate right with extend variant
0000	1	xxxx	0000011	1111	TST (register) - Rotate right with extend variant
0001	x	xxxx	!= 0000011	xxxx	BIC, BICS (register) - BICS, shift or rotate by value variant
0001	x	xxxx	0000011	xxxx	BIC, BICS (register) - BICS, rotate right with extend variant
0010	0	!= 1111	!= 0000011	xxxx	ORR, ORRS (register) - ORR, shift or rotate by value variant
0010	0	!= 1111	0000011	xxxx	ORR, ORRS (register) - ORR, rotate right with extend variant
0010	0	1111	!= 0000011	xxxx	MOV, MOVS (register) - MOV, shift or rotate by value variant
0010	0	1111	0000011	xxxx	MOV, MOVS (register) - MOV, rotate right with extend variant
0010	1	!= 1111	!= 0000011	xxxx	ORR, ORRS (register) - ORRS, shift or rotate by value variant
0010	1	!= 1111	0000011	xxxx	ORR, ORRS (register) - ORRS, rotate right with extend variant
0010	1	1111	!= 0000011	xxxx	MOV, MOVS (register) - MOVS, shift or rotate by value variant
0010	1	1111	0000011	xxxx	MOV, MOVS (register) - MOVS, rotate right with extend variant
0011	0	!= 1111	!= 0000011	xxxx	ORN, ORNS (register) - ORN, shift or rotate by value variant
0011	0	!= 1111	0000011	xxxx	ORN, ORNS (register) - ORN, rotate right with extend variant
0011	0	1111	!= 0000011	xxxx	MVN, MVNS (register) - MVN, shift or rotate by value variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
0011	0	1111	0000011	xxxx	MVN, MVNS (register) - MVN, rotate right with extend variant
0011	1	!= 1111	!= 0000011	xxxx	ORN, ORNS (register) - ORNS, shift or rotate by value variant
0011	1	!= 1111	0000011	xxxx	ORN, ORNS (register) - ORNS, rotate right with extend variant
0011	1	1111	!= 0000011	xxxx	MVN, MVNS (register) - MVNS, shift or rotate by value variant
0011	1	1111	0000011	xxxx	MVN, MVNS (register) - MVNS, rotate right with extend variant
0100	0	xxxx	!= 0000011	xxxx	EOR, EORS (register) - EOR, shift or rotate by value variant
0100	0	xxxx	0000011	xxxx	EOR, EORS (register) - EOR, rotate right with extend variant
0100	1	xxxx	!= 0000011	!= 1111	EOR, EORS (register) - EORS, shift or rotate by value variant
0100	1	xxxx	!= 0000011	1111	TEQ (register) - Shift or rotate by value variant
0100	1	xxxx	0000011	!= 1111	EOR, EORS (register) - EORS, rotate right with extend variant
0100	1	xxxx	0000011	1111	TEQ (register) - Rotate right with extend variant
0101	x	xxxx	xxxxxxx	xxxx	Unallocated.
0110	0	xxxx	xxxxx00	xxxx	PKHBT, PKHTB - PKHBT variant
0110	0	xxxx	xxxxx01	xxxx	Unallocated.
0110	0	xxxx	xxxxx10	xxxx	PKHBT, PKHTB - PKHTB variant
0110	0	xxxx	xxxxx11	xxxx	Unallocated.
0111	x	xxxx	xxxxxxx	xxxx	Unallocated.
1000	0	!= 1101	!= 0000011	xxxx	ADD, ADDS (register) - ADD, shift or rotate by value variant
1000	0	!= 1101	0000011	xxxx	ADD, ADDS (register) - ADD, rotate right with extend variant
1000	0	1101	!= 0000011	xxxx	ADD, ADDS (SP plus register) - ADD, shift or rotate by value variant
1000	0	1101	0000011	xxxx	ADD, ADDS (SP plus register) - ADD, rotate right with extend variant
1000	1	xxxx	!= 0000011	1111	CMN (register) - Shift or rotate by value variant
1000	1	!= 1101	!= 0000011	!= 1111	ADD, ADDS (register) - ADDS, shift or rotate by value variant
1000	1	!= 1101	0000011	!= 1111	ADD, ADDS (register) - ADDS, rotate right with extend variant
1000	1	xxxx	0000011	1111	CMN (register) - Rotate right with extend variant
1000	1	1101	!= 0000011	!= 1111	ADD, ADDS (SP plus register) - ADDS, shift or rotate by value variant
1000	1	1101	0000011	!= 1111	ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant
1001	x	xxxx	xxxxxxx	xxxx	Unallocated.
1010	x	xxxx	!= 0000011	xxxx	ADC, ADCS (register) - ADCS, shift or rotate by value variant
1010	x	xxxx	0000011	xxxx	ADC, ADCS (register) - ADCS, rotate right with extend variant
1011	x	xxxx	!= 0000011	xxxx	SBC, SBCS (register) - SBCS, shift or rotate by value variant
1011	x	xxxx	0000011	xxxx	SBC, SBCS (register) - SBCS, rotate right with extend variant
1100	x	xxxx	xxxxxxx	xxxx	Unallocated.
1101	0	!= 1101	!= 0000011	xxxx	SUB, SUBS (register) - SUB, shift or rotate by value variant
1101	0	!= 1101	0000011	xxxx	SUB, SUBS (register) - SUB, rotate right with extend variant
1101	0	1101	!= 0000011	xxxx	SUB, SUBS (SP minus register) - SUB, shift or rotate by value variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
1101	0	1101	0000011	xxxx	SUB, SUBS (SP minus register) - SUB, rotate right with extend variant
1101	1	xxxx	!= 0000011	1111	CMP (register) - Shift or rotate by value variant
1101	1	!= 1101	!= 0000011	!= 1111	SUB, SUBS (register) - SUBS, shift or rotate by value variant
1101	1	!= 1101	0000011	!= 1111	SUB, SUBS (register) - SUBS, rotate right with extend variant
1101	1	xxxx	0000011	1111	CMP (register) - Rotate right with extend variant
1101	1	1101	!= 0000011	!= 1111	SUB, SUBS (SP minus register) - SUBS, shift or rotate by value variant
1101	1	1101	0000011	!= 1111	SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant
1110	x	xxxx	!= 0000011	xxxx	RSB, RSBS (register) - RSBS, shift or rotate by value variant
1110	x	xxxx	0000011	xxxx	RSB, RSBS (register) - RSBS, rotate right with extend variant
1111	x	xxxx	xxxxxxx	xxxx	Unallocated.

F3.1.6.3 Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. The encodings in this section are decoded from 32-bit.

15 14 13 12	11	10	9 8	5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	op1	S	Rn	0	imm3	Rd	imm8

Table F3-29

Decode fields				Instruction page
op1	S	Rn	Rd	
0000	0	xxxx	xxxx	AND, ANDS (immediate) - AND variant
0000	1	xxxx	!= 1111	AND, ANDS (immediate) - ANDS variant
0000	1	xxxx	1111	TST (immediate)
0001	x	xxxx	xxxx	BIC, BICS (immediate)
0010	0	!= 1111	xxxx	ORR, ORRS (immediate) - ORR variant
0010	0	1111	xxxx	MOV, MOVS (immediate) - MOV variant
0010	1	!= 1111	xxxx	ORR, ORRS (immediate) - ORRS variant
0010	1	1111	xxxx	MOV, MOVS (immediate) - MOVS variant
0011	0	!= 1111	xxxx	ORN, ORNS (immediate) - Not flag setting variant
0011	0	1111	xxxx	MVN, MVNS (immediate) - MVN variant
0011	1	!= 1111	xxxx	ORN, ORNS (immediate) - Flag setting variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
op1	S	Rn	Rd	
0011	1	1111	xxxx	MVN, MVNS (immediate) - MVNS variant
0100	0	xxxx	xxxx	EOR, EORS (immediate) - EOR variant
0100	1	xxxx	!= 1111	EOR, EORS (immediate) - EORS variant
0100	1	xxxx	1111	TEQ (immediate)
0101	x	xxxx	xxxx	Unallocated.
011x	x	xxxx	xxxx	Unallocated.
1000	0	!= 1101	xxxx	ADD, ADDS (immediate) - ADD variant
1000	0	1101	xxxx	ADD, ADDS (SP plus immediate) - ADD variant
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) - ADDS variant
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) - ADDS variant
1000	1	xxxx	1111	CMN (immediate)
1001	x	xxxx	xxxx	Unallocated.
1010	x	xxxx	xxxx	ADC, ADCS (immediate)
1011	x	xxxx	xxxx	SBC, SBCS (immediate)
1100	x	xxxx	xxxx	Unallocated.
1101	0	!= 1101	xxxx	SUB, SUBS (immediate) - SUB variant
1101	0	1101	xxxx	SUB, SUBS (SP minus immediate) - SUB variant
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) - SUBS variant
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) - SUBS variant
1101	1	xxxx	1111	CMP (immediate)
1110	x	xxxx	xxxx	RSB, RSBS (immediate)
1111	x	xxxx	xxxx	Unallocated.

F3.1.6.4 Long multiply and divide

This section describes the encoding of the Long multiply and divide instruction class. The encodings in this section are decoded from 32-bit.

15 14 13 12	11 10 9 8	7 6 4	3 0	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 1 1	op1	Rn	RdLo	RdHi	op2	Rm

Table F3-30

Decode fields		Instruction page
op1	op2	
000	!= 0000	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page
op1	op2	
000	0000	SMULL, SMULLS
001	!= 1111	Unallocated.
001	1111	SDIV
010	!= 0000	Unallocated.
010	0000	UMULL, UMULLS
011	!= 1111	Unallocated.
011	1111	UDIV
100	0000	SMLAL, SMLALS
100	0001	Unallocated.
100	001x	Unallocated.
100	01xx	Unallocated.
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBB variant
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBT variant
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTB variant
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTT variant
100	1100	SMLALD, SMLALDX - SMLALD variant
100	1101	SMLALD, SMLALDX - SMLALDX variant
100	111x	Unallocated.
101	0xxx	Unallocated.
101	10xx	Unallocated.
101	1100	SMLSLD, SMLSLDX - SMLSLD variant
101	1101	SMLSLD, SMLSLDX - SMLSLDX variant
101	111x	Unallocated.
110	0000	UMLAL, UMLALS
110	0001	Unallocated.
110	001x	Unallocated.
110	010x	Unallocated.
110	0110	UMAAL
110	0111	Unallocated.
110	1xxx	Unallocated.
111	xxxx	Unallocated.

F3.1.7 System register access, Advanced SIMD, and floating-point

This section describes the encoding of the System register access, Advanced SIMD, and floating-point group. The encodings in this section are decoded from 32-bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

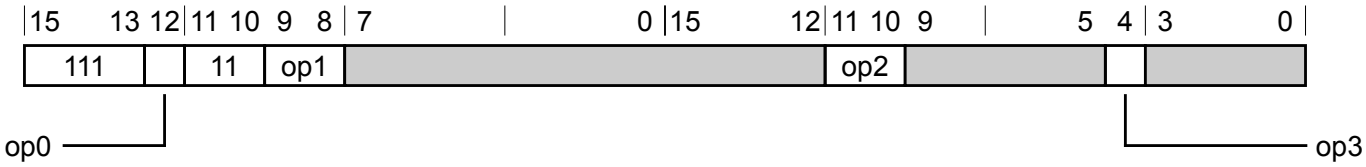

Table F3-31 Encoding table for the System register access, Advanced SIMD, and floating-point group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
x	0x	0x	x	Unallocated.
x	10	0x	x	Unallocated.
x	11	xx	x	Advanced SIMD data-processing
0	0x	1x	x	Advanced SIMD and System register load/store and 64-bit move
0	10	1x	1	Advanced SIMD and System register 32-bit move
0	10	10	0	Floating-point data-processing
0	10	11	0	Unallocated.
1	!= 11	1x	x	Additional Advanced SIMD and floating-point instructions

F3.1.8 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding

Table F3-32 Encoding table for the Advanced SIMD data-processing group

Decode fields		Decode group or instruction page
op0	op1	
0	x	Advanced SIMD three registers of the same length

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Decode group or instruction page
op0	op1	
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

F3.1.8.1 Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from Advanced SIMD data-processing.

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	0	D	size	Vn	Vd	opc	N	Q	M	o1	Vm

Table F3-33

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
0	0x	1100	x	1	VFMA	-
0	0x	1101	x	0	VADD (floating-point)	-
0	0x	1101	x	1	VMLA (floating-point)	-
0	0x	1110	x	0	VCEQ (register) - T2	-
0	0x	1111	x	0	VMAX (floating-point)	-
0	0x	1111	x	1	VRECPS	-
x	xx	0000	x	0	VHADD	-
0	00	0001	x	1	VAND (register)	-
x	xx	0000	x	1	VQADD	-
x	xx	0001	x	0	VRHADD	-
0	00	1100	x	0	SHA1C	FEAT_SHA1
x	xx	0010	x	0	VHSUB	-
0	01	0001	x	1	VBIC (register)	-
x	xx	0010	x	1	VQSUB	-
x	xx	0011	x	0	VCGT (register) - T1	-
x	xx	0011	x	1	VCGE (register) - T1	-
0	01	1100	x	0	SHA1P	FEAT_SHA1
0	1x	1100	x	1	VFMS	-
0	1x	1101	x	0	VSUB (floating-point)	-
0	1x	1101	x	1	VMLS (floating-point)	-
0	1x	1110	x	0	Unallocated.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
0	1x	1111	x	0	VMIN (floating-point)	-
0	1x	1111	x	1	VRSQRTS	-
x	xx	0100	x	0	VSHL (register)	-
0	xx	1000	x	0	VADD (integer)	-
0	10	0001	x	1	VORR (register)	-
0	xx	1000	x	1	VTST	-
x	xx	0100	x	1	VQSHL (register)	-
0	xx	1001	x	0	VMLA (integer)	-
x	xx	0101	x	0	VRSHL	-
x	xx	0101	x	1	VQRSHL	-
0	xx	1011	x	0	VQDMULH	-
0	10	1100	x	0	SHA1M	FEAT_SHA1
0	xx	1011	x	1	VPADD (integer)	-
x	xx	0110	x	0	VMAX (integer)	-
0	11	0001	x	1	VORN (register)	-
x	xx	0110	x	1	VMIN (integer)	-
x	xx	0111	x	0	VABD (integer)	-
x	xx	0111	x	1	VABA	-
0	11	1100	x	0	SHA1SU0	FEAT_SHA1
1	0x	1101	x	0	VPADD (floating-point)	-
1	0x	1101	x	1	VMUL (floating-point)	-
1	0x	1110	x	0	VCGE (register) - T2	-
1	0x	1110	x	1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	x	1	VMAXNM	-
1	00	0001	x	1	VEOR	-
x	xx	1001	x	1	VMUL (integer and polynomial)	-
1	00	1100	x	0	SHA256H	FEAT_SHA256
x	xx	1010	0	0	VPMAX (integer)	-
1	01	0001	x	1	VBSL	-
x	xx	1010	0	1	VPMIN (integer)	-
x	xx	1010	1	x	Unallocated.	-
1	01	1100	x	0	SHA256H2	FEAT_SHA256
1	1x	1101	x	0	VABD (floating-point)	-
1	1x	1110	x	0	VCGT (register) - T2	-
1	1x	1110	x	1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111	x	1	VMINNM	-
1	xx	1000	x	0	VSUB (integer)	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
1	10	0001	x	1	VBIT	-
1	xx	1000	x	1	VCEQ (register) - T1	-
1	xx	1001	x	0	VMLS (integer)	-
1	xx	1011	x	0	VQRDMULH	-
1	10	1100	x	0	SHA256SU1	FEAT_SHA256
1	xx	1011	x	1	VQRDMLAH	FEAT_RDM
1	11	0001	x	1	VBIF	-
1	xx	1100	x	1	VQRDMLSH	FEAT_RDM
1	xx	1111	1	0	Unallocated.	-

F3.1.9 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from Advanced SIMD data-processing.

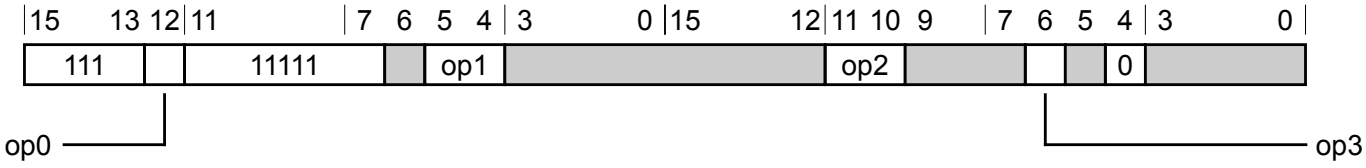

Table F3-34 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	11	xx	x	VEXT (byte elements)
1	11	0x	x	Advanced SIMD two registers misc
1	11	10	x	VTBL, VTBX
1	11	11	x	Advanced SIMD duplicate (scalar)
x	!= 11	xx	0	Advanced SIMD three registers of different lengths
x	!= 11	xx	1	Advanced SIMD two registers and a scalar

F3.1.9.1 Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11 10	7 6 5 4	3 0
1 1 1 1	1 1 1 1	1 D 1 1	size opc1	Vd	0 opc2	Q M 0	Vm

Table F3-35

Decode fields				Instruction page	Feature
size	**opc1**	**opc2**	**Q**		
xx	00	0000	x	VREV64	-
xx	00	0001	x	VREV32	-
xx	00	0010	x	VREV16	-
xx	00	0011	x	Unallocated.	-
xx	00	010x	x	VPADDL	-
xx	00	0110	0	AESE	FEAT_AES
xx	00	0110	1	AESD	FEAT_AES
xx	00	0111	0	AESMC	FEAT_AES
xx	00	0111	1	AESIMC	FEAT_AES
xx	00	1000	x	VCLS	-
00	10	0000	x	VSWP	-
xx	00	1001	x	VCLZ	-
xx	00	1010	x	VCNT	-
xx	00	1011	x	VMVN (register)	-
00	10	1100	1	Unallocated.	-
xx	00	110x	x	VPADAL	-
xx	00	1110	x	VQABS	-
xx	00	1111	x	VQNEG	-
xx	01	x000	x	VCGT (immediate #0)	-
xx	01	x001	x	VCGE (immediate #0)	-
xx	01	x010	x	VCEQ (immediate #0)	-
xx	01	x011	x	VCLE (immediate #0)	-
xx	01	x100	x	VCLT (immediate #0)	-
xx	01	x110	x	VABS	-
xx	01	x111	x	VNEG	-
xx	01	0101	1	SHA1H	FEAT_SHA1
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
xx	10	0001	x	VTRN	-
xx	10	0010	x	VUZP	-
xx	10	0011	x	VZIP	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
size	opc1	opc2	Q		
xx	10	0100	0	VMOVN	-
xx	10	0100	1	VQMOVN, VQMOVUN - Unsigned result variant	-
xx	10	0101	x	VQMOVN, VQMOVUN - Signed result variant	-
xx	10	0110	0	VSHLL	-
xx	10	0111	0	SHA1SU1	FEAT_SHA1
xx	10	0111	1	SHA256SU0	FEAT_SHA256
xx	10	1000	x	VRINTN (Advanced SIMD)	-
xx	10	1001	x	VRINTX (Advanced SIMD)	-
xx	10	1010	x	VRINTA (Advanced SIMD)	-
xx	10	1011	x	VRINTZ (Advanced SIMD)	-
10	10	1100	1	Unallocated.	-
xx	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant	-
xx	10	1101	x	VRINTM (Advanced SIMD)	-
xx	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant	-
xx	10	1110	1	Unallocated.	-
xx	10	1111	x	VRINTP (Advanced SIMD)	-
xx	11	000x	x	VCVTA (Advanced SIMD)	-
xx	11	001x	x	VCVTN (Advanced SIMD)	-
xx	11	010x	x	VCVTP (Advanced SIMD)	-
xx	11	011x	x	VCVTM (Advanced SIMD)	-
xx	11	10x0	x	VRECPE	-
xx	11	10x1	x	VRSQRTE	-
11	10	1100	1	Unallocated.	-
xx	11	11xx	x	VCVT (between floating-point and integer, Advanced SIMD)	-

F3.1.9.2 Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10	9 7	6	5	4	3 0
1 1 1 1	1 1 1 1	1	D	1 1	imm4	Vd	1 1	opc	Q	M	0	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-36

Decode fields opc	Instruction page
000	VDUP (scalar)
001	Unallocated.
01x	Unallocated.
1xx	Unallocated.

F3.1.9.3 Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	!=11 size	Vn	Vd	opc	N	0	M	0	Vm

Table F3-37

Decode fields U	 opc	Instruction page
x	0000	VADDL
x	0001	VADDW
x	0010	VSUBL
0	0100	VADDHN
x	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
x	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
x	0111	VABDL (integer)
x	1000	VMLAL (integer)
x	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
x	11x0	VMULL (integer and polynomial)
1	1001	Unallocated.
1	1011	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page
U	opc	
1	1101	Unallocated.
x	1111	Unallocated.

F3.1.9.4 Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

15 14 13	12	11 10 9 8	7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	Q	1 1 1 1	1	D	!=11 (size)	Vn	Vd	opc	N	1	M	0	Vm

Table F3-38

Decode fields		Instruction page	Feature
Q	opc		
x	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
x	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
x	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
x	0110	VMLSL (by scalar)	-
x	100x	VMUL (by scalar)	-
1	0011	Unallocated.	-
x	1010	VMULL (by scalar)	-
1	0111	Unallocated.	-
x	1100	VQDMULH	-
x	1101	VQRDMULH	-
1	1011	Unallocated.	-
x	1110	VQRDMLAH	FEAT_RDM
x	1111	VQRDMLSH	FEAT_RDM

F3.1.10 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from Advanced SIMD data-processing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 13	12	11 7	6	5 0	15 7	6	5 4	3 0
111		11111		op0			1	

Table F3-39 Encoding table for the Advanced SIMD shifts and immediate generation group

Decode fields op0	Decode group or instruction page
`000xxxxxxxxxxx0`	Advanced SIMD one register and modified immediate
`!= 000xxxxxxxxxxx0`	Advanced SIMD two registers and shift amount

F3.1.10.1 Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.

15 14 13	12	11 10 9 8	7	6	5 4	3 2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	i	1 1 1 1	1	D	0 0	0 imm3	Vd	cmode	0	Q	op	1	imm4

Table F3-40

Decode fields cmode	op	Instruction page
`0xx0`	`0`	VMOV (immediate) - T1
`0xx0`	`1`	VMVN (immediate) - T1
`0xx1`	`0`	VORR (immediate) - T1
`0xx1`	`1`	VBIC (immediate) - T1
`10x0`	`0`	VMOV (immediate) - T3
`10x0`	`1`	VMVN (immediate) - T2
`10x1`	`0`	VORR (immediate) - T2
`10x1`	`1`	VBIC (immediate) - T2
`11xx`	`0`	VMOV (immediate) - T4
`110x`	`1`	VMVN (immediate) - T3
`1110`	`1`	VMOV (immediate) - T5
`1111`	`1`	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1.10.2 Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.

15 14 13	12	11 10 9 8	7	6	5 3	2 0	15 12	11 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1	1	D	imm3H	imm3L	Vd	opc	L	Q	M	1	Vm

This decode also imposes the constraint:

- `!(imm3H == '000' && L == '0').`

Table F3-41

Decode fields						Instruction page
U	**imm3H**	**imm3L**	**opc**	**L**	**Q**	
x	xxx	xxx	1xxx	1	x	Unallocated.
x	000	xxx	0000	1	0	VSHR 64-bit SIMD vector
x	000	xxx	0000	1	1	VSHR 128-bit SIMD vector
x	000	xxx	0001	1	0	VSRA 64-bit SIMD vector
x	000	xxx	0001	1	1	VSRA 128-bit SIMD vector
x	000	xxx	0010	1	0	VRSHR 64-bit SIMD vector
x	000	xxx	0010	1	1	VRSHR 128-bit SIMD vector
x	000	xxx	0011	1	0	VRSRA 64-bit SIMD vector
x	000	xxx	0011	1	1	VRSRA 128-bit SIMD vector
x	000	xxx	0111	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	000	xxx	0111	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	001	000	1010	0	0	VMOVL
x	001	!= 000	1010	0	0	VSHLL
x	010	000	1010	0	0	VMOVL
x	010	!= 000	1010	0	0	VSHLL
x	011	xxx	1010	0	0	VSHLL
x	100	000	1010	0	0	VMOVL
x	100	!= 000	1010	0	0	VSHLL
x	101	xxx	1010	0	0	VSHLL
x	11x	xxx	1010	0	0	VSHLL
x	!= 000	xxx	0000	x	0	VSHR 64-bit SIMD vector
x	!= 000	xxx	0000	x	1	VSHR 128-bit SIMD vector
x	!= 000	xxx	0001	x	0	VSRA 64-bit SIMD vector
x	!= 000	xxx	0001	x	1	VSRA 128-bit SIMD vector

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
x	!= 000	xxx	0010	x	0	VRSHR 64-bit SIMD vector
x	!= 000	xxx	0010	x	1	VRSHR 128-bit SIMD vector
x	!= 000	xxx	0011	x	0	VRSRA 64-bit SIMD vector
x	!= 000	xxx	0011	x	1	VRSRA 128-bit SIMD vector
x	!= 000	xxx	0111	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	!= 000	xxx	0111	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	!= 000	xxx	1001	0	0	VQSHRN, VQSHRUN — Signed result
x	!= 000	xxx	1001	0	1	VQRSHRN, VQRSHRUN — Signed result
x	!= 000	xxx	1010	0	1	Unallocated.
x	!= 000	xxx	1011	0	x	Unallocated.
x	!= 000	xxx	11xx	0	0	VCVT (between floating-point and fixed-point, Advanced SIMD) — 64-bit SIMD vector
x	!= 000	xxx	11xx	0	1	VCVT (between floating-point and fixed-point, Advanced SIMD) — 128-bit SIMD vector
0	xxx	xxx	01x0	1	x	Unallocated.
0	000	xxx	0101	1	0	VSHL (immediate) — 64-bit SIMD vector
0	000	xxx	0101	1	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	01x0	0	x	Unallocated.
0	!= 000	xxx	0101	x	0	VSHL (immediate) — 64-bit SIMD vector
0	!= 000	xxx	0101	x	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	1000	0	0	VSHRN
0	!= 000	xxx	1000	0	1	VRSHRN
1	000	xxx	0100	1	0	VSRI — 64-bit SIMD vector
1	000	xxx	0100	1	1	VSRI — 128-bit SIMD vector
1	000	xxx	0101	1	0	VSLI — 64-bit SIMD vector
1	000	xxx	0101	1	1	VSLI — 128-bit SIMD vector
1	000	xxx	0110	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	000	xxx	0110	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	0100	x	0	VSRI — 64-bit SIMD vector
1	!= 000	xxx	0100	x	1	VSRI — 128-bit SIMD vector
1	!= 000	xxx	0101	x	0	VSLI — 64-bit SIMD vector
1	!= 000	xxx	0101	x	1	VSLI — 128-bit SIMD vector
1	!= 000	xxx	0110	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	!= 000	xxx	0110	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	1000	0	0	VQSHRN, VQSHRUN — Unsigned result
1	!= 000	xxx	1000	0	1	VQRSHRN, VQRSHRUN — Unsigned result

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1.11 Advanced SIMD and System register load/store and 64-bit move

This section describes the encoding of the Advanced SIMD and System register load/store and 64-bit move group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding

15 … 9	8 … 5	4 … 0	15 … 12	11	10 … 9	8 … 0
1110110	op0			1	op1	

Table F3-42 Encoding table for the Advanced SIMD and System register load/store and 64-bit move group

Decode fields		Decode group or instruction page
op0	op1	
`00x0`	`0x`	Advanced SIMD and floating-point 64-bit move
`00x0`	`11`	System register 64-bit move
`!= 00x0`	`0x`	Advanced SIMD and floating-point load/store
`!= 00x0`	`11`	System register Load/Store
`xxxx`	`10`	Unallocated.

F3.1.11.1 Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

15 14 13 12	11 10 9 8	7 6	5	4	3 … 0	15 … 12	11 10	9 8	7 6	5	4	3 … 0
1 1 1 0	1 1 0 0	0 D	0	op	Rt2	Rt	1 0	size	opc2	M	o3	Vm

Table F3-43

Decode fields					Instruction page
D	op	size	opc2	o3	
`0`	`x`	`xx`	`xx`	`x`	Unallocated.
`1`	`x`	`xx`	`xx`	`0`	Unallocated.
`1`	`x`	`0x`	`00`	`1`	Unallocated.
`1`	`x`	`xx`	`01`	`x`	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page
D	op	size	opc2	o3	
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - From general-purpose registers variant
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - From general-purpose registers variant
1	x	xx	1x	x	Unallocated.
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - To general-purpose registers variant
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - To general-purpose registers variant

F3.1.11.2 System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

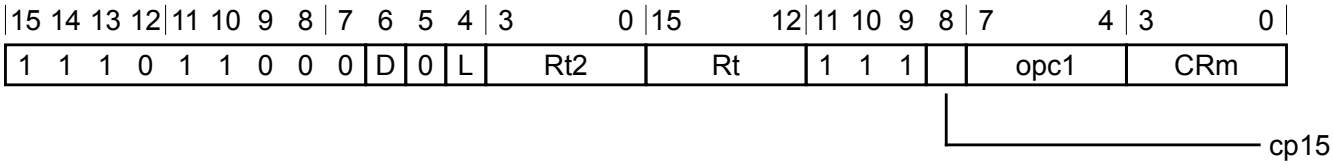

Table F3-44

Decode fields		Instruction page
D	L	
0	x	Unallocated.
1	0	MCRR
1	1	MRRC

F3.1.11.3 Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

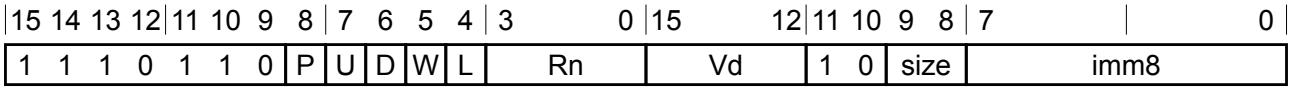

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-45

Decode fields							Instruction page	Feature
P	U	W	L	Rn	size	imm8		
0	0	1	x	xxxx	xx	xxxxxxxx	Unallocated.	-
0	1	x	x	xxxx	0x	xxxxxxxx	Unallocated.	-
0	1	x	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Increment After variant	-
0	1	x	0	xxxx	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - Increment After variant	-
0	1	x	0	xxxx	11	xxxxxxx1	FSTMDBX, FSTMIAX - Increment After variant	-
0	1	x	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Increment After variant	-
0	1	x	1	xxxx	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - Increment After variant	-
0	1	x	1	xxxx	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Increment After variant	-
1	x	0	0	xxxx	01	xxxxxxxx	VSTR - Half-precision scalar variant	FEAT_FP16
1	x	0	0	xxxx	10	xxxxxxxx	VSTR - Single-precision scalar variant	-
1	x	0	0	xxxx	11	xxxxxxxx	VSTR - Double-precision scalar variant	-
1	x	0	1	!= 1111	01	xxxxxxxx	VLDR (immediate) - Half-precision scalar variant	FEAT_FP16
1	x	0	1	!= 1111	10	xxxxxxxx	VLDR (immediate) - Single-precision scalar variant	-
1	x	0	1	!= 1111	11	xxxxxxxx	VLDR (immediate) - Double-precision scalar variant	-
1	0	1	x	xxxx	0x	xxxxxxxx	Unallocated.	-
1	0	1	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Decrement Before variant	-
1	0	1	0	xxxx	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - Decrement Before variant	-
1	0	1	0	xxxx	11	xxxxxxx1	FSTMDBX, FSTMIAX - Decrement Before variant	-
1	0	1	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Decrement Before variant	-
1	0	1	1	xxxx	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - Decrement Before variant	-
1	0	1	1	xxxx	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Decrement Before variant	-
1	x	0	1	1111	01	xxxxxxxx	VLDR (literal) - Half-precision scalar variant	FEAT_FP16
1	x	0	1	1111	10	xxxxxxxx	VLDR (literal) - Single-precision scalar variant	-
1	x	0	1	1111	11	xxxxxxxx	VLDR (literal) - Double-precision scalar variant	-
1	1	1	x	xxxx	xx	xxxxxxxx	Unallocated.	-

F3.1.11.4 System register Load/Store

This section describes the encoding of the System register Load/Store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7	6	5	4	3 0	15 12	11 10 9	8	7 0
1 1 1 0	1 1 0	P	U	D	W	L	Rn	CRd	1 1 1	imm8

cp15

Table F3-46

Decode fields						Instruction page
P:U:W	**D**	**L**	**Rn**	**CRd**	**cp15**	
`!= 000`	`x`	`x`	`xxxx`	`!= 0101`	`0`	Unallocated.
`!= 000`	`0`	`1`	`1111`	`0101`	`0`	LDC (literal)
`!= 000`	`x`	`x`	`xxxx`	`xxxx`	`1`	Unallocated.
`!= 000`	`1`	`x`	`xxxx`	`0101`	`0`	Unallocated.
`0x1`	`0`	`0`	`xxxx`	`0101`	`0`	STC - Post-indexed variant
`0x1`	`0`	`1`	`!= 1111`	`0101`	`0`	LDC (immediate) - Post-indexed variant
`010`	`0`	`0`	`xxxx`	`0101`	`0`	STC - Unindexed variant
`010`	`0`	`1`	`!= 1111`	`0101`	`0`	LDC (immediate) - Unindexed variant
`1x0`	`0`	`0`	`xxxx`	`0101`	`0`	STC - Offset variant
`1x0`	`0`	`1`	`!= 1111`	`0101`	`0`	LDC (immediate) - Offset variant
`1x1`	`0`	`0`	`xxxx`	`0101`	`0`	STC - Pre-indexed variant
`1x1`	`0`	`1`	`!= 1111`	`0101`	`0`	LDC (immediate) - Pre-indexed variant

F3.1.12 Advanced SIMD and System register 32-bit move

This section describes the encoding of the Advanced SIMD and System register 32-bit move group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point.

15 8	7 5	4 0	15 12	11	10 8	7 5	4	3 0
11101110	op0			1	op1		1	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-47 Encoding table for the Advanced SIMD and System register 32-bit move group

Decode fields		Decode group or instruction page	Feature
op0	op1		
000	000	Unallocated.	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	Unallocated.	-
01x	010	Unallocated.	-
10x	010	Unallocated.	-
110	010	Unallocated.	-
111	010	Floating-point move special register	-
xxx	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
xxx	10x	Unallocated.	-
xxx	11x	System register 32-bit move	-

F3.1.12.1 Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7	6	5	4	3	2	1	0
1 1 1 0	1 1 1 0	1 1 1 L	reg	Rt	1 0 1 0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Table F3-48

Decode fields	Instruction page
L	
0	VMSR
1	VMRS

F3.1.12.2 Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move.

15 14 13 12	11 10 9 8	7 5	4	3 0	15 12	11 10 9 8	7	6 5	4	3	2	1	0
1 1 1 0	1 1 1 0	opc1	L	Vn	Rt	1 0 1 1	N	opc2	1	(0)	(0)	(0)	(0)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-49

Decode fields			Instruction page
opc1	**L**	**opc2**	
`0xx`	`0`	`xx`	VMOV (general-purpose register to scalar)
`xxx`	`1`	`xx`	VMOV (scalar to general-purpose register)
`1xx`	`0`	`0x`	VDUP (general-purpose register)
`1xx`	`0`	`1x`	Unallocated.

F3.1.12.3 System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move.

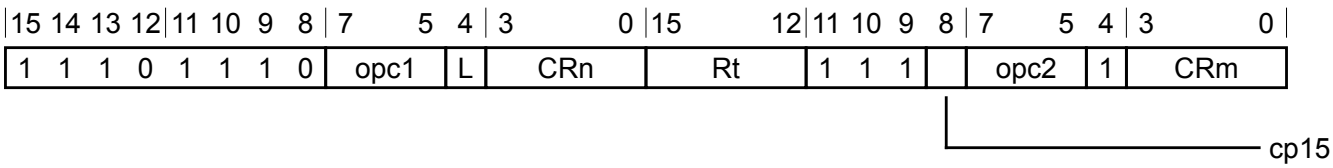

Table F3-50

Decode fields	Instruction page
L	
`0`	MCR
`1`	MRC

F3.1.13 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding

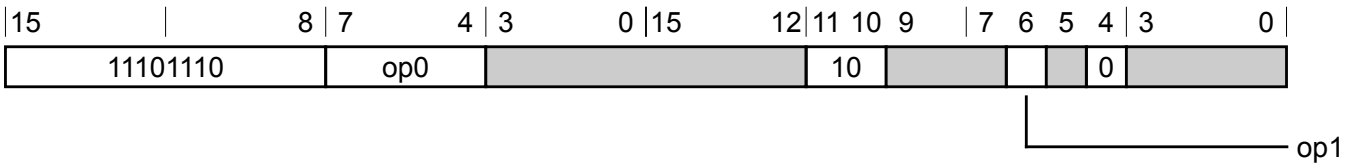

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-51 Encoding table for the Floating-point data-processing group

Decode fields		Decode group or instruction page
op0	op1	
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11	x	Floating-point data-processing (three registers)

F3.1.13.1 Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

15 14 13 12	11 10 9 8	7	6	5	4	3	2 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 0	1	D	1	1	o1	opc2	Vd	1 0	size	o3	1	M	0	Vm

Table F3-52

Decode fields				Instruction page	Feature
o1	opc2	size	o3		
x	xxx	00	x	Unallocated.	-
0	000	01	0	Unallocated.	-
0	000	01	1	VABS - Half-precision scalar variant	FEAT_FP16
0	000	10	0	VMOV (register) - Single-precision scalar variant	-
0	000	10	1	VABS - Single-precision scalar variant	-
0	000	11	0	VMOV (register) - Double-precision scalar variant	-
0	000	11	1	VABS - Double-precision scalar variant	-
0	001	01	0	VNEG - Half-precision scalar variant	FEAT_FP16
0	001	01	1	VSQRT - Half-precision scalar variant	FEAT_FP16
0	001	10	0	VNEG - Single-precision scalar variant	-
0	001	10	1	VSQRT - Single-precision scalar variant	-
0	001	11	0	VNEG - Double-precision scalar variant	-
0	001	11	1	VSQRT - Double-precision scalar variant	-
0	010	01	x	Unallocated.	-
0	010	10	0	VCVTB - Half-precision to single-precision variant	-
0	010	10	1	VCVTT - Half-precision to single-precision variant	-
0	010	11	0	VCVTB - Half-precision to double-precision variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
o1	opc2	size	o3		
0	010	11	1	VCVTT - Half-precision to double-precision variant	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB - Single-precision to half-precision variant	-
0	011	10	1	VCVTT - Single-precision to half-precision variant	-
0	011	11	0	VCVTB - Double-precision to half-precision variant	-
0	011	11	1	VCVTT - Double-precision to half-precision variant	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR - Half-precision scalar variant	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) - Half-precision scalar variant	FEAT_FP16
0	110	10	0	VRINTR - Single-precision scalar variant	-
0	110	10	1	VRINTZ (floating-point) - Single-precision scalar variant	-
0	110	11	0	VRINTR - Double-precision scalar variant	-
0	110	11	1	VRINTZ (floating-point) - Double-precision scalar variant	-
0	111	01	0	VRINTX (floating-point) - Half-precision scalar variant	FEAT_FP16
0	111	01	1	Unallocated.	-
0	111	10	0	VRINTX (floating-point) - Single-precision scalar variant	-
0	111	10	1	VCVT (between double-precision and single-precision) - Single-precision to double-precision variant	-
0	111	11	0	VRINTX (floating-point) - Double-precision scalar variant	-
0	111	11	1	VCVT (between double-precision and single-precision) - Double-precision to single-precision variant	-
1	000	01	x	VCVT (integer to floating-point, floating-point) - Half-precision scalar variant	FEAT_FP16
1	000	10	x	VCVT (integer to floating-point, floating-point) - Single-precision scalar variant	-
1	000	11	x	VCVT (integer to floating-point, floating-point) - Double-precision scalar variant	-
1	001	01	x	Unallocated.	-
1	001	10	x	Unallocated.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
o1	opc2	size	o3		
1	001	11	0	Unallocated.	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01	x	VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10	x	VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11	x	VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01	x	VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10	x	VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11	x	VCVT (between floating-point and fixed-point, floating-point)	-

F3.1.13.2 Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from Floating-point data-processing.

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 0	1	D	1 1	imm4H	Vd	1 0	size	(0)	0	(0)	0	imm4L

Table F3-53

Decode fields	Instruction page	Feature
size		
00	Unallocated.	-
01	VMOV (immediate) - Half-precision scalar variant	FEAT_FP16
10	VMOV (immediate) - Single-precision scalar variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields size	Instruction page	Feature
11	VMOV (immediate) - Double-precision scalar variant	-

F3.1.13.3 Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

15 14 13 12	11 10 9 8	7	6	5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 0	1 1 1 0	o0	D	o1	Vn	Vd	1 0	size	N	o2	M	0	Vm

Table F3-54

Decode fields o0:o1	size	o2	Instruction page	Feature
!= 111	00	x	Unallocated.	-
000	01	0	VMLA (floating-point) - Half-precision scalar variant	FEAT_FP16
000	01	1	VMLS (floating-point) - Half-precision scalar variant	FEAT_FP16
000	10	0	VMLA (floating-point) - Single-precision scalar variant	-
000	10	1	VMLS (floating-point) - Single-precision scalar variant	-
000	11	0	VMLA (floating-point) - Double-precision scalar variant	-
000	11	1	VMLS (floating-point) - Double-precision scalar variant	-
001	01	0	VNMLS - Half-precision scalar variant	FEAT_FP16
001	01	1	VNMLA - Half-precision scalar variant	FEAT_FP16
001	10	0	VNMLS - Single-precision scalar variant	-
001	10	1	VNMLA - Single-precision scalar variant	-
001	11	0	VNMLS - Double-precision scalar variant	-
001	11	1	VNMLA - Double-precision scalar variant	-
010	01	0	VMUL (floating-point) - Half-precision scalar variant	FEAT_FP16
010	01	1	VNMUL - Half-precision scalar variant	FEAT_FP16
010	10	0	VMUL (floating-point) - Single-precision scalar variant	-
010	10	1	VNMUL - Single-precision scalar variant	-
010	11	0	VMUL (floating-point) - Double-precision scalar variant	-
010	11	1	VNMUL - Double-precision scalar variant	-
011	01	0	VADD (floating-point) - Half-precision scalar variant	FEAT_FP16
011	01	1	VSUB (floating-point) - Half-precision scalar variant	FEAT_FP16
011	10	0	VADD (floating-point) - Single-precision scalar variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields			Instruction page	Feature
o0:o1	size	o2		
011	10	1	VSUB (floating-point) - Single-precision scalar variant	-
011	11	0	VADD (floating-point) - Double-precision scalar variant	-
011	11	1	VSUB (floating-point) - Double-precision scalar variant	-
100	01	0	VDIV - Half-precision scalar variant	FEAT_FP16
100	10	0	VDIV - Single-precision scalar variant	-
100	11	0	VDIV - Double-precision scalar variant	-
101	01	0	VFNMS - Half-precision scalar variant	FEAT_FP16
101	01	1	VFNMA - Half-precision scalar variant	FEAT_FP16
101	10	0	VFNMS - Single-precision scalar variant	-
101	10	1	VFNMA - Single-precision scalar variant	-
101	11	0	VFNMS - Double-precision scalar variant	-
101	11	1	VFNMA - Double-precision scalar variant	-
110	01	0	VFMA - Half-precision scalar variant	FEAT_FP16
110	01	1	VFMS - Half-precision scalar variant	FEAT_FP16
110	10	0	VFMA - Single-precision scalar variant	-
110	10	1	VFMS - Single-precision scalar variant	-
110	11	0	VFMA - Double-precision scalar variant	-
110	11	1	VFMS - Double-precision scalar variant	-

F3.1.14 Additional Advanced SIMD and floating-point instructions

This section describes the encoding of the Additional Advanced SIMD and floating-point instructions group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point.

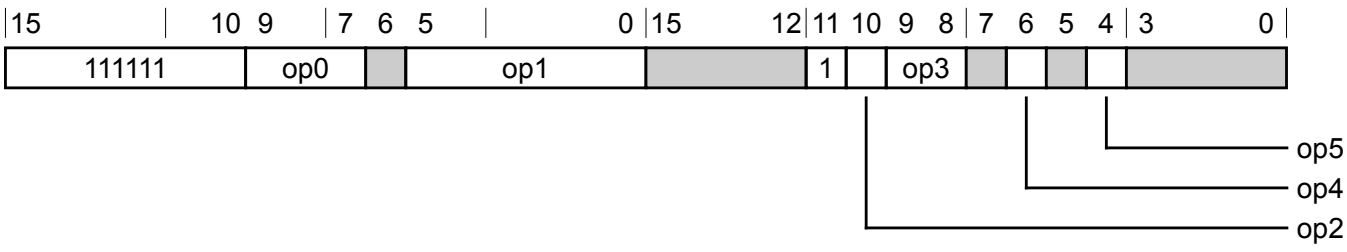

This decode also imposes the constraint:

- `op0<2:1> != 11`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-55 Encoding table for the Additional Advanced SIMD and floating-point instructions group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0xx	xxxxxx	x	0x	x	x	Advanced SIMD three registers of the same length extension
100	xxxxxx	0	!= 00	0	0	Floating-point conditional select
101	00xxxx	0	!= 00	x	0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x	xxxxxx	0	00	x	x	Advanced SIMD and floating-point multiply with accumulate
10x	xxxxxx	1	0x	x	x	Advanced SIMD and floating-point dot product

F3.1.14.1 Advanced SIMD three registers of the same length extension

This section describes the encoding of the Advanced SIMD three registers of the same length extension instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

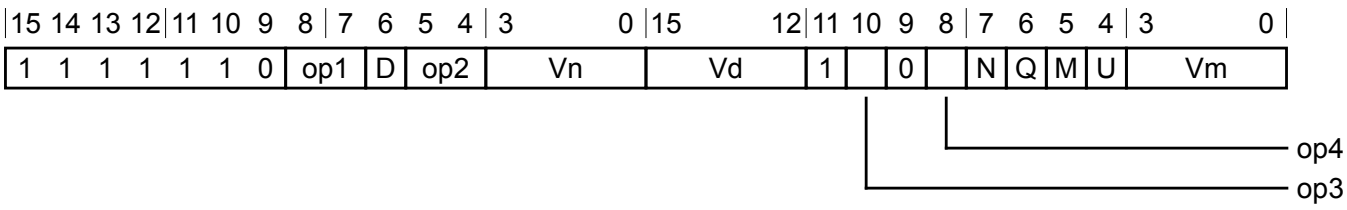

Table F3-56

Decode fields						Instruction page	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD - 64-bit SIMD vector variant	FEAT_FCMA
x1	0x	0	0	0	1	Unallocated.	-
x1	0x	0	0	1	0	VCADD - 128-bit SIMD vector variant	FEAT_FCMA
x1	0x	0	0	1	1	Unallocated.	-
00	0x	0	0	x	x	Unallocated.	-
00	0x	0	1	x	x	Unallocated.	-
00	00	1	0	0	0	Unallocated.	-
00	00	1	0	0	1	Unallocated.	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	Unallocated.	-
00	00	1	1	0	0	VDOT (vector) - 64-bit SIMD vector variant	FEAT_AA32BF16
00	00	1	1	0	1	Unallocated.	-
00	00	1	1	1	0	VDOT (vector) - 128-bit SIMD vector variant	FEAT_AA32BF16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields						Instruction page	Feature
op1	op2	op3	op4	Q	U		
00	00	1	1	1	1	Unallocated.	-
00	01	1	0	x	x	Unallocated.	-
00	01	1	1	x	x	Unallocated.	-
00	10	0	0	x	1	VFMAL (vector)	FEAT_FHM
00	10	0	1	x	x	Unallocated.	-
00	10	1	0	0	x	Unallocated.	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) - 64-bit SIMD vector variant	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) - 64-bit SIMD vector variant	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) - 128-bit SIMD vector variant	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) - 128-bit SIMD vector variant	FEAT_DotProd
00	11	0	0	x	1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1	x	x	Unallocated.	-
00	11	1	0	x	x	Unallocated.	-
00	11	1	1	x	x	Unallocated.	-
01	10	0	0	x	1	VFMSL (vector)	FEAT_FHM
01	10	0	1	x	x	Unallocated.	-
01	10	1	0	0	x	Unallocated.	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	Unallocated.	-
01	10	1	1	0	0	VUSDOT (vector) - 64-bit SIMD vector variant	FEAT_AA32I8MM
01	10	1	1	x	1	Unallocated.	-
01	10	1	1	1	0	VUSDOT (vector) - 128-bit SIMD vector variant	FEAT_AA32I8MM
01	11	0	1	x	x	Unallocated.	-
01	11	1	0	x	x	Unallocated.	-
01	11	1	1	x	x	Unallocated.	-
xx	1x	0	0	x	0	VCMLA	FEAT_FCMA
10	11	0	1	x	x	Unallocated.	-
10	11	1	0	x	x	Unallocated.	-
10	11	1	1	x	x	Unallocated.	-
11	11	0	1	x	x	Unallocated.	-
11	11	1	0	x	x	Unallocated.	-
11	11	1	1	x	x	Unallocated.	-

F3.1.14.2 Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0 D cc	Vn	Vd	1 0	!=00	N	0	M	0	Vm
						size					

Table F3-57

Decode fields	Instruction page	Feature
size		
01	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, half-precision scalar variant	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, single-precision scalar variant	-
11	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, double-precision scalar variant	-

F3.1.14.3 Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1 D 0 0	Vn	Vd	1 0	!=00	N	op	M	0	Vm
						size					

Table F3-58

Decode fields		Instruction page	Feature
size	op		
01	0	VMAXNM - Half-precision scalar variant	FEAT_FP16
01	1	VMINNM - Half-precision scalar variant	FEAT_FP16
10	0	VMAXNM - Single-precision scalar variant	-
10	1	VMINNM - Single-precision scalar variant	-
11	0	VMAXNM - Double-precision scalar variant	-
11	1	VMINNM - Double-precision scalar variant	-

F3.1.14.4 Floating-point extraction and insertion

This section describes the encoding of the Floating-point extraction and insertion instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 0 1	D 1 1 0	0 0 0	Vd	1 0	!=00 op 1	M 0 Vm
						size	

Table F3-59

Decode fields		Instruction page	Feature
size	op		
01	x	Unallocated.	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11	x	Unallocated.	-

F3.1.14.5 Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 1 1 0 1	D 1 1 1	o1 RM	Vd	1 0	!=00 op 1	M 0 Vm
						size	

Table F3-60

Decode fields				Instruction page	Feature
o1	RM	size	op		
0	xx	!= 00	1	Unallocated.	-
0	00	01	0	VRINTA (floating-point) - Half-precision scalar variant	FEAT_FP16
0	00	10	0	VRINTA (floating-point) - Single-precision scalar variant	-
0	00	11	0	VRINTA (floating-point) - Double-precision scalar variant	-
0	01	01	0	VRINTN (floating-point) - Half-precision scalar variant	FEAT_FP16
0	01	10	0	VRINTN (floating-point) - Single-precision scalar variant	-
0	01	11	0	VRINTN (floating-point) - Double-precision scalar variant	-
0	10	01	0	VRINTP (floating-point) - Half-precision scalar variant	FEAT_FP16
0	10	10	0	VRINTP (floating-point) - Single-precision scalar variant	-
0	10	11	0	VRINTP (floating-point) - Double-precision scalar variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
o1	RM	size	op		
0	11	01	0	VRINTM (floating-point) - Half-precision scalar variant	FEAT_FP16
0	11	10	0	VRINTM (floating-point) - Single-precision scalar variant	-
0	11	11	0	VRINTM (floating-point) - Double-precision scalar variant	-
1	00	01	x	VCVTA (floating-point) - Half-precision scalar variant	FEAT_FP16
1	00	10	x	VCVTA (floating-point) - Single-precision scalar variant	-
1	00	11	x	VCVTA (floating-point) - Double-precision scalar variant	-
1	01	01	x	VCVTN (floating-point) - Half-precision scalar variant	FEAT_FP16
1	01	10	x	VCVTN (floating-point) - Single-precision scalar variant	-
1	01	11	x	VCVTN (floating-point) - Double-precision scalar variant	-
1	10	01	x	VCVTP (floating-point) - Half-precision scalar variant	FEAT_FP16
1	10	10	x	VCVTP (floating-point) - Single-precision scalar variant	-
1	10	11	x	VCVTP (floating-point) - Double-precision scalar variant	-
1	11	01	x	VCVTM (floating-point) - Half-precision scalar variant	FEAT_FP16
1	11	10	x	VCVTM (floating-point) - Single-precision scalar variant	-
1	11	11	x	VCVTM (floating-point) - Double-precision scalar variant	-

F3.1.14.6 Advanced SIMD and floating-point multiply with accumulate

This section describes the encoding of the Advanced SIMD and floating-point multiply with accumulate instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

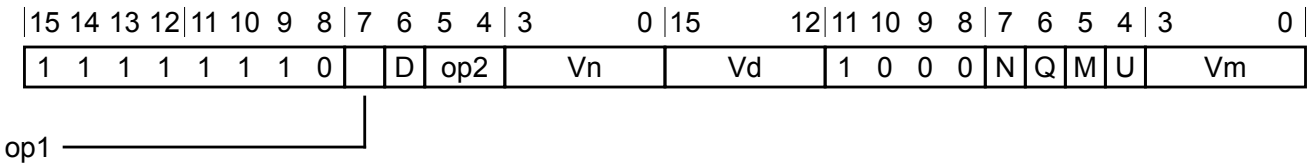

Table F3-61

Decode fields				Instruction page	Feature
op1	op2	Q	U		
0	xx	x	0	VCMLA (by element) - 128-bit SIMD vector of half-precision floating-point variant	FEAT_FCMA
0	00	x	1	VFMAL (by scalar)	FEAT_FHM
0	01	x	1	VFMSL (by scalar)	FEAT_FHM
0	10	x	1	Unallocated.	-
0	11	x	1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1	xx	0	0	VCMLA (by element) - 64-bit SIMD vector of single-precision floating-point variant	FEAT_FCMA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
op1	op2	Q	U		
1	xx	x	1	Unallocated.	-
1	xx	1	0	VCMLA (by element) - 128-bit SIMD vector of single-precision floating-point variant	FEAT_FCMA

F3.1.14.7 Advanced SIMD and floating-point dot product

This section describes the encoding of the Advanced SIMD and floating-point dot product instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

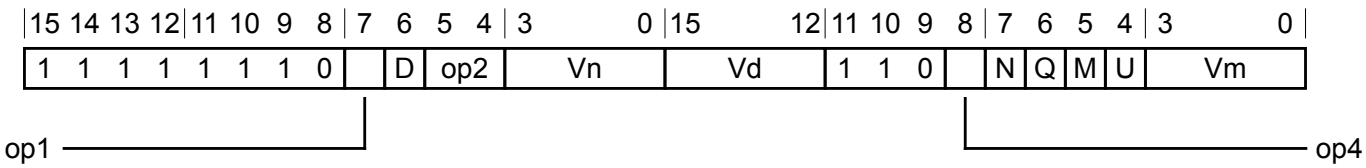

Table F3-62

Decode fields					Instruction page	Feature
op1	op2	op4	Q	U		
0	00	0	x	x	Unallocated.	-
0	00	1	0	0	VDOT (by element) - 64-bit SIMD vector variant	FEAT_AA32BF16
0	00	1	x	1	Unallocated.	-
0	00	1	1	0	VDOT (by element) - 128-bit SIMD vector variant	FEAT_AA32BF16
0	01	0	x	x	Unallocated.	-
0	10	0	x	x	Unallocated.	-
0	10	1	0	0	VSDOT (by element) - 64-bit SIMD vector variant	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) - 64-bit SIMD vector variant	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) - 128-bit SIMD vector variant	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) - 128-bit SIMD vector variant	FEAT_DotProd
0	11	x	x	x	Unallocated.	-
1	xx	0	x	x	Unallocated.	-
1	00	1	0	0	VUSDOT (by element) - 64-bit SIMD vector variant	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) - 64-bit SIMD vector variant	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) - 128-bit SIMD vector variant	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) - 128-bit SIMD vector variant	FEAT_AA32I8MM
1	01	1	x	x	Unallocated.	-
1	1x	1	x	x	Unallocated.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1.15 Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

This section describes the encoding of the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group. The encodings in this section are decoded from 32-bit.

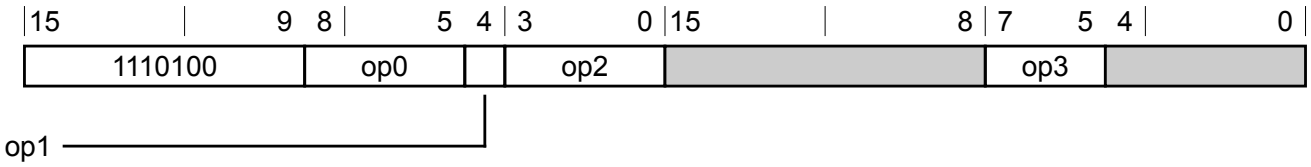

This decode also imposes the constraint:

- `op0<1> == 1`.

Table F3-63 Encoding table for the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0010	x	xxxx	xxx	Load/store exclusive
0110	0	xxxx	000	Unallocated.
0110	1	xxxx	000	TBB, TBH
0110	x	xxxx	01x	Load/store exclusive byte/half/dual
0110	x	xxxx	1xx	Load-acquire / Store-release
0x11	x	!= 1111	xxx	Load/store dual (immediate, post-indexed)
1x10	x	!= 1111	xxx	Load/store dual (immediate)
1x11	x	!= 1111	xxx	Load/store dual (immediate, pre-indexed)
!= 0xx0	x	1111	xxx	LDRD (literal)

F3.1.15.1 Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

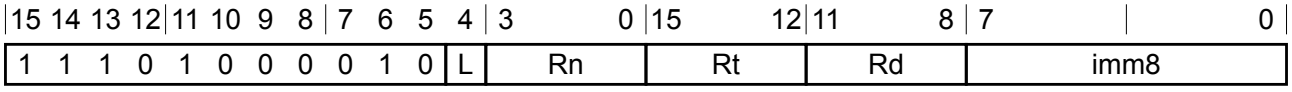

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-64

Decode fields	Instruction page
L	
0	STREX
1	LDREX

F3.1.15.2 Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 0	1 0 0 0	1 1 0	L	Rn	Rt	Rt2	0 1	sz	Rd

Table F3-65

Decode fields		Instruction page
L	**sz**	
0	00	STREXB
0	01	STREXH
0	10	Unallocated.
0	11	STREXD
1	00	LDREXB
1	01	LDREXH
1	10	Unallocated.
1	11	LDREXD

F3.1.15.3 Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 0	1 0 0 0	1 1 0	L	Rn	Rt	Rt2	1 op	sz	Rd

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-66

Decode fields			Instruction page
L	**op**	**sz**	
0	0	00	STLB
0	0	01	STLH
0	0	10	STL
0	0	11	Unallocated.
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA
1	0	11	Unallocated.
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

F3.1.15.4 Load/store dual (immediate, post-indexed)

This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

15 14 13 12	11 10 9 8	7	6 5	4	3 0	15 12	11 8	7 0
1 1 1 0	1 0 0 0	U	1 1	L	!=1111	Rt	Rt2	imm8
					Rn			

Table F3-67

Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

F3.1.15.5 Load/store dual (immediate)

This section describes the encoding of the Load/store dual (immediate) instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7	6 5 4	3 0	15 12	11 8	7 0
1 1 1 0	1 0 0 1	U	1 0 L	!=1111 (Rn)	Rt	Rt2	imm8

Table F3-68

Decode fields L	Instruction page
0	STRD (immediate)
1	LDRD (immediate)

F3.1.15.6 Load/store dual (immediate, pre-indexed)

This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

15 14 13 12	11 10 9 8	7	6 5 4	3 0	15 12	11 8	7 0
1 1 1 0	1 0 0 1	U	1 1 L	!=1111 (Rn)	Rt	Rt2	imm8

Table F3-69

Decode fields L	Instruction page
0	STRD (immediate)
1	LDRD (immediate)

F3.1.16 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. The encodings in this section are decoded from 32-bit.

15	11	10	9 6	5 4	3 0	15	14 12	11	10 8	7 6	5	4 0
11110			op1	op2		1	op3		op4			

op0 — op5

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-70 Encoding table for the Branches and miscellaneous control group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0	xxx	0	MSR (register)
0	1110	0x	0x0	xxx	1	MSR (Banked register)
0	1110	10	0x0	000	x	Hints
0	1110	10	0x0	!= 000	x	Change processor state
0	1110	11	0x0	xxx	x	Miscellaneous system
0	1111	00	0x0	xxx	x	BXJ
0	1111	01	0x0	xxx	x	Exception return
0	1111	1x	0x0	xxx	0	MRS
0	1111	1x	0x0	xxx	1	MRS (Banked register)
1	1110	00	000	xxx	x	DCPS
1	1110	00	010	xxx	x	Unallocated.
1	1110	01	0x0	xxx	x	Unallocated.
1	1110	1x	0x0	xxx	x	Unallocated.
1	1111	0x	0x0	xxx	x	Unallocated.
1	1111	1x	0x0	xxx	x	Exception generation
x	!= 111x	x	0x0	xxx	x	B - T3 variant
x	xxxx	x	0x1	xxx	x	B - T4 variant
x	xxxx	x	1x0	xxx	x	BL, BLX (immediate) - T2 variant
x	xxxx	x	1x1	xxx	x	BL, BLX (immediate) - T1 variant

F3.1.16.1 Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from Branches and miscellaneous control.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 4	3 0
1 1 1 1	0 0 1 1	1 0 1 0	(1)(1)(1)(1)	1 0 (0) 0	(0) 0 0 0	hint	option

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-71

Decode fields		Instruction page	Feature
hint	**option**		
0000	0000	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP.	-
0000	1xxx	Reserved hint, behaves as NOP.	-
0001	0000	ESB	FEAT_RAS
0001	0001	Reserved hint, behaves as NOP.	-
0001	0010	TSB	FEAT_TRF
0001	0011	Reserved hint, behaves as NOP.	-
0001	0100	CSDB	-
0001	0101	Reserved hint, behaves as NOP.	-
0001	0110	CLRBHB	FEAT_CLRBHB
0001	0111	Reserved hint, behaves as NOP.	-
0001	1xxx	Reserved hint, behaves as NOP.	-
001x	xxxx	Reserved hint, behaves as NOP.	-
01xx	xxxx	Reserved hint, behaves as NOP.	-
10xx	xxxx	Reserved hint, behaves as NOP.	-
110x	xxxx	Reserved hint, behaves as NOP.	-
1110	xxxx	Reserved hint, behaves as NOP.	-
1111	xxxx	DBG	-

F3.1.16.2 Change processor state

This section describes the encoding of the Change processor state instruction class. The encodings in this section are decoded from Branches and miscellaneous control.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	0
1 1 1 1	0 0 1 1	1 0 1 0	(1) (1) (1) (1)	1 0 (0) 0	(0) imod M	A I F	mode

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-72

Decode fields		Instruction page
imod	**M**	
00	1	CPS, CPSID, CPSIE - Change mode variant
01	x	Unallocated.
10	x	CPS, CPSID, CPSIE - Interrupt enable and change mode variant
11	x	CPS, CPSID, CPSIE - Interrupt disable and change mode variant

F3.1.16.3 Miscellaneous system

This section describes the encoding of the Miscellaneous system instruction class. The encodings in this section are decoded from Branches and miscellaneous control.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 4	3 0
1 1 1 1	0 0 1 1	1 0 1 1	(1)(1)(1)(1)	1 0 (0) 0	(1)(1)(1)(1)	opc	option

Table F3-73

Decode fields		Instruction page	Feature
opc	**option**		
000x	xxxx	Unallocated.	-
0010	xxxx	CLREX	-
0011	xxxx	Unallocated.	-
0100	!= 0x00	DSB	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101	xxxx	DMB	-
0110	xxxx	ISB	-
0111	xxxx	SB	FEAT_SB
1xxx	xxxx	Unallocated.	-

F3.1.16.4 Exception return

This section describes the encoding of the Exception return instruction class. The encodings in this section are decoded from Branches and miscellaneous control.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13 12	11 10 9 8	7 0
1 1 1 1	0 0 1 1	1 1 0 1	Rn	1 0 (0) 0	(1)(1)(1)(1)	imm8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-74

Decode fields Rn:imm8	Instruction page
`!= 111000000000`	SUB, SUBS (immediate)
`111000000000`	ERET

F3.1.16.5 DCPS

This section describes the encoding of the DCPS instruction class. The encodings in this section are decoded from Branches and miscellaneous control.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	imm4	1	0	0	0	imm10	opt	

Table F3-75

Decode fields imm4	imm10	opt	Instruction page
`!= 1111`	`xxxxxxxxxx`	`xx`	Unallocated.
`1111`	`!= 0000000000`	`xx`	Unallocated.
`1111`	`0000000000`	`00`	Unallocated.
`1111`	`0000000000`	`01`	DCPS1
`1111`	`0000000000`	`10`	DCPS2
`1111`	`0000000000`	`11`	DCPS3

F3.1.16.6 Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from Branches and miscellaneous control.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 0
1	1	1	1	0	1	1	1	1	1	1	o1	imm4	1	0	o2	0	imm12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-76

Decode fields		Instruction page
o1	**o2**	
0	0	HVC
0	1	Unallocated.
1	0	SMC
1	1	UDF

F3.1.17 Data-processing (plain binary immediate)

This section describes the encoding of the Data-processing (plain binary immediate) group. The encodings in this section are decoded from 32-bit.

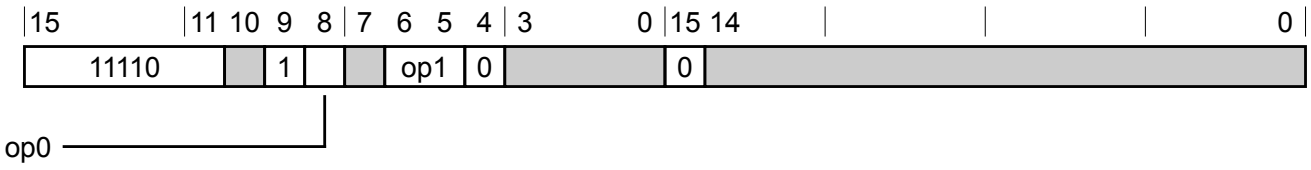

Table F3-77 Encoding table for the Data-processing (plain binary immediate) group

Decode fields		Decode group or instruction page
op0	**op1**	
0	0x	Data-processing (simple immediate)
0	10	Move Wide (16-bit immediate)
0	11	Unallocated.
1	xx	Saturate, Bitfield

F3.1.17.1 Data-processing (simple immediate)

This section describes the encoding of the Data-processing (simple immediate) instruction class. The encodings in this section are decoded from Data-processing (plain binary immediate).

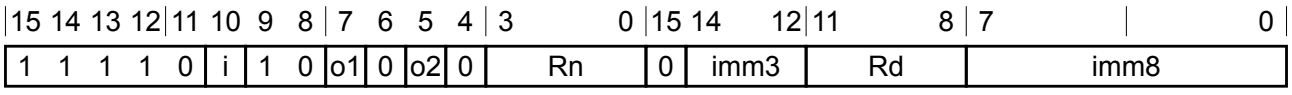

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-78

Decode fields			Instruction page
o1	**o2**	**Rn**	
0	0	!= 11x1	ADD, ADDS (immediate)
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR - T3
0	1	xxxx	Unallocated.
1	0	xxxx	Unallocated.
1	1	!= 11x1	SUB, SUBS (immediate)
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR - T2

F3.1.17.2 Move Wide (16-bit immediate)

This section describes the encoding of the Move Wide (16-bit immediate) instruction class. The encodings in this section are decoded from Data-processing (plain binary immediate).

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 12	11 8	7 0
1 1 1 1 0	i 1 0	o1 1 0 0	imm4	0 imm3	Rd	imm8

Table F3-79

Decode fields	Instruction page
o1	
0	MOV, MOVS (immediate)
1	MOVT

F3.1.17.3 Saturate, Bitfield

This section describes the encoding of the Saturate, Bitfield instruction class. The encodings in this section are decoded from Data-processing (plain binary immediate).

15 14 13 12	11 10 9 8	7 5 4	3 0	15 14 12	11 8	7 6 5 4	0
1 1 1 1 0	(0) 1 1	op1 0	Rn	0 imm3	Rd	imm2 (0)	widthm1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-80

Decode fields			Instruction page
op1	Rn	imm3:imm2	
000	xxxx	xxxxx	SSAT - Logical shift left variant
001	xxxx	!= 00000	SSAT - Arithmetic shift right variant
001	xxxx	00000	SSAT16
010	xxxx	xxxxx	SBFX
011	!= 1111	xxxxx	BFI
011	1111	xxxxx	BFC
100	xxxx	xxxxx	USAT - Logical shift left variant
101	xxxx	!= 00000	USAT - Arithmetic shift right variant
101	xxxx	00000	USAT16
110	xxxx	xxxxx	UBFX
111	xxxx	xxxxx	Unallocated.

F3.1.18 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from 32-bit.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding

Table F3-81 Encoding table for the Advanced SIMD element or structure load/store group

Decode fields		Decode group or instruction page
op0	op1	
0	xx	Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

F3.1.18.1 Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 0	D L 0	Rn	Vd	itype	size	align	Rm

Table F3-82

Decode fields			Instruction page
L	**itype**	**Rm**	
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields			Instruction page
L	itype	Rm	
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
x	1011	xxxx	Unallocated.
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
x	11xx	xxxx	Unallocated.
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

F3.1.18.2 Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 1	D L 0	Rn	Vd	1 1 N	size T a	Rm

Table F3-83

Decode fields				Instruction page
L	N	a	Rm	
0	x	x	xxxx	Unallocated.
1	00	x	!= 11x1	VLD1 (single element to all lanes)
1	00	x	1101	VLD1 (single element to all lanes)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
L	N	a	Rm	
1	00	x	1111	VLD1 (single element to all lanes)
1	01	x	!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01	x	1101	VLD2 (single 2-element structure to all lanes)
1	01	x	1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1	xxxx	Unallocated.
1	11	x	!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11	x	1101	VLD4 (single 4-element structure to all lanes)
1	11	x	1111	VLD4 (single 4-element structure to all lanes)

F3.1.18.3 Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 4	3 0
1 1 1 1	1 0 0 1 1	D L 0	Rn	Vd	!=11 size	N	index_align	Rm

Table F3-84

Decode fields				Instruction page
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
L	size	N	Rm	
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
L	size	N	Rm	
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

F3.1.19 Load/store single

This section describes the encoding of the Load/store single group. The encodings in this section are decoded from 32-bit.

This decode also imposes the constraint:

- `op0<1>:op1 != 10`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-85 Encoding table for the Load/store single group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
00	x	!= 1111	000000	Load/store, unsigned (register offset)
00	x	!= 1111	000001	Unallocated.
00	x	!= 1111	00001x	Unallocated.
00	x	!= 1111	0001xx	Unallocated.
00	x	!= 1111	001xxx	Unallocated.
00	x	!= 1111	01xxxx	Unallocated.
00	x	!= 1111	10x0xx	Unallocated.
00	x	!= 1111	10x1xx	Load/store, unsigned (immediate, post-indexed)
00	x	!= 1111	1100xx	Load/store, unsigned (negative immediate)
00	x	!= 1111	1110xx	Load/store, unsigned (unprivileged)
00	x	!= 1111	11x1xx	Load/store, unsigned (immediate, pre-indexed)
01	x	!= 1111	xxxxxx	Load/store, unsigned (positive immediate)
0x	x	1111	xxxxxx	Load, unsigned (literal)
10	1	!= 1111	000000	Load/store, signed (register offset)
10	1	!= 1111	000001	Unallocated.
10	1	!= 1111	00001x	Unallocated.
10	1	!= 1111	0001xx	Unallocated.
10	1	!= 1111	001xxx	Unallocated.
10	1	!= 1111	01xxxx	Unallocated.
10	1	!= 1111	10x0xx	Unallocated.
10	1	!= 1111	10x1xx	Load/store, signed (immediate, post-indexed)
10	1	!= 1111	1100xx	Load/store, signed (negative immediate)
10	1	!= 1111	1110xx	Load/store, signed (unprivileged)
10	1	!= 1111	11x1xx	Load/store, signed (immediate, pre-indexed)
11	1	!= 1111	xxxxxx	Load/store, signed (positive immediate)
1x	1	1111	xxxxxx	Load, signed (literal)

F3.1.19.1 Load/store, unsigned (register offset)

This section describes the encoding of the Load/store, unsigned (register offset) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 0 0	size	L	!=1111	Rt	0 0 0 0 0 0	imm2	Rm
				Rn				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-86

Decode fields			Instruction page
size	**L**	**Rt**	
00	0	xxxx	STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD, PLDW (register) - Preload read variant
01	0	xxxx	STRH (register)
01	1	!= 1111	LDRH (register)
01	1	1111	PLD, PLDW (register) - Preload write variant
10	0	xxxx	STR (register)
10	1	xxxx	LDR (register)
11	x	xxxx	Unallocated.

F3.1.19.2 Load/store, unsigned (immediate, post-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, post-indexed) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9	8	7 0
1 1 1 1	1 0 0 0	0	size	L	!=1111	Rt	1 0	U	1	imm8
					Rn					

Table F3-87

Decode fields		Instruction page
size	**L**	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	x	Unallocated.

F3.1.19.3 Load/store, unsigned (negative immediate)

This section describes the encoding of the Load/store, unsigned (negative immediate) instruction class. The encodings in this section are decoded from Load/store single.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0 0	0 size	L	!=1111 Rn	Rt	1 1 0 0	imm8

Table F3-88

Decode fields			Instruction page
size	**L**	**Rt**	
00	0	xxxx	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - Preload read variant
01	0	xxxx	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - Preload write variant
10	0	xxxx	STR (immediate)
10	1	xxxx	LDR (immediate)
11	x	xxxx	Unallocated.

F3.1.19.4 Load/store, unsigned (unprivileged)

This section describes the encoding of the Load/store, unsigned (unprivileged) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0 0	0 size	L	!=1111 Rn	Rt	1 1 1 0	imm8

Table F3-89

Decode fields		Instruction page
size	**L**	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page
size	L	
10	1	LDRT
11	x	Unallocated.

F3.1.19.5 Load/store, unsigned (immediate, pre-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9	8	7 0
1 1 1 1	1 0 0 0	0 size	L	!=1111	Rt	1 1	U	1	imm8
			Rn					

Table F3-90

Decode fields		Instruction page
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	x	Unallocated.

F3.1.19.6 Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 0
1 1 1 1	1 0 0 0	1 size	L	!=1111	Rt	imm12
			Rn		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-91

Decode fields			Instruction page
size	**L**	**Rt**	
00	0	xxxx	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - Preload read variant
01	0	xxxx	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - Preload write variant
10	0	xxxx	STR (immediate)
10	1	xxxx	LDR (immediate)

F3.1.19.7 Load, unsigned (literal)

This section describes the encoding of the Load, unsigned (literal) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 12	11 0
1 1 1 1	1 0 0 0	U	size L	1 1 1 1	Rt	imm12

Table F3-92

Decode fields			Instruction page
size	**L**	**Rt**	
0x	1	1111	PLD (literal)
00	1	!= 1111	LDRB (literal)
01	1	!= 1111	LDRH (literal)
10	1	xxxx	LDR (literal)
11	x	xxxx	Unallocated.

F3.1.19.8 Load/store, signed (register offset)

This section describes the encoding of the Load/store, signed (register offset) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	1 0 0 1 0	size 1	!=1111	Rt	0 0 0 0	0 0 imm2	Rm
			Rn				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-93

Decode fields		Instruction page
size	**Rt**	
`00`	`!= 1111`	LDRSB (register)
`00`	`1111`	PLI (register)
`01`	`!= 1111`	LDRSH (register)
`01`	`1111`	Reserved hint, behaves as NOP.
`1x`	`xxxx`	Unallocated.

F3.1.19.9 Load/store, signed (immediate, post-indexed)

This section describes the encoding of the Load/store, signed (immediate, post-indexed) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10	9	8	7 0
1 1 1 1	1 0 0 1 0	size	1	!=1111	Rt	1 0	U	1	imm8
			Rn					

Table F3-94

Decode fields	Instruction page
size	
`00`	LDRSB (immediate)
`01`	LDRSH (immediate)
`1x`	Unallocated.

F3.1.19.10 Load/store, signed (negative immediate)

This section describes the encoding of the Load/store, signed (negative immediate) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0 1 0	size	1	!=1111	Rt	1 1 0 0	imm8
			Rn			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-95

Decode fields		Instruction page
size	**Rt**	
`00`	`!= 1111`	LDRSB (immediate)
`00`	`1111`	PLI (immediate, literal)
`01`	`!= 1111`	LDRSH (immediate)
`01`	`1111`	Reserved hint, behaves as NOP.
`1x`	`xxxx`	Unallocated.

F3.1.19.11 Load/store, signed (unprivileged)

This section describes the encoding of the Load/store, signed (unprivileged) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0 1 0	size	1	!=1111	Rt	1 1 1 0	imm8
				Rn		

Table F3-96

Decode fields	Instruction page
size	
`00`	LDRSBT
`01`	LDRSHT
`1x`	Unallocated.

F3.1.19.12 Load/store, signed (immediate, pre-indexed)

This section describes the encoding of the Load/store, signed (immediate, pre-indexed) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0 1 0	size	1	!=1111	Rt	1 1 U 1	imm8
				Rn		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-97

Decode fields size	Instruction page
`00`	LDRSB (immediate)
`01`	LDRSH (immediate)
`1x`	Unallocated.

F3.1.19.13 Load/store, signed (positive immediate)

This section describes the encoding of the Load/store, signed (positive immediate) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 12	11 0
1 1 1 1	1 0 0 1 1	size	1	!=1111	Rt	imm12
				Rn	

Table F3-98

Decode fields size	Rt	Instruction page
`00`	`!= 1111`	LDRSB (immediate)
`00`	`1111`	PLI (immediate, literal)
`01`	`!= 1111`	LDRSH (immediate)
`01`	`1111`	Reserved hint, behaves as NOP.

F3.1.19.14 Load, signed (literal)

This section describes the encoding of the Load, signed (literal) instruction class. The encodings in this section are decoded from Load/store single.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 12	11 0
1 1 1 1	1 0 0 1	U	size	1 1 1 1 1	Rt	imm12

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-99

Decode fields		Instruction page
size	**Rt**	
00	!= 1111	LDRSB (literal)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (literal)
01	1111	Reserved hint, behaves as NOP.
1x	xxxx	Unallocated.

F3.1.20 Data-processing (register)

This section describes the encoding of the Data-processing (register) group. The encodings in this section are decoded from 32-bit.

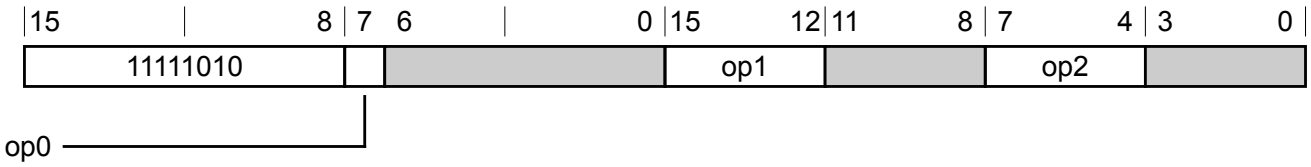

Table F3-100 Encoding table for the Data-processing (register) group

Decode fields			Decode group or instruction page
op0	**op1**	**op2**	
0	1111	0000	MOV, MOVS (register-shifted register) - Flag setting variant
0	1111	0001	Unallocated.
0	1111	001x	Unallocated.
0	1111	01xx	Unallocated.
0	1111	1xxx	Register extends
1	1111	0xxx	Parallel add-subtract
1	1111	10xx	Data-processing (two source registers)
1	1111	11xx	Unallocated.
x	!= 1111	xxxx	Unallocated.

F3.1.20.1 Register extends

This section describes the encoding of the Register extends instruction class. The encodings in this section are decoded from Data-processing (register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 5 4	3 0	15 14 13 12	11 8	7	6	5 4	3 0
1 1 1 1	1 0 1 0	0 op1	U Rn	1 1 1 1	Rd	1	(0)	rotate	Rm

Table F3-101

Decode fields			Instruction page
op1	U	Rn	
00	0	!= 1111	SXTAH
00	0	1111	SXTH
00	1	!= 1111	UXTAH
00	1	1111	UXTH
01	0	!= 1111	SXTAB16
01	0	1111	SXTB16
01	1	!= 1111	UXTAB16
01	1	1111	UXTB16
10	0	!= 1111	SXTAB
10	0	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11	x	xxxx	Unallocated.

F3.1.20.2 Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. The encodings in this section are decoded from Data-processing (register).

15 14 13 12	11 10 9 8	7 6 4	3 0	15 14 13 12	11 8	7	6	5	4	3 0
1 1 1 1	1 0 1 0	1 op1	Rn	1 1 1 1	Rd	0	U	H	S	Rm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F3-102

Decode fields				Instruction page
op1	U	H	S	
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	Unallocated.
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
000	1	1	1	Unallocated.
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	Unallocated.
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	Unallocated.
010	0	0	0	SASX
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	Unallocated.
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	Unallocated.
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	0	1	1	Unallocated.
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	Unallocated.
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	Unallocated.
101	1	0	0	USUB16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
op1	U	H	S	
101	1	0	1	UQSUB16
101	1	1	0	UHSUB16
101	1	1	1	Unallocated.
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	Unallocated.
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	Unallocated.
111	x	x	x	Unallocated.

F3.1.20.3 Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. The encodings in this section are decoded from Data-processing (register).

15 14 13 12	11 10 9 8	7 6 4	3 0	15 14 13 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 0 1	op1	Rn	1 1 1 1	Rd	1 0	op2	Rm

Table F3-103

Decode fields		Instruction page	Feature
op1	op2		
000	00	QADD	-
000	01	QDADD	-
000	10	QSUB	-
000	11	QDSUB	-
001	00	REV	-
001	01	REV16	-
001	10	RBIT	-
001	11	REVSH	-
010	00	SEL	-
010	01	Unallocated.	-
010	1x	Unallocated.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page	Feature
op1	op2		
011	00	CLZ	-
011	01	Unallocated.	-
011	1x	Unallocated.	-
100	00	CRC32 - CRC32B variant	FEAT_CRC32
100	01	CRC32 - CRC32H variant	FEAT_CRC32
100	10	CRC32 - CRC32W variant	FEAT_CRC32
100	11	CONSTRAINED UNPREDICTABLE	-
101	00	CRC32C - CRC32CB variant	FEAT_CRC32
101	01	CRC32C - CRC32CH variant	FEAT_CRC32
101	10	CRC32C - CRC32CW variant	FEAT_CRC32
101	11	CONSTRAINED UNPREDICTABLE	-
11x	xx	Unallocated.	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.

F3.1.21 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. The encodings in this section are decoded from 32-bit.

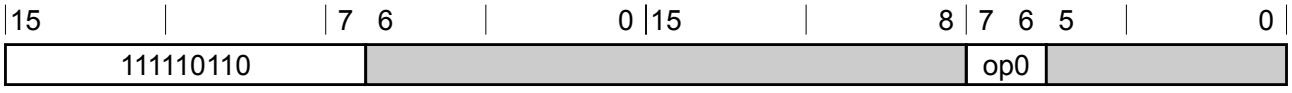

Table F3-104 Encoding table for the Multiply, multiply accumulate, and absolute difference group

Decode fields	Decode group or instruction page
op0	
00	Multiply and absolute difference
01	Unallocated.
1x	Unallocated.

F3.1.21.1 Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. The encodings in this section are decoded from Multiply, multiply accumulate, and absolute difference.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

15 14 13 12	11 10 9 8	7 6 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 1	0	op1	Rn	Ra	Rd	0 0	op2	Rm

Table F3-105

Decode fields			Instruction page
op1	Ra	op2	
000	!= 1111	00	MLA, MLAS
000	xxxx	01	MLS
000	xxxx	1x	Unallocated.
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT - SMLABB variant
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT - SMLABT variant
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT - SMLATB variant
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT - SMLATT variant
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT - SMULBB variant
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT - SMULBT variant
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT - SMULTB variant
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT - SMULTT variant
010	!= 1111	00	SMLAD, SMLADX - SMLAD variant
010	!= 1111	01	SMLAD, SMLADX - SMLADX variant
010	xxxx	1x	Unallocated.
010	1111	00	SMUAD, SMUADX - SMUAD variant
010	1111	01	SMUAD, SMUADX - SMUADX variant
011	!= 1111	00	SMLAWB, SMLAWT - SMLAWB variant
011	!= 1111	01	SMLAWB, SMLAWT - SMLAWT variant
011	xxxx	1x	Unallocated.
011	1111	00	SMULWB, SMULWT - SMULWB variant
011	1111	01	SMULWB, SMULWT - SMULWT variant
100	!= 1111	00	SMLSD, SMLSDX - SMLSD variant
100	!= 1111	01	SMLSD, SMLSDX - SMLSDX variant
100	xxxx	1x	Unallocated.
100	1111	00	SMUSD, SMUSDX - SMUSD variant
100	1111	01	SMUSD, SMUSDX - SMUSDX variant
101	!= 1111	00	SMMLA, SMMLAR - SMMLA variant
101	!= 1111	01	SMMLA, SMMLAR - SMMLAR variant
101	xxxx	1x	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields			Instruction page
op1	Ra	op2	
101	1111	00	SMMUL, SMMULR - SMMUL variant
101	1111	01	SMMUL, SMMULR - SMMULR variant
110	xxxx	00	SMMLS, SMMLSR - SMMLS variant
110	xxxx	01	SMMLS, SMMLSR - SMMLSR variant
110	xxxx	1x	Unallocated.
111	!= 1111	00	USADA8
111	xxxx	01	Unallocated.
111	xxxx	1x	Unallocated.
111	1111	00	USAD8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All T32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit T32 instruction decode structure. Table F3-106 shows these instruction groups, and where each group is decoded from the overall T32 decode structure:

Table F3-106 Advanced SIMD and floating-point instructions in the T32 decode structure

Advanced SIMD and floating-point instruction group	T32 decode is from
Advanced SIMD and System register load/store and 64-bit move	System register access, Advanced SIMD, and floating-point
Floating-point data-processing	System register access, Advanced SIMD, and floating-point
Advanced SIMD and System register 32-bit move	System register access, Advanced SIMD, and floating-point
Advanced SIMD data-processing	System register access, Advanced SIMD, and floating-point
Advanced SIMD element or structure load/store	32-bit

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F4
A32 Instruction Set Encoding

This chapter describes the encoding of the A32 instruction set. It contains the following sections:

- A32 instruction set encoding.
- About the A32 Advanced SIMD and floating-point instructions and their encoding.

In this chapter:

- In the decode tables, an entry of `x` for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1 A32 instruction set encoding

The A32 instruction stream is a sequence of 32-bit word-aligned words. Each instruction is a single word in the stream.

Most A32 instructions can be conditional, with a condition determined by bits[31:28] of the instruction, the `cond` field. For more information see The Condition code field in A32 instruction encodings. This applies to all instructions except those with the `cond` field equal to `0b111`.

The behavior of an attempt to execute an unallocated instruction is described in UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space.

For more information on A32 instruction encodings see About the T32 and A32 Instruction Descriptions.

The A32 instruction encoding is:

Table F4-1 Main encoding table for the A32 instruction set

Decode fields			Decode group or instruction page
cond	**op0**	**op1**	
`!= 1111`	`00x`	`x`	Data-processing and miscellaneous instructions
`!= 1111`	`010`	`x`	Load/Store Word, Unsigned Byte (immediate, literal)
`!= 1111`	`011`	`0`	Load/Store Word, Unsigned Byte (register)
`!= 1111`	`011`	`1`	Media instructions
`xxxx`	`10x`	`x`	Branch, branch with link, and block data transfer
`xxxx`	`11x`	`x`	System register access, Advanced SIMD, floating-point, and Supervisor call
`1111`	`0xx`	`x`	Unconditional instructions

F4.1.1 Data-processing and miscellaneous instructions

This section describes the encoding of the Data-processing and miscellaneous instructions group. The encodings in this section are decoded from A32 instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-2 Encoding table for the Data-processing and miscellaneous instructions group

Decode fields					Decode group or instruction page
op0	op1	op2	op3	op4	
0	xxxxx	1	!= 00	1	Extra load/store
0	0xxxx	1	00	1	Multiply and Accumulate
0	1xxxx	1	00	1	Synchronization primitives and Load-Acquire/Store-Release
0	10xx0	0	xx	x	Miscellaneous
0	10xx0	1	xx	0	Halfword Multiply and Accumulate
0	!= 10xx0	x	xx	0	Data-processing register (immediate shift)
0	!= 10xx0	0	xx	1	Data-processing register (register shift)
1	xxxxx	x	xx	x	Data-processing immediate

F4.1.1.1 Multiply and Accumulate

This section describes the encoding of the Multiply and Accumulate instruction class. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

31–28	27 26 25 24	23–21	20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111	0 0 0 0	opc	S	RdHi	RdLo	Rm	1 0 0 1	Rn
cond								

Table F4-3

Decode fields		Instruction page
opc	S	
000	x	MUL, MULS
001	x	MLA, MLAS
010	0	UMAAL
010	1	Unallocated.
011	0	MLS
011	1	Unallocated.
100	x	UMULL, UMULLS
101	x	UMLAL, UMLALS
110	x	SMULL, SMULLS
111	x	SMLAL, SMLALS

F4.1.1.2 Halfword Multiply and Accumulate

This section describes the encoding of the Halfword Multiply and Accumulate instruction class. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27 26 25 24	23 22 21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0 0 0 1 0	opc	0	Rd	Ra	Rm	1	M	N	0	Rn

Table F4-4

Decode fields			Instruction page
opc	M	N	
00	x	x	SMLABB, SMLABT, SMLATB, SMLATT
01	0	0	SMLAWB, SMLAWT - SMLAWB variant
01	0	1	SMULWB, SMULWT - SMULWB variant
01	1	0	SMLAWB, SMLAWT - SMLAWT variant
01	1	1	SMULWB, SMULWT - SMULWT variant
10	x	x	SMLALBB, SMLALBT, SMLALTB, SMLALTT
11	x	x	SMULBB, SMULBT, SMULTB, SMULTT

F4.1.2 Extra load/store

This section describes the encoding of the Extra load/store group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

31–28	27–25	24	23	22 21	8	7	6 5	4	3–0
!=1111	000					1	!=00	1	

op0

Table F4-5 Encoding table for the Extra load/store group

Decode fields	Decode group or instruction page
op0	
0	Load/Store Dual, Half, Signed Byte (register)
1	Load/Store Dual, Half, Signed Byte (immediate, literal)

F4.1.2.1 Load/Store Dual, Half, Signed Byte (register)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (register) instruction class. The encodings in this section are decoded from Extra load/store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6 5	4	3–0
!=1111	0	0	0	P	U	0	W	o1	Rn	Rt	(0)	(0)	(0)	(0)	1	!=00	1	Rm
cond																op2		

Table F4-6

Decode fields				Instruction page
P	**W**	**o1**	**op2**	
0	0	0	01	STRH (register) - Post-indexed variant
0	0	0	10	LDRD (register) - Post-indexed variant
0	0	0	11	STRD (register) - Post-indexed variant
0	0	1	01	LDRH (register) - Post-indexed variant
0	0	1	10	LDRSB (register) - Post-indexed variant
0	0	1	11	LDRSH (register) - Post-indexed variant
0	1	0	01	STRHT
0	1	0	10	Unallocated.
0	1	0	11	Unallocated.
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1	x	0	01	STRH (register) - Pre-indexed variant
1	x	0	10	LDRD (register) - Pre-indexed variant
1	x	0	11	STRD (register) - Pre-indexed variant
1	x	1	01	LDRH (register) - Pre-indexed variant
1	x	1	10	LDRSB (register) - Pre-indexed variant
1	x	1	11	LDRSH (register) - Pre-indexed variant

F4.1.2.2 Load/Store Dual, Half, Signed Byte (immediate, literal)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (immediate, literal) instruction class. The encodings in this section are decoded from Extra load/store.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111	0	0	0	P	U	1	W	o1	Rn	Rt	imm4H	1	!=00	1	imm4L
cond													op2		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-7

Decode fields				Instruction page
P:W	**o1**	**Rn**	**op2**	
xx	0	1111	10	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) - Post-indexed variant
00	0	xxxx	01	STRH (immediate) - Post-indexed variant
00	0	xxxx	11	STRD (immediate) - Post-indexed variant
00	1	!= 1111	01	LDRH (immediate) - Post-indexed variant
00	1	!= 1111	10	LDRSB (immediate) - Post-indexed variant
00	1	!= 1111	11	LDRSH (immediate) - Post-indexed variant
01	0	!= 1111	10	Unallocated.
01	0	xxxx	01	STRHT
01	0	xxxx	11	Unallocated.
01	1	xxxx	01	LDRHT
01	1	xxxx	10	LDRSBT
01	1	xxxx	11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) - Offset variant
10	0	xxxx	01	STRH (immediate) - Offset variant
10	0	xxxx	11	STRD (immediate) - Offset variant
10	1	!= 1111	01	LDRH (immediate) - Offset variant
10	1	!= 1111	10	LDRSB (immediate) - Offset variant
10	1	!= 1111	11	LDRSH (immediate) - Offset variant
11	0	!= 1111	10	LDRD (immediate) - Pre-indexed variant
11	0	xxxx	01	STRH (immediate) - Pre-indexed variant
11	0	xxxx	11	STRD (immediate) - Pre-indexed variant
11	1	!= 1111	01	LDRH (immediate) - Pre-indexed variant
11	1	!= 1111	10	LDRSB (immediate) - Pre-indexed variant
11	1	!= 1111	11	LDRSH (immediate) - Pre-indexed variant

F4.1.3 Synchronization primitives and Load-Acquire/Store-Release

This section describes the encoding of the Synchronization primitives and Load-Acquire/Store-Release group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

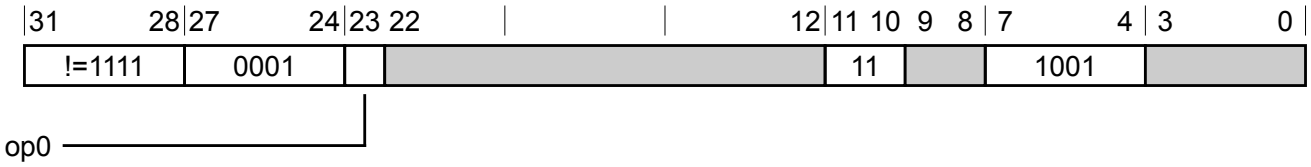

Table F4-8 Encoding table for the Synchronization primitives and Load-Acquire/Store-Release group

Decode fields op0	Decode group or instruction page
0	Unallocated.
1	Load/Store Exclusive and Load-Acquire/Store-Release

F4.1.3.1 Load/Store Exclusive and Load-Acquire/Store-Release

This section describes the encoding of the Load/Store Exclusive and Load-Acquire/Store-Release instruction class. The encodings in this section are decoded from Synchronization primitives and Load-Acquire/Store-Release.

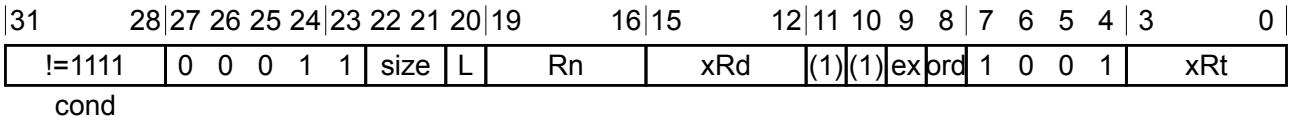

Table F4-9

Decode fields size	L	ex	ord	Instruction page
00	0	0	0	STL
00	0	0	1	Unallocated.
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	0	1	Unallocated.
00	1	1	0	LDAEX
00	1	1	1	LDREX
01	0	0	x	Unallocated.
01	0	1	0	STLEXD
01	0	1	1	STREXD

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
size	L	ex	ord	
01	1	0	x	Unallocated.
01	1	1	0	LDAEXD
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	0	1	Unallocated.
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	0	1	Unallocated.
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	0	1	Unallocated.
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	0	1	Unallocated.
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

F4.1.4 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

31 28	27 23	22 21 20	19 8	7	6 4	3 0
!=1111	00010	op0	0	0	op1	

Table F4-10 Encoding table for the Miscellaneous group

Decode fields		Decode group or instruction page
op0	op1	
00	001	Unallocated.
00	010	Unallocated.
00	011	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Decode group or instruction page
op0	op1	
00	110	Unallocated.
01	001	BX
01	010	BXJ
01	011	BLX (register)
01	110	Unallocated.
10	001	Unallocated.
10	010	Unallocated.
10	011	Unallocated.
10	110	Unallocated.
11	001	CLZ
11	010	Unallocated.
11	011	Unallocated.
11	110	ERET
xx	111	Exception Generation
xx	000	Move special register (register)
xx	100	Cyclic Redundancy Check
xx	101	Integer Saturating Arithmetic

F4.1.4.1 Exception Generation

This section describes the encoding of the Exception Generation instruction class. The encodings in this section are decoded from Miscellaneous.

31 28	27 26 25 24	23 22 21 20	19 8	7 6 5 4	3 0
!=1111	0 0 0 1 0	opc	0	imm12	0 1 1 1	imm4
cond					

Table F4-11

Decode fields	Instruction page
opc	
00	HLT
01	BKPT
10	HVC
11	SMC

F4.1.4.2 Move special register (register)

This section describes the encoding of the Move special register (register) instruction class. The encodings in this section are decoded from Miscellaneous.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	opc		0	mask	Rd	(0)	(0)	B	m	0	0	0	0	Rn
cond																			

Table F4-12

Decode fields		Instruction page
opc	**B**	
`x0`	`0`	MRS
`x0`	`1`	MRS (Banked register)
`x1`	`0`	MSR (register)
`x1`	`1`	MSR (Banked register)

F4.1.4.3 Cyclic Redundancy Check

This section describes the encoding of the Cyclic Redundancy Check instruction class. The encodings in this section are decoded from Miscellaneous.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	0	sz		0	Rn	Rd	(0)	(0)	C	(0)	0	1	0	0	Rm
cond																			

Table F4-13

Decode fields		Instruction page	Feature
sz	**C**		
`00`	`0`	CRC32 - CRC32B variant	FEAT_CRC32
`00`	`1`	CRC32C - CRC32CB variant	FEAT_CRC32
`01`	`0`	CRC32 - CRC32H variant	FEAT_CRC32
`01`	`1`	CRC32C - CRC32CH variant	FEAT_CRC32
`10`	`0`	CRC32 - CRC32W variant	FEAT_CRC32
`10`	`1`	CRC32C - CRC32CW variant	FEAT_CRC32
`11`	`x`	CONSTRAINED UNPREDICTABLE	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1.4.4 Integer Saturating Arithmetic

This section describes the encoding of the Integer Saturating Arithmetic instruction class. The encodings in this section are decoded from Miscellaneous.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	1	0	opc		0	Rn	Rd	(0)	(0)	(0)	(0)	0	1	0	1	Rm

Table F4-14

Decode fields opc	Instruction page
00	QADD
01	QSUB
10	QDADD
11	QDSUB

F4.1.5 Data-processing register (immediate shift)

This section describes the encoding of the Data-processing register (immediate shift) group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

31–28	27–25	24	23	22	21	20	19–5	4	3–0
!=1111	000	op0						0	

op1

This decode also imposes the constraint:

- `op0:op1 != 100`.

Table F4-15 Encoding table for the Data-processing register (immediate shift) group

Decode fields op0	op1	Decode group or instruction page
0x	x	Integer Data Processing (three register, immediate shift)
10	1	Integer Test and Compare (two register, immediate shift)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Decode group or instruction page
op0	op1	
11	x	Logical Arithmetic (three register, immediate shift)

F4.1.5.1 Integer Data Processing (three register, immediate shift)

This section describes the encoding of the Integer Data Processing (three register, immediate shift) instruction class. The encodings in this section are decoded from Data-processing register (immediate shift).

31–28	27 26 25 24	23–21	20	19–16	15–12	11–7	6 5	4	3–0
!=1111 (cond)	0 0 0 0	opc	S	Rn	Rd	imm5	stype	0	Rm

Table F4-16

Decode fields				Instruction page
opc	S	Rn	imm5:stype	
000	x	xxxx	!= 0000011	AND, ANDS (register) - ANDS, shift or rotate by value variant
000	x	xxxx	0000011	AND, ANDS (register) - ANDS, rotate right with extend variant
001	x	xxxx	!= 0000011	EOR, EORS (register) - EORS, shift or rotate by value variant
001	x	xxxx	0000011	EOR, EORS (register) - EORS, rotate right with extend variant
010	0	!= 1101	!= 0000011	SUB, SUBS (register) - SUB, shift or rotate by value variant
010	0	!= 1101	0000011	SUB, SUBS (register) - SUB, rotate right with extend variant
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) - SUB, shift or rotate by value variant
010	0	1101	0000011	SUB, SUBS (SP minus register) - SUB, rotate right with extend variant
010	1	!= 1101	!= 0000011	SUB, SUBS (register) - SUBS, shift or rotate by value variant
010	1	!= 1101	0000011	SUB, SUBS (register) - SUBS, rotate right with extend variant
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) - SUBS, shift or rotate by value variant
010	1	1101	0000011	SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant
011	x	xxxx	!= 0000011	RSB, RSBS (register) - RSBS, shift or rotate by value variant
011	x	xxxx	0000011	RSB, RSBS (register) - RSBS, rotate right with extend variant
100	0	!= 1101	!= 0000011	ADD, ADDS (register) - ADD, shift or rotate by value variant
100	0	!= 1101	0000011	ADD, ADDS (register) - ADD, rotate right with extend variant
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) - ADD, shift or rotate by value variant
100	0	1101	0000011	ADD, ADDS (SP plus register) - ADD, rotate right with extend variant
100	1	!= 1101	!= 0000011	ADD, ADDS (register) - ADDS, shift or rotate by value variant
100	1	!= 1101	0000011	ADD, ADDS (register) - ADDS, rotate right with extend variant
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) - ADDS, shift or rotate by value variant
100	1	1101	0000011	ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
opc	S	Rn	imm5:stype	
101	x	xxxx	!= 0000011	ADC, ADCS (register) - ADCS, shift or rotate by value variant
101	x	xxxx	0000011	ADC, ADCS (register) - ADCS, rotate right with extend variant
110	x	xxxx	!= 0000011	SBC, SBCS (register) - SBCS, shift or rotate by value variant
110	x	xxxx	0000011	SBC, SBCS (register) - SBCS, rotate right with extend variant
111	x	xxxx	!= 0000011	RSC, RSCS (register) - RSCS, shift or rotate by value variant
111	x	xxxx	0000011	RSC, RSCS (register) - RSCS, rotate right with extend variant

F4.1.5.2 Integer Test and Compare (two register, immediate shift)

This section describes the encoding of the Integer Test and Compare (two register, immediate shift) instruction class. The encodings in this section are decoded from Data-processing register (immediate shift).

31–28	27–23	22–21	20	19–16	15	14	13	12	11–7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 0	opc	1	Rn	(0)	(0)	(0)	(0)	imm5	stype	0	Rm

Table F4-17

Decode fields		Instruction page
opc	imm5:stype	
00	!= 0000011	TST (register) - Shift or rotate by value variant
00	0000011	TST (register) - Rotate right with extend variant
01	!= 0000011	TEQ (register) - Shift or rotate by value variant
01	0000011	TEQ (register) - Rotate right with extend variant
10	!= 0000011	CMP (register) - Shift or rotate by value variant
10	0000011	CMP (register) - Rotate right with extend variant
11	!= 0000011	CMN (register) - Shift or rotate by value variant
11	0000011	CMN (register) - Rotate right with extend variant

F4.1.5.3 Logical Arithmetic (three register, immediate shift)

This section describes the encoding of the Logical Arithmetic (three register, immediate shift) instruction class. The encodings in this section are decoded from Data-processing register (immediate shift).

31–28	27–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 1	opc	S	Rn	Rd	imm5	stype	0	Rm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-18

Decode fields		Instruction page
opc	**imm5:stype**	
00	!= 0000011	ORR, ORRS (register) - ORRS, shift or rotate by value variant
00	0000011	ORR, ORRS (register) - ORRS, rotate right with extend variant
01	!= 0000011	MOV, MOVS (register) - MOVS, shift or rotate by value variant
01	0000011	MOV, MOVS (register) - MOVS, rotate right with extend variant
10	!= 0000011	BIC, BICS (register) - BICS, shift or rotate by value variant
10	0000011	BIC, BICS (register) - BICS, rotate right with extend variant
11	!= 0000011	MVN, MVNS (register) - MVNS, shift or rotate by value variant
11	0000011	MVN, MVNS (register) - MVNS, rotate right with extend variant

F4.1.6 Data-processing register (register shift)

This section describes the encoding of the Data-processing register (register shift) group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

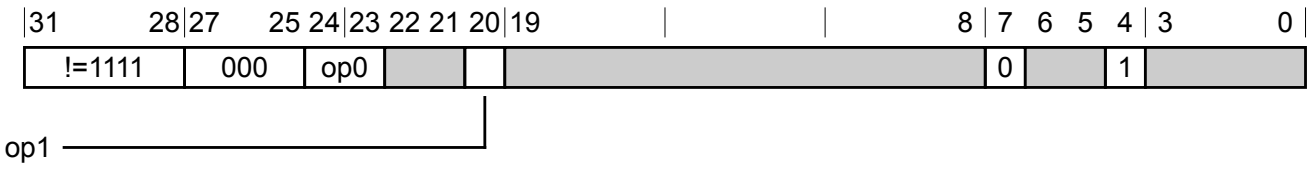

This decode also imposes the constraint:

- `op0:op1 != 100`.

Table F4-19 Encoding table for the Data-processing register (register shift) group

Decode fields		Decode group or instruction page
op0	**op1**	
0x	x	Integer Data Processing (three register, register shift)
10	1	Integer Test and Compare (two register, register shift)
11	x	Logical Arithmetic (three register, register shift)

F4.1.6.1 Integer Data Processing (three register, register shift)

This section describes the encoding of the Integer Data Processing (three register, register shift) instruction class. The encodings in this section are decoded from Data-processing register (register shift).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27 26 25 24	23–21	20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 0	opc	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

Table F4-20

Decode fields opc	Instruction page
000	AND, ANDS (register-shifted register)
001	EOR, EORS (register-shifted register)
010	SUB, SUBS (register-shifted register)
011	RSB, RSBS (register-shifted register)
100	ADD, ADDS (register-shifted register)
101	ADC, ADCS (register-shifted register)
110	SBC, SBCS (register-shifted register)
111	RSC, RSCS (register-shifted register)

F4.1.6.2 Integer Test and Compare (two register, register shift)

This section describes the encoding of the Integer Test and Compare (two register, register shift) instruction class. The encodings in this section are decoded from Data-processing register (register shift).

31–28	27 26 25 24	23 22 21	20	19–16	15	14	13	12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1 0	opc	1	Rn	(0)	(0)	(0)	(0)	Rs	0	stype	1	Rm
cond													

Table F4-21

Decode fields opc	Instruction page
00	TST (register-shifted register)
01	TEQ (register-shifted register)
10	CMP (register-shifted register)
11	CMN (register-shifted register)

F4.1.6.3 Logical Arithmetic (three register, register shift)

This section describes the encoding of the Logical Arithmetic (three register, register shift) instruction class. The encodings in this section are decoded from Data-processing register (register shift).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27 26 25 24	23 22 21	20	19–16	15–12	11–8	7	6 5	4	3–0
!=1111	0 0 0 1	1 opc	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

Table F4-22

Decode fields	Instruction page
opc	
00	ORR, ORRS (register-shifted register)
01	MOV, MOVS (register-shifted register)
10	BIC, BICS (register-shifted register)
11	MVN, MVNS (register-shifted register)

F4.1.7 Data-processing immediate

This section describes the encoding of the Data-processing immediate group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

31–28	27–25	24	23 22 21	20	19–0
!=1111	001	op0		op1	

Table F4-23 Encoding table for the Data-processing immediate group

Decode fields		Decode group or instruction page
op0	**op1**	
0x	xx	Integer Data Processing (two register and immediate)
10	00	Move Halfword (immediate)
10	10	Move Special Register and Hints (immediate)
10	x1	Integer Test and Compare (one register and immediate)
11	xx	Logical Arithmetic (two register and immediate)

F4.1.7.1 Integer Data Processing (two register and immediate)

This section describes the encoding of the Integer Data Processing (two register and immediate) instruction class. The encodings in this section are decoded from Data-processing immediate.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27 26 25 24	23–21	20	19–16	15–12	11–0
!=1111	0 0 1 0	opc	S	Rn	Rd	imm12
cond						

Table F4-24

Decode fields			Instruction page
opc	**S**	**Rn**	
000	x	xxxx	AND, ANDS (immediate)
001	x	xxxx	EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) - SUB variant
010	0	1101	SUB, SUBS (SP minus immediate) - SUB variant
010	0	1111	ADR - A2
010	1	!= 1101	SUB, SUBS (immediate) - SUBS variant
010	1	1101	SUB, SUBS (SP minus immediate) - SUBS variant
011	x	xxxx	RSB, RSBS (immediate)
100	0	!= 11x1	ADD, ADDS (immediate) - ADD variant
100	0	1101	ADD, ADDS (SP plus immediate) - ADD variant
100	0	1111	ADR - A1
100	1	!= 1101	ADD, ADDS (immediate) - ADDS variant
100	1	1101	ADD, ADDS (SP plus immediate) - ADDS variant
101	x	xxxx	ADC, ADCS (immediate)
110	x	xxxx	SBC, SBCS (immediate)
111	x	xxxx	RSC, RSCS (immediate)

F4.1.7.2 Move Halfword (immediate)

This section describes the encoding of the Move Halfword (immediate) instruction class. The encodings in this section are decoded from Data-processing immediate.

31–28	27 26 25 24	23	22	21 20	19–16	15–12	11–0
!=1111	0 0 1 1 0	H		0 0	imm4	Rd	imm12
cond							

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-25

Decode fields	Instruction page
H	
0	MOV, MOVS (immediate)
1	MOVT

F4.1.7.3 Move Special Register and Hints (immediate)

This section describes the encoding of the Move Special Register and Hints (immediate) instruction class. The encodings in this section are decoded from Data-processing immediate.

31 28	27 26 25 24	23	22	21 20	19 16	15	14	13	12	11 0
!=1111	0 0 1 1 0	R	1	0	imm4	(1)	(1)	(1)	(1)	imm12
cond										

Table F4-26

Decode fields		Instruction page	Feature
R:imm4	**imm12**		
!= 00000	xxxxxxxxxxxx	MSR (immediate)	-
00000	xxxx00000000	NOP	-
00000	xxxx00000001	YIELD	-
00000	xxxx00000010	WFE	-
00000	xxxx00000011	WFI	-
00000	xxxx00000100	SEV	-
00000	xxxx00000101	SEVL	-
00000	xxxx0000011x	Reserved hint, behaves as NOP.	-
00000	xxxx00001xxx	Reserved hint, behaves as NOP.	-
00000	xxxx00010000	ESB	FEAT_RAS
00000	xxxx00010001	Reserved hint, behaves as NOP.	-
00000	xxxx00010010	TSB	FEAT_TRF
00000	xxxx00010011	Reserved hint, behaves as NOP.	-
00000	xxxx00010100	CSDB	-
00000	xxxx00010101	Reserved hint, behaves as NOP.	-
00000	xxxx00010110	CLRBHB	FEAT_CLRBHB
00000	xxxx00010111	Reserved hint, behaves as NOP.	-
00000	xxxx00011xxx	Reserved hint, behaves as NOP.	-
00000	xxxx001xxxxx	Reserved hint, behaves as NOP.	-
00000	xxxx01xxxxxx	Reserved hint, behaves as NOP.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page	Feature
R:imm4	imm12		
00000	xxxx10xxxxxx	Reserved hint, behaves as NOP.	-
00000	xxxx110xxxxx	Reserved hint, behaves as NOP.	-
00000	xxxx1110xxxx	Reserved hint, behaves as NOP.	-
00000	xxxx1111xxxx	DBG	-

F4.1.7.4 Integer Test and Compare (one register and immediate)

This section describes the encoding of the Integer Test and Compare (one register and immediate) instruction class. The encodings in this section are decoded from Data-processing immediate.

31–28	27 26 25 24	23 22 21 20	19–16	15 14 13 12	11–0
!=1111	0 0 1 1 0	opc	Rn	(0)(0)(0)(0)	imm12
cond					

Table F4-27

Decode fields	Instruction page
opc	
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

F4.1.7.5 Logical Arithmetic (two register and immediate)

This section describes the encoding of the Logical Arithmetic (two register and immediate) instruction class. The encodings in this section are decoded from Data-processing immediate.

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–0
!=1111	0 0 1 1 1	opc	S	Rn	Rd	imm12
cond					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-28

Decode fields	Instruction page
opc	
00	ORR, ORRS (immediate)
01	MOV, MOVS (immediate)
10	BIC, BICS (immediate)
11	MVN, MVNS (immediate)

F4.1.8 Load/Store Word, Unsigned Byte (immediate, literal)

This section describes the encoding of the Load/Store Word, Unsigned Byte (immediate, literal) instruction class. The encodings in this section are decoded from A32 instruction set encoding.

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 0
!=1111	0 1 0	P	U	o2	W	o1	Rn	Rt	imm12
cond									

Table F4-29

Decode fields				Instruction page
P:W	**o2**	**o1**	**Rn**	
!= 01	0	1	1111	LDR (literal)
!= 01	1	1	1111	LDRB (literal)
00	0	0	xxxx	STR (immediate) - Post-indexed variant
00	0	1	!= 1111	LDR (immediate) - Post-indexed variant
00	1	0	xxxx	STRB (immediate) - Post-indexed variant
00	1	1	!= 1111	LDRB (immediate) - Post-indexed variant
01	0	0	xxxx	STRT
01	0	1	xxxx	LDRT
01	1	0	xxxx	STRBT
01	1	1	xxxx	LDRBT
10	0	0	xxxx	STR (immediate) - Offset variant
10	0	1	!= 1111	LDR (immediate) - Offset variant
10	1	0	xxxx	STRB (immediate) - Offset variant
10	1	1	!= 1111	LDRB (immediate) - Offset variant
11	0	0	xxxx	STR (immediate) - Pre-indexed variant
11	0	1	!= 1111	LDR (immediate) - Pre-indexed variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
P:W	o2	o1	Rn	
11	1	0	xxxx	STRB (immediate) - Pre-indexed variant
11	1	1	!= 1111	LDRB (immediate) - Pre-indexed variant

F4.1.9 Load/Store Word, Unsigned Byte (register)

This section describes the encoding of the Load/Store Word, Unsigned Byte (register) instruction class. The encodings in this section are decoded from A32 instruction set encoding.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0	1	1	P	U	o2	W	o1	Rn	Rt	imm5	stype	0	Rm
cond														

Table F4-30

Decode fields				Instruction page
P	o2	W	o1	
0	0	0	0	STR (register) - Post-indexed variant
0	0	0	1	LDR (register) - Post-indexed variant
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) - Post-indexed variant
0	1	0	1	LDRB (register) - Post-indexed variant
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0	x	0	STR (register) - Pre-indexed variant
1	0	x	1	LDR (register) - Pre-indexed variant
1	1	x	0	STRB (register) - Pre-indexed variant
1	1	x	1	LDRB (register) - Pre-indexed variant

F4.1.10 Media instructions

This section describes the encoding of the Media instructions group. The encodings in this section are decoded from A32 instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27–25	24–20	19–8	7–5	4	3–0
!=1111	011	op0		op1	1	

Table F4-31 Encoding table for the Media instructions group

Decode fields		Decode group or instruction page
op0	op1	
00xxx	xxx	Parallel Arithmetic
01000	101	SEL
01000	001	Unallocated.
01000	xx0	PKHBT, PKHTB
01001	x01	Unallocated.
01001	xx0	Unallocated.
0110x	x01	Unallocated.
0110x	xx0	Unallocated.
01x10	001	Saturate 16-bit
01x10	101	Unallocated.
01x11	x01	Reverse Bit/Byte
01x1x	xx0	Saturate 32-bit
01xxx	111	Unallocated.
01xxx	011	Extend and Add
10xxx	xxx	Signed multiply, Divide
11000	000	Unsigned Sum of Absolute Differences
11000	100	Unallocated.
11001	x00	Unallocated.
1101x	x00	Unallocated.
110xx	111	Unallocated.
1110x	111	Unallocated.
1110x	x00	Bitfield Insert
11110	111	Unallocated.
11111	111	Permanently UNDEFINED
1111x	x00	Unallocated.
11x0x	x10	Unallocated.
11x1x	x10	Bitfield Extract
11xxx	011	Unallocated.
11xxx	x01	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1.10.1 Parallel Arithmetic

This section describes the encoding of the Parallel Arithmetic instruction class. The encodings in this section are decoded from Media instructions.

31–28	27 26 25 24	23 22–20	19–16	15–12	11	10	9	8	7	6 5	4	3–0
!=1111	0 1 1 0	0	op1	Rn	Rd	(1)	(1)	(1)	(1)	B	op2	1	Rm
cond												

Table F4-32

Decode fields			Instruction page
op1	B	op2	
000	x	xx	Unallocated.
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	Unallocated.
001	1	10	Unallocated.
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	Unallocated.
010	1	10	Unallocated.
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	Unallocated.
011	1	10	Unallocated.
011	1	11	SHSUB8
100	x	xx	Unallocated.
101	0	00	UADD16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields			Instruction page
op1	B	op2	
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	Unallocated.
101	1	10	Unallocated.
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	Unallocated.
110	1	10	Unallocated.
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	Unallocated.
111	1	10	Unallocated.
111	1	11	UHSUB8

F4.1.10.2 Saturate 16-bit

This section describes the encoding of the Saturate 16-bit instruction class. The encodings in this section are decoded from Media instructions.

31 28	27 26 25 24	23	22	21 20	19 16	15 12	11	10	9	8	7 6 5 4	3 0
!=1111	0 1 1 0 1	U		1 0	sat_imm	Rd	(1)	(1)	(1)	(1)	0 0 1 1	Rn
cond												

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-33

Decode fields	Instruction page
U	
0	SSAT16
1	USAT16

F4.1.10.3 Reverse Bit/Byte

This section describes the encoding of the Reverse Bit/Byte instruction class. The encodings in this section are decoded from Media instructions.

31 28	27 26 25 24	23	22	21 20	19 18 17 16	15 12	11 10 9 8	7	6 5 4	3 0
!=1111	0 1 1 0	1	o1	1 1	(1)(1)(1)(1)	Rd	(1)(1)(1)(1)	o2	0 1 1	Rm
cond										

Table F4-34

Decode fields		Instruction page
o1	**o2**	
0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

F4.1.10.4 Saturate 32-bit

This section describes the encoding of the Saturate 32-bit instruction class. The encodings in this section are decoded from Media instructions.

31 28	27 26 25 24	23	22	21	20 16	15 12	11 7	6	5 4	3 0
!=1111	0 1 1 0	1	U	1	sat_imm	Rd	imm5	sh	0 1	Rn
cond										

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-35

Decode fields	Instruction page
U	
0	SSAT
1	USAT

F4.1.10.5 Extend and Add

This section describes the encoding of the Extend and Add instruction class. The encodings in this section are decoded from Media instructions.

31–28	27 26 25 24	23	22	21 20	19–16	15–12	11 10	9	8	7 6 5 4	3–0
!=1111	0 1 1 0	1	U	op	Rn	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond											

Table F4-36

Decode fields			Instruction page
U	**op**	**Rn**	
0	00	!= 1111	SXTAB16
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

F4.1.10.6 Signed multiply, Divide

This section describes the encoding of the Signed multiply, Divide instruction class. The encodings in this section are decoded from Media instructions.

31–28	27 26 25 24	23 22 20	19–16	15–12	11–8	7–5	4	3–0
!=1111	0 1 1 1	0 op1	Rd	Ra	Rm	op2	1	Rn
cond								

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-37

Decode fields			Instruction page
op1	Ra	op2	
000	!= 1111	000	SMLAD, SMLADX - SMLAD variant
000	!= 1111	001	SMLAD, SMLADX - SMLADX variant
000	!= 1111	010	SMLSD, SMLSDX - SMLSD variant
000	!= 1111	011	SMLSD, SMLSDX - SMLSDX variant
000	xxxx	1xx	Unallocated.
000	1111	000	SMUAD, SMUADX - SMUAD variant
000	1111	001	SMUAD, SMUADX - SMUADX variant
000	1111	010	SMUSD, SMUSDX - SMUSD variant
000	1111	011	SMUSD, SMUSDX - SMUSDX variant
001	xxxx	000	SDIV
001	xxxx	!= 000	Unallocated.
010	xxxx	xxx	Unallocated.
011	xxxx	000	UDIV
011	xxxx	!= 000	Unallocated.
100	xxxx	000	SMLALD, SMLALDX - SMLALD variant
100	xxxx	001	SMLALD, SMLALDX - SMLALDX variant
100	xxxx	010	SMLSLD, SMLSLDX - SMLSLD variant
100	xxxx	011	SMLSLD, SMLSLDX - SMLSLDX variant
100	xxxx	1xx	Unallocated.
101	!= 1111	000	SMMLA, SMMLAR - SMMLA variant
101	!= 1111	001	SMMLA, SMMLAR - SMMLAR variant
101	xxxx	01x	Unallocated.
101	xxxx	10x	Unallocated.
101	xxxx	110	SMMLS, SMMLSR - SMMLS variant
101	xxxx	111	SMMLS, SMMLSR - SMMLSR variant
101	1111	000	SMMUL, SMMULR - SMMUL variant
101	1111	001	SMMUL, SMMULR - SMMULR variant
11x	xxxx	xxx	Unallocated.

F4.1.10.7 Unsigned Sum of Absolute Differences

This section describes the encoding of the Unsigned Sum of Absolute Differences instruction class. The encodings in this section are decoded from Media instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27 26 25 24	23 22 21 20	19–16	15–12	11–8	7 6 5 4	3–0
!=1111	0 1 1 1	1 0 0 0	Rd	Ra	Rm	0 0 0 1	Rn
cond							

Table F4-38

Decode fields Ra	Instruction page
`!= 1111`	USADA8
`1111`	USAD8

F4.1.10.8 Bitfield Insert

This section describes the encoding of the Bitfield Insert instruction class. The encodings in this section are decoded from Media instructions.

31–28	27 26 25 24	23 22 21 20	16	15–12	11–	7 6 5 4	3–0
!=1111	0 1 1 1	1 1 0	msb	Rd	lsb	0 0 1	Rn
cond							

Table F4-39

Decode fields Rn	Instruction page
`!= 1111`	BFI
`1111`	BFC

F4.1.10.9 Permanently UNDEFINED

This section describes the encoding of the Permanently UNDEFINED instruction class. The encodings in this section are decoded from Media instructions.

31–28	27 26 25 24	23 22 21 20	19–	–8	7 6 5 4	3–0
!=1111	0 1 1 1	1 1 1 1	imm12		1 1 1 1	imm4
cond						

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-40

Decode fields cond	Instruction page
`0xxx`	Unallocated.
`10xx`	Unallocated.
`110x`	Unallocated.
`1110`	UDF

F4.1.10.10 Bitfield Extract

This section describes the encoding of the Bitfield Extract instruction class. The encodings in this section are decoded from Media instructions.

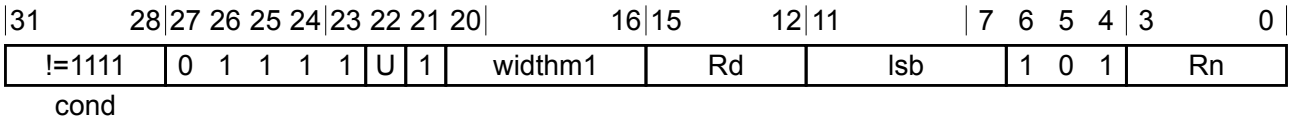

Table F4-41

Decode fields U	Instruction page
`0`	SBFX
`1`	UBFX

F4.1.11 Branch, branch with link, and block data transfer

This section describes the encoding of the Branch, branch with link, and block data transfer group. The encodings in this section are decoded from A32 instruction set encoding.

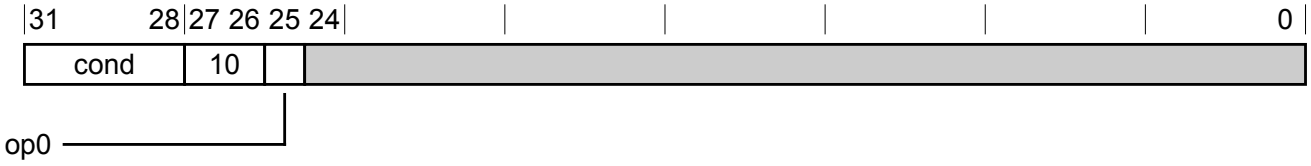

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-42 Encoding table for the Branch, branch with link, and block data transfer group

Decode fields		Decode group or instruction page
cond	**op0**	
1111	0	Exception Save/Restore
!= 1111	0	Load/Store Multiple
xxxx	1	Branch (immediate)

F4.1.11.1 Exception Save/Restore

This section describes the encoding of the Exception Save/Restore instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer.

31 30 29 28	27 26 25	24	23	22	21	20	19 16	15 5	4 0
1 1 1 1	1 0 0	P	U	S	W	L	Rn	op	mode

Table F4-43

Decode fields				Instruction page
P	**U**	**S**	**L**	
x	x	0	0	Unallocated.
0	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement After variant
0	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement After variant
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment After variant
0	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment After variant
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement Before variant
1	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement Before variant
x	x	1	1	Unallocated.
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment Before variant
1	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment Before variant

F4.1.11.2 Load/Store Multiple

This section describes the encoding of the Load/Store Multiple instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer.

31 28	27 26 25	24	23	22	21	20	19 16	15 0
!=1111	1 0 0	P	U	op	W	L	Rn	register_list
cond								

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-44

Decode fields					Instruction page
P	**U**	**op**	**L**	**register_list**	
0	0	0	0	xxxxxxxxxxxxxxxx	STMDA, STMED
0	0	0	1	xxxxxxxxxxxxxxxx	LDMDA, LDMFA
0	1	0	0	xxxxxxxxxxxxxxxx	STM, STMIA, STMEA
0	1	0	1	xxxxxxxxxxxxxxxx	LDM, LDMIA, LDMFD
x	x	1	0	xxxxxxxxxxxxxxxx	STM (User registers)
1	0	0	0	xxxxxxxxxxxxxxxx	STMDB, STMFD
1	0	0	1	xxxxxxxxxxxxxxxx	LDMDB, LDMEA
x	x	1	1	0xxxxxxxxxxxxxxx	LDM (User registers)
1	1	0	0	xxxxxxxxxxxxxxxx	STMIB, STMFA
1	1	0	1	xxxxxxxxxxxxxxxx	LDMIB, LDMED
x	x	1	1	1xxxxxxxxxxxxxxx	LDM (exception return)

F4.1.11.3 Branch (immediate)

This section describes the encoding of the Branch (immediate) instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer.

31 28	27 26 25	24	23 0
cond	1 0 1	H	imm24

Table F4-45

Decode fields		Instruction page
cond	**H**	
!= 1111	0	B
!= 1111	1	BL, BLX (immediate) - A1
1111	x	BL, BLX (immediate) - A2

F4.1.12 System register access, Advanced SIMD, floating-point, and Supervisor call

This section describes the encoding of the System register access, Advanced SIMD, floating-point, and Supervisor call group. The encodings in this section are decoded from A32 instruction set encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-46 Encoding table for the System register access, Advanced SIMD, floating-point, and Supervisor call group

Decode fields				Decode group or instruction page
cond	**op0**	**op1**	**op2**	
`xxxx`	`0x`	`0x`	`x`	Unallocated.
`xxxx`	`10`	`0x`	`x`	Unallocated.
`xxxx`	`11`	`xx`	`x`	Supervisor call
`1111`	`!= 11`	`1x`	`x`	Unconditional Advanced SIMD and floating-point instructions
`!= 1111`	`0x`	`1x`	`x`	Advanced SIMD and System register load/store and 64-bit move
`!= 1111`	`10`	`1x`	`1`	Advanced SIMD and System register 32-bit move
`!= 1111`	`10`	`10`	`0`	Floating-point data-processing
`!= 1111`	`10`	`11`	`0`	Unallocated.

F4.1.13 Supervisor call

This section describes the encoding of the Supervisor call group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call.

Table F4-47 Encoding table for the Supervisor call group

Decode fields	Decode group or instruction page
cond	
`1111`	Unallocated.
`!= 1111`	SVC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1.14 Unconditional Advanced SIMD and floating-point instructions

This section describes the encoding of the Unconditional Advanced SIMD and floating-point instructions group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call.

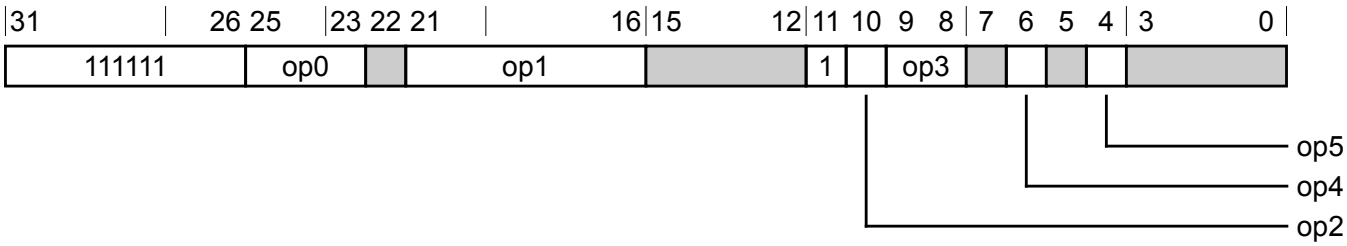

This decode also imposes the constraint:

- `op0<2:1> != 11`.

Table F4-48 Encoding table for the Unconditional Advanced SIMD and floating-point instructions group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0xx	xxxxxx	x	0x	x	x	Advanced SIMD three registers of the same length extension
100	xxxxxx	0	!= 00	0	0	Floating-point conditional select
101	00xxxx	0	!= 00	x	0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x	xxxxxx	0	00	x	x	Advanced SIMD and floating-point multiply with accumulate
10x	xxxxxx	1	0x	x	x	Advanced SIMD and floating-point dot product

F4.1.14.1 Advanced SIMD three registers of the same length extension

This section describes the encoding of the Advanced SIMD three registers of the same length extension instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

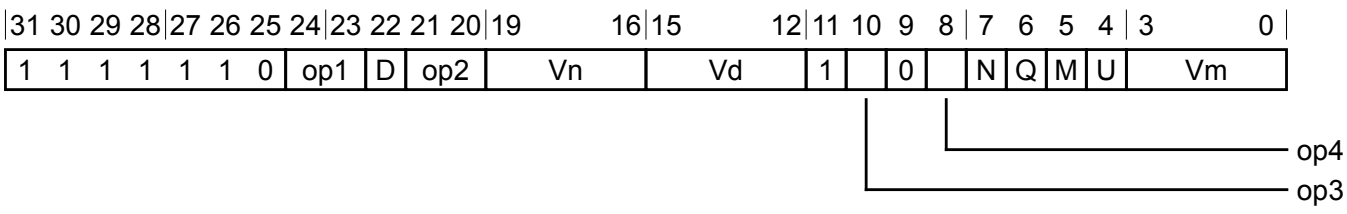

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-49

Decode fields						Instruction page	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD - 64-bit SIMD vector variant	FEAT_FCMA
x1	0x	0	0	0	1	Unallocated.	-
x1	0x	0	0	1	0	VCADD - 128-bit SIMD vector variant	FEAT_FCMA
x1	0x	0	0	1	1	Unallocated.	-
00	0x	0	0	x	x	Unallocated.	-
00	0x	0	1	x	x	Unallocated.	-
00	00	1	0	0	0	Unallocated.	-
00	00	1	0	0	1	Unallocated.	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	Unallocated.	-
00	00	1	1	0	0	VDOT (vector) - 64-bit SIMD vector variant	FEAT_AA32BF16
00	00	1	1	0	1	Unallocated.	-
00	00	1	1	1	0	VDOT (vector) - 128-bit SIMD vector variant	FEAT_AA32BF16
00	00	1	1	1	1	Unallocated.	-
00	01	1	0	x	x	Unallocated.	-
00	01	1	1	x	x	Unallocated.	-
00	10	0	0	x	1	VFMAL (vector)	FEAT_FHM
00	10	0	1	x	x	Unallocated.	-
00	10	1	0	0	x	Unallocated.	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) - 64-bit SIMD vector variant	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) - 64-bit SIMD vector variant	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) - 128-bit SIMD vector variant	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) - 128-bit SIMD vector variant	FEAT_DotProd
00	11	0	0	x	1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1	x	x	Unallocated.	-
00	11	1	0	x	x	Unallocated.	-
00	11	1	1	x	x	Unallocated.	-
01	10	0	0	x	1	VFMSL (vector)	FEAT_FHM
01	10	0	1	x	x	Unallocated.	-
01	10	1	0	0	x	Unallocated.	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	Unallocated.	-
01	10	1	1	0	0	VUSDOT (vector) - 64-bit SIMD vector variant	FEAT_AA32I8MM
01	10	1	1	x	1	Unallocated.	-
01	10	1	1	1	0	VUSDOT (vector) - 128-bit SIMD vector variant	FEAT_AA32I8MM

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields						Instruction page	Feature
op1	op2	op3	op4	Q	U		
01	11	0	1	x	x	Unallocated.	-
01	11	1	0	x	x	Unallocated.	-
01	11	1	1	x	x	Unallocated.	-
xx	1x	0	0	x	0	VCMLA	FEAT_FCMA
10	11	0	1	x	x	Unallocated.	-
10	11	1	0	x	x	Unallocated.	-
10	11	1	1	x	x	Unallocated.	-
11	11	0	1	x	x	Unallocated.	-
11	11	1	0	x	x	Unallocated.	-
11	11	1	1	x	x	Unallocated.	-

F4.1.14.2 Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	0 D cc	Vn	Vd	1 0	!=00	N	0	M	0	Vm
						size					

Table F4-50

Decode fields	Instruction page	Feature
size		
01	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, half-precision scalar variant	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, single-precision scalar variant	-
11	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, double-precision scalar variant	-

F4.1.14.3 Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1 D 0 0	Vn	Vd	1 0	!=00	N	op	M	0	Vm
						size					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-51

Decode fields		Instruction page	Feature
size	op		
01	0	VMAXNM - Half-precision scalar variant	FEAT_FP16
01	1	VMINNM - Half-precision scalar variant	FEAT_FP16
10	0	VMAXNM - Single-precision scalar variant	-
10	1	VMINNM - Single-precision scalar variant	-
11	0	VMAXNM - Double-precision scalar variant	-
11	1	VMINNM - Double-precision scalar variant	-

F4.1.14.4 Floating-point extraction and insertion

This section describes the encoding of the Floating-point extraction and insertion instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

31 30 29 28	27 26 25 24	23	22	21 20	19 18 17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1	0 0 0 0	Vd	1 0	!=00	op	1	M	0	Vm
								size					

Table F4-52

Decode fields		Instruction page	Feature
size	op		
01	x	Unallocated.	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11	x	Unallocated.	-

F4.1.14.5 Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

31 30 29 28	27 26 25 24	23	22	21 20	19 18	17 16	15 12	11 10	9 8	7	6	5	4	3 0
1 1 1 1	1 1 1 0	1	D	1 1	1	o1	RM	Vd	1 0	!=00	op	1	M	0	Vm
									size					

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-53

Decode fields				Instruction page	Feature
o1	RM	size	op		
0	xx	!= 00	1	Unallocated.	-
0	00	01	0	VRINTA (floating-point) - Half-precision scalar variant	FEAT_FP16
0	00	10	0	VRINTA (floating-point) - Single-precision scalar variant	-
0	00	11	0	VRINTA (floating-point) - Double-precision scalar variant	-
0	01	01	0	VRINTN (floating-point) - Half-precision scalar variant	FEAT_FP16
0	01	10	0	VRINTN (floating-point) - Single-precision scalar variant	-
0	01	11	0	VRINTN (floating-point) - Double-precision scalar variant	-
0	10	01	0	VRINTP (floating-point) - Half-precision scalar variant	FEAT_FP16
0	10	10	0	VRINTP (floating-point) - Single-precision scalar variant	-
0	10	11	0	VRINTP (floating-point) - Double-precision scalar variant	-
0	11	01	0	VRINTM (floating-point) - Half-precision scalar variant	FEAT_FP16
0	11	10	0	VRINTM (floating-point) - Single-precision scalar variant	-
0	11	11	0	VRINTM (floating-point) - Double-precision scalar variant	-
1	00	01	x	VCVTA (floating-point) - Half-precision scalar variant	FEAT_FP16
1	00	10	x	VCVTA (floating-point) - Single-precision scalar variant	-
1	00	11	x	VCVTA (floating-point) - Double-precision scalar variant	-
1	01	01	x	VCVTN (floating-point) - Half-precision scalar variant	FEAT_FP16
1	01	10	x	VCVTN (floating-point) - Single-precision scalar variant	-
1	01	11	x	VCVTN (floating-point) - Double-precision scalar variant	-
1	10	01	x	VCVTP (floating-point) - Half-precision scalar variant	FEAT_FP16
1	10	10	x	VCVTP (floating-point) - Single-precision scalar variant	-
1	10	11	x	VCVTP (floating-point) - Double-precision scalar variant	-
1	11	01	x	VCVTM (floating-point) - Half-precision scalar variant	FEAT_FP16
1	11	10	x	VCVTM (floating-point) - Single-precision scalar variant	-
1	11	11	x	VCVTM (floating-point) - Double-precision scalar variant	-

F4.1.14.6 Advanced SIMD and floating-point multiply with accumulate

This section describes the encoding of the Advanced SIMD and floating-point multiply with accumulate instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

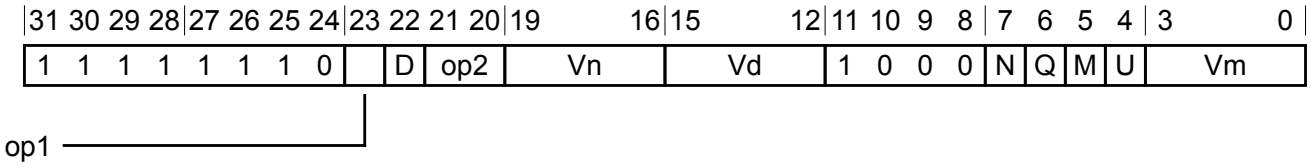

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-54

Decode fields				Instruction page	Feature
op1	op2	Q	U		
0	xx	x	0	VCMLA (by element) - 128-bit SIMD vector of half-precision floating-point variant	FEAT_FCMA
0	00	x	1	VFMAL (by scalar)	FEAT_FHM
0	01	x	1	VFMSL (by scalar)	FEAT_FHM
0	10	x	1	Unallocated.	-
0	11	x	1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1	xx	0	0	VCMLA (by element) - 64-bit SIMD vector of single-precision floating-point variant	FEAT_FCMA
1	xx	x	1	Unallocated.	-
1	xx	1	0	VCMLA (by element) - 128-bit SIMD vector of single-precision floating-point variant	FEAT_FCMA

F4.1.14.7 Advanced SIMD and floating-point dot product

This section describes the encoding of the Advanced SIMD and floating-point dot product instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions.

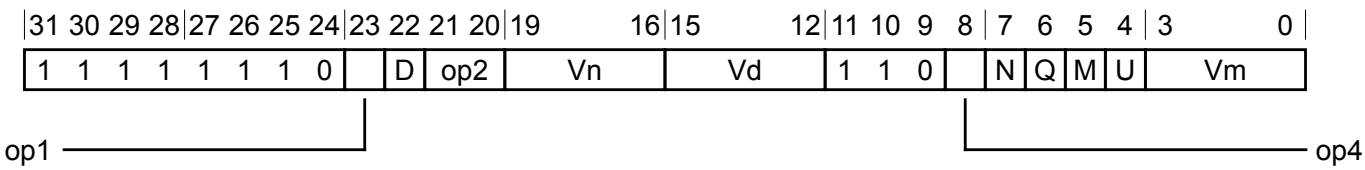

Table F4-55

Decode fields					Instruction page	Feature
op1	op2	op4	Q	U		
0	00	0	x	x	Unallocated.	-
0	00	1	0	0	VDOT (by element) - 64-bit SIMD vector variant	FEAT_AA32BF16
0	00	1	x	1	Unallocated.	-
0	00	1	1	0	VDOT (by element) - 128-bit SIMD vector variant	FEAT_AA32BF16
0	01	0	x	x	Unallocated.	-
0	10	0	x	x	Unallocated.	-
0	10	1	0	0	VSDOT (by element) - 64-bit SIMD vector variant	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) - 64-bit SIMD vector variant	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) - 128-bit SIMD vector variant	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) - 128-bit SIMD vector variant	FEAT_DotProd
0	11	x	x	x	Unallocated.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page	Feature
op1	op2	op4	Q	U		
1	xx	0	x	x	Unallocated.	-
1	00	1	0	0	VUSDOT (by element) - 64-bit SIMD vector variant	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) - 64-bit SIMD vector variant	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) - 128-bit SIMD vector variant	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) - 128-bit SIMD vector variant	FEAT_AA32I8MM
1	01	1	x	x	Unallocated.	-
1	1x	1	x	x	Unallocated.	-

F4.1.15 Advanced SIMD and System register load/store and 64-bit move

This section describes the encoding of the Advanced SIMD and System register load/store and 64-bit move group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding

31–28	27–25	24–21	20–12	11	10–9	8–0
!=1111	110	op0		1	op1	

Table F4-56 Encoding table for the Advanced SIMD and System register load/store and 64-bit move group

Decode fields		Decode group or instruction page
op0	op1	
00x0	0x	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register load/store
xxxx	10	Unallocated.

F4.1.15.1 Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
!=1111	1	1	0	0	0	D	0	op	Rt2	Rt	1	0	size	opc2		M	o3	Vm
cond																		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-57

Decode fields					Instruction page
D	**op**	**size**	**opc2**	**o3**	
0	x	xx	xx	x	Unallocated.
1	x	xx	xx	0	Unallocated.
1	x	0x	00	1	Unallocated.
1	x	xx	01	x	Unallocated.
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - From general-purpose registers variant
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - From general-purpose registers variant
1	x	xx	1x	x	Unallocated.
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - To general-purpose registers variant
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - To general-purpose registers variant

F4.1.15.2 System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

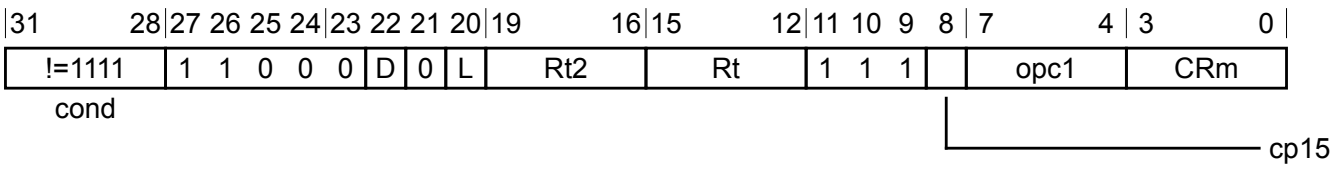

Table F4-58

Decode fields		Instruction page
D	**L**	
0	x	Unallocated.
1	0	MCRR
1	1	MRRC

F4.1.15.3 Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7–0
!=1111	1	1	0	P	U	D	W	L	Rn	Vd	1	0	size	imm8
cond														

Table F4-59

Decode fields							Instruction page	Feature
P	**U**	**W**	**L**	**Rn**	**size**	**imm8**		
0	0	1	x	xxxx	xx	xxxxxxxx	Unallocated.	-
0	1	x	x	xxxx	0x	xxxxxxxx	Unallocated.	-
0	1	x	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Increment After variant	-
0	1	x	0	xxxx	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - Increment After variant	-
0	1	x	0	xxxx	11	xxxxxxx1	FSTMDBX, FSTMIAX - Increment After variant	-
0	1	x	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Increment After variant	-
0	1	x	1	xxxx	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - Increment After variant	-
0	1	x	1	xxxx	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Increment After variant	-
1	x	0	0	xxxx	01	xxxxxxxx	VSTR - Half-precision scalar variant	FEAT_FP16
1	x	0	0	xxxx	10	xxxxxxxx	VSTR - Single-precision scalar variant	-
1	x	0	0	xxxx	11	xxxxxxxx	VSTR - Double-precision scalar variant	-
1	x	0	1	!= 1111	01	xxxxxxxx	VLDR (immediate) - Half-precision scalar variant	FEAT_FP16
1	x	0	1	!= 1111	10	xxxxxxxx	VLDR (immediate) - Single-precision scalar variant	-
1	x	0	1	!= 1111	11	xxxxxxxx	VLDR (immediate) - Double-precision scalar variant	-
1	0	1	x	xxxx	0x	xxxxxxxx	Unallocated.	-
1	0	1	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Decrement Before variant	-
1	0	1	0	xxxx	11	xxxxxxx0	VSTM, VSTMDB, VSTMIA - Decrement Before variant	-
1	0	1	0	xxxx	11	xxxxxxx1	FSTMDBX, FSTMIAX - Decrement Before variant	-
1	0	1	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Decrement Before variant	-
1	0	1	1	xxxx	11	xxxxxxx0	VLDM, VLDMDB, VLDMIA - Decrement Before variant	-
1	0	1	1	xxxx	11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Decrement Before variant	-
1	x	0	1	1111	01	xxxxxxxx	VLDR (literal) - Half-precision scalar variant	FEAT_FP16
1	x	0	1	1111	10	xxxxxxxx	VLDR (literal) - Single-precision scalar variant	-
1	x	0	1	1111	11	xxxxxxxx	VLDR (literal) - Double-precision scalar variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields							Instruction page	Feature
P	U	W	L	Rn	size	imm8		
1	1	1	x	xxxx	xx	xxxxxxxx	Unallocated.	-

F4.1.15.4 System register load/store

This section describes the encoding of the System register load/store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

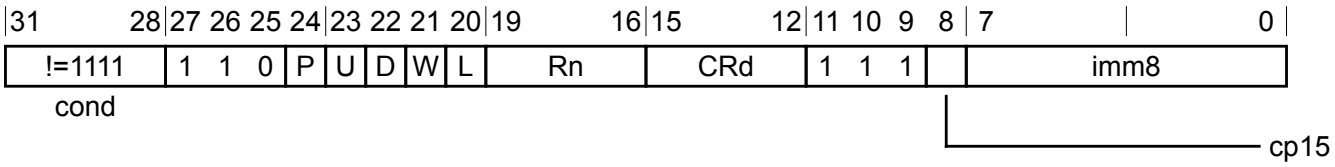

Table F4-60

Decode fields						Instruction page
P:U:W	D	L	Rn	CRd	cp15	
!= 000	0	x	xxxx	!= 0101	0	Unallocated.
!= 000	0	1	1111	0101	0	LDC (literal)
!= 000	x	x	xxxx	xxxx	1	Unallocated.
!= 000	1	x	xxxx	0101	0	Unallocated.
0x1	0	0	xxxx	0101	0	STC - Post-indexed variant
0x1	0	1	!= 1111	0101	0	LDC (immediate) - Post-indexed variant
010	0	0	xxxx	0101	0	STC - Unindexed variant
010	0	1	!= 1111	0101	0	LDC (immediate) - Unindexed variant
1x0	0	0	xxxx	0101	0	STC - Offset variant
1x0	0	1	!= 1111	0101	0	LDC (immediate) - Offset variant
1x1	0	0	xxxx	0101	0	STC - Pre-indexed variant
1x1	0	1	!= 1111	0101	0	LDC (immediate) - Pre-indexed variant

F4.1.16 Advanced SIMD and System register 32-bit move

This section describes the encoding of the Advanced SIMD and System register 32-bit move group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-61 Encoding table for the Advanced SIMD and System register 32-bit move group

Decode fields		Decode group or instruction page	Feature
op0	op1		
000	000	Unallocated.	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	Unallocated.	-
01x	010	Unallocated.	-
10x	010	Unallocated.	-
110	010	Unallocated.	-
111	010	Floating-point move special register	-
xxx	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
xxx	10x	Unallocated.	-
xxx	11x	System register 32-bit move	-

F4.1.16.1 Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move.

31 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7	6	5	4	3	2	1	0
!=1111	1 1 1 0	1 1 1 L	reg	Rt	1 0 1 0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond													

Table F4-62

Decode fields	Instruction page
L	
0	VMSR
1	VMRS

F4.1.16.2 Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–28	27 26 25 24	23–21	20	19–16	15–12	11 10 9 8	7	6 5	4	3	2	1	0
!=1111	1 1 1 0	opc1	L	Vn	Rt	1 0 1 1	N	opc2	1	(0)	(0)	(0)	(0)
cond													

Table F4-63

Decode fields			Instruction page
opc1	**L**	**opc2**	
`0xx`	`0`	`xx`	VMOV (general-purpose register to scalar)
`xxx`	`1`	`xx`	VMOV (scalar to general-purpose register)
`1xx`	`0`	`0x`	VDUP (general-purpose register)
`1xx`	`0`	`1x`	Unallocated.

F4.1.16.3 System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move.

31–28	27 26 25 24	23–21	20	19–16	15–12	11 10 9 8	7–5	4	3–0
!=1111	1 1 1 0	opc1	L	CRn	Rt	1 1 1	opc2	1	CRm
cond									

cp15

Table F4-64

Decode fields	Instruction page
L	
`0`	MCR
`1`	MRC

F4.1.17 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-65 Encoding table for the Floating-point data-processing group

Decode fields		Decode group or instruction page
op0	**op1**	
`1x11`	`1`	Floating-point data-processing (two registers)
`1x11`	`0`	Floating-point move immediate
`!= 1x11`	`x`	Floating-point data-processing (three registers)

F4.1.17.1 Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

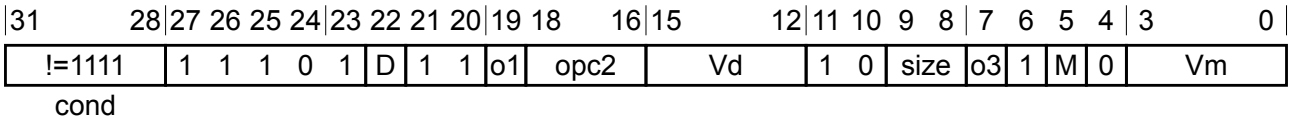

Table F4-66

Decode fields				Instruction page	Feature
o1	**opc2**	**size**	**o3**		
`x`	`xxx`	`00`	`x`	Unallocated.	-
`0`	`000`	`01`	`0`	Unallocated.	-
`0`	`000`	`01`	`1`	VABS - Half-precision scalar variant	FEAT_FP16
`0`	`000`	`10`	`0`	VMOV (register) - Single-precision scalar variant	-
`0`	`000`	`10`	`1`	VABS - Single-precision scalar variant	-
`0`	`000`	`11`	`0`	VMOV (register) - Double-precision scalar variant	-
`0`	`000`	`11`	`1`	VABS - Double-precision scalar variant	-
`0`	`001`	`01`	`0`	VNEG - Half-precision scalar variant	FEAT_FP16
`0`	`001`	`01`	`1`	VSQRT - Half-precision scalar variant	FEAT_FP16
`0`	`001`	`10`	`0`	VNEG - Single-precision scalar variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
o1	opc2	size	o3		
0	001	10	1	VSQRT - Single-precision scalar variant	-
0	001	11	0	VNEG - Double-precision scalar variant	-
0	001	11	1	VSQRT - Double-precision scalar variant	-
0	010	01	x	Unallocated.	-
0	010	10	0	VCVTB - Half-precision to single-precision variant	-
0	010	10	1	VCVTT - Half-precision to single-precision variant	-
0	010	11	0	VCVTB - Half-precision to double-precision variant	-
0	010	11	1	VCVTT - Half-precision to double-precision variant	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB - Single-precision to half-precision variant	-
0	011	10	1	VCVTT - Single-precision to half-precision variant	-
0	011	11	0	VCVTB - Double-precision to half-precision variant	-
0	011	11	1	VCVTT - Double-precision to half-precision variant	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR - Half-precision scalar variant	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) - Half-precision scalar variant	FEAT_FP16
0	110	10	0	VRINTR - Single-precision scalar variant	-
0	110	10	1	VRINTZ (floating-point) - Single-precision scalar variant	-
0	110	11	0	VRINTR - Double-precision scalar variant	-
0	110	11	1	VRINTZ (floating-point) - Double-precision scalar variant	-
0	111	01	0	VRINTX (floating-point) - Half-precision scalar variant	FEAT_FP16
0	111	01	1	Unallocated.	-
0	111	10	0	VRINTX (floating-point) - Single-precision scalar variant	-
0	111	10	1	VCVT (between double-precision and single-precision) - Single-precision to double-precision variant	-
0	111	11	0	VRINTX (floating-point) - Double-precision scalar variant	-
0	111	11	1	VCVT (between double-precision and single-precision) - Double-precision to single-precision variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
o1	opc2	size	o3		
1	000	01	x	VCVT (integer to floating-point, floating-point) - Half-precision scalar variant	FEAT_FP16
1	000	10	x	VCVT (integer to floating-point, floating-point) - Single-precision scalar variant	-
1	000	11	x	VCVT (integer to floating-point, floating-point) - Double-precision scalar variant	-
1	001	01	x	Unallocated.	-
1	001	10	x	Unallocated.	-
1	001	11	0	Unallocated.	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01	x	VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10	x	VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11	x	VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01	x	VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10	x	VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11	x	VCVT (between floating-point and fixed-point, floating-point)	-

F4.1.17.2 Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from Floating-point data-processing.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
!=1111	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L
cond																		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-67

Decode fields size	Instruction page	Feature
00	Unallocated.	-
01	VMOV (immediate) - Half-precision scalar variant	FEAT_FP16
10	VMOV (immediate) - Single-precision scalar variant	-
11	VMOV (immediate) - Double-precision scalar variant	-

F4.1.17.3 Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

31 28	27 26 25 24	23	22	21 20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
!=1111	1 1 1 0	o0	D	o1	Vn	Vd	1 0	size	N	o2	M	0	Vm
cond													

Table F4-68

Decode fields o0:o1	size	o2	Instruction page	Feature
!= 111	00	x	Unallocated.	-
000	01	0	VMLA (floating-point) - Half-precision scalar variant	FEAT_FP16
000	01	1	VMLS (floating-point) - Half-precision scalar variant	FEAT_FP16
000	10	0	VMLA (floating-point) - Single-precision scalar variant	-
000	10	1	VMLS (floating-point) - Single-precision scalar variant	-
000	11	0	VMLA (floating-point) - Double-precision scalar variant	-
000	11	1	VMLS (floating-point) - Double-precision scalar variant	-
001	01	0	VNMLS - Half-precision scalar variant	FEAT_FP16
001	01	1	VNMLA - Half-precision scalar variant	FEAT_FP16
001	10	0	VNMLS - Single-precision scalar variant	-
001	10	1	VNMLA - Single-precision scalar variant	-
001	11	0	VNMLS - Double-precision scalar variant	-
001	11	1	VNMLA - Double-precision scalar variant	-
010	01	0	VMUL (floating-point) - Half-precision scalar variant	FEAT_FP16
010	01	1	VNMUL - Half-precision scalar variant	FEAT_FP16
010	10	0	VMUL (floating-point) - Single-precision scalar variant	-
010	10	1	VNMUL - Single-precision scalar variant	-
010	11	0	VMUL (floating-point) - Double-precision scalar variant	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields			Instruction page	Feature
o0:o1	size	o2		
010	11	1	VNMUL - Double-precision scalar variant	-
011	01	0	VADD (floating-point) - Half-precision scalar variant	FEAT_FP16
011	01	1	VSUB (floating-point) - Half-precision scalar variant	FEAT_FP16
011	10	0	VADD (floating-point) - Single-precision scalar variant	-
011	10	1	VSUB (floating-point) - Single-precision scalar variant	-
011	11	0	VADD (floating-point) - Double-precision scalar variant	-
011	11	1	VSUB (floating-point) - Double-precision scalar variant	-
100	01	0	VDIV - Half-precision scalar variant	FEAT_FP16
100	10	0	VDIV - Single-precision scalar variant	-
100	11	0	VDIV - Double-precision scalar variant	-
101	01	0	VFNMS - Half-precision scalar variant	FEAT_FP16
101	01	1	VFNMA - Half-precision scalar variant	FEAT_FP16
101	10	0	VFNMS - Single-precision scalar variant	-
101	10	1	VFNMA - Single-precision scalar variant	-
101	11	0	VFNMS - Double-precision scalar variant	-
101	11	1	VFNMA - Double-precision scalar variant	-
110	01	0	VFMA - Half-precision scalar variant	FEAT_FP16
110	01	1	VFMS - Half-precision scalar variant	FEAT_FP16
110	10	0	VFMA - Single-precision scalar variant	-
110	10	1	VFMS - Single-precision scalar variant	-
110	11	0	VFMA - Double-precision scalar variant	-
110	11	1	VFMS - Double-precision scalar variant	-

F4.1.18 Unconditional instructions

This section describes the encoding of the Unconditional instructions group. The encodings in this section are decoded from A32 instruction set encoding.

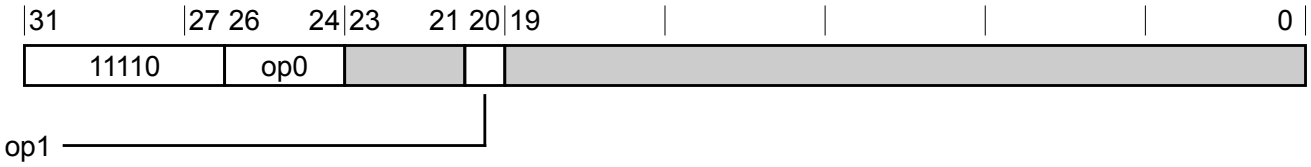

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-69 Encoding table for the Unconditional instructions group

Decode fields		Decode group or instruction page
op0	op1	
00x	x	Miscellaneous
01x	x	Advanced SIMD data-processing
1xx	1	Memory hints and barriers
100	0	Advanced SIMD element or structure load/store
101	0	Unallocated.
11x	0	Unallocated.

F4.1.19 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from Unconditional instructions.

31 – 25	24 – 20	19 – 8	7 – 4	3 – 0
1111000	op0		op1	

Table F4-70 Encoding table for the Miscellaneous group

Decode fields		Decode group or instruction page	Feature
op0	op1		
0xxxx	xxxx	Unallocated.	-
10000	xx0x	Change Process State	-
10001	1000	Unallocated.	-
10001	x100	Unallocated.	-
10001	xx01	Unallocated.	-
10001	0000	SETPAN	FEAT_PAN
1000x	0111	Unallocated.	-
10010	0111	CONSTRAINED UNPREDICTABLE	-
10011	0111	Unallocated.	-
1001x	xx0x	Unallocated.	-
100xx	0011	Unallocated.	-
100xx	0x10	Unallocated.	-
100xx	1x1x	Unallocated.	-
101xx	xxxx	Unallocated.	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

11xxx	xxxx	Unallocated.	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.

F4.1.19.1 Change Process State

This section describes the encoding of the Change Process State instruction class. The encodings in this section are decoded from Miscellaneous.

31	30	29	28	27	26	25	24	23	22	21	20	19 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	op	(0)	(0)	(0)	(0)	(0)	(0)	E	A	I	F	0	mode	

Table F4-71

Decode fields						Instruction page
imod	**M**	**op**	**I**	**F**	**mode**	
xx	x	1	0	0	0xxxx	SETEND
00	1	0	x	x	xxxxx	CPS, CPSID, CPSIE - Change mode variant
10	x	0	x	x	xxxxx	CPS, CPSID, CPSIE - Interrupt enable and change mode variant
xx	x	1	0	0	1xxxx	Unallocated.
xx	x	1	0	1	xxxxx	Unallocated.
xx	x	1	1	x	xxxxx	Unallocated.
11	x	0	x	x	xxxxx	CPS, CPSID, CPSIE - Interrupt disable and change mode variant

F4.1.20 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from Unconditional instructions.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding

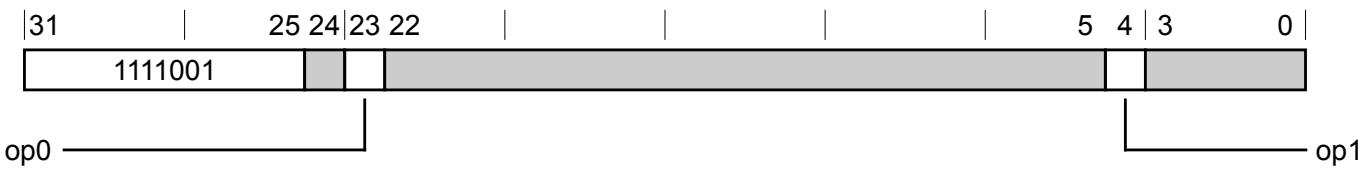

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-72 Encoding table for the Advanced SIMD data-processing group

Decode fields		Decode group or instruction page
op0	op1	
0	x	Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

F4.1.20.1 Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from Advanced SIMD data-processing.

31 30 29 28	27 26 25	24	23	22	21 20	19 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1	U	0	D	size	Vn	Vd	opc	N	Q	M	o1	Vm

Table F4-73

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
0	0x	1100	x	1	VFMA	-
0	0x	1101	x	0	VADD (floating-point)	-
0	0x	1101	x	1	VMLA (floating-point)	-
0	0x	1110	x	0	VCEQ (register) - A2	-
0	0x	1111	x	0	VMAX (floating-point)	-
0	0x	1111	x	1	VRECPS	-
x	xx	0000	x	0	VHADD	-
0	00	0001	x	1	VAND (register)	-
x	xx	0000	x	1	VQADD	-
x	xx	0001	x	0	VRHADD	-
0	00	1100	x	0	SHA1C	FEAT_SHA1
x	xx	0010	x	0	VHSUB	-
0	01	0001	x	1	VBIC (register)	-
x	xx	0010	x	1	VQSUB	-
x	xx	0011	x	0	VCGT (register) - A1	-
x	xx	0011	x	1	VCGE (register) - A1	-
0	01	1100	x	0	SHA1P	FEAT_SHA1
0	1x	1100	x	1	VFMS	-
0	1x	1101	x	0	VSUB (floating-point)	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
0	1x	1101	x	1	VMLS (floating-point)	-
0	1x	1110	x	0	Unallocated.	-
0	1x	1111	x	0	VMIN (floating-point)	-
0	1x	1111	x	1	VRSQRTS	-
x	xx	0100	x	0	VSHL (register)	-
0	xx	1000	x	0	VADD (integer)	-
0	10	0001	x	1	VORR (register)	-
0	xx	1000	x	1	VTST	-
x	xx	0100	x	1	VQSHL (register)	-
0	xx	1001	x	0	VMLA (integer)	-
x	xx	0101	x	0	VRSHL	-
x	xx	0101	x	1	VQRSHL	-
0	xx	1011	x	0	VQDMULH	-
0	10	1100	x	0	SHA1M	FEAT_SHA1
0	xx	1011	x	1	VPADD (integer)	-
x	xx	0110	x	0	VMAX (integer)	-
0	11	0001	x	1	VORN (register)	-
x	xx	0110	x	1	VMIN (integer)	-
x	xx	0111	x	0	VABD (integer)	-
x	xx	0111	x	1	VABA	-
0	11	1100	x	0	SHA1SU0	FEAT_SHA1
1	0x	1101	x	0	VPADD (floating-point)	-
1	0x	1101	x	1	VMUL (floating-point)	-
1	0x	1110	x	0	VCGE (register) - A2	-
1	0x	1110	x	1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	x	1	VMAXNM	-
1	00	0001	x	1	VEOR	-
x	xx	1001	x	1	VMUL (integer and polynomial)	-
1	00	1100	x	0	SHA256H	FEAT_SHA256
x	xx	1010	0	0	VPMAX (integer)	-
1	01	0001	x	1	VBSL	-
x	xx	1010	0	1	VPMIN (integer)	-
x	xx	1010	1	x	Unallocated.	-
1	01	1100	x	0	SHA256H2	FEAT_SHA256
1	1x	1101	x	0	VABD (floating-point)	-
1	1x	1110	x	0	VCGT (register) - A2	-
1	1x	1110	x	1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
1	1x	1111	x	1	VMINNM	-
1	xx	1000	x	0	VSUB (integer)	-
1	10	0001	x	1	VBIT	-
1	xx	1000	x	1	VCEQ (register) - A1	-
1	xx	1001	x	0	VMLS (integer)	-
1	xx	1011	x	0	VQRDMULH	-
1	10	1100	x	0	SHA256SU1	FEAT_SHA256
1	xx	1011	x	1	VQRDMLAH	FEAT_RDM
1	11	0001	x	1	VBIF	-
1	xx	1100	x	1	VQRDMLSH	FEAT_RDM
1	xx	1111	1	0	Unallocated.	-

F4.1.21 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from Advanced SIMD data-processing.

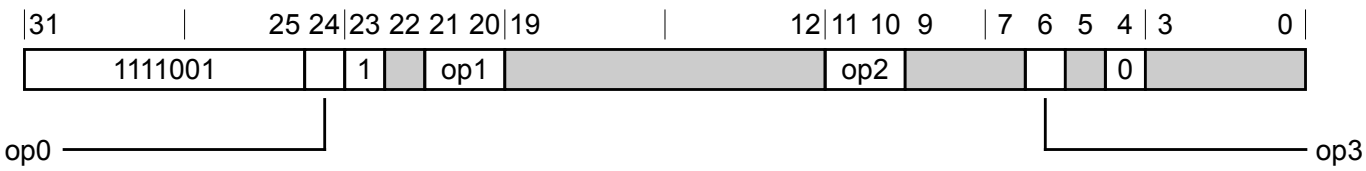

Table F4-74 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	11	xx	x	VEXT (byte elements)
1	11	0x	x	Advanced SIMD two registers misc
1	11	10	x	VTBL, VTBX
1	11	11	x	Advanced SIMD duplicate (scalar)
x	!= 11	xx	0	Advanced SIMD three registers of different lengths
x	!= 11	xx	1	Advanced SIMD two registers and a scalar

F4.1.21.1 Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30 29 28	27 26 25 24	23 22	21 20	19 18 17 16	15 12	11 10	7	6	5	4	3 0
1 1 1 1	0 0 1 1 1	D	1 1	size	opc1	Vd	0	opc2	Q	M	0	Vm

Table F4-75

Decode fields				Instruction page	Feature
size	**opc1**	**opc2**	**Q**		
xx	00	0000	x	VREV64	-
xx	00	0001	x	VREV32	-
xx	00	0010	x	VREV16	-
xx	00	0011	x	Unallocated.	-
xx	00	010x	x	VPADDL	-
xx	00	0110	0	AESE	FEAT_AES
xx	00	0110	1	AESD	FEAT_AES
xx	00	0111	0	AESMC	FEAT_AES
xx	00	0111	1	AESIMC	FEAT_AES
xx	00	1000	x	VCLS	-
00	10	0000	x	VSWP	-
xx	00	1001	x	VCLZ	-
xx	00	1010	x	VCNT	-
xx	00	1011	x	VMVN (register)	-
00	10	1100	1	Unallocated.	-
xx	00	110x	x	VPADAL	-
xx	00	1110	x	VQABS	-
xx	00	1111	x	VQNEG	-
xx	01	x000	x	VCGT (immediate #0)	-
xx	01	x001	x	VCGE (immediate #0)	-
xx	01	x010	x	VCEQ (immediate #0)	-
xx	01	x011	x	VCLE (immediate #0)	-
xx	01	x100	x	VCLT (immediate #0)	-
xx	01	x110	x	VABS	-
xx	01	x111	x	VNEG	-
xx	01	0101	1	SHA1H	FEAT_SHA1
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
xx	10	0001	x	VTRN	-
xx	10	0010	x	VUZP	-
xx	10	0011	x	VZIP	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page	Feature
size	opc1	opc2	Q		
xx	10	0100	0	VMOVN	-
xx	10	0100	1	VQMOVN, VQMOVUN - Unsigned result variant	-
xx	10	0101	x	VQMOVN, VQMOVUN - Signed result variant	-
xx	10	0110	0	VSHLL	-
xx	10	0111	0	SHA1SU1	FEAT_SHA1
xx	10	0111	1	SHA256SU0	FEAT_SHA256
xx	10	1000	x	VRINTN (Advanced SIMD)	-
xx	10	1001	x	VRINTX (Advanced SIMD)	-
xx	10	1010	x	VRINTA (Advanced SIMD)	-
xx	10	1011	x	VRINTZ (Advanced SIMD)	-
10	10	1100	1	Unallocated.	-
xx	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant	-
xx	10	1101	x	VRINTM (Advanced SIMD)	-
xx	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant	-
xx	10	1110	1	Unallocated.	-
xx	10	1111	x	VRINTP (Advanced SIMD)	-
xx	11	000x	x	VCVTA (Advanced SIMD)	-
xx	11	001x	x	VCVTN (Advanced SIMD)	-
xx	11	010x	x	VCVTP (Advanced SIMD)	-
xx	11	011x	x	VCVTM (Advanced SIMD)	-
xx	11	10x0	x	VRECPE	-
xx	11	10x1	x	VRSQRTE	-
11	10	1100	1	Unallocated.	-
xx	11	11xx	x	VCVT (between floating-point and integer, Advanced SIMD)	-

F4.1.21.2 Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

31 30 29 28	27 26 25 24	23 22	21 20	19 16	15 12	11 10	9	7	6	5	4	3 0
1 1 1 1	0 0 1 1 1	D	1 1	imm4	Vd	1 1	opc		Q	M	0	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-76

Decode fields opc	Instruction page
000	VDUP (scalar)
001	Unallocated.
01x	Unallocated.
1xx	Unallocated.

F4.1.21.3 Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

31 30 29 28	27 26 25	24	23	22	21 20	19 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1	U	1	D	!=11 size	Vn	Vd	opc	N	0	M	0	Vm

Table F4-77

Decode fields U	 opc	Instruction page
x	0000	VADDL
x	0001	VADDW
x	0010	VSUBL
0	0100	VADDHN
x	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
x	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
x	0111	VABDL (integer)
x	1000	VMLAL (integer)
x	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
x	11x0	VMULL (integer and polynomial)
1	1001	Unallocated.
1	1011	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields		Instruction page
U	opc	
1	1101	Unallocated.
x	1111	Unallocated.

F4.1.21.4 Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
1	1	1	1	0	0	1	Q	1	D	size != 11		Vn	Vd	opc	N	1	M	0	Vm

Table F4-78

Decode fields		Instruction page	Feature
Q	opc		
x	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
x	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
x	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
x	0110	VMLSL (by scalar)	-
x	100x	VMUL (by scalar)	-
1	0011	Unallocated.	-
x	1010	VMULL (by scalar)	-
1	0111	Unallocated.	-
x	1100	VQDMULH	-
x	1101	VQRDMULH	-
1	1011	Unallocated.	-
x	1110	VQRDMLAH	FEAT_RDM
x	1111	VQRDMLSH	FEAT_RDM

F4.1.22 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from Advanced SIMD data-processing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 … 25	24	23	22	21 … 7	6	5	4	3 … 0
1111001		1		op0			1	

Table F4-79 Encoding table for the Advanced SIMD shifts and immediate generation group

Decode fields op0	Decode group or instruction page
`000xxxxxxxxxxx0`	Advanced SIMD one register and modified immediate
`!= 000xxxxxxxxxxx0`	Advanced SIMD two registers and shift amount

F4.1.22.1 Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.

31 30 29 28	27 26 25	24	23	22	21 20 19	18 … 16	15 … 12	11 … 8	7	6	5	4	3 … 0
1 1 1 1	0 0 1	i	1	D	0 0 0	imm3	Vd	cmode	0	Q	op	1	imm4

Table F4-80

Decode fields cmode	op	Instruction page
`0xx0`	`0`	VMOV (immediate) - A1
`0xx0`	`1`	VMVN (immediate) - A1
`0xx1`	`0`	VORR (immediate) - A1
`0xx1`	`1`	VBIC (immediate) - A1
`10x0`	`0`	VMOV (immediate) - A3
`10x0`	`1`	VMVN (immediate) - A2
`10x1`	`0`	VORR (immediate) - A2
`10x1`	`1`	VBIC (immediate) - A2
`11xx`	`0`	VMOV (immediate) - A4
`110x`	`1`	VMVN (immediate) - A3
`1110`	`1`	VMOV (immediate) - A5
`1111`	`1`	Unallocated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1.22.2 Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.

31 30 29 28	27 26 25 24	23	22	21	19 18 16	15 12	11 8	7	6	5	4	3 0
1 1 1 1	0 0 1	U	1	D	imm3H	imm3L	Vd	opc	L	Q	M	1	Vm

This decode also imposes the constraint:

- `!(imm3H == '000' && L == '0').`

Table F4-81

Decode fields						Instruction page
U	**imm3H**	**imm3L**	**opc**	**L**	**Q**	
x	xxx	xxx	1xxx	1	x	Unallocated.
x	000	xxx	0000	1	0	VSHR 64-bit SIMD vector
x	000	xxx	0000	1	1	VSHR 128-bit SIMD vector
x	000	xxx	0001	1	0	VSRA 64-bit SIMD vector
x	000	xxx	0001	1	1	VSRA 128-bit SIMD vector
x	000	xxx	0010	1	0	VRSHR 64-bit SIMD vector
x	000	xxx	0010	1	1	VRSHR 128-bit SIMD vector
x	000	xxx	0011	1	0	VRSRA 64-bit SIMD vector
x	000	xxx	0011	1	1	VRSRA 128-bit SIMD vector
x	000	xxx	0111	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	000	xxx	0111	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	001	000	1010	0	0	VMOVL
x	001	!= 000	1010	0	0	VSHLL
x	010	000	1010	0	0	VMOVL
x	010	!= 000	1010	0	0	VSHLL
x	011	xxx	1010	0	0	VSHLL
x	100	000	1010	0	0	VMOVL
x	100	!= 000	1010	0	0	VSHLL
x	101	xxx	1010	0	0	VSHLL
x	11x	xxx	1010	0	0	VSHLL
x	!= 000	xxx	0000	x	0	VSHR 64-bit SIMD vector
x	!= 000	xxx	0000	x	1	VSHR 128-bit SIMD vector
x	!= 000	xxx	0001	x	0	VSRA 64-bit SIMD vector
x	!= 000	xxx	0001	x	1	VSRA 128-bit SIMD vector

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
x	!= 000	xxx	0010	x	0	VRSHR 64-bit SIMD vector
x	!= 000	xxx	0010	x	1	VRSHR 128-bit SIMD vector
x	!= 000	xxx	0011	x	0	VRSRA 64-bit SIMD vector
x	!= 000	xxx	0011	x	1	VRSRA 128-bit SIMD vector
x	!= 000	xxx	0111	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	!= 000	xxx	0111	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	!= 000	xxx	1001	0	0	VQSHRN, VQSHRUN — Signed result
x	!= 000	xxx	1001	0	1	VQRSHRN, VQRSHRUN — Signed result
x	!= 000	xxx	1010	0	1	Unallocated.
x	!= 000	xxx	1011	0	x	Unallocated.
x	!= 000	xxx	11xx	0	0	VCVT (between floating-point and fixed-point, Advanced SIMD) — 64-bit SIMD vector
x	!= 000	xxx	11xx	0	1	VCVT (between floating-point and fixed-point, Advanced SIMD) — 128-bit SIMD vector
0	xxx	xxx	01x0	1	x	Unallocated.
0	000	xxx	0101	1	0	VSHL (immediate) — 64-bit SIMD vector
0	000	xxx	0101	1	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	01x0	0	x	Unallocated.
0	!= 000	xxx	0101	x	0	VSHL (immediate) — 64-bit SIMD vector
0	!= 000	xxx	0101	x	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	1000	0	0	VSHRN
0	!= 000	xxx	1000	0	1	VRSHRN
1	000	xxx	0100	1	0	VSRI — 64-bit SIMD vector
1	000	xxx	0100	1	1	VSRI — 128-bit SIMD vector
1	000	xxx	0101	1	0	VSLI — 64-bit SIMD vector
1	000	xxx	0101	1	1	VSLI — 128-bit SIMD vector
1	000	xxx	0110	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	000	xxx	0110	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	0100	x	0	VSRI — 64-bit SIMD vector
1	!= 000	xxx	0100	x	1	VSRI — 128-bit SIMD vector
1	!= 000	xxx	0101	x	0	VSLI — 64-bit SIMD vector
1	!= 000	xxx	0101	x	1	VSLI — 128-bit SIMD vector
1	!= 000	xxx	0110	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	!= 000	xxx	0110	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	1000	0	0	VQSHRN, VQSHRUN — Unsigned result
1	!= 000	xxx	1000	0	1	VQRSHRN, VQRSHRUN — Unsigned result

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1.23 Memory hints and barriers

This section describes the encoding of the Memory hints and barriers group. The encodings in this section are decoded from Unconditional instructions.

Table F4-82 Encoding table for the Memory hints and barriers group

Decode fields		Decode group or instruction page
op0	**op1**	
00xx1	x	CONSTRAINED UNPREDICTABLE
01001	x	CONSTRAINED UNPREDICTABLE
01011	x	Barriers
011x1	x	CONSTRAINED UNPREDICTABLE
0xxx0	x	Preload (immediate)
1xxx0	0	Preload (register)
1xxx1	0	CONSTRAINED UNPREDICTABLE
1xxxx	1	Unallocated.

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.

F4.1.23.1 Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from Memory hints and barriers.

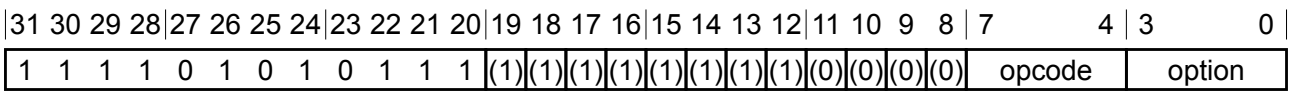

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-83

Decode fields		Instruction page	Feature
opcode	**option**		
0000	xxxx	CONSTRAINED UNPREDICTABLE	-
0001	xxxx	CLREX	-
001x	xxxx	CONSTRAINED UNPREDICTABLE	-
0100	!= 0x00	DSB	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101	xxxx	DMB	-
0110	xxxx	ISB	-
0111	xxxx	SB	FEAT_SB
1xxx	xxxx	CONSTRAINED UNPREDICTABLE	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.

F4.1.23.2 Preload (immediate)

This section describes the encoding of the Preload (immediate) instruction class. The encodings in this section are decoded from Memory hints and barriers.

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15	14	13	12	11 0
1 1 1 1	0 1 0 D	U	R	0 1	Rn	(1)	(1)	(1)	(1)	imm12

Table F4-84

Decode fields			Instruction page
D	**R**	**Rn**	
0	0	xxxx	Reserved hint, behaves as NOP.
0	1	xxxx	PLI (immediate, literal)
1	x	1111	PLD (literal)
1	0	!= 1111	PLD, PLDW (immediate) - Preload write variant
1	1	!= 1111	PLD, PLDW (immediate) - Preload read variant

F4.1.23.3 Preload (register)

This section describes the encoding of the Preload (register) instruction class. The encodings in this section are decoded from Memory hints and barriers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30 29 28	27 26 25 24	23	22	21 20	19 16	15	14	13	12	11 7	6 5	4	3 0
1 1 1 1	0 1 1 D	U	o2	0 1	Rn	(1)	(1)	(1)	(1)	imm5	stype	0	Rm

Table F4-85

Decode fields			Instruction page
D	**o2**	**imm5:stype**	
0	0	xxxxxxx	Reserved hint, behaves as NOP.
0	1	!= 0000011	PLI (register) - Shift or rotate by value variant
0	1	0000011	PLI (register) - Rotate right with extend variant
1	0	!= 0000011	PLD, PLDW (register) - Preload write, optional shift or rotate variant
1	0	0000011	PLD, PLDW (register) - Preload write, rotate right with extend variant
1	1	!= 0000011	PLD, PLDW (register) - Preload read, optional shift or rotate variant
1	1	0000011	PLD, PLDW (register) - Preload read, rotate right with extend variant

F4.1.24 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from Unconditional instructions.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding

31 24	23	22 21	20	19 12	11 10 9	0
11110100			0		op1	

op0

Table F4-86 Encoding table for the Advanced SIMD element or structure load/store group

Decode fields		Decode group or instruction page
op0	**op1**	
0	xx	Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1.24.1 Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 8	7 6	5 4	3 0
1 1 1 1	0 1 0 0	0 D L 0	Rn	Vd	itype	size	align	Rm

Table F4-87

Decode fields			Instruction page
L	**itype**	**Rm**	
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields			Instruction page
L	**itype**	**Rm**	
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
x	1011	xxxx	Unallocated.
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
x	11xx	xxxx	Unallocated.
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

F4.1.24.2 Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10 9 8	7 6 5 4	3 0
1 1 1 1	0 1 0 0 1	D L 0	Rn	Vd	1 1 N	size T a	Rm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table F4-88

Decode fields				Instruction page
L	**N**	**a**	**Rm**	
0	xx	x	xxxx	Unallocated.
1	00	x	!= 11x1	VLD1 (single element to all lanes)
1	00	x	1101	VLD1 (single element to all lanes)
1	00	x	1111	VLD1 (single element to all lanes)
1	01	x	!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01	x	1101	VLD2 (single 2-element structure to all lanes)
1	01	x	1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1	xxxx	Unallocated.
1	11	x	!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11	x	1101	VLD4 (single 4-element structure to all lanes)
1	11	x	1111	VLD4 (single 4-element structure to all lanes)

F4.1.24.3 Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 10	9 8	7 4	3 0
1 1 1 1	0 1 0 0 1	D L 0	Rn	Vd	!=11 size	N	index_align	Rm

Table F4-89

Decode fields				Instruction page
L	**size**	**N**	**Rm**	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
L	size	N	Rm	
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode fields				Instruction page
L	size	N	Rm	
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All A32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit A32 instruction decode structure. Table F4-90 shows these instruction groups, and where each group is decoded from the overall A32 decode structure:

Table F4-90 Advanced SIMD and floating-point instructions in the A32 decode structure

Advanced SIMD and floating-point instruction group	A32 decode is from
Advanced SIMD and System register load/store and 64-bit move	System register access, Advanced SIMD, floating-point, and Supervisor call
Floating-point data-processing	System register access, Advanced SIMD, floating-point, and Supervisor call
Advanced SIMD and System register 32-bit move	System register access, Advanced SIMD, floating-point, and Supervisor call
Advanced SIMD data-processing	Unconditional instructions
Advanced SIMD element or structure load/store	Unconditional instructions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F5
T32 and A32 Base Instruction Set Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- Alphabetical list of T32 and A32 base instruction set instructions.
- Encoding and use of banked register transfer instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1 Alphabetical list of T32 and A32 base instruction set instructions

This section lists every instruction in the T32 and A32 base instruction sets. For details of the format used see Format of instruction descriptions.

This section is formatted so that each instruction description starts on a new page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.1 ADC, ADCS (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

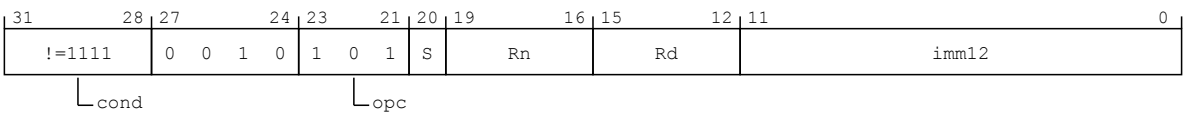

Encoding for the ADC variant

Applies when `(S == 0)`

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ADCS variant

Applies when `(S == 1)`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

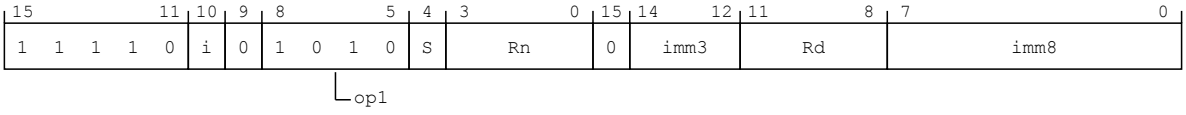

Encoding for the ADC variant

Applies when `(S == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ADCS variant

Applies when `(S == 1)`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ADC” and “A1 ADCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 ADC” and “T1 ADCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ADC” and “A1 ADCS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 ADC” and “T1 ADCS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 ADC” and “A1 ADCS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1 ADC” and “T1 ADCS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.2 ADC, ADCS (register)

Add with Carry (register) adds a register value, the Carry flag value, and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	1 0 1	S	Rn	Rd	imm5	stype	0	Rm
cond		opc							

Encoding for the ADC, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADC, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ADCS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADCS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

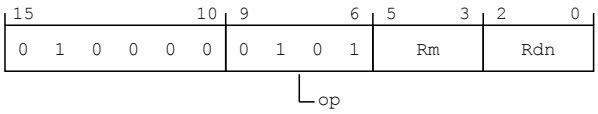

Encoding

```
ADC<c>{<q>}  {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
ADCS{<q>}  {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

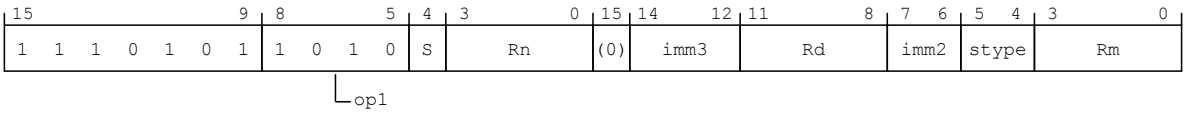

Encoding for the ADC, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADC, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ADC<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the ADCS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADCS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ADCS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ADC, rotate right with extend”, “A1 ADC, shift or rotate by value”, “A1 ADCS, rotate right with extend”, and “A1 ADCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2 ADC, rotate right with extend”, “T2 ADC, shift or rotate by value”, “T2 ADCS, rotate right with extend”, and “T2 ADCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ADC, rotate right with extend”, “A1 ADC, shift or rotate by value”, “A1 ADCS, rotate right with extend”, and “A1 ADCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 ADC, rotate right with extend”, “T2 ADC, shift or rotate by value”, “T2 ADCS, rotate right with extend”, and “T2 ADCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 ADC, rotate right with extend”, “A1 ADC, shift or rotate by value”, “A1 ADCS, rotate right with extend”, and “A1 ADCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 ADC, rotate right with extend”, “T2 ADC, shift or rotate by value”, “T2 ADCS, rotate right with extend”, and “T2 ADCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ADC, shift or rotate by value” and “A1 ADCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 ADC, shift or rotate by value” and “T2 ADCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if `ADCS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ADCS <Rd>, <Rn>` had been written.
- Inside an IT block, if `ADC<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ADC<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the .W qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.3 ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0 0	1 0 1	S	Rn	Rd	Rs	0	stype	1	Rm
cond		opc								

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
ADCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
ADC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.4 ADD (immediate, to PC)

Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is a pseudo-instruction of ADR. This means:

- The encodings in this description are named to match the encodings of ADR.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ADR gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T3).

A1

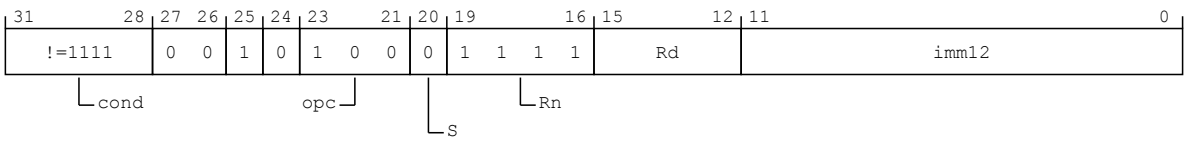

Encoding for the A1 variant

```
ADD{<c>}{<q>}  <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>}  <Rd>, <label>
```

T1

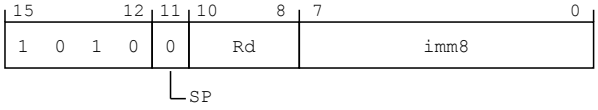

Encoding for the T1 variant

```
ADD{<c>}{<q>}  <Rd>, PC, #<imm8>
```

is equivalent to

```
ADR{<c>}{<q>}  <Rd>, <label>
```

T3

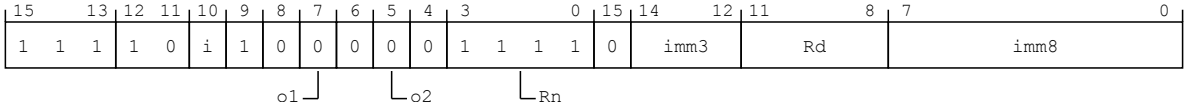

Encoding for the T3 variant

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
ADD{<c>}{<q>}  <Rd>, PC, #<imm12>
```

```
ADDW{<c>}{<q>}  <Rd>, PC, #<imm12> // (UInt(Rd) < 8 && UInt(i :: imm3 :: imm8) < 256)
```

is equivalent to

```
ADR{<c>}{<q>}  <Rd>, <label>
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1” variant: is the general-purpose destination register, encoded in the “Rd” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T1” and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field.

<const>

An immediate value. See Modified immediate constants in A32 instructions for the range of values.

<imm8>

Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the “imm8” field as <imm8>/4.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

The description of ADR gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.5 ADD, ADDS (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, and T4).

A1

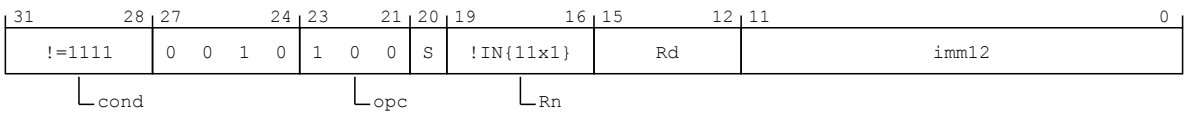

Encoding for the ADD variant

Applies when `(S == 0 && Rn != 11x1)`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ADDS variant

Applies when `(S == 1 && Rn != 1101)`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' && S == '0' then See("ADR"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

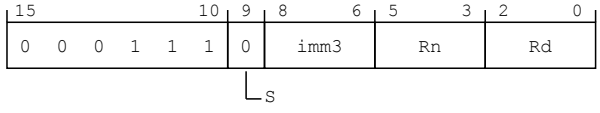

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
ADD<c>{<q>}  <Rd>, <Rn>, #<imm3> // (InITBlock())
```

```
ADDS{<q>}  <Rd>, <Rn>, #<imm3> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm3);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

op

Encoding

```
ADD<c>{<q>}  <Rdn>, #<imm8> // (InITBlock() && UInt(imm8) < 8)
```

```
ADD<c>{<q>}  {<Rdn>, }<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

```
ADDS{<q>}  <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)
```

```
ADDS{<q>}  {<Rdn>, }<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	!=1101				0	imm3			Rd				imm8							

op1 Rn

Encoding for the ADD variant

Applies when `(S == 0)`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

```
ADD<c>.W  {<Rd>, }<Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1
    ↪or T2)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the ADDS variant

Applies when `(S == 1 && Rd != 1111)`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

```
ADDS.W  {<Rd>, }<Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
     ↪T2)
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (immediate)"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

T4

15–11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	1	0	0	0	0	0	!=11x1	0	imm3	Rd	imm8
				o1		o2		Rn				

Encoding

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
```

```
ADDW{<c>}{<q>}  {<Rd>, }<Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)
```

Decode for this encoding

```
if Rn == '1111' then See("ADR"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the "A1 ADD" and "A1 ADDS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction.

For the “T1”, “T3 ADD”, “T3 ADDS”, and “T4” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ADD”, “A1 ADDS”, and “T4” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see ADD (SP plus immediate). If the PC is used, see ADR.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

For the “T3 ADD” and “T3 ADDS” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see ADD (SP plus immediate).

<const>

For the “A1 ADD” and “A1 ADDS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T3 ADD” and “T3 ADDS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

<imm3>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “imm3” field.

<Rdn>

Is the general-purpose source and destination register, encoded in the “Rdn” field.

<imm8>

Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
        if d == 15 then          // Can only occur for A32 encoding
            if setflags then
                ALUExceptionReturn(result);
            else
                ALUWritePC(result);
            end;
        else
            R(d) = result;
            if setflags then
                PSTATE.[N,Z,C,V] = nzcv;
            end;
        end;
    end;
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ConditionPassed() then
        EncodingSpecificOperations();
        let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.6 ADD, ADDS (register)

Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	1 0 0	S	!=1101	Rd	imm5	stype	0	Rm
cond		opc		Rn					

Encoding for the ADD, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADD, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ADDS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

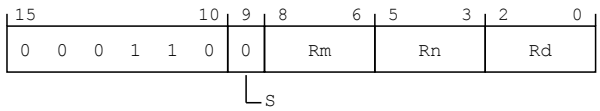

Encoding

```
ADD<c>{<q>}  <Rd>, <Rn>, <Rm> // (InITBlock())
```

```
ADDS{<q>}  {<Rd>, }<Rn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

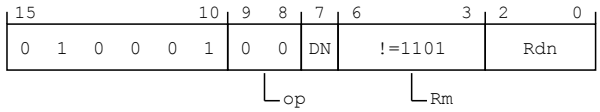

Encoding

Applies when `(!(DN == '1' && Rdn == '101'))`

```
ADD{<c>}{<q>}  {<Rdn>, }<Rdn>, <Rm>
```

```
ADD<c>{<q>}  <Rdn>, <Rm> // (Preferred syntax, Inside IT block)
```

Decode for this encoding

```
if (DN::Rdn) == '1101' || Rm == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(DN::Rdn);
let n : integer = d;
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if n == 15 && m == 15 then UnpredictableProcedure(); end;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	1 0 0 0 (op1)	S	!=1101 (Rn)	(0)	imm3	Rd	imm2	stype	Rm

Encoding for the ADD, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADD, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ADD<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

```
ADD{<c>}.W  {<Rd>, }<Rn>, <Rm> // (<Rd> == <Rn>, and <Rd>, <Rn>, <Rm> can be represented in T2)
```

Encoding for the ADDS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when `(S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ADDS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2)
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (register)"); end;
if Rn == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction.

For the “T1” variant: is the general-purpose destination register, encoded in the “Rd” field.

When used inside an IT block, `<Rd>` must be specified. When used outside an IT block, `<Rd>` is optional, and:

- If omitted, this register is the same as `<Rn>`.
- If present, encoding T1 is preferred to encoding T2.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used. If the SP is used, see ADD (SP plus register).

For the “T1” variant: is the first general-purpose source register, encoded in the “Rn” field.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. If the SP is used, see ADD (SP plus register).

<Rm>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ADD, shift or rotate by value” and “A1 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T3 ADD, shift or rotate by value” and “T3 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the general-purpose source and destination register, encoded in the “DN:Rdn” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

The assembler language allows `<Rdn>` to be specified once or twice in the assembler syntax. When used inside an IT block, and `<Rdn>` and `<Rm>` are in the range R0 to R7, `<Rdn>` must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when `<Rdn>` is specified once or twice.

Inside an IT block, if `ADD<c> <Rd>, <Rn>, <Rd>` cannot be assembled using encoding T1, it is assembled using encoding T2 as though `ADD<c> <Rd>, <Rn>` had been written. To prevent this happening, use the `.W` qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.7 ADD, ADDS (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	1 0 0 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
ADDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
ADD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.8 ADD, ADDS (SP plus immediate)

Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, and T4).

A1

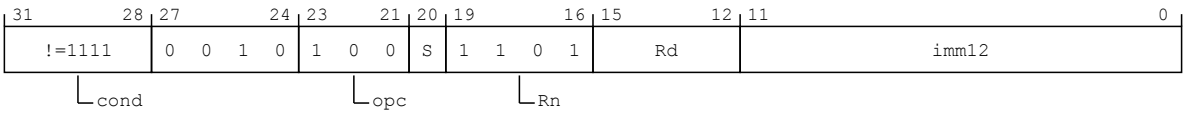

Encoding for the ADD variant

Applies when `(S == 0)`

```
ADD{<c>}{<q>}  {<Rd>, }SP, #<const>
```

Encoding for the ADDS variant

Applies when `(S == 1)`

```
ADDS{<c>}{<q>}  {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

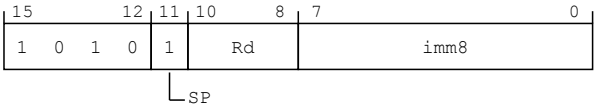

Encoding

```
ADD{<c>}{<q>}  <Rd>, SP, #<imm8>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
```

T2

15			12	11			8	7	6						0
1	0	1	1	0	0	0	0	0	imm7						
								S							

Encoding

```
ADD{<c>}{<q>}  {SP, }SP, #<imm7>
```

Decode for this encoding

```
let d : integer = 13;
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm7::'00');
```

T3

15				11	10	9	8				5	4	3			0	15	14	12	11	8	7	0
1	1	1	1	0	i	0	1	0	0	0		S	1	1	0	1	0	imm3		Rd		imm8	
								op1							Rn								

Encoding for the ADD variant

Applies when `(S == 0)`

```
ADD{<c>}{<q>}  {<Rd>, }SP, #<const>
```

```
ADD{<c>}.W  {<Rd>, }SP, #<const> // (<Rd>, <const> can be represented in T1 or T2)
```

Encoding for the ADDS variant

Applies when `(S == 1 && Rd != 1111)`

```
ADDS{<c>}{<q>}  {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (immediate)"); end;
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if d == 15 && !setflags then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T4

15–11	10	9–8	7	6	5	4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	1 0	0	0	0	0	1 1 0 1	0	imm3	Rd	imm8
			o1		o2		Rn				

Encoding

```
ADD{<c>}{<q>}  {<Rd>, }SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
```

```
ADDW{<c>}{<q>}  {<Rd>, }SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ADD” and “A1 ADDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1” variant: is the general-purpose destination register, encoded in the “Rd” field.

For the “T3 ADD”, “T3 ADDS”, and “T4” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<const>

For the “A1 ADD” and “A1 ADDS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T3 ADD” and “T3 ADDS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

<imm8>

Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the “imm8” field as <imm8>/4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SP

Is the stack pointer.

<imm7>

Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the “imm7” field as <imm7>/4.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), imm32, '0');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.9 ADD, ADDS (SP plus register)

Add to SP (register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	1 0 0	S	1 1 0 1	Rd	imm5	stype	0	Rm
cond		opc		Rn					

Encoding for the ADD, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
ADD{<c>}{<q>}  {<Rd>, }SP, <Rm> , RRX
```

Encoding for the ADD, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
ADD{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Encoding for the ADDS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
ADDS{<c>}{<q>}  {<Rd>, }SP, <Rm> , RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
ADDS{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

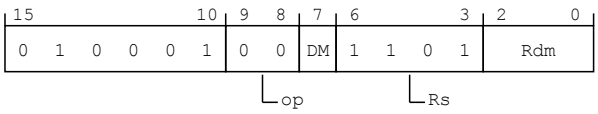

Encoding

```
ADD{<c>}{<q>}  {<Rdm>, }SP, <Rdm>
```

Decode for this encoding

```
let d : integer = UInt(DM::Rdm);
let m : integer = UInt(DM::Rdm);
let setflags : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

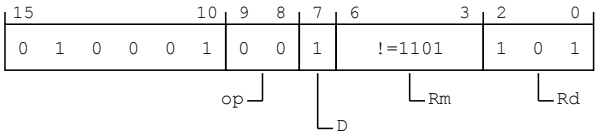

Encoding

```
ADD{<c>}{<q>}  {SP, }SP, <Rm>
```

Decode for this encoding

```
if Rm == '1101' then See("encoding T1"); end;
let d : integer = 13;
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T3

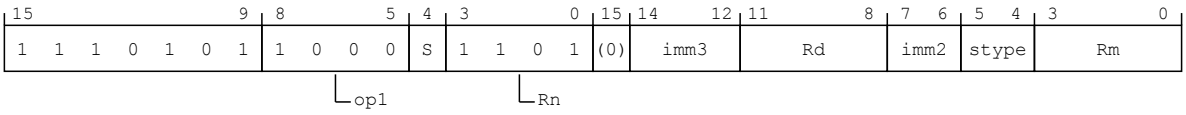

Encoding for the ADD, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ADD{<c>}{<q>}  {<Rd>, }SP, <Rm>, RRX
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the ADD, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ADD{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

```
ADD{<c>}.W  {<Rd>, }SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)
```

Encoding for the ADDS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)`

```
ADDS{<c>}{<q>}  {<Rd>, }SP, <Rm>, RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when `(S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ADDS{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (register)"); end;
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rm>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, “A1 ADDS, shift or rotate by value”, and “T2” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ADD, shift or rotate by value” and “A1 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T3 ADD, shift or rotate by value” and “T3 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdm>

Is the general-purpose destination and second source register, encoded in the “Rdm” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

SP

Is the stack pointer.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.10 ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the alias SUB (immediate, from PC).

This instruction is used by the pseudo-instruction ADD (immediate, to PC).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1, T2, and T3).

A1

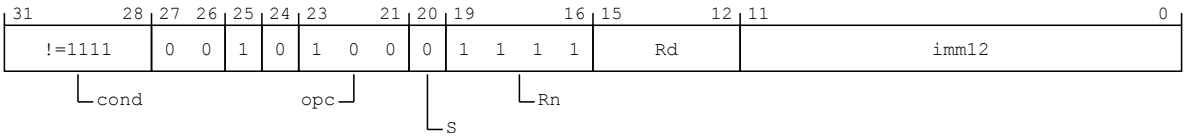

Encoding

```
ADR{<c>}{<q>}  <Rd>, <label>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = A32ExpandImm(imm12);
let add : boolean = TRUE;
```

A2

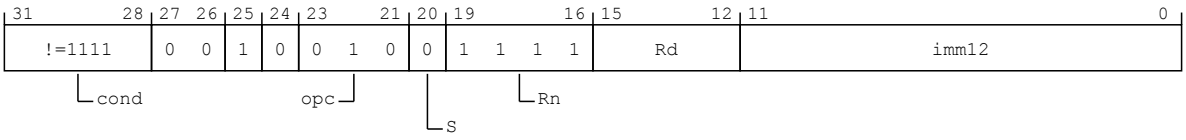

Encoding

```
ADR{<c>}{<q>}  <Rd>, <label>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = A32ExpandImm(imm12);
let add : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

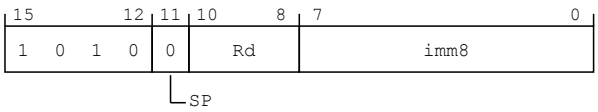

Encoding

```
ADR{<c>}{<q>}  <Rd>, <label>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = TRUE;
```

T2

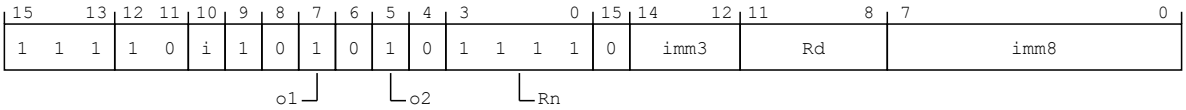

Encoding

```
ADR{<c>}{<q>}  <Rd>, <label>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
let add : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T3

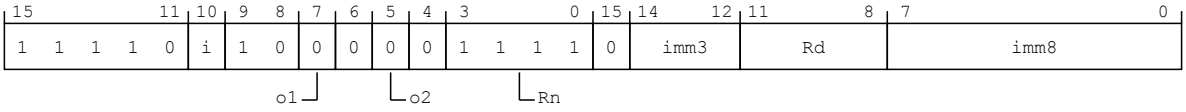

Encoding

```
ADR{<c>}{<q>}  <Rd>, <label>
```

```
ADR{<c>}.W  <Rd>, <label> // (<Rd>, <label> can be presented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
let add : boolean = TRUE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the "A1" and "A2" variants: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the "T1", "T2", and "T3" variants: is the general-purpose destination register, encoded in the "Rd" field.

<label>

For the "A1" and "A2" variants: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset.

If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are any of the constants described in Modified immediate constants in A32 instructions.

For the "T1" variant: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For the "T2" and "T3" variants: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset.

If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are 0-4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Of variant	Is preferred when
ADD (immediate, to PC)		Never
SUB (immediate, from PC)	A2	imm12 == '000000000000'
SUB (immediate, from PC)	T2	i : imm3 : imm8 == '000000000000'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                             else (AlignDownSize(PC32(),4) - imm32));
    if d == 15 then          // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R(d) = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.11 AND, ANDS (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

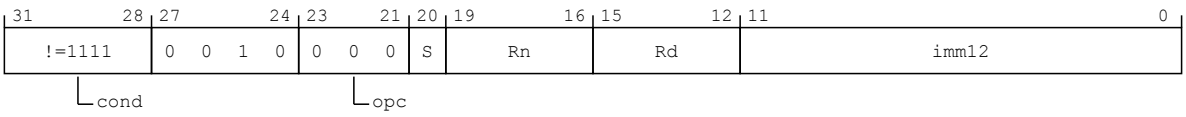

Encoding for the AND variant

Applies when `(S == 0)`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ANDS variant

Applies when `(S == 1)`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

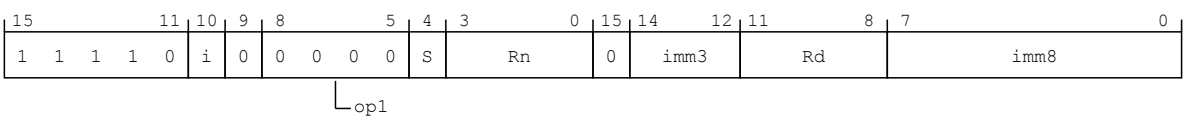

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the AND variant

Applies when `(S == 0)`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ANDS variant

Applies when `(S == 1 && Rd != 1111)`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TST (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 AND” and “A1 ANDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 AND” and “T1 ANDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 AND” and “A1 ANDS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 AND” and “T1 ANDS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 AND” and “A1 ANDS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1 AND” and “T1 ANDS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) AND imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.12 AND, ANDS (register)

Bitwise AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	0 0 0	S	Rn	Rd	imm5	stype	0	Rm
cond		opc							

Encoding for the AND, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the AND, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ANDS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ANDS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	0 0 0 0	Rm	Rdn
	op		

Encoding

```
AND<c>{<q>}  {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
ANDS{<q>}  {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

15–9	8–5	4	3–0	15	14–12	11–8	7 6	5 4	3–0
1 1 1 0 1 0 1	0 0 0 0	S	Rn	(0)	imm3	Rd	imm2	stype	Rm
	op1								

Encoding for the AND, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the AND, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
AND<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the ANDS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ANDS, shift or rotate by value variant

Applies when `(S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ANDS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TST (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 AND, rotate right with extend”, “A1 AND, shift or rotate by value”, “A1 ANDS, rotate right with extend”, and “A1 ANDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2 AND, rotate right with extend”, “T2 AND, shift or rotate by value”, “T2 ANDS, rotate right with extend”, and “T2 ANDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 AND, rotate right with extend”, “A1 AND, shift or rotate by value”, “A1 ANDS, rotate right with extend”, and “A1 ANDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 AND, rotate right with extend”, “T2 AND, shift or rotate by value”, “T2 ANDS, rotate right with extend”, and “T2 ANDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 AND, rotate right with extend”, “A1 AND, shift or rotate by value”, “A1 ANDS, rotate right with extend”, and “A1 ANDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 AND, rotate right with extend”, “T2 AND, shift or rotate by value”, “T2 ANDS, rotate right with extend”, and “T2 ANDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 AND, shift or rotate by value” and “A1 ANDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 AND, shift or rotate by value” and “T2 ANDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if `ANDS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ANDS <Rd>, <Rn>` had been written.
- Inside an IT block, if `AND<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `AND<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.13 AND, ANDS (register-shifted register)

Bitwise AND (register-shifted register) performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0 0	0 0 0	S	Rn	Rd	Rs	0	stype	1	Rm
cond		opc								

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
ANDS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
AND{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.14 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

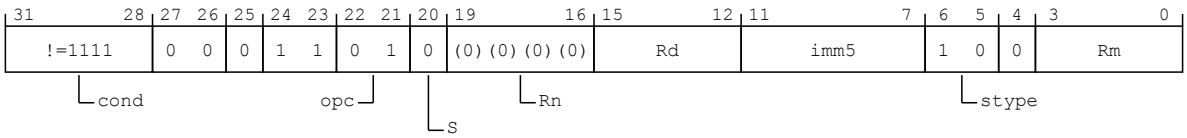

Encoding for the MOV, shift or rotate by value variant

```
ASR{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

T2

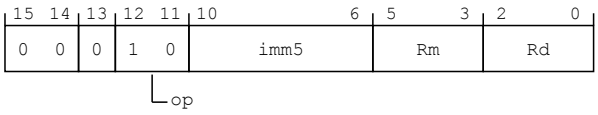

Encoding for the T2 variant

```
ASR<c>{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV<c>{<q>}  <Rd>, <Rm>, ASR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3

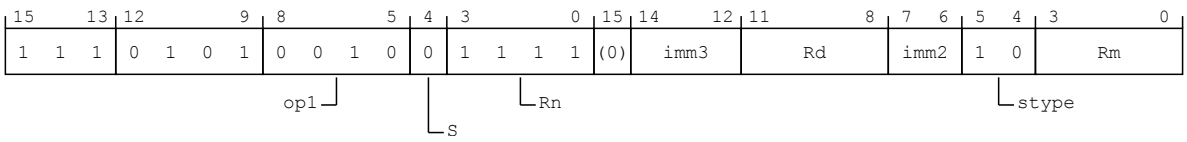

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the MOV, shift or rotate by value variant

```
ASR{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

```
ASR<c>.W  {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the "A1 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.

<Rm>

For the "A1 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.

<imm>

For the "A1 MOV, shift or rotate by value" and "T2" variants: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For the "T3 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.15 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

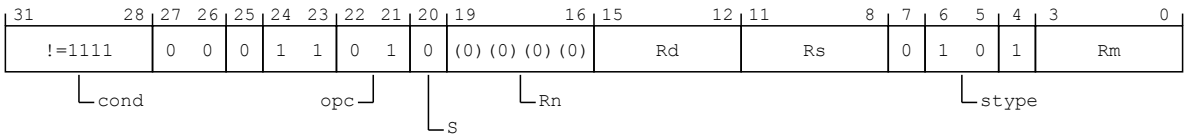

Encoding for the Not flag setting variant

```
ASR{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

T1

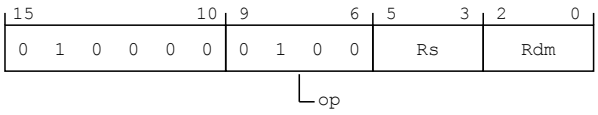

Encoding for the Arithmetic shift right variant

```
ASR<c>{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOV<c>{<q>}  <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

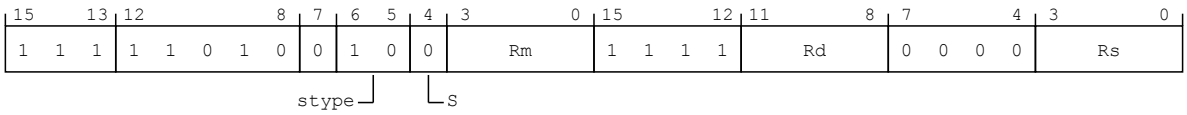

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Not flag setting variant

```
ASR{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
ASR<c>.W  {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.16 ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

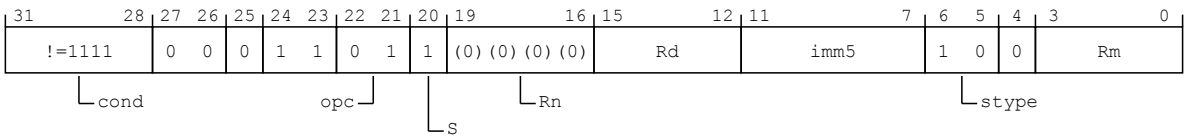

Encoding for the MOVS, shift or rotate by value variant

```
ASRS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

T2

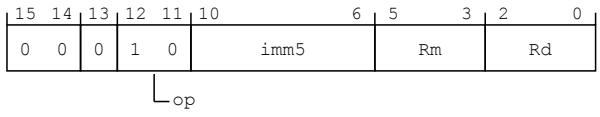

Encoding for the T2 variant

```
ASRS{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS<c>{<q>}  <Rd>, <Rm>, ASR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15–13	12–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 1	0 0 1 0	1	1 1 1 1	(0)	imm3	Rd	imm2	1 0	Rm
		op1	S	Rn					stype	

Encoding for the MOVS, shift or rotate by value variant

```
ASRS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

```
ASRS.W  {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.17 ASRS (register)

Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

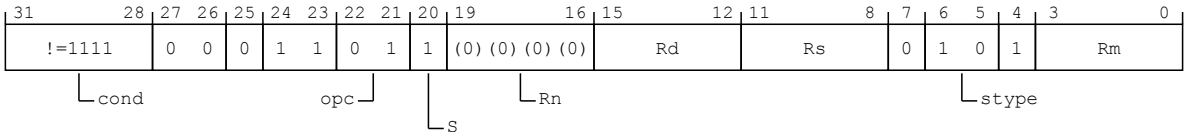

Encoding for the Flag setting variant

```
ASRS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

T1

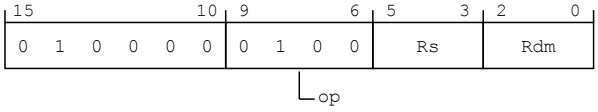

Encoding for the Arithmetic shift right variant

```
ASRS{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOVS{<q>}  <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12–8	7	6–5	4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1 0	0	1 0	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype	S					

Encoding for the Flag setting variant

```
ASRS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
ASRS.W  {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.18 B

Branch causes a branch to a target address.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, and T4).

A1

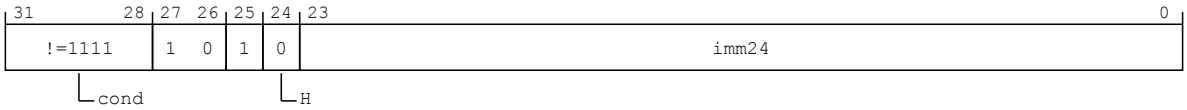

Encoding

```
B{<c>}{<q>}  <label>
```

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm24::'00');
```

T1

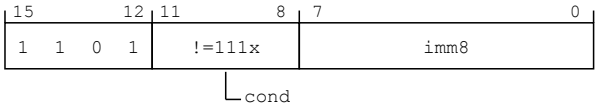

Encoding

```
B<c>{<q>}  <label> // (Not permitted in IT block)
```

Decode for this encoding

```
if cond == '1110' then See("UDF"); end;
if cond == '1111' then See("SVC"); end;
let imm32 : bits(32) = SignExtend{}(imm8::'0');
if InITBlock() then UnpredictableProcedure(); end;
```

T2

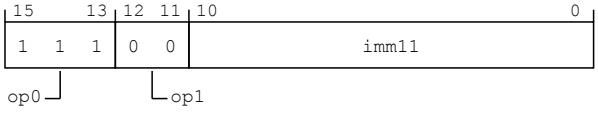

Encoding

```
B{<c>}{<q>}  <label> // (Outside or last in IT block)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm11::'0');
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T3

15				11	10	9	6	5	0	15	14	13	12	11	10	0
1	1	1	1	0	S	!=111x		imm6		1	0	J1	0	J2	imm11	

cond

Encoding

```
B<c>.W  <label> // (Not permitted in IT block, and <label> can be represented in T1)
```

```
B<c>{<q>}  <label> // (Not permitted in IT block)
```

Decode for this encoding

```
if cond[3:1] == '111' then See("Related encodings"); end;
let imm32 : bits(32) = SignExtend{}(S::J2::J1::imm6::imm11::'0');
if InITBlock() then UnpredictableProcedure(); end;
```

T4

15				11	10	9	0	15	14	13	12	11	10	0
1	1	1	1	0	S	imm10		1	0	J1	1	J2	imm11	

Encoding

```
B{<c>}{<q>}  <label>
```

```
B{<c>}.W  <label> // (<label> can be represented in T2)
```

Decode for this encoding

```
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{}(S::I1::I2::imm10::imm11::'0');
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Branches and miscellaneous control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1”, “T2”, and “T4” variants: see Standard assembler syntax fields.

For the “T1” and “T3” variants: see Standard assembler syntax fields. Must not be `AL` or omitted.

<q>

See Standard assembler syntax fields.

<label>

For the “A1” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For the “T1” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –256 to 254.

For the “T2” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –2048 to 2046.

For the “T3” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are even numbers in the range –1048576 to 1048574.

For the “T4” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are even numbers in the range –16777216 to 16777214.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC32() + imm32, BranchType_DIR);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.19 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–21	20–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0 1 1	1 1 1 0	msb	Rd	lsb	0 0	1	1 1 1 1 (Rn)

Encoding

```
BFC{<c>}{<q>}  <Rd>, #<lsb>, #<width>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(lsb);
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15–13	12–11	10	9	8	7–5	4	3–0	15	14–12	11–8	7–6	5	4–0
1 1 1	1 0	(0)	1	1	0 1 1 (op1)	0	1 1 1 1 (Rn)	0	imm3	Rd	imm2	(0)	msb

Encoding

```
BFC{<c>}{<q>}  <Rd>, #<lsb>, #<width>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<lsb>

For the “A1” variant: is the least significant bit to be cleared, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the “msb” field as <lsb>+<width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[msbit:lsbit] = Replicate{(msbit-lsbit)+1}('0');
    // Other bits of R[d] are unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.20 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–21	20–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0 1 1	1 1 1 0	msb	Rd	lsb	0 0	1	!=1111 (Rn)

Encoding

```
BFI{<c>}{<q>}  <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if Rn == '1111' then See("BFC"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(lsb);
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15–13	12–11	10	9	8	7–5	4	3–0	15	14–12	11–8	7–6	5	4–0
1 1 1	1 0	(0)	1	1	0 1 1 (op1)	0	!=1111 (Rn)	0	imm3	Rd	imm2	(0)	msb

Encoding

```
BFI{<c>}{<q>}  <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if Rn == '1111' then See("BFC"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “A1” variant: is the least significant destination bit, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the least significant destination bit, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the “msb” field as <lsb>+<width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[msbit:lsbit] = R(n)[(msbit-lsbit):0];
    // Other bits of R[d] are unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.21 BIC, BICS (immediate)

Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

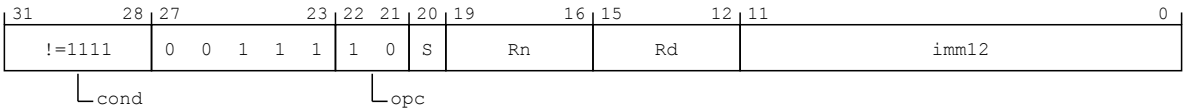

Encoding for the BIC variant

Applies when `(S == 0)`

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the BICS variant

Applies when `(S == 1)`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

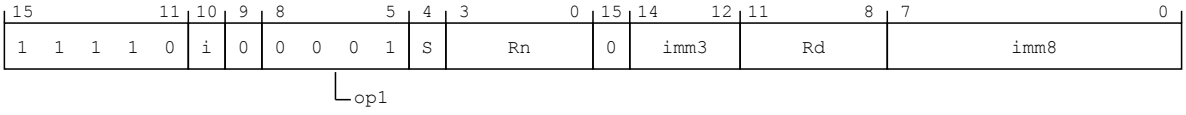

Encoding for the BIC variant

Applies when `(S == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the BICS variant

Applies when `(S == 1)`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 BIC” and “A1 BICS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 BIC” and “T1 BICS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 BIC” and “A1 BICS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 BIC” and “T1 BICS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 BIC” and “A1 BICS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1 BIC” and “T1 BICS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) AND NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.22 BIC, BICS (register)

Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 1	1 0 (opc)	S	Rn	Rd	imm5	stype	0	Rm

Encoding for the BIC, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the BIC, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the BICS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the BICS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	1 1 1 0	Rm	Rdn

op

Encoding

```
BIC<c>{<q>}  {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
BICS{<q>}  {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

15–9	8–5	4	3–0	15	14–12	11–8	7 6	5 4	3–0
1 1 1 0 1 0 1	0 0 0 1	S	Rn	(0)	imm3	Rd	imm2	stype	Rm

op1

Encoding for the BIC, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the BIC, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
BIC<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the BICS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the BICS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
BICS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 BIC, rotate right with extend”, “A1 BIC, shift or rotate by value”, “A1 BICS, rotate right with extend”, and “A1 BICS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2 BIC, rotate right with extend”, “T2 BIC, shift or rotate by value”, “T2 BICS, rotate right with extend”, and “T2 BICS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 BIC, rotate right with extend”, “A1 BIC, shift or rotate by value”, “A1 BICS, rotate right with extend”, and “A1 BICS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 BIC, rotate right with extend”, “T2 BIC, shift or rotate by value”, “T2 BICS, rotate right with extend”, and “T2 BICS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 BIC, rotate right with extend”, “A1 BIC, shift or rotate by value”, “A1 BICS, rotate right with extend”, and “A1 BICS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 BIC, rotate right with extend”, “T2 BIC, shift or rotate by value”, “T2 BICS, rotate right with extend”, and “T2 BICS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 BIC, shift or rotate by value” and “A1 BICS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2 BIC, shift or rotate by value” and “T2 BICS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.23 BIC, BICS (register-shifted register)

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

31–28	27–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 1	1 0 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
BICS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
BIC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.24 BKPT

Breakpoint causes a Breakpoint Instruction exception.

Breakpoint is always unconditional, even when inside an IT block.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23–22	21	20	19–8	7	6–4	3–0
!=1111 (cond)	0	0	0	1	0	0	imm12	0	1 1 1	imm4

Encoding

```
BKPT{<q>}  {#}<imm>
```

Decode for this encoding

```
let imm16 : bits(16) = imm12::imm4;
if cond != '1110' then UnpredictableProcedure(); end;  // BKPT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15–12	11–8	7–0
1 0 1 1	1 1 1 0	imm8

Encoding

```
BKPT{<q>}  {#}<imm>
```

Decode for this encoding

```
let imm16 : bits(16) = ZeroExtend{}(imm8);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<q>

See Standard assembler syntax fields. A `BKPT` instruction must be unconditional.

<imm>

For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

For the "T1" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

Operation

```
EncodingSpecificOperations();
AArch32_SoftwareBreakpoint(imm16);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.25 BL, BLX (immediate)

Branch with Link and optional Exchange (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–28	27	26	25	24	23–0
!=1111 (cond)	1	0	1	1 (H)	imm24

Encoding

```
BL{<c>}{<q>}  <label>
```

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm24::'00');
let targetInstrSet : InstrSet = InstrSet_A32;
```

A2

31–28	27	26	25	24	23–0
1 1 1 1	1	0	1	H	imm24

Encoding

```
BLX{<c>}{<q>}  <label>
```

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm24::H::'0');
let targetInstrSet : InstrSet = InstrSet_T32;
```

T1

15–13	12	11	10	9–0	15	14	13	12	11	10–0
1 1 1	1	0	S	imm10	1	1	J1	1	J2	imm11

Encoding

```
BL{<c>}{<q>}  <label>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{}(S::I1::I2::imm10::imm11::'0');
let targetInstrSet : InstrSet = InstrSet_T32;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15		13	12	11	10	9 … 0	15	14	13	12	11	10 … 1	0
1	1	1	1	0	S	imm10H	1	1	J1	0	J2	imm10L	H

Encoding

```
BLX{<c>}{<q>}  <label>
```

Decode for this encoding

```
if H == '1' then Undefined(); end;
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{}(S::I1::I2::imm10H::imm10L::'00');
let targetInstrSet : InstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “A1”, “T1”, and “T2” variants: see Standard assembler syntax fields.

For the “A2” variant: see Standard assembler syntax fields. `<c>` must be `AL` or omitted.

<q>

See Standard assembler syntax fields.

<label>

For the “A1” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For the “A2” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `BLX` instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are even numbers in the range –33554432 to 33554430.

For the “T1” variant: the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are even numbers in the range –16777216 to 16777214.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “T2” variant: the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `BLX` instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR() = PC32() - 4;
    else
        LR() = PC32()[31:1] :: '1';
    end;
    var targetAddress : bits(32);
    if targetInstrSet == InstrSet_A32 then
        targetAddress = AlignDownSize(PC32(),4) + imm32;
    else
        targetAddress = PC32() + imm32;
    end;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.26 BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27 26	25–23	22 21	20	19–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	0 1 (op0)	0	(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)	0	0 1 1	Rm

Encoding

```
BLX{<c>}{<q>}  <Rm>
```

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
```

T1

15–10	9 8	7	6–3	2–0
0 1 0 0 0 1	1 1	1 (L)	Rm	(0)(0)(0)

Encoding

```
BLX{<c>}{<q>}  <Rm>
```

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rm>

Is the general-purpose register holding the address to be branched to, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let target : bits(32) = R(m);
    var next_instr_addr : bits(32);
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC32() - 4;
        LR() = next_instr_addr;
    else
        next_instr_addr = PC32() - 2;
        LR() = next_instr_addr[31:1] :: '1';
    end;
    BXWritePC(target, BranchType_INDCALL);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23–22	21	20	19–8	7	6–4	3–0
!=1111 (cond)	0	0	0	1	0	0	(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)	0	0 0 1	Rm

Encoding

```
BX{<c>}{<q>}  <Rm>
```

Decode for this encoding

```
let m : integer = UInt(Rm);
```

T1

15–10	9	8	7	6–3	2–0
0 1 0 0 0 1	1	1	0 (L)	Rm	(0)(0)(0)

Encoding

```
BX{<c>}{<q>}  <Rm>
```

Decode for this encoding

```
let m : integer = UInt(Rm);
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rm>

For the “A1” variant: is the general-purpose register holding the address to be branched to, encoded in the “Rm” field. The PC can be used.

For the “T1” variant: is the general-purpose register holding the address to be branched to, encoded in the “Rm” field. The PC can be used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If `<Rm>` is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the `BXWritePC()` pseudocode function has bits<1:0> = ‘10’.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R(m), BranchType_INDIR);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.28 BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.

`BXJ` behaves as a `BX` instruction, see BX. This means it causes a branch to an address and instruction set specified by a register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–23	22–21	20	19–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	0 1 (op0)	0	(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)	0	0 1 0	Rm

Encoding

```
BXJ{<c>}{<q>}  <Rm>
```

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–6	5	4–0
1 1 1	1 0	0	1 1 1 1	0 0	Rm	1	0 (0) 0	(1)	(1) (1) (1)	(0) (0)	(0)	(0) (0) (0) (0) (0)

Encoding

```
BXJ{<c>}{<q>}  <Rm>
```

Decode for this encoding

```
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rm>

Is the general-purpose register holding the address to be branched to, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R(m), BranchType_INDIR);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.29 CBNZ, CBZ

Compare and Branch on Nonzero or Zero

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

15			12	11	10	9	8	7	3	2	0
1	0	1	1	op	0	i	1	imm5		Rn	

Encoding for the CBNZ variant

Applies when `(op == 1)`

```
CBNZ{<q>}  <Rn>, <label>
```

Encoding for the CBZ variant

Applies when `(op == 0)`

```
CBZ{<q>}  <Rn>, <label>
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(i::imm5::'0');
let nonzero : boolean = (op == '1');
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose register to be tested, encoded in the “Rn” field.

<label>

Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as “i:imm5” times 2.

Operation

```
EncodingSpecificOperations();
if nonzero != IsZero(R(n)) then
    CBWritePC(PC32() + imm32);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.30 CLRBHB

Clear Branch History clears the branch history for the current context to the extent that branch history information created before the `CLRBHB` instruction cannot be used by code before the `CLRBHB` instruction to exploitatively control the execution of any indirect branches in code in the current context that appear in program order after the instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_CLRBHB)

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11–0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)(0)(0)(0)00010110
cond						R			imm4								imm12

Encoding

```
CLRBHB{<c>}{<q>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then ExecuteAsNOP(); end;
```

T1

(FEAT_CLRBHB)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	1	1	0
																								hint						option	

Encoding

```
CLRBHB{<c>}{<q>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then ExecuteAsNOP(); end;
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_CLRBHB();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.31 CLREX

Clear-Exclusive clears the local monitor of the executing PE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25–21	20	19–8	7–4	3–0
1 1 1 1	0	1	0 1 0 1 1	1	(1) (1) (1) (1) (1) (1) (1) (1) (0) (0) (0) (0)	0 0 0 1 (opcode)	(1) (1) (1) (1) (option)

Encoding

```
CLREX{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–13	12 11	10	9–6	5 4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 1	(1) (1) (1) (1)	1	0 (0) 0	(1)	(1) (1) (1)	0 0 1 0 (opc)	(1) (1) (1) (1) (option)

Encoding

```
CLREX{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. Must be `AL` or omitted.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.32 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–23	22–21	20	19–16	15–12	11–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	1 1 (op0)	0	(1) (1) (1) (1)	Rd	(1) (1) (1) (1)	0	0 0 1	Rm

Encoding

```
CLZ{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1	1 1 0 1 0	1	0 1 1 (op1)	Rn	1 1 1 1	Rd	1 0	0 0 (op2)	Rm

Encoding

```
CLZ{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” variant: is the general-purpose source register, encoded in the “Rm” field.

For the “T1” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = CountLeadingZeroBits(R(m));
    R(d) = result[31:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.33 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

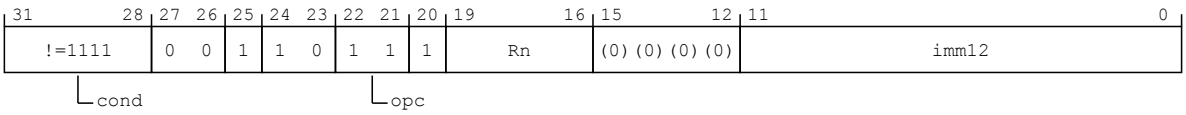

Encoding

```
CMN{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

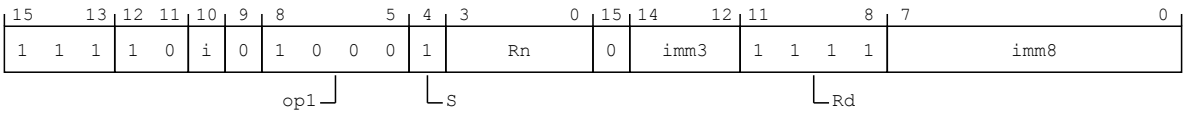

Encoding

```
CMN{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1” variant: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.34 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

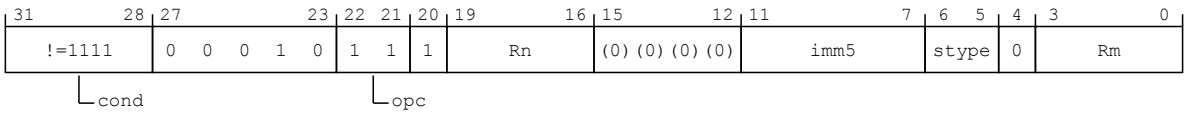

Encoding for the Rotate right with extend variant

Applies when `(imm5 == 00000 && stype == 11)`

```
CMN{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm5 == 00000 && stype == 11))`

```
CMN{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```

T1

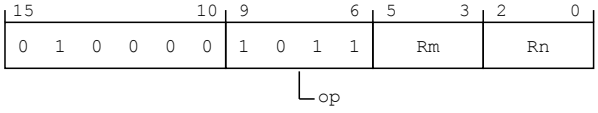

Encoding

```
CMN{<c>}{<q>}  <Rn>, <Rm>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	1 0 0 0 (op1)	1 (S)	Rn	(0)	imm3	1 1 1 1 (Rd)	imm2	stype	Rm

Encoding for the Rotate right with extend variant

Applies when `(imm3 == 000 && imm2 == 00 && stype == 11)`

```
CMN{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00 && stype == 11))`

```
CMN{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```

```
CMN{<c>}.W  <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.35 CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

31–28	27–26	25	24–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0	0	1 0	1 1 (opc)	1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

Encoding

```
CMN{<c>}{<q>}  <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.36 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

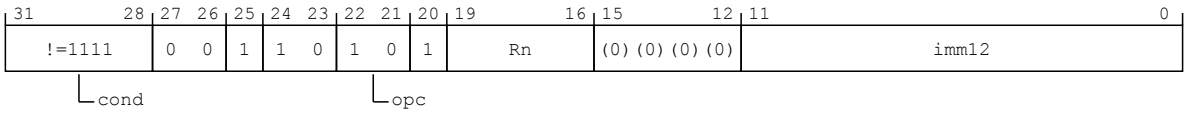

Encoding

```
CMP{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

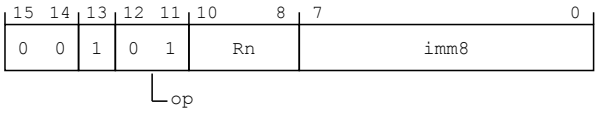

Encoding

```
CMP{<c>}{<q>}  <Rn>, #<imm8>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T2

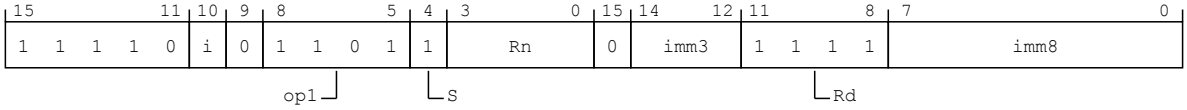

Encoding

```
CMP{<c>}{<q>}  <Rn>, #<const>
```

```
CMP{<c>}.W  <Rn>, #<const> // (<Rd>, <const> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the "A1" variant: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For the "T1" variant: is a general-purpose source register, encoded in the "Rn" field.

For the "T2" variant: is the general-purpose source register, encoded in the "Rn" field.

<const>

For the "A1" variant: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the "T2" variant: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

<imm8>

Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), '1');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.37 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

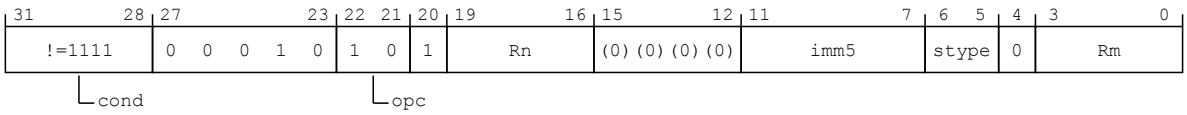

Encoding for the Rotate right with extend variant

Applies when `(imm5 == 00000 && stype == 11)`

```
CMP{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm5 == 00000 && stype == 11))`

```
CMP{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```

T1

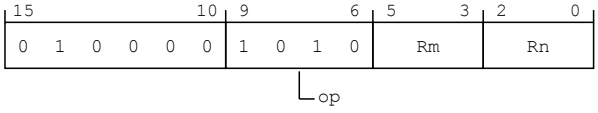

Encoding

```
CMP{<c>}{<q>}  <Rn>, <Rm> // (<Rn> and <Rm> both from R0-R7)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

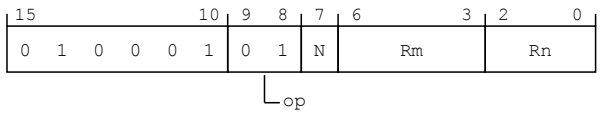

Encoding

```
CMP{<c>}{<q>}  <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)
```

Decode for this encoding

```
let n : integer = UInt(N::Rn);
let m : integer = UInt(Rm);
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if n < 8 && m < 8 then UnpredictableProcedure(); end;
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n < 8 && m < 8`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

T3

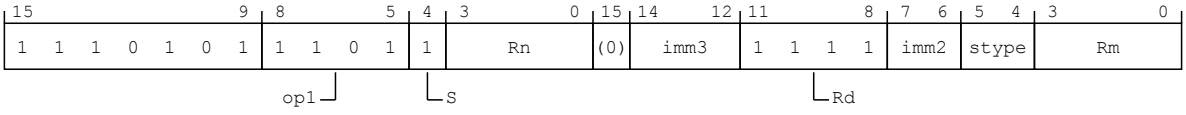

Encoding for the Rotate right with extend variant

Applies when `(imm3 == 000 && imm2 == 00 && stype == 11)`

```
CMP{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00 && stype == 11))`

```
CMP{<c>}{<q>}  <Rn>, <Rm>, <shift> #<amount>
```

```
CMP{<c>}.W  <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1 or T2)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T3 Rotate right with extend”, and “T3 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

For the “T2” variant: is the first general-purpose source register, encoded in the “N:Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2”, “T3 Rotate right with extend”, and “T3 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T3 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.38 CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

31–28	27–26	25	24–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0	0	1 0	1 0 (opc)	1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

Encoding

```
CMP{<c>}{<q>}  <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.39 CPS, CPSID, CPSIE

Change PE State changes one or more of the PSTATE.{A, I, F} interrupt mask bits and, optionally, the PSTATE.M mode field, without changing any other PSTATE bits.

`CPS` is treated as `NOP` if executed in User mode unless it is defined as being CONSTRAINED UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31											20	19 18	17	16	15						10	9	8	7	6	5	4	0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	0	(0)	(0)	(0)	(0)	(0)	(0)		(0)	A	I	F	0	mode	
														op								E						

Encoding for the Change mode variant

Applies when `(imod == 00 && M == 1)`

```
CPS{<q>}  #<mode> // (Cannot be conditional)
```

Encoding for the Interrupt disable variant

Applies when `(imod == 11 && M == 0)`

```
CPSID{<q>}  <iflags> // (Cannot be conditional)
```

Encoding for the Interrupt disable and change mode variant

Applies when `(imod == 11 && M == 1)`

```
CPSID{<q>}  <iflags> , #<mode> // (Cannot be conditional)
```

Encoding for the Interrupt enable variant

Applies when `(imod == 10 && M == 0)`

```
CPSIE{<q>}  <iflags> // (Cannot be conditional)
```

Encoding for the Interrupt enable and change mode variant

Applies when `(imod == 10 && M == 1)`

```
CPSIE{<q>}  <iflags> , #<mode> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if mode != '00000' && M == '0' then UnpredictableProcedure(); end;
if (imod[1] == '1' && A::I::F == '000') || (imod[1] == '0' && A::I::F != '000') then
    UnpredictableProcedure();
end;
let enable : boolean = (imod == '10');
let disable : boolean = (imod == '11');
let changemode : boolean = (M == '1');
let pemode : bits(5) = mode;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `imod == '00' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == ‘0’.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == ‘1’.
- The instruction behaves as if A:I:F == ‘000’.

T1

15									6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	A	I	F
										op					

Encoding for the Interrupt disable variant

Applies when `(im == 1)`

```
CPSID{<q>}  <iflags> // (Not permitted in IT block)
```

Encoding for the Interrupt enable variant

Applies when `(im == 0)`

```
CPSIE{<q>}  <iflags> // (Not permitted in IT block)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if A::I::F == '000' then UnpredictableProcedure(); end;
let enable : boolean = (im == '0');
let disable : boolean = (im == '1');
let changemode : boolean = FALSE;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
let pemode : bits(5) = ARBITRARY : bits(5);
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T2

15 … 11	10 9	8	7	6	5	4 … 0
111100111010(1)(1)(1)(1)10(0)0(0)	imod	M	A	I	F	mode

Encoding for the Change mode variant

Applies when `(imod == 00 && M == 1)`

```
CPS{<q>}  #<mode> // (Not permitted in IT block)
```

Encoding for the Interrupt disable variant

Applies when `(imod == 11 && M == 0)`

```
CPSID.W  <iflags> // (Not permitted in IT block)
```

Encoding for the Interrupt disable and change mode variant

Applies when `(imod == 11 && M == 1)`

```
CPSID{<q>}  <iflags>, #<mode> // (Not permitted in IT block)
```

Encoding for the Interrupt enable variant

Applies when `(imod == 10 && M == 0)`

```
CPSIE.W  <iflags> // (Not permitted in IT block)
```

Encoding for the Interrupt enable and change mode variant

Applies when `(imod == 10 && M == 1)`

```
CPSIE{<q>}  <iflags>, #<mode> // (Not permitted in IT block)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if imod == '00' && M == '0' then See("Hint instructions"); end;
if mode != '00000' && M == '0' then UnpredictableProcedure(); end;
if (imod[1] == '1' && A::I::F == '000') || (imod[1] == '0' && A::I::F != '000') then
    UnpredictableProcedure();
end;
let enable : boolean = (imod == '10');
let disable : boolean = (imod == '11');
let changemode : boolean = (M == '1');
let pemode : bits(5) = mode;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
if imod == '01' || InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == ‘0’.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == ‘1’.
- The instruction behaves as if A:I:F == ‘000’.

Hint instructions: In encoding T2, if the `imod` field is `0b00` and the `M` bit is `0b0`, a hint instruction is encoded. To determine which hint instruction, see Branches and miscellaneous control.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<mode>

Is the number of the mode to change to, in the range 0 to 31, encoded in the “mode” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<iflags>

Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

a Sets the A bit in the instruction, causing the specified effect on PSTATE.A, the SError interrupt mask bit.
i Sets the I bit in the instruction, causing the specified effect on PSTATE.I, the IRQ interrupt mask bit.
f Sets the F bit in the instruction, causing the specified effect on PSTATE.F, the FIQ interrupt mask bit.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0'; end;
            if affectI then PSTATE.I = '0'; end;
            if affectF then PSTATE.F = '0'; end;
        end;
        if disable then
            if affectA then PSTATE.A = '1'; end;
            if affectI then PSTATE.I = '1'; end;
            if affectF then PSTATE.F = '1'; end;
        end;
        if changemode then
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32_WriteModeByInstr(pemode);
        end;
    end;
else
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0'; end;
            if affectI then PSTATE.I = '0'; end;
            if affectF then PSTATE.F = '0'; end;
        end;
        if disable then
            if affectA then PSTATE.A = '1'; end;
            if affectI then PSTATE.I = '1'; end;
            if affectF then PSTATE.F = '1'; end;
        end;
        if changemode then
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32_WriteModeByInstr(pemode);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.40 CRC32

CRC32

`CRC32` performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x04C11DB7` is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_CRC32)

31–28	27–23	22–21	20	19–16	15–12	11–10	9	8–4	3–0
!=1111	0 0 0 1 0	sz	0	Rn	Rd	(0)(0)	0	(0)0100	Rm
cond							C		

Encoding for the CRC32B variant

Applies when `(sz == 00)`

```
CRC32B{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32H variant

Applies when `(sz == 01)`

```
CRC32H{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32W variant

Applies when `(sz == 10)`

```
CRC32W{<q>}  <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
if cond != '1110' then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

(FEAT_CRC32)

15 … 5	4	3 … 0	15 … 12	11 … 8	7 6	5 4	3 … 0
1 1 1 1 1 0 1 0 1 1 0	0	Rn	1 1 1 1	Rd	1 0	sz	Rm
	C						

Encoding for the CRC32B variant

Applies when `(sz == 00)`

```
CRC32B{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32H variant

Applies when `(sz == 01)`

```
CRC32H{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32W variant

Applies when `(sz == 10)`

```
CRC32W{<q>}  <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields. A `CRC32` instruction must be unconditional.

<Rd>

Is the general-purpose accumulator output register, encoded in the “Rd” field.

<Rn>

Is the general-purpose accumulator input register, encoded in the “Rn” field.

<Rm>

Is the general-purpose data source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    let acc : bits(32) = R(n);             // accumulator
    let val : bits(size) = R(m)[size-1:0]; // input value
    let poly : bits(32) = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)[31:0];
    let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
    let tempval : bits(32+size) = BitReverse{size}(val)::Zeros{32};
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.41 CRC32C

CRC32C

`CRC32C` performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_CRC32)

31–28	27–23	22–21	20	19–16	15–12	11–10	9	8–4	3–0
!=1111	0 0 0 1 0	sz	0	Rn	Rd	(0)(0)	1	(0)0100	Rm
cond							C		

Encoding for the CRC32CB variant

Applies when `(sz == 00)`

```
CRC32CB{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32CH variant

Applies when `(sz == 01)`

```
CRC32CH{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32CW variant

Applies when `(sz == 10)`

```
CRC32CW{<q>}  <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
if cond != '1110' then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

(FEAT_CRC32)

15–5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 1 1 0 1 0 1 1 0	1	Rn	1 1 1 1	Rd	1 0	sz	Rm
	C						

Encoding for the CRC32CB variant

Applies when `(sz == 00)`

```
CRC32CB{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32CH variant

Applies when `(sz == 01)`

```
CRC32CH{<q>}  <Rd>, <Rn>, <Rm>
```

Encoding for the CRC32CW variant

Applies when `(sz == 10)`

```
CRC32CW{<q>}  <Rd>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: size = 32;.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields. A `CRC32C` instruction must be unconditional.

<Rd>

Is the general-purpose accumulator output register, encoded in the “Rd” field.

<Rn>

Is the general-purpose accumulator input register, encoded in the “Rn” field.

<Rm>

Is the general-purpose data source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    let acc : bits(32) = R(n);             // accumulator
    let val : bits(size) = R(m)[size-1:0]; // input value
    let poly : bits(32) = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)[31:0];
    let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
    let tempval : bits(32+size) = BitReverse{size}(val)::Zeros{32};
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.42 CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution arising from data value prediction. For more information and details of the semantics, see Consumption of Speculative Data Barrier (CSDB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

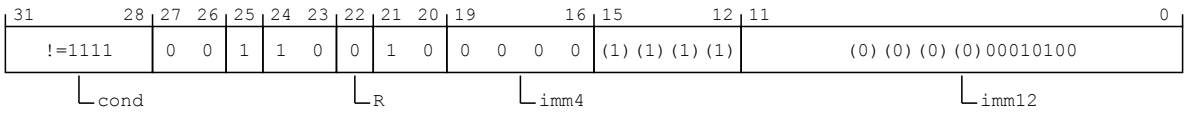

Encoding

```
CSDB{<c>}{<q>}
```

Decode for this encoding

```
if cond != '1110' then UnpredictableProcedure(); end; // CSDB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

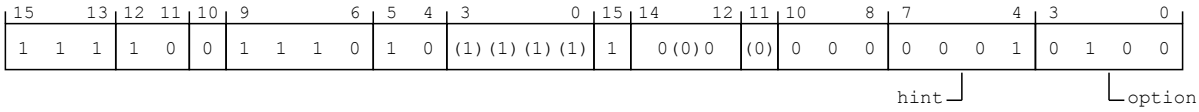

Encoding

```
CSDB{<c>}{<q>}
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    ConsumptionOfSpeculativeDataBarrier();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.43 DBG

Debug hint

`DBG` executes as a `NOP`. Arm deprecates any use of the `DBG` instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–4	3–0
!=1111 (cond)	0	0	1	1	0	0 (R)	1	0	0 0 0 0 (imm4)	(1)(1)(1)(1)	(0)(0)(0)(0)1111	option

Encoding

```
DBG{<c>}{<q>}  #<option>
```

Decode for this encoding

```
// DBG executes as a NOP. The 'option' field is ignored
```

T1

15–13	12	11	10	9–6	5	4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1	0	0	1 1 1 0	1	0	(1)(1)(1)(1)	1	0(0)0	(0)	0 0 0	1 1 1 1 (hint)	option

Encoding

```
DBG{<c>}{<q>}  #<option>
```

Decode for this encoding

```
// DBG executes as a NOP. The 'option' field is ignored
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<option>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “option” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.44 DCPS1

Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from EL0 or to a specific mode at the current Exception level.

`DCPS1` is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is implemented, EL2 is implemented and enabled in the current Security state, and any of:
  - EL2 is using AArch32 and HCR.TGE is set to 1.
  - EL2 is using AArch64 and HCR_EL2.TGE is set to 1.

When the PE executes `DCPS1` at EL0, EL1 or EL3:

- If EL3 or EL1 is using AArch32, the PE enters SVC mode and LR_svc, SPSR_svc, DLR, and DSPSR become UNKNOWN. If `DCPS1` is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL1 is using AArch64, the PE enters EL1 using AArch64, selects SP_EL1, and ELR_EL1, ESR_EL1, SPSR_EL1, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

When the PE executes `DCPS1` at EL2 the PE does not change mode, and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see DCPS.

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11–2	1–0
1 1 1	1 0	1	1 1 1 0	0 0	1 1 1 1	1	0 0 0	0 0 0 0 0 0 0 0 0 0	0 1
					imm4			imm10	opt

Encoding

```
DCPS1
```

Decode for this encoding

```
// No additional decoding required.
```

Operation

```
if !Halted() then Undefined(); end;

if EL2Enabled() && PSTATE.EL == EL0 then
    let tge : bit = if ELUsingAArch32(EL2) then HCR().TGE else HCR_EL2().TGE;
    if tge == '1' then Undefined(); end;
end;

if PSTATE.EL != EL0 || ELUsingAArch32(EL1) then
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    if PSTATE.EL != EL2 then
        AArch32_WriteMode(M32_Svc);
        PSTATE.E = SCTLR().EE;
        if IsFeatureImplemented(FEAT_PAN) && SCTLR().SPAN == '0' then PSTATE.PAN = '1'; end;
        Rmode(14, M32_Svc) = ARBITRARY : bits(32);    // LR_svc
        SPSR_svc() = ARBITRARY : bits(32);
    else
        PSTATE.E = HSCTLR().EE;
        ELR_hyp() = ARBITRARY : bits(32);
        HSR() = ARBITRARY : bits(32);
        SPSR_hyp() = ARBITRARY : bits(32);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
else                                        // Targeting EL1 using AArch64
    AArch64_MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL1);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL1;
    if IsFeatureImplemented(FEAT_PAN) && SCTLR_EL1().SPAN == '0' then PSTATE.PAN = '1'; end;
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;

    ELR_EL1() = ARBITRARY : bits(64);
    ESR_EL1() = ARBITRARY : bits(64);
    SPSR_EL1() = ARBITRARY : bits(64);

    DLR_EL0() = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    // SCTLR_EL1.IESB might be ignored in Debug state.
    if (IsFeatureImplemented(FEAT_IESB) && SCTLR_EL1().IESB == '1' &&
          !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
        SynchronizeErrors();
    end;
end;

UpdateEDSCRFields();                        // Update EDSCR PE state flags
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.45 DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

`DCPS2` is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
  - Secure EL2 is not implemented.
  - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes `DCPS2`:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see DCPS.

15 13	12 11	10	9 6	5 4	3 0	15	14 12	11 2	1 0
1 1 1	1 0	1	1 1 1 0	0 0	1 1 1 1	1	0 0 0	0 0 0 0 0 0 0 0 0 0	1 0
					imm4			imm10	opt

Encoding

```
DCPS2
```

Decode for this encoding

```
if !HaveEL(EL2) then Undefined(); end;
```

Operation

```
if !Halted() || !EL2Enabled() then Undefined(); end;

if ELUsingAArch32(EL2) then
    AArch32_WriteMode(M32_Hyp);
    PSTATE.E = HSCTLR().EE;

    ELR_hyp() = ARBITRARY : bits(32);
    HSR() = ARBITRARY : bits(32);
    SPSR_hyp() = ARBITRARY : bits(32);

    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
else                                        // Targeting EL2 using AArch64
    AArch64_MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL2);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL2;
    if IsFeatureImplemented(FEAT_PAN) && SCTLR_EL2().SPAN == '0' && ELIsInHost(EL0) then
        PSTATE.PAN = '1';
    end;
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;

    ELR_EL2() = ARBITRARY : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    ESR_EL2() = ARBITRARY : bits(64);
    SPSR_EL2() = ARBITRARY : bits(64);

    DLR_EL0() = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    // SCTLR_EL2.IESB might be ignored in Debug state.
    if (IsFeatureImplemented(FEAT_IESB) && SCTLR_EL2().IESB == '1' &&
          !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
        SynchronizeErrors();
    end;
end;

UpdateEDSCRFields();                        // Update EDSCR PE state flags
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.46 DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception level or to a specific mode at the current Exception level.

`DCPS3` is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes `DCPS3`:

- If EL3 is using AArch32, the PE enters Monitor mode and LR_mon, SPSR_mon, DLR and DSPSR become UNKNOWN. If `DCPS3` is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP_EL3, and ELR_EL3, ESR_EL3, SPSR_EL3, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see DCPS.

15 13	12 11	10	9 6	5 4	3 0	15	14 12	11 2	1 0
1 1 1	1 0	1	1 1 1 0	0 0	1 1 1 1	1	0 0 0	0 0 0 0 0 0 0 0 0 0	1 1
					imm4			imm10	opt

Encoding

```
DCPS3
```

Decode for this encoding

```
if !HaveEL(EL3) then Undefined(); end;
```

Operation

```
if !Halted() || EDSCR().SDD == '1' then Undefined(); end;

if ELUsingAArch32(EL3) then
    let from_secure : boolean = (CurrentSecurityState() == SS_Secure);
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    AArch32_WriteMode(M32_Monitor);
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR().SPAN == '0' then
            PSTATE.PAN = '1';
        end;
    end;
    PSTATE.E = SCTLR().EE;

    LR_mon = ARBITRARY : bits(32);
    SPSR_mon() = ARBITRARY : bits(32);

    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
else                                        // Targeting EL3 using AArch64
    AArch64_MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL3);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL3;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;

    ELR_EL3() = ARBITRARY : bits(64);
    ESR_EL3() = ARBITRARY : bits(64);
    SPSR_EL3() = ARBITRARY : bits(64);

    DLR_EL0() = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    var sync_errors : boolean = IsFeatureImplemented(FEAT_IESB) && SCTLR_EL3().IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) && EffectiveEA() == '1' && SCR_EL3().NMEA == '1' then
        sync_errors = TRUE;
    end;
    // SCTLR_EL3.IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    end;
    if sync_errors then SynchronizeErrors(); end;
end;

UpdateEDSCRFields();                        // Update EDSCR PE state flags
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.47 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier (DMB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25–21	20	19–8	7–4	3–0
1 1 1 1	0	1	0 1 0 1 1	1	(1) (1) (1) (1) (1) (1) (1) (1) (0) (0) (0) (0)	0 1 0 1 (opcode)	option

Encoding

```
DMB{<c>}{<q>}  {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 1	(1) (1) (1) (1)	1	0 (0) 0	(1)	(1) (1) (1)	0 1 0 1 (opc)	option

Encoding

```
DMB{<c>}{<q>}  {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. Must be `AL` or omitted.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<option>

Specifies an optional limitation on the barrier operation. Values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = `0b1111`.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. `SYST` is a synonym for `ST`. Encoded as option = `0b1110`.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1101`.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b1011`.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b1010`.

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1001`.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = `0b0111`.

NSHST Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = `0b0110`.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0101`.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b0011`.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b0010`.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0001`.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB). All other encodings of option are reserved. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

Note

The instruction supports the following alternative `<option>` values, but Arm recommends that software does not use these alternative values:

- `SH` as an alias for `ISH`.
- `SHST` as an alias for `ISHST`.
- `UN` as an alias for `NSH`.
- `UNST` as an alias for `NSHST`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var types : MBReqTypes;
    case option[1:0] of
        when '00' => types = MBReqTypes_All;
        when '01' => types = MBReqTypes_Reads;
        when '10' => types = MBReqTypes_Writes;
        when '11' => types = MBReqTypes_All;
    end;

    DataMemoryBarrier(types);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.48 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier (DSB).

An AArch32 DSB instruction does not require the completion of any AArch64 TLB maintenance instructions, regardless of the nXS qualifier, appearing in program order before the AArch32 DSB.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

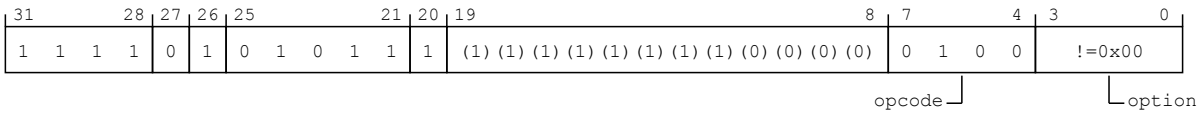

Encoding

```
DSB{<c>}{<q>}  {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

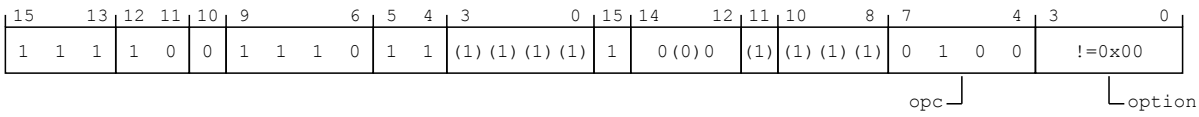

Encoding

```
DSB{<c>}{<q>}  {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. Must be `AL` or omitted.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = `0b1111`.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. `SYST` is a synonym for `ST`. Encoded as option = `0b1110`.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1101`.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b1011`.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b1010`.

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b1001`.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = `0b0111`.

NSHST Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = `0b0110`.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0101`.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = `0b0011`.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = `0b0010`.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = `0b0001`.

For more information on whether an access is before or after a barrier instruction, see Data Synchronization Barrier (DSB). All other encodings of option are reserved, other than the values `0b0000` and `0b0100`. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

Note

The value `0b0000` is used to encode SSBB and the value `0b0100` is used to encode PSSBB.
The instruction supports the following alternative `<option>` values, but Arm recommends that software does not use these alternative values:

- `SH` as an alias for `ISH`.
- `SHST` as an alias for `ISHST`.
- `UN` as an alias for `NSH`.
- `UNST` as an alias for `NSHST`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var nXS : boolean;
    if IsFeatureImplemented(FEAT_XS) then
        nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) &&
               IsHCRXEL2Enabled() && HCRX_EL2().FnXS == '1');
    else
        nXS = FALSE;
    end;
    var scope : MBMaintenanceScope;
    var types : MBReqTypes;
    case option[1:0] of
        when '00' =>
            types = MBReqTypes_All;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            scope = MBMaintenanceScope_OuterShareable;
        when '01' =>
            types = MBReqTypes_Reads;
            scope = MBMaintenanceScope_None;
        when '10' =>
            types = MBReqTypes_Writes;
            scope = MBMaintenanceScope_None;
        when '11' =>
            types = MBReqTypes_All;
            case option[3:2] of
                when '00' => scope = MBMaintenanceScope_OuterShareable;
                when '01' => scope = MBMaintenanceScope_Nonshareable;
                when '10' => scope = MBMaintenanceScope_InnerShareable;
                when '11' => scope = MBMaintenanceScope_OuterShareable;
            end;
    end;

    DataSynchronizationBarrier(scope, types, nXS);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.49 EOR, EORS (immediate)

Bitwise Exclusive-OR (immediate) performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

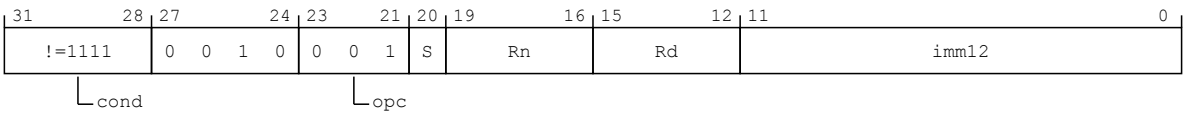

Encoding for the EOR variant

Applies when `(S == 0)`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the EORS variant

Applies when `(S == 1)`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

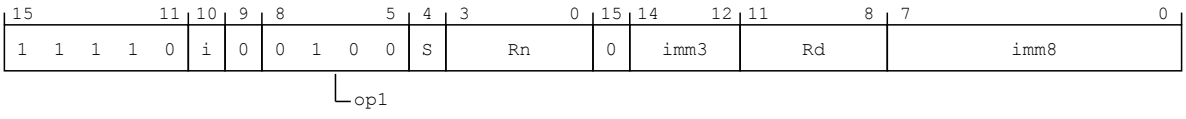

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the EOR variant

Applies when `(S == 0)`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the EORS variant

Applies when `(S == 1 && Rd != 1111)`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TEQ (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 EOR” and “A1 EORS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 EOR” and “T1 EORS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 EOR” and “A1 EORS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 EOR” and “T1 EORS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 EOR” and “A1 EORS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1 EOR” and “T1 EORS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) XOR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.50 EOR, EORS (register)

Bitwise Exclusive-OR (register) performs a bitwise exclusive-OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	0 0 1	S	Rn	Rd	imm5	stype	0	Rm
cond		opc							

Encoding for the EOR, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the EOR, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the EORS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the EORS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	0 0 0 1	Rm	Rdn

op

Encoding

```
EOR<c>{<q>}  {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
EORS{<q>}  {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

15–9	8–5	4	3–0	15	14–12	11–8	7 6	5 4	3–0
1 1 1 0 1 0 1	0 1 0 0	S	Rn	(0)	imm3	Rd	imm2	stype	Rm

op1

Encoding for the EOR, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the EOR, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
EOR<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the EORS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the EORS, shift or rotate by value variant

Applies when `(S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
EORS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TEQ (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 EOR, rotate right with extend”, “A1 EOR, shift or rotate by value”, “A1 EORS, rotate right with extend”, and “A1 EORS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2 EOR, rotate right with extend”, “T2 EOR, shift or rotate by value”, “T2 EORS, rotate right with extend”, and “T2 EORS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 EOR, rotate right with extend”, “A1 EOR, shift or rotate by value”, “A1 EORS, rotate right with extend”, and “A1 EORS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 EOR, rotate right with extend”, “T2 EOR, shift or rotate by value”, “T2 EORS, rotate right with extend”, and “T2 EORS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 EOR, rotate right with extend”, “A1 EOR, shift or rotate by value”, “A1 EORS, rotate right with extend”, and “A1 EORS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 EOR, rotate right with extend”, “T2 EOR, shift or rotate by value”, “T2 EORS, rotate right with extend”, and “T2 EORS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 EOR, shift or rotate by value” and “A1 EORS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 EOR, shift or rotate by value” and “T2 EORS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if `EORS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `EORS <Rd>, <Rn>` had been written
- Inside an IT block, if `EOR<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `EOR<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.51 EOR, EORS (register-shifted register)

Bitwise Exclusive-OR (register-shifted register) performs a bitwise exclusive-OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0 0	0 0 1	S	Rn	Rd	Rs	0	stype	1	Rm
cond		opc								

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
EORS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
EOR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.52 ERET

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores PSTATE from SPSR_<current_mode>.

The register holding the preferred return address is:

- ELR_hyp, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of ERET executes the DRPS operation.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23–22	21	20	19–8	7	6–4	3–0
!=1111 (cond)	0	0	0	1	1	0	(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)	0	1 1 0	(1)(1)(1)(0)

op0

Encoding

```
ERET{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–13	12	11	10	9–6	5	4	3–0	15	14–12	11	10–8	7–0
1 1 1	1	0	0	1 1 1 1	0	1	1 1 1 0 (Rn)	1	0(0)0	(1)	(1)(1)(1)	0 0 0 0 0 0 0 0 (imm8)

Encoding

```
ERET{<c>}{<q>}
```

Decode for this encoding

```
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Halted() then
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();                       // UNDEFINED or NOP
        else
            let new_pc_value : bits(32) = if PSTATE.EL == EL2 then ELR_hyp() else R(14);
            AArch32_ExceptionReturn(new_pc_value, SPSR_curr());
        end;
    else                                            // Perform DRPS operation in Debug state
        if PSTATE.M == M32_User then
            Undefined();
        elsif PSTATE.M == M32_System then
            UnpredictableProcedure();                       // UNDEFINED or NOP
        else
            SynchronizeContext();
            DebugRestorePSR();
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.53 ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR and VDISR. This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. For more information, see RAS PE architecture and Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100).

If FEAT_RAS is not implemented, this instruction executes as a `NOP`.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_RAS)

31–28	27–26	25	24–23	22	21–20	19–16	15–12	11–0
!=1111 (cond)	0 0	1	1 0	0 (R)	1 0	0 0 0 0 (imm4)	(1)(1)(1)(1)	(0)(0)(0)(0)00010000 (imm12)

Encoding

```
ESB{<c>}{<q>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then ExecuteAsNOP(); end;
if cond != '1110' then UnpredictableProcedure(); end; // ESB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

(FEAT_RAS)

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 0	(1)(1)(1)(1)	1	0(0)0	(0)	0 0 0	0 0 0 1 (hint)	0 0 0 0 (option)

Encoding

```
ESB{<c>}{<q>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then ExecuteAsNOP(); end;
if InITBlock() then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    SynchronizeErrors();
    AArch32_ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        AArch32_vESBOperation();
    elsif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && !ELUsingAArch32(EL3) then
        AArch64_dESBOperation();
    end;
    TakeUnmaskedSErrorInterrupts();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.54 HLT

Halting Breakpoint

Halting breakpoint causes a software breakpoint to occur.

Halting breakpoint is always unconditional, even inside an IT block.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–23	22–21	20	19–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	0 0 (opc)	0	imm12	0	1 1 1	imm4

Encoding

```
HLT{<q>}  {#}<imm>
```

Decode for this encoding

```
if EDSCR().HDE == '0' || !HaltingAllowed() then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end; // HLT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15–12	11–8	7–6	5–0
1 0 1 1	1 0 1 0	1 0	imm6

Encoding

```
HLT{<q>}  {#}<imm>
```

Decode for this encoding

```
if EDSCR().HDE == '0' || !HaltingAllowed() then Undefined(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<q>

See Standard assembler syntax fields. An `HLT` instruction must be unconditional.

<imm>

For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

For the "T1" variant: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Operation

```
EncodingSpecificOperations();
let is_async : boolean = FALSE;
let fault : FaultRecord = NoFault();
Halt(DebugHalt_HaltInstruction, is_async, fault);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.55 HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see Hypervisor Call (HVC) exception. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The `HVC` instruction is UNDEFINED:

- When EL3 is implemented and using AArch64, and SCR_EL3.HCE is set to 0.
- In Non-secure EL1 modes when EL3 is implemented and using AArch32, and SCR.HCE is set to 0.
- When EL3 is not implemented and either HCR_EL2.HCD is set to 1 or HCR.HCD is set to 1.
- When EL2 is not implemented.
- In Secure state, if EL2 is not enabled in the current Security state.
- In User mode.

The `HVC` instruction is CONSTRAINED UNPREDICTABLE in Hyp mode when EL3 is implemented and using AArch32, and SCR.HCE is set to 0.

On executing an `HVC` instruction, the HSR, Hyp Syndrome Register reports the exception as a Hypervisor Call exception, using the EC value `0x12`, and captures the value of the immediate argument, see Use of the HSR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

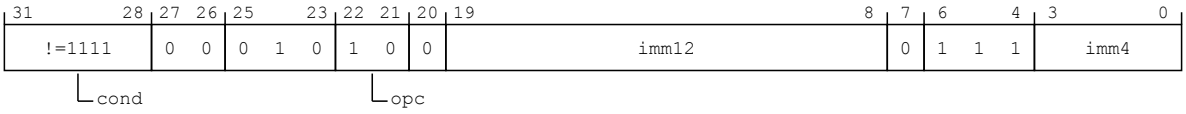

Encoding

```
HVC{<q>}  {#}<imm16>
```

Decode for this encoding

```
if cond != '1110' then UnpredictableProcedure(); end;
let imm16 : bits(16) = imm12::imm4;
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

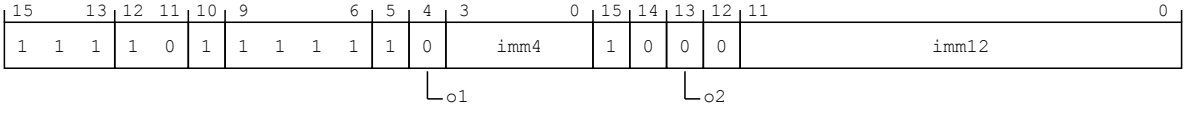

Encoding

```
HVC{<q>}  {#}<imm16>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let imm16 : bits(16) = imm4::imm12;
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields. An HVC instruction must be unconditional.

<imm16>

For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For the "T1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

Operation

```
EncodingSpecificOperations();
if PSTATE.EL IN {EL0, EL3} || !EL2Enabled() then
    Undefined();
end;

var hvc_enable : bit;
if HaveEL(EL3) then
    if ELUsingAArch32(EL3) then
        if SCR().HCE == '0' && PSTATE.EL == EL2 then UnpredictableProcedure(); end;
        hvc_enable = SCR().HCE;
    else
        hvc_enable = SCR_EL3().HCE;
    end;
else
    hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR().HCD) else NOT(HCR_EL2().HCD);
end;

if hvc_enable == '0' then
    Undefined();
else
    AArch32_CallHypervisor(imm16);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.56 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25–21	20	19–8	7–4	3–0
1 1 1 1	0	1	0 1 0 1 1	1	(1) (1) (1) (1) (1) (1) (1) (1) (0) (0) (0) (0)	0 1 1 0 (opcode)	option

Encoding

```
ISB{<c>}{<q>}  {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–13	12 11	10	9–6	5 4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 1	(1) (1) (1) (1)	1	0 (0) 0	(1)	(1) (1) (1)	0 1 1 0 (opc)	option

Encoding

```
ISB{<c>}{<q>}  {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. Must be `AL` or omitted.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY Full system barrier operation, encoded as option = `0b1111`. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.57 IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the `IT` instruction specifies for the first instruction in the block.

The `IT` instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than `CMP`, `CMN` and `TST`, do not set the condition flags. An `IT` instruction with the `AL` condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the CPSR restores PSTATE.IT to a state consistent with the conditions specified by the `IT` instruction. Any other exception return to an instruction in an IT block is UNPREDICTABLE. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is UNPREDICTABLE what condition is used when executing that target instruction and any subsequent instruction in the IT block.

Many uses of the IT instruction are deprecated for performance reasons, and an implementation might include ITD controls that can disable those uses of IT, making them UNDEFINED.

For more information see Conditional execution and Conditional instructions. The first of these sections includes more information about the ITD controls.

15–12	11–8	7–4	3–0
1 0 1 1	1 1 1 1	firstcond	!=0000

mask

Encoding

```
IT{<x>{<y>{<z>}}}{<q>}  <cond>
```

Decode for this encoding

```
if mask == '0000' then See("Related encodings"); end;
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then
    UnpredictableProcedure();
end;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other cond_base value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Miscellaneous 16-bit instructions.

Assembler Symbols

<x>

The condition for the second instruction in the IT block. If omitted, the "mask" field is set to `0b1000`. If present it is encoded in the "mask[3]" field:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T firstcond[0]
E NOT firstcond[0]

<y>

The condition for the third instruction in the IT block. If omitted and `<x>` is present, the “mask[2:0]” field is set to `0b100`. If `<y>` is present it is encoded in the “mask[2]” field:

T firstcond[0]
E NOT firstcond[0]

<z>

The condition for the fourth instruction in the IT block. If omitted and `<y>` is present, the “mask[1:0]” field is set to `0b10`. If `<z>` is present, the “mask[0]” field is set to 1, and it is encoded in the “mask[1]” field:

T firstcond[0]
E NOT firstcond[0]

<q>

See Standard assembler syntax fields.

<cond>

The condition for the first instruction in the IT block, encoded in the “firstcond” field. See Condition codes for the range of conditions available, and the encodings.

The conditions specified in an `IT` instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check `IT` instruction syntax for validity but do not generate assembled instructions for them. See Conditional instructions.

Operation

```
EncodingSpecificOperations();
AArch32_CheckITEnabled(mask);
PSTATE.IT[7:0] = firstcond::mask;
ShouldAdvanceIT = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.58 LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

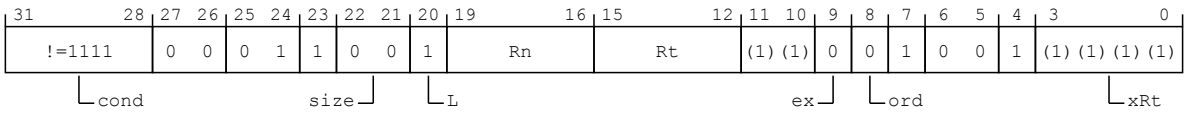

Encoding

```
LDA{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

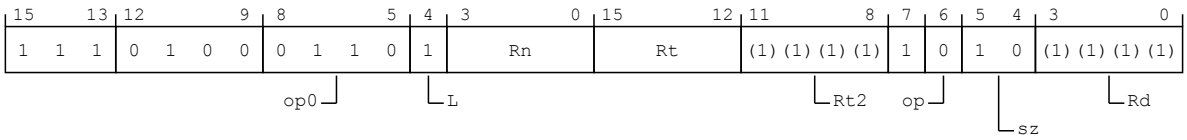

Encoding

```
LDA{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = MemO{32}(address);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.59 LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

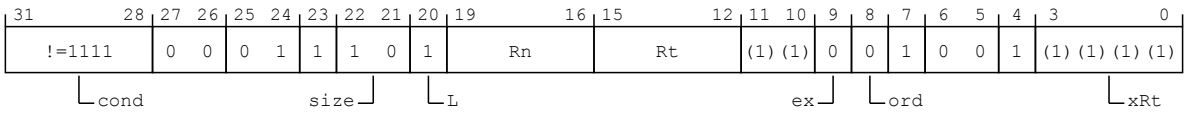

Encoding

```
LDAB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

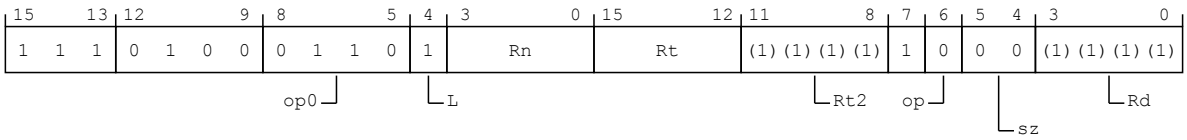

Encoding

```
LDAB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = ZeroExtend{32}(MemO{8}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.60 LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

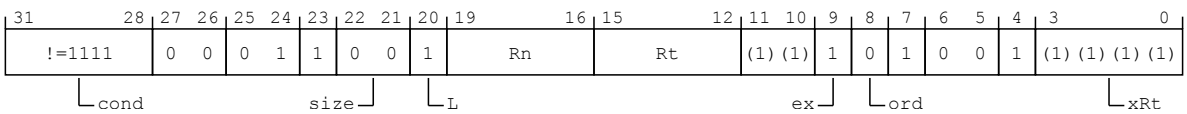

Encoding

```
LDAEX{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

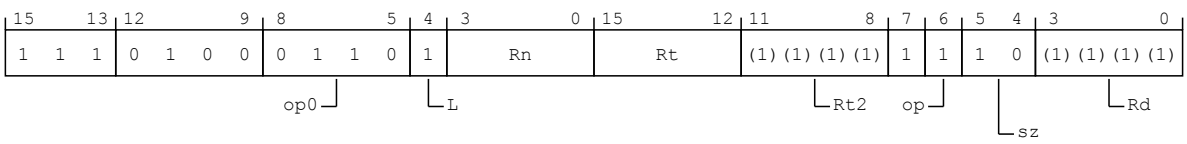

Encoding

```
LDAEX{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 4);
    R(t) = MemO{32}(address);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.61 LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

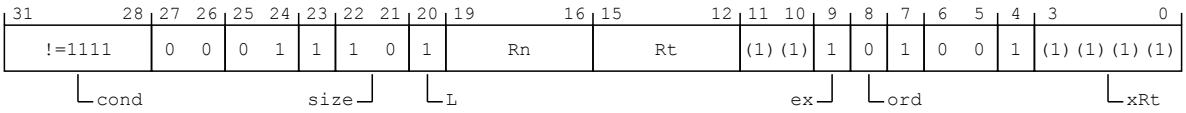

Encoding

```
LDAEXB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

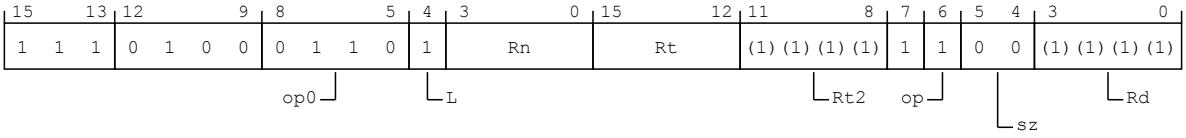

Encoding

```
LDAEXB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 1);
    R(t) = ZeroExtend{32}(MemO{8}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.62 LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

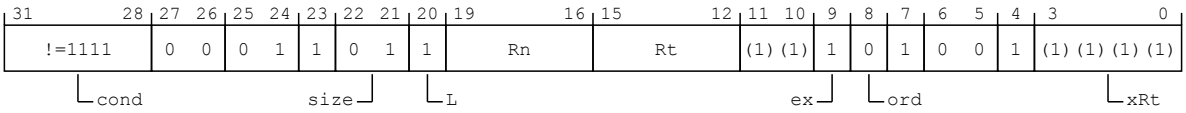

Encoding

```
LDAEXD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
if Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction is handled as described in Using R15.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

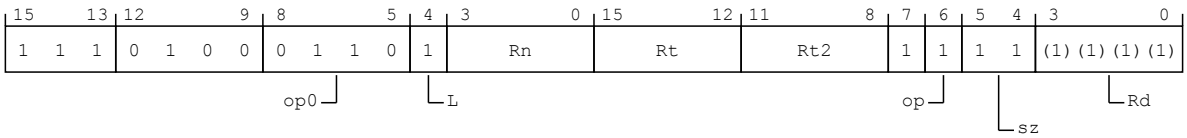

Encoding

```
LDAEXD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. `<Rt>` must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 8);
    let value : bits(64) = MemO{64}(address);
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R(t)  = if BigEndian(AccessType_GPR) then value[63:32] else value[31:0];
    R(t2) = if BigEndian(AccessType_GPR) then value[31:0]  else value[63:32];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.63 LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

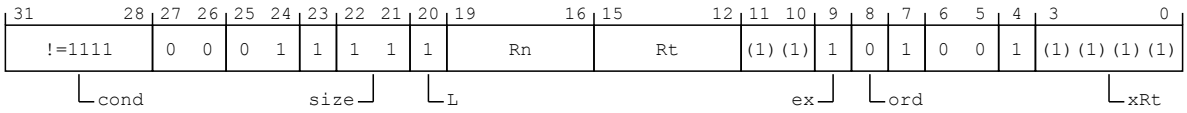

Encoding

```
LDAEXH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

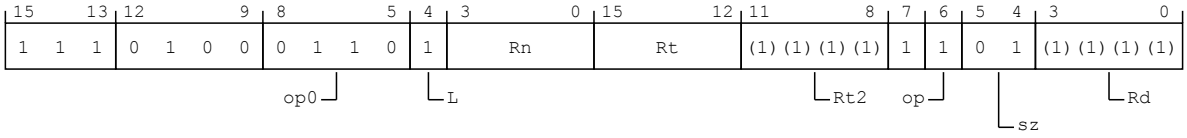

Encoding

```
LDAEXH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 2);
    R(t) = ZeroExtend{32}(MemO{16}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.64 LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

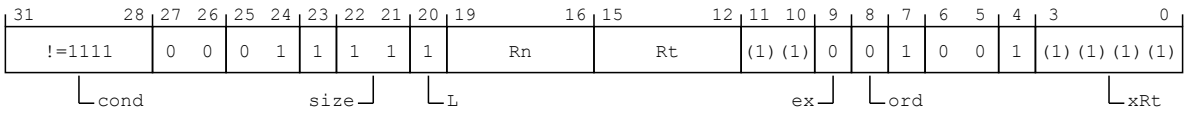

Encoding

```
LDAH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

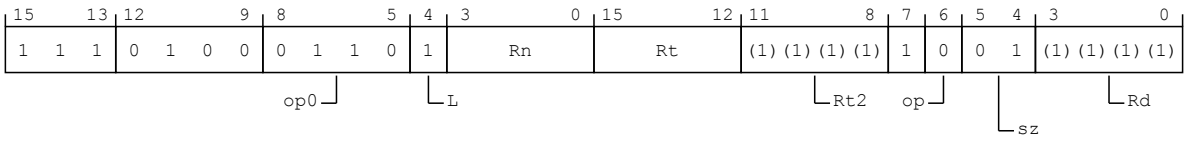

Encoding

```
LDAH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = ZeroExtend{32}(MemO{16}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.65 LDC (immediate)

Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the DBGDTRTXint System register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see Memory accesses.

In an implementation that includes EL2, the permitted `LDC` access to DBGDTRTXint can be trapped to Hyp mode, meaning that an attempt to execute an `LDC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see HDCR.TDA.

For simplicity, the `LDC` pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

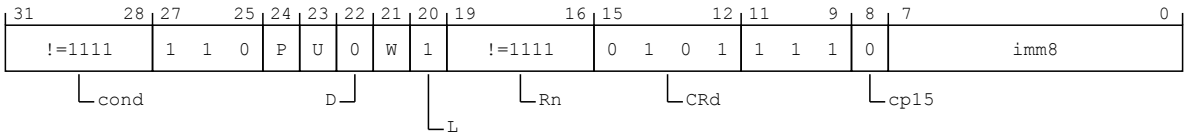

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>, #{+/-}<imm>]!
```

Encoding for the Unindexed variant

Applies when `(P == 0 && U == 1 && W == 0)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDC (literal)"); end;
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

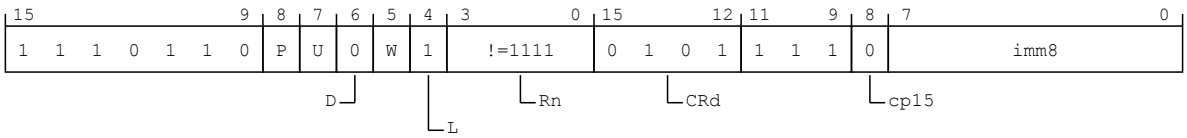

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>, #{+/-}<imm>]!
```

Encoding for the Unindexed variant

Applies when `(P == 0 && U == 1 && W == 0)`

```
LDC{<c>}{<q>}  p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDC (literal)"); end;
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If the PC is used, see LDC (literal).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as <imm>/4.

<option>

Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);

    // System register write to DBGDTRTXint.
    AArch32_SysRegWriteM(cp, ThisInstr(), address);

    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.66 LDC (literal)

Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the DBGDTRTXint System register. For information about memory accesses, see Memory accesses.

In an implementation that includes EL2, the permitted `LDC` access to DBGDTRTXint can be trapped to Hyp mode, meaning that an attempt to execute an `LDC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see HDCR.TDA.

For simplicity, the `LDC` pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

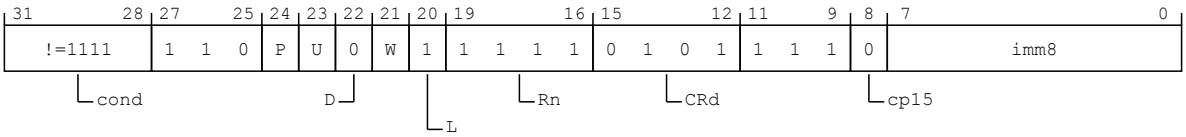

Encoding

Applies when `(!(P == '0' && U == '0' && W == '0'))`

```
LDC{<c>}{<q>}  p14, c5, <label> // (Preferred syntax)
```

```
LDC{<c>}{<q>}  p14, c5, [PC, #{+/-}<imm>]
```

```
LDC{<c>}{<q>}  p14, c5, [PC], <option>
```

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

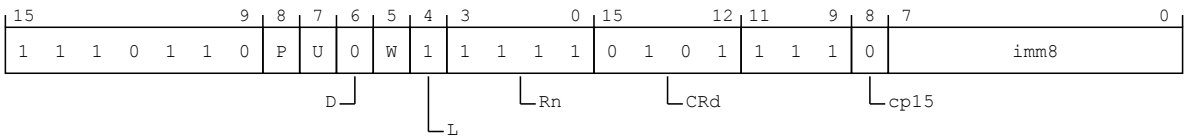

Encoding

Applies when `(!(P == '0' && U == '0' && W == '0'))`

```
LDC{<c>}{<q>}  p14, c5, <label> // (Preferred syntax)
```

```
LDC{<c>}{<q>}  p14, c5, [PC, #{+/-}<imm>]
```

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes as `LDC` (immediate) with writeback to the PC. The instruction is handled as described in Using R15.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<label>

The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE` (encoded as U == 1).

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE (encoded as U == 0)`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	+/-
0	-
1	+

<imm>

Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as <imm>/4.

<option>

Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                                  else (AlignDownSize(PC32(),4) - imm32));
    let address : bits(32) = if index then offset_addr else AlignDownSize(PC32(),4);

    // System register write to DBGDTRTXint.
    AArch32_SysRegWriteM(cp, ThisInstr(), address);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.67 LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Related system instructions are LDM (User registers) and LDM (exception return).

This instruction is used by the alias POP (multiple registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

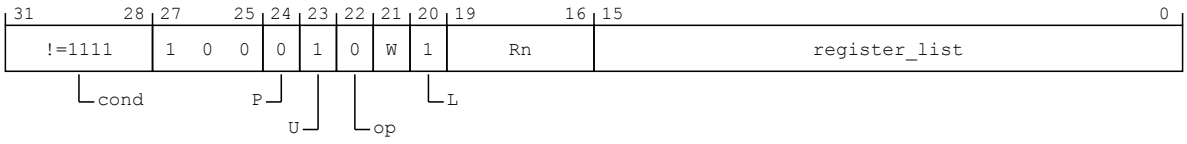

Encoding

```
LDM{IA}{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFD{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

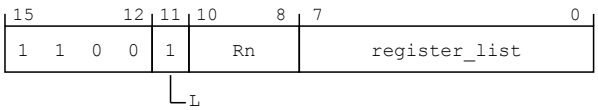

Encoding

```
LDM{IA}{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFD{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = '00000000'::register_list;
let wback : boolean = (registers[n] == '0');
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T2

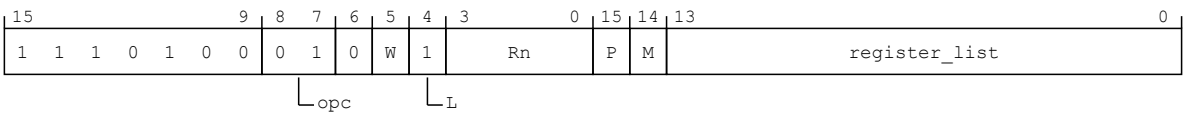

Encoding

```
LDM{IA}{<c>}.W  <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be
    ↪represented in T1)
```

```
LDMFD{<c>}.W  <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and
    ↪<registers> can be represented in T1)
```

```
LDM{IA}{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFD{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

IA

Is an optional suffix for the Increment After form.

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

For the “A1” and “T2” variants: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

For the “T1” variant: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if `<Rn>` is included in `<registers>`, otherwise it must be present.

<registers>

For the “A1” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For the “T1” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the “register_list” field.

For the “T2” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain one of the LR or the PC. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0. If the PC is in the list, the “P” field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Alias Conditions

Alias	Of variant	Is preferred when
`POP (multiple registers)`	`A1`	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`
`POP (multiple registers)`	`T2`	`W == '1' && Rn == '1101' && BitCount(P : M : register_list) > 1`
`POP (multiple registers)`	`T2`	`BitCount(P : M : register_list) > 1`

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) + 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.68 LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

The PE checks the encoding that is copied to the CPSR for an illegal return event. See Illegal return events from AArch32 state.

Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.

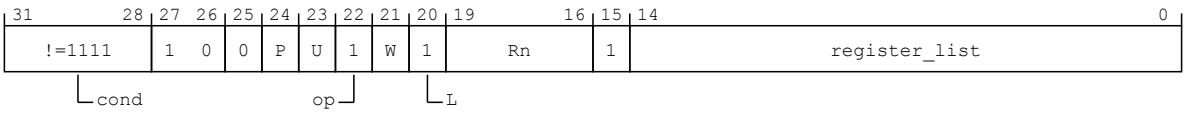

Encoding

```
LDM{<amode>}{<c>}{<q>}  <Rn>{!}, <registers_with_pc>^
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(15) = register_list;
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<amode>

is one of:

DA Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

FA Full Ascending. For this instruction, a synonym for DA.

DB Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EA Empty Ascending. For this instruction, a synonym for `DB`.

IA Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

FD Full Descending. For this instruction, a synonym for `IA`.

IB Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

ED Empty Descending. For this instruction, a synonym for `IB`.

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers_with_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also Encoding of lists of general-purpose registers and the PC.

Instructions with similar syntax but without the PC included in the registers list are described in LDM (User registers).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.M IN {M32_User,M32_System} then
        UnpredictableProcedure();                        // UNDEFINED or NOP
    else
        let length : integer = 4*BitCount(registers) + 4;
        var address : bits(32) = if increment then R(n) else R(n)-length;
        if wordhigher then address = address+4; end;
        var data : bits(32);

        for i = 0 to 14 do
            if registers[i] == '1' then
                if i != n then
                    R(i) = MemS{32}(address);
                else
                    data = MemS{32}(address);
                end;
                address = address + 4;
            end;
        end;
        let new_pc_value : bits(32) = MemS{32}(address);

        if wback && registers[n] == '1' then
            R(n) = ARBITRARY : bits(32);
        end;
        if wback && registers[n] == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            R(n) = if increment then R(n)+length else R(n)-length;
        end;
        if !wback && registers[n] == '1' then
            R(n) = data;
        end;

        AArch32_ExceptionReturn(new_pc_value, SPSR_curr());
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.69 LDM (User registers)

Load Multiple (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC.

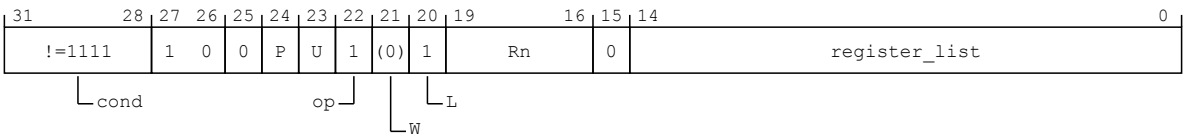

Encoding

```
LDM{<amode>}{<c>}{<q>}  <Rn>, <registers_without_pc>^
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(15) = register_list;
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<amode>

is one of:

DA Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
FA Full Ascending. For this instruction, a synonym for `DA`.
DB Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
EA Empty Ascending. For this instruction, a synonym for `DB`.
IA Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
FD Full Descending. For this instruction, a synonym for `IA`.
IB Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ED Empty Descending. For this instruction, a synonym for `IB`.

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<registers_without_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the `LDM` instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also Encoding of lists of general-purpose registers and the PC.

Instructions with similar syntax but with the PC included in `<registers_without_pc>` are described in LDM (exception return).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then Undefined();
    elsif PSTATE.M IN {M32_User,M32_System} then UnpredictableProcedure();
    else
        let length : integer = 4*BitCount(registers);
        var address : bits(32) = if increment then R(n) else R(n)-length;
        if wordhigher then address = address+4; end;
        for i = 0 to 14 do
            if registers[i] == '1' then  // Load User mode register
                Rmode(i, M32_User) = MemS{32}(address);  address = address + 4;
            end;
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.70 LDMDA, LDMFA

Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Related system instructions are LDM (User registers) and LDM (exception return).

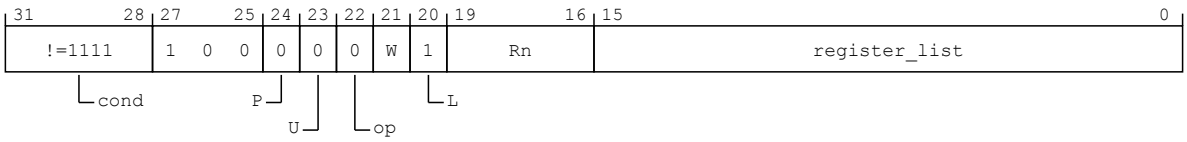

Encoding

```
LDMDA{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFA{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = (R(n) - 4*BitCount(registers)) + 4;
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) - 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.71 LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Related system instructions are LDM (User registers) and LDM (exception return).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

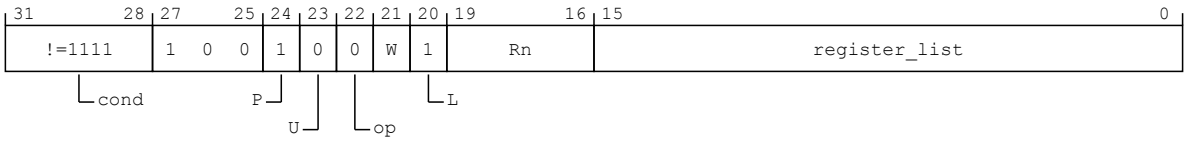

Encoding

```
LDMDB{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMEA{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15						9	8	7	6	5	4	3		0	15	14	13										0
1	1	1	0	1	0	0	1	0	0	W	1		Rn		P	M					register_list						
							opc				L																

Encoding

```
LDMDB{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMEA{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers>

For the "A1" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For the "T1" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) - 4*BitCount(registers);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) - 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.72 LDMIB, LDMED

Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Related system instructions are LDM (User registers) and LDM (exception return).

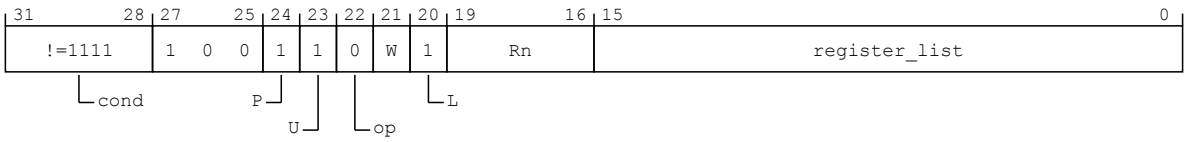

Encoding

```
LDMIB{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMED{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) + 4;
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) + 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.73 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

This instruction is used by the alias POP (single register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, and T4).

A1

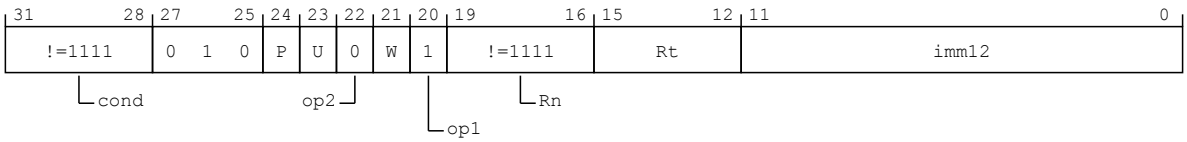

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback && n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

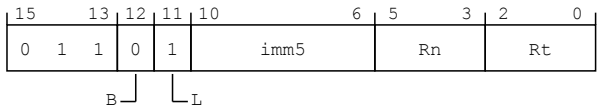

Encoding

```
LDR{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

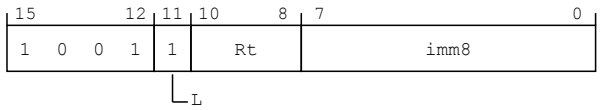

Encoding

```
LDR{<c>}{<q>}  <Rt>, [SP{, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = 13;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T3

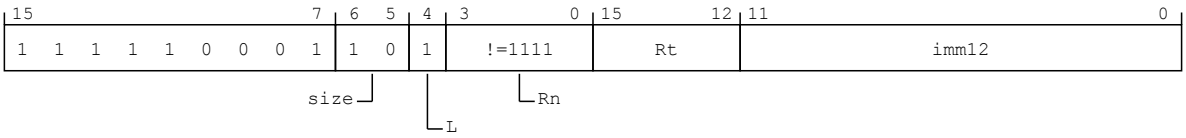

Encoding

```
LDR{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

```
LDR{<c>}.W  <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T4

15								7	6	5	4	3			0	15			12	11	10	9	8	7							0
1	1	1	1	1	0	0	0	0	1	0	1	!=1111				Rt				1	P	U	W	imm8							
									size		L	Rn																			

Encoding for the Offset variant

Applies when `(P == 1 && U == 0 && W == 0)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then
    UnpredictableProcedure();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T1” and “T2” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

For the “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, “A1 Pre-indexed”, “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. For PC use see LDR (literal).

For the “T1” variant: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the “imm5” field as <imm>/4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “T2” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the “imm8” field as <imm>/4.

For the “T3” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Alias Conditions

Alias	Of variant	Is preferred when
POP (single register)	A1 Post-indexed	P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'
POP (single register)	T4 Post-indexed	Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100'

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(32) = MemU{32}(address);
        if wback then R(n) = offset_addr; end;
        if t == 15 then
            if address[1:0] == '00' then
                LoadWritePC(data);
            else
                UnpredictableProcedure();
            end;
        else
            R(t) = data;
        end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(32) = MemU{32}(address);
        if wback then R(n) = offset_addr; end;
        if t == 15 then
            if address[1:0] == '00' then
                LoadWritePC(data);
            else
                UnpredictableProcedure();
            end;
        else
            R(t) = data;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.74 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

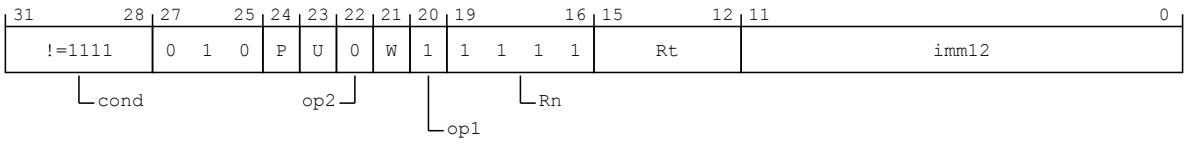

Encoding

Applies when `(!(P == '0' && W == '1'))`

```
LDR{<c>}{<q>}  <Rt>, <label> // (Normal form)
```

```
LDR{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDR (immediate). The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

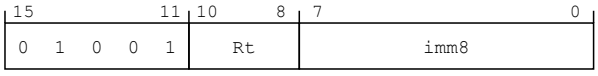

Encoding

```
LDR{<c>}{<q>}  <Rt>, <label> // (Normal form)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = TRUE;
```

T2

15							8	7	6	5	4	3			0	15	12	11	0
1	1	1	1	1	0	0	0	U	1	0	1	1	1	1	1	Rt		imm12	
									size		L								

Encoding

```
LDR{<c>}.W  <Rt>, <label> // (Preferred syntax)
```

```
LDR{<c>}{<q>}  <Rt>, <label> // (Preferred syntax, and <Rt>, <label> can be represented in T1)
```

```
LDR{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field.

For the “T2” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

<label>

For the “A1” and “T2” variants: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For the "T1" variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For the "T2" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(32) = MemU{32}(address);
    if t == 15 then
        if address[1:0] == '00' then
            LoadWritePC(data);
        else
            UnpredictableProcedure();
        end;
    else
        R(t) = data;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.75 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see Memory accesses.

The T32 form of `LDR` (register) does not support register writeback.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 1 1	P	U	0	W	1	Rn	Rt	imm5	stype	0	Rm
cond			op2			op1						

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDR{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

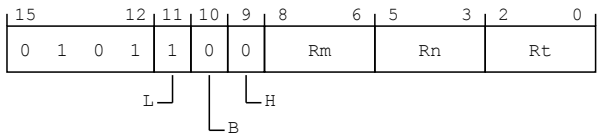

Encoding

```
LDR{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRType  = SRType_LSL;
let shift_n : integer = 0;
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

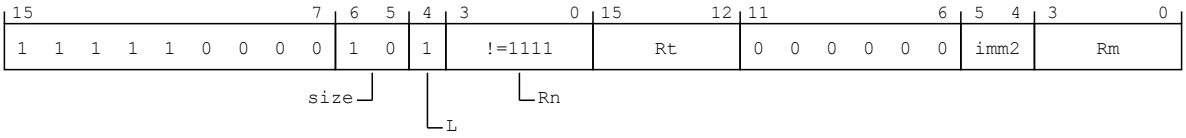

Encoding

```
LDR{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
LDR{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field.

For the “T2” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
        let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(32) = MemU{32}(address);
        if wback then R(n) = offset_addr; end;
        if t == 15 then
            if address[1:0] == '00' then
                LoadWritePC(data);
            else
                UnpredictableProcedure();
            end;
        else
            R(t) = data;
        end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
        let offset_addr : bits(32) = (R(n) + offset);
        let address : bits(32) = offset_addr;
        let data : bits(32) = MemU{32}(address);
        if t == 15 then
            if address[1:0] == '00' then
                LoadWritePC(data);
            else
                UnpredictableProcedure();
            end;
        else
            R(t) = data;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.76 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

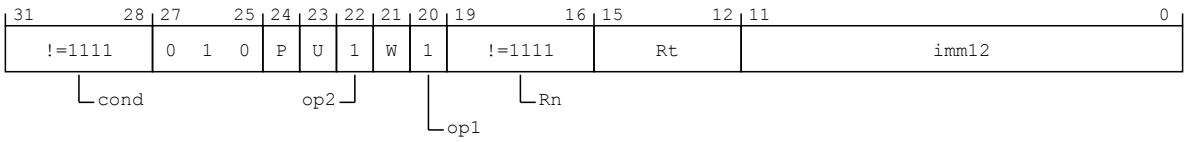

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRB (literal)"); end;
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

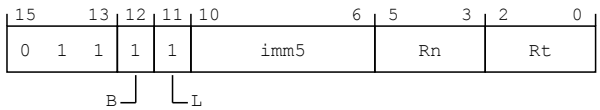

Encoding

```
LDRB{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

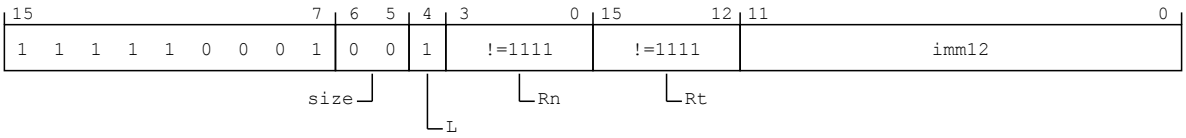

Encoding

```
LDRB{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

```
LDRB{<c>}.W  <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

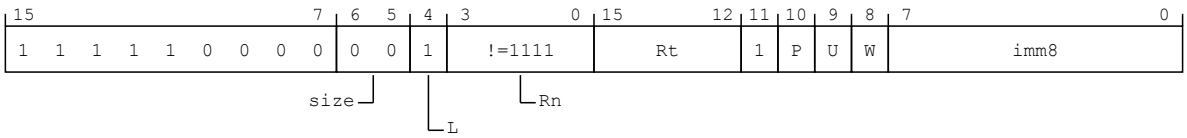

Encoding for the Offset variant

Applies when `(Rt != 1111 && P == 1 && U == 0 && W == 0)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLD, PLDW (immediate)"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRBT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if  (t == 15 &&  W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, “A1 Pre-indexed”, “T2”, “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. For PC use see LDRB (literal).

For the “T1” variant: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T2” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        R(t) = ZeroExtend{32}(MemU{8}(address));
        if wback then R(n) = offset_addr; end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t) = ZeroExtend{32}(MemU{8}(address));
        if wback then R(n) = offset_addr; end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.77 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

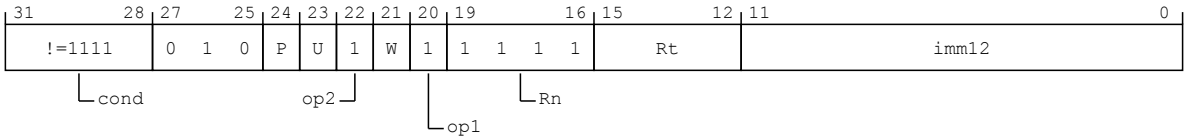

Encoding

Applies when `(!(P == '0' && W == '1'))`

```
LDRB{<c>}{<q>}  <Rt>, <label> // (Normal form)
```

```
LDRB{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRB (immediate). The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

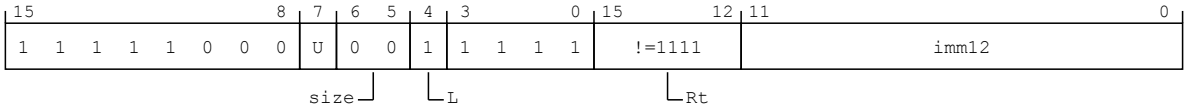

Encoding

```
LDRB{<c>}{<q>}  <Rt>, <label> // (Preferred syntax)
```

```
LDRB{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label>

The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    R(t) = ZeroExtend{32}(MemU{8}(address));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.78 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 1 1	P	U	1	W	1	Rn	Rt	imm5	stype	0	Rm
cond			op2			op1						

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

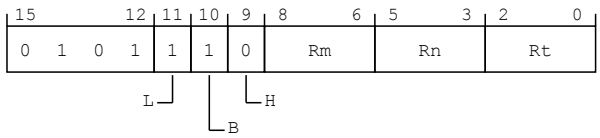

Encoding

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

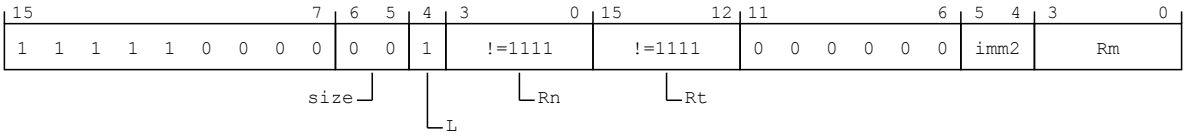

Encoding

```
LDRB{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
LDRB{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = ZeroExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.79 LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRBT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

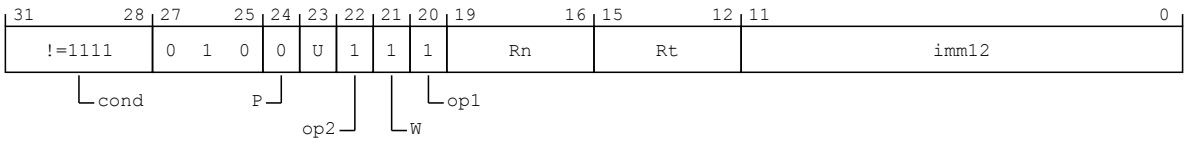

Encoding

```
LDRBT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0 1 1	0 (P)	U (op2)	1	1 (W)	1 (op1)	Rn	Rt	imm5	stype	0	Rm

Encoding

```
LDRBT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–8	7–0
1 1 1	1 1 0 0	0 0	0 0 (size)	1 (L)	!=1111 (Rn)	Rt	1 1 1 0	imm8

Encoding

```
LDRBT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “A2” and “T1” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;                // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                             else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    R(t) = ZeroExtend{32}(MemU_unpriv{8}(address));
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRB` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.80 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

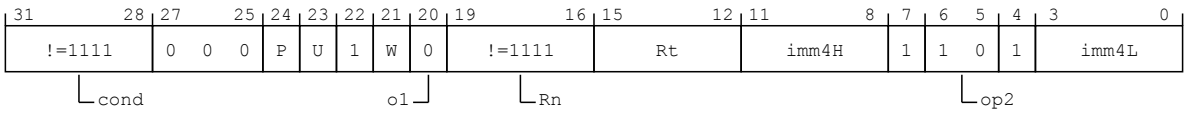

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRD (literal)"); end;
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If Rt<0> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == ‘1111’.

T1

15–9	8	7	6	5	4	3–0	15–12	11–8	7–0
1 1 1 0 1 0 0	P	U	1	W	1	!=1111	Rt	Rt2	imm8
					L	Rn			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
if Rn == '1111' then See("LDRD (literal)"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Load/store dual, load/store exclusive, table branch.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. For PC use see LDRD (literal).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the “imm8” field as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType_GPR)  then
            R(t)  = data[63:32];
            R(t2) = data[31:0];
        else
            R(t)  = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t)  = data1;
        R(t2) = data2;
    end;
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.81 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

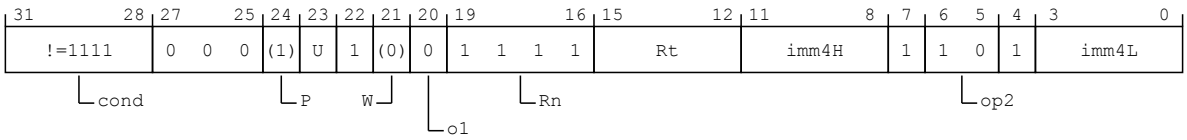

Encoding

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, <label> // (Normal form)
```

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let add : boolean = (U == '1');
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = ‘0’;.
- The instruction executes with the additional decode: t2 = t;.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == ‘1111’.

If `P == '0' || W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as if P == 1 and W == 0.’

T1

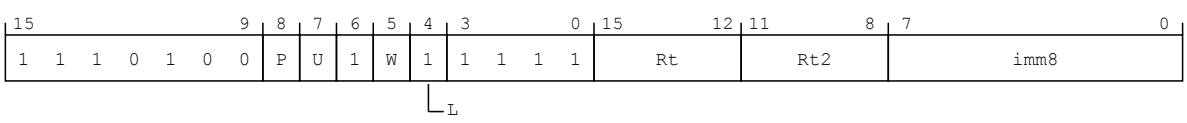

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

Applies when `(!(P == '0' && W == '0'))`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, <label> // (Normal form)
```

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
if W == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Load/Store dual, Load/Store-Exclusive, Load-Acquire/Store-Release, table branch.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<label>

For the "A1" variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For the "T1" variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For the "T1" variant: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                                     else (AlignDownSize(PC32(),4) - imm32));
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType_GPR) then
            R(t)  = data[63:32];
            R(t2) = data[31:0];
        else
            R(t)  = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t)  = data1;
        R(t2) = data2;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.82 LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0	P	U	0	W	0	Rn	Rt	(0)(0)(0)(0)1	1 0	1	Rm
cond						o1				op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if t2 == 15 || m == 15 || m == t || m == t2 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `m == t || m == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads register Rm with an UNKNOWN value.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == ‘1111’.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

<Rt2>

Is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + R(m)) else (R(n) - R(m));
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType_GPR) then
            R(t)  = data[63:32];
            R(t2) = data[31:0];
        else
            R(t)  = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t)  = data1;
        R(t2) = data2;
    end;

    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.83 LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

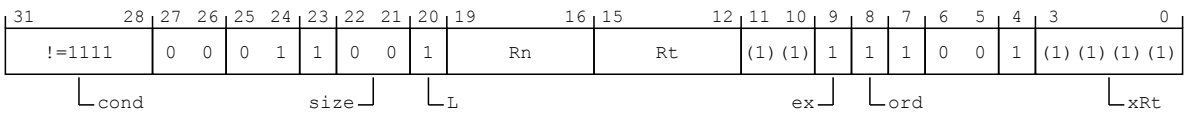

Encoding

```
LDREX{<c>}{<q>}  <Rt>, [<Rn> {, {#}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = Zeros{}; // Zero offset
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

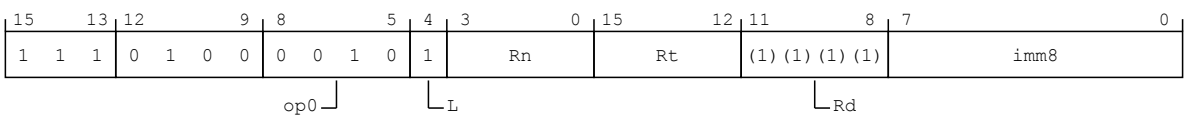

Encoding

```
LDREX{<c>}{<q>}  <Rt>, [<Rn> {, #<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<imm>

For the “A1” variant: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can only be 0 or omitted.

For the “T1” variant: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n) + imm32;
    AArch32_SetExclusiveMonitors(address,4);
    R(t) = MemA{32}(address);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.84 LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

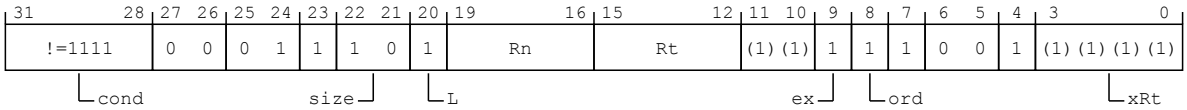

Encoding

```
LDREXB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

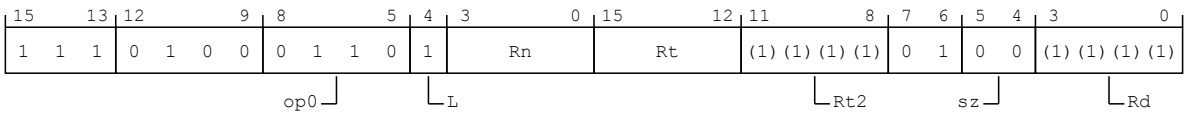

Encoding

```
LDREXB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address,1);
    R(t) = ZeroExtend{32}(MemA{8}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.85 LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	0	1	1	Rn	Rt	(1)	(1)	1	1	1	0	0	1	(1)(1)(1)(1)
cond					size			L					ex	ord					xRt

Encoding

```
LDREXD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
if Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction is handled as described in Using R15.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 0	0 1 1 0	1	Rn	Rt	Rt2	0 1	1 1	(1) (1) (1) (1)
		op0	L					sz	Rd

Encoding

```
LDREXD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. `<Rt>` must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address,8);
    let value : bits(64) = MemA{64}(address);

    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R(t)  = if BigEndian(AccessType_GPR) then value[63:32] else value[31:0];
    R(t2) = if BigEndian(AccessType_GPR) then value[31:0] else value[63:32];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.86 LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

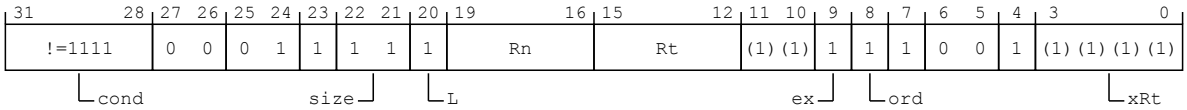

Encoding

```
LDREXH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

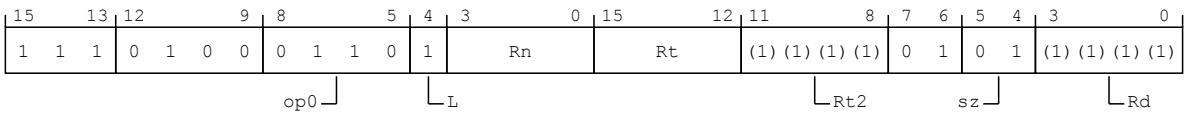

Encoding

```
LDREXH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address,2);
    R(t) = ZeroExtend{32}(MemA{16}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.87 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0	P	U	1	W	1	!=1111	Rt	imm4H	1	0 1	1	imm4L
cond						o1	Rn				op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

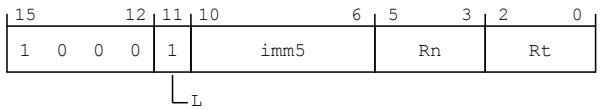

Encoding

```
LDRH{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'0');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

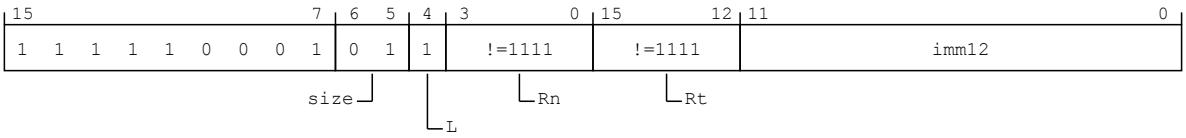

Encoding

```
LDRH{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

```
LDRH{<c>}.W  <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD (immediate)"); end;
if Rn == '1111' then See("LDRH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

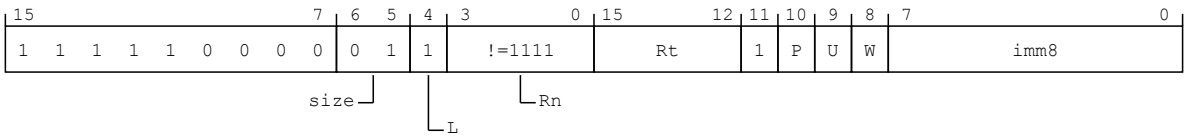

Encoding for the Offset variant

Applies when `(Rt != 1111 && P == 1 && U == 0 && W == 0)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLDW (immediate)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRHT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, “A1 Pre-indexed”, “T2”, “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. For PC use see LDRH (literal).

For the “T1” variant: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the “imm5” field as <imm>/2.

For the “T2” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(16) = MemU{16}(address);
        if wback then R(n) = offset_addr; end;
        R(t) = ZeroExtend{32}(data);
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(16) = MemU{16}(address);
        if wback then R(n) = offset_addr; end;
        R(t) = ZeroExtend{32}(data);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.88 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

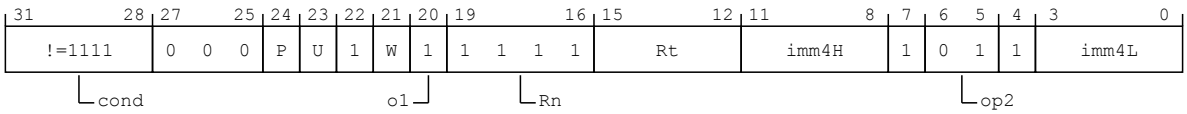

Encoding

Applies when `(!(P == '0' && W == '1'))`

```
LDRH{<c>}{<q>}  <Rt>, <label> // (Normal form)
```

```
LDRH{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRH (immediate). The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

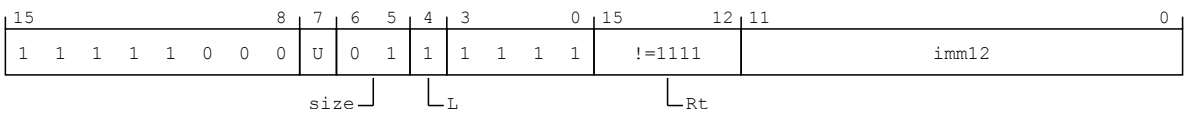

Encoding

```
LDRH{<c>}{<q>}  <Rt>, <label> // (Preferred syntax)
```

```
LDRH{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rt == '1111' then See("PLD (literal)"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<label>

For the “A1” variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For the “T1” variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(16) = MemU{16}(address);
    R(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.89 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6	5	4	3–0
!=1111	0 0 0	P	U	0	W	1	Rn	Rt	(0)(0)(0)(0)1	0	1	1	Rm
cond						o1					op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

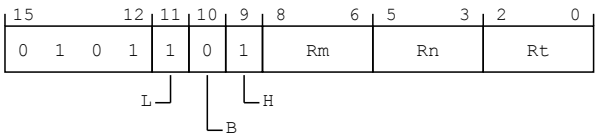

Encoding

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

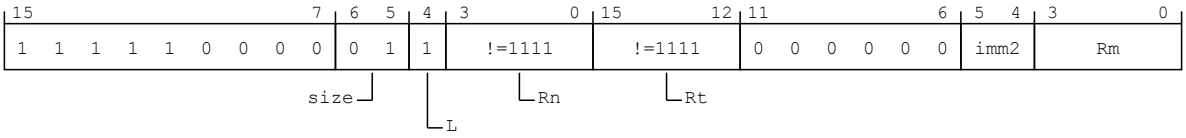

Encoding

```
LDRH{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
LDRH{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
if Rt == '1111' then See("PLDW (register)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.90 LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRHT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

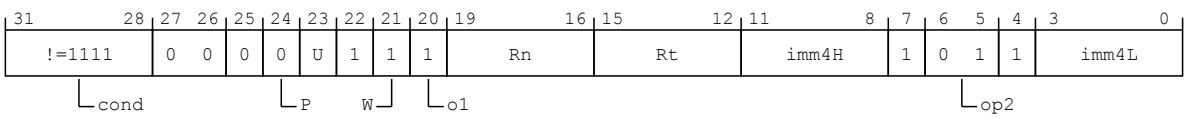

Encoding

```
LDRHT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	0	1	1	Rn	Rt	(0)(0)(0)(0)	1	0	1	1	Rm
cond				P		W		o1						op2		

Encoding

```
LDRHT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15–13	12–9	8	7	6	5	4	3–0	15–12	11–8	7–0
1 1 1	1 1 0 0	0	0	0	1	1	!=1111	Rt	1 1 1 0	imm8
				size		L	Rn			

Encoding

```
LDRHT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;               // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(16) = MemU_unpriv{16}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRH` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.91 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0	P	U	1	W	1	!=1111	Rt	imm4H	1	1 0	1	imm4L
cond						o1	Rn				op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSB (literal)"); end;
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–0
1 1 1	1 1 0 0	1 1	0 0	1	!=1111	!=1111	imm12
			size		Rn	Rt	

Encoding

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2

15–7	6–5	4	3–0	15–12	11	10	9	8	7–0
1 1 1 1 1 0 0 1 0	0 0	1	!=1111	Rt	1	P	U	W	imm8
	size		Rn						

Encoding for the Offset variant

Applies when `(Rt != 1111 && P == 1 && U == 0 && W == 0)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRSBT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. For PC use see LDRSB (literal).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “T1” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2 Offset”, “T2 Post-indexed”, and “T2 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = SignExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.92 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

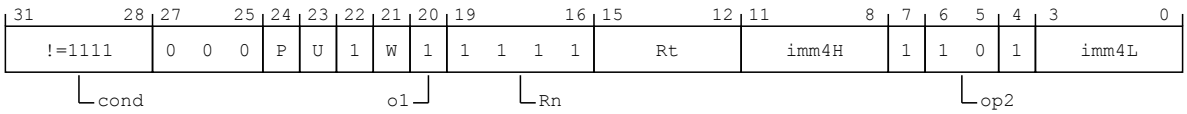

Encoding

Applies when `(!(P == '0' && W == '1'))`

```
LDRSB{<c>}{<q>}  <Rt>, <label> // (Normal form)
```

```
LDRSB{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSB (immediate). The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

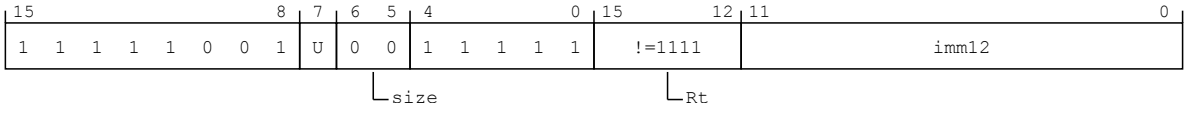

Encoding

```
LDRSB{<c>}{<q>}  <Rt>, <label> // (Preferred syntax)
```

```
LDRSB{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label>

For the "A1" variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For the "T1" variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    R(t) = SignExtend{32}(MemU{8}(address));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.93 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0	P	U	0	W	1	Rn	Rt	(0)(0)(0)(0)1	1 0	1	Rm
cond						o1				op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

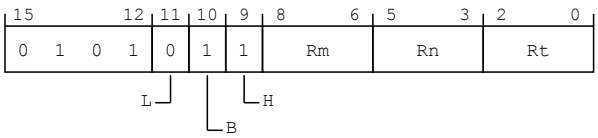

Encoding

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

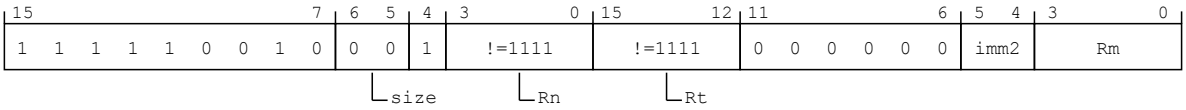

Encoding

```
LDRSB{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
LDRSB{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = SignExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.94 LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRSBT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

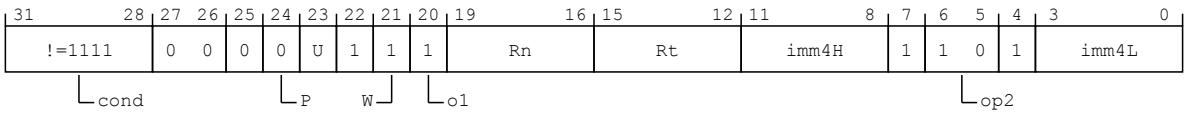

Encoding

```
LDRSBT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	0	1	1	Rn	Rt	(0)(0)(0)(0)	1	1	0	1	Rm
cond				P		W		o1						op2		

Encoding

```
LDRSBT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15–13	12–9	8	7	6	5	4	3–0	15–12	11–8	7–0
1 1 1	1 1 0 0	1	0	0	0	1	!=1111	Rt	1 1 1 0	imm8
				size			Rn			

Encoding

```
LDRSBT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;              // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    R(t) = SignExtend{32}(MemU_unpriv{8}(address));
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRSB` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.95 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0	P	U	1	W	1	!=1111	Rt	imm4H	1	1 1	1	imm4L
cond						o1	Rn				op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–0
1 1 1	1 1 0 0	1 1	0 1	1	!=1111	!=1111	imm12
			size		Rn	Rt	

Encoding

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2

15–7	6–5	4	3–0	15–12	11	10	9	8	7–0
1 1 1 1 1 0 0 1 0	0 1	1	!=1111	Rt	1	P	U	W	imm8
	size		Rn						

Encoding for the Offset variant

Applies when `(Rt != 1111 && P == 1 && U == 0 && W == 0)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("Related instructions"); end;
if P == '1' && U == '1' && W == '0' then See("LDRSHT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related instructions: Load/store single.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. For PC use see LDRSH (literal).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “T1” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2 Offset”, “T2 Post-indexed”, and “T2 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.96 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

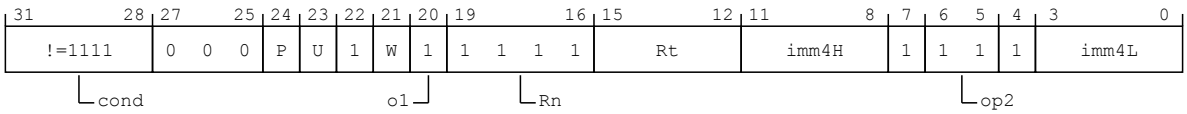

Encoding

Applies when `(!(P == '0' && W == '1'))`

```
LDRSH{<c>}{<q>}  <Rt>, <label> // (Normal form)
```

```
LDRSH{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSH (immediate). The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

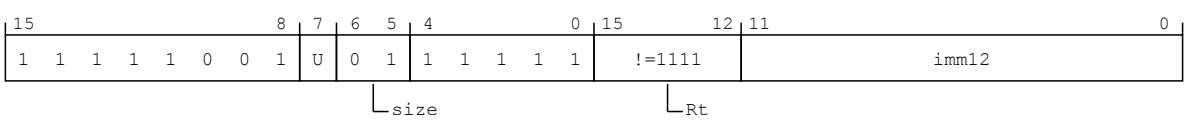

Encoding

```
LDRSH{<c>}{<q>}  <Rt>, <label> // (Preferred syntax)
```

```
LDRSH{<c>}{<q>}  <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related instructions: Load, signed (literal).

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<label>

For the “A1” variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For the “T1” variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(16) = MemU{16}(address);
    R(t) = SignExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.97 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6	5	4	3–0
!=1111 (cond)	0 0 0	P	U	0	W	1 (o1)	Rn	Rt	(0)(0)(0)(0)1	1	1 (op2)	1	Rm

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

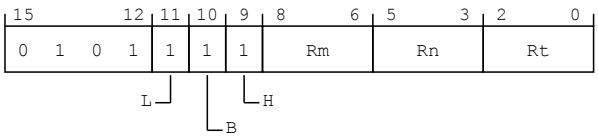

Encoding

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

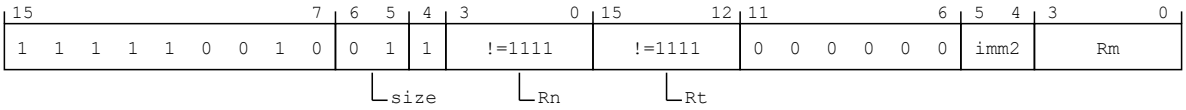

Encoding

```
LDRSH{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
LDRSH{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related instructions: Load/store, signed (register offset).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.98 LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRSHT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

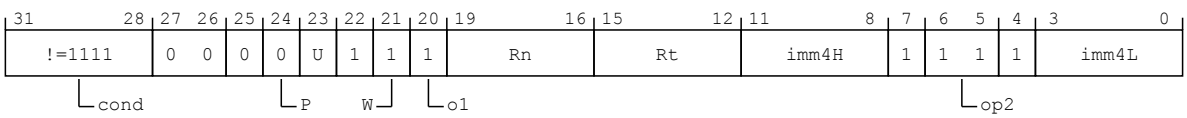

Encoding

```
LDRSHT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	0	1	1	Rn	Rt	(0)(0)(0)(0)	1	1	1	1	Rm
cond				P		W		o1						op2		

Encoding

```
LDRSHT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–8	7–0
1 1 1	1 1 0 0	1 0	0 1	1	!=1111	Rt	1 1 1 0	imm8
			size		Rn			

Encoding

```
LDRSHT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;              // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(16) = MemU_unpriv{16}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRSH` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.99 LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`LDRT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

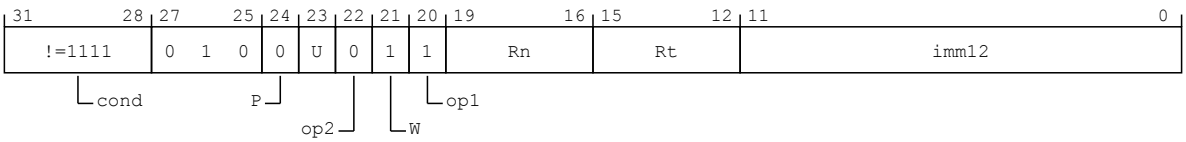

Encoding

```
LDRT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == ‘1’ and bit[21] == ‘0’. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

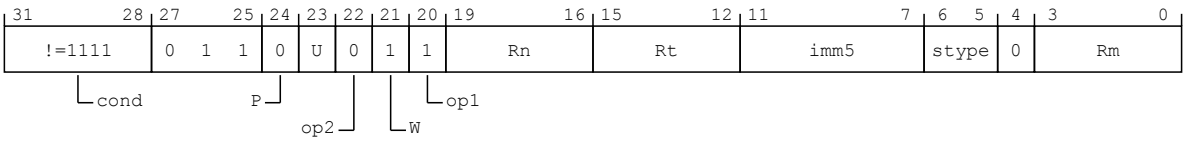

Encoding

```
LDRT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

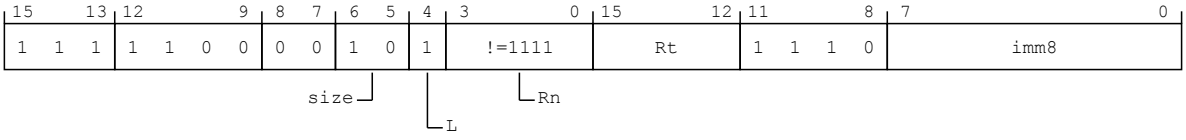

Encoding

```
LDRT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “A2” and “T1” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;                // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                             else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(32) = MemU_unpriv{32}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = data;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDR` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.100 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

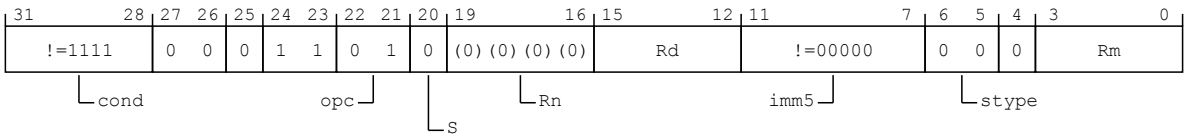

Encoding for the MOV, shift or rotate by value variant

```
LSL{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2

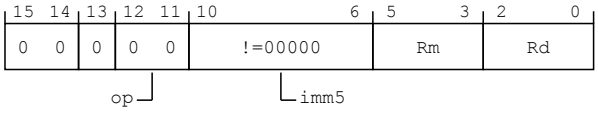

Encoding for the T2 variant

```
LSL<c>{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV<c>{<q>}  <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3

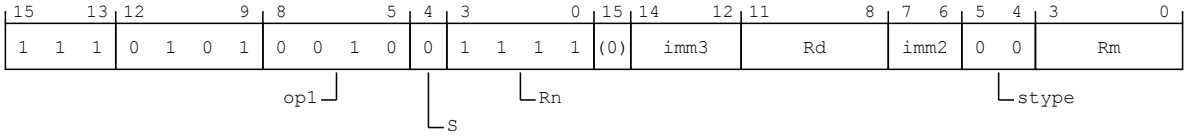

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the MOV, shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00))`

```
LSL{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

```
LSL<c>.W  {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOV, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm5” field as <imm> modulo 32.

For the “T2” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field as <amount> modulo 32.

For the “T3 MOV, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.101 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

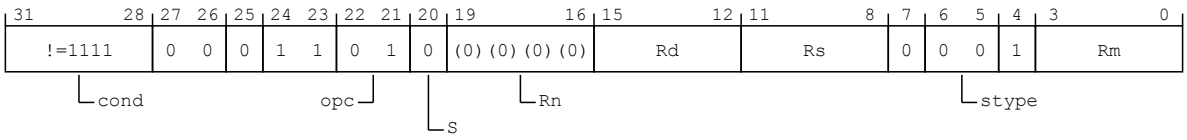

Encoding for the Not flag setting variant

```
LSL{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1

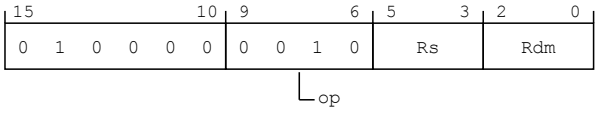

Encoding for the Logical shift left variant

```
LSL<c>{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOV<c>{<q>}  <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

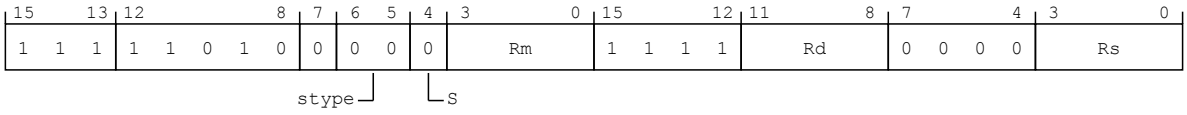

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Not flag setting variant

```
LSL{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
LSL<c>.W  {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.102 LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

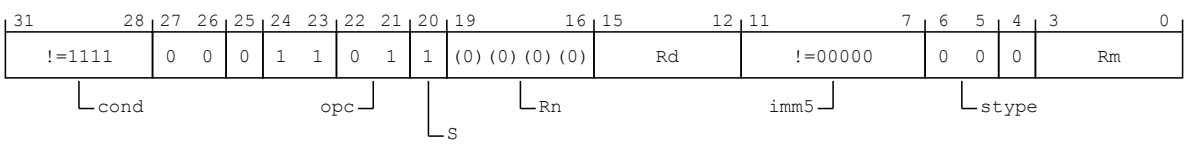

Encoding for the MOVS, shift or rotate by value variant

```
LSLS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2

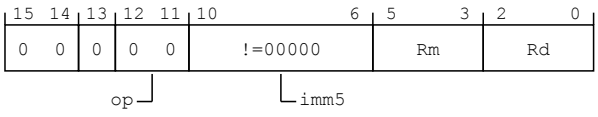

Encoding for the T2 variant

```
LSLS{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS<c>{<q>}  <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15–13	12–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 1	0 0 1 0	1	1 1 1 1	(0)	imm3	Rd	imm2	0 0	Rm
		op1	S	Rn					stype	

Encoding for the MOVS, shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00))`

```
LSLS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

```
LSLS.W  {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm5” field as <imm> modulo 32.

For the “T2” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field as <amount> modulo 32.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.103 LSLS (register)

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

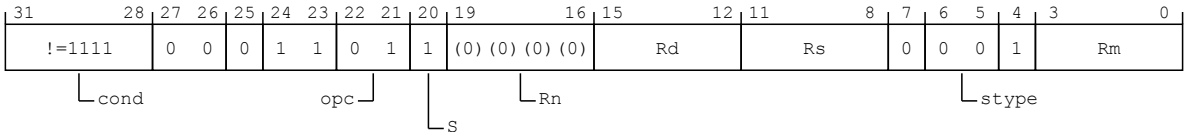

Encoding for the Flag setting variant

```
LSLS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1

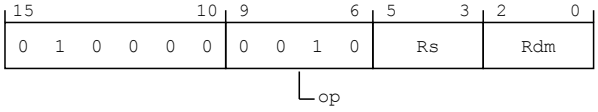

Encoding for the Logical shift left variant

```
LSLS{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOVS{<q>}  <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15 13	12 8	7	6 5	4	3 0	15 12	11 8	7 4	3 0
1 1 1	1 1 0 1 0	0	0 0	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype	S					

Encoding for the Flag setting variant

```
LSLS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
LSLS.W  {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.104 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

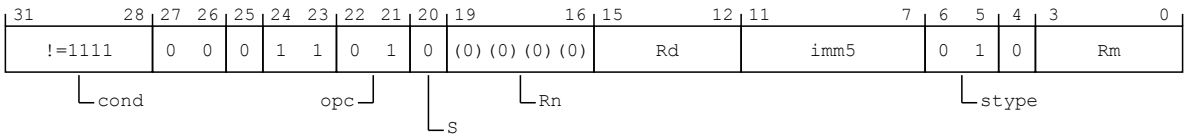

Encoding for the MOV, shift or rotate by value variant

```
LSR{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2

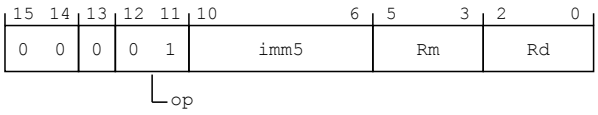

Encoding for the T2 variant

```
LSR<c>{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV<c>{<q>}  <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3

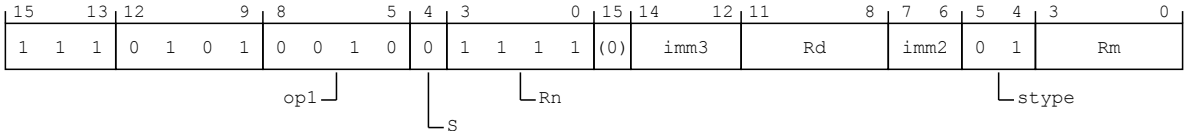

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the MOV, shift or rotate by value variant

```
LSR{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

```
LSR<c>.W  {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOV, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOV, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.105 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

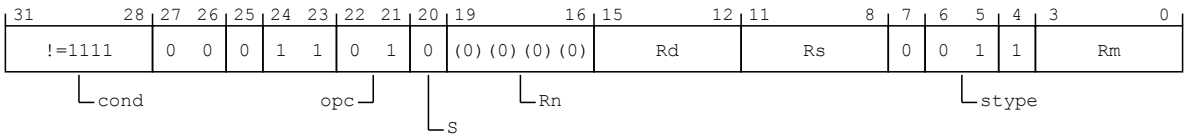

Encoding for the Not flag setting variant

```
LSR{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

T1

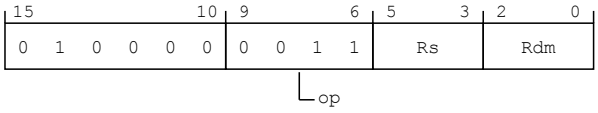

Encoding for the Logical shift right variant

```
LSR<c>{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOV<c>{<q>}  <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

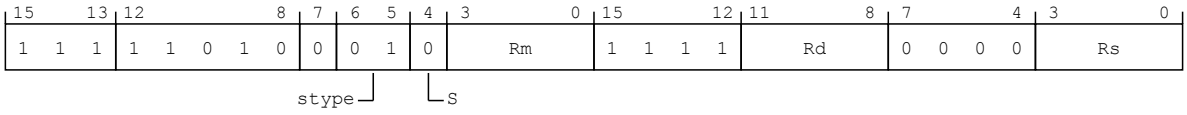

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Not flag setting variant

```
LSR{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
LSR<c>.W  {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.106 LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

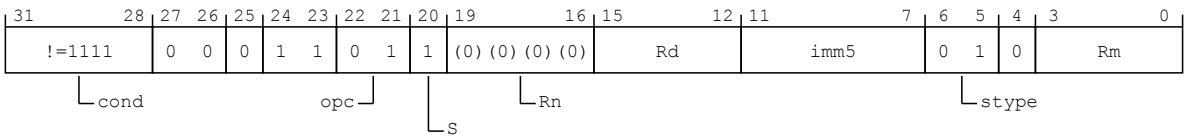

Encoding for the MOVS, shift or rotate by value variant

```
LSRS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2

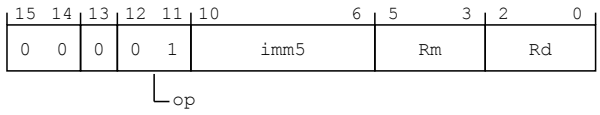

Encoding for the T2 variant

```
LSRS{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS<c>{<q>}  <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15–13	12–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 1	0 0 1 0	1	1 1 1 1	(0)	imm3	Rd	imm2	0 1	Rm
		op1	S	Rn					stype	

Encoding for the MOVS, shift or rotate by value variant

```
LSRS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

```
LSRS.W  {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.107 LSRS (register)

Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

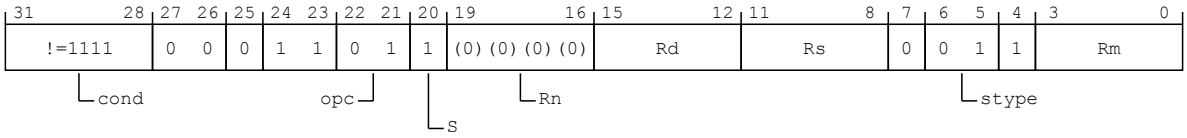

Encoding for the Flag setting variant

```
LSRS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

T1

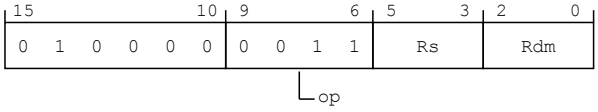

Encoding for the Logical shift right variant

```
LSRS{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOVS{<q>}  <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12–8	7	6–5	4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1 0	0	0 1	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype	S					

Encoding for the Flag setting variant

```
LSRS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
LSRS.W  {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.108 MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

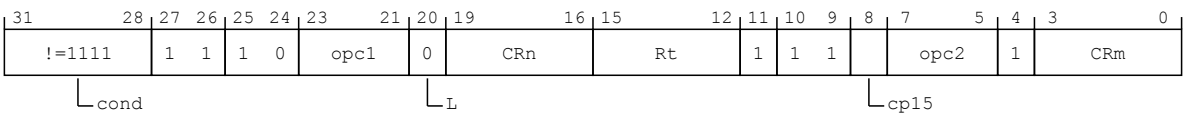

Encoding

```
MCR{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15  then UnpredictableProcedure(); end;
```

T1

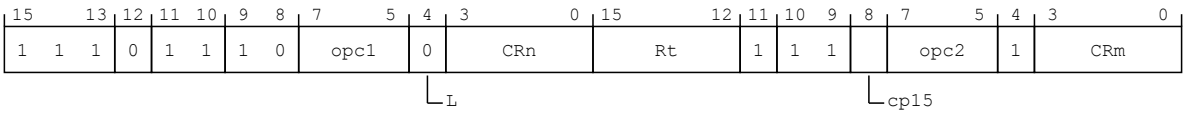

Encoding

```
MCR{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15  then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc1” field.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<CRn>

Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRn” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

<opc2>

Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc2” field.

The possible values of { `<coproc>`, `<opc1>`, `<CRn>`, `<CRm>`, `<opc2>` } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegWrite(cp, ThisInstr(), t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.109 MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

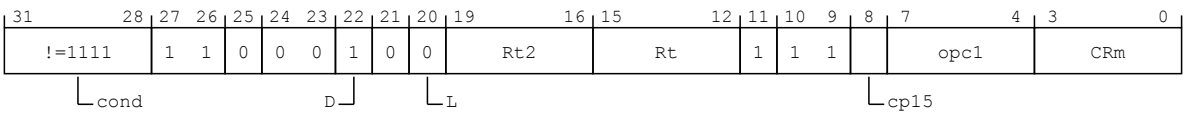

Encoding

```
MCRR{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

T1

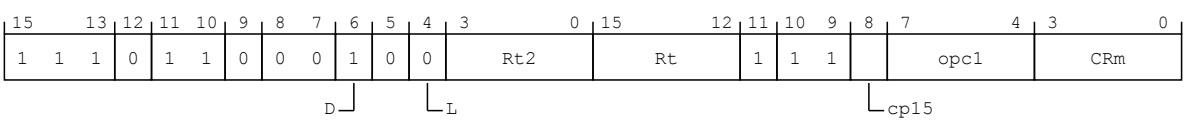

Encoding

```
MCRR{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the “opc1” field.

<Rt>

Is the first general-purpose register that is transferred into, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that is transferred into, encoded in the “Rt2” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

The possible values of { `<coproc>`, `<opc1>`, `<CRm>` } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, `<Rt2>` transfers bits[63:32] of the selected System register, while `<Rt>` transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegWrite64(cp, ThisInstr(), t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.110 MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

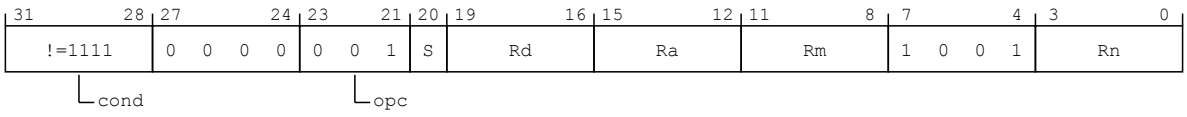

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
MLAS{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
MLA{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let setflags : boolean = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

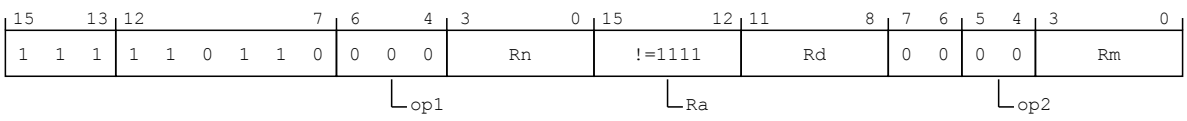

Encoding

```
MLA{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Ra == '1111' then See("MUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let setflags : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m  == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let addend : integer   = SInt(R(a)); // addend   = UInt(R[a]) produces the same final results
    let result : integer = operand1 * operand2 + addend;
    R(d) = result[31:0];
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result[31:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.111 MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	0	0	1	1	0 (S)	Rd	Ra	Rm	1	0	0	1	Rn

opc

Encoding

```
MLS{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–7	6–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	1 1 0 1 1 0	0 0 0 (op1)	Rn	Ra	Rd	0 0	0 1 (op2)	Rm

Encoding

```
MLS{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let addend : integer   = SInt(R(a)); // addend   = UInt(R[a]) produces the same final results
    let result : integer = addend - operand1 * operand2;
    R(d) = result[31:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.112 MOV, MOVS (immediate)

Move (immediate) writes an immediate value to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1, T2, and T3).

A1

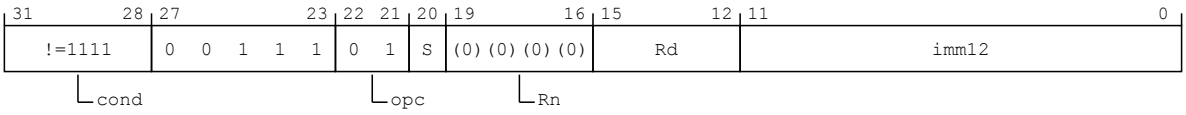

Encoding for the MOV variant

Applies when `(S == 0)`

```
MOV{<c>}{<q>}  <Rd>, #<const>
```

Encoding for the MOVS variant

Applies when `(S == 1)`

```
MOVS{<c>}{<q>}  <Rd>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let (imm32, carry) : (bits(32), bit) = A32ExpandImm_C(imm12, PSTATE.C);
```

A2

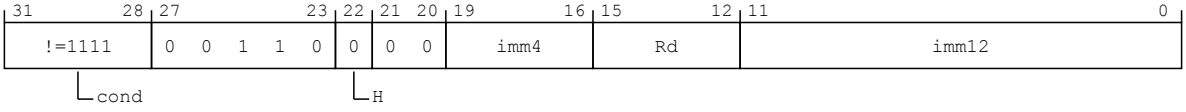

Encoding

```
MOV{<c>}{<q>}  <Rd>, #<imm16> // (<imm16> can not be represented in A1)
```

```
MOVW{<c>}{<q>}  <Rd>, #<imm16> // (<imm16> can be represented in A1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4::imm12);
let carry : bit = ARBITRARY : bit;
if d == 15 then UnpredictableProcedure(); end;
```

T1

15 13	12 11	10 8	7 0
0 0 1	0 0	Rd	imm8
	op		

Encoding

```
MOV<c>{<q>}  <Rd>, #<imm8> // (InITBlock())
```

```
MOVS{<q>}  <Rd>, #<imm8> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);
let carry : bit = PSTATE.C;
```

T2

15 11	10	9	8 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	0 0 1 0	S	1 1 1 1	0	imm3	Rd	imm8
			op1		Rn				

Encoding for the MOV variant

Applies when `(S == 0)`

```
MOV{<c>}{<q>}  <Rd>, #<const>
```

```
MOV<c>.W  <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)
```

Encoding for the MOVS variant

Applies when `(S == 1)`

```
MOVS{<c>}{<q>}  <Rd>, #<const>
```

```
MOVS.W  <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let (imm32, carry) : (bits(32), bit) = T32ExpandImm_C(i::imm3::imm8, PSTATE.C);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T3

15				11	10	9	8	7	6		4	3		0	15	14	12	11		8	7		0
1	1	1	1	0	i	1	0	0	1	0	0	imm4			0	imm3		Rd			imm8		

o1

Encoding

```
MOV{<c>}{<q>}  <Rd>, #<imm16> // (<imm16> cannot be represented in T1 or T2)
```

```
MOVW{<c>}{<q>}  <Rd>, #<imm16> // (<imm16> can be represented in T1 or T2)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4::i::imm3::imm8);
let carry : bit = ARBITRARY : bit;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOV” and “A1 MOVS” variants: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “A2”, “T1”, “T2 MOV”, “T2 MOVS”, and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<const>

For the “A1 MOV” and “A1 MOVS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T2 MOV” and “T2 MOVS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

<imm16>

For the “A2” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field.

For the “T3” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:i:imm3:imm8” field.

<imm8>

Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : bits(32) = imm32;
    if d == 15 then          // Can only occur for encoding A1
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.113 MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

31–28	27–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 1 1	0 1	S	(0)(0)(0)(0)	Rd	imm5	stype	0	Rm
cond		opc		Rn					

Encoding for the MOV, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
MOV{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MOV, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
MOV{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

Encoding for the MOVS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MOVS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
MOVS{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```

T1

15					10	9	8	7	6			3	2		0
0	1	0	0	0	1	1	0	D	Rm				Rd		

op

Encoding

```
MOV{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(D::Rd);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15		13	12	11	10			6	5		3	2		0
0	0	0	!=11		imm5				Rm			Rd		

op

Encoding

```
MOV<c>{<q>}  <Rd>, <Rm> {, <shift> #<amount>} // (InITBlock())
```

```
MOVS{<q>}  <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '00' && imm5 == '00000' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as `NOP`, as if it failed its condition code check.
- The instruction executes as `MOV` Rd, Rm.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	0 0 1 0	S	1 1 1 1	(0)	imm3	Rd	imm2	stype	Rm
	op1		Rn						

Encoding for the MOV, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
MOV{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MOV, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
MOV{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

```
MOV{<c>}.W  <Rd>, <Rm> {, LSL #0} // (<Rd>, <Rm> can be represented in T1)
```

```
MOV<c>.W  <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be
    ↪represented in T2)
```

Encoding for the MOVS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MOVS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
MOVS{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

```
MOVS.W  <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be
    ↪represented in T1 or T2)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the "A1 MOV, rotate right with extend", "A1 MOV, shift or rotate by value", "A1 MOVS, rotate right with extend", and "A1 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Arm deprecates use of the instruction if `<Rn>` is the PC.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or `RRX` argument is specified.

For the "T1" variant: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The instruction must either be outside an IT block or the last instruction of an IT block.

For the "T2", "T3 MOV, rotate right with extend", "T3 MOV, shift or rotate by value", "T3 MOVS, rotate right with extend", and "T3 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.

<Rm>

For the "A1 MOV, rotate right with extend", "A1 MOV, shift or rotate by value", "A1 MOVS, rotate right with extend", "A1 MOVS, shift or rotate by value", and "T1" variants: is the general-purpose source register, encoded in the "Rm" field. The PC can be used. Arm deprecates use of the instruction if `<Rd>` is the PC.

For the "T2", "T3 MOV, rotate right with extend", "T3 MOV, shift or rotate by value", "T3 MOVS, rotate right with extend", and "T3 MOVS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.

<shift>

For the "A1 MOV, shift or rotate by value", "A1 MOVS, shift or rotate by value", "T3 MOV, shift or rotate by value", and "T3 MOVS, shift or rotate by value" variants: is the type of shift to be applied to the source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

For the "T2" variant: is the type of shift to be applied to the source register, encoded in "op":

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	<shift>
00	LSL
01	LSR
10	ASR

<amount>

For the “A1 MOV, shift or rotate by value” and “A1 MOVS, shift or rotate by value” variants: is the shift amount, in the range 0 to 31 (when <shift> = LSL), or 1 to 31 (when <shift> = ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T3 MOV, shift or rotate by value” and “T3 MOVS, shift or rotate by value” variants: is the shift amount, in the range 0 to 31 (when <shift> = LSL) or 1 to 31 (when <shift> = ROR), or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (immediate)	T3 MOVS, shift or rotate by value, A1 MOVS, shift or rotate by value	S == '1' && stype == '10'
ASRS (immediate)	T2	op == '10' && !InITBlock()
ASRS (immediate)	T3 MOVS, shift or rotate by value	Unconditionally
ASR (immediate)	T3 MOV, shift or rotate by value, A1 MOV, shift or rotate by value	S == '0' && stype == '10'
ASR (immediate)	T2	op == '10' && InITBlock()
ASR (immediate)	T3 MOV, shift or rotate by value	Unconditionally
LSLS (immediate)	A1 MOVS, shift or rotate by value	S == '1' && imm5 != '00000' && stype == '00'
LSLS (immediate)	T2	op == '00' && imm5 != '00000' && !InITBlock()
LSLS (immediate)	T3 MOVS, shift or rotate by value	(S == '1' && stype == '00' && !(imm3 == '000')) || (S == '1' && stype == '00' && !(imm2 == '00'))
LSLS (immediate)	T3 MOVS, shift or rotate by value	Unconditionally
LSL (immediate)	A1 MOV, shift or rotate by value	S == '0' && imm5 != '00000' && stype == '00'
LSL (immediate)	T2	op == '00' && imm5 != '00000' && InITBlock()
LSL (immediate)	T3 MOV, shift or rotate by value	(S == '0' && stype == '00' && !(imm3 == '000')) || (S == '0' && stype == '00' && !(imm2 == '00'))

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Of variant	Is preferred when
LSL (immediate)	T3 MOV, shift or rotate by value	Unconditionally
LSRS (immediate)	T3 MOVS, shift or rotate by value, A1 MOVS, shift or rotate by value	S == '1' && stype == '01'
LSRS (immediate)	T2	op == '01' && !InITBlock()
LSRS (immediate)	T3 MOVS, shift or rotate by value	Unconditionally
LSR (immediate)	T3 MOV, shift or rotate by value, A1 MOV, shift or rotate by value	S == '0' && stype == '01'
LSR (immediate)	T2	op == '01' && InITBlock()
LSR (immediate)	T3 MOV, shift or rotate by value	Unconditionally
RORS (immediate)	A1 MOVS, shift or rotate by value	S == '1' && imm5 != '00000' && stype == '11'
RORS (immediate)	T3 MOVS, shift or rotate by value	(S == '1' && stype == '11' && !(imm3 == '000')) || (S == '1' && stype == '11' && !(imm2 == '00'))
ROR (immediate)	A1 MOV, shift or rotate by value	S == '0' && imm5 != '00000' && stype == '11'
ROR (immediate)	T3 MOV, shift or rotate by value	(S == '0' && stype == '11' && !(imm3 == '000')) || (S == '0' && stype == '11' && !(imm2 == '00'))
RRXS	A1 MOVS, rotate right with extend	S == '1' && imm5 == '00000' && stype == '11'
RRXS	T3 MOVS, rotate right with extend	S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'
RRX	A1 MOV, rotate right with extend	S == '0' && imm5 == '00000' && stype == '11'
RRX	T3 MOV, rotate right with extend	S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.114 MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), and ROR (register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

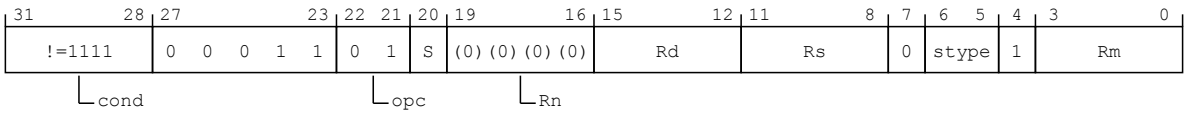

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
MOV{<c>}{<q>}  <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

T1

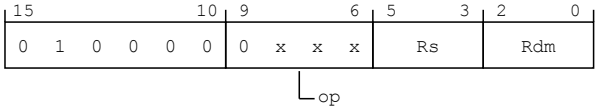

Encoding for the Arithmetic shift right variant

Applies when `(op == 0100)`

```
MOV<c>{<q>}  <Rdm>, <Rdm>, ASR <Rs> // (InITBlock())
```

```
MOVS{<q>}  <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Logical shift left variant

Applies when `(op == 0010)`

```
MOV<c>{<q>}  <Rdm>, <Rdm>, LSL <Rs> // (InITBlock())
```

```
MOVS{<q>}  <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)
```

Encoding for the Logical shift right variant

Applies when `(op == 0011)`

```
MOV<c>{<q>}  <Rdm>, <Rdm>, LSR <Rs> // (InITBlock())
```

```
MOVS{<q>}  <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)
```

Encoding for the Rotate right variant

Applies when `(op == 0111)`

```
MOV<c>{<q>}  <Rdm>, <Rdm>, ROR <Rs> // (InITBlock())
```

```
MOVS{<q>}  <Rdm>, <Rdm>, ROR <Rs> // (Outside IT block)
```

Decode for all variants of this encoding

```
if ! op IN {'0010', '0011', '0100', '0111'} then See("Related encodings"); end;
let d : integer = UInt(Rdm);
let m : integer = UInt(Rdm);
let s : integer = UInt(Rs);
let setflags : boolean = !InITBlock();
let shift_t : SRType = DecodeRegShift(op[2]::op[0]);
```

T2

15 … 7	6 5	4	3 … 0	15 … 12	11 … 8	7 … 4	3 … 0
1 1 1 1 1 0 1 0 0	stype	S	Rm	1 1 1 1	Rd	0 0 0 0	Rs

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, <shift> <Rs>
```

```
MOVS.W  <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented
    ↪in T1)
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
MOV{<c>}{<q>}  <Rd>, <Rm>, <shift> <Rs>
```

```
MOV<c>.W  <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented
    ↪in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

Related encodings: In encoding T1, for an `op` field value that is not described above, see Data-processing (two low registers).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the general-purpose source register and the destination register, encoded in the “Rdm” field.

Alias Conditions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Of variant	Is preferred when
ASRS (register)	A1 Flag setting	S == '1' && stype == '10'
ASRS (register)	T1 Arithmetic shift right	op == '0100' && !InITBlock()
ASRS (register)	T2 Flag setting	stype == '10' && S == '1'
ASRS (register)	T2 Flag setting	Unconditionally
ASR (register)	A1 Not flag setting	S == '0' && stype == '10'
ASR (register)	T1 Arithmetic shift right	op == '0100' && InITBlock()
ASR (register)	T2 Not flag setting	stype == '10' && S == '0'
ASR (register)	T2 Not flag setting	Unconditionally
LSLS (register)	A1 Flag setting	S == '1' && stype == '00'
LSLS (register)	T1 Logical shift left	op == '0010' && !InITBlock()
LSLS (register)	T2 Flag setting	stype == '00' && S == '1'
LSLS (register)	T2 Flag setting	Unconditionally
LSL (register)	A1 Not flag setting	S == '0' && stype == '00'
LSL (register)	T1 Logical shift left	op == '0010' && InITBlock()
LSL (register)	T2 Not flag setting	stype == '00' && S == '0'
LSL (register)	T2 Not flag setting	Unconditionally
LSRS (register)	A1 Flag setting	S == '1' && stype == '01'
LSRS (register)	T1 Logical shift right	op == '0011' && !InITBlock()
LSRS (register)	T2 Flag setting	stype == '01' && S == '1'
LSRS (register)	T2 Flag setting	Unconditionally
LSR (register)	A1 Not flag setting	S == '0' && stype == '01'
LSR (register)	T1 Logical shift right	op == '0011' && InITBlock()
LSR (register)	T2 Not flag setting	stype == '01' && S == '0'
LSR (register)	T2 Not flag setting	Unconditionally
RORS (register)	A1 Flag setting	S == '1' && stype == '11'
RORS (register)	T1 Rotate right	op == '0111' && !InITBlock()
RORS (register)	T2 Flag setting	stype == '11' && S == '1'
RORS (register)	T2 Flag setting	Unconditionally
ROR (register)	A1 Not flag setting	S == '0' && stype == '11'
ROR (register)	T1 Rotate right	op == '0111' && InITBlock()
ROR (register)	T2 Not flag setting	stype == '11' && S == '0'
ROR (register)	T2 Not flag setting	Unconditionally

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (result, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.115 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

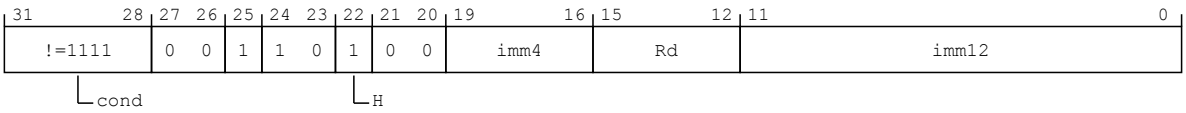

Encoding

```
MOVT{<c>}{<q>}  <Rd>, #<imm16>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm16 : bits(16) = imm4::imm12;
if d == 15 then UnpredictableProcedure(); end;
```

T1

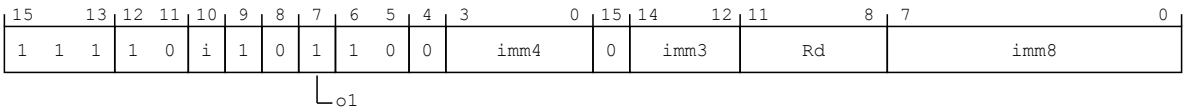

Encoding

```
MOVT{<c>}{<q>}  <Rd>, #<imm16>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm16 : bits(16) = imm4::i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm16>

For the “A1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field.

For the “T1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:i:imm3:imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[31:16] = imm16;
    // R[d]<15:0> unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.116 MRC

Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MRC accesses to system control registers can be trapped to Hyp mode, meaning that an attempt to execute an MRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

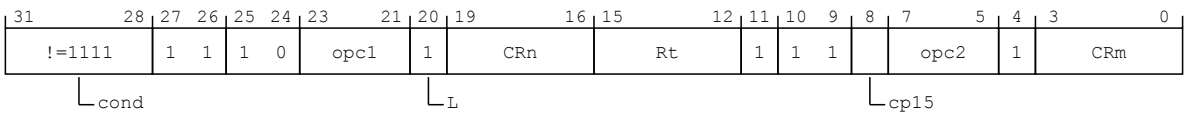

Encoding

```
MRC{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1

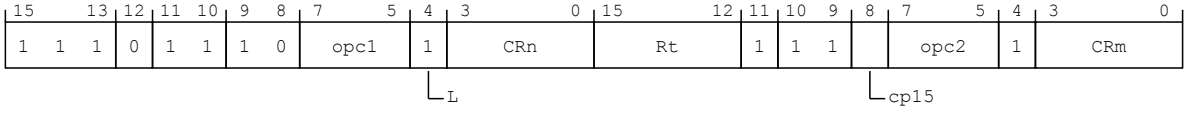

Encoding

```
MRC{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc1” field.

<Rt>

Is the general-purpose register to be transferred or `APSR_nzcv` (encoded as `0b1111`), encoded in the “Rt” field. If `APSR_nzcv` is used, bits [31:28] of the transferred value are written to the PSTATE condition flags.

<CRn>

Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRn” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

<opc2>

Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc2” field.

The possible values of { `<coproc>`, `<opc1>`, `<CRn>`, `<CRm>`, `<opc2>` } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if t != 15 || AArch32_SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        AArch32_SysRegRead(cp, ThisInstr(), t);
    else
        UnpredictableProcedure();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.117 MRRC

Move to two general-purpose registers from System register. This instruction copies the value of a System register to two general-purpose registers.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MRRC accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MRRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MRRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

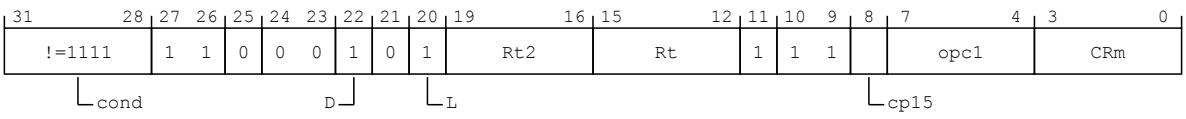

Encoding

```
MRRC{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

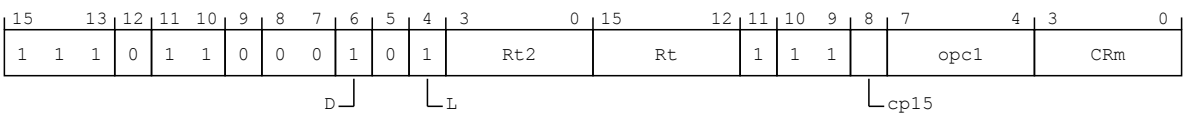

Encoding

```
MRRC{<c>}{<q>}  <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the “opc1” field.

<Rt>

Is the first general-purpose register that is transferred into, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that is transferred into, encoded in the “Rt2” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

The possible values of { `<coproc>`, `<opc1>`, `<CRm>` } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, `<Rt2>` transfers bits[63:32] of the selected System register, while `<Rt>` transfers bits[31:0].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegRead64(cp, ThisInstr(), t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.118 MRS

Move Special register to general-purpose register moves the value of the APSR, CPSR, or SPSR_<current_mode> into a general-purpose register.

Arm recommends the `APSR` form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see APSR.

An `MRS` that accesses the SPSRs is UNPREDICTABLE if executed in User mode or System mode.

An `MRS` that is executed in User mode and accesses the CPSR returns an UNKNOWN value for the CPSR.{E, A, I, F, M} fields.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23	22	21	20	19–16	15–12	11	10	9	8	7	6	4	3–0
!=1111	0	0	0	1	0	R	0	0	(1)(1)(1)(1)	Rd	(0)	(0)	0	(0)	0	0	0	(0)(0)(0)(0)
cond									mask				B		m			Rn

Encoding

```
MRS{<c>}{<q>}  <Rd>, <spec_reg>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
if d == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12	11	10	9–6	5	4	3–0	15	14–12	11–8	7	6	5	4–0
1 1 1	1	0	0	1 1 1 1	1	R	(1)(1)(1)(1)	1	0(0)0	Rd	(0)	(0)	0	(0)(0)(0)(0)(0)

Encoding

```
MRS{<c>}{<q>}  <Rd>, <spec_reg>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<spec_reg>

Is the special register to be accessed, encoded in “R”:

R	<spec_reg>
0	CPSR|APSR
1	SPSR

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        else
            R(d) = SPSR_curr();
        end;
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        let maskval : bits(32) = '11111000 11101111 00000011 11011111';
        var psr_val : bits(32) = GetPSRFromPSTATE{32}(AArch32_NonDebugState) AND maskval;
        if PSTATE.EL == EL0 then
            // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
            // and for the M field, bits<4:0>
            psr_val[22] = ARBITRARY : bits(1);
            psr_val[9:6] = ARBITRARY : bits(4);
            psr_val[4:0] = ARBITRARY : bits(5);
        end;
        R(d) = psr_val;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User, M32_System} && read_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.119 MRS (Banked register)

Move Banked or Special register to general-purpose register

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or Saved Program Status Registers (SPSRs) of the specified mode, or the value of ELR_hyp, to a general-purpose register.

MRS (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see Usage restrictions on the Banked register transfer instructions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23–22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	1 0	R	0	0	M1	Rd		(0)	(0)	1 (B)	M	0	0	0	0	(0) (0) (0) (0) (Rn)

Encoding

```
MRS{<c>}{<q>}  <Rd>, <banked_reg>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
if d == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

T1

15–13	12	11	10	9–6	5	4	3–0	15	14–12	11–8	7	6	5	4	3–0
1 1 1	1	0	0	1 1 1 1	1	R	M1	1	0(0)0	Rd	(0)	(0)	1	M	(0) (0) (0) (0)

Encoding

```
MRS{<c>}{<q>}  <Rd>, <banked_reg>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<banked_reg>

Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R	M	M1	<banked_reg>
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_BankedRegister);
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
            when Constraint_UNKNOWN =>
                R(d) = ARBITRARY : bits(32);
        end;
    else
        let mode : bits(5) = PSTATE.M;
        if read_spsr then
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint_NONE;
            (valid, c)   = SPSRaccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '01110' => R(d) = SPSR_fiq()[31:0];
                    when '10000' => R(d) = SPSR_irq()[31:0];
                    when '10010' => R(d) = SPSR_svc()[31:0];
                    when '10100' => R(d) = SPSR_abt()[31:0];
                    when '10110' => R(d) = SPSR_und()[31:0];
                    when '11100' =>
                        if !ELUsingAArch32(EL3) then AArch64_MonitorModeTrap(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                R(d) = SPSR_mon();
            when '11110' => R(d) = SPSR_hyp()[31:0];
        end;
    else
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
            when Constraint_UNKNOWN =>
                R(d) = ARBITRARY : bits(32);
            when Constraint_ANYREG =>
                R(d) = ReadAnyAllocatedSPSR();
        end;
    end;
else
    // Check for CONSTRAINED UNPREDICTABLE cases
    var valid : boolean;
    var c : Constraint = Constraint_NONE;
    (valid, c)   = BankedRegisterAccessValid(SYSm, mode);

    if valid then
        case SYSm of
            when '00xxx' =>                        // Access the User mode registers
                let m : integer = UInt(SYSm[2:0]) + 8;
                R(d) = Rmode(m,M32_User);
            when '01xxx' =>                        // Access the FIQ mode registers
                let m : integer = UInt(SYSm[2:0]) + 8;
                R(d) = Rmode(m,M32_FIQ);
            when '1000x' =>                        // Access the IRQ mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m,M32_IRQ);
            when '1001x' =>                        // Access the Supervisor mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m,M32_Svc);
            when '1010x' =>                         // Access the Abort mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m,M32_Abort);
            when '1011x' =>                         // Access the Undefined mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m,M32_Undef);
            when '1110x' =>                        // Access Monitor registers
                if !ELUsingAArch32(EL3) then AArch64_MonitorModeTrap(); end;
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m,M32_Monitor);
            when '11110' =>                        // Access ELR_hyp register
                R(d) = ELR_hyp();
            when '11111' =>                        // Access SP_hyp register
                R(d) = Rmode(13,M32_Hyp);
        end;
    else
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
            when Constraint_UNKNOWN =>
                R(d) = ARBITRARY :bits(32);
            when Constraint_ANYREG =>
                R(d) = ReadAnyAllocatedRegister();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.120 MSR (Banked register)

Move general-purpose register to Banked or Special register

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or Saved Program Status Registers (SPSRs) of the specified mode, or to ELR_hyp.

`MSR` (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an `MSR` (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see Usage restrictions on the Banked register transfer instructions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23–22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111 (cond)	0	0	0	1 0	R	1	0	M1	(1)(1)(1)(1) (Rd)	(0)	(0)	1 (B)	M	0	0	0	0	Rn

Encoding

```
MSR{<c>}{<q>}  <banked_reg>, <Rn>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if n == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

T1

15–13	12	11	10	9–6	5	4	3–0	15	14–12	11–8	7	6	5	4	3–0
1 1 1	1	0	0	1 1 1 0	0	R	Rn	1	0(0)0	M1	(0)	(0)	1	M	(0)(0)(0)(0)

Encoding

```
MSR{<c>}{<q>}  <banked_reg>, <Rn>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<banked_reg>

Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R	M	M1	<banked_reg>
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_BankedRegister);
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP, Constraint_UNKNOWN =>
                ExecuteAsNOP();
        end;
    else
        let mode : bits(5) = PSTATE.M;
        if write_spsr then
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint_NONE;
            (valid, c)   = SPSRaccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '01110' => SPSR_fiq()[31:0] = R(n);
                    when '10000' => SPSR_irq()[31:0] = R(n);
                    when '10010' => SPSR_svc()[31:0] = R(n);
                    when '10100' => SPSR_abt()[31:0] = R(n);
                    when '10110' => SPSR_und()[31:0] = R(n);
                    when '11100' =>
                        if !ELUsingAArch32(EL3) then AArch64_MonitorModeTrap(); end;
                        SPSR_mon()[31:0] = R(n);
                    when '11110' => SPSR_hyp()[31:0] = R(n);
                end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                else
                    case c of
                        when Constraint_UNDEF =>
                            Undefined();
                        when Constraint_NOP, Constraint_UNKNOWN =>
                            ExecuteAsNOP();
                        when Constraint_ANYREG =>
                            WriteAnyAllocatedSPSR(R(n));
                    end;
                end;
            else
                // Check for CONSTRAINED UNPREDICTABLE cases
                var valid : boolean;
                var c : Constraint = Constraint_NONE;
                (valid, c)   = BankedRegisterAccessValid(SYSm, mode);

                if valid then
                    case SYSm of
                        when '00xxx' =>                        // Access the User mode registers
                            let m : integer = UInt(SYSm[2:0]) + 8;
                            Rmode(m,M32_User) = R(n);
                        when '01xxx' =>                        // Access the FIQ mode registers
                            let m : integer = UInt(SYSm[2:0]) + 8;
                            Rmode(m,M32_FIQ) = R(n);
                        when '1000x' =>                        // Access the IRQ mode registers
                            // LR when SYSm[0] == 0, otherwise SP
                            let m : integer = 14 - UInt(SYSm[0]);
                            Rmode(m,M32_IRQ) = R(n);
                        when '1001x' =>                        // Access the Supervisor mode registers
                            // LR when SYSm[0] == 0, otherwise SP
                            let m : integer = 14 - UInt(SYSm[0]);
                            Rmode(m,M32_Svc) = R(n);
                        when '1010x' =>                        // Access the Abort mode registers
                            // LR when SYSm[0] == 0, otherwise SP
                            let m : integer = 14 - UInt(SYSm[0]);
                            Rmode(m,M32_Abort) = R(n);
                        when '1011x' =>                        // Access the Undefined mode registers
                            // LR when SYSm[0] == 0, otherwise SP
                            let m : integer = 14 - UInt(SYSm[0]);
                            Rmode(m,M32_Undef) = R(n);
                        when '1110x' =>                        // Access Monitor registers
                            if !ELUsingAArch32(EL3) then AArch64_MonitorModeTrap(); end;
                            // LR when SYSm[0] == 0, otherwise SP
                            let m : integer = 14 - UInt(SYSm[0]);
                            Rmode(m,M32_Monitor) = R(n);
                        when '11110' =>                        // Access ELR_hyp register
                            ELR_hyp() = R(n);
                        when '11111' =>                        // Access SP_hyp register
                            Rmode(13,M32_Hyp) = R(n);
                    end;
                else
                    case c of
                        when Constraint_UNDEF =>
                            Undefined();
                        when Constraint_NOP, Constraint_UNKNOWN =>
                            ExecuteAsNOP();
                        when Constraint_ANYREG =>
                            WriteAnyAllocatedRegister(R(n));
                    end;
                end;
            end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.121 MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the APSR, CPSR, or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of `MSR` is normally only useful at the Application level for writing to `APSR_nzcvq` (`CPSR_f`).

If an `MSR` (immediate) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M.

An `MSR` (immediate) executed in User mode:

- Is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher,

An `MSR` (immediate) executed in System mode is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an `MSR` instruction. Arm deprecates using this to change its value.

31–28	27–26	25	24–23	22	21–20	19–16	15–12	11–0
!=1111 (cond)	0 0	1	1 0	R	1 0	mask	(1)(1)(1)(1)	imm12

Encoding

Applies when `(!(R == 0 && mask == 0000))`

```
MSR{<c>}{<q>}  <spec_reg>, #<imm>
```

Decode for this encoding

```
if mask == '0000' && R == '0' then See("Related encodings"); end;
let imm32 : bits(32) = A32ExpandImm(imm12);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000' && R == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Move Special Register and Hints (immediate).

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<spec_reg>

Is one of:

- `APSR_<bits>`.
- `CPSR_<fields>`.
- `SPSR_<fields>`.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of `nzcvq`, `g`, or `nzcvqg`. These map to the following CPSR_<fields> values:

- `APSR_nzcvq` is the same as `CPSR_f` (mask== '1000').
- `APSR_g` is the same as `CPSR_s` (mask == '0100').
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == '1100').

Arm recommends the `APSR_<bits>` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see The Application Program Status Register, APSR.

<imm>

Is an immediate value. See Modified immediate constants in A32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        else
            SPSRWriteByInstr(imm32, mask);
        end;
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System} && write_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.122 MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the APSR, CPSR or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the `MSR` instruction is being used at Application level and its destination is not `APSR_nzcvq` (`CPSR_f`).

If an `MSR` (register) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M.

An `MSR` (register) executed in User mode:

- Is UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher.

An `MSR` (register) executed in System mode is UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an `MSR` instruction. Arm deprecates using this to change its value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27 26	25–23	22	21	20	19–16	15–12	11 10	9	8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	R	1	0	mask	(1)(1)(1)(1) (Rd)	(0)(0)	0 (B)	(0) (m)	0	0 0 0	Rn

Encoding

```
MSR{<c>}{<q>}  <spec_reg>, <Rn>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
if n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

T1

15–13	12 11	10	9–6	5	4	3–0	15	14–12	11–8	7 6	5	4–0
1 1 1	1 0	0	1 1 1 0	0	R	Rn	1	0(0)0	mask	(0)(0)	0	(0)(0)(0)(0)(0)

Encoding

```
MSR{<c>}{<q>}  <spec_reg>, <Rn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<spec_reg>

Is one of:

- `APSR_<bits>`.
- `CPSR_<fields>`.
- `SPSR_<fields>`.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c mask<0> = ‘1’ to enable writing of bits<7:0> of the destination PSR.
x mask<1> = ‘1’ to enable writing of bits<15:8> of the destination PSR.
s mask<2> = ‘1’ to enable writing of bits<23:16> of the destination PSR.
f mask<3> = ‘1’ to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of `nzcvq`, `g`, or `nzcvqg`. These map to the following CPSR_<fields> values:

- `APSR_nzcvq` is the same as `CPSR_f` (mask== ‘1000’).
- `APSR_g` is the same as `CPSR_s` (mask == ‘0100’).
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == ‘1100’).

Arm recommends the `APSR_<bits>` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see The Application Program Status Register, APSR.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        else
            SPSRWriteByInstr(R(n), mask);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R(n), mask);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `write_spsr && PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.123 MUL, MULS

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

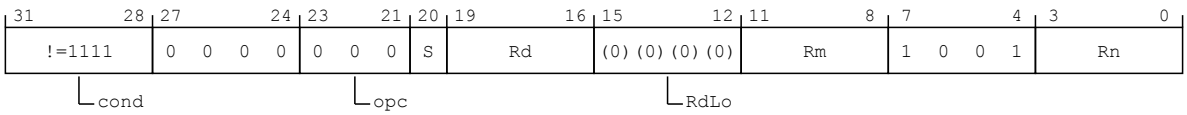

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
MULS{<c>}{<q>}  <Rd>, <Rn>{, <Rm>}
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
MUL{<c>}{<q>}  <Rd>, <Rn>{, <Rm>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

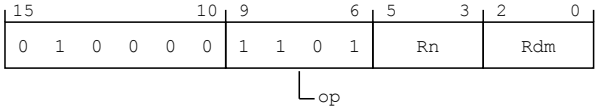

Encoding

```
MUL<c>{<q>}  <Rdm>, <Rn>{, <Rdm>} // (InITBlock())
```

```
MULS{<q>}  <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdm);
let n : integer = UInt(Rn);
let m : integer = UInt(Rdm);
let setflags : boolean = !InITBlock();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15 … 7	6 … 4	3 … 0	15 … 12	11 … 8	7 … 6	5 … 4	3 … 0
1 1 1 1 1 0 1 1 0	0 0 0 (op1)	Rn	1 1 1 1 (Ra)	Rd	0 0	0 0 (op2)	Rm

Encoding

```
MUL{<c>}{<q>}  <Rd>, <Rn>{, <Rm>}
```

```
MUL<c>.W  <Rd>, <Rn>{, <Rm>} // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field. If omitted, `<Rd>` is used.

<Rdm>

Is the second general-purpose source register holding the multiplier and the destination register, encoded in the “Rdm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let result : integer = operand1 * operand2;
    R(d) = result[31:0];
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result[31:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.124 MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

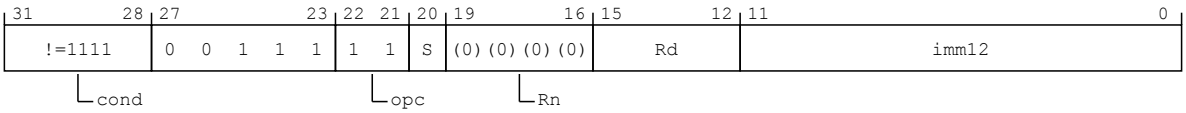

Encoding for the MVN variant

Applies when `(S == 0)`

```
MVN{<c>}{<q>}  <Rd>, #<const>
```

Encoding for the MVNS variant

Applies when `(S == 1)`

```
MVNS{<c>}{<q>}  <Rd>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

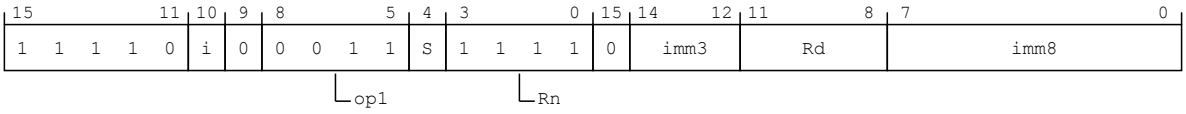

Encoding for the MVN variant

Applies when `(S == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MVN{<c>}{<q>}  <Rd>, #<const>
```

Encoding for the MVNS variant

Applies when `(S == 1)`

```
MVNS{<c>}{<q>}  <Rd>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the "A1 MVN" and "A1 MVNS" variants: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the "T1 MVN" and "T1 MVNS" variants: is the general-purpose destination register, encoded in the "Rd" field.

<const>

For the "A1 MVN" and "A1 MVNS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the "T1 MVN" and "T1 MVNS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.125 MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 1 1	1 1	S	(0)(0)(0)(0)	Rd	imm5	stype	0	Rm
cond		opc		Rn					

Encoding for the MVN, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
MVN{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MVN, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
MVN{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

Encoding for the MVNS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
MVNS{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MVNS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
MVNS{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	1 1 1 1 (op)	Rm	Rd

Encoding

```
MVN<c>{<q>}  <Rd>, <Rm> // (InITBlock())
```

```
MVNS{<q>}  <Rd>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	0 0 1 1 (op1)	S	1 1 1 1 (Rn)	(0)	imm3	Rd	imm2	stype	Rm

Encoding for the MVN, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
MVN{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MVN, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
MVN{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

```
MVN<c>.W  <Rd>, <Rm> // (Inside IT block, and <Rd>, <Rm> can be represented in T1)
```

Encoding for the MVNS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
MVNS{<c>}{<q>}  <Rd>, <Rm>, RRX
```

Encoding for the MVNS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
MVNS{<c>}{<q>}  <Rd>, <Rm> {, <shift> #<amount>}
```

```
MVNS.W  <Rd>, <Rm> // (Outside IT block, and <Rd>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MVN, rotate right with extend”, “A1 MVN, shift or rotate by value”, “A1 MVNS, rotate right with extend”, and “A1 MVNS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1”, “T2 MVN, rotate right with extend”, “T2 MVN, shift or rotate by value”, “T2 MVNS, rotate right with extend”, and “T2 MVNS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MVN, rotate right with extend”, “A1 MVN, shift or rotate by value”, “A1 MVNS, rotate right with extend”, and “A1 MVNS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 MVN, rotate right with extend”, “T2 MVN, shift or rotate by value”, “T2 MVNS, rotate right with extend”, and “T2 MVNS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “A1 MVN, shift or rotate by value” and “A1 MVNS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2 MVN, shift or rotate by value” and “T2 MVNS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.126 MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

31–28	27–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 1	1 1 (opc)	S	(0)(0)(0)(0) (Rn)	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
MVNS{<c>}{<q>}  <Rd>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
MVN{<c>}{<q>}  <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.127 NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–26	25	24–23	22	21–20	19–16	15–12	11–0
!=1111 (cond)	0 0	1	1 0	0 (R)	1 0	0 0 0 0 (imm4)	(1) (1) (1) (1)	(0) (0) (0) (0) 00000000 (imm12)

Encoding

```
NOP{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–12	11–8	7–4	3–0
1 0 1 1	1 1 1 1	0 0 0 0 (hint)	0 0 0 0

Encoding

```
NOP{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 0	(1) (1) (1) (1)	1	0 (0) 0	(0)	0 0 0	0 0 0 0 (hint)	0 0 0 0 (option)

Encoding

```
NOP{<c>}.W
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.128 ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

15–11	10	9	8–5	4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	0 0 1 1 (op1)	S	!=1111 (Rn)	0	imm3	Rd	imm8

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
ORNS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
ORN{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("MVN (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<const>

An immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) OR NOT(imm32);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.129 ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	0 0 1 1 (op1)	S	!=1111 (Rn)	(0)	imm3	Rd	imm2	stype	Rm

Encoding for the ORN, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ORN{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORN, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ORN{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ORNS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ORNS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORNS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ORNS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("MVN (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.130 ORR, ORRS (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

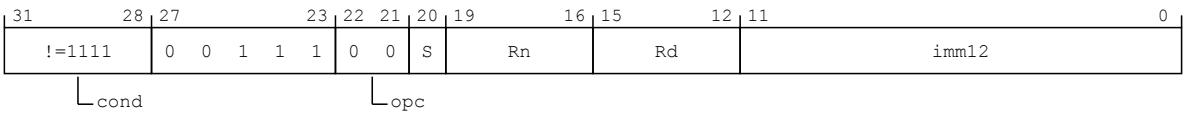

Encoding for the ORR variant

Applies when `(S == 0)`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ORRS variant

Applies when `(S == 1)`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

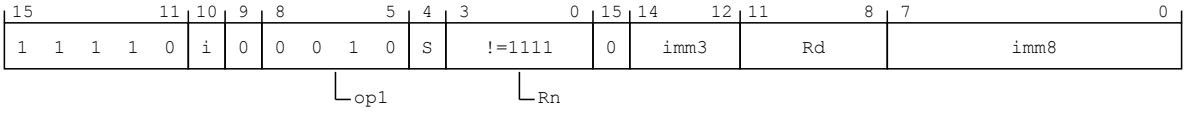

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the ORR variant

Applies when `(S == 0)`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the ORRS variant

Applies when `(S == 1)`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("MOV (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ORR” and “A1 ORRS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 ORR” and “T1 ORRS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ORR” and “A1 ORRS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 ORR” and “T1 ORRS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 ORR” and “A1 ORRS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1 ORR” and “T1 ORRS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) OR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.131 ORR, ORRS (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 1	0 0 (opc)	S	Rn	Rd	imm5	stype	0	Rm

Encoding for the ORR, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORR, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ORRS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORRS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	1 1 0 0	Rm	Rdn
	op		

Encoding

```
ORR<c>{<q>}  {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
ORRS{<q>}  {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

15–9	8–5	4	3–0	15	14–12	11–8	7 6	5 4	3–0
1 1 1 0 1 0 1	0 0 1 0	S	!=1111	(0)	imm3	Rd	imm2	stype	Rm
	op1		Rn						

Encoding for the ORR, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORR, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ORR<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the ORRS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORRS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ORRS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rn == '1111' then See("Related encodings"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Data-processing (shifted register)

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 ORR, rotate right with extend”, “A1 ORR, shift or rotate by value”, “A1 ORRS, rotate right with extend”, and “A1 ORRS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2 ORR, rotate right with extend”, “T2 ORR, shift or rotate by value”, “T2 ORRS, rotate right with extend”, and “T2 ORRS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ORR, rotate right with extend”, “A1 ORR, shift or rotate by value”, “A1 ORRS, rotate right with extend”, and “A1 ORRS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 ORR, rotate right with extend”, “T2 ORR, shift or rotate by value”, “T2 ORRS, rotate right with extend”, and “T2 ORRS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 ORR, rotate right with extend”, “A1 ORR, shift or rotate by value”, “A1 ORRS, rotate right with extend”, and “A1 ORRS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 ORR, rotate right with extend”, “T2 ORR, shift or rotate by value”, “T2 ORRS, rotate right with extend”, and “T2 ORRS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ORR, shift or rotate by value” and “A1 ORRS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2 ORR, shift or rotate by value” and “T2 ORRS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if `ORRS <Rd>, <Rn>, <Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ORRS <Rd>, <Rn>` had been written.
- Inside an IT block, if `ORR<c> <Rd>, <Rn>, <Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ORR<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.132 ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

31–28	27–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 1 1	0 0 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
ORRS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
ORR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.133 PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–20	19–16	15–12	11–7	6	5	4	3–0
!=1111 (cond)	0 1 1 0 1 0 0 0	Rn	Rd	imm5	tb	0	1	Rm

Encoding for the PKHBT variant

Applies when `(tb == 0)`

```
PKHBT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, LSL #<imm>}
```

Encoding for the PKHTB variant

Applies when `(tb == 1)`

```
PKHTB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ASR #<imm>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let tbform : boolean = (tb == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(tb::'0', imm5);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5	4	3–0
1 1 1 0 1 0 1	0 1 1 0 (op1)	0 (S)	Rn	(0)	imm3	Rd	imm2	tb	0 (T)	Rm

Encoding for the PKHBT variant

Applies when `(tb == 0)`

```
PKHBT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, LSL #<imm>} // (tbform == FALSE)
```

Encoding for the PKHTB variant

Applies when `(tb == 1)`

```
PKHTB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ASR #<imm>} // (tbform == TRUE)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if S == '1' || T == '1' then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let tbform : boolean = (tb == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(tb::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 PKHBT” and “A1 PKHTB” variants: the shift to apply to the value read from `<Rm>`, encoded in the “imm5” field.

For `PKHBT`, it is one of:

omitted No shift, encoded as `0b00000`.
1-31 Left shift by specified number of bits, encoded as a binary number.

For `PKHTB`, it is one of:

omitted Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.
1-32 Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For the “T1 PKHBT” and “T1 PKHTB” variants: the shift to apply to the value read from `<Rm>`, encoded in the “imm3:imm2” field.

For `PKHBT`, it is one of:

omitted No shift, encoded as `0b00000`.
1-31 Left shift by specified number of bits, encoded as a binary number.

For `PKHTB`, it is one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

omitted Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.

1-32 Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    R(d)[15:0]  = if tbform then operand2[15:0] else R(n)[15:0];
    R(d)[31:16] = if tbform then R(n)[31:16]    else operand2[31:16];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.134 PLD, PLDW (immediate)

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

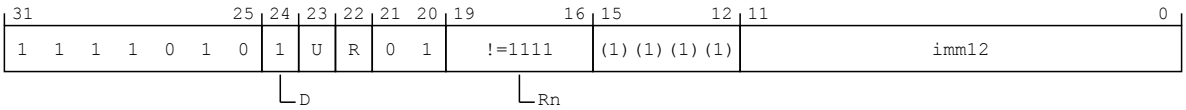

Encoding for the Preload read variant

Applies when `(R == 1)`

```
PLD{<c>}{<q>}  [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Preload write variant

Applies when `(R == 0)`

```
PLDW{<c>}{<q>}  [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let is_pldw : boolean = (R == '0');
```

T1

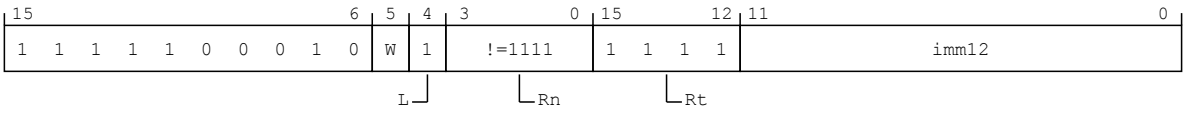

Encoding for the Preload read variant

Applies when `(W == 0)`

```
PLD{<c>}{<q>}  [<Rn> {, #{+}<imm>}]
```

Encoding for the Preload write variant

Applies when `(W == 1)`

```
PLDW{<c>}{<q>}  [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = TRUE;
let is_pldw : boolean = (W == '1');
```

T2

15									6	5	4	3		0	15			12	11	10	9	8	7		0
1	1	1	1	1	0	0	0	0	0	W	1	!=1111			1	1	1	1	1	1	0	0	imm8		
											L	Rn			Rt				P		U				

Encoding for the Preload read variant

Applies when `(W == 0)`

```
PLD{<c>}{<q>}  [<Rn> {, #-<imm>}]
```

Encoding for the Preload write variant

Applies when `(W == 1)`

```
PLDW{<c>}{<q>}  [<Rn> {, #-<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let add : boolean = FALSE;
let is_pldw : boolean = (W == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “A1 Preload read” and “A1 Preload write” variants: see Standard assembler syntax fields. Must be `AL` or omitted.

For the “T1 Preload read”, “T1 Preload write”, “T2 Preload read”, and “T2 Preload write” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If the PC is used, see PLD (literal).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Preload read” and “A1 Preload write” variants: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T1 Preload read” and “T1 Preload write” variants: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2 Preload read” and “T2 Preload write” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.135 PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

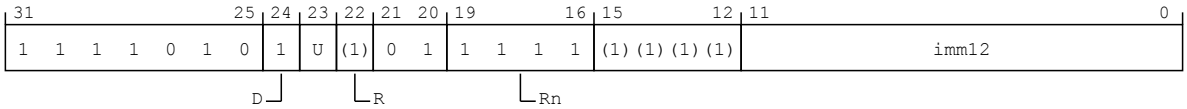

Encoding

```
PLD{<c>}{<q>}  <label> // (Normal form)
```

```
PLD{<c>}{<q>}  [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

T1

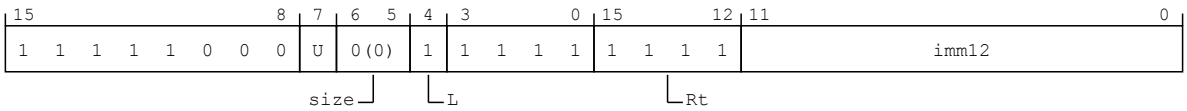

Encoding

```
PLD{<c>}{<q>}  <label> // (Preferred syntax)
```

```
PLD{<c>}{<q>}  [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. Must be `AL` or omitted.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<label>

The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                              else (AlignDownSize(PC32(),4) - imm32));
    Hint_PreloadData(address);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.136 PLD, PLDW (register)

Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
1 1 1 1 0 1 1	1	U	R	0	1	Rn	(1) (1) (1) (1)	imm5	stype	0	Rm
	D										

Encoding for the Preload read, optional shift or rotate variant

Applies when `(R == 1 && !(imm5 == 00000 && stype == 11))`

```
PLD{<c>}{<q>}  [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Encoding for the Preload read, rotate right with extend variant

Applies when `(R == 1 && imm5 == 00000 && stype == 11)`

```
PLD{<c>}{<q>}  [<Rn>, {+/-}<Rm> , RRX]
```

Encoding for the Preload write, optional shift or rotate variant

Applies when `(R == 0 && !(imm5 == 00000 && stype == 11))`

```
PLDW{<c>}{<q>}  [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Encoding for the Preload write, rotate right with extend variant

Applies when `(R == 0 && imm5 == 00000 && stype == 11)`

```
PLDW{<c>}{<q>}  [<Rn>, {+/-}<Rm> , RRX]
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = (U == '1');
let is_pldw : boolean = (R == '0');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 && is_pldw) then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–6	5	4	3–0	15–12	11–6	5–4	3–0
1 1 1 1 1 0 0 0 0 0	W	1	!=1111	1 1 1 1	0 0 0 0 0 0	imm2	Rm
		L	Rn	Rt			

Encoding for the Preload read variant

Applies when `(W == 0)`

```
PLD{<c>}{<q>}  [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Encoding for the Preload write variant

Applies when `(W == 1)`

```
PLDW{<c>}{<q>}  [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = TRUE;
let is_pldw : boolean = (W == '1');
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “Preload read, optional shift or rotate”, “Preload read, rotate right with extend”, “Preload write, optional shift or rotate”, and “Preload write, rotate right with extend” variants: see Standard assembler syntax fields. `<c>` must be `AL` or omitted.

For the “Preload read” and “Preload write” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “Preload read, optional shift or rotate”, “Preload read, rotate right with extend”, “Preload write, optional shift or rotate”, and “Preload write, rotate right with extend” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used.

For the “Preload read” and “Preload write” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “Preload read, optional shift or rotate” and “Preload write, optional shift or rotate” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “Preload read” and “Preload write” variants: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the “imm2” field.

+

Specifies the index register is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let address : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.137 PLI (immediate, literal)

Preload Instruction (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a `PLI` instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

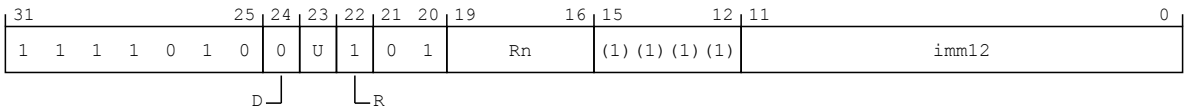

Encoding

```
PLI{<c>}{<q>}  [<Rn> {, #{+/-}<imm>}] // (Preferred syntax)
```

```
PLI{<c>}{<q>}  <label> // (Normal form)
```

```
PLI{<c>}{<q>}  [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

T1

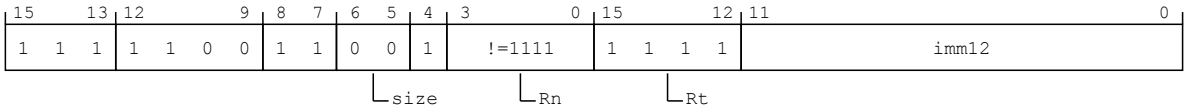

Encoding

```
PLI{<c>}{<q>}  [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("encoding T3"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12–9	8–7	6–5	4	3–0	15–12	11	10	9	8	7–0
1 1 1	1 1 0 0	1 0	0 0	1	!=1111	1 1 1 1	1	1	0	0	imm8
			size		Rn	Rt	P		U	W	

Encoding

```
PLI{<c>}{<q>}  [<Rn> {, #-<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("encoding T3"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let add : boolean = FALSE;
```

T3

15–8	7	6–5	4–0	15–12	11–0
1 1 1 1 1 0 0 1	U	0 0	1 1 1 1 1	1 1 1 1	imm12
		size		Rt	

Encoding

```
PLI{<c>}{<q>}  <label> // (Preferred syntax)
```

```
PLI{<c>}{<q>}  [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
let n : integer = 15;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “A1” variant: see Standard assembler syntax fields. Must be AL or omitted.

For the “T1”, “T2”, and “T3” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T1” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2” variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

For the “T3” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

<label>

The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`.

+

Specifies the offset is added to the base register.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as `0b1111` in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.138 PLI (register)

Preload Instruction (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a `PLI` instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–7	6–5	4	3–0
1 1 1 1 0 1 1	0	U	1	0 1	Rn	(1) (1) (1) (1)	imm5	stype	0	Rm
	D		o2							

Encoding for the Rotate right with extend variant

Applies when `(imm5 == 00000 && stype == 11)`

```
PLI{<c>}{<q>}  [<Rn>, {+/-}<Rm> , RRX]
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm5 == 00000 && stype == 11))`

```
PLI{<c>}{<q>}  [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = (U == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–6	5–4	3–0
1 1 1	1 1 0 0	1 0	0 0	1	!=1111	1 1 1 1	0 0 0 0 0 0	imm2	Rm
			size		Rn	Rt			

Encoding

```
PLI{<c>}{<q>}  [<Rn>, {+}<Rm> {, LSL #<amount>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then See("PLI (immediate, literal)"); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = TRUE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “Rotate right with extend” and “Shift or rotate by value” variants: see Standard assembler syntax fields. `<c>` must be `AL` or omitted.

For the “T1” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<amount>

For the “Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T1” variant: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the “imm2” field.

+

Specifies the index register is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let address : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    Hint_PreloadInstr(address);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.139 POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

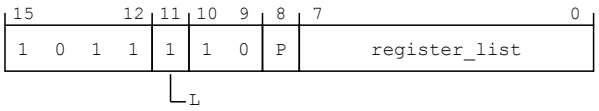

Encoding

```
POP{<c>}{<q>}  <registers> // (Preferred syntax)
```

```
LDM{<c>}{<q>}  SP!, <registers> // (Alternate syntax)
```

Decode for this encoding

```
let registers : bits(16) = P::'0000000'::register_list;
let UnalignedAllowed : boolean = FALSE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<registers>

Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the “register_list” field, and can optionally include the PC. If the PC is in the list, the “P” field is set to 1, otherwise this field defaults to 0.

If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(13);
    for i = 0 to 14 do
        if registers[i] == '1' then
            R(i) = if UnalignedAllowed then MemU{32}(address) else MemA{32}(address);
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        if UnalignedAllowed then
            if address[1:0] == '00' then
                LoadWritePC(MemU{32}(address));
            else
                UnpredictableProcedure();
            end;
        else
            LoadWritePC(MemA{32}(address));
        end;
    end;
    if registers[13] == '0' then R(13) = R(13) + 4*BitCount(registers); end;
    if registers[13] == '1' then R(13) = ARBITRARY : bits(32); end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.140 POP (multiple registers)

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

This is an alias of LDM, LDMIA, LDMFD. This means:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.
- The description of LDM, LDMIA, LDMFD gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2).

A1

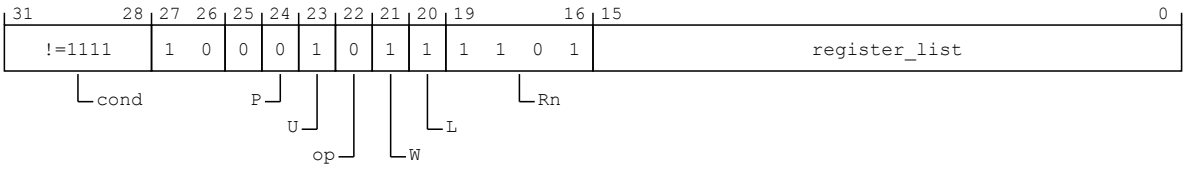

Encoding for the A1 variant

```
POP{<c>}{<q>}  <registers>
```

is equivalent to

```
LDM{IA}{<c>}{<q>}  SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T2

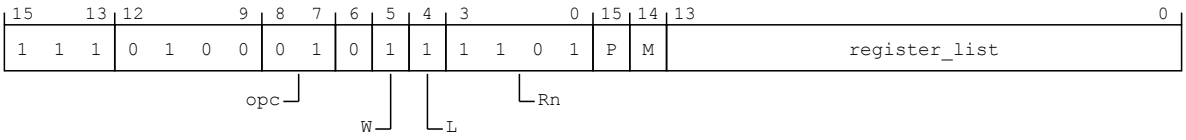

Encoding for the T2 variant

```
POP{<c>}{<q>}  <registers>
```

```
POP{<c>}.W  <registers> // (M == 0 && register_list[13:8] == '000000')
```

is equivalent to

```
LDM{IA}{<c>}.W  SP!, <registers>
```

and is the preferred disassembly when `BitCount(P :: M :: register_list) > 1`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<registers>

For the "A1" variant: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

If the SP is in the list, the value of the SP after such an instruction is UNKNOWN.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

Arm deprecates the use of this instruction with both the LR and the PC in the list.

For the "T2" variant: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

Operational Information

The description of LDM, LDMIA, LDMFD gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.141 POP (single register)

Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data.

This is an alias of LDR (immediate). This means:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T4).

A1

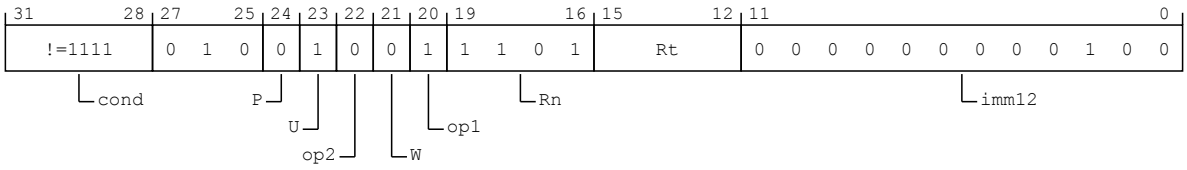

Encoding for the Post-indexed variant

```
POP{<c>}{<q>}  <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>}  <Rt>, [SP], #4
```

and is always the preferred disassembly.

T4

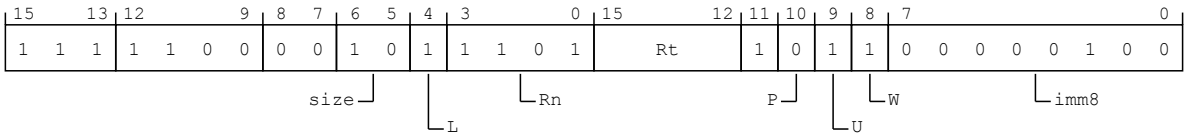

Encoding for the Post-indexed variant

```
POP{<c>}{<q>}  <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>}  <Rt>, [SP], #4
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<single_register_list>

Is the general-purpose register `<Rt>` to be loaded surrounded by { and }.

Operation

The description of LDR (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of LDR (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.142 PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see Physical Speculative Store Bypass Barrier (PSSBB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25–21	20	19–8	7–4	3–0
1 1 1 1	0	1	0 1 0 1 1	1	(1) (1) (1) (1) (1) (1) (1) (1) (0) (0) (0) (0)	0 1 0 0 (opcode)	0 1 0 0 (option)

Encoding

```
PSSBB{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 1	(1) (1) (1) (1)	1	0 (0) 0	(1)	(1) (1) (1)	0 1 0 0 (opc)	0 1 0 0 (option)

Encoding

```
PSSBB{<q>}
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

Operation

```
EncodingSpecificOperations();
SpeculativeStoreBypassBarrierToPA();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.143 PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

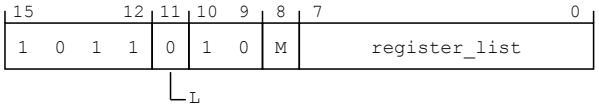

Encoding

```
PUSH{<c>}{<q>}  <registers> // (Preferred syntax)
```

```
STMDB{<c>}{<q>}  SP!, <registers> // (Alternate syntax)
```

Decode for this encoding

```
let registers : bits(16) = '0'::M::'000000'::register_list;
let UnalignedAllowed : boolean = FALSE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<registers>

Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the “register_list” field, and can optionally include the LR. If the LR is in the list, the “M” field is set to 1, otherwise this field defaults to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(13) - 4*BitCount(registers);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA{32}(address) = ARBITRARY : bits(32);
            else
                if UnalignedAllowed then
                    MemU{32}(address) = R(i);
                else
                    MemA{32}(address) = R(i);
                end;
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then  // Only possible for encoding A1 or A2
        if UnalignedAllowed then
            MemU{32}(address) = PCStoreValue();
        else
            MemA{32}(address) = PCStoreValue();
        end;
    end;
    R(13) = R(13) - 4*BitCount(registers);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.144 PUSH (multiple registers)

Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of STMDB, STMFD. This means:

- The encodings in this description are named to match the encodings of STMDB, STMFD.
- The description of STMDB, STMFD gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

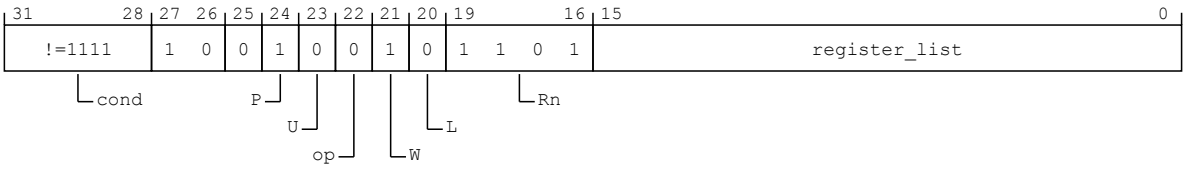

Encoding for the A1 variant

```
PUSH{<c>}{<q>}  <registers>
```

is equivalent to

```
STMDB{<c>}{<q>}  SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T1

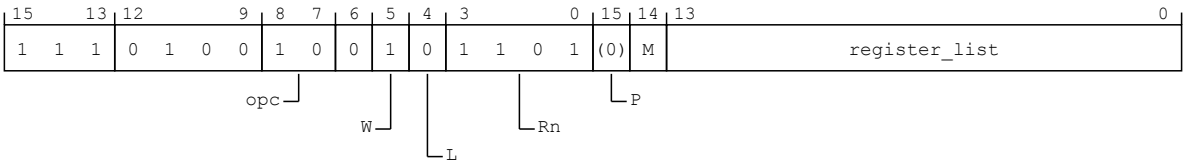

Encoding for the T1 variant

```
PUSH{<c>}{<q>}  <registers>
```

```
PUSH{<c>}.W  <registers> // (register_list[13:8] == '000000')
```

is equivalent to

```
STMDB{<c>}{<q>}  SP!, <registers>
```

and is the preferred disassembly when `BitCount(M :: register_list) > 1`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<registers>

For the "A1" variant: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

The SP and PC can be in the list. However:

- Arm deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

For the "T1" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation

The description of STMDB, STMFD gives the operational pseudocode for this instruction.

Operational Information

The description of STMDB, STMFD gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.145 PUSH (single register)

Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of STR (immediate). This means:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T4).

A1

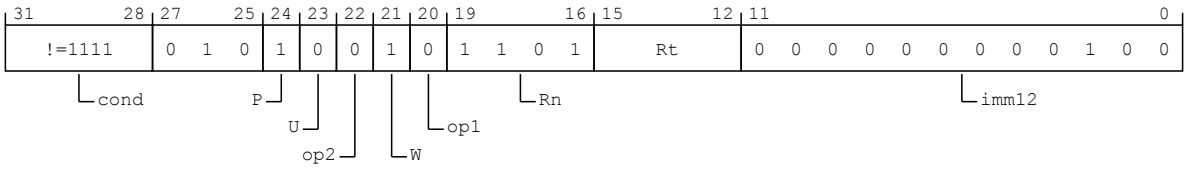

Encoding for the Pre-indexed variant

```
PUSH{<c>}{<q>}  <single_register_list>
```

is equivalent to

```
STR{<c>}{<q>}  <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

T4

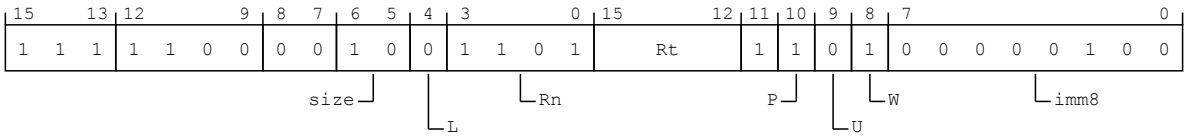

Encoding for the Pre-indexed variant

```
PUSH{<c>}{<q>}  <single_register_list> // (Standard syntax)
```

is equivalent to

```
STR{<c>}{<q>}  <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<single_register_list>

Is the general-purpose register `<Rt>` to be stored surrounded by { and }.

Operation

The description of STR (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of STR (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.146 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–23	22–21	20	19–16	15–12	11–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	0 0 (opc)	0	Rn	Rd	(0)(0)(0)(0)	0	1 0 1	Rm

Encoding

```
QADD{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1	1 1 0 1 0	1	0 0 0 (op1)	Rn	1 1 1 1	Rd	1 0	0 0 (op2)	Rm

Encoding

```
QADD{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (r, sat) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) + SInt(R(n)));
    R(d) = r;
    if sat then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.147 QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

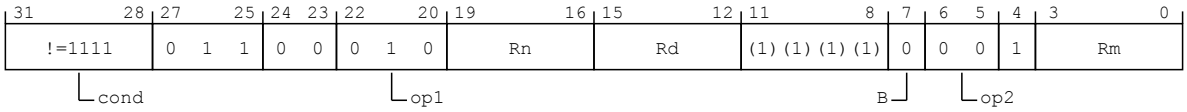

Encoding

```
QADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

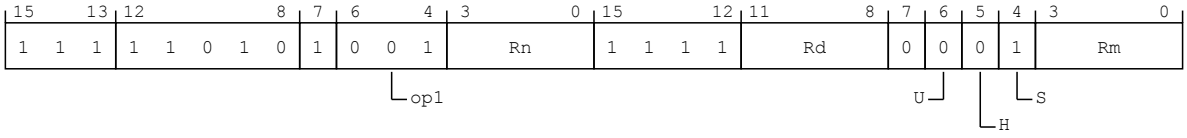

Encoding

```
QADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0]  = SignedSat{16}(sum1);
    R(d)[31:16] = SignedSat{16}(sum2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.148 QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

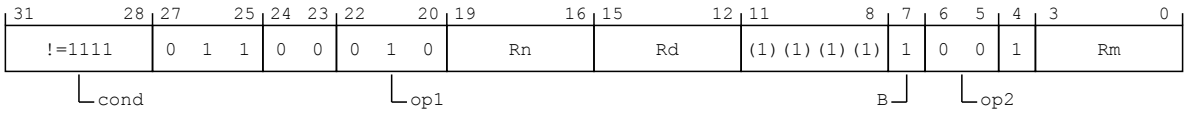

Encoding

```
QADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

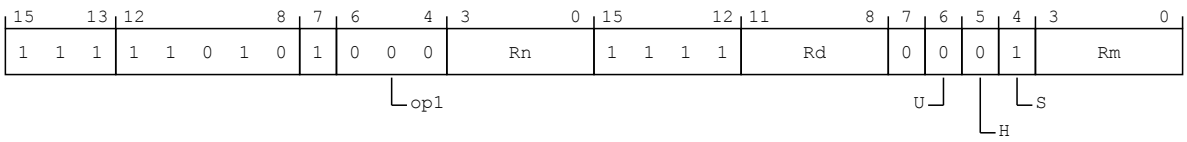

Encoding

```
QADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0]   = SignedSat{8}(sum1);
    R(d)[15:8]  = SignedSat{8}(sum2);
    R(d)[23:16] = SignedSat{8}(sum3);
    R(d)[31:24] = SignedSat{8}(sum4);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.149 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

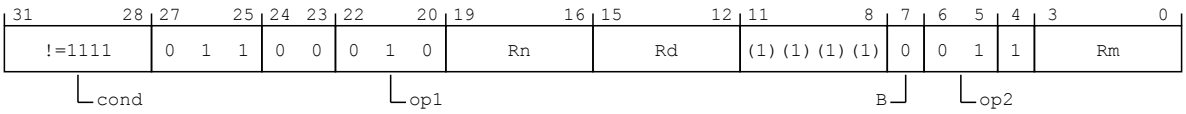

Encoding

```
QASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

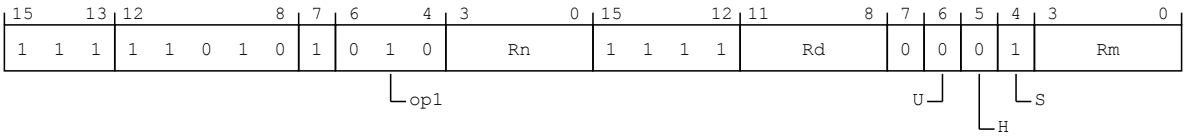

Encoding

```
QASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer  = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0]  = SignedSat{16}(diff);
    R(d)[31:16] = SignedSat{16}(sum);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.150 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets PSTATE.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–23	22–21	20	19–16	15–12	11–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	1 0 (opc)	0	Rn	Rd	(0)(0)(0)(0)	0	1 0 1	Rm

Encoding

```
QDADD{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1	1 1 0 1 0	1	0 0 0 (op1)	Rn	1 1 1 1	Rd	1 0	0 1 (op2)	Rm

Encoding

```
QDADD{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (doubled, sat1) : (bits(32), boolean) = SignedSatQ{32}(2 * SInt(R(n)));
    let (r, sat2) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) + SInt(doubled));
    R(d) = r;
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.151 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets PSTATE.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–23	22–21	20	19–16	15–12	11–8	7	6–4	3–0
!=1111 (cond)	0 0	0 1 0	1 1 (opc)	0	Rn	Rd	(0)(0)(0)(0)	0	1 0 1	Rm

Encoding

```
QDSUB{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1	1 1 0 1 0	1	0 0 0 (op1)	Rn	1 1 1 1	Rd	1 0	1 1 (op2)	Rm

Encoding

```
QDSUB{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (doubled, sat1) : (bits(32), boolean) = SignedSatQ{32}(2 * SInt(R(n)));
    let (r, sat2) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) - SInt(doubled));
    R(d) = r;
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.152 QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

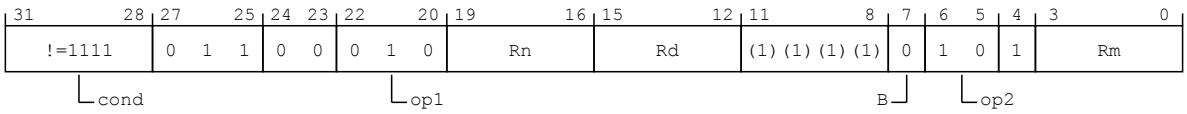

Encoding

```
QSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

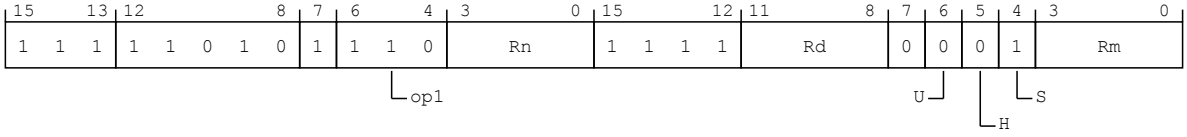

Encoding

```
QSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer  = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0]  = SignedSat{16}(sum);
    R(d)[31:16] = SignedSat{16}(diff);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.153 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23	22	21	20	19–16	15–12	11–8	7	6–4	3–0
!=1111 (cond)	0	0	0	0	0	1 (opc)	0	Rn	Rd	(0)(0)(0)(0)	0	1 0 1	Rm

Encoding

```
QSUB{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1	1 1 0 1 0	1	0 0 0 (op1)	Rn	1 1 1 1	Rd	1 0	1 0 (op2)	Rm

Encoding

```
QSUB{<c>}{<q>}  {<Rd>, }<Rm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (r, sat) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) - SInt(R(n)));
    R(d) = r;
    if sat then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.154 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

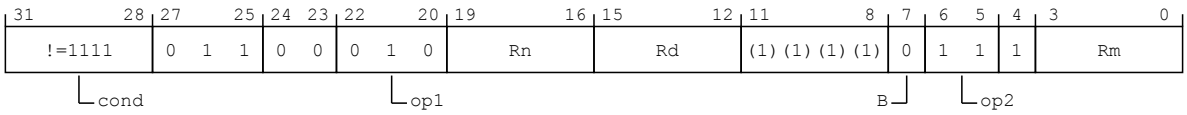

Encoding

```
QSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

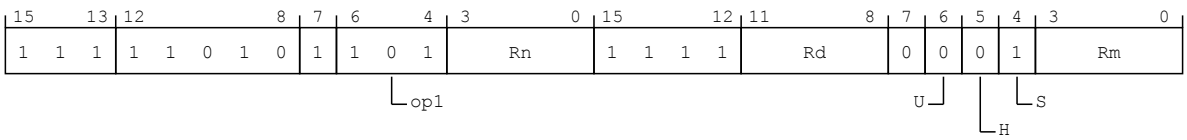

Encoding

```
QSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0]  = SignedSat{16}(diff1);
    R(d)[31:16] = SignedSat{16}(diff2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.155 QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

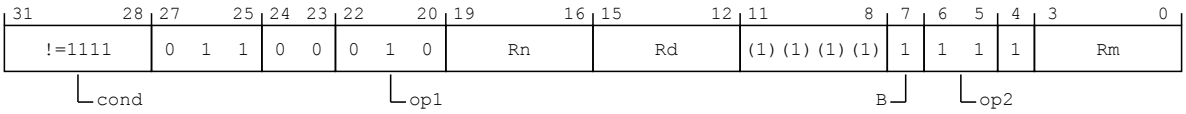

Encoding

```
QSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

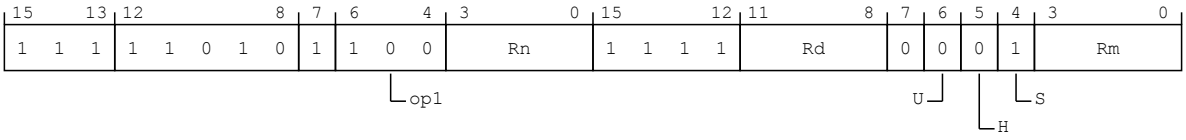

Encoding

```
QSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0]   = SignedSat{8}(diff1);
    R(d)[15:8]  = SignedSat{8}(diff2);
    R(d)[23:16] = SignedSat{8}(diff3);
    R(d)[31:24] = SignedSat{8}(diff4);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.156 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22	21–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 1 1	0 1	1 (o1)	11(1)(1)(1)(1)	Rd	(1)(1)(1)(1)	0 (o2)	0 1	1	Rm

Encoding

```
RBIT{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	1 1 0 1 0	1	0 0 1 (op1)	Rn	1 1 1 1	Rd	1 0	1 0 (op2)	Rm

Encoding

```
RBIT{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” variant: is the general-purpose source register, encoded in the “Rm” field.

For the “T1” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    for i = 0 to 31 do
        result[31-i] = R(m)[i];
    end;
    R(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.157 REV16

Byte-Reverse Packed Halfword reverses the byte order in each16-bit halfword of a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24–23	22	21–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 1 1	0 1	0 (o1)	11(1)(1)(1)(1)	Rd	(1)(1)(1)(1)	1 (o2)	0 1	1	Rm

Encoding

```
REV16{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–12	11–8	7–6	5–3	2–0
1 0 1 1	1 0 1 0	0 1 (op)	Rm	Rd

Encoding

```
REV16{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2

15–7	6–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 1 1 0 1 0 1	0 0 1 (op1)	Rn	1 1 1 1	Rd	1 0	0 1 (op2)	Rm

Encoding

```
REV16{<c>}{<q>}  <Rd>, <Rm>
```

```
REV16{<c>}.W  <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” and “T1” variants: is the general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:24] = R(m)[23:16];
    result[23:16] = R(m)[31:24];
    result[15:8]  = R(m)[7:0];
    result[7:0]   = R(m)[15:8];
    R(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.158 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24–23	22	21–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 1 1	0 1	0	11(1)(1)(1)(1)	Rd	(1)(1)(1)(1)	0	0 1	1	Rm
cond			o1				o2			

Encoding

```
REV{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–12	11–8	7–6	5–3	2–0
1 0 1 1	1 0 1 0	0 0	Rm	Rd
		op		

Encoding

```
REV{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2

15–7	6–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 1 1 0 1 0 1	0 0 1	Rn	1 1 1 1	Rd	1 0	0 0	Rm
	op1					op2	

Encoding

```
REV{<c>}{<q>}  <Rd>, <Rm>
```

```
REV{<c>}.W  <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” and “T1” variants: is the general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:24] = R(m)[7:0];
    result[23:16] = R(m)[15:8];
    result[15:8]  = R(m)[23:16];
    result[7:0]   = R(m)[31:24];
    R(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.159 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

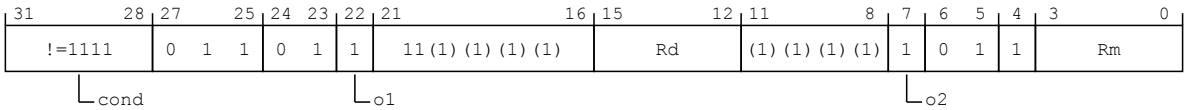

Encoding

```
REVSH{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

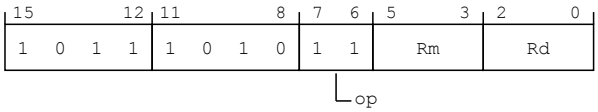

Encoding

```
REVSH{<c>}{<q>}  <Rd>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2

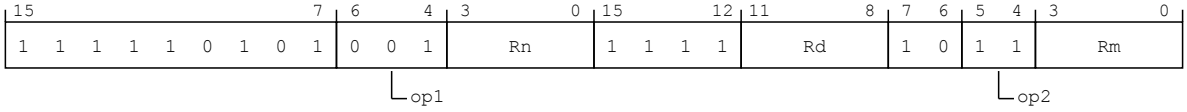

Encoding

```
REVSH{<c>}{<q>}  <Rd>, <Rm>
```

```
REVSH{<c>}.W  <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” and “T1” variants: is the general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:8]  = SignExtend{24}(R(m)[7:0]);
    result[7:0]   = R(m)[15:8];
    R(d) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.160 RFE, RFEDA, RFEDB, RFEIA, RFEIB

Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore PSTATE. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See Illegal return events from AArch32 state.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

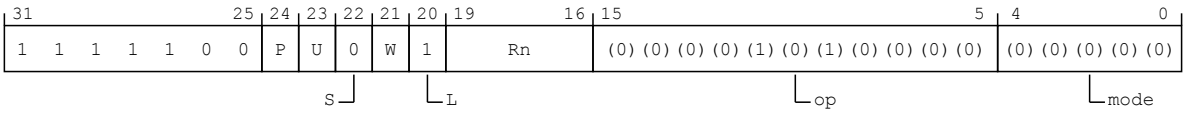

Encoding for the Decrement After variant

Applies when `(P == 0 && U == 0)`

```
RFEDA{<c>}{<q>}  <Rn>{!} // (Preferred syntax)
```

```
RFEFA{<c>}{<q>}  <Rn>{!} // (Alternate syntax, Full Ascending stack)
```

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0)`

```
RFEDB{<c>}{<q>}  <Rn>{!} // (Preferred syntax)
```

```
RFEEA{<c>}{<q>}  <Rn>{!} // (Alternate syntax, Empty Ascending stack)
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
RFE{IA}{<c>}{<q>}  <Rn>{!} // (Preferred syntax)
```

```
RFEFD{<c>}{<q>}  <Rn>{!} // (Alternate syntax, Full Descending stack)
```

Encoding for the Increment Before variant

Applies when `(P == 1 && U == 1)`

```
RFEIB{<c>}{<q>}  <Rn>{!} // (Preferred syntax)
```

```
RFEED{<c>}{<q>}  <Rn>{!} // (Alternate syntax, Empty Descending stack)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 then UnpredictableProcedure(); end;
```

T1

15–9	8	7	6	5	4	3–0	15	14	13–0
1 1 1 0 1 0 0	0	0	0	W	1	Rn	(1)	(1)	(0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0)
	opc				L		P	M	register_list

Encoding

```
RFEDB{<c>}{<q>}  <Rn>{!} // (Outside or last in IT block, preferred syntax)
```

```
RFEFA{<c>}{<q>}  <Rn>{!} // (Outside or last in IT block, alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = FALSE;
let wordhigher : boolean = FALSE;
if n == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15–9	8	7	6	5	4	3–0	15	14	13–0
1 1 1 0 1 0 0	1	1	0	W	1	Rn	(1)	(1)	(0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0)
	opc				L		P	M	register_list

Encoding

```
RFE{IA}{<c>}{<q>}  <Rn>{!} // (Outside or last in IT block, preferred syntax)
```

```
RFEFD{<c>}{<q>}  <Rn>{!} // (Outside or last in IT block, alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = TRUE;
let wordhigher : boolean = FALSE;
if n == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “Decrement After”, “Decrement Before”, “Increment After”, and “Increment Before” variants: see Standard assembler syntax fields. `<c>` must be `AL` or omitted.

For the “T1” and “T2” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

IA

For the “Increment After” variant: is an optional suffix to indicate the Increment After variant.

For the “T2” variant: is an optional suffix for the Increment After form.

`RFEFA`, `RFEEA`, `RFEFD`, and `RFEED` are pseudo-instructions for `RFEDA`, `RFEDB`, `RFEIA`, and `RFEIB` respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.EL == EL0 then
        UnpredictableProcedure();                        // UNDEFINED or NOP
    else
        var address : bits(32) = if increment then R(n) else R(n)-8;
        if wordhigher then address = address+4; end;
        let new_pc_value : bits(32) = MemA{32}(address);
        let spsr : bits(32) = MemA{32}(address+4);
        if wback then R(n) = if increment then R(n)+8 else R(n)-8; end;
        AArch32_ExceptionReturn(new_pc_value, spsr);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.161 ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

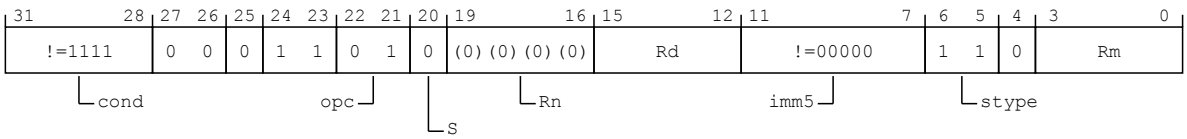

Encoding for the MOV, shift or rotate by value variant

```
ROR{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

T3

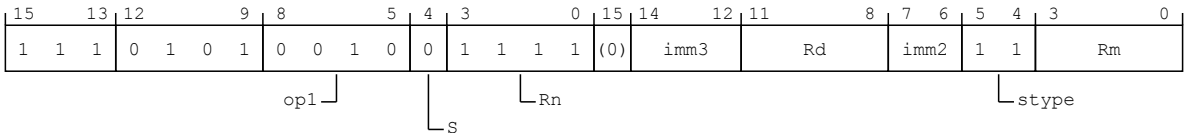

Encoding for the MOV, shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00))`

```
ROR{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the "A1 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the "T3 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field.

<Rm>

For the "A1 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T3 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field.

<imm>

For the "A1 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For the "T3 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.162 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

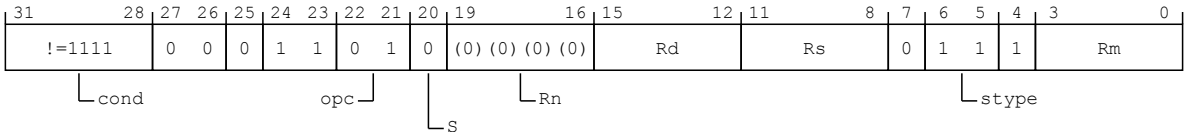

Encoding for the Not flag setting variant

```
ROR{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1

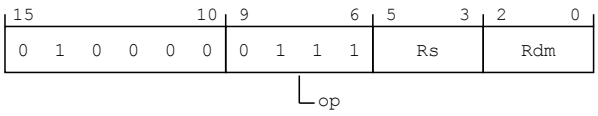

Encoding for the Rotate right variant

```
ROR<c>{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOV<c>{<q>}  <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12–8	7	6–5	4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1 0	0	1 1	0	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype	S					

Encoding for the Not flag setting variant

```
ROR{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
ROR<c>.W  {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.163 RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

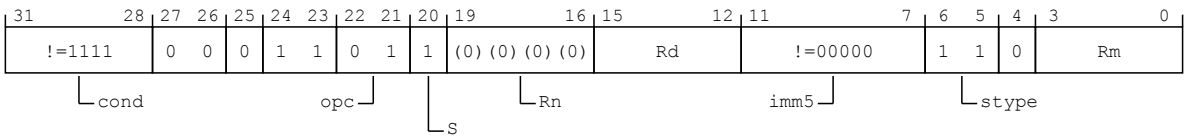

Encoding for the MOVS, shift or rotate by value variant

```
RORS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

T3

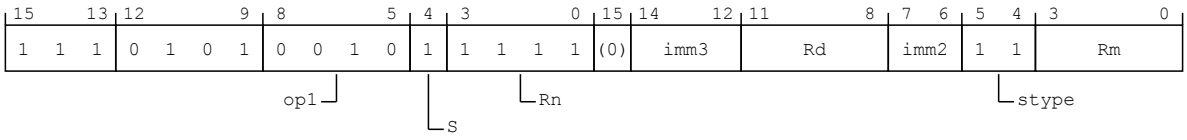

Encoding for the MOVS, shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00))`

```
RORS{<c>}{<q>}  {<Rd>, }<Rm>, #<imm>
```

is equivalent to

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T3 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 31, encoded in the “imm3:imm2” field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.164 RORS (register)

Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

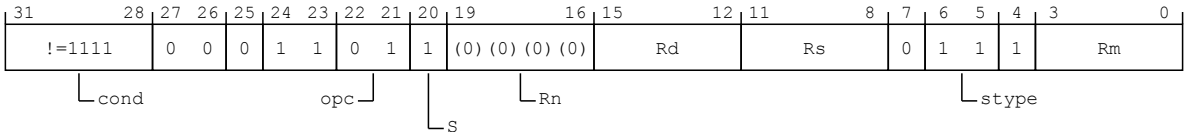

Encoding for the Flag setting variant

```
RORS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1

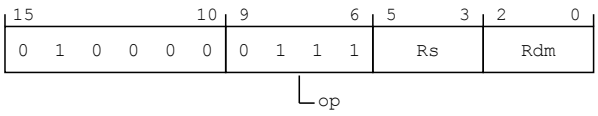

Encoding for the Rotate right variant

```
RORS{<q>}  {<Rdm>, }<Rdm>, <Rs>
```

is equivalent to

```
MOVS{<q>}  <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12–8	7	6–5	4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1 0	0	1 1	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
			stype	S					

Encoding for the Flag setting variant

```
RORS{<c>}{<q>}  {<Rd>, }<Rm>, <Rs>
```

```
RORS.W  {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register-shifted register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.165 RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

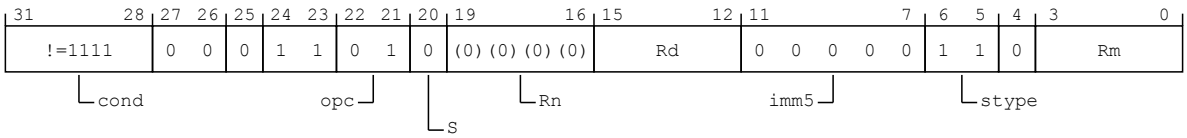

Encoding for the MOV, rotate right with extend variant

```
RRX{<c>}{<q>}  {<Rd>, }<Rm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

T3

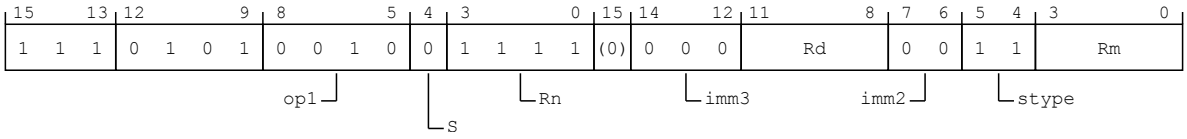

Encoding for the MOV, rotate right with extend variant

```
RRX{<c>}{<q>}  {<Rd>, }<Rm>
```

is equivalent to

```
MOV{<c>}{<q>}  <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rd>

For the “A1 MOV, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T3 MOV, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOV, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.166 RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

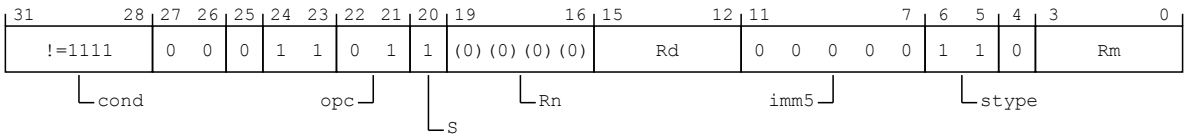

Encoding for the MOVS, rotate right with extend variant

```
RRXS{<c>}{<q>}  {<Rd>, }<Rm>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

T3

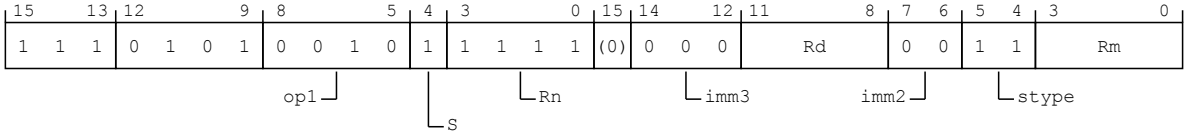

Encoding for the MOVS, rotate right with extend variant

```
RRXS{<c>}{<q>}  {<Rd>, }<Rm>
```

is equivalent to

```
MOVS{<c>}{<q>}  <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 MOVS, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T3 MOVS, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOVS, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Operational Information

The description of MOV, MOVS (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.167 RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

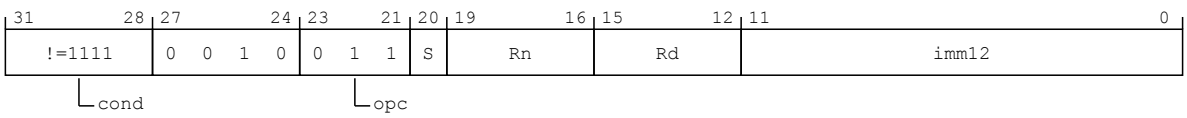

Encoding for the RSB variant

Applies when `(S == 0)`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the RSBS variant

Applies when `(S == 1)`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

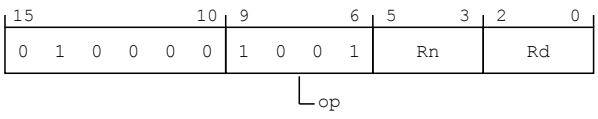

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
RSB<c>{<q>}  {<Rd>, }<Rn>, #0 // (InITBlock())
```

```
RSBS{<q>}  {<Rd>, }<Rn>, #0 // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = Zeros{}; // immediate = #0
```

T2

15				11	10	9	8			5	4	3	0	15	14	12	11	8	7	0
1	1	1	1	0	i	0	1	1	1	0	S	Rn		0	imm3		Rd		imm8	

op1

Encoding for the RSB variant

Applies when `(S == 0)`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

```
RSB<c>.W  {<Rd>, }<Rn>, #0 // (Inside IT block)
```

Encoding for the RSBS variant

Applies when `(S == 1)`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

```
RSBS.W  {<Rd>, }<Rn>, #0 // (Outside IT block)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rd>

For the “A1 RSB” and “A1 RSBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1”, “T2 RSB”, and “T2 RSBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 RSB” and “A1 RSBS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 RSB”, and “T2 RSBS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 RSB” and “A1 RSBS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T2 RSB” and “T2 RSBS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), imm32, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.168 RSB, RSBS (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	0 1 1	S	Rn	Rd	imm5	stype	0	Rm
cond		opc							

Encoding for the RSB, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the RSB, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the RSBS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the RSBS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	1 1 1 0 (op1)	S	Rn	(0)	imm3	Rd	imm2	stype	Rm

Encoding for the RSB, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the RSB, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the RSBS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the RSBS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 RSB, rotate right with extend”, “A1 RSB, shift or rotate by value”, “A1 RSBS, rotate right with extend”, and “A1 RSBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 RSB, rotate right with extend”, “T1 RSB, shift or rotate by value”, “T1 RSBS, rotate right with extend”, and “T1 RSBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 RSB, rotate right with extend”, “A1 RSB, shift or rotate by value”, “A1 RSBS, rotate right with extend”, and “A1 RSBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 RSB, rotate right with extend”, “T1 RSB, shift or rotate by value”, “T1 RSBS, rotate right with extend”, and “T1 RSBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 RSB, rotate right with extend”, “A1 RSB, shift or rotate by value”, “A1 RSBS, rotate right with extend”, and “A1 RSBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1 RSB, rotate right with extend”, “T1 RSB, shift or rotate by value”, “T1 RSBS, rotate right with extend”, and “T1 RSBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 RSB, shift or rotate by value” and “A1 RSBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T1 RSB, shift or rotate by value” and “T1 RSBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.169 RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	0 1 1 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
RSBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
RSB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, '1');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.170 RSC, RSCS (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

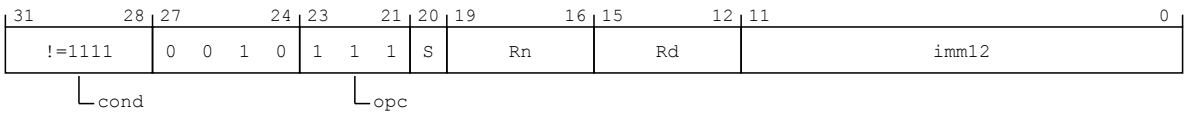

Encoding for the RSC variant

Applies when `(S == 0)`

```
RSC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the RSCS variant

Applies when `(S == 1)`

```
RSCS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

<const>

An immediate value. See Modified immediate constants in A32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.171 RSC, RSCS (register)

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	1 1 1	S	Rn	Rd	imm5	stype	0	Rm
cond		opc							

Encoding for the RSC, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
RSC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the RSC, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
RSC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the RSCS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
RSCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the RSCS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
RSCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

<Rn>

Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

<Rm>

Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

<shift>

Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.172 RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	1 1 1 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
RSCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
RSC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.173 SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

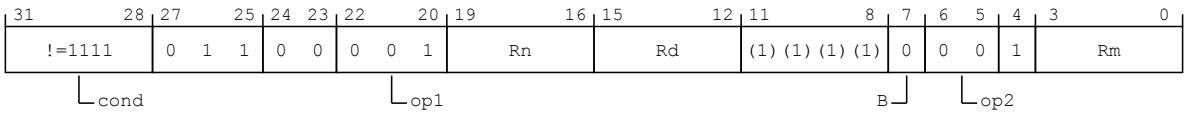

Encoding

```
SADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

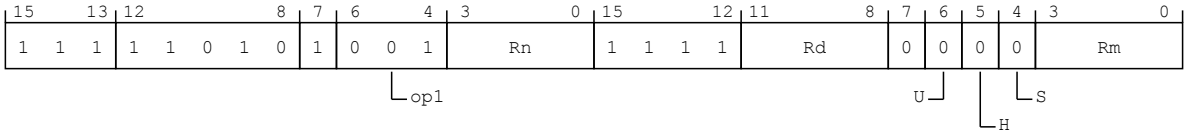

Encoding

```
SADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0]  = sum1[15:0];
    R(d)[31:16] = sum2[15:0];
    PSTATE.GE[1:0] = if sum1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum2 >= 0 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.174 SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

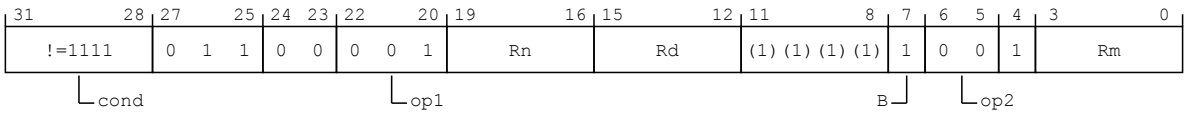

Encoding

```
SADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

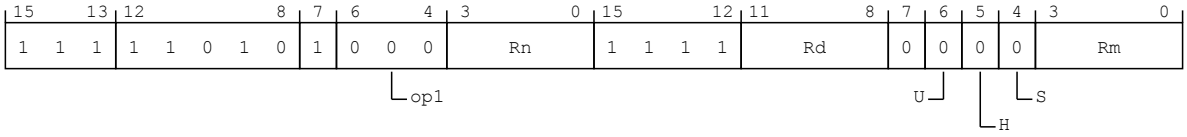

Encoding

```
SADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0]   = sum1[7:0];
    R(d)[15:8]  = sum2[7:0];
    R(d)[23:16] = sum3[7:0];
    R(d)[31:24] = sum4[7:0];
    PSTATE.GE[0]  = if sum1 >= 0 then '1' else '0';
    PSTATE.GE[1]  = if sum2 >= 0 then '1' else '0';
    PSTATE.GE[2]  = if sum3 >= 0 then '1' else '0';
    PSTATE.GE[3]  = if sum4 >= 0 then '1' else '0';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.175 SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

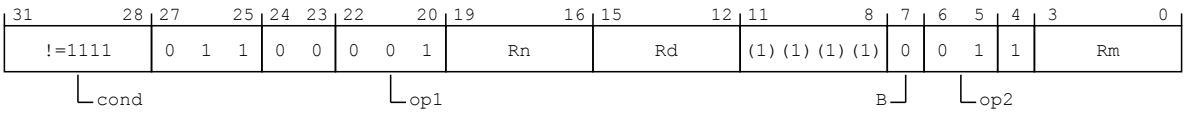

Encoding

```
SASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

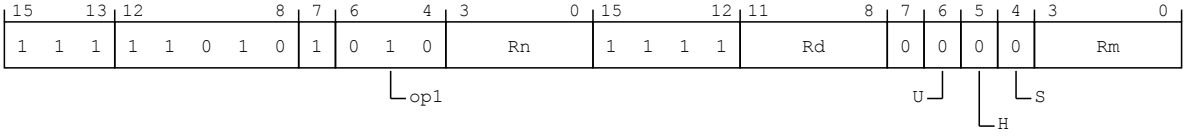

Encoding

```
SASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer  = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0]  = diff[15:0];
    R(d)[31:16] = sum[15:0];
    PSTATE.GE[1:0] = if diff >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum  >= 0 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.176 SB

Speculation Barrier is a barrier that controls speculation. For more information and details of the semantics, see Speculation Barrier (SB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SB)

31–28	27	26	25–21	20	19–8	7–4	3–0
1 1 1 1	0	1	0 1 0 1 1	1	(1) (1) (1) (1) (1) (1) (1) (1) (0) (0) (0) (0)	0 1 1 1 (opcode)	(0) (0) (0) (0) (option)

Encoding

```
SB{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

(FEAT_SB)

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 1	(1) (1) (1) (1)	1	0 (0) 0	(1)	(1) (1) (1)	0 1 1 1 (opc)	(0) (0) (0) (0) (option)

Encoding

```
SB{<q>}
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculationBarrier();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.177 SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

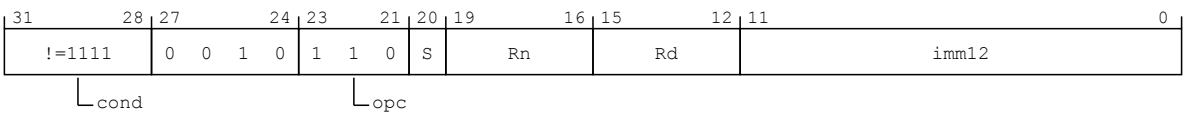

Encoding for the SBC variant

Applies when `(S == 0)`

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the SBCS variant

Applies when `(S == 1)`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

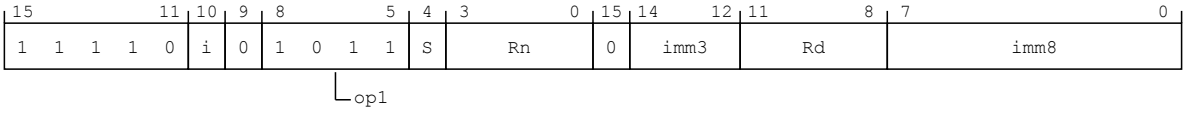

Encoding for the SBC variant

Applies when `(S == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the SBCS variant

Applies when `(S == 1)`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 SBC” and “A1 SBCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 SBC” and “T1 SBCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 SBC” and “A1 SBCS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 SBC” and “T1 SBCS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 SBC” and “A1 SBCS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1 SBC” and “T1 SBCS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.178 SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	1 1 0	S	Rn	Rd	imm5	stype	0	Rm
cond		opc							

Encoding for the SBC, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SBC, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the SBCS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SBCS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

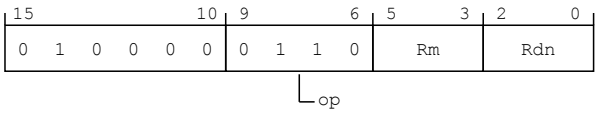

Encoding

```
SBC<c>{<q>}  {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
SBCS{<q>}  {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

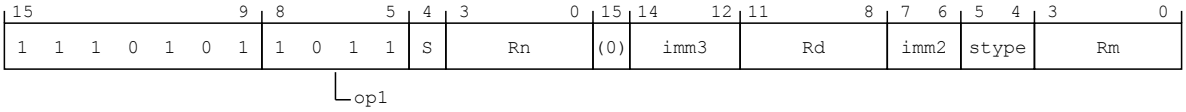

Encoding for the SBC, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SBC, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
SBC<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the SBCS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SBCS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
SBCS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 SBC, rotate right with extend”, “A1 SBC, shift or rotate by value”, “A1 SBCS, rotate right with extend”, and “A1 SBCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T2 SBC, rotate right with extend”, “T2 SBC, shift or rotate by value”, “T2 SBCS, rotate right with extend”, and “T2 SBCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 SBC, rotate right with extend”, “A1 SBC, shift or rotate by value”, “A1 SBCS, rotate right with extend”, and “A1 SBCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 SBC, rotate right with extend”, “T2 SBC, shift or rotate by value”, “T2 SBCS, rotate right with extend”, and “T2 SBCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 SBC, rotate right with extend”, “A1 SBC, shift or rotate by value”, “A1 SBCS, rotate right with extend”, and “A1 SBCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 SBC, rotate right with extend”, “T2 SBC, shift or rotate by value”, “T2 SBCS, rotate right with extend”, and “T2 SBCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 SBC, shift or rotate by value” and “A1 SBCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 SBC, shift or rotate by value” and “T2 SBCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.179 SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	1 1 0 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
SBCS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
SBC{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.180 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22	21	20–16	15–12	11–7	6–5	4	3–0
!=1111	0 1 1	1 1	0	1	widthm1	Rd	lsb	1 0	1	Rn
cond			U							

Encoding

```
SBFX{<c>}{<q>}  <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(lsb);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15–13	12–11	10	9	8	7–5	4	3–0	15	14–12	11–8	7–6	5	4–0
1 1 1	1 0	(0)	1	1	0 1 0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1
					op1								

Encoding

```
SBFX{<c>}{<q>}  <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(imm3::imm2);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “A1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the width of the field, in the range 1 to 32-<lsb>, encoded in the “widthm1” field as <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d) = SignExtend{32}(R(n)[msbit:lsbit]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.181 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22–20	19–16	15–12	11–8	7–5	4	3–0
!=1111	0 1 1	1 0	0 0 1	Rd	(1) (1) (1) (1)	Rm	0 0 0	1	Rn
cond			op1		Ra		op2		

Encoding

```
SDIV{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

T1

15–13	12–9	8–7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1 1	0 0 1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm
			op1		Ra		op2	

Encoding

```
SDIV{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the dividend, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the divisor, encoded in the “Rm” field.

Overflow

If the signed integer division `0x80000000` / `0xFFFFFFFF` is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to `<Rd>` must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is `0x80000000`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let dividend : integer = SInt(R(n));
    let divisor : integer  = SInt(R(m));
    var result : integer;
    if divisor == 0 then
        result = 0;
    elsif (dividend < 0) == (divisor < 0) then
        result = Abs(dividend) DIVRM Abs(divisor);    // same signs - positive result
    else
        result = -(Abs(dividend) DIVRM Abs(divisor)); // different signs - negative result
    end;
    R(d) = result[31:0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.182 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the PSTATE.GE flags.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–20	19–16	15–12	11–8	7–5	4	3–0
!=1111 (cond)	0 1 1	0 1 0 0 0	Rn	Rd	(1) (1) (1) (1)	1 0 1	1	Rm

Encoding

```
SEL{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	1 1 0 1 0	1	0 1 0 (op1)	Rn	1 1 1 1	Rd	1 0	0 0 (op2)	Rm

Encoding

```
SEL{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[7:0]   = if PSTATE.GE[0] == '1' then R(n)[7:0]   else R(m)[7:0];
    R(d)[15:8]  = if PSTATE.GE[1] == '1' then R(n)[15:8]  else R(m)[15:8];
    R(d)[23:16] = if PSTATE.GE[2] == '1' then R(n)[23:16] else R(m)[23:16];
    R(d)[31:24] = if PSTATE.GE[3] == '1' then R(n)[31:24] else R(m)[31:24];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.183 SETEND

Set Endianness writes a new value to PSTATE.E.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

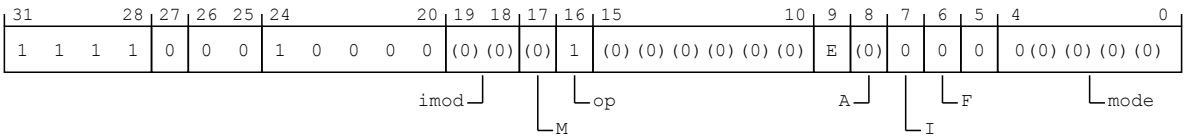

Encoding

```
SETEND{<q>}  <endian_specifier> // (Cannot be conditional)
```

Decode for this encoding

```
let set_bigend : boolean = (E == '1');
```

T1

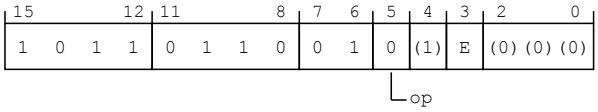

Encoding

```
SETEND{<q>}  <endian_specifier> // (Not permitted in IT block)
```

Decode for this encoding

```
let set_bigend : boolean = (E == '1');
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<endian_specifier>

Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in “E”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E	<endian_specifier>
0	LE
1	BE

Operation

```
EncodingSpecificOperations();
AArch32_CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.184 SETPAN

Set Privileged Access Never writes a new value to PSTATE.PAN.

This instruction is available only in privileged mode and it is a NOP when executed in User mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_PAN)

31–28	27	26–25	24–20	19–10	9	8	7–4	3–0
1 1 1 1	0	0 0	1 0 0 0 1	(0) (0) (0) (0) (0) (0) (0) (0) (0) (0)	imm1	(0)	0 0 0 0	(0) (0) (0) (0)

Encoding

```
SETPAN{<q>}  #<imm> // (Cannot be conditional)
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAN) then Undefined(); end;
let value : bit = imm1;
```

T1

(FEAT_PAN)

15–12	11–8	7	6	5	4	3	2–0
1 0 1 1	0 1 1 0	0	0	0	(1)	imm1	(0) (0) (0)

Encoding

```
SETPAN{<q>}  #<imm> // (Not permitted in IT block)
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_PAN) then Undefined(); end;
let value : bit = imm1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<imm>

Is the unsigned immediate 0 or 1, encoded in the “imm1” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
if PSTATE.EL != EL0 then
    PSTATE.PAN = value;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.185 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see Wait For Event and Send Event.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	0	1	1	0	0	1	0	0 0 0 0	(1) (1) (1) (1)	(0) (0) (0) (0) 00000100
cond						R			imm4		imm12

Encoding

```
SEV{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–12	11–8	7–4	3–0
1 0 1 1	1 1 1 1	0 1 0 0	0 0 0 0
		hint	

Encoding

```
SEV{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15–13	12	11	10	9–6	5	4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1	0	0	1 1 1 0	1	0	(1) (1) (1) (1)	1	0 (0) 0	(0)	0 0 0	0 0 0 0	0 1 0 0
												hint	option

Encoding

```
SEV{<c>}.W
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.186 SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

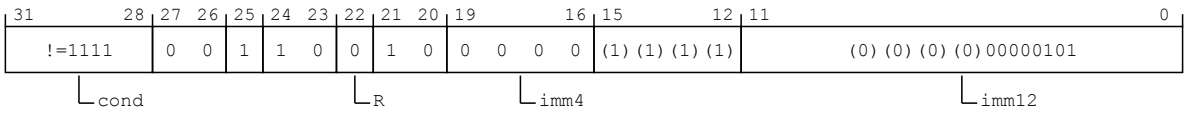

Encoding

```
SEVL{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

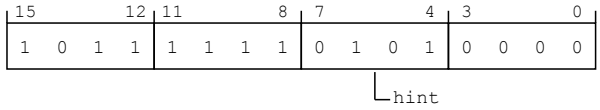

Encoding

```
SEVL{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

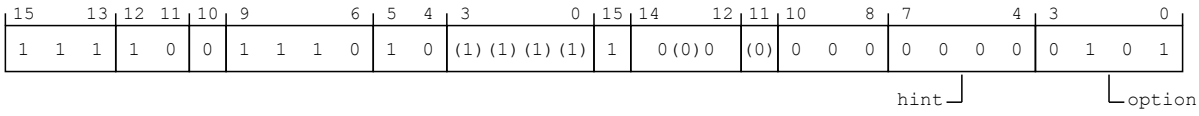

Encoding

```
SEVL{<c>}.W
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.187 SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

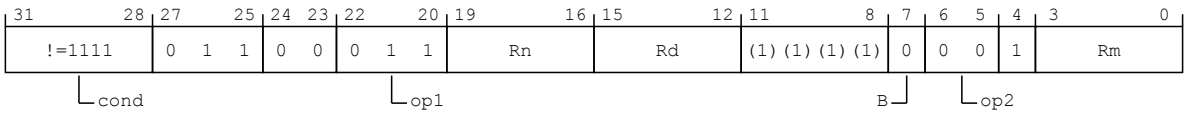

Encoding

```
SHADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

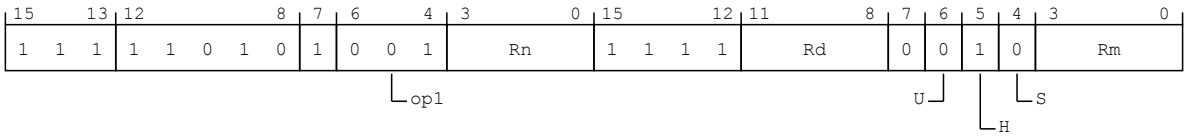

Encoding

```
SHADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0]  = sum1[16:1];
    R(d)[31:16] = sum2[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.188 SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

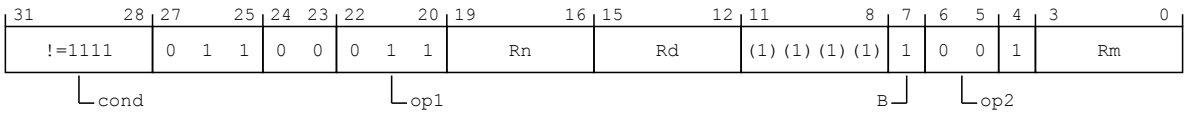

Encoding

```
SHADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

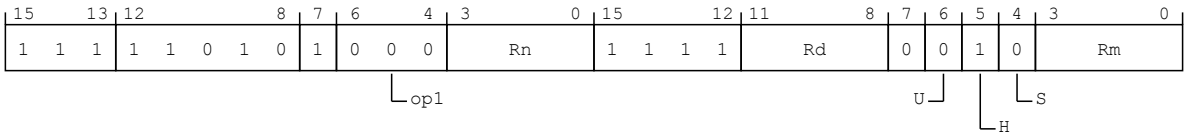

Encoding

```
SHADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0]   = sum1[8:1];
    R(d)[15:8]  = sum2[8:1];
    R(d)[23:16] = sum3[8:1];
    R(d)[31:24] = sum4[8:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.189 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

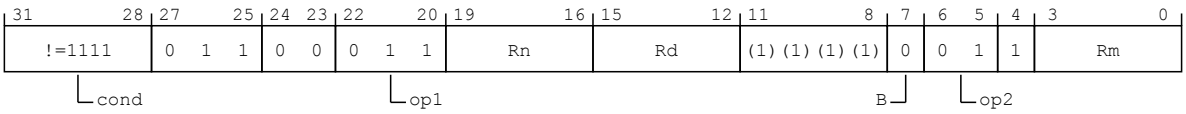

Encoding

```
SHASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

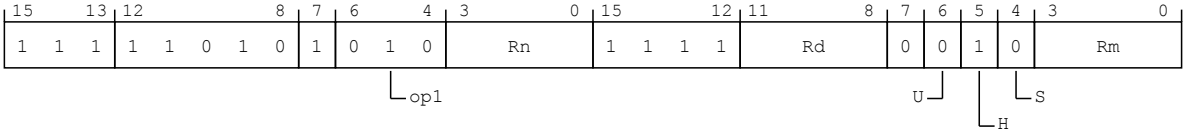

Encoding

```
SHASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer  = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0]  = diff[16:1];
    R(d)[31:16] = sum[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.190 SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

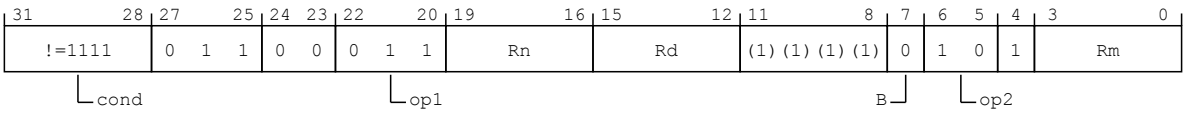

Encoding

```
SHSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

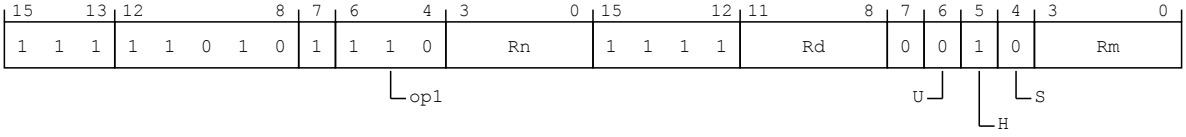

Encoding

```
SHSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer  = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0]  = sum[16:1];
    R(d)[31:16] = diff[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.191 SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

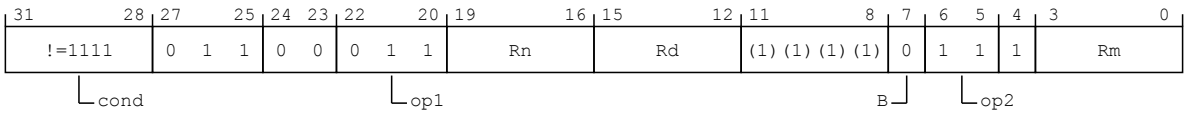

Encoding

```
SHSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

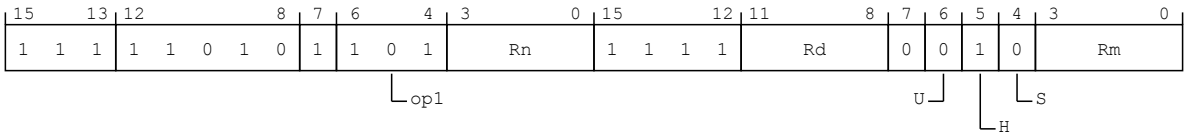

Encoding

```
SHSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0]  = diff1[16:1];
    R(d)[31:16] = diff2[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.192 SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22–20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 1 1	0 0	0 1 1 (op1)	Rn	Rd	(1) (1) (1) (1)	1 (B)	1 1 (op2)	1	Rm

Encoding

```
SHSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–8	7	6–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1 1 0 1 0	1	1 0 0 (op1)	Rn	1 1 1 1	Rd	0	0 (U)	1 (H)	0 (S)	Rm

Encoding

```
SHSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0]   = diff1[8:1];
    R(d)[15:8]  = diff2[8:1];
    R(d)[23:16] = diff3[8:1];
    R(d)[31:24] = diff4[8:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.193 SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information, see Secure Monitor Call (SMC) exception.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of HCR.TSC and SCR.SCD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the SCR_EL3.SMD bit, rather than the SCR.SCD bit, that can change the effect of executing an SMC instruction.

If the value of HCR.TSC is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of SCR.SCD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information, see HCR.TSC.

If the value of HCR.TSC is 0 and the value of SCR.SCD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	23–22	21	20	19–8	7	6–4	3–0
!=1111	0	0	0	1 0	1	0	(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)	0	1 1 1	imm4
cond					opc		imm12			

Encoding

```
SMC{<c>}{<q>}  {#}<imm4>
```

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

T1

15–13	12–11	10	9–6	5	4	3–0	15	14	13	12	11–0
1 1 1	1 0	1	1 1 1 1	1	1	imm4	1	0	0	0	(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
					o1				o2		imm12

Encoding

```
SMC{<c>}{<q>}  {#}<imm4>
```

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<imm4>

Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the “imm4” field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    AArch32_CheckForSMCUndefOrTrap();

    if !ELUsingAArch32(EL3) then
        if SCR_EL3().SMD == '1' then
            // SMC disabled.
            Undefined();
        end;
    else
        if SCR().SCD == '1' then
            // SMC disabled
            if CurrentSecurityState() == SS_Secure then
                // Executes either as a NOP or UNALLOCATED.
                let c : Constraint = ConstrainUnpredictable(Unpredictable_SMD);
                assert c IN {Constraint_NOP, Constraint_UNDEF};
                if c == Constraint_NOP then ExecuteAsNOP(); end;
            end;
            Undefined();
        end;
    end;

    if !ELUsingAArch32(EL3) then
        AArch64_CallSecureMonitor(Zeros{16});
    else
        AArch32_TakeSMCException();
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && CurrentSecurityState() == SS_Secure`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.194 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. It is not possible for overflow to occur during the multiplication.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

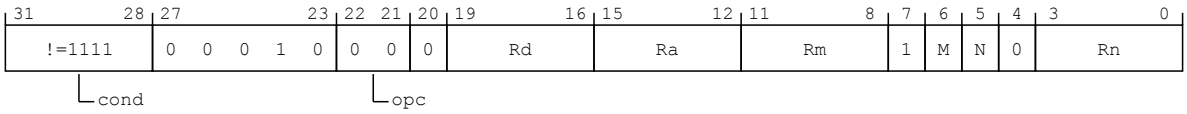

Encoding for the SMLABB variant

Applies when `(M == 0 && N == 0)`

```
SMLABB{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLABT variant

Applies when `(M == 1 && N == 0)`

```
SMLABT{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLATB variant

Applies when `(M == 0 && N == 1)`

```
SMLATB{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLATT variant

Applies when `(M == 1 && N == 1)`

```
SMLATT{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–7	6–4	3–0	15–12	11–8	7–6	5	4	3–0
1 1 1 1 1 0 1 1 0	0 0 1 (op1)	Rn	!=1111 (Ra)	Rd	0 0	N	M	Rm

Encoding for the SMLABB variant

Applies when `(N == 0 && M == 0)`

```
SMLABB{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLABT variant

Applies when `(N == 0 && M == 1)`

```
SMLABT{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLATB variant

Applies when `(N == 1 && M == 0)`

```
SMLATB{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLATT variant

Applies when `(N == 1 && M == 1)`

```
SMLATT{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMULBB, SMULBT, SMULTB, SMULTT"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*:16];
    let operand2 : bits(16) = R(m)[m_index*:16];
    let result : integer = SInt(operand1) * SInt(operand2) + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then  // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.195 SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

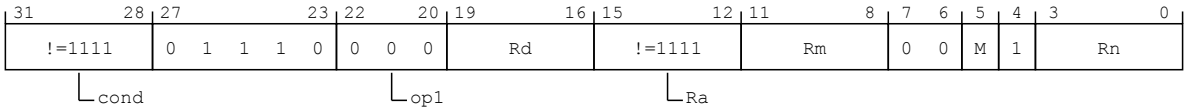

Encoding for the SMLAD variant

Applies when `(M == 0)`

```
SMLAD{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLADX variant

Applies when `(M == 1)`

```
SMLADX{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUAD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

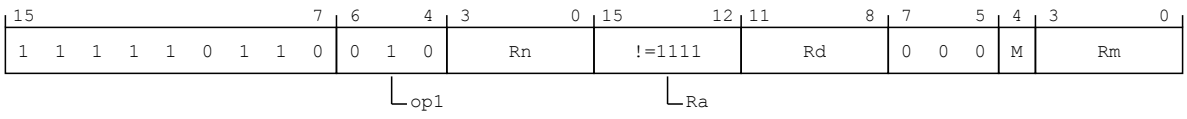

Encoding for the SMLAD variant

Applies when `(M == 0)`

```
SMLAD{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SMLADX variant

Applies when `(M == 1)`

```
SMLADX{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUAD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2 + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then  // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.196 SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–24	23–21	20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 1 1 (opc)	S	RdHi	RdLo	Rm	1 0 0 1	Rn

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
SMLALS{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
SMLAL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–13	12–9	8 7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1 1	1 0 0 (op1)	Rn	RdLo	RdHi	0 0 0 0 (op2)	Rm

Encoding

```
SMLAL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = SInt(R(n)) * SInt(R(m)) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.197 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords)

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

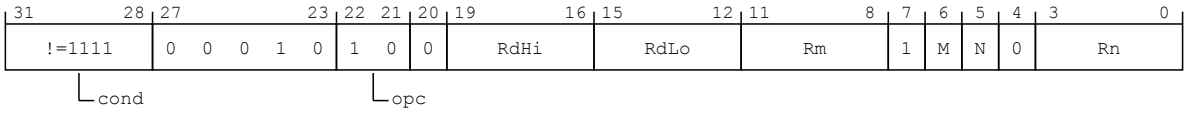

Encoding for the SMLALBB variant

Applies when `(M == 0 && N == 0)`

```
SMLALBB{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALBT variant

Applies when `(M == 1 && N == 0)`

```
SMLALBT{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALTB variant

Applies when `(M == 0 && N == 1)`

```
SMLALTB{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALTT variant

Applies when `(M == 1 && N == 1)`

```
SMLALTT{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15 … 7	6 … 4	3 … 0	15 … 12	11 … 8	7 … 6	5	4	3 … 0
1 1 1 1 1 0 1 1 1	1 0 0 (op1)	Rn	RdLo	RdHi	1 0	N	M	Rm

Encoding for the SMLALBB variant

Applies when `(N == 0 && M == 0)`

```
SMLALBB{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALBT variant

Applies when `(N == 0 && M == 1)`

```
SMLALBT{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALTB variant

Applies when `(N == 1 && M == 0)`

```
SMLALTB{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALTT variant

Applies when `(N == 1 && M == 1)`

```
SMLALTT{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

For the “A1 SMLALBB”, “A1 SMLALBT”, “A1 SMLALTB”, and “A1 SMLALTT” variants: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the “Rn” field.

For the “T1 SMLALBB”, “T1 SMLALBT”, “T1 SMLALTB”, and “T1 SMLALTT” variants: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the “Rn” field.

<Rm>

For the “A1 SMLALBB”, “A1 SMLALBT”, “A1 SMLALTB”, and “A1 SMLALTT” variants: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the “Rm” field.

For the “T1 SMLALBB”, “T1 SMLALBT”, “T1 SMLALTB”, and “T1 SMLALTT” variants: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*:16];
    let operand2 : bits(16) = R(m)[m_index*:16];
    let result : integer = SInt(operand1) * SInt(operand2) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.198 SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual

Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

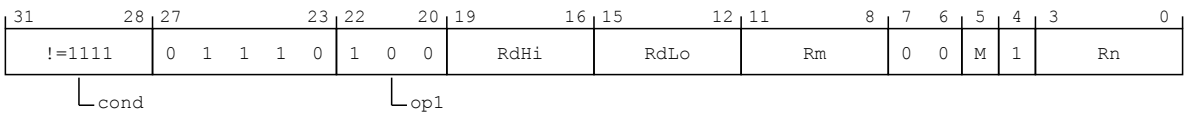

Encoding for the SMLALD variant

Applies when `(M == 0)`

```
SMLALD{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALDX variant

Applies when `(M == 1)`

```
SMLALDX{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–7	6–4	3–0	15–12	11–8	7–5	4	3–0
1 1 1 1 1 0 1 1 1	1 0 0	Rn	RdLo	RdHi	1 1 0	M	Rm

op1

Encoding for the SMLALD variant

Applies when `(M == 0)`

```
SMLALD{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLALDX variant

Applies when `(M == 1)`

```
SMLALDX{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2 + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.199 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. No overflow can occur during the multiplication.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

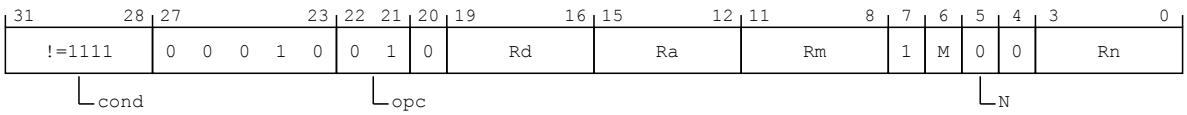

Encoding for the SMLAWB variant

Applies when `(M == 0)`

```
SMLAWB{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLAWT variant

Applies when `(M == 1)`

```
SMLAWT{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_high : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

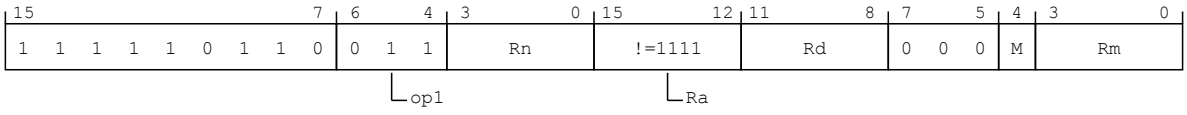

Encoding for the SMLAWB variant

Applies when `(M == 0)`

```
SMLAWB{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SMLAWT variant

Applies when `(M == 1)`

```
SMLAWT{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMULWB, SMULWT"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_high : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(16) = if m_high then R(m)[31:16] else R(m)[15:0];
    let result : integer = SInt(R(n)) * SInt(operand2) + (SInt(R(a)) << 16);
    R(d) = result[47:16];
    if (result >> 16) != SInt(R(d)) then  // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.200 SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

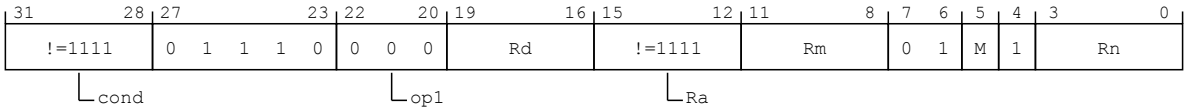

Encoding for the SMLSD variant

Applies when `(M == 0)`

```
SMLSD{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMLSDX variant

Applies when `(M == 1)`

```
SMLSDX{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUSD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

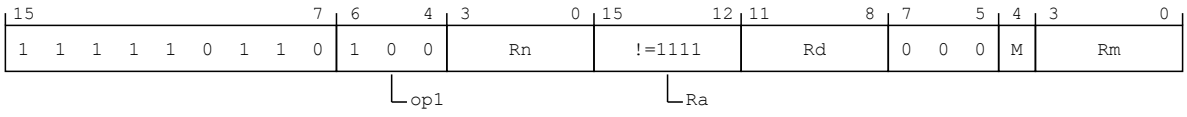

Encoding for the SMLSD variant

Applies when `(M == 0)`

```
SMLSD{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SMLSDX variant

Applies when `(M == 1)`

```
SMLSDX{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUSD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = (product1 - product2) + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then  // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.201 SMLSLD, SMLSLDX

Signed Multiply Subtract Long Dual

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

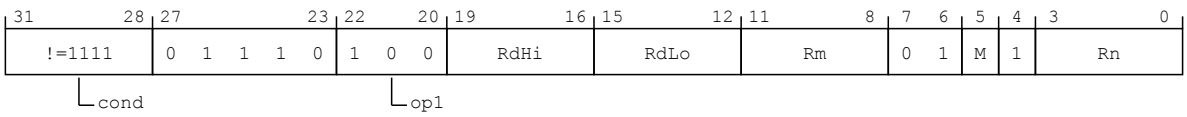

Encoding for the SMLSLD variant

Applies when `(M == 0)`

```
SMLSLD{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLSLDX variant

Applies when `(M == 1)`

```
SMLSLDX{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–7	6–4	3–0	15–12	11–8	7–5	4	3–0
1 1 1 1 1 0 1 1 1	1 0 1 (op1)	Rn	RdLo	RdHi	1 1 0	M	Rm

Encoding for the SMLSLD variant

Applies when `(M == 0)`

```
SMLSLD{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the SMLSLDX variant

Applies when `(M == 1)`

```
SMLSLDX{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = (product1 - product2) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.202 SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

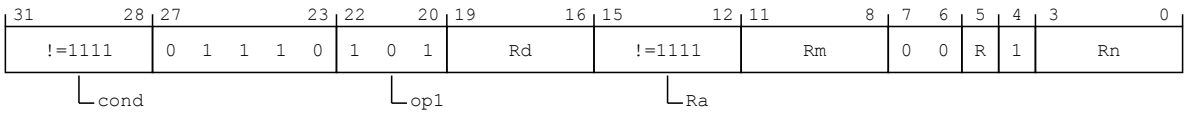

Encoding for the SMMLA variant

Applies when `(R == 0)`

```
SMMLA{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMMLAR variant

Applies when `(R == 1)`

```
SMMLAR{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMMUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

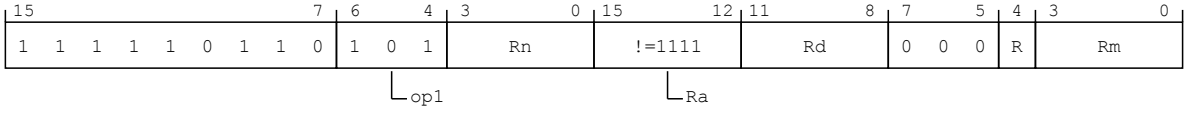

Encoding for the SMMLA variant

Applies when `(R == 0)`

```
SMMLA{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SMMLAR variant

Applies when `(R == 1)`

```
SMMLAR{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMMUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = (SInt(R(a)) << 32) + SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.203 SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the result of the subtraction before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–23	22–20	19–16	15–12	11–8	7–6	5	4	3–0
!=1111 (cond)	0 1 1 1 0	1 0 1 (op1)	Rd	Ra	Rm	1 1	R	1	Rn

Encoding for the SMMLS variant

Applies when `(R == 0)`

```
SMMLS{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMMLSR variant

Applies when `(R == 1)`

```
SMMLSR{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

15–7	6–4	3–0	15–12	11–8	7–5	4	3–0
1 1 1 1 1 0 1 1 0	1 1 0 (op1)	Rn	Ra	Rd	0 0 0	R	Rm

Encoding for the SMMLS variant

Applies when `(R == 0)`

```
SMMLS{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Encoding for the SMMLSR variant

Applies when `(R == 1)`

```
SMMLSR{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = (SInt(R(a)) << 32) - SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.204 SMMUL, SMMULR

Signed Most Significant Word Multiply

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

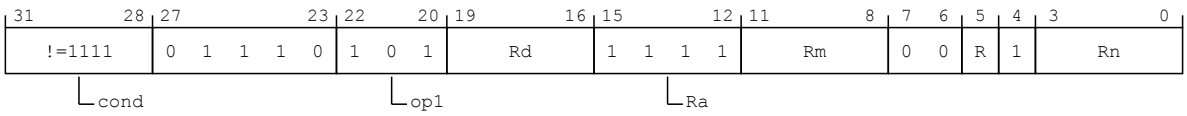

Encoding for the SMMUL variant

Applies when `(R == 0)`

```
SMMUL{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMMULR variant

Applies when `(R == 1)`

```
SMMULR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

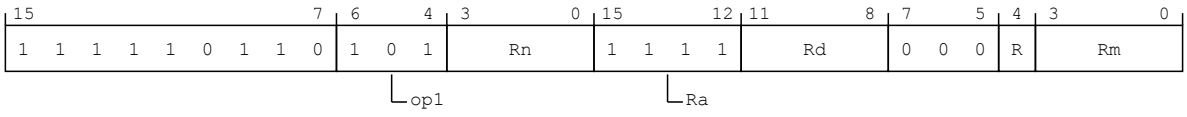

Encoding for the SMMUL variant

Applies when `(R == 0)`

```
SMMUL{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMMULR variant

Applies when `(R == 1)`

```
SMMULR{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.205 SMUAD, SMUADX

Signed Dual Multiply Add

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the addition overflows. The multiplications cannot overflow.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

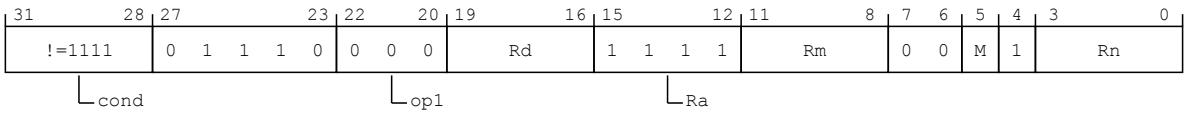

Encoding for the SMUAD variant

Applies when `(M == 0)`

```
SMUAD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMUADX variant

Applies when `(M == 1)`

```
SMUADX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

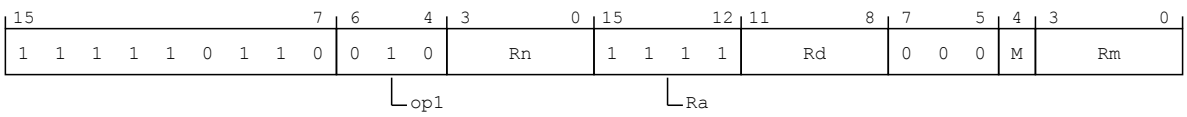

Encoding for the SMUAD variant

Applies when `(M == 0)`

```
SMUAD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SMUADX variant

Applies when `(M == 1)`

```
SMUADX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2;
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then  // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.206 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords)

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

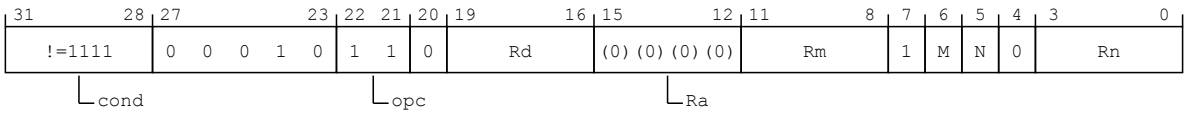

Encoding for the SMULBB variant

Applies when `(M == 0 && N == 0)`

```
SMULBB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULBT variant

Applies when `(M == 1 && N == 0)`

```
SMULBT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULTB variant

Applies when `(M == 0 && N == 1)`

```
SMULTB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULTT variant

Applies when `(M == 1 && N == 1)`

```
SMULTT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–7	6–4	3–0	15–12	11–8	7–6	5	4	3–0
1 1 1 1 1 0 1 1 0	0 0 1 (op1)	Rn	1 1 1 1 (Ra)	Rd	0 0	N	M	Rm

Encoding for the SMULBB variant

Applies when `(N == 0 && M == 0)`

```
SMULBB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULBT variant

Applies when `(N == 0 && M == 1)`

```
SMULBT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULTB variant

Applies when `(N == 1 && M == 0)`

```
SMULTB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULTT variant

Applies when `(N == 1 && M == 1)`

```
SMULTT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*:16];
    let operand2 : bits(16) = R(m)[m_index*:16];
    let result : integer = SInt(operand1) * SInt(operand2);
    R(d) = result[31:0];
    // Signed overflow cannot occur
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.207 SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–24	23–21	20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 1 0 (opc)	S	RdHi	RdLo	Rm	1 0 0 1	Rn

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
SMULLS{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
SMULL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–13	12–9	8–7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1 1	0 0 0 (op1)	Rn	RdLo	RdHi	0 0 0 0 (op2)	Rm

Encoding

```
SMULL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = SInt(R(n)) * SInt(R(m));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.208 SMULWB, SMULWT

Signed Multiply (word by halfword)

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

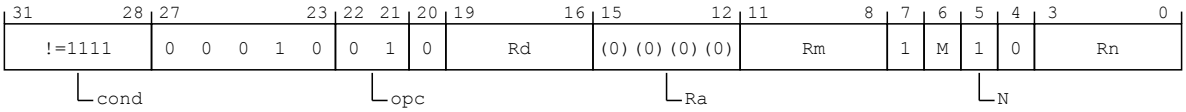

Encoding for the SMULWB variant

Applies when `(M == 0)`

```
SMULWB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULWT variant

Applies when `(M == 1)`

```
SMULWT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_high : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

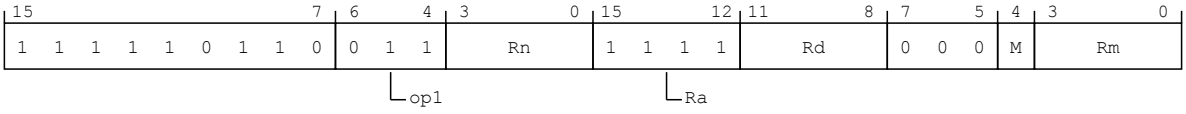

Encoding for the SMULWB variant

Applies when `(M == 0)`

```
SMULWB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMULWT variant

Applies when `(M == 1)`

```
SMULWT{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_high : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(16) = if m_high then R(m)[31:16] else R(m)[15:0];
    let product : integer = SInt(R(n)) * SInt(operand2);
    R(d) = product[47:16];
    // Signed overflow cannot occur
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.209 SMUSD, SMUSDX

Signed Multiply Subtract Dual

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

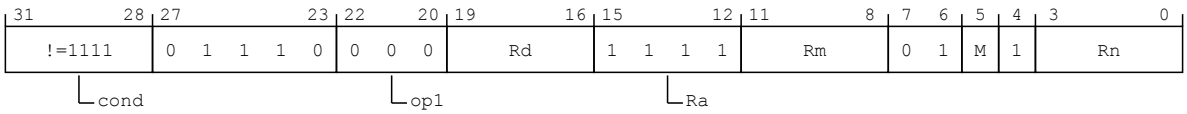

Encoding for the SMUSD variant

Applies when `(M == 0)`

```
SMUSD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Encoding for the SMUSDX variant

Applies when `(M == 1)`

```
SMUSDX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

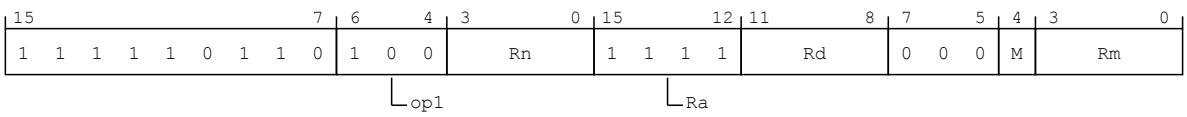

Encoding for the SMUSD variant

Applies when `(M == 0)`

```
SMUSD{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SMUSDX variant

Applies when `(M == 1)`

```
SMUSDX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 - product2;
    R(d) = result[31:0];
    // Signed overflow cannot occur
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.210 SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see Memory accesses.

`SRS` is UNDEFINED in Hyp mode.

`SRS` is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that AArch32 PE modes does not show.
- Hyp mode.
- Monitor mode, if the `SRS` instruction is executed in Non-secure state.

If EL3 is using AArch64 and an `SRS` instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

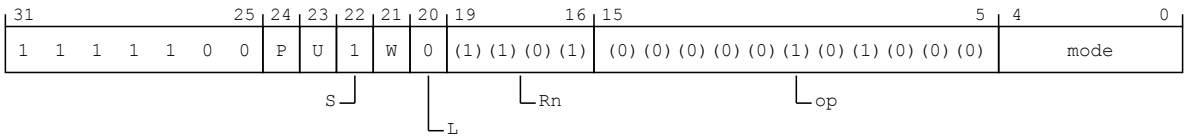

Encoding for the Decrement After variant

Applies when `(P == 0 && U == 0)`

```
SRSDA{<c>}{<q>}  SP{!}, #<mode>
```

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0)`

```
SRSDB{<c>}{<q>}  SP{!}, #<mode>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
SRS{IA}{<c>}{<q>}  SP{!}, #<mode>
```

Encoding for the Increment Before variant

Applies when `(P == 1 && U == 1)`

```
SRSIB{<c>}{<q>}  SP{!}, #<mode>
```

Decode for all variants of this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

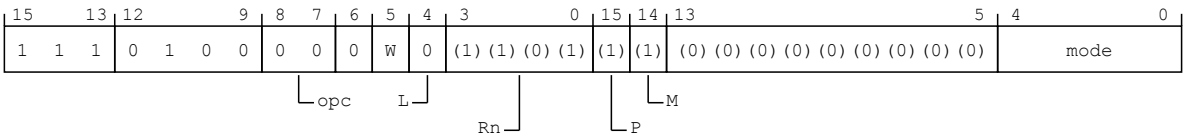

Encoding

```
SRSDB{<c>}{<q>}  SP{!}, #<mode>
```

Decode for this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = FALSE;
let wordhigher : boolean = FALSE;
```

T2

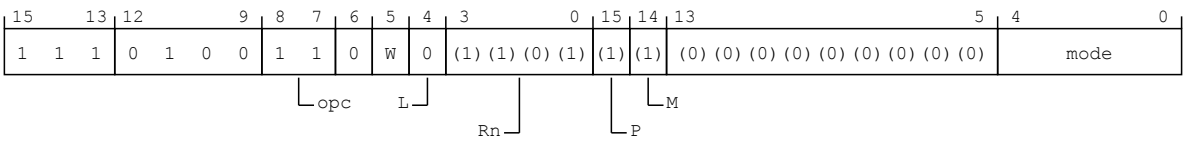

Encoding

```
SRS{IA}{<c>}{<q>}  SP{!}, #<mode>
```

Decode for this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = TRUE;
let wordhigher : boolean = FALSE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly SRS (T32) and SRS (A32).

Assembler Symbols

<c>

For the “Decrement After”, “Decrement Before”, “Increment After”, and “Increment Before” variants: see Standard assembler syntax fields. `<c>` must be `AL` or omitted.

For the “T1” and “T2” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<mode>

Is the number of the mode whose Banked SP is used as the base register, encoded in the “mode” field. For details of PE modes and their numbers see AArch32 state PE modes.

IA

For the “Increment After” variant: is an optional suffix to indicate the Increment After variant.

For the “T2” variant: is an optional suffix for the Increment After form.

`SRSFA`, `SRSEA`, `SRSFD`, and `SRSED` are pseudo-instructions for `SRSIB`, `SRSIA`, `SRSDB`, and `SRSDA` respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            Undefined();
        end;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UnpredictableProcedure();
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
                UnpredictableProcedure();
            elsif !ELUsingAArch32(EL3) then
                AArch64_MonitorModeTrap();
            end;
        elsif BadMode(mode) then
            UnpredictableProcedure();
        end;

        let base : bits(32) = Rmode(13,mode);
        var address : bits(32) = if increment then base else base-8;
        if wordhigher then address = address+4; end;
        MemA{32}(address)   = LR();
        MemA{32}(address+4) = SPSR_curr();
        if wback then Rmode(13,mode) = if increment then base+8 else base-8; end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            Undefined();
        end;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UnpredictableProcedure();
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
                UnpredictableProcedure();
            elsif !ELUsingAArch32(EL3) then
                AArch64_MonitorModeTrap();
            end;
        elsif BadMode(mode) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            UnpredictableProcedure();
        end;

        let base : bits(32) = Rmode(13,mode);
        var address : bits(32) = if increment then base else base-8;
        if wordhigher then address = address+4; end;
        MemA{32}(address)   = LR();
        MemA{32}(address+4) = SPSR_curr();
        if wback then Rmode(13,mode) = if increment then base+8 else base-8; end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `mode == M32_Hyp`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `mode == M32_Monitor && (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `BadMode(mode)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction stores to the stack of the mode in which it is executed.
- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.211 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–23	22	21	20–16	15–12	11–7	6	5	4	3–0
!=1111 (cond)	0 1 1 0 1	0 (U)	1	sat_imm	Rd	imm5	sh	0	1	Rn

Encoding for the Arithmetic shift right variant

Applies when `(sh == 1)`

```
SSAT{<c>}{<q>}  <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when `(sh == 0)`

```
SSAT{<c>}{<q>}  <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(sh::'0', imm5);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15–6	5	4	3–0	15	14–12	11–8	7–6	5	4–0
11110(0)1100	sh	0	Rn	0	imm3	Rd	imm2	(0)	sat_imm

Encoding for the Arithmetic shift right variant

Applies when `(sh == 1 && !(imm3 == 000 && imm2 == 00))`

```
SSAT{<c>}{<q>}  <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when `(sh == 0)`

```
SSAT{<c>}{<q>}  <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if sh == '1' && (imm3::imm2) == '00000' then See("SSAT16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(sh::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 1 to 32, encoded in the “sat_imm” field as <imm>-1.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<amount>

For the “A1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 32 encoded in the “imm5” field as <amount> modulo 32.

For the “A1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 31 encoded in the “imm3:imm2” field as <amount>.

For the “T1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm3:imm2” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand : bits(32) = Shift{}(R(n), shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    let (result, sat) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(operand));
    R(d) = SignExtend{32}(result);
    if sat then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.212 SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22	21–20	19–16	15–12	11–8	7–5	4	3–0
!=1111 (cond)	0 1 1	0 1	0 (U)	1 0	sat_imm	Rd	(1)(1)(1)(1)	0 0 1	1	Rn

Encoding

```
SSAT16{<c>}{<q>}  <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–11	10	9	8	7–5	4	3–0	15	14–12	11–8	7–6	5	4	3–0
1 1 1	1 0	(0)	1	1	0 0 1 (op1)	0	Rn	0	0 0 0 (imm3)	Rd	0 0 (imm2)	(0)	(0)	sat_imm

Encoding

```
SSAT16{<c>}{<q>}  <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 1 to 16, encoded in the “sat_imm” field as <imm>-1.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result1, sat1) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(R(n)[15:0]));
    let (result2, sat2) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(R(n)[31:16]));
    R(d)[15:0] = SignExtend{16}(result1);
    R(d)[31:16] = SignExtend{16}(result2);
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.213 SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

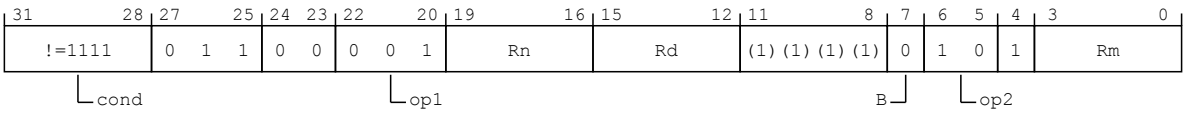

Encoding

```
SSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

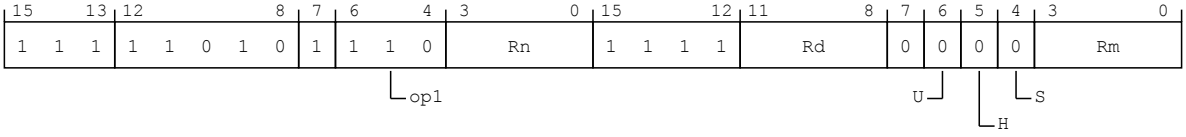

Encoding

```
SSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer  = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0]  = sum[15:0];
    R(d)[31:16] = diff[15:0];
    PSTATE.GE[1:0] = if sum  >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff >= 0 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.214 SSBB

Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see Speculative Store Bypass Barrier (SSBB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25–21	20	19–8	7–4	3–0
1 1 1 1	0	1	0 1 0 1 1	1	(1) (1) (1) (1) (1) (1) (1) (1) (0) (0) (0) (0)	0 1 0 0 (opcode)	0 0 0 0 (option)

Encoding

```
SSBB{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–4	3–0
1 1 1	1 0	0	1 1 1 0	1 1	(1) (1) (1) (1)	1	0 (0) 0	(1)	(1) (1) (1)	0 1 0 0 (opc)	0 0 0 0 (option)

Encoding

```
SSBB{<q>}
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

Operation

```
EncodingSpecificOperations();
SpeculativeStoreBypassBarrierToVA();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.215 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

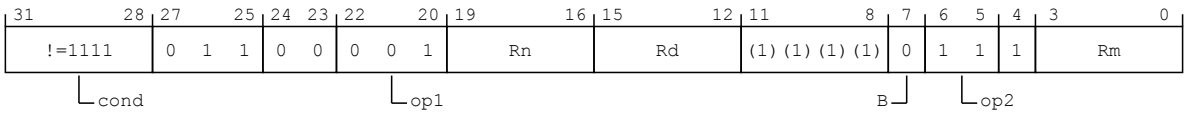

Encoding

```
SSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

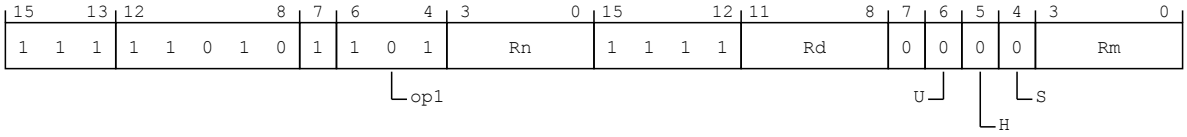

Encoding

```
SSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0]  = diff1[15:0];
    R(d)[31:16] = diff2[15:0];
    PSTATE.GE[1:0] = if diff1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff2 >= 0 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.216 SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

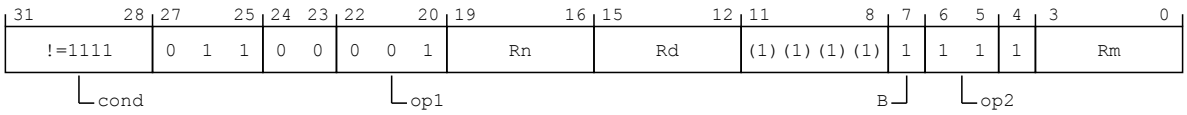

Encoding

```
SSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

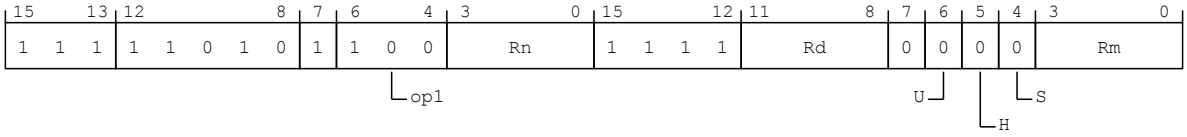

Encoding

```
SSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0]   = diff1[7:0];
    R(d)[15:8]  = diff2[7:0];
    R(d)[23:16] = diff3[7:0];
    R(d)[31:24] = diff4[7:0];
    PSTATE.GE[0]  = if diff1 >= 0 then '1' else '0';
    PSTATE.GE[1]  = if diff2 >= 0 then '1' else '0';
    PSTATE.GE[2]  = if diff3 >= 0 then '1' else '0';
    PSTATE.GE[3]  = if diff4 >= 0 then '1' else '0';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.217 STC

Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the DBGDTRRXint System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see Memory accesses.

In an implementation that includes EL2, the permitted `STC` access to DBGDTRRXint can be trapped to Hyp mode, meaning that an attempt to execute an `STC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see HDCR.TDA.

For simplicity, the `STC` pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

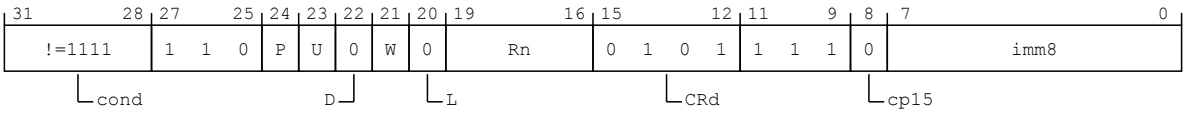

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>, #{+/-}<imm>]!
```

Encoding for the Unindexed variant

Applies when `(P == 0 && U == 1 && W == 0)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

T1

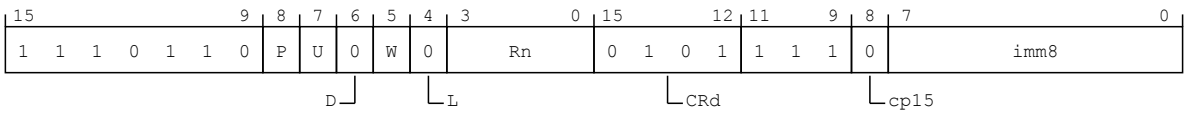

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>, #{+/-}<imm>]!
```

Encoding for the Unindexed variant

Applies when `(P == 0 && U == 1 && W == 0)`

```
STC{<c>}{<q>}  p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the "A1 Offset" and "A1 Unindexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For the "A1 Post-indexed", "A1 Pre-indexed", "T1 Offset", "T1 Post-indexed", "T1 Pre-indexed", and "T1 Unindexed" variants: is the general-purpose base register, encoded in the "Rn" field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

<option>

Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);

    // System register read from DBGDTRRXint.
    AArch32_SysRegRead(cp, ThisInstr(), address[31:0]);
    if wback then R(n) = offset_addr; end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.218 STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

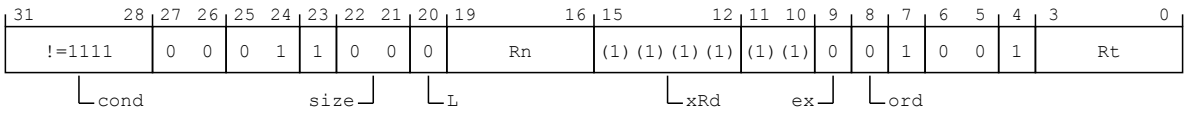

Encoding

```
STL{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

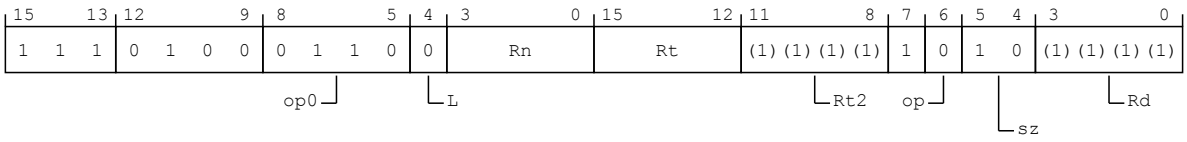

Encoding

```
STL{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{32}(address) = R(t);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.219 STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

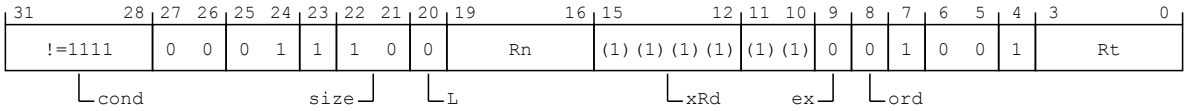

Encoding

```
STLB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

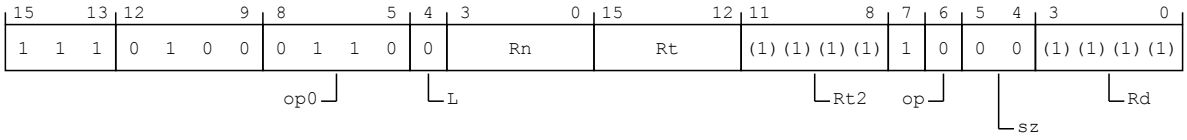

Encoding

```
STLB{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{8}(address) = R(t)[7:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.220 STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

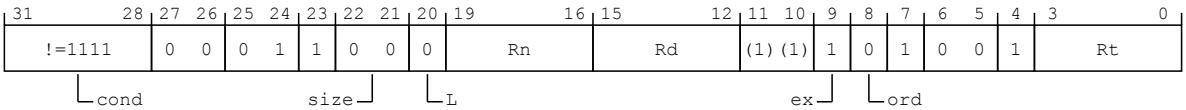

Encoding

```
STLEX{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

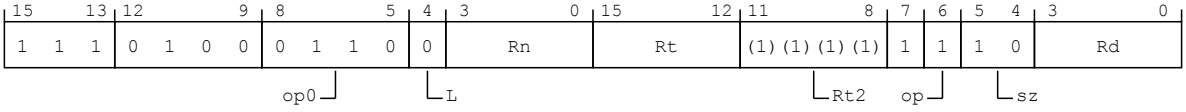

Encoding

```
STLEX{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32_ExclusiveMonitorsPass(address,4) then
        MemO{32}(address) = R(t);
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.221 STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

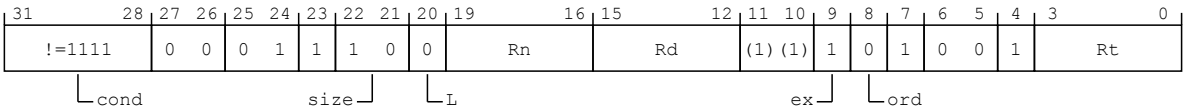

Encoding

```
STLEXB{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

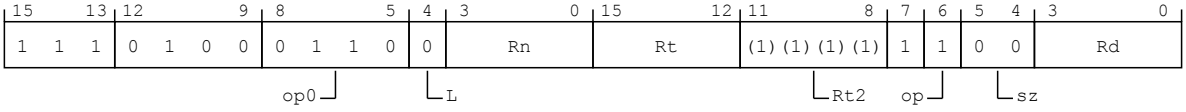

Encoding

```
STLEXB{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32_ExclusiveMonitorsPass(address,1) then
        MemO{8}(address) = R(t)[7:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.222 STLEXD

Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	0	0	0	1	1	0	1	0	Rn	Rd	(1)	(1)	1	0	1	0	0	1	Rt
cond					size			L					ex	ord					

Encoding

```
STLEXD{<c>}{<q>}  <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
if d == 15 || Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: Rt<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 0	0 1 1 0	0	Rn	Rt	Rt2	1 1	1 1	Rd
		op0	L				op	sz	

Encoding

```
STLEXD{<c>}{<q>}  <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. `<Rt>` must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    let value : bits(64) = if BigEndian(AccessType_GPR) then R(t)::R(t2) else R(t2)::R(t);
    if AArch32_ExclusiveMonitorsPass(address, 8) then
        MemO{64}(address) = value;
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.223 STLEXH

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

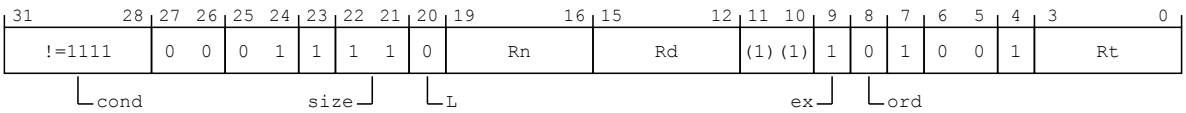

Encoding

```
STLEXH{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

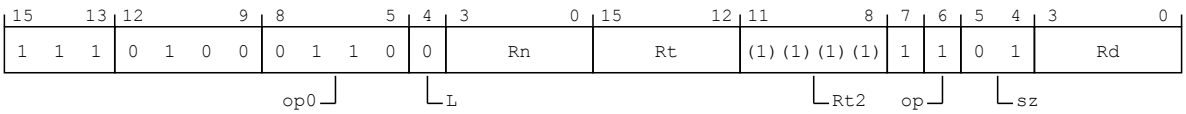

Encoding

```
STLEXH{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32_ExclusiveMonitorsPass(address,2) then
        MemO{16}(address) = R(t)[15:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.224 STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

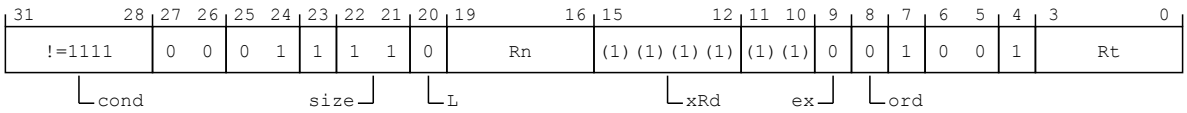

Encoding

```
STLH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

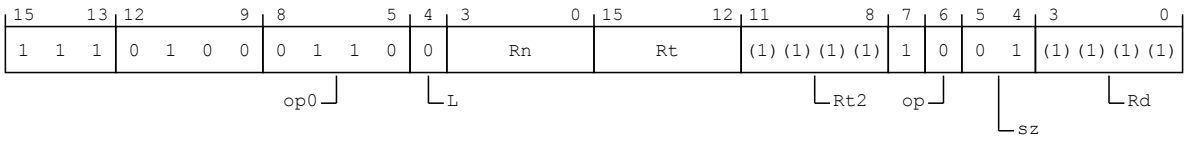

Encoding

```
STLH{<c>}{<q>}  <Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{16}(address) = R(t)[15:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.225 STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

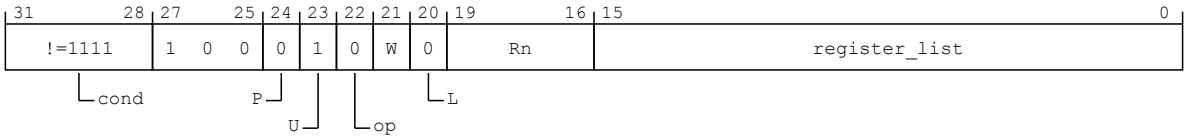

Encoding

```
STM{IA}{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

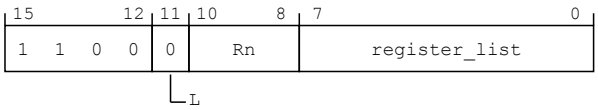

Encoding

```
STM{IA}{<c>}{<q>}  <Rn>!, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q>}  <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = '00000000'::register_list;
let wback : boolean = TRUE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

T2

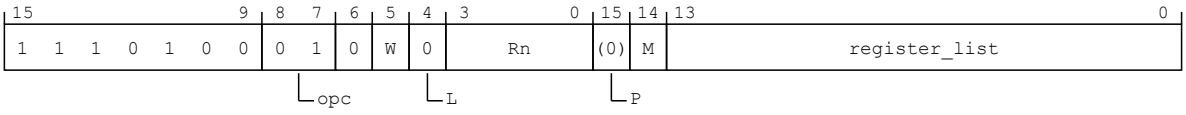

Encoding

```
STM{IA}{<c>}.W  <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be
    ↪represented in T1)
```

```
STMEA{<c>}.W  <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and
    ↪<registers> can be represented in T1)
```

```
STM{IA}{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

IA

Is an optional suffix for the Increment After form.

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

For the “A1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For the “T1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the “register_list” field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For the “T2” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain the LR. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32);  // Only possible for encodings T1 and A1
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then  // Only possible for encoding A1
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) + 4*BitCount(registers); end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.226 STM (User registers)

Store Multiple (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC.

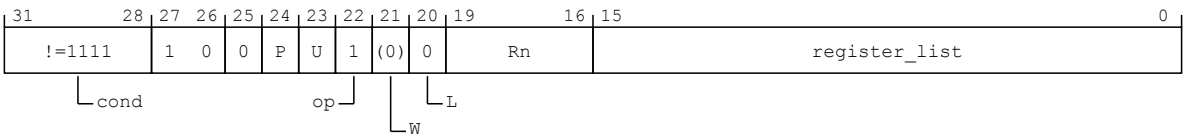

Encoding

```
STM{<amode>}{<c>}{<q>}  <Rn>, <registers>^
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<amode>

is one of:

DA Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
ED Empty Descending. For this instruction, a synonym for `DA`.
DB Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
FD Full Descending. For this instruction, a synonym for `DB`.
IA Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
EA Empty Ascending. For this instruction, a synonym for `IA`.
IB Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FA Full Ascending. For this instruction, a synonym for `IB`.

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<registers>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the `STM` instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.M IN {M32_User,M32_System} then
        UnpredictableProcedure();
    else
        let length : integer = 4*BitCount(registers);
        var address : bits(32) = if increment then R(n) else R(n)-length;
        if wordhigher then address = address+4; end;
        for i = 0 to 14 do
            if registers[i] == '1' then  // Store User mode register
                MemS{32}(address) = Rmode(i, M32_User);
                address = address + 4;
            end;
        end;
        if registers[15] == '1' then
            MemS{32}(address) = PCStoreValue();
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.227 STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

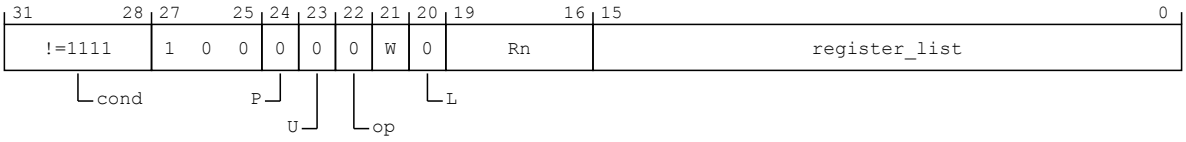

Encoding

```
STMDA{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMED{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = (R(n) - 4*BitCount(registers)) + 4;
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32);
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) - 4*BitCount(registers); end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.228 STMDB, STMFD

Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

This instruction is used by the alias PUSH (multiple registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

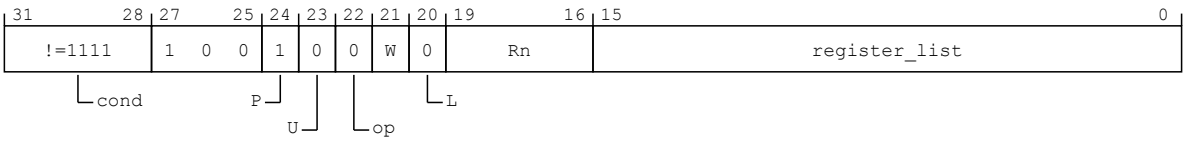

Encoding

```
STMDB{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMFD{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

T1

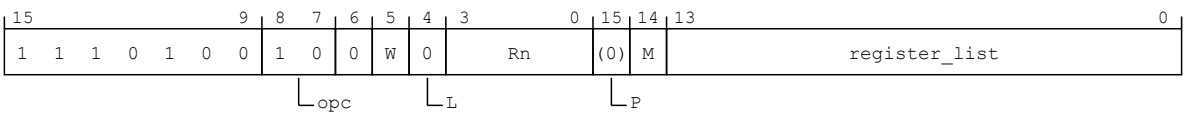

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STMDB{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMFD{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

For the “A1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For the “T1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain the LR. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (multiple registers)	A1	W == '1' && Rn == '1101' && BitCount(register_list) > 1
PUSH (multiple registers)	T1	W == '1' && Rn == '1101' && BitCount(M : register_list) > 1
PUSH (multiple registers)	T1	BitCount(M : register_list) > 1

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) - 4*BitCount(registers);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32);  // Only possible for encoding A1
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then  // Only possible for encoding A1
        MemS{32}(address) = PCStoreValue();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    if wback then R(n) = R(n) - 4*BitCount(registers); end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.229 STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

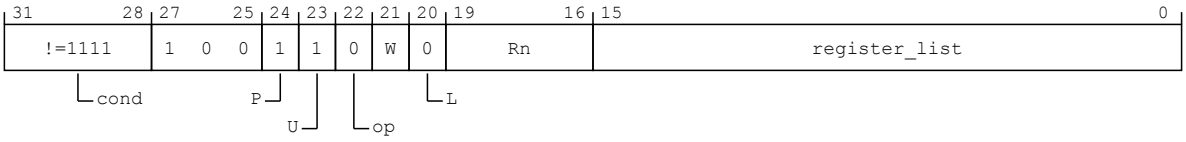

Encoding

```
STMIB{<c>}{<q>}  <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMFA{<c>}{<q>}  <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) + 4;
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32);
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) + 4*BitCount(registers); end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.230 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

This instruction is used by the alias PUSH (single register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, and T4).

A1

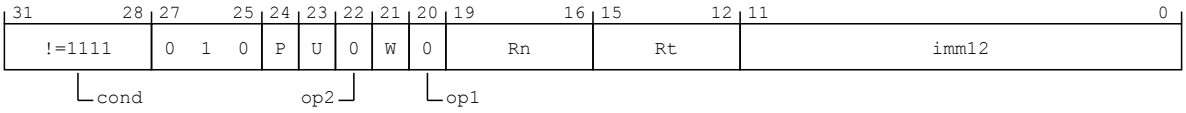

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STR{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

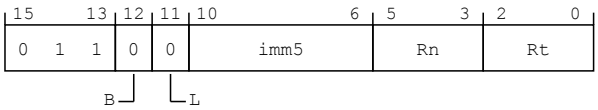

Encoding

```
STR{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

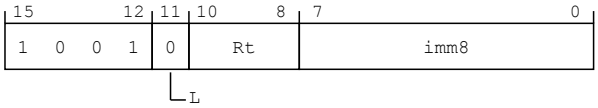

Encoding

```
STR{<c>}{<q>}  <Rt>, [SP{, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = 13;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T3

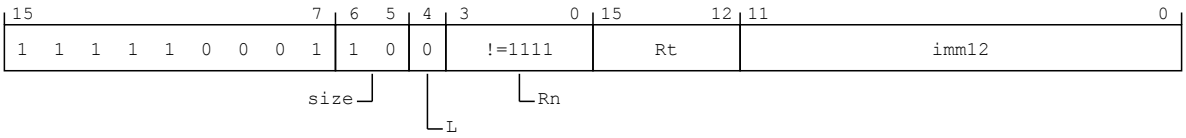

Encoding

```
STR{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

```
STR{<c>}.W  <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T4

15								7	6	5	4	3			0	15			12	11	10	9	8	7							0
1	1	1	1	1	0	0	0	0	1	0	0	!=1111				Rt				1	P	U	W	imm8							
									size		L	Rn																			

Encoding for the Offset variant

Applies when `(P == 1 && U == 0 && W == 0)`

```
STR{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “T1”, “T2”, “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1”, “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the “imm5” field as <imm>/4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “T2” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the “imm8” field as <imm>/4.

For the “T3” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (single register)	A1 Pre-indexed	P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'
PUSH (single register)	T4 Pre-indexed	Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        MemU{32}(address) = if t == 15 then PCStoreValue() else R(t);
        if wback then R(n) = offset_addr; end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        MemU{32}(address) = R(t);
        if wback then R(n) = offset_addr; end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.231 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 1 1	P	U	0	W	0	Rn	Rt	imm5	stype	0	Rm
cond				op2		op1						

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STR{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

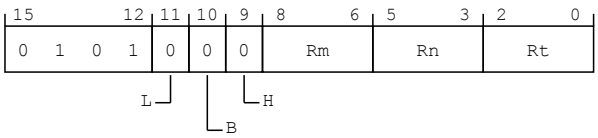

Encoding

```
STR{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

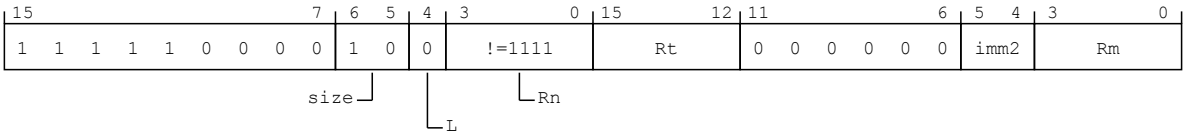

Encoding

```
STR{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
STR{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    var data : bits(32);
    if t == 15 then  // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R(t);
    end;
    MemU{32}(address) = data;
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.232 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

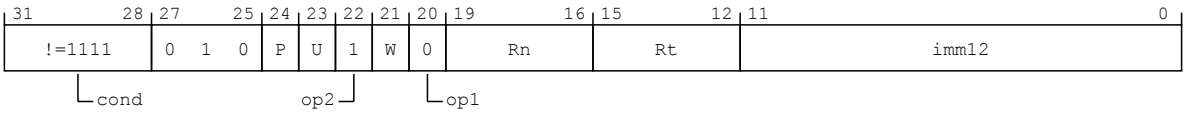

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

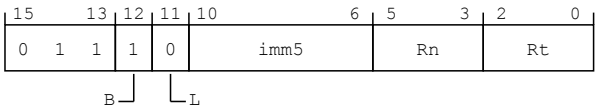

Encoding

```
STRB{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

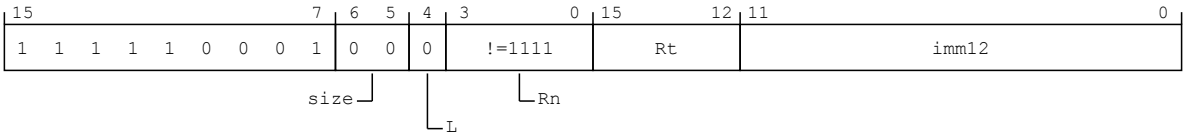

Encoding

```
STRB{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

```
STRB{<c>}.W  <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

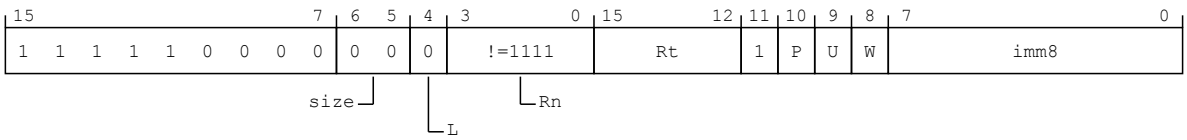

Encoding for the Offset variant

Applies when `(P == 1 && U == 0 && W == 0)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRBT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>

For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For the "T1", "T2", "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For the "T1" variant: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For the "T2" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    MemU{8}(address) = R(t)[7:0];
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.233 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

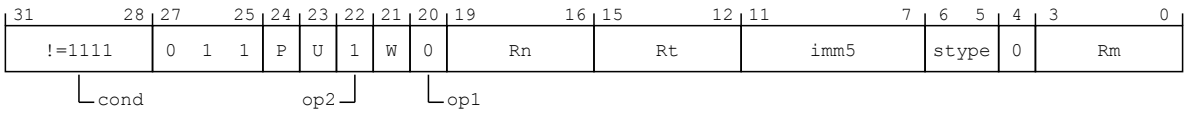

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRB{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

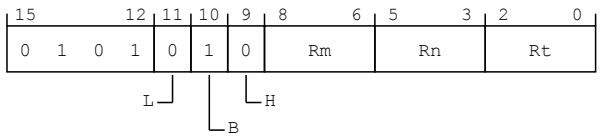

Encoding

```
STRB{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

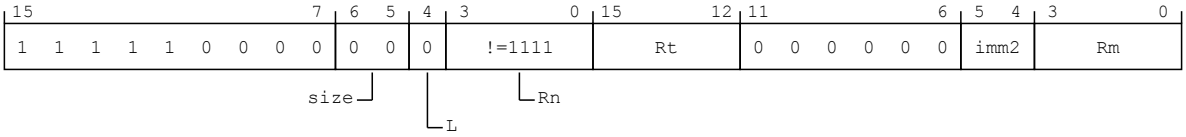

Encoding

```
STRB{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
STRB{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{8}(address) = R(t)[7:0];
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.234 STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`STRBT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

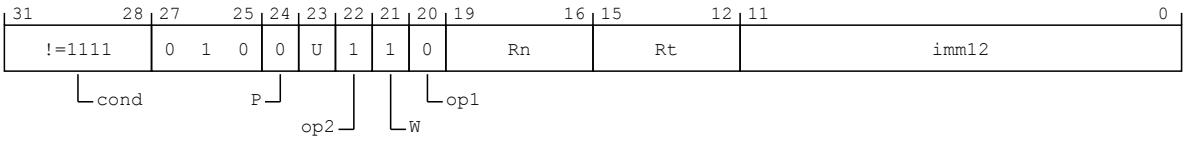

Encoding

```
STRBT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

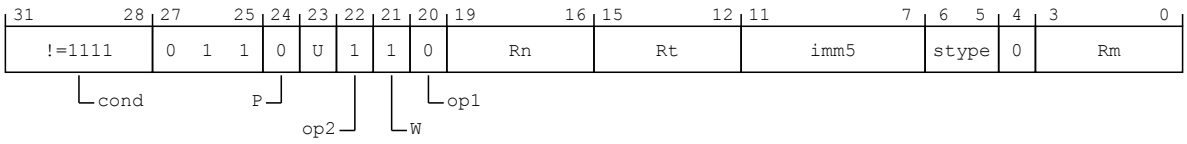

Encoding

```
STRBT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–8	7–0
1 1 1	1 1 0 0	0 0	0 0	0	!=1111	Rt	1 1 1 0	imm8
			size	L	Rn			

Encoding

```
STRBT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “A2” and “T1” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;                // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                                else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    MemU_unpriv{8}(address) = R(t)[7:0];
    if postindex then R(n) = offset_addr; end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STRB` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.235 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1	1 1	1	imm4L
cond						o1					op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == ‘1111’.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as an `LDRD` using one of offset, post-indexed, or pre-indexed addressing.

T1

15–9	8	7	6	5	4	3–0	15–12	11–8	7–0
1 1 1 0 1 0 0	P	U	1	W	0	!=1111	Rt	Rt2	imm8
					L	Rn			

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || t == 15 || t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Load/store dual, load/store exclusive, table branch.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rt2>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the “imm8” field as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        var data : bits(64);
        if BigEndian(AccessType_GPR) then
            data[63:32] = R(t);
            data[31:0] = R(t2);
        else
            data[31:0] = R(t);
            data[63:32] = R(t2);
        end;
        MemA{64}(address) = data;
    else
        MemA{32}(address) = R(t);
        MemA{32}(address+4) = R(t2);
    end;
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.236 STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

31–28	27–25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0	P	U	0	W	0	Rn	Rt	(0)(0)(0)(0)1	1 1	1	Rm
cond						o1				op2		

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRD{<c>}{<q>}  <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if t2 == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: t<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == ‘1111’.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: P = ‘1’; W = ‘0’.
- The instruction executes with the additional decode: P = ‘1’; W = ‘1’.
- The instruction executes with the additional decode: P = ‘0’; W = ‘0’.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

<Rt2>

Is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + R(m)) else (R(n) - R(m));
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        var data : bits(64);
        if BigEndian(AccessType_GPR) then
            data[63:32] = R(t);
            data[31:0] = R(t2);
        else
            data[31:0] = R(t);
            data[63:32] = R(t2);
        end;
        MemA{64}(address) = data;
    else
        MemA{32}(address) = R(t);
        MemA{32}(address+4) = R(t2);
    end;
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.237 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

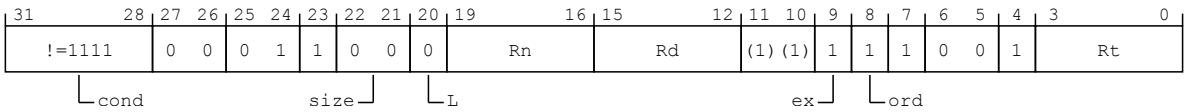

Encoding

```
STREX{<c>}{<q>}  <Rd>, <Rt>, [<Rn> {, {#}<imm>}]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = Zeros{}; // Zero offset
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

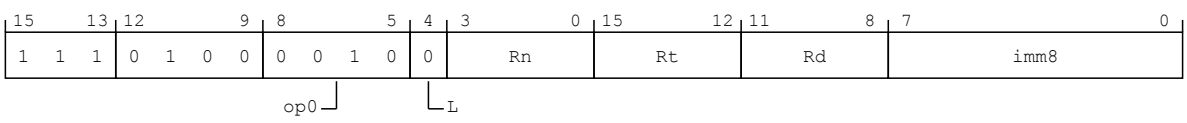

Encoding

```
STREX{<c>}{<q>}  <Rd>, <Rt>, [<Rn> {, #<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<imm>

For the “A1” variant: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can only be 0 or omitted.

For the “T1” variant: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n) + imm32;
    if AArch32_ExclusiveMonitorsPass(address,4) then
        MemA{32}(address) = R(t);
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.238 STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

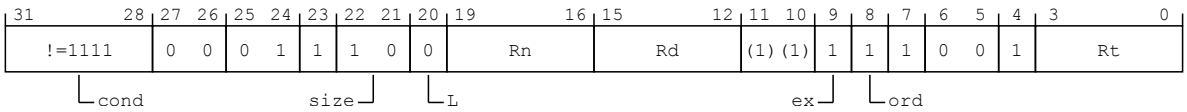

Encoding

```
STREXB{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

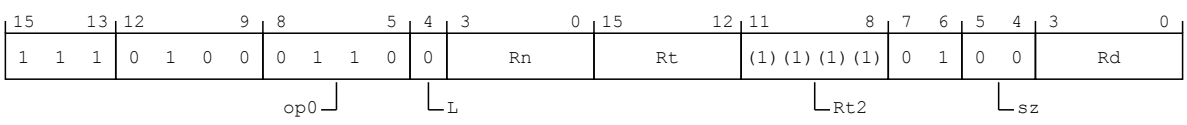

Encoding

```
STREXB{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32_ExclusiveMonitorsPass(address,1) then
        MemA{8}(address) = R(t)[7:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.239 STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

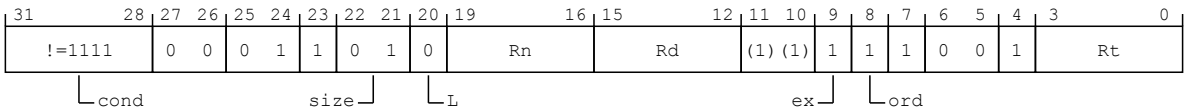

Encoding

```
STREXD{<c>}{<q>}  <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
if d == 15 || Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: Rt<0> = ‘0’.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 0	0 1 1 0	0	Rn	Rt	Rt2	0 1	1 1	Rd
		op0	L					sz	

Encoding

```
STREXD{<c>}{<q>}  <Rd>, <Rt>, <Rt2>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

`<Rd>` must not be the same as `<Rn>`, `<Rt>`, or `<Rt2>`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. `<Rt>` must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);

    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    let value : bits(64) = if BigEndian(AccessType_GPR) then R(t)::R(t2) else R(t2)::R(t);

    if AArch32_ExclusiveMonitorsPass(address,8) then
        MemA{64}(address) = value;
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.240 STREXH

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–24	23	22–21	20	19–16	15–12	11–10	9	8	7	6–5	4	3–0
!=1111	0 0	0 1	1	1 1	0	Rn	Rd	(1) (1)	1	1	1	0 0	1	Rt
cond			size		L				ex	ord				

Encoding

```
STREXH{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

15–13	12–9	8–5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	0 1 0 0	0 1 1 0	0	Rn	Rt	(1) (1) (1) (1)	0 1	0 1	Rd
		op0	L			Rt2		sz	

Encoding

```
STREXH{<c>}{<q>}  <Rd>, <Rt>, [<Rn>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32_ExclusiveMonitorsPass(address,2) then
        MemA{16}(address) = R(t)[15:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.241 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

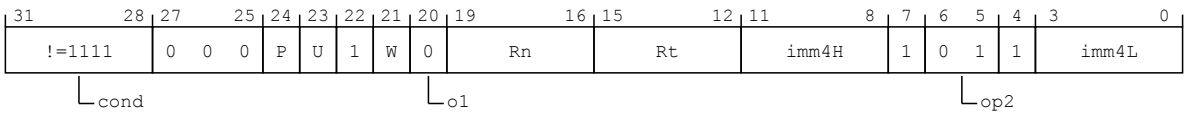

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

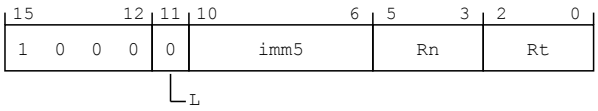

Encoding

```
STRH{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'0');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

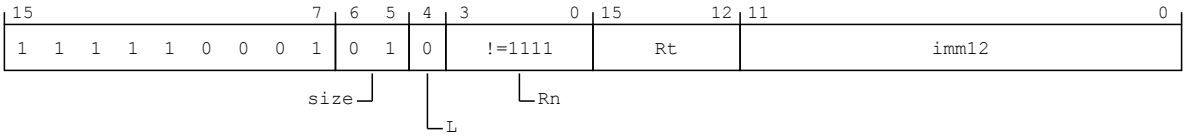

Encoding

```
STRH{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

```
STRH{<c>}.W  <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

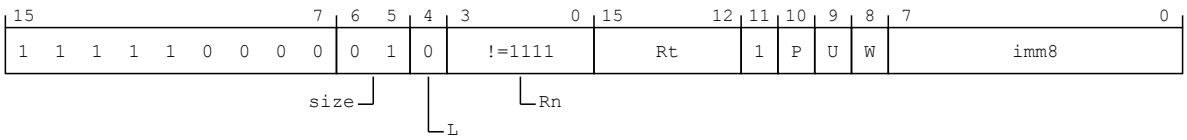

Encoding for the Offset variant

Applies when `(P == 1 && U == 0 && W == 0)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 1)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRHT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>

For the "A1 Offset" variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For the "A1 Post-indexed", "A1 Pre-indexed", "T1", "T2", "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For the "T1" variant: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For the "T2" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

+

Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let address : bits(32) = if index then offset_addr else R(n);
        MemU{16}(address) = R(t)[15:0];
        if wback then R(n) = offset_addr; end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        MemU{16}(address) = R(t)[15:0];
        if wback then R(n) = offset_addr; end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.242 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

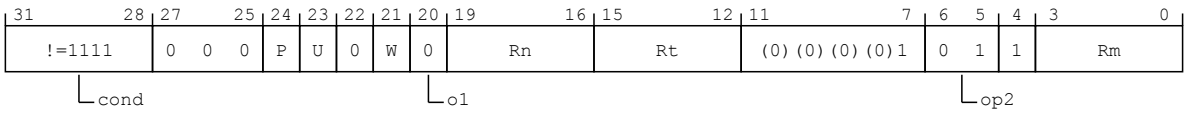

Encoding for the Offset variant

Applies when `(P == 1 && W == 0)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when `(P == 0 && W == 0)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when `(P == 1 && W == 1)`

```
STRH{<c>}{<q>}  <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

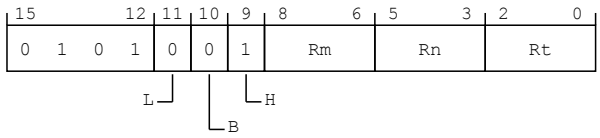

Encoding

```
STRH{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

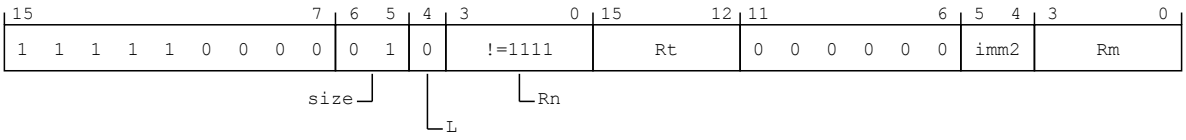

Encoding

```
STRH{<c>}{<q>}  <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
STRH{<c>}.W  <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRType = SRType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{16}(address) = R(t)[15:0];
    if wback then R(n) = offset_addr; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.243 STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

`STRHT` is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

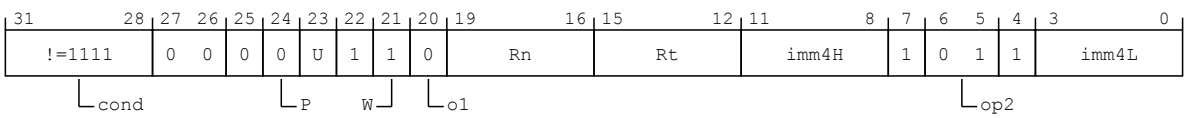

Encoding

```
STRHT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	U	0	1	0	Rn	Rt	(0)(0)(0)(0)	1	0	1	1	Rm
cond				P			W	o1						op2		

Encoding

```
STRHT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12–9	8–7	6–5	4	3–0	15–12	11–8	7–0
1 1 1	1 1 0 0	0 0	0 1	0	!=1111	Rt	1 1 1 0	imm8
			size	L	Rn			

Encoding

```
STRHT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;               // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    MemU_unpriv{16}(address) = R(t)[15:0];
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STRH` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.244 STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

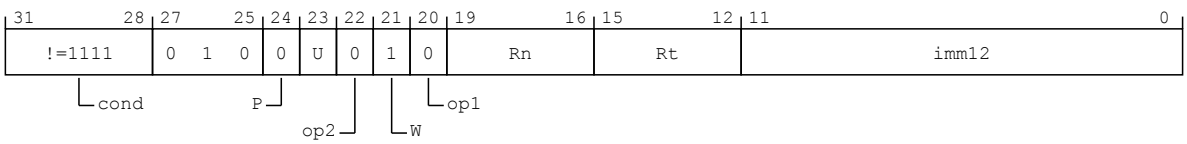

Encoding

```
STRT{<c>}{<q>}  <Rt>, [<Rn>] {, #{+/-}<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
if n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

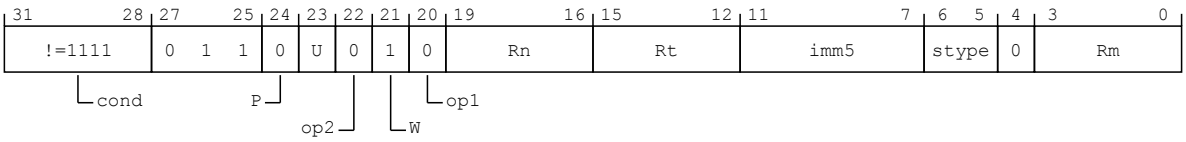

Encoding

```
STRT{<c>}{<q>}  <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

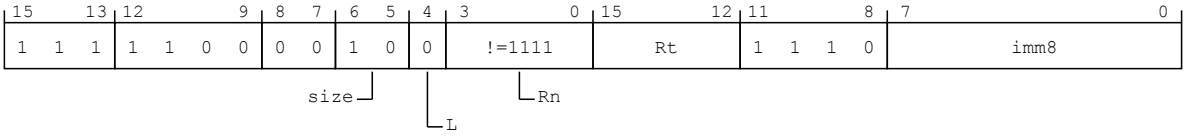

Encoding

```
STRT{<c>}{<q>}  <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRType = ARBITRARY : SRType;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

For the “A1” and “A2” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;                // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                                else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    var data : bits(32);
    if t == 15 then  // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R(t);
    end;
    MemU_unpriv{32}(address) = data;
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STR` (immediate).

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.245 SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is an alias of ADR. This means:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A2) and T32 (T2).

A2

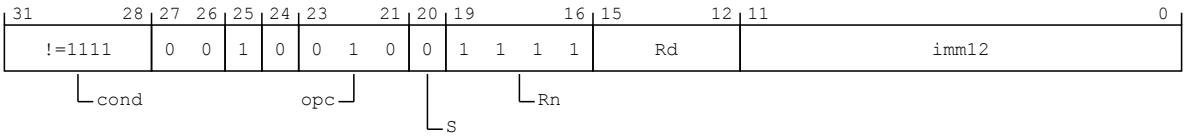

Encoding for the A2 variant

```
SUB{<c>}{<q>}  <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>}  <Rd>, <label>
```

and is the preferred disassembly when `imm12 == '000000000000'`.

T2

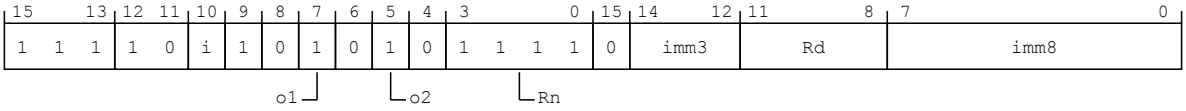

Encoding for the T2 variant

```
SUB{<c>}{<q>}  <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>}  <Rd>, <label>
```

and is the preferred disassembly when `i :: imm3 :: imm8 == '000000000000'`.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rd>

For the “A2” variant: is the general-purpose destination register, encoded in the “Rd” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For the “T2” variant: is the general-purpose destination register, encoded in the “Rd” field.

<const>

An immediate value. See Modified immediate constants in A32 instructions for the range of values.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

The description of ADR gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.246 SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with `<imm8>` set to zero, which is the encoding for the `ERET` instruction, see ERET.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, T4, and T5).

A1

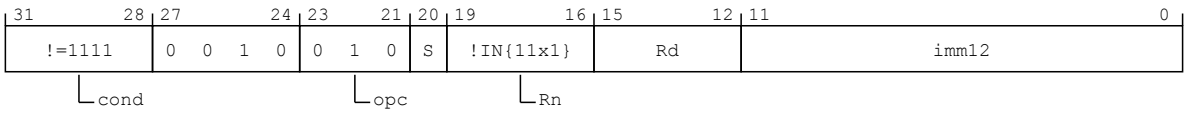

Encoding for the SUB variant

Applies when `(S == 0 && Rn != 11x1)`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Encoding for the SUBS variant

Applies when `(S == 1 && Rn != 1101)`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' && S == '0' then See("ADR"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

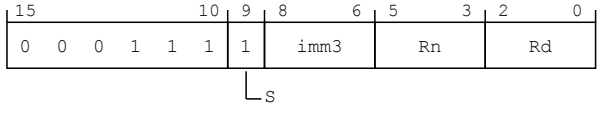

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SUB<c>{<q>}  <Rd>, <Rn>, #<imm3> // (InITBlock())
```

```
SUBS{<q>}  <Rd>, <Rn>, #<imm3> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm3);
```

T2

15 13	12 11	10 8	7 0
0 0 1	1 1	Rdn	imm8
	op		

Encoding

```
SUB<c>{<q>}  <Rdn>, #<imm8> // (InITBlock() && UInt(Rdn) < 8 && UInt(imm8) < 8)
```

```
SUB<c>{<q>}  {<Rdn>, }<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

```
SUBS{<q>}  <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)
```

```
SUBS{<q>}  {<Rdn>, }<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T3

15 11	10	9	8 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	1 1 0 1	S	!=1101	0	imm3	Rd	imm8
			op1		Rn				

Encoding for the SUB variant

Applies when `(S == 0)`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

```
SUB<c>.W  {<Rd>, }<Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1
    ↪or T2)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the SUBS variant

Applies when `(S == 1 && Rd != 1111)`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, #<const>
```

```
SUBS.W  {<Rd>, }<Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
    ↪T2)
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (immediate)"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

T4

15–11	10	9–8	7	6	5	4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	1 0	1	0	1	0	!=11x1	0	imm3	Rd	imm8
			o1		o2		Rn				

Encoding

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
```

```
SUBW{<c>}{<q>}  {<Rd>, }<Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)
```

Decode for this encoding

```
if Rn == '1111' then See("ADR"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T5

15–13	12–11	10	9–6	5–4	3–0	15	14–12	11	10–8	7–0
1 1 1	1 0	0	1 1 1 1	0 1	Rn	1	0(0)0	(1)	(1)(1)(1)	imm8

Encoding

Applies when `(!(Rn == 1110 && imm8 == 00000000))`

```
SUBS{<c>}{<q>}  PC, LR, #<imm8>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if Rn == '1110' && IsZero(imm8) then See("ERET"); end;
let d : integer = 15;
let n : integer = UInt(Rn);
let setflags : boolean = TRUE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
if n != 14 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly SUBS PC. LR and related instructions (A32) and SUBS PC, LR and related instructions (T32).

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 SUB” and “A1 SUBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless `<Rn>` is the LR.

For the “T1”, “T3 SUB”, “T3 SUBS”, and “T4” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 SUB”, “A1 SUBS”, and “T4” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see SUB (SP minus immediate). If the PC is used, see ADR.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

For the “T3 SUB” and “T3 SUBS” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see SUB (SP minus immediate).

<const>

For the “A1 SUB” and “A1 SUBS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T3 SUB” and “T3 SUBS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

<imm3>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “imm3” field.

<Rdn>

Is the general-purpose source and destination register, encoded in the “Rdn” field.

<imm8>

For the “T2” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “T5” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. If `<Rn>` is the LR, and zero is used, see ERET.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

In the T32 instruction set, `MOVS{<c>}{<q>} PC, LR` is a pseudo-instruction for `SUBS{<c>}{<q>} PC, LR, #0`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.247 SUB, SUBS (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

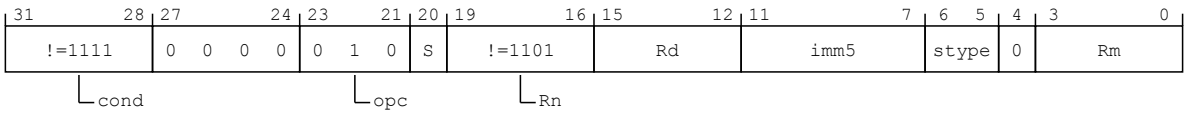

Encoding for the SUB, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SUB, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the SUBS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SUBS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1101' then See("SUB (SP minus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–10	9	8–6	5–3	2–0
0 0 0 1 1 0	1	Rm	Rn	Rd
	S			

Encoding

```
SUB<c>{<q>}  <Rd>, <Rn>, <Rm> // (InITBlock())
```

```
SUBS{<q>}  {<Rd>, }<Rn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```

T2

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	1 1 0 1	S	!=1101	(0)	imm3	Rd	imm2	stype	Rm
	op1		Rn						

Encoding for the SUB, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SUB, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
SUB<c>.W  {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the SUBS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the SUBS, shift or rotate by value variant

Applies when `(S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
SUBS.W  {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (register)"); end;
if Rn == '1101' then See("SUB (SP minus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1”, “T2 SUB, rotate right with extend”, “T2 SUB, shift or rotate by value”, “T2 SUBS, rotate right with extend”, and “T2 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated. If the SP is used, see SUB (SP minus register).

For the “T1” variant: is the first general-purpose source register, encoded in the “Rn” field.

For the “T2 SUB, rotate right with extend”, “T2 SUB, shift or rotate by value”, “T2 SUBS, rotate right with extend”, and “T2 SUBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. If the SP is used, see SUB (SP minus register).

<Rm>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 SUB, rotate right with extend”, “T2 SUB, shift or rotate by value”, “T2 SUBS, rotate right with extend”, and “T2 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 SUB, shift or rotate by value” and “A1 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 SUB, shift or rotate by value” and “T2 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.248 SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

31–28	27–24	23–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0 0 0	0 1 0 (opc)	S	Rn	Rd	Rs	0	stype	1	Rm

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
SUBS{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
SUB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.249 SUB, SUBS (SP minus immediate)

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1

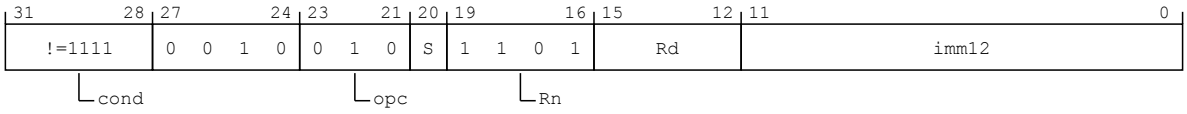

Encoding for the SUB variant

Applies when `(S == 0)`

```
SUB{<c>}{<q>}  {<Rd>, }SP, #<const>
```

Encoding for the SUBS variant

Applies when `(S == 1)`

```
SUBS{<c>}{<q>}  {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

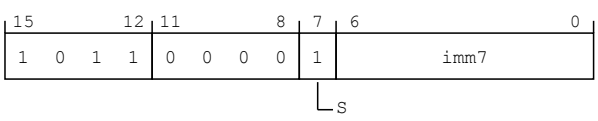

Encoding

```
SUB{<c>}{<q>}  {SP, }SP, #<imm7>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let d : integer = 13;
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm7::'00');
```

T2

15–11	10	9	8–5	4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	0	1 1 0 1 (op1)	S	1 1 0 1 (Rn)	0	imm3	Rd	imm8

Encoding for the SUB variant

Applies when `(S == 0)`

```
SUB{<c>}{<q>}  {<Rd>, }SP, #<const>
```

```
SUB{<c>}.W  {<Rd>, }SP, #<const> // (<Rd>, <const> can be represented in T1)
```

Encoding for the SUBS variant

Applies when `(S == 1 && Rd != 1111)`

```
SUBS{<c>}{<q>}  {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (immediate)"); end;
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if d == 15 && !setflags then UnpredictableProcedure(); end;
```

T3

15–11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7–0
1 1 1 1 0	i	1	0	1 (o1)	0	1 (o2)	0	1 1 0 1 (Rn)	0	imm3	Rd	imm8

Encoding

```
SUB{<c>}{<q>}  {<Rd>, }SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
```

```
SUBW{<c>}{<q>}  {<Rd>, }SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 SUB” and “A1 SUBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless `<Rn>` is the LR.

For the “T2 SUB”, “T2 SUBS”, and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<const>

For the “A1 SUB” and “A1 SUBS” variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T2 SUB” and “T2 SUBS” variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

SP

Is the stack pointer.

<imm7>

Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the “imm7” field as <imm7>/4.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), NOT(imm32), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.250 SUB, SUBS (SP minus register)

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
!=1111	0 0 0 0	0 1 0	S	1 1 0 1	Rd	imm5	stype	0	Rm
cond		opc		Rn					

Encoding for the SUB, rotate right with extend variant

Applies when `(S == 0 && imm5 == 00000 && stype == 11)`

```
SUB{<c>}{<q>}  {<Rd>, }SP, <Rm> , RRX
```

Encoding for the SUB, shift or rotate by value variant

Applies when `(S == 0 && !(imm5 == 00000 && stype == 11))`

```
SUB{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Encoding for the SUBS, rotate right with extend variant

Applies when `(S == 1 && imm5 == 00000 && stype == 11)`

```
SUBS{<c>}{<q>}  {<Rd>, }SP, <Rm> , RRX
```

Encoding for the SUBS, shift or rotate by value variant

Applies when `(S == 1 && !(imm5 == 00000 && stype == 11))`

```
SUBS{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	1 1 0 1 (op1)	S	1 1 0 1 (Rn)	(0)	imm3	Rd	imm2	stype	Rm

Encoding for the SUB, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
SUB{<c>}{<q>}  {<Rd>, }SP, <Rm>, RRX
```

Encoding for the SUB, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
SUB{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>} // (Preferred syntax)
```

```
SUB{<c>}.W  {<Rd>, }SP, <Rm>
```

Encoding for the SUBS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)`

```
SUBS{<c>}{<q>}  {<Rd>, }SP, <Rm>, RRX
```

Encoding for the SUBS, shift or rotate by value variant

Applies when `(S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
SUBS{<c>}{<q>}  {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (register)"); end;
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For the “T1 SUB, rotate right with extend”, “T1 SUB, shift or rotate by value”, “T1 SUBS, rotate right with extend”, and “T1 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<Rm>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1 SUB, rotate right with extend”, “T1 SUB, shift or rotate by value”, “T1 SUBS, rotate right with extend”, and “T1 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 SUB, shift or rotate by value” and “A1 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T1 SUB, shift or rotate by value” and “T1 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.251 SVC

Supervisor Call causes a Supervisor Call exception. For more information, see Supervisor Call (SVC) exception.

Note

`SVC` was previously called `SWI`, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

In the following cases, the Supervisor Call exception generated by the `SVC` instruction is taken to Hyp mode:

- If the `SVC` is executed in Hyp mode.
- If HCR.TGE is set to 1, and the `SVC` is executed in Non-secure User mode. For more information, see Supervisor Call exception, when HCR.TGE is set to 1

In these cases, the HSR, Hyp Syndrome Register identifies that the exception entry was caused by a Supervisor Call exception, EC value `0x11`, see Use of the HSR. The immediate field in the HSR:

- If the `SVC` is unconditional:
  - For the T32 instruction, is the zero-extended value of the `imm8` field.
  - For the A32 instruction, is the least-significant 16 bits the `imm24` field.
- If the `SVC` is conditional, is UNKNOWN.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

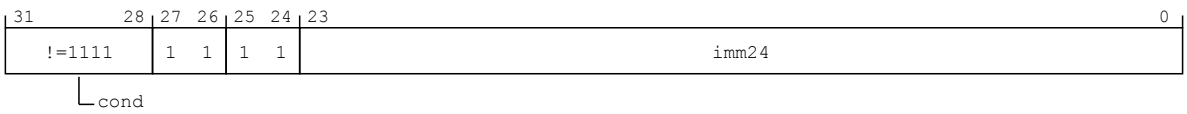

Encoding

```
SVC{<c>}{<q>}  {#}<imm>
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm24);
```

T1

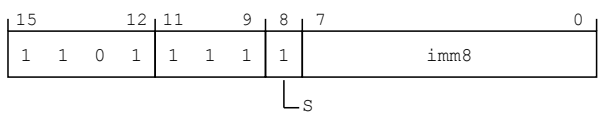

Encoding

```
SVC{<c>}{<q>}  {#}<imm>
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm8);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<imm>

For the “A1” variant: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the “imm24” field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service.

For the “T1” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_CheckForSVCTrap(imm32[15:0]);
    AArch32_CallSupervisor(imm32[15:0]);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.252 SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

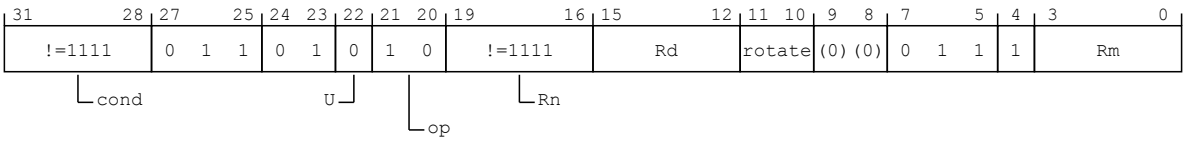

Encoding

```
SXTAB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

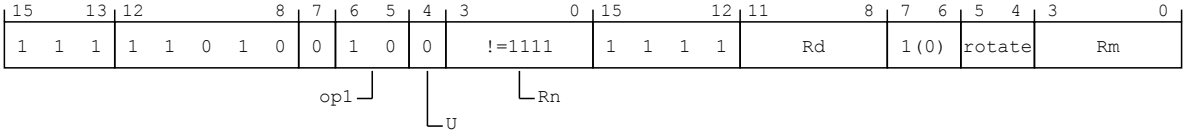

Encoding

```
SXTAB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + SignExtend{32}(rotated[7:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.253 SXTAB16

Signed Extend and Add Byte 16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

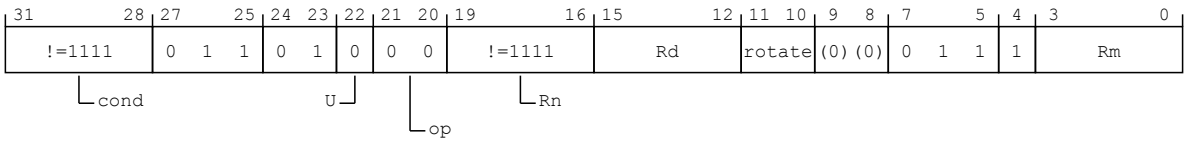

Encoding

```
SXTAB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

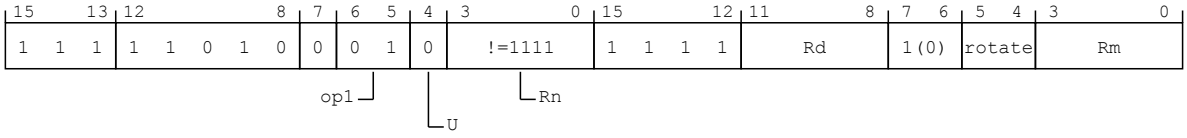

Encoding

```
SXTAB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0]  = R(n)[15:0] + SignExtend{16}(rotated[7:0]);
    R(d)[31:16] = R(n)[31:16] + SignExtend{16}(rotated[23:16]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.254 SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

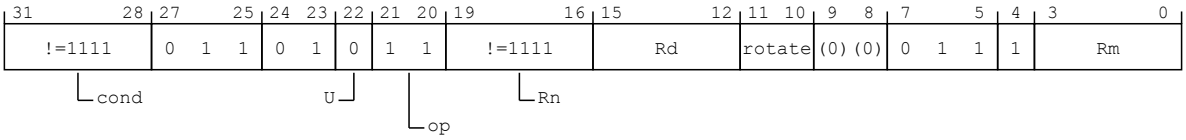

Encoding

```
SXTAH{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

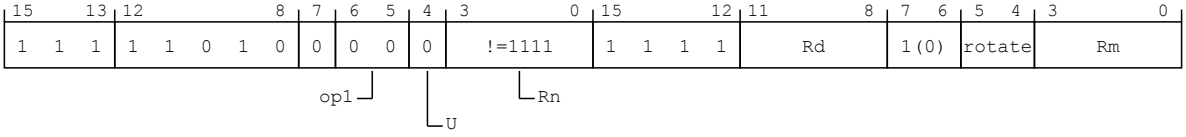

Encoding

```
SXTAH{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + SignExtend{32}(rotated[15:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.255 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

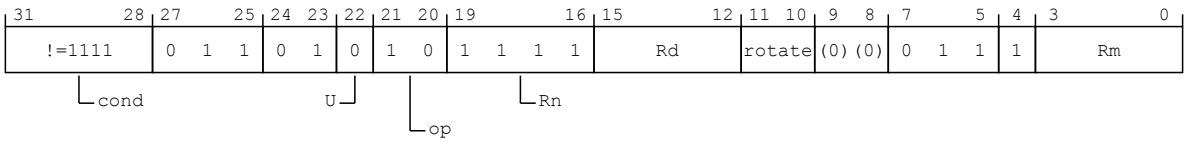

Encoding

```
SXTB{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

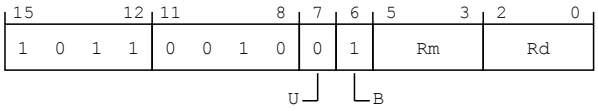

Encoding

```
SXTB{<c>}{<q>}  {<Rd>, }<Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2

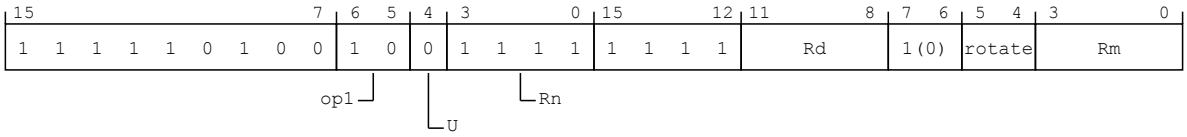

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SXTB{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

```
SXTB{<c>}.W  {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = SignExtend{32}(rotated[7:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.256 SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

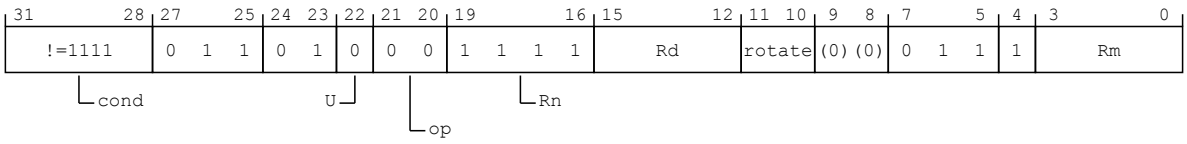

Encoding

```
SXTB16{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

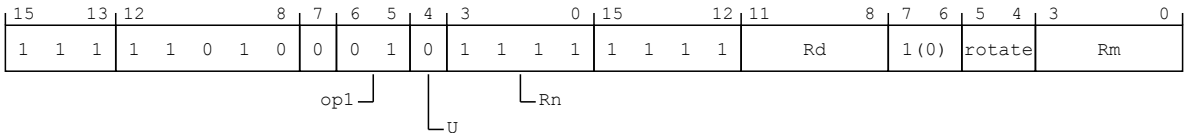

Encoding

```
SXTB16{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0]  = SignExtend{16}(rotated[7:0]);
    R(d)[31:16] = SignExtend{16}(rotated[23:16]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.257 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

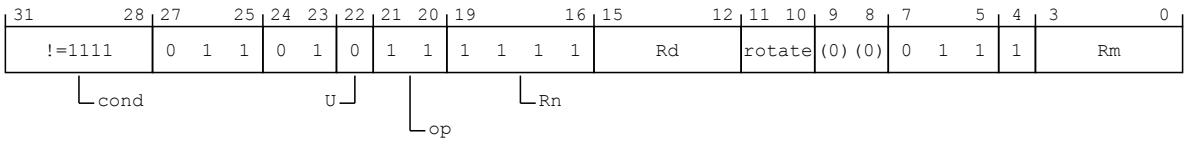

Encoding

```
SXTH{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

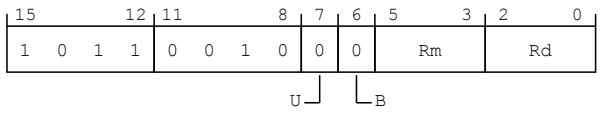

Encoding

```
SXTH{<c>}{<q>}  {<Rd>, }<Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2

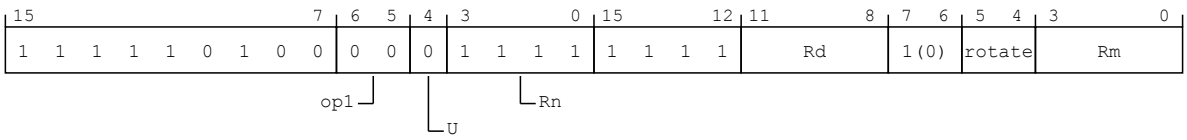

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SXTH{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

```
SXTH{<c>}.W  {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = SignExtend{32}(rotated[15:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.258 TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

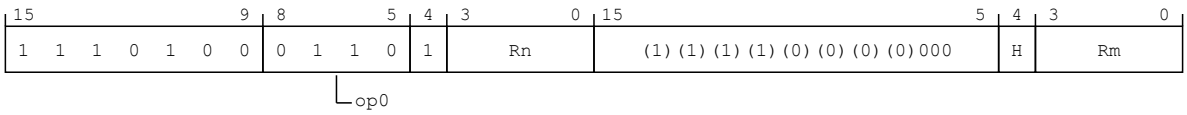

Encoding for the Byte variant

Applies when `(H == 0)`

```
TBB{<c>}{<q>}  [<Rn>, <Rm>] // (Outside or last in IT block)
```

Encoding for the Halfword variant

Applies when `(H == 1)`

```
TBH{<c>}{<q>}  [<Rn>, <Rm>, LSL #1] // (Outside or last in IT block)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let is_tbh : boolean = (H == '1');
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register holding the address of the table of branch lengths, encoded in the “Rn” field. The PC can be used. If it is, the table immediately follows this instruction.

<Rm>

For the “Byte” variant: is the general-purpose index register, encoded in the “Rm” field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For the “Halfword” variant: is the general-purpose index register, encoded in the “Rm” field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var halfwords : integer;
    if is_tbh then
        halfwords = UInt(MemU{16}(R(n)+LSL(R(m), 1)));
    else
        halfwords = UInt(MemU{8}(R(n)+R(m)));
    end;
    BranchWritePC(PC32() + 2*halfwords, BranchType_INDIR);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.259 TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111 (cond)	0	0	1	1	0	0	1 (opc)	1	Rn	(0)(0)(0)(0)	imm12

Encoding

```
TEQ{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15–13	12	11	10	9	8–5	4	3–0	15	14–12	11–8	7–0
1 1 1	1	0	i	0	0 1 0 0 (op1)	1 (S)	Rn	0	imm3	1 1 1 1 (Rd)	imm8

Encoding

```
TEQ{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1” variant: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) XOR imm32;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.260 TEQ (register)

Test Equivalence (register) performs a bitwise exclusive-OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

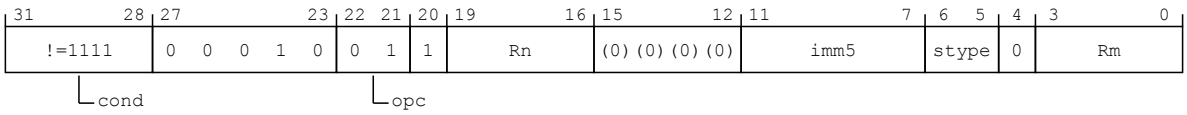

Encoding for the Rotate right with extend variant

Applies when `(imm5 == 00000 && stype == 11)`

```
TEQ{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm5 == 00000 && stype == 11))`

```
TEQ{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```

T1

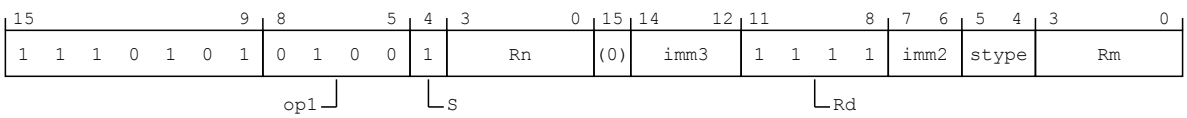

Encoding for the Rotate right with extend variant

Applies when `(imm3 == 000 && imm2 == 00 && stype == 11)`

```
TEQ{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00 && stype == 11))`

```
TEQ{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 Rotate right with extend” and “T1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1 Rotate right with extend” and “T1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.261 TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive-OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

31–28	27–26	25	24–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0	0	1 0	0 1 (opc)	1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

Encoding

```
TEQ{<c>}{<q>}  <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.262 TSB

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions, see Trace Synchronization Barrier (TSB).

If FEAT_TRF is not implemented, this instruction executes as a `NOP`.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_TRF)

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11–0
!=1111	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)(0)(0)(0)00010010
cond						R			imm4								imm12

Encoding

```
TSB{<c>}{<q>}  CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then ExecuteAsNOP(); end;
if cond != '1110' then UnpredictableProcedure(); end; // TSB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

(FEAT_TRF)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	1	0
																								hint						option	

Encoding

```
TSB{<c>}{<q>}  CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then ExecuteAsNOP(); end;
if InITBlock() then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    TraceSynchronizationBarrier();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.263 TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!=1111	0	0	1	1	0	0	0	1	Rn	(0)(0)(0)(0)	imm12
cond						opc					

Encoding

```
TST{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15–13	12	11	10	9	8–5	4	3–0	15	14–12	11–8	7–0
1 1 1	1	0	i	0	0 0 0 0	1	Rn	0	imm3	1 1 1 1	imm8
					op1	S				Rd	

Encoding

```
TST{<c>}{<q>}  <Rn>, #<const>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For the “T1” variant: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                                   else T32ExpandImm_C(imm, PSTATE.C));
    let result : bits(32) = R(n) AND imm32;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.264 TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

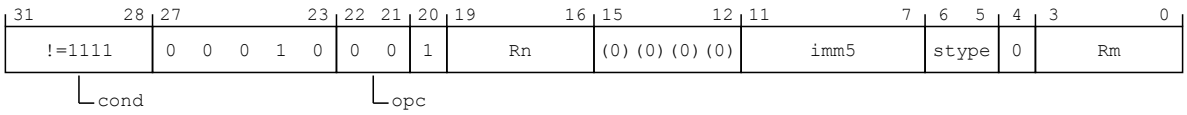

Encoding for the Rotate right with extend variant

Applies when `(imm5 == 00000 && stype == 11)`

```
TST{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm5 == 00000 && stype == 11))`

```
TST{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
```

T1

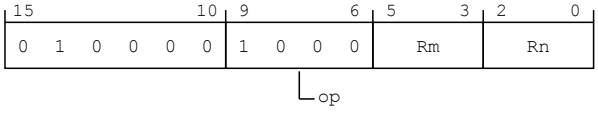

Encoding

```
TST{<c>}{<q>}  <Rn>, <Rm>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRType = SRType_LSL;
let shift_n : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	0 0 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	stype	Rm
	op1	S				Rd			

Encoding for the Rotate right with extend variant

Applies when `(imm3 == 000 && imm2 == 00 && stype == 11)`

```
TST{<c>}{<q>}  <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00 && stype == 11))`

```
TST{<c>}{<q>}  <Rn>, <Rm> {, <shift> #<amount>}
```

```
TST{<c>}.W  <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.265 TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

31–28	27–26	25	24–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!=1111 (cond)	0 0	0	1 0	0 0 (opc)	1	Rn	(0)(0)(0)(0)	Rs	0	stype	1	Rm

Encoding

```
TST{<c>}{<q>}  <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.266 UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

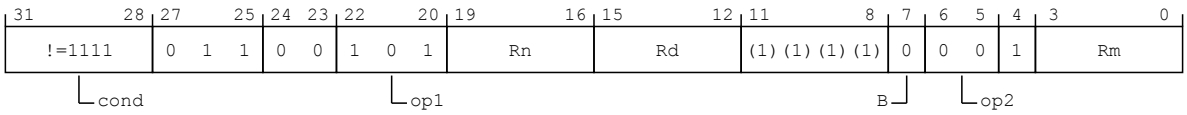

Encoding

```
UADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

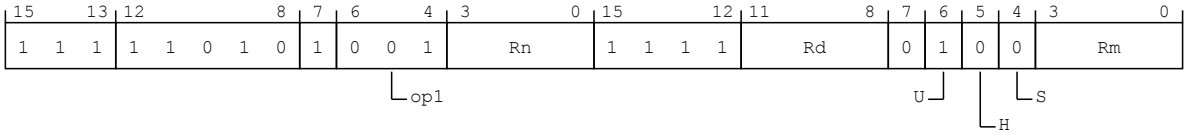

Encoding

```
UADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0]  = sum1[15:0];
    R(d)[31:16] = sum2[15:0];
    PSTATE.GE[1:0] = if sum1 >= 0x10000 then '11' else '00';
    PSTATE.GE[3:2] = if sum2 >= 0x10000 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.267 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

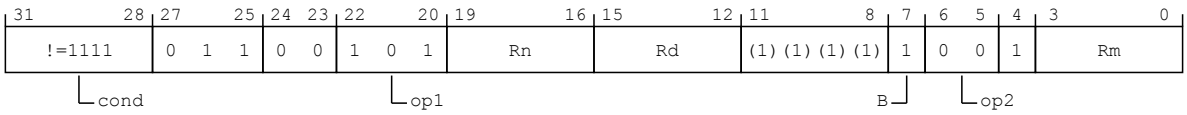

Encoding

```
UADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

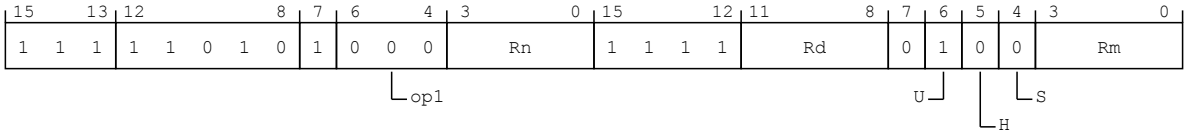

Encoding

```
UADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m  == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0]   = sum1[7:0];
    R(d)[15:8]  = sum2[7:0];
    R(d)[23:16] = sum3[7:0];
    R(d)[31:24] = sum4[7:0];
    PSTATE.GE[0]  = if sum1 >= 0x100 then '1' else '0';
    PSTATE.GE[1]  = if sum2 >= 0x100 then '1' else '0';
    PSTATE.GE[2]  = if sum3 >= 0x100 then '1' else '0';
    PSTATE.GE[3]  = if sum4 >= 0x100 then '1' else '0';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.268 UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

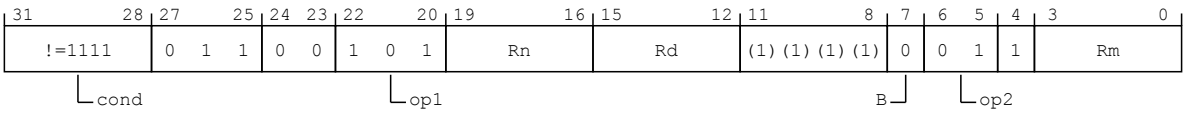

Encoding

```
UASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

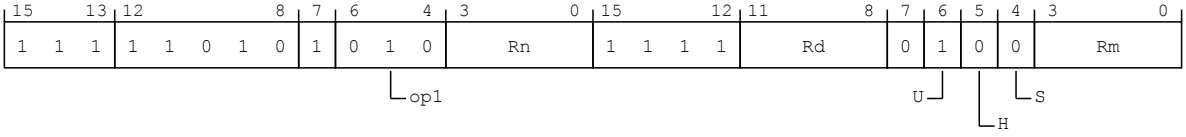

Encoding

```
UASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer  = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0]  = diff[15:0];
    R(d)[31:16] = sum[15:0];
    PSTATE.GE[1:0] = if diff >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum  >= 0x10000 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.269 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22	21	20–16	15–12	11–7	6–5	4	3–0
!=1111	0 1 1	1 1	1	1	widthm1	Rd	lsb	1 0	1	Rn
cond			U							

Encoding

```
UBFX{<c>}{<q>}  <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(lsb);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15–13	12–11	10	9	8	7–5	4	3–0	15	14–12	11–8	7–6	5	4–0
1 1 1	1 0	(0)	1	1	1 1 0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1
					op1								

Encoding

```
UBFX{<c>}{<q>}  <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(imm3::imm2);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “A1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the width of the field, in the range 1 to 32-<lsb>, encoded in the “widthm1” field as <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d) = ZeroExtend{32}(R(n)[msbit:lsbit]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.270 UDF

Permanently Undefined generates an Undefined Instruction exception.

The encodings for UDF used in this section are defined as permanently UNDEFINED. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

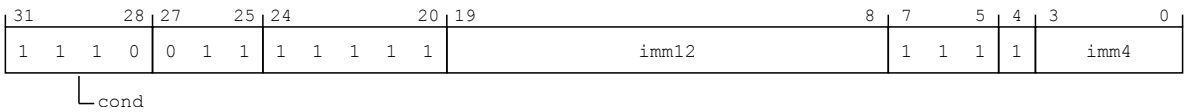

Encoding

```
UDF{<c>}{<q>}  {#}<imm>
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12::imm4);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T1

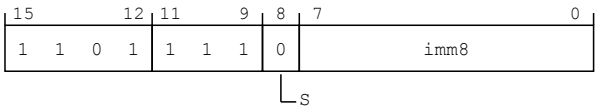

Encoding

```
UDF{<c>}{<q>}  {#}<imm>
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm8);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T2

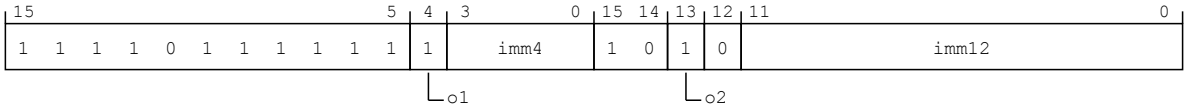

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
UDF{<c>}{<q>}  {#}<imm>
```

```
UDF{<c>}.W  {#}<imm> // (<imm> can be represented in T1)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm4::imm12);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. `<c>` must be `AL` or omitted.

For the "T1" and "T2" variants: see Standard assembler syntax fields. Arm deprecates using any `<c>` value other than `AL`.

<q>

See Standard assembler syntax fields.

<imm>

For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. The PE ignores the value of this constant.

For the "T1" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.

For the "T2" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.271 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

See Divide instructions for more information about this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22–20	19–16	15–12	11–8	7–5	4	3–0
!=1111	0 1 1	1 0	0 1 1	Rd	(1) (1) (1) (1)	Rm	0 0 0	1	Rn
cond			op1		Ra		op2		

Encoding

```
UDIV{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

T1

15–13	12–9	8–7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1 1	0 1 1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm
			op1		Ra		op2	

Encoding

```
UDIV{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the dividend, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the divisor, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let dividend : integer = UInt(R(n));
    let divisor : integer  = UInt(R(m));
    var result : integer;
    if divisor == 0 then
        result = 0;
    else
        result = dividend DIVRM divisor;
    end;
    R(d) = result[31:0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.272 UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

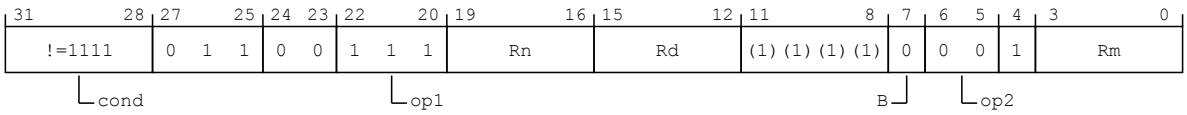

Encoding

```
UHADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

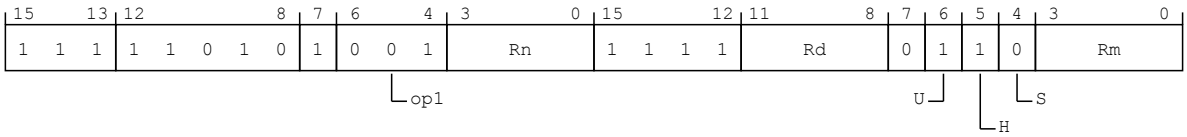

Encoding

```
UHADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0]  = sum1[16:1];
    R(d)[31:16] = sum2[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.273 UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

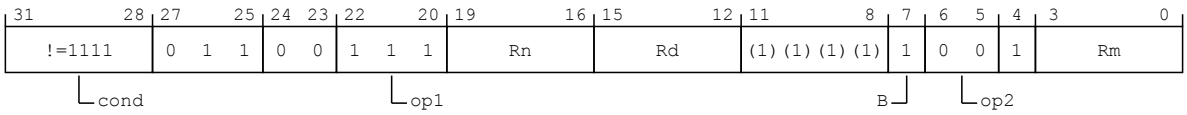

Encoding

```
UHADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

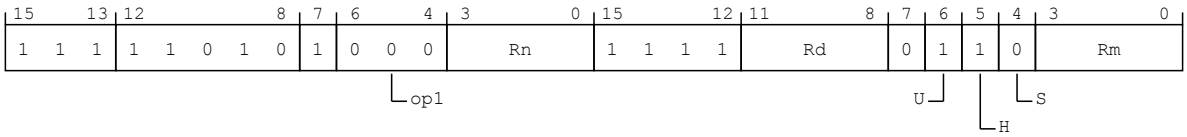

Encoding

```
UHADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0]   = sum1[8:1];
    R(d)[15:8]  = sum2[8:1];
    R(d)[23:16] = sum3[8:1];
    R(d)[31:24] = sum4[8:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.274 UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

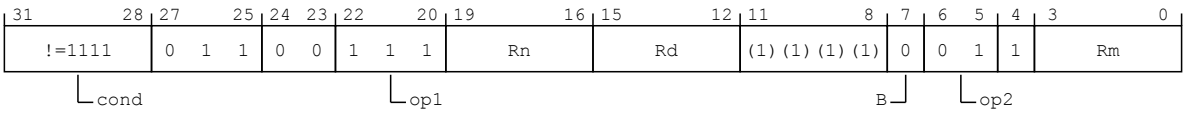

Encoding

```
UHASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

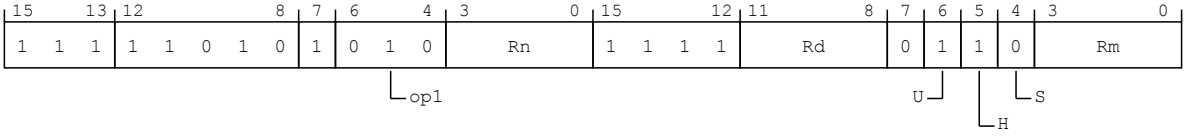

Encoding

```
UHASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer  = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0]  = diff[16:1];
    R(d)[31:16] = sum[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.275 UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

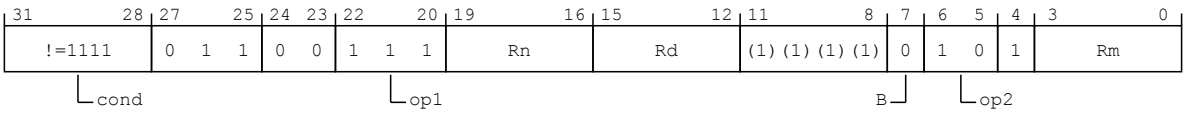

Encoding

```
UHSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

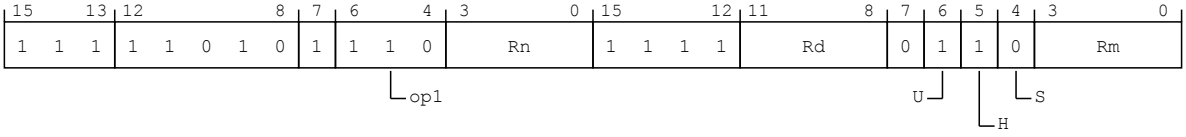

Encoding

```
UHSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer  = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0]  = sum[16:1];
    R(d)[31:16] = diff[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.276 UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

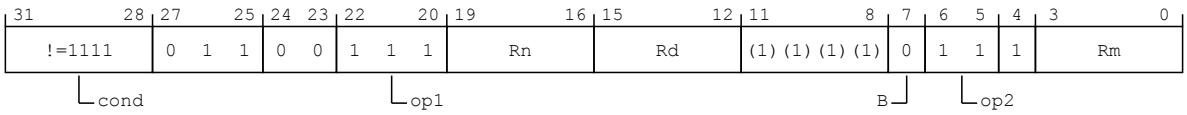

Encoding

```
UHSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

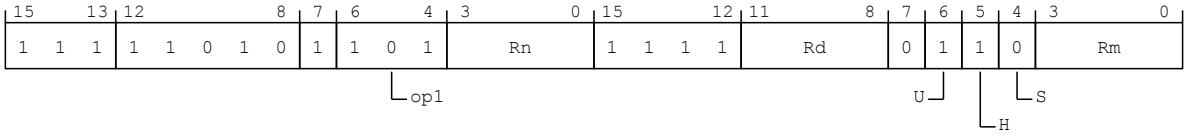

Encoding

```
UHSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0]  = diff1[16:1];
    R(d)[31:16] = diff2[16:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.277 UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

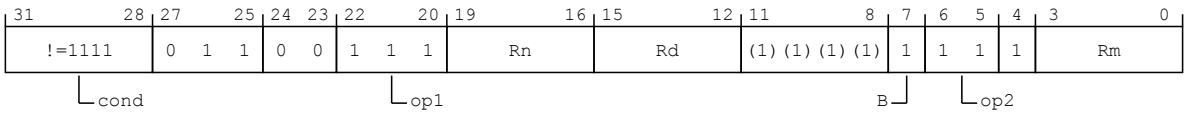

Encoding

```
UHSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

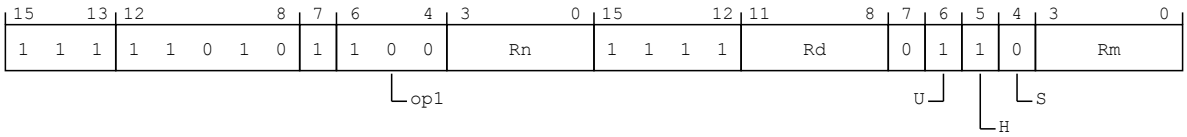

Encoding

```
UHSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0]   = diff1[8:1];
    R(d)[15:8]  = diff2[8:1];
    R(d)[23:16] = diff3[8:1];
    R(d)[31:24] = diff4[8:1];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.278 UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!=1111	0	0	0	0	0	1	0	0	RdHi	RdLo	Rm	1	0	0	1	Rn
cond					opc			S								

Encoding

```
UMAAL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–13	12–9	8	7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1	1	1 1 0	Rn	RdLo	RdHi	0 1 1 0	Rm
				op1				op2	

Encoding

```
UMAAL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m)) + UInt(R(dHi)) + UInt(R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.279 UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–24	23–21	20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 0 1 (opc)	S	RdHi	RdLo	Rm	1 0 0 1	Rn

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
UMLALS{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
UMLAL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–13	12–9	8 7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1 1	1 1 0 (op1)	Rn	RdLo	RdHi	0 0 0 0 (op2)	Rm

Encoding

```
UMLAL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m)) + UInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.280 UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–24	23–21	20	19–16	15–12	11–8	7–4	3–0
!=1111 (cond)	0 0 0 0	1 0 0 (opc)	S	RdHi	RdLo	Rm	1 0 0 1	Rn

Encoding for the Flag setting variant

Applies when `(S == 1)`

```
UMULLS{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the Not flag setting variant

Applies when `(S == 0)`

```
UMULL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1

15–13	12–9	8–7	6–4	3–0	15–12	11–8	7–4	3–0
1 1 1	1 1 0 1	1 1	0 1 0 (op1)	Rn	RdLo	RdHi	0 0 0 0 (op2)	Rm

Encoding

```
UMULL{<c>}{<q>}  <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<RdLo>

Is the general-purpose destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.281 UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

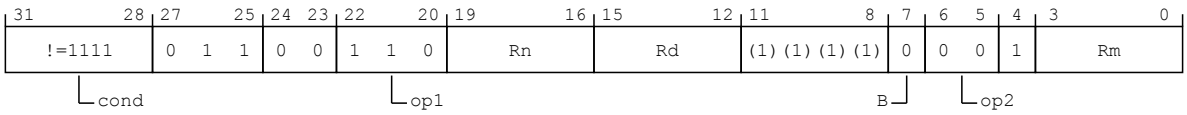

Encoding

```
UQADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

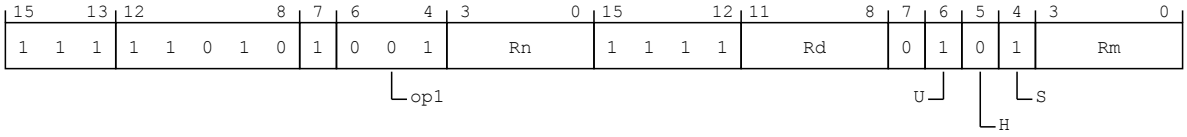

Encoding

```
UQADD16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0]  = UnsignedSat{16}(sum1);
    R(d)[31:16] = UnsignedSat{16}(sum2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.282 UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 2^8 - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

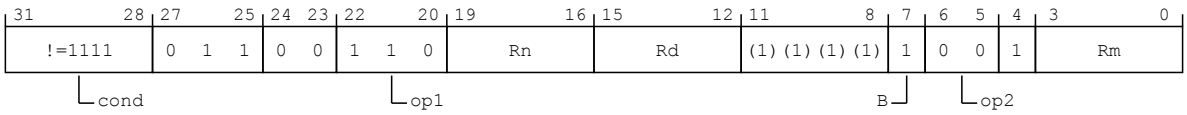

Encoding

```
UQADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

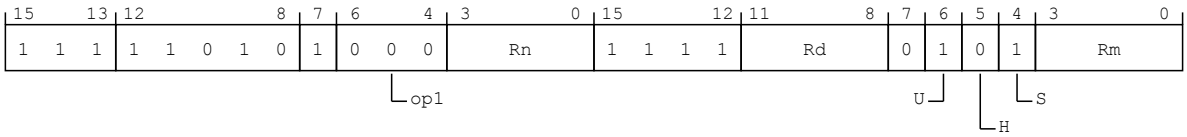

Encoding

```
UQADD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0]   = UnsignedSat{8}(sum1);
    R(d)[15:8]  = UnsignedSat{8}(sum2);
    R(d)[23:16] = UnsignedSat{8}(sum3);
    R(d)[31:24] = UnsignedSat{8}(sum4);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.283 UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

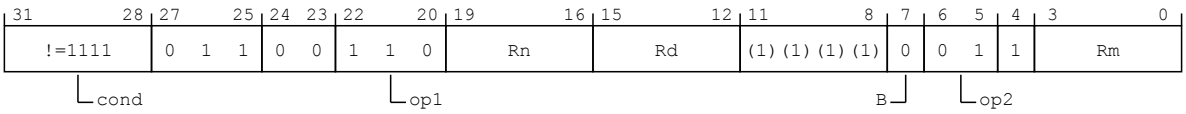

Encoding

```
UQASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

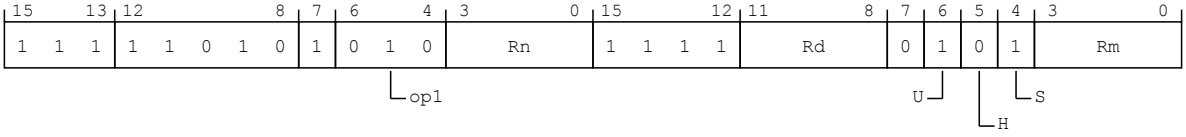

Encoding

```
UQASX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer  = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0]  = UnsignedSat{16}(diff);
    R(d)[31:16] = UnsignedSat{16}(sum);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.284 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

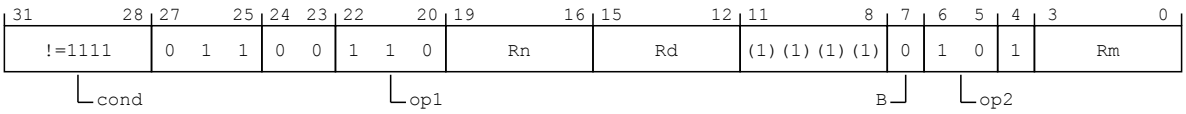

Encoding

```
UQSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

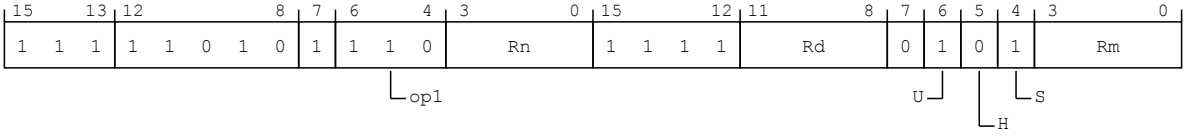

Encoding

```
UQSAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer  = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0]  = UnsignedSat{16}(sum);
    R(d)[31:16] = UnsignedSat{16}(diff);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.285 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

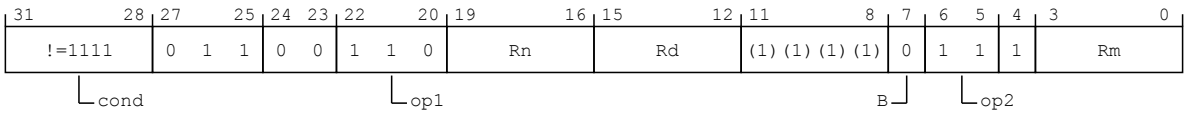

Encoding

```
UQSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

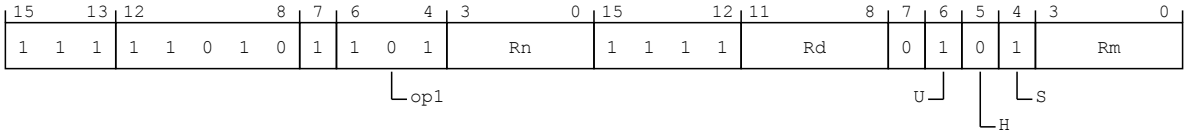

Encoding

```
UQSUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0]  = UnsignedSat{16}(diff1);
    R(d)[31:16] = UnsignedSat{16}(diff2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.286 UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 2^8 - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

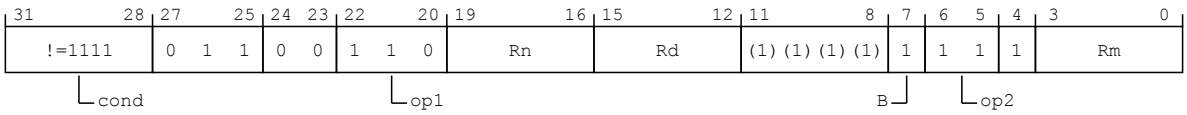

Encoding

```
UQSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

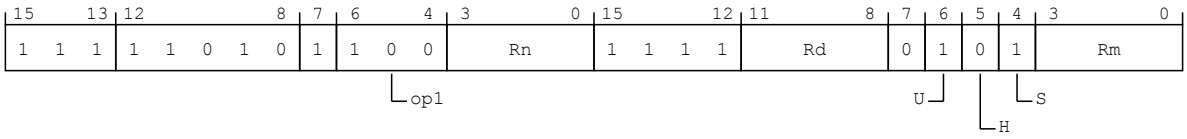

Encoding

```
UQSUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0]   = UnsignedSat{8}(diff1);
    R(d)[15:8]  = UnsignedSat{8}(diff2);
    R(d)[23:16] = UnsignedSat{8}(diff3);
    R(d)[31:24] = UnsignedSat{8}(diff4);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.287 USAD8

Unsigned Sum of Absolute Differences

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–20	19–16	15–12	11–8	7–5	4	3–0
!=1111 (cond)	0 1 1	1 1 0 0 0	Rd	1 1 1 1 (Ra)	Rm	0 0 0	1	Rn

Encoding

```
USAD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–7	6–4	3–0	15–12	11–8	7 6	5 4	3–0
1 1 1	1 1 0 1 1 0	1 1 1 (op1)	Rn	1 1 1 1 (Ra)	Rd	0 0	0 0 (op2)	Rm

Encoding

```
USAD8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let absdiff1 : integer = Abs(UInt(R(n)[7:0])   - UInt(R(m)[7:0]));
    let absdiff2 : integer = Abs(UInt(R(n)[15:8])  - UInt(R(m)[15:8]));
    let absdiff3 : integer = Abs(UInt(R(n)[23:16]) - UInt(R(m)[23:16]));
    let absdiff4 : integer = Abs(UInt(R(n)[31:24]) - UInt(R(m)[31:24]));
    let result : integer = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R(d) = result[31:0];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.288 USADA8

Unsigned Sum of Absolute Differences and Accumulate

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–20	19–16	15–12	11–8	7–5	4	3–0
!=1111 (cond)	0 1 1	1 1 0 0 0	Rd	!=1111 (Ra)	Rm	0 0 0	1	Rn

Encoding

```
USADA8{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then See("USAD8"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–7	6–4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1	1 1 0 1 1 0	1 1 1 (op1)	Rn	!=1111 (Ra)	Rd	0 0	0 0 (op2)	Rm

Encoding

```
USADA8{<c>}{<q>}  <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then See("USAD8"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let absdiff1 : integer = Abs(UInt(R(n)[7:0])   - UInt(R(m)[7:0]));
    let absdiff2 : integer = Abs(UInt(R(n)[15:8])  - UInt(R(m)[15:8]));
    let absdiff3 : integer = Abs(UInt(R(n)[23:16]) - UInt(R(m)[23:16]));
    let absdiff4 : integer = Abs(UInt(R(n)[31:24]) - UInt(R(m)[31:24]));
    let result : integer = UInt(R(a)) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R(d) = result[31:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.289 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–23	22	21	20–16	15–12	11–7	6	5	4	3–0
!=1111 (cond)	0 1 1 0 1	1 (U)	1	sat_imm	Rd	imm5	sh	0	1	Rn

Encoding for the Arithmetic shift right variant

Applies when `(sh == 1)`

```
USAT{<c>}{<q>}  <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when `(sh == 0)`

```
USAT{<c>}{<q>}  <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(sh::'0', imm5);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15–6	5	4	3–0	15	14–12	11–8	7 6	5	4–0
11110(0)1110	sh	0	Rn	0	imm3	Rd	imm2	(0)	sat_imm

Encoding for the Arithmetic shift right variant

Applies when `(sh == 1 && !(imm3 == 000 && imm2 == 00))`

```
USAT{<c>}{<q>}  <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when `(sh == 0)`

```
USAT{<c>}{<q>}  <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if sh == '1' && (imm3::imm2) == '00000' then See("USAT16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(sh::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 0 to 31, encoded in the “sat_imm” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<amount>

For the “A1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 32 encoded in the “imm5” field as <amount> modulo 32.

For the “A1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 31 encoded in the “imm3:imm2” field as <amount>.

For the “T1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm3:imm2” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand : bits(32) = Shift{}(R(n), shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    let (result, sat) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(SInt(operand));
    R(d) = ZeroExtend{32}(result);
    if sat then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.290 USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24–23	22	21–20	19–16	15–12	11–8	7–5	4	3–0
!=1111 (cond)	0 1 1	0 1	1 (U)	1 0	sat_imm	Rd	(1)(1)(1)(1)	0 0 1	1	Rn

Encoding

```
USAT16{<c>}{<q>}  <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12–11	10	9	8	7–5	4	3–0	15	14–12	11–8	7–6	5	4	3–0
1 1 1	1 0	(0)	1	1	1 0 1 (op1)	0	Rn	0	0 0 0 (imm3)	Rd	0 0 (imm2)	(0)	(0)	sat_imm

Encoding

```
USAT16{<c>}{<q>}  <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 0 to 15, encoded in the “sat_imm” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 = SInt(R(n)[15:0]);
    let operand2 = SInt(R(n)[31:16]);
    let (result1, sat1) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(operand1);
    let (result2, sat2) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(operand2);
    R(d)[15:0] = ZeroExtend{16}(result1);
    R(d)[31:16] = ZeroExtend{16}(result2);
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.291 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

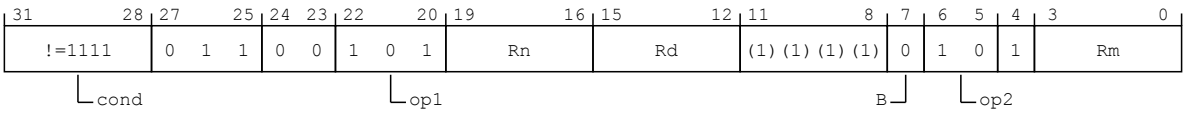

Encoding

```
USAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

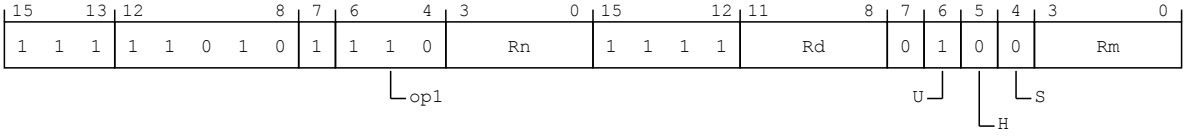

Encoding

```
USAX{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer  = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0]  = sum[15:0];
    R(d)[31:16] = diff[15:0];
    PSTATE.GE[1:0] = if sum  >= 0x10000 then '11' else '00';
    PSTATE.GE[3:2] = if diff >= 0 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.292 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

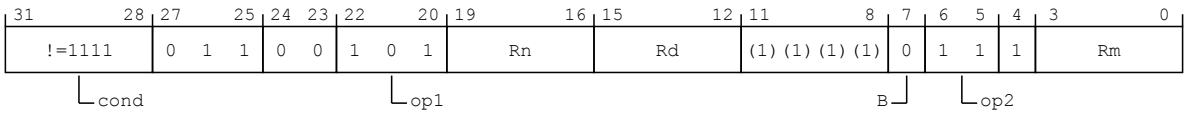

Encoding

```
USUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

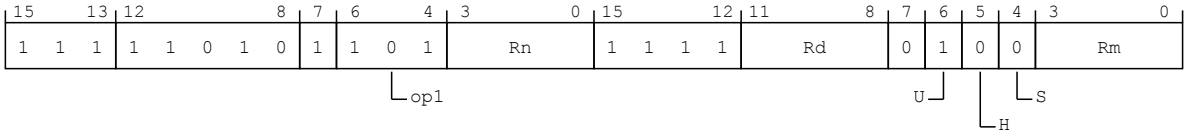

Encoding

```
USUB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0]  = diff1[15:0];
    R(d)[31:16] = diff2[15:0];
    PSTATE.GE[1:0] = if diff1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff2 >= 0 then '11' else '00';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.293 USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

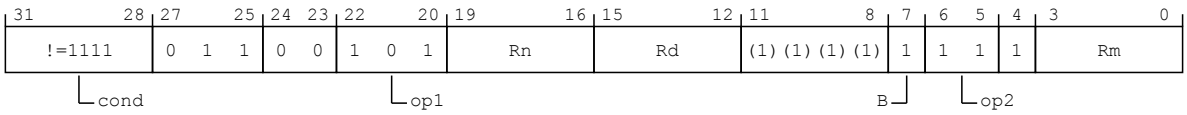

Encoding

```
USUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

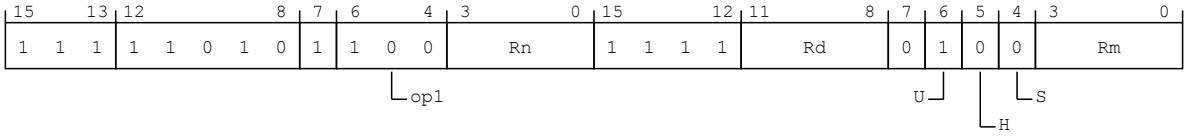

Encoding

```
USUB8{<c>}{<q>}  {<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0]   = diff1[7:0];
    R(d)[15:8]  = diff2[7:0];
    R(d)[23:16] = diff3[7:0];
    R(d)[31:24] = diff4[7:0];
    PSTATE.GE[0]  = if diff1 >= 0 then '1' else '0';
    PSTATE.GE[1]  = if diff2 >= 0 then '1' else '0';
    PSTATE.GE[2]  = if diff3 >= 0 then '1' else '0';
    PSTATE.GE[3]  = if diff4 >= 0 then '1' else '0';
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.294 UXTAB

Unsigned Extend and Add Byte

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

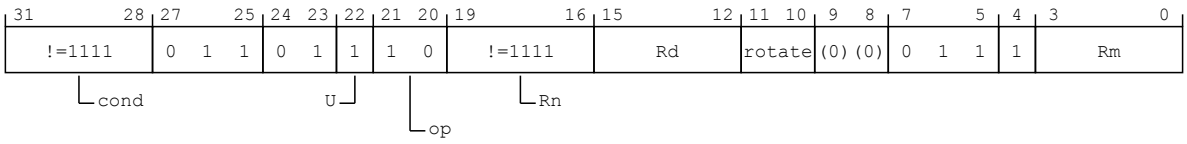

Encoding

```
UXTAB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

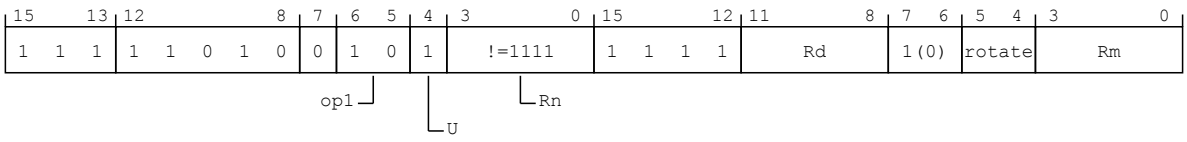

Encoding

```
UXTAB{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + ZeroExtend{32}(rotated[7:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.295 UXTAB16

Unsigned Extend and Add Byte 16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

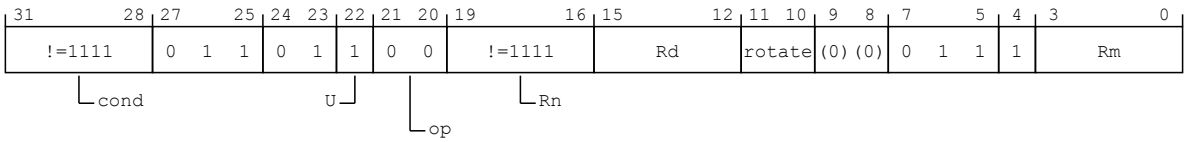

Encoding

```
UXTAB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

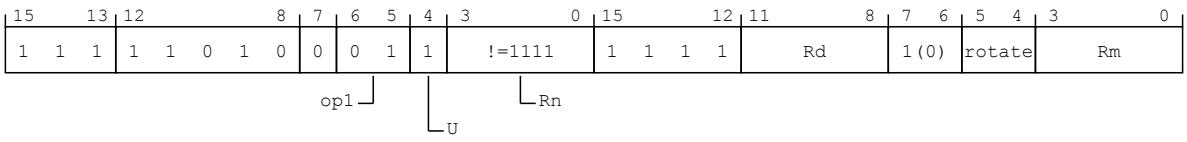

Encoding

```
UXTAB16{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0]  = R(n)[15:0] + ZeroExtend{16}(rotated[7:0]);
    R(d)[31:16] = R(n)[31:16] + ZeroExtend{16}(rotated[23:16]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.296 UXTAH

Unsigned Extend and Add Halfword

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

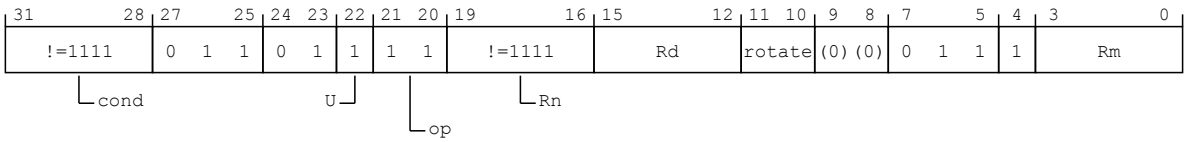

Encoding

```
UXTAH{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

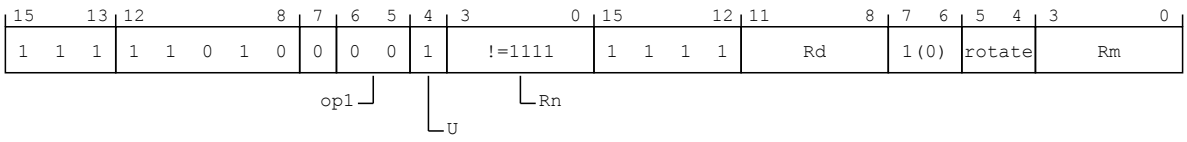

Encoding

```
UXTAH{<c>}{<q>}  {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + ZeroExtend{32}(rotated[15:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.297 UXTB

Unsigned Extend Byte

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

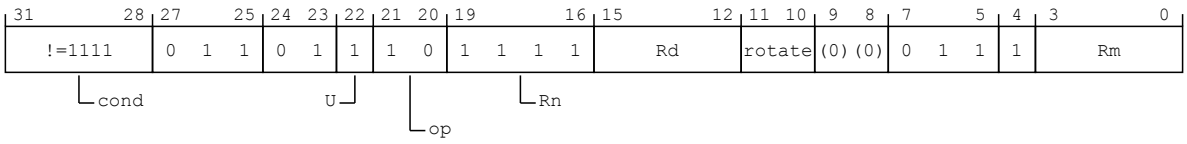

Encoding

```
UXTB{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

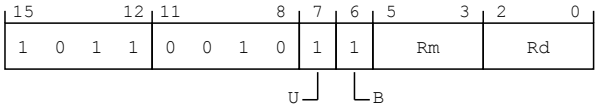

Encoding

```
UXTB{<c>}{<q>}  {<Rd>, }<Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

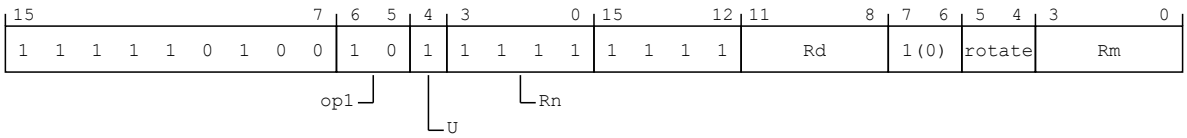

Encoding

```
UXTB{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

```
UXTB{<c>}.W  {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = ZeroExtend{32}(rotated[7:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.298 UXTB16

Unsigned Extend Byte 16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

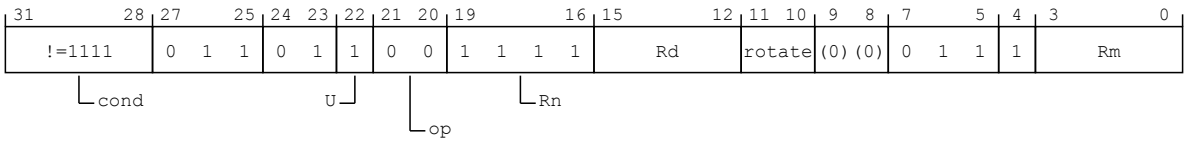

Encoding

```
UXTB16{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

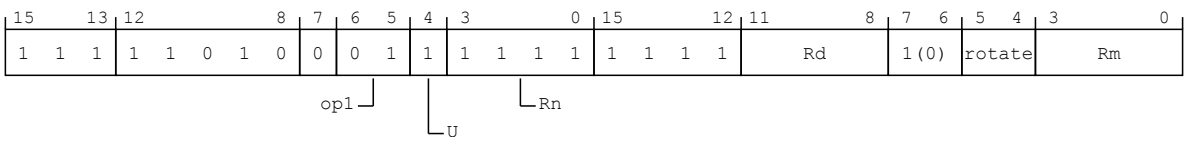

Encoding

```
UXTB16{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” variant: is the general-purpose source register, encoded in the “Rm” field.

For the “T1” variant: is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0]  = ZeroExtend{16}(rotated[7:0]);
    R(d)[31:16] = ZeroExtend{16}(rotated[23:16]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.299 UXTH

Unsigned Extend Halfword

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

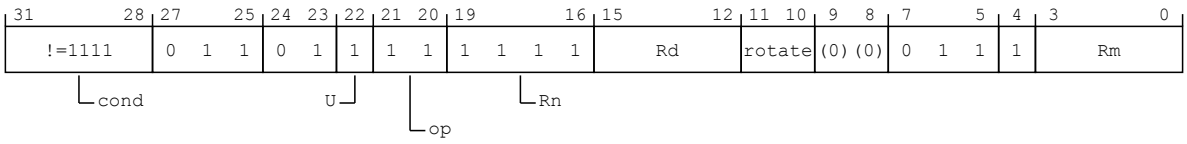

Encoding

```
UXTH{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

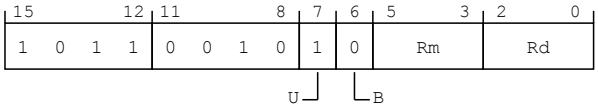

Encoding

```
UXTH{<c>}{<q>}  {<Rd>, }<Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1 1 1 1 1 0 1 0 0	0	0	1	1 1 1 1	1 1 1 1	Rd	1(0)		rotate		Rm
	op1		U	Rn							

Encoding

```
UXTH{<c>}{<q>}  {<Rd>, }<Rm> {, ROR #<amount>}
```

```
UXTH{<c>}.W  {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = ZeroExtend{32}(rotated[15:0]);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.300 WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see Wait For Event and Send Event.

As described in Wait For Event and Send Event, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see HCR.TWE, SCR.TWE, and SCTLR.nTWE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

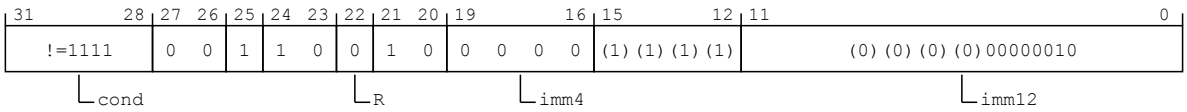

Encoding

```
WFE{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

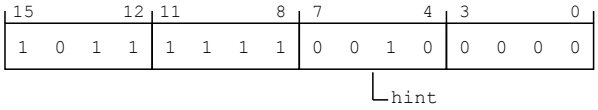

Encoding

```
WFE{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

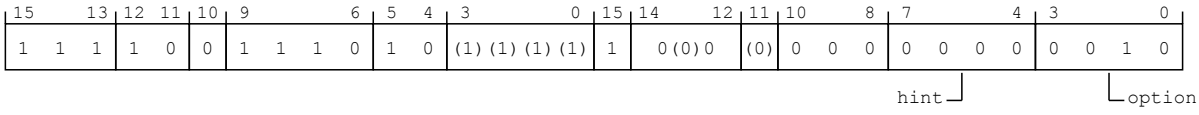

Encoding

```
WFE{<c>}.W
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if HaveEL(EL3) && EL3SDDUndefPriority() then
            // Check for traps described by the Secure Monitor.
            // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
            AArch32_CheckForWFxTrap(EL3, WFxType_WFE);
        end;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32_CheckForWFxTrap(EL1, WFxType_WFE);
        end;
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32_CheckForWFxTrap(EL2, WFxType_WFE);
        end;
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32_CheckForWFxTrap(EL3, WFxType_WFE);
        end;
        WaitForEvent();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.301 WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see Wait For Interrupt.

As described in Wait For Interrupt, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see HCR.TWI, SCR.TWI, and SCTLR.nTWI.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11–0
!=1111 (cond)	0	0	1	1	0	0 (R)	1	0	0	0	0	0 (imm4)	(1)	(1)	(1)	(1)	(0)(0)(0)(0)00000011 (imm12)

Encoding

```
WFI{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1 (hint)	0	0	0	0

Encoding

```
WFI{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0 (hint)	0	0	1	1 (option)

Encoding

```
WFI{<c>}.W
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !AArch32_InterruptPending() then
        if HaveEL(EL3) && EL3SDDUndefPriority() then
            // Check for traps described by the Secure Monitor.
            // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
            AArch32_CheckForWFxTrap(EL3, WFxType_WFI);
        end;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32_CheckForWFxTrap(EL1, WFxType_WFI);
        end;
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32_CheckForWFxTrap(EL2, WFxType_WFI);
        end;
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32_CheckForWFxTrap(EL3, WFxType_WFI);
        end;
        WaitForInterrupt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.1.302 YIELD

Yield hint

`YIELD` is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction see The Yield instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

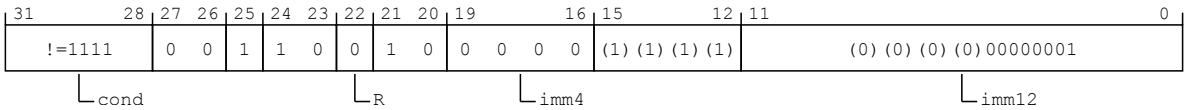

Encoding

```
YIELD{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T1

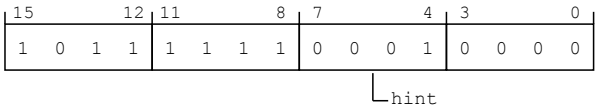

Encoding

```
YIELD{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

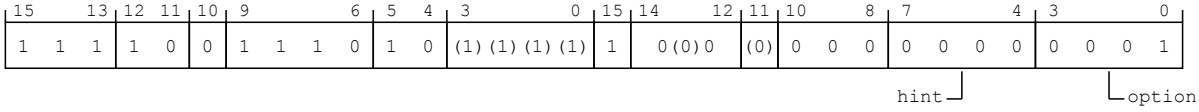

Encoding

```
YIELD{<c>}.W
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F5.2 Encoding and use of banked register transfer instructions

Software executing at EL1 or higher can use the `MRS` (banked register) and `MSR` (banked register) instructions to transfer values between the general-purpose registers and Special-purpose registers. One particular use of these instructions is for a hypervisor to save or restore the register values of a Guest OS. The following sections give more information about these instructions:

- Register arguments in the banked register transfer instructions.
- Usage restrictions on the banked register transfer instructions.
- Encoding the register argument in the banked register transfer instructions.
- Pseudocode support for the banked register transfer instructions.

For descriptions of the instructions see MRS (Banked register) and MSR (Banked register).

F5.2.1 Register arguments in the banked register transfer instructions

Figure F5-1 shows the banked general-purpose registers and Special-purpose registers:

	Associated PE mode							
	User or System	Hyp	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
General-purpose registers	R8_usr							R8_fiq
	R9_usr							R9_fiq
	R10_usr							R10_fiq
	R11_usr							R11_fiq
	R12_usr							R12_fiq
	SP_usr	SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
	LR_usr		LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
Special-purpose registers		SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
		ELR_hyp						

For the general-purpose registers, if no other register is shown, the *current mode register* is the _usr register. So, for example, the full set of current mode registers, including the registers that are not banked:

- For Hyp mode, is {R0_usr - R12_usr, SP_hyp, LR_usr, SPSR_hyp, ELR_hyp}.
- For Abort mode, is {R0_usr - R12_usr, SP_abt, LR_abt, SPSR_abt}.

Figure F5-1 Banking of general-purpose and Special-purpose registers

Figure F5-1 is based on Figure G1-2, that shows the complete set of general-purpose registers and Special-purpose registers accessible in each mode.

Note

- System mode uses the same set of registers as User mode. Neither of these modes can access an SPSR, except that System mode can use the `MRS` (banked register) and `MSR` (banked register) instructions to access some SPSRs, as described in Usage restrictions on the banked register transfer instructions.
- General-purpose registers R0-R7, that are not banked, cannot be accessed using the `MRS` (banked register) and `MSR` (banked register) instructions.
- In addition to the registers shown in Figure F5-1, the DLR and DSPSR are AArch32 System registers that map onto the AArch64 Special-purpose registers DLR_EL0 and DSPSR_EL0. However, DLR and DSPSR are not accessible using the `MRS` (banked register) and `MSR` (banked register) instructions.

Software using an `MRS` (banked register) or `MSR` (banked register) instruction specifies one of these registers using a name shown in Figure F5-1, or an alternative name for SP or LR. These registers can be grouped as follows:

R8-R12 Each of these registers has two banked copies, _usr and _fiq, for example R8_usr and R8_fiq.

SP There is a banked copy of SP for every mode except System mode. For example, SP_svc is the SP for Supervisor mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LR There is a banked copy of LR for every mode except System mode and Hyp mode. For example, LR_svc is the LR for Supervisor mode.

SPSR There is a banked copy of SPSR for every mode except System mode and User mode.

ELR_hyp Except for the operations provided by `MRS` (banked register) and `MSR` (banked register), ELR_hyp is accessible only from Hyp mode. It is not banked.

F5.2.2 Usage restrictions on the banked register transfer instructions

`MRS` (banked register) and `MSR` (banked register) instructions are CONSTRAINED UNPREDICTABLE if any of the following applies:

- The instruction is executed in User mode.
- The instruction accesses a banked register that is not implemented, or that either:
  - Is not accessible from the current Privilege level and Security state.
  - Can be accessed from the current mode by using a different instruction.

MSR (banked register) and MRS (banked register) describes the permitted CONSTRAINED UNPREDICTABLE behavior.

An `MRS` (banked register) instruction or an `MSR` (banked register) instruction executed:

- At Non-secure EL1 cannot access any Hyp mode banked registers.
- At Non-secure EL1 or EL2 cannot access any Monitor mode banked registers.
- In a Secure mode other than Monitor mode cannot access any Hyp banked registers.

This means that the banked registers that `MRS` (banked register) and `MSR` (banked register) instructions cannot access are:

From Monitor mode

- The current mode registers R8_usr-R12_usr, SP_mon, LR_mon, and SPSR_mon.

From Hyp mode

- The Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The current mode registers R8_usr-R12_usr, SP_hyp, LR_usr, and SPSR_hyp.

Note

`MRS` (banked register) and `MSR` (banked register) instructions can access the current mode register ELR_hyp.

From FIQ mode

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_fiq-R12_fiq, SP_fiq, LR_fiq, and SPSR_fiq.

From System mode

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_usr, and LR_usr.

From Supervisor mode, Abort mode, Undefined mode, and IRQ mode

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_<current_mode>, LR_<current_mode>, and SPSR_<current_mode>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If EL3 is using AArch64, all `MRS` (banked register) and `MSR` (banked register) accesses to the Monitor mode registers from Secure EL1 modes are trapped to EL3.

For more information, see:

- Encoding the register argument in the banked register transfer instructions.
- Pseudocode support for the banked register transfer instructions.
- MRS (Banked register).
- MSR (Banked register).

Note

CONSTRAINED UNPREDICTABLE behavior must not give access to registers that are not accessible from the current Privilege level and Security state.

F5.2.3 Encoding the register argument in the banked register transfer instructions

The `MRS` (banked register) and `MSR` (banked register) instructions include a 5-bit field, SYSm, and an R bit, that together encode the register argument for the instruction.

When the R bit is set to 0, the argument is a register other than a banked copy of the SPSR, and Table F5-116 shows how the SYSm field defines the required register argument.

Table F5-116 Banked register encodings when R==0

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b000	R8_usr	R8_fiq	LR_irq	CONSTRAINED UNPREDICTABLE
0b001	R9_usr	R9_fiq	SP_irq	CONSTRAINED UNPREDICTABLE
0b010	R10_usr	R10_fiq	LR_svc	CONSTRAINED UNPREDICTABLE
0b011	R11_usr	R11_fiq	SP_svc	CONSTRAINED UNPREDICTABLE
0b100	R12_usr	R12_fiq	LR_abt	LR_mon
0b101	SP_usr	SP_fiq	SP_abt	SP_mon
0b110	LR_usr	LR_fiq	LR_und	ELR_hyp
0b111	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SP_und	SP_hyp

When the R bit is set to 1, the argument is a banked copy of the SPSR, and Table F5-117 shows how the SYSm field defines the required register argument.

Table F5-117 Banked register encodings when R==1

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b000	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SPSR_irq	CONSTRAINED UNPREDICTABLE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b001	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE
0b010	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SPSR_svc	CONSTRAINED UNPREDICTABLE
0b011	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE
0b100	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SPSR_abt	SPSR_mon
0b101	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE
0b110	CONSTRAINED UNPREDICTABLE	SPSR_fiq	SPSR_und	SPSR_hyp
0b111	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE

F5.2.4 Pseudocode support for the banked register transfer instructions

The pseudocode functions `BankedRegisterAccessValid()` and `SPSRaccessValid()` check the validity of `MRS` (banked register) and `MSR` (banked register) accesses. That is, they filter the accesses that are CONSTRAINED UNPREDICTABLE either because:

- They attempt to access a register that Usage restrictions on the banked register transfer instructions shows is not accessible.
- They use an `SYSm<4:0>` encoding that Encoding the register argument in the banked register transfer instructions shows as CONSTRAINED UNPREDICTABLE.

`BankedRegisterAccessValid()` applies to accesses to the banked general-purpose registers, or to ELR_hyp, and `SPSRaccessValid()` applies to accesses to the SPSRs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F6
T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- Alphabetical list of Advanced SIMD and floating-point instructions.

Note

Some headings in this chapter use the term *floating-point register*. This is an abbreviated description, and means a register in the Advanced SIMD and floating-point register file.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

This section lists every Advanced SIMD and floating-point instruction in the T32 and A32 instruction sets. For details of the format used see Format of instruction descriptions.

This section is formatted so that each instruction description starts on a new page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.1 AESD

AES single round decryption.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AES)

31			28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10				7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd		0	0	1	1	0	1	M	0	Vm	
														opc1					opc2					Q			

Encoding

```
AESD.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AES)

15			13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10				7	6	5	4	3	0
1	1	1		1	1	1	1	1	1	D	1	1	size		0	0	Vd		0	0	1	1	0		1	M	0	Vm	
															opc1					opc2					Q				

Encoding

```
AESD.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    let operand1 : bits(128) = Q(d>>1);
    let operand2 : bits(128) = Q(m>>1);
    Q(d>>1) = AESInvSubBytes(AESInvShiftRows(operand1 XOR operand2));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.2 AESE

AES single round encryption.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AES)

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd	0	0	1	1	0	0	M	0	Vm
														opc1				opc2				Q			

Encoding

```
AESE.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AES)

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd	0	0	1	1	0	0	M	0	Vm
														opc1				opc2				Q			

Encoding

```
AESE.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    let operand1 : bits(128) = Q(d>>1);
    let operand2 : bits(128) = Q(m>>1);
    Q(d>>1) = AESSubBytes(AESShiftRows(operand1 XOR operand2));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.3 AESIMC

AES inverse mix columns.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AES)

31–28	27	26–25	24	23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1	0	0 1	1	1	D	1 1	size	0 0	Vd	0	0 1 1 1	1	M	0	Vm
								opc1			opc2	Q			

Encoding

```
AESIMC.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AES)

15–13	12	11–10	9–8	7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1	1	1 1	1 1	1	D	1 1	size	0 0	Vd	0	0 1 1 1	1	M	0	Vm
								opc1			opc2	Q			

Encoding

```
AESIMC.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q(d>>1) = AESInvMixColumns(Q(m>>1));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.4 AESMC

AES mix columns.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AES)

31–28	27	26–25	24	23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1	0	0 1	1	1	D	1 1	size	0 0	Vd	0	0 1 1 1	0	M	0	Vm
								opc1			opc2	Q			

Encoding

```
AESMC.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AES)

15–13	12	11–10	9–8	7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1	1	1 1	1 1	1	D	1 1	size	0 0	Vd	0	0 1 1 1	0	M	0	Vm
								opc1			opc2	Q			

Encoding

```
AESMC.<dt>  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q(d>>1) = AESMixColumns(Q(m>>1));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.5 FLDM*X (FLDMDBX, FLDMIAX)

FLDM*X

FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM*X loads multiple SIMD&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–10	9–8	7–0
!=1111	1 1 0	P	U	D	W	1	Rn	Vd	1 0	1 1	x x x x x x x 1
cond						L				size	imm8

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
FLDMDBX{<c>}{<q>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
FLDMIAX{<c>}{<q>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15						9	8	7	6	5	4	3	0	15	12	11	10	9	8	7							0
1	1	1	0	1	1	0	P	U	D	W	1	Rn		Vd		1	0	1	1	x	x	x	x	x	x	x	1
											L							size						imm8			

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
FLDMDBX{<c>}{<q>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
FLDMIAX{<c>}{<q>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: See Advanced SIMD and floating-point 64-bit move for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move for the A32 instruction set.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in "W":

W	!
0	[absent]
1	[present]

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;

    for r = 0 to regs-1 do
        if single_regs then
            S(d+r) = MemA{32}(address);
            address = address+4;
        else
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            address = address+8;

            // Combine the word-aligned words in the correct order for current endianness.
            D(d+r) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.6 FSTMDBX, FSTMIAX

FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

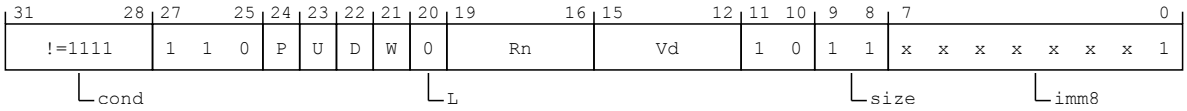

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
FSTMDBX{<c>}{<q>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
FSTMIAX{<c>}{<q>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 1	x x x x x x x 1
					L				size	imm8

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
FSTMDBX{<c>}{<q>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
FSTMIAX{<c>}{<q>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: See Advanced SIMD and floating-point 64-bit move for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move for the A32 instruction set.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in "W":

W	!
0	[absent]
1	[present]

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;
    for r = 0 to regs-1 do
        if single_regs then
            MemA{32}(address) = S(d+r);
            address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address) =  D(d+r)[63:32];
                MemA{32}(address+4) =  D(d+r)[31:0];
            else
                MemA{32}(address) = D(d+r)[31:0];
                MemA{32}(address+4) = D(d+r)[63:32];
            end;

            address = address+8;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.7 SHA1C

SHA1 hash update (choose).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA1)

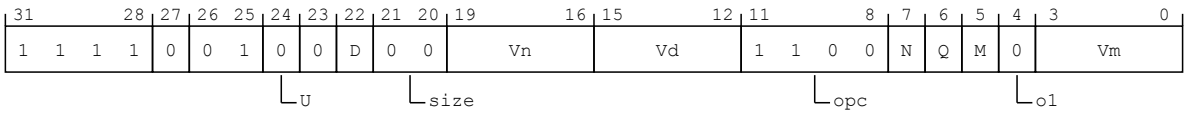

Encoding

```
SHA1C.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA1)

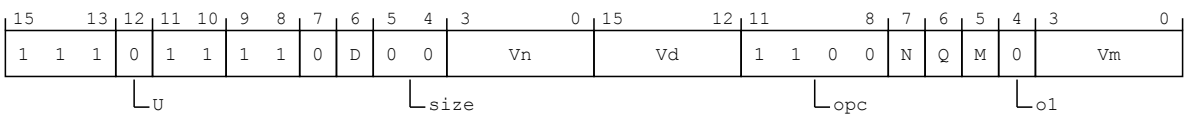

Encoding

```
SHA1C.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAchoose(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*:32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+:32], yx[0+:128]);
    end;
    Q(d>>1) = x;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.8 SHA1H

SHA1 fixed rotate.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA1)

31–28	27	26–25	24	23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1	0	0 1	1	1	D	1 1	size	0 1	Vd	0	0 1 0 1	1	M	0	Vm
								opc1			opc2	Q			

Encoding

```
SHA1H.32  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA1)

15–13	12	11–10	9–8	7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1	1	1 1	1 1	1	D	1 1	size	0 1	Vd	0	0 1 0 1	1	M	0	Vm
								opc1			opc2	Q			

Encoding

```
SHA1H.32  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    Q(d>>1) = ZeroExtend{128}(ROL(Q(m>>1)[31:0], 30));
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.9 SHA1M

SHA1 hash update (majority).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA1)

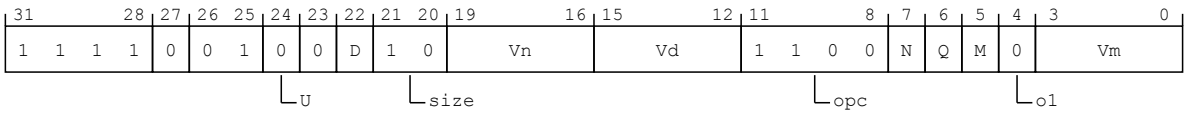

Encoding

```
SHA1M.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA1)

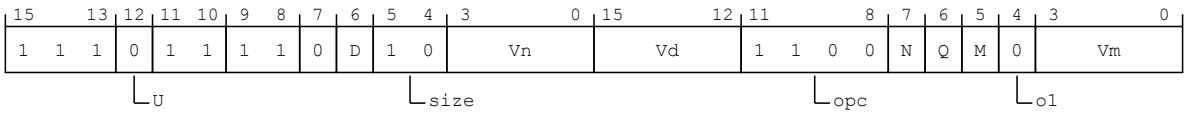

Encoding

```
SHA1M.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAmajority{}(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*:32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+:32], yx[0+:128]);
    end;
    Q(d>>1) = x;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.10 SHA1P

SHA1 hash update (parity).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA1)

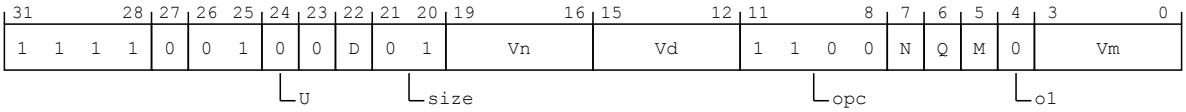

Encoding

```
SHA1P.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA1)

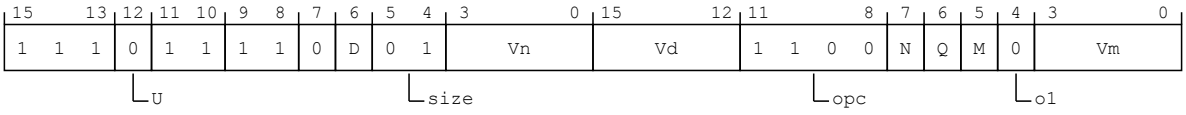

Encoding

```
SHA1P.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAparity(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*:32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+:32], yx[0+:128]);
    end;
    Q(d>>1) = x;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.11 SHA1SU0

SHA1 schedule update 0.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA1)

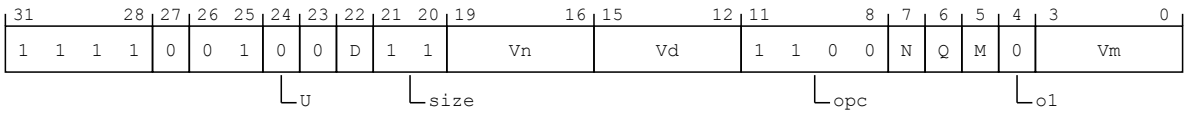

Encoding

```
SHA1SU0.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA1)

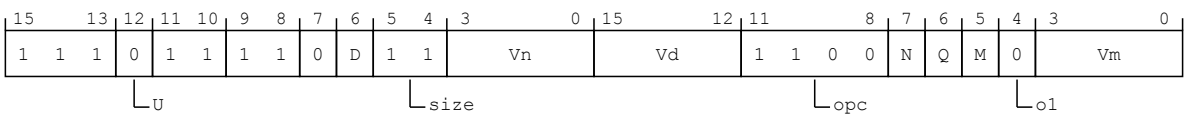

Encoding

```
SHA1SU0.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let op1 : bits(128) = Q(d>>1);
    var op2 : bits(128) = Q(n>>1);
    let op3 : bits(128) = Q(m>>1);
    op2 = op2[63:0] :: op1[127:64];
    Q(d>>1) = op1 XOR op2 XOR op3;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.12 SHA1SU1

SHA1 schedule update 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA1)

31–28	27	26–25	24	23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1	0	0 1	1	1	D	1 1	size	1 0 (opc1)	Vd	0	0 1 1 1 (opc2)	0 (Q)	M	0	Vm

Encoding

```
SHA1SU1.32  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA1)

15–13	12	11–10	9–8	7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1	1	1 1	1 1	1	D	1 1	size	1 0 (opc1)	Vd	0	0 1 1 1 (opc2)	0 (Q)	M	0	Vm

Encoding

```
SHA1SU1.32  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let X : bits(128) = Q(d>>1);
    let Y : bits(128) = Q(m>>1);
    let T : bits(128) = X XOR LSR(Y, 32);
    let W0 : bits(32) = ROL(T[31:0], 1);
    let W1 : bits(32) = ROL(T[63:32], 1);
    let W2 : bits(32) = ROL(T[95:64], 1);
    let W3 : bits(32) = ROL(T[127:96], 1) XOR ROL(T[31:0], 2);
    Q(d>>1) = W3::W2::W1::W0;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.13 SHA256H

SHA256 hash update part 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA256)

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0 1	1	0	D	0 0	Vn	Vd	1 1 0 0	N	Q	M	0	Vm
			U			size			opc				o1	

Encoding

```
SHA256H.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA256)

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1	1 1	1 1	0	D	0 0	Vn	Vd	1 1 0 0	N	Q	M	0	Vm
	U					size			opc				o1	

Encoding

```
SHA256H.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let X : bits(128) = Q(d>>1);
    let Y : bits(128) = Q(n>>1);
    let W : bits(128) = Q(m>>1);
    let part1 : boolean = TRUE;
    Q(d>>1) = SHA256hash(X, Y, W, part1);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.14 SHA256H2

SHA256 hash update part 2.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA256)

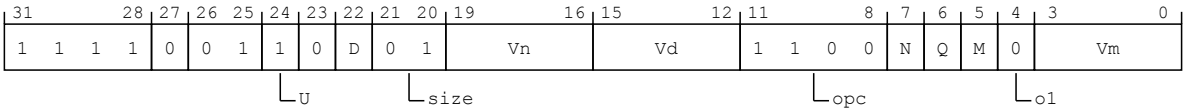

Encoding

```
SHA256H2.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA256)

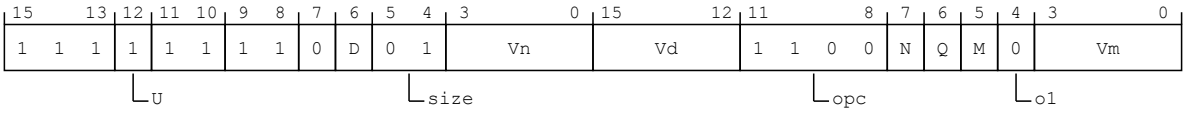

Encoding

```
SHA256H2.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let X : bits(128) = Q(n>>1);
    let Y : bits(128) = Q(d>>1);
    let W : bits(128) = Q(m>>1);
    let part1 : boolean = FALSE;
    Q(d>>1) = SHA256hash(X, Y, W, part1);
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.15 SHA256SU0

SHA256 schedule update 0.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA256)

31–28	27	26	25	24	23	22	21	20	19–18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	1	1	1	1	M	0	Vm
										opc1				opc2				Q			

Encoding

```
SHA256SU0.32  <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA256)

15–13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1	1	1	1	1	1	1	D	1	1	size		1	0	Vd	0	0	1	1	1	1	M	0	Vm
												opc1				opc2				Q			

Encoding

```
SHA256SU0.32  <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    var result : bits(128);
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let x : bits(128) = Q(d>>1);
    let y : bits(128) = Q(m>>1);
    let t : bits(128) = y[31:0] :: x[127:32];
    for e = 0 to 3 do
        var elt : bits(32) = t[e*:32];
        elt = ROR(elt, 7) XOR ROR(elt, 18) XOR LSR(elt, 3);
        result[e*:32] = elt + x[e*:32];
    end;
    Q(d>>1) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.16 SHA256SU1

SHA256 schedule update 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_SHA256)

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0	1	1	0	D	1	0	Vn	Vd	1 1 0 0	N	Q	M	0	Vm
				U			size				opc				o1	

Encoding

```
SHA256SU1.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_SHA256)

15–13	12	11	10	9	8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1	1	1	1	1	0	D	1	0	Vn	Vd	1 1 0 0	N	Q	M	0	Vm
	U							size				opc				o1	

Encoding

```
SHA256SU1.32  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var elt : bits(32);
    var result : bits(128);
    let x : bits(128) = Q(d>>1);
    let y : bits(128) = Q(n>>1);
    let z : bits(128) = Q(m>>1);
    let T0 : bits(128) = z[31:0] :: y[127:32];

    var T1 : bits(64) = z[127:64];
    for e = 0 to 1 do
        elt = T1[e*:32];
        elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
        elt = elt + x[e*:32] + T0[e*:32];
        result[e*:32] = elt;
    end;

    T1 = result[63:0];
    for e = 2 to 3 do
        elt = T1[(e - 2)*:32];
        elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
        elt = elt + x[e*:32] + T0[e*:32];
        result[e*:32] = elt;
    end;

    Q(d>>1) = result;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.17 VABA

Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 1 1 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 1 1 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + absdiff;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + absdiff;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.18 VABAL

Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0 1 0 1	N	0	M	0	Vm
						size			opc					

Encoding

```
VABAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0 1 0 1	N	0	M	0	Vm
						size			opc					

Encoding

```
VABAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + absdiff;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + absdiff;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.19 VABD (floating-point)

Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

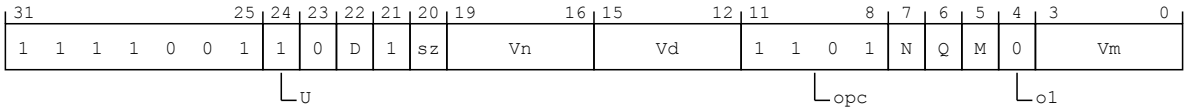

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

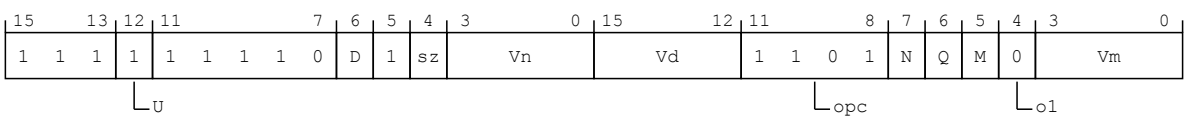

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            D(d+r)[e*:esize] = FPAbs{esize}(FPSub{esize}(op1elt, op2elt, fpcr), fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.20 VABD (integer)

Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 1 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 1 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = absdiff[2*esize-1:0];
            else
                D(d+r)[e*:esize] = absdiff[esize-1:0];
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.21 VABDL (integer)

Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0 1 1 1	N	0	M	0	Vm
						size			opc					

Encoding

```
VABDL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0 1 1 1	N	0	M	0	Vm
						size			opc					

Encoding

```
VABDL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = absdiff[2*esize-1:0];
            else
                D(d+r)[e*:esize] = absdiff[esize-1:0];
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.22 VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 1	Vd	0	F	1 1 0	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABS{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABS{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31–28	27–23	22	21–20	19	18–16	15–12	11–10	9–8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1 1	0	0 0 0	Vd	1 0	size	1	1	M	0	Vm
cond				o1	opc2				o3				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VABS{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VABS{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VABS{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15 … 7	6	5	4	3	2	1	0	15 … 12	11	10	9	7	6	5	4	3 … 0
1 1 1 1 1 1 1 1 1	D	1	1	size		0	1	Vd	0	F	1 1	0	Q	M	0	Vm
						opc1										

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VABS{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VABS{<c>}{<q>}.<dt>  <Qd>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15–7	6	5–4	3	2–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 0 1 1 1 0 1	D	1 1	0	0 0 0	Vd	1 0	size	1	1	M	0	Vm
			o1	opc2				o3				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VABS{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VABS{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VABS{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                if floating_point then
                    D(d+r)[e*:esize] = FPAbs{esize}(D(m+r)[e*:esize], fpcr);
                else
                    let result : integer = Abs(SInt(D(m+r)[e*:esize]));
                    D(d+r)[e*:esize] = result[esize-1:0];
                end;
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 => H(d) = FPAbs{16}(H(m), fpcr);
            when 32 => S(d) = FPAbs{32}(S(m), fpcr);
            when 64 => D(d) = FPAbs{64}(D(m), fpcr);
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.23 VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VACLE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

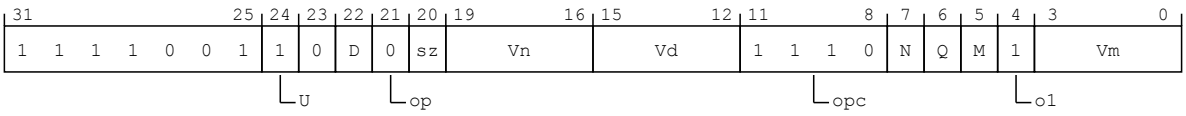

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACGE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACGE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

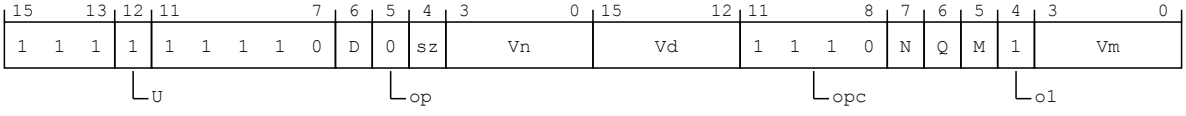

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACGE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACGE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = FPAbs{}(D(n+r)[e*:esize], fpcr);
            let op2 : bits(esize) = FPAbs{}(D(m+r)[e*:esize], fpcr);
            var test_passed : boolean;
            if or_equal then
                test_passed = FPCompareGE{esize}(op1, op2, fpcr);
            else
                test_passed = FPCompareGT{esize}(op1, op2, fpcr);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.24 VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VACLT.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

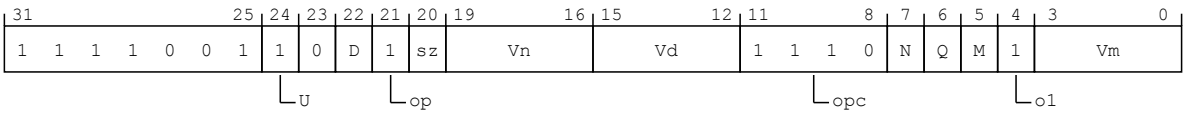

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACGT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACGT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

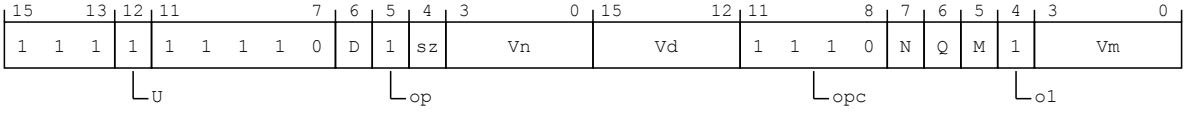

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACGT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACGT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = FPAbs{}(D(n+r)[e*:esize], fpcr);
            let op2 : bits(esize) = FPAbs{}(D(m+r)[e*:esize], fpcr);
            var test_passed : boolean;
            if or_equal then
                test_passed = FPCompareGE{esize}(op1, op2, fpcr);
            else
                test_passed = FPCompareGT{esize}(op1, op2, fpcr);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.25 VACLE

Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VACGE. This means:

- The encodings in this description are named to match the encodings of VACGE.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGE gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

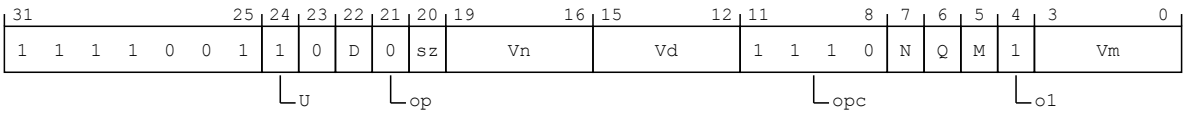

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACLE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACLE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

T1

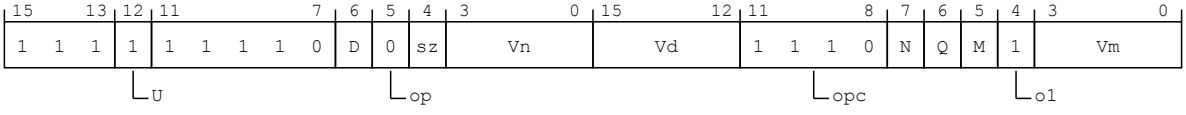

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACLE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACLE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VACGE gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.26 VACLT

Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VACGT. This means:

- The encodings in this description are named to match the encodings of VACGT.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGT gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

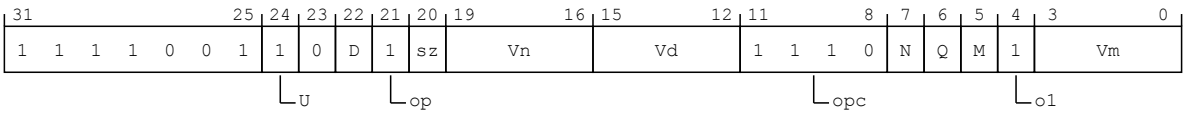

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACLT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACLT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

T1

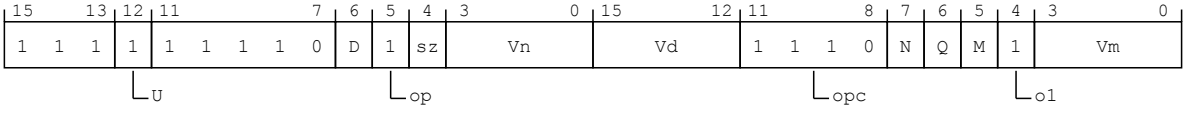

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VACLT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VACLT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VACGT gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.27 VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

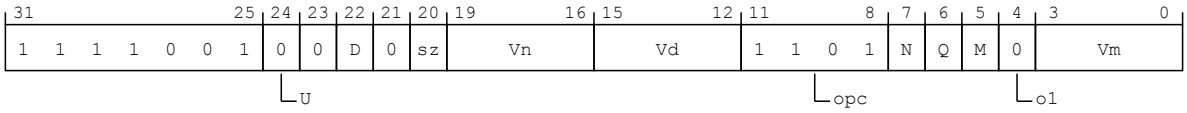

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

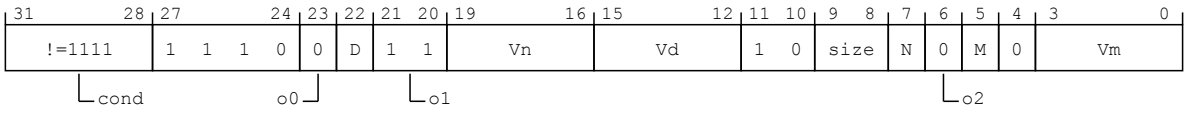

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VADD{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VADD{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VADD{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 0 1	N	Q	M	0	Vm
	U							opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	0	D	1	1	Vn		Vd		1	0	size		N	0	M	0	Vm	
								o0			o1										o2				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VADD{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VADD{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VADD{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPAdd{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                H(d) = FPAdd{16}(H(n), H(m), fpcr);
            when 32 =>
                S(d) = FPAdd{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPAdd{64}(D(n), D(m), fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.28 VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	0	0	D	size	Vn	Vd	1 0 0 0	N	Q	M	0	Vm
	U						opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	size	Vn	Vd	1 0 0 0	N	Q	M	0	Vm
	U						opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = D(n+r)[e*:esize] + D(m+r)[e*:esize];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.29 VADDHN

Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see VRADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

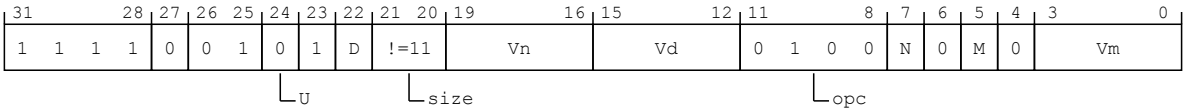

Encoding

```
VADDHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1

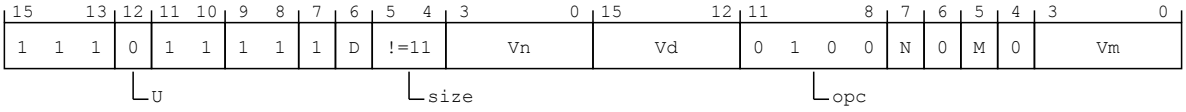

Encoding

```
VADDHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "size":

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = Qin(n>>1)[e*:(2*esize)] + Qin(m>>1)[e*:(2*esize)];
        D(d)[e*:esize] = result[2*esize-1:esize];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.30 VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0 0 0	0	N	0	M	0	Vm
						size				op					

Encoding

```
VADDL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0 0 0	0	N	0	M	0	Vm
						size				op					

Encoding

```
VADDL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the second operand vector, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vaddw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 + element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.31 VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0 0 0	1	N	0	M	0	Vm
						size				op					

Encoding

```
VADDW{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0 0 0	1	N	0	M	0	Vm
						size				op					

Encoding

```
VADDW{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the second operand vector, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vaddw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 + element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.32 VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

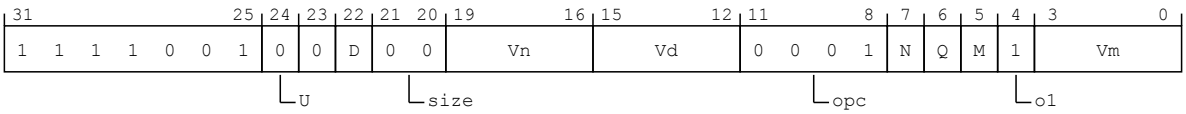

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VAND{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VAND{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

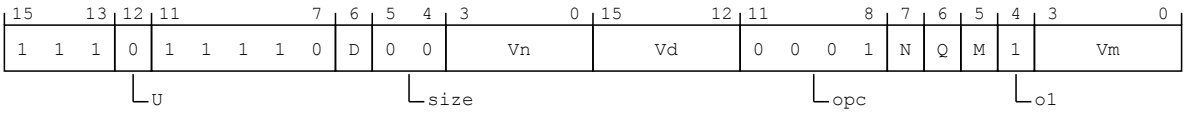

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VAND{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VAND{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) AND D(m+r);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.33 VAND (immediate)

Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of VBIC (immediate). This means:

- The encodings in this description are named to match the encodings of VBIC (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VBIC (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31						25	24	23	22	21			19	18		16	15			12	11			8	7	6	5	4	3			0
1	1	1	1	0	0	1	i	1	D	0	0	0		imm3			Vd				0	x	x	1	0	Q	1	1	imm4			
																					cmode						op					

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VAND{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32  <Dd>, #~<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VAND{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32  <Qd>, #~<imm>
```

A2

31						25	24	23	22	21			19	18		16	15			12	11			8	7	6	5	4	3			0
1	1	1	1	0	0	1	i	1	D	0	0	0		imm3			Vd				1	0	x	1	0	Q	1	1	imm4			
																					cmode						op					

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VAND{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16  <Dd>, #~<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VAND{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16  <Qd>, #~<imm>
```

T1

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	0 x x 1	0	Q	1	1	imm4
							cmode			op		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VAND{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32  <Dd>, #~<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VAND{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32  <Qd>, #~<imm>
```

T2

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 0 x 1	0	Q	1	1	imm4
							cmode			op		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VAND{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16  <Dd>, #~<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VAND{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16  <Qd>, #~<imm>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

The description of VBIC (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of VBIC (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.34 VBIC (immediate)

Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VAND (immediate).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

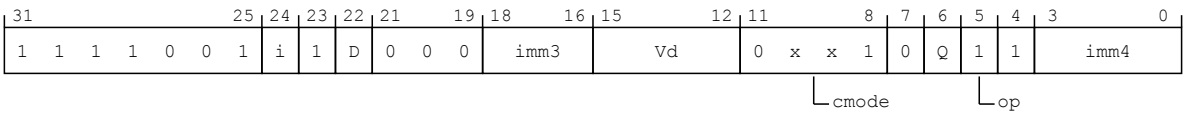

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIC{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIC{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

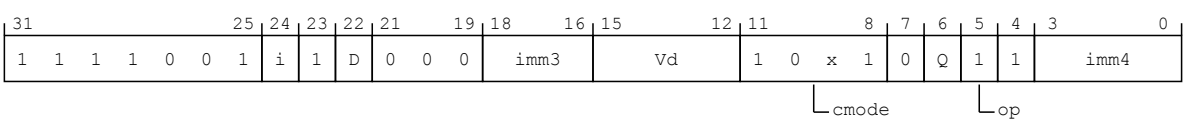

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIC{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIC{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	0 x x 1 (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIC{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIC{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 0 x 1 (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIC{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIC{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(d+r) AND NOT(imm64);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.35 VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

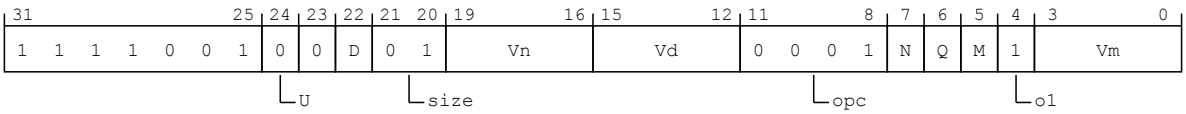

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIC{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIC{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

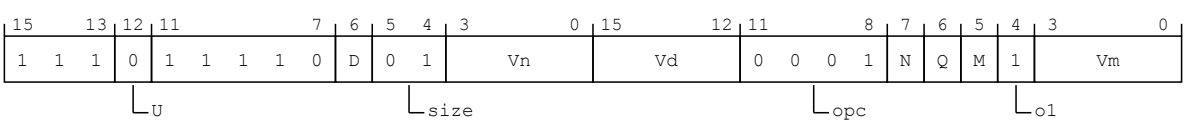

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIC{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIC{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) AND NOT(D(m+r));
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.36 VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

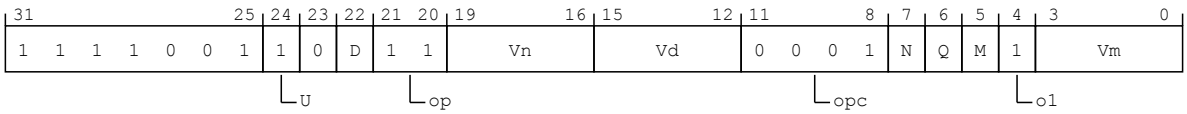

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIF{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIF{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

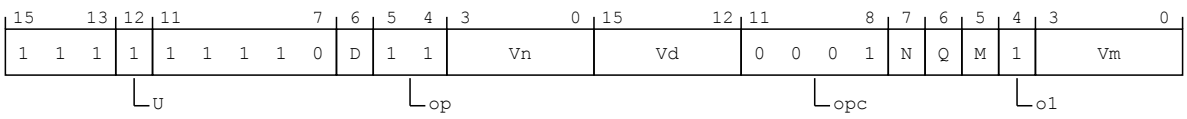

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIF{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIF{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps_VBIF =>  D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps_VBIT =>  D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps_VBSL =>  D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.37 VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

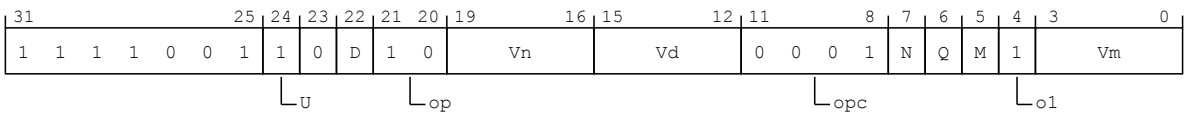

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIT{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIT{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

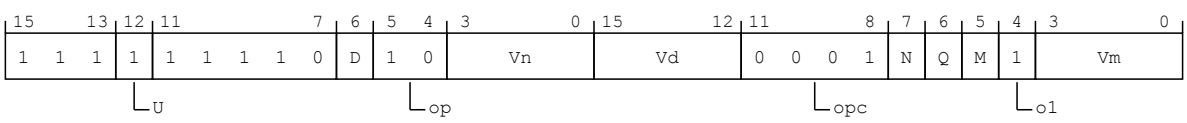

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBIT{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBIT{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps_VBIF =>  D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps_VBIT =>  D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps_VBSL =>  D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.38 VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

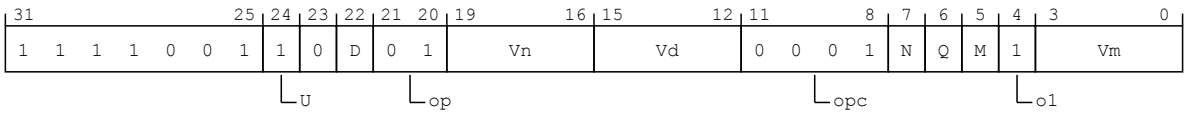

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBSL{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBSL{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

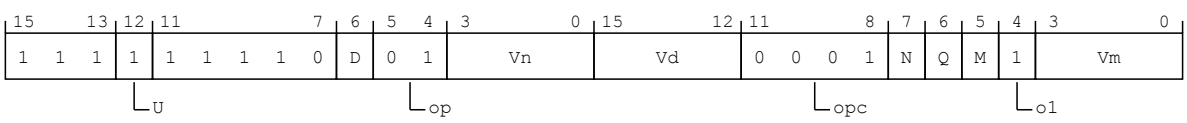

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VBSL{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VBSL{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps_VBIF =>  D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps_VBIT =>  D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps_VBSL =>  D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.39 VCADD

Vector Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FCMA)

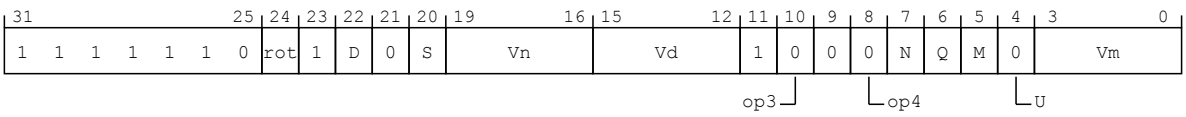

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCADD{<q>}.<dt>  <Dd>, <Dn>, <Dm>, #<rotate>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCADD{<q>}.<dt>  <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if S == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 16 << UInt(S);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FCMA)

15–9	8	7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 1 1 0	rot	1	D	0	S	Vn	Vd	1	0	0	0	N	Q	M	0	Vm
								op3			op4				U	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCADD{<q>}.<dt>  <Dd>, <Dn>, <Dm>, #<rotate>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCADD{<q>}.<dt>  <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if S == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 16 << UInt(S);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “S”:

S	<dt>
0	F16
1	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<rotate>

Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
0	90
1	270

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    let operand1 : bits(64) = D(n+r);
    let operand2 : bits(64) = D(m+r);
    let fpcr : FPCR_Type = StandardFPCR();
    for e = 0 to (elements DIV 2)-1 do
        var element1 : bits(esize);
        var element3 : bits(esize);
        case rot of
            when '0' =>
                element1 = FPNeg{esize}(operand2[(e*2+1)*:esize], fpcr);
                element3 = operand2[(e*2)*:esize];
            when '1' =>
                element1 = operand2[(e*2+1)*:esize];
                element3 = FPNeg{esize}(operand2[(e*2)*:esize], fpcr);
        end;
        let result1 : bits(esize) = FPAdd{}(operand1[(e*2)*:esize],element1,fpcr);
        let result2 : bits(esize) = FPAdd{}(operand1[(e*2+1)*:esize],element3,fpcr);
        D(d+r)[(e*2)*:esize] = result1;
        D(d+r)[(e*2+1)*:esize] = result2;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.40 VCEQ (immediate #0)

Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 1 (opc1)	Vd	0	F	0 1 0	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCEQ{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCEQ{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 1 (opc1)	Vd	0	F	0 1 0	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCEQ{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCEQ{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareEQ{esize}(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (D(m+r)[e*:esize] == Zeros{esize});
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.41 VCEQ (register)

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

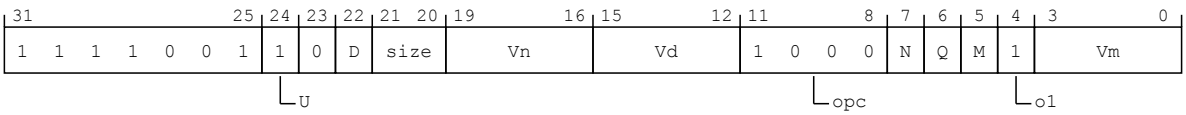

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCEQ{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCEQ{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let int_operation : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

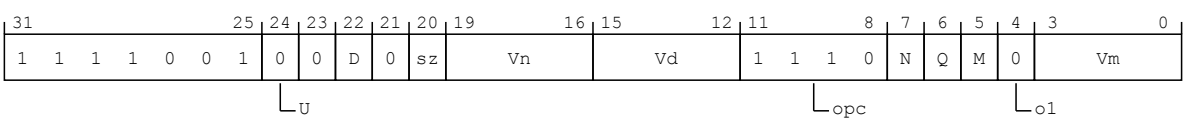

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCEQ{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCEQ{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let int_operation : boolean = FALSE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15 13	12	11 7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	1	1 1 1 1 0	D	size	Vn	Vd	1 0 0 0	N	Q	M	1	Vm
	U						opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCEQ{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCEQ{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let int_operation : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	0	Vm
	U							opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCEQ{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCEQ{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let int_operation : boolean = FALSE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt>
00	I8
01	I16
10	I32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            if int_operation then
                test_passed = (op1elt == op2elt);
            else
                test_passed = FPCompareEQ{esize}(op1elt, op2elt, StandardFPCR());
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.42 VCGE (immediate #0)

Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 1	Vd	0	F	0 0 1	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGE{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGE{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 1	Vd	0	F	0 0 1	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGE{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGE{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareGE{esize}(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) >= 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.43 VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VCLE (register).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

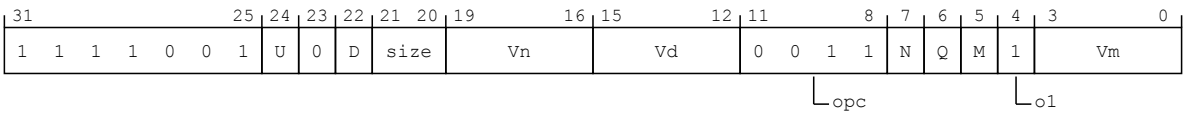

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGEType = if U == '1' then VCGEType_unsigned else VCGEType_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

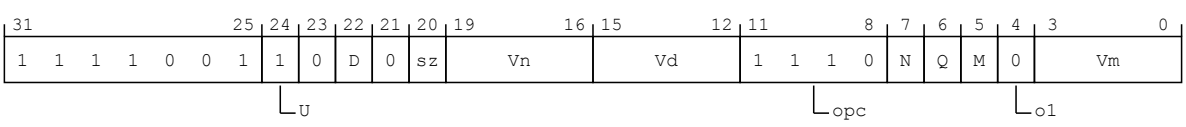

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCGE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let vtype : VCGEType = VCGEType_fp;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0 1 1 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGEType = if U == '1' then VCGEType_unsigned else VCGEType_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

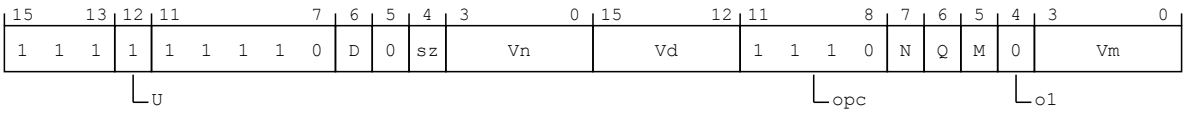

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VCGEType = VCGEType_fp;
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            case vtype of
                when VCGEType_signed =>    test_passed = (SInt(op1elt) >= SInt(op2elt));
                when VCGEType_unsigned =>  test_passed = (UInt(op1elt) >= UInt(op2elt));
                when VCGEType_fp =>        test_passed = FPCompareGE{esize}(op1elt, op2elt,
                                                                                StandardFPCR());
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.44 VCGT (immediate #0)

Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 1	Vd	0	F	0 0 0	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGT{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGT{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 1	Vd	0	F	0 0 0	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGT{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGT{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareGT{esize}(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) > 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.45 VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VCLT (register).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 0 1 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGTtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31–25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	1 (U)	0	D	1	sz	Vn	Vd	1 1 1 0 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCGT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let vtype : VCGTtype = VCGTtype_fp;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15 13	12	11 7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0 1 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGTtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

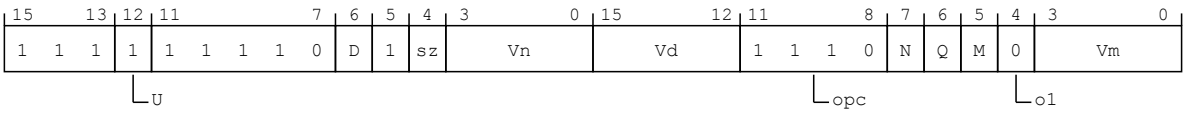

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCGT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCGT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VCGTtype = VCGTtype_fp;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            case vtype of
                when VCGTtype_signed =>    test_passed = (SInt(op1elt) > SInt(op2elt));
                when VCGTtype_unsigned =>  test_passed = (UInt(op1elt) > UInt(op2elt));
                when VCGTtype_fp =>        test_passed = FPCompareGT{esize}(op1elt, op2elt,
                                                                                StandardFPCR());
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.46 VCLE (immediate #0)

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 1	Vd	0	F	0 1 1	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLE{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLE{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 1	Vd	0	F	0 1 1	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLE{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLE{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareGE{esize}(zero, D(m+r)[e*:esize], StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) <= 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.47 VCLE (register)

Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VCGE (register). This means:

- The encodings in this description are named to match the encodings of VCGE (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGE (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 0 1 1 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

A2

31–25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	1 (U)	0	D	0	sz	Vn	Vd	1 1 1 0 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0 1 1	N	Q	M	1	Vm
							opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

T2

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	0	Vm
	U							opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLE{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLE{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VCGE (register) gives the operational pseudocode for this instruction.

Operational Information

The description of VCGE (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.48 VCLS

Vector Count Leading Sign Bits determines the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The result does not include the topmost bit itself.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 0 0 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLS{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLS{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 0 0 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLS{<c>}{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLS{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = CountLeadingSignBits(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.49 VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 1 (opc1)	Vd	0	F	1 0 0	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLT{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLT{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 1 (opc1)	Vd	0	F	1 0 0	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLT{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, #0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLT{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareGT{esize}(zero, D(m+r)[e*:esize], StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) < 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.50 VCLT (register)

Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VCGT (register). This means:

- The encodings in this description are named to match the encodings of VCGT (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGT (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 0 1 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

A2

31–25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	1 (U)	0	D	1	sz	Vn	Vd	1 1 1 0 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0 1 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

T2

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1 (U)	1 1 1 1 0	D	1	sz	Vn	Vd	1 1 1 0 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLT{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Dd>, <Dm>, <Dn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLT{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt>  <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "T1 128-bit SIMD vector", and "T1 64-bit SIMD vector" variants: is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VCGT (register) gives the operational pseudocode for this instruction.

Operational Information

The description of VCGT (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.51 VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 0 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLZ{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLZ{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 0 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCLZ{<c>}{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCLZ{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = CountLeadingZeroBits(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.52 VCMLA

Vector Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FCMA)

31–25	24–23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 1 1 0	rot	D	1	S	Vn	Vd	1	0	0	0	N	Q	M	0	Vm
							op3			op4				U	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCMLA{<q>}.<dt>  <Dd>, <Dn>, <Dm>, #<rotate>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCMLA{<q>}.<dt>  <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FCMA)

15–9	8–7	6	5	4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 1 1 0	rot	D	1	S	Vn	Vd	1	0	0	0	N	Q	M	0	Vm
							op3			op4				U	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCMLA{<q>}.<dt>  <Dd>, <Dn>, <Dm>, #<rotate>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCMLA{<q>}.<dt>  <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “S”:

S	<dt>
0	F16
1	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<rotate>

Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    let operand1 : bits(64) = D(n+r);
    let operand2 : bits(64) = D(m+r);
    let operand3 : bits(64) = D(d+r);
    for e = 0 to (elements DIV 2)-1 do
        var element1 : bits(esize);
        var element2 : bits(esize);
        var element3 : bits(esize);
        var element4 : bits(esize);
        case rot of
            when '00' =>
                element1 = operand2[(e*2)*:esize];
                element2 = operand1[(e*2)*:esize];
                element3 = operand2[(e*2+1)*:esize];
                element4 = operand1[(e*2)*:esize];
            when '01' =>
                element1 = FPNeg{esize}(operand2[(e*2+1)*:esize], fpcr);
                element2 = operand1[(e*2+1)*:esize];
                element3 = operand2[(e*2)*:esize];
                element4 = operand1[(e*2+1)*:esize];
            when '10' =>
                element1 = FPNeg{esize}(operand2[(e*2)*:esize], fpcr);
                element2 = operand1[(e*2)*:esize];
                element3 = FPNeg{esize}(operand2[(e*2+1)*:esize], fpcr);
                element4 = operand1[(e*2)*:esize];
            when '11' =>
                element1 = operand2[(e*2+1)*:esize];
                element2 = operand1[(e*2+1)*:esize];
                element3 = FPNeg{esize}(operand2[(e*2)*:esize], fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                element4 = operand1[(e*2+1)*:esize];
        end;
        let result1 : bits(esize) = FPMulAdd{}(operand3[(e*2)*:esize],
                                                element2,element1, fpcr);
        let result2 : bits(esize) = FPMulAdd{}(operand3[(e*2+1)*:esize],
                                                element4,element3,fpcr);
        D(d+r)[(e*2)*:esize] = result1;
        D(d+r)[(e*2+1)*:esize] = result2;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.53 VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FCMA)

31–24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0	S	D	rot	Vn	Vd	1 0 0 0	N	Q	M	0	Vm
										U	

Encoding for the 64-bit SIMD vector of half-precision floating-point variant

Applies when `(S == 0 && Q == 0)`

```
VCMLA{<q>}.F16  <Dd>, <Dn>, <Dm>[<index>], #<rotate>
```

Encoding for the 64-bit SIMD vector of single-precision floating-point variant

Applies when `(S == 1 && Q == 0)`

```
VCMLA{<q>}.F32  <Dd>, <Dn>, <Dm>[0], #<rotate>
```

Encoding for the 128-bit SIMD vector of half-precision floating-point variant

Applies when `(S == 0 && Q == 1)`

```
VCMLA{<q>}.F16  <Qd>, <Qn>, <Dm>[<index>], #<rotate>
```

Encoding for the 128-bit SIMD vector of single-precision floating-point variant

Applies when `(S == 1 && Q == 1)`

```
VCMLA{<q>}.F32  <Qd>, <Qn>, <Dm>[0], #<rotate>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if S == '1' then UInt(M::Vm) else UInt(Vm);
let index : integer = if S == '1' then 0 else UInt(M);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

(FEAT_FCMA)

15 … 8	7	6	5 4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1 1 1 1 1 1 1 0	S	D	rot	Vn	Vd	1 0 0 0	N	Q	M	0 (U)	Vm

Encoding for the 64-bit SIMD vector of half-precision floating-point variant

Applies when `(S == 0 && Q == 0)`

```
VCMLA{<q>}.F16  <Dd>, <Dn>, <Dm>[<index>], #<rotate>
```

Encoding for the 64-bit SIMD vector of single-precision floating-point variant

Applies when `(S == 1 && Q == 0)`

```
VCMLA{<q>}.F32  <Dd>, <Dn>, <Dm>[0], #<rotate>
```

Encoding for the 128-bit SIMD vector of half-precision floating-point variant

Applies when `(S == 0 && Q == 1)`

```
VCMLA{<q>}.F16  <Qd>, <Qn>, <Dm>[<index>], #<rotate>
```

Encoding for the 128-bit SIMD vector of single-precision floating-point variant

Applies when `(S == 1 && Q == 1)`

```
VCMLA{<q>}.F32  <Qd>, <Qn>, <Dm>[0], #<rotate>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if S == '1' then UInt(M::Vm) else UInt(Vm);
let index : integer = if S == '1' then 0 else UInt(M);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

For the “A1 128-bit SIMD vector of half-precision floating-point”, “A1 64-bit SIMD vector of half-precision floating-point”, “T1 128-bit SIMD vector of half-precision floating-point”, and “T1 64-bit SIMD vector of half-precision floating-point” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm” field.

For the “A1 128-bit SIMD vector of single-precision floating-point”, “A1 64-bit SIMD vector of single-precision floating-point”, “T1 128-bit SIMD vector of single-precision floating-point”, and “T1 64-bit SIMD vector of single-precision floating-point” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<index>

Is the element index in the range 0 to 1, encoded in the “M” field.

<rotate>

Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    let operand1 : bits(64) = D(n+r);
    let operand2 : bits(64) = Din(m);
    let operand3 : bits(64) = D(d+r);
    for e = 0 to (elements DIV 2)-1 do
        var element1 : bits(esize);
        var element2 : bits(esize);
        var element3 : bits(esize);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        var element4 : bits(esize);
        case rot of
            when '00' =>
                element1 = operand2[(index*2)*:esize];
                element2 = operand1[(e*2)*:esize];
                element3 = operand2[(index*2+1)*:esize];
                element4 = operand1[(e*2)*:esize];
            when '01' =>
                element1 = FPNeg{esize}(operand2[(index*2+1)*:esize], fpcr);
                element2 = operand1[(e*2+1)*:esize];
                element3 = operand2[(index*2)*:esize];
                element4 = operand1[(e*2+1)*:esize];
            when '10' =>
                element1 = FPNeg{esize}(operand2[(index*2)*:esize], fpcr);
                element2 = operand1[(e*2)*:esize];
                element3 = FPNeg{esize}(operand2[(index*2+1)*:esize], fpcr);
                element4 = operand1[(e*2)*:esize];
            when '11' =>
                element1 = operand2[(index*2+1)*:esize];
                element2 = operand1[(e*2+1)*:esize];
                element3 = FPNeg{esize}(operand2[(index*2)*:esize], fpcr);
                element4 = operand1[(e*2+1)*:esize];
        end;
        let result1 : bits(esize) = FPMulAdd{}(operand3[(e*2)*:esize],
                                               element2,element1, fpcr);
        let result2 : bits(esize) = FPMulAdd{}(operand3[(e*2+1)*:esize],
                                               element4,element3,fpcr);
        D(d+r)[(e*2)*:esize] = result1;
        D(d+r)[(e*2+1)*:esize] = result2;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.54 VCMP

Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent `VMRS` instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

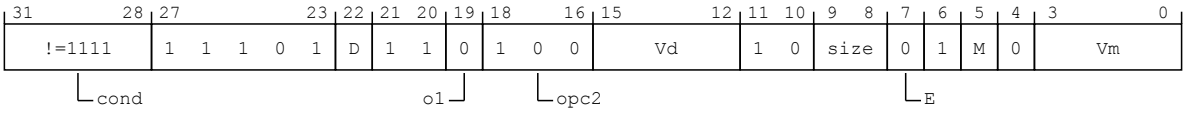

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMP{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMP{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMP{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–28	27–23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1	1	0	1	0	1	Vd	1	0	size		0	1	(0)	0	(0) (0) (0) (0)
cond					o1			opc2						E		M		Vm

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMP{<c>}{<q>}.F16  <Sd>, #0.0
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMP{<c>}{<q>}.F32  <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMP{<c>}{<q>}.F64  <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 0 1 1 1 0 1	D	1	1	0	1	0	0	Vd	1	0	size		0	1	M	0	Vm
				o1		opc2							E				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMP{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMP{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMP{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15								7	6	5	4	3	2		0	15		12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd			1	0	size		0	1	(0)	0	(0)	(0)	(0)	(0)
												o1			opc2								E		M					Vm

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMP{<c>}{<q>}.F16  <Sd>, #0.0
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMP{<c>}{<q>}.F32  <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMP{<c>}{<q>}.F64  <Dd>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    var nzcv : bits(4);
    case esize of
        when 16 =>
            let op16 : bits(16) = if with_zero then FPZero{16}('0') else H(m);
            nzcv = FPCompare{16}(H(d), op16, quiet_nan_exc, fpcr);
        when 32 =>
            let op32 : bits(32) = if with_zero then FPZero{32}('0') else S(m);
            nzcv = FPCompare{32}(S(d), op32, quiet_nan_exc, fpcr);
        when 64 =>
            let op64 : bits(64) = if with_zero then FPZero{64}('0') else D(m);
            nzcv = FPCompare{64}(D(d), op64, quiet_nan_exc, fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    FPSCR()[31:28] = nzcv; // FPSCR.[N,Z,C,V] set to nzcv
end;
```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the FPSCR condition flags to N=0, Z=0, C=1, and V=1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.55 VCMPE

Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent `VMRS` instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–28	27–23	22	21–20	19	18–16	15–12	11–10	9–8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1 1	0	1 0 0	Vd	1 0	size	1	1	M	0	Vm
cond				o1	opc2				E				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMPE{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMPE{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMPE{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–28	27–23	22	21	20	19	18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1	1	0	1	0	1	Vd	1	0	size		1	1	(0)	0	(0) (0) (0) (0)
cond					o1			opc2						E		M		Vm

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMPE{<c>}{<q>}.F16  <Sd>, #0.0
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMPE{<c>}{<q>}.F32  <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMPE{<c>}{<q>}.F64  <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 0 1 1 1 0 1	D	1	1	0	1	0	0	Vd	1	0	size		1	1	M	0	Vm
				o1		opc2							E				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMPE{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMPE{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMPE{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							7	6	5	4	3	2		0	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd		1	0	size		1	1	(0)	0	(0)	(0)	(0)	(0)
												o1			opc2							E		M				Vm

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCMPE{<c>}{<q>}.F16  <Sd>, #0.0
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCMPE{<c>}{<q>}.F32  <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCMPE{<c>}{<q>}.F64  <Dd>, #0.0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    var nzcv : bits(4);
    case esize of
        when 16 =>
            let op16 : bits(16) = if with_zero then FPZero{16}('0') else H(m);
            nzcv = FPCompare{16}(H(d), op16, quiet_nan_exc, fpcr);
        when 32 =>
            let op32 : bits(32) = if with_zero then FPZero{32}('0') else S(m);
            nzcv = FPCompare{32}(S(d), op32, quiet_nan_exc, fpcr);
        when 64 =>
            let op64 : bits(64) = if with_zero then FPZero{64}('0') else D(m);
            nzcv = FPCompare{64}(D(d), op64, quiet_nan_exc, fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    FPSCR()[31:28] = nzcv; // FPSCR.[N,Z,C,V] set to nzcv
end;
```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the FPSCR condition flags to N=0, Z=0, C=1, and V=1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.56 VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 1 0 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCNT{<c>}{<q>}.8  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCNT{<c>}{<q>}.8  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8;
let elements : integer = 8;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 1 0 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCNT{<c>}{<q>}.8  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCNT{<c>}{<q>}.8  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8;
let elements : integer = 8;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = BitCount(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.57 VCVT (from single-precision to BFloat16, Advanced SIMD)

Vector Convert from single-precision to BFloat16 converts each 32-bit element in a vector from single-precision floating-point to BFloat16 format, and writes the result into a second vector. The result vector elements are half the width of the source vector elements.

Unlike the BFloat16 multiplication instructions, this instruction uses the Round to Nearest rounding mode, and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

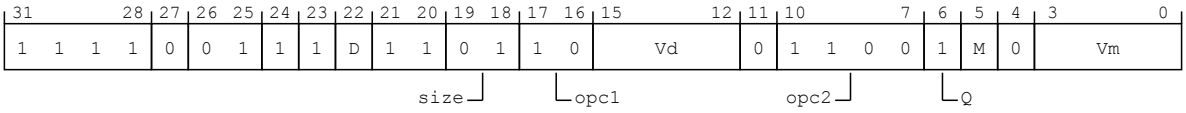

Encoding

```
VCVT{<c>}{<q>}.BF16.F32  <Dd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AA32BF16)

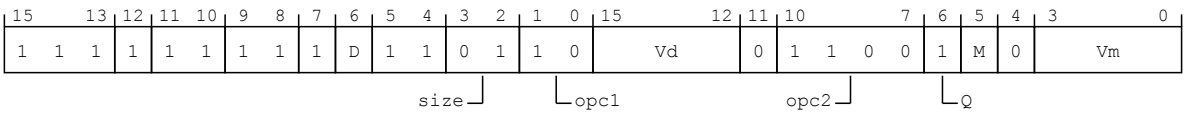

Encoding

```
VCVT{<c>}{<q>}.BF16.F32  <Dd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
var operand : bits(128);
var result : bits(64);

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();

    operand = Q(m>>1);
    for e = 0 to 3 do
        let op : bits(32) = operand[e*:32];
        result[e*:16] = FPConvertBF(op, fpcr);
    end;
    D(d) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.58 VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

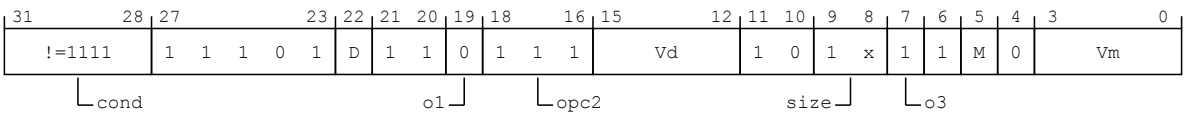

Encoding for the Single-precision to double-precision variant

Applies when `(size == 10)`

```
VCVT{<c>}{<q>}.F64.F32  <Dd>, <Sm>
```

Encoding for the Double-precision to single-precision variant

Applies when `(size == 11)`

```
VCVT{<c>}{<q>}.F32.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let double_to_single : boolean = (size == '11');
let d : integer = if double_to_single then UInt(Vd::D) else UInt(D::Vd);
let m : integer = if double_to_single then UInt(M::Vm) else UInt(Vm::M);
```

T1

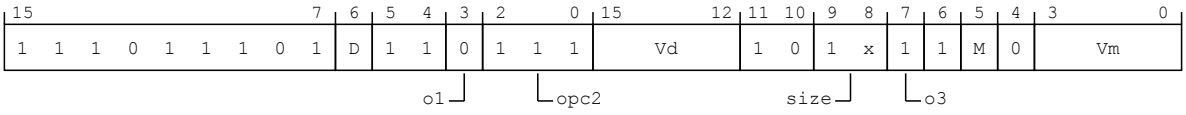

Encoding for the Single-precision to double-precision variant

Applies when `(size == 10)`

```
VCVT{<c>}{<q>}.F64.F32  <Dd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision to single-precision variant

Applies when `(size == 11)`

```
VCVT{<c>}{<q>}.F32.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let double_to_single : boolean = (size == '11');
let d : integer = if double_to_single then UInt(Vd::D) else UInt(D::Vd);
let m : integer = if double_to_single then UInt(M::Vm) else UInt(Vm::M);
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    if double_to_single then
        S(d) = FPConvert{32, 64}(D(m), EffectiveFPCR());
    else
        D(d) = FPConvert{64, 32}(S(m), EffectiveFPCR());
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.59 VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

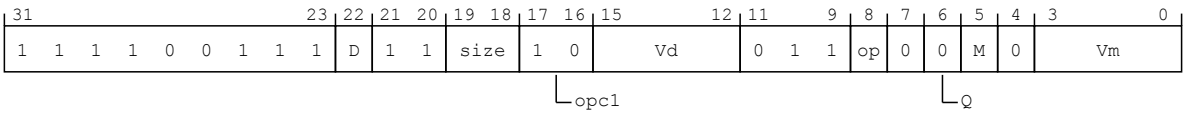

Encoding for the Half-precision to single-precision variant

Applies when `(op == 1)`

```
VCVT{<c>}{<q>}.F32.F16  <Qd>, <Dm>
```

Encoding for the Single-precision to half-precision variant

Applies when `(op == 0)`

```
VCVT{<c>}{<q>}.F16.F32  <Dd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '01' then Undefined(); end;
if op == '1' && Vd[0] == '1' then Undefined(); end;
if op == '0' && Vm[0] == '1' then Undefined(); end;
let half_to_single : boolean = (op == '1');
let esize : integer{} = 16;
let elements : integer = 4;
let m : integer = UInt(M::Vm);
let d : integer = UInt(D::Vd);
```

T1

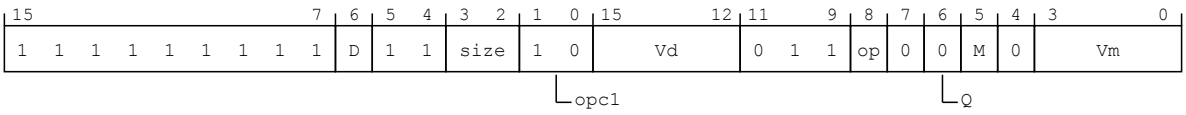

Encoding for the Half-precision to single-precision variant

Applies when `(op == 1)`

```
VCVT{<c>}{<q>}.F32.F16  <Qd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision to half-precision variant

Applies when `(op == 0)`

```
VCVT{<c>}{<q>}.F16.F32  <Dd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '01' then Undefined(); end;
if op == '1' && Vd[0] == '1' then Undefined(); end;
if op == '0' && Vm[0] == '1' then Undefined(); end;
let half_to_single : boolean = (op == '1');
let esize : integer{} = 16;
let elements : integer = 4;
let m : integer = UInt(M::Vm);
let d : integer = UInt(D::Vd);
```

Assembler Symbols

<c>

For the “A1 Half-precision to single-precision” and “A1 Single-precision to half-precision” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Half-precision to single-precision” and “T1 Single-precision to half-precision” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for e = 0 to elements-1 do
        if half_to_single then
            Q(d>>1)[e*:32] = FPConvert{32, 16}(Din(m)[e*:16], fpcr);
        else
            D(d)[e*:16] = FPConvert{16, 32}(Qin(m>>1)[e*:32], fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.60 VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–9	8–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 1	Vd	0 1 1	op	Q	M	0	Vm
				opc1							

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let to_integer : boolean = (op[1] == '1');
let unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–9	8–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 1	Vd	0 1 1	op	Q	M	0	Vm
				opc1							

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let to_integer : boolean = (op[1] == '1');
let unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt1>

Is the data type for the elements of the destination vector, encoded in “size:op”:

size	op	<dt1>
01	0x	F16
01	10	S16
01	11	U16
10	0x	F32
10	10	S32
10	11	U32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt2>

Is the data type for the elements of the source vector, encoded in “size:op”:

size	op	<dt2>
01	00	S16
01	01	U16
01	1x	F16
10	00	S32
10	01	U32
10	1x	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    var result : bits(esize);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(m+r)[e*:esize];
            if to_integer then
                result = FPToFixed{esize, esize}(operand1, 0, unsigned, fpcr, FPRounding_ZERO);
            else
                result = FixedToFP{esize, esize}(operand1, 0, unsigned, fpcr, FPRounding_TIEEVEN);
            end;
            D(d+r)[e*:esize] = result;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.61 VCVT (floating-point to integer, floating-point)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

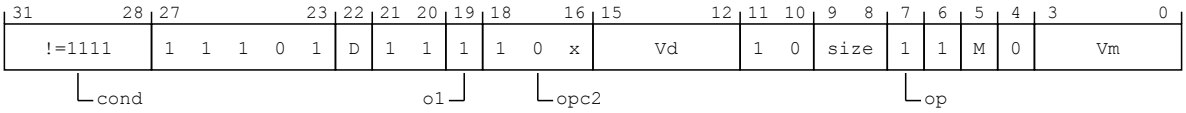

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 100 && size == 01)`

```
VCVT{<c>}{<q>}.U32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 101 && size == 01)`

```
VCVT{<c>}{<q>}.S32.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 100 && size == 10)`

```
VCVT{<c>}{<q>}.U32.F32  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 101 && size == 10)`

```
VCVT{<c>}{<q>}.S32.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 100 && size == 11)`

```
VCVT{<c>}{<q>}.U32.F64  <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 101 && size == 11)`

```
VCVT{<c>}{<q>}.S32.F64  <Sd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15							7	6	5	4	3	2		0	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	1	1	1	0	1	D	1	1	1	1	0	x	Vd		1	0	size		1	1	M	0	Vm		
												o1		opc2								op						

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 100 && size == 01)`

```
VCVT{<c>}{<q>}.U32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 101 && size == 01)`

```
VCVT{<c>}{<q>}.S32.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 100 && size == 10)`

```
VCVT{<c>}{<q>}.U32.F32  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 101 && size == 10)`

```
VCVT{<c>}{<q>}.S32.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 100 && size == 11)`

```
VCVT{<c>}{<q>}.U32.F64  <Sd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 101 && size == 11)`

```
VCVT{<c>}{<q>}.S32.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Floating-point data-processing for the T32 instruction set, or Floating-point data-processing for the A32 instruction set.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16 =>
                S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 16 =>
                H(d) = FixedToFP{16, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.62 VCVT (integer to floating-point, floating-point)

Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the FPSCR, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–23	22	21–20	19	18–16	15–12	11–10	9–8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1 1	1	0 0 0	Vd	1 0	size	op	1	M	0	Vm
cond				o1	opc2								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVT{<c>}{<q>}.F16.<dt>  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVT{<c>}{<q>}.F32.<dt>  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVT{<c>}{<q>}.F64.<dt>  <Dd>, <Sm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2		0	15			12	11	10	9		8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	1	D	1	1	1	0	0	0	Vd				1	0	size			op	1	M	0	Vm			
												o1			opc2																	

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVT{<c>}{<q>}.F16.<dt>  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVT{<c>}{<q>}.F32.<dt>  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVT{<c>}{<q>}.F64.<dt>  <Dd>, <Sm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Floating-point data-processing for the T32 instruction set, or Floating-point data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the operand, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16 =>
                S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 16 =>
                H(d) = FixedToFP{16, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.63 VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

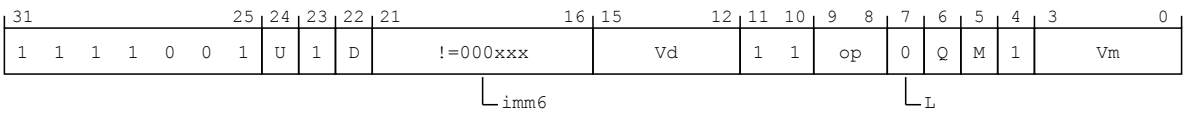

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Dd>, <Dm>, #<fbits>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if op[1] == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if op[1] == '0' && imm6 == '10xxxx' then Undefined(); end;
if imm6 == '0xxxxx' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let to_fixed : boolean = (op[0] == '1');
let frac_bits : integer = 64 - UInt(imm6);
let unsigned : boolean = (U == '1');
let esize : integer{} = 16 << UInt(op[1]);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

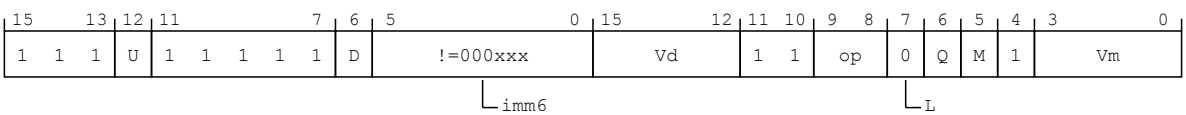

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Dd>, <Dm>, #<fbits>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVT{<c>}{<q>}.<dt1>.<dt2>  <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if op[1] == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if op[1] == '0' && imm6 == '10xxxx' then Undefined(); end;
if imm6 == '0xxxxx' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let to_fixed : boolean = (op[0] == '1');
let frac_bits : integer = 64 - UInt(imm6);
let unsigned : boolean = (U == '1');
let esize : integer{} = 16 << UInt(op[1]);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt1>

Is the data type for the elements of the destination vector, encoded in “op:U”:

op	U	<dt1>
00	x	F16
01	0	S16
01	1	U16
10	x	F32
11	0	S32
11	1	U32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt2>

Is the data type for the elements of the source vector, encoded in “op:U”:

op	U	<dt2>
00	0	S16
00	1	U16
01	x	F16
10	0	S32
10	1	U32
11	x	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<fbits>

The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- (64 - `<fbits>`) is encoded in imm6.

An assembler can permit an `<fbits>` value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see VCVT (between floating-point and integer, Advanced SIMD).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : bits(esize);
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(m+r)[e*:esize];
            if to_fixed then
                result = FPToFixed{esize, esize}(operand1, frac_bits, unsigned, fpcr,
                                                 FPRounding_ZERO);
            else
                result = FixedToFP{esize, esize}(operand1, frac_bits, unsigned, fpcr,
                                                 FPRounding_TIEEVEN);
            end;
            D(d+r)[e*:esize] = result;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.64 VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–23	22	21–20	19	18	17	16	15–12	11–10	9–8	7	6	5	4	3–0
!=1111	1 1 1 0 1	D	1 1	1	op	1	U	Vd	1 0	sf	sx	1	i	0	imm4
cond				o1											

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(op == 0 && sf == 01)`

```
VCVT{<c>}{<q>}.F16.<dt>  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(op == 1 && sf == 01)`

```
VCVT{<c>}{<q>}.<dt>.F16  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when `(op == 0 && sf == 10)`

```
VCVT{<c>}{<q>}.F32.<dt>  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when `(op == 1 && sf == 10)`

```
VCVT{<c>}{<q>}.<dt>.F32  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when `(op == 0 && sf == 11)`

```
VCVT{<c>}{<q>}.F64.<dt>  <Ddm>, <Ddm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when `(op == 1 && sf == 11)`

```
VCVT{<c>}{<q>}.<dt>.F64  <Ddm>, <Ddm>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if sf == '00' || (sf == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if sf == '01' && cond != '1110' then UnpredictableProcedure(); end;
let to_fixed : boolean = (op == '1');
let unsigned : boolean = (U == '1');
let size : integer{} = if sx == '0' then 16 else 32;
let frac_bits : integer = size - UInt(imm4::i);
let fp_size : integer = 8 << UInt(sf);
let d : integer = if sf == '11' then UInt(D::Vd) else UInt(Vd::D);
if frac_bits < 0 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15								7	6	5	4	3	2	1	0	15			12	11	10	9		8	7	6	5	4	3				0
1	1	1	0	1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	sf			sx	1	i	0	imm4				
												o1																					

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(op == 0 && sf == 01)`

```
VCVT{<c>}{<q>}.F16.<dt>  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(op == 1 && sf == 01)`

```
VCVT{<c>}{<q>}.<dt>.F16  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when `(op == 0 && sf == 10)`

```
VCVT{<c>}{<q>}.F32.<dt>  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when `(op == 1 && sf == 10)`

```
VCVT{<c>}{<q>}.<dt>.F32  <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when `(op == 0 && sf == 11)`

```
VCVT{<c>}{<q>}.F64.<dt>  <Ddm>, <Ddm>, #<fbits>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision scalar variant

Applies when `(op == 1 && sf == 11)`

```
VCVT{<c>}{<q>}.<dt>.F64  <Ddm>, <Ddm>, #<fbits>
```

Decode for all variants of this encoding

```
if sf == '00' || (sf == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if sf == '01' && InITBlock() then UnpredictableProcedure(); end;
let to_fixed : boolean = (op == '1');
let unsigned : boolean = (U == '1');
let size : integer{} = if sx == '0' then 16 else 32;
let frac_bits : integer = size - UInt(imm4::i);
let fp_size : integer = 8 << UInt(sf);
let d : integer = if sf == '11' then UInt(D::Vd) else UInt(Vd::D);
if frac_bits < 0 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VCVT (between floating-point and fixed-point).

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the fixed-point number, encoded in “U:sx”:

U	sx	<dt>
0	0	S16
0	1	S32
1	0	U16
1	1	U32

<Sdm>

Is the 32-bit name of the SIMD&FP destination and source register, encoded in the “Vd:D” field.

<fbits>

The number of fraction bits in the fixed-point number:

- If `<dt>` is `S16` or `U16`, `<fbits>` must be in the range 0-16. (16 - `<fbits>`) is encoded in `[imm4, i]`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If `<dt>` is `S32` or `U32`, `<fbits>` must be in the range 1-32. (32 - `<fbits>`) is encoded in `[imm4, i]`.

<Ddm>

Is the 64-bit name of the SIMD&FP destination and source register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if to_fixed then
        var result : bits(size);
        case fp_size of
            when 16 =>
                result = FPToFixed{size, 16}(H(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                S(d) = Extend{32}(result, unsigned);
            when 32 =>
                result = FPToFixed{size, 32}(S(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                S(d) = Extend{32}(result, unsigned);
            when 64 =>
                result = FPToFixed{size, 64}(D(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                D(d) = Extend{64}(result, unsigned);
        end;
    else
        case fp_size of
            when 16 =>
                H(d) = FixedToFP{16, size}(S(d)[size-1:0], frac_bits, unsigned, fpcr,
                                           FPRounding_TIEEVEN);
            when 32 =>
                S(d) = FixedToFP{32, size}(S(d)[size-1:0], frac_bits, unsigned, fpcr,
                                           FPRounding_TIEEVEN);
            when 64 =>
                D(d) = FixedToFP{64, size}(D(d)[size-1:0], frac_bits, unsigned, fpcr,
                                           FPRounding_TIEEVEN);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.65 VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

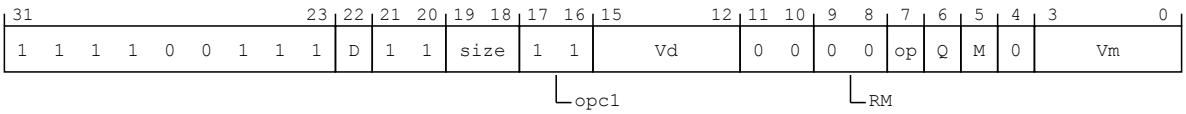

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVTA{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTA{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

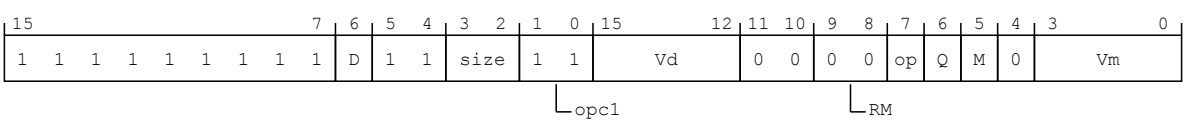

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVTA{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTA{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                   0, unsigned, fpcr, rounding);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.66 VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–19	18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	0 0	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTA{<q>}.<dt>.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTA{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTA{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15–7	6	5–3	2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	0 0	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVTA{<q>}.<dt>.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTA{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTA{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
    when 32 =>
        S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
    when 64 =>
        S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.67 VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

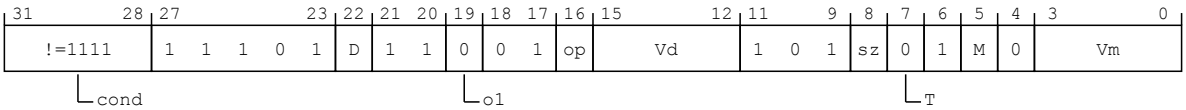

Encoding for the Half-precision to single-precision variant

Applies when `(op == 0 && sz == 0)`

```
VCVTB{<c>}{<q>}.F32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision to double-precision variant

Applies when `(op == 0 && sz == 1)`

```
VCVTB{<c>}{<q>}.F64.F16  <Dd>, <Sm>
```

Encoding for the Single-precision to half-precision variant

Applies when `(op == 1 && sz == 0)`

```
VCVTB{<c>}{<q>}.F16.F32  <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when `(op == 1 && sz == 1)`

```
VCVTB{<c>}{<q>}.F16.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			9	8	7	6	5	4	3				0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1		sz	0	1	M	0	Vm			
												o1													T							

Encoding for the Half-precision to single-precision variant

Applies when `(op == 0 && sz == 0)`

```
VCVTB{<c>}{<q>}.F32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision to double-precision variant

Applies when `(op == 0 && sz == 1)`

```
VCVTB{<c>}{<q>}.F64.F16  <Dd>, <Sm>
```

Encoding for the Single-precision to half-precision variant

Applies when `(op == 1 && sz == 0)`

```
VCVTB{<c>}{<q>}.F16.F32  <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when `(op == 1 && sz == 1)`

```
VCVTB{<c>}{<q>}.F16.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var hp : bits(16);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if convert_from_half then
        hp = S(m)[lowbit+15:lowbit];
        if uses_double then
            D(d) = FPConvert{64, 16}(hp, fpcr);
        else
            S(d) = FPConvert{32, 16}(hp, fpcr);
        end;
    else
        if uses_double then
            hp = FPConvert{16, 64}(D(m), fpcr);
        else
            hp = FPConvert{16, 32}(S(m), fpcr);
        end;
        S(d)[lowbit+15:lowbit] = hp;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.68 VCVTB (BFloat16)

Converts from a single-precision value to a BFloat16 value in the bottom half of a single-precision register

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

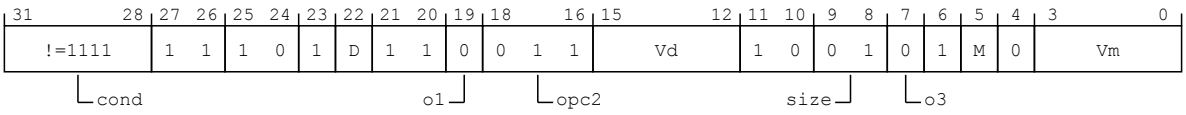

Encoding

```
VCVTB{<c>}{<q>}.BF16.F32  <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1

(FEAT_AA32BF16)

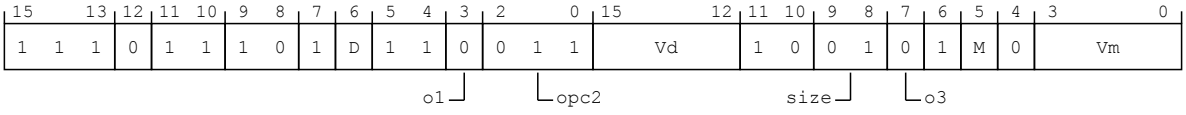

Encoding

```
VCVTB{<c>}{<q>}.BF16.F32  <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S(d)[15:0] = FPConvertBF(S(m), EffectiveFPCR());
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.69 VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 1 (opc1)	Vd	0 0	1 1 (RM)	op	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVTM{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTM{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 1 (opc1)	Vd	0 0	1 1 (RM)	op	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVTM{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTM{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                   0, unsigned, fpcr, rounding);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.70 VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–19	18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	1 1	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTM{<q>}.<dt>.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTM{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTM{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15–7	6	5–3	2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	1 1	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTM{<q>}.<dt>.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTM{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTM{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
    when 32 =>
        S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
    when 64 =>
        S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.71 VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 1 (opc1)	Vd	0 0	0 1 (RM)	op	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVTN{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTN{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 1 (opc1)	Vd	0 0	0 1 (RM)	op	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVTN{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTN{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                   0, unsigned, fpcr, rounding);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.72 VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–19	18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	0 1	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTN{<q>}.<dt>.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTN{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTN{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15–7	6	5–3	2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	0 1	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTN{<q>}.<dt>.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTN{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTN{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
    when 32 =>
        S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
    when 64 =>
        S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.73 VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 1 (opc1)	Vd	0 0	1 0 (RM)	op	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VCVTP{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTP{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 1 (opc1)	Vd	0 0	1 0 (RM)	op	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVTP{<q>}.<dt>.<dt2>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VCVTP{<q>}.<dt>.<dt2>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                   0, unsigned, fpcr, rounding);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.74 VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–19	18	17–16	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	1 0	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTP{<q>}.<dt>.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTP{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTP{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15–7	6	5–3	2	1–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0 1	D	1 1 1	1	1 0	Vd	1 0	size	op	1	M	0	Vm
			op1	RM								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VCVTP{<q>}.<dt>.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VCVTP{<q>}.<dt>.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VCVTP{<q>}.<dt>.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
    when 32 =>
        S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
    when 64 =>
        S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.75 VCVTR

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the FPSCR and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

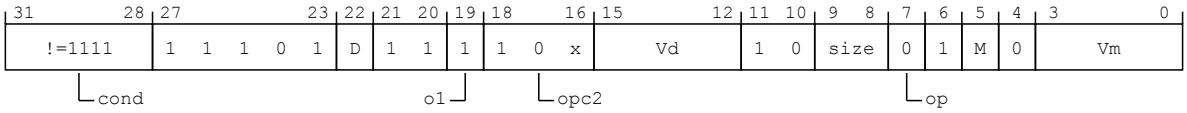

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 100 && size == 01)`

```
VCVTR{<c>}{<q>}.U32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 101 && size == 01)`

```
VCVTR{<c>}{<q>}.S32.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 100 && size == 10)`

```
VCVTR{<c>}{<q>}.U32.F32  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 101 && size == 10)`

```
VCVTR{<c>}{<q>}.S32.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 100 && size == 11)`

```
VCVTR{<c>}{<q>}.U32.F64  <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 101 && size == 11)`

```
VCVTR{<c>}{<q>}.S32.F64  <Sd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15								7	6	5	4	3	2		0	15			12	11	10	9		8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	1	D	1	1	1	1	0	x	Vd				1	0	size			0	1	M	0	Vm			
												o1			opc2										op							

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 100 && size == 01)`

```
VCVTR{<c>}{<q>}.U32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(opc2 == 101 && size == 01)`

```
VCVTR{<c>}{<q>}.S32.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 100 && size == 10)`

```
VCVTR{<c>}{<q>}.U32.F32  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 101 && size == 10)`

```
VCVTR{<c>}{<q>}.S32.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 100 && size == 11)`

```
VCVTR{<c>}{<q>}.U32.F64  <Sd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 101 && size == 11)`

```
VCVTR{<c>}{<q>}.S32.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Floating-point data-processing for the T32 instruction set, or Floating-point data-processing for the A32 instruction set.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16 =>
                S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 16 =>
                H(d) = FixedToFP{16, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.76 VCVTT

Convert to or from a half-precision value in the top half of a single-precision register

Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

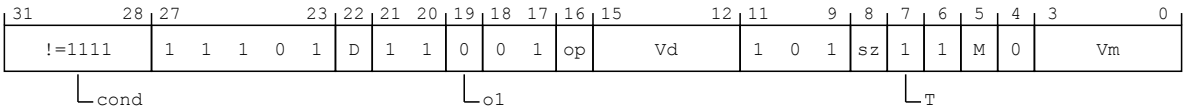

Encoding for the Half-precision to single-precision variant

Applies when `(op == 0 && sz == 0)`

```
VCVTT{<c>}{<q>}.F32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision to double-precision variant

Applies when `(op == 0 && sz == 1)`

```
VCVTT{<c>}{<q>}.F64.F16  <Dd>, <Sm>
```

Encoding for the Single-precision to half-precision variant

Applies when `(op == 1 && sz == 0)`

```
VCVTT{<c>}{<q>}.F16.F32  <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when `(op == 1 && sz == 1)`

```
VCVTT{<c>}{<q>}.F16.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			9	8	7	6	5	4	3				0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1		sz	1	1	M	0	Vm				
												o1													T								

Encoding for the Half-precision to single-precision variant

Applies when `(op == 0 && sz == 0)`

```
VCVTT{<c>}{<q>}.F32.F16  <Sd>, <Sm>
```

Encoding for the Half-precision to double-precision variant

Applies when `(op == 0 && sz == 1)`

```
VCVTT{<c>}{<q>}.F64.F16  <Dd>, <Sm>
```

Encoding for the Single-precision to half-precision variant

Applies when `(op == 1 && sz == 0)`

```
VCVTT{<c>}{<q>}.F16.F32  <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when `(op == 1 && sz == 1)`

```
VCVTT{<c>}{<q>}.F16.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var hp : bits(16);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if convert_from_half then
        hp = S(m)[lowbit+15:lowbit];
        if uses_double then
            D(d) = FPConvert{64, 16}(hp, fpcr);
        else
            S(d) = FPConvert{32, 16}(hp, fpcr);
        end;
    else
        if uses_double then
            hp = FPConvert{16, 64}(D(m), fpcr);
        else
            hp = FPConvert{16, 32}(S(m), fpcr);
        end;
        S(d)[lowbit+15:lowbit] = hp;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.77 VCVTT (BFloat16)

Converts from a single-precision value to a BFloat16 value in the top half of a single-precision register

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

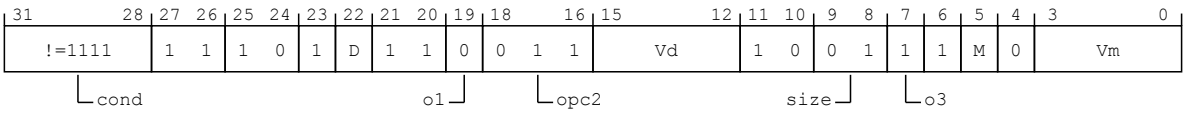

Encoding

```
VCVTT{<c>}{<q>}.BF16.F32  <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1

(FEAT_AA32BF16)

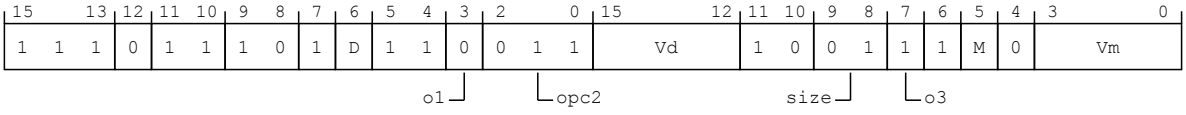

Encoding

```
VCVTT{<c>}{<q>}.BF16.F32  <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S(d)[31:16] = FPConvertBF(S(m), EffectiveFPCR());
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.78 VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

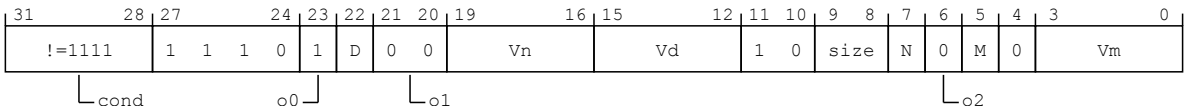

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VDIV{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VDIV{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VDIV{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–8	7	6	5–4	3–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 0 1 1 1 0	1	D	0 0	Vn	Vd	1 0	size	N	0	M	0	Vm
	o0		o1						o2			

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VDIV{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VDIV{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VDIV{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            H(d) = FPDiv{16}(H(n), H(m), fpcr);
        when 32 =>
            S(d) = FPDiv{32}(S(n), S(m), fpcr);
        when 64 =>
            D(d) = FPDiv{64}(D(n), D(m), fpcr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.79 VDOT (vector)

BFloat16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the FPSCR exception status.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

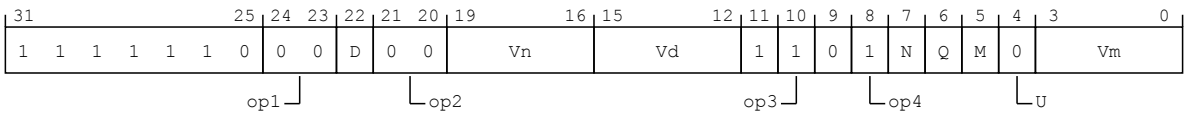

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VDOT{<q>}.BF16  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VDOT{<q>}.BF16  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32BF16)

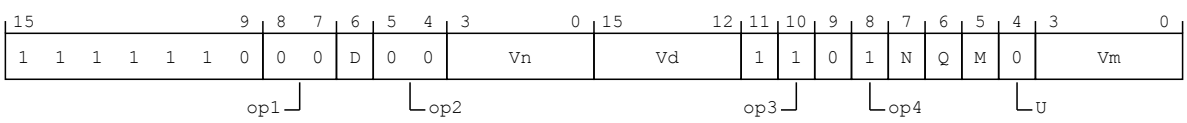

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VDOT{<q>}.BF16  <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VDOT{<q>}.BF16  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

for r = 0 to regs-1 do
    operand1 = Din(n+r);
    operand2 = Din(m+r);
    result = Din(d+r);
    for e = 0 to 1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
        let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
        let sum : bits(32) = FPAdd_BF16(BFMulH(elt1_a, elt2_a, fpcr),
                                        BFMulH(elt1_b, elt2_b, fpcr), fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result[e*:32] = FPAdd_BF16(result[e*:32], sum, fpcr);
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.80 VDOT (by element)

BFloat16 floating-point indexed dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the indexed pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BFloat16 values in the first source vector. The instruction does not update the FPSCR exception status.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

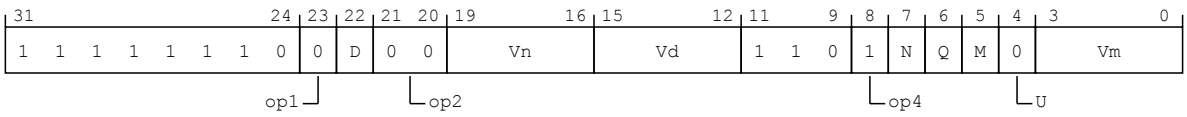

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VDOT{<q>}.BF16  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VDOT{<q>}.BF16  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32BF16)

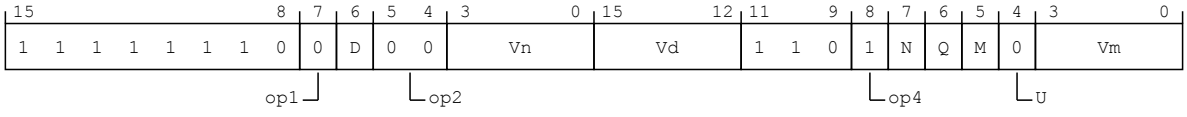

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VDOT{<q>}.BF16  <Dd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VDOT{<q>}.BF16  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm” field.

<index>

Is the element index in the range 0 to 1, encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

operand2 = Din(m);
for r = 0 to regs-1 do
    operand1 = Din(n+r);
    result = Din(d+r);
    for e = 0 to 1 do
        let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
        let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
        let elt2_a : bits(16) = operand2[(2 * i + 0)*:16];
        let elt2_b : bits(16) = operand2[(2 * i + 1)*:16];
        let sum : bits(32) = FPAdd_BF16(BFMulH(elt1_a, elt2_a, fpcr),
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                        BFMulH(elt1_b, elt2_b, fpcr), fpcr);
        result[e*:32] = FPAdd_BF16(result[e*:32], sum, fpcr);
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.81 VDUP (general-purpose register)

Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

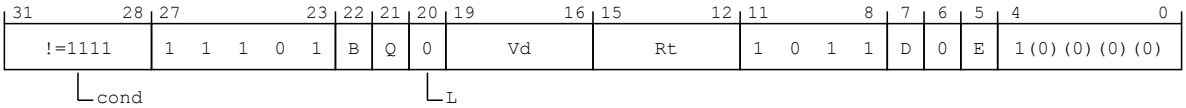

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VDUP{<c>}{<q>}.<size>  <Dd>, <Rt>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VDUP{<c>}{<q>}.<size>  <Qd>, <Rt>
```

Decode for all variants of this encoding

```
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
let regs : integer = if Q == '0' then 1 else 2;
if B::E == '11' then Undefined(); end;
let esize : integer{} = 32 >> UInt(B::E);
let elements : integer = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

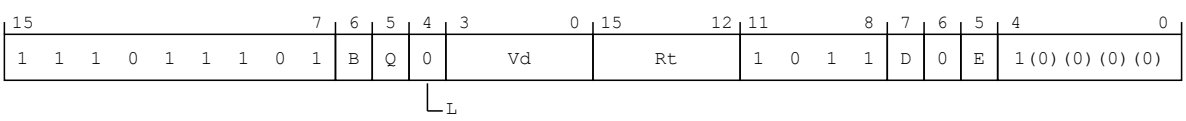

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VDUP{<c>}{<q>}.<size>  <Dd>, <Rt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VDUP{<c>}{<q>}.<size>  <Qd>, <Rt>
```

Decode for all variants of this encoding

```
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
let regs : integer = if Q == '0' then 1 else 2;
if B::E == '11' then Undefined(); end;
let esize : integer{} = 32 >> UInt(B::E);
let elements : integer = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields. Arm strongly recommends that any `VDUP` instruction is unconditional, see Conditional execution.

<q>

See Standard assembler syntax fields.

<size>

The data size for the elements of the destination vector. It must be one of:

8 Encoded as [b, e] = `0b10`.
16 Encoded as [b, e] = `0b01`.
32 Encoded as [b, e] = `0b00`.

<Dd>

The destination vector for a doubleword operation.

<Rt>

The Arm source register.

<Qd>

The destination vector for a quadword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let scalar : bits(esize) = R(t)[esize-1:0];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = scalar;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.82 VDUP (scalar)

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector.

The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–16	15–12	11–10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	imm4	Vd	1 1	0 0 0	Q	M	0	Vm
						opc				

Encoding

Applies when `(Q == 0)`

```
VDUP{<c>}{<q>}.<size>  <Dd>, <Dm[x]>
```

Encoding

Applies when `(Q == 1)`

```
VDUP{<c>}{<q>}.<size>  <Qd>, <Dm[x]>
```

Decode for all variants of this encoding

```
if imm4 == 'x000' then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let lsb : integer{} = LowestSetBit(imm4[2:0]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(imm4[3:lsb+1]);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer{} = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–0	15–12	11–10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	imm4	Vd	1 1	0 0 0	Q	M	0	Vm
						opc				

Encoding

Applies when `(Q == 0)`

```
VDUP{<c>}{<q>}.<size>  <Dd>, <Dm[x]>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

Applies when `(Q == 1)`

```
VDUP{<c>}{<q>}.<size>  <Qd>, <Dm[x]>
```

Decode for all variants of this encoding

```
if imm4 == 'x000' then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let lsb : integer{} = LowestSetBit(imm4[2:0]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(imm4[3:lsb+1]);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer{} = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the "A1 Advanced SIMD duplicate (scalar)" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Advanced SIMD duplicate (scalar)" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

The data size. It must be one of:

8 Encoded as imm4<0> = '1'. imm4<3:1> encodes the index `[x]` of the scalar.
16 Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index `[x]` of the scalar.
32 Encoded as imm4<2:0> = '100'. imm4<3> encodes the index `[x]` of the scalar.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm[x]>

The scalar. For details of how `[x]` is encoded, see the description of `<size>`.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let scalar : bits(esize) = D(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = scalar;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.83 VEOR

Vector Bitwise Exclusive-OR performs a bitwise exclusive-OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

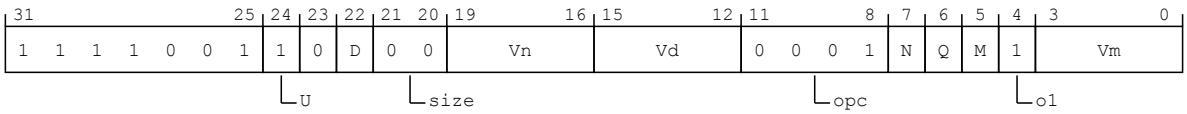

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VEOR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VEOR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

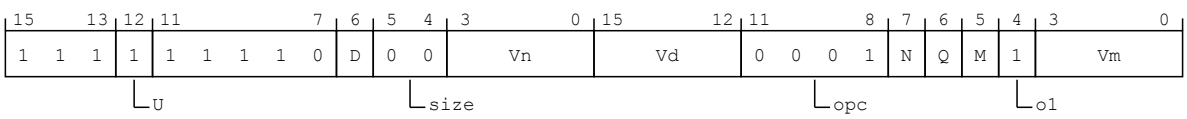

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VEOR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VEOR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) XOR D(m+r);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.84 VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

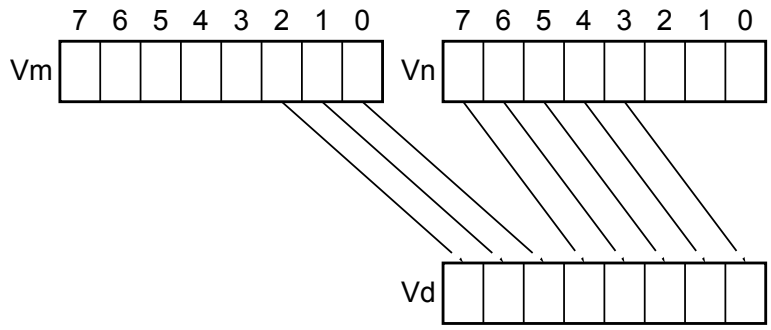

Figure F6-1 VEXT doubleword operation for imm = 3

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VEXT (multibyte elements).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31								23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn		Vd		imm4		N	Q	M	0	Vm	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VEXT{<c>}{<q>}.8  {<Dd>, }<Dn>, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VEXT{<c>}{<q>}.8  {<Qd>, }<Qn>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if Q == '0' && imm4[3] == '1' then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let position : integer{} = 8 * UInt(imm4);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn		Vd		imm4		N	Q	M	0	Vm	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VEXT{<c>}{<q>}.8  {<Dd>, }<Dn>, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VEXT{<c>}{<q>}.8  {<Qd>, }<Qn>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if Q == '0' && imm4[3] == '1' then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let position : integer{} = 8 * UInt(imm4);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the “imm4” field.

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the “imm4” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        Q(d>>1) = (Q(m>>1)::Q(n>>1))[position+127:position];
    else
        D(d) = (D(m)::D(n))[position+63:position];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.85 VEXT (multibyte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

This is a pseudo-instruction of VEXT (byte elements). This means:

- The encodings in this description are named to match the encodings of VEXT (byte elements).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VEXT (byte elements) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1 0 1	D	1 1	Vn	Vd	imm4	N	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VEXT{<c>}{<q>}.<size>  {<Dd>, }<Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8  {<Dd>, }<Dn>, <Dm>, #<imm*(size/8)>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VEXT{<c>}{<q>}.<size>  {<Qd>, }<Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8  {<Qd>, }<Qn>, <Qm>, #<imm*(size/8)>
```

T1

15–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 0 1 1 1 1 1	D	1 1	Vn	Vd	imm4	N	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VEXT{<c>}{<q>}.<size>  {<Dd>, }<Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8  {<Dd>, }<Dn>, <Dm>, #<imm*(size/8)>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VEXT{<c>}{<q>}.<size>  {<Qd>, }<Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8  {<Qd>, }<Qn>, <Qm>, #<imm*(size/8)>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the size of the operation, and can be one of 16 or 32.

For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the size of the operation, and can be one of 16, 32 or 64.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/`<size>`)-1.

For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/`<size>`)-1.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

The description of VEXT (byte elements) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.86 VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

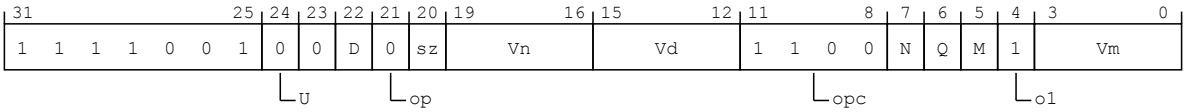

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

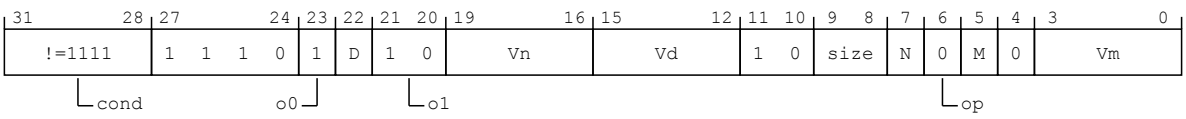

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFMA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFMA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFMA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 0 0	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3			0	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	1	D	1	0	Vn				Vd				1	0	size		N	0	M	0	Vm			
								o0			o1														op						

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFMA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFMA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFMA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var op1elt : bits(esize) = D(n+r)[e*:esize];
                if op1_neg then op1elt = FPNeg{esize}(op1elt, fpcr); end;
                D(d+r)[e*:esize] = FPMulAdd{esize}(D(d+r)[e*:esize],
                                                   op1elt, D(m+r)[e*:esize], fpcr);
            end;
        end;

    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                let op16 : bits(16) = if op1_neg then FPNeg{16}(H(n), fpcr) else H(n);
                H(d) = FPMulAdd{16}(H(d), op16, H(m), fpcr);
            when 32 =>
                let op32 : bits(32) = if op1_neg then FPNeg{32}(S(n), fpcr) else S(n);
                S(d) = FPMulAdd{32}(S(d), op32, S(m), fpcr);
            when 64 =>
                let op64 : bits(64) = if op1_neg then FPNeg{64}(D(n), fpcr) else D(n);
                D(d) = FPMulAdd{64}(D(d), op64, D(m), fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.87 VFMAB, VFMAT (BFloat16, vector)

BFloat16 floating-point widening multiply-add long (vector)

The Bfloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

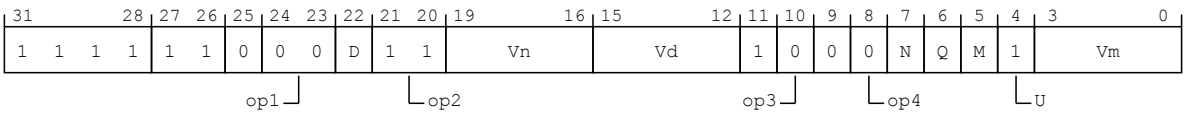

Encoding

```
VFMA<bt>{<q>}.BF16  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

T1

(FEAT_AA32BF16)

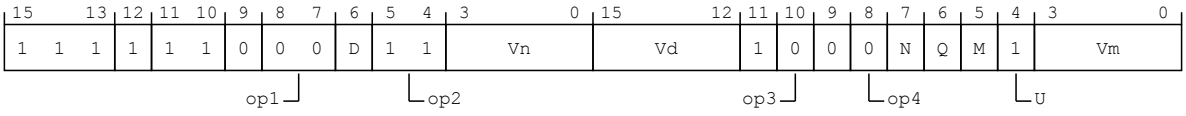

Encoding

```
VFMA<bt>{<q>}.BF16  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<bt>

Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let operand3 : bits(128) = Q(d>>1);
var result : bits(128);

for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[(2 * e + sel)*:16] :: Zeros{16};
    let element2 : bits(32) = operand2[(2 * e + sel)*:16] :: Zeros{16};
    let addend : bits(32) = operand3[e*:32];
    result[e*:32] = FPMulAdd{32}(addend, element1, element2, fpcr);
end;

Q(d>>1) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.88 VFMAB, VFMAT (BFloat16, by scalar)

BFloat16 floating-point widening multiply-add long (by scalar)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and an indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

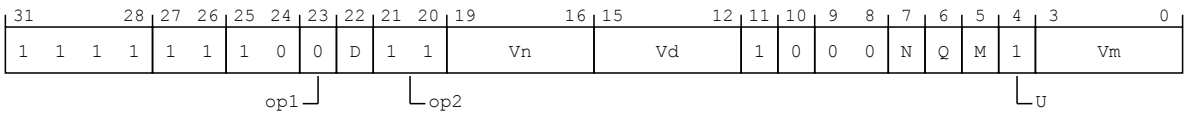

Encoding

```
VFMA<bt>{<q>}.BF16  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm[2:0]);
let i : integer = UInt(M::Vm[3]);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

T1

(FEAT_AA32BF16)

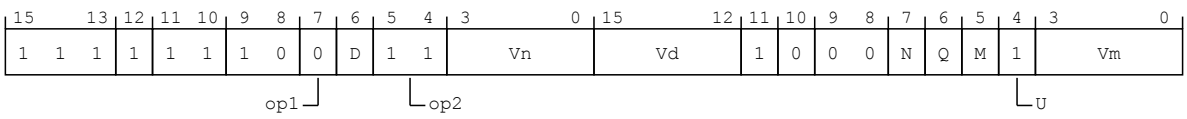

Encoding

```
VFMA<bt>{<q>}.BF16  <Qd>, <Qn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm[2:0]);
let i : integer = UInt(M::Vm[3]);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

Assembler Symbols

<bt>

Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field.

<index>

Is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(64) = D(m);
let operand3 : bits(128) = Q(d>>1);
var result : bits(128);

let element2 : bits(32) = operand2[i*:16] :: Zeros{16};

for e = 0 to elements-1 do
    let element1 : bits(32) = operand1[(2 * e + sel)*:16] :: Zeros{16};
    let addend : bits(32) = operand3[e*:32];
    result[e*:32] = FPMulAdd{32}(addend, element1, element2, fpcr);
end;

Q(d>>1) = result;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.89 VFMAL (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FHM)

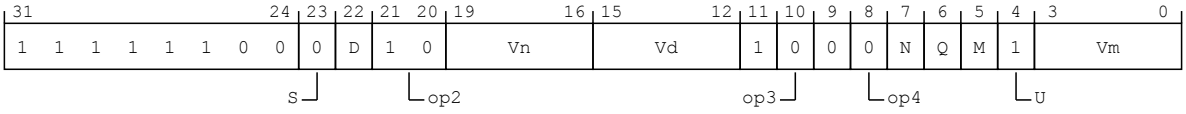

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMAL{<q>}.F16  <Dd>, <Sn>, <Sm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMAL{<q>}.F16  <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FHM)

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn		Vd		1	0	0	0	N	Q	M	1	Vm	
								S		op2						op3			op4				U		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMAL{<q>}.F16  <Dd>, <Sn>, <Sm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMAL{<q>}.F16  <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let regs : integer = 1 << UInt(Q);
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(datasize);
var operand2 : bits(datasize);
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;

for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        element2 = operand2[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.90 VFMAL (by scalar)

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FHM)

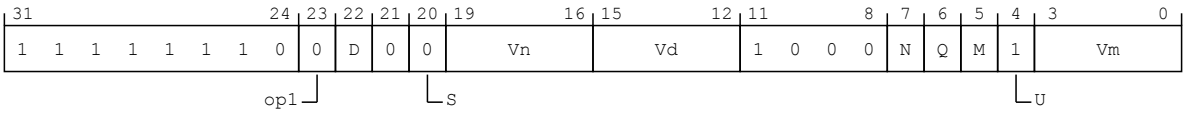

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMAL{<q>}.F16  <Dd>, <Sn>, <Sm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMAL{<q>}.F16  <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FHM)

15–8	7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1 1 1 1 1 0	0	D	0	0	Vn	Vd	1 0 0 0	N	Q	M	1	Vm
	op1			S							U	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMAL{<q>}.F16  <Dd>, <Sn>, <Sm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMAL{<q>}.F16  <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>:M” field.

<index>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the element index in the range 0 to 1, encoded in the “Vm<3>” field.

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(datasize) ;
var operand2 : bits(datasize) ;
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);
let fpcr : FPCR_Type = StandardFPCR();

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;
element2 = operand2[index*:(esize DIV 2)];
for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.91 VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

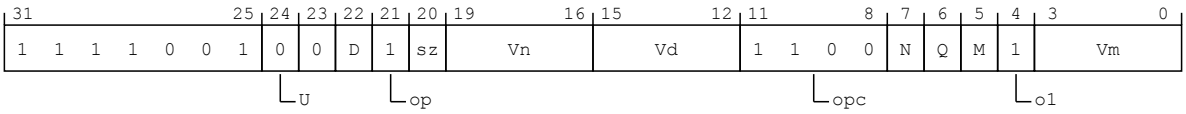

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

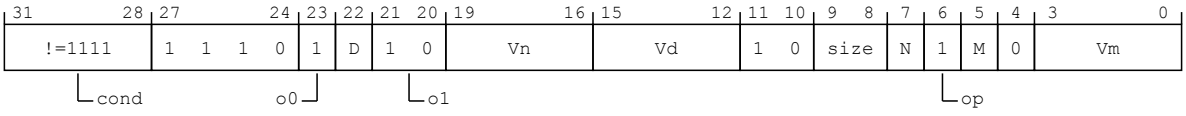

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFMS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFMS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFMS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	1	sz	Vn	Vd	1 1 0 0	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	1	0	Vn		Vd		1	0	size		N	1	M	0	Vm	
								o0			o1										op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFMS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFMS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFMS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var op1elt : bits(esize) = D(n+r)[e*:esize];
                if op1_neg then op1elt = FPNeg{esize}(op1elt, fpcr); end;
                D(d+r)[e*:esize] = FPMulAdd{esize}(D(d+r)[e*:esize],
                                                   op1elt, D(m+r)[e*:esize], fpcr);
            end;
        end;

    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                let op16 : bits(16) = if op1_neg then FPNeg{16}(H(n), fpcr) else H(n);
                H(d) = FPMulAdd{16}(H(d), op16, H(m), fpcr);
            when 32 =>
                let op32 : bits(32) = if op1_neg then FPNeg{32}(S(n), fpcr) else S(n);
                S(d) = FPMulAdd{32}(S(d), op32, S(m), fpcr);
            when 64 =>
                let op64 : bits(64) = if op1_neg then FPNeg{64}(D(n), fpcr) else D(n);
                D(d) = FPMulAdd{64}(D(d), op64, D(m), fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.92 VFMSL (vector)

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FHM)

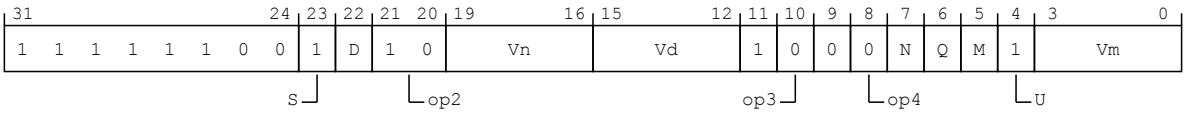

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMSL{<q>}.F16  <Dd>, <Sn>, <Sm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMSL{<q>}.F16  <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FHM)

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	0	0	1	D	1	0	Vn		Vd		1	0	0	0	N	Q	M	1	Vm	
								S		op2						op3			op4				U		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMSL{<q>}.F16  <Dd>, <Sn>, <Sm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMSL{<q>}.F16  <Qd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let regs : integer = 1 << UInt(Q);
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(datasize);
var operand2 : bits(datasize);
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;

for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        element2 = operand2[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.93 VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FHM)

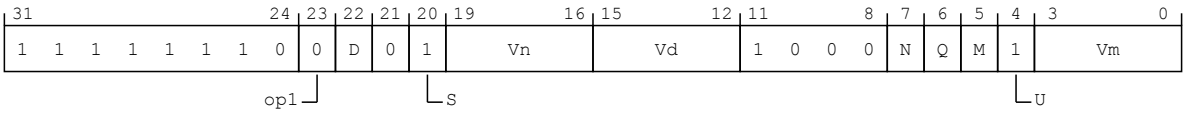

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMSL{<q>}.F16  <Dd>, <Sn>, <Sm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMSL{<q>}.F16  <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FHM)

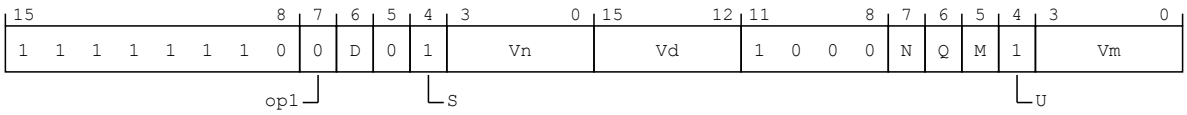

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VFMSL{<q>}.F16  <Dd>, <Sn>, <Sm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VFMSL{<q>}.F16  <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>:M” field.

<index>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the element index in the range 0 to 1, encoded in the “Vm<3>” field.

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(datasize) ;
var operand2 : bits(datasize) ;
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);
let fpcr : FPCR_Type = StandardFPCR();

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;
element2 = operand2[index*:(esize DIV 2)];
for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*:esize] = FPMulAddH(operand3[e*:esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.94 VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

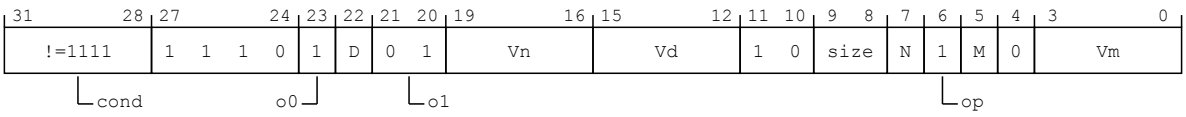

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFNMA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFNMA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFNMA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	0	1	Vn		Vd		1	0	size		N	1	M	0	Vm	
								o0			o1										op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFNMA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFNMA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFNMA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let op16 : bits(16) = if op1_neg then FPNeg{16}(H(n), fpcr) else H(n);
            H(d) = FPMulAdd{16}(FPNeg{16}(H(d), fpcr), op16, H(m), fpcr);
        when 32 =>
            let op32 : bits(32) = if op1_neg then FPNeg{32}(S(n), fpcr) else S(n);
            S(d) = FPMulAdd{32}(FPNeg{32}(S(d), fpcr), op32, S(m), fpcr);
        when 64 =>
            let op64 : bits(64) = if op1_neg then FPNeg{64}(D(n), fpcr) else D(n);
            D(d) = FPMulAdd{64}(FPNeg{64}(D(d), fpcr), op64, D(m), fpcr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.95 VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

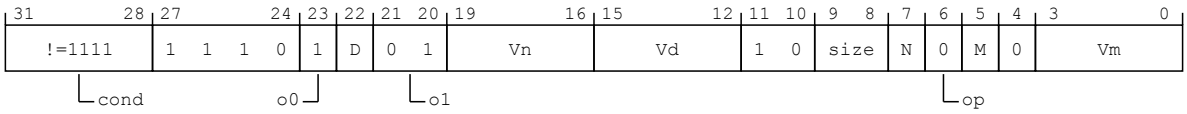

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFNMS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFNMS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFNMS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	0	1	Vn		Vd		1	0	size		N	0	M	0	Vm	
								o0			o1										op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VFNMS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VFNMS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VFNMS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let op16 : bits(16) = if op1_neg then FPNeg{16}(H(n), fpcr) else H(n);
            H(d) = FPMulAdd{16}(FPNeg{16}(H(d), fpcr), op16, H(m), fpcr);
        when 32 =>
            let op32 : bits(32) = if op1_neg then FPNeg{32}(S(n), fpcr) else S(n);
            S(d) = FPMulAdd{32}(FPNeg{32}(S(d), fpcr), op32, S(m), fpcr);
        when 64 =>
            let op64 : bits(64) = if op1_neg then FPNeg{64}(D(n), fpcr) else D(n);
            D(d) = FPMulAdd{64}(FPNeg{64}(D(d), fpcr), op64, D(m), fpcr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.96 VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see VRHADD).

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 0	0	0	N	Q	M	0	Vm
								op					o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VHADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VHADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0	0	0	N	Q	M	0	Vm
								op					o1	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VHADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VHADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if add then element1+element2 else element1-element2) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.97 VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 0	1	0	N	Q	M	0	Vm
								op					o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VHSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VHSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 0	1	0	N	Q	M	0	Vm
								op					o1	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VHSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VHSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if add then element1+element2 else element1-element2) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.98 VINS

Vector move Insertion. This instruction copies the lower 16 bits of the 32-bit source SIMD&FP register into the upper 16 bits of the 32-bit destination SIMD&FP register, while preserving the values in the remaining bits.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FP16)

31–28	27–26	25–23	22	21–16	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1 1	1 1	1 0 1	D	1 1 0 0 0 0	Vd	1	0	1 0	1	1	M	0	Vm
							size		op				

Encoding

```
VINS{<q>}.F16  <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1

(FEAT_FP16)

15–13	12	11–10	9–7	6	5–0	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1	1	1 1	1 0 1	D	1 1 0 0 0 0	Vd	1	0	1 0	1	1	M	0	Vm
								size		op				

Encoding

```
VINS{<q>}.F16  <Sd>, <Sm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    S(d)[31:16] = H(m);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.99 VJCVT

JavaScript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_JSCVT)

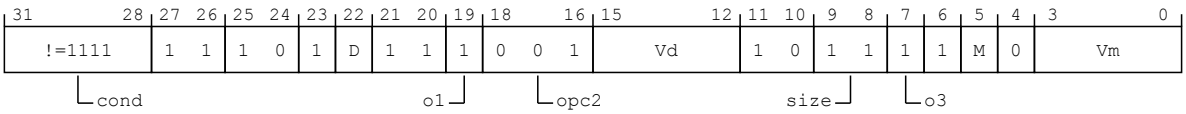

Encoding

```
VJCVT{<q>}.S32.F64  <Sd>, <Dm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_JSCVT)

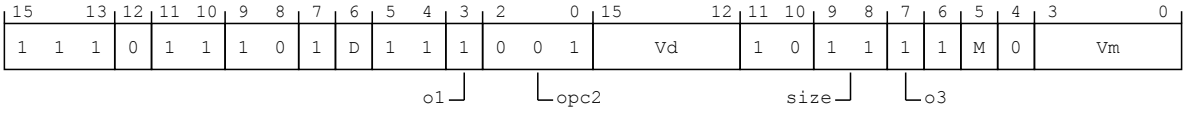

Encoding

```
VJCVT{<q>}.S32.F64  <Sd>, <Dm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fltval : bits(64) = D(m);
var intval : bits(32);
var Z : bit;
(intval, Z) = FPToFixedJS(fltval, EffectiveFPCR());
FPSCR()[31:28] = '0'::Z::'00';
S(d) = intval;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.100 VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	1	0	Rn	Vd	0 0	0 0	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	1	0	Rn	Vd	0 1	0 0	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A3

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	1	0	Rn	Vd	1 0	0 0	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T1

15–7	6	5	4	3–0	15–12	11–10	9–8	7–4	3–0
1 1 1 1 1 0 0 1 1	D	1	0	Rn	Vd	0 0	0 0	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	1	0	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	0	0	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be `{ <Dd>[<index>] }`.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` == **8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` == **16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` == **32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on `<size>`:

`<size>` == **8** Encoded in the “index_align<0>” field as 0.
`<size>` == **16** Encoded in the “index_align<1:0>” field as `0b00`.
`<size>` == **32** Encoded in the “index_align<2:0>” field as `0b000`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size>` == **16** `<align>` is 16, meaning 16-bit alignment, encoded in the “index_align<1:0>” field as `0b01`.
`<size>` == **32** `<align>` is 32, meaning 32-bit alignment, encoded in the “index_align<2:0>” field as `0b011`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index*:(8*ebytes)] = MemU{8*ebytes}(address);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.101 VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

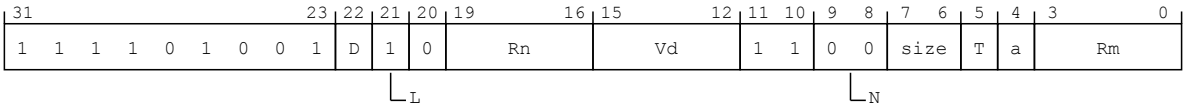

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || (size == '00' && a == '1') then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let regs : integer = if T == '0' then 1 else 2;
let alignment : integer{} = if a == '0' then 1 else ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7		6	5	4	3			0
1	1	1	1	1	0	0	1	1	D	1	0		Rn				Vd			1	1	0	0		size		T	a		Rm	
										L													N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || (size == '00' && a == '1') then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let regs : integer = if T == '0' then 1 else 2;
let alignment : integer{} = if a == '0' then 1 else ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD1 (single element to all lanes).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>[] }` Encoded in the "T" field as 0.
`{ <Dd>[], <Dd+1>[] }` Encoded in the "T" field as 1.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "a" field as 0.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size>` == **16** `<align>` is 16, meaning 16-bit alignment, encoded in the "a" field as 1.
`<size>` == **32** `<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let esize : integer{} = 8 * ebytes;
    let element : bits(esize) = MemU{8*ebytes}(address);
    let replicated_element : bits(64) = Replicate{}(element);
    for r = 0 to regs-1 do
        D(d+r) = replicated_element;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.102 VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, A3, and A4) and T32 (T1, T2, T3, and T4).

A1

31–23	22	21	20	19–16	15–12	11–8	7–6	5–4	3–0
1 1 1 1 0 1 0 0 0	D	1	0	Rn	Vd	0 1 1 1	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–23	22	21	20	19–16	15–12	11–8	7–6	5–4	3–0
1 1 1 1 0 1 0 0 0	D	1	0	Rn	Vd	1 0 1 0	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31–23	22	21	20	19–16	15–12	11–8	7–6	5–4	3–0
1 1 1 1 0 1 0 0 0	D	1	0	Rn	Vd	0 1 1 0	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A4

31								23	22	21	20	19	16	15	12	11			8	7	6	5	4	3	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn		Vd		0	0	1	0	size		align		Rm	
										L								itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15								7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn		Vd		0	1	1	1	size		align		Rm	
										L						itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T2

15								7	6	5	4	3			0	15			12	11			8	7	6	5	4	3			0
1	1	1	1	1	0	0	1	0	D	1	0	Rn				Vd				1	0	1	0	size		align		Rm			
										L												itype									

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3		0	15		12	11				8	7	6	5	4	3		0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			0	1	1	0		size		align		Rm		
										L											itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T4

15								7	6	5	4	3		0	15		12	11				8	7	6	5	4	3		0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			0	0	1	0		size		align		Rm		
										L											itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD1 (multiple single elements).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, “A3 Post-indexed”, “A4 Offset”, and “A4 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, “T3 Post-indexed”, “T4 Offset”, and “T4 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32
11	64

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd> }` Single register. Selects the A1 and T1 encodings of the instruction.
`{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
`{ <Dd>, <Dd+1>, <Dd+2> }` Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as `0b01`.
128 128-bit alignment, encoded in the "align" field as `0b10`. Available only if `<list>` contains two or four registers.
256 256-bit alignment, encoded in the "align" field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                        tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var data : bits(ebytes*8);
            if ebytes != 8 then
                data = MemU{8*ebytes}(address);
            else
                if !IsAlignedSize(address, ebytes) && AlignmentEnforced() then
                    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
                    AArch32_Abort(fault);
                end;

                if BigEndian(AccessType_ASIMD) then
                    data[31:0] = MemU{32}(address+4);
                    data[63:32] = MemU{32}(address);
                else
                    data[31:0] = MemU{32}(address);
                    data[63:32] = MemU{32}(address+4);
                end;
            end;

            D(d+r)[e*:(8*ebytes)] = data;
            address = address + ebytes;
        end;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 8*regs;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.103 VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

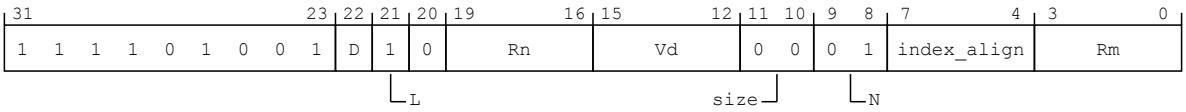

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19			16	15			12	11	10	9	8	7			4	3			0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	0	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31								23	22	21	20	19	16	15	12	11	10	9	8	7	4	3	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn		Vd		1	0	0	1	index_align		Rm	
										L						size			N				

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	0	0	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	1	0	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	0	0	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD2 (single 2-element structure to one lane).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>] }` Single-spaced registers, encoded as “spacing” = 0.
`{ <Dd>[<index>], <Dd+2>[<index>] }` Double-spaced registers, encoded as “spacing” = 1. Not permitted when `<size>` == 8.

The encoding of “spacing” depends on `<size>`:

`<size>` == **16** “spacing” is encoded in the “index_align<1>” field.
`<size>` == **32** “spacing” is encoded in the “index_align<2>” field.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` == **8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` == **16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` == **32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on `<size>`:

`<size>` == **8** Encoded in the “index_align<0>” field as 0.
`<size>` == **16** Encoded in the “index_align<0>” field as 0.
`<size>` == **32** Encoded in the “index_align<1:0>” field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<size>` == **8** `<align>` is 16, meaning 16-bit alignment, encoded in the “index_align<0>” field as 1.
`<size>` == **16** `<align>` is 32, meaning 32-bit alignment, encoded in the “index_align<0>” field as 1.
`<size>` == **32** `<align>` is 64, meaning 64-bit alignment, encoded in the “index_align<1:0>” field as `0b01`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index*:(8*ebytes)] = MemU{8*ebytes}(address);
    D(d2)[index*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.104 VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

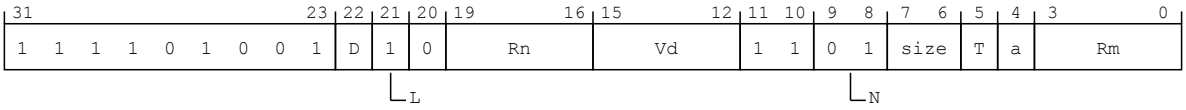

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let alignment : integer{} = if a == '0' then 1 else 2*ebytes;
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7		6	5	4	3			0
1	1	1	1	1	0	0	1	1	D	1	0		Rn				Vd			1	1	0	1	size			T	a		Rm		
										L													N									

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let alignment : integer{} = if a == '0' then 1 else 2*ebytes;
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD2 (single 2-element structure to all lanes).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of two SIMD&FP registers.

The list must be one of:

`{ <Dd>[], <Dd+1>[] }` Single-spaced registers, encoded in the "T" field as 0.
`{ <Dd>[], <Dd+2>[] }` Double-spaced registers, encoded in the "T" field as 1.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "a" field as 0.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size>` == **8** `<align>` is 16, meaning 16-bit alignment, encoded in the "a" field as 1.
`<size>` == **16** `<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as 1.
`<size>` == **32** `<align>` is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let esize : integer{} = 8 * ebytes;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.105 VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see Element and structure load/store instructions. Every element of each register is loaded. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31								23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn		Vd		1	0	0	x	size		align	Rm
										L						itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–23	22	21	20	19–16	15–12	11–8	7–6	5–4	3–0
1 1 1 1 0 1 0 0 0	D	1	0	Rn	Vd	0 0 1 1	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 6	5 … 4	3 … 0
1 1 1 1 1 0 0 1 0	D	1	0	Rn	Vd	1 0 0 x	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 6	5 … 4	3 … 0
1 1 1 1 1 0 0 1 0	D	1	0	Rn	Vd	0 0 1 1	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD2 (multiple 2-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, and “A2 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, and “T2 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as `0b1000`.
`{ <Dd>, <Dd+2> }` Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as `0b1001`.
`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register `<Dd>` is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the “align” field as `0b00`.

Whenever `<align>` is present, the permitted values are:

64 64-bit alignment, encoded in the “align” field as `0b01`.
128 128-bit alignment, encoded in the “align” field as `0b10`.
256 256-bit alignment, encoded in the “align” field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to pairs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:(8*ebytes)] = MemU{8*ebytes}(address);
            D(d2+r)[e*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
            address = address + 2*ebytes;
        end;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 16*pairs;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.106 VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

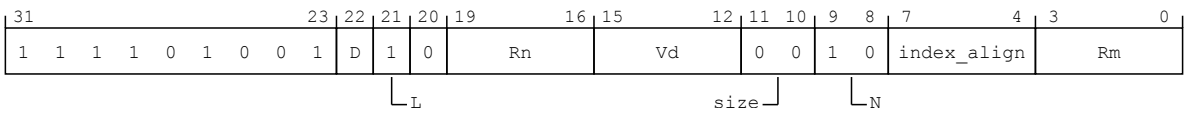

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19			16	15			12	11	10	9	8	7			4	3			0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	1	0	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	1	0	Rn	Vd	1 0	1 0	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	0	1	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	1	1	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	0	1	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD3 (single 3-element structure to one lane).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }` Single-spaced registers, encoded as “spacing” = 0.
`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }` Double-spaced registers, encoded as “spacing” = 1. Not permitted when `<size>` == 8.

The encoding of “spacing” depends on `<size>`:

`<size>` **== 8** “spacing” is encoded in the “index_align<0>” field.
`<size>` **== 16** “spacing” is encoded in the “index_align<1>” field, and “index_align<0>” is set to 0.
`<size>` **== 32** “spacing” is encoded in the “index_align<2>” field, and “index_align<1:0>” is set to `0b00`.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` **== 8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` **== 16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` **== 32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Alignment

Standard alignment rules apply, see Alignment support.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    D(d)[index*:(8*ebytes)] = MemU{8*ebytes}(address);
    D(d2)[index*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
    D(d3)[index*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.107 VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

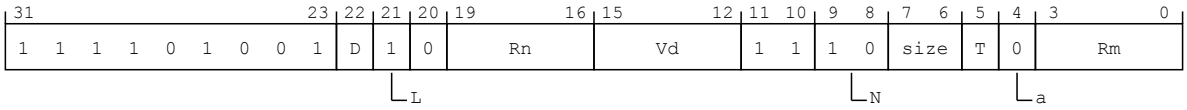

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || a == '1' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7		6	5	4	3			0
1	1	1	1	1	0	0	1	1	D	1	0		Rn				Vd			1	1	1	0	size			T	0		Rm		
										L													N					a				

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || a == '1' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD3 (single 3-element structure to all lanes).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of three SIMD&FP registers.

The list must be one of:

`{ <Dd>[], <Dd+1>[], <Dd+2>[] }` Single-spaced registers, encoded in the “T” field as 0.
`{ <Dd>[], <Dd+2>[], <Dd+4>[] }` Double-spaced registers, encoded in the “T” field as 1.

The register `<Dd>` is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Alignment

Standard alignment rules apply, see Alignment support.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    let esize : integer{} = ebytes * 8;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    let element3 : bits(esize) = MemU{8*ebytes}(address+2*ebytes);

    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);
    D(d3) = Replicate{64}(element3);
    if wback then
        if register_index then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.108 VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see Element and structure load/store instructions. Every element of each register is loaded. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

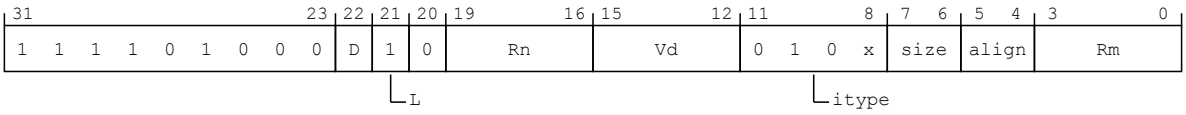

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '010x' then See("Related encodings"); end;
if size == '11' || align[1] == '1' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

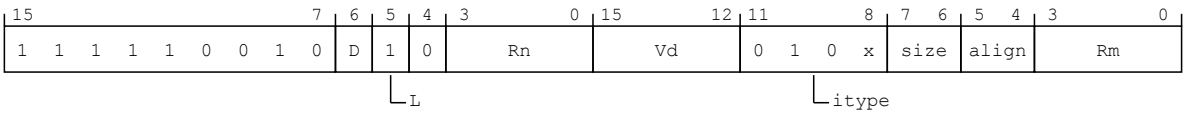

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '010x' then See("Related encodings"); end;
if size == '11' || align[1] == '1' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD3 (multiple 3-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2> }` Single-spaced registers, encoded in the "itype" field as `0b0100`.
`{ <Dd>, <Dd+2>, <Dd+4> }` Double-spaced registers, encoded in the "itype" field as `0b0101`.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as `0b01`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about `<Rn>`, `!`, and `<Rm>`, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        D(d)[e*:(8*ebytes)] = MemU{8*ebytes}(address);
        D(d2)[e*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
        D(d3)[e*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
        address = address + 3*ebytes;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 24;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.109 VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

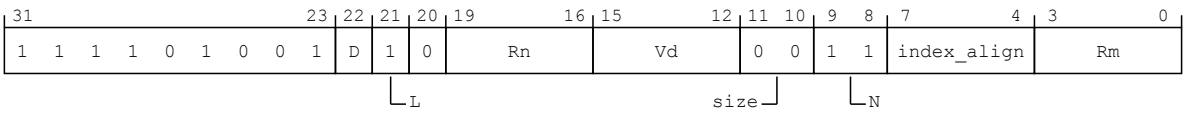

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer  = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	1	0	Rn	Vd	0 1	1 1	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31								23	22	21	20	19	16	15	12	11	10	9	8	7	4	3	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn		Vd		1	0	1	1	index_align		Rm	
										L						size			N				

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	0	1	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer  = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	1	1	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	0	1	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD4 (single 4-element structure to one lane).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }` Single-spaced registers, encoded as “spacing” = 0.

`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }` Double-spaced registers, encoded as “spacing” = 1. Not permitted when `<size>` == 8.

The encoding of “spacing” depends on `<size>`:

`<size>` == **16** “spacing” is encoded in the “index_align<1>” field.
`<size>` == **32** “spacing” is encoded in the “index_align<2>” field.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` == **8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` == **16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` == **32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on `<size>`:

`<size>` == **8** Encoded in the “index_align<0>” field as 0.
`<size>` == **16** Encoded in the “index_align<0>” field as 0.
`<size>` == **32** Encoded in the “index_align<1:0>” field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<size>` == **8** `<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
`<size>` == **16** `<align>` is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.
`<size>` == **32** `<align>` can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as `0b01`, and 128-bit alignment is encoded in the "index_align<1:0>" field as `0b10`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index*:(8*ebytes)] = MemU{8*ebytes}(address);
    D(d2)[index*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
    D(d3)[index*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
    D(d4)[index*:(8*ebytes)] = MemU{8*ebytes}(address+3*ebytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.110 VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

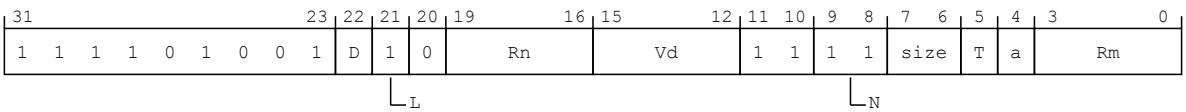

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' && a == '0' then Undefined(); end;
let ebytes : integer{} = if size == '11' then 4 else 1 << UInt(size);
let alignment : integer{} = (if a == '0' && size != '11' then 1
                             else 4 << (UInt(size[1]) + UInt(size[0])));
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7		6	5	4	3			0
1	1	1	1	1	0	0	1	1	D	1	0		Rn				Vd			1	1	1	1	size			T	a		Rm	
										L													N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' && a == '0' then Undefined(); end;
let ebytes : integer{} = if size == '11' then 4 else 1 << UInt(size);
let alignment : integer{} = (if a == '0' && size != '11' then 1
                             else 4 << (UInt(size[1]) + UInt(size[0])));
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD4 (single 4-element structure to all lanes).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
1x	32

<list>

Is a list containing the 64-bit names of four SIMD&FP registers.

The list must be one of:

`{ <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] }` Single-spaced registers, encoded in the “T” field as 0.
`{ <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] }` Double-spaced registers, encoded in the “T” field as 1.

The register `<Dd>` is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the “a” field as 0.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size>` == **8** `<align>` is 32, meaning 32-bit alignment, encoded in the “a” field as 1.
`<size>` == **16** `<align>` is 64, meaning 64-bit alignment, encoded in the “a” field as 1.
`<size>` == **32** `<align>` can be 64 or 128. 64-bit alignment is encoded in the “a:size<0>” field as `0b10`, and 128-bit alignment is encoded in the “a:size<0>” field as `0b11`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let esize : integer{} = ebytes * 8;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    let element3 : bits(esize) = MemU{8*ebytes}(address+2*ebytes);
    let element4 : bits(esize) = MemU{8*ebytes}(address+3*ebytes);
    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);
    D(d3) = Replicate{64}(element3);
    D(d4) = Replicate{64}(element4);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.111 VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see Element and structure load/store instructions. Every element of each register is loaded. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

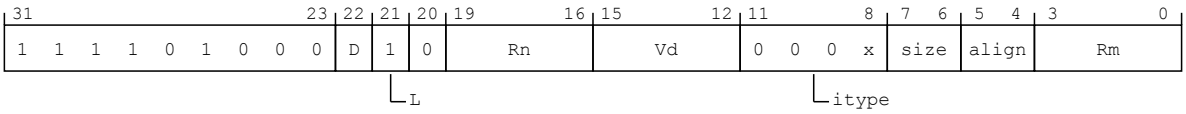

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '000x' then See("Related encodings"); end;
if size == '11' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15–7	6	5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 1 1 0 0 1 0	D	1	0	Rn	Vd	0 0 0 x	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VLD4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '000x' then See("Related encodings"); end;
if size == '11' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD4 (multiple 4-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Single-spaced registers, encoded in the "itype" field as `0b0000`.
`{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }` Double-spaced registers, encoded in the "itype" field as `0b0001`.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64 64-bit alignment, encoded in the “align” field as `0b01`.
128 128-bit alignment, encoded in the “align” field as `0b10`.
256 256-bit alignment, encoded in the “align” field as `0b11`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        D(d)[e*:(8*ebytes)] = MemU{8*ebytes}(address);
        D(d2)[e*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
        D(d3)[e*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
        D(d4)[e*:(8*ebytes)] = MemU{8*ebytes}(address+3*ebytes);
        address = address + 4*ebytes;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 32;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.112 VLDM, VLDMDB, VLDMIA

Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

This instruction is used by the alias VPOP.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

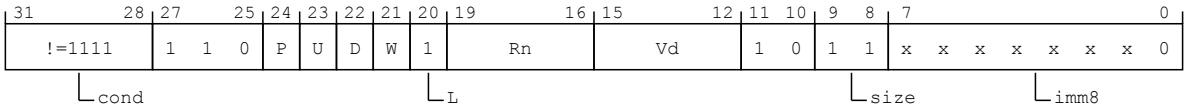

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VLDMDB{<c>}{<q>}{.<size>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VLDM{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

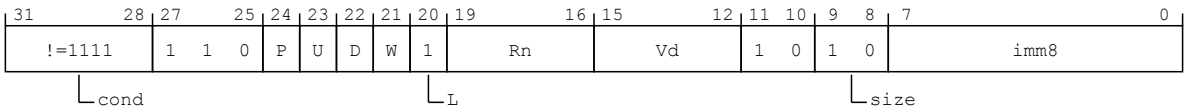

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VLDMDB{<c>}{<q>}{.<size>}  <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VLDM{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	P	U	D	W	1	Rn	Vd	1 0	1 1	x x x x x x x 0
					L				size	imm8

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VLDMDB{<c>}{<q>}{.<size>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VLDM{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	P	U	D	W	1	Rn	Vd	1 0	1 0	imm8
					L				size	

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VLDMDB{<c>}{<q>}{.<size>}  <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VLDM{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLDM.

Related encodings: See Advanced SIMD and floating-point 64-bit move for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If writeback is not specified, the PC can be used.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in “W”:

W	!
0	[absent]
1	[present]

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Alias Conditions

Alias	Is preferred when
VPOP	P == '0' && U == '1' && W == '1' && Rn == '1101'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;

    for r = 0 to regs-1 do
        if single_regs then
            S(d+r) = MemA{32}(address);
            address = address+4;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            address = address+8;

            // Combine the word-aligned words in the correct order for current endianness.
            D(d+r) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.113 VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

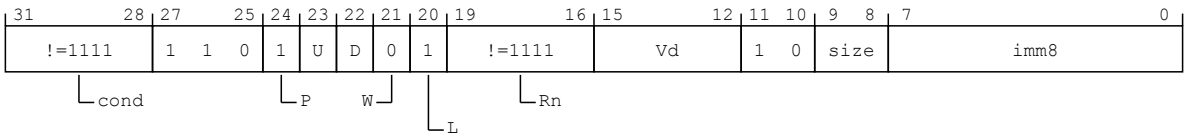

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VLDR{<c>}{<q>}.16  <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VLDR{<c>}{<q>}{.32}  <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VLDR{<c>}{<q>}{.64}  <Dd>, [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	1	U	D	0	1	!=1111	Vd	1 0	size	imm8
	P			W	L	Rn				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VLDR{<c>}{<q>}.16  <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VLDR{<c>}{<q>}{.32}  <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VLDR{<c>}{<q>}{.64}  <Dd>, [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Half-precision scalar” and “T1 Half-precision scalar” variants: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the “imm8” field as <imm>/2.

For the “A1 Double-precision scalar”, “A1 Single-precision scalar”, “T1 Double-precision scalar”, and “T1 Single-precision scalar” variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16 =>
            H(d) = MemA{16}(address);
        when 32 =>
            S(d) = MemA{32}(address);
        when 64 =>
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            // Combine the word-aligned words in the correct order for current endianness.
            D(d) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.114 VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–25	24	23	22	21	20	19–16	15–12	11–10	9–8	7–0
!=1111	1 1 0	1	U	D	0	1	1 1 1 1	Vd	1 0	size	imm8
cond		P		W		L	Rn				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VLDR{<c>}{<q>}.16  <Sd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}.16  <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VLDR{<c>}{<q>}{.32}  <Sd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.32}  <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VLDR{<c>}{<q>}{.64}  <Dd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.64}  <Dd>, [PC, #{+/-}<imm>]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	1	U	D	0	1	1 1 1 1	Vd	1 0	size	imm8
	P		W		L	Rn				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VLDR{<c>}{<q>}.16  <Sd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}.16  <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VLDR{<c>}{<q>}{.32}  <Sd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.32}  <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VLDR{<c>}{<q>}{.64}  <Dd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.64}  <Dd>, [PC, #{+/-}<imm>]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<label>

The label of the literal data item to be loaded.

For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020.

For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510.

If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>

For the "A1 Half-precision scalar" and "T1 Half-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

For the "A1 Double-precision scalar", "A1 Single-precision scalar", "T1 Double-precision scalar", and "T1 Single-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16 =>
            H(d) = MemA{16}(address);
        when 32 =>
            S(d) = MemA{32}(address);
        when 64 =>
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            // Combine the word-aligned words in the correct order for current endianness.
            D(d) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.115 VMAX (floating-point)

Vector Maximum (floating-point)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

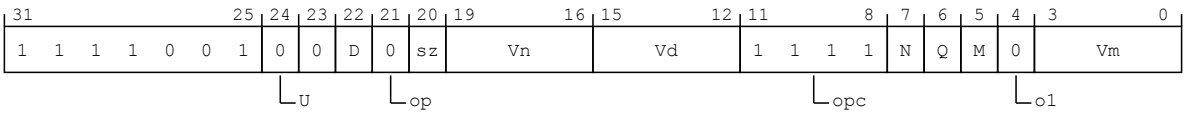

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMAX{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

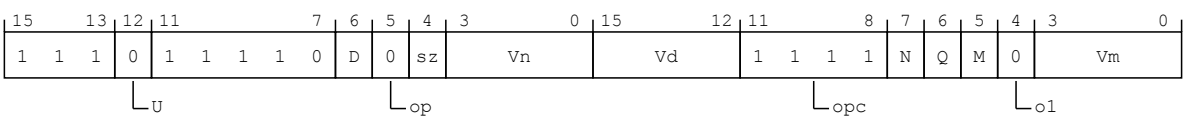

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMAX{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Floating-point maximum and minimum

- max(+0.0, -0.0) = +0.0
- If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = D(n+r)[e*:esize];
            let op2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMax{esize}(op1, op2, fpcr);
            else
                D(d+r)[e*:esize] = FPMin{esize}(op1, op2, fpcr);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.116 VMAX (integer)

Vector Maximum (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 1 0 (opc)	N	Q	M	0 (op)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMAX{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 1 0 (opc)	N	Q	M	0 (op)	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMAX{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if maximum then Max(element1, element2)
                                    else Min(element1, element2));
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.117 VMAXNM

Floating-point Maximum Number

This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE 754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMAX`.

This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

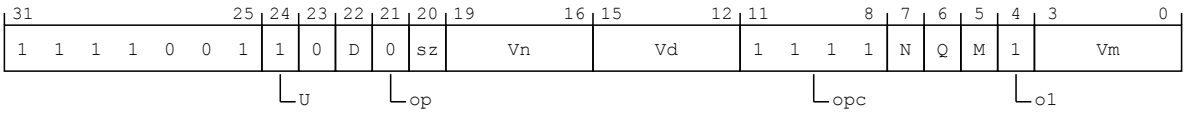

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMAXNM{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMAXNM{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

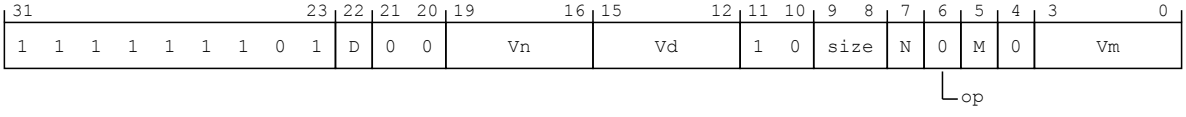

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMAXNM{<q>}.F16  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMAXNM{<q>}.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMAXNM{<q>}.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

T1

15 … 13	12	11 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1 1 1	1	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 1 1	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMAXNM{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMAXNM{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15								7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn		Vd		1	0	size		N	0	M	0	Vm	
																					op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMAXNM{<q>}.F16  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMAXNM{<q>}.F32  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMAXNM{<q>}.F64  <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(n+r)[e*:esize];
            let operand2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMaxNum{esize}(operand1, operand2, fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            else
                D(d+r)[e*:esize] = FPMinNum{esize}(operand1, operand2, fpcr);
            end;
        end;
    end;
else                        // VFP instruction
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            if maximum then
                H(d) = FPMaxNum{16}(H(n), H(m), fpcr);
            else
                H(d) = FPMinNum{16}(H(n), H(m), fpcr);
            end;
        when 32 =>
            if maximum then
                S(d) = FPMaxNum{32}(S(n), S(m), fpcr);
            else
                S(d) = FPMinNum{32}(S(n), S(m), fpcr);
            end;
        when 64 =>
            if maximum then
                D(d) = FPMaxNum{64}(D(n), D(m), fpcr);
            else
                D(d) = FPMinNum{64}(D(n), D(m), fpcr);
            end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.118 VMIN (floating-point)

Vector Minimum (floating-point)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

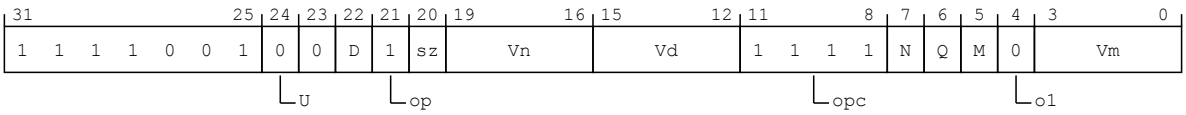

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMIN{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

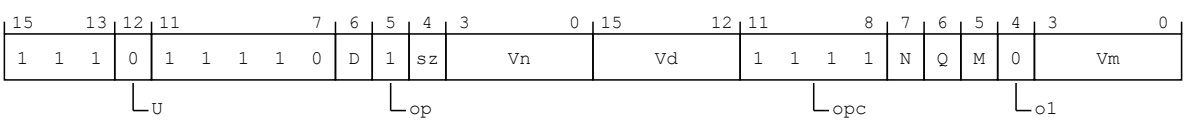

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMIN{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Floating-point minimum

- min(+0.0, -0.0) = -0.0
- If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = D(n+r)[e*:esize];
            let op2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMax{esize}(op1, op2, fpcr);
            else
                D(d+r)[e*:esize] = FPMin{esize}(op1, op2, fpcr);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.119 VMIN (integer)

Vector Minimum (integer)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 1 0 (opc)	N	Q	M	1 (op)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMIN{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 1 0 (opc)	N	Q	M	1 (op)	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMIN{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if maximum then Max(element1, element2)
                                    else Min(element1, element2));
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.120 VMINNM

Floating-point Minimum Number

This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE 754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMIN`.

This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

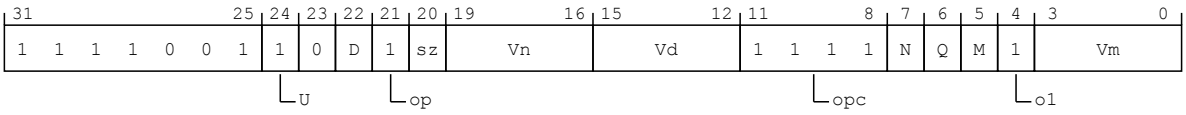

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMINNM{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMINNM{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

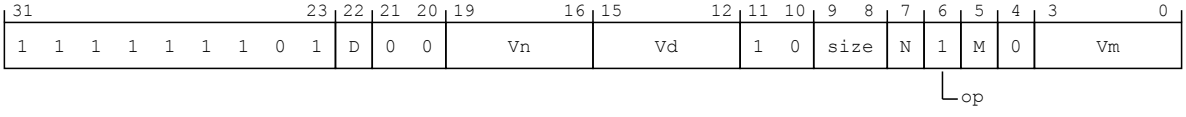

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMINNM{<q>}.F16  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMINNM{<q>}.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMINNM{<q>}.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

T1

15 … 13	12	11 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1 1 1	1	1 1 1 1 0	D	1	sz	Vn	Vd	1 1 1 1	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMINNM{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMINNM{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15								7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn		Vd		1	0	size		N	1	M	0	Vm	
																					op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMINNM{<q>}.F16  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMINNM{<q>}.F32  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMINNM{<q>}.F64  <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(n+r)[e*:esize];
            let operand2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMaxNum{esize}(operand1, operand2, fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            else
                D(d+r)[e*:esize] = FPMinNum{esize}(operand1, operand2, fpcr);
            end;
        end;
    end;
else                        // VFP instruction
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            if maximum then
                H(d) = FPMaxNum{16}(H(n), H(m), fpcr);
            else
                H(d) = FPMinNum{16}(H(n), H(m), fpcr);
            end;
        when 32 =>
            if maximum then
                S(d) = FPMaxNum{32}(S(n), S(m), fpcr);
            else
                S(d) = FPMinNum{32}(S(n), S(m), fpcr);
            end;
        when 64 =>
            if maximum then
                D(d) = FPMaxNum{64}(D(n), D(m), fpcr);
            else
                D(d) = FPMinNum{64}(D(n), D(m), fpcr);
            end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.121 VMLA (floating-point)

Vector Multiply Accumulate (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

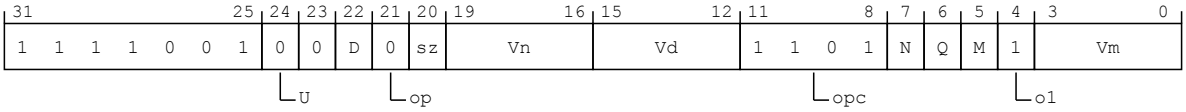

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

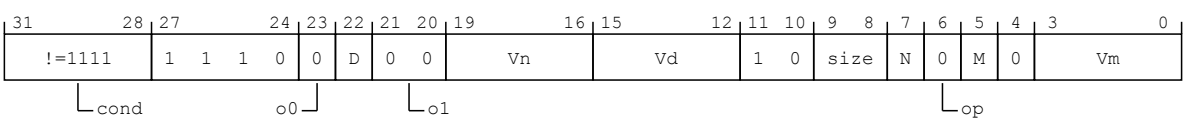

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMLA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMLA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMLA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 0 1	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	0	D	0	0	Vn		Vd		1	0	size		N	0	M	0	Vm	
								o0			o1										op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMLA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMLA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMLA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var addend : bits(esize) = FPMul{}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
                if !add then addend = FPNeg{esize}(addend, fpcr); end;
                D(d+r)[e*:esize] = FPAdd{esize}(D(d+r)[e*:esize], addend, fpcr);
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                var addend16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
                if !add then addend16 = FPNeg{16}(addend16, fpcr); end;
                H(d) = FPAdd{16}(H(d), addend16, fpcr);
            when 32 =>
                var addend32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
                if !add then addend32 = FPNeg{32}(addend32, fpcr); end;
                S(d) = FPAdd{32}(S(d), addend32, fpcr);
            when 64 =>
                var addend64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
                if !add then addend64 = FPNeg{64}(addend64, fpcr); end;
                D(d) = FPAdd{64}(D(d), addend64, fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.122 VMLA (integer)

Vector Multiply Accumulate (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

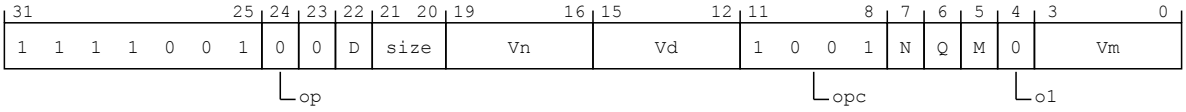

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

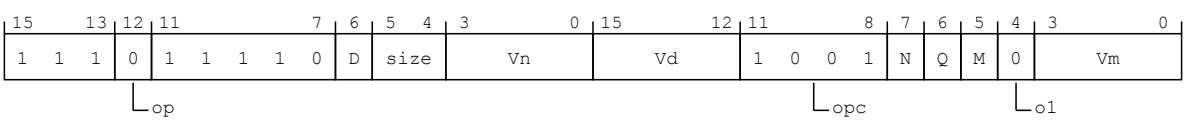

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.123 VMLA (by scalar)

Vector Multiply Accumulate (by scalar)

Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 0 0 1	Q	1	D	!=11 (size)	Vn	Vd	0	0 (op)	0	F	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	0	0 (op)	0	F	N	1	M	0	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLA{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLA{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    Undefined();
end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `I16` or `F16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `I32` or `F32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                  else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                 fpcr),
                                                                    fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.124 VMLAL (integer)

Vector Multiply Accumulate Long (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	1 0	0	0	N	0	M	0	Vm
						size				op						

Encoding

```
VMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	1 0	0	0	N	0	M	0	Vm
						size				op						

Encoding

```
VMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.125 VMLAL (by scalar)

Vector Multiply Accumulate Long (by scalar)

Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0	0	1 0	N	1	M	0	Vm
						size				op						

Encoding

```
VMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0	0	1 0	N	1	M	0	Vm
						size				op						

Encoding

```
VMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm[x]>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16` or `U16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `S32` or `U32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                  else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                  fpcr),
                                                                          fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.126 VMLS (floating-point)

Vector Multiply Subtract (floating-point)

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

Note

Arm recommends that software does not use the `VMLS` instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	0	0	D	1	sz	Vn	Vd	1 1 0 1	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31–28	27–24	23	22	21	20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
!=1111	1 1 1 0	0	D	0	0	Vn	Vd	1	0	size	N	1	M	0	Vm
cond		o0			o1							op			

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMLS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMLS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMLS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	1	sz	Vn	Vd	1 1 0 1	N	Q	M	1	Vm
	U			op				opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	0	D	0	0	Vn		Vd		1	0	size		N	1	M	0	Vm	
								o0			o1										op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMLS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMLS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMLS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "A2 Double-precision scalar", "A2 Half-precision scalar", "A2 Single-precision scalar", "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var addend : bits(esize) = FPMul{}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
                if !add then addend = FPNeg{esize}(addend, fpcr); end;
                D(d+r)[e*:esize] = FPAdd{esize}(D(d+r)[e*:esize], addend, fpcr);
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                var addend16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
                if !add then addend16 = FPNeg{16}(addend16, fpcr); end;
                H(d) = FPAdd{16}(H(d), addend16, fpcr);
            when 32 =>
                var addend32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
                if !add then addend32 = FPNeg{32}(addend32, fpcr); end;
                S(d) = FPAdd{32}(S(d), addend32, fpcr);
            when 64 =>
                var addend64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
                if !add then addend64 = FPNeg{64}(addend64, fpcr); end;
                D(d) = FPAdd{64}(D(d), addend64, fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.127 VMLS (integer)

Vector Multiply Subtract (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

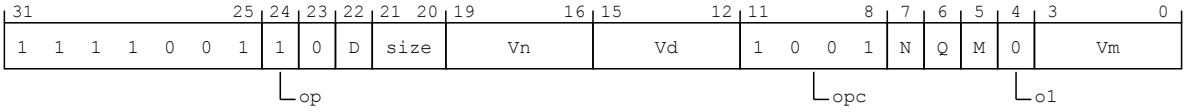

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

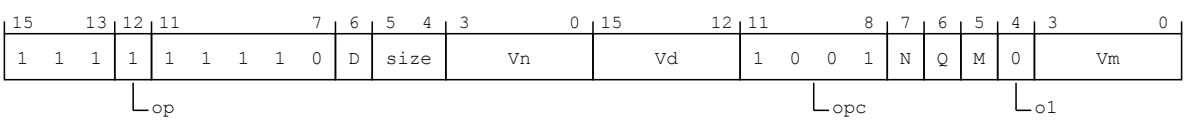

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.128 VMLS (by scalar)

Vector Multiply Subtract (by scalar)

Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 0 0 1	Q	1	D	!=11 (size)	Vn	Vd	0	1 (op)	0	F	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	0	1 (op)	0	F	N	1	M	0	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMLS{<c>}{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMLS{<c>}{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    Undefined();
end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `I16` or `F16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `I32` or `F32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                  else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                 fpcr),
                                                                    fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.129 VMLSL (integer)

Vector Multiply Subtract Long (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	1 0	1	0	N	0	M	0	Vm
						size				op						

Encoding

```
VMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–10	9	8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	1 0	1	0	N	0	M	0	Vm
						size				op						

Encoding

```
VMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.130 VMLSL (by scalar)

Vector Multiply Subtract Long (by scalar)

Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0	1	1 0	N	1	M	0	Vm
						size				op						

Encoding

```
VMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0	1	1 0	N	1	M	0	Vm
						size				op						

Encoding

```
VMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm[x]>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16` or `U16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `S32` or `U32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                  else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                 fpcr),
                                                                    fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.131 VMMLA

BFloat16 floating-point matrix multiply-accumulate. This instruction multiplies the 2x4 matrix of BF16 values in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction does not update the FPSCR exception status.

Note

Arm expects that the VMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two VDOT instructions, with a goal that it should have significantly higher throughput.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32BF16)

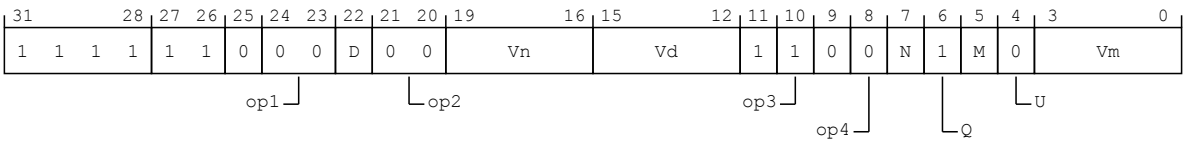

Encoding

```
VMMLA{<q>}.BF16  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 2;
```

T1

(FEAT_AA32BF16)

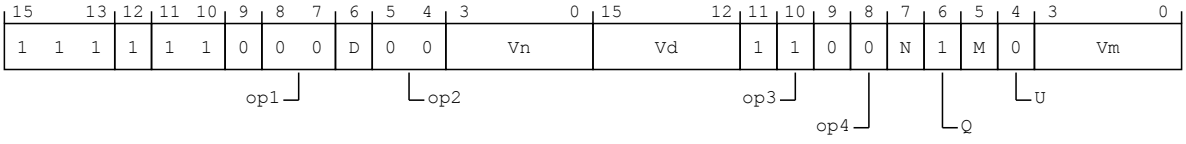

Encoding

```
VMMLA{<q>}.BF16  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 2;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();

let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let acc : bits(128) = Q(d>>1);
let fpcr : FPCR_Type = EffectiveFPCR();

Q(d>>1) = BFMatMulAddH(acc, operand1, operand2, fpcr);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.132 VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

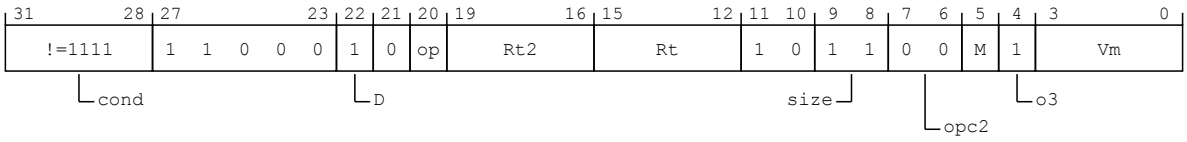

Encoding for the From general-purpose registers variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}  <Dm>, <Rt>, <Rt2>
```

Encoding for the To general-purpose registers variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}  <Rt>, <Rt2>, <Dm>
```

Decode for all variants of this encoding

```
let to_arm_registers : boolean = (op == '1');
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(M::Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	0	0	0	1	0	op	Rt2		Rt		1	0	1	1	0	0	M	1	Vm	
									D									size		opc2			o3	

Encoding for the From general-purpose registers variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}  <Dm>, <Rt>, <Rt2>
```

Encoding for the To general-purpose registers variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}  <Rt>, <Rt2>, <Dm>
```

Decode for all variants of this encoding

```
let to_arm_registers : boolean = (op == '1');
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(M::Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VMOV (between two general-purpose registers and a doubleword floating-point register).

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dm>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “M:Vm” field.

<Rt>

Is the first general-purpose register that `<Dm>`[31:0] will be transferred to or from, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that `<Dm>`[63:32] will be transferred to or from, encoded in the “Rt2” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R(t) = D(m)[31:0];
        R(t2) = D(m)[63:32];
    else
        D(m)[31:0] = R(t);
        D(m)[63:32] = R(t2);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.133 VMOV (between general-purpose register and half-precision)

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in the bottom 16 bits of a 32-bit SIMD&FP register to the bottom 16 bits of a general-purpose register, or the value held in the bottom 16 bits of a general-purpose register to the bottom 16 bits of a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FP16)

31–28	27–21	20	19–16	15–12	11–8	7	6–0
!=1111 (cond)	1 1 1 0 0 0 0	op	Vn	Rt	1 0 0 1	N	(0)(0)1(0)(0)(0)(0)

Encoding for the From general-purpose register variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}.F16  <Sn>, <Rt>
```

Encoding for the To general-purpose register variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}.F16  <Rt>, <Sn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end;
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

(FEAT_FP16)

15										5	4	3	0	15	12	11	8	7	6	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn		Rt		1 0 0 1		N	(0)(0)1(0)(0)(0)(0)	

Encoding for the From general-purpose register variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}.F16  <Sn>, <Rt>
```

Encoding for the To general-purpose register variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}.F16  <Rt>, <Sn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sn>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Vn:N” field.

<Rt>

Is the general-purpose register that `<Sn>` will be transferred to or from, encoded in the “Rt” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    if to_arm_register then
        R(t) = Zeros{16} :: S(n)[15:0];
    else
        S(n) = Zeros{16} :: R(t)[15:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.134 VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, A3, A4, and A5) and T32 (T1, T2, T3, T4, and T5).

A1

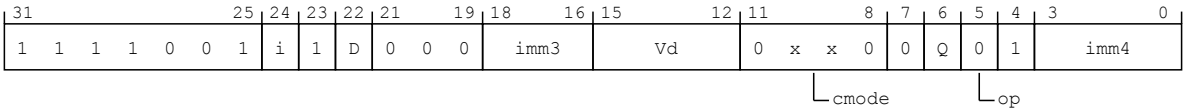

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.I32  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.I32  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

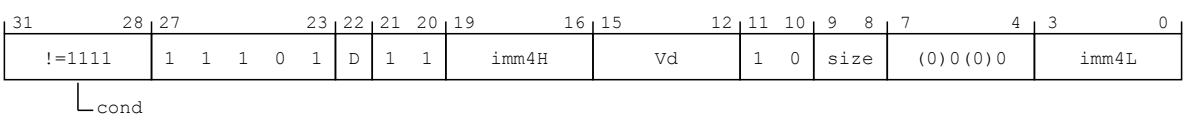

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMOV{<c>}{<q>}.F16  <Sd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMOV{<c>}{<q>}.F32  <Sd>, #<imm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMOV{<c>}{<q>}.F64  <Dd>, #<imm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let imm : bits(esize) = VFPExpandImm{}(imm4H::imm4L);
let advsimd : boolean = FALSE;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let regs : integer = if size == '11' then 1 else 0;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A3

31						25	24	23	22	21		19	18	16	15	12	11			8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd		1	0	x	0	0	Q	0	1	imm4	
																			cmode				op			

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.I16  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.I16  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A4

31						25	24	23	22	21		19	18		16	15		12	11			8	7	6	5	4	3		0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			1	1	x	x	0	Q	0	1	imm4		
																					cmode				op				

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.<dt>  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.<dt>  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A5

31						25	24	23	22	21		19	18		16	15		12	11			8	7	6	5	4	3		0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			1	1	1	0	0	Q	1	1	imm4		
																					cmode				op				

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.I64  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.I64  <Qd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	0 x x 0 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.I32  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.I32  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15–7	6	5 4	3–0	15–12	11 10	9 8	7–4	3–0
1 1 1 0 1 1 1 0 1	D	1 1	imm4H	Vd	1 0	size	(0)0(0)0	imm4L

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMOV{<c>}{<q>}.F16  <Sd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMOV{<c>}{<q>}.F32  <Sd>, #<imm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMOV{<c>}{<q>}.F64  <Dd>, #<imm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let imm : bits(esize) = VFPExpandImm{}(imm4H::imm4L);
let advsimd : boolean = FALSE;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let regs : integer = if size == '11' then 1 else 0;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T3

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 0 x 0 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.I16  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.I16  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T4

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 1 x x (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.<dt>  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.<dt>  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T5

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 1 1 0 (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}.I64  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}.I64  <Qd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A3 128-bit SIMD vector”, “A3 64-bit SIMD vector”, “A4 128-bit SIMD vector”, “A4 64-bit SIMD vector”, “A5 128-bit SIMD vector”, and “A5 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, “T2 Single-precision scalar”, “T3 128-bit SIMD vector”, “T3 64-bit SIMD vector”, “T4 128-bit SIMD vector”, “T4 64-bit SIMD vector”, “T5 128-bit SIMD vector”, and “T5 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A3 128-bit SIMD vector”, “A3 64-bit SIMD vector”, “A4 128-bit SIMD vector”, “A4 64-bit SIMD vector”, “A5 128-bit SIMD vector”, “A5 64-bit SIMD vector”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T3 128-bit SIMD vector”, “T3 64-bit SIMD vector”, “T4 128-bit SIMD vector”, “T4 64-bit SIMD vector”, “T5 128-bit SIMD vector”, and “T5 64-bit SIMD vector” variants: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in “imm4H:imm4L”. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 floating-point instructions.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<dt>

The data type, encoded in “cmode”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cmode	<dt>
110x	I32
1110	I8
1111	F32

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if esize <= 32 then
        S(d) = ZeroExtend{32}(imm);
    else
        for r = 0 to regs-1 do
            D(d+r) = ZeroExtend{64}(imm);
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.135 VMOV (register)

Copy between FP registers copies the contents of one FP register to another.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A2) and T32 (T2).

A2

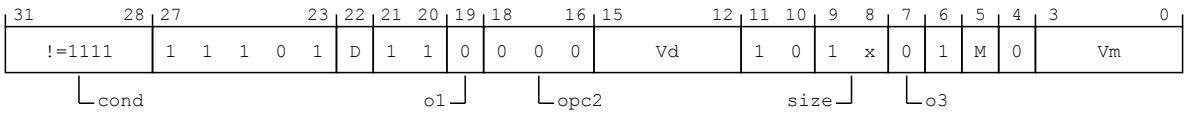

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMOV{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMOV{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let single_register : boolean = (size == '10');
let advsimd : boolean = FALSE;
```

T2

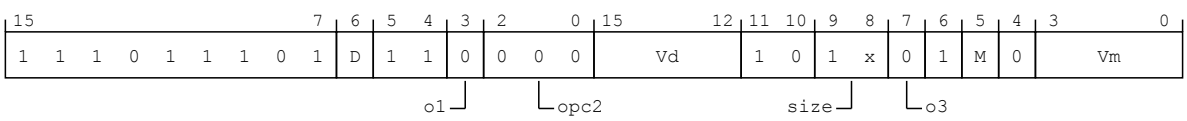

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMOV{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMOV{<c>}{<q>}.F64  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let single_register : boolean = (size == '10');
let advsimd : boolean = FALSE;
```

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S(d) = S(m);
    else
        D(d) = D(m);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.136 VMOV (general-purpose register to scalar)

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–24	23	22–21	20	19–16	15–12	11	10–8	7	6–5	4	3–0
!=1111 (cond)	1 1	1 0	0	opc1	0 (L)	Vd	Rt	1	0 1 1	D	opc2	1	(0)(0)(0)(0)

Encoding

```
VMOV{<c>}{<q>}{.<size>}  <Dd[x]>, <Rt>
```

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if opc == '0x10' then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12	11–10	9–8	7	6–5	4	3–0	15–12	11	10–8	7	6–5	4	3–0
1 1 1	0	1 1	1 0	0	opc1	0 (L)	Vd	Rt	1	0 1 1	D	opc2	1	(0)(0)(0)(0)

Encoding

```
VMOV{<c>}{<q>}{.<size>}  <Dd[x]>, <Rt>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if opc == '0x10' then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

The data size. It must be one of:

8 Encoded as `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
16 Encoded as `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
32 Encoded as `opc1<1>` = 0, `opc2` = `0b00`. `[x]` is encoded in `opc1<0>`.
omitted Equivalent to `32`.

<Dd[x]>

The scalar. The register `<Dd>` is encoded in D:Vd. For details of how `[x]` is encoded, see the description of `<size>`.

<Rt>

The source general-purpose register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    D(d)[index*:esize] = R(t)[esize-1:0];
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.137 VMOV (between general-purpose register and single-precision)

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–21	20	19–16	15–12	11–8	7	6–0
!=1111 (cond)	1 1 1 0 0 0 0	op	Vn	Rt	1 0 1 0	N	(0)(0)1(0)(0)(0)(0)

Encoding for the From general-purpose register variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}  <Sn>, <Rt>
```

Encoding for the To general-purpose register variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}  <Rt>, <Sn>
```

Decode for all variants of this encoding

```
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15–5	4	3–0	15–12	11–8	7	6–0
1 1 1 0 1 1 1 0 0 0 0	op	Vn	Rt	1 0 1 0	N	(0)(0)1(0)(0)(0)(0)

Encoding for the From general-purpose register variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}  <Sn>, <Rt>
```

Encoding for the To general-purpose register variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}  <Rt>, <Sn>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sn>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Vn:N” field.

<Rt>

Is the general-purpose register that `<Sn>` will be transferred to or from, encoded in the “Rt” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    if to_arm_register then
        R(t) = S(n);
    else
        S(n) = R(t);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.138 VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22–21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
!=1111 (cond)	1	1	1	0	U	opc1	1 (L)	Vn	Rt	1	0	1	1	N	opc2	1	(0)(0)(0)(0)

Encoding

```
VMOV{<c>}{<q>}{.<dt>}  <Rt>, <Dn[x]>
```

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if (U::opc) IN {'10x00', 'x0x10'} then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let unsigned : boolean = (U == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(N::Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15–13	12	11	10	9	8	7	6–5	4	3–0	15–12	11	10	9	8	7	6–5	4	3–0
1 1 1	0	1	1	1	0	U	opc1	1 (L)	Vn	Rt	1	0	1	1	N	opc2	1	(0)(0)(0)(0)

Encoding

```
VMOV{<c>}{<q>}{.<dt>}  <Rt>, <Dn[x]>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if (U::opc) IN {'10x00', 'x0x10'} then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let unsigned : boolean = (U == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(N::Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

The data type. It must be one of:

S8 Encoded as U = 0, `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
S16 Encoded as U = 0, `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
U8 Encoded as U = 1, `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
U16 Encoded as U = 1, `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
32 Encoded as U = 0, `opc1<1>` = 0, `opc2` = `0b00`. `[x]` is encoded in `opc1<0>`.
omitted Equivalent to `32`.

<Rt>

The destination general-purpose register.

<Dn[x]>

The scalar. For details of how `[x]` is encoded see the description of `<dt>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R(t) = ZeroExtend{32}(D(n)[index*:esize]);
    else
        R(t) = SignExtend{32}(D(n)[index*:esize]);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.139 VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

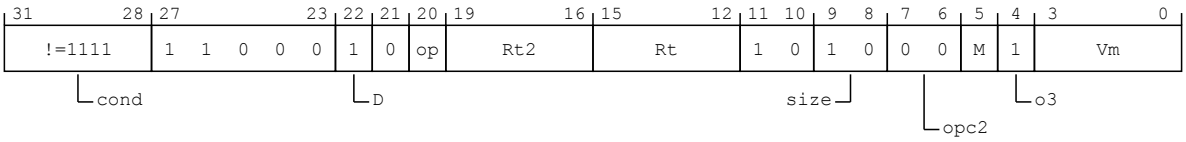

Encoding for the From general-purpose registers variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}  <Sm>, <Sm1>, <Rt>, <Rt2>
```

Encoding for the To general-purpose registers variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}  <Rt>, <Rt2>, <Sm>, <Sm1>
```

Decode for all variants of this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(Vm::M);
let m2 : integer = m + 1;
let to_arm_registers : boolean = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

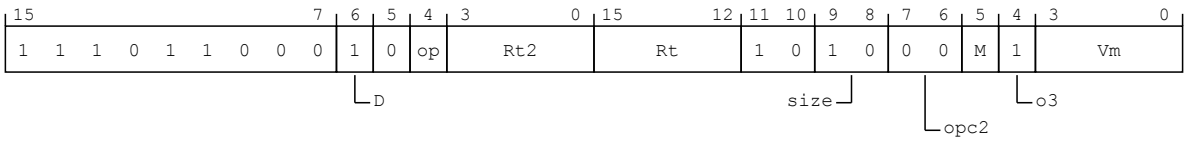

Encoding for the From general-purpose registers variant

Applies when `(op == 0)`

```
VMOV{<c>}{<q>}  <Sm>, <Sm1>, <Rt>, <Rt2>
```

Encoding for the To general-purpose registers variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}  <Rt>, <Rt2>, <Sm>, <Sm1>
```

Decode for all variants of this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(Vm::M);
let m2 : integer = m + 1;
let to_arm_registers : boolean = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VMOV (between two general-purpose registers and two single-precision registers).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sm>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Vm:M” field.

<Sm1>

Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after `<Sm>`.

<Rt>

Is the first general-purpose register that `<Sm>` will be transferred to or from, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that `<Sm1>` will be transferred to or from, encoded in the “Rt2” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R(t) = S(m);
        R(t2) = S(m2);
    else
        S(m) = R(t);
        S(m2) = R(t2);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.140 VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another.

This is an alias of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

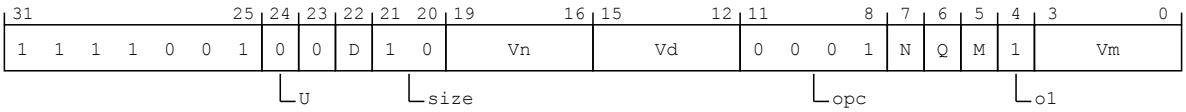

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}{.<dt>}  <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dm>, <Dm>
```

and is the preferred disassembly when `N :: Vn == M :: Vm`.

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}{.<dt>}  <Qd>, <Qm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qm>, <Qm>
```

and is the preferred disassembly when `N :: Vn == M :: Vm`.

T1

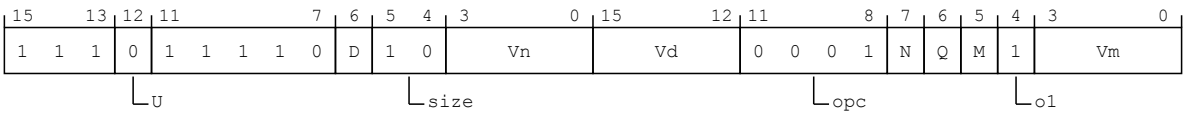

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMOV{<c>}{<q>}{.<dt>}  <Dd>, <Dm>
```

is equivalent to

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dm>, <Dm>
```

and is the preferred disassembly when `N :: Vn == M :: Vm`.

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMOV{<c>}{<q>}{.<dt>}  <Qd>, <Qm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qm>, <Qm>
```

and is the preferred disassembly when `N :: Vn == M :: Vm`.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. `<dt>` must not be `F64`, but it is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

Operation

The description of VORR (register) gives the operational pseudocode for this instruction.

Operational Information

The description of VORR (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.141 VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

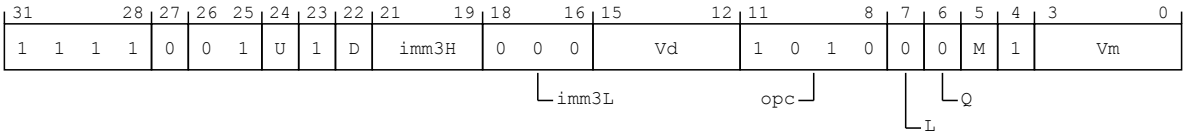

Encoding

Applies when `(imm3H IN {001, 010, 100})`

```
VMOVL{<c>}{<q>}.<dt>  <Qd>, <Dm>
```

Decode for this encoding

```
if imm3H == '000' then See("Related encodings"); end;
if ! imm3H IN {'001', '010', '100'} then See("VSHLL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm3H);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let unsigned : boolean = (U == '1');
```

T1

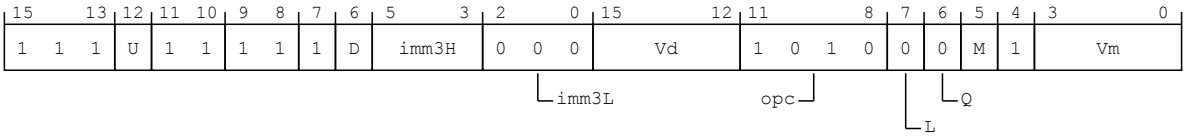

Encoding

Applies when `(imm3H IN {001, 010, 100})`

```
VMOVL{<c>}{<q>}.<dt>  <Qd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if imm3H == '000' then See("Related encodings"); end;
if ! imm3H IN {'001', '010', '100'} then See("VSHLL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm3H);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let unsigned : boolean = (U == '1');
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operand, encoded in "U:imm3H":

U	imm3H	<dt>
0	001	S8
0	010	S16
0	100	S32
1	001	U8
1	010	U16
1	100	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = Din(m)[e*:esize];
        let result : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.142 VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instructions VRSHRN (zero) and VSHRN (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

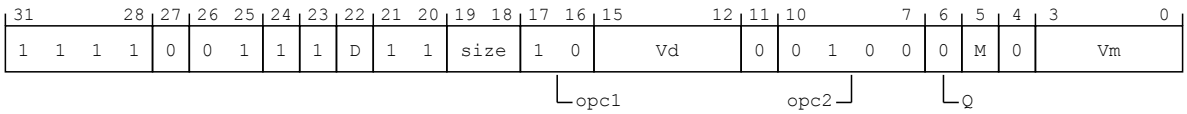

Encoding

```
VMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

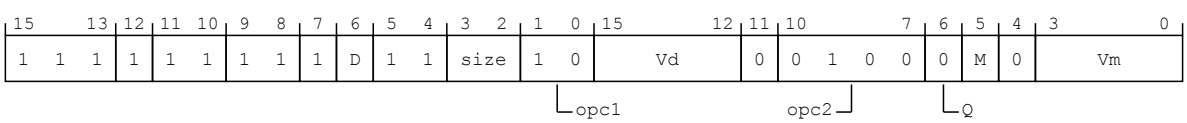

Encoding

```
VMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operand, encoded in "size":

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        D(d)[e*:esize] = Qin(m>>1)[e*:(2*esize)][esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.143 VMOVX

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_FP16)

31–28	27–26	25–23	22	21–16	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1 1	1 1	1 0 1	D	1 1 0 0 0 0	Vd	1	0	1 0	0	1	M	0	Vm
							size		op				

Encoding

```
VMOVX{<q>}.F16  <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1

(FEAT_FP16)

15–13	12	11–10	9–7	6	5–0	15–12	11	10	9–8	7	6	5	4	3–0
1 1 1	1	1 1	1 0 1	D	1 1 0 0 0 0	Vd	1	0	1 0	0	1	M	0	Vm
								size		op				

Encoding

```
VMOVX{<q>}.F16  <Sd>, <Sm>
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    H(d) = S(m)[31:16];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.144 VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the FPSCR, a form of the instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

When these settings permit the execution of Advanced SIMD and floating-point instructions, if the specified floating-point System register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when HCR.TID0 is set to 1, any `VMRS` access to FPSID from a Non-secure EL1 mode that would be permitted if HCR.TID0 was set to 0 generates a Hyp Trap exception. For more information, see EL2 configurable controls.

For simplicity, the `VMRS` pseudocode does not show the possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27 26	25 24	23–21	20	19–16	15–12	11	10–8	7–5	4	3–0
!=1111 (cond)	1 1	1 0	1 1 1	1 (L)	reg	Rt	1	0 1 0	(0) (0) (0)	1	(0) (0) (0) (0)

Encoding

```
VMRS{<c>}{<q>}  <Rt>, <spec_reg>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `! reg IN {'000x', '0101', '011x', '1000'}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value `0b1111`, the specified target register is the APSR.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the `Rt` field, R0 - R14.

T1

15–13	12	11 10	9 8	7–5	4	3–0	15–12	11	10–8	7–5	4	3–0
1 1 1	0	1 1	1 0	1 1 1	1 (L)	reg	Rt	1	0 1 0	(0) (0) (0)	1	(0) (0) (0) (0)

Encoding

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMRS{<c>}{<q>}  <Rt>, <spec_reg>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `! reg IN {'000x', '0101', '011x', '1000'}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value `0b1111`, the specified target register is the APSR.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the `Rt` field, R0 - R14.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Rt>

Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

R0-R14 General-purpose register.
APSR_nzcv Permitted only when `<spec_reg>` is `FPSCR`. Encoded as `0b1111`. The instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

<spec_reg>

Is the source Advanced SIMD and floating-point System register, encoded in "reg":

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
0100	UNPREDICTABLE
0101	MVFR2
0110	MVFR1
0111	MVFR0
1000	FPEXC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

reg	<spec_reg>
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                 // FPSCR
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.[N,Z,C,V] = FPSR().[N,Z,C,V];
        else
            R(t) = FPSCR();
        end;
    elsif PSTATE.EL == EL0 then
        Undefined();                        // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
        AArch32_CheckAdvSIMDOrFPRegisterTraps(reg);
        case reg of
            when '0000' =>  R(t) = FPSID();
            when '0101' =>  R(t) = MVFR2();
            when '0110' =>  R(t) = MVFR1();
            when '0111' =>  R(t) = MVFR0();
            when '1000' =>  R(t) = FPEXC();
            otherwise =>    unreachable;   // Dealt with above or in encoding-specific pseudocode
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.145 VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

When these settings permit the execution of Advanced SIMD and floating-point instructions:

- If the specified floating-point System register is FPSID or FPEXC, the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the FPSID and the instruction is executed in a mode other than User mode, the instruction is ignored.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27–26	25–24	23–21	20	19–16	15–12	11	10–8	7–5	4	3–0
!=1111	1 1	1 0	1 1 1	0	reg	Rt	1	0 1 0	(0) (0) (0)	1	(0) (0) (0) (0)
cond				L							

Encoding

```
VMSR{<c>}{<q>}  <spec_reg>, <Rt>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
if reg != '000x' && reg != '1000' then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF =>
            Undefined();
        when Constraint_NOP =>
            ExecuteAsNOP();
    end;
end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15 13	12	11 10	9 8	7 5	4	3 0	15 12	11	10 8	7 5	4	3 0
1 1 1	0	1 1	1 0	1 1 1	0	reg	Rt	1	0 1 0	(0) (0) (0)	1	(0) (0) (0) (0)
					L							

Encoding

```
VMSR{<c>}{<q>}  <spec_reg>, <Rt>
```

Decode for this encoding

```
let t : integer = UInt(Rt);
if reg != '000x' && reg != '1000' then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF =>
            Undefined();
        when Constraint_NOP =>
            ExecuteAsNOP();
    end;
end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<spec_reg>

Is the destination Advanced SIMD and floating-point System register, encoded in “reg”:

reg	<spec_reg>
0000	FPSID

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

reg	<spec_reg>
0001	FPSCR
001x	UNPREDICTABLE
01xx	UNPREDICTABLE
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

<Rt>

Is the general-purpose source register, encoded in the “Rt” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                 // FPSCR
        CheckVFPEnabled(TRUE);
        FPSCR() = R(t);
    elsif PSTATE.EL == EL0 then
        Undefined();                        // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
        case reg of
            when '0000' =>                    // VMSR access to FPSID is ignored
                pass;
            when '1000' =>
                FPEXC() = R(t);
            otherwise =>
                unreachable;   // Dealt with above or in encoding-specific pseudocode
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.146 VMUL (floating-point)

Vector Multiply (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

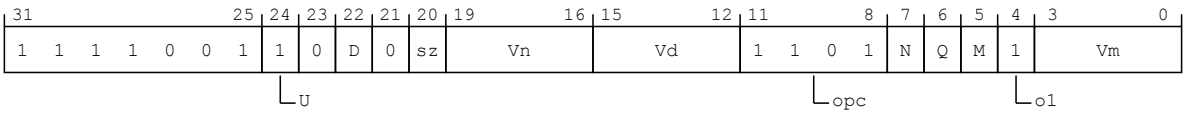

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMUL{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMUL{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

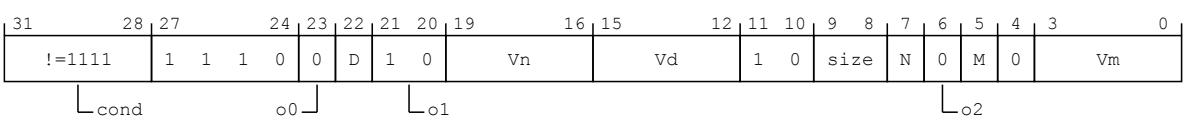

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMUL{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMUL{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMUL{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1	1 1 1 1 0	D	0	sz	Vn	Vd	1 1 0 1	N	Q	M	1	Vm
	U							opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMUL{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMUL{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3		0	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	1	1	1	0	0	D	1	0	Vn			Vd			1	0	size		N	0	M	0	Vm		
								o0			o1												o2					

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VMUL{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMUL{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMUL{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "A2 Double-precision scalar", "A2 Half-precision scalar", "A2 Single-precision scalar", "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPMul{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                H(d) = FPMul{16}(H(n), H(m), fpcr);
            when 32 =>
                S(d) = FPMul{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPMul{64}(D(n), D(m), fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.147 VMUL (integer and polynomial)

Vector Multiply (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors.

For information about multiplying polynomials, see Polynomial arithmetic over {0, 1}.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

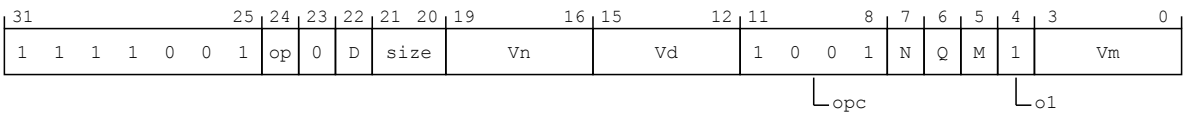

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMUL{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMUL{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let polynomial : boolean = (op == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

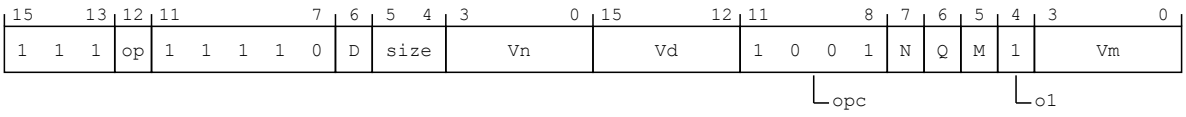

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMUL{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMUL{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let polynomial : boolean = (op == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “op:size”:

op	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	00	P8

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            var product : bits(2 * esize);
            if polynomial then
                product = PolynomialMult{esize, esize}(op1elt, op2elt);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                product = (element1 * element2)[2*esize-1:0];
            end;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = product;
            else
                D(d+r)[e*:esize] = product[esize-1:0];
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.148 VMUL (by scalar)

Vector Multiply (by scalar)

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1 0 0 1	Q	1	D	!=11 (size)	Vn	Vd	1 0 0	F	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMUL{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMUL{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	1 0 0	F	N	1	M	0	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMUL{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMUL{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE;  // "Don't care" value: TRUE produces same functionality
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register. When `<dt>` is I16 or F16, this is encoded in the “Vm<2:0>” field. Otherwise it is encoded in the “Vm” field.

<index>

Is the element index. When `<dt>` is I16 or F16, this is in the range 0 to 3 and is encoded in the “M:Vm<3>” field. Otherwise it is in the range 0 to 1 and is encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                D(d+r)[e*:esize] = FPMul{esize}(op1elt, op2elt, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = (element1 * element2)[2*esize-1:0];
                else
                    D(d+r)[e*:esize] = (element1 * element2)[esize-1:0];
                end;
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.149 VMULL (integer and polynomial)

Vector Multiply Long (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see Polynomial arithmetic over {0, 1}.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1	0	0	1	U	1	D	!=11	Vn	Vd	1	1	op	0	N	0	M	0	Vm
							size											

Encoding

```
VMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if op == '1' && (U == '1' || size == '01') then Undefined(); end;
if op == '1' && size =='10' && !IsFeatureImplemented(FEAT_PMULL) then Undefined(); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let polynomial : boolean = (op == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = if polynomial && size == '10' then 64 else 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	1	1	op	0	N	0	M	0	Vm
						size											

Encoding

```
VMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if op == '1' && (U == '1' || size == '01') then Undefined(); end;
if op == '1' && size == '10' && InITBlock() then UnpredictableProcedure(); end;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then
    UnpredictableProcedure();
end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let polynomial : boolean = (op == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = if polynomial && size == '10' then 64 else 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '1' && size == '10' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “op:U:size”:

op	U	size	<dt>
0	0	00	S8
0	0	01	S16
0	0	10	S32
0	1	00	U8
0	1	01	U16
0	1	10	U32
1	0	00	P8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	U	size	<dt>
1	0	10	P64

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            var product : bits(2 * esize);
            if polynomial then
                product = PolynomialMult{esize, esize}(op1elt, op2elt);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                product = (element1 * element2)[2*esize-1:0];
            end;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = product;
            else
                D(d+r)[e*:esize] = product[esize-1:0];
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.150 VMULL (by scalar)

Vector Multiply Long (by scalar)

Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

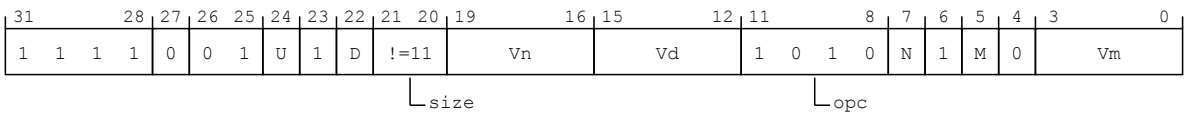

Encoding

```
VMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let floating_point : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

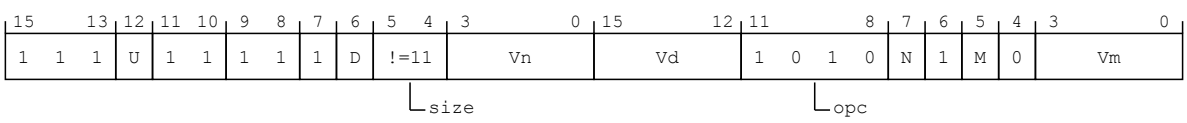

Encoding

```
VMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let floating_point : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in "U:size":

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when `<dt>` is S16 or U16, otherwise the "Vm" field.

<index>

Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when `<dt>` is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                D(d+r)[e*:esize] = FPMul{esize}(op1elt, op2elt, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = (element1 * element2)[2*esize-1:0];
                else
                    D(d+r)[e*:esize] = (element1 * element2)[esize-1:0];
                end;
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.151 VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

31–25	24	23	22	21–19	18–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	i	1	D	0 0 0	imm3	Vd	0 x x 0 (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}.I32  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}.I32  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31–25	24	23	22	21–19	18–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	i	1	D	0 0 0	imm3	Vd	1 0 x 0 (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}.I16  <Dd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}.I16  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A3

31						25	24	23	22	21		19	18		16	15		12	11			8	7	6	5	4	3		0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			1	1	0	x	0	Q	1	1	imm4		
																					cmode				op				

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}.I32  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}.I32  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15		13	12	11				7	6	5		3	2		0	15		12	11			8	7	6	5	4	3		0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			0	x	x	0	0	Q	1	1	imm4		
																					cmode				op				

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}.I32  <Dd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}.I32  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 0 x 0 (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}.I16  <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}.I16  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T3

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 1 0 x (cmode)	0	Q	1 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}.I32  <Dd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}.I32  <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “A3 128-bit SIMD vector”, and “A3 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, “T2 64-bit SIMD vector”, “T3 128-bit SIMD vector”, and “T3 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = NOT(imm64);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.152 VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 1 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}{.<dt>}  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}{.<dt>}  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0	1 0 1 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VMVN{<c>}{<q>}{.<dt>}  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VMVN{<c>}{<q>}{.<dt>}  <Qd>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = NOT(D(m+r));
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.153 VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

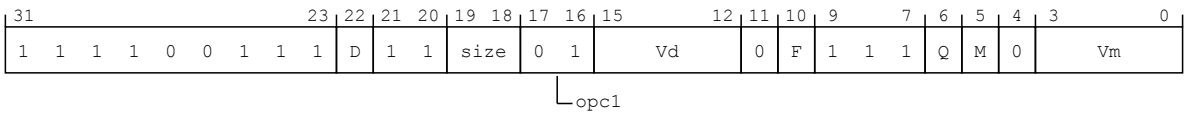

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VNEG{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VNEG{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

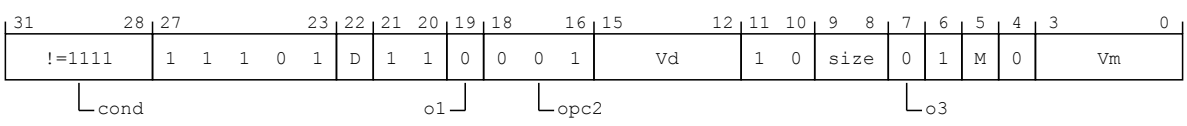

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNEG{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNEG{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNEG{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15								7	6	5	4	3	2	1	0	15			12	11	10	9			7	6	5	4	3				0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	1		Q	M	0	Vm				
														opc1																			

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VNEG{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VNEG{<c>}{<q>}.<dt>  <Qd>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    Undefined();
end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15–7	6	5–4	3	2–0	15–12	11–10	9–8	7	6	5	4	3–0
1 1 1 0 1 1 1 0 1	D	1 1	0	0 0 1	Vd	1 0	size	0	1	M	0	Vm
			o1	opc2				o3				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNEG{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNEG{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNEG{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Half-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Half-precision scalar”, and “T2 Single-precision scalar” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                if floating_point then
                    D(d+r)[e*:esize] = FPNeg{esize}(D(m+r)[e*:esize], fpcr);
                else
                    let result : integer = -SInt(D(m+r)[e*:esize]);
                    D(d+r)[e*:esize] = result[esize-1:0];
                end;
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 => H(d) = FPNeg{16}(H(m), fpcr);
            when 32 => S(d) = FPNeg{32}(S(m), fpcr);
            when 64 => D(d) = FPNeg{64}(D(m), fpcr);
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.154 VNMLA

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Note

Arm recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

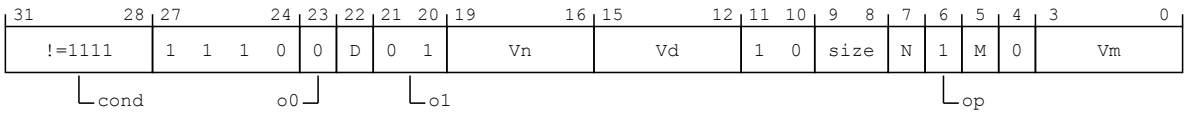

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNMLA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNMLA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNMLA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	0	D	0	1	Vn		Vd		1	0	size		N	1	M	0	Vm	
								o0			o1										op				

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNMLA{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNMLA{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNMLA{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let product16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr),
                                                          FPNeg{16}(product16, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr), product16, fpcr);
                when VFPNegMul_VNMUL =>  H(d) = FPNeg{16}(product16, fpcr);
            end;
        when 32 =>
            let product32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr),
                                                          FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL =>  S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr),
                                                          FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL =>  D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.155 VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

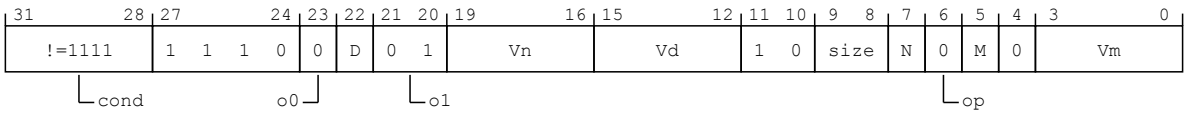

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNMLS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNMLS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNMLS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15							8	7	6	5	4	3			0	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	0	D	0	1		Vn				Vd			1	0	size		N	0	M	0		Vm		
								o0			o1														op						

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNMLS{<c>}{<q>}.F16  <Sd>, <Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNMLS{<c>}{<q>}.F32  <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNMLS{<c>}{<q>}.F64  <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let product16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr),
                                                          FPNeg{16}(product16, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr), product16, fpcr);
                when VFPNegMul_VNMUL =>  H(d) = FPNeg{16}(product16, fpcr);
            end;
        when 32 =>
            let product32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr),
                                                          FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL =>  S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr),
                                                          FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL =>  D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.156 VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

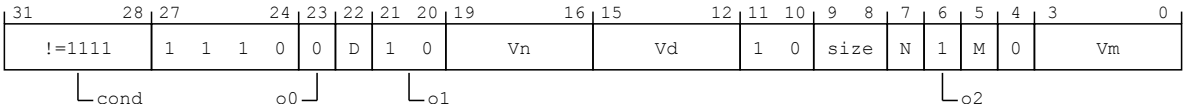

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNMUL{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNMUL{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNMUL{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let vtype : VFPNegMul = VFPNegMul_VNMUL;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15							8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	0	D	1	0	Vn		Vd		1	0	size		N	1	M	0	Vm	
								o0			o1														

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VNMUL{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VNMUL{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VNMUL{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let vtype : VFPNegMul = VFPNegMul_VNMUL;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let product16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr),
                                                          FPNeg{16}(product16, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr), product16, fpcr);
                when VFPNegMul_VNMUL =>  H(d) = FPNeg{16}(product16, fpcr);
            end;
        when 32 =>
            let product32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr),
                                                          FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL =>  S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA =>  D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr),
                                                          FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS =>  D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL =>  D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.157 VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

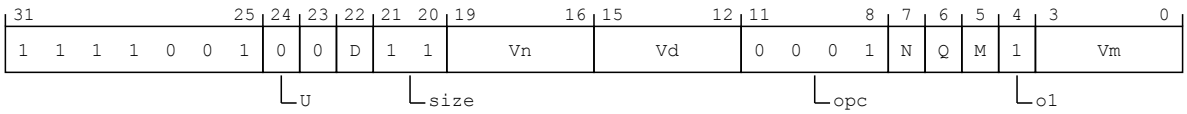

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORN{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORN{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

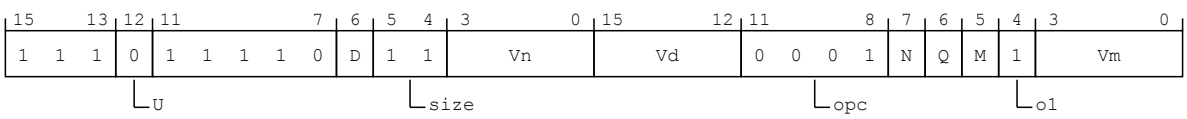

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORN{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORN{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) OR NOT(D(m+r));
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.158 VORN (immediate)

Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of VORR (immediate). This means:

- The encodings in this description are named to match the encodings of VORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–25	24	23	22	21–19	18–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	i	1	D	0 0 0	imm3	Vd	0 x x 1 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORN{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32  <Dd>, #~<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORN{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32  <Qd>, #~<imm>
```

A2

31–25	24	23	22	21–19	18–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	i	1	D	0 0 0	imm3	Vd	1 0 x 1 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORN{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16  <Dd>, #~<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORN{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16  <Qd>, #~<imm>
```

T1

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	0 x x 1	0	Q	0	1	imm4
							cmode			op		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORN{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32  <Dd>, #~<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORN{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32  <Qd>, #~<imm>
```

T2

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 0 x 1	0	Q	0	1	imm4
							cmode			op		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORN{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16  <Dd>, #~<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORN{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16  <Qd>, #~<imm>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

The description of VORR (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of VORR (immediate) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.159 VORR (immediate)

Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VORN (immediate).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–25	24	23	22	21–19	18–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	i	1	D	0 0 0	imm3	Vd	0 x x 1 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORR{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORR{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31–25	24	23	22	21–19	18–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	i	1	D	0 0 0	imm3	Vd	1 0 x 1 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORR{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORR{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	0 x x 1 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORR{<c>}{<q>}.I32  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORR{<c>}{<q>}.I32  {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15–13	12	11–7	6	5–3	2–0	15–12	11–8	7	6	5	4	3–0
1 1 1	i	1 1 1 1 1	D	0 0 0	imm3	Vd	1 0 x 1 (cmode)	0	Q	0 (op)	1	imm4

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORR{<c>}{<q>}.I16  {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORR{<c>}{<q>}.I16  {<Qd>, }<Qd>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(d+r) OR imm64;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.160 VORR (register)

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the alias VMOV (register, SIMD).

This instruction is used by the pseudo-instructions VRSHR (zero) and VSHR (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

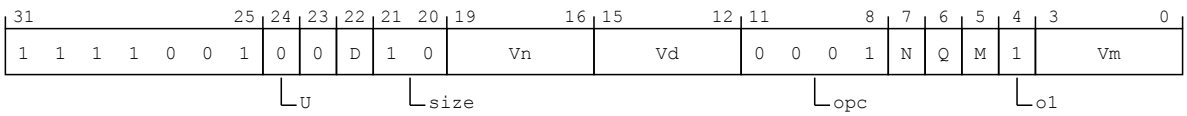

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

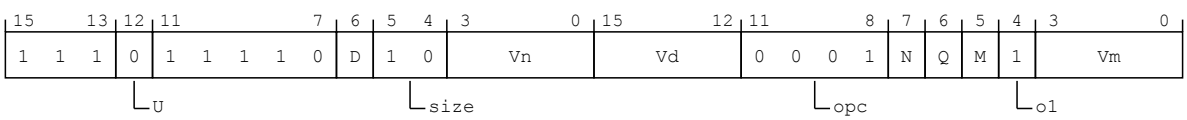

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Alias Conditions

Alias	Is preferred when
`VMOV (register, SIMD)`	`N : Vn == M : Vm`
`VRSHR (zero)`	`Never`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
VSHR (zero)	Never

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) OR D(m+r);
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.161 VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

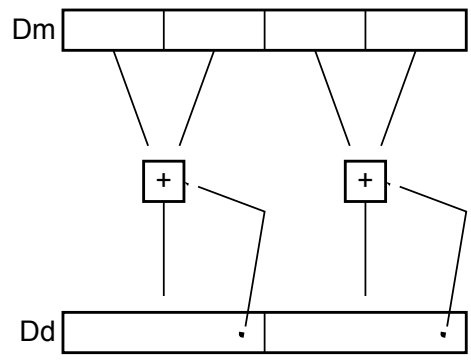

Figure F6-2 VPADAL doubleword operation for data type S16

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31								23	22	21	20	19	18	17	16	15			12	11			8	7	6	5	4	3			0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			
														opc1																	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VPADAL{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VPADAL{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			8	7	6	5	4	3			0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			
														opc1																	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VPADAL{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VPADAL{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “op:size”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
x	11	RESERVED
1	00	U8
1	01	U16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	size	<dt>
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        for e = 0 to h-1 do
            let op1elt : bits(esize) = D(m+r)[(2*e)*:esize];
            let op2elt : bits(esize) = D(m+r)[(2*e+1)*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = element1 + element2;
            D(d+r)[e*:(2*esize)] = D(d+r)[e*:(2*esize)] + result;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.162 VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

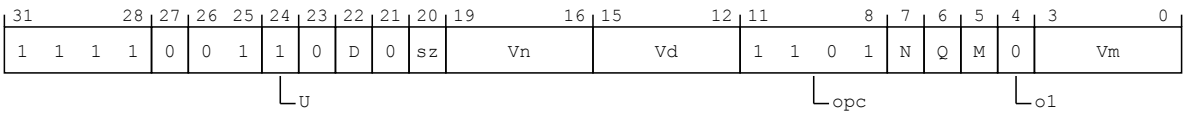

Encoding

```
VPADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if Q == '1' then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

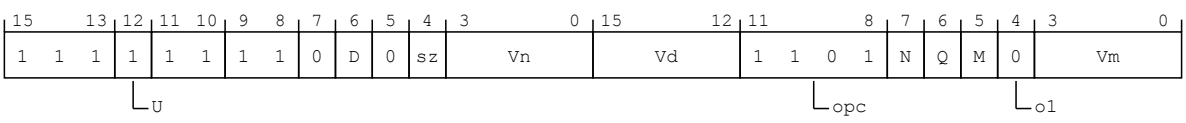

Encoding

```
VPADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if Q == '1' then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        dest[e*:esize]   = FPAdd{esize}(D(n)[(2*e)*:esize], D(n)[(2*e+1)*:esize], fpcr);
        dest[(e+h)*:esize] = FPAdd{esize}(D(m)[(2*e)*:esize], D(m)[(2*e+1)*:esize], fpcr);
    end;

    D(d) = dest;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.163 VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

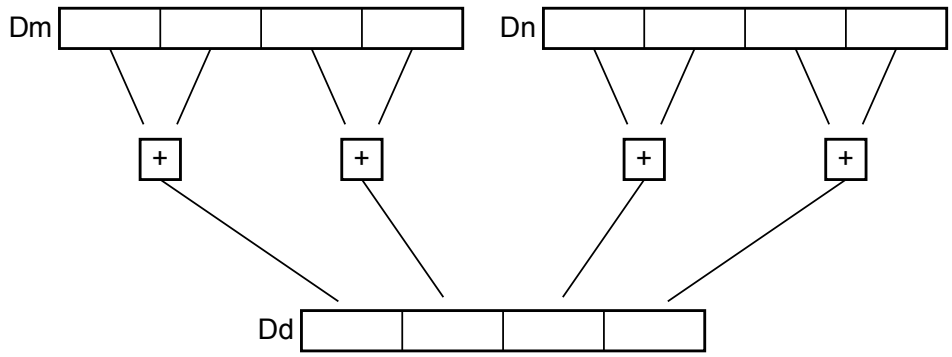

Figure F6-3 VPADD doubleword operation for data type I16

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0	1	0	0	D	size	Vn	Vd	1 0 1 1	N	Q	M	1	Vm
				U						opc				o1	

Encoding

```
VPADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' || Q == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1	1 1	0	D	size	Vn	Vd	1 0 1 1	N	Q	M	1	Vm
	U								opc				o1	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
VPADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' || Q == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "size":

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        dest[e*:esize]   = D(n)[(2*e)*:esize] + D(n)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = D(m)[(2*e)*:esize] + D(m)[(2*e+1)*:esize];
    end;

    D(d) = dest;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.164 VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

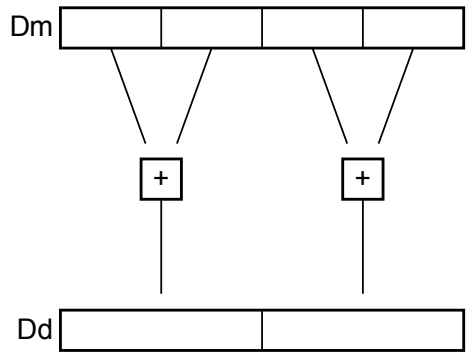

Figure F6-4 VPADDL doubleword operation for data type S16

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31								23	22	21	20	19	18	17	16	15	12	11			8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd		0	0	1	0	op	Q	M	0	Vm	

opc1

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VPADDL{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VPADDL{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			8	7	6	5	4	3			0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			
														opc1																	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VPADDL{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VPADDL{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “op:size”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
x	11	RESERVED
1	00	U8
1	01	U16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	size	<dt>
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        for e = 0 to h-1 do
            let op1elt : bits(esize) = D(m+r)[(2*e)*:esize];
            let op2elt : bits(esize) = D(m+r)[(2*e+1)*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = element1 + element2;
            D(d+r)[e*:(2*esize)] = result[2*esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.165 VPMAX (floating-point)

Vector Pairwise Maximum (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

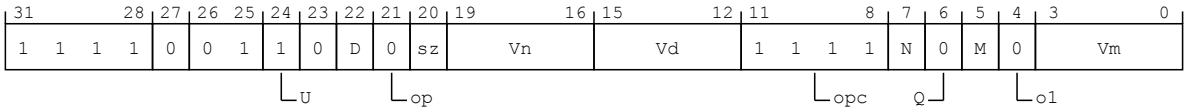

Encoding

```
VPMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

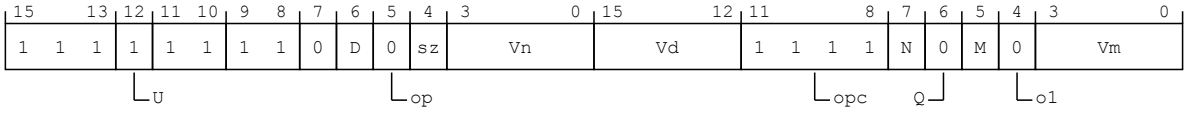

Encoding

```
VPMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let fpcr : FPCR_Type = StandardFPCR();
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        dest[e*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                          else FPMin{esize}(op1elt, op2elt, fpcr));
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                              else FPMin{esize}(op1elt, op2elt, fpcr));
    end;

    D(d) = dest;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.166 VPMAX (integer)

Vector Pairwise Maximum (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

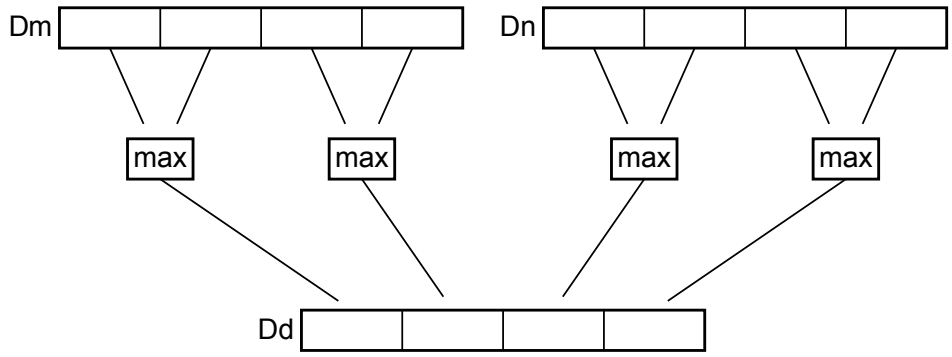

Figure F6-5 VPMAX doubleword operation for data type S16 or U16

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0	1	U	0	D	size	Vn	Vd	1 0 1 0 (opc)	N	0 (Q)	M	0 (op)	Vm

Encoding

```
VPMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	0	D	size	Vn	Vd	1 0 1 0 (opc)	N	0 (Q)	M	0 (op)	Vm

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
VPMAX{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        var element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        var element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        var result : integer = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[e*:esize] = result[esize-1:0];
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        element2 = if unsigned then UInt(op2elt) else SInt(op2elt);
        result = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[(e+h)*:esize] = result[esize-1:0];
    end;

    D(d) = dest;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.167 VPMIN (floating-point)

Vector Pairwise Minimum (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

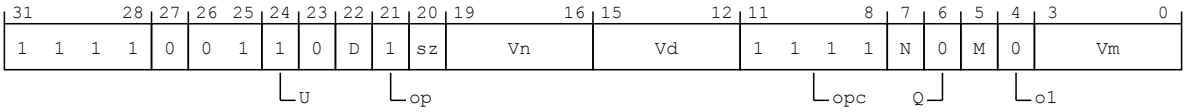

Encoding

```
VPMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

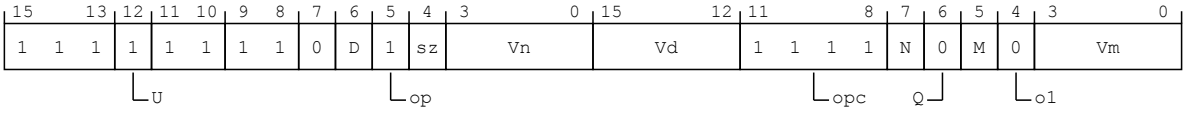

Encoding

```
VPMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let fpcr : FPCR_Type = StandardFPCR();
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        dest[e*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                          else FPMin{esize}(op1elt, op2elt, fpcr));
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                              else FPMin{esize}(op1elt, op2elt, fpcr));
    end;

    D(d) = dest;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.168 VPMIN (integer)

Vector Pairwise Minimum (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	0	D	size	Vn	Vd	1 0 1 0	N	0	M	1	Vm
									opc		Q		op	

Encoding

```
VPMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	0	D	size	Vn	Vd	1 0 1 0	N	0	M	1	Vm
									opc		Q		op	

Encoding

```
VPMIN{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        var element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        var element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        var result : integer = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[e*:esize] = result[esize-1:0];
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        element2 = if unsigned then UInt(op2elt) else SInt(op2elt);
        result = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[(e+h)*:esize] = result[esize-1:0];
    end;

    D(d) = dest;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.169 VPOP

Pop SIMD&FP registers from stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack.

This is an alias of VLDM, VLDMDB, VLDMIA. This means:

- The encodings in this description are named to match the encodings of VLDM, VLDMDB, VLDMIA.
- The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

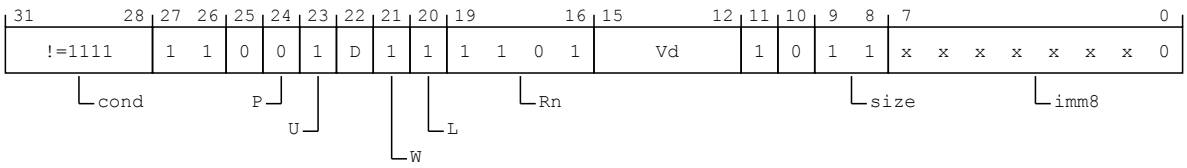

Encoding for the Increment After variant

```
VPOP{<c>}{<q>}{.<size>}  <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>}  SP!, <dreglist>
```

and is always the preferred disassembly.

A2

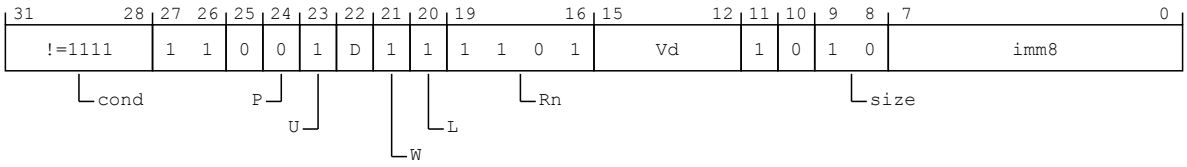

Encoding for the Increment After variant

```
VPOP{<c>}{<q>}{.<size>}  <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>}  SP!, <sreglist>
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

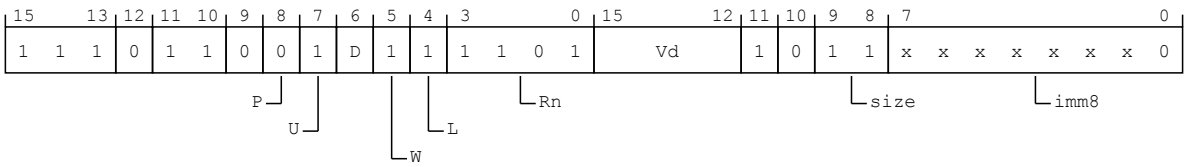

Encoding for the Increment After variant

```
VPOP{<c>}{<q>}{.<size>}  <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>}  SP!, <dreglist>
```

and is always the preferred disassembly.

T2

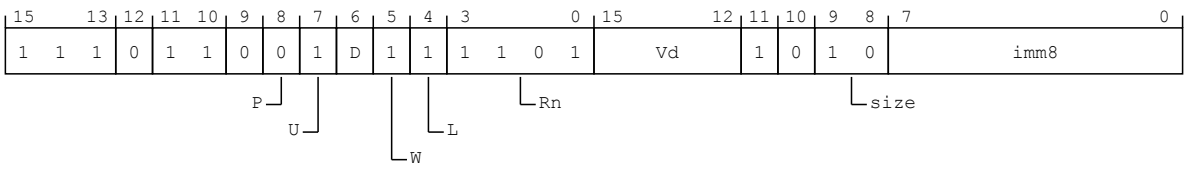

Encoding for the Increment After variant

```
VPOP{<c>}{<q>}{.<size>}  <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>}  SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Operation

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

Operational Information

The description of VLDM, VLDMDB, VLDMIA gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.170 VPUSH

Push SIMD&FP registers to stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack.

This is an alias of VSTM, VSTMDB, VSTMIA. This means:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

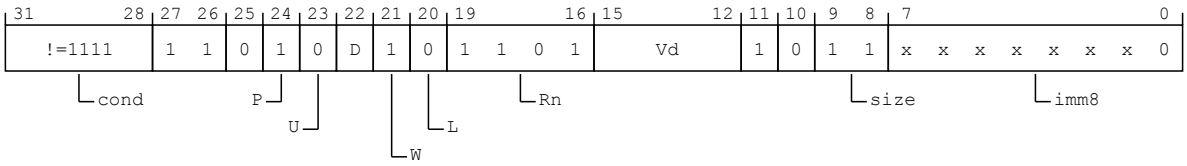

Encoding for the Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>}  <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>}  SP!, <dreglist>
```

and is always the preferred disassembly.

A2

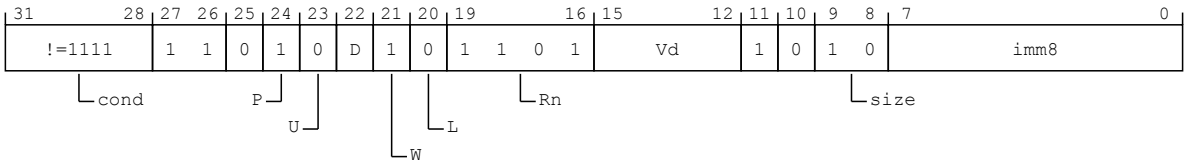

Encoding for the Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>}  <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>}  SP!, <sreglist>
```

and is always the preferred disassembly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

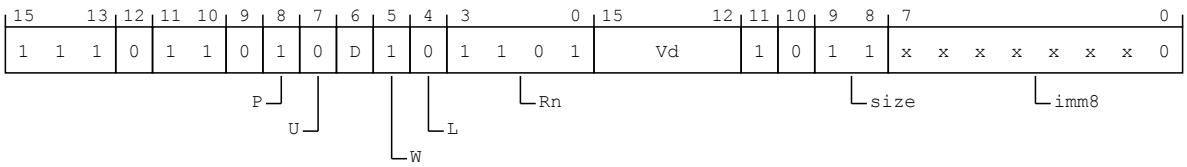

Encoding for the Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>}  <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>}  SP!, <dreglist>
```

and is always the preferred disassembly.

T2

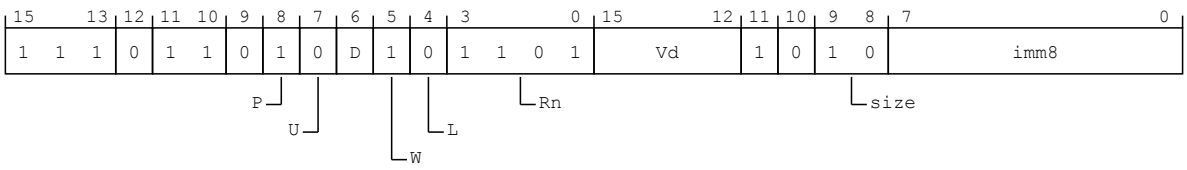

Encoding for the Decrement Before variant

```
VPUSH{<c>}{<q>}{.<size>}  <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>}  SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Operation

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

Operational Information

The description of VSTM, VSTMDB, VSTMIA gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.171 VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

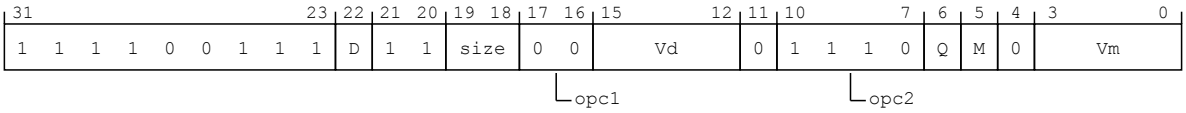

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQABS{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQABS{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

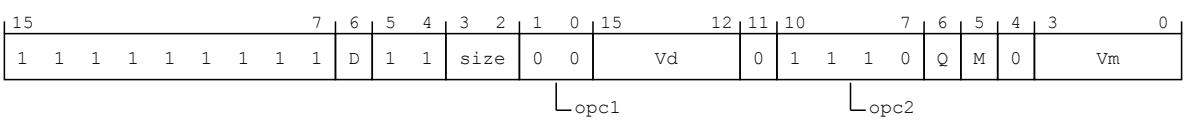

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQABS{<c>}{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQABS{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let result : integer = Abs(SInt(D(m+r)[e*:esize]));
            let (res, sat) : (bits(esize), boolean) = SignedSatQ{esize}(result);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.172 VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

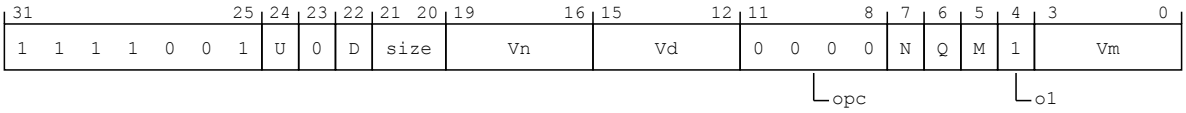

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

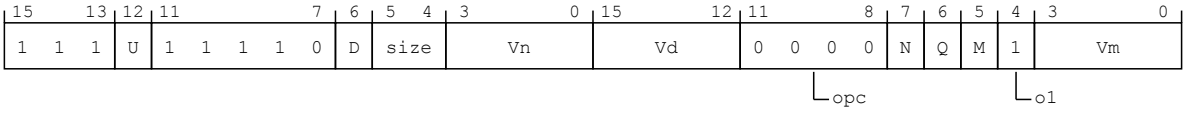

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let sum : integer = element1 + element2;
            let (res, sat) : (bits(esize), boolean) = SatQ{esize}(sum, unsigned);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.173 VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

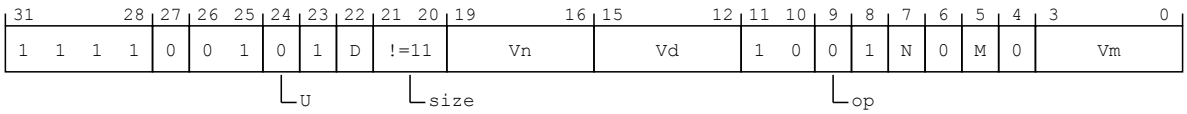

Encoding

```
VQDMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

A2

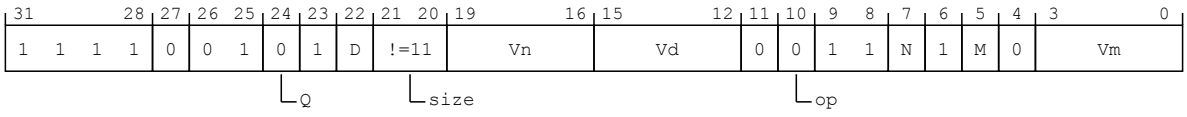

Encoding

```
VQDMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15 13	12	11 10	9 8	7	6	5 4	3 0	15 12	11 10	9	8	7	6	5	4	3 0
1 1 1	0	1 1	1 1	1	D	!=11	Vn	Vd	1 0	0	1	N	0	M	0	Vm
	U					size				op						

Encoding

```
VQDMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

T2

15 13	12	11 10	9 8	7	6	5 4	3 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1	0	1 1	1 1	1	D	!=11	Vn	Vd	0	0	1 1	N	1	M	0	Vm
	Q					size				op						

Encoding

```
VQDMLAL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” and “A2” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” and “T2” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

For the “A1” and “T1” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

For the “A2” and “T2” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field when `<dt>` is S16, otherwise the “Vm” field.

<index>

Is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field when `<dt>` is S16, otherwise in range 0 to 1, encoded in the “M” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat1) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        var result : integer;
        if add then
            result = SInt(Qin(d>>1)[e*:(2*esize)]) + SInt(product);
        else
            result = SInt(Qin(d>>1)[e*:(2*esize)]) - SInt(product);
        end;
        var sat2 : boolean;
        var res : bits(2*esize);
        (res, sat2) = SignedSatQ{2*esize}(result);
        Q(d>>1)[e*:(2*esize)] = res;
        if sat1 || sat2 then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.174 VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1	0	0	1	0	1	D	!=11	Vn	Vd	1	0	1	1	N	0	M	0	Vm
				U			size					op						

Encoding

```
VQDMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

A2

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1	0	0	1	0	1	D	!=11	Vn	Vd	0	1	1	1	N	1	M	0	Vm
				Q			size				op							

Encoding

```
VQDMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15 13	12	11 10	9 8	7	6	5 4	3 0	15 12	11 10	9	8	7	6	5	4	3 0
1 1 1	0	1 1	1 1	1	D	!=11	Vn	Vd	1 0	1	1	N	0	M	0	Vm
	U					size				op						

Encoding

```
VQDMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

T2

15 13	12	11 10	9 8	7	6	5 4	3 0	15 12	11	10	9 8	7	6	5	4	3 0
1 1 1	0	1 1	1 1	1	D	!=11	Vn	Vd	0	1	1 1	N	1	M	0	Vm
	Q					size				op						

Encoding

```
VQDMLSL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>[<index>]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” and “A2” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” and “T2” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

For the “A1” and “T1” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

For the “A2” and “T2” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field when `<dt>` is S16, otherwise the “Vm” field.

<index>

Is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field when `<dt>` is S16, otherwise in range 0 to 1, encoded in the “M” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat1) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        var result : integer;
        if add then
            result = SInt(Qin(d>>1)[e*:(2*esize)]) + SInt(product);
        else
            result = SInt(Qin(d>>1)[e*:(2*esize)]) - SInt(product);
        end;
        let (res, sat2) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(result);
        Q(d>>1)[e*:(2*esize)] = res;
        if sat1 || sat2 then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.175 VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see VQRDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

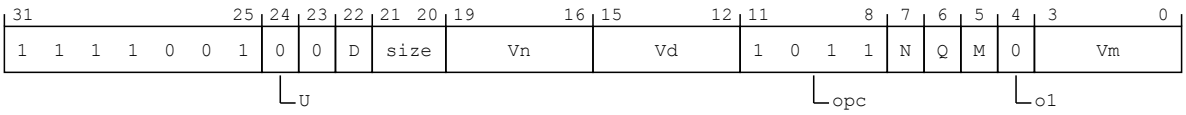

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

A2

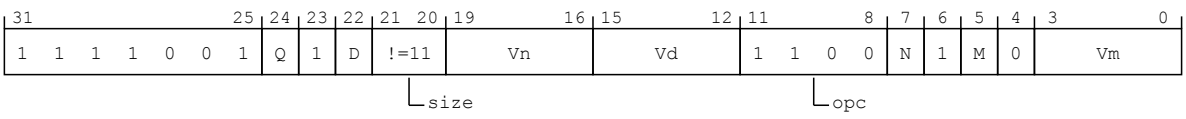

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VQDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15 13	12	11 7	6	5 4	3 0	15 12	11 8	7	6	5	4	3 0
1 1 1	0	1 1 1 1 0	D	size	Vn	Vd	1 0 1 1	N	Q	M	0	Vm
	U						opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	1 1 0 0 (opc)	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(D(m)[index*:esize]); end;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if !scalar_form then op2 = SInt(D(m+r)[e*:esize]); end;
            let op1 : integer = SInt(D(n+r)[e*:esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}((2*op1*op2) >> esize);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.176 VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0	1	0	1	D	!=11	Vn	Vd	1 1 0 1	N	0	M	0	Vm
				U			size			opc					

Encoding

```
VQDMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

A2

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1	0	0	1	0	1	D	!=11	Vn	Vd	1 0 1 1	N	1	M	0	Vm
				Q			size			opc					

Encoding

```
VQDMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm[x]>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1	1 1	1	D	!=11	Vn	Vd	1 1 0 1	N	0	M	0	Vm
	U					size			opc					

Encoding

```
VQDMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

T2

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1	1 1	1	D	!=11	Vn	Vd	1 0 1 1	N	1	M	0	Vm
	Q					size			opc					

Encoding

```
VQDMULL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm[x]>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” and “A2” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” and “T2” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        Q(d>>1)[e*:(2*esize)] = product;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.177 VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instructions VQRSHRN (zero), VQRSHRUN (zero), VQSHRN (zero), and VQSHRUN (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31								23	22	21	20	19	18	17	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd		0	0	1	0	op		M	0	Vm	

opc1

Encoding for the Signed result variant

Applies when `(op == 1x)`

```
VQMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Encoding for the Unsigned result variant

Applies when `(op == 01)`

```
VQMOVUN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Decode for all variants of this encoding

```
if op == '00' then See("VMOVN"); end;
if size == '11' || Vm[0] == '1' then Undefined(); end;
let src_unsigned : boolean = (op == '11');
let dest_unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15 … 7	6	5	4	3	2	1	0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1 1 1 1 1 1 1 1 1	D	1	1	size		1	0	Vd	0	0	1	0	op		M	0	Vm

opc1

Encoding for the Signed result variant

Applies when `(op == 1x)`

```
VQMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Encoding for the Unsigned result variant

Applies when `(op == 01)`

```
VQMOVUN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Decode for all variants of this encoding

```
if op == '00' then See("VMOVN"); end;
if size == '11' || Vm[0] == '1' then Undefined(); end;
let src_unsigned : boolean = (op == '11');
let dest_unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 Signed result” and “A1 Unsigned result” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Signed result” and “T1 Unsigned result” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 Signed result” and “T1 Signed result” variants: is the data type for the elements of the operand, encoded in “op:size”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op	size	<dt>
11	01	U32
11	10	U64

For the “A1 Unsigned result” and “T1 Unsigned result” variants: is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e*:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let (res, sat) : (bits(esize), boolean) = SatQ{esize}(element, dest_unsigned);
        D(d)[e*:esize] = res;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.178 VQNEG

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

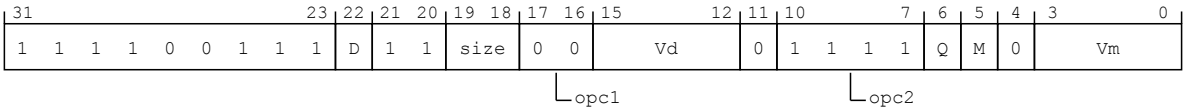

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQNEG{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQNEG{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

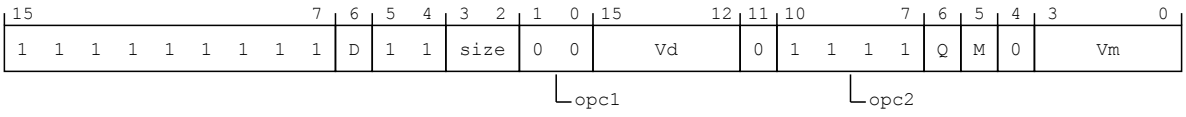

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQNEG{<c>}{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQNEG{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let result : integer = -SInt(D(m+r)[e*:esize]);
            let (res, sat) : (bits(esize), boolean) = SignedSatQ{esize}(result);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.179 VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

(FEAT_RDM)

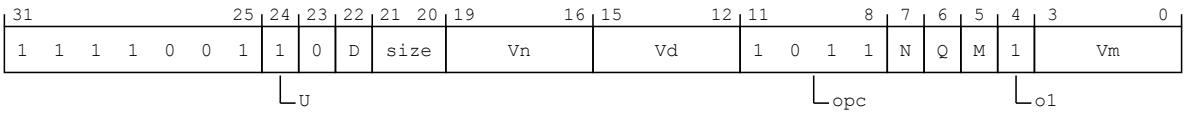

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLAH{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLAH{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

(FEAT_RDM)

31						25	24	23	22	21 20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	Q	1	D	!=11	Vn		Vd		1 1 1 0		N	1	M	0	Vm	
										size					opc							

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLAH{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLAH{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

(FEAT_RDM)

15	13	12	11	7	6	5 4	3	0	15	12	11	8	7	6	5	4	3	0
1 1 1		1	1 1 1 1 0		D	size	Vn		Vd		1 0 1 1		N	Q	M	1	Vm	
		U									opc					o1		

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLAH{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLAH{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

(FEAT_RDM)

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	1 1 1 0 (opc)	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLAH{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLAH{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "size":

size	<dt>
01	S16
10	S32

<Dd>

Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
var operand2 : integer;
let round : boolean = TRUE;
if scalar_form then operand2 = SInt(D(m)[index*:esize]); end;
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : integer = SInt(D(n+r)[e*:esize]);
        let operand3 : integer = SInt(D(d+r)[e*:esize]) << esize;
        if !scalar_form then operand2 = SInt(D(m+r)[e*:esize]); end;
        let rdmlah : integer = RShr(operand3 + 2*(operand1*operand2), esize, round);
        let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmlah);
        D(d+r)[e*:esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.180 VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

(FEAT_RDM)

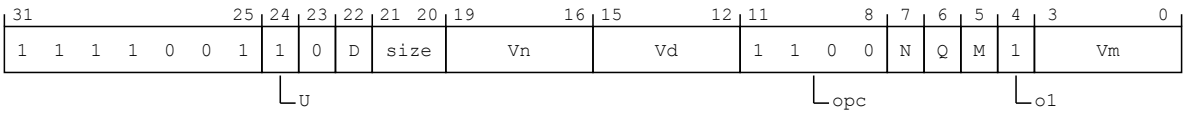

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLSH{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLSH{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

(FEAT_RDM)

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	Q	1	D	!=11 (size)	Vn	Vd	1 1 1 1 (opc)	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLSH{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLSH{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

(FEAT_RDM)

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1 (U)	1 1 1 1 0	D	size	Vn	Vd	1 1 0 0 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLSH{<q>}.<dt>  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLSH{<q>}.<dt>  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

(FEAT_RDM)

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	1 1 1 1 (opc)	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMLSH{<q>}.<dt>  <Dd>, <Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMLSH{<q>}.<dt>  <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "size":

size	<dt>
01	S16
10	S32

<Dd>

Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in "Vm<2:0>", and `x` is encoded in "M:Vm<3>". If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in "Vm", and `x` is encoded in "M".

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
var operand2 : integer;
let round : boolean = TRUE;
if scalar_form then operand2 = SInt(D(m)[index*:esize]); end;
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : integer = SInt(D(n+r)[e*:esize]);
        let operand3 : integer = SInt(D(d+r)[e*:esize]) << esize;
        if !scalar_form then operand2 = SInt(D(m+r)[e*:esize]); end;
        let rdmlsh : integer = RShr(operand3 - 2*(operand1*operand2), esize, round);
        let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmlsh);
        D(d+r)[e*:esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.181 VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see VQDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

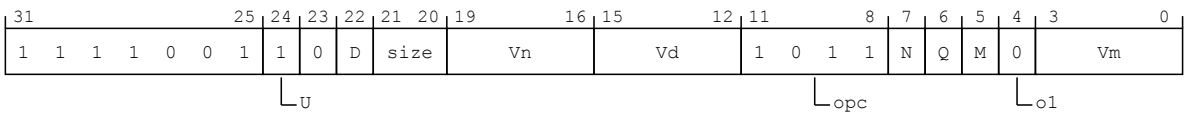

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

A2

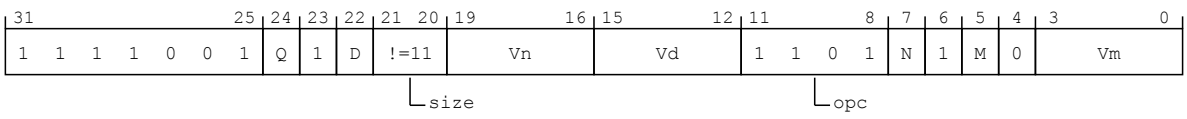

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VQRDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	1	1 1 1 1 0	D	size	Vn	Vd	1 0 1 1	N	Q	M	0	Vm
	U						opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	Q	1 1 1 1 1	D	!=11 (size)	Vn	Vd	1 1 0 1 (opc)	N	1	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRDMULH{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. `Dm` is encoded in “Vm<2:0>”, and `x` is encoded in “M:Vm<3>”. If `<dt>` is `S32`, `Dm` is restricted to D0-D15. `Dm` is encoded in “Vm”, and `x` is encoded in “M”.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    let round : boolean = TRUE;
    if scalar_form then op2 = SInt(D(m)[index*:esize]); end;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : integer = SInt(D(n+r)[e*:esize]);
            if !scalar_form then op2 = SInt(D(m+r)[e*:esize]); end;
            let rdmulh : integer = RShr(2*op1*op2, esize, round);
            let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmulh);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.182 VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see VQSHL (register).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 0 1 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 0 1 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQRSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQRSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	<dt>
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : bits(esize);
    var sat : boolean;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            var element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            if shift >= 0 then // left shift
                element = element << shift;
            else               // rounding right shift
                shift = -shift;
                element = (element + (1 << (shift - 1))) >> shift;
            end;
            (result, sat) = SatQ{esize}(element, unsigned);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.183 VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see VQSHRN and VQSHRUN.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

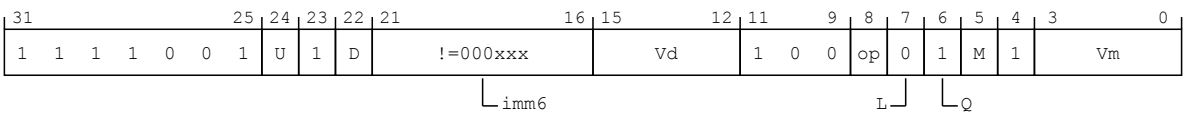

Encoding for the Signed result variant

Applies when `(op == 1)`

```
VQRSHRN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Encoding for the Unsigned result variant

Applies when `(U == 1 && op == 0)`

```
VQRSHRUN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VRSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15	13	12	11	7	6	5	0	15	12	11	9	8	7	6	5	4	3	0
1 1 1		U	1 1 1 1 1		D	!=000xxx		Vd		1 0 0		op	0	1	M	1	Vm	
						imm6							L	Q				

Encoding for the Signed result variant

Applies when `(op == 1)`

```
VQRSHRN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Encoding for the Unsigned result variant

Applies when `(U == 1 && op == 0)`

```
VQRSHRUN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VRSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Signed result” and “A1 Unsigned result” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Signed result” and “T1 Unsigned result” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

For the “A1 Signed result” and “T1 Signed result” variants: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1	U

For the “A1 Unsigned result” and “T1 Unsigned result” variants: is S when “U” is 1, and is encoded in “U”.

<size>

Is the data size for the elements of the vectors, encoded in “imm6”:

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to `<size>`/2, encoded in the “imm6” field as `<size>`/2 - `<imm>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e*:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let rshrn : integer = RShr(element, shift_amount, round);
        let (result, sat) : (bits(esize), boolean) = SatQ{esize}(rshrn, dest_unsigned);
        D(d)[e*:esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.184 VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

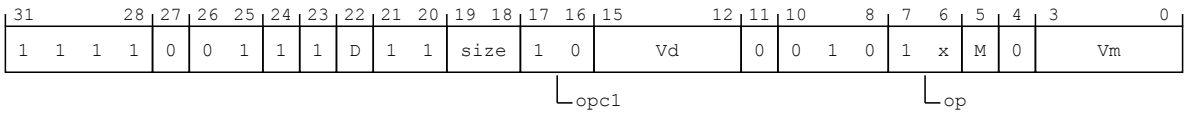

Encoding for the Signed result variant

```
VQRSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

T1

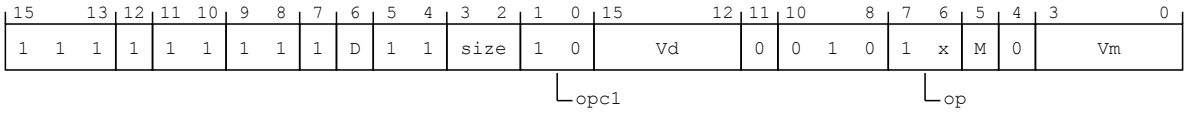

Encoding for the Signed result variant

```
VQRSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Assembler Symbols

<c>

For the “A1 Signed result” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Signed result” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt>

Is the data type for the elements of the operand, encoded in “op:size”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.185 VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21	20	19–18	17	16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm
										opc1							op				

Encoding for the Unsigned result variant

```
VQRSHRUN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

T1

15–13	12	11	10	9	8	7	6	5	4	3–2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1	1	1	1	1	1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm
											opc1							op				

Encoding for the Unsigned result variant

```
VQRSHRUN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Assembler Symbols

<c>

For the “A1 Unsigned result” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Unsigned result” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.186 VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	1	D	imm6	Vd	0 1 1	op	L	Q	M	1	Vm

Encoding for the 64-bit SIMD vector, signed result variant

Applies when `(op == 1 && Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHL{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when `(U == 1 && op == 0 && Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHLU{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector, signed result variant

Applies when `(op == 1 && Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHL{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when `(U == 1 && op == 0 && Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHLU{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	13	12	11	7	6	5	0	15	12	11	9	8	7	6	5	4	3	0
1 1 1		U	1 1 1 1 1		D	imm6		Vd		0 1 1		op	L	Q	M	1	Vm	

Encoding for the 64-bit SIMD vector, signed result variant

Applies when `(op == 1 && Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHL{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when `(U == 1 && op == 0 && Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHLU{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector, signed result variant

Applies when `(op == 1 && Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHL{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when `(U == 1 && op == 0 && Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VQSHLU{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector, signed result”, “A1 128-bit SIMD vector, unsigned result”, “A1 64-bit SIMD vector, signed result”, and “A1 64-bit SIMD vector, unsigned result” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector, signed result”, “T1 128-bit SIMD vector, unsigned result”, “T1 64-bit SIMD vector, signed result”, and “T1 64-bit SIMD vector, unsigned result” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 0 to `<size>`-1, encoded in the “imm6” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
            let (result, sat) : (bits(esize), boolean) = SatQ{esize}(element << shift_amount,
                                                                     dest_unsigned);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.187 VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see VQRSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 0 0 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 0 0 (opc)	N	Q	M	1 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	<dt>
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var sat : boolean;
            var result : bits(esize);
            if shift >= 0 then
                (result,sat) = SatQ{esize}(element << shift, unsigned);
            else
                (result,sat) = SatQ{esize}(element >> -shift, unsigned);
            end;
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.188 VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see VQRSHRN and VQRSHRUN.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

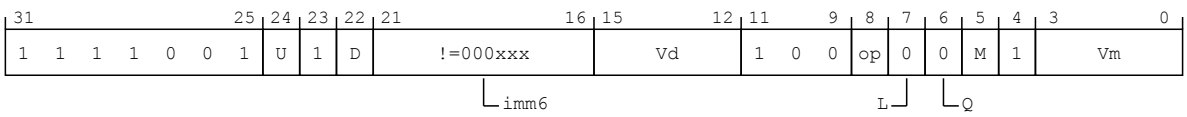

Encoding for the Signed result variant

Applies when `(op == 1)`

```
VQSHRN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Encoding for the Unsigned result variant

Applies when `(U == 1 && op == 0)`

```
VQSHRUN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15	13	12	11	7	6	5	0	15	12	11	9	8	7	6	5	4	3	0
1 1 1		U	1 1 1 1 1		D	!=000xxx		Vd		1 0 0		op	0	0	M	1	Vm	
						imm6							L	Q				

Encoding for the Signed result variant

Applies when `(op == 1)`

```
VQSHRN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Encoding for the Unsigned result variant

Applies when `(U == 1 && op == 0)`

```
VQSHRUN{<c>}{<q>}.<type><size>  <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Signed result” and “A1 Unsigned result” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Signed result” and “T1 Unsigned result” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

For the “A1 Signed result” and “T1 Signed result” variants: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1	U

For the “A1 Unsigned result” and “T1 Unsigned result” variants: is S when “U” is 1, and is encoded in “U”.

<size>

Is the data size for the elements of the vectors, encoded in “imm6”:

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to `<size>`/2, encoded in the “imm6” field as `<size>`/2 - `<imm>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e*:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let (result, sat) : (bits(esize), boolean) = SatQ{esize}(element >> shift_amount,
                                                                 dest_unsigned);
        D(d)[e*:esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.189 VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

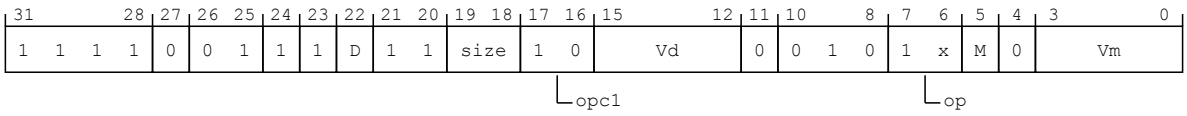

Encoding for the Signed result variant

```
VQSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

T1

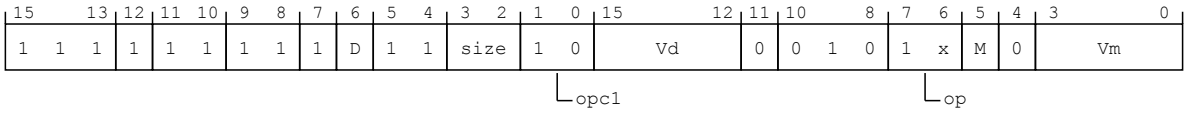

Encoding for the Signed result variant

```
VQSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Assembler Symbols

<c>

For the “A1 Signed result” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Signed result” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt>

Is the data type for the elements of the operand, encoded in “op:size”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.190 VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

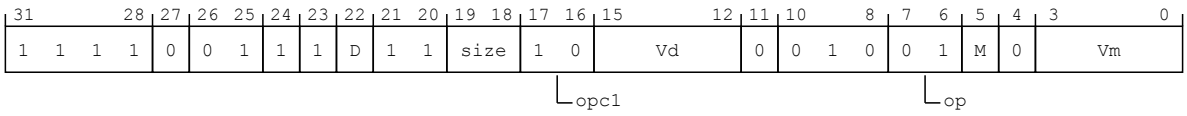

Encoding for the Unsigned result variant

```
VQSHRUN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

T1

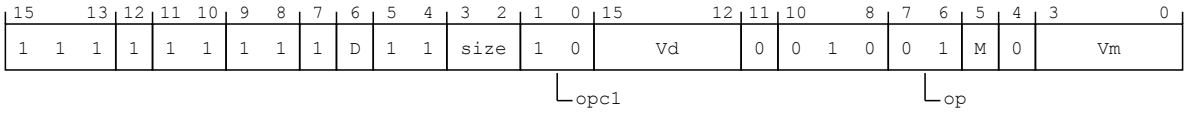

Encoding for the Unsigned result variant

```
VQSHRUN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Assembler Symbols

<c>

For the “A1 Unsigned result” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Unsigned result” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.191 VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

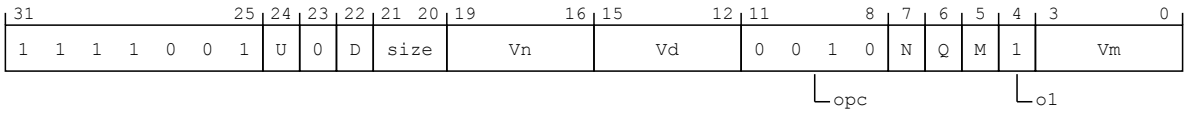

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VQSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

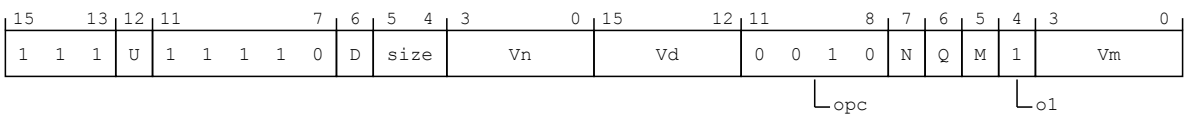

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VQSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VQSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let diff : integer = element1 - element2;
            let (res, sat) : (bits(esize), boolean) = SatQ{esize}(diff, unsigned);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.192 VRADDHN

Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see VADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

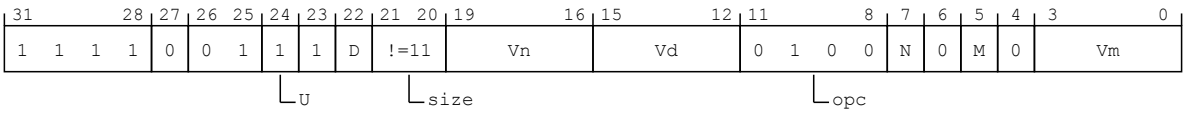

Encoding

```
VRADDHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

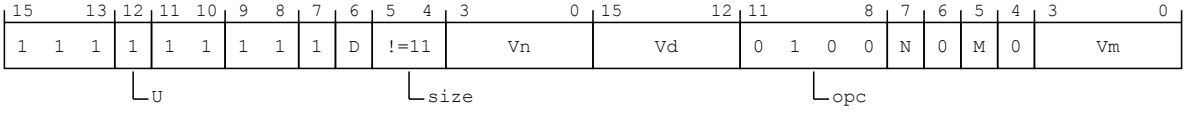

Encoding

```
VRADDHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = (RShr(UInt(Qin(n>>1)[e*:(2*esize)] +
                                    Qin(m>>1)[e*:(2*esize)]), esize, round));
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.193 VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see Floating-point reciprocal square root estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 1	Vd	0 1 0	F	0	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRECPE{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRECPE{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    Undefined();
end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 1	Vd	0 1 0	F	0	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRECPE{<c>}{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRECPE{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    Undefined();
end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see Floating-point reciprocal estimate and step.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if floating_point then
                D(d+r)[e*:esize] = FPRecipEstimate{esize}(D(m+r)[e*:esize], fpcr);
            else
                D(d+r)[e*:esize] = UnsignedRecipEstimate{esize}(D(m+r)[e*:esize]);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.194 VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

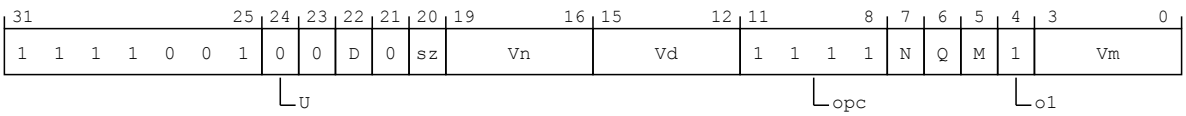

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRECPS{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRECPS{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

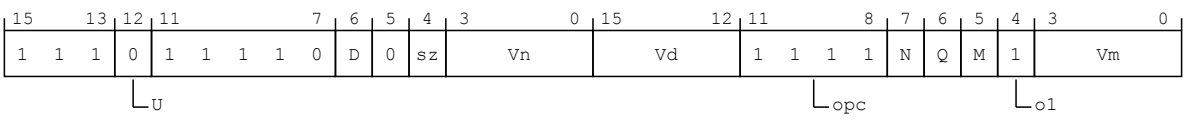

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRECPS{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRECPS{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see Floating-point reciprocal estimate and step.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = FPRecipStep{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize]);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.195 VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

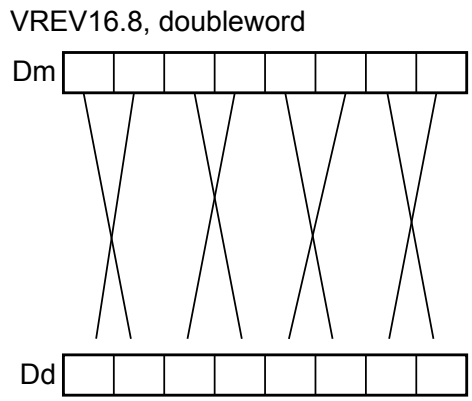

Figure F6-6 VREV16 doubleword operation

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–9	8–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0 0 0	1 0 (op)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VREV16{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VREV16{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			9	8	7	6	5	4	3				0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0		1	0	Q	M	0	Vm				
														opc1										op									

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VREV16{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VREV16{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*:esize] = D(m+r)[element*:esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.196 VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

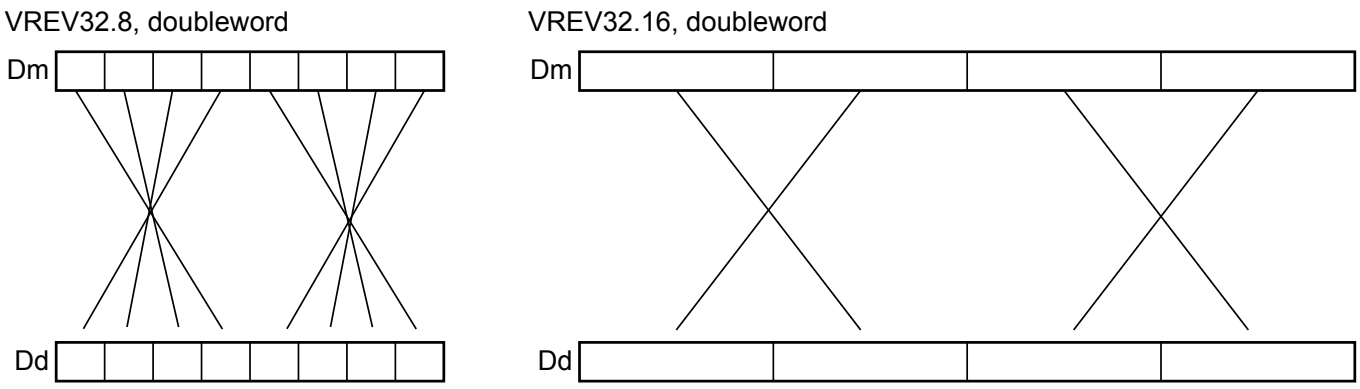

Figure F6-7 VREV32 doubleword operations

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–9	8–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0 0 0	0 1 (op)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VREV32{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VREV32{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			9	8	7	6	5	4	3				0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0		0	1	Q	M	0	Vm				
														opc1										op									

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VREV32{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VREV32{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*:esize] = D(m+r)[element*:esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.197 VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

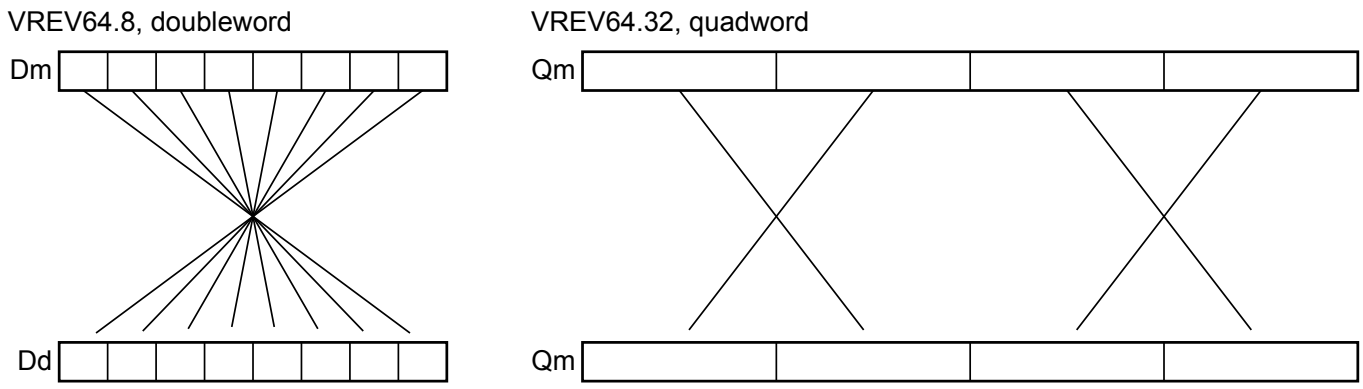

Figure F6-8 VREV64 doubleword operations

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–9	8–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	0 0 (opc1)	Vd	0 0 0	0 0 (op)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VREV64{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VREV64{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11			9	8	7	6	5	4	3				0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0		0	0	Q	M	0	Vm				
														opc1										op									

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VREV64{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VREV64{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*:esize] = D(m+r)[element*:esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.198 VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see VHADD.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

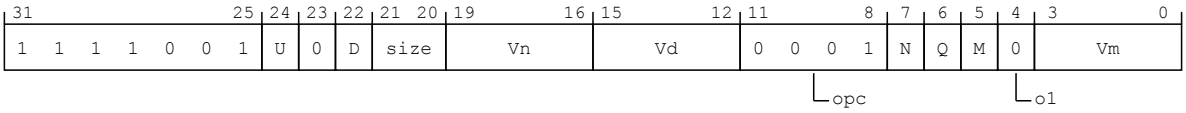

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRHADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRHADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

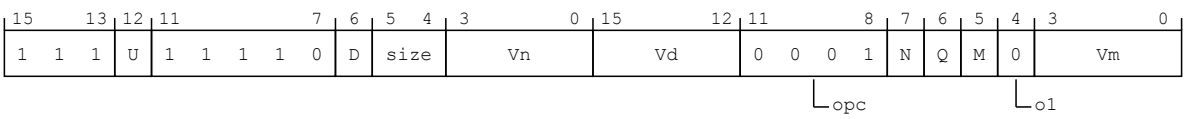

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VRHADD{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRHADD{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (element1 + element2 + 1) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.199 VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0 1	0 1 0 (op)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTA{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTA{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0 1	0 1 0 (op)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTA{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTA{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : bits(esize) = D(m+r)[e*:esize];
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
        D(d+r)[e*:esize] = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.200 VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

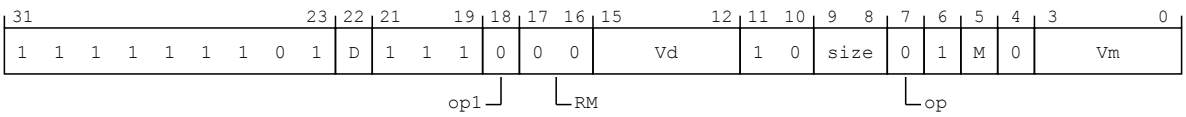

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTA{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTA{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTA{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

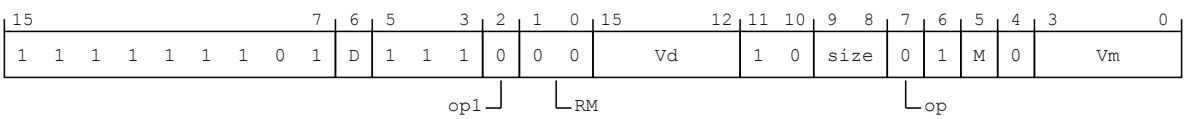

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTA{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTA{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTA{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.201 VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

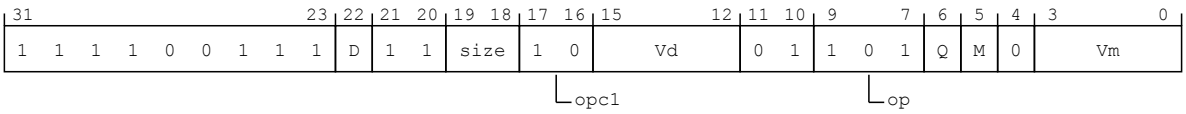

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTM{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTM{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

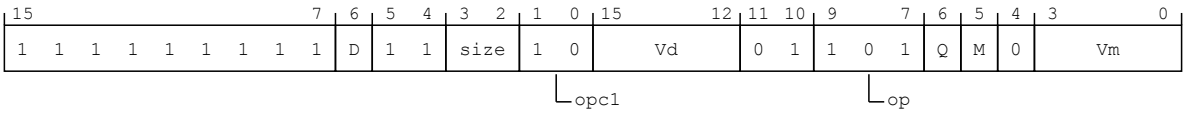

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTM{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTM{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : bits(esize) = D(m+r)[e*:esize];
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
        D(d+r)[e*:esize] = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.202 VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

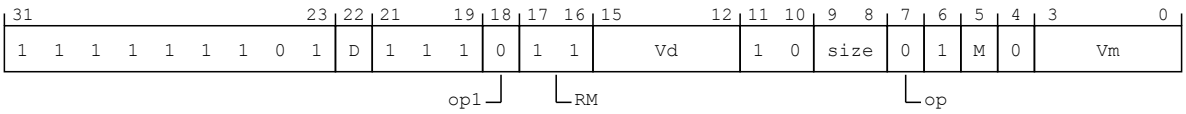

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTM{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTM{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTM{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

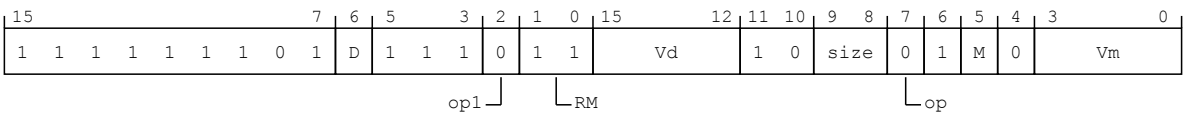

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTM{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTM{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTM{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.203 VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

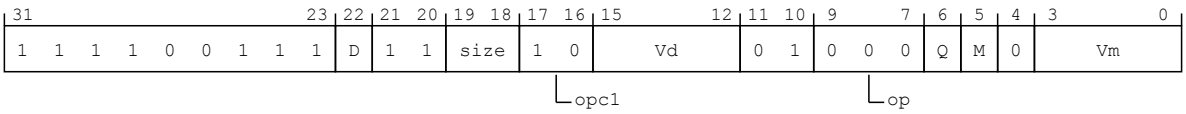

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTN{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTN{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

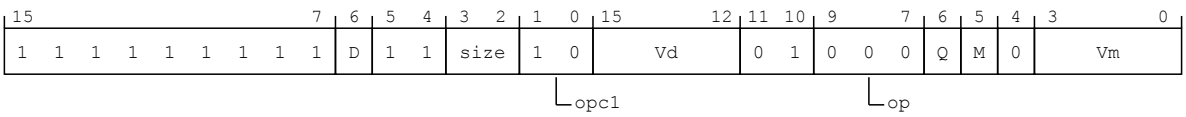

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTN{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTN{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : bits(esize) = D(m+r)[e*:esize];
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
        D(d+r)[e*:esize] = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.204 VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

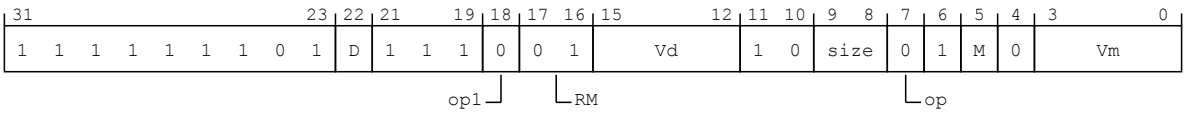

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTN{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTN{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTN{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

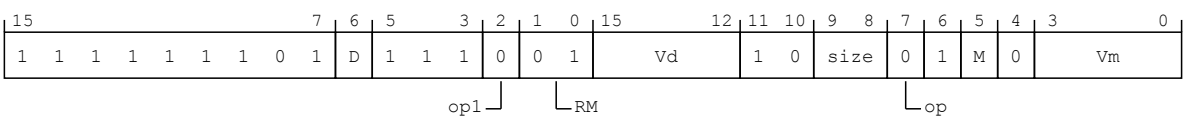

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTN{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTN{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTN{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.205 VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–10	9–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 0	Vd	0 1	1 1 1	Q	M	0	Vm
				opc1			op				

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTP{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTP{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–10	9–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 0	Vd	0 1	1 1 1	Q	M	0	Vm
				opc1			op				

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTP{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTP{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : bits(esize) = D(m+r)[e*:esize];
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
        D(d+r)[e*:esize] = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.206 VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

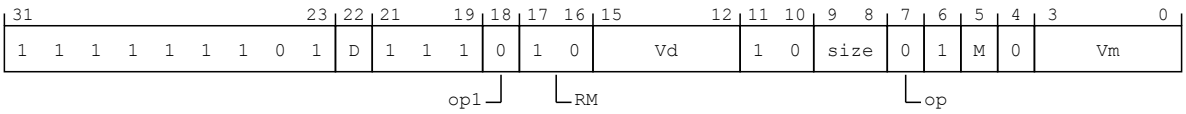

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTP{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTP{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTP{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

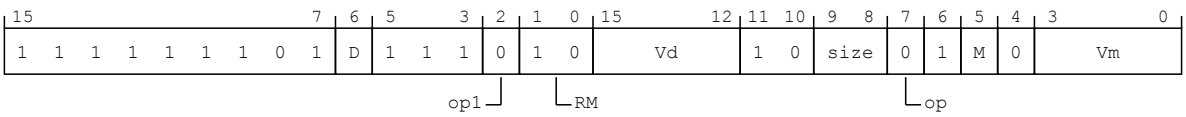

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTP{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTP{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTP{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 16 =>
        H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.207 VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

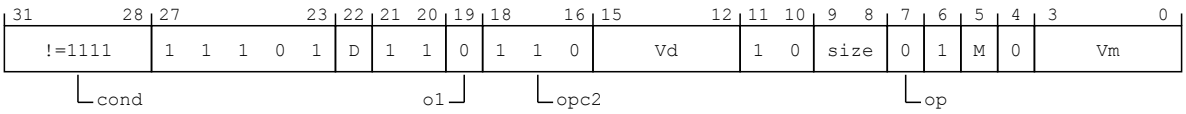

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTR{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTR{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTR{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

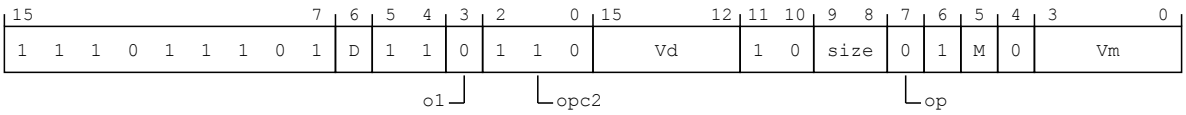

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTR{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTR{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTR{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 16 =>
            H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.208 VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0	1 0 0 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTX{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTX{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding_TIEEVEN;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0	1 0 0 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTX{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTX{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding_TIEEVEN;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : bits(esize) = D(m+r)[e*:esize];
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
        D(d+r)[e*:esize] = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.209 VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

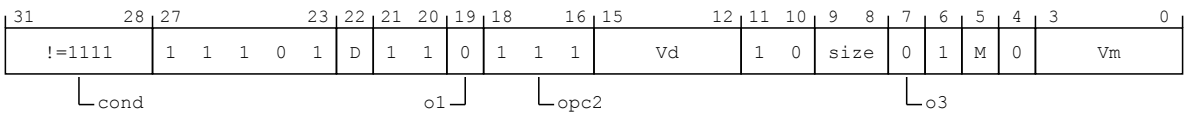

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTX{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTX{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTX{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

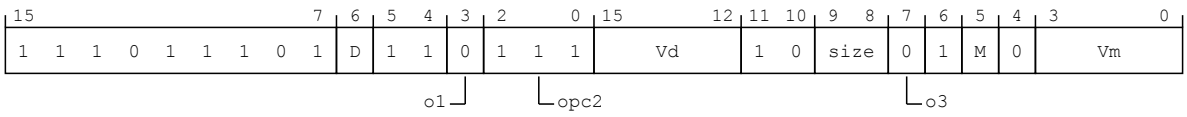

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTX{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTX{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTX{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = FPRoundingMode(fpcr);
    case esize of
        when 16 =>
            H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.210 VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

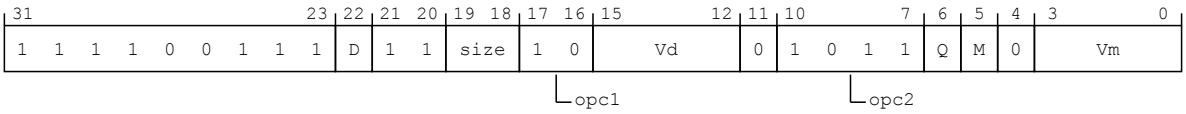

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTZ{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTZ{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding_ZERO;
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

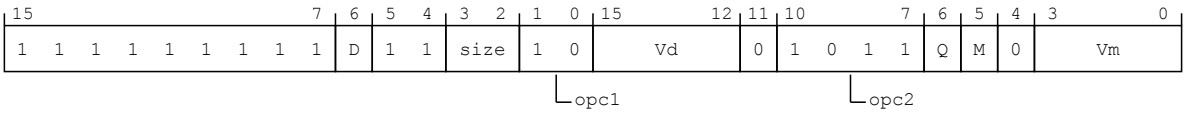

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRINTZ{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRINTZ{<q>}.<dt>  <Qd>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding_ZERO;
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : bits(esize) = D(m+r)[e*:esize];
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
        D(d+r)[e*:esize] = result;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.211 VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

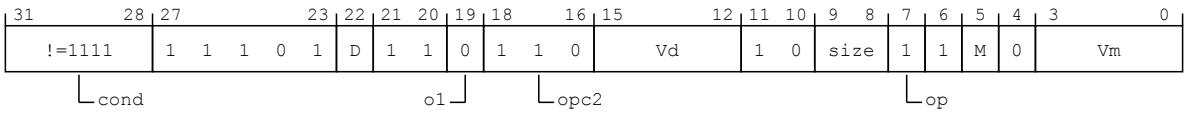

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTZ{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTZ{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTZ{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

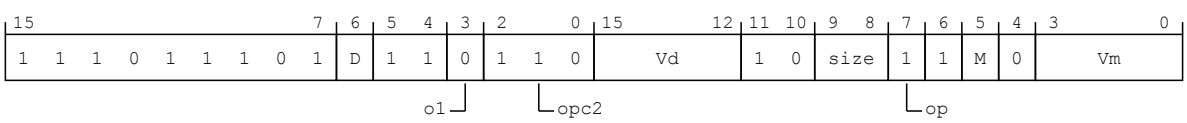

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VRINTZ{<c>}{<q>}.F16  <Sd>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VRINTZ{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VRINTZ{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 16 =>
            H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.212 VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see `VSHL`.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	0	D	size	Vn	Vd	0 1 0 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 0	D	size	Vn	Vd	0 1 0 1 (opc)	N	Q	M	0 (o1)	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VRSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : integer;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            if shift >= 0 then  // left shift
                result = element << shift;
            else                // rounding right shift
                shift = -shift;
                result = (element + (1 << (shift - 1))) >> shift;
            end;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.213 VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

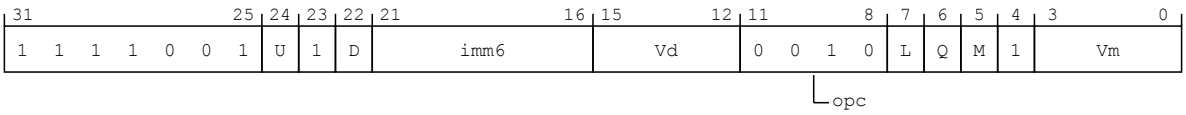

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VRSHR{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VRSHR{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

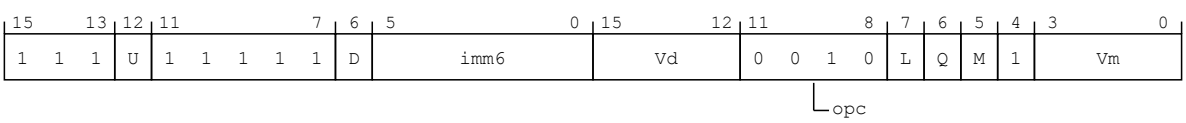

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VRSHR{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VRSHR{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to `<size>`, encoded in the “imm6” field as `<size>` - `<imm>`.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = RShr(element, shift_amount, round);
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.214 VRSHR (zero)

Vector Rounding Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

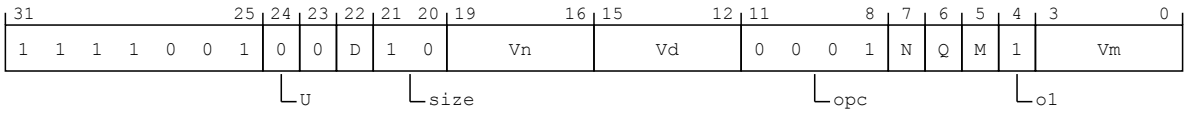

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSHR{<c>}{<q>}.<dt>  <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dm>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSHR{<c>}{<q>}.<dt>  <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qm>, <Qm>
```

T1

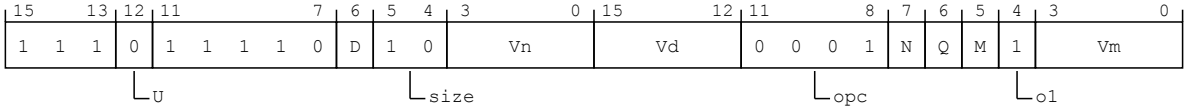

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSHR{<c>}{<q>}.<dt>  <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dm>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSHR{<c>}{<q>}.<dt>  <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qm>, <Qm>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

Operation

The description of VORR (register) gives the operational pseudocode for this instruction.

Operational Information

The description of VORR (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.215 VRSHRN

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

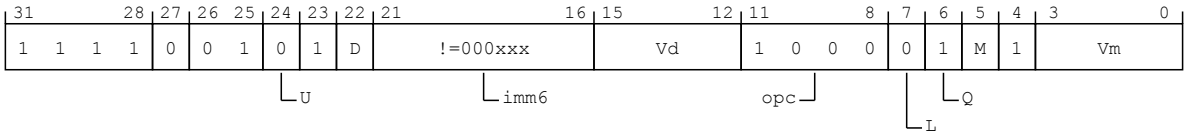

Encoding

```
VRSHRN{<c>}{<q>}.I<size>  <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

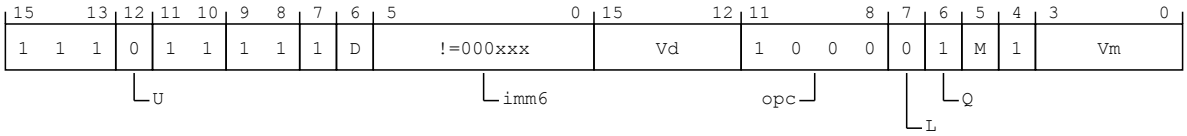

Encoding

```
VRSHRN{<c>}{<q>}.I<size>  <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size for the elements of the vectors, encoded in "imm6":

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to `<size>`/2, encoded in the "imm6" field as `<size>`/2 - `<imm>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = RShr(UInt(Qin(m>>1)[e*:(2*esize)]), shift_amount, round);
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.216 VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector.

This is a pseudo-instruction of VMOVN. This means:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

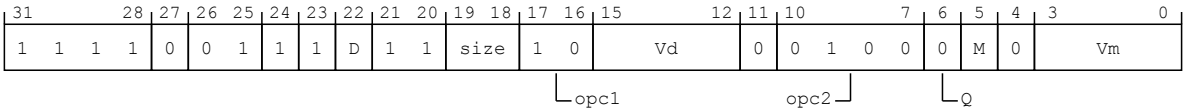

Encoding for the A1 variant

```
VRSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

T1

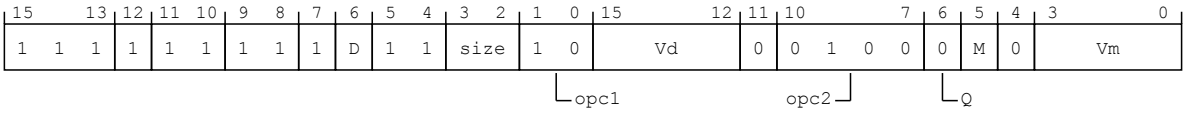

Encoding for the T1 variant

```
VRSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VMOVN gives the operational pseudocode for this instruction.

Operational Information

The description of VMOVN gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.217 VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 1	Vd	0 1 0	F	1	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSQRTE{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSQRTE{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    Undefined();
end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–7	6	5–4	3–2	1–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 1	Vd	0 1 0	F	1	Q	M	0	Vm
				opc1								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSQRTE{<c>}{<q>}.<dt>  <Dd>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSQRTE{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see Floating-point reciprocal estimate and step.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if floating_point then
                D(d+r)[e*:esize] = FPRSqrtEstimate{esize}(D(m+r)[e*:esize], fpcr);
            else
                D(d+r)[e*:esize] = UnsignedRSqrtEstimate{esize}(D(m+r)[e*:esize]);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.218 VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

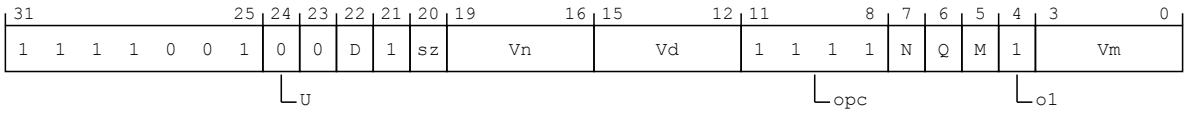

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSQRTS{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSQRTS{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

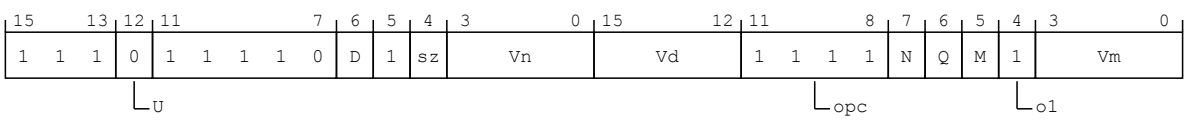

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VRSQRTS{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VRSQRTS{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see Floating-point reciprocal estimate and step.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = FPRSqrtStep{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize]);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.219 VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector. For truncated results, see VSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

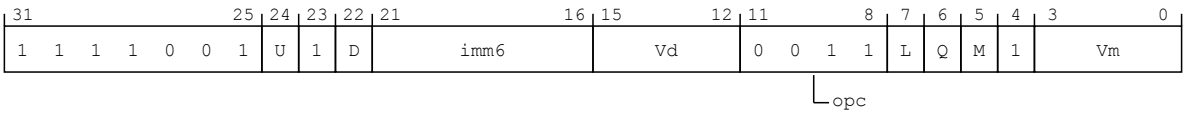

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VRSRA{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VRSRA{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

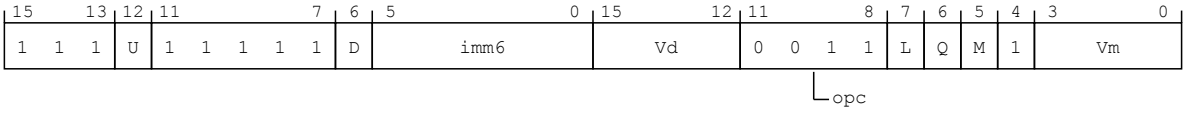

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VRSRA{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VRSRA{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to `<size>`, encoded in the “imm6” field as `<size>` - `<imm>`.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = RShr(element, shift_amount, round);
            D(d+r)[e*:esize] = D(d+r)[e*:esize] + result;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.220 VRSUBHN

Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see VSUBHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

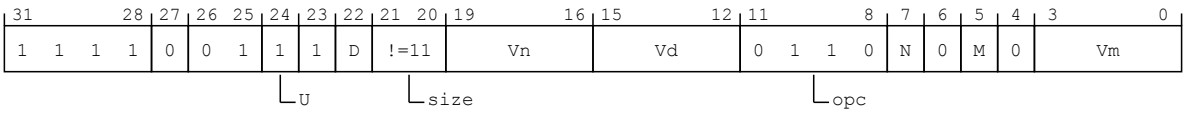

Encoding

```
VRSUBHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

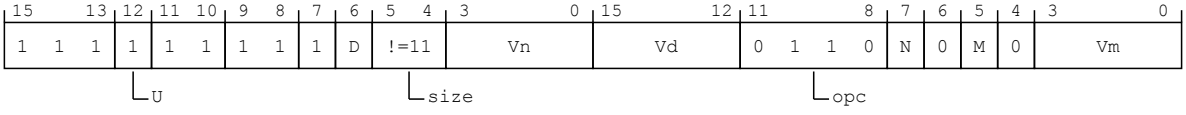

Encoding

```
VRSUBHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "size":

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = (RShr(UInt(Qin(n>>1)[e*:(2*esize)] -
                                        Qin(m>>1)[e*:(2*esize)]), esize, round));
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.221 VSDOT (vector)

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_DotProd)

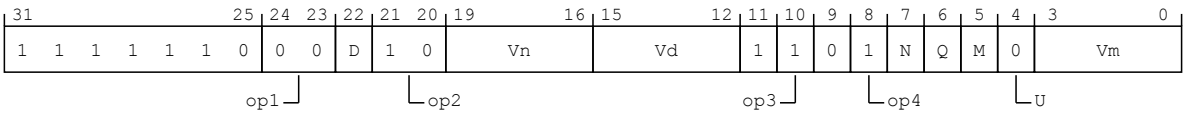

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSDOT{<q>}.S8  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)

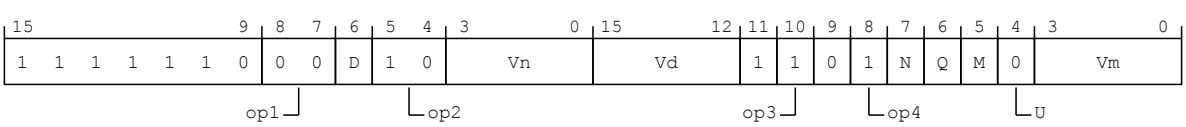

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSDOT{<q>}.S8  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    operand2 = D(m+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if signed then
                element1 = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*:esize] = result[e*:esize] + res;
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.222 VSDOT (by element)

Dot Product index form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_DotProd)

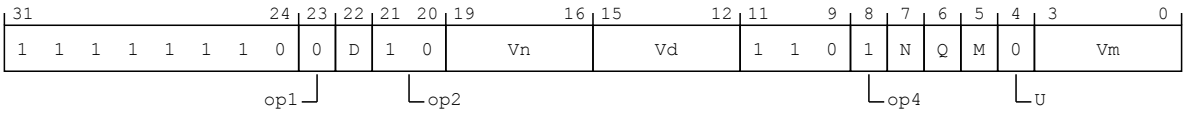

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSDOT{<q>}.S8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)

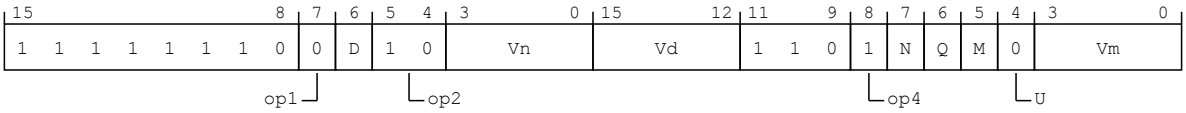

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSDOT{<q>}.S8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm” field.

<index>

Is the element index in the range 0 to 1, encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
var operand1 : bits(64);
let operand2 : bits(64) = D(m);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if signed then
                element1 = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = SInt(operand2[(4 * index + i)*:(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = UInt(operand2[(4 * index + i)*:(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*:esize] = result[e*:esize] + res;
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.223 VSELEQ, VSELGE, VSELGT, VSELVS

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the APSR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31								23	22	21	20	19			16	15			12	11	10	9		8	7	6	5	4	3		0
1	1	1	1	1	1	1	0	0	D	cc		Vn				Vd				1	0	size			N	0	M	0	Vm		

Encoding for the Equal, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 00 && size == 01)`

```
VSELEQ.F16  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Equal, single-precision scalar variant

Applies when `(cc == 00 && size == 10)`

```
VSELEQ.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Equal, double-precision scalar variant

Applies when `(cc == 00 && size == 11)`

```
VSELEQ.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Encoding for the Greater than or Equal, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 10 && size == 01)`

```
VSELGE.F16  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Greater than or Equal, single-precision scalar variant

Applies when `(cc == 10 && size == 10)`

```
VSELGE.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Greater than or Equal, double-precision scalar variant

Applies when `(cc == 10 && size == 11)`

```
VSELGE.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Encoding for the Greater than, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 11 && size == 01)`

```
VSELGT.F16  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Greater than, single-precision scalar variant

Applies when `(cc == 11 && size == 10)`

```
VSELGT.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Greater than, double-precision scalar variant

Applies when `(cc == 11 && size == 11)`

```
VSELGT.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Encoding for the Unordered, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 01 && size == 01)`

```
VSELVS.F16  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Unordered, single-precision scalar variant

Applies when `(cc == 01 && size == 10)`

```
VSELVS.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Unordered, double-precision scalar variant

Applies when `(cc == 01 && size == 11)`

```
VSELVS.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let condition : bits(4) = cc::(cc[1] XOR cc[0])::'0';
```

T1

15 … 7	6	5 4	3 … 0	15 … 12	11 10	9 8	7	6	5	4	3 … 0
1 1 1 1 1 1 1 0 0	D	cc	Vn	Vd	1 0	size	N	0	M	0	Vm

Encoding for the Equal, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 00 && size == 01)`

```
VSELEQ.F16  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Equal, single-precision scalar variant

Applies when `(cc == 00 && size == 10)`

```
VSELEQ.F32  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Equal, double-precision scalar variant

Applies when `(cc == 00 && size == 11)`

```
VSELEQ.F64  <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Greater than or Equal, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 10 && size == 01)`

```
VSELGE.F16  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Greater than or Equal, single-precision scalar variant

Applies when `(cc == 10 && size == 10)`

```
VSELGE.F32  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Greater than or Equal, double-precision scalar variant

Applies when `(cc == 10 && size == 11)`

```
VSELGE.F64  <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Encoding for the Greater than, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 11 && size == 01)`

```
VSELGT.F16  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Greater than, single-precision scalar variant

Applies when `(cc == 11 && size == 10)`

```
VSELGT.F32  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Greater than, double-precision scalar variant

Applies when `(cc == 11 && size == 11)`

```
VSELGT.F64  <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Encoding for the Unordered, half-precision scalar variant

(FEAT_FP16) Applies when `(cc == 01 && size == 01)`

```
VSELVS.F16  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Unordered, single-precision scalar variant

Applies when `(cc == 01 && size == 10)`

```
VSELVS.F32  <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Unordered, double-precision scalar variant

Applies when `(cc == 01 && size == 11)`

```
VSELVS.F64  <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let condition : bits(4) = cc::(cc[1] XOR cc[0])::'0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
case esize of
    when 16 =>
        H(d) = if ConditionHolds(condition) then H(n) else H(m);
    when 32 =>
        S(d) = if ConditionHolds(condition) then S(n) else S(m);
    when 64 =>
        D(d) = if ConditionHolds(condition) then D(n) else D(m);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.224 VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

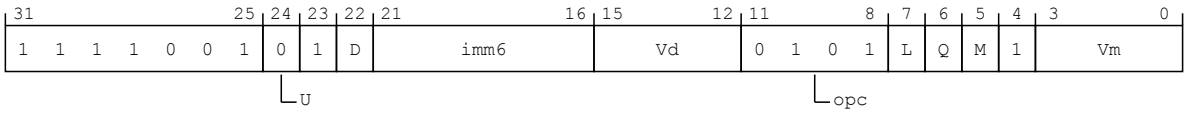

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSHL{<c>}{<q>}.I<size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSHL{<c>}{<q>}.I<size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

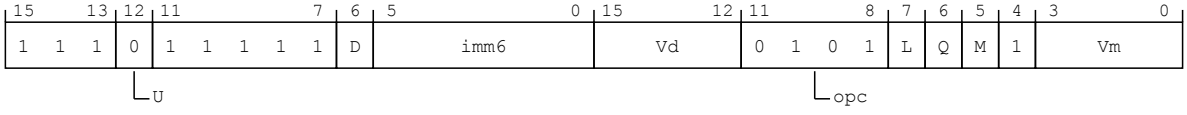

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSHL{<c>}{<q>}.I<size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSHL{<c>}{<q>}.I<size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 0 to `<size>`-1, encoded in the “imm6” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = LSL{esize}(D(m+r)[e*:esize], shift_amount);
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.225 VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see VRSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

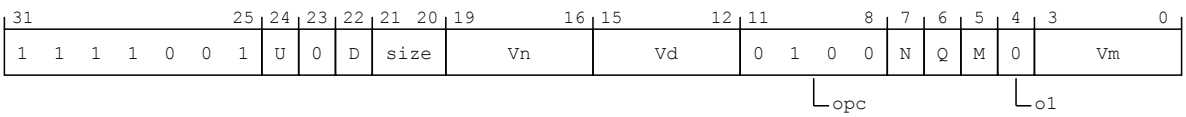

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

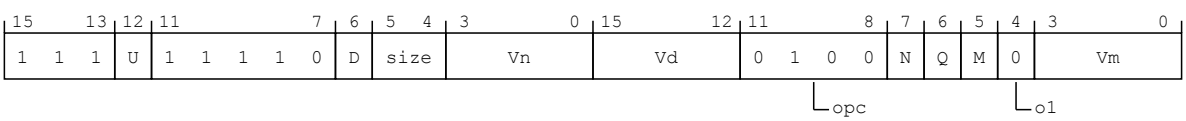

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSHL{<c>}{<q>}.<dt>  {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSHL{<c>}{<q>}.<dt>  {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var result : integer;
            if shift >= 0 then
                result = element << shift;
            else
                result = element >> -shift;
            end;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.226 VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

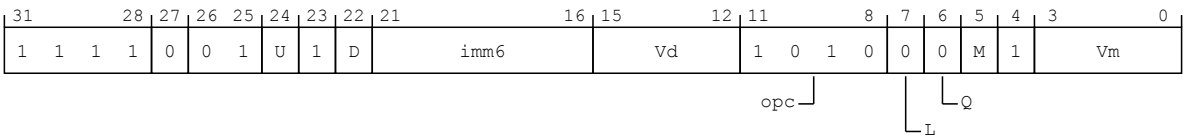

Encoding

Applies when `(imm6 IN {100001, 10001x, 1001xx, 101xxx, 11xxxx, 010001, 01001x, 0101xx, 011xxx, 001001, 00101x, 0011xx})`

```
VSHLL{<c>}{<q>}.<type><size>  <Qd>, <Dm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if imm6 IN {'001000', '010000', '100000'} then See("VMOVL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(imm6) - esize;
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

A2

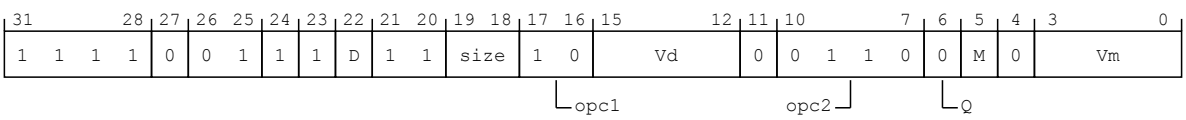

Encoding

```
VSHLL{<c>}{<q>}.<type><size>  <Qd>, <Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' || Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let shift_amount : integer = esize;
let unsigned : boolean = FALSE;  // Or TRUE without change of functionality
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15–13	12	11–10	9–8	7	6	5–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	imm6	Vd	1 0 1 0 (opc)	0 (L)	0 (Q)	M	1	Vm

Encoding

Applies when `(imm6 IN {100001, 10001x, 1001xx, 101xxx, 11xxxx, 010001, 01001x, 0101xx, 011xxx, 001001, 00101x, 0011xx})`

```
VSHLL{<c>}{<q>}.<type><size>  <Qd>, <Dm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if imm6 IN {'001000', '010000', '100000'} then See("VMOVL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(imm6) - esize;
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T2

15–13	12	11–10	9–8	7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1	1	1 1	1 1	1	D	1 1	size	1 0 (opc1)	Vd	0	0 1 1 0 (opc2)	0 (Q)	M	0	Vm

Encoding

```
VSHLL{<c>}{<q>}.<type><size>  <Qd>, <Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if size == '11' || Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let shift_amount : integer = esize;
let unsigned : boolean = FALSE;  // Or TRUE without change of functionality
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" and "A2" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" and "T2" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<size>	Encoding T1/A1	Encoding T2/A2
8	Encoded as imm6<5:3> = `0b001`	Encoded as size = `0b00`
16	Encoded as imm6<5:4> = `0b01`	Encoded as size = `0b01`
32	Encoded as imm6<5> = 1	Encoded as size = `0b10`

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

The immediate value. `<imm>` must lie in the range 1 to `<size>`, and:

- If `<size>` == `<imm>`, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
  - If `<size>` == 8, `<imm>` is encoded in imm6<2:0>.
  - If `<size>` == 16, `<imm>` is encoded in imm6<3:0>.
  - If `<size>` == 32, `<imm>` is encoded in imm6<4:0>.

<type>

The data type for the elements of the operand. It must be one of:

S Signed. In encoding T1/A1, encoded as U = 0.
U Unsigned. In encoding T1/A1, encoded as U = 1.
I Untyped integer, Available only in encoding T2/A2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(esize) = Din(m)[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        let result : integer = element << shift_amount;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.227 VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	1	D	imm6	Vd	0 0 0 0 (opc)	L	Q	M	1	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSHR{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSHR{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 1	D	imm6	Vd	0 0 0 0 (opc)	L	Q	M	1	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSHR{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSHR{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to `<size>`, encoded in the “imm6” field as `<size>` - `<imm>`.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = element >> shift_amount;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.228 VSHR (zero)

Vector Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

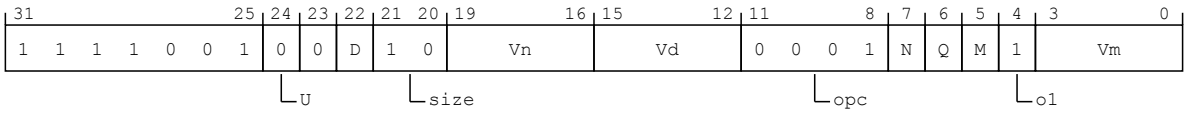

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSHR{<c>}{<q>}.<dt>  <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dm>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSHR{<c>}{<q>}.<dt>  <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qm>, <Qm>
```

T1

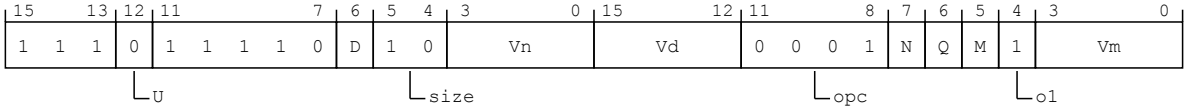

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSHR{<c>}{<q>}.<dt>  <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Dd>, }<Dm>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSHR{<c>}{<q>}.<dt>  <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>}  {<Qd>, }<Qm>, <Qm>
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

Operation

The description of VORR (register) gives the operational pseudocode for this instruction.

Operational Information

The description of VORR (register) gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.229 VSHRN

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

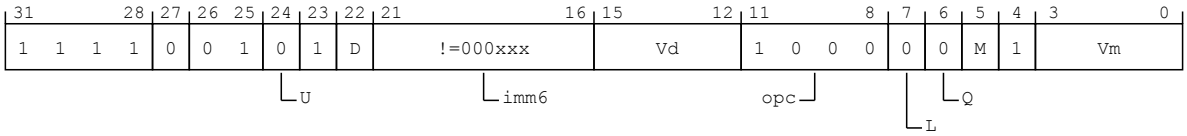

Encoding

```
VSHRN{<c>}{<q>}.I<size>  <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

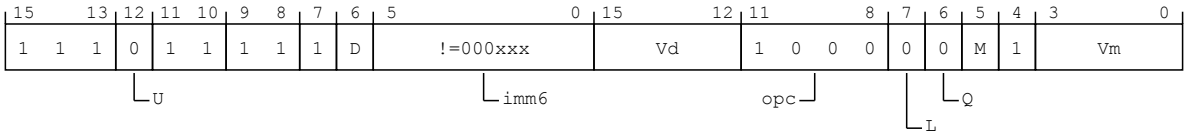

Encoding

```
VSHRN{<c>}{<q>}.I<size>  <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size for the elements of the vectors, encoded in "imm6":

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to `<size>`/2, encoded in the "imm6" field as `<size>`/2 - `<imm>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = LSR(Qin(m>>1)[e*:(2*esize)], shift_amount);
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.230 VSHRN (zero)

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector.

This is a pseudo-instruction of VMOVN. This means:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

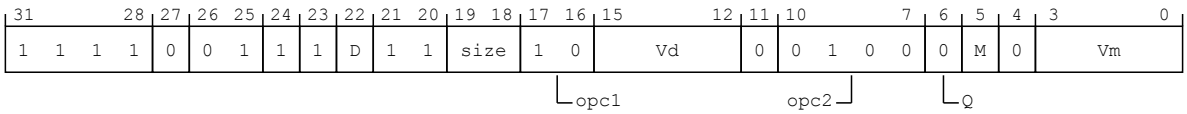

Encoding for the A1 variant

```
VSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

T1

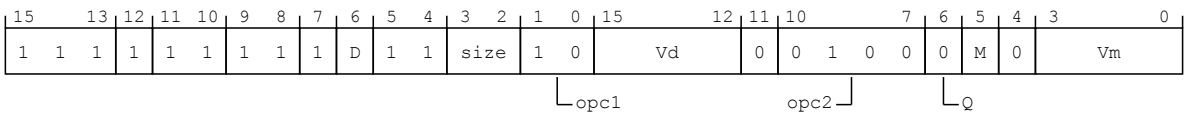

Encoding for the T1 variant

```
VSHRN{<c>}{<q>}.<dt>  <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt>  <Dd>, <Qm>
```

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of VMOVN gives the operational pseudocode for this instruction.

Operational Information

The description of VMOVN gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.231 VSLI

Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

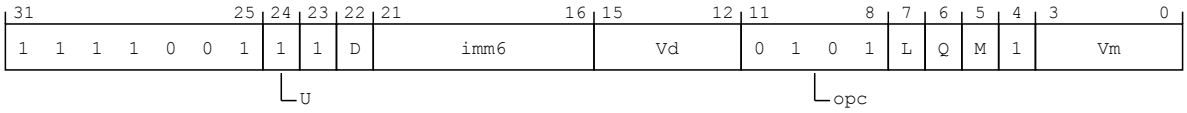

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSLI{<c>}{<q>}.<size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSLI{<c>}{<q>}.<size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

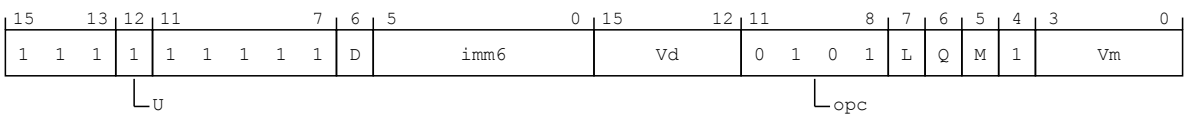

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSLI{<c>}{<q>}.<size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSLI{<c>}{<q>}.<size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 0 to `<size>`-1, encoded in the “imm6” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let mask : bits(esize) = LSL{}(Ones{esize}, shift_amount);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shifted_op : bits(esize) = LSL{}(D(m+r)[e*:esize], shift_amount);
            D(d+r)[e*:esize] = (D(d+r)[e*:esize] AND NOT(mask)) OR shifted_op;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.232 VSMMLA

Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of signed 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32I8MM)

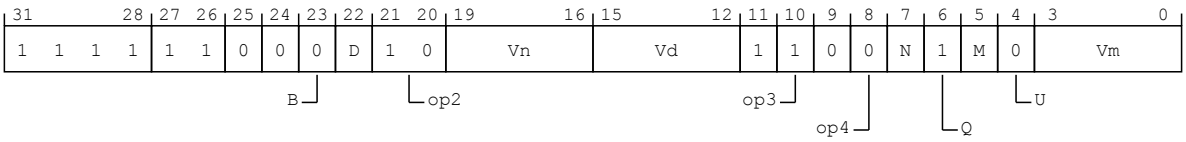

Encoding

```
VSMMLA{<q>}.S8  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
    when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AA32I8MM)

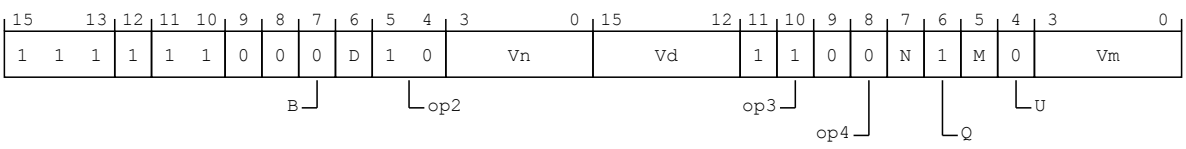

Encoding

```
VSMMLA{<q>}.S8  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
    when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let addend : bits(128)   = Q(d>>1);

Q(d>>1) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.233 VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

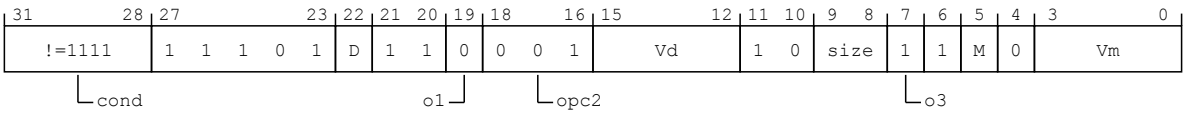

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VSQRT{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSQRT{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSQRT{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2		0	15			12	11	10	9		8	7	6	5	4	3				0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	size			1	1	M	0	Vm				
												o1			opc2										o3								

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VSQRT{<c>}{<q>}.F16  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSQRT{<c>}{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSQRT{<c>}{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 => H(d) = FPSqrt{16}(H(m), fpcr);
        when 32 => S(d) = FPSqrt{32}(S(m), fpcr);
        when 64 => D(d) = FPSqrt{64}(D(m), fpcr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.234 VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see VRSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–25	24	23	22	21–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	U	1	D	imm6	Vd	0 0 0 1 (opc)	L	Q	M	1	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSRA{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSRA{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15–13	12	11–7	6	5–0	15–12	11–8	7	6	5	4	3–0
1 1 1	U	1 1 1 1 1	D	imm6	Vd	0 0 0 1 (opc)	L	Q	M	1	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSRA{<c>}{<q>}.<type><size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSRA{<c>}{<q>}.<type><size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to `<size>`, encoded in the “imm6” field as `<size>` - `<imm>`.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = element >> shift_amount;
            D(d+r)[e*:esize] = D(d+r)[e*:esize] + result;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.235 VSRI

Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

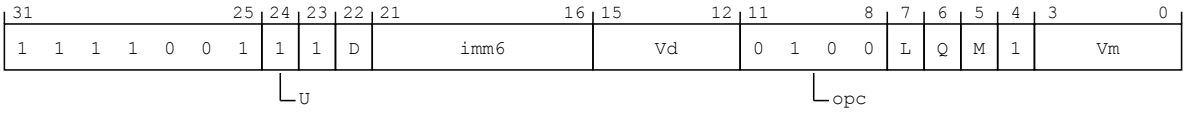

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSRI{<c>}{<q>}.<size>  {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSRI{<c>}{<q>}.<size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

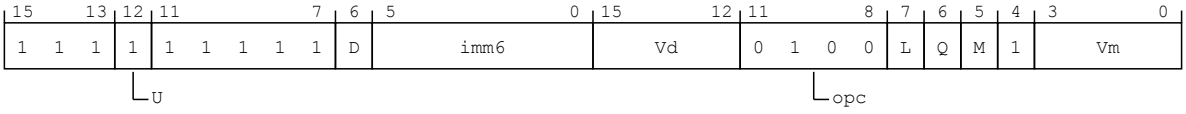

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0 && !(imm6[5:3] == 000 && L == 0))`

```
VSRI{<c>}{<q>}.<size>  {<Dd>, }<Dm>, #<imm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1 && !(imm6[5:3] == 000 && L == 0))`

```
VSRI{<c>}{<q>}.<size>  {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to `<size>`, encoded in the “imm6” field as `<size>` - `<imm>`.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let mask : bits(esize) = LSR(Ones{esize}, shift_amount);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shifted_op : bits(esize) = LSR(D(m+r)[e*:esize], shift_amount);
            D(d+r)[e*:esize] = (D(d+r)[e*:esize] AND NOT(mask)) OR shifted_op;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.236 VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

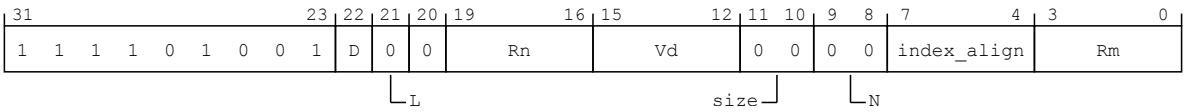

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A2

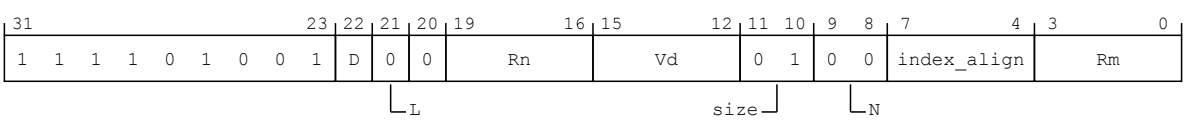

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A3

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	0	0	Rn	Vd	1 0	0 0	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T1

15								7	6	5	4	3	0	15	12	11	10	9	8	7	4	3	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn		Vd		0	0	0	0	index_align		Rm	
										L						size			N				

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	1	0	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				1	0	0	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be `{ <Dd>[<index>] }`.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` == **8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` == **16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` == **32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on `<size>`:

`<size>` == **8** Encoded in the “index_align<0>” field as 0.
`<size>` == **16** Encoded in the “index_align<1:0>” field as `0b00`.
`<size>` == **32** Encoded in the “index_align<2:0>” field as `0b000`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

`<size>` == **16** `<align>` is 16, meaning 16-bit alignment, encoded in the “index_align<1:0>” field as `0b01`.
`<size>` == **32** `<align>` is 32, meaning 32-bit alignment, encoded in the “index_align<2:0>” field as `0b011`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*ebytes}(address) = D(d)[index*:(8*ebytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.237 VST1 (multiple single elements)

Store multiple single elements from one, two, three, or four registers

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, A3, and A4) and T32 (T1, T2, T3, and T4).

A1

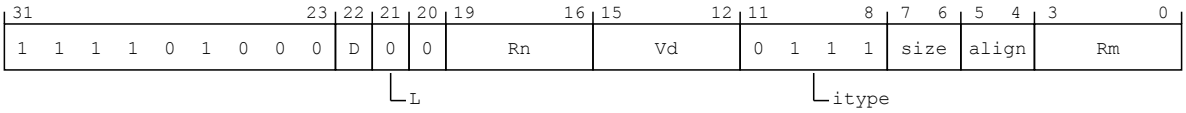

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn		Vd		1	0	1	0	size		align		Rm	
										L						itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31								23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn		Vd		0	1	1	0	size		align		Rm	
										L						itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A4

31								23	22	21	20	19			16	15			12	11			8	7	6	5	4	3			0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				0	0	1	0	size		align		Rm			
										L												itype									

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15							7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn	Vd		0	1	1	1	size		align	Rm
										L					itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T2

15							7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn	Vd		1	0	1	0	size		align	Rm
										L					itype							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15–7	6	5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 1 1 0 0 1 0	D	0	0	Rn	Vd	0 1 1 0	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T4

15								7	6	5	4	3			0	15			12	11			8	7		6	5		4	3			0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				0	0	1	0	size			align			Rm			
										L												itype										

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST1{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST1 (multiple single elements).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, “A3 Post-indexed”, “A4 Offset”, and “A4 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, “T3 Post-indexed”, “T4 Offset”, and “T4 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	64

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd> }` Single register. Selects the A1 and T1 encodings of the instruction.
`{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
`{ <Dd>, <Dd+1>, <Dd+2> }` Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register `<Dd>` is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as `0b01`.
128 128-bit alignment, encoded in the "align" field as `0b10`. Available only if `<list>` contains two or four registers.
256 256-bit alignment, encoded in the "align" field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about `<Rn>`, `!`, and `<Rm>`, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                        tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if ebytes != 8 then
                MemU{8*ebytes}(address) = D(d+r)[e*:(8*ebytes)];
            else
                if !IsAlignedSize(address, ebytes) && AlignmentEnforced() then
                    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
                    AArch32_Abort(fault);
                end;

                let data : bits(64) = D(d+r)[e*:64];
                if BigEndian(AccessType_ASIMD) then
                    MemU{32}(address) = data[63:32];
                    MemU{32}(address+4) = data[31:0];
                else
                    MemU{32}(address) = data[31:0];
                    MemU{32}(address+4) = data[63:32];
                end;
            end;

            address = address + ebytes;
        end;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            R(n) = R(n) + 8*regs;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.238 VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	0	0	Rn	Vd	0 0	0 1	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19			16	15			12	11	10	9	8	7			4	3			0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	1	0	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31								23	22	21	20	19	16	15	12	11	10	9	8	7	4	3	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn		Vd		1	0	0	1	index_align		Rm	
										L						size			N				

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	0	0	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	1	0	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				1	0	0	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST2 (single 2-element structure from one lane).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>] }` Single-spaced registers, encoded as “spacing” = 0.
`{ <Dd>[<index>], <Dd+2>[<index>] }` Double-spaced registers, encoded as “spacing” = 1. Not permitted when `<size>` == 8.

The encoding of “spacing” depends on `<size>`:

`<size>` == **16** “spacing” is encoded in the “index_align<1>” field.
`<size>` == **32** “spacing” is encoded in the “index_align<2>” field.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` == **8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` == **16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` == **32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on `<size>`:

`<size>` == **8** Encoded in the “index_align<0>” field as 0.
`<size>` == **16** Encoded in the “index_align<0>” field as 0.
`<size>` == **32** Encoded in the “index_align<1:0>” field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<size>` == **8** `<align>` is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.
`<size>` == **16** `<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
`<size>` == **32** `<align>` is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as `0b01`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*ebytes}(address       ) = D(d)[index*:(8*ebytes)];
    MemU{8*ebytes}(address+ebytes) = D(d2)[index*:(8*ebytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.239 VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see Element and structure load/store instructions. Every element of each register is saved. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

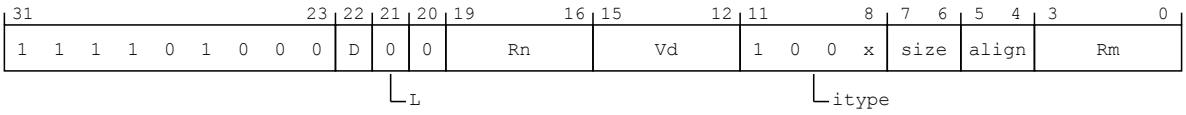

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19			16	15			12	11			8	7	6	5	4	3			0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				0	0	1	1	size		align		Rm			
										L												itype									

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 6	5 … 4	3 … 0
1 1 1 1 1 0 0 1 0	D	0	0	Rn	Vd	1 0 0 x	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 6	5 … 4	3 … 0
1 1 1 1 1 0 0 1 0	D	0	0	Rn	Vd	0 0 1 1	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST2{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST2 (multiple 2-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, and “A2 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, and “T2 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1> }` Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as `0b1000`.
`{ <Dd>, <Dd+2> }` Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as `0b1001`.
`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register `<Dd>` is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the “align” field as `0b00`.

Whenever `<align>` is present, the permitted values are:

64 64-bit alignment, encoded in the “align” field as `0b01`.
128 128-bit alignment, encoded in the “align” field as `0b10`.
256 256-bit alignment, encoded in the “align” field as `0b11`. Available only if `<list>` contains four registers.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to pairs-1 do
        for e = 0 to elements-1 do
            MemU{8*ebytes}(address       ) = D(d+r)[e*:(8*ebytes)];
            MemU{8*ebytes}(address+ebytes) = D(d2+r)[e*:(8*ebytes)];
            address = address + 2*ebytes;
        end;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 16*pairs;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.240 VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

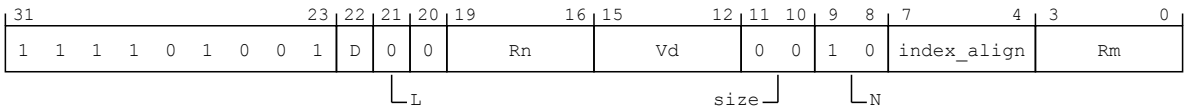

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19			16	15			12	11	10	9	8	7			4	3			0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	1	1	0	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31								23	22	21	20	19	16	15	12	11	10	9	8	7	4	3	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn		Vd		1	0	1	0	index_align		Rm	
										L						size			N				

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	0	1	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	1	1	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				1	0	1	0	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST3 (single 3-element structure from one lane).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }` Single-spaced registers, encoded as “spacing” = 0.
`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }` Double-spaced registers, encoded as “spacing” = 1. Not permitted when `<size>` == 8.

The encoding of “spacing” depends on `<size>`:

`<size>` **== 8** “spacing” is encoded in the “index_align<0>” field.
`<size>` **== 16** “spacing” is encoded in the “index_align<1>” field, and “index_align<0>” is set to 0.
`<size>` **== 32** “spacing” is encoded in the “index_align<2>” field, and “index_align<1:0>” is set to `0b00`.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` **== 8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` **== 16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` **== 32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Alignment

Standard alignment rules apply, see Alignment support.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    MemU{8*ebytes}(address        ) = D(d)[index*:(8*ebytes)];
    MemU{8*ebytes}(address+ebytes ) = D(d2)[index*:(8*ebytes)];
    MemU{8*ebytes}(address+2*ebytes) = D(d3)[index*:(8*ebytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.241 VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see Element and structure load/store instructions. Every element of each register is saved. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

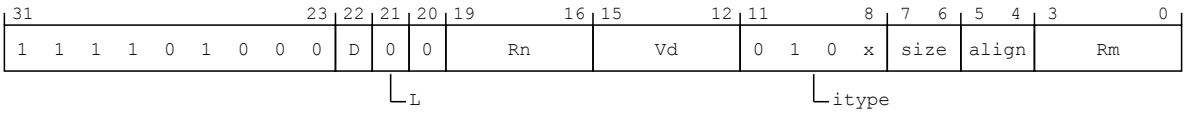

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || align[1] == '1' then Undefined(); end;
var inc : integer;
case itype of
    when '0100' =>
        inc = 1;
    when '0101' =>
        inc = 2;
    otherwise =>
        See("Related encodings");
end;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15–7	6	5	4	3–0	15–12	11–8	7–6	5–4	3–0
1 1 1 1 1 0 0 1 0	D	0	0	Rn	Vd	0 1 0 x	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST3{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || align[1] == '1' then Undefined(); end;
var inc : integer;
case itype of
    when '0100' =>
        inc = 1;
    when '0101' =>
        inc = 2;
    otherwise =>
        See("Related encodings");
end;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST3 (multiple 3-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2> }` Single-spaced registers, encoded in the "itype" field as `0b0100`.
`{ <Dd>, <Dd+2>, <Dd+4> }` Double-spaced registers, encoded in the "itype" field as `0b0101`.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as `0b01`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

: is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        MemU{8*ebytes}(address         ) = D(d)[e*:(8*ebytes)];
        MemU{8*ebytes}(address+ebytes  ) = D(d2)[e*:(8*ebytes)];
        MemU{8*ebytes}(address+2*ebytes) = D(d3)[e*:(8*ebytes)];
        address = address + 3*ebytes;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 24;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.242 VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1

31–23	22	21	20	19–16	15–12	11–10	9–8	7–4	3–0
1 1 1 1 0 1 0 0 1	D	0	0	Rn	Vd	0 0	1 1	index_align	Rm
		L				size	N		

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '00' then See("Related encodings"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2

31								23	22	21	20	19			16	15			12	11	10	9	8	7			4	3			0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	1	1	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '01' then See("Related encodings"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A3

31								23	22	21	20	19			16	15			12	11	10	9	8	7			4	3			0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				1	0	1	1	index_align				Rm			
										L										size			N								

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '10' then See("Related encodings"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	0	1	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '00' then See("Related encodings"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	1	1	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '01' then See("Related encodings"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T3

15								7	6	5	4	3			0	15			12	11	10	9	8	7			4	3			0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				1	0	1	1	index_align				Rm			
										L										size			N							

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '10' then See("Related encodings"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST4 (single 4-element structure from one lane).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

`{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }` Single-spaced registers, encoded as “spacing” = 0.

`{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }` Double-spaced registers, encoded as “spacing” = 1. Not permitted when `<size>` == 8.

The encoding of “spacing” depends on `<size>`:

`<size>` == **16** “spacing” is encoded in the “index_align<1>” field.
`<size>` == **32** “spacing” is encoded in the “index_align<2>” field.

The register `<Dd>` is encoded in the “D:Vd” field.

The permitted values and encoding of `<index>` depend on `<size>`:

`<size>` == **8** `<index>` is in the range 0 to 7, encoded in the “index_align<3:1>” field.
`<size>` == **16** `<index>` is in the range 0 to 3, encoded in the “index_align<3:2>” field.
`<size>` == **32** `<index>` is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on `<size>`:

`<size>` == **8** Encoded in the “index_align<0>” field as 0.
`<size>` == **16** Encoded in the “index_align<0>” field as 0.
`<size>` == **32** Encoded in the “index_align<1:0>” field as `0b00`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<size>` == **8** `<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
`<size>` == **16** `<align>` is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.
`<size>` == **32** `<align>` can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as `0b01`, and 128-bit alignment is encoded in the "index_align<1:0>" field as `0b10`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*ebytes}(address         ) = D(d)[index*:(8*ebytes)];
    MemU{8*ebytes}(address+ebytes  ) = D(d2)[index*:(8*ebytes)];
    MemU{8*ebytes}(address+2*ebytes) = D(d3)[index*:(8*ebytes)];
    MemU{8*ebytes}(address+3*ebytes) = D(d4)[index*:(8*ebytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*ebytes;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.243 VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see Element and structure load/store instructions. Every element of each register is saved. For details of the addressing mode, see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

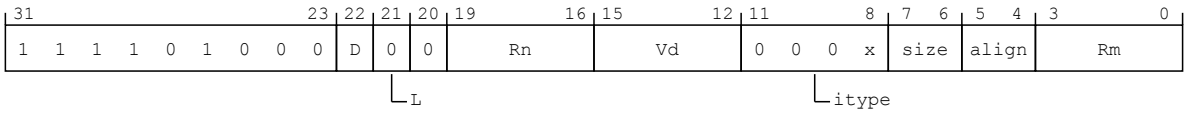

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if itype != '000x' then See("Related encodings"); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15 … 7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 6	5 … 4	3 … 0
1 1 1 1 1 0 0 1 0	D	0	0	Rn	Vd	0 0 0 x	size	align	Rm
		L				itype			

Encoding for the Offset variant

Applies when `(Rm == 1111)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when `(Rm == 1101)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when `(Rm != 11x1)`

```
VST4{<c>}{<q>}.<size>  <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if itype != '000x' then See("Related encodings"); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST4 (multiple 4-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Single-spaced registers, encoded in the "itype" field as `0b0000`.
`{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }` Double-spaced registers, encoded in the "itype" field as `0b0001`.

The register `<Dd>` is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as `0b00`.

Whenever `<align>` is present, the permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

64 64-bit alignment, encoded in the “align” field as `0b01`.
128 128-bit alignment, encoded in the “align” field as `0b10`.
256 256-bit alignment, encoded in the “align” field as `0b11`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see Advanced SIMD addressing mode.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean  = PSTATE.EL != EL0;
    let tagchecked : boolean  = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                     tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        MemU{8*ebytes}(address         ) = D(d)[e*:(8*ebytes)];
        MemU{8*ebytes}(address+ebytes  ) = D(d2)[e*:(8*ebytes)];
        MemU{8*ebytes}(address+2*ebytes) = D(d3)[e*:(8*ebytes)];
        MemU{8*ebytes}(address+3*ebytes) = D(d4)[e*:(8*ebytes)];
        address = address + 4*ebytes;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 32;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.244 VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

This instruction is used by the alias VPUSH.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

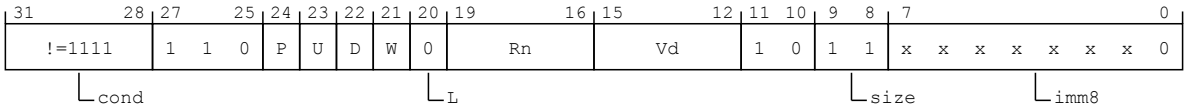

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VSTMDB{<c>}{<q>}{.<size>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VSTM{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31–28	27–25	24	23	22	21	20	19–16	15–12	11–10	9–8	7–0
!=1111	1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 0	imm8
cond						L				size	

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VSTMDB{<c>}{<q>}{.<size>}  <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VSTM{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 1	x x x x x x x 0
					L				size	imm8

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VSTMDB{<c>}{<q>}{.<size>}  <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VSTM{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2;  // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T2

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	P	U	D	W	0	Rn	Vd	1 0	1 0	imm8
					L				size	

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VSTMDB{<c>}{<q>}{.<size>}  <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VSTM{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>}  <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VSTM.

Related encodings: See Advanced SIMD and floating-point 64-bit move for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in “W”:

W	!
0	[absent]
1	[present]

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Alias Conditions

Alias	Is preferred when
VPUSH	P == '1' && U == '0' && W == '1' && Rn == '1101'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;
    for r = 0 to regs-1 do
        if single_regs then
            MemA{32}(address) = S(d+r);
            address = address+4;
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address) =  D(d+r)[63:32];
                MemA{32}(address+4) =  D(d+r)[31:0];
            else
                MemA{32}(address) = D(d+r)[31:0];
                MemA{32}(address+4) = D(d+r)[63:32];
            end;

            address = address+8;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.245 VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

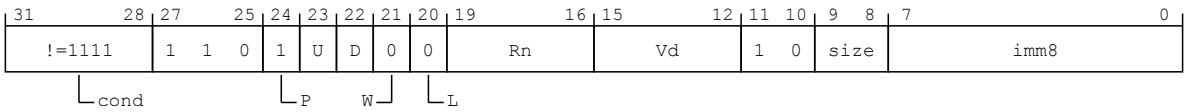

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VSTR{<c>}{<q>}.16  <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSTR{<c>}{<q>}{.32}  <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSTR{<c>}{<q>}{.64}  <Dd>, [<Rn>{, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–9	8	7	6	5	4	3–0	15–12	11–10	9–8	7–0
1 1 1 0 1 1 0	1	U	D	0	0	Rn	Vd	1 0	size	imm8
	P			W	L					

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VSTR{<c>}{<q>}.16  <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSTR{<c>}{<q>}{.32}  <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSTR{<c>}{<q>}{.64}  <Dd>, [<Rn>{, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

See Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Sd>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vd:D” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Half-precision scalar” and “T1 Half-precision scalar” variants: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the “imm8” field as <imm>/2.

For the “A1 Double-precision scalar”, “A1 Single-precision scalar”, “T1 Double-precision scalar”, and “T1 Single-precision scalar” variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP source register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let address : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    case esize of
        when 16 =>
            MemA{16}(address) = H(d);
        when 32 =>
            MemA{32}(address) = S(d);
        when 64 =>
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address)   = D(d)[63:32];
                MemA{32}(address+4) = D(d)[31:0];
            else
                MemA{32}(address)   = D(d)[31:0];
                MemA{32}(address+4) = D(d)[63:32];
            end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.246 VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

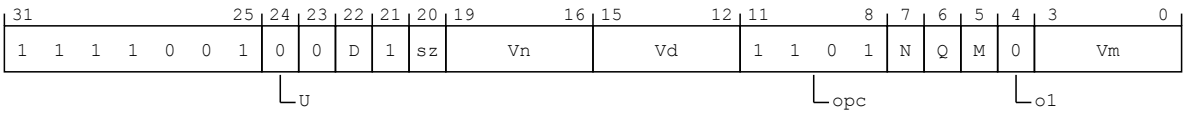

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

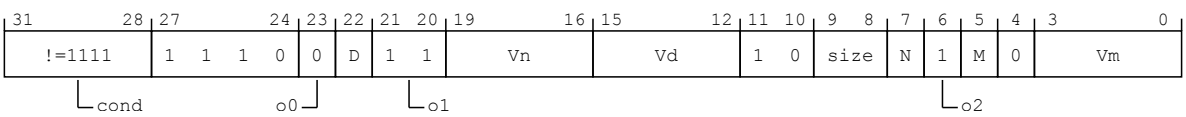

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VSUB{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSUB{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSUB{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15–13	12	11–7	6	5	4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	1	sz	Vn	Vd	1 1 0 1	N	Q	M	0	Vm
	U							opc				o1	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15							8	7	6	5	4	3			0	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	0	D	1	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
								o0			o1														o2						

Encoding for the Half-precision scalar variant

(FEAT_FP16) Applies when `(size == 01)`

```
VSUB{<c>}{<q>}.F16  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSUB{<c>}{<q>}.F32  {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSUB{<c>}{<q>}.F64  {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock()  then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the "A2 Double-precision scalar", "A2 Half-precision scalar", "A2 Single-precision scalar", "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPSub{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else             // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                H(d) = FPSub{16}(H(n), H(m), fpcr);
            when 32 =>
                S(d) = FPSub{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPSub{64}(D(n), D(m), fpcr);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.247 VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

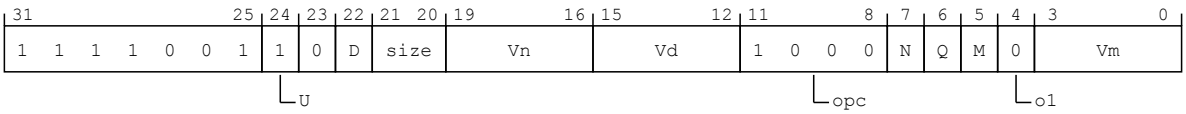

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

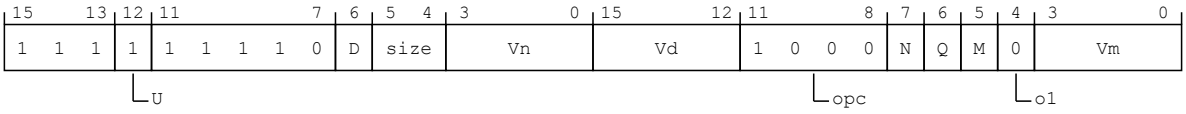

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSUB{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSUB{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = D(n+r)[e*:esize] - D(m+r)[e*:esize];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.248 VSUBHN

Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see VRSUBHN.

There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

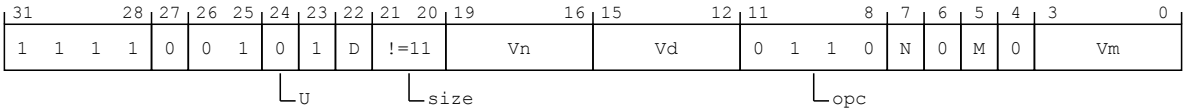

Encoding

```
VSUBHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

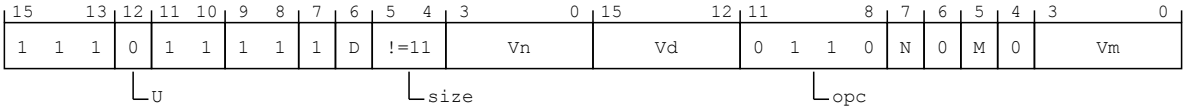

Encoding

```
VSUBHN{<c>}{<q>}.<dt>  <Dd>, <Qn>, <Qm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in "size":

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = Qin(n>>1)[e*:(2*esize)] - Qin(m>>1)[e*:(2*esize)];
        D(d)[e*:esize] = result[2*esize-1:esize];
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.249 VSUBL

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26	25	24	23	22	21–20	19–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1	0	0	1	U	1	D	!=11	Vn	Vd	0 0 1	0	N	0	M	0	Vm
							size				op					

Encoding

```
VSUBL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1

15–13	12	11	10	9	8	7	6	5–4	3–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1	U	1	1	1	1	1	D	!=11	Vn	Vd	0 0 1	0	N	0	M	0	Vm
								size				op					

Encoding

```
VSUBL{<c>}{<q>}.<dt>  <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the second operand vector, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vsubw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 - element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.250 VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–28	27	26–25	24	23	22	21–20	19–16	15–12	11–9	8	7	6	5	4	3–0
1 1 1 1	0	0 1	U	1	D	!=11	Vn	Vd	0 0 1	1	N	0	M	0	Vm
						size				op					

Encoding

```
VSUBW{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1

15–13	12	11–10	9–8	7	6	5–4	3–0	15–12	11–9	8	7	6	5	4	3–0
1 1 1	U	1 1	1 1	1	D	!=11	Vn	Vd	0 0 1	1	N	0	M	0	Vm
						size				op					

Encoding

```
VSUBW{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the "T1" variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the second operand vector, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vsubw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 - element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.251 VSUDOT (by element)

Dot Product index form with signed and unsigned integers (by element)

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32I8MM)

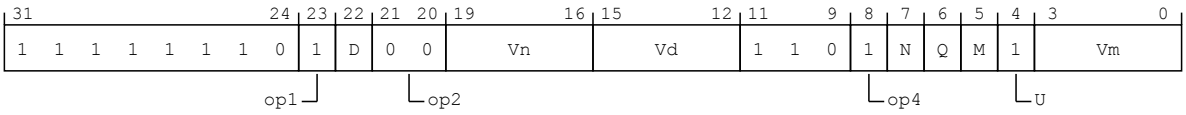

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSUDOT{<q>}.U8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSUDOT{<q>}.U8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32I8MM)

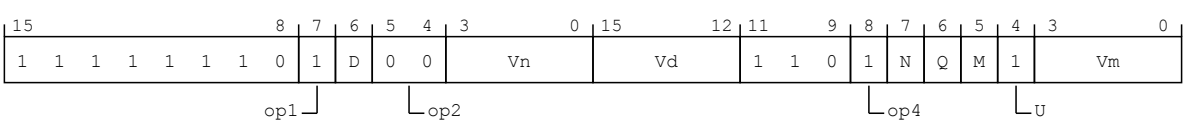

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VSUDOT{<q>}.U8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSUDOT{<q>}.U8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm” field.

<index>

Is the element index in the range 0 to 1, encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

operand2 = Din(m);
for r = 0 to regs-1 do
    operand1 = Din(n+r);
    result = Din(d+r);
    for e = 0 to 1 do
        var res : bits(32) = result[e*:32];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        for b = 0 to 3 do
            let op1elt : bits(8) = operand1[(4 * e + b)*:8];
            let op2elt : bits(8) = operand2[(4 * i + b)*:8];
            let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
            res = res + element1 * element2;
        end;
        result[e*:32] = res;
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.252 VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

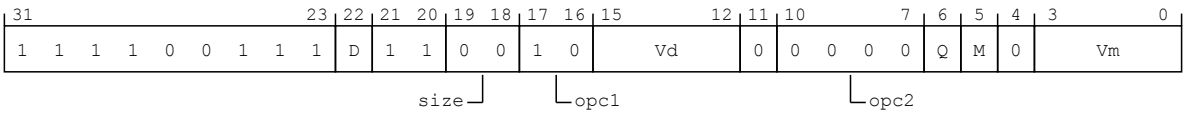

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSWP{<c>}{<q>}{.<dt>}  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSWP{<c>}{<q>}{.<dt>}  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

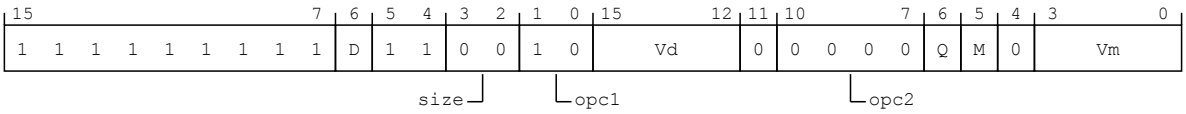

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VSWP{<c>}{<q>}{.<dt>}  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VSWP{<c>}{<q>}{.<dt>}  <Qd>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        if d == m then
            D(d+r) = ARBITRARY : bits(64);
        else
            D(d+r) = Din(m+r);
            D(m+r) = Din(d+r);
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.253 VTBL, VTBX

Vector Table Lookup and Extension

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31								23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	1	0	0	1	1	1	D	1	1	Vn				Vd				1	0	len		N	op	M	0	Vm			

Encoding for the VTBL variant

Applies when `(op == 0)`

```
VTBL{<c>}{<q>}.8  <Dd>, <list>, <Dm>
```

Encoding for the VTBX variant

Applies when `(op == 1)`

```
VTBX{<c>}{<q>}.8  <Dd>, <list>, <Dm>
```

Decode for all variants of this encoding

```
let is_vtbl : boolean = (op == '0');
let length : integer = UInt(len)+1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
if n+length > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3			0	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	1	1	1	1	1	1	D	1	1	Vn				Vd				1	0	len		N	op	M	0	Vm			

Encoding for the VTBL variant

Applies when `(op == 0)`

```
VTBL{<c>}{<q>}.8  <Dd>, <list>, <Dm>
```

Encoding for the VTBX variant

Applies when `(op == 1)`

```
VTBX{<c>}{<q>}.8  <Dd>, <list>, <Dm>
```

Decode for all variants of this encoding

```
let is_vtbl : boolean = (op == '0');
let length : integer = UInt(len)+1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
if n+length > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>

For the “A1 VTBL” and “A1 VTBX” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 VTBL” and “T1 VTBX” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<list>

The vectors containing the table. It must be one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

{<Dn>} Encoded as len = `0b00`.
{<Dn>, <Dn+1>} Encoded as len = `0b01`.
{<Dn>, <Dn+1>, <Dn+2>} Encoded as len = `0b10`.
{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>} Encoded as len = `0b11`.

<Dm>

Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    let table3 : bits(64) = if length == 4 then D(n+3) else Zeros{64};
    let table2 : bits(64) = if length >= 3 then D(n+2) else Zeros{64};
    let table1 : bits(64) = if length >= 2 then D(n+1) else Zeros{64};
    let table : bits(256) = table3 :: table2 :: table1 :: D(n);

    for i = 0 to 7 do
        let index : integer = UInt(D(m)[i*:8]);
        if index < 8*length then
            D(d)[i*:8] = table[index*:8];
        else
            if is_vtbl then
                D(d)[i*:8] = Zeros{8};
            // else D[d]<i*:8> unchanged
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.254 VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

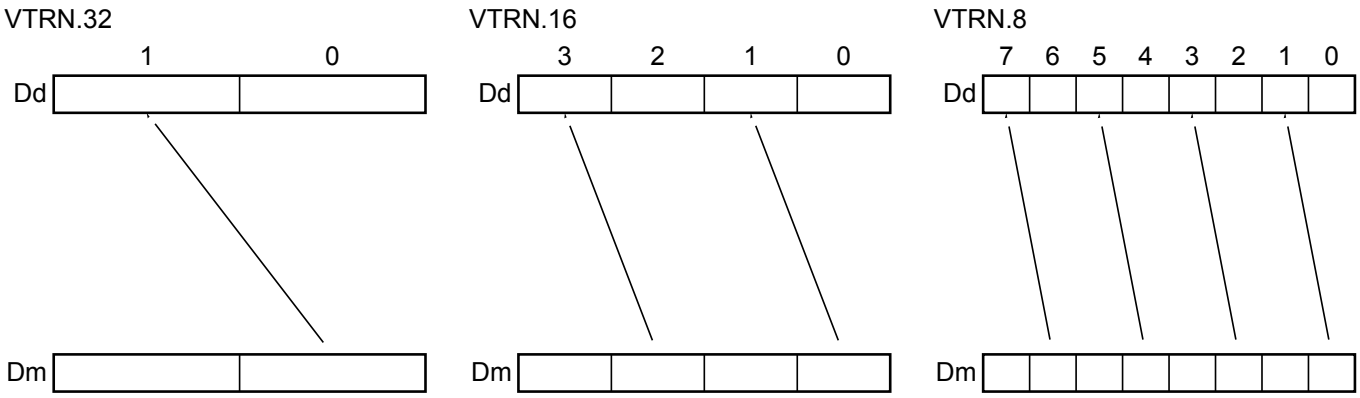

Figure F6-9 VTRN doubleword operations

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instructions VUZP (alias) and VZIP (alias).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0	0 0 0 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VTRN{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VTRN{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15–7	6	5–4	3–2	1–0	15–12	11	10–7	6	5	4	3–0
1 1 1 1 1 1 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0	0 0 0 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VTRN{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VTRN{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        if d == m then
            D(d+r) = ARBITRARY : bits(64);
        else
            for e = 0 to h-1 do
                D(d+r)[(2*e+1)*:esize] = Din(m+r)[(2*e)*:esize];
                D(m+r)[(2*e)*:esize] = Din(d+r)[(2*e+1)*:esize];
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.255 VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

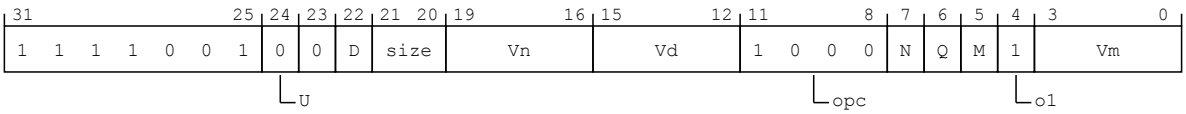

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VTST{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VTST{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

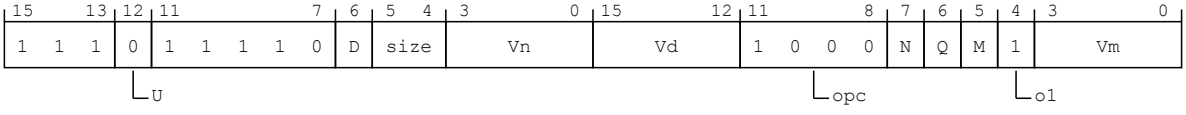

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VTST{<c>}{<q>}.<dt>  {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VTST{<c>}{<q>}.<dt>  {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	8
01	16
10	32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if !IsZero(D(n+r)[e*:esize] AND D(m+r)[e*:esize]) then
                D(d+r)[e*:esize] = Ones{esize};
            else
                D(d+r)[e*:esize] = Zeros{esize};
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.256 VUDOT (vector)

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_DotProd)

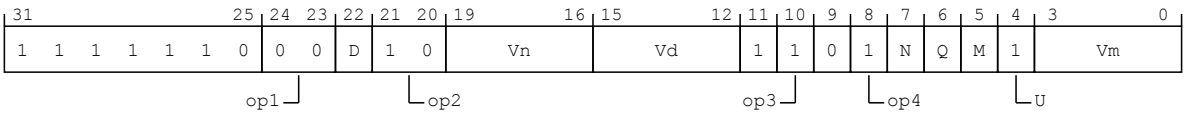

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUDOT{<q>}.U8  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUDOT{<q>}.U8  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)

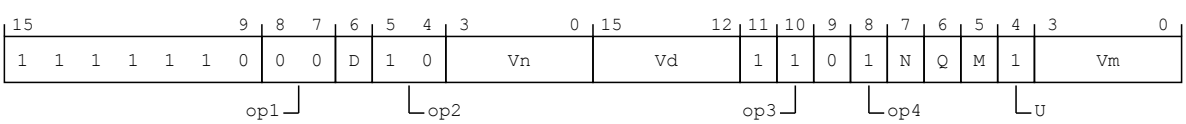

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VUDOT{<q>}.U8  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUDOT{<q>}.U8  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    operand2 = D(m+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if signed then
                element1 = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = SInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = UInt(operand2[(4 * e + i)*:(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*:esize] = result[e*:esize] + res;
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.257 VUDOT (by element)

Dot Product index form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_DotProd)

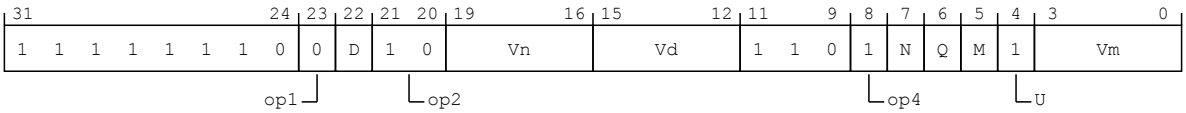

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUDOT{<q>}.U8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUDOT{<q>}.U8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)

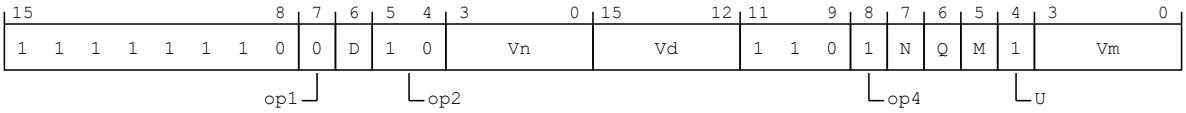

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VUDOT{<q>}.U8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUDOT{<q>}.U8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm” field.

<index>

Is the element index in the range 0 to 1, encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
var operand1 : bits(64);
let operand2 : bits(64) = D(m);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if signed then
                element1 = SInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = SInt(operand2[(4 * index + i)*:(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i)*:(esize DIV 4)]);
                element2 = UInt(operand2[(4 * index + i)*:(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*:esize] = result[e*:esize] + res;
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.258 VUMMLA

Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32I8MM)

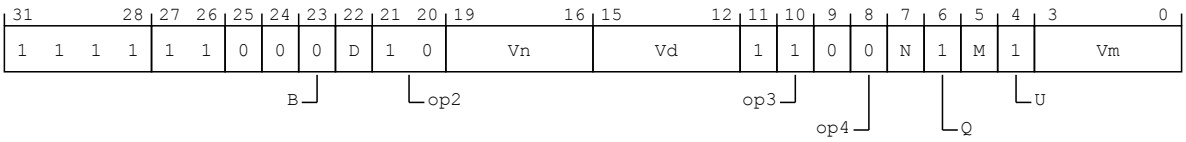

Encoding

```
VUMMLA{<q>}.U8  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
    when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AA32I8MM)

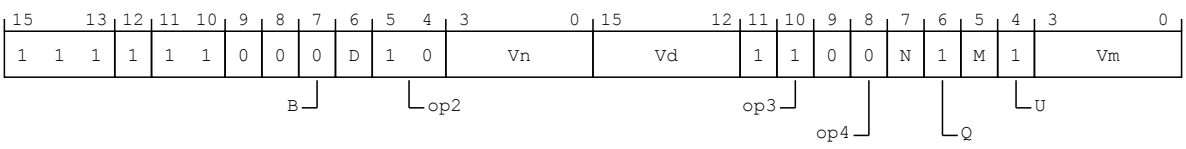

Encoding

```
VUMMLA{<q>}.U8  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
    when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let addend : bits(128)   = Q(d>>1);

Q(d>>1) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.259 VUSDOT (vector)

Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32I8MM)

31–25	24–23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 1 1 0	0 1	D	1 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm
	op1		op2			op3			op4				U	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUSDOT{<q>}.S8  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32I8MM)

15–9	8–7	6	5–4	3–0	15–12	11	10	9	8	7	6	5	4	3–0
1 1 1 1 1 1 0	0 1	D	1 0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm
	op1		op2			op3			op4				U	

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUSDOT{<q>}.S8  <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP third source and destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

for r = 0 to regs-1 do
    operand1 = Din(n+r);
    operand2 = Din(m+r);
    result = Din(d+r);
    for e = 0 to 1 do
        var res : bits(32) = result[e*:32];
        for b = 0 to 3 do
            let element1 : integer = UInt(operand1[(4 * e + b)*:8]);
            let element2 : integer = SInt(operand2[(4 * e + b)*:8]);
            res = res + element1 * element2;
        end;
        result[e*:32] = res;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.260 VUSDOT (by element)

Dot Product index form with unsigned and signed integers (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32I8MM)

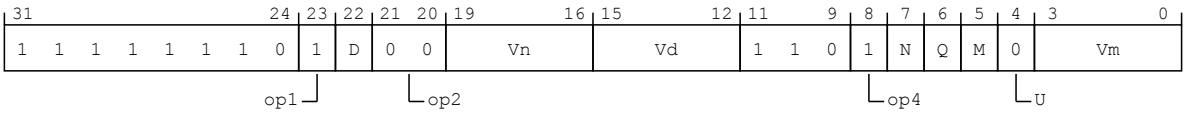

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUSDOT{<q>}.S8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32I8MM)

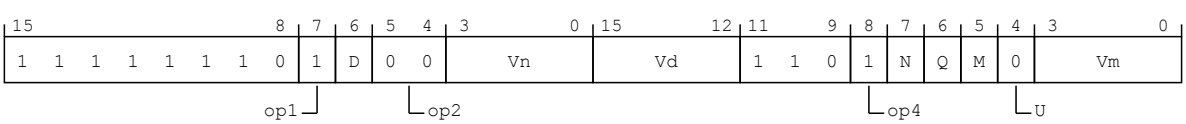

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VUSDOT{<q>}.S8  <Dd>, <Dn>, <Dm>[<index>]
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUSDOT{<q>}.S8  <Qd>, <Qn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm” field.

<index>

Is the element index in the range 0 to 1, encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

operand2 = Din(m);
for r = 0 to regs-1 do
    operand1 = Din(n+r);
    result = Din(d+r);
    for e = 0 to 1 do
        var res : bits(32) = result[e*:32];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        for b = 0 to 3 do
            let op1elt : bits(8) = operand1[(4 * e + b)*:8];
            let op2elt : bits(8) = operand2[(4 * i + b)*:8];
            let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
            res = res + element1 * element2;
        end;
        result[e*:32] = res;
    end;
    D(d+r) = result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.261 VUSMMLA

Widening 8-bit mixed integer matrix multiply-accumulate into 2x2 matrix

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(FEAT_AA32I8MM)

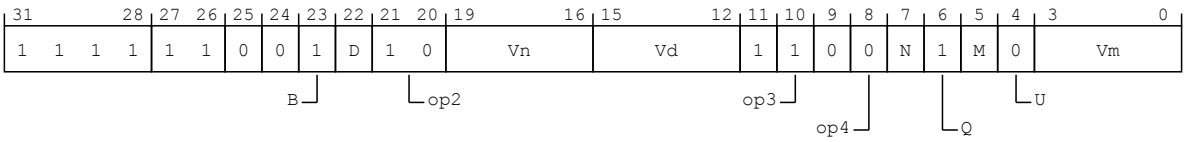

Encoding

```
VUSMMLA{<q>}.S8  <Qd>, <Qn>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
    when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AA32I8MM)

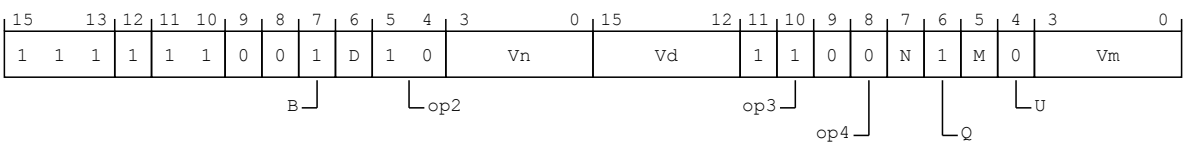

Encoding

```
VUSMMLA{<q>}.S8  <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
    when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<q>

See Standard assembler syntax fields.

<Qd>

Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let addend : bits(128)   = Q(d>>1);

Q(d>>1) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.262 VUZP

Vector Unzip de-interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

VUZP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

Figure F6-10 VUZP doubleword operation for data type 8

VUZP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

Figure F6-11 VUZP quadword operation for data type 32

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

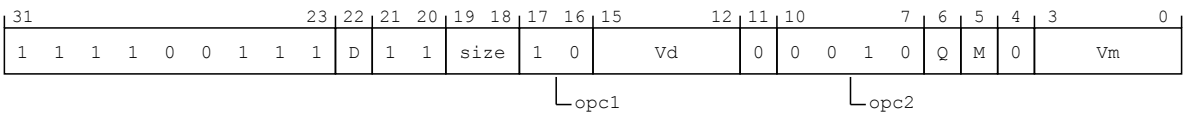

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUZP{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUZP{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11	10			7	6	5	4	3			0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			
														opc1									opc2								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VUZP{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VUZP{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q(d>>1) = ARBITRARY : bits(128);
        else
            let zipped_q : bits(256) = Q(m>>1)::Q(d>>1);
            for e = 0 to (128 DIV esize) - 1 do
                Q(d>>1)[e*:esize] = zipped_q[(2*e)*:esize];
                Q(m>>1)[e*:esize] = zipped_q[(2*e+1)*:esize];
            end;
        end;
    else
        if d == m then
            D(d) = ARBITRARY : bits(64);
        else
            let zipped_d : bits(128) = D(m)::D(d);
            for e = 0 to (64 DIV esize) - 1 do
                D(d)[e*:esize] = zipped_d[(2*e)*:esize];
                D(m)[e*:esize] = zipped_d[(2*e+1)*:esize];
            end;
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.263 VUZP (alias)

Vector Unzip de-interleaves the elements of two vectors.

This is a pseudo-instruction of VTRN. This means:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

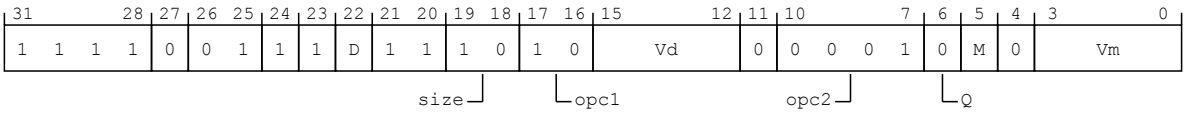

Encoding for the 64-bit SIMD vector variant

```
VUZP{<c>}{<q>}.32  <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

T1

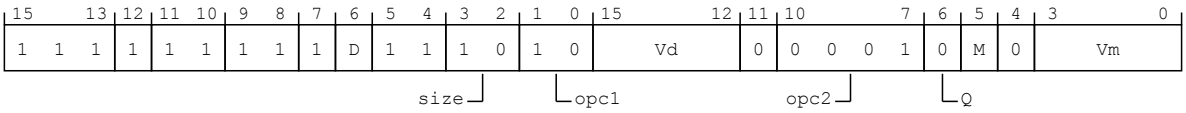

Encoding for the 64-bit SIMD vector variant

```
VUZP{<c>}{<q>}.32  <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Assembler Symbols

<c>

For the “A1 64-bit SIMD vector” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 64-bit SIMD vector” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

The description of VTRN gives the operational pseudocode for this instruction.

Operational Information

The description of VTRN gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.264 VZIP

Vector Zip interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

VZIP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B3	A3	B2	A2	B1	A1	B0	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	A7	B6	A6	B5	A5	B4	A4

Figure F6-12 VZIP doubleword operation for data type 8

VZIP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B1	A1	B0	A0
Qm	B3	B2	B1	B0	B3	A3	B2	A2

Figure F6-13 VZIP quadword operation for data type 32

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31–23	22	21–20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1 1 1 1 0 0 1 1 1	D	1 1	size	1 0 (opc1)	Vd	0	0 0 1 1 (opc2)	Q	M	0	Vm

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VZIP{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VZIP{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T1

15								7	6	5	4	3	2	1	0	15			12	11	10			7	6	5	4	3			0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			
														opc1									opc2								

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

```
VZIP{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

```
VZIP{<c>}{<q>}.<dt>  <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q(d>>1) = ARBITRARY : bits(128);
        else
            var zipped_q : bits(256);
            for e = 0 to (128 DIV esize) - 1 do
                zipped_q[(2*e)*:esize] = Q(d>>1)[e*:esize];
                zipped_q[(2*e+1)*:esize] = Q(m>>1)[e*:esize];
            end;
            Q(d>>1) = zipped_q[127:0];  Q(m>>1) = zipped_q[255:128];
        end;
    else
        if d == m then
            D(d) = ARBITRARY : bits(64);
        else
            var zipped_d : bits(128);
            for e = 0 to (64 DIV esize) - 1 do
                zipped_d[(2*e)*:esize] = D(d)[e*:esize];
                zipped_d[(2*e+1)*:esize] = D(m)[e*:esize];
            end;
            D(d) = zipped_d[63:0];  D(m) = zipped_d[127:64];
        end;
    end;
end;
```

Operational Information

This instruction is a data-independent-time instruction as described in About the DIT bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F6.1.265 VZIP (alias)

Vector Zip interleaves the elements of two vectors.

This is a pseudo-instruction of VTRN. This means:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

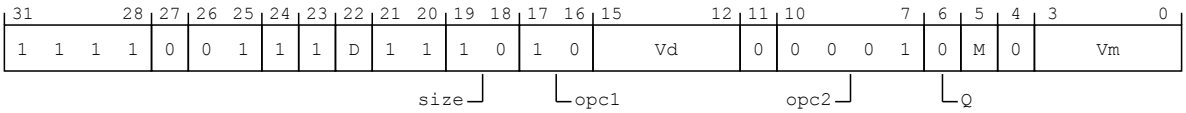

Encoding for the 64-bit SIMD vector variant

```
VZIP{<c>}{<q>}.32  <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

T1

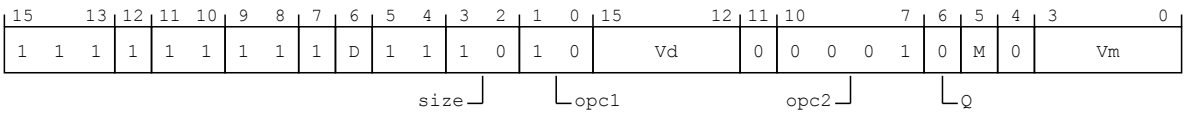

Encoding for the 64-bit SIMD vector variant

```
VZIP{<c>}{<q>}.32  <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.<dt>  <Dd>, <Dm>
```

Assembler Symbols

<c>

For the “A1 64-bit SIMD vector” variant: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 64-bit SIMD vector” variant: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

The description of VTRN gives the operational pseudocode for this instruction.

Operational Information

The description of VTRN gives the operational information for this instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part G
The AArch32 System Level Architecture

Chapter G1
The AArch32 System Level Programmers' Model

This chapter gives a system level description of the programmers' model for execution in AArch32 state. It contains the following sections:

- About the AArch32 System level programmers' model.
- Exception levels.
- Exception terminology.
- Execution state.
- Instruction Set state.
- Security state.
- Security state, Exception levels, and AArch32 execution privilege.
- Virtualization.
- AArch32 state PE modes.
- AArch32 general-purpose registers, the PC, and the Special-purpose registers.
- Process state, PSTATE.
- Instruction set states.
- Handling exceptions that are taken to an Exception level using AArch32.
- Routing of aborts taken to AArch32 state.
- Exception return to an Exception level using AArch32.
- Asynchronous exception behavior for exceptions taken from AArch32 state.
- AArch32 state exception descriptions.
- Reset into AArch32 state.
- Mechanisms for entering a low-power state.
- The AArch32 System register interface.
- Advanced SIMD and floating-point support.
- Configurable instruction controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1 About the AArch32 System level programmers' model

An application programmer has only a restricted view of the system. The System level programmers' model supports this application level view of the system, and includes features that are required for one or both of an operating system (OS) and a hypervisor to provide the programming environment seen by an application. This chapter describes the System level programmers' model when executing at EL1 or higher in an Exception level that is using AArch32.

The system level programmers' model includes all of the system features required to support operating systems and to handle hardware events.

The following sections give a system level introduction to the basic concepts of the Arm architecture AArch32 state, and the terminology that is used for describing the architecture when executing in this state:

- Exception levels.
- Exception terminology.
- Execution state.
- Instruction Set state.
- Security state.
- Virtualization.

The rest of this chapter describes the system level programmers' model when executing in AArch32 state.

The other chapters in this part describe:

- The memory system architecture, as seen when executing in an Exception level that is using AArch32:
  - The AArch32 System Level Memory Model describes the general features of the Armv8 memory model, when executing in AArch32 state, that are not visible at the application level.

    Note

    The AArch32 Application Level Memory Model describes the application level view of the memory model.

  - The AArch32 Virtual Memory System Architecture describes the *Virtual Memory System Architecture* (VMSA) used in AArch32 state.
- The AArch32 System registers, see AArch32 System Register Descriptions.

Note

The T32 and A32 instruction sets include instructions that provide system level functionality, such as returning from an exception. See, for example, ERET.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2 Exception levels

The architecture defines a set of Exception levels, EL0 to EL3, where:

- If ELn is the Exception level, increased values of *n* indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

Note

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

Effect of not implementing an Exception level provides information on implementations.

When executing in AArch32 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

Exception levels exist within *Security states*. The Armv8-A security model describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then because EL3 is only implemented in Secure state, execution at EL3 can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see The AArch32 Virtual Memory System Architecture.

G1.2.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

EL0 Applications.

EL1 OS kernel and associated functions that are typically described as *privileged*.

EL2 Hypervisor.

EL3 Secure monitor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.3 Exception terminology

The following subsections define the terms that are used when describing exceptions:

- Terminology for taking an exception.
- Terminology for returning from an exception.
- Exception levels.
- Definition of a precise exception.
- Definitions of synchronous and asynchronous exceptions.

G1.3.1 Terminology for taking an exception

An exception is *generated* when the PE first responds to an exceptional condition. The PE state at this time is the state that the exception is *taken from*. The PE state immediately after taking the exception is the state that the exception is *taken to*.

G1.3.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction. The PE state when an exception return instruction is committed for execution is the state the exception *returns from*. The PE state immediately after the execution of that instruction is the state that the exception *returns to*.

G1.3.3 Exception levels

An Exception level, EL*n*, with a larger value of *n* than another Exception level, is described as being a *higher* Exception level than the other Exception level. For example, EL3 is a higher Exception level than EL1.

An Exception level with a smaller value of *n* than another Exception level is described as being a *lower* Exception level than the other Exception level. For example, EL0 is a lower Exception level than EL1.

An Exception level is described as:

- *Using AArch64* when execution in that Exception level is in the AArch64 Execution state.
- *Using AArch32* when execution in that Exception level is in the AArch32 Execution state.

G1.3.4 Definition of a precise exception

An exception is described as *precise* when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

An exception is described as *imprecise* if it is not precise.

Other than the SError exception all exceptions that are taken to AArch32 state are required to be precise. For each occurrence of an SError exception, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

The terms precise and imprecise can also apply to Debug state entry. See Imprecise entry to Debug state.

Where a synchronous exception that is taken to AArch32 state is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by those instructions can be UNKNOWN, provided that:

- The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
- The registers are not involved in the calculation of the memory address that is used by the instruction.

In AArch32 state, examples of instructions that perform more than one single-copy atomic memory access are the `LDM` and `STM` instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For the definition of a single-copy atomic access, see Properties of single-copy atomic accesses.

G1.3.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about exceptions, see Handling exceptions that are taken to an Exception level using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.4 Execution state

The Execution states are:

AArch64 The 64-bit Execution state.

AArch32 The 32-bit Execution state.

Execution state gives more information about them.

Exception levels *use* Execution states. For example, EL0, EL1 and EL2 might all be using AArch32, under EL3 using AArch64.

This means that:

- Different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.
- The PE can change Execution states only either:
  - At reset.
  - On a change of Exception level.

Note

- Typical Exception level usage model shows which Exception levels different software layers might typically use.
- Effect of not implementing an Exception level gives information on supported configurations of Exception levels and Execution states.

The interaction between the AArch64 and AArch32 Execution states is called *interprocessing*. Interprocessing describes this.

G1.4.1 About the AArch32 PE modes

AArch32 state provides a set of *PE modes* that support normal software execution and handle exceptions. The current mode determines the set of registers that are available, as described in AArch32 general-purpose registers, the PC, and the Special-purpose registers.

The AArch32 modes are:

- Monitor mode. This mode always executes at Secure EL3.
- Hyp mode. This mode always executes at Non-secure EL2.
- System, Supervisor, Abort, Undefined, IRQ, and FIQ modes. The Exception level these modes execute at depends on the Security state, as described in Security state.
- User mode. This mode always executes at EL0.

Note

AArch64 state does not support modes. Modes are a concept that is specific to AArch32 state. Modes that execute at a particular Exception level are only implemented if that Exception level supports using AArch32 state.

For more information on modes, see AArch32 state PE modes.

The mode in use immediately before an exception is taken is described as the mode the exception is *taken from*. The mode that is used on taking the exception is described as the mode the exception is *taken to*.

All of the following define the mode that an exception is taken to:

- The type of exception.
- The mode the exception is taken from.
- Configuration settings defined at EL2 and EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Monitor mode and Hyp mode can create system traps that cause exceptions to EL3 or EL2 respectively. There is an architected hierarchy where EL2 and EL3 configuration settings affect a common condition, for example interrupt routing. When no traps are enabled for a particular condition, the AArch32 mode an exception is taken to is called the *default mode* for that exception.

In AArch32 state, a number of different modes can exist at the same Exception level. All modes at a particular Exception level have the same *execution privilege*, meaning they have the same access rights for accesses to memory and to System registers. However, the mapping of PE modes to Exception levels depends on the Security state, as described in Security state. Security state, Exception levels, and AArch32 execution privilege gives more information about the PE modes, their associated execution privilege, and how this maps onto the Exception levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.5 Instruction Set state

In AArch32 state, the *Instruction Set state* determines the instruction set that the PE is executing. In an implementation that follows the Arm recommendations, the available Instruction Set states are:

T32 state The PE is executing T32 instructions.

A32 state The PE is executing A32 instructions.

Note

In previous versions of the Arm architecture:

- The T32 instruction set was called the Thumb instruction set.
- The A32 instruction set was called the ARM instruction set.

For more information, see Process state, PSTATE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.6 Security state

The architecture provides two Security states, each with an associated physical memory address space, as follows:

Secure state When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

Non-secure state When in this state, the PE:

- Can access only the Non-secure physical address space.
- Cannot access the Secure system control resources.

For information on how virtual addresses translate onto Secure physical and Non-secure addresses, see About VMSA v8-32.

G1.6.1 The Armv8-A security model

The principles of the Armv8-A security model are defined in Security states.

G1.6.1.1 The AArch32 security model, and execution privilege

The Exception level hierarchy of four Exception levels, EL0, EL1, EL2, and EL3, applies to execution in both Execution states. This section describes the mapping between Exception levels, AArch32 modes, and execution privilege.

The AArch32 modes Monitor, System, Supervisor, Abort, Undefined, IRQ, and FIQ all have the same execution privilege.

In Secure state:

- Monitor mode executes only at EL3, and is accessible only when EL3 is using AArch32.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode all:
  - Execute at EL1 when EL3 is using AArch64.
  - Execute at EL3 when EL3 is using AArch32.

This means that there is a difference in the Secure state hierarchy that the PE is using, depending on which Execution state EL3 is using:

- If EL3 is using AArch64:
  - There is no support for Monitor mode.
  - If EL1 is using AArch32, System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL1.
- If EL3 is using AArch32:
  - Monitor mode is supported, and executes at Secure EL3.
  - System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL3.
  - There is no support for a Secure EL1 Exception level.

See Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64 for more information about operation in a Secure EL1 mode when EL3 is using AArch64.

In Non-secure state, the PL1 modes System, Supervisor, Abort, Undefined, IRQ, and FIQ always execute at EL1.

User mode always executes at EL0 and has the lowest possible execution privilege.

Hyp mode always executes in Non-secure state at EL2 and has higher execution privilege than all of:

- User mode.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Limited use of Privilege level in AArch32 state describes how, in some contexts, the concept of *Privilege levels* can be used to represent the execution privilege hierarchy.

For more information about the modes, see About the AArch32 PE modes.

Figure G1-1 shows the security model when EL3 is using AArch32, and shows the expected use of the different Exception levels, and which modes execute at which Exception levels.

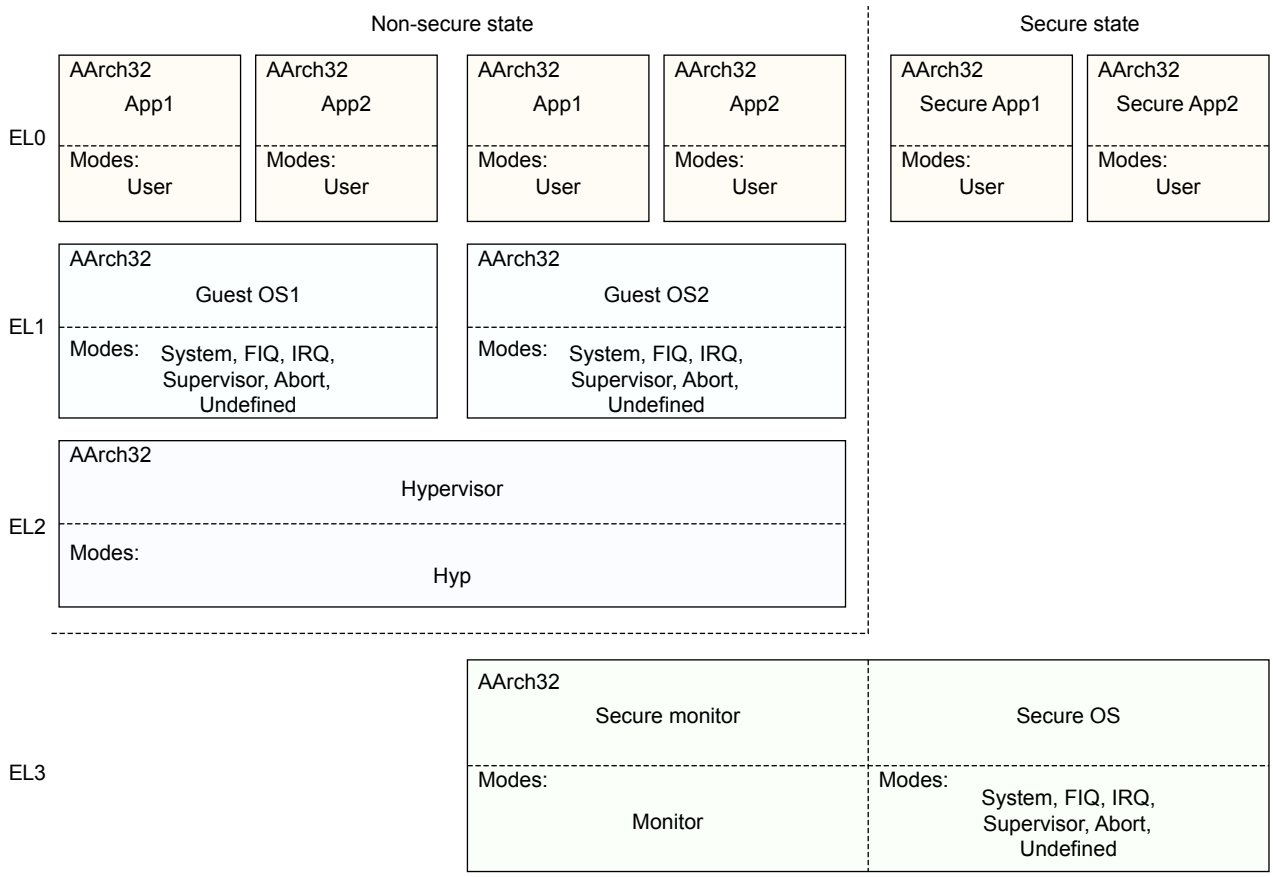

Figure G1-1 Armv8-A Security model when EL3 is using AArch32

Note

For an overview of the Security models when EL3 is using AArch64 and EL2, EL1, and EL0 are all using AArch32, see Figure G1-2.

Figure G1-1 shows that when EL3 is using AArch32, the Exception levels and modes available in each Security state are as follows:

Secure state

EL0 User mode.

EL3 Any mode that is available in Secure state, other than User mode.

Non-secure state

EL0 User mode.

EL1 Any mode that is available in Non-secure state, other than Hyp mode and User mode.

EL2 Hyp mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Execution at EL0 is described as *unprivileged execution*.

A mode associated with a particular Exception level, EL*n*, is described as an EL*n* mode.

Note

The Exception level defines the ability to access resources in the current Security state, and does not imply anything about the ability to access resources in the other Security state.

When EL3 is using AArch32, many AArch32 System registers accessible at PL1 are *banked* between the Secure and Non-secure states.

When EL3 is using AArch64 and Secure EL1 is using AArch32, System registers accessible at PL1 are not banked between the Non-secure and Secure states. Software running at EL3 is expected to switch the content of the PL1-accessible System registers between the Secure and Non-secure context, in a similar manner to switching the contents of general purpose registers. For information on the relationship between AArch64 and AArch32 System registers in an interprocessing environment, see Mapping of the System registers between the Execution states.

For more information on the System registers, see The AArch32 System register interface.

The *Secure Monitor Call* (`SMC`) instruction provides software with a system call to EL3. When executing at a privileged Exception level, `SMC` instructions generates exceptions. For more information, see Secure Monitor Call (SMC) exception and SMC.

Note

For more information about the Privilege level terminology, see Security state, Exception levels, and AArch32 execution privilege.

G1.6.1.2 Changing from Secure state to Non-secure state

Monitor mode is provided to support switching between Secure and Non-secure states. When executing in an Exception level that is using AArch32, except in Monitor mode and Hyp mode, the Security state is controlled:

- By the SCR.NS bit, when EL3 is using AArch32.
- By the SCR_EL3.NS bit, when EL3 is using AArch64.

The mapping of AArch32 privileged modes to the exception hierarchy means that it is possible when EL3 is using AArch32 to change from EL3 to Non-secure EL1 without an exception return. This can occur in one of the following ways:

- Using an `MSR` or `CPS` instruction to switch from Monitor mode to another privileged mode while SCR.NS is 1.
- Using an `MCR` instruction that writes SCR.NS to change from Secure to Non-secure state when in a privileged mode other than Monitor mode.

Arm strongly recommends that software executing at EL3 using AArch32 does not use either of these mechanisms to change from EL3 to Non-secure EL1 without an exception return. The use of both of these mechanisms is deprecated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.7 Security state, Exception levels, and AArch32 execution privilege

The hierarchy of software execution privilege, within a particular Security state, is defined by the Exception levels, with higher Exception level numbers indicating higher privilege. Table G1-1 shows this hierarchy for each Security state.

Table G1-1 Execution privilege and Exception levels, by Security state

Execution privilege	Secure state	Non-secure state	Typical use
Highest	EL3	-[a]	Secure monitor
-	EL2[b]	EL2	Hypervisor
-	EL1	EL1	Secure or Non-secure OS
Lowest, Unprivileged	EL0	EL0	Secure or Non-secure application

a. EL3 is never implemented in Non-secure state.

b. If FEAT_SEL2 is implemented in AArch64 state, EL2 can be enabled in Secure state.

When executing in AArch32 state, within a given Security state, the current PE state, including the execution privilege, is primarily indicated by the current *PE mode*. In Secure state, how the PE modes map onto the Exception levels depends on whether EL3 is using AArch32 or is using AArch64, and:

- Figure G1-1 shows this mapping when EL3 is using AArch32.
- Figure G1-2 shows this mapping when EL3 is using AArch64.

Table G1-2 shows this mapping. In interpreting this table:

- Monitor mode is implemented only in Secure state, and only if EL3 is using AArch32.
- Hyp mode is implemented only in Non-secure state, and only if EL2 is using AArch32.
- System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented:
  In Secure state If either:
  — EL3 is using AArch32.
  — EL3 is using AArch64 and EL1 is using AArch32.
  In Non-secure state If EL1 is using AArch32.
- User mode is implemented if EL0 is using AArch32.

Table G1-2 Mapping of AArch32 PE modes to Exception levels

Exception level	PE modes in the given Security state, and EL3 Execution state		
	Secure state, EL3 using AArch32	Secure state, EL3 using AArch64[a]	Non-secure state
EL3	Monitor, System, FIQ, IRQ, Supervisor, Abort, Undefined	-	-
EL2	-	-	Hyp
EL1	-	System, FIQ, IRQ, Supervisor, Abort, Undefined	System, FIQ, IRQ, Supervisor, Abort, Undefined
EL0	User	User	User

a. If EL3 is not implemented, but FEAT_SEL2 is implemented, this column applies.

Because AArch32 behavior is described in terms of the PE modes, and transitions between PE modes, the Exception levels are implicit in most of the description of operation in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.7.1 Limited use of Privilege level in AArch32 state

As described in The VMSAv8-32 translation regimes, a *translation regime* maps a virtual address (VA) to the corresponding physical address (PA). The VMSAv8-64 translation regimes are defined by the Exception levels that use them. However, because the mapping between PE modes and Exception levels in Secure state depends on whether EL3 is using AArch32 or is using AArch64, as shown in Table G1-2, the VMSAv8-32 translation regimes cannot be described simply in terms of either the Exception levels or the PE modes that use them.

To provide a consistent description of address translation as seen from AArch32 state, the VMSAv8-32 translation regimes are described in terms of the *Privilege levels* originally defined in the Armv7 descriptions of AArch32 state. Table G1-3 shows how the PE modes map to these Privilege levels:

Table G1-3 Mapping of PE modes to AArch32 Privilege levels

Privilege level	Secure state	Non-secure state
PL2	-	Hyp[a]
PL1	Monitor[b], System, FIQ, IRQ, Supervisor, Abort, Undefined	System, FIQ, IRQ, Supervisor, Abort, Undefined
PL0	User	User

a. Implemented only in Non-secure state, and only if EL2 is using AArch32 state.

b. Implemented only in Secure state, and only if EL3 is using AArch32 state.

Comparing Table G1-3 with Table G1-2 shows that:

In Non-secure state

Each privilege level maps to the corresponding Exception level. For example, PL1 maps to EL1.

In Secure state

PL0 maps to EL0.

The mapping of PL1 depends on the Execution state being used by EL3, as follows:

EL3 using AArch64 Secure PL1 maps to Secure EL1. Monitor mode is not implemented.

EL3 using AArch32 Secure PL1 maps to Secure EL3. Monitor mode is implemented as one of the Secure PL1 modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.8 Virtualization

The support for virtualization described in this section applies only to an implementation that includes EL2. A PE is in *Hyp mode* when it is executing at EL2 in the AArch32 state. An exception return from Hyp mode to software running at EL1 or EL0 is performed using the `ERET` instruction.

EL2 provides a set of features that support virtualizing the Non-secure state of an A-profile implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between *virtual machines*. A virtual machine is comprised of Non-secure EL1 and Non-secure EL0.
- A number of Guest operating systems, that each run in Non-secure EL1, on a virtual machine.
- For each Guest operating system, applications, that usually run in Non-secure EL0, on a virtual machine.

Note

In some systems, a Guest OS is unaware that it is running on a virtual machine, and is unaware of any other Guest OS. In other systems, a hypervisor makes the Guest OS aware of these facts. The architecture supports both of these models.

The hypervisor assigns a VMID to each virtual machine.

In AArch32 state, EL2 is implemented only in Non-secure state, to support Guest OS management. EL2 provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest OS or the applications for a Guest OS returns the virtual value.
- *Trap* various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2.
- Route interrupts to the appropriate one of:
  - The current Guest OS.
  - A Guest OS that is not currently running.
  - The hypervisor.

In Non-secure state:

- The implementation provides an independent *translation regime* for memory accesses from EL2.
- For the PL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the *virtual address* (VA) to an *intermediate physical address* (IPA). This is managed at EL1, usually by a Guest OS. The Guest OS believes that the IPA is the *physical address* (PA).
  - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest OS might be unaware of this stage.

For more information on the translation regimes, see The AArch32 Virtual Memory System Architecture.

G1.8.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- Hypervisor Call (HVC) exception.
- Traps to EL2. EL2 configurable controls, describes these.
- All of the virtual interrupts:
  - Virtual SError.
  - Virtual IRQ.
  - Virtual FIQ.

HVC exceptions are always taken to EL2. All virtual interrupts are always taken to EL1, and can only be taken from Non-secure EL1 or EL0.

Each of the virtual interrupts can be independently enabled using controls at EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Each of the virtual interrupts has a corresponding physical interrupt. See Virtual interrupts.

When a virtual interrupt is enabled, its corresponding physical exception is taken to EL2, unless EL3 has configured that physical exception to be taken to EL3. For more information, see Asynchronous exception behavior for exceptions taken from AArch32 state.

An implementation that includes EL2 also:

- Provides controls that can be used to route some synchronous exceptions, taken from Non-secure state, to EL2. For more information, see:
  - Routing exceptions from Non-secure EL0 to EL2.
  - Routing debug exceptions to EL2 using AArch32.
  - Routing of aborts taken to AArch32 state.
- Provides mechanisms to trap PE operations to EL2. See EL2 configurable controls.
  When an operation is trapped to EL2, the hypervisor typically either:
  - Emulates the required operation. The application running in the Guest OS is unaware of the trap.
  - Returns an error to the Guest OS.

G1.8.1.1 Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in Table G1-4.

Table G1-4 The virtual interrupts

Physical interrupt	Corresponding virtual interrupt
External SError	Virtual SError
IRQ	Virtual IRQ
FIQ	Virtual FIQ

Software executing at EL2 can use virtual interrupts to signal physical interrupts to Non-secure EL1 and Non-secure EL0. Example G1-1 shows a usage model for virtual interrupts.

Example G1-1 Virtual interrupt usage model

A usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.
2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest OS in EL1. If the interrupt requires routing to a Guest OS:
   - If the Guest OS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.
   - If the Guest OS is not currently running, the physical interrupt is marked as pending for the guest OS. When the hypervisor next switches to the virtual machine that is running that Guest OS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.

Non-secure EL1 and Non-secure EL0 modes cannot distinguish a virtual interrupt from the corresponding physical interrupt.

For more information, see Virtual exceptions when an implementation includes EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.9 AArch32 state PE modes

Table G1-5 shows the PE modes defined by the Arm architecture, for execution in AArch32 state. In this table:

- The *PE mode* column gives the name of each mode and the abbreviation used, for example, in the general-purpose register name suffixes used in AArch32 general-purpose registers, the PC, and the Special-purpose registers.
- The *Encoding* column gives the corresponding PSTATE.M field.
- The *Exception level* column gives the Exception level at which the mode is implemented, including dependencies on the current Security state and on whether EL3 is using AArch32, see Exception levels.

Table G1-5 AArch32 PE modes

PE mode		Encoding	Security state	Exception level	Implemented
User	usr	10000	Both	EL0	Always
FIQ	fiq	10001	Non-secure Secure	EL1 EL1 or EL3[a]	Always
IRQ	irq	10010	Non-secure Secure	EL1 EL1 or EL3[a]	Always
Supervisor	svc	10011	Non-secure Secure	EL1 EL1 or EL3[a]	Always
Monitor	mon	10110	Secure	EL3	If EL3 implemented and using AArch32
Abort	abt	10111	Non-secure Secure	EL1 EL1 or EL3[a]	Always
Hyp	hyp	11010	Non-secure	EL2	If EL2 implemented and using AArch32
Undefined	und	11011	Non-secure Secure	EL1 EL1 or EL3[a]	Always
System	sys	11111	Non-secure Secure	EL1 EL1 or EL3[a]	Always

a. EL3 if EL3 is using AArch32. EL1 if EL3 is using AArch64 and EL1 is using AArch32.

Note

FEAT_SEL2 is not supported if EL2 is using AArch32.

Mode changes can be made under software control, or can be caused by an external or internal exception.

G1.9.1 Notes on the AArch32 PE modes

PE modes are defined only in AArch32 state. Because each mode is implemented as part of a particular Exception level that is using AArch32, the set of available modes depends on which Exception levels are implemented and using AArch32, as described in Effect of the EL3 Execution state on the PE modes and Exception levels.

This section gives more information about each of the modes, when it is implemented.

User mode

Software executing in User mode executes at EL0. Execution in User mode is sometimes described as unprivileged execution. Application programs normally execute in User mode, and any program executed in User mode:

- Makes only unprivileged accesses to system resources, meaning it cannot access protected system resources.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Makes only unprivileged access to memory.
- Cannot change mode except by causing an exception, see Handling exceptions that are taken to an Exception level using AArch32.

System mode

System mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels.

System mode has the same registers available as User mode, and is not entered by any exception.

Supervisor mode

Supervisor mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels.

Supervisor mode is the default mode to which a Supervisor Call exception is taken. Executing an `SVC` (Supervisor Call) instruction generates a Supervisor Call exception.

In an implementation where the highest implemented Exception level is using AArch32, if that Exception level is EL3 or EL1, a PE enters Supervisor mode on reset.

Abort mode

Abort mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels.

Abort mode is the default mode to which a Data Abort exception or Prefetch Abort exception is taken.

Undefined mode

Undefined mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels.

Undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an UNDEFINED instruction, is taken.

FIQ mode

FIQ mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels.

FIQ mode is the default mode to which an FIQ interrupt is taken.

IRQ mode

IRQ mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels.

IRQ mode is the default mode to which an IRQ interrupt is taken.

Hyp mode

Hyp mode is the Non-secure EL2 mode.

Hyp mode is entered on taking an exception from Non-secure state that must be taken to EL2.

In an implementation where the highest implemented Exception level is EL2 and EL2 uses AArch32 on reset, a PE enters Hyp mode on reset.

The Hypervisor Call exception and Hyp Trap exception are implemented as part of EL2 and are always taken to Hyp mode when EL2 is using AArch32.

Executing an `HVC` (Hypervisor Call) instruction generates a Hypervisor Call exception. See Hypervisor Call (HVC) exception.

For more information, see Hyp mode.

Monitor mode

Monitor mode is the Secure EL3 mode. This means it is always in the Secure state, regardless of the value of the SCR.NS bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Monitor mode is the mode to which a Secure Monitor Call exception is taken. In a Non-secure EL1 mode, or a Secure EL3 mode, executing an `SMC` (Secure Monitor Call) instruction generates a Secure Monitor Call exception.

When EL3 is using AArch32, some exceptions that are taken to a different mode by default can be configured to be taken to EL3, see PE mode for taking exceptions.

When EL3 is using AArch32, software executing in Monitor mode:

- Has access to both the Secure and Non-secure copies of System registers.
- Can perform an exception return to Secure state, or to Non-secure state.

This means that, when EL3 is using AArch32, Monitor mode provides the only recommended method of changing between the Secure and Non-secure Security states.

Secure and Non-secure modes

In an implementation that includes EL3, the names of most implemented modes can be qualified as Secure or Non-secure, to indicate whether the PE is also in Secure state or Non-secure state. For example:

- If a PE is in Supervisor mode and Secure state, it is in *Secure Supervisor mode*.
- If a PE is in User mode and Non-secure state, it is in *Non-secure User mode*.

Note

As indicated in the appropriate Mode descriptions:

- Monitor mode is a Secure mode, meaning it is always in the Secure state.
- Hyp mode is a Non-secure mode, meaning it is accessible only in Non-secure state.

G1.9.1.1 Effect of the EL3 Execution state on the PE modes and Exception levels

Figure G1-1 shows the PE modes, Exception levels, and Security states, for an implementation that includes all of the Exception levels, when EL3 is using AArch32. Figure G1-2 shows how the implemented modes change when EL3 is using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

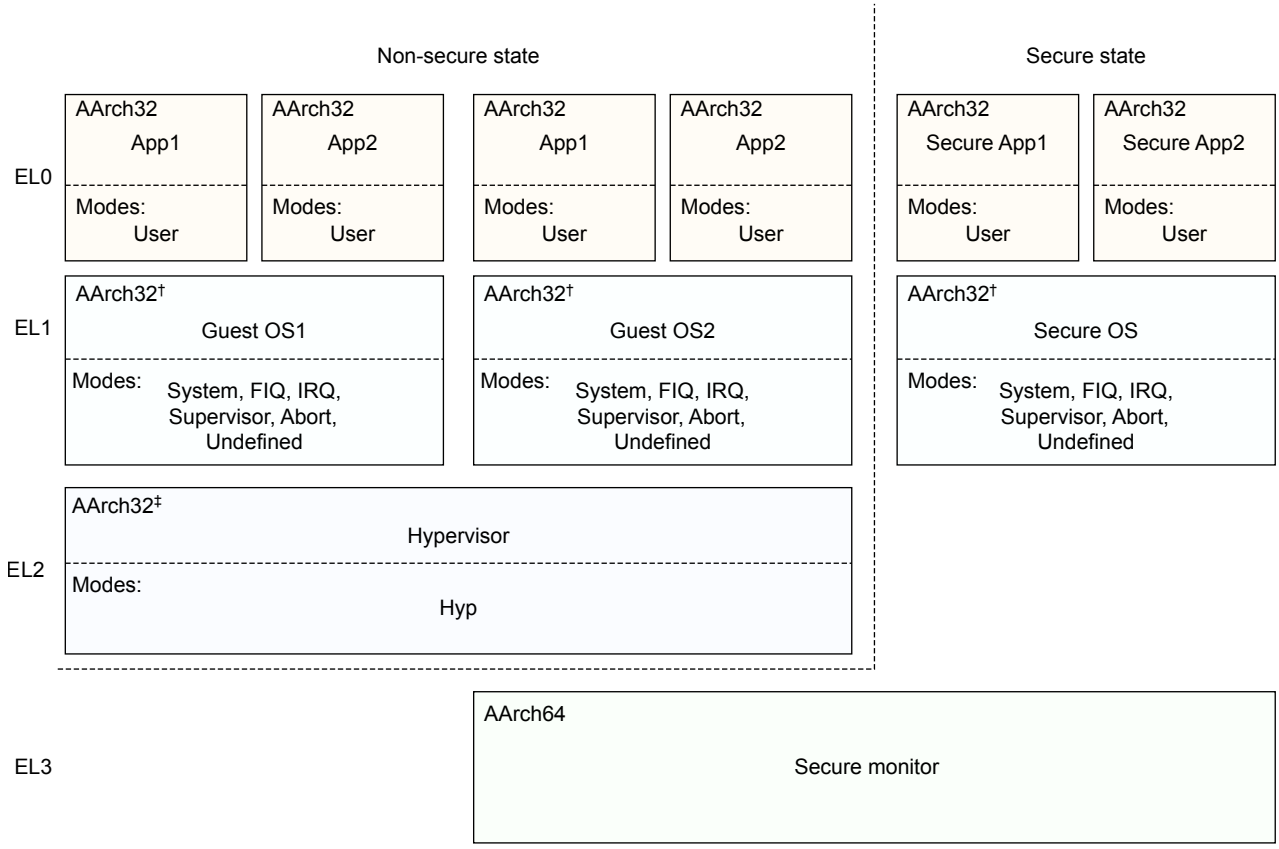

† When EL1 is using AArch64, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are not implemented
‡ When EL2 is using AArch64, Hyp mode is not implemented

Figure G1-2 Exception levels, and PE modes, when EL3 is using AArch64

Comparing Figure G1-1 and Figure G1-2 shows how, in Secure state only, the implementation of System, FIQ, IRQ, Supervisor, Abort, and Undefined mode depends on the Execution state that EL3 is using. That is, these modes are implemented as follows:

Non-secure state

If Non-secure EL1 is using AArch32, then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Non-secure state.

Secure state

The implementation of these modes depends on the Execution state that EL3 is using, as follows:

EL3 using AArch64

If Secure EL1 is using AArch32, then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Secure state.

EL3 using AArch32

In Secure state, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL3, see Figure G1-1.

G1.9.2 Hyp mode

Hyp mode is the Non-secure EL2 mode. When EL2 is using AArch32, it provides the usual method of controlling the virtualization of Non-secure execution at EL1 and EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The alternative method of controlling this functionality is by accessing the EL2 controls from EL3 with the SCR_EL3.NS or SCR.NS bit set to 1.

This section summarizes how Hyp mode differs from the other modes, and references where this part of the manual describes the features of Hyp mode in more detail:

- Software executing in Hyp mode executes at EL2, see Figure G1-1.
- Hyp mode is accessible only in Non-secure state. In Secure state, an attempt by a `CPS` or an `MSR` instruction to change PSTATE.M to Hyp mode is an illegal change to PSTATE.M, as described in Illegal changes to PSTATE.M.
- In Non-debug state, the only mechanisms for changing to Hyp mode are:
  - An exception taken from a Non-secure EL1 or EL0 mode.
  - When EL3 is using AArch32, an exception return from Secure Monitor mode.
  - When EL3 is using AArch64, an exception return from EL3.
- In Hyp mode, the only exception return is execution of an `ERET` instruction, see ERET.
- In Hyp mode, the CPACR has no effect on the execution of:
  - System register access instructions.
  - Advanced SIMD and floating-point instructions.

  The HCPTR controls execution of these instructions in Hyp mode.
- If software running in Hyp mode executes an `SVC` instruction, the Supervisor Call exception generated by the instruction is taken to Hyp mode, see SVC.
- An exception return with restored PSTATE specifying Hyp mode is an *illegal return event*, as described in Illegal return events from AArch32 state, if any of the following applies:
  - EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  - EL3 is using AArch32 and the value of SCR.NS is 0.
  - The return is from a Non-secure EL1 mode.
- The instructions described in the following sections are UNDEFINED if executed in Hyp mode:
  - SRS. See SRS, SRSDA, SRSDB, SRSIA, SRSIB.
  - RFE. See RFE, RFEDA, RFEDB, RFEIA, RFEIB.
  - LDM (exception return).
  - LDM (User registers).
  - STM (User registers).
  - The `SUBS PC, LR` forms of the instructions described in SUB, SUBS (immediate).

    Note

    In T32 state, `ERET` is encoded as `SUBS PC, LR, #0`, and therefore this is a valid instruction.

  - The exception return form of the instructions described in MOV, MOVS (register).

  In addition, deprecated forms of the A32 `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS`, and `SUBS` instructions with the PC as the destination register are UNDEFINED if executed in Hyp mode. The instruction descriptions identify these UNDEFINED cases.
- The Load unprivileged and Store unprivileged instructions `LDRT`, `LDRSHT`, `LDRHT`, `LDRBT`, `STRT`, `STRHT`, and `STRBT`, are CONSTRAINED UNPREDICTABLE if executed in Hyp mode.

To permit entry to Hyp mode using the Hypervisor Call exception, Secure software must enable use of the `HVC` instruction:

- By setting the SCR_EL3.HCE bit to 1, if EL3 is using AArch64.
- By setting the SCR.HCE bit to 1, if EL3 is using AArch32.

If EL3 is implemented and using AArch32, and SCR.HCE is set to 0, the `HVC` instruction is UNPREDICTABLE in Hyp mode. The instruction is either UNDEFINED or executes as a `NOP`.

If EL3 is implemented and using AArch64, and SCR_EL3.HCE is set to 0, the `HVC` instruction is UNDEFINED in Hyp mode.

If EL3 is not implemented and HCR.HCD is set to 1, the `HVC` instruction is UNDEFINED in Hyp mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.9.3 Pseudocode description of mode operations

The `BadMode()` function tests whether a 5-bit mode number corresponds to one of the permitted modes.

The `BadMode()` function is defined in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.10 AArch32 general-purpose registers, the PC, and the Special-purpose registers

The general-purpose registers, and the PC, in AArch32 state describes the application level view of the general-purpose registers, and the PC. This view provides:

- The general-purpose registers R0-R14, of which:
  - The preferred name for R13 is SP (*stack pointer*).
  - The preferred name for R14 is LR (*link register*).
- The PC, which can be described as R15.

These registers are selected from a larger set of registers that includes *banked* copies of some registers, with the current register selected by the execution mode. The implementation and banking of the general-purpose registers depends on whether or not the implementation includes EL2 and EL3, and whether those Exception levels are using AArch32. Figure G1-3 shows the full set of banked general-purpose registers, and the Special-purpose registers:

- The Program Status Registers CPSR and SPSR.
- ELR_hyp.

Note

The architecture uses system level register names, such as R0_usr, R8_usr, and R8_fiq, when it must identify a specific register. The application level names refer to the registers for the current mode, and usually are sufficient to identify a register.

Application level view	System level view								
	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

‡ Part of EL3. Exists only in Secure state, and only when EL3 is using AArch32.
† Part of EL2. Exists only in Non-secure state, and only when EL2 is using AArch32.
Cells with no entry indicate that the User mode register is used.

Figure G1-3 AArch32 general-purpose registers, PC, and Special-purpose registers, showing banking

As described in PE mode for taking exceptions, on taking an exception the PE changes mode, unless it is already in the mode to which it must take the exception. Each mode that the PE might enter in this way has:

- A banked copy of the stack pointer, for example SP_irq and SP_hyp.
- A register that holds a preferred return address for the exception. This is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - For the EL2 mode, Hyp mode, the Special-purpose register ELR_hyp.
  - For the other privileged modes to which exceptions can be taken, a banked copy of the link register, for example LR_und and LR_mon.
- A saved copy of PSTATE, made on exception entry, for example SPSR_irq and SPSR_hyp.

In addition, FIQ mode has banked copies of the general-purpose registers R8 to R12.

User mode and System mode share the same general-purpose registers.

User mode, System mode, and Hyp mode share the same LR.

For more information about the application level view of the SP, LR, and PC, and the alternative descriptions of them as R13, R14 and R15, see The general-purpose registers, and the PC, in AArch32 state.

In AArch32 state, the Special-purpose registers are:

- The CPSR and its view as the APSR.
- The SPSR, including the banked copies SPSR_abt, SPSR_fiq, SPSR_hyp, SPSR_irq, SPSR_mon, SPSR_svc, and SPSR_und.
- The ELR_hyp.

G1.10.1 Pseudocode description of general-purpose register and PC operations

The following pseudocode gives access to the general-purpose registers and the PC. These registers are an array, `_R`, indexed by parameter `n`. This array is common to AArch32 and AArch64 operation and therefore contains 31 64-bit registers. `_PC` is the Program Counter, and its definition is common to AArch32 and AArch64 operation and therefore its size is 64-bit.

`LookUpRIndex()` looks up the index value, `n`, for the specified register number and PE mode, using `RBankSelect()` to evaluates the result.

`_R` accesses the specified general-purpose register in the current PE mode, using `Rmode()` to access the register, accessing `_R` if necessary. `SP` accesses the stack pointer, `LR` accesses the link register, and `PC32` accesses the Program Counter. Each function has a non-assignment form for register reads and an assignment form for register writes, other than `PC32`, which has only a non-assignment form.

`BranchTo()` performs a branch to the specified address.

The `_R` `_PC`, `LR`, `PC32`, `SP`, `LookUpRIndex()`, `RBankSelect()`, `Rmode()`, and `BranchTo()` functions are defined in A-profile Architecture Pseudocode.

G1.10.2 Saved Program Status Registers (SPSRs)

The Saved Program Status Registers (SPSRs) are used to save PE state on taking exceptions. In AArch32 state, there is an SPSR for every mode that an exception can be taken to, as shown in Figure G1-3. For example, the SPSR for Monitor mode is called SPSR_mon.

Note

Exceptions cannot be taken to EL0.

When the PE takes an exception, PE state is saved from PSTATE in the SPSR for the mode the exception is taken to. For example, if the PE takes an exception to Monitor mode, PE state is saved in SPSR_mon. For more information on PSTATE, see Process state, PSTATE.

Note

All PSTATE fields are saved, including those which have no direct read and write access.

Saving the PSTATE fields means the exception handler can:

- On return from the exception, restore the PE state to the values it had immediately before the exception was taken. When the PE returns from an exception, PE state is restored to the state stored in the SPSR of the mode the exception is returning from, if the exception return is made using one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - ERET.
  - LDM.
  - The Exception return form of the instruction described in MOV, MOVS (register).
  - The Exception return form of the instruction described in SUB, SUBS (immediate).

  For example, on returning from Monitor mode, PE state is restored to the state stored in SPSR_mon. If the exception return is made using the RFE instruction, the PE restores the PE state from an SPSR valued read from memory.

- Examine the value that PSTATE had when the exception was taken, for example to determine the instruction set state and privilege level in which the instruction that caused an Undefined Instruction exception was executed.

The SPSRs are UNKNOWN on a Warm reset. Any operation in a Non-secure EL1 or EL0 mode makes SPSR_hyp unknown.

SPSR bits that are defined as RES0 on an exception taken from AArch32 state are ignored on any exception return to AArch32 state.

For more information on SPSR, see SPSR.

G1.10.2.1 Pseudocode description of SPSR operations

The following pseudocode gives access to the SPSRs.

The `SPSR_ELx []` function accesses the current SPSR and is common to AArch32 and AArch64 operation.

The `SPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update the current SPSR.

The `SPSR_ELx []` and `SPSRWriteByInstr()` functions are defined in A-profile Architecture Pseudocode.

G1.10.3 ELR_hyp

Hyp mode does not provide its own banked copy of LR. Instead, on taking an exception to Hyp mode, the preferred return address is stored in ELR_hyp, a 32-bit Special-purpose register implemented for this purpose.

ELR_hyp can be accessed explicitly only by executing:

- An MRS or MSR instruction that targets ELR_hyp, see:
  - MRS (Banked register).
  - MSR (Banked register).

The ERET instruction uses the value in ELR_hyp as the return address for the exception. For more information, see ERET.

Software execution in any Non-secure EL1 or EL0 mode makes ELR_hyp UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.11 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the PSTATE structure.

The PSTATE fields that are meaningful in AArch32 state are:

The Condition flags

N Negative Condition flag.

Z Zero Condition flag.

C Carry Condition flag.

V Overflow Condition flag.

Process state, PSTATE gives more information about these.

The overflow or saturation flag

Q See Process state, PSTATE.

The greater than or equal flags

GE[3:0] See Process state, PSTATE.

The PE state controls

J, T Instruction set state. See PSTATE. J is RES0. On a Warm reset to AArch32 state, T is set to an IMPLEMENTATION DEFINED value. On taking an exception to:

- A PL1 mode using AArch32, T is set to SCTLR.TE.
- EL2 using AArch32, T is set to HSCTLR.TE.

IT[7:0] IT block state bits. See Process state, PSTATE. On a Warm reset or taking an exception to AArch32 state, these bits are set to 0.

E Endianness of data accesses. See Process state, PSTATE. If an implementation provides both Big-endian and Little-endian support, then:

- On a Warm reset to AArch32 state this bit is set to the IMPLEMENTATION DEFINED reset value of:
  - SCTLR.EE if the highest implemented Exception level is not EL2.
  - HSCTLR.EE if the highest implemented Exception level is EL2.
- On taking an exception to:
  - A PL1 mode using AArch32, this bit is set to SCTLR.EE.
  - EL2 using AArch32, this bit is set to HSCTLR.EE

IL Illegal Execution state bit. See The Illegal Execution state exception. On a Warm reset or taking an exception to AArch32 state, this bit is set to 0.

For information on how the J, T, IT[7:0], E, and IL fields can be accessed, see Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The asynchronous exception mask bits

A SError mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

On a Warm reset to AArch32 state, these bits are set to 1.

On taking an exception to AArch32 state, one or more of these bits are set to 1.

For more information, see both:

- Asynchronous exception masking controls.
- PE state on exception entry.

The mode bits

M[4:0] Current mode of the PE. Table G1-5 lists the permitted values of this field. All other values are reserved. Illegal changes to PSTATE.M describes the effect of setting M[4:0] to a reserved value.

M[4] is:

M[4], Execution state

The current Execution state:

0 AArch64 state.

1 AArch32 state.

Note

This is consistent with the use of M[4:0] in previous versions of the architecture.

On a Warm reset to AArch32 state, M[4:0] is set to:

- `0b10011`, meaning Supervisor mode, if the highest implemented Exception level is not EL2.
- `0b11010`, meaning Hyp mode, if the highest implemented Exception level is EL2.

On taking an exception to AArch32 state, M[4:0] is set to the target mode for the exception type.

For more information about the PE modes, see:

- AArch32 state PE modes.
- PE state on exception entry.

Inject Undefined Instruction exception bit, if FEAT_UINJ is implemented.

UINJ On a Warm reset, this bit is set to 0. On taking an exception, this bit is set to 0.

For information on how this bit can be accessed, see Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit.

Access control bits, from Armv8.1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN *Privileged Access Never* (PAN) state bit, see About the PAN bit.

Timing control bits

DIT *Data Independent Timing* (DIT) bit. For more information, see About the DIT bit.

This bit is implemented only when FEAT_DIT is implemented.

On a Warm reset to AArch32 state, this bit is set to 0.

Speculation control bits

SSBS *Speculative Store Bypass Safe* (SSBS) bit. For more information, see AArch32 Speculative Store Bypass Safe.

This bit is implemented only when FEAT_SSBS is implemented.

On a Warm reset to AArch32 state, this bit is set to an IMPLEMENTATION DEFINED value.

G1.11.1 Accessing PSTATE fields

The PSTATE fields can be accessed as described in the following subsections:

- The Current Program Status Register, CPSR.
- Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit.
- The CPS instruction.
- The SETEND instruction.
- The SETPAN instruction.

G1.11.1.1 The Current Program Status Register, CPSR

Some PSTATE fields can be accessed using the Special-purpose *Current Program Status Register* (CPSR). The CPSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

The CPSR bit assignments are:

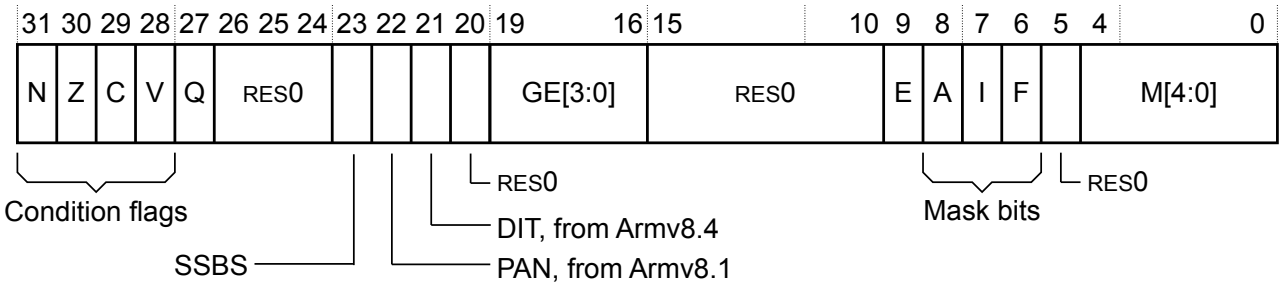

N, Z, C, V, bits [31:28]

The PSTATE Condition flags.

Q, bit [27] The PSTATE overflow or saturation flag.

Bits[26:23, 20, 15:10, 5]

Reserved, RES0.

SSBS, bit [23] Speculative Store Bypass Safe (SSBS) bit, see Access permissions for instruction execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit[22] In Armv8.0, Reserved, RES0.

In Armv8.1, *Privileged Access Never* (PAN) state bit, see About the PAN bit.

DIT, bit [21] Shows the value of CPSR.DIT immediately before the exception was taken.

GE[3:0], bits [19:16]

The PSTATE greater than or equal flags.

E, bit [9] The PSTATE endianness bit.

A, I, F, bits [8:6]

The PSTATE asynchronous exception mask bits.

M[4:0], bits [4:0]

The PSTATE mode bits.

The other PSTATE fields cannot be accessed by using the CPSR. For information on how to access them, see Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit.

The application level alias for the CPSR is the APSR. The APSR is a subset of the CPSR. See The Application Program Status Register, APSR.

Writes to the CPSR have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the CPSR write. This means that they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

The privilege level and address space of memory accesses associated with fetching instructions depend on the current Exception level and Security state. Writes to PSTATE.M can change one or both of the Exception level and Security state. The effect, on memory accesses associated with fetching instructions, of a change of Exception level or Security state is:

- Synchronous to the change of Exception level or Security state, if that change is caused by an exception entry or exception return.
- Guaranteed not to be visible to any memory access caused by fetching an earlier instruction in the execution stream.
- Guaranteed to be visible to any memory access caused by fetching any instruction after the next Context Synchronization event in the execution stream.
- Might or might not affect memory accesses caused by fetching instructions between the mode change instruction and the point where the mode change is guaranteed to be visible.

See Exception return to an Exception level using AArch32 for the definition of exception return instructions.

G1.11.1.2 Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit

The PE state controls are the PSTATE.{IL, IT[7:0], J, E, T} fields. Software can read or write these in an SPSR.

In the CPSR:

- The PE state controls, other than PSTATE.E, are RAZ when read by an MRS instruction.
- Writes to the PE state controls, other than PSTATE.E, by MSR (register) or MSR (immediate), are ignored in all modes.

Instructions other than MRS, MSR (register), or MSR (immediate) that access the PE state controls can read and write them in any mode.

Unlike the other PSTATE PE state controls, PSTATE.E can be read by an MRS instruction and might be written by MSR (register) or MSR (immediate). However, Arm deprecates PSTATE.E having a different value from the equivalent System register EE bit, see Mixed-endian support in AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

To determine the current endianness, software can use an `LDR` instruction to load a word from memory with a known value that differs if the endianness is reversed. For example, using an `LDR` instruction to load a word whose four bytes are `0x01`, `0x00`, `0x00`, and `0x00` in ascending order of memory address loads the destination register with:

- `0x0000_0001` if the current endianness is little-endian.
- `0x0100_0000` if the current endianness is big-endian.

The PSTATE.M[4] bit is the Execution state bit. When read by an MRS instruction in AArch32 state, this bit always reads as 1. When written by an MSR (register) instruction or MSR (immediate) instruction, writing a value other than 1 is an illegal change to the PSTATE.M field. See Illegal changes to PSTATE.M.

The PSTATE.UINJ bit is the Undefined Instruction exception injection bit, implemented when FEAT_UINJ is implemented. Software can read or write this bit in AArch64 state in SPSR_ELx. See Undefined Instruction exception and Undefined Instruction exceptions injected by PSTATE.UINJ.

G1.11.1.3 The `CPS` instruction

The T32 and A32 instruction sets both include an instruction to manipulate PSTATE.{A, I, F} and PSTATE.M:

`CPSIE <iflags> {, #<mode>}` Sets the specified PSTATE. {A, I, F} exception masks to 0, enabling the exception, and optionally changes to the specified mode.

`CPSID <iflags> {, #<mode>}` Sets the specified PSTATE.{A, I, F} exception masks to 1, disabling the exception, and optionally changes to the specified mode.

`CPS #<mode>` Changes to the specified mode without affecting the PSTATE.{A, I, F} exception masks.

The `CPS` instruction is unconditional. For more information, see CPS, CPSID, CPSIE.

G1.11.1.4 The `SETEND` instruction

The T32 and A32 instruction sets both include an instruction to manipulate PSTATE.E:

`SETEND BE` Sets PSTATE.E to 1, for big-endian operation.

`SETEND LE` Sets PSTATE.E to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see SETEND. Arm deprecates use of the `SETEND` instruction.

G1.11.1.5 The `SETPAN` instruction

FEAT_PAN adds the `SETPAN` instruction to the T32 and A32 instruction sets, to manipulate PSTATE.PAN:

`SETPAN #0` Sets PSTATE.PAN to 0, disabling Privileged access-never operation.

`SETPAN #1` Sets PSTATE.PAN to 1, enabling Privileged access-never operation.

The `SETPAN` instruction is unconditional.

- SETPAN.
- About the PAN bit.

G1.11.2 The Saved Program Status Registers (SPSRs)

On taking an exception, PSTATE is preserved in the SPSR of the mode to which the exception is taken. The SPSRs are described in Saved Program Status Registers (SPSRs).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.11.3 Illegal changes to PSTATE.M

In AArch32 PE modes other than User mode, `MSR` and `CPS` instructions can explicitly change PSTATE.M. The following changes to PSTATE.M by `MSR` or `CPS` instructions are illegal:

- A change to an encoding that Table G1-5 does not show.
- A change to a mode that is not implemented.
- A change to a mode that is not accessible from the context the `MRS` or `CPS` instruction is executed in, as follows:
  - A change to a mode that would cause entry to a higher Exception level.
  - When executing in Non-secure state, a change to Monitor mode.
  - When executing in Secure EL1, a change to Monitor mode when EL3 is using AArch64.
  - A change to Hyp mode from any other mode.
  - A change from Hyp mode to any other mode.
  - When the value of HCR.TGE is 1, attempting to change from Monitor mode to a Non-secure PL1 mode, see Trapping of general exceptions to Hyp mode.

On executing an instruction that attempts an illegal change to PSTATE.M:

- PSTATE.M is unchanged, and the current mode remains unchanged.
- PSTATE.IL is set to 1.
- All other PSTATE fields are written to as normal.

Note

For the PSTATE fields that `MSR` and `CPS` instructions update, see the instruction descriptions:

- MSR (register).
- MSR (immediate).
- CPS, CPSID, CPSIE.

When the value of PSTATE.IL is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See The Illegal Execution state exception.

Note

The PE ignores writes to PSTATE.M when executing at PL0.

G1.11.4 Pseudocode description of PSTATE operations

The `CPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update PSTATE.

The `SetPSTATEFromPSR()` function updates PSTATE from a CPSR or SPSR.

A-profile Architecture Pseudocode defines these functions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.12 Instruction set states

The instruction set states are described in The AArch32 Application Level Memory Model and application level operations on them are described there. This section supplies more information about how they interact with system level functionality, in the sections:

- Exceptions and instruction set state.
- Unimplemented instruction sets.

G1.12.1 Exceptions and instruction set state

If an exception is taken to an EL1 mode, the SCTLR.TE bit for the Security state the exception is taken to determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

If the exception is taken to Hyp mode, the HSCTLR.TE bit determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

On coming out of reset, if the highest implemented Exception level is using AArch32:

- If the highest implemented Exception level is EL2, the PE starts execution in Hyp mode, in the instruction set state determined by the reset value of HSCTLR.TE.
- Otherwise, the PE starts execution in Supervisor mode, in the instruction set state determined by the reset value of SCTLR.TE. If the implementation includes EL3, this execution is in Secure Supervisor mode.

For more information about exception entry, see Overview of exception entry.

G1.12.2 Unimplemented instruction sets

The PSTATE.T bit defines the current instruction set state, see Process state, PSTATE.

From the introduction of the Armv8 architecture, there is no support for the hardware acceleration of Java bytecodes, and the Jazelle Instruction set state is obsolete. Every AArch32 implementation must support the Trivial Jazelle implementation described in Trivial implementation of the Jazelle extension.

Note

In previous versions of the Arm architecture, the PSTATE.{J, T} bits determined the Instruction set state. From the introduction of Armv8, PSTATE.J is RES0.

G1.12.2.1 Trivial implementation of the Jazelle extension

The implementation of AArch32 state is required to include the trivial Jazelle implementation.

In a trivial implementation of the Jazelle extension:

- At EL1, EL2, or EL3, if the Exception level is using AArch32:
  - The JMCR and JOSCR are RAZ/WI.
  - The JIDR is a RAZ read-only register.
- At EL0 when EL0 is using AArch32:
  - It is IMPLEMENTATION DEFINED whether the JMCR and JOSCR are RAZ/WI or UNDEFINED.
  - It is IMPLEMENTATION DEFINED whether JIDR is RAZ or UNDEFINED.
- The BXJ instruction behaves identically to the BX instruction in all circumstances.

  Note

  This is consistent with the JMCR.JE bit being RAZ, and means that the T32 and A32 instruction sets do not provide any mechanism for attempting to enter Jazelle state.

- Jazelle state, as defined in previous versions of the Arm architecture, is an unimplemented instruction set state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

These requirements ensure that operating systems that support an EJVM execute correctly.

A trivial implementation is not required to extend the PC to 32 bits, that is, it can implement PC[0] as RAZ/WI.

Note

This is because the only way that PC[0] is visible in A32 or T32 state is as a result of an exception occurring during Jazelle state execution, and Jazelle state execution cannot occur on a trivial implementation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.13 Handling exceptions that are taken to an Exception level using AArch32

An exception causes the PE to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources.

Normally, when an exception is taken the PE state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the PE is handling an exception.

The following sections describe exception handling:

- Exception vectors and the exception base address.
- Exception prioritization for exceptions taken to AArch32 state.
- Overview of exception entry.
- PE mode for taking exceptions.
- PE state on exception entry.
- Routing exceptions from Non-secure EL0 to EL2.
- Routing debug exceptions to EL2 using AArch32.

See also:

- Routing of aborts taken to AArch32 state.
- Exception return to an Exception level using AArch32.
- Asynchronous exception behavior for exceptions taken from AArch32 state.
- AArch32 state exception descriptions.

G1.13.1 Exception vectors and the exception base address

When an exception is taken, PE execution is forced to an address that corresponds to the type of exception. This address is called the *exception vector* for that exception. The vectors for the different types of exception form a *vector table*.

Note

There are significant differences in the sets of exception vectors for exceptions taken to an Exception level that is using AArch32 and for exceptions taken to an Exception level that is using AArch64. This part of this manual describes only how exceptions are taken to an Exception level that is using AArch32.
When an exception is taken to an Exception level that is using AArch64, then the exception is taken as described in The AArch64 System Level Programmers' Model using the exception vectors described in Exception vectors.
AArch32 state defines exception vector tables for exceptions taken to EL2 and EL3 when those Exception levels are using AArch32. Those vector tables are not used when the corresponding Exception levels are using AArch64.

A set of exception vectors for an Exception level that is using AArch32 comprises eight consecutive word-aligned memory addresses, starting at an *exception base address*. These eight vectors form an AArch32 *vector table*.

The number of possible exception base addresses, and therefore the number of vector tables, depends on the implemented Exception levels, as follows:

Implementation that does not include EL3

Any implementation that does not include EL3 must include the following AArch32 vector table if EL1 can use AArch32:

- An exception table for exceptions taken to EL1 modes other than System mode. This is the EL1 vector table, and is in the address space of the PL1&0 translation regime.

  Note

  Exceptions cannot be taken to System mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For this vector table:

— When SCTLR.V == 0, the VBAR holds the exception base address.

— When SCTLR.V == 1, the exception base address is `0xFFFF_0000`.

Implementation that includes EL2

Any implementation that includes EL2 must include the following additional AArch32 vector table if EL2 can use AArch32:

- An exception table for exceptions taken to Hyp mode. This is the Hyp vector table, and is in the address space of the Non-secure PL2 translation regime.
  For this vector table, HVBAR holds the exception base address.

Implementation that includes EL3

Any implementation that includes EL3 must include the following AArch32 vector tables:

- If EL3 can use AArch32, a vector table for exceptions taken to Secure Monitor mode. This is the Monitor vector table, and is in the address space of the Secure PL1&0 translation regime. For this vector table, MVBAR holds the exception base address.
- If Secure EL1 can use AArch32, a vector table for exceptions taken to Secure privileged modes other than Monitor mode and System mode. This is the Secure vector table, and is in the address space of the Secure PL1&0 translation regime.
  — When the Secure instance of SCTLR.V == 0, the Secure instance of VBAR holds the exception base address.
  — When the Secure instance of SCTLR.V == 1, the exception base address is `0xFFFF_0000`.
- If Non-secure EL1 can use AArch32, a vector table for exceptions taken to Non-secure PL1 modes. This is the Non-secure vector table, and is in the address space of the Non-secure PL1&0 translation regime.
  — When the Non-secure instance of SCTLR.V == 0, the Non-secure instance of VBAR holds the exception base address.
  — When the Non-secure instance of SCTLR.V == 1, the exception base address is `0xFFFF_0000`.

The following subsections give more information:

- The vector tables and exception offsets.
- Pseudocode determination of the exception base address.

G1.13.1.1 The vector tables and exception offsets

Table G1-6 defines the AArch32 vector table entries. In this table:

- The *Hyp* column defines the vector table entries for exceptions taken to Hyp mode.
- The *Monitor* column defines the vector table entries for exceptions taken to Monitor mode.
- The *Secure* and *Non-secure* columns define the Secure and Non-secure vector table entries, that are used for exceptions taken to modes other than Monitor mode, Hyp mode, System mode, and User mode. Table G1-7 shows the mode to which each of these exceptions is taken. Each of these modes is described as the *default* mode for taking the corresponding exception.

  Note

  Exceptions cannot be taken to System mode or User mode.

For more information about determining the mode to which an exception is taken, see PE mode for taking exceptions.

When EL2 is using AArch32, it provides a number of additional exceptions, some of which are not shown explicitly in the vector tables. For more information, see Offsets of AArch32 exceptions provided by EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G1-6 The AArch32 vector tables

Offset	Vector tables			
	Hyp[a]	Monitor[b]	Secure[c]	Non-secure[c]
`0x00`	Not used	Not used	Not used[d]	Not used
`0x04`	Undefined Instruction, from Hyp mode	Monitor Trap	Undefined Instruction	Undefined Instruction
`0x08`	Hypervisor Call, from Hyp mode	Secure Monitor Call	Supervisor Call	Supervisor Call
`0x0C`	Prefetch Abort, from Hyp mode	Prefetch Abort	Prefetch Abort	Prefetch Abort
`0x10`	Data Abort, from Hyp mode	Data Abort	Data Abort	Data Abort
`0x14`	Hyp Trap, or Hyp mode entry[e]	Not used	Not used	Not used
`0x18`	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
`0x1C`	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

a. Non-secure state only. Implemented only if the implementation includes EL2 and EL2 can use AArch32.

b. Secure state only. Implemented only if the implementation includes EL3 and EL3 can use AArch32.

c. If the implementation does not include EL3 then there is a single vector table for exceptions taken to EL1 when EL1 is using AArch32. That table holds the vectors shown in the Secure column of this table.

d. In previous versions of the architecture, this entry has been used for the Reset vector, meaning the address at which execution starts on coming out of reset. From the introduction of Armv8, the AArch32 Reset vector is IMPLEMENTATION DEFINED. An implementation might use this vector table entry to hold the Reset vector.

e. See Use of offset `0x14` in the Hyp vector table.

Table G1-7 Modes for taking the exceptions shown in the Secure or Non-secure vector table

Exception	Mode taken to
Undefined Instruction	Undefined
Supervisor Call	Supervisor
Prefetch Abort	Abort
Data Abort	Abort
IRQ interrupt	IRQ
FIQ interrupt	FIQ

For more information about use of the vector tables, see Overview of exception entry.

G1.13.1.1.1 Offsets of AArch32 exceptions provided by EL2

EL2 provides the following exceptions. When EL2 is using AArch32, these exceptions are taken to Hyp mode, and the PE enters the handlers for these exceptions using the following vector table entries shown in Table G1-6:

Hypervisor Call

If taken from Hyp mode, shown explicitly in the Hyp mode vector table. Otherwise, see Use of offset `0x14` in the Hyp vector table.

Hyp Trap Shown explicitly in the Hyp mode vector table.

Virtual Abort Entered through the Data Abort vector in the Non-secure vector table.

Virtual IRQ Entered through the IRQ vector in the Non-secure vector table.

Virtual FIQ Entered through the FIQ vector in the Non-secure vector table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Virtual exceptions when an implementation includes EL2 gives more information about the virtual exceptions.

G1.13.1.1.2 Use of offset `0x14` in the Hyp vector table

The vector at offset `0x14` in the Hyp vector table is used for all exceptions that cause entry to Hyp mode from Non-secure EL0 and EL1, except for IRQ and FIQ exceptions.

Note

Virtual exceptions are never taken to Hyp mode.

G1.13.1.1.3 Pseudocode determination of the exception base address

For an exception taken to a PL1 mode, the `ExcVectorBase()` function determines the exception base address.

The `ExcVectorBase()` function is defined in A-profile Architecture Pseudocode.

Note

The PL1 modes to which exceptions can be taken are Supervisor mode, Undefined mode, Abort mode, IRQ mode, and FIQ mode. In Non-secure state, and in Secure state when EL3 is using AArch64, these are EL1 modes. However, in Secure state when EL3 is using AArch32, these are EL3 modes. For more information, see Security state, Exception levels, and AArch32 execution privilege.

G1.13.2 Exception prioritization for exceptions taken to AArch32 state

The following sections describe the requirements for the prioritization of synchronous exceptions, and the limits on when asynchronous exceptions can be taken:

- Synchronous exception prioritization for exceptions taken to AArch32 state.
- Architectural requirements for taking asynchronous exceptions.

See also:

- AArch32 state prioritization of synchronous aborts from a single stage of address translation, for information about:
  - The prioritization of aborts on a single memory access in a VMSA implementation.
  - The prioritization of exceptions generated during address translation.
- Debug state entry and debug event prioritization for information about the relative prioritization of exceptions and the debug events that cause entry to Debug state.

G1.13.2.1 Synchronous exception prioritization for exceptions taken to AArch32 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. This section describes the prioritization of such exceptions when they are taken to an Exception level that is using AArch32.

Note

- An exception that is taken to an Exception level that is using AArch32 must have been taken from an Exception level that is using AArch32.
- The priority numbering in this list correlates with the equivalent AArch64 list in Prioritization of Synchronous exceptions taken to AArch64 state.

For an exception that is taken to an Exception level that is using AArch32, exceptions are prioritized as follows, where 1 is the highest priority.

1-5 These priority numbers are used by AArch64 exceptions or debug events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6 PC alignment fault exceptions. A PC alignment fault exception can only be taken to an Exception level that is using AArch32 as a result of:

- The CONSTRAINED UNPREDICTABLE handling of a branch to an unaligned address, see Branching to an unaligned PC.
- Exiting from Debug state to AArch32 specifying an unaligned PC value, see Exiting Debug state.

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, see Prefetch Abort exception reporting a PC alignment fault exception.

7 Illegal Execution state exceptions. It is IMPLEMENTATION DEFINED whether this is prioritized here or at 10. See The Illegal Execution state exception.

8 Prefetch Abort exceptions. See Prefetch Abort exception and AArch32 state prioritization of synchronous aborts from a single stage of address translation.

9 Breakpoint exceptions or Address Matching Vector Catch exceptions. See:

- Breakpoint exceptions.
- Vector Catch exceptions.

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in this section. This means that it does not have its own entry in this list.

10 Illegal Execution state exceptions. It is IMPLEMENTATION DEFINED whether this is prioritized here or at 7. See The Illegal Execution state exception.

11 This priority number is used by an AArch64 exception.

12 Software Breakpoint Exceptions caused by the execution of a `BKPT` Exception generating instruction.

13-16 These priority numbers are used by AArch64 exceptions.

17 Exceptions taken from EL1 to EL2 because of one of the following configuration settings:

- HSTR.T*n*.
- HCR.TIDCP.

18 UNDEFINED Instruction exceptions that occur as a result of one or more of the following:

- An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
- An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
- Debug state execution of an instruction encoding that is not accessible in Debug state.
- Non-debug state execution of an instruction encoding that is not accessible in Non-debug state.
- Execution of an `HVC` instruction when `HVC` instructions are disabled by SCR.HCE or HCR.HCD.
- Execution of an `HLT` instruction when `HLT` instructions are disabled by EDSCR.HDE or when halting is prohibited.
- When FEAT_FGT and FEAT_PMUv3 are implemented, executing an `MSR` or `MRS` instruction in AArch64 state, or an MCR or MRC instruction in AArch32 state, that accesses a register associated with an unimplemented event counter.
- In Debug state:
  - Execution of a `DCPS1` instruction in Non-secure EL0 when HCR.TGE is 1.
  - Execution of a `DCPS2` instruction in EL1 or EL0 when SCR.NS is 0 or when EL2 is disabled or not implemented in the current Security state.
  - Execution of a `DCPS3` instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
  - When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Execution of an instruction that is UNDEFINED as a result of any of:
  - Being in an IT block when SCTLR.ITD is 1, or when HSCTLR.ITD is 1.
  - Executing a `SETEND` instruction when SCTLR.SED is 1, or when HSCTLR.SED is 1.
  - Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR.CP15BEN is 0, or when HSCTLR.CP15BEN is 0.
- Execution of an instruction that is UNDEFINED because at least one of FPSCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported. See Floating-point exceptions and exception traps.

19 This priority number is used by AArch64 exceptions.

20 Exceptions taken to EL1, or taken to EL2 because the value of HCR.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 6-18.

21 Exceptions taken from EL0 to EL2 because of one of the following configuration settings:

- HSTR.T*n*.
- HCR.TIDCP.

22 This priority number is used by AArch64 exceptions.

23 Exceptions taken to EL2 because of configuration settings in the HCPTR.

24 Exceptions taken to EL2 because of one of the following configuration settings:

- Any setting in HCR, other than the TIDCP bit.
- Any setting in CNTHCTL.
- Any setting in HDCR.
- If EL1 is using AArch64 state, any of the fine-grained traps in HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGITR_EL2, HFGRTR_EL2, HFGWTR_EL2.

25 Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 6-24.

26 Exceptions caused by the `SMC` instruction being UNDEFINED because the value of SCR.SCD is 1.

27 Exceptions caused by the execution of an Exception generating instruction, `SVC`, `HVC`, or `SMC`.

28-30 These priority numbers are used by AArch64 exceptions.

31 Exceptions taken to EL3 from EL0, EL1, or EL2 because of configuration settings in SDCR.

32 Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 6-31.

33-37 These priority numbers are used by AArch64 exceptions.

38 Trapped floating-point exceptions, if supported. See Floating-point exceptions and exception traps.

39-42 These priority numbers are used by AArch64 exceptions and debug events.

43 In descending priority order:

a) Data Abort exceptions on translation table walks and translation table entry updates.

b) Data Abort exceptions due to synchronous External aborts on translation table walks and translation table entry updates.

c) Data Abort exceptions arising from an MMU fault not on a translation table walk, that are not covered by priorities 44 or 46.

See Data Abort exception and AArch32 state prioritization of synchronous aborts from a single stage of address translation.

44 Data Abort exceptions due to synchronous External aborts on the final physical address of the address translation process.

Whether these are prioritized here or as priority 46 is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See External aborts.

45 Watchpoint exceptions. See Watchpoint exceptions.

46 The exceptions listed for priority 44 if they are prioritized as 46.

47 This priority number is used by AArch64 exceptions.

For priorities 43-46, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

Note

Exceptions generated by a translation table walk are reported and prioritized as either a Prefetch Abort exception, priority 8 in this list, or a Data Abort exception, priority 43 in this list. See also AArch32 state prioritization of synchronous aborts from a single stage of address translation.

G1.13.2.2 Architectural requirements for taking asynchronous exceptions

The Arm architecture does not define when asynchronous exceptions are taken. The prioritization of asynchronous exceptions, including virtual asynchronous exceptions, is IMPLEMENTATION DEFINED.

An asynchronous exception that is pending before a Context Synchronization event in the following list, is taken before the first instruction after the context synchronization event, provided that the pending asynchronous event is not masked:

- Execution of an `ISB` instruction that does not fail its Condition code check.
- Exception entry.
- Exception return.
- Exit from Debug state.

Note

- If the first instruction after the context synchronization event generates a synchronous exception, then the architecture does not define the order in which that synchronous exception and the asynchronous exception are taken.
- The ISR identifies any pending asynchronous exceptions.
- Interrupts are masked when the PE is in Debug state, and therefore this list of context synchronization events does not include the `DCPS` and `DRPS` instructions.

In the absence of a specific requirement to take an asynchronous exception, the only requirement of the architecture is that an unmasked asynchronous exception is taken in finite time.

Note

The taking of an unmasked asynchronous exception in finite time must occur with all code sequences, including with a sequence that consists of unconditional loops.

If an unmasked interrupt was pending but is changed to not pending before it is taken, then the architecture permits the interrupt to be taken, but does not require this to happen. If the interrupt is taken, then it must be taken before the first Context Synchronization event after the interrupt was changed to not pending.

PSTATE includes a mask bit for each type of asynchronous exception. Setting one of these bits to 1 can prevent the corresponding asynchronous exception from being taken, although when the PE is in Non-secure state other controls can modify the effect of these bits. For more information, see Asynchronous exception behavior for exceptions taken from AArch32 state.

Taking an exception sets an exception-dependent subset of these mask bits.

Note

In some contexts, the PSTATE.{A, I, F} bits mask the taking of asynchronous exceptions. The way these are set on exception entry, described in PSTATE.{A, I, F, M} values on exception entry, can prevent an exception handler being interrupted by an asynchronous exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.13.3 Overview of exception entry

There are some significant differences between the handling of exceptions taken to Hyp mode and exceptions taken to other modes. Because Hyp mode is the EL2 mode, this means that the following descriptions sometimes distinguish between the *EL2 mode* and the *non-EL2 modes*.

On taking an exception to an Exception level that is using AArch32:

1. The hardware determines the mode to which the exception must be taken, see PE mode for taking exceptions.
2. A link value, indicating the *preferred return address* for the exception, is saved. This is a possible return address for the exception handler, and depends on:
   - The exception type.
   - Whether the exception is taken to the EL2 mode or to a non-EL2 mode.
   - For some exceptions taken to non-EL2 modes, the instruction set state when the exception was taken.

   Where the link value is saved depends on whether the exception is taken to the EL2 mode.

   For more information, see Link values saved on exception entry.
3. The value of PSTATE is saved in the SPSR for the mode to which the exception must be taken. The value saved in SPSR.IT[7:0] is always correct for the preferred return address.
4. In an implementation that includes EL3, when EL3 is using AArch32:
   - If the exception is taken from Monitor mode, SCR.NS is cleared to 0.
   - Otherwise, taking the exception leaves SCR.NS unchanged.

   When EL3 is using AArch64, Monitor mode is not available.
5. PSTATE is updated with new context information for the exception handler. This includes:
   - Setting PSTATE.M to the PE mode to which the exception is taken.
   - Setting the appropriate PSTATE mask bits. This can disable the corresponding exceptions, preventing uncontrolled nesting of exception handlers.
   - Setting the instruction set state to the state required for exception entry.
   - Setting the endianness to the required value for exception entry.
   - Clearing the PSTATE.IT[7:0] bits to 0.

   For more information, see PE state on exception entry.
6. The appropriate exception vector is loaded into the PC, see Exception vectors and the exception base address.
7. Execution continues from the address held in the PC.

For an exception taken to a non-EL2 mode, on exception entry, the exception handler can use the `SRS` instruction to store the return state onto the stack of any mode at the same Exception level and in the same Security state, and can use the CPS instruction to change mode. For more information about the instructions, see SRS, SRSDA, SRSDB, SRSIA, SRSIB and CPS, CPSID, CPSIE.

Later sections of this chapter describe each of the possible exceptions, and each of these descriptions includes a pseudocode description of the PE state changes on taking that exception. Table G1-8 gives an index to these descriptions:

Table G1-8 Pseudocode descriptions of exception entry for exceptions taken to AArch32 state

Exception	Description of exception entry
Reset	Pseudocode descriptions of reset
Undefined Instruction	Pseudocode description of taking the Undefined Instruction exception
Hyp Trap	Pseudocode description of taking the Hyp Trap exception
Monitor Trap	Pseudocode description of taking the Monitor Trap exception
Supervisor Call	Pseudocode description of taking the Supervisor Call exception
Secure Monitor Call	Pseudocode description of taking the Secure Monitor Call exception

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception	Description of exception entry
Hypervisor Call	Pseudocode description of taking the Hypervisor Call exception
Prefetch Abort	Pseudocode description of taking the Prefetch Abort exception
Data Abort	Pseudocode description of taking the Data Abort exception
Virtual Abort	Pseudocode description of taking the Virtual SError interrupt exception
IRQ	Pseudocode description of taking the physical IRQ exception
Virtual IRQ	Pseudocode description of taking the Virtual IRQ exception
FIQ	Pseudocode description of taking the FIQ exception
Virtual FIQ	Pseudocode description of taking the Virtual FIQ exception

The following sections give more information about the PE state changes, for different architecture implementations. However, you must refer to the pseudocode for a full description of the state changes:

- PE mode for taking exceptions.
- PE state on exception entry.

G1.13.3.1 Link values saved on exception entry

On exception entry, a link value for use on return from the exception, is saved. This link value is based on the *preferred return address for the exception*, as shown in Table G1-9:

Table G1-9 Exception return addresses for exceptions taken to AArch32 state

Exception	Preferred return address	Taken to a mode at
Undefined Instruction	Address of the UNDEFINED instruction	Non-EL2[a], or EL2[c]
Hyp Trap	Address of the trapped instruction	EL2 only[c]
Monitor Trap	Address of the trapped instruction	EL3 only
Supervisor Call	Address of the instruction after the `SVC` instruction	Non-EL2[a] or EL2[c]
Secure Monitor Call	Address of the instruction after the `SMC` instruction	EL3[b], and only in Secure state
Hypervisor Call	Address of the instruction after the `HVC` instruction	EL2 only[c]
Prefetch Abort	Address of aborted instruction fetch	Non-EL2[a] or EL2[c]
Data Abort	Address of instruction that generated the abort	Non-EL2[a] or EL2[c]
Virtual Abort	Address of next instruction to execute	EL1, and only in Non-secure state
IRQ or FIQ	Address of next instruction to execute	Non-EL2[a] or EL2[c]
Virtual IRQ or Virtual FIQ	Address of next instruction to execute	EL1, and only in Non-secure state

a. EL1 if the exception is taken to a Non-secure mode, or is taken to a Secure mode when EL3 is using AArch64. EL3 if the exception is taken to a Secure mode when EL3 is using AArch64.

b. A Secure Monitor Call exception is taken to EL3, and therefore is taken to AArch32 state only if EL3 is using AArch32, in which case it is taken to Monitor mode.

c. EL2 is implemented only in Non-secure state when using AArch32 state. Therefore, an exception can be taken to EL2 mode only if it is taken from Non-secure state when using AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Although Reset is described as an exception, it differs significantly from other exceptions. The architecture has no concept of a return from a Reset and therefore it is not listed in this section.
- For each exception, the preferred return address is not affected by the Exception level from which the exception was taken.

The link value saved, and where it is saved, depend on whether the exception is taken to a non-EL2 mode, or to an EL2 mode, as follows:

Exception taken to a non-EL2 mode

The link value is saved in the LR for the mode to which the exception is taken.

The saved link value is the preferred return address for the exception, plus an offset that depends on the instruction set state when the exception was taken, as Table G1-10 shows:

Table G1-10 Offsets applied to Link value for exceptions taken to non-EL2 modes

Exception	Offset, for PE state of:	
	A32	T32
Undefined Instruction	+4	+2
Monitor Trap	+4	+2
Supervisor Call	None	None
Secure Monitor Call	None	None
Prefetch Abort	+4	+4
Data Abort	+8	+8
Virtual Abort	+8	+8
IRQ or FIQ	+4	+4
Virtual IRQ or Virtual FIQ	+4	+4

Exception taken to an EL2 mode

The link value is saved in the ELR_hyp Special-purpose register.

The saved link value is the preferred return address for the exception, as shown in Table G1-9, with no offset.

G1.13.4 PE mode for taking exceptions

The following principles determine the Exception level to which an exception is taken, and if that Exception level is using AArch32, the PE mode to which the exception is taken:

- An exception cannot be taken to the EL0 mode.
- An exception is taken either:
  - To the Exception level at which the PE was executing when it took the exception.
  - To a higher Exception level.

  This means that, in Secure state:
  - When EL3 is using AArch32, an exception is always taken to an EL3 mode.
  - When EL3 is using AArch64, an exception that is taken to AArch32 state is taken to an EL1 mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Configuration options and other features provided by EL2 and EL3 can determine the mode to which some exceptions are taken, as follows:

  In an implementation that does not include EL2 or EL3

  An exception is always taken to the default mode for that exception.

  In an implementation that includes EL3

  A Secure Monitor Call exception is always taken to EL3. This means:

  — If EL3 is using AArch32 the exception is taken to Secure Monitor mode.

  — If EL3 is using AArch64, then executing the instruction generates an exception that is taken to EL3, see Execution of an SMC instruction from a privileged Exception level that is using AArch32.

  IRQ, FIQ, and External abort exceptions can be configured to be taken to EL3. Therefore, if EL3 is using AArch32 the exceptions are taken to Secure Monitor mode.

  When EL3 is using AArch32, a Monitor Trap exception is taken to Secure Monitor mode.

  Any exception taken from Secure state that is not taken to Secure Monitor mode is taken to Secure state in the default mode for that exception. As described in Security state, Exception levels, and AArch32 execution privilege, this means it is taken to:

  — An EL3 mode other than Monitor mode if EL3 is using AArch32.

  — An EL1 mode if EL3 is using AArch64.

  If the implementation does not include EL2, any exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state to the default mode for that exception. The default mode will be an EL1 mode.

  In an implementation that includes EL2

  An exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state and:

  — If the exception is taken from Hyp mode, then it is taken to Hyp mode.

  — Otherwise, the exception is either taken to Hyp mode, as described in Exceptions taken to Hyp mode, or taken to the default mode for the exception.

  Note

  — Hyp mode is the EL2 mode. The other modes to which an exception can be taken in Non-secure state are EL1 modes.

  — Hyp mode has no effect on the handling of exceptions taken from Secure state.

Table G1-7 shows the default mode to which each exception is taken.

Asynchronous exception routing controls describes the exception routing controls provided by EL2 and EL3.

Routing of aborts taken to AArch32 state gives more information about the modes to which memory aborts are taken.

The possible modes for taking each exception shows all modes to which each exception might be taken, in any implementation. That is, it applies to implementations:

- That include neither EL2 nor EL3.
- That include EL2 but not EL3.
- That do not include EL2 but include EL3.
- That include both EL2 and EL3.

G1.13.4.1 Exceptions taken to Hyp mode

In an implementation that includes EL2 and EL3, when EL2 is using AArch32:

- Any exception taken from Hyp mode that is not routed to EL3 by the controls described in Asynchronous exception routing controls is taken to Hyp mode.
- The following exceptions, if taken from Non-secure state, are taken to Hyp mode:
  — An abort that Routing of aborts taken to AArch32 state identifies as taken to Hyp mode.
  — A Hyp Trap exception, see EL2 configurable controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— A Hypervisor Call exception. This is generated by executing an `HVC` instruction in a Non-secure mode.
— An SError exception, IRQ exception or FIQ exception that is not routed to EL3 but is explicitly routed to Hyp mode, as described in Asynchronous exception routing controls.
— A synchronous External abort, Alignment fault, Undefined Instruction exception, or Supervisor Call exception taken from the Non-secure EL0 mode and explicitly routed to Hyp mode, as described in Routing exceptions from Non-secure EL0 to EL2.

Note

A synchronous External abort can be routed to Hyp mode only if it is not routed to EL3.

— A debug exception that is explicitly routed to Hyp mode, as described in Routing debug exceptions to EL2 using AArch32.

Note

The virtual exceptions cannot be taken to Hyp mode. They are always taken to a Non-secure EL1 mode.

G1.13.4.2 Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64

As described in The Armv8-A security model, when EL3 is using AArch64, lower Exception levels, in either Security state, can be using AArch32. This means software executing in those Exception levels might try to access AArch32 security features that are not available. The following subsections describe the associated behaviors:

- Execution of an SMC instruction from a privileged Exception level that is using AArch32.
- Non-secure reads of the NSACR.
- Secure EL1 operations when Secure EL1 is using AArch32 state.

G1.13.4.2.1 Execution of an SMC instruction from a privileged Exception level that is using AArch32

When EL3 is using AArch64, an `SMC` instruction executed from Secure or Non-secure EL1 using AArch32, or from Non-secure EL2 using AArch32 when the value of HCR.TSC is 0, generates an exception that is taken to EL3. The exception syndrome is reported with an EC value of `0x13`, `SMC` instruction executed in AArch32 state.

G1.13.4.2.2 Non-secure reads of the NSACR

The NSACR is defined as being RO from Non-secure PE modes other than User mode. When EL3 is using AArch64, a read of the NSACR returns a fixed value of `0x0000_0C00` in the following cases:

- If the read is from a Non-secure EL1 mode when EL1 is using AArch32.
- If the read is from Hyp mode when EL2 is using AArch32.

G1.13.4.2.3 Secure EL1 operations when Secure EL1 is using AArch32 state

When Secure EL1 is using AArch32 and if FEAT_SEL2 is implemented and enabled or EL3 is using AArch64:

- Any of the following operations performed in a Secure EL1 mode is trapped to Secure EL3:
  — A read or write of any of the SCR, NSACR, MVBAR, and SDCR.
  — Executing any of the ATS12NSO** instructions.
  — Executing an `SRS` instruction that would use SP_mon, see SRS, SRSDA, SRSDB, SRSIA, SRSIB.
  — Executing an `MRS` (banked register) or `MSR` (banked register) instruction that would access SPSR_mon, SP_mon, or LR_mon, see MRS (Banked register) and MSR (Banked register).
- Any attempt to move into Hypervisor mode, either by an exception return or by executing a `CPS` or `MSR` instruction, is treated as an illegal operation and is handled as described in Illegal return events from AArch32 state.
- Any attempt to move into Monitor mode, either by an exception return or by executing a `CPS` or `MSR` instruction, is treated as an illegal operation and is handled as described in Illegal return events from AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This functionality supports a usage model where:

- EL3 uses AArch64.
- Secure software executed in Secure EL1 using AArch32 and Secure EL0 using AArch32.
- The Non-secure state uses AArch64.

G1.13.4.3 The possible modes for taking each exception

Each of the exception descriptions in AArch32 state exception descriptions includes a subsection that describes the modes to which each exception can be taken. Those subsections are:

- The PE mode to which the Undefined Instruction exception is taken.
- The PE mode to which the Hyp Trap exception is taken.
- The PE mode to which the Monitor Trap exception is taken.
- The PE mode to which the Supervisor Call exception is taken.
- The PE mode to which the Secure Monitor Call exception is taken.
- The PE mode to which the Hypervisor Call exception is taken.
- The PE mode to which the Prefetch Abort exception is taken.
- The PE mode to which the Data Abort exception is taken.
- The PE mode to which the Virtual SError interrupt exception is taken.
- The PE mode to which the physical IRQ exception is taken.
- The PE mode to which the Virtual IRQ exception is taken.
- The PE mode to which the physical FIQ exception is taken.
- The PE mode to which the Virtual FIQ exception is taken.

These descriptions also show the vector offset for the exception entry for each mode. These descriptions assume that all Exception levels are using AArch32, meaning:

- HCR, rather than HCR_EL2, controls the routing of exceptions to EL2.
- SCR, rather than SCR_EL3, controls the routing of exceptions to EL3.

For more information about:

- Vector offsets, see Exception vectors and the exception base address.
- The routing of synchronous External aborts or SError, IRQ, and FIQ interrupt exceptions, and the virtual exceptions, see Asynchronous exception routing controls.

G1.13.4.3.1 UNPREDICTABLE cases when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, exceptions that would otherwise be taken to EL1 are, instead, routed to EL2, see Routing exceptions from Non-secure EL0 to EL2. Related to this, when the value of HCR.TGE is 1, execution in a Non-secure EL1 mode is UNPREDICTABLE. The architecture does not constrain this UNPREDICTABLE behavior, but software that follows the Arm recommendations cannot get to this state. When following the Arm recommendations, any attempt to move to a Non-secure EL1 mode when the value of HCR.TGE is 1 is either:

- An illegal exception return, see Illegal return events from AArch32 state.
- An illegal PE mode change, see Illegal changes to PSTATE.M.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.13.5 PE state on exception entry

The description of each exception includes a pseudocode description of entry to that exception, as Table G1-8 shows. The following sections describe the PE state changes on entering an exception, for different implementations and operating states. However, you must always see the exception entry pseudocode for a full description of the state changes on exception entry:

- Instruction set state on exception entry.
- PSTATE.E value on exception entry.
- PSTATE.{A, I, F, M} values on exception entry.

Note

The descriptions in these sections assume that EL2 and EL3, which control some aspects of the routing of exceptions taken from EL1 or EL0, are both using AArch32. If this is not the case:

- If EL2 is using AArch64:
  - Controls shown as provided by the HSCTLR are provided by the SCTLR_EL2.
  - Controls shown as provided by the HCR are provided by the HCR_EL2.
- If EL3 is using AArch64, controls shown as provided by the SCR are provided by the SCR_EL3.

G1.13.5.1 Instruction set state on exception entry

Exception handlers can execute in either T32 state or A32 state. On exception entry, PSTATE.T is set to the required value, as determined by SCTLR.TE or HSCTLR.TE, depending on the mode the exception is taken to. Table G1-11 shows this:

Table G1-11 PSTATE.T bit value on exception entry

Mode to which exception is taken	HSCTLR.TE	SCTLR.TE	PSTATE.T	Exception handler state
Not Hyp mode	x	0	0	A32
		1	1	T32
Hyp mode	0	x	0	A32
	1	x	1	T32

When an implementation includes EL3 and EL3 is using AArch32, SCTLR is banked for Secure and Non-secure states, and therefore the TE bit value might be different for Secure and Non-secure states. For an exception taken to a PE mode other than Hyp mode, the SCTLR.TE bit for the Security state to which the exception is taken determines the instruction set state for the exception handler. This means the instruction set state in which an exception handler might execute depends on the Security state to which the exception is taken.

G1.13.5.2 PSTATE.E value on exception entry

PSTATE.E controls the load and store endianness for data handling. Table G1-12 show the value to which this bit is set on exception entry:

Table G1-12 PSTATE.E value on exception entry

Exception mode	HSCTLR.EE	SCTLR.EE	Endianness for data loads and stores	PSTATE.E
Secure or Non-secure EL1	x	0	Little-endian	0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception mode	HSCTLR.EE	SCTLR.EE	Endianness for data loads and stores	PSTATE.E
		1	Big-endian	1
Hyp	0	x	Little-endian	0
	1	x	Big-endian	1

For more information, see the bit description in Saved Program Status Registers (SPSRs).

G1.13.5.3 PSTATE.{A, I, F, M} values on exception entry

On exception entry, PSTATE.M is set to the value for the mode to which the exception is taken, as described in PE mode for taking exceptions.

Table G1-13 shows the cases where PSTATE.{A, I, F} bits are set to 1 on an exception entry, and how this depends on the mode and Security state to which an exception is taken. If the table entry for a particular mode and Security state does not define a value for a PSTATE.{A, I, F} bit then that bit is unchanged by the exception entry. In this table:

- The *PE mode exception is taken to* column is the mode to which the exception is taken.
- The *Non-secure* column applies to exceptions taken to Non-secure state in an implementation that includes EL3 but does not include EL2.
- The *Secure* column applies to:
  - Exceptions taken to Secure state.
  - Implementations that do not include the EL3.
  - Exceptions taken to Non-secure state in an implementation that includes EL2.

Table G1-13 PSTATE.{A, I, F} values on exception entry

PE mode exception is taken to	Security state	
	Non-secure	Secure
Hyp	If SCR.EA==0 then PSTATE.A is set to 1 If SCR.IRQ==0 then PSTATE.I is set to 1 If SCR.FIQ==0 then PSTATE.F is set to 1	-
Monitor	-	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1
FIQ	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1
IRQ, Abort	PSTATE.A is set to 1 PSTATE.I is set to 1	PSTATE.A is set to 1 PSTATE.I is set to 1
Undefined, Supervisor	PSTATE.I is set to 1	PSTATE.I is set to 1

Asynchronous exception behavior for exceptions taken from AArch32 state describes how, in some situations, the PSTATE.{A, I, F} bits mask the taking of SError exceptions, IRQ interrupts, and FIQ interrupts.

G1.13.6 Routing exceptions from Non-secure EL0 to EL2

Note

The routing control described in this section permits a Non-secure state usage model where applications execute in User mode under a hypervisor, which executes in Hyp mode, without a Guest OS running at Non-secure EL1. This control applies when the PE is executing in Non-secure EL0 using AArch32 and EL2 is using AArch32 and the value of HCR.TGE is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the PE is in Non-secure User mode, any exception that would otherwise be taken to Non-secure EL1 is taken to EL2 if either:

- EL2 is using AArch32 and the value of HCR.TGE is 1.
  In this case the exception is taken to Hyp mode, instead of to the default Non-secure mode for handling the exception. For more information, see Exception reporting when HCR.TGE routes an exception to EL2 using AArch32.
- EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.
  In this case the exception is taken to EL2 using AArch64, see Exception entry.

Any exception that is routed to Secure Monitor mode or to EL3 using AArch64 is unaffected by the value of HCR.TGE or HCR_EL2.TGE.

When the value of HCR.TGE is 1, meaning TGE routing from Non-secure EL0 using AArch32 to EL2 using AArch32 applies:

- The SCTLR.M bit is treated as 0 for all purposes other than a direct read of the SCTLR register.
- Each of the HCR.{FMO, IMO, AMO} bits is treated as 1 for all purposes other than a direct read of the HCR register.
- Each of the HDCR.{TDE, TDA, TDRA, TDOSA} bits is treated as 1 for all purposes other than a direct read of the HDCR register.
- An exception return to Non-secure EL1 is treated as an illegal exception return, see Illegal return events from AArch32 state.
- All virtual interrupts, including any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts, are disabled.

G1.13.6.1 Exception reporting when HCR.TGE routes an exception to EL2 using AArch32

The following sections give more information about the behavior of synchronous exceptions that are routed to Hyp mode because the value of HCR.TGE is 1:

- Undefined Instruction exception, when the value of HCR.TGE is 1.
- Supervisor Call exception, when the value of HCR.TGE is 1.
- Abort exceptions, when the value of HCR.TGE is 1.
- Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1.

G1.13.6.1.1 Undefined Instruction exception, when the value of HCR.TGE is 1

When HCR.TGE is set to 1, if the PE is executing in Non-secure User mode and attempts to execute an UNDEFINED instruction, it takes the Hyp Trap exception, instead of an Undefined Instruction exception. On taking the Hyp Trap exception, the HSR reports an unknown reason for the exception, using the EC value `0x00`. For more information, see Use of the HSR.

G1.13.6.1.2 Supervisor Call exception, when the value of HCR.TGE is 1

When HCR.TGE is set to 1, if the PE executes an `SVC` instruction in Non-secure User mode, the Supervisor Call exception generated by the instruction is taken to Hyp mode.

The HSR reports that entry to Hyp mode was because of a Supervisor Call exception, and:

- If the `SVC` is unconditional, takes for the `imm16` value in the HSR:
  - A zero-extended 8-bit immediate value for the T32 `SVC` instruction.

    Note

    The only T32 encoding for `SVC` is a 16-bit instruction encoding.

  - The bottom16 bits of the immediate value for the A32 `SVC` instruction.
- If the `SVC` is conditional, the `imm16` value in the HSR is UNKNOWN.

If the `SVC` is conditional, the PE takes the exception only if the instruction passes its Condition code check.

The HSR reports the exception as a Supervisor Call exception taken to Hyp mode, using the EC value `0x11`. For more information, see Use of the HSR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The effect of setting HCR.TGE to 1 is to route the Supervisor Call exception to Hyp mode, not to trap the execution of the `SVC` instruction. This means that the preferred return address for the exception, when routed to Hyp mode in this way, is the instruction after the `SVC` instruction.

G1.13.6.1.3 Abort exceptions, when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, if the PE is executing in Non-secure User mode then any abort exception that is not routed to Secure Monitor mode or to EL3 using AArch64 generates an exception that is taken as a Hyp Trap exception. Where an attempt to execute an instruction causes an abort, on taking the Hyp Trap exception, the HSR indicates whether a Data Abort exception or a Prefetch Abort exception caused the Hyp Trap exception entry, and presents a valid syndrome in the HSR.

When SCR.EA is set to 1, External aborts and SError exceptions are routed to EL3, and this routing takes priority over the HCR.TGE routing. For more information, see Routing of aborts taken to AArch32 state.

An SError exception that is routed to Hyp mode because the value of HCR.TGE is 1 is reported as a Data Abort exception routed to Hyp mode.

The HSR reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information about the exception reporting, see Use of the HSR.

G1.13.6.1.4 Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1

PL1 configurable controls describes controls that, when the value of HCR.TGE is 0, can generate exceptions that are taken from Non-secure EL0 to EL1. When EL2 is using AArch32 and the value of HCR.TGE is 1, the exceptions generated by these controls are routed to Hyp mode. Table G1-14 shows how these exceptions are then reported in the HSR.

Table G1-14 Syndrome reporting in HSR from HCR.TGE routing of traps, disables, and enables

Control provided by PL1	Control type[a]	Syndrome reporting in HSR
SCTLR.{nTWE, nTWI}	T	Uses EC syndrome value `0x00`, Exception for an unknown reason
SCTLR.{SED, ITD}	D	Uses EC syndrome value `0x00`, Exception for an unknown reason
SCTLR.CP15BEN	E	Uses EC syndrome value `0x00`, Exception for an unknown reason
CPACR.TRCDIS	T	Uses EC syndrome value `0x00`, Exception for an unknown reason
CPACR.{cp11, cp10}	E	Uses EC syndrome value `0x00`, Exception for an unknown reason
FPEXC.EN	E	Uses EC syndrome value `0x00`, Exception for an unknown reason
CPACR.ASEDIS	D	Uses EC syndrome value `0x00`, Exception for an unknown reason
DBGDSCRext.UDCCdis	T	Uses EC syndrome value `0x00`, Exception for an unknown reason
CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN}	T	Uses EC syndrome value `0x00`, Exception for an unknown reason
PMUSERENR.{ER, CR, SW, EN}	T	Uses EC syndrome value `0x00`, Exception for an unknown reason

a. T indicates a trap control, E indicates an instruction enable, and D indicates an instruction disable. For the definition of these terms, see the list that begins with Instruction enables and instruction disables.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.13.7 Routing debug exceptions to EL2 using AArch32

When the value of HDCR.TDE is 1, if the PE is executing in a Non-secure mode other than Hyp mode, any Debug exception is routed to Hyp mode. This means it generates a Hyp Trap exception. This applies to:

- Debug exceptions associated with an instruction fetch, that would otherwise generate a Prefetch Abort exception. These are the Breakpoint, Breakpoint Instruction, and Vector Catch exception, see AArch32 Self-hosted Debug.
- Watchpoint exceptions associated with data accesses, that would otherwise generate a Data Abort exception. See Watchpoint exceptions.

When the value of HDCR.TDE is 1, each of the HDCR.{TDRA, TDOSA, TDA} bits is treated as 1 for all purposes other than reading the HDCR register.

Note

- A Breakpoint or Watchpoint debug event that generates entry to Debug state cannot be trapped to Hyp mode. See Breakpoint and Watchpoint debug events.
- When HDCR.TDE is set to 1, the Hyp Trap exception is generated instead of the Prefetch Abort exception or Data Abort exception that is otherwise generated by the Debug exception.
- Debug exceptions, other than Breakpoint Instruction exceptions, are never generated in Hyp mode.

When a Hyp Trap exception is generated because the value of HDCR.TDE is 1, The HSR reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information, see Use of the HSR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.14 Routing of aborts taken to AArch32 state

A memory abort is either a Data Abort exception or a Prefetch Abort exception. When executing in AArch32 state, depending on the cause of the abort, and possibly on configuration settings, an abort is taken either:

- To the Exception level of the PE mode from which the abort is taken. In this case the abort is taken to AArch32 state.
- To a higher Exception level. In this case the Exception level to which the abort is taken is either:
  - Using AArch32. In this case, this chapter describes how the abort is handled.
  - Using AArch64. In this case, The AArch64 Virtual Memory System Architecture describes how the abort is handled.

For an abort taken to an Exception level that is using AArch32, the mode to which a memory abort is taken depends on the reason for the exception, the mode the PE is in when it takes the exception, and configuration settings, as follows:

Memory aborts taken to Monitor mode

If an implementation includes EL3, when the value of SCR.EA is 1, all External aborts are taken to EL3, and if EL3 is using AArch32 they are taken to Monitor mode. This applies to aborts taken from Secure modes and from Non-secure modes.

Memory aborts taken to Secure Abort mode

If an implementation includes EL3, when the PE is executing in Secure state, all memory aborts that are not routed to Monitor mode are taken to Secure Abort mode.

Note

The only memory aborts that can be routed to Monitor mode are External aborts.

Memory aborts taken to Hyp mode

If an implementation includes EL2, when the PE is executing in Non-secure state, the following aborts are taken to EL2. If EL2 is using AArch32 this means they are taken to Hyp mode:

- Alignment faults taken:
  - When the PE is in Hyp mode.
  - When the PE is in a Non-secure PL1 or EL0 mode and the exception is generated because the Non-secure PL1&0 stage 2 translation identifies the target of an unaligned access as any type of Device memory.
  - When the PE is in Non-secure User mode and HCR.TGE is set to 1. For more information, see Abort exceptions, when the value of HCR.TGE is 1.
- When the PE is using the Non-secure PL1&0 translation regime:
  - MMU faults from stage 2 translations, for which the stage 1 translation did not cause an MMU fault.
  - Any abort taken during the stage 2 translation of an address accessed in a stage 1 translation table walk that is not routed to Secure Monitor mode, see Stage 2 fault on a stage 1 translation table walk.
- When the PE is using the Non-secure EL2 translation regime, MMU faults from stage 1 translations.

  Note

  The Non-secure EL2 translation regime has only one stage of translation.

- External aborts, if SCR.EA is set to 0 and any of the following applies:
  - The PE was executing in Hyp mode when it took the exception.
  - The PE was executing in a Non-secure PL1 or EL0 mode when it took the exception, the abort is asynchronous, and HCR.AMO is set to 1. For more information, see Asynchronous exception routing controls.
  - The PE was executing in the Non-secure User mode when it took the exception, the abort is synchronous, and HCR.TGE is set to 1. For more information, see Abort exceptions, when the value of HCR.TGE is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - FEAT_RAS is implemented, the PE was executing in a Non-secure PL1 or EL0 mode when it took the exception, the abort is synchronous, and the value of HCR2.TEA is 1.
  - The abort occurred on a stage 2 translation table walk.
- Debug exceptions, if HDCR.TDE is set to 1. For more information, see Routing debug exceptions to EL2 using AArch32.

Memory aborts taken to Non-secure Abort mode

In an implementation that does not include EL3, all memory aborts that are taken to an Exception level that is using AArch32 are taken to Abort mode.

Otherwise, when the PE is executing in Non-secure state, the following aborts are taken to Non-secure Abort mode:

- When the PE is in a Non-secure PL1 or EL0 mode, Alignment faults taken for any of the following reasons:
  - SCTLR.A is set to 1.
  - An instruction that does not support unaligned accesses is committed for execution, and the instruction accesses an unaligned address.
  - The PL1&0 stage 1 translation identifies the target of an unaligned access as any type of Device memory.

    Note

    In an implementation that does not include EL2, this case results in a CONSTRAINED UNPREDICTABLE memory access, see Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE and Loads and Stores to unaligned locations.

  If an implementation includes EL2 and the PE is in Non-secure User mode, these exceptions are taken to Abort mode only if the value of HCR.TGE is 0.
- When the PE is using the Non-secure PL1&0 translation regime, an MMU fault from a stage 1 translation.
- External aborts, if the PE was executing in a Non-secure PL1 or EL0 mode when it took the exception and both:
  - The value of SCR.EA is 0, meaning the abort is not taken to EL3.
  - The abort is not taken to EL2 for one of the reasons defined in Memory aborts taken to Hyp mode.
- Virtual Aborts, see Virtual exceptions when an implementation includes EL2.
- When the value of HDCR.TDE is 0, Debug exceptions. For more information, see Routing debug exceptions to EL2 using AArch32.

Note

If EL0 is using AArch32 and EL1 is using AArch64, then any of these memory aborts taken from User mode are taken to EL1, as described in The AArch64 Virtual Memory System Architecture.

Memory aborts with IMPLEMENTATION DEFINED behavior

In addition, a PE can generate an abort for an IMPLEMENTATION DEFINED reason associated with lockdown. In an implementation that includes EL2, whether such an abort is taken to Non-secure Abort mode or is taken to EL2 is IMPLEMENTATION DEFINED, and an implementation might include a mechanism to select whether the abort is routed to Non-secure Abort mode or to EL2.

When the PE is in a Non-secure mode other than Hyp mode, if multiple factors cause an Alignment fault, the abort is taken to Non-secure Abort mode if any of the factors require the abort to be taken to Abort mode. For example, if the SCTLR.A bit is set to 1, and the access is an unaligned access to an address that the stage 2 translation tables mark as Device-nGnRnE, then the abort is taken to Non-secure Abort mode.

For more information, see Handling exceptions that are taken to an Exception level using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.15 Exception return to an Exception level using AArch32

In the Arm architecture, *exception return* to an Exception level that is using AArch32 requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception. Typically, exception return involves returning to one of:

- The instruction after the instruction boundary at which an asynchronous exception was taken.
- The instruction following an `SVC`, `SMC`, or `HVC` instruction, for an exception generated by one of those instructions.
- The instruction that caused the exception, after the reason for the exception has been removed.
- The subsequent instruction, if the instruction that caused the exception has been emulated in the exception handler.

The Arm architecture defines a *preferred return address* for each exception other than Reset, see Link values saved on exception entry. The values of the SPSR.IT[7:0] bits generated on exception entry are always correct for this preferred return address, but might require adjustment by the exception handler if returning elsewhere.

In some cases, to calculate the appropriate preferred return address for a return to an Exception level that is using AArch32, a subtraction must be performed on the link value saved on taking the exception. The description of each exception includes any value that must be subtracted from the link value, and other information about the required exception return.

On an exception return, the PSTATE takes either:

- The value loaded by the `RFE` instruction.
- If the exception return is not performed by executing an `RFE` instruction, the value of the current SPSR at the time of the exception return.

If FEAT_MTE is implemented PSTATE.TCO is not updated on Exception return to AArch32 state.

Illegal return events from AArch32 state describes the behavior if the restored PE state would not be valid for the Exception level, PE mode, and Security state targeted by the exception return.

G1.15.1 Exception return instructions

The instructions that an exception handler can use to return from an exception depend on whether the exception was taken to an EL1 mode, or in an EL2 mode, see:

- Return from an exception taken to a PE mode other than Hyp mode.
- Return from an exception taken to Hyp mode.

G1.15.1.1 Return from an exception taken to a PE mode other than Hyp mode

For an exception taken to a PE mode other than Hyp mode, the Arm AArch32 architecture provides the following *exception return instructions*:

- From privileged modes other than System mode, the `ERET` instruction. After the exception return, execution resumes from the address held in the LR (R14) for the mode in which `ERET` is executed. See ERET.
- Data-processing instructions with the S bit set and the PC as a destination, see MOV, MOVS (register) and SUB, SUBS (immediate).

Note

The A32 instruction set includes other instructions that can be used for an exception return, but Arm deprecates any use of those instructions.

Typically:

— A return where no subtraction is required uses `SUBS` with an operand of 0, or the equivalent `MOVS` instruction.

— A return requiring subtraction uses `SUBS` with a nonzero operand.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `RFE` instruction, see RFE, RFEDA, RFEDB, RFEIA, RFEIB. If a subtraction is required, typically it is performed before saving the LR value to memory. After the exception return, execution resumes from the address held in the memory location indicated by the base register specified by the `RFE` instruction.
- In A32 state, a form of the `LDM` instruction in which the PC is one of the registers loaded, see LDM (exception return). If a subtraction is required, typically it is performed before saving the LR value to memory.

G1.15.1.2 Return from an exception taken to Hyp mode

For an exception taken to Hyp mode, the Arm architecture provides the `ERET` instruction, see ERET. An exception handler executing in Hyp mode must return using the `ERET` instruction.

Hyp mode is implemented only as part of EL2.

G1.15.2 Alignment of exception returns

The T bit of the value transferred to the PSTATE by an exception return controls the target instruction set of that return. The behavior of the hardware for exception returns for different values of the T bit is as follows:

T == 0 The target instruction set state is A32 state. Bits[1:0] of the address transferred to the PC are ignored by the hardware.

T == 1 The target instruction set state is T32 state:

- Bit[0] of the address transferred to the PC is ignored by the hardware.
- Bit[1] of the address transferred to the PC is part of the instruction address.

Note

In previous versions of the Arm architecture, the PSTATE.{J, T} bits determined the Instruction set state. From the introduction of Armv8, PSTATE.J is RES0.

Arm deprecates any dependence on the requirements that the hardware ignores bits of the address. Arm recommends that the address transferred to the PC for an exception return is correctly aligned for the target instruction set.

After an exception entry other than Reset, the LR value has the correct alignment for the instruction set indicated by the SPSR.T bit. This means that if exception return instructions are used with the LR and SPSR values produced by such an exception entry, the only precaution software needs to take to ensure correct alignment is that any subtraction is of a multiple of four if returning to A32 state, or a multiple of two if returning to T32 state.

G1.15.3 Illegal return events from AArch32 state

Throughout this section:

Return In AArch32 state, refers to any of:

- Execution of any exception return instruction.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

If an exception or debug return from an Exception level using AArch32 triggers an illegal exception return, then bit[1] of the PC is either:

- Zero.
- The value of bit[1] of the return address for the exception or debug return.

The choice between these two alternatives is made by the implementation, and might differ from instance to instance of an illegal exception return.

Note

This means software must support both alternatives.

Saved process state value In AArch32 state, refers to any of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The value held in the SPSR for any exception return other than an exception return made by executing an `RFE` instruction.
- The value read from memory that is to be restored to PSTATE by the execution of an `RFE` instruction.
- The value held in the SPSR for the execution of a `DRPS` instruction in Debug state.
- The value held in the DSPSR for a Debug state exit.

Link address In AArch32 state, refers to any of:

- The address held in the link register for any exception return other than an exception return made by executing an `ERET`, `LDM`, or `RFE` instruction.
- The address held in ELR_hyp for any exception return made by executing an `ERET` instruction.
- The address read from memory that is to be restored to the PC by the execution of an `LDM` or `RFE` instruction.
- The address held in the DLR for Debug state exit.

Configured from reset

Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

The architecture has a generic mechanism for handling exception or debug returns to a mode or state that is illegal. In AArch32 state, this can occur as a result of any of the following situations:

- A return where the Exception level being returned to is higher than the current Exception level.
- A return where the mode being returned to is not implemented. For example:
  - A return to Hyp mode when EL2 is not implemented.
  - A return to Monitor mode, when EL3 is either not implemented or using AArch64 state.
- A return to EL2 when:
  - EL3 is implemented and using AArch64, and the values of SCR_EL3.{NS, EEL2} 0.
  - EL3 is implemented and using AArch32, and the value of the SCR.NS bit is 0.
- A return to Non-secure EL1 when:
  - EL2 is implemented and using AArch64, and the value of the HCR_EL2.TGE bit is 1.
  - EL2 is implemented and using AArch32, and the value of the HCR.TGE bit is 1.
- A return where the value of the saved process state M[4:0] field is not a valid AArch32 PE mode for the implementation. Table G1-5 shows the valid M[4:0] values for AArch32 PE modes.

In these cases:

- PSTATE.IL is set to 1, to indicate an illegal return.
- PSTATE.M is unchanged. This means the PE mode does not change.
- The SS bit is handled in the same way as any other exception or debug return, see Software Step exceptions.
- The following PSTATE bits are restored from the saved process state value:
  - The N, Z, C, V Condition flags.
  - The Q Overflow or saturation flag.
  - The GE Greater than or Equal flags.
  - The E Endianness mapping bit.
  - The A, I, F exception mask bits.
  - The DIT Data Independent Timing bit.
- The PSTATE.{IT, T} bits are each either:
  - Set to 0.
  - Copied from the saved process state in the SPSR for the PE mode in which the exception is handled.

  The choice between these two options is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PC is restored from the link address, unless the illegal return is the execution of a `DRPS` instruction in Debug state.

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See The Illegal Execution state exception.

All aspects of the illegal return, other than the effects described in this section, are the same as for a legal return.

G1.15.4 Legal returns that set PSTATE.IL to 1

In this section, return, saved process state value, and link address have the meaning that is defined in Illegal return events from AArch32 state.

If the IL bit in the saved process state value is 1, then it is copied to PSTATE meaning that PSTATE.IL is set to 1. In this case, the PSTATE.{IT, T} bits are each either:

- Set to 0.
- Copied from the SPSR, or loaded from memory if the exception return was performed by executing an `RFE` instruction.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. This means software must regard each value as being an UNKNOWN choice between the two permitted values.

Because the return sets the PSTATE.IL bit to 1, any attempt to execute any instruction results in an Illegal Execution state exception. See The Illegal Execution state exception.

G1.15.5 The Illegal Execution state exception

When the value of the PSTATE.IL bit is 1, any attempt to execute an instruction generates an Illegal Execution state exception. In AArch32 state, the PSTATE.IL bit can be set to 1 by one of the following:

- An illegal return, as described in Illegal return events from AArch32 state.
- An illegal change to PSTATE.M, as described in Illegal changes to PSTATE.M.
- A legal return that sets PSTATE.IL to 1, as described in Legal returns that set PSTATE.IL to 1.

An Illegal Execution state exception is taken in the same way as an Undefined Instruction exception in the current Exception level. If the current Exception level is EL2 using AArch32 state, the HSR provides additional syndrome information for the exception, see Use of the HSR.

An Illegal Execution state exception has priority over any other Undefined Instruction exception that might arise from instruction execution.

Note

This section only describes the handling of an Illegal Execution state exception that is taken to an Exception level that is using AArch32 state.

On taking any exception to an Exception level that is using AArch32 state:

1. The value of the PSTATE.IL bit is 1 and this is copied to the SPSR.IL bit for the PE mode to which the exception is taken.
2. The PSTATE.IL bit is cleared to 0.

Note

This means that it is not possible for software to observe the value of PSTATE.IL.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.15.5.1 Pseudocode description of exception return

The `AArch32_ExceptionReturn()` function transfers the return address to the PC and restores PSTATE to its saved value.

This function uses the function `SetPSTATEFromPSR()`.

The `IllegalExceptionReturn()` function checks for an Illegal Execution state exception.

A-profile Architecture Pseudocode includes the definitions of these functions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.16 Asynchronous exception behavior for exceptions taken from AArch32 state

In an implementation that does not include EL2 or EL3, the asynchronous exceptions behave as follows when EL1 and EL0 are both using AArch32:

- An SError exception is taken to Abort mode.
- An IRQ exception is taken to IRQ mode.
- An FIQ exception is taken to FIQ mode.

These are the *default PE modes* for taking these exceptions.

However, the PSTATE.{A, I, F} bits *mask* the asynchronous exceptions, meaning that when the value of one of these PSTATE bits is 1, the corresponding exception is not taken.

If a masked asynchronous exception remains signaled, then the exception remains pending unless the value of the PSTATE bit is changed to 0.

EL2 and EL3 provide controls that affect:

- The routing of these exceptions, see Asynchronous exception routing controls.
- Masking of these exceptions in Non-secure state, see Asynchronous exception masking controls.

Similar register control bits are provided regardless of whether EL2 and EL3 are using AArch32 or AArch64:

- The EL2 controls are provided by the HCR when EL2 is using AArch32, and by the HCR_EL2 when EL2 is using AArch64.
- The EL3 controls are provided by the SCR when EL3 is using AArch32, and by the SCR_EL3 when EL3 is using AArch64.

Therefore, most references to the HCR or SCR in this section are to entries in Table K14-1, which disambiguates between AArch32 registers and AArch64 registers. However, the Execution states used by EL2 and EL3 do affect some aspects of the routing and masking of the asynchronous exceptions, see Asynchronous exception routing and masking with higher Exception levels using AArch64.

G1.16.1 Virtual exceptions when an implementation includes EL2

When implemented, EL2 provides the following virtual exceptions, which correspond to the physical asynchronous exceptions:

- Virtual SError, which corresponds to a physical external SError exception.
- Virtual IRQ, which corresponds to a physical IRQ.
- Virtual FIQ, which corresponds to a physical FIQ.

When the value of HCR.TGE is 0 and the value of an HCR.{AMO, IMO, FMO} routing control bit is 1, the corresponding virtual interrupt is enabled and a virtual exception is generated either:

- By setting the corresponding virtual interrupt pending bit, HCR.{VA, VI, VF}, to 1.
- For a Virtual IRQ or Virtual FIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the *ARM Generic Interrupt Controller Architecture Specification*.

When the value of HCR_EL2.TGE is 1, all virtual interrupts are disabled.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In AArch32 state, a virtual exception is taken only from a Non-secure EL1 or EL0 mode. In any other mode, if the exception is generated it is not taken.

A virtual exception is taken in Non-secure state to the default mode for the corresponding physical exception. This means:

- A Virtual SError is taken to Non-secure Abort mode.
- A Virtual IRQ is taken to Non-secure IRQ mode.
- A Virtual FIQ is taken to Non-secure FIQ mode.

Table G1-15 summarizes the HCR bits that route asynchronous exceptions to EL2, and the bits that generate the virtual exceptions.

Table G1-15 HCR bits controlling asynchronous exceptions

Exception	Routing the physical exception to EL2	Generating the virtual exception
SError	HCR.AMO	HCR.VA
IRQ	HCR.IMO	HCR.VI
FIQ	HCR.FMO	HCR.VF

The HCR.{VA, VI, VF} bits generate a virtual exception only if set to 1 when the value of the corresponding HCR.{AMO, IMO, FMO} is 1.

Similarly, if the implementation also includes EL3, the HCR.{AMO, IMO, FMO} bits route the corresponding physical exception to Hyp mode only if the physical exception is not routed to Monitor mode by the SCR.{EA, IRQ, FIQ} bit. For more information, see Asynchronous exception routing controls.

When the value of an HCR.{AMO, IMO, FMO} control bit is 1, the corresponding mask bit in PSTATE:

- Does not mask the physical exception.
- Masks the virtual exception when the PE is executing in a Non-secure EL1 or EL0 mode.

Taking a Virtual Abort exception clears HCR.VA to zero. Taking a Virtual IRQ exception or a Virtual FIQ exception does not affect the value of HCR.VI or HCR.VF.

Note

This means that the exception handler for a Virtual IRQ exception or a Virtual FIQ exception must cause software that is executing at EL2 or EL3 to update the HCR to clear the appropriate virtual exception bit to 0.

See WFE wakeup events and Wait For Interrupt for information about how virtual exceptions affect wake up from power-saving states.

Note

A hypervisor can use virtual exceptions to signal exceptions to the current Guest OS. The Guest OS takes a virtual exception exactly as it would take the corresponding physical exception, and is unaware of any distinction between virtual exception and the corresponding physical exception.

G1.16.1.1 Effects of the HCR.{AMO, IMO, FMO} bits

As described in this section, the HCR.{AMO, IMO, FMO} bits are part of the mechanism for enabling the virtual exceptions. In addition, for exceptions generated in Non-secure state:

- As mentioned in this section, affect the routing of the exceptions. See Asynchronous exception routing controls.
- Affect the masking of the exceptions. See Asynchronous exception masking controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.16.2 Asynchronous exception routing controls

Note

This section describes the behavior when all Exception levels are using AArch32. For the differences when this is not the case see Asynchronous exception routing and masking with higher Exception levels using AArch64.

In an implementation that includes EL3 the following bits in the SCR control the routing of asynchronous exceptions:

SCR.EA When the value of this bit is 1, any SError exception is taken to Monitor mode.

Note

Although this section describes the asynchronous exception routing controls, SCR.EA also controls the routing of synchronous External aborts, see Routing of aborts taken to AArch32 state.

SCR.FIQ When the value of this bit is 1, any FIQ exception is taken to Monitor mode.

SCR.IRQ When the value of this bit is 1, any IRQ exception is taken to Monitor mode.

Only Secure software can change the values of these bits.

In an implementation that includes EL2, the following bits in the HCR route asynchronous exceptions to EL2, for exceptions that are both:

- Taken from a Non-secure EL1 or EL0 mode.
- If the implementation also includes EL3, not configured, by the SCR.{EA, FIQ, IRQ} controls, to be taken to EL3.

HCR.AMO When the value of this bit is 1, an SError exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure Abort mode. If the implementation also includes EL3, this control applies only if the value of SCR.EA is 0. When the value of SCR.EA is 1, the value of the AMO bit is ignored.

HCR.FMO When the value of this bit is 1, an FIQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure FIQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.FIQ is 0. When the value of SCR.FIQ is 1, the value of the FMO bit is ignored.

HCR.IMO When the value of this bit is 1, an IRQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure IRQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.IRQ is 0. When the value of SCR.IRQ is 1, the value of the IMO bit is ignored.

When EL2 is using AArch32 and the value of one of the HCR.{AMO, FMO, IMO} bits is 1, the exception is taken to Hyp mode.

Only software executing in Hyp mode, or Secure software executing at EL3 with SCR.NS set to 1, can change the values of these bits. If EL3 is using AArch32, this requires the Secure software to be executing in Monitor mode.

The HCR.{AMO, FMO, IMO} bits also affect the masking of asynchronous exceptions in Non-secure state, as described in Asynchronous exception masking controls.

The SCR.{EA, FIQ, IRQ} and HCR.{AMO, FMO, IMO} bits have no effect on the routing of Virtual Abort, Virtual FIQ, and Virtual IRQ exceptions.

Note

When the PE is in Hyp mode:

- Physical asynchronous exceptions that are not routed to Monitor mode are taken to Hyp mode.
- Virtual exceptions are not signaled to the PE.

See also Asynchronous exception behavior for exceptions taken from AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.16.3 Asynchronous exception masking controls

Note

This section describes the behavior when all Exception levels are using AArch32. For the differences when this is not the case see Asynchronous exception routing and masking with higher Exception levels using AArch64.

The PSTATE.{A, I, F} bits can mask the taking of the corresponding exceptions from AArch32 state, as follows:

- PSTATE.A can mask SError exceptions.
- PSTATE.I can mask IRQ exceptions.
- PSTATE.F can mask FIQ exceptions.

In an implementation that does not include either of EL2 and EL3, setting one of these bits to 1 masks the corresponding exception, meaning the exception cannot be taken.

In an implementation that includes EL2, the HCR.{AMO, IMO, FMO} bits modify the masking of exceptions taken from Non-secure state.

Similarly, in an implementation that includes EL3, the SCR.{AW, FW} bits modify the masking of exceptions taken from Non-secure state by the PSTATE.{A, F} bits.

An implementation that includes only EL1 and EL0 does not provide any masking of the PSTATE.{A, I, F} bits. The following subsections describe the masking of these bits in other implementations:

- Asynchronous exception masking in an implementation that includes EL2 but not EL3.
- Asynchronous exception masking in an implementation that includes EL3 but not EL2.
- Asynchronous exception masking in an implementation that includes both EL2 and EL3.
- Summary of the asynchronous exception masking controls.

G1.16.3.1 Asynchronous exception masking in an implementation that includes EL2 but not EL3

The HCR.{AMO, IMO, FMO} bits modify the effect of the PSTATE.{A, I, F} bits. When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when the exception is taken from a Non-secure mode other than Hyp mode.

G1.16.3.2 Asynchronous exception masking in an implementation that includes EL3 but not EL2

The SCR.{AW, FW} bits modify the effect of the PSTATE.{A, F} bits. When the value of one of the SCR.{AW, FW} bits is 0, the corresponding PSTATE bit is ignored when both of the follow apply:

- The corresponding exception is taken from Non-secure state.
- The value of the corresponding SCR.{EA, FIQ} bit is 1, routing the exception to EL3. This means the exception is routed to Monitor mode if EL3 is using AArch32.

Note

Whenever the value of PSTATE.I is 1, IRQ exceptions are masked and cannot be taken.

G1.16.3.3 Asynchronous exception masking in an implementation that includes both EL2 and EL3

When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when both of the following apply:

- The exception is taken from Non-secure state.
- Either:
  - The corresponding SCR.{EA, IRQ, FIQ} bit routes the exception to Monitor mode.
  - The exception is taken from a Non-secure mode other than Hyp mode.

In addition, when the value of an SCR.{AW, FW} bit is 0, the value of the corresponding PSTATE.{A, F} bit is ignored when all of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The exception is taken from Non-secure state.
- The corresponding SCR.{EA, FIQ} bit routes the exception to Monitor mode.
- The corresponding HCR.{AMO, FMO} mask override bit is set to 0.

G1.16.3.4 Summary of the asynchronous exception masking controls

The tables in this section show the masking controls for each of the PSTATE.{A, I, F} bits. For an implementation that does not include all of the Exception levels:

If the implementation includes only EL1 and EL0

The PSTATE bits cannot be masked. The behavior is as shown in the *Secure* row of the tables.

If the implementation includes EL2 but not EL3

The behavior is as shown in the *Non-secure* table rows when the control bits in the SCR are both 0.

If the implementation includes EL3 but not EL2

The behavior is as shown in the table rows where the control bit in the HCR is 0.

Table G1-16 shows the controls of the masking of SError exceptions by PSTATE.A.

Table G1-16 Control of masking by PSTATE.A

Security state	HCR.AMO	SCR.EA	SCR.AW	Mode	PSTATE.A
Secure	x	x	x	x	Masks SError exception, when set to 1
Non-secure	0	0	x	x	Masks SError exception, when set to 1
		1	0	x	Ignored
			1	x	Masks SError exception, when set to 1
	1	x	x	Not Hyp	Ignored
		0	x	Hyp	Masks SError exception, when set to 1
		1	x	x	Ignored

Table G1-17 shows the controls of the masking of IRQ exceptions by PSTATE.I.

Table G1-17 Control of masking by PSTATE.I

Security state	HCR.IMO	SCR.IRQ	Mode	PSTATE.I
Secure	x	x	x	Masks IRQs, when set to 1
Non-secure	0	x	x	Masks IRQs, when set to 1
	1	x	Not Hyp	Ignored
		0	Hyp	Masks IRQs, when set to 1
		1	x	Ignored

Table G1-18 shows the controls of the masking of FIQ exceptions by PSTATE.F.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G1-18 Control of masking by PSTATE.F

Security state	HCR.FMO	SCR.FIQ	SCR.FW	Mode	PSTATE.F
Secure	x	x	x	x	Masks FIQs, when set to 1
Non-secure	0	0	x	x	Masks FIQs, when set to 1
		1	0	x	Ignored
			1	x	Masks FIQs, when set to 1
	1	x	x	Not Hyp	Ignored
		0	x	Hyp	Masks FIQs, when set to 1
		1	x	x	Ignored

G1.16.4 Asynchronous exception routing and masking with higher Exception levels using AArch64

Asynchronous exception routing controls and Asynchronous exception masking controls give full descriptions of the routing and masking of the asynchronous exceptions when all Exception levels are using AArch32. However, when EL0 and EL1 are using AArch32:

- As already described, the SCR and HCR controls might be from Exception levels that are using AArch64.
- If EL3 is using AArch64, or EL2 is using AArch64, there are some changes to the asynchronous exception behaviors.

Therefore, the following sections summarize the asynchronous exception behaviors, taking account of the Execution state being used at EL2 and EL3:

- Summary of physical interrupt routing.
- Summary of physical interrupt masking.

G1.16.4.1 Summary of physical interrupt routing

The Table G1-19 shows the routing of physical FIQ and IRQ interrupts, and physical SError exceptions when the highest Exception level is using AArch32.

In this table:

NS The Effective value of the field that determines the Security state of the PE in SCR.

FIQ IRQ EA The Effective value of the field that handles the asynchronous exception type in SCR.

TGE The Effective value of the field that handles general exceptions in HCR if EL2 is using AArch32, or HCR_EL2 if EL2 is using AArch64.

FMO IMO AMO The Effective value of the mask override field for the asynchronous exception type in HCR if EL2 is using AArch32, or HCR_EL2 if EL2 is using AArch64.

FIQ IRQ Abt The exception is taken to the FIQ mode, the IRQ mode or the Abort mode according to the type of asynchronous exception.

Hyp The exception is taken to AArch32 Hyp mode.

Mon The exception is taken to AArch32 Monitor mode.

n/a This field does not exist, or the Exception level is not accessible in this configuration.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G1-19 Routing of physical asynchronous exceptions

Control bits				Target when taken from			
NS	FIQ IRQ EA	TGE	FMO IMO AMO	EL0	EL1	EL2	EL3
0	0	x	x	FIQ IRQ Abt	n/a	n/a	FIQ IRQ Abt
	1	x	x	Mon	n/a	n/a	Mon
1	0	0	0	FIQ IRQ Abt	FIQ IRQ Abt	Hyp	FIQ IRQ Abt
		0	1	Hyp	Hyp	Hyp	FIQ IRQ Abt
		1	x	Hyp	n/a	Hyp	FIQ IRQ Abt
	1	0	x	Mon	Mon	Mon	Mon
		1	x	Mon	n/a	Mon	Mon

G1.16.4.2 Summary of physical interrupt masking

Table G1-20 shows the masking of physical FIQ and IRQ interrupts, and physical SError exceptions when the highest Exception level is using AArch32.

In this table:

NS The Effective value of the field that determines the Security state of the PE in SCR.

FIQ IRQ EA The Effective value of the field that handles the asynchronous exception type in SCR.

TGE The Effective value of the field that handles general exceptions in HCR if EL2 is using AArch32, or HCR_EL2 if EL2 is using AArch64.

FMO IMO AMO The Effective value of the mask override field for the asynchronous exception type in HCR.

FW AW For FIQ interrupts, the SCR.FW field, and for SError exceptions, the SCR.AW field. For IRQ interrupts, there is no equivalent field, so the Effective value is 0 and rows where this cell is 1 should be ignored.

A When the interrupt is asserted, it is taken regardless of the value of the PSTATE mask bit.

B When the interrupt is asserted, it is subject to the corresponding PSTATE mask bit. If the value of the mask is 1, the interrupt is not taken. If the value of the mask is 0, the interrupt is taken.

n/a This field does not exist, or the Exception level is not accessible in this configuration.

Table G1-20 Masking of physical asynchronous exceptions

Control bits					Effect of the interrupt mask when executing at:			
NS	FW AW	FIQ IRQ EA	TGE	FMO IMO AMO	EL0	EL1	EL2	EL3
0	x	x	x	x	B	n/a	n/a	B
1	x	0	0	0	B	B	B	B
				1	A	A	B	B
			1	x	A	n/a	B	B
	0	1	0	x	A	A	A	B
			1	x	A	n/a	A	B
	1	1	0	0	B	B	B	B
				1	A	A	A	B

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Control bits					Effect of the interrupt mask when executing at:			
NS	FW AW	FIQ IRQ EA	TGE	FMO IMO AMO	EL0	EL1	EL2	EL3
			1	x	A	n/a	A	B

G1.16.5 Taking an interrupt or other exception during a multiple-register load or store

In AArch32 state, an interrupt cannot be taken during a sequence of memory accesses caused by a single load or store instruction, except that when FEAT_LSMAOC is implemented and the value of the applicable LSMAOE field is 0, an interrupt can be taken between two memory accesses made by a single AArch32 Load Multiple (`LDM`) or Store Multiple (`STM`) instruction.

The applicable LSMAOE field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the `LDM` or `STM` instruction is executed.

When the value of the LSMAOE bit is 0 and an interrupt is taken between two memory accesses made by a single AArch32 `LDM` or `STM` instruction, then:

- For a load, any register being loaded by the instruction other than a register used in the generation of the address by the instruction or the PC, can contain an UNKNOWN value. Any register used in the generation of the address is restored to its initial value and the LR is set on the interrupt to a value consistent with returning to the instruction.
- For a store, any data location being stored to by the instruction can contain an UNKNOWN value.
- For either a load or store, if the instruction specifies writeback of the base address, then that register is restored to its initial value.

Armv8.2 deprecates software relying on interrupts not being taken during the sequence of memory accesses caused by a single load or store instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.17 AArch32 state exception descriptions

Handling exceptions that are taken to an Exception level using AArch32 gives general information about exception handling. This section describes each of the exceptions, in the following subsections:

- Undefined Instruction exception.
- Monitor Trap exception.
- Hyp Trap exception.
- Supervisor Call (SVC) exception.
- Secure Monitor Call (SMC) exception.
- Hypervisor Call (HVC) exception.
- Prefetch Abort exception.
- Data Abort exception.
- Virtual SError interrupt exception.
- IRQ exception.
- Virtual IRQ exception.
- FIQ exception.
- Virtual FIQ exception.

Additional pseudocode functions for exception handling gives additional pseudocode that is used in the pseudocode descriptions of a number of the exceptions.

G1.17.1 Undefined Instruction exception

An Undefined Instruction exception might be caused by:

- A System register access, floating-point, or Advanced SIMD instruction that is not accessible because of the settings in one or more of the CPACR, NSACR, HCPTR, and DBGDSCRext.
- A System register access, floating-point, or Advanced SIMD instruction that is not implemented.
- A System register access, floating-point, or Advanced SIMD instruction that causes an exception during execution. This includes:
  - Trapped floating-point exceptions that are taken to AArch32, if an implementation supports these traps. See Floating-point exceptions and exception traps.
  - Execution of certain floating-point instructions when one or both of the FPSCR.{Stride, Len} fields in nonzero, in an implementation in which those fields are RW. The description of FPEXC specifies the instructions to which this applies.
- If FEAT_UINJ is implemented, an attempt at EL0 using AArch32 to execute any instruction when PSTATE.UINJ is 1. See Undefined Instruction exceptions injected by PSTATE.UINJ.
- An instruction that is UNDEFINED.

Note

The Undefined Instruction exception is taken using offset `0x04` in the Hyp, Secure, or Non-secure vector table. In the Monitor vector table this offset is used for the Monitor Trap exception. See Monitor Trap exception and The vector tables and exception offsets.

By default, an Undefined Instruction exception is taken to Undefined mode, but an Undefined Instruction exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see The PE mode to which the Undefined Instruction exception is taken.

The Undefined Instruction exception can provide:

- Signaling of an illegal instruction execution.
- *Lazy context switching* of System registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The preferred return address for an Undefined Instruction exception is the address of the instruction that generated the exception. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Secure or Non-secure Undefined mode, the exception return uses the SPSR and LR_und values generated by the exception entry, as follows:
  - If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
  - If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.
- If returning from Hyp mode, the exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see Exception return to an Exception level using AArch32.

Note

If handling the Undefined Instruction exception requires instruction emulation, followed by return to the next instruction after the instruction that caused the exception, the instruction emulator must use the instruction length to calculate the correct return address, and to calculate the updated values of the IT bits if necessary.

G1.17.1.1 The PE mode to which the Undefined Instruction exception is taken

Figure G1-4 shows how the implementation, state, and configuration options determine the PE mode to which an Undefined Instruction exception is taken, when the exception is taken to an Exception level that is using AArch32.

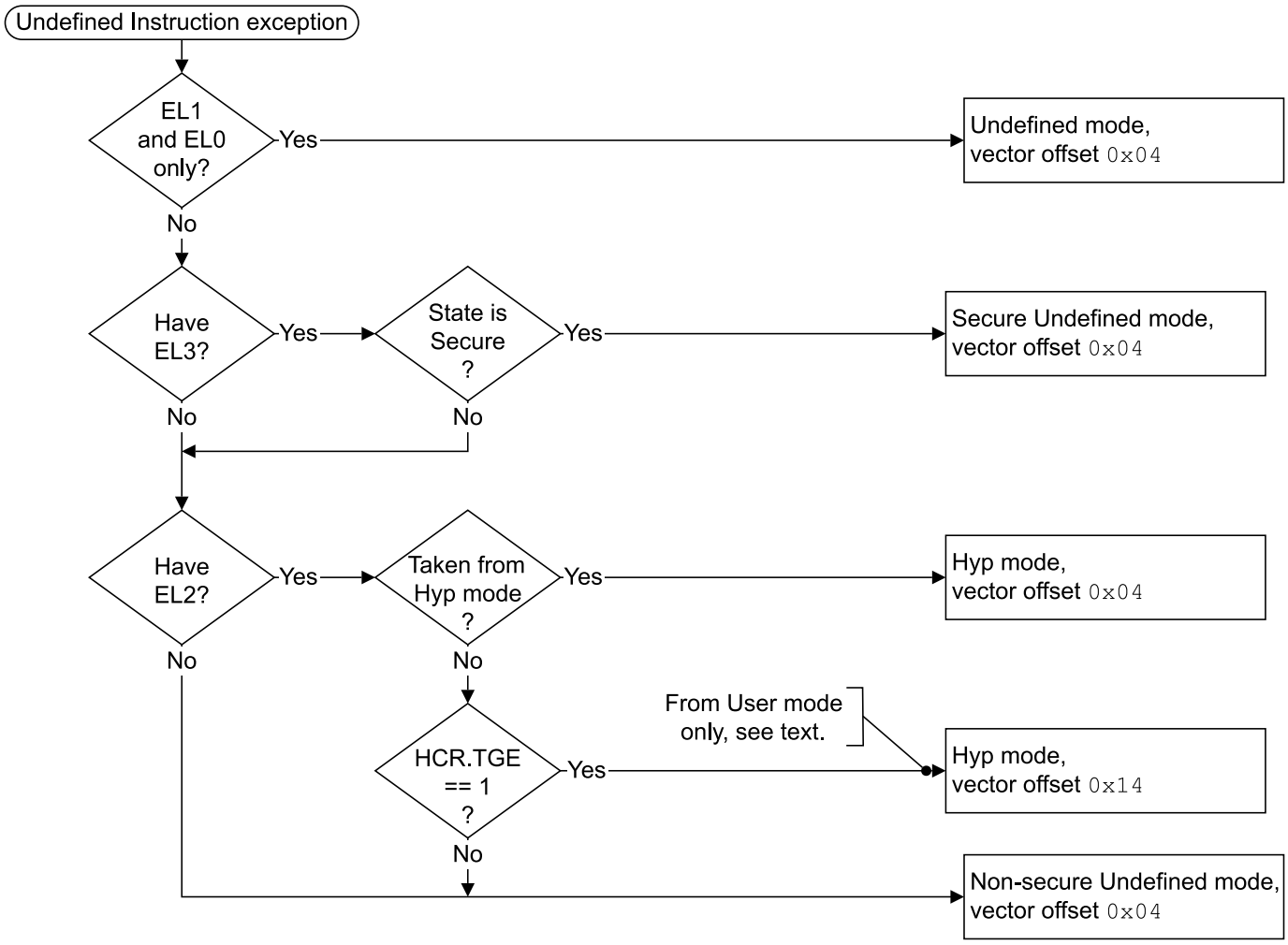

Figure G1-4 The PE mode an Undefined Instruction exception is taken to in AArch32 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also UNPREDICTABLE cases when the value of HCR.TGE is 1.

G1.17.1.2 Pseudocode description of taking the Undefined Instruction exception

The `AArch32_Undefined()` pseudocode procedure determines whether the Undefined Instruction exception is taken to AArch32 state. If it is taken to AArch32 state, the `AArch32_TakeUndefInstrException()` pseudocode procedure describes how the PE takes the exception.

An Undefined Instruction exception is taken to an Exception level using AArch64 if either:

- It is generated in User mode when EL1 is using AArch64.
- It is generated in User mode when EL2 is enabled in the current Security state and is using AArch64 and the value of HCR_EL2.TGE is 1.

G1.17.1.3 Conditional execution of undefined instructions

The conditional execution rules described in Conditional execution apply to all instructions. This includes undefined instructions and other instructions that would cause entry to the Undefined Instruction exception.

If such an instruction fails its condition check, the behavior depends on the potential cause of entry to the Undefined Instruction exception, as follows:

- If the potential cause is the execution of the instruction itself and depends on data values used by the instruction, the instruction executes as a `NOP` and does not cause an Undefined Instruction exception.
- In the following cases, it is IMPLEMENTATION DEFINED whether the instruction executes as a `NOP` or causes an Undefined Instruction exception:
  - The potential cause is the execution of an earlier System register access instruction, floating-point instruction, or Advanced SIMD instruction.
  - The potential cause is the execution of the instruction itself without dependence on the data values used by the instruction.

  An implementation must handle all such cases in the same way.

G1.17.1.4 Interaction of UNDEFINED instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED then the instruction is CONSTRAINED UNPREDICTABLE.

Note

An example of this is where both:

- An instruction, or instruction class, is made UNDEFINED by some general principle, or by a configuration field.
- A particular encoding of that instruction or instruction class is specified as CONSTRAINED UNPREDICTABLE.

G1.17.2 Monitor Trap exception

The Monitor Trap exception is implemented only as part of EL3, and can be generated only if EL3 is using AArch32.

Note

The Monitor Trap exception is taken using offset `0x04` in the Monitor vector table. In the other vector tables, this offset is used for the Undefined Instruction exception. See Undefined Instruction exception and The vector tables and exception offsets.

A Monitor Trap exception is generated if the PE is running in a mode other than Monitor mode, and commits for execution a `WFI` or `WFE` instruction that would otherwise cause suspension of execution when:

- In the case of the `WFI` instruction, the value of the SCR.TWI bit is 1.
- In the case of the `WFE` instruction, the value of the SCR.TWE bit is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Since a `WFE` or `WFI` can complete at any time, even without a Wakeup event, the traps on `WFE` of `WFI` are not guaranteed to be taken, even if the `WFE` or `WFI` is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The preferred return address for a Monitor Trap exception is the address of the instruction that generated the exception. The exception return uses the SPSR and LR_mon values generated by the exception entry, as follows:

- If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
- If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.

For more information, see Exception return to an Exception level using AArch32.

G1.17.2.1 The PE mode to which the Monitor Trap exception is taken

When EL3 is using AArch32, a Monitor Trap exception is taken to Monitor mode, using a vector offset of `0x04` from the Monitor exception base address.

G1.17.2.2 Pseudocode description of taking the Monitor Trap exception

The `AArch32_TakeMonitorTrapException()` pseudocode procedure describes how the PE takes the exception.

G1.17.3 Hyp Trap exception

The Hyp Trap exception provides the standard mechanism for trapping Guest OS functions to the hypervisor.

The Hyp Trap exception is implemented only as part of EL2 and can be generated only if EL2 is using AArch32.

A Hyp Trap exception is generated if the PE is running in a Non-secure mode other than Hyp mode, and commits for execution an instruction that is trapped to Hyp mode. Instruction traps are enabled by setting bits to 1 in the HCR, HCPTR, HDCR, or HSTR. For more information, see EL2 configurable controls.

Traps to Hyp mode never apply in Secure state, regardless of the value of the SCR.NS bit.

The preferred return address for a Hyp Trap exception is the address of the trapped instruction. The exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

Note

The SPSR and ELR_hyp values generated on exception entry can be used, without modification, for an exception return to re-execute the trapped instruction. If the exception handler emulates the trapped instruction, and must return to the following instruction, the emulation of the instruction must include modifying ELR_hyp, and possibly updating SPSR_hyp.

When the PE enters the handler for a Hyp Trap exception, the HSR holds syndrome information for the exception. For more information, see Use of the HSR.

G1.17.3.1 The PE mode to which the Hyp Trap exception is taken

A Hyp Trap exception is taken to Hyp mode, using a vector offset of `0x14` from the Hyp exception base address.

G1.17.3.2 Pseudocode description of taking the Hyp Trap exception

The `AArch32_TakeHypTrapException()` pseudocode procedure describes how the PE takes the exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.17.4 Supervisor Call (SVC) exception

The Supervisor Call instruction, `SVC`, requests a supervisor function, typically to request an operating system function. When EL1 is using AArch32, executing an `SVC` instruction causes the PE to enter Supervisor mode. For more information, see SVC.

Note

In an implementation that includes EL2, when EL2 is using AArch32:

- When an `SVC` instruction is executed in Hyp mode, the Supervisor Call exception is taken to Hyp mode. For more information, see SVC.
- When the HCR.TGE bit is set to 1, the Supervisor Call exception generated by execution of an `SVC` instruction in Non-secure User mode is routed to Hyp mode. For more information, see Supervisor Call exception, when the value of HCR.TGE is 1.

By default, a Supervisor Call exception that is taken to AArch32 state is taken to Supervisor mode, but a Supervisor Call exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see The PE mode to which the Supervisor Call exception is taken.

The preferred return address for a Supervisor Call exception is the address of the next instruction after the `SVC` instruction. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Secure or Non-secure Supervisor mode, the exception return uses the SPSR and LR_svc values generated by the exception entry, in an exception return instruction without subtraction.
- If returning from Hyp mode, the exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see Exception return to an Exception level using AArch32.

G1.17.4.1 The PE mode to which the Supervisor Call exception is taken

Figure G1-5 shows how the implementation, state, and configuration options determine the PE mode to which a Supervisor Call exception is taken, when the exception is taken to an Exception level that is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

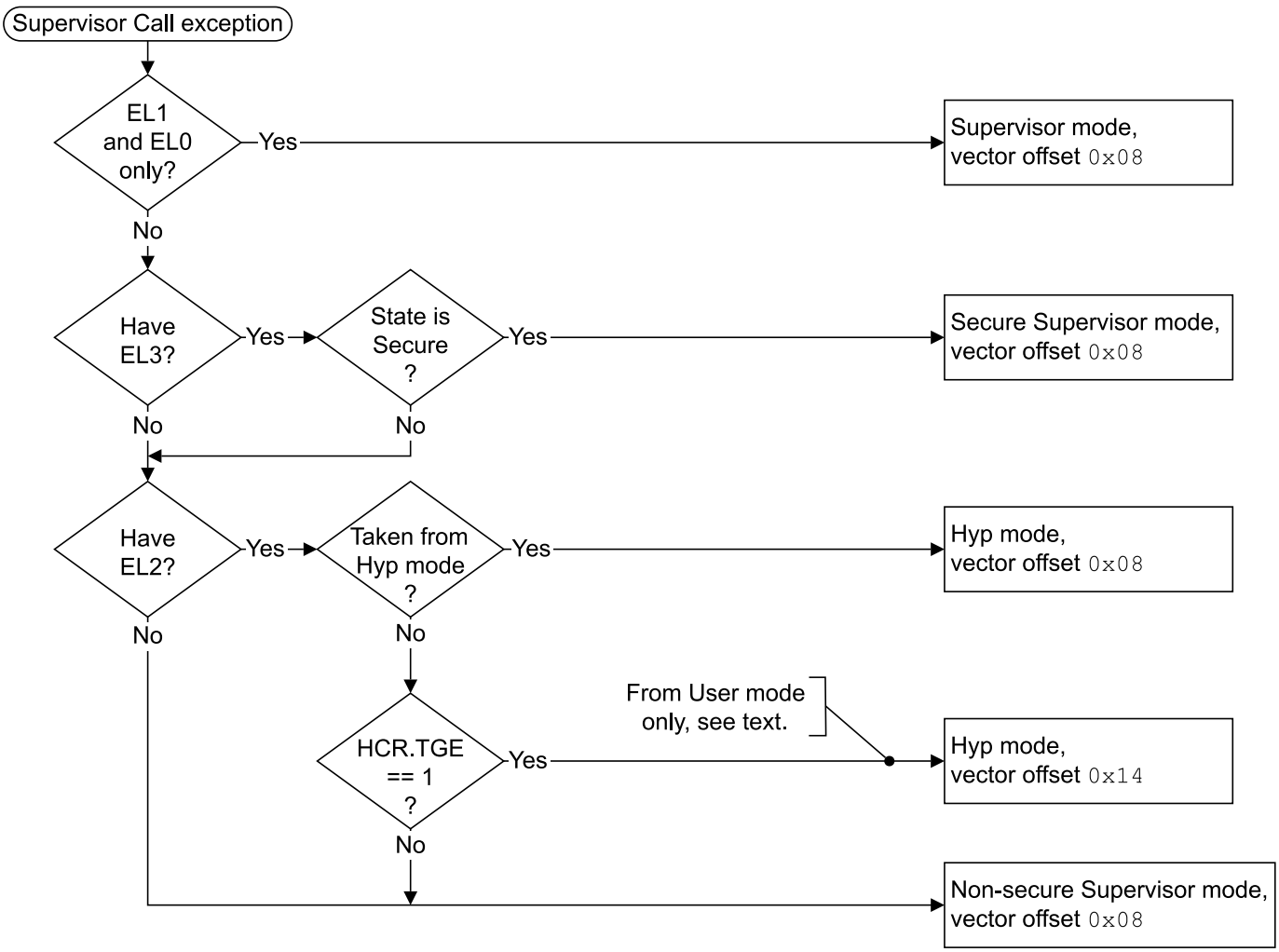

Figure G1-5 The PE mode the Supervisor Call exception is taken to in AArch32 state

See also UNPREDICTABLE cases when the value of HCR.TGE is 1.

G1.17.4.2 Pseudocode description of taking the Supervisor Call exception

The `AArch32_CallSupervisor()` pseudocode procedure determines whether the Supervisor Call exception is taken to AArch32 state. If it is taken to AArch32 state, the `AArch32_TakeSVCException()` pseudocode procedure describes how the PE takes the exception.

An Supervisor Call exception is taken to an Exception level using AArch64 if either:

- It is generated by executing an `SVC` instruction in User mode when EL1 is using AArch64.
- It is generated by executing an `SVC` instruction in Non-secure User mode when EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

G1.17.5 Secure Monitor Call (SMC) exception

The Secure Monitor Call exception is implemented only as part of EL3. When EL3 is using AArch32, the exception is taken to Monitor mode.

The Secure Monitor Call instruction, `SMC`, requests a Secure Monitor function. When EL3 is using AArch32, executing an `SMC` instruction causes the PE to enter Monitor mode. For more information, see SMC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- In an implementation that includes EL2, execution of an `SMC` instruction in a Non-secure EL1 mode can be trapped to EL2. When EL2 is using AArch32, this means that when HCR.TSC 1, execution of an `SMC` instruction in a Non-secure EL1 mode generates a Hyp Trap Exception that is taken to Hyp mode.
- The Operation pseudocode in the description of the AArch32 `SMC` instruction, in SMC, identifies cases where execution of the instruction generates an exception that is taken to EL3 using AArch64.

The preferred return address for a Secure Monitor Call exception is the address of the next instruction after the `SMC` instruction. For an exception taken to AArch32 state, this return is performed using the SPSR and LR_mon values generated by the exception entry, using an exception return instruction without a subtraction.

For more information, see Exception return to an Exception level using AArch32.

Note

For an exception taken to AArch32 state, the exception handler can return to the `SMC` instruction itself by returning using a subtraction of 4, without any adjustment to the SPSR.IT[7:0] bits. If it does this, the return occurs, then asynchronous exceptions might occur and be handled, then the `SMC` instruction is re-executed and another Secure Monitor Call exception occurs.
This relies on:

- The `SMC` instruction being used correctly, either outside an IT block or as the last instruction in an IT block, so that the SPSR.IT[7:0] bits indicate unconditional execution.
- The Secure Monitor Call handler not changing the result of the original conditional execution test for the `SMC` instruction.

G1.17.5.1 The PE mode to which the Secure Monitor Call exception is taken

The Secure Monitor Call exception is supported only as part of EL3. When EL3 is using AArch32, a Secure Monitor Call exception is taken to Monitor mode, using vector offset `0x08` from the Monitor exception base address.

Note

- An `SMC` instruction that is trapped to Hyp mode because HCR.TSC is set to 1 generates a Hyp Trap exception, see The PE mode to which the Hyp Trap exception is taken.
- If EL3 is using AArch64 then Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64 describes the effect of executing an `SMC` instruction at an Exception level that is using EL1.

G1.17.5.2 Pseudocode description of taking the Secure Monitor Call exception

The `AArch32_TakeSMCException()` pseudocode procedure describes how the PE takes the exception when the exception is taken to an Exception level that is using AArch32.

G1.17.6 Hypervisor Call (HVC) exception

The Hypervisor Call exception is implemented only as part of EL2.

The Hypervisor Call instruction, `HVC`, requests a hypervisor function. When EL2 is using AArch32, executing an `HVC` instruction generates a Hypervisor Call exception that is taken to Hyp mode. For more information, see HVC.

Note

- Execution of `HVC` instructions is disabled when the value of SCR.HCE is 0. Descriptions of `HVC` instruction execution elsewhere in this section assume the Effective value of SCR.HCE is 1.
- When EL2 is using AArch64 an `HVC` instruction executed in a Non-secure EL1 mode generates an exception that is taken to EL2 using AArch64.

The preferred return address for a Hypervisor Call exception is the address of the next instruction after the `HVC` instruction. The exception return is performed by an `ERET` instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see Exception return to an Exception level using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL2 is using AArch32, executing an `HVC` instruction transfers the immediate argument of the instruction to the HSR. The exception handler retrieves the argument from the HSR, and therefore does not have to access the original HVC instruction. For more information, see Use of the HSR.

G1.17.6.1 The PE mode to which the Hypervisor Call exception is taken

The Hypervisor Call exception is supported only as part of EL2. When EL2 is using AArch32, a Hypervisor Call exception is taken to Hyp mode, using a vector offset that depends on the mode from which the exception is taken, as Figure G1-6 shows. This offset is from the Hyp exception base address.

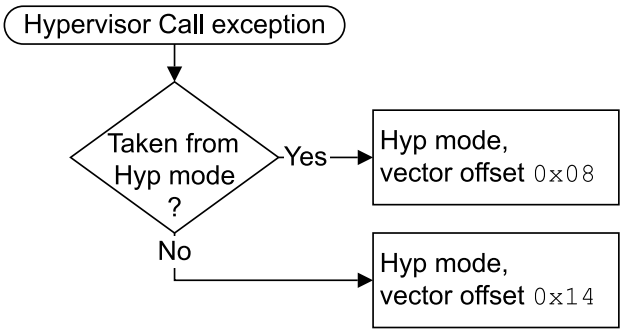

Figure G1-6 The PE mode the Hypervisor Call exception is taken to in AArch32 state

G1.17.6.2 Pseudocode description of taking the Hypervisor Call exception

The `AArch32_CallHypervisor()` pseudocode procedure determines whether the valid execution of an `HVC` instruction in AArch32 state generates an exception that is taken to EL2 using AArch64, or generates a Hypervisor Call exception taken to Hyp mode. The `AArch32_TakeHVCException()` pseudocode procedure describes how the PE takes a Hypervisor Call exception.

G1.17.7 Prefetch Abort exception

A Prefetch Abort exception can be generated by:

- A synchronous memory abort on an instruction fetch.

  Note

  Asynchronous External aborts on instruction fetches are reported as SError exceptions using the Data Abort exception, see Data Abort exception.

  A Prefetch Abort exception entry is synchronous to the instruction whose fetch aborted.
  For more information about memory aborts see VMSAv8-32 memory aborts.

- A Breakpoint, Vector Catch or Breakpoint Instruction exception, see AArch32 Self-hosted Debug.

Note

If an implementation fetches instructions speculatively, it must handle a synchronous abort on such an instruction fetch by:

- Generating a Prefetch Abort exception only if the instruction would be executed in a simple sequential execution of the program.
- Ignoring the abort if the instruction would not be executed in a simple sequential execution of the program.

By default, when EL1 is using AArch32, a Prefetch Abort exception is taken to Abort mode, but a Prefetch Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the Prefetch Abort exception is taken to an Exception level that is using AArch32, see The PE mode to which the Prefetch Abort exception is taken.
- About cases where the Prefetch Abort generates an exception that is taken to an Exception level that is using AArch64, see Pseudocode description of taking the Prefetch Abort exception.

The preferred return address for a Prefetch Abort exception is the address of the aborted instruction. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_abt and LR_abt if returning from Abort mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information about the handling of Prefetch Abort exceptions in AArch32 state, see Exception return to an Exception level using AArch32.

G1.17.7.1 Prefetch Abort exception reporting a PC alignment fault exception

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, and:

If the exception is taken to EL1 using AArch32 or EL3 using AArch32

- The IFSR indicates the cause of the exception:
  - If the value of TTBCR.EAE is 0, IFSR.FS takes the value `0b00001`.
  - If the value of TTBCR.EAE is 1, IFSR.STATUS takes the value `0b100001`.
- IFAR holds the value of the address that faulted, including the misaligned low order bit or bits.
- R14_abt holds the address that faulted, including the misaligned low order bit or bits, with the standard offset for a Prefetch Abort exception.

If the exception is taken to EL2 using AArch32

- HSR.EC takes the value `0b100010`.
- HSR.IL is UNKNOWN.
- HSR.ISS is RES0.
- HIFAR and ELR_hyp each hold the value of the address that faulted, including the misaligned low order bit or bits.

For a PC alignment fault exception taken to an Exception level that is using AArch32:

- If the exception occurred because of the CONSTRAINED UNPREDICTABLE behavior of a branch to an unaligned PC value, as described in Branching to an unaligned PC, then bit[0] of the faulting address is forced to zero, and therefore the misalignment is because the value of bit[1] of this address is 1.
- If the exception occurred on an exit from Debug state, as described in Exiting Debug state, then it is CONSTRAINED UNPREDICTABLE whether bit[0] of the faulting address is forced to zero.

G1.17.7.2 The PE mode to which the Prefetch Abort exception is taken

Figure G1-7 shows how the implementation, state, and configuration options determine the PE mode to which a Prefetch Abort exception is taken, when the exception is taken to an Exception level that is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In this figure, the Effective value of HCR2.TEA is 0 if FEAT_RAS is not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

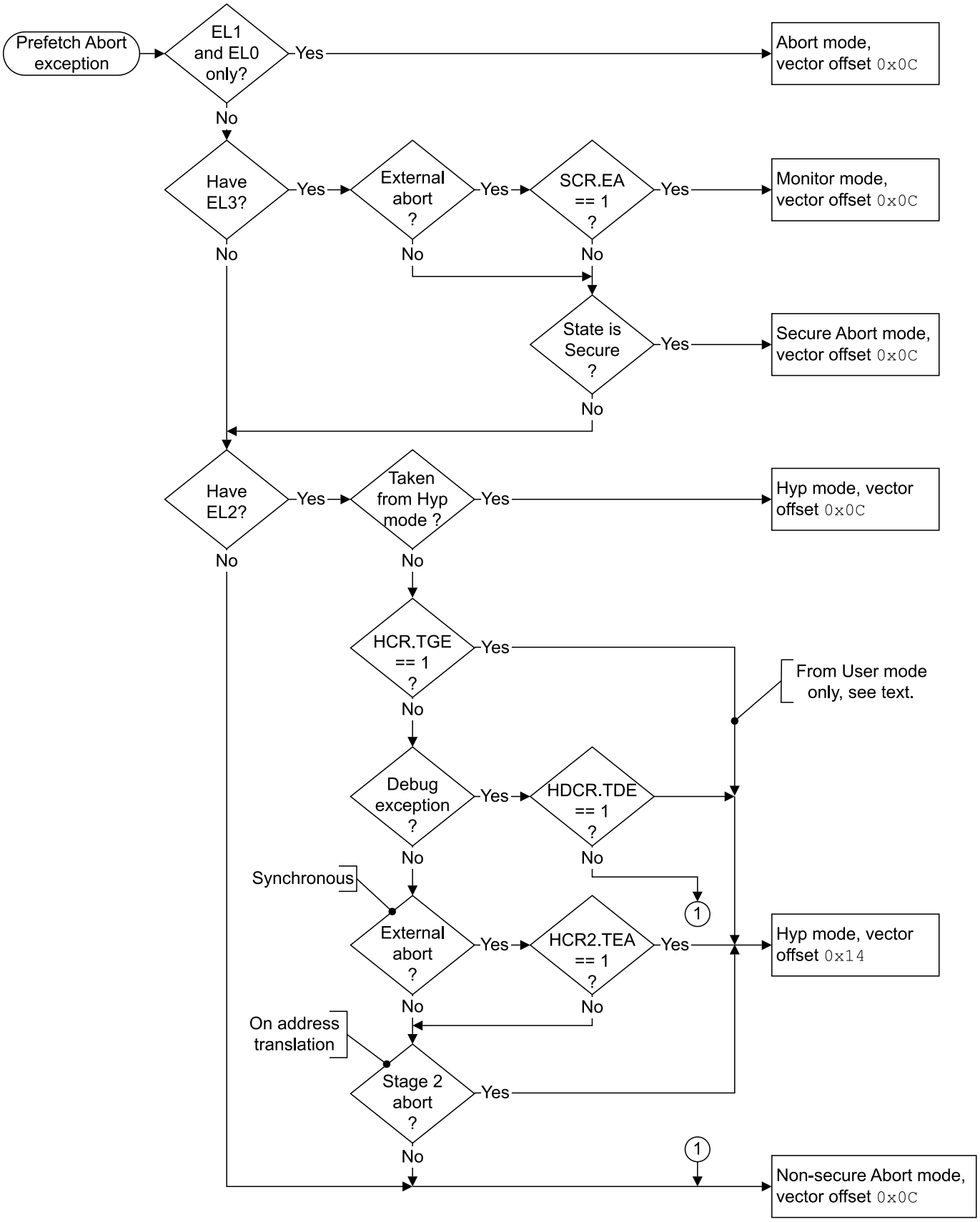

Figure G1-7 The PE mode the Prefetch Abort exception is taken to in AArch32 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also UNPREDICTABLE cases when the value of HCR.TGE is 1.

G1.17.7.3 Pseudocode description of taking the Prefetch Abort exception

The `AArch32_Abort()` pseudocode function determines whether the Prefetch Abort condition generates an exception that is taken to an Exception level that is using AArch64, or generates a Prefetch Abort exception that is taken in AArch32 state. When the exception is taken in AArch32 state, the `AArch32_TakePrefetchAbortException()` pseudocode procedure describes how the PE takes the exception.

The exception is taken to an Exception level using AArch64 if one of the following applies:

- The exception is generated in User mode when EL1 is using AArch64.
- The implementation includes EL2, EL2 is using AArch64, and one of the following applies:
  - The value of HCR_EL2.TGE is 1 and the exception is generated in Non-secure User mode.
  - The value of MDCR_EL2.TDE is 1 and the exception is generated by a Debug exception in a Non-secure EL1 or Non-secure EL0 mode.
  - The exception is generated by a stage 2 fault during a stage 1 translation table walk using the AArch32 Non-secure EL1&0 translation regime.
- The implementation includes EL3, EL3 is using AArch64, the value of SCR_EL3.EA is 1. and the exception is generated by an External abort in AArch32 state.

G1.17.8 Data Abort exception

In AArch32 state, a Data Abort exception can be generated by:

- A synchronous abort on a data read or write memory access. Exception entry is synchronous to the instruction that generated the memory access.
- An SError exception. The SError exception might be caused by an External abort on a memory access, which can be any of:
  - A data read or write access.
  - An instruction fetch.
  - In a VMSA memory system, a translation table access.

  Exception entry occurs asynchronously.

  As described in Asynchronous exception masking controls, SError exceptions can be masked. When this happens, a generated SError exception is not taken until it is not masked.
- A watchpoint, see Watchpoint exceptions.

By default, when EL1 is using AArch32 a Data Abort exception is taken to Abort mode, but a Data Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the Data Abort exception is taken to an Exception level that is using AArch32, see The PE mode to which the Data Abort exception is taken.
- About memory aborts in AArch32 state, see VMSAv8-32 memory aborts.
- About cases where the Data Abort generates an exception that is taken to an Exception level that is using AArch64, see Pseudocode description of taking the Data Abort exception.

The preferred return address for a Data Abort exception is the address of the instruction that generated the aborting memory access, or the address of the instruction following the instruction boundary at which an SError exception was taken. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 8. This means using:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SPSR_abt and LR_abt if returning from Abort mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information about the handling of Data Abort exceptions in AArch32 state, see Exception return to an Exception level using AArch32.

G1.17.8.1 The PE mode to which the Data Abort exception is taken

Figure G1-8 shows the determination of the mode to which a Data Abort exception is taken when the exception is taken to an Exception level that is using AArch32.

Note

In this figure, the Effective value of HCR2.TEA is 0 if FEAT_RAS is not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

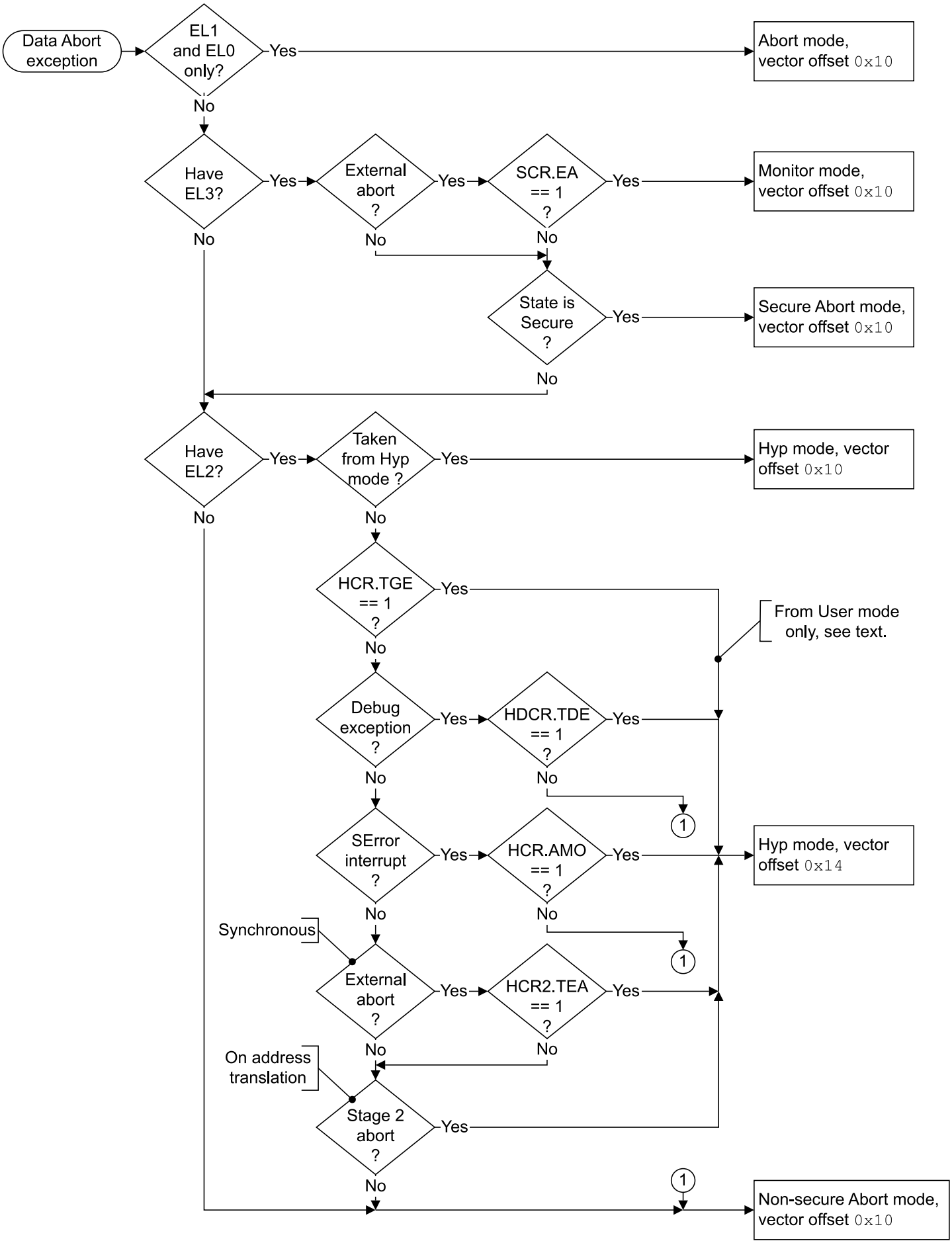

Figure G1-8 The PE mode the Data Abort exception is taken to in AArch32 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also UNPREDICTABLE cases when the value of HCR.TGE is 1.

G1.17.8.2 Pseudocode description of taking the Data Abort exception

The `AArch32_Abort()` pseudocode function determines whether the Data Abort condition generates an exception that is taken to an Exception level that is using AArch64, or generates a Data Abort exception that is taken in AArch32 state. When the exception is taken in AArch32 state, the `AArch32_TakeDataAbortException()` pseudocode procedure describes how the PE takes the exception.

The exception is taken to an Exception level using AArch64 if one of the following applies:

- The exception is generated in User mode when EL1 is using AArch64.
- The implementation includes EL2, EL2 is using AArch64, and one of the following applies:
  - The value of HCR_EL2.TGE is 1 and the exception is generated in Non-secure User mode.
  - The value of MDCR_EL2.TDE is 1 and the exception is generated by a Debug exception in a Non-secure EL1 or Non-secure EL0 mode.
  - The exception is generated by a stage 2 fault during a stage 1 translation table walk using the AArch32 Non-secure EL1&0 translation regime.
- The implementation includes EL3, EL3 is using AArch64, the value of SCR_EL3.EA is 1. and the exception is generated by an External abort in AArch32 state.

G1.17.8.3 Effects of data-aborted instructions

An instruction that accesses data memory can modify memory by storing one or more values. If the execution of such an instruction generates a Data Abort exception, or causes Debug state entry because of a watchpoint set on the location, the value of each memory location that the instruction stores to is:

- Unchanged for any location for which one of the following applies:
  - An Alignment fault is generated.
  - An MMU fault is generated.
  - A Watchpoint is generated.
  - An External abort is generated, if that External abort is taken synchronously.
- UNKNOWN for any location for which no exception and no debug event is generated.

If the access to a memory location generates an External abort that is taken asynchronously, it is outside the scope of the architecture to define the effect of the store on that memory location, because this depends on the system-specific nature of the External abort. However, in general, Arm recommends that such locations are unchanged.

For External aborts and Watchpoints, where in principle faulting could be identified at byte or halfword granularity, the size of a location in this definition is the size for which a memory access is single-copy atomic.

In AArch32 state, instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers. The registers loaded can include the PC.
- By loading values into one or more of the registers in the Advanced SIMD and floating-point register file.
- By specifying *base register writeback*, in which the base register used in the address calculation has a modified value written to it. All instructions that support base register writeback have CONSTRAINED UNPREDICTABLE results if base register writeback is specified with the PC as the base register. Only general-purpose registers can be modified reliably in this way.
- By a direct transfer to or from the Debug Communication Channel (DCC) register, using the LDC and STC instructions. For more information, see The Debug Communication Channel and Instruction Transfer Register.

  If the instruction that accesses the DCC System registers is an LDC or STC instruction, UNKNOWN values are left in the Data Transfer Register and DCC flow-control flags.
- By modifying PSTATE.

If the execution of such an instruction generates a synchronous Data Abort exception, the following rules determine the values left in these registers:

- On entry to the Data Abort exception handler:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The PC value is the Data Abort vector address, see Exception vectors and the exception base address.
  - The LR_abt value is determined from the address of the aborted instruction.

  Neither value is affected by the results of any load specified by the instruction.
- The base register is restored to its original value if either:
  - The aborted instruction is a load and the list of registers to be loaded includes the base register.
  - The base register is being written back.
- If the instruction only loads one general-purpose register the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, UNKNOWN values are left in destination registers other than the PC and the base register of the instruction.
- If the instruction affects any registers in the Advanced SIMD and floating-point register file, UNKNOWN values are left in the registers that are affected.
- PSTATE bits that are not defined as updated on exception entry retain their current value.
- If the instruction is a `STREX`, `STREXB`, `STREXH`, or `STREXD`, `<Rd>` is not updated.

After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN. Therefore, Arm strongly recommends that the abort handler performs a `CLREX` instruction, or a dummy `STREX` instruction, to clear the Exclusives monitor state.

An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted. The error that caused the External abort might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see RAS PE Architecture.

G1.17.8.4 The Arm abort model

The abort model used by an Arm PE is described as a *Base Restored Abort Model*. This means that if a synchronous Data Abort exception is generated by executing an instruction that specifies base register writeback, the value in the base register is unchanged.

The abort model applies uniformly across all instructions.

G1.17.9 Virtual SError interrupt exception

The Virtual SError interrupt exception is implemented only as part of EL2 is enabled in the current Security state.

A Virtual SError interrupt exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.A is 0.
- Either:
  - EL2 is using AArch32 and the values of the HCR.{TGE, AMO, VA} bits are {0, 1, 1}.
  - EL2 is using AArch64 and the values of the HCR_EL2.{TGE, AMO, VA} bits are {0, 1, 1}.

The preferred return address for a Virtual SError interrupt exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state, this return is performed using the SPSR and LR_abt values generated by the exception entry, using an exception return instruction without subtraction.

G1.17.9.1 The PE mode to which the Virtual SError interrupt exception is taken

The Virtual SError interrupt exception is taken using a vector offset of `0x10` from the Non-secure exception base address.

The conditions for generating a Virtual SError interrupt exception in AArch32 state mean the exception is:

- Taken from a EL1 or EL0 mode.
- Taken to Abort mode if EL1 is using AArch32.
- Taken to EL1, when EL0 is using AArch32 and EL1 is using AArch64.

For more information, see Virtual exceptions when an implementation includes EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Because a Virtual SError interrupt exception taken to AArch32 state is always taken to Abort mode, on exception entry the preferred return address is always saved to LR_abt.

G1.17.9.2 Pseudocode description of taking the Virtual SError interrupt exception

The `AArch32_TakeVirtualSErrorException()` pseudocode procedure describes how the PE takes the exception.

G1.17.10 IRQ exception

The IRQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an IRQ interrupt request input to the PE.

When an IRQ exception is taken, exception entry is precise to an instruction boundary.

As described in Asynchronous exception masking controls, IRQ exceptions can be masked. When this happens, a generated IRQ exception is not taken until it is not masked.

By default, when EL1 is using AArch32, an IRQ exception is taken to IRQ mode, but an IRQ exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the exception is taken to an Exception level using AArch32, see The PE mode to which the physical IRQ exception is taken.
- About cases where the exception is taken to an Exception level using AArch64, see Pseudocode description of taking the physical IRQ exception.

The preferred return address for an IRQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_irq and LR_irq if returning from IRQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information, see Exception return to an Exception level using AArch32.

G1.17.10.1 The PE mode to which the physical IRQ exception is taken

Figure G1-9 shows how the implementation, state, and configuration options determine the mode to which an IRQ exception is taken when the exception is taken to an Exception level that is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

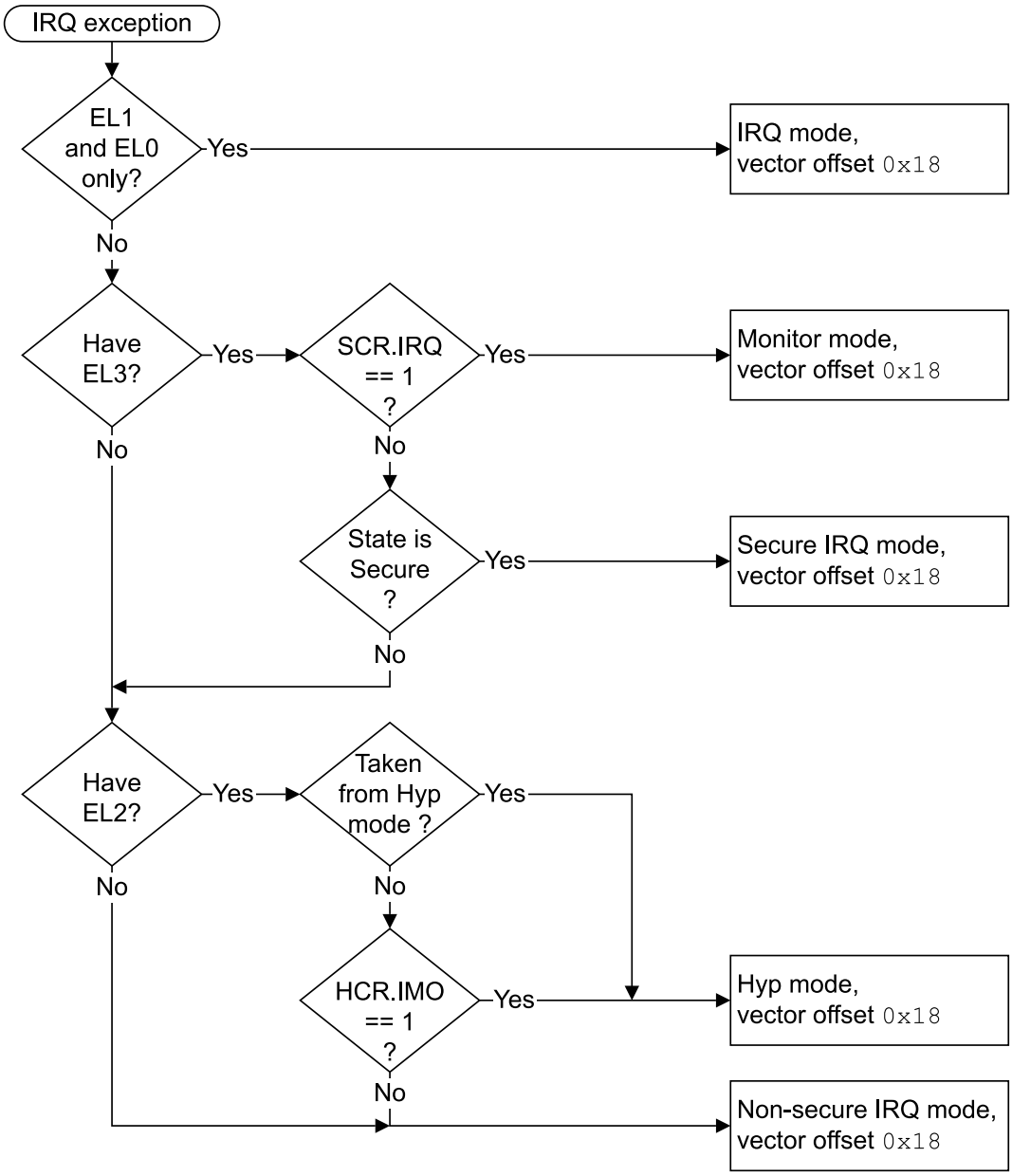

Figure G1-9 The PE mode the IRQ exception is taken to in AArch32 state

G1.17.10.2 Pseudocode description of taking the physical IRQ exception

The `AArch32_TakePhysicalIRQException()` pseudocode procedure describes how the PE takes the exception. This procedure includes the case where the exception is taken to an Exception level that is using AArch64. This happens if one of the following applies:

- The exception is taken from User mode and EL1 is using AArch64. The Exception is taken to EL1 using AArch64.
- The exception is taken from User mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.TGE is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from EL0 or EL1 mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.IMO is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from a PE mode other than Monitor mode, EL3 is implemented and using AArch64, and the value of SCR_EL3.IRQ is 1. The Exception is taken to EL3 using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.17.11 Virtual IRQ exception

The Virtual IRQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual IRQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.I is 0.
- Either:
  - EL2 is using AArch32 and the value of HCR.{TGE, IMO} is {0, 1}.
  - EL2 is using AArch64 and the value of HCR_EL2.{TGE, IMO} is {0, 1}.
- One of the following applies:
  - EL2 is using AArch32 and the value of HCR.VI is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VI is 1.
  - A Virtual IRQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual IRQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

G1.17.11.1 The PE mode to which the Virtual IRQ exception is taken

The Virtual IRQ exception uses a vector offset of `0x18`.

The conditions for generating a Virtual IRQ exception in AArch32 state mean the exception is:

- Taken from an EL1 or EL0 mode.
- Taken to IRQ mode if EL1 is using AArch32.
- Taken to EL1 if EL0 is using AArch32 and EL1 is using AArch64.

For more information, see Virtual exceptions when an implementation includes EL2.

G1.17.11.2 Pseudocode description of taking the Virtual IRQ exception

The `AArch32_TakeVirtualIRQException()` pseudocode procedure describes how the PE takes the exception.

G1.17.12 FIQ exception

The FIQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an FIQ interrupt request input to the PE.

When an FIQ exception is taken, exception entry is precise to an instruction boundary.

As described in Asynchronous exception masking controls, FIQ exceptions can be masked. When this happens, a generated FIQ exception is not taken until it is not masked.

By default, an FIQ exception is taken to FIQ mode, but an FIQ exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the exception is taken to an Exception level using AArch32, see The PE mode to which the physical FIQ exception is taken.
- About cases where the exception is taken to an Exception level using AArch64, see Pseudocode description of taking the FIQ exception.

The preferred return address for an FIQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_fiq and LR_fiq if returning from FIQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information, see Exception return to an Exception level using AArch32.

G1.17.12.1 The PE mode to which the physical FIQ exception is taken

Figure G1-10 shows how the implementation, state, and configuration options determine the PE mode to which an FIQ exception is taken when the exception is taken to an Exception level that is using AArch32.

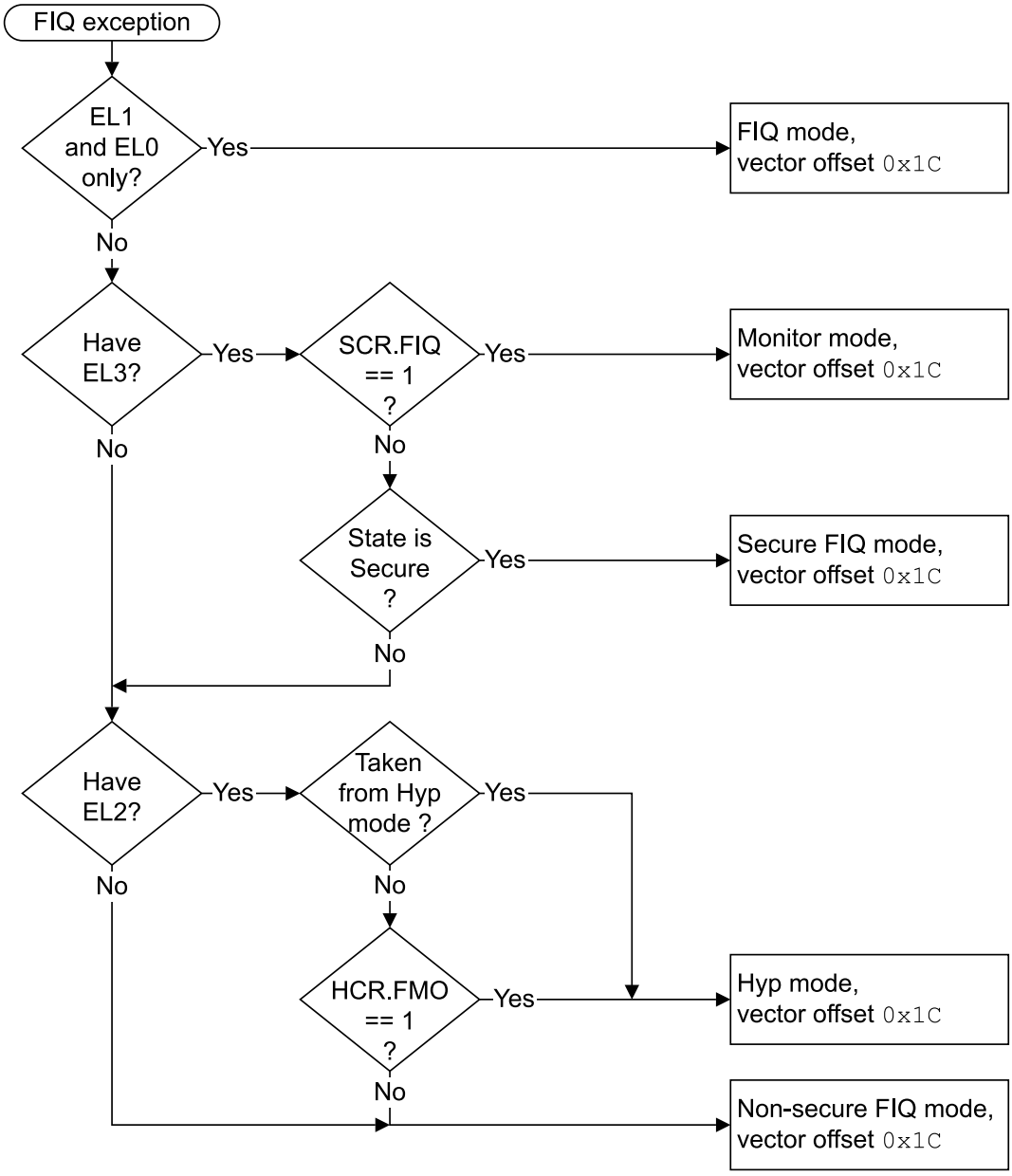

Figure G1-10 The PE mode the FIQ exception is taken to in AArch32 state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.17.12.2 Pseudocode description of taking the FIQ exception

The `AArch32_TakePhysicalFIQException()` pseudocode procedure describes how the PE takes the exception. This procedure includes the case where the exception is taken to an Exception level that is using AArch64. This happens if one of the following applies:

- The exception is taken from User mode and EL1 is using AArch64. The Exception is taken to EL1 using AArch64.
- The exception is taken from User mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.TGE is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from an EL0 or EL1 mode, EL2 is implemented in the current Security state and using AArch64, and the value of HCR_EL2.FMO is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from a PE mode other than Monitor mode, EL3 is implemented and using AArch64, and the value of SCR_EL3.FIQ is 1. The Exception is taken to EL3 using AArch64.

G1.17.13 Virtual FIQ exception

The Virtual FIQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual FIQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.F is 0.
- Either:
  - EL2 is using AArch32 and the value of HCR.{TGE, FMO} is {0, 1}.
  - EL2 is using AArch64 and the value of HCR_EL2.{TGE, FMO} is {0, 1}.
- One of the following applies:
  - EL2 is using AArch32 and the value of HCR.VF is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VF is 1.
  - A Virtual FIQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual FIQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

G1.17.13.1 The PE mode to which the Virtual FIQ exception is taken

The Virtual FIQ exception is taken using a vector offset of `0x1C`.

The conditions for generating a Virtual FIQ exception in AArch32 state mean the exception is:

- Taken from EL1 or EL0.
- Taken to FIQ mode if EL1 is using AArch32.
- Taken to EL1 if EL0 is using AArch32 and EL1 is using AArch64.

For more information, see Virtual exceptions when an implementation includes EL2.

G1.17.13.2 Pseudocode description of taking the Virtual FIQ exception

The `AArch32_TakeVirtualFIQException()` pseudocode procedure describes how the PE takes the exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.17.14 Additional pseudocode functions for exception handling

The `AArch32_EnterMonitorMode()` pseudocode function changes the PE mode to Monitor mode, with the required state changes.

The `AArch32_EnterHypMode()` pseudocode function changes the PE mode to Hyp mode, with the required state changes.

The `AArch32_EnterMode()` pseudocode function changes the PE mode to a PL1 mode, with the required state changes. It is used for all exceptions that are not routed to Hyp mode or Monitor mode.

The `AArch32_EnterMonitorMode()`, `AArch32_EnterHypMode()`, and `AArch32_EnterMode()` functions are described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.18 Reset into AArch32 state

Resets and power domains describes the Armv8 reset model, including the defined levels of reset. When reset is deasserted, the PE starts executing instructions in the highest implemented Exception level. If that Exception level is using AArch32, then it starts execution:

- In Secure state, if the implementation includes EL3.
- With interrupts disabled:
  - In Hyp mode, if the highest implemented Exception level is EL2.
  - In Supervisor mode, otherwise.

Note

- This section describes the architectural requirements for a reset into AArch32 state. It takes no account of whether Arm licenses any particular combination of Exception levels and Execution state. For more information about the licensed combinations, see Exception levels.
- The Execution state in which the highest implemented Execution level starts executing instructions on coming out of reset might be determined by a configuration input signal.

Reset returns some PE state to architecturally-defined or IMPLEMENTATION DEFINED values, and makes other state UNKNOWN, as described in PE state on reset into AArch32 state. For more information about behavior when resetting into an Exception level using AArch32, see:

- Behavior of caches at reset.
- Enabling stages of address translation.
- TLB behavior at reset.
- Reset and debug.

When reset is deasserted, if the PE resets into an Exception level that is using AArch32, it is IMPLEMENTATION DEFINED whether execution starts:

- From an IMPLEMENTATION DEFINED address.
- If reset is into EL3 or EL1, from the low or high reset vector address, as determined by the reset value of the SCTLR.V bit. This reset value can be determined by an IMPLEMENTATION DEFINED configuration input signal.

  Note

  This option might be implemented for compatibility with earlier versions of the architecture.

Software might be able to identify the reset address:

- If reset is into EL3, by reading the reset value of MVBAR. That is, after coming out of reset, by reading MVBAR before the boot software has updated it. It is IMPLEMENTATION DEFINED whether this discovery mechanism is supported.
- If reset is into EL2 or EL1, by reading RVBAR. RVBAR can only be implemented at the highest implemented Exception level, and only if that Exception level is not EL3.

If RVBAR is not implemented, and at all Exception levels other than the highest implemented Exception level, the encoding for RVBAR is UNDEFINED.

The Arm architecture does not define any way of returning to a previous Execution state from a reset.

G1.18.1 PE state on reset into AArch32 state

Immediately after a reset, much of the PE state is architecturally UNKNOWN. However, some of the PE state is defined, see the individual register descriptions for more information. The state that is reset to known values is sufficient to permit predictable initial execution at the highest Exception level, such that this execution is then capable of initializing the remaining state of the system where necessary before use.

If the PE resets to AArch32 state using either a Cold or a Warm reset, the PE state that is defined is as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The global exclusive monitor and local exclusive monitor for the PE are UNKNOWN.
- If reset is into EL2 using AArch32, then reset is into Hyp mode and PSTATE.M resets to `0b1010`, otherwise reset is into Supervisor mode and PSTATE.M resets to `0b0011`.
- PSTATE.IL resets to 0.

G1.18.2 Pseudocode descriptions of reset

The `AArch32_TakeReset()` pseudocode procedure describes how the PE behaves when reset is deasserted.

The `AArch32_ResetGeneralRegisters()` pseudocode function resets the general-purpose registers.

The `AArch32_ResetSIMDFPRegisters()` pseudocode function resets the SIMD and floating-point registers.

The `AArch32_ResetSpecialRegisters()` pseudocode function resets the Special-purpose registers, and the debug System registers DLR and DSPSR, which are used for handling Debug exceptions.

The `AArch32_ResetSystemRegisters()` pseudocode function resets all System registers in the (`coproc == 0b111x`) encoding space to their reset state as defined in the register descriptions in AArch32 System Register Descriptions.

Note

The `AArch32_ResetSystemRegisters()` function only resets the System registers. It has no effect on memory-mapped registers.

The `ResetExternalDebugRegisters()` pseudocode function resets all external debug registers to their reset state as defined in the register descriptions in External Debug Register Descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.19 Mechanisms for entering a low-power state

The following sections describe the architectural mechanisms that a PE can use to request entry to a low-power state:

- Wait For Event and Send Event.
- Wait For Interrupt.

G1.19.1 Wait For Event and Send Event

The Wait For Event (WFE) mechanism permits a PE to request entry to a low-power state, and, if the request succeeds, to remain in that state until an event is generated by a Send Event operation, or another WFE wakeup event occurs. Example G1-2 describes how a spinlock implementation might use this mechanism to save energy.

Example G1-2 Spinlock as an example of using Wait For Event and Send Event

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device it might need to add data received from the device to a queue. If another PE is removing data from the queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the lock has been released. It cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in Synchronization and semaphores.
- If the PE obtains the lock, it performs its memory operation and releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. At this point, it again attempts to obtain the lock.

A spin-lock mechanism is not ideal for all situations:

- In a low-power system, the tight read loop is undesirable because it uses energy to no effect.
- In a multithreaded implementation, the execution of spin-locks by waiting threads can significantly degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock. In this situation, a PE that fails to obtain a lock can execute a Wait For Event instruction, `WFE`, to request entry to a low-power state. When a PE releases a lock, it must execute a Send Event instruction, `SEV`, causing any waiting PEs to wake up. Then, these PEs can attempt to gain the lock again.

The execution of a `WFE` instruction can cause suspension of execution only if all of the following are true:

- The instruction does not cause any other exception.
- When the instruction is executed:
  - The Event Register is not set.
  - There is not a pending WFE wakeup event.

For more information about the trap to EL2, see HCR.{WFI, WFE}.

The architecture does not define the exact nature of the low power state entered as a result of executing a `WFE` instruction, but the execution of a `WFE` instruction must not cause a loss of memory coherency or architectural state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Although a complex operating system can contain thousands of distinct locks, the event sent by this mechanism does not indicate which lock has been released. If the event relates to a different lock, or if another PE acquires the lock more quickly, the PE fails to acquire the lock and can reenter the low-power state waiting for the next event.

The Wait For Event system relies on hardware and software working together to achieve energy saving:

- The hardware provides the mechanism to enter the Wait For Event low-power state.
- The operating system software is responsible for issuing:
  - A Wait For Event instruction, to request entry to the low-power state, used in the example when waiting for a spin-lock.
  - A Send Event instruction, required in the example when releasing a spin-lock.

The mechanism depends on the interaction of:

- WFE wakeup events, see WFE wakeup events.
- The Event Register, see The Event Register.
- The Send Event instructions, see The Send Event instructions.
- The Wait For Event instruction, see The Wait For Event instruction.

G1.19.1.1 The Event Register

The Event Register is a single bit register for each PE. When set, an event register indicates that an event has occurred, since the register was last cleared, that might require some action by the PE. Therefore, the PE must not suspend operation on issuing a `WFE` instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by:

- The execution of an `SEV` instruction on any PE in the multiprocessor system.
- The execution of an `SEVL` instruction by the PE.
- An exception return.
- An event from a Generic Timer event stream, see Event streams.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

As shown in this list, the Event Register might be set by IMPLEMENTATION DEFINED mechanisms.

The Event Register is cleared only by a Wait For Event instruction.

Software cannot read or write the value of the Event Register directly.

G1.19.1.2 The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is 1, the instruction clears the register and completes immediately. Normally, if this happens the software makes another attempt to claim the lock.
- If the Event Register is 0, the PE can suspend execution, and hardware might enter a low-power state. The PE can remain suspended until a WFE wakeup event or a reset occurs. When a WFE wakeup event occurs, or earlier if the implementation chooses, the `WFE` instruction completes.

The execution in AArch32 state of a `WFE` instruction that would otherwise cause suspension of execution might be trapped, see:

- SCTLR.{nTWE, nTWI}.
- HCR.{TWE, TWI}.
- SCR.{TWE, TWI}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Wait For Event instruction, `WFE`, is available at all privilege levels, see WFE.

Software using the Wait For Event mechanism must tolerate spurious wakeup events, including multiple wake ups.

G1.19.1.3 WFE wakeup events

The following events are *WFE wakeup events*:

- The execution of an `SEV` instruction on any PE in the system.
- The execution of an `SEVL` instruction on the PE.
- A physical IRQ interrupt, unless masked by the PSTATE.I bit.
- A physical FIQ interrupt, unless masked by the PSTATE.F bit.
- A physical SError exception, unless masked by the PSTATE.A bit.
- In Non-secure state in any mode other than Hyp mode:
  - When HCR.IMO is set to 1, a virtual IRQ interrupt, unless masked by the PSTATE.I bit.
  - When HCR.FMO is set to 1, a virtual FIQ interrupt, unless masked by the PSTATE.F bit.
  - When HCR.AMO is set to 1, a virtual SError exception, unless masked by the PSTATE.A bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed*, see Halting allowed and halting prohibited.
  See also External Debug Request debug event.
- An event sent by the timer event stream, see Event streams.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- An event caused by the clearing of the global monitor associated with the PE.

In addition to the possible masking of WFE wakeup events shown in this list, when invasive debug is enabled and EDSCR.HDE is set to 1, EDSCR.INTdis can mask interrupts, including masking them acting as WFE wakeup events. See the register description for more information.

As shown in the list of wakeup events, an implementation can include IMPLEMENTATION DEFINED hardware mechanisms to generate wakeup events.

Note

For more information about PSTATE masking, see Asynchronous exception masking controls. If the configuration of the masking controls provided by EL2 and EL3 mean that a PSTATE mask bit cannot mask the corresponding exception, then the physical exception is a WFE wakeup event, regardless of the value of the PSTATE mask bit.

G1.19.1.4 The Send Event instructions

The Send Event instructions are:

SEV, Send Event This causes an event to be signaled to all PEs in the multiprocessor system.

SEVL, Send Event Local

This must set the local Event Register. It might signal an event to other PEs, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the `SEV` instruction. Therefore, Arm recommends that software includes a `DSB` instruction before any `SEV` instruction.

Note

A `DSB` instruction ensures that no instruction, including any `SEV` instruction, that appears in program order after the `DSB` instruction, can execute until the `DSB` instruction has completed. For more information, see Data Synchronization Barrier.

The `SEVL` instruction appears to execute in program order relative to any subsequent `WFE` instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.

Execution of the Send Event instruction sets the Event Register.

The Send Event instructions are available at all privilege levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.19.1.5 Pseudocode description of the Wait For Event mechanism

This section defines pseudocode functions that describe the operation of the Wait For Event mechanism.

The `ClearEventRegister()` pseudocode procedure clears the Event Register of the current PE.

The `IsEventRegisterSet()` pseudocode function returns TRUE if the Event Register of the current PE is 1 and FALSE if it is 0.

The `WaitForEvent()` pseudocode procedure optionally suspends execution until a WFE wakeup event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a `ClearEventRegister()` to occur.

The `SendEvent()` pseudocode procedure sets the Event Register of every PE in the system.

G1.19.2 Wait For Interrupt

AArch32 state supports Wait For Interrupt through an instruction, `WFI`, which is provided in the T32 and A32 instruction sets. For more information, see WFI.

When a PE issues a `WFI` instruction, its execution can be suspended, and a low-power state can be entered.

The execution in AArch32 state of a `WFI` instruction that would otherwise cause suspension of execution might be trapped, see:

- SCTLR.nTWI.
- HCR.WFI.
- SCR.WFI.

The execution of a `WFI` instruction can cause suspension of execution only if both:

- The instruction does not cause any other exception.
- When the instruction is executed, there is not a pending WFI wakeup event.

G1.19.2.1 WFI wakeup events

The PE can remain suspended in its WFI state until it is reset, or one of the following *WFI wakeup events* occurs:

- A physical IRQ interrupt, regardless of the value of the PSTATE.I bit.
- A physical FIQ interrupt, regardless of the value of the PSTATE.F bit.
- A physical SError exception, regardless of the value of the PSTATE.A bit.
- In Non-secure state in any mode other than Hyp mode:
  - When HCR.IMO is set to 1, a virtual IRQ interrupt, regardless of the value of the PSTATE.I bit.
  - When HCR.FMO is set to 1, a virtual FIQ interrupt, regardless of the value of the PSTATE.F bit.
  - When HCR.AMO is set to 1, a virtual SError exception, regardless of the value of the PSTATE.A bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed*, see Halting allowed and halting prohibited.
  See also External Debug Request debug event.

An implementation can include other IMPLEMENTATION DEFINED hardware mechanisms to generate WFI wakeup events.

When a WFI wakeup event is detected, or earlier if the implementation chooses, the `WFI` instruction completes.

WFI wakeup events cannot be masked by the mask bits in the PSTATE.

The architecture does not define the exact nature of the low power state, except that:

- The execution of a `WFI` instruction must not cause a loss of memory coherency.
- If the system is configured such that the `WFI` instruction can be completed, then the `WFI` instruction must not cause a loss of architectural state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wakeup events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

Note

- Because debug events are WFI wakeup events, Arm strongly recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.
- In some previous implementations of Wait For Interrupt, the idle loop is followed by exit functions that must be executed before taking the interrupt. The operation of Wait For Interrupt remains consistent with this model, and therefore differs from the operation of Wait For Event.
- Some implementations of Wait For Interrupt drain down any pending memory activity before suspending execution. The Arm architecture does not require this operation, and software must not rely on Wait For Interrupt operating in this way.

G1.19.2.2 Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into powerdown routines with a `WFI` instruction. Typically, the `WFI` instruction:

1. Forces the completion of execution of any instructions that are in progress, and of all associated bus activity.
2. Suspends the execution of instructions by the PE.

The control logic required to do this tracks the activity of the bus interfaces used by the PE. This means it can signal to an external power controller when there is no ongoing bus activity.

However, memory-mapped and external debug interface accesses to debug registers must continue to be processed while the PE is in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

If FEAT_DoubleLock is implemented and the value of DBGOSDLR.DLK, the OS Double Lock status bit, is set to 1, this idle state must not be signaled to the PE unless the system can guarantee, also, that the debug interface is idle.

Note

When separate Core and Debug power domains are implemented, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see Power domains and debug.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

G1.19.2.3 Pseudocode description of Wait For Interrupt

The `WaitForInterrupt` () pseudocode function optionally suspends execution until a WFI wakeup event or reset occurs, or until some earlier time if the implementation chooses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.20 The AArch32 System register interface

Most System registers are accessed using the instructions described in System register access instructions. The System register interface provides access to those instructions, and:

- These registers are encoded using the parameters {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}, with permitted `coproc` values of `0b1110` and `0b1111`.
- Some of these encodings provide the AArch32 System instructions.
- To maintain compatibility with previous versions of the Arm architecture, the access controls for the AArch32 System registers include the access controls for AArch32 Advanced SIMD and floating-point functionality.

Note

See Background to the System register interface for more information.

The following sections give more information about the AArch32 System register interface:

- System registers in the coproc == `0b111x` encoding space.
- Access to System registers.
- Access controls for Advanced SIMD and floating-point functionality.
- Background to the System register interface.

G1.20.1 System registers in the `coproc == 0b111x` encoding space

In AArch32 state:

- The `coproc == 0b1110` encoding space is reserved for the configuration and control of:
  - Debug features, see Debug System registers.
  - Trace features, see the *Embedded Trace Macrocell Architecture Specification*.
  - Identification registers for the Trivial Jazelle implementation, see Trivial implementation of the Jazelle extension.
- The `coproc == 0b1111` encoding space is reserved for the control and configuration of the PE, including architecture and feature identification. This means these encodings provide access to the System registers that control and return status information for PE operation.

For more information, see AArch32 System Register Descriptions.

G1.20.2 Access to System registers

Most System registers are accessible only from EL1 or higher. For possible accesses from EL0 the register descriptions in AArch32 System Register Descriptions indicate whether a register is accessible from EL0.

G1.20.3 Access controls for Advanced SIMD and floating-point functionality

The CPACR controls access to Advanced SIMD and floating-point functionality from software executing at PL1 or EL0 in AArch32 state:

- The {cp10, cp11} fields control access to all Advanced SIMD and floating-point functionality, and can:
  - Disable EL0 and PL1 access to this functionality.
  - Enable access to this functionality at PL1 only.
  - Enable access to this functionality at EL0 and PL1.
- The ASEDIS field controls access to Advanced SIMD instructions that are not also floating-point instructions.

Initially on powerup or reset into AArch32 state, all access to all Advanced SIMD and floating-point functionality from PL1 and EL0 is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The CPACR has no effect on accesses from Hyp mode.

If an implementation includes EL3, the NSACR determines whether Advanced SIMD and floating-point functionality can be accessed from Non-secure state:

- The {cp10, cp11} fields control Non-secure access to all Advanced SIMD and floating-point functionality.
- The NSASEDIS field controls Non-secure access to Advanced SIMD instructions that are not also floating-point instructions.

If an implementation includes EL2, the HCPTR provides additional controls on Non-secure accesses to Advanced SIMD and floating-point functionality. For accesses that are otherwise permitted by the CPACR and NSACR settings, setting HCPTR bits to 1:

- Traps otherwise-permitted accesses from EL1 or EL0 to EL2. When EL2 is using AArch32, these accesses are trapped to Hyp mode.
- Makes accesses from EL2 mode UNDEFINED. When EL2 is using AArch32, this makes accesses from Hyp mode UNDEFINED.

In the HCPTR:

- The {TCP10, TCP11} fields control access to all Advanced SIMD and floating-point functionality.
- The TASE field controls access to Advanced SIMD instructions that are not also floating-point instructions.
- The TCPAC field traps Non-secure EL1 accesses to the CPACR to Hyp mode.

Note

Whenever a pair of fields control the access to the Advanced SIMD and floating-point functionality, the values of each field of the pair must be identical. If these settings are not identical the behavior of the Advanced SIMD and floating-point functionality is CONSTRAINED UNPREDICTABLE, see Handling of System register control fields for Advanced SIMD and floating-point operation.

For more information about Advanced SIMD and floating-point support, see Advanced SIMD and floating-point support.

G1.20.4 Background to the System register interface

Note

This section is not part of the Architecture specification. It is included only to present the rationale of some aspects of the System register interface.

The interface to the System registers was originally defined as part of a generic *coprocessor* interface that gave access to 15 coprocessors, CP0 - CP15. Of these, CP8 - CP15 were reserved for use by Arm, while CP0 - CP7 were available for IMPLEMENTATION DEFINED coprocessors.

The coprocessors were accessed using coprocessor instructions. These instructions remain part of the T32 and A32 instruction sets, see System register access instructions.

In the Arm coprocessor model, a coprocessor included both:

- Primary and secondary coprocessor registers, which form part of the coprocessor interface.
- A number of internal registers.

When accessing a 32-bit internal coprocessor register, using an `MCR` or `MRC` instruction, the instruction specified:

- The target coprocessor, specified by the `coproc` parameter and taking a value between `p0` for CP0 and `p15` for CP15.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The primary coprocessor register, specified by the `CRn` parameter and taking a value between `c0` and `c15`.
- The secondary coprocessor register, specified by the `CRm` parameter and taking a value between `c0` and `c15`.
- Up to two additional parameters, `opc1` and `opc2`, taking values between 0 and 7.

Other instructions in the group described in System register access instructions take a subset of these parameters:

- In the Armv7 definitions, `LDC` and `STC` instructions take parameters {`coproc`, `CRd`}, where `CRd` is the primary coprocessor register.
- `MCRR` and `MRRC` instructions take parameters {`coproc`, `opc1`, `CRm`}, where `CRm` is the primary coprocessor register.

To maintain backward compatibility, the arguments to an `MCR` or `MRC` instruction remain {`coproc`, `opc1`, `CRn`, `CRm`, `opc2`}. Correspondingly, the encoding of the AArch64 System registers is described using the parameters {`op0`, `op1`, `CRn`, `CRm`, `op2`}. However:

- The naming of these parameters no longer has any particular significance.
- While the `coproc` field is a 4-bit field, `op0` is a 2-bit field.

Of the coprocessors reserved for use by Arm, in earlier versions of the architecture:

- CP15 provided access to the System registers relating to non-debug operation, and was originally called the System control coprocessor. From the introduction of Armv8, these registers are described as being in the `coproc == 0b1111` encoding space.
- CP14 provided access to additional System registers, including those relating to debug and trace. From the introduction of Armv8, these registers are described as being in the `coproc == 0b1110` encoding space.
- CP10 and CP11 were used for Advanced SIMD and floating-point control, and many coprocessor instruction encodings targeting CP10 and CP11 were used as floating-point instruction encodings:
  - Generally from the introduction of Armv8 the architecture does not relate these instructions to the coprocessor encoding space, but the naming of registers and register fields for Advanced SIMD and floating-point control reflects the historic coprocessor model.
  - Because the Advanced SIMD and floating-point functionality used both CP10 and CP11, some System register controls of this functionality have a pair of fields, for example NSACR.{cp10, cp11}. In these cases, both fields must be set to the same value. For more information, see Access controls for Advanced SIMD and floating-point functionality.

From the introduction of Armv8:

- The AArch32 System registers include registers that were described as Special registers in Armv7 and earlier versions of the architecture. This means that the System registers include registers that are outside the earlier coprocessor model.
- The Armv7 AArch32 instruction encodings for `LDC`, `STC`, `MCR`, `MRC`, `MCRR`, and `MRRC` instructions with `coproc` field values other than {`1010`, `1011`, `1110`, `1111`} are available for reuse. Armv8.2 re-uses some encodings in this way.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.21 Advanced SIMD and floating-point support

Advanced SIMD and floating-point instructions introduces:

- The scalar floating-point instructions in the T32 and A32 instruction sets.
- The Advanced SIMD integer and floating-point vector instructions in the T32 and A32 instruction sets.
- The SIMD and floating-point register file, which can be viewed as:
  - Singleword registers S0 - S31.
  - Doubleword registers D0 - D31.
  - Quadword registers Q0 - Q15.
- The *Floating-Point Status and Control Register* (FPSCR).
- Floating-point exceptions and exception traps.

For more information about the System registers for the Advanced SIMD and floating-point operation, see Advanced SIMD and floating-point System registers. Software can interrogate these registers to discover the implemented Advanced SIMD and floating-point support.

AArch32 implications of not including support for Advanced SIMD and floating-point summarizes the effects of not supporting these instructions, and the following subsections give more information about the Advanced SIMD and Floating-point support:

- Enabling Advanced SIMD and floating-point support.
- Advanced SIMD and floating-point System registers.
- Context switching when using Advanced SIMD and floating-point functionality.

G1.21.1 AArch32 implications of not including support for Advanced SIMD and floating-point

The architecture generally requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. However, for implementations targeting specialized markets, Arm might produce or license implementations that do not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- The CPACR.{ASEDIS, cp11, cp10} fields are RES0.
- The NSACR.{NSASEDIS, cp11, cp10} fields are RES0.
- The HCPTR.{TASE, TCP11, TCP10} fields are RES1.
- The FPEXC, FPSCR, FPSID, MVFR0, MVFR1, and MVFR2 registers are not implemented and their encodings are UNDEFINED.
- Attempted accesses to Advanced SIMD and floating-point functionality are UNDEFINED. This means:
  - All Advanced SIMD and floating-point instructions are UNDEFINED.
  - Attempts to access the Advanced SIMD and floating-point System registers are UNDEFINED.

G1.21.2 Enabling Advanced SIMD and floating-point support

Software must ensure that the required access to the Advanced SIMD and floating-point features is enabled. Most of those controls are described in Configurable instruction controls, and this section:

- Summarizes those controls.
- Provides additional information in the following subsections:
  - FPEXC control of access to Advanced SIMD and floating-point functionality.
  - EL0 access to Advanced SIMD and floating-point functionality.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This section shows the controls when the controlling Exception levels are using AArch32. Similar controls are provided when the Exception levels are using AArch64, and then apply to lower Exception levels that are using AArch32.

The controls of access to Advanced SIMD and floating-point functionality are:

General {cp10, cp11} or {TCP10, TCP11} controls

This relates to the CPACR.{cp10, cp11}, NSACR.{cp10, cp11}, and HCPTR.{TCP10, TCP11} controls.

Note

Background to the System register interface explains the naming of these controls.

The {cp10, cp11} controls provide general control of the use of Advanced SIMD and floating-point functionality, as follows:

- CPACR.{cp10, cp11} control access from PE modes other than Hyp mode.
  These fields have no effect on accesses to Advanced SIMD and floating-point functionality from Hyp mode.
- In an implementation that includes EL3, NSACR.{cp10, cp11} control access from Non-secure state.
- In an implementation that includes EL2, if NSACR.{cp10, cp11} permit Non-secure accesses, or if EL3 is not implemented, HCPTR.{TCP10, TCP11} provide an additional control on those accesses.

In each case, the {cp10, cp11} controls must be programmed to the same value, otherwise operation is CONSTRAINED UNPREDICTABLE. The CONSTRAINED UNPREDICTABLE behavior is that, for all purposes other than reading the value of the register field, behavior is as if the cp11 field has the same value as the cp10 field. For more information, see Handling of System register control fields for Advanced SIMD and floating-point operation.

For more information about these controls, see:

- FPEXC.
- CPACR.
- HCPTR.
- NSACR.

Control of accesses to the CPACR from Non-secure PL1 modes

As stated in General {cp10, cp11} or {TCP10, TCP11} controls, the CPACR controls access to Advanced SIMD and floating-point functionality from PE modes other than Hyp mode. Accesses to the CPACR from Non-secure PL1 modes can be trapped to EL2.

Additional controls of Advanced SIMD functionality

- If implemented as an RW field, CPACR.ASEDIS can make all Advanced SIMD instructions UNDEFINED in all modes other than Hyp mode.
- In an implementation that includes EL3, when CPACR.ASEDIS permits use of the Advanced SIMD instructions or if the CPACR.ASEDIS control is not implemented, NSACR.NSASEDIS can make all Advanced SIMD instructions UNDEFINED in Non-secure state.
- In an implementation that includes EL2, when the CPACR and NSACR settings permit Non-secure use of the Advanced SIMD instructions, if HCPTR.TASE is implemented as an RW field it can make these instructions UNDEFINED in Hyp mode, and trap to Hyp mode any use of these instructions in a Non-secure PL0 or PL1 mode.

Pseudocode description of enabling SIMD and floating-point functionality provides links to the pseudocode descriptions of all of these controls.

G1.21.2.1 FPEXC control of access to Advanced SIMD and floating-point functionality

In addition, FPEXC.EN is an enable bit for most Advanced SIMD and floating-point operations. When FPEXC.EN is 0, all Advanced SIMD and floating-point instructions are treated as UNDEFINED except for:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A VMSR to the FPEXC or FPSID register.
- A VMRS from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2 register.

These instructions can be executed only at EL1 or higher.

Note

- When the FPSID is accessible, any write access to the FPSID is ignored.
- When FPEXC.EN is 0, these operations are treated as UNDEFINED :
  - A VMSR to the FPSCR.
  - A VMRS from the FPSCR.

When executing at EL0, the PE behaves as if the value of FPEXC.EN is 1 if either:

- EL1 is using AArch64.
- EL2 is enabled in the current Security state and is using AArch64. and the value of HCR_EL2.TGE is 1.

Note

In Non-secure state, if the value of HCR_EL2.RW is 0 then it is permitted for the value of FPEXC32_EL2.EN to control whether Advanced SIMD and floating-point functionality is enabled. However, Arm deprecates using the value of FPEXC32_EL2.EN to determine behavior.

G1.21.2.2 EL0 access to Advanced SIMD and floating-point functionality

When the access controls summarized in this section permit EL0 access to the Advanced SIMD and floating-point functionality, this applies only to the subset of functionality that is available at EL0. In particular:

- Only Advanced SIMD and Floating-point System register that is accessible is the FPSCR.
- The Advanced SIMD and floating-point instructions are available.

Execution at EL0 corresponds to the application level view of the Advanced SIMD and floating-point functionality, as described in Advanced SIMD and floating-point System registers.

G1.21.3 Advanced SIMD and floating-point System registers

AArch32 state provides a common set of System registers for the Advanced SIMD and floating-point functionality. This section gives general information about this set of registers, and indicates where each register is described in detail. It contains the following subsections:

- Register map of the Advanced SIMD and floating-point System registers.
- Accessing the Advanced SIMD and floating-point System registers.

G1.21.3.1 Register map of the Advanced SIMD and floating-point System registers

Table G1-21 shows the register map of the Advanced SIMD and Floating-point registers. Each register is 32 bits wide.

Table G1-21 Floating-point registers

Name	Permitted access
FPEXC	RW
FPSCR	RW
FPSID	RW[a]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Permitted access
MVFR0	RO
MVFR1	RO
MVFR2	RO

a. When FPSID is accessible, VMSR accesses to FPSID are ignored.

In an implementation that includes EL3, the Advanced SIMD and Floating-point registers are common registers, see Common System registers.

G1.21.3.2 Accessing the Advanced SIMD and floating-point System registers

Software accesses the Advanced SIMD and floating-point System registers using the VMRS and VMSR instructions, see:

- VMRS.
- VMSR.

For example:

```
VMRS <Rt>, FPSID    ; Read  Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read  Media and VFP Feature Register 1
VMSR FPSCR, <Rt>    ; Write Floating-Point System Control Register
```

Software can access the Advanced SIMD and floating-point System registers only if the access controls permit the access, see Enabling Advanced SIMD and floating-point support.

Note

All hardware ID information can be accessed only from EL1 or higher. This means:

The FPSID is accessible only from EL1 or higher.
This is a change introduced from VFPv3. Previously, the FPSID register can be accessed in all modes.

The MVFR registers are accessible only from EL1 or higher.
Unprivileged software must issue a system call to determine what features are supported.

G1.21.4 Context switching when using Advanced SIMD and floating-point functionality

When the Advanced SIMD and floating-point functionality is used by only a subset of processes, the operating system might implement lazy context switching of the Advanced SIMD and floating-point register file and System registers.

In the simplest lazy context switch implementation, the primary context switch software uses the CPACR.{cp10, cp11} controls to disable access to the Advanced SIMD and floating-point functionality, see Enabling Advanced SIMD and floating-point support. Subsequently, when a process or thread attempts to use an Advanced SIMD or floating-point instruction, it triggers an Undefined Instruction exception. The operating system responds by saving and restoring the Advanced SIMD and floating-point register file and System registers. Typically, it then re-executes the Advanced SIMD or floating-point instruction that generated the Undefined Instruction exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.22 Configurable instruction controls

This section describes the *Configurable instruction controls* provided by the AArch32 state. For information about the configurable instruction controls provided by the AArch64 state, see Configurable instruction controls.

Each control is categorized as an *instruction enable*, an *instruction disable*, or a *trap control*. The definitions of each type overlap, and in some cases are historical. Each configurable instruction control defines how an exception that is generated as a consequence of the configurable instruction control is handled or reported.

Instruction enables and instruction disables

Enable or disable the use of one or more particular instructions at a particular Privilege level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED.

The exception generated by attempting to execute an UNDEFINED instruction is:

- Taken to EL1 if the UNDEFINED instruction was executed at EL0, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in Routing exceptions from Non-secure EL0 to EL2.
  When the exception is taken to EL1, it is taken to Undefined mode.
- Otherwise, taken to the Exception level at which the UNDEFINED instruction was executed:
  — If the instruction was executed in Hyp mode the exception is taken to Hyp mode.
  — Otherwise, the exception is taken to Undefined mode.

Trap controls

Control whether one or more instructions, when executed at a particular Privilege level, are *trapped*.

Note

AArch32 trap controls are described in terms of Privilege levels, rather than Exception levels, because the PL1 traps apply at and are controlled from:

EL1
In Non-secure state, and in Secure state when EL3 is using AArch64.

EL3
In Secure state when EL3 is using AArch32.

For more information, see Security state, Exception levels, and AArch32 execution privilege.

Trap controls are grouped as:

PL1, excluding Monitor mode

Trapped instructions generate Undefined Instruction exceptions that are taken to Undefined mode, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in Routing exceptions from Non-secure EL0 to EL2.

For more information about these traps, see PL1 configurable controls.

Hyp mode (PL2)

These traps apply only to execution in Non-secure state. This section only describes the traps that apply when EL2 is using AArch32.

Trapped instructions generate:

- Hyp Trap exceptions, taken to Hyp mode, if trapped from a mode other than Hyp mode.
- Undefined Instruction exceptions taken to Hyp mode, if trapped from Hyp mode.

For more information about these traps, see EL2 configurable controls.

See also Routing exceptions from Non-secure EL0 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Monitor mode (Secure PL1)

This section describes only the traps that apply when EL3 is using AArch32.

Trapped instructions generate Monitor Trap exceptions, which are taken to Monitor mode.

For more information about these traps, see EL3 configurable controls.

An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if the instruction does not also generate a higher priority exception. Exception prioritization for exceptions taken to AArch32 state defines the prioritization of different exceptions on the same instruction.

Exceptions generated as a result of these controls are synchronous exceptions.

For exceptions taken to an Exception level that is using AArch32, only exceptions that are taken to Hyp mode are reported in a syndrome register, the HSR.

An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

When a configurable instruction control causes an exception, the exception is taken and the instruction is not executed, and therefore all the following are true:

- The preferred exception return address of the exception is the instruction that generates the exception.
- There are no changes to the registers accessed by the instruction, including as a result of side-effects of a register access.

This section is organized as follows:

- Instructions that fail their Condition code check.
- Instructions that are UNPREDICTABLE.
- Register access instructions.
- PL1 configurable controls.
- EL2 configurable controls.
- EL3 configurable controls.
- Pseudocode description of configurable instruction enables, disables, and traps.

G1.22.1 Instructions that fail their Condition code check

When a configurable instruction control causes a conditional instruction to generate an exception in AArch32 state, it is IMPLEMENTATION DEFINED whether the exception applies to conditional AArch32 instructions that fail their condition code check.

For instructions that fail their Condition code check, see Conditional execution of undefined instructions.

For an instruction that has a Hyp trap or Monitor trap set that fails its Condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Hyp Trap or Monitor Trap exception.
  - Executes as a NOP.

Any implementation must be consistent in its handling of instructions that fail their Condition code check. This means that:

- Whenever a Hyp trap or Monitor trap is set on an instruction it must either:
  - Always generate a Hyp Trap or Monitor Trap exception.
  - Always treat the instruction as a NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The IMPLEMENTATION DEFINED part of the requirements of Conditional execution of undefined instructions must be consistent with the handling of Hyp traps or Monitor traps on instructions that fail their Condition code check. Table G1-22 shows this:

Table G1-22 Consistent handling of instructions that fail their Condition code check

Behavior of conditional undefined instruction[a]	Hyp trap or Monitor trap on instruction that fails its Condition code check[b]
Executes as a NOP	Executes as a NOP
Generates an Undefined Instruction exception	Generates a Hyp Trap or Monitor Trap exception

a. As defined in Conditional execution of undefined instructions. In Non-secure EL0 and EL1 modes, this applies only if no Hyp trap or Monitor trap is set for the instruction, otherwise see the behavior in the other column of the table.
b. For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

G1.22.2 Instructions that are UNPREDICTABLE

It is UNPREDICTABLE / CONSTRAINED UNPREDICTABLE whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed, except in the following cases:

- If the instruction description explicitly states that the configurable instruction control is applied with higher priority than the CONSTRAINED UNPREDICTABLE behavior, then the configurable instruction control generates an exception.
- For an instruction where one of the CONSTRAINED UNPREDICTABLE behaviors is that a register value becomes UNKNOWN, then either:
  - The configurable instruction controls generate an exception.
  - The instruction executes with one of the CONSTRAINED UNPREDICTABLE behaviors that does not cause a register value to become UNKNOWN.

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of configurable instruction controls, but the architecture does not require them to do so.

Note

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in:

- Non-secure state at EL1 or EL0.
- Modes other than Monitor Mode.

This indirectly affects the permitted behavior of UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions. Many CONSTRAINED UNPREDICTABLE behaviors for instructions include an allowance that the CONSTRAINED UNPREDICTABLE instruction behaves the same way as a closely related instruction that is not CONSTRAINED UNPREDICTABLE. In those cases, the instruction enable, disable, or trap control that causes an exception on the closely related instruction will cause the same exception on the CONSTRAINED UNPREDICTABLE instruction.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a Non-secure EL1 or EL0 mode must not generate a Hyp Trap exception or Monitor Trap exception.

G1.22.3 Register access instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means that if the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.22.4 PL1 configurable controls

In AArch32 state, each control is associated with a particular System register field that is accessible:

- When EL3 is using AArch64, or when an implementation does not include EL3, from EL1.
- When EL3 is using AArch32:
  - In Non-secure state, from EL1.
  - In Secure state, from EL3.

This means that the controls are described as PL1 controls, because PL1 is defined as being the Privilege level of software that is executing:

- At EL3, if the PE is executing in EL3 and EL3 is using AArch32.
- At EL1 under all other conditions.

Where there is an AArch64 control that is equivalent to an AArch32 PL1 control, the AArch64 control is an EL1 control.

Any exception that is generated because of an AArch32 PL1 control is taken to a PL1 mode.

Note

Any exception generated because of an AArch32 PL1 control is taken to AArch32 state.

Table G1-23 shows the AArch32 System registers that contain these controls.

Table G1-23 PL1 System registers that contain configurable instruction controls

Register name	Register description
SCTLR	System Control Register
FPEXC	Floating-point Exception Control Register
CPACR	Architectural Feature Access Control Register
DBGDSCRext	Monitor System Debug Control Register
PMUSERENR	Performance Monitors User Enable Register
AMUSERENR	Activity Monitors User Enable Register
CNTKCTL	Counter-timer Kernel Control Register

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in Routing exceptions from Non-secure EL0 to EL2.

G1.22.5 EL2 configurable controls

These controls are ignored in Secure state when using AArch32.

Table G1-24 shows the System registers that contain these controls.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G1-24 EL2 System registers that contain configurable instruction controls

Register name	Register description
FPEXC	Floating-point Exception Control Register
HCR	Hypervisor Configuration Register
HSTR	Hypervisor System Trap Register
HCPTR	Hyp Architectural Feature Trap Register
HDCR	Hyp Debug Control Register
HSCTLR	Hyp System Control Register
HCR2	Hypervisor Configuration Register 2
CNTHCTL	Counter-timer Hyp Control Register

G1.22.6 EL3 configurable controls

Table G1-25 shows the System registers that contain these controls.

Table G1-25 EL3 System registers that contain configurable instruction controls

Register name	Register description
SCR	Secure Configuration Register
NSACR	Non-secure Access Control Register
SDCR	Secure Debug Control Register

G1.22.7 Pseudocode description of configurable instruction enables, disables, and traps

The pseudocode function `AArch32_CheckITEnabled()` checks whether the T32 IT instruction is enabled.

The pseudocode function `AArch32_CheckSETENDEnabled()` checks whether the `SETEND` instruction is disabled.

The pseudocode function for `AArch32_CheckForSMCUndefOrTrap()` checks for traps on an `SMC` instruction.

The `AArch32_CheckForWFxTrap()` pseudocode function checks for traps on `WFE` and `WFI` instructions:

G1.22.7.1 Pseudocode description of enabling SIMD and floating-point functionality

The `AArch32_CheckAdvSIMDOrFPEnabled()` and `AArch32_CheckFPAdvSIMDTrap()` pseudocode functions take appropriate action if an SIMD or floating-point instruction is used when the SIMD and floating-point functionality is not enabled or is trapped.

The `CheckAdvSIMDOrVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` wrapper functions support the `AArch32_CheckAdvSIMDOrFPEnabled()` and `AArch32_CheckFPAdvSIMDTrap()` functions.

The `AArch32_CheckAdvSIMDOrFPEnabled()`, `AArch32_CheckFPAdvSIMDTrap()`, `CheckAdvSIMDOrVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` functions are described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G2
AArch32 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch32 self-hosted debug Exception model. It is organized as follows:

Introductory information

- About self-hosted debug.
- The debug exception enable controls.

The debug Exception model

- Routing debug exceptions.
- Enabling debug exceptions.
- The effect of powerdown on debug exceptions.
- Summary of permitted routing and enabling of debug exceptions.
- Pseudocode description of debug exceptions.

The debug exceptions

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.

Synchronization requirements

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a Context Synchronization event guarantees the effects of the changes:

- Synchronization and debug exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of *debug exceptions*, that are taken using the Exception model described in:

- The AArch64 System Level Programmers' Model, if the exception is taken to AArch64 state.
- The AArch32 System Level Programmers' Model, if the exception is taken to AArch32 state.

This section introduces some terms used in describing self-hosted debug, and then introduces the debug exceptions. See:

- Definition of a debugger in the context of self-hosted debug.
- Context ID and Process ID.

G2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

G2.1.2 Context ID and Process ID

In AArch32 state, the CONTEXTIDR identifies the current *Context ID*, that is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

When using the Long-descriptor translation table format, the CONTEXTIDR has a single field, PROCID, that is defined as the *Process Identifier* (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical when using this translation table format.

When using the Short-descriptor translation table format:

- CONTEXTIDR[31:0] defines the Context ID, that is used for breakpoint and watchpoint matching.
- CONTEXTIDR[31:8] defines the Process ID.
- CONTEXTIDR[7:0] define the ASID. See Global and process-specific translation table entries. This means that, when using the Short-descriptor translation table format, the ASID is always bits[7:0] of the Context ID.

G2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions. The debug exceptions that can be generated in an AArch32 stage 1 translation regime are:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions, generated by hardware breakpoints.
- Watchpoint exceptions, generated by hardware watchpoints.
- Vector Catch exceptions.

Note

In addition, *Software Step exceptions* can be generated in stage 1 of an AArch32 translation regime. However, these are always taken to AArch64 state. Software Step exceptions describes this.

Other than Breakpoint Instruction exceptions, the PE can generate a particular debug exception only if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- Debug exceptions are not disabled from the current Exception level and Security state.

  See Enabling debug exceptions. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
- A debugger has not disabled that particular debug exception.

  All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the DBGDSCRext. See The debug exception enable controls.

Breakpoint Instruction exceptions are always enabled.

Note

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.
For the definition of halting is allowed, see Halting allowed and halting prohibited.

When a debug exception is taken to an Exception level that is using AArch32:

- If the debug exception is a Watchpoint exception, it is taken as a Data Abort exception.
- Otherwise, it is taken as a Prefetch Abort exception.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the T32 and A32 instruction sets is `BKPT #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

PE behavior

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Privilege level and AArch32 mode.
- The current Security state.

For more information, see Breakpoint Instruction exceptions.

Breakpoint exceptions

The architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on one or more of instruction addresses, accesses to any address in an instruction address range, or particular PE contexts.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.

The architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch32 translation regime:

- Address:
  - Address Match.
  - Address Mismatch.

  Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID value held in the CONTEXTIDR.
  - VMID Match. Matches with the VMID value held in the VTTBR.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match or mismatch occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

PE behavior

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Debug State.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see Breakpoint exceptions.

Watchpoint exceptions

The architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000` - `0x101F`.

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a Context matching breakpoint with linking enabled, or it can link to an address breakpoint. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

PE behavior

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Debug State.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see Watchpoint exceptions.

Vector Catch exceptions

These are used to trap exceptions. The architecture provides two forms of vector catch, *address-matching* and *exception-trapping*. Only one form can be implemented.

Whichever form is implemented, a debugger must enable Vector Catch exceptions for one or more exception vectors by programming the DBGVCR. Generation of Vector Catch exceptions is then as follows:

- For the address-matching form, a Vector Catch exception is generated whenever the virtual address of an instruction matches a vector that Vector Catch exceptions are enabled for.
- For the Exception-trapping form, a Vector Catch exception is generated as part of exception entry for exception types that correspond to vectors that Vector Catch exceptions are enabled for.

PE behavior

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If debug exceptions are:

- Enabled, Vector Catch exceptions can be generated.
- Disabled, vector catch is ignored.

For more information, see Vector Catch exceptions.

Table G2-1 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

Table G2-1 PE behavior and pseudocode for each of the debug exceptions

Debug exception	PE behavior if debug exceptions are:		Pseudocode, see
	Enabled	Disabled	
Breakpoint Instruction exception	Takes Prefetch Abort exception	Takes Prefetch Abort exception	`AArch32_SoftwareBreakpoint()`
Breakpoint exception	Takes Prefetch Abort exception[a]	Ignored	Pseudocode description of Breakpoint exceptions taken from AArch32 state
Watchpoint exception	Takes Data Abort exception[a]	Ignored	Pseudocode description of Watchpoint exceptions taken from AArch32 state
Vector Catch exception	Takes Prefetch Abort exception	Ignored	Pseudocode description of Vector Catch exceptions

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Debug State.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.2 Routing debug exceptions

Debug exceptions are usually routed to Abort mode. However, if EL2 is implemented, the routing of debug exceptions depends on the Effective values of HDCR.TDE and HCR.TGE:

If the Effective value of {HDCR.TDE, HCR.TGE} is not {0, 0}

Debug exceptions taken from Non-secure state are routed to Hyp mode.

If EL2 is using AArch64 and FEAT_SEL2 is implemented, debug exceptions taken from Secure EL0 and Secure EL1 may be routed to Secure EL2. For more information, see Routing debug exceptions.

Otherwise

In Non-secure state debug exceptions behave as follows:

- Debug exceptions taken from Non-secure EL1 and Non-secure EL0 are routed to Non-secure Abort mode.
- Breakpoint Instruction exceptions taken from Hyp mode are routed to Hyp mode.
- All other debug exceptions are disabled from Hyp mode.

Note

If EL2 is not implemented, the Effective value of HCR.TGE is 0 and the Effective value of HDCR.TDE is 0.

Table G2-2, Table G2-3, and Table G2-4 show the routing of debug exceptions taken from an Exception level that is using AArch32 to an Exception level that is using AArch32. In these tables:

TDE Means the logical OR of HDCR.TDE and HCR.TGE.

(Hyp mode) Means:

- All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Privilege level.
- Breakpoint Instruction exceptions taken from this Privilege level are taken to Hyp mode.

Table G2-2 Routing when both EL3 and EL2 are implemented

TDE	Target AArch32 mode when executing in:			
	Non-secure:			Secure state
	PL0	PL1	PL2	
0	Non-secure Abort mode	Non-secure Abort mode	(Hyp mode)	Secure Abort mode
1	Hyp mode	Hyp mode	(Hyp mode)	Secure Abort mode

Table G2-3 Routing when EL3 is implemented and EL2 is not implemented

Target AArch32 mode when executing in:	
Non-secure state	Secure state
Non-secure Abort mode	Secure Abort mode

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G2-4 Routing when EL3 is not implemented and EL2 is implemented

TDE	Target AArch32 mode when executing in Non-secure:		
	PL0	PL1	PL2
0	Non-secure Abort mode	Non-secure Abort mode	(Hyp mode)
1	Hyp mode	Hyp mode	(Hyp mode)

G2.2.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level. `DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.3 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

DBGDSCRext.MDBGen, plus an enable control for each breakpoint, DBGBCR<n>.E.

Watchpoint exceptions

DBGDSCRext.MDBGen, plus an enable control for each watchpoint, DBGWCR<n>.E.

Vector Catch exceptions

DBGDSCRext.MDBGen.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See Enabling debug exceptions.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

There is no mechanism to access the extended breakpoints and watchpoints in AArch32 state.

G2.3.1 Enabling debug exceptions

A debug exception can only be taken if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- The debug exception is enabled from the current Privilege level.
- The debug exception is enabled from the current Security state.

Table G2-5 shows when debug exceptions are enabled from the current Privilege level.

Table G2-5 Whether debug exceptions are enabled from the current Privilege level

Current Privilege level	Breakpoint Instruction exceptions	All other debug exceptions
PL2	Enabled	Disabled
PL1	Enabled	Enabled
PL0	Enabled	Enabled

Table G2-6 shows when debug exceptions are enabled from the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G2-6 Whether debug exceptions are enabled from the current Security state

Current Security state	Breakpoint Instruction exceptions	All other debug exceptions
Non-secure	Enabled	Enabled from PL1 and PL0 only.
Secure	Enabled	Depends on SDCR.SPD and SDER.SUIDEN. See Disabling debug exceptions from Secure state.

G2.3.2 Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can enable or disable all debug exceptions taken from Secure PL1 other than Breakpoint Instruction exceptions, by using one of:

- The *Secure Privileged Debug* field, SDCR.SPD, if EL3 is using AArch32.
- The *AArch32 Secure Privileged Debug* field, MDCR_EL3.SPD32, if EL3 is using AArch64.

If debug exceptions are disabled from Secure PL1, software executing at Secure PL1 can set the *Secure User Invasive Debug Enable* bit, SDER.SUIDEN, to 1 to enable all debug exceptions taken from Secure PL0 other than Breakpoint Instruction exceptions.

Note

Breakpoint Instruction exceptions are always enabled.

The architecture does not support disabling debug in Non-secure state.

Note

If the boot software that is executed when reset is deasserted programs SUIDEN and SPD so that all debug exceptions are disabled from Secure state, software operating at EL3 never has to switch any of the debug registers between the Security states.

G2.3.3 Pseudocode description of enabling debug exceptions

`AArch64_GenerateDebugExceptions()` determines whether debug exceptions are enabled from the current Exception level and Security state. `AArch64_GenerateDebugExceptionsFrom()` determines whether debug exceptions are enabled from the specified Exception level and Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.4 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

Debug exceptions other than Breakpoint Instruction exceptions are enabled only if both the OS Lock is unlocked and `DoubleLockStatus()` == FALSE.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.5 Summary of permitted routing and enabling of debug exceptions

Behavior is as follows:

Breakpoint Instruction exceptions

These are always enabled, regardless of the current Privilege level and Security state. Table G2-7 shows the routing of these. In the table, n/a means not applicable.

Table G2-7 Routing of Breakpoint Instruction exceptions

Current Security state	HDCR.TDE[a]:	Target when enabled from:		
		PL0	PL1	PL2
Secure	X	Secure Abort mode[b]	Secure Abort mode[b]	n/a
Non-secure	0	Non-secure Abort mode	Non-secure Abort mode	Hyp mode
	1	Hyp mode	Hyp mode	Hyp mode

a. If EL2 is not implemented, behavior is as if the value of this bit is 0. Otherwise, if the value of HCR.TGE is 1, HDCR.TDE is treated as being 1 other than for a direct read of HDCR.

b. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1.

All other debug exceptions

The enabling and permitted routing is controlled by all of the following:

- SDCR.SPD.
- SDER.SUIDEN.
- HDCR.TDE.
- The IMPLEMENTATION DEFINED authentication interface.

Table G2-8 shows the valid combinations of the values of SDCR.SPD, SDER.SUIDEN, HDCR.TDE, and, in the Auth column, the input from the IMPLEMENTATION DEFINED authentication interface described by the pseudocode function `AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled()`. For each combination, the table shows where debug exceptions are enabled from and where they are taken to.

In the table, n/a means not applicable and a dash, -, means that debug exceptions are disabled from that Exception level.

Table G2-8 Breakpoint, Watchpoint, and Vector Catch exceptions

Debug state	Lock[a]	Current Security state	SPD[b]	Auth [c]	SUIDEN[b]	TDE[d]	Target AArch32 mode when enabled from:		
							PL0	PL1	PL2
Yes	X	X	`0bXX`	X	X	X	-	-	-
No	TRUE	X	`0bXX`	X	X	X	-	-	-
No	FALSE	Secure	`0b00`	FALSE	0	X	-	-	n/a
No	FALSE	Secure	`0b00`	FALSE	1	X	Secure Abort mode[e]	-	n/a
No	FALSE	Secure	`0b00`	TRUE	X	X	Secure Abort mode[e]	Secure Abort mode[e]	n/a

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug state	Lock[a]	Current Security state	SPD[b]	Auth [c]	SUIDEN[b]	TDE[d]	Target AArch32 mode when enabled from:		
							PL0	PL1	PL2
No	FALSE	Secure	`0b10`	X	0	X	-	-	n/a
No	FALSE	Secure	`0b10`	X	1	X	Secure Abort mode[e]	-	n/a
No	FALSE	Secure	`0b11`	X	X	X	Secure Abort mode[e]	Secure Abort mode[e]	n/a
No	FALSE	Non-secure	`0bXX`	X	X	0	Non-secure Abort mode	Non-secure Abort mode	-
No	FALSE	Non-secure	`0bXX`	X	X	1	Hyp mode	Hyp mode	-

a. The value of (OSLSR_EL1.OSLK == 1 || `DoubleLockStatus()`).

b. If EL3 is not implemented, behavior is as if SPD is `0b11` and SUIDEN is `0b0`.

c. See the text that introduces this table for an explanation of the Auth column. An entry of TRUE indicates that the authentication mechanism permits the debug exceptions to be taken to their default target PE mode.

d. If HCR.TGE is 1, this bit is treated as being 1 other than for a direct read of HDCR. If EL2 is not implemented, behavior is as if TDE is 0.

e. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.6 Pseudocode description of debug exceptions

The `AArch32_Abort()` function processes `FaultRecord()`, as described in Abort exceptions, and generates:

- Data Abort exceptions for watchpoints.
- Prefetch Abort exceptions for all other debug exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.7 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch32 translation regime.

Note

When the PE is executing in EL0 using AArch32 and EL1 is using AArch64, it is using the AArch64 EL1&0 translation regime. A T32 or A32 `BKPT` instruction executed at EL0 can generate a Breakpoint Instruction exception that is taken to an Exception level that is using AArch64. For more information about the handling of these exceptions, see Breakpoint Instruction exceptions.

It contains the following subsections:

- About Breakpoint Instruction exceptions.
- Breakpoint instruction in the T32 and A32 instruction sets.
- BKPT instructions as the first instruction in an IT block.
- Exception syndrome information and preferred return address for a BKPT instruction.
- Pseudocode description of Breakpoint Instruction exceptions.

G2.7.1 About Breakpoint Instruction exceptions

A *breakpoint* is an event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint Instruction exceptions, that this section describes, are software breakpoints. Breakpoint exceptions describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

- The current Exception level.
- The current Security state.
- Whether the *debug target Exception level*, EL_D, is using AArch64 or AArch32.

Note

- EL_D is the Exception level that debug exceptions are targeting. See Enabling debug exceptions.
- Debuggers using breakpoint instructions must be aware of the rules for concurrent modification and execution of instructions. See Concurrent modification and execution of instructions.

G2.7.2 Breakpoint instruction in the T32 and A32 instruction sets

The breakpoint instruction, in both instruction sets, is:

- `BKPT #<immediate>`

For details of the instruction encoding, see BKPT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.7.2.1 About whether the BKPT instruction is conditional

In the T32 instruction set, `BKPT` instructions are always unconditional.

In the A32 instruction set:

- If the Condition code field is AL, the `BKPT` instruction is unconditional.
- If the Condition code field is anything other than AL, behavior is CONSTRAINED UNPREDICTABLE, and is one of the following:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP` instruction.
  - The instruction is executed unconditionally.
  - The instruction is executed conditionally.

G2.7.3 BKPT instructions as the first instruction in an IT block

If the first instruction in an IT block is a T32 `BKPT` instruction, then in an implementation that supports the ITD control, if ITD field that applies to the current Exception level is:

0 The `BKPT` instruction generates a Breakpoint Instruction exception.

1 The combination of `IT` instruction and `BKPT` instruction is UNDEFINED. Either the `IT` instruction or the `BKPT` instruction generates an Undefined Instruction exception.

In such an implementation, to ensure consistent behavior when making the first instruction in one or more IT blocks a `BKPT` instruction, the debugger must replace the `IT` instruction.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

HSCTLR.ITD Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD Applies to execution at EL0 or EL1 when EL1 is using AArch32.

SCTLR_EL1.ITD

Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

Note

T32 `BKPT` instructions are always unconditional.

G2.7.4 Exception syndrome information and preferred return address for a BKPT instruction

See the following:

- Exception syndrome information for a Breakpoint Instruction exception.
- Preferred return address for a Breakpoint Instruction exception.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.7.4.1 Exception syndrome information for a Breakpoint Instruction exception

The PE takes a Breakpoint Instruction exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp Trap exception, if it is taken to PL2 because either HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

PL1 Abort mode

The PE sets all of the following:

- DBGDSCRext.MOE to `0b0011`, to indicate a Breakpoint Instruction exception.
- IFSR.FS to the code for a debug, `0b00010`.
- The IFAR with an UNKNOWN value.

PL2 Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-9.
- Sets DBGDSCRext.MOE to `0b0011`, to indicate a Breakpoint Instruction exception.
- Sets the HIFAR to an unknown value.

Table G2-9 Information recorded in the HSR

HSR field	Information recorded	
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`.	
Instruction Length, IL	The PE sets this to: • 0 for a T32 `BKPT` instruction. • 1 for an A32 `BKPT` instruction.	
Instruction Specific Syndrome, ISS	**ISS[24:10]**	RES0.
	ISS[9]	*External Abort type* (EA). The PE sets this to 0.
	ISS[8:6]	RES0.
	ISS[5:0]	*Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`.

Note

For information about how debug exceptions can be routed to PL2, see Routing debug exceptions.

G2.7.4.2 Preferred return address for a Breakpoint Instruction exception

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

G2.7.5 Pseudocode description of Breakpoint Instruction exceptions

`AArch32_SoftwareBreakpoint()` generates a Prefetch Abort exception that is taken from AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.8 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:

- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

This section contains the following subsections:

- About Breakpoint exceptions.
- Breakpoint types and linking of breakpoints.
- Execution conditions for which a breakpoint generates Breakpoint exceptions.
- Breakpoint instruction address comparisons.
- Breakpoint context comparisons.
- Using breakpoints.
- Exception syndrome information and preferred return address for a Breakpoint exception.
- Pseudocode description of Breakpoint exceptions taken from AArch32 state.

G2.8.1 About Breakpoint exceptions

A *breakpoint* is an event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint exceptions are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in Breakpoint Instruction exceptions.

An implementation can include between 2-16 hardware breakpoints. DBGDIDR.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>. This holds a value used for breakpoint matching, that is one of:
  - An instruction virtual address.
  - A Context ID.
- If EL2 is implemented, the *Breakpoint Extended Value Register*, DBGBXVR<n>, that holds a VMID value used for breakpoint matching.

These registers are numbered, so that:

- DBGBCR1, DBGBVR1, and DBGBXVR1 are for breakpoint number one.
- DBGBCR2, DBGBVR2, and DBGBXVR2 are for breakpoint number two.
- . . .
- . . .
- DBGBCR<n>, DBGBVR<n>, and DBGBXVR<n> are for breakpoint number <n>.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<n>.E, is 1.
- The conditions specified in the DBGBCR<n> are met.
- The comparisons with the values held in one or both of the DBGBVR<n> and DBGBXVR<n>, as applicable, are successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its Condition code check.
- The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are

- Enabled, Breakpoint debug events generate Breakpoint exceptions
- Disabled, Breakpoint debug events are ignored.

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions. However, the behavior described also applies if breakpoints are causing entry to Debug state.

The debug exception enable controls describes the enable controls for Breakpoint debug events.

G2.8.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match.
  - An instruction address mismatch.
  - A Context ID match, with the value held in the CONTEXTIDR.
  - A VMID match, with the value held in the VTTBR.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match or an instruction address mismatch.

DBGDIDR.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match or mismatch is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case, the breakpoint is called an *Unlinked breakpoint*.
- Enabled for linking to another breakpoint. In this case, the breakpoint is called a *Linked breakpoint*.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match or mismatch occurs. For example, a debugger might:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Linked Context ID Match breakpoint*.
3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, DBGBCR<n>.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See Specifying the halfword-aligned address that an Address breakpoint matches on.

G2.8.2.1 Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGBCR<n>.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See Other usage constraints for Address breakpoints.
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

  Note

  Multiple Linked watchpoints can also link to a single Linked Context breakpoint. Watchpoint exceptions describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked Address Mismatch breakpoints.
- Multiple Linked watchpoints.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure G2-1 shows an example of permitted breakpoint and watchpoint linking.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

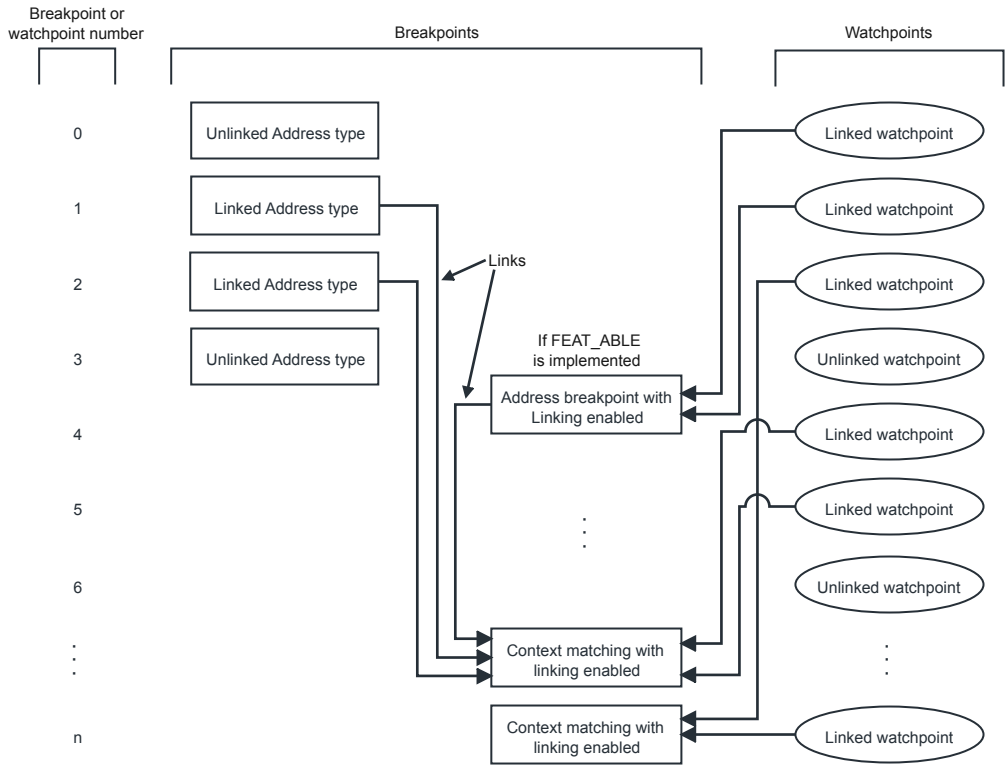

Figure G2-1 The role of linking in Breakpoint and Watchpoint exception generation

In Figure G2-1, each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

G2.8.2.2 Breakpoint types defined by DBGBCRn.BT

The following list provides more detail about each breakpoint type:

`0b0000`**, Unlinked Address Match breakpoint**

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address match, as described in Breakpoint instruction address comparisons.

DBGBCR<n>.{LBNX, LBN} for this breakpoint is ignored.

`0b0001`**, Linked Address Match breakpoint**

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address match defined by this breakpoint, as described in Breakpoint instruction address comparisons.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A successful context match defined by the Context breakpoint with linking enabled that this breakpoint links to.

DBGBCR<n>.{LBNX, LBN} for this breakpoint selects the Context breakpoint with linking enabled that this breakpoint links to.

`0b0010`**, Context ID Match breakpoint**

BT == `0b0010` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.

The value of DBGBVR<n>.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the Effective value of HCR_EL2.E2H is 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR holds the current Context ID.

DBGBCR<n>.{LBNX, LBN, BAS} for this breakpoint are ignored

`0b0011`**, Context ID Match breakpoint with linking enabled**

BT == `0b0011` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, either:

- This breakpoint does not generate any debug exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The value of DBGBVR<n>.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the Effective value of HCR_EL2.E2H is 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR holds the current Context ID.

DBGBCR<n>.{LBNX, LBN, SSC, HMC, BAS PMC} for this breakpoint are ignored.

`0b0100`**, Unlinked Address Mismatch breakpoint**

Generation of a Breakpoint exception depends on both:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address mismatch, as described in Breakpoint instruction address comparisons.

DBGBCR<n>.{LBNX, LBN} for this breakpoint is ignored.

`0b0101`**, Linked Address Mismatch breakpoint**

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address mismatch defined by this breakpoint, as described in Breakpoint instruction address comparisons.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>.{LBNX, LBN} for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

`0b0110`**, CONTEXTIDR_EL1 Match breakpoint**

BT == `0b0110` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR, or CONTEXTIDR_EL1. The value of DBGBVR<n>.ContextID is compared with the Context ID value held in CONTEXTIDR or CONTEXTIDR_EL1.

DBGBCR<n>.{LBNX, LBN, BAS} for this breakpoint are ignored.

`0b0111`**, CONTEXTIDR_EL1 Match breakpoint with linking enabled**

BT == `0b0111` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR, or CONTEXTIDR_EL1. The value of DBGBVR<n>.ContextID is compared with the Context ID value held in CONTEXTIDR or CONTEXTIDR_EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DBGBCR<n>.{LBNX, LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1000`**, VMID Match breakpoint**

BT == `0b1000` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful VMID match, as described in Breakpoint context comparisons.

DBGBCR<n>.{LBNX, LBN, BAS} for this breakpoint are ignored.

`0b1001`**, VMID Match breakpoint with linking enabled**

BT == `0b1000` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, either:

- This breakpoint does not generate any debug exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See Breakpoint instruction address comparisons.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons.

DBGBCR<n>.{LBNX, LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1010`**, Context ID and VMID Match breakpoint**

BT == `0b1010` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-matching breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.
- A successful VMID match.

The value of DBGBVR<n>.ContextID is compared with CONTEXTIDR.

Breakpoint context comparisons describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>.{LBNX, LBN, BAS} for this breakpoint are ignored.

`0b1011`**, Context ID and VMID Match breakpoint with linking enabled**

BT == `0b1011` is a reserved value if either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-matching breakpoints, either:

- This breakpoint does not generate any debug exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on all of the following:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
  - A successful VMID match defined by this breakpoint.

The value of DBGBVR<n>.ContextID is compared with CONTEXTIDR.

Breakpoint context comparisons describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>.{LBNX, LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1100`, **CONTEXTIDR_EL2 Match breakpoint**

BT == `0b1100` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful CONTEXTIDR_EL2 match. The value of DBGBVR<n>.ContextID2 is compared with the Context ID value held in CONTEXTIDR_EL2, as described in Breakpoint context comparisons.

The check against CONTEXTIDR_EL2 means this breakpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>.{LBNX, LBN, BAS} for this breakpoint are ignored.

`0b1101`, **CONTEXTIDR_EL2 Match with linking enabled**

BT == `0b1101` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful CONTEXTIDR_EL2 match. The value of DBGBVR<n>.ContextID2 is compared with the Context ID value held in CONTEXTIDR_EL2, as described in Breakpoint context comparisons.

The check against the CONTEXTIDR_EL2 means the breakpoint or watchpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>.{LBNX, LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1110`**, Full Context ID Match breakpoint**

BT == `0b1110` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.

The Context ID check is made by checking both:

- The value of DBGBVR<n>.ContextID against the value in CONTEXTIDR, or CONTEXTIDR_EL1.
- The value of DBGBXVR<n>.ContextID2 against the value in CONTEXTIDR_EL2.

Both comparisons must match for the check to succeed.

The check against the CONTEXTIDR_EL2 means this breakpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>.{LBNX, LBN, BAS} for this breakpoint are ignored.

`0b1111`**, Full Context ID Match breakpoint with linking enabled**

BT == `0b1111` is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match, as described in Breakpoint context comparisons.

The Context ID check is made by checking both:

- The value of DBGBVR<n>.ContextID against the value in CONTEXTIDR, or CONTEXTIDR_EL1.
- The value of DBGBXVR<n>.ContextID2 against the value in CONTEXTIDR_EL2.

Both comparisons must match for the check to succeed.

The check against the CONTEXTIDR_EL2 means the breakpoint or watchpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

Note

The operation of this breakpoint does not depend on the Effective value of HCR_EL2.E2H.

DBGBCR<n>.{LBNX, LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

Note

See Reserved DBGBCR<n>.BT values for the behavior of breakpoints programmed with reserved BT values.

G2.8.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at PL0 in Secure state.

DBGBCR<n>.{SSC, HMC, PMC} define the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

Higher Mode Control, HMC, and Privileged Mode Control, PMC

HMC and PMC together control which AArch32 modes the breakpoint generates Breakpoint exceptions in.

Table G2-10 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Privilege levels breakpoints generate Breakpoint exceptions in.

In the table:

Y Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row can generate Breakpoint exceptions in AArch32 modes at that Privilege level.

- Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row cannot generate Breakpoint exceptions in AArch32 modes at that Privilege level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A combination in Table G2-10 might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented, see Reserved DBGBCR<n>.{SSC, HMC, PMC} values.

Table G2-10 Summary of breakpoint HMC, SSC, and PMC encodings

HMC	SSC	PMC	Security state the breakpoint is programmed to match in	PL2[a]	PL1	PL0
0	00	00	Both	-	Y[b]	Y
0	00	01		-	Y	-
0	00	10		-	-	Y
0	00	11		-	Y	Y
0	01	00	Non-secure	-	Y[b]	Y
0	01	01		-	Y	-
0	01	10		-	-	Y
0	01	11		-	Y	Y
0	10	00	Secure	-	Y[b]	Y
0	10	01		-	Y	-
0	10	10		-	-	Y
0	10	11		-	Y	Y
0	11	01	Secure	Y	Y	-
0	11	11		Y	Y	Y
1	00	01	Both	Y	Y	-
1	00	11		Y	Y	Y
1	01	00	Non-secure	Y		-
1	01	01		Y	Y	-
1	01	11		Y	Y	Y
1	10	01	Secure	Y	Y	-
1	10	11		Y	Y	Y
1	11	00	Both	Y	-	-
1	11	01		Y	Y	-
1	11	11		Y	Y	Y

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at PL2 using AArch32.

b. Only in User, System and Supervisor modes.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See Table G2-13.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.8.4 Breakpoint instruction address comparisons

Address comparisons are made for each instruction in the program flow. The following subsections describe the criteria for a successful address comparison, for:

- Address Match breakpoints.
- Address Mismatch breakpoints.

G2.8.4.1 Address Match breakpoints

An address match comparison is successful if both:

- Bits [31:2] of the current instruction virtual address are equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS matches. That is, either:
  - DBGBCR<n>.BAS is programmed with `0b0011` or `0b1111`, and the instruction is at a word-aligned address.
  - DBGBCR<n>.BAS is programmed with `0b1100`, and the instruction is not at a word-aligned address.

  See Specifying the halfword-aligned address that an Address breakpoint matches on.

Note

DBGBVR<n>[1:0] are RES0 and are ignored.

G2.8.4.2 Address Mismatch breakpoints

An address mismatch comparison is successful if either:

- Bits [31:2] of the current instruction virtual address are not equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS does not match. That is, either:
  - DBGBCR<n>.BAS is programmed with `0b0011` or `0b1111`, and the instruction is not at a word-aligned address.
  - DBGBCR<n>.BAS is programmed with `0b1100`, and the instruction is at a word-aligned address.

  See Specifying the halfword-aligned address that an Address breakpoint matches on.

Note

- DBGBVR<n>[1:0] are RES0 and are ignored.
- Address Mismatch breakpoints can be used to single-step through code. See Using an Address Mismatch breakpoint to single-step an instruction.

G2.8.4.3 Specifying the halfword-aligned address that an Address breakpoint matches on

For an Address breakpoint, a debugger can use the *Byte Address Selection* field, DBGBCR<n>.BAS, so that the address comparison is successful on one of:

- The whole word starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address ((DBGBVR<n>[31:2]:00) + 2).

Note

The address programmed into the DBGBVR<n> must be word-aligned.

DBGBCR<n>.BAS can be used in both Address Match breakpoints and Address Mismatch breakpoints, as follows:

- For an Address Match breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must generate a Breakpoint exception for. This means that an address comparison is successful only if both of the following match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The halfword defined by the BAS field.

That is, a successful address comparison= DBGBVR<n>[31:2] match AND BAS match.

- For an Address Mismatch breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must not generate a Breakpoint exception for. This means that an address comparison is successful if either or both of the following do not match:
  — The instruction address held in bits [31:2] of the DBGBVR<n>.
  — The halfword defined by the BAS field.

  That is, a successful address comparison = NOT (DBGBVR<n>[31:2] match AND BAS match).

The following subsections show the supported BAS values:

- Using the BAS field in Address Match breakpoints.
- Using the BAS field in Address Mismatch breakpoints.

For Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.

G2.8.4.3.1 Using the BAS field in Address Match breakpoints

The supported BAS values are:

`0b0000` This value is reserved. Behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The breakpoint is disabled.
- The breakpoint behaves as if BAS is `0b0011`, `0b1100`, or `0b1111`.

`0b0011` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

`0b1100` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b10`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for both of the following:

- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction starting at a word-aligned address.

`0b1111` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size a debugger can program breakpoints to match on is a halfword.

Figure G2-2 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 2 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).
- Row 6 shows that a breakpoint with a BAS value of either `0b0011` or `0b1111` generates Breakpoint exceptions for A32 instructions. A32 instructions are always at word-aligned addresses.

In the figure:

Yes Means that the breakpoint generates a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Instruction set	Size	Location of instruction[a]								BAS[3:0]		
			-2	-1	0	+1	+2	+3	+4	+5	0b0011	0b1100	0b1111
Row 1	T32	16-bit			■	■					Yes	No	Yes
Row 2		16-bit					■	■			No	Yes	UNP
Row 3	T32	32-bit	■	■	■	■					UNP	No	UNP
Row 4		32-bit			■	■	■	■			Yes	UNP	Yes
Row 5		32-bit					■	■	■	■	No	Yes	UNP
Row 6	A32	32-bit			■	■	■	■			Yes	UNP	Yes

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:
- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

Figure G2-2 Summary of BAS field meanings for Address Match breakpoints

G2.8.4.3.2 Using the BAS field in Address Mismatch breakpoints

An Address Mismatch breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed to match.

The supported BAS values are:

`0b0000` The breakpoint ignores the address held in the DBGBVR<n> and generates Breakpoint exceptions for all instruction addresses.

`0b0011` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

`0b1100` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b10`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for either of the following:

- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction at a word-aligned address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b1111` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size that a breakpoint can never generate a Breakpoint exception for is a halfword.

Figure G2-3 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 1 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the word-aligned address held in the DBGBVR<n>.
- Row 5 shows that a breakpoint with a BAS value of `0b0011` generates Breakpoint exceptions for 32-bit T32 instructions starting at the halfword-aligned address immediately after the word aligned address held in the DBGBVR<n>.

In the figure:

Yes Means that the breakpoint does generate a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Instruction set	Size	Location of instruction[a]								BAS[3:0]			
			-2	-1	0	+1	+2	+3	+4	+5	0b0000	0b0011	0b1100	0b1111
Row 1	T32	16-bit			■	■					Yes	No	Yes	No
Row 2		16-bit					■	■			Yes	Yes	No	UNP
Row 3	T32	32-bit	■	■	■	■					Yes	UNP	Yes	UNP
Row 4		32-bit			■	■	■	■			Yes	No	UNP	No
Row 5		32-bit					■	■	■	■	Yes	Yes	No	UNP
Row 6	A32	32-bit			■	■	■	■			Yes	No	UNP	No

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:
- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

Figure G2-3 Summary of BAS field meanings for Address Mismatch breakpoints

G2.8.5 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<n>.BT determines what context comparison is required, if any. Table G2-11 shows the BT values that require a comparison, and the match required for the comparison to be successful.

Table G2-11 Breakpoint Context ID and VMID comparison tests

DBGBCR<n>.BT	Test required for successful context comparison
0b001x	• When FEAT_VHE is implemented, EL2 is using AArch64, the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR_EL2 must match the DBGBVR<n>. ContextID value. • Otherwise, CONTEXTIDR must match the DBGBVR<n>.ContextID value.
0b011x	CONTEXTIDR, or CONTEXTIDR_EL1, must match the DBGBVR<n>.ContextID value.
0b100x	VTTBR.VMID must match the DBGBXVR<n>.VMID value.
0b101x	CONTEXTIDR, or CONTEXTIDR_EL1, must match the DBGBVR<n>.ContextID value, and VTTBR.VMID must match the DBGBXVR<n>.VMID value.
0b110x	CONTEXTIDR_EL2 must match the DBGBXVR<n>.ContextID2 value.
0b111x	Both: • CONTEXTIDR, or CONTEXTIDR_EL1, must match the DBGBVR<n>.ContextID value. • CONTEXTIDR_EL2 must match the DBGBXVR<n>.ContextID2 value.

No context comparison is required for other valid DBGBCR<n>.BT values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR, or CONTEXTIDR_EL1, and any of:
  - The PE is executing at EL3 using AArch64.
  - The PE is executing at EL2.
  - FEAT_VHE is implemented, EL2 is using AArch64, EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- The comparison uses the value of CONTEXTIDR_EL2 and any of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Neither FEAT_VHE is implemented, nor FEAT_Debugv8p2 is implemented.
  - EL2 is either not implemented or not enabled in the current Security state.
  - EL2 is using AArch32.
- The comparison uses the current VMID value and any of:
  - EL2 is not implemented.
  - EL2 is either not implemented or not enabled in the current Security state.
  - The PE is executing at EL2.
  - FEAT_VHE is implemented, EL2 is using AArch64, EL2 is implemented and enabled in the current Security state, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Note

- For all Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.
- For Linked Context breakpoints, DBGBCR<n>.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

G2.8.6 Using breakpoints

This section contains the following:

- Using an Address Mismatch breakpoint to single-step an instruction.
- ITD control effects on address breakpoints on the first instruction in an IT block.
- Breakpoint usage constraints.

G2.8.6.1 Using an Address Mismatch breakpoint to single-step an instruction

In execution conditions that an Address Mismatch breakpoint matches, defined by DBGBCR<n>.{LBN, SSC, PMC}, the breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed with. Figure G2-4 shows an example of Address Mismatch breakpoint operation, for an Address Mismatch breakpoint programmed with address `0x1014`.

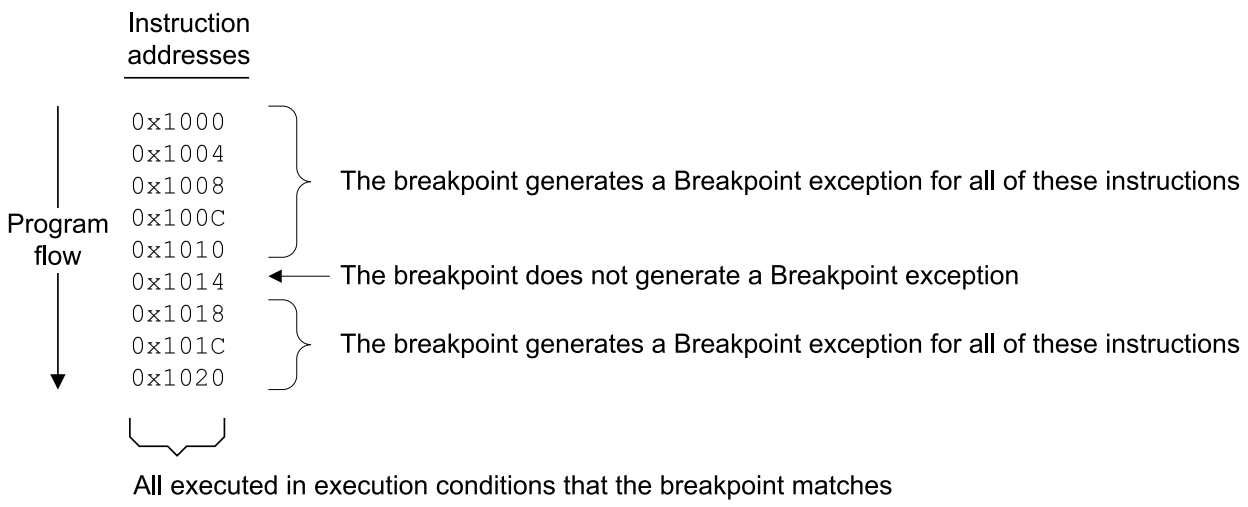

Figure G2-4 Operation of an Address Mismatch breakpoint

This means that an Address Mismatch breakpoint can be used to single-step an instruction.

In the example shown in Figure G2-4:

- If the target of a branch is an instruction other than the instruction at address `0x1014`, the breakpoint generates a Breakpoint exception when the instruction is committed for execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the target of a branch is the instruction at address `0x1014`, the PE executes the instruction at `0x1014` and the breakpoint does not generate a Breakpoint exception until the instruction at address `0x1018` is committed for execution. The instruction at address `0x1014` is therefore single-stepped.

  However, if the instruction at `0x1014` generates a synchronous exception, or if the PE takes an asynchronous exception while the instruction is being stepped, the breakpoint is evaluated again after taking the exception. This means that behavior is as follows:

  — If the exception handler executes in execution conditions that the breakpoint matches, the breakpoint generates a Breakpoint exception for the exception vector, because the exception vector is not address `0x1014`. This means that software execution steps into the exception.

  — If the exception handler executes in execution conditions that the breakpoint does not match, the breakpoint does not generate any Breakpoint exceptions after the PE has taken the exception, until the exception handler completes and executes an exception return instruction. The effect is to step over the exception. Whether the instruction is stepped again depends on whether the target of the exception return instruction is the instruction at `0x1014` or the instruction at `0x1018`.

  If the instruction at `0x1014` is single-stepped and branches to itself, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception after the PE has executed the branch.

This means that an instruction is only single-stepped if it is the target of a branch instruction and its address matches the address the breakpoint is programmed for. In the example shown in Figure G2-4, this is `0x1014`.

Usually this branch instruction is an exception return instruction that changes PE mode, branching from a PE mode in which the breakpoint does not generate a Breakpoint exception. A branch instruction that does not change PE mode would itself generate a Breakpoint exception. However, it might be a branch-to-self instruction as described above.

Because Address Mismatch breakpoints can single-step instructions, the behavior of an address mismatch Breakpoint exception is similar to the behavior of an AArch64 Software Step exception.

Note

- The example shown in Figure G2-4 assumes an A32 instruction. The same behavior applies for both 32-bit and 16-bit T32 instructions.
- Software Step exceptions are the highest priority synchronous exception. Breakpoint exceptions are lower priority. See Prioritization of Synchronous exceptions taken to AArch64 state.

G2.8.6.2 ITD control effects on address breakpoints on the first instruction in an IT block

In an implementation that supports the ITD control, if the value of the ITD field that applies to the current Exception level is 1, all of the following are true:

- An `IT` instruction can only be used to apply to one 16-bit T32 instruction.
- Only certain combinations of an `IT` instruction and second single 16-bit T32 instruction are permitted.
- For a permitted combination, it is IMPLEMENTATION DEFINED whether the implementation treats the combination as:
  — A pair of 16-bit instructions.
  — One 32-bit instruction.

If the implementation treats the combination as one 32-bit instruction, then as described in Other usage constraints for Address breakpoints, an Address breakpoint might not generate a Breakpoint exception for an address match only on the second halfword of the instruction.

For this reason, if the ITD bit associated with the current Exception level is 1, Arm recommends that a debugger that wants to program a breakpoint to match on the second T32 instruction programs it to match on the `IT` instruction instead.

However, if returning from an exception whose preferred return address is the address of the second T32 instruction, then because the debugger is aware that the implementation has treated the combination as a pair of 16-bit instructions, the debugger is permitted to program the breakpoint to match on the second T32 instruction.

The ITD control fields are:

HSCTLR.ITD Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD Applies to execution at EL0 or EL1 when EL1 is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCTLR_EL1.ITD

Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0, and therefore the information in this section does not apply to such an implementation.

Note

Programming the breakpoint to match on the second T32 instruction might be necessary when using an Address Mismatch breakpoint for single stepping.

G2.8.6.3 Breakpoint usage constraints

See the following sections:

- Reserved DBGBCR<n>.BT values.
- Reserved DBGBCR<n>.{SSC, HMC, PMC} values.
- Reserved DBGBCR<n>.BAS values.
- Reserved DBGBCR<n>.LBN values.
- Other usage constraints for Address breakpoints.
- Other usage constraints for Context breakpoints.

G2.8.6.3.1 Reserved DBGBCR<n>.BT values

Table G2-12 shows when particular DBGBCR<n>.BT values are reserved.

Table G2-12 Reserved BT values

BT value	Breakpoint type	Reserved
`0b001x`	Context ID Match	If the breakpoint is not context-aware
`0b010x`	Address Mismatch	If EDSCR.HDE is 1 and halting is allowed
`0b011x`	CONTEXTIDR_EL1 Match	If FEAT_VHE is not implemented, or the breakpoint is not context-aware
`0b100x`	VMID Match	If EL2 is not implemented, or the breakpoint is not context-aware
`0b101x`	Context ID and VMID Match	
`0b110x`	CONTEXTIDR_EL2 Match	If FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, or if the breakpoint is not context-aware For these BT values, breakpoints are not generated if EL2 is using AArch32.
`0b111x`	Full Context ID Match	

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<n>.
- For a direct or external read of DBGBCR<n>, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.8.6.3.2 Reserved DBGBCR<n>.{SSC, HMC, PMC} values

Table G2-13 shows when particular combinations of DBGBCR<n>.{SSC, HMC, PMC} are reserved.

Table G2-13 Reserved HMC, SSC, and PMC combinations

HMC, SSC, and PMC combination	Reserved
All combinations with SSC set to `0b01` or `0b10`.	When EL3 is not implemented and EL2 is implemented.
Any combination where HMC or SSC is nonzero.	When both of EL2 and EL3 are not implemented.
The combinations with SSC set to `0b11` and the combination with HMC set to 1, SSC set to `0b01` and PMC set to `0b00`.	When EL2 is not implemented and EL3 is implemented.
Any combination where HMC or SSC is nonzero.	When RME is not implemented.
The combination with HMC set to 1, SSC set to `0b01` and PMC set to `0b00`, and the combinations with SSC set to 1, SSC set to `0b11` and PMC set to `0b00`.	When both of EL2 and EL3 are implemented, but Secure EL2 is not implemented.
Combinations not included in Table G2-10.	Always.

For all breakpoints except Linked Context breakpoints, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<n>.{SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<n>.{SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Linked Context breakpoint, then:

- The values of HMC, SSC, and PMC are ignored.
- A direct or external read of DBGBCR<n>.{SSC, HMC, PMC} returns UNKNOWN values

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

G2.8.6.3.3 Reserved DBGBCR<n>.BAS values

For all Context breakpoints

DBGBCR<n>.BAS is RES1 and is ignored.

For all Address breakpoints

The supported values of the BAS field for the Address Match and Address Mismatch breakpoints are shown in Specifying the halfword-aligned address that an Address breakpoint matches on.

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BAS value that is not reserved, other than for a direct or external read of DBGBCR<n>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A direct or external read of DBGBCR<n>.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

G2.8.6.3.4 Reserved DBGBCR<n>.LBN values

For all Context breakpoints

DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGBCR<n>.LBN return an unknown value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked Address breakpoint that links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Address breakpoints

DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

G2.8.6.3.5 Other usage constraints for Address breakpoints

For all Address breakpoints

- DBGBVR<n>[1:0] are RES0 and are ignored.
- The DBGBXVR<n> is ignored.

For Address Match breakpoints

- For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
- If DBGBCR<n>.BAS is `0b1111`, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[31:2]:00) + 2). For T32 instructions, Arm recommends that the debugger programs the BAS field with either `0b0011` or `0b1100`.

For Address Mismatch breakpoints

The constraints are the same as those described in For Address Match breakpoints, except that if two Address Mismatch breakpoints are programmed to match in the same Exception level and Security state, it is CONSTRAINED UNPREDICTABLE whether or not the instruction is stepped or a Breakpoint debug even is generated.

G2.8.6.3.6 Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n> and DBGBXVR<n> that are not used to specify Context ID or VMID are RES0 and are ignored.

Note

This means that for Context ID Match breakpoints, the DBGBXVR<n> is RES0 and is ignored, and for VMID Match breakpoints, the DBGBVR<n> is RES0 and is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For Linked Context breakpoints

If no Linked Address breakpoints or Linked Watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

G2.8.7 Exception syndrome information and preferred return address for a Breakpoint exception

See the following:

- Exception syndrome information for a Breakpoint exception.
- Preferred return address for a Breakpoint exception.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

G2.8.7.1 Exception syndrome information for a Breakpoint exception

The PE takes a Breakpoint exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

- DBGDSCRext.MOE to `0b0001`, to indicate a Breakpoint exception.
- IFSR.FS to the code for a debug exception, `0b00010`.
- The IFAR with an UNKNOWN value.

Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-14.
- Sets DBGDSCRext.MOE to `0b0001`, to indicate a Breakpoint exception.
- Sets the HIFAR to an unknown value.

Table G2-14 Information recorded in the HSR

HSR field	Information recorded	
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`.	
Instruction Length, IL	The PE sets this to 1.	
Instruction Specific Syndrome, ISS	**ISS[24:10]**	RES0.
	ISS[9]	*External Abort type* (EA). The PE sets this to 0.
	ISS[8:6]	RES0.
	ISS[5:0]	*Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For information about how debug exceptions can be routed to PL2, see Routing debug exceptions.

G2.8.7.2 Preferred return address for a Breakpoint exception

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

G2.8.8 Pseudocode description of Breakpoint exceptions taken from AArch32 state

`AArch32_BreakpointValueMatch()` returns a pair of results:

- A result for Address Match and Context breakpoints.
- A result for Address Mismatch breakpoints.

`AArch32_StateMatch()` tests the values in DBGBCR<n>.{SSC, HMC, PMC} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

`AArch32_BreakpointMatch()` tests a committed instruction against all breakpoints.

`AArch32_CheckBreakpoint()` generates a `FaultRecord`. A Breakpoint exception is taken if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions.
- All of the conditions required for Breakpoint exception generation are met. See About Breakpoint exceptions.

Note

`AArch32_CheckBreakpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The `AArch32_Abort()` function processes the `FaultRecord` object returned by `AArch32_CheckBreakpoint()`, as described in Abort exceptions. When a Breakpoint exception is taken to AArch32 state, the `AArch32_Abort()` function generates a Prefetch Abort exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.9 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:

- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

This section contains the following subsections:

- About Watchpoint exceptions.
- Watchpoint types and linking of watchpoints.
- Execution conditions for which a watchpoint generates Watchpoint exceptions.
- Watchpoint data address comparisons.
- Watchpoint behavior for certain instruction classes.
- Watchpoint usage constraints.
- Exception syndrome information and preferred return address.
- Pseudocode description of Watchpoint exceptions taken from AArch32 state.

G2.9.1 About Watchpoint exceptions

A *watchpoint* is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, DBGDIDR.WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>. This holds control information for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>. This holds the data virtual address used for watchpoint matching.

The registers are numbered, so that:

- DBGWCR1 and DBGWVR1 are for watchpoint number one.
- DBGWCR2 and DBGWVR2 are for watchpoint number two.
- . . .
- . . .
- DBGWCRn and DBGWVRn are for watchpoint number n.

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Linked Context breakpoint*, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>.BAS, to select the bytes. See Programming a watchpoint with eight bytes or fewer.
- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>.MASK, to program a watchpoint with eight bytes to 2GB. See Programming a watchpoint with eight or more bytes.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint exceptions is CONSTRAINED UNPREDICTABLE. See Programming dependencies of the BAS and MASK fields.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The conditions specified in the DBGWCR<n> are met. This includes all of the following:
  - The watchpoint is enabled. See DBGWCR<n>.E.
  - The type of the access. See DBGWCR<n>.LST.
  - The Security state, Exception level, and privilege of the access. See Execution conditions for which a watchpoint generates Watchpoint exceptions.
  - If the watchpoint links to a breakpoint with linking enabled, the comparison or comparisons made by the breakpoint. See Rules for linking watchpoints.

  Together, these define the *context* of the data access that the watchpoint matches.
- The comparison with the address held in the DBGWVR<n> is successful. See Watchpoint data address comparisons.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.

If multiple watchpoints can generate a Watchpoint debug event for an access, then for all watchpoints that are enabled and match the context of the data access, the Boolean equality comparison results of all the watchpoints are OR'd together to generate a result.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:

- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

Note

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions. However, the behavior described also applies if watchpoints are causing entry to Debug state.

The debug exception enable controls describes the enable controls for Watchpoint debug events.

G2.9.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:

- Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case, the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.
3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the VMID matches.

The *Watchpoint Type* field for a watchpoint, DBGWCR<n>.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<n>.WT is 1, the watchpoint is enabled for linking.

G2.9.2.1 Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGWCR<n>.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>.WT.{SSC, HMC, PAC} for the Linked watchpoint define the execution conditions that the watchpoint generates Watchpoint exceptions for. See Execution conditions for which a watchpoint generates Watchpoint exceptions.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See Watchpoint usage constraints.
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

Note

Multiple Address breakpoints can also link to a single Linked Context breakpoint. Breakpoint exceptions describes breakpoints.

Figure G2-1 shows an example of permitted watchpoint linking.

G2.9.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at EL2.

DBGWCR<n>.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

Higher Mode Control, HMC, and Privileged Access Control, PAC

HMC and PAC together control which Exception levels and Privilege levels the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access and Privilege level at which the access was made.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G2-1

PAC condition for generating a Watchpoint exception

This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at PL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on PL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A32/T32 Load unprivileged and Store unprivileged instructions are `LDRT` and `STRT`.

Table G2-15 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Privilege levels watchpoints generate Watchpoint exceptions in.

In the table:

Y or - Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

Y Can generate Watchpoint exceptions at that Privilege level.

- Cannot generate Watchpoint exceptions at that Privilege level.

Res Means that the combination of HMC, SSC, and PAC is reserved. See Reserved DBGWCR<n>.{SSC, HMC, PAC} values.

Table G2-15 Summary of watchpoint HMC, SSC, and PAC encodings

HMC	SSC	PAC	Security state the watchpoint is programmed to match in	PL2[a]	PL1	PL0	Implementation	
							No EL3	No EL2 and no EL3
0	00	01	Both	-	Y	-	-	-
0	00	10		-	-	Y	-	-
0	00	11		-	Y	Y	-	-
0	01	01	Non-secure	-	Y	-	Res	Res
0	01	10		-	-	Y	Res	Res
0	01	11		-	Y	Y	Res	Res
0	10	01	Secure	-	Y	-	Res	Res
0	10	10		-	-	Y	Res	Res
0	10	11		-	Y	Y	Res	Res
0	11	01	Secure	Y	Y	-	-	Res
0	11	11		Y	Y	Y	-	Res
1	00	01	Both	Y	Y	-	-	Res
1	00	11		Y	Y	Y	-	Res
1	01	00	Non-secure	Y	-	-		
1	01	01		Y	Y	-	Res	Res
1	01	11		Y	Y	Y	Res	Res
1	10	01	Secure	-	Y	-	Res	Res
1	10	11		-	Y	Y	Res	Res

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HMC	SSC	PAC	Security state the watchpoint is programmed to match in	PL2[a]	PL1	PL0	Implementation	
							No EL3	No EL2 and no EL3
1	11	00	Both	Y	-	-	-	Res if no EL2[b]
1	11	01		Y	Y	-		
1	11	11		Y	Y	Y		

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PAC that generate Watchpoint exceptions at PL2 using AArch32.

b. This encoding is only reserved when EL2 is not implemented, regardless of whether EL3 is implemented.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See Reserved DBGWCR<n>.{SSC, HMC, PAC} values.

G2.9.4 Watchpoint data address comparisons

An address comparison is successful if bits [31:2] of the current data virtual address are equal to DBGWVR<n>[31:2], taking into account all of the following:

- The size of the access. See Size of the data access.
- The bytes selected by DBGWVR<n>.BAS. See Programming a watchpoint with eight bytes or fewer.
- Any address ranges indicated by DBGWVR<n>.MASK. See Programming a watchpoint with eight or more bytes.

Note

DBGWVR<n>[1:0] are RES0 and are ignored.

G2.9.4.1 Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each access must be taken into account. See Example G2-2.

Example G2-2

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint must generate a Watchpoint exception.

The size of data accesses initiated by `DCIMVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR.DminLine defines.
  - 2KB.
- A power-of-two.

The lowest address accessed by a `DCIMVAC` instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.

The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, Watchpoint behavior on accesses by DCIMVAC instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.9.4.2 Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<n> the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<n> is:

- Doubleword-aligned:
  - All eight bits of DBGWCR<n>.BAS are used, and the descriptions given in Table G2-16 apply.
- Word-aligned but not doubleword-aligned:
  - Only DBGWCR<n>.BAS[3:0] are used, and the descriptions given in Table G2-17 apply. In this case, DBGWCR<n>.BAS[7:4] are RES0.

Table G2-16 Supported BAS values when the DBGWVRn address alignment is doubleword

BAS value	Description
`0b00000000`	No byte selected.
BAS[0] == 1	Byte at address DBGWVR<n>[31:3]:000 selected.
BAS[1] == 1	Byte at address DBGWVR<n>[31:3]:001 selected.
BAS[2] == 1	Byte at address DBGWVR<n>[31:3]:010 selected.
BAS[3] == 1	Byte at address DBGWVR<n>[31:3]:011 selected.
BAS[4] == 1	Byte at address DBGWVR<n>[31:3]:100 selected.
BAS[5] == 1	Byte at address DBGWVR<n>[31:3]:101 selected.
BAS[6] == 1	Byte at address DBGWVR<n>[31:3]:110 selected.
BAS[7] == 1	Byte at address DBGWVR<n>[31:3]:111 selected.

Table G2-17 Supported BAS values when the DBGWVRn address alignment is word

BAS value[a]	Description
`0b00000000`	No byte selected.
BAS[0] == 1	Byte at address DBGWVR<n>[31:2]:00 selected.
BAS[1] == 1	Byte at address DBGWVR<n>[31:2]:01 selected.
BAS[2] == 1	Byte at address DBGWVR<n>[31:2]:10 selected.
BAS[3] == 1	Byte at address DBGWVR<n>[31:2]:11 selected.

a. DBGWCR<n>.BAS[7:4] are RES0.

If the BAS field selects more than one byte, the bytes it selects must be contiguous. For watchpoint behavior when its BAS field selects non-contiguous bytes, see Other usage constraints.

When programming the BAS field with anything other than `0b11111111`, a debugger must also program DBGWCR<n>.MASK to be `0b00000`. See Programming dependencies of the BAS and MASK fields.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>[31:3]. See Example G2-2.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address `0x1003`, program:
  - DBGWVR<n> with `0x1000`.
  - DBGWCR<n>_EL1.BAS to be `0b00001000`.
- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses `0x2003`, `0x2004` and `0x2005`, program:
  - DBGWVR<n> with `0x2000`.
  - DBGWCR<n>_EL1.BAS to be `0b00111000`.
- If the address programmed into the DBGWVR<n> is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>.BAS to be `0b00001111`.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>[31:3]:100 is accessed, program DBGWCR<n>.BAS to be `0b11110000`.

Note

Arm deprecates programming a DBGWVR<n> with an address that is not doubleword-aligned.

G2.9.4.3 Programming a watchpoint with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>.MASK, to program a single watchpoint with a data address range. The data address range must meet all of the following criteria:

- It is a size that is both:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked.

If *n* least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>[31:*n*]:000. . .
- Address DBGWVR<n>[31:*n*]:111. . .
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>[31:4]:0000 and DBGWVR<n>[31:4]:1111, including these addresses.

Note

- The most significant bit cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see Reserved DBGWCR<n>.MASK values.

When masking address bits, a debugger must both:

- Program DBGWCR<n>.BAS to be `0b11111111`. See Programming dependencies of the BAS and MASK fields.
- In the DBGWVR<n>, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see Other usage constraints.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.9.5 Watchpoint behavior for certain instruction classes

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Prefetch instructions.
- All data cache maintenance instructions except `DCIMVAC`.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an IMPLEMENTATION DEFINED subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache maintenance, address translation, and TLB maintenance instructions are IMPLEMENTATION DEFINED.

The access size of the IMPLEMENTATION DEFINED instruction cache maintenance, address translation, and TLB maintenance instructions that generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also:

- Watchpoint behavior on accesses by Store-Exclusive instructions.
- Watchpoint behavior on accesses by DCIMVAC instructions.

G2.9.5.1 Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

G2.9.5.2 Watchpoint behavior on accesses by DCIMVAC instructions

It is IMPLEMENTATION DEFINED whether `DCIMVAC` operations can generate Watchpoint exceptions. If they can, they are treated as data stores. This means that for a watchpoint to match on an access caused by a `DCIMVAC` instruction, the debugger must program DBGWCR<n>.LSC to be one of the following:

10 Match on data stores only.

11 Match on data stores and data loads.

Note

For the size of data accesses performed by `DCIMVAC` instructions, see Watchpoint data address comparisons. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

G2.9.6 Watchpoint usage constraints

See the following:

- Reserved DBGWCR<n>.{SSC, HMC, PAC} values.
- Reserved DBGWCR<n>.LBN values.
- Programming dependencies of the BAS and MASK fields.
- Reserved DBGWCR<n>.BAS values.
- Reserved DBGWCR<n>.MASK values.
- Other usage constraints.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.9.6.1 Reserved DBGWCR<n>.{SSC, HMC, PAC} values

Table G2-18 shows when particular combinations of DBGWCR<n>.{SSC, HMC, PAC} are reserved.

Table G2-18 Reserved SSC, HMC, and PAC combinations

HMC, SSC, and PAC combination	Reserved
All combinations with SSC set to `0b01` or `0b10`.	When EL3 is not implemented and EL2 is implemented.
Any combination where HMC or SSC is nonzero	When both of EL2 and EL3 are not implemented
The combination with HMC set to 1, SSC set to `0b11`, and PAC set to `0b00`	When EL2 is not implemented
The combinations with SSC set to `0b11` and PAC set to `0b01` or `0b11`	When Secure EL2 is not implemented
The combination with HMC set to 1, SSC set to `0b01` and PAC set to `0b00`	When Secure EL2 is not implemented
Combinations not included in Table G2-15.	Always

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct or external read of DBGWCR<n>.
- For a direct or external read of DBGWCR<n>, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of SSC, HMC, and PAC is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

G2.9.6.2 Reserved DBGWCR<n>.LBN values

For Linked watchpoints

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any DBGWCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGWCR<n>.LBN return an UNKNOWN value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked watchpoints

For Unlinked watchpoints, DBGWCR<n>.LBN reads UNKNOWN and its value is ignored.

G2.9.6.3 Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n> the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is nonzero and a BAS field that is not set to `0b11111111`, then for each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception is generated.

G2.9.6.4 Reserved DBGWCR<n>.BAS values

The BAS field must be programmed with a value `Zeros (8-n-m) : Ones (n) : Zeros (m)`, where:

- `n` is a nonzero positive integer less-than-or-equal-to 8.
- `m` is a positive integer less-than 8.
- `n+m` is less-than-or-equal-to 8.

All other values are reserved.

Note

If `x` is zero, then `Zeros (x)` is an empty bitstring.

If DBGWVR<n>[2] is 1, DBGWCR<n>.BAS[7:4] are RES0 and are ignored.

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>.
- A direct or external read of DBGWCR<n>.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

G2.9.6.5 Reserved DBGWCR<n>.MASK values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`, other than for a direct or external read of DBGWCR<n>.
- A direct or external read of DBGWCR<n>.MASK returns an UNKNOWN value.

G2.9.6.6 Other usage constraints

For all watchpoints:

- DBGWVR<n>[1:0] are RES0 and are ignored.
- If DBGWCR<n>.MASK is nonzero, and any masked bits of DBGWVR<n> are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>.LSC is `0b00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.9.7 Exception syndrome information, fault address information, and preferred return address

See the following:

- Exception syndrome information.
- Fault address information.
- Preferred return address.

G2.9.7.1 Exception syndrome information

The PE takes a Watchpoint exception as either:

- A Data Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

- DBGDSCRext.MOE to `0b1010`, to indicate a Watchpoint exception.
- DFSR.CM to indicate whether a cache maintenance instruction caused the exception.
- DFSR.WnR to indicate whether the exception was generated on a read instruction or a write instruction.

In addition, if using the:

- Short-descriptor format, the PE sets DFSR.FS to the code for a debug exception, `0b00010`, and DFSR.Domain to an UNKNOWN value.
- Long-descriptor format, the PE sets DFSR.STATUS to the code for a debug exception, `0b100010`.

Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-19.
- Sets DBGDSCRext.MOE to `0b1001`, to indicate a Watchpoint exception.

Table G2-19 Information recorded in the HSR

HSR field	Information recorded
Exception Class, EC	The PE sets this to the code for a Data Abort exception routed to Hyp mode, `0x24`.
Instruction Length, IL	The PE sets this to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HSR field	Information recorded	
Instruction Specific Syndrome, ISS	**ISV[24]**	*Instruction Syndrome Valid* (ISV). The PE sets this to 0.
	ISS[23:10]	RES0.
	ISS[9]	*External Abort type* (EA). The PE sets this to 0.
	ISS[8]	*Cache Maintenance* (CM). The PE sets this to indicate whether a cache maintenance instruction caused the exception.
	ISS[7]	RES0 .
	ISS[6]	*Write not Read* (WnR). The PE sets this to indicate whether the exception was generated on a read instruction or a write instruction.
	ISS[5:0]	*Data Fault Status Code* (DFSC). The PE sets this to the code for a debug exception, `0b100010`.

Note

For information about how debug exceptions can be routed to PL2, see Routing debug exceptions.

G2.9.7.2 Fault address information

On a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is one of the following:

- For a Watchpoint taken as an exception to Abort mode, DFAR.
- For a Watchpoint taken as an exception to Hyp mode, EHDFAR.
- For a Watchpoint debug event that causes entry to Debug state, EDWAR.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

Note

If Debug state was entered from AArch32 state, then EDWAR[63:32] is UNKNOWN and must be ignored by the debugger.

In the following subsection, a *watchpointed address* is defined as any address for which at least one of the watchpoints generates a watchpoint match.

G2.9.7.2.1 Address recorded for Watchpoint exceptions generated by data cache maintenance instructions

For a watchpoint triggered by a Data Cache maintenance operation, the address recorded in DFAR, HDFAR, or EDWAR is both:

- Any address accessed by the instruction.
- Within a naturally-aligned block of memory.

G2.9.7.2.2 Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions

The address recorded in DFAR, HDFAR, or EDWAR is both:

- From the inclusive range between:
  - The lowest address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
  - The highest watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.
  - No larger than the DC ZVA block size.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.

The size of the naturally-aligned block of memory is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Example G2-3 Address recorded for a watchpoint programmed on `0x8019`

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte `0x8019`.

An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.

If the `DC ZVA` block size is:

- 32 bytes, the address that the PE records must be between `0x8004` and `0x8019` inclusive.
- 16 bytes, the address that the PE records must be between `0x8010` and `0x8019` inclusive.

G2.9.7.3 Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

G2.9.8 Pseudocode description of Watchpoint exceptions taken from AArch32 state

`AArch32_WatchpointByteMatch()` tests an individual byte accessed by an operation.

`AArch32_StateMatch()` tests the values in DBGWCR<n>.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to.

`AArch32_WatchpointMatch()` tests the value in DBGWVR<n>.

`AArch32_CheckWatchpoint()` generates a `FaultRecord`. A Watchpoint exception is taken if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions.
- All of the conditions required for Watchpoint exception generation are met. See About Watchpoint exceptions.

Note

`AArch32_CheckWatchpoint` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The `AArch32_Abort()` function processes the `FaultRecord` object returned by `AArch32_CheckWatchpoint()`, as described in Abort exceptions. If a Watchpoint exception is taken to AArch32 state, the `AArch32_Abort()` function generates a Data Abort exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.10 Vector Catch exceptions

Arm deprecates the use of vector catch.

This section describes Vector Catch exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:

- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

Note

Vector Catch exceptions cannot be generated when the PE is using an AArch64 translation regime.

This section contains the following subsections:

- About Vector Catch exceptions.
- Exception vectors that Vector Catch exceptions can be enabled for.
- Generation of Vector Catch exceptions.
- Usage constraints.
- Exception syndrome information and preferred return address for a Vector Catch exception.
- Pseudocode description of Vector Catch exceptions.

G2.10.1 About Vector Catch exceptions

Whenever the PE takes an exception, execution is forced to an address that is the *exception vector* for that exception. Vector catch permits a debugger to trap exceptions based on the exception vector, or based on the exception type associated with the exception vector, as follows:

- If the *address-matching* form of vector catch is implemented, the debugger can trap exceptions based on the exception vector.
- If the *exception-trapping* form of vector catch is implemented, the debugger can trap exceptions based on the exception type associated with the exception vector.

The architecture supports only these two forms of vector catch. Only one form can be implemented, and which is implemented is IMPLEMENTATION DEFINED. The DBGDEVID indicates which form is implemented.

Regardless of the form of vector catch implemented, a debugger enables Vector Catch exceptions for exception vectors or types by programming the DBGVCR. This register contains *vector catch enable bits*. Each of these bits corresponds to a different vector. When a debugger sets a vector catch enable bit to 1, Vector Catch exceptions are enabled for the corresponding exception vector or type.

Note

EL2 using AArch64 or EL3 using AArch64 can enable Vector Catch exceptions for vectors by programming the DBGVCR32_EL2. The DBGVCR32_EL2 is architecturally mapped to the DBGVCR.

When Vector Catch exceptions are enabled for an exception vector, this is called an *enabled vector catch*. The set of exception vectors that Vector Catch exceptions are enabled for is called the *enabled vector catch set*.

If the form of vector catch implemented is the:

Address-matching form:

The PE compares the virtual address of each instruction in the program flow with a subset of the enabled vector catch set.

If an address match occurs, a Vector Catch exception is generated when the instruction that caused the match is committed for execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception-trapping form

Whenever the PE takes an exception, if the vector the exception is taken to is included in a subset of the enabled vector catch set, a Vector Catch exception is generated.

The Vector Catch exception is generated as part of entry to the exception, and must be taken before the PE either executes any instructions or takes any further exceptions.

The addresses that comprise the subset depend on whether EL3 is implemented and, for the:

- Address-matching form, the current Security state.
- Exception-trapping form, the Security state that the exception is handled in.

See Generation of Vector Catch exceptions.

Table G2-20 summarizes the differences between the address-matching and exception-trapping forms.

Table G2-20 Differences in behavior of the address-matching and exception-trapping forms of vector catch

Address-matching	Exception-trapping
An enabled vector catch generates a Vector Catch exception when an instruction that is fetched from the vector is committed for execution. This means that spurious Vector Catch exceptions might occur, where the Vector Catch exception does not result from an exception entry, but is instead caused by a branch to the vector. A branch to the vector might occur, for example, on a return from a nested exception or when simulating an exception entry.	An enabled vector catch generates a Vector Catch exception immediately after the PE takes the exception that is associated with the vector. This means that Vector Catch exceptions always result from exception entry, and not from branches to exception vectors.
A Vector Catch exception is generated as a result of an instruction fetch. This means that the Vector Catch exception has a priority relative to the other synchronous exceptions that result from an instruction fetch. Prioritization of Synchronous exceptions taken to AArch64 state describes this prioritization.	A Vector Catch exception is generated as a result of an exception entry. This means that the Vector Catch exception is part of the exception that caused the Vector Catch exception. Therefore, the Vector Catch exception has no priority associated with it. For this reason, Vector Catch exceptions are outside the scope of the prioritization that Prioritization of Synchronous exceptions taken to AArch64 state describes.
A Vector Catch exception can be preempted by another exception. If this happens, the Vector Catch exception is generated again when the exception handler branches back to the vector.	Vector Catch exceptions must be taken before other exceptions.
A Vector Catch exception can be generated as a result of an instruction fetch executed in any AArch32 mode except Hyp mode, including User mode.	Because a Vector Catch exception is generated as the result of an exception entry, the Vector Catch exception is only generated when the PE is in the AArch32 exception handling mode.
If HCR.TGE is 1, Vector Catch exceptions can be generated for User mode instruction fetches from Non-secure PL1 vectors.	If HCR.TGE is 1, Vector Catch exceptions are never generated in Non-secure state, because: • Exceptions are routed away from Non-secure PL1 vectors, to PL2. • The architecture does not provide vector catch enable bits for the Hyp exception vectors.

Depending on the implementation, some vector catch enable bits in the DBGVCR might be RES0. For example, if EL3 is not implemented or is implemented but is using AArch64, Monitor mode is not implemented, and so the enable bits for exception vectors for exceptions taken to Monitor mode are RES0. See Exception vectors that Vector Catch exceptions can be enabled for for the vector catch enable bits that exist for different implementations.

The debug exception enable controls describes the enable controls for Vector Catch exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.10.2 Exception vectors that Vector Catch exceptions can be enabled for

When the PE takes an exception, the exception vector is contained in a *vector table* at the Privilege level the exception is taken to.

Depending on the Security state and AArch32 mode the exception is taken to, when the exception is taken, the vector table used is the table that contains one of:

- *Local exception vectors*.
- *Non-secure Local exception vectors*.
- *Secure Local exception vectors*.
- *Hyp exception vectors*.
- *Monitor exception vectors*.

Table G2-21 shows which vector tables are implemented for different implementations. In the table:

- A dash, -, means that the Exception level is not implemented.
- 64 means that the Exception level is using AArch64.
- 32 means that the Exception level is using AArch32.

Table G2-21 Vector tables implemented for different implementations

Implementation				Vector table or tables implemented
EL0	**EL1**	**EL2**	**EL3**	
32	32	-	-	Local exception vectors.
		64	-	Non-secure Local exception vectors.
		32	-	Non-secure Local exception vectors. Hyp exception vectors.
		-	64	Secure Local exception vectors. Non-secure Local exception vectors.
		-	32	Secure Local exception vectors. Non-secure Local exception vectors. Monitor exception vectors.
		64	64	Secure Local exception vectors. Non-secure Local exception vectors.
		32	64	Secure Local exception vectors. Non-secure Local exception vectors. Hyp exception vectors.
		32	32	Secure Local exception vectors. Non-secure Local exception vectors. Hyp exception vectors. Monitor exception vectors.

For example, in an AArch32-only implementation that includes EL0, EL1, and EL3, when the PE takes an exception to Monitor mode, it uses the vector table containing Monitor exception vectors.

The tables that follow show the vectors that Vector Catch exceptions can be enabled for, and their corresponding vector catch enable bits in the DBGVCR:

- Table G2-22 shows the Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for.
- Table G2-23 shows the Monitor exception vectors that Vector Catch exceptions can be enabled for.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The architecture does not provide vector catch enable bits for the Hyp exception vectors.

Table G2-22 Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for

Vector catch enable bit Local or Secure Local exception vectors	Non-secure Local exception vectors	Exception type	Local exception vectors Normal. SCTLR.V is 0.[a]	High. SCTLR.V is 1.
SF	NSF	FIQ interrupt	VBAR + `0x0000_001C`	`0xFFFF_001C`
SI	NSI	IRQ interrupt	VBAR + `0x0000_0018`	`0xFFFF_0018`
SD	NSD	Data Abort	VBAR + `0x0000_0010`	`0xFFFF_0010`
SP	NSP	Prefetch Abort	VBAR + `0x0000_000C`	`0xFFFF_000C`
SS	NSS	Supervisor Call	VBAR + `0x0000_0008`	`0xFFFF_0008`
SU	NSU	Undefined Instruction	VBAR + `0x0000_0004`	`0xFFFF_0004`

a. If EL3 is implemented and is using AArch32, VBAR is banked. The Secure Local exception vectors use VBARS and the Non-secure Local Exception vectors use VBARNS.

Table G2-23 Monitor exception vectors that Vector Catch exceptions can be enabled for

Vector catch enable bit	Exception type	Monitor exception vectors
MF	FIQ interrupt	MVBAR + `0x0000_001C`
MI	IRQ interrupt	MVBAR + `0x0000_0018`
MD	Data Abort	MVBAR + `0x0000_0010`
MP	Prefetch Abort	MVBAR + `0x0000_000C`
MS	Secure Monitor Call	MVBAR + `0x0000_0008`

Note

There is no vector catch enable bit for Monitor trap exceptions.

G2.10.3 Generation of Vector Catch exceptions

How Vector Catch exceptions are generated depends on which form is implemented:

- Address-matching form.
- Exception-trapping form.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.10.3.1 Address-matching form

The PE compares the virtual address of each instruction in the program flow is with some or all of the addresses in the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE compares the virtual address of each instruction in the program flow, including those executed at EL0, with all addresses in the enabled vector catch set.
- If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  - Monitor exception vectors, if EL3 is using AArch32.
  - Secure Local exception vectors.
  - Non-secure Local exception vectors.

  In this case, Table G2-24 shows which addresses, in the enabled vector catch set, the virtual address of each instruction in the program flow is compared with.

Table G2-24 Comparisons made if the implementation includes EL3

EL3 is using	For exceptions taken to:	
	Secure PL1 modes	Non-secure PL1 modes
AArch64	Secure Local exception vectors	Non-secure Local exception vectors
AArch32	Secure Local exception vectors and Monitor exception vectors	

For example, for exceptions taken to a Secure PL1 mode when EL3 is using AArch64, the virtual address of each instruction in the program flow is compared with each Secure Local exception vector in the enabled vector catch set.

For each instruction in the program flow, the PE tests for any possible Vector Catch exceptions before executing the instruction. If a match occurs, a Vector Catch exception is generated when the instruction is committed for execution, regardless of all of the following:

- Whether the instruction passes its Condition code check.
- Whether the instruction is executed as part of exception entry.
- If EL2 is implemented, what HCR.{IMO, FMO, AMO} are set to.
- If EL3 is implemented, what SCR.{IRQ, FIQ, EA} are set to.

G2.10.3.2 Exception-trapping form

When the PE takes an exception, it tests whether the exception is by branching to an exception vector in a subset of the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE tests whether the exception is by branching to any address in the enabled vector catch set.
- If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  - Monitor exception vectors, if EL3 is using AArch32.
  - Secure Local exception vectors.
  - Non-secure Local exception vectors.

  In this case, the PE tests whether the exception is by branching to a vector in one of the subsets that Table G2-25 shows. In the table, n/a means not applicable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G2-25 Subsets that the PE tests within if EL3 is implemented

EL3 is using	For exceptions taken to:		
	Monitor mode	Other Secure PL1 modes	Non-secure PL1 modes
AArch64	n/a	Secure Local exception vectors	Non-secure Local exception vectors
AArch32	Monitor exception vectors		

For example, for an exception taken to a Secure PL1 mode when EL3 is using AArch64, the PE tests whether the exception is by branching to any of the Secure Local exception vectors in the enabled vector address set.

If the exception is by branching to a vector in the subset, a Vector Catch exception is generated as part of exception entry. That is, a Vector Catch exception is generated instead of the exception handler executing its first instruction.

G2.10.4 Usage constraints

See the following subsections:

- Usage constraints that apply to both forms of vector catch.
- Usage constraints that apply only to the address-matching form.

G2.10.4.1 Usage constraints that apply to both forms of vector catch

For Vector Catch exceptions enabled for either the Prefetch Abort exception vector or the Data Abort exception vector, if one of these exception types is taken to the Exception level that debug exceptions are targeting, behavior is CONSTRAINED UNPREDICTABLE. Either:

- Vector catch is ignored, therefore a Vector Catch exception is not generated.
- Vector catch generates a Prefetch Abort debug exception. For Vector Catch exceptions enabled for the Prefetch Abort exception vector, the PE might enter a recursive loop of Prefetch Abort exceptions causing Vector Catch exceptions and Vector Catch exceptions causing Prefetch Abort exceptions.

Note

The Exception level that debug exceptions are targeting is called the *debug target Exception level*, EL_D. Routing debug exceptions describes how EL_D is derived.

G2.10.4.2 Usage constraints that apply only to the address-matching form

Exception vectors are at word-aligned addresses, and:

- It is CONSTRAINED UNPREDICTABLE whether an enabled vector catch generates a Vector Catch exception for a 32-bit T32 instruction starting at the halfword-aligned address immediately prior to the vector address.
- T32 instructions that start at the halfword-aligned address immediately after the exception vector do not generate Vector Catch exceptions.

For the address-matching form, Vector Catch exceptions have the same priority as Breakpoint exceptions. If a single instruction causes both a Vector Catch exception and a Breakpoint exception, it is CONSTRAINED UNPREDICTABLE which of these debug exceptions the PE takes.

G2.10.5 Exception syndrome information and preferred return address for a Vector Catch exception

See the following:

- Exception syndrome information for a Vector Catch exception.
- Preferred return address for a Vector Catch exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

G2.10.5.1 Exception syndrome information for a Vector Catch exception

The PE takes a Vector Catch exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

PL1 Abort mode

The PE sets all of the following:

- IFSR.FS to the code for a debug exception, `0b00010`.
- DBGDSCRext.MOE to `0b0101`, to indicate a Vector Catch exception.
- The IFAR with an UNKNOWN value.

PL2 Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-26.
- Sets DBGDSCRext.MOE to `0b0101`, to indicate a Vector Catch exception.
- Sets the HIFAR to an unknown value.

Table G2-26 Information recorded in the HSR

HSR field	Information recorded	
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`.	
Instruction Length, IL	The PE sets this to 1.	
Instruction Specific Syndrome, ISS	**ISS[24:10]**	RES0.
	ISS[9]	*External Abort type* (EA). The PE sets this to 0.
	ISS[8:6]	RES0.
	ISS[5:0]	*Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`.

Note

For information about how debug exceptions can be routed to PL2, see Routing debug exceptions.

G2.10.5.2 Preferred return address for a Vector Catch exception

The preferred return address of a Vector Catch exceptions is the address of the instruction that was not executed because the PE took the Vector Catch exception instead.

This means that the preferred return address is the exception vector. This is true regardless of whether the address-matching form or the exception trapping form is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.10.6 Pseudocode description of Vector Catch exceptions

The `AArch32_VCRMatch()` pseudocode function checks whether the instruction at `address` generates a Vector Catch exception. It therefore shows the address-matching form of vector catch.

The `AArch32_CheckVectorCatch()` pseudocode function uses `AArch32_VCRMatch()` to test whether the instruction generates a Vector Catch exception, and if `AArch32_VCRMatch()` returns TRUE it generates that event.

The `AArch32_Abort()` function processes the `FaultRecord` object returned by `AArch32_CheckVectorCatch()`, as described in Abort exceptions. If there is a Vector Catch exception, the `AArch32_Abort()` function generates a Prefetch Abort exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G2.11 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a Context Synchronization event has occurred.

For any instructions executed between the time when the change is made and the time when the next Context Synchronization event occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example G2-4 Example of synchronization and Breakpoint exception generation

1. Software changes DBGDSCRext.MDBGen from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A Context Synchronization event occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example G2-5 Example of synchronization and debug exceptions generation

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A Context Synchronization event occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

Note

For more information, see:

- Synchronization of changes to AArch32 System registers.
- Accessing PSTATE fields.
- Synchronization of changes to the external debug registers.

G2.11.1 State and mode changes without explicit Context Synchronization events

Most changes to the Exception level, and most changes to the Security state if EL3 is implemented, happen as a result of operations that are an explicit Context Synchronization event. This is because taking an exception and returning from an exception are both explicit Context Synchronization events, and the Privilege level and Security state can only change as a result of taking or returning from an exception.

However, some Security state and AArch32 mode changes can happen because of operations that are not an explicit Context Synchronization event. These are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32 mode changes caused by `MSR` and `CPS` instructions. A mode change might be to a mode at a lower Privilege level.
- If EL3 is using AArch32, a Security state change caused by a direct write to the SCR in a privileged mode other than Monitor mode, to set SCR.NS to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G3
AArch32 Self-hosted Trace

This chapter describes the AArch32 self-hosted trace:

Introductory information:

- About self-hosted trace.
- Trace Sinks.
- Register controls to enable self-hosted trace.

Prohibited regions in trace:

- Controls to prohibit trace at Exception levels.
- Self-hosted trace and address translation.

Timestamps and Synchronization:

- Self-hosted trace timestamps.
- Synchronization in self-hosted trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G3.1 About self-hosted trace

A trace unit generates trace data to describe the program flow of the PE.

The trace unit can be an implementation of any of the following:

- In Armv8, a standard Arm Embedded Trace Macrocell (ETM). See *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
- In Armv9, the Embedded Trace Extension (ETE). See The Embedded Trace Extension.
- An IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture trace unit that includes the ETM System register interface, FEAT_TRF must be implemented.

If an Armv9-compliant PE implements FEAT_ETE, FEAT_TRF must be implemented.

If any of the following are true, then Arm recommends that FEAT_TRF is implemented, but this is not mandatory:

- An Armv8.4-compliant PE or Armv9-compliant PE implements an IMPLEMENTATION DEFINED trace unit.
- An Armv8.4-compliant PE implements an ETM Architecture trace unit but does not implement the ETM System register interface.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

The trace unit and the PE must have the same view of the debug authentication interface. If FEAT_TRF is implemented, `ExternalNoninvasiveDebugEnabled()` is always TRUE.

G3.1.1 Trace Sinks

The trace unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

If FEAT_TRBE is implemented, the PE includes a Trace Buffer Unit. Trace data is written directly to memory by the Trace Buffer Unit. See The Trace Buffer Extension.

In Armv8, Arm recommends that a system that includes FEAT_TRF incorporates an ETR, and follows the system architecture described by the *CoreSight Base System Architecture (CS-BSA)*.

G3.1.2 Register controls to enable self-hosted trace

R_{QFYNM} For EL1 using AArch64, see AArch64 Self-hosted Trace.

If FEAT_TRF is implemented, self-hosted trace is enabled if any of the following are true, and disabled otherwise:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, EL3 is implemented, SDCR.STE == 1, and `ExternalSecureNoninvasiveDebugEnabled()` == FALSE.
- EDSCR.TFO ==1, EL3 is not implemented, the PE executes in Secure state, and `ExternalSecureNoninvasiveDebugEnabled()` = FALSE.
- EDSCR.TFO ==1, FEAT_RME is implemented, MDCR_EL3.RLTE == 1, and `ExternalRealmNoninvasiveDebugEnabled()` = FALSE.

The pseudocode function `SelfHostedTraceEnabled()` shows these rules.

I_{QNXVJ} If FEAT_TRF is not implemented, self-hosted trace is disabled.

I_{ZJMLF} While self-hosted trace is disabled, `ExternalNoninvasiveDebugAllowed()` controls whether tracing is prohibited or allowed in each Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{TTFDJ} The self-hosted trace extensions do not provide any mechanism to control software access to the trace unit external debug interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed()` indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

If self-hosted trace is enabled, tracing is prohibited in Secure state when SDCR.STE == 0.

If self-hosted trace is enabled, tracing is prohibited in Realm state when MDCR_EL3.RLTE == 0.

G3.2.1 Controls to prohibit trace at Exception levels

If self-hosted trace is enabled, TRFCR, TRFCR_EL1, TRFCR_EL2 and HTRFCR control whether trace is prohibited at an Exception level. While self-hosted trace is disabled, these registers are ignored.

If self-hosted trace is enabled, tracing is prohibited at EL0 if one of the following is true:

- The Effective value of HCR_EL2.TGE == 0 and TRFCR_EL1.E0TRE == 0.
- The Effective value of HCR.TGE == 0 and TRFCR.E0TRE == 0.
- The Effective value of HCR_EL2.TGE == 1 and TRFCR_EL2.E0HTRE == 0.

If self-hosted trace is enabled, tracing is prohibited at EL1 if TRFCR.E1TRE == 0.

If self-hosted trace is enabled, tracing is prohibited at EL2 if HTRFCR.E2TRE == 0.

If self-hosted trace is enabled, tracing is prohibited at EL3 if one of the following is true:

- EL3 is in AArch64 state.
- EL3 is in AArch32 state and TRFCR.E1TRE == 0.

The pseudocode `TraceAllowed()` shows the preceding rules.

If FEAT_ETE is not implemented, when self-hosted trace is enabled, no events are exported to the trace unit when tracing is prohibited.

If FEAT_ETE is not implemented, when self-hosted trace is disabled, no events are exported to the trace unit when the PE is in Secure state and counting in Secure state is prohibited.

If FEAT_ETE is not implemented, when PMCR_EL0.X==0 or PMCR.X==0, no PMU events are exported to the trace unit.

If FEAT_ETE is implemented, see External inputs for the rules governing export of events to the trace unit.

Otherwise, PMU events are exported to the trace unit.

G3.2.2 Self-hosted trace and address translation

A hypervisor can use HTRFCR.CX to control visibility of VTTBR.VMID.

If self-hosted trace is enabled, and HTRFCR.CX == 0, or if EL2 is not implemented:

- The value of VTTBR.VMID is not traced.
- Comparisons with VTTBR.VMID do not match and results of comparison are not exposed through the comparators.

The trace unit may either prohibit trace for these values, or may record a VTTBR.VMID value of zero in the trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G3.3 Self-hosted trace timestamps

For EL1 using AArch64, see AArch64 Self-hosted Trace.

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While self-hosted trace is disabled, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When self-hosted trace is enabled, the trace time stamp is one of the following:

- Physical time, which is defined by the physical count value returned by `PhysicalCountInt()`.
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of `PhysicalCountInt()` minus a physical offset.
  If any of the following are true the value of the physical offset is zero:
  — EL3 is using AArch32.
  — EL2 is using AArch32.
  — EL2 is not implemented.
  — FEAT_ECV_POFF is not implemented.
  — The Effective value of SCR_EL3.{NSE, NS, RW} is {0, 1, 0}.
  — CNTHCTL_EL2.ECV is 0.
  — EL3 is implemented and using AArch64, and SCR_EL3.ECVEn is 0.
  Otherwise the value of the physical offset is the value of CNTPOFF_EL2.
- Virtual time, which is defined as the value of `PhysicalCountInt()` minus a virtual offset.
  If any of the following are true, the virtual offset is zero:
  — EL2 is not implemented.
  Otherwise, the virtual offset is always CNTVOFF, including when a read of CNTVCT at the current Exception level would treat the virtual offset as zero.

The following fields control which counter is used for self-hosted trace:

- TRFCR_EL2.TS.
- HTRFCR.TS.
- TRFCR.TS.

The timestamp used for trace is shown in Table G3-1.

Table G3-1 Timestamp used for trace.

`SelfHostedTraceEnabled()`	**TRFCR_EL2.TS or HTRFCR.TS**	**TRFCR.TS**	**Timestamp traced**
FALSE	xx	xx	CoreSight time
TRUE	0b00	0b01	`PhysicalCountInt()` - virtual offset
	0b00	0b10	`PhysicalCountInt()` - physical offset
	0b00	0b11	`PhysicalCountInt()`
	0b01	xx	`PhysicalCountInt()` - virtual offset
	0b10[a]	xx	`PhysicalCountInt()` - physical offset
	0b11	xx	`PhysicalCountInt()`

a. Only valid when EL2 is using AArch64 and is the value in TRFCR_EL2.TS.

Note

The counter value used for the trace timestamp is not affected by the Effective value of HCR_EL2.E2H, or whether EL2 is enabled or disabled in the current Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G3.4 Synchronization in self-hosted trace

The trace unit is an indirect observer of the trace control registers.

While self-hosted trace is enabled, indirect reads of the trace filter control fields, TRFCR.{E1TRE, E0TRE} and HTRFCR.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced, and are subject to the standard requirements for synchronization of System register accesses.

The `TSB CSYNC` operation is used to ensure that a trace operation, due to a trace unit generating trace for an instruction has completed. The `TSB CSYNC` operation may be reordered with respect to other instructions, so must be combined with at least one Context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR or HTRFCR is guaranteed to be observed by the trace unit only after a subsequent Context synchronization event. For more information, see Trace Synchronization Barrier.

While self-hosted trace is disabled, the trace unit might impose stronger synchronization requirements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G4
The AArch32 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- About the memory system architecture.
- Address space.
- Mixed-endian support in AArch32.
- AArch32 cache and branch predictor support.
- System register support for IMPLEMENTATION DEFINED memory features.
- External aborts.
- Memory barrier instructions.
- Pseudocode description of general memory System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system, see Memory hierarchy.

G4.1.1 Form of the memory system architecture

The A-profile architecture includes a *Virtual Memory System Architecture* (VMSA). The AArch32 Virtual Memory System Architecture describes the AArch32 view of the VMSA.

G4.1.2 Memory attributes

Memory types and attributes describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table G4-1 shows the memory attributes that are visible at the system level.

Table G4-1 Memory attribute summary

Memory type	Shareability	Cacheability
Device[a]	Outer Shareable.	Non-cacheable.
Normal	One of: • Non-shareable. • Inner Shareable. • Outer Shareable.	One of[b]: • Non-cacheable. • Write-Through Cacheable. • Write-Back Cacheable.

a. Takes additional attributes, see Device memory.

b. See also Cacheability, cache allocation hints, and cache transient hints.

For more information on Cacheability and Shareability, see The Cacheability and Shareability memory attributes, Non-shareable Normal memory, and Caches and memory hierarchy.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.2 Address space

The architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *Physical Address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see About VMSA v8-32.

G4.2.1 Address space overflow or underflow

This subsection describes address space overflow or underflow:

G4.2.1.1 Instruction address space overflow

When a PE performs a normal, sequential execution of instructions, it calculates:

```
address_of_next_instruction = address_of_current_instruction + size_of_executed_instruction
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an A32 or T32 instruction overflows `0xFFFF_FFFF`, the Program Counter becomes UNKNOWN.

If the PE executes an instruction for which the instruction address, size, and alignment mean that it contains the bytes `0xFFFF_FFFF` and `0x0000_0000`, the bytes that apparently from `0x0000_0000` onwards come from an UNKNOWN address.

G4.2.1.2 Data address space overflow and underflow

If the PE executes a load or store instruction for which the calculated address, total access size, and alignment mean that it accesses bytes `0xFFFF_FFFF` and `0x0000_0000`, then the bytes that apparently come from `0x0000_0000` onwards come from UNKNOWN addresses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.3 Mixed-endian support in AArch32

Table G4-2 shows the endianness of explicit data accesses and translation table walks.

Table G4-2 Endianness support

Exception level	Explicit data accesses	Stage 1 translation table walks	Stage 2 translation table walks
EL0	PSTATE.E	SCTLR(S/NS).EE	HSCTLR.EE
EL1	PSTATE.E	SCTLR(S/NS).EE	HSCTLR.EE
EL2	PSTATE.E	HSCTLR.EE	n/a
EL3	PSTATE.E	SCTLR(S).EE	n/a

AArch32 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR(S/NS).EE, HSCTLR.EE, and PSTATE.E are RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES0. PSTATE.E is RW when in EL0 and RES0 when in EL1, EL2, or EL3. SPSR.E is also RES0 when not returning to EL0.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES1. PSTATE.E is RW when in EL0 and RES1 when in EL1, EL2, or EL3. SPSR.E is also RES1 when not returning to EL0.
- All Exception levels support only little-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES0.
- All Exception levels support only big-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1.

Note

- When using AArch32, Arm deprecates PSTATE.E having a different value from the equivalent System register EE bit when in EL1, EL2 or EL3. The use of the `SETEND` instruction is also deprecated.
- If the higher Exception levels are using AArch64, the corresponding registers are:
  - SCTLR_EL1 for SCTLR(NS).
  - SCTLR_EL2 for HSCTLR.
  - SCTLR_EL3 for SCTLR(S).

The `BigEndian()` function determines whether the current Exception level and Execution state is using big-endian data.

For more information about endianness in the Arm architecture, see Endian support.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.4 AArch32 cache and branch predictor support

The following sections describe the support for caches and branch predictors in AArch32 state:

- General behavior of the caches.
- Cache identification.
- Cacheability, cache allocation hints, and cache transient hints.
- Enabling and disabling the caching of memory accesses in AArch32 state.
- Behavior of caches at reset.
- About cache maintenance in AArch32 state.
- AArch32 cache and branch predictor maintenance instructions.
- Execution and data prediction restriction System instructions.
- Cache lockdown.
- System level caches.

See also The AArch32 Virtual Memory System Architecture, and in particular Caches in VMSAv8-32.

Note

- Branch predictors typically use a form of cache to hold branch target data. Therefore, they are included in this section.
- In the instruction mnemonics, MVA is a synonym for VA.

G4.4.1 General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For more information, see The interaction of cache lockdown with cache maintenance instructions.

- Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.
- It is guaranteed that no memory location that does not have a Normal Cacheable attribute is allocated into the cache.
- It is guaranteed that no memory location is allocated to the cache if it has a Normal Non-cacheable attribute or any type of Device memory attribute in both:
  - The translation regime at the current Exception level.
  - The translation regime at any higher Exception level.
- For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all Requesters in its Shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the Shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR.{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

G4.4.2 Cache identification

The cache identification consists of a set of registers that describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes cache maintenance instructions that:

- Affect the entire cache, for example ICIALLUIS.
- Operate by VA, for example ICIMVAU.
- Operate by set/way, for example DCISW.

The cache identification registers are:

- A single Cache Type Register, CTR, that defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instructions.
  - The cache indexing and tagging policy of the Level 1 instruction cache.

  Note

  It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches, see System level caches.

- A single Cache Level ID Register, CLIDR, that defines:
  - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The *Level of Unification Inner Shareable* (LoUIS), *Level of Coherence* (LoC) and the *Level of Unification* (LoU) for the caches. See Terms used in describing the cache maintenance instructions for a definition of these terms.
  - An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.
- A single Cache Size Selection Register, CSSELR, that selects the cache level and sort of cache (Instruction, Data/Unified/Tag) of the current Cache Size Identification Register.
- For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, that defines:
  - The number of sets, associativity, and line length of the cache. See Terms used in describing the cache maintenance instructions for a definition of these terms.

Note

From Armv8.3, it is possible to have multiple Cache Size Identification Registers. For more details, see Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.
3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - An instruction cache.
     - A data or unified cache.
   - Read the Cache Size Identification Register to find details of the cache.

G4.4.2.1 Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2

From Armv8.3, two different formats are available for defining the number of sets and associativity of the currently selected cache. For a definition of these terms, see Terms used in describing the cache maintenance instructions.

When FEAT_CCIDX is implemented:

- There are two Cache Size Identification Registers, CCSIDR and CCSIDR2.
- The length of the CCSIDR.Assoc field is 21 bits. This limits the associativity of the currently selected cache to 2^{21}.
- The length of the CCSIDR2.NumSets field is 24 bits. This limits the number of sets in the currently selected cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.

When FEAT_CCIDX is not implemented:

- There is a single Cache Size Identification Register, CCSIDR.
- The length of the CCSIDR.Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10}.
- The length of the CCSIDR.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to 2^{15}.

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability applies only to Normal memory, and is defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in Memory types and attributes, the memory attributes include a Cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a *cache transient hint* is supported, see Transient Cacheability hint.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be used:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
- For VMSAv8-32 translation table walks using the Long-descriptor translation table format, the appropriate TCR.{IRGN*n*, ORGN*n*} fields define the memory attributes of the translation tables, including the Cacheability. However, this assignment supports only a subset of the Cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

G4.4.3.1 Transient Cacheability hint

It is IMPLEMENTATION DEFINED whether a Transient hint is supported for the VMSAv8-32 translation scheme when using the Long-descriptor translation table format. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Note

The architecture does not specify what is meant by a *relatively short period*.

When using the Short-descriptor translation table format, VMSAv8-32 cannot support the Transient hint.

The description of the MAIR0, MAIR1, HMAIR0, and HMAIR1 registers includes the assignment of the Transient attribute in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

G4.4.4 Enabling and disabling the caching of memory accesses in AArch32 state

Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

In AArch32 state, the Cacheability control fields and their effects are as follows:

For the Non-secure PL1&0 translation regime

The Non-secure instance of SCTLR holds the EL1 controls that affect Cacheability:

- When the value of SCTLR.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 2 translation tables are Non-cacheable.
- When the value of HCR2.ID is 1:
  - All stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR.DC is 1, all Non-secure stage 1 translations and all accesses to the Non-secure EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTLR.C. This applies to translations for both data and instruction accesses.

In addition, when the value of SCTLR.M is 0, indicating that the stage 1 translations are disabled for the translation regime, then if EL2 is using AArch32 and the value of HCR.DC is 0 or if EL2 is using AArch64 and the value of HCR_EL2.DC is 0, then:

- If the value of SCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of SCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- In Non-secure state, the stage 1 and stage 2 Cacheability attributes are combined as described in Combining the Cacheability attribute.
- The Non-secure SCTLR.{C, I} and HCR.DC fields have no effect on the Secure PL1&0 and EL2 translation regimes.
- The HCR2.{ID, CD} fields affect only stage 2 of the Non-secure PL1&0 translation regime.
- In Non-secure state, the PL1&0 translation regime can be described as the Non-secure EL1&0 translation regime. This is consistent with the equivalent AArch64 descriptions.
- When FEAT_XS is implemented SCTLR.{C, I} and HCR2.{ID, CD} fields have no effect on the value of the XS attribute.

For the Secure PL1&0 translation regime

The Secure instance of SCTLR holds the controls that determine Cacheability:

- When the value of SCTLR.C is 0:
  - All data accesses to Normal memory using the Secure PL1&0 translation regime are Non-cacheable.
  - All accesses to the Secure PL1&0 translation tables are Non-cacheable.
- When the value of SCTLR.I is 0:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— All instruction accesses to Normal memory using the Secure PL1&0 translation regime are Non-cacheable.

In addition, when the value of SCTLR.M is 0, indicating that stage 1 translations are disabled, then:

- If the value of SCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of SCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- The Secure SCTLR.{I, C, M} fields have no effect on the Non-secure PL1&0 and EL2 translation regimes.
- When FEAT_XS is implemented, the SCTLR.{I, C} fields have no effect on the value of the XS attribute.

For the EL2 translation regime

- When the value of HSCTLR.C is 0:
  — All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
  — All accesses to the EL2 translation tables are Non-cacheable.
- When the value of HSCTLR.I is 0:
  — All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

In addition, when the value of HSCTLR.M is 0, indicating that stage 1 translations are disabled, then:

- If the value of HSCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of HSCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- The HSCTLR.{I, C, M} fields have no effect on the PL1&0 and EL3 translation regimes.
- When FEAT_XS is implemented, the HSCTLR.{I, C} fields have no effect on the value of the XS attribute.

The effect of the SCTLR.C or HSCTLR.C and HCR2.CD bits is reflected in the result of the address translation instructions in the PAR.

Note

- The requirements in this section mean the architecturally required effects of SCTLR.I and HSCTLR.I are limited to their effects on caching instruction accesses in unified caches.
- This specification can give rise to different Cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in Mismatched memory attributes must be followed to manage the corresponding loss of coherency.

G4.4.5 Behavior of caches at reset

The following rules apply to caches at reset:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in Enabling and disabling the caching of memory accesses in AArch32 state reset to values that force all memory locations to be treated as Non-cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This applies only to the controls that apply to the Translation regime that is used by the Exception level, PE mode, and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

  In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.
- Arm recommends that whenever an invalidation routine is required, it is based on the cache maintenance instructions.

Similar rules apply to:

- Branch predictor behavior, see Behavior of the branch predictors at reset.
- TLB behavior, see TLB behavior at reset.

G4.4.6 About cache maintenance in AArch32 state

The following sections give general information about cache maintenance:

- Terms used in describing the cache maintenance instructions.
- Abstraction of the cache hierarchy.

The following sections describe the AArch32 state cache maintenance instructions:

- AArch32 instruction cache maintenance instructions (IC*).
- AArch32 data cache maintenance instructions (DC*).

Note

Some descriptions of the cache maintenance instructions refer to the Cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR.{I, C} controls.

G4.4.6.1 Terms used in describing the cache maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instructions can be defined:

- By the virtual address of the memory location to be maintained, referred to as operating *by VA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches and branch predictors, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- Terminology for cache maintenance instructions operating by set/way.
- Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see The effects of disabling address translation stages on VMSAv8-32 behavior.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.4.6.1.1 Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR.

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See Memory hierarchy.

Set Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the Arm architecture, sets are numbered from 0.

Way The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

G4.4.6.1.2 Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

Clean A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.

Invalidate A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Clean and Invalidate

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instruction operating by VA, two conceptual points are defined:

Point of Coherency (PoC)

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or Cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

Note

The presence of system caches can affect the determination of the point of coherency as described in System level caches.

Point of Unification (PoU)

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable Shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable Shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

The following fields in the CLIDR relate to these conceptual points:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

— A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.

— Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

LoUU, Level of Unification, uniprocessor

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

— This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain. As with LoC, the LoUIS value is a cache level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If the LoUIS field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain.

— If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

For more information, see the CLIDR description.

G4.4.6.2 Abstraction of the cache hierarchy

The following subsections describe the abstraction of the cache hierarchy:

- Cache maintenance instructions that operate by VA.
- Cache maintenance instructions that operate by set/way.

G4.4.6.2.1 Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being either:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.

See Terms used in describing the cache maintenance instructions for definitions of Point of Coherency and Point of Unification, and more information about possible meanings of VA.

AArch32 cache and branch predictor maintenance instructions lists the VA-based maintenance instructions.

The CTR holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of addresses, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

G4.4.6.2.2 Cache maintenance instructions that operate by set/way

AArch32 cache and branch predictor maintenance instructions lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

G4.4.7 AArch32 cache and branch predictor maintenance instructions

The instruction and data cache maintenance instructions have the same functionality in AArch32 state and in AArch64 state. Table G4-3 shows the AArch32 System instructions. Instructions that take an argument include Rt in the instruction description.

If FEAT_CLRBHB is not implemented, then the architecture does not define any branch predictor maintenance instructions for AArch32 state.

When FEAT_CLRBHB is implemented, the `CLRBHB` instruction is available. When the CLRBHB instruction is executed, the branch history is cleared for the current context to the extent that branch history information created before the CLRBHB instruction cannot be used by code before the CLRBHB instruction to exploitatively control the execution of any code in the current context appearing in program order after the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- In Table G4-3 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.
- In AArch32 state, all of the maintenance instructions are available from EL1 or higher.
- In AArch64 state, branch predictors are always invisible to software, and therefore AArch64 state does not provide any branch predictor maintenance instructions.

Table G4-3 AArch32 System instructions for cache maintenance

Register	Instruction
Instruction cache maintenance instructions	
ICIALLUIS	Invalidate all to Point of Unification, Inner Shareable
ICIALLU	Invalidate all to Point of Unification
ICIMVAU, Rt	Invalidate by virtual address to Point of Unification
Data cache maintenance instructions	
DCIMVAC, Rt	Invalidate by virtual address to Point of Coherency
DCISW, Rt	Invalidate by set/way
DCCMVAC, Rt	Clean by virtual address to Point of Coherency
DCCSW, Rt	Clean by set/way
DCCMVAU, Rt	Clean by virtual address to Point of Unification
DCCIMVAC, Rt	Clean and invalidate by virtual address to Point of Coherency
DCCISW, Rt	Clean and invalidate by set/way
Branch prediction maintenance instructions	
BPIMVA, Rt	Invalidate the virtual address from the branch predictors
BPIALLIS, Rt	Invalidate all entries from branch predictors, Inner Shareable
BPIALL, Rt	Invalidate all entries from branch predictors
CLRBHB	Clear branch history

A `DSB` instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

In an implementation where the branch predictors are architecturally invisible, the BPIMVA, BPIALLIS, and BPIALL instructions can execute as `NOP`s.

The following subsections give more information about these instructions:

- AArch32 instruction cache maintenance instructions (IC*).
- AArch32 data cache maintenance instructions (DC*).
- Branch predictors.
- General requirements for the scope of cache and branch predictor maintenance instructions.
- Effects of instructions that operate by VA to the Point of Coherency.
- Effects of instructions that operate by VA but not to the Point of Coherency.
- Effects of All and set/way maintenance instructions.
- Effects of virtualization and security on the AArch32 cache maintenance instructions.
- Boundary conditions for cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Ordering of cache and branch predictor maintenance instructions.
- Performing cache maintenance instructions.

G4.4.7.1 AArch32 instruction cache maintenance instructions (IC*)

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or at EL1.
- For an instruction executed at EL1, the VMID.

That VA to PA translation might fault. However for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the operation can generate a Data Abort exception for a Translation fault or an Access flag fault.
- The operation cannot generate a Data Abort exception for a Domain fault or a Permission fault, except for the Permission fault case on a Stage 2 fault on a stage 1 translation table walk.

For more information about the possible faults on an instruction that operates by VA, see Types of MMU faults.

An instruction cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the completion requirements for cache and branch predictor maintenance instructions in Completion and endpoint ordering.

See also Ordering of cache and branch predictor maintenance instructions.

G4.4.7.2 AArch32 data cache maintenance instructions (DC*)

Data cache maintenance instructions that take a set/way/level argument take a 32-bit register.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see Out of range values of the Set/Way/Index fields in cache maintenance instructions or the instruction description.

DCISW instructions executed at EL1 perform a clean and invalidate, meaning it performs the same maintenance as a DCCISW instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - EL2 is using AArch32 and the value of HCR.SWIO is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.SWIO is 1.

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or from EL1.
- For an instruction executed from EL1, the VMID.

That VA to PA translation might fault. However a data or unified cache maintenance instruction that operates by VA cannot generate a Data Abort exception for a Domain fault, and cannot generate a Data Abort exception for a Permission fault, except for the Permission fault case on a Stage 2 fault on a stage 1 translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about the possible faults on an instruction that operates by VA, see Types of MMU faults.

DCIMVAC and DCISW instructions executed at EL1 perform a clean and invalidate, meaning they perform the same maintenance as a DCCIMVAC or DCCISW instruction respectively, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- PL1&0 stage two address translation is enabled, meaning either:
  - EL2 is using AArch32 and the value of HCR.VM is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VM is 1.

If a memory fault that sets FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

See also Ordering of cache and branch predictor maintenance instructions.

G4.4.7.3 Branch predictors

In AArch32 state it is IMPLEMENTATION DEFINED whether branch prediction is architecturally visible. This means that under some circumstances software must perform branch predictor maintenance to avoid incorrect execution caused by out-of-date entries in the branch predictor. For example, to ensure correct operation it might be necessary to invalidate branch predictor entries on a change to instruction memory, or a change of instruction address mapping. For more information, see Specific requirements for branch predictor maintenance instructions.

In an implementation where the branch predictors are architecturally invisible, the branch predictor maintenance instructions can execute as NOPs.

An invalidate all operation on the branch predictor ensures that any location held in the branch predictor has no functional effect on execution. An invalidate branch predictor by VA instruction operates on the address of the branch instruction, but can affect other branch predictor entries.

Note

The architecture does not make visible the range of addresses in a branch predictor to which the invalidate operation applies. This means the address used in the invalidate by VA operation must be the address of the branch to be invalidated.

If branch prediction is architecturally visible, an instruction cache invalidate all operation also invalidates all branch predictors.

See also Ordering of cache and branch predictor maintenance instructions.

G4.4.7.3.1 Specific requirements for branch predictor maintenance instructions

If, for a given translation regime and a given ASID and VMID as appropriate, the instructions at any virtual address change, then branch predictor maintenance instructions must be performed to invalidate entries in the branch predictor, to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

- For any translation regime other than the Non-secure PL1&0 translation regime, enabling or disabling stage 1 translations.
- For the Non-secure PL1&0 translation regime:
  - When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
  - When stage 2 translations are disabled, enabling or disabling stage 1 translations.
  - Enabling or disabling stage 2 translations.
- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers, unless:
  - For a change to the Secure PL1&0 translation regime, the change is accompanied by a change to the ASID.
  - For a change to the stage 1 translations of the Non-secure PL1&0 translation regime, the change is accompanied by a change to the ASID or a change to the VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Any change to the VTTBR or VTCR registers, unless accompanied by a change to the VMID.

Note

Invalidation is not required if the changes to the translations are such that the instructions associated with the non-faulting translations of a virtual address, for a given translation regime and a given ASID and VMID, as appropriate, remain unchanged throughout the sequence of changes to the translations. Examples of translation changes to which this applies are:

- Changing a valid translation to a translation that generates an MMU fault.
- Changing a translation that generates an MMU fault to a valid translation.

Failure to invalidate entries might give CONSTRAINED UNPREDICTABLE results, caused by the execution of old branches. For more information, see Ordering of cache and branch predictor maintenance instructions.

Note

- There is no requirement to use the branch predictor maintenance operations to invalidate the branch predictor after:
  - Changing the ContextID or VMID.
  - A cache maintenance instruction that is identified as also flushing the branch predictors, see AArch32 cache and branch predictor maintenance instructions.

Cache maintenance System instructions shows the branch predictor maintenance operations in a VMSA implementation.

G4.4.7.3.2 Behavior of the branch predictors at reset

In AArch32 state:

- If branch predictors are not architecturally invisible:
  - The branch predictors reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
  - The branch predictors are disabled at reset.
- An implementation can require the use of a specific branch predictor initialization routine to invalidate the branch predictor storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
- Arm recommends that whenever an invalidation routine is required, it is based on the AArch32 branch predictor maintenance operations.

Similar rules apply:

- To cache behavior, see Behavior of caches at reset.
- To TLB behavior, see TLB behavior at reset.

G4.4.7.4 General requirements for the scope of cache and branch predictor maintenance instructions

The specification of the cache maintenance and branch predictor instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in General behavior of the caches, Behavior of caches at reset, and Prefetching into cache.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

As a side-effect of a branch predictor maintenance instruction, any entry in the branch predictor might be invalidated.

Note

Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.4.7.5 Effects of instructions that operate by VA to the Point of Coherency

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of other PEs in the Shareability domain described by the Shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable Shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the Point of Coherency.

Table G4-4 PEs affected by cache maintenance instructions to the Point of Coherency

Shareability	PEs affected	Effective to
Non-shareable	The PE performing the operation	The Point of Coherency of the entire system
Inner Shareable	All PEs in the same Inner Shareable Shareability domain as the PE performing the operation	The Point of Coherency of the entire system
Outer Shareable	All PEs in the same Outer Shareable Shareability domain as the PE performing the operation	The Point of Coherency of the entire system

G4.4.7.6 Effects of instructions that operate by VA but not to the Point of Coherency

The following instruction operate by VA but not to the Point of Coherency:

- Clean data or unified cache line by MVA to the Point of Unification, DCCMVAU.
- Invalidate instruction cache line by MVA to Point of Unification, ICIMVAU.
- Invalidate by MVA from branch predictors, BPIMVA.

For these instructions, Table G4-5 shows how, for a VA in a Normal or Device memory location, the Shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table G4-5 PEs affected by cache maintenance instructions to the Point of Unification

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction
Inner Shareable or Outer Shareable	All PEs in the same Inner Shareable Shareability domain as the PE executing the instruction	The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain as the PE executing the instruction

Note

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable Shareability domain containing the PE executing the instruction.

G4.4.7.7 Effects of All and set/way maintenance instructions

The ICIALLU, BPIALL and DC* set/way instructions apply only to the caches and branch predictors of the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLU also performs a BPIALL operation.

The ICIALLUIS and BPIALLIS instructions can affect the caches and branch predictors of all PEs in the same Inner Shareable Shareability domain as the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLUIS also performs a BPIALLIS operation. These instructions have an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The possible presence of system caches, as described in System level caches, means architecture does not guarantee that all levels of cache can be maintained using set/way instructions.

G4.4.7.8 Effects of virtualization and security on the AArch32 cache maintenance instructions

Each Security state has its own physical address space, and therefore cache entries are associated with physical address space. In addition, cache maintenance and branch predictor instructions performed in Non-secure state have to take account of:

- Whether the instruction was performed at EL1 or at EL2.
- For instructions that operate by VA, the current VMID.

Table G4-6 shows the effects of virtualization and security on these maintenance instructions.

Table G4-6 Effects of virtualization and security on the AArch32 cache maintenance instructions

Cache maintenance instructions	Specified entry
Data or unified cache maintenance instructions	
Invalidate, Clean, or Clean and Invalidate by VA: DCIMVAC, DCCMVAC, DCCMVAU, DCCIMVAC	All lines that hold the PA that, in the current translation regime, are mapped to by the combination of all of : • The specified VA. • For an instruction executed at EL1, the current ASID if the location is mapped to by a non-global page. • For a Non-secure instruction executed at EL1, the current VMID[a]. • For a Non-secure instruction executed at EL0, when EL2 is using AArch32 or when EL2 is using AArch64 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the current VMID[a]. • For a Secure instruction executed at EL1, when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, the current VMID [a]. • For a Secure instruction executed at EL0, when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the current VMID[a].
Invalidate, Clean, or Clean and Invalidate by set/way: DCISW, DCCSW, DCCISW	For a Non-secure instruction, the line specified by set/way provided that the entry comes from the Non-secure PA space. For a Secure instruction, the line specified by set/way regardless of the PA space that the entry has come from.
Instruction cache maintenance instructions	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cache maintenance instructions	Specified entry
Invalidate by VA: ICIMVAU	All lines corresponding to the specified VA[b] in the current translation regime and: • For an instruction executed at EL1 or EL0, the current ASID. • For a Non-secure instruction executed at EL1 or EL0, the current VMID[a]. • For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, the current VMID[a]. • For a Secure instruction executed at EL0 when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the current VMID[a].
Invalidate All: ICIALLU, ICIALLUIS	Can invalidate any unlocked entry in the instruction cache, and are required to invalidate: • For a Non-secure instruction executed at EL1, all instruction cache lines containing Non-secure entries associated with the current VMID. • For a Non-secure instruction executed at EL2, all instruction lines containing Non-secure entries. • For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, all instruction cache lines containing entries associated with the current VMID. • For a Secure instruction executed at EL1 when EL3 is using AArch64 and the Effective value of SCR_EL3.EEL2 is 0, all instruction cache lines. • For a Secure instruction executed at EL3 all instruction cache lines.
Branch predictor instructions[c]	
Invalidate by VA: BPIMVA	All lines that, in the current translation regime, are mapped to by the combination of all of: • The specified VA. • For an instruction executed at EL1 or EL0, the current ASID. • For a Non-secure instruction executed at EL1 or EL0, the current VMID[a]. • For a Secure instruction executed at EL1, when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, the current VMID[a]. • For a Secure instruction executed at EL0, when EL3 is using AArch64, SCR_EL3.EEL2 is 1, and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, the current VMID[a].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cache maintenance instructions	Specified entry
Invalidate all: BPIALL, BPIALLIS	Can invalidate any unlocked entry in the branch predictor, and are required to invalidate: • For a Non-secure instruction executed at EL1, all lines containing Non-secure entries associated with the current VMID. • For a Non-secure instruction executed at EL2, all lines containing Non-secure entries. • For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, all lines containing entries associated with the current VMID. • For a Secure instruction executed at EL1 when EL3 is using AArch64 and the Effective value of SCR_EL3.EEL2 is 0, all lines. • For a Secure instruction executed at EL3, all lines.

a. Dependencies on the VMID apply even when either EL2 is using AArch32 and the value of HCR.VM is 0 or EL2 is using AArch64 when enabled for the current Security state, and the value of HCR_EL2.VM is 0. If the PE resets into an Exception level that is using AArch32, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset. However, if the PE resets into an Exception level that is using AArch64, VTTBR_EL2.VMID resets to a value that is architecturally UNKNOWN, and the VTTBR_EL2.VMID field must be set to a known value, that might be zero, as part of the PE initialization sequence.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:

- For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
- For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.

For information of types of instruction cache, see Instruction caches.

c. In an implementation where the branch predictors are architecturally invisible, these instructions can execute as NOPs.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in The interaction of cache lockdown with cache maintenance instructions applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, EL1 invalidate by set/way instructions executed in at EL1 when HCR_EL2.VM or HCR.VM is 1 and EL2 is enabled can, instead, perform a clean and invalidate by set/way.
- The AArch32 Data cache invalidate instructions DCIMVAC and DCISW perform a cache clean as well as a cache invalidate, meaning DCIMVAC performs the same invalidation as a DCCIMVAC instruction, and DCISW performs the same invalidation as a DCCISW instruction, if both of the following apply:
  - EL2 is using AArch32, the value of HCR.VM is 1, and the instruction is executed at Non-secure EL1.
  - EL2 is using AArch64, the value of HCR_EL2.VM is 1, EL2 is enabled, and the instruction is executed at EL1.
- The AArch32 Data cache invalidate by set/way instruction DCISW performs a cache clean as well as a cache invalidate, meaning it performs the same invalidation as a DCCISW instruction, if either of the following apply:
  - EL2 is using AArch32, the value of HCR.SWIO is 1, and the instruction is executed at Non-secure EL1.
  - EL2 is using AArch64, the value of HCR_EL2.SWIO is 1, EL2 is enabled, and the instruction is executed at EL1.
- TLB and instruction cache invalidate instructions are broadcast across the Inner Shareable domain when either:
  - EL2 is using AArch32, the value of HCR.FB is 1, and execution is at Non-secure EL1.
  - EL2 is using AArch64, the value of HCR_EL2.FB is 1, EL2 is enabled, and the instruction is executed at EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL1 is using AArch32, this applies to the TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL, and ICIALLU instructions. This means the instruction performs the invalidation that would be performed by the corresponding Inner Shareable instruction, for example ICIALLU performs the invalidation that would be performed by ICIALLUIS, and BPIALL performs the invalidation that would be performed by BPIALLIS.

For more information about the cache maintenance instructions, see About cache maintenance in AArch32 state, AArch32 cache and branch predictor maintenance instructions, and The AArch32 Virtual Memory System Architecture.

G4.4.7.9 Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instructions operate on the caches regardless of the memory type and Cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  - Is Normal memory or Device memory.
  - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

G4.4.7.10 Ordering of cache and branch predictor maintenance instructions

The following rules describe the effect of the memory order model on the cache and branch predictor maintenance instructions:

- All cache and branch predictor maintenance instructions that do not specify an address execute, relative to each other, in program order.

  All cache and branch predictor instructions that specify an address:
  - Execute in program order relative to all cache and branch predictor operations that do not specify an address.
  - Execute in program order relative to all cache and branch predictor operations that specify the same address.
  - Can execute in any order relative to cache and branch predictor operations that specify a different address.
- Where a cache maintenance or branch predictor instruction appears in program order before a change to the translation tables, the architecture guarantees that the cache or branch predictor maintenance instruction uses the translations that were visible before the change to the translation tables.
- Where a change of the translation tables appears in program order before a cache maintenance or branch predictor instruction, software must execute the sequence outlined in Ordering and completion of TLB maintenance instructions before performing the cache or branch predictor maintenance instruction, to ensure that the maintenance operation uses the new translations.
- A `DMB` instruction causes the effect of all data or unified cache maintenance instructions appearing in program order before the `DMB` to be visible to all explicit memory read and write effects appearing in program order after the `DMB`.

  Also, a `DMB` instruction ensures that the effects of any data or unified cache maintenance instruction appearing in program order before the `DMB` are observable by any observer in the same required Shareability domain before any data or unified cache maintenance or explicit memory operations appearing in program order after the `DMB` are observed by the same observer. Completion of the `DMB` does not guarantee the visibility of all data to other observers. For example, all data might not be visible to a translation table walk, or to instruction fetches.
- A `DSB` is required to guarantee the completion of all cache maintenance instruction that appear in program order before the `DSB` instruction.
- A Context Synchronization event is required to guarantee the effects of any branch predictor maintenance operation. This means a Context Synchronization event causes the effect of all completed branch predictor maintenance operations appearing in program order before the Context Synchronization event to be visible to all instructions after the Context Synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This means that, if a branch instruction appears after an invalidate branch predictor operation and before any Context Synchronization event, it is CONSTRAINED UNPREDICTABLE whether the branch instruction is affected by the invalidate. Software must avoid this ordering of instructions, because it might cause CONSTRAINED UNPREDICTABLE behavior.

- Any data or unified cache maintenance instruction by VA must be executed in program order relative to any explicit memory read or write effect on the same PE to an address covered by the VA of the cache instruction if that load or store is to Normal Cacheable memory. The order of memory accesses that result from the cache maintenance instruction, relative to any other memory accesses to Normal Cacheable memory, are subject to the memory ordering rules. For more information, see Definition of the memory model.

  Any data or unified cache maintenance instruction by VA can be executed in any order relative to any explicit memory read or write effect on the same PE to an address covered by the VA of the cache maintenance instruction if that load or store is not to Normal Cacheable memory.
- There is no restriction on the ordering of data or unified cache maintenance instruction by VA relative to any explicit memory read or write effect on the same PE where the address of the explicit memory read or write effect is not covered by the VA of the cache instruction. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
- There is no restriction on the ordering of a data or unified cache maintenance instruction by set/way relative to any explicit memory read or write effect on the same PE. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
- Software must execute a Context Synchronization event after the completion of an instruction cache maintenance instruction, to guarantee that the effect of the maintenance instruction is visible to any instruction fetch.

A DSB instruction intended to ensure the completion of cache maintenance instructions or branch predictor instructions must have an access type of both loads and stores.

See also the completion requirements for cache and branch predictor maintenance instructions in Completion and endpoint ordering.

The scope of instruction cache maintenance depends on the type of the instruction cache. For more information, see Instruction caches.

Example G4-1 Cache cleaning operations for self-modifying code

The sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
        STR Rt, [Rn]
        DCCMVAU Rn            ; Clean data cache by MVA to point of unification (PoU)
        DSB                   ; Ensure visibility of the data cleaned from cache
        ICIMVAU Rn            ; Invalidate instruction cache by MVA to PoU
        BPIMVA  Rn            ; Invalidate branch predictor by MVA to PoU
        DSB                   ; Ensure completion of the invalidations
        ISB                   ; Synchronize the fetched instruction stream
```

G4.4.7.11 Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

- For data or unified cache maintenance, uses the CTR.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the CTR.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

G4.4.7.11.1 Example code for cache maintenance instructions

The cache maintenance instructions by set/way can be used to clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Note

Because the set/way instructions operate only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Because any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the cache line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

System level caches refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes the current Cache Size Identification Register is in 32-bit format. For more information, see Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2.

```
    MRC p15, 1, R0, c0, c0, 1   ; Read CLIDR into R0
    ANDS R3, R0, #0x07000000
    MOV R3, R3, LSR #23         ; Cache level value (naturally aligned)
    BEQ Finished
    MOV R10, #0
Loop1
    ADD R2, R10, R10, LSR #1    ; Work out 3 x cache level
    MOV R1, R0, LSR R2          ; bottom 3 bits are the Cache type for this level
    AND R1, R1, #7              ; get those 3 bits alone
    CMP R1, #2
    BLT Skip                    ; no cache or only instruction cache at this level
    MCR p15, 2, R10, c0, c0, 0  ; write CSSELR from R10
    ISB                         ; ISB to sync the change to the CCSIDR
    MRC p15, 1, R1, c0, c0, 0   ; read current CCSIDR to R1
    AND R2, R1, #7              ; extract the line length field
    ADD R2, R2, #4              ; add 4 for the line length offset (log2 16 bytes)
    MOV R4, #0x3FF
    ANDS R4, R4, R1, LSR #3     ; R4 is the max number on the way size (right aligned)
    CLZ R5, R4                  ; R5 is the bit position of the way size increment
    MOV R9, R4                  ; R9 working copy of the max way size (right aligned)
Loop2
    MOV R7, #0x00007FFF
    ANDS R7, R7, R1, LSR #13    ; R7 is the max number of the index size (right aligned)
Loop3
    ORR R11, R10, R9, LSL R5    ; factor in the way number and cache number into R11
    ORR R11, R11, R7, LSL R2    ; factor in the index number
    MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
    SUBS R7, R7, #1             ; decrement the index
    BGE Loop3
    SUBS R9, R9, #1             ; decrement the way number
    BGE Loop2
Skip
    ADD R10, R10, #2            ; increment the cache number
    CMP R3, R10
    DSB                         ; ensure completion of previous cache maintenance instruction
    BGT Loop1
Finished
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Similar approaches can be used for all cache maintenance instructions.

G4.4.8 Execution, prefetching and data prediction restriction System instructions

When FEAT_SPECRES is implemented, either alone or alongside FEAT_SPECRES2, the System instructions for prediction restriction listed in Table G4-7 prevent predictions based on information gathered from earlier execution within a particular execution context (CTX), from affecting the later speculative execution within that CTX, to the extent that the speculation execution is observable through side-channels.

The prediction restriction System instructions being used by a particular CTX apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

Table G4-7 Prediction restriction System instructions

	Register	Instruction
FEAT_SPECRES	CFPRCTX	Control Flow Prediction Restriction by Context
	CPPRCTX	Cache Prefetch Prediction Restriction by Context
	DVPRCTX	Data Value Prediction Restriction by Context
FEAT_SPECRES2	COSPRCTX	Clear Other Speculative Prediction Restriction by Context

For these System instructions, the CTX is defined by:

- The Security state.
- The Exception level.
- When executing at EL1, the VMID.
- When executing at EL0 when using the PL1&0 translation regime, the ASID and VMID.

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 applies only to the current hardware defined context.
- Execution of the System instruction at EL1 applies only to the current VMID and Security state, and does not apply to EL2 or EL3.
- Execution of the System instruction at EL2 can apply only to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to a combination of Security state and Exception level that is not implemented, or an Exception level which is higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a `NOP`.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.
- Prediction restriction System instructions are permitted to invalidate more prediction information than is defined by the supplied CTX.

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch32 state, EL0 access to the System instructions is controlled by SCTLR.EnRCTX.

G4.4.9 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

G4.4.9.1 The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DCCSW.
  - Cache invalidate by set/way, DCISW.
  - Cache clean and invalidate by set/way, DCISW.
  - Instruction cache invalidate all, ICIALLU and ICIALLUIS.

  However, one of the following approaches must be adopted in all these cases:
  1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
  2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See Data Abort exception.

  This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, DCCMVAC and DCCMVAU.
- Cache invalidate by virtual address, DCIMVAC.
- Cache clean and invalidate by virtual address, DCCIMVAC.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See DFSR or HSR.

In an implementation that includes EL2, if HCR.TIDCP is set to 1, any exception relating to lockdown of an entry associated with Non-secure memory is routed to EL2.

Note

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.
In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.
The reset setting of the cache must be that no cache entries are locked.

G4.4.9.1.1 Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED space.

G4.4.10 System level caches

The Arm Architecture defines a *system cache* as a cache that is not described in the PE Cache Identification registers, CCSIDR, CCSIDR2, and CLIDR, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.
2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entire system cache must be performed, as is the case for power management, it must be performed using non-architectural mechanisms.
3. System caches which lie beyond the point of coherency and so are invisible to software. The management of such caches is outside the scope of architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.5 System register support for IMPLEMENTATION DEFINED memory features

The VMSAv8-32 defines the following registers for describing IMPLEMENTATION DEFINED features of the memory system:

- The TCM Type Register, TCMTR must be implemented on any implementation where EL1 or above supports AArch32. The format of this register is IMPLEMENTATION DEFINED.
- The System register encoding space with {`coproc` == `0b1111`, `CRn` == `c9`, `CRm` == {`c0`-`c2`, `c5`-`c7`}} is IMPLEMENTATION DEFINED for all values of `opc2` and `opc1`. This space is reserved for branch predictor, cache and TCM functionality, for example maintenance, override behaviors and lockdown.
- In a VMSAv8-32 implementation, part of the System register encoding space with {`coproc` == `0b1111`, `CRn` == `c10`} is IMPLEMENTATION DEFINED and reserved for TLB functionality, see TLB lockdown.
- The System register encoding space with {`coproc` == `0b1111`, `CRn` == `c11`, `CRm` == {`c0`-`c8`, `c15`}} is IMPLEMENTATION DEFINED for all values of `opc2` and `opc1`. This space is reserved for DMA operations to and from the TCMs.

In addition, the System register encoding space with {`coproc` == `0b1111`, `CRn` == `c15`}is reserved for IMPLEMENTATION DEFINED registers, and can provide additional registers for the memory system. For more information, see Organization of registers in the (coproc == `0b1111`) encoding space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.6 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU or Debug hardware. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see RAS PE Architecture.

An External abort is one of:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see Exception terminology.

The RAS Extension provides an expanded taxonomy for describing aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

VMSAv8-32 permits External aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous. The reported fault code identifies whether the External abort is synchronous or asynchronous.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported. Asynchronous External aborts generate SError exceptions.

In AArch32 state:

- SError exceptions are taken as asynchronous Data Abort exceptions.
- Synchronous External aborts:
  - On data accesses are taken as synchronous Data Abort exceptions.
  - On instruction fetches, or prefetches, are taken as synchronous Prefetch Abort exceptions.

See also:

- External abort on a translation table walk.
- Handling exceptions that are taken to an Exception level using AArch32.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running. Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:

- Provision for classification of External aborts.
- Parity or ECC error reporting, FEAT_RAS not implemented.

The section Exception reporting in a VMSAv8-32 implementation describes the reporting of External aborts.

G4.6.1 Provision for classification of External aborts

For an External abort taken to a privileged mode other than Hyp mode, an implementation can use the DFSR.ExT or IFSR.ExT bits to provide more information about the External abort:

- DFSR.ExT provides an IMPLEMENTATION DEFINED classification of External aborts on data accesses.
- IFSR.ExT provides an IMPLEMENTATION DEFINED classification of External aborts on instruction accesses.

For an External abort taken to Hyp mode, the HSR.EA bit, provides an IMPLEMENTATION DEFINED classification of External aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all aborts other than External aborts these bits return a value of 0.

If the FEAT_RAS is implemented:

- The HSR.AET field provides information about the state of the PE following an SError exception taken to Hyp mode.
- The DFSR.AET field provides information about the state of the PE following an asynchronous Data Abort exception.
- The implementation might define error record registers.
- DFSR.{AET, ExT} is set to values from VDFSR or VSESR_EL2 in the case of a virtual SError exception.

For more information on the RAS Extension, see RAS PE Architecture.

G4.6.2 Parity or ECC error reporting, FEAT_RAS not implemented

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache systems. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see Exception reporting in a VMSAv8-32 implementation. However when parity or ECC error reporting is implemented it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code, or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.7 Memory barrier instructions

Memory barriers describes the memory barrier instructions. This section describes the system level controls of those instructions.

G4.7.1 EL2 control of the maintenance scope of Data Synchronization Barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 and that supports maintenance scope limitations on the `DSB` instructions, the HCR.BSU field can modify the required maintenance scope of an instruction that is executed at EL0 or EL1 in Non-secure state. Table G4-8 shows the encoding of this field:

Table G4-8 EL2 control of DSB maintenance scope, for constraining both loads and stores executed at EL0 or EL1

HCR.BSU	Minimum maintenance scope of barrier instructions
00	No effect, maintenance scope is as specified by the instruction
01	Inner Shareable
10	Outer Shareable
11	Full system

For an instruction executed at EL0 or EL1 in Non-secure state, Table G4-9 shows how the HCR.BSU is combined with the maintenance scope specified by the argument of the `DSB` to both loads and stores to give the scope of the instruction.

Table G4-9 Effect of the HCR.BSU on barrier instructions executed at Non-secure EL0 or EL1

Maintenance scope specified by the `DSB` argument	HCR.BSU	Resultant maintenance scope
Full system	Any	Full system
Outer Shareable	`00`, `01`, or `10`	Outer Shareable
	`11`, Full system	Full system
Inner Shareable	`00` or `01`	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system
Non-shareable	`00`, No effect	Non-shareable
	`01`, Inner Shareable	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.8 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- Memory data type definitions.
- Basic memory access.
- Aligned memory access.
- Unaligned memory access.
- Exclusives monitors operations.
- Access permission checking.
- Abort exceptions.
- Memory barriers.

G4.8.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the `AddressDescriptor` type.
- Full address, defined by the `FullAddress` type.
- Memory attributes, defined by the `MemoryAttributes` type.
- Memory type, defined by the `MemType` enumeration.
- Device memory type, defined by the `DeviceType` enumeration.
- Normal memory attributes, defined by the `MemAttrHints` type.
- Cacheability attributes, defined by the `MemAttr_NC`, `MemAttr_WT`, and `MemAttr_WB` constants.
- Allocation hints, defined by the `MemHint_No`, `MemHint_WA`, `MemHint_RA`, and `MemHint_RWA` constants.
- Access permissions, defined by the `Permissions` type.

G4.8.2 Basic memory access

The `PhysMemRead()` and `PhysMemWrite()` functions perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

The attributes in `memaddrdesc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in Memory types and attributes, Definition of the memory model, and Atomicity in the Arm architecture.

G4.8.3 Aligned memory access

The `AArch32_MemSingle()` functions make atomic, little-endian accesses of `size` bytes.

G4.8.4 Unaligned memory access

See Unaligned data access for details of the SCTLR.A and HSCTLR.A controls on the generation of alignment faults. The HSCTLR control applies to Normal memory accesses from Hyp mode, and the SCTLR control applies to Normal memory accesses from all other modes.

The `Mem_with_type()` functions make an access of the required type. If that access is naturally aligned, each form of the function performs an atomic access by making a single call to `AArch32_MemSingle()`. If that access is not aligned but passes the `IsAlignedSize()` checks, each form of the function synthesizes the required access from multiple calls to `AArch32_MemSingle()`. It also reverses the byte order if the access is big-endian.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.8.5 Exclusives monitors operations

The `AArch32_SetExclusiveMonitors()` function sets the Exclusives monitors for a Load-Exclusive instruction, for a block of bytes. The size of the blocks is determined by `size`, at the VA `address`. The `AArch32_ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors and therefore completes successfully.

The `AArch32_ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors, by checking whether the Exclusives monitors are set to include the location of the memory block specified by `size`, at the virtual address defined by `address`. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `AArch32_SetExclusiveMonitors()`.

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `AArch32_ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The `MarkExclusiveGlobal()` procedure takes as arguments a `FullAddress.paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records that the PE `processorid` has requested exclusive access covering at least `size` bytes from address `paddress`. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The `MarkExclusiveLocal()` procedure takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records in a local record that PE `processorid` has requested exclusive access to an address covering at least `size` bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

The `IsExclusiveGlobal()` function takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked in a global record an address range as exclusive access requested that covers at least `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

The `IsExclusiveLocal()` function takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of `size` bytes from address `paddress`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `ClearExclusiveByAddress()` procedure takes as arguments a `FullAddress` `paddress`, the PE identifier `processorid` and the size of the transfer. The procedure clears the global records of all PEs, other than `processorid`, for which an address region including any of `size` bytes starting from `paddress` has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

G4.8.6 Access permission checking

The `AArch32_S1LDHasPermissionsFault()`, `AArch32_S1SDHasPermissionsFault()`, and `AArch32_S2HasPermissionsFault()` functions are used by the architecture to perform access permission checking based on attributes derived from the Translation Table descriptors.

The interpretation of access permission is shown in Memory access control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G4.8.7 Abort exceptions

The function `AArch32_Abort()` generates a Data Abort exception or a Prefetch Abort exception by calling the `AArch32_TakeDataAbortException()` or `AArch32_TakePrefetchAbortException()` function.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault. Pseudocode description of VMSAv8-32 memory system operations provides a number of wrappers to generate a FaultRecord.

The function `NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a FaultRecord contains a fault.

G4.8.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBMaintenanceScope` and `MBReqTypes`.

These enumerations define the required Shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier()`, `DataSynchronizationBarrier()`, and `InstructionSynchronizationBarrier()` perform the memory barriers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G5
The AArch32 Virtual Memory System Architecture

This chapter describes the A-profile AArch32 *Virtual Memory System Architecture* (VMSA). It includes the following sections:

- About VMSA v8-32.
- The effects of disabling address translation stages on VMSAv8-32 behavior.
- Translation tables.
- The VMSAv8-32 Short-descriptor translation table format.
- The VMSAv8-32 Long-descriptor translation table format.
- Memory access control.
- Memory region attributes.
- Translation Lookaside Buffers.
- TLB maintenance requirements.
- Caches in VMSAv8-32.
- VMSAv8-32 memory aborts.
- Exception reporting in a VMSAv8-32 implementation.
- Address translation instructions.
- Pseudocode description of VMSAv8-32 memory system operations.
- About the System registers for VMSAv8-32.
- Functional grouping of VMSAv8-32 System registers.

Note

This chapter must be read with The AArch32 System Level Memory Model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.1 About VMSAv8-32

This chapter describes the A-profile VMSA for AArch32 state, VMSAv8-32.

This chapter describes the control of the VMSA by Exception levels that are using AArch32. Security state, Exception levels, and AArch32 execution privilege summarizes how the AArch32 PE modes map onto the Exception levels.

FEAT_SEL2, if implemented, is not available in AArch32 state and EL2 only executes in Non-secure state.

FEAT_S2FWB, if implemented, is not available in AArch32 state. If EL2 is executing in AArch64 state 2 stage translations might be affected. For more information, see The AArch64 Virtual Memory System Architecture.

The AArch64 Virtual Memory System Architecture describes the control of the VMSA by Exception levels that are using AArch64.

The main function of the VMSA is to perform address translation, and access permissions and memory attribute determination and checking, for memory accesses made by the PE. Address translation, and permissions and attribute determination and checking, is performed by a *stage* of address translation.

In VMSAv8-32, the *Memory Management Unit* (MMU) provides a number of stages of address translation. This chapter describes only the stages that are visible from Exception levels that are using AArch32, which are as follows:

For operation in Secure state

A single stage of address translation, for use when executing at PL1 or EL0. This is the *Secure PL1&0 stage 1* address translation stage.

For operation in Non-secure state

- A single stage of address translation for use when executing at EL2. This is the *Non-secure EL2 stage 1* address translation stage.
- Two stages of address translation for use when executing at PL1 or EL0. These are:
  - The *Non-secure PL1&0 stage 1* address translation stage.
  - The *Non-secure PL1&0 stage 2* address translation stage.

The System registers provide independent control of each supported stage of address translation, including a control to disable that stage of translation.

However, if the PE is executing at EL0 using AArch32 when EL1 is using AArch64 then it is using the VMSAv8-64 EL1&0 translation regime, described in The AArch64 Virtual Memory System Architecture.

These features mean the VMSAv8-32 can support a hierarchy of software supervision, for example an Operating System and a hypervisor.

Each stage of address translation uses address translations and associated memory properties held in memory mapped tables called *translation tables*.

For information about how the MMU features differ if an implementation does not include all of the Exception levels, see About address translation for VMSAv8-32.

The translation tables define the following properties:

Access to the Secure or Non-secure address map

The translation table entries determine whether an access from Secure state accesses the Secure or the Non-secure address map. Any access from Non-secure state accesses the Non-secure address map.

Memory access permission control

This controls whether a program is permitted to access a memory region. For instruction and data access, the possible settings are:

- No access.
- Read-only.
- Write-only. This is possible only in a translation regime with two stages of translation.
- Read/write.

For instruction accesses, additional controls determine whether instructions can be fetched and executed from the memory region.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If a PE attempts an access that is not permitted, a memory fault is signaled to the PE.

Memory region attributes

These describe the properties of a memory region. The top-level attribute, the Memory type, is one of Normal, or a type of Device memory, as follows:

- Both translation table formats support the following Device memory types:
  - Device-nGnRnE
  - Device-nGnRE
- The Long-descriptor translation table format supports, in addition, the following Device memory types:
  - Device-nGRE
  - Device-GRE

Note

Armv8 added the Device-nGRE and Device-GRE memory types. Also, in versions of the Arm architecture before Armv8:

- Device-nGnRnE memory is described as Strongly-ordered memory.
- Device-nGnRE memory is described as Device memory.

Normal memory regions can have additional attributes.

For more information, see Memory types and attributes.

Address translation mappings

An address translation maps an *input address* to an *output address*.

A stage 1 translation takes the address of an explicit data access or instruction fetch, a *virtual address* (VA), as the input address, and translates it to a different output address:

- If only one stage of translation is provided, this output address is the *physical address* (PA).
- If two stages of address translation are provided, the output address of the stage 1 translation is an *intermediate physical address* (IPA).

Note

In the Armv8-32 architecture, a software agent, such as an Operating System, that uses or defines stage 1 memory translations, might be unaware of the distinction between IPA and PA.

A stage 2 translation translates the IPA to a PA.

The possible Security states and privilege levels of memory accesses define a set of *translation regimes*, where a translation regime maps an input VA to the corresponding PA, using one or two stages of translation. See The VMSAv8-32 translation regimes.

System registers control VMSAv8-32, including defining the location of the translation tables, and enabling and configuring the MMU, including enabling and disabling the different address translation stages. Also, they report any faults that occur on a memory access. For more information, see Functional grouping of VMSAv8-32 System registers.

The following sections give an overview of VMSAv8-32, and of the implementation options for VMSAv8-32:

- The VMSAv8-32 translation regimes.
- Address types used in a VMSAv8-32 description.
- Address spaces in VMSAv8-32.
- About address translation for VMSAv8-32.

The remainder of the chapter fully describes the VMSA, including the different implementation options, as summarized in Organization of the remainder of this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.1.1 The VMSAv8-32 translation regimes

As introduced in Address translation mappings, a translation regime maps an input VA to the corresponding PA, using one or two stages of translation. Figure G5-1 shows the VMSAv8-32 translation regimes, and their associated translation stages and the Exception levels from which they are controlled.

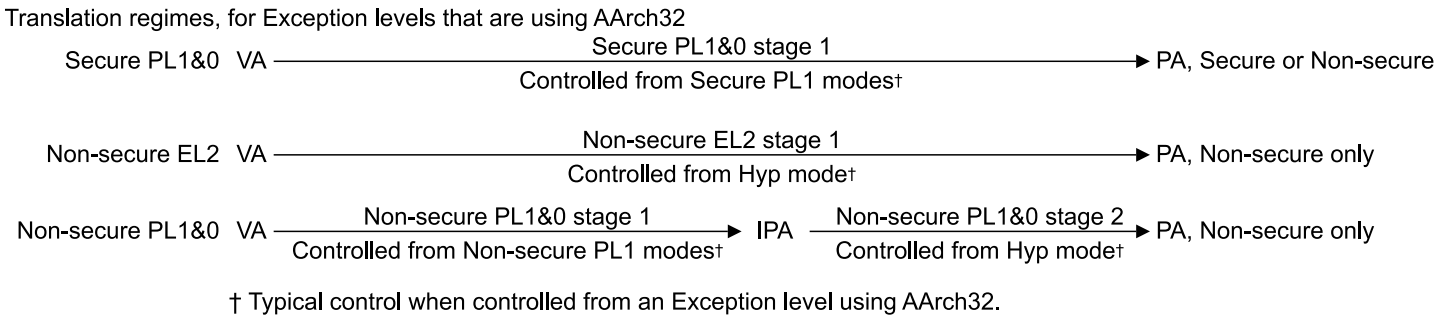

Figure G5-1 VMSAv8-32 translation regimes, and associated control

Note

Conceptually, a translation regime that has only a stage 1 address translation is equivalent to a regime with a fixed, flat stage 2 mapping from IPA to PA.

Limited use of Privilege level in AArch32 state describes the mapping between the PE modes and the Privilege levels (PLs).

G5.1.1.1 Alternative descriptions of the PL1&0 translation regime

The PL1&0 is described in terms of Privilege level because of the way the AArch32 PE modes map onto the Exception levels, as described in Limited use of Privilege level in AArch32 state. The description of this translation regime in terms of the Exception levels using depends on the current state of the PE, as follows:

- In Non-secure state, PL1 always maps to EL1, and therefore the Non-secure PL1&0 translation regime could be described as the Non-secure EL1&0 translation regime.
- In Secure state:
  - When EL3 is using AArch32, PL1 maps to EL3, and therefore under these conditions the Secure PL1&0 translation regime could be described as the Secure EL3&0 translation regime,
  - When EL3 is using AArch64, Secure PL1 maps to Secure EL1, and therefore under these conditions the Secure PL1&0 translation regime could be described as the Secure EL1&0 translation regime,

However, these descriptions all refer to the same translation regime, with the same System registers associated with its stage 1 translations. Therefore, the regime is generally referred to as the PL1&0 translation regime.

Note

As Figure G5-1 shows, stage 2 translation is supported only in Non-secure state.

G5.1.2 Address types used in a VMSAv8-32 description

A description of VMSAv8-32 refers to the following address types.

Note

These descriptions relate to a VMSAv8-32 description and therefore sometimes differ from the generic definitions given in the Glossary.

Virtual address (VA)

An address used in an instruction, as a data or instruction address, is a Virtual Address (VA).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An address held in the PC, LR, or SP, is a VA.

The VA map runs from zero to the size of the VA space. For AArch32 state, the maximum VA space is 4GB, giving a maximum VA range of `0x0000_0000-0xFFFF_FFFF`.

Intermediate physical address (IPA)

In a translation regime that provides two stages of address translation, the IPA is the address after the stage 1 translation, and is the input address for the stage 2 translation.

In a translation regime that provides only one stage of address translation, the IPA is identical to the PA.

A VMSAv8-32 implementation provides only one stage of address translation:

- If the implementation does not include EL2.
- When executing in Secure state.
- When executing in Hyp mode.

Physical address (PA)

The address of a location in the Secure or Non-secure memory map. That is, an output address from the PE to the memory system.

G5.1.3 Address spaces in VMSAv8-32

For execution in AArch32 state, the architecture supports:

- A VA space of up to 32 bits. The actual width is IMPLEMENTATION DEFINED.
- An IPA space of up to 40 bits. The translation tables and associated System registers define the width of the implemented address space.

Note

AArch32 defines two translation table formats. The *Long-descriptor* format gives access to the full 40-bit IPA or PA space at a granularity of 4KB. The *Short-descriptor* format:

- Gives access to a 32-bit PA space at 4KB granularity.
- Gives access to a 40-bit PA space, but only at 16MB granularity, by the use of Supersections.

If an implementation includes EL3, the address maps are defined independently for Secure and Non-secure operation, providing two independent 40-bit address spaces, where:

- A VA accessed from Non-secure state can only be translated to the Non-secure address map.
- A VA accessed from Secure state can be translated to either the Secure or the Non-secure address map.

G5.1.4 About address translation for VMSAv8-32

Address translation is the process of mapping one address type to another, for example, mapping VAs to IPAs, or mapping VAs to PAs. A *translation table* defines the mapping from one address type to another, and a *Translation table base register* (TTBR) indicates the start of a translation table. Each implemented stage of address translation shown in Figure G5-1 requires its own translation tables.

For PL1&0 stage 1 translations, the mapping can be split between two tables, one controlling the lower part of the VA space, and the other controlling the upper part of the VA space. This can be used, for example, so that:

- One table defines the mapping for operating system and I/O addresses, that do not change on a context switch.
- A second table defines the mapping for application-specific addresses, and therefore might require updating on a context switch.

The VMSAv8-32 implementation options determine the supported address translation stages. The following descriptions apply when all implemented Exception levels are using AArch32:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMSAv8-32 without EL2 or EL3

Supports only a single PL1&0 stage 1 address translation. Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by TTBR0 and TTBR1, and controlled by TTBCR.

VMSAv8-32 with EL3 but without EL2

Supports only the Secure PL1&0 stage 1 address translation and the Non-secure PL1&0 stage 1 address translation. In each Security state, this stage of translation can be split between two sets of translation tables, with base addresses defined by the Secure and Non-secure copies of TTBR0 and TTBR1, and controlled by the Secure and Non-secure copies of TTBCR.

VMSAv8-32 with EL2 but without EL3

The implementation supports the following stages of address translation:

Non-secure EL2 stage 1 address translation

The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.

Non-secure PL1&0 stage 1 address translation

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure instance of TTBCR.

Non-secure PL1&0 stage 2 address translation

The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

VMSAv8-32 with EL2 and EL3

The implementation supports all of the stages of address translation, as follows:

Secure PL1&0 stage 1 address translation

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Secure copies of TTBR0 and TTBR1, and controlled by the Secure instance of TTBCR.

Non-secure EL2 stage 1 address translation

The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.

Non-secure PL1&0 stage 1 address translation

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure instance of TTBCR.

Non-secure PL1&0 stage 2 address translation

The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

Figure G5-2 shows the translation regimes and stages in a VMSAv8-32 implementation that includes all of the Exception levels, and indicates the PE mode that, typically, defines each set of translation tables, if that stage of address translation is controlled by a Privilege level that is using AArch32:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

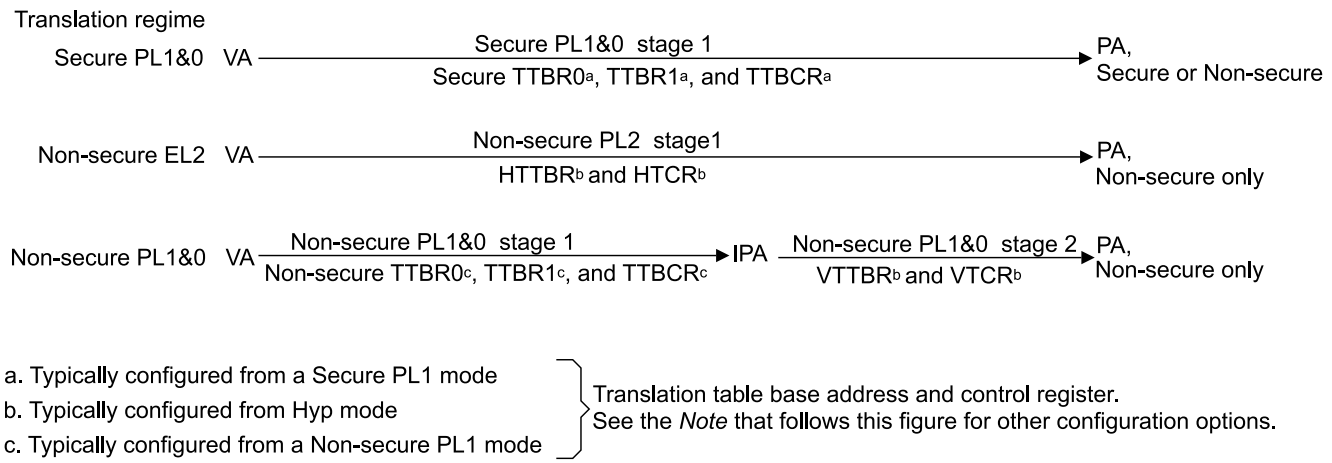

Figure G5-2 VMSAv8-32 address translation summary

Note

The term *Typically configured* is used in Figure G5-2 to indicate the expected software usage. However, stages of address translation used in AArch32 state can also be configured:

- From an Exception level higher than the Exception level of the configuring PE mode shown in Figure G5-2, regardless of whether that Exception level is using AArch32 or is using AArch64, except that a Non-secure Exception level can never configure a stage of address translation that is used in Secure state.
- From an Exception level that is using AArch64 and is higher than the level at which the translation stage is being used. For example, if Non-secure EL0 is the only Non-secure Exception level that is using AArch32, then the Non-secure PL1&0 stage of address translation is configured from Non-secure EL1, that is using AArch64.

In general:

- The translation from VA to PA can require multiple *stages* of address translation, as Figure G5-2 shows.
- A single stage of address translation takes an *input address* and translates it to an *output address*.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single input address to output address translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:

- The required output address.
- The *attributes* and *access permissions* of the addressed memory.

Translation Lookaside Buffers (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and VMSAv8-32 provides TLB maintenance instructions for the management of TLB contents.

Note

The Arm architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault, an Address size fault, or an Access flag fault.

To reduce the software overhead of TLB maintenance, for the PL1&0 translation regimes VMSAv8-32 distinguishes between *Global pages* and *Process-specific pages*. The ASID identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures.

If an implementation includes EL2, the VMID identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

changing from one virtual machine to another, if the virtual machines have different VMIDs. For stage 2 translations, all translations are associated with the current VMID. There is no mechanism to associate a particular stage 2 translation with multiple virtual machines.

G5.1.4.1 Atomicity of register changes on changing virtual machine

From the viewpoint of software executing at Non-secure PL1 or EL0, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for the Non-secure PL1&0 translation regime. This means that all of the following registers must change atomically:

- The registers associated with the stage 1 translations:
  - MAIR0, MAIR1, AMAIR0, and AMAIR1.
  - TTBR0, TTBR1, TTBCR, TTBCR2, and CONTEXTIDR.
  - SCTLR.
- The registers associated with the stage 2 translations:
  - VTTBR and VTCR.
  - HSCTLR.

Note

Only some fields of SCTLR affect the stage 1 translation, and only some fields of HSCTLR affect the stage 2 translation. However, in each case, changing these fields requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution using the Non-secure PL1&0 translation regime. However, when updated as part of a switch of virtual machines they are updated by software executing at EL2. This means the registers are out of context when they are updated, and no synchronization precautions are required.

G5.1.4.2 Use of out-of-context translation regimes

The architecture requires that:

- When executing at EL3 or EL2, the PE must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.
- When executing at EL3 the PE must not use the registers associated with the EL2 translation regime for speculative memory accesses.
- When executing at EL3, EL2, or Non-secure EL1, the PE must not use the registers associated with the Secure PL1&0 translation regime for speculative memory accesses.

If the Statistical Profiling Unit (SPU) is not in use for a lower Exception level when entering an Exception level on completion of a `DSB` instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

If the SPU is in use for a lower Exception level when entering an Exception level on completion of a `PSB CSYNC` and a subsequent `DSB` instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers, to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

Note

- This does not require that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.
- This requirement does not imply that, on taking an exception to a higher Exception level, any translation table walks started before the exception was taken will be completed by the time the higher Exception level is entered, and therefore memory accesses required for such a translation table walk might, in effect, be performed speculatively. However, the execution of a `DSB` on entry to the higher Exception level ensures that these accesses are complete.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.1.5 Organization of the remainder of this chapter

The remainder of this chapter is organized as follows.

The next part of the chapter describes address translation and the associated memory properties held in the translation table entries, in the following sections:

- The effects of disabling address translation stages on VMSAv8-32 behavior.
- Translation tables.
- Secure and Non-secure address spaces.
- The VMSAv8-32 Short-descriptor translation table format.
- The VMSAv8-32 Long-descriptor translation table format.
- Memory access control.
- Memory region attributes.
- Translation Lookaside Buffers.
- TLB maintenance requirements.

Caches in VMSAv8-32 describes VMSAv8-32-specific cache requirements.

The following sections then describe aborts on VMSAv8-32 memory accesses, and how these and other faults are reported:

- VMSAv8-32 memory aborts.
- Exception reporting in a VMSAv8-32 implementation.

Address translation instructions then describes these operations, and how they relate to address translation.

A number of sections then describe the System registers for VMSAv8-32. The following sections give general information about the System registers, and the organization of the registers in the primary encoding spaces, (`coproc == 0b1110`) and (`coproc == 0b1111`) for these registers:

- About the System registers for VMSAv8-32.
- Functional grouping of VMSAv8-32 System registers.

Note

The System registers in the (`coproc == 0b1110`) encoding space provide the following functionality:

- Self-hosted debug. These registers are described in Debug System registers.
- The System register interface to a trace unit. These registers are not described in this manual.
- Jazelle registers. These registers are summarized in Legacy feature registers and system instructions.

Therefore, there is no summary of these registers by functional groups.

Pseudocode description of VMSAv8-32 memory system operations then summarizes the pseudocode functions that describe many features of VMSAv8-32 operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.2 The effects of disabling address translation stages on VMSAv8-32 behavior

About VMSA v8-32 defines the translation regimes and the associated stages of address translation, each of which has its own System registers for control and configuration. VMSAv8-32 includes an enable bit for each stage of address translation, as follows:

- SCTLR.M, in the Secure instance of the register, controls Secure PL1&0 stage 1 address translation.
- SCTLR.M, in the Non-secure instance of the register, controls Non-secure PL1&0 stage 1 address translation.
- HCR.VM controls Non-secure PL1&0 stage 2 address translation.
- HSCTLR.M controls Non-secure EL2 stage 1 address translation.

Note

- The descriptions throughout this chapter describe address translation as seen by Exception levels that are using AArch32. However, for the Non-secure PL1&0 translation regime, the stage 2 translation:
  - Is controlled by the HCR if EL2 is using AArch32.
  - Is controlled by the HCR_EL2 if EL2 is using AArch64.

  For this reason, links to the HCR link to a table that disambiguates between the AArch32 HCR and the AArch64 HCR_EL2.
- If EL2 is using AArch64, then the equivalent of the Non-secure EL2 translation regime is described in The AArch64 Virtual Memory System Architecture, not in this chapter.

The following sections describe the effect on VMSAv8-32 behavior of disabling each stage of translation:

- VMSAv8-32 behavior when stage 1 address translation is disabled.
- VMSAv8-32 behavior when stage 2 address translation is disabled.
- Behavior of instruction fetches when all associated address translations are disabled.

Enabling stages of address translation gives more information about each stage of address translation, in particular after a reset on an implementation that includes EL3.

G5.2.1 VMSAv8-32 behavior when stage 1 address translation is disabled

When stage 1 address translation is disabled, memory accesses that would otherwise be translated by that stage of address translation are treated as follows:

Non-secure PL1 and EL0 accesses when EL2 is implemented and HCR.DC is set to 1

In an implementation that includes EL2, for an access from a Non-secure PL1 or EL0 mode when HCR.DC is set to 1, the stage 1 translation assigns the Normal Non-shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate memory attributes.

When FEAT_XS is implemented and HCR.DC is 1, the XS attribute is set to 0 at stage 1 of the translation. Otherwise, the XS attribute is set to 1 at stage 1 of the translation.

See also Effect of the HCR.DC field.

All other accesses

For all other accesses, when a stage 1 address translation is disabled, the assigned attributes depend on whether the access is a data access or an instruction access, as follows:

Data access

The stage 1 translation assigns the Device-nGnRnE memory type.

Instruction access

The stage 1 translation assigns Normal memory attribute, with the Cacheability and Shareability attributes determined by the value of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Secure instance of SCTLR.I for the Secure PL1&0 translation regime.
- The Non-secure instance of SCTLR.I for the Non-secure PL1&0 translation regime.
- HSCTLR.I for the Non-secure EL2 translation regime.

In these cases, the meaning of the I field is as follows:

When I is set to 0

The stage 1 translation assigns the attributes Outer Shareable, Non-cacheable.

When I is set to 1

The stage 1 translation assigns the attributes Inner Write-Through Read-Allocate No Write-Allocate, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

Note

On some implementations, if the SCTLR.TRE field is set to 0 then this behavior can be changed by the remap settings in the memory remap registers. The details of TEX remap when SCTLR.TRE is set to 0 are IMPLEMENTATION DEFINED, see SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers.

For this stage of translation, no memory access permission checks are performed, and therefore no MMU Permission faults relating to this stage of translation can be generated.

Note

Alignment checking is performed, and therefore Alignment faults can occur.

For every access, when stage 1 translation is disabled, the output address of the stage 1 translation is equal to the input address. This is called a flat address mapping. If the implementation supports output addresses of more than 32 bits then the output address bits above bit[31] are zero. For example, for a VA to PA translation on an implementation that supports 40-bit PAs, PA[39:32] is `0x00`.

For a Non-secure PL1 or EL0 access, if the PL1&0 stage 2 address translation is enabled, the stage 1 memory attribute assignments and output address can be modified by the stage 2 translation.

See also Behavior of instruction fetches when all associated address translations are disabled.

G5.2.1.1 Effect of the HCR.DC field

The HCR.DC field determines the default memory attributes assigned for the first stage of the Non-secure PL1&0 translation regime when that stage of translation is disabled.

When executing in a Non-secure PL1 or EL0 mode with HCR.DC set to 1:

- For all purposes other than reading the value of the SCTLR, the PE behaves as if the value of the SCTLR.M field is 0. This means Non-secure PL1&0 stage 1 address translation is disabled.
- For all purposes other than reading the value of the HCR, the PE behaves as if the value of the HCR.VM field is 1. This means Non-secure PL1&0 stage 2 address translation is enabled.

The effect of HCR.DC might be held in TLB entries associated with a particular VMID. Therefore, if software executing at EL2 changes the HCR.DC value without also changing the current VMID, it must also invalidate all TLB entries associated with the current VMID. Otherwise, the behavior of Non-secure software executing at EL1 or EL0 is CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

G5.2.1.2 Effect of disabling translation on maintenance and address translation instructions

Cache maintenance instructions act on the target cache whether address translation is enabled or not, and regardless of the values of the memory attributes. However, if a stage of translation is disabled, they use the flat address mapping for that stage, and all mappings are considered global.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLB invalidate operations act on the target TLB whether address translation is enabled or not.

When the Non-secure PL1&0 stage 1 address translation is disabled, any ATS1C** or ATS12NSO** address translation instruction that accesses the Non-secure state translation reflects the effect of the HCR.DC field.

G5.2.2 VMSAv8-32 behavior when stage 2 address translation is disabled

When stage 2 address translation is disabled:

- The IPA output from the stage 1 translation maps flat to the PA.
- The memory attributes and permissions from the stage 1 translation apply to the PA.

If the stage 1 address translation and the stage 2 address translation are both disabled, see Behavior of instruction fetches when all associated address translations are disabled.

G5.2.3 Behavior of instruction fetches when all associated address translations are disabled

The information in this section applies to memory accesses:

- From Secure PL1 and EL0 modes, when the Secure PL1&0 stage 1 address translation is disabled.
- From Hyp mode, when the Non-secure EL2 stage 1 address translation is disabled.
- From Non-secure PL1 and EL0 modes, when all of the following apply:
  - The Non-secure PL1&0 stage 1 address translation is disabled.
  - The Non-secure PL1&0 stage 2 address translation is disabled.
  - HCR.DC is set to 0.

In these cases, when execution is in AArch32 state a memory location might be accessed as a result of an instruction fetch if either:

- The memory location is in the same 4KB block of memory, aligned to 4KB, as an instruction which a simple sequential execution of the program either requires to be fetched now or has required to be fetched since the last reset, or is in the 4KB block immediately following such a block.
- The memory location is the target of a direct branch that a simple sequential execution of the program would have taken since the most recent of:
  - The last reset.
  - If the branch predictor is architecturally invisible, the last synchronization of instruction cache maintenance targeting the address of the branch instruction.
  - If the branch predictor is not architecturally invisible, the last synchronization of branch predictor maintenance targeting the address of the branch instruction.

These accesses can be caused by speculative instruction fetches, regardless of whether the prefetched instruction is committed for execution.

Note

To ensure architectural compliance, software must ensure that both of the following apply:

- Instructions that will be executed when address translation is disabled are located in 4KB blocks of the address space that contain only memory that is tolerant to speculative accesses.
- Each 4KB block of the address space that immediately follows a 4KB block that holds instructions that will be executed when address translation is disabled also contains only memory that is tolerant to speculative accesses.

G5.2.4 Enabling stages of address translation

On powerup or Warm reset, only the SCTLR.M field for the Exception level and Security state entered on reset is reset to 0, disabling address translation for the initial state of the PE. All other SCTLR.M and HSCTLR.M fields that are implemented are UNKNOWN after the reset.

This means, on powerup or reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On an implementation that includes EL3, where EL3 is using AArch32:
  - The PL1&0 stage 1 address translation enable bit, SCTLR.M, is banked, meaning there are separate enables for operation in Secure and Non-secure state.
  - If EL3 is using AArch32, only the Secure instance of the SCTLR.M field resets to 0, disabling the Secure state PL1&0 stage 1 address translation. The reset value of the Non-secure instance of SCTLR.M is UNKNOWN.
- On an implementation that includes EL2, where EL2 is using AArch32, the HSCTLR.M field, that controls the Non-secure EL2 stage 1 address translation:
  - If the implementation does not include EL3, resets to 0.
  - Otherwise, is UNKNOWN.
- On an implementation that does not include either EL2 or EL3, there is a single stage of translation. This is controlled by SCTLR.M, that resets to 0.

Note

If, for the software that enables or disables a stage of address translation, the input address of a stage 1 translation differs from the output address of that stage 1 translation, and the software is running in translation regime that is affected by that stage of translation, then the requirement to synchronize changes to the System registers means it is uncertain where in the instruction stream the change of the translation takes place. For this reason, Arm strongly recommends that the input address and the output address are identical in this situation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.3 Translation tables

VMSAv8-32 defines two alternative translation table formats:

Short-descriptor format

It uses 32-bit descriptor entries in the translation tables, and provides:

- Up to two levels of address lookup.
- 32-bit input addresses.
- Output addresses of up to 40 bits.
- Support for PAs of more than 32 bits by use of supersections, with 16MB granularity.
- Support for No access, Client, and Manager domains.

Long-descriptor format

It uses 64-bit descriptor entries in the translation tables, and provides:

- Up to three levels of address lookup.
- Input addresses of up to 40 bits, when used for stage 2 translations.
- Output addresses of up to 40 bits.
- 4KB assignment granularity across the entire PA range.
- No support for domains, all memory regions are treated as in a Client domain.
- Fixed 4KB table size, unless truncated by the size of the input address space.

Note

— Translation with a 40-bit input address range requires two concatenated 4KB top-level tables, aligned to 8KB.

— The VMSAv8-64 Long-descriptor translation table format is generally similar to this format, but supports input and output addresses of up to 48 bits, and has an assignment granularity and table size defined by its *translation granule*. This can be 4KB, 16KB, or 64KB. See Translation table descriptor formats.

In all implementations, of the possible address translations shown in Figure G5-2, for stages of address translation that are using AArch32:

- In a particular Security state, the translation tables for the PL1&0 stage 1 translations can use either translation table format, and the TTBCR.EAE field indicates the current translation table format.
- The translation tables for the Non-secure EL2 stage 1 translations, and for the Non-secure PL1&0 stage 2 translations, must use the Long-descriptor translation table format.

Many aspects of performing a translation table walk depend on the current translation table format. Therefore, the following sections describe the two formats, including how the MMU performs a translation table walk for each format:

- The VMSAv8-32 Short-descriptor translation table format.
- The VMSAv8-32 Long-descriptor translation table format.

The following subsections describe aspects of the translation tables and translation table walks, for memory accesses from AArch32 state, that are independent of the translation table format:

- Translation table walks for memory accesses using VMSAv8-32 translation regimes.
- Information returned by a translation table lookup.
- Determining the translation table base address in the VMSAv8-32 translation regimes.
- Control of translation table walks on a TLB miss.
- Access to the Secure or Non-secure PA map.

See also TLB maintenance requirements.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.3.1 Translation table walks for memory accesses using VMSAv8-32 translation regimes

A translation table walk occurs as the result of a TLB miss, and starts with a read of the appropriate starting-level translation table. The result of that read determines whether additional translation table reads are required, for this stage of translation, as described in either:

- Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format.
- Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format.

Note

When using the Short-descriptor translation table format, the starting level for a translation table walk is always a level 1 lookup. However, with the Long-descriptor translation table format, the starting-level can be either a level 1or a level 2 lookup.

For the PL1&0 stage 1 translations, SCTLR.EE determines the endianness of the translation table lookups. SCTLR is banked, and therefore the endianness is determined independently for each Security state.

HSCTLR.EE defines the endianness for the Non-secure EL2 stage 1 and Non-secure PL1&0 stage 2 translations.

Note

Dynamically changing translation table endianness

Because any change to SCTLR.EE or HSCTLR.EE requires synchronization before it is visible to subsequent operations, Arm strongly recommends that:

- SCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by SCTLR.EE.
  - Executing with SCTLR.M set to 0.
- HSCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by HSCTLR.EE.
  - Executing with HSCTLR.M set to 0.

The PA of the base of the starting-level translation table is determined from the appropriate TTBR, see Determining the translation table base address in the VMSAv8-32 translation regimes.

For more information, see Ordering and completion of TLB maintenance instructions.

Translation table walks must access data or unified caches, or data and unified caches, of other agents participating in the coherency protocol, according to the Shareability attributes described in the TTBR. These Shareability attributes must be consistent with the Shareability attributes for the translation tables themselves.

G5.3.2 Information returned by a translation table lookup

When an associated stage of address translation is enabled, a memory access requires one or more translation table lookups. If the required Translation Table descriptor is not held in a TLB, a translation table walk is performed to obtain the descriptor. A lookup, whether from the TLB or as the result of a translation table walk, returns both:

- An output address that corresponds to the input address for the lookup.
- A set of properties that correspond to that output address.

The returned properties are classified as providing *address map control*, *access controls*, or *region attributes*. This classification determines how the descriptions of the properties are grouped. The classification is based on the following model:

Address map control

Memory accesses from Secure state can access either the Secure or the Non-secure address map, as summarized in Access to the Secure or Non-secure PA map.

Memory accesses from Non-secure state can only access the Non-secure address map.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access controls

Determine whether the PE, in its current state, can access the output address that corresponds to the given input address. If not, an MMU fault is generated and there is no memory access.

Memory access control describes the properties in this group.

Attributes Are valid only for an output address that the PE, in its current state, can access. The attributes define aspects of the required behavior of accesses to the target memory region.

Memory region attributes describes the properties in this group.

G5.3.3 Determining the translation table base address in the VMSAv8-32 translation regimes

On a TLB miss, the VMSA must perform a translation table walk, and therefore must find the base address of the translation table to use for its lookup. A TTBR holds this address. As Figure G5-2 shows:

- For a Non-secure EL2 stage 1 translation, the HTTBR holds the required base address. The HTCR is the control register for these translations.
- For a Non-secure PL1&0 stage 2 translation, the VTTBR holds the required base address. The VTCR is the control register for these translations.
- For a PL1&0 stage 1 translation, either TTBR0 or TTBR1 holds the required base address. The TTBCR is the control register for these translations.
  The Non-secure copies of TTBR0, TTBR1, and TTBCR, relate to the Non-secure PL1&0 stage 1 translation. The Secure copies of TTBR0, TTBR1, and TTBCR, relate to the Secure PL1&0 stage 1 translation.

For the PL1&0 translation table walks:

- TTBR0 can be configured to describe the translation of VAs in the entire address map, or to describe only the translation of VAs in the lower part of the address map.
- If TTBR0 is configured to describe the translation of VAs in the lower part of the address map, TTBR1 is configured to describe the translation of VAs in the upper part of the address map.

The contents of the appropriate instance of the TTBCR determine whether the address map is separated into two parts, and where the separation occurs. The details of the separation depend on the current translation table format, see:

- Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format.
- Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format.

Example G5-1 shows a typical use of the two sets of translation tables:

Example G5-1 Example use of TTBR0 and TTBR1

An example of using the two TTBRs for PL1&0 stage 1 address translations is:

TTBR0 Used for process-specific addresses.

Each process maintains a separate level 1 translation table. On a context switch:

- TTBR0 is updated to point to the level 1 translation table for the new context.
- TTBCR is updated if this change changes the size of the translation table.
- The CONTEXTIDR is updated.

TTBCR can be programmed so that all translations use TTBR0 in a manner compatible with architecture versions before Armv6.

TTBR1 Used for operating system and I/O addresses, that do not change on a context switch.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.3.4 Control of translation table walks on a TLB miss

Two fields in the TCR for the translation stage required by a memory access control whether a translation table walk is performed on a TLB miss. These two fields are the:

- PD0 and PD1 fields, on a PE using the Short-descriptor translation table format.
- EPD0 and EPD1 fields, on a PE using the Long-descriptor translation table format.

Note

For the VMSAv8-32 translation regimes, the different field names are because the fields are in different positions in TTBCR, depending on the translation table format.

The effect of these fields is:

{E}PDx == 0 If a TLB miss occurs based on TTBRx, a translation table walk is performed. The current Security state determines whether the memory access is Secure or Non-secure.

{E}PDx == 1 If a TLB miss occurs based on TTBRx, a level 1 Translation fault is returned, and no translation table walk is performed.

G5.3.5 Access to the Secure or Non-secure PA map

As stated in Address spaces in VMSAv8-32, a PE can access independent Secure and Non-secure address maps. When the PL1 Exception level is using AArch32, these are defined by the translation tables identified by the Secure TTBR0 and TTBR1. In both translation table formats in the Secure translation tables, the NS field in a descriptor indicates whether the descriptor refers to the Secure or the Non-secure address map:

NS == 0 Access the Secure PA space.

NS == 1 Access the Non-secure PA space.

Note

In the Non-secure translation tables, the corresponding field is SBZ. Non-secure accesses always access the Non-secure PA space, regardless of the value of this field.

The Long-descriptor translation table format extends this control, adding an NSTable field to the Secure translation tables, as described in Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format. In the Non-secure translation tables, the corresponding field is SBZ, and Non-secure accesses ignore the value of this field.

The following sections describe the address map controls in the two implementations:

- Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format.
- Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format.

The following subsection gives more information.

G5.3.5.1 Secure and Non-secure address spaces

EL3 provides two PA spaces, a Secure PA space and a Non-secure PA space.

As described in Access to the Secure or Non-secure PA map, for the PL1&0 stage 1 translations when controlled from an Exception level using AArch32, the registers that control the stage of translation, TTBR0, TTBR1, TTBCR, and TTBCR2 are banked to provide independent Secure and Non-secure instances of the registers, and the Security state of the PE when it performs a memory access whether the Secure or Non-secure instances are used. This means that for stage 1 of the PL1&0 translation regime there are independent Secure and Non-secure translation tables, and translation table walks are made to the PA space corresponding to the Security state of the translation tables used.

For a translation table walk caused by a memory access from Non-secure state, all memory accesses are to the Non-secure address space.

For a translation table walk caused by a memory access from Secure state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When address translation is using the Long-descriptor translation table format:
  - The initial lookup performed must access the Secure address space.
  - If a Table descriptor read from the Secure address space has the NSTable field set to 0, then the next level of lookup is from the Secure address space.
  - If a Table descriptor read from the Secure address space has the NSTable field set to 1, then the next level of lookup, and any subsequent level of lookup, is from the Non-secure address space.

  For more information, see Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format.
- Otherwise, all memory accesses are to the Secure address space.

Note

- When executing in Non-secure state, additional translations are supported. For memory accesses from AArch32 state, these are:
  - Non-secure EL2 stage 1 translation.
  - Non-secure PL1&0 stage 2 translation.

  These translations can access only the Non-secure address space.
- A system implementation can alias parts of the Secure PA space to the Non-secure PA space in an IMPLEMENTATION SPECIFIC way. As with any other aliasing of physical memory, the use of aliases in this way can require the use of cache maintenance instructions to ensure that changes to memory made using one alias of the physical memory are visible to accesses to the other alias of the physical memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.4 The VMSAv8-32 Short-descriptor translation table format

The Short-descriptor translation table format supports a memory map based on memory sections or pages:

Supersections Consist of 16MB blocks of memory. Support for Supersections is optional, except that an implementation that supports more than 32 bits of PA must also support Supersections to provide access to the entire PA space.

Sections Consist of 1MB blocks of memory.

Large pages Consist of 64KB blocks of memory.

Small pages Consist of 4KB blocks of memory.

Supersections, Sections, and Large pages map large regions of memory using only a single TLB entry.

Note

- Whether a VMSAv8-32 implementation of the Short-descriptor format translation tables supports supersections is IMPLEMENTATION DEFINED.
- The EL2 translation regime cannot use the Short-descriptor translation table format.

When using the Short-descriptor translation table format, two levels of translation tables are held in memory:

Level 1 table

Holds *level 1 descriptors* that contain the base address and

- Translation properties for a Section and Supersection.
- Translation properties and pointers to a level 2 table for a Large page or a Small page.

Level 2 tables

Hold *level 2 descriptors* that contain the base address and translation properties for a Small page or a Large page. With the Short-descriptor format, level 2 tables can be referred to as *translation tables*.

A level 2 table requires 1KB of memory.

In the translation tables, in general, a descriptor is one of:

- An invalid or fault entry.
- A translation table entry, that points to a next-level translation table.
- A page or section entry, that defines the memory properties for the access.
- A reserved format.

Bits[1:0] of the descriptor give the primary indication of the descriptor type.

Figure G5-3 gives a general view of address translation when using the Short-descriptor translation table format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

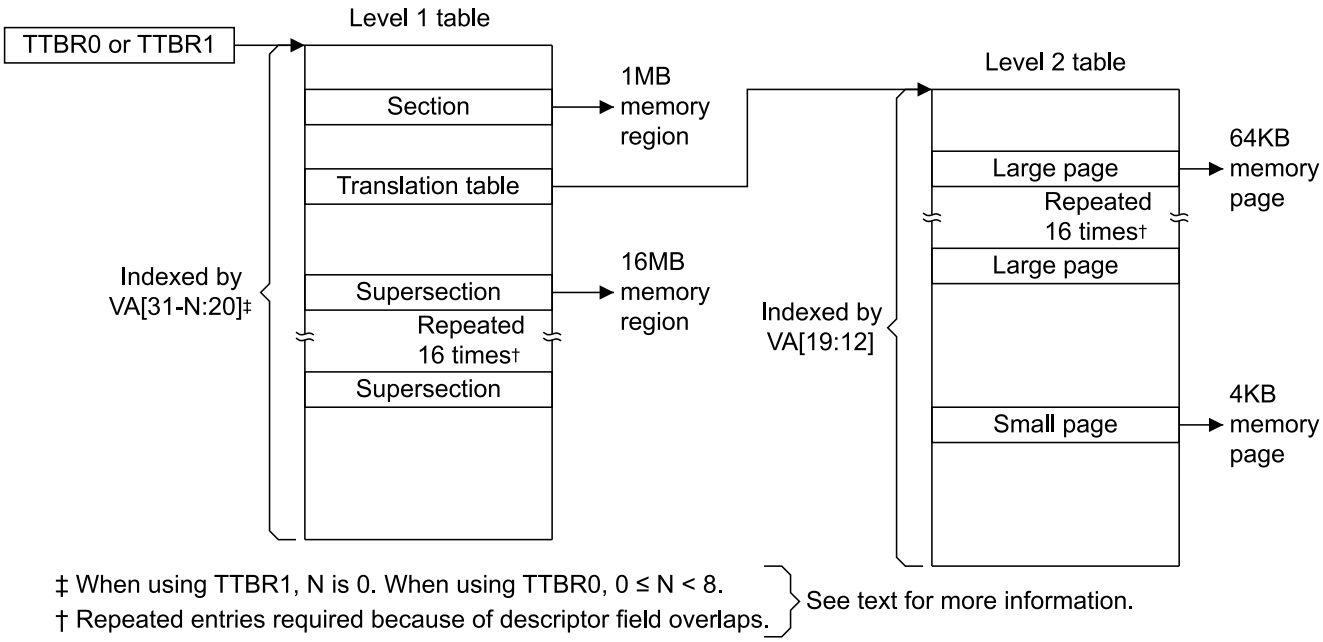

Figure G5-3 General view of address translation using VMSAv8-32 Short-descriptor format translation tables

Additional requirements for Short-descriptor format translation tables describes why, when using the Short-descriptor format, Supersection and Large page entries must be repeated 16 times, as shown in Figure G5-3.

VMSAv8-32 Short-descriptor Translation Table format descriptors, Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors, and Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format describe the format of the descriptors in the Short-descriptor format translation tables.

The following sections then describe the use of this translation table format:

- Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format.
- Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format.

G5.4.1 VMSAv8-32 Short-descriptor Translation Table format descriptors

The following sections describe the formats of the entries in the Short-descriptor Translation Tables:

- Short-descriptor Translation Table level 1 descriptor formats.
- Short-descriptor Translation Table level 2 descriptor formats.

For more information about level 2 translation tables, see Additional requirements for Short-descriptor format translation tables.

Note

Previous versions of the *Arm Architecture Reference Manual*, and some other documentation, describes the AP[2] bit in the translation table entries as the APX bit.

Information returned by a translation table lookup describes the classification of the non-address fields in the descriptors as address map control, access control, or attribute fields.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.4.1.1 Short-descriptor Translation Table level 1 descriptor formats

Each entry in the level 1 table describes the mapping of the associated 1MB VA range.

Figure G5-4 shows the possible level 1 descriptor formats.

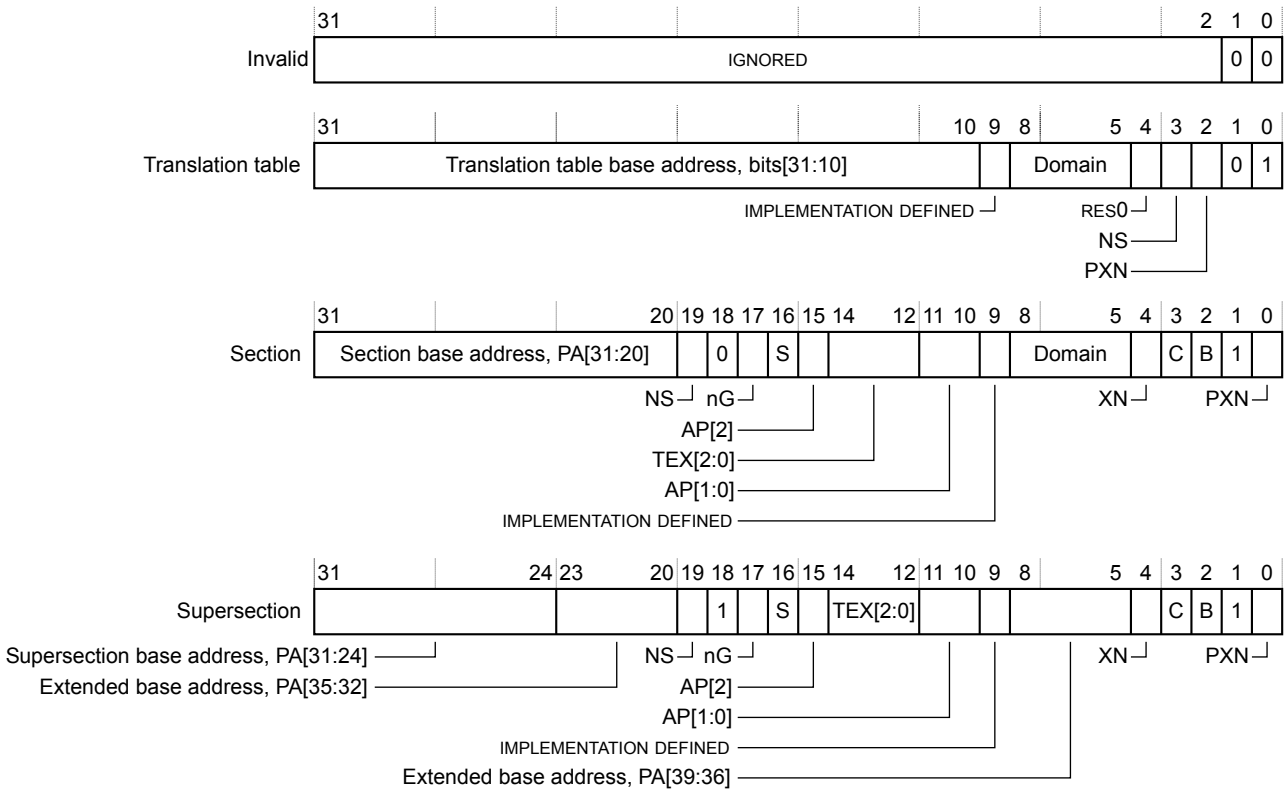

Figure G5-4 VMSAv8-32 Short-descriptor level 1 descriptor formats

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

`0b00`**, Invalid entry**

The associated VA is unmapped, and any attempt to access it generates a Translation fault.

Bits[31:2] of the descriptor are IGNORED, see IGNORED. This means software can use these bits for its own purposes.

`0b01`**, Translation table**

The descriptor gives the address of a level 2 translation table, that specifies the mapping of the associated 1MByte VA range.

`0b10`**, Section or Supersection**

The descriptor gives the base address of the Section or Supersection. Bit[18] determines whether the entry describes a Section or a Supersection.

This encoding also defines the PXN field as 0.

`0b11`**, Section or Supersection, if the implementation supports the PXN attribute**

This encoding is identical to `0b10`, except that it defines the PXN field as 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A VMSAv8-32 implementation can use the Short-descriptor translation table format for the PL1&0 stage 1 translations, by setting TTBCR.EAE to 0.

The address information in the level 1 descriptors is:

Translation table Bits[31:10] of the descriptor are bits[31:10] of the address of a translation table.

Section Bits[31:20] of the descriptor are bits[31:20] of the address of the Section.

Supersection Bits[31:24] of the descriptor are bits[31:24] of the address of the Supersection.

Optionally, bits[8:5, 23:20] of the descriptor are bits[39:32] of the extended Supersection address.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the translation table, Section, or Supersection. Otherwise, the address is the PA of the translation table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors.

G5.4.1.2 Short-descriptor Translation Table level 2 descriptor formats

Figure G5-5 shows the possible formats of a level 2 descriptor.

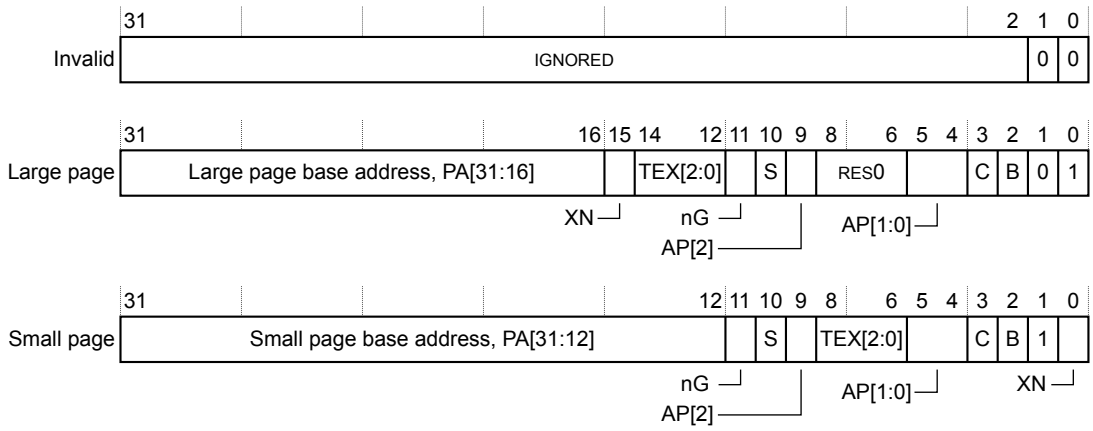

Figure G5-5 Short-descriptor level 2 descriptor formats

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

`0b00`**, Invalid entry**

The associated VA is unmapped, and attempting to access it generates a Translation fault.

Bits[31:2] of the descriptor are IGNORED, see IGNORED. This means software can use these bits for its own purposes.

`0b01`**, Large page**

The descriptor gives the base address and properties of the Large page.

`0b1x`**, Small page**

The descriptor gives the base address and properties of the Small page.

In this descriptor format, bit[0] of the descriptor is the XN field.

The address information in the level 2 descriptors is:

Large page Bits[31:16] of the descriptor are bits[31:16] of the address of the Large page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Small page Bits[31:12] of the descriptor are bits[31:12] of the address of the Small page.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the translation table, Section, or Supersection. Otherwise, the address is the PA of the translation table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors.

G5.4.1.3 Additional requirements for Short-descriptor format translation tables

When using Supersection or Large Page descriptors in the Short-descriptor translation table format, the input address field that defines the Supersection or Large Page descriptor address overlaps the table address field. In each case, the size of the overlap is 4 bits. The following diagrams show these overlaps:

- Figure K8-14 for the level 1 translation table entry for a Supersection.
- Figure K8-16 for the level 2 translation table entry for a Large page.

Considering the case of using Large Page descriptors in a level 2 translation table, this overlap means that for any specific Large page, the bottom four bits of the level 2 translation table entry might take any value from `0b0000` to `0b1111`. Therefore, each of these 16 index values must point to a separate copy of the same descriptor.

This means that each Large page or Supersection descriptor must:

- Occur first on a sixteen-word boundary.
- Be repeated in 16 consecutive memory locations.

G5.4.2 Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors

This section describes the descriptor fields other than the descriptor type field and the address field:

TEX[2:0], C, B

Memory region attribute fields, see Memory region attributes.

These fields are not present in a descriptor for a translation table.

XN bit The Execute-never field, see Access permissions for instruction execution.

This bit is not present in a descriptor for a translation table.

PXN bit The Privileged execute-never field, see Access permissions for instruction execution.

When this field is set to 1 in the descriptor for a translation table, it indicates that all memory pages described in the corresponding translation table are Privileged execute-never.

NS bit Non-secure bit. Specifies whether the translated PA is in the Secure or Non-secure address map, see Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format.

This bit is not present in level 2 descriptors. The value of the NS bit in a level 1 descriptor for a translation table applies to all entries in the corresponding level 2 translation table.

Domain Domain field, see Domains, Short-descriptor format only.

This field is not present in a Supersection entry. Memory described by Supersections is in domain 0.

This bit is not present in level 2 descriptors. The value of the Domain field in the level 1 descriptor for a translation table applies to all entries in the corresponding level 2 translation table.

An IMPLEMENTATION DEFINED bit

This bit is not present in level 2 descriptors.

AP[2], AP[1:0]

Access Permissions bits, see Memory access control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AP[0] can be configured as the *Access flag*, see The Access flag.

These bits are not present in a descriptor for a translation table.

S bit Shareable bit. Used in determining the Shareability of the addressed region, see Memory region attributes.

Note

The naming of this bit as the *Shareable* bit is carried forward from early versions of the Arm architecture. This name is no longer an adequate description of the interpretation of the bit.

This bit is not present in a descriptor for a translation table.

nG bit The not global bit. If a lookup using this descriptor is cached in a TLB, determines whether the TLB entry applies to all ASID values, or only to the current ASID value. See Global and process-specific translation table entries.

This bit is not present in a descriptor for a translation table.

Bit[18], when bits[1:0] indicate a *Section or Supersection* descriptor

0 Descriptor is for a Section.

1 Descriptor is for a Supersection.

G5.4.3 Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format

Access to the Secure or Non-secure PA map describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Short-descriptor translation table format, the NS bit is defined only in the level 1 translation tables. This means that, in a level 1 descriptor for a translation table, the NS bit defines the PA map, Secure or Non-secure, for all of the Large pages and Small pages of memory described by that table.

The NS bit of a level 1 descriptor for a translation table has no effect on the PA map in which that translation table is held. As stated in Secure and Non-secure address spaces, the PA of that translation table is in:

- The Secure address map if the translation table walk is in Secure state.
- The Non-secure address map if the translation table walk is in Non-secure state.

This means the granularity of the Secure and Non-secure memory maps is 1MB. However, in these memory maps, table entries can define physical memory regions with a granularity of 4KB.

G5.4.4 Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format

As described in Determining the translation table base address in the VMSAv8-32 translation regimes, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N == 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- If N > 0 then:
  - If bits[31:32-N] of the input VA are all zero, then use TTBR0.
  - Otherwise use TTBR1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G5-1 shows how the value of N determines the lowest address translated using TTBR1, and the size of the level 1 translation table addressed by TTBR0.

Table G5-1 Effect of TTBCR.N on address translation, Short-descriptor format

TTBCR.N	First address translated with TTBR1	TTBR0 table	
		Size	Index range
0b000	TTBR1 not used	16KB	VA[31:20]
0b001	0x8000_0000	8KB	VA[30:20]
0b010	0x4000_0000	4KB	VA[29:20]
0b011	0x2000_0000	2KB	VA[28:20]
0b100	0x1000_0000	1KB	VA[27:20]
0b101	0x0800_0000	512 bytes	VA[26:20]
0b110	0x0400_0000	256 bytes	VA[25:20]
0b111	0x0200_0000	128 bytes	VA[24:20]

Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

Figure G5-6 shows how the value of TTBCR.N controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

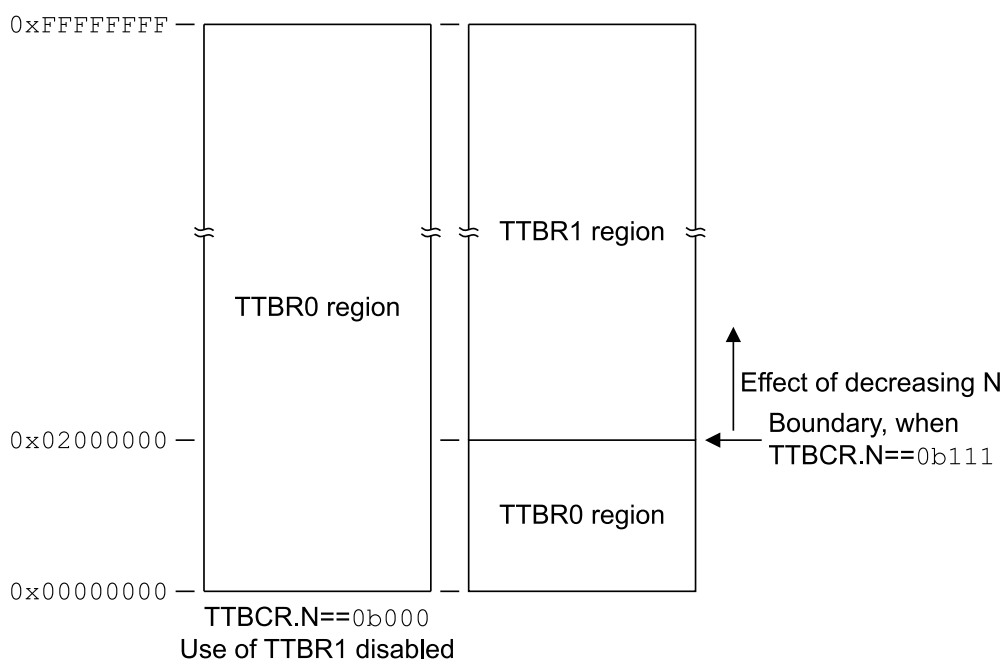

Figure G5-6 How TTBCR.N controls the boundary between the TTBRs, Short-descriptor format

In the selected TTBR, bits RGN, S, and IRGN[1:0] define the memory region attributes for the translation table walk.

Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format describes the translation.

G5.4.5 Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format

When using the Short-descriptor translation table format, and a memory access requires a translation table walk:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A section-mapped access only requires a read of the level 1 translation table.
- A page-mapped access also requires a read of the level 2 translation table.

Reading a level 1 translation table describes how either TTBR1 or TTBR0 is used, with the accessed VA, to determine the address of the level 1 descriptor.

Reading a level 1 translation table shows the output address as A[39:0]:

- For a Non-secure PL1&0 stage 1 translation, this is the IPA of the required descriptor. A Non-secure PL1&0 stage 2 translation of this address is performed to obtain the PA of the descriptor.
- Otherwise, this address is the PA of the required descriptor.

The full translation flow for Sections, Supersections, Small pages and Large pages then shows the complete translation flow for each valid memory access.

G5.4.5.1 Reading a level 1 translation table

When performing a fetch based on TTBR0:

- The address bits taken from TTBR0 vary between bits[31:14] and bits[31:7].
- The address bits taken from the VA, that is the input address for the translation, vary between bits[31:20] and bits[24:20].

The width of the TTBR0 and VA fields depend on the value of TTBCR.N, as Figure G5-7 shows.

When performing a fetch based on TTBR1, Bits TTBR1[31:14] are concatenated with bits[31:20] of the VA. This makes the fetch equivalent to that shown in Figure G5-7, with N==0.

Note

See The address and Properties fields shown in the translation flows for more information about the *Properties* label used in this and other figures.

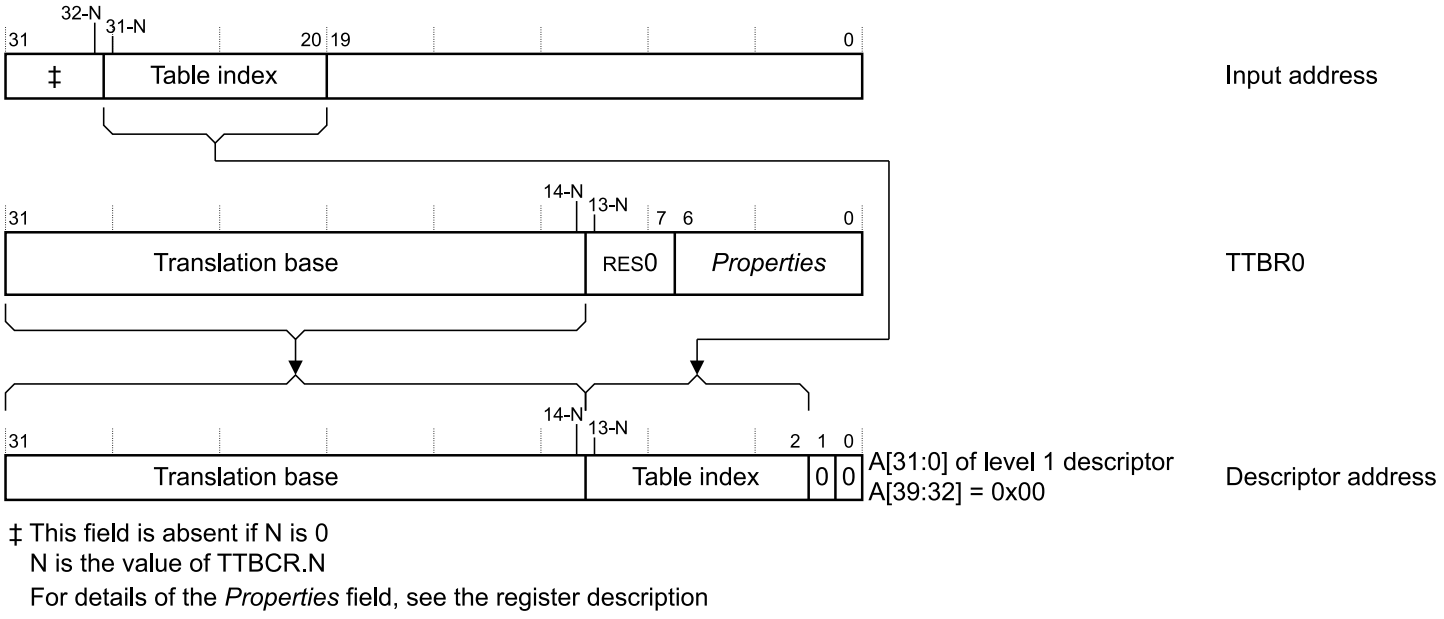

Figure G5-7 Accessing level 1 translation table based on TTBR0, Short-descriptor format

Regardless of which register is used as the base for the fetch, the resulting output address selects a four-byte translation table entry that is one of:

- A level 1 descriptor for a Section or Supersection.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A descriptor for a *translation table*, that points to a level 2 translation table. In this case:
  - A second fetch is performed to retrieve a level 2 descriptor.
  - The descriptor also contains some attributes for the access, see Figure G5-4.
- A faulting entry.

G5.4.5.2 The full translation flow for Sections, Supersections, Small pages and Large pages

In a translation table walk, only the initial lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The Table descriptor read in the previous lookup.
- The input address.

Address translation examples using the VMSAv8-32 Short descriptor translation table format shows the full translation flow for each of the memory section and page options. As described in VMSAv8-32 Short-descriptor Translation Table format descriptors, these options are:

Supersection A 16 MB memory region, see Translation flow for a Supersection.

Section A 1 MB memory region, see Translation flow for a Section.

Large page A 64 KB memory region, described by the combination of:

- A level 1 translation table entry that indicates the address of a level 2 translation table.
- A level 2 descriptor that indicates a Large page.

See Translation flow for a Large page.

Small page A 4 KB memory region, described by the combination of:

- A level 1 translation table entry that indicates the address of a level 2 translation table.
- A level 2 descriptor that indicates a Small page.

See Translation flow for a Small page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.5 The VMSAv8-32 Long-descriptor translation table format

The VMSAv8-32 Long-descriptor translation table format supports the assignment of memory attributes to memory *Pages,* at a granularity of 4KB, across the complete input address range. It also supports the assignment of memory attributes to *blocks* of memory, where a block can be 2MB or 1GB.

Note

- Although the VMSAv8-32 Long-descriptor format is limited to three levels of address lookup, its design and naming conventions support extension to additional levels, to support a larger input address range.
- Similarly, while the VMSAv8-32 implementation limits the output address range to 40 bits, its design supports extension to a larger output address range.

Figure G5-2 shows the different address translation stages. The Long-descriptor translation table format:

- Is used for:
  - The Non-secure EL2 stage 1 translation.
  - The Non-secure PL1&0 stage 2 translation.
- Can be used for the Secure and Non-secure PL1&0 translations.

When used for a stage 1 translation, the translation tables support an input address of up to 32 bits, corresponding to the VA address range of the PE.

When used for a stage 2 translation, the translation tables support an input address range of up to 40 bits, to support the translation from IPA to PA. If the input address for the stage 2 translation is a 32-bit address, then this address is zero-extended to 40 bits.

Note

When the Short-descriptor translation table format is used for the Non-secure stage 1 translations, this generates 32-bit IPAs. These are zero-extended to 40 bits to provide the input address for the stage 2 translation.

Overview of VMSAv8-32 address translation using Long-descriptor translation tables summarizes address translation from AArch32 state when using the Long-descriptor format translation tables.

The following sections then describe the format of the descriptors in the Long-descriptor format translation tables:

- VMSAv8-32 Long-descriptor Translation Table format descriptors.
- Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors.
- Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format.

The following sections then describe this translation table format:

- Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format.
- VMSAv8-32 Long-descriptor translation table format address lookup levels.
- Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format.
- The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format.

G5.5.1 Overview of VMSAv8-32 address translation using Long-descriptor translation tables

Figure G5-8 gives a general view of VMSAv8-32 stage 1 address translation when using the Long-descriptor translation table format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

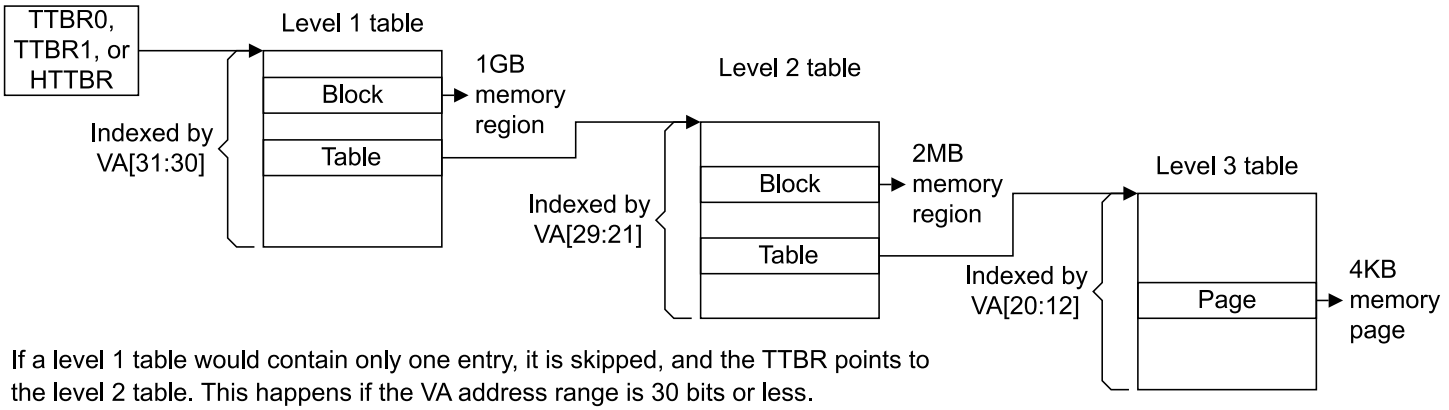

Figure G5-8 General view of VMSAv8-32 stage 1 address translation using Long-descriptor format

Figure G5-9 gives a general view of VMSAv8-32 stage 2 address translation. Stage 2 translation always uses the Long-descriptor translation table format.

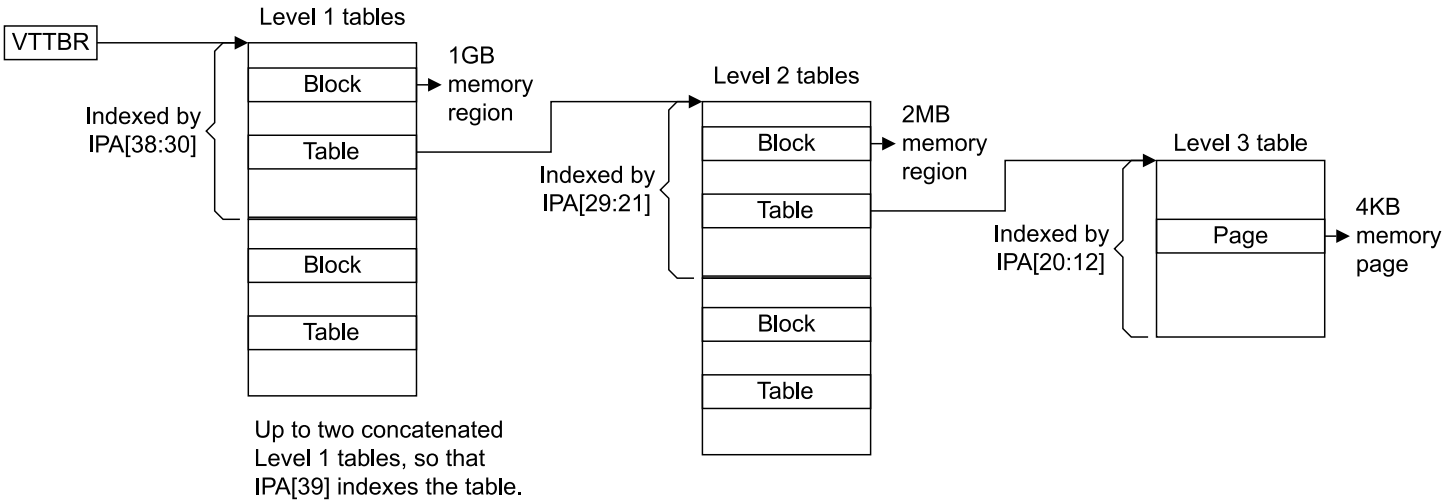

Figure G5-9 General view of VMSAv8-32 stage 2 address translation, Long-descriptor translation table format

Use of concatenated translation tables for the initial stage 2 lookup describes how using concatenated level 2 tables means lookup can start at level 2, as referred to in Figure G5-9.

G5.5.2 VMSAv8-32 Long-descriptor Translation Table format descriptors

As described in VMSAv8-32 Long-descriptor translation table format address lookup levels, the Long-descriptor translation table format provides up to three levels of address lookup. A translation table walk starts either at level 1 or level 2 of the address lookup.

In general, a descriptor is one of:

- An invalid or fault entry.
- A table entry, that points to the next-level translation table.
- A block entry, that defines the memory properties for the access.
- A reserved format.

Bit[1] of the descriptor indicates the descriptor type, and bit[0] indicates whether the descriptor is valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following sections describe the Long-descriptor Translation Table descriptor formats:

- VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats.
- VMSAv8-32 Long-descriptor translation table level 3 descriptor formats.

Information returned by a translation table lookup describes the classification of the non-address fields in the descriptors between *address map control*, *access controls*, and *region attributes*.

G5.5.2.1 VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats

In the Long-descriptor translation tables, the formats of the level 1 and level 2 descriptors differ only in the size of the block of memory addressed by the Block descriptor. A block entry:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

Figure G5-10 shows the Long-descriptor level 1 and level 2 descriptor formats:

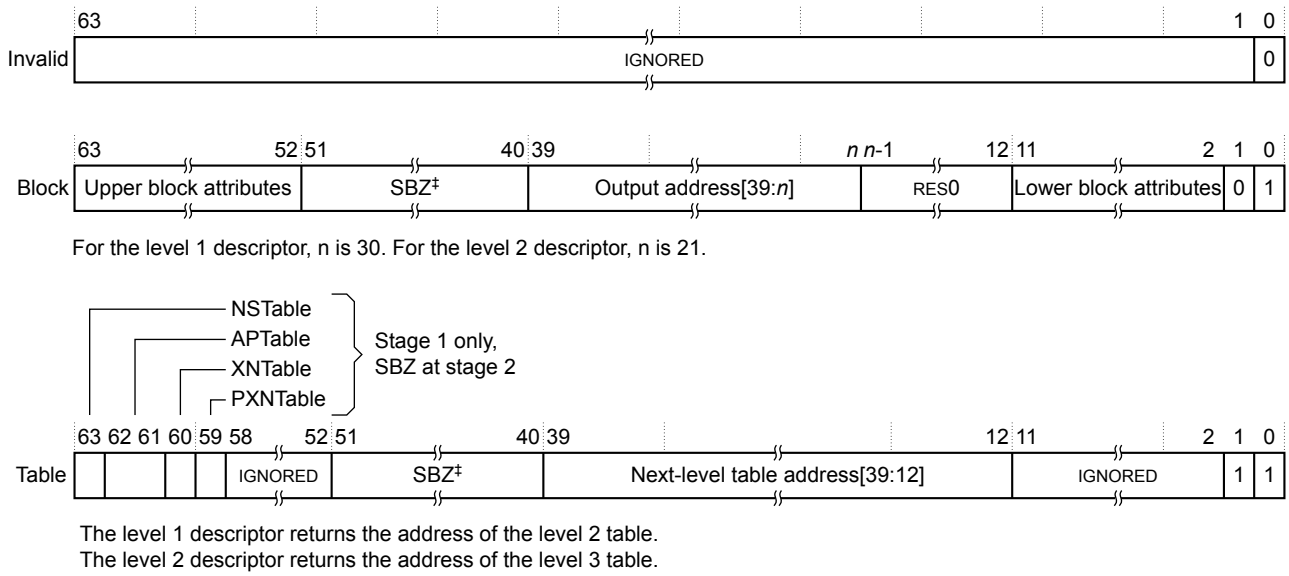

‡ See the descriptions of the address fields for more information about bits[47:40] of the Block and Table descriptors.

Figure G5-10 VMSAv8-32 Long-descriptor level 1and level 2 descriptor formats

G5.5.2.1.1 Descriptor encodings, Long-descriptor level 1 and level 2 formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Block The descriptor gives the base address of a block of memory, and the attributes for that memory region.

1, Table The descriptor gives the address of the next level of translation table, and for a stage 1 translation, some attributes for that translation.

The other fields in the valid descriptors are:

Block descriptor

Gives the base address and attributes of a block of memory:

- For a level 1 Block descriptor, bits[39:30] are bits[39:30] of the output address that specifies a 1GB block of memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a level 2 Block descriptor, bits[39:21] are bits[39:21] of the output address that specifies a 2MB block of memory.

In both cases, if bits[47:40] of the descriptor are not zero then a translation that uses the descriptor will generate an Address size fault, see Address size fault.

Bits[63:52, 11:2] provide attributes for the target memory block, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors. The position and contents of these bits is identical in the level 2 Block descriptor and in the level 3 Page descriptor.

Table descriptor

Bits[39:m] are bits[39:m] of the address of the required next-level table. Bits[m-1:0] of the table address are zero:

- For a level 1 Table descriptor, this is the address of a level 2 table.
- For a level 2 Table descriptor, this is the address of a level 3 table.

In both cases, if bits[47:40] of the descriptor are not zero then a translation that uses the descriptor will generate an Address size fault, see Address size fault.

For a stage 1 translation only, bits[63:59] provide attributes for the next-level lookup, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors.

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target block or table. Otherwise, it is the PA of the target block or table.

G5.5.2.2 VMSAv8-32 Long-descriptor translation table level 3 descriptor formats

Each entry in a level 3 table describes the mapping of the associated 4KB input address range.

Figure G5-11 shows the Long-descriptor level 3 descriptor formats.

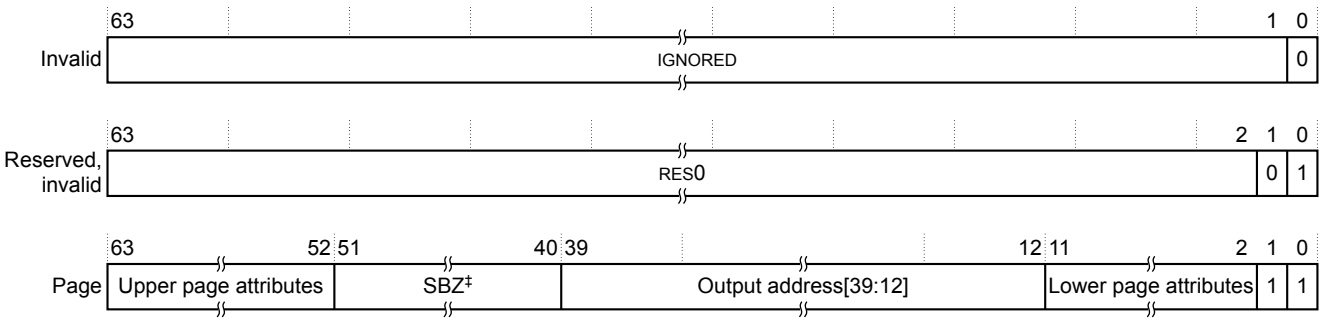

‡ See the description of the address field for more information about bits[47:40] of the Page descriptor.

Figure G5-11 VMSAv8-32 Long-descriptor level 3 descriptor formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Reserved, invalid

Behaves identically to encodings with bit[0] set to 0.

This encoding must not be used in level 3 translation tables.

1, Page Gives the address and attributes of a 4KB page of memory.

At this level, the only valid format is the Page descriptor. The other fields in the Page descriptor are:

Page descriptor

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits[39:12] are bits[39:12] of the output address for a page of memory.

If bits[47:40] of the descriptor are not zero, then a translation that uses the descriptor will generate an Address size fault, see Address size fault.

Bits[63:52, 11:2] provide attributes for the target memory page, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors. The position and contents of these bits are identical in the level 1 Block descriptor and in the level 2 Block descriptor.

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target page. Otherwise, it is the PA of the target page.

G5.5.3 Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors

The memory attributes in the VMSAv8-32 Long-descriptor translation tables are based on those in the Short-descriptor translation table format, with some extensions. Memory region attributes describes these attributes. In the Long-descriptor translation table format:

- Table entries for stage 1 translations define attributes for the next level of lookup, see Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors.

  The hierarchical permissions in the translation tables, APTable, XNTable, and PXNTable, permit subtrees of the translation tables to be used by different agents. Not all operating systems use this functionality, and so FEAT_AA32HPD adds a facility to disable these bits.

  This ability to disable hierarchical permissions bits has no effect on the NSTable bit.
- Block and Page entries define memory attributes for the target block or page of memory. Stage 1 and stage 2 translations have some differences in these attributes, see:
  - Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors.
  - Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors.

G5.5.3.1 Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the following attributes for the next-level translation table access:

NSTable, bit[63] For memory accesses from Secure state, specifies the Security state for subsequent levels of lookup, see Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format.

For memory accesses from Non-secure state, this bit is ignored.

APTable, bits[62:61] Access permissions limit for subsequent levels of lookup, see Hierarchical control of access permissions, Long-descriptor format.

APTable[0] is reserved, SBZ, in the Non-secure EL2 stage 1 translation tables.

From Armv8.2, when FEAT_AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1, and the value of TTBCR.T2E is also 1:

- The value of the corresponding APTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding APTable field is 0, that is to say, the APTable field has an Effective value of 0.

XNTable, bit[60] XN limit for subsequent levels of lookup, see Hierarchical control of instruction fetching, Long-descriptor format.

From Armv8.2, when FEAT_AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1, and the value of TTBCR.T2E is also 1:

- The value of the corresponding XNTable field is IGNORED by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding XNTable field is 0, that is to say, the XNTable field has an Effective value of 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PXNTable, bit[59] PXN limit for subsequent levels of lookup, see Hierarchical control of instruction fetching, Long-descriptor format.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

From Armv8.2, when FEAT_AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1 and the value of TTBCR.T2E is also 1:

- The value of the corresponding PXNTable field is ignored by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding PXNTable field is 0, that is to say, the PXNTable field has an Effective value of 0.

G5.5.3.2 Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block as shown for a stage 1 translation:

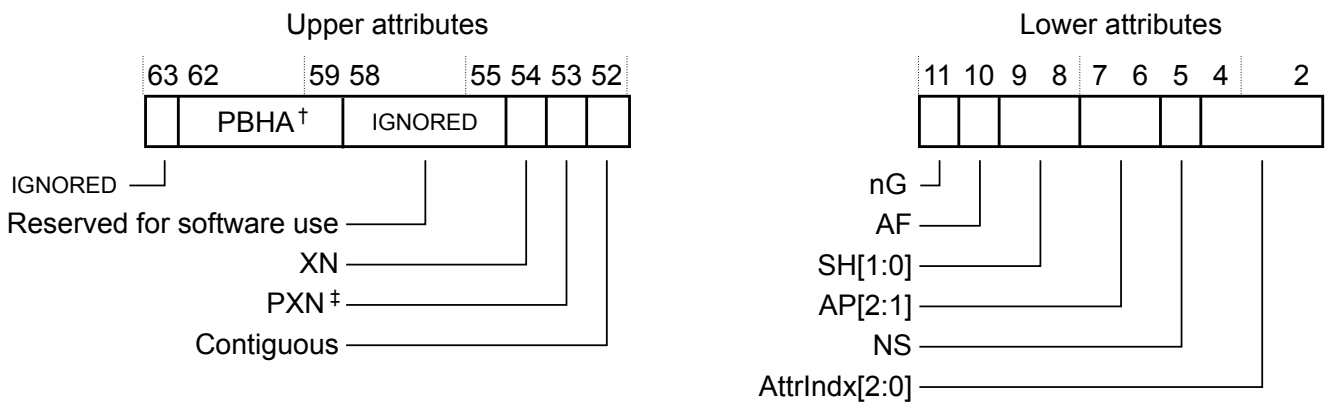

† IGNORED if FEAT_HPDS2 is not implemented.
‡ RES0 for a translation regime that cannot apply to execution at EL0.

For a stage 1 descriptor, the attributes are:

PBHA, bits[62:59]

Page-based hardware attributes bits.

These bits are IGNORED when FEAT_HPDS2 is not implemented.

When FEAT_HPDS2 is implemented, the HTCR and the TTBCR2 registers both contain a control bit for each PBHA bit in the translation tables that they control. When the value of that control bit is 1, and the value of the corresponding Hierarchical permission disables bit is 1, hardware can use that PBHA bit for IMPLEMENTATION DEFINED purposes. When the PBHA bit is used for IMPLEMENTATION DEFINED purposes, the value of 0 in the PBHA bit is a safe default setting that gives the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.

The control bits for this feature are:

For a Non-secure EL2 translation regime:

HTCR.HWUnn

Controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of HTCR.HPD is 1.

For a PL1&0 translation regime:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTBCR2.HWU1nn

For the translation tables indicated by TTBR1, controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TTBCR2.HPD1 is 1 and the value of TTBCR.T2E is 1.

TTBCR2.HWU0nn

For the translation tables indicated by TTBR0, controls whether Block or Page descriptor bit[nn] can be used by hardware.

These controls apply only when the value of TTBCR2.HPD0 is 1 and the value of TTBCR.T2E is 1.

Implementation of FEAT_HPDS2 requires the implementation of FEAT_AA32HPD, which provides the Hierarchical permission disables bits. If FEAT_AA32HPD is implemented but FEAT_HPDS2 is not implemented, then the control bits are RAZ/WI but other aspects of FEAT_AA32HPD functionality are implemented. If neither feature is implemented, then:

- The control bits are RAZ/WI.
- The FEAT_AA32HPD identification registers indicate that the functionality is not supported, see FEAT_AA32HPD.
- The TTBCR2 register encoding is treated as unallocated.

XN, bit[54] The Execute-never field, see Access permissions for instruction execution.

PXN, bit[53] The Privileged execute-never field, see Access permissions for instruction execution.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

Contiguous, bit[52]

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see Contiguous bit.

nG, bit[11] The not global bit. Determines how the translation is marked in the TLB, see Global and process-specific translation table entries.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

AF, bit[10] The Access flag, see The Access flag.

SH, bits[9:8] Shareability field, see Memory region attributes.

AP[2:1], bits[7:6]

Access Permissions bits, see Memory access control.

Note

For consistency with the Short-descriptor translation table formats, the Long-descriptor format defines AP[2:1] as the Access Permissions bits, and does not define an AP[0] bit.

AP[1] is RES1 in the Non-secure EL2 stage 1 translation tables.

NS, bit[5] Non-secure bit. For memory accesses from Secure state, specifies whether the output address is in Secure or Non-secure memory, see Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format.

For memory accesses from Non-secure state, this bit is RES0 and is ignored by the PE.

AttrIndx[2:0], bits[4:2]

Stage 1 memory attributes index field, for the indicated Memory Attribute Indirection Register, see VMSAv8-32 Long-descriptor format memory region attributes.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see IGNORED. For more information about these fields, see Other fields in the Long-descriptor translation table format descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.5.3.3 Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block as shown for a stage 2 translation:

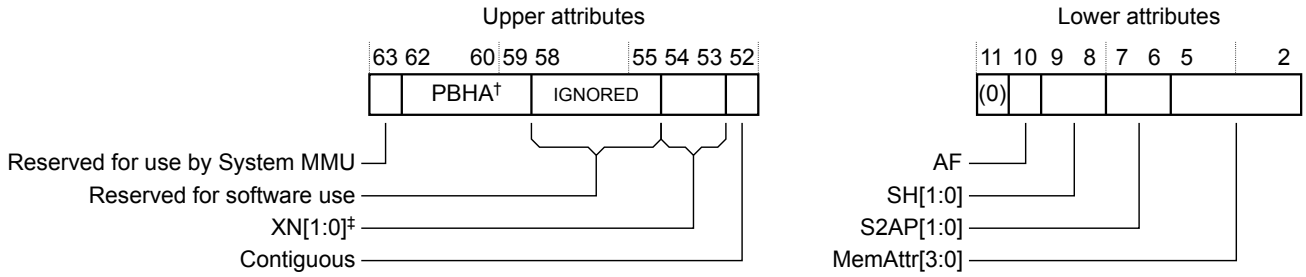

‡ Bit[53] is RES0 if FEAT_XNX is not implemented.

† Bits [62:60] are IGNORED and reserved for use by System MMU if FEAT_HPDS2 is not implemented.
Bits [59] is IGNORED if FEAT_HPDS2 is not implemented.

For a stage 2 descriptor, the attributes are:

PBHA[3:1], bits[62:60]

Page-based hardware attributes bits.

These bits are IGNORED and reserved for System MMU use when FEAT_HPDS2 is not implemented.

When FEAT_HPDS2 is implemented, VTCR_EL2 has a control bit for each PBHA bit in the EL1&0 stage 2 translation tables:

- When the value of that control bit is 1, hardware can use the corresponding PBHA bit for IMPLEMENTATION DEFINED purposes. When the PBHA bit is used for IMPLEMENTATION DEFINED purposes, the value of 0 in the PBHA bit is a safe default setting that gives the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.
- When the value of that control bit is 0, the corresponding PBHA bit is IGNORED and reserved for System MMU use.

PBHA[0], bit[59]

Page-based hardware attributes bit.

This bit is IGNORED when FEAT_HPDS2 is not implemented.

When FEAT_HPDS2 is implemented, VTCR_EL2 has a control bit for this bit in the EL1&0 stage 2 translation tables:

- When the value of that control bit is 1, hardware can use this bit for IMPLEMENTATION DEFINED purposes. When the PBHA bit is used for IMPLEMENTATION DEFINED purposes, the value of 0 in the PBHA bit is a safe default setting that gives the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.
- When the value of that control bit is 0, this bit is IGNORED.

XN[1:0], bits[54:53]

The stage 2 Execute-never field, see Access permissions for instruction execution.

If FEAT_XNX is not implemented, bit[53] is RES0.

Contiguous, bit[52]

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see Contiguous bit.

AF, bit[10] The Access flag, see The Access flag.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH, bits[9:8] Shareability field, see EL2 control of Non-secure memory region attributes.

S2AP, bits[7:6]

Stage 2 Access Permissions bits, see Hyp mode control of Non-secure access permissions.

Note

In the original VMSAv7-32 Long-descriptor attribute definition, this field was called HAP[2:1], for consistency with the AP[2:1] field in the stage 1 descriptors and despite there being no HAP[0] bit. VMSAv8-32 renames the field for greater clarity.

MemAttr, bits[5:2]

Stage 2 memory attributes, see EL2 control of Non-secure memory region attributes.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see IGNORED. For more information about these fields, see Other fields in the Long-descriptor translation table format descriptors.

G5.5.4 Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format

Access to the Secure or Non-secure PA map describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Long-descriptor format:

- The NS bit relates only to the memory block or page at the output address defined by the descriptor.
- The descriptors also include an NSTable bit, see Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format.

The NS and NSTable bits are valid only for memory accesses from Secure state. Memory accesses from Non-secure state ignore the values of these bits.

G5.5.4.1 Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format

For Long-descriptor Format Table descriptors for stage 1 translations, the descriptor includes an NSTable bit, which indicates whether the table identified in the descriptor is in Secure or Non-secure memory. For accesses from Secure state, the meaning of the NSTable bit is:

NSTable == 0 The defined table address is in the Secure PA map. In the descriptors in that translation table, NS bits and NSTable bits have their defined meanings.

NSTable == 1 The defined table address is in the Non-secure PA map. Because this table is fetched from the Non-secure address map, the NS and NSTable bits in the descriptors in this table must be ignored. This means that, for this table:

- The value of the NS bit in any Block or Page descriptor is ignored. The block or page address refers to Non-secure memory.
- The value of the NSTable bit in any Table descriptor is ignored, and the table address refers to Non-secure memory. When this table is accessed, the NS bit in any Block or Page descriptor is ignored, and all descriptors in the table refer to Non-secure memory.

In addition, an entry fetched in Secure state is treated as non-global if it is read from Non-secure memory. That is, these entries must be treated as if nG==1, regardless of the value of the nG bit. For more information about the nG bit, see Global and process-specific translation table entries.

The effect of NSTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to NSTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- When using the Long-descriptor Format, Table descriptors are defined only for the level 1 and level 2 of lookup.
- Stage 2 translations are performed only for operations in Non-secure state, that can access only the Non-secure address map. Therefore, the stage 2 descriptors do not include NS or NSTable bits.

G5.5.5 Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format

As described in Determining the translation table base address in the VMSAv8-32 translation regimes, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. The Long-descriptor translation table format provides more flexibility in defining the boundary between using TTBR0 and using TTBR1. When a PL1&0 stage 1 address translation is enabled, TTBR0 is always used. If TTBR1 is also used then:

- TTBR1 is used for the top part of the input address range.
- TTBR0 is used for the bottom part of the input address range.

The TTBCR.T0SZ and TTBCR.T1SZ size fields control the use of TTBR0 and TTBR1, as Table G5-2 shows.

Table G5-2 Use of TTBR0 and TTBR1, Long-descriptor format

TTBCR		Input address range using:	
T0SZ	**T1SZ**	**TTBR0**	**TTBR1**
`0b000`	`0b000`	All addresses	Not used
M[a]	`0b000`	Zero to $(2^{(32-M)}-1)$	2^{32-M} to maximum input address
`0b000`	N[a]	Zero to $(2^{32}-2^{(32-N)}-1)$	$2^{32}-2^{(32-N)}$ to maximum input address
M[a]	N[a]	Zero to $(2^{(32-M)}-1)$	$2^{32}-2^{(32-N)}$ to maximum input address

a. M, N must be greater than 0. The maximum possible value for each of T0SZ and T1SZ is 7.

For stage 1 translations, the input address is always a VA, and the maximum possible VA is $(2^{32}-1)$.

When address translation is using the Long-descriptor translation table format:

- Figure G5-12 shows how, when TTBCR.T1SZ is zero, the value of TTBCR.T0SZ controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

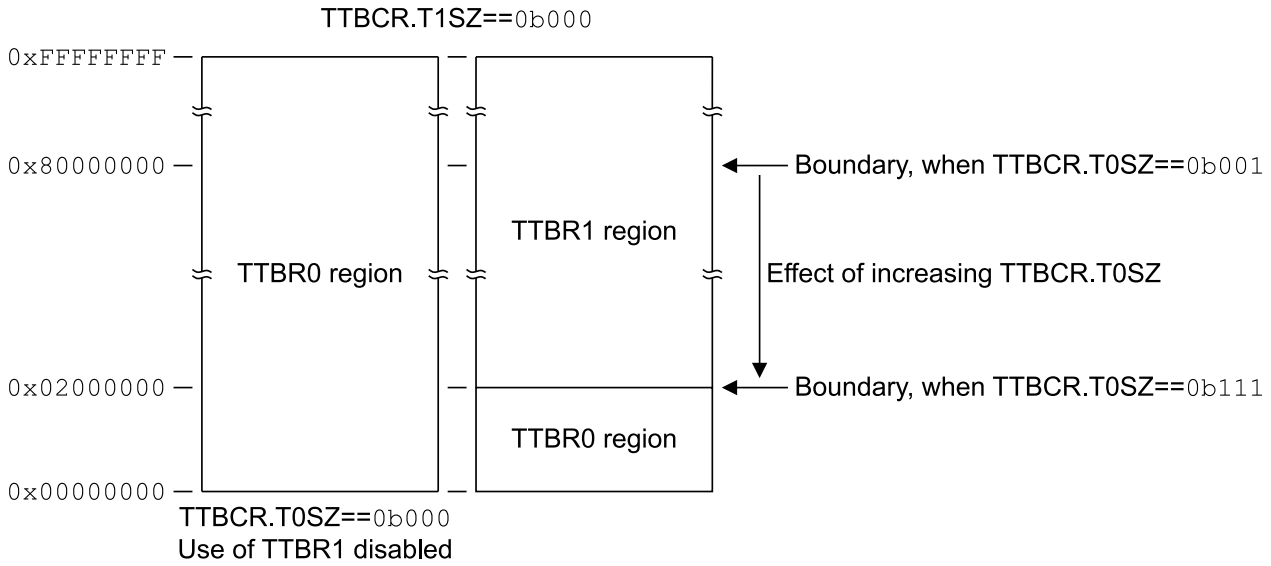

Figure G5-12 Control of TTBR boundary, when TTBCR.T1SZ is zero

- Figure G5-13 shows how, when TTBCR.T1SZ is nonzero, the values of TTBCR.T0SZ and TTBCR.T1SZ control the boundaries between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

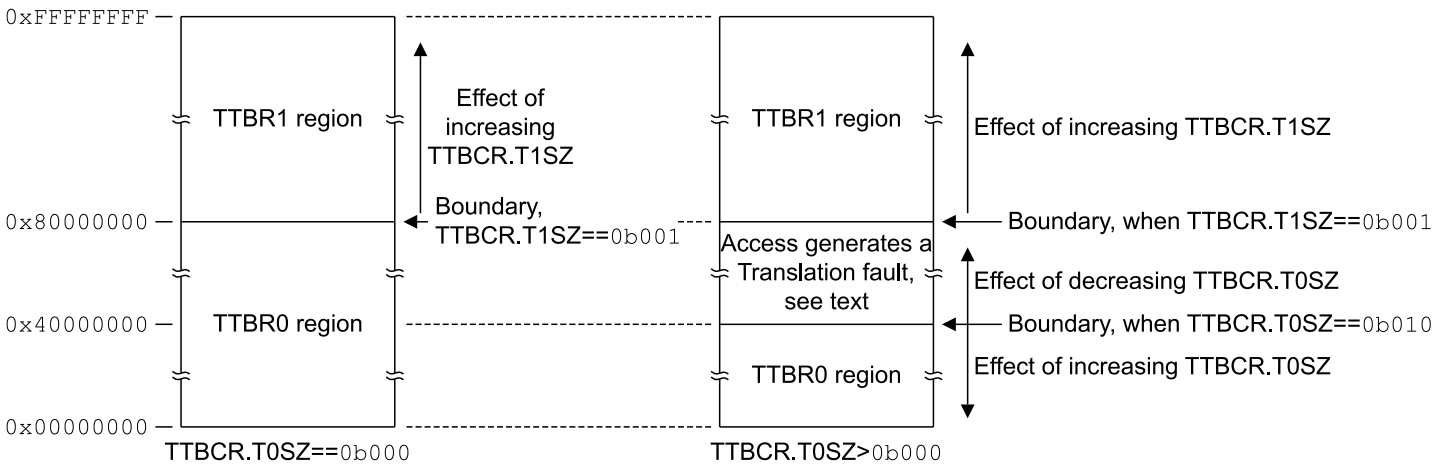

Figure G5-13 Control of TTBR boundaries, when TTBCR.T1SZ is nonzero

When T0SZ and T1SZ are both nonzero:

— If both fields are set to `0b001`, the boundary between the two regions is `0x8000_0000`. This is identical to having T0SZ set to `0b000` and T1SZ set to `0b001`.

— Otherwise, the TTBR0 and TTBR1 regions are non-contiguous. In this case, any attempt to access an address that is in that gap between the TTBR0 and TTBR1 regions generates a Translation fault.

Note

The handling of the Contiguous bit can mean that the boundary between the translation regions defined by the TCR_EL1.TnSZ values and the region for which an access generates a Translation fault is wider than shown in Figure G5-13. That is, if the descriptor for an access to the region shown as generating a fault has the Contiguous bit set to 1, the access might not generate a fault. Possible errors in programming the translation table registers describes this possibility.

When using the Long-descriptor translation table format:

- The TTBCR contains fields that define memory region attributes for the translation table walk, for each TTBR. These are the SH0, ORGN0, IRGN0, SH1, ORGN1, and IRGN1 bits.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TTBR0 and TTBR1 each contain an ASID field, and the TTBCR.A1 field selects which ASID to use.

For this translation table format, VMSAv8-32 Long-descriptor translation table format address lookup levels summarizes the lookup levels, and Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format describes the possible translations.

G5.5.5.1 Possible errors in programming the translation table registers

In all the descriptions in this subsection, the *size of the input address* supported for a PL1&0 stage 1 translation refers to the size specified by a TTBCR.TxSZ field.

Note

For a PL1&0 stage 1 translation, the input address range can be split so that the lower addresses are translated by TTBR0 and the higher addresses are translated by TTBR1. In this case, each of input address sizes specified by TTBCR.{T0SZ, T1SZ} is smaller than the total address size supported by the stage of translation.

The following are possible errors in the programming of TTBR0, TTBR1, and TTBCR. For the translation of a particular address at a particular stage of translation, either:

- The block size being used to translate the address is larger than the size of the input address supported at a stage of translation used in performing the required translation. This can occur only for the PL1&0 stage 1 translations, and only when either TTBCR.T0SZ or TTBCR.T1SZ is zero, meaning there is no gap between the address range translated by TTBR0 and the range translated by TTBR1. In this case, this programming error occurs if a block translated from the region that has TxSZ set to zero straddles the boundary between the two address ranges. Example G5-2 shows an example of this mis-programming.
- The address range translated by a set of blocks marked as contiguous, by use of the contiguous bit, is larger than the size of the input address supported at a stage of translation used in performing the required translation.

Example G5-2 Error in programming the translation table registers

If TTBCR.T0SZ is programmed to 0 and TTBCR.T1SZ is programmed to 7, this means:

- TTBR0 translates addresses in the range `0x0000_0000-0xFDFF_FFFF`.
- TTBR1 translates addresses in the range `0xFE00_0000-0xFFFF_FFFF`.

The translation table indicated by TTBR0 might be programmed with a block entry for a 1GB region starting at `0xC000_0000`. This covers the address range `0xC000_0000-0xFFFF_FFFF`, that overlaps the TTBR1 address range. This means this block size is larger than the input address size supported for translations using TTBR0, and therefore this is a programming error.

To understand why this must be a programming error, consider a memory access to address `0xFFFF_0000`. According to the TTBCR.{T0SZ, T1SZ} values, this must be translated using TTBR1. However, the access matches a TLB entry for the translation, using TTBR0, of the block at `0xC000_0000`. Hardware is not required to detect that the access to `0xFFFF_0000` is being translated incorrectly.

In these cases, an implementation might use one of the following approaches:

- Treat such a block as causing a Translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation.
  The block might be a block within a contiguous set of blocks.
- Treat such a block as not causing a Translation fault, even though the address accessed within that block is outside the size of the input address supported at a stage of translation, provided that both of the following apply:
  - The block is valid.
  - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation.

  The block might be a block within a contiguous set of blocks.

Additional constraints apply to programming the VTCR, see Determining the required initial lookup level for stage 2 translations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.5.6 VMSAv8-32 Long-descriptor translation table format address lookup levels

As stated at the start of this section, because the Long-descriptor translation table format is used for the Non-secure PL1&0 stage 2 translations, the format must support input addresses of up to 40 bits.

Table G5-3 summarizes the properties of the different levels of address lookup when using this format.

Table G5-3 Properties of the three levels of address lookup with VMSAv8-32 Long-descriptor translation tables

Level	Input address		Output address[a]		Number of entries
	Size	Address range[b]	Size	Address range	
First	Up to 512GB	Up to Address[38:0]	1GB	Address[39:30]	Up to 512
Second	Up to 1GB	Up to Address[29:0]	2MB	Address[39:21]	Up to 512
Third	2MB	Address[20:0]	4KB	Address[39:12]	512

a. Output address when an entry addresses a block of memory or a memory page. If an entry addresses the next level of address lookup it specifies Address[39:12] for the next-level translation table.

b. Input address range for the translation table. See Use of concatenated level 1 translation tables for details of support for additional bits of address at a given level, including possible support of a 40-bit input address range for stage 2 translations at level 1. For stage 1 translations at level 1 the input address range is limited to the VA size of [31:0].

For level 1 and level 2 tables, reducing the input address range reduces the number of addresses in the table and therefore reduces the table size. The appropriate Translation Table Control Register specifies the input address range.

Stage 1 translations require an input address range of up to 32 bits, corresponding to VA[31:0]. For these translations:

- For a memory access from a mode other than Hyp mode, the Secure or Non-secure TTBR0 or TTBR1 holds the translation table base address, and the Secure or Non-secure TTBCR is the control register.
- For a memory access from Hyp mode, HTTBR holds the translation table base address, and HTCR is the control register.

Note

For translations controlled by TTBR0 and TTBR1, if neither TTBR has an input address range larger than 1GB, then translation starts at level 2. Together, TTBR0 and TTBR1 can still cover the 32-bit VA input address range.

Stage 2 translations require an input address range of up to 40 bits, corresponding to IPA[39:0], and the supported input address size is configurable in the range 25-40 bits. Table G5-3 indicates a requirement for the translation mechanism to support a 39-bit input address range, Address[38:0]. Use of concatenated translation tables for the initial stage 2 lookup describes how a 40-bit IPA address range is supported. For stage 2 translations:

- VTTBR holds the translation table base address, and VTCR is the control register.
- If a supplied input address is larger than the configured input address size, a Translation fault is generated.

G5.5.6.1 Use of concatenated translation tables for the initial stage 2 lookup

If a stage 2 translation would require 16 entries or fewer in its top-level translation table, that stage of translation can, instead, be configured so that:

- It requires the corresponding number of concatenated translation tables at the next translation level, aligned to the size of the block of concatenated translation tables.
- The stage 2 translation starts at that next translation level.

Note

Stage 2 translations always use the Long-descriptor translation table format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This use of concatenated translation tables is:

- Required when the stage 2 translation supports a 40-bit input address range, see Use of concatenated level 1 translation tables.
- Supported for a stage 2 translation with an input address range of 31-34 bits, see Use of concatenated level 2 translation tables.

The use of concatenated translation tables requires the software that is defining the translation to:

- Define the concatenated translation tables with the required overall alignment.
- Program VTTBR to hold the address of the first of the concatenated translation tables.
- Program VTCR to indicate the required input address range and initial lookup level.

Note

The use of concatenated translation tables avoids the overhead of an additional level of translation.

G5.5.6.1.1 Use of concatenated level 1 translation tables

The Long-descriptor format translation tables provide 9 bits of address resolution at each level of lookup. However, a 40-bit input address range with a translation granularity of 4KB requires a total of 28 bits of address resolution. Therefore, a stage 2 translation that supports a 40-bit input address range requires two concatenated level 1 translation tables, together aligned to 8KB, where:

- The table at the address with PA[12:0]==`0b0_0000_0000_0000` defines the translations for input addresses with bit[39]==0.
- The table at the address with PA[12:0]==`0b1_0000_0000_0000` defines the translations for input addresses with bit[39]==1.
- The 8KB alignment requirement means that both tables have the same value for PA[39:13].

G5.5.6.1.2 Use of concatenated level 2 translation tables

A stage 2 translation with an input address range of 31-34 bits can start the translation either:

- With a level 1 lookup, accessing a level 1 translation table with 2-16 entries.
- With a level 2 lookup, accessing a set of concatenated level 2 translation tables.

Table G5-4 shows these options, for each of the input address ranges that can use this scheme.

Note

Because these are stage 2 translations, the input address range is an IPA range.

Table G5-4 Possible uses of concatenated translation tables for level 2 lookup

Input address range		Lookup starts at level 1	Lookup starts at level 2	
IPA range	Size	Required level 1 entries	Number of concatenated tables	Required alignment[a]
IPA[30:0]	2^{31} bytes	2	2	8KB
IPA[31:0]	2^{32} bytes	4	4	16KB
IPA[32:0]	2^{33} bytes	8	8	32KB
IPA[33:0]	2^{34} bytes	16	16	64KB

a. Required alignment of the set of concatenated level 2 tables.

See also Determining the required initial lookup level for stage 2 translations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.5.7 Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format

Figure G5-2 shows the possible address translations. If a stage of translation is controlled from an Exception level that is using AArch32, the input and output address constraints and the registers that control the translation are as follows:

Stage 1 translations

For all stage 1 translations:

- The input address range is up to 32 bits, as determined by either:
  - TTBCR.T0SZ or TTBCR.T1SZ, for a PL1&0 stage 1 translation.
  - HTCR.T0SZ, for an EL2 stage 1 translation.
- The output address range is 40 bits.

The stage 1 translations are:

Non-secure PL1&0 stage 1 translation

The stage 1 translation for memory accesses from Non-secure modes other than Hyp mode. This translates a VA to an IPA. For this translation, when Non-secure EL1 is using AArch32:

- Non-secure TTBR0 or TTBR1 holds the translation table base address.
- Non-secure TTBCR determines which TTBR is used.

Non-secure EL2 stage 1 translation

The stage 1 translation for memory accesses from Hyp mode, translates a VA to a PA. For this translation, when EL2 is using AArch32, HTTBR holds the translation table base address.

Secure PL1&0 stage 1 translation

The stage 1 translation for memory accesses from Secure modes, translates a VA to a PA. For this translation, when the Secure PL1 modes are using AArch32:

- Secure TTBR0 or TTBR1 holds the translation table base address.
- Secure TTBCR determines which TTBR is used.

Stage 2 translation

Non-secure PL1&0 stage 2 translation

The stage 2 translation for memory accesses from Non-secure modes other than Hyp mode, and translates an IPA to a PA. For this translation, when EL2 is using AArch32:

- The input address range is 40 bits, and VTCR.T0SZ determines the input address size.
- The output address range depends on the implemented memory system, and is up to 40 bits.
- VTTBR holds the translation table base address.
- VTCR specifies the required input address range, and whether the initial lookup is at level 1 or at level 2.

The descriptions of the VMSAv8-32 translation stages state that the maximum output address size is 40 bits. However, the register and Long-descriptor Format descriptor fields that hold these addresses are 48 bits wide. If bits[47:40] of an output address are not all zero, then the address generates an Address size fault.

The Long-descriptor translation table format provides up to three levels of address lookup, as described in VMSAv8-32 Long-descriptor translation table format address lookup levels, and the initial lookup, in which the MMU reads the translation table base address, is at either level 1 or level 2. The following determines the level of the initial lookup:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a stage 1 translation, the required input address range. For more information, see Determining the required initial lookup level for stage 1 translations.
- For a stage 2 translation, the level specified by the VTCR.SL0 field. For more information, see Determining the required initial lookup level for stage 2 translations.

Note

For a stage 2 translation, the size of the required input address range constrains the VTCR.SL0 value.

Figure G5-14 shows how the descriptor address for the initial lookup for a translation using the Long-descriptor translation table format is determined from the input address and the TTBR value. This figure shows the lookup for a translation that starts with a level 1 lookup, that translates bits[39:30] of the input address, zero extended if necessary.

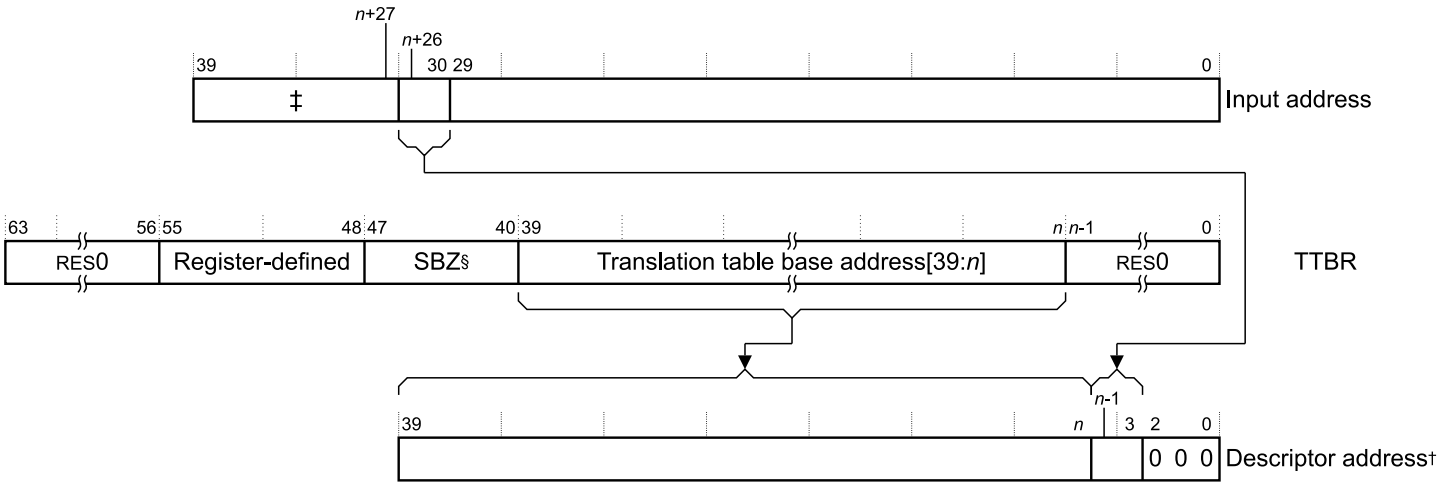

See text for more information about the translation table base register used, and the value of *n*.
‡ This field is absent if *n* is 13.
† For a Non-secure PL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.
§ See the lookup description for more information about bits[40:47] of the TTBR

Figure G5-14 VMSAv8-32 Long-descriptor initial lookup, starting at level 1

If bits[47:40] of the TTBR are not zero then the initial lookup will generate an Address size fault, see Address size fault.

For a translation that starts with a level 1 lookup, as shown in Figure G5-14:

For a stage 1 translation

n is in the range 4-5 and:

- For a memory access from Hyp mode:
  - HTTBR is the TTBR.
  - *n*=5-(HTCR.T0SZ).
- For other accesses:
  - The Secure or Non-secure instance of TTBR0 or TTBR1 is the TTBR.
  - *n*=(5-TTBCR.T*x*SZ), where *x* is 0 when using TTBR0, and 1 when using TTBR1.

For a stage 2 translation

n is in the range 4-13 and:

- VTTBR is the TTBR.
- *n*=5-(VTCR.T0SZ).

For a translation that starts with a level 2 lookup, the descriptor address is obtained in the same way, except that bits[(*n*+17):21] of the input address provide bits[(*n*-1):3] of the descriptor address, where:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a stage 1 translation

n is in the range 7-12. As Determining the required initial lookup level for stage 1 translations shows, for a stage 1 translation to start with a level 2 lookup, the corresponding T0SZ or T1SZ field must be 2 or more. This means:

- For a memory access from Hyp mode, *n*=14-HTCR.T0SZ.
- For other memory accesses, *n*=14-(TTBCR.T*x*SZ), where *x* is 0 when using TTBR0, and 1 when using TTBR1.

For a stage 2 translation

n is in the range 7-16. For a stage 2 translation to start with a level 2 lookup, VTCR.SL0 is `0b00`, and *n*=14-(VTCR.T0SZ).

The following sections describe how the level of the initial lookup is determined:

- Determining the required initial lookup level for stage 1 translations.
- Determining the required initial lookup level for stage 2 translations.

Address translation examples using the VMSAv8-32 Long descriptor translation table format shows examples of full translation flows, to an entry for a 4KB memory page, for lookups starting at level 1 and lookups starting at level 2.

G5.5.7.1 Determining the required initial lookup level for stage 1 translations

For a stage 1 translation, the required input address range, indicated by a T0SZ or T1SZ field in a translation table control register, determines the initial lookup level. The size of this input address region is $2^{(32-T\mathit{x}SZ)}$ bytes, and if this size is:

- Less than or equal to 2^{30} bytes, the required start is at level 2, and translation requires two levels of table to map to 4KB pages. This corresponds to a T*x*SZ value of 2 or more.
- More than 2^{30} bytes, the required start is at level 1, and translation requires three levels of table to map to 4KB pages. This corresponds to a T*x*SZ value that is less than 2.

For the PL1&0 stage 1 translations, the TTBCR:

- Splits the 32-bit VA input address range between TTBR0 and TTBR1, see Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format.
- Holds the input address range sizes for TTBR0 and TTBR1, in the TTBCR.T0SZ and TTBCR.T1SZ fields.

For the EL2 stage 1 translations, HTCR.T0SZ indicates the size of the required input address range. For example, if this field is `0b000`, it indicates a 32-bit VA input address range, and translation lookup must start at level 1.

G5.5.7.2 Determining the required initial lookup level for stage 2 translations

For a PL1&0 stage 2 translation, the output address range from the PL1&0 stage 1 translations determines the required input address range for the stage 2 translation.

VTCR.SL0 indicates the starting level for the lookup. The permitted SL0 values are:

`0b00` Stage 2 translation lookup must start at level 2.

`0b01` Stage 2 translation lookup must start at level 1.

In addition, VTCR.T0SZ must indicate the required input address range. The size of the input address region is $2^{(32-T0SZ)}$ bytes.

Note

VTCR.T0SZ holds a four-bit signed integer value, meaning it supports values from -8 to 7. This is different from the other translation control registers, where T*n*SZ holds a three-bit unsigned integer, supporting values from 0 to 7.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The programming of VTCR must follow the constraints shown in Table G5-5, otherwise any attempt to perform a translation table walk that uses the stage 2 address translation generates a stage 2 level 1 Translation Fault. The table also shows how the VTCR.SL0 and VTCR.T0SZ values determine the VTTBR.BADDR field width.

Note

If VTCR.SL0 is programmed to a reserved value then the constraints shown in Table G5-5 are not met, and a translation table walk that uses stage 2 translation generates a stage 2 level 1 Translation fault.

Table G5-5 Input address range constraints on programming VTCR

VTCR.SL0	VTCR.T0SZ	Input address range, R	Initial lookup level	BADDR[39:x] width[a]
0b00	2 to 7	$R \leq 2^{30}$ bytes	Level 2	[39:12] to [39:7]
0b00	-2 to 1	$2^{30} < R \leq 2^{34}$ bytes	Level 2	[39:16] to [39:13]
0b01	-2 to 1		Level 1	[39:7] to [39:4]
0b01	-8 to -3	$2^{34} < R$	Level 1	[39:13] to [39:8]

a. The first range corresponds to the first T0SZ value, the second range to the second T0SZ value.

In addition, VTCR.S must be programmed to the value of T0SZ[3], otherwise behavior is CONSTRAINED UNPREDICTABLE with the resulting behavior being that VTCR.T0SZ is treated as an UNKNOWN value.

Note

VTCR.T0SZ being treated as an UNKNOWN value results in a stage 2 level 1 Translation Fault if that UNKNOWN value is not consistent with the programmed value of VTCR.SL0.

CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR describes these CONSTRAINED UNPREDICTABLE behaviors.

Where necessary, the initial lookup level provides multiple concatenated translation tables, as described in Use of concatenated level 2 translation tables. This section also gives more information about the alternatives, shown in Table G5-5, when R is in the range 2^{31} - 2^{34}.

G5.5.8 The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format

This section gives the algorithm for finding the translation table entry that corresponds to a given IA, for each required level of lookup. The algorithm encodes the descriptions of address translation given earlier in this section. The VMSAv8-32 Long-descriptor format uses a 4KB translation granule.

The description uses the following terms:

BaseAddr The base address for the level of lookup, as defined by:

- For the initial lookup level, the TTBR.BADDR base address field in the appropriate TTBR, see the description of T*n*SZ.
- Otherwise, the translation table address returned by the previous level of lookup.

IA The supplied IA for this stage of translation.

T*n*SZ The translation table size for this stage of translation:

For PL1&0 stage 1 Either:

- TTBCR.T0SZ if the translation is using TTBR0.
- TTBCR.T1SZ if the translation is using TTBR1.

For PL1&0 stage 2 VTCR.T0SZ. The translation uses VTTBR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For EL2 stage 1 HTCR.T0SZ. The translation uses HTTBR.

SL0 VTCR.SL0. Applies to the Non-secure PL1&0 stage 2 translation only.

Table G5-6 shows the Translation Table descriptor address, for each level of lookup. The table shows only architecturally-valid programming of the TCR. See also Possible errors in programming the translation table registers.

Table G5-6 Translation table entry addresses, VMSAv8-32 using Long-descriptor format

Lookup level	Entry address and conditions Stage 1 translation	Stage 2 translation	General conditions
One	BaseAddr[39:*x*]:IA[*y*:30]:`0b000` if[a] $0 \le$ T*n*SZ ≤ 1 then $x = (5 - \text{T}n\text{SZ})$	BaseAddr[39:*x*]:IA[*y*:30]:`0b000` if SL0[b] == 1 then if[a] $-8 \le$ T0SZ ≤ 1 then $x = (5 - \text{T0SZ})$	$y = (x + 26)$
Two	BaseAddr[39:*x*]:IA[*y*:21]:`0b000` if[a] $2 \le$ T*n*SZ ≤ 7 then $x = (14 - \text{T}n\text{SZ})$ else[c] $x = 12$	BaseAddr[39:*x*]:IA[*y*:21]:`0b000` if SL0 == 0 then if[a] $-2 \le$ T0SZ ≤ 7 then $x = (14 - \text{T0SZ})$ elsif[c] SL0[b] == 1 then $x = 12$	$y = (x + 17)$
Three	BaseAddr[39:12]:IA[20:12]:`0b000`	BaseAddr[39:12]:IA[20:12]:`0b000`	-

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see Use of concatenated translation tables for the initial stage 2 lookup.

b. SL0 == 0 if the initial lookup is level 2, SL0 == 1 if the initial lookup is level 1.

c. This is the case where this level of lookup is not the initial level of lookup.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.6 Memory access control

In addition to an output address, a translation table entry that refers to page or region of memory includes fields that define properties of the target memory region. Information returned by a translation table lookup describes the classification of those fields as address map control, access control, and memory attribute fields. The access control fields, described in this section, determine whether the PE, in its current state, is permitted to perform the required access to the output address given in the Translation Table descriptor. If a translation stage does not permit the access, then an MMU fault is generated for that translation stage, and no memory access is performed.

The following sections describe the memory access controls:

- About access permissions.
- About the PAN bit.
- Access permissions for instruction execution.
- Domains, Short-descriptor format only.
- The Access flag.
- Hyp mode control of Non-secure access permissions.

G5.6.1 About access permissions

The Translation Table descriptors include fields that define access permissions for data accesses and for instruction fetches. This section introduces those fields. In addition:

- System register controls can prevent execution from writable locations, see Preventing execution from writable locations.
- In Armv8.1, the PSTATE.PAN can affect the access permissions for privileged data accesses, see About the PAN bit.

Note

This section gives a general description of memory access permissions. Software executing at PL1 in Non-secure state can see only the access permissions defined by the Non-secure PL1&0 stage 1 translations. However, software executing at EL2 can modify these permissions, as described in Hyp mode control of Non-secure access permissions. This modification is invisible to Non-secure software executing at EL1 or EL0.

Access permission bits in a Translation Table descriptor control access to the corresponding memory region. The details of this control depend on the translation table format, as follows:

Short-descriptor format

This format supports two options for defining the access permissions:

- Three bits, AP[2:0], define the access permissions.
- Two bits, AP[2:1], define the access permissions, and AP[0] can be used as an Access flag.

SCTLR.AFE selects the access permissions option. Setting this bit to 1, to enable the Access flag, also selects use of AP[2:1] to define access permissions.

Arm deprecates any use of the AP[2:0] scheme for defining access permissions.

Long-descriptor format

AP[2:1] to control the access permissions, and the descriptors provide an AF bit for use as an Access flag. This means VMSAv8-32 behaves as if the value of SCTLR.AFE is 1, regardless of the value that software has written to this bit.

Note

When use of the Long-descriptor format is enabled, SCTLR.AFE is UNK/SBOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Access flag describes the Access flag, for both translation table formats.

The XN and PXN bits provide additional access controls for instruction fetches, see Access permissions for instruction execution.

An attempt to perform a memory access that the translation table access permission bits do not permit generates a Permission fault, for the corresponding stage of translation. However, when using the Short-descriptor translation table format, it generates the fault only if the access is to memory in the Client domain, see Domains, Short-descriptor format only.

Note

For the Non-secure PL1&0 translation regime, memory accesses are subject to two stages of translation. Each stage of translation has its own, independent, fault checking. Fault handling is different for the two stages, see Exception reporting in a VMSAv8-32 implementation.

The following sections describe the two access permissions models:

- AP[2:1] access permissions model.
- AP[2:0] access permissions control, Short-descriptor format only. This section includes some information on access permission control in earlier versions of the Arm VMSA.

G5.6.1.1 AP[2:1] access permissions model

Note

Arm recommends that this model is always used, even where the AP[2:0] model is permitted. Some documentation describes the AP[2:1] model as the *simplified access permissions* model.

This access permissions model is used if the translation is either:

- Using the Long-descriptor translation table format.
- Using Short-descriptor translation table format, and the SCTLR.AFE bit is set to 1.

In this model:

- One bit, AP[2], selects between read-only and read/write access.
- A second bit, AP[1], selects between Application level (EL0) and System level (PL1) control.
  For the Non-secure EL2 stage 1 translations, AP[1] is SBO.

This provides four access combinations:

- Read-only at all privilege levels.
- Read/write at all privilege levels.
- Read-only at PL1, no access by software executing at EL0.
- Read/write at PL1, no access by software executing at EL0.

Table G5-7 shows this access control model.

Table G5-7 VMSAv8-32 AP[2:1] access permissions model

AP[2], disable write access	AP[1], enable unprivileged access	Access
0	0[a]	Read/write, only at PL1
0	1	Read/write, at any privilege level

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AP[2], disable write access	AP[1], enable unprivileged access	Access
1	0[a]	Read-only, only at PL1
1	1	Read-only, at any privilege level

a. Not valid for Non-secure EL2 stage 1 translation tables. AP[1] is SBO in these tables.

G5.6.1.1.1 Hierarchical control of access permissions, Long-descriptor format

The Long-descriptor translation table format introduces a mechanism that entries at one level of translation table lookup can use to set limits on the permitted entries at subsequent levels of lookup. This applies to the access permissions, and also to the restrictions on instruction fetching described in Hierarchical control of instruction fetching, Long-descriptor format.

The restrictions apply only to subsequent levels of lookup at the same stage of translation. The APTable[1:0] field restricts the access permissions, as Table G5-8 shows.

However, in an implementation that includes FEAT_AA32HPD, when hierarchical control of data access permissions is disabled for a translation regime, the information in this subsection does not apply. See Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors.

As stated in the table footnote, for the Non-secure EL2 stage 1 translation tables, APTable[0] is reserved, SBZ.

Table G5-8 Effect of APTable[1:0] on subsequent levels of lookup

APTable[1:0]	Effect
00	No effect on permissions in subsequent levels of lookup.
01[a]	Access at EL0 not permitted, regardless of permissions in subsequent levels of lookup.
10	Write access not permitted, at any Exception level, regardless of permissions in subsequent levels of lookup.
11[a]	Regardless of permissions in subsequent levels of lookup: • Write access not permitted, at any Exception level. • Read access not permitted at EL0.

a. Not valid for the Non-secure EL2 stage 1 translation tables. In those tables, APTable[0] is SBZ.

Note

The APTable[1:0] settings are combined with the translation table access permissions in the Translation Table descriptors accessed in subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The Long-descriptor format provides APTable[1:0] control only for the stage 1 translations. The corresponding bits are SBZ in the stage 2 Translation Table descriptors.

The effect of APTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to APTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

G5.6.1.2 AP[2:0] access permissions control, Short-descriptor format only

This access permissions model applies when using the Short-descriptor translation tables format, and the SCTLR.AFE bit is set to 0. Arm deprecates any use of this access permissions model.

When SCTLR.AFE is set to 0, ensuring that the AP[0] bit is always set to 1 effectively changes the access model to the simpler model described in AP[2:1] access permissions model.

Table G5-9 shows the full AP[2:0] access permissions model:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G5-9 VMSAv8-32 MMU access permissions

AP[2]	AP[1:0]	PL1 access	Unprivileged access	Description
0	00	No access	No access	All accesses generate Permission faults
	01	Read/write	No access	Access only at PL1
	10	Read/write	Read-only	Writes at EL0 generate Permission faults
	11	Read/write	Read/write	Full access
1	00	-	-	Reserved
	01	Read-only	No access	Read-only, only at PL1
	10	Read-only	Read-only	Read-only at any Exception level, deprecated[a]
	11	Read-only	Read-only	Read-only at any Exception level[b]

a. From VMSAv7, Arm strongly recommends use of the `0b11` encoding for Read-only at any Exception level.

b. This mapping was introduced in VMSAv7, and is reserved in earlier versions of the VMSA.

Note

- VMSAv8-32 supports the full set of access permissions shown in Table G5-9 only when SCTLR.AFE is set to 0. When SCTLR.AFE is set to 1, the only supported access permissions are those described in AP[2:1] access permissions model.
- Some old documentation describes the AP[2] bit in the translation table entries as the APX bit.

G5.6.2 About the PAN bit

When the value of PSTATE.PAN is 1, any privileged data access from PL1 or EL2 to a virtual memory address that is accessible at EL0 generates a Permission fault.

When the value of PSTATE.PAN is 0, the translation system is the same as in Armv8.0.

A corresponding PAN bit is added to CPSR and SPSR for exception returns, and DSPSR for entry to and exit from Debug state.

A new SPAN bit is added to SCTLR that controls whether the PAN state bit is set on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

CPSR.PAN bit can be written using an `MSR` instruction at PL1 or higher. Data writes to CPSR.PAN using an `MSR` instruction at EL0 are ignored. The value that is returned for an `MRS` instruction of CPSR from EL0 is UNKNOWN. In keeping with all other writes to the CPSR, other than for instruction fetches, the effect of the PAN state bit does not need to be explicitly synchronized.

The PAN state bit has no effect on:

- Data Cache instructions.
- Address translation instructions, other than `ATS1CPRP` and `ATS1CPWP` when FEAT_PAN2 is implemented.
- Unprivileged instructions, `LDRBT`, `LDRHT`, `LDRT`, `LDRSBT`, `LDRSHT`, `STRBT`, `STRHT`, `STRT`, `STRSBT`, and `STRSHT`, unless the Effective value of HCR_EL2.{E2H, TGE} is {1, 0}.
- Instruction accesses.
- Manager domains.

The PAN bit has no effect when the first stage of translation is disabled for the current translation regime or when the first stage of translation for the current translation regime does not describe the permissions for access at EL0.

If access is disabled, then the access will give rise to a stage 1 Permission fault.

On an exception taken from AArch32:

- CPSR.PAN is copied to SPSR_ELx.PAN, when the target Exception level is AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CPSR.PAN is copied to SPSR.PAN, when the target Exception level is AArch32.

On an exception return from AArch32 to AArch32, SPSR.PAN is copied to CPSR.PAN.

On entry to Debug state, CPSR.PAN is copied to DSPSR.PAN.

On exit from Debug state, DSPSR.PAN is copied to CPSR.PAN.

The CPSR.PAN bit is not an Execution state bit.

Note

- In Non-debug state, in AArch32 state, software can use the `SETPAN #imm` instruction to modify PSTATE.PAN.
- In Debug state, in AArch32 state, a debugger can use the `ERET` instruction to perform a DRPS operation to modify PSTATE.PAN.

G5.6.3 Access permissions for instruction execution

Execute-never controls provide an additional level of control on memory accesses permitted by the access permissions settings. These controls are:

XN, Execute-never

Descriptor bit[54], defined as XN for:

- Stage 1 of any translation regime.
- Stage 2 translations when FEAT_XNX is not implemented.

Note

XN[1:0], Execute-never, stage 2 only describes the stage 2 control when FEAT_XNX is implemented.

This field applies to execution at any Exception level to which the stage of translation applies. A value of 0 indicates that this control permits execution.

PXN, Privileged execute-never, stage 1 only

Descriptor bit[53], used only for stage 1 of any translation regime for which the stage 1 translation can support two VA ranges:

- For stage 1 of a translation regime for which the stage 1 translation supports only a single VA range the stage 1 descriptors define a PXN field that is RES0, meaning it is ignored by hardware.

This field applies only to execution at an Exception level higher than EL0. A value of 0 indicates that this control permits execution.

XN[1:0], Execute-never, stage 2 only

Descriptor bits[54:53], defined as XN[1:0] for:

- Stage 2 translations when FEAT_XNX is implemented.

Table G5-10 shows the operation of this control.

Table G5-10 XN[1:0] stage 2 access permissions model

XN[1]	XN[0]	Access
0	0	The stage 2 control permits execution at EL1 and EL0 if read access is permitted.
0	1	The stage 2 control does not permit execution at EL1, but permits execution at EL0 if read access is permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

XN[1]	XN[0]	Access
1	0	The stage 2 control does not permit execution at EL1 or at EL0.
1	1	The stage 2 control permits execution at EL1 if read access is permitted, but does not permit execution at EL0.

Note

For stage 2 translations when FEAT_XNX is not implemented, descriptor bit[53] is RES0, meaning it is ignored by hardware, and bit[54] is the XN control, see XN, Execute-never.

Executing an instruction at ELx in a particular Security state generates a Permission fault unless all of the following are true for the instruction address:

- Any stage 1 execute-never control that applies to execution at ELx in the current Security state permits execution.
- If the translation regime that applies to ELx in the current Security state has two stages of translations, the stage 2 execute-never control that applies to execution at ELx permits execution.
- Read access is permitted.

However, if a stage 1 translation is using the Short-descriptor translation table format and the address is in a Managers domain the stage 1 access permissions are not checked, and therefore the access cannot cause a stage 1 Permission fault, see Domains, Short-descriptor format only.

See also Hyp mode control of Non-secure access permissions.

In addition, System register controls can enforce execute-never restrictions, regardless of the settings in the translation table XN and PXN fields, see:

- Restriction on Secure instruction fetch.
- Preventing execution from writable locations.

The execute-never controls apply also to speculative instruction fetching. This means a speculative instruction fetch from a memory region that is execute-never at the current level of privilege is prohibited.

The execute-never controls means that, when the stage of address translation is enabled, the PE can fetch, or speculatively fetch, an instruction from a memory location only if all of the following apply:

- If using the Short-descriptor translation table format, the Translation Table descriptor for the location does not indicate that it is in a No access domain.
- If using the Long-descriptor translation table format, or using the Short descriptor format and the descriptor indicates that the location is in a Client domain, in the descriptor for the location the following apply:
  - The stage 1 execute-never control for the Exception level at which the instruction is executed permits execution.
  - For a translation regime with two stages of address translation, the stage 2 execute-never control that applies to the Exception level at which the instruction is executed permits execution.
  - The access permissions permit a read access from the current PE mode.
- No other Prefetch Abort condition exists.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- The PXN control applies to the PE privilege when it attempts to execute the instruction. In an implementation that fetches instructions speculatively, this might not be the privilege when the instruction was prefetched. Therefore, the architecture does not require the PXN control to prevent instruction fetching.
- Although the XN control applies to speculative fetching, on a speculative instruction fetch from an XN location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an XN location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated.
- The software that defines a translation table must mark any region of memory that is read-sensitive as XN, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as XN. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
- When using the Short-descriptor translation table format, the XN attribute is not checked for domains marked as Manager. Therefore, the system must not include read-sensitive memory in domains marked as Manager, because the XN field does not prevent speculative fetches from a Manager domain.

When no stage of address translation for the translation regime is enabled, memory regions cannot have XN or PXN attributes assigned. Behavior of instruction fetches when all associated address translations are disabled describes how disabling all MMUs affects instruction fetching.

G5.6.3.1 Hierarchical control of instruction fetching, Long-descriptor format

The Long-descriptor translation table format introduces a mechanism that means entries at one level of translation tables lookup can set limits on the permitted entries at subsequent levels of lookup. This applies to the restrictions on instruction fetching, and also to the access permissions described in Hierarchical control of access permissions, Long-descriptor format.

Note

Similar hierarchical controls apply to data accesses, see Hierarchical control of access permissions, Long-descriptor format.

However, in an implementation that includes FEAT_AA32HPD, when hierarchical control of instruction fetching is disabled for a translation regime, the information in this subsection does not apply. See Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors.

The restrictions apply only to subsequent levels of lookup at the same stage of translation, and:

- XNTable restricts the XN control:
  - When XNTable is set to 1, the XN field is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the field.
  - When XNTable is set to 0 it has no effect.
- PXNTable restricts the PXN control:
  - When PXNTable is set to 1, the PXN field is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the field.
  - When PXNTable is set to 0 it has no effect.

Note

The XNTable and PXNTable settings are combined with the XN and PXN fields in the Translation Table descriptors accessed at subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The XNTable and PXNTable controls are provided only in the Long-descriptor translation table format, and only for stage 1 translations. The corresponding bits are SBZ in the stage 2 Translation Table descriptors.

The effect of XNTable or PXNTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to XNTable or PXNTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.6.3.2 Preventing execution from writable locations

The architecture provides control bits that, when the corresponding stage 1 address translation is enabled, force writable memory to be treated as XN or PXN, regardless of the value of the XN or PXN field. When the translation stages are controlled by an Exception level that is using AArch32:

- For PL1&0 stage 1 translations, when SCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.
- For PL1&0 stage 1 translations, when SCTLR.UWXN is set to 1, an instruction fetch is treated as accessing a PXN region if it accesses a region that software executing at EL0 can write to.
- For Non-secure EL2 stage 1 translations, when HSCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.

Note

- The SCTLR.WXN controls are intended to be used in systems with very high security requirements.
- Setting a WXN or UWXN bit to 1 changes the interpretation of the translation table entry, overriding a zero value of an XN or PXN field. It does not cause any change to the translation table entry.

For any given virtual machine, Arm expects WXN and UWXN to remain static in normal operation. In particular, it is IMPLEMENTATION DEFINED whether TLB entries associated with a particular VMID reflect the effect of the values of these fields. A generic sequence to ensure synchronization of a change to these fields, when that change is made without a corresponding change of VMID, is:

```
Change the WXN or UWXN bit
ISB                     ; This ensures synchronization of the change
Invalidate entire TLB of associated entries
DSB                     ; This completes the TLB Invalidation
ISB                     ; This ensures instruction synchronization
```

As with all Permission fault checking, if the stage 1 translation is using the Short-descriptor translation table format, the permission checks are performed only for Client domains. For more information, see About access permissions.

For more information about address translation, see About address translation for VMSAv8-32.

G5.6.3.3 Restriction on Secure instruction fetch

EL3 provides a Secure instruction fetch bit, SCR.SIF. When this bit is 1, any attempt in Secure state to execute an instruction fetched from Non-secure physical memory causes a Permission fault. As with all Permission fault checking, when using the Short-descriptor format translation tables the check applies only to Client domains, see About access permissions.

Arm expects SCR.SIF to be static during normal operation. In particular, whether the TLB holds the effect of the SIF bit is IMPLEMENTATION DEFINED. The generic sequence to ensure visibility of a change to the SIF bit is:

```
Change the SCR.SIF bit
ISB                     ; This ensures synchronization of the change
Invalidate entire TLB
DSB                     ; This completes the TLB Invalidation
ISB                     ; This ensures instruction synchronization
```

G5.6.4 Domains, Short-descriptor format only

A domain is a collection of memory regions. The Short-descriptor translation table format supports 16 domains, and requires the software that defines a translation table to assign each VMSAv8-32 memory region to a domain. When using the Short-descriptor format:

- Level 1 translation table entries for translation tables and Sections include a domain field.
- Translation table entries for Supersections do not include a domain field. The Short-descriptor format defines Supersections as being in domain 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Level 2 translation table entries inherit a domain setting from the parent level 1 Translation Table descriptor.
- Each TLB entry includes a domain field.

The domain field specifies which of the 16 domains the entry is in, and a two-bit field in the DACR defines the permitted access for each domain. The possible settings for each domain are:

No access Any access using the Translation Table descriptor generates a Domain fault.

Clients On an access using the Translation Table descriptor, the access permission attributes are checked. Therefore, the access might generate a Permission fault.

Managers On an access using the Translation Table descriptor, the access permission attributes are not checked. Therefore, the access cannot generate a Permission fault.

See The MMU fault-checking sequence for more information about how, when using the Short-descriptor translation table format, the Domain attribute affects the checking of the other attributes in the Translation Table descriptor.

Note

A single program might:

- Be a Client of some domains.
- Be a Manager of some other domains.
- Have no access to the remaining domains.

The Long-descriptor translation table format does not support domains. When a stage of translation is using this format, all memory is treated as being in a Client domain, and the settings in the DACR are ignored.

G5.6.5 The Access flag

The Access flag indicates when a page or section of memory is accessed for the first time since the Access flag in the corresponding Translation Table descriptor was set to 0:

- If address translation is using the Short-descriptor translation table format, it must set SCTLR.AFE to 1 to enable use of the Access flag. Setting this bit to 1 redefines the AP[0] bit in the Translation Table descriptors as an Access flag, and limits the access permissions information in the Translation Table descriptors to AP[2:1], as described in AP[2:1] access permissions model.
- The Long-descriptor format always supports an Access flag bit in the Translation Table descriptors, and address translation using this format behaves as if SCTLR.AFE is set to 1, regardless of the value of that bit.

In Armv8.0, the Access flag is managed by software as described in Software management of the Access flag.

Note

Previous versions of the Arm architecture optionally supported hardware management of the Access flag. Armv8.0 obsolete this option. However, FEAT_HAF provides a new mechanism for hardware management of the Access flag, that is supported only for the VMSAv8-64 translation regimes.

G5.6.5.1 Software management of the Access flag

Armv8.0 requires that software manages the Access flag. This means an Access flag fault is generated whenever an attempt is made to read into the TLB a Translation Table descriptor entry for which the value of the Access flag is 0.

Note

When using the Short-descriptor translation table format, Access flag faults are generated only if SCTLR.AFE is set to 1, to enable use of a Translation Table descriptor bit as an Access flag.

The Access flag mechanism expects that, when an Access flag fault occurs, software resets the Access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time that memory location is accessed. Entries with the Access flag set to 0 are never held in the TLB, meaning software does not have to flush the entry from the TLB after setting the flag.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If a system incorporates components that can autonomously update translation table entries that are shared with the Arm PE, then the software must be aware of the possibility that such components can update the access flag autonomously. In such a system, system software should perform any changes of translation table entries with an Access flag of 0, other than changes to the Access flag value, by using an Load-Exclusive/Store-Exclusive loop, to allow for the possibility of simultaneous updates.

G5.6.6 Hyp mode control of Non-secure access permissions

When EL2 is using AArch32, Non-secure software executing in Hyp mode controls two sets of translation tables, both of which use the Long-descriptor translation table format:

- The translation tables that control the Non-secure EL2 stage 1 translations. These map VAs to PAs, for memory accesses made when executing in Non-secure state in Hyp mode, and are indicated and controlled by the HTTBR and HTCR.

  These translations have similar access controls to other Non-secure stage 1 translations using the Long-descriptor translation table format, as described in:

  — AP[2:1] access permissions model.

  — Access permissions for instruction execution.

  The differences from the Non-secure stage 1 translations are that:

  — The APTable[0], PXNTable, and PXN bits are reserved, SBZ.

  — AP[1] is reserved, SBO.

- The translation tables that control the Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, for memory accesses made when executing in Non-secure state at PL1 or EL0, and are indicated and controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define stage 2 access permissions, that are combined with the permissions defined in the stage 1 translation. This section describes this combining of access permissions.

Note

The level 2 access permissions mean a hypervisor can define additional access restrictions to those defined by a Guest OS in the stage 1 translation tables. For a particular access, the actual access permission is the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

The stage 2 access controls defined from Hyp mode:

- Affect only the Non-secure stage 1 access permissions settings.
- Take no account of whether the accesses are from a Non-secure PL1 mode or a Non-secure EL0 mode.
- Permit software executing in Hyp mode to assign a write-only attribute to a memory region.

The S2AP field in the stage 2 descriptors define the stage 2 access permissions, as Table G5-11 shows:

Table G5-11 Stage 2 control of access permissions

S2AP	Access permission
00	No access permitted.
01	Read-only. Writes to the region are not permitted, regardless of the stage 1 permissions.
10	Write-only. Reads from the region are not permitted, regardless of the stage 1 permissions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S2AP	Access permission
11	Read/write. The stage 1 permissions determine the access permissions for the region.

For more information about the S2AP field, see Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors.

If the stage 2 permissions cause a Permission fault, this is a stage 2 MMU fault. Stage 2 MMU faults are taken to Hyp mode, and reported in the HSR using an EC code of `0x20` or `0x24`. For more information, see Use of the HSR.

Note

In the HSR, the combination of the EC code and the DFSC or IFSC value in the ISS indicate that the fault is a stage 2 MMU fault.

The stage 2 permissions include an XN attribute. If this identifies the region as execute-never, execution from the region is not permitted, regardless of the value of the XN or UXN attribute in the stage 1 translation. If a Permission fault is generated because the stage 2 XN field identifies the region as execute-never, this is reported as a stage 2 MMU fault.

Note

The stage 2 XN attribute:

- Is a single bit if FEAT_XNX is not implemented, see XN, Execute-never.
- Is a 2-bit field if FEAT_XNX is implemented, see XN[1:0], Execute-never, stage 2 only.

AArch32 state prioritization of synchronous aborts from a single stage of address translation describes the abort prioritization if both stages of a translation generate a fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.7 Memory region attributes

In addition to an output address, a translation table entry that refers to a page or region of memory includes fields that define properties of that target memory region. Information returned by a translation table lookup describes the classification of those fields as address map control, access control, and memory attribute fields. The memory region attribute fields control the memory type, Cacheability, and Shareability of the region.

The following sections describe the assignment of memory region attributes for stage 1 translations:

- Overview of memory region attributes for stage 1 translations.
- Short-descriptor format memory region attributes, without TEX remap.
- Short-descriptor format memory region attributes, with TEX remap.
- VMSAv8-32 Long-descriptor format memory region attributes.

For an implementation that is operating in Secure state, or in Hyp mode, these assignments define the memory attributes of the accessed region.

For an implementation that is operating in a Non-secure PL1 or EL0 mode, the Non-secure PL1&0 stage 2 translation can modify the memory attributes assigned by the stage 1 translation. EL2 control of Non-secure memory region attributes describes these stage 2 assignments.

G5.7.1 Overview of memory region attributes for stage 1 translations

The description of the memory region attributes in a Translation descriptor divides into:

Memory type and attributes

These are described either:

- Directly, by bits in the Translation Table descriptor.
- Indirectly, by registers referenced by bits in the Table descriptor. This is described as *remapping* the memory type and attribute description.

The Short-descriptor translation table format can use either of these approaches, selected by the SCTLR.TRE bit:

TRE == 0 Remap disabled. The TEX[2:0], C, and B bits in the Translation Table descriptor define the memory region attributes. Short-descriptor format memory region attributes, without TEX remap describes this encoding.

Note

With the Short-descriptor format, remapping is called *TEX remap*, and the SCTLR.TRE bit is the *TEX remap enabled* bit.

The description of the TRE == 0 encoding includes information about the encoding in previous versions of the architecture.

TRE == 1 Remap enabled. The TEX[0], C, and B bits in the Translation Table descriptor are index bits to the remap registers, that define the memory region attributes:

- The Primary Region Remap Register, PRRR.
- The Normal Memory Remap Register, NMRR.

Short-descriptor format memory region attributes, with TEX remap describes this encoding scheme.

This scheme reassigns Translation Table descriptor bits TEX[2:1] for use as bits managed by the operating system.

The Long-descriptor translation table format always uses remapping. This means that when the value of TTBCR.EAE is 1, enabling use of the Long-descriptor translation table format, SCTLR.TRE is RES1.

VMSAv8-32 Long-descriptor format memory region attributes describes this encoding.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shareability In the Short-descriptor translation table format, the S bit in the Translation Table descriptor is used in determining the Shareability of the region. How the S bit is interpreted depends on whether TEX remap is enabled, see:

- Shareability and the S bit, without TEX remap.
- Determining the Shareability, with TEX remap.

In the Long-descriptor translation table format, the SH[1:0] field in the Translation Table descriptor encodes the Shareability of the region, see Shareability, Long-descriptor format.

Note

Shareability is one of Non-shareable, Inner Shareable, and Outer Shareable. However, when using the Short-descriptor translation table format without TEX remap, VMSAv8-32 does not support any distinction between Inner Shareable and Outer Shareable memory, and a memory region is either Non-shareable or Outer Shareable.

G5.7.1.1 Stage 1 definition of the XS attribute

When FEAT_XS is implemented, all stage 1 memory types defined in the MAIR0, MAIR1, HMAIR0, HMAIR1, PRRR, and NMRR registers, or the TTBCR or HTCR registers, or in the translation tables, have the XS attribute set to 1, unless they are Inner Write-Back Cacheable, Outer Write-back Cacheable, which have the XS attribute set to 0. This includes any memory types that are treated as Write-Back Cacheable as a result of IMPLEMENTATION DEFINED choices in the architecture.

G5.7.2 Short-descriptor format memory region attributes, without TEX remap

When using the Short-descriptor translation table formats, TEX remap is disabled when the value of SCTLR.TRE is 0.

Note

- The Short-descriptor format scheme without TEX remap is the scheme used in VMSAv6.
- The B (Bufferable), C (Cacheable), and TEX (Type extension) bit names are inherited from earlier versions of the architecture. These names no longer adequately describe the function of the B, C, and TEX bits.

Table G5-12 shows the C, B, and TEX[2:0] encodings when TEX remap is disabled. In the Page Shareability column, an entry of *S bit* indicates that the S bit in the Translation Table descriptor determines the Shareability, see Shareability and the S bit, without TEX remap.

Table G5-12 TEX, C, and B encodings when TRE == 0

TEX[2:0]	C	B	Description	Memory type	Page Shareability
000	0	0	Device-nGnRnE	Device-nGnRnE	Outer Shareable
		1	Device-nGnRE[a]	Device-nGnRE	Outer Shareable[a]
	1	0	Outer and Inner Write-Through, Read-Allocate No Write-Allocate	Normal	S bit
		1	Outer and Inner Write-Back, Read-Allocate No Write-Allocate	Normal	S bit
001	0	0	Outer and Inner Non-cacheable	Normal	Outer Shareable[b]
		1	Reserved	-	-
	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
		1	Outer and Inner Write-Back, Read-Allocate Write-Allocate	Normal	S bit
010	0	0	Device-nGnRE[a]	Device-nGnRE	Outer Shareable[a]
		1	Reserved	-	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TEX[2:0]	C	B	Description		Memory type	Page Shareability
	1	x	Reserved		-	-
011	x	x	Reserved		-	-
1BB	A	A	Cacheable memory:	AA = Inner attribute[c] BB = Outer attribute	Normal	S bit

a. In Armv8, all Device memory types are Outer Shareable. For the Device-nGnRE memory type this is a change from previous versions of the architecture. This is why Device-nGnRE memory has two entries in this table. For compatibility with Armv7 software should use the {TEX, C, B} values {000, 0, 1}.

b. In Armv8, Normal Inner Non-cacheable, Outer Non-cacheable memory is always Outer Shareable. This is a change from previous versions of the architecture, where the S bit determined the Shareability. For compatibility with Armv7 software should set S to 1.

c. For more information, see Cacheability attributes, without TEX remap.

See Memory types and attributes for an explanation of Normal memory, and the types of Device memory, and of the Shareability attribute.

G5.7.2.1 Cacheability attributes, without TEX remap

When the value of TEX[2] is 0, the same Cacheability attribute applies to Inner Cacheable and Outer Cacheable memory regions, and the {TEX[1:0], C, B} values identify this attribute, as Table G5-12 shows.

When the value of TEX[2] is 1, the memory described by the translation table entry is cacheable, and the rest of the encoding defines the Inner Cacheability and Outer Cacheability attributes:

TEX[1:0] Define the Outer Cacheability attribute.

C, B Define the Inner Cacheability attribute.

The translation table entries use the same encoding for the Outer and Inner Cacheability attributes, as Table G5-13 shows.

Table G5-13 Inner and Outer Cacheability attribute encoding

Encoding	Cacheability attribute
00	Non-cacheable
01	Write-Back, Read-Allocate Write-Allocate
10	Write-Through, Read Allocate No Write-Allocate
11	Write-Back, Read Allocate No Write-Allocate

G5.7.2.2 Shareability and the S bit, without TEX remap

The Short-descriptor format translation table entries include an S bit. This bit:

- Is ignored if the entry refers to any type of Device memory, or to Normal memory that is Inner Non-cacheable, Outer Non-cacheable.
- For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, determines whether the memory region is Outer Shareable or Non-shareable:

  S == 0 Normal memory region is Non-shareable.

  S == 1 Normal memory region is Outer Shareable.

Without TEX remapping there is no distinction between Inner Shareable and Outer Shareable memory, meaning the S bit determines whether the region is Non-shareable or Outer Shareable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.7.3 Short-descriptor format memory region attributes, with TEX remap

When using the Short-descriptor translation table formats, TEX remap is enabled when the value of SCTLR.TRE is 1. In this configuration:

- The software that defines the translation tables must program the PRRR and NMRR to define seven possible memory region attributes.
- The TEX[0], C, and B bits of the Translation Table descriptors define the memory region attributes, by indexing PRRR and NMRR.
- Hardware makes no use of TEX[2:1], see The OS managed translation table bits.

When TEX remap is enabled:

- For seven of the eight possible combinations of the TEX[0], C and B bits, fields in the PRRR and NMRR define the region attributes, as described in this section.
- The meaning of the eighth combination for the TEX[0], C and B bits is IMPLEMENTATION DEFINED.
- If the TEX[0], C and B bits determine that the region is a Device memory type, or is Normal Inner Non-cacheable, Outer Non-cacheable, then the region is Outer Shareable. Otherwise, the Shareability is determined by the combination of:
  - The S bit from the Translation Table descriptor.
  - The value of the PRRR.NS0 or PRRR.NS1 bit.
  - The value of the appropriate PRRR.NOS*n* bit, as shown in Table G5-14.

  For more information, see Determining the Shareability, with TEX remap.

For each of the possible encodings of the TEX[0], C, and B bits in a translation table entry, Table G5-14 shows which fields of the PRRR and NMRR registers describe the memory region attributes.

Table G5-14 TEX, C, and B encodings when TRE == 1

Encoding			Memory type[a]	Cache attributes[a, b]:		Outer Shareable attribute[a, c]
TEX[0]	C	B		Inner Cacheability	Outer Cacheability	
0	0	0	PRRR.TR0	NMRR.IR0	NMRR.OR0	NOT(PRRR.NOS0)
		1	PRRR.TR1	NMRR.IR1	NMRR.OR1	NOT(PRRR.NOS1)
	1	0	PRRR.TR2	NMRR.IR2	NMRR.OR2	NOT(PRRR.NOS2)
		1	PRRR.TR3	NMRR.IR3	NMRR.OR3	NOT(PRRR.NOS3)
1	0	0	PRRR.TR4	NMRR.IR4	NMRR.OR4	NOT(PRRR.NOS4)
		1	PRRR.TR5	NMRR.IR5	NMRR.OR5	NOT(PRRR.NOS5)
	1	0	IMPLEMENTATION DEFINED			
		1	PRRR.TR7	NMRR.IR7	NMRR.OR7	NOT(PRRR.NOS7)

a. For details of the *Memory type* and *Outer Shareable* encodings see the description of the PRRR. For details of the *Cache attributes* encodings the description of the NMRR.

b. Applies only if the memory type for the region is mapped as Normal memory.

c. Applies only if both of the following apply:

- The memory type for the region is mapped as Normal memory that is not Inner Non-cacheable and Outer Non-cacheable.
- The region is not Non-shareable.

See Determining the Shareability, with TEX remap.

As Table G5-14 shows, when TEX remap is enabled, for a given set of {TEX[0], C, B} bits from a Translation Table descriptor:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. The primary mapping, to memory type, is given by the PRRR.TR*n* field as shown in the Memory type column.
2. For any region that the PRRR.TR*n* maps as Normal memory, NMRR.IR*n* determines the Inner Cacheability attribute, and NMRR.OR*n* determines the Outer Cacheability attribute.
3. For a region that PRRR.TR*n* maps as Normal memory, if NMRR.{IR*n*, OR*n*} do not map the region as Inner Non-cacheable, Outer Non-cacheable, PRRR.{NS0, NS1} and PRRR.NOS*n* are used to determine the Shareability of the region, see Determining the Shareability, with TEX remap.

The TEX remap registers and the SCTLR.TRE bit are banked between the Secure and Non-secure Security states. For more information, see The effect of EL3 on TEX remap.

The TEX remap registers must be static during normal operation. In particular, when the remap registers are changed:

- It is IMPLEMENTATION DEFINED when the changes take effect.
- It is CONSTRAINED UNPREDICTABLE whether the TLB caches the effect of the TEX remap on translation tables, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

The software sequence to ensure the synchronization of changes to the TEX remap registers is:

1. Execute a `DSB` instruction. This ensures any memory accesses using the old mapping have completed.
2. Write the TEX remap registers or SCTLR.TRE bit.
3. Execute an `ISB` instruction. This ensures synchronization of the register updates.
4. Invalidate the entire TLB.
5. Execute a `DSB` instruction. This ensures completion of the entire TLB operation.
6. Clean and invalidate all caches. This removes any cached information associated with the old mapping.
7. Execute a `DSB` instruction. This ensures completion of the cache maintenance.
8. Execute an `ISB` instruction. This ensures instruction synchronization.

This extends the standard rules for the synchronization of changes to System registers described in Synchronization of changes to AArch32 System registers, and provides implementation freedom as to whether or not the effect of the TEX remap is cached.

G5.7.3.1 Determining the Shareability, with TEX remap

The memory type of a region, as indicated in the Memory type column of Table G5-14, provides the first level of control of the Shareability of the region:

- If the memory is any type of Device memory, then the region is Outer Shareable, and any Shareability attributes in the Translation Table descriptor and PRRR for that region are ignored.

  This applies also to a Normal memory region that the NMRR attributes identify as Inner Non-cacheable and Outer Non-cacheable,
- If using the Short descriptor translation table format then the Shareability of the region is determined using the value of the S bit in the Translation Table descriptor to index one of the PRRR.{NS1. NS0} bits, as described in this section.

Table G5-15 shows how the translation table S bit indexes into the PRRR:

Table G5-15 Determining the Shareability attribute, with TEX remap

Memory type	Remapping when S == 0	Remapping when S == 1
Device or Normal Inner Non-cacheable, Outer Non-cacheable	Outer Shareable	Outer Shareable
Normal, not Inner Non-cacheable, Outer Non-cacheable	PRRR.NS0	PRRR.NS1

For a Normal memory region that is not Inner Non-cacheable, Outer Non-cacheable, the appropriate bit of the PRRR indicates whether the region is Non-shareable, as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PRRR.NS*n* == 0 Non-shareable.

PRRR.{NOS7:NOS0} are ignored.

PRRR.NS*n* == 1 The appropriate PRRR.NOS*m* field, as shown in Table G5-14, indicates whether the region is Inner Shareable or Outer Shareable:

PRRR.NOS*m* == 0 Region is Outer Shareable.

PRRR.NOS*m* == 1 Region is Inner Shareable.

Note

This means that TEX remapping can map a translation table entry with S == 0 as shareable memory.

G5.7.3.2 SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers

When TEX remap is disabled, because the value of the SCTLR.TRE bit is 0:

- The effect of the PRRR and NMRR registers can be IMPLEMENTATION DEFINED.
- The interpretation of the fields of the PRRR and NMRR registers can differ from the description given earlier in this section. One implication of this is that the implementation can provide an IMPLEMENTATION DEFINED mechanism to interpret the PRRR.{NOS7:NOS0} fields.

VMSAv8-32 requires that the effect of these registers is limited to remapping the attributes of memory locations. These registers must not change whether any cache hardware or stages of address translation are enabled. The mechanism by which the TEX remap registers have an effect when the value of the SCTLR.TRE bit is 0 is IMPLEMENTATION DEFINED. The AArch32 architecture requires that from reset, if the IMPLEMENTATION DEFINED mechanism has not been invoked:

- If the PL1&0 stage 1 address translation is enabled and is using the Short-descriptor format translation tables, the architecturally-defined behavior of the TEX[2:0], C, and B bits must apply, without reference to the TEX remap functionality. In other words, memory attribute assignment must comply with the scheme described in Short-descriptor format memory region attributes, without TEX remap.
- If the PL1&0 stage 1 address translation is disabled, then the architecturally-defined behavior of VMSAv8-32 with address translation disabled must apply, without reference to the TEX remap functionality. See The effects of disabling address translation stages on VMSAv8-32 behavior.

Possible mechanisms for enabling the IMPLEMENTATION DEFINED effect of the TEX remap registers when the value of SCTLR.TRE is 0 include:

- A control bit in the ACTLR, or in an IMPLEMENTATION DEFINED System register.
- Changing the behavior when the PRRR and NMRR registers are changed from their IMPLEMENTATION DEFINED reset values.

In addition, if the stage of address translation is disabled and the value of the SCTLR.TRE bit is 1, the architecturally-defined behavior of the VMSAv8-32 with the stage of address translation disabled must apply without reference to the TEX remap functionality.

In an implementation that includes EL3, the IMPLEMENTATION DEFINED effect of these registers must only take effect in the Security state of the registers. See also The effect of EL3 on TEX remap.

G5.7.3.3 The OS managed translation table bits

When TEX remap is enabled, the TEX[2:1] bits in the Translation Table descriptors are available as two bits that can be managed by the operating system. In VMSAv8-32, as long as the SCTLR.TRE bit is set to 1, the values of the TEX[2:1] bits are IGNORED by the PE. Software can write any value to these bits in the translation tables.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.7.3.4 The effect of EL3 on TEX remap

In an implementation that includes EL3, when EL3 is using AArch32, the TEX remap registers are banked between the Secure and Non-secure Security states. When EL3 is using AArch32, write accesses to the Secure register for the current security state apply to all PL1&0 stage 1 translation table lookups in that state. The SCTLR.TRE bit is banked in the Secure and Non-secure copies of the register, and the appropriate version of this bit determines whether TEX remap is applied to translation table lookups in the current security state.

Write accesses to the Secure copies of the TEX remap registers are disabled when the `CP15SDISABLE` input is asserted HIGH, meaning the `MCR` operations to access these registers are UNDEFINED. For more information, see The CP15SDISABLE and CP15SDISABLE2 input signals.

G5.7.4 VMSAv8-32 Long-descriptor format memory region attributes

When a PE is using the VMSAv8-32 Long-descriptor translation table format, the AttrIndx[2:0] field in a block or page Translation Table descriptor for a stage 1 translation indicates the 8-bit field, in the appropriate MAIR register, that specifies the attributes for the corresponding memory region, as follows:

- AttrIndx[2] indicates the MAIR register to be used:
  AttrIndx[2] == 0 Use MAIR0.
  AttrIndx[2] == 1 Use MAIR1.
- AttrIndx[2:0] indicates the required Attr field, Attr*n*, where *n* = AttrIndx[2:0].

Each AttrIndx field defines, for the corresponding memory region:

- The memory type, Normal or a type of Device memory.
- For Normal memory:
  — The Inner Cacheability and Outer Cacheability attributes, each of which is one of Non-cacheable, Write-Through Cacheable, or Write-Back Cacheable.
  — For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is *Allocate* or *No allocate*.

For more information about the AttrIndx[2:0] descriptor field, see Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors.

G5.7.4.1 Shareability, Long-descriptor format

When a PE is using the Long-descriptor translation table format, the SH[1:0] field in a Block or Page Translation Table descriptor specifies the Shareability attributes of the corresponding memory region, if the MAIR entry for that region identifies it as Normal memory that is not both Inner Non-cacheable and Outer Non-cacheable. Table G5-16 shows the encoding of this field.

Table G5-16 SH[1:0] field encoding for Normal memory, Long-descriptor format

SH[1:0]	Normal memory
00	Non-shareable.
01	Reserved, CONSTRAINED UNPREDICTABLE, see Reserved values in System and memory-mapped registers and translation table entries for the permitted behavior.
10	Outer Shareable.
11	Inner Shareable.

See Combining the Shareability attribute for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

For any type of Device memory, and for Normal Inner Non-cacheable, Outer Non-cacheable memory, the value of the SH[1:0] field of the Translation Table descriptor is ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.7.4.2 Other fields in the Long-descriptor translation table format descriptors

The following subsections describe the other fields in the Translation Table Block and Page descriptors when a PE is using the Long-descriptor translation table format:

- Contiguous bit.
- IGNORED fields.
- Field reserved for software use.

G5.7.4.2.1 Contiguous bit

The Long-descriptor Translation Table Format descriptors contain a Contiguous bit. Setting this bit to 1 indicates that 16 adjacent translation table entries point to a *contiguous output address range*. These 16 entries must be aligned in the translation table so that the top five bits of their input addresses, that index their position in the translation table, are the same. For example, to use this bit for a block of 16 entries in the level 3 translation table, bits[20:16] of the input addresses for the 16 entries must be the same.

The contiguous output address range must be aligned to size of 16 translation table entries at the same translation table level.

Use of this bit means that the TLB can cache a single entry to cover the 16 translation table entries.

This bit acts as a hint. The architecture does not require a PE to cache TLB entries in this way. To avoid TLB coherency issues, any TLB maintenance by address must not assume any optimization of the TLB tables that might result from use of this bit.

Note

The use of the contiguous bit is similar to the approach used, in the Short-descriptor translation table format, for optimized caching of Large Pages and Supersections in the TLB. However, an important difference in the contiguous bit capability is that TLB maintenance must be performed based on the size of the underlying translation table entries, to avoid TLB coherency issues. That is, any use of the contiguous bit has no effect on the minimum size of entry that must be invalidated from the TLB.

G5.7.4.2.2 IGNORED fields

In the VMSAv8-32 translation table long-descriptor format, the following fields are defined as IGNORED, meaning the architecture guarantees that a PE makes no use of these fields:

- In the stage 1 and stage 2 Table descriptors, bits[58:52] and bits[11:2].
- In the stage 1 and stage 2 Block and Page descriptors, bit[63] and bits[58:55].
- In the stage 1 and stage 2 Block and Page descriptors in an implementation that does not include FEAT_HPDS2, bits[62:59].

Of these fields:

- In the stage 1 and stage 2 Block and Page descriptors, bits[58:55] are reserved for software use, see Field reserved for software use.
- In the stage 2 Block and Page descriptors:
  - Bit[63] is reserved for use by a System MMU.
  - In an implementation that does not include FEAT_HPDS2, bits[62:59] are reserved for use by a System MMU.

G5.7.4.2.3 Field reserved for software use

The architecture reserves a 4-bit IGNORED field in the Block and Translation Table descriptors, bits[58:55], for software use. In considering migration from using the Short-descriptor format to the Long-descriptor format, this field is an extension of the Short-descriptor field described in The OS managed translation table bits.

Note

The definition of IGNORED means there is no need to invalidate the TLB if these bits are changed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.7.5 EL2 control of Non-secure memory region attributes

Software executing at EL2 controls two sets of translation tables, both of which use the Long-descriptor translation table format. These are:

- The translation tables that control Non-secure EL2 stage 1 translations. These map VAs to PAs, and when EL2 is using AArch32 they are indicated and controlled by the HTTBR and HTCR.

  These translations have exactly the same memory region attribute controls as any other stage 1 translations, as described in VMSAv8-32 Long-descriptor format memory region attributes.
- The translation tables that control Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, and are indicated and when EL2 is using AArch32 they are controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define level 2 memory region attributes, that are combined with the attributes defined in the stage 1 translation. This section describes this combining of attributes.

VMSAv8-32 Long-descriptor Translation Table format descriptors describes the format of the entries in these tables.

Note

In a virtualization implementation, a hypervisor might usefully:

- Reduce the permitted Cacheability of a region.
- Increase the required Shareability of a region.

The combining of attributes from stage 1 and stage 2 translations supports both of these options.

In the stage 2 Translation Table descriptors for memory regions and pages, the MemAttr[3:0] and SH[1:0] fields describe the stage 2 memory region attributes:

- The definition of the stage 2 SH[1:0] field is identical to the same field for a stage 1 translation, see Shareability, Long-descriptor format.
- MemAttr[3:2] give a top-level definition of the memory type, and of the cacheability of a Normal memory region, as Table G5-17 shows:

Table G5-17 Long-descriptor MemAttr[3:2] encoding, stage 2 translation

MemAttr[3:2]	Memory type	Cacheability
00	Device, of type determined by MemAttr[1:0]	Not applicable
01	Normal, Inner Cacheability determined by MemAttr[1:0]	Outer Non-cacheable
10		Outer Write-Through Cacheable
11		Outer Write-Back Cacheable

The encoding of MemAttr[1:0] depends on the Memory type indicated by MemAttr[3:2]:

- When MemAttr[3:2] == `0b00`, indicating a type of Device memory, Table G5-18 shows the encoding of MemAttr[1:0]:

Table G5-18 MemAttr[1:0] encoding for the types of Device memory

MemAttr[1:0]	Meaning when MemAttr[3:2] == `0b00`
00	Region is Device-nGnRnE memory
01	Region is Device-nGnRE memory

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MemAttr[1:0]	Meaning when MemAttr[3:2] == `0b00`
10	Region is Device-nGRE memory
11	Region is Device-GRE memory

- When MemAttr[3:2] != `0b00`, indicating Normal memory, Table G5-19 shows the encoding of MemAttr[1:0]:

Table G5-19 MemAttr[1:0] encoding for Normal memory

MemAttr[1:0]	Meaning when MemAttr[3:2] != `0b00`
00	Reserved, CONSTRAINED UNPREDICTABLE, See Reserved values in System and memory-mapped registers and translation table entries for the permitted behavior.
01	Inner Non-cacheable.
10	Inner Write-Through Cacheable.
11	Inner Write-Back Cacheable.

Note

The stage 2 translation does not assign any allocation hints.

The following sections describe how the memory type attributes assigned at stage 2 of the translation are combined with those assigned at stage 1:

- Combining the memory type attribute.
- Combining the Cacheability attribute.
- Combining the Shareability attribute.

Note

- The following stage 2 translation table attribute settings leave the stage 1 settings unchanged:
  - MemAttr[3:2] == `0b11`, Normal memory, Outer Write-Back Cacheable.
  - MemAttr[1:0] == `0b11`, Inner Write-Back Cacheable.
- In addition to the attribute combinations described in this section, Access permissions for instruction execution describes how the stage 1 and stage 2 execute-never permission fields are combined, so that a region is execute-never if it is defined as execute-never in at least one stage of translation.

G5.7.5.1 Combining the memory type attribute

Table G5-20 shows how the stage 1 and stage 2 memory type assignments are combined:

Table G5-20 Combining the stage 1 and stage 2 memory type assignments

Assignment in stage 1	Assignment in stage 2	Resultant type
Device-nGnRnE	Any	Device-nGnRnE
Device-nGnRE	Device-nGnRnE	Device-nGnRnE
	Not Device-nGnRnE	Device-nGnRE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assignment in stage 1	Assignment in stage 2	Resultant type
Device-nGRE	Device-nGnRnE	Device-nGnRnE
	Device-nGnRE	Device-nGnRE
	Not (Device-nGnRnE or Device-nGnRE)	Device-nGRE
Device-GRE	Device-nGnRnE	Device-nGnRnE
	Device-nGnRE	Device-nGnRE
	Device-nGRE	Device-nGRE
	Device-GRE or Normal	Device-GRE
Normal	Any type of Device	Device type assigned at stage 2
	Normal	Normal

See Combining the Shareability attribute for information about the Shareability of:

- A region for which the resultant type is any Device type.
- A region with a resultant type of Normal for which the resultant cacheability, described in Combining the Cacheability attribute, is Inner Non-cacheable, Outer Non-cacheable.

The combining of the memory type attribute means a translation table walk for a stage 1 translation can be made to a type of Device memory. If this occurs, then:

- If the value of HCR.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be done speculatively.
- If the value of HCR.PTW is 1, then the memory access generates a stage 2 Permission fault.

G5.7.5.2 Combining the Cacheability attribute

For a Normal memory region, Table G5-21 shows how the stage 1 and stage 2 Cacheability assignments are combined. This combination applies, independently, for the Inner Cacheability and Outer Cacheability attributes:

Table G5-21 Combining the stage 1 and stage 2 cacheability assignments

Assignment in stage 1	Assignment in stage 2	Resultant cacheability
Non-cacheable	Any	Non-cacheable
Any	Non-cacheable	Non-cacheable
Write-Through Cacheable	Write-Through or Write-Back Cacheable	Write-Through Cacheable
Write-Through or Write-Back Cacheable	Write-Through Cacheable	Write-Through Cacheable
Write-Back Cacheable	Write-Back Cacheable	Write-Back Cacheable

Note

Only Normal memory has a Cacheability attribute.

G5.7.5.3 Combining the Shareability attribute

In the following cases, a memory region is treated as Outer Shareable, regardless of any shareability assignments at either stage of translation:

- The resultant memory type attribute, described in Combining the memory type attribute, is any type of Device memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The resultant memory type attribute is Normal memory, and the resultant Cacheability, described in Combining the Cacheability attribute, is Inner Non-cacheable Outer Non-cacheable.

For a memory region with a resultant memory type attribute of Normal that is not Inner Non-cacheable Outer Non-cacheable, Table G5-22 shows how the stage 1 and stage 2 shareability assignments are combined:

Table G5-22 Combining the stage 1 and stage 2 Shareability assignments

Assignment in stage 1	Assignment in stage 2	Resultant Shareability
Outer Shareable	Any	Outer Shareable
Inner Shareable	Outer Shareable	Outer Shareable
Inner Shareable	Inner Shareable	Inner Shareable
Inner Shareable	Non-shareable	Inner Shareable
Non-shareable	Outer Shareable	Outer Shareable
Non-shareable	Inner Shareable	Inner Shareable
Non-shareable	Non-shareable	Non-shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.8 Translation Lookaside Buffers

Translation Lookaside Buffers (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement to perform a translation table walk in memory for every memory access. The Arm architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.
- The architecture does not guarantee that an unlocked TLB entry remains in the TLB.
- The architecture guarantees that a locked TLB entry remains in the TLB. However, a locked TLB entry might be updated by subsequent updates to the translation tables. Therefore, when a change is made to the translation tables, the architecture does not guarantee that a locked TLB entry remains incoherent with an entry in the translation table.
- The architecture guarantees that a translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault is not held in the TLB. However a translation table entry that generates a Domain fault or a Permission fault might be held in the TLB.
- When address translation is enabled, any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault and is not from a translation regime for an Exception level that is lower than the current Exception level can be allocated to a TLB at any time. The only translation table entries guaranteed not to be held in the TLB are those that generate a Translation fault, an Address size fault, or an Access flag fault.

  Note

  A TLB can hold translation table entries that do not generate a Translation fault but point to subsequent tables in the translation table walk. This can be referred to as *intermediate caching* of TLB entries.

- Software can rely on the fact that between disabling and re-enabling a stage of address translation, entries in the TLB relating to that stage of translation have not been corrupted to give incorrect translations.

The following sections give more information about TLB implementation:

- Global and process-specific translation table entries.
- TLB matching.
- TLB behavior at reset.
- TLB lockdown.
- TLB conflict aborts.

See also TLB maintenance requirements.

Note

In addition to the functions described in this section, the TLB might cache information from control registers that are described as being “permitted to be cached in a TLB”, even when any or all of the stages of translation are disabled. This caching of information gives rise to the maintenance requirements described in General TLB maintenance requirements

G5.8.1 Global and process-specific translation table entries

For VMSAv8-32, system software can divide a virtual memory map used by memory accesses at PL1 and EL0 into global and non-global regions, indicated by the nG bit in the Translation Table descriptors:

nG == 0 The translation is global, meaning the region is available for all processes.

nG == 1 The translation is non-global, or process-specific, meaning it relates to the current ASID, as defined by:

- TTBR0.ASID or TTBR1.ASID, if using the Long-descriptor translation table format. In this case, TTBCR.A1 selects which ASID is current.
- CONTEXTIDR.ASID, if using the Short-descriptor translation table format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Each non-global region has an associated ASID. These identifiers mean different translation table mappings can co-exist in a caching structure such as a TLB. This means that software can create a new mapping of a non-global memory region without removing previous mappings.

For a symmetric multiprocessor cluster where a single operating system is running on the set of PEs, the architecture requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all PEs in the system.

In AArch32 state, the translation regime used for accesses made at EL2 never supports ASIDs, and all pages are treated as global.

When a PE is using the Long-descriptor translation table format, and is in Secure state, a translation must be treated as non-global, regardless of the value of the nG bit, if NSTable is set to 1 at any level of the translation table walk.

Translation table entries from lookup levels other than the final lookup level are treated as non-global, regardless of the value of the nG bit in the final lookup level.

For more information, see Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format.

G5.8.2 TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. Use of the TLB maintenance instructions described in TLB maintenance requirements can prevent any TLB incoherency becoming a problem.

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID and VMID are changed without suitable invalidation of the TLB. This can occur only if the architecturally-required break-before-make sequence described in Using break-before-make when updating translation table entries is not used. If the break-before-make sequence is not used, the TLB can hold two mappings for the same address, and this:

- Might generate an exception that is reported using the TLB Conflict fault code, see TLB conflict aborts.
- Might lead to CONSTRAINED UNPREDICTABLE behavior. In this case, behavior will be consistent with one of the mappings held in the TLB, or with some amalgamation of the values held in the TLB, but cannot give access to regions of memory with permissions or attributes that could not be assigned by valid translation table entries in the translation regime being used for the access. See CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

G5.8.3 TLB behavior at reset

The Arm architecture does not require a reset to invalidate the TLBs, and recognizes that an implementation might require caches, including TLBs, to maintain context over a system reset. Possible reasons for doing so include power management and debug requirements.

Therefore:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- All TLBs are disabled from reset. All stages of address translation that are used from the PE state entered on coming out of reset are disabled from reset, and the contents of the TLBs have no effect on address translation. For more information, see Enabling stages of address translation.
- An implementation can require the use of a specific TLB invalidation routine, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

  Arm recommends that if an invalidation routine is required for this purpose, and the PE resets into AArch32 state, the routine is based on the AArch32 TLB maintenance instructions described in The scope of TLB maintenance instructions.

Similar rules apply:

- To cache behavior, see Behavior of caches at reset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- To branch predictor behavior, see Behavior of the branch predictors at reset.

G5.8.4 TLB lockdown

The Arm architecture recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

- The architecture does not require TLB lockdown support.
- If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.

This means that:

- The TLB Type Register, TLBTR, does not define the lockdown scheme in use.
- In AArch32 state, a region of the {`coproc` == `0b1111`, `CRn` == `c10`} encodings is reserved for IMPLEMENTATION DEFINED TLB functions, such as TLB lockdown functions. The reserved encodings are those with:
  - `<CRm>` == {c0, c1, c4, c8}.
  - All values of `<opc2>` and `<opc1>`.

An implementation might use some of the {`coproc` == `0b1111`, `CRn` == `c10`} encodings that are reserved for IMPLEMENTATION DEFINED TLB functions to implement additional TLB control functions. These functions might include:

- Unlock all locked TLB entries.
- Prefetch into a specific level of TLB. This is beyond the scope of the `PLI` and `PLD` hint instructions.

The inclusion of EL2 in an implementation does not affect the TLB lockdown requirements. However, in an implementation that includes EL2, exceptions generated as a result of TLB lockdown when executing in a Non-secure PL1 mode or in Non-secure User mode can be routed to either:

- Non-secure Abort mode, using the Non-secure Data Abort exception vector.
- Hyp mode, using the Hyp Trap exception vector.

For more information, see HCR.

G5.8.5 TLB conflict aborts

If an address matches multiple entries in the TLB, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

An implementation can generate TLB conflict aborts on either or both instruction fetches and data accesses. A TLB conflict abort is classified as an MMU fault, see Types of MMU faults. This means:

- A TLB conflict abort on an instruction fetch is reported as a Prefetch Abort exception,
- A TLB conflict abort on a data access is reported as a Data Abort exception,

Fault status codes for TLB conflict aborts are defined for both the Short-descriptor and Long-descriptor translation table formats, see:

- PL1 fault reporting with the Short-descriptor translation table format
- PL1 fault reporting with the Long-descriptor translation table format.

On a TLB conflict abort, the fault address register returns the address that generated the fault. That is, it returns the address that was being looked up in the TLB.

It is IMPLEMENTATION DEFINED whether a TLB conflict abort is a stage 1 abort or a stage 2 abort.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- An address can hit multiple entries in the TLB if the TLB has been invalidated inappropriately, for example if TLB invalidation required by this manual has not been performed.
- A stage 2 abort cannot be generated if the Non-secure PL1&0 stage 2 address translation is disabled.

The priority of the TLB conflict abort is IMPLEMENTATION DEFINED, because it depends on the form of any TLB that can generate the abort. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.

If an address matches multiple entries in the TLB and no TLB conflict abort not generated, the resulting behavior is CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values. The CONSTRAINED UNPREDICTABLE behavior must not permit access to regions of memory with permissions or attributes that mean they cannot be accessed in the current Security state at the current Privilege level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.9 TLB maintenance requirements

Translation Lookaside Buffers describes the Arm architectural provision for TLBs. Although the Arm architecture does not specify the form of any TLB structures, it does define the mechanisms by which TLBs can be maintained. The following sections describe the VMSAv8-32 TLB maintenance instructions:

- General TLB maintenance requirements.
- Maintenance requirements on changing System register values.
- Atomicity of register changes on changing virtual machine.
- Synchronization of changes of ASID and TTBR.
- The scope of TLB maintenance instructions.

G5.9.1 General TLB maintenance requirements

TLB maintenance instructions provide a mechanism to invalidate entries from a TLB. As Translation Lookaside Buffers describes, when address translation is enabled translation table entries can be allocated to a TLB at any time. This means that software must perform TLB maintenance between updating translation table entries that apply in a particular context and accessing memory locations whose translation is determined by those entries in that context.

Note

This requirement applies to any translation table entry at any level of the translation tables, including an entry that points to further levels of the tables, provided that the entry in that level of the tables does not cause a Translation fault, an Address size fault, or an Access flag fault.

The effects of certain System register control fields are permitted to be cached in a TLB. For more information, see TLB maintenance.

Note

For AArch32, use of a single DTLBI or ITLBI instruction when invalidating TLB entries is insufficient, as both data and instruction TLBs must be invalidated. Arm provides a set of TLBI instructions that are intended for this purpose.

In addition to any TLB maintenance requirement, when changing the cacheability attributes of an area of memory, software must ensure that any cached copies of affected locations are removed from the caches. For more information, see Cache maintenance requirement created by changing translation table attributes.

Because a TLB never holds any translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault, a change from a translation table entry that causes a Translation, Address size, or Access flag fault to one that does not fault, does not require any TLB or branch predictor invalidation. However, a Context Synchronization event is required to ensure that instruction fetches are affected by a completed change to translation table entries that, before the change, generated a Translation, Address size, or Access flag fault.

Special considerations apply to translation table updates that change the memory type, cacheability, or output address of an entry, see Using break-before-make when updating translation table entries.

In addition, software must perform TLB maintenance after updating the System registers if the update means that the TLB might hold information that applies to a current translation context, but is no longer valid for that context. Maintenance requirements on changing System register values gives more information about this maintenance requirement.

Each of the translation regimes defined in Figure G5-1 is a different context, and:

- For the Non-secure PL1&0 regime, a change in the VMID or ASID value changes the context.
- For the Secure PL1&0 regime, a change in the ASID value changes the context.

For operation in Non-secure PL1 or EL0 modes, a change of HCR.VM, unless made at the same time as a change of VMID, requires the invalidation of all TLB entries for the Non-secure PL1&0 translation regime that apply to the current VMID. Otherwise, there is no guarantee that the effect of the change of HCR.VM is visible to software executing in the Non-secure PL1 and EL0 modes.

Any TLB maintenance instruction can affect any other TLB entries that are not locked down.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch32 state defines {`coproc` == `0b1111`, `CRn` == `c8`} System instructions for TLB maintenance instructions, and supports the following operations:

- Invalidate all unlocked entries in the TLB.
- Invalidate a single TLB entry, by VA, or VA and ASID for a non-global entry.
- Invalidate all TLB entries that match a specified ASID.
- Invalidate all TLB entries that match a specified VA, regardless of the ASID.
- Operations that apply across multiprocessors in the same Inner Shareable domain.

Note

An address-based TLB maintenance instruction that applies to the Inner Shareable domain does so regardless of the Shareability attributes of the address supplied as an argument to the instruction.

A TLB maintenance instruction that specifies a VA that would generate any MMU fault, including a VA that is not in the range of VAs that can be translated, does not generate an abort.

EL2 provides additional TLB maintenance instructions for use in AArch32 state at EL2, and has some implications for the effect of the other TLB maintenance instructions, see The scope of TLB maintenance instructions.

In an implementation that includes EL3, the TLB maintenance instructions take account of the current Security state, as part of the address translation required for the TLB maintenance instruction.

Some TLB maintenance instructions are defined as operating only on instruction TLBs, or only on data TLBs. AArch32 state includes these instructions for backward compatibility. However, more recent TLB maintenance instructions do not support this distinction. Arm deprecates any use of Instruction TLB maintenance instructions, or of Data TLB maintenance instructions, and developers must not rely on this distinction being maintained in future revisions of the Arm architecture.

The Arm architecture does not dictate the form in which the TLB stores translation table entries. However, for TLB invalidate instructions, the minimum size of the table entry that is invalidated from the TLB must be at least the size that appears in the translation table entry.

The scope of TLB maintenance instructions describes the TLB maintenance instructions. The following subsections give more information about the general requirements for TLB maintenance:

- Using break-before-make when updating translation table entries.
- The interaction of TLB lockdown with TLB maintenance instructions.
- Ordering and completion of TLB maintenance instructions.
- Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

G5.9.1.1 Using break-before-make when updating translation table entries

To avoid possibly creating multiple TLB entries for the same address, and to avoid the effects of TLB caching possibly breaking coherency, single-copy atomicity properties, ordering guarantees or uniprocessor semantics, or possibly failing to clear the Exclusives monitors, the architecture requires the use of a break-before-make sequence when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation table entries involves any of:

- A change of the memory type, including shareability.
- A change of the cacheability attributes.
- A change of the *output address* (OA), if the OA of at least one of the old translation table entries and the new translation table entry is writable.
- A change to the size of block used by the translation system. This applies both:
  - When changing from a smaller size to a larger size, for example by replacing a table mapping with a block mapping in a stage 2 translation table.
  - When changing from a larger size to a smaller size, for example by replacing a block mapping with a table mapping in a stage 2 translation table.
- A change of the output address (OA), if the contents of memory at the new OA do not match the contents of memory at the previous OA.
- Creating a global entry when there might be non-global entries in a TLB that overlap with that global entry.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Changes to the output address (OA) include changing between Secure and Non-secure output addresses.

A break-before-make sequence on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a `DSB` instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a `DSB` instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a `DSB` instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution, and therefore the problems described at the start of this subsection cannot arise.

G5.9.1.2 The interaction of TLB lockdown with TLB maintenance instructions

The precise interaction of TLB lockdown with the TLB maintenance instructions is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance instructions must comply with these rules:

- The effect on locked entry of a TLB invalidate all unlocked entries instruction or a TLB invalidate by VA all ASID instruction that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  - The instructions have no effect on entries that are locked down.
  - The instructions generate an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the {`coproc` == `0b1111`, `CRn` == `c5`} fault status register definitions include a Fault status code for cache and TLB lockdown faults, see Table G5-26 for the codes used with the Short-descriptor translation table formats, or Table G5-27 for the codes used with the Long-descriptor translation table formats.
    In an implementation that includes EL2, if EL2 is using AArch32 and the value of HCR.TIDCP is 1, any such exceptions taken from a Non-secure PL1 mode are routed to Hyp mode, see HCR.

  This permits a usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.
- The effect on a locked TLB entry of a TLB invalidate by VA instruction or a TLB invalidate by ASID match instruction that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  - A locked entry is invalidated in the TLB.
  - The instruction has no effect on a locked entry in the TLB. In the case of the Invalidate single entry by VA, this means the PE treats the instruction as a NOP.
  - The instruction generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the {`coproc` == `0b1111`, `CRn` == `c5`} fault status register definitions include a Fault status code for cache and TLB lockdown faults, see Table G5-26 and Table G5-27.

Note

Any implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required invalidation on entries that are not locked down.
- Implement one of the other specified alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use the architecturally-defined maintenance instructions. This minimizes the number of customized maintenance operations required.
In addition, an implementation that uses an abort mechanism for handling TLB maintenance instructions on entries that can be locked down but are not actually locked down must also provide a mechanism that ensures that no TLB entries are locked.
Similar rules apply to cache lockdown, see The interaction of cache lockdown with cache maintenance instructions.

The architecture does not guarantee that any unlocked entry in the TLB remains in the TLB. This means that, as a side-effect of a TLB maintenance instruction, any unlocked entry in the TLB might be invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.9.1.3 Ordering and completion of TLB maintenance instructions

The following rules describe the relations between the memory order model and the TLB maintenance instructions:

- A TLB maintenance instruction executed by a PE, PEe, causes a TLB maintenance operation to be generated on each PE within the shareability domain of PEe that is specified by the instruction.
  - At EL2 or EL3, or at EL1 when the Effective value of HCRX_EL2.FnXS is 0, the associated TLB maintenance operations do not have the nXS qualifier.
  - At EL1, when the Effective value of HCRX_EL2.FnXS is 1, the behavior of the associated TLB maintenance operations is the same as described for the AArch64 TLB maintenance instructions with the nXS qualifier. See Ordering and completion of TLB maintenance instructions.

  Note

  When FEAT_XS is not implemented, all TLB maintenance instructions do not have the nXS qualifier and the Effective value of HCRX_EL2 is 0.

- A TLB maintenance operation generated by a TLB maintenance instruction is finished for a PE when:
  - All memory accesses generated by that PE using in-scope old translation information are complete.
  - All memory accesses RWx generated by that PE are complete.

  RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I_1 executed by that PE where all of the following apply:
  - I_1 uses the in-scope old translation information.
  - The use of the in-scope old translation information generates a synchronous Data Abort.
  - If I_1 did not generate an abort from use of the in-scope old translation information, I_1 would generate a memory access that RWx would be locally-ordered-before.

  In-scope old translation information is any translation information, for addresses that are in the scope of the TLB maintenance instruction, that is not consistent with either:
  - The architectural translation information held in the translation tables at the time that the TLB maintenance instruction is executed by PEe.
  - Any architecture translation information that is Coherence-after the information held in the translation tables at the time that the TLB maintenance instruction is executed by PEe.

  Note

  - Old translation information of this type might be held in TLBs or other non-coherent caching structures.

  A TLB maintenance instruction is complete when the TLB maintenance operations specified by the TLB maintenance instruction are finished for all PEs.

  After the TLB maintenance instruction is complete, no new memory accesses using the in-scope old translation information will be architecturally performed by any observer that is affected by the TLB maintenance instruction.

  Note

  Speculative memory accesses can be performed using those entries if it is impossible for software running on any observer to tell that those memory accesses have been performed.

- A TLB maintenance instruction is only guaranteed to be complete after the execution of a `DSB` instruction.
- An `ISB` instruction, or a return from an exception, causes the effect of all completed TLB maintenance instructions that appear in program order before the `ISB` or return from exception to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- An exception causes all completed TLB maintenance instructions, that appear in the instruction stream before the point where the exception is taken, to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- All TLB maintenance instructions are executed in program order relative to each other.
- The execution of a Data or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent explicit memory read or write effect instruction after both:
  - The execution of a `DSB` instruction to ensure the completion of the TLB maintenance instruction.
  - Execution of a subsequent Context Synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The execution of an Instruction or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent instruction fetch after both:
  - The execution of a DSB instruction to ensure the completion of the TLB maintenance instruction.
  - Execution of a subsequent Context Synchronization event.

A DSB instruction can be used to ensure the completion of TLB maintenance instructions. For more information, see Data Synchronization Barrier.

The following rules apply when writing translation table entries. They ensure that the updated entries are visible to subsequent accesses and cache maintenance instructions.

For TLB maintenance, the translation table walk is treated as a separate observer. This means:

- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by an explicit memory read or write effect after the execution of both a DSB and an ISB.

  However, the architecture guarantees that any writes to the translation tables are not seen by any explicit memory effect that occurs in program order before the write to the translation tables.
- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by the instruction fetch of an instruction that follows the write to the translation tables after both a DSB and an ISB.

Therefore, in a uniprocessor system, an example instruction sequence for writing a translation table entry, covering changes to the instruction or data mappings is:

```
STR rx, [Translation table entry]          ; write new entry to the translation table
DSB            ; ensures visibility of the new entry
Invalidate TLB entry by VA (and ASID if non-global) [page address]
Invalidate BTC
DSB            ; ensure completion of the Invalidate TLB instruction
ISB            ; ensure table changes visible to instruction fetch
```

G5.9.1.4 Use of ASIDs and VMIDs to reduce TLB maintenance requirements

To reduce the need for TLB maintenance on context switches, the lookups from some translation regimes can be associated with an ASID, or with an ASID and a VMID.

Note

The use of ASIDs and VMIDs in VMSAv8-32 is generally similar to their use in VMSAv8-64, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

For more information about the use of ASIDs in VMSAv8-32 see Global and process-specific translation table entries.

G5.9.1.4.1 Common not private translations in VMSAv8-32

In an implementation that includes FEAT_TTCNP, multiple PEs in the same Inner Shareable domain can use the same translation table entries for a given stage of address translation in a particular translation regime. This sharing is enabled by the TTBR.CnP field for the stage of address translation.

When the value of a TTBR.CnP field is 1, translation table entries pointed to by that TTBR are shared with all other PEs in the Inner Shareable domain for which the following conditions are met:

- The corresponding TTBR.CnP field has the value 1.
- That TTBR is using the Long-descriptor translation table format.
- If an ASID applies to the stage of translation corresponding to that TTBR then the current ASID value must be the same for all of the PEs that are sharing entries for any translation table entry that is not global or not leaf level.
- If a VMID applies to the stage of translation corresponding to that TTBR then the current VMID value must be the same for all of the PEs that are sharing entries.

Note

In an implementation that includes EL3, the Secure instances of TTBR0 and TTBR1 relate to the Secure PL1&0 translation regime, and the Non-secure instances of TTBR0 and TTBR1 relate to the Non-secure PL1&0 translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a translation regime with both stage 1 and stage 2 translations, where a TLB combines information from stage 1 and stage 2 translation table entries into a single entry, this entry can be shared between different PEs only if the value of the TTBR.CnP bit is 1 for both stage 1 and stage 2 of the translation table walk.

The TTBR.CnP bit can be cached in a TLB.

For a given TTBR, if the value of TTBR.CnP is 1 on multiple PEs in the same Inner Shareable domain, and those PEs meet the other conditions for sharing translation table entries as defined in this section, but those TTBRs do not point to the same translation table entries, then the system is misconfigured, and performing an address translation using that TTBR:

- Might generate multiple hits in the TLB, and as a result generate an exception that is reported using the TLB conflict fault code, see TLB conflict aborts.
- Otherwise, has a CONSTRAINED UNPREDICTABLE result, as described in CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

G5.9.2 Maintenance requirements on changing System register values

The TLB contents can be influenced by control bits in a number of System registers. This means the TLB entries associated with a translation regime affected by these control bits must be invalidated after any changes to these bits, unless the changes are accompanied by a change to the VMID or ASID, if appropriate depending on the translation regime, that defines the context to which the bits apply. The general form of the required invalidation sequence is as follows:

```
; Change control bits in System registers
ISB        ; Synchronize changes to the control bits
; Perform TLB invalidation of all entries that might be affected by the changed control bits
```

The System register changes that this applies to are:

- Any change to the NMRR, PRRR, MAIR0, MAIR1, HMAIR0 or HMAIR1 registers.
- Any change to the SCTLR.AFE bit, see Changing the Access flag enable.
- Any change to any of the SCTLR.{TRE, WXN, UWXN} bits.
- Any change to the Translation table base 0 address in TTBR0.
- Any change to the Translation table base 1 address in TTBR1.
- Any change to HTTBR.BADDR.
- Any change to VTTBR.BADDR.
- Changing TTBCR.EAE, see Changing the current Translation table format.
- In an implementation that includes EL3, any change to the SCR.SIF bit.
- In an implementation that includes EL2:
  - Any change to the HCR.VM bit.
  - Any change to HCR.PTW bit, see Changing HCR.PTW.
- When using the Short-descriptor translation table format:
  - Any change to the RGN, IRGN, S, or NOS fields in TTBR0 or TTBR1.
  - Any change to the N, EAE, PD0 or PD1 fields in TTBCR.
- When using the Long-descriptor translation table format:
  - Any change to the EAE, T*n*SZ, ORGN*n*, IRGN*n*, SH*n*, or EPD*n* fields in the TTBCR, where *n* is 0 or 1.
  - Any change to the TTBCR2.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the HTCR.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the VTCR.

G5.9.2.1 Changing the Access flag enable

In a PE that is using the Short-descriptor translation table format, it is CONSTRAINED UNPREDICTABLE whether the TLB caches the effect of the SCTLR.AFE bit on translation tables. This means that, after changing the SCTLR.AFE bit software must invalidate the TLB before it relies on the effect of the new value of the SCTLR.AFE bit, otherwise behavior is CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

There is no enable bit for use of the Access flag when using the Long-descriptor translation table format.

G5.9.2.2 Changing HCR.PTW

When EL2 is using AArch32 and the value of the Protected table walk bit, HCR.PTW, is 1, a stage 1 translation table access in the Non-secure PL1&0 translation regime, to an address that is mapped to any type of Device memory by its stage 2 translation, generates a stage 2 Permission fault. A TLB associated with a particular VMID might hold entries that depend on the effect of HCR.PTW. Therefore, if the value of HCR.PTW is changed without a change to the VMID value, all TLB entries associated with the current VMID must be invalidated before executing software in a Non-secure PL1 or EL0 mode. If this is not done, behavior is CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

G5.9.2.3 Changing the current Translation table format

The effect of changing TTBCR.EAE when executing in the translation regime affected by TTBCR.EAE with any stage of address translation for that translation regime enabled is CONSTRAINED UNPREDICTABLE. This means that, when TTBCR.EAE is changed for a given context, the TLB must be invalidated before resuming execution in that context, otherwise the effect is CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.

G5.9.3 Atomicity of register changes on changing virtual machine

From the viewpoint of software executing in a Non-secure PL1 or EL0 mode, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for:

- Non-secure PL1&0 stage 1 address translations. This means that all of the following registers must change atomically:
  - PRRR and NMRR, if using the Short-descriptor translation table format.
  - MAIR0 and MAIR1, if using the Long-descriptor translation table format.
  - TTBR0, TTBR1, TTBCR, TTBCR2, DACR, and CONTEXTIDR.
  - The SCTLR.
- Non-secure PL1&0 stage 2 address translations. When EL2 is using AArch32, this means that all of the following registers and register fields must change atomically:
  - VTTBR and VTCR.
  - HMAIR0 and HMAIR1.
  - The HSCTLR.

Note

Only some bits of SCTLR affect the stage 1 translation, and only some bits of HSCTLR affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution in Non-secure PL1&0 modes. However, when updated as part of a switch of virtual machines they are updated by software executing in Hyp mode. This means the registers are *out of context* when they are updated, and no synchronization precautions are required.

Note

By contrast, a translation table change associated with a change of ASID, made by software executing at PL1, can require changes to registers that are *in context*. Synchronization of changes of ASID and TTBR describes appropriate precautions for such a change.

Software executing in Hyp mode, or in Secure state, must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.9.4 Synchronization of changes of ASID and TTBR

A common virtual memory management requirement is to change the ASID and TTBR together to associate the new ASID with different translation tables, without any change to the current translation regime. When using the Short-descriptor translation table format, different registers hold the ASID and the translation table base address, meaning these two values cannot be updated atomically. Since a PE can perform a speculative memory access at any time, this lack of atomicity is a problem that software must address. Such a change is complicated by:

- The depth of speculative fetch being IMPLEMENTATION DEFINED.
- The use of branch prediction.

When using the Short-descriptor translation table format, the virtual memory management operations must ensure the synchronization of changes of the ContextID and the translation table registers. For example, some or all of the TLBs, branch predictors, and other caching of ASID and translation information might become corrupt with invalid translations. Synchronization is required to avoid either:

- The old ASID being associated with translation table walks from the new translation tables.
- The new ASID being associated with translation table walks from the old translation tables.

There are a number of possible solutions to this problem, and the most appropriate approach depends on the system. Example G5-3, Example G5-4, and Example G5-5 describe three possible approaches.

Note

Another instance of the synchronization problem occurs if a branch is encountered between changing the ASID and performing the synchronization. In this case the value in the branch predictor might be associated with the incorrect ASID. Software can address this possibility using any of these approaches, but instead software might be written in a way that avoids such branches.

Example G5-3 Using a reserved ASID to synchronize ASID and TTBR changes

In this approach, a particular ASID value is reserved for use by the operating system, and is used only for the synchronization of the ASID and TTBR. This example uses the value of 0 for this purpose, but any value could be used.

This approach can be used only when the size of the mapping for any given VA is the same in the old and new translation tables.

The maintenance software uses the following sequence, which must be executed from memory marked as global:

```
Change ASID to 0
ISB
Change TTBR
ISB
Change ASID to new value
```

This approach ensures that any non-global pages fetched at a time when it is uncertain whether the old or new translation tables are being accessed are associated with the unused ASID value of 0. Since the ASID value of 0 is not used for any normal operations these entries cannot cause corruption of execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G5-4 Using translation tables containing only global mappings when changing the ASID

A second approach involves switching the translation tables to a set of translation tables that only contain global mappings while switching the ASID.

The maintenance software uses the following sequence, which must be executed from memory marked as global:

```
Change TTBR to the global-only mappings
ISB
Change ASID to new value
ISB
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

This approach works without the need for TLB invalidations in systems that have caching of intermediate levels of translation tables, as described in General TLB maintenance requirements, provided that the translation tables containing only global mappings have only level 1 translation table entries of the following kinds:

- Entries that are global.
- Pointers to level 2 tables that hold only global entries, and that are the same level 2 tables that are used for accessing global entries by both:
  - The set of translation tables that were used under the old ASID value.
  - The set of translation tables that will be used with the new ASID value.
- Invalid level 1 entries.

In addition, all sets of translation tables in this example should have the same Shareability and Cacheability attributes, as held in the TTBR0.{ORGN, IRGN} or TTBR1.{ORGN, IRGN} fields.

If these rules are not followed, then the implementation might cache level 1 translation table entries that require explicit invalidation.

Example G5-5 Disabling non-global mappings when changing the ASID

In systems where only the translation tables indexed by TTBR0 hold non-global mappings, maintenance software can use the TTBCR.PD0 field to disable use of TTBR0 during the change of ASID. This means the system does not require a set of global-only mappings.

The maintenance software uses the following sequence, which must be executed from a memory region with a translation that is accessed using the base address in the TTBR1 register, and is marked as global:

```
Set TTBCR.PD0 = 1
ISB
Change ASID to new value
Change TTBR to new value
ISB
Set TTBCR.PD0 = 0
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

When using the Long-descriptor translation table format, TTBCR.A1 holds the number, 0 or 1, of the TTBR that holds the current ASID. This means the current TTBR can also hold the current ASID, and the current translation table base address and ASID can be updated atomically when:

- TTBR0 is the only TTBR being used. TTBCR.A1 must be set to 0.
- TTBR0 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 0.
- TTBR1 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In these cases, software can update the current translation table base address and ASID atomically, by updating the appropriate TTBR, and does not require a specific routine to ensure synchronization of the change of ASID and base address.

However, in all other cases using the Long-descriptor format, the synchronization requirements are identical to those when using the Short-descriptor formats, and the examples in this section indicate how synchronization might be achieved.

Note

When using the Long-descriptor translation table format, CONTEXTIDR.ASID has no significance for address translation, and is only an extension of the Context ID value.

G5.9.5 The scope of TLB maintenance instructions

TLB maintenance instructions provide a mechanism for invalidating entries from TLB caching structures, to ensure that changes to the translation tables are reflected correctly in the TLB caching structures. To support TLB maintenance in multiprocessor systems, there are maintenance operations that apply to the TLBs of all PEs in the same Inner Shareable domain.

The architecture permits the caching of any translation table entry that has been returned from memory without a fault and that does not, itself, cause a Translation Fault, an Address size fault, or an Access Flag fault. This means the TLB:

- Cannot hold an entry that, when used for a translation table lookup, causes a Translation fault, an Address size fault, or an Access Flag fault.
- Can hold an entry for a translation table lookup for a translation that causes a Translation Fault, an Address size fault, or an Access Flag fault at a subsequent level of translation table lookup. For example, it can hold an entry for the level 1 lookup of a translation that causes a Translation fault, an Address size fault, or an Access Flag fault at level 2 or level 3 of lookup.

This means that entries cached in the TLB can include:

- Translation table entries that point to a subsequent table to be used in the current stage of translation.
- In an implementation that includes EL2:
  - Stage 2 translation table entries that are used as part of a stage 1 translation table walk.
  - Stage 2 translation table entries for translating the output address of a stage 1 translation.

Such entries might be held in intermediate TLB caching structures that are used during a translation table walk and that are distinct from the data caches in that they are not required to be invalidated as the result of writes of the data. The architecture makes no restriction on the form of these intermediate TLB caching structures when these caches are indexed by their input address. The architecture does not restrict having either:

- Translation table entry caching that is indexed by the physical address of the location holding the translation table entry.
- Translation table entry caching that is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry. However, FEAT_nTLBPA allows software discoverability of whether such caches exist, such that if FEAT_nTLBPA is implemented, such caching is not implemented.

If all of the following are true, a TLB maintenance instruction will ensure that any physical address or intermediate physical address indexed cached copies of translation table entries are invalidated for a PE:

- The TLB maintenance instruction applies to that PE with the context information that is relevant to translation table entry caching that is either:
  - Indexed by the physical address of the location holding the translation table entry.
  - Stage 1 translation information that is indexed by the intermediate physical address of the location holding the translation table entry.
- FEAT_nTLBPA is not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Any TLB caching based on the physical address or intermediate physical address obeys the other rules regarding the caching to TLB entries described in this manner, including restrictions on types of entries that cannot be held in a TLB, and a requirement that entries held in a TLB are distinguished by context information such as translation regime, VMID, and ASID.

The architecture does not intend to restrict the form of TLB caching structures used for holding translation table entries. In particular for translation regimes that involve two stages of translation, it recognizes that such caching structures might contain:

- At any level of the translation table walk, entries containing information from stage 1 translation table entries.
- In an implementation that includes EL2:
  - At any level of the translation table walk, entries containing information from stage 2 translation table entries.
  - At any level of the translation table walk, entries combining information from both stage 1 and stage 2 translation table entries.

Note

For the purpose of TLB maintenance, the term *TLB entry* denotes any structure, including temporary working registers in translation table walk hardware, that holds a translation table entry.

For the TLB maintenance instructions:

- If a TLB maintenance instruction is required to apply to stage 1 entries then it must apply to any cached entry in the caching structures that includes any stage 1 information that would be used to translate the address being invalidated, including any entry that combines information from both stage 1 and stage 2 translation table entries.

  Note

  - Where stage 1 information has been cached in multiple TLB entries, as could occur from splintering a page when caching in the TLB, then the invalidation must apply to each cached entry containing stage 1 information from the page that is used to translate the address being invalidated, regardless of whether or not that cached entry would be used to translate the address being invalidated.
  - As stated in Global and process-specific translation table entries, translation table entries from levels of translation other than the final level are treated as being non-global. Arm expects that, in at least some implementations, cached copies of levels of the translation table walk other than the last level are tagged with their ASID, regardless of whether the final level is global. This means that TLB invalidations that involve the ASID require the ASID to match such entries to perform the required invalidation.

- If a TLB maintenance instruction is required to apply to stage 2 entries only, then:
  - It is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
  - It must apply to caching structures that contain information only from stage 2 translation table entries.
- If a TLB maintenance instruction is required to apply to both stage 1 and stage 2 entries, then it must apply to any entry in the caching structures that includes information from either a stage 1 translation table entry or a stage 2 translation table entry, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Table G5-23 summarizes the required effect of the AArch32 TLB maintenance instructions, that operate only on TLBs on the PE that executes the instruction. Additional TLB maintenance instructions that:

- Apply across all PEs in the same Inner Shareable domain. Each instruction shown in the table has an Inner Shareable equivalent, identified by an `IS` suffix. For example, the Inner Shareable equivalent of TLBIALL is TLBIALLIS. See also EL2 forced broadcasting of TLB maintenance instructions.
- Can apply to separate Instruction or Data TLBs. These instructions are indicated by a footnote to the table. Arm deprecates any use of these instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- The architecture permits a TLB invalidation instruction to affect any unlocked entry in the TLB. Table G5-23 defines only the entries that each instruction must invalidate.
- All TLB instructions, including those that operate on a VA match, operate as described regardless of the value of SCTLR.M.

When interpreting the table:

Related operations Each instruction description applies also to any equivalent instruction that either:

- Applies to all PEs in the same Inner Shareable domain.
- Applies only to a data TLB, or only to an instruction TLB.

So, for example, the TLBIALL instruction description applies also to TLBIALLIS, ITLBIALL, and DTLBIALL.

TLB maintenance System instructions lists all of the TLB maintenance instructions.

Matches the VA Means the VA argument for the instruction must match the VA value in the TLB entry.

Matches the ASID Means the ASID argument for the instruction must match the ASID in use when the TLB entry was assigned.

Matches the current VMID

Means the current VMID must match the VMID in use when the TLB entry was assigned.

The dependency on the VMID applies even when the value of HCR.VM is 0, including situations where there is no use of virtualization. However, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset.

Execution at EL2 Descriptions of operations at EL2 apply only to implementations that include EL2.

For the definitions of the translation regimes referred to in the table see About VMSA v8-32.

Table G5-23 Effect of the TLB maintenance instructions

Instruction	Executed from		Effect, must invalidate any entry that matches all stated conditions[a]
	State	Mode	
TLBIALL[b]	Secure	PL1	All entries for the Secure PL1&0 translation regime. That is, all entries that were allocated in Secure state.
	Non-secure	PL1	All entries for stage 1 of the Non-secure PL1&0 translation regime that match the current VMID.
		Hyp	All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime that match the current VMID.
TLBIMVA[b]	Secure	PL1	Any entry for the Secure PL1&0 translation regime that both: • Matches the VA argument. • Matches the ASID argument, or is global.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime to which all of the following apply. The entry: • Matches the VA argument. • Matches the ASID argument, or is global. • Matches the current VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	Executed from		Effect, must invalidate any entry that matches all stated conditions[a]
	State	Mode	
TLBIASID[b]	Secure	PL1	Any entry for the Secure PL1&0 translation regime that matches the specified ASID and either: • Is from a level of lookup above the final level. • Is a non-global entry from the final level of lookup.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that both: • Matches the specified ASID and either: — Is from a level of lookup above the final level. — Is a non-global entry from the final level of lookup. • Matches the current VMID.
TLBIMVAA	Secure	PL1	Any entry for the Secure PL1&0 translation regime that matches the VA argument.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that both: • Matches the VA argument. • Matches the current VMID.
TLBIALLNSNH[c]	Secure	Monitor	All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.
	Non-secure	Hyp	
TLBIALLH[c]	Secure	Monitor	All entries for the Non-secure EL2 translation regime. That is, any entry that was allocated in Non-secure state from Hyp mode.
	Non-secure	Hyp	
TLBIMVAL	Secure	PL1	Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and both: • Matches the VA argument. • Matches the ASID argument, or is global.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and to which all of the following apply. The entry: • Matches the VA argument. • Matches the ASID argument, or is global. • Matches the current VMID.
TLBIMVAAL	Secure	PL1	Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and matches the VA argument.
	Non-secure	PL1 or Hyp	Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and both: • Matches the VA argument. • Matches the current VMID.
TLBIMVAH[c]	Secure	Monitor	Any entry for the Non-secure EL2 translation regime that matches the VA argument.
	Non-secure	Hyp	
TLBIMVALH[c]	Secure	Monitor	Any entry for the Non-secure EL2 translation regime that is from the last level of the translation table walk and matches the VA argument.
	Non-secure	Hyp	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	Executed from		Effect, must invalidate any entry that matches all stated conditions[a]
	State	Mode	
TLBIIPAS2[c, d]	Secure	Monitor[e]	Any entry for stage 2 of the PL1&0 translation regime that both: • Matches the IPA argument. • Matches the current VMID.
	Non-secure	Hyp	
TLBIIPAS2L[c, d]	Secure	Monitor[e]	Any entry for stage 2 of the PL1&0 translation regime that is from the last level of translation and both: • Matches the IPA argument. • Matches the current VMID.
	Non-secure	Hyp	

a. When a TLB maintenance instruction is executed at Secure EL1 in AArch32 state when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

b. The architecture defines variants of these instructions that apply only to instruction TLBs, and only to data TLBs. Arm deprecates any use of these variants. For more information, see the referenced description of the operation.

c. Available only in an implementation that includes EL2. See also EL2 forced broadcasting of TLB maintenance instructions.

d. This instruction is CONSTRAINED UNPREDICTABLE if executed in any AArch32 Secure privileged mode.

e. This instruction executes as a NOP when SCR.NS == 0.

G5.9.5.1 EL2 forced broadcasting of TLB maintenance instructions

In an implementation that includes EL2, when the value of HCR.FB is 1, the TLB maintenance instructions that are not broadcast across the Inner Shareable domain are forced to operate across the Inner Shareable domain when executed in a Non-secure PL1 mode. For example, when the value of HCR.FB is 1, a TLBIMVA instruction executed in a Non-secure PL1 mode performs the same invalidation as the invalidation performed by a TLBIMVAIS instruction.

G5.9.5.2 TLB maintenance with different translation granule sizes

If a TLB maintenance instruction specifying a VA affecting the EL2 translation regime is broadcast from a PE using AArch32 to a PE using AArch64 using a translation granule size that is different from the AArch32 translation granule size for that same translation regime, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.

If a TLB maintenance instruction specifying a VA affecting the PL1 translation regime is broadcast from a PE using AArch32 using one translation granule size for that translation regime for a particular ASID, VMID (if applicable), and Security state, to a PE using AArch64 where EL1 for the same ASID, VMID (if applicable), and Security state, is using a translation granule size that is different from the AArch32 translation granule size, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.10 Caches in VMSAv8-32

The Arm architecture describes the required behavior of an implementation of the architecture. As far as possible it does not restrict the implemented microarchitecture, or the implementation techniques that might achieve the required behavior.

Maintaining this level of abstraction is difficult when describing the relationship between memory address translation and caches, especially regarding the indexing and tagging policy of caches. This section:

- Summarizes the architectural requirements for the interaction between caches and memory translation.
- Gives some information about the likely implementation impact of the required behavior.

The following sections give this information:

- Data and unified caches.
- Instruction caches.

In addition Cache maintenance requirement created by changing translation table attributes describes the cache maintenance required after updating the translation tables to change the attributes of an area of memory.

For more information about cache maintenance see:

- AArch32 cache and branch predictor support. This section describes the Arm cache maintenance instructions.
- Cache maintenance System instructions. This section summarizes the System register encodings used for these operations when executing in AArch32 state.

G5.10.1 Data and unified caches

For data and unified caches, the use of memory address translation is entirely transparent to any data access other than as described in Mismatched memory attributes.

This means that the behavior of accesses from the same observer to different VAs, that are translated to the same PA with the same memory attributes, is fully coherent. This means these accesses behave as follows, regardless of which VA is accessed:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

The memory system behaves in this way without any requirement to use barrier or cache maintenance instructions.

In addition, if cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

These properties are consistent with implementing all caches that can handle data accesses as *Physically-indexed, physically-tagged* (PIPT) caches.

G5.10.2 Instruction caches

In the Arm architecture, an instruction cache is a cache that is accessed only as a result of an instruction fetch. Therefore, an instruction cache is never written to by any load or store instruction executed by the PE.

The Arm architecture permits different behaviors for instruction caches. These are identified by descriptions of the associated expected implementation. The following subsections describe the behavior associated with these cache types, including any occasions where explicit cache maintenance is required to make the use of memory address translation transparent to the instruction cache:

- Physically-indexed, physically-tagged instruction caches.
- Virtually-indexed, physically-tagged instruction caches.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The IVIPT architecture Extension.

In AArch32 state, the CTR.L1Ip field identifies the form of the instruction caches.

Note

For software to be portable between implementations that might use any of PIPT instruction caches or VIPT instruction caches, software must invalidate the instruction cache whenever any condition occurs that would require instruction cache maintenance for at least one of the instruction cache types.

G5.10.2.1 Physically-indexed, physically-tagged instruction caches

For a *Physically-indexed, physically-tagged* (PIPT) instruction cache:

- The use of memory address translation is entirely transparent to all instruction fetches other than as described in Mismatched memory attributes.
- If cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

An implementation that provides PIPT instruction caches implements the IVIPT Extension, see The IVIPT architecture Extension.

G5.10.2.2 Virtually-indexed, physically-tagged instruction caches

For a *Virtually-indexed, physically-tagged* (VIPT) instruction cache:

- The use of memory address translation is transparent to all instruction fetches other than for the effect of memory address translation on instruction cache invalidate by address operations or as described in Mismatched memory attributes.

  Note

  Cache invalidation is the only cache maintenance instruction that can be performed on an instruction cache.

- If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the VA supplied with the operation. The effect of the invalidation might not be visible to any other VA aliases of that physical memory location.

The only architecturally-guaranteed way to invalidate all aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

An implementation that provides VIPT instruction caches implements the IVIPT Extension, see The IVIPT architecture Extension.

G5.10.2.3 The IVIPT architecture Extension

Any permitted instruction cache implementation can be described as implementing the *IVIPT Extension* to the Arm architecture.

The formal definition of the Arm IVIPT Extension is that it reduces the instruction cache maintenance requirement to the following condition:

- Instruction cache maintenance is required only after writing new data to a PA that holds an instruction.

Note

Previous versions of the Arm architecture have permitted an instruction cache option that does not implement the Arm IVIPT Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.10.3 Cache maintenance requirement created by changing translation table attributes

Any change to the translation tables to change the attributes of an area of memory can require maintenance of the translation tables, as described in General TLB maintenance requirements. If the change affects the cacheability attributes of the area of memory, including any change between Write-Through and Write-Back attributes, software must ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the levels of cache that might hold copies of the locations affected by the attribute change. Any of the following changes to the Inner Cacheability or Outer Cacheability attribute creates this maintenance requirement:

- Write-Back to Write-Through.
- Write-Back to Non-cacheable.
- Write-Through to Non-cacheable.
- Write-Through to Write-Back.

The cache clean and invalidate avoids any possible coherency errors caused by mismatched memory attributes.

Similarly, to avoid possible coherency errors caused by mismatched memory attributes, the following sequence must be followed when changing the Shareability attributes of a cacheable memory location:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the location from them cache.
3. Change the Shareability attributes to the required new values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.11 VMSAv8-32 memory aborts

In a VMSAv8-32 implementation, the following mechanisms cause a PE to take an exception on a failed memory access:

Debug exception An exception caused by the debug configuration, see AArch32 Self-hosted Debug.

Alignment fault An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information, see Unaligned data access and Alignment faults.

MMU fault An MMU fault is a fault generated by the fault checking sequence for the current translation regime. See Types of MMU faults.

External abort Any memory system fault other than a Debug exception, an Alignment fault, or an MMU fault.

Collectively, these mechanisms are called *aborts*. AArch32 Self-hosted Debug and Halting Debug Events describe Debug exceptions, and the remainder of this section describes Alignment faults, MMU faults, and External aborts.

An access that causes an abort is said to be aborted, and uses the *Fault Address Registers* (FARs) and *Fault Status Registers* (FSRs) or *Exception Syndrome Registers* (ESRs) to record context information.

The exception generated on a synchronous memory abort:

- On an instruction fetch is called the Prefetch Abort exception.
- On a data access is called the Data Abort exception.

Note

The Prefetch Abort exception applies to any synchronous memory abort on an instruction fetch. It is not restricted to speculative instruction fetches.

The Exception level and PE mode that a VMSAv8-32 memory abort is taken to depends on the translation regime and stage that generate the abort. The fault context is dependent on whether:

- The abort is reported as a Prefetch Abort or as a Data Abort.
- The exception is taken from the same or a lower Exception level.

Note

A memory access from AArch32 state may be subject to one or more VMSAv8-64 translation stages. For example, a Non-secure EL0 access when EL1 is using AArch64 is subject to both stages of the VMSAv8-64 Non-secure EL1&0 translation regime. A memory abort generated on a VMSAv8-64 translation stage is handled as described in Memory aborts.

For more information, see Routing of aborts taken to AArch32 state.

External aborts can be reported synchronously or asynchronously. Asynchronous External aborts are reported using the SError exception. For more information, see External aborts.

In AArch32 state, asynchronous memory aborts are a type of External abort, and are treated as a type of Data Abort exception.

The following sections describe the abort mechanisms:

- Types of MMU faults.
- VMSAv8-32 MMU fault terminology.
- The MMU fault-checking sequence.
- Alignment faults.
- External abort on a translation table walk.
- AArch32 state prioritization of synchronous aborts from a single stage of address translation.

An access that causes an abort is said to be aborted. On an abort, System registers are used to record context information. For more information, see Exception reporting in a VMSAv8-32 implementation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.11.1 Types of MMU faults

This section describes the faults that might be detected during one of the fault-checking sequences described in The MMU fault-checking sequence. Unless indicated otherwise, information in this section applies to the fault checking sequences for both the Short-descriptor translation table format and the Long-descriptor translation table format.

MMU faults are always synchronous.

When an MMU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as any type of Device memory.

The MMU faults that might be detected during a fault checking sequence are:

- Permission fault.
- Translation fault.
- Address size fault.
- Access flag fault.
- Domain fault, Short-descriptor translation tables only.
- TLB conflict abort.

See also External abort on a translation table walk.

Note

- Although the TLB conflict abort is classified as an MMU fault, it is described in the section Translation Lookaside Buffers.
- In VMSAv8-64 an External abort on a translation table walk is classified as an MMU fault. However, in VMSAv8-32, for consistency with earlier versions of the architecture these aborts are not classified as MMU faults.

G5.11.1.1 Permission fault

A Permission fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. See About access permissions for information about conditions that cause a Permission fault.

Note

When using the Short-descriptor translation table format, the Translation Table descriptors are checked for Permission faults only for accesses to memory regions in Client domains.

A TLB might hold a translation table entry that cause a Permission fault. Therefore, if the handling of a Permission fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in Ordering and completion of TLB maintenance instructions.

In an implementation that includes EL2, this maintenance requirement applies to Permission faults in both stage 1 and stage 2 translations.

Cache or branch predictor maintenance operations cannot cause a Permission fault, except that:

- A stage 1 translation table walk performed as part of a cache or branch predictor maintenance operation can generate a stage 2 Permission fault as described in Stage 2 fault on a stage 1 translation table walk.
- When FEAT_CMOW is implemented and the Effective value of HCRX_EL2.CMOW is 1, the DCCIMVAC and ICIMVAU cache maintenance instructions can generate a stage 2 Permission fault if they do not have read and write permission at EL1 or EL0. See Permission fault.

G5.11.1.2 Translation fault

A Translation fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. A Translation fault is generated if bits[1:0] of a Translation Table descriptor identify the descriptor as either a Fault encoding or a reserved encoding. For more information, see:

- VMSAv8-32 Short-descriptor Translation Table format descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- VMSAv8-32 Long-descriptor Translation Table format descriptors.

In addition, a Translation fault is generated if the input address for a translation either does not map onto an address range of a TTBR, or the TTBR range that it maps onto is disabled. In these cases the fault is reported as a level 1 Translation fault on the translation stage at which the mapping to a region described by a TTBR failed.

The architecture guarantees that any translation table entry that causes a Translation fault is not cached, meaning the TLB never holds such an entry. Therefore, when a Translation fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

A data or unified cache maintenance by VA instruction can generate a Translation fault. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA to the Point of Coherency instruction can generate a Translation fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate a Translation fault.

G5.11.1.3 Address size fault

An Address size fault can be generated at any level of lookup, and the reported fault code identifies the lookup level.

An Address size fault is generated if the translation table entries or the TTBR for the stage of translation have nonzero address bits above the most significant bit of the maximum output address size. Because VMSAv8-32 supports a maximum PA and IPA size of 40 bits, this means any case where a translation table entry or the TTBR holds an address for which A[47:40] is nonzero generates an Address size fault.

A data or unified cache maintenance by VA instruction can generate an Address size fault. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA instruction can generate an Address size fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate an Address size fault.

The architecture guarantees that any translation table entry that causes an Address size fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Address size fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

G5.11.1.4 Access flag fault

An Access flag fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Access flag fault is generated only if all of the following apply:

- The translation tables support an Access flag bit:
  - The Short-descriptor format supports an Access flag only when SCTLR.AFE is set to 1.
  - The Long-descriptor format always supports an Access flag.
- A Translation Table descriptor with the Access flag bit set to 0 is loaded.

For more information about the Access flag bit see:

- VMSAv8-32 Short-descriptor Translation Table format descriptors
- VMSAv8-32 Long-descriptor Translation Table format descriptors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The architecture guarantees that any translation table entry that causes an Access flag fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Access flag fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

Whether any cache maintenance instruction by VA can generate Access flag faults is IMPLEMENTATION DEFINED.

Whether branch predictor invalidate by VA operations can generate Access flag faults is IMPLEMENTATION DEFINED.

For more information, see The Access flag.

G5.11.1.5 Domain fault, Short-descriptor format translation tables only

When using the Short-descriptor translation table format, a Domain fault can be generated at level 1or level 2 of lookup. The reported fault code identifies the lookup level. The conditions for generating a Domain fault are:

Level 1 When a level 1 descriptor fetch returns a valid Section level 1 descriptor, the domain field of that descriptor is checked against the DACR. A level 1 Domain fault is generated if this check fails.

Level 2 When a level 2 descriptor fetch returns a valid level 2 descriptor, the domain field of the level 1 descriptor that required the level 2 fetch is checked against the DACR, and a level 2 Domain fault is generated if this check fails.

For more information, see Domains, Short-descriptor format only.

Domain faults cannot occur on cache or branch predictor maintenance operations.

A TLB might hold a translation table entry that cause a Domain fault. Therefore, if the handling of a Domain fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in Ordering and completion of TLB maintenance instructions.

Any change to the DACR must be synchronized by a Context Synchronization event. For more information, see Synchronization of changes to AArch32 System registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.11.2 VMSAv8-32 MMU fault terminology

The Armv7 Large Physical Address Extension introduced new terminology for faults on a stage of address translation, to provide consistent terminology across all implementations. Table G5-24 shows the terminology used in this manual for an MMU faults, compared with older Arm documentation. The current terms are the same for faults that occur with the Short-descriptor translation table format and with the Long-descriptor format, and also apply to faults in a level 3 lookup when using the Long-descriptor translation table format.

Table G5-24 MMU fault terminology

Current term	Old term	Note
Level 1 Translation fault	Section Translation fault	-
Level 2 Translation fault	Page Translation fault	-
Level 3 Translation fault	-	Long-descriptor translation table format only.
Level 1 Access flag fault	Section Access flag fault	-
Level 2 Access flag fault	Page Access flag fault	-
Level 3 Access flag fault	-	Long-descriptor translation table format only.
Level 1 Domain fault	Section Domain fault	Short-descriptor translation table format only, except for reporting faults on address translation instructions in the 64-bit PAR, see Determining the PAR format. Cannot occur at level 3.
Level 2 Domain fault	Page Domain fault	
Level 1 Permission fault	Section Permission fault	-
Level 2 Permission fault	Page Permission fault	-
Level 3 Permission fault	-	Long-descriptor translation table format only.

In an implementation that includes EL2, MMU faults are also classified by the translation stage at which the fault is generated. This means that a memory access from a Non-secure PL1 or EL0 mode can generate:

- A stage 1 MMU fault, for example, a stage 1 Translation fault.
- A stage 2 MMU fault, for example, a stage 2 Translation fault.

G5.11.3 The MMU fault-checking sequence

This section describes the MMU checks made for the memory accesses required for instruction fetches and for explicit memory effects:

- If an instruction fetch faults it generates a Prefetch Abort exception.
- If an data memory access faults it generates a Data Abort exception.

For more information about Prefetch Abort exceptions and Data Abort exceptions see Handling exceptions that are taken to an Exception level using AArch32.

In VMSAv8-32, all memory accesses require VA to PA translation. Therefore, when a corresponding stage of address translation is enabled, each access requires a lookup of the Translation Table descriptor for the accessed VA. For more information, see Translation tables and subsequent sections of this chapter. MMU fault checking is performed for each level of translation table lookup. If an implementation includes EL2 and is operating in Non-secure state, MMU fault checking is performed for each stage of address translation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In an implementation that includes EL2, if a PE is executing in Non-secure state, the operating system or similar Non-secure system software defines the stage 1 translation tables in the IPA address map, and typically is unaware of the stage 2 translation from IPA to PA. However, each Non-secure stage 1 translation table access is subject to stage 2 address translation, and might be faulted at that stage.

The MMU fault checking sequence is largely independent of the translation table format, as the figures in this section show. The differences are:

When using the Short-descriptor format

- There are one or two levels of lookup.
- Lookup always starts at level 1.
- The final level of lookup checks the Domain field of the descriptor and:
  - Faults if there is no access to the Domain.
  - Checks the access permissions only for Client domains.

When using the Long-descriptor format

- There are one, two, or three levels of lookup.
- Lookup starts at either level 1 or level 2.
- Domains are not supported. All accesses are treated as Client domain accesses.

The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see About address translation for VMSAv8-32.

Note

The descriptions in this section do not include the possibility that the attempted address translation generates a TLB conflict abort, as described in TLB conflict aborts.

Types of MMU faults describes the faults that an MMU fault-checking sequence can report.

Figure G5-15 shows the process of fetching a descriptor from the translation table. For the top-level fetch for any translation, the descriptor is fetched only if the input address passes any required alignment check. As the figure shows, in an implementation that includes EL2, if the translation is stage 1 of the Non-secure PL1&0 translation regime, then the descriptor address is in the IPA address map, and is subject to a stage 2 translation to obtain the required PA. This stage 2 translation requires a recursive entry to the fault checking sequence.

Note

Figure G5-15 and Figure G5-16 give an overview of the fault checking performed by the MMU. See AArch32 state prioritization of synchronous aborts from a single stage of address translation for the complete set of possible faults and their prioritization.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

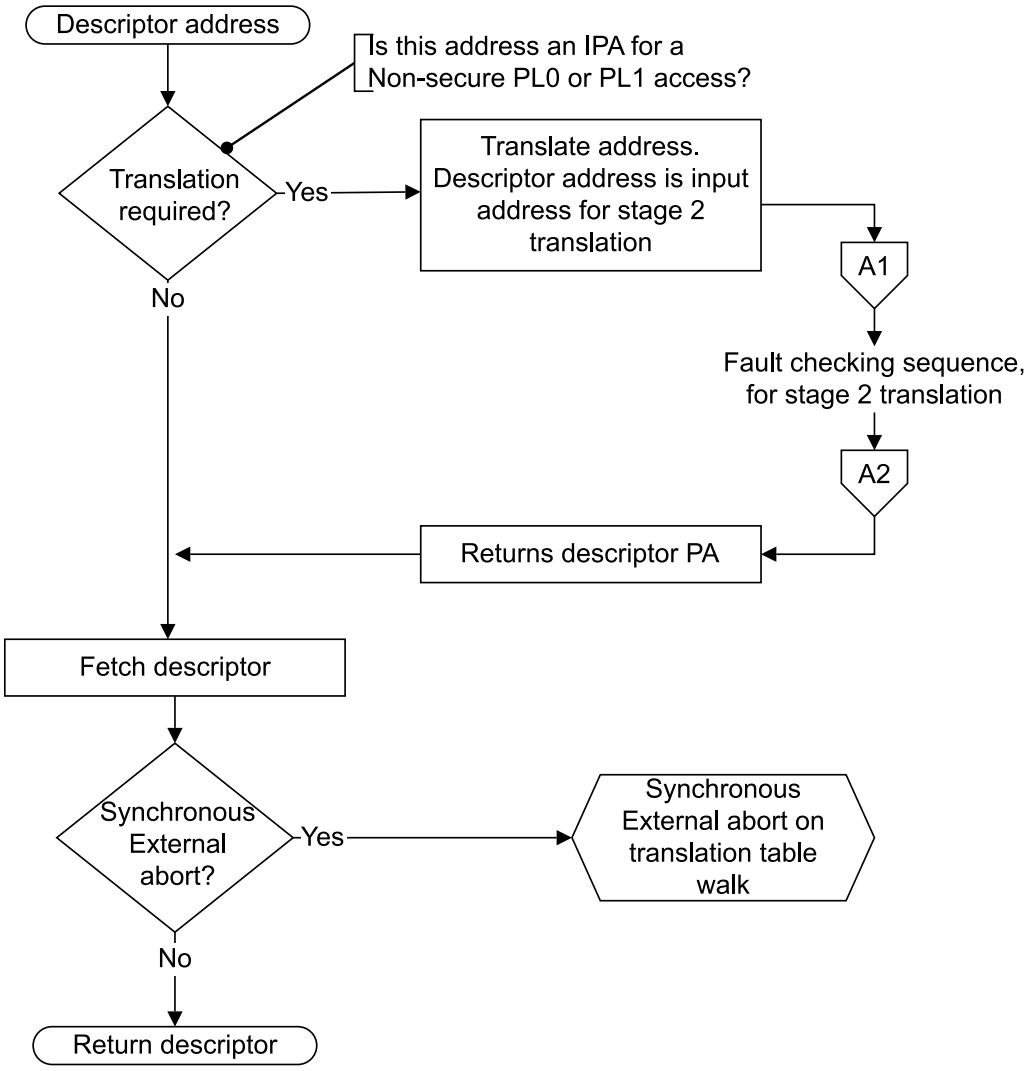

Figure G5-15 Fetching the descriptor in a VMSAv8-32 translation table walk

Figure G5-16 shows the full VMSAv8-32 fault checking sequence, including the alignment check on the initial access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

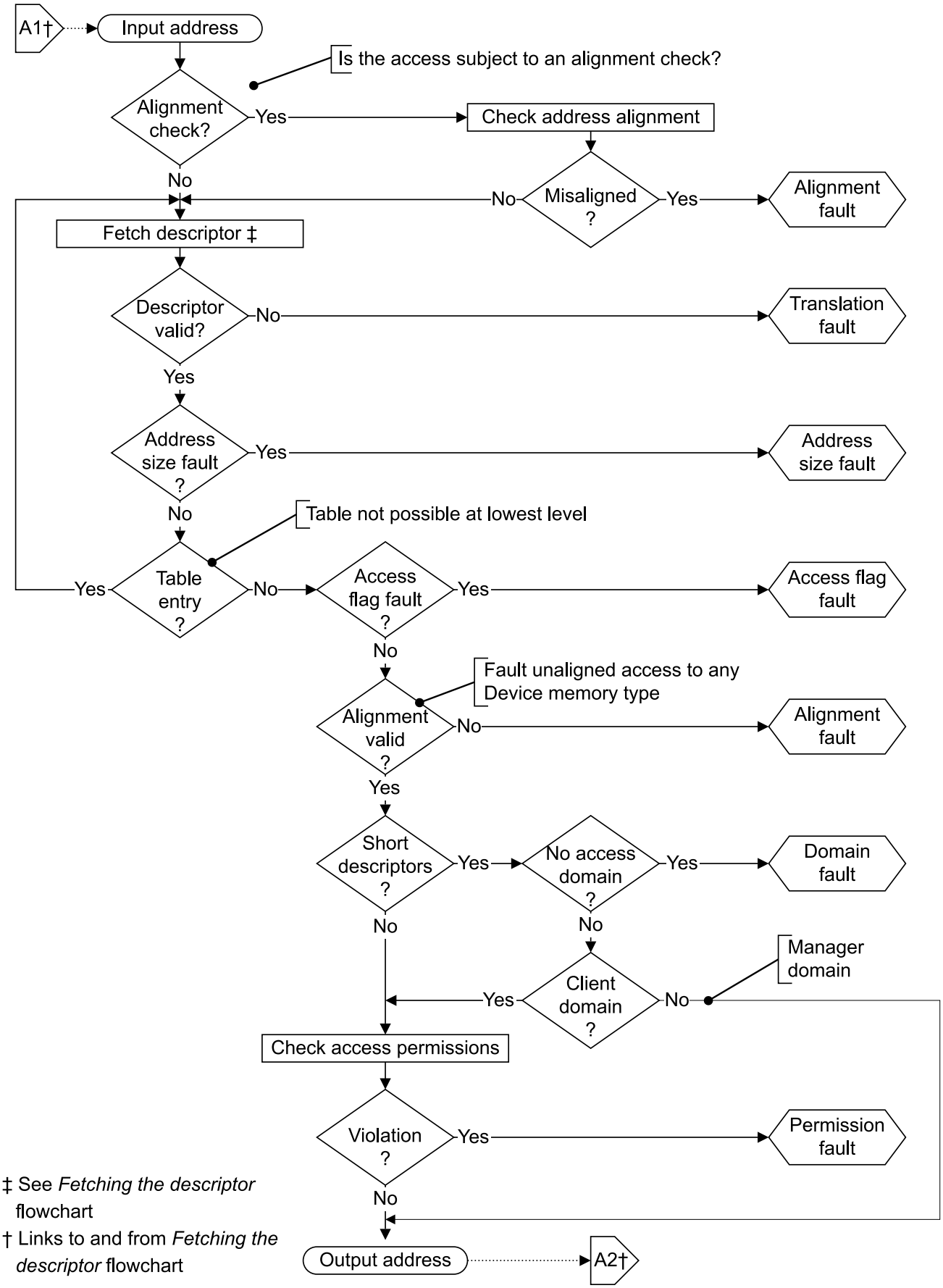

‡ See *Fetching the descriptor* flowchart

† Links to and from *Fetching the descriptor* flowchart

Figure G5-16 VMSAv8-32 fault checking sequence

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.11.3.1 Stage 2 fault on a stage 1 translation table walk

When an implementation that includes EL2 is operating in a Non-secure PL1 or EL0 mode, any memory access goes through two stages of translation:

- Stage 1, from VA to IPA.
- Stage 2, from IPA to PA.

Note

In a virtualized system that is using AArch32, typically, a Guest OS operating in a Non-secure PL1 mode defines the translation tables and translation table register entries controlling the Non-secure PL1&0 stage 1 translations. A Guest OS has no awareness of the stage 2 address translation, and therefore believes it is specifying translation table addresses in the PA map. However, it actually specifies these addresses in its IPA map. Therefore, to support virtualization, translation table addresses for the Non-secure PL1&0 stage 1 translations are always defined in the IPA address map.

On performing a translation table walk for the stage 1 translations, the descriptor addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation table lookup might generate, on a stage 2 translation:

- A Translation fault, Access flag fault, or Permission fault.
- A synchronous External abort on the memory access.

If SCR.EA is set to 1, a synchronous External abort is taken to EL3, and if EL3 is using AArch32 it is taken to Secure Monitor mode. Otherwise, these faults are reported as stage 2 memory aborts. When EL2 is using AArch32, HSR.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation table walk, and the part of the ISS field that might contain details of the instruction is invalid. For more information, see Use of the HSR.

Alternatively, a memory access made as part of a stage 1 translation table lookup might target an area of memory with the Device memory attribute assigned on the stage 2 translation of the address accessed. When the value of the HCR.PTW bit is 1, such an access generates a stage 2 Permission fault.

Note

- On most systems, such a mapping to a Device memory type on the stage 2 translation is likely to indicate a Guest OS error, where the stage 1 translation table is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.

A TLB might hold entries that depend on the effect of HCR.PTW. Therefore, if HCR.PTW is changed without changing the current VMID, the TLBs must be invalidated before executing in a Non-secure PL1 or EL0 mode. For more information, see Changing HCR.PTW.

A cache maintenance instruction executed at Non-secure PL1 can cause a stage 1 translation table walk that might generate a stage 2 Permission fault, as described in this section. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a cache maintenance by VA instruction can generate a Permission fault in this way.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Permission fault in this way.

Note

This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.

G5.11.3.1.1 The level associated with MMU faults

When an MMU fault is from a stage of translation that is using Long-descriptor translation table format, Table G5-25 shows how the LL bits in the STATUS field of DFSR, IFSR, and HSR encode the lookup level associated with the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G5-25 Use of LL bits to encode the lookup level at which the fault occurred

LL bits	Meaning
00	Level 0 of translation or translation table base register.
01	Level 1.
10	Level 2.
11	Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see Synchronous External abort errors from address translation caching structures.

G5.11.4 Alignment faults

The Arm memory architecture requires support for strict alignment checking. This checking is controlled by:

- SCTLR.A, for accesses made from any PE mode other than Hyp mode.
- HSCTLR.A, for accesses made from Hyp mode.

In addition, some instructions do not support unaligned accesses, regardless of the value of SCTLR.A or HSCTLR.A.

Unaligned data access:

- Defines when Alignment faults are generated, for both values of SCTLR.A or HSCTLR.A.
- Describes the possible generation of Alignment faults on accesses to Device memory by AArch32 Load Multiple or Store Multiple instructions when FEAT_LSMAOC is implemented.

An Alignment fault can occur on an access for which the stage of address translation is disabled.

Any unaligned access to memory region with any Device memory type attribute generates an Alignment fault.

Routing of aborts taken to AArch32 state defines the mode to which an Alignment fault is taken.

The prioritization of Alignment faults depends on whether the fault was generated because of an access to a Device memory type, or for another reason. For more information, see AArch32 state prioritization of synchronous aborts from a single stage of address translation.

G5.11.5 External abort on a translation table walk

An External abort on a translation table walk can be either synchronous or asynchronous. For more information on External aborts, see External aborts.

An External abort on a translation table walk is reported:

- If the External abort is synchronous, using:
  - A synchronous Prefetch Abort exception if the translation table walk is for an instruction fetch.
  - A synchronous Data Abort exception if the translation table walk is for a data access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the External abort is asynchronous, using an SError exception, which is taken as an asynchronous Data Abort exception.

If an implementation reports the error in the translation table walk asynchronously from executing the instruction whose instruction fetch or memory access caused the translation table walk, these aborts behave essentially as interrupts. The aborts are masked when PSTATE.A is set to 1, otherwise they are reported using the Data Abort exception.

G5.11.5.1 Behavior of External aborts on a translation table walk caused by address translation instructions

The address translation instructions summarized in Address translation System instructions require translation table walks. An External abort can occur in the translation table walk. The abort generates a Data Abort exception, and can be synchronous or asynchronous. For more information, see Handling of faults and aborts during an address translation instruction.

G5.11.6 AArch32 state prioritization of synchronous aborts from a single stage of address translation

Exception prioritization for exceptions taken to AArch32 state describes the prioritization of exceptions taken from an Exception level that is using AArch32. This section gives additional information about the prioritization of MMU faults from VMSAv8-32 translation regimes.

If a single instruction generates aborts on more than one memory access, the architecture does not define any prioritization between those aborts.

In general, the Arm architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.

Note

The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a VMSAv8-32 translation regime. This numbering has no global significance and, for example, does not correlate with the equivalent AArch64 list in MMU fault prioritization from a single address translation stage.

For a single stage of translation in a VMSAv8-32 translation regime, the following numbered list shows the priority of the possible memory management faults on a memory access. In this list:

- For memory accesses that undergo two stages of translation, the *italic entries show where the faults from the stage 2 translation can occur*. A stage 2 fault within a stage 1 translation table walk follows the same prioritization of faults.
- For synchronous External aborts from translation table walks see also Synchronous External abort errors from address translation caching structures.

The priority order, from highest priority to lowest priority, is:

1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only.
2. Translation fault due to the input address being out of the address range to be translated or requiring an AArch32 TTBR that is disabled. This includes VTCR.SL0 being inconsistent with VTCR.T0SZ or programmed to a reserved value.
3. Address size fault on an AArch32 TTBR caused by the PA being out of the range implemented.
4. *Second stage abort on a level 1 lookup of a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.* This is second stage abort during a first stage translation table walk.
5. Synchronous parity or ECC error on a level 1 lookup of a translation table walk.
6. Synchronous External abort on a level 1 lookup level of a translation table walk.
7. Translation fault on a level 1 translation table entry.
8. Address size fault on a level 1 lookup translation table entry caused by the output address being out of the range implemented.
9. *Second stage abort on a level 2 lookup of a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.* This is second stage abort during a first stage translation table walk.
10. Synchronous parity or ECC error on a level 2 lookup of a translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

11. Synchronous External abort on a level 2 lookup level of a translation table walk.
12. Translation fault on a level 2 translation table entry.
13. Address size fault on a level 2 lookup translation table entry caused by the output address being out of the range implemented.
14. *Second stage abort on a level 3 lookup of a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.* This is second stage abort during a first stage translation table walk.
15. Synchronous parity or ECC error on a level 3 lookup of a translation table walk.
16. Synchronous External abort on a level 3 lookup level of a translation table walk.
17. Translation fault on a level 3 translation table entry.
18. Address size fault on a level 3 lookup translation table entry caused by the output address being out of the range implemented.
19. Access Flag fault.
20. Alignment fault caused by the memory type.
21. Domain fault.

    Note

    Domain faults are possible only when using the VMSAv8-32 Short-descriptor translation table format, see Domain fault, Short-descriptor format translation tables only.

22. Permission fault.
23. *A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.*
24. Synchronous parity or ECC error on the memory access.
25. Synchronous External abort on the memory access.

Note

- The prioritization of TLB Conflict aborts is IMPLEMENTATION DEFINED, as the exact cause of these aborts depends on the form of TLBs implemented. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.
- The prioritization of IMPLEMENTATION DEFINED MMU faults for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

See also The MMU fault-checking sequence.

G5.11.6.1 Synchronous External abort errors from address translation caching structures

A caching structure used for caching translation table walks might support:

- An arbitrary number of levels of translation table lookup.
- One or more stages of translation, which might not correspond to the stages of an address translation lookup.

This might mean that, on a synchronous External abort arising from the caching structure, such as from a parity or ECC error, the PE cannot precisely determine one or both of the translation stage and level of lookup at which the error occurred. In this case:

- If the PE cannot determine precisely the translation stage at which the error occurred, it is reported and prioritized as a stage 1 error.
- If the PE cannot determine precisely the lookup level at which the error occurred, the level is reported and prioritized as either:
  - The lowest-numbered level that could have given rise to the error.
  - Level 1 if it the PE cannot determine any information about the level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.12 Exception reporting in a VMSAv8-32 implementation

This section describes exception reporting, in AArch32 state, in a VMSAv8-32 implementation. That is, it describes only the reporting of exceptions that are taken to an Exception level that is using AArch32. EL2 provides an enhanced reporting mechanism for exceptions taken to the Non-secure EL2 mode, Hyp mode. This means that, for VMSAv8-32, the exception reporting depends on the mode to which the exception is taken.

Note

The enhanced reporting mechanism for exceptions that are taken to Hyp mode is generally similar to the reporting of exceptions that are taken to an Exception level that is using AArch64.

About exception reporting introduces the general approach to exception reporting, and the following sections then describe exception reporting at different privilege levels:

- Reporting exceptions taken to PL1 modes.
- Fault reporting in PL1 modes.
- Summary of register updates on faults taken to PL1 modes.
- Reporting exceptions taken to Hyp mode.
- Use of the HSR.
- Summary of register updates on exceptions taken to Hyp mode.

Note

The registers used for exception reporting also report information about debug exceptions. For more information, see:

- Data Abort exceptions, taken to a PL1 mode.
- Prefetch Abort exceptions, taken to a PL1 mode.
- Reporting exceptions taken to Hyp mode.

G5.12.1 About exception reporting

In an implementation that includes EL2 and EL3, exceptions can be taken to:

- Monitor mode, if EL3 is using AArch32.
- Hyp mode, if EL2 is using AArch32.
- A Secure or Non-secure PL1 mode.

Monitor mode is a PL1 mode, but:

- It is accessible only when EL3 is using AArch32.
- It is present only in Secure state.
- When EL3 is using AArch32, System register controls route some exceptions from Non-secure state to Monitor mode. These are the only cases where taking an exception to an Exception level that is using AArch32 changes the Security state of the PE.

Exception reporting in Hyp mode differs significantly from that in the other modes, but in general, exception reporting returns:

- Information about the exception:
  - On taking an exception to Hyp mode, the *Hyp Syndrome Register*, HSR, returns syndrome information.
  - On taking an exception to any other mode, a *Fault Status Register* (FSR) returns status information.
- For synchronous exceptions, one or more addresses associated with the exceptions, returned in *Fault Address Registers* (FARs). For a permitted exception to this requirement see Fault address reporting on synchronous External aborts.

In all modes, additional IMPLEMENTATION DEFINED registers can provide additional information about exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- PE mode for taking exceptions describes how the mode to which an exception is taken is determined.
- EL2 provides:
  - Specific exception types, that can only be taken from Non-secure PL1 and EL0 modes, and are always taken to Hyp mode.
  - Routing controls that can route some exceptions from Non-secure PL1 and EL0 modes to Hyp mode.

  These exceptions are reported using the same mechanism as the Hyp mode reporting of VMSAv8-32 memory aborts, as described in this section.

Memory system faults generate either a Data Abort exception or a Prefetch Abort exception, as summarized in:

- Reporting exceptions taken to PL1 modes.
- Memory fault reporting in Hyp mode.

On an access that might have multiple aborts, the MMU fault checking sequence and the prioritization of aborts determine which abort occurs. For more information, see The MMU fault-checking sequence and AArch32 state prioritization of synchronous aborts from a single stage of address translation.

G5.12.1.1 Fault address reporting on synchronous External aborts

The general architectural requirement is that, on a synchronous abort, the faulting address is recorded in a *Fault Address Register* (FAR). This requirement is relaxed for the case of a synchronous External abort that is not a synchronous External abort on a translation table walk. In this case only:

- It is IMPLEMENTATION DEFINED whether the faulting address is recorded in a FAR.
- A bit in a fault reporting register, the FnV bit, indicates whether a valid address is recorded.

For exceptions taken to an Exception level that is using AArch32, the details of this reporting depend on whether the exception is taken to:

- A PL1 mode, as described in Reporting exceptions taken to PL1 modes.
- Hyp mode, as described in Reporting exceptions taken to Hyp mode.

G5.12.2 Reporting exceptions taken to PL1 modes

The following sections give general information about the reporting of exceptions when they are taken to a Secure or Non-secure PL1 mode:

- Registers used for reporting exceptions taken to PL1 modes.
- Data Abort exceptions, taken to a PL1 mode.
- Prefetch Abort exceptions, taken to a PL1 mode.

Fault reporting in PL1 modes then describes the fault reporting in these modes, including the encodings used for reporting the faults.

Note

Security state, Exception levels, and AArch32 execution privilege describes how the Secure and Non-secure PL1 modes map onto the Exception levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.12.2.1 Registers used for reporting exceptions taken to PL1 modes

AArch32 state defines the following registers, and register encodings, for exceptions taken to PL1 modes:

- The DFSR holds information about a Data Abort exception.
- The DFAR holds the faulting address for some synchronous Data Abort exceptions.
- The IFSR holds information about a Prefetch Abort exception.
- The IFAR holds the faulting address for some synchronous Prefetch Abort exceptions.

In addition, if implemented, the optional ADFSR and AIFSR can provide additional fault information, see Auxiliary Fault Status Registers.

G5.12.2.1.1 Auxiliary Fault Status Registers

AArch32 state defines the following Auxiliary Fault Status Registers:

- The Auxiliary Data Fault Status Register, ADFSR.
- The Auxiliary Instruction Fault Status Register, AIFSR.

The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED. An implementation can use these registers to return additional fault status information. An example use of these registers is to return more information for diagnosing parity or ECC errors.

An implementation that does not need to report additional fault information must implement these registers as RES0. This ensures that an attempt to access these registers from software executing at PL1 does not cause an Undefined Instruction exception.

G5.12.2.2 Data Abort exceptions, taken to a PL1 mode

On taking a Data Abort exception to a PL1 mode:

- If the exception is on an instruction cache or branch predictor maintenance operation by VA, its reporting depends on the value of TTBCR.EAE. For more information about the registers used when reporting the exception, see Data Abort on an instruction cache or branch predictor maintenance instruction by VA.
- Otherwise, the DFSR is updated with details of the fault, including the appropriate Fault status code. If the Data Abort exception is synchronous, DFSR.WnR is updated to indicate whether the faulted instruction was a read or a write. However, if the fault is on a cache maintenance instruction, or on an address translation instruction, WnR is set to 1, to indicate a fault on a write instruction, and the CM bit is set to 1.

  If the Data Abort is external, then DFSR provides fields for additional classification of the abort, see Provision for classification of External aborts.

  See the register description for more information about the returned fault information. See also Data Abort on a Watchpoint exception.

  If the Data Abort exception is

  — Synchronous, the DFAR is updated with the VA that caused the exception, but see Fault address reporting on synchronous External aborts for a permitted exception to this requirement.

  — Asynchronous, the DFAR becomes UNKNOWN.

DFSR.WnR and DFSR.CM are UNKNOWN on an asynchronous Data Abort exception.

For all Data Abort exceptions, if the implementation includes EL3, the Security state of the PE in the mode to which the Data Abort exception is taken determines whether the Secure or Non-secure DFSR and DFAR are updated.

G5.12.2.2.1 Data Abort on an instruction cache or branch predictor maintenance instruction by VA

If an instruction cache invalidation by VA or branch predictor invalidation by VA operation generates a Data Abort exception that is taken to a PL1 mode, the DFAR is updated to hold the faulting VA. However, the reporting of the fault depends on the value of TTBCR.EAE:

TTBCR.EAE == 0

When the value of TTBCR.EAE is 0, it is IMPLEMENTATION DEFINED which of the following is used when reporting the fault:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The DFSR indicates an Instruction cache maintenance instruction fault, and the IFSR is valid and indicates the cause of the fault, a Translation fault or Access flag fault.
- The DFSR indicates the cause of the fault, a Translation fault or Access flag fault. The IFSR is UNKNOWN.

In either case:

- DFSR.WnR is set to 1.
- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.

TTBCR.EAE == 1

When the value of TTBCR.EAE is 1:

- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.
- DFSR.STATUS indicates the cause of the fault, a Translation or Access flag fault.
- DFSR.WnR is set to 1.
- The IFSR is UNKNOWN.

G5.12.2.2.2 Data Abort on a Watchpoint exception

On taking a Data Abort exception caused by a watchpoint:

- DFSR.FS is updated to indicate a debug exception.
- DFSR.{WnR, Domain} are UNKNOWN.
- DFAR is set to the address that generated the watchpoint.

Note

- lr_ABT indicates the address of the instruction that triggered the watchpoint.

A watchpointed address can be any byte-aligned address. The address reported in DFAR might not be the watchpointed address, and:

- For a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:
  - The lowest address accessed by the instruction that triggered the watchpoint.
  - The highest watchpointed address accessed by that instruction.

  If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

  The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

  Note

  - In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.
  - The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

- For a watchpoint due to a Data Cache operation, the address is the address passed to the instruction. This might be an address that is above the watchpointed location.

G5.12.2.3 Prefetch Abort exceptions, taken to a PL1 mode

For a Prefetch Abort exception generated by an instruction fetch, the Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction, no Prefetch Abort exception is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In addition, Breakpoint Instruction, Breakpoint, and Vector Catch exceptions, generate a Prefetch Abort exception, see the following for more information:

- Exception syndrome information and preferred return address for a BKPT instruction.
- Exception syndrome information and preferred return address for a Breakpoint exception.
- Exception syndrome information and preferred return address for a Vector Catch exception.

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced sections use the term more generally, to include exception information reported in the IFSR.

On taking a Prefetch Abort exception to a PL1 mode:

- The IFSR is updated with details of the fault, including the appropriate fault code. If appropriate, the fault code indicates that the exception was generated by a debug exception.
  See the register description for more information about the returned fault information.
- For a Prefetch Abort exception generated by an instruction fetch, the IFAR is updated with the VA that caused the exception, but see Fault address reporting on synchronous External aborts for a permitted exception to this requirement.
- For a Prefetch Abort exception generated by a debug exception, the IFAR is UNKNOWN.

If the implementation includes EL3, the security state of the PE in the mode to which it takes the Prefetch Abort exception determines whether the exception updates the Secure or Non-secure IFSR and IFAR.

G5.12.3 Fault reporting in PL1 modes

The FSRs provide fault information, including an indication of the fault that occurred. The following subsections describe fault reporting in PL1 modes for each of the translation table formats:

- PL1 fault reporting with the Short-descriptor translation table format.
- PL1 fault reporting with the Long-descriptor translation table format.

Reserved encoding in the IFSR and DFSR encodings tables gives some additional information about the encodings for both formats.

Summary of register updates on faults taken to PL1 modes shows which registers are updated on each of the reported faults.

Reporting of External aborts taken from Non-secure state to Monitor mode describes how the fault status register format is determined for those aborts. For all other aborts, the current translation table format determines the format of the fault status registers.

G5.12.3.1 Reporting of External aborts taken from Non-secure state to Monitor mode

When an External abort is taken from Non-secure state to Monitor mode:

- For a Data Abort exception, the Secure DFSR and DFAR hold information about the abort.
- For a Prefetch Abort exception, the Secure IFSR and IFAR hold information about the abort.
- The abort does not affect the contents of the Non-secure copies of the fault reporting registers.

Normally, the current translation table format determines the format of the DFSR and IFSR. However, when SCR.EA is set to 1, to route External aborts to Monitor mode, and an External abort is taken from Non-secure state, this section defines the DFSR and IFSR format.

For an External abort taken from Non-secure state to Monitor mode, the DFSR or IFSR uses the format associated with the Long-descriptor translation table format, as described in PL1 fault reporting with the Long-descriptor translation table format, if any of the following applies:

- The value of the Secure TTBCR.EAE field is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The External abort is synchronous and is taken from either:
  - Hyp mode.
  - A Non-secure PL1 or EL0 mode, and the value of the Non-secure TTBCR.EAE field is 1.

Otherwise:

- For a synchronous External abort from a stage 2 translation routed to Monitor mode when the value of the Secure TTBCR.EAE field is 0 it is IMPLEMENTATION DEFINED whether:
  - The format associated with the Long-descriptor translation table format is used, as described in PL1 fault reporting with the Long-descriptor translation table format.
  - The format associated with the Short-descriptor translation table format is used, as described in PL1 fault reporting with the Short-descriptor translation table format. Arm deprecates using this format.
    When this format is used, the value of DFSR.FS[1] or IFSR.FS[1] is UNKNOWN when reporting a synchronous External abort, or a synchronous parity or ECC error, on the stage 2 translation.
- In all other cases the DFSR or IFSR uses the format associated with the Short-descriptor translation table format, as described in PL1 fault reporting with the Short-descriptor translation table format.

G5.12.3.2 PL1 fault reporting with the Short-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1 when address translation is using the Short-descriptor translation table format.

On taking an exception, bit[9] of the FSR is RAZ, or set to 0, if the PE is using this FSR format.

An FSR encodes the fault in a 5-bit FS field, that comprises FSR[10, 3:0]. Table G5-26 shows the encoding of that field. Summary of register updates on faults taken to PL1 modes shows:

- Whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.
  - For a fault reported in the DFSR, whether the DFAR holds a valid address.
- For faults that update DFSR, whether DFSR.Domain is valid.

When reading Table G5-26:

- FS values not shown in the table are reserved.
- FS values shown as DFSR only are reserved for the IFSR.

Table G5-26 FSR encodings when using the Short-description translation table format

FS	Source		Notes
00001	Alignment fault.		DFSR only. Fault on initial lookup.
00100	Fault on instruction cache maintenance.		DFSR only.
01100 01110	Synchronous External abort on translation table walk[a, b].	Level 1 Level 2	-
11100 11110	Synchronous parity or ECC error on translation table walk[a, b].	Level 1 Level 2	-
00101 00111	Translation fault[a].	Level 1 Level 2	MMU fault.
00011[c] 00110	Access flag fault[a].	Level 1 Level 2	MMU fault.
01001 01011	Domain fault[a].	Level 1 Level 2	MMU fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FS	Source		Notes
01101 01111	Permission fault[a].	Level 1 Level 2	MMU fault.
00010	Debug exception.		See AArch32 Self-hosted Debug.
01000	Synchronous External abort.		-
10000	TLB conflict abort.		See TLB conflict aborts.
10100	IMPLEMENTATION DEFINED.		Lockdown.
10101	IMPLEMENTATION DEFINED.		Unsupported Exclusive access.
11001	Synchronous parity or ECC error on memory access.		-
10110	SError exception[d].		DFSR only.
11000	SError exception[d] from a parity or ECC error on memory access.		DFSR only.

a. See The level associated with MMU faults on a Short-descriptor translation table lookup.

b. FS[1] is UNKNOWN if the reported error is from a stage 2 translation.

c. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use

d. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

G5.12.3.2.1 The level associated with MMU faults on a Short-descriptor translation table lookup

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, Permission fault, or Domain fault, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Supersection, Section, or Page descriptor.

Also see Synchronous External abort errors from address translation caching structures.

G5.12.3.2.2 The Domain field in the DFSR

The DFSR includes a Domain field. This is inherited from previous versions of the VMSA. The IFSR does not include a Domain field. Summary of register updates on faults taken to PL1 modes describes when DFSR.Domain is valid.

Arm deprecates any use of the Domain field in the DFSR. The Long-descriptor translation table format does not support a Domain field, and future versions of the Arm architecture might not support a Domain field in the Short-descriptor translation table format. Arm strongly recommends that new software does not use this field.

For both Data Abort exceptions and Prefetch Abort exceptions, software can find the domain information by performing a translation table read for the faulting address and extracting the Domain field from the translation table entry.

G5.12.3.3 PL1 fault reporting with the Long-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1mode when address translation is using the Long-descriptor translation table format.

When the PE takes an exception, bit[9] of the FSR is set to 1 if the PE is using this FSR format.

The FSRs encode the fault in a 6-bit STATUS field, that comprises FSR[5:0]. Table G5-27 shows the encoding of that field. In addition:

- For a fault taken to a PL1 mode, Summary of register updates on faults taken to PL1 modes shows whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— For a fault reported in the DFSR, whether the DFAR holds a valid address.

- For a fault taken to the Hyp mode, Summary of register updates on exceptions taken to Hyp mode shows what registers are updated on the fault.

Table G5-27 FSR encodings when using the Long-descriptor translation table format

STATUS[a]	Source	Notes
0000LL	Address size fault. LL bits indicate level[b].	MMU fault.
0001LL	Translation fault. LL bits indicate level[b].	MMU fault.
0010LL	Access flag fault. LL bits indicate level[b].	MMU fault.
0011LL	Permission fault. LL bits indicate level[b].	MMU fault.
010000	Synchronous External abort.	-
011000	Synchronous parity or ECC error on memory access.	-
010001	SError exception[c].	DFSR only.
011001	SError exception[c] from a parity or ECC error on memory access.	DFSR only.
0101LL	Synchronous External abort on translation table walk. LL bits indicate level[b].	-
0111LL	Synchronous parity or ECC error on memory access on translation table walk. LL bits indicate level[b].	-
100001	Alignment fault.	Fault on initial lookup.
100010	Debug exception.	See AArch32 Self-hosted Debug.
110000	TLB conflict abort.	See TLB conflict aborts.
110100	IMPLEMENTATION DEFINED.	Lockdown, DFSR only.
110101	IMPLEMENTATION DEFINED.	Unsupported Exclusive access.
1111LL	Domain fault. LL bits indicate level[b].	MMU fault. 64-bit PAR only, level 1 or level 2 only. Never used in DFSR, IFSR, or HSR[d].

a. STATUS values not shown in this table are reserved. STATUS values not supported in the IFSR or DFSR are reserved for the register or registers in which they are not supported.

b. See The level associated with MMU faults on a Long-descriptor translation table lookup.

c. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

d. A Domain fault can be reported using the Long-descriptor STATUS encodings only as a result of a fault on an address translation instruction. For more information, see MMU fault on an address translation instruction.

G5.12.3.3.1 The level associated with MMU faults on a Long-descriptor translation table lookup

For MMU faults, Table G5-28 shows how the LL bits in the xFSR.STATUS field encode the lookup level associated with the fault.

Table G5-28 Use of LL bits to encode the lookup level at which the fault occurred

LL bits	Meaning
00	**Address size fault** Address size fault in TTBR0 or TTBR1. **All other faults** Reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LL bits	Meaning
01	Level 1.
10	Level 2.
11	Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see Synchronous External abort errors from address translation caching structures.

G5.12.3.4 Reserved encoding in the IFSR and DFSR encodings tables

With both the Short-descriptor and the Long-descriptor FSR format, the fault encodings reserve a single encoding for Cache and TLB lockdown faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.

G5.12.4 Summary of register updates on faults taken to PL1 modes

For faults that generate exceptions that are taken to a PL1 mode, Table G5-29 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

For faults that update the DFSR using the Short-descriptor format FSR encodings, Table G5-30 shows whether DFSR.Domain is valid.

Table G5-29 Effect of a fault taken to a PL1 mode on the reporting registers

Fault	IFSR	IFAR	DFSR	DFAR
Faults reported as Prefetch Abort exceptions:				
MMU fault, always synchronous	Yes	Yes	-	-
Synchronous External abort on translation table walk	Yes	Yes	-	-
Synchronous parity or ECC error on translation table walk	Yes	Yes	-	-
Synchronous External abort	Yes	IMP DEF[a]	-	-
Synchronous parity or ECC error on memory access	Yes	Yes	-	-
TLB conflict abort	Yes	Yes	-	-
Fault reported as Data Abort exception:				

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Fault		IFSR	IFAR	DFSR	DFAR
Alignment fault, always synchronous		-	-	Yes	Yes
MMU fault, always synchronous		-	-	Yes	Yes
Fault on instruction cache maintenance, when using Long-descriptor translation table format[b]		UNK	-	Yes	Yes
Fault on instruction cache maintenance, when using Short descriptor translation table format[c]	*either*	Yes	-	Yes	Yes
	or	UNK	-	Yes	Yes
Synchronous External abort on translation table walk		-	-	Yes	Yes
Synchronous parity or ECC error on translation table walk		-	-	Yes	Yes
Synchronous External abort		-	-	Yes	IMP DEF[a]
Synchronous parity or ECC error on memory access		-	-	Yes	Yes
SError exception		-	-	Yes	UNK
SError exception from a parity or ECC error on memory access		-	-	Yes	UNK
TLB conflict abort		-	-	Yes	Yes
Debug exceptions:					
Breakpoint, Breakpoint Instruction, or Vector Catch[d]		Yes	UNK	-	-
Watchpoint[e]		-	-	Yes	Yes

a. IMPLEMENTATION DEFINED. The IFSR.FnV or DFSR.FnV bit indicates whether the register holds a valid address. See Fault address reporting on synchronous External aborts.

b. When using the Long-descriptor translation table format, there is not a specific fault code for a fault on an instruction cache maintenance instruction. For more information, see Data Abort on an instruction cache or branch predictor maintenance instruction by VA.

c. The two lines of this entry show the alternative ways of reporting the fault when using the Short-descriptor translation table format. It is IMPLEMENTATION DEFINED which methods is used, see Data Abort on an instruction cache or branch predictor maintenance instruction by VA.

d. Generates a Prefetch Abort exception.

e. Generates a Data Abort exception.

For those faults for which Table G5-29 shows that the DFSR is updated, if the fault is reported using the Short-descriptor FSR encodings, Table G5-30 shows whether DFSR.Domain is valid. In this table, UNK indicates that the fault makes DFSR.Domain UNKNOWN.

Table G5-30 Validity of Domain field on faults that update the DFSR when using the Short-descriptor encodings

DFSR.FS	Source		DFSR.Domain	Notes
00001	Alignment fault		UNK	-
00100	Fault on instruction cache maintenance instruction		UNK	-
01100 01110	Synchronous External abort on translation table walk	Level 1 Level 2	UNK Valid	-
11100 11110	Synchronous parity or ECC error on translation table walk	Level 1 Level 2	UNK Valid	-
00101 00111	Translation fault	Level 1 Level 2	UNK Valid	MMU fault
00011[a] 00110	Access flag fault	Level 1 Level 2	UNK Valid	MMU fault
01001 01011	Domain fault	Level 1 Level 2	Valid Valid	MMU fault

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSR.FS	Source		DFSR.Domain	Notes
01101 01111	Permission fault	Level 1 Level 2	UNK UNK	MMU fault
01000	Synchronous External abort		UNK	-
10000	TLB conflict abort		UNK	-
11001	Synchronous parity or ECC error on memory access		UNK	-
10110	SError exception[b]		UNK	-
11000	SError exception[b] from a parity or ECC error on memory access		UNK	-
00010	Watchpoint		UNK	

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use

b. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

G5.12.5 Reporting exceptions taken to Hyp mode

Hyp mode is the Non-secure EL2 mode. It is entered by taking an exception to Hyp mode.

Note

Software executing in Monitor mode, or at EL3 when EL3 is using AArch64, can perform an exception return to Hyp mode. This means Hyp mode can be entered either by taking an exception, or by a permitted exception return.

When EL2 is using AArch32, the following exceptions are taken to Hyp mode:

- SError exceptions, IRQ exceptions, and FIQ exceptions, from Non-secure PL1 and EL0 modes, if not routed to Secure Monitor mode, and if configured by the AMO, FMO or IMO bits. For more information, see Asynchronous exception routing controls.
- When HCR.TGE is set to 1, all exceptions that would be routed to Non-secure PL1 modes.
  For more information, see Routing exceptions from Non-secure EL0 to EL2.
- When HDCR.TDE is set to 1, any debug exception that would otherwise be taken to a Non-secure PL1 mode, see Routing debug exceptions to EL2 using AArch32.
- The privilege rules for taking exceptions mean that any exception taken from Hyp mode, if not routed to EL3, must be taken to Hyp mode.
- An abort that Routing of aborts taken to AArch32 state identifies as taken to Hyp mode.
- Hypervisor Call exceptions, and Hyp Trap exceptions, are always taken to Hyp mode. These exceptions are supported only as part of EL2.
  When EL2 is implemented, various operations from Non-secure PL1 and EL0 modes can be *trapped* to Hyp mode, using the Hyp Trap exception. For more information, see EL2 configurable controls.

Synchronous exceptions taken to Hyp mode provide syndrome information in the HSR.

On an abort exception taken to Hyp mode, the syndrome information in the HSR includes the Fault status code otherwise provided by the fault status register, and extends the fault reporting compared to that available for an exception taken to a PL1 mode.

In addition, for a Debug exception taken to Hyp mode, DBGDSCRint.MOE or DBGDSCRext.MOE shows what caused the Debug exception. This field is valid regardless of whether the Debug exception was taken from Hyp mode or from another Non-secure mode.

For more information, see the following subsections:

- Registers used for reporting exceptions taken to Hyp mode.
- Memory fault reporting in Hyp mode.
- Use of the HSR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.12.5.1 Registers used for reporting exceptions taken to Hyp mode

The following registers are used for reporting exceptions taken to Hyp mode:

- The HSR holds syndrome information for the exception.
- The HDFAR holds the VA associated with a Data Abort exception.
- The HIFAR holds the VA associated with a Prefetch Abort exception.
- The HPFAR holds bits[39:12] of the IPA associated with some aborts on stage 2 address translations.

In addition, if implemented, the optional HADFSR and HAIFSR can provide additional fault information, see Hyp Auxiliary Fault Syndrome Registers.

G5.12.5.1.1 Hyp Auxiliary Fault Syndrome Registers

EL2 also defines encodings for the following Hyp Auxiliary Fault Syndrome Registers:

- The Hyp Auxiliary Data Fault Syndrome Register, HADFSR.
- The Hyp Auxiliary Instruction Fault Syndrome Register, HAIFSR.

An implementation can use these registers to return additional fault status information for aborts taken to Hyp mode. They are the Hyp mode equivalents of the registers described in Auxiliary Fault Status Registers. An example use of these registers is to return more information for diagnosing parity or ECC errors.

The architectural requirements for the HADFSR and HAIFSR are:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.
- An implementation with no requirement for additional fault reporting can implement these registers as RES0, but the architecture does not require it to do so.

G5.12.5.2 Memory fault reporting in Hyp mode

Prefetch Abort and Data Abort exceptions taken to Hyp mode report memory faults. For these aborts, the HSR contains the following fault status information:

- The HSR.EC field indicates the type of abort, as Table G5-31 shows.
- The HSR.ISS field holds more information about the abort. In particular:
  - Bits[5:0] of this field hold the STATUS field for the abort, using the encodings defined in PL1 fault reporting with the Long-descriptor translation table format.
  - Other subfields of the ISS give more information about the exception, equivalent to the information returned in the FSR for a memory fault reported at PL1.

  See the descriptions of the ISS fields for the memory faults, referenced from the *Syndrome description* column of Table G5-31, for information about the returned fault information.

Table G5-31 HSR.EC encodings for aborts taken to Hyp mode

HSR.EC	Abort	Syndrome description
`0x20`	Prefetch Abort taken from Non-secure PL1 or EL0 mode	ISS encoding for exception from a Prefetch Abort
`0x21`	Prefetch Abort taken from Hyp mode	
`0x24`	Data Abort taken from Non-secure PL1 or EL0 mode	ISS encoding for exception from a Data Abort
`0x25`	Data Abort taken from Hyp mode	

For more information, see Use of the HSR.

A Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction that would abort if the PE attempted to execute it, no Prefetch Abort exception is generated.

G5.12.5.2.1 Register updates on exception reporting in Hyp mode

The use of the HSR, and of the other registers listed in Registers used for reporting exceptions taken to Hyp mode, depends on the cause of the Abort. In reporting these faults, in general:

- If the fault generates a synchronous Data Abort exception, the HDFAR holds the associated VA, but see Fault address reporting on synchronous External aborts for a permitted exception to this requirement.
- If the fault generates a Prefetch Abort exception, the HIFAR holds the associated VA, but see Fault address reporting on synchronous External aborts for a permitted exception to this requirement.
- In the following cases, the HPFAR holds the faulting IPA:
  - A Translation or Access flag fault on a stage 2 translation.
  - A Translation, Access flag, or Permission fault on the stage 2 translation of an address accessed in a stage 1 translation table walk.
  - A stage 2 Address size fault.

  In all other cases, the HPFAR is UNKNOWN.
- On a Data Abort exception that is taken to Hyp mode, the HIFAR is UNKNOWN.
- On a Prefetch Abort exception that is taken to Hyp mode, the HDFAR is UNKNOWN.

In addition, the reporting of particular aborts is as follows:

Abort on the stage 1 translation for a memory access from Hyp mode

The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission. The HPFAR is UNKNOWN.

Abort on the stage 2 translation for a memory access from a Non-secure PL1 or EL0 mode

This includes aborts on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation. The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission.

For any Access flag fault or Translation fault, and also for any Permission fault on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation, the HPFAR holds the IPA that caused the fault. Otherwise, the HPFAR is UNKNOWN.

Abort caused by a synchronous External abort, or synchronous parity or ECC error, and taken to Hyp mode

The HDFAR or HIFAR holds the VA that caused the fault, but see Fault address reporting on synchronous External aborts for a permitted exception to this requirement. The HPFAR is UNKNOWN.

Data Abort caused by a Watchpoint exception and routed to Hyp mode because HDCR.TDE is set to 1

When HDCR.TDE is set to 1, a Watchpoint exception generated in a Non-secure PL1 or EL0 mode, that would otherwise generate a Data Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

HDFAR is set to the address that generated the watchpoint.

Note

ELR_hyp indicates the address of the instruction that triggered the watchpoint.

A watchpointed address can be any byte-aligned address. The address reported in HDFAR might not be the watchpointed address, and, for a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:

- The lowest address accessed by the instruction that triggered the watchpoint.
- The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.

The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

Note

The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

See also Watchpoint exceptions.

In all cases, HPFAR is UNKNOWN.

Prefetch Abort caused by a Breakpoint Instruction exception and taken to Hyp mode

This abort is generated if a `BKPT` instruction is executed in Hyp mode. The abort leaves the HIFAR and HPFAR UNKNOWN.

See also Breakpoint Instruction exceptions.

Prefetch Abort caused by a Breakpoint Instruction, Breakpoint, or Vector Catch exception, and routed to Hyp mode because HDCR.TDE is set to 1

When HDCR.TDE is set to 1, a debug exception, generated in a Non-secure PL1 or EL0 mode, that would otherwise generate a Prefetch Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

The abort leaves the HIFAR and HPFAR UNKNOWN. This is identical to the reporting of a Prefetch Abort exception caused by a Debug exception on a `BKPT` instruction that is executed in Hyp mode.

Note

The difference between these two cases is:

- The Debug exception on a `BKPT` instruction executed in Hyp mode generates a Prefetch Abort exception, taken to Hyp mode, and reported in the HSR using EC value `0x21`.
- Aborts generated because HDCR.TDE is set to 1 generate a Hyp Trap exception, and are reported in the HSR using EC value `0x20`.

G5.12.5.3 Use of the HSR

The HSR holds syndrome information for any synchronous exception taken to Hyp mode. Compared with the reporting of exceptions taken to PL1 modes, the HSR:

- Always provides details of the fault. The DFSR and IFSR are not used.
- Provides more extensive information, for a wider range of exceptions.

Note

IRQ and FIQ exceptions taken to Hyp mode do not report any syndrome information in the HSR.

This section summarizes the general form of the HSR register, to show how it encodes exception syndrome information, see the register description for more information. The register comprises:

- A 6-bit Exception class field, EC, that indicates the cause of the exception.
- An instruction length bit, IL. When an exception is caused by trapping an instruction to Hyp mode, this bit indicates the length of the trapped instruction, as follows:

  0 16-bit instruction trapped.

  1 32-bit instruction trapped.

  In other cases the IL field is not valid and is RES1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An instruction specific syndrome field, ISS. Architecturally, this field could be defined independently for each defined Exception class (EC), but in practice several ISS formats are common to more than one EC.

The format of the HSR depends on the value of the EC field, as follows:

`0b000000` < **EC** ≤ `0b001100`

The ISS part of the returned value includes the CV and COND fields described in Encoding of ISS[24:20] when `0b000000` < EC ≤ `0b001100`. Figure G5-17 shows the HSR format in this case.

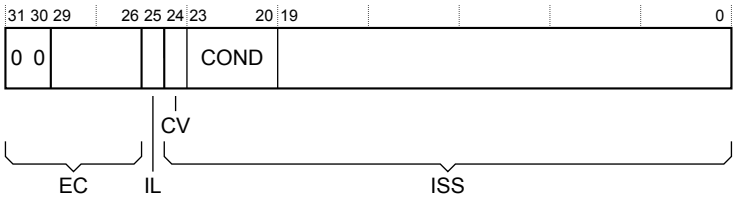

Figure G5-17 HSR format when the ISS includes CV and COND fields

EC == `0b000000` **or EC** == `0b001110`

There are no generic fields within the ISS. Figure G5-18 shows the HSR format in this case.

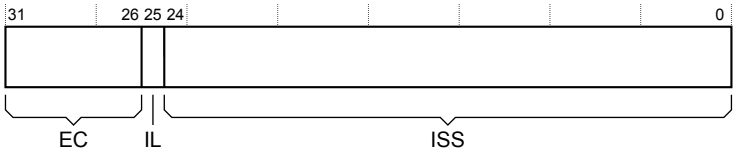

Figure G5-18 HSR format when the ISS does not include a COND field

G5.12.5.3.1 Encoding of ISS[24:20] when `0b000000` < EC ≤ `0b001100`

For EC values that are nonzero and less than or equal to `0b001100`, ISS[24:20] provides the Condition code field for the trapped instruction, together with a valid flag for this field. The encoding of this part of the ISS field is:

CV, ISS[24] Condition code valid. Possible values of this bit are:

0 The COND field is not valid.

1 The COND field is valid.

COND, ISS[23:20]

The Condition code for the trapped instruction. This field is valid only when CV is set to 1.

If CV is set to 0, this field is RES0.

The full descriptions of the HSR.ISS formats give more information about the CV field.

Note

In some circumstances, it is IMPLEMENTATION DEFINED whether a conditional instruction that fails its Condition code check generates an Undefined Instruction exception, see Conditional execution of undefined instructions.

G5.12.5.3.2 HSR exception classes

Table G5-32 shows the encoding of the HSR exception class field, EC. Values of EC not shown in the table are reserved. For each EC value, the table references a subsection of the description of the HSR that describes the associated ISS format and gives information about the cause of the exception, for example the configuration required to enable the associated trap.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G5-32 HSR.EC field encoding

EC	Exception class	ISS description, or notes
0b000000	Unknown reason	ISS encoding for exceptions with an unknown reason.
0b000001	Trapped WFI or WFE instruction	ISS encoding for exception from a WFI or WFE instruction.
0b000011	Trapped MCR or MRC access with (coproc == 0b1111)	ISS encoding for exception from an MCR or MRC access.
0b000100	Trapped MCRR or MRRC access with (coproc == 0b1111)	ISS encoding for exception from an MCRR or MRRC access.
0b000101	Trapped MCR or MRC access with (coproc == 0b1110)	ISS encoding for exception from an MCR or MRC access.
0b000110	Trapped LDC or STC access	ISS encoding for exception from an LDC or STC instruction.
0b000111	Advanced SIMD or floating-point functionality trapped by a HCPTR.{TASE, TCP10} control	ISS encoding for exception from an access to SIMD or floating-point functionality, resulting from HCPTR.
0b001000	Trapped VMRS access, from ID group traps, that is not reported using EC 0b000111	ISS encoding for exception from an MCR or MRC access. This trap is not taken if the HCPTR settings trap the access.
0b001100	Trapped MRRC access with (coproc == 0b1110)	ISS encoding for exception from an MCRR or MRRC access.
0b001110	Illegal exception return to AArch32 state	ISS encoding for exception from an Illegal state or PC alignment fault.
0b010001	Exception on SVC execution in AArch32 state routed to EL2	ISS encoding for exception from HVC or SVC instruction execution.
0b010010	HVC instruction execution in AArch32 state, when HVC is not disabled	
0b010011	Trapped execution of SMC instruction in AArch32 state	ISS encoding for exception from SMC instruction execution.
0b100000	Prefetch Abort from a lower Exception level	ISS encoding for exception from a Prefetch Abort.
0b100001	Prefetch Abort taken without a change in Exception level	
0b100010	PC alignment exception.	ISS encoding for exception from an Illegal state or PC alignment fault.
0b100100	Data Abort from a lower Exception level	ISS encoding for exception from a Data Abort.
0b100101	Data Abort taken without a change in Exception level	

All EC encodings not shown in Table G5-31 are reserved by Arm.

G5.12.6 Summary of register updates on exceptions taken to Hyp mode

For memory system faults that generate exceptions that are taken to Hyp mode, Table G5-33 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

Note

For a list of the MMU faults, see Types of MMU faults.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G5-33 Effect of an exception taken to Hyp mode on the reporting registers

Fault	HSR	HIFAR	HDFAR	HPFAR
Faults reported as Prefetch Abort exceptions:				
MMU fault [a] at stage 1.	Yes	Yes	UNK	UNK
Translation or Access flag MMU fault[a] at stage 2.	Yes	Yes	UNK	Yes
Other[b] MMU fault[a] at stage 2.	Yes	Yes	UNK	UNK
Stage 2 MMU fault[a] on a stage 1 translation.	Yes	Yes	UNK	Yes
Synchronous External abort on translation table walk.	Yes	Yes	UNK	UNK
Synchronous parity or ECC error on translation table walk.	Yes	Yes	UNK	UNK
Synchronous External abort.	Yes	IMP DEF[c]	UNK	UNK
Synchronous parity or ECC error on memory access.	Yes	Yes	UNK	UNK
Fault reported as Data Abort exception:				
MMU fault [a] at stage 1.	Yes	UNK	Yes	UNK
Translation or Access flag MMU fault[a] at stage 2.	Yes	UNK	Yes	Yes
Other[b] MMU fault[a] at stage 2.	Yes	UNK	Yes	UNK
Stage 2 MMU fault[a] on a stage 1 translation.	Yes	UNK	Yes	Yes
Synchronous External abort on translation table walk.	Yes	UNK	Yes	UNK
Synchronous parity or ECC error on translation table walk.	Yes	UNK	Yes	UNK
Synchronous External abort.	Yes	UNK	IMP DEF[c]	UNK
Synchronous parity or ECC error on memory access.	Yes	UNK	Yes	UNK
SError exception.	Yes	UNK	UNK	UNK
SError exception from a parity or ECC error on memory access.	Yes	UNK	UNK	UNK
Debug exception:				
Breakpoint Instruction[d], generates a Prefetch Abort exception.	Yes	UNK	-	UNK
Debug exception routed to Hyp mode because HDCR.TDE is set to 1. Generates a Hyp Trap exception.				
Breakpoint Instruction or Vector Catch.	Yes	UNK	-	UNK
Watchpoint.	Yes	-	Yes	UNK

a. For more information, see Classification of MMU faults taken to Hyp mode

b. MMU fault other than a Translation fault or an Access flag fault.

c. IMPLEMENTATION DEFINED. The FnV bit in the HSR.ISS field indicates whether the register holds a valid address. See Fault address reporting on synchronous External aborts.

d. All other debug exceptions are not permitted in Hyp mode.

Note

Unlike Table G5-29, the Hyp mode fault reporting table does not include an entry for a fault on an instruction cache maintenance instruction. That is because, when the fault is taken to Hyp mode, the reporting indicates the cause of the fault, for example a Translation fault, and ISS.CM is set to 1 to indicate that the fault was on a cache maintenance instruction, see ISS encoding for exception from a Data Abort.

G5.12.6.1 Classification of MMU faults taken to Hyp mode

This subsection gives more information about the MMU faults shown in Table G5-33.

Note

All MMU faults are synchronous.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The table uses the following descriptions for MMU faults taken to Hyp mode:

MMU fault at stage 1

This is an MMU fault generated on a stage 1 translation performed in the Non-secure EL2 translation regime.

MMU fault at stage 2

This is an MMU fault generated on a stage 2 translation performed in the Non-secure PL1&0 translation regime.

As the table shows, for the faults in this group:

- Translation and Access flag faults update the HPFAR.
- Permission faults leave the HPFAR UNKNOWN.

MMU stage 2 fault on a stage 1 translation

This is an MMU fault generated on the stage 2 translation of an address accessed in a stage 1 translation table walk performed in the Non-secure PL1&0 translation regime. For more information about these faults see Stage 2 fault on a stage 1 translation table walk.

Figure G5-1 shows the different translation regimes and associated stages of translation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.13 Address translation instructions

The System register encoding space includes encodings for instructions that either:

- Translate a *virtual address* (VA) to a *physical address* (PA).
- Translate a *virtual address* (VA) to an *intermediate physical address* (IPA).

Address translation System instructions summarizes these instructions.

When using the Short-descriptor translation table format, all translations performed by these instructions take account of TEX remap when this is enabled, see Short-descriptor format memory region attributes, with TEX remap.

An address translation instruction that executes successfully returns the output address, a PA or an IPA, in the PAR. This is a 64-bit register that can hold addresses of up to 40 bits.

It is IMPLEMENTATION DEFINED whether the address translation instructions return the values held in a TLB or the result of a translation table walk. Therefore, Arm recommends that these instructions are not used at a time when the TLB entries might be different from the underlying translation tables held in memory.

The following sections give more information about these instructions:

- Address translation instruction naming and operation summary.
- Encoding and availability of the address translation instructions.
- Determining the PAR format.
- Handling of faults and aborts during an address translation instruction.

G5.13.1 Address translation instruction naming and operation summary

Some older documentation uses the original names for the address translation instructions that were included in the original Armv7 documentation. Table G5-34 summarizes the instructions that are available in AArch32 state, and relates the old instruction names to the current names.

Table G5-34 Naming of address translation instructions

Name	Old name	Description
ATS1CPR, ATS1CPW, ATS1CUR, ATS1CUW ATS1CPRP, ATS1CPWP	V2PCWPR, V2PCWPW, V2PCWUR, V2PCWUW Not applicable[a]	See ATS1C**, Address translation stage 1, current security state
ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW	V2POWPR, V2POWPW, V2POWUR, V2POWUW	See ATS12NSO**, Address translation stages 1 and 2, Non-secure state only
ATS1HR, ATS1HW	Not applicable[b]	See ATS1H*, Address translation stage 1, Hyp mode

a. Instructions are added by FEAT_PAN2 and do not have a previous name.

b. Instructions are part of EL2 and have no equivalent in the older descriptions.

In an implementation that does not include EL2, there is no distinction between stage 1 translations and stage 1 and 2 combined translations.

For the *stage 1 current state* and *stages 1 and 2 Non-secure state only* instructions, the meanings of the final letters of the names are:

PR PL1 mode, read operation.

PRP PL1 mode, read operation, taking account of PSTATE.PAN.

PW PL1 mode, write operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PWP PL1 mode, write operation, taking account of PSTATE.PAN.

UR User mode, read operation.

UW User mode, write operation.

Note

User mode can be described as the unprivileged mode. It is the only EL0 mode.

For the *stage 1 Hyp mode* instructions, the last letter of the instruction name is `R` for the read operation and `W` for the write operation.

See also Encoding and availability of the address translation instructions.

G5.13.1.1 ATS1C**, Address translation stage 1, current security state

Any VMSAv8-32 implementation supports the ATS1C** instructions. They can be executed by any software executing at PL1 or higher, in either Security state.

The ATS1C** instructions are ATS1CPR, ATS1CPW, ATS1CUR, and ATS1CUW and, when FEAT_PAN2 is implemented, ATS1CPRP and ATS1CPWP. These instructions perform the address translations of the PL1&0 translation regime.

In an implementation that includes EL2, when executed in Non-secure state, these instructions return the IPA that is the output address of the stage 1 translation. Figure G5-1 shows the different translation regimes.

Note

The Non-secure PL1 and EL0 modes have no visibility of the stage 2 address translations, that can be defined only at EL2, and translate IPAs to be PAs.

See Determining the PAR format for the format used when returning the result of these instructions.

G5.13.1.2 ATS12NSO**, Address translation stages 1 and 2, Non-secure state only

A VMSAv8-32 implementation supports the ATS12NSO** instructions only if it includes EL2. In an implementation that includes EL2, in AArch32 state, they can be executed:

- By software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
- If the implementation includes EL3, when EL3 is using AArch32, by software executing in Secure state at PL1.

The ATS12NSO** instructions are ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, and ATS12NSOUW.

In an implementation that includes EL3, when EL3 is using AArch64 and EL1 is using AArch32, any execution of an ATS12NSO** instruction at Secure EL1 is trapped as an exception that is taken to EL3.

In an implementation that does not include EL2, but includes EL3, when EL3 is using AArch32 these instructions are not UNDEFINED but each instruction behaves in the same way as the equivalent ATS1C** instruction.

If an implementation does not include EL2 and does not include EL3 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see Unallocated System register access instructions.

Arm deprecates use of these instructions from any Secure PL1 mode other than Monitor mode.

In Secure state and in Non-secure Hyp mode these instructions perform the translations made by the Non-secure PL1&0 translation regime.

These instructions always return the PA and final attributes generated by the translation. That is, for an implementation that includes EL2, they return:

- The result of the two stages of address translation for the specified Non-secure input address.
- The memory attributes obtained by the combination of the stage 1 and stage 2 attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

From Hyp mode, the ATS1C** and ATS12NSO** instructions both return the results of address translations that would be performed in the Non-secure modes other than Hyp mode. The difference is:

- The ATS1C** instructions return the Non-secure PL1 view of the associated address translation. That is, they return the IPA output address corresponding to the VA input address.
- The ATS12NSO** instructions return the EL2, or Hyp mode, view of the associated address translation. That is, they return the PA output address corresponding to the VA input address, generated by two stages of translation.

See Determining the PAR format for the format used when returning the result of these instructions.

G5.13.1.3 ATS1H*, Address translation stage 1, Hyp mode

A VMSAv8-32 implementation supports the ATS1H* instructions only if it includes EL2. They can be executed by:

- Software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
- Software executing in Secure state in Monitor mode.

The ATS1H* instructions are ATS1HR and ATS1HW. In an implementation that includes EL3, these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure PL1 mode other than Monitor mode.

If an implementation does not include EL2 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see Unallocated System register access instructions.

These instructions perform the translations made by the Non-secure EL2 translation regime. The instruction takes a VA input address and returns a PA output address.

These instructions always return a result in a 64-bit format PAR.

G5.13.2 Encoding and availability of the address translation instructions

Software executing at EL0 never has any visibility of the address translation instructions, but software executing at PL1 or higher can use the unprivileged address translation instructions to find the address translations used for memory accesses by software executing at PL1 and EL0.

Note

For information about translations when the stage of address translation is disabled see The effects of disabling address translation stages on VMSAv8-32 behavior.

Table G5-35 shows the encodings for the address translation instructions, and their availability in different implementations in different PE modes and states.

Table G5-35 Address translation instructions in AArch32 state

opc1	CRm	opc2	Name	Type	Description
All VMSAv8-32 implementations, in all modes, at PL1 or higher, see ATS1C**, Address translation stage 1, current security state					
0	c8	0	ATS1CPR	WO	PL1 stage 1 read translation, current state
		1	ATS1CPW	WO	PL1 stage 1 write translation, current state
		2	ATS1CUR	WO	Unprivileged stage 1 read translation, current state
		3	ATS1CUW	WO	Unprivileged stage 1 write translation, current state
	c9	0	ATS1CPRP[a]	WO	PL1 stage 1 read translation, current state, PSTATE.PAN[a]
		1	ATS1CPWP[a]	WO	PL1 stage 1 write translation, current state, PSTATE.PAN[a]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc1	CRm	opc2	Name	Type	Description
Implementation includes EL2, in Non-secure Hyp mode and Secure PL1 modes, see ATS12NSO**, Address translation stages 1 and 2, Non-secure state only					
0	c8	4	ATS12NSOPR	WO	Non-secure PL1 stage 1 and 2 read translation
		5	ATS12NSOPW	WO	Non-secure PL1 stage 1 and 2 write translation
		6	ATS12NSOUR	WO	Non-secure unprivileged stage 1 and 2 read translation
		7	ATS12NSOUW	WO	Non-secure unprivileged stage 1 and 2 write translation
Implementation includes EL2, in Non-secure Hyp mode and Secure Monitor mode, see ATS1H*, Address translation stage 1, Hyp mode					
4	c8	0	ATS1HR	WO	Hyp mode stage 1 read translation
		1	ATS1HW	WO	Hyp mode stage 1 write translation

a. Instruction only supported when FEAT_PAN2 is implemented.

The result of an instruction is always returned in the PAR. The PAR is a RW register and:

- In all implementations, the 32-bit format PAR is accessed using an `MCR` or `MRC` instruction with CRn set to c7, CRm set to c4, and opc1 and opc2 both set to 0.
- The 64-bit format PAR is accessed using an `MCRR` or `MRRC` instruction with CRm set to c7, and opc1 set to 0.

Address translation instructions that are not available in a particular implementation are reserved and CONSTRAINED UNPREDICTABLE. For example:

- In an implementation that does not include EL2, the encodings with an `opc1` value of 4 are reserved and CONSTRAINED UNPREDICTABLE. These are the ATS1H* instructions.
- In an implementation that does not include either EL2 or EL3, the encodings with `opc2` values of 4-7 are reserved and CONSTRAINED UNPREDICTABLE. These are the ATS12NSO** instructions.

The CONSTRAINED UNPREDICTABLE behavior of these encodings is that they are UNDEFINED, see Unallocated System register access instructions.

G5.13.3 Determining the PAR format

The PAR is a 64-bit register that supports both 32-bit and 64-bit PAR formats. This section describes how the PAR format is determined, for returning a result from each of the groups of address translation instructions. The returned result might be the translated address, or might indicate a fault on the translation, see Handling of faults and aborts during an address translation instruction.

ATS1C instructions**

Address translations for the current state. From modes other than Hyp mode:

- TTBCR.EAE determines whether the result is returned using the 32-bit or the 64-bit PAR format.
- If the implementation includes EL3, the translation performed is for the current security state and, depending on that state:
  - The Secure or Non-secure TTBCR.EAE determines the PAR format.
  - The result is returned to the Secure or Non-secure instance of the PAR

Instructions executed in Hyp mode always return a result to the Non-secure PAR, using the 64-bit format.

ATS12NSO instructions**

Address translations for the Non-secure PL1 and EL0 modes. These instructions return a result using the 64-bit PAR format if at least one of the following is true:

- The Non-secure TTBCR.EAE bit is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The implementation includes EL2, and the value of HCR.VM is 1.

Otherwise, the instruction returns a result using the 32-bit PAR format.

Instructions executed in a Secure PL1 mode return a result to the Secure PAR. Instructions executed in Hyp mode return a result to the Non-secure PAR.

ATS1H* instructions

Address translations from Hyp mode. These instructions always return a result using the 64-bit PAR format.

Instructions executed in Secure Monitor mode return a result to the Secure PAR. Instructions executed in Non-secure Hyp mode return a result to the Non-secure PAR.

G5.13.4 Handling of faults and aborts during an address translation instruction

When a stage of address translation is enabled, any corresponding address translation instruction requires a translation table lookup, and this might require a translation table walk. However, the input address for the translation might be a faulting address, either because:

- The translation table entries used for the translation indicate a fault.
- A stage 2 fault or an External abort occurs on the required translation table walk.

VMSAv8-32 memory aborts describes the faults that might occur on a translation table walk in AArch32 state.

How the fault is handled, and whether it generates an exception, depends on the cause of the fault, as described in:

- MMU fault on an address translation instruction.
- External abort during an address translation instruction.
- Stage 2 fault on a current state address translation instruction.

G5.13.4.1 MMU fault on an address translation instruction

In the following cases, an MMU fault on an address translation is reported in the PAR, and no abort is taken. This applies:

- For a faulting address translation instruction executed in Hyp mode, or in a Secure PL1 mode.
- For a faulting address translation instruction executed in a Non-secure PL1 mode, for cases where the fault would generate a stage 1 abort if it occurred on the equivalent load or store operation.

Using the PAR to report a fault on an address translation instruction gives more information about how these faults are reported.

Note

- The Domain fault encodings shown in Table G5-27 are used only for reporting a fault on an address translation instruction that uses the 64-bit PAR format. That is, they are used only in an implementation that includes EL2, and are used for reporting a Domain fault on either:
  - An ATS1C** instruction executed in Hyp mode.
  - An ATS12NSO** instruction executed when the value of HCR.VM is 1.

  These encodings are never used for fault reporting in the DFSR, IFSR, or HSR.
- For an address translation instruction executed in a Non-secure PL1 mode, for a fault that would generate a stage 2 abort if it occurred on the equivalent load or store operation, the stage 2 abort is generated as described in Stage 2 fault on a current state address translation instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.13.4.1.1 Using the PAR to report a fault on an address translation instruction

For a fault on an address translation instruction for which no abort is taken, the PAR is updated with the following information, to indicate the fault:

- The fault code, that would normally be written to the Fault status register. The code used depends on the current translation table format, as described in either:
  - PL1 fault reporting with the Short-descriptor translation table format.
  - PL1 fault reporting with the Long-descriptor translation table format.

  See also the Note at the start of Determining the PAR format about the Domain fault encodings shown in Table G5-27.
- A status bit, that indicates that the translation operation failed.

The fault does not update any Fault Address Register.

G5.13.4.2 External abort during an address translation instruction

As stated in External abort on a translation table walk, an External abort on a translation table walk generates a Data Abort exception. The abort can be synchronous or asynchronous, and behaves as follows:

Synchronous External abort on a translation table walk

The fault status and fault address registers of the Security state to which the abort is taken are updated. The fault status register indicates the appropriate External abort on a Translation fault, and the fault address register indicates the input address for the translation.

The PAR is UNKNOWN.

Asynchronous External abort on a translation table walk

The fault status register of the Security state to which the abort is taken is updated, to indicate the asynchronous External abort. No fault address registers are updated.

The PAR is UNKNOWN.

G5.13.4.3 Stage 2 fault on a current state address translation instruction

If the PE is in a Non-secure PL1 mode and executes one of the ATS1C** instructions, then a fault in the stage 2 translation of an address accessed in a stage 1 translation table lookup generates an exception. This is equivalent to the case described in Stage 2 fault on a stage 1 translation table walk. When this fault occurs on an ATS1C** address translation instruction:

- A Hyp Trap exception is taken to Hyp mode.
- The PAR is UNKNOWN.
- The HSR indicates that:
  - The fault occurred on a translation table walk.
  - The operation that faulted was a cache maintenance instruction.
- The HPFAR holds the IPA that faulted.
- The HDFAR holds the VA that the executing software supplied to the address translation instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.14 Pseudocode description of VMSAv8-32 memory system operations

This section contains a list of pseudocode functions describing VMSAv8-32 memory operations. The following subsections describe the pseudocode functions:

- Full Physical Address.
- Translation regime.
- Address translation.
- Long-descriptor Translation table walk.
- Short-descriptor Translation table walk.
- Memory attribute decoding.
- Fault detection.

See also the descriptions of pseudocode for general memory system operations in Pseudocode description of general memory System instructions.

G5.14.1 Full Physical Address

A complete physical address necessary to identify a location in physical memory is captured by the type `FullAddress`. This is composed of:

- A bitstring address, which identifies the physical address.
- An enumeration `paspace`, which identifies the physical address space.

G5.14.2 Translation regime

The architecture specifies translation regimes in terms of Privilege Level (PL). An alternative approach is used in pseudocode where regimes are expressed in terms of Exception Levels instead, mirroring regimes in AArch64. The pseudocode and ARM use a differently named but equivalent set of regimes:

Table G5-36 Pseudocode and equivalent ARM regimes

Pseudocode Regime	Equivalent ARM regime
Regime_EL10	Secure PL1&0 when EL3 is AArch64 or Non-Secure PL1&0
Regime_EL30	Secure PL1&0 when EL3 is AArch32
Regime_EL2	Non-Secure PL2

G5.14.3 Address translation

`AArch32_TranslateAddress()` acts as the entry point to VMSAv8-32 and performs the required address translation based on the provided parameters and system register configurations. The function returns an `AddressDescriptor` structure holding valid data for either of the following:

- Target memory address and attributes for a non-faulting translation.
- Fault details holding data to be populated in syndrome registers.

`AArch32_FullTranslate()` selects the translation regime and performs first and potentially second stage of translation returning the physical address (PA) and attributes of target memory. `AArch32_S1TranslateLD()` carries out the first stage of translation when stage 1 is not disabled and Long-descriptor format is used, mapping the virtual address (VA) to the intermediate physical address (IPA) and carrying out permission checks. Alternatively `AArch32_S1TranslateSD()` carries out the first stage of translation using the Short-descriptor format along with

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Domain checks and TEX memory attribute mapping. Otherwise, `AArch32_S1DisabledOutput()` assigns the appropriate memory attributes and flat maps the input address to the output address. `AArch32_S2Translate()` carries out stage 2 translation for Regime_EL10 when enabled, mapping the IPA to the PA. Otherwise, the IPA is the PA.

G5.14.4 Long-descriptor Translation table walk

A separate walk function is dedicated for Stage 1 Long-descriptor format, `AArch32_S1WalkLD()`, and Stage 2, `AArch32_S2Walk()`, which supports only Long-descriptor format. Each use walk parameters extracted from related system registers and held in S1TTWParams for stage 1 and S2TTWParams for stage 2. Parameters are collected based on the active translation regime. For instance, stage 1 EL2 translation regime parameters are obtained and returned by the function `AArch32_S1TTWParamsEL2()`. Given these parameters, a walk initializes a walk state of the type TTWState, holding the base address of the first translation table.

The walk progressively fetches and decodes Translation Table descriptors, updating the walk state to the next base address as it descends through the levels of tables until a Block or Page descriptor is discovered or an invalid descriptor is fetched. Decoding the descriptor for both stage 1 and stage 2 walks is carried out by the function `AArch32_DecodeDescriptorTypeLD()`.

For a non-faulting walk, a valid final walk state is returned, otherwise a faulting walk could report one of the following at a specified level:

- Translation Fault.
- Address Size Fault.
- Access Flag Fault.

G5.14.5 Short-descriptor Translation table walk

Short-Descriptor format is only supported for Regime_EL10 and Regime_EL30 (PL1&0) Stage 1 and a separate walk function is dedicated for that, `AArch32_S1WalkSD()`. The limited number of parameters are collected in the walk function and would otherwise follow a similar flow to Long-descriptor formats of iteratively updating the walk state. The walk notably collects the domain and Short-descriptor format type which are unique to Short-descriptor formats. The descriptor type is decoded using `AArch32_DecodeDescriptorTypeSD()`.

For a non-faulting walk, a valid final walk state is returned, otherwise a faulting walk could report one of the following at a specified level:

- Translation Fault.
- Address Size Fault.
- Access Flag Fault (when SCTLR.AFE is configured to support Access flags).

G5.14.6 Memory attribute decoding

If a stage of translation is enabled, Fetched Leaf descriptors encode memory attributes assigned to the output of translation. Stage 1 Long-descriptor format memory attributes are decoded by the function `S1DecodeMemAttrs()`. Likewise, stage 2 memory attributes are decoded by the function `S2DecodeMemAttrs()` followed by combining stage 1 and stage 2 attributes by the function `S2CombineS1MemAttrs()`. A separate set of functions are used to assign memory attributes to the output of Short-descriptor format. `AArch32_DefaultTEXDecode()` is used when TEX remapping is disabled, otherwise `AArch32_RemappedTEXDecode()` defines output memory attributes.

G5.14.7 Fault detection

As soon as translation is invoked, a reserve FaultRecord accompanies the process, capturing the stage and level of translation as it proceeds. When a fault is detected, it is reflected in the FaultRecord and reported back as the result of translation with the most recent state to be reported already captured within. The following functions detect a certain type of fault, their outputs are all boolean with a TRUE value on detection:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `AArch32_S1LDHasPermissionsFault()` and `AArch32_S2HasPermissionsFault()` detect a permissions fault for stage 1 and stage 2 respectively for Long-descriptor format.
  `AArch32_S1SDHasPermissionsFault()` detects a permissions fault for a translation in Short-descriptor format. Note that for atomic instructions introduced by FEAT_LSE, these functions are called twice, once to check for read permissions and another for write allowing the correct failure to be reported.
- `AArch32_S1HasAlignmentFaultDueToMemType()` and `AArch32_S2HasAlignmentFaultDueToMemType()` detect an alignment fault for stage 1 and stage 2 respectively.
- `AArch32_S2InconsistentSL()` detects a stage 2 translation fault caused by erroneous configuration of the VTCR.SL0 field.
- `AArch32_VAIsOutOfRange()` detects a stage 1 translation fault caused by virtual addresses larger than the address input size configured. Similarly, `AArch32_IPAIsOutOfRange()` detects a stage 2 translation fault caused by the output of stage 1 being larger than the configured input size for stage 2. Both are solely part of Long-descriptor format translation.

Note

Domain faults are detected inline as part of `AArch32_S1TranslateSD()` since they are a simple equality check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.15 About the System registers for VMSAv8-32

The System registers and System instructions that are accessible in AArch32 state are almost all in the encoding space described in The AArch32 System register interface. This section gives general information about these registers, which comprise:

- Registers in the (`coproc == 0b1111`) encoding space, that provide control and status information for the PE in Non-debug state.
- Registers in the (`coproc == 0b1110`) encoding space, including:
  - Debug registers.
  - Trace registers.
  - Legacy execution environment registers.

Organization of registers in the (coproc == `0b1110`) encoding space summarizes the registers in the (`coproc == 0b1110`) encoding space, and indicates where these registers are described, either in this manual or in other architecture specifications.

Organization of registers in the (coproc == `0b1111`) encoding space summarizes the registers in the (`coproc == 0b1111`) encoding space, and indicates where in this manual these registers are described.

Note

Many implementations include other interfaces to some System registers, for example a memory-mapped interface to some debug System registers. These are described in the appropriate sections of this manual.

G5.15.1 Classification of System registers

Features provided by EL3 and EL2 integrate with many features of the architecture. Therefore, the descriptions of the individual System registers include information about how these Exception levels affect the register. This section:

- Summarizes how EL3 and EL2 affect the implementation of the System registers, and the classification of those registers.
- Summarizes how EL3 controls access to the System registers.
- Describes an EL3 signal that can control access to some registers in the (`coproc == 0b1111`) encoding space.

It contains the following subsections:

- Banked System registers.
- Restricted access System registers.
- Configurable access System registers.
- EL2-mode System registers.
- Common System registers.
- Access to registers from Monitor mode.
- The CP15SDISABLE and CP15SDISABLE2 input signals.

Note

EL3 defines the register classifications of Banked, Restricted access, Configurable, and Common. EL2 defines the EL2-mode classification.

It is IMPLEMENTATION DEFINED whether each IMPLEMENTATION DEFINED register is Banked, Restricted access, Configurable, EL2-mode, or Common.

G5.15.1.1 Banked System registers

In an implementation that includes EL3 using AArch32, some System registers are banked. Banked System registers have two copies, one Secure and one Non-secure. The SCR.NS bit selects the Secure or Non-secure instance of the register.

A Banked System register can contain a mixture of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Fields that are banked.
- Fields that are read-only in Non-secure PL1 or EL2 modes but read/write in the Secure state.

The System Control Register SCTLR is an example of a register of that contains this mixture of fields.

The Secure copies of the Banked System registers are sometimes referred to as the Secure Banked System registers. The Non-secure copies of the Banked System registers are sometimes referred to as the Non-secure Banked System registers.

G5.15.1.2 Restricted access System registers

In an implementation that includes EL3, some System registers are present only in Secure state. These are called *Restricted access* registers, and their read/write access permissions are:

- In Non-secure state, software cannot modify Restricted access registers.
- For the NSACR, in Non-secure state:
  - Software running at PL1 or higher can read the register.
  - Unprivileged software, meaning software running at EL0, cannot read the register.

  This means that Non-secure software running at PL1 or higher can read the access permissions for System registers that have Configurable access.

  If EL3 is using AArch64, then any read of the NSACR from Non-secure EL2 using AArch32, or Non-secure EL1 using AArch32, returns the value `0x0000_0C00`.
- For all other Restricted access registers, Non-secure software cannot read the register.

In an implementation that does not include EL3:

- SDER is implemented only in Secure state.
- Any read of the NSACR returns the value `0x0000_0C00`.
- All other accesses to Restricted access System registers are UNDEFINED.

G5.15.1.3 Configurable access System registers

Secure software can configure the access to some System registers. These registers are called Configurable access registers, and the control can be:

- A bit in the control register determines whether the register is:
  - Accessible from Secure state only.
  - Accessible from both Secure and Non-secure states.
- A bit in the control register changes the accessibility of a register bit or field. For example, setting a bit in the control register might mean that an RW field behaves as RAZ/WI when accessed from Non-secure state.

Bits in the NSACR control access.

In an AArch32 implementation that includes EL3:

- There are no Configurable access System registers in the (`coproc` == `0b1110`) encoding space.
- The only required Configurable access register in the (`coproc` == `0b1111`) encoding space is the CPACR.
-
  - Floating-point Status and Control Register, FPSCR
  - Floating-point Exception register, FPEXC.
  - Floating-point System ID register, FPSID.
  - Media and VFP Feature Register 0, MVFR0.
  - Media and VFP Feature Register 1, MVFR1.
  - Media and VFP Feature Register 2, MVFR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.15.1.4 EL2-mode System registers

In an implementation that includes EL2, if EL2 can use AArch32, the implementation provides a number of registers for use in the EL2 mode, Hyp mode. As with other System register encodings, some of these register encodings provide write-only operations. When the implementation includes EL3 and EL3 is using AArch32, these registers are also accessible from Monitor mode when the value of SCR.NS is 1.

The following subsections describe the EL2-mode registers:

- Hyp mode read/write registers in the (coproc == `0b1111`) encoding space.
- Hyp mode encodings for shared (coproc == `0b1111`) System registers.
- Hyp mode (coproc == `0b1111`) write-only System instructions.

There are no EL2-mode registers in the (`coproc` == `0b1110`) encoding space.

G5.15.1.4.1 Hyp mode read/write registers in the (`coproc` == `0b1111`) encoding space

These registers are implemented only in Non-secure state, and in Non-secure state they are accessible only from Hyp mode.

Except for accesses to CNTVOFF in an implementation that includes EL3 but not EL2, the behavior of accesses to these registers is as follows:

- In Secure state, the registers can be accessed from EL3 when SCR.NS is set to 1, see Access to registers from Monitor mode.
- The following accesses are UNDEFINED:
  - Accesses from Non-secure PL1 modes.
  - Accesses in Secure state when SCR.NS is set to 0.

In an implementation that includes EL3 but not EL2, the behavior of accesses to CNTVOFF is as follows:

- Any access from Secure Monitor mode is CONSTRAINED UNPREDICTABLE, regardless of the value of SCR.NS. The CONSTRAINED UNPREDICTABLE behavior is that the access is UNDEFINED, see Unallocated System register access instructions.
- All other accesses are UNDEFINED.

Note

Except for CNTVOFF, the Hyp mode registers are part of EL2, meaning they are implemented only if the implementation includes EL2. However, conceptually, CNTVOFF is part of any implementation of the Generic Timer, see The virtual offset register. This means the behavior of CNTVOFF in an implementation that does not include EL2 is not covered by the general definition of the behavior of the Hyp mode (`coproc` == `0b1111`) read/write registers.

G5.15.1.4.2 Hyp mode encodings for shared (`coproc` == `0b1111`) System registers

Some Hyp mode registers share the Secure instance of an existing banked register. In this case, the implementation includes an encoding for the register that is accessible only in Hyp mode, or in Monitor mode when SCR.NS is set to 1.

For these registers, the following accesses are UNDEFINED:

- Accesses from Non-secure PL1 modes.
- Accesses in Secure state when SCR.NS is set to 0.

In Monitor mode, the Secure copies of these registers can be accessed either:

- Using the DFAR or IFAR encoding with SCR.NS set to 0.
- Using the HDFAR or HIFAR encoding with SCR.NS set to 1.

However, between accessing a register using one alias and accessing the register using the other alias, a Context Synchronization event is required to ensure the ordering of the accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.15.1.4.3 Hyp mode (coproc == 0b1111) write-only System instructions

Architecturally, these encodings are an extension of the banked register encodings described in Banked System registers, where:

- The implementation does not implement the operation in Secure state.
- In Non-secure state, the operation is accessible only at EL2, that is, only from Hyp mode.

In Secure state:

- These instructions can be accessed from Monitor mode regardless of the value of SCR.NS, see Access to registers from Monitor mode.
- Accesses to these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure mode other than Monitor mode.

Accesses to these instructions are UNDEFINED if accessed from a Non-secure PL1 mode.

G5.15.1.5 Common System registers

Some System registers and operations are common to the Secure and Non-secure Security states. These are described as the *Common access* registers, or simply as the *Common* registers. These registers include:

- Read-only registers that hold configuration information.
- Register encodings used for various memory system operations, rather than to access registers.
- The ISR.
- All System registers in the (coproc == 0b1110) encoding space.

G5.15.1.6 Secure System registers for the (coproc == 0b1111) encoding space

The Secure System registers in the (coproc == 0b1111) encoding space comprise:

- The Secure copies of the Banked System registers in the (coproc == 0b1111) encoding space.
- The Restricted access System registers in the (coproc == 0b1111) encoding space.
- The Configurable access System registers in the (coproc == 0b1111) encoding space that are configured to be accessible only from Secure state.

In an implementation that includes EL3, the Non-secure System registers are the System registers other than the Secure System registers.

G5.15.1.7 Access to registers from Monitor mode

When the PE is in Monitor mode, the PE is in Secure state regardless of the value of the SCR.NS bit. In Monitor mode, the SCR.NS bit determines whether, for System registers in the (coproc == 0b1111) encoding space, valid uses of the MRC, MCR, MRRC, and MCRR instructions access the Secure Banked System registers or the Non-secure Banked System registers. That is, when:

NS == 0 Common, Restricted access, and Secure Banked System registers are accessed by MRC, MCR, MRRC, and MCRR instructions that target the (coproc == 0b1111) encoding space.

If the implementation includes EL2, the registers listed in Hyp mode read/write registers in the (coproc == 0b1111) encoding space and Hyp mode encodings for shared (coproc == 0b1111) System registers are not accessible, and any attempt to access them generates an Undefined Instruction exception.

Note

The operations listed in Hyp mode (coproc == 0b1111) write-only System instructions are accessible in Monitor mode regardless of the value of SCR.NS.

System instructions in the (coproc == 0b1111) encoding space use the Security state to determine all resources used, that is, all operations performed by these instructions are performed in Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS == 1 Common, Restricted access and Non-secure Banked System registers are accessed by `MRC`, `MCR`, `MRRC`, and `MCRR` instructions that target the (`coproc == 0b1111`) encoding space.

If the implementation includes EL2, all the registers and operations listed in the subsections of EL2-mode System registers are accessible, using the `MRC`, `MCR`, `MRRC`, or `MCRR` instructions required to access them from Hyp mode.

System instructions in the (`coproc == 0b1111`) encoding space use the Security state to determine all resources used, that is, all operations by these instructions are performed in Secure state.

The Security state determines whether the Secure or Non-secure banked registers determine the control state.

Note

Where the contents of a register select the value accessed by an `MRC` or `MCR` access to a different register, then the register that is used for selection is being used as control state. For example, CSSELR selects the current Cache Size Identification Register, and therefore CSSELR is used as control state. Therefore, in Monitor mode:

- SCR.NS determines whether the Secure or Non-secure CSSELR is accessible.
- Because the PE is in Secure state, the Secure CSSELR selects the current Cache Size Identification Register.

From Armv8.3, it is possible to have multiple Cache Size Identification Registers. For more details, see Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2.

G5.15.1.8 The CP15SDISABLE and CP15SDISABLE2 input signals

When EL3 is using AArch32, it provides an input signal, `CP15SDISABLE`, that disables write access to some of the Secure registers when asserted HIGH. The `CP15SDISABLE` signal has no effect on:

- Register accesses from AArch64 state.
- Register accesses from Secure EL1 when EL3 is using AArch64 and EL1 is using AArch32.

Note

When EL3 is using AArch32, the interaction between `CP15SDISABLE` and any IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.

On a Warm reset by the external system that resets the PE into EL3 using AArch32, the `CP15SDISABLE` input signal must be taken LOW. This permits the Reset code to set up the configuration of EL3 features. When the input is asserted HIGH, any attempt to write to the Secure registers that are affected by `CP15SDISABLE` results in an Undefined Instruction exception.

The `CP15SDISABLE` input does not affect reading Secure registers, or reading or writing Non-secure registers. It is IMPLEMENTATION DEFINED how the input is changed and when changes to this input are reflected in the PE, and an implementation might not provide any mechanism for driving the `CP15SDISABLE` input HIGH. However, in an implementation in which the `CP15SDISABLE` input can be driven HIGH, changes in the state of `CP15SDISABLE` must be reflected as quickly as possible. Any change must occur before completion of an Instruction Synchronization Barrier operation, issued after the change, is visible to the PE with respect to instruction execution boundaries. Software must perform an Instruction Synchronization Barrier operation meeting the above conditions to ensure all subsequent instructions are affected by the change to `CP15SDISABLE`.

When EL3 is using AArch32, use of `CP15SDISABLE` means key Secure features that are accessible only at PL1 can be locked in a known state. This provides an additional level of overall system security. Arm expects control of `CP15SDISABLE` to reside in the system, in a block dedicated to security.

When FEAT_CP15SDISABLE2 is implemented and EL3 is using AArch32, EL3 provides a second input signal, `CP15SDISABLE2`. `CP15SDISABLE2` has all of the properties of `CP15SDISABLE` described above. The difference between `CP15SDISABLE` and `CP15SDISABLE2` is only in the set of registers each signal affects.

Information on whether a given register is affected by `CP15SDISABLE`, or `CP15SDISABLE2` when it is implemented, can be found in the access pseudocode for that register, as described in AArch32 System Register Descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G5.16 Functional grouping of VMSAv8-32 System registers

This section describes how the System registers in an VMSAv8-32 implementation divide into functional groups. The functional groups of AArch32 registers are:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors registers.
- Activity monitors registers.
- Debug registers.
- RAS registers.
- Generic timer registers.
- Cache maintenance System instructions.
- Address translation System instructions.
- TLB maintenance System instructions.
- Base system registers.
- Legacy feature registers and System instructions.

For a list of these functional groups and the registers in each group, see Functional index of AArch32 registers and System instructions.

AArch32 System Register Descriptions describes each of these registers.

Note

- The functional groups defined in this section mainly consist of the VMSAv8-32 System registers, but include some additional System registers.
- Some registers belong to more than one functional group.

For other related information, see:

- The AArch32 System register interface for general information about the access to the AArch32 System registers, including the main register access instructions `MRC` and `MCR`.
- About the System registers for VMSAv8-32.
- Organization of registers in the (coproc == `0b1110`) encoding space.
- Organization of registers in the (coproc == `0b1111`) encoding space.
- About the AArch32 System registers.

The register descriptions in AArch32 System Register Descriptions, assume you are familiar with these functional groups, and use conventions and other information from them without any explanation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G6
The Generic Timer in AArch32 state

This chapter describes the implementation of the Arm Generic Timer as an extension to an Armv8 AArch32 implementation. It includes an overview of the AArch32 System register interface to an Arm Generic Timer.

It contains the following sections:

- About the Generic Timer in AArch32 state.
- The AArch32 view of the Generic Timer.

The Generic Timer in AArch64 state describes the AArch64 view of the Generic Timer, including additional timers that can be implemented in AArch64 state, and System Level Implementation of the Generic Timer describes the system level implementation of the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G6.1 About the Generic Timer in AArch32 state

Figure G6-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

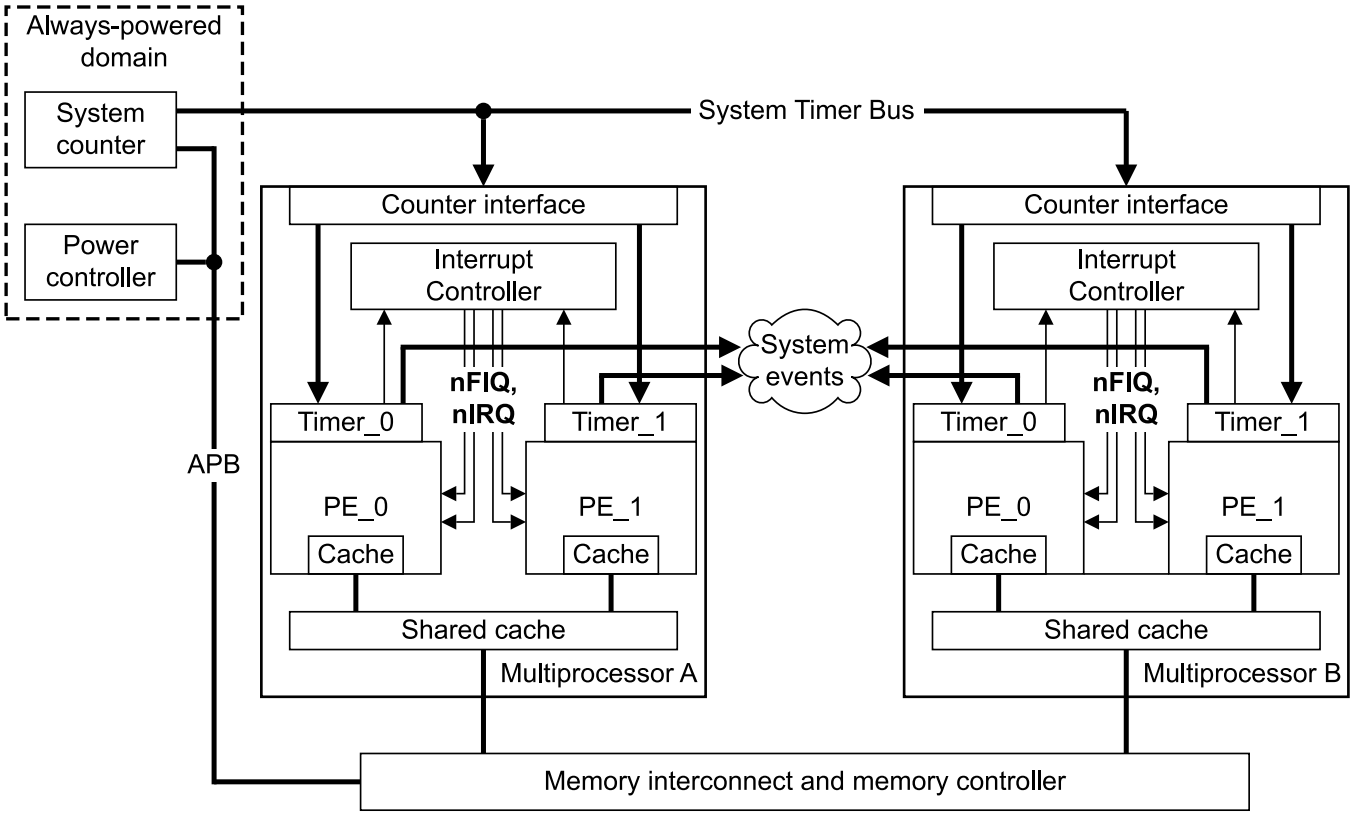

Figure G6-1 Generic Timer example

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

  Note

  The Generic Timer can also provide other components at a system level, but Figure G6-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure G6-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch32 state. The Generic Timer in AArch64 state describes access to this component from AArch64 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The reset requirements of Generic Timer registers are more strict when they are accessed from AArch32 state than when they are accessed from AArch64 state.

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure G6-1.
- Optionally, can provide timer components for use at a system level.

System Level Implementation of the Generic Timer describes this memory-mapped component.

G6.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see The system counter. Because this must be implemented at the system level, it is accessed through The system level memory-mapped implementation of the Generic Timer. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, that gives access to the count value of the system counter. When FEAT_ECV is implemented, EL2 is using AArch64, and EL2 is implemented and enabled in the current Security state, the CNTPOFF_EL2 register allows offsetting of AArch32 physical timers and counters.
- A virtual counter, that gives access to virtual time. In AArch32 state, the CNTVOFF register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch32 state, the timers that are accessible from AArch32 state are:
  - A Secure PL1 physical timer.
  - A Non-secure EL1 physical timer.
  - A Non-secure EL2 physical timer.
  - An EL1 virtual timer.
  - A Non-secure EL2 virtual timer.
  - A Secure EL2 virtual timer.
  - A Secure EL2 physical timer.

  The Non-secure EL2 virtual timer is available when FEAT_VHE is implemented.
  The Secure EL2 timers are available when FEAT_SEL2 is implemented, but are only accessible in AArch32 state if using EL0, when EL0 is using AArch32, Secure EL2 is using AArch64, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

  Note

  The Secure PL1 physical timer uses the Secure banked instances of the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers, and the Non-secure EL1 physical timer uses the Non-secure instances of the same registers.

The AArch32 view of the Generic Timer describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into *frames*. The Generic Timer architecture defines the offset of each

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally. provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0 - 7.

System Level Implementation of the Generic Timer describes these components.

G6.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width

From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.

From Armv8.6, must be 64 bits wide.

Effective frequency

This indicates the amount of time each unit of the counter represents. For example:

- An effective frequency of 50MHz means each unit of the counter represents 20ns.
- An effective frequency of 1GHz means each unit of the counter represents 1ns.
- The effective frequency is indicated in the CNTFRQ register.

From Armv8.0 to Armv8.5 inclusive, the effective frequency is a fixed value, typically in the range 1-50MHz.

From Armv8.6, the effective frequency is a fixed value of 1GHz.

The counter might not be driven by a clock running at the effective frequency. For example, a counter with an effective frequency of 1GHz might have an actual clock running at 125MHz but the counter increments by 8 on each tick of the clock.

Roll-over

Roll-over time of not less than 40 years.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Counter resolution

The counter resolution is a representation of how frequently the counter is updated.

For example, a counter might have an effective frequency of 1GHz, but the actual clock runs at 125MHz and therefore the counter resolution is 125Mhz.

From Armv8.6, Arm recommends the counter resolution is not less than 50MHz in normal running operation.

The counter resolution might vary over time, to accommodate different operating modes of the system such as lower power modes. Such variation might be invisible to the programmer and be system specific, or might be controllable by the programmer through the frequency modes table.

Accuracy

Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.

Use of different counter resolutions must not affect the implemented accuracy.

Start-up

Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can operate with a lower counter resolution by incrementing by larger amounts at a lower actual clock frequency. For example, a system counter with an effective frequency of 10MHz might either increment.

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the actual clock frequency, to reduce power consumption.

In this case, the counter must support transitions between higher resolution operation and lower resolution operation, without any impact on the required accuracy of the counter.

From Armv8.6 the counter operates at a fixed effective frequency of 1GHz.

The CNTFRQ register is intended to hold a copy of the effective frequency to allow fast reference to this frequency by software running on the PE. For more information, see Initializing and reading the system effective frequency.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. Gray count scheme for timer distribution scheme gives more information about this possibility.

G6.1.2.1 Initializing and reading the system effective frequency

The CNTFRQ register must be programmed to the effective frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system effective frequency to the CNTFRQ register. Only software executing at the highest implemented Exception level can write to CNTFRQ.

Note

The CNTFRQ register is UNKNOWN at reset, and therefore the effective frequency must be set as part of the system boot process.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Software can read the CNTFRQ register, to determine the effective frequency, in the following states and modes:

- Hyp mode.
- Secure PL1 modes and Non-secure EL1 modes.
- When CNTKCTL.{PL0PCTEN, PL0VCTEN} is not {0, 0}, Secure and Non-secure EL0 modes.

G6.1.2.2 Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the counter resolution.
- Enabling Halt-on-debug, which a debugger can then use to suspend counting.

For descriptions of these controls, see System Level Implementation of the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G6.2 The AArch32 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch32 state:

- The physical counter.
- The virtual counter.
- Event streams.
- Timers.

G6.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT register holds the current physical counter value. When FEAT_ECV is implemented and EL2 is executing in AArch64 state, the CNTPOFF_EL2 register holds the optional physical offset that can be applied to EL0 and EL1 whether EL0 and EL1 are using AArch64 state or AArch32 state. For more information, see The physical offset register.

Reads of CNTPCT can occur speculatively and out of order relative to other instructions executed on the same PE.

G6.2.1.1 The self-synchronized view of the physical counter

When FEAT_ECV is implemented, an alternative way to read the physical counter is supported. The CNTPCTSS register is a non-speculative view of the physical counter, as seen from the Exception level that CNTPCTSS is read from.

Access to the CNTPCTSS are subject to the same traps as accesses to the CNTPCT.

Reads of CNTPCT occur in program order relative to reads of CNTPCT or CNTPCTSS.

Reads of CNTPCTSS occur in program order relative to reads of CNTPCT or CNTPCTSS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G6-1 Ensuring reads of the physical counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTPCT must be read, an `ISB` is used to ensure that the read of CNTPCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                  ; polling for some communication to indicate a requirement to
                      ; read the timer
    LDR R1, [R2]
    CMP R1, #1        ; has had the value 1 written to it
    BNE loop
    ISB               ; without this the CNTPCT could be read before the memory location in [R2]
    MRC R1, CNTPCT
```

When FEAT_ECV is implemented, an access to CNTPCTSS can be used in place of the CNTPCT which, because it cannot be accessed speculatively, allows the `ISB` to be removed. This means that the following code sequence can be used:

```
loop                    ; polling for some communication to indicate a requirement to
                        ; read the timer
    LDR R1, [R2]
    CMP R1, #1          ; has had the value 1 written to it
    BNE loop
    MRC R1, CNTPCTSS
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
  ...                 ; earlier loads and stores
    DSB               ; completes earlier loads and stores
    ISB               ; without this the CNTPCT could be read before the completion
                      ; of the earlier loads and stores
    MRC R1, CNTPCT
```

Or, if FEAT_ECV is implemented:

```
  ...                   ; earlier loads and stores
    DSB                 ; completes earlier loads and stores
    MRC R1, CNTPCTSS
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers and memory dependency from the read of the physical counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G6-2 Ensuring reads of the physical counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

```
 DSB
   ISB
   MRC Rn, CNTPCT{SS}    ; either view of the physical counter has the same effect in
                         ; this example
```

or

```
 DSB
   ISB
   CBZ Ra, next
next
   ISB                     ; this ISB is not needed if the MRC is accessing CNTPCT{SS}
   MRC Rn, CNTPCT{SS}
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
 MRC Rn, CNTPCT{SS}          ; either view of the physical counter has the same effect
                             ; in this example
   ISB
   LDR Ra, [Rb]              ; this load will be executed after the timer has been read
```

or

```
 MRC Rn, CNTPCT{SS}            ; either view of the physical counter has the same effect
                               ; in this example
   EOR Rm, Rn, Rn
   LDR Ra, [Rb]                ; this load will be executed after the counter has been read
```

G6.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing in a Non-secure EL1 or EL0 mode, the virtual offset value relates to the current virtual machine.

The CNTVOFF register contains the virtual offset, see The virtual offset register.

The CNTVCT register holds the current virtual counter value.

Reads of CNTVCT can occur speculatively and out of order relative to other instructions executed on the same PE.

G6.2.2.1 The self-synchronized view of the virtual counter

When FEAT_ECV is implemented, an alternative way to read the virtual counter is supported. The CNTVCTSS register is a non-speculative view of the virtual counter, as seen from the Exception level that CNTVCTSS is read from.

Accesses to the CNTVCTSS are subject to the same traps as accesses to the CNTVCT.

Reads of CNTVCT occur in program order relative to reads of CNTVCT or CNTVCTSS.

Reads of CNTVCTSS occur in program order relative to reads of CNTVCT or CNTVCTSS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G6-3 Ensuring reads of virtual counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTVCT must be read, an `ISB` is used to ensure that the read of CNTVCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                    ; polling for some communication to indicate a requirement to
                        ; read the timer
    LDR R1, [R2]
    CMP R1, #1          ; has had the value 1 written to it
    BNE loop
    ISB                 ; without this the CNTVCT could be read before the memory
                        ; location in [R2]
    MRC R1, CNTVCT
```

When FEAT_ECV is implemented, an access to CNTVCTSS can be used in place of the CNTVCT, which, because it cannot be accessed speculatively, allows the `ISB` to be removed. This means that the following code sequence can be used:

```
loop                    ; polling for some communication to indicate a requirement to
                        ; read the timer
    LDR R1, [R2]
    CMP R1, #1          ; has had the value 1 written to it
    BNE loop
    MRC R1, CNTVCTSS
```

Similarly where a read of the virtual counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following two sequences can be used:

```
  ...                   ; earlier loads and stores
    DSB                 ; completes earlier loads and stores
    ISB                 ; without this the CNTVCT could be read before the completion
                        ; of the earlier loads and stores
    MRC R1, CNTVCT
```

Or, if FEAT_ECV is implemented:

```
  ...                     ; earlier loads and stores
    DSB                   ; completes earlier loads and stores
    MRC R1, CNTVCTSS
```

Neither view of the virtual counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers and memory dependency from the read of the virtual counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G6-4 Ensuring reads of virtual counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

```
  DSB
    ISB
    MRC Rn, CNTVCT{SS}    ; either view of the virtual counter has the same effect in
                          ; this example
```

or

```
  DSB
    ISB
    CBZ Ra, next
next
    ISB                   ; this ISB is not needed if the MRS is accessing CNTVCT{SS}
    MRC Rn, CNTVCT{SS}
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
  MRC Rn, CNTVCT{SS}    ; either view of the virtual counter has the same effect in
                        ; this example
    ISB
    LDR Ra, [Rb]        ; this load will be executed after the timer has been read
```

or

```
  MRC Rn, CNTVCT{SS}    ; either view of the virtual counter has the same effect in
                        ; this example
    EOR Rm, Rn, Rn
    LDR Ra, [Rb]        ; this load will be executed after the counter has been read
```

G6.2.2.2 The virtual offset register

The virtual counter is a counter that has a *virtual offset* relative to the physical counter as viewed from EL2 and EL3. This virtual offset is held in the register CNTVOFF. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented and enabled, then the virtual counter uses a fixed virtual offset of zero.

G6.2.3 Event streams

An implementation that includes the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wakeup events as part of the mechanism described in Wait for Event.

Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The CNTKCTL.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the virtual counter.

In all implementations the CNTHCTL.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:

- EVNTI selects the counter bit that triggers the event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_ECV is not implemented, EVNTI selects between bits[0:15].
- If FEAT_ECV is implemented, EVNTIS selects whether ENVTI selects between bits[0:15] or bits[8:23].
- EVNTDIR selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

Note

If the event stream is configured to produce events from the low order bits of the counter when the effective frequency is very high (for example 1GHz), then the practical update rate of the counter might mean that the event stream is not generated as the low order bit might not change. Software can rely on an event stream rate of at least 1MHz in normal operation.

The pseudocode descriptions of the operation of an event stream are `SetEventRegister`, `TestEventCNTV`, and `TestEventCNTP`.

G6.2.4 Timers

In an implementation of the Generic Timer that includes EL3 the following timers are accessible from AArch32 state, provided the appropriate Exception level can use AArch32:

- A Non-secure EL1 physical timer. A Non-secure EL1 control determines whether this register is accessible from Non-secure EL0.
- A Secure PL1 physical timer. This timer is accessible from EL3 when EL3 is using AArch32.

  Note

  When EL3 is using AArch64, the AArch32 EL1 timers are not banked between Secure and Non-secure state.

  A Secure PL1 control determines whether this register is accessible from Secure EL0.
- A Non-secure EL2 physical timer, accessible from Non-secure EL2, or EL3 when SCR.NS is set to 1.
- An EL1 virtual timer.
- When FEAT_VHE is implemented, a Non-secure EL2 virtual timer.
- When FEAT_SEL2 is implemented, a Secure EL2 physical timer.
- When FEAT_SEL2 is implemented, a Secure EL2 virtual timer.

Note

The Secure EL2 timers are accessible in AArch32 state if using EL0, when EL0 is using AArch32 state, Secure EL2 is using AArch64, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:

- Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.
- Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.
- Has, in addition, a 32-bit Control register.

In all implementations, the AArch32 System registers for the EL1 (or PL1) physical timer are banked, to provide the Secure and Non-secure implementations of the timer. Table G6-1 shows the physical timer registers and Table G6-2 show the virtual timer registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G6-1 Physical timer registers summary for the Generic Timer

Timer register[a]	Secure PL1 or Non-secure EL1 physical timer	Non-secure EL2 physical timer	Secure EL2 physical timer[b]
CV	CNTP_CVAL[c]	CNTHP_CVAL	CNTHPS_CVAL
TV	CNTP_TVAL[c]	CNTHP_TVAL	CNTHPS_TVAL
Control	CNTP_CTL[c]	CNTHP_CTL	CNTHPS_CTL

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Only present when the implementation implements FEAT_SEL2.

c. In AArch32 state, these registers are banked to provide the Non-secure EL1 physical timer and the Secure PL1 physical timer.

Table G6-2 Virtual timer register summary for the Generic Timer

Timer register[a]	EL1 virtual timer	Non-secure EL2 virtual timer[b]	Secure EL2 virtual timer[c]
CV	CNTV_CVAL	CNTHV_CVAL	CNTHVS_CVAL
TV	CNTV_TVAL	CNTHV_TVAL	CNTHVS_TVAL
Control	CNTV_CTL	CNTHV_CTL	CNTHVS_CTL

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Only when the implementation includes FEAT_VHE.

c. Only present when the implementation includes FEAT_SEL2.

G6.2.4.1 Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register. When the timer condition is met an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, CNTHPS_CTL, CNTV_CTL, CNTHV_CTL, or CNTHVS_CTL. For CNTP_CTL, the interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch64 register CNTP_CTL_EL0.

The operation of this view of a timer is:

```
TimerConditionMet = (((Counter[63:0] - Offset[63:0])[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`TimerConditionMet` Is TRUE if the timer condition for this counter is met, and FALSE otherwise.

`Counter` The physical counter value, that can be read from the CNTPCT register.

`Offset` For the EL1 physical timer, if ID_AA64MMFR0_EL1.ECV is `0b10`, EL2 is using AArch64 and is implemented and enabled in the current Security state, and CNTHCTL_EL2.ECV is 1, then the offset value is held in the CNTPOFF_EL2. Otherwise the offset value for the EL1 physical timer is zero.

For the EL1 virtual timer, the offset value is held in the CNTVOFF register.

For the EL2 physical and virtual timers, the offset value is zero.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL, CNTHP_CVAL, CNTHPS_CVAL, CNTV_CVAL, CNTHV_CVAL, or CNTHVS_CVAL.

In this view of a timer, `Counter`, `Offset`, and `CompareValue` are all 64-bit unsigned values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.
Software can observe the counter value by the offset in some situations by reading CNTVCT.

G6.2.4.2 Operation of the TimerValue views of the timers

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, CNTHPS_CTL, CNTV_CTL, CNTHV_CTL, or CNTHVS_CTL.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads

```
TimerValue = (CompareValue - (Counter - Offset))[31:0]
```

Writes

```
CompareValue = ((Counter - Offset)[63:0] + SignExtend{64}(TimerValue))
```

Where the arguments other than `TimerValue` have the definitions used in Operation of the CompareValue views of the timers, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL, CNTHP_TVAL, CNTHPS_TVAL, CNTV_TVAL, CNTHV_TVAL, or CNTHVS_TVAL.

In this view of a timer, values are signed, in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

Note

- Operation of the CompareValue views of the timers gives a strict definition of `TimerConditionMet`. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x80000000`, the TimerValue view can be used as giving an effect equivalent to:

  `TimerConditionMet = (TimerValue ≤ 0)`

- Programming TimerValue to a negative number with magnitude greater than (Counter - Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays meeting the timer condition for an extremely long period of time.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G7
AArch32 System Register Encoding

This chapter describes the AArch32 System register encoding space. It contains the following sections:

- The AArch32 System register encoding space.
- Organization of registers in the (coproc == `0b1110`) encoding space.
- Organization of registers in the (coproc == `0b1111`) encoding space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G7.1 The AArch32 System register encoding space

The T32 and A32 instruction sets includes instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- The cache, branch predictor, and TLB maintenance instructions, and address translation instructions.

The AArch32 System register interface describes the instructions that provide access to these registers and instructions. AArch32 System Register Descriptions describes these registers and encodings.

When accessing 32-bit registers, or executing these instructions, entries in the encoding space are characterized by the parameter set {`coproc`, `CRn`, `opc1`, `CRm`, `opc2`}. This encoding space is defined only for the `coproc` values `0b1110` and `0b1111`.

Note

- When accessing 64-bit registers entries in the encoding space are characterized by the parameter set {`coproc`, `CRm`, `opc1`}, for the `coproc` values `0b1110` and `0b1111`. A `CRm` value in this parameter set is equivalent to a `CRn` value in the parameter set for accessing 32-bit registers.
- Background to the System register interface gives more information about this encoding model.

The following describe this encoding space:

- Organization of registers in the (coproc == `0b1110`) encoding space.
- Organization of registers in the (coproc == `0b1111`) encoding space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G7.2 Organization of registers in the (coproc == 0b1110) encoding space

The System registers in the (coproc == 0b1110) encoding space provide a number of distinct control functions, covering:

- Debug.
- Trace.
- Execution environment control, for identification of the trivial Jazelle implementation.

Because these functions are distinct, the descriptions of these registers are distributed, as follows:

- In this manual, Debug System registers describes the Debug System registers.
- The *Embedded Trace Macrocell Architecture Specification* describes the Trace registers.

This section summarizes the allocation of the System registers in the (coproc == 0b1110) encoding space between these different functions, and the register encodings in this space that are reserved.

The 32-bit System register encodings are classified by the {opc1, CRn, opc2, CRm} values required to access them using an MCR or an MRC instruction. The 64-bit System register encodings are classified by the {opc1, CRm} values required to access them using an MCRR or an MRRC instruction. For the registers in the (coproc == 0b1110) encoding space, the opc1 value determines the primary allocation of these registers, as follows:

opc1==0 Debug System registers.

opc1==1 Trace registers.

opc1==7 Jazelle registers. Jazelle registers are implemented as required for a trivial Jazelle implementation.

Other opc1 values

Reserved.

Note

Primary allocation of (coproc == 0b1110) register function by opc1 value differs from the allocation of (coproc == 0b1111) registers, where primary allocation is by CRn value for 32-bit register accesses, or CRm value for 64-bit register accesses.

For the Debug System and Jazelle registers, Table G7-1 defines:

- The {opc1, CRn, opc2, CRm} values used for accessing the 32-bit registers using the MRC and MCR instructions.
- The {opc1, CRm} values used for accessing the 64-bit register using the MRRC instruction.

Some Debug System registers can also be accessed using the LDC and STC instructions. Table G7-1 defines the CRn values used for accessing the registers using these instructions.

Note

The only permitted uses of the LDC and STC instructions are:

- An LDC access to load data from memory to DBGDTRTXint.
- An STC access to store data to memory from DBGDTRRXint.

In the LDC and STC syntax descriptions in this Manual, the required coproc value of p14 and CRn value of c5 are given explicitly.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G7.2.1 Register access instruction arguments, (coproc == 0b1110) registers

Table G7-1 shows the MCR, MRC, and MRRC instruction arguments required for accesses to each register that can be visible in the System register interface in the (coproc == 0b1110) encoding.

Table G7-1

coproc	opc1	CRn	CRm	opc2	Access	Mnemonic	Register
1110	000	0000	0000	000	RO	DBGDIDR	DBGDIDR
1110	000	0000	0000	010	RW	DBGDTRRXext	DBGDTRRXext
1110	000	0000	0001	000	RO	DBGDSCRint	DBGDSCRint
1110	000	0000	0010	000	RW	DBGDCCINT	DBGDCCINT
1110	000	0000	0010	010	RW	DBGDSCRext	DBGDSCRext
1110	000	0000	0011	010	RW	DBGDTRTXext	DBGDTRTXext
1110	000	0000	0101	000	•	DBGDTRRXint	•
1110	000	0000	0101	000	•	DBGDTRTXint	•
1110	000	0000	0110	000	RW	DBGWFAR	DBGWFAR
1110	000	0000	0110	010	RW	DBGOSECCR	DBGOSECCR
1110	000	0000	0111	000	RW	DBGVCR	DBGVCR
1110	000	0000	m[3:0]	100	RW	DBGBVR<n>	DBGBVR[]
1110	000	0000	m[3:0]	101	RW	DBGBCR<n>	DBGBCR[]
1110	000	0000	m[3:0]	110	RW	DBGWVR<n>	DBGWVR[]
1110	000	0000	m[3:0]	111	RW	DBGWCR<n>	DBGWCR[]
1110	000	0001	0000	000	RO	DBGDRAR	DBGDRAR
1110	000	0001	0000	100	WO	DBGOSLAR	DBGOSLAR
1110	000	0001	0001	100	RO	DBGOSLSR	DBGOSLSR
1110	000	0001	0011	100	RW	DBGOSDLR	DBGOSDLR
1110	000	0001	0100	100	RW	DBGPRCR	DBGPRCR
1110	000	0001	m[3:0]	001	RW	DBGBXVR<n>	DBGBXVR[]
1110	000	0010	0000	000	RO	DBGDSAR	DBGDSAR
1110	000	0111	0000	111	RO	DBGDEVID2	DBGDEVID2
1110	000	0111	0001	111	RO	DBGDEVID1	DBGDEVID1
1110	000	0111	0010	111	RO	DBGDEVID	DBGDEVID
1110	000	0111	1000	110	RW	DBGCLAIMSET	DBGCLAIMSET
1110	000	0111	1001	110	RW	DBGCLAIMCLR	DBGCLAIMCLR
1110	000	0111	1110	110	RO	DBGAUTHSTATUS	DBGAUTHSTATUS
1110	111	0000	0000	000	RO	JIDR	JIDR
1110	111	0001	0000	000	RO	JOSCR	JOSCR
1110	111	0010	0000	000	RO	JMCR	JMCR
1110	0000	0001	•	DBGDRAR	•		
1110	0000	0010	•	DBGDSAR	•		

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G7-2 shows the LDC and STC instruction arguments required for accesses to the Debug registers that can be accessed using these instructions.

Table G7-2 Mapping of LDC and STC instruction arguments to System registers

Name	CRn	Instruction	Description
DBGDTRTXint	c5	LDC	Debug Data Transfer Register, Transmit, Internal View
DBGDTRRXint	c5	STC	Debug Data Transfer Register, Receive, Internal View

Note

In the instruction syntax descriptions for the LDC and STC instructions, the required coproc and CRn values are given explicitly as coproc == p14, CRn == c5.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G7.3 Organization of registers in the (coproc == 0b1111) encoding space

For 32-bit accesses to the System registers in the (coproc == 0b1111) encoding space, the ordered set of parameters {CRn, opc1, CRm, opc2} determine the register order. Within this ordering, the CRn value originally provided a functional grouping of these registers. As the number of System registers has increased this ordering has become less appropriate.

This document now:

- Groups the System registers in the (coproc == 0b1111) encoding space by functional group, see Functional index of AArch32 registers and System instructions.
- Describes all of the System registers for the AArch32 VMSA, in AArch32 System Register Descriptions.
- Gives additional information about the organization of the AArch32 VMSA System registers in the (coproc == 0b1111) encoding space, in the remainder of this section.

Note

Not all System registers introduced by architectural extensions are described in AArch32 System Register Descriptions. For information about the System registers introduced by architectural extensions, see A-profile Architecture Extensions.

This section presents information about the register ordering by {CRn, opc1, CRm, opc2}. It contains the following subsections:

- System register summary for (coproc == 0b1111) encodings by CRn value.
- Full list of AArch32 VMSA System registers in the (coproc == 0b1111) encoding space.

Note

The ordered listing of (coproc == 0b1111) registers by the {CRn, opc1, CRm, opc2} encoding of the 32-bit registers is most likely to be useful to those implementing AArch32 state, and to those validating such implementations. However, otherwise, the grouping of registers by function is more logical.

In addition, the indexes in Registers Index include all of the System registers.

G7.3.1 System register summary for (coproc == 0b1111) encodings by CRn value

Figure G7-1 summarizes the grouping of the System registers in the (coproc == 0b1111) encoding space, for an AArch32 VMSA implementation, by the value of CRn used for a 32-bit access to the register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRn	opc1	CRm	opc2	
c0	{0-2, 4}	{c0-c2}	{0-7}	ID registers
c1	{0, 4}	{c0, c1}	{0-7}	System control registers
c2	{0, 4}	{c0, c1}	{0-2}	Memory system control registers
c3	0	c0	0	
c4	0	c6	0	GIC System register *, Debug exception registers
c5	{0, 4}	{c0,c1}	{0,1}	Memory system fault registers
c6	{0, 4}	c0	{0, 2, 4}	
c7	{0, 4}	Various	Various	Cache maintenance, address translations, legacy operations
c8	{0, 4}	Various	Various	TLB maintenance operations
c9	{0-7}	Various	{0-7}	Performance monitors
c10	{0-7}	Various	{0-7}	Memory mapping registers and TLB operations
c11	{0-7}	{c0-c8,c15}	{0-7}	Reserved for DMA operations for TCM access
c12	{0-2, 4, 6}	Various	{0,1}	System control registers, GIC System registers *
c13	{0, 4}	Various	Various	Process, Context, Thread ID registers, Activity Monitors registers *
c14	{0-7}	{c0-c15}	{0-7}	Generic Timer registers *, Performance Monitors registers *
c15	{0-7}	{c0-c15}	{0-7}	IMPLEMENTATION DEFINED registers

* If implemented

Figure G7-1 AArch32 System register groupings for (coproc == 0b1111), for 32-bit registers

Note

For the System registers in the (`coproc` == `0b1111`) encoding space, Figure G7-1 gives only an overview of the assigned encodings for 32-bit registers for each of the `CRn` values `c0` - `c15`. For more information, see:

- The full list of registers in the (`coproc` == `0b1111`) encoding space, in Full list of AArch32 VMSA System registers in the (coproc == `0b1111`) encoding space, for the definition of the assigned and unassigned encodings for that register.
- The register definitions in AArch32 System Register Descriptions for any dependencies on the implemented Exception levels.

In general, System register accesses using an unallocated set of {`CRn`, `opc1`, `CRm`, `opc2`} values are UNDEFINED. Behavior of AArch32 VMSA System registers with (coproc == `0b1111`, CRn == c0) described the only exceptions to this rule.

The 32-bit System registers with (`coproc` == `0b1111`, CRn == c15), and the corresponding 64-bit System registers, are reserved for IMPLEMENTATION DEFINED registers. For more information, see Reserved encodings in the AArch32 VMSA System register (coproc == `0b1111`) space.

G7.3.1.1 The HSTR.Tn trap on (`coproc` == `0b1111`) System registers

When the value of HSTR.T*n* is 1, Non-secure PL1 accesses to System registers in the (`coproc` == `0b1111`) encoding space using a `CRn` or `CRm` value that corresponds to the value of T*n* are trapped to EL2, even if the encoding is UNDEFINED when the value of HSTR.T*n* is 0. This applies:

- For 32 bit register accesses when the value of `Rn` in the `MCR` or `MRC` instruction corresponds to T*n*.
- For 64 bit register accesses when the value of `Rm` in the `MCRR` or `MRRC` instruction corresponds to T*n*.

If there are matching System register encodings that are accessible from Non-secure EL0 then those accesses are also trapped to EL2 when the value of HSTR.T*n* is 1.

G7.3.1.2 Behavior of AArch32 VMSA System registers with (`coproc` == `0b1111`, CRn == c0)

In the (`coproc` == `0b1111`) encoding space, the 32-bit System registers with (`CRn` == `c0`) provide device and feature identification.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table G7-3 shows all of the architecturally required System registers with {coproc == 0b1111, CRn == c0}. The behavior of 32-bit System register encodings in this group that are not shown in the table, and encodings that are part of an unimplemented Exception level, depends on the value of opc1, and possibly on the value of CRm and opc2, as follows:

opc1 == 0 All write accesses to the encodings are UNDEFINED.

For read accesses:

- The following encodings return an UNKNOWN value:
  - CRm == 3, opc2 == {0, 1, 2}.
  - CRm == {4, 6, 7}, opc2 == {0, 1}.
  - CRm == 5, opc2 == {0, 1, 4, 5}.
- All other encodings are RES0.

opc1 > 0 All accesses to the encodings are UNDEFINED.

See also Accesses to unallocated encodings in the (coproc == 0b111x) encoding space.

Note

Some of these registers were previously described as being part of the CPUID identification scheme, see The CPUID identification scheme.

G7.3.1.3 Reserved encodings in the AArch32 VMSA System register (coproc == 0b1111) space

AArch32 state reserves a number of regions in the (coproc == 0b1111) encoding space for IMPLEMENTATION DEFINED System registers. These reservations are defined in terms of the encoding of 32-bit accesses to the System register encoding space. That is, they are defined by the reserved 32-bit {CRn, opc1, CRm, opc2} encodings.

Reserved encodings that do not have an IMPLEMENTATION DEFINED function are UNDEFINED.

The following subsections give more information about these reserved encodings:

- Reserved 32-bit encodings with { coproc == 0b1111, CRn == c9 }.
- Reserved 32-bit encodings with { coproc == 0b1111, CRn == c10 }.
- Reserved 32-bit encodings with { coproc == 0b1111, CRn == c11 }.
- Reserved 32-bit encodings with { coproc == 0b1111, CRn == c15 }.

G7.3.1.3.1 Reserved 32-bit encodings with { coproc == 0b1111, CRn == c9}

In the AArch32 encoding space, for 32-bit encodings with {coproc == 0b1111, CRn == c9}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc == 0b1111, CRn == c9, opc1 == {0 - 7}, opc2== {0 - 7}, CRm == {c0 - c2, c5 - c8}} are reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations.
- Encodings with {coproc == 0b1111, CRn == c9, opc1 == {0 - 7}, opc2== {0 - 7}, CRm == c15 } are reserved for IMPLEMENTATION DEFINED performance monitors.

  Note

  These are distinct from the OPTIONAL Arm Performance Monitors Extension, the registers for which use the encoding space {coproc == 0b1111, CRn == c9, opc1 == {0 - 7}, opc2== {0 - 7}, CRm == {c12 - c14}}.

G7.3.1.3.2 Reserved 32-bit encodings with { coproc == 0b1111, CRn == c10}

In the AArch32 encoding space, for 32-bit encodings with {coproc == 0b1111, CRn == c10}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc == 0b1111, CRn == c10, opc == {0 - 7}, CRm == {c0, c1, c4, c8}} are reserved for IMPLEMENTATION DEFINED TLB lockdown operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G7.3.1.3.3 Reserved 32-bit encodings with { coproc == 0b1111, CRn == c11}

In the AArch32 encoding space, for 32-bit encodings with {coproc == 0b1111, CRn == c11}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc == 0b1111, CRn == c11, opc == {0 - 7}, CRm == {c0 - c8, c15}} are reserved for IMPLEMENTATION DEFINED DMA operations for TCM access.

The remainder of the AArch32 {coproc == 0b1111, CRn == c11} encoding space is UNDEFINED.

G7.3.1.3.4 Reserved 32-bit encodings with { coproc == 0b1111, CRn == c15}

The AArch32 System register encodings are reserved with (coproc == 0b1111, CRn == c15) for IMPLEMENTATION DEFINED purposes, and there are no restrictions on the use of these encodings. The documentation of the Arm implementation must describe fully any registers implemented in the {coproc == 0b1111, CRn == c15} encoding space. Normally, for processor implementations by Arm, this information is included in the *Technical Reference Manual* for the processor.

Typically, an implementation uses the {coproc == 0b1111, CRn == c15} encodings to provide test features, and any required configuration options that are not covered by this Manual.

This reservation means that the AArch32 64-bit encodings with {coproc == 0b1111, CRm == c15} are also reserved for IMPLEMENTATION DEFINED purposes, without any restrictions on the use of these encodings.

G7.3.2 Full list of AArch32 VMSA System registers in the (coproc == 0b1111) encoding space

Table G7-3 shows the System registers in the (coproc == 0b1111) encoding space in the AArch32 VMSA, in the order of the {CRn, opc1, CRm, opc2} parameter values used in MCR or MRC accesses to the 32-bit registers:

- For MCR or MRC accesses to the 32-bit registers, CRn is the primary identifier of the target System register for the access. This applies, also, to MCR or MRC instructions that provide 32-bit accesses to a single word of a 64-bit System register.
- For MCRR or MRRC accesses to the 64-bit registers, CRm is the primary identifier of the target System register for the access. Table G7-3 orders the 64-bit registers with the 32-bit registers accessed using the same primary register identifier. For example, the 64-bit encoding of TTBR0, that is accessed with (CRm == c2), is listed with the 32-bit registers that are accessed with (CRn == c2).

Table G7-3 AArch32 VMSA (coproc == 0b1111) register summary, in MCR/MRC parameter order

Name	CRn	opc1	CRm	opc2	Source
MIDR	c0	0	c0	0	v8.0
CTR				1	v8.0
TCMTR				2	v8.0
TLBTR				3	v8.0
MIDR				4, 6 [a], 7	v8.0
MPIDR				5	v8.0
REVIDR				6 [a]	v8.0
ID_PFR0			c1	0	v8.0
ID_PFR1				1	v8.0
ID_DFR0				2	v8.0
ID_AFR0				3	v8.0
ID_MMFR0				4	v8.0
ID_MMFR1				5	v8.0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
ID_MMFR2				6	v8.0
ID_MMFR3				7	v8.0
ID_ISAR0			c2	0	v8.0
ID_ISAR1				1	v8.0
ID_ISAR2				2	v8.0
ID_ISAR3				3	v8.0
ID_ISAR4				4	v8.0
ID_ISAR5				5	v8.0
ID_MMFR4				6	v8.0
ID_ISAR6				7	v8.0
ID_PFR2			c3	4	v8.0
ID_DFR1				5	v8.6
ID_MMFR5				6	v8.6
CCSIDR		1	c0	0	v8.0
CLIDR				1	v8.0
CCSIDR2				2	v8.3[b]
AIDR				7	v8.0
CSSELR	c0	2	c0	0	v8.0
VPIDR [c]		4	c0	0	v8.0
VMPIDR [c]				5	v8.0
SCTLR	c1	0	c0	0	v8.0
ACTLR				1	v8.0
CPACR				2	v8.0
ACTLR2				3	v8.0
SCR [d]			c1	0	v8.0
SDER [d]				1	v8.0
NSACR [d]				2	v8.0
TRFCR			c2	1	v8.4
SDCR			c3	1	v8.0
HSCTLR [c]		4	c0	0	v8.0
HACTLR [c]				1	v8.0
HACTLR2 [c]				3	v8.0
HCR [c]			c1	0	v8.0
HDCR [c]				1	v8.0
HCPTR [c]				2	v8.0
HSTR [c]				3	v8.0
HCR2 [c]				4	v8.0
HACR [c]				7	v8.0
HTRFCR			c2	1	v8.4
TTBR0, 32-bit view	c2	0	c0	0	v8.0
TTBR0, 64-bit view	-	0	c2	-	v8.0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
TTBR1, 32-bit view	c2	0	c0	1	v8.0
TTBR1, 64-bit view	-	1	c2	-	v8.0
TTBCR	c2	0	c0	2	v8.0
TTBCR2				3	v8.2
HTCR [c]		4	c0	2	v8.0
VTCR [c]			c1	2	v8.0
HTTBR [c], 64-bit view	-	4	c2	-	v8.0
VTTBR [c]. 64-bit view	-	6	c2	-	v8.0
DACR	c3	0	c0	0	v8.0
ICC_PMR ICV_PMR	c4	0	c6	0	GIC[e]
DSPSR [f]	c4	3	c5	0	v8.0
DLR				1	v8.0
DFSR	c5	0	c0	0	v8.0
IFSR				1	v8.0
ADFSR			c1	0	v8.0
AIFSR				1	v8.0
ERRIDR			c3	0	RAS[g]
ERRSELR				1	RAS[g]
ERXFR			c4	0	RAS[g]
ERXCTLR				1	RAS[g]
ERXSTATUS				2	RAS[g]
ERXADDR				3	RAS[g]
ERXFR2				4	RAS[g]
ERXCTLR2				5	RAS[g]
ERXADDR2				7	RAS[g]
ERXMISC0			c5	0	RAS[g]
ERXMISC1				1	RAS[g]
ERXMISC4				2	RAS[g]
ERXMISC5				3	RAS[g]
ERXMISC2				4	RAS[g]
ERXMISC3				5	RAS[g]
ERXMISC6				6	RAS[g]
ERXMISC7				7	RAS[g]
HADFSR [c]		4	c1	0	v8.0
HAIFSR				1	v8.0
HSR [c]			c2	0	v8.0
VDFSR				3	RAS[g]
DFAR	c6	0	c0	0	v8.0
IFAR				2	v8.0
HDFAR [c]		4	c0	0	v8.0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
HIFAR [c]				2	v8.0
HPFAR [c]	c6	4	c0	4	v8.0
ICIALLUIS	c7	0	c1	0	v8.0
BPIALLIS				6	v8.0
CFPRCTX			c3	4	v8.0[h]
DVPRCTX				5	v8.0[h]
COSPRCTX				6	v8.9
CPPRCTX				7	v8.0[h]
PAR, 32-bit view			c4	0	v8.0
PAR, 64-bit view	-	0	c7	-	v8.0
ICIALLU	c7	0	c5	0	v8.0
ICIMVAU				1	v8.0
CP15ISB [i]				4	v8.0
BPIALL				6	v8.0
BPIMVA				7	v8.0
DCIMVAC			c6	1	v8.0
DCISW				2	v8.0
ATS1CPR			c8	0	v8.0
ATS1CPW				1	v8.0
ATS1CUR				2	v8.0
ATS1CUW				3	v8.0
ATS12NSOPR [d]				4	v8.0
ATS12NSOPW [d]				5	v8.0
ATS12NSOUR [d]				6	v8.0
ATS12NSOUW [d]				7	v8.0
DCCMVAC			c10	1	v8.0
DCCSW				2	v8.0
CP15DSB [i]				4	v8.0
CP15DMB [i]				5	v8.0
DCCMVAU	c7	0	c11	1	v8.0
DCCIMVAC			c14	1	v8.0
DCCISW				2	v8.0
ATS1HR [c]		4	c8	0	v8.0
ATS1HW [c]				1	v8.0
TLBIALLIS	c8	0	c3	0	v8.0
TLBIMVAIS				1	v8.0
TLBIASIDIS				2	v8.0
TLBIMVAAIS				3	v8.0
TLBIMVALIS				5	v8.0
TLBIMVAALIS				7	v8.0
ITLBIALL			c5	0	v8.0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
ITLBIMVA				1	v8.0
ITLBIASID				2	v8.0
DTLBIALL			c6	0	v8.0
DTLBIMVA				1	v8.0
DTLBIASID				2	v8.0
TLBIALL			c7	0	v8.0
TLBIMVA				1	v8.0
TLBIASID				2	v8.0
TLBIMVAA				3	v8.0
TLBIMVAL				5	v8.0
TLBIMVAAL				7	v8.0
TLBIIPAS2IS		4	c0	1	v8.0
TLBIIPAS2LIS				5	v8.0
TLBIALLHIS [c]			c3	0	v8.0
TLBIMVAHIS [c]				1	v8.0
TLBIALLNSNHIS [c]				4	v8.0
TLBIMVALHIS				5	v8.0
TLBIIPAS2			c4	1	v8.0
TLBIIPAS2L				5	v8.0
TLBIALLH [c]	c8	4	c7	0	v8.0
TLBIMVAH [c]				1	v8.0
TLBIALLNSNH [c]				4	v8.0
TLBIMVALH				5	v8.0
Reserved [j]	c9	0 - 7	c0 - c2	0 - 7	-
Reserved [j]			c5 - c8	0 - 7	-
PMCR [k]		0	c12	0	v8.0
PMCNTENSET [k]				1	v8.0
PMCNTENCLR [k]				2	v8.0
PMOVSR [k]				3	v8.0
PMSWINC [k]				4	v8.0
PMSELR [k]				5	v8.0
PMCEID0 [k]				6	v8.0
PMCEID1 [k]				7	v8.0
PMCCNTR [k], 32-bit view			c13	0	v8.0
PMCCNTR_EL0 [k], 64-bit view	-	0	c9	-	v8.0
PMXEVTYPER [k]	c9	0	c13	1	v8.0
PMXEVCNTR [k]				2	v8.0
PMUSERENR [k]			c14	0	v8.0
PMINTENSET [k]				1	v8.0
PMINTENCLR [k]				2	v8.0
PMOVSSET [c, k]				3	v8.0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
PMCEID2 [k]				4	v8.1
PMCEID3 [k]				5	v8.1
PMMIR				6	v8.4
Reserved [l]		0 - 7	c15	0 - 7	-
Reserved [m]	c10	0	c0 - c1	0 - 7	-
PRRR [n]			c2	0	v8.0
MAIR0 [n]					v8.0
NMRR [n]				1	v8.0
MAIR1 [n]					v8.0
AMAIR0			c3	0	v8.0
AMAIR1				1	v8.0
Reserved [m]			c4, c8	0 - 7	-
Reserved [m]		1- 3	c0, c1, c4, c8	0 - 7	-
Reserved [m]		4	c0, c1	0 - 7	-
HMAIR0 [c]			c2	0	v8.0
HMAIR1 [c]				1	v8.0
HAMAIR0 [c]			c3	0	v8.0
HAMAIR1 [c]				1	v8.0
Reserved [m]			c4, c8	0 - 7	-
Reserved [m]		5 - 7	c0, c1, c4, c8	0 - 7	-
Reserved [o]	c11	0-7	c0-c8	0 - 7	-
Reserved [o]			c15	0 - 7	-
ICC_SGI1R, 64-bit view	-	0	c12	-	GIC[e]
VBAR	c12	0	c0	0	v8.0
MVBAR [d]				1	v8.0
RVBAR					v8.0
RMR [p]				2	v8.0
ISR [d]			c1	0	v8.0
DISR				1	RAS[g]
VDISR		4	c1	1	RAS[g]
ICC_IAR0 ICV_IAR0		0	c8	0	GIC[e]
ICC_EOIR0 ICV_EOIR0				1	GIC[e]
ICC_HPPIR0 ICV_HPPIR0				2	GIC[e]
ICC_BPR0 ICV_BPR0	c12	0	c8	3	GIC[e]
ICC_AP0R0 ICV_AP0R0				4	GIC[e]
ICC_AP0R1 ICV_AP0R1				5	GIC[e]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
ICC_AP0R2 ICV_AP0R2				6	GIC[e]
ICC_AP0R3 ICV_AP0R3				7	GIC[e]
ICC_AP1R0 ICV_AP1R0			c9	0	GIC[e]
ICC_AP1R1 ICV_AP1R1				1	GIC[e]
ICC_AP1R2 ICV_AP1R2				2	GIC[e]
ICC_AP1R3 ICV_AP1R3				3	GIC[e]
ICC_DIR ICV_DIR			c11	1	GIC[e]
ICC_RPR ICV_RPR				3	GIC[e]
ICC_IAR1 ICV_IAR1			c12	0	GIC[e]
ICC_EOIR1 ICV_EOIR1				1	GIC[e]
ICC_HPPIR1 ICV_HPPIR1				2	GIC[e]
ICC_BPR1 ICV_BPR1				3	GIC[e]
ICC_CTLR ICV_CTLR				4	GIC[e]
ICC_SRE				5	GIC[e]
ICC_IGRPEN0 ICV_IGRPEN0				6	GIC[e]
ICC_IGRPEN1 ICV_IGRPEN1				7	GIC[e]
ICC_ASGI1R, 64-bit view	-	1	c12	-	GIC[e]
ICC_SGI0R, 64-bit view	-	2	c12	-	GIC[e]
HVBAR [c]	c12	4	c0	0	v8.0[e]
HRMR [p]				2	v8.0[e]
ICH_AP0R0			c8	0	GIC[e]
ICH_AP0R1				1	GIC[e]
ICH_AP0R2				2	GIC[e]
ICH_AP0R3	c12	4	c8	3	GIC[e]
ICH_AP1R0			c9	0	GIC[e]
ICH_AP1R1				1	GIC[e]
ICH_AP1R2				2	GIC[e]
ICH_AP1R3				3	GIC[e]
ICC_HSRE				5	GIC[e]
ICH_HCR			c11	0	GIC[e]
ICH_VTR				1	GIC[e]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
ICH_MISR				2	GIC[e]
ICH_EISR				3	GIC[e]
ICH_ELRSR				5	GIC[e]
ICH_VMCR				7	GIC[e]
ICH_LR, for n == 0 to 7			c12	0-7	GIC[e]
ICH_LR, for n == 8 to 15			c13	0-7	GIC[e]
ICH_LRC, for n == 0 to 7			c14	0-7	GIC[e]
ICH_LRC, for n == 8 to 15			c15	0-7	GIC[e]
ICC_MCTLR		6	c12	4	GIC[e]
ICC_MSRE				5	GIC[e]
ICC_MGRPEN1				7	GIC[e]
FCSEIDR	c13	0	c0	0	v8.0
CONTEXTIDR				1	v8.0
TPIDRURW				2	v8.0
TPIDRURO				3	v8.0
TPIDRPRW				4	v8.0
AMCR			c2	0	AMU[q]
AMCFGR			c2	1	AMU[q]
AMCGCR			c2	2	AMU[q]
AMUSERENR			c2	3	AMU[q]
AMCNTENCLR0			c2	4	AMU[q]
AMCNTENSET0			c2	5	AMU[q]
AMCNTENCLR1			c3	0	AMU[q]
AMCNTENSET1			c3	1	AMU[q]
AMEVTYPER0<n>, for n == 0 to 7			c6	0-7	AMU[q]
AMEVTYPER0<n>, for n == 8 to 15			c7		AMU[q]
AMEVTYPER1<n>, for n == 0 to 7			c14		AMU[q]
AMEVTYPER1<n>, for n == 8 to 15			c15		AMU[q]
AMEVCNTR0<n>, for n == 0 to 7, 64-bit view	-	0-7	c0	-	AMU[q]
AMEVCNTR0<n>, for n == 8 to 15, 64-bit view	-		c1		AMU[q]
AMEVCNTR1<n> for n == 0 to 7, 64-bit view	-		c4		AMU[q]
AMEVCNTR1<n>, for n == 8 to 15, 64-bit view	-		c5		AMU[q]
HTPIDR [c]	c13	4	c0	2	v8.0
CNTPCT [r], 64-bit view	-	0	c14	-	v8.0
CNTFRQ [r]	c14	0	c0	0	v8.0
CNTKCTL [r]			c1	0	v8.0
CNTP_TVAL [r]			c2	0	v8.0
CNTP_CTL [r]				1	v8.0
CNTV_TVAL [r]			c3	0	v8.0
CNTV_CTL [r]				1	v8.0
PMEVCNTR<n>, for n == 0 to 7 [k]			c8	0-7	v8.0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	CRn	opc1	CRm	opc2	Source
PMEVCNTR<n>, for n == 8 to 15 [k]			c9	0-7	v8.0
PMEVCNTR<n>, for n == 16 to 23 [k]			c10	0-7	v8.0
PMEVCNTR<n>, for n == 24 to 30 [k]			c11	0-6	v8.0
PMEVTYPER<n>, for n == 0 to 7 [k]			c12	0-7	v8.0
PMEVTYPER<n>, for n == 8 to 15 [k]			c13	0-7	v8.0
PMEVTYPER<n>, for n == 16 to 23 [k]			c14	0-7	v8.0
PMEVTYPER<n>, for n == 17 to 30 [k]			c15	0-6	v8.0
PMCCFILTR [k]			c15	7	v8.0
CNTVCT [r], 64-bit view	-	1	c14	-	v8.0
CNTP_CVAL [r], 64-bit view	-	2	c14	-	v8.0
CNTV_CVAL [r], 64-bit view	-	3	c14	-	v8.0
CNTVOFF [s], 64-bit view	-	4	c14	-	v8.0
CNTHCTL [r]	c14	4	c1	0	v8.0
CNTHP_TVAL [r]	c14	4	c2	0	v8.0
CNTHP_CTL [r]				1	v8.0
CNTHP_CVAL [r], 64-bit view	-	6	c14	-	v8.0
CNTPCTSS [r], 64-bit view	-	8	c14	-	v8.6
CNTVCTSS [r], 64-bit view	-	9	c14	-	v8.6
Reserved [t]	c15	0-7	c0-c15	0-7	-

a. REVIDR is an optional register. If it is not implemented, the encoding with opc2 set to 6 is an alias of MIDR.

b. When FEAT_CCIDX is implemented, CCSIDR2 is implemented.

c. Implemented only as part of EL2 when EL2 is using AArch32. Otherwise, encoding is unallocated and UNDEFINED.

d. Implemented only as part of EL3 when EL3 is using AArch32. Otherwise, encoding is unallocated and UNDEFINED.

e. GIC System register, see About the GIC System registers. As that subsection describes, each ICV_* register uses the same encoding as the corresponding ICC_* register.

f. This register is accessible only in Debug state.

g. RAS Extension System registers, see RAS registers.

h. When FEAT_SPECRES is implemented, the execution and data prediction restriction instructions are implemented, see Execution and data prediction restriction System instructions.

i. For performance reasons, Arm deprecates any use of these memory barrier operations.

j. Reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations, see Reserved 32-bit encodings with { coproc == 0b1111, CRn == c9}.

k. Performance Monitors Extension System register, see Performance Monitors registers.

l. Reserved for IMPLEMENTATION DEFINED performance monitors, see Reserved 32-bit encodings with { coproc == 0b1111, CRn == c9}.

m. Reserved for IMPLEMENTATION DEFINED TLB lockdown operations, see Reserved 32-bit encodings with { coproc == 0b1111, CRn == c10}.

n. When an implementation is using the Long descriptor translation table format, these encodings access the MAIR0 and MAIR1 registers. Otherwise, they use PRRR and NMRR.

o. Reserved for IMPLEMENTATION DEFINED DMA operations for TCM access, see Reserved 32-bit encodings with { coproc == 0b1111, CRn == c11}.

p. Only one of RMR and HRMR is implemented, corresponding to the highest implemented Exception level, and the register is implemented only if that Exception level is using AArch32.

q. Activity Monitors System register, see Activity Monitors registers.

r. Generic Timer System register, see Generic Timer registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

s. Implemented as RW as part of the Generic Timer on an implementation that includes EL2 and when EL2 is using AArch32. For more information, see The virtual offset register.

t. Reserved for IMPLEMENTATION DEFINED purposes, see Reserved 32-bit encodings with { coproc == 0b1111, CRn == c15}.

G7.3.2.1 About the GIC System registers

From version 3 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_ GIC physical CPU interface System registers.

ICH_ GIC virtual interface control System registers.

ICV_ GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

In AArch32, the GIC System registers are all in the (coproc == 0b1111) encoding space with (CRn == c12). The ICV_* registers have the same {CRn, opc1, CRm, op2} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

For more information, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G8
AArch32 System Register Descriptions

This chapter describes each of the AArch32 System registers.

It contains the following sections:

- About the AArch32 System registers.
- General system control registers.
- Debug System registers.
- Performance Monitors registers.
- Activity Monitors registers.
- RAS registers.
- Generic Timer registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.1 About the AArch32 System registers

For general information about the AArch32 System registers, see:

In The AArch32 Virtual Memory System Architecture:

- About the System registers for VMSAv8-32.
- Functional grouping of VMSAv8-32 System registers.

In AArch32 System Register Encoding:

- Organization of registers in the (coproc == `0b1110`) encoding space.
- Organization of registers in the (coproc == `0b1111`) encoding space.

In this chapter:

- Fixed values in the System register descriptions.
- General behavior of System registers.
- Principles of the ID scheme for fields in ID registers.
- About AArch32 System register accesses.

The remainder of this chapter describes the AArch32 System registers, in the following sections:

- General system control registers.
- Debug System registers.
- Performance Monitors registers.
- Generic Timer registers.

G8.1.1 Fixed values in the System register descriptions

See Fixed values in AArch32 instruction and System register descriptions. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

G8.1.2 General behavior of System registers

Except where indicated, System registers are 32-bits wide. As stated in About the System registers for VMSAv8-32, there are some 64-bit registers, and these include cases where software can access either a 32-bit view or a 64-bit view of a register. The register summaries, and the individual register descriptions, identify the 64-bit registers and how they can be accessed.

The following sections give information about the general behavior of these registers:

- Register names.
- Read-only bits in read/write registers.
- The CPUID identification scheme.
- IMPLEMENTATION DEFINED performance monitors.
- UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses.
- Read-only and write-only register encodings.
- Reset behavior of AArch32 System registers.
- Synchronization of changes to AArch32 System registers.

Unless otherwise indicated, information in the listed sections applies to all AArch32 System registers

See also About AArch32 System register accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.1.2.1 Register names

The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_*, where appropriate.

G8.1.2.2 Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

G8.1.2.3 The CPUID identification scheme

The ID_* registers were originally called the CPUID identification scheme registers. However, functionally, there is no value in separating these registers from the slightly larger Identification registers functional group. See Table K14-30 for a list of the ID_ * registers.

G8.1.2.4 IMPLEMENTATION DEFINED performance monitors

VMSAv8-32 reserves some additional System register encodings in the (`coproc` == `0b1111`) encoding space for optional additional IMPLEMENTATION DEFINED performance monitors. Table G8-1 shows the allocation of these encodings:

Table G8-1 Performance Monitors System register encoding allocations

CRn	opc1	CRm	opc2	Name	Type
c9	0-7	c12-c14	0-7	Performance Monitors Extension registers, see Performance monitors System registers	RW or RO[a]
		c15	0-7	IMPLEMENTATION DEFINED	[b]

a. The table referenced in the Name entry shows the type of each of the OPTIONAL Performance Monitors Extension registers.

b. Access depends on the register or operation, and is IMPLEMENTATION DEFINED.

G8.1.2.5 UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses

This section defines UNPREDICTABLE and UNDEFINED behaviors for accesses to System registers, including those cases where the Armv8 behavior is CONSTRAINED UNPREDICTABLE.

In AArch32 state, the following operations are UNDEFINED:

- All `LDC` and `STC` accesses, except for the `LDC` access to DBGDTRTXint and the `STC` access to DBGDTRRXint specified in Table G7-2.
- All `MCRR` and `MRRC` operations to the (`coproc` == `0b111x`) encoding space, except for those explicitly defined as accessing 64-bit System registers specified in Table G7-1 and Table G7-3.

Unless otherwise indicated in the individual register descriptions:

- Reserved fields in registers are RES0.
- Assigning a reserved value to a field has a CONSTRAINED UNPREDICTABLE effect, see Reserved values in System and memory-mapped registers and translation table entries.

The following subsections give more information about UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for accesses to the (`coproc` == `0b111x`) encoding space:

- Accesses to unallocated encodings in the (coproc == `0b111x`) encoding space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Additional rules for MCR and MRC accesses to System registers.
- Effects of EL3 and EL2 on System register accesses.

G8.1.2.5.1 Accesses to unallocated encodings in the (`coproc` == `0b111x`) encoding space

In Armv8-A, accesses to unallocated register encodings in the (`coproc` == `0b111x`) encoding space are UNDEFINED.

Note

In Armv7, except for 32-bit registers encoded with a `CRn` value of `c12`, accesses to unallocated 32-bit registers were UNPREDICTABLE. The Armv8 CONSTRAINED UNPREDICTABLE behavior of these accesses is that they are UNDEFINED, see Unallocated System register access instructions.

G8.1.2.5.2 Additional rules for MCR and MRC accesses to System registers

The following operations are CONSTRAINED UNPREDICTABLE for all encodings in the (`coproc` == `0b111x`) encoding space:

- All `MCR` operations from the PC.
- All `MRC` operations to `APSR_nzcv`, except for the (`coproc` == `0b1110`) `MRC` operation to `APSR_nzcv` from DBGDSCRint.

The CONSTRAINED UNPREDICTABLE behavior of these operations is described in Using R15 by instruction.

For registers and operations that are accessible from a particular Privilege level, any attempt to access those registers from a lower Privilege level is UNDEFINED.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Unless explicitly stated otherwise in this manual, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for `MRC` and `MCR` accesses.

See also Read-only and write-only register encodings.

G8.1.2.5.3 Effects of EL3 and EL2 on System register accesses

EL2 and EL3 introduce classes of System registers, described in Classification of System registers. Some of these classes of register are either:

- Accessible only from certain modes or states.
- Accessible from certain modes or states only when configuration settings permit the access.

Accesses to these registers that are not permitted are UNDEFINED, meaning execution of the register access instruction generates an Undefined Instruction exception.

Note

This section applies only to registers that are accessible from some modes and states. That is, it applies only to register access instructions using an encoding that, under some circumstances, would perform a valid register access.

The following register classes restrict access in this way:

Restricted access System registers

This register class is defined in any implementation that includes EL3.

Restricted access registers other than the NSACR are accessible only from Secure EL3 modes. All other accesses to these registers are UNDEFINED.

The NSACR is a special case of a Restricted access register and:

- The NSACR is:
  - Read/write accessible from Secure PL1 modes.
  - Is Read-only accessible from Non-secure PL2 and PL1 modes.
- All other accesses to the NSACR are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, including behavior when EL3 is using AArch64 or is not implemented, see Restricted access System registers.

Configurable access System registers

This register class is defined in any implementation that includes EL3.

Most Configurable access registers are accessible from Non-secure state only if control bits in the NSACR permit Non-secure access to the register. Otherwise, a Non-secure access to the register is UNDEFINED.

For other Configurable access registers, control bits in the NSACR control the behavior of bits or fields in the register when it is accessed from Non-secure state. That is, Non-secure accesses to the register are permitted, but the NSACR controls how they behave. The only architecturally-defined register of this type is the CPACR.

For more information, see Configurable access System registers.

EL2-mode System registers

This register class is defined only in an implementation that includes EL2.

EL2-mode registers are accessible only from:

- The Non-secure EL2 mode, Hyp mode.
- Secure Monitor mode when SCR.NS is set to 1.

All other accesses to these registers are UNDEFINED.

For more information, see Hyp mode read/write registers in the (coproc == `0b1111`) encoding space and Hyp mode encodings for shared (coproc == `0b1111`) System registers.

EL2-mode write-only operations

This register class is defined only in an implementation that includes EL2.

EL2-mode write-only operations are accessible only from:

- The Non-secure EL2 mode, Hyp mode.
- Secure Monitor mode, regardless of the value of SCR.NS.

Write accesses to these operations are:

- CONSTRAINED UNPREDICTABLE in Secure EL3 modes other than Monitor mode.
- UNDEFINED in Non-secure modes other than Hyp mode.

For more information, see Hyp mode (coproc == `0b1111`) write-only System instructions.

In addition, in any implementation that includes EL3, when EL3 is using AArch32, if write access to a register is disabled by the `CP15SDISABLE` signal then any `MCR` access to that register is UNDEFINED.

G8.1.2.6 Read-only and write-only register encodings

Some System registers are *read-only* (RO) or *write-only* (WO). For example:

- Most identification registers are read-only.
- Most encodings that perform an operation, such as a cache maintenance instruction, are write-only.

If a particular Privilege level defines a register to be:

- RO, then any attempt to write to that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L == 0 is UNDEFINED.
- WO, then any attempt to read from that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L== 1 is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This section applies only to registers that this manual defines as RO or WO. It does not apply to registers for which other access permissions are explicitly defined.

G8.1.2.7 Reset behavior of AArch32 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch32, see:

- PE state on reset into AArch32 state.
- The appropriate Trace architecture specification, for the Trace System registers.

When the PE resets into an Exception level that is using AArch64, PE state that relates to execution in AArch32 state, including the System register values, is UNKNOWN. The only exception to this is state that applies to execution in both AArch64 state and AArch32 state and that has a defined reset value on the reset into AArch64 state. An example of such PE state is the EDPRSR.SR bit.

For a PE reset into an Exception level that is using AArch32, the architecture defines which AArch32 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and PE state on reset into AArch32 state summarizes these architectural requirements. Otherwise, RW registers reset to an architecturally unknown value.

Note

In an implementation that includes EL3, unless this manual explicitly states otherwise, only the Secure instance of a banked register is reset to the defined value. This means that software must program the Non-secure instance of the register with the required values. Typically, this programming is part of the PE boot sequence.

G8.1.2.7.1 Pseudocode description of resetting System registers

The `AArch32_ResetControlRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and PE state on reset into AArch32 state.

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

G8.1.2.8 Synchronization of changes to AArch32 System registers

In this section, *this PE* means the PE on which accesses are being synchronized.

Note

See Definitions of direct and indirect reads and writes and their side-effects for definitions of the terms *direct write*, *direct read*, *indirect write*, and *indirect read*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A *direct write* to a System register might become visible at any point after the change to the register, but without a Context Synchronization event there is no guarantee that the change becomes visible.

Any direct write to a System register is guaranteed not to affect any instruction that appears, in program order, before the instruction that performed the direct write, and any direct write to a System register must be synchronized before any instruction that appears after the direct write, in program order, can rely on the effect of that write. The only exceptions to this are:

- All direct writes to the same register, using the same encoding, are guaranteed to occur in program order.
- All direct writes to a register are guaranteed to occur in program order relative to all direct reads of the same register using the same encoding.
- Any System register access that an Arm *Architecture Specification* or equivalent specification defines as not requiring synchronization.
- If an instruction that appears in program order before the direct write performs a memory access, such as a memory-mapped register access, that causes an indirect read or write to a register, that memory access is subject to the memory order model. In this case, if permitted by the memory order model, the instruction that appears in program order before the direct write can be affected by the direct write. For information about the memory order model, see Definition of the memory model.

These rules mean that an instruction that writes to one of the address translation instructions described in Address translation instructions must be explicitly synchronized to guarantee that the result of the address translation instruction is visible in the PAR.

Note

In this case, the direct write to the encoding of the address translation instruction causes an *indirect write* to the PAR. Without a Context Synchronization event after the direct write, there is no guarantee that the indirect write to the PAR is visible.

Conceptually, the explicit synchronization occurs as the first step of any Context Synchronization event. This means that if the operation uses the state that had been changed but not synchronized before the operation occurred, the operation is guaranteed to use the state as if it had been synchronized.

Note

- This explicit synchronization is applied as the first step of the execution of any instruction that causes the synchronization operation. This means it does not synchronize any effect of changes to the System registers that might affect the fetch and decode of the instructions that cause the operation, such as breakpoints or changes to translation tables.
- For a synchronous exception, the control state in use at the time the exception is generated determines the exception syndrome information, and this syndrome information is not changed by this synchronization at the start of taking the exception.

Except for the register reads listed in Registers with some architectural guarantee of ordering or observability, if no Context Synchronization event is performed, direct reads of System registers can occur in any order.

Table G8-2 shows the synchronization requirement between two reads or writes that access the same System register. In the column headings, *First* and *Second* refer to:

- Program order, for any read or write caused by the execution of an instruction by this PE, other than a read or write caused by a memory access made by that instruction.
- The order of arrival of asynchronous reads or writes made by this PE relative to the execution of instructions by this PE.

In addition:

- For indirect reads or writes caused by an external agent, such as a debugger, the mechanism that determines the order of the reads or writes is defined by that external agent. The external agent can provide mechanisms that ensure that any read or write it makes arrives at the PE. These indirect reads and writes are asynchronous to software execution on the PE.
- For indirect reads or writes caused by memory-mapped reads or writes made by this PE, the ordering of the memory accesses is subject to the memory order model, including the effect of the memory type of the accessed

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

memory address. This applies, for example, if this PE reads or writes one of its registers in a memory-mapped register interface.

The mechanism for ensuring completion of these memory accesses, including ensuring the arrival of the asynchronous read or write at the PE, is defined by the system.

Note

Such accesses are likely to be given a Device memory attribute, but requiring this is outside the scope of the architecture.

- For indirect reads or writes caused by autonomous asynchronous events that are counted, for example events caused by the passage of time, the events are ordered so that:
  - Counts progress monotonically.
  - The events arrive at the PE in finite time and without undue delay.

Table G8-2 Synchronization requirements for updates to System registers

First read or write	Second read or write	Context Synchronization event required
Direct read	Direct read	No
	Direct write	No
	Indirect read	No[a]
	Indirect write	No[a], but see text in this section for exceptions
Direct write	Direct read	No
	Direct write	No
	Indirect read	Yes[a]
	Indirect write	No, but see text in this section for exceptions
Indirect read	Direct read	No
	Direct write	No
	Indirect read	No
	Indirect write	No
Indirect write	Direct read	Yes, but see text in this section for exceptions
	Direct write	No, but see text in this section for exceptions
	Indirect read	Yes, but see text in this section for exceptions
	Indirect write	No, but see text in this section for exceptions

a. Although no synchronization is required between a Direct write and a Direct read, or between a Direct read and an Indirect write, this does not imply that a Direct read causes synchronization of a previous Direct write. This means that the sequence Direct write followed by Direct read followed by Indirect read, with no intervening context synchronization, does not guarantee that the Indirect read observes the result of the Direct write.

If the indirect write is to a register that Registers with some architectural guarantee of ordering or observability shows as having some guarantee of the visibility of an indirect write, synchronization might not be required.

If a direct read or a direct write to a register is followed by an indirect write to that register that is caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the indirect write relative to the direct read or direct write.

If an indirect write caused by a direct write is followed by an indirect write caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the two indirect writes.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect. For other information about indirect writes after a direct read or a direct write, see Definitions of direct and indirect reads and writes and their side-effects.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Where a register has more that one encoding, a direct write to the register using a particular encoding is not an indirect write to the same register with a different encoding.

Where an indirect write is caused by the action of an external agent, such as a debugger, or by a memory-mapped read or write by the PE, then an indirect write by that agent to a register using a particular access mechanism, followed by an indirect read by that agent to the same register using the same access mechanism and address does not need synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For information about the additional synchronization requirements for memory-mapped registers, see Synchronization requirements for AArch64 System registers.

To guarantee the visibility of changes to some registers, additional operations might be required before the Context Synchronization event. For such a register, the definition of the register identifies these additional requirements.

In this manual, unless the context indicates otherwise:

- *Accessing* a System register refers to a direct read or write of the register.
- *Using* a System register refers to an indirect read or write of the register.

G8.1.2.8.1 Registers with some architectural guarantee of ordering or observability

For the registers for which Table G8-3 shows that the ordering of direct reads is guaranteed, multiple direct reads of a single register, using the same encoding, occur in program order without any explicit ordering.

For the registers for which Table G8-3 shows that some observability of indirect writes is guaranteed, an indirect write to the register caused by an external agent, an autonomous asynchronous event, or as a result of a memory-mapped write, is both:

- Observable to direct reads of the register, in finite time, without explicit synchronization.
- Observable to subsequent indirect reads of the register without explicit synchronization.

These two sets of registers are similar, as Table G8-3 shows:

Table G8-3 Registers with a guarantee of ordering or observability, VMSAv8-32

Register	Ordering of direct reads	Observability of indirect writes	Notes
ISR	Guaranteed	Guaranteed	Interrupt Status Register
DBGCLAIMCLR	Guaranteed	Guaranteed	Debug CLAIM registers
DBGCLAIMSET	-	Guaranteed	
DBGDTRRXint	Guaranteed	Guaranteed	Debug Communication Channel registers
DBGDTRTXint	-	Guaranteed	
The DCC flags in DBGDSCRint	Guaranteed	Guaranteed	
CNTPCT	Guaranteed	Guaranteed	Generic Timer registers
CNTP_TVAL	Guaranteed	Guaranteed	
CNTVCT	Guaranteed	Guaranteed	
CNTV_TVAL	Guaranteed	Guaranteed	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Ordering of direct reads	Observability of indirect writes	Notes
CNTHP_TVAL	Guaranteed	Guaranteed	
PMCCNTR	Guaranteed	Guaranteed	Performance Monitors Extension registers, if the implementation includes the extension
PMEVCNTR<n>	Guaranteed	Guaranteed	
PMXEVCNTR	Guaranteed	Guaranteed	
PMOVSSET	Guaranteed	Guaranteed	
PMOVSR	Guaranteed	Guaranteed	
EDSCR.PipeAdv and the DCC flags in EDSCR	-	Guaranteed	Fields of the External Debug Status and Control Register

In addition to the requirements shown in Table G8-3:

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  — OSLAR_EL1. OSLAR_EL1 is indirectly read to determine whether the subsequent access is permitted.

  Note

  OSLAR_EL1 maps to the AArch32 System register DBGOSLAR.

  — EDLAR, if implemented. EDLAR is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

  Note

  This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table G8-3 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.
  For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a Context Synchronization event. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.
  Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.
- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See DCC and ITR access in Debug state.

The possibility that direct reads can occur *early*, in the absence of context synchronization, described in Ordering of reads of System registers, still applies to the registers listed in Table G8-3.

G8.1.2.8.2 Definitions of direct and indirect reads and writes and their side-effects

Direct and indirect reads and writes are defined as follows:

Direct read Is a read of a register, using an `MRC`, `MRRC`, or `STC` instruction, that the architecture permits for the current PE state.

If a direct read of a register has a side-effect of changing the value of a register, the effect of a direct read on that register is defined to be an *indirect write*, and has the synchronization requirements of an indirect write. This means the indirect write is guaranteed to have occurred, and to be visible to subsequent direct or indirect reads and writes only if synchronization is performed after the direct read.

Note

The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Direct write Is a write to a register, using an `MCR`, `MCRR`, or `LDC` instruction, that the architecture permits for the current PE state.

In the following cases, the side-effect of the direct write is defined to be an indirect write of the affected register, and has the synchronization requirements of an indirect write:

- If the direct write has a side-effect of changing the value of a register other than the register accessed by the direct write.
- If the direct write has a side-effect of changing the value of the register accessed by the direct write, so that the value in that register might not be the value that the direct write wrote to the register.

In both cases, this means that the indirect write is not guaranteed to be visible to subsequent direct or indirect reads and writes unless synchronization is performed after the direct write.

Note

- As an example of a direct write to a register having an effect that is an indirect write of that register, writing 1 to a PMCNTENCLR.P*x* bit is also an indirect write, because if the P*x* bit had the value 1 before the direct write, the side-effect of the write changes the value of that bit to 0.
- The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

  For example, writing 1 to a PMCNTENCLR.P*x* bit that is set to 1 also changes the corresponding PMCNTENSET.P*x* bit from 1 to 0. This means that the direct write to the PMCNTENCLR defines indirect writes to both itself and to the PMCNTENSET.

Indirect read Is a use of the register by an instruction to establish the operating conditions for the instruction. Examples of operating conditions that might be determined by an indirect read are the translation table base address, or whether memory accesses are forced to be Non-cacheable.

Indirect reads include situations where the value of one register determines what value is returned by a second register. This means that any read of the second register is an indirect read of the register that determines what value is returned.

Indirect reads also include:

- Reads of the System registers by external agents, such as debuggers, as described in Debug System registers.
- Memory-mapped reads of the System registers made by the PE on which the System registers are implemented.

Where an indirect read of a register has a side-effect of changing the value of a register, that change is defined to be an indirect write, and has the synchronization requirements of an indirect write.

Indirect write Is an update to the value of a register as a consequence of either:

- An exception, operation, or execution of an instruction that is not a direct write to that register.
- The asynchronous operation of an external agent.

This can include:

- The passage of time, as seen in counters or timers, including performance counters.
- The assertion of an interrupt.
- A write from an external agent, such as a debugger.

However, for some registers, the architecture gives some guarantee of visibility without any explicit synchronization, see Registers with some architectural guarantee of ordering or observability.

Note

Taking an exception is a Context Synchronization event. Any indirect write performed as part of an exception entry does not require additional synchronization. This includes the indirect writes to the registers that report the exception, as described in Exception reporting in a VMSAv8-32 implementation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.1.3 Principles of the ID scheme for fields in ID registers

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields,* Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. AArch32 ID registers to which this scheme applies identifies the set of ID registers that are accessible from AArch32 state.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

Note

- The ID fields described here are unlike the ones that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors.
- ID fields that do not follow this scheme include the ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, see Alternative ID scheme used for the Performance Monitors Extension version.
- The presence of an ID field for a feature does not imply that the feature is optional.

To provide forward compatibility, software can rely on the features of these fields that are described in this section.

The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, if a value of `0x1` indicates the presence of some instructions, then the value `0x2` will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```
if (value >= number) {
    // do something that relies on the value of the feature
}
```

For ID fields where the value `0x0` defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value `0xF` indicates that the feature is not implemented.
- The field value `0x0` indicates that the feature is implemented.
- Software that depends on the feature can use the test:

```
if value >= 0 {
    // Software features that depend on the presence of the hardware feature
}
```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a *fractional* field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example G8-1 shows the use of such a field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example G8-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value `0x1` was defined as indicating that item A is present.
- The value `0x2` was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value `0x1` when A and B only are present. This fractional field is valid only when the original ID field has the value `0x1`.

This approach means that:

- Software that depends on the test if `(value >= 0x2)` can rely on features A, B, and C being present,
- Software that depends on the test `if (value >= 0x1)` can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:

```
if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {
    // Software features that depend on A and B only
}
```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

G8.1.3.1 AArch32 ID registers to which this scheme applies

- The Auxiliary Feature register ID_AFR0.
- The Processor Feature registers ID_PFR0 and ID_PFR1.
- The Debug Feature register ID_DFR0.
- The Memory Model Feature registers ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4.
- The Instruction Set Attribute registers ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
- The Media and VFP Feature registers MVFR0, MVFR1, and MVFR2.

Note

Principles of the ID scheme for fields in ID registers includes information about the AArch64 System registers and the memory-mapped registers to which this scheme applies.

G8.1.3.2 Alternative ID scheme used for the Performance Monitors Extension version

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon, and EDDFR.PMUVer fields, which identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0xF` indicates that the Arm-architected Performance Monitors Extension is not implemented.
- If the field value is not `0xF` the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```
if (value != 0xF and value >= number) {
    // do something that relies on version 'number' of the feature
}
```

For these fields, Arm deprecates use of the value `0xF` in new implementations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.1.4 About AArch32 System register accesses

The following subsections give more information about accesses to the AArch32 System registers:

- Ordering of reads of System registers.
- Accessing 32-bit System registers.
- Accessing 64-bit System registers.

G8.1.4.1 Ordering of reads of System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, as specified in Synchronization of changes to AArch32 System registers, including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent Context Synchronization event to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous Context Synchronization event. For example, where a memory access is used to communicate a read of such a counter, an `ISB` must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

G8.1.4.2 Accessing 32-bit System registers

Software accesses most 32-bit System registers using the generic `MCR` and `MRC` System register access instructions, specifying some or all of the parameters {`coproc`, `CRn`, `opc1`, `CRm`, `opc2`}, where:

`coproc` Identifies the primary region of the System register encoding space. Takes one of the values:

- `p14` Encoded as `0b1110`.
- `p15` Encoded as `0b1111`.

`CRn` Takes a value in the range `c0`- `c15`, encoded the corresponding 4-bit binary value, `0b0000` - `0b1111`.

In the (`coproc`== `0b1110`) encoding space, the `opc1` value identifies the System register functional group, and `CRn` is the most significant identifier for the required register within that group.

In the (`coproc`== `0b1111`) encoding space, `CRn` is the most significant identifier for the required register.

`opc1` Takes a value in the range 0 - 7, encoded as its 3-bit binary value.

In the (`coproc`== `0b1110`) encoding space, the `opc1` value identifies the System register functional group, and can take the following values:

- `0` Debug System registers.
- `1` Trace System registers.
- `7` Legacy Jazelle System registers.

In the (`coproc`== `0b1111`) encoding space, opc1 can take any value in the range 0 - 7.

`CRm` Takes a value in the range `c0`- `c15`, encoded the corresponding 4-bit binary value, `0b0000` - `0b1111`.

`opc2` Takes a value in the range 0 - 7, encoded as its 3-bit binary value.

`opc2` is optional in the `MCR` and `MRC` instruction syntax, and if no value is specified the encoding defaults to `0b000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`Rt` A general-purpose register to hold a 32-bit value to transfer to or from the System register. Takes a value in the range `R0` - `R14`, encoded as the corresponding 4-bit binary value, `0b0000` - `0b1110`.

This means an `MCR` or `MRC` access to a specific 32-bit System register uses:

- A unique combination of `coproc`, `CRn`, `opc1`, `CRm`, and `opc2`, to specify the required System register.
- A general-purpose register, `Rt`, for the transferred 32-bit value.

See also:

- MCR.
- MRC.

A small number of AArch32 debug System registers are accessed using `LDC` or `STC` instructions. In these cases, the register to be accessed is identified in the instruction syntax by the use of `p14, c5` where:

`p14` Identifies that the access is to the (`coproc` == `0b1110`) encoding space.

`c5` Identifies the target debug System register.

See the instruction descriptions:

- LDC (immediate).
- LDC (literal).
- STC.

The only uses of `LDC` and `STC` permitted in Armv8-A are:

- An `LDC` access to load data from memory to DBGDTRTXint, see LDC (immediate) and LDC (literal).
- An `STC` access to store data to memory from DBGDTRRXint, see STC.

A small number of AArch32 System registers are accessed using `MRS`, `MSR`, `VMRS`, or `VMSR` instructions, see the appropriate register and instruction description for more information, see:

- MRS.
- MSR (immediate),
- MSR (register).
- VMRS.
- VMSR.

Note

- For example:
  - The APSR, CPSR, and SPSR are accessed using `MRS` or `MSR` instructions.
  - The MVFR0, MVFR1, and MVFR2 are accessed using `VMRS` or `VMSR` instructions.
- In addition, the banked register forms of the `MRS` and `MSR` instructions can be used to access some System registers associated with PE modes other than the mode in which the PE is currently executing, see MRS (Banked register) and MSR (Banked register).

G8.1.4.3 Accessing 64-bit System registers

Software accesses a 64-bit System register using the generic `MCRR` and `MRRC` System register access instructions, specifying the parameters {`coproc`, `CRm`, `opc1`}, where:

`coproc` Identifies the primary region of the System register encoding space. Takes one of the values:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`p14` Encoded as `0b1110`.

`p15` Encoded as `0b1111`.

`CRm` Takes a value in the range `c0`- `c15`, encoded the corresponding 4-bit binary value, `0b0000` - `0b1111`.

In the (`coproc`== `0b1110`) encoding space, the `opc1` value identifies the System register functional group, and `CRm` is the most significant identifier for the required register within that group.

In the (`coproc`== `0b1111`) encoding space, `CRm` is the most significant identifier for the required register.

`opc1` Takes a value in the range 0 - 15, encoded as its 3-bit binary value.

In the (`coproc`== `0b1110`) encoding space, the `opc1` value identifies the System register functional group, and can take the following values:

`0` Debug System registers.

`1` Trace System registers.

In the (`coproc`== `0b1111`) encoding space, opc1 can take any value in the range 0 - 15.

`Rt` A general-purpose register to hold bits[31:0] of the value to transfer to or from the System register. Takes a value in the range `R0` - `R14`, encoded as the corresponding 4-bit binary value, `0b0000` - `0b1110`.

`Rt2` A general-purpose register to hold bits[63:32] of the value to transfer to or from the System register. Takes a value in the range `R0` - `R14`, encoded as the corresponding 4-bit binary value, `0b0000` - `0b1110`.

This means an `MCRR` or `MRRC` access to a specific 64-bit System register uses:

- A unique combination of `coproc`, `CRm` and `opc1`, to specify the required 64-bit System register.
- Two general-purpose registers, each holding 32 bits of the value to transfer.

This means a PE can access a 64-bit System register using:

- An `MCRR` instruction to write to a System register, see MCRR.
- An `MRRC` instruction to read a System register, see MRRC.

When using an `MCRR` or `MRRC` instruction the System register access is 64-bit atomic.

Some 64-bit registers also have an `MCR` and `MRC` encoding. The `MCR` and `MRC` encodings for these registers access the least significant 32 bits of the register. For example, to access the PAR, software can:

- Use the following instructions to access all 64 bits of the register:

```
MRRC p15, 0, <Rt>, <Rt2>, c7    ; Read 64-bit PAR into Rt (low word) and Rt2 (high word)
```

```
MCRR p15, 0, <Rt>, <Rt2>, c7    ; Write Rt (low word) and Rt2 (high word) to 64-bit PAR
```

- Use the following instructions to access the least-significant 32 bits of the register:

```
MRC p15, 0, <Rt>, c7, c4, 0    ; Read PAR[31:0] into Rt
```

```
MCR p15, 0, <Rt>, c7, c4, 0    ; Write Rt to PAR[31:0]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2 General system control registers

This section lists the System registers in AArch32 state that are not part of one of the other listed groups.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

Configuration

Some bits might define global configuration settings, and be common to the Secure and Non-secure instances of the register.

AArch32 System register ACTLR bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL1[31:0].

This register is banked between ACTLR and ACTLR_S and ACTLR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ACTLR are UNDEFINED.

Attributes

ACTLR is a 32-bit register.

This register has the following instances:

- ACTLR, when EL3 is not implemented or FEAT_AA64 is implemented.
- ACTLR_S, when FEAT_AA32EL3 is implemented.
- ACTLR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TACR
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TAC == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = ACTLR_NS();
    else
        R(t) = ACTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = ACTLR_NS();
    else
        R(t) = ACTLR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = ACTLR_S();
    else
        R(t) = ACTLR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TACR
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TAC == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ACTLR_NS() = R(t);
    else
        ACTLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        ACTLR_NS() = R(t);
    else
        ACTLR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        ACTLR_S() = R(t);
    else
        ACTLR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.2 ACTLR2, Auxiliary Control Register 2

The ACTLR2 characteristics are:

Purpose

Provides additional space to the ACTLR register to hold IMPLEMENTATION DEFINED trap functionality for execution at EL1 and EL0.

Configuration

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining ID_MMFR4.AC2.

From Armv8.2 this register must be implemented.

AArch32 System register ACTLR2 bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL1[63:32].

This register is banked between ACTLR2 and ACTLR2_S and ACTLR2_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ACTLR2 are UNDEFINED.

Attributes

ACTLR2 is a 32-bit register.

This register has the following instances:

- ACTLR2, when EL3 is not implemented or FEAT_AA64 is implemented.
- ACTLR2_S, when FEAT_AA32EL3 is implemented.
- ACTLR2_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TACR
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TAC == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = ACTLR2_NS();
    else
        R(t) = ACTLR2();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = ACTLR2_NS();
    else
        R(t) = ACTLR2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = ACTLR2_S();
    else
        R(t) = ACTLR2_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TACR
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TAC == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ACTLR2_NS() = R(t);
    else
        ACTLR2() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        ACTLR2_NS() = R(t);
    else
        ACTLR2() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        ACTLR2_S() = R(t);
    else
        ACTLR2_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.3 ADFSR, Auxiliary Data Fault Status Register

The ADFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for Data Abort exceptions taken to EL1 modes, and EL3 modes when EL3 is implemented and is using AArch32.

Configuration

AArch32 System register ADFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR0_EL1[31:0].

This register is banked between ADFSR and ADFSR_S and ADFSR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ADFSR are UNDEFINED.

Attributes

ADFSR is a 32-bit register.

This register has the following instances:

- ADFSR, when EL3 is not implemented or FEAT_AA64 is implemented.
- ADFSR_S, when FEAT_AA32EL3 is implemented.
- ADFSR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ADFSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
            R(t) = ADFSR_NS();
        else
            R(t) = ADFSR();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
            R(t) = ADFSR_NS();
        else
            R(t) = ADFSR();
        end;
    elsif PSTATE.EL == EL3 then
        if SCR().NS == '0' then
            R(t) = ADFSR_S();
        else
            R(t) = ADFSR_NS();
        end;
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        ADFSR_NS() = R(t);
    else
        ADFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ADFSR_NS() = R(t);
    else
        ADFSR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        ADFSR_S() = R(t);
    else
        ADFSR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.4 AIDR, Auxiliary ID Register

The AIDR characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED identification information.

The value of this register must be used in conjunction with the value of MIDR.

Configuration

AArch32 System register AIDR bits [31:0] are architecturally mapped to AArch64 System register AIDR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to AIDR are UNDEFINED.

Attributes

AIDR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing AIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID1
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID1 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = AIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = AIDR();
elsif PSTATE.EL == EL3 then
    R(t) = AIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.5 AIFSR, Auxiliary Instruction Fault Status Register

The AIFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for Prefetch Abort exceptions taken to EL1 modes, and EL3 modes when EL3 is implemented and is using AArch32.

Configuration

AArch32 System register AIFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR1_EL1[31:0].

This register is banked between AIFSR and AIFSR_S and AIFSR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to AIFSR are UNDEFINED.

Attributes

AIFSR is a 32-bit register.

This register has the following instances:

- AIFSR, when EL3 is not implemented or FEAT_AA64 is implemented.
- AIFSR_S, when FEAT_AA32EL3 is implemented.
- AIFSR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AIFSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = AIFSR_NS();
    else
        R(t) = AIFSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = AIFSR_NS();
    else
        R(t) = AIFSR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = AIFSR_S();
    else
        R(t) = AIFSR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        AIFSR_NS() = R(t);
    else
        AIFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AIFSR_NS() = R(t);
    else
        AIFSR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        AIFSR_S() = R(t);
    else
        AIFSR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.6 AMAIR0, Auxiliary Memory Attribute Indirection Register 0

The AMAIR0 characteristics are:

Purpose

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR0.

Configuration

AArch32 System register AMAIR0 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL1[31:0].

This register is banked between AMAIR0 and AMAIR0_S and AMAIR0_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to AMAIR0 are UNDEFINED.

Attributes

AMAIR0 is a 32-bit register.

This register has the following instances:

- AMAIR0, when EL3 is not implemented or FEAT_AA64 is implemented.
- AMAIR0_S, when FEAT_AA32EL3 is implemented.
- AMAIR0_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR0(S) gives the value for memory accesses from Secure state.
- AMAIR0(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = AMAIR0_NS();
    else
        R(t) = AMAIR0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = AMAIR0_NS();
    else
        R(t) = AMAIR0();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = AMAIR0_S();
    else
        R(t) = AMAIR0_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        AMAIR0_NS() = R(t);
    else
        AMAIR0() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        AMAIR0_NS() = R(t);
    else
        AMAIR0() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            AMAIR0_S() = R(t);
        else
            AMAIR0_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.7 AMAIR1, Auxiliary Memory Attribute Indirection Register 1

The AMAIR1 characteristics are:

Purpose

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR1.

Configuration

When EL3 is using AArch32, write access to AMAIR1(S) is disabled when the **CP15SDISABLE** signal is asserted HIGH.

AArch32 System register AMAIR1 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL1[63:32].

This register is banked between AMAIR1 and AMAIR1_S and AMAIR1_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to AMAIR1 are UNDEFINED.

Attributes

AMAIR1 is a 32-bit register.

This register has the following instances:

- AMAIR1, when EL3 is not implemented or FEAT_AA64 is implemented.
- AMAIR1_S, when FEAT_AA32EL3 is implemented.
- AMAIR1_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR1(S) gives the value for memory accesses from Secure state.
- AMAIR1(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMAIR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = AMAIR1_NS();
    else
        R(t) = AMAIR1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = AMAIR1_NS();
    else
        R(t) = AMAIR1();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = AMAIR1_S();
    else
        R(t) = AMAIR1_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        AMAIR1_NS() = R(t);
    else
        AMAIR1() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        AMAIR1_NS() = R(t);
    else
        AMAIR1() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            AMAIR1_S() = R(t);
        else
            AMAIR1_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.8 APSR, Application Program Status Register

The APSR characteristics are:

Purpose

Hold program status and control information.

Note

Some of the fields in this register are permitted to return the value of the PSTATE field on a read. This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

For more information see ‘The Application Program Status Register, APSR’.

Configuration

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to APSR are UNDEFINED.

Attributes

APSR is a 32-bit register.

Field descriptions

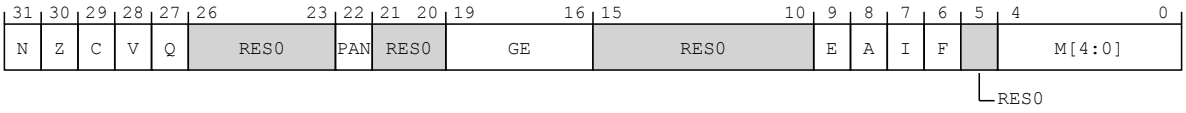

N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two’s complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Bits [26:23]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. This field is UNKNOWN, but is permitted to return the value of PSTATE.PAN field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [21:20]

Reserved, RES0.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:10]

Reserved, RES0.

E, bit [9]

Endianness. This field is UNKNOWN, but is permitted to return the value of PSTATE.E field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.A field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.I field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.F field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4:0], bits [4:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mode. This field is UNKNOWN, but is permitted to return the value of PSTATE.M[4:0] field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APSR

APSR can be read using the MRS instruction and written using the MSR (register) or MSR (immediate) instructions.

The MSR (register) and MRS instructions used to access APSR are data-independent-time instructions as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.9 ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read

The ATS12NSOPR characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to ATS12NSOPR are UNDEFINED.

Attributes

ATS12NSOPR is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOPR

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL1, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_12, EL1, ATAccess_Read);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.10 ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write

The ATS12NSOPW characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to ATS12NSOPW are UNDEFINED.

Attributes

ATS12NSOPW is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOPW

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_12, EL1, ATAccess_Write);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.11 ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read

The ATS12NSOUR characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to ATS12NSOUR are UNDEFINED.

Attributes

ATS12NSOUR is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOUR

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL0, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_12, EL0, ATAccess_Read);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.12 ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write

The ATS12NSOUW characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to ATS12NSOUW are UNDEFINED.

Attributes

ATS12NSOUW is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOUW

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL0, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_12, EL0, ATAccess_Write);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.13 ATS1CPR, Address Translate Stage 1 Current state PL1 Read

The ATS1CPR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ATS1CPR are UNDEFINED.

Attributes

ATS1CPR is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPR

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Read);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Read);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        AArch32_AT(R(t), TranslationStage_1, EL3, ATAccess_Read);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Read);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.14 ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN

The ATS1CPRP characteristics are:

Purpose

Performs a stage 1 address translation at PL1 and in the current Security state, where the value of PSTATE.PAN determines if a read from a location will generate a Permission fault for a privileged access.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented and FEAT_PAN2 is implemented. Otherwise, direct accesses to ATS1CPRP are UNDEFINED.

Attributes

ATS1CPRP is a 32-bit System instruction.

Field descriptions

31 … 0
Input address for translation

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPRP

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1001	0b000

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PAN2)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_ReadPAN);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_ReadPAN);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        AArch32_AT(R(t), TranslationStage_1, EL3, ATAccess_ReadPAN);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_ReadPAN);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.15 ATS1CPW, Address Translate Stage 1 Current state PL1 Write

The ATS1CPW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ATS1CPW are UNDEFINED.

Attributes

ATS1CPW is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPW

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Write);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Write);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        AArch32_AT(R(t), TranslationStage_1, EL3, ATAccess_Write);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Write);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.16 ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN

The ATS1CPWP characteristics are:

Purpose

Performs a stage 1 address translation at PL1 and in the current Security state, where the value of PSTATE.PAN determines if a write to the location will generate a Permission fault for a privileged access.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented and FEAT_PAN2 is implemented. Otherwise, direct accesses to ATS1CPWP are UNDEFINED.

Attributes

ATS1CPWP is a 32-bit System instruction.

Field descriptions

31 … 0
Input address for translation

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPWP

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1001	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PAN2)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_WritePAN);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_WritePAN);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        AArch32_AT(R(t), TranslationStage_1, EL3, ATAccess_WritePAN);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_WritePAN);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.17 ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read

The ATS1CUR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ATS1CUR are UNDEFINED.

Attributes

ATS1CUR is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CUR

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Read);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Read);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Read);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.18 ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write

The ATS1CUW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ATS1CUW are UNDEFINED.

Attributes

ATS1CUW is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CUW

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Write);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Write);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Write);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.19 ATS1HR, Address Translate Stage 1 Hyp mode Read

The ATS1HR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if reading from the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to ATS1HR are UNDEFINED.

Attributes

ATS1HR is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

Executing ATS1HR

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0111	0b1000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL2, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_1, EL2, ATAccess_Read);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.20 ATS1HW, Address Translate Stage 1 Hyp mode Write

The ATS1HW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if writing to the given virtual address.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to ATS1HW are UNDEFINED.

Attributes

ATS1HW is a 32-bit System instruction.

Field descriptions

31 … 0
Input address for translation

IA, bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

Executing ATS1HW

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0111	0b1000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL2, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch32_AT(R(t), TranslationStage_1, EL2, ATAccess_Write);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.21 BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

Purpose

Invalidate all entries from branch predictors.

Configuration

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to BPIALL are UNDEFINED.

Attributes

BPIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing BPIALL

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When HCR.FB is 1, at Non-secure EL1 this instruction executes as a BPIALLIS.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().FB == '1'
        ↪then
        BPIALLIS();
    else
        BPIALL();
    end;
elsif PSTATE.EL == EL2 then
    BPIALL();
elsif PSTATE.EL == EL3 then
    BPIALL();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.22 BPIALLIS, Branch Predictor Invalidate All, Inner Shareable

The BPIALLIS characteristics are:

Purpose

Invalidate all entries from branch predictors Inner Shareable.

Configuration

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to BPIALLIS are UNDEFINED.

Attributes

BPIALLIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing BPIALLIS

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0001	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        BPIALLIS();
    end;
elsif PSTATE.EL == EL2 then
    BPIALLIS();
elsif PSTATE.EL == EL3 then
    BPIALLIS();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.23 BPIMVA, Branch Predictor Invalidate by VA

The BPIMVA characteristics are:

Purpose

Invalidate virtual address from branch predictors.

Configuration

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to BPIMVA are UNDEFINED.

Attributes

BPIMVA is a 32-bit System instruction.

Field descriptions

31 … 0
Virtual address to use

VA, bits [31:0]

Virtual address to use.

Executing BPIMVA

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        BPIMVA(R(t));
    end;
elsif PSTATE.EL == EL2 then
    BPIMVA(R(t));
elsif PSTATE.EL == EL3 then
    BPIMVA(R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.24 CCSIDR, Current Cache Size ID Register

The CCSIDR characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

When FEAT_CCIDX is implemented, this register is used in conjunction with CCSIDR2.

Configuration

The implementation includes one CCSIDR for each cache that it can access. CSSELR and the Security state select which Cache Size ID Register is accessible.

AArch32 System register CCSIDR bits [31:0] are architecturally mapped to AArch64 System register CCSIDR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CCSIDR are UNDEFINED.

Attributes

CCSIDR is a 32-bit register.

Field descriptions

When FEAT_CCIDX is implemented:

31 – 24	23 – 3	2 – 0
RES0	Associativity	LineSize

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [31:24]

Reserved, RES0.

Associativity, bits [23:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

31–28	27–13	12–3	2–0
UNKNOWN	NumSets	Associativity	LineSize

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

Accessing CCSIDR

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR read is treated as NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID2
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR2().TID4 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CCSIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CCSIDR();
elsif PSTATE.EL == EL3 then
    R(t) = CCSIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.25 CCSIDR2, Current Cache Size ID Register 2

The CCSIDR2 characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

Configuration

The implementation includes one CCSIDR2 for each cache that it can access. CSSELR and the Security state select which Cache Size ID Register is accessible.

AArch32 System register CCSIDR2 bits [31:0] are architecturally mapped to AArch64 System register CCSIDR2_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_CCIDX is implemented. Otherwise, direct accesses to CCSIDR2 are UNDEFINED.

Attributes

CCSIDR2 is a 32-bit register.

Field descriptions

31 – 24	23 – 0
RES0	NumSets

Bits [31:24]

Reserved, RES0.

NumSets, bits [23:0]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Accessing CCSIDR2

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR2 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2 read is treated as NOP.
- The CCSIDR2 read is UNDEFINED.
- The CCSIDR2 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_CCIDX)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID2
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR2().TID4 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CCSIDR2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CCSIDR2();
elsif PSTATE.EL == EL3 then
    R(t) = CCSIDR2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.26 CFPRCTX, Control Flow Prediction Restriction by Context

The CFPRCTX characteristics are:

Purpose

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_AA32 is implemented and FEAT_SPECRES is implemented. Otherwise, direct accesses to CFPRCTX are UNDEFINED.

Attributes

CFPRCTX is a 32-bit System instruction.

Field descriptions

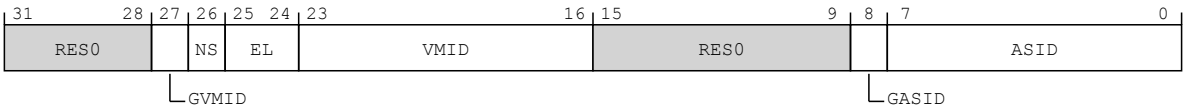

Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
0b00	EL0.	
0b01	EL1	
0b10	EL2	FEAT_EL2 is implemented
0b11	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and the target execution context is either:

- EL1.
- EL0 when EL2 is using AArch32 or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0, this field is treated as the current VMID if any of the following are true:

- EL2 is using AArch32.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.
0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field is treated as 0.

ASID, bits [7:0]

Only applies for an EL0 target execution context and when bit[8] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

Executing CFPRCTX

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b100

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_SPECRES)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGITR_EL2().CFPRCTX == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch32_RestrictPrediction(R(t), RestrictType_ControlFlow);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_ControlFlow);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_RestrictPrediction(R(t), RestrictType_ControlFlow);
elsif PSTATE.EL == EL3 then
    AArch32_RestrictPrediction(R(t), RestrictType_ControlFlow);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.27 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

Purpose

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

Configuration

AArch32 System register CLIDR bits [31:0] are architecturally mapped to AArch64 System register CLIDR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CLIDR are UNDEFINED.

Attributes

CLIDR is a 32-bit register.

Field descriptions

31 30	29 27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
ICB	LoUU	LoC	LoUIS	Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1

ICB, bits [31:30]

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ICB	Meaning
0b00	Not disclosed by this mechanism.
0b01	L1 cache is the highest Inner Cacheable level.
0b10	L2 cache is the highest Inner Cacheable level.
0b11	L3 cache is the highest Inner Cacheable level.

Access to this field is RO.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

This field does not describe the requirements for instruction cache invalidation. See CTR.DIC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoC, bits [26:24]

Level of Coherence for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

This field does not describe the requirements for instruction cache invalidation. See CTR.DIC.

Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Ctype<n> , bits [3(n-1)+2:3(n-1)], for n = 7 to 1

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy.

Ctype<n>	Meaning
`0b000`	No cache.
`0b001`	Instruction cache only.
`0b010`	Data cache only.
`0b011`	Separate instruction and data caches.
`0b100`	Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of `0b000`, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of `0b000`, the values of Ctype4 to Ctype7 must be ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CLIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID2
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR2().TID4 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CLIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CLIDR();
elsif PSTATE.EL == EL3 then
    R(t) = CLIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.28 CONTEXTIDR, Context ID Register

The CONTEXTIDR characteristics are:

Purpose

Identifies the current Process Identifier and, when using the Short-descriptor translation table format, the Address Space Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configuration

The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

AArch32 System register CONTEXTIDR bits [31:0] are architecturally mapped to AArch64 System register CONTEXTIDR_EL1[31:0].

This register is banked between CONTEXTIDR and CONTEXTIDR_S and CONTEXTIDR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CONTEXTIDR are UNDEFINED.

Attributes

CONTEXTIDR is a 32-bit register.

This register has the following instances:

- CONTEXTIDR, when EL3 is not implemented or FEAT_AA64 is implemented.
- CONTEXTIDR_S, when FEAT_AA32EL3 is implemented.
- CONTEXTIDR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

31 … 8	7 … 0
PROCID	ASID

PROCID, bits [31:8]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ASID, bits [7:0]

Address Space Identifier. This field is programmed with the value of the current ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TTBCR.EAE == '1':

31	0
PROCID	

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CONTEXTIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CONTEXTIDR_NS();
    else
        R(t) = CONTEXTIDR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CONTEXTIDR_NS();
    else
        R(t) = CONTEXTIDR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = CONTEXTIDR_S();
    else
        R(t) = CONTEXTIDR_NS();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CONTEXTIDR_NS() = R(t);
    else
        CONTEXTIDR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CONTEXTIDR_NS() = R(t);
    else
        CONTEXTIDR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CONTEXTIDR_S() = R(t);
    else
        CONTEXTIDR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.29 COSPRCTX, Clear Other Speculative Prediction Restriction by Context

The COSPRCTX characteristics are:

Purpose

Clear Other Speculative Prediction Restriction by Context applies to prediction resources not managed by other speculation restriction System instructions.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any predictions occurring after the instruction is complete and synchronized.

This instruction applies to all speculative access except:

- Cache Prefetch predictions.
- Control Flow predictions.
- Data Value predictions.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the PE that executed the original restriction instruction, and a subsequent Context Synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations, the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_AA32 is implemented and FEAT_SPECRES2 is implemented. Otherwise, direct accesses to COSPRCTX are UNDEFINED.

Attributes

COSPRCTX is a 32-bit System instruction.

Field descriptions

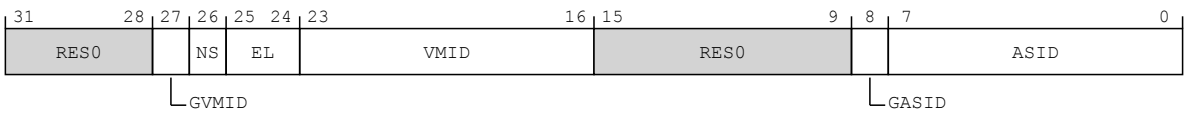

Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

If the instruction is executed in Non-secure state, this field is treated as 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
0b00	EL0.	
0b01	EL1	
0b10	EL2	FEAT_EL2 is implemented
0b11	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and the target execution context is either:

- EL1.
- EL0 when EL2 is using AArch32 or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0, this field is treated as the current VMID if any of the following are true:

- EL2 is using AArch32.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs, or a specified ASID.

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.
0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [7:0]

Only applies to an EL0 target execution context and when bit[8] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

Executing COSPRCTX

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b110

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_SPECRES2)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGITR_EL2().COSPRCTX == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_Other);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_Other);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_RestrictPrediction(R(t), RestrictType_Other);
elsif PSTATE.EL == EL3 then
    AArch32_RestrictPrediction(R(t), RestrictType_Other);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.30 CP15DMB, Data Memory Barrier System instruction

The CP15DMB characteristics are:

Purpose

Performs a Data Memory Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DMB instruction instead.

Configuration

This system instruction is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CP15DMB are UNDEFINED.

Attributes

CP15DMB is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing CP15DMB

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b101

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪SCTLR_EL1().CP15BEN == '0' then
        Undefined();
    elsif ELIsInHost(EL0) && SCTLR_EL2().CP15BEN == '0' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().CP15BEN == '0' then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CP15DMB();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif SCTLR().CP15BEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        CP15DMB();
    end;
elsif PSTATE.EL == EL2 then
    if HSCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15DMB();
    end;
elsif PSTATE.EL == EL3 then
    if SCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15DMB();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.31 CP15DSB, Data Synchronization Barrier System instruction

The CP15DSB characteristics are:

Purpose

Performs a Data Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DSB instruction instead.

Configuration

This system instruction is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CP15DSB are UNDEFINED.

Attributes

CP15DSB is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing CP15DSB

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b100

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪SCTLR_EL1().CP15BEN == '0' then
        Undefined();
    elsif ELIsInHost(EL0) && SCTLR_EL2().CP15BEN == '0' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().CP15BEN == '0' then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CP15DSB();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif SCTLR().CP15BEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        CP15DSB();
    end;
elsif PSTATE.EL == EL2 then
    if HSCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15DSB();
    end;
elsif PSTATE.EL == EL3 then
    if SCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15DSB();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.32 CP15ISB, Instruction Synchronization Barrier System instruction

The CP15ISB characteristics are:

Purpose

Performs an Instruction Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the ISB instruction instead.

Configuration

This system instruction is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CP15ISB are UNDEFINED.

Attributes

CP15ISB is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing CP15ISB

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b100

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪SCTLR_EL1().CP15BEN == '0' then
        Undefined();
    elsif ELIsInHost(EL0) && SCTLR_EL2().CP15BEN == '0' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().CP15BEN == '0' then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CP15ISB();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif SCTLR().CP15BEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        CP15ISB();
    end;
elsif PSTATE.EL == EL2 then
    if HSCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15ISB();
    end;
elsif PSTATE.EL == EL3 then
    if SCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15ISB();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.33 CPACR, Architectural Feature Access Control Register

The CPACR characteristics are:

Purpose

Controls access to trace, and to Advanced SIMD and floating-point functionality from EL0, EL1, and EL3.

In an implementation that includes EL2, the CPACR has no effect on instructions executed at EL2.

Configuration

Bits in the NSACR control Non-secure access to the CPACR fields. See the field descriptions for more information.

Note

In the register field descriptions, controls are described as applying at specified Privilege levels. This is because, in Secure state, a PL1 control:

- Applies to execution in a Secure EL3 mode when EL3 is using AArch32.
- Applies to execution in a Secure EL1 mode when EL3 is using AArch64.

See 'Security state, Exception levels, and AArch32 execution privilege'.

AArch32 System register CPACR bits [31:0] are architecturally mapped to AArch64 System register CPACR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CPACR are UNDEFINED.

Attributes

CPACR is a 32-bit register.

Field descriptions

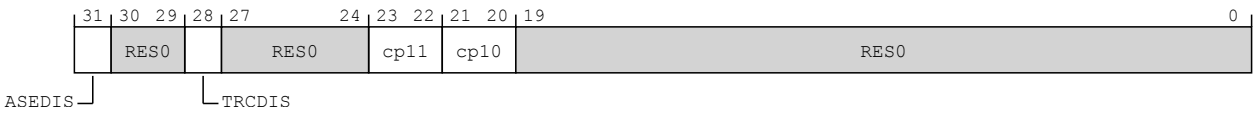

ASEDIS, bit [31]

Disables PL0 and PL1 execution of Advanced SIMD instructions.

ASEDIS	Meaning
0b0	This control permits execution of Advanced SIMD instructions at PL0 and PL1.
0b1	All instruction encodings that are Advanced SIMD instruction encodings, but are not also floating-point instruction encodings, are UNDEFINED at PL0 and PL1.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the list of instructions affected by this field, see 'Controls of Advanced SIMD operation that do not apply to floating-point operation'.

See the description of CPACR.cp10 for a list of other controls that can disable or trap execution of Advanced SIMD instructions in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bits [30:29]

Reserved, RES0.

TRCDIS, bit [28]

Traps PL0 and PL1 System register accesses to all implemented trace System registers to Undefined mode.

TRCDIS	Meaning
`0b0`	This control has no effect on PL0 and PL1 System register accesses to trace System registers.
`0b1`	PL0 and PL1 System register accesses to all implemented trace System registers are trapped to Undefined mode.

If the implementation does not include a trace unit, or does not include a System register interface to the trace unit registers, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

Note

- FEAT_ETMv4 and FEAT_ETE do not permit EL0 to access the trace System registers. EL0 accesses to the trace System registers are UNDEFINED.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:24]

Reserved, RES0.

cp11, bits [23:22]

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the CPACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'00'`.

When EL3 is implemented, EL3 is using AArch32, !IsCurrentSecurityState(SS_Secure), and NSACR.cp10 == '0', access to this field is RAZ/WI.

cp10, bits [21:20]

Defines the access rights for the Advanced SIMD and floating-point functionality. Possible values of the field are:

cp10	Meaning
`0b00`	PL0 and PL1 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
`0b01`	PL0 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
`0b10`	Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE choice between: • The value of this field is treated as `0b00`, `0b01`, or `0b11` for all purposes other than reading the value of the field. • PL1 accesses to floating-point and Advanced SIMD registers or instructions are UNDEFINED.
`0b11`	This control permits full access to the Advanced SIMD and floating-point functionality from PL0 and PL1.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

Note

The CPACR has no effect on Advanced SIMD and floating-point accesses from PL2. These can be disabled by the HCPTR.TCP10 field.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSASEDIS.

See the descriptions of the controls for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'00'`.

When EL3 is implemented, EL3 is using AArch32, !IsCurrentSecurityState(SS_Secure), and NSACR.cp10 == '0', access to this field is RAZ/WI.

Bits [19:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CPACR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TCPAC
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TCPAC ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TCPAC
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = CPACR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TCPAC
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = CPACR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = CPACR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TCPAC
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TCPAC ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TCPAC
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        CPACR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TCPAC
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        CPACR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    CPACR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.34 CPPRCTX, Cache Prefetch Prediction Restriction by Context

The CPPRCTX characteristics are:

Purpose

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

This instruction applies to all:

- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_AA32 is implemented and FEAT_SPECRES is implemented. Otherwise, direct accesses to CPPRCTX are UNDEFINED.

Attributes

CPPRCTX is a 32-bit System instruction.

Field descriptions

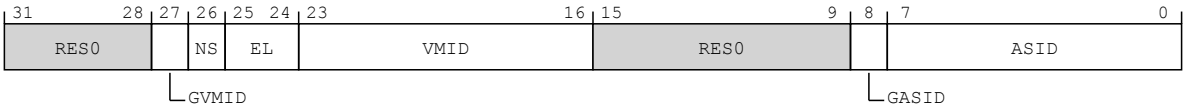

Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

If the instruction is executed in Non-secure state, this field is treated as 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
0b00	EL0.	
0b01	EL1	
0b10	EL2	FEAT_EL2 is implemented
0b11	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and the target execution context is either:

- EL1.
- EL0 when EL2 is using AArch32 or the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0, this field is treated as the current VMID if any of the following are true:

- EL2 is using AArch32.
- The Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.
0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [7:0]

Only applies for an EL0 target execution context and when bit[8] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

Executing CPPRCTX

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b111

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_SPECRES)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGITR_EL2().CPPRCTX == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_CachePrefetch);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_CachePrefetch);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_RestrictPrediction(R(t), RestrictType_CachePrefetch);
elsif PSTATE.EL == EL3 then
    AArch32_RestrictPrediction(R(t), RestrictType_CachePrefetch);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.35 CPSR, Current Program Status Register

The CPSR characteristics are:

Purpose

Holds PE status and control information.

Configuration

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CPSR are UNDEFINED.

Attributes

CPSR is a 32-bit register.

Field descriptions

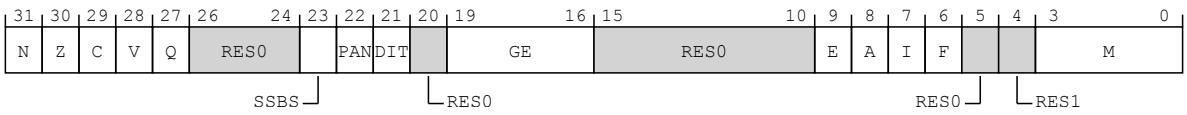

N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Bits [26:24]

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass Safe.

Prohibits speculative loads or stores that might practically allow a cache timing side channel.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

SSBS	Meaning
0b0	Hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.
0b1	When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

The value of this bit is usually set to the value described by the SCTLR.DSSBS bit on exceptions to any mode except Hyp mode, and the value described by HSCTLR.DSSBS on exceptions to Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never.

PAN	Meaning
0b0	The translation system is the same as Armv8.0.
0b1	Disables privileged read and write accesses to addresses accessible at EL0.

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the SCTLR.SPAN bit for the current Security state is 0, this bit is set to 1.
- When the target of the exception is EL3, from Secure state, and the value of the Secure SCTLR.SPAN is 0, this bit is set to 1.
- When the target of the exception is EL3, from Non-secure state, this bit is set to 0 regardless of the value of the Secure SCTLR.SPAN bit.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DIT	Meaning
0b0	The architecture makes no statement about the timing properties of any instructions.
0b1	The architecture requires that the execution time of a data-independent-time sequence of code must be independent of all data-independent-time values. For more information, see About the DIT bit.

The Operational Information section of a data processing instruction description indicates if that instruction is affected by this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:10]

Reserved, RES0.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

E	Meaning
0b0	Little-endian operation
0b1	Big-endian operation.

Instruction fetches ignore this bit.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

A, bit [8]

SError exception mask bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A	Meaning
0b0	Exception not masked.
0b1	Exception masked.

I, bit [7]

IRQ mask bit.

I	Meaning
0b0	Exception not masked.
0b1	Exception masked.

F, bit [6]

FIQ mask bit.

F	Meaning
0b0	Exception not masked.
0b1	Exception masked.

Bit [5]

Reserved, RES0.

Bit [4]

Reserved, RES1.

M, bits [3:0]

Current PE mode.

M	Meaning
0b0000	User.
0b0001	FIQ.
0b0010	IRQ.
0b0011	Supervisor.
0b0110	Monitor.
0b0111	Abort.
0b1010	Hyp.
0b1011	Undefined.
0b1111	System.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CPSR

CPSR can be read using the MRS instruction and written using the MSR (register) or MSR (immediate) instructions.

The MSR (register) and MRS instructions used to access CPSR are data-independent-time instructions as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.36 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

Purpose

Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type, which is either instruction cache or data cache.

If FEAT_CCIDX is implemented, CSSELR also selects the current CCSIDR2.

Configuration

AArch32 System register CSSELR bits [31:0] are architecturally mapped to AArch64 System register CSSELR_EL1[31:0].

This register is banked between CSSELR and CSSELR_S and CSSELR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CSSELR are UNDEFINED.

Attributes

CSSELR is a 32-bit register.

This register has the following instances:

- CSSELR, when EL3 is not implemented or FEAT_AA64 is implemented.
- CSSELR_S, when FEAT_AA32EL3 is implemented.
- CSSELR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

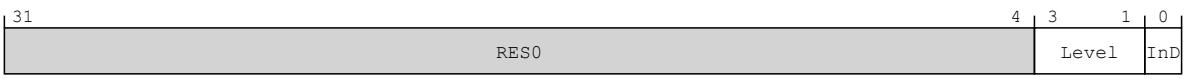

Bits [31:4]

Reserved, RES0.

Level, bits [3:1]

Cache level of required cache. Permitted values are:

Level	Meaning
0b000	Level 1 cache.
0b001	Level 2 cache.
0b010	Level 3 cache.
0b011	Level 4 cache.
0b100	Level 5 cache.
0b101	Level 6 cache.
0b110	Level 7 cache.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

InD, bit [0]

Instruction not Data bit. Permitted values are:

InD	Meaning
0b0	Data or unified cache.
0b1	Instruction cache.

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CSSELR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b010	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID2
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR2().TID4 == '1' then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CSSELR_NS();
    else
        R(t) = CSSELR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CSSELR_NS();
    else
        R(t) = CSSELR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = CSSELR_S();
    else
        R(t) = CSSELR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b010	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID2
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR_EL2().TID4 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_EVT) && HCR2().TID4 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CSSELR_NS() = R(t);
    else
        CSSELR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CSSELR_NS() = R(t);
    else
        CSSELR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CSSELR_S() = R(t);
    else
        CSSELR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.37 CTR, Cache Type Register

The CTR characteristics are:

Purpose

Provides information about the architecture of the caches.

Configuration

AArch32 System register CTR bits [31:0] are architecturally mapped to AArch64 System register CTR_EL0[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CTR are UNDEFINED.

Attributes

CTR is a 32-bit register.

Field descriptions

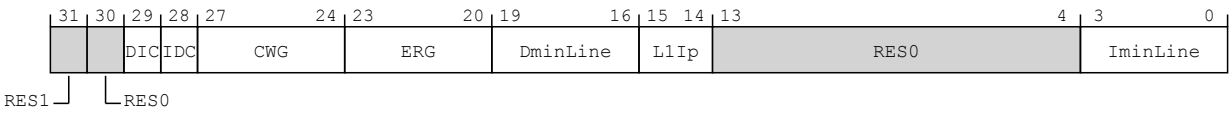

Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

DIC, bit [29]

Instruction cache invalidation requirements for data to instruction coherence.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DIC	Meaning
0b0	Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.
0b1	Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.

All PEs in the same Inner Shareable shareability domain must have a common value of this field.

Access to this field is RO.

IDC, bit [28]

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDC	Meaning
0b0	Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR.LoC == 0b000 or (CLIDR.LoUIS == 0b000 and CLIDR.LoUU == 0b000).
0b1	Data cache clean to the Point of Unification is not required for instruction to data coherence.

If CTR.DIC is 1, then the value reported in this field must be 1.

The Effective value of IDC is 1 if any of the following are true:

- CTR.IDC == 1.
- CLIDR.LoC == 0b000.
- CLIDR.LoUIS == 0b000 and CLIDR.LoUU == 0b000.

All PEs in the same Inner Shareable shareability domain must have a common Effective value of IDC.

Access to this field is RO.

CWG, bits [27:24]

Cache writeback granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of 0b0000 indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

Arm recommends that an implementation that does not support cache write-back implements this field as 0b0001. This applies, for example, to an implementation that supports only write-through caches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

ERG, bits [23:20]

Exclusives reservation granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value 0b0000 is deprecated.

The value 0b0001 and values greater than 0b1001 are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L1Ip	Meaning
0b00	Reserved.
0b01	ASID-tagged Virtual Index, Virtual Tag (AIVIVT).
0b10	Virtual Index, Physical Tag (VIPT).
0b11	Physical Index, Physical Tag (PIPT).

From Armv8.0, the value 0b01 is not permitted.

Access to this field is RO.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID2
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CTR();
elsif PSTATE.EL == EL3 then
    R(t) = CTR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.38 DACR, Domain Access Control Register

The DACR characteristics are:

Purpose

Defines the access permission for each of the sixteen memory domains.

Configuration

This register has no function when TTBCR.EAE is set to 1, to select the Long-descriptor translation table format.

AArch32 System register DACR bits [31:0] are architecturally mapped to AArch64 System register DACR32_EL2[31:0].

This register is banked between DACR and DACR_S and DACR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DACR are UNDEFINED.

Attributes

DACR is a 32-bit register.

This register has the following instances:

- DACR, when EL3 is not implemented or FEAT_AA64 is implemented.
- DACR_S, when FEAT_AA32EL3 is implemented.
- DACR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

D<n> , bits [2n+1:2n], for n = 15 to 0

Domain n access permission, where n = 0 to 15. Permitted values are:

D<n>	Meaning
`0b00`	No access. Any access to the domain generates a Domain fault.
`0b01`	Client. Accesses are checked against the permission bits in the translation tables.
`0b11`	Manager. Accesses are not checked against the permission bits in the translation tables.

The value `0b10` is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DACR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0011	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T3 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = DACR_NS();
    else
        R(t) = DACR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = DACR_NS();
    else
        R(t) = DACR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = DACR_S();
    else
        R(t) = DACR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0011	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T3 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T3 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        DACR_NS() = R(t);
    else
        DACR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        DACR_NS() = R(t);
    else
        DACR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            DACR_S() = R(t);
        else
            DACR_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.39 DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC

The DCCIMVAC characteristics are:

Purpose

Clean and Invalidate data or unified cache line by virtual address to PoC.

Configuration

AArch32 System instruction DCCIMVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCCIMVAC are UNDEFINED.

Attributes

DCCIMVAC is a 32-bit System instruction.

Field descriptions

31 … 0
Virtual address to use

VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCCIMVAC

Execution of this instruction might require an address translation from VA to PA, and that translation might fault.

For more information about faults, see 'Permission fault'.

For more information about data cache maintenance instructions, see 'AArch32 data cache maintenance instructions (DC*)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1110	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapDC(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPCP
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPC == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatDCAsNOP(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapDC(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch32_TreatDCAsNOP(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch32_TreatDCAsNOP(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.40 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

Purpose

Clean and Invalidate data or unified cache line by set/way.

Configuration

AArch32 System instruction DCCISW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCCISW are UNDEFINED.

Attributes

DCCISW is a 32-bit System instruction.

Field descriptions

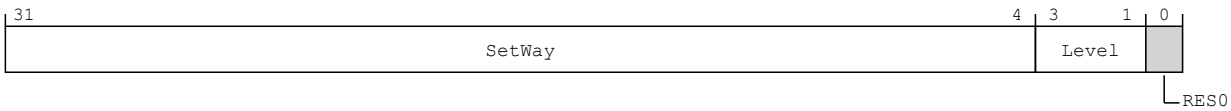

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCCISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1110	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TSW ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TSW == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.41 DCCMVAC, Data Cache line Clean by VA to PoC

The DCCMVAC characteristics are:

Purpose

Clean data or unified cache line by virtual address to PoC.

Configuration

AArch32 System instruction DCCMVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCCMVAC are UNDEFINED.

Attributes

DCCMVAC is a 32-bit System instruction.

Field descriptions

31 … 0
Virtual address to use

VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCCMVAC

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'AArch32 data cache maintenance instructions (DC*)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapDC(CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPCP
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPC == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatDCAsNOP(CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapDC(CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch32_TreatDCAsNOP(CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch32_TreatDCAsNOP(CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.42 DCCMVAU, Data Cache line Clean by VA to PoU

The DCCMVAU characteristics are:

Purpose

Clean data or unified cache line by virtual address to PoU.

Configuration

AArch32 System instruction DCCMVAU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCCMVAU are UNDEFINED.

Attributes

DCCMVAU is a 32-bit System instruction.

Field descriptions

31 … 0
Virtual address to use

VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCCMVAU

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'AArch32 data cache maintenance instructions (DC*)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPU ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TOCU
        ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPU == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TOCU ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoU);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL3 then
    AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoU);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.43 DCCSW, Data Cache line Clean by Set/Way

The DCCSW characteristics are:

Purpose

Clean data or unified cache line by set/way.

Configuration

AArch32 System instruction DCCSW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCCSW are UNDEFINED.

Attributes

DCCSW is a 32-bit System instruction.

Field descriptions

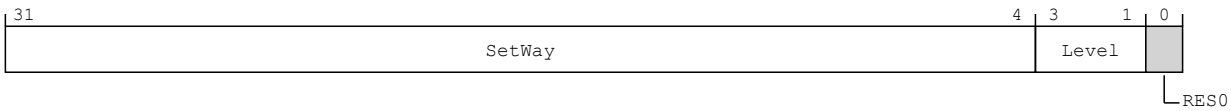

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCCSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TSW ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TSW == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.44 DCIMVAC, Data Cache line Invalidate by VA to PoC

The DCIMVAC characteristics are:

Purpose

Invalidate data or unified cache line by virtual address to PoC.

Configuration

AArch32 System instruction DCIMVAC bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCIMVAC are UNDEFINED.

Attributes

DCIMVAC is a 32-bit System instruction.

Field descriptions

31	0
Virtual address to use	

VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCIMVAC

It is IMPLEMENTATION DEFINED whether, when this instruction is executed, it can generate a watchpoint. If this instruction can generate a watchpoint this is prioritized in the same way as other watchpoints.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'AArch32 data cache maintenance instructions (DC*)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapDC(CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPCP
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPC == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatDCAsNOP(CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapDC(CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch32_TreatDCAsNOP(CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch32_TreatDCAsNOP(CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_PoC);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.45 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

Purpose

Invalidate data or unified cache line by set/way.

Configuration

AArch32 System instruction DCISW bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DCISW are UNDEFINED.

Attributes

DCISW is a 32-bit System instruction.

Field descriptions

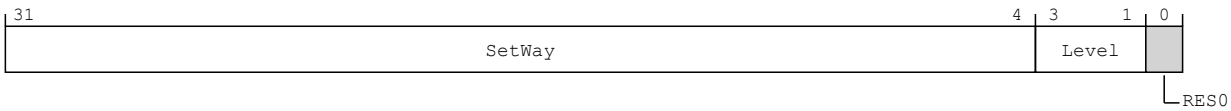

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log_2(ASSOCIATIVITY), L = Log_2(LINELEN), B = (L + S), S = Log_2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TSW ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TSW == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_SetWay);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.46 DFAR, Data Fault Address Register

The DFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception.

Configuration

AArch32 System register DFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL1[31:0].

When FEAT_AA32EL2 is implemented and FEAT_AA32EL3 is implemented, AArch32 System register DFAR bits [31:0] are architecturally mapped to AArch32 System register HDFAR[31:0].

This register is banked between DFAR and DFAR_S and DFAR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DFAR are UNDEFINED.

Attributes

DFAR is a 32-bit register.

This register has the following instances:

- DFAR, when EL3 is not implemented or FEAT_AA64 is implemented.
- DFAR_S, when FEAT_AA32EL3 is implemented.
- DFAR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
VA of faulting address of synchronous Data Abort exception

VA, bits [31:0]

VA of faulting address of synchronous Data Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DFAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = DFAR_NS();
    else
        R(t) = DFAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = DFAR_NS();
    else
        R(t) = DFAR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = DFAR_S();
    else
        R(t) = DFAR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        DFAR_NS() = R(t);
    else
        DFAR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        DFAR_NS() = R(t);
    else
        DFAR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        DFAR_S() = R(t);
    else
        DFAR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.47 DFSR, Data Fault Status Register

The DFSR characteristics are:

Purpose

Holds status information about the last data fault.

Configuration

The current translation table format determines which format of the register is used.

AArch32 System register DFSR bits [31:0] are architecturally mapped to AArch64 System register ESR_EL1[31:0].

This register is banked between DFSR and DFSR_S and DFSR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DFSR are UNDEFINED.

Attributes

DFSR is a 32-bit register.

This register has the following instances:

- DFSR, when EL3 is not implemented or FEAT_AA64 is implemented.
- DFSR_S, when FEAT_AA32EL3 is implemented.
- DFSR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

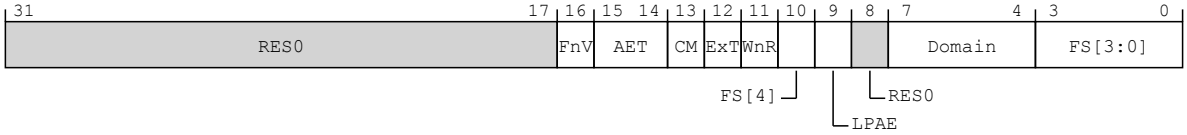

Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	DFAR is valid.
0b1	DFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AET, bits [15:14]

When FEAT_RAS is implemented:

Asynchronous Error Type. When DFSC is `0b010001`, describes the PE error state after taking the SError exception. Possible values are:

AET	Meaning
`0b00`	Uncontainable (UC).
`0b01`	Unrecoverable state (UEU).
`0b10`	Restartable state (UEO).
`0b11`	Recoverable state (UER).

This field is valid only if the DFSC code is `0b010001`. It is RES0 for all other aborts.

In the event of multiple errors taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CM, bit [13]

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault.

CM	Meaning
`0b0`	Abort not caused by execution of a cache maintenance instruction.
`0b1`	Abort caused by execution of a cache maintenance instruction, or on an address translation.

On a synchronous Data Abort exception on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ExT, bit [12]

External abort type.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For aborts other than External aborts this bit always returns 0.

This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [11]

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction.

WnR	Meaning
`0b0`	Abort caused by a read instruction.
`0b1`	Abort caused by a write instruction.

For faults on the cache maintenance and address translation System instructions in the (coproc==`0b1111`) encoding space this bit always returns a value of 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FS, bits [10, 3:0]

Fault status bits. Possible values of FS[4:0] are:

FS	Meaning	Applies when
`0b00001`	Alignment fault.	
`0b00010`	Debug exception.	
`0b00011`	Access flag fault, level 1.	
`0b00100`	Fault on instruction cache maintenance.	
`0b00101`	Translation fault, level 1.	
`0b00110`	Access flag fault, level 2.	
`0b00111`	Translation fault, level 2.	
`0b01000`	Synchronous External abort, not on translation table walk.	
`0b01001`	Domain fault, level 1.	
`0b01011`	Domain fault, level 2.	
`0b01100`	Synchronous External abort, on translation table walk, level 1.	
`0b01101`	Permission fault, level 1.	
`0b01110`	Synchronous External abort, on translation table walk, level 2.	
`0b01111`	Permission fault, level 2.	
`0b10000`	TLB conflict abort.	
`0b10100`	IMPLEMENTATION DEFINED fault (Lockdown fault).	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FS	Meaning	Applies when
0b10101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault).	
0b10110	SError exception.	
0b11000	SError exception, from a parity or ECC error on memory access.	FEAT_RAS is not implemented
0b11001	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b11100	Synchronous parity or ECC error on translation table walk, level 1.	FEAT_RAS is not implemented
0b11110	Synchronous parity or ECC error on translation table walk, level 2.	FEAT_RAS is not implemented

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Short-descriptor translation table lookup'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

LPAE	Meaning
0b0	Using the Short-descriptor translation table formats.
0b1	Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Domain, bits [7:4]

The domain of the fault address.

Arm deprecates any use of this field, see 'The Domain field in the DFSR'.

This field is UNKNOWN for certain faults where the DFSR is updated and reported using the Short-descriptor FSR encodings, see 'Validity of Domain field on faults that update the DFSR when using the Short-descriptor encodings'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When TTBCR.EAE == '1':

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

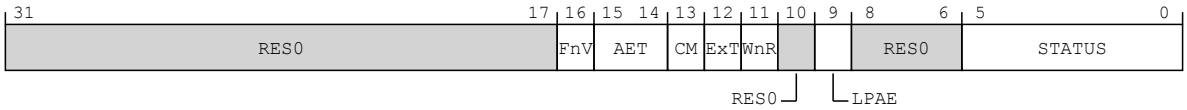

Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
`0b0`	DFAR is valid.
`0b1`	DFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AET, bits [15:14]

When FEAT_RAS is implemented:

Asynchronous Error Type. When DFSC is `0b010001`, describes the PE error state after taking the SError exception. Possible values are:

AET	Meaning
`0b00`	Uncontainable (UC).
`0b01`	Unrecoverable state (UEU).
`0b10`	Restartable state (UEO).
`0b11`	Recoverable state (UER).

This field is valid only if the DFSC code is `0b010001`. It is RES0 for all other aborts.

In the event of multiple errors taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

CM, bit [13]

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault.

CM	Meaning
`0b0`	Abort not caused by execution of a cache maintenance instruction.
`0b1`	Abort caused by execution of a cache maintenance instruction.

On a synchronous Data Abort exception on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ExT, bit [12]

External abort type.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [11]

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction.

WnR	Meaning
`0b0`	Abort caused by a read instruction.
`0b1`	Abort caused by a write instruction.

For faults on the cache maintenance and address translation System instructions in the (coproc==`0b1111`) encoding space this bit always returns a value of 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [10]

Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LPAE	Meaning
0b0	Using the Short-descriptor translation table formats.
0b1	Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

STATUS	Meaning	Applies when
0b000000	Address size fault in translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk.	
0b010001	Asynchronous SError exception.	
0b010101	Synchronous External abort on translation table walk, level 1.	
0b010110	Synchronous External abort on translation table walk, level 2.	
0b010111	Synchronous External abort on translation table walk, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STATUS	Meaning	Applies when
0b011001	Asynchronous SError exception, from a parity or ECC error on memory access.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Debug exception.	
0b110000	TLB conflict abort.	
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).	

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Long-descriptor translation table lookup'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DFSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = DFSR_NS();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t) = DFSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = DFSR_NS();
    else
        R(t) = DFSR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = DFSR_S();
    else
        R(t) = DFSR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        DFSR_NS() = R(t);
    else
        DFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        DFSR_NS() = R(t);
    else
        DFSR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        DFSR_S() = R(t);
    else
        DFSR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.48 DTLBIALL, Data TLB Invalidate All

The DTLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DTLBIALL are UNDEFINED.

Attributes

DTLBIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing DTLBIALL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    AArch32_DTLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, VMID(), Broadcast_NSH, TLBI_AllAttr);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.49 DTLBIASID, Data TLB Invalidate by ASID match

The DTLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DTLBIASID are UNDEFINED.

Attributes

DTLBIASID is a 32-bit System instruction.

Field descriptions

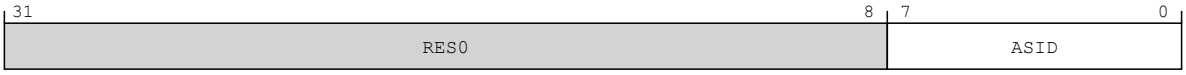

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing DTLBIASID

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
            ↪R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_DTLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBI_AllAttr,
        ↪R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.50 DTLBIMVA, Data TLB Invalidate by VA

The DTLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DTLBIMVA are UNDEFINED.

Attributes

DTLBIMVA is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing DTLBIMVA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_DTLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.51 DVPRCTX, Data Value Prediction Restriction by Context

The DVPRCTX characteristics are:

Purpose

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Note

The prediction of the PSTATE.{N,Z,C,V} values is not considered a data value for this purpose.

Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configuration

This system instruction is present only when FEAT_AA32 is implemented and FEAT_SPECRES is implemented. Otherwise, direct accesses to DVPRCTX are UNDEFINED.

Attributes

DVPRCTX is a 32-bit System instruction.

Field descriptions

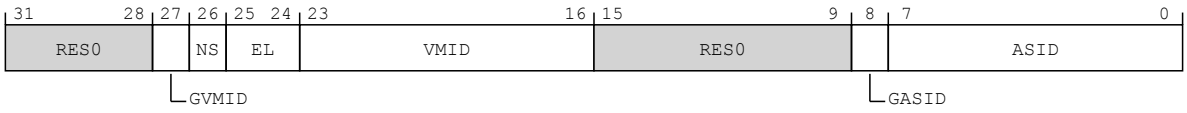

Bits [31:28]

Reserved, RES0.

GVMID, bit [27]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

If the instruction is executed in Non-secure state, this field has an Effective value of 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning	Applies when
0b00	EL0.	
0b01	EL1	
0b10	EL2	FEAT_EL2 is implemented
0b11	EL3	FEAT_EL3 is implemented

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and the target execution context is either:

- EL1.
- EL0 when the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1} or EL2 is using AArch32.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and (the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1} or ELUsingAArch32(EL2)), this field is treated as the current VMID.

When the instruction is executed at EL0 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Bits [15:9]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.
0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [7:0]

Only applies for an EL0 target execution context and when bit[8] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

Executing DVPRCTX

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0011	0b101

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_SPECRES)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪SCTLR_EL1().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && SCTLR().EnRCTX == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T7 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T7 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HFGITR_EL2().DVPRCTX == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && SCTLR_EL2().EnRCTX == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_DataValue);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_RestrictPrediction(R(t), RestrictType_DataValue);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_RestrictPrediction(R(t), RestrictType_DataValue);
elsif PSTATE.EL == EL3 then
    AArch32_RestrictPrediction(R(t), RestrictType_DataValue);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.52 ELR_hyp, Exception Link Register (Hyp mode)

The ELR_hyp characteristics are:

Purpose

When taking an exception to Hyp mode, holds the address to return to.

Configuration

AArch32 System register ELR_hyp bits [31:0] are architecturally mapped to AArch64 System register ELR_EL2[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to ELR_hyp are UNDEFINED.

Attributes

ELR_hyp is a 32-bit register.

Field descriptions

31	0
Return address	

ADDR, bits [31:0]

Return address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_hyp

ELR_hyp is accessible only at Hyp mode and Monitor mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, ELR_hyp
```

R	M	M1
0b0	0b1	0b1110

```
MSR{<c>}{<q>} ELR_hyp, <Rn>
```

R	M	M1
0b0	0b1	0b1110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.53 FCSEIDR, FCSE Process ID register

The FCSEIDR characteristics are:

Purpose

Identifies whether the Fast Context Switch Extension (FCSE) is implemented.

From Armv8.0, the FCSE is not implemented, so this register is RAZ/WI. Software can access this register to determine that the implementation does not include the FCSE.

Configuration

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to FCSEIDR are UNDEFINED.

Attributes

FCSEIDR is a 32-bit register.

Field descriptions

31	0
RAZ/WI	

Bits [31:0]

Reserved, RAZ/WI.

Accessing FCSEIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = FCSEIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = FCSEIDR();
elsif PSTATE.EL == EL3 then
    R(t) = FCSEIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        FCSEIDR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    FCSEIDR() = R(t);
elsif PSTATE.EL == EL3 then
    FCSEIDR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.54 FPEXC, Floating-Point Exception Control register

The FPEXC characteristics are:

Purpose

Provides a global enable for the implemented Advanced SIMD and floating-point functionality, and reports floating-point status information.

Configuration

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

AArch32 System register FPEXC bits [31:0] are architecturally mapped to AArch64 System register FPEXC32_EL2[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to FPEXC are UNDEFINED.

Attributes

FPEXC is a 32-bit register.

Field descriptions

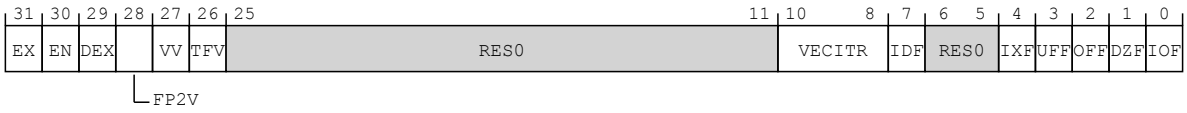

EX, bit [31]

Exception bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RAZ/WI.

EN, bit [30]

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

EN	Meaning
0b0	Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.
0b1	This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSASEDIS.

See the descriptions of the controls for more information.

Note

When executing at EL0 using AArch32:

- If EL1 is using AArch64, then the Effective value of FPEXC.EN is 1. This includes when EL2 is using AArch64 and is enabled in the current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1.
- If EL2 is using AArch64 and is enabled in the current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 0, then it is IMPLEMENTATION DEFINED whether the Effective value of FPEXC.EN is 1 or the value written to FPEXC.EN. However, Arm deprecates using the value of FPEXC.EN to determine behavior.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

DEX, bit [29]

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function `ExecutingCP10or11Instr()` returning TRUE. This field also indicates whether the FPEXC.TFV field is valid.

The meaning of this bit is:

DEX	Meaning
`0b0`	The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function `ExecutingCP10or11Instr()`. If FPEXC.TFV is RW then it is invalid and UNKNOWN. If FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.
`0b1`	The exception was generated during the execution of an allocated encoding. FPEXC.TFV is valid and indicates the cause of the exception.

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the FPSCR.{Stride, Len} fields as RAZ, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FP2V, bit [28]

FPINST2 instruction Valid bit. From Armv8.0, this bit is RES0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

VV, bit [27]

VECITR Valid bit. From Armv8.0, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} are valid.

TFV	Meaning
`0b0`	The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR.{Stride, Len} was nonzero. If FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.
`0b1`	FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When an implementation does not implement trapping of floating-point exceptions, access to this field is RAZ/WI.
- When an implementation implements FPSCR.{Stride,Len} as RAZ, access to this field is RAO/WI.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector Iteration count. From Armv8.0, this field is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES1.

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDF	Meaning
0b0	Input Denormal exception has not occurred.
0b1	Input Denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of FPSCR.FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

IXF	Meaning
0b0	Inexact exception has not occurred.
0b1	Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFF, bit [3]

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

UFF	Meaning
0b0	Underflow exception has not occurred.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UFF	Meaning
0b1	Underflow exception has occurred.

Underflow trapped exceptions can occur:

- On half-precision data-processing instructions only when FPSCR.FZ16 is 0.
- Otherwise only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFF, bit [2]

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

OFF	Meaning
0b0	Overflow exception has not occurred.
0b1	Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZF, bit [1]

Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while FPSCR.DZE was 1:

DZF	Meaning
0b0	Divide by Zero exception has not occurred.
0b1	Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOF, bit [0]

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

IOF	Meaning
0b0	Invalid Operation exception has not occurred.
0b1	Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

Accessing FPEXC

Accesses to this register use the following encodings in the System register encoding space:

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

reg
0b1000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
        ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
        ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
        ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPEXC();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
        ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPEXC();
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        R(t) = FPEXC();
    end;
end;
```

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

reg
0b1000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
        ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
        ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
        ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPEXC() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
         ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPEXC() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        FPEXC() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.55 FPSCR, Floating-Point Status and Control Register

The FPSCR characteristics are:

Purpose

Provides floating-point system status information and control.

Configuration

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to nonzero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

AArch32 System register FPSCR bits [31:27] are architecturally mapped to AArch64 System register FPSR[31:27].

AArch32 System register FPSCR bit [7] is architecturally mapped to AArch64 System register FPSR[7].

AArch32 System register FPSCR bits [4:0] are architecturally mapped to AArch64 System register FPSR[4:0].

AArch32 System register FPSCR bits [26:15] are architecturally mapped to AArch64 System register FPCR[26:15].

AArch32 System register FPSCR bits [12:8] are architecturally mapped to AArch64 System register FPCR[12:8].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to FPSCR are UNDEFINED.

Attributes

FPSCR is a 32-bit register.

Field descriptions

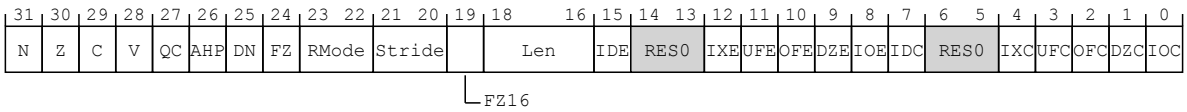

N, bit [31]

Negative condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

V, bit [28]

Overflow condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AHP, bit [26]

Alternative half-precision control bit:

AHP	Meaning
`0b0`	IEEE half-precision format selected.
`0b1`	Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN mode control bit:

DN	Meaning
`0b0`	NaN operands propagate through to the output of a floating-point operation.
`0b1`	Any operation involving one or more NaNs returns the Default NaN.

The value of this bit controls only scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Default NaN setting, regardless of the value of the DN bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flush-to-zero mode control bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FZ	Meaning
0b0	Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.
0b1	Flush-to-zero mode enabled.

The value of this bit controls only scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.

This bit has no effect on half-precision calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field. The encoding of this field is:

RMode	Meaning
0b00	Round to Nearest (RN) mode.
0b01	Round towards Plus Infinity (RP) mode.
0b10	Round towards Minus Infinity (RM) mode.
0b11	Round towards Zero (RZ) mode.

The specified rounding mode is used by almost all scalar floating-point instructions. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.{Stride,Len} as RAZ, access to this field is RAZ/WI.

FZ16, bit [19]

When FEAT_FP16 is implemented:

Flush-to-zero mode control bit on half-precision data-processing instructions:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FZ16	Meaning
0b0	Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.
0b1	Flush-to-zero mode enabled.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.{Stride,Len} as RAZ, access to this field is RAZ/WI.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

IDE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the IDC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IDC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bits [14:13]

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IXE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the IXC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IXC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFE, bit [11]

Underflow floating-point exception trap enable.

UFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the UFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the UFC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFE, bit [10]

Overflow floating-point exception trap enable.

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the OFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the OFC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

DZE	Meaning
`0b0`	Untrapped exception handling selected. If the floating-point exception occurs, the DZC bit is set to 1.
`0b1`	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the DZC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOE, bit [8]

Invalid Operation floating-point exception trap enable.

IOE	Meaning
`0b0`	Untrapped exception handling selected. If the floating-point exception occurs, the IOC bit is set to 1.
`0b1`	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IOC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IDE bit.

Advanced SIMD instructions set this bit if the Input Denormal floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IDE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IXE bit.

Advanced SIMD instructions set this bit if the Inexact floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IXE bit.

The criteria for the Inexact floating-point exception to occur are different in Flush-to-zero mode. For more information, see 'Flush-to-zero'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the UFE bit.

Advanced SIMD instructions set this bit if the Underflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, if FPSCR.UFE is 0 or if Flush-to-zero is enabled.

The criteria for the Underflow floating-point exception to occur are different in Flush-to-zero mode. For more information, see 'Flush-to-zero'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the OFE bit.

Advanced SIMD instructions set this bit if the Overflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the OFE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the DZE bit.

Advanced SIMD instructions set this bit if the Divide by Zero floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the DZE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IOE bit.

Advanced SIMD instructions set this bit if the Invalid Operation floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IOE bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPSCR

Accesses to this register use the following encodings in the System register encoding space:

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

reg
0b0001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x00);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x07);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) &&
        ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
        ↪NSACR().cp10 == '0') || CPACR().cp10 IN {'0x'}) then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && ELIsInHost(EL0) &&
        ↪CPTR_EL2().FPEN != '11' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && ELIsInHost(EL2) &&
        ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1)) &&
        ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
        ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPSCR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
        ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
         ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
         ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPSCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
         ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPSCR();
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        R(t) = FPSCR();
    end;
end;
```

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

reg
0b0001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CPACR_EL1().FPEN != '11' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_AArch32SystemAccessTrap(EL2, 0x00);
            else
                AArch64_AArch32SystemAccessTrap(EL1, 0x07);
            end;
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) &&
             ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
             ↪NSACR().cp10 == '0') || CPACR().cp10 IN {'0x'}) then
            Undefined();
        elsif IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && ELIsInHost(EL0) &&
             ↪CPTR_EL2().FPEN != '11' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x07);
        elsif IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && ELIsInHost(EL2) &&
             ↪CPTR_EL2().FPEN IN {'x0'} then
            AArch64_AArch32SystemAccessTrap(EL2, 0x07);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL2) && CPTR_EL2().TFP == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x07);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1)) &&
             ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
             ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
            AArch32_TakeHypTrapException(0x08);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x07);
            end;
        else
            FPSCR() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
            Undefined();
        elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
             ↪== '0') || CPACR().cp10 == '00' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
             ↪&& CPTR_EL2().TFP == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x07);
        elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪CPTR_EL2().FPEN IN {'x0'} then
            AArch64_AArch32SystemAccessTrap(EL2, 0x07);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
             ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
             ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
            AArch32_TakeHypTrapException(0x08);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x07);
            end;
        else
            FPSCR() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
            Undefined();
        elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
             ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
            AArch32_TakeHypTrapException(0x00);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        FPSCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        FPSCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.56 FPSID, Floating-Point System ID register

The FPSID characteristics are:

Purpose

Provides top-level information about the floating-point implementation.

This register largely duplicates information held in the MIDR. Arm deprecates use of it.

Configuration

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to FPSID are UNDEFINED.

Attributes

FPSID is a 32-bit register.

Field descriptions

31 – 24	23	22 – 16	15 – 8	7 – 4	3 – 0
Implementer	SW	Subarchitecture	PartNum	Variant	Revision

Implementer, bits [31:24]

Implementer codes are the same as those used for the MIDR.

For an implementation by Arm this field is `0x41`, the ASCII code for A.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SW, bit [23]

Software bit.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SW	Meaning
`0b0`	The implementation provides a hardware implementation of the floating-point instructions.
`0b1`	The implementation supports only software emulation of the floating-point instructions.

In Armv8, the only permitted value is 0.

Access to this field is RO.

Subarchitecture, bits [22:16]

Subarchitecture version number.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Subarchitecture	Meaning
0b0000000	VFPv1 architecture with an IMPLEMENTATION DEFINED subarchitecture.
0b0000001	VFPv2 architecture with Common VFP subarchitecture v1.
0b0000010	VFPv3 architecture, or later, with Common VFP subarchitecture v2. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.
0b0000011	VFPv3 architecture, or later, with Null subarchitecture. The entire floating-point implementation is in hardware, and no software support code is required. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers. This value can be used only by an implementation that does not support the trap enable bits in the FPSCR.
0b0000100	VFPv3 architecture, or later, with Common VFP subarchitecture v3, and support for trap enable bits in FPSCR. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

For a subarchitecture designed by Arm the most significant bit of this field, register bit[22], is 0. Values with a most significant bit of 0 that are not listed here are reserved.

When the subarchitecture designer is not Arm, the most significant bit of this field, register bit[22], must be 1. Each implementer must maintain its own list of subarchitectures it has designed, starting at subarchitecture version number `0x40`.

In Armv8-A, the permitted values are `0b0000011` and `0b0000100`.

Access to this field is RO.

PartNum, bits [15:8]

Part Number for the floating-point implementation, assigned by the implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [7:4]

Variant number. Typically, this field distinguishes between different production variants of a single product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the floating-point implementation.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing FPSID

Accesses to this register use the following encodings in the System register encoding space:

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

reg
0b0000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
         ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
         ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
         ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID0
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID0 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPSID();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
         ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = FPSID();
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        R(t) = FPSID();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

reg
0b0000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
        ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
        ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
        ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID0
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        return;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
        ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        return;
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        return;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.57 HACR, Hyp Auxiliary Configuration Register

The HACR characteristics are:

Purpose

Controls trapping to Hyp mode of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HACR bits [31:0] are architecturally mapped to AArch64 System register HACR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HACR are UNDEFINED.

Attributes

HACR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HACR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HACR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HACR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HACR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.58 HACTLR, Hyp Auxiliary Control Register

The HACTLR characteristics are:

Purpose

Controls IMPLEMENTATION DEFINED features of Hyp mode operation.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HACTLR bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HACTLR are UNDEFINED.

Attributes

HACTLR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACTLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HACTLR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HACTLR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HACTLR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HACTLR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.59 HACTLR2, Hyp Auxiliary Control Register 2

The HACTLR2 characteristics are:

Purpose

Provides additional space to the HACTLR register to hold IMPLEMENTATION DEFINED trap functionality.

Configuration

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining ID_MMFR4.AC2.

From Armv8.2 this register must be implemented.

AArch32 System register HACTLR2 bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL2[63:32].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HACTLR2 are UNDEFINED.

Attributes

HACTLR2 is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACTLR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HACTLR2();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HACTLR2();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HACTLR2() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HACTLR2() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.60 HADFSR, Hyp Auxiliary Data Fault Status Register

The HADFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED syndrome information for Data Abort exceptions taken to Hyp mode.

Configuration

This is an optional register. An implementation that does not require this register can implement it as RES0.

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HADFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR0_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HADFSR are UNDEFINED.

Attributes

HADFSR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HADFSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HADFSR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HADFSR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HADFSR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HADFSR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.61 HAIFSR, Hyp Auxiliary Instruction Fault Status Register

The HAIFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED syndrome information for Prefetch Abort exceptions taken to Hyp mode.

Configuration

This is an optional register. An implementation that does not require this register can implement it as RES0.

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HAIFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR1_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HAIFSR are UNDEFINED.

Attributes

HAIFSR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HAIFSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HAIFSR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HAIFSR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HAIFSR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HAIFSR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.62 HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0

The HAMAIR0 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR0. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR0.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HAMAIR0 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HAMAIR0 are UNDEFINED.

Attributes

HAMAIR0 is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HAMAIR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HAMAIR0();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HAMAIR0();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HAMAIR0() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HAMAIR0() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.63 HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1

The HAMAIR1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR1. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR1.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HAMAIR1 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL2[63:32].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HAMAIR1 are UNDEFINED.

Attributes

HAMAIR1 is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HAMAIR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HAMAIR1();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HAMAIR1();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HAMAIR1() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HAMAIR1() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.64 HCPTR, Hyp Architectural Feature Trap Register

The HCPTR characteristics are:

Purpose

Controls:

- Trapping to Hyp mode of Non-secure access, at EL1 or EL0, to trace, and to Advanced SIMD and floating-point functionality.
- Hyp mode access to trace, and to Advanced SIMD and floating-point functionality.

Note

Accesses to this functionality:

- From Non-secure modes other than Hyp mode are also affected by settings in the CPACR and NSACR.
- From Hyp mode are also affected by settings in the NSACR.

Exceptions generated by the CPACR and NSACR controls are higher priority than those generated by the HCPTR controls.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HCPTR bits [31:0] are architecturally mapped to AArch64 System register CPTR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HCPTR are UNDEFINED.

Attributes

HCPTR is a 32-bit register.

Field descriptions

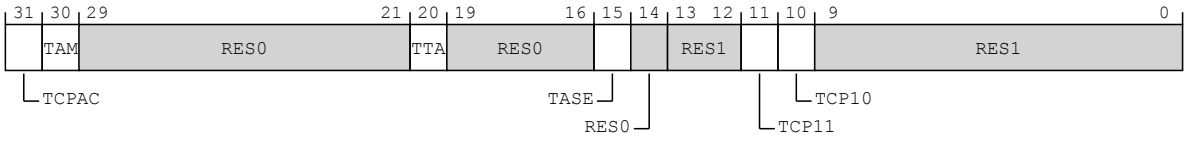

TCPAC, bit [31]

Traps Non-secure EL1 MRC and MCR accesses to the CPACR to Hyp mode, reported using EC syndrome value `0x03`.

TCPAC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 accesses to the CPACR are trapped to Hyp mode.

Note

The CPACR is not accessible at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps Non-secure EL1 and EL0 MRC, MCR, MRRC, and MCRR accesses to all Activity Monitors System registers to EL2, reported using EC syndrome values `0x03` and `0x04`.

TAM	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Accesses from Non-secure EL1 and EL0 to Activity Monitors System registers are trapped to Hyp mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

Traps Non-secure System register MRC, MCR, MRRC, and MCRR accesses to all implemented trace System registers to Hyp mode, reported using EC syndrome values `0x05` and `0x0C`.

TTA	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any Non-secure System register access to an implemented trace System register is trapped to Hyp mode, unless the access is trapped to EL1 by a CPACR or NSACR control, or the access is from Non-secure EL0 and the definition of the register in the appropriate trace architecture specification indicates that the register is not accessible from EL0. A trapped instruction generates: • A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1. • An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

If the implementation does not include a trace unit, or does not include a System register interface to the trace unit registers, it is IMPLEMENTATION DEFINED whether this bit:

- Is RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Is RES1.
- Can be written from Hyp mode, and from Secure Monitor mode when SCR.NS is 1.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

Note

- FEAT_ETMv4 and FEAT_ETE do not permit EL0 to access the trace System registers. EL0 accesses to the trace System registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than an HCPTR.TTA Hyp Trap exception.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [19:16]

Reserved, RES0.

TASE, bit [15]

Traps Non-secure execution of Advanced SIMD instructions to Hyp mode, reported using EC syndrome value `0x07`, when the value of HCPTR.TCP10 is 0.

TASE	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	When the value of HCPTR.TCP10 is 0, any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates: • A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1. • An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

When the value of HCPTR.TCP10 is 1, the value of this field is ignored.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, then it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, in Non-secure state this field behaves as RAO/WI, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see ‘Controls of Advanced SIMD operation that do not apply to floating-point operation’.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

TCP11, bit [11]

When FEAT_FP is implemented and FEAT_AdvSIMD is implemented:

The value of this field is ignored. If this field is programmed with a different value to the TCP10 bit then this field is UNKNOWN on a direct read of the HCPTR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 0, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When EL3 is implemented, EL3 is using AArch32, !IsCurrentSecurityState(SS_Secure), and NSACR.cp10 == ‘0’, access to this field is RAO/WI.

Otherwise:

Reserved, RES1.

TCP10, bit [10]

When FEAT_FP is implemented and FEAT_AdvSIMD is implemented:

Trap Non-secure accesses to Advanced SIMD and floating-point functionality to Hyp mode, reported using EC syndrome value `0x07`:

TCP10	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempted access to Advanced SIMD and floating-point functionality from Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates: • A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1. • An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 0, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When EL3 is implemented, EL3 is using AArch32, !IsCurrentSecurityState(SS_Secure), and NSACR.cp10 == '0', access to this field is RAO/WI.

Otherwise:

Reserved, RES1.

Bits [9:0]

Reserved, RES1.

Accessing HCPTR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TCPAC
        ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = HCPTR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = HCPTR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TCPAC == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TCPAC
        ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        HCPTR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HCPTR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.65 HCR, Hyp Configuration Register

The HCR characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HCR bits [31:0] are architecturally mapped to AArch64 System register HCR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HCR are UNDEFINED.

Attributes

HCR is a 32-bit register.

Field descriptions

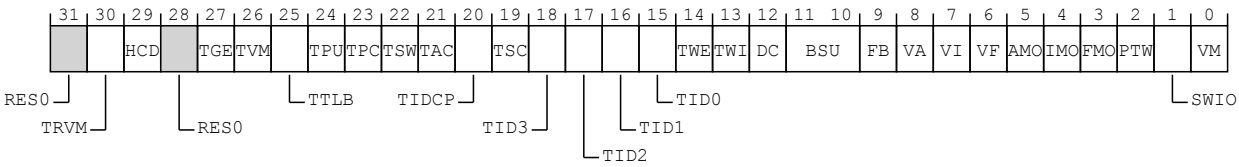

Bit [31]

Reserved, RES0.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps Non-secure EL1 reads of the virtual memory control registers to EL2, when EL2 is enabled in the current Security state.

MRC reads of the following registers are trapped and reported using EC syndrome value `0x03` and MRRC reads are trapped and reported using EC syndrome value `0x04`:

SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

TRVM	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 read accesses to the specified Virtual Memory controls are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HCD, bit [29]

When EL3 is not implemented:

HVC instruction disable.

Disables Non-secure EL1 and EL2 execution of HVC instructions, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x00`.

HCD	Meaning
`0b0`	HVC instruction execution is enabled at EL2 and EL1.
`0b1`	HVC instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

Note

HVC instructions are always UNDEFINED at EL0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [28]

Reserved, RES0.

TGE, bit [27]

Trap General Exceptions, from Non-secure EL0.

TGE	Meaning
`0b0`	This control has no effect on execution at EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TGE	Meaning
0b1	When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0. When EL2 is enabled in the current Security state, then: • All exceptions that would be routed to EL1 are routed to EL2. • The SCTLR.M bit is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR. • The HCR.{FMO, IMO, AMO} bits are treated as being 1 for all purposes other than returning the result of a direct read of HCR. • All virtual interrupts are disabled. • Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled. • An exception return to EL1 is treated as an illegal exception return. • Monitor mode execution of an MSR or CPS instruction that changes PSTATE.M to a Non-secure EL1 mode is an illegal change to PSTATE.M. For more information see 'Illegal changes to PSTATE.M'.

Also, when HCR.TGE is 1:

- If EL3 is using AArch32, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing SCR.NS from 0 to 1 results in SCR.NS remaining as 0.
- The HDCR.{TDRA, TDOSA, TDA, TDE} bits are ignored and treated as being 1 other than for the purpose of a direct read of HDCR.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TVM, bit [26]

Trap Virtual Memory controls. Traps Non-secure EL1 writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state.

MCR writes of the following registers are trapped and reported using EC syndrome value `0x03` and MCRR writes are trapped and reported using EC syndrome value `0x04`:

SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

TVM	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure EL1 write accesses to the specified virtual memory control registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TTLB, bit [25]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trap TLB maintenance instructions. Traps Non-secure EL1 execution of a TLBI instruction to EL2, when EL2 is enabled in the current Security state.

MCR and MRC accesses to the following system instructions are trapped and reported using EC syndrome value `0x03`:

TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL

TTLB	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 accesses to the specified TLB maintenance instructions are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps Non-secure EL1 execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state.

MRC and MCR accesses of the following system instructions are trapped and reported using EC syndrome value `0x03`:

- ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU.

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TPU	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPC, bit [23]

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps Non-secure EL1 execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state.

MRC and MCR accesses of the following system instructions are trapped and reported using EC syndrome value `0x03`:

- DCIMVAC, DCCIMVAC, DCCMVAC.

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TPC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps Non-secure EL1 execution of those cache maintenance instructions by set/way to EL2, when EL2 is enabled in the current Security state.

MRC and MCR accesses of the following system instructions are trapped and reported using EC syndrome value `0x03`:

- DCISW, DCCSW, DCCISW.

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TSW	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, this field resets to an architecturally UNKNOWN value.

TAC, bit [21]

Trap Auxiliary Control Registers. Traps Non-secure EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, from both Execution states.

MRC and MCR accesses of the following registers are trapped and reported using EC syndrome value `0x03`:

ACTLR and, if implemented, ACTLR2.

TAC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 accesses to the specified registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps Non-secure EL1 accesses to the encodings for IMPLEMENTATION DEFINED System Registers to EL2, when EL2 is enabled in the current Security state.

MRC and MCR accesses of the following encodings are trapped and reported using EC syndrome value `0x03`:

- All coproc==p15, CRn==c9, Opcode1 = {0-7}, CRm == {c0-c2, c5-c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c10, Opcode1 ==={0-7}, CRm == {c0, c1, c4, c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c11, Opcode1==={0-7}, CRm == {c0-c8, c15}, opcode2 == {0-7}.

TIDCP	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Non-secure EL1 accesses to the specified System register encodings for IMPLEMENTATION DEFINED functionality are trapped to EL2.

When HCR.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure EL0 is trapped to EL2. Otherwise, it is UNDEFINED and the PE takes an Undefined Instruction exception to Non-secure Undefined mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TSC, bit [19]

Trap SMC instructions. Traps Non-secure EL1 execution of SMC instructions to Hyp mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TSC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

The Armv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

Note

- This trap is implemented only if the implementation includes EL3.
- SMC instructions are always UNDEFINED at PL0.
- This bit traps execution of the SMC instruction, reported using EC syndrome value 0x13. It is not a routing control for the SMC exception. Hyp Trap exceptions and SMC exceptions have different preferred return addresses.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TID3, bit [18]

Trap ID group 3. Traps Non-secure EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state as follows:

- VMRS access to MVFR0, MVFR1, and MVFR2, reported using EC syndrome value 0x08, unless access is also trapped by HCPTR which takes priority.
- MRC access to the following registers are reported using EC syndrome value 0x03:
  - ID_PFR0, ID_PFR1, ID_PFR2, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
  - If FEAT_FGT is implemented:
    - ID_MMFR4 and ID_MMFR5 are trapped to EL2.
    - ID_ISAR6 is trapped to EL2.
    - ID_DFR1 is trapped to EL2.
    - This field traps all MRC accesses to registers in the following range that are not already mentioned in this field description: coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.
  - If FEAT_FGT is not implemented:
    - ID_MMFR4 and ID_MMFR5 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 or ID_MMFR5 are trapped.
    - ID_ISAR6 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_ISAR6 are trapped to EL2.
    - ID_DFR1 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1 are trapped to EL2.
    - Otherwise, it is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to registers not already mentioned, with coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.

TID3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified Non-secure EL1 read accesses to ID group 3 registers are trapped to EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TID2, bit [17]

Trap ID group 2. Traps the following register MRC and MCR accesses to EL2, reported using EC syndrome value `0x03`, when EL2 is enabled in the current Security state:

- Non-secure EL1 and EL0 reads of the CTR, CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- Non-secure EL1 and EL0 writes to the CSSELR.

TID2	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified Non-secure EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TID1, bit [16]

Trap ID group 1. Traps Non-secure EL1 MRC reads of the following registers to EL2, reported using EC syndrome value `0x03`, when EL2 is enabled in the current Security state:

TCMTR, TLBTR, REVIDR, AIDR.

TID1	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified Non-secure EL1 read accesses to ID group 1 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TID0, bit [15]

Trap ID group 0. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

- Non-secure EL1 VMRS reads of FPSID reported using EC syndrome value `0x08`.
- Non-secure EL0 and EL1 MCR and MRC accesses of JIDR reported using EC syndrome value `0x05`.

Note

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
- The FPSID is not accessible at EL0.
- Writes to the FPSID are ignored, and not trapped by this control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TID0	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified Non-secure EL1 read accesses to ID group 0 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TWE, bit [14]

Traps Non-secure EL0 and EL1 execution of WFE instructions to EL2, reported using EC syndrome value 0x01, when EL2 is enabled in the current Security state.

TWE	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TWI, bit [13]

Traps Non-secure EL0 and EL1 execution of WFI instructions to EL2, reported using EC syndrome value 0x01, when EL2 is enabled in the current Security state.

TWI	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

DC, bit [12]

Default Cacheability.

DC	Meaning
`0b0`	This control has no effect on the Non-secure EL1&0 translation regime.
`0b1`	In Non-secure state: • The SCTLR.M field behaves as 0 for all purposes other than a direct read of the value of the field. • The HCR.VM field behaves as 1 for all purposes other than a direct read of the value of the field. • The memory type produced by the first stage of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.

This field has no effect on the EL2 and EL3 translation regimes.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from Non-secure EL1 or Non-secure EL0:

BSU	Meaning
`0b00`	No effect.
`0b01`	Inner Shareable.
`0b10`	Outer Shareable.
`0b11`	Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

FB, bit [9]

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from Non-secure EL1:

BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALLU, TLBIMVAL, TLBIMVAAL.

FB	Meaning
`0b0`	This field has no effect on the operation of the specified instructions.
`0b1`	When one of the specified instruction is executed at Non-secure EL1, the instruction is broadcast within the Inner Shareable shareability domain.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

VA, bit [8]

Virtual SError exception.

VA	Meaning
`0b0`	This mechanism is not making a virtual SError exception pending.
`0b1`	A virtual SError exception is pending because of this mechanism.

The virtual SError exception is enabled only when the value of HCR.{TGE, AMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

VI, bit [7]

Virtual IRQ exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VI	Meaning
0b0	This mechanism is not making a virtual IRQ pending.
0b1	A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR.{TGE, IMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

VF, bit [6]

Virtual FIQ exception.

VF	Meaning
0b0	This mechanism is not making a virtual FIQ pending.
0b1	A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR.{TGE, FMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

AMO, bit [5]

SError exception Mask Override. When this bit is set to 1, it overrides the effect of PSTATE.A, and enables virtual exception signaling by the VA bit.

If the value of HCR.TGE is 0, then virtual SError exceptions are enabled in Non-secure state.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

IMO, bit [4]

IRQ Mask Override. When this bit is set to 1, it overrides the effect of PSTATE.I, and enables virtual exception signaling by the VI bit.

If the value of HCR.TGE is 0, then Virtual IRQ interrupts are enabled in the Non-secure state.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

FMO, bit [3]

FIQ Mask Override. When this bit is set to 1, it overrides the effect of PSTATE.F, and enables virtual exception signaling by the VF bit.

If the value of HCR.TGE is 0, then Virtual FIQ interrupts are enabled in the Non-secure state.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

PTW, bit [2]

Protected Table Walk. In the Non-secure PL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs then the value of this bit determines the behavior:

PTW	Meaning
`0b0`	The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.
`0b1`	The memory access generates a stage 2 Permission fault.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

SWIO, bit [1]

Set/Way Invalidation Override. Causes Non-secure EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way.

SWIO	Meaning
`0b0`	This control has no effect on the operation of data cache invalidate by set/way instructions.
`0b1`	Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When this bit is set to 1, DCISW performs the same invalidation as a DCCISW instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

As a result of changes to the behavior of DCISW, this bit is redundant in Armv8. This bit can be implemented as RES1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime.

VM	Meaning
`0b0`	Non-secure EL1&0 stage 2 address translation disabled.
`0b1`	Non-secure EL1&0 stage 2 address translation enabled.

If the HCR.DC bit is set to 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with HCR.VM being 1, regardless of the actual value of HCR.VM, other than the value returned by an explicit read of HCR.VM.

When the value of this bit is 1, data cache invalidate instructions executed at Non-secure EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR.SWIO bit.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
`0b1111`	`0b100`	`0b0001`	`0b0001`	`0b000`

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HCR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HCR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.66 HCR2, Hyp Configuration Register 2

The HCR2 characteristics are:

Purpose

Provides additional configuration controls for virtualization.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HCR2 bits [31:0] are architecturally mapped to AArch64 System register HCR_EL2[63:32].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HCR2 are UNDEFINED.

Attributes

HCR2 is a 32-bit register.

Field descriptions

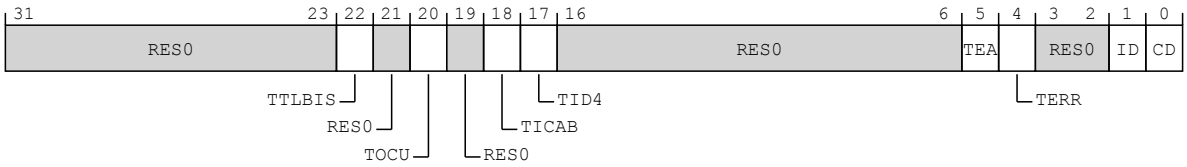

Bits [31:23]

Reserved, RES0.

TTLBIS, bit [22]

When FEAT_EVT is implemented:

Trap TLB maintenance instructions that operate on the Inner Shareable domain. Traps execution of the following TLB maintenance instructions at EL1 to EL2:

TLBIALLIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVAALIS, TLBIMVAIS, and TLBIMVALIS

TTLBIS	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure EL1 execution of the specified TLB maintenance instructions is trapped to EL2.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

TOCU, bit [20]

When FEAT_EVT is implemented:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of DCCMVAU, ICIALLU, and ICIMVAU at EL1 to EL2.

TOCU	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [19]

Reserved, RES0.

TICAB, bit [18]

When FEAT_EVT is implemented:

Trap ICIALLUIS cache maintenance instructions. Traps execution of those cache maintenance instructions at EL1 to EL2.

This applies to the following instructions:

ICIALLUIS.

TICAB	Meaning
0b0	This control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TICAB	Meaning
0b1	Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID4, bit [17]

When FEAT_EVT is implemented:

Trap ID group 4. Traps the following register accesses to EL2:

- EL1 reads of CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- EL1 writes to CSSELR.

TID4	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified Non-secure EL1 and EL0 accesses to ID group 4 registers are trapped to EL2.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this field is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:6]

Reserved, RES0.

TEA, bit [5]

When FEAT_RAS is implemented:

Route synchronous External abort exceptions from EL0 and EL1 to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TEA	Meaning
0b0	Does not route synchronous External abort exceptions from Non-secure EL0 and EL1 to EL2.
0b1	Route synchronous External abort exceptions from Non-secure EL0 and EL1 to EL2, if not routed to EL3.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TERR, bit [4]

When FEAT_RAS is implemented:

Trap Error record accesses from EL1 to EL2. Traps MRC or MCR accesses, reported using EC syndrome value 0x03, and MRRC or MCRR accesses, reported using EC syndrome value 0x04, to the following registers from EL1 to EL2:

ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.

When FEAT_RASv1p1 is implemented, ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

TERR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Accesses to the specified registers from EL1 generate a Trap exception to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [3:2]

Reserved, RES0.

ID, bit [1]

Stage 2 Instruction access cacheability disable. For the Non-secure PL1&0 translation regime, when HCR.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ID	Meaning
0b0	This control has no effect on stage 2 of the Non-secure PL1&0 translation regime.
0b1	For the Non-secure PL1&0 translation regime, forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

CD, bit [0]

Stage 2 Data access cacheability disable. When HCR.VM==1, this forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable for the Non-secure PL1&0 translation regime.

CD	Meaning
0b0	This control has no effect on stage 2 of the Non-secure PL1&0 translation regime for data accesses and translation table walks.
0b1	For the Non-secure PL1&0 translation regime, forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HCR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HCR2();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HCR2();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HCR2() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HCR2() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.67 HDFAR, Hyp Data Fault Address Register

The HDFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception that is taken to Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HDFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL2[31:0].

When FEAT_AA32EL2 is implemented and FEAT_AA32EL3 is implemented, AArch32 System register HDFAR bits [31:0] are architecturally mapped to AArch32 System register DFAR[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HDFAR are UNDEFINED.

Attributes

HDFAR is a 32-bit register.

Field descriptions

31 … 0
VA of faulting address of synchronous Data Abort exception taken to Hyp mode

VA, bits [31:0]

VA of faulting address of synchronous Data Abort exception taken to Hyp mode.

On a Prefetch Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HDFAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HDFAR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HDFAR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HDFAR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HDFAR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.68 HIFAR, Hyp Instruction Fault Address Register

The HIFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception that is taken to Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HIFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL2[63:32].

When EL2 is implemented and EL3 is implemented, AArch32 System register HIFAR bits [31:0] are architecturally mapped to AArch32 System register IFAR[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HIFAR are UNDEFINED.

Attributes

HIFAR is a 32-bit register.

Field descriptions

31	0
VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode	

VA, bits [31:0]

VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode.

On a Data Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HIFAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HIFAR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HIFAR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HIFAR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HIFAR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.69 HMAIR0, Hyp Memory Attribute Indirection Register 0

The HMAIR0 characteristics are:

Purpose

Along with HMAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIndx[2] indicates the HMAIR register to be used:

- When AttrIndx[2] is 0, HMAIR0 is used.
- When AttrIndx[2] is 1, HMAIR1 is used.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HMAIR0 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HMAIR0 are UNDEFINED.

Attributes

HMAIR0 is a 32-bit register.

Field descriptions

When TTBCR.EAE == '1':

31 – 24	23 – 16	15 – 8	7 – 0
Attr3	Attr2	Attr1	Attr0

Attr<n> , bits [8n+7:8n], for n = 3 to 0

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
`0b00`RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.
`0b01`RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
`0b10`RW	Normal memory, Outer Write-Through Non-transient.
`0b11`RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HMAIR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMAIR0();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HMAIR0();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HMAIR0() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HMAIR0() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.70 HMAIR1, Hyp Memory Attribute Indirection Register 1

The HMAIR1 characteristics are:

Purpose

Along with HMAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIndx[2] indicates the HMAIR register to be used:

- When AttrIndx[2] is 0, HMAIR0 is used.
- When AttrIndx[2] is 1, HMAIR1 is used.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HMAIR1 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL2[63:32].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HMAIR1 are UNDEFINED.

Attributes

HMAIR1 is a 32-bit register.

Field descriptions

When TTBCR.EAE == '1':

31 – 24	23 – 16	15 – 8	7 – 0
Attr7	Attr6	Attr5	Attr4

Attr<n> , bits [8(n-4)+7:8(n-4)], for n = 7 to 4

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
`0b00`RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.
`0b01`RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
`0b10`RW	Normal memory, Outer Write-Through Non-transient.
`0b11`RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is `0b0000`	Meaning when Attr<n>[7:4] is not `0b0000`
`0b0000`	Device-nGnRnE memory	UNPREDICTABLE
`0b00`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
`0b0100`	Device-nGnRE memory	Normal memory, Inner Non-cacheable
`0b01`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
`0b1000`	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=`0b00`)
`0b10`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
`0b1100`	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=`0b00`)
`0b11`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
`0b0`	No Allocate
`0b1`	Allocate

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HMAIR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
`0b1111`	`0b100`	`0b1010`	`0b0010`	`0b001`

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMAIR1();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HMAIR1();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HMAIR1() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HMAIR1() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.71 HPFAR, Hyp IPA Fault Address Register

The HPFAR characteristics are:

Purpose

Holds the faulting IPA for some aborts on a stage 2 translation taken to Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HPFAR bits [31:0] are architecturally mapped to AArch64 System register HPFAR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HPFAR are UNDEFINED.

Attributes

HPFAR is a 32-bit register.

Field descriptions

31 … 4	3 … 0
FIPA[39:12]	RES0

Execution in any Non-secure mode other than Hyp mode makes this register UNKNOWN.

FIPA[39:12], bits [31:4]

Bits [39:12] of the faulting intermediate physical address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:0]

Reserved, RES0.

Accessing HPFAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HPFAR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HPFAR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HPFAR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HPFAR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.72 HRMR, Hyp Reset Management Register

The HRMR characteristics are:

Purpose

If EL2 is the highest implemented Exception level and this register is implemented:

- A write to the register at EL2 can request a Warm reset.
- If EL2 can use AArch32 and AArch64, this register specifies the Execution state that the PE boots into on a Warm reset.

Configuration

Only implemented if EL2 is the highest implemented Exception level. In this case:

- If EL2 can use AArch32 and AArch64 then this register must be implemented.
- If EL2 cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

AArch32 System register HRMR bits [31:0] are architecturally mapped to AArch64 System register RMR_EL2[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to HRMR are UNDEFINED.

Attributes

HRMR is a 32-bit register.

Field descriptions

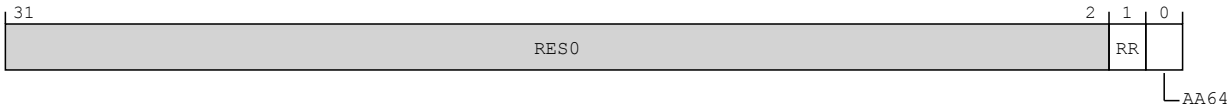

Bits [31:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

AA64, bit [0]

When Implementation can reset into AArch64 state:

Determines which Execution state the PE boots into after a Warm reset:

AA64	Meaning
`0b0`	AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

Accessing HRMR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) &&
     ↪!ELUsingAArch32(EL2) && HSTR_EL2().T12 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) &&
     ↪ELUsingAArch32(EL2) && HSTR().T12 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    R(t) = HRMR();
else
    Undefined();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) &&
     ↪!ELUsingAArch32(EL2) && HSTR_EL2().T12 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) &&
     ↪ELUsingAArch32(EL2) && HSTR().T12 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
    HRMR() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.73 HSCTLR, Hyp System Control Register

The HSCTLR characteristics are:

Purpose

Provides top-level control of the system operation in Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HSCTLR bits [31:0] are architecturally mapped to AArch64 System register SCTLR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HSCTLR are UNDEFINED.

Attributes

HSCTLR is a 32-bit register.

Field descriptions

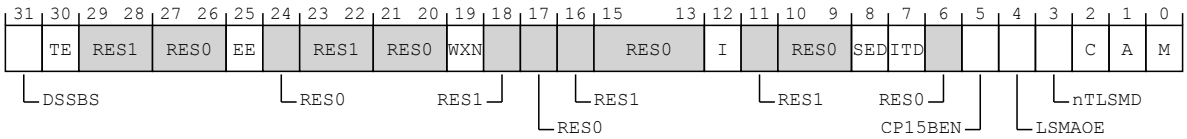

DSSBS, bit [31]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to Hyp mode.
0b1	PSTATE.SSBS is set to 1 on an exception to Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to EL2 are taken to A32 or T32 state:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TE	Meaning
0b0	Exceptions, including reset, taken to A32 state.
0b1	Exceptions, including reset, taken to T32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bits [29:28]

Reserved, RES1.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

When FEAT_MixedEnd is implemented:

The value of the PSTATE.E bit on entry to Hyp mode, the endianness of stage 1 translation table walks in the EL2 translation regime, and the endianness of stage 2 translation table walks in the PL1&0 translation regime.

EE	Meaning
0b0	Little-endian. PSTATE.E is cleared to 0 on entry to Hyp mode. Stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are little-endian.
0b1	Big-endian. PSTATE.E is set to 1 on entry to Hyp mode. Stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are big-endian.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

Reserved, RES0.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

Bits [21:20]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
`0b0`	This control has no effect on memory access permissions.
`0b1`	Any region that is writable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when HSCTLR.M bit is set.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Bits [15:13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2:

I	Meaning
`0b0`	All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache. If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
`0b1`	All instruction access to Normal memory from EL2 can be cached at all levels of instruction and unified cache. If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the PL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL2.

SED	Meaning
0b0	SETEND instruction execution is enabled at EL2.
0b1	SETEND instructions are UNDEFINED at EL2.

If the implementation does not support mixed-endian operation at EL2, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL2.

ITD	Meaning
0b0	All IT instruction functionality is enabled at EL2.

0b1	Any attempt at EL2 to execute any of the following is UNDEFINED:
	• All encodings of the IT instruction with hw1[3:0]!=1000.
	• All encodings of the subsequent instruction with the following values for hw1:
	  • 0b11xxxxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
	  • 0b1011xxxxxxxxxxxx: All instructions in 'Miscellaneous 16-bit instructions'.
	  • 0b10100xxxxxxxxxxx: ADD Rd, PC, #imm
	  • 0b01001xxxxxxxxxxx: LDR Rd, [PC, #imm]
	  • 0b0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
	  • 0b010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.
	These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.
	It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:
	• A 16-bit instruction, that can only be followed by another 16-bit instruction.
	• The first half of a 32-bit instruction.
	This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.
	An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an instruction in an active IT block that would be disabled by this field sets this field to 1, then behavior is CONSTRAINED UNPREDICTABLE. For more information, see 'Changes to an ITD control by an instruction in an IT block'.

ITD is optional, but if it is implemented in the HSCTLR, then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement ITD, access to this field is RAZ/WI.

Bit [6]

Reserved, RES0.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from EL2:

CP15BEN	Meaning
`0b0`	EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
`0b1`	EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the HSCTLR, then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement CP15BEN, access to this field is RAO/WI.

LSMAOE, bit [4]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
`0b0`	For all memory accesses at EL2, T32 and A32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
`0b1`	The ordering and interrupt behavior of T32 and A32 Load Multiple and Store Multiple at EL2 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'1'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES1.

nTLSMD, bit [3]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
`0b0`	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL2 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
`0b1`	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL2 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'1'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL2:

C	Meaning
`0b0`	All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.
`0b1`	All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, can be cached at all levels of data and unified cache.

This bit has no effect on the PL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A	Meaning
0b0	Alignment fault checking disabled when executing at EL2. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element or data elements being accessed.
0b1	Alignment fault checking enabled when executing at EL2. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element or data elements being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:

M	Meaning
0b0	EL2 stage 1 address translation disabled. See the HSCTLR.I field for the behavior of instruction accesses to Normal memory.
0b1	EL2 stage 1 address translation enabled.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HSCTLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HSCTLR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HSCTLR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSCTLR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HSCTLR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.74 HSR, Hyp Syndrome Register

The HSR characteristics are:

Purpose

Holds syndrome information for an exception taken to Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HSR bits [31:0] are architecturally mapped to AArch64 System register ESR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HSR are UNDEFINED.

Attributes

HSR is a 32-bit register.

Field descriptions

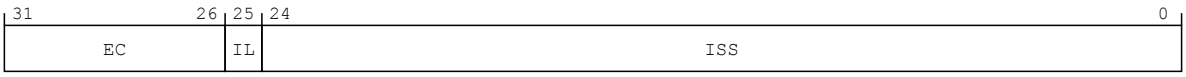

Execution in any Non-secure PE mode other than Hyp mode makes this register UNKNOWN.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of HSR is UNKNOWN. The value written to HSR must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about. Possible values of this field are:

EC	Meaning	ISS
`0b000000`	Unknown reason.	ISS encoding for exceptions with an unknown reason
`0b000001`	Trapped WFI or WFE instruction execution. Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.	ISS encoding for Exception from a WFI or WFE instruction
`0b000011`	Trapped MCR or MRC access with (coproc==`0b1111`) that is not reported using EC value `0b000000`.	ISS encoding for Exception from an MCR or MRC access
`0b000100`	Trapped MCRR or MRRC access with (coproc==`0b1111`) that is not reported using EC value `0b000000`.	ISS encoding for Exception from an MCRR or MRRC access
`0b000101`	Trapped MCR or MRC access with (coproc==`0b1110`).	ISS encoding for Exception from an MCR or MRC access

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS
0b000110	Trapped LDC or STC access. The only architected uses of these instructions are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS encoding for Exception from an LDC or STC instruction
0b000111	Access to Advanced SIMD or floating-point functionality trapped by a HCPTR.{TASE, TCP10} control. Excludes exceptions generated because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.	ISS encoding for Exception from an access to SIMD or floating-point functionality, resulting from HCPTR
0b001000	Trapped VMRS access, from ID group trap, that is not reported using EC value 0b000111.	ISS encoding for Exception from an MCR or MRC access
0b001100	Trapped MRRC access with (coproc==0b1110).	ISS encoding for Exception from an MCRR or MRRC access
0b001110	Illegal exception return to AArch32 state.	ISS encoding for Exception from an Illegal state or PC alignment fault
0b010001	Exception on SVC instruction execution in AArch32 state routed to EL2.	ISS encoding for Exception from HVC or SVC instruction execution
0b010010	HVC instruction execution in AArch32 state, when HVC is not disabled.	ISS encoding for Exception from HVC or SVC instruction execution
0b010011	Trapped execution of SMC instruction in AArch32 state.	ISS encoding for Exception from SMC instruction execution
0b100000	Prefetch Abort from a lower Exception level.	ISS encoding for Exception from a Prefetch Abort
0b100001	Prefetch Abort taken without a change in Exception level.	ISS encoding for Exception from a Prefetch Abort
0b100010	PC alignment fault exception.	ISS encoding for Exception from an Illegal state or PC alignment fault
0b100100	Data Abort exception from a lower Exception level.	ISS encoding for Exception from a Data Abort
0b100101	Data Abort exception taken without a change in Exception level.	ISS encoding for Exception from a Data Abort

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction length bit. Indicates the size of the instruction that has been trapped to Hyp mode. When this bit is valid, possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IL	Meaning
0b0	16-bit instruction trapped.
0b1	32-bit instruction trapped.

This field is RES1 and not valid for the following cases:

- When the EC value is 0b000000, indicating an exception with an unknown reason.
- Prefetch Aborts.
- Data Abort exceptions for which the HSR.ISS.ISV field is 0.
- When the EC value is 0b001110, indicating an Illegal state exception.

The IL field is not valid and is UNKNOWN on an exception from a PC alignment fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

ISS encoding for exceptions with an unknown reason

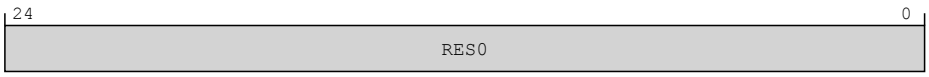

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for exceptions with an unknown reason

This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or is not accessible in the current PE mode in the current Security state, including:
  - A read access using a System register encoding pattern that is not allocated for reads or that does not permit reads in the current PE mode and Security state.
  - A write access using a System register encoding pattern that is not allocated for writes or that does not permit writes in the current PE mode and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- The attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR.HCD, SCR.HCE, or SCR_EL3.HCE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - An SMC instruction when disabled by SCR.SCD or SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- An HVC instruction when disabled by HCR.HCD, SCR.HCE, or SCR_EL3.HCE. An SMC instruction when disabled by SCR.SCD or SCR_EL3.SMD. An HLT instruction when disabled by EDSCR.HDE.
- An exception generated because of the attempted execution of an MSR (Banked register) or MRS (Banked register) instruction that would access a Banked register that is not accessible from the Security state and PE mode at which the instruction was executed.

Note

An exception is generated only if the CONSTRAINED UNPREDICTABLE behavior of the instruction is that it is UNDEFINED, see 'MSR (banked register) and MRS (banked register)'.

- Attempted execution, in Debug state, of:
  - A DCPS1 instruction in Non-secure state from EL0 when EL2 is using AArch32 and the value of HCR.TGE is 1.
  - A DCPS2 instruction at EL1 or EL0 when EL2 is not implemented, or when EL3 is using AArch32 and the value of SCR.NS is 0, or when EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  - A DCPS3 instruction when EL3 is not implemented, or when the value of EDSCR.SDD is 1.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, and EL0 of an instruction that is configured to trap to EL3.

'Undefined Instruction exception, when the value of HCR.TGE is 1' describes the configuration settings for a trap that returns an EC value of `0b000000`.

ISS encoding for Exception from a WFI or WFE instruction

24	23 20	19 1	0
CV	COND	RES0	TI

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

TI	Meaning
`0b0`	WFI trapped.
`0b1`	WFE trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from a WFI or WFE instruction

HCR.{TWE, TWI} describe the configuration settings for this trap.

ISS encoding for Exception from an MCR or MRC access

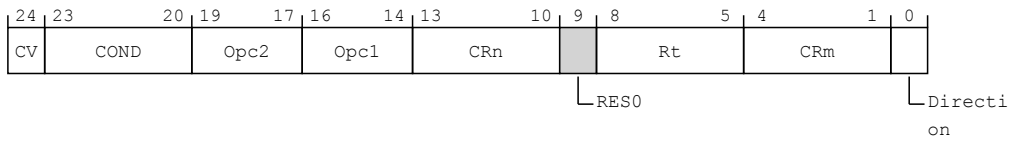

CV, bit [24]

Condition code valid. Possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
`0b0`	Write to System register space. MCR instruction.
`0b1`	Read from System register space. MRC or VMRS instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an MCR or MRC access

The following fields describe configuration settings for traps from an MCR or MCR access using coproc `0b1111` that are reported using EC value `0b000011`:

- HCR.{TID1, TID2, TID3}, for Non-secure accesses to the ID registers at EL0 and EL1, trapped to EL2.
- HCR.TIDCP, for Non-secure accesses to lockdown, DMA, and TCM operations at EL0 and EL1, trapped to EL2.
- HCR.{TSW, TPC, TPU}, for Non-secure execution of cache maintenance instructions at EL1, trapped to EL2.
- HCR.TTLB, for Non-secure execution of TLB maintenance instructions at EL1, trapped to EL2.
- HCR.TAC, for Non-secure accesses to the Auxiliary Control Register at EL1, trapped to EL2.
- HDCR.{TPM, TPMCR}, for Non-secure accesses to Performance Monitors registers at EL0 and EL1, trapped to EL2.
- HCPTR.TAM, for Non-secure accesses to Activity Monitors System registers at EL0 and EL1, trapped to EL2.
- HCPTR.TCPAC, for Non-secure accesses to the CPACR at EL1, trapped to EL2.
- HCR.{TRVM, TVM}, for Non-secure accesses to virtual memory control registers at EL1, trapped to EL2.
- HSTR.T<n>, for Non-secure accesses to System registers in the (coproc == 1111) encoding space at EL0 and EL1, trapped to EL2.
- HDCR.TTRF, for Non-secure accesses to trace filter control registers from System register, trapped to EL2.
- CNTHCTL.PL1PCEN, for Non-secure accesses to the Generic Timer registers at EL0 and EL1, trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- HCR2.TERR, for Non-secure accesses to the RAS error record registers at EL1, trapped to EL2.

The following fields describe configuration settings for traps from an MCR or MRC access using coproc `0b1110` that are reported using EC value `0b000101`:

- HCR.TID0, for Non-secure accesses to the Primary device identification registers at EL0 and EL1, trapped to EL2.
- HCPTR.TTA, for Non-secure accesses to trace System registers from System register, trapped to EL2.
- HDCR.TDRA, for Non-secure accesses to Debug ROM registers from System register, trapped to EL2.
- HDCR.TDOSA, for Non-secure accesses to powerdown debug System registers from System register, trapped to EL2.
- HDCR.TDA, for Non-secure accesses to debug System registers from System register, trapped to EL2.

The following fields describes configuration settings for traps from a VMSR or VMRS access that are reported using EC value `0b001000`:

- HCR.TID0, for Non-secure accesses to the Primary device identification registers at EL1, for ID group traps trapped to EL2.
- HCR.TID3, for Non-secure accesses to the Detailed feature identification registers at EL0 and EL1, for ID group traps trapped to EL2.
- HCPTR.{TCP10, TCP11}, for Non-secure accesses to FPSCR, FPSID, FPEXC, MVFR0, MVFR1, and MVFR2, trapped to EL2.

ISS encoding for Exception from an MCRR or MRRC access

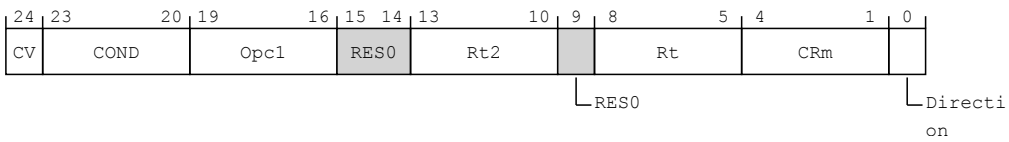

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:14]

Reserved, RES0.

Rt2, bits [13:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Direction	Meaning
0b0	Write to System register space. MCRR instruction.
0b1	Read from System register space. MRRC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an MCRR or MRRC access

The following fields describe configuration settings for traps from an MCRR or MRRC access using coproc 0b1111 that are reported using EC value 0b000100:

- HCR.{TRVM, TVM}, for Non-secure accesses to virtual memory control registers at EL1, trapped to EL2.
- HDCR.TPM, for Non-secure accesses to Performance Monitors registers at EL0 and EL1, trapped to EL2.
- HCPTR.TAM, for Non-secure accesses to Activity Monitors System registers at EL0 and EL1, trapped to EL2.
- CNTHCTL.{PL1PCEN, PL1PCTEN}, for Non-secure accesses to the Generic Timer registers at EL0 and EL1, trapped to EL2.
- HSTR.T<n>, for Non-secure accesses to System registers in the (coproc == 1111) encoding space at EL0 and EL1, trapped to EL2.
- HCR2.TERR, for Non-secure accesses to the RAS error record registers at EL1, trapped to EL2.

The following fields describe configuration settings for traps from an MCRR or MRRC access using coproc 0b1110 that are reported using EC value 0b001100:

- HCPTR.TTA, for Non-secure accesses to trace System registers from System register, trapped to EL2.
- HDCR.TDRA, for Non-secure accesses to Debug ROM registers from System register, trapped to EL2.

ISS encoding for Exception from an LDC or STC instruction

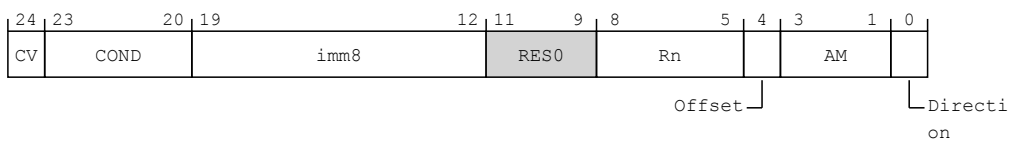

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:9]

Reserved, RES0.

Rn, bits [8:5]

The Rn value from the issued instruction. Valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction.

When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Offset	Meaning
`0b0`	Subtract offset.
`0b1`	Add offset.

This bit corresponds to the U bit in the instruction encoding.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

AM	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	Literal unindexed. LDC instruction in A32 instruction set only. For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	Literal offset. LDC instruction only. For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an LDC or STC instruction

HDCR.TDA describes the configuration settings for the trap that is reported using EC value 0b000110.

ISS encoding for Exception from an access to SIMD or floating-point functionality, resulting from HCPTR

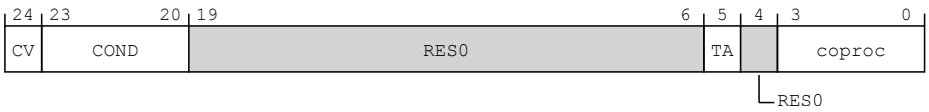

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Excludes exceptions that occur because Advanced SIMD and floating-point functionality is not implemented, or because the value of HCR.TGE or HCR_EL2.TGE is 1. These are reported with EC value `0b000000`.

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:6]

Reserved, RES0.

TA, bit [5]

Indicates trapped use of Advanced SIMD functionality.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TA	Meaning
0b0	Exception was not caused by trapped use of Advanced SIMD functionality.
0b1	Exception was caused by trapped use of Advanced SIMD functionality.

Any use of an Advanced SIMD instruction that is not also a floating-point instruction that is trapped to Hyp mode because of a trap configured in the HCPTR sets this bit to 1.

For a list of these instructions, see 'Controls of Advanced SIMD operation that do not apply to floating-point operation'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [4]

Reserved, RES0.

coproc, bits [3:0]

When the HSR.TA field returns the value 1, this field returns the value 0b1010. Otherwise, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an access to SIMD or floating-point functionality, resulting from HCPTR

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:

- HCPTR.{TCP11, TCP10}, for Non-secure accesses to the SIMD and floating-point registers, trapped to EL2.
- HCPTR.TASE, for Non-secure accesses to Advanced SIMD functionality, trapped to EL2.

ISS encoding for Exception from HVC or SVC instruction execution

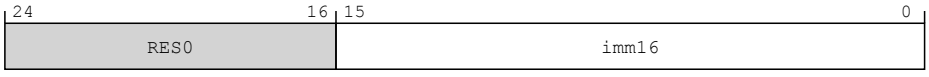

Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, this is the value of the imm16 field of the issued instruction.

For an SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- For the T32 instruction, this field is zero-extended from the imm8 field of the instruction. For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from HVC or SVC instruction execution

The HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

'Supervisor Call exception, when the value of HCR.TGE is 1' describes the configuration settings for the trap reported with EC value `0b010001`.

ISS encoding for Exception from SMC instruction execution

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

CCKNOWNPASS	Meaning
`0b0`	The instruction was unconditional, or was conditional and passed its condition code check.
`0b1`	The instruction was conditional, and might have failed its condition code check.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [18:0]

Reserved, RES0.

Additional Information for ISS encoding for Exception from SMC instruction execution

HCR.TSC describes the configuration settings for this trap for instructions executed in Non-secure EL1.

ISS encoding for Exception from a Prefetch Abort

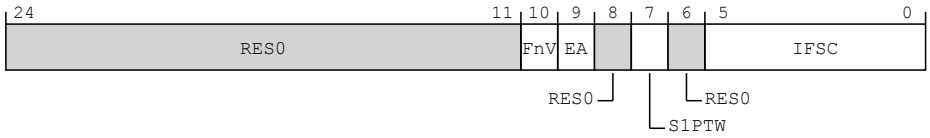

Bits [24:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
`0b0`	HIFAR is valid.
`0b1`	HIFAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is `0b010000`. It is RES0 for all other aborts.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code. Possible values of this field are:

IFSC	Meaning	Applies when
0b000000	Address size fault in translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001101	Permission fault, level 1.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk.	
0b010101	Synchronous External abort on translation table walk, level 1.	
0b010110	Synchronous External abort on translation table walk, level 2.	
0b010111	Synchronous External abort on translation table walk, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3.	FEAT_RAS is not implemented
0b100010	Debug exception.	
0b110000	TLB conflict abort.	

All other values are reserved.

For more information about the lookup level associated with a fault, see ‘The level associated with MMU faults on a Long-descriptor translation table lookup’.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from a Prefetch Abort

The following sections describe cases where Prefetch Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100000:

- ‘Abort exceptions, when the value of HCR.TGE is 1’.
- ‘Routing debug exceptions to EL2 using AArch32’.

ISS encoding for Exception from an Illegal state or PC alignment fault

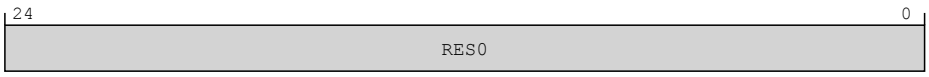

Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for Exception from an Illegal state or PC alignment fault

For more information about the Illegal state exception, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ‘Illegal changes to PSTATE.M’.
- ‘Illegal return events from AArch32 state’.
- ‘Legal returns that set PSTATE.IL to 1’.
- ‘The Illegal Execution state exception’.

For more information about the PC alignment fault exception, see ‘Branching to an unaligned PC’.

ISS encoding for Exception from a Data Abort

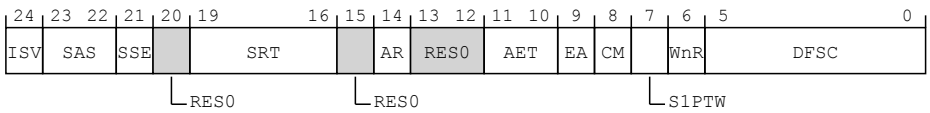

ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

ISV	Meaning
0b0	No valid instruction syndrome. ISS[23:14] are RES0.
0b1	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults except Data Abort exceptions generated by stage 2 address translations for which all the following apply to the instruction that generated the Data Abort exception:

- The instruction is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
- The instruction is not performing register writeback.
- The instruction is not using the PC as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in Memory access mode, as described in ‘Data Abort exceptions in Memory access mode’, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

Note

In the A32 instruction set, LDR*T and STR*T instructions always perform register writeback and therefore never return a valid instruction syndrome.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.

ISV is set to 0 on a stage 2 abort on a stage 1 translation table walk.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SAS	Meaning
0b00	Byte
0b01	Halfword
0b10	Word
0b11	Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

SSE	Meaning
0b0	Sign-extension not required.
0b1	Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [20]

Reserved, RES0.

SRT, bits [19:16]

Syndrome Register Transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [13:12]

Reserved, RES0.

Bits [11:10]

When FEAT_RAS is implemented:

AET, bits [11:10]

Asynchronous Error Type. When DFSC is 0b010001, describes the PE error state after taking the SError exception.

AET	Meaning
0b00	Uncontainable (UC).
0b01	Unrecoverable state (UEU).
0b10	Restartable state (UEO).
0b11	Recoverable state (UER).

On a synchronous Data Abort exception, this field is RES0.

In the event of multiple errors taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

When FEAT_RAS is not implemented, or when DFSC is not 0b010001:

- Bit[11] is RES0.
- Bit[10] forms the FnV field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

FnV, bit [0] of bits [11:10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FnV	Meaning
0b0	HDFAR is valid.
0b1	HDFAR is not valid, and holds an UNKNOWN value.

When FEAT_RAS is not implemented, this field is valid only if DFSC is 0b010000. It is RES0 for all other aborts.

When FEAT_RAS is implemented:

- If DFSC is 0b010000, this field is valid.
- If DFSC is 0b010001, this bit forms part of the AET field, becoming AET[0].
- This field is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External Abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache Maintenance. For a synchronous fault, identifies fault that comes from a cache maintenance or address translation instruction. For synchronous faults, the possible values of this bit are:

CM	Meaning
0b0	Fault not generated by a cache maintenance or address translation instruction.
0b1	Fault generated by a cache maintenance or address translation instruction.

For an asynchronous Data Abort exception, this bit is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction.

WnR	Meaning
0b0	Abort caused by a read instruction.
0b1	Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

On an asynchronous Data Abort exception:

- When FEAT_RAS is not implemented, this bit is UNKNOWN.
- When FEAT_RAS is implemented, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code. Possible values of this field are:

DFSC	Meaning	Applies when
0b000000	Address size fault in translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk.	
0b010001	Asynchronous SError exception.	
0b010101	Synchronous External abort on translation table walk, level 1.	
0b010110	Synchronous External abort on translation table walk, level 2.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b010111	Synchronous External abort on translation table walk, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011001	Asynchronous SError exception, from a parity or ECC error on memory access.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Debug exception.	
0b110000	TLB conflict abort.	
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).	

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Long-descriptor translation table lookup'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from a Data Abort

The following describe cases where Data Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100100:

- 'Abort exceptions, when the value of HCR.TGE is 1'.
- 'Routing debug exceptions to EL2 using AArch32'.

The following describe cases that can cause a Data Abort exception that is taken to Hyp mode, and reported in the HSR with EC value of 0b100000 or 0b100100:

- 'Hyp mode control of Non-secure access permissions'.
- 'Memory fault reporting in Hyp mode'.

Accessing HSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HSR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HSR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HSR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.75 HSTR, Hyp System Trap Register

The HSTR characteristics are:

Purpose

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to System registers in the coproc == `0b1111` encoding space:

- By the CRn value used to access the register using MCR or MRC instruction.
- By the CRm value used to access the register using MCRR or MRRC instruction.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HSTR bits [31:0] are architecturally mapped to AArch64 System register HSTR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HSTR are UNDEFINED.

Attributes

HSTR is a 32-bit register.

Field descriptions

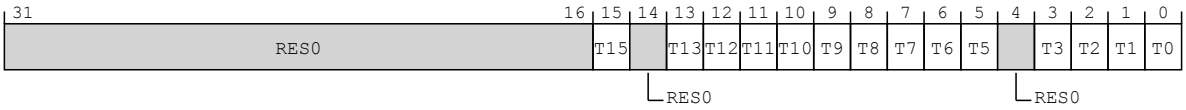

Bits [31:16, 14, 4]

Reserved, RES0.

T<n> , bits [n], for n = 15 to 15, 13 to 5, 3 to 0

The remaining fields control whether Non-secure EL0 and EL1 accesses, using MCR or MRC instructions, reported using EC syndrome value `0x03`, and MCRR or MRRC instructions, reported using EC syndrome value `0x04`, to the System registers in the coproc == `0b1111` encoding space are trapped to Hyp mode:

T<n>	Meaning
`0b0`	This control has no effect on Non-secure EL0 or EL1 accesses to System registers.
`0b1`	Any Non-secure EL1 MCR or MRC access with coproc == `0b1111` and CRn == <n> is trapped to Hyp mode. A Non-secure EL0 MCR or MRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0. Any Non-secure EL1 MCRR or MRRC access with coproc == `0b1111` and CRm == <n> is trapped to Hyp mode. A Non-secure EL0 MCRR or MRRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0.

For example, when HSTR.T7 is 1, for instructions executed at Non-secure EL1:

- An MCR or MRC instruction with coproc set to `0b1111` and `<CRn>` set to c7 is trapped to Hyp mode.
- An MCRR or MRRC instruction with coproc set to `0b1111` and `<CRm>` set to c7 is trapped to Hyp mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the expression `0x0000`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HSTR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HSTR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HSTR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSTR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HSTR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.76 HTCR, Hyp Translation Control Register

The HTCR characteristics are:

Purpose

The control register for stage 1 of the EL2 translation regime.

Note

This stage of translation always uses the Long-descriptor translation table format.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HTCR bits [31:0] are architecturally mapped to AArch64 System register TCR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HTCR are UNDEFINED.

Attributes

HTCR is a 32-bit register.

Field descriptions

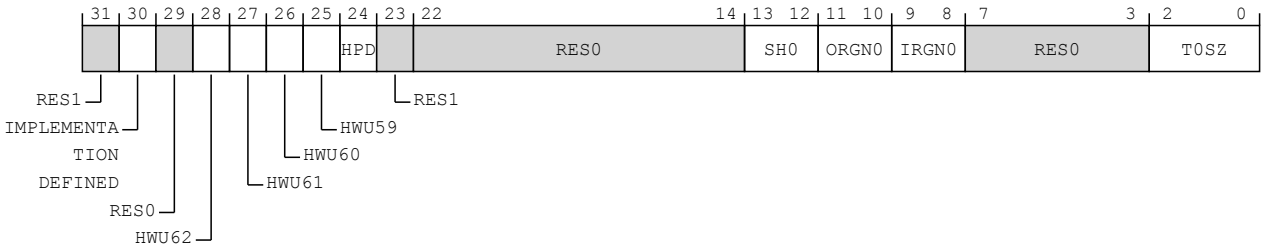

Bit [31]

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [30]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU62	Meaning
0b0	Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

HWU61	Meaning
0b0	Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

HWU60	Meaning
0b0	Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

HWU59	Meaning
0b0	Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD, bit [24]

When FEAT_AA32HPD is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the PL2 translation regime.

HPD	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

Bits [22:14]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using HTTBR.

SH0	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using HTTBR.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using HTTBR.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:3]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T0SZ, bits [2:0]

The size offset of the memory region addressed by HTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HTCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HTCR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HTCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HTCR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HTCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.77 HTPIDR, Hyp Software Thread ID Register

The HTPIDR characteristics are:

Purpose

Provides a location where software running in Hyp mode can store thread identifying information that is not visible to Non-secure software executing at EL0 or EL1, for hypervisor management purposes.

The PE makes no use of this register.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

Note

The PE never updates this register.

AArch32 System register HTPIDR bits [31:0] are architecturally mapped to AArch64 System register TPIDR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HTPIDR are UNDEFINED.

Attributes

HTPIDR is a 32-bit register.

Field descriptions

31 … 0
Thread ID

TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HTPIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HTPIDR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HTPIDR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HTPIDR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HTPIDR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.78 HTTBR, Hyp Translation Table Base Register

The HTTBR characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL2 translation regime, and other information for this translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HTTBR bits [47:0] are architecturally mapped to AArch64 System register TTBR0_EL2[47:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HTTBR are UNDEFINED.

Attributes

HTTBR is a 64-bit register.

Field descriptions

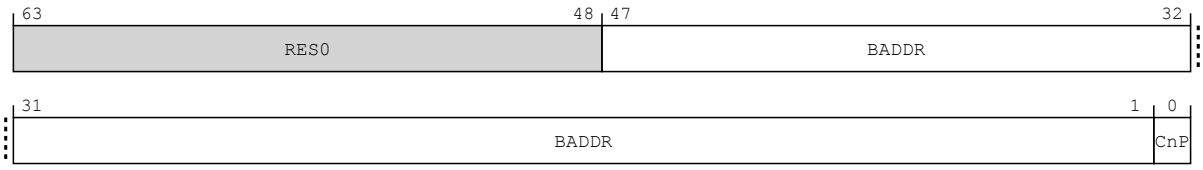

Bits [63:48]

Reserved, RES0.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of HTCR.T0SZ as follows:

- If HTCR.T0SZ is 0 or 1, x = 5 - HTCR.T0SZ.
- If HTCR.T0SZ is greater than 1, x = 14 - HTCR.T0SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Common not Private. This bit indicates whether each entry that is pointed to by HTTBR is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of HTTBR.CnP is 1.

CnP	Meaning
0b0	The translation table entries pointed to by HTTBR are permitted to differ from corresponding entries for HTTBR for other PEs in the Inner Shareable domain. This is not affected by the value of HTTBR.CnP on those other PEs.
0b1	The translation table entries pointed to by HTTBR are the same as the translation table entries pointed to by HTTBR on every other PE in the Inner Shareable domain for which the value of HTTBR.CnP is 1.

Note

If the value of the HTTBR.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those HTTBRs do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing HTTBR

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = HTTBR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t, t2) = HTTBR();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HTTBR() = R(t, t2);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HTTBR() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.79 HVBAR, Hyp Vector Base Address Register

The HVBAR characteristics are:

Purpose

Holds the vector base address for any exception that is taken to Hyp mode.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register HVBAR bits [31:0] are architecturally mapped to AArch64 System register VBAR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HVBAR are UNDEFINED.

Attributes

HVBAR is a 32-bit register.

Field descriptions

31 … 5	4 … 0
Vector Base Address	RES0

VBA, bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:0]

Reserved, RES0.

Accessing HVBAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HVBAR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HVBAR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HVBAR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HVBAR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.80 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

Purpose

Invalidate all instruction caches of the PE executing the instruction to the Point of Unification. If branch predictors are architecturally visible, also flush branch predictors.

Configuration

AArch32 System instruction ICIALLU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ICIALLU are UNDEFINED.

Attributes

ICIALLU is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing ICIALLU

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When HCR.FB is 1, at Non-secure EL1 this instruction executes as a ICIALLUIS.

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPU ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TOCU
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPU == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TOCU ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLU);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLU);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch32_IC(CacheOpScope_ALLU);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.81 ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The ICIALLUIS characteristics are:

Purpose

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification. If branch predictors are architecturally visible, also flush branch predictors.

Configuration

AArch32 System instruction ICIALLUIS bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ICIALLUIS are UNDEFINED.

Attributes

ICIALLUIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing ICIALLUIS

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPU ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TICAB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPU == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TICAB ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLUIS);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLUIS);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch32_IC(CacheOpScope_ALLUIS);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.82 ICIMVAU, Instruction Cache line Invalidate by VA to PoU

The ICIMVAU characteristics are:

Purpose

Invalidate instruction cache line by virtual address to PoU.

Configuration

AArch32 System instruction ICIMVAU bits [31:0] performs the same function as bits [31:0].

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ICIMVAU are UNDEFINED.

Attributes

ICIMVAU is a 32-bit System instruction.

Field descriptions

31	0
Virtual address to use	

VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing ICIMVAU

Execution of this instruction might require an address translation from VA to PA, and that translation might fault.

For more information about faults, see ‘Permission fault’.

For more information about data cache maintenance instructions, see ’AArch32 instruction cache maintenance instructions (IC*)’.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TPU ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TOCU
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TPU == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TOCU ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(R(t), CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(R(t), CacheOpScope_PoU);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        AArch32_IC(R(t), CacheOpScope_PoU);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.83 ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_AFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_AFR0_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_AFR0 are UNDEFINED.

Attributes

ID_AFR0 is a 32-bit register.

Field descriptions

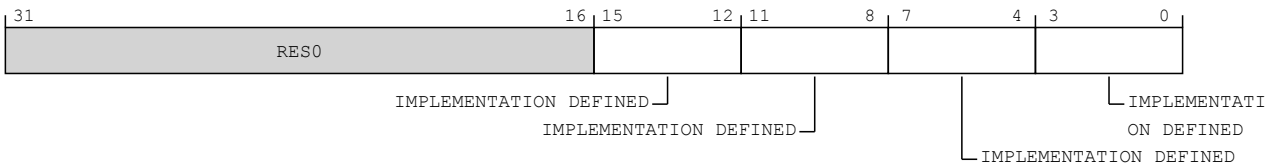

Bits [31:16]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing ID_AFR0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_AFR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_AFR0();
elsif PSTATE.EL == EL3 then
    R(t) = ID_AFR0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.84 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_DFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_DFR0_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_DFR0 are UNDEFINED.

Attributes

ID_DFR0 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
TraceFilt	PerfMon	MProfDbg	MMapTrc	CopTrc	MMapDbg	CopSDbg	CopDbg

TraceFilt, bits [31:28]

Armv8.4 Self-hosted Trace Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceFilt	Meaning
0b0000	Armv8.4 Self-hosted Trace Extension not implemented.
0b0001	Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_ETMv4 is implemented, the value 0b0000 is not permitted.

If FEAT_ETE is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

PerfMon, bits [27:24]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PerfMon	Meaning
0b0000	Performance Monitors Extension not implemented.
0b0001	Performance Monitors Extension, PMUv1 implemented.
0b0010	Performance Monitors Extension, PMUv2 implemented.
0b0011	Performance Monitors Extension, PMUv3 implemented.
0b0100	PMUv3 for Armv8.1. As 0b0011, and adds support for: • Extended 16-bit PMEVTYPER<n>.evtCount field. • If EL2 is implemented, the HDCR.HPMD control.
0b0101	PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR register.
0b0110	PMUv3 for Armv8.5. As 0b0101, and adds support for: • 64-bit event counters. • If EL2 is implemented, the HDCR.HCCD control. • If EL3 is implemented, the SDCR.SCCD control.
0b0111	PMUv3 for Armv8.7. As 0b0110, and adds support for: • The PMCR.FZO and, if EL2 is implemented, HDCR.HPMFZO controls. • If EL3 is implemented and using AArch64, the MDCR_EL3.{MPMX,MCCD} controls.
0b1000	PMUv3 for Armv8.8. As 0b0111, and: • Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF. • Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.
0b1001	PMUv3 for Armv8.9. As 0b1000, and: • Updates the definitions of existing PMU events. • Adds support for the EDECR.PME control.
0b1111	IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0011.

FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.

FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.

FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.

FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.

FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

FEAT_PMUv3p9 implements the functionality identified by the value 0b1001.

In any Armv8 implementation, the values 0b0001 and 0b0010 are not permitted.

From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0011 is not permitted.

From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.

From Armv8.5, if FEAT_PMUv3 is implemented, the value 0b0101 is not permitted.

From Armv8.7, if FEAT_PMUv3 is implemented, the value 0b0110 is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv8.8, if FEAT_PMUv3 is implemented, the value `0b0111` is not permitted.

From Armv8.9, if FEAT_PMUv3 is implemented, the value `0b1000` is not permitted.

Note

In Armv7, the value `0b0000` can mean that PMUv1 is implemented. PMUv1 and PMUv2 are not permitted in an Armv8 implementation.

Access to this field is RO.

MProfDbg, bits [23:20]

M-profile Debug. Support for memory-mapped debug model for M-profile processors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MProfDbg	Meaning
`0b0000`	Not supported.
`0b0001`	Support for M-profile Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

MMapTrc, bits [19:16]

Memory-mapped Trace. Support for memory-mapped trace model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MMapTrc	Meaning
`0b0000`	Not supported.
`0b0001`	Support for Arm trace architecture, with memory-mapped access.

All other values are reserved.

In Armv8, the permitted values are `0b0000` and `0b0001`.

For more information, see the Arm® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

Access to this field is RO.

CopTrc, bits [15:12]

Support for System registers-based trace model, using registers in the coproc == `0b1110` encoding space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CopTrc	Meaning
0b0000	Not supported.
0b0001	Support for Arm trace architecture, with System registers access.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

For more information, see the Arm® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

Access to this field is RO.

MMapDbg, bits [11:8]

Memory-mapped Debug. Support for Armv7 memory-mapped debug model for A and R-profile processors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MMapDbg	Meaning
0b0000	Not supported.
0b0100	Support for Armv7, v7 Debug architecture, with memory-mapped access.
0b0101	Support for Armv7, v7.1 Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

The optional memory map defined by Armv8 is not compatible with Armv7.

Access to this field is RO.

CopSDbg, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = 0b1110 encoding space, for an A-profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CopDbg, bits [3:0]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CopDbg	Meaning
0b0000	Not supported.
0b0010	Armv6, v6 Debug architecture, with System registers access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CopDbg	Meaning
0b0011	Armv6, v6.1 Debug architecture, with System registers access.
0b0100	Armv7, v7 Debug architecture, with System registers access.
0b0101	Armv7, v7.1 Debug architecture, with System registers access.
0b0110	Armv8 debug architecture.
0b0111	Armv8.1 debug architecture, FEAT_Debugv8p1.
0b1000	Armv8.2 debug architecture, FEAT_Debugv8p2.
0b1001	Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010	Armv8.8 debug architecture, FEAT_Debugv8p8.
0b1011	Armv8.9 debug architecture, FEAT_Debugv8p9.

All other values are reserved.

The values 0b0000, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted in Armv8.

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

FEAT_Debugv8p8 implements the functionality identified by the value 0b1010.

FEAT_Debugv8p9 implements the functionality identified by the value 0b1011.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

From Armv8.8, the value 0b1001 is not permitted.

From Armv8.9, the value 0b1010 is not permitted.

Access to this field is RO.

Accessing ID_DFR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_DFR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_DFR0();
elsif PSTATE.EL == EL3 then
    R(t) = ID_DFR0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.85 ID_DFR1, Debug Feature Register 1

The ID_DFR1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch32 System register ID_DFR1 bits [31:0] are architecturally mapped to AArch64 System register ID_DFR1_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_DFR1 are UNDEFINED.

Attributes

ID_DFR1 is a 32-bit register.

Field descriptions

31 – 8	7 – 4	3 – 0
RES0	HPMN0	MTPMU

Bits [31:8]

Reserved, RES0.

HPMN0, bits [7:4]

Zero PMU event counters for a Guest operating system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HPMN0	Meaning
0b0000	Setting HDCR.HPMN to zero has CONSTRAINED UNPREDICTABLE behavior.
0b0001	Setting HDCR.HPMN to zero has defined behavior.

All other values are reserved.

If FEAT_PMUv3 is not implemented, FEAT_FGT is not implemented, or EL2 is not implemented, the only permitted value is 0b0000.

FEAT_HPMN0 implements the functionality identified by the value 0b0001.

From Armv8.8, in an implementation that includes FEAT_PMUv3, FEAT_FGT, and EL2, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MTPMU, bits [3:0]

Multi-threaded PMU extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MTPMU	Meaning
0b0000	FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>.MT are read/write or RES0.
0b0001	FEAT_MTPMU and FEAT_PMUv3 implemented. PMEVTYPER<n>.MT are read/write. When FEAT_MTPMU is disabled, the Effective values of PMEVTYPER<n>.MT are 0.
0b1111	FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, PMEVTYPER<n>.MT are RES0.

All other values are reserved.

FEAT_MTPMU implements the functionality identified by the value 0b0001.

From Armv8.6, in an implementation that includes FEAT_PMUv3, the value 0b0000 is not permitted.

In an implementation that does not include FEAT_PMUv3, the value 0b0001 is not permitted.

Access to this field is RO.

Accessing ID_DFR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_DFR1()) || ImpDefBool("ID_DFR1 trapped by
        ↪HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_DFR1()) || ImpDefBool("ID_DFR1 trapped by
        ↪HCR.TID3")) && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_DFR1();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    R(t) = ID_DFR1();
elsif PSTATE.EL == EL3 then
    R(t) = ID_DFR1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.86 ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_ISAR0 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR0_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR0 are UNDEFINED.

Attributes

ID_ISAR0 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
RES0	Divide	Debug	Coproc	CmpBranch	BitField	BitCount	Swap

Bits [31:28]

Reserved, RES0.

Divide, bits [27:24]

Indicates the implemented Divide instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Divide	Meaning
0b0000	None implemented.
0b0001	Adds SDIV and UDIV in the T32 instruction set.
0b0010	As for 0b0001, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Debug, bits [23:20]

Indicates the implemented Debug instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug	Meaning
0b0000	None implemented.
0b0001	Adds BKPT.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Coproc, bits [19:16]

Indicates the implemented System register access instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Coproc	Meaning
0b0000	None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.
0b0001	Adds generic CDP, LDC, MCR, MRC, and STC.
0b0010	As for 0b0001, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.
0b0011	As for 0b0010, and adds generic MCRR and MRRC.
0b0100	As for 0b0011, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

CmpBranch, bits [15:12]

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CmpBranch	Meaning
0b0000	None implemented.
0b0001	Adds CBNZ and CBZ.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

BitField, bits [11:8]

Indicates support for BitField instructions BFC, BFI, SBFX, and UBFX.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

BitField	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

BitCount, bits [7:4]

Indicates the implemented Bit Counting instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BitCount	Meaning
0b0000	None implemented.
0b0001	Adds CLZ.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Swap, bits [3:0]

Indicates the implemented Swap instructions in the A32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Swap	Meaning
0b0000	None implemented.
0b0001	Adds SWP and SWPB.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_ISAR0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR0();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.87 ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_ISAR1 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR1_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR1 are UNDEFINED.

Attributes

ID_ISAR1 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
Jazelle	Interwork	Immediate	IfThen	Extend	Except_AR	Except	Endian

Jazelle, bits [31:28]

Indicates the implemented Jazelle extension instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Jazelle	Meaning
0b0000	No support for Jazelle.
0b0001	Adds the BXJ instruction, and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Interwork, bits [27:24]

Indicates the implemented Interworking instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Interwork	Meaning
0b0000	None implemented.
0b0001	Adds the BX instruction, and the T bit in the PSR.
0b0010	As for 0b0001, and adds the BLX instruction. PC loads have BX-like behavior.
0b0011	As for 0b0010, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

Access to this field is RO.

Immediate, bits [23:20]

Indicates support for data-processing instructions with long immediates:

- The MOVT instruction
- The MOV instruction encodings with zero-extended 16-bit immediates.
- The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Immediate	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

IfThen, bits [19:16]

Indicates the implemented If-Then instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IfThen	Meaning
0b0000	None implemented.
0b0001	Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Extend, bits [15:12]

Indicates the implemented Extend instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Extend	Meaning
0b0000	No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.
0b0001	Adds the SXTB, SXTH, UXTB, and UXTH instructions.
0b0010	As for 0b0001, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Except_AR, bits [11:8]

Indicates the implemented A and R-profile exception-handling instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Except_AR	Meaning
0b0000	None implemented.
0b0001	Adds the SRS and RFE instructions, and the A and R-profile forms of the CPS instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Except, bits [7:4]

Indicates the implemented exception-handling instructions in the A32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Except	Meaning
0b0000	Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.
0b0001	Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

Endian, bits [3:0]

Indicates the implemented Endian instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Endian	Meaning
0b0000	None implemented.
0b0001	Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Accessing ID_ISAR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR1();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.88 ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_ISAR2 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR2_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR2 are UNDEFINED.

Attributes

ID_ISAR2 is a 32-bit register.

Field descriptions

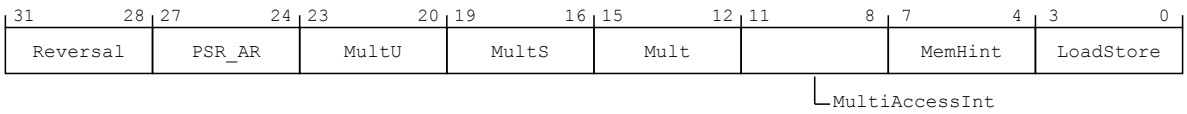

Reversal, bits [31:28]

Indicates the implemented Reversal instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Reversal	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the `REV`, `REV16`, and `REVSH` instructions.
`0b0010`	As for `0b0001`, and adds the `RBIT` instruction.

All other values are reserved.

In Armv8, the only permitted value is `0b0010`.

Access to this field is RO.

PSR_AR, bits [27:24]

Indicates the implemented A and R-profile instructions to manipulate the PSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSR_AR	Meaning
0b0000	None implemented.
0b0001	Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC, LR, #N instruction.

Access to this field is RO.

MultU, bits [23:20]

Indicates the implemented advanced unsigned Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultU	Meaning
0b0000	None implemented.
0b0001	Adds the UMULL and UMLAL instructions.
0b0010	As for 0b0001, and adds the UMAAL instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

MultS, bits [19:16]

Indicates the implemented advanced signed Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultS	Meaning
0b0000	None implemented.
0b0001	Adds the SMULL and SMLAL instructions.
0b0010	As for 0b0001, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.
0b0011	As for 0b0010, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In Armv8, the only permitted value is `0b0011`.

Access to this field is RO.

Mult, bits [15:12]

Indicates the implemented additional Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Mult	Meaning
`0b0000`	No additional instructions implemented. This means only `MUL` is implemented.
`0b0001`	Adds the `MLA` instruction.
`0b0010`	As for `0b0001`, and adds the `MLS` instruction.

All other values are reserved.

In Armv8, the only permitted value is `0b0010`.

Access to this field is RO.

MultiAccessInt, bits [11:8]

Indicates the support for interruptible multi-access instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultiAccessInt	Meaning
`0b0000`	No support. This means the `LDM` and `STM` instructions are not interruptible.
`0b0001`	`LDM` and `STM` instructions are restartable.
`0b0010`	`LDM` and `STM` instructions are continuable.

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

MemHint, bits [7:4]

Indicates the implemented Memory Hint instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MemHint	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the `PLD` instruction.
`0b0010`	Adds the `PLD` instruction. (`0b0001` and `0b0010` have identical effects.)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MemHint	Meaning
0b0011	As for 0b0001 (or 0b0010), and adds the PLI instruction.
0b0100	As for 0b0011, and adds the PLDW instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0100.

Access to this field is RO.

LoadStore, bits [3:0]

Indicates the implemented additional load/store instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LoadStore	Meaning
0b0000	No additional load/store instructions implemented.
0b0001	Adds the LDRD and STRD instructions.
0b0010	As for 0b0001, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Accessing ID_ISAR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR2();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.89 ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_ISAR3 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR3_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR3 are UNDEFINED.

Attributes

ID_ISAR3 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
T32EE	TrueNOP	T32Copy	TabBranch	SynchPrim	SVC	SIMD	Saturate

T32EE, bits [31:28]

Indicates the implemented T32EE instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

T32EE	Meaning
0b0000	None implemented.
0b0001	Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

TrueNOP, bits [27:24]

Indicates the implemented true NOP instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TrueNOP	Meaning
0b0000	None implemented. This means there are no NOP instructions that do not have any register dependencies.
0b0001	Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

T32Copy, bits [23:20]

Indicates the support for T32 non flag-setting MOV instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

T32Copy	Meaning
0b0000	Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.
0b0001	Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

TabBranch, bits [19:16]

Indicates the implemented Table Branch instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TabBranch	Meaning
0b0000	None implemented.
0b0001	Adds the TBB and TBH instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

SynchPrim, bits [15:12]

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SynchPrim	Meaning
0b0000	If SynchPrim_frac == 0b000, no Synchronization Primitives implemented.
0b0001	If SynchPrim_frac == 0b000, adds the LDREX and STREX instructions. If SynchPrim_frac == 0b011, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.
0b0010	If SynchPrim_frac == 0b000, as for [0b001, 0b011] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

SVC, bits [11:8]

Indicates the implemented SVC instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SVC	Meaning
0b0000	Not implemented.
0b0001	Adds the SVC instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

SIMD, bits [7:4]

Indicates the implemented SIMD instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMD	Meaning
0b0000	None implemented.
0b0001	Adds the SSAT and USAT instructions, and the Q bit in the PSRs.
0b0011	As for 0b0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports Advanced SIMD and floating-point instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

Access to this field is RO.

Saturate, bits [3:0]

Indicates support for Saturate instructions `QADD`, `QDADD`, `QDSUB`, and `QSUB` instructions, and the Q bit in the PSRs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Saturate	Meaning
`0b0000`	None implemented. This means no non-Advanced SIMD saturate instructions are implemented.
`0b0001`	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is `0b0001`.

Access to this field is RO.

Accessing ID_ISAR3

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
`0b1111`	`0b000`	`0b0000`	`0b0010`	`0b011`

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR3();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR3();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.90 ID_ISAR4, Instruction Set Attribute Register 4

The ID_ISAR4 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR5.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_ISAR4 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR4_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR4 are UNDEFINED.

Attributes

ID_ISAR4 is a 32-bit register.

Field descriptions

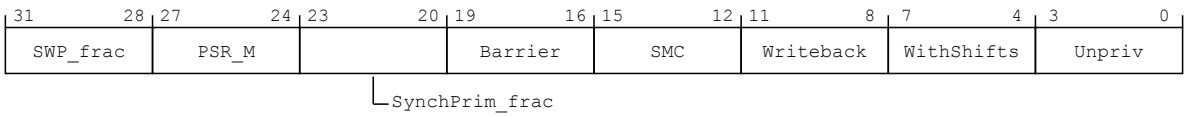

SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for `SWP` or `SWPB` instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SWP_frac	Meaning
`0b0000`	`SWP` or `SWPB` instructions not implemented.
`0b0001`	`SWP` or `SWPB` implemented but only in a uniprocessor context. `SWP` and `SWPB` do not guarantee whether memory accesses from other Requesters can come between the load memory access and the store memory access of the `SWP` or `SWPB`.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is `0b0000`.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

PSR_M, bits [27:24]

Indicates the implemented M-profile instructions to modify the PSRs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSR_M	Meaning
0b0000	None implemented.
0b0001	Adds the M-profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

SynchPrim_frac, bits [23:20]

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SynchPrim_frac	Meaning
0b0000	If SynchPrim == 0b0000, no Synchronization Primitives implemented. If SynchPrim == 0b0001, adds the LDREX and STREX instructions. If SynchPrim == 0b0010, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.
0b0011	If SynchPrim == 0b0001, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Barrier, bits [19:16]

Indicates support for Barrier instructions DMB, DSB, and ISB in the T32 and A32 instruction sets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Barrier	Meaning
0b0000	None implemented. Barrier operations are provided only as System instructions in the (coproc==0b1111) encoding space.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SMC, bits [15:12]

Indicates the implemented `SMC` instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SMC	Meaning
`0b0000`	None implemented.
`0b0001`	Adds the `SMC` instruction.

All other values are reserved.

In Armv8, the permitted values are:

- If EL3 is implemented, the only permitted value is `0b0001`.
- If neither EL3 nor EL2 is implemented, the only permitted value is `0b0000`.

Access to this field is RO.

Writeback, bits [11:8]

Indicates the support for Writeback addressing modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Writeback	Meaning
`0b0000`	Basic support. Only the `LDM`, `STM`, `PUSH`, `POP`, `SRS`, and `RFE` instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.
`0b0001`	Adds support for all of the writeback addressing modes.

All other values are reserved.

In Armv8, the only permitted value is `0b0001`.

Access to this field is RO.

WithShifts, bits [7:4]

Indicates the support for instructions with shifts.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WithShifts	Meaning
`0b0000`	Nonzero shifts supported only in `MOV` and shift instructions.
`0b0001`	Adds support for shifts of loads and stores over the range LSL 0-3.
`0b0011`	As for `0b0001`, and adds support for other constant shift options, both on load/store and other instructions.
`0b0100`	As for `0b0011`, and adds support for register-controlled shift options.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

In Armv8, the only permitted value is `0b0100`.

Access to this field is RO.

Unpriv, bits [3:0]

Indicates the implemented unprivileged instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Unpriv	Meaning
`0b0000`	None implemented. No T variant instructions are implemented.
`0b0001`	Adds the `LDRBT`, `LDRT`, `STRBT`, and `STRT` instructions.
`0b0010`	As for `0b0001`, and adds the `LDRHT`, `LDRSBT`, `LDRSHT`, and `STRHT` instructions.

All other values are reserved.

In Armv8, the only permitted value is `0b0010`.

Access to this field is RO.

Accessing ID_ISAR4

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
`0b1111`	`0b000`	`0b0000`	`0b0010`	`0b100`

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR4();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR4();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR4();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.91 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR4.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_ISAR5 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR5_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR5 are UNDEFINED.

Attributes

ID_ISAR5 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
VCMA	RDM	RES0	CRC32	SHA2	SHA1	AES	SEVL

VCMA, bits [31:28]

Indicates AArch32 support for complex number addition and multiplication where numbers are stored in vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VCMA	Meaning
0b0000	The VCMLA and VCADD instructions are not implemented in AArch32.
0b0001	The VCMLA and VCADD instructions are implemented in AArch32.

All other values are reserved.

FEAT_FCMA implements the functionality identified by 0b0001.

From Armv8.3, the value 0b0000 is not permitted.

Access to this field is RO.

RDM, bits [27:24]

Indicates support for the VQRDMLAH and VQRDMLSH instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RDM	Meaning
0b0000	No VQRDMLAH and VQRDMLSH instructions implemented.
0b0001	VQRDMLAH and VQRDMLSH instructions implemented.

All other values are reserved.

FEAT_RDM implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates support for the CRC32 instructions CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CRC32	Meaning
0b0000	CRC32 instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_CRC32 implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

SHA2, bits [15:12]

Indicates support for the SHA2 instructions SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA2	Meaning
0b0000	No SHA2 instructions implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SHA1, bits [11:8]

Indicates support for the SHA1 instructions SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA1	Meaning
0b0000	No SHA1 instructions implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

AES, bits [7:4]

Indicates support for the AES instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AES	Meaning
0b0000	No AES instructions implemented.
0b0001	AESE, AESD, AESMC, and AESIMC implemented.
0b0010	As for 0b0001, plus VMULL (polynomial) instructions operating on 64-bit data quantities.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

Access to this field is RO.

SEVL, bits [3:0]

Indicates support for the SEVL instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SEVL	Meaning
0b0000	SEVL is implemented as a NOP.
0b0001	SEVL is implemented as Send Event Local.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Accessing ID_ISAR5

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR5();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR5();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR5();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.92 ID_ISAR6, Instruction Set Attribute Register 6

The ID_ISAR6 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

AArch32 System register ID_ISAR6 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR6_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_ISAR6 are UNDEFINED.

Attributes

ID_ISAR6 is a 32-bit register.

Field descriptions

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
CLRBHB	I8MM	BF16	SPECRES	SB	FHM	DP	JSCVT

CLRBHB, bits [31:28]

Indicates support for the `CLRBHB` instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CLRBHB	Meaning
`0b0000`	`CLRBHB` instruction is not implemented.
`0b0001`	`CLRBHB` instruction is implemented.

All other values are reserved.

FEAT_CLRBHB implements the functionality identified by `0b0001`.

From Armv8.9, the value `0b0000` is not permitted.

Access to this field is RO.

I8MM, bits [27:24]

Indicates support for Advanced SIMD and floating-point Int8 matrix multiplication instructions `VSMMLA`, `VSUDOT`, `VUMMLA`, `VUSMMLA`, and `VUSDOT` in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

I8MM	Meaning
0b0000	Int8 matrix multiplication instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_AA32I8MM implements the functionality identified by 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

BF16, bits [23:20]

Indicates support for Advanced SIMD and floating-point BFloat16 instructions VCVT, VCVTB, VCVTT, VDOT, VFMAB, VFMAT, and VMMLA instructions with BF16 operand or result types in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BF16	Meaning
0b0000	BFloat16 instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_AA32BF16 implements the functionality identified by 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SPECRES, bits [19:16]

Indicates support for prediction invalidation instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SPECRES	Meaning
0b0000	Prediction invalidation instructions are not implemented.
0b0001	CFPRCTX, DVPRCTX, and CPPRCTX instructions are implemented.
0b0010	As 0b0001, and the COSPRCTX instruction is implemented.

All other values are reserved.

FEAT_SPECRES implements the functionality identified by 0b0001.

FEAT_SPECRES2 implements the functionality identified by 0b0010.

From Armv8.5, the value 0b0000 is not permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From Armv8.9, the value `0b0001` is not permitted.

Access to this field is RO.

SB, bits [15:12]

Indicates support for `SB` instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SB	Meaning
`0b0000`	`SB` instruction is not implemented.
`0b0001`	`SB` instruction is implemented.

All other values are reserved.

From Armv8.5, the value `0b0000` is not permitted.

Access to this field is RO.

FHM, bits [11:8]

Indicates support for Advanced SIMD and floating-point `VFMAL` and `VMFSL` instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FHM	Meaning
`0b0000`	`VFMAL` and `VMFSL` instructions not implemented.
`0b0001`	`VFMAL` and `VMFSL` instructions implemented.

FEAT_FHM implements the functionality identified by the value `0b0001`.

Access to this field is RO.

DP, bits [7:4]

Indicates support for dot product instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DP	Meaning
`0b0000`	No dot product instructions implemented.
`0b0001`	`VUDOT` and `VSDOT` instructions implemented.

All other values are reserved.

FEAT_DotProd implements the functionality identified by the value `0b0001`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

JSCVT, bits [3:0]

Indicates support for the JavaScript conversion instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

JSCVT	Meaning
0b0000	The VJCVT instruction is not implemented.
0b0001	The VJCVT instruction is implemented.

All other values are reserved.

In Armv8.0, the only permitted value is 0b0000.

FEAT_JSCVT implements the functionality identified by 0b0001.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the value 0b0000 is not permitted.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_ISAR6

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_ISAR6()) || ImpDefBool("ID_ISAR6 trapped by
        ↪HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_ISAR6()) || ImpDefBool("ID_ISAR6 trapped by
        ↪HCR.TID3")) && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR6();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR6();
elsif PSTATE.EL == EL3 then
    R(t) = ID_ISAR6();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.93 ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_MMFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR0_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_MMFR0 are UNDEFINED.

Attributes

ID_MMFR0 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
InnerShr	FCSE	AuxReg	TCM	ShareLvl	OuterShr	PMSA	VMSA

InnerShr, bits [31:28]

Innermost Shareability. Indicates the innermost shareability domain implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

InnerShr	Meaning
0b0000	Implemented as Non-cacheable.
0b0001	Implemented with hardware coherency support.
0b1111	Shareability ignored.

All other values are reserved.

In Armv8, the permitted values are 0b0000, 0b0001, and 0b1111.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0.ShareLvl having the value 0b0001.

When ID_MMFR0.ShareLvl is zero, this field is UNKNOWN.

Access to this field is RO.

FCSE, bits [27:24]

Indicates whether the implementation includes the FCSE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FCSE	Meaning
0b0000	Not supported.
0b0001	Support for FCSE.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxReg	Meaning
0b0000	None supported.
0b0001	Support for Auxiliary Control Register only.
0b0010	Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Note

Accesses to unimplemented Auxiliary registers are UNDEFINED.

Access to this field is RO.

TCM, bits [19:16]

Indicates support for TCMs and associated DMAs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TCM	Meaning
0b0000	Not supported.
0b0001	Support is IMPLEMENTATION DEFINED.
0b0010	Support for TCM only, Armv6 implementation.
0b0011	Support for TCM and DMA, Armv6 implementation.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of shareability levels implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ShareLvl	Meaning
0b0000	One level of shareability implemented.
0b0001	Two levels of shareability implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost shareability domain implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OuterShr	Meaning
0b0000	Implemented as Non-cacheable.
0b0001	Implemented with hardware coherency support.
0b1111	Shareability ignored.

All other values are reserved.

In Armv8, the permitted values are 0b0000, 0b0001, and 0b1111.

Access to this field is RO.

PMSA, bits [7:4]

Indicates support for a PMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMSA	Meaning
0b0000	Not supported.
0b0001	Support for IMPLEMENTATION DEFINED PMSA.
0b0010	Support for PMSAv6, with a Cache Type Register implemented.
0b0011	Support for PMSAv7, with support for memory subsections. Armv7-R profile.
0b0100	Support for PMSAv8-32, with base and limit. Armv8-R AArch32 profile.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

VMSA, bits [3:0]

Indicates support for a VMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMSA	Meaning
0b0000	Not supported.
0b0001	Support for IMPLEMENTATION DEFINED VMSA.
0b0010	Support for VMSAv6, with Cache and TLB Type Registers implemented.
0b0011	Support for VMSAv7, with support for remapping and the Access flag. ARMv7-A profile.
0b0100	As for 0b0011, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.
0b0101	As for 0b0100, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0101.

Access to this field is RO.

Accessing ID_MMFR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR0();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    R(t) = ID_MMFR0();
elsif PSTATE.EL == EL3 then
    R(t) = ID_MMFR0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.94 ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_MMFR1 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR1_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_MMFR1 are UNDEFINED.

Attributes

ID_MMFR1 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
BPred	L1TstCln	L1Uni	L1Hvd	L1UniSW	L1HvdSW	L1UniVA	L1HvdVA

BPred, bits [31:28]

Branch Predictor. Indicates branch predictor management requirements.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPred	Meaning
0b0000	No branch predictor, or no MMU present. Implies a fixed MPU configuration.
0b0001	Branch predictor requires flushing on: • Enabling or disabling a stage of address translation. • Writing new data to instruction locations. • Writing new mappings to the translation tables. • Changes to the TTBR0, TTBR1, or TTBCR registers. • Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.
0b0010	Branch predictor requires flushing on: • Enabling or disabling a stage of address translation. • Writing new data to instruction locations. • Writing new mappings to the translation tables. • Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BPred	Meaning
0b0011	Branch predictor requires flushing only on writing new data to instruction locations.
0b0100	For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In Armv8, the permitted values are 0b0010, 0b0011, or 0b0100. For values other than 0b0000 and 0b0100, the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

Access to this field is RO.

L1TstCln, bits [27:24]

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1TstCln	Meaning
0b0000	None supported.
0b0001	Supported Level 1 data cache test and clean operations are: • Test and clean data cache.
0b0010	As for 0b0001, and adds: • Test, clean, and invalidate data cache.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1Uni, bits [23:20]

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Uni	Meaning
0b0000	None supported.
0b0001	Supported entire Level 1 cache operations are: • Invalidate cache, including branch predictor if appropriate. • Invalidate branch predictor, if appropriate.
0b0010	As for 0b0001, and adds: • Clean cache, using a recursive model that uses the cache dirty status bit. • Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

L1Hvd, bits [19:16]

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Hvd	Meaning
`0b0000`	None supported.
`0b0001`	Supported entire Level 1 cache operations are: • Invalidate instruction cache, including branch predictor if appropriate. • Invalidate branch predictor, if appropriate.
`0b0010`	As for `0b0001`, and adds: • Invalidate data cache. • Invalidate data cache and instruction cache, including branch predictor if appropriate.
`0b0011`	As for `0b0010`, and adds: • Clean data cache, using a recursive model that uses the cache dirty status bit. • Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

L1UniSW, bits [15:12]

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1UniSW	Meaning
`0b0000`	None supported.
`0b0001`	Supported Level 1 unified cache line maintenance operations by set/way are: • Clean cache line by set/way.
`0b0010`	As for `0b0001`, and adds: • Clean and invalidate cache line by set/way.
`0b0011`	As for `0b0010`, and adds: • Invalidate cache line by set/way.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

L1HvdSW, bits [11:8]

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdSW	Meaning
`0b0000`	None supported.
`0b0001`	Supported Level 1 Harvard cache line maintenance operations by set/way are: • Clean data cache line by set/way. • Clean and invalidate data cache line by set/way.
`0b0010`	As for `0b0001`, and adds: • Invalidate data cache line by set/way.
`0b0011`	As for `0b0010`, and adds: • Invalidate instruction cache line by set/way.

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

L1UniVA, bits [7:4]

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1UniVA	Meaning
`0b0000`	None supported.
`0b0001`	Supported Level 1 unified cache line maintenance operations by VA are: • Clean cache line by VA. • Invalidate cache line by VA. • Clean and invalidate cache line by VA.
`0b0010`	As for `0b0001`, and adds: • Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L1HvdVA, bits [3:0]

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdVA	Meaning
0b0000	None supported.
0b0001	Supported Level 1 Harvard cache line maintenance operations by VA are: • Clean data cache line by VA. • Invalidate data cache line by VA. • Clean and invalidate data cache line by VA. • Clean instruction cache line by VA.
0b0010	As for 0b0001, and adds: • Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_MMFR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR1();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR1();
elsif PSTATE.EL == EL3 then
    R(t) = ID_MMFR1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.95 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_MMFR2 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR2_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_MMFR2 are UNDEFINED.

Attributes

ID_MMFR2 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
HWAccFlg	WFIStall	MemBarr	UniTLB	HvdTLB	L1HvdRng	L1HvdBG	L1HvdFG

HWAccFlg, bits [31:28]

Hardware Access Flag. In earlier versions of the Arm Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HWAccFlg	Meaning
0b0000	Not supported.
0b0001	Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

WFIStall, bits [27:24]

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WFIStall	Meaning
0b0000	Not supported.
0b0001	Support for WFI stalling.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

MemBarr, bits [23:20]

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc == 1111) encoding space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MemBarr	Meaning
0b0000	None supported.
0b0001	Supported memory barrier System instructions are: • Data Synchronization Barrier (DSB).
0b0010	As for 0b0001, and adds: • Instruction Synchronization Barrier (ISB). • Data Memory Barrier (DMB).

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Arm deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

Access to this field is RO.

UniTLB, bits [19:16]

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

UniTLB	Meaning
0b0000	Not supported.
0b0001	Supported unified TLB maintenance operations are: • Invalidate all entries in the TLB. • Invalidate TLB entry by VA.
0b0010	As for 0b0001, and adds: • Invalidate TLB entries by ASID match.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UniTLB	Meaning
0b0011	As for 0b0010, and adds: • Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation
0b0100	As for 0b0011, and adds: • Invalidate Hyp mode unified TLB entry by VA. • Invalidate entire Non-secure PL1&0 unified TLB. • Invalidate entire Hyp mode unified TLB.
0b0101	As for 0b0100, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL, TLBIMVALH.
0b0110	As for 0b0101, and adds the following operations: TLBIIPAS2IS, TLBIIPAS2LIS, TLBIIPAS2, TLBIIPAS2L.

All other values are reserved.

In Armv8, the only permitted value is 0b0110.

Access to this field is RO.

HvdTLB, bits [15:12]

If the value of ID_MMFR2.UniTLB is not 0b0000, then the meaning of this field is IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1HvdRng, bits [11:8]

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdRng	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache maintenance range operations are: • Invalidate data cache range by VA. • Invalidate instruction cache range by VA. • Clean data cache range by VA. • Clean and invalidate data cache range by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdBG, bits [7:4]

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdBG	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache background fetch operations are: • Fetch instruction cache range by VA. • Fetch data cache range by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdFG	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache foreground fetch operations are: • Fetch instruction cache range by VA. • Fetch data cache range by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_MMFR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR2();
elsif PSTATE.EL == EL3 then
    R(t) = ID_MMFR2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.96 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_MMFR3 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR3_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_MMFR3 are UNDEFINED.

Attributes

ID_MMFR3 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
Supersec	CMemSz	CohWalk	PAN	MaintBcst	BPMaint	CMaintSW	CMaintVA

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Supersec	Meaning
0b0000	Supersections supported.
0b1111	Supersections not supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b1111.

Access to this field is RO.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CMemSz	Meaning
0b0000	4GB, corresponding to a 32-bit physical address range.
0b0001	64GB, corresponding to a 36-bit physical address range.
0b0010	1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CohWalk	Meaning
0b0000	Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.
0b0001	Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

PAN, bits [19:16]

Privileged Access Never. Indicates support for the PAN bit in CPSR, SPSR, and DSPSR in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PAN	Meaning
0b0000	PAN not supported.
0b0001	PAN supported.
0b0010	PAN supported and ATS1CPRP and ATS1CPWP instructions supported.

All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

From Armv8.2, the value 0b0001 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MaintBcst, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MaintBcst	Meaning
`0b0000`	Cache, TLB, and branch predictor operations only affect local structures.
`0b0001`	Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.
`0b0010`	Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8, the only permitted value is `0b0010`.

Access to this field is RO.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPMaint	Meaning
`0b0000`	None supported.
`0b0001`	Supported branch predictor maintenance operations are: • Invalidate all branch predictors.
`0b0010`	As for `0b0001`, and adds: • Invalidate branch predictors by VA.

All other values are reserved.

In Armv8, the only permitted value is `0b0010`.

Access to this field is RO.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CMaintSW	Meaning
0b0000	None supported.
0b0001	Supported hierarchical cache maintenance instructions by set/way are: • Invalidate data cache by set/way. • Clean data cache by set/way. • Clean and invalidate data cache by set/way.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

Access to this field is RO.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CMaintVA	Meaning
0b0000	None supported.
0b0001	Supported hierarchical cache maintenance operations by VA are: • Invalidate data cache by VA. • Clean data cache by VA. • Clean and invalidate data cache by VA. • Invalidate instruction cache by VA. • Invalidate all instruction cache entries.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Access to this field is RO.

Accessing ID_MMFR3

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR3();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR3();
elsif PSTATE.EL == EL3 then
    R(t) = ID_MMFR3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.97 ID_MMFR4, Memory Model Feature Register 4

The ID_MMFR4 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_MMFR4 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR4_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_MMFR4 are UNDEFINED.

Attributes

ID_MMFR4 is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
EVT	CCIDX	LSM	HPDS	CnP	XNX	AC2	SpecSEI

EVT, bits [31:28]

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR2.{TTLBIS, TOCU, TICAB, TID4} traps.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EVT	Meaning
0b0000	HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are not supported.
0b0001	HCR2.{TOCU, TICAB, TID4} traps are supported.
0b0010	As 0b0001, and HCR2.TTLBIS trap is supported.

All other values are reserved.

FEAT_EVT implements the functionality identified by the value 0b0001.

FEAT_EVT2 implements the functionality identified by the value 0b0010.

If EL2 is not implemented or does not support AArch32, the only permitted value is 0b0000.

From Armv8.5, if EL2 is implemented and supports AArch32, the value 0b0001 is not permitted.

Access to this field is RO.

CCIDX, bits [27:24]

Support for use of the revised CCSIDR format and the presence of the CCSIDR2 is indicated.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCIDX	Meaning
0b0000	32-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is not implemented.
0b0001	64-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is implemented.

All other values are reserved.

FEAT_CCIDX implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

LSM, bits [23:20]

Indicates support for LSMAOE and nTLSMD bits in HSCTLR and SCTLR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LSM	Meaning
0b0000	LSMAOE and nTLSMD bits not supported.
0b0001	LSMAOE and nTLSMD bits supported.

All other values are reserved.

FEAT_LSMAOC implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

HPDS, bits [19:16]

Hierarchical permission disables bits in translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HPDS	Meaning
0b0000	Disabling of hierarchical controls not supported.
0b0001	Supports disabling of hierarchical controls using the TTBCR2.HPD0, TTBCR2.HPD1, and HTCR.HPD bits.
0b0010	As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

FEAT_AA32HPD implements the functionality identified by the value 0b0001.

FEAT_HPDS2 implements the functionality added by the value 0b0010.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The value `0b0000` implies that the encoding for TTBCR2 is UNDEFINED.

Access to this field is RO.

CnP, bits [15:12]

Common not Private translations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CnP	Meaning
`0b0000`	Common not Private translations not supported.
`0b0001`	Common not Private translations supported.

All other values are reserved.

FEAT_TTCNP implements the functionality identified by the value `0b0001`.

From Armv8.2, the value `0b0000` is not permitted.

Access to this field is RO.

XNX, bits [11:8]

Support for execute-never control distinction by Exception level at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

XNX	Meaning
`0b0000`	Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
`0b0001`	Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

FEAT_XNX implements the functionality identified by the value `0b0001`.

When FEAT_XNX is implemented:

- If all of the following conditions are true, it is IMPLEMENTATION DEFINED whether the value of ID_MMFR4.XNX is `0b0000` or `0b0001`:
  - ID_AA64MMFR1_EL1.XNX ==1.
  - EL2 cannot use AArch32.
  - EL1 can use AArch32.
- If EL2 can use AArch32 then the value `0b0000` is not permitted.

Access to this field is RO.

AC2, bits [7:4]

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AC2	Meaning
0b0000	ACTLR2 and HACTLR2 are not implemented.
0b0001	ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.2, the only permitted value is 0b0001.

Access to this field is RO.

SpecSEI, bits [3:0]

When FEAT_RAS is implemented:

Describes whether the PE can generate SError exceptions from speculative reads of memory, including speculative instruction fetches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SpecSEI	Meaning
0b0000	The PE never generates an SError exception due to an External abort on a speculative read.
0b0001	The PE might generate an SError exception due to an External abort on a speculative read.

All other values are reserved.

FEAT_SpecSEI implements the functionality identified by the value 0b0001.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing ID_MMFR4

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR4()) || ImpDefBool("ID_MMFR4 trapped by
         ↪HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR4()) || ImpDefBool("ID_MMFR4 trapped by
         ↪HCR.TID3")) && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR4();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR4();
elsif PSTATE.EL == EL3 then
    R(t) = ID_MMFR4();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.98 ID_MMFR5, Memory Model Feature Register 5

The ID_MMFR5 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_MMFR5 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR5_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_MMFR5 are UNDEFINED.

Attributes

ID_MMFR5 is a 32-bit register.

Field descriptions

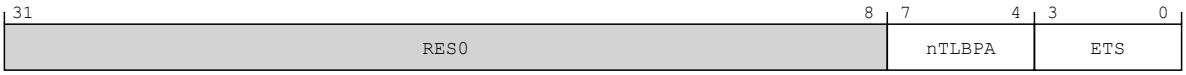

Bits [31:8]

Reserved, RES0.

nTLBPA, bits [7:4]

Indicates support for intermediate caching of translation table walks.

The value of this field is an IMPLEMENTATION DEFINED choice of:

nTLBPA	Meaning
0b0000	The intermediate caching of translation table walks might include non-coherent physical translation caches.
0b0001	The intermediate caching of translation table walks does not include non-coherent physical translation caches.

Non-coherent physical translation caches are non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE, where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

FEAT_nTLBPA implements the functionality identified by the value 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ETS, bits [3:0]

Indicates support for Enhanced Translation Synchronization.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ETS	Meaning
0b0000	Enhanced Translation Synchronization is not supported.
0b0001	Enhanced Translation Synchronization is not supported.
0b0010	FEAT_ETS2 is implemented.
0b0011	FEAT_ETS3 is implemented.

All other values are reserved.

FEAT_ETS2 implements the functionality identified by the value 0b0010.

FEAT_ETS3 implements the functionality identified by the value 0b0011.

From Armv8.8, the values 0b0000 and 0b0001 are not permitted.

From Armv9.5, the value 0b0010 is not permitted.

Access to this field is RO.

Accessing ID_MMFR5

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0011	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR5()) || ImpDefBool("ID_MMFR5 trapped by
        ↪HCR_EL2.TID3")) && HCR_EL2().TID3 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪(IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR5()) || ImpDefBool("ID_MMFR5 trapped by
        ↪HCR.TID3")) && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR5();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    R(t) = ID_MMFR5();
elsif PSTATE.EL == EL3 then
    R(t) = ID_MMFR5();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.99 ID_PFR0, Processor Feature Register 0

The ID_PFR0 characteristics are:

Purpose

Gives top-level information about the instruction sets and other features supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_PFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_PFR0_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_PFR0 are UNDEFINED.

Attributes

ID_PFR0 is a 32-bit register.

Field descriptions

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
RAS	DIT	AMU	CSV2	State3	State2	State1	State0

RAS, bits [31:28]

RAS Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS	Meaning
0b0000	The RAS Extension is not implemented.
0b0001	The RAS Extension is implemented, FEAT_RAS provides the ESB instruction and the Error synchronization event.
0b0010	FEAT_RASv1p1 implemented. As 0b0001, and adds support for additional ERXMISC<m> System registers. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.
0b0011	FEAT_RASv2 implemented. As 0b0010, and requires that error records accessed through System registers conform to either RAS System Architecture v1.1 or RAS System Architecture v2.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0001.

FEAT_DoubleFault implements the functionality identified by the value 0b0010.

When ERRIDR.NUM is zero:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_DoubleFault is not implemented, then this field is permitted to have the value `0b0001`, regardless of whether FEAT_RASv1p1 or FEAT_RASv2 are implemented.
- If FEAT_DoubleFault is implemented, then this field is permitted to have the value `0b0010`, regardless of whether FEAT_RASv2 is implemented.

When ERRIDR.NUM is nonzero:

- FEAT_RASv1p1 implements the functionality identified by the value `0b0010`.
- FEAT_RASv2 implements the functionality identified by the value `0b0011`.

From Armv8.2, the value `0b0000` is not permitted.

From Armv8.4, if FEAT_DoubleFault is implemented or ERRIDR.NUM is nonzero, the value `0b0001` is not permitted.

Note

When the value of this field is `0b0001`, ID_PFR2.RAS_frac indicates whether FEAT_RASv1p1 is implemented.

From Armv8.9, if ERRIDR.NUM is nonzero, the value `0b0010` is not permitted.

Access to this field is RO.

DIT, bits [27:24]

Data Independent Timing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DIT	Meaning
`0b0000`	AArch32 does not guarantee constant execution time of any instructions.
`0b0001`	AArch32 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

FEAT_DIT implements the functionality identified by the value `0b0001`.

From Armv8.4, the value `0b0000` is not permitted.

Access to this field is RO.

AMU, bits [23:20]

Indicates support for Activity Monitors Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AMU	Meaning
`0b0000`	Activity Monitors Extension is not implemented.
`0b0001`	FEAT_AMUv1 is implemented.
`0b0010`	FEAT_AMUv1p1 is implemented. As `0b0001` and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_AMUv1 implements the functionality identified by the value `0b0001`.

FEAT_AMUv1p1 implements the functionality identified by the value `0b0010`.

In Armv8.0, the only permitted value is `0b0000`.

In Armv8.4, the permitted values are `0b0000` and `0b0001`.

From Armv8.6, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

Access to this field is RO.

CSV2, bits [19:16]

Speculative use of out of context branch targets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV2	Meaning
`0b0000`	The implementation does not disclose whether FEAT_CSV2 is implemented.
`0b0001`	FEAT_CSV2 is implemented, but FEAT_CSV2_1p1 is not implemented.
`0b0010`	FEAT_CSV2_1p1 is implemented.

All other values are reserved.

FEAT_CSV2 implements the functionality identified by the value `0b0001`.

FEAT_CSV2_1p1 implements the functionality identified by the value `0b0010`.

From Armv8.5, the permitted values are `0b0001` and `0b0010`.

Access to this field is RO.

State3, bits [15:12]

T32EE instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State3	Meaning
`0b0000`	Not implemented.
`0b0001`	T32EE instruction set implemented.

All other values are reserved.

In Armv8, the only permitted value is `0b0000`.

Access to this field is RO.

State2, bits [11:8]

Jazelle extension support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

State2	Meaning
0b0000	Not implemented.
0b0001	Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.
0b0010	Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

State1, bits [7:4]

T32 instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State1	Meaning
0b0000	T32 instruction set not implemented.
0b0001	T32 encodings before the introduction of Thumb-2 technology implemented: • All instructions are 16-bit. • A BL or BLX is a pair of 16-bit instructions. • 32-bit instructions other than BL and BLX cannot be encoded.
0b0011	T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

Access to this field is RO.

State0, bits [3:0]

A32 instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State0	Meaning
0b0000	A32 instruction set not implemented.
0b0001	A32 instruction set implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Accessing ID_PFR0

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_PFR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_PFR0();
elsif PSTATE.EL == EL3 then
    R(t) = ID_PFR0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.100 ID_PFR1, Processor Feature Register 1

The ID_PFR1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_PFR1 bits [31:0] are architecturally mapped to AArch64 System register ID_PFR1_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_PFR1 are UNDEFINED.

Attributes

ID_PFR1 is a 32-bit register.

Field descriptions

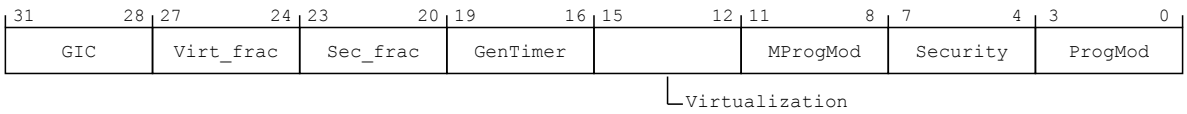

GIC, bits [31:28]

System register GIC CPU interface.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface system registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

Access to this field is RO.

Virt_frac, bits [27:24]

Virtualization fractional field. When the Virtualization field is 0b0000, determines the support for Virtualization Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Virt_frac	Meaning
0b0000	No Virtualization Extensions are implemented.
0b0001	The following Virtualization Extensions are implemented: • The SCR.SIF bit, if EL3 is implemented. • The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented. • The MSR (banked register) and MRS (banked register) instructions. • The ERET instruction.

All other values are reserved.

In Armv8, the permitted values are:

- 0b0000 when EL2 is implemented.
- 0b0001 when EL2 is not implemented.

This field is valid only when the value of ID_PFR1.Virtualization is 0, otherwise it holds the value 0b0000.

Note

The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

Access to this field is RO.

Sec_frac, bits [23:20]

Security fractional field. When the Security field is 0b0000, determines the support for Security Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Sec_frac	Meaning
0b0000	No Security Extensions are implemented.
0b0001	The following Security Extensions are implemented: • The VBAR register. • The TTBCR.PD0 and TTBCR.PD1 bits.
0b0010	As for 0b0001, and the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8, the permitted values are:

- 0b0000 when EL3 is implemented.
- 0b0001 or 0b0010 when EL3 is not implemented.

This field is valid only when the value of ID_PFR1.Security is 0, otherwise it holds the value 0b0000.

Access to this field is RO.

GenTimer, bits [19:16]

Generic Timer support.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

GenTimer	Meaning
0b0000	Generic Timer is not implemented.
0b0001	Generic Timer is implemented.
0b0010	Generic Timer is implemented, and also includes support for CNTHCTL.EVNTIS and CNTKCTL.EVNTIS fields, and CNTPCTSS and CNTVCTSS counter views.

All other values are reserved.

FEAT_ECV implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0001.

From Armv8.6, the only permitted value is 0b0010.

Access to this field is RO.

Virtualization, bits [15:12]

Virtualization support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Virtualization	Meaning
0b0000	EL2, Hyp mode, and the HVC instruction not implemented.
0b0001	EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == 0b0001 implemented.

All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 when EL2 is not implemented.
- 0b0001 when EL2 is implemented.

In an implementation that includes EL2, if EL2 cannot use AArch32 but EL1 can use AArch32, then this field has the value 0b0001.

Note

The ID_ISARs do not identify whether the HVC instruction is implemented.

Access to this field is RO.

MProgMod, bits [11:8]

M-profile programmers' model support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MProgMod	Meaning
0b0000	Not supported.
0b0010	Support for two-stack programmers' model.

All other values are reserved.

In Armv8-A and Armv8-R, the only permitted value is 0b0000.

Access to this field is RO.

Security, bits [7:4]

Security support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Security	Meaning
0b0000	EL3, Monitor mode, and the SMC instruction not implemented.
0b0001	EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0b0001 implemented.
0b0010	As for 0b0001, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In Armv8, the permitted values are:

- 0b0000 when EL3 is not implemented.
- 0b0001 when EL3 is implemented.

In an implementation that includes EL3, if EL3 cannot use AArch32 but EL1 can use AArch32, then this field has the value 0b0001.

Access to this field is RO.

ProgMod, bits [3:0]

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ProgMod	Meaning
0b0000	Not supported.
0b0001	Supported.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ID_PFR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_PFR1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_PFR1();
elsif PSTATE.EL == EL3 then
    R(t) = ID_PFR1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.101 ID_PFR2, Processor Feature Register 2

The ID_PFR2 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0 and ID_PFR1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

AArch32 System register ID_PFR2 bits [31:0] are architecturally mapped to AArch64 System register ID_PFR2_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ID_PFR2 are UNDEFINED.

Attributes

ID_PFR2 is a 32-bit register.

Field descriptions

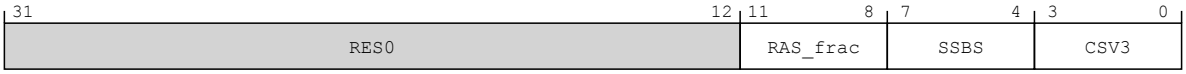

Bits [31:12]

Reserved, RES0.

RAS_frac, bits [11:8]

RAS Extension fractional field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS_frac	Meaning
0b0000	If ID_PFR0.RAS == 0b0001, support for the Reliability, Availability, and Serviceability Extension is implemented.
0b0001	If ID_PFR0.RAS == 0b0001, as 0b0000 and adds support for additional ERXMISC<m> System registers. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0000.

When ERRIDR.NUM is zero, this field is permitted to have the value 0b0000, regardless of whether FEAT_RASv1p1 is implemented.

When ERRIDR.NUM is nonzero, FEAT_RASv1p1 implements the functionality identified by the value 0b0001.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is valid only if ID_PFR0.RAS == 0b0001.

Access to this field is RO.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SSBS	Meaning
0b0000	AArch32 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001	AArch32 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

All other values are reserved.

Access to this field is RO.

CSV3, bits [3:0]

Speculative use of faulting data.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV3	Meaning
0b0000	This PE does not disclose whether data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, can be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.
0b0001	Data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, cannot be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.

All other values are reserved.

FEAT_CSV3 implements the functionality identified by the value 0b0001.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

Access to this field is RO.

Accessing ID_PFR2

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_PFR2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_PFR2();
elsif PSTATE.EL == EL3 then
    R(t) = ID_PFR2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.102 IFAR, Instruction Fault Address Register

The IFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception.

Configuration

AArch32 System register IFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL1[63:32].

When EL2 is implemented and EL3 is implemented, AArch32 System register IFAR bits [31:0] are architecturally mapped to AArch32 System register HIFAR[31:0].

This register is banked between IFAR and IFAR_S and IFAR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to IFAR are UNDEFINED.

Attributes

IFAR is a 32-bit register.

This register has the following instances:

- IFAR, when EL3 is not implemented or FEAT_AA64 is implemented.
- IFAR_S, when FEAT_AA32EL3 is implemented.
- IFAR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
VA of faulting address of synchronous Prefetch Abort exception

VA, bits [31:0]

VA of faulting address of synchronous Prefetch Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing IFAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = IFAR_NS();
    else
        R(t) = IFAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = IFAR_NS();
    else
        R(t) = IFAR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = IFAR_S();
    else
        R(t) = IFAR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T6 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T6 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        IFAR_NS() = R(t);
    else
        IFAR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        IFAR_NS() = R(t);
    else
        IFAR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        IFAR_S() = R(t);
    else
        IFAR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.103 IFSR, Instruction Fault Status Register

The IFSR characteristics are:

Purpose

Holds status information about the last instruction fault.

Configuration

The current translation table format determines which format of the register is used.

AArch32 System register IFSR bits [31:0] are architecturally mapped to AArch64 System register IFSR32_EL2[31:0].

This register is banked between IFSR and IFSR_S and IFSR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to IFSR are UNDEFINED.

Attributes

IFSR is a 32-bit register.

This register has the following instances:

- IFSR, when EL3 is not implemented or FEAT_AA64 is implemented.
- IFSR_S, when FEAT_AA32EL3 is implemented.
- IFSR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

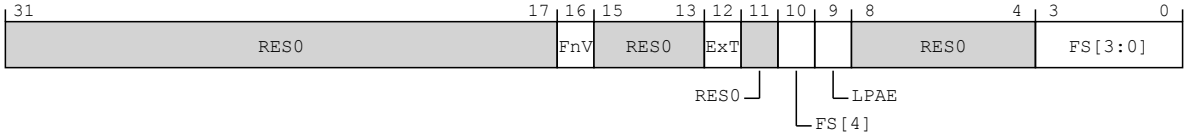

Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	IFAR is valid.
0b1	IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]

Fault Status bits. Bits [10] and [3:0] are interpreted together.

FS	Meaning	Applies when
0b00001	PC alignment fault.	
0b00010	Debug exception.	
0b00011	Access flag fault, level 1.	
0b00101	Translation fault, level 1.	
0b00110	Access flag fault, level 2.	
0b00111	Translation fault, level 2.	
0b01000	Synchronous External abort, not on translation table walk.	
0b01001	Domain fault, level 1.	
0b01011	Domain fault, level 2.	
0b01100	Synchronous External abort, on translation table walk, level 1.	
0b01101	Permission fault, level 1.	
0b01110	Synchronous External abort, on translation table walk, level 2.	
0b01111	Permission fault, level 2.	
0b10000	TLB conflict abort.	
0b10100	IMPLEMENTATION DEFINED fault (Lockdown fault).	
0b11001	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b11100	Synchronous parity or ECC error on translation table walk, level 1.	FEAT_RAS is not implemented
0b11110	Synchronous parity or ECC error on translation table walk, level 2.	FEAT_RAS is not implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Short-descriptor translation table lookup'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

LPAE	Meaning
0b0	Using the Short-descriptor translation table formats.
0b1	Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

When TTBCR.EAE == '1':

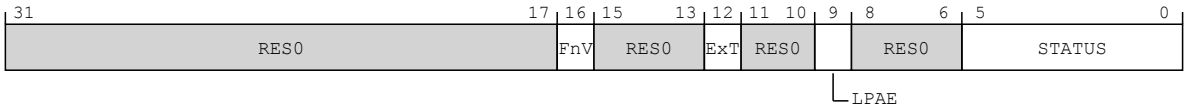

Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	IFAR is valid.
0b1	IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

LPAE	Meaning
0b0	Using the Short-descriptor translation table formats.
0b1	Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

STATUS	Meaning	Applies when
0b000000	Address size fault in translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STATUS	Meaning	Applies when
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk.	
0b010101	Synchronous External abort on translation table walk, level 1.	
0b010110	Synchronous External abort on translation table walk, level 2.	
0b010111	Synchronous External abort on translation table walk, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3.	FEAT_RAS is not implemented
0b100001	PC alignment fault.	
0b100010	Debug exception.	
0b110000	TLB conflict abort.	

All other values are reserved.

When FEAT_RAS is implemented, 0b011000, 0b011101, 0b011110, and 0b011111 are reserved.

For more information about the lookup level associated with a fault, see 'The level associated with MMU faults on a Long-descriptor translation table lookup'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing IFSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = IFSR_NS();
    else
        R(t) = IFSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = IFSR_NS();
    else
        R(t) = IFSR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = IFSR_S();
    else
        R(t) = IFSR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        IFSR_NS() = R(t);
    else
        IFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        IFSR_NS() = R(t);
    else
        IFSR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        IFSR_S() = R(t);
    else
        IFSR_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.104 ISR, Interrupt Status Register

The ISR characteristics are:

Purpose

Shows the pending status of the IRQ and FIQ interrupts and the SError exceptions.

Configuration

AArch32 System register ISR bits [31:0] are architecturally mapped to AArch64 System register ISR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ISR are UNDEFINED.

Attributes

ISR is a 32-bit register.

Field descriptions

31 … 9	8	7	6	5 … 0
RES0	A	I	F	RES0

Bits [31:9]

Reserved, RES0.

A, bit [8]

SError exception pending bit:

A	Meaning
0b0	No pending SError exception.
0b1	An SError exception is pending.

If all of the following apply then this field shows the pending status of virtual SError exceptions:

- EL2 is implemented and enabled in the current Security state.
- Any of the following apply:
  - EL2 is using AArch64 and HCR_EL2.AMO is 1.
  - EL2 is using AArch64, FEAT_DoubleFault2 is implemented, and the Effective value of HCRX_EL2.TMEA is 1.
  - EL2 is using AArch32 and HCR.AMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical SError exceptions.

If the SError exception is edge-triggered, this field is cleared to zero when the physical SError exception is taken.

I, bit [7]

IRQ pending bit. Indicates whether an IRQ interrupt is pending:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I	Meaning
0b0	No pending IRQ.
0b1	An IRQ interrupt is pending.

If all of the following apply then this field shows the pending status of virtual IRQ interrupts:

- EL2 is implemented and enabled in the current Security state.
- Any of the following apply:
  - EL2 is using AArch64 and HCR_EL2.IMO is 1.
  - EL2 is using AArch32 and HCR.IMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical IRQ interrupts.

F, bit [6]

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

F	Meaning
0b0	No pending FIQ.
0b1	An FIQ interrupt is pending.

If all of the following apply then this field shows the pending status of virtual FIQ interrupts:

- EL2 is implemented and enabled in the current Security state.
- Any of the following apply:
  - EL2 is using AArch64 and HCR_EL2.FMO is 1.
  - EL2 is using AArch32 and HCR.FMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical FIQ interrupts.

Bits [5:0]

Reserved, RES0.

Accessing ISR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ISR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ISR();
elsif PSTATE.EL == EL3 then
    R(t) = ISR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.105 ITLBIALL, Instruction TLB Invalidate All

The ITLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ITLBIALL are UNDEFINED.

Attributes

ITLBIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing ITLBIALL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    AArch32_ITLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, VMID(), Broadcast_NSH, TLBI_AllAttr);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.106 ITLBIASID, Instruction TLB Invalidate by ASID match

The ITLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ITLBIASID are UNDEFINED.

Attributes

ITLBIASID is a 32-bit System instruction.

Field descriptions

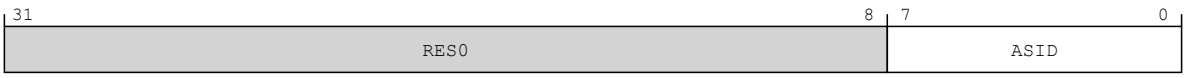

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing ITLBIASID

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
            ↪R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_ITLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBI_AllAttr,
        ↪R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.107 ITLBIMVA, Instruction TLB Invalidate by VA

The ITLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ITLBIMVA are UNDEFINED.

Attributes

ITLBIMVA is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing ITLBIMVA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_ITLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.108 JIDR, Jazelle ID Register

The JIDR characteristics are:

Purpose

A Jazelle register, which identified the Jazelle architecture version.

Configuration

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to JIDR are UNDEFINED.

Attributes

JIDR is a 32-bit register.

Field descriptions

31	0
RAZ	

Bits [31:0]

Reserved, RAZ.

Accessing JIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ImpDefBool("JIDR UNDEFINED at EL0") then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HCR_EL2().TID0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TID0 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = JIDR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID0 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID0 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = JIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = JIDR();
elsif PSTATE.EL == EL3 then
    R(t) = JIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.109 JMCR, Jazelle Main Configuration Register

The JMCR characteristics are:

Purpose

A Jazelle register, which provides control of the Jazelle extension.

Configuration

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to JMCR are UNDEFINED.

Attributes

JMCR is a 32-bit register.

Field descriptions

31	0
RAZ/WI	

Bits [31:0]

Reserved, RAZ/WI.

Accessing JMCR

For accesses from EL0 it is IMPLEMENTATION DEFINED whether the register is RW or UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ImpDefBool("JMCR UNDEFINED at EL0") then
        Undefined();
    else
        R(t) = JMCR();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = JMCR();
elsif PSTATE.EL == EL2 then
    R(t) = JMCR();
elsif PSTATE.EL == EL3 then
    R(t) = JMCR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ImpDefBool("JMCR UNDEFINED at EL0") then
        Undefined();
    else
        return;
    end;
elsif PSTATE.EL == EL1 then
    return;
elsif PSTATE.EL == EL2 then
    return;
elsif PSTATE.EL == EL3 then
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.110 JOSCR, Jazelle OS Control Register

The JOSCR characteristics are:

Purpose

A Jazelle register, which provides operating system control of the Jazelle Extension.

Configuration

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to JOSCR are UNDEFINED.

Attributes

JOSCR is a 32-bit register.

Field descriptions

31	0
RAZ/WI	

Bits [31:0]

Reserved, RAZ/WI.

Accessing JOSCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ImpDefBool("JOSCR UNDEFINED at EL0") then
        Undefined();
    else
        R(t) = JOSCR();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = JOSCR();
elsif PSTATE.EL == EL2 then
    R(t) = JOSCR();
elsif PSTATE.EL == EL3 then
    R(t) = JOSCR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if ImpDefBool("JOSCR UNDEFINED at EL0") then
        Undefined();
    else
        return;
    end;
elsif PSTATE.EL == EL1 then
    return;
elsif PSTATE.EL == EL2 then
    return;
elsif PSTATE.EL == EL3 then
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.111 MAIR0, Memory Attribute Indirection Register 0

The MAIR0 characteristics are:

Purpose

Along with MAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

Configuration

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

When EL3 is using AArch32, write access to MAIR0(S) is disabled when the **CP15SDISABLE** signal is asserted HIGH.

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register MAIR0 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[31:0].

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register MAIR0 bits [31:0] are architecturally mapped to AArch32 System register PRRR[31:0].

When EL3 is using AArch32, AArch32 System register MAIR0 bits [31:0] are architecturally mapped to AArch32 System register PRRR[31:0].

When EL3 is using AArch32, AArch32 System register MAIR0 bits [31:0] are architecturally mapped to AArch32 System register PRRR[31:0].

This register is banked between MAIR0 and MAIR0_S and MAIR0_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MAIR0 are UNDEFINED.

Attributes

MAIR0 is a 32-bit register.

This register has the following instances:

- MAIR0, when EL3 is not implemented or FEAT_AA64 is implemented.
- MAIR0_S, when FEAT_AA32EL3 is implemented.
- MAIR0_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '1':

31 – 24	23 – 16	15 – 8	7 – 0
Attr3	Attr2	Attr1	Attr0

Attr<n> , bits [8n+7:8n], for n = 3 to 0

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
`0b00`RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.
`0b01`RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
`0b10`RW	Normal memory, Outer Write-Through Non-transient.
`0b11`RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is `0b0000`	Meaning when Attr<n>[7:4] is not `0b0000`
`0b0000`	Device-nGnRnE memory	UNPREDICTABLE
`0b00`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
`0b0100`	Device-nGnRE memory	Normal memory, Inner Non-cacheable
`0b01`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
`0b1000`	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=`0b00`)
`0b10`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
`0b1100`	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=`0b00`)
`0b11`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
`0b0`	No Allocate
`0b1`	Allocate

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing MAIR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR0_NS();
        else
            R(t) = PRRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR0();
        else
            R(t) = PRRR();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR0_NS();
        else
            R(t) = PRRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR0();
        else
            R(t) = PRRR();
        end;
    end;
elsif PSTATE.EL == EL3 then
    if TTBCR().EAE == '1' then
        if SCR().NS == '0' then
            R(t) = MAIR0_S();
        else
            R(t) = MAIR0_NS();
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if SCR().NS == '0' then
            R(t) = PRRR_S();
        else
            R(t) = PRRR_NS();
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR0_NS() = R(t);
        else
            PRRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR0() = R(t);
        else
            PRRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR0_NS() = R(t);
        else
            PRRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR0() = R(t);
        else
            PRRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if TTBCR().EAE == '1' then
            if SCR().NS == '0' then
                MAIR0_S() = R(t);
            else
                MAIR0_NS() = R(t);
            end;
        else
            if SCR().NS == '0' then
                PRRR_S() = R(t);
            else
                PRRR_NS() = R(t);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.112 MAIR1, Memory Attribute Indirection Register 1

The MAIR1 characteristics are:

Purpose

Along with MAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

Configuration

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

When EL3 is using AArch32, write access to MAIR1(S) is disabled when the **CP15SDISABLE** signal is asserted HIGH.

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register MAIR1 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[63:32].

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register MAIR1 bits [31:0] are architecturally mapped to AArch32 System register NMRR[31:0].

When EL3 is using AArch32, AArch32 System register MAIR1 bits [31:0] are architecturally mapped to AArch32 System register NMRR[31:0].

When EL3 is using AArch32, AArch32 System register MAIR1 bits [31:0] are architecturally mapped to AArch32 System register NMRR[31:0].

This register is banked between MAIR1 and MAIR1_S and MAIR1_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MAIR1 are UNDEFINED.

Attributes

MAIR1 is a 32-bit register.

This register has the following instances:

- MAIR1, when EL3 is not implemented or FEAT_AA64 is implemented.
- MAIR1_S, when FEAT_AA32EL3 is implemented.
- MAIR1_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '1':

31 – 24	23 – 16	15 – 8	7 – 0
Attr7	Attr6	Attr5	Attr4

Attr<n> , bits [8(n-4)+7:8(n-4)], for n = 7 to 4

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
`0b0000`	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
`0b00`RW, RW not `0b00`	Normal memory, Outer Write-Through Transient.
`0b0100`	Normal memory, Outer Non-cacheable.
`0b01`RW, RW not `0b00`	Normal memory, Outer Write-Back Transient.
`0b10`RW	Normal memory, Outer Write-Through Non-transient.
`0b11`RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is `0b0000`	Meaning when Attr<n>[7:4] is not `0b0000`
`0b0000`	Device-nGnRnE memory	UNPREDICTABLE
`0b00`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
`0b0100`	Device-nGnRE memory	Normal memory, Inner Non-cacheable
`0b01`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
`0b1000`	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=`0b00`)
`0b10`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
`0b1100`	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=`0b00`)
`0b11`RW, RW not `0b00`	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
`0b0`	No Allocate
`0b1`	Allocate

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing MAIR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR1_NS();
        else
            R(t) = NMRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR1();
        else
            R(t) = NMRR();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR1_NS();
        else
            R(t) = NMRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR1();
        else
            R(t) = NMRR();
        end;
    end;
elsif PSTATE.EL == EL3 then
    if TTBCR().EAE == '1' then
        if SCR().NS == '0' then
            R(t) = MAIR1_S();
        else
            R(t) = MAIR1_NS();
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if SCR().NS == '0' then
            R(t) = NMRR_S();
        else
            R(t) = NMRR_NS();
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR1_NS() = R(t);
        else
            NMRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR1() = R(t);
        else
            NMRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR1_NS() = R(t);
        else
            NMRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR1() = R(t);
        else
            NMRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if TTBCR().EAE == '1' then
            if SCR().NS == '0' then
                MAIR1_S() = R(t);
            else
                MAIR1_NS() = R(t);
            end;
        else
            if SCR().NS == '0' then
                NMRR_S() = R(t);
            else
                NMRR_NS() = R(t);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.113 MIDR, Main ID Register

The MIDR characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

Some fields of the MIDR are IMPLEMENTATION DEFINED. For more information about the values of these fields for a particular Armv8 implementation, and any implementation-specific significance of these values, see the product documentation.

AArch32 System register MIDR bits [31:0] are architecturally mapped to AArch64 System register MIDR_EL1[31:0].

AArch32 System register MIDR bits [31:0] are architecturally mapped to External register MIDR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MIDR are UNDEFINED.

Attributes

MIDR is a 32-bit register.

Field descriptions

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
Implementer	Variant	Architecture	PartNum	Revision

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Implementer	Meaning
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        R(t) = VPIDR_EL2()[31:0];
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
        R(t) = VPIDR();
    else
        R(t) = MIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MIDR();
elsif PSTATE.EL == EL3 then
    R(t) = MIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.114 MPIDR, Multiprocessor Affinity Register

The MPIDR characteristics are:

Purpose

In a multiprocessor system, provides an additional PE identification mechanism.

Configuration

In a uniprocessor system, Arm recommends that each Aff<n> field of this register returns a value of 0.

AArch32 System register MPIDR bits [31:0] are architecturally mapped to AArch64 System register MPIDR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MPIDR are UNDEFINED.

Attributes

MPIDR is a 32-bit register.

Field descriptions

31	30	29 25	24	23 16	15 8	7 0
M	U	RES0	MT	Aff2	Aff1	Aff0

M, bit [31]

Indicates whether this implementation includes the functionality introduced by the Armv7 Multiprocessing Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

M	Meaning
0b0	This implementation does not include the Armv7 Multiprocessing Extensions functionality.
0b1	This implementation includes the Armv7 Multiprocessing Extensions functionality.

Access to this field is RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        R(t) = VMPIDR_EL2()[31:0];
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
        R(t) = VMPIDR();
    else
        R(t) = MPIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MPIDR();
elsif PSTATE.EL == EL3 then
    R(t) = MPIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.115 MVBAR, Monitor Vector Base Address Register

The MVBAR characteristics are:

Purpose

When EL3 is implemented and can use AArch32, holds the vector base address for any exception that is taken to Monitor mode.

Secure software must program the MVBAR with the required initial value as part of the PE boot sequence.

Configuration

It is IMPLEMENTATION DEFINED whether MVBAR[0] has a fixed value and ignored writes, or takes the last value written to it.

On a Warm reset into EL3 using AArch32, the reset value of MVBAR is an IMPLEMENTATION DEFINED choice between the following:

- MVBAR[31:5] = an IMPLEMENTATION DEFINED value, which might be UNKNOWN, MVBAR[4:1] = RES0, and MVBAR[0] = 0.
- MVBAR[31:1] = an IMPLEMENTATION DEFINED value that is bits[31:1] of the AArch32 reset address, and MVBAR[0] = 1.

This register is present only when FEAT_AA32EL3 is implemented. Otherwise, direct accesses to MVBAR are UNDEFINED.

Attributes

MVBAR is a 32-bit register.

Field descriptions

When programmed with a vector base address:

31 … 5	4 … 0
Vector Base Address	Reserved

VBA, bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

Reserved, bits [4:0]

Reserved, see Configurations.

Accessing MVBAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsHighestEL(EL1) then
        R(t) = RVBAR();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if IsHighestEL(EL2) then
        R(t) = RVBAR();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = MVBAR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if CP15SDISABLE == HIGH then
        Undefined();
    elsif CP15SDISABLE2 == HIGH then
        Undefined();
    else
        MVBAR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.116 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR1 and MVFR2.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

AArch32 System register MVFR0 bits [31:0] are architecturally mapped to AArch64 System register MVFR0_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MVFR0 are UNDEFINED.

Attributes

MVFR0 is a 32-bit register.

Field descriptions

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
FPRound	FPShVec	FPSqrt	FPDivide	FPTrap	FPDP	FPSP	SIMDReg

FPRound, bits [31:28]

Floating-Point Rounding modes. Indicates whether the floating-point implementation provides support for rounding modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPRound	Meaning
0b0000	Not implemented, or only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.
0b0001	All rounding modes supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

FPShVec, bits [27:24]

Short Vectors. Indicates whether the floating-point implementation provides support for the use of short vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPShVec	Meaning
0b0000	Short vectors not supported.
0b0001	Short vector operation supported.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

FPSqrt, bits [23:20]

Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPSqrt	Meaning
0b0000	Not supported in hardware.
0b0001	Supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

Access to this field is RO.

FPDivide, bits [19:16]

Indicates whether the floating-point implementation provides support for VFP divide operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDivide	Meaning
0b0000	Not supported in hardware.
0b0001	Supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

Access to this field is RO.

FPTrap, bits [15:12]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Floating-Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPTrap	Meaning
0b0000	Not supported.
0b0001	Supported.

All other values are reserved.

A value of 0b0001 indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

Access to this field is RO.

FPDP, bits [11:8]

Floating-Point Double-Precision. Indicates whether the floating-point implementation provides support for double-precision operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDP	Meaning
0b0000	Not supported in hardware.
0b0001	Supported, VFPv2.
0b0010	Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

A value of 0b0001 or 0b0010 indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is 0b0001.
- VDIV.F64 is only available if the Divide field is 0b0001.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

Access to this field is RO.

FPSP, bits [7:4]

Floating-Point Single-Precision. Indicates whether the floating-point implementation provides support for single-precision operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPSP	Meaning
0b0000	Not supported in hardware.
0b0001	Supported, VFPv2.
0b0010	Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

A value of 0b0001 or 0b0010 indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is 0b0001.
- VDIV.F32 is only available if the Divide field is 0b0001.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

Access to this field is RO.

SIMDReg, bits [3:0]

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDReg	Meaning
0b0000	The implementation has no Advanced SIMD and floating-point support.
0b0001	The implementation includes floating-point support with 16 x 64-bit registers.
0b0010	The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

Access to this field is RO.

Accessing MVFR0

Accesses to this register use the following encodings in the System register encoding space:

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

reg
0b0111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
         ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
         ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
         ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = MVFR0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
         ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = MVFR0();
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        R(t) = MVFR0();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.117 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR0 and MVFR2.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

AArch32 System register MVFR1 bits [31:0] are architecturally mapped to AArch64 System register MVFR1_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MVFR1 are UNDEFINED.

Attributes

MVFR1 is a 32-bit register.

Field descriptions

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SIMDFMAC	FPHP	SIMDHP	SIMDSP	SIMDInt	SIMDLS	FPDNaN	FPFtZ

SIMDFMAC, bits [31:28]

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the Advanced SIMD implementation provides fused multiply accumulate instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDFMAC	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.

Access to this field is RO.

FPHP, bits [27:24]

Floating-Point Half-Precision. Indicates the level of half-precision floating-point support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPHP	Meaning
0b0000	Not supported.
0b0001	Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.
0b0010	As for 0b0001, and adds instructions for conversion between double-precision and half-precision.
0b0011	As for 0b0010, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8, the permitted values are:

- 0b0000 in an implementation without floating-point support.
- 0b0010 in an implementation with floating-point support that does not include the FEAT_FP16 extension.
- 0b0011 in an implementation with floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

Half-Precision instructions supported	FPHP	SIMDHP
No support	0b0000	0b0000
Conversions only	0b0010	0b0001
Conversions and arithmetic	0b0011	0b0010

Access to this field is RO.

SIMDHP, bits [23:20]

Advanced SIMD Half-Precision. Indicates the level of half-precision floating-point support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDHP	Meaning
0b0000	Not supported.
0b0001	SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.
0b0010	As for 0b0001, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8, the permitted values are:

- 0b0000 in an implementation without SIMD floating-point support.
- 0b0001 in an implementation with SIMD floating-point support that does not include the FEAT_FP16 extension.
- 0b0010 in an implementation with SIMD floating-point support that includes the FEAT_FP16 extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

Half-Precision instructions supported	FPHP	SIMDHP
No support	0b0000	0b0000
Conversions only	0b0010	0b0001
Conversions and arithmetic	0b0011	0b0010

Access to this field is RO.

SIMDSP, bits [19:16]

Advanced SIMD Single-Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDSP	Meaning
0b0000	Not implemented.
0b0001	Implemented. This value is permitted only if the SIMDInt field is 0b0001.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SIMDInt, bits [15:12]

Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDInt	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SIMDLS, bits [11:8]

Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIMDLS	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

FPDNaN, bits [7:4]

Default NaN mode. Indicates whether the floating-point implementation provides support only for the Default NaN mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDNaN	Meaning
0b0000	Not implemented, or hardware supports only the Default NaN mode.
0b0001	Hardware supports propagation of NaN values.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

FPFtZ, bits [3:0]

Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPFtZ	Meaning
0b0000	Not implemented, or hardware supports only the Flush-to-Zero mode of operation.
0b0001	Hardware supports full denormalized number arithmetic.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Accessing MVFR1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

reg
0b0110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
        ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
        ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
        ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x08);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = MVFR1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
        ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = MVFR1();
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        R(t) = MVFR1();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.118 MVFR2, Media and VFP Feature Register 2

The MVFR2 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR0 and MVFR1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

AArch32 System register MVFR2 bits [31:0] are architecturally mapped to AArch64 System register MVFR2_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to MVFR2 are UNDEFINED.

Attributes

MVFR2 is a 32-bit register.

Field descriptions

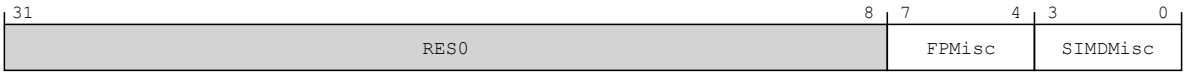

Bits [31:8]

Reserved, RES0.

FPMisc, bits [7:4]

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPMisc	Meaning
0b0000	Not implemented, or no support for miscellaneous features.
0b0001	Support for Floating-point selection.
0b0010	As 0b0001, and Floating-point Conversion to Integer with Directed Rounding modes.
0b0011	As 0b0010, and Floating-point Round to Integer Floating-point.
0b0100	As 0b0011, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0100.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIMDMisc, bits [3:0]

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDMisc	Meaning
0b0000	Not implemented, or no support for miscellaneous features.
0b0001	Floating-point Conversion to Integer with Directed Rounding modes.
0b0010	As 0b0001, and Floating-point Round to Integer Floating-point.
0b0011	As 0b0010, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0011.

Access to this field is RO.

Accessing MVFR2

Accesses to this register use the following encodings in the System register encoding space:

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

reg
0b0101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif (IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' && NSACR().cp10
         ↪== '0') || CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2)
         ↪&& CPTR_EL2().TFP == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CPTR_EL2().FPEN IN {'x0'} then
        AArch64_AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1' &&
         ↪NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID3
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x08);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID3 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x08);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = MVFR2();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TFP == '1' then
        Undefined();
    elsif EL2Enabled() && ((IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().NS == '1'
         ↪&& NSACR().cp10 == '0') || HCPTR().TCP10 == '1') then
        AArch32_TakeHypTrapException(0x00);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TFP ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x07);
        end;
    else
        R(t) = MVFR2();
    end;
elsif PSTATE.EL == EL3 then
    if CPACR().cp10 == '00' then
        Undefined();
    else
        R(t) = MVFR2();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.119 NMRR, Normal Memory Remap Register

The NMRR characteristics are:

Purpose

Provides additional mapping controls for memory regions that are mapped as Normal memory by their entry in the PRRR.

Used in conjunction with the PRRR.

Configuration

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register NMRR bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[63:32].

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register NMRR bits [31:0] are architecturally mapped to AArch32 System register MAIR1[31:0].

When EL3 is using AArch32, AArch32 System register NMRR bits [31:0] are architecturally mapped to AArch32 System register MAIR1[31:0].

When EL3 is using AArch32, AArch32 System register NMRR bits [31:0] are architecturally mapped to AArch32 System register MAIR1[31:0].

This register is banked between NMRR and NMRR_S and NMRR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to NMRR are UNDEFINED.

Attributes

NMRR is a 32-bit register.

This register has the following instances:

- NMRR, when EL3 is not implemented or FEAT_AA64 is implemented.
- NMRR_S, when FEAT_AA32EL3 is implemented.
- NMRR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
OR7	OR6	OR5	OR4	OR3	OR2	OR1	OR0	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0

OR<n> , bits [2n+17:2n+16], for n = 7 to 0

Outer Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OR<n>	Meaning
0b00	Region is Non-cacheable.
0b01	Region is Write-Back, Write-Allocate.
0b10	Region is Write-Through, no Write-Allocate.
0b11	Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

When FEAT_XS is implemented, stage 1 Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IR<n> , bits [2n+1:2n], for n = 7 to 0

Inner Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated.

IR<n>	Meaning
0b00	Region is Non-cacheable.
0b01	Region is Write-Back, Write-Allocate.
0b10	Region is Write-Through, no Write-Allocate.
0b11	Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing NMRR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR1_NS();
        else
            R(t) = NMRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR1();
        else
            R(t) = NMRR();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR1_NS();
        else
            R(t) = NMRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR1();
        else
            R(t) = NMRR();
        end;
    end;
elsif PSTATE.EL == EL3 then
    if TTBCR().EAE == '1' then
        if SCR().NS == '0' then
            R(t) = MAIR1_S();
        else
            R(t) = MAIR1_NS();
        end;
    else
        if SCR().NS == '0' then
            R(t) = NMRR_S();
        else
            R(t) = NMRR_NS();
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR1_NS() = R(t);
        else
            NMRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR1() = R(t);
        else
            NMRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR1_NS() = R(t);
        else
            NMRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR1() = R(t);
        else
            NMRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if TTBCR().EAE == '1' then
            if SCR().NS == '0' then
                MAIR1_S() = R(t);
            else
                MAIR1_NS() = R(t);
            end;
        else
            if SCR().NS == '0' then
                NMRR_S() = R(t);
            else
                NMRR_NS() = R(t);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.120 NSACR, Non-Secure Access Control Register

The NSACR characteristics are:

Purpose

When EL3 is implemented and can use AArch32, defines the Non-secure access permissions to Trace, Advanced SIMD and floating-point functionality. Also includes IMPLEMENTATION DEFINED bits that can define Non-secure access permissions for IMPLEMENTATION DEFINED functionality.

Configuration

Note

In AArch64 state, the NSACR controls are replaced by controls in CPTR_EL3.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to NSACR are UNDEFINED.

Attributes

NSACR is a 32-bit register.

Field descriptions

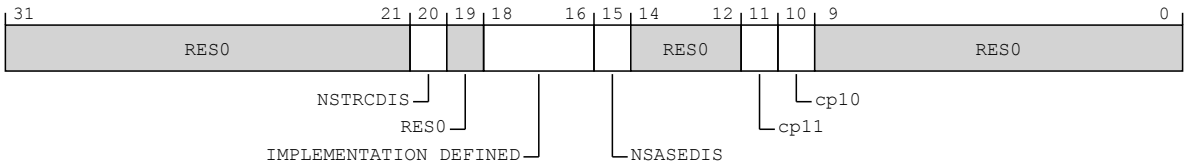

If EL3 is implemented and is using AArch64 then:

- Any read of the NSACR from Non-secure EL2 or Non-secure EL1 returns a value of `0x00000C00`.
- Any read or write to NSACR from Secure EL1 is trapped as an exception to EL3.

If EL3 is not implemented, then any read of the NSACR from EL2 or EL1 returns a value of `0x00000C00`.

Bits [31:21]

Reserved, RES0.

NSTRCDIS, bit [20]

Disables Non-secure System register accesses to all implemented trace System registers.

NSTRCDIS	Meaning
`0b0`	This control has no effect on: • System register access to implemented trace System registers. • The behavior of CPACR.TRCDIS and HCPTR.TTA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSTRCDIS	Meaning
0b1	Non-secure System register accesses to all implemented trace System registers are disabled, meaning: • CPACR.TRCDIS behaves as RAO/WI in Non-secure state, regardless of its actual value. • HCPTR.TTA behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.TRCDIS field:

- If CPACR.TRCDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.TRCDIS is RW, this field is RW.

Note

- FEAT_ETMv4 and FEAT_ETE do not permit EL0 to access the trace System registers. EL0 accesses to the trace System registers are UNDEFINED.
- The Arm architecture does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Bit [19]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [18:16]

IMPLEMENTATION DEFINED.

NSASEDIS, bit [15]

Disables Non-secure access to the Advanced SIMD functionality.

NSASEDIS	Meaning
0b0	This control has no effect on: • Non-secure access to Advanced SIMD functionality. • The behavior of CPACR.ASEDIS and HCPTR.TASE.
0b1	Non-secure access to the Advanced SIMD functionality is disabled, meaning: • CPACR.ASEDIS behaves as RAO/WI in Non-secure state, regardless of its actual value. • HCPTR.TASE behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.ASEDIS field:

- If CPACR.ASEDIS is RES0, this field is RES0. If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.
- If CPACR.ASEDIS is RAZ/WI, this field is RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If CPACR.ASEDIS is RW, this field is RW.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Bits [14:12]

Reserved, RES0.

cp11, bit [11]

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the NSACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

cp10, bit [10]

Enable Non-secure access to the Advanced SIMD and floating-point features. Possible values of the fields are:

cp10	Meaning
`0b0`	Advanced SIMD and floating-point features can be accessed only from Secure state. Any attempt to access this functionality from Non-secure state is UNDEFINED. When the PE is in Non-secure state: • The CPACR.{cp11, cp10} fields ignore writes and read as `0b00`, access denied. • The HCPTR.{TCP11, TCP10} fields behave as RAO/WI, regardless of their actual values.
`0b1`	Advanced SIMD and floating-point features can be accessed from both Security states.

If Non-secure access to the Advanced SIMD and floating-point functionality is enabled, the CPACR must be checked to determine the level of access that is permitted.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing NSACR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    elsif !HaveEL(EL3) || (IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().NS ==
        ↪'1') then
        R(t) = Zeros{20} :: '1100' :: Zeros{8};
    else
        R(t) = NSACR();
    end;
elsif PSTATE.EL == EL2 then
    if !HaveEL(EL3) || (IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().NS ==
        ↪'1') then
        R(t) = Zeros{20} :: '1100' :: Zeros{8};
    else
        R(t) = NSACR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = NSACR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CP15SDISABLE2 == HIGH then
        Undefined();
    else
        NSACR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.121 PAR, Physical Address Register

The PAR characteristics are:

Purpose

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Configuration

PAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits[31:0] and do not modify bits[63:32].

The Configurations section specifies the cases where each PAR format is used.

PAR is accessed as a 32-bit value:

- When the PE is not in Hyp mode and is using the Short-descriptor translation table format.
- When the PE is in Hyp mode and executes an ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, or ATS12NSOUW instruction and the value of HCR.VM is 0 and the value of TTBCR.EAE is 0.

In these cases, PAR[63:32] is RES0.

Otherwise, the PAR is accessed as a 64-bit value, if any of the following is true:

- When using the Long-descriptor translation table format.
- If the stage 1 address translation is disabled and TTBCR.EAE is set to 1.
- In an implementation that includes EL2, for the result of an ATS1Cxx instruction performed from Hyp mode.

For PL1&0 stage 1 translations, TTBCR.EAE selects the translation table format.

AArch32 System register PAR bits [63:0] are architecturally mapped to AArch64 System register PAR_EL1[63:0].

This register is banked between PAR and PAR_S and PAR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to PAR are UNDEFINED.

Attributes

PAR is a 64-bit register.

This register has the following instances:

- PAR, when EL3 is not implemented or FEAT_AA64 is implemented.
- PAR_S, when FEAT_AA32EL3 is implemented.
- PAR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When the instruction returned a 32-bit value to the PAR, PAR.F==0:

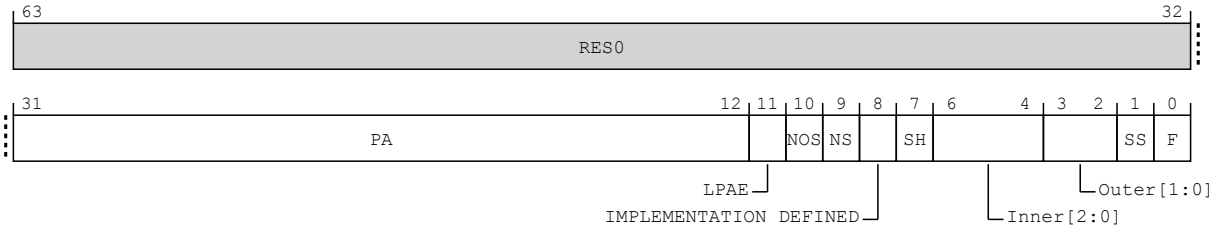

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors. This applies to the NOS, SH, Inner, and Outer fields.
- See the NS bit description for constraints on the value it returns.

Bits [63:32]

Reserved, RES0.

PA, bits [31:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[31:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

LPAE	Meaning
`0b0`	Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NOS, bit [10]

Not Outer Shareable. When the returned value of PAR.SH is 1, indicates the Shareability attribute for the physical memory region:

NOS	Meaning
`0b0`	Memory region is Outer Shareable.
`0b1`	Memory region is Inner Shareable.

When the returned value of PAR.SH is 0 the value returned to this field is UNKNOWN.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bit [8]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bit [7]

Shareability. Indicates whether the physical memory region is Non-shareable:

SH	Meaning
`0b0`	Memory is Non-shareable.
`0b1`	Memory is shareable, and PAR.NOS indicates whether the region is Outer Shareable or Inner Shareable.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Inner[2:0], bits [6:4]

Inner cacheability attribute for the region. Permitted values are:

Inner[2:0]	Meaning
`0b000`	Non-cacheable.
`0b001`	Device-nGnRnE.
`0b011`	Device-nGnRE.
`0b101`	Write-Back, Write-Allocate.
`0b110`	Write-Through.
`0b111`	Write-Back, no Write-Allocate.

The values `0b010` and `0b100` are reserved.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Outer[1:0], bits [3:2]

Outer cacheability attribute for the region. Permitted values are:

Outer[1:0]	Meaning
`0b00`	Non-cacheable.
`0b01`	Write-Back, Write-Allocate.
`0b10`	Write-Through, no Write-Allocate.
`0b11`	Write-Back, no Write-Allocate.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [1]

Supersection. Used to indicate if the result is a Supersection:

SS	Meaning
`0b0`	Result is not a Supersection. PAR[31:12] contains OA[31:12].
`0b1`	Result is a Supersection, and: • PAR[31:24] contains OA[31:24]. • PAR[23:16] contains OA[39:32]. • PAR[15:12] contains `0b0000`. If an implementation supports less than 40 bits of physical address, the bits in the PAR field that correspond to physical address bits that are not implemented are UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
`0b0`	Address translation completed successfully.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When the instruction returned a 32-bit value to the PAR, PAR.F==1:

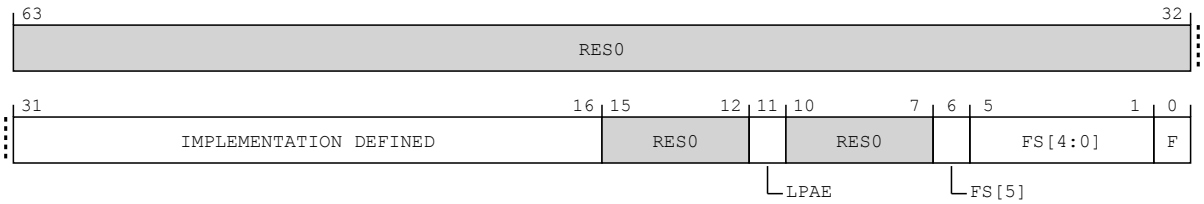

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:16]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:12]

Reserved, RES0.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

LPAE	Meaning
0b0	Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:7]

Reserved, RES0.

FS[5], bit [6]

Fault status bits, External abort type. Provides an IMPLEMENTATION DEFINED classification of an External abort. Values are as in the DFSR.ExT field when using the Short-descriptor translation table format.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FS[4:0], bits [5:1]

Fault status bits. Values are as in the DFSR.FS field when using the Short-descriptor translation table format.

FS[4:0]	Meaning	Applies when
0b00001	Alignment fault.	
0b00011	Access flag fault, level 1.	
0b00100	Fault on instruction cache maintenance.	
0b00101	Translation fault, level 1.	
0b00110	Access flag fault, level 2.	
0b00111	Translation fault, level 2.	
0b01001	Domain fault, level 1.	
0b01011	Domain fault, level 2.	
0b01100	Synchronous External abort, on translation table walk, level 1.	
0b01101	Permission fault, level 1.	
0b01110	Synchronous External abort, on translation table walk, level 2.	
0b01111	Permission fault, level 2.	
0b10000	TLB conflict abort.	
0b11001	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b11100	Synchronous parity or ECC error on translation table walk, level 1.	FEAT_RAS is not implemented
0b11110	Synchronous parity or ECC error on translation table walk, level 2.	FEAT_RAS is not implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When the instruction returned a 64-bit value to the PAR, PAR.F==0:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

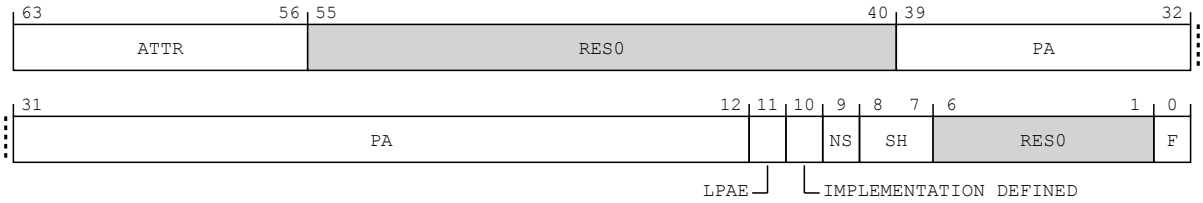

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors. This applies to the ATTR and SH fields.
- See the NS bit description for constraints on the value it returns.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR0 and MAIR1.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [55:40]

Reserved, RES0.

PA, bits [39:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[39:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

LPAE	Meaning
0b1	Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bit [10]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address. Permitted values are:

SH	Meaning
`0b00`	Non-shareable.
`0b10`	Outer Shareable.
`0b11`	Inner Shareable.

The value `0b01` is reserved.

Note

This field returns the value `0b10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:1]

Reserved, RES0.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
`0b0`	Address translation completed successfully.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When the instruction returned a 64-bit value to the PAR, PAR.F==1:

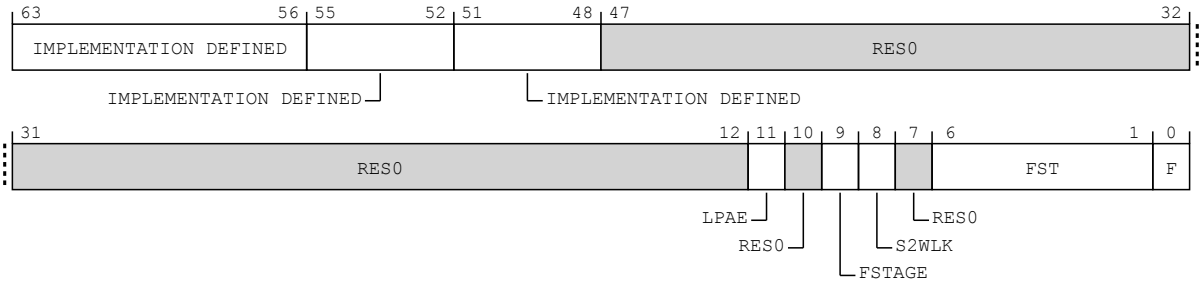

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:12]

Reserved, RES0.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

LPAE	Meaning
0b1	Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [10]

Reserved, RES0.

FSTAGE, bit [9]

Indicates the translation stage at which the translation aborted:

FSTAGE	Meaning
0b0	Translation aborted because of a fault in the stage 1 translation.
0b1	Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S2WLK, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status field. Values are as in the DFSR.STATUS and IFSR.STATUS fields when using the Long-descriptor translation table format.

FST	Meaning	Applies when
0b000000	Address size fault in translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b010101	Synchronous External abort on translation table walk, level 1.	
0b010110	Synchronous External abort on translation table walk, level 2.	
0b010111	Synchronous External abort on translation table walk, level 3.	
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3.	FEAT_RAS is not implemented
0b110000	TLB conflict abort.	

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = PAR_NS()[31:0];
    else
        R(t) = PAR()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = PAR_NS()[31:0];
    else
        R(t) = PAR()[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = PAR_S()[31:0];
    else
        R(t) = PAR_NS()[31:0];
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        PAR_NS()[31:0] = R(t);
    else
        PAR()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        PAR_NS()[31:0] = R(t);
    else
        PAR()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        PAR_S()[31:0] = R(t);
    else
        PAR_NS()[31:0] = R(t);
    end;
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	CRm	opc1
0b1111	0b0111	0b0000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = PAR_NS();
    else
        R(t, t2) = PAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = PAR_NS();
    else
        R(t, t2) = PAR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t, t2) = PAR_S();
    else
        R(t, t2) = PAR_NS();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0111	0b0000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T7 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T7 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        PAR_NS() = R(t, t2);
    else
        PAR() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        PAR_NS() = R(t, t2);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PAR() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        PAR_S() = R(t, t2);
    else
        PAR_NS() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.122 PRRR, Primary Region Remap Register

The PRRR characteristics are:

Purpose

Controls the top-level mapping of the TEX[0], C, and B memory region attributes.

Configuration

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register PRRR bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[31:0].

When EL3 is not implemented or EL3 is using AArch64, AArch32 System register PRRR bits [31:0] are architecturally mapped to AArch32 System register MAIR0[31:0].

When EL3 is using AArch32, AArch32 System register PRRR bits [31:0] are architecturally mapped to AArch32 System register MAIR0[31:0].

When EL3 is using AArch32, AArch32 System register PRRR bits [31:0] are architecturally mapped to AArch32 System register MAIR0[31:0].

This register is banked between PRRR and PRRR_S and PRRR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to PRRR are UNDEFINED.

Attributes

PRRR is a 32-bit register.

This register has the following instances:

- PRRR, when EL3 is not implemented or FEAT_AA64 is implemented.
- PRRR_S, when FEAT_AA32EL3 is implemented.
- PRRR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

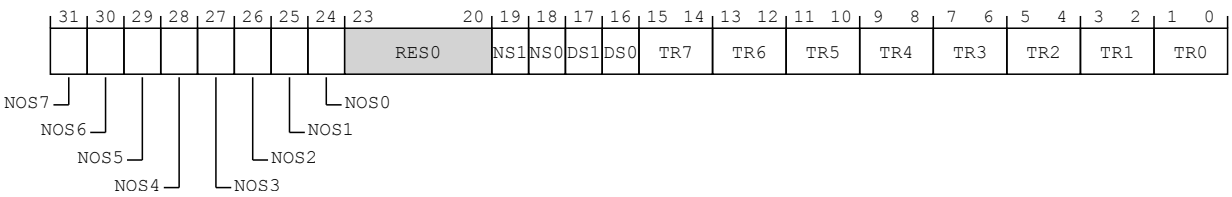

NOS<n> , bits [n+24], for n = 7 to 0

Not Outer Shareable. NOS<n> is the Outer Shareable property for memory attributes n, if the region is mapped as Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, and the appropriate PRRR.{NS0, NS1} field identifies the region as shareable. n is the value of the concatenation of the {TEX[0], C, B} bits from the Translation table descriptor. The possible values of each NOS<n> field other than NOS6 are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NOS<n>	Meaning
0b0	Memory region is Outer Shareable.
0b1	Memory region is Inner Shareable.

The value of this bit is ignored if the region is:

- Device memory
- Normal memory that is at least one of:
  - Inner Non-cacheable, Outer Non-cacheable.
  - Identified by the appropriate PRRR.{NS0, NS1} field as Non-shareable.

The meaning of the NOS6 field is IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:20]

Reserved, RES0.

NS1, bit [19]

Mapping of S = 1 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:

- Is not Inner Non-cacheable, Outer Non-cacheable.
- Has the S bit in the Translation table descriptor set to 1.

NS1	Meaning
0b0	Region is Non-shareable.
0b1	Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS0, bit [18]

Mapping of S = 0 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:

- Is not Inner Non-cacheable, Outer Non-cacheable.
- Has the S bit in the Translation table descriptor set to 0.

NS0	Meaning
0b0	Region is Non-shareable.
0b1	Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DS1, bit [17]

Mapping of S = 1 attribute for Device memory. From Armv8.0, all types of Device memory are Outer Shareable, and therefore this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DS0, bit [16]

Mapping of S = 0 attribute for Device memory. From Armv8.0, all types of Device memory are Outer Shareable, and therefore this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TR<n> , bits [2n+1:2n], for n = 7 to 0

TR<n> is the primary TEX mapping for memory attributes n, and defines the mapped memory type for a region with attributes n. n is the value of the concatenation of the {TEX[0], C, B} bits from the Translation table descriptor. The possible values for each field other than TR6 are:

TR<n>	Meaning
`0b00`	Device-nGnRnE memory
`0b01`	Device-nGnRE memory
`0b10`	Normal memory

The value `0b11` is reserved. The effect of programming a field to `0b11` is CONSTRAINED UNPREDICTABLE.

The meaning of the TR6 field is IMPLEMENTATION DEFINED.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRRR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
`0b1111`	`0b000`	`0b1010`	`0b0010`	`0b000`

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR0_NS();
        else
            R(t) = PRRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR0();
        else
            R(t) = PRRR();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            R(t) = MAIR0_NS();
        else
            R(t) = PRRR_NS();
        end;
    else
        if TTBCR().EAE == '1' then
            R(t) = MAIR0();
        else
            R(t) = PRRR();
        end;
    end;
elsif PSTATE.EL == EL3 then
    if TTBCR().EAE == '1' then
        if SCR().NS == '0' then
            R(t) = MAIR0_S();
        else
            R(t) = MAIR0_NS();
        end;
    else
        if SCR().NS == '0' then
            R(t) = PRRR_S();
        else
            R(t) = PRRR_NS();
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T10 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T10 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR0_NS() = R(t);
        else
            PRRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR0() = R(t);
        else
            PRRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if TTBCR().EAE == '1' then
            MAIR0_NS() = R(t);
        else
            PRRR_NS() = R(t);
        end;
    else
        if TTBCR().EAE == '1' then
            MAIR0() = R(t);
        else
            PRRR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if TTBCR().EAE == '1' then
            if SCR().NS == '0' then
                MAIR0_S() = R(t);
            else
                MAIR0_NS() = R(t);
            end;
        else
            if SCR().NS == '0' then
                PRRR_S() = R(t);
            else
                PRRR_NS() = R(t);
            end;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.123 REVIDR, Revision ID Register

The REVIDR characteristics are:

Purpose

Provides implementation-specific minor revision information.

Configuration

If REVIDR has the same value as MIDR, then its contents have no significance.

AArch32 System register REVIDR bits [31:0] are architecturally mapped to AArch64 System register REVIDR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to REVIDR are UNDEFINED.

Attributes

REVIDR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing REVIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID1
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID1 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = REVIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = REVIDR();
elsif PSTATE.EL == EL3 then
    R(t) = REVIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.124 RMR, Reset Management Register

The RMR characteristics are:

Purpose

If EL1 or EL3 is the highest implemented Exception level and this register is implemented:

- A write to the register at the highest implemented Exception level can request a Warm reset.
- If the highest implemented Exception level can use AArch32 and AArch64, this register specifies the Execution state that the PE boots into on a Warm reset.

Configuration

Only implemented if EL1 or EL3 is the highest implemented Exception level. In this case:

- If the highest implemented Exception level can use AArch32 and AArch64 then this register must be implemented.
- If the highest implemented Exception level cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When the highest implemented Exception level is EL1, AArch32 System register RMR bits [31:0] are architecturally mapped to AArch64 System register RMR_EL1[31:0].

When EL3 is implemented, AArch32 System register RMR bits [31:0] are architecturally mapped to AArch64 System register RMR_EL3[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to RMR are UNDEFINED.

Attributes

RMR is a 32-bit register.

Field descriptions

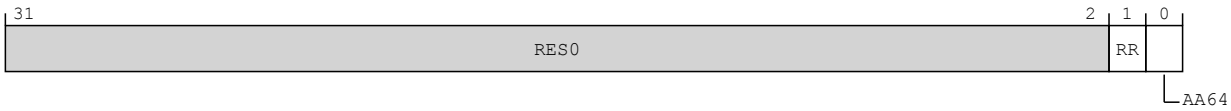

Bits [31:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

AA64, bit [0]

When Implementation can reset into AArch64 state:

Determines which Execution state the PE boots into after a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AA64	Meaning
0b0	AArch32.
0b1	AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

Accessing RMR

When EL3 is implemented, Arm deprecates accessing this register from any PE mode other than Monitor mode.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL IN {EL1, EL3} && IsHighestEL(PSTATE.EL) then
    R(t) = RMR();
else
    Undefined();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsHighestEL(EL1) then
        RMR() = R(t);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if CP15SDISABLE == HIGH then
        Undefined();
    elsif CP15SDISABLE2 == HIGH then
        Undefined();
    else
        RMR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.125 RVBAR, Reset Vector Base Address Register

The RVBAR characteristics are:

Purpose

If EL3 is not implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch32 state.

Configuration

This register is implemented only if the highest Exception level implemented is capable of using AArch32, and is not EL3.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to RVBAR are UNDEFINED.

Attributes

RVBAR is a 32-bit register.

Field descriptions

ResetAddress, bits [31:1]

Bits [31:1] of the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 32-bit state.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [0]

Reserved, RES1.

Accessing RVBAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if IsHighestEL(EL1) then
        R(t) = RVBAR();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if IsHighestEL(EL2) then
        R(t) = RVBAR();
    else
        Undefined();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = MVBAR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.126 SCR, Secure Configuration Register

The SCR characteristics are:

Purpose

When EL3 is implemented and can use AArch32, defines the configuration of the current Security state. It specifies:

- The Security state, either Secure or Non-secure.
- What mode the PE branches to if an IRQ, FIQ, or External abort occurs.
- Whether the PSTATE.F or PSTATE.A bits can be modified when SCR.NS==1.

Configuration

This register is present only when FEAT_AA32EL3 is implemented. Otherwise, direct accesses to SCR are UNDEFINED.

Attributes

SCR is a 32-bit register.

Field descriptions

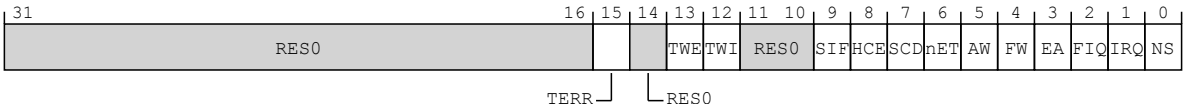

Bits [31:16]

Reserved, RES0.

TERR, bit [15]

When FEAT_RAS is implemented:

Trap Error record accesses. Generate a Monitor Trap exception on MRC, MCR, MRRC, or MCRR accesses to the following registers from modes other than Monitor mode, reported using EC syndrome values `0x03` and `0x04`:

ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS. When FEAT_RASv1p1 is implemented, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7.

TERR	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Accesses to the specified registers from modes other than Monitor mode generate a Monitor Trap exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [14]

Reserved, RES0.

TWE, bit [13]

Traps WFE instructions to Monitor mode.

TWE	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFE instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2 by SCTLR.nTWE or HCR.TWE. Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

TWI, bit [12]

Traps WFI instructions to Monitor mode.

TWI	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFI instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2 by SCTLR.nTWI or HCR.TWI. Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Bits [11:10]

Reserved, RES0.

SIF, bit [9]

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction execution from Non-secure memory.

SIF	Meaning
`0b0`	Secure state instruction execution from Non-secure memory is permitted.
`0b1`	Secure state instruction execution from Non-secure memory is not permitted.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

HCE, bit [8]

Hypervisor Call instruction enable. If EL2 is implemented, enables execution of HVC instructions at Non-secure EL1 and EL2.

HCE	Meaning

`0b0`	HVC instructions are:
	• UNDEFINED at Non-secure EL1. The Undefined Instruction exception is taken from PL1 to PL1.
	• UNPREDICTABLE at EL2. Behavior is one of the following:
	  • The instruction is UNDEFINED.
	  • The instruction executes as a NOP.

`0b1`	HVC instructions are enabled at Non-secure EL1 and EL2.

Note

HVC instructions are always UNDEFINED at EL0 and in Secure state.

If EL2 is not implemented, this bit is RES0 and HVC is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

SCD, bit [7]

Secure Monitor Call disable. Disables SMC instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCD	Meaning
0b0	SMC instructions are enabled.
0b1	In Non-secure state, SMC instructions are UNDEFINED. The Undefined Instruction exception is taken from the current Exception level to the current Exception level. In Secure state, behavior is one of the following: • The instruction is UNDEFINED. • The instruction executes as a NOP.

Note

SMC instructions are always UNDEFINED at PL0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

nET, bit [6]

Not Early Termination. This bit disables early termination.

nET	Meaning
0b0	Early termination permitted. Execution time of data operations can depend on the data values.
0b1	Disable early termination. The number of cycles required for data operations is forced to be independent of the data values.

This IMPLEMENTATION DEFINED mechanism can disable data dependent timing optimizations from multiplies and data operations. It can provide system support against information leakage that might be exploited by timing correlation types of attack.

On implementations that do not support early termination or do not support disabling early termination, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

AW, bit [5]

When the value of SCR.EA is 1 and the value of HCR.AMO is 0, this bit controls whether PSTATE.A masks an External abort taken from Non-secure state.

AW	Meaning
0b0	External aborts taken from Non-secure state are not masked by PSTATE.A, and are taken to EL3. External aborts taken from Secure state are masked by PSTATE.A.
0b1	External aborts taken from either Security state are masked by PSTATE.A. When PSTATE.A is 0, the abort is taken to EL3.

When SCR.EA is 0 or HCR.AMO is 1, this bit has no effect.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

FW, bit [4]

When the value of SCR.FIQ is 1 and the value of HCR.FMO is 0, this bit controls whether PSTATE.F masks an FIQ interrupt taken from Non-secure state.

FW	Meaning
`0b0`	An FIQ taken from Non-secure state is not masked by PSTATE.F, and is taken to EL3. An FIQ taken from Secure state is masked by PSTATE.F.
`0b1`	An FIQ taken from either Security state is masked by PSTATE.F. When PSTATE.F is 0, the FIQ is taken to EL3.

When SCR.FIQ is 0 or HCR.FMO is 1, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

EA, bit [3]

External Abort handler. This bit controls which mode takes External aborts and SError exceptions.

EA	Meaning
`0b0`	External aborts taken to Abort mode.
`0b1`	External aborts taken to Monitor mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

FIQ, bit [2]

FIQ handler. This bit controls which mode takes FIQ exceptions.

FIQ	Meaning
`0b0`	FIQs taken to FIQ mode.
`0b1`	FIQs taken to Monitor mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRQ, bit [1]

IRQ handler. This bit controls which mode takes IRQ exceptions.

IRQ	Meaning
0b0	IRQs taken to IRQ mode.
0b1	IRQs taken to Monitor mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

NS, bit [0]

Non-secure bit. Except when the PE is in Monitor mode, this bit determines the Security state of the PE:

NS	Meaning
0b0	PE is in Secure state.
0b1	PE is in Non-secure state.

If the HCR.TGE bit is set, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing the SCR.NS bit from 0 to 1 results in the SCR.NS bit remaining as 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Accessing SCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    R(t) = SCR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    SCR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.127 SCTLR, System Control Register

The SCTLR characteristics are:

Purpose

Provides the top-level control of the system, including its memory system.

Configuration

Some bits in the register are read-only. These bits relate to non-configurable features of an implementation, and are provided for compatibility with previous versions of the architecture.

AArch32 System register SCTLR bits [31:0] are architecturally mapped to AArch64 System register SCTLR_EL1[31:0].

This register is banked between SCTLR and SCTLR_S and SCTLR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to SCTLR are UNDEFINED.

Attributes

SCTLR is a 32-bit register.

This register has the following instances:

- SCTLR, when EL3 is not implemented or FEAT_AA64 is implemented.
- SCTLR_S, when FEAT_AA32EL3 is implemented.
- SCTLR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

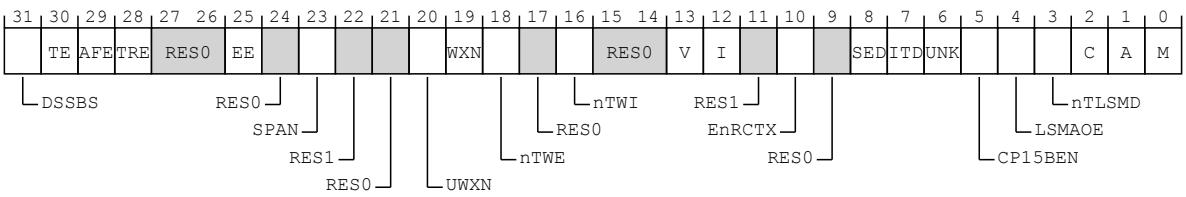

DSSBS, bit [31]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to any mode in this security state except Hyp mode
0b1	PSTATE.SSBS is set to 1 on an exception to any mode in this security state except Hyp mode

Note

When EL3 is implemented and is using AArch32, this bit is banked between the two Security states.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to an Exception level that is executing at PL1 are taken to A32 or T32 state:

TE	Meaning
`0b0`	Exceptions, including reset, taken to A32 state.
`0b1`	Exceptions, including reset, taken to T32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

AFE, bit [29]

Access Flag Enable. When using the Short-descriptor translation table format for the PL1&0 translation regime, this bit enables use of the AP[0] bit in the translation descriptors as the Access flag, and restricts access permissions in the translation descriptors to the simplified model.

AFE	Meaning
`0b0`	In the Translation table descriptors, AP[0] is an access permissions bit. The full range of access permissions is supported. No Access flag is implemented.
`0b1`	In the Translation table descriptors, AP[0] is the Access flag. Only the simplified model for access permissions is supported.

When using the Long-descriptor translation table format, the VMSA behaves as if this bit is set to 1, regardless of the value of this bit.

The AFE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

TRE, bit [28]

TEX remap enable. This bit enables remapping of the TEX[2:1] bits in the PL1&0 translation regime for use as two translation table bits that can be managed by the operating system. Enabling this remapping also changes the scheme used to describe the memory region attributes in the VMSA.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRE	Meaning
0b0	TEX remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory region attributes.
0b1	TEX remap enabled. TEX[2:1] are reassigned for use as bits managed by the operating system. The TEX[0], C, and B bits are used to describe the memory region attributes, with the MMU remap registers.

When the value of TTBCR.EAE is 1, this bit is RES1.

The TRE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

The value of the PSTATE.E bit on branch to an exception vector or coming out of reset, and the endianness of stage 1 translation table walks in the PL1&0 translation regime.

EE	Meaning
0b0	Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are little-endian.
0b1	Big-endian. PSTATE.E is set to 1 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support for data accesses at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support for data accesses at Exception levels higher than EL0, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

SPAN, bit [23]

When FEAT_PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SPAN	Meaning
0b0	PSTATE.PAN is set to 1 in the following situations: • In Non-secure state, on taking an exception to EL1. • In Secure state, when EL3 is using AArch64, on taking an exception to EL1. • In Secure state, when EL3 is using AArch32, on taking an exception to EL3.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bit [22]

Reserved, RES1.

Bit [21]

Reserved, RES0.

UWXN, bit [20]

Unprivileged write permission implies PL1 XN (Execute-never). This bit can force all memory regions that are writable at PL0 to be treated as XN for accesses from software executing at PL1.

UWXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable at PL0 forced to XN for accesses from software executing at PL1.

The UWXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

WXN, bit [19]

Write permission implies XN (Execute-never). For the PL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable in the PL1&0 translation regime is forced to XN for accesses from software executing at PL1 or PL0.

This bit applies only when SCTLR.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to Undefined mode.

nTWE	Meaning
`0b0`	Any attempt to execute a WFE instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
`0b1`	This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to Undefined mode.

nTWI	Meaning
`0b0`	Any attempt to execute a WFI instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
`0b1`	This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Bits [15:14]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

V, bit [13]

Vectors bit. This bit selects the base address of the exception vectors for exceptions taken to a PE mode other than Monitor mode or Hyp mode:

V	Meaning
`0b0`	Normal exception vectors. Base address is held in VBAR.
`0b1`	High exception vectors (Hivecs), base address `0xFFFF0000`. This base address cannot be remapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

I, bit [12]

Instruction access Cacheability control, for accesses at EL1 and EL0:

I	Meaning
`0b0`	All instruction access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of instruction and unified cache. If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
`0b1`	All instruction access to Normal memory from PL1 and PL0 can be cached at all levels of instruction and unified cache. If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

Instruction accesses to Normal memory from EL1 and EL0 are Cacheable regardless of the value of the SCTLR.I bit if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bit [11]

Reserved, RES1.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Enable EL0 access to the following System instructions:

- CFPRCTX.
- DVPRCTX.
- CPPRCTX.
- If FEAT_SPECRES2 is implemented, COSPRCTX.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnRCTX	Meaning
0b0	EL0 access to these instructions is disabled, and these instructions are trapped to EL1.
0b1	EL0 access to these instructions is enabled.

Note

When EL3 is implemented and is using AArch32, this bit is banked between the two Security states.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at PL0 and PL1.

SED	Meaning
0b0	SETEND instruction execution is enabled at PL0 and PL1.
0b1	SETEND instructions are UNDEFINED at PL0 and PL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at PL1 and PL0.

ITD	Meaning
0b0	All IT instruction functionality is enabled at PL1 and PL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITD	Meaning

`0b1`	Any attempt at PL1 or PL0 to execute any of the following is UNDEFINED:
	• All encodings of the IT instruction with hw1[3:0]!=1000.
	• All encodings of the subsequent instruction with the following values for hw1:
	  • `0b11xxxxxxxxxxxxxx`: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
	  • `0b1011xxxxxxxxxxxx`: All instructions in 'Miscellaneous 16-bit instructions'.
	  • `0b10100xxxxxxxxxxx`: ADD Rd, PC, #imm
	  • `0b01001xxxxxxxxxxx`: LDR Rd, [PC, #imm]
	  • `0b0100x1xxx1111xxx`: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
	  • `0b010001xx1xxxx111`: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.
	These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.
	It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:
	• A 16-bit instruction, that can only be followed by another 16-bit instruction.
	• The first half of a 32-bit instruction.
	This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.
	An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1, then behavior is CONSTRAINED UNPREDICTABLE. For more information see 'Changes to an ITD control by an instruction in an IT block'.

ITD is optional, but if it is implemented in the SCTLR, then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and HSCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

When an implementation does not implement ITD, access to this field is RAZ/WI.

UNK, bit [6]

Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==`0b1111`) encoding space from PL1 and PL0:

CP15BEN	Meaning
`0b0`	PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CP15BEN	Meaning
0b1	PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR, then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and HSCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

When an implementation does not implement CP15BEN, access to this field is RAO/WI.

LSMAOE, bit [4]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
0b0	For all memory accesses at EL1 or EL0, T32 and A32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
0b1	The ordering and interrupt behavior of T32 and A32 Load Multiple and Store Multiple at EL1 or EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Otherwise:

Reserved, RES1.

nTLSMD, bit [3]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
0b0	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL1 or EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
0b1	All memory accesses by T32 and A32 Load Multiple and Store Multiple at EL1 or EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL1 and EL0:

C	Meaning
0b0	All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache.
0b1	All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, can be cached at all levels of data and unified cache.

The PE ignores SCTLR.C, and data accesses to Normal memory from EL1 and EL0 are Cacheable, if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at PL1 and PL0:

A	Meaning
0b0	Alignment fault checking disabled when executing at PL1 or PL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
0b1	Alignment fault checking enabled when executing at PL1 or PL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

M, bit [0]

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Meaning
0b0	EL1 and EL0 stage 1 address translation disabled. See the SCTLR.I field for the behavior of instruction accesses to Normal memory.
0b1	EL1 and EL0 stage 1 address translation enabled.

The PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if either:

- EL2 is using AArch32 and the value of HCR.{DC, TGE} is not {0, 0}.
- EL2 is using AArch64 and the value of HCR_EL2.{DC, TGE} is not {0, 0}.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing SCTLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = SCTLR_NS();
    else
        R(t) = SCTLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = SCTLR_NS();
    else
        R(t) = SCTLR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = SCTLR_S();
    else
        R(t) = SCTLR_NS();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        SCTLR_NS() = R(t);
    else
        SCTLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        SCTLR_NS() = R(t);
    else
        SCTLR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            SCTLR_S() = R(t);
        else
            SCTLR_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.128 SPSR, Saved Program Status Register

The SPSR characteristics are:

Purpose

Holds the saved process state for the current mode.

Configuration

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to SPSR are UNDEFINED.

Attributes

SPSR is a 32-bit register.

Field descriptions

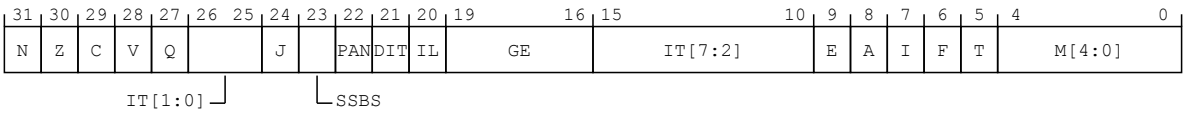

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to the current mode, and copied to PSTATE.N on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to the current mode, and copied to PSTATE.Z on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to the current mode, and copied to PSTATE.C on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to the current mode, and copied to PSTATE.V on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to the current mode, and copied to PSTATE.Q on executing an exception return operation in the current mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to the current mode, and copied to PSTATE.IT on executing an exception return operation in the current mode.

On executing an exception return operation in the current mode, SPSR.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to the current mode, and copied to PSTATE.SSBS on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to the current mode, and copied to PSTATE.PAN on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to the current mode, and copied to PSTATE.DIT on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to the current mode, and copied to PSTATE.IL on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to the current mode, and copied to PSTATE.GE on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to the current mode, and copied to PSTATE.E on executing an exception return operation in the current mode.

If the implementation does not support big-endian operation, SPSR.E is RES0. If the implementation does not support little-endian operation, SPSR.E is RES1. On executing an exception return operation in the current mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to the current mode, and copied to PSTATE.A on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to the current mode, and copied to PSTATE.I on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to the current mode, and copied to PSTATE.F on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to the current mode, and copied to PSTATE.T on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to the current mode, and copied to PSTATE.M[4:0] on executing an exception return operation in the current mode.

M[4:0]	Meaning	Applies when
`0b10000`	User.	
`0b10001`	FIQ.	
`0b10010`	IRQ.	
`0b10011`	Supervisor.	
`0b10110`	Monitor.	FEAT_EL3 is implemented
`0b10111`	Abort.	
`0b11010`	Hyp.	FEAT_EL3 is implemented
`0b11011`	Undefined.	
`0b11111`	System.	

Other values are reserved. If SPSR.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in the current mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR

SPSR can be read using the MRS instruction and written using the MSR (register) or MSR (immediate) instructions.

The MSR (register) and MRS instructions used to access SPSR are data-independent-time instructions as described in About PSTATE.DIT.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.129 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configuration

AArch32 System register SPSR_abt bits [31:0] are architecturally mapped to AArch64 System register SPSR_abt[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to SPSR_abt are UNDEFINED.

Attributes

SPSR_abt is a 32-bit register.

Field descriptions

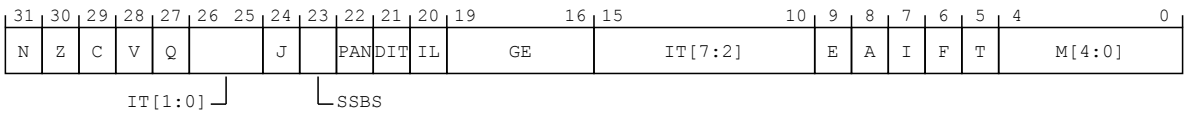

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Abort mode, and copied to PSTATE.IT on executing an exception return operation in Abort mode.

On executing an exception return operation in Abort mode, SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and copied to PSTATE.SSBS on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and copied to PSTATE.PAN on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_abt

SPSR_abt is accessible in all modes other than User mode and Abort mode.

The MSR (register) and MRS instructions used to access SPSR_abt are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_abt
```

R	M	M1
0b1	0b1	0b0100

```
MSR{<c>}{<q>} SPSR_abt, <Rn>
```

R	M	M1
0b1	0b1	0b0100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.130 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configuration

AArch32 System register SPSR_fiq bits [31:0] are architecturally mapped to AArch64 System register SPSR_fiq[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to SPSR_fiq are UNDEFINED.

Attributes

SPSR_fiq is a 32-bit register.

Field descriptions

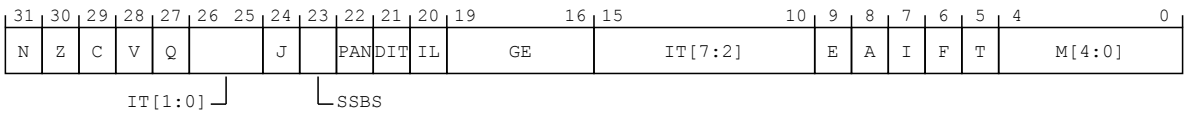

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to FIQ mode, and copied to PSTATE.IT on executing an exception return operation in FIQ mode.

On executing an exception return operation in FIQ mode, SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_fiq

SPSR_fiq is accessible in all modes other than User mode and FIQ mode.

The MSR (register) and MRS instructions used to access SPSR_fiq are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_fiq
```

R	M	M1
0b1	0b0	0b1110

```
MSR{<c>}{<q>} SPSR_fiq, <Rn>
```

R	M	M1
0b1	0b0	0b1110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.131 SPSR_hyp, Saved Program Status Register (Hyp mode)

The SPSR_hyp characteristics are:

Purpose

Holds the saved process state when an exception is taken to Hyp mode.

Configuration

AArch32 System register SPSR_hyp bits [31:0] are architecturally mapped to AArch64 System register SPSR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to SPSR_hyp are UNDEFINED.

Attributes

SPSR_hyp is a 32-bit register.

Field descriptions

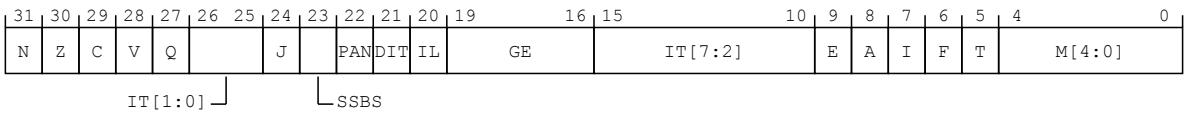

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Hyp mode, and copied to PSTATE.N on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Hyp mode, and copied to PSTATE.Z on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Hyp mode, and copied to PSTATE.C on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Hyp mode, and copied to PSTATE.V on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Hyp mode, and copied to PSTATE.Q on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Hyp mode, and copied to PSTATE.IT on executing an exception return operation in Hyp mode.

On executing an exception return operation in Hyp mode, SPSR_hyp.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Hyp mode, and copied to PSTATE.SSBS on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Hyp mode, and copied to PSTATE.PAN on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Hyp mode, and copied to PSTATE.DIT on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Hyp mode, and copied to PSTATE.IL on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Hyp mode, and copied to PSTATE.GE on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Hyp mode, and copied to PSTATE.E on executing an exception return operation in Hyp mode.

If the implementation does not support big-endian operation, SPSR_hyp.E is RES0. If the implementation does not support little-endian operation, SPSR_hyp.E is RES1. On executing an exception return operation in Hyp mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_hyp.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_hyp.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Hyp mode, and copied to PSTATE.A on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Hyp mode, and copied to PSTATE.I on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Hyp mode, and copied to PSTATE.F on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Hyp mode, and copied to PSTATE.T on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Hyp mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Hyp mode.

M[4:0]	Meaning	Applies when
0b10000	User.	
0b10001	FIQ.	
0b10010	IRQ.	
0b10011	Supervisor.	
0b10111	Abort.	
0b11010	Hyp.	FEAT_EL3 is implemented
0b11011	Undefined.	
0b11111	System.	

Other values are reserved. If SPSR_hyp.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Hyp mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_hyp

SPSR_hyp is accessible only in Monitor mode.

The MSR (register) and MRS instructions used to access SPSR_hyp are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_hyp
```

R	M	M1
0b1	0b1	0b1110

```
MSR{<c>}{<q>} SPSR_hyp, <Rn>
```

R	M	M1
0b1	0b1	0b1110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.132 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configuration

AArch32 System register SPSR_irq bits [31:0] are architecturally mapped to AArch64 System register SPSR_irq[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to SPSR_irq are UNDEFINED.

Attributes

SPSR_irq is a 32-bit register.

Field descriptions

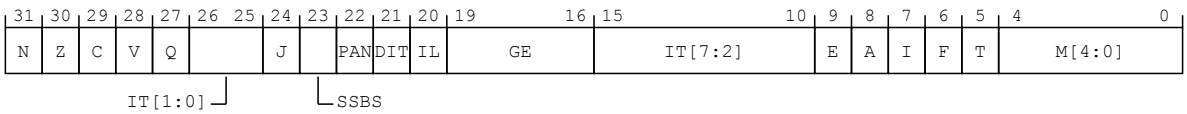

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to IRQ mode, and copied to PSTATE.IT on executing an exception return operation in IRQ mode.

On executing an exception return operation in IRQ mode, SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_irq

SPSR_irq is accessible in all modes other than User mode and IRQ mode.

The MSR (register) and MRS instructions used to access SPSR_irq are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_irq
```

R	M	M1
0b1	0b1	0b0000

```
MSR{<c>}{<q>} SPSR_irq, <Rn>
```

R	M	M1
0b1	0b1	0b0000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.133 SPSR_mon, Saved Program Status Register (Monitor mode)

The SPSR_mon characteristics are:

Purpose

Holds the saved process state when an exception is taken to Monitor mode.

Configuration

This register is present only when FEAT_AA32EL3 is implemented. Otherwise, direct accesses to SPSR_mon are UNDEFINED.

Attributes

SPSR_mon is a 32-bit register.

Field descriptions

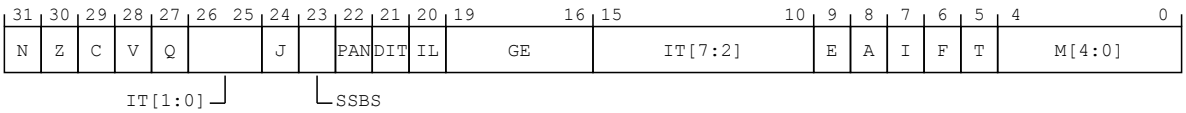

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Monitor mode, and copied to PSTATE.N on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Monitor mode, and copied to PSTATE.Z on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Monitor mode, and copied to PSTATE.C on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Monitor mode, and copied to PSTATE.V on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Monitor mode, and copied to PSTATE.Q on executing an exception return operation in Monitor mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Monitor mode, and copied to PSTATE.IT on executing an exception return operation in Monitor mode.

On executing an exception return operation in Monitor mode, SPSR_mon.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Monitor mode, and copied to PSTATE.SSBS on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Monitor mode, and copied to PSTATE.PAN on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Monitor mode, and copied to PSTATE.DIT on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Monitor mode, and copied to PSTATE.IL on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Monitor mode, and copied to PSTATE.GE on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Monitor mode, and copied to PSTATE.E on executing an exception return operation in Monitor mode.

If the implementation does not support big-endian operation, SPSR_mon.E is RES0. If the implementation does not support little-endian operation, SPSR_mon.E is RES1. On executing an exception return operation in Monitor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_mon.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_mon.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Monitor mode, and copied to PSTATE.A on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Monitor mode, and copied to PSTATE.I on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Monitor mode, and copied to PSTATE.F on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Monitor mode, and copied to PSTATE.T on executing an exception return operation in Monitor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Monitor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Monitor mode.

M[4:0]	Meaning	Applies when
0b10000	User.	
0b10001	FIQ.	
0b10010	IRQ.	
0b10011	Supervisor.	
0b10110	Monitor.	FEAT_EL3 is implemented
0b10111	Abort.	
0b11010	Hyp.	FEAT_EL3 is implemented
0b11011	Undefined.	
0b11111	System.	

Other values are reserved. If SPSR_mon.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Monitor mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_mon

SPSR_mon is only accessible in EL3 modes other than Monitor mode.

The MSR (register) and MRS instructions used to access SPSR_mon are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_mon
```

R	M	M1
0b1	0b1	0b1100

```
MSR{<c>}{<q>} SPSR_mon, <Rn>
```

R	M	M1
0b1	0b1	0b1100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.134 SPSR_svc, Saved Program Status Register (Supervisor mode)

The SPSR_svc characteristics are:

Purpose

Holds the saved process state when an exception is taken to Supervisor mode.

Configuration

AArch32 System register SPSR_svc bits [31:0] are architecturally mapped to AArch64 System register SPSR_EL1[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to SPSR_svc are UNDEFINED.

Attributes

SPSR_svc is a 32-bit register.

Field descriptions

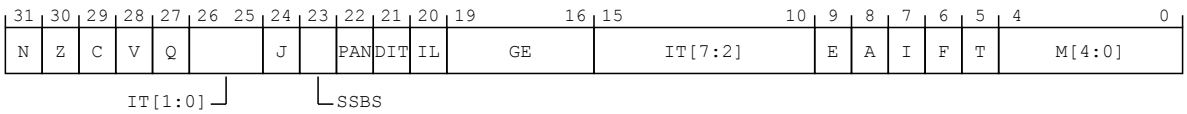

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Supervisor mode, and copied to PSTATE.N on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Supervisor mode, and copied to PSTATE.Z on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Supervisor mode, and copied to PSTATE.C on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Supervisor mode, and copied to PSTATE.V on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Supervisor mode, and copied to PSTATE.Q on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Supervisor mode, and copied to PSTATE.IT on executing an exception return operation in Supervisor mode.

On executing an exception return operation in Supervisor mode, SPSR_svc.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Supervisor mode, and copied to PSTATE.SSBS on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Supervisor mode, and copied to PSTATE.PAN on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Supervisor mode, and copied to PSTATE.DIT on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Supervisor mode, and copied to PSTATE.IL on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Supervisor mode, and copied to PSTATE.GE on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Supervisor mode, and copied to PSTATE.E on executing an exception return operation in Supervisor mode.

If the implementation does not support big-endian operation, SPSR_svc.E is RES0. If the implementation does not support little-endian operation, SPSR_svc.E is RES1. On executing an exception return operation in Supervisor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_svc.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_svc.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Supervisor mode, and copied to PSTATE.A on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Supervisor mode, and copied to PSTATE.I on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Supervisor mode, and copied to PSTATE.F on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Supervisor mode, and copied to PSTATE.T on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Supervisor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Supervisor mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_svc.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Supervisor mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_svc

SPSR_svc is accessible in all modes other than User mode and Supervisor mode.

The MSR (register) and MRS instructions used to access SPSR_svc are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_svc
```

R	M	M1
0b1	0b1	0b0010

```
MSR{<c>}{<q>} SPSR_svc, <Rn>
```

R	M	M1
0b1	0b1	0b0010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.135 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configuration

AArch32 System register SPSR_und bits [31:0] are architecturally mapped to AArch64 System register SPSR_und[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to SPSR_und are UNDEFINED.

Attributes

SPSR_und is a 32-bit register.

Field descriptions

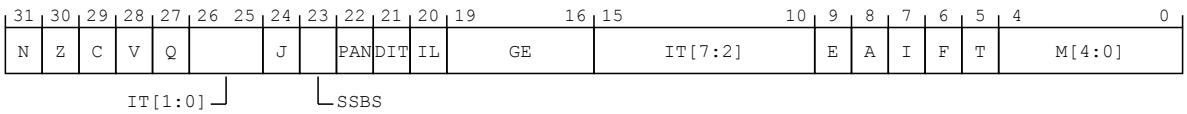

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Undefined mode, and copied to PSTATE.IT on executing an exception return operation in Undefined mode.

On executing an exception return operation in Undefined mode, SPSR_und.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in 'Illegal return events from AArch32 state'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_und

SPSR_und is accessible in all modes other than User mode and Undefined mode.

The MSR (register) and MRS instructions used to access SPSR_und are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS{<c>}{<q>} <Rd>, SPSR_und
```

R	M	M1
0b1	0b1	0b0110

```
MSR{<c>}{<q>} SPSR_und, <Rn>
```

R	M	M1
0b1	0b1	0b0110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.136 TCMTR, TCM Type Register

The TCMTR characteristics are:

Purpose

Provides information about the implementation of the TCM.

Configuration

If EL1 or above can use AArch32 then this register must be implemented.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TCMTR are UNDEFINED.

Attributes

TCMTR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing TCMTR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID1
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TCMTR();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL2 then
    R(t) = TCMTR();
elsif PSTATE.EL == EL3 then
    R(t) = TCMTR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.137 TLBIALL, TLB Invalidate All

The TLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIALL are UNDEFINED.

Attributes

TLBIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, Broadcast_NSH, TLBI_ExcludeXS);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.138 TLBIALLH, TLB Invalidate All, Hyp mode

The TLBIALLH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIALLH are UNDEFINED.

Attributes

TLBIALLH is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch32_TLBI_ALL(SS_NonSecure, Regime_EL2, Broadcast_NSH, TLBI_AllAttr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.139 TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable

The TLBIALLHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIALLHIS are UNDEFINED.

Attributes

TLBIALLHIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        AArch32_TLBI_ALL(SS_NonSecure, Regime_EL2, Broadcast_ISH, TLBI_AllAttr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.140 TLBIALLIS, TLB Invalidate All, Inner Shareable

The TLBIALLIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIALLIS are UNDEFINED.

Attributes

TLBIALLIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLBIS
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TTLBIS ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, Broadcast_ISH, TLBI_ExcludeXS);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.141 TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp

The TLBIALLNSNH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIALLNSNH are UNDEFINED.

Attributes

TLBIALLNSNH is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLNSNH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    else
        AArch32_TLBI_ALL(SS_NonSecure, Regime_EL10, Broadcast_NSH, TLBI_AllAttr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.142 TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable

The TLBIALLNSNHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIALLNSNHIS are UNDEFINED.

Attributes

TLBIALLNSNHIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLNSNHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !HaveEL(EL2) then
        Undefined();
    else
        AArch32_TLBI_ALL(SS_NonSecure, Regime_EL10, Broadcast_ISH, TLBI_AllAttr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.143 TLBIASID, TLB Invalidate by ASID match

The TLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIASID are UNDEFINED.

Attributes

TLBIASID is a 32-bit System instruction.

Field descriptions

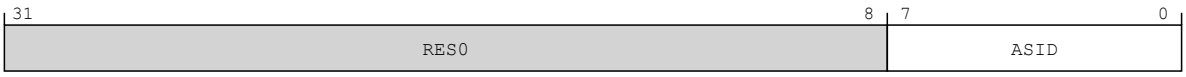

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing TLBIASID

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
            ↪R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.144 TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable

The TLBIASIDIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIASIDIS are UNDEFINED.

Attributes

TLBIASIDIS is a 32-bit System instruction.

Field descriptions

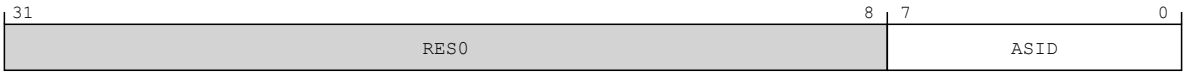

Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing TLBIASIDIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLBIS
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TTLBIS ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
            ↪R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_ISH, TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.145 TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2

The TLBIIPAS2 characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIIPAS2 are UNDEFINED.

Attributes

TLBIIPAS2 is a 32-bit System instruction.

Field descriptions

31 – 28	27 – 0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    elsif SCR().NS == '0' then
        return;
    else
        AArch32_TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.146 TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable

The TLBIIPAS2IS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIIPAS2IS are UNDEFINED.

Attributes

TLBIIPAS2IS is a 32-bit System instruction.

Field descriptions

31:28	27:0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2IS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    elsif SCR().NS == '0' then
        return;
    else
        AArch32_TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.147 TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level

The TLBIIPAS2L characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIIPAS2L are UNDEFINED.

Attributes

TLBIIPAS2L is a 32-bit System instruction.

Field descriptions

31 – 28	27 – 0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2L

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    elsif SCR().NS == '0' then
        return;
    else
        AArch32_TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.148 TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable

The TLBIIPAS2LIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIIPAS2LIS are UNDEFINED.

Attributes

TLBIIPAS2LIS is a 32-bit System instruction.

Field descriptions

31:28	27:0
RES0	IPA[39:12]

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2LIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    elsif SCR().NS == '0' then
        return;
    else
        AArch32_TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.149 TLBIMVA, TLB Invalidate by VA

The TLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVA are UNDEFINED.

Attributes

TLBIMVA is a 32-bit System instruction.

Field descriptions

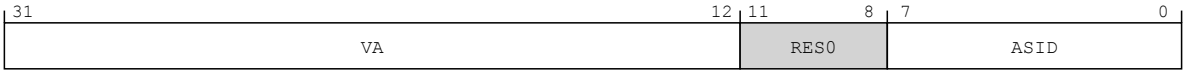

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing TLBIMVA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.150 TLBIMVAA, TLB Invalidate by VA, All ASID

The TLBIMVAA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVAA are UNDEFINED.

Attributes

TLBIMVAA is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.151 TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable

The TLBIMVAAIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVAAIS are UNDEFINED.

Attributes

TLBIMVAAIS is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAAIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b011

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLBIS
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TTLBIS ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.152 TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level

The TLBIMVAAL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVAAL are UNDEFINED.

Attributes

TLBIMVAAL is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAAL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.153 TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable

The TLBIMVAALIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVAALIS are UNDEFINED.

Attributes

TLBIMVAALIS is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAALIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLBIS
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TTLBIS ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.154 TLBIMVAH, TLB Invalidate by VA, Hyp mode

The TLBIMVAH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIMVAH are UNDEFINED.

Attributes

TLBIMVAH is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    else
        AArch32_TLBI_VA(SS_NonSecure, Regime_EL2, VMID(), Broadcast_NSH, TLBILevel_Any, TLBI_AllAttr,
            ↪R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.155 TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable

The TLBIMVAHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIMVAHIS are UNDEFINED.

Attributes

TLBIMVAHIS is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 0
VA	RES0

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    else
        AArch32_TLBI_VA(SS_NonSecure, Regime_EL2, VMID(), Broadcast_ISH, TLBILevel_Any, TLBI_AllAttr,
            ↪R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.156 TLBIMVAIS, TLB Invalidate by VA, Inner Shareable

The TLBIMVAIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVAIS are UNDEFINED.

Attributes

TLBIMVAIS is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBIMVAIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLBIS
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TTLBIS ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.157 TLBIMVAL, TLB Invalidate by VA, Last level

The TLBIMVAL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVAL are UNDEFINED.

Attributes

TLBIMVAL is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBIMVAL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.158 TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode

The TLBIMVALH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIMVALH are UNDEFINED.

Attributes

TLBIMVALH is a 32-bit System instruction.

Field descriptions

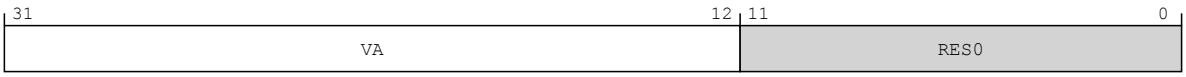

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVALH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    else
        AArch32_TLBI_VA(SS_NonSecure, Regime_EL2, VMID(), Broadcast_NSH, TLBILevel_Last, TLBI_AllAttr,
            ↪R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.159 TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable

The TLBIMVALHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to TLBIMVALHIS are UNDEFINED.

Attributes

TLBIMVALHIS is a 32-bit System instruction.

Field descriptions

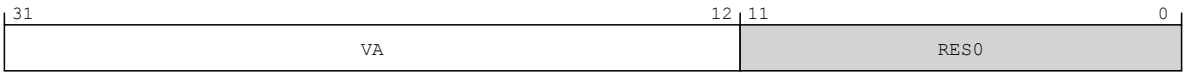

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVALHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        Undefined();
    else
        AArch32_TLBI_VA(SS_NonSecure, Regime_EL2, VMID(), Broadcast_ISH, TLBILevel_Last, TLBI_AllAttr,
            ↪R(t));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.160 TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable

The TLBIMVALIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This System instruction is not implemented in architecture versions before Armv8.

This system instruction is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBIMVALIS are UNDEFINED.

Attributes

TLBIMVALIS is a 32-bit System instruction.

Field descriptions

31 … 12	11 … 8	7 … 0
VA	RES0	ASID

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBIMVALIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the Pseudocode for AArch32 operation.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T8 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T8 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLB
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TTLBIS
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TTLB == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TTLBIS ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
elsif PSTATE.EL == EL3 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.161 TLBTR, TLB Type Register

The TLBTR characteristics are:

Purpose

Provides information about the TLB implementation. The register must define whether the implementation provides separate instruction and data TLBs, or a unified TLB. Normally, the IMPLEMENTATION DEFINED information in this register includes the number of lockable entries in the TLB.

Configuration

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TLBTR are UNDEFINED.

Attributes

TLBTR is a 32-bit register.

Field descriptions

31 … 1	0
IMPLEMENTATION DEFINED	nU

IMPLEMENTATION DEFINED, bits [31:1]

IMPLEMENTATION DEFINED.

nU, bit [0]

Not Unified TLB. Indicates whether the implementation has a unified TLB.

The value of this field is an IMPLEMENTATION DEFINED choice of:

nU	Meaning
0b0	Unified TLB.
0b1	Separate Instruction and Data TLBs.

Access to this field is RO.

Accessing TLBTR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TID1
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TID1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TLBTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = TLBTR();
elsif PSTATE.EL == EL3 then
    R(t) = TLBTR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.162 TPIDRPRW, PL1 Software Thread ID Register

The TPIDRPRW characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is not visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configuration

Note

The PE never updates this register.

AArch32 System register TPIDRPRW bits [31:0] are architecturally mapped to AArch64 System register TPIDR_EL1[31:0].

This register is banked between TPIDRPRW and TPIDRPRW_S and TPIDRPRW_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TPIDRPRW are UNDEFINED.

Attributes

TPIDRPRW is a 32-bit register.

This register has the following instances:

- TPIDRPRW, when EL3 is not implemented or FEAT_AA64 is implemented.
- TPIDRPRW_S, when FEAT_AA32EL3 is implemented.
- TPIDRPRW_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
Thread ID

TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRPRW

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TPIDRPRW_NS();
    else
        R(t) = TPIDRPRW();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TPIDRPRW_NS();
    else
        R(t) = TPIDRPRW();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TPIDRPRW_S();
    else
        R(t) = TPIDRPRW_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TPIDRPRW_NS() = R(t);
    else
        TPIDRPRW() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TPIDRPRW_NS() = R(t);
    else
        TPIDRPRW() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        TPIDRPRW_S() = R(t);
    else
        TPIDRPRW_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.163 TPIDRURO, PL0 Read-Only Software Thread ID Register

The TPIDRURO characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configuration

Note

The PE never updates this register.

AArch32 System register TPIDRURO bits [31:0] are architecturally mapped to AArch64 System register TPIDRRO_EL0[31:0].

This register is banked between TPIDRURO and TPIDRURO_S and TPIDRURO_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to TPIDRURO are UNDEFINED.

Attributes

TPIDRURO is a 32-bit register.

This register has the following instances:

- TPIDRURO, when EL3 is not implemented or FEAT_AA64 is implemented.
- TPIDRURO_S, when FEAT_AA32EL3 is implemented.
- TPIDRURO_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
Thread ID

TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRURO

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && !ELIsInHost(EL0)
        ↪&& HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().TPIDRRO_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        R(t) = TPIDRURO();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TPIDRURO_NS();
    else
        R(t) = TPIDRURO();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TPIDRURO_NS();
    else
        R(t) = TPIDRURO();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TPIDRURO_S();
    else
        R(t) = TPIDRURO_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b011

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        TPIDRURO_NS() = R(t);
    else
        TPIDRURO() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TPIDRURO_NS() = R(t);
    else
        TPIDRURO() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        TPIDRURO_S() = R(t);
    else
        TPIDRURO_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.164 TPIDRURW, PL0 Read/Write Software Thread ID Register

The TPIDRURW characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configuration

Note

The PE never updates this register.

AArch32 System register TPIDRURW bits [31:0] are architecturally mapped to AArch64 System register TPIDR_EL0[31:0].

This register is banked between TPIDRURW and TPIDRURW_S and TPIDRURW_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to TPIDRURW are UNDEFINED.

Attributes

TPIDRURW is a 32-bit register.

This register has the following instances:

- TPIDRURW, when EL3 is not implemented or FEAT_AA64 is implemented.
- TPIDRURW_S, when FEAT_AA32EL3 is implemented.
- TPIDRURW_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
Thread ID

TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRURW

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && !ELIsInHost(EL0)
        ↪&& HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGRTR_EL2().TPIDR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        R(t) = TPIDRURW();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TPIDRURW_NS();
    else
        R(t) = TPIDRURW();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TPIDRURW_NS();
    else
        R(t) = TPIDRURW();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TPIDRURW_S();
    else
        R(t) = TPIDRURW_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && !ELIsInHost(EL0)
        ↪&& HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HFGWTR_EL2().TPIDR_EL0 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        TPIDRURW() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13 ==
     ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
     ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TPIDRURW_NS() = R(t);
    else
        TPIDRURW() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TPIDRURW_NS() = R(t);
    else
        TPIDRURW() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        TPIDRURW_S() = R(t);
    else
        TPIDRURW_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.165 TTBCR, Translation Table Base Control Register

The TTBCR characteristics are:

Purpose

The control register for stage 1 of the PL1&0 translation regime. Its controls include:

- Where the VA range is split between addresses translated using TTBR0 and addresses translated using TTBR1.
- The translation table format used by this stage of translation.

From Armv8.2, when the value of TTBCR.{EAE, T2E} is {1, 1}, TTBCR is used with TTBCR2.

Configuration

The current translation table format determines which format of the register is used.

Some RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32, then:

- The EAE bit resets to 0 in both the Secure and the Non-secure instances of the register.
- Other reset values apply only to the Secure instance of the register.

AArch32 System register TTBCR bits [31:0] are architecturally mapped to AArch64 System register TCR_EL1[31:0].

This register is banked between TTBCR and TTBCR_S and TTBCR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TTBCR are UNDEFINED.

Attributes

TTBCR is a 32-bit register.

This register has the following instances:

- TTBCR, when EL3 is not implemented or FEAT_AA64 is implemented.
- TTBCR_S, when FEAT_AA32EL3 is implemented.
- TTBCR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

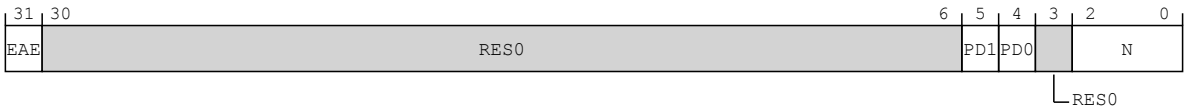

EAE, bit [31]

Extended Address Enable.

EAE	Meaning
0b0	Use the VMSAv8-32 translation system with the Short-descriptor translation table format.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Bits [30:6]

Reserved, RES0.

PD1, bit [5]

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1.

PD1	Meaning
`0b0`	Perform translation table walks using TTBR1.
`0b1`	A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

PD0, bit [4]

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss for an address that is translated using TTBR0.

PD0	Meaning
`0b0`	Perform translation table walks using TTBR0.
`0b1`	A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bit [3]

Reserved, RES0.

N, bits [2:0]

Indicate the width of the base address held in TTBR0. In TTBR0, the base address field is bits[31:14-N]. The value of N also determines:

- Whether TTBR0 or TTBR1 is used as the base address for translation table walks.
- The size of the translation table pointed to by TTBR0.

N can take any value from 0 to 7, that is, from `0b000` to `0b111`.

When N has its reset value of 0, the translation table base is compatible with Armv5 and Armv6.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'000'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TTBCR.EAE == '1':

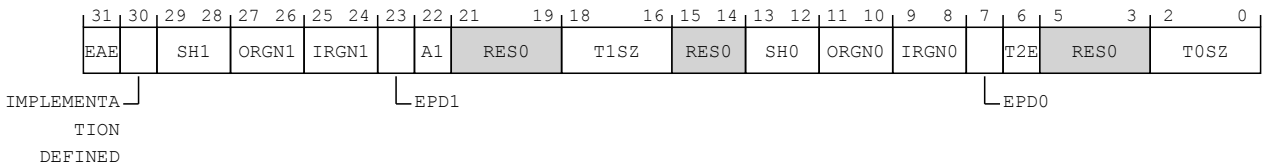

EAE, bit [31]

Extended Address Enable.

EAE	Meaning
`0b1`	Use the VMSAv8-32 translation system with the Long-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

IMPLEMENTATION DEFINED, bit [30]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using TTBR1.

SH1	Meaning
`0b00`	Non-shareable.
`0b10`	Outer Shareable.
`0b11`	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'00'`.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using TTBR1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ORGN1	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using TTBR1.

IRGN1	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

EPD1, bit [23]

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1.

EPD1	Meaning
0b0	Perform translation table walks using TTBR1.
0b1	A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

A1, bit [22]

Selects whether TTBR0 or TTBR1 defines the ASID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1	Meaning
0b0	TTBR0.ASID defines the ASID.
0b1	TTBR1.ASID defines the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [21:19]

Reserved, RES0.

T1SZ, bits [18:16]

See 'Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format' for how TTBCR.{T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '000'.

Bits [15:14]

Reserved, RES0.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0.

SH0	Meaning
0b00	Non-shareable
0b10	Outer Shareable
0b11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

EPD0, bit [7]

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0.

EPD0	Meaning
0b0	Perform translation table walks using TTBR0.
0b1	A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

T2E, bit [6]

When FEAT_AA32HPD is implemented:

TTBCR2 Enable.

T2E	Meaning
0b0	TTBCR2 is disabled. The contents of TTBCR2 are treated as 0 for all purposes other than reading or writing the register.
0b1	TTBCR2 is enabled.

If TTBCR.EAE==0, then the behavior is as if the bit is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [5:3]

Reserved, RES0.

T0SZ, bits [2:0]

See 'Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format' for how TTBCR.{T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'000'`.

Accessing TTBCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBCR_NS();
    else
        R(t) = TTBCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBCR_NS();
    else
        R(t) = TTBCR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TTBCR_S();
    else
        R(t) = TTBCR_NS();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBCR_NS() = R(t);
    else
        TTBCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBCR_NS() = R(t);
    else
        TTBCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            TTBCR_S() = R(t);
        else
            TTBCR_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.166 TTBCR2, Translation Table Base Control Register 2

The TTBCR2 characteristics are:

Purpose

The second control register for stage 1 of the PL1&0 translation regime.

If FEAT_AA32HPD is not implemented then this register is not implemented and its encoding is UNDEFINED. Otherwise:

- When the value of TTBCR.{EAE, T2E} is not {1, 1} the contents of TTBCR2 are treated as zero for all purposes other than reading or writing the register.
- When the value of TTBCR.{EAE, T2E} is {1, 1} TTBCR2 is used with TTBCR.

Configuration

AArch32 System register TTBCR2 bits [31:0] are architecturally mapped to AArch64 System register TCR_EL1[63:32].

This register is banked between TTBCR2 and TTBCR2_S and TTBCR2_NS.

This register is present only when FEAT_AA32EL1 is implemented and FEAT_AA32HPD is implemented. Otherwise, direct accesses to TTBCR2 are UNDEFINED.

Attributes

TTBCR2 is a 32-bit register.

This register has the following instances:

- TTBCR2, when EL3 is not implemented or FEAT_AA64 is implemented.
- TTBCR2_S, when FEAT_AA32EL3 is implemented.
- TTBCR2_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

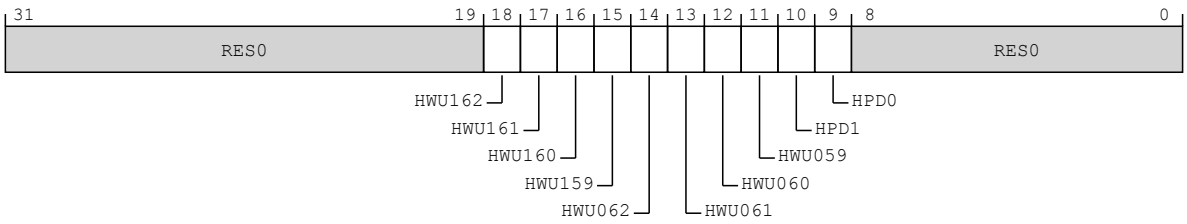

Bits [31:19]

Reserved, RES0.

HWU162, bit [18]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU162	Meaning
0b0	For translations using TTBR1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU161, bit [17]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1.

HWU161	Meaning
0b0	For translations using TTBR1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [16]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1.

HWU160	Meaning
0b0	For translations using TTBR1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU160	Meaning
0b1	For translations using TTBR1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [15]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1.

HWU159	Meaning
0b0	For translations using TTBR1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [14]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0.

HWU062	Meaning
0b0	For translations using TTBR0, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [13]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0.

HWU061	Meaning
0b0	For translations using TTBR0, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU060, bit [12]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0.

HWU060	Meaning
0b0	For translations using TTBR0, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU059, bit [11]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0.

HWU059	Meaning
0b0	For translations using TTBR0, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [10]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the translation tables pointed to by TTBR1.

HPD1	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled if TTBCR.T2E == 1.

When disabled, the permissions are treated as if the bits are 0.

The Effective value of this field is 0 if the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HPD0, bit [9]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the translation tables pointed to by TTBR0.

HPD0	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled if TTBCR.T2E ==1.

When disabled, the permissions are treated is as if the bits are 0.

The Effective value of this field is 0 if the value of TTBCR.T2E is 0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:0]

Reserved, RES0.

Accessing TTBCR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b011

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA32HPD)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBCR2_NS();
    else
        R(t) = TTBCR2();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBCR2_NS();
    else
        R(t) = TTBCR2();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TTBCR2_S();
    else
        R(t) = TTBCR2_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_AA32HPD)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBCR2_NS() = R(t);
    else
        TTBCR2() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBCR2_NS() = R(t);
    else
        TTBCR2() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            TTBCR2_S() = R(t);
        else
            TTBCR2_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.167 TTBR0, Translation Table Base Register 0

The TTBR0 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the PL1&0 translation regime, and other information for this translation regime.

Configuration

TTBR0 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

TTBCR.EAE determines which TTBR0 format is used:

- TTBCR.EAE is 0: 32-bit format is used. TTBR0[63:32] are ignored.
- TTBCR.EAE is 1: 64-bit format is used.

When EL3 is using AArch32, write access to TTBR0(S) is disabled when the **CP15SDISABLE** signal is asserted HIGH.

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

AArch32 System register TTBR0 bits [63:0] are architecturally mapped to AArch64 System register TTBR0_EL1[63:0].

This register is banked between TTBR0 and TTBR0_S and TTBR0_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TTBR0 are UNDEFINED.

Attributes

TTBR0 is a 64-bit register.

This register has the following instances:

- TTBR0, when EL3 is not implemented or FEAT_AA64 is implemented.
- TTBR0_S, when FEAT_AA32EL3 is implemented.
- TTBR0_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

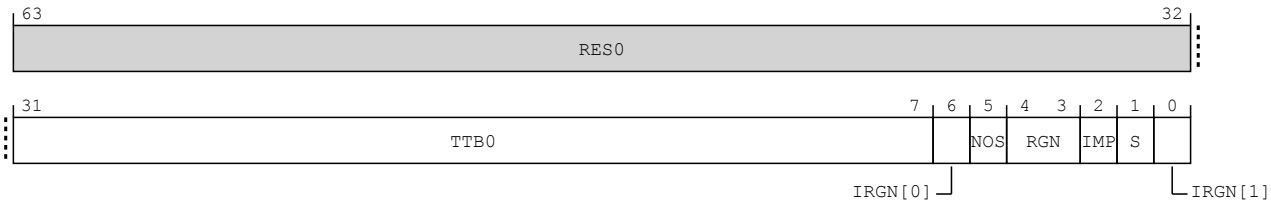

Bits [63:32]

Reserved, RES0.

TTB0, bits [31:7]

Translation table base address, bits[31:x], where x is 14-(TTBCR.N). Register bits [x-1:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Register bits [x-1:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN, bits [0, 6]

Inner region bits. Bits [0,6] of this register together indicate the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

IRGN	Meaning
`0b00`	Normal memory, Inner Non-cacheable.
`0b01`	Normal memory, Inner Write-Back Write-Allocate Cacheable.
`0b10`	Normal memory, Inner Write-Through Cacheable.
`0b11`	Normal memory, Inner Write-Back no Write-Allocate Cacheable.

Note

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for ARMv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NOS, bit [5]

Not Outer Shareable. When the value of TTBR0.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

NOS	Meaning
`0b0`	Memory is Outer Shareable.
`0b1`	Memory is Inner Shareable.

This bit is ignored when the value of TTBR0.S is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RGN, bits [4:3]

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RGN	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Cacheable.
0b11	Normal memory, Outer Write-Back no Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMP, bit [2]

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [1]

Shareable. Indicates whether the memory associated with the translation table walks is Shareable:

S	Meaning
0b0	Memory is Non-shareable.
0b1	Memory is Shareable. The TTBR0.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When TTBCR.EAE == '1':

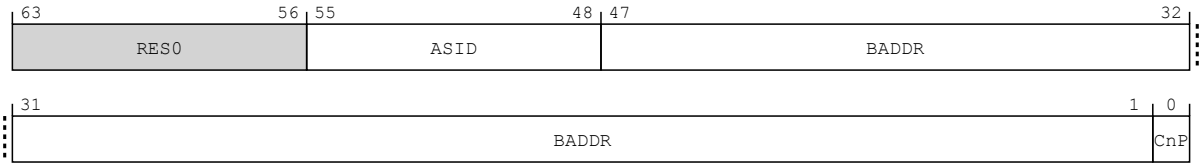

Bits [63:56]

Reserved, RES0.

ASID, bits [55:48]

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T0SZ as follows:

- If TTBCR.T0SZ is 0 or 1, x = 5 - TTBCR.T0SZ.
- If TTBCR.T0SZ is greater than 1, x = 14 - TTBCR.T0SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. When TTBCR.EAE ==1, this bit indicates whether each entry that is pointed to by TTBR0 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by this instance of TTBR0, for the current ASID, are permitted to differ from corresponding entries for this instance of TTBR0 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR0.CnP on those other PEs. • The value of TTBCR.EAE on those other PEs. • The value of the current ASID or, for the Non-secure instance of TTBR0, the value of the current VMID.
`0b1`	The translation table entries pointed to by this instance of TTBR0 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0.CnP is 1 for this instance of TTBR0 and all of the following apply: • The translation table entries are pointed to by this instance of TTBR0. • The value of the applicable TTBCR.EAE field is 1. • The ASID is the same as the current ASID. • For the Non-secure instance of TTBR0, the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR0.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBR0_NS()[31:0];
    else
        R(t) = TTBR0()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBR0_NS()[31:0];
    else
        R(t) = TTBR0()[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TTBR0_S()[31:0];
    else
        R(t) = TTBR0_NS()[31:0];
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS()[31:0] = R(t);
    else
        TTBR0()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS()[31:0] = R(t);
    else
        TTBR0()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            TTBR0_S()[31:0] = R(t);
        else
            TTBR0_NS()[31:0] = R(t);
        end;
    end;
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = TTBR0_NS();
    else
        R(t, t2) = TTBR0();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = TTBR0_NS();
    else
        R(t, t2) = TTBR0();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t, t2) = TTBR0_S();
    else
        R(t, t2) = TTBR0_NS();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS() = R(t, t2);
    else
        TTBR0() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR0_NS() = R(t, t2);
    else
        TTBR0() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            TTBR0_S() = R(t, t2);
        else
            TTBR0_NS() = R(t, t2);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.168 TTBR1, Translation Table Base Register 1

The TTBR1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the PL1&0 translation regime, and other information for this translation regime.

Configuration

TTBR1 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

TTBCR.EAE determines which TTBR1 format is used:

- TTBCR.EAE is 0: 32-bit format is used. TTBR1[63:32] are ignored.
- TTBCR.EAE is 1: 64-bit format is used.

Used in conjunction with the TTBCR. When the 64-bit TTBR1 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR1.

AArch32 System register TTBR1 bits [63:0] are architecturally mapped to AArch64 System register TTBR1_EL1[63:0].

This register is banked between TTBR1 and TTBR1_S and TTBR1_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to TTBR1 are UNDEFINED.

Attributes

TTBR1 is a 64-bit register.

This register has the following instances:

- TTBR1, when EL3 is not implemented or FEAT_AA64 is implemented.
- TTBR1_S, when FEAT_AA32EL3 is implemented.
- TTBR1_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

When TTBCR.EAE == '0':

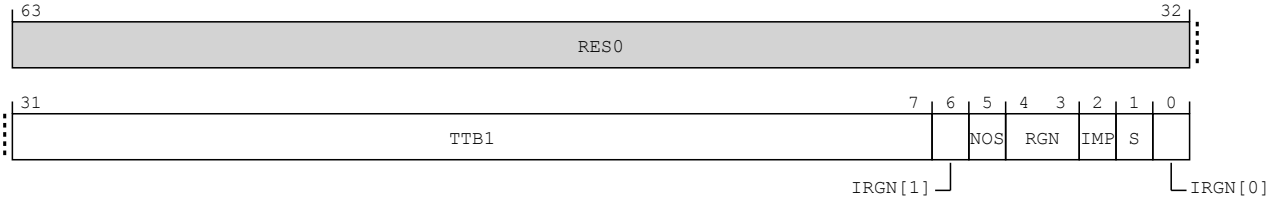

Bits [63:32]

Reserved, RES0.

TTB1, bits [31:7]

Translation table base address, bits[31:14]. Register bits [13:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [13:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN, bits [6, 0]

Inner region bits. IRGN[1:0] indicate the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

IRGN	Meaning
`0b00`	Normal memory, Inner Non-cacheable.
`0b01`	Normal memory, Inner Write-Back Write-Allocate Cacheable.
`0b10`	Normal memory, Inner Write-Through Cacheable.
`0b11`	Normal memory, Inner Write-Back no Write-Allocate Cacheable.

Note

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for Armv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NOS, bit [5]

Not Outer Shareable. When the value of TTBR1.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

NOS	Meaning
`0b0`	Memory is Outer Shareable.
`0b1`	Memory is Inner Shareable.

This bit is ignored when the value of TTBR1.S is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RGN, bits [4:3]

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RGN	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Cacheable.
0b11	Normal memory, Outer Write-Back no Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMP, bit [2]

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [1]

Shareable. Indicates whether the memory associated with the translation table walks is Shareable:

S	Meaning
0b0	Memory is Non-shareable.
0b1	Memory is Shareable. The TTBR1.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When TTBCR.EAE == '1':

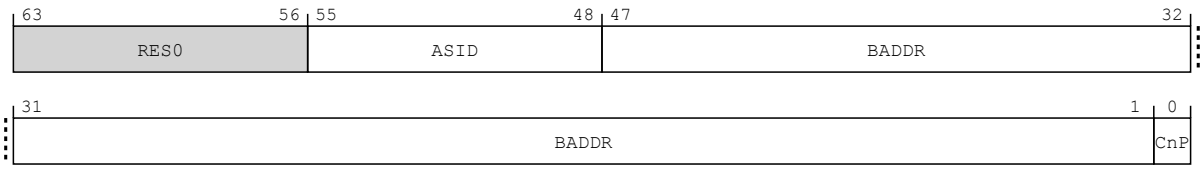

Bits [63:56]

Reserved, RES0.

ASID, bits [55:48]

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T1SZ as follows:

- If TTBCR.T1SZ is 0 or 1, x = 5 - TTBCR.T1SZ.
- If TTBCR.T1SZ is greater than 1, x = 14 - TTBCR.T1SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. When TTBCR.EAE ==1, this bit indicates whether each entry that is pointed to by TTBR1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by this instance of TTBR1, for the current ASID, are permitted to differ from corresponding entries for this instance of TTBR1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR1.CnP on those other PEs. • The value of TTBCR.EAE on those other PEs. • The value of the current ASID or, for the Non-secure instance of TTBR1, the value of the current VMID.
`0b1`	The translation table entries pointed to by this instance of TTBR1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1.CnP is 1 for this instance of TTBR1 and all of the following apply: • The translation table entries are pointed to by this instance of TTBR1. • The value of the applicable TTBCR.EAE field is 1. • The ASID is the same as the current ASID. • For the Non-secure instance of TTBR1, the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBR1_NS()[31:0];
    else
        R(t) = TTBR1()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = TTBR1_NS()[31:0];
    else
        R(t) = TTBR1()[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = TTBR1_S()[31:0];
    else
        R(t) = TTBR1_NS()[31:0];
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0010	0b0000	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS()[31:0] = R(t);
    else
        TTBR1()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS()[31:0] = R(t);
    else
        TTBR1()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            TTBR1_S()[31:0] = R(t);
        else
            TTBR1_NS()[31:0] = R(t);
        end;
    end;
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TRVM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TRVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = TTBR1_NS();
    else
        R(t, t2) = TTBR1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = TTBR1_NS();
    else
        R(t, t2) = TTBR1();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t, t2) = TTBR1_S();
    else
        R(t, t2) = TTBR1_NS();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0001

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TVM ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().TVM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS() = R(t, t2);
    else
        TTBR1() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS() = R(t, t2);
    else
        TTBR1() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            TTBR1_S() = R(t, t2);
        else
            TTBR1_NS() = R(t, t2);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.169 VBAR, Vector Base Address Register

The VBAR characteristics are:

Purpose

When high exception vectors are not selected, holds the vector base address for exceptions that are not taken to Monitor mode or to Hyp mode.

Software must program VBAR(NS) with the required initial value as part of the PE boot sequence.

Configuration

AArch32 System register VBAR bits [31:0] are architecturally mapped to AArch64 System register VBAR_EL1[31:0].

This register is banked between VBAR and VBAR_S and VBAR_NS.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to VBAR are UNDEFINED.

Attributes

VBAR is a 32-bit register.

This register has the following instances:

- VBAR, when EL3 is not implemented or FEAT_AA64 is implemented.
- VBAR_S, when FEAT_AA32EL3 is implemented.
- VBAR_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 5	4 … 0
Vector Base Address	RES0

VBA, bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bits [4:0]

Reserved, RES0.

Accessing VBAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = VBAR_NS();
    else
        R(t) = VBAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = VBAR_NS();
    else
        R(t) = VBAR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = VBAR_S();
    else
        R(t) = VBAR_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        VBAR_NS() = R(t);
    else
        VBAR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        VBAR_NS() = R(t);
    else
        VBAR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' && CP15SDISABLE == HIGH then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif SCR().NS == '0' && CP15SDISABLE2 == HIGH then
        Undefined();
    else
        if SCR().NS == '0' then
            VBAR_S() = R(t);
        else
            VBAR_NS() = R(t);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.170 VMPIDR, Virtualization Multiprocessor ID Register

The VMPIDR characteristics are:

Purpose

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by Non-secure EL1 reads of MPIDR, which in a multiprocessor system, provides an additional PE identification system.

Configuration

If EL2 is not implemented but EL3 is implemented, this register takes the value of the MPIDR.

AArch32 System register VMPIDR bits [31:0] are architecturally mapped to AArch64 System register VMPIDR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to VMPIDR are UNDEFINED.

Attributes

VMPIDR is a 32-bit register.

Field descriptions

31	30	29 25	24	23 16	15 8	7 0
M	U	RES0	MT	Aff2	Aff1	Aff0

M, bit [31]

Indicates whether this implementation includes the functionality introduced by the Armv7 Multiprocessing Extensions.

M	Meaning
0b0	This implementation does not include the Armv7 Multiprocessing Extensions functionality.
0b1	This implementation includes the Armv7 Multiprocessing Extensions functionality.

Access to this field is RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

The reset behavior of this field is:

- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MPIDR.U.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels.

MT	Meaning
`0b0`	Performance of PEs at the lowest affinity level is largely independent.
`0b1`	Performance of PEs at the lowest affinity level is very interdependent.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MPIDR.MT.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MPIDR.Aff2.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MPIDR.Aff1.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MPIDR.Aff0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing VMPIDR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VMPIDR();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        R(t) = MPIDR();
    elsif SCR().NS == '0' then
        Undefined();
    else
        R(t) = VMPIDR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VMPIDR() = R(t);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif SCR().NS == '0' then
        Undefined();
    else
        VMPIDR() = R(t);
    end;
end;
```

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b101

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        R(t) = VMPIDR_EL2()[31:0];
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
        R(t) = VMPIDR();
    else
        R(t) = MPIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MPIDR();
elsif PSTATE.EL == EL3 then
    R(t) = MPIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.171 VPIDR, Virtualization Processor ID Register

The VPIDR characteristics are:

Purpose

Holds the value of the Virtualization Processor ID. This is the value returned by Non-secure EL1 reads of MIDR.

Configuration

If EL2 is not implemented but EL3 is implemented, this register takes the value of the MIDR.

AArch32 System register VPIDR bits [31:0] are architecturally mapped to AArch64 System register VPIDR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to VPIDR are UNDEFINED.

Attributes

VPIDR is a 32-bit register.

Field descriptions

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
Implementer	Variant	Architecture	PartNum	Revision

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MIDR.Implementer.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MIDR.Variant.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Architecture, bits [19:16]

Architecture version.

Architecture	Meaning
`0b0001`	Armv4.
`0b0010`	Armv4T.
`0b0011`	Armv5 (obsolete).
`0b0100`	Armv5T.
`0b0101`	Armv5TE.
`0b0110`	Armv5TEJ.
`0b0111`	Armv6.
`0b1111`	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MIDR.Architecture.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MIDR.PartNum.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the value in MIDR.Revision.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing VPIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VPIDR();
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        R(t) = MIDR();
    elsif SCR().NS == '0' then
        Undefined();
    else
        R(t) = VPIDR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VPIDR() = R(t);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return;
    elsif SCR().NS == '0' then
        Undefined();
    else
        VPIDR() = R(t);
    end;
end;
```

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T0 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T0 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        R(t) = VPIDR_EL2()[31:0];
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
        R(t) = VPIDR();
    else
        R(t) = MIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MIDR();
elsif PSTATE.EL == EL3 then
    R(t) = MIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.172 VTCR, Virtualization Translation Control Register

The VTCR characteristics are:

Purpose

The control register for stage 2 of the Non-secure PL1&0 translation regime.

Note

This stage of translation always uses the Long-descriptor translation table format.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register VTCR bits [31:0] are architecturally mapped to AArch64 System register VTCR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to VTCR are UNDEFINED.

Attributes

VTCR is a 32-bit register.

Field descriptions

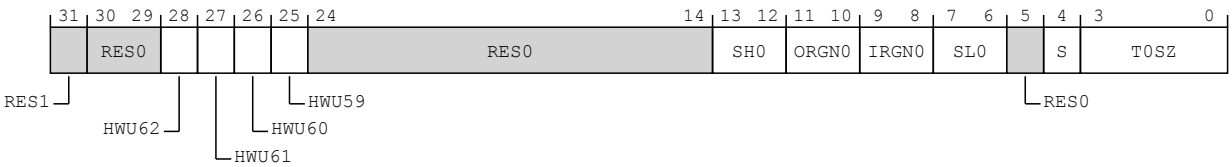

Bit [31]

Reserved, RES1.

Bits [30:29]

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 2 translation table Block or Page entry.

HWU62	Meaning
0b0	Bit[62] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[62] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 2 translation table Block or Page entry.

HWU61	Meaning
0b0	Bit[61] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[61] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 2 translation table Block or Page entry.

HWU60	Meaning
0b0	Bit[60] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[60] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 2 translation table Block or Page entry.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU59	Meaning
0b0	Bit[59] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	Bit[59] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:14]

Reserved, RES0.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using VTTBR.

SH0	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using VTTBR.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Inner cacheability attribute for memory associated with translation table walks using VTTBR.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SL0, bits [7:6]

Starting level for translation table walks using VTTBR.

SL0	Meaning
0b00	Start at level 2
0b01	Start at level 1

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of T0SZ, then a stage 2 level 1 Translation fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

S, bit [4]

Sign extension bit. This bit must be programmed to the value of T0SZ[3]. If it is not, then the stage 2 T0SZ value is treated as an UNKNOWN value

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T0SZ, bits [3:0]

The size offset of the memory region addressed by VTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

This field holds a four-bit signed integer value, meaning it supports values from -8 to 7.

Note

This is different from the other translation control registers, where TnSZ holds a three-bit unsigned integer, supporting values from 0 to 7.

If this field is programmed to a value that is not consistent with the programming of SL0 then a stage 2 level 1 Translation fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing VTCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VTCR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = VTCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VTCR() = R(t);
elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if SCR().NS == '0' then
        Undefined();
    else
        VTCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.2.173 VTTBR, Virtualization Translation Table Base Register

The VTTBR characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the Non-secure PL1&0 translation regime, and other information for this translation regime.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register VTTBR bits [63:0] are architecturally mapped to AArch64 System register VTTBR_EL2[63:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to VTTBR are UNDEFINED.

Attributes

VTTBR is a 64-bit register.

Field descriptions

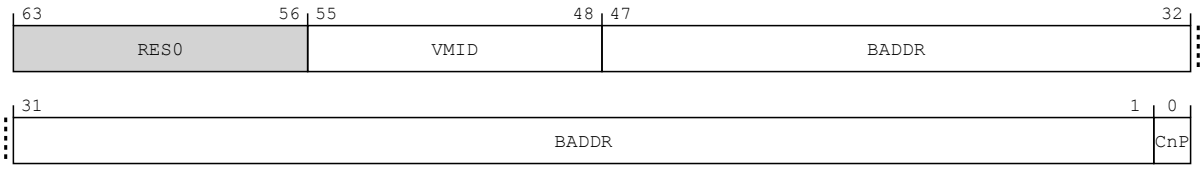

Bits [63:56]

Reserved, RES0.

VMID, bits [55:48]

The VMID for the translation table.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to the expression `0x00`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of VTCR.SL0 and VTCR.T0SZ as follows:

- If VTCR.SL0 is `0b00`, meaning that lookup starts at level 2, then x is 14 - VTCR.T0SZ.
- If VTCR.SL0 is `0b01`, meaning that lookup starts at level 1, then x is 5 - VTCR.T0SZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If VTCR.SL0 is either `0b10` or `0b11` then a stage 2 level 1 Translation fault is generated.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by VTTBR are permitted to differ from the entries for VTTBR for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.
`0b1`	The translation table entries pointed to by VTTBR are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR.CnP is 1 and the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the VTTBR.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBRs do not point to the same translation table entries when the VMID value is the same as the current VMID, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing VTTBR

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
`0b1111`	`0b0010`	`0b0110`

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = VTTBR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t, t2) = VTTBR();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b0010	0b0110

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T2 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T2 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x04);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VTTBR() = R(t, t2);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        VTTBR() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3 Debug System registers

This section lists the Debug System registers in AArch32 state, in alphabetic order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.1 DBGAUTHSTATUS, Debug Authentication Status register

The DBGAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configuration

This register is required in all implementations.

AArch32 System register DBGAUTHSTATUS bits [31:0] are architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

AArch32 System register DBGAUTHSTATUS bits [31:0] are architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGAUTHSTATUS are UNDEFINED.

Attributes

DBGAUTHSTATUS is a 32-bit register.

Field descriptions

31 … 8	7 6	5 4	3 2	1 0
RES0	SNID	SID	NSNID	NSID

Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:

Secure Non-Invasive Debug.

This field has the same value as DBGAUTHSTATUS.SID.

Otherwise:

Secure Non-Invasive Debug.

SNID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. `ExternalSecureNoninvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure Invasive Debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. `ExternalSecureInvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalSecureInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

NSNID, bits [3:2]

When FEAT_Debugv8p4 is implemented:

Non-secure Non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b11	Implemented and enabled. EL3 is implemented or the Effective value of SCR.NS is 1.

All other values are reserved.

Otherwise:

Non-secure Non-Invasive Debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. `ExternalNoninvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure Invasive Debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. `ExternalInvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DBGAUTHSTATUS

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1110	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGAUTHSTATUS();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGAUTHSTATUS();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGAUTHSTATUS();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.2 DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n> characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

Configuration

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGBCR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

AArch32 System register DBGBCR<n> bits [31:0] are architecturally mapped to External register DBGBCR<n>_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGBCR<n> are UNDEFINED.

Attributes

DBGBCR<n> is a 32-bit register.

Field descriptions

31 – 24	23 – 20	19 – 16	15 – 14	13	12 – 9	8 – 5	4 – 3	2 – 1	0
RES0	BT	LBN	SSC	HMC	RES0	BAS	RES0	PMC	E

When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

BT	Meaning
0b0000	Unlinked instruction address match. DBGBVR<n> is the address of an instruction.
0b0001	As 0b0000, but linked to a Context matching breakpoint.
0b0010	Unlinked Context ID match. If the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>.ContextID is compared against CONTEXTIDR_EL2. Otherwise, DBGBVR<n>.ContextID is compared against CONTEXTIDR.
0b0011	As 0b0010 with linking enabled.
0b0100	Unlinked instruction address mismatch. DBGBVR<n> is the address of an instruction to be stepped.
0b0101	As 0b0100, but linked to a Context matching breakpoint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning
0b0110	Unlinked CONTEXTIDR match. DBGBVR<n>.ContextID is a Context ID compared against CONTEXTIDR.
0b0111	As 0b0110 with linking enabled.
0b1000	Unlinked VMID match. DBGBXVR<n>.VMID is a VMID compared against VTTBR.VMID.
0b1001	As 0b1000 with linking enabled.
0b1010	Unlinked VMID and Context ID match. DBGBVR<n>.ContextID is a Context ID compared against CONTEXTIDR, and DBGBXVR<n>.VMID is a VMID compared against VTTBR.VMID.
0b1011	As 0b1010 with linking enabled.
0b1100	Unlinked CONTEXTIDR_EL2 match. DBGBXVR<n>.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.
0b1101	As 0b1100 with linking enabled.
0b1110	Unlinked Full Context ID match. DBGBVR<n>.ContextID is compared against CONTEXTIDR, and DBGBXVR<n>.ContextID2 is compared against CONTEXTIDR_EL2.
0b1111	As 0b1110 with linking enabled.

For more information on Breakpoints and their constraints, see ‘Breakpoint exceptions’ and ‘Reserved DBGBCR<n>.BT values’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number. For Linked address matching breakpoints, this specifies the index of the context-matching breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>.E is 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields.

For more information, see ‘Execution conditions for which a breakpoint generates Breakpoint exceptions’ and ‘Reserved DBGBCR<n>.{SSC, HMC, PMC} values’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

For more information on the operation of the SSC, HMC, and PMC fields, see ‘Execution conditions for which a breakpoint generates Breakpoint exceptions’.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which halfwords an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0011	DBGBVR<n>	Use for T32 instructions
0b1100	DBGBVR<n>+2	Use for T32 instructions
0b1111	DBGBVR<n>	Use for A32 instructions

All other values are reserved. For more information, see 'Reserved DBGBCR<n>.BAS values'.

For more information on using the BAS field in Address Match breakpoints, see 'Using the BAS field in Address Match breakpoints'.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Step instruction at	Constraint for debuggers
0b0000	•	Use for a match anywhere breakpoint
0b0011	DBGBVR<n>	Use for T32 instructions
0b1100	DBGBVR<n>+2	Use for T32 instructions
0b1111	DBGBVR<n>	Use for A32 instructions

All other values are reserved. For more information, see 'Reserved DBGBCR<n>.BAS values'.

For more information on using the BAS field in address mismatch breakpoints, see 'Using the BAS field in Address Match breakpoints'.

For Context matching breakpoints, this field is RES1 and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>.SSC description.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint DBGBVR<n>. Possible values are:

E	Meaning
0b0	Breakpoint disabled.
0b1	Breakpoint enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b101

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = DBGBCR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBCR(m);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBCR(m);
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b101

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        DBGBCR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR(m) = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.3 DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, either the virtual address of an instruction or a context ID. Forms breakpoint n together with control register DBGBCR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

Configuration

How this register is interpreted depends on the value of DBGBCR<n>.BT.

- When DBGBCR<n>.BT is `0b0x0x`, this register holds a virtual address.
- When DBGBCR<n>.BT is `0bxx1x`, this register holds a Context ID.

For other values of DBGBCR<n>.BT, this register is RES0.

Some breakpoints might not support Context ID comparison. For more information, see the description of the DBGDIDR.CTX_CMPs field.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGBVR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGBVR<n>_EL1[31:0].

AArch32 System register DBGBVR<n> bits [31:0] are architecturally mapped to External register DBGBVR<n>_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGBVR<n> are UNDEFINED.

Attributes

DBGBVR<n> is a 32-bit register.

Field descriptions

When DBGBCR<n>.BT IN {'`0x0`x'}:

31 … 2	1 … 0
VA[31:2]	RES0

VA[31:2], bits [31:2]

Bits[31:2] of the address value for comparison.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>.BT IN {'001x'}:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	0
ContextID	

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when all of the following are true:

- The Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- The PE is executing at EL0.

Otherwise, the value is compared against CONTEXTIDR.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>.BT IN {'101x'} and EL2 is implemented:

31	0
ContextID	

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>.BT IN {'x11x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

31	0
ContextID	

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<n>

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b100

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBVR(m);
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b100

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR(m) = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.4 DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15

The DBGBXVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, to support VMID matching. Used in conjunction with a control register DBGBCR<n> and a value register DBGBVR<n>, where EL2 is implemented and breakpoint n supports Context matching.

Configuration

How this register is interpreted depends on the value of DBGBCR<n>.BT.

- When DBGBCR<n>.BT is `0b10xx`, this register holds a VMID.
- When DBGBCR<n>.BT is `0b11xx`, this register holds a Context ID.

For other values of DBGBCR<n>.BT, this register is RES0.

Accesses to this register are UNDEFINED in any of the following cases:

- Breakpoint n is not implemented.
- Breakpoint n does not support Context matching.
- EL2 is not implemented.

For more information, see the description of the DBGDIDR.CTX_CMPs field.

AArch32 System register DBGBXVR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:32].

AArch32 System register DBGBXVR<n> bits [31:0] are architecturally mapped to External register DBGBVR<n>_EL1[63:32].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGBXVR<n> are UNDEFINED.

Attributes

DBGBXVR<n> is a 32-bit register.

Field descriptions

When DBGBCR<n>.BT IN {'10xx'} and EL2 is implemented:

31 … 16	15 … 8	7 … 0
RES0	VMID[15:8]	VMID[7:0]

Bits [31:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID[7:0], bits [7:0]

VMID value for comparison. The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>.BT IN {'11xx'} and EL2 is implemented:

31 … 0
ContextID2

ContextID2, bits [31:0]

When FEAT_Debugv8p1 is implemented:

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing DBGBXVR<n>

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	m[3:0]	0b001

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
        ↪!= '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBXVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBXVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBXVR(m);
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	m[3:0]	0b001

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBXVR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBXVR(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBXVR(m) = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.5 DBGCLAIMCLR, Debug CLAIM Tag Clear register

The DBGCLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGCLAIMCLR are UNDEFINED.

Attributes

DBGCLAIMCLR is a 32-bit register.

Field descriptions

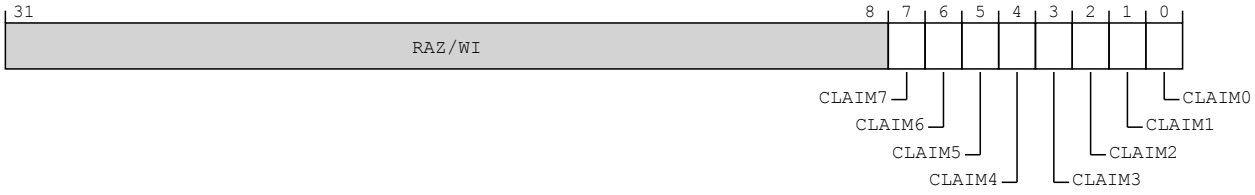

Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is W1C.

Accessing DBGCLAIMCLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1001	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGCLAIMCLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = DBGCLAIMCLR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGCLAIMCLR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1001	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGCLAIMCLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGCLAIMCLR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGCLAIMCLR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.6 DBGCLAIMSET, Debug CLAIM Tag Set register

The DBGCLAIMSET characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGCLAIMSET are UNDEFINED.

Attributes

DBGCLAIMSET is a 32-bit register.

Field descriptions

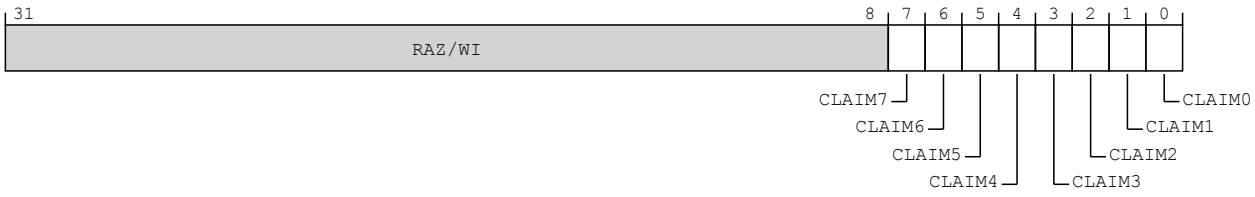

Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m>	Meaning
0b0	On a write: Ignored.
0b1	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/W1S.

Accessing DBGCLAIMSET

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1000	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGCLAIMSET();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGCLAIMSET();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGCLAIMSET();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1000	0b110

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGCLAIMSET() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGCLAIMSET() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGCLAIMSET() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.7 DBGDCCINT, DCC Interrupt Enable Register

The DBGDCCINT characteristics are:

Purpose

Enables interrupt requests to be signaled based on the DCC status flags.

Configuration

AArch32 System register DBGDCCINT bits [31:0] are architecturally mapped to AArch64 System register MDCCINT_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDCCINT are UNDEFINED.

Attributes

DBGDCCINT is a 32-bit register.

Field descriptions

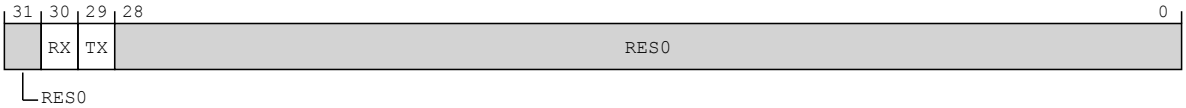

Bit [31]

Reserved, RES0.

RX, bit [30]

DCC interrupt request enable control for DTRRX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

RX	Meaning
`0b0`	No interrupt request generated by DTRRX.
`0b1`	Interrupt request will be generated on RXfull == 1.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

TX, bit [29]

DCC interrupt request enable control for DTRTX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TX	Meaning
0b0	No interrupt request generated by DTRTX.
0b1	Interrupt request will be generated on TXfull == 0.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bits [28:0]

Reserved, RES0.

Accessing DBGDCCINT

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDCCINT();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDCCINT();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDCCINT();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = DBGDCCINT();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    DBGDCCINT() = R(t);
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDCCINT() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDCCINT() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        DBGDCCINT() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.8 DBGDEVID, Debug Device ID register 0

The DBGDEVID characteristics are:

Purpose

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

Configuration

This register is required in all implementations.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDEVID are UNDEFINED.

Attributes

DBGDEVID is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
CIDMask	AuxRegs	DoubleLock	VirtExtns	VectorCatch	BPAddrMask	WPAddrMask	PCSample

CIDMask, bits [31:28]

Indicates the level of support for the Context ID matching breakpoint masking capability.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CIDMask	Meaning
0b0000	Context ID masking is not implemented.
0b0001	Context ID masking is implemented.

All other values are reserved. The value of this for Armv8 is 0b0000.

Access to this field is RO.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxRegs	Meaning
0b0000	None supported.
0b0001	Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DoubleLock, bits [23:20]

OS Double Lock implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DoubleLock	Meaning
0b0000	OS Double Lock is not implemented. DBGOSDLR is RAZ/WI.
0b0001	OS Double Lock is implemented. DBGOSDLR is RW.

FEAT_DoubleLock implements the functionality identified by the value 0b0001.

All other values are reserved.

Access to this field is RO.

VirtExtns, bits [19:16]

Indicates whether EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VirtExtns	Meaning
0b0000	EL2 is not implemented.
0b0001	EL2 is implemented.

All other values are reserved.

Access to this field is RO.

VectorCatch, bits [15:12]

Defines the form of Vector Catch exception implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VectorCatch	Meaning
0b0000	Address matching Vector Catch exception implemented.
0b0001	Exception matching Vector Catch exception implemented.

All other values are reserved.

Access to this field is RO.

BPAddrMask, bits [11:8]

Indicates the level of support for the instruction address matching breakpoint masking capability.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BPAddrMask	Meaning
0b0000	Breakpoint address masking might be implemented. If not implemented, DBGBCR<n>[28:24] is RAZ/WI.
0b0001	Breakpoint address masking is implemented.
0b1111	Breakpoint address masking is not implemented. DBGBCR<n>[28:24] is RES0.

All other values are reserved. The value of this for Armv8 is 0b1111.

Access to this field is RO.

WPAddrMask, bits [7:4]

Indicates the level of support for the data address matching watchpoint masking capability.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WPAddrMask	Meaning
0b0000	Watchpoint address masking might be implemented. If not implemented, DBGWCR<n>.MASK (Address mask) is RAZ/WI.
0b0001	Watchpoint address masking is implemented.
0b1111	Watchpoint address masking is not implemented. DBGWCR<n>.MASK (Address mask) is RES0.

All other values are reserved. The value of this for Armv8 is 0b0001.

Access to this field is RO.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the external debug registers space.
0b0010	Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.
0b0011	EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DBGDEVID

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0010	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDEVID();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDEVID();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDEVID();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.9 DBGDEVID1, Debug Device ID register 1

The DBGDEVID1 characteristics are:

Purpose

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

Configuration

This register is required in all implementations.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDEVID1 are UNDEFINED.

Attributes

DBGDEVID1 is a 32-bit register.

Field descriptions

31 … 4	3 … 0
RES0	PCSROffset

Bits [31:4]

Reserved, RES0.

PCSROffset, bits [3:0]

This field indicates the offset applied to PC samples returned by reads of EDPCSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSROffset	Meaning
0b0000	EDPCSR is not implemented.
0b0010	EDPCSR implemented. Samples have no offset applied and do not sample the instruction set state in AArch32 state.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Access to this field is RO.

Accessing DBGDEVID1

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0001	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDEVID1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDEVID1();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDEVID1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.10 DBGDEVID2, Debug Device ID register 2

The DBGDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

Configuration

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDEVID2 are UNDEFINED.

Attributes

DBGDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing DBGDEVID2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0000	0b111

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = DBGDEVID2();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDEVID2();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDEVID2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.11 DBGDIDR, Debug ID Register

The DBGDIDR characteristics are:

Purpose

Specifies which version of the Debug architecture is implemented, and some features of the debug implementation.

Configuration

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DBGDIDR are UNDEFINED.

Attributes

DBGDIDR is a 32-bit register.

Field descriptions

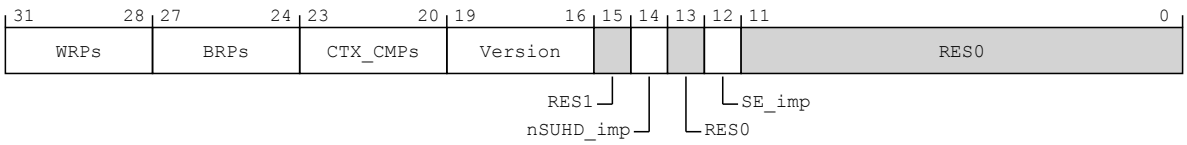

WRPs, bits [31:28]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
`0b0001..0b1111`	The number of watchpoints, minus 1.

If FEAT_Debugv8p9 is implemented and 16 or more watchpoints are implemented, this field reads as `0b1111`.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 watchpoints.

The value `0b0000` is reserved.

Access to this field is RO.

BRPs, bits [27:24]

Number of breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BRPs	Meaning
0b0001..0b1111	The number of breakpoints, minus 1.

If FEAT_Debugv8p9 is implemented and 16 or more breakpoints are implemented, this field reads as 0b1111.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 breakpoints.

The value 0b0000 is reserved.

Access to this field is RO.

CTX_CMPs, bits [23:20]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
0b0000..0b1111	The number of context-aware breakpoints, minus 1.

The value of this field is never greater than DBGDIDR.BRPs.

If FEAT_Debugv8p9 is implemented and 16 or more context-aware breakpoints are implemented, this field reads as 0b1111.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 breakpoints.

Access to this field is RO.

Version, bits [19:16]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Version	Meaning
0b0000	Not supported.
0b0001	Armv6, v6 Debug architecture, with System registers access.
0b0010	Armv6, v6.1 Debug architecture, with System registers access.
0b0011	Armv7, v7 Debug architecture, with only baseline System registers.
0b0100	Armv7, v7 Debug architecture, with all System registers implemented.
0b0101	Armv7, v7.1 Debug architecture, with System registers access.
0b0110	Armv8.0 debug architecture.
0b0111	Armv8.1 debug architecture, FEAT_Debugv8p1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Version	Meaning
0b1000	Armv8.2 debug architecture, FEAT_Debugv8p2.
0b1001	Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010	Armv8.8 debug architecture, FEAT_Debugv8p8.
0b1011	Armv8.9 debug architecture, FEAT_Debugv8p9.

All other values are reserved.

From Armv8.0, the values 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted.

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

FEAT_Debugv8p8 implements the functionality identified by the value 0b1010.

FEAT_Debugv8p9 implements the functionality identified by the value 0b1011.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

From Armv8.8, the value 0b1001 is not permitted.

From Armv8.9, the value 0b1010 is not permitted.

Access to this field is RO.

Bit [15]

Reserved, RES1.

nSUHD_imp, bit [14]

Previously indicated that Secure User Halting Debug is not implemented.

The value of this field must match the value of the SE_imp field.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [13]

Reserved, RES0.

SE_imp, bit [12]

EL3 implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SE_imp	Meaning
0b0	EL3 not implemented.
0b1	EL3 implemented.

The value of this field must match the value of the nSUHD_imp field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

Bits [11:0]

Reserved, RES0.

Accessing DBGDIDR

Arm deprecates any access to this register from EL0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDIDR();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x05);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDIDR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDIDR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDIDR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDIDR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.12 DBGDRAR, Debug ROM Address Register

The DBGDRAR characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Use of this register is deprecated.

Configuration

DBGDRAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

AArch32 System register DBGDRAR bits [63:0] are architecturally mapped to AArch64 System register MDRAR_EL1[63:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DBGDRAR are UNDEFINED.

Attributes

DBGDRAR is a 64-bit register.

Field descriptions

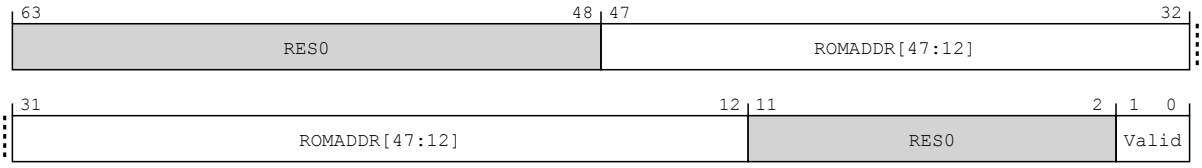

Bits [63:48]

Reserved, RES0.

ROMADDR[47:12], bits [47:12]

Bits[47:12] of the ROM table physical address.

If the physical address size in bits (PAsize) is less than 48 then the register bits corresponding to ROMADDR [47:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

If DBGDRAR.Valid == `0b00`, then this field is UNKNOWN.

In an implementation that includes EL3, ROMADDR is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

Bits [11:2]

Reserved, RES0.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Valid	Meaning
0b00	ROM Table address is not valid. Software must ignore ROMADDR.
0b11	ROM Table address is valid.

Other values are reserved.

Arm recommends implementations set this field to zero.

Accessing DBGDRAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDRAR()[31:0];
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x05);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDRA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDRA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        R(t) = DBGDRAR()[31:0];
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDRA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDRA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDRAR()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDRAR()[31:0];
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDRAR()[31:0];
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1110	0b0001	0b0000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t, t2) = DBGDRAR();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL1, 0x0C);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDRA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDRA] != '00') then
        AArch32_TakeHypTrapException(0x0C);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x0C);
        end;
    else
        R(t, t2) = DBGDRAR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDRA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDRA]
        ↪!= '00' then
        AArch32_TakeHypTrapException(0x0C);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x0C);
        end;
    else
        R(t, t2) = DBGDRAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x0C);
        end;
    else
        R(t, t2) = DBGDRAR();
    end;
elsif PSTATE.EL == EL3 then
    R(t, t2) = DBGDRAR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.13 DBGDSAR, Debug Self Address Register

The DBGDSAR characteristics are:

Purpose

In earlier versions of the Arm Architecture, this register defines the offset from the base address defined in DBGDRAR of the physical base address of the debug registers for the PE. Use of this register is deprecated.

Configuration

DBGDSAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DBGDSAR are UNDEFINED.

Attributes

DBGDSAR is a 64-bit register.

Field descriptions

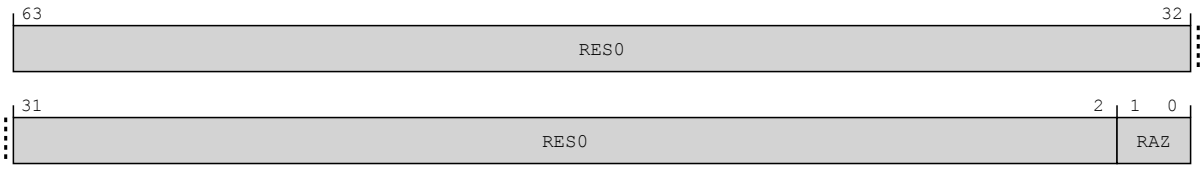

Bits [63:2]

Reserved, RES0.

Bits [1:0]

Reserved, RAZ.

This field indicates whether the debug self address offset is valid. For ARMv8, this field is always `0b00`, the offset is not valid.

Accessing DBGDSAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0010	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDSAR()[31:0];
elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x05);
            else
                AArch64_AArch32SystemAccessTrap(EL1, 0x05);
            end;
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x05);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
            ↪== '1' || MDCR_EL2().[TDE,TDRA] != '00') then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
            ↪'1' || HDCR().[TDE,TDRA] != '00') then
            AArch32_TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            R(t) = DBGDSAR()[31:0];
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
            ↪MDCR_EL2().[TDE,TDRA] != '00' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDRA]
            ↪!= '00' then
            AArch32_TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            R(t) = DBGDSAR()[31:0];
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        R(t) = DBGDSAR()[31:0];
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDSAR()[31:0];
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1110	0b0010	0b0000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t, t2) = DBGDSAR();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x0C);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDRA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDRA] != '00') then
        AArch32_TakeHypTrapException(0x0C);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x0C);
        end;
    else
        R(t, t2) = DBGDSAR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDRA] != '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x0C);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDRA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x0C);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x0C);
        end;
    else
        R(t, t2) = DBGDSAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x0C);
        end;
    else
        R(t, t2) = DBGDSAR();
    end;
elsif PSTATE.EL == EL3 then
    R(t, t2) = DBGDSAR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.14 DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

This register is required in all implementations.

AArch32 System register DBGDSCRext bits [31:0] are architecturally mapped to AArch64 System register MDSCR_EL1[31:0].

AArch32 System register DBGDSCRext bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to External register EDSCR[31:29, 27:26, 23:21, 19, 14, 6].

AArch32 System register DBGDSCRext bit [15] is architecturally mapped to AArch32 System register DBGDSCRint[15].

AArch32 System register DBGDSCRext bit [12] is architecturally mapped to AArch32 System register DBGDSCRint[12].

AArch32 System register DBGDSCRext bits [5:2] are architecturally mapped to AArch32 System register DBGDSCRint[5:2].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDSCRext are UNDEFINED.

Attributes

DBGDSCRext is a 32-bit register.

Field descriptions

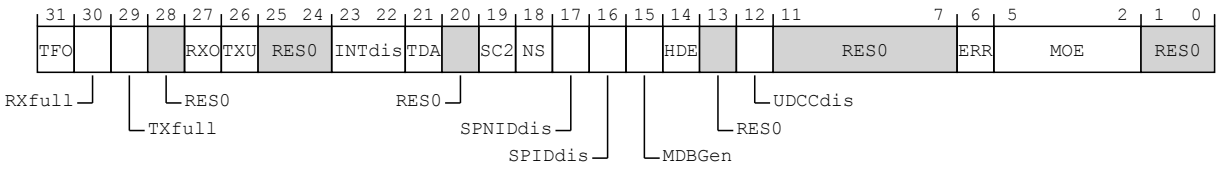

TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter override. Used for save/restore of EDSCR.TFO.

When the OS Lock is unlocked, DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When the OS Lock is locked, DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RXfull, bit [30]

DTRRX full. Used for save/restore of EDSCR.RXfull.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

Arm deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRRX full status.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

TXfull, bit [29]

DTRTX full. Used for save/restore of EDSCR.TXfull.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

Arm deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRTX full status.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

When DBGOSLSR.OSLK == 1, if bits [27,6] of the value written to DBGDSCRext are {1,0}, that is, the RXO bit is 1 and the ERR bit is 0, the PE sets EDSCR.{RXO,ERR} to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

When DBGOSLSR.OSLK == 1, if bits [26,6] of the value written to DBGDSCRext are {1,0}, that is, the TXU bit is 1 and the ERR bit is 0, the PE sets EDSCR.{TXU,ERR} to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == '1', access to this field is RW.
- When DBGOSLSR.OSLK == '0', access to this field is RO.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of EDSCR.INTdis.

When DBGOSLSR.OSLK == 0, software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == '1', access to this field is RW.
- When DBGOSLSR.OSLK == '0', access to this field is RO.

TDA, bit [21]

Used for save/restore of EDSCR.TDA.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == '1', access to this field is RW.
- When DBGOSLSR.OSLK == '0', access to this field is RO.

Bit [20]

Reserved, RES0.

SC2, bit [19]

When FEAT_PCSRv8 is implemented, FEAT_VHE is implemented, and FEAT_PCSRv8p2 is not implemented:

Used for save/restore of EDSCR.SC2.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.SC2. Reads and writes of this bit are indirect accesses to EDSCR.SC2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

Otherwise:

Reserved, RES0.

NS, bit [18]

Non-secure status.

Arm deprecates use of this field.

NS	Meaning
`0b0`	Secure state.
`0b1`	Non-secure state.

Access to this field is RO.

SPNIDdis, bit [17]

When EL3 is implemented:

Secure privileged profiling disabled status bit.

SPNIDdis	Meaning
`0b0`	Profiling allowed in Secure privileged modes.
`0b1`	Profiling prohibited in Secure privileged modes.

This field reads as 0 if any of the following applies, and reads as 1 otherwise:

- FEAT_Debugv8p2 is not implemented and `ExternalSecureNoninvasiveDebugEnabled()` returns TRUE.
- EL3 is using AArch32 and the value of SDCR.SPME is 1.
- EL3 is using AArch64 and the value of MDCR_EL3.SPME is 1.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

SPIDdis, bit [16]

When EL3 is implemented:

Secure privileged AArch32 invasive self-hosted debug disabled status bit. The value of this bit depends on the value of SDCR.SPD and the pseudocode function `AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SPIDdis	Meaning
0b0	Self-hosted debug enabled in Secure privileged AArch32 modes.
0b1	Self-hosted debug disabled in Secure privileged AArch32 modes.

This bit reads as 1 if any of the following is true and reads as 0 otherwise:

- EL3 is using AArch32 and SDCR.SPD has the value 0b10.
- EL3 is using AArch64 and MDCR_EL3.SPD32 has the value 0b10.
- EL3 is using AArch32, SDCR.SPD has the value 0b00, and AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled() returns FALSE.
- EL3 is using AArch64, MDCR_EL3.SPD32 has the value 0b00, and AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled() returns FALSE.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

MDBGen, bit [15]

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

MDBGen	Meaning
0b0	Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
0b1	Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

Bit [13]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UDCCdis, bit [12]

Traps EL0 accesses to the DCC registers to Undefined mode.

UDCCdis	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, and DBGDRAR are trapped to Undefined mode.

Note

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.ERR. Reads and writes of this bit are indirect accesses to EDSCR.ERR.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == ‘1’, access to this field is RW.
- When DBGOSLSR.OSLK == ‘0’, access to this field is RO.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

MOE	Meaning
`0b0001`	Breakpoint.
`0b0011`	Software breakpoint (BKPT) instruction.
`0b0101`	Vector catch.
`0b1010`	Watchpoint.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGDSCRext

Individual fields within this register might have restricted accessibility when the OS Lock is unlocked, DBGOSLSR.OSLK == 0. See the field descriptions for more detail.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDSCRext();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDSCRext();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGDSCRext();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDSCRext() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDSCRext() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGDSCRext() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.15 DBGDSCRint, Debug Status and Control Register, Internal View

The DBGDSCRint characteristics are:

Purpose

Main control register for the debug implementation. This is an internal, read-only view.

Configuration

This register is required in all implementations.

DBGDSCRint.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} are UNKNOWN when the register is accessed at EL0. However, although these values are not accessible at EL0 by instructions that are neither UNPREDICTABLE nor return UNKNOWN values, it is permissible for an implementation to return the values of DBGDSCRext.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} for these fields at EL0.

It is also permissible for an implementation to return the same values as defined for a read of DBGDSCRint at EL1 or above. (This is the case even if the implementation does not support AArch32 at EL1 or above.)

AArch32 System register DBGDSCRint bits [30:29] are architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

AArch32 System register DBGDSCRint bits [30:29] are architecturally mapped to External register EDSCR[30:29].

AArch32 System register DBGDSCRint bit [15] is architecturally mapped to AArch64 System register MDSCR_EL1[15].

AArch32 System register DBGDSCRint bit [12] is architecturally mapped to AArch64 System register MDSCR_EL1[12].

AArch32 System register DBGDSCRint bits [5:2] are architecturally mapped to AArch64 System register MDSCR_EL1[5:2].

AArch32 System register DBGDSCRint bit [15] is architecturally mapped to AArch32 System register DBGDSCRext[15].

AArch32 System register DBGDSCRint bit [12] is architecturally mapped to AArch32 System register DBGDSCRext[12].

AArch32 System register DBGDSCRint bits [5:2] are architecturally mapped to AArch32 System register DBGDSCRext[5:2].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DBGDSCRint are UNDEFINED.

Attributes

DBGDSCRint is a 32-bit register.

Field descriptions

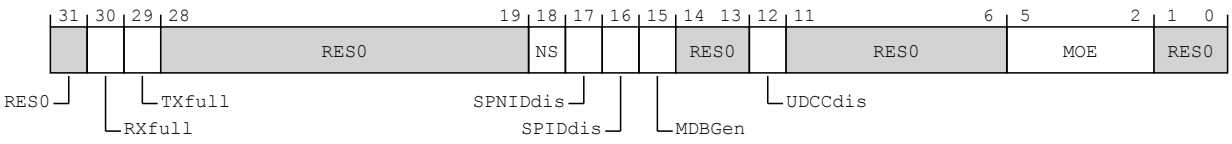

Bit [31]

Reserved, RES0.

RXfull, bit [30]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

TXfull, bit [29]

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

Bits [28:19]

Reserved, RES0.

NS, bit [18]

Non-secure status.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

SPNIDdis, bit [17]

Secure privileged non-invasive debug disable.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

SPIDdis, bit [16]

Secure privileged invasive debug disable.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

MDBGen, bit [15]

Monitor debug events enable.

Read-only view of the equivalent bit in the DBGDSCRext.

Bits [14:13]

Reserved, RES0.

UDCCdis, bit [12]

User mode access to Debug Communications Channel disable.

Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

Bits [11:6]

Reserved, RES0.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

MOE	Meaning
`0b0001`	Breakpoint
`0b0011`	Software breakpoint (BKPT) instruction
`0b0101`	Vector catch
`0b1010`	Watchpoint

Read-only view of the equivalent bit in the DBGDSCRext.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [1:0]

Reserved, RES0.

Accessing DBGDSCRint

When <Rt> is APSR_nzcv, encoded as R15, then instead of reading the entire register, the access copies DBGDSCRint[31:28] into the PSTATE NZCV flags.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    if t == 15 then
        ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
    else
        R(t) = DBGDSCRint();
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x05);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
         ↪IsFeatureImplemented(FEAT_FGT) && HDCR().TDCC == '1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
         ↪== '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
             ↪'1' || HDCR().[TDE,TDA] != '00') then
            AArch32_TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch32_TakeMonitorTrapException();
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            if t == 15 then
                if Halted() then
                    ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
                else
                    PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
                end;
            else
                R(t) = DBGDSCRint();
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
             ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪MDCR_EL2().[TDE,TDA] != '00' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
             ↪!= '00' then
            AArch32_TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL3SDDUndef() then
                Undefined();
            else
                AArch32_TakeMonitorTrapException();
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            if t == 15 then
                if Halted() then
                    ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
                else
                    PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
                end;
            else
                R(t) = DBGDSCRint();
            end;
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
             ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch32_TakeMonitorTrapException();
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            if t == 15 then
                if Halted() then
                    ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
                else
                    PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
                end;
            else
                R(t) = DBGDSCRint();
            end;
        end;
    elsif PSTATE.EL == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        if t == 15 then
            if Halted() then
                ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
            else
                PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
            end;
        else
            R(t) = DBGDSCRint();
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.16 DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View

The DBGDTRRXext characteristics are:

Purpose

Used for save/restore of DBGDTRRXint. It is a component of the Debug Communications Channel.

Configuration

AArch32 System register DBGDTRRXext bits [31:0] are architecturally mapped to AArch64 System register OSDTRRX_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDTRRXext are UNDEFINED.

Attributes

DBGDTRRXext is a 32-bit register.

Field descriptions

31	0
Update DTRRX without side-effect	

DTRRX, bits [31:0]

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see ‘The Debug Communication Channel and Instruction Transfer Register’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRXext

Arm deprecates reads and writes of DBGDTRRXext through the System register interface when the OS Lock is unlocked, DBGOSLSR.OSLK == 0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t) = DBGDTRRXext();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
             ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪MDCR_EL2().[TDE,TDA] != '00' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
             ↪!= '00' then
            AArch32_TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch32_TakeMonitorTrapException();
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            R(t) = DBGDTRRXext();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
             ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDTRRXext();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = DBGDTRRXext();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    DBGDTRRXext() = R(t);
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDTRRXext() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDTRRXext() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        DBGDTRRXext() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.17 DBGDTRRXint, Debug Data Transfer Register, Receive

The DBGDTRRXint characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configuration

AArch32 System register DBGDTRRXint bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

AArch32 System register DBGDTRRXint bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DBGDTRRXint are UNDEFINED.

Attributes

DBGDTRRXint is a 32-bit register.

Field descriptions

31	0
Update DTRRX	

DTRRX, bits [31:0]

Update DTRRX.

Reads of this register:

- If RXfull is 1, return the last value written to DTRRX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRXint

Data can be stored to memory from this register using STC.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() then
    R(t) = Read_DBGDTR_EL0{32}();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x05);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪IsFeatureImplemented(FEAT_FGT) && HDCR().TDCC == '1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    else
        R(t) = Read_DBGDTR_EL0{32}();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
        ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    else
        R(t) = Read_DBGDTR_EL0{32}();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    else
        R(t) = Read_DBGDTR_EL0{32}();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = Read_DBGDTR_EL0{32}();
    end;
end;
```

```
STC{<c>}{<q>} <coproc>, <CRd>, <addressing_mode>
```

coproc	CRd
0b1110	0b0101

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() then
    MemA{32}(address) = Read_DBGDTR_EL0{32}();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x06);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
             ↪IsFeatureImplemented(FEAT_FGT) && HDCR().TDCC == '1' then
            AArch32_TakeHypTrapException(0x06);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
             ↪== '1' || MDCR_EL2().[TDE,TDA] != '00') then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
             ↪'1' || HDCR().[TDE,TDA] != '00') then
            AArch32_TakeHypTrapException(0x06);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL3, 0x06);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            AArch32_TakeMonitorTrapException();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL3, 0x06);
        else
            MemA{32}(address) = Read_DBGDTR_EL0{32}();
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x06);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪MDCR_EL2().[TDE,TDA] != '00' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
             ↪!= '00' then
            AArch32_TakeHypTrapException(0x06);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL3, 0x06);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            AArch32_TakeMonitorTrapException();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL3, 0x06);
        else
            MemA{32}(address) = Read_DBGDTR_EL0{32}();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL3, 0x06);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            AArch32_TakeMonitorTrapException();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL3, 0x06);
        else
            MemA{32}(address) = Read_DBGDTR_EL0{32}();
        end;
    elsif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
            AArch32_TakeMonitorTrapException();
        else
            MemA{32}(address) = Read_DBGDTR_EL0{32}();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.18 DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit

The DBGDTRTXext characteristics are:

Purpose

Used for save/restore of DBGDTRTXint. It is a component of the Debug Communication Channel.

Configuration

AArch32 System register DBGDTRTXext bits [31:0] are architecturally mapped to AArch64 System register OSDTRTX_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGDTRTXext are UNDEFINED.

Attributes

DBGDTRTXext is a 32-bit register.

Field descriptions

31	0
Return DTRTX without side-effect	

DTRTX, bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see ‘The Debug Communication Channel and Instruction Transfer Register’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTXext

Arm deprecates reads and writes of DBGDTRTXext through the System register interface when the OS Lock is unlocked, DBGOSLSR.OSLK == 0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t) = DBGDTRTXext();
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
             ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪MDCR_EL2().[TDE,TDA] != '00' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
             ↪!= '00' then
            AArch32_TakeHypTrapException(0x05);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch32_TakeMonitorTrapException();
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        else
            R(t) = DBGDTRTXext();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
             ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
            Undefined();
        elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
             ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x05);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
             ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGDTRTXext();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = DBGDTRTXext();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0011	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    DBGDTRTXext() = R(t);
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDTRTXext() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TDCC == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
         ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGDTRTXext() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        DBGDTRTXext() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.19 DBGDTRTXint, Debug Data Transfer Register, Transmit

The DBGDTRTXint characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configuration

AArch32 System register DBGDTRTXint bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

AArch32 System register DBGDTRTXint bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DBGDTRTXint are UNDEFINED.

Attributes

DBGDTRTXint is a 32-bit register.

Field descriptions

31	0
Return DTRTX	

DTRTX, bits [31:0]

DTRTX. Writes to this register:

- If TXfull is 1, DTRTX is set to an UNKNOWN value.
- If TXfull is 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTXint

Data can be loaded from memory into this register using 'LDC (immediate)' and 'LDC (literal)'.

Accesses to this register use the following encodings in the System register encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0101	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() then
    Write_DBGDTR_EL0{32}(R(t));
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x05);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x05);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪IsFeatureImplemented(FEAT_FGT) && HDCR().TDCC == '1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    else
        Write_DBGDTR_EL0{32}(R(t));
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
        ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    else
        Write_DBGDTR_EL0{32}(R(t));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x05);
    else
        Write_DBGDTR_EL0{32}(R(t));
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        Write_DBGDTR_EL0{32}(R(t));
    end;
end;
```

LDC{<c>}{<q>} <coproc>, <CRd>, <addressing_mode>

coproc	CRd
0b1110	0b0101

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif Halted() then
    Write_DBGDTR_EL0{32}(MemA{32}(address));
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && MDSCR_EL1().TDCC == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x06);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x06);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x06);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪IsFeatureImplemented(FEAT_FGT) && HDCR().TDCC == '1' then
        AArch32_TakeHypTrapException(0x06);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && (HCR_EL2().TGE
        ↪== '1' || MDCR_EL2().[TDE,TDA] != '00') then
        AArch64_AArch32SystemAccessTrap(EL2, 0x06);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && (HCR().TGE ==
        ↪'1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x06);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x06);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x06);
    else
        Write_DBGDTR_EL0{32}(MemA{32}(address));
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL2().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x06);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TDCC ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x06);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x06);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
        ↪!= '00' then
        AArch32_TakeHypTrapException(0x06);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x06);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x06);
    else
        Write_DBGDTR_EL0{32}(MemA{32}(address));
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsFeatureImplemented(FEAT_FGT) && MDCR_EL3().TDCC == '1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x06);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TDCC == '1'
        ↪then
        AArch32_TakeMonitorTrapException();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x06);
    else
        Write_DBGDTR_EL0{32}(MemA{32}(address));
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TDCC == '1' then
        AArch32_TakeMonitorTrapException();
    else
        Write_DBGDTR_EL0{32}(MemA{32}(address));
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.20 DBGOSDLR, Debug OS Double Lock Register

The DBGOSDLR characteristics are:

Purpose

Locks out the external debug interface.

Configuration

AArch32 System register DBGOSDLR bits [31:0] are architecturally mapped to AArch64 System register OSDLR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGOSDLR are UNDEFINED.

Attributes

DBGOSDLR is a 32-bit register.

Field descriptions

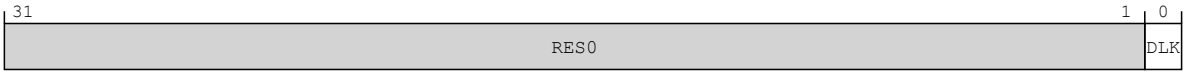

Bits [31:1]

Reserved, RES0.

DLK, bit [0]

When FEAT_DoubleLock is implemented:

OS Double Lock control bit.

DLK	Meaning
0b0	OS Double Lock unlocked.
0b1	OS Double Lock locked, if DBGPRCR.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

Accessing DBGOSDLR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL2.TDOSA")) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪HDCR().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped
         ↪by HDCR.TDOSA")) then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' && (IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA"))
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGOSDLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' && (IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA"))
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGOSDLR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGOSDLR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL2.TDOSA")) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪HDCR().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped
         ↪by HDCR.TDOSA")) then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' && (IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA"))
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGOSDLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) ||
         ↪ImpDefBool("Trapped by MDCR_EL3.TDOSA")) then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' && (IsFeatureImplemented(FEAT_DoubleLock) || ImpDefBool("Trapped by MDCR_EL3.TDOSA"))
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGOSDLR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGOSDLR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.21 DBGOSECCR, Debug OS Lock Exception Catch Control Register

The DBGOSECCR characteristics are:

Purpose

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

Configuration

If DBGOSLSR.OSLK == 0 then DBGOSECCR returns an UNKNOWN value on reads and ignores writes.

AArch32 System register DBGOSECCR bits [31:0] are architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

AArch32 System register DBGOSECCR bits [31:0] are architecturally mapped to External register EDECCR[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGOSECCR are UNDEFINED.

Attributes

DBGOSECCR is a 32-bit register.

Field descriptions

When DBGOSLSR.OSLK == '1':

31	0
EDECCR	

EDECCR, bits [31:0]

Used for save/restore to EDECCR over powerdown.

Reads or writes to this field are indirect accesses to EDECCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to the expression `0x00`.

Accessing DBGOSECCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = DBGOSECCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = DBGOSECCR();
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = DBGOSECCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' then
        return;
    else
        DBGOSECCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' then
        return;
    else
        DBGOSECCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' then
        return;
    else
        DBGOSECCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.22 DBGOSLAR, Debug OS Lock Access Register

The DBGOSLAR characteristics are:

Purpose

Provides a lock for the debug registers. The OS Lock also disables some debug exceptions and debug events.

Configuration

The OS Lock can also be locked or unlocked using the AArch64 System register OSLAR_EL1 and External register OSLAR_EL1.

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGOSLAR are UNDEFINED.

Attributes

DBGOSLAR is a 32-bit register.

Field descriptions

31	0
OSLA	

OSLA, bits [31:0]

OS Lock Access. Writing the value `0xC5ACCE55` to the DBGOSLAR sets the OS Lock to 1. Writing any other value sets the OS Lock to 0.

Use DBGOSLSR.OSLK to check the current status of the lock.

Accessing DBGOSLAR

Accesses to this register use the following encodings in the System register encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0000	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGOSLAR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGOSLAR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGOSLAR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.23 DBGOSLSR, Debug OS Lock Status Register

The DBGOSLSR characteristics are:

Purpose

Provides status information for the OS Lock.

Configuration

The OS Lock status is also visible in the external debug interface through EDPRSR.

AArch32 System register DBGOSLSR bits [31:0] are architecturally mapped to AArch64 System register OSLSR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGOSLSR are UNDEFINED.

Attributes

DBGOSLSR is a 32-bit register.

Field descriptions

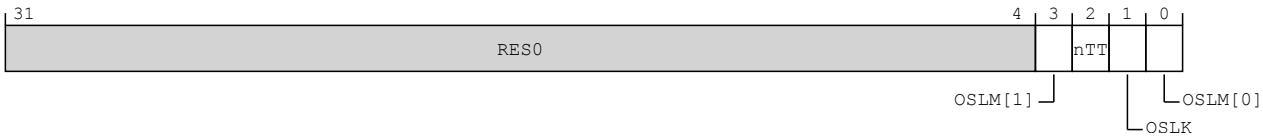

Bits [31:4]

Reserved, RES0.

OSLM, bits [3, 0]

OS Lock model implemented. Identifies the form of OS save and restore mechanism implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b00	OS Lock not implemented.
0b10	OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value 0b00 is not permitted.

Access to this field is RO.

nTT, bit [2]

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLK, bit [1]

OS Lock Status.

OSLK	Meaning
0b0	OS Lock unlocked.
0b1	OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'1'`.

Accessing DBGOSLSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGOSLSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGOSLSR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGOSLSR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.24 DBGPRCR, Debug Power Control Register

The DBGPRCR characteristics are:

Purpose

Controls behavior of the PE on powerdown request.

Configuration

AArch32 System register DBGPRCR bits [31:0] are architecturally mapped to AArch64 System register DBGPRCR_EL1[31:0].

AArch32 System register DBGPRCR bit [0] is architecturally mapped to External register EDPRCR[0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGPRCR are UNDEFINED.

Attributes

DBGPRCR is a 32-bit register.

Field descriptions

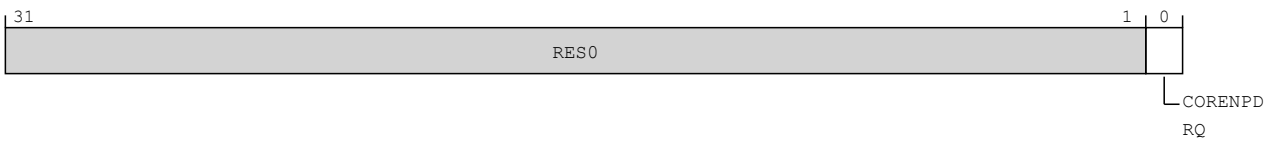

Bits [31:1]

Reserved, RES0.

CORENPDRQ, bit [0]

When FEAT_DoPD is implemented:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see ‘Core power domain power states’.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is set to an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.

Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.

Accessing DBGPRCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGPRCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGPRCR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGPRCR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDOSA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
         ↪HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGPRCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDOSA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDOSA
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGPRCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGPRCR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.25 DBGVCR, Debug Vector Catch Register

The DBGVCR characteristics are:

Purpose

Controls Vector Catch debug events.

Configuration

This register is required in all implementations.

AArch32 System register DBGVCR bits [31:0] are architecturally mapped to AArch64 System register DBGVCR32_EL2[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGVCR are UNDEFINED.

Attributes

DBGVCR is a 32-bit register.

Field descriptions

When EL3 is implemented and EL3 is using AArch32:

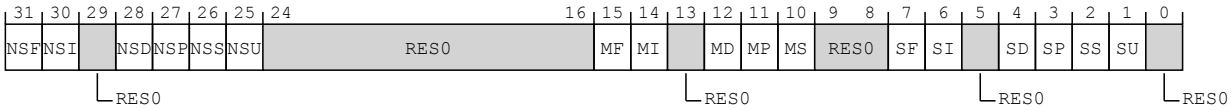

NSF, bit [31]

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSI, bit [30]

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

NSD, bit [28]

Data Abort exception vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSP, bit [27]

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSS, bit [26]

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSU, bit [25]

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:16]

Reserved, RES0.

MF, bit [15]

FIQ vector catch enable in Monitor mode.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MI, bit [14]

IRQ vector catch enable in Monitor mode.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

MD, bit [12]

Data Abort exception vector catch enable in Monitor mode.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MP, bit [11]

Prefetch Abort vector catch enable in Monitor mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

MS, bit [10]

Secure Monitor Call (SMC) vector catch enable in Monitor mode.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort exception vector catch enable in Secure state.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

When EL3 is implemented and EL3 is using AArch64:

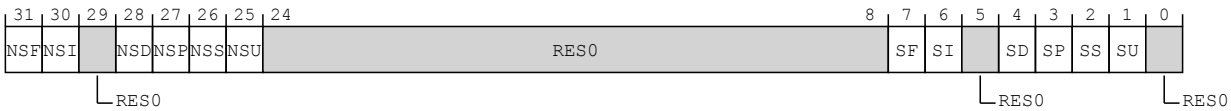

NSF, bit [31]

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSI, bit [30]

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

NSD, bit [28]

Data Abort exception vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSP, bit [27]

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSS, bit [26]

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSU, bit [25]

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort exception vector catch enable in Secure state.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

When EL3 is not implemented:

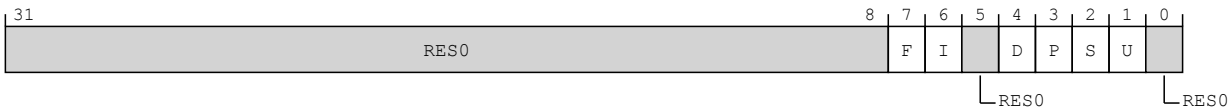

Bits [31:8]

Reserved, RES0.

F, bit [7]

FIQ vector catch enable.

The exception vector offset is `0x1C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [6]

IRQ vector catch enable.

The exception vector offset is `0x18`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

D, bit [4]

Data Abort exception vector catch enable.

The exception vector offset is `0x10`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P, bit [3]

Prefetch Abort vector catch enable.

The exception vector offset `0x0C`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [2]

Supervisor Call (SVC) vector catch enable.

The exception vector offset is `0x08`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]

Undefined Instruction vector catch enable.

The exception vector offset is `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing DBGVCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
        ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        R(t) = DBGVCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGVCR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGVCR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0111	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGVCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGVCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGVCR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.26 DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n> characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>.

Configuration

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGWCR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

AArch32 System register DBGWCR<n> bits [31:0] are architecturally mapped to External register DBGWCR<n>_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGWCR<n> are UNDEFINED.

Attributes

DBGWCR<n> is a 32-bit register.

Field descriptions

31 29	28 24	23 21	20	19 16	15 14	13	12 5	4 3	2 1	0
RES0	MASK	RES0	WT	LBN	SSC	HMC	BAS	LSC	PAC	E

When the E field is zero, all the other fields in the register are ignored.

Bits [31:29]

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only objects up to 2GB can be watched using a single mask.

MASK	Meaning
`0b00000`	No mask.
`0b00011..0b11111`	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

If programmed with a reserved value, the watchpoint behaves as if either:

- DBGWCR<n>.MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCR<n>.
- The watchpoint is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [23:21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked data address match.
0b1	Linked data address match.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number. For Linked data address watchpoints, this specifies the index of the context-matching breakpoint linked to.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information, see ‘Execution conditions for which a watchpoint generates Watchpoint exceptions’, and ‘Reserved DBGWCR<n>.{SSC, HMC, PAC} values’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see ‘Execution conditions for which a watchpoint generates Watchpoint exceptions’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or doubleword addressed by DBGWVR<n> is being watched.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS	Description
0bxxxxxxx1	Match byte at DBGWVR<n>
0bxxxxxx1x	Match byte at DBGWVR<n>+1
0bxxxxx1xx	Match byte at DBGWVR<n>+2
0bxxxx1xxx	Match byte at DBGWVR<n>+3

In cases where DBGWVR<n> addresses a doubleword:

BAS	Description, if DBGWVR<n>[2] == 0
0bxxx1xxxx	Match byte at DBGWVR<n>+4
0bxx1xxxxx	Match byte at DBGWVR<n>+5
0bx1xxxxxx	Match byte at DBGWVR<n>+6
0b1xxxxxxx	Match byte at DBGWVR<n>+7

If DBGWVR<n>[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. Arm deprecates setting DBGWVR<n>[2] == 1.

The valid values for BAS are nonzero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See 'Reserved DBGWCR<n>.BAS values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
0b01	Match instructions that load from a watchpointed address.
0b10	Match instructions that store to a watchpointed address.
0b11	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n. Possible values are:

E	Meaning
0b0	Watchpoint disabled.
0b1	Watchpoint enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b111

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_WATCHPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWCR(m);
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWCR(m);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWCR(m);
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b111

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_WATCHPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR(m) = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.27 DBGWFAR, Debug Watchpoint Fault Address Register

The DBGWFAR characteristics are:

Purpose

Previously returned information about the address of the instruction that accessed a watchpointed address. Is now deprecated and RES0.

Configuration

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGWFAR are UNDEFINED.

Attributes

DBGWFAR is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing DBGWFAR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
        ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGWFAR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        R(t) = DBGWFAR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DBGWFAR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGWFAR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    else
        DBGWFAR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DBGWFAR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.28 DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n> characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>.

Configuration

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGWVR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGWVR<n>_EL1[31:0].

AArch32 System register DBGWVR<n> bits [31:0] are architecturally mapped to External register DBGWVR<n>_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DBGWVR<n> are UNDEFINED.

Attributes

DBGWVR<n> is a 32-bit register.

Field descriptions

31 … 2	1 … 0
VA	RES0

VA, bits [31:2]

Bits[31:2] of the address value for comparison.

Arm deprecates setting DBGWVR<n>[2] == 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_WATCHPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWVR(m);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWVR(m);
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b110

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[3:0]);

if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif m >= NUM_WATCHPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().[TDE,TDA]
         ↪!= '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x05);
        end;
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR(m) = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.29 DLR, Debug Link Register

The DLR characteristics are:

Purpose

In Debug state, holds the address to restart from.

Configuration

AArch32 System register DLR bits [31:0] are architecturally mapped to AArch64 System register DLR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DLR are UNDEFINED.

Attributes

DLR is a 32-bit register.

Field descriptions

31	0
Restart address	

ADDR, bits [31:0]

Restart address.

Accessing DLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    R(t) = DLR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    DLR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.30 DSPSR, Debug Saved Program Status Register

The DSPSR characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configuration

AArch32 System register DSPSR bits [31:0] are architecturally mapped to AArch64 System register DSPSR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to DSPSR are UNDEFINED.

Attributes

DSPSR is a 32-bit register.

Field descriptions

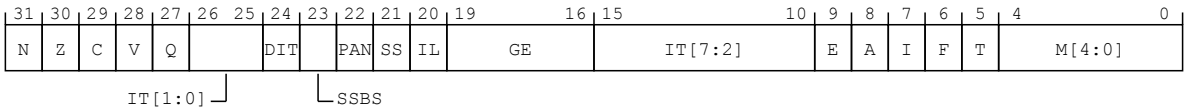

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on entering Debug state, and copied to PSTATE.Q on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on entering Debug state, and copied to PSTATE.IT on exiting Debug state.

On exiting Debug state, DSPSR.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on entering Debug state, and copied to PSTATE.GE on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on entering Debug state, and copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR.E is RES0. If the implementation does not support little-endian operation, DSPSR.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on entering Debug state, and copied to PSTATE.T on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on entering Debug state, and copied to PSTATE.M[4:0] on exiting Debug state.

M[4:0]	Meaning	Applies when
0b10000	User.	
0b10001	FIQ.	
0b10010	IRQ.	
0b10011	Supervisor.	
0b10110	Monitor.	FEAT_EL3 is implemented
0b10111	Abort.	
0b11010	Hyp.	FEAT_EL3 is implemented
0b11011	Undefined.	
0b11111	System.	

Other values are reserved. If DSPSR.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in ‘Illegal return events from AArch32 state’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DSPSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    R(t) = DSPSR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    DSPSR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.31 DSPSR2, Debug Saved Process State Register 2

The DSPSR2 characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configuration

When FEAT_Debugv8p9 is implemented, AArch32 System register DSPSR2 bits [31:0] are architecturally mapped to AArch64 System register DSPSR_EL0[63:32].

This register is present only when FEAT_Debugv8p9 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to DSPSR2 are UNDEFINED.

Attributes

DSPSR2 is a 32-bit register.

Field descriptions

Bits [31:5]

Reserved, RES0.

UINJ, bit [4]

When FEAT_UINJ is implemented:

Inject Undefined Instruction exception. Set to the value of PSTATE.UINJ on entering Debug state, and copied to PSTATE.UINJ on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [3:0]

Reserved, RES0.

Accessing DSPSR2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_Debugv8p9) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    R(t) = DSPSR2();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_Debugv8p9) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif !Halted() then
    Undefined();
else
    DSPSR2() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.32 HDCR, Hyp Debug Control Register

The HDCR characteristics are:

Purpose

Controls the trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors Extension.

Configuration

If EL2 is not implemented, this register is RES0 from EL3, and other than for a direct read of the register, the PE behaves as if HDCR.HPMN == PMCR.N.

AArch32 System register HDCR bits [31:0] are architecturally mapped to AArch64 System register MDCR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to HDCR are UNDEFINED.

Attributes

HDCR is a 32-bit register.

Field descriptions

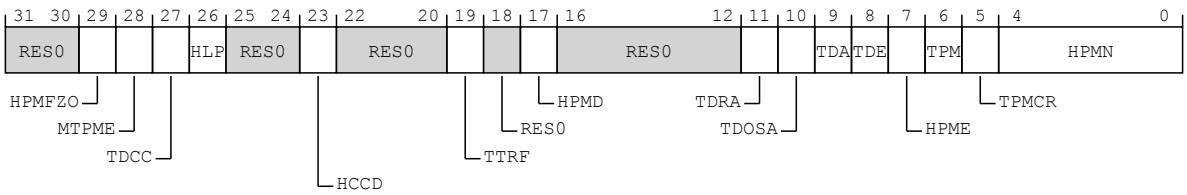

Bits [31:30]

Reserved, RES0.

HPMFZO, bit [29]

When FEAT_PMUv3p7 is implemented:

Hyp Performance Monitors Freeze-on-overflow. Stop event counters on overflow.

HPMFZO	Meaning
0b0	Do not freeze on overflow.
0b1	Event counters do not count when PMOVSR[(PMCR.N-1):HDCR.HPMN] is nonzero.

If HDCR.HPMN is less than PMCR.N, this field affects the operation of event counters in the range [HDCR.HPMN .. (PMCR.N-1)].

This field does not affect the operation of other event counters and PMCCNTR.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

MTPME, bit [28]

When FEAT_MTPMU is implemented and EL3 is not implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>.MT bits.

MTPME	Meaning
0b0	FEAT_MTPMU is disabled. The Effective value of PMEVTYPER<n>.MT is zero.
0b1	PMEVTYPER<n>.MT bits not affected by this bit.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0.

The reset behavior of this field is:

- On a Cold reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '1'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDCC, bit [27]

When FEAT_FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL1 and EL0 to EL2.

TDCC	Meaning
0b0	This control does not cause any register accesses to be trapped.
0b1	If EL2 is implemented and enabled in the current Security state, accesses to the DCC System registers at EL1 and EL0 generate a Hyp Trap exception, unless the access also generates a higher priority exception. Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC System registers trapped by this control are:

- DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- 0x05 for trapped MRC and MCR accesses with coproc == 0b1110.
- 0x06 for trapped LDC to DBGDTRTXint and STC from DBGDTRRXint.

When the PE is in Debug state, HDCR.TDCC does not trap any accesses to:

- DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HLP, bit [26]

When FEAT_PMUv3p5 is implemented:

Hypervisor Long event counter enable. Determines when unsigned overflow is recorded by an event counter overflow bit.

HLP	Meaning
`0b0`	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[31:0].
`0b1`	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[63:0].

If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this bit is read/write or RAZ/WI.

If HDCR.HPMN is less than PMCR.N, this bit affects the operation of event counters in the range [HDCR.HPMN..(PMCR.N-1)].

This field does not affect the operation of other event counters.

The operation of this field applies even when EL2 is disabled in the current Security state.

Note

PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [25:24]

Reserved, RES0.

HCCD, bit [23]

When FEAT_PMUv3p5 is implemented:

Hypervisor Cycle Counter Disable. Prohibits PMCCNTR from counting at EL2.

HCCD	Meaning
`0b0`	Cycle counting by PMCCNTR is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR is prohibited at EL2.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [22:20]

Reserved, RES0.

TTRF, bit [19]

When FEAT_TRF is implemented:

Traps use of the Trace Filter Control registers at EL1 to EL2 for MRC or MCR accesses, reported using EC syndrome value `0x03`.

TTRF	Meaning
`0b0`	Accesses to TRFCR at EL1 are not affected by this control bit.
`0b1`	Accesses to TRFCR at EL1 generate a Hyp Trap exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

When FEAT_PMUv3p1 is implemented and FEAT_Debugv8p2 is implemented:

Guest Performance Monitors Disable. Controls PMU operation in Hyp mode.

HPMD	Meaning
`0b0`	Counters are not affected by this mechanism.
`0b1`	Affected counters are prohibited from counting in Hyp mode. If PMCR.DP is 1, then PMCCNTR is disabled in Hyp mode. Otherwise, PMCCNTR is not affected by this mechanism.

The counters affected by this field are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Event counters PMEVCNTR<n> for values of n less than HDCR.HPMN.
- If PMCR.DP is 1, the cycle counter PMCCNTR.

Other event counters are not affected by this field.

When PMCR.DP is 0, PMCCNTR is not affected by this field.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3p1 is implemented:

Guest Performance Monitors Disable. Controls PMU operation in Hyp mode when `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.

HPMD	Meaning
`0b0`	Counters are not affected by this mechanism.
`0b1`	If `ExternalSecureNoninvasiveDebugEnabled()` is FALSE, then all the following apply: • Affected event counters are prohibited from counting in Hyp mode. • If PMCR.DP is 1, then PMCCNTR is disabled in Hyp mode. Otherwise, PMCCNTR is not affected by this mechanism.

If `ExternalSecureNoninvasiveDebugEnabled()` is TRUE, then the event counters and PMCCNTR are not affected by this field.

Otherwise, the counters affected by this field are:

- Event counters PMEVCNTR<n> for values of n less than HDCR.HPMN.
- If PMCR.DP is 1, the cycle counter, PMCCNTR.

Other event counters are not affected by this field. When PMCR.DP is 0, PMCCNTR is not affected by this field.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps Non-secure EL0 and EL1 System register MRC or MCR accesses, reported using EC syndrome value `0x05`, and MRRC accesses, reported using EC syndrome value `0x0C`, to the Debug ROM registers to Hyp mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TDRA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure EL0 and EL1 System register accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode, unless it is trapped by DBGDSCRext.UDCCdis.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TDOSA, bit [10]

When FEAT_DoubleLock is implemented:

Trap debug OS-related register access. Traps Non-secure EL1 System register MRC or MCR accesses, reported using EC syndrome value `0x05`, to the powerdown debug System registers to Hyp mode.

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure EL1 System register accesses to the powerdown debug System registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, DBGOSDLR, and DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps Non-secure EL1 System register MRC or MCR accesses, reported using EC syndrome value `0x05`, to the powerdown debug System registers to Hyp mode.

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TDOSA	Meaning
0b1	Non-secure EL1 System register accesses to the powerdown debug System registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, and DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to DBGOSDLR are trapped.

Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap debug access. Traps Non-secure EL0 and EL1 System register MRC or MCR accesses, reported using EC syndrome value `0x05`, to those debug System registers in the (coproc==`0b1110`) encoding space that are not trapped by either of the following:

- HDCR.TDRA.
- HDCR.TDOSA.

TDA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure EL0 or EL1 System register accesses to the debug System registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA, are trapped to Hyp mode, unless it is trapped by DBGDSCRext.UDCCdis.

Traps of AArch32 accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TDE, bit [8]

Trap Debug exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL_D.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TDE	Meaning
0b0	The debug target Exception level is EL1.
0b1	If EL2 is enabled for the current Effective value of SCR.NS, the debug target Exception level is EL2, otherwise the debug target Exception level is EL1. The HDCR.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

For more information, see 'Routing debug exceptions'.

When HCR.TGE == 1, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

HPME, bit [7]

When FEAT_PMUv3 is implemented:

Hyp Enable.

HPME	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET.

The counters affected by this field are event counters PMEVCNTR<n> for values of n greater than or equal to HDCR.HPMN and less than PMCR.N. This applies even when EL2 is disabled in the current Security state.

Other event counters and PMCCNTR are not affected by this field.

If HDCR.HPMN is equal to PMCR.N, then this field has no effect.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When FEAT_PMUv3 is implemented:

Trap accesses of PMU registers. Enables a trap to EL2 on accesses of PMU registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TPM	Meaning
0b0	Accesses of the specified PMU registers are not trapped by this mechanism.
0b1	Accesses of the specified PMU registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.

The instructions affected by this control are:

- MRC and MCR accesses to PMCCFILTR, PMCCNTR, PMCNTENCLR, PMCNTENSET, PMCR, PMEVCNTR<n>, PMEVTYPER<n>, PMINTENCLR, PMINTENSET, PMOVSR, PMOVSSET, PMSELR, PMSWINC, PMUSERENR, PMXEVCNTR, and PMXEVTYPER.
- MRC accesses to PMCEID0 and PMCEID1.
- MRRC and MCRR accesses to PMCCNTR.
- If FEAT_PMUv3p1 is implemented, MRC accesses to PMCEID2 and PMCEID3.
- If FEAT_PMUv3p4 is implemented, MRC accesses to PMMIR.

Unless the instruction generates a higher priority exception, trapped instructions generate a Hyp Trap exception.

Trapped instructions are reported using EC syndrome value 0x03 for MRC and MCR accesses, and 0x04 for MRRC and MCRR accesses.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPMCR, bit [5]

When FEAT_PMUv3 is implemented:

Trap PMCR accesses. Traps Non-secure EL0 and EL1 MCR or MRC accesses to the PMCR to Hyp mode, reported using EC syndrome value 0x03.

TPMCR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Non-secure EL0 and EL1 accesses to the PMCR are trapped to Hyp mode, unless it is trapped by PMUSERENR.EN.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPMN, bits [4:0]

When FEAT_PMUv3 is implemented:

Defines the number of event counters PMEVCNTR<n> that are accessible from EL1 and, if permitted, from EL0.

HDCR.HPMN divides the event counters into a first range and a second range.

If HDCR.HPMN is not 0 and is less than the number of PMU event counters implemented by the PE, `NUM_PMU_COUNTERS`, then event counters [0..(HDCR.HPMN-1)] are in the first range, and the remaining event counters [HDCR.HPMN..(`NUM_PMU_COUNTERS`-1)] are in the second range.

If FEAT_HPMN0 is implemented and HDCR.HPMN is 0, then all of the following apply:

- No event counters are in the first range.
- All event counters are in the second range.

If HDCR.HPMN is equal to `NUM_PMU_COUNTERS`, or EL2 is not implemented, then all of the following apply:

- All event counters are in the first range.
- No event counters are in the second range.

All of the following apply for an event counter PMEVCNTR<n> in the first range:

- The counter is accessible from EL1, EL2, and EL3.
- The counter is accessible from EL0 if permitted by PMUSERENR.
- If FEAT_PMUv3p5 is implemented, then PMCR.LP determines whether the counter overflow flag PMOVSSET[n] is set on unsigned overflow of PMEVCNTR<n>[31:0] or PMEVCNTR<n>[63:0]. PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.
- PMCR.E and PMCNTENSET[n] enable the operation of the event counter.

All of the following apply for an event counter PMEVCNTR<n> in the second range:

- The counter is accessible from EL2 and EL3.
- If EL2 is disabled in the current Security state, then the event counter is accessible from EL1, and from EL0 if permitted by PMUSERENR.
- If FEAT_PMUv3p5 is implemented, HDCR.HLP determines whether the counter overflow flag PMOVSSET[n] is set on unsigned overflow of PMEVCNTR<n>[31:0] or PMEVCNTR<n>[63:0]. PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.
- HDCR.HPME and PMCNTENSET[n] enable the operation of the event counter.

Values greater than `NUM_PMU_COUNTERS` are reserved. If FEAT_HPMN0 is not implemented, then the value 0 is reserved.

If this field is set to a reserved value, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- The number of event counters in each of the first and second ranges is UNKNOWN. That is, either:
  - The PE behaves as if HDCR.HPMN is set to an UNKNOWN nonzero value less than or equal to `NUM_PMU_COUNTERS`.
  - All counters are in the second range and none are in the first range.

The reset behavior of this field is:

- On a Warm reset, this field resets to the expression `NUM_PMU_COUNTERS`.

Otherwise:

Reserved, RES0.

Accessing HDCR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = HDCR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = HDCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TDA == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        HDCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HDCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.33 HTRFCR, Hyp Trace Filter Control Register

The HTRFCR characteristics are:

Purpose

Provides EL2 controls for Trace.

Configuration

If EL2 is not implemented, this register is RES0 from Monitor mode when SCR.NS == 1.

AArch32 System register HTRFCR bits [31:0] are architecturally mapped to AArch64 System register TRFCR_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented and FEAT_TRF is implemented. Otherwise, direct accesses to HTRFCR are UNDEFINED.

Attributes

HTRFCR is a 32-bit register.

Field descriptions

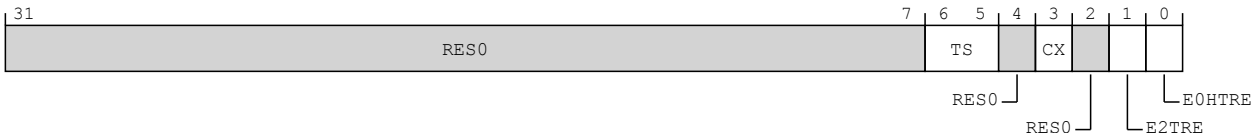

Bits [31:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

TS	Meaning
0b00	The timestamp is controlled by TRFCR.TS.
0b01	Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF.
0b11	Physical timestamp. The traced timestamp is the physical counter value.

When `SelfHostedTraceEnabled()` == FALSE, this field is ignored.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'00'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [4]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CX, bit [3]

VMID Trace Enable.

CX	Meaning
0b0	VMID tracing is not allowed.
0b1	VMID tracing is allowed.

When `SelfHostedTraceEnabled()` == FALSE, this field is ignored.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Bit [2]

Reserved, RES0.

E2TRE, bit [1]

EL2 Trace Enable.

E2TRE	Meaning
0b0	Tracing is prohibited at EL2.
0b1	Tracing is allowed at EL2.

When `SelfHostedTraceEnabled()` == FALSE, this field is ignored.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

E0HTRE, bit [0]

EL0 Trace Enable.

E0HTRE	Meaning
0b0	Tracing is prohibited at EL0 when HCR.TGE == 1.
0b1	Tracing is allowed at EL0 when HCR.TGE == 1.

This field is ignored if any of the following are true:

- The PE is in Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `SelfHostedTraceEnabled()` == FALSE.
- HCR.TGE == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HTRFCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL2) && IsFeatureImplemented(FEAT_TRF)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
        ↪ELUsingAArch32(EL3) && SDCR().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TTRF
        ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TTRF == '1'
        ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = HTRFCR();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t) = HTRFCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL2) && IsFeatureImplemented(FEAT_TRF)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
        ↪ELUsingAArch32(EL3) && SDCR().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TTRF
        ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TTRF == '1'
        ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        HTRFCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        HTRFCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.34 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation to software.

Configuration

AArch32 System register PMMIR bits [31:0] are architecturally mapped to AArch64 System register PMMIR_EL1[31:0].

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented, AArch32 System register PMMIR bits [31:0] are architecturally mapped to External register PMMIR[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR are UNDEFINED.

Attributes

PMMIR is a 32-bit register.

Field descriptions

31 – 28	27 – 24	23 – 20	19 – 16	15 – 8	7 – 0
RES0	EDGE	THWIDTH	BUS_WIDTH	BUS_SLOTS	SLOTS

Bits [31:28]

Reserved, RES0.

EDGE, bits [27:24]

PMU event edge detection. With PMMIR.THWIDTH, indicates implementation of event counter thresholding features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDGE	Meaning
0b0000	FEAT_PMUv3_EDGE is not implemented.
0b0001	FEAT_PMUv3_EDGE is implemented.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, the only permitted value is 0b0000.

FEAT_PMUv3_EDGE implements the functionality identified by the value 0b0001.

Note

PMEVTYPER<n>_EL0.TE cannot be accessed through PMEVTYPER<n>.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

THWIDTH, bits [23:20]

PMEVTYPER<n>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<n>_EL0.TH field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

THWIDTH	Meaning
0b0000	FEAT_PMUv3_TH is not implemented.
0b0001	1 bit. PMEVTYPER<n>_EL0.TH[11:1] are RES0.
0b0010	2 bits. PMEVTYPER<n>_EL0.TH[11:2] are RES0.
0b0011	3 bits. PMEVTYPER<n>_EL0.TH[11:3] are RES0.
0b0100	4 bits. PMEVTYPER<n>_EL0.TH[11:4] are RES0.
0b0101	5 bits. PMEVTYPER<n>_EL0.TH[11:5] are RES0.
0b0110	6 bits. PMEVTYPER<n>_EL0.TH[11:6] are RES0.
0b0111	7 bits. PMEVTYPER<n>_EL0.TH[11:7] are RES0.
0b1000	8 bits. PMEVTYPER<n>_EL0.TH[11:8] are RES0.
0b1001	9 bits. PMEVTYPER<n>_EL0.TH[11:9] are RES0.
0b1010	10 bits. PMEVTYPER<n>_EL0.TH[11:10] are RES0.
0b1011	11 bits. PMEVTYPER<n>_EL0.TH[11] is RES0.
0b1100	12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<n>_EL0.TH is $2^{(PMMIR.THWIDTH)}$ minus one.

Note

PMEVTYPER<n>_EL0.TH cannot be accessed through PMEVTYPER<n>.

Access to this field is RO.

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
0b0000	The information is not available.
0b0011	Four bytes.
0b0100	8 bytes.
0b0101	16 bytes.
0b0110	32 bytes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BUS_WIDTH	Meaning
0b0111	64 bytes.
0b1000	128 bytes.
0b1001	256 bytes.
0b1010	512 bytes.
0b1011	1024 bytes.
0b1100	2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment by in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b110

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PMUv3p4)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMMIR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMMIR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMMIR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.35 SDCR, Secure Debug Control Register

The SDCR characteristics are:

Purpose

Provides EL3 configuration options for self-hosted debug, trace, and the Performance Monitors Extension.

Configuration

This register is present only when FEAT_AA32EL3 is implemented. Otherwise, direct accesses to SDCR are UNDEFINED.

Attributes

SDCR is a 32-bit register.

Field descriptions

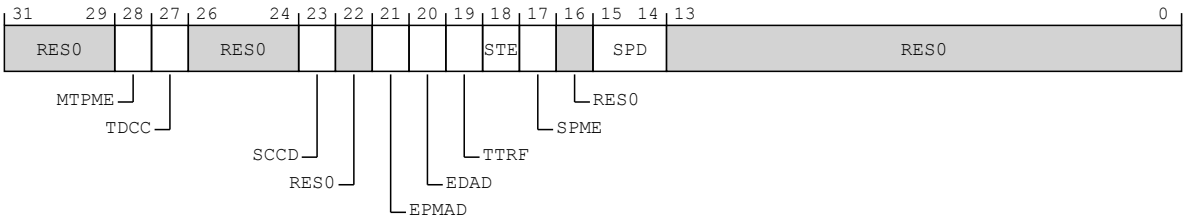

Bits [31:29]

Reserved, RES0.

MTPME, bit [28]

When FEAT_MTPMU is implemented:

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<n>.MT bits.

MTPME	Meaning
0b0	FEAT_MTPMU is disabled. The Effective value of PMEVTYPER<n>.MT is 0.
0b1	PMEVTYPER<n>.MT bits not affected by this field.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this field is 0.

The reset behavior of this field is:

- On a Cold reset:
  - When the highest implemented Exception level is EL3, this field resets to '1'.

Otherwise:

Reserved, RES0.

TDCC, bit [27]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel in modes other than Monitor mode to Monitor mode.

TDCC	Meaning
0b0	This control does not cause any register accesses to be trapped.
0b1	Accesses to the DCC registers in modes other than Monitor mode generate a Monitor Trap exception, unless the access also generates a higher priority exception. Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

- DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

When the PE is in Debug state, SDCR.TDCC does not trap any accesses to:

- DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [26:24]

Reserved, RES0.

SCCD, bit [23]

When FEAT_PMUv3p5 is implemented:

Secure Cycle Counter Disable. Prohibits PMCCNTR from counting in Secure state and EL3.

SCCD	Meaning
0b0	Cycle counting by PMCCNTR is not affected by this mechanism.
0b1	Cycle counting by PMCCNTR is prohibited in Secure state and EL3.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bit [22]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPMAD, bit [21]

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Non-secure access disable. Controls Non-secure access to Performance Monitors registers by an external debugger.

EPMAD	Meaning
`0b0`	No accesses from an external debugger to the Performance Monitor registers are prohibited by this control.
`0b1`	Non-secure accesses from an external debugger to the affected Performance Monitor registers are prohibited.

Otherwise, if EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

When FEAT_PMUv3_EXT is implemented:

External Performance Monitors access disable. Controls access to Performance Monitors registers by an external debugger.

EPMAD	Meaning
`0b0`	No accesses from an external debugger to the Performance Monitor registers are prohibited by this control.
`0b1`	If the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()` returns FALSE, then accesses from an external debugger to the affected Performance Monitor registers are prohibited.

Otherwise, if EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

EDAD, bit [20]

When FEAT_Debugv8p4 is implemented:

External debug Non-secure access disable. Controls Non-secure access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDAD	Meaning
0b0	No accesses from an external debugger to the debug registers are prohibited by this control.
0b1	Non-secure accesses from an external debugger to the affected debug registers are prohibited.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

When FEAT_Debugv8p2 is implemented:

External debug access disable. Controls access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

EDAD	Meaning
0b0	No accesses from an external debugger to the debug registers are prohibited by this control.
0b1	If the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()` returns FALSE, then accesses from an external debugger to the affected debug registers are prohibited.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

External debug access disable. Controls access to breakpoint, watchpoint, and optionally OSLAR_EL1 registers by an external debugger.

EDAD	Meaning
0b0	No accesses from an external debugger to the debug registers are prohibited by this control.
0b1	If the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()` returns FALSE, then accesses from an external debugger to the affected debug registers are prohibited.

It is IMPLEMENTATION DEFINED whether accesses to OSLAR_EL1 from an external debugger are affected by this control.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

TTRF, bit [19]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TRF is implemented:

Trap Trace Filter controls. Controls whether accesses in modes other than Monitor mode to the trace filter control registers generate a Monitor Trap exception.

TTRF	Meaning
0b0	Accesses to HTRFCR and TRFCR are not affected by this control bit.
0b1	When not in Monitor mode, accesses to HTRFCR and TRFCR generate a Monitor Trap exception, unless the access generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

Otherwise:

Reserved, RES0.

STE, bit [18]

When FEAT_TRF is implemented:

Secure Trace Enable. This field enables tracing in Secure state and controls the level of authentication required by an external debugger to enable external tracing.

STE	Meaning
0b0	Trace is prohibited in Secure state unless overridden by the IMPLEMENTATION DEFINED authentication interface.
0b1	Trace in Secure state is not affected by this field.

This field also controls the level of authentication required by an external debugger to enable external tracing. See 'Register controls to enable self-hosted trace'.

If EL3 is not implemented and the Effective value of SCR.NS is 0, the PE behaves as if this field is set to 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '0'.

Otherwise:

Reserved, RES0.

SPME, bit [17]

When FEAT_PMUv3 is implemented and FEAT_Debugv8p2 is implemented:

Secure Performance Monitors Enable. Controls event counting in Secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SPME	Meaning
0b0	Event counting is prohibited in Secure state. If PMCR.DP is 1, PMCCNTR is disabled in Secure state. Otherwise, PMCCNTR is not affected by this mechanism.
0b1	Event counting and PMCCNTR are not affected by this mechanism.

This field affects the operation of all event counters in Secure state, and if PMCR.DP is 1, the operation of PMCCNTR in Secure state. When PMCR.DP is 0, PMCCNTR is not affected by this field.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

When FEAT_PMUv3 is implemented:

Secure Performance Monitors Enable. Controls event counting in Secure state.

SPME	Meaning
0b0	If `ExternalSecureNoninvasiveDebugEnabled()` is FALSE, event counting is prohibited in Secure state, and if PMCR.DP is 1, PMCCNTR is disabled in Secure state.
0b1	Event counting and PMCCNTR are not affected by this mechanism.

If `ExternalSecureNoninvasiveDebugEnabled()` is TRUE, the event counters and PMCCNTR are not affected by this field.

Otherwise, this field affects the operation of all event counters in Secure state, and if PMCR.DP is 1, the operation of PMCCNTR in Secure state. When PMCR.DP is 0, PMCCNTR is not affected by this field.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [16]

Reserved, RES0.

SPD, bits [15:14]

AArch32 Secure self-hosted Privileged Debug. Enables or disables debug exceptions from EL3, other than Breakpoint Instruction exceptions.

SPD	Meaning
0b00	Legacy mode. Debug exceptions from EL3 are enabled by the authentication interface.
0b10	Secure privileged debug disabled. Debug exceptions from EL3 are disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SPD	Meaning
0b11	Secure privileged debug enabled. Debug exceptions from EL3 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as 0b00. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Breakpoint Instruction exceptions. These are always enabled.

This field is ignored in Non-secure state.

If debug exceptions from EL3 are enabled, then debug exceptions from Secure EL0 are also enabled.

Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER.SUIDEN is 1.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 0b11.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3, this field resets to '00'.

Bits [13:0]

Reserved, RES0.

Accessing SDCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    R(t) = SDCR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32EL3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) &&
        ↪IsCurrentSecurityState(SS_Secure) then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CP15SDISABLE2 == HIGH then
        Undefined();
    else
        SDCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.36 SDER, Secure Debug Enable Register

The SDER characteristics are:

Purpose

Controls invasive and non-invasive debug in the Secure EL0 mode.

Configuration

When EL2 is implemented and FEAT_SEL2 is implemented, AArch32 System register SDER bits [31:0] are architecturally mapped to AArch64 System register SDER32_EL2[31:0].

When EL3 is implemented, AArch32 System register SDER bits [31:0] are architecturally mapped to AArch64 System register SDER32_EL3[31:0].

This register is present only when (HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3)) || ((IsFeatureImplemented(FEAT_AA32EL1) && HaveELUsingSecurityState(EL1, TRUE)) && IsFeatureImplemented(FEAT_Secure)). Otherwise, direct accesses to SDER are UNDEFINED.

Attributes

SDER is a 32-bit register.

Field descriptions

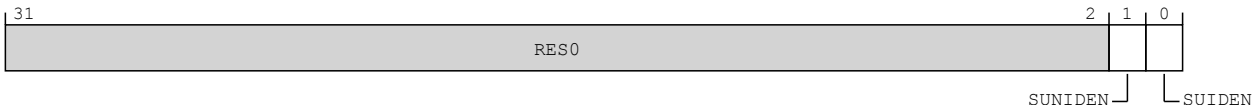

Bits [31:2]

Reserved, RES0.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug Enable.

SUNIDEN	Meaning
`0b0`	This bit has no effect on non-invasive debug.
`0b1`	Non-invasive debug is allowed in Secure EL0 using AArch32.

When EL3 or Secure EL1 is using AArch32, the forms of non-invasive debug affected by this control are:

- The PC Sample-based Profiling Extension. See About the PC Sample-based Profiling Extension.
- When `SelfHostedTraceEnabled()` == FALSE, processor trace.
- When EL3 is implemented, Performance Monitors.

When Secure EL1 is using AArch64, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

SUIDEN, bit [0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL3 is implemented:

Secure User Invasive Debug Enable.

SUIDEN	Meaning
0b0	This bit does not affect the generation of debug exceptions at Secure EL0.
0b1	If EL3 or EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Accessing SDER

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b001

```
if !((HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3)) || (IsFeatureImplemented(FEAT_AA32EL1) &&
     ↪HaveELUsingSecurityState(EL1, TRUE) && IsFeatureImplemented(FEAT_Secure))) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        R(t) = SDER();
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    R(t) = SDER();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b001

```
if !((HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3)) || (IsFeatureImplemented(FEAT_AA32EL1) &&
     ↪HaveELUsingSecurityState(EL1, TRUE) && IsFeatureImplemented(FEAT_Secure))) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif !IsCurrentSecurityState(SS_Secure) then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪MDCR_EL2().[TDE,TDA] != '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TDA ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL3, 0x03);
    else
        SDER() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    Undefined();
elsif PSTATE.EL == EL3 then
    if CP15SDISABLE2 == HIGH then
        Undefined();
    else
        SDER() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.3.37 TRFCR, Trace Filter Control Register

The TRFCR characteristics are:

Purpose

Provides EL1 controls for Trace.

Configuration

AArch32 System register TRFCR bits [31:0] are architecturally mapped to AArch64 System register TRFCR_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_TRF is implemented. Otherwise, direct accesses to TRFCR are UNDEFINED.

Attributes

TRFCR is a 32-bit register.

Field descriptions

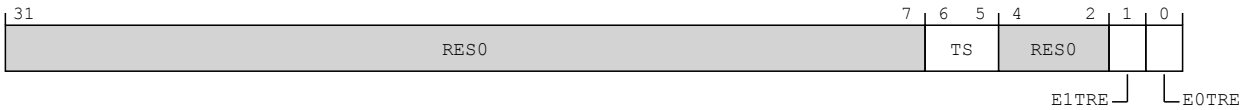

Bits [31:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

TS	Meaning	Applies when
0b01	Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF.	
0b10	Guest physical timestamp. The traced timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2: • EL3 is implemented and is using AArch32. • EL3 is implemented, using AArch64, and SCR_EL3.ECVEn is 0. • EL2 is using AArch32. • EL2 is using AArch64 and CNTHCTL_EL2.ECV is 0. • FEAT_ECV_POFF is not implemented.	FEAT_ECV is implemented
0b11	Physical timestamp. The traced timestamp is the physical counter value.	

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is ignored by the PE when any of the following are true:

- EL2 is implemented and HTRFCR.TS != `0b00`.
- `SelfHostedTraceEnabled()` == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:2]

Reserved, RES0.

E1TRE, bit [1]

EL1 Trace Enable.

E1TRE	Meaning
`0b0`	Tracing is prohibited in PL1 modes.
`0b1`	Tracing is allowed in PL1 modes.

This field is ignored if `SelfHostedTraceEnabled()` == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

E0TRE, bit [0]

EL0 Trace Enable.

E0TRE	Meaning
`0b0`	Tracing is prohibited at EL0.
`0b1`	Tracing is allowed at EL0.

This field is ignored if any of the following are true:

- `SelfHostedTraceEnabled()` == FALSE.
- EL2 is implemented and enabled in the current security state and HCR.TGE == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRFCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_TRF)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDCR().TTRF == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TTRF
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TTRF ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TTRF
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SDCR().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = TRFCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TTRF
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TTRF == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = TRFCR();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TTRF == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = TRFCR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_TRF)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDCR().TTRF == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T1 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T1 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TTRF
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TTRF ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TTRF
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SDCR().TTRF == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        TRFCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TTRF == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SDCR().TTRF == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TTRF
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SDCR().TTRF == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        TRFCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR().TTRF == '1' then
        AArch32_TakeMonitorTrapException();
    else
        TRFCR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4 Performance Monitors registers

This section lists the Performance Monitors registers in AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.1 PMCCFILTR, Performance Monitors Cycle Count Filter Register

The PMCCFILTR characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR, increments.

Configuration

AArch32 System register PMCCFILTR bits [31:0] are architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

AArch32 System register PMCCFILTR bits [31:0] are architecturally mapped to External register PMCCFILTR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR are UNDEFINED.

Attributes

PMCCFILTR is a 32-bit register.

Field descriptions

31	30	29	28	27	26 … 22	21	20 … 0
P	U	NSK	NSU	NSH	RES0	RLU	RES0

P, bit [31]

Privileged filtering. Controls counting cycles in EL1 and, if EL3 is using AArch32, EL3.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1 and, if EL3 is using AArch32, EL3.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR.NSK.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering. Controls counting cycles in EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR.NSU.

If FEAT_RME is implemented, then counting cycles in Realm EL0 is further controlled by PMCCFILTR.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting cycles in Non-secure EL1. If PMCCFILTR.NSK is not equal to PMCCFILTR.P, then the PE does not count cycles in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL1.

NSK	Meaning
0b0	When PMCCFILTR.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR.P == 1, the PE does not count cycles in Non-secure EL1.
0b1	When PMCCFILTR.P == 0, the PE does not count cycles in Non-secure EL1. When PMCCFILTR.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting cycles in Non-secure EL0. If PMCCFILTR.NSU is not equal to PMCCFILTR.U, then the PE does not count cycles in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSU	Meaning
`0b0`	When PMCCFILTR.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR.U == 1, the PE does not count cycles in Non-secure EL0.
`0b1`	When PMCCFILTR.U == 0, the PE does not count cycles in Non-secure EL0. When PMCCFILTR.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
`0b0`	The PE does not count cycles in EL2.
`0b1`	This mechanism has no effect on filtering of cycles.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting cycles in Secure EL2 is further controlled by PMCCFILTR.SH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [26:22]

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting cycles in Realm EL0. If PMCCFILTR.RLU is not equal to PMCCFILTR.U, then the PE does not count cycles in Realm EL0. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLU	Meaning
0b0	When PMCCFILTR.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR.U == 1, the PE does not count cycles in Realm EL0.
0b1	When PMCCFILTR.U == 0, the PE does not count cycles in Realm EL0. When PMCCFILTR.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [20:0]

Reserved, RES0.

Accessing PMCCFILTR

PMCCFILTR can also be accessed by using PMXEVTYPER with PMSELR.SEL set to 0b11111.

Permitted reads and writes of PMCCFILTR are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.C is 0.

Permitted writes of PMCCFILTR are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.{UEN,CR} is {1,1}.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b1111	0b111

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMCCFILTR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().C == '0' then
            R(t) = Zeros{32};
        else
            R(t) = PMCCFILTR();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        R(t) = PMCCFILTR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCCFILTR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCCFILTR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b1111	0b111

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGWTR_EL2().PMCCFILTR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().C == '0' ||
             ↪PMUSERENR_EL0().CR == '1') then
            return;
        else
            PMCCFILTR() = R(t);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCCFILTR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCCFILTR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCCFILTR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.2 PMCCNTR, Performance Monitors Cycle Count Register

The PMCCNTR characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See 'Time as measured by the Performance Monitors cycle counter' for more information.

PMCCFILTR determines the modes and states in which the PMCCNTR can increment.

Configuration

PMCCNTR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR continues to increment when clocks are stopped by WFI and WFE instructions.

AArch32 System register PMCCNTR bits [63:0] are architecturally mapped to AArch64 System register PMCCNTR_EL0[63:0].

AArch32 System register PMCCNTR bits [63:0] are architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR are UNDEFINED.

Attributes

PMCCNTR is a 64-bit register.

Field descriptions

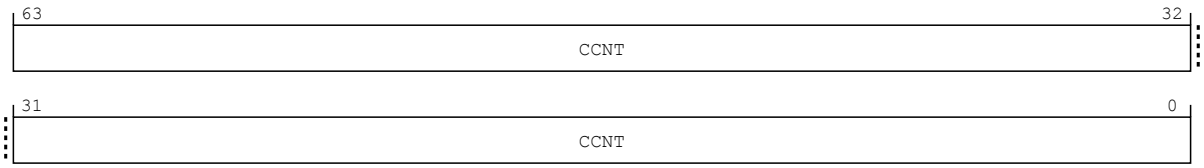

CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR.C sets this field to 0.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR

Permitted reads and writes of PMCCNTR are RAZ/WI if all of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.C is 0.

Permitted writes of PMCCNTR are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.{UEN,CR} is {1,1}.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,CR,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[CR,EN] == '00')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[CR,EN] == '00' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().C == '0' then
            R(t) = Zeros{32};
        else
            R(t) = PMCCNTR()[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCCNTR()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCCNTR()[31:0];
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCCNTR()[31:0];
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGWTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().C == '0' ||
             ↪PMUSERENR_EL0().CR == '1') then
            return;
        else
            PMCCNTR()[31:0] = R(t);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCCNTR()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCCNTR()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCCNTR()[31:0] = R(t);
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1001	0b0000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
        ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,CR,EN] == '000') ||
        ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[CR,EN] == '00')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[CR,EN] == '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x04);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGRTR_EL2().PMCCNTR_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x04);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x04);
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1().C == '0' then
                R(t, t2) = Zeros{64};
            else
                R(t, t2) = PMCCNTR();
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x04);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x04);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x04);
            end;
        else
            R(t, t2) = PMCCNTR();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        R(t, t2) = PMCCNTR();
    end;
elsif PSTATE.EL == EL3 then
    R(t, t2) = PMCCNTR();
end;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1001	0b0000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
        ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HDFGWTR_EL2().PMCCNTR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1().C == '0' ||
             ↪PMUSERENR_EL0().CR == '1') then
            return;
        else
            PMCCNTR() = R(t, t2);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        PMCCNTR() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        PMCCNTR() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    PMCCNTR() = R(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.3 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x0000` to `0x001F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Configuration

AArch32 System register PMCEID0 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[31:0].

AArch32 System register PMCEID0 bits [31:0] are architecturally mapped to External register PMCEID0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0 are UNDEFINED.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions

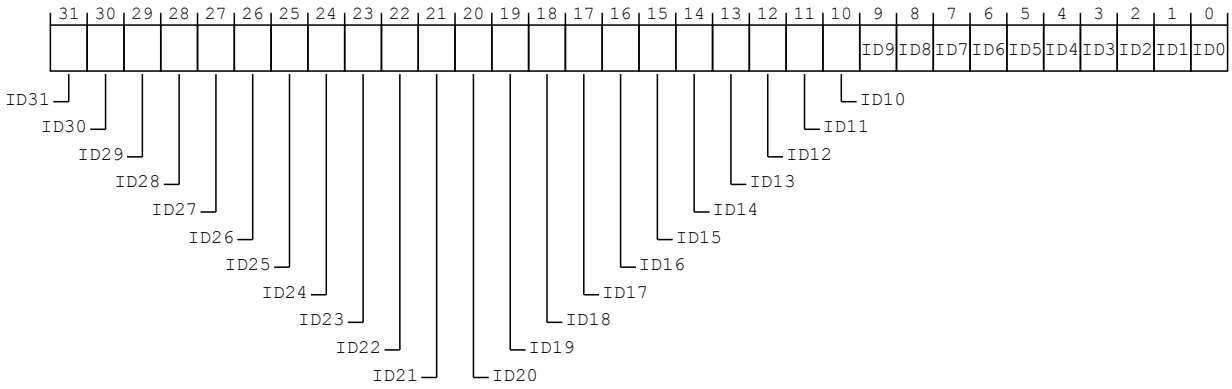

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b110

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID0();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    R(t) = PMCEID0();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.4 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x0020` to `0x003F`.

For more information about the Common events and the use of the PMCEIDn registers see 'The PMU event number space and common events'.

Configuration

AArch32 System register PMCEID1 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[31:0].

AArch32 System register PMCEID1 bits [31:0] are architecturally mapped to External register PMCEID1[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1 are UNDEFINED.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions

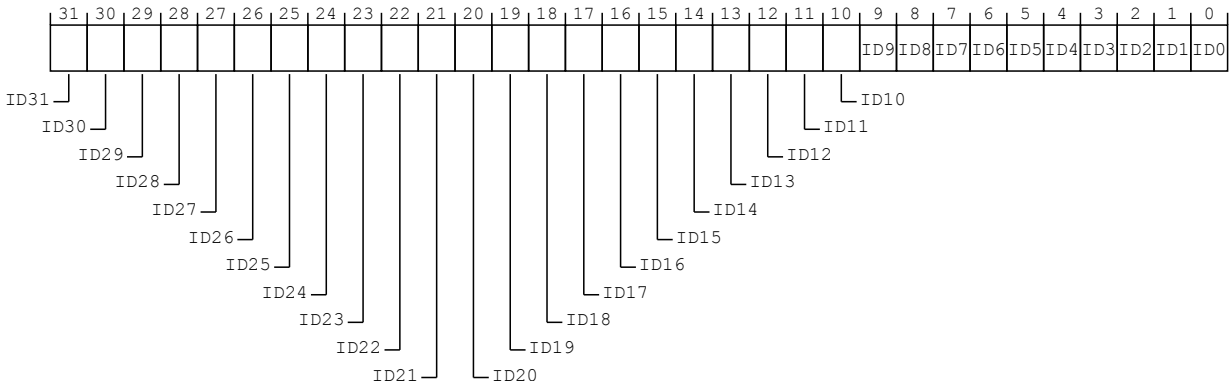

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (`0x0020` + n).

For each bit:

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID1

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b111

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID1();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID1();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID1();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    R(t) = PMCEID1();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.5 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4000` to `0x401F`.

For more information about the Common events and the use of the PMCEIDn registers see ‘The PMU event number space and common events’.

Configuration

AArch32 System register PMCEID2 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[63:32].

AArch32 System register PMCEID2 bits [31:0] are architecturally mapped to External register PMCEID2[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID2 are UNDEFINED.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

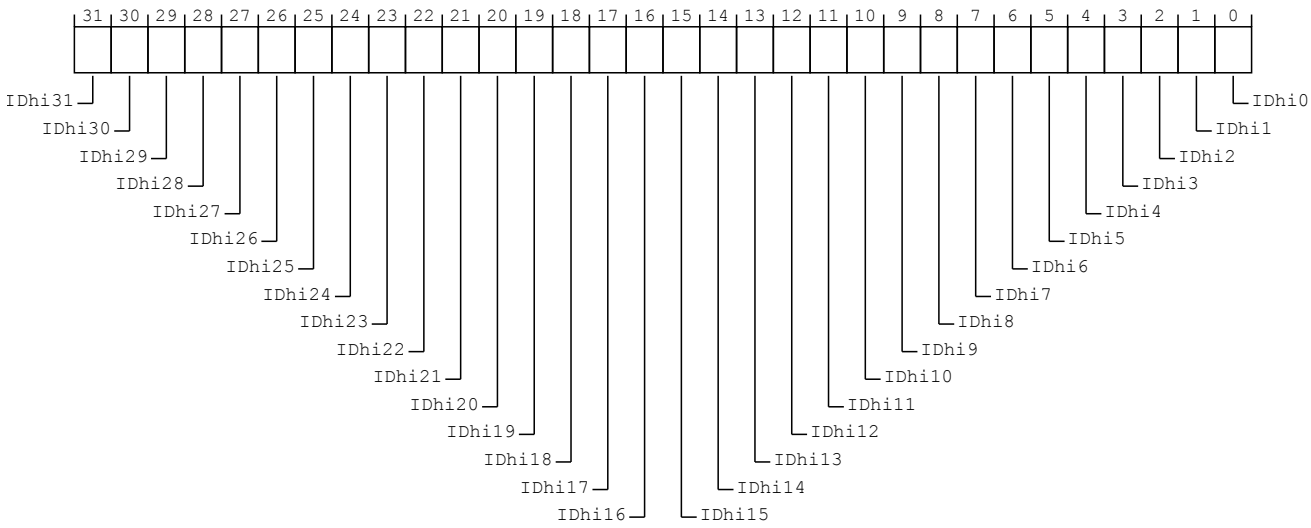

IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4000` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b100

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3p1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID2();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID2();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        R(t) = PMCEID2();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCEID2();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.6 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4020` to `0x403F`.

For more information about the Common events and the use of the PMCEIDn registers see 'The PMU event number space and common events'.

Configuration

AArch32 System register PMCEID3 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[63:32].

AArch32 System register PMCEID3 bits [31:0] are architecturally mapped to External register PMCEID3[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID3 are UNDEFINED.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

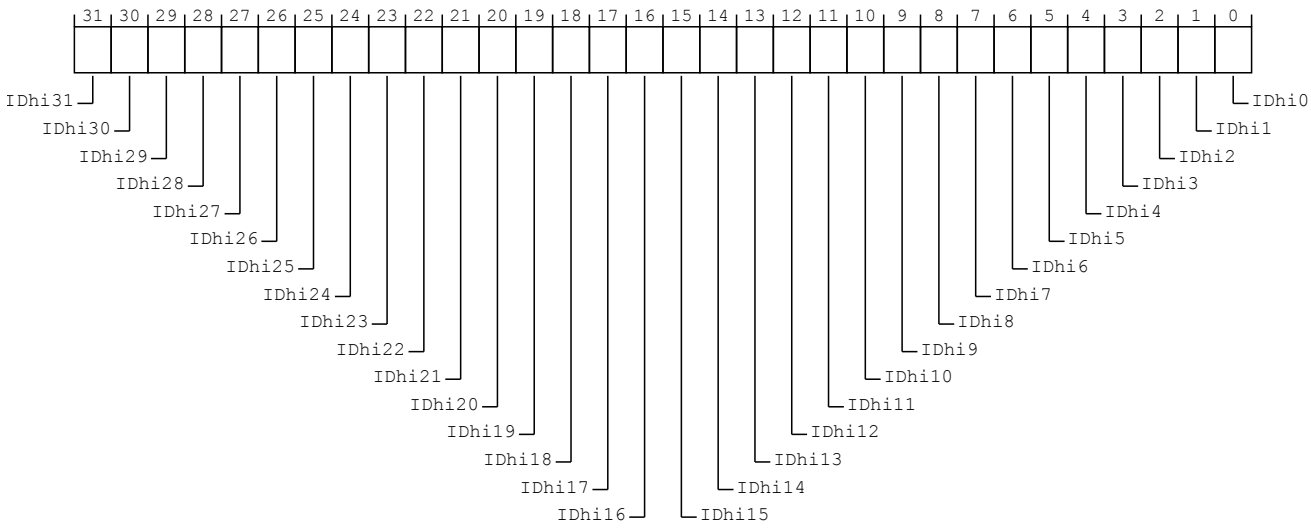

IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4020` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID3

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b101

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3p1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) &&
         ↪IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR().TID == '1' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGRTR_EL2().PMCEIDn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID3();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCEID3();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        R(t) = PMCEID3();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCEID3();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.7 PMCNTENCLR, Performance Monitors Count Enable Clear register

The PMCNTENCLR characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter PMCCNTR.
- The event counters PMEVCNTR<n>.

Reading from this register shows which counters are enabled.

Configuration

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR are UNDEFINED.

Attributes

PMCNTENCLR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

PMCCNTR disable. On writes, allows software to disable PMCCNTR. On reads, returns the PMCCNTR enable status.

C	Meaning
0b0	PMCCNTR disabled.
0b1	PMCCNTR enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.C == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == '11'
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m> disable. On writes, allows software to disable PMEVCNTR<m>. On reads, returns the PMEVCNTR<m> enable status.

P<m>	Meaning
0b0	PMEVCNTR<m> disabled.
0b1	PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.P<m> == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == '11'
- Otherwise, access to this field is W1C.

Accessing PMCNTENCLR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b010

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
        ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HDFGRTR_EL2().PMCNTEN == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCNTENCLR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
        ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCNTENCLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCNTENCLR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCNTENCLR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b010

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                 ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGWTR_EL2().PMCNTEN == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMCNTENCLR() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMCNTENCLR() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCNTENCLR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCNTENCLR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.8 PMCNTENSET, Performance Monitors Count Enable Set register

The PMCNTENSET characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter PMCCNTR.
- The event counters PMEVCNTR<n>.

Reading from this register shows which counters are enabled.

Configuration

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET are UNDEFINED.

Attributes

PMCNTENSET is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

PMCCNTR enable. On writes, allows software to enable PMCCNTR. On reads, returns the PMCCNTR enable status.

C	Meaning
0b0	PMCCNTR disabled.
0b1	PMCCNTR enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.C == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == ‘11’
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m> enable. On writes, allows software to enable PMEVCNTR<m>. On reads, returns the PMEVCNTR<m> enable status.

P<m>	Meaning
0b0	PMEVCNTR<m> disabled.
0b1	PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == ‘1’
  - PMUACR_EL1.P<m> == ‘0’
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == ‘11’
- Otherwise, access to this field is W1S.

Accessing PMCNTENSET

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
        ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HDFGRTR_EL2().PMCNTEN == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCNTENSET();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
        ↪'1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCNTENSET();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCNTENSET();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCNTENSET();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b001

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                 ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGWTR_EL2().PMCNTEN == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMCNTENSET() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMCNTENSET() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCNTENSET() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCNTENSET() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.9 PMCR, Performance Monitors Control Register

The PMCR characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configuration

AArch32 System register PMCR bits [31:0] are architecturally mapped to AArch64 System register PMCR_EL0[31:0].

AArch32 System register PMCR bits [10:0] are architecturally mapped to External register PMCR_EL0[10:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR are UNDEFINED.

Attributes

PMCR is a 32-bit register.

Field descriptions

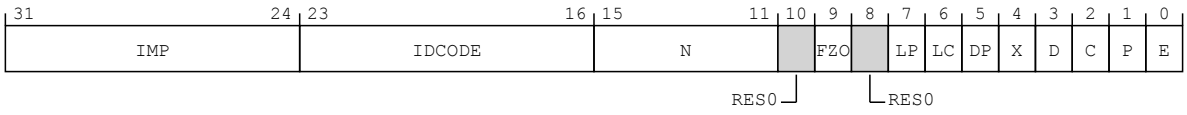

IMP, bits [31:24]

When FEAT_PMUv3p7 is not implemented:

Implementer code.

If this field is zero, then PMCR.IDCODE is RES0 and software must use MIDR to identify the PE.

Otherwise, this field and PMCR.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A nonzero value has the same interpretation as MIDR.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

IDCODE, bits [23:16]

When PMCR.IMP != '00000000':

Identification code. Arm deprecates use of this field.

Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of `0b00000`-`0b11111`. If the value is `0b00000`, then only PMCCNTR is implemented. If the value is `0b11111`, then PMCCNTR and 31 event counters are implemented.

If EL2 is implemented, then all of the following apply:

- If EL2 is using AArch32, then reads of this field from Non-secure EL1 and Non-secure EL0 return the Effective value of HDCR.HPMN.
- If EL2 is enabled in the current Security state and using AArch64, then reads of this field from EL1 and EL0 return the Effective value of MDCR_EL2.HPMN.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

FZO	Meaning
`0b0`	Do not freeze on overflow.
`0b1`	Affected counters do not count when PMOVSR[m] is 1 for any event counter PMEVCNTR<m> in the first range.

The counters affected by this field are:

- The event counters in the first range.
- If PMCR.DP is 1, the cycle counter PMCCNTR.

Other event counters are not affected by this field.

When PMCR.DP is 0, PMCCNTR is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN or HDCR.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

LP, bit [7]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by PMOVSR.P[n].

LP	Meaning
0b0	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[31:0].
0b1	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[63:0].

The counters affected by this field are the event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN or HDCR.HPMN.

Other event counters and PMCCNTR are not affected by this field.

PMEVCNTR<n>[63:32] is not accessible in AArch32 state.

If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this field is read/write or RAZ/WI.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSR.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR[63:0].

Arm deprecates use of PMCR.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DP, bit [5]

When ((HaveEL(EL3) || (IsFeatureImplemented(FEAT_PMUv3p1) && HaveEL(EL2))) || IsFeatureImplemented(FEAT_PMUv3p7)) || IsFeatureImplemented(FEAT_SPE_DPFZS):

Disable cycle counter when event counting is prohibited.

DP	Meaning
0b0	Cycle counting by PMCCNTR is not affected by this mechanism.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DP	Meaning
0b1	Cycle counting by PMCCNTR is disabled in prohibited regions and when event counting is frozen: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD or HDCR.HPMD is 1, then cycle counting by PMCCNTR is disabled at EL2. • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR.FZO, then cycle counting by PMCCNTR is disabled. • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR is disabled at EL3. • If EL3 is implemented, MDCR_EL3.SPME or SDCR.SPME is 0, and one of FEAT_PMUv3p7 is not implemented, EL3 is using AArch32, or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR is disabled at EL3 and in Secure state.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited or frozen. For more information about event counter ranges, see MDCR_EL2.HPMN or HDCR.HPMN.

For more information, see 'Prohibiting event and cycle counting'.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETE is implemented, this field does not affect the use of PMU events as an External Input by the trace unit.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

Clock divider.

D	Meaning
0b0	When enabled, PMCCNTR counts every clock cycle.
0b1	When enabled, PMCCNTR counts once every 64 clock cycles.

If the Effective value of PMCR.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR.D = 1.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR to zero.

Note

Resetting PMCCNTR does not change the cycle counter overflow field. The value of PMCR.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
0b0	No action.
0b1	Reset all affected event counters PMEVCNTR<n> to zero.

The event counters affected by this field are:

- All event counters in the first range.
- If any of the following are true, all event counters in the second range:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL2 is disabled or not implemented in the current Security state.
- The PE is executing at EL2 or EL3.

Writes to this field do not affect other event counters or the cycle counter PMCCNTR.

For more information about event counter ranges, see MDCR_EL2.HPMN or HDCR.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP or HDCR.HLP and PMCR.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN or HDCR.HPMN.
- The cycle counter PMCCNTR.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PMCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' ||
        ↪(IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_AArch32SystemAccessTrap(EL1, 0x03);
            end;
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪MDCR_EL2().TPMCR == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPMCR ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMCR();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPMCR
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPMCR ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMCR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMCR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' ||
         ↪(IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().UEN == '1')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGWTR_EL2().PMCR_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪MDCR_EL2().TPMCR == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPMCR ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMCR() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPMCR
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPMCR ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMCR() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMCR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMCR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.10 PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n> characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configuration

AArch32 System register PMEVCNTR<n> bits [31:0] are architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0[31:0].

AArch32 System register PMEVCNTR<n> bits [31:0] are architecturally mapped to External register PMEVCNTR<n>_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n> are UNDEFINED.

Attributes

PMEVCNTR<n> is a 32-bit register.

Field descriptions

31	0
Event counter n	

EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

If FEAT_PMUv3p5 is implemented, the event counter is 64 bits and only the least-significant part of the event counter is accessible in AArch32 state:

- Reads from PMEVCNTR<n> return bits [31:0] of the counter.
- Writes to PMEVCNTR<n> update bits [31:0] and leave bits [63:32] unchanged.
- There is no means to access bits [63:32] directly from AArch32 state.
- If the implementation does not support AArch64, bits [63:32] are not required to be implemented.

If FEAT_PMUv3p5 is not implemented, the event counter is 32 bits.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>

PMEVCNTR<n> can also be accessed by using PMXEVCNTR with PMSELR.SEL set to n.

If FEAT_FGT is implemented and <n> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVCNTR<n> is as follows:

- If <n> is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_FGT is not implemented and <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVCNTR<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMEVCNTR<n> are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.P<n> is 0.

Permitted writes of PMEVCNTR<n> are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.{UEN,ER} is {1,1}.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.{ER,EN} or PMUSERENR_EL0.{UEN,ER,EN}.
If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see HDCR.HPMN and MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b10:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
            ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
            ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00')) then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL1, 0x03);
            end;
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[ER,EN] == '00' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGRTR_EL2().PMEVCNTRn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1()[m] == '0' then
                R(t) = Zeros{32};
            else
                R(t) = PMEVCNTR(m);
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMEVCNTR(m);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMEVCNTR(m);
        end;
    elsif PSTATE.EL == EL3 then
        R(t) = PMEVCNTR(m);
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b10:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGWTR_EL2().PMEVCNTRn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1()[m] == '0' ||
                ↪PMUSERENR_EL0().ER == '1') then
                return;
            else
                PMEVCNTR(m) = R(t);
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMEVCNTR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMEVCNTR(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMEVCNTR(m) = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.11 PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n> characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

AArch32 System register PMEVTYPER<n> bits [31:0] are architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[31:0].

AArch32 System register PMEVTYPER<n> bits [31:0] are architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n> are UNDEFINED.

Attributes

PMEVTYPER<n> is a 32-bit register.

Field descriptions

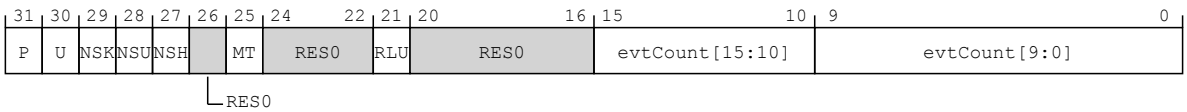

P, bit [31]

Privileged filtering. Controls counting events in EL1 and, if EL3 is using AArch32, EL3.

P	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL1 and, if EL3 is using AArch32, EL3.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>.NSK.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering. Controls counting events in EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>.NSU.

If FEAT_RME is implemented, then counting events in Realm EL0 is further controlled by PMEVTYPER<n>.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting events in Non-secure EL1. If PMEVTYPER<n>.NSK is not equal to PMEVTYPER<n>.P, then the PE does not count events in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL1.

NSK	Meaning
0b0	When PMEVTYPER<n>.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>.P == 1, the PE does not count events in Non-secure EL1.
0b1	When PMEVTYPER<n>.P == 0, the PE does not count events in Non-secure EL1. When PMEVTYPER<n>.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting events in Non-secure EL0. If PMEVTYPER<n>.NSU is not equal to PMEVTYPER<n>.U, then the PE does not count events in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSU	Meaning
0b0	When PMEVTYPER<n>.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>.U == 1, the PE does not count events in Non-secure EL0.
0b1	When PMEVTYPER<n>.U == 0, the PE does not count events in Non-secure EL0. When PMEVTYPER<n>.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting events in Secure EL2 is further controlled by PMEVTYPER<n>.SH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

MT, bit [25]

When FEAT_MTPMU is implemented or an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See 'Multithreaded implementations' and 'Cycle event counting in multithreaded implementations'.

From Armv8.6, the IMPLEMENTATION DEFINED multi-threaded PMU extension is not permitted, meaning if FEAT_MTPMU is not implemented, this field is RES0. See ID_DFR1.MTPMU.

This field is ignored by the PE and treated as zero when FEAT_MTPMU is implemented and disabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:22]

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting events in Realm EL0. If PMEVTYPER<n>.RLU is not equal to PMEVTYPER<n>.U, then the PE does not count events in Realm EL0. Otherwise, this mechanism has no effect on filtering of events in Realm EL0.

RLU	Meaning
0b0	When PMEVTYPER<n>.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>.U == 1, the PE does not count events in Realm EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLU	Meaning
`0b1`	When PMEVTYPER<n>.U == 0, the PE does not count events in Realm EL0. When PMEVTYPER<n>.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [20:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When FEAT_PMUv3p1 is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMEVCNTR<n>.

The ranges of event numbers allocated to each type of event are shown in 'Allocation of the PMU event number space'.

If FEAT_PMUv3p8 is implemented and PMEVTYPER<n>.evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is the value written to the field.

Note

Arm recommends this behavior for all implementations of FEAT_PMUv3.

Otherwise, if PMEVTYPER<n>.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is the value written to the field.
- If FEAT_PMUv3p1 is implemented, for the range `0x4000` to `0x403F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>

PMEVTYPER<n> can also be accessed by using PMXEVTYPER with PMSELR.SEL set to n.

If FEAT_FGT is implemented and <n> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVTYPER<n> is as follows:

- If <n> is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMEVTYPER<n> are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.P<n> is 0.

Permitted writes of PMEVTYPER<n> are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.{UEN,ER} is {1,1}.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.EN or PMUSERENR_EL0.{UEN,EN}.
If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see HDCR.HPMN and MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b11:m[4:3]	m[2:0]

```
let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1()[m] == '0' then
                R(t) = Zeros{32};
            else
                R(t) = PMEVTYPER(m);
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMEVTYPER(m);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMEVTYPER(m);
        end;
    elsif PSTATE.EL == EL3 then
        R(t) = PMEVTYPER(m);
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b11:m[4:3]	m[2:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGWTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1()[m] == '0' ||
             ↪PMUSERENR_EL0().ER == '1') then
            return;
        else
            PMEVTYPER(m) = R(t);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMEVTYPER(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMEVTYPER(m) = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMEVTYPER(m) = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.12 PMINTENCLR, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR characteristics are:

Purpose

Allows software to disable the generation of interrupt requests on overflows from the following counters:

- The cycle counter PMCCNTR.
- The event counters PMEVCNTR<n>.

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR are UNDEFINED.

Attributes

PMINTENCLR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b010

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMINTENCLR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMINTENCLR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMINTENCLR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b010

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMINTENCLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMINTENCLR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMINTENCLR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.13 PMINTENSET, Performance Monitors Interrupt Enable Set register

The PMINTENSET characteristics are:

Purpose

Allows software to enable the generation of interrupt requests on overflows from the following counters:

- The cycle counter PMCCNTR.
- The event counters PMEVCNTR<n>.

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET are UNDEFINED.

Attributes

PMINTENSET is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing PMINTENSET

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMINTENSET();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMINTENSET();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMINTENSET();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMINTENSET() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMINTENSET() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMINTENSET() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.14 PMOVSR, Performance Monitors Overflow Flag Status Register

The PMOVSR characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter PMCCNTR.
- The event counters PMEVCNTR<n>.

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to AArch32 System register PMOVSSET[31:0].

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to External register PMOVS[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSR are UNDEFINED.

Attributes

PMOVSR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR to 0. On reads, returns the unsigned overflow flag for PMCCNTR overflow status.

C	Meaning
0b0	PMCCNTR has not overflowed.
0b1	PMCCNTR has overflowed.

PMCR.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR[31:0] or unsigned overflow of PMCCNTR[63:0].

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.C == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == '11'
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m> clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m> to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m> overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m> has not overflowed.
0b1	PMEVCNTR<m> has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP, HDCR.HLP, and PMCR.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>[31:0] or unsigned overflow of PMEVCNTR<m>[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.P<m> == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == '11'
- Otherwise, access to this field is W1C.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMOVSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b011

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMOVS == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMOVSR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMOVSR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMOVSR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMOVSR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b011

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGWTR_EL2().PMOVS == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMOVSR() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMOVSR() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMOVSR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMOVSR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.15 PMOVSSET, Performance Monitors Overflow Flag Status Set register

The PMOVSSET characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter PMCCNTR.
- The event counters PMEVCNTR<n>.

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to External register PMOVS[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET are UNDEFINED.

Attributes

PMOVSSET is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR set. On writes, allows software to set the unsigned overflow flag for PMCCNTR to 1. On reads, returns the unsigned overflow flag for PMCCNTR overflow status.

C	Meaning
0b0	PMCCNTR has not overflowed.
0b1	PMCCNTR has overflowed.

PMCR.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR[31:0] or unsigned overflow of PMCCNTR[63:0].

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.C == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.CR] == '11'
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m> set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m> to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m> overflow status.

P<m>	Meaning
`0b0`	PMEVCNTR<m> has not overflowed.
`0b1`	PMEVCNTR<m> has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP, HDCR.HLP, and PMCR.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>[31:0] or unsigned overflow of PMEVCNTR<m>[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - PMUSERENR_EL0.UEN == '1'
  - PMUACR_EL1.P<m> == '0'
- Access to this field is RO if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.ER] == '11'
- Otherwise, access to this field is W1S.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMOVSSET

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b011

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMOVS == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMOVSSET();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMOVSSET();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMOVSSET();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMOVSSET();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b011

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGWTR_EL2().PMOVS == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMOVSSET() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMOVSSET() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMOVSSET() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMOVSSET() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.16 PMSELR, Performance Monitors Event Counter Selection Register

The PMSELR characteristics are:

Purpose

Selects the current event counter PMEVCNTR<n> or the cycle counter PMCCNTR.

Used in conjunction with PMXEVTYPER to determine the event that increments a selected counter, and the modes and states in which the selected counter increments.

Used in conjunction with PMXEVCNTR to determine the value of a selected counter.

Configuration

AArch32 System register PMSELR bits [31:0] are architecturally mapped to AArch64 System register PMSELR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSELR are UNDEFINED.

Attributes

PMSELR is a 32-bit register.

Field descriptions

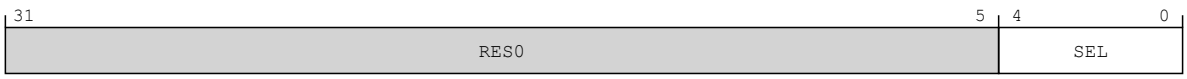

Bits [31:5]

Reserved, RES0.

SEL, bits [4:0]

Event counter select. Selects the counter accessed by subsequent accesses to PMXEVTYPER and PMXEVCNTR.

SEL	Meaning
0b00000..0b11110	Select event counter PMEVCNTR<n>, where n is the value of this field: • MRC and MCR of PMXEVTYPER access PMEVTYPER<n>. • MRC and MCR of PMXEVCNTR access PMEVCNTR<n>.
0b11111	Select the cycle counter, PMCCNTR: • MRC and MCR of PMXEVTYPER access PMCCFILTR. • MRC and MCR of PMXEVCNTR are CONSTRAINED UNPREDICTABLE. For more information, see PMXEVCNTR.

For more information about the results of accesses to the event counters, including when PMSELR.SEL is set to the index of an unimplemented or inaccessible event counter, see PMXEVTYPER and PMXEVCNTR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMSELR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b101

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[ER,EN] == '00' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMSELR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMSELR();
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMSELR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMSELR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMSELR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b101

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            else
                AArch64_AArch32SystemAccessTrap(EL1, 0x03);
            end;
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[ER,EN] == '00' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                 ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                 ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGWTR_EL2().PMSELR_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMSELR() = R(t);
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            PMSELR() = R(t);
        end;
    elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMSELR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSELR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.17 PMSWINC, Performance Monitors Software Increment register

The PMSWINC characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see 'SW_INCR'.

Configuration

AArch32 System register PMSWINC bits [31:0] are architecturally mapped to AArch64 System register PMSWINC_EL0[31:0].

When FEAT_PMUv3p9 is not implemented, AArch32 System register PMSWINC bits [31:0] are architecturally mapped to External register PMSWINC_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSWINC are UNDEFINED.

Attributes

PMSWINC is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0
RES0																															

Bit [31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Software increment.

P<m>	Meaning
`0b0`	Write is ignored.
`0b1`	Increment PMEVCNTR<m>, if PMEVCNTR<m> is configured to count software increment events.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3p9 is implemented
  - EL0 is the current Exception level
  - EL1 is using AArch64
  - [PMUSERENR_EL0.UEN, PMUSERENR_EL0.SW] == '10'
  - PMUACR_EL1.P<m> == '0'
- Otherwise, access to this field is WO/RAZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMSWINC

Accesses to this register use the following encodings in the System register encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b100

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,SW,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[SW,EN] == '00')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[SW,EN] == '00' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGWTR_EL2().PMSWINC_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMSWINC() = R(t);
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMSWINC() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMSWINC() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMSWINC() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.18 PMUSERENR, Performance Monitors User Enable Register

The PMUSERENR characteristics are:

Purpose

Enables or disables EL0 access to the Performance Monitors.

Configuration

AArch32 System register PMUSERENR bits [31:0] are architecturally mapped to AArch64 System register PMUSERENR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMUSERENR are UNDEFINED.

Attributes

PMUSERENR is a 32-bit register.

Field descriptions

31 … 7	6	5 … 4	3	2	1	0
RES0	TID	RES0	ER	CR	SW	EN

Bits [31:7]

Reserved, RES0.

TID, bit [6]

When FEAT_PMUv3p9 is implemented:

Trap ID registers. Traps EL0 read access to common event identification registers.

TID	Meaning
0b0	Accesses to PMCEID<n> are not trapped by this mechanism.
0b1	EL0 read accesses to PMCEID<n> are trapped.

The register accesses affected by this control are:

- MRC reads of PMCEID0, PMCEID1, PMCEID2, and PMCEID3.

When trapped, reads are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:4]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ER, bit [3]

Event counters Read enable.

When PMUSERENR.EN is 0, PMUSERENR.ER enables EL0 reads of the event counters and EL0 reads and writes of the select register.

ER	Meaning
0b0	EL0 reads of the event counters and EL0 reads and writes of the select register are disabled, unless enabled by PMUSERENR.EN.
0b1	EL0 reads of the event counters and EL0 reads and writes of the select register are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MRC reads of PMEVCNTR<n> and PMXEVCNTR.
- MRC and MCR accesses to PMSELR.

When disabled, reads and writes are UNDEFINED.

This field is ignored by the PE when PMUSERENR.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CR, bit [2]

Cycle counter Read enable.

When PMUSERENR.EN is 0, PMUSERENR.CR enables EL0 reads of the cycle counter.

CR	Meaning
0b0	EL0 reads of the cycle counter are disabled, unless enabled by PMUSERENR.EN.
0b1	EL0 reads of the cycle counter are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MRC reads of PMCCNTR.
- MRRC reads of PMCCNTR.

When disabled, reads are UNDEFINED.

This field is ignored by the PE when PMUSERENR.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SW, bit [1]

Software increment register Write enable.

When PMUSERENR.EN is 0, PMUSERENR.SW enables EL0 writes to the Software increment register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SW	Meaning
0b0	EL0 writes to the Software increment register are disabled, unless enabled by PMUSERENR.EN.
0b1	EL0 writes to the Software increment register are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MCR writes to PMSWINC.

When disabled, writes are UNDEFINED.

This field is ignored by the PE when PMUSERENR.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enable. Enables EL0 read/write access to PMU registers.

EN	Meaning
0b0	EL0 accesses to the specified PMU System registers are trapped, unless enabled by PMUSERENR.{ER,CR,SW}.
0b1	EL0 accesses to the specified PMU System registers are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MRC or MCR accesses to PMCCFILTR, PMCCNTR, PMCNTENCLR, PMCNTENSET, PMCR, PMEVCNTR<n>, PMEVTYPER<n>, PMOVSR, PMOVSSET, PMSELR, PMXEVCNTR, and PMXEVTYPER.
- MRC reads of the following registers:
  - PMCEID0 and PMCEID1.
  - If FEAT_PMUv3p1 is implemented, PMCEID2 and PMCEID3.
- MCR writes to PMSWINC.
- MRRC or MCRR accesses to PMCCNTR.

When trapped, reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMUSERENR

When FEAT_PMUv3p9 is implemented and EL1 is using AArch64, PMUSERENR_EL0 contains additional controls that affect the behavior of this register.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGRTR_EL2().PMUSERENR_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMUSERENR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMUSERENR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = PMUSERENR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = PMUSERENR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMUSERENR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMUSERENR() = R(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    PMUSERENR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.19 PMXEVCNTR, Performance Monitors Selected Event Count Register

The PMXEVCNTR characteristics are:

Purpose

Reads or writes the value of the selected event counter, PMEVCNTR<n>. PMSELR.SEL determines which event counter is selected.

Configuration

AArch32 System register PMXEVCNTR bits [31:0] are architecturally mapped to AArch64 System register PMXEVCNTR_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVCNTR are UNDEFINED.

Attributes

PMXEVCNTR is a 32-bit register.

Field descriptions

31	0
PMEVCNTR<n>	

PMEVCNTR<n>, bits [31:0]

Value of the selected event counter, PMEVCNTR<n>, where n is the value stored in PMSELR.SEL.

If FEAT_PMUv3p5 is implemented, the event counter is 64 bits and only the least-significant part of the event counter is accessible in AArch32 state:

- Reads from PMXEVCNTR return bits [31:0] of the counter.
- Writes to PMXEVCNTR update bits [31:0] and leave bits [63:32] unchanged.
- There is no means to access bits [63:32] directly from AArch32 state.
- If the implementation does not support AArch64, bits [63:32] are not required to be implemented.

If FEAT_PMUv3p5 is not implemented, the event counter is 32 bits.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMXEVCNTR

If FEAT_FGT is implemented and PMSELR.SEL is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVCNTR is as follows:

- If PMSELR.SEL is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and PMSELR.SEL is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVCNTR are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if PMSELR.SEL has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMXEVCNTR are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.UEN is 1.
- PMUACR_EL1.P<UInt(PMSELR.SEL)> is 0.

Permitted writes of PMXEVCNTR are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.{UEN,ER} is {1,1}.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.{ER,EN} or PMUSERENR_EL0.{UEN,ER,EN}.
If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is determined by the Effective value of PMCCR.EPMN. For more information, see HDCR.HPMN, MDCR_EL2.HPMN and PMCCR.EPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b010

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) &&
         ↪((IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[UEN,ER,EN] == '000') ||
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) && PMUSERENR_EL0().[ER,EN] == '00')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().[ER,EN] == '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGRTR_EL2().PMEVCNTRn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && PMUACR_EL1()[UInt(PMSELR().SEL)] ==
                ↪'0' then
                R(t) = Zeros{32};
            else
                R(t) = PMEVCNTR(UInt(PMSELR().SEL));
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMEVCNTR(UInt(PMSELR().SEL));
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = PMEVCNTR(UInt(PMSELR().SEL));
        end;
    elsif PSTATE.EL == EL3 then
        R(t) = PMEVCNTR(UInt(PMSELR().SEL));
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b010

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
            ↪'1') && HDFGWTR_EL2().PMEVCNTRn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && (PMUACR_EL1()[UInt(PMSELR().SEL)]
                ↪== '0' || PMUSERENR_EL0().ER == '1') then
                return;
            else
                PMEVCNTR()[UInt(PMSELR().SEL)] = R(t);
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
            ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMEVCNTR()[UInt(PMSELR().SEL)] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        PMEVCNTR()[UInt(PMSELR().SEL)] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    PMEVCNTR()[UInt(PMSELR().SEL)] = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.4.20 PMXEVTYPER, Performance Monitors Selected Event Type Register

The PMXEVTYPER characteristics are:

Purpose

When PMSELR.SEL selects an event counter, this accesses a PMEVTYPER<n> register. When PMSELR.SEL selects the cycle counter, this accesses PMCCFILTR.

Configuration

AArch32 System register PMXEVTYPER bits [31:0] are architecturally mapped to AArch64 System register PMXEVTYPER_EL0[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVTYPER are UNDEFINED.

Attributes

PMXEVTYPER is a 32-bit register.

Field descriptions

31	0
Event type register or PMCCFILTR	

ETR, bits [31:0]

Event type register or PMCCFILTR.

When PMSELR.SEL == 31, this register accesses PMCCFILTR.

Otherwise, this register accesses PMEVTYPER<n> where n is the value in PMSELR.SEL.

Accessing PMXEVTYPER

If FEAT_FGT is implemented, and PMSELR.SEL is not 31 and is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVTYPER is as follows:

- If PMSELR.SEL is greater than or equal to the Effective value of PMCCR.EPMN, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented, and PMSELR.SEL is not 31 and is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVTYPER are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if PMSELR.SEL has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Permitted reads and writes of PMXEVTYPER are RAZ/WI if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMUSERENR_EL0.UEN is 1.
- Any of the following are true:
  - PMSELR.SEL is not '31' and PMUACR_EL1.P<UInt(PMSELR.SEL)> is 0.
  - PMSELR.SEL is '31' and PMUACR_EL1.C is 0.

Permitted writes of PMXEVTYPER are ignored if all of the following are true:

- FEAT_PMUv3p9 is implemented.
- EL0 is the current Exception level.
- EL1 is using AArch64.
- PMUSERENR_EL0.UEN is 1.
- Any of the following are true:
  - PMSELR.SEL is not 31 and PMUSERENR_EL0.ER is 1.
  - PMSELR.SEL is 31 and PMUSERENR_EL0.CR is 1.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.EN or PMUSERENR_EL0.{UEN,EN}.
If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is determined by the Effective value of PMCCR.EPMN. For more information, see HDCR.HPMN, MDCR_EL2.HPMN and PMCCR.EPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b001

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
             ↪'1') && HDFGRTR_EL2().PMEVTYPERn_EL0 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
             ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
             ↪GetNumEventCountersAccessible() then
            if !IsFeatureImplemented(FEAT_FGT) then
                ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
            elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
                AArch32_TakeHypTrapException(0x03);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            end;
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
             ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
                 ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && ((UInt(PMSELR().SEL) != 31 &&
                 ↪PMUACR_EL1()[UInt(PMSELR().SEL)] == '0') || (UInt(PMSELR().SEL) == 31 && PMUACR_EL1().C
                 ↪== '0')) then
                R(t) = Zeros{32};
            elsif UInt(PMSELR().SEL) == 31 then
                R(t) = PMCCFILTR();
            else
                R(t) = PMEVTYPER(UInt(PMSELR().SEL));
            end;
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
             ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
             ↪'1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
             ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
             ↪GetNumEventCountersAccessible() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if UInt(PMSELR().SEL) == 31 then
            R(t) = PMCCFILTR();
        else
            R(t) = PMEVTYPER(UInt(PMSELR().SEL));
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if UInt(PMSELR().SEL) == 31 then
            R(t) = PMCCFILTR();
        else
            R(t) = PMEVTYPER(UInt(PMSELR().SEL));
        end;
    end;
elsif PSTATE.EL == EL3 then
    if UInt(PMSELR().SEL) == 31 then
        R(t) = PMCCFILTR();
    else
        R(t) = PMEVTYPER(UInt(PMSELR().SEL));
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b001

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_PMUv3)) then
    Undefined();
elsif UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    if IsFeatureImplemented(FEAT_FGT) then
        Undefined();
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && (PMUSERENR_EL0().EN == '0' &&
         ↪(!IsFeatureImplemented(FEAT_PMUv3p9) || PMUSERENR_EL0().UEN == '0')) then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && PMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T9 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T9 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HDFGWTR_EL2().PMEVTYPERn_EL0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && MDCR_EL2().TPM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HDCR().TPM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
         ↪GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_AA64EL1) &&
             ↪!ELUsingAArch32(EL2) && PMUSERENR_EL0().UEN == '1' && ((UInt(PMSELR().SEL) != 31 &&
             ↪(PMUACR_EL1()[UInt(PMSELR().SEL)] == '0' || PMUSERENR_EL0().ER == '1')) ||
             ↪(UInt(PMSELR().SEL) == 31 && (PMUACR_EL1().C == '0' || PMUSERENR_EL0().CR == '1'))) then
            return;
        elsif UInt(PMSELR().SEL) == 31 then
            PMCCFILTR() = R(t);
        else
            PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T9 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T9 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && MDCR_EL2().TPM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HDCR().TPM == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
        ↪GetNumEventCountersAccessible() then
        if !IsFeatureImplemented(FEAT_FGT) then
            ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
        elsif IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            AArch32_TakeHypTrapException(0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if UInt(PMSELR().SEL) == 31 then
            PMCCFILTR() = R(t);
        else
            PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && MDCR_EL3().TPM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && MDCR_EL3().TPM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        if UInt(PMSELR().SEL) == 31 then
            PMCCFILTR() = R(t);
        else
            PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
        end;
    end;
elsif PSTATE.EL == EL3 then
    if UInt(PMSELR().SEL) == 31 then
        PMCCFILTR() = R(t);
    else
        PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5 Activity Monitors registers

This section lists the Activity Monitoring registers in AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

Configuration

AArch32 System register AMCFGR bits [31:0] are architecturally mapped to AArch64 System register AMCFGR_EL0[31:0].

AArch32 System register AMCFGR bits [31:0] are architecturally mapped to External register AMCFGR[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCFGR are UNDEFINED.

Attributes

AMCFGR is a 32-bit register.

Field descriptions

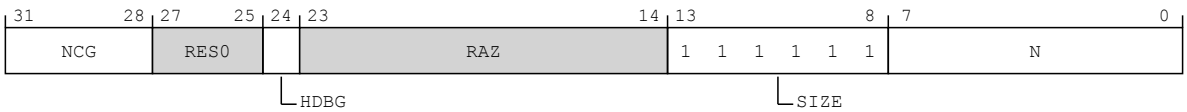

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

Access to this field is RO.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

This feature must be supported, and so this bit is 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HDBG	Meaning
0b0	AMCR.HDBG is RES0.
0b1	AMCR.HDBG is read/write.

Access to this field is RO.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of the activity monitor event counters, minus one.

The counters are 64-bit, so the value of this field is 0b111111.

This field is used by software to determine the spacing of the counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as 0b111111

Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters implemented in all groups. One less than the sum of AMCGCR.CG1NC and AMCGCR.CG0NC.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMCFGR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b001

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCFGR();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCFGR();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCFGR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMCFGR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configuration

AArch32 System register AMCGCR bits [31:0] are architecturally mapped to AArch64 System register AMCGCR_EL0[31:0].

AArch32 System register AMCGCR bits [31:0] are architecturally mapped to External register AMCGCR[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCGCR are UNDEFINED.

Attributes

AMCGCR is a 32-bit register.

Field descriptions

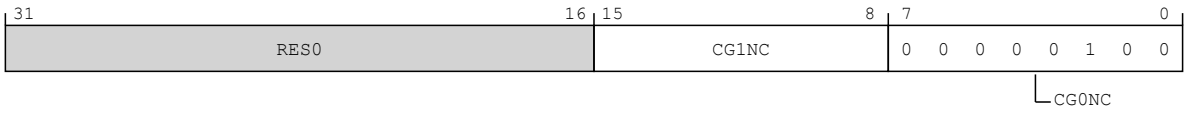

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. Highest index of the implemented counters in the auxiliary counter group, plus one. Zero indicates that there are no counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16.

If FEAT_AMUv1p1 is implemented, then AMCG1IDR_EL0 further defines which auxiliary counters are implemented, and the counters in the auxiliary counter group might be indexed non-contiguously.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as `0x04`

Access to this field is RO.

Accessing AMCGCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b010

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCGCR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCGCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCGCR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMCGCR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.3 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

Purpose

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configuration

AArch32 System register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

AArch32 System register AMCNTENCLR0 bits [31:0] are architecturally mapped to External register AMCNTENCLR0[31:0].

AArch32 System register AMCNTENCLR0 bits [31:0] are architecturally mapped to External register AMCNTENSET0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCNTENCLR0 are UNDEFINED.

Attributes

AMCNTENCLR0 is a 32-bit register.

Field descriptions

31 … 16	15 … 4	3	2	1	0
RES0	RAZ/WI	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	When read, means that AMEVCNTR0<n> is enabled. When written, disables AMEVCNTR0<n>.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b100

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HAFGRTR_EL2().AMCNTEN0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCNTENCLR0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCNTENCLR0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCNTENCLR0();
        end;
    elsif PSTATE.EL == EL3 then
        R(t) = AMCNTENCLR0();
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪HSTR_EL2().T13 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
     ↪HSTR().T13 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENCLR0() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.4 AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configuration

AArch32 System register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

AArch32 System register AMCNTENCLR1 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

AArch32 System register AMCNTENCLR1 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCNTENCLR1 are UNDEFINED.

Attributes

AMCNTENCLR1 is a 32-bit register.

Field descriptions

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.
0b1	When read, means that AMEVCNTR1<n> is enabled. When written, disables AMEVCNTR1<n>.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= UInt(AMCGCR.CG1NC), access to this field is RAZ/WI.
- When !IsG1ActivityMonitorImplemented(n), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMCNTENCLR1

If there are no auxiliary monitor event counters implemented, reads and writes of AMCNTENCLR1 are UNDEFINED.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HAFGRTR_EL2().AMCNTEN1 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = AMCNTENCLR1();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCNTENCLR1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCNTENCLR1();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMCNTENCLR1();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪HSTR_EL2().T13 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
     ↪HSTR().T13 == '1' then
    AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENCLR1() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.5 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configuration

AArch32 System register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

AArch32 System register AMCNTENSET0 bits [31:0] are architecturally mapped to External register AMCNTENSET0[31:0].

AArch32 System register AMCNTENSET0 bits [31:0] are architecturally mapped to External register AMCNTENCLR0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCNTENSET0 are UNDEFINED.

Attributes

AMCNTENSET0 is a 32-bit register.

Field descriptions

31 … 16	15 … 4	3	2	1	0
RES0	RAZ/WI	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	When read, means that AMEVCNTR0<n> is enabled. When written, enables AMEVCNTR0<n>.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET0

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b101

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HAFGRTR_EL2().AMCNTEN0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCNTENSET0();
        end;
    elsif PSTATE.EL == EL1 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
            Undefined();
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
            ↪then
            AArch32_TakeHypTrapException(0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
            ↪'1' then
            AArch32_TakeHypTrapException(0x03);
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCNTENSET0();
        end;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
            ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
            Undefined();
        elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
            ↪'1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AArch32SystemAccessTrap(EL3, 0x03);
            end;
        else
            R(t) = AMCNTENSET0();
        end;
    elsif PSTATE.EL == EL3 then
        R(t) = AMCNTENSET0();
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b101

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪HSTR_EL2().T13 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
     ↪HSTR().T13 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENSET0() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.6 AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configuration

AArch32 System register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

AArch32 System register AMCNTENSET1 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

AArch32 System register AMCNTENSET1 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCNTENSET1 are UNDEFINED.

Attributes

AMCNTENSET1 is a 32-bit register.

Field descriptions

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.
0b1	When read, means that AMEVCNTR1<n> is enabled. When written, enables AMEVCNTR1<n>.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= UInt(AMCGCR.CG1NC), access to this field is RAZ/WI.
- When !IsG1ActivityMonitorImplemented(n), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMCNTENSET1

If there are no auxiliary monitor event counters implemented, reads and writes of AMCNTENSET1 are UNDEFINED.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HAFGRTR_EL2().AMCNTEN1 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = AMCNTENSET1();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCNTENSET1();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCNTENSET1();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMCNTENSET1();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪HSTR_EL2().T13 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
     ↪HSTR().T13 == '1' then
    AArch32_TakeHypTrapException(0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif IsHighestEL(PSTATE.EL) then
    AMCNTENSET1() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.7 AMCR, Activity Monitors Control Register

The AMCR characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR is applicable to both the architected and the auxiliary counter groups.

Configuration

AArch32 System register AMCR bits [31:0] are architecturally mapped to AArch64 System register AMCR_EL0[31:0].

AArch32 System register AMCR bits [31:0] are architecturally mapped to External register AMCR[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMCR are UNDEFINED.

Attributes

AMCR is a 32-bit register.

Field descriptions

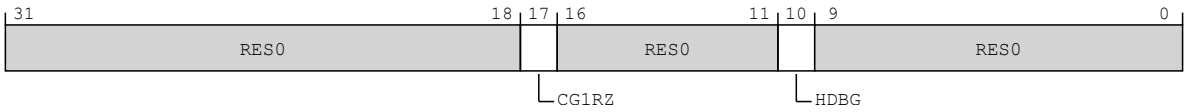

Bits [31:18]

Reserved, RES0.

CG1RZ, bit [17]

When FEAT_AMUv1p1 is implemented:

Counter Group 1 Read Zero.

CG1RZ	Meaning
0b0	System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.
0b1	If the current Exception level is the highest implemented Exception level, System register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

Note

Reads from the memory-mapped view are unaffected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

HDBG	Meaning
0b0	Activity monitors do not halt counting when the PE is halted in Debug state.
0b1	Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing AMCR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b000

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMCR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMCR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b000

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪HSTR_EL2().T13 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
     ↪HSTR().T13 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
    AMCR() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.8 AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 3

The AMEVCNTR0<n> characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configuration

AArch32 System register AMEVCNTR0<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR0<n>_EL0[63:0].

AArch32 System register AMEVCNTR0<n> bits [63:0] are architecturally mapped to External register AMEVCNTR0<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMEVCNTR0<n> are UNDEFINED.

Attributes

AMEVCNTR0<n> is a 64-bit register.

Field descriptions

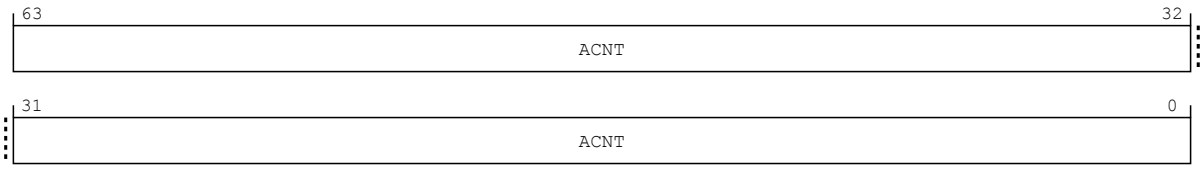

ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 3.

If FEAT_AMUv1p1 is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, and EL2 is using AArch64 and is implemented in the current Security state, access to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF0<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR0<n> is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x0000000000000000`.

Accessing AMEVCNTR0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> are UNDEFINED.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm> ; Where m = 0-3
```

coproc	CRm	opc1
0b1111	0b000:m[3]	0b0:m[2:0]

```
let m:integer = UInt(CRm[0] :: opc1[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && m < 8 && HSTR_EL2().T0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && m < 8 &&
        ↪HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HAFGRTR_EL2()[m + 1] == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        R(t, t2) = AMEVCNTR0(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && m < 8 &&
         ↪HSTR_EL2().T0 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && m < 8 &&
         ↪HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        R(t, t2) = AMEVCNTR0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    else
        R(t, t2) = AMEVCNTR0(m);
    end;
elsif PSTATE.EL == EL3 then
    R(t, t2) = AMEVCNTR0(m);
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm> ; Where m = 0-3
```

coproc	CRm	opc1
0b1111	0b000:m[3]	0b0:m[2:0]

```
let m:integer = UInt(CRm[0] :: opc1[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪m < 8 && HSTR_EL2().T0 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x04);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && m
     ↪< 8 && HSTR().T0 == '1' then
    AArch32_TakeHypTrapException(0x04);
elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR0(m) = R(t2) :: R(t);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.9 AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n> characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configuration

AArch32 System register AMEVCNTR1<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR1<n>_EL0[63:0].

AArch32 System register AMEVCNTR1<n> bits [63:0] are architecturally mapped to External register AMEVCNTR1<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMEVCNTR1<n> are UNDEFINED.

Attributes

AMEVCNTR1<n> is a 64-bit register.

Field descriptions

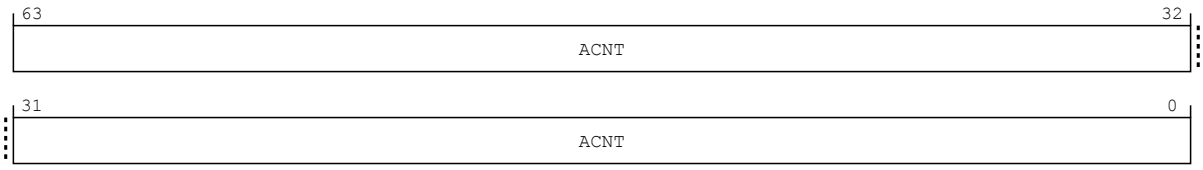

ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If FEAT_AMUv1p1 is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}, EL2 is using AArch64 and is implemented in the current Security state, and AMCR_EL0.CG1RZ is 0, reads to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF1<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR1<n> is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x0000000000000000`.

Accessing AMEVCNTR1<n>

If <n> is greater than or equal to AMCGCR.CG1NC or FEAT_AMUv1p1 is implemented and auxiliary counter <n> is not implemented, then reads and writes of AMEVCNTR1<n> are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm> ; Where m = 0-15
```

coproc	CRm	opc1
0b1111	0b010:m[3]	0b0:m[2:0]

```
let m:integer = UInt(CRm[0] :: opc1[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && m >= 8 && HSTR_EL2().T5 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && m >= 8 &&
         ↪HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
         ↪'1') && HAFGRTR_EL2()[(2 * m) + 18] == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA64) && AMCR_EL0().CG1RZ == '1' then
        R(t, t2) = Zeros{64};
    elsif !IsFeatureImplemented(FEAT_AA64) && AMCR().CG1RZ == '1' then
        R(t, t2) = Zeros{64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t, t2) = AMEVCNTR1(m);
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && m >= 8 &&
         ↪HSTR_EL2().T5 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && m >= 8 &&
         ↪HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    elsif !IsHighestEL(PSTATE.EL) && IsFeatureImplemented(FEAT_AA64) && AMCR_EL0().CG1RZ == '1' then
        R(t, t2) = Zeros{64};
    elsif !IsHighestEL(PSTATE.EL) && !IsFeatureImplemented(FEAT_AA64) && AMCR().CG1RZ == '1' then
        R(t, t2) = Zeros{64};
    else
        R(t, t2) = AMEVCNTR1(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x04);
        end;
    elsif !IsHighestEL(PSTATE.EL) && IsFeatureImplemented(FEAT_AA64) && AMCR_EL0().CG1RZ == '1' then
        R(t, t2) = Zeros{64};
    elsif !IsHighestEL(PSTATE.EL) && !IsFeatureImplemented(FEAT_AA64) && AMCR().CG1RZ == '1' then
        R(t, t2) = Zeros{64};
    else
        R(t, t2) = AMEVCNTR1(m);
    end;
elsif PSTATE.EL == EL3 then
    R(t, t2) = AMEVCNTR1(m);
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm> ; Where m = 0-15
```

coproc	CRm	opc1
0b1111	0b010:m[3]	0b0:m[2:0]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let m:integer = UInt(CRm[0] :: opc1[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪m >= 8 && HSTR_EL2().T5 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x04);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && m
     ↪>= 8 && HSTR().T5 == '1' then
    AArch32_TakeHypTrapException(0x04);
elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR1(m) = R(t2) :: R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.10 AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n> characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

Configuration

AArch32 System register AMEVTYPER0<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[31:0].

AArch32 System register AMEVTYPER0<n> bits [31:0] are architecturally mapped to External register AMEVTYPER0<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are UNDEFINED.

Attributes

AMEVTYPER0<n> is a 32-bit register.

Field descriptions

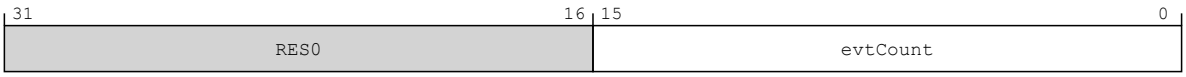

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

evtCount	Meaning	Applies when
0x0011	Processor frequency cycles.	n == 0
0x4004	Constant frequency cycles.	n == 1
0x0008	Instructions retired.	n == 2
0x4005	Memory stall cycles.	n == 3

Access to this field is RO.

Accessing AMEVTYPER0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n> are UNDEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-3
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b011:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= 4 then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMEVTYPER0(m);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMEVTYPER0(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMEVTYPER0(m);
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMEVTYPER0(m);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.11 AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n> characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n> counts.

Configuration

AArch32 System register AMEVTYPER1<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER1<n>_EL0[31:0].

AArch32 System register AMEVTYPER1<n> bits [31:0] are architecturally mapped to External register AMEVTYPER1<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMEVTYPER1<n> are UNDEFINED.

Attributes

AMEVTYPER1<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1<n>, then:

- It is UNPREDICTABLE which event will be counted.
- The value read back is UNKNOWN.

The event counted by AMEVCNTR1<n> might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED.

If the corresponding counter AMEVCNTR1<n> is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMEVTYPER1<n>

If <n> is greater than or equal to AMCGCR.CG1NC or FEAT_AMUv1p1 is implemented and auxiliary counter <n> is not implemented, then reads and writes of AMEVTYPER1<n> are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b111:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && AMUSERENR_EL0().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && AMUSERENR().EN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn ==
        ↪'1') && HAFGRTR_EL2()[(2 * m) + 19] == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMEVTYPER1(m);
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMEVTYPER1(m);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMEVTYPER1(m);
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMEVTYPER1(m);
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b111:m[3]	m[2:0]

```
let m:integer = UInt(CRm[0] :: opc2[2:0]);

if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif m >= NUM_AMU_CG1_MONITORS then
    Undefined();
elsif !IsG1ActivityMonitorImplemented(m) then
    Undefined();
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪HSTR_EL2().T13 == '1' then
    AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 && EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
     ↪HSTR().T13 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) && !ImpDefBool("AMEVCNTR1()[m] is fixed") then
    AMEVTYPER1(m) = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.5.12 AMUSERENR, Activity Monitors User Enable Register

The AMUSERENR characteristics are:

Purpose

Global user enable register for the activity monitors. Enables or disables EL0 access to the activity monitors. AMUSERENR is applicable to both the architected and the auxiliary counter groups.

Configuration

AArch32 System register AMUSERENR bits [31:0] are architecturally mapped to AArch64 System register AMUSERENR_EL0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AA32 is implemented. Otherwise, direct accesses to AMUSERENR are UNDEFINED.

Attributes

AMUSERENR is a 32-bit register.

Field descriptions

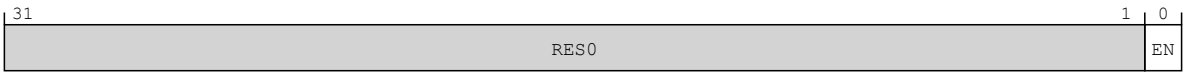

Bits [31:1]

Reserved, RES0.

EN, bit [0]

Traps EL0 accesses to the activity monitors registers to EL1.

EN	Meaning
0b0	EL0 accesses to the activity monitors registers are trapped to EL1.
0b1	This control does not cause any instructions to be trapped. Software can access all activity monitor registers at EL0.

Note

- AMUSERENR can always be read at EL0 and is not governed by this bit.

Note

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMUSERENR

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && HSTR_EL2().T13 == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HSTR().T13 ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMUSERENR();
    end;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
        ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
        ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
        ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMUSERENR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        R(t) = AMUSERENR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = AMUSERENR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_AMUv1) && IsFeatureImplemented(FEAT_AA32)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T13
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T13 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && CPTR_EL2().TAM
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCPTR().TAM ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    else
        AMUSERENR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && CPTR_EL3().TAM == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && CPTR_EL3().TAM ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        AMUSERENR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    AMUSERENR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6 RAS registers

This section lists RAS Extension registers in AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.1 DISR, Deferred Interrupt Status Register

The DISR characteristics are:

Purpose

Records that an SError exception has been consumed by an `ESB` instruction.

Configuration

AArch32 System register DISR bits [31:0] are architecturally mapped to AArch64 System register DISR_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to DISR are UNDEFINED.

Attributes

DISR is a 32-bit register.

Field descriptions

When the ESB instruction is executed at EL2:

31	30 … 12	11 10	9	8 … 6	5 … 0
A	RES0	AET	EA	RES0	DFSC

A, bit [31]

Set to 1 when an `ESB` instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:12]

Reserved, RES0.

AET, bits [11:10]

Asynchronous Error Type. See the description of HSR.AET for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort Type. See the description of HSR.EA for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

Fault Status Code. See the description of HSR.DFSC for an SError exception.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When the ESB instruction is executed at EL0 or EL1 and TTBCR.EAE == '0':

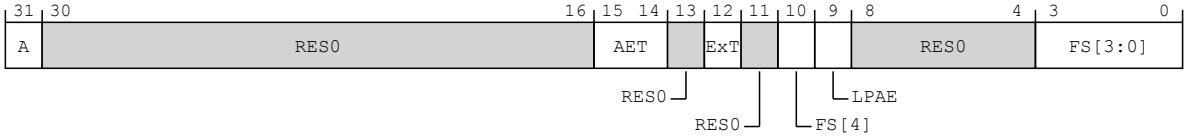

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

Asynchronous Error Type. See the description of DFSR.AET for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

External abort Type. See the description of DFSR.ExT for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]

Fault Status Code. See the description of DFSR.FS for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

Format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LPAE	Meaning
0b0	Using the Short-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

When the ESB instruction is executed at EL0 or EL1 and TTBCR.EAE == '1':

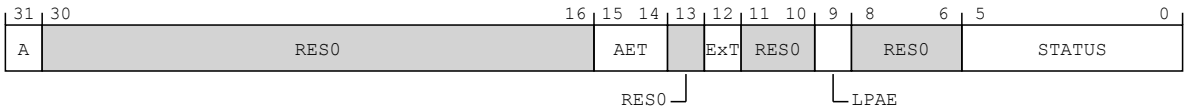

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

Asynchronous Error Type. See the description of DFSR.AET for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

External abort Type. See the description of DFSR.ExT for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LPAE	Meaning
0b1	Using the Long-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault Status Code. See the description of DFSR.FS for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DISR

An indirect write to DISR made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR occurring in program order after the ESB instruction.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && (HCR_EL2().AMO
        ↪== '1' || (IsFeatureImplemented(FEAT_DoubleFault2) && IsHCRXEL2Enabled() && HCRX_EL2().TMEA
        ↪== '1')) then
        R(t) = VDISR_EL2()[31:0];
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().AMO == '1'
        ↪then
        R(t) = VDISR();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        R(t) = Zeros{32};
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        R(t) = Zeros{32};
    else
        R(t) = DISR();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        R(t) = Zeros{32};
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        R(t) = Zeros{32};
    else
        R(t) = DISR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DISR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && (HCR_EL2().AMO
        ↪== '1' || (IsFeatureImplemented(FEAT_DoubleFault2) && IsHCRXEL2Enabled() && HCRX_EL2().TMEA
        ↪== '1')) then
        VDISR_EL2()[31:0] = R(t);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().AMO == '1'
        ↪then
        VDISR() = R(t);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        return;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        return;
    else
        DISR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        return;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        return;
    else
        DISR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DISR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.2 ERRIDR, Error Record ID Register

The ERRIDR characteristics are:

Purpose

Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

Configuration

AArch32 System register ERRIDR bits [31:0] are architecturally mapped to AArch64 System register ERRIDR_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERRIDR are UNDEFINED.

Attributes

ERRIDR is a 32-bit register.

Field descriptions

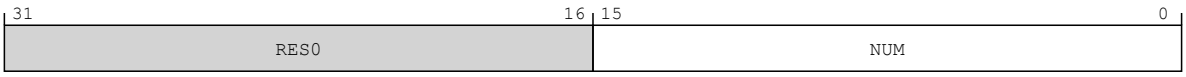

Bits [31:16]

Reserved, RES0.

NUM, bits [15:0]

Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates that no records can be accessed through the Error Record System registers.

Each implemented record is owned by a node. A node might own multiple records.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing ERRIDR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0011	0b000

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRIDR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRIDR();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRIDR();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.3 ERRSELR, Error Record Select Register

The ERRSELR characteristics are:

Purpose

Selects an error record to be accessed through the Error Record System registers.

Configuration

If ERRIDR indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR is UNDEFINED or RES0.

AArch32 System register ERRSELR bits [31:0] are architecturally mapped to AArch64 System register ERRSELR_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERRSELR are UNDEFINED.

Attributes

ERRSELR is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	SEL

Bits [31:16]

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR.SEL is `0x0004`, then direct reads and writes of ERXSTATUS access ERR4STATUS.

If ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then all of the following apply:

- The value read back from ERRSELR.SEL is UNKNOWN.
- One of the following occurs:
  - An UNKNOWN error record is selected.
  - The ERX* registers are RAZ/WI.
  - ERX* register reads and writes are NOPs.
  - ERX* register reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ERRSELR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRSELR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        R(t) = ERRSELR();
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRSELR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0011	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRSELR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRSELR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRSELR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.4 ERXADDR, Selected Error Record Address Register

The ERXADDR characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>ADDR for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXADDR bits [31:0] are architecturally mapped to AArch64 System register ERXADDR_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXADDR are UNDEFINED.

Attributes

ERXADDR is a 32-bit register.

Field descriptions

31	0
Bits [31:0] of ERRnADDR	

ERRnADDRlo, bits [31:0]

ERXADDR accesses bits [31:0] of ERR<n>ADDR, where <n> is the value in ERRSELR.SEL.

Accessing ERXADDR

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR is RAZ/WI.
- Direct reads and writes of ERXADDR are NOPs.
- Direct reads and writes of ERXADDR are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRADDR(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRADDR(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRADDR(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRADDR(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRADDR(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRADDR(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.5 ERXADDR2, Selected Error Record Address Register 2

The ERXADDR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>ADDR for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXADDR2 bits [31:0] are architecturally mapped to AArch64 System register ERXADDR_EL1[63:32].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXADDR2 are UNDEFINED.

Attributes

ERXADDR2 is a 32-bit register.

Field descriptions

31 … 0
Bits [63:32] of ERRnADDR

ERRnADDRhi, bits [31:0]

ERXADDR2 accesses bits [63:32] of ERR<n>ADDR, where <n> is the value in ERRSELR.SEL.

Accessing ERXADDR2

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR2 is RAZ/WI.
- Direct reads and writes of ERXADDR2 are NOPs.
- Direct reads and writes of ERXADDR2 are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRADDR(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRADDR(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRADDR(UInt(ERRSELR().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRADDR(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRADDR(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRADDR(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.6 ERXCTLR, Selected Error Record Control Register

The ERXCTLR characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>CTLR for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXCTLR bits [31:0] are architecturally mapped to AArch64 System register ERXCTLR_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXCTLR are UNDEFINED.

Attributes

ERXCTLR is a 32-bit register.

Field descriptions

31	0
Bits [31:0] of ERRnCTLR	

ERRnCTLRlo, bits [31:0]

ERXCTLR accesses bits [31:0] of ERR<n>CTLR, where <n> is the value in ERRSELR.SEL.

Accessing ERXCTLR

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR is RAZ/WI.
- Direct reads and writes of ERXCTLR are NOPs.
- Direct reads and writes of ERXCTLR are UNDEFINED.

If ERRSELR.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR[31:0] is not present, meaning reads and writes of ERXCTLR are RES0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRCTLR(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRCTLR(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRCTLR(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRCTLR(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRCTLR(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRCTLR(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.7 ERXCTLR2, Selected Error Record Control Register 2

The ERXCTLR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>CTLR for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXCTLR2 bits [31:0] are architecturally mapped to AArch64 System register ERXCTLR_EL1[63:32].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXCTLR2 are UNDEFINED.

Attributes

ERXCTLR2 is a 32-bit register.

Field descriptions

31	0
Bits [63:32] of ERRnCTLR	

ERRnCTLRhi, bits [31:0]

ERXCTLR2 accesses bits [63:32] of ERR<n>CTLR, where <n> is the value in ERRSELR.SEL.

Accessing ERXCTLR2

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR2 is RAZ/WI.
- Direct reads and writes of ERXCTLR2 are NOPs.
- Direct reads and writes of ERXCTLR2 are UNDEFINED.

If ERRSELR.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR[63:32] is not present, meaning reads and writes of ERXCTLR2 are RES0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b101

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRCTLR(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRCTLR(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRCTLR(UInt(ERRSELR().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b101

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRCTLR(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRCTLR(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRCTLR(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.8 ERXFR, Selected Error Record Feature Register

The ERXFR characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>FR for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXFR bits [31:0] are architecturally mapped to AArch64 System register ERXFR_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXFR are UNDEFINED.

Attributes

ERXFR is a 32-bit register.

Field descriptions

31	0
Bits [31:0] of ERRnFR	

ERRnFRlo, bits [31:0]

ERXFR accesses bits [31:0] of ERR<n>FR, where <n> is the value in ERRSELR.SEL.

Accessing ERXFR

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR is RAZ.
- Direct reads of ERXFR are NOPs.
- Direct reads of ERXFR are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b000

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRFR(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRFR(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRFR(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.9 ERXFR2, Selected Error Record Feature Register 2

The ERXFR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>FR for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXFR2 bits [31:0] are architecturally mapped to AArch64 System register ERXFR_EL1[63:32].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXFR2 are UNDEFINED.

Attributes

ERXFR2 is a 32-bit register.

Field descriptions

31 … 0
Bits [63:32] of ERRnFR

ERRnFRhi, bits [31:0]

ERXFR2 accesses bits [63:32] of ERR<n>FR, where <n> is the value in ERRSELR.SEL.

Accessing ERXFR2

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR2 is RAZ.
- Direct reads of ERXFR2 are NOPs.
- Direct reads of ERXFR2 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b100

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRFR(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRFR(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRFR(UInt(ERRSELR().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.10 ERXMISC0, Selected Error Record Miscellaneous Register 0

The ERXMISC0 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC0 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC0 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC0_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC0 are UNDEFINED.

Attributes

ERXMISC0 is a 32-bit register.

Field descriptions

ERRnMISC0lo

31	0
Bits [31:0] of ERRnMISC0	

ERRnMISC0lo, bits [31:0]

ERXMISC0 accesses bits [31:0] of ERR<n>MISC0, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC0

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0 is RAZ/WI.
- Direct reads and writes of ERXMISC0 are NOPs.
- Direct reads and writes of ERXMISC0 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b000

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b000

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.11 ERXMISC1, Selected Error Record Miscellaneous Register 1

The ERXMISC1 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC0 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC1 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC0_EL1[63:32].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC1 are UNDEFINED.

Attributes

ERXMISC1 is a 32-bit register.

Field descriptions

31	0
Bits [63:32] of ERRnMISC0	

ERRnMISC0hi, bits [31:0]

ERXMISC1 accesses bits [63:32] of ERR<n>MISC0, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC1

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC1 is RAZ/WI.
- Direct reads and writes of ERXMISC1 are NOPs.
- Direct reads and writes of ERXMISC1 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC0(UInt(ERRSELR_EL1().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC0(UInt(ERRSELR_EL1().SEL))[63:32] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.12 ERXMISC2, Selected Error Record Miscellaneous Register 2

The ERXMISC2 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC1 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC2 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC1_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC2 are UNDEFINED.

Attributes

ERXMISC2 is a 32-bit register.

Field descriptions

31	0
Bits [31:0] of ERRnMISC1	

ERRnMISC1lo, bits [31:0]

ERXMISC2 accesses bits [31:0] of ERR<n>MISC1, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC2

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2 is RAZ/WI.
- Direct reads and writes of ERXMISC2 are NOPs.
- Direct reads and writes of ERXMISC2 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b100

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC1(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC1(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC1(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b100

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC1(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC1(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC1(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.13 ERXMISC3, Selected Error Record Miscellaneous Register 3

The ERXMISC3 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC1 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC3 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC1_EL1[63:32].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC3 are UNDEFINED.

Attributes

ERXMISC3 is a 32-bit register.

Field descriptions

31	0
Bits [63:32] of ERRnMISC1	

ERRnMISC1hi, bits [31:0]

ERXMISC3 accesses bits [63:32] of ERR<n>MISC1, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC3

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3 is RAZ/WI.
- Direct reads and writes of ERXMISC3 are NOPs.
- Direct reads and writes of ERXMISC3 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC3.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC1(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC1(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC1(UInt(ERRSELR().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC1(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC1(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC1(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.14 ERXMISC4, Selected Error Record Miscellaneous Register 4

The ERXMISC4 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC2 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC4 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC2_EL1[31:0].

This register is present only when FEAT_RASv1p1 is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC4 are UNDEFINED.

Attributes

ERXMISC4 is a 32-bit register.

Field descriptions

31	0
Bits [31:0] of ERRnMISC2	

ERRnMISC2lo, bits [31:0]

ERXMISC4 accesses bits [31:0] of ERR<n>MISC2, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC4

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC4 is RAZ/WI.
- Direct reads and writes of ERXMISC4 are NOPs.
- Direct reads and writes of ERXMISC4 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC4.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC2(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC2(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC2(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b010

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC2(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC2(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC2(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.15 ERXMISC5, Selected Error Record Miscellaneous Register 5

The ERXMISC5 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC2 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC5 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC2_EL1[63:32].

This register is present only when FEAT_RASv1p1 is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC5 are UNDEFINED.

Attributes

ERXMISC5 is a 32-bit register.

Field descriptions

31	0
Bits [63:32] of ERRnMISC2	

ERRnMISC2hi, bits [31:0]

ERXMISC5 accesses bits [63:32] of ERR<n>MISC2, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC5

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC5 is RAZ/WI.
- Direct reads and writes of ERXMISC5 are NOPs.
- Direct reads and writes of ERXMISC5 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC5.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC2(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC2(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC2(UInt(ERRSELR().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b011

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC2(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC2(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC2(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.16 ERXMISC6, Selected Error Record Miscellaneous Register 6

The ERXMISC6 characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>MISC3 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC6 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC3_EL1[31:0].

This register is present only when FEAT_RASv1p1 is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC6 are UNDEFINED.

Attributes

ERXMISC6 is a 32-bit register.

Field descriptions

31	0
Bits [31:0] of ERRnMISC3	

ERRnMISC3lo, bits [31:0]

ERXMISC6 accesses bits [31:0] of ERR<n>MISC3, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC6

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC6 is RAZ/WI.
- Direct reads and writes of ERXMISC6 are NOPs.
- Direct reads and writes of ERXMISC6 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC6.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b110

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC3(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC3(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC3(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b110

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC3(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC3(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC3(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.17 ERXMISC7, Selected Error Record Miscellaneous Register 7

The ERXMISC7 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC3 for the error record <n> selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXMISC7 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC3_EL1[63:32].

This register is present only when FEAT_RASv1p1 is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXMISC7 are UNDEFINED.

Attributes

ERXMISC7 is a 32-bit register.

Field descriptions

31	0
Bits [63:32] of ERRnMISC3	

ERRnMISC3hi, bits [31:0]

ERXMISC7 accesses bits [63:32] of ERR<n>MISC3, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC7

If ERRIDR.NUM is `0x0000` or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC7 is RAZ/WI.
- Direct reads and writes of ERXMISC7 are NOPs.
- Direct reads and writes of ERXMISC7 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC7.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC3(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRMISC3(UInt(ERRSELR().SEL))[63:32];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRMISC3(UInt(ERRSELR().SEL))[63:32];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0101	0b111

```
if !(IsFeatureImplemented(FEAT_RASv1p1) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC3(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRMISC3(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRMISC3(UInt(ERRSELR().SEL))[63:32] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.18 ERXSTATUS, Selected Error Record Primary Status Register

The ERXSTATUS characteristics are:

Purpose

Accesses bits [31:0] of ERR<n>STATUS for the error record selected by ERRSELR.SEL.

Configuration

AArch32 System register ERXSTATUS bits [31:0] are architecturally mapped to AArch64 System register ERXSTATUS_EL1[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to ERXSTATUS are UNDEFINED.

Attributes

ERXSTATUS is a 32-bit register.

Field descriptions

31 … 0
Bits [31:0] of ERRnSTATUS

ERRnSTATUSlo, bits [31:0]

ERXSTATUS accesses bits [31:0] of ERR<n>STATUS, where n is the value in ERRSELR.SEL.

Accessing ERXSTATUS

If ERRIDR.NUM == 0 or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN record is selected.
- ERXSTATUS is RAZ/WI.
- Direct reads and writes of ERXSTATUS are NOPs.
- Direct reads and writes of ERXSTATUS are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
        ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRSTATUS(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        R(t) = ERRSTATUS(UInt(ERRSELR().SEL))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        R(t) = ERRSTATUS(UInt(ERRSELR().SEL))[31:0];
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0100	0b010

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
         ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HCR_EL2().TERR
         ↪== '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR2().TERR ==
         ↪'1' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && PSTATE.M !=
         ↪M32_Monitor && SCR().TERR == '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRSTATUS(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA64EL3) &&
         ↪!ELUsingAArch32(EL3) && SCR_EL3().TWERR == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Undefined();
    elsif HaveEL(EL3) && EL3SDDUndefPriority() && IsFeatureImplemented(FEAT_AA32EL3) &&
         ↪ELUsingAArch32(EL3) && SCR().TERR == '1' then
        Undefined();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TERR ==
         ↪'1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && SCR_EL3().TWERR
         ↪== '1' then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch64_AArch32SystemAccessTrap(EL3, 0x03);
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && SCR().TERR == '1'
         ↪then
        if EL3SDDUndef() then
            Undefined();
        else
            AArch32_TakeMonitorTrapException();
        end;
    else
        ERRSTATUS(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR().TERR == '1' then
        AArch32_TakeMonitorTrapException();
    else
        ERRSTATUS(UInt(ERRSELR().SEL))[31:0] = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.19 VDFSR, Virtual SError Exception Syndrome Register

The VDFSR characteristics are:

Purpose

Provides the syndrome value reported to software on taking a virtual SError exception exception to EL1, or on executing an `ESB` instruction at EL1.

When the virtual SError exception injected using HCR.VA is taken to EL1 using AArch32, then the syndrome value is reported in DFSR.{AET, ExT} and the remainder of DFSR is set as defined by VMSAv8-32. For more information, see The AArch32 Virtual Memory System Architecture.

If the virtual SError exception injected using HCR.VA is deferred by an `ESB` instruction, then the syndrome value is written to VDISR.

Configuration

If EL2 is not implemented, then VDFSR is RES0 from Monitor mode when SCR.NS == 1.

AArch32 System register VDFSR bits [31:0] are architecturally mapped to AArch64 System register VSESR_EL2[31:0].

This register is present only when FEAT_RAS is implemented and FEAT_AA32EL1 is implemented. Otherwise, direct accesses to VDFSR are UNDEFINED.

Attributes

VDFSR is a 32-bit register.

Field descriptions

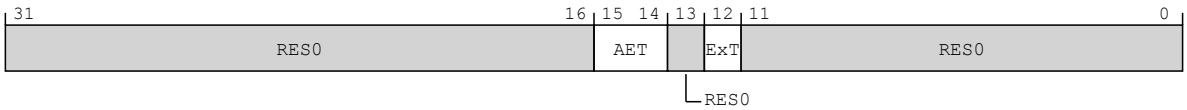

Bits [31:16]

Reserved, RES0.

AET, bits [15:14]

When a virtual SError exception is taken to EL1 using AArch32, DFSR[15:14] is set to VDFSR.AET.

When a virtual SError exception is deferred by an `ESB` instruction, VDISR[15:14] is set to VDFSR.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

When a virtual SError exception is taken to EL1 using AArch32, DFSR[12] is set to VDFSR.ExT.

When a virtual SError exception is deferred by an `ESB` instruction, VDISR[12] is set to VDFSR.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [11:0]

Reserved, RES0.

Accessing VDFSR

Direct reads and writes of VDFSR are UNDEFINED if EL3 is implemented and using AArch32 in all Secure privileged modes other than Monitor mode.

If EL2 is not implemented, then VDFSR is RES0 from Monitor mode when SCR.NS == 1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VDFSR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = VDFSR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b011

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T5 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T5 == '1'
         ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VDFSR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        VDFSR() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.6.20 VDISR, Virtual Deferred Interrupt Status Register

The VDISR characteristics are:

Purpose

Records that an SError exception has been consumed by an `ESB` instruction.

Configuration

If EL2 is not implemented, then VDISR is RES0 from Monitor mode when SCR.NS == 1.

AArch32 System register VDISR bits [31:0] are architecturally mapped to AArch64 System register VDISR_EL2[31:0].

This register is present only when FEAT_AA32EL1 is implemented and FEAT_RAS is implemented. Otherwise, direct accesses to VDISR are UNDEFINED.

Attributes

VDISR is a 32-bit register.

Field descriptions

When TTBCR.EAE == '0':

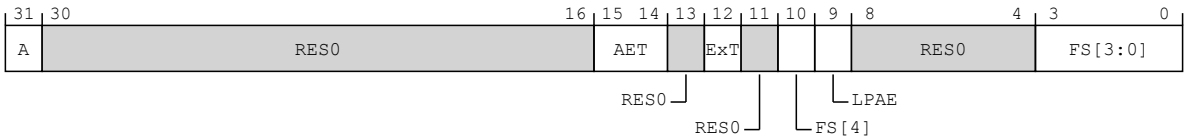

A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VDFSR.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VDFSR.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]

Fault status code. Set to `0b10110` when an `ESB` instruction defers a virtual SError exception.

FS	Meaning
`0b10110`	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an `ESB` instruction defers a virtual SError exception.

LPAE	Meaning
`0b0`	Using the Short-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

When TTBCR.EAE == '1':

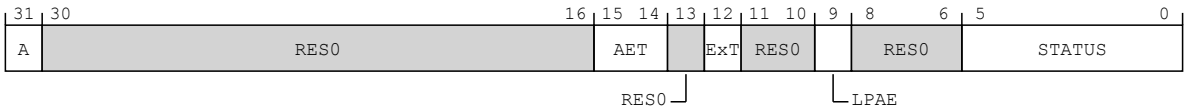

A, bit [31]

Set to 1 when an `ESB` instruction defers a virtual SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AET, bits [15:14]

The value copied from VDFSR.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VDFSR.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.

Set to TTBCR.EAE when an ESB instruction defers a virtual SError exception.

LPAE	Meaning
0b1	Using the Long-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status code. Set to 0b010001 when an ESB instruction defers a virtual SError exception.

STATUS	Meaning
0b010001	Asynchronous SError exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing VDISR

Direct reads and writes of VDFSR are UNDEFINED if EL3 is implemented and using AArch32 in all Secure privileged modes other than Monitor mode.

An indirect write to VDISR made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR occurring in program order after the ESB instruction.

If EL2 is not implemented, then VDISR is RES0 from Monitor mode when SCR.NS == 1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_RAS)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VDISR();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = VDISR();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_AA32EL1) && IsFeatureImplemented(FEAT_RAS)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VDISR() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        VDISR() = R(t);
    end;
end;
```

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && (HCR_EL2().AMO
        ↪== '1' || (IsFeatureImplemented(FEAT_DoubleFault2) && IsHCRXEL2Enabled() && HCRX_EL2().TMEA
        ↪== '1')) then
        R(t) = VDISR_EL2()[31:0];
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().AMO == '1'
        ↪then
        R(t) = VDISR();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        R(t) = Zeros{32};
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        R(t) = Zeros{32};
    else
        R(t) = DISR();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        R(t) = Zeros{32};
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        R(t) = Zeros{32};
    else
        R(t) = DISR();
    end;
elsif PSTATE.EL == EL3 then
    R(t) = DISR();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b001

```
if !(IsFeatureImplemented(FEAT_RAS) && IsFeatureImplemented(FEAT_AA32EL1)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && HSTR_EL2().T12 ==
        ↪'1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HSTR().T12 == '1'
        ↪then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && (HCR_EL2().AMO
        ↪== '1' || (IsFeatureImplemented(FEAT_DoubleFault2) && IsHCRXEL2Enabled() && HCRX_EL2().TMEA
        ↪== '1')) then
        VDISR_EL2()[31:0] = R(t);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && HCR().AMO == '1'
        ↪then
        VDISR() = R(t);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        return;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        return;
    else
        DISR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA64EL3) && !ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR_EL3().EA == '1' then
        return;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) && !Halted() &&
        ↪SCR().EA == '1' then
        return;
    else
        DISR() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    DISR() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7 Generic Timer registers

This section lists the Generic Timer registers in AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.1 CNTFRQ, Counter-timer Frequency register

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configuration

AArch32 System register CNTFRQ bits [31:0] are architecturally mapped to AArch64 System register CNTFRQ_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTFRQ are UNDEFINED.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions

31	0
Clock frequency	

ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().[EL0PCTEN,EL0VCTEN] == '00' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PCTEN == '0' &&
         ↪CNTKCTL().PL0VCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().[EL0PCTEN,EL0VCTEN] == '00' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        R(t) = CNTFRQ();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = CNTFRQ();
elsif PSTATE.EL == EL2 then
    R(t) = CNTFRQ();
elsif PSTATE.EL == EL3 then
    R(t) = CNTFRQ();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0000	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif IsHighestEL(PSTATE.EL) then
    CNTFRQ() = R(t);
else
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.2 CNTHCTL, Counter-timer Hyp Control register

The CNTHCTL characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 modes to the physical counter and the Non-secure EL1 physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register CNTHCTL bits [31:0] are architecturally mapped to AArch64 System register CNTHCTL_EL2[31:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to CNTHCTL are UNDEFINED.

Attributes

CNTHCTL is a 32-bit register.

Field descriptions

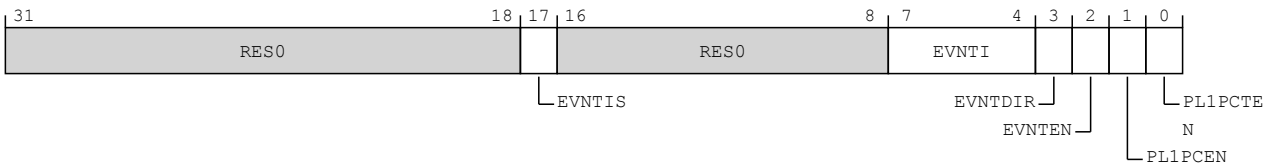

Bits [31:18]

Reserved, RES0.

EVNTIS, bit [17]

When FEAT_ECV is implemented:

Controls the scale of the generation of the event stream.

EVNTIS	Meaning
0b0	The CNTHCTL.EVNTI field applies to CNTPCT[15:0].
0b1	The CNTHCTL.EVNTI field applies to CNTPCT[23:8].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:8]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVNTI, bits [7:4]

Selects which bit of CNTPCT, as seen from EL2, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTHCTL.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTPCT.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTPCT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTPCT trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
`0b0`	A 0 to 1 transition of the trigger bit triggers an event.
`0b1`	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from CNTPCT as seen from EL2.

EVNTEN	Meaning
`0b0`	Disables the event stream.
`0b1`	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL1PCEN, bit [1]

Traps Non-secure EL0 and EL1 MRC or MCR accesses, reported using EC syndrome value `0x03`, and MRRC or MCRR accesses, reported using EC syndrome value `0x04`, to the physical timer registers to Hyp mode.

PL1PCEN	Meaning
`0b0`	Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to Hyp mode, unless trapped by CNTKCTL.PL0PTEN.
`0b1`	This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL1PCTEN, bit [0]

Traps Non-secure EL0 and EL1 MRRC or MCRR accesses, reported using EC syndrome value `0x04`, to the physical counter register to Hyp mode.

PL1PCTEN	Meaning
`0b0`	Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to Hyp mode, unless it is trapped by CNTKCTL.PL0PCTEN.
`0b1`	This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHCTL

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
`0b1111`	`0b100`	`0b1110`	`0b0001`	`0b000`

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t) = CNTHCTL();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = CNTHCTL();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHCTL() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        CNTHCTL() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.3 CNTHP_CTL, Counter-timer Hyp Physical Timer Control register

The CNTHP_CTL characteristics are:

Purpose

Control register for the Hyp mode physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register CNTHP_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHP_CTL_EL2[31:0].

This register is banked between CNTHP_CTL and CNTHP_CTL_S and CNTHP_CTL_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTHP_CTL are UNDEFINED.

Attributes

CNTHP_CTL is a 32-bit register.

This register has the following instances:

- CNTHP_CTL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTHP_CTL_S, when FEAT_AA32EL3 is implemented.
- CNTHP_CTL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

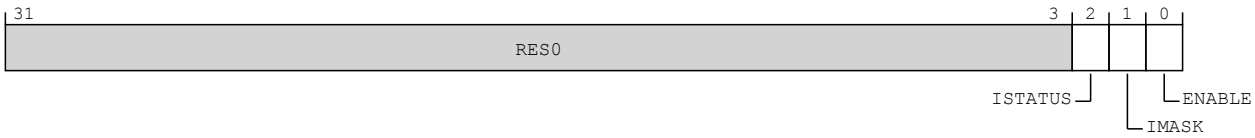

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL3 or the highest implemented Exception level is EL2, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CTL

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t) = CNTHP_CTL();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t) = CNTHP_CTL();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHP_CTL() = R(t);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        CNTHP_CTL() = R(t);
    end;
end;
```

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t) = CNTHPS_CTL_EL2()[31:0];
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t) = CNTHP_CTL_EL2()[31:0];
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CNTP_CTL_NS();
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CNTP_CTL_NS();
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = CNTP_CTL_S();
    else
        R(t) = CNTP_CTL_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CTL_EL2()[31:0] = R(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CTL_EL2()[31:0] = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS() = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS() = R(t);
    else
        CNTP_CTL() = R(t);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CTL_S() = R(t);
    else
        CNTP_CTL_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.4 CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register

The CNTHP_CVAL characteristics are:

Purpose

Holds the compare value for the Hyp mode physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register CNTHP_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHP_CVAL_EL2[63:0].

This register is banked between CNTHP_CVAL and CNTHP_CVAL_S and CNTHP_CVAL_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTHP_CVAL are UNDEFINED.

Attributes

CNTHP_CVAL is a 64-bit register.

This register has the following instances:

- CNTHP_CVAL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTHP_CVAL_S, when FEAT_AA32EL3 is implemented.
- CNTHP_CVAL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

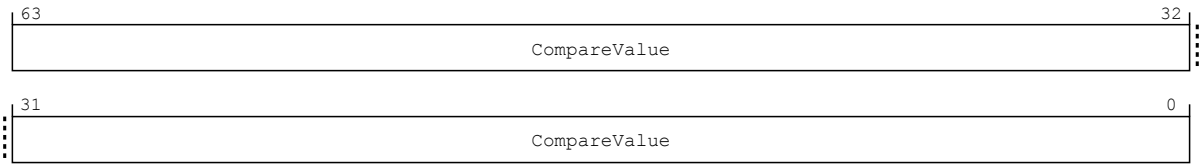

CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHP_CTL.ISTATUS is set to 1.
- If CNTHP_CTL.IMASK is 0, an interrupt is generated.

When CNTHP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CVAL

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0110

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTHP_CVAL();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t, t2) = CNTHP_CVAL();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0110

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL() = R(t, t2);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        CNTHP_CVAL() = R(t, t2);
    end;
end;
```

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0010

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
         ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t, t2) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t, t2) = CNTHP_CVAL_EL2();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = CNTP_CVAL_NS();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = CNTP_CVAL_NS();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t, t2) = CNTP_CVAL_S();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t, t2) = CNTP_CVAL_NS();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
         ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = R(t, t2);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CNTP_CVAL_NS() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CVAL_S() = R(t, t2);
    else
        CNTP_CVAL_NS() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.5 CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register

The CNTHP_TVAL characteristics are:

Purpose

Holds the timer value for the Hyp mode physical timer.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

AArch32 System register CNTHP_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHP_TVAL_EL2[31:0].

This register is banked between CNTHP_TVAL and CNTHP_TVAL_S and CNTHP_TVAL_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTHP_TVAL are UNDEFINED.

Attributes

CNTHP_TVAL is a 32-bit register.

This register has the following instances:

- CNTHP_TVAL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTHP_TVAL_S, when FEAT_AA32EL3 is implemented.
- CNTHP_TVAL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHP_CTL.ENABLE is 1, the value returned is (CNTHP_CVAL - CNTPCT).

On a write of this register, CNTHP_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTHP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHP_CTL.ISTATUS is set to 1.
- If CNTHP_CTL.IMASK is 0, an interrupt is generated.

When CNTHP_CTL.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_TVAL

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if CNTHP_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTHP_CVAL() - PhysicalCountInt())[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        if CNTHP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        CNTHP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) &&
        ↪!ELUsingAArch32(EL2)) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
        ↪&& !ELIsInHost(EL0) then
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0];
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if SCR().NS == '1' then
            if CNTP_CTL_NS().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                R(t) = ARBITRARY:bits(32);
            else
                R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
            end;
        else
            if CNTP_CTL_S().ENABLE == '0' then
                R(t) = ARBITRARY:bits(32);
            else
                R(t) = (CNTP_CVAL_S() - PhysicalCountInt())[31:0];
            end;
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
     ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
     ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
     ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
     ↪then
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0];
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        if CNTP_CTL_S().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_S() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) &&
        ↪!ELUsingAArch32(EL2)) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
        ↪&& !ELIsInHost(EL0) then
        CNTP_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTPOFF_EL2();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if SCR().NS == '1' then
            CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
        else
            CNTP_CVAL_S() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
         ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
         ↪then
        CNTP_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTPOFF_EL2();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CVAL_S() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.6 CNTHPS_CTL, Counter-timer Secure Physical Timer Control Register (EL2)

The CNTHPS_CTL characteristics are:

Purpose

Provides AArch32 access from EL0 to the Secure EL2 physical timer.

Configuration

AArch32 System register CNTHPS_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHPS_CTL_EL2[31:0].

This register is banked between CNTHPS_CTL and CNTHPS_CTL_S and CNTHPS_CTL_NS.

This register is present only when FEAT_AA32 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CTL are UNDEFINED.

Attributes

CNTHPS_CTL is a 32-bit register.

This register has the following instances:

- CNTHPS_CTL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTHPS_CTL_S, when FEAT_AA32EL3 is implemented.
- CNTHPS_CTL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

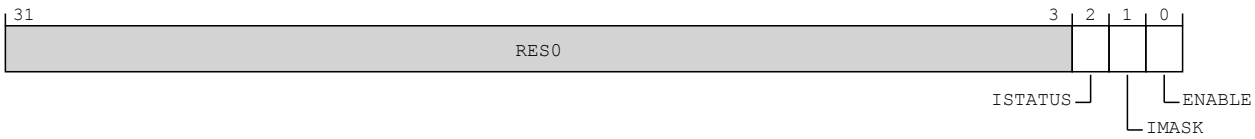

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the CNTHPS_CTL.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHPS_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_CTL

This register is accessed using the encoding for CNTP_CTL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
         ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t) = CNTHPS_CTL_EL2()[31:0];
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t) = CNTHP_CTL_EL2()[31:0];
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CNTP_CTL_NS();
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CNTP_CTL_NS();
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = CNTP_CTL_S();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t) = CNTP_CTL_NS();
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CTL_EL2()[31:0] = R(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CTL_EL2()[31:0] = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CNTP_CTL_NS() = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS() = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CTL_S() = R(t);
    else
        CNTP_CTL_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.7 CNTHPS_CVAL, Counter-timer Secure Physical Timer CompareValue Register (EL2)

The CNTHPS_CVAL characteristics are:

Purpose

Provides AArch32 access from EL0 to the compare value for the Secure EL2 physical timer.

Configuration

AArch32 System register CNTHPS_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHPS_CVAL_EL2[63:0].

This register is banked between CNTHPS_CVAL and CNTHPS_CVAL_S and CNTHPS_CVAL_NS.

This register is present only when FEAT_AA32 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CVAL are UNDEFINED.

Attributes

CNTHPS_CVAL is a 64-bit register.

This register has the following instances:

- CNTHPS_CVAL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTHPS_CVAL_S, when FEAT_AA32EL3 is implemented.
- CNTHPS_CVAL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

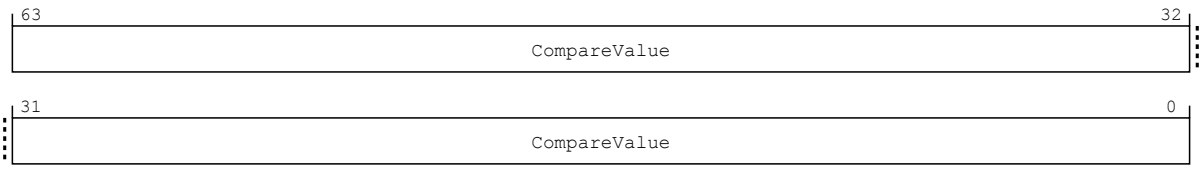

CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHPS_CTL.ISTATUS is set to 1.
- If CNTHPS_CTL.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_CVAL

This register is accessed using the encoding for CNTP_CVAL.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t, t2) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t, t2) = CNTHP_CVAL_EL2();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = CNTP_CVAL_NS();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = CNTP_CVAL_NS();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t, t2) = CNTP_CVAL_S();
    else
        R(t, t2) = CNTP_CVAL_NS();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = R(t, t2);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CVAL_S() = R(t, t2);
    else
        CNTP_CVAL_NS() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.8 CNTHPS_TVAL, Counter-timer Secure Physical Timer TimerValue Register (EL2)

The CNTHPS_TVAL characteristics are:

Purpose

Provides AArch32 access from EL0 to the timer value for the Secure EL2 physical timer.

Configuration

AArch32 System register CNTHPS_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHPS_TVAL_EL2[31:0].

This register is banked between CNTHPS_TVAL and CNTHPS_TVAL_S and CNTHPS_TVAL_NS.

This register is present only when FEAT_AA32 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_TVAL are UNDEFINED.

Attributes

CNTHPS_TVAL is a 32-bit register.

This register has the following instances:

- CNTHPS_TVAL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTHPS_TVAL_S, when FEAT_AA32EL3 is implemented.
- CNTHPS_TVAL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL.ENABLE is 1, the value returned is (CNTHPS_CVAL - CNTPCT).

On a write of this register, CNTHPS_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTHPS_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHPS_CTL.ISTATUS is set to 1.
- If CNTHPS_CTL.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_TVAL

This register is accessed using the encoding for CNTP_TVAL.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) &&
        ↪!ELUsingAArch32(EL2)) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
        ↪&& !ELIsInHost(EL0) then
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0];
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if SCR().NS == '1' then
            if CNTP_CTL_NS().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                R(t) = ARBITRARY:bits(32);
            else
                R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
            end;
        else
            if CNTP_CTL_S().ENABLE == '0' then
                R(t) = ARBITRARY:bits(32);
            else
                R(t) = (CNTP_CVAL_S() - PhysicalCountInt())[31:0];
            end;
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
     ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
     ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
     ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
     ↪then
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0];
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        if CNTP_CTL_S().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_S() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) &&
        ↪!ELUsingAArch32(EL2)) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
        ↪&& !ELIsInHost(EL0) then
        CNTP_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTPOFF_EL2();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if SCR().NS == '1' then
            CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
        else
            CNTP_CVAL_S() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
         ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
         ↪then
        CNTP_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTPOFF_EL2();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CVAL_S() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.9 CNTHV_CTL, Counter-timer Virtual Timer Control register (EL2)

The CNTHV_CTL characteristics are:

Purpose

Provides AArch32 access to the control register for the EL2 virtual timer.

Configuration

AArch32 System register CNTHV_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHV_CTL_EL2[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHV_CTL are UNDEFINED.

Attributes

CNTHV_CTL is a 32-bit register.

Field descriptions

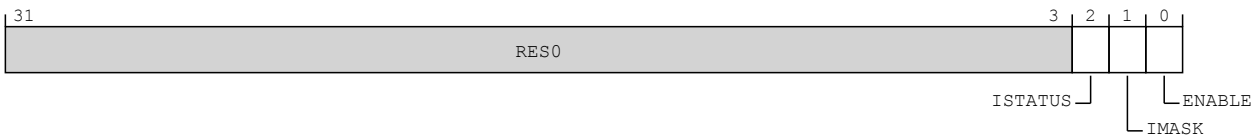

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHV_CTL

This register is accessed using the encoding for CNTV_CTL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪CNTHCTL_EL2().EL0VTEN == '0' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            R(t) = CNTHVS_CTL_EL2()[31:0];
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            R(t) = CNTHV_CTL_EL2()[31:0];
        else
            R(t) = CNTV_CTL();
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
            ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            R(t) = CNTV_CTL();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = CNTV_CTL();
    elsif PSTATE.EL == EL3 then
        R(t) = CNTV_CTL();
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = R(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2()[31:0] = R(t);
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CTL() = R(t);
elsif PSTATE.EL == EL3 then
    CNTV_CTL() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.10 CNTHV_CVAL, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL characteristics are:

Purpose

Provides AArch32 access to the compare value for the EL2 virtual timer.

Configuration

AArch32 System register CNTHV_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHV_CVAL_EL2[63:0].

This register is present only when FEAT_AA32 is implemented and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHV_CVAL are UNDEFINED.

Attributes

CNTHV_CVAL is a 64-bit register.

Field descriptions

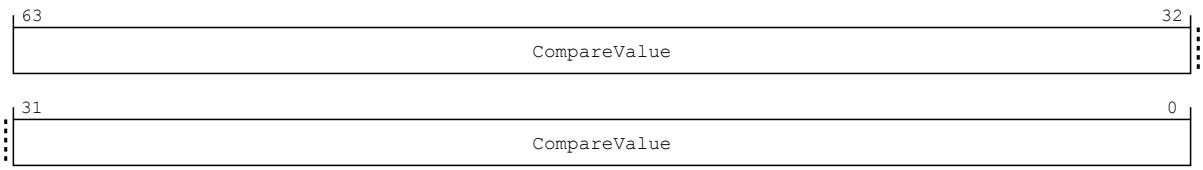

CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHV_CTL.ISTATUS is set to 1.
- If CNTHV_CTL.IMASK is 0, an interrupt is generated.

When CNTHV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

Accessing CNTHV_CVAL

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t, t2) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t, t2) = CNTHV_CVAL_EL2();
    else
        R(t, t2) = CNTV_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        R(t, t2) = CNTV_CVAL();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTV_CVAL();
elsif PSTATE.EL == EL3 then
    R(t, t2) = CNTV_CVAL();
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = R(t, t2);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = R(t, t2);
    else
        CNTV_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CVAL() = R(t, t2);
elsif PSTATE.EL == EL3 then
    CNTV_CVAL() = R(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.11 CNTHV_TVAL, Counter-timer Virtual Timer TimerValue register (EL2)

The CNTHV_TVAL characteristics are:

Purpose

Provides AArch32 access to the timer value for the EL2 virtual timer.

Configuration

AArch32 System register CNTHV_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHV_TVAL_EL2[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHV_TVAL are UNDEFINED.

Attributes

CNTHV_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHV_CTL.ENABLE is 1, the value returned is (CNTHV_CVAL - CNTVCT).

On a write of this register, CNTHV_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTHV_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHV_CTL.ISTATUS is set to 1.
- If CNTHV_CTL.IMASK is 0, an interrupt is generated.

When CNTHV_CTL.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

Accessing CNTHV_TVAL

This register is accessed using the encoding for CNTV_TVAL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHV_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0];
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
        else
            R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0];
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
        else
            R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    elsif HaveEL(EL2) then
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    else
        R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
        else
            CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
        else
            CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) then
        CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
    else
        CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.12 CNTHVS_CTL, Counter-timer Secure Virtual Timer Control Register (EL2)

The CNTHVS_CTL characteristics are:

Purpose

Provides AArch32 access from EL0 to the Secure EL2 virtual timer.

Configuration

AArch32 System register CNTHVS_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHVS_CTL_EL2[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHVS_CTL are UNDEFINED.

Attributes

CNTHVS_CTL is a 32-bit register.

Field descriptions

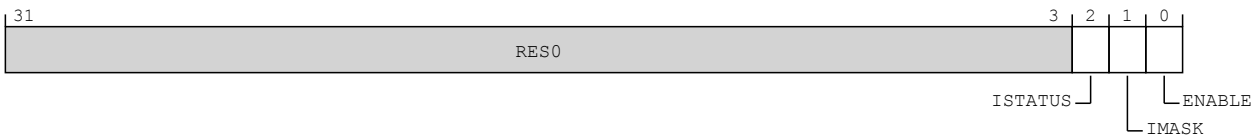

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHVS_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHVS_CTL

This register is accessed using the encoding for CNTV_CTL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
                ↪== '1' then
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪CNTHCTL_EL2().EL0VTEN == '0' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
            ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            R(t) = CNTHVS_CTL_EL2()[31:0];
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            R(t) = CNTHV_CTL_EL2()[31:0];
        else
            R(t) = CNTV_CTL();
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
            ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            R(t) = CNTV_CTL();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = CNTV_CTL();
    elsif PSTATE.EL == EL3 then
        R(t) = CNTV_CTL();
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = R(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2()[31:0] = R(t);
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CTL() = R(t);
elsif PSTATE.EL == EL3 then
    CNTV_CTL() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.13 CNTHVS_CVAL, Counter-timer Secure Virtual Timer CompareValue Register (EL2)

The CNTHVS_CVAL characteristics are:

Purpose

Provides AArch32 access to the compare value for the Secure EL2 virtual timer.

Configuration

AArch32 System register CNTHVS_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHVS_CVAL_EL2[63:0].

This register is present only when FEAT_AA32 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHVS_CVAL are UNDEFINED.

Attributes

CNTHVS_CVAL is a 64-bit register.

Field descriptions

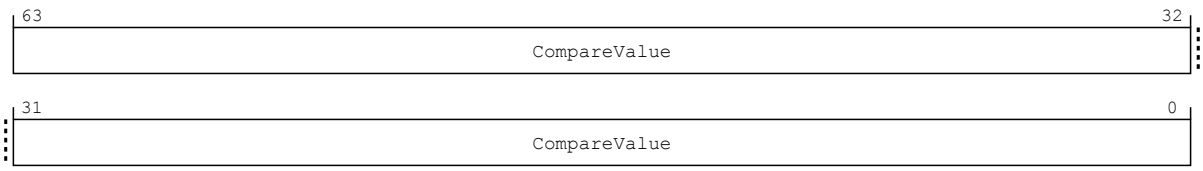

CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHVS_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHVS_CTL.ISTATUS is set to 1.
- If CNTHVS_CTL.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

Accessing CNTHVS_CVAL

This register is accessed using the encoding for CNTV_CVAL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t, t2) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t, t2) = CNTHV_CVAL_EL2();
    else
        R(t, t2) = CNTV_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        R(t, t2) = CNTV_CVAL();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTV_CVAL();
elsif PSTATE.EL == EL3 then
    R(t, t2) = CNTV_CVAL();
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = R(t, t2);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = R(t, t2);
    else
        CNTV_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CVAL() = R(t, t2);
elsif PSTATE.EL == EL3 then
    CNTV_CVAL() = R(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.14 CNTHVS_TVAL, Counter-timer Secure Virtual Timer TimerValue Register (EL2)

The CNTHVS_TVAL characteristics are:

Purpose

Provides AArch32 access to the timer value for the Secure EL2 virtual timer.

Configuration

AArch32 System register CNTHVS_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHVS_TVAL_EL2[31:0].

This register is present only when FEAT_AA32 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHVS_TVAL are UNDEFINED.

Attributes

CNTHVS_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHVS_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHVS_CTL.ENABLE is 1, the value returned is (CNTHVS_CVAL - CNTVCT).

On a write of this register, CNTHVS_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHVS_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTHVS_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHVS_CTL.ISTATUS is set to 1.
- If CNTHVS_CTL.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

Accessing CNTHVS_TVAL

This register is accessed using the encoding for CNTV_TVAL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHV_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0];
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
        else
            R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0];
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
        else
            R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    elsif HaveEL(EL2) then
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    else
        R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
        else
            CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
        else
            CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) then
        CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
    else
        CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.15 CNTKCTL, Counter-timer Kernel Control register

The CNTKCTL characteristics are:

Purpose

Controls the generation of an event stream from the virtual counter, and access from EL0 modes to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

Configuration

AArch32 System register CNTKCTL bits [31:0] are architecturally mapped to AArch64 System register CNTKCTL_EL1[31:0].

This register is present only when FEAT_AA32EL1 is implemented. Otherwise, direct accesses to CNTKCTL are UNDEFINED.

Attributes

CNTKCTL is a 32-bit register.

Field descriptions

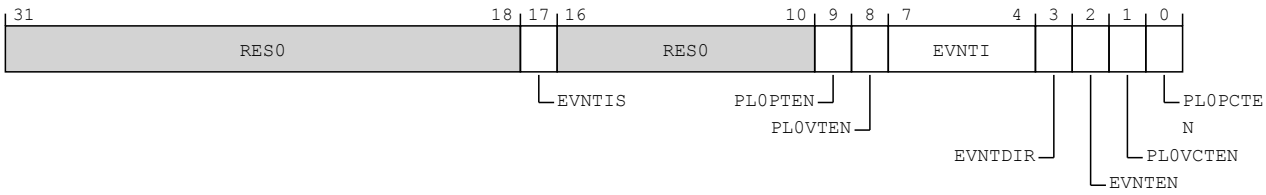

Bits [31:18]

Reserved, RES0.

EVNTIS, bit [17]

When FEAT_ECV is implemented:

Controls the scale of the generation of the event stream.

EVNTIS	Meaning
0b0	The CNTKCTL.EVNTI field applies to CNTVCT[15:0].
0b1	The CNTKCTL.EVNTI field applies to CNTVCT[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:10]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PL0PTEN, bit [9]

Traps PL0 accesses to the physical timer registers to Undefined mode.

PL0PTEN	Meaning
0b0	PL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are trapped to Undefined mode.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL0VTEN, bit [8]

Traps PL0 accesses to the virtual timer registers to Undefined mode.

PL0VTEN	Meaning
0b0	PL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to Undefined mode.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit of CNTVCT, as seen from EL1, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTKCTL.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTVCT.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTVCT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTVCT trigger bit, as seen from EL1 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from CNTVCT as seen from EL1.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL0VCTEN, bit [1]

Traps PL0 accesses to the frequency register and virtual counter register to Undefined mode.

PL0VCTEN	Meaning
0b0	PL0 accesses to the CNTVCT are trapped to Undefined mode. PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0PCTEN is also 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL0PCTEN, bit [0]

Traps PL0 accesses to the frequency register and physical counter register to Undefined mode.

PL0PCTEN	Meaning
0b0	PL0 accesses to the CNTPCT are trapped to Undefined mode. PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0VCTEN is also 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTKCTL

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    R(t) = CNTKCTL();
elsif PSTATE.EL == EL2 then
    R(t) = CNTKCTL();
elsif PSTATE.EL == EL3 then
    R(t) = CNTKCTL();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_AA32EL1) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    CNTKCTL() = R(t);
elsif PSTATE.EL == EL2 then
    CNTKCTL() = R(t);
elsif PSTATE.EL == EL3 then
    CNTKCTL() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.16 CNTP_CTL, Counter-timer Physical Timer Control register

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

AArch32 System register CNTP_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTP_CTL_EL0[31:0].

This register is banked between CNTP_CTL and CNTP_CTL_S and CNTP_CTL_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTP_CTL are UNDEFINED.

Attributes

CNTP_CTL is a 32-bit register.

This register has the following instances:

- CNTP_CTL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTP_CTL_S, when FEAT_AA32EL3 is implemented.
- CNTP_CTL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

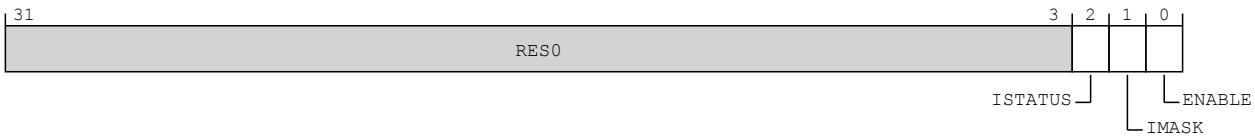

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing CNTP_CTL

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t) = CNTHPS_CTL_EL2()[31:0];
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t) = CNTHP_CTL_EL2()[31:0];
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CNTP_CTL_NS();
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t) = CNTP_CTL_NS();
    else
        R(t) = CNTP_CTL();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t) = CNTP_CTL_S();
    else
        R(t) = CNTP_CTL_NS();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CTL_EL2()[31:0] = R(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CTL_EL2()[31:0] = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS() = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS() = R(t);
    else
        CNTP_CTL() = R(t);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CTL_S() = R(t);
    else
        CNTP_CTL_NS() = R(t);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.17 CNTP_CVAL, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configuration

AArch32 System register CNTP_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTP_CVAL_EL0[63:0].

This register is banked between CNTP_CVAL and CNTP_CVAL_S and CNTP_CVAL_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTP_CVAL are UNDEFINED.

Attributes

CNTP_CVAL is a 64-bit register.

This register has the following instances:

- CNTP_CVAL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTP_CVAL_S, when FEAT_AA32EL3 is implemented.
- CNTP_CVAL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

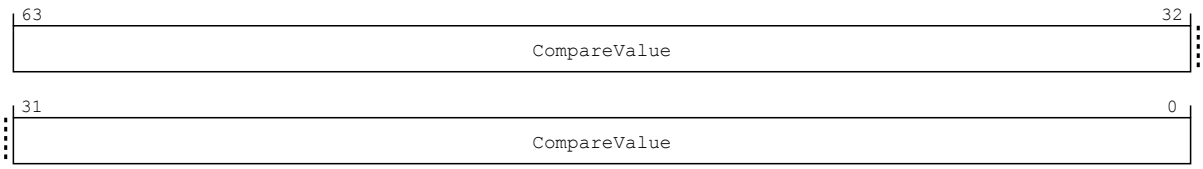

CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t, t2) = CNTHPS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t, t2) = CNTHP_CVAL_EL2();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
        ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
        ↪== '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = CNTP_CVAL_NS();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        R(t, t2) = CNTP_CVAL_NS();
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        R(t, t2) = CNTP_CVAL_S();
    else
        R(t, t2) = CNTP_CVAL_NS();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = R(t, t2);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL1 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = R(t, t2);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CVAL_S() = R(t, t2);
    else
        CNTP_CVAL_NS() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.18 CNTP_TVAL, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

AArch32 System register CNTP_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTP_TVAL_EL0[31:0].

This register is banked between CNTP_TVAL and CNTP_TVAL_S and CNTP_TVAL_NS.

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTP_TVAL are UNDEFINED.

Attributes

CNTP_TVAL is a 32-bit register.

This register has the following instances:

- CNTP_TVAL, when EL3 is not implemented or FEAT_AA64 is implemented.
- CNTP_TVAL_S, when FEAT_AA32EL3 is implemented.
- CNTP_TVAL_NS, when FEAT_AA32EL3 is implemented.

Field descriptions

31 … 0
TimerValue

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CNTP_CVAL - CNTPCT).

On a write of this register, CNTP_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHPS_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHPS_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHP_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHP_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) &&
        ↪!ELUsingAArch32(EL2)) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
        ↪&& !ELIsInHost(EL0) then
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0];
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if SCR().NS == '1' then
            if CNTP_CTL_NS().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                R(t) = ARBITRARY:bits(32);
            else
                R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
            end;
        else
            if CNTP_CTL_S().ENABLE == '0' then
                R(t) = ARBITRARY:bits(32);
            else
                R(t) = (CNTP_CVAL_S() - PhysicalCountInt())[31:0];
            end;
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
     ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
     ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
     ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
     ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
     ↪then
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - (PhysicalCountInt() - CNTPOFF_EL2()))[31:0];
        end;
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        if CNTP_CTL_S().ENABLE == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_S() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTP_CTL_NS().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL_NS() - PhysicalCountInt())[31:0];
        end;
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHP_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) &&
        ↪!ELUsingAArch32(EL2)) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
        ↪&& !ELIsInHost(EL0) then
        CNTP_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTPOFF_EL2();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if SCR().NS == '1' then
            CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
        else
            CNTP_CVAL_S() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && !ELIsInHost(EL2) &&
         ↪CNTHCTL_EL2().EL1PCEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1PTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) && CNTHCTL().PL1PCEN
         ↪== '0' then
        AArch32_TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
         ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV == '1'
         ↪then
        CNTP_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTPOFF_EL2();
    elsif HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && IsFeatureImplemented(FEAT_AA32EL3) && ELUsingAArch32(EL3) then
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        CNTP_CVAL_S() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        CNTP_CVAL_NS() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.19 CNTPCT, Counter-timer Physical Count register

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configuration

All reads to the CNTPCT occur in program order relative to reads to CNTPCTSS or CNTPCT.

AArch32 System register CNTPCT bits [63:0] are architecturally mapped to AArch64 System register CNTPCT_EL0[63:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTPCT are UNDEFINED.

Attributes

CNTPCT is a 64-bit register.

Field descriptions

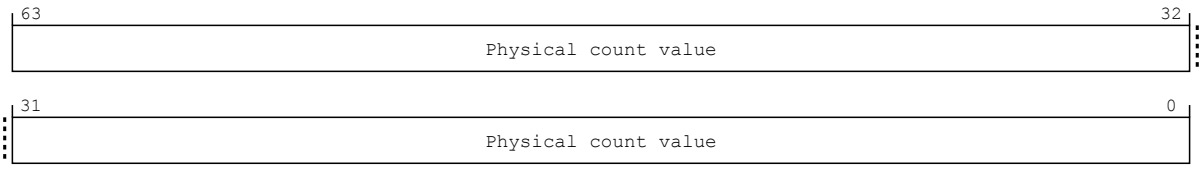

PhysicalCount, bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCTEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
            ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV ==
            ↪'1' && !ELIsInHost(EL0) then
            R(t, t2) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪CNTHCTL().PL1PCTEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
            ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV ==
            ↪'1' then
            R(t, t2) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    R(t, t2) = PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.20 CNTPCTSS, Counter-timer Self-Synchronized Physical Count register

The CNTPCTSS characteristics are:

Purpose

Holds the 64-bit physical count value.

Configuration

All reads to the CNTPCTSS occur in program order relative to reads to CNTPCT or CNTPCTSS.

This register is a view of the CNTPCT register for which reads appear to occur in program order relative to other instructions, without the need for any explicit synchronization. Reads of this register return a value consistent with the counter not being read until the read instruction is known to be non-speculative.

AArch32 System register CNTPCTSS bits [63:0] are architecturally mapped to AArch64 System register CNTPCTSS_EL0[63:0].

This register is present only when FEAT_AA32 is implemented and FEAT_ECV is implemented. Otherwise, direct accesses to CNTPCTSS are UNDEFINED.

Attributes

CNTPCTSS is a 64-bit register.

Field descriptions

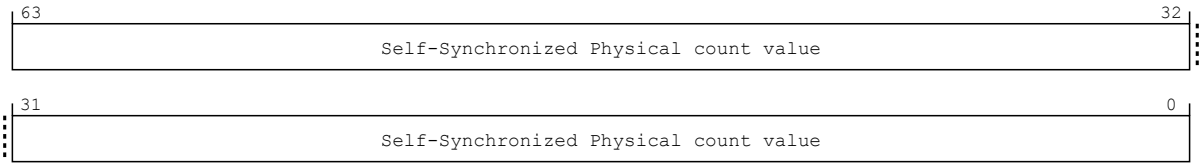

SSPhysicalCount, bits [63:0]

Self-Synchronized Physical count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCTSS

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b1000

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_ECV)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0PCTEN == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0PCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL2) && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL2) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪HCR_EL2().TGE == '0' && CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) &&
        ↪CNTHCTL().PL1PCTEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
            ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV ==
            ↪'1' && !ELIsInHost(EL0) then
            R(t, t2) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
        ↪CNTHCTL_EL2().EL1PCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) &&
        ↪CNTHCTL().PL1PCTEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        if IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) &&
            ↪!ELUsingAArch32(EL2) && (!HaveEL(EL3) || SCR_EL3().ECVEn == '1') && CNTHCTL_EL2().ECV ==
            ↪'1' then
            R(t, t2) = PhysicalCountInt() - CNTPOFF_EL2();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    R(t, t2) = PhysicalCountInt();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.21 CNTV_CTL, Counter-timer Virtual Timer Control register

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configuration

AArch32 System register CNTV_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTV_CTL_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTV_CTL are UNDEFINED.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions

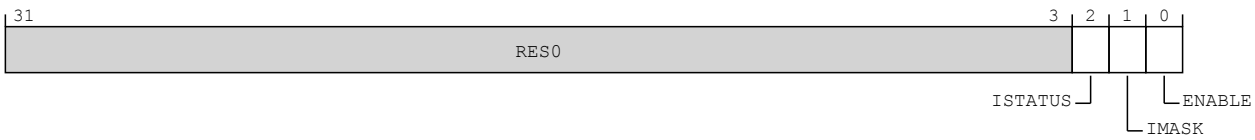

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing CNTV_CTL

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
                 ↪== '1' then
                AArch32_TakeHypTrapException(0x00);
            else
                Undefined();
            end;
        elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪CNTHCTL_EL2().EL0VTEN == '0' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
            R(t) = CNTHVS_CTL_EL2()[31:0];
        elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
            R(t) = CNTHV_CTL_EL2()[31:0];
        else
            R(t) = CNTV_CTL();
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            R(t) = CNTV_CTL();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = CNTV_CTL();
    elsif PSTATE.EL == EL3 then
        R(t) = CNTV_CTL();
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2() = R(t);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CTL_EL2()[31:0] = R(t);
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CTL() = R(t);
elsif PSTATE.EL == EL3 then
    CNTV_CTL() = R(t);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.22 CNTV_CVAL, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL characteristics are:

Purpose

Holds the compare value for the virtual timer.

Configuration

AArch32 System register CNTV_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTV_CVAL_EL0[63:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTV_CVAL are UNDEFINED.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions

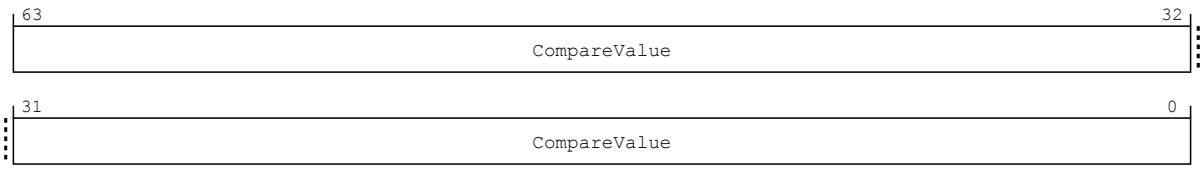

CompareValue, bits [63:0]

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0011

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        R(t, t2) = CNTHVS_CVAL_EL2();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        R(t, t2) = CNTHV_CVAL_EL2();
    else
        R(t, t2) = CNTV_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        R(t, t2) = CNTV_CVAL();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTV_CVAL();
elsif PSTATE.EL == EL3 then
    R(t, t2) = CNTV_CVAL();
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
             ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = R(t, t2);
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = R(t, t2);
    else
        CNTV_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
             ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL() = R(t, t2);
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CVAL() = R(t, t2);
elsif PSTATE.EL == EL3 then
    CNTV_CVAL() = R(t, t2);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.23 CNTV_TVAL, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configuration

AArch32 System register CNTV_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTV_TVAL_EL0[31:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTV_TVAL are UNDEFINED.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CNTV_CVAL - CNTVCT).

On a write of this register, CNTV_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHVS_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        if CNTHV_CTL_EL2().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTHV_CVAL_EL2() - PhysicalCountInt())[31:0];
        end;
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0];
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
        else
            R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF_EL2()))[31:0];
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL2 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    end;
elsif PSTATE.EL == EL3 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    elsif HaveEL(EL2) then
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    else
        R(t) = (CNTV_CVAL() - PhysicalCountInt())[31:0];
    end;
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
        ↪CNTKCTL_EL1().EL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x03);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
            ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
            ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪CNTHCTL_EL2().EL0VTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
        ↪!ELIsInHost(EL0) && IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    elsif ELIsInHost(EL0) && IsCurrentSecurityState(SS_Secure) && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    elsif ELIsInHost(EL0) && !IsCurrentSecurityState(SS_Secure) then
        CNTHV_CVAL_EL2() = SignExtend{64}(R(t)) + PhysicalCountInt();
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
        else
            CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EL1TVT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x03);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
            CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
        else
            CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) then
        CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
    else
        CNTV_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.24 CNTVCT, Counter-timer Virtual Count register

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF.

Configuration

The value of this register is the same as the value of CNTPCT in the following conditions:

- When EL2 is not implemented.
- When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, and this register is read from Non-secure EL0.

All reads to the CNTVCT occur in program order relative to reads to CNTVCTSS or CNTVCT.

AArch32 System register CNTVCT bits [63:0] are architecturally mapped to AArch64 System register CNTVCT_EL0[63:0].

This register is present only when FEAT_AA32 is implemented. Otherwise, direct accesses to CNTVCT are UNDEFINED.

Attributes

CNTVCT is a 64-bit register.

Field descriptions

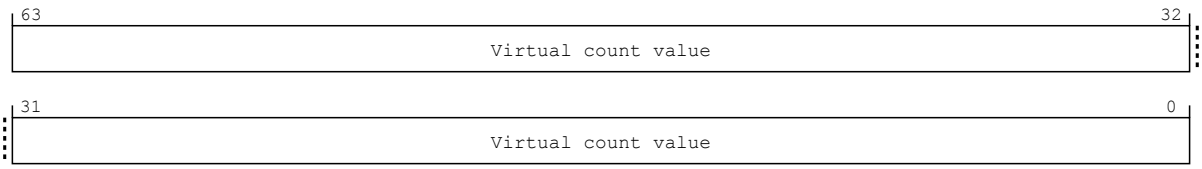

VirtualCount, bits [63:0]

Virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && (!EL2Enabled()
             ↪|| !ELIsInHost(EL0)) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && !ELUsingAArch32(EL2) &&
             ↪(!EL2Enabled() || !ELIsInHost(EL0)) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL2) then
        R(t, t2) = PhysicalCountInt() - CNTVOFF();
    else
        R(t, t2) = PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        R(t, t2) = PhysicalCountInt() - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        R(t, t2) = PhysicalCountInt() - CNTVOFF();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t, t2) = PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.25 CNTVCTSS, Counter-timer Self-Synchronized Virtual Count register

The CNTVCTSS characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT minus the virtual offset visible in CNTVOFF.

Configuration

All reads to the CNTVCTSS occur in program order relative to reads to CNTVCT or CNTVCTSS.

This register is a view of the CNTVCT register for which reads appear to occur in program order relative to other instructions, without the need for any explicit synchronization. Reads of this register return a value consistent with the counter not being read until the read instruction is known to be non-speculative.

AArch32 System register CNTVCTSS bits [63:0] are architecturally mapped to AArch64 System register CNTVCTSS_EL0[63:0].

This register is present only when FEAT_AA32 is implemented and FEAT_ECV is implemented. Otherwise, direct accesses to CNTVCTSS are UNDEFINED.

Attributes

CNTVCTSS is a 64-bit register.

Field descriptions

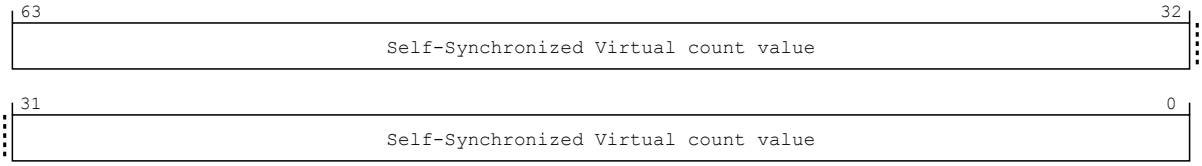

SSVirtualCount, bits [63:0]

Self-Synchronized Virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCTSS

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b1001

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_AA32) && IsFeatureImplemented(FEAT_ECV)) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_AA64EL1) && !ELUsingAArch32(EL1) && !ELIsInHost(EL0) &&
         ↪CNTKCTL_EL1().EL0VCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x04);
        end;
    elsif IsFeatureImplemented(FEAT_AA32EL1) && ELUsingAArch32(EL1) && CNTKCTL().PL0VCTEN == '0' then
        if EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) && HCR_EL2().TGE
             ↪== '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2)) && HCR().TGE
             ↪== '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif ELIsInHost(EL0) && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪CNTHCTL_EL2().EL0VCTEN == '0' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && (IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2)) &&
         ↪!ELIsInHost(EL0) && CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) && (!EL2Enabled()
             ↪|| !ELIsInHost(EL0)) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && !ELUsingAArch32(EL2) &&
             ↪(!EL2Enabled() || !ELIsInHost(EL0)) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) &&
         ↪CNTHCTL_EL2().EL1TVCT == '1' then
        AArch64_AArch32SystemAccessTrap(EL2, 0x04);
    else
        if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) && ELUsingAArch32(EL2) then
            R(t, t2) = PhysicalCountInt() - CNTVOFF();
        else
            R(t, t2) = PhysicalCountInt();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL2) then
        R(t, t2) = PhysicalCountInt() - CNTVOFF();
    else
        R(t, t2) = PhysicalCountInt();
    end;
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_AA64EL2) && !ELUsingAArch32(EL2) then
        R(t, t2) = PhysicalCountInt() - CNTVOFF_EL2();
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_AA32EL2) then
        R(t, t2) = PhysicalCountInt() - CNTVOFF();
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        R(t, t2) = PhysicalCountInt();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G8.7.26 CNTVOFF, Counter-timer Virtual Offset register

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT and the virtual count value visible in CNTVCT.

Configuration

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

Note

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the virtual counter uses a fixed virtual offset of zero when CNTVCT is read from Non-secure EL0.

AArch32 System register CNTVOFF bits [63:0] are architecturally mapped to AArch64 System register CNTVOFF_EL2[63:0].

This register is present only when FEAT_AA32EL2 is implemented. Otherwise, direct accesses to CNTVOFF are UNDEFINED.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

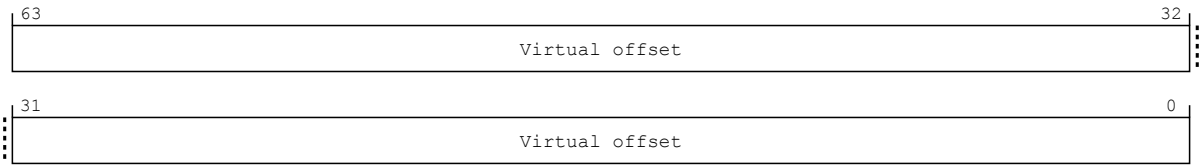

VOffset, bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0100

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTVOFF();
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        R(t, t2) = CNTVOFF();
    end;
end;
```

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

coproc	CRm	opc1
0b1111	0b1110	0b0100

```
if !IsFeatureImplemented(FEAT_AA32EL2) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTVOFF() = R(t, t2);
elsif PSTATE.EL == EL3 then
    if SCR().NS == '0' then
        Undefined();
    else
        CNTVOFF() = R(t, t2);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part H
External Debug

Chapter H1
About External Debug

This chapter gives an overview of A-profile external debug and specifies the required debug authentication. It contains the following sections:

- Introduction to external debug.
- External debug.
- Required debug authentication.

Note

For information about self-hosted debug, see AArch64 Self-hosted Debug and AArch32 Self-hosted Debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1 Introduction to external debug

The A-profile supports both:

Self-hosted debug

The PE itself hosts a debugger. That is, developers developing software to run on the PE use debugger software running on the same PE.

External debug

The debugger is external to the PE. The debugging might be either on-chip, for example in a second PE, or off-chip, for example a JTAG debugger that accesses the chip through a Debug Access Port.

External debug is particularly useful for:

- Hardware bring-up. That is, debugging during development when a system is first powered up and not all of the software functionality is available.
- PEs that are deeply embedded inside systems.

To support external debug, the Arm architecture defines required features that are called *external debug features*.

Note

An external debugger has a potentially high level of control over and visibility into the PE. The system sets this level using debug authentication. See Required debug authentication.
If the debug authentication level is set too low, agents may be able to bypass elements of the security and privilege models. This includes both off-chip agents and on-chip agents such as unprivileged or Non-secure software.

H1.1.1 Definition and constraints of a debugger in the context of external debug

When the description of external debug in this Part of the manual describes a *debugger* as controlling external debug this debugger might be a second on-chip PE or an off-chip device such as a JTAG debugger using a Debug Access Port (DAP).

If a Debug Access Port is implemented:

- When debug is prohibited at the Debug Access Port, the port must not generate accesses to the external debug interface of the PE.
- When debug is prohibited for a Security state at the Debug Access Port, it must not be possible to generate accesses to PA spaces that are not accessible for that Security state.
- When accesses for a Security state are allowed at the Debug Access Port, the port must be able to generate accesses to PA spaces that are accessible for that Security state.

If FEAT_Debugv8p4 is not implemented, accesses to the PE are controlled by the external authentication interface functions, `ExternalInvasiveDebugEnabled()`, `ExternalNoninvasiveDebugEnabled()`, `ExternalSecureNoninvasiveDebugEnabled()`, and `ExternalSecureInvasiveDebugEnabled()`.

If FEAT_TRBE and FEAT_RME are implemented, the external authentication interface functions `ExternalRealmNoninvasiveDebugEnabled()`, `ExternalRealmInvasiveDebugEnabled()`, `ExternalRootNoninvasiveDebugEnabled()`, and `ExternalRootInvasiveDebugEnabled()` are added.

If FEAT_Debugv8p4 is implemented, the bus Requester, which may be the Debug Access Port, controls the accesses it makes to the PE and MDCR_EL3.{EPMAD, EDAD} and, if FEAT_TRBE is implemented, MDCR_EL3.ETAD, control access to registers by an external debugger.

If FEAT_TRBE and FEAT_RME are implemented, MDCR_EL3.{ETAD, EPMAD, EDAD, EDADE, ETADE, EPMADE} control the Security state of access to registers by an external debugger.

If FEAT_Debugv8p4 is not implemented, the external authentication interface functions override MDCR_EL3.{EPMAD, EDAD}.

The Debug Access Port is not required to use the same authentication interface as the PE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm recommends the following authentication interface:

- When `ExternalSecureInvasiveDebugEnabled()` == FALSE at the PE, Secure debug is disabled at the DAP.
- When `ExternalRealmInvasiveDebugEnabled()` == FALSE at the PE, Realm debug is disabled at the DAP.
- When `ExternalRootInvasiveDebugEnabled()` == FALSE at the PE, Root debug is disabled at the DAP.
- When `ExternalInvasiveDebugEnabled()` == FALSE at the PE, all debug is prohibited at the DAP.

See also Required debug authentication.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.2 External debug

Debug events allow an external debugger to halt the PE. The A-profile provides the following debug events:

- Halting Step debug events:
  - The debugger can use this resource to make the PE step through code one line at a time.
- Halt Instruction debug event:
  - This might occur when software executes the Halting breakpoint instruction, `HLT`.
- Exception Catch debug event:
  - This can be programmed to occur on all entries to a given Exception level.
- External Debug Request debug event:
  - An embedded cross-trigger can signal this debug event.
- OS Unlock Catch debug event:
  - This might occur when the state of the OS Lock changes from locked to unlocked.
- Reset Catch debug events:
  - This might occur when the PE exits reset state.
- Software Access debug event:
  - This can be programmed to occur when software tries to access the Breakpoint Value registers, the Breakpoint Control registers, the Watchpoint value registers, or the Watchpoint Control registers. It caused a trap to Debug state.

Breakpoints and watchpoints can also halt the PE.

When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state:
  - The ITR contains a single register, EDITR, and associated flow-control flags.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger:
  - The DCC includes the data transfer registers, DTRRX and DTRTX, and associated flow-control flags.
  - Although the DCC is an essential part of Debug state operation, it can also be used in Non-debug state.
- The PE cannot service any interrupts in Debug state.

Debug State describes Debug state in more detail.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.3 Required debug authentication

A PE implements the debug authentication interface defined in this section. The debug authentication interface controls:

- Whether halting the PE and entering Debug state is allowed or prohibited.
- Whether some aspects of non-invasive debug are allowed or prohibited.
- Some legacy aspects of the AArch32 self-hosted debug model.

The pseudocode functions shown in Table H1-1, and the conditions that follow that table, define the architectural requirements for debug authentication.

Table H1-1 Debug authentication functions

Pseudocode function	Description
`ExternalNoninvasiveDebugEnabled()`	Returns TRUE if Non-secure non-invasive debug is allowed.
`ExternalInvasiveDebugEnabled()`	Returns TRUE if Non-secure invasive debug is allowed.
`ExternalSecureNoninvasiveDebugEnabled()`	Returns TRUE if Secure non-invasive debug is allowed.
`AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled()`	Returns TRUE if Secure invasive self-hosted debug is allowed in AArch32.
`ExternalSecureInvasiveDebugEnabled()`	Returns TRUE if Secure invasive debug is allowed.
`ExternalRealmNoninvasiveDebugEnabled()`	Returns TRUE if Realm non-invasive debug is allowed.
`ExternalRealmInvasiveDebugEnabled()`	Returns TRUE if Realm invasive debug is allowed.
`ExternalRootNoninvasiveDebugEnabled()`	Returns TRUE if Root non-invasive debug is allowed.
`ExternalRootInvasiveDebugEnabled()`	Returns TRUE if Root invasive debug is allowed.

The following conditions always apply:

- If `ExternalInvasiveDebugEnabled()` is FALSE, then `ExternalSecureInvasiveDebugEnabled()` is FALSE.
- If `ExternalNoninvasiveDebugEnabled()` is FALSE, then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.
- If `ExternalInvasiveDebugEnabled()` is TRUE, then `ExternalNoninvasiveDebugEnabled()` is TRUE.
- If `ExternalSecureInvasiveDebugEnabled()` is TRUE, then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

If FEAT_Debugv8p4 is implemented, then all of the following apply:

- `ExternalNoninvasiveDebugEnabled()` always returns TRUE.
- `ExternalSecureNoninvasiveDebugEnabled()` returns the same value as `ExternalSecureInvasiveDebugEnabled()`.

If FEAT_RME is implemented, then all of the following apply:

- If `ExternalInvasiveDebugEnabled()` is FALSE, then `ExternalRealmInvasiveDebugEnabled()` is FALSE.
- If any of `ExternalInvasiveDebugEnabled()`, `ExternalSecureInvasiveDebugEnabled()`, or `ExternalRealmInvasiveDebugEnabled()` are FALSE, then `ExternalRootInvasiveDebugEnabled()` is FALSE.
- `ExternalRealmNoninvasiveDebugEnabled()` returns the same value as `ExternalRealmInvasiveDebugEnabled()`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `ExternalRootNoninvasiveDebugEnabled()` returns the same value as `ExternalRootInvasiveDebugEnabled()`.

If EL3 is implemented, then all of the following apply:

- If FEAT_RME is not implemented, then `ExternalSecureInvasiveDebugEnabled()` determines whether halting is allowed or prohibited at EL3.
- If FEAT_RME is implemented, then `ExternalRootInvasiveDebugEnabled()` determines whether halting is allowed or prohibited at EL3.

Arm recommends the use of the interface described in Recommended authentication interface to provide this debug authentication. The pseudocode functions in A-profile Architecture Pseudocode, which are linked to by the entries in the Pseudocode function column of Table H1-1, assume that this interface is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H2
Debug State

This chapter describes Debug state. It contains the following sections:

- About Debug state.
- Halting the PE on debug events.
- Entering Debug state.
- Behavior in Debug state.
- Exiting Debug state.

Note

Table K14-1 disambiguates the general register references used in this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.1 About Debug state

In external debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the Program Counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger.

The PE cannot service any interrupts in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.2 Halting the PE on debug events

For details of debug events, see Introduction to Halting debug events and Breakpoint and Watchpoint debug events.

On a debug event, the PE must do one of the following:

- Enter Debug state.
- Pend the debug event.
- Generate a debug exception.
- Ignore the debug event.

This behavior depends on both:

- Whether halting is allowed by the current state of the debug authentication interface. See Halting allowed and halting prohibited.
- The type of debug event and the programming of the debug control registers.
  - See Halting debug events for all Halting debug events.
  - See Breakpoint and Watchpoint debug events for Breakpoint and Watchpoint debug events.

See also Other debug exceptions.

This means that behavior can be CONSTRAINED UNPREDICTABLE if the conditions change. See Synchronization and Halting debug events.

Table H3-1 summarizes the possible outcomes of each type of debug event.

H2.2.1 Halting allowed and halting prohibited

Halting can be either allowed or prohibited:

- Halting is always prohibited in Debug state.
- Halting is always prohibited when `DoubleLockStatus()` == TRUE.
  - This means that FEAT_DoubleLock is implemented and OS Double lock is locked.
- Halting is also controlled by the IMPLEMENTATION DEFINED authentication interface, and is prohibited when all of the following apply:
  - The PE is in Non-secure state and `ExternalInvasiveDebugEnabled()` == FALSE.
  - The PE is in Secure state and `ExternalSecureInvasiveDebugEnabled()` == FALSE.
  - The PE is in Realm state and `ExternalRealmInvasiveDebugEnabled()` == FALSE.
  - The PE is in Root state and `ExternalRootInvasiveDebugEnabled()` == FALSE.
- Otherwise, halting is allowed.

For more information, see:

- Pseudocode description of Halting on debug events.
- Required debug authentication.
- Recommended External Debug Interface.

H2.2.2 Halting debug events

The Halting debug events are described in Halting Debug Events.

When a Halting debug event is generated, it causes entry to Debug state if all of the following are true:

- Halting is allowed. See Halting allowed and halting prohibited.
- The Halting debug event is one of:
  - A Halt Instruction debug event and EDSCR.HDE == 1.
  - A Software Access debug event and OSLSR_EL1.OSLK == 0, meaning that the OS Lock is unlocked.
  - Neither a Halt Instruction debug event nor a Software Access debug event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

— A Halt Instruction debug event is the only Halting debug event that relies on EDSCR.HDE == 1.

— Halting on Breakpoint and Watchpoint debug events is also controlled by EDSCR.HDE. See Breakpoint and Watchpoint debug events.

— EDSCR.HDE can be written by software when the OS Lock is locked. Privileged code can use MDCR_EL3.TDOSA and HDCR.TDOSA to trap writes to OS Lock registers.

If a Halting debug event does not generate entry to Debug state because the conditions listed in this section do not hold, then:

- If the Halting debug event is a Halt Instruction debug event, the instruction that generated the Halting debug event is treated as UNDEFINED.
- If the Halting debug event is a Software Access debug event, it is ignored.
- If the Halting debug event is an Exception Catch debug event, it is ignored except when FEAT_Debugv8p8 is implemented, in which case the event might be pended.

In all other cases the Halting debug event is pended, see Pending Halting debug events.

Summary of actions from debug events summarizes the possible outcome for each type of Debug event.

Note

Halting debug events never generate debug exceptions.

H2.2.3 Breakpoint and Watchpoint debug events

A breakpoint or watchpoint generates an entry to Debug state if all of the following conditions hold:

- Halting debug is enabled, that is EDSCR.HDE == 1.
- Halting is allowed. See Halting allowed and halting prohibited.
- The OS Lock is unlocked, that is OSLSR.OSLK == 0.

The Address Mismatch breakpoint type is reserved when all of these conditions are met.

MDSCR_EL1.MDE or DBGDSCRext.MDBGen is ignored when determining whether to enter Debug state. A breakpoint or watchpoint that generates entry to Debug state is a Breakpoint or Watchpoint debug event and does not generate a debug exception.

A breakpoint or watchpoint that does not generate an entry to Debug state either:

- Generates a Breakpoint or Watchpoint exception.
- Is ignored.

Note

EDSCR.HDE is ignored when determining whether to generate a debug exception. The debug exception is suppressed only if the PE enters Debug state. This means that the use of Halting debug mode in one Security state does not affect the Exception model in another Security state.

See AArch64 Self-hosted Debug, AArch32 Self-hosted Debug, and the Note in Other debug exceptions.

H2.2.4 Other debug exceptions

The following events never generate entry to Debug state:

- Breakpoint Instruction exceptions.
- Software Step exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Vector Catch exceptions.

The behavior of these events is unchanged when Halting debug mode is enabled, that is when EDSCR.HDE == 1. This means that these events can do one of the following:

- They can generate a debug exception.
- They can be ignored.

For additional information, see AArch64 Self-hosted Debug and AArch32 Self-hosted Debug.

H2.2.5 Debug state entry and debug event prioritization

The following are synchronous debug events:

- Breakpoint debug event.
- Watchpoint debug event.
- Halting Step debug event.
- Halt Instruction debug event.
- Exception Catch debug event.
- Software Access debug event.
- Reset Catch debug event.

Each of these synchronous debug events are treated as a synchronous exception generated by an instruction, or by the taking of an exception or reset. That is, if halting is allowed, the synchronous debug event must be taken before any subsequent instructions are executed. Reset Catch debug events must be taken before the PE executes the instruction at the reset vector.

Note

- Reset Catch and Exception Catch debug events might be generated asynchronously, because they can result from an asynchronous exception. However, if halting is allowed after the reset or asynchronous exception has been processed, the Reset Catch or Exception Catch debug event is taken synchronously.
- The Halting Step debug event is generated by the instruction after the stepped instruction. Therefore, if the stepped instruction generates any other synchronous exceptions or debug events these are taken first.

If halting is prohibited then Halting Step debug events and Reset Catch debug events might be pended and taken asynchronously. OS Unlock Catch debug events are always pended and taken asynchronously. See Pending Halting debug events.

The architecture does not define when asynchronous debug events are taken, and therefore the prioritization of asynchronous debug events is IMPLEMENTATION DEFINED. See Synchronization and Halting debug events.

The following list shows how the synchronous debug events are prioritized, with 1 being the highest priority.

Note

The priority numbering is the same as the numbering for AArch64 synchronous exception priorities listed in Synchronous exception types, and in particular Prioritization of asynchronous exceptions. This numbering correlates with the equivalent AArch32 list in Exception prioritization for exceptions taken to AArch32 state.

The priority for synchronous debug events is as follows:

1 Reset Catch debug event. See Reset Catch debug events.

This debug event has a higher priority than the synchronous exceptions listed in Synchronous exception types.

2 Exception Catch debug event if it has a priority of 2. See Exception Catch debug event.

This debug event can be assigned one of two priorities, 2 or 5. When it has a priority of 2, it has a higher priority than the synchronous exceptions listed in the Exception model.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3	Halting Step debug event. See Halting Step debug events. This debug event has a higher priority than the synchronous exceptions listed in the Exception model.
4	This event is not a debug event.
5	Exception Catch debug event if it has a priority of 5. See Exception Catch debug event. This debug event can be assigned one of two priorities, 2 or 5.
6-8	These events are not debug events.
9	Breakpoint debug events. See Breakpoint and Watchpoint debug events.
10-11	These events are not debug events.
12	Halt Instruction debug event. See Halt Instruction debug event.
13-38	These events are not debug events.
39	Software Access debug event. See Software Access debug event.
40-44	These events are not debug events.
45	Watchpoint debug events. See Breakpoint and Watchpoint debug events.
46-47	These events are not debug events.

For Reset Catch debug events and Halting Step debug events, the priorities listed in this section apply only when halting is allowed at the time the event is generated. This means that the event is taken synchronously and not pended.

For more information on the prioritization of exceptions, see:

- Synchronous exception types.
- Prioritization of asynchronous exceptions.
- Exception prioritization for exceptions taken to AArch32 state. This section covers synchronous and asynchronous exceptions.

H2.2.5.1 Breakpoint debug events and Vector Catch exception

An Address Matching Vector Catch exception has the same priority as a Breakpoint debug event. See Prioritization of Synchronous exceptions taken to AArch64 state.

The prioritization of these events is unchanged even if the breakpoint generates entry to Debug state instead of a Breakpoint exception. This means that if a single instruction generates both an Address Matching Vector Catch exception and a Breakpoint debug event, there is a CONSTRAINED UNPREDICTABLE choice of:

- The PE entering Debug state due to the Breakpoint debug event.
- A Vector Catch exception.

This applies only if all of the following are true:

- Halting debug is enabled.
- Halting is allowed.
- The OS Lock is unlocked.

An Exception Trapping Vector Catch exception must be generated immediately following the exception that generated it. This means that it does not appear in the priority table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.2.6 Imprecise entry to Debug state

Debug state entry is normally *precise*. This means that the PE cannot enter Debug state if it can neither complete nor abandon all currently executing instructions and leave the PE in a precise state. See Definition of a precise exception and imprecise exception.

A debugger can write a value of 1 to EDRCR.CBRRQ to allow imprecise entry to Debug state. An External Debug Request debug event must be pending before writing 1 to this bit. If 1 is written when an External Debug Request debug event is not pending, and the PE does not ignore the write, then the resulting PE behavior is UNPREDICTABLE. Support for allowing imprecise entry to Debug state using EDRCR.CBRRQ is OPTIONAL, and it is IMPLEMENTATION DEFINED when this mechanism is effective at forcing entry to Debug state.

The PE ignores writes to EDRCR.CBRRQ if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- FEAT_RME is not implemented and `ExternalSecureInvasiveDebugEnabled()` == FALSE, and either:
  - EL3 is not implemented and the implemented Security state is Secure state.
  - EL3 is implemented.
- FEAT_RME is implemented and `ExternalRootInvasiveDebugEnabled()` == FALSE.
- Any IMPLEMENTATION DEFINED reason.

Arm recommends that the PE ignores writes to EDRCR.CBRRQ when an External Debug Request debug event is not pending.

If forcing this PE to enter Debug state imprecisely could have an adverse effect on the wider system, then Arm recommends that a write to EDRCR.CBRRQ is ignored if the highest level of debug authentication is not enabled at the system level.

Example H2-1 shows how entry to Debug state can be forced.

Example H2-1 Forcing entry to Debug state

The debugger pends an External Debug Request debug event through the CTI to halt a program that has stopped responding. However, the memory system is not responding and a memory access instruction cannot complete. This means that Debug state cannot be entered precisely. The debugger writes a value of 1 to EDRCR.CBRRQ. The PE cancels all outstanding memory accesses and enters Debug state. As some instructions might not have completed correctly, entry to Debug state is imprecise.

When Debug state is entered imprecisely, all memory access instructions executed through the ITR have CONSTRAINED UNPREDICTABLE behavior. The value of all registers is UNKNOWN, but might be useful for diagnostic purposes.

H2.2.7 Summary of actions from debug events

Table H2-1 shows the Software and Halting debug events. In Table H2-1, the columns have the following meaning:

Debug event type

This means the type of debug event where:

Other software

Means one of:

- Software Step exceptions.
- Breakpoint Instruction exceptions.
- Vector Catch exceptions for AArch64 state or Vector Catch exceptions for AArch32 state.

Other Halting

Means one of the following:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Halting Step debug events.
- External Debug Request debug event.
- Reset Catch debug events.
- OS Unlock Catch debug event.

Other debug events are referred to explicitly.

Authentication

This means halting is allowed by the IMPLEMENTATION DEFINED external authentication interface. It is the result of one of the following pseudocode functions:

In Non-secure state `ExternalInvasiveDebugEnabled()`.

In Secure state `ExternalSecureInvasiveDebugEnabled()`.

In Realm state `ExternalRealmInvasiveDebugEnabled()`.

In Root state `ExternalRootInvasiveDebugEnabled()`.

DLK This indicates whether FEAT_DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE.

OSLK This is the value of OSLSR.OSLK. It indicates whether the OS Lock is locked.

HDE This is the value of EDSCR.HDE. It indicates whether Halting debug is enabled.

The letter X in Table H2-1 indicates that the value can be either 0 or 1.

Table H2-1 Debug authentication for external debug

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
Other software	X	X	X	X	Handled by the Exception model
Breakpoint or Watchpoint debug event	X	TRUE	X	X	Handled by the Exception model (ignored)
	X	FALSE	1	X	Handled by the Exception model (ignored)
	FALSE	FALSE	0	X	Handled by the Exception model
	TRUE	FALSE	0	0	Handled by the Exception model
	TRUE	FALSE	0	1	Entry to Debug state
Halt Instruction debug event	FALSE	X	X	X	UNDEFINED
	TRUE	TRUE	X	X	UNDEFINED
	TRUE	FALSE	X	0	UNDEFINED
	TRUE	FALSE	X	1	Entry to Debug state
Exception Catch debug event	FALSE	X	X	X	Might be pended[a]
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	X	X	Entry to Debug state
Software Access debug event	FALSE	X	X	X	Ignored
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	1	X	Ignored

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
	TRUE	FALSE	0	X	Entry to Debug state
Other Halting	FALSE	X	X	X	Debug event is pended
	TRUE	TRUE	X	X	Debug event is pended
	TRUE	FALSE	X	X	Entry to Debug state

a. See Exception Catch debug event.

H2.2.8 Pseudocode description of Halting on debug events

The `Halted()`, `Restarting()`, `HaltingAllowed()`, and `HaltOnBreakpointOrWatchpoint()` functions are described in the pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.3 Entering Debug state

On entry to Debug state, the preferred restart address and PSTATE are saved in DLR and DSPSR. The PE remains in the mode and Security state from which it entered Debug state.

If EDRCR.CBRRQ has a value of 0, entry to Debug state is precise. If EDRCR.CBRRQ has a value of 1, then imprecise entry to Debug state is permitted.

If a Watchpoint debug event causes an entry to Debug state, the PE records all of the following:

- When FEAT_EDHSR is not implemented or the PE sets EDHSR.FnV to 0, the address of the access that generated the Watchpoint debug event is recorded in EDWAR.
- When FEAT_EDHSR is implemented, information about the access that generated the Watchpoint debug event is recorded in EDHSR.

For more information, see:

- Exception syndrome information, fault address information, and preferred return address for a debug event taken from AArch64 state.
- Exception syndrome information, fault address information, and preferred return address for a debug event taken from AArch32 state.

Other than the effect on PSTATE and EDSCR, entry to Debug state is not a Context Synchronization event. The effects of entry to Debug state on PSTATE and EDSCR are synchronized.

H2.3.1 Entering Debug state from AArch32 state

When entering Debug state from AArch32 state, the PE remains in AArch32 state. In AArch32 Debug state the PE executes T32 instructions, regardless of the value of PSTATE.T before entering Debug state.

To allow the debugger to determine the state of the PE, the current Execution state for all four Exception levels can be read from EDSCR.RW, and the current Exception level can be read from EDSCR.EL.

The current endianness state, PSTATE.E, is unchanged on entry to Debug state.

Note

- If EL1 is using AArch32 state, the current endianness state can differ from that indicated by SCTLR.EE.
- If EL2 is using AArch32 state, the current endianness state can differ from that indicated by HSCTLR.EE.
- On entry to Debug state from AArch32 state, PSTATE.SS is copied to DSPSR.SS, even though the PE remains in AArch32 state.

See also Effect of entering Debug state on PSTATE.

H2.3.2 Effect of Debug state entry on DLR and DSPSR

DLR is set to the preferred restart address for the debug event, which depends on the event type. The value of PSTATE is saved in DSPSR.

For entry to Debug state from AArch32 state, the values saved in DSPSR.IT are always correct for the preferred restart address.

For synchronous Halting debug events, the preferred restart address is the address of the instruction that generated the debug event.

It is CONSTRAINED UNPREDICTABLE whether DSPSR_EL0.BTYPE is set to the value of PSTATE.BTYPE or 0 for synchronous debug events other than the following debug events:

- A Halting Step debug event.
- A Breakpoint debug event.
- A Halt Instruction debug event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For asynchronous Halting debug events, including pending Halting debug events taken asynchronously, the preferred restart address is the address of the first instruction that must be executed on exit from Debug state.

This means that:

- For Breakpoint and Watchpoint debug events, the preferred restart address is the same as the preferred return address for a debug exception, as described in AArch64 Self-hosted Debug and AArch32 Self-hosted Debug.
- For Halt Instruction debug events, DLR is set to the address of the `HLT` instruction and DSPSR.IT is correct for the `HLT` instruction.
- For Software Access debug events, DLR is set to the address of the accessing instruction and DSPSR.IT is correct for this instruction.
- For Halting Step debug events taken synchronously, DLR and DSPSR are set as the ELR and SPSR would be set for a Software Step exception. This is usually the address of, and PSTATE for, the instruction after the one that was stepped.
- For Exception Catch debug events, DLR is set to the address of the first instruction that must be executed on exit from Debug state, and DSPSR is correct for this instruction. This is the exception vector or reset vector address.
  - If the debug event is generated on taking an exception to a trapped Exception level, the DLR is set to the address of the exception vector the PE would have started fetching from. This is UNKNOWN if the VBAR for the Exception level has never been initialized. The DSPSR records the value of PSTATE after taking the exception. The Exception Catch occurs after the SPSR and the Link register are set, and the debugger can use these registers to determine where in the application program the exception occurred.

    Note

    Depending on the target Exception level and Execution state for the exception, the Link register is one of ELR_EL1, ELR_EL2, ELR_EL3, ELR_hyp, or LR (R14).

  - If the debug event is generated on an exception return to a trapped Exception level, the DLR is set to the target address of the exception return and the DSPSR records the value of PSTATE after the exception return.
- Reset Catch debug events taken synchronously behave like Exception Catch debug events.
- For Reset Catch debug events and Exception Catch debug events generated on reset to a trapped Exception level, the DLR is set to is set to the reset address and the DSPSR records the reset value of PSTATE.
- For pending Halting debug events and External Debug Request debug events, DLR is set to the address of the first instruction that must be executed on exit from Debug state and DSPSR.IT is correct for this instruction. See Pending Halting debug events.

Normally DLR is aligned according to the instruction set state indicated in DSPSR. However, a debug event might be taken at a point where the PC is not aligned.

H2.3.3 Effect of Debug state entry on System registers, the Event register, and Exclusives monitors

Entering Debug state has no effect on System registers other than DLR and DSPSR. In particular, ESRs, FARs, and FSRs are not updated on entering Debug state. SCR is unchanged, even when entering Debug state from EL3 when EL3 is using AArch32.

Entering Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors.

Note

Entry to Debug state might set the Event Register or clear the Exclusives monitors, or both. However, this is not a requirement, and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

Unless otherwise described in this reference manual, instructions executed in Debug state have their architecturally-defined effects on the System registers, the Event register, and Exclusives monitors.

H2.3.4 Effect of entering Debug state on PSTATE

The effect of an entry to Debug state on PSTATE is described in Entering Debug state and Entering Debug state from AArch32 state.

On entry to Debug state after PSTATE is saved in DSPSR:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PSTATE.{IL, PACM} is cleared to 0.
- PSTATE.UINJ is cleared to 0.
- PSTATE.TCO is set to 1.
- PSTATE.BTYPE is set to `0b00`.
- PSTATE.{IT, T, SS, D, A, I, F, SSBS, ALLINT, PM} are set to UNKNOWN values

PSTATE.{N, Z, C, V, nRW, EL, SP, PAN, UAO, DIT, EXLOCK, DIT} are unchanged.

If FEAT_AA32EL0 is implemented, then PSTATE.{Q, GE, E, M} are unchanged.

For more information, see PSTATE in Debug state.

H2.3.5 Entering Debug state during loads and stores

The PE can enter Debug state during instructions that perform a sequence of memory accesses, as opposed to a single single-copy atomic access, because of a Watchpoint debug event. The effect of entering Debug state on such an instruction is the same as taking a Data Abort exception during such an instruction.

In addition, when executing in AArch64 state, the PE can enter Debug state because of an External Debug Request debug event during instructions that perform a sequence of memory accesses. The effect of entering Debug state on such an instruction is the same as taking an interrupt exception during such an instruction.

This applies to all memory types.

H2.3.6 Entering Debug state and Software Step

When Software Step is active, a debug event that causes entry to Debug state behaves like an exception taken to an Exception level above the debug target Exception level. That is:

- If the instruction that is stepped generates a synchronous debug event that causes entry to Debug state, or an asynchronous debug event is taken before the step completes, the PE enters Debug state with DSPSR.SS set to 1.
- A pending Halting debug event or an asynchronous debug event can be taken after the step has completed. In this case the PE enters Debug state with DSPSR.SS set to 0.

In addition:

- If the instruction that is stepped generates an exception trapped by an Exception Catch debug event, the PE enters Debug state at the exception vector with DSPSR.SS set to 0. This is because PSTATE.SS is set to 0 by taking the exception.
- If the PE is reset, PSTATE.SS is reset to 0. If the following debug events are enabled, and halting is allowed, the PE enters Debug state with DSPSR.SS set to 0:
  - Reset Catch debug event at the reset Exception level.
  - Exception Catch debug event at the reset Exception level.
  - Halting Step debug event.
- If Halting Step is also active, then Halting Step and Software Step operate in parallel and can both become active-pending. In this case Halting step has a higher priority than Software step. This means that the PE enters Debug state and DSPSR.SS is set to 0.

H2.3.7 Pseudocode description of entering Debug state

The `DebugHalt` constants are described in shared/debug/halting/DebugHalt in the pseudocode. The `UpdateEDSCRFields()` and `Halt()` functions are described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4 Behavior in Debug state

Instructions are executed in Debug state from the Instruction Transfer Register, ITR. The debugger controls which instructions are executed in Debug state by writing the instructions to the External Debug Instruction Transfer register, EDITR.

The Execution state of the PE determines which instruction set is executed:

- If the PE is in AArch64 state it executes A64 instructions.
- If the PE is in AArch32 state it executes T32 instructions:
  - For a 32-bit T32 instruction, EDITR[15:0] specifies the first halfword and EDITR[31:16] specifies the second halfword.
  - For a 16-bit T32 instruction, EDITR[15:0] contains the instruction and EDITR[31:16] is ignored. All 16-bit T32 instructions are UNPREDICTABLE in Debug state.

The PE does not execute A32 instructions in Debug state.

Some instructions are available only in Debug state. See Debug state operations, DCPS, DRPS, MRS, MSR. In Non-debug state these instructions are UNDEFINED.

The following sections describe behavior in Debug state:

- PSTATE in Debug state.
- Executing instructions in Debug state.
- Decode tables.
- Security in Debug state.
- Privilege in Debug state.
- Debug state operations, DCPS, DRPS, MRS, MSR.
- Exceptions in Debug state.
- Accessing registers in Debug state.
- Accessing memory in Debug state.

This section specifies the CONSTRAINED UNPREDICTABLE behaviors that apply in Debug state, but see Changing the value of EDECR.SS when not in Debug state for a change in Non-debug state that causes CONSTRAINED UNPREDICTABLE behavior.

H2.4.1 PSTATE in Debug state

PSTATE.{N, Z, C, V, Q, IT, GE, SS, ALLINT, SSBS, BTYPE, D, A, I, F, T, PM} are all ignored in Debug state:

- There are no conditional instructions in Debug state.
- In AArch32 state, the PE executes only T32 instructions and PSTATE.IT is ignored.
- Software step is inactive.
- Asynchronous exceptions and debug events are ignored.
- If FEAT_SSBS is implemented, then hardware is permitted to load or store speculatively, regardless of the value of PSTATE.SSBS.
- In AArch64 state, PSTATE.BTYPE is treated as `0b00`.

Instructions executed in Debug state indirectly read PSTATE.{TCO, UAO, PAN, IL, E, M, nRW, EL, SP} as they would in Non-debug state.

Note

PSTATE.DIT is not guaranteed to have any effect in Debug state.

In Debug state:

- PSTATE.PAN is set to 1 by:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - A `DCPS<n>` instruction to EL1 using AArch64 if SCTLR_EL1.SPAN == 0.
  - A `DCPS<n>` instruction to EL2 using AArch64 if SCTLR_EL2.SPAN == 0.
- PSTATE.UINJ is set to 0 by a `DCPS<n>` instruction to AArch64 state.
- PSTATE.UAO is set to 0 by a `DCPS<n>` instruction to AArch64 state.
- PSTATE.TCO is set to 1 by a `DCPS<n>` instruction to AArch64 state.
- PSTATE.EXLOCK is set to 0 by a `DCPS<n>` instruction to AArch64 state.
- If FEAT_SSBS is implemented, the `DCPS<n>` instruction leaves the PSTATE.SSBS bit UNKNOWN.
- If FEAT_PAuth_LR is implemented, the `DCPS<n>` instruction sets the PSTATE.PACM bit to 0.

PSTATE can also be changed by taking exceptions in Debug state, and by the execution of `DCPS<n>` and `DRPS` instructions.

When FEAT_MTE is implemented, if Memory access mode is enabled and PSTATE.TCO is 0, reads and writes to the external debug interface DTR registers are CONSTRAINED UNPREDICTABLE, with the following permitted behaviors:

- The PE behaves as if PSTATE.TCO is 0. That is, the load or store operation performs the tag check if required.
- The PE behaves as if PSTATE.TCO is 1. That is, the load or store operation does not perform the tag check.

For more information, see The Memory Tagging Extension.

H2.4.2 Executing instructions in Debug state

The instructions executed in Debug state must be either A64 instructions or T32 instructions, depending on the current Execution state.

Each instruction falls into one of the following groups:

- Debug state instructions. These are instructions that are changed in Debug state. See A64 instructions that are changed in Debug state and T32 instructions that are changed in Debug state.
- Instructions that are unchanged in Debug state. See A64 instructions that are unchanged in Debug state and T32 instructions that are unchanged in Debug state. Where these sections describe the behavior of an instruction as unchanged in Debug state, or having the same behavior as in Non-debug state, and that instruction would generate an exception in Non-debug state, then the behavior of the exception is as described in Exceptions in Debug state.
- Instructions that are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state. See A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state and T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state.

All T32 instructions are treated as unconditional, regardless of PSTATE.IT. See PSTATE in Debug state.

If FEAT_RME is implemented and EDSCR.SDD == 1, then exceptions from Non-secure, Secure, Realm state are never taken to Root state. If FEAT_RME is not implemented and EDSCR.SDD == 1, then an exception from Non-secure state is never taken to Secure state.

See Security in Debug state.

H2.4.2.1 Executing A64 instructions in Debug state

The following sections describe the behavior of the A64 instructions in Debug state:

- A64 instructions that are changed in Debug state.
- A64 instructions that are unchanged in Debug state.
- A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.2.1.1 A64 instructions that are changed in Debug state

The following instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- `DCPS`.

  Note

  `DCPS` can be UNDEFINED in certain conditions in Debug state. See DCPS<n>.

- `DRPS`.
- `MRS` (DLR_EL0), `MRS` (DSPSR_EL0), `MSR` (DLR_EL0), `MSR` (DSPSR_EL0)
- If FEAT_SME2 is implemented, then `MOVT` (table to scalar) and `MOVT` (scalar to table).

The following instructions are defined in both Non-Debug and Debug states and have changed behavior in Debug state:

- If FEAT_SVE is implemented, then `CMPNE`(immediate) with byte element size.

  Note

  In Debug state, `CMPNE`(immediate) with byte element size sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values. However, the instruction is unchanged with respect to the SVE vector and SVE predicate source and destination registers.

For more information, see Debug state operations, DCPS, DRPS, MRS, MSR.

H2.4.2.1.2 A64 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state:

Any instruction that is UNDEFINED in Non-debug state

This list of instructions excludes:

- Any instruction listed in A64 instructions that are changed in Debug state.
- Any instruction listed in A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state that is UNDEFINED because an enable or disable bit is not RES0 or RES1

Instructions that move System or Special-purpose registers to or from a general-purpose register

This list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes PSTATE access instructions.

These instructions are:

- `MRS <special_reg>`, `MSR <special_reg>`.

  Note

  This does not include NZCV, DAIF, DAIFSet, DAIFClr, SPSel, CurrentEL, PAN, UAO, DIT, TCO, and PM.

- `MRS <system_reg>`, `MSR <system_reg>`.
- If FEAT_SYSREG128 is implemented, then `MRRS <system_reg>`, `MSRR <system_reg>`.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `FMOV` (between a general-purpose register and a half-precision register).
- `FMOV` (between a general-purpose register and a single-precision register).
- `FMOV` (between a general-purpose register and a double-precision register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `FMOV` (between a general-purpose register and a SIMD element).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `INS` (from a general-purpose register to a SIMD element).
- `UMOV` (from a SIMD element to a general-purpose register).

SVE instructions

If FEAT_SVE is implemented, then these instructions are:

- `CPY` (immediate, zeroing) with byte element size and a shift amount of 0.
- `DUP`(scalar).
- `EXT`, destructive variant.
- `INSR`(scalar).
- `PMOV`(to predicate).
- `PMOV`(to vector).
- `PTRUE` with ALL constraint and byte element size.
- `RDFFR`(unpredicated).
- `RDVL`.
- `WRFFR`.

SME instructions

If FEAT_SME is implemented, then these instructions are:

- `MOVA` (tile to vector, single), `MOVA` (vector to tile, single).
- `MRS SVCR`, `MSR SVCR`.
- `RDSVL`.

Barriers These instructions are:

- `DMB`.
- `DSB`.
- `ISB`.
- `CSDB`.
- `SSBB`.
- `PSSBB`.
- If FEAT_SB is implemented, then `SB`.
- If FEAT_SPE is implemented, then `PSB`.
- If FEAT_TRBE is implemented, then `TSB`.
- If FEAT_RAS is implemented, then `ESB`.

Memory access instructions at various access sizes

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Unscaled (9-bit signed) immediate offset.
  - Immediate (9-bit signed) post-indexed.
  - Immediate (9-bit signed) pre-indexed.
  - Unprivileged (9-bit signed).
- Not literal.
- One of the following types:
  - (Single) register.
  - Exclusive.
  - Exclusive pair.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive pair.
- 32-bit and 64-bit target register variants.

These instructions are:

- `LDR`, `LDRB`, `LDRH`, `LDRSB`, `LDRSH`, `LDRSW` (immediate, not literal).
- `LDUR`, `LDURB`, `LDURH`, `LDURSB`, `LDURSH`, `LDURSW` (immediate).
- `LDTR`, `LDTRB`, `LDTRH`, `LDTRSB`, `LDTRSH`, `LDTRSW` (immediate).
- `LDAR`, `LDARB`, `LDARH`, `LDXR`, `LDXRB`, `LDXRH`, `LDAXR`, `LDAXRB`, `LDAXRH`.
- `LDXP`, `LDAXP`.
- `STR`, `STRB`, `STRH` (immediate).
- `STUR`, `STURB`, `STURH` (immediate).
- `STTR`,`STTRB`, `STTRH` (immediate).
- `STLR`, `STLRB`, `STLRH`, `STXR`, `STXRB`, `STXRH`, `STLXR`, `STLXRB`, `STLXRH`.
- `STXP`, `STLXP`.
- If FEAT_LOR is implemented, then:
  - `LDLAR`, `LDLARB`, `LDLARH`.
  - `STLLR`, `STLLRBB`, `STLLRH`.
- If FEAT_LSE is implemented, then:
  - `CAS`, `CASB`, `CASH`, `CASP`.
  - `SWP`, `SWPB`, `SWPH`.
  - `LDADD`, `LDADDB`, `LDADDH`.
  - `LDCLR`, `LDCLRB`, `LDCLRH`.
  - `LDEOR`, `LDEORB`, `LDEORH`.
  - `LDSET`, `LDSETB`, `LDSETH`.
  - `LDSMAX`, `LDSMAXB`, `LDSMAXH`.
  - `LDSMIN`, `LDSMINB`, `LDSMINH`.
  - `LDUMAX`, `LDUMAXB`, `LDUMAXH`.
  - `LDUMIN`, `LDUMINB`, `LDUMINH`.
  - `STADD`, `STADDB`, `STADDH`.
  - `STCLR`, `STCLB`, `STCLRH`.
  - `STEOR`, `STEORB`, `STEORH`.
  - `STSET`, `STSETB`, `STSETH`.
  - `STSMAX`, `STSMAXB`, `STSMAXH`.
  - `STSMIN`, `STSMINB`, `STSMINH`.
  - `STUMAX`, `STUMAXB`, `STUMAXH`.
  - `STUMIN`, `STUMINB`, `STUMINH`.
- If FEAT_LSE128 is implemented, then:
  - `LDCLRP`.
  - `LDSETP`.
  - `SWPP`.
- If FEAT_LRCPC is implemented, then `LDAPR, LDAPRB, LDAPRH`.
- If FEAT_LRCPC2 is implemented, then:
  - `LDAPURH, LDAPURSH, LDAPUR, LDAPURSW, LDAPURSB, LDAPURB`.
  - `STLUR, STLURH, STLURB`.
- If FEAT_LRCPC3 is implemented, then:
  - `LDAP1`(SIMD&FP), `LDAPR, LDAPUR` (SIMD&FP), `LDAPUR`, `LDIAPP, STLR`.
  - `STL1`(SIMD&FP), `STILP`, `STLUR` (SIMD&FP).
- If FEAT_LS64 is implemented, regardless of whether FEAT_LS64WB is implemented, then:
  - `LD64B`.
  - `ST64B`.
- If FEAT_LS64_V is implemented, then `ST64BV`.
- If FEAT_LS64_ACCDATA is implemented, then `ST64BV0`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_LSUI is implemented:
  - LDTXR, STTXR.
  - LDATXR, STLTXR.
  - CAST, CASAT, CASALT, CASLT.
  - CASPT, CASPAT, CASPALT, CASPLT.
  - SWPT, SWPTA, SWPTL, SWPTAL.
  - LDTADD, LDTADDA, LDTADDAL, LDTADDL.
  - LDTSET, LDTSETA, LDTSETAL, LDTSETL.
  - LDTCLR, LDTCLRA, LDTCLRAL, LDTCLRL.
  - STTADD, STTADDL.
  - STTSET, STTSETL.
  - STTCLR, STTCLRL.

Move immediate to general-purpose register

These instructions are:

- MOVZ, MOVN, MOVK (immediate).
- MOV (between a general-purpose register and the stack pointer).

System instructions, Send Event, NOP, Clear Exclusive, and Prediction

In this context, the System instructions are the Cache maintenance instructions, TLB maintenance instructions, Address translation instructions, the prediction restriction instructions, and the check feature instruction.

These instructions are:

- IC.
- DC.
- TLBI and, if FEAT_D128 is implemented, then TLBIP.
- AT.
- SEV, SEVL.
- NOP.
- If FEAT_CLRBHB is implemented, then CLRBHB.
- CLREX.
- CFP.
- If FEAT_SPECRES2 is implemented, then COSP.
- CPP.
- DVP.
- If FEAT_CHK is implemented, then CHKFEAT.

Pointer authentication instructions

These instructions are:

- If FEAT_PAuth is implemented, then:
  - AUTIA, AUTIAZ, AUTIZA.
  - AUTIB, AUTIBZ, AUTIZB.
  - AUTDA, AUTDZA.
  - AUTDB, AUTDZB.
  - PACIA, PACIAZ, PACIZA.
  - PACIB, PACIBZ, PACIZB.
  - PACDA, PACDZA.
  - PACDB, PACDZB.
  - PACGA.
  - XPACD, XPACI, XPACLRI.
- If FEAT_PAuth_LR is not implemented or PSTATE.PACM is 0, then:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - AUTIASP, AUTIA1716.
  - AUTIBSP, AUTIB1716.
  - PACIASP, PACIA1716.
  - PACIBSP, PACIB1716.
- If FEAT_PAuth_LR is implemented, then:
  - PACIA171615 and PACIB171615.
  - AUTIA171615 and AUTIB171615.

Memory Tagging Extension instructions

These instructions are:

- If FEAT_MTE is implemented, then:
  - ADDG.
  - SUBG.
  - STG (signed offset).
  - STZG (signed offset).
  - ST2G (signed offset).
  - STZ2G (signed offset).
  - STGP (signed offset).
  - LDG.
- If FEAT_MTE2 is implemented, then:
  - LDGM.
  - STGM.
  - STZGM.

BRBE instructions

These instructions are:

- If FEAT_BRBE is implemented, then:
  - BRB IALL.
  - BRB INJ.

GCS instructions

These instructions are:

- If FEAT_GCS is implemented, then:
  - GCSB.
  - GCSPUSHM, GCSPOPM.
  - GCSPUSHX, GCSPOPX.
  - GCSSS1, GCSSS2.
  - GCSSTR, GCSSTTR.

H2.4.2.1.3 A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state

This subsection describes all instructions that are not listed in either:

- A64 instructions that are changed in Debug state.
- A64 instructions that are unchanged in Debug state.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for A64 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed. An example of this is `CCMP`.

Exception-generating instructions

These instructions are:

- `SVC`.
- `HVC`.
- `SMC`.
- `BRK`.
- `HLT`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- `SVC` behaves as `DCPS1`.
- `HVC` behaves as `DCPS2`.
- `SMC` behaves as `DCPS3`.
- They generate the exception that the instruction would generate in Non-debug state. The exception is taken as described in Exceptions in Debug state.

Note

`SMC` does not generate a Secure Monitor Call exception if EDSCR.SDD is 1.

Instructions that explicitly write to the PC (branches)

These instructions are:

- `B`, `B.cond`, `BL`, `BLR`, `BR`, `CBZ`, `CBNZ`, `RET`, `TBZ`, `TBNZ`.
- If FEAT_HBC is implemented, then `BC.cond`.
- If FEAT_PAuth is implemented, then `RETAA`, `RETAB`, `BRAA`, `BRAB`, `BLRAA`, `BLRAB`, `BLRAAZ`, `BLRABZ`.
- If FEAT_PAuth_LR is implemented, then `RETAASPPC`, `RETABSPPC`, `RETAASPPCR`, `RETABSPPCR`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching and set DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Exception return and related instructions

These instructions are:

- `ERET`.
- If FEAT_PAuth is implemented, then `ERETAA`, `ERETAB`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching, set DSPSR_EL0 and DLR_EL0 to UNKNOWN values, and either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- — Execute the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
- — Do not change the Exception level.

Instructions that request entry to a low-power state

These instructions are:

- WFE, WFI.
- If FEAT_WFxT is implemented, then WFET, WFIT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See Configurable instruction controls.
- A WFE instruction clears the Event register if it is set.

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:

- LDR (literal), LDRSW (literal).
- ADR, ADRP.
- PRFM (literal).
- If FEAT_PAuth_LR is implemented, then:
  - — AUTIASPPC (immediate), AUTIBSPPC (immediate).
  - — PACIASPPC, PACIBSPPC.
  - — PACNBIASPPC, PACNBIBSPPC.
  - — RETAASPPC, RETABSPPC.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, using an UNKNOWN value for the PC operand.

Instructions that explicitly modify PSTATE, other than DCPS and DRPS

These instructions are:

- ADDS, SUBS, ADCS, SBCS, ANDS, BICS, CCMN, CCMP.
- FCMP, FCMPE, FCCMP, FCCMPE.
- MSR DAIFSet (immediate), MSR DAIFClr (immediate), MSR SPSel (immediate).
- MSR NZCV (register), MSR DAIF (register), MSR SPSel (register).
- If FEAT_PAN is implemented, then:
  - — MSR PAN (immediate).
  - — MSR PAN (register).
- If FEAT_UAO is implemented, then:
  - — MSR UAO (immediate).
  - — MSR UAO (register).
- If FEAT_FlagM is implemented, then:
  - — CFINV.
  - — RMIF.
  - — SETF8.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - SETF16.
- If FEAT_DIT is implemented, then MSR DIT.
- If FEAT_FlagM2 is implemented, then:
  - AXFLAG.
  - XAFLAG.
- If FEAT_MTE is implemented, then MSR TCO.
- If FEAT_RNG is implemented, then:
  - MRS RNDR.
  - MRS RNDRRS.
- If FEAT_NMI is implemented, then:
  - MSR ALLINT (immediate).
  - MSR ALLINT (register).
- If FEAT_EBEP is implemented, then:
  - MSR PM (immediate).
  - MSR PM (register).
- If FEAT_MOPS is implemented, then:
  - CPYFP, CPYFPN, CPYFPRN, CPYFPRT, CPYFPRTN, CPYFPRTRN, CPYFPRTWN, CPYFPT, CPYFPTN, CPYFPTRN, CPYFPTWN, CPYFPWN, CPYFPWT, CPYFPWTN, CPYFPWTRN, CPYFPWTWN.
  - CPYP, CPYPN, CPYPRN, CPYPRT, CPYPRTN, CPYPRTRN, CPYPRTWN, CPYPT, CPYPTN, CPYPTRN, CPYPTWN, CPYPWN, CPYPWT, CPYPWTN, CPYPWTRN, CPYPWTWN.
  - SETP, SETPN, SETPT, SETPTN.
  - SETGP, SETGPN, SETGPT, SETGPTN.
- If FEAT_SVE is implemented, then:
  - ANDS, BICS, EORS, NANDS, NORS, ORNS, ORRS, PTEST.
  - BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS.
  - CMPEQ, CMPGE, CMPGT, CMPHI, CMPHS, CMPLE, CMPLO, CMPLS, CMPLT, CMPNE.
  - MATCH, NMATCH.
  - PFIRST, PNEXT, CTERMEQ, CTERMNE.
  - PTRUES.
  - RDFFRS.
  - WHILEGE, WHILEGT, WHILEHI, WHILEHS (predicate).
  - WHILELE, WHILELO, WHILELS, WHILELT (predicate).
  - WHILERW, WHILEWR.
- If FEAT_SME2 or FEAT_SVE2p1 is implemented, then:
  - WHILEGE, WHILEGT, WHILEHI, WHILEHS (predicate pair).
  - WHILELE, WHILELO, WHILELS, WHILELT (predicate pair).
  - WHILEGE, WHILEGT, WHILEHI, WHILEHS (predicate as counter).
  - WHILELE, WHILELO, WHILELS, WHILELT (predicate as counter).
- If FEAT_PAuth_LR is implemented, then:
  - PACM.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, setting DSPSR_EL0, DLR_EL0, and any PSTATE field written by the instruction to UNKNOWN values.

Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields

These instructions are:

- CSEL, CSINC, CSINV, CSNEG, CCMN, CCMP, FCSEL, FCCMP, FCCMPE.
- ADC, ADCS, SBC, SBCS.
- MRS NZCV, MRS DAIF, MRS SPSel, MRS CurrentEL.
- If FEAT_PAN is implemented, then MRS PAN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_UAO is implemented, then `MRS UAO`.
- If FEAT_FlagM is implemented, then `CFINV`.
- If FEAT_FlagM2 is implemented, then `AXFLAG`, `XAFLAG`.
- If FEAT_DIT is implemented, then `MRS DIT`.
- If FEAT_MTE is implemented, then `MRS TCO`.
- If FEAT_NMI is implemented, then `MRS ALLINT`.
- If FEAT_EBEP is implemented, then `MRS PM`.
- If FEAT_MOPS is implemented, then:
  - `CPYFM`, `CPYFMN`, `CPYFMRN`, `CPYFMRT`, `CPYFMRTN`, `CPYFMRTRN`, `CPYFMRTWN`, `CPYFMT`, `CPYFMTN`, `CPYFMTRN`, `CPYFMTWN`, `CPYFMWN`, `CPYFMWT`, `CPYFMWTN`, `CPYFMWTRN`, `CPYFMWTWN`.
  - `CPYFE`, `CPYFEN`, `CPYFERN`, `CPYFERT`, `CPYFERTN`, `CPYFERTRN`, `CPYFERTWN`, `CPYFET`, `CPYFETN`, `CPYFETRN`, `CPYFETWN`, `CPYFEWN`, `CPYFEWT`, `CPYFEWTN`, `CPYFEWTRN`, `CPYFEWTWN`.
  - `CPYM`, `CPYMN`, `CPYMRN`, `CPYMRT`, `CPYMRTN`, `CPYMRTRN`, `CPYMRTWN`, `CPYMT`, `CPYMTN`, `CPYMTRN`, `CPYMTWN`, `CPYMWN`, `CPYMWT`, `CPYMWTN`, `CPYMWTRN`, `CPYMWTWN`.
  - `CPYE`, `CPYEN`, `CPYERN`, `CPYERT`, `CPYERTN`, `CPYERTRN`, `CPYERTWN`, `CPYET`, `CPYETN`, `CPYETRN`, `CPYETWN`, `CPYEWN`, `CPYEWT`, `CPYEWTN`, `CPYEWTRN`, `CPYEWTWN`.
  - `SETM`, `SETMN`, `SETMT`, `SETMTN`.
  - `SETE`, `SETEN`, `SETET`, `SETETN`.
  - `SETGM`, `SETGMN`, `SETGMT`, `SETGMTN`.
  - `SETGE`, `SETGEN`, `SETGET`, `SETGETN`.
- If FEAT_SVE is implemented, then all SVE instructions that use a predicate.
- If FEAT_THE is implemented, then `RCW` and `RCWS`.
- If FEAT_PAuth_LR is implemented, then:
  - If PSTATE.PACM is 1, then `PACIASP`, `PACIBSP`, `PACIA1716`, `PACIB1716`, `AUTIASP`, `AUTIBSP`, `AUTIA1716`, `AUTIB1716`, `RETAA`, and `RETAB`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state:
  - For the conditional operations and those that use Condition flags as an input, these instructions use UNKNOWN values for the Condition flag.
  - For the `MRS` instruction, they return an UNKNOWN value.
  - For the `RCW` and `RCWS` instructions, they return an UNKNOWN value.
  - For the `PACIASP` and `PACIBSP` instructions when PSTATE.PACM is 1, they use an UNKNOWN value for the PC operand.

Hint instructions

If FEAT_DGH is implemented, then this instruction is:

- `DGH`.

This instruction behaves in one of the following ways:

- It executes as a `NOP`.
- It executes as in Non-debug state.

All other instructions

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state.

H2.4.2.2 Executing T32 instructions in Debug state

The following sections describe the behavior of the T32 instructions in Debug state:

- T32 instructions that are changed in Debug state.
- T32 instructions that are unchanged in Debug state.
- T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state.

H2.4.2.2.1 T32 instructions that are changed in Debug state

The following T32 instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- `DCPS`.

  Note

  `DCPS` can be UNDEFINED in certain conditions in Debug state. See DCPS<n>.

- `MRC p15,3,<Rt>,c4,c5,0` (DSPSR).
- `MCR p15,3,<Rt>,c4,c5,0` (DSPSR).
- `MRC p15,3,<Rt>,c4,c5,1` (DLR).
- `MCR p15,3,<Rt>,c4,c5,1` (DLR).

In addition, `ERET` executes the DRPS operation in Debug state.

For more information, see Debug state operations, DCPS, DRPS, MRS, MSR.

H2.4.2.2.2 T32 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state. Any T32 instruction that uses the PC or APSR.{N, Z, C, V} as the source or destination register is not included in the list. Moreover, the list includes only the 32-bit T32 encodings.

Any instruction that is UNDEFINED in Non-debug state

The list of instructions:

- Excludes any instruction listed in T32 instructions that are changed in Debug state.
- Excludes any instruction listed in T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state that is UNDEFINED because an enable or disable bit is not RES0 or RES1.

Instructions that move System or Special-purpose registers to or from a general-purpose register

The list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes APSR and CPSR access instructions.
- Excludes instructions for accessing banked registers for the current mode.

These instructions are:

- `MRS <banked_reg>, MSR <banked_reg>`.

  Note

  This does not apply to cases which are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state in the current mode.

- `MRC, MCR`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This includes all allocated System registers in the (`coproc == 0b111x`) encoding space other than an `MRC` move to APSR_nzcv.

- `MRS SPSR`, `MSR SPSR_fsxc` (register).
- `VMRS <vfp_system_reg>, VMSR <vfp_system_reg>`.

Note

This includes all allocated Advanced SIMD and floating-point System registers, other than an a `VMRS` move to APSR_nzcv.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `VMOV` (between a general-purpose register and a single-precision register).
- `VMOV` (between a general-purpose register and a doubleword floating-point register).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `VMOV` (between a general-purpose register and a scalar).

Barriers These instructions are:

- `CSDB`.
- `DMB`.
- `DSB`.
- `ISB`.
- `PSSBB`.
- `SSBB`.
- If FEAT_RAS is implemented, then `ESB`.
- If FEAT_SB is implemented, then `SB`.

Memory access instructions at various access sizes

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Immediate (8-bit or 12-bit) offset.
  - Immediate (8-bit) post-indexed.
  - Immediate (8-bit) pre-indexed.
  - Unprivileged (8-bit).
- Not literal.
- One of the following types:
  - (Single) register.
  - Dual.
  - Exclusive.
  - Exclusive doubleword.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive doubleword.

These instructions are:

- `LDR.W, LDRB.W, LDRH.W, LDRD, LDRSB.W, LDRSH.W` (immediate, not literal).
- `LDRT, LDRBT, LDRHT, LDRSBT, LDRSHT`(immediate).
- `LDREX, LDREXB, LDREXH, LDA, LDAB, LDAH, LDAEX, LDAEXB, LDAEXH`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- LDREXD, LDAEXD.
- STR.W, STRB.W, STRH.W, STRD (immediate).
- STRT, STRBT, STRHT (immediate).
- STREX, STREXB, STREXH, STL, STLB, STLH, STLEX, STLEXB, STLEXH.
- STREXD, STLEXD.

Move to general-purpose register

These instructions are:

- MOVW, MOVT(immediate).

System instructions, Send Event, NOP, and Clear Exclusive

The System instructions are Cache maintenance instructions, TLB maintenance instructions, and Address translation instructions. These are encoded in the (coproc == 0b1111) System register encoding space.

These instructions are:

- ICIALLU, ICIALLUIS, ICIMVAU.
- DCCIMVAC, DCCISW, DCCMVAC, DCCMVAU, DCCSW, DCIMVAC, DCISW.
- TLBIALL, TLBIALLH, TLBIALLHIS, TLBIALLIS, TLBIALLNSNH, TLBIALLNSNHIS, TLBIASID, TLBIASIDIS, TLBIIPAS2, TLBIIPAS2IS, TLBIIPAS2L, TLBIIPAS2LIS, TLBIMVA, TLBIMVAA, TLBIMVAAIS, TLBIMVAAL, TLBIMVAALIS, TLBIMVAH, TLBIMVAHIS, TLBIMVAIS, TLBIMVAL, TLBIMVALH, TLBIMVALHIS, TLBIMVALIS.
- ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW, ATS1CPR, ATS1CPW. ATS1CUR, ATS1CUW, ATS1HR, ATS1HW.
- BPIALL, BPIALLIS, BPIMVA.
- SEV.W, SEVL.W.
- NOP.W.
- CLREX.

H2.4.2.2.3 T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state

This subsection describes all instruction not listed in either:

- T32 instructions that are changed in Debug state.
- T32 instructions that are unchanged in Debug state.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction generates an Undefined Instruction exception.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for T32 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed.

Exception-generating instructions

These instructions are:

- SVC.
- HVC.
- SMC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `UDF`.
- `BKPT`.
- `HLT`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- `SVC` behaves as `DCPS1`.
- `HVC` behaves as `DCPS2`.
- `SMC` behaves as `DCPS3`.
- They generate the exception the instruction would generate in Non-debug state. The exception is taken as described in Exceptions in Debug state.

Note

`SMC` must not generate a Secure Monitor Call exception if EDSCR.SDD is 1.

Instructions that explicitly write to the PC (branches)

These instructions are:

- `B, B` (conditional), `CBZ, CBNZ BL`.
- `BX, BLX` (register or immediate).
- `BXJ, TBB, TBH`.
- `MOV pc` and related instructions.
- `LDR pc, LDM` (with a register list includes the PC), `POP` (with a register list that includes the PC).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching and set DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, to UNKNOWN values.

Exception return and related instructions, other than ERET

These instructions are:

- `SRS, RFE, SUBS pc, 1r`, and related instructions.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching, setting DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, to UNKNOWN values, and either:
  - Execute the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
  - Not changing Exception level or PE mode.

Instructions that request entry to a low-power state

These instructions are:

- `WFE`, `WFI`, `WFET`, `WFIT`

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See Configurable instruction controls.
- A `WFE` instruction is permitted to clear the Event register if it is set.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:

- `LDR` (literal), `LDRB` (literal), `LDRH` (literal), `LDRSB` (literal), `LDRSH` (literal).
- `ADR`, `ADRL, ADRH`.
- `PLD` (literal), `PLI` (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state using an UNKNOWN value for the PC operand.

Instructions that explicitly modify PSTATE, other than `DCPS` and `ERET`

These instructions are:

- `CMP, TST, TEQ, CMN`.
- `<opc>S`.
- `MRC p14,0,APSR_nzcv,c0,c1,0` (accessing DBGDSCRint).
- `CPS, SETEND, IT`.
- `MSR CPSR` (immediate), `MSR CPSR` (register), `MSR APSR` (immediate), `MSR APSR` (register).
- `VMRS APSR_nzcv, FPSCR`.
- `QADD, QDADD, QSUB, QDSUB`.
- `SMLABB`, `SMLABT`, `SMLATB`, `SMLATT`, `SMLAD`, `SMLAWB`, `SMLAWT`, `SMLSD`, `SMUAD`.
- `SSAT, SSAT16, USAT, USAT16`.
- `SADD`, `SADD8`, `SADD16`, `SASX`, `SSAX`, `SSUB`, `SSUB8`, `SSUB16`.
- `UADD`, `UADD8`, `UADD16`, `UASX`, `USAX`, `USAUB`, `USUN8`, `USUB16`.
- If FEAT_PAN is implemented, then `SETPAN`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, setting DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, and any PSTATE field written by the instruction to UNKNOWN values.

Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields

These instructions are:

- `SEL`, `VSEL`.
- `ADC`, `SBC`, all instructions with an `RRX` shift.
- `MRS CPSR`".

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state:
  - For the conditional operations and those using the PSTATE.C flag as an input, these instructions use an UNKNOWN value for the Condition flag.
  - For the `MRS` instruction, they return an UNKNOWN value.

All other instructions

These instructions behave in one of the following ways:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- They are UNDEFINED.
- They execute as a NOP.
- They have the same behavior as in Non-debug state.

Note

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are CONSTRAINED UNPREDICTABLE in Debug state. This includes some T32 instructions that specify R15 as a destination or source register.
Architectural Constraints on UNPREDICTABLE Behaviors describes the CONSTRAINED UNPREDICTABLE behavior for these instructions. In Debug state these CONSTRAINED UNPREDICTABLE choices are further restricted:

- Instructions that specify R15 as a destination register:
  - Are not permitted to branch, because the architecture does not define a branch operation in Debug state.
  - Might set DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, to UNKNOWN values.
  - Might have any of the other permitted behaviors.
- Instructions that specify R15 as a source operand:
  - Cannot use PC + offset, because there is no architecturally-defined PC in Debug state.
  - Might have any of the other permitted behaviors, including using an UNKNOWN value.

H2.4.3 Decode tables

The syntax in the decode tables is defined as follows:

1 The bit has a fixed value of 1.

0 The bit has a fixed value of 0.

!= The field has any value other than the value or values specified. The field might be an encoding field in the instruction whose value is supplied by the debugger.

Note

The instruction encodings in A64 Base Instruction Descriptions and T32 and A32 Base Instruction Set Instruction Descriptions might show these bits as (0) or (1). A debugger must set these bits to 0 or 1, as appropriate.

Any other value indicates an encoding field in the instruction whose value is supplied by the debugger. Some values might be reserved or UNDEFINED, in which case the instruction is UNDEFINED or CONSTRAINED UNPREDICTABLE in Debug state, as it is in Non-debug state.

For more information about the instruction encodings, see:

- A64 Base Instruction Descriptions.
- T32 and A32 Base Instruction Set Instruction Descriptions.

For information about the syntax used in Figure H2-1, Figure H2-2, Figure H2-3, and Figure H2-4, see:

- Common syntax terms.
- Assembler symbols.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.3.1 A64 instructions that are modified in Debug state

Figure H2-1 shows the A64 instructions that are modified in Debug state. For information of how these are packed in the EDITR, see the register description.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 0 1	0 1 0 0	1 0 1	imm16			0	0 0 !=00	DCPS<opt>
1 1 0 1	0 1 0 1	0 0 L 1	1 0 1 1	0 1 0 0	0 1 0 1	0 0 0	Rt	MRS|MSR accessing DSPSR_EL0
1 1 0 1	0 1 0 1	0 0 L 1	1 0 1 1	0 1 0 0	0 1 0 1	0 0 1	Rt	MRS|MSR accessing DLR_EL0
1 1 0 1	0 1 1 0	1 0 1 1	1 1 1 1	0 0 0 0	0 0 1 1	1 1 1 0	0 0 0 0	DRPS
0 0 1 0	0 1 0 1	size 0	imm5	1 0 0	Pg	Zn 1	Pd	CMPNE (immediate)
1 1 0 0	0 0 0 0	0 1 0 0	1 1 1 0	0 off3	0 0 1 1	1 1 1	Rt	MOVT (scalar to table)
1 1 0 0	0 0 0 0	0 1 0 0	1 1 0 0	0 off3	0 0 1 1	1 1 1	Rt	MOVT (table to scalar)

Figure H2-1 Modified A64 instructions in Debug state

H2.4.3.2 T32 instructions that are modified in Debug state

Figure H2-2 shows the T32 instructions that are modified in Debug state, with the first halfword on the left side and the second halfword on the right side. For information of how these are packed in the EDITR, see the register description.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 1 0	1 1 1 0	0 1 1 L	0 1 0 0	!=1111	1 1 1 1	0 0 0 1	0 1 0 1	MRC|MCR accessing DSPSR
1 1 1 0	1 1 1 0	0 1 1 L	0 1 0 0	!=1111	1 1 1 1	0 0 1 1	0 1 0 1	MRC|MCR accessing DLR
1 1 1 1	0 0 1 1	1 1 0 1	1 1 1 0	1 0 0 0	1 1 1 1	0 0 0 0	0 0 0 0	ERET
1 1 1 1	0 1 1 1	1 0 0 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 !=00	DCPS<opt>

Figure H2-2 Modified T32 instructions in Debug state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.3.3 A64 instructions that are unchanged in Debug state

This section shows the A64 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30 29 28	27 26	25 24	23 22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
0 0 0 0	0 1	0 0 1	0	1	1 1 1 1	0 1 0 1	0 imm6		Rd	RDVL
0 0 0 0	0 1	0 0 1	0	1	1 1 1 1	0 1 0 1	1 imm6		Rd	RDSVL
0 0 0 0	0 1	0 1 0	0	1	imm8h	0 0 0	imm8l	Zm	Zdn	EXT
0 0 0 0	0 1	0 1	0	0 1	Pg	0 0 0	imm8		Zd	CPY (immediate, zeroing)
0 0 0 0	0 1	0 1	size	1	0 0 0 0	0 0 1	1 1 0	Rn	Zd	DUP (scalar)
0 0 0 0	0 1	0 1	size	1	0 0 1 0	0 0 1	1 1 0	Rm	Zdn	INSR (scalar)
sf 0	0 1	0 0	0 1	0	0 0 0 0	0 0 0	0 0 0	1 1 1 1	Rd	MOV <Rn>,SP
sf 0	0 1	0 0	0 1	0	0 0 0 0	0 0 0	0 0 0	Rn	1 1 1 1 1	MOV SP,<Rn>
0 0	0 1	1 0	0 1	A	R 1	Rt2	0 0 1	0 Rn	Rt	LDCLRP
0 0	0 1	1 0	0 1	A	R 1	Rt2	0 0 1	1 0 Rn	Rt	LDSETP
0 0	0 1	1 0	0 1	A	R 1	Rt2	1 0 0	0 0 Rn	Rt	SWPP
0 0	1 0	0 1	0 1	0	0 0 1	1 0 0	1 1 1	1 0 0	0 0 0 0 0 0 Pd	RDFFR (unpredicated)
0 0	1 0	0 1	0 1	0	0 1 0	1 0 0	0 1 0	0 1 0 0	0 Pn	0 0 0 0 0 WRFFR
0 0	1 0	0 1	0 1	0	0 0 1	1 0 0	0 1 1	1 0 0	0 0 0 0 0 0 Pd	PTRUE
1 0 0	1 1 0	1 0	1 1	0	Rm	0 0 1	1 0 0	Rn	Rd	PACGA
sf !=01	1 0 0	1 0	1	hw	imm16				Rd	MOVN, MOVK, MOVZ
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0 0	0 1 0 0 1	1 0 0 0 0	GCSB
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 0 0 0	0 1 1 1 1	NOP
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 1 0 1 1	0 1 1 1 1	CLRBHB
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 0 1 0	L 1 1 1 1 1	SEV, SEVL
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 0 1 1	1 1 1 1 1	XPACLRI
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 0 1	op2 1 1 1 1 1	When FEAT_PAuth_LR is not implemented, or PSTATE.PACM is 0: PAC(IA|IB)1716, AUT(IA|IB)1716
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 1 0	!=010 1 1 1 1 1	CSDB, ESB, PSB
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 1 0	0 1 0 1 1 1 1 1	TSB
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 0 1 1	op2 1 1 1 1 1	PAC(IA|IB)Z, AUT(IA|IB)Z When FEAT_PAuth_LR is not implemented, or PSTATE.PACM is 0: PAC(IA|IB)SP, AUT(IA|IB)SP
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 0	0 1 0 1 0 0	1 1 1 1 1	CHKFEAT
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 1	CRm	0 1 0 1 1 1 1 1	CLREX
1 1 0	1 0 1	0 1	0	0	0 0 0 1	1 0 0	1 1	option	1 op2 1 1 1 1 1	DSB, DMB, ISB, SB, SSBB, PSSBB
1 1 0	1 0 1	0 1	0	0	0 1 0 1	1 0 1	1 1	0	1 1 1 1 0	Rt COSP
1 1 0	1 0 1	0 1	0	0	0 1 0 0	0 0 1	1 1	1	1 1 1 0 0	Rt GCSPUSHX
1 1 0	1 0 1	0 1	0	0	0 1 0 0	0 0 1	1 1	1	1 1 1 1 0	Rt GCSPOPX
1 1 0	1 0 1	0 1	0	0	0 1 0 1	1 0 1	1 1	1	0 1 1 0 0	Rt GCSSS1
1 1 0	1 0 1	0 1	0	0	0 1 0 1	1 0 1	1 1	1	0 0 0 0 0	Rt GCSPUSHM
1 1 0	1 0 1	0 1	0	0	0 0 1	op1	CRn	CRm	op2 Rt	IC, DC, TLBI, TLBIP, AT
1 1 0	1 1 0	0 1	0	0	0 1 1 1	1 1 0	0 0 0	1 1	Rn	Rt GCSSTR
1 1 0	1 1 0	0 1	0	0	0 1 1 1	1 1 0	0 0 1	1 1	Rn	Rt GCSSTTR
1 1 0	1 0 1	0 1	0	0	1 0 1 0	1 1 0	1 1	1	0 0 1	Rt GCSPOPM

Figure H2-3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 30 29 28	27 26	25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 0 1	0 1	0 1	0 0 1	0 1 0 1	1	0 1 1 1 0	1 1 1 0 1 1	Rt	GCSSS2
1 1 0 1	0 1	0 1	0 0 L	1 0	op1	CRn	CRm	op2	Rt	MRS|MSR accessing System register
1 1 0 1	0 1	0 1	0 0 L	1 1	op1	!=0100	CRm	op2	Rt	MRS|MSR accessing System register
1 1 0 1	0 1	0 1	0 0	0 0 0	0 0 1	0 1 0 0	0 0 1x	0 0 0	1 1 1 1 1	MSR PM, #<imm>
1 1 0 1	0 1	0 1	0 0	0 1 o0	op1	CRn	CRm	op2	Rt	MRS|MSR accessing PM
1 1 0 1	0 1	0 1	0 0 L	1 1	0 1 1	0 1 0 0	0 0 10	0 1 0	Rt	MRS|MSR accessing SVCR
1 1 0 1	0 1	0 1	0 0 L	1 1	op1	0 1 0 0	!=0010	op2	Rt	MRS|MSR accessing Special-purpose register
1 1 0 1	0 1	0 1	0 1 L	1 0	op1	CRn	CRm	op2	Rt	MRRS|MSRR
1 1 0 1	0 1	1 0	0 1 0	1 1 1 1	1	0 0 0 0	1 M 1 1 1 1	1 1 1 1 1	RETAA, RETAB
1 1 0 1	0 1	1 0	1 0 0	1 1 1 1	1	0 0 0 0	1 M 1 1 1 1	1 1 1 1 1	ERETAA, ERETAB
1 1 0 1	0 1	1 1	0 0 0	1 1 1 1	1	0 0 0 0	1 M	Rn	Rm	BRAA, BRAB
1 1 0 1	0 1	1 Z	0 0 1	1 1 1 1	1	0 0 0 0	1 M	Rn	Rm	BLRAA, BLRAB, BLRAAZ, BLRABZ
1 1 0	1 1 0	1 0	1 1 0	0 0 0 0	1	0 0 0	opc	Rn	Rd	PAC(IA|IB|DA|DB), AUT(IA|IB|DA|DB)
1 1 0	1 1 0	1 0	1 1 0	0 0 0 0	1	0 0 1	opc	1 1 1 1 1	Rd	PAC(IZA|IZB|DZA|DZB), AUT(IZA|IZB|DZA|DZB)
1 1 0	1 1 0	1 0	1 1 0	0 0 0 0	1	0 1 0 0 0	op	1 1 1 1 1	Rd	XPAC(I|D)
1 1 0	1 1 0	1 0	1 1 0	0 0 0 0	1	1 0 0 0 1	op	1 1 1 1 1	Rd	PACI(A|B)171615
1 1 0	1 1 0	1 0	1 1 0	0 0 0 0	1	1 0 1 1 1	op	1 1 1 1 1	Rd	AUTI(A|B)171615
size	0 0 1	0 0	o2 L	0	Rs	o0	Rt2	Rn	Rt	LD(A|LA|X|AX)R{B|H}, ST(L|LL|X|LX)R{B|H}, CAS{A|L|AL}{B|H}
size	0 0 1	0 0	o2 L	1	Rs	o0	Rt2	Rn	Rt	LD{A}XP, ST{L}XP, CASP{A|L|AL}
0 x	0 1 1	0 0 1	opc	0	imm9		0 0	Rn	Rt	LDAPUR(B|H|SB|SH), STLUR(B|H)
1 0	0 1 1	0 0 1	!=11	0	imm9		0 0	Rn	Rt	LDAPUR{SW}, STLUR
1 1	0 1 1	0 0 1	!=1x	0	imm9		0 0	Rn	Rt	LDAPUR, STLUR
0 x	1 1 1	0 0 0	opc	0	imm9		0 0	Rn	Rt	LDUR(B|H|SB|SH),STUR(B|H)
1 0	1 1 1	0 0 0	!=11	0	imm9		0 0	Rn	Rt	LDUR{SW},STUR
1 1	1 1 1	0 0 0	!=1x	0	imm9		0 0	Rn	Rt	LDUR,STUR
size	1 1 1	0 0 0	opc	0	imm9		1 0	Rn	Rt	LDTR{B|H|SB|SH|SW}, STTR{B|H}
size	1 1 1	0 0 0	opc	0	imm9		P 1	Rn	Rt	LDR{B|H|SB|SH|SW}, STR{B|H}
size	1 1 1	0 0 0	A R	1	Rs	0	opc	0 0	Rn	Rt	LD<op>{A|L|AL}{B|H}, ST<op>{A|L|AL}{B|H}
size	1 1 1	0 0 0	A R	1	Rs	1	0 0 0 0	0	Rn	Rt	SWP{A|L|AL}{B|H}
size	1 1 1	0 0 0	1 0	1	Rs	1	1 0 0 0	0	Rn	Rt	LDAPR{B|H}
0 1 0	0 1 1	1 0	0 0 0	imm5		0	0 0 1	1 1	Rn	Rd	INS <Vd>.<Ts>[<index>],<R><n>
0 Q 0	0 1 1	1 0	0 0 0	imm5		0	0 1 1	1 1	Rn	Rd	UMOV <R><d>,<Vn>.<Ts>[<index>]
0 0 0	1 1 1	1 0	0 0 1	0 0	1 1 op	0	0 0 0	0 0	Rn	Rd	FMOV <Sd>,<Wn>, FMOV <Wd>,<Sn>
0 0 0	1 1 1	1 0	1 1 1	0 0	1 1 op	0	0 0 0	0 0	Rn	Rd	FMOV <Hd>, <Wn>, FMOV <Wd>, <Hn>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Description
0	S	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	x	Rn						Rt				RCW{S}CAS{P}{A|L|AL}
0	S	0	1	1	0	0	1	A	R	1	Rt2					1	0	1	0	0	0	Rn						Rt				RCW{S}SWPP{A|L|AL}
0	S	0	1	1	0	0	1	A	R	1	Rt2					1	0	x	1	0	0	Rn						Rt				RCW{S}(CLRP|SETP){A|L|AL}
0	x	1	1	1	0	0	0	A	R	1	Rs					1	0	1	0	0	0	Rn						Rt				RCW{S}SWP{A|L|AL}
0	x	1	1	1	0	0	0	A	R	1	Rs					1	0	x	1	0	0	Rn						Rt				RCW{S}(CLR|SET){A|L|AL}
1	0	0	1	1	1	1	0	ft	1	1	0	0	1	1	op	0	0	0	0	0	0	Rn						Rd				FMOV <Dd|Hd>,<Xn>, FMOV <Xd>,<Dn|Hn>
1	0	0	1	1	1	1	0	1	0	1	0	1	1	1	op	0	0	0	0	0	0	Rn						Rd				FMOV <Vd>.D[1],<Xn> FMOV <Xd>,<Vn>.D[1]
1	0	0	1	0	0	0	1	1	0	uimm6				(0)	(0)	uimm4						Xn						Xd				ADDG
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Q	V	Rs			Pg				Zn			0	off4				MOVA (vector to tile, single)
1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	Q	V	Rs			Pg			0	off4				Zd				MOVA (tile to vector, single)
1	1	0	1	1	0	0	1	0	0	1	imm9									1	0	Xn						Xd				STG - Signed offset
1	1	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn						Xd				STGM
1	1	0	1	1	0	0	1	1	0	1	imm9									1	0	Xn						Xd				ST2G - Signed offset
1	1	0	1	1	0	0	1	0	1	1	imm9									1	0	Xn						Xd				STZG - Signed offset
1	1	0	1	1	0	0	1	1	1	1	imm9									1	0	Xn						Xd				STZ2G - Signed offset
1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn						Xd				STZGM
0	1	1	0	1	0	0	1	0	0	simm7						Xt2						Xn						Xt				STGP - Signed offset
1	1	0	1	0	0	0	1	1	0	uimm6						op3		uimm4				Xn						Xd				SUBG
1	1	0	1	1	0	0	1	0	1	1	imm9									0	0	Xn						Xd				LDG
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	Xn						Xd				LDGM
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	0	0	Rn						Rt				ST64B
1	1	1	1	1	0	0	0	0	0	1	Rs					1	0	1	0	0	0	Rn						Rt				ST64BV0
1	1	1	1	1	0	0	0	0	0	1	Rs					1	0	1	1	0	0	Rn						Rt				ST64BV
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0	0	Rn						Rt				LD64B

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.3.4 T32 instructions that are unchanged in Debug state

Figure H2-4 shows the T32 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions. It shows the T32 instructions with the first halfword on the left side and the second halfword on the right side.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 1 0	1 1 0 0	0 1 0 op	!=1111	!=1111	1 0 1 1	0 0 M 1	Vm	VMOV <Dm>,<Rt>,<Rt2> VMOV <Rt>,<Rt2>,<Dm>
1 1 1 0	1 1 1 0	0 0 0 op	Vn	!=1111	1 0 1 0	N 0 0 1	0 0 0 0	VMOV <Sn>,<Rt>, VMOV <Rt>,<Sn>
1 1 1 0	1 1 1 0	0 opc 0	Vd	!=1111	1 0 1 1	D opc2 1	0 0 0 0	VMOV.<size> <Dd>[<x>],<Rt>
1 1 1 0	1 1 1 0	U opc 1	Vn	!=1111	1 0 1 1	D opc2 1	0 0 0 0	VMOV.<dt> <Rt>,<Dd>[<x>]
1 1 1 0	1 1 1 0	1 1 1 op	reg	!=1111	1 0 1 0	0 0 0 1	0 0 0 0	VMRS, VMSR
1 1 1 0	1 1 0 0	0 1 0 op	!=1111	!=1111	1 1 1 cp	opc1	CRm	MCRR|MRRC accessing System registers
1 1 1 0	1 1 1 0	opc1 op	CRn	!=1111	1 1 1 cp	opc2 1	CRm	MCR|MRC accessing System registers
1 1 1 0	1 0 0 0	0 1 0 L	!=1111	!=1111	Rd	imm8		LDREX, STREX
1 1 1 0	1 0 0 0	1 1 0 L	!=1111	!=1111	Rt2	0 1 !=10	Rd	LDREX(B|H|D), STREX(B|H|D)
1 1 1 0	1 0 0 0	1 1 0 L	!=1111	!=1111	Rt2	1 op3	Rd	LDA{EX}{B|H|D}, STL{EX}{B|H|D}
1 1 1 0	1 0 0	!=0x10 !=xx0x L	!=1111	!=1111	!=1111	imm8		LDRD, STRD
1 1 1 1	0 i 1 0	T 1 0 0	imm4	0 imm3	!=1111	imm8		MOVW, MOVT
1 1 1 1	0 0 1 1	1 0 0 R	!=1111	1 0 0 0	M1	0 0 1 M	0 0 0 0	MSR <spec_reg><mode>,<Rn>
1 1 1 1	0 0 1 1	1 0 0 1	!=1111	1 0 0 0	1 1 1 1	0 0 0 0	0 0 0 0	MSR SPSR, <Rn>
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	NOP.W
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 1 0 L	SEV.W, SEVL.W
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 1	0 op 0 0	ESB, CSDB
1 1 1 1	0 0 1 1	1 0 1 1	1 1 1 1	1 0 0 0	1 1 1 1	0 0 1 0	1 1 1 1	CLREX
1 1 1 1	0 0 1 1	1 0 1 1	1 1 1 1	1 0 0 0	1 1 1 1	0 1 op	option	DSB, DMB, ISB, SSBB, PSSBB, SB
1 1 1 1	0 0 1 1	1 1 1 R	M1	1 0 0 0	!=1111	0 0 1 M	0 0 0 0	MRS <Rd>,<spec_reg><mode>
1 1 1 1	0 0 1 1	1 1 1 1	1 1 1 1	1 0 0 0	!=1111	0 0 0 0	0 0 0 0	MRS <Rd>,SPSR
1 1 1 1	1 0 0 0	1 !=11 0	!=1111	!=1111	imm12			STR{B|H}.W (12-bit immediate)
1 1 1 1	1 0 0 0	0 !=11 0	!=1111	!=1111	1 !=000	imm8		STR{B|H|}{T} (8-bit immediate)
1 1 1 1	1 0 0 S	1 !=11 1	!=1111	!=1111	imm12			LDR{SB|SH|B|H}.W (12-bit immediate)
1 1 1 1	1 0 0 S	0 !=11 1	!=1111	!=1111	1 !=000	imm8		LDR{SB|SH|B|H}{T} (8-bit immediate)

Figure H2-4 T32 instructions that are unchanged in Debug state

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.4 Security in Debug state

If EL3 is implemented or the implemented Security state is Secure state, security in Debug state is governed by EDSCR.SDD.

On entry to Debug state

If FEAT_RME is implemented, EDSCR.SDD is the EL3 debug disabled flag. If entering Debug state from EL3, EDSCR.SDD is set to 0. Otherwise:

- If `ExternalRootInvasiveDebugEnabled()` == TRUE, EDSCR.SDD is set to 0.
- If `ExternalRootInvasiveDebugEnabled()` == FALSE, EDSCR.SDD is set to 1.

If FEAT_RME is not implemented, EDSCR.SDD is the Secure debug disabled flag. If entering Debug state from Secure state, EDSCR.SDD is set to 0. Otherwise:

- If `ExternalSecureInvasiveDebugEnabled()` == TRUE, EDSCR.SDD is set to 0.
- If `ExternalSecureInvasiveDebugEnabled()` == FALSE, EDSCR.SDD is set to 1.

Note

If halting is prohibited in a Security state, it is not possible to enter Debug state from that Security state. However, because changes to the authentication signals require a Context Synchronization event to guarantee their effect, there is a period during which the PE might halt even though the authentication signals prohibit halting.

In Debug state

The value of EDSCR.SDD does not change, even if `ExternalRootInvasiveDebugEnabled()` or `ExternalSecureInvasiveDebugEnabled()` changes.

Note

- DBGAUTHSTATUS_EL1 is not frozen in Debug state. However, a Context Synchronization event is required to guarantee that changes are visible in DBGAUTHSTATUS_EL1.
- If EDSCR.SDD is 1 in Debug state, there is no means to enter a Security state where halting was prohibited on entry to Debug state. This means that it is not possible for the PE to be in such a Security state when EDSCR.SDD is 1. This is a general principle of behavior in Debug state.

In Non-debug state

If FEAT_RME is implemented, then EDSCR.SDD returns the inverse of `ExternalRootInvasiveDebugEnabled()`.

If FEAT_RME is not implemented, then EDSCR.SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control ExternalRootInvasiveDebugEnabled() or `ExternalSecureInvasiveDebugEnabled()` change, a Context Synchronization event is required to guarantee their effect.

Note

- In Non-debug state, EDSCR.SDD is unaffected by the Security state of the PE.
- A Context Synchronization event is also required to guarantee that changes in the authentication signals are visible in DBGAUTHSTATUS_EL1.

If EL3 is not implemented and the implemented Security state is Non-Secure state, EDSCR.SDD is RES1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.5 Privilege in Debug state

The only additional privileges offered to Debug state are:

- The privilege to execute Debug state operations, DCPS, DRPS, MRS, MSR.
- The privilege to execute DTR access instructions regardless of the Exception level and traps.

The DTR access instructions can be executed at any Exception level, including EL0, regardless of any control register settings that might force these instructions to be UNDEFINED or trapped in Non-debug state. These instruction are:

- The `MRS` and `MSR` instructions that access DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 in AArch64 state.
- The `MRC` and `MCR` instructions that access DBGDTRTXint and DBGDTRRXint in AArch32 state.

All other instructions operate with the privilege determined by the current Exception level and Security state. This applies to all Special-purpose and System registers accesses, memory accesses, and UNDEFINED instructions, and includes generating exceptions when the System registers trap or disable an instruction.

H2.4.6 Debug state operations, DCPS, DRPS, MRS, MSR

The architecture defines operations to change between Exception levels in Debug state. These operations can also change the mode at the current Exception level.

H2.4.6.1 DCPS<n>

Executing a `DCPS<n>` instruction in Debug state moves the PE to a higher Exception level or to a specific mode at the current Exception level.

If the `DCPS<n>` instruction is executed in AArch32 state and the target Exception level is using AArch64:

- The current instruction set switches from T32 to A64.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for system calls in Non-debug state. See Execution state.

Otherwise, the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- In AArch64 state the current endianness is determined by the value of SCTLR_ELx.EE for the target Exception level.
- In AArch32 state the current endianness is determined by the value of SCTLR.EE or HSCTLR.EE for the target Exception level.

The `DCPS<n>` instructions are:

In AArch64 state

- `DCPS1`
- `DCPS2`
- `DCPS3`

In AArch32 state, in the T32 instruction set only

- `DCPS1`
- `DCPS2`
- `DCPS3`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The DCPS instructions are UNDEFINED in Non-debug state.

Table H2-2 shows the target of the instruction. In Table H2-2, the entries have the following meaning:

EL1h/Svc This means that the target is:

- EL1h if EL1 is using AArch64.
- EL1 and Supervisor mode if EL1 is using AArch32.

EL2h/Hyp This means that the target is:

- EL2h if EL2 is using AArch64.
- EL2 and Hyp mode if EL2 is using AArch32.

EL3h/Monitor This means that the target is:

- EL3h if EL3 is using AArch64.
- EL3 and Monitor mode if EL3 is using AArch32.

Svc Secure Supervisor mode, in EL3 using AArch32.

Monitor Secure Monitor mode, in EL3 using AArch32.

Table H2-2 Target for DCPS instructions in Debug state

Instruction	Target when DCPS instruction executed at stated Exception level:				
	EL0	EL1	EL2	EL3 (AArch64)	EL3 (AArch32)
DCPS1	EL1h/Svc	EL1h/Svc	EL2h/Hyp	EL3h	Svc, clears SCR.NS to 0
DCPS2	EL2h/Hyp	EL2h/Hyp	EL2h/Hyp	EL3h	UNDEFINED
DCPS3	EL3h/Monitor	EL3h/Monitor	EL3h/Monitor	EL3h	Monitor, clears SCR.NS to 0

In AArch32 Monitor mode, DCPS1 and DCPS3 clear SCR.NS to 0.

Note

In AArch64 state, at EL3, DCPS<n> does not change SCR_EL3.NS.

However:

- DCPS1 is UNDEFINED at EL0 if either:
  - EL2 is implemented and enabled in the current Security state, and is using AArch64 and HCR_EL2.TGE == 1.
  - In Non-secure state, EL2 is implemented and using AArch32 and HCR.TGE == 1.
- DCPS2 is UNDEFINED at all Exception levels if EL2 is not implemented.
- DCPS2 is UNDEFINED at the following Exception levels if EL2 is implemented:
  - At EL0 and EL1 in Secure state if EL2 is disabled in the current Security state.
  - At EL3 if EL3 is using AArch32.
- DCPS3 is UNDEFINED at all Exception levels if either:
  - EDSCR.SDD == 1.
  - EL3 is not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The references to DCPS1, DCPS2, and DCPS3 in this section link to the descriptions of the instructions in the A64 instruction set.
The DCPS<n> instructions are also defined in the T32 instruction set, see DCPS1, DCPS2, DCPS3. These instructions are not defined in the A32 instruction set, because A32 instructions cannot be executed in Debug state.

On executing a DCPS instruction:

- If the target Exception level is using AArch64:
  - ELR_ELx of the target Exception level becomes UNKNOWN.
  - SPSR_ELx of the target Exception level becomes UNKNOWN.
  - ESR_ELx of the target Exception level becomes UNKNOWN.
  - DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- If the target Exception level is using AArch32, DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, become UNKNOWN and:
  - If the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.
  - If the target Exception level is EL2, then ELR_hyp, SPSR_hyp, and HSR become UNKNOWN.

If the target Exception level is using AArch32, and the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.

The DCPSInstruction() function is described in A-profile Architecture Pseudocode.

H2.4.6.2 DRPS

Executing the DRPS operation in Debug state moves the PE to a lower Exception level, or the current Exception level, by copying the current SPSR to PSTATE.

If DRPS is executed in AArch64 state and the target Exception level is using AArch32:

- The current instruction set switches from A64 to T32.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for exception returns in Non-debug state. See Execution state.

Otherwise, the instruction set state does not change.

If the target Exception level is the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- If targeting an Exception level using AArch64, current endianness is set according to SCTLR_ELx.EE, or SCTLR_EL1.E0E for the target Exception level.
- If targeting an Exception level using AArch32, current endianness is set by SPSR.E as appropriate.

The DRPS instructions are:

In AArch64 state

- DRPS

In AArch32 state, in the T32 instruction set only

- ERET

If the SPSR specifies an illegal exception return, then PSTATE.{M, nRW, EL, SP} are unchanged, PSTATE.UINJ is set to 0, and PSTATE.IL is set to 1. For further information on illegal exception returns, see Illegal exception returns from AArch64 state.

Some PSTATE fields are ignored in Debug state. The effect of the DRPS operation on them is to set them to an UNKNOWN value that might be the value from the SPSR. For more information, see PSTATE in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other PSTATE fields are copied from SPSR.

DRPS is UNDEFINED at EL0 and in Non-debug state.

Note

Unlike an exception return, the DRPS operation has no architecturally-defined effect on the Event Register and Exclusives monitors. DRPS might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

On executing a DRPS instruction:

- If the target Exception level is using AArch64, then DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- If the target Exception level is using AArch32, then DLR, DSPSR, and if FEAT_Debugv8p9 is implemented, DSPSR2, become UNKNOWN.
- If FEAT_PAuth_LR is implemented, then:
  - For a trivial implementation of PSTATE.PACM, PSTATE.PACM is set to 0.
  - If the effects of PACM instructions are disabled at the target Exception level indicated in SPSR_ELx, PSTATE.PACM is set to 0.
  - Otherwise, PSTATE.PACM is copied from SPSR_ELx.PACM.

The DRPSInstruction() function is described in A-profile Architecture Pseudocode.

H2.4.6.3 MRS and MSR

The other Debug state instructions are used to read or write DLR_EL0 and DSPSR_EL0.

These instructions are:

In AArch64 state

- MRS
- MSR (register)

```
MRS <Xt>, DLR_EL0          ; Copy DLR_EL0 to <Xt>
```

```
MRS <Xt>, DSPSR_EL0    ; Copy DSPSR_EL0 to <Xt>
```

```
MSR DLR_EL0, <Xt>      ; Copy <Xt> to DLR_EL0
```

```
MSR DSPSR_EL0, <Xt>    ; Copy <Xt> to DSPSR_EL0
```

In AArch32 state

- MRC
- MCR

```
MRC <Rn>, DLR                          ; Copy DLR to <Rn>
```

```
MRC <Rn>, DSPSR                    ; Copy DSPSR to <Rn>
```

```
MCR DLR, <Rn>                      ; Copy <Rn> to DLR
```

```
MCR DSPSR, <Rn>                        ; Copy <Rn> to DSPSR
```

If FEAT_Debugv8p9 is implemented:

```
MRC <Rn>, DSPSR2                   ; Copy DSPSR2 to <Rn>
```

```
MCR DSPSR2, <Rn>                       ; Copy <Rn> to DSPSR2
```

These instructions can be executed at any Exception level when in Debug state, including EL0. They are UNDEFINED in Non-debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.7 Exceptions in Debug state

The following sections describe how exceptions are handled in Debug state:

- Generating exceptions when in Debug state.
- Taking exceptions when in Debug state.
- Reset in Debug state.

H2.4.7.1 Generating exceptions when in Debug state

In Debug state:

- Instruction Abort exceptions cannot happen because instructions are not fetched from memory.
- Interrupts, including SError and virtual interrupts are ignored and remain pending:
  — The pending interrupt remains visible in ISR.
- Debug exceptions and debug events are ignored.
- SCR.EA is treated as 0, regardless of its actual state, other than for the purpose of a direct read.
- Any attempt to execute an instruction bit pattern that is an allocated instruction at the current Exception level, but is listed in Executing instructions in Debug state as UNDEFINED in Debug state, generates an exception.

  Note

  If the exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

  The priority and syndrome for these exceptions is the same as for executing an encoding that does not have an allocated instruction.
- Instructions executed at EL2, EL1 and EL0 that are configured by EL3 control registers to trap to EL3:
  — When the value of EDSCR.SDD is 0, generate the appropriate trap exception that is taken to EL3.
  — When the value of EDSCR.SDD is 1, are treated as UNDEFINED and generate an exception.
  If the exception is taken to an Exception level using AArch64 or to AArch32 Hyp mode, then it is reported with an EC syndrome value of `0x00`.
- If FEAT_RME is implemented and EDSCR.SDD is 1, SCR_EL3.GPF is treated as 0, regardless of its actual state, other than for the purpose of a direct read.

Otherwise, synchronous exceptions are generated as they would be in Non-debug state and taken to the appropriate Exception level in Debug state.

Note

If FEAT_RME is implemented and EDSCR.SDD == 1, then exceptions from Non-secure, Secure, and Realm state are never taken to Root state. If FEAT_RME is not implemented and EDSCR.SDD == 1, then an exception from Non-secure state is never taken to Secure state. See Security in Debug state.

H2.4.7.2 Taking exceptions when in Debug state

When the PE is in Debug state, all exceptions are synchronous. When an exception is generated, it is taken to Debug state. This means that:

- The target Exception level is as defined for the exception in Non-debug state.
- If the target Exception level is using AArch32 then the target PE mode is as defined for the exception in Non-debug state.
- The exception syndrome is reported as defined for the exception in Non-debug state, except for the case described in Data Aborts in Memory access mode for which the reporting requirements are relaxed.
  The exception syndrome is reported using the syndrome register or registers for the target Exception level. In AArch64 state, these are ESR_ELx, and FAR_ELx, and if FEAT_RME is implemented, MFAR_EL3. In AArch32 state, these are DFSR, DFAR, HSR, HDFAR, and HPFAR. For example:
  — If a Data Abort exception is taken to Abort mode at EL1 or EL3 and the exception is taken from AArch32 state and using the Short-descriptor translation table format, the DFSR reports the exception using the Short-descriptor format fault encoding. For exceptions other than Data Abort exceptions taken to Abort mode, DFSR is not updated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If an instruction is trapped to an Exception level using AArch64 due to a configurable trap, disable, or enable, the exception code reported is the same as it would be in Non-debug state.

The effect on auxiliary syndrome registers, such as AFSR, is IMPLEMENTATION DEFINED.

Note

Generally, the AArch32 Fault Address Registers (FARs) and Fault Status Registers (FSRs) are not described as *syndrome registers*, although the term is appropriate to their function.

- The PE remains in Debug state and changes to the target mode.
- If EL3 is using AArch32 and the exception is taken from Monitor mode, SCR.NS is cleared to 0.
- If the exception is taken to an Exception level using AArch32, the PE continues to execute T32 instructions, regardless of the TE bit in the System register for the target Exception level.
- The endianness switches to that indicated by the EE bit of the System register for the target Exception level.
- The SPSR for the target Exception level or mode is corrupted and becomes UNKNOWN.
- ELR_ELx for the target Exception level becomes UNKNOWN.
- If the target Exception level is EL2 using AArch32, ELR_hyp becomes UNKNOWN.
- If the target Exception level is EL1 or EL3 using AArch32, LR_<mode> for the target mode becomes UNKNOWN.
- DLR and DSPSR become UNKNOWN.
- The cumulative error flag, EDSCR.ERR, is set to 1. See Cumulative error flag.
- PSTATE.IL is cleared to 0.
- If FEAT_UINJ is implemented, PSTATE.UINJ is cleared to 0.
- PSTATE.{IT, T, SS, D, A, I, F, ALLINT} are set to UNKNOWN values, and PSTATE.{N, Z, C, V, Q, GE} are unchanged. However, these fields are ignored and are not observable in Debug state. For more information, see PSTATE in Debug state.

The debugger must save any state that can be corrupted by an exception before executing an instruction that might generate another exception.

H2.4.7.3 Pseudocode description of taking exceptions in Debug state

`AArch64_TakeException()` shows the behavior when the PE takes an exception to an Exception level using AArch64 in Non-debug state. In Debug state, this redirects to `AArch64_TakeExceptionInDebugState()`.

`AArch32_EnterMode()`, `AArch32_EnterHypMode()`, and `AArch32_EnterMonitorMode()` show the behavior when the PE takes an exception to an Exception level using AArch32 in Non-debug state. In Debug state:

- `AArch32_EnterMode()` redirects to `AArch32_EnterModeInDebugState()`.
- `AArch32_EnterHypMode()` redirects to `AArch32_EnterHypModeInDebugState()`.
- `AArch32_EnterMonitorMode()` redirects to `AArch32_EnterMonitorModeInDebugState()`.

H2.4.7.4 Reset in Debug state

If the PE is reset when in Debug state, it exits Debug state and enters Non-debug reset state. When the PE is in reset state, EDSCR.STATUS == `0b000010` and writes to EDITR are ignored.

Note

If EDECR.RCE == 1 or CTIDEVCTL.RCE ==1, meaning that a Reset Catch debug event is programmed, and if halting is allowed on exiting reset state, then on exiting reset state the PE halts and reenters Debug state. See Reset Catch debug events. All PE registers have taken their reset values, which might be UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.8 Accessing registers in Debug state

Register accesses are unchanged in Debug state. The view of each register is determined by either the current Exception level or the mode, or both, and accesses might be disabled or trapped by controls at a higher Exception level.

H2.4.8.1 General-purpose register access, other than AArch64 state SP access

A single general-purpose register can be read by issuing an `MSR` instruction through the ITR to write DBGDTR_EL0 in AArch64 state, or an `MCR` instruction through the ITR to write DBGDTRTXint in AArch32 state. The debugger can then read the DTR register or registers through the external debug interface. The reverse sequence writes to a general-purpose register.

Figure H2-5 shows the reading and writing of general-purpose registers, other than SP, in Debug state in AArch64 state.

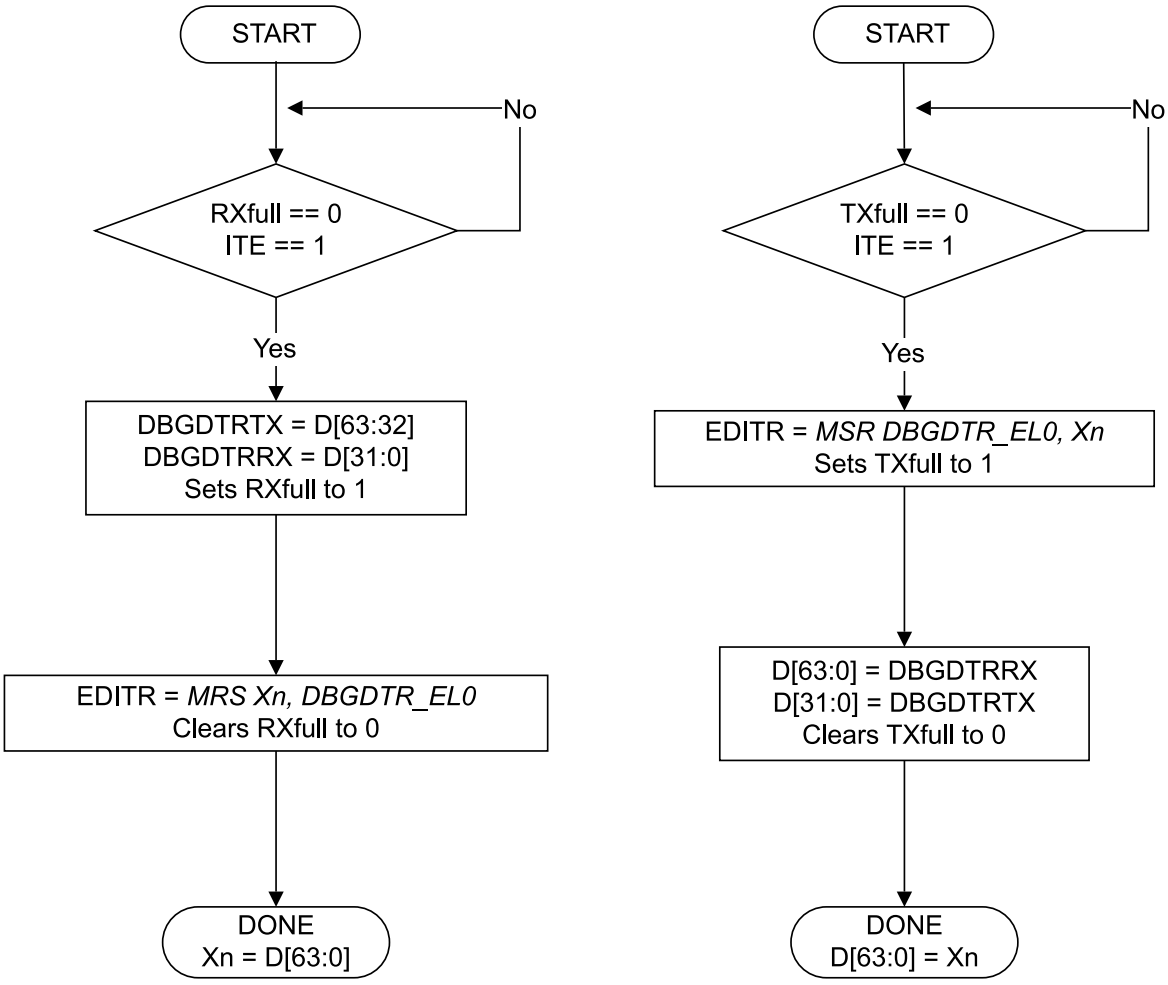

Figure H2-5 Reading and writing general-purpose registers, other than SP, in Debug state in AArch64 state

Figure H2-6 shows the reading and writing of general-purpose registers in Debug state in AArch32 state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

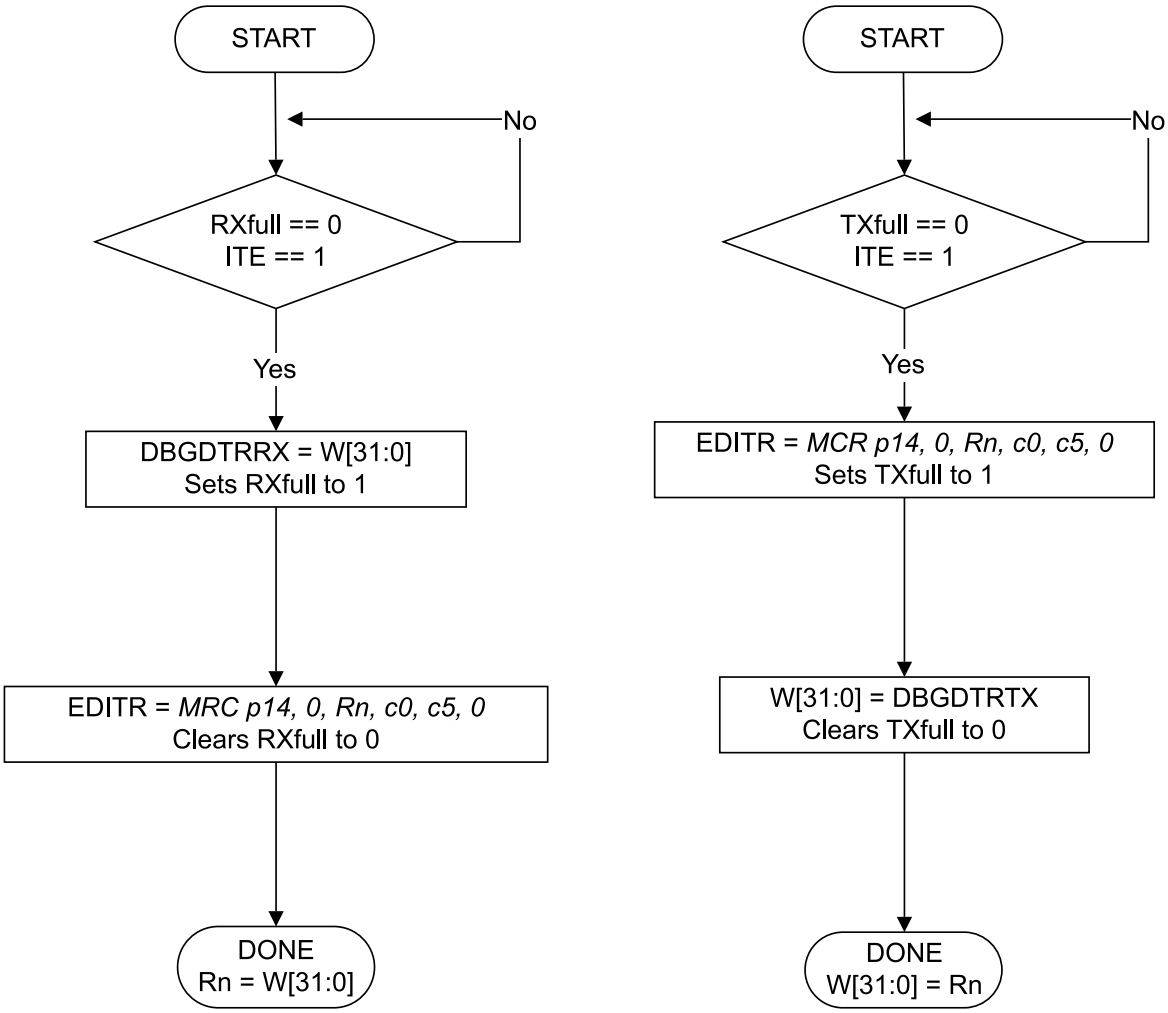

Figure H2-6 Reading and writing general-purpose registers in Debug state in AArch32 state

H2.4.8.2 PC and PSTATE access

The debugger reads the Program Counter and PSTATE of the process being debugged through the DLR_EL0 and DSPSR_EL0 System registers. The actual values of PC and PSTATE cannot be directly observed in Debug state:

- Instructions that are used for direct reads and writes of PC and PSTATE in Non-debug state are CONSTRAINED UNPREDICTABLE in Debug state.
- On taking an exception, ELR_ELx and SPSR_ELx at the target Exception level are UNKNOWN. They do not record the PC and PSTATE.

PSTATE.{TCO, UAO, PAN, IL, E, M, nRW, EL, SP} are indirectly read by instructions executed in Debug state, but all other PSTATE fields are ignored and cannot be observed. See also:

- PSTATE in Debug state.
- Executing instructions in Debug state.
- Exceptions in Debug state.

H2.4.8.3 Other register accesses

To access other registers, the debugger can use the operations described in General-purpose register access, other than AArch64 state SP access and the operations described below to build a sequence of operations to move the registers to or from a general-purpose register.

To access a SIMD&FP, SP, or System register, the debugger can use an instruction from the following list to transfer the register to or from a general-purpose register:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The `UMOV` instruction in AArch64 state or the `VMOV` instruction in AArch32 state for SIMD transfers from SIMD&FP registers.
- The `FMOV` instruction in AArch64 state or the `VMOV` instruction in AArch32 state for floating-point transfers to or from SIMD&FP registers.
- `MOV` (to/from SP) `SP,Xs` and `MOV` (to/from SP) `Xd,SP` instructions in AArch64 state for transfers to or from the SP register.
- `MSR`, `MSRR`, `MRRS`, and `MRS` instructions in AArch64 state or `MCR`, `MCRR`, `MRRC`, and `MRC` instructions in AArch32 state for transfers to or from System registers.

To access an SVE scalable vector register, the debugger can use the following instruction sequences to transfer the register to or from a general-purpose register, 64 bits at a time:

- To read, the following sequence to copy the 64-bit element 0 of `Vs` to general-purpose register `Xd` and rotate `Zs` register right by 64 bits:

```
UMOV Xd, Vs.D[0]
EXT Zs.B, Zs.B, Zs.B, #8
```

- To write, the `INSR ZD.D,Xs` instruction to shift vector register `Zd` left by one 64-bit element and copy general-purpose register `Xs` to the 64-bit element 0.

To access an SVE predicate register, the debugger can use a sequence from the following list to transfer the register to or from an SVE scalable vector register:

- If FEAT_SVE2p1 is implemented or FEAT_SME2p1 is implemented, the `PMOV Zd,Ps.B` and `PMOV Pd.B,Zs` instructions.
- If FEAT_SVE2p1 is not implemented and FEAT_SME2p1 is not implemented:
  - To read, the `CPY Zd.B,Ps/Z,#1` instruction to expand predicate elements in `PS` into zero and one 8-bit element values in `Zd`.
  - To write, the following sequence to convert zero and non-zero 8-bit elements in `Zs` to a corresponding 1-bit predicate elements in `Pd`:

```
PTRUE Pd.B
CMPNE Pd.B, Pd/Z, Zs.B, #0
```

To access the SVE First Fault Register, FFR, the debugger can use the `RDFFR` and `WRFFR` instructions to transfer the register to or from an SVE predicate register.

To access the SME ZA storage, the debugger can use the `MOVA` (tile to vector, single) and `MOVA` (vector to tile, single) instructions to transfer to or from an SVE scalable vector register.

To access the SME ZT0 register, the debugger can use the `MOVT` (table to scalar) and `MOVT` (scalar to table) instructions to transfer to or from a general-purpose register, 64 bits at a time.

H2.4.9 Accessing memory in Debug state

How the PE accesses memory is unchanged in Debug state. This includes:

- The operation of the MMU, including address translation, tagged address handling, access permissions, memory attribute determination, and the operation of any TLBs.
- The operation of any caches and coherency mechanisms.
- Alignment support.
- Endianness support.
- The Memory order model.
- The operation of the MPU, including access permissions and memory attribute determination.

H2.4.9.1 Simple memory transfers

Simple memory accesses can be performed in Debug state by issuing memory access instructions through the ITR and passing data through the DTR registers. Executing instructions in Debug state lists the memory access instructions that are supported in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.4.9.2 Bulk memory transfers

Memory access mode can accelerate bulk memory transfers in Debug state. See DCC and ITR access modes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.5 Exiting Debug state

The PE exits Debug state when it receives a Restart request trigger event. If EDSCR.ITE == 0, when the PE receives a Restart Request trigger event, the behavior of any instruction issued through the ITR in Normal access mode or an operation issued by a DTR access in Memory access mode that has not completed execution is one of the following:

- The instruction completes execution in Debug state and the PE ignores the restart request and remains in Debug state.
- The instruction completes execution in Debug state before the PE executes the restart sequence.
- The instruction completes execution in Non-debug state after the PE executes the restart sequence.
- The instruction is abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

Note

- Implementations can set EDSCR.ITE to 1 to indicate that further instructions can be accepted by ITR before the previous instructions have completed. If any previous instruction has not completed and EDSCR.ITE == 1, then the PE must complete these instructions in Debug state before executing the restart sequence. EDSCR.ITE == 0 indicates that the PE is not ready to restart.
- A debugger must observe that any instructions issued through EDITR that might generate a synchronous exception, as complete, before issuing a restart request. It can do this by observing the completion of a later instruction, as synchronous exceptions must occur in program order. For example, a debugger can observe that an instruction that reads or writes a DTR register is complete because of its effect on the EDSCR.{TXfull, RXfull} flags.

On exiting Debug state, the PE sets the Program Counter to the address in DLR, where:

- If exiting to AArch32 state:
  - Bits[31:1] of the PC are set to the value of bits[31:1] of DLR.
  - Bit[0] of the PC is set to a CONSTRAINED UNPREDICTABLE choice of 0 or the value of bit[0] in DLR.
- If exiting to AArch64 state:
  - Bits[63:56] of DLR_EL0 might be ignored as part of tagged address handling. See Address tagging.
  - Otherwise the PC is set from DLR_EL0.

Note

Bits[63:32] of DLR_EL0 are ignored when exiting to AArch32 state.

Exit from Debug state can give rise to a PC alignment fault exception when the Program Counter is used. Unlike an exception return, this might also happen when returning to AArch32 state. For more information, see PC alignment checking.

On exiting Debug state, PSTATE is set from DSPSR in the same way that an exception return sets PSTATE from SPSR_ELx:

- The same illegal exception return checks that apply to an exception return also apply to exiting Debug state. If the return from Debug state is an illegal exception return then the effect on PSTATE and the PC is the same as for any other illegal exception return. See Exception return and Exception return to an Exception level using AArch32.
- The checks on the PSTATE.IT bits that apply to exiting Debug state into AArch32 state are the same as those that apply to an exception return. See Architectural Constraints on UNPREDICTABLE Behaviors.
- PSTATE.SS is copied from DSPSR.SS if all of the following hold:
  - MDSCR_EL1.SS == 1.
  - The debug target Exception level is using AArch64.
  - Software step exceptions from the restart Exception level are enabled.

  Otherwise PSTATE.SS is set to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Unlike a return using ERET, PSTATE.SS must be restored from DSPSR.SS because otherwise it is UNKNOWN.

However, if OSDLR.DLK == 1 and DBGPRCR.CORENPDRQ == 0, meaning FEAT_DoubleLock is implemented and locked in Non-debug state and therefore Software Step exceptions are disabled, but otherwise Software Step exceptions would be enabled from the restart Exception level, it is CONSTRAINED UNPREDICTABLE whether PSTATE.SS is copied from DSPSR.SS.

- If FEAT_SSBS is implemented, DSPSR.SSBS is copied to PSTATE.SSBS.
- If FEAT_PAN is implemented, DSPSR.PAN is copied to PSTATE.PAN.
- If FEAT_UAO is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.UAO is copied to PSTATE.UAO. On exit from Debug state to AArch32 state, PSTATE.UAO is not updated.
- If FEAT_DIT is implemented, on exit from Debug state, DSPSR.DIT is copied to PSTATE.DIT.
- If FEAT_UINJ is implemented, DSPSR.UINJ is copied to PSTATE.UINJ.
- If FEAT_MTE is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.TCO is copied to PSTATE.TCO. On exit from Debug state to AArch32 state, PSTATE.TCO is not updated.
- If FEAT_BTI is implemented, DSPSR_EL0.BTYPE is copied to PSTATE.BTYPE.
- If FEAT_NMI is implemented, DSPSR_EL0.ALLINT is copied to PSTATE.ALLINT.
- If FEAT_GCS is implemented, DSPSR_EL0.EXLOCK is copied to PSTATE.EXLOCK.
- If FEAT_EBEP, FEAT_SPE_EXC or FEAT_TRBE_EXC is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.PM is copied to PSTATE.PM. On exit from Debug state to AArch32 state, PSTATE.PM is not updated.
- If FEAT_PAuth_LR is implemented, all of the following apply:
  - For a trivial implementation of PSTATE.PACM, PSTATE.PACM is set to 0.
  - If the effects of PACM instructions are disabled at the target Exception level, PSTATE.PACM is set to 0.
  - Otherwise, DSPSR_EL0.PACM is copied to PSTATE.PACM.

Note

- One important difference between Debug state exit and an exception return is that the PE can exit Debug state at EL0. Despite this, the behavior of an exit from Debug state is similar to an exception return. For example, PSTATE.{D, A, I, F} is updated regardless of the value of SCTLR_EL1.UMA.
- Exit from Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors. An exit from Debug state might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

The `ExitDebugState()` function is described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H3
Halting Debug Events

This chapter describes a particular class of debug events. It contains the following sections:

- Introduction to Halting debug events.
- Halting Step debug events.
- Halt Instruction debug event.
- Exception Catch debug event.
- External Debug Request debug event.
- OS Unlock Catch debug event.
- Reset Catch debug events.
- Software Access debug event.
- Synchronization and Halting debug events.

Note

Table K14-1 disambiguates the general register references used in this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.1 Introduction to Halting debug events

External debug defines Halting debug events. The following Halting debug events are available from the introduction of Armv8:

- Halting Step debug events.
- Halt Instruction debug event.
- Exception Catch debug event.
- External Debug Request debug event.
- OS Unlock Catch debug event.
- Reset Catch debug events.
- Software Access debug event.

If halting is allowed, a Halting debug event halts the PE. The PE enters Debug state.

In addition, breakpoints and watchpoints might halt the PE if halting is allowed. See Breakpoint and Watchpoint debug events. Because breakpoints and watchpoints can generate an exception or halt the PE, Breakpoint and Watchpoint debug events are not classified as Halting debug events.

For a definition of Debug state, see Debug State. For a definition of halting allowed, see Halting allowed and halting prohibited.

Debug state entry and debug event prioritization describes the behavior when multiple debug events are generated by an instruction.

See also Synchronization and Halting debug events.

Table H3-1 shows the behavior of Breakpoint, Watchpoint, and Halting debug events.

Table H3-1 Summary of debug events and possible outcomes

Debug event type	PE behavior when halting is:	
	Allowed	Prohibited
Breakpoint and Watchpoint debug events	Halt	See Table D2-1 and Table G2-1
Halt Instruction debug event	Halt	UNDEFINED
Software Access debug event	Halt	Ignored
Exception Catch debug event	Halt	Ignored or pended
Halting Step debug events	Halt	Pended
External Debug Request debug event	Halt	Pended
Reset Catch debug events	Halt	Pended
OS Unlock Catch debug event	Pended	Pended

Table H3-2 shows where the pseudocode for each Halting debug event type is located.

Table H3-2 Pseudocode description of Halting debug events

Halting debug event type	Pseudocode
Halt Instruction debug event	HLT for AArch64 and HLT for AArch32

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Halting debug event type	Pseudocode
Software Access debug event	Pseudocode description of Software Access debug event
Exception Catch debug event	Pseudocode description of Exception Catch debug events
Halting Step debug events	Pseudocode description of Halting Step debug events
External Debug Request debug event	Pseudocode description of External Debug Request debug events
Reset Catch debug events	Pseudocode description of Reset Catch debug event
OS Unlock Catch debug event	Pseudocode description of OS Unlock Catch debug event

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.2 Halting Step debug events

Halting Step is a debug resource that a debugger can use to make the PE step through code one instruction at a time. This section describes the Halting Step debug events. It is divided into the following sections:

- Overview of a Halting Step debug event.
- The Halting Step state machine.
- Using Halting Step.
- Detailed Halting Step state machine behavior.
- Synchronization and the Halting Step state machine.
- Stepping T32 IT instructions.
- Disabling interrupts while stepping.
- Syndrome information on Halting Step.
- Pseudocode description of Halting Step debug events.

The architecture describes the behavior as a simple Halting Step state machine. See The Halting Step state machine.

H3.2.1 Overview of a Halting Step debug event

The behavior of Halting Step is defined by a state machine, shown in Figure H3-1. A Halting Step debug event executes a single instruction and then returns control to the debugger. When the debugger software wants to execute a Halting Step:

1. With the PE in Debug state, the debugger activates Halting Step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

However, an exception might be generated while the instruction is being stepped. That is either:

- A synchronous exception generated by the instruction being stepped.
- An asynchronous exception taken before or after the instruction being stepped.

Halting Step has its own enable control bit, EDECR.SS and EDESR.SS.

Note

Because the Halting Step state machine states occur as a result of normal PE operation, the states can be described as both:

- PE states.
- Halting Step states.

H3.2.2 The Halting Step state machine

The state machine states are:

Inactive Halting Step is inactive. No Halting Step debug events can be generated, therefore execution is not affected by Halting Step. The PE is in this state whenever either of the following is true:

- Halting Step is disabled. That is, EDECR.SS is set to 0 and EDESR.SS is set to 0.
- Halting is prohibited. See Halting the PE on debug events. In this state, if EDECR.SS is set to 1, then a Halting Step debug event is pending.

In Figure H3-1, this state is shown in red.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Active-not-pending

Halting Step is enabled and active. This is the state in which the PE steps an instruction. EDECR.SS == 1 and EDESR.SS == 0. Software must not set EDECR.SS to 1 unless the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE, as described in Changing the value of EDECR.SS when not in Debug state.

In Figure H3-1, this state is shown in green.

Active-pending

Halting Step is enabled and active. The step has completed and the PE enters Debug state. EDESR.SS == 1.

In Figure H3-1, this state is shown in green.

Whenever Halting Step is enabled and active, whether the state machine is in the active-not-pending state or in the active-pending state depends on EDESR.SS. Table H3-3 shows this.

In the simple sequential execution of the program, the PE executes the Halting Step state machine as follows:

1. Initially, Halting Step is inactive.
2. After exiting Debug state, Halting Step is active-not-pending.
3. The PE executes an instruction and Halting Step is active-pending.
4. The pending Debug state entry is taken on the next instruction and the step is complete.

Exceptions and other changes to the PE context can interrupt this sequence.

Figure H3-1 shows a Halting Step state machine.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

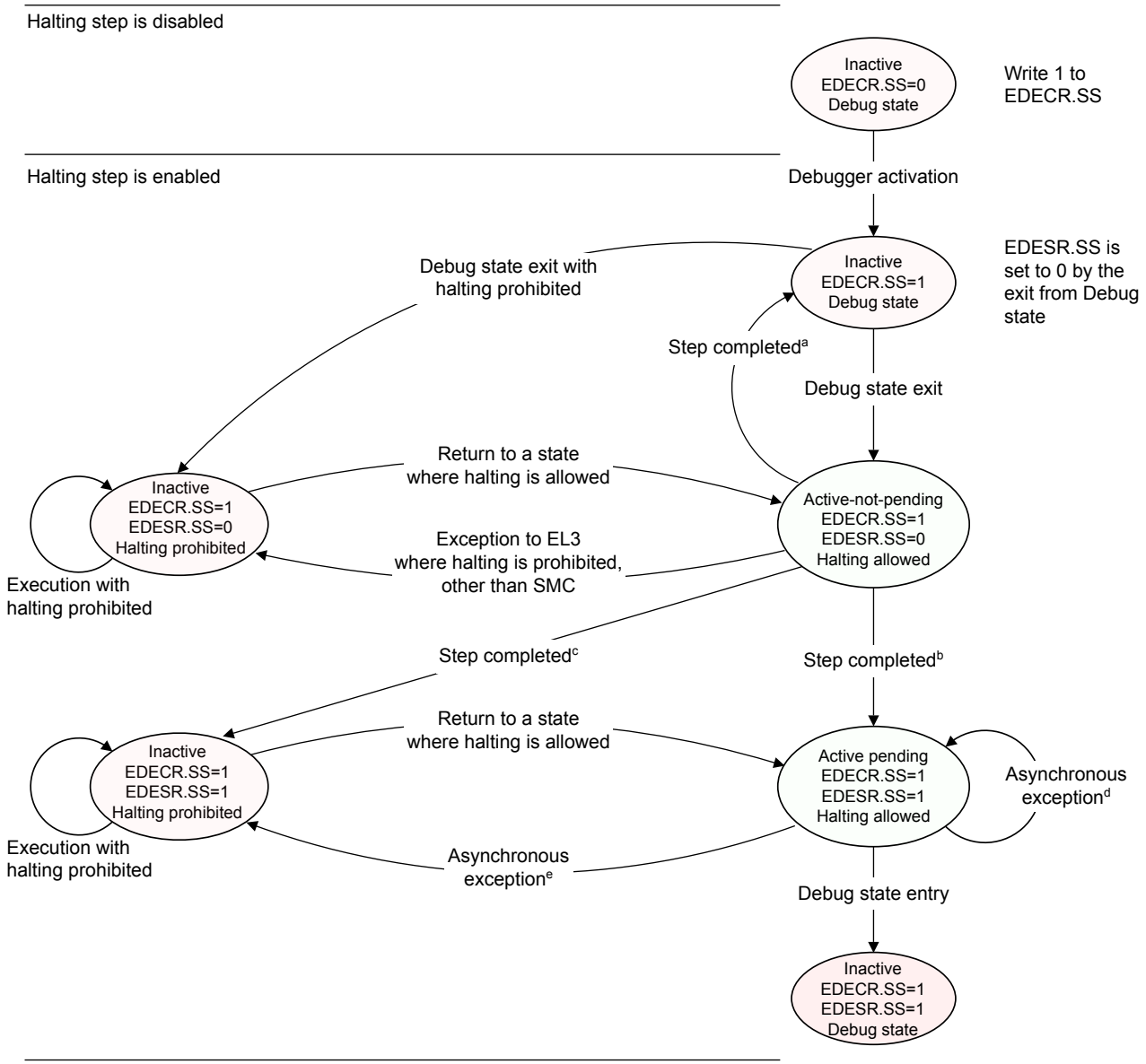

a. Step completed occurs when:
· A debug event, other than a Halting Step debug event, causes entry into Debug state.

b. Step completed occurs when:
· An instruction is executed without taking an exception.
· An exception is taken to a state where halting is allowed.
· A reset.

c. Step completed occurs when:
· An SMC exception is taken to EL3 where halting is prohibited.

d. An asynchronous exception taken to a state where halting is allowed.

e. An asynchronous exception taken to EL3 where halting is prohibited.

Figure H3-1 Halting Step state machine

Note

Figure H3-1 describes only state transitions to and from the inactive state by exit from Debug state, executing an exception return, or taking an exception. Other changes to the PE context, including writes to registers such as EDECR and OSDLR and changes to the authentication interface can also cause changes to the Halting Step state machine. These can lead to UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior. See Synchronization and the Halting Step state machine.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following bits control the state machine, as shown in Table H3-3:

- EDECR.SS. This is the Halting Step enable bit.
- EDESR.SS. This is the Halting Step debug event pending bit.

Table H3-3 shows the Halting Step state machine states. The letter X in a register column means that the relevant bit can be set to either zero or one.

Table H3-3 Halting Step state machine states

Halting	EDECR.SS	EDESR.SS	Halting Step state
Prohibited	X	X	Inactive (Halting Step debug even not pending)
Prohibited	X	1	Inactive (Halting Step debug event pending)
Allowed	0	0	Inactive
Allowed	1	0	Active-not-pending
Allowed	X	1	Active-pending

H3.2.3 Using Halting Step

To step a single instruction the PE must be in Debug state:

1. The debugger sets EDECR.SS to 1 to enable Halting step.
2. The debugger signals the PE to exit Debug state with DLR set to the address of the instruction being stepped. The PE clears EDESR.SS to 0 and the Halting Step state machine enters the active-not-pending state.
3. The PE executes the instruction being stepped.
   If an exception is taken to a state where halting is prohibited, then EDESR.SS is always correct for the preferred return address of the exception.
4. The PE enters Debug state before executing the next instruction and the step is complete.

Note

- If FEAT_DoPD is not implemented, EDECR.SS value is in the Debug power domain, meaning that the state machine is maintained over a powerdown of the Core power domain.
- If FEAT_DoPD is implemented, the values of EDECR.SS and EDESR.SS are set to 0 on a Cold reset, and, if the PE was stepping an instruction, EDESR.SS is effectively UNKNOWN after a Warm reset. A debugger must use a Reset Catch debug event to step over a powerdown state.
- A debugger must only change the value of EDECR.SS when the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE as described in Changing the value of EDECR.SS when not in Debug state.

H3.2.4 Detailed Halting Step state machine behavior

The behavior of the Halting Step state machine is described in the following sections:

- Entering the active-not-pending state.
- PE behavior in the active-not-pending state.
- Entering the active-pending state.
- PE behavior in the active-pending state.
- PE behavior in the inactive state when in Non-debug state.
- PE behavior in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.2.4.1 Entering the active-not-pending state

The PE enters the active-not-pending state:

- By exiting Debug state to a state where halting is allowed with EDECR.SS == 1.
- By an exception return from a state where halting is prohibited to a state where halting is allowed with EDECR.SS == 1 and EDESR.SS == 0.
- As described in Synchronization and the Halting Step state machine.

H3.2.4.2 PE behavior in the active-not-pending state

When the PE is in the active-not-pending state it does one of the following:

- It executes one instruction and does one of the following:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception generated by the instruction.
  - Generates a debug event that causes entry to Debug state.
- It takes an asynchronous exception without executing any instruction.
- It takes an asynchronous debug event into Debug state.

H3.2.4.2.1 If no exception or debug event is generated

If no exception or debug event is generated the PE sets EDESR.SS to 1. This means that the Halting Step state machine advances to the active-pending state.

H3.2.4.2.2 If an exception or debug event is generated

The PE sets EDESR.SS according to all of the following:

- The type of exception.
- The target Exception level of the exception.
- If the exception is taken to EL3, whether halting is prohibited at EL3. This is determined by:
  - The result of `ExternalRootInvasiveDebugEnabled()`, if FEAT_RME is implemented.
  - The result of `ExternalSecureInvasiveDebugEnabled()`, if FEAT_RME is not implemented.

If an exception or debug event is generated, the PE sets EDESR.SS to 1 if the exception is an SMC exception or is taken to a state where halting is allowed.

Otherwise, EDESR.SS is unchanged.

It is UNPREDICTABLE whether EDESR.SS is set to 1 or unchanged when an SError exception is taken to EL3 without executing the instruction, and halting is prohibited at EL3.

If halting is prohibited after taking the exception or debug event, then the Halting Step state machine advances to the inactive state. Otherwise, the Halting Step state machine advances to the active-pending state.

Note

The underlying criteria for the value of EDESR.SS on an exception are:

- Whether halting is allowed at the target of the exception. If halting is allowed, the PE must step into the exception. If halting is prohibited, the PE must step over the exception.
- Whether the preferred return address of the exception is the instruction itself or the next instruction, if the PE steps over the exception.

Table H3-4 shows the behavior of the active-not-pending state. The letter X indicates that halting can be either allowed or prohibited at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H3-4 Summary of active-not-pending state behavior

Event	Target Exception level	Halting is allowed at EL3	Value written to EDESR.SS
No exception or debug event	Not applicable	X	1
SMC exception	EL3	X	1
Reset	Highest	X	1
Exception, other than SMC exception	EL1	X	1
	EL2	X	1
	EL3	TRUE	1
		FALSE	Unchanged
Debug event	Debug state	X	Unchanged

H3.2.4.3 Entering the active-pending state

The PE enters the active-pending state by one of the following:

- From the active-not-pending state by:
  - Executing an instruction without taking an exception.
  - Taking an exception so that the PE remains in a state where halting is allowed.
- An exception return from a state where halting is prohibited when EDESR.SS == 1.

  Note

  For example, an exception return from EL3 with invasive debug prohibited, to a lower Exception level where invasive debug is allowed.

- A reset when the value of EDECR.SS == 1, regardless of the state the PE was in before the reset occurred.
- From the active-pending state by taking an asynchronous exception to a state where halting is allowed.
- Following the description in Synchronization and the Halting Step state machine.

H3.2.4.4 PE behavior in the active-pending state

When the PE is in the active-pending state, it enters Debug state before executing an instruction.

The entry into Debug state has higher priority than all other types of synchronous debug event and synchronous exception. However, the architecture does not define the prioritization of this Debug state entry with respect to any unmasked pending asynchronous exception. If an asynchronous exception is prioritized over the entry to Debug state, then EDESR.SS is unchanged.

For more information on the prioritization of debug events, see Debug state entry and debug event prioritization.

H3.2.4.5 PE behavior in the inactive state when in Non-debug state

EDESR.SS is not updated by the execution of an instruction or the taking of an exception when Halting Step is inactive. This means that EDESR.SS is not changed by an exception handled in a state where halting is prohibited.

On return to a state where halting is allowed, the Halting Step state machine is restored either to the active-pending state or the active-not-pending state, depending on the value of EDESR.SS. The return to a state where halting is allowed is normally by an exception return, which in some situations is a Context Synchronization event.

See also Synchronization and the Halting Step state machine.

H3.2.4.6 PE behavior in Debug state

Halting Step is inactive in Debug state because halting is prohibited, see Halting allowed and halting prohibited.

Entry to Debug state does not change EDESR.SS.

EDESR.SS is cleared to 0 on exiting Debug state as the result of a restart request. If EDECR.SS == 1, Halting Step enters the active-not-pending state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This means that EDESR.SS is never cleared to 0 by the execution of an instruction in Debug state, or by taking an exception when in Debug state as described in PE behavior in the active-not-pending state, because the Halting Step state machine is not in the active-not-pending state. EDESR.SS can be cleared by a write to EDESR, see the register description.

However, if the PE exits Debug state as the result of a PE reset and EDECR.SS == 1, then Halting Step immediately enters the active-pending state, as EDESR.SS is set to the value of EDECR.SS.

H3.2.5 Synchronization and the Halting Step state machine

The Halting Step state machine also changes state if:

- Halting becomes allowed or prohibited other than by exit from Debug state, an exception return, or taking an exception. This means that halting becomes allowed or prohibited because:
  - The Security state changes without an exception return. See State and mode changes without explicit Context Synchronization events.
  - The external authentication interface changes.
  - FEAT_DoubleLock is implemented and the status, `DoubleLockStatus()`, changes.
- A write to EDECR when the PE is in Non-debug state changes the value of EDECR.SS.

  Note

  Behavior is CONSTRAINED UNPREDICTABLE if the value of EDECR.SS is changed when the PE is in Non-debug state, see Changing the value of EDECR.SS when not in Debug state.

- A write to EDESR when the PE is in Non-debug state clears EDESR.SS to 0.

These operations are guaranteed to take effect only after a Context Synchronization event. If the instruction being stepped generates a Context Synchronization event, then the PE might use the old or new state.

The PE must perform the required behavior of the new state before or immediately following the next Context Synchronization event, but it is not required to do so immediately. This means that the PE can perform the required behavior of the old state before the next Context Synchronization event. This is illustrated in Example H3-1 and Example H3-2.

Example H3-1 Synchronization requirements 1

EDECR.SS is set to 1 in Debug state, requesting the active-not-pending state on exit from Debug state. On exit from Debug state the PE immediately takes an exception to EL3, where halting is prohibited. The PE does not step any instructions but executes the software at EL3 as normal. EDESR.SS remains set to 0. If halting is subsequently allowed, the PE must perform the required behavior of the active-not-pending state before or immediately following the next Context Synchronization event, but it is not required to do so immediately.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example H3-2 Synchronization requirements 2

EDECR.SS is set to 1 in Debug state. On exit from Debug the PE executes an `MSR` instruction that sets OSDLR_EL1.DLK to 1 and `DoubleLockStatus()` becomes TRUE. This change requires a Context Synchronization event to guarantee its effect, meaning it is CONSTRAINED UNPREDICTABLE whether:

- Halting is allowed:
  - The PE enters Debug state on the next instruction.
- Halting is prohibited:
  - The PE does not enter Debug state.

The value in EDESR.SS depends on whether halting was allowed or prohibited when the write to OSDLR_EL1.DLK completed, and so it might be 0 or 1. If a second `MSR` instruction clears OSDLR_EL1.DLK to 0, the PE must perform the required behavior of the state indicated by EDESR.SS before or immediately following the next Context Synchronization event, but it is not required to do so immediately.

See also Synchronization and Halting debug events.

H3.2.5.1 Changing the value of EDECR.SS when not in Debug state

If software changes the value of EDECR.SS when the PE is not in Debug state then behavior is CONSTRAINED UNPREDICTABLE, and one or more of the following behaviors occurs:

- The value of EDECR.SS becomes UNKNOWN.
- The state of the Halting Step state machine becomes UNKNOWN.
- On a reset of the PE, the value of EDECR.SS and the state of the Halting Step state machine are UNKNOWN.

H3.2.6 Stepping T32 `IT` instructions

In an implementation that supports the ITD control, the architecture permits a combination of one T32 `IT` instruction and another 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers such a pair of instructions to be one instruction.
- The PE considers such a pair of instructions be two instructions.

It is IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD bit. For example:

- The debug logic might consider such a pair of instructions as one instruction, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as two instructions, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as one instruction when the value of the applicable ITD field is 1, and as two instructions when the value of the ITD field is 0.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

HSCTLR.ITD

Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD

Applies to execution at EL0 or EL1 when EL1 is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCTLR_EL1.ITD

Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

H3.2.7 Disabling interrupts while stepping

When using Halting Step, the sequence of entering Debug state, interacting with the debugger, and then exiting Debug state for each instruction reduces the rate at which the PE executes instructions. However, the rate at which certain interrupts, such as timer interrupts, are generated might be fixed by the system. This means it might be necessary to disable interrupts while using Halting Step by setting EDSCR.INTdis, to allow the code being debugged to make forward progress.

H3.2.8 Syndrome information on Halting Step

Three EDSCR.STATUS encodings record different scenarios for entering Debug state on a Halting Step debug event:

Halting Step, normal

An instruction other than a Load-Exclusive instruction was stepped.

Halting Step, exclusive

A Load-Exclusive instruction was stepped.

Halting Step, no syndrome

The syndrome data is not available.

If the PE enters Debug state due to a Halting Step debug event immediately after stepping an instruction in the active-not-pending state, EDSCR.STATUS is set to either:

- Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
- Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

If the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code check, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, normal, or Halting Step, exclusive.

Otherwise, the PE enters Debug state without stepping an instruction. This means that the Halting Step state machine enters the active-pending state directly from the inactive state, without going through active-not-pending state. In this case, EDSCR.STATUS is set to Halting Step, no syndrome.

This happens when:

- The PE enters directly into the active-pending state on an exception return to a state where halting is allowed from EL3 when halting is prohibited at EL3.
- The active-pending state is entered for other reasons. See Synchronization and the Halting Step state machine

In addition, EDSCR.STATUS is CONSTRAINED UNPREDICTABLE when:

- A different exception is taken before the Halting Step debug event.
  In this case EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of:
  - Halting Step, no syndrome, or Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
  - Halting Step, no syndrome, or Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.
- The instruction that was stepped was an Exception Return instruction or an `ISB`. As these instructions are not in the Load-Exclusive instructions, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.
- The PE enters directly into the active-pending state on a Warm reset because EDECR.SS is set to 1. EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In all cases, if EDSCR.STATUS is not set to Halting Step, no syndrome, then it must indicate whether the stepped instruction was a Load-Exclusive instruction by setting EDSCR.STATUS to Halting Step, normal or Halting Step, exclusive.

Note

If the PE cannot determine whether the stepped instruction was a Load-Exclusive instruction or not, then it sets EDSCR.STATUS to Halting Step, no syndrome. For example, the exception is taken before the PE decodes the stepped instruction, or the exception means the PE has no valid stepped instruction to decode.

Note

The architecture does not permit an implementation to always set EDSCR.STATUS to Halting Step, no syndrome.

H3.2.9 Pseudocode description of Halting Step debug events

There are two pseudocode functions for Halting Step debug events:

- `HSAdvance()`. This is called after an instruction has executed and any exception generated by the instruction is taken, except for the case of an SVC, HVC, or SMC exception, in which case it is called before the exception is taken. `HSAdvance()` affects the next instruction.
- `CheckHaltingStep()`. This is called before the next instruction is executed. If a step is pending, it generates the debug event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.3 Halt Instruction debug event

A Halt Instruction debug event is generated when EDSCR.HDE == 1, halting is allowed, and the PE executes the Halt instruction, `HLT`.

The pseudocode for Halt Instruction debug events is described in HLT for A64 and HLT for T32 and A32.

`HLT` never generates a debug exception. It is treated as UNDEFINED if EDSCR.HDE == 0, or if halting is prohibited.

Note

A debugger can replace a program instruction with a Halt instruction to generate a Halt Instruction debug event. Debuggers that use the `HLT` instruction must be aware of the rules for concurrent modification of executable code, CMODX. The rules for concurrent modification and execution of instructions do not allow one thread of execution or an external debugger to replace an instruction with an `HLT` instruction when these same instructions are being executed by a different thread of execution. See Concurrent modification and execution of instructions.

The T32 `HLT` instruction is unconditionally executed inside an `IT` block, even when it is treated as undefined. The A32 `HLT` instruction is CONSTRAINED UNPREDICTABLE if the Condition code field is not `0b1110`, with the set of behaviors the same as for `BKPT`. See Architectural Constraints on UNPREDICTABLE Behaviors.

Note

The `HLT` instruction is part of the external debug solution from the introduction of Armv8-A. As such, the presence of the `HLT` instruction is not indicated in the ID registers. In particular, the AArch32 System register ID_ISAR0. Debug does not indicate the presence of the `HLT` instruction.

H3.3.1 HLT instructions as the first instruction in a T32 IT block

In an implementation that supports the ITD control, the architecture permits a combination of one T32 `IT` instruction and certain other 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

The T32 `HLT` instruction cannot be combined with an `IT` instruction in this way. In an implementation that supports the ITD control, if the first instruction in an IT block is an `HLT` instruction, then the behavior of the instruction depends on the value of the applicable ITD field:

- If the value of the ITD field is 1, then the combination is treated as undefined and an Undefined Instruction exception is generated either by the `IT` instruction or by the `HLT` instruction.
- If the value of the ITD field is 0, then the `HLT` instruction unconditionally executed.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

To set an Halt Instruction debug event on the first instruction of an IT block, debuggers must replace the `IT` instruction with an `HLT` instruction to ensure consistent behavior.

The ITD control fields are:

HSCTLR.ITD

Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD

Applies to execution at EL0 or EL1 when EL1 is using AArch32.

SCTLR_EL1.ITD

Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

Note

An `HLT` instruction is always unconditional, even within an IT block.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.4 Exception Catch debug event

Exception Catch debug events:

- Are generated when the corresponding bit in the Exception Catch Control Register, EDECCR, is set to 1 on all entries to a given Exception level. This means:
  - Exceptions taken to the Exception level.
  - Exception returns to the Exception level.
  - It is IMPLEMENTATION DEFINED whether a reset into an Exception level generates an Exception Catch debug event.
- Might be taken synchronously, after the exception or reset entry or the exception return has been processed by the PE.
- Ignore the Execution state of the target Exception level.
- Might be ignored if halting is prohibited.
- If FEAT_Debugv8p8 is implemented, might be pended.

The EDECCR register contains fields that control when Exception Catch debug events are generated.

See Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented and Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented.

For exception returns, the final Exception level of the exception return determines whether an Exception Catch debug event is generated. On an illegal exception return, an Exception Catch debug event is generated only if EDECCR is programmed to generate an Exception Catch debug event for an exception return to the current Exception level.

H3.4.1 Prioritization of Exception Catch debug events

The following rules define the prioritization of Exception Catch debug events:

- It is IMPLEMENTATION DEFINED whether Exception Catch debug events are higher or lower priority than Software Step exceptions and Halting Step debug events.
- Exception Catch debug events are higher priority than all synchronous exceptions other than Software Step exceptions and debug events other than Halting Step debug events.
- Exception Catch debug events are lower priority than Reset Catch debug events.
- When FEAT_Debugv8p2 is implemented and FEAT_Debugv8p8 is not implemented, Exception Catch debug events are higher priority than pending asynchronous exceptions. Otherwise, the prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event, is IMPLEMENTATION DEFINED.

Note

As described in Prioritization of Synchronous exceptions taken to AArch64 state, an exception trapping form of a Vector Catch debug event might generate a second debug exception as part of the exception entry, before the Exception Catch debug event is taken. See Vector Catch exceptions or Vector Catch exceptions.

H3.4.2 Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented

When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE halts and enters Debug state:

- The PE halts and enters Debug state before the first instruction at the handler is executed, after the exception entry has updated the program counter, PSTATE, and syndrome registers for the exception.
- The PE does not fetch instructions from the vector address before entering Debug state, if address translation is disabled in the translation regime at the target Exception level.
- A second unmasked asynchronous exception can be taken before the PE enters Debug state. If this second exception does not generate an Exception Catch debug event, the exception handler executed at the higher Exception level later returns to the trapped Exception level, causing the Exception Catch debug event to be generated again.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When an Exception Catch debug event is generated on exception return and halting is allowed at the target of the exception return:

- There is no prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event generated on an exception return.
- The PE halts and enters Debug state after the exception return has updated the program counter and PSTATE, and before the execution of the first instruction at the return address is completed.

Otherwise, when the PE is executing code at a given Exception level, and the corresponding EDECCR bit is 1, it is CONSTRAINED UNPREDICTABLE whether an Exception Catch debug event is generated.

Examples of this are:

- If the debugger writes to EDECCR so that the current Exception level is trapped.
- If the OS restore code writes to OSECCR so that the current Exception level is trapped.
- If the code executing in AArch32 state changes the Exception level or Security state other than by an exception return, and the target Exception level is trapped. See State and mode changes without explicit Context Synchronization events.

See also Debug state entry and debug event prioritization.

Note

It is possible to generate Exception Catch debug events:

- As a trap on all instruction fetches from the trapped Exception level as part of an instruction fetch.
- On entry to the Exception level, as described in Detailed Halting Step state machine behavior.

This is similar to the implementation options allowed for Vector Catch debug events. The architecture does not require that the event is generated following an `ISB` operation executed at the Exception level.

H3.4.3 Generating Exception Catch debug events when FEAT_Debugv8p2 is implemented and FEAT_Debugv8p8 is not implemented

Exception Catch debug events are generated as described in Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented, except:

- An Exception Catch debug event is generated only on exception entry or return.
- When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE halts and enters Debug state before any other asynchronous exception or debug event.

H3.4.4 Generating Exception Catch debug events when FEAT_Debugv8p8 is implemented

Exception Catch debug events are generated as described in Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented, except:

- An Exception Catch debug event is generated only on exception entry or return.
- When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE sets EDESR.EC to 1, and one of the following occurs:
  - The PE halts immediately. On entry to Debug state, EDESR.EC might be 0 or 1.
  - The PE might take an unmasked asynchronous exception, changing Exception level. This means that the PE might enter a state where halting is prohibited before the PE halts and enters Debug state.
- When an Exception Catch debug event is generated on exception return and halting is allowed at the target of the exception, EDESR.EC is unchanged.
- When EDESR.EC is 1 and halting is allowed, the PE halts and enters Debug state before completing any instruction.
- When EDESR.EC is 1 and halting is prohibited, an Exception Catch debug event is pending.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDESR.EC is cleared to 0 upon:
  - A write of 1 to EDESR.EC.
  - Exit from Debug state.

Note

If EDESR.EC is 1 and the PE executes a Context synchronizing exception return from a state where halting is prohibited to a state where halting is allowed, then the Exception Catch debug event is prioritized over any synchronous exception or synchronous debug event generated by the first instruction after the Context synchronization event, other than possibly a Halting Step debug event or a Software Step exception.

H3.4.5 Controlling Exception Catch debug events when FEAT_RME is implemented

When FEAT_RME is implemented, the fields EDECCR.{RLR, RLE, NSR, SR, NSE, SE} control generation of Exception Catch debug events:

- On exception entry but not on exception return.
- On exception return but not on exception entry.
- On exception entry and exception return.

Exception entry, reset and exception return Exception Catch debug events are enabled as shown in Table H3-6.

Table H3-5 Summary of Exception Catch debug event control when FEAT_RME is implemented

RLR<n> NSR<n> SR<n>	RLE<n> NSE<n> SR<n>	Behavior on exception return to ELn	Behavior on exception taken to ELn, and if resets are Exception Catch debug events, reset into ELn
0	0	No action.	No action.
0	1	Halt if allowed.	Halt if allowed.
1	0	Halt if allowed.	No action.
1	1	No action.	Halt if allowed.

H3.4.6 Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented

When FEAT_Debugv8p2 is implemented and FEAT_RME is not implemented, the fields EDECCR.{NSR, SR, NSE, SE} control generation of Exception Catch debug events:

- On exception entry but not on exception return.
- On exception return but not on exception entry.
- On exception entry and exception return.

Exception entry, reset and exception return Exception Catch debug events are enabled as shown in Table H3-6.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H3-6 Summary of Exception Catch debug event control when FEAT_Debugv8p2 is implemented and FEAT_RME is not implemented

(N)SR<n>	(N)SE<n>	Behavior on exception return to ELn	Behavior on exception taken to ELn, and if resets are Exception Catch debug events, reset into ELn
0	0	No action.	No action.
0	1	Halt if allowed.	Halt if allowed.
1	0	Halt if allowed.	No action.
1	1	No action.	Halt if allowed.

H3.4.7 Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented

When FEAT_Debugv8p2 is not implemented and FEAT_RME is not implemented, all Exception Catch debug events are enabled by a combination of the fields NSE and SE in EDECCR, as shown in Table H3-7.

Table H3-7 Summary of Exception Catch debug event control when FEAT_Debugv8p2 and FEAT_RME are not implemented

(N)SE<n>	Behavior on exception taken to ELn, return to ELn, and if resets are Exception Catch debug events, reset into ELn
0	No action.
1	Halt if allowed.

H3.4.8 Examples of Exception Catch debug events

If EDECCR == `0x0020`, meaning that the Exception Catch debug event is enabled for Non-secure EL1, then the following exceptions generate Exception Catch debug events:

- An exception taken from Non-secure EL0 to Non-secure EL1.
- An exception return from EL2 to Non-secure EL1.
- An exception return from EL3 to Non-secure EL1.

For example, on taking a Data Abort exception from Non-secure EL0 to Non-secure EL1, using AArch64:

- ELR_EL1 and SPSR_EL1 are written with the preferred return address and PE state for a return to EL0.
- ESR_EL1 and FAR_EL1 are written with the syndrome information for the exception.
- DLR_EL0 is set to VBAR_EL1 + `0x400`, the synchronous exception vector.
- DSPSR_EL0 is written with the PE state for an exit to EL1.

The following do not generate Exception Catch debug events:

- An exception taken from EL0 to EL2, in either Security state, or EL3.
- An exception return from EL2, in either Security state, to EL0.
- An exception taken from Secure EL0 to Secure EL1.
- An exception return from EL3 to Secure EL1.

H3.4.9 Pseudocode description of Exception Catch debug events

The pseudocode functions `CheckExceptionCatch()` and `CheckPendingExceptionCatch()` are described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.5 External Debug Request debug event

External Debug Request debug events are asynchronous debug events.

An External Debug Request debug event can be generated by any of the following:

- When signaled by the embedded cross-trigger. See The Embedded Cross-Trigger Interface.
- The trace unit. See ETE external debug request.
- The Trace Buffer Unit. See Trace Buffer Unit external debug request.
- The Performance Monitoring Unit. See PMU Overflow external debug request.
- An implementation might also support IMPLEMENTATION DEFINED ways of generating an External Debug Request debug event.

H3.5.1 Synchronization and External Debug Request debug events

For all Context synchronization events, if an External Debug Request debug event is asserted before the Context Synchronization event, and the External Debug Request debug event remains asserted and halting is allowed after the Context synchronization event, then the debug event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution.

An External Debug Request debug event that is being asserted when the PE comes out of reset is taken, and the PE enters Debug state before the first instruction after the reset completes its execution, provided that halting is allowed when the PE exits reset state.

If the first instruction after the Context Synchronization event or after coming out of reset generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Otherwise, when all of the following are true, External Debug Request debug events must be taken in finite time, without requiring the synchronization of any necessary change to the external authentication interface:

- Halting is allowed.
- The PE is not in a low-power state that is not required to exit on an external debug request. See Core power domain power states.

External debug request is a wakeup event for `WFI`, `WFIT`, `WFE`, or `WFET` instructions. See Mechanisms for entering a low-power state.

Note

These rules are based on the rules that apply when taking asynchronous exceptions. See Asynchronous exception types.

If an unmasked External Debug Request debug event was pending but is changed to not pending before it is taken, then the architecture permits the External Debug Request debug event to be taken, but does not require this to happen. If the External Debug Request debug event is taken then it must be taken before the first Context Synchronization event after the External Debug Request debug event was changed to not pending.

Example H3-3 shows an example of the synchronization requirements.

Example H3-3 Synchronization requirements

Secure software locks up in a tight loop, so it executes indefinitely without any synchronization operations. An External debug request must be able to break the PE out of that loop. This is a requirement even if **DBGEN** or **SPIDEN** or both are LOW on entry to the loop, meaning that halting is prohibited, and are only asserted HIGH later.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.5.2 ETE external debug request

If FEAT_Debugv8p9 and FEAT_ETEv1p3 are implemented, and all of the following are true, the PE halts and enters Debug state:

- EDECR.TRCE is 1.
- ETEEvent 0 occurs.
- Halting is allowed.

When the PE halts and enters Debug state due to an ETE external debug request:

- DLR_EL0 is set to the address of the first instruction that must be executed on exit from Debug state. DSPSR_EL0 is correct for this instruction.
- EDSCR.STATUS is set to `0b010011`, external debug request.

See also Trace Buffer Unit external debug request.

H3.5.3 Trace Buffer Unit external debug request

When FEAT_Debugv8p9 and FEAT_TRBE_EXT are implemented, and all of the following apply, the PE halts and enters Debug state:

- EDECR.TRBE is 1.
- The Trace Buffer Unit is enabled.
- TRBSR_EL1.IRQ is 1.
- Halting is allowed.

Note

The Trace Buffer Unit is not required to be using External mode.

When the PE halts and enters Debug state due to a Trace Buffer Unit external debug request:

- DLR_EL0 is set to the address of the first instruction that must be executed on exit from Debug state. DSPSR_EL0 is correct for this instruction.
- EDSCR.STATUS is set to `0b010011`, external debug request.

H3.5.4 PMU Overflow external debug request

When FEAT_Debugv8p9 and FEAT_PMUv3p9 are implemented and all of the following are true, the PE halts and enters Debug state:

- EDECR.PME is 1.
- Any of the following are true:
  - An event counter <n> is implemented, the global enable control for the event counter is 1, PMINTENSET_EL1.P<n> is 1, and PMOVSSET_EL0.P<n> is 1.
  - PMCR_EL0.E is 1, PMINTENSET_EL1.C is 1, and PMOVSSET_EL0.C is 1.
  - FEAT_PMUv3_ICNTR is implemented, PMCR_EL0.E is 1, PMINTENSET_EL1.F0 is 1, and PMOVSSET_EL0.F0 is 1.

  That is, the Performance Monitors overflow trigger event is asserted.
- Halting is allowed.

When the PE halts and enters Debug state because of a PMU Overflow external debug request:

- DLR_EL0 is set to the address of the first instruction that must be executed on exit from Debug state. DSPSR_EL0 is correct for this instruction.
- EDSCR.STATUS is set to `0b010011`, external debug request.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.5.5 Pseudocode description of External Debug Request debug events

The `ExternalDebugRequest()` function is described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.6 OS Unlock Catch debug event

An OS Unlock Catch debug event is generated when enabled and the state of the OS Lock changes from locked to unlocked. When FEAT_DoPD is implemented, CTIDEVCTL.OSUCE enables an OS Unlock Catch debug event, otherwise EDECR.OSUCE enables an OS Unlock Catch debug event.

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled, and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. However, this is an indirect write to EDESR.OSUC, meaning the OS Unlock Catch debug event is not guaranteed to be taken before a subsequent Context synchronization event. If the PE enters Debug state or the OS Unlock Catch debug event is disabled before EDESR.OSUC becomes set to 1, then EDESR.OSUC might not be set.

OS Unlock Catch debug events are not generated if the OS Lock is unlocked when the PE is in Debug state. See also Synchronization and Halting debug events.

EDESR.OSUC is cleared to 0 on a Warm reset and on exiting Debug state.

H3.6.1 Using the OS Unlock Catch debug event

When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a debug register that is implemented in the External debug power domain. However, if a debugger attempts to access a debug register that is implemented in the Core power domain when the Core power domain registers cannot be accessed, and that access returns an error, the debugger must retry the access.

Regularly powering down the Core power domain can result in unreliable debugger behavior.

The debugger can program a Reset Catch debug event to halt the PE when it has powered up, and can program the debug registers from Debug state. However, if the PE boot software restores the debug registers, as described in Debug OS Save and Restore sequences, then newly written values are overwritten by the restore sequence.

The debugger can program an OS Unlock Catch debug event to halt the PE after the restore sequence has completed, and program the debug registers from Debug state.

H3.6.2 Pseudocode description of OS Unlock Catch debug event

The `CheckOSUnlockCatch()` function is called when the OS Lock is unlocked.

The `CheckPendingOSUnlockCatch()` function is called before an instruction is executed. If an OS Unlock Catch is pending, it generates the debug event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.7 Reset Catch debug events

A Reset Catch debug event is generated when enabled, and the PE exits reset state. When the Reset Catch debug event is generated, it is recorded by setting EDESR.RC to 1. When FEAT_DoPD is implemented, CTIDEVCTL.RCE enables a Reset Catch debug event, otherwise EDECR.RCE enables a Reset Catch debug event.

If halting is allowed when the event is generated, the Reset Catch debug event is taken immediately and synchronously. On entering Debug state, DLR has the address of the reset vector. The PE must not fetch any instructions from memory.

Otherwise, the Reset Catch debug event is pended and taken when halting is allowed. See Synchronization and Halting debug events for more information.

This means that EDESR.RC is set to the value of EDECR.RCE or CTIDEVCTL.RCE on a Warm reset. EDESR.RC is cleared to 0 on exiting Debug state.

H3.7.1 Pseudocode description of Reset Catch debug event

The `CheckResetCatch()` function is called after reset before executing any instruction.

The `CheckPendingResetCatch()` function is called before an instruction is executed. If a Reset Catch is pending, it generates the Reset Catch debug event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.8 Software Access debug event

When the value of EDSCR.TDA is 1, software access to the following AArch64 and AArch32 debug System registers generate a Software Access debug event:

- The Breakpoint Value Registers, DBGBVR.
- The Breakpoint Control Registers, DBGBCR.
- The Watchpoint Value Registers, DBGWVR.
- The Watchpoint Control Registers, DBGWCR.

However, EDSCR.TDA is ignored if any of the following applies:

- The value of OSLSR.OSLK == 1, meaning that the OS Lock is locked.
- Halting is prohibited. See Halting allowed and halting prohibited.
- The register access generates a higher priority synchronous exception or debug event.

If FEAT_TRBE_EXT or FEAT_ETEv1p3 is implemented, then when all of the following apply, `MRS` and `MSR` accesses to ETE and TRBE System registers generate a Software Access debug event:

- EDSCR2.TTA is 1.
- The value of OSLSR_EL1.OSLK == 0, meaning that the OS Lock is unlocked.
- Halting is allowed.
- The instruction does not generate any higher priority synchronous exception or debug event.

Note

- The only accesses to the specified registers that generate a Software Access debug event are:
  - Accesses to System registers in AArch64 state.
  - Accesses to System registers in the (`coproc` == `0b1110`) encoding space in AArch32 state.
- Accesses by a PE using the external debug interface never generate a Software Access debug event.

H3.8.1 Pseudocode description of Software Access debug event

The `CheckSoftwareAccessToDebugRegisters()` function is described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.9 Synchronization and Halting debug events

The behavior of external debug depends on:

- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.
- The state of the external authentication interface.

For some registers, all read and write accesses that update the register occur in program order, without any additional synchronization, but others require an explicit Context Synchronization event. For more information on the synchronization of register updates, see:

- Synchronization requirements for AArch64 System registers.
- Synchronization of changes to the external debug registers.
- State and mode changes without explicit Context Synchronization events.

Changes to the external authentication interface do not require explicit synchronization to affect External Debug Request debug events. See Synchronization and External Debug Request debug events.

For changes that require explicit synchronization, it is CONSTRAINED UNPREDICTABLE whether instructions between the change and the Context Synchronization event observe the old state or the new state.

This means that any change to these registers or the external authentication interface requires explicit synchronization by a Context Synchronization event before the change takes effect. This ensures that for instructions appearing in program order after the change, the change affects the following:

- The generation and behavior of Breakpoint and Watchpoint debug events. See Synchronization and debug exceptions for exceptions taken from AArch64 state, or Synchronization and debug exceptions for exceptions taken from AArch32 state.
- The generation of all Halting debug events by instructions.
- Taking a pending Halting debug event or other asynchronous debug event. See:
  - Pending Halting debug events.
  - Synchronization and External Debug Request debug events.
- The behavior of the Halting Step state machine. See Synchronization and the Halting Step state machine.

H3.9.1 Pending Halting debug events

A Halting debug event might be pending:

- If Halting Step of an instruction sets EDESR.SS to 1, and halting is prohibited following the step, then the Halting Step state machine is inactive but a Halting Step debug event is pending.
- If a Reset Catch debug event sets EDESR.RC to 1, and halting is prohibited following reset, then a Reset Catch debug event is pending.
- If an OS Unlock Catch debug event sets EDESR.OSUC to 1, then an OS Unlock Catch debug event is pending.
- If FEAT_Debugv8p8 is implemented and an Exception Catch debug event sets EDESR.EC to 1, and before the PE takes the Exception Catch debug event, the PE takes an asynchronous exception, then an Exception Catch debug event is pending.
- If FEAT_Debugv8p9 and FEAT_ETEv1p3 are implemented, and an ETE external debug request is generated, and the PE enters a state where halting is prohibited before the PE halts and enters Debug state.
- If FEAT_Debugv8p9 and FEAT_TRBE_EXT are implemented, and a Trace Buffer Unit external debug request is generated, and the PE enters a state where halting is prohibited before the PE halts and enters Debug state.
- If FEAT_Debugv8p9 and FEAT_PMUv3p9 are implemented, and a PMU overflow external debug request is generated, and the PE enters a state where halting is prohibited before the PE halts and enters Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pending Halting debug events are taken asynchronously when halting is allowed.

Pending Halting debug events are discarded by a Cold reset. The debugger can also force a pending event to be dropped by writing to EDESR.

Any Halting debug event that is observed as pending in the EDESR before a Context Synchronization event is taken and the PE enters Debug state before the first instruction following the Context Synchronization event completes its execution. This is possible only if halting is allowed after completion of the Context Synchronization event.

If the first instruction after the Context Synchronization event generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken, unless both:

- A Halting Step debug event is pending. EDESR.SS == 1.
- The Context Synchronization event is an exception return from EL3 when halting is prohibited at EL3 to a state where halting is allowed.

Example H3-4

Context synchronization event

When FEAT_RME is not implemented, FEAT_ExS is implemented, and SCTLR_EL3.EOS is `0b1`, an exception return from EL3 using AArch64 with `ExternalSecureInvasiveDebugEnabled()` == FALSE to Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE is such an event.

In this case the order in which the debug events are handled is specified to avoid a double-step. See Entering the active-pending state.

If FEAT_Debugv8p8 is implemented and EDESR.EC is 1, when a pending Exception Catch debug event is taken following a Context Synchronization event, the PE enters Debug state before fetching any instruction:

- At ELn in Non-secure state, if EDECCR.NSE<n> is 1 and address translation is disabled at ELn.
- At ELn in Secure state, if EDECCR.SE<n> is 1 and address translation is disabled at ELn.
- At ELn in Realm state, if EDECCR.RLE<n> is 1 and address translation is disabled at ELn.

If an asynchronous exception is also pending after the Context Synchronization event then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Note

These rules are based on the rules that apply to taking asynchronous exceptions. See Asynchronous exception types.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H4
The Debug Communication Channel and Instruction Transfer Register

This chapter describes communication between a debugger and the implemented debug logic, using the *Debug Communications Channel* (DCC) and the *Instruction Transfer Register* (ITR), and associated control flags. It contains the following sections:

- Introduction.
- DCC and ITR registers.
- DCC and ITR access modes.
- Flow control of the DCC and ITR registers.
- Synchronization of DCC and ITR accesses.
- Interrupt-driven use of the DCC.
- Pseudocode description of the operation of the DCC and ITR registers.

Note

Where necessary, Table K14-1 disambiguates the general register references used in this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.1 Introduction

The *Debug Communications Channel*, DCC, is a channel for passing data between the PE and an external agent, such as a debugger. The DCC provides a communications channel between:

- An external debugger, described as the *debug host*.
- The debug implementation on the PE, described as the *debug target*.

The DCC can be used:

- As a 32-bit full-duplex channel.
- As a 64-bit half-duplex channel.

The DCC is an essential part of Debug state operation and can also be used in Non-debug state.

The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state.

The PE includes flow-control mechanisms for both the DCC and ITR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.2 DCC and ITR registers

The DCC comprises *data transfer registers*, the DTRs, and associated flow-control flags. The data transfer registers are DTRRX and DTRTX.

The ITR comprises a single register, EDITR, and associated flow-control flags.

In AArch64 state, software can access the data transfer registers as:

- A receive and transmit pair for 32-bit full duplex operation:
  - The write-only DBGDTRTX_EL0 register to transmit data.
  - The read-only DBGDTRRX_EL0 register to receive data.
- A single 64-bit read/write register, DBGDTR_EL0, for 64-bit half-duplex operation.
- The read/write OSDTRTX_EL1 and OSDTRRX_EL1 registers for save and restore.

In AArch32 state, software can access the data transfer registers only as:

- A receive and transmit pair, for 32-bit full duplex operation:
  - The write-only DBGDTRTXint register to transmit data.
  - The read-only DBGDTRRXint register to receive data.
- The read/write DBGDTRTXext and DBGDTRRXext registers for save and restore.

The data transfer registers are also accessible by the external debug interface as a pair of 32-bit registers, DBGDTRRX_EL0 and DBGDTRTX_EL0. Both registers are read/write, allowing both 32-bit full-duplex and 64-bit half-duplex operation.

The DCC flow-control flags are EDSCR.{RXfull, TXfull, RXO, TXU}:

- The RXfull and TXfull ready flags are used for flow-control and are visible to software in the Debug system registers in DCCSR.
- The RX overrun flag, RXO, and the TX underrun flag, TXU, report flow-control errors.
- The flow-control flags are also accessible by software as simple read/write bits for saving and restoring over a powerdown when the OS Lock is locked in DSCR.
- The flow-control flags are accessible from the external debug interface in EDSCR.

Figure H4-1 shows the System register and external debug interface views of the EDSCR and DTR registers in both AArch64 state and AArch32 state. These figures do not include the save and restore views.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

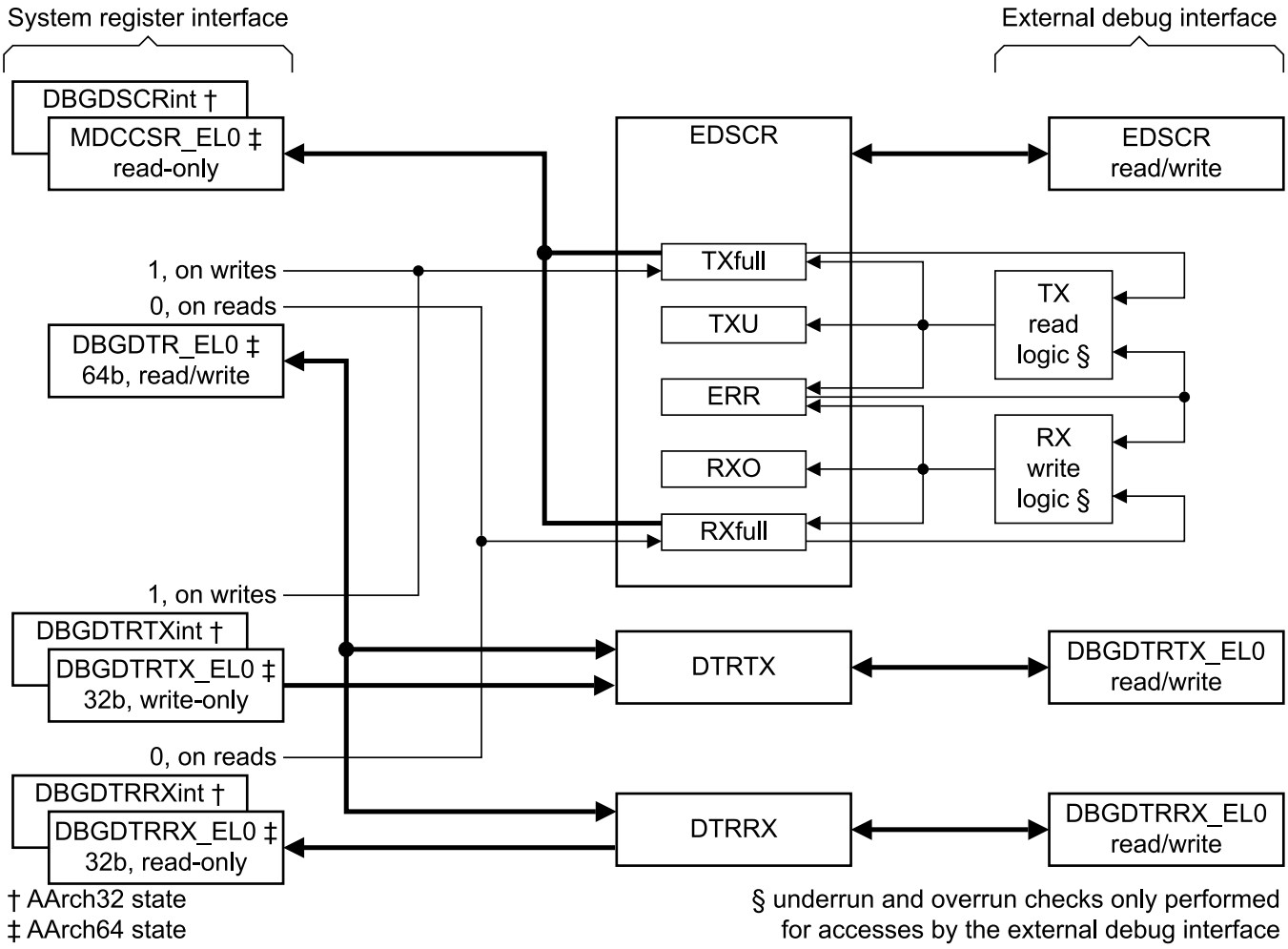

Figure H4-1 System register and external debug interface views of EDSCR and DTR registers, Normal access mode

EDITR and the ITR flow-control flags, EDSCR.{ITE, ITO} are accessible only by the external debug interface:

- The EDITR specifies an instruction to execute in Debug state.
- The ITR empty flag, ITE, is used for flow-control.
- The ITR overrun flag, ITO, reports flow-control errors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

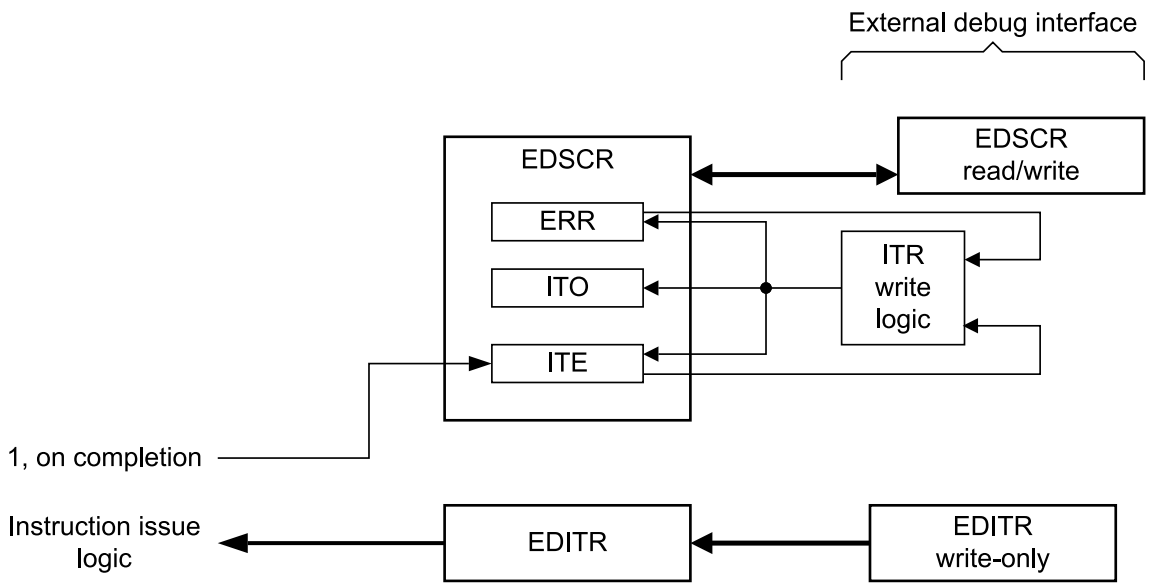

Figure H4-2 External debug interface views of EDSCR and EDITR registers, Normal access mode

The sticky overflow flag, EDSCR.ERR, is used by both the DCC and ITR to report flow-control errors.

To save and restore the DCC registers for an external debugger over powerdown, software uses:

- The MDSCR_EL1, OSDTRTX_EL1, and OSDTRRX_EL1 registers in AArch64 state.
- The DBGDSCRext, DBGDTRTXext, and DBGDTRRXext registers in AArch32 state.

Note

There is no save and restore mechanism for the ITR registers as the ITR is used only in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

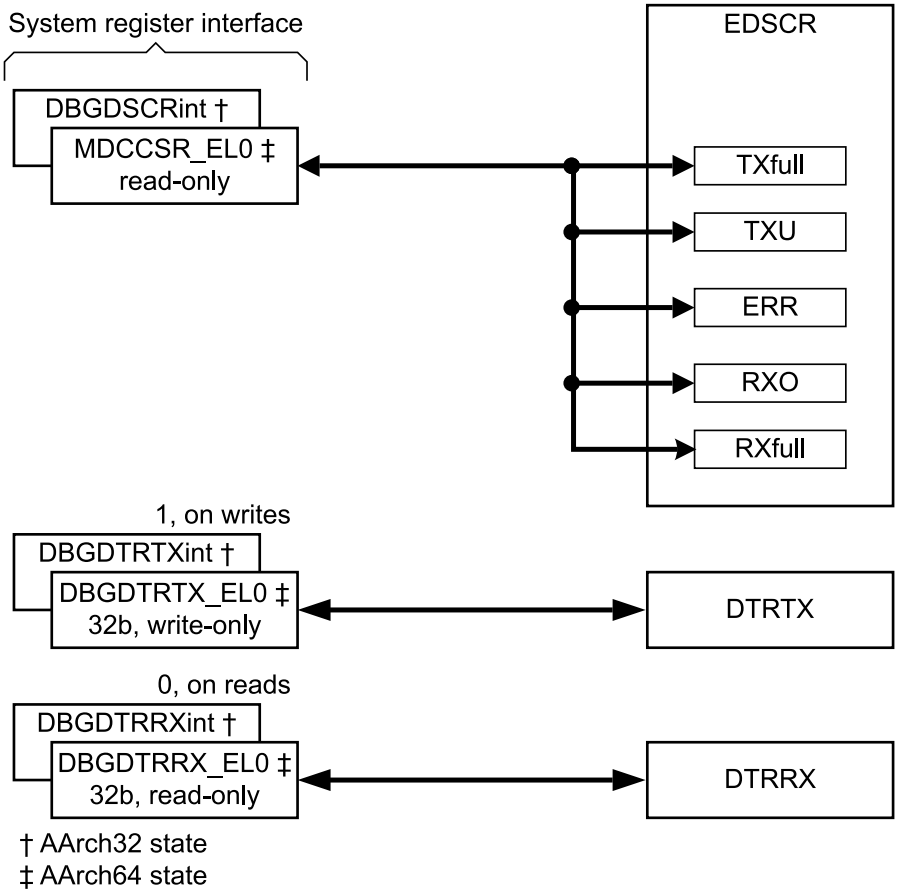

Figure H4-3 System register views of EDSCR and DTR registers for save and restore

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.3 DCC and ITR access modes

The DCC and ITR support two access modes:

DCC and ITR access mode, links to description	Applies when:
Normal access mode	EDSCR.MA == 0 or the PE is in Non-debug state
Memory access mode	EDSCR.MA == 1 and the PE is in Debug state

H4.3.1 Normal access mode

The Normal access mode allows use of the DCC as a communications channel between target and host. It also allows the use of the ITR for issuing instructions to the PE in Debug state.

In Normal access mode, if there is no overrun or underrun, the following occurs:

For accesses by software:

- Direct writes to DBGDTRTX update the value in DTRTX and indirectly write 1 to TXfull.
- Direct reads from DBGDTRRX return the value in DTRRX and indirectly write 0 to RXfull.
- In AArch64 state, direct writes to DBGDTR_EL0 update both DTRTX and DTRRX, indirectly write 1 to TXfull, and do not change RXfull:
  - DTRTX is set from bits[31:0] of the transfer register.
  - DTRRX is set from bits[63:32] of the transfer register.
- In AArch64 state, direct reads from DBGDTR_EL0 return the concatenation of DTRRX and DTRTX, indirectly write 0 to RXfull, and do not change TXfull:
  - Bits[31:0] of the transfer register are set from DTRRX.
  - Bits[63:32] of the transfer register are set from DTRTX.

Note

For DBGDTR_EL0, the word order is reversed for reads with respect to writes.

Software reads TXfull and RXfull using DCCSR.

For accesses by the external debug interface:

- Writes to EDITR trigger the instruction to be executed if the PE is in Debug state:
  - If the PE is in AArch64 state, this is an A64 instruction.
  - If the PE is in AArch32 state, this is a T32 instruction. The T32 instruction is a pair of halfwords where the first halfword is taken from the lower 16-bits, and the second halfword is taken from the upper 16-bits.
- Reads of DBGDTRTX_EL0 return the value in DTRTX and indirectly write 0 to TXfull.
- Writes to DBGDTRTX_EL0 update the value in DTRTX and do not change TXfull.
- Reads of DBGDTRRX_EL0 return the value in DTRRX and do not change RXfull.
- Writes to DBGDTRRX_EL0 update the value in DTRRX and indirectly write 1 to RXfull. TXfull and RXfull are visible to the external debug interface in EDSCR.

The PE detects overrun and underrun by the external debug interface, and records errors in EDSCR.{TXU, RXO, ITO, ERR}. See Flow control of the DCC and ITR registers.

See also Synchronization of DCC and ITR accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.3.2 Memory access mode

When the PE is in Debug state, *Memory access mode* can be selected to accelerate word-aligned block reads or writes of memory by an external debugger. Memory access mode can be enabled only in Debug state, and no instructions can be issued directly by the debugger when in Memory access mode.

If there is no overrun or underrun when in Memory access mode, an access by the external debug interface results in the following:

- External reads from DBGDTRTX_EL0 cause:
  1. The existing value in DTRTX to be returned. This clears EDSCR.TXfull to 0.
  2. The equivalent of `LDR W1,[X0],#4`, if in AArch64 state, or `LDR R1,[R0],#4`, if in AArch32 state, to be executed.
  3. The equivalent of the `MSR DBGDTRTX_EL0,X1` instruction, if in AArch64 state, or the `MCR p14,0,R1,c0,c5,0` instruction, if in AArch32 state, to be executed.
  4. EDSCR.{TXfull, ITE} to be set to {1, 1}, and X1 or R1 to be set to an UNKNOWN value.
- External writes to DBGDTRRX_EL0 cause:
  1. The value in DTRRX to be updated. This sets EDSCR.RXfull to 1.
  2. The equivalent of the instruction `MRS X1,DBGDTRRX_EL0`, if in AArch64 state, or `MRC p14,0,R1,c0,c5,0` if in AArch32 state, to be executed.
  3. The equivalent of the instruction `STR W1,[X0],#4`, if in AArch64 state, or `STR R1,[R0],#4`, if in AArch32 state, to be executed.
  4. EDSCR.{RXfull, ITE} to be set to {0, 1}, and X1 or R1 to be set to an UNKNOWN value.
- External reads from DBGDTRRX_EL0 return the last value written to DTRRX.
- External writes to EDITR generate an overrun error.

During these accesses, EDSCR.{TXfull, RXfull, ITE} are used for flow control.

Note

An overrun or underrun might result in EDSCR.ERR being set to 1 asynchronously to the sequence of operations that are outlined in this section. As this is timing-dependent, it is UNPREDICTABLE when the EDSCR.ERR flag affects the instructions and therefore whether neither instruction, only the first instruction, or both instructions are executed. If the second instruction is executed, then the first instruction must have been executed. However, in each case X1 or R1 is set to an UNKNOWN value. This means that:

- In both cases, if the memory access instruction is not executed, then the base register X0 or R0 is not updated, meaning the debugger can determine the last accessed location.
- In the list describing External reads from DBGDTRTX_EL0, DTRTX and EDSCR.TXfull get set to UNKNOWN values. If the load was executed, then the value that was read by the PE is lost. This means the operation might need to be repeated by the debugger, and it is not advisable to use Memory access mode to read from read-sensitive locations using the underrun and overrun detection for flow control.
- In the list describing External writes to DBGDTRRX_EL0, EDSCR.RXfull is set to an UNKNOWN value.

A Data Abort from the memory access can also set EDSCR.ERR to 1. See Data Aborts in Memory access mode.

The architecture does not require precisely when these flags are set or cleared by the sequence of operations outlined in this section. For example, in the case of an external write to DBGDTRRX_EL0, in AArch64 state, RXfull might be cleared after step 2, or it might not be cleared until after step 3, as an implementation is free to fuse these steps into a single operation. The architecture does require that the flags are set as at step 4 when the PE is ready to accept a further read or write without causing an overrun error or an underrun error.

The process outlined in this section represents a simple sequential execution model of Memory access mode. An implementation is free to pipeline, buffer, and reorder instructions and transactions, as long as the following remain true:

- Data items are transferred into and out of the DTR in order and without loss of data, other than as a result of an overrun or an underrun.
- Data Aborts occur in order.
- The constraints of the memory type are met.
- In the list describing External reads from:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The MSR equivalent operation at step 3 of the sequence reads the value loaded by step 2.
  - If the list is performed in a loop, for all but the first iteration of this list, the value read by step 1 returns the values written by the MSR equivalent operation at the previous iteration of step 3.
- In the list describing External writes to:
  - The MRS equivalent operation at step 2 of the sequence returns the value written at step 1.
  - The STR equivalent at step 3 of the sequence writes the value read at step 2.
- If the PE cannot accept a read or write, as applicable, during the sequence, then the flags are updated to indicate an overrun or underrun.

See Flow control of the DCC and ITR registers for more information on overrun and underrun.

H4.3.2.1 Ordering, access sizes and effect on Exclusives monitors

For the purposes of memory ordering, access sizes, and effect on the Exclusives monitor, accesses in Memory access mode are consistent with load/store word instructions executed by the PE.

The simple sequential access model of Memory access mode, as stated in Memory access mode, must also be ordered with respect to instructions executed as a result of External writes to EDITR in Normal mode both before and after accesses to the DTR registers in Memory access mode.

H4.3.2.2 Data Aborts in Memory access mode

If a memory access generates a Data Abort, then:

- The Data Abort exception is taken. See Exceptions in Debug state:
  - This means EDSCR.ERR is set to 1, see Cumulative error flag.
  - If the Data Abort occurs on stage 2 of an address translation, then the values returned in the ISV field and in bits[23:14] of the ISS are UNKNOWN.
    If this Data Abort is taken to EL2 using AArch64, the ISS is returned by ESR_EL2. ISS encoding for an exception from a Data Abort describes the usual encoding of this ISS.
    If EL2 is using AArch32 and this Data Abort is taken to Hyp mode, the ISS is returned by HSR. ISS encoding for exception from a Data Abort describes the usual encoding of this ISS.
- Register R0 retains the address that generated the abort.
- Register R1 is set to an UNKNOWN value.
- EDSCR.TXfull, for a load, or EDSCR.RXfull, for a store, is set to an UNKNOWN value.
- DTRTX, for a load, or DTRRX, for a store, is set to an UNKNOWN value.
- EDSCR.ITE is set to 1.

H4.3.2.3 Illegal Execution state exception

If PSTATE.IL is set to 1 when EDSCR.MA == 1, then on an external write access to DBGDTRRX_EL0 or an external read from DBGDTRTX_EL0, it is CONSTRAINED UNPREDICTABLE whether the PE:

- Takes an Illegal Execution state exception without performing any operations. In this case:
  - EDSCR.ERR is set to 1, see Cumulative error flag.
  - Register R0 is unchanged.
  - Register R1 is set to an UNKNOWN value.
  - EDSCR.TXfull or EDSCR.RXfull, as applicable, is set to an UNKNOWN value.
  - DTRTX or DTRRX, as applicable, is set an UNKNOWN value.
  - EDSCR.ITE is set to 1.

  See also Exceptions in Debug state.
- Ignores PSTATE.IL.

Note

The typical usage model for Memory access mode involves executing instructions in Normal access mode to set up X0 before setting EDSCR.MA to 1. These instructions generate an Illegal state exception if PSTATE.IL is set to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.3.2.4 Alignment constraints

If the address in R0 is not aligned to a multiple of four, the behavior is as follows:

- For each external DTR access a CONSTRAINED UNPREDICTABLE choice of:
  1. The PE makes an unaligned memory access to R0. If alignment checking is enabled for the memory access, this generates an Alignment fault.
  2. The PE makes a memory access to `Align(X[0],4)` in AArch64 state, or `Align(R[0],4)` in AArch32 state.
  3. The PE generates an Alignment fault, regardless of whether alignment checking is enabled.
  4. The PE does nothing.
- Following each memory access, if there is no Data Abort, R0 is updated with an UNKNOWN value.
- For external writes to DBGDTRRX_EL0, if the PE writes to memory, an UNKNOWN value is written.
- For external reads of DBGDTRTX_EL0 an UNKNOWN value is returned.
- The RXfull and TXfull flags are left in an UNKNOWN state, meaning that a DBGDTRTX_EL0 read can trigger a TX underrun, and a DBGDTRTX_EL0 write can trigger an RX overrun.

H4.3.3 Memory-mapped accesses to the DCC and ITR

Writes to the flags in EDSCR by external debug interface accesses to the DCC and the ITR registers are indirect writes, because they are a side-effect of the access. The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see Register access permissions for memory-mapped accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.4 Flow control of the DCC and ITR registers

- Ready flags.
- Buffering writes to EDITR.
- Overrun and underrun flags.
- Cumulative error flag.

H4.4.1 Ready flags

In Normal access mode:

- For the DTR registers there are two ready flags:
  - EDSCR.RXfull == 1 indicates that DBGDTRRX_EL0 contains a valid value that has been written by the external debugger and not yet read by software running on the target.
  - EDSCR.TXfull == 1 indicates that DBGDTRTX_EL0 contains a valid value that has been written by software running on the target and not yet read by an external debugger.
- For the ITR register there is a single ready flag:
  - EDSCR.ITE == 1 indicates that the PE is ready to accept an instruction to the ITR.

    Note

    The architecture permits a PE to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, as long as those instructions are discarded if EDSCR.ERR is 1, either by an underrun or overrun or by any of the other error conditions described in this architecture, such as an instruction generating an abort.

In Memory access mode:

- EDSCR.{RXfull, ITE} == {0, 1} indicates that DBGDTRRX_EL0 is empty and the PE is ready to accept a word external write to DBGDTRRX_EL0.
- EDSCR.{TXfull, ITE} == {1, 1} indicates that DBGDTRTX_EL0 is full and the PE is ready to accept a word external read from DBGDTRTX_EL0.

All other values indicate that the PE is not ready, and result in a DTR overrun or underrun error, an ITR overrun error, or both, as defined in Overrun and underrun flags.

EDSCR.{ITE, RXfull, TXfull} shows the status of the ITR and DCC registers. It ignores the question of whether a read or write cannot be accepted because, for example, EDSCR.ERR is set or the OPTIONAL Software Lock is locked for memory-mapped accesses (EDLSR.SLK == 1).

H4.4.2 Buffering writes to EDITR

The architecture permits a processor to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, provided that:

- Those instructions are discarded if EDSCR.ERR is set to 1, either by an underrun or an overrun, or by any other error conditions described in this architecture, such as an instruction generating an abort.
- The PE maintains the simple sequential execution model with the order of instructions determined by the order in which the PE accepts the EDITR writes. In particular, the buffered instructions must be executed in the Execution state consistent with a simple sequential execution of the instructions, even if one of the previous instructions is a state changing operation, such as `DCPS` or `DRPS`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.4.3 Overrun and underrun flags

Each of the ready flags has a corresponding overrun or a corresponding underrun flag. These are sticky status flags that are set if the register is accessed using the external debug interface when the corresponding ready flag is not in the ready state.

If the PE is in Debug state and Memory access mode, the corresponding error flag is also set if the PE is not ready to accept an operation because a previous load or store is still in progress. The sticky status flag remains set until cleared by writing 1 to EDRCR.CSE.

Note

The architecture permits a PE to continue to accept and buffer data to write to memory in Memory access mode.

Table H4-2 shows DCC and ITR ready flags and the overrun and underrun flags associated with them.

Table H4-2 DCC and ITR ready flags and the associated overrun/underrun flags

External debug interface access	Overrun/Underrun condition	EDSCR flag
Write DBGDTRRX_EL0	`EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')`	RXO
Read DBGDTRTX_EL0	`EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')`	TXU
Write EDITR	`Halted() && (EDSCR.ITE == '0' || EDSCR.MA == '1')`	ITO

When an overrun or underrun flag is set to 1, the cumulative error flag, EDSCR.ERR, described in Cumulative error flag, is also set to 1.

In the event of an external write to DBGDTRRX_EL0 or EDITR generating an overrun, or an external read from DBGDTRTX_EL0 generating an underrun:

- For a write, the written value is ignored.
- For a read, an UNKNOWN value is returned.
- EDSCR.TXfull, EDSCR.RXfull or EDSCR.ITE, as applicable, are not updated.

There is no overrun or underrun detection on external reads of DBGDTRRX_EL0 or external writes of DBGDTRTX_EL0.

There is no overrun or underrun detection of direct reads and direct writes of the DTR System registers by software:

- If RXfull == 0, a direct read of DBGDTRRX or DBGDTR_EL0 returns UNKNOWN.
- If TXfull == 1, a direct write of:
  - DBGDTRTX sets DTRTX to UNKNOWN.
  - DBGDTR_EL0 sets DTRRX and DTRTX to UNKNOWN.

See DCC accesses in Non-debug state for more information.

H4.4.3.1 Accessing 64-bit data

In AArch64 state, a software access to the DBGDTR_EL0 register and an external debugger access to both DBGDTRRX_EL0 and DBGDTRTX_EL0 can perform a 64-bit half-duplex operation.

However, there is only overrun and underrun detection on one of the external debug registers. That is:

- If software directly writes a 64-bit value to DBGDTR_EL0, only TXfull is set to 1, meaning:
  - A subsequent external write to DBGDTRRX_EL0 would not be detected as an overrun.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If the external debugger reads DBGDTRTX_EL0 first, software might observe MDCCSR_EL0.TXfull == 0 and send a second value before the external debugger reads DBGDTRRX_EL0, leading to an undetected overrun.
- On external writes to both DBGDTRRX_EL0 and DBGDTRTX_EL0, only RXfull is set to 1, meaning:
  - A subsequent direct write of DBGDTRTX_EL0 would not be detected as an overrun.
  - If the external debugger writes to DBGDTRRX_EL0 first, software might observe MDCCSR_EL0.RXfull == 1 and read a full 64-bit value, before the external debugger writes to DBGDTRTX_EL0, leading to an undetected underrun.

To avoid this, debuggers need to be aware of the data size used by software for transfers and ensure that 64-bit data is read or written in the correct order. If the PE is in Non-debug state, this order is as follows:

- The external debugger must check EDSCR.{RXfull, TXfull} before each transfer.
- To receive a 64-bit value from the target, the external debugger must read DBGDTRRX_EL0 before reading DBGDTRTX_EL0.
- To send a 64-bit value to the target, the external debugger must write to DBGDTRTX_EL0 before writing DBGDTRRX_EL0.

Because three accesses are required to transfer 64 bits of data, 64-bit transfers are not recommended for regular communication between host and target. The use of underrun and overrun detection means that only one access is required for 32 bits of data when using 32-bit transfers.

In Debug state, the debugger controls the instructions executed by the PE, so these limitations do not apply. 64-bit transfers provide a means to transfer a 64-bit general register between the host and the target in Debug state.

H4.4.4 Cumulative error flag

The cumulative error flag, EDSCR.ERR, is set to 1:

- On taking an exception from Debug state.
- On any signaled overrun or underrun in the DCC or ITR.

When EDSCR.ERR == 1:

- External reads of DBGDTRTX_EL0 do not have any side-effects.
- External writes to DBGDTRRX_EL0 are ignored.
- External writes to EDITR are ignored.
- No further instructions can be issued in Debug state. This includes any instructions previously accepted as external writes to EDITR that occur in program order after the instruction or access that caused the error.

This allows a debugger to stream data, or, in Debug state, instructions, to the target without having to:

- Check EDSCR.{RXfull, TXfull, ITE} before each access.
- Check EDSCR.{ITO, RXO, TXU} following each access, for overrun or underrun.
- Check PSTATE or other syndrome registers, or both, for an exception following each instruction executed in Debug state that might generate a synchronous exception.

The cumulative error flag remains set until cleared to 0 by writing 1 to EDRCR.CSE. However, the effect of writing 1 to EDRCR.CSE to clear EDSCR.ERR is CONSTRAINED UNPREDICTABLE when both of the following apply:

- The PE is in Debug state.
- The value of EDSCR.ITE is 0.

When these conditions apply and a value of 1 is written to EDRCR.CSE, either or both of the following might occur:

- EDSCR.ERR is not cleared to 0.
- Any instructions in EDITR that have not been executed might be executed subsequently, rather than being ignored.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This means that a debugger must poll EDSCR.ITE until it has the value 1, indicating that EDITR is empty, before writing to EDRCR.CSE to clear the EDSCR.ERR flag to 0.

For overruns and underruns, EDSCR.{ITO, RXO, TXU} record the error type.

H4.4.4.1 Pseudocode description of clearing the error flag

The `ClearStickyErrors()` pseudocode function is described in A-profile Architecture Pseudocode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.5 Synchronization of DCC and ITR accesses

In addition to the standard synchronization requirements for register accesses, the following subsections describe additional requirements that apply for the DCC and ITR registers:

- Summary of System register accesses to the DCC.
- DCC accesses in Non-debug state.
- Synchronization of DCC interrupt request signals.
- DCC and ITR access in Debug state.

In these sections, accesses by the external debug interface are referred to as external reads and external writes. Accesses to System registers are referred to as direct reads, direct writes, indirect reads, and indirect writes.

Note

In Synchronization requirements for AArch64 System registers external reads and external writes are described as forms of indirect access. This whole section uses more explicit terminology.

The DTR registers and the DCC flags, TXfull and RXfull, form a communication channel, with one end operating asynchronously to the other. Implementations must respect the ordering of accesses to these registers in order to maintain the correct behavior of the channel.

External reads of, and external writes to DBGDTRRX_EL0 and DBGDTRTX_EL0 are asynchronous to direct reads of, and direct writes to, DBGDTRRX, DBGDTRTX, and in AArch64 state DBGDTR_EL0, made by software using System register access instructions. The direct reads and direct writes indirectly write to the DCC flags. The external reads and external writes indirectly read the DCC flags to check for underrun and overrun.

Throughout this section:

DCC flags Means any or all of the following:

- The EDSCR.{RXfull.TXfull} ready flags.
- The EDSCR.RXO overrun flag.
- The EDSCR.TXU underrun flag.
- The EDSCR.ERR cumulative error flag.

ITR flags Means any or all of the following:

- The EDSCR.ITE ready flag.
- The EDSCR.ITO overrun flag.
- The EDSCR.ERR cumulative error flag.

H4.5.1 Summary of System register accesses to the DCC

System register accesses to the DTR registers are direct reads and writes of those registers, as shown in Table H4-3. Several of these instructions access the same registers using different encodings.

DBGDTRRX_EL0 and DBGDTRTX_EL0 are encoded as MRS and MSR accesses respectively to the same System register, even though they access different underlying register values. DBGDTRRX and DBGDTRTX are similarly encoded as MRC and MCR accesses respectively to the same System register. The encoding means that direct reads and writes using these encodings must be ordered with respect to each other. For more information, see Synchronization requirements for AArch64 System registers and Synchronization of changes to AArch32 System registers.

Table H4-3 shows a summary of System register accesses to the DCC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H4-3 Summary of System register accesses to the DCC

Operation	OS Lock	AArch64 (MRS/MSR)	AArch32 (MRC/MCR)	Description
Read	-	DBGDTRRX_EL0	DBGDTRRXint	Direct read of DTRRX. Indirect write to the DCC flags. An STC instruction that reads DBGDTRRXint makes an indirect write to DBGDSCRint.RXfull.
Write	-	DBGDTRTX_EL0	DBGDTRTXint	Direct write of DTRTX. Indirect write to the DCC flags. An LDC instruction that writes to DBGDTRTXint using a value read from memory is a direct write to DBGDTRTXint.
Read/write	-	DBGDTR_EL0	-	Direct read/write of both DTRRX and DTRTX. Indirect write to the DCC flags.
Read	-	MDCCSR_EL0	DBGDSCRint	Direct read of the DCC flags.
Read/write	-	OSDTRRX_EL1	DBGDTRRXext	Direct read/write of DTRRX.
Read/write	-	OSDTRTX_EL1	DBGDTRTXext	Direct read/write of DTRTX.
Read	Unlocked	MDSCR_EL1	DBGDSCRext	Direct read of DCC flags.
Read/write	Locked	MDSCR_EL1	DBGDSCRext	Direct read/write of DCC flags.

H4.5.2 DCC accesses in Non-debug state

In Non-debug state DCC accesses are as described in Normal access mode:

- If a direct read of DCCSR returns RXfull == 1, then a following direct read of DBGDTRRX, or in AArch64 state of DBGDTR_EL0, returns valid data and indirectly writes 0 to DCCSR.RXfull as a side-effect.
- If a direct read of DCCSR returns TXfull == 0, then a following direct write to DBGDTRTX, or in AArch64 state to DBGDTR_EL0, writes the intended value, and indirectly writes 1 to DCCSR.TXfull as a side-effect.

No Context Synchronization event is required between these two instructions. Overrun and underrun detection prevents intervening external reads and external writes affecting the outcome of the second instruction.

The indirect write to the DCC flags as part of the DTR access instruction is made atomically with the DTR access.

Because a direct read of DBGDTRRX is an indirect write to DCCSR.RXfull, it must occur in program order with respect to the direct read of DCCSR, meaning it must not return a speculative value for DTTRX that predates the RXfull flag returned by the read of DCCSR. The direct write to DBGDTRTX must not be executed speculatively.

Direct reads of DBGDTRRX, or in AArch64 state DBGDTR_EL0, and DCCSR, must occur in program order with respect to other direct reads of the same register using the same encoding.

The following accesses have an implied order within the atomic access:

- In the simple sequential execution of the program the indirect write of the DCC flags occurs immediately after the direct DTR access.

  Note

  For an access to DBGDTR_EL0, this means the indirect write happens after both DBGDTRRX_EL0 and DBGDTRTX_EL0 have been accessed.

- In the simple sequential execution model, for an external read of DBGDTRTX_EL0 or an external write of DBGDTRRX_EL0:
  - The check of the DCC flags for overrun or underrun occurs immediately before the access.
  - If there is no underrun or overrun, the update of the DCC flags occurs immediately after the access.
  - If there is underrun or overrun, the update of the DCC underrun or overrun flags occurs immediately after the access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All observers must observe the same order for accesses.

Note

These requirements do not create order where order does not otherwise exist. It applies only for ordered accesses.

Without explicit synchronization following external writes and external reads:

- The value written by the external write to DBGDTRRX_EL0 that does not overrun, must be observable to direct reads of DBGDTRRX and DBGDTR_EL0 in finite time.
- The DCC flags that are updated as a side-effect of the external write or external read must be observable:
  - To subsequent external reads of EDSCR.
  - To subsequent external reads of DBGDTRRX_EL0 when checking for underrun.
  - To subsequent external writes to DBGDTRTX_EL0 when checking for overrun.
  - To direct reads of DCCSR in finite time.

However, explicit synchronization is required to guarantee that a direct read of DCCSR returns up-to-date DCC flags. This means that if a signal is received from another agent that indicates that DCCSR must be read, an `ISB` is required to ensure that the direct read of DCCSR occurs after the signal has been received. This also synchronizes the value in DBGDTRRX, if applicable. However, if that signal is an interrupt exception triggered by **COMMIRQ**, **COMMTX**, or **COMMRX**, the exception entry is sufficient synchronization. See Synchronization of DCC interrupt request signals.

Explicit synchronization is required following a direct read or direct write:

- To ensure that a value directly written to DBGDTRTX is observable to external reads of DBGDTRTX_EL0.
- To ensure that a value directly written to DBGDTR_EL0 is observable to external reads of DBGDTRTX_EL0 and DBGDTRRX_EL0.
- To guarantee that the indirect writes to the DCC flags that were a side-effect of the direct read or direct write have occurred, and therefore that the updated values are:
  - Observable to external reads of EDSCR.
  - Observable to external reads of DBGDTRRX_EL0 when checking for underrun.
  - Observable to external writes of DBGDTRTX_EL0 when checking for overrun.
  - Returned by a following direct read of DCCSR.

See also Memory-mapped accesses to the DCC and ITR and Synchronization of changes to the external debug registers.

Note

These ordering rules mean that software:

- Must not read DBGDTRRX or DBGDTR_EL0 without first checking DCCSR.RXfull or if the previously-read value of DCCSR.RXfull is 0.

  It is not sufficient to read both registers and then later decide whether to discard the read value, as there might be an intervening write from the external debug interface.
- Must not write DBGDTRTX or DBGDTR_EL0 without first checking DCCSR.TXfull or if the previously-read value of DCCSR.TXfull is 1.

  The write to DBGDTRTX overwrites the value in DTRTX, and the external debugger might or might not have read this value.
- Must ensure there is an explicit Context Synchronization event following a DTR access, even if not immediately returning to read DCCSR again. This synchronization operation might be an exception return.

H4.5.2.1 Derived requirements

The rules for DCC accesses in Non-debug state are as follows:

- Following a direct read of DBGDTRRX when RXfull is 1:
  - If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by a following external write to DBGDTRRX, and the following external write does not overrun.
- Following a direct read of DBGDTR_EL0, when RXfull is 1:
  - If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write nor by a following direct write to DBGDTRTX.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by subsequent external writes to DBGDTRRX and DBGDTRTX in any order, and the following external write of DBGDTRRX will not overrun.
- Following a direct write to DBGDTRTX, when TXfull is 0:
  - If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the value returned by the external read must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by a following external read of DBGDTRRX must be the value written by the previous direct read, and the subsequent external read will not underrun.
- Following a direct write to DBGDTR_EL0, when TXfull is 0:
  - If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the values returned by the external read and by a subsequent external read of DBGDTRRX must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by subsequent external reads of DBGDTRRX and DBGDTRTX, in any order, must be the value written by the previous direct read, and the subsequent external read of DBGDTRTX does not underrun.
- Following an external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the value returned by the external read must not be affected by a following direct write to DBGDTRTX.
- Following a first external read DBGDTRRX and a following second external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the values returned by the external reads must not be affected by a following direct write to DBGDTR_EL0.
- Following an external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a following direct read of DBGDTRRX or DBGDTR_EL0 must be the value written by the previous external write.
- Following a first external write to DBGDTRTX and a following second external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a subsequent direct read of DBGDTR_EL0 must return the values written by the previous external writes.

H4.5.3 Synchronization of DCC interrupt request signals

Following an external read or external write access to the DTR registers, the interrupt request signals, **COMMIRQ**, **COMMTX**, and **COMMRX**, must be updated in finite time without explicit synchronization.

The updated values must be observable to a direct read of DCCSR or DBGDTRRX, or a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request. The updated values must also be observable to a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request.

Note

The requirement that indirect writes to registers are observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time. For more information, see Synchronization requirements for AArch64 System registers and Synchronization of changes to AArch32 System registers.

Following a direct read of DBGDTRRX or a direct write to DBGDTRRX, software must execute a Context Synchronization event to guarantee the interrupt request signals have been updated in finite time. This synchronization operation can be an exception return.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.5.4 DCC and ITR access in Debug state

In Debug state, stricter observability rules apply for instructions issued through the ITR, to maintain communication between a debugger and the PE, without requiring excessive explicit synchronization.

In Normal access mode, without explicit synchronization:

- A direct read or direct write of the DTR registers by an instruction written to EDITR must be observable to an external write or an external read in finite time:
  - A direct read of DBGDTRRX must be observable to an external write of DBGDTRRX_EL0.
  - A direct read of DBGDTR_EL0 must be observable to an external write of DBGDTRRX_EL0 and DBGDTRTX_EL0.
  - A direct write of DBGDTRTX must be observable to an external read of DBGDTRTX_EL0.
  - A direct write of DBGDTR_EL0 must be observable to an external read of DBGDTRRX_EL0 and DBGDTRTX_EL0.

  This includes the indirect write to the DCC flags that occurs atomically with the access as described in DCC accesses in Non-debug state.

  The subsequent external write or external read must observe either the old or the new values of both the DTR contents and DCC flags. If the old values are observed, this typically results in overrun or underrun, assuming the old values of the DCC flags indicate an overrun or underrun condition, as would normally be the case.

  This means the debugger can observe the direct read or direct write without explicit synchronization and without explicitly testing the DCC flags in EDSCR, because it can rely on overrun and underrun tests.
- External reads of DBGDTRTX_EL0 that do not underrun and external writes to DBGDTRRX_EL0 that do not overrun must be observable to an instruction subsequently written to EDITR on completion of the first external access. This includes the indirect write to the DCC flags.

  This means that without explicit synchronization and without the need to first check the DCC flags in DCCSR:
  - If the instruction is a direct read of DBGDTRRX, it observes the external write.
  - If the instruction is a direct write of DBGDTRTX, it observes the external read.
- Writes to EDITR that do not overrun commit an instruction for execution immediately. The instruction must complete execution in finite time without requiring any further operation by the debugger.
- After an external write to the EDITR, the ITR flags that are updated as a side effect of that write must be observable by:
  - An external read of the EDSCR that follows the external write to the EDITR.
  - When checking for overrun, another external write to the EDITR that follows the original external write to the EDITR.

In Memory access mode, these requirements shift to the instructions implicitly executed by external reads and external writes of the DTR registers, as described in Memory access mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.6 Interrupt-driven use of the DCC

Arm recommends implementations provide a level-sensitive DCC interrupt request through the IMPLEMENTATION DEFINED interrupt controller as a Private Peripheral Interrupt for the originating PE.

Note

- In addition to connection to the interrupt controller Arm also recommends **COMMIRQ**, **COMMTX**, and **COMMRX** signals that might be implemented for use by any legacy system peripherals.
- GICv3 reserves a Private Peripheral Interrupt number for the **COMMIRQ** interrupt.

The DCCINT register provides a first level of interrupt masking within the PE, meaning only a single interrupt source, **COMMIRQ**, is needed at the interrupt controller.

See also Synchronization of DCC interrupt request signals.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.7 Pseudocode description of the operation of the DCC and ITR registers

The basic operation of the DCC and ITR registers is shown by the following pseudocode functions. These functions do not cover the behavior when OSLSR.OSLK == 1, meaning that the OS Lock is locked:

- `Write_DBGDTR_EL0()`.
- `Write_DBGDTRRX_EL0()`.
- `Write_DBGDTRTX_EL0()`.
- `Write_EDITR()`.
- `CheckForDCCInterrupts()`.

For the definition of the DTR Registers, see shared/debug/dccanditr/DTR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H5
The Embedded Cross-Trigger Interface

This chapter describes the Embedded Cross-Trigger interface. It contains the following sections:

- About the Embedded Cross-Trigger.
- Basic operation on the ECT.
- Cross-triggers on a PE in an Arm A-profile implementation.
- Description and allocation of CTI triggers.
- CTI registers programmers' model.
- Examples.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.1 About the Embedded Cross-Trigger

The *Embedded Cross-Trigger* (ECT) allows a debugger to:

- Send trigger events to a PE. For example, this might be done to halt the PE.
- Send a trigger event to one or more PEs, or other system components, when a trigger event occurs on another PE or system component. For example, this might be done to halt all PEs when one individual PE halts.

Figure H5-1 shows the logical structure of an ECT.

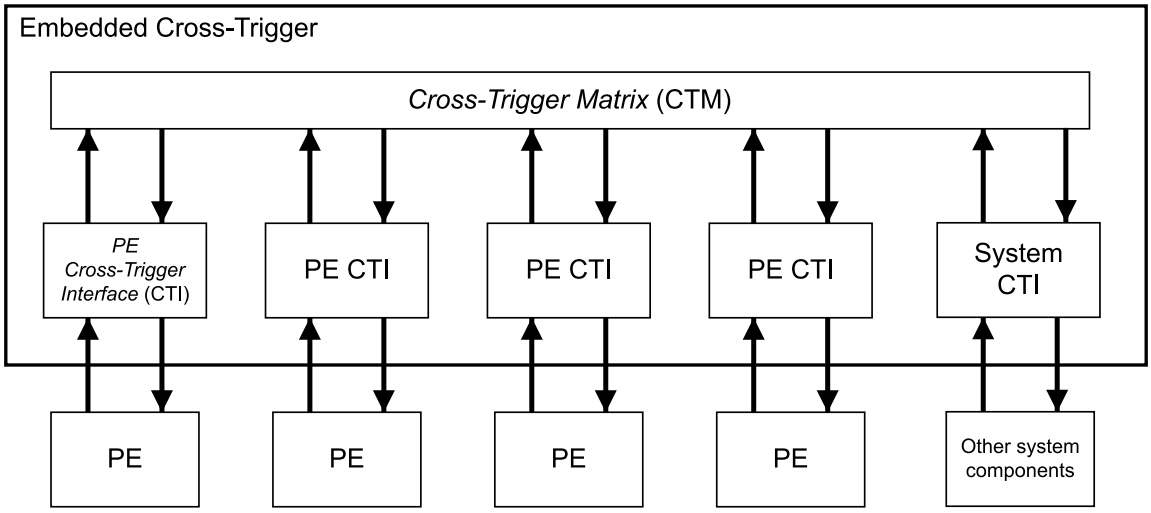

Figure H5-1 Structure of an Embedded Cross-Trigger

The ECT can deliver many types of trigger events, which are described in the following sections:

- Debug request trigger event.
- Restart request trigger event.
- Cross-halt trigger event.
- Performance Monitors overflow trigger event.
- Generic trace external input trigger events.
- Generic trace external output trigger events.
- Generic CTI interrupt trigger event.

From the introduction of Armv8, an implementation must:

- Include a Cross-Trigger interface, CTI.
- Implement at least the input and output triggers defined in this architecture.

In addition, see Cross-triggers on a PE in an Arm A-profile implementation.

Arm recommends that this Cross-Trigger interface includes:

- The ability to route trigger events between Trace Units, which typically have advanced event triggering logic.
- An output trigger to the interrupt controller.

Also, Arm recommends that the Embedded Cross-Trigger includes the capability to send and receive IMPLEMENTATION DEFINED system trigger events to and from other system components, including a system counter, using a system CTI. See Halt-on-debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The ECT and CTI must only signal trigger events for external debugging. They must not route software events, such as interrupts. For example, the Performance Monitors overflow input trigger is provided to allow entry to Debug state on a counter overflow, and the output trigger to the interrupt controller is provided to generally allow events from the external debug subsystem to be routed to a software agent. However, the combination of the two must not be used as a mechanism to route Performance Monitors overflows to an interrupt controller.

Note

CTI version 1 (CTIv1) is defined by the *CoreSight™ SoC Technical Reference Manual*. CTIv2 extends CTIv1 with the addition of the input channel gate. See Implementation with CTIv2.

H5.1.1 Implementation with a CoreSight CTI

For details of the recommended connections for an A-profile implementation, see Recommended External Debug Interface. See also *CoreSight™ SoC Technical Reference Manual.*

H5.1.2 Implementation with CTIv2

If the CTI implemented is CTIv2 then:

- The CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1 registers must be implemented.
- If the channel gate function is implemented, it applies to both input and output channels.
- The input channel gate function must be implemented if either of the following is true:
  - The CTM is implemented and the architecture variant is Armv8.5 or higher.
  - The CTIDEVARCH.REVISION field reads as `0b0001` or higher.

Implementation of CTIv2 features in architecture variants below Armv8.5 is OPTIONAL, but Arm recommends that CTIv2 is implemented, CTIv2 must be implemented from Armv8.5.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.2 Basic operation on the ECT

The ECT comprises a Cross-Trigger Matrix, CTM, and one Cross-Trigger Interface, CTI, for each PE. The ECT might also include other CTIs for other system components. The CTM passes events between the CTI blocks over channels. The CTM can have a maximum of 32 channels.

The main interfaces of the Cross-Trigger interface, CTI, are:

- The input triggers:
  - These are trigger event inputs from the PE to the CTI.
- The output triggers:
  - These are trigger event outputs from the CTI to the PE.
- The input channels:
  - These are channel event inputs from the Cross-Trigger matrix, CTM, to the CTI.
- The output channels:
  - These are channel event outputs from the CTI to the CTM.

Each CTI block has:

- Up to 32 input triggers that come from the PE:
  - The input triggers are numbered 0-31.
- Up to 32 output triggers that go to the PE:
  - The output triggers are numbered 0-31.

If the CTI is not powered up when the Core power domain is powered up, the CTI ignores all input triggers and input channel events, and does not generate any output triggers or output channel events.

Figure H5-2 shows the logical internal structure of a CTI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

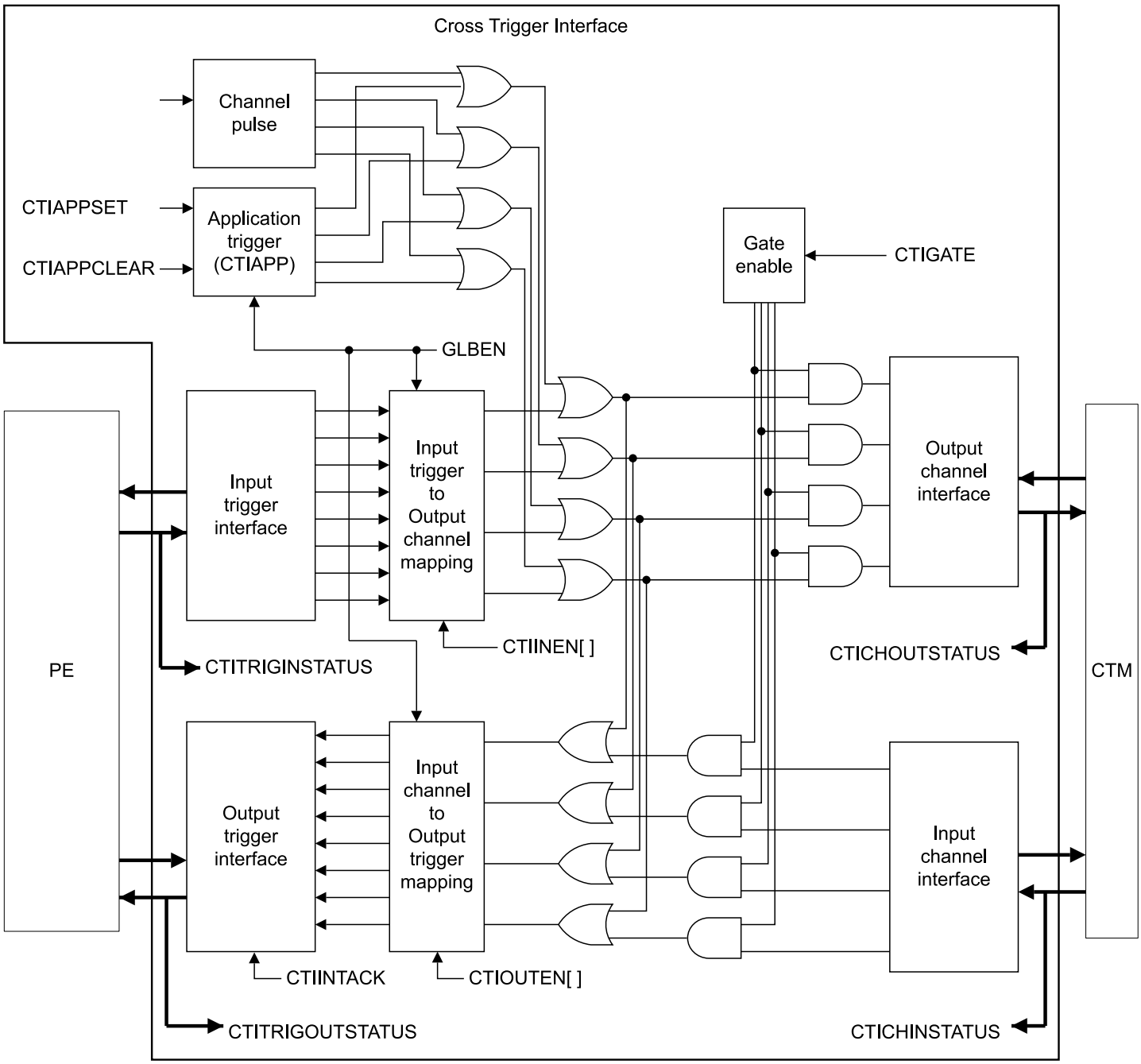

Figure H5-2 Structure of a Cross-Trigger interface

Note

- The number of triggers in IMPLEMENTATION DEFINED. Figure H5-2 shows eight input and eight output triggers.
- The number of channels is IMPLEMENTATION DEFINED. Figure H5-2 shows four channels.
- In Figure H5-2 the input channel gate function is a CTIv2 feature.

When the CTI receives an input trigger event, this generates channel events on one or more internal channels, according to the *mapping function* defined by the *Input trigger→output channel mapping registers*, CTIINEN<n>.

The CTI also contains an *application trigger* and *channel pulse* to allow a debugger to create channel events directly on internal channels by writing to the CTI control registers.

Channel events on each internal channel are passed to a corresponding output channel that is controlled by a *channel gate*. The channel gate can block propagation of channel events from an internal channel to an output channel.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If the CTM is implemented:

- The gate function must be implemented.
- If the CTI is CTIv1, the gate function applies to output triggers only.

The output channels from a CTI are combined, using a logical OR function, with the output channels from all other CTIs to form the input channels on other CTIs. The input channels of this CTI are the logical OR of the output channels on all other CTIs. This is the *Cross-Trigger Matrix*, CTM. Therefore, the number of input channels must equal the number of output channels.

Note

The number of input triggers and output triggers is not required to be the same.

The internal channels form an internal Cross-Trigger matrix within the CTI. This delivers events directly from the input triggers to the output triggers. Therefore the number of internal channels is the same as the number of input and output channels on the external CTM, and there is a direct mapping between the two.

Channel events received on each input channel are passed to the corresponding internal channel. It is IMPLEMENTATION DEFINED whether the Cross-Trigger gate also blocks propagation of channel events from input channels to internal channels.

Note

If CTIv2 is implemented, the Cross-Trigger gate also blocks propagation of channel events from input channels to internal channels.

When the CTI receives a channel event on an internal channel this generates trigger events on one or more output triggers, according to the mapping function defined by the *Input channel → output trigger mapping registers*, CTIOUTEN<n>.

The CTI contains the input and output trigger interfaces to the PE and the interface of the Cross-Trigger matrix. The architecture does not define the signal protocol used on the trigger interfaces, and:

- It is IMPLEMENTATION DEFINED whether the CTI supports multicycle input trigger events.
- It is IMPLEMENTATION DEFINED whether the CTM supports multicycle channel events.

See Multicycle events.

However, an output trigger is asserted until acknowledged. The output trigger can be:

- Self-acknowledging. This means that no further action is required from the debugger.
- Acknowledged by the debugger writing 1 to the corresponding bit of CTIINTACK.

The time taken to propagate a trigger event from the first PE, through its CTI, across the CTM to another CTI, and thereby to a second PE is IMPLEMENTATION DEFINED.

Note

Arm recommends that this path is not longer than the shortest software communication path between those PEs. This is because if the first PE halts, the Cross-halt trigger event can propagate through the ECT and halt the second PE without causing software on the second PE to malfunction because the first PE is in Debug state and is not responding.

H5.2.1 Multicycle events

A multicycle event is one with a continuous state that might persist over many cycles, as opposed to a discrete event. A typical implementation of a multicycle event is a level-based signal interface, whereas a discrete event might be implemented as a pulse signal or message.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CTI support for multicycle trigger events is IMPLEMENTATION DEFINED. Use of multicycle trigger events is deprecated. Of the architecturally defined input trigger events, the Performance Monitors overflow trigger event and Generic trace external output trigger events can be multicycle input triggers.

CTM support for multicycle channel events is IMPLEMENTATION DEFINED. A CTM that does not support multicycle channel events cannot propagate a multicycle trigger event between CTIs.

Note

A full ECT might comprise a mix of CTIs, some of which can support multicycle trigger events. In bridging these components, multicycle channel events become single channel events at the boundary between the CTIs.

H5.2.1.1 An ECT that supports multicycle trigger events

When an ECT supports multicycle trigger events, an input trigger event to the CTI continuously asserts channel events on all output channels mapped to it until either:

- The input trigger event is removed.
- The channel mapping function is disabled.

This means that an input trigger that is asserted for multiple cycles causes any channels that are mapped to it to become active for multiple cycles. Consequently, any output triggers mapped from that channel are asserted for multiple cycles.

Note

The output trigger remains asserted for at least as long as the channel remains active. This means that even if the output trigger is acknowledged, it remains asserted until the channel deactivates.

The CTI does not guarantee that these events have precisely the same duration, as the triggers and channels can cross between clock domains.

CTIAPPSET and CTIAPPCLEAR can set a channel active for multiple cycles. CTIAPPPULSE generates a single channel event. CTICHINSTATUS and CTICHOUTSTATUS can report whether a channel is active.

H5.2.1.2 An ECT that does not support multicycle trigger events

When an ECT does not support multicycle trigger events, an input trigger event to the CTI generates a single channel event on all output channels mapped to it, regardless of how long the input trigger event is asserted.

This means that an input trigger event that is asserted for multiple cycles generates a single channel event on any channels mapped to it. Consequently any self-acknowledging output triggers mapped from those channels are single trigger events.

Note

A single event is typically a single cycle, but there is no guarantee that this is always the case.

CTIAPPSET and CTIAPPCLEAR can only generate a single channel event. CTIAPPPULSE generates a single channel event. If the ECT does not support multicycle channel events, use of CTIAPPSET and CTIAPPCLEAR is deprecated, and the debugger must only use CTIAPPPULSE. CTICHINSTATUS and CTICHOUTSTATUS must be treated as UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.3 Cross-triggers on a PE in an Arm A-profile implementation

An A-Profile PE must include a Cross-Trigger interface, and the implementation must include at least the input and output triggers defined in this architecture. The number of channels in the Cross-Trigger matrix is IMPLEMENTATION DEFINED, but there must be a minimum of three. Software can read CTIDEVID.NUMCHAN to discover the number of implemented channels.

The CTM must connect to all PEs in the same Inner Shareability domain as the PE that includes the Cross-Trigger interface, but can also connect to additional PEs. Arm strongly recommends that the CTM connects all PEs implementing a CTI in the system. This includes other PEs that can be connected using a CoreSight CTI module.

Note

In a uniprocessor system the CTM is OPTIONAL. In a multiprocessor system the CTM is required. The CTM might be connected other CTI modules for non-PEs, such as triggers for system visibility components. Arm recommends that the CTM is implemented.

Any CTI connected to a PE that is not an Arm PE must implement at least:

- The Debug request trigger event.
- The Restart trigger event.
- The Cross-halt trigger event.

For more information about the CTI, see the *CoreSight™ SoC Technical Reference Manual*. Arm architecture refines the generic CTI by defining roles for each of the implemented input and output triggers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.4 Description and allocation of CTI triggers

Table H5-1 shows the output trigger events defined by the architecture and the related trigger numbers.

Table H5-1 Allocation of CTI output trigger events

Number	Source	Destination	Event description
0	CTI	PE	Debug request trigger event
1	CTI	PE	Restart request trigger event
2	CTI	IRQ controller	Generic CTI interrupt trigger event
3	-	-	Reserved
4 - 7	CTI	Trace unit	OPTIONAL Generic trace external input trigger events
8 - 15	-	-	If FEAT_TRBE_EXT is implemented, then Reserved. Otherwise IMPLEMENTATION DEFINED.

Note

Output triggers from the CTI are inputs to other blocks.

Table H5-2 shows the input trigger events defined by the architecture and the related trigger numbers.

Table H5-2 Allocation of CTI input trigger events

Number	Source	Destination	Event description
0	PE	CTI	Cross-halt trigger event
1	PE	CTI	Performance Monitors overflow trigger event
2	PE	CTI	Statistical Profiling Extension sample trigger event
3	-	-	Reserved
4 - 7	Trace unit	CTI	OPTIONAL Generic trace external output trigger events
8	PE TRBU	CTI	If FEAT_TRBE_EXT is implemented, Trace buffer stopped trigger event
9	PE TRBU	CTI	If FEAT_TRBE_EXT is implemented, Trace buffer management trigger event
10	PE TRBU	CTI	If FEAT_TRBE_EXT is implemented, Trace buffer wrap trigger event
11 - 15	-	-	If FEAT_TRBE_EXT is implemented, then Reserved. Otherwise IMPLEMENTATION DEFINED.

Note

Input triggers to the CTI are outputs from other blocks.

Table H5-1 and Table H5-2 show the minimum set of trigger events defined by the architecture. However:

- The Generic trace external input and output trigger events are required only if the OPTIONAL trace unit is implemented. If the OPTIONAL trace unit is not implemented, these trigger events are reserved.
- Support for the generic CTI interrupt trigger event is IMPLEMENTATION DEFINED because details of interrupt handling in the system, including any interrupt controllers, are IMPLEMENTATION DEFINED. Details regarding how the CTI interrupt is connected to an interrupt controller and its allocated interrupt number lie outside the scope of

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

the architecture. Arm strongly recommends that implementations provide a means to generate interrupts based on external debug events.

- If FEAT_TRBE_EXT is implemented, then CTI input triggers 8-15 and output triggers 11-15 are reserved for use by the architecture.

  If FEAT_TRBE_EXT is not implemented, then CTI input triggers 8-15 and output triggers 8-15 are IMPLEMENTATION DEFINED.
- The other trigger events are required by the architecture.

From the introduction of Armv8-A, an implementation can extend the CTI with additional triggers. When FEAT_TRBE_EXT is not implemented, these start with number eight. When FEAT_TRBE_EXT is implemented, these start with number 16.

H5.4.1 Debug request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to force the PE into Debug state. The trigger event is asserted until acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[0].

Note

A debugger must poll CTITRIGOUTSTATUS[0] until it reads as 0, to confirm that the output trigger has been deasserted before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

If the PE is already in Debug state, the PE ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger. See also External Debug Request debug event.

H5.4.2 Restart request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to request the PE to exit Debug state. If the PE is in Non-debug state, the request is ignored by the PE.

If a Restart request trigger event is received at or about the same time as the PE enters Debug state, it is CONSTRAINED UNPREDICTABLE whether:

- The request is ignored by the PE. In this case the PE enters Debug state and remains in Debug state.
- The PE enters Debug state and then immediately restarts.

Debuggers must program the CTI to send Restart request trigger events only to PEs that are halted. To enable the PE to disambiguate discrete Restart request trigger events, after sending a Restart request trigger event, the debugger must confirm that the PE has restarted and halted before sending another Restart request trigger event. Debuggers can use EDPRSR.{SDR, HALTED} to determine the Execution state of the PE.

Note

Before generating a Restart request trigger event for a PE, a debugger must ensure any Debug request trigger event targeting that PE is cleared. Debug request trigger event describes how to do this.

The trigger event is self-acknowledging, meaning that the debugger requires no further action to remove the trigger event. The trigger event is acknowledged even if the request is ignored by the PE. See also Exiting Debug state.

H5.4.3 Cross-halt trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted by a PE when it is entering Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

To reduce the latency of halting, Arm recommends that an implementation issues the Cross-halt trigger event early in the committed process of entering Debug state. This means that there is no requirement to wait until all aspects of entry to Debug state have completed before issuing the trigger event. Speculative emission of Cross-halt trigger events is not allowed. The Cross-halt trigger event must not be issued early enough for a subsequent Debug request trigger event, which might be derived from the Cross-halt trigger event, to be recorded in the EDSCR.STATUS field. This applies to Debug request trigger events that are acting as inputs to the PE.

H5.4.4 Performance Monitors overflow trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted each time the PE records a Performance Monitors counter overflow. See The Performance Monitors Extension.

If the CTI supports multicycle trigger events, then the trigger event remains asserted while any of the following are true, and deasserted otherwise:

- An event counter <n> is implemented, the global enable control for the event counter is 1, PMINTENSET.P[n] is 1, and PMOVSSET.P[n] is 1. The global enable control is defined in Enabling PMU counters.
- PMCR.E is 1, PMINTENSET.C is 1, and PMOVSSET.C is 1.
- FEAT_PMUv3_ICNTR is implemented, PMCR_EL0.E is 1, PMINTENSET_EL1.F0 is 1 and PMOVSSET_EL0.F0 is 1.

Note

- This does not replace the recommended connection of Performance Monitors overflow trigger event to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI.
- Events can be counted when `ExternalNoninvasiveDebugEnabled()` == FALSE, and, in Secure state, when `ExternalSecureNoninvasiveDebugEnabled()` == FALSE. Secure software must be aware that overflow trigger events are nevertheless visible to the CTI.
- This applies in both AArch64 and AArch32 states.

If the CTI does not support multicycle trigger events, then the trigger event is asserted whenever a counter overflows, causing the PE to set PMOVSSET.P[n] to 1, or a direct write to a System register means the above conditions become true.

For more information, see An ECT that supports multicycle trigger events.

H5.4.5 Statistical Profiling Extension sample trigger event

If the Statistical Profiling Extension is implemented, and a sample record is written to memory, CTI input trigger 2 is asserted. This trigger might also be directly connected to other IMPLEMENTATION DEFINED debug features.

For more information, see The Statistical Profiling Extension.

H5.4.6 Generic trace external input trigger events

These are output trigger events from the CTI, and input trigger events to the OPTIONAL trace unit, that are used in conjunction with the Generic trace external output trigger events to pass trigger events between:

- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external input trigger events.

The trigger events are self-acknowledging. This means that the debugger does not have to take any further action to remove the events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.4.7 Generic trace external output trigger events

These are input trigger events to the CTI, and output trigger events from the OPTIONAL trace unit, used in conjunction with the Generic trace external input trigger events to pass trigger events between:

- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external output trigger events.

H5.4.8 Generic CTI interrupt trigger event

This is an output trigger event from the CTI, and an input to an IMPLEMENTATION DEFINED interrupt controller, and can transfer trigger events from the PE, trace units, or any other component attached to the CTI and CTM to software as an interrupt. The Generic CTI interrupt trigger event must be connected to the interrupt controller as an interrupt that can target the originating PE.

Note

- Arm recommends that the Generic CTI interrupt trigger event is a private peripheral interrupt, but implementations might instead make this trigger event available as a shared peripheral interrupt or a local peripheral interrupt.
- GICv3 reserves a private peripheral interrupt number for this interrupt.

It is IMPLEMENTATION DEFINED whether this trigger event is:

- Self-acknowledging. This means that the debugger is not required to take any further action, and that the interrupt controller must treat the trigger event as a pulse or edge-sensitive interrupt.
- Acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[2]. This means that the interrupt controller must treat the trigger event as a level-sensitive interrupt.

Arm recommends that the Generic CTI interrupt trigger event is a self-acknowledging trigger event.

H5.4.9 Trace buffer stopped trigger event

If FEAT_TRBE_EXT is implemented, the Trace buffer stopped trigger event is defined.

The Trace buffer stopped trigger event is asserted when Collection is stopped. The trigger is asserted when an event causes the PE to set TRBSR_EL1.S to 1 and TRBSR_EL1.S was previously 0.

This is an output trigger from the Trace Buffer Unit, and an input trigger to the CTI.

H5.4.10 Trace buffer management trigger event

If FEAT_TRBE_EXT is implemented, the Trace buffer management trigger event is defined.

The Trace buffer management trigger event is asserted when a trace buffer management event occurs. The trigger is asserted when an event causes the PE to set TRBSR_EL1.IRQ to 1 and TRBSR_EL1.IRQ was previously 0.

This is an output trigger from the Trace Buffer Unit, and an input trigger to the CTI.

H5.4.11 Trace buffer wrap trigger event

If FEAT_TRBE_EXT is implemented, the Trace buffer wrap trigger event is defined.

The Trace buffer wrap trigger event is asserted when the current write pointer wrapped to the Base pointer. The trigger is asserted when an event causes the PE to set TRBSR_EL1.WRAP to 1, including when TRBSR_EL1.WRAP was previously 1. The trigger is asserted regardless of the trace buffer mode.

This is an output trigger from the Trace Buffer Unit, and an input trigger to the CTI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.5 CTI registers programmers' model

The CTI registers programmers' model is described in About the External Debug Registers. The following sections contain information specific to the CTI:

- External debug register resets.
- External debug interface register access permissions.
- Cross-trigger interface registers.
- The individual register descriptions in Cross-Trigger Interface registers.

See also Memory-mapped accesses to the external debug interface.

H5.5.1 CTI reset

An External Debug reset resets the CTI. See External debug register resets for details of CTI register resets. All CTI output triggers and output channels are deasserted on an External Debug reset.

Note

An indirect read of an output trigger might not observe the deasserted state until the processor is Cold reset. For more information, see Synchronization of changes to the external debug registers.

H5.5.2 CTI authentication

The CTI ignores the state of the IMPLEMENTATION DEFINED authentication interface. This means that:

- CTITRIGINSTATUS shows the status of the input triggers and CTICHINSTATUS shows the status of the input channels, regardless of the value of `ExternalNoninvasiveDebugEnabled()`.

  Note

  The PE does not generate the Cross-halt trigger event and the trace unit does not generate Generic trace external output trigger events when `ExternalNoninvasiveDebugEnabled()` == FALSE. However, the PE can generate Performance Monitors overflow trigger events.

- The CTI can generate external triggers regardless of the value of `ExternalInvasiveDebugEnabled()` and `ExternalNoninvasiveDebugEnabled()`.

  Note

  The destination of the trigger is responsible for authenticating the trigger request:

  — The PE ignores Debug request events when `ExternalInvasiveDebugEnabled()` is FALSE.

  — It is IMPLEMENTATION DEFINED whether the PE ignores Restart request trigger events when `ExternalInvasiveDebugEnabled()` is FALSE.

  — The trace unit ignores Generic trace external input trigger events when `ExternalNoninvasiveDebugEnabled()` is FALSE.

  — The behavior of Generic CTI interrupt requests is part of the IMPLEMENTATION DEFINED handling of these interrupts, but it is permissible for an interrupt controller to receive these requests even when `ExternalInvasiveDebugEnabled()` is FALSE, meaning software might need to configure the interrupt controller to ignore these requests.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H5.6 Examples

The CTI is fully programmable and allows for flexible cross-triggering of events within a PE and between PEs in a multiprocessor system. For example:

- The Cross-halt trigger event and the Debug request trigger event can be used for cross-triggering in a multiprocessor system.
- The Cross-halt trigger event and the Generic interrupt trigger event can be used for event-driven debugging in a multiprocessor system.
- The Performance Monitors overflow trigger event and the Debug request trigger event can force entry to Debug state on overflow of a Performance Monitors event counter, for event-driven profiling.

  Note

  This does not replace the recommended connection of Performance Monitors overflow trigger events to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI. Arm recommends that the Performance Monitors overflow signal is directly available as a local interrupt source.

- The Generic trace external input and Generic trace external output trigger events can pass trace events into and out of the event logic of the trace unit. They can do this:
  - To pass trace events between Trace Units.
  - In conjunction with the Performance Monitors overflow trigger event, to couple the Performance Monitors to the PE trace unit.
  - In conjunction with the Debug request trigger event, to trigger entry to Debug state on a trace event.
  - In conjunction with other CTIs, to signal a trace trigger event onto a CoreSight trace interconnect.

The following sections describe some examples in more detail:

- Example H5-1.
- Example H5-2.
- Example H5-3.
- Example H5-4.

Example H5-1 Halting a single PE

To halt a single PE, set:

1. CTIGATE[0] to 0, so that the CTI does not pass channel events on internal channel 0 to the CTM.
2. CTIOUTEN0[0] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 0.

   Note

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instance of CTIOUTEN<n> for which <n> is 0.

3. CTIAPPPULSE[0] to 1, to generate a channel event on channel 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example H5-2 Halting all PEs in a group when any one PE halts

To program a group of PEs so that when one PE in the group halts, all of the PEs in that group halt, set the following registers for each PE in the group:

1. CTIGATE[2] to 1, so that each CTI passes channel events on internal channel 2 to the CTM.
2. CTIINEN0[2] to 1, so that each CTI generates a channel event on channel 2 in response to a Cross-halt trigger event.
3. CTIOUTEN0[2] to 1, so that each CTI generates a Debug request trigger event in response to a channel event on channel 2.

   Note

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instances of CTIINEN<n> and CTIOUTEN<n> for which <n> is 0.

When a PE has halted, clear the Debug request trigger event by writing a value of 1to CTIINTACK[0], before restarting the PE.

Example H5-3 Synchronously restarting a group of PEs

To restart a group of PEs, for each PE in the group:

1. If the PE was halted because of a Debug request trigger event, the debugger must ensure the trigger event is deasserted. It can do this by:
   a. Writing 1 to CTIINTACK[0] to clear the Debug request trigger event.
   b. Polling CTITRIGOUTSTATUS[0], until it reads as 0, to confirm that the trigger event has been deasserted.
2. Set CTIGATE[1] to 1, so that each CTI passes channel events on internal channel 1 to the CTM.
3. Set CTIOUTEN1[1] to 1, so that each CTI generates a Restart request trigger event in response to a channel event on channel 1.

   Note

   This example must use the instance of CTIOUTEN<n> for which <n> is 1.

4. Set CTIAPPPULSE[1] to 1 on any one PE in the group, to generate a channel event on channel 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example H5-4 Halting a single PE on Performance Monitors overflow

To halt a single PE on a Performance Monitors overflow set:

1. CTIGATE[3] to 0, so that the CTI does not pass channel events on internal channel 3 to the CTM.
2. CTIINEN1[3] to 1, so that the CTI generates a channel event on channel 3 in response to a Performance Monitors overflow trigger event.

   Note

   This step of this example must use the instance of CTIINEN<n> for which <n> is 1.

3. CTIOUTEN0[3] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 3.

   Note

   This step of this example must use the instance of CTIOUTEN<n> for which <n> is 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE. Clear the overflow status by writing to PMOVSCLR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H6
Debug Reset and Powerdown Support

This chapter describes the reset and powerdown support in the Debug architecture. It contains the following sections:

- About Debug over powerdown.
- Power domains and debug.
- Core power domain power states.
- Emulating low-power states.
- Powerup request mechanism.
- Debug OS Save and Restore sequences.
- Reset and debug.

Note

Where necessary, Table K14-1 disambiguates the general register references used in this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.1 About Debug over powerdown

Arm A-profile external debug defines a logical model for the hardware on which a PE executes. This hardware is logically split into the *Core power domain* and the *Debug power domain*, and the model contains descriptions of the states of those domains. See:

- Power domains and debug.
- Core power domain power states.

An implementation may allow power domains to be powered up and down independently. Debug over powerdown provides:

- A facility for software executing on the PE to save and restore the PE state on behalf of a self-hosted or external debugger or both. See Debug OS Save and Restore sequences.
- A facility for an external debugger to request power up of the Core power domain. See Powerup request mechanism.
- A facility for an external debugger, or software executing on the PE, to request emulation of powerdown of the Core power domain. See Emulating low-power states.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.2 Power domains and debug

Arm A-profile external debug has two logical power domains, each with its own reset:

- The Debug power domain contains the interface between the PE and the external debugger, and is powered up whenever an external debugger is connected to the SoC. It remains powered up while the external debugger is connected. When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a register that is implemented in the Debug power domain. Registers in this domain are reset by an External Debug reset.
- The Core power domain contains the rest of the PE, and might be allowed to power up and power down independently of the Debug power domain.

Note

- The model of two logical power domains has an impact on the reset and access permission requirements of the debug programmers' model.
- The power domains are described as logical because the architecture defines the requirements but does not require two physical power domains. Any power domain split that meets the requirements of the programmers' model is a valid implementation.

The Core power domain contains several types of registers:

- Non-debug logic refers to all registers and logic that are not associated with debug.
- Self-hosted debug logic refers to registers and logic associated solely with the self-hosted debug aspects of the architecture.
- Shared debug logic refers to registers and logic associated with both the self-hosted and external debug aspects of the architecture.
- External debug logic refers to registers and logic associated solely with the external debug aspects of the architecture.

For information about which groups of registers and components are in each power domain, and which registers change power domain if FEAT_DoPD is implemented, see:

- Access permissions for the External debug interface registers.
- Cross-trigger interface registers.
- Management register access permissions.
- Access permissions for external views of the Performance Monitors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.3 Core power domain power states

The Arm architecture does not define the power states of the PE as these are not normally visible to software. However, they are visible to the external debugger. Arm A-profile external debug uses a four logical power states model for the Core power domain. Mechanisms for entering a low-power state describes the architectural mechanisms for entering low-power states.

When the PE enters a low-power state other than Powerdown by executing a `WFI`, `WFIT`, `WFE`, or `WFET` instruction, it remains in that low-power state until it receives a wakeup event. See Mechanisms for entering a low-power state for the definition of wakeup events. When halting is allowed, an External Debug Request is a wakeup event. In addition, if the PE enters the Standby state for any reason, it will leave that state to service an External Debug Request debug event.

The four logical power states are as follows:

Normal The Core power domain is fully powered up and the debug registers are accessible.

Standby The Core power domain is on, but there are measures to reduce energy consumption. There can be other IMPLEMENTATION DEFINED measures the OS can take to enter standby.

The PE preserves the PE state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the PE.

Standby is the least invasive OS energy saving state. Standby implies only that the PE is unavailable and does not clear any debug settings. For standby, the Debug architecture requires only the following:

- If the external debug interface is accessed, the PE must respond to that access. Arm recommends that, if the PE executed a `WFI` or `WFE` instruction to enter standby, then it does not retire that instruction.

Note

When FEAT_WFxT is implemented, this also applies to the `WFET` and `WFIT` instructions.

Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

Retention The PE state, including debug settings, is preserved in low-power structures, allowing the Core power domain to be at least partially turned off.

Changing from low-power retention to normal operation does not involve a reset of the PE. The saved PE state is restored on changing from low-power retention state to normal operation. If software has to use an IMPLEMENTATION DEFINED code sequence before entering, or after leaving, a retention state, this is referred to as a *software-visible retention state*. It is IMPLEMENTATION DEFINED whether the value of DBGPRCR.CORENPDRQ is set to its Cold reset value on leaving the software-visible retention state. See the description of DBGPRCR.CORENPDRQ for more information.

Note

- This model of retention does not include implementations where the PE exits the state in response to a debug register access. From the Debug architecture perspective, implementations like this are forms of standby.

Powerdown These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown.

If FEAT_DoubleLock is implemented, it is used during powerdown.

Changing from powerdown to normal operation must include:

- A Cold reset of the PE after the power level has been restored.
- The OS restoring the saved PE state.

External Debug Request debug events stay pending and debug registers in the Core power domain cannot be accessed.

An implementation might support enabling and disabling threads, either dynamically or once at reset time. Threads that are disabled in this way must appear to the external debugger as either:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Powered off, meaning they are either:
  - In a powerdown state.
  - In a retention state.
- Held in reset state.

Arm A-profile external debug uses a simpler two states model for the Debug power domain. The two states are:

Off The Debug power domain is turned off.

On The Debug power domain is turned on.

The available power states, including the cross-product of Core power domain and Debug power domain power states is IMPLEMENTATION DEFINED. Implementations are not required to implement all of these states and might include additional states. These additional states must appear to the debugger as one of the logical power states defined by this model. The control of power states is IMPLEMENTATION DEFINED.

Note

As a result, it is IMPLEMENTATION DEFINED whether it is possible for the Debug power domain to be on when the Core power domain is off.

If the Debug power domain is implemented but is not powered up when the Core power domain is powered up, the Reset Catch debug event and the OS Unlock Catch debug event are disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.4 Powerup request mechanism

If a powerup request mechanism is implemented, asserting the powerup request requests the power controller to power up the Core power domain, and to emulate any subsequent powerdown requests, until the powerup request mechanism is deasserted.

H6.4.1 Powerup request mechanism if FEAT_DoPD is implemented

If FEAT_DoPD is implemented, the external debug component implements an OPTIONAL powerup request mechanism.

If the powerup request mechanism is implemented, Arm strongly recommends that the powerup request is a CoreSight Class `0x9` ROM table block that contains both:

- A parent entry for the debug registers of the PE.
- A parent entry for the PMU registers of the PE, if the OPTIONAL PMU with an external debug interface is implemented.

A parent entry of a component is an entry in a ROM table that either locates the component, or locates another ROM table that contains the parent entry for the component.

Note

The ROM table and any descendants might describe other debug components, including debug components for other PEs. The ROM table might have a parent entry in a second ROM table and that parent entry might also have a powerup request mechanism in the second ROM table. This applies recursively.

The parent entries for the debug components have the following properties:

For the debug registers and Performance Monitors registers: These components are in the Core power domain.

The POWERIDVALID bit is 1.

All parent entries must have the same IMPLEMENTATION DEFINED POWERID value.

Note

The IMPLEMENTATION DEFINED POWERID value does not need to be unique for each PE.

For the CTI registers: This component is in the Debug power domain.

The POWERIDVALID bit is IMPLEMENTATION DEFINED.

If the POWERIDVALID bit is 1, the entries must have a valid POWERID value.

Note

If the Core power domain can be powered down independently of the Debug power domain, Arm recommends the system implements an external debug component with a powerup request mechanism which can request the Core power domain to be powered up.

For more information about Coresight Class `0x9` ROM Tables, see *ARM® CoreSight™ Architecture Specification*.

On a Cold reset, if FEAT_DoPD is implemented, DBGPRCR.CORENPDRQ is set to an IMPLEMENTATION DEFINED choice between 0 and 1 if the powerup request is implemented and asserted, and 0 otherwise.

H6.4.2 Powerup request mechanism if FEAT_DoPD is not implemented

If FEAT_DoPD is not implemented, the bit EDPRCR.COREPURQ is the powerup request mechanism.

The control registers DBGPRCR.CORENPDRQ and EDPRCR.COREPURQ provide an interface between the power controller and the PE. They typically map directly to signals in the recommended external debug interface.

On Cold reset, if FEAT_DoPD is not implemented, DBGPRCR.CORENPDRQ is set to the value of EDPRCR.COREPURQ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.5 Emulating low-power states

DBGPRCR.CORENPDRQ and the powerup request mechanism can request the power controller to emulate states where the Core power domain is completely off or in a low-power state where the Core power domain registers cannot be accessed. This simplifies the requirements on software by sacrificing entirely realistic behavior.

If FEAT_DoPD is not implemented, EDPRSR.{SPD, PU} indicates the Core power domain power state. For more information, see:

- The DBGPRCR_EL1 and DBGPRCR System register descriptions.
- The EDPRCR and EDPRSR external debug register descriptions.
- Recommended External Debug Interface.

The measures to emulate powerdown are IMPLEMENTATION DEFINED. The ability of the debugger to access the state of the PE and the system might be limited as a result of the measures adopted.

In an emulated powerdown state, the debugger must be able to access all debug, PMU, CTI, and trace unit registers that are accessible on the external debug interface and are in one of:

- The Debug power domain.
- The Core power domain.
- When a trace unit with a separate trace unit Core power domain is implemented, and the trace unit Core power domain is powered on, the trace unit Core power domain.

That is, the debugger must be able to read and write to such registers without receiving errors. This allows an external debugger to debug the powerup sequence.

Arm recommends that any IMPLEMENTATION DEFINED registers that are on the external debug interface and in either the Core power domain or the Debug power domain are also accessible in an emulated powerdown state.

If FEAT_DoubleLock is implemented, `DoubleLockStatus() == FALSE` when DBGPRCR.CORENPDRQ == 1.

Otherwise, the behavior of the PE in emulated powerdown must be similar to that in a real powerdown state. In particular, the PE must not respond to other system stimuli, such as interrupts.

Example H6-1 and Example H6-2 are examples of two approaches to emulating powerdown.

Example H6-1 An example of emulating powerdown

The PE is held in Standby state, isolated from any system stimuli. It is IMPLEMENTATION DEFINED whether the PE can respond to debug stimuli such as an External Debug Request debug event.

If the PE can enter Debug state, then the external debugger is able to use the ITR to execute instructions, such as loads and stores. This causes the external debugger to interact with the system. If the external debugger restarts the PE, the PE leaves Standby state and restarts fetching instructions from memory.

Example H6-2 Another example of emulating powerdown

The PE is held in Warm reset. This limits the ability of an external debugger to access the resources of the PE. For example, the PE cannot be put into Debug state.

On exit from emulated powerdown the PE is reset. However, the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset. As such, the effect of accessing any register that is reset by a Warm reset while the PE is in the emulated powerdown state will have an IMPLEMENTATION DEFINED effect on that register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Warm reset and Cold reset have different effects apart from resetting the debug registers. In particular, RMR_ELx is reset by a Cold reset and controls the reset state on a Warm reset. This means that if a Cold reset is substituted by a Warm reset, the behavior of the reset code might be different.
- The timing effects of powering down are typically not factored in the powerdown emulation. Examples of these timing effects are clock and voltage stabilization.
- Emulation does not model the state lost during powerdown, meaning that it might mask errors in the state storage and recovery routines.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.6 Debug OS Save and Restore sequences

From the introduction of Armv8-A, the following registers provide the OS Save and Restore mechanism:

- The *OS Lock Access Register*, OSLSR, locks the OS Lock to restrict access to debug registers before starting an OS Save sequence, and unlocks the OS Lock after an OS Restore sequence.
- The *OS Lock Status Register*, OSLSR, shows the status of the OS Lock.
- The PE can be configured to generate an OS Unlock Catch debug event when the OS Lock is unlocked.
- If FEAT_DoubleLock is implemented, the OS Double Lock locks out an external debug interface entirely. This is only used immediately before a powerdown sequence.

See also:

- FEAT_DoubleLock
- Reset and debug
- Example OS Save and Restore sequences

H6.6.1 EDPRSR.{DLK, SPD, PU} and the Core power domain

If FEAT_DoPD is not implemented, a debugger uses EDPRSR.{DLK, SPD, PU} to determine whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read.

Table H6-1 Interpretation of the EDPRSR.{DLK, SPD, PU} bits

EDPRSR			Core power domain			Notes
DLK	SPD	PU	Power	Accesses	State lost	
0	0	1	On	OK	No	-
0	1	1	On	OK	Yes	SPD is cleared to 0 following the read.
1	X	1	On	Error	Not known	FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. Software locks the OS Double Lock before removing power.
X	1	0	Off	Error	Yes	A Cold reset will be asserted on exiting powerdown state, but not on exiting low-power retention state.
X	0	0	Not known	Error	Not known	

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When FEAT_Debugv8p4 is implemented, and whenever FEAT_DoubleLock is not implemented, EDPRSR.DLK is always 0.

If FEAT_DoubleLock is not implemented, `DoubleLockStatus()` always returns FALSE.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, then:

- When FEAT_Debugv8p2 is not implemented, EDPRSR.{DLK, SPD, PU} can read either {1, UNKNOWN, 1} or {UNKNOWN, 0, 0}.
- When FEAT_Debugv8p2 is implemented, and FEAT_Debugv8p4 is not implemented, EDPRSR.{DLK, SPD, PU} can only read {UNKNOWN, 0, 0}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.6.2 EDPRSR.SPD when the Core domain is in either retention or powerdown state

If FEAT_DoPD is not implemented, when the Core power domain is in either the retention or powerdown state, EDPRSR.SPD is not cleared following a read of EDPRSR and it is IMPLEMENTATION DEFINED whether:

- EDPRSR.SPD shows whether the state of the debug registers in the Core power domain has been lost since the last time that EDPRSR was read. This means that:
  - When the Core power domain is in the powerdown state, EDPRSR.SPD is RAO, this indicates that the state of the debug registers has been lost.
  - When the Core power domain is in the retention state, EDPRSR.SPD indicates whether the state of the debug registers was lost before the Core power domain entered retention state.
- EDPRSR.SPD is RAZ, and:
  - On leaving the powerdown state, EDPRSR.SPD is set to 1 which indicates that the state of the debug registers has been lost.
  - On leaving the retention state, EDPRSR.SPD reverts the value it had on entering the retention state.

Note

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state.

H6.6.3 EDPRSR.{DLK, R} and reset state

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When FEAT_Debugv8p4 is implemented, and whenever FEAT_DoubleLock is not implemented, EDPRSR.DLK is always 0.

If FEAT_DoubleLock is not implemented, `DoubleLockStatus()` always returns FALSE.

If FEAT_DoubleLock is implemented and enabled, the behavior of all registers and fields except EDPRSR.DLK is the same as their behavior if FEAT_Debugv8p4 is not implemented.

If FEAT_Debugv8p4 is implemented EDPRSR.DLK is always 0 and does not give any information about the OS Double Lock.

EDPRSR.R is UNKNOWN when `DoubleLockStatus()` == TRUE. OSDLR_EL1.DLK is cleared to 0 by a reset. If the Core power domain is powered up and entered reset state with the OS Double Lock locked, it is CONSTRAINED UNPREDICTABLE whether a read of EDPRSR while the PE is in reset state returns:

- EDPRSR.{DLK, R, PU} == {1, UNKNOWN, 1} indicating that the OS Double Lock is locked. This is not permitted from Armv8.2.
- EDPRSR.{DLK, R, PU} == {0, 1, 1} indicating that the PE is in reset state.
- EDPRSR.{DLK, R, PU} == {UNKNOWN, UNKNOWN, 0} indicating that the registers in the Core power domain cannot be accessed because the OS Double Lock is locked.

If the PE was powered up and the OS Double Lock was unlocked when the PE was reset, then EDPRSR.{DLK, R, PU} reads as {0, 1, 1} while the PE is in reset state.

On leaving reset state, EDPRSR.{DLK, R} reads as {0, 0}.

H6.6.4 Debug registers to save over powerdown

Table H6-2 shows the different requirements for self-hosted debug over powerdown and external debug over powerdown:

- The column labeled Self-hosted lists registers that software must preserve over powerdown so that it can support self-hosted debug over powerdown. This does not require use of the OS Save and Restore mechanism.
- The column labeled External lists registers that software must preserve over powerdown so that it can support external debug over powerdown. This requires use of the OS Save and Restore mechanism:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Some external debug registers are not normally accessible to software executing on the PE. Additional debug registers are provided that give software the required access to save and restore these external debug registers when OSLSR.OSLK is locked. These registers include OSECCR, OSDTRRX, and OSDTRTX.

- Some registers might only present in some implementations, or might not be accessible at all Exception levels or in Non-secure state. DBGVCR32_EL2 and SDER32_EL3 are only required to support AArch32.

Table H6-2 does not include registers for the OPTIONAL Trace and Performance Monitor extensions.

Table H6-2 Debug registers to save over powerdown

Register in AArch64 state	Register in AArch32 state	Self-hosted	External
MDSCR_EL1	DBGDSCRext	Yes	Yes
DBGBVR<n>_EL1	DBGBVR<n>	Yes	Yes
DBGBCR<n>_EL1	DBGBCR<n>	Yes	Yes
DBGWVR<n>_EL1	DBGWVR<n>	Yes	Yes
DBGWCR<n>_EL1	DBGWCR<n>	Yes	Yes
DBGVCR32_EL2	DBGVCR	Yes	-
MDCR_EL2	HDCR	Yes	-
SDER32_EL3	SDER	Yes	-
MDCR_EL3	SDCR	Yes	-
MDCCINT_EL1	DBGDCCINT	-	Yes
DBGCLAIMSET_EL1 DBGCLAIMCLR_EL1	DBGCLAIMSET, DBGCLAIMCLR	-	Yes
OSECCR_EL1	DBGOSECCR	-	Yes
OSDTRRX_EL1 OSDTRTX_EL1	DBGDTRRXext DBGDTRTXext	-	Yes

H6.6.5 OS Save sequence

To preserve the debug logic state over a powerdown, the state must be saved to nonvolatile storage. This means the OS Save sequence must:

1. Lock the OS Lock by:
   - Writing the key value `0xC5ACCE55` to the DBGOSLAR in AArch32 state.
   - Writing 1 to OSLAR_EL1.OSLK in AArch64 state.
2. Execute an `ISB` instruction.
3. Walk through the debug registers listed in Debug registers to save over powerdown and save the values to the nonvolatile storage.

If the FEAT_DoubleLock is implemented, before removing power from the Core power domain, software must:

1. Lock the OS Double Lock by:
   - Writing 1 to DBGOSDLR.DLK in AArch32 state.
   - Writing 1 to OSDLR_EL1.DLK in AArch64 state.

   If FEAT_DoubleLock is not implemented, OSDLR_EL1 and DBGOSDLR ignore writes.
2. Execute a Context Synchronization event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.6.6 OS Restore sequence

After a powerdown, the OS Restore sequence must perform the following steps to restore the debug logic state from the non-volatile storage:

1. Lock the OS Lock, as described in OS Save sequence. The OS Lock is generally locked by the Cold reset, but this step ensures that it is locked.
2. Execute an `ISB` instruction.
3. To ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated:
   - Write 0 to MDSCR_EL1 if executing in AArch64 state.
   - Write 0 to DBGDSCRext if executing in AArch32 state.
4. Walk through the debug registers listed in Debug registers to save over powerdown, and restore the values from the nonvolatile storage. The last register to be restored must be:
   - MDSCR_EL1 if executing in AArch64 state.
   - DBGDSCRext if executing in AArch32 state.
5. Execute an `ISB` instruction.
6. Unlock the OS Lock by:
   - Writing 0 to OSLAR_EL1.OSLK if executing in AArch64 state.
   - Writing any non-key value to DBGOSLAR if executing in AArch32 state.
7. Execute a Context Synchronization event.

Note

The OS Restore sequence overwrites the debug registers with the values that were saved. If there are valid values in these registers immediately before the restore sequence, then those values are lost.

H6.6.7 Debug behavior when the OS Lock is locked

The main purpose of the OS Lock is to prevent updates to debug registers during an OS Save or OS Restore operation. The OS Lock is locked on a Cold reset.

When the OS Lock is locked:

- Access to debug System registers through the System register interface is mainly unchanged except that:
  - Certain registers are read and written without side-effects.
  - Fields in DSCR and OSECCR that are normally read-only become read/write.

  This allows the state to be saved or restored. For more information, see the relevant register description in External Debug Register Descriptions.
- Access to debug registers by the external debug interface is restricted to prevent an external debugger modifying the registers that are being saved or restored. For more information, see External debug interface register access permissions summary.
- Debug exceptions, other than Breakpoint Instruction exceptions are not generated.
- Breakpoint and Watchpoint debug events are not generated. The OS Lock has no effect on Breakpoint Instruction exceptions and other debug events.

H6.6.8 Debug behavior when the OS Lock is unlocked

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. See OS Unlock Catch debug event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.6.9 Debug behavior when the OS Double Lock is locked

If the FEAT_DoubleLock is implemented, software locks the OS Double Lock immediately before a powerdown sequence.

The OS Double Lock ensures that it is safe to remove core power by forcing the debug interfaces to be quiescent.

When `DoubleLockStatus()` == TRUE:

- The external debug interface has only restricted access to the debug registers, so that it is quiescent before removing power. See External debug interface register access permissions summary.
- Debug exceptions, other than Breakpoint Instruction exceptions, are not generated.
- Halting is prohibited. See Halting allowed and halting prohibited.

  Note

  Pending Halting debug events might be lost when core power is removed.

- No asynchronous debug events are `WF*` wakeup events.

If the FEAT_DoubleLock is not implemented, the PE ensures these conditions are met before allowing power to be removed.

Software must synchronize the update to OSDLR before it indicates to the system that core power can be removed. The interface between the PE and its power controller is IMPLEMENTATION DEFINED.

Typically software indicates that core power can be removed by entering the Wait For Interrupt state. This means that software must explicitly synchronize the OSDLR update before issuing the `WFI` instruction.

OSDLR.DLK is ignored and `DoubleLockStatus()` == FALSE if either:

- The PE is in Debug state.
- DBGPRCR.CORENPDRQ is set to 1.

Note

It is possible to enter Debug state with OSDLR.DLK set to 1. This is because a Context Synchronization event is required to ensure the OS Double Lock is locked, meaning that Debug state might be entered before the OSDLR update is synchronized.

Because OSDLR.DLK is ignored when DBGPRCR.CORENPDRQ is set to 1, an external debugger can write to DBGPRCR.CORENPDRQ, and the FEAT_DoubleLock is not always implemented, software must not rely on using the OS Double Lock to disable debug exceptions or to prohibit halting, or both. Arm deprecates use of the OS Double Lock for these purposes, and instead recommends that software:

- Uses the OS Lock to disable debug exceptions during save or restore sequences.
- Uses the debug authentication interface to prohibit halting and external debug access to debug registers at times other than immediately prior to removing power.

As the purpose of the OS Double Lock is to ensure that it is safe to remove core power, if the FEAT_DoubleLock is implemented, it is important to avoid race conditions that defeat this purpose. Arm recommends that:

- Once the write to OSDLR.DLK has been synchronized by a Context Synchronization event and `DoubleLockStatus()` == TRUE, a PE must:
  - Not allow a debug event generated before the Context Synchronization event to cause an entry to Debug state or act as a wakeup event for a `WFI` or `WFE` instruction after the Context Synchronization event has completed.

    Note

    When FEAT_WFxT is implemented, this also applies to the `WFET` and `WFIT` instructions.

  - Complete any external debug access started before the Context Synchronization event by the time the Context Synchronization event completes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A debug register access might be in progress when software sets OSDLR.DLK to 1. An implementation must not permit the synchronization of locking the OS Double Lock to stall indefinitely while waiting for that access to complete. This means that any debug register access that is in progress when software sets OSDLR.DLK to 1 must complete or return an error in finite time.

- If a write to DBGPRCR or EDPRCR made when OSDLR.DLK == 1 changes DBGPRCR.CORENPDRQ or EDPRCR.CORENPDRQ from 1 to 0, meaning `DoubleLockStatus()` changes from FALSE to TRUE, then before signaling to the system that the CORENPDRQ field has been cleared and emulation of powerdown is no longer requested, meaning the system can remove core power, the PE must ensure that all the requirements for `DoubleLockStatus()` == TRUE listed in this section are met.

In a standard OS Save sequence, the OS Lock is locked before the OS Double Lock is locked. This means that writes to CORENPDRQ are ignored by the time the OS Double Lock is locked. However, if `DoubleLockStatus()` == FALSE, an external debugger can clear the OS Lock at any time, and then write to EDPRCR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H6.7 Reset and debug

All registers in the Core power domain are either:

- Reset by both a Cold and a Warm reset.
- Reset only by a Cold reset and are not changed by a Warm reset.

For more information, see Resets and power domains.

All registers in the Debug power domain are reset by an External Debug reset.

If FEAT_SPMU is implemented, the reset signals for a System PMU are IMPLEMENTATION DEFINED.

Figure H6-1 shows this reset scheme. The following three reset signals are an example implementation of the reset scheme:

- **CORERESET**, which must be asserted for a Warm reset.
- **CPUPORESET**, which must be asserted for a Cold reset.
- **PRESETDBG**, which must be asserted for an External Debug reset.

As shown in the figure, the external debug logic is split between the Debug power domain and the Core power domain.

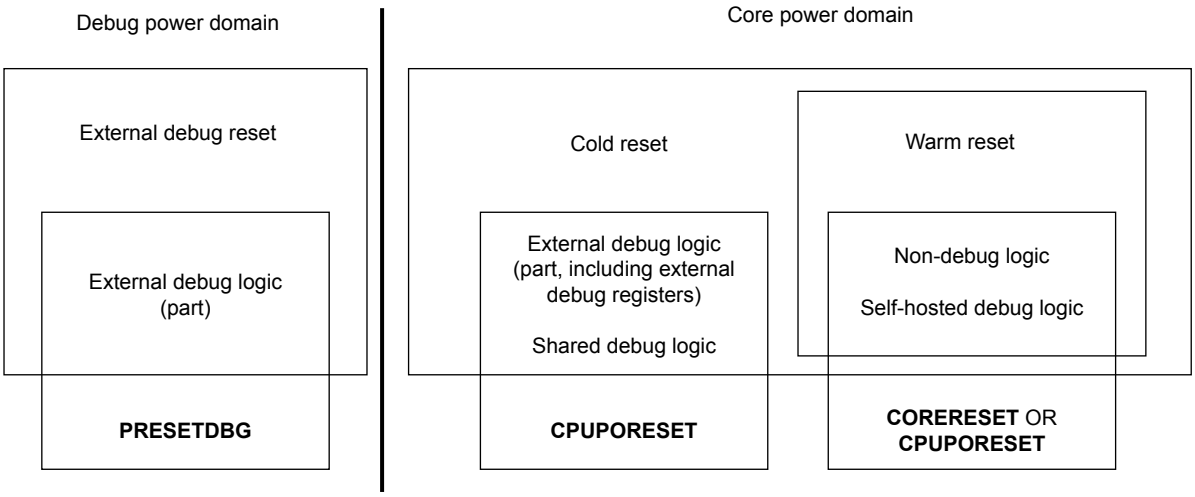

Figure H6-1 Power and reset domains

For more information about power domains and power states, see Power domains and debug.

When power is first applied to the Debug power domain, **PRESETDBG** must be asserted.

When power is first applied to the Core power domain, **CPUPORESET** must be asserted.

Note

In this scheme, logic in the Warm reset domain is reset by asserting either **CORERESET** or **CPUPORESET**. This implies a particular implementation style that permits these approaches.

CPUPORESET is not normally asserted on moving from a low-power state, where power has not been removed, to a full-power state. This can occur, for example, on exiting a low-power retention state. See also Emulating low-power states and the EDPRSR register description.

H6.7.1 External debug interface accesses to registers in reset

If a reset signal is asserted and the external debug interface:

- Writes a register, or indirectly writes a register or register field as a side-effect of an access:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Then, if the register or register field is reset by that reset signal, it is CONSTRAINED UNPREDICTABLE whether the register or register field takes the reset value or the value written. The reset value might be UNKNOWN.
  - Otherwise, the register or register field takes the value that is written.
- Reads a register, or indirectly reads a register or register field, as part of an access:
  - Then, if the register or register field is reset by that reset signal, the value returned in UNKNOWN.
  - Otherwise, the value of the register or register field is returned.

It is IMPLEMENTATION DEFINED whether any register can be accessed when External Debug reset is being asserted. The result of these accesses is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H7
The PC Sample-based Profiling Extension

This chapter describes the OPTIONAL PC Sample-based Profiling Extension that provides a non-invasive external debug component.

It contains the following section:

- About the PC Sample-based Profiling Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H7.1 About the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is an OPTIONAL extension that provides coarse-grained, non-invasive profiling by an external debugger. See also Non-invasive behavior.

When the PC Sample-based Profiling Extension is implemented, the PE implements a PC sample register and one or more context sample registers as external registers. Reads of the PC sample register return a PC sample value and capture the PE context in the context sample registers. For example, tools can use these samples to populate a statistical model of the performance of software executing on the PE.

Note

Data returned by periodic sampling of PC Sample-based Profiling registers is sufficient to allow tools to estimate the distribution of time spent executing software on the PE.
The delay between an instruction being executed by the PE and its address appearing in the PC Sample Register is not defined, and the architecture does not require that the sampled instruction was recently executed. For example, if a piece of software executes a load instruction that reads the PC Sample Register of the PE it is running on, there is no guaranteed relationship between the address of the load instruction and the value read. The PC Sample Register is intended only for use by an external agent to provide statistical information for software profiling.

It must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. The branch target for a conditional branch instruction that fails its condition check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

FEAT_PCSRv8p9 adds a mechanism to suspend PC Sample-based Profiling in systems where:

- It is required to be able to use PC Sample-based Profiling on an *ad hoc* basis.
- It is required to be able to disable or suspend PC Sample-based Profiling after use.
- It is not acceptable to reset or power down the PE in order to disable or suspend PC Sample-based Profiling.

To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the sampled data is acceptable. Arm does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:

- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the sample to represent an instruction that was not committed for execution.
- Under unusual non-repeating pathological cases, the sample can represent an instruction that was not committed for execution. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in sampling is very unlikely.
- Under normal operating conditions, the sample must reference an instruction that was committed for execution, including its context, and must not reference instructions that are fetched but not committed for execution.

Note

In the Armv7 PC Sample-based Profiling Extension, an offset was applied to the sampled Program Counter value and this offset and the instruction set state indicated in bits [1:0] of the sampled value. From the introduction of the Armv8 PC Sample-based Profiling Extension, the sampled value is the address of an instruction that has executed, with no offset and no indication of the instruction set state.

- Controlling the PC Sample-based Profiling Extension.
- Registers implemented by the PC Sample-based Profiling Extension.
- Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN.
- Pseudocode description of PC Sample-based Profiling.

H7.1.1 Controlling the PC Sample-based Profiling Extension

PC Sample-based Profiling is controlled by the IMPLEMENTATION DEFINED authentication interface `ExternalNoninvasiveDebugEnabled()`.

PC Sample-based Profiling is allowed if both of the following apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `ExternalNoninvasiveDebugEnabled()` == TRUE.
- At least one of the following applies:
  - The PE is executing in Non-secure state.
  - EL3 is not implemented.
  - EL3 is implemented, the PE is executing in Secure state, and `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.
  - The PE is executing in Realm state and `ExternalRealmNoninvasiveDebugEnabled()` == TRUE.
  - The PE is executing in Root state and `ExternalRootNoninvasiveDebugEnabled()` == TRUE.
  - EL3 or EL1 is using AArch32, the PE is executing at EL0 in Secure state, and the value of SDER.SUNIDEN is 1.

The state of the IMPLEMENTATION DEFINED authentication interface is visible through DBGAUTHSTATUS_EL1. See Recommended authentication interface.

H7.1.1.1 Suspending and activating PC Sample-based Profiling

If FEAT_PCSRv8p9 is implemented and the value of PMPCSCTL.IMP is 1, then all of the following apply:

- When PMPCSCTL.EN is 0, PC Sample-based Profiling is *suspended* unless otherwise stated.
- When PMPCSCTL.EN is 1, PC Sample-based Profiling is *active*.

Otherwise, all of the following apply:

- If FEAT_PCSRv8p9 is not implemented, then PMPCSCTL.EN is RES0.
- The state of PC Sample-based Profiling at Warm reset is an IMPLEMENTATION DEFINED one of suspended or active.

When all of the following apply, reading PMPCSR[31:0] or PMPCSR[63:0] makes PC Sample-based Profiling active:

- PC Sample-based Profiling is suspended.
- PC sampling is allowed.
- Sampling on reads is enabled.

A subsequent external read of PMPCSR will observe the updated state.

This means that if FEAT_PCSRv8p9 is implemented and PMPCSCTL.IMP reads as 1, then PMPCSCTL.EN is set to 1 as a side-effect of the read of PMPCSR[31:0] or PMPCSR[63:0].

Note

- A read of PMPCSR[63:32] in isolation does not make PC Sample-based Profiling active, and so does not have the side-effect of setting PMPCSCTL.EN to 1.
- If FEAT_PCSRv8p9 is implemented and PMPCSCTL.IMP reads as 0, then there is no mechanism to return to suspended state, other than Warm reset.

When FEAT_PMUv3_SS is implemented and PMPCSCTL.SS is 1, all the following apply:

- PC Sample-based Profiling is active, regardless of the value of PMPCSCTL.EN, if implemented.
- Sampling on PMU snapshot Capture events is enabled.
- Sampling on reads is disabled.

Otherwise, sampling on PMU snapshot Capture events is disabled, and sampling on reads is enabled.

When sampling on reads is disabled, all of the following apply to reads of PMPCSR:

- The reads have none of the side-effects described in PMPCSR.
- Return the value of the PC when it was last sampled, or the Cold reset value of the register if the PC has not been sampled since the last Cold reset.

For more information on the operation of Capture events, see PMU snapshots.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H7.1.2 Registers implemented by the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is implemented by either FEAT_PCSRv8 or FEAT_PCSRv8p2:

- If FEAT_PCSRv8 is implemented then the PC Sample-based Profiling Extension registers are implemented in the external debug register space. EDDEVID1.PCSample identifies when FEAT_PCSRv8 is implemented and which PC Sample-based Profiling Extension registers are implemented.
- If FEAT_PCSRv8p2 is implemented then the PC Sample-based Profiling Extension registers are implemented in the Performance Monitors memory-mapped register space. PMDEVID.PCSample identifies when FEAT_PCSRv8p2 is implemented.

An implementation is not permitted to include both FEAT_PCSRv8 and FEAT_PCSRv8p2. FEAT_PCSRv8 is not permitted from Armv8.2.

If FEAT_PCSRv8 is implemented:

- The following external debug registers can be implemented:
  - EDCIDSR.
  - EDPCSR.
  - EDVIDSR.

  See External debug interface registers.
- If FEAT_VHE is implemented, EDSCR.SC2 controls what PC Sample-based Profiling samples.

If FEAT_PCSRv8p2 is implemented and the 64-bit external PMU programmers' model extension is not implemented, the following PC Sample-based Profiling Extension registers can be implemented in the Performance Monitors memory-mapped register space:

- PMCID1SR and PMCID2SR.
- PMPCSR.
- PMVIDSR.

When the 64-bit external PMU programmers' model extension is implemented, the PC Sample-based Profiling Extension registers are:

- PMCCIDSR.
- PMPCSR.
- PMVCIDSR.

See Performance Monitors external register views.

If FEAT_PCSRv8p2 is implemented but the Performance Monitors Extension is not implemented, then the PC Sample-based Profiling Extension registers are implemented in their own memory-mapped register space, within the area that is reserved for the Performance Monitors, see Table H7-1. If CoreSight compliance is required:

- The management registers are defined as in Table K5-4.
- The support for PC Sample-based profiling is defined in the following registers:
  - PMDEVTYPE.MAJOR has the value `0x0`.
  - PMDEVARCH.ARCHID has the value `0x0A10` or `0x0A20`.

For more information, see PMDEVARCH.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H7-1 PC Sample-based Profiling register map without the Performance Monitors Extension

Offset	Description
`0x200`	PMPCSR[31:0]
`0x204`	PMPCSR[63:32]
`0x208`	PMCID1SR
`0x20C`	PMVIDSR
`0x220`	PMPCSR[31:0] (alias)
`0x224`	PMPCSR[63:32] (alias)
`0x228`	PMCID1SR (alias)
`0x22C`	PMCID2SR
`0x600-0x6FC`	IMPLEMENTATION DEFINED
`0xE80-0xEFC`	IMPLEMENTATION DEFINED for CoreSight compliance
`0xFF0-0xFFc`	Management and CoreSight compliance registers

H7.1.3 Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN

The architecture permits IMPLEMENTATION DEFINED extensions to external debug to define mechanisms that make the values of the PC Sample-based profiling registers UNKNOWN. However, it requires that any such mechanism is disabled by default. This means that powerup or a hard reset of the PE must leave the PE in a state where the PC Sample-based Profiling Extension, if implemented, exhibits its architecturally-defined behavior.

Note

A mechanism that, when enabled, makes the PC Sample-based profiling registers UNKNOWN might use other sample-based profiling events that are appropriate for a use that is independent of PC Sample-based Profiling.

If no branch instruction has been retired since the PE left Debug state, suspended state, or a state where PC Sample-based profiling is prohibited, then the sampled value is UNKNOWN.

If all of the following apply, then a read of PMPCSR[31:0] or PMPCSR[63:0] is permitted but not required to return `0xFFFF_FFFF` for PMPCSR[31:0].

- The PE has not retired any branch instruction since leaving a state where PC Sample-based Profiling was suspended.
- Sampling on reads is not disabled.

This includes a read of PMPCSR[31:0] or PMPCSR[63:0] that indirectly makes PC Sample-based Profiling active.

When FEAT_MOPS and FEAT_PCSRv8p2 are not implemented, if no branch instruction has been retired since the last read of PMPCSR[31:0], then the value of PMPCSR[31:0] is UNKNOWN.

H7.1.4 Pseudocode description of PC Sample-based Profiling

When FEAT_PCSRv8 is implemented, the functionality is described by the pseudocode functions:

- `CreatePCSample()`, which populates a variable of type `PCSample`.
- `Read_EDPCSRlo()`, which writes a PC sample to the EDPCSR and associated registers.

When FEAT_PCSRv8p2 is implemented, the functionality is described by the pseudocode functions:

- `CreatePCSample()`, which populates a variable of type `PCSample`.
- `Read_PMPCSR()`, which writes a PC Sample to the PMPCSR and associated registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H8
About the External Debug Registers

This chapter provides some additional information about the external debug registers. It contains the following sections:

- Relationship between external debug and System registers.
- Endianness and supported access sizes.
- Synchronization of changes to the external debug registers.
- Memory-mapped accesses to the external debug interface.
- External debug interface register access permissions.
- External debug interface registers.
- Cross-trigger interface registers.
- External debug register resets.

Note

Where necessary, Table K14-1 disambiguates the general register references used in this chapter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.1 Relationship between external debug and System registers

Table H8-1 shows the relationship between external debug registers and System registers. Where no relationship exists, the registers are not listed.

Table H8-1 Equivalence between external debug and System registers

External debug register	System register AArch64	AArch32	Notes
DBGDTRRX_EL0	DBGDTRRX_EL0	DBGDTRRXint	See also Summary of System register accesses to the DCC
DBGDTRTX_EL0	DBGDTRTX_EL0	DBGDTRTXint	
OSLAR_EL1	OSLAR_EL1	DBGOSLAR	-
DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	DBGBVR<n> DBGBXVR<n>	-
DBGBCR<n>_EL1[31:0] DBGBCR<n>_EL1[63:32]	DBGBCR<n>_EL1[31:0] DBGBCR<n>_EL1[63:32]	DBGBCR<n>	-
DBGWVR<n>_EL1[31:0] DBGWVR<n>_EL1[63:32]	DBGWVR<n>_EL1[31:0] DBGWVR<n>_EL1[63:32]	DBGWVR<n>	-
DBGWCR<n>_EL1[31:0] DBGWCR<n>_EL1[63:32]	DBGWCR<n>_EL1[31:0] DBGWCR<n>_EL1[63:32]	DBGWCR<n>	-
DBGCLAIMSET_EL1	DBGCLAIMSET_EL1	DBGCLAIMSET	-
DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1	DBGCLAIMCLR	-
DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS	Read-only
EDSCR	MDSCR_EL1	DBGDSCRext	Only some fields map
EDECCR	OSECCR_EL1	DBGOSECCR	Applies when the OS Lock is locked
MIDR_EL1	MIDR_EL1	MIDR	Read-only copies of Processor ID Registers
EDDEVAFF0 EDDEVAFF1	MPIDR_EL1[31:0][a] MPIDR_EL1[63:32][a]	MPIDR	Read-only copies of system ID registers

a. This is a word of a 64-bit register.

In addition:

- EDSCR.{TXfull, RXfull} are read-only aliases for DCCSR.{TXfull, RXfull}.
- EDPRCR.CORENPDRQ is a read/write alias for DBGPRCR.CORENPDRQ.
- EDPRSR.OSLK is a read-only alias for OSLSR.OSLK.
- If the FEAT_DoubleLock is implemented, EDPRSR.DLK is a read-only function of OSDLR.DLK.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.2 Endianness and supported access sizes

The debug registers, Performance Monitors registers, and CTI registers are implemented as memory-mapped peripherals. The Arm architecture requires memory-mapped peripherals to be little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the debug registers, Performance Monitors registers, and CTI registers, implementations must:

- Comply with the requirements of Supported access sizes.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the memory-mapped peripheral support making 64-bit accesses.

Note

These requirements mean that a system implementing the debug registers using a 32-bit bus, such as an AMBA APB3, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

For accesses from the external debug interface, the size of an access is determined by the interface. For an access from an ADIv5-compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.3 Synchronization of changes to the external debug registers

This section describes the synchronization requirements for the external debug interface.

For more information on how these requirements affect debug, see:

- Synchronization and debug exceptions for exceptions taken from AArch64 state.
- Synchronization and debug exceptions for exceptions taken from AArch32 state.
- Synchronization and Halting debug events.
- Synchronization of DCC and ITR accesses.

This section refers to accesses from the external debug interface as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers and Synchronization of changes to AArch32 System registers define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads, and external writes as indirect writes.

For general information about synchronization, access completion, ordering, and observability, see Synchronization of memory-mapped registers.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. With the exception of DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, and DBGWVR<n>_EL1, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Synchronization of DCC and ITR accesses describes the synchronization requirements for the DCC and ITR.

Changes to the IMPLEMENTATION DEFINED authentication interface are external writes to the authentication status registers by the Requester of the authentication interface. See Synchronization and the authentication interface.

The external agent must be able to guarantee completion of a write. For example by:

- Marking the memory as Device-nGnRnE and executing a `DSB` barrier, if the system supports this property.
- Reading back the value written.
- Some guaranteed property of the connection between the PE and the external agent.

Note

For an external Debug Access Port, access completion is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, accesses complete in order.

However, the external agent cannot force synchronization of completed writes without halting the PE. Executing an `ISB` instruction, either in Debug state or in Non-debug state, and exiting from Debug state forces synchronization. If the PE is in Debug state, executing an `ISB` instruction is guaranteed to explicitly synchronize any external reads, external writes, and changes to the authentication interface that are ordered before the external write to EDITR.

For any given observer, external writes to the following register groups are guaranteed to be observable in the same order in which they complete:

- The breakpoint registers, DBGBCR<n>_EL1 and DBGBVR<n>_EL1.
- The watchpoint registers, DBGWCR<n>_EL1 and DBGWVR<n>_EL1.

This guarantee applies only to external writes to registers within one of these groups. There is no guarantee regarding the ordering of the observability of external writes within these groups with respect to external writes to registers, for example EDSCR, or between breakpoints and watchpoints, including watchpoints linked to context matching breakpoints.

Note

This means that a debugger can rely on the external writes to be observed in the same order in which they complete. It does not mean that a debugger can rely on the external writes being observed in finite time.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In a simple sequential execution an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

If two or more interfaces simultaneously access a register, the behavior must be as if the accesses occurred atomically and in any order. This is described in Examples of the synchronization of changes to the external debug registers.

Note

Where simultaneous writes occur, once an order has been established by the implementation, some registers have behavior that is incompatible with the implementation simply dropping the older write and only performing the actions of the younger write. In these cases, the implementation must ensure that the register update is consistent with both writes having occurred. For example, writing one to write-one-to-clear or write-one-to-set fields in the older write can produce effects that persist even after writing zero to the same field in the younger write. Examples of registers with this behavior include DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, PMCR_EL0, PMOVSSET_EL0, PMOVSCLR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMSWINC_EL0, and PMZR_EL0, all of which are simultaneously accessible via external debug and System register access.

H8.3.1 Synchronization and the authentication interface

Changes to the authentication interface are indirect writes to the state of the PE by the Requester of the authentication interface.

For an external debug interface read of any Authentication Status register, or an indirect read of the authentication interface made in determining the response to a subsequent external debug interface access, a change on the authentication interface must be observable following a subsequent explicit Context Synchronization event, and:

- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable in finite time.
- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable following an entry to Debug state.

For a System register read of DBGAUTHSTATUS_EL1, a change on the authentication interface is guaranteed to be observable only after a Context Synchronization event.

Note

- In some systems, the authentication interface is fixed by configuration or is changed under the control of software. These systems can require explicit synchronization for any change to the authentication interface.
- In other systems, the authentication interface is controlled dynamically by an external agent. In these systems, it is desirable that changes to the authentication interface do not require explicit synchronization by software executing on the PE to be observable by subsequent external debug interface accesses, and are either observable in finite time or are synchronized by entry to Debug state. Otherwise, there are scenarios where a debugger is not able to halt and debug the system.

H8.3.2 Examples of the synchronization of changes to the external debug registers

Example H8-1, Example H8-2, and Example H8-3 show the synchronization of changes to the external debug registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example H8-1 Order of synchronization of Breakpoint and Watchpoint register writes

Initially DBGBVR<n>_EL1 is `0x8000` and DBGBCR<n>_EL1 is `0x0181`. This means that a breakpoint is enabled on the halfword T32 instruction at address `0x8000`.

A sequence of external writes occurs in the following order:

1. `0x0000` is written to DBGBCR<n>_EL1, disabling the breakpoint.
2. `0x9000` is written to DBGBVR<n>_EL1[31:0].
3. `0x0061` is written to DBGBCR<n>_EL1, enabling a breakpoint on the halfword at address `0x9002`.

The external writes must be observable to indirect reads in the same order as the external writes complete. This means that at no point is there a breakpoint enabled on either of the halfwords at address `0x8002` and `0x9000`.

Similarly a breakpoint or watchpoint must be disabled:

- If both halves of a 64-bit address have to be updated.
- If any of the DBGBCR<n>_EL1 or DBGWCR<n>_EL1 fields are modified at the same time as updating the address.

Example H8-2 Simultaneous accesses to DTR registers

Initially EDSCR.{TXfull, TXU, ERR} are 0. Then:

- `0x0DCCDA7A` is directly written to DBGDTRTX_EL0 by an `MSR` instruction.
- DBGDTRTX_EL0 is indirectly read by the external debug interface.

These accesses might happen at the same time and in any order.

If the direct write of `0x0DCCDA7A` to DBGDTRTX_EL0 is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 clears EDSCR.TXfull to 0.
- EDSCR.{TXU, ERR} are unchanged.
- The external debug interface read returns `0x0DCCDA7A`.

If the indirect read of DBGDTRTX_EL0 by the external debug interface is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 causes an underrun and as a result EDSCR.{TXU, ERR} are both set to 1.
- The external debug interface returns an UNKNOWN value.
- Writing `0x0DCCDA7A` to DBGDTRTX_EL0 sets DTRTX to `0x0DCCDA7A` and EDSCR.TXfull to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example H8-3 Simultaneous writes to CLAIM registers

Initially all CLAIM tag bits are 0. Then:

- `0x01` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit Context Synchronization event.
- `0x02` is written to DBGCLAIMSET_EL1 by an external write.

These events might happen at the same time and in either order.

After this:

- DBGCLAIMCLR_EL1 is read by a direct read.
- DBGCLAIMCLR_EL1 is read by an external read.

In this case, a direct read can return either `0x01` or `0x03`, and the external read can return either `0x02` or `0x03`.

The only permitted final result for the CLAIM tags is the value `0x03`, because this would be the result regardless of whether `0x01` or `0x02` is written first. This is because the external write is guaranteed to be observable to a direct read in finite time. See Synchronization requirements for AArch64 System registers.

It is not possible for a direct read to return `0x01` and the external read to return `0x02`, because the writes to DBGCLAIMCLR_EL1 are serialized.

In the following scenario, there is only one permitted result. Both observers observe the value `0x03`, and then, at the same time, two writes occur:

- `0x04` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit Context Synchronization event.
- `0x01` is written to DBGCLAIMCLR_EL1 by an external write.

In this case, the only permitted final result for the CLAIM tags is the value `0x06`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is OPTIONAL. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is CoreSight compliant, then an OPTIONAL Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is OPTIONAL and deprecated. If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by EDLSR and EDLAR, PMLSR and PMLAR, and CTILSR and CTILAR. See Management registers and CoreSight compliance.

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components in the Core power domain.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

Note

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in Armv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when OSLSR.OSLK == 1, meaning that the OS Lock is locked.

The following sections give more information about these memory-mapped accesses:

- Register access permissions for memory-mapped accesses.
- Synchronization of memory-mapped accesses to external debug registers.

See also Supported access sizes.

H8.4.1 Register access permissions for memory-mapped accesses

It is IMPLEMENTATION DEFINED whether unprivileged memory-mapped accesses are allowed. Privileged software is responsible for controlling memory-mapped accesses using the MMU.

If FEAT_Debugv8p4 is implemented, the Security state view of a debug component is mapped into the physical memory for that Security state.

If FEAT_Debugv8p4 is implemented, the access permissions are different in each Security state, but Secure and Non-secure views of the debug components are identical. Arm recommends the views are located at the same address in the Secure and Non-secure physical address maps.

If memory-mapped accesses are made through an ADIv5 interface, the Debug Access Port can block the access using DBGSWENABLE. This is outside the scope of the Arm architecture. See *Arm ® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*.

H8.4.1.1 Effect of the OPTIONAL Software Lock on memory-mapped access

For memory-mapped accesses, if other controls permit access to a register, the OPTIONAL Software Lock is implemented, and EDLSR.SLK, PMLSR.SLK, or CTILSR.SLK is set to 1, meaning the Software Lock is locked, then with the exception of the LAR itself:

- If other controls permit access to a register, then writes are ignored. That is:
  - Read/write (RW) registers become read-only, writes ignored (RO/WI).
  - Write-only (WO) registers become writes ignored (WI).
- Reads and writes have no side-effects. A side-effect is where a direct read or a direct write of a register creates an indirect write of the same or another register. When the Software Lock is locked, the indirect write does not occur.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Writes to EDLAR, PMLAR, and CTILAR are unaffected.

This behavior must also apply to all IMPLEMENTATION DEFINED registers.

For example, if EDLSR.SLK is set to 1:

- EDSCR.{TXfull, TXU, ERR} are unchanged by a memory-mapped read from DBGDTRTX_EL0.
- EDSCR.{RXfull, RXO, ERR} are unchanged by a memory-mapped write to DBGDTRRX_EL0 that is ignored.
- EDSCR.{ITE, ITO, ERR} are unchanged by a memory-mapped write to EDITR that is ignored.
- OSLSR.OSLK is unchanged by a memory-mapped write to OSLAR_EL1 that is ignored.
- EDPCSR[63:32], EDCIDSR, and EDVIDSR are unchanged by a memory-mapped read from EDPCSR[31:0].

  Note

  Updating EDVIDSR, EDCIDSR, and EDPCSRhi are side-effects of reading EDPCSRlo, such that these registers contain the matching context for EDPCSRlo. The process that updates EDPCSRlo with PC samples is not a side-effect of the access. Reads of EDPCSRlo made when the Software Lock is locked can be used to profile software.

- PMPCSR[63:32], PMCID1SR/PMCID2SR, and PMVIDSR are unchanged by a memory-mapped read from PMPCSR[31:0].

  Note

  Updating PMVIDSR, PMCID1SR/PMCID2SR, and PMPCSR[31:0] are side-effects of reading PMPCSR[63:32], such that these registers contain the matching context for PMPCSR[63:32]. The process that updates PMPCSR[63:32] with PC samples is not a side-effect of the access. Reads of PMPCSR[63:32] made when the Software Lock is locked can be used to profile software.

- EDPRSR.{SDR, SPMAD, SDAD, SR, SPD} are unchanged by a memory-mapped read from EDPRSR.
- EDPRSR.SDAD is not set if an error response is returned due to a memory-mapped read or write of any debug register as the result of the value of the EDAD field.
- The CLAIM tags are unchanged by memory-mapped writes to DBGCLAIMSET_EL1 and DBGCLAIMCLR_EL1 which are ignored.

Similarly, if PMLSR.SLK is set to 1, then EDPRSR.SPMAD is not set if an error response is returned to a memory-mapped read or write of any Performance Monitors register due to the value of the EPMAD field.

H8.4.1.2 Behavior of a not permitted memory-mapped access

Where the architecture requires that an external debug interface access generates an error response, a memory-mapped access must also generate an error response. However, it is IMPLEMENTATION DEFINED how the error response is handled, as this depends on the system.

Arm recommends that the error is returned as either:

- A synchronous external Data Abort.
- An SError exception.

H8.4.2 Synchronization of memory-mapped accesses to external debug registers

The synchronization requirements for memory-mapped accesses to the external debug interface is described in Synchronization of changes to the external debug registers and Synchronization of memory-mapped registers.

The synchronization requirements between different routes to the external debug interface, that is, between Debug Access Port accesses and memory-mapped accesses are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.5 External debug interface register access permissions

Some external accesses to debug registers and Performance Monitor registers are not permitted and return an error response if:

- The Core power domain is powered down or is in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1. The OS Lock is locked.
- FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.
- The access is disabled by either the authentication interface or secure monitor.

Not all registers are affected in all of these cases. For more information, see External debug interface register access permissions summary.

H8.5.1 External debug over powerdown and locks

Accessing registers using the external debug interface is not possible when the Debug power domain is off. In this case, all accesses return an error.

External accesses to debug and Performance Monitors registers in the Core power domain are not permitted and return an error response if:

- The Core power domain is off or in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1, meaning that the OS Lock is locked. This allows software to prevent external debugger modification of the registers while it saves and restores them over powerdown.
- FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. This means that the OS Double Lock is locked. The OS Double Lock ensures that it is safe to remove Core power by forcing the debug interface to be quiescent.

If FEAT_DoubleLock is not implemented, the hardware must provide another method to safely remove Core power.

The OS Lock condition does not apply to the following debug registers:

- OSLAR_EL1. This means that an external debugger can override this lock.
- EDESR. This means that an external debugger can program a debug event for when software unlocks the OS Lock. See OS Unlock Catch debug event.
- The ID registers that describe the PE to the debugger.

See also Debug registers to save over powerdown.

H8.5.2 External access disabled

Accesses are further controlled by the external authentication interface. An untrusted external debugger cannot program the breakpoint and watchpoint registers to generate spurious debug exceptions. If external invasive debugging is not enabled, these external accesses to the registers are disabled. If EL3 is implemented, then SDCR provides additional external access controls for those registers.

The disable applies to:

- The DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, and DBGWCR<n>_EL1 registers.
- From Armv8.2, the OSLAR_EL1 register.

If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether the disable applies to OSLAR_EL1.

If FEAT_Debugv8p4 is not implemented, the external debug interface cannot access these registers if any of the following are true:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- `ExternalSecureInvasiveDebugEnabled()` == FALSE, EL3 is not implemented, and the PE behaves as if the Security state is Secure.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `ExternalSecureInvasiveDebugEnabled()` == FALSE, EL3 is implemented and SDCR.EDAD == 1.

If FEAT_Debugv8p4 is implemented, accesses from the external debug interface to these registers are not permitted if any of the following are true:

- EL3 is not implemented, the PE behaves as if the Security state is Secure, and the access is Non-secure.
- The access is not permitted by SDCR.EDAD and, if FEAT_RME is implemented, MDCR_EL3.EDADE.

The `AllowExternalDebugAccess()` pseudocode function describes these accessibility rules.

PEs might also provide an OPTIONAL external debug interface to the Performance Monitor registers. The authentication interface and SDCR provide similar external access disable controls for those registers.

If FEAT_Debugv8p4 is not implemented, the external debug interface cannot access the Performance Monitor registers if any of the following are true:

- `ExternalNoninvasiveDebugEnabled()` == FALSE.
- `ExternalSecureNoninvasiveDebugEnabled()` == FALSE, EL3 is not implemented and the PE behaves as if the Security state is Secure.
- `ExternalSecureNoninvasiveDebugEnabled()` == FALSE, EL3 is implemented and SDCR.EPMAD == 1.

Note

Arm recommends that Secure software that is not making use of debug hardware does not lock out the external debug interface.

If FEAT_Debugv8p4 is implemented, accesses from the external debug interface to these registers are not permitted if any of the following are true:

- EL3 is not implemented, the PE behaves as if the Security state is Secure, and the access is Non-secure.
- The access is not permitted by SDCR.EDAD and, if FEAT_RME is implemented, MDCR_EL3.EDADE.

The `AllowExternalPMUAccess()` pseudocode function describes these accessibility rules.

H8.5.3 Behavior of a not permitted access

For an external debug interface access by a Debug Access Port, the Debug Access Port receives the error response and must signal this to the external debugger. For an ADIv5 implementation of a Debug Access Port, the error sets a sticky error flag in the Debug Access Port that the debugger can poll, and that suppresses further accesses until it is explicitly cleared.

When an error is returned because external access is disabled, and this is the highest priority error condition, a sticky error flag in EDPRSR is indirectly written to 1 as a side-effect of the access:

- For a debug register access when `AllowExternalDebugAccess()` == FALSE, EDPRSR.SDAD is indirectly written to 1.
- For Performance Monitor register access when `AllowExternalPMUAccess()` == FALSE, EDPRSR.SPMAD is indirectly written to 1.

The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see Register access permissions for memory-mapped accesses.

If no error is returned, or the error is returned because of a higher priority error condition, the flag in EDPRSR is unchanged.

See also Behavior of a not permitted memory-mapped access.

For more information, see *Arm® Debug Interface Architecture Specification*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.5.4 External debug interface register access permissions summary

For accesses to:

- IMPLEMENTATION DEFINED registers, see IMPLEMENTATION DEFINED registers.
- OPTIONAL registers for CoreSight compliance, see Management registers and CoreSight compliance.
- Reserved, unallocated, or unimplemented registers, writes to read-only registers, and reads of write-only registers, see Reserved and unallocated registers.

For all other external debug interface, CTI, and Performance Monitor registers, Table H8-3, Table H8-4, Table H8-6 and Table I3-2, show the response of the PE to accesses by the external debug interface.

H8.5.5 IMPLEMENTATION DEFINED registers

Register access permissions for IMPLEMENTATION DEFINED debug, Performance Monitors, and CTI registers are IMPLEMENTATION DEFINED, other than as follows:

- If FEAT_DoPD is implemented, then all of the following apply:
  - All debug and Performance Monitors registers, including IMPLEMENTATION DEFINED registers, are implemented in the Core power domain and accesses return an error when the Core power domain is off or in a low-power state.
  - CTI registers, including IMPLEMENTATION DEFINED registers, are implemented in the Debug power domain.
- When memory-mapped access to the external debug interface is supported, the additional constraints in Register access permissions for memory-mapped accesses apply to IMPLEMENTATION DEFINED registers.

H8.5.6 Reserved and unallocated registers

The default access requirements for reserved and unallocated registers are described in Access requirements for reserved and unallocated registers.

Note

Reads of WO and writes to RO refers to the default access permissions for a register. For example, when the SLK field is 1, meaning that the relevant registers become RO, a memory-mapped write to a RW register is ignored, and not treated as a reserved access.

The following reserved registers are RES0 in all conditions, other than when debug power is off:

- All reserved CTI registers.
- For the debug registers, and Performance Monitors registers, if the implementation is CoreSight architecture compliant, and either FEAT_DoPD is not implemented or the Core power domain is on, all reserved registers in the range `0xF00 - 0xFFC`. See Management register access permissions.

Otherwise, the architecture defines that:

1. If debug power is off, all register accesses, including reserved accesses, return an error.
2. For reserved debug registers and Performance Monitors registers, if FEAT_DoPD is implemented, and the Core power domain is off or in a low-power state, the response is an error. Otherwise, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0, when any of the following hold:

   Off The Core power domain is either completely off or in a low-power state in which the Core power domain registers cannot be accessed.

   DLK FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

   OSLK OSLSR.OSLK == 1. The OS Lock is locked.
3. In addition, for reserved debug registers in the address ranges `0x400 - 0x4FC` and `0x800 - 0x8FC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EDAD `AllowExternalDebugAccess()` == FALSE. External debug is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

See also Behavior of a not permitted access.

4. In addition, for reserved Performance Monitors registers in the address ranges `0x000 - 0xEFC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EPMAD `AllowExternalPMUAccess()` == FALSE. External Performance Monitor access is disabled.

   Note

   See also Behavior of a not permitted access.

5. For reads of WO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a write to the location to return an error.
6. For writes of RO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a read to the location to return an error.
7. For reads and writes of locations for features that are not implemented, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires an access to the location to return an error if the feature is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.6 External debug interface registers

The external debug interface register map is described by:

- Summary of external debug registers.
- Cross-trigger interface registers.
- Performance Monitors external register views.

Table H8-2 Summary of external debug registers

Offset	Name	Description	Access
0x020	EDESR	External Debug Event Status Register	RW
0x024	EDECR	External Debug Execution Control Register	RW
0x028	EDSCR2	External Debug Status and Control Register 2	RW
0x030	EDWAR	External Debug Watchpoint Address Register	RO
0x038	EDHSR	External Debug Halting Syndrome Register	RO
0x080	DBGDTRRX_EL0	Debug Data Transfer Register, Receive	RW
0x084	EDITR	External Debug Instruction Transfer Register	WO
0x088	EDSCR	External Debug Status and Control Register	RW
0x08C	DBGDTRTX_EL0	Debug Data Transfer Register, Transmit	RW
0x090	EDRCR	External Debug Reserve Control Register	WO
0x094	EDACR	External Debug Auxiliary Control Register	RW
0x098	EDECCR	External Debug Exception Catch Control Register	RW
0x0A0	EDPCSR	External Debug Program Counter Sample Register	RO
0x0A0	EDPCSR	External Debug Program Counter Sample Register	RO
0x0A4	EDCIDSR	External Debug Context ID Sample Register	RO
0x0A8	EDVIDSR	External Debug Virtual Context Sample Register	RO
0x0AC	EDPCSR	External Debug Program Counter Sample Register	RO
0x0AC	EDPCSR	External Debug Program Counter Sample Register	RO
0x300	OSLAR_EL1	OS Lock Access Register	WO
0x310	EDPRCR	External Debug Power/Reset Control Register	RW
0x314	EDPRSR	External Debug Processor Status Register	RO
0x400 + (16 * n)	DBGBVR<n>_EL1	Debug Breakpoint Value Registers	RW
0x408 + (16 * n)	DBGBCR<n>_EL1	Debug Breakpoint Control Registers	RW
0x800 + (16 * n)	DBGWVR<n>_EL1	Debug Watchpoint Value Registers	RW
0x808 + (16 * n)	DBGWCR<n>_EL1	Debug Watchpoint Control Registers	RW
0xD00	MIDR_EL1	Main ID Register	RO
0xD20	EDPFR	External Debug Processor Feature Register	RO
0xD28	EDDFR	External Debug Feature Register	RO
0xD48	EDDFR1	External Debug Feature Register 1	RO
0xD60	EDAA32PFR	External Debug Auxiliary Processor Feature Register	RO
0xF00	EDITCTRL	External Debug Integration mode Control register	RW

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Name	Description	Access
0xFA0	DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register	RW
0xFA4	DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register	RW
0xFA8	EDDEVAFF0	External Debug Device Affinity register 0	RO
0xFAC	EDDEVAFF1	External Debug Device Affinity register 1	RO
0xFB0	EDLAR	External Debug Lock Access Register	WO
0xFB4	EDLSR	External Debug Lock Status Register	RO
0xFB8	DBGAUTHSTATUS_EL1	Debug Authentication Status Register	RO
0xFBC	EDDEVARCH	External Debug Device Architecture Register	RO
0xFC0	EDDEVID2	External Debug Device ID register 2	RO
0xFC4	EDDEVID1	External Debug Device ID Register 1	RO
0xFC8	EDDEVID	External Debug Device ID register 0	RO
0xFCC	EDDEVTYPE	External Debug Device Type register	RO
0xFD0	EDPIDR4	External Debug Peripheral Identification Register 4	RO
0xFE0	EDPIDR0	External Debug Peripheral Identification Register 0	RO
0xFE4	EDPIDR1	External Debug Peripheral Identification Register 1	RO
0xFE8	EDPIDR2	External Debug Peripheral Identification Register 2	RO
0xFEC	EDPIDR3	External Debug Peripheral Identification Register 3	RO
0xFF0	EDCIDR0	External Debug Component Identification Register 0	RO
0xFF4	EDCIDR1	External Debug Component Identification Register 1	RO
0xFF8	EDCIDR2	External Debug Component Identification Register 2	RO
0xFFC	EDCIDR3	External Debug Component Identification Register 3	RO

H8.6.1 Access permissions for the External debug interface registers

Table H8-3 and Table H8-4 show the access permissions for the external debug interface registers in an Arm A-profile Debug implementation. The terms are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If FEAT_DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table H8-3.

If FEAT_DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table H8-4.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

The conditions are:

Off The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed, and if FEAT_DoPD is not implemented, EDPRSR.PU will read as 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

When the Core power domain is off, or in a low-power state, a debugger is permitted to access a debug register that is implemented in the external Debug power domain. When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

DLK FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked. If FEAT_DoPD is implemented, FEAT_DoubleLock is not implemented and so Table H8-3 does not include this column.

OSLK OSLSR.OSLK == 1. The OS Lock is locked.

EDAD `AllowExternalDebugAccess()` == FALSE. External debug access is disabled for the access. See also Behavior of a not permitted access.

EPMAD `AllowExternalPMUAccess()` == FALSE. Access to the external Performance Monitors is disabled for the access. See also Behavior of a not permitted access.

SLK The Software Lock is implemented and `SoftwareLockStatus()` == TRUE. This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked for the access. See Register access permissions for memory-mapped accesses. If FEAT_DoPD is implemented, the Software Lock is not locked or not implemented, then this column is ignored.

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO This means that the register or field is read-only, and:

- Unless the register description states otherwise, a RO field in an RW register ignores writes.
- Where the `SLK` control makes a RW register RO, the register ignores writes.

RW This means that the register or field is read/write. Individual fields within the register might be RO or WO. See the relevant register description for details.

RC This means that a read of the register bit clears the field to 0.

WO This means that the register or field is write-only. Unless the register description states otherwise, a WO field in a RW register returns an UNKNOWN value on a read of the register.

WI This means that the register or field ignores writes.

IMP DEF This means that the access permissions are IMPLEMENTATION DEFINED.

DBGBVR<n>_EL1 and DBGWVR<n>_EL1, and if FEAT_Debugv8p9 is implemented, DBGBCR<n>_EL1 and DBGWCR<n>_EL1, are 64-bit registers mapped to pairs of 32-bit locations in the external debug interface. Doubleword accesses to these register are not guaranteed to be 64-bit single copy atomic. Software must take care before altering the values of DBGBVR<n>_EL1, DBGWVR<n>_EL1, DBGBCR<n>_EL1, and DBGWCR<n>_EL1 to prevent UNPREDICTABLE behavior. See Endianness and supported access sizes for more information.

If FEAT_Debugv8p9 is not implemented, DBGBCR<n>_EL1 and DBGWCR<n>_EL1 are 32-bit registers in the external debug interface, and accesses to DBGBCR<n>_EL1[63:32] and DBGWCR<n>_EL1[63:32] are RES0.

If OPTIONAL memory-mapped access to the external debug interface is supported, there might be additional constraints on memory-mapped accesses. See Register access permissions for memory-mapped accesses.

For the reset values for the external debug interface registers, see Table H8-7.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H8-3 Access permissions for the external debug interface registers if FEAT_DoPD is implemented

Offset	Register	Domain	Conditions (priority from left to right)			
			Off	OSLK	EDAD	Default
0x020	EDESR	Core	Error	-	-	RW
0x024	EDECR	Core	Error	-	-	RW
0x028	EDSCR2	Core	Error	Error	-	RW
0x030 0x034	EDWAR[31:0] EDWAR[63:32]	Core Core	Error Error	Error Error	- -	RO RO
0x038 0x03C	EDHSR[31:0] EDHSR[63:32]	Core Core	Error Error	Error Error	- -	RO RO
0x080	DBGDTRRX_EL0	Core	Error	Error	-	RW
0x084	EDITR	Core	Error	Error	-	WO
0x088	EDSCR	Core	Error	Error	-	RW
0x08C	DBGDTRTX_EL0	Core	Error	Error	-	RW
0x090	EDRCR	Core	Error	Error	-	WO
0x094	EDACR	Core	Error	IMP DEF	-	RW
0x098	EDECCR	Core	Error	Error	-	RW
0x0A0	EDPCSR[31:0][a]	Core	Error	Error	-	RO
0x0A4	EDCIDSR[a]	Core	Error	Error	-	RO
0x0A8	EDVIDSR[a]	Core	Error	Error	-	RO
0x0AC	EDPCSR[63:32][a]	Core	Error	Error	-	RO
0x300	OSLAR_EL1	Core	Error	-	Error	WO
0x310	EDPRCR	Core	Error	-	-	RW
0x314	EDPRSR	Core	Error	-	-	RO
0x400+16*n 0x404+16*n	DBGBVR<n>_EL1[31:0][b] DBGBVR<n>_EL1[63:32][b]	Core Core	Error Error	Error Error	Error Error	RW RW
0x408+16*n 0x40C+16*n	DBGBCR<n>_EL1[31:0][b] DBGBCR<n>_EL1[63:32] [b,c]	Core Core	Error Error	Error Error	Error Error	RW RW
0x800+16*n 0x804+16*n	DBGWVR<n>_EL1[31:0][b] DBGWVR<n>_EL1[63:32][b]	Core Core	Error Error	Error Error	Error Error	RW RW
0x808+16*n 0x80C+16*n	DBGWCR<n>_EL1[31:0][b] DBGWCR<n>_EL1[63:32] [b,c]	Core Core	Error Error	Error Error	Error Error	RW RW
0xD00	MIDR_EL1	Core	Error	-	-	RO
0xD20	EDPFR[31:0]	Core	Error	-	-	RO
0xD24	EDPFR[63:32]	Core	Error	-	-	RO
0xD28	EDDFR[31:0]	Core	Error	-	-	RO
0xD2C	EDDFR[63:32]	Core	Error	-	-	RO
0xD48	EDDFR1[31:0]	Core	Error	-	-	RO
0xD4C	EDDFR1[63:32]	Core	Error	-	-	RO
0xD60	EDAA32PFR[31:0]	Core	Error	-	-	RO
0xD64	EDAA32PFR[63:32]	Core	Error	-	-	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	-	RW

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority from left to right)			
			Off	OSLK	EDAD	Default
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	-	RW
0xFA8	EDDEVAFF0	Core	Error	-	-	RO
0xFAC	EDDEVAFF1	Core	Error	-	-	RO
0xFB8	DBGAUTHSTATUS_EL1	Core	Error	-	-	RO
0xFC0	EDDEVID2	Core	Error	-	-	RO
0xFC4	EDDEVID1	Core	Error	-	-	RO
0xFC8	EDDEVID	Core	Error	-	-	RO

a. Implemented only if the PC Sample-based Profiling Extension is implemented and FEAT_PCSRv8p2 is not implemented.

b. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

c. When FEAT_Debugv8p9 is implemented.

Table H8-4 Access permissions for the external debug interface registers if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority from left to right)					
			Off	DLK	OSLK	EDAD	Default	SLK
0x020	EDESR	Core	Error	Error	-	-	RW	RO
0x024	EDECR	Debug	-	-	-	-	RW	RO
0x028	EDSCR2	Core	Error	Error	Error	-	RW	RO
0x030	EDWAR[31:0]	Core	Error	Error	Error	-	RO	-
0x034	EDWAR[63:32]	Core	Error	Error	Error	-	RO	-
0x038	EDHSR[31:0]	Core	Error	Error	Error	-	RO	RO
0x03C	EDHSR[63:32]	Core	Error	Error	Error	-	RO	RO
0x080	DBGDTRRX_EL0	Core	Error	Error	Error	-	RW	RO
0x084	EDITR	Core	Error	Error	Error	-	WO	WI
0x088	EDSCR	Core	Error	Error	Error	-	RW	RO
0x08C	DBGDTRTX_EL0	Core	Error	Error	Error	-	RW	RO
0x090	EDRCR	Core	Error	Error	Error	-	WO	WI
0x094	EDACR	IMP DEF	IMP DEF	IMP DEF	IMP DEF	-	RW	RO
0x098	EDECCR	Core	Error	Error	Error	-	RW	RO
0x0A0	EDPCSR[31:0][a]	Core	Error	Error	Error	-	RO	RO
0x0A4	EDCIDSR[a]	Core	Error	Error	Error	-	RO	RO
0x0A8	EDVIDSR[a]	Core	Error	Error	Error	-	RO	RO
0x0AC	EDPCSR[63:32]	Core	Error	Error	Error	-	RO	RO
0x300	OSLAR_EL1	Core	Error	Error	-	IMP DEF[b]	WO	WI
0x310	EDPRCR	Core and Debug[c]	-	-	-	-	RW	RO
0x314	EDPRSR	Core and Debug[c]	-	-	-	-	RO	RO
0x400+16*n	DBGBVR<n>_EL1[31:0][d]	Core	Error	Error	Error	Error	RW	RO
0x404+16*n	DBGBVR<n>_EL1[63:32][d]	Core	Error	Error	Error	Error	RW	RO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

			Conditions (priority from left to right)					
Offset	**Register**	**Domain**	**Off**	**DLK**	**OSLK**	**EDAD**	**Default**	**SLK**
`0x408+16*n`	DBGBCR<n>_EL1[31:0][d]	Core	Error	Error	Error	Error	RW	RO
`0x40C+16*n`	DBGBCR<n>_EL1[63:32][d,e]	Core	Error	Error	Error	Error	RW	RO
`0x800+16*n`	DBGWVR<n>_EL1[31:0][d]	Core	Error	Error	Error	Error	RW	RO
`0x804+16*n`	DBGWVR<n>_EL1[63:32][d]	Core	Error	Error	Error	Error	RW	RO
`0x808+16*n`	DBGWCR<n>_EL1[31:0][d]	Core	Error	Error	Error	Error	RW	RO
`0x80C+16*n`	DBGWCR<n>_EL1[63:32][d,e]	Core	Error	Error	Error	Error	RW	RO
`0xD00`	MIDR_EL1	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD20`	EDPFR[31:0]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD24`	EDPFR[63:32]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD28`	EDDFR[31:0]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD2C`	EDDFR[63:32]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD48`	EDDFR1[31:0]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD48`	EDDFR1[63:32]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD60`	EDAA32PFR[31:0]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xD64`	EDAA32PFR[63:32]	IMP DEF	IMP DEF[f]	IMP DEF[f]	-	-	RO	RO
`0xFA0`	DBGCLAIMSET_EL1	Core	Error	Error	Error	-	RW	RO
`0xFA4`	DBGCLAIMCLR_EL1	Core	Error	Error	Error	-	RW	RO
`0xFA8`	EDDEVAFF0	Debug	-	-	-	-	RO	RO
`0xFAC`	EDDEVAFF1	Debug	-	-	-	-	RO	RO
`0xFB8`	DBGAUTHSTATUS_EL1	Debug	-	-	-	-	RO	RO
`0xFC0`	EDDEVID2	Debug	-	-	-	-	RO	RO
`0xFC4`	EDDEVID1	Debug	-	-	-	-	RO	RO
`0xFC8`	EDDEVID	Debug	-	-	-	-	RO	RO

a. Implemented only if the PC Sample-based Profiling Extension is implemented.

b. If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether an error is returned. See External access disabled. If no error is returned, the access is permitted.

c. Some bits are in the Debug power domain and some bits are in the Core power domain. See register field descriptions for information.

d. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

e. When FEAT_Debugv8p9 is implemented.

f. It is IMPLEMENTATION DEFINED whether an error is returned. See External debug over powerdown and locks. If no error is returned, the access is permitted.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.7 Cross-trigger interface registers

The Embedded Cross-Trigger Interface, CTI, is located within its own block of the external debug memory map. There must be one such block for each PE.

If the CTI of a PE does not implement the CTIDEVAFF0 or CTIDEVAFF1 registers it must be located 64KB above the debug registers in the external debug interface.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

Table H8-5 shows the CTI register map.

Table H8-5 Summary of the Embedded Cross-Trigger Interface, CTI, registers

Offset	Name	Description	Access
`0x000`	CTICONTROL	CTI Control register	RW
`0x010`	CTIINTACK	CTI Output Trigger Acknowledge register	WO
`0x014`	CTIAPPSET	CTI Application Trigger Set register	RW
`0x018`	CTIAPPCLEAR	CTI Application Trigger Clear register	WO
`0x01C`	CTIAPPPULSE	CTI Application Pulse register	WO
`0x020 + (4 * n)`	CTIINEN<n>	CTI Input Trigger to Output Channel Enable registers	RW
`0x0A0 + (4 * n)`	CTIOUTEN<n>	CTI Input Channel to Output Trigger Enable registers	RW
`0x130`	CTITRIGINSTATUS	CTI Trigger In Status register	RO
`0x134`	CTITRIGOUTSTATUS	CTI Trigger Out Status register	RO
`0x138`	CTICHINSTATUS	CTI Channel In Status register	RO
`0x13C`	CTICHOUTSTATUS	CTI Channel Out Status register	RO
`0x140`	CTIGATE	CTI Channel Gate Enable register	RW
`0x144`	ASICCTL	CTI External Multiplexer Control register	RO
`0x150`	CTIDEVCTL	CTI Device Control register	RW
`0xF00`	CTIITCTRL	CTI Integration mode Control register	RW
`0xFA0`	CTICLAIMSET	CTI CLAIM Tag Set register	RW
`0xFA4`	CTICLAIMCLR	CTI CLAIM Tag Clear register	RW
`0xFA8`	CTIDEVAFF0	CTI Device Affinity register 0	RO
`0xFAC`	CTIDEVAFF1	CTI Device Affinity register 1	RO
`0xFB0`	CTILAR	CTI Lock Access Register	WO
`0xFB4`	CTILSR	CTI Lock Status Register	RO
`0xFB8`	CTIAUTHSTATUS	CTI Authentication Status register	RO
`0xFBC`	CTIDEVARCH	CTI Device Architecture register	RO
`0xFC0`	CTIDEVID2	CTI Device ID register 2	RO
`0xFC4`	CTIDEVID1	CTI Device ID register 1	RO
`0xFC8`	CTIDEVID	CTI Device ID register 0	RO
`0xFCC`	CTIDEVTYPE	CTI Device Type register	RO
`0xFD0`	CTIPIDR4	CTI Peripheral Identification Register 4	RO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Name	Description	Access
0xFE0	CTIPIDR0	CTI Peripheral Identification Register 0	RO
0xFE4	CTIPIDR1	CTI Peripheral Identification Register 1	RO
0xFE8	CTIPIDR2	CTI Peripheral Identification Register 2	RO
0xFEC	CTIPIDR3	CTI Peripheral Identification Register 3	RO
0xFF0	CTICIDR0	CTI Component Identification Register 0	RO
0xFF4	CTICIDR1	CTI Component Identification Register 1	RO
0xFF8	CTICIDR2	CTI Component Identification Register 2	RO
0xFFC	CTICIDR3	CTI Component Identification Register 3	RO

Table H8-6 shows the access permissions for the CTI registers in an Arm A-profile Debug implementation. For a definition of the terms used, see External debug interface registers.

Table H8-6 Access permissions for the CTI registers

			Conditions (priority from left to right)					
Offset	Register	Domain	Off	DLK	OSLK	EDAD	Default	SLK
0x000	CTICONTROL	Debug	-	-	-	-	RW	RO
0x010	CTIINTACK	Debug	-	-	-	-	WO	WI
0x014	CTIAPPSET	Debug	-	-	-	-	RW	RO
0x018	CTIAPPCLEAR	Debug	-	-	-	-	WO	WI
0x01C	CTIAPPPULSE	Debug	-	-	-	-	WO	WI
0x020+4×n	CTIINEN<n>	Debug	-	-	-	-	RW	RO
0x0A0+4×n	CTIOUTEN<n>	Debug	-	-	-	-	RW	RO
0x130	CTITRIGINSTATUS	Debug	-	-	-	-	RO	RO
0x134	CTITRIGOUTSTATUS	Debug	-	-	-	-	RO	RO
0x138	CTICHINSTATUS	Debug	-	-	-	-	RO	RO
0x13C	CTICHOUTSTATUS	Debug	-	-	-	-	RO	RO
0x140	CTIGATE	Debug	-	-	-	-	RW	RO
0xFC0	CTIDEVID2	Debug	-	-	-	-	RO	RO
0xFC4	CTIDEVID1	Debug	-	-	-	-	RO	RO
0xFC8	CTIDEVID	Debug	-	-	-	-	RO	RO

For the reset values of the CTI registers, see Table H8-8.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H8.8 External debug register resets

Each register or field has a defined reset domain:

- Registers and fields in the Warm reset domain are also reset by a Cold reset and unchanged by an External Debug reset that is not coincident with a Cold reset or a Warm reset.
- Registers and fields in the Cold reset domain are unchanged by a Warm reset or an External Debug reset that is not coincident with a Cold reset.
- Registers and fields in the External Debug reset domain are unchanged by a Cold reset or a Warm reset that is not coincident with an External Debug reset.

A reset might change the value of a register. Specific rules apply to the observability of registers in the External Debug reset domain by indirect reads from the Core power domain when an External Debug reset is asserted without a coincident Cold reset. For more information, see Synchronization of changes to the external debug registers.

Table H8-7 and Table H8-8 show the external debug register and CTI register resets. For other debug registers, see Management register resets.

Note

By reference to Figure H6-1 the power domain can be deduced from the reset domain. Table K5-10 also shows reset power domains.

Table H8-7 and Table H8-8 do not include:

- Read-only identification registers, such as Processor ID Registers and PMCFGR, that have a fixed value from reset.
- Read-only status registers, such as EDSCR.RW, that are evaluated each time the register is read and that have no meaningful reset value.
- Write-only registers, such as EDRCR, that only have an effect on writes, and have no meaningful reset value.
- Read/write registers, such as breakpoint and watchpoint registers, and EDPRCR.CORENPDRQ, that alias other registers. The reset values are described by the descriptions of those other registers.
- IMPLEMENTATION DEFINED registers. The reset values and reset domains of these registers are also IMPLEMENTATION DEFINED and might be UNKNOWN.

All other fields in the registers are set to an IMPLEMENTATION DEFINED value that can be UNKNOWN. The register is in the specified reset domain.

Note

An IMPLEMENTATION DEFINED reset value, which can be UNKNOWN, means that hardware is not required to reset the register on the specified reset, but software must not rely on the register being preserved over reset.

Table H8-7 Summary of external debug register resets, debug registers

Register	Reset domain	Field	Value	Description
DBGPRCR_EL1	Cold into AArch64 state	CORENPDRQ	The value of the powerup request[a]	Debug Power Control Register.
DBGPRCR	Cold into AArch32 state	CORENPDRQ	The value of the powerup request[a]	Debug Power Control Register.
EDESR, if FEAT_DoPD is not implemented	Warm	SS	EDECR.SS	Halting Step debug event pending
		RC	EDECR.RCE	Reset Catch debug event pending
		OSUC	0	OS Unlock Catch debug event pending

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Reset domain	Field	Value	Description
EDESR, if FEAT_DoPD is implemented	Cold	SS	0	Halting Step debug event pending
	Warm	RC	CTIDEVCTL.RCE	Reset Catch debug event pending
		OSUC	0	OS Unlock Catch debug event pending
EDECR, if FEAT_DoPD is implemented	Cold	SS	0	Halting Step debug event enable
EDECR, if FEAT_DoPD is not implemented	External debug	SS	0	Halting Step debug event enable
		RCE	0	Reset Catch debug event enable
		OSUCE	0	OS Unlock Catch debug event enable
EDWAR	Cold	-	-	All fields
EDSCR	Cold	RXfull	0	DTRRX register full
		TXfull	0	DTRTX register full
		RXO	0	DTRRX overrun
		TXU	0	DTRTX underrun
		INTdis	0	Interrupt disable
		TDA	0	Trap debug register accesses to Debug state
		MA	0	Memory access mode in Debug state
		HDE	0	Halting debug mode enable
		ERR	0	Cumulative error flag
EDSCR2	Cold	EHBWE	0	Extended Halting Breakpoint and Watchpoint Enable
		TTA	0	Trap Trace Accesses
EDECCR	Cold	NSE[2:1]	`0b00`	Coarse-grained Non-secure Exception Catch
		SE[3,1]	`0b00`	Coarse-grained Secure Exception Catch
EDPCSR	Cold	-	-	All fields
EDCIDSR	Cold	-	-	All fields
EDVIDSR	Cold	-	-	All fields
EDPRCR, if FEAT_DoPD is implemented	Cold	-	-	-
EDPRCR, if FEAT_DoPD is not implemented	External debug	COREPURQ[b]	-	Core powerup request
EDPRSR	Warm	SDR	-	Sticky debug restart
	Cold	SPMAD	0	Sticky EPMAD error
		SDAD	0	Sticky EDAD error
	Warm	SR	1	Sticky reset status
	Cold	SPD	1	Sticky powerdown status

a. If FEAT_DoPD is not implemented, the powerup request is the EDPRCR.COREPURQ control bit.

b. If FEAT_DoPD is not implemented, on a Cold reset into AArch64 state, DBGPRCR_EL1.CORENPDRQ resets to the value of EDPRCR.COREPURQ. On a Cold reset into AArch32 state, DBGPRCR.CORENPDRQ resets to the value of EDPRCR.COREPURQ. If an External Debug reset and a Cold reset coincide, both EDPRCR.COREPURQ and the CORENPDRQ field of the appropriate System register are

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

reset to 0.

Table H8-5 shows the reset values for the CTI registers

Table H8-8 Summary of external debug register resets, CTI registers

Register	Reset domain	Field	Value	Description
CTICONTROL	External debug	GLBEN	0	CTI global enable
CTIDEVCTL	External debug	RCE	0	If FEAT_DoPD is implemented, Reset Catch debug event enable
		OSUCE	0	If FEAT_DoPD is implemented, OS Unlock Catch debug event enable
CTIAPPSET	External debug	-	-	All fields
CTIINEN<n>	External debug	-	-	All fields
CTIOUTEN<n>	External debug	-	-	All fields
CTIGATE	External debug	-	-	All fields
ASICCTL	IMPLEMENTATION DEFINED	-	IMPLEMENTATION DEFINED	All of register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter H9
External Debug Register Descriptions

This chapter provides a description of the external debug registers. It contains the following sections:

- About the external debug registers.
- External debug registers.
- External trace registers.
- External Trace Buffer registers.
- Cross-Trigger Interface registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.1 About the external debug registers

The following sections describe the registers that are accessible through the external debug interface:

- External debug registers.
- External trace registers.
- External Trace Buffer registers.
- Cross-Trigger Interface registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2 External debug registers

This section describes the debug registers that are accessible through the external debug interface and are used for external debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.1 DBGAUTHSTATUS_EL1, Debug Authentication Status Register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configuration

When FEAT_DoPD is implemented, DBGAUTHSTATUS_EL1 is in the Core power domain. Otherwise, DBGAUTHSTATUS_EL1 is in the Debug power domain

External register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

External register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGAUTHSTATUS[31:0].

Attributes

DBGAUTHSTATUS_EL1 is a 32-bit register.

Field descriptions

31 28	27 26	25 24	23 16	15 14	13 12	11 8	7 6	5 4	3 2	1 0
RES0	RTNID	RTID	RES0	RLNID	RLID	RES0	SNID	SID	NSNID	NSID

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
0b00	Not implemented.
0b10	Implemented and disabled. ExternalRootInvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalRootInvasiveDebugEnabled() == TRUE.

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is 0b00.

Bits [23:16]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLID.

RLID, bits [13:12]

Realm invasive debug.

RLID	Meaning
`0b00`	Not implemented.
`0b10`	Implemented and disabled. `ExternalRealmInvasiveDebugEnabled()` == FALSE.
`0b11`	Implemented and enabled. `ExternalRealmInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is `0b00`.

Bits [11:8]

Reserved, RES0.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:

Secure non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise:

Secure non-invasive debug.

SNID	Meaning
`0b00`	Secure state is not implemented.
`0b10`	Implemented and disabled. `ExternalSecureNoninvasiveDebugEnabled()` == FALSE.
`0b11`	Implemented and enabled. `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. `ExternalSecureInvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalSecureInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

NSNID, bits [3:2]

When FEAT_Debugv8p4 is implemented:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b11	Implemented and enabled. `ExternalNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

Otherwise:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. `ExternalNoninvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalNoninvasiveDebugEnabled()` == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. `ExternalInvasiveDebugEnabled()` == FALSE.
0b11	Implemented and enabled. `ExternalInvasiveDebugEnabled()` == TRUE.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DBGAUTHSTATUS_EL1

DBGAUTHSTATUS_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFB8`	DBGAUTHSTATUS_EL1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configuration

DBGBCR<n>_EL1 is in the Core power domain

If breakpoint n is not implemented, then accesses to this register are:

- RES0 when `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalDebugAccess()`.
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

External register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

When FEAT_Debugv8p9 is implemented, External register DBGBCR<n>_EL1 bits [63:32] are architecturally mapped to AArch64 System register DBGBCR<n>_EL1[63:32].

External register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBCR<n>[31:0].

Attributes

DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions

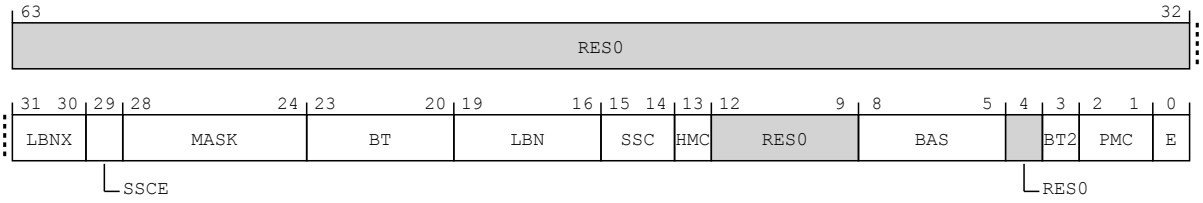

When the E field is zero, all the other fields in the register are ignored.

Bits [63:32]

Reserved, RES0.

LBNX, bits [31:30]

When FEAT_Debugv8p9 is implemented:

Linked Breakpoint Number.

For Linked address matching breakpoints, with DBGBCR<n>_EL1.LBN, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

This field extends DBGBCR<n>_EL1.LBN to support up to 64 implemented breakpoints.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

SSCE, bit [29]

When FEAT_RME is implemented:

Security State Control Extended.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MASK, bits [28:24]

When FEAT_BWE is implemented:

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
`0b00000`	No mask.
`0b00011..0b11111`	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

If any of the following apply, then the behavior of the breakpoint is CONSTRAINED UNPREDICTABLE:

- DBGBCR<n>_EL1.MASK is a reserved value.
- DBGBCR<n>_EL1.MASK is zero, DBGBCR<n>_EL1.{BT2, BT} is {0, `0b010x`} (address mismatch breakpoint without linking enabled), and AArch32 is not supported at EL1.
- DBGBCR<n>_EL1.MASK is a valid nonzero value and any of the following apply:
  - DBGBCR<n>_EL1.BAS is not `0b1111`, and AArch32 is supported at EL0.
  - DBGBVR<n>_EL1[(MASK-1):0] is nonzero.
  - DBGBCR<n>_EL1.{BT2, BT} is {0, `0b0x1x`} or {0, `0b1xxx`} (Context matching breakpoint).

When any of these conditions apply, the breakpoint behaves as if one of the following:

- DBGBCR<n>_EL1.MASK has been programmed with a defined value, which might be `0b00000` (no mask), other than for a direct read of DBGBCR<n>_EL1.
- The breakpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BT, bits [23:20]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Breakpoint Type.

With DBGBCR<n>_EL1.BT2 when implemented, specifies breakpoint type.

BT	Meaning	Applies when
`0b0000`	Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.	
`0b0001`	Linked instruction address match. As `0b0000`, but linked to a breakpoint that has linking enabled.	
`0b0010`	Unlinked Context ID match. If the Effective value of HCR_EL2.E2H is 1 and either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL2. Otherwise, DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1.	breakpoint n is context-aware
`0b0011`	As `0b0010`, with linking enabled.	breakpoint n is context-aware
`0b0100`	Unlinked instruction address mismatch. DBGBVR<n>_EL1 is the address of an instruction.	FEAT_BWE is implemented or EL1 is using AArch32
`0b0101`	Linked instruction address mismatch. As `0b0100`, but linked to a breakpoint that has linking enabled.	FEAT_BWE is implemented or EL1 is using AArch32
`0b0110`	Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
`0b0111`	As `0b0110`, with linking enabled.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
`0b1000`	Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	EL2 is implemented and breakpoint n is context-aware
`0b1001`	As `0b1000`, with linking enabled.	EL2 is implemented and breakpoint n is context-aware
`0b1010`	Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	EL2 is implemented and breakpoint n is context-aware
`0b1011`	As `0b1010`, with linking enabled.	EL2 is implemented and breakpoint n is context-aware
`0b1100`	Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
`0b1101`	As `0b1100`, with linking enabled.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware
`0b1110`	Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning	Applies when
`0b1111`	As `0b1110`, with linking enabled.	EL2 is implemented, FEAT_Debugv8p1 is implemented, and breakpoint n is context-aware

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked address matching breakpoints, with DBGBCR<n>_EL1.LBNX when implemented, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see 'Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values'.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

When FEAT_AA32 is implemented:

Byte address select. Defines which halfwords an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS	Match instruction at	Constraint for debuggers
0b0011	DBGBVR<n>_EL1	Use for T32 instructions.
0b1100	DBGBVR<n>_EL1 + 2	Use for T32 instructions.
0b1111	DBGBVR<n>_EL1	Use for A64 and A32 instructions.

All other values are reserved.

For more information, see 'Using the BAS field in Address Match breakpoints'.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0000	•	Use for a match anywhere breakpoint.
0b0011	DBGBVR<n>_EL1	Use for stepping T32 instructions.
0b1100	DBGBVR<n>_EL1 + 2	Use for stepping T32 instructions.
0b1111	DBGBVR<n>_EL1	Use for stepping A64 and A32 instructions.

All other values are reserved.

For more information, see 'Using the BAS field in Address Match breakpoints'.

For Context matching breakpoints, this field is RES1 and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Bit [4]

Reserved, RES0.

BT2, bit [3]

When FEAT_ABLE is implemented and breakpoint n supports address breakpoint linking:

Breakpoint Type 2. With DBGBCR<n>_EL1.BT, specifies breakpoint type.

BT2	Meaning
0b0	As DBGBCR<n>_EL1.BT.
0b1	As DBGBCR<n>_EL1.BT, but with linking enabled. This value is only defined for the following DBGBCR<n>_EL1.BT values: 0b0000, 0b0001, 0b0100, and 0b0101. All other values are reserved.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint n.

E	Meaning
`0b0`	Breakpoint n disabled.
`0b1`	Breakpoint n enabled.

This field is ignored by the PE and treated as zero when all of the following are true:

- Any of the following are true:
  - `HaltOnBreakpointOrWatchpoint()` is FALSE and the Effective value of MDSCR_EL1.EMBWE is 0.
  - `HaltOnBreakpointOrWatchpoint()` is TRUE and the Effective value of EDSCR2.EHBWE is 0.
- FEAT_Debugv8p9 is implemented.
- n >= 16.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>_EL1

When FEAT_Debugv8p9 is not implemented, this register is 32-bits wide and offset `0x40C` + (16 * n) is reserved.

DBGBCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x408 + (16 * n)`	DBGBCR<n>_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 63

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

Configuration

DBGBVR<n>_EL1 is in the Core power domain

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

- When DBGBCR<n>_EL1.BT is `0b0x0x`, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is `0b001x`, `0b011x`, or `0b110x`, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is `0b100x`, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is `0b101x`, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is `0b111x`, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

If breakpoint n is not implemented, then accesses to this register are:

- RES0 when `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalDebugAccess()`.
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

External register DBGBVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:0].

External register DBGBVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBVR<n>[31:0].

External register DBGBVR<n>_EL1 bits [63:32] are architecturally mapped to AArch32 System register DBGBXVR<n>[31:0].

Attributes

DBGBVR<n>_EL1 is a 64-bit register.

Field descriptions

When DBGBCR<n>_EL1.BT IN {'`0x0`x'}:

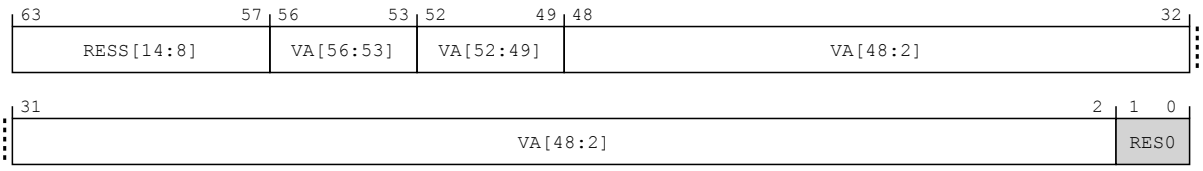

RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [56:53]

When FEAT_LVA3 is implemented:

VA[56:53], bits [56:53]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

If the address is being matched in an AArch64 stage 1 translation regime:

- This field contains bits[48:2] of the address for comparison.
- When FEAT_LVA3 is implemented, (VA[56:53]:VA[52:49]) forms the upper part of the address value. If FEAT_LVA3 is not implemented, bits VA[56:53] are part of the RESS field.
- When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

If the address is being matched in an AArch32 stage 1 translation regime, the first 20 bits of this field are RES0, and the rest of the field contains bits[31:2] of the address for comparison.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'001x'}:

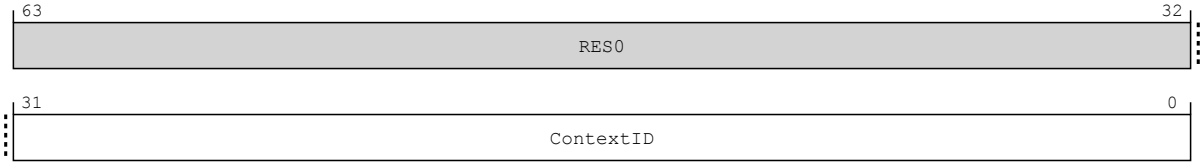

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when the Effective value of HCR_EL2.E2H is 1, and either:

- The PE is executing at EL2.
- HCR_EL2.TGE is 1, the PE is executing at EL0, and EL2 is enabled in the current Security state.

Otherwise, the value is compared against the following:

- CONTEXTIDR when the PE is executing at AArch32.
- CONTEXTIDR_EL1 when the PE is executing at AArch64.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'011x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

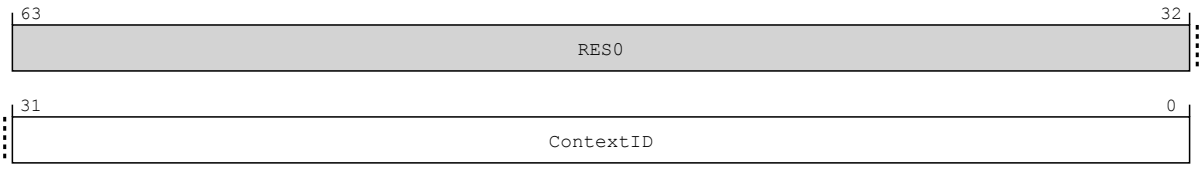

Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'100x'} and EL2 is implemented:

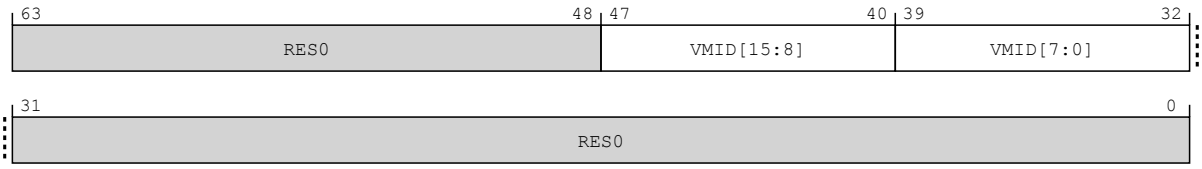

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

If EL2 is using AArch32, this field is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'101x'} and EL2 is implemented:

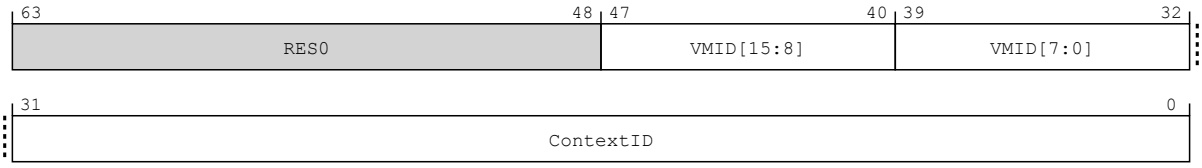

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

If EL2 is using AArch32, or if the implementation has an 8-bit VMID, this field is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'110x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

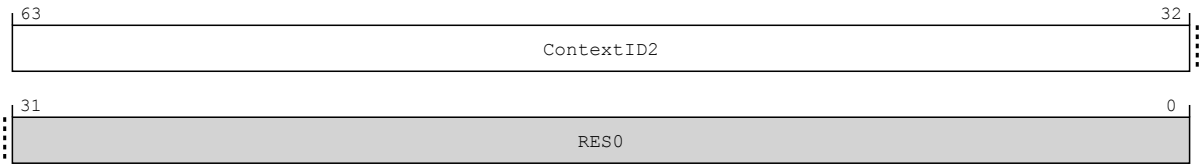

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'111x'}, EL2 is implemented, and FEAT_Debugv8p1 is implemented:

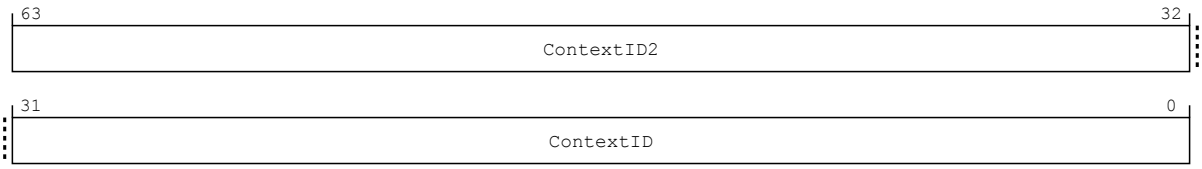

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DBGBVR<n>_EL1

DBGBVR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x400 + (16 * n)`	DBGBVR<n>_EL1	63:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear Register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET_EL1 register.

Configuration

DBGCLAIMCLR_EL1 is in the Core power domain

An implementation must include eight CLAIM tag bits.

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

Attributes

DBGCLAIMCLR_EL1 is a 32-bit register.

Field descriptions

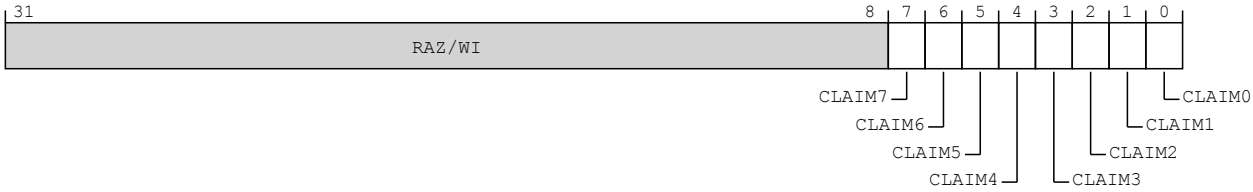

Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m>	Meaning
`0b0`	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
`0b1`	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is W1C.

Accessing DBGCLAIMCLR_EL1

DBGCLAIMCLR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFA4`	DBGCLAIMCLR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set Register

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

Configuration

DBGCLAIMSET_EL1 is in the Core power domain

An implementation must include eight CLAIM tag bits.

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

Attributes

DBGCLAIMSET_EL1 is a 32-bit register.

Field descriptions

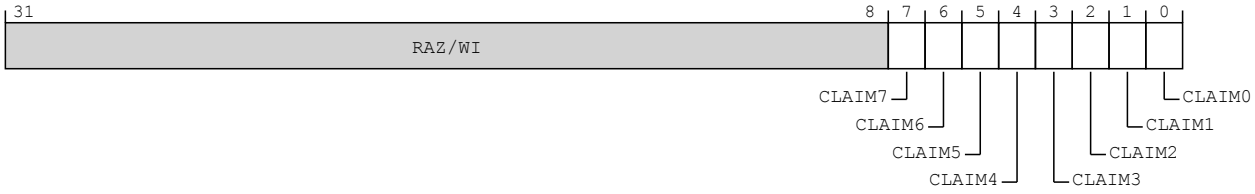

Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m>	Meaning
`0b0`	On a write: Ignored.
`0b1`	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/W1S.

Accessing DBGCLAIMSET_EL1

DBGCLAIMSET_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFA0`	DBGCLAIMSET_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.6 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configuration

DBGDTRRX_EL0 is in the Core power domain

External register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

External register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

Attributes

DBGDTRRX_EL0 is a 32-bit register.

Field descriptions

31	0
Update DTRRX	

DTRRX, bits [31:0]

Update DTRRX.

Writes to this register:

- If RXfull is 0, update the value in DTRRX and set RXfull to 1.
- If RXfull is 1, the written value is ignored and RXO is set to 1.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see ‘The Debug Communication Channel and Instruction Transfer Register’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRX_EL0

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in Memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTRRX_EL0 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0x080`	DBGDTRRX_EL0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.7 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configuration

DBGDTRTX_EL0 is in the Core power domain

External register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

External register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

Attributes

DBGDTRTX_EL0 is a 32-bit register.

Field descriptions

31	0
Return DTRTX	

DTRTX, bits [31:0]

Return DTRTX.

Reads of this register:

- If TXfull is 1, return the value in DTRTX and clear TXfull to 0.
- If TXfull is 0, return an UNKNOWN value and set TXU to 1.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see ‘The Debug Communication Channel and Instruction Transfer Register’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTX_EL0

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in Memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTRTX_EL0 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0x08C`	DBGDTRTX_EL0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.8 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 63

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configuration

DBGWCR<n>_EL1 is in the Core power domain

If watchpoint n is not implemented, then accesses to this register are:

- When `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalDebugAccess()`, RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

External register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

When FEAT_Debugv8p9 is implemented, External register DBGWCR<n>_EL1 bits [63:32] are architecturally mapped to AArch64 System register DBGWCR<n>_EL1[63:32].

External register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions

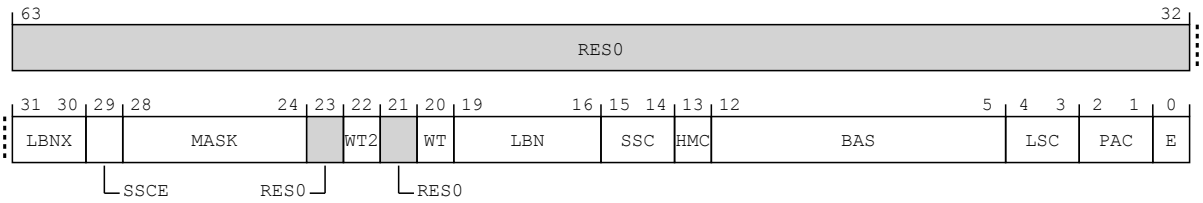

Bits [63:32]

Reserved, RES0.

LBNX, bits [31:30]

When FEAT_Debugv8p9 is implemented:

Linked Breakpoint Number.

For Linked data address watchpoints, with DBGWCR<n>_EL1.LBN, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

This field extends DBGWCR<n>_EL1.LBN to support up to 64 implemented breakpoints.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SSCE, bit [29]

When FEAT_RME is implemented:

Security State Control Extended.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
`0b00000`	No mask.
`0b00011..0b11111`	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

If programmed with a reserved value, the watchpoint behaves as if one of the following:

- DBGWCR<n>_EL1.MASK has been programmed with a defined value, which might be `0b00000` (no mask), other than for a direct read of DBGWCR<n>_EL1.
- The watchpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [23]

Reserved, RES0.

WT2, bit [22]

When FEAT_BWE2 is implemented:

Watchpoint Type 2. With DBGWCR<n>_EL1.WT, specifies watchpoint type.

WT2	Meaning
`0b0`	Watchpoint n is an address match watchpoint.
`0b1`	Watchpoint n is an address mismatch watchpoint.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked data address match.
0b1	Linked data address match.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked data address watchpoints, with DBGWCR<n>_EL1.LBNX when implemented, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or doubleword addressed by DBGWVR<n>_EL1 is being watched.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

In cases where DBGWVR<n>_EL1 addresses a doubleword:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] is used. Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are nonzero binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software. See 'Reserved DBGWCR<n>.BAS values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
`0b01`	Match instructions that load from a watchpointed address.
`0b10`	Match instructions that store to a watchpointed address.
`0b11`	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n.

E	Meaning
`0b0`	Watchpoint n disabled.
`0b1`	Watchpoint n enabled.

This field is ignored by the PE and treated as zero when all of the following are true:

- Any of the following are true:
  - `HaltOnBreakpointOrWatchpoint()` is FALSE and the Effective value of MDSCR_EL1.EMBWE is 0.
  - `HaltOnBreakpointOrWatchpoint()` is TRUE and the Effective value of EDSCR2.EHBWE is 0.
- FEAT_Debugv8p9 is implemented.
- n >= 16.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>_EL1

When FEAT_Debugv8p9 is not implemented, this register is 32-bits wide and offset `0x80C` + (16 * n) is reserved.

DBGWCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x808 + (16 * n)`	DBGWCR<n>_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.9 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 63

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configuration

DBGWVR<n>_EL1 is in the Core power domain

If watchpoint n is not implemented, then accesses to this register are:

- When `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalDebugAccess()`, RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

External register DBGWVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGWVR<n>_EL1[63:0].

External register DBGWVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWVR<n>[31:0].

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions

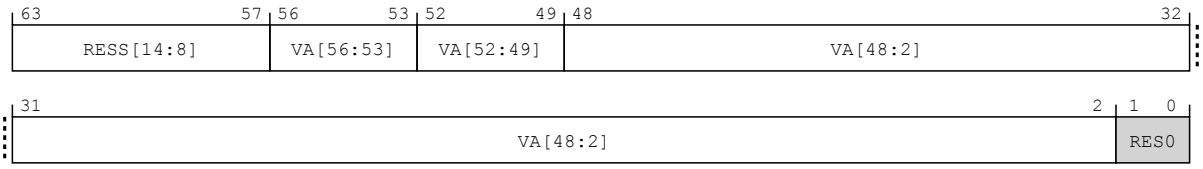

RESS[14:8], bits [63:57]

Reserved, Sign extended. Hardware and software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

Bits [56:53]

When FEAT_LVA3 is implemented:

VA[56:53], bits [56:53]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA3 is implemented, (VA[56:53]:VA[52:49]) forms the upper part of the address value. If FEAT_LVA3 is not implemented, bits VA[56:53] are part of the RESS field.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>_EL1

DBGWVR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x800 + (16 * n)`	DBGWVR<n>_EL1	63:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.10 EDAA32PFR, External Debug Auxiliary Processor Feature Register

The EDAA32PFR characteristics are:

Purpose

Provides information about implemented PE features.

Note

The register mnemonic, EDAA32PFR, is derived from previous definitions of this register that defined this register only when AArch64 was not supported.

For general information about the interpretation of the ID registers, see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

EDAA32PFR is a 64-bit register.

Field descriptions

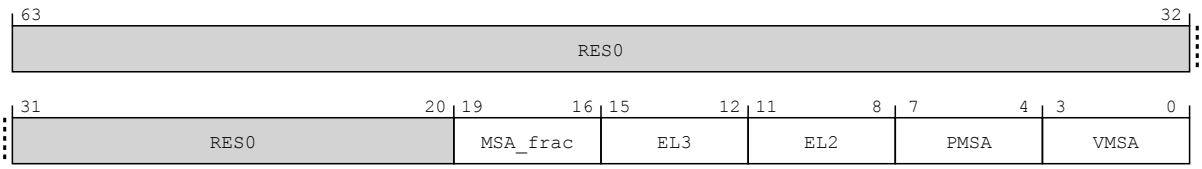

Bits [63:20]

Reserved, RES0.

MSA_frac, bits [19:16]

When EDAA32PFR.PMSA == '0000' and EDAA32PFR.VMSA == '1111':

Memory System Architecture fractional field. This holds the information on additional Memory System Architectures supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MSA_frac	Meaning
0b0001	PMSAv8-64 supported in all translation regimes. VMSAv8-64 not supported.
0b0010	PMSAv8-64 supported in all translation regimes. In addition to PMSAv8-64, stage 1 EL1&0 translation regime also supports VMSAv8-64.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EL3, bits [15:12]

When EDPFR.EL3 == '0000':

AArch32 EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented or can be executed in AArch64 state.
0b0001	EL3 can be executed in AArch32 state only.

All other values are reserved.

Note

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

EL2, bits [11:8]

When EDPFR.EL2 == '0000':

AArch32 EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented or can be executed in AArch64 state.
0b0001	EL2 can be executed in AArch32 state only.

All other values are reserved.

Note

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

PMSA, bits [7:4]

Indicates support for a 32-bit PMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSA	Meaning
0b0000	PMSA-32 not supported.
0b0100	PMSAv8-32 supported.

In Armv8-A, the only permitted value is 0b0000. All other values are reserved.

Access to this field is RO.

VMSA, bits [3:0]

When EDAA32PFR.PMSA != '0000':

Indicates support for a VMSA in addition to a 32-bit PMSA.

VMSA	Meaning
0b0000	VMSA not supported.

All other values are reserved.

Access to this field is RO.

When EDAA32PFR.PMSA == '0000':

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMSA	Meaning
0b0000	VMSAv8-64 supported.
0b1111	Memory system architecture described by EDAA32PFR.MSA_frac.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDAA32PFR

EDAA32PFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD60	EDAA32PFR

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.11 EDACR, External Debug Auxiliary Control Register

The EDACR characteristics are:

Purpose

Allows implementations to support IMPLEMENTATION DEFINED controls.

Configuration

When FEAT_DoPD is implemented, EDACR is in the Core power domain. Otherwise, it is IMPLEMENTATION DEFINED whether EDACR is implemented in the Core power domain or in the Debug power domain

If this register is implemented, EDDEVID.AuxRegs == `0b0001`.

If FEAT_DoPD is implemented, any mechanism to preserve control bits in EDACR over power down is optional and IMPLEMENTATION DEFINED.

If FEAT_DoPD is not implemented and EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and when the Core is powered off.

Changing this register from its reset value causes IMPLEMENTATION DEFINED behavior, including possible deviation from the architecturally-defined behavior.

Implementation of this register is OPTIONAL.

Attributes

EDACR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to an architecturally UNKNOWN value.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to an architecturally UNKNOWN value.
  - On a Warm reset, the value of this field is unchanged.

Accessing EDACR

EDACR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x094`	EDACR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.12 EDCIDR0, External Debug Component Identification Register 0

The EDCIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see 'About the Component Identification scheme'.

Configuration

When FEAT_DoPD is implemented, EDCIDR0 is in the Core power domain. Otherwise, EDCIDR0 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR0 is a 32-bit register.

Field descriptions

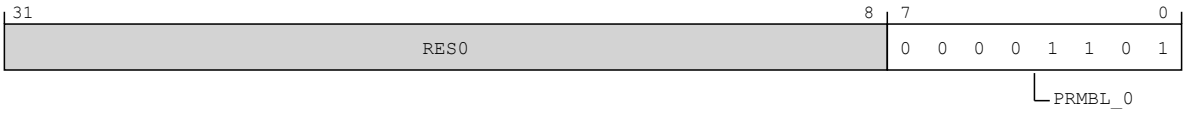

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing EDCIDR0

EDCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF0`	EDCIDR0

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.13 EDCIDR1, External Debug Component Identification Register 1

The EDCIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see 'About the Component Identification scheme'.

Configuration

When FEAT_DoPD is implemented, EDCIDR1 is in the Core power domain. Otherwise, EDCIDR1 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR1 is a 32-bit register.

Field descriptions

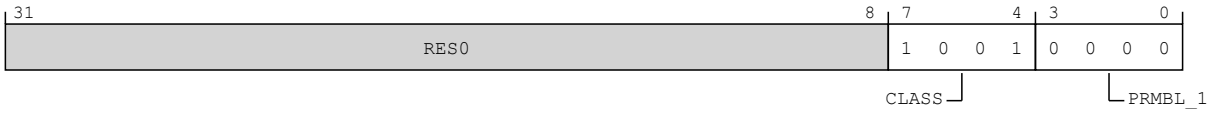

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing EDCIDR1

EDCIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF4`	EDCIDR1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.14 EDCIDR2, External Debug Component Identification Register 2

The EDCIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Component Identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDCIDR2 is in the Core power domain. Otherwise, EDCIDR2 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR2 is a 32-bit register.

Field descriptions

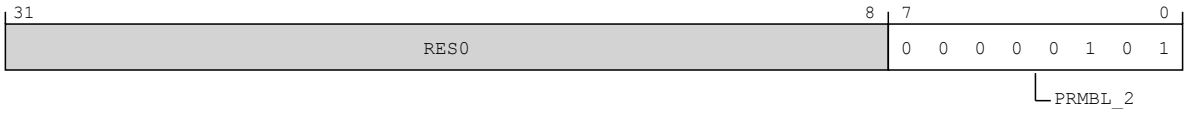

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing EDCIDR2

EDCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF8`	EDCIDR2

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.15 EDCIDR3, External Debug Component Identification Register 3

The EDCIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Component Identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDCIDR3 is in the Core power domain. Otherwise, EDCIDR3 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR3 is a 32-bit register.

Field descriptions

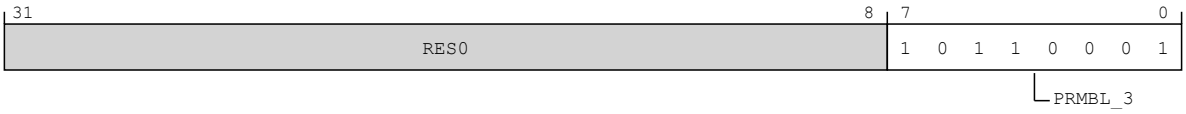

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing EDCIDR3

EDCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFFC`	EDCIDR3

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.16 EDCIDSR, External Debug Context ID Sample Register

The EDCIDSR characteristics are:

Purpose

Contains the sampled value of the Context ID, captured on reading EDPCSR[31:0].

Configuration

EDCIDSR is in the Core power domain

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDCIDSR are RES0.

Attributes

EDCIDSR is a 32-bit register.

Field descriptions

31	0
CONTEXTIDR	

CONTEXTIDR, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent EDPCSR sample. When the most recent EDPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register, and EDCIDSR samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent EDPCSR sample.

Because the value written to EDCIDSR is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether EDCIDSR is set to the original or new value if EDPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDCIDSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

EDCIDSR can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0x0A4`	EDCIDSR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.17 EDDEVAFF0, External Debug Device Affinity register 0

The EDDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

Configuration

When FEAT_DoPD is implemented, EDDEVAFF0 is in the Core power domain. Otherwise, EDDEVAFF0 is in the Debug power domain

Attributes

EDDEVAFF0 is a 32-bit register.

Field descriptions

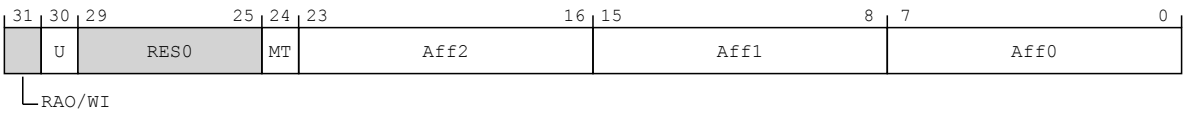

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDEVAFF0

EDDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFA8	EDDEVAFF0

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.18 EDDEVAFF1, External Debug Device Affinity register 1

The EDDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

Configuration

When FEAT_DoPD is implemented, EDDEVAFF1 is in the Core power domain. Otherwise, EDDEVAFF1 is in the Debug power domain

Attributes

EDDEVAFF1 is a 32-bit register.

Field descriptions

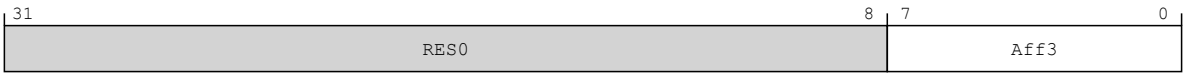

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of EDDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDEVAFF1

EDDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFAC`	EDDEVAFF1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.19 EDDEVARCH, External Debug Device Architecture Register

The EDDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the external debug component.

Configuration

When FEAT_DoPD is implemented, EDDEVARCH is in the Core power domain. Otherwise, EDDEVARCH is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

EDDEVARCH is a 32-bit register.

Field descriptions

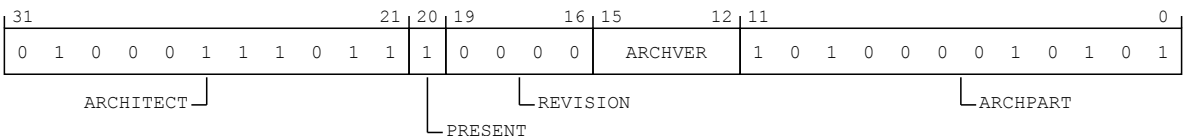

ARCHITECT, bits [31:21]

Defines the architect of the component. For External Debug, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the EDDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For debug, the revision defined by Armv8 is `0x0`.

All other values are reserved.

Reads as `0b0000`

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARCHVER	Meaning
0b0110	Armv8 debug architecture.
0b0111	Armv8.1 debug architecture, FEAT_Debugv8p1.
0b1000	Armv8.2 debug architecture, FEAT_Debugv8p2.
0b1001	Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010	Armv8.8 debug architecture, FEAT_Debugv8p8.
0b1011	Armv8.9 debug architecture, FEAT_Debugv8p9.

EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHVER is EDDEVARCH.ARCHID[15:12].

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

FEAT_Debugv8p8 implements the functionality identified by the value 0b1010.

FEAT_Debugv8p9 implements the functionality identified by the value 0b1011.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

From Armv8.8, the value 0b1001 is not permitted.

From Armv8.9, the value 0b1010 is not permitted.

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA15	Armv8-A debug architecture.

EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHPART is EDDEVARCH.ARCHID[11:0].

Armv8-A debug architecture.

Access to this field is RO.

Accessing EDDEVARCH

EDDEVARCH can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0xFBC`	EDDEVARCH

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.20 EDDEVID, External Debug Device ID register 0

The EDDEVID characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configuration

When FEAT_DoPD is implemented, EDDEVID is in the Core power domain. Otherwise, EDDEVID is in the Debug power domain

Attributes

EDDEVID is a 32-bit register.

Field descriptions

31 28	27 24	23 8	7 4	3 0
RES0	AuxRegs	RES0	DebugPower	PCSample

Bits [31:28]

Reserved, RES0.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxRegs	Meaning
0b0000	None supported.
0b0001	Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

Access to this field is RO.

Bits [23:8]

Reserved, RES0.

DebugPower, bits [7:4]

Indicates support for the FEAT_DoPD feature.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DebugPower	Meaning
0b0000	FEAT_DoPD not implemented. Registers in the external debug interface register map are implemented in a mix of the Debug and Core power domains.
0b0001	FEAT_DoPD implemented. All registers in the external debug interface register map are implemented in the Core power domain.

FEAT_DoPD implements the functionality added by the value 0b0001.

All other values are reserved.

Access to this field is RO.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the external debug registers space.
0b0010	Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.
0b0011	EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Access to this field is RO.

Accessing EDDEVID

EDDEVID can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC8	EDDEVID

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.21 EDDEVID1, External Debug Device ID Register 1

The EDDEVID1 characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configuration

When FEAT_DoPD is implemented, EDDEVID1 is in the Core power domain. Otherwise, EDDEVID1 is in the Debug power domain

Attributes

EDDEVID1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	HSR	PCSROffset

Bits [31:8]

Reserved, RES0.

HSR, bits [7:4]

Indicates support for the External Debug Halt Status Register, EDHSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HSR	Meaning
0b0000	EDHSR not implemented, and the PE follows behaviors consistent with all of the EDHSR fields having a zero value.
0b0001	EDHSR implemented.
0b0010	As 0b0001, but extends EDHSR to include the VNCR, CM, and WnR fields.

All other values are reserved.

FEAT_EDHSR implements the functionality identified by the value 0b0001.

FEAT_Debugv8p9 implements the functionality identified by the value 0b0010.

When FEAT_Debugv8p2 is not implemented, the only permitted value is 0b0000.

From Armv8.9, the values 0b0000 and 0b0001 are not permitted.

Access to this field is RO.

PCSROffset, bits [3:0]

Indicates the offset applied to PC samples returned by reads of EDPCSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCSROffset	Meaning
0b0000	EDPCSR not implemented.
0b0010	EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Access to this field is RO.

Accessing EDDEVID1

EDDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC4	EDDEVID1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.22 EDDEVID2, External Debug Device ID register 2

The EDDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

Configuration

When FEAT_DoPD is implemented, EDDEVID2 is in the Core power domain. Otherwise, EDDEVID2 is in the Debug power domain

Attributes

EDDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing EDDEVID2

EDDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFC0`	EDDEVID2

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.23 EDDEVTYPE, External Debug Device Type register

The EDDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's debug logic.

Configuration

When FEAT_DoPD is implemented, EDDEVTYPE is in the Core power domain. Otherwise, EDDEVTYPE is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

EDDEVTYPE is a 32-bit register.

Field descriptions

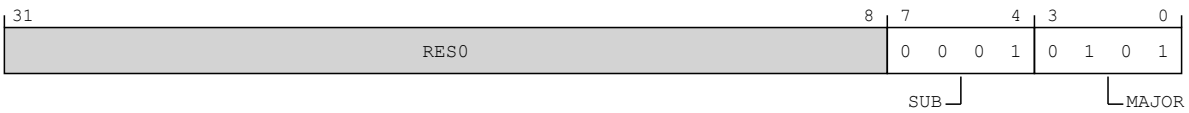

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type. Indicates this is a debug logic component.

Reads as `0b0101`

Access to this field is RO.

Accessing EDDEVTYPE

EDDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFCC`	EDDEVTYPE

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.24 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

Purpose

Provides top-level information about the debug system.

Note

Debuggers must use EDDEVARCH to determine the Debug architecture version.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

There are no configuration notes.

Attributes

EDDFR is a 64-bit register.

Field descriptions

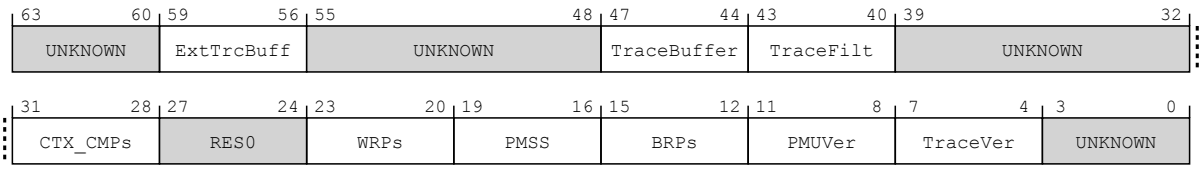

Bits [63:60]

Reserved, UNKNOWN.

ExtTrcBuff, bits [59:56]

Trace Buffer External Mode Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ExtTrcBuff	Meaning
`0b0000`	Trace Buffer Extension not implemented or Trace Buffer External Mode not implemented.
`0b0001`	Trace Buffer Extension implemented and Trace Buffer External Mode implemented.

All other values are reserved.

If FEAT_TRBE is not implemented, the only permitted value is `0b0000`.

FEAT_TRBE_EXT implements the functionality identified by the value `0b0001`.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR0_EL1.ExtTrcBuff.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [55:48]

Reserved, UNKNOWN.

TraceBuffer, bits [47:44]

When FEAT_TRBE_EXT is implemented:

Trace Buffer Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceBuffer	Meaning
0b0000	Trace Buffer Extension not implemented.
0b0001	Trace Buffer Extension implemented.
0b0010	As 0b0001, and adds: • If EL2 and FEAT_FGT are implemented, a fine-grained trap on the TSB CSYNC instruction. • If EL2 is implemented, an EL2 control to override TRBLIMITR_EL1.nVM. • The TRBE Profiling exception extension, FEAT_TRBE_EXC.

All other values are reserved.

FEAT_TRBE implements the functionality identified by the value 0b0001.

FEAT_TRBEv1p1 implements the functionality identified by the value 0b0010.

In any Armv9 implementation, if FEAT_ETE is implemented, the value 0b0000 is not permitted.

From Armv9.6, if FEAT_TRBE is implemented, the value 0b0001 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, UNKNOWN.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceFilt	Meaning
0b0000	Armv8.4 Self-hosted Trace Extension not implemented.
0b0001	Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_ETMv4 is implemented, the value 0b0000 is not permitted.

If FEAT_ETE is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [39:32]

Reserved, UNKNOWN.

CTX_CMPs, bits [31:28]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
`0b0000..0b1111`	The number of context-aware breakpoints, minus 1.

The value of this field is never greater than EDDFR.BRPs.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR0_EL1.CTX_CMPs.

If FEAT_Debugv8p9 is implemented and 16 or more context-aware breakpoints are implemented, then this field reads as `0b1111` and EDDFR1.CTX_CMPs indicates the number of context-aware breakpoints.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 breakpoints.

Access to this field is RO.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
`0b0001..0b1111`	The number of watchpoints, minus 1.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR0_EL1.WRPs.

If FEAT_Debugv8p9 is implemented and 16 or more watchpoints are implemented, then this field reads as `0b1111` and EDDFR1.WRPs indicates the number of watchpoints.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 watchpoints.

The value `0b0000` is reserved.

Access to this field is RO.

PMSS, bits [19:16]

This field either has the same value as ID_AA64DFR0_EL1.PMSS or reads as zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

BRPs, bits [15:12]

Number of breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
`0b0001..0b1111`	The number of breakpoints, minus 1.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR0_EL1.BRPs.

If FEAT_Debugv8p9 is implemented and 16 or more breakpoints are implemented, then this field reads as `0b1111` and EDDFR1.BRPs indicates the number of breakpoints.

Note

If AArch32 is supported at EL1, then the PE does not implement more than 16 breakpoints.

The value `0b0000` is reserved.

Access to this field is RO.

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMUVer	Meaning
`0b0000`	Performance Monitors Extension not implemented.
`0b0001`	Performance Monitors Extension, PMUv3 implemented.
`0b0100`	PMUv3 for Armv8.1. As `0b0001`, and adds support for: • Extended 16-bit PMEVTYPER<n>_EL0.evtCount field. • If EL2 is implemented, the MDCR_EL2.HPMD control.
`0b0101`	PMUv3 for Armv8.4. As `0b0100`, and adds support for the PMMIR_EL1 register.
`0b0110`	PMUv3 for Armv8.5. As `0b0101`, and adds support for: • 64-bit event counters. • If EL2 is implemented, the MDCR_EL2.HCCD control. • If EL3 is implemented, the MDCR_EL3.SCCD control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMUVer	Meaning
0b0111	PMUv3 for Armv8.7. As 0b0110, and adds support for: • The PMCR_EL0.FZO and, if EL2 is implemented, MDCR_EL2.HPMFZO controls. • If EL3 is implemented, the MDCR_EL3.{MPMX,MCCD} controls.
0b1000	PMUv3 for Armv8.8. As 0b0111, and: • Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF. • Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.
0b1001	PMUv3 for Armv8.9. As 0b1000, and: • Updates the definitions of existing PMU events. • Adds support for the PMUSERENR_EL0.UEN control and the PMUACR_EL1 register. • Adds support for the EDECR.PME control.
0b1111	IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0001.

FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.

FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.

FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.

FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.

FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

FEAT_PMUv3p9 implements the functionality identified by the value 0b1001.

From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0001 is not permitted.

From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.

From Armv8.5, if FEAT_PMUv3 is implemented, the value 0b0101 is not permitted.

From Armv8.7, if FEAT_PMUv3 is implemented, the value 0b0110 is not permitted.

From Armv8.8, if FEAT_PMUv3 is implemented, the value 0b0111 is not permitted.

From Armv8.9, if FEAT_PMUv3 is implemented, the value 0b1000 is not permitted.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR0_EL1.PMUVer.

Access to this field is RO.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace unit is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceVer	Meaning
0b0000	Trace unit System registers not implemented.
0b0001	Trace unit System registers implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

A value of `0b0000` only indicates that no System register interface to a trace unit is implemented. A trace unit might nevertheless be implemented without a System register interface.

In an implementation that supports AArch64, this field returns the value of ID_AA64DFR0_EL1.TraceVer.

Access to this field is RO.

Bits [3:0]

Reserved, UNKNOWN.

Accessing EDDFR

EDDFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xD28`	EDDFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.25 EDDFR1, External Debug Feature Register 1

The EDDFR1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64.

Configuration

There are no configuration notes.

Attributes

EDDFR1 is a 64-bit register.

Field descriptions

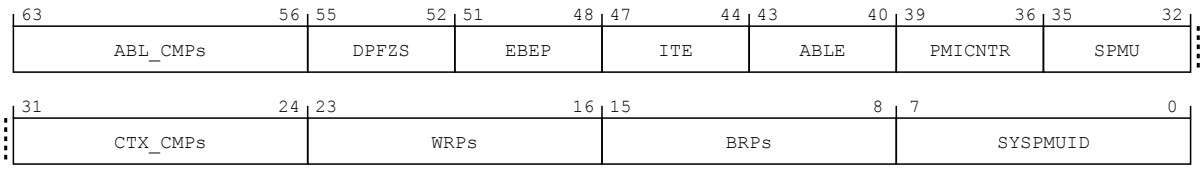

ABL_CMPs, bits [63:56]

When FEAT_ABLE is implemented:

Number of breakpoints that support address linking, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ABL_CMPs	Meaning
`0x00..0x3F`	Number of breakpoints that support address linking minus 1.

All other values are reserved.

The number of breakpoints that support address linking is never more than either the number of breakpoints or the number of watchpoints.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR1_EL1.ABL_CMPs.

Access to this field is RO.

Otherwise:

Reserved, RES0.

DPFZS, bits [55:52]

This field either has the same value as ID_AA64DFR1_EL1.DPFZS or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EBEP, bits [51:48]

This field either has the same value as ID_AA64DFR1_EL1.EBEP or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

ITE, bits [47:44]

This field either has the same value as ID_AA64DFR1_EL1.ITE or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

ABLE, bits [43:40]

Address Breakpoint Linking Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ABLE	Meaning
`0b0000`	Address Breakpoint Linking Extension not implemented.
`0b0001`	Address Breakpoint Linking Extension implemented.

All other values are reserved.

FEAT_BWE implements the address range breakpoints and mismatch breakpoints part of the functionality identified by the value `0b0001`.

FEAT_ABLE implements the functionality identified by the value `0b0001`.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR1_EL1.ABLE.

Access to this field is RO.

PMICNTR, bits [39:36]

This field either has the same value as ID_AA64DFR1_EL1.PMICNTR or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SPMU, bits [35:32]

This field either has the same value as ID_AA64DFR1_EL1.SPMU or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CTX_CMPs, bits [31:24]

Context-aware breakpoints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CTX_CMPs	Meaning
`0x00`	EDDFR.CTX_CMPs is the number of context-aware breakpoints, minus 1.
`0x01..0x3F`	Number of context-aware breakpoints minus 1.

All other values are reserved.

The value of this field is never greater than EDDFR1.BRPs.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR1_EL1.CTX_CMPs.

Access to this field is RO.

WRPs, bits [23:16]

Watchpoints.

WRPs	Meaning
`0x00`	EDDFR.WRPs is the number of watchpoints, minus 1.
`0x01..0x3F`	Number of watchpoints minus 1.

All other values are reserved.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR1_EL1.WRPs.

BRPs, bits [15:8]

Breakpoints.

BRPs	Meaning
`0x00`	EDDFR.BRPs is the number of breakpoints, minus 1.
`0x01..0x3F`	Number of breakpoints minus 1.

All other values are reserved.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR1_EL1.BRPs.

SYSPMUID, bits [7:0]

This field either has the same value as ID_AA64DFR1_EL1.SYSPMUID or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDFR1

EDDFR1 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0xD48`	EDDFR1

Accessible as follows:

- When (IsCorePowered() && (!DoubleLockStatus())) && ((!IsZero(EDDFR1())) || (!OSLockStatus())), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.26 EDDFR2, External Debug Feature Register 2

The EDDFR2 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64.

Configuration

There are no configuration notes.

Attributes

EDDFR2 is a 64-bit register.

Field descriptions

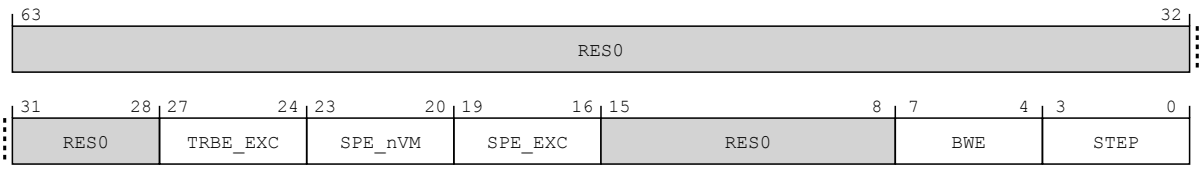

Bits [63:28]

Reserved, RES0.

TRBE_EXC, bits [27:24]

This field either has the same value as ID_AA64DFR2_EL1.TRBE_EXC or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SPE_nVM, bits [23:20]

This field either has the same value as ID_AA64DFR2_EL1.SPE_nVM or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SPE_EXC, bits [19:16]

This field either has the same value as ID_AA64DFR2_EL1.SPE_EXC or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [15:8]

Reserved, RES0.

BWE, bits [7:4]

Breakpoints and watchpoint enhancements.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BWE	Meaning
0b0000	This field does not indicate whether DBGBCR<n>_EL1.MASK and address mismatch breakpoints are implemented.
0b0001	DBGBCR<n>_EL1.MASK and address mismatch breakpoints are implemented.
0b0010	As 0b0001, and address mismatch watchpoints are implemented.

All other values are reserved.

FEAT_BWE implements the functionality identified by the value 0b0001.

FEAT_BWE2 implements the functionality identified by the value 0b0010.

When this field is 0b0000, ID_AA64DFR1_EL1.ABLE might indicate the presence of support for DBGBCR<n>_EL1.MASK and address mismatch breakpoints.

From Armv9.5, the value 0b0001 is not permitted.

In an implementation that supports AArch64, this field has the same value as ID_AA64DFR2_EL1.BWE.

Access to this field is RO.

STEP, bits [3:0]

This field either has the same value as ID_AA64DFR2_EL1.STEP or reads as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDFR2

EDDFR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD50	EDDFR2

Accessible as follows:

- When (IsCorePowered() && (!DoubleLockStatus())) && ((!IsZero(EDDFR2())) || (!OSLockStatus())), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.27 EDECCR, External Debug Exception Catch Control Register

The EDECCR characteristics are:

Purpose

Controls Exception Catch debug events. For more information, see ‘Exception Catch debug event’.

Configuration

EDECCR is in the Core power domain

External register EDECCR bits [31:0] are architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

External register EDECCR bits [31:0] are architecturally mapped to AArch32 System register DBGOSECCR[31:0].

Attributes

EDECCR is a 32-bit register.

Field descriptions

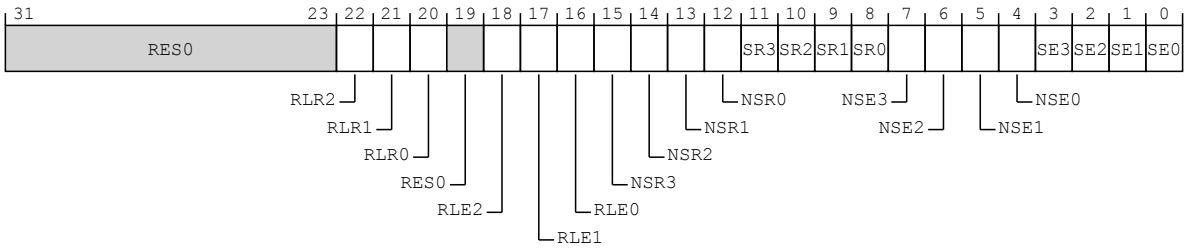

Bits [31:23]

Reserved, RES0.

RLR2, bit [22]

When FEAT_RME is implemented:

Controls exception catch on exception return to Realm EL2 in conjunction with EDECCR.RLE2.

RLR2	Meaning
0b0	If EDECCR.RLE2 is 0, then Exception Catch debug events are disabled for Realm EL2. If EDECCR.RLE2 is 1, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL2.
0b1	If EDECCR.RLE2 is 0, then Exception Catch debug events are enabled for exception returns to Realm EL2. If EDECCR.RLE2 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

RLR1, bit [21]

When FEAT_RME is implemented:

Controls exception catch on exception return to Realm EL1 in conjunction with EDECCR.RLE1.

RLR1	Meaning
`0b0`	If EDECCR.RLE1 is 0, then Exception Catch debug events are disabled for Realm EL1. If EDECCR.RLE1 is 1, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL1.
`0b1`	If EDECCR.RLE1 is 0, then Exception Catch debug events are enabled for exception returns to Realm EL1. If EDECCR.RLE1 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

RLR0, bit [20]

When FEAT_RME is implemented:

Controls exception catch on exception return to Realm EL0.

RLR0	Meaning
`0b0`	Exception Catch debug events are disabled for Realm EL0.
`0b1`	Exception Catch debug events are enabled for exception returns to Realm EL0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [19]

Reserved, RES0.

RLE2, bit [18]

When FEAT_RME is implemented:

Controls exception catch on exception entry to Realm EL2. Also controls exception catch on exception return to Realm EL2 in conjunction with EDECCR.RLR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLE2	Meaning
0b0	If EDECCR.RLR2 is 0, then Exception Catch debug events are disabled for Realm EL2. If EDECCR.RLR2 is 1, then Exception Catch debug events are enabled for exception returns to Realm EL2.
0b1	If EDECCR.RLR2 is 0, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL2. If EDECCR.RLR2 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

RLE1, bit [17]

When FEAT_RME is implemented:

Controls exception catch on exception entry to Realm EL1. Also controls exception catch on exception return to Realm EL1 in conjunction with EDECCR.RLR1.

RLE1	Meaning
0b0	If EDECCR.RLR1 is 0, then Exception Catch debug events are disabled for Realm EL1. If EDECCR.RLR1 is 1, then Exception Catch debug events are enabled for exception returns to Realm EL1.
0b1	If EDECCR.RLR1 is 0, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL1. If EDECCR.RLR1 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

RLE0, bit [16]

Access to this field is RES0.

NSR3, bit [15]

Access to this field is RES0.

NSR2, bit [14]

When FEAT_Debugv8p2 is implemented and Non-secure EL2 is implemented:

Controls exception catch on exception return to Non-secure EL2 in conjunction with EDECCR.NSE2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSR2	Meaning
0b0	If EDECCR.NSE2 is 0, then Exception Catch debug events are disabled for Non-secure EL2. If EDECCR.NSE2 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL2.
0b1	If EDECCR.NSE2 is 0, then Exception Catch debug events are enabled for exception returns to Non-secure EL2. If EDECCR.NSE2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

NSR1, bit [13]

When FEAT_Debugv8p2 is implemented and Non-secure EL1 is implemented:

Controls exception catch on exception return to Non-secure EL1 in conjunction with EDECCR.NSE1.

NSR1	Meaning
0b0	If EDECCR.NSE1 is 0, then Exception Catch debug events are disabled for Non-secure EL1. If EDECCR.NSE1 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL1.
0b1	If EDECCR.NSE1 is 0, then Exception Catch debug events are enabled for exception returns to Non-secure EL1. If EDECCR.NSE1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

NSR0, bit [12]

When FEAT_Debugv8p2 is implemented and Non-secure EL0 is implemented:

Controls exception catch on exception return to Non-secure EL0.

NSR0	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL0.
0b1	Exception Catch debug events are enabled for exception returns to Non-secure EL0.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SR3, bit [11]

When FEAT_Debugv8p2 is implemented and EL3 is implemented:

Controls exception catch on exception return to EL3 in conjunction with EDECCR.SE3.

SR3	Meaning
`0b0`	If EDECCR.SE3 is 0, then Exception Catch debug events are disabled for EL3. If EDECCR.SE3 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to EL3.
`0b1`	If EDECCR.SE3 is 0, then Exception Catch debug events are enabled for exception returns to EL3. If EDECCR.SE3 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SR2, bit [10]

When FEAT_Debugv8p2 is implemented and FEAT_SEL2 is implemented:

Controls exception catch on exception return to Secure EL2 in conjunction with EDECCR.SE2.

SR2	Meaning
`0b0`	If EDECCR.SE2 is 0, then Exception Catch debug events are disabled for Secure EL2. If EDECCR.SE2 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL2.
`0b1`	If EDECCR.SE2 is 0, then Exception Catch debug events are enabled for exception returns to Secure EL2. If EDECCR.SE2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SR1, bit [9]

When FEAT_Debugv8p2 is implemented and Secure EL1 is implemented:

Controls exception catch on exception return to Secure EL1 in conjunction with EDECCR.SE1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SR1	Meaning
0b0	If EDECCR.SE1 is 0, then Exception Catch debug events are disabled for Secure EL1. If EDECCR.SE1 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL1.
0b1	If EDECCR.SE1 is 0, then Exception Catch debug events are enabled for exception returns to Secure EL1. If EDECCR.SE1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SR0, bit [8]

When FEAT_Debugv8p2 is implemented and Secure EL0 is implemented:

Controls exception catch on exception return to Secure EL0.

SR0	Meaning
0b0	Exception Catch debug events are disabled for Secure EL0.
0b1	Exception Catch debug events are enabled for exception returns to Secure EL0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

NSE3, bit [7]

Access to this field is RES0.

NSE2, bit [6]

When FEAT_Debugv8p2 is implemented and Non-secure EL2 is implemented:

Controls exception catch on exception entry to Non-secure EL2. Also controls exception catch on exception return to Non-secure EL2 in conjunction with EDECCR.NSR2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE2	Meaning
0b0	If EDECCR.NSR2 is 0, then Exception Catch debug events are disabled for Non-secure EL2. If EDECCR.NSR2 is 1, then Exception Catch debug events are enabled for exception returns to Non-secure EL2.
0b1	If EDECCR.NSR2 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL2. If EDECCR.NSR2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL2.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

When Non-secure EL2 is implemented:

Coarse-grained exception catch for Non-secure EL2. Controls Exception Catch debug events for Non-secure EL2.

NSE2	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL2.
0b1	Exception Catch debug events are enabled for Non-secure EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

NSE1, bit [5]

When FEAT_Debugv8p2 is implemented and Non-secure EL1 is implemented:

Controls exception catch on exception entry to Non-secure EL1. Also controls exception catch on exception return to Non-secure EL1 in conjunction with EDECCR.NSR1.

NSE1	Meaning
0b0	If EDECCR.NSR1 is 0, then Exception Catch debug events are disabled for Non-secure EL1. If EDECCR.NSR1 is 1, then Exception Catch debug events are enabled for exception returns to Non-secure EL1.
0b1	If EDECCR.NSR1 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL1. If EDECCR.NSR1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

When Non-secure EL1 is implemented:

Coarse-grained exception catch for Non-secure EL1. Controls Exception Catch debug events for Non-secure EL1.

NSE1	Meaning
`0b0`	Exception Catch debug events are disabled for Non-secure EL1.
`0b1`	Exception Catch debug events are enabled for Non-secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

NSE0, bit [4]

Access to this field is RES0.

SE3, bit [3]

When FEAT_Debugv8p2 is implemented and EL3 is implemented:

Controls exception catch on exception entry to EL3. Also controls exception catch on exception return to EL3 in conjunction with EDECCR.SR3.

SE3	Meaning
`0b0`	If EDECCR.SR3 is 0, then Exception Catch debug events are disabled for EL3. If EDECCR.SR3 is 1, then Exception Catch debug events are enabled for exception returns to EL3.
`0b1`	If EDECCR.SR3 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to EL3. If EDECCR.SR3 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to EL3.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_Debugv8p2 is not implemented and EL3 is implemented:

Coarse-grained exception catch for EL3. Controls Exception Catch debug events for EL3.

SE3	Meaning
`0b0`	Exception Catch debug events are disabled for EL3.
`0b1`	Exception Catch debug events are enabled for EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SE2, bit [2]

When FEAT_Debugv8p2 is implemented and FEAT_SEL2 is implemented:

Controls exception catch on exception entry to Secure EL2. Also controls exception catch on exception return to Secure EL2 in conjunction with EDECCR.SR2.

SE2	Meaning
`0b0`	If EDECCR.SR2 is 0, then Exception Catch debug events are disabled for Secure EL2. If EDECCR.SR2 is 1, then Exception Catch debug events are enabled for exception returns to Secure EL2.
`0b1`	If EDECCR.SR2 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL2. If EDECCR.SR2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL2.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SE1, bit [1]

When FEAT_Debugv8p2 is implemented and Secure EL1 is implemented:

Controls exception catch on exception entry to Secure EL1. Also controls exception catch on exception return to Secure EL1 in conjunction with EDECCR.SR1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SE1	Meaning
0b0	If EDECCR.SR1 is 0, then Exception Catch debug events are disabled for Secure EL1. If EDECCR.SR1 is 1, then Exception Catch debug events are enabled for exception returns to Secure EL1.
0b1	If EDECCR.SR1 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL1. If EDECCR.SR1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL1.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

When Secure EL1 is implemented:

Coarse-grained exception catch for Secure EL1. Controls Exception Catch debug events for Secure EL1.

SE1	Meaning
0b0	Exception Catch debug events are disabled for Secure EL1.
0b1	Exception Catch debug events are enabled for Secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

SE0, bit [0]

Access to this field is RES0.

Accessing EDECCR

EDECCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x098	EDECCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.28 EDECR, External Debug Execution Control Register

The EDECR characteristics are:

Purpose

Controls Halting debug events.

Configuration

When FEAT_DoPD is implemented, EDECR is in the Core power domain. Otherwise, EDECR is in the Debug power domain

Attributes

EDECR is a 32-bit register.

Field descriptions

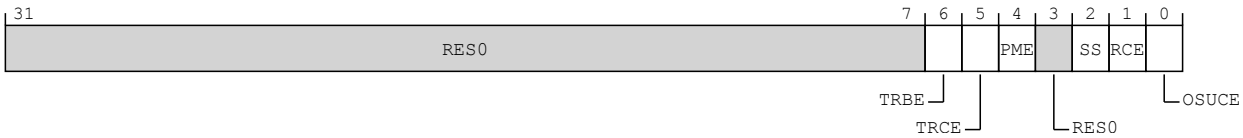

Bits [31:7]

Reserved, RES0.

TRBE, bit [6]

When FEAT_Debugv8p9 is implemented and FEAT_TRBE_EXT is implemented:

Trace Buffer External Debug Request Enable.

TRBE	Meaning
0b0	Trace Buffer External Debug Request disabled.
0b1	Trace Buffer External Debug Request enabled.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

TRCE, bit [5]

When FEAT_ETEv1p3 is implemented and FEAT_Debugv8p9 is implemented:

ETE External Debug Request Enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCE	Meaning
0b0	ETE External Debug Request disabled.
0b1	ETE External Debug Request enabled.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

PME, bit [4]

When FEAT_Debugv8p9 is implemented and FEAT_PMUv3p9 is implemented:

PMU Overflow External Debug Request Enable.

PME	Meaning
0b0	PMU Overflow External Debug Request disabled.
0b1	PMU Overflow External Debug Request enabled.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bit [3]

Reserved, RES0.

SS, bit [2]

Halting step enable. Possible values of this field are:

SS	Meaning
0b0	Halting step debug event disabled.
0b1	Halting step debug event enabled.

If the value of EDECR.SS is changed when the PE is in Non-debug state, behavior is CONSTRAINED UNPREDICTABLE as described in 'Changing the value of EDECR.SS when not in Debug state'.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_DoPD is implemented, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a External debug reset:
  - When FEAT_DoPD is not implemented, this field resets to `'0'`.

RCE, bit [1]

When FEAT_DoPD is not implemented:

Reset Catch Enable.

RCE	Meaning
`0b0`	Reset Catch debug event disabled.
`0b1`	Reset Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

OSUCE, bit [0]

When FEAT_DoPD is not implemented:

OS Unlock Catch Enable.

OSUCE	Meaning
`0b0`	OS Unlock Catch debug event disabled.
`0b1`	OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Accessing EDECR

EDECR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x024`	EDECR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.29 EDESR, External Debug Event Status Register

The EDESR characteristics are:

Purpose

Indicates the status of internally pending Halting debug events.

Configuration

EDESR is in the Core power domain

Attributes

EDESR is a 32-bit register.

Field descriptions

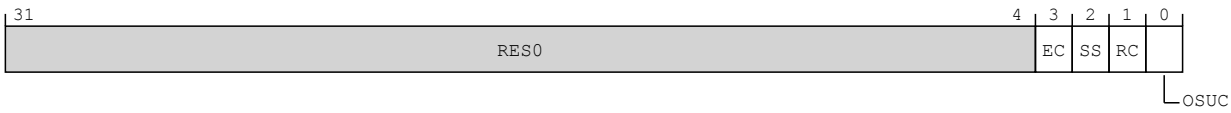

Bits [31:4]

Reserved, RES0.

EC, bit [3]

When FEAT_Debugv8p8 is implemented:

Exception Catch debug event pending.

EC	Meaning
`0b0`	Exception Catch debug event is not pending.
`0b1`	Exception Catch debug event is pending.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Access to this field is W1C.

Otherwise:

Reserved, RES0.

SS, bit [2]

When FEAT_DoPD is implemented:

Halting step debug event pending. Possible values of this field are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS	Meaning
0b0	Reading this means that a Halting step debug event is not pending. Writing this means no action.
0b1	Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Halting step debug event pending. Possible values of this field are:

SS	Meaning
0b0	Reading this means that a Halting step debug event is not pending. Writing this means no action.
0b1	Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in EDECR.SS.

RC, bit [1]

Reset Catch debug event pending. Possible values of this field are:

RC	Meaning
0b0	Reading this means that a Reset Catch debug event is not pending. Writing this means no action.
0b1	Reading this means that a Reset Catch debug event is pending. Writing this clears the pending Reset Catch debug event.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_DoPD is implemented, this field resets to the value in CTIDEVCTL.RCE.
  - When FEAT_DoPD is not implemented, this field resets to the value in EDECR.RCE.

OSUC, bit [0]

OS Unlock Catch debug event pending. Possible values of this field are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSUC	Meaning
0b0	Reading this means that an OS Unlock Catch debug event is not pending. Writing this means no action.
0b1	Reading this means that an OS Unlock Catch debug event is pending. Writing this clears the pending OS Unlock Catch debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing EDESR

If a request to clear a pending Halting debug event is received at or about the time when halting becomes allowed, it is CONSTRAINED UNPREDICTABLE whether the event is taken.

If Core power is removed while a Halting debug event is pending, it is lost. However, it might become pending again when the Core is powered back on and Cold reset.

EDESR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x020	EDESR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.30 EDHSR, External Debug Halting Syndrome Register

The EDHSR characteristics are:

Purpose

Holds syndrome information for a debug event.

Configuration

EDHSR is in the Core power domain

The value of this register is UNKNOWN if the PE is in Non-debug state, or if EDSCR.STATUS is not `0b101011`.

This register is present only when FEAT_EDHSR is implemented. Otherwise, direct accesses to EDHSR are RES0.

Attributes

EDHSR is a 64-bit register.

Field descriptions

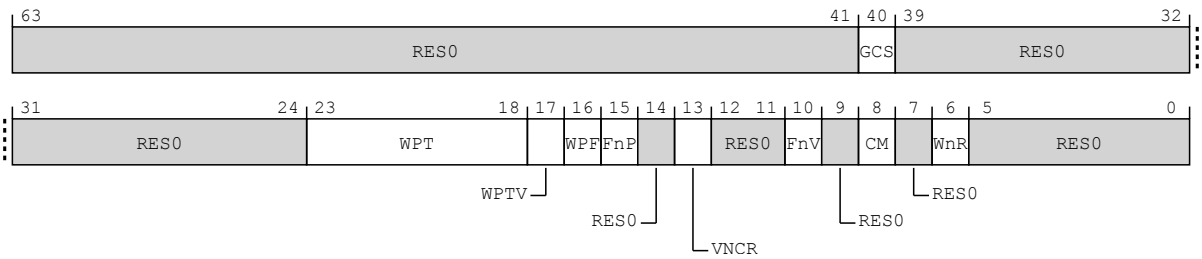

Bits [63:41]

Reserved, RES0.

GCS, bit [40]

When FEAT_GCS is implemented and FEAT_Debugv8p9 is implemented:

Guarded control stack data access.

Indicates that the Watchpoint debug event is due to a Guarded control stack data access.

GCS	Meaning
`0b0`	The Watchpoint debug event is not due to a Guarded control stack data access.
`0b1`	The Watchpoint debug event is due to a Guarded control stack data access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [39:24]

Reserved, RES0.

WPT, bits [23:18]

Watchpoint number. When EDHSR.WPTV is 1, holds the index of a watchpoint that triggered the Watchpoint debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WPTV, bit [17]

Watchpoint number valid.

WPTV	Meaning	Applies when
`0b0`	EDHSR.WPT field is not valid, and holds an UNKNOWN value.	FEAT_Debugv8p9 is not implemented
`0b1`	EDHSR.WPT field is valid, and holds the number of a watchpoint that triggered the Watchpoint debug event.	

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WPF, bit [16]

Watchpoint might be false-positive.

WPF	Meaning	Applies when
`0b0`	The watchpoint matched an address or address range that was accessed by the instruction.	
`0b1`	The watchpoint matched an address or address range that might not have been accessed by the instruction.	FEAT_SVE is implemented or FEAT_SME is implemented

Arm strongly recommends that this bit is set to 0, other than when one of the following instructions might generate a watchpoint match for an address or address range that the instruction does not access:

- An SVE contiguous vector load/store instruction, when the PE is in Streaming SVE mode.
- An SME load/store instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FnP, bit [15]

EDWAR not Precise.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FnP	Meaning	Applies when
0b0	If the EDWAR is valid, it holds the virtual address of an access or sequence of contiguous accesses that triggered the Watchpoint debug event.	
0b1	If the EDWAR is valid, it holds any virtual address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered the Watchpoint debug event.	FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [14]

Reserved, RES0.

VNCR, bit [13]

When FEAT_Debugv8p9 is implemented:

VNCR_EL2 access. Indicates that the Watchpoint debug event came from use of VNCR_EL2 register by EL1 code.

VNCR	Meaning	Applies when
0b0	The Watchpoint debug event was not generated by the use of VNCR_EL2 by EL1 code.	
0b1	The Watchpoint debug event was generated by the use of VNCR_EL2 by EL1 code.	FEAT_NV2 is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [12:11]

Reserved, RES0.

FnV, bit [10]

EDWAR not Valid.

FnV	Meaning	Applies when
0b0	EDWAR is valid.	
0b1	EDWAR is not valid, and holds an UNKNOWN value.	FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

CM, bit [8]

When FEAT_Debugv8p9 is implemented:

Cache maintenance. Indicates whether the Watchpoint debug event came from a cache maintenance instruction.

CM	Meaning
0b0	The Watchpoint debug event was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Watchpoint debug event was generated by the execution of a cache maintenance instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not cache maintenance instructions, and therefore do not cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

WnR, bit [6]

When FEAT_Debugv8p9 is implemented:

Write not Read. Indicates whether the Watchpoint debug event was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Watchpoint debug event caused by an instruction reading from a memory location.
0b1	Watchpoint debug event caused by an instruction writing to a memory location.

For Watchpoint debug events on cache maintenance instructions, this field is set to 1.

For Watchpoint debug events from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint debug event, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [5:0]

Reserved, RES0.

Accessing EDHSR

EDHSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x038`	EDHSR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.31 EDITCTRL, External Debug Integration mode Control register

The EDITCTRL characteristics are:

Purpose

Enables the external debug to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

The power domain of EDITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

EDITCTRL is a 32-bit register.

Field descriptions

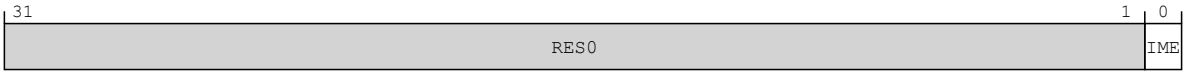

Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- Whichever power domain the register is implemented in, this field resets to 0.
- Otherwise, the value of this field is unchanged.

Accessing EDITCTRL

EDITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xF00	EDITCTRL

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.32 EDITR, External Debug Instruction Transfer Register

The EDITR characteristics are:

Purpose

Used in Debug state for passing instructions to the PE for execution.

Configuration

EDITR is in the Core power domain

Attributes

EDITR is a 32-bit register.

Field descriptions

When FEAT_AA32 is implemented and UsingAArch32():

31 … 16	15 … 0
hw2	hw1

hw2, bits [31:16]

Second halfword of the T32 instruction to be executed on the PE. When EDITR contains a 16-bit T32 instruction, this field is ignored. For more information, see ‘Behavior in Debug state’.

Note

The hw2 field is displayed on the left. This is not the usual convention for display of T32 instruction halfwords.

hw1, bits [15:0]

First halfword of the T32 instruction to be executed on the PE.

Note

The hw1 field is displayed on the right. This is not the usual convention for display of T32 instruction halfwords.

When FEAT_AA64 is implemented and !UsingAArch32():

31 … 0
A64 instruction to be executed on the PE

A64_Instruction, bits [31:0]

A64 instruction to be executed on the PE.

Accessing EDITR

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any instruction issued through the ITR in Normal access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

EDITR ignores writes if the PE is in Non-debug state.

EDITR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x084`	EDITR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.33 EDLAR, External Debug Lock Access Register

The EDLAR characteristics are:

Purpose

Allows or disallows access to the external debug registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

When FEAT_DoPD is implemented, EDLAR is in the Core power domain. Otherwise, EDLAR is in the Debug power domain

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE.

Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLAR is a 32-bit register.

Field descriptions

When Debug Software Lock is implemented:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Accessing EDLAR

EDLAR can be accessed through the memory-mapped interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0xFB0`	EDLAR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.34 EDLSR, External Debug Lock Status Register

The EDLSR characteristics are:

Purpose

Indicates the current status of the software lock for external debug registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

When FEAT_DoPD is implemented, EDLSR is in the Core power domain. Otherwise, EDLSR is in the Debug power domain

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE.

Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLSR is a 32-bit register.

Field descriptions

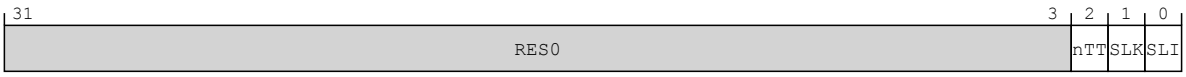

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When Debug Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.
0b1	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_DoPD is implemented, this field resets to `'1'`.
- On a External debug reset:
  - When FEAT_DoPD is not implemented, this field resets to `'1'`.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
`0b0`	Software Lock not implemented or not memory-mapped access.
`0b1`	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing EDLSR

EDLSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
Debug	`0xFB4`	EDLSR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.35 EDPCSR, External Debug Program Counter Sample Register

The EDPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Configuration

EDPCSR is in the Core power domain

If FEAT_Debugv8p1 is implemented, the format of this register differs depending on the value of EDSCR.SC2.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDPCSR are RES0.

Attributes

EDPCSR is a 64-bit register.

Field descriptions

When FEAT_Debugv8p1 is not implemented or EDSCR.SC2 == '0':

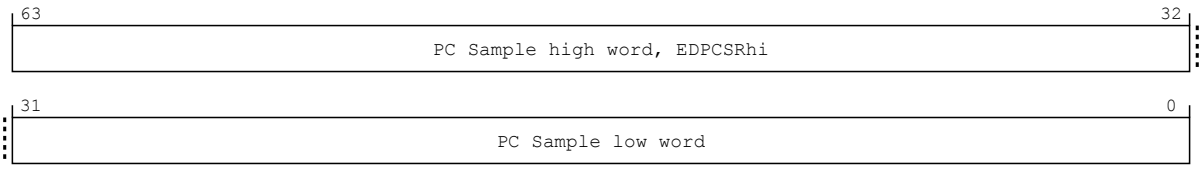

EDPCSRhi, bits [63:32]

PC Sample high word, EDPCSRhi. If EDVIDSR.HV == 0, then this field is RAZ, otherwise bits [63:32] of the sampled instruction address value. The translation regime that EDPCSR samples can be determined from EDVIDSR.{NS,E2,E3}.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

EDPCSRlo, bits [31:0]

PC Sample low word. EDPCSRlo, bits[31:0] of the sampled instruction address value.

EDPCSRlo reads as `0xFFFFFFFF` when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of EDPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

EDPCSRlo reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- No branch instruction has retired since the last read of EDPCSR[31:0].

For the cases where a read of EDPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting EDPCSRhi, EDCIDSR, and EDVIDSR to UNKNOWN values.

Otherwise, a read of EDPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to EDPCSRhi, EDCIDSR, and EDVIDSR. The translation regime that EDPCSR samples can be determined from EDVIDSR.{NS,E2,E3}.

For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and EDPCSRhi, EDCIDSR, and EDVIDSR are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When FEAT_Debugv8p1 is implemented and EDSCR.SC2 == '1':

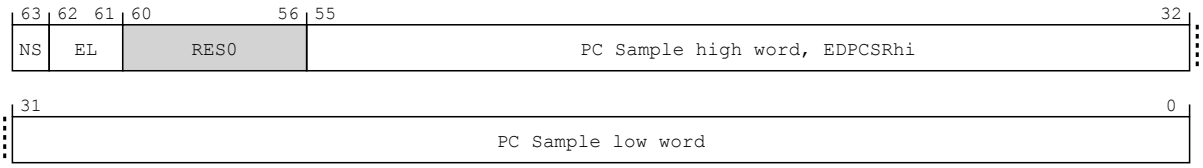

NS, bit [63]

Non-secure state sample. Indicates the Security state that is associated with the most recent EDPCSR sample or, when it is read as a single atomic 64-bit read, the current EDPCSR sample. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

NS	Meaning
`0b0`	Sample is from Secure state.
`0b1`	Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level that is associated with the most recent EDPCSR sample or, when it is read as a single atomic 64-bit read, the current EDPCSR sample. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

EL	Meaning
`0b00`	Sample is from EL0.
`0b01`	Sample is from EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL	Meaning
0b10	Sample is from EL2.
0b11	Sample is from EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [60:56]

Reserved, RES0.

EDPCSRhi, bits [55:32]

PC Sample high word, EDPCSRhi. Bits [55:32] of the sampled instruction address value. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

EDPCSRlo, bits [31:0]

PC Sample low word. EDPCSRlo, bits[31:0] of the sampled instruction address value.

EDPCSRlo reads as `0xFFFFFFFF` when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of EDPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

EDPCSRlo reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of EDPCSR[31:0].

For the cases where a read of EDPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting EDPCSRhi, EDCIDSR, and EDVIDSR to UNKNOWN values.

Otherwise, a read of EDPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to EDPCSRhi, EDCIDSR, and EDVIDSR. The translation regime that EDPCSR samples can be determined from EDPCSR.{NS,EL}.

For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and EDPCSRhi, EDCIDSR, and EDVIDSR are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDPCSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'

EDPCSR[63:32] and EDPCSR[31:0] are accessed at 32-bit memory mapped addresses that are not contiguous.

EDPCSR can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance	Range
Debug	`0x0A0`	EDPCSR	31:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Component	Offset	Instance	Range
Debug	`0x0AC`	EDPCSR	63:32

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.36 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

Purpose

Provides information about implemented PE features.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

There are no configuration notes.

Attributes

EDPFR is a 64-bit register.

Field descriptions

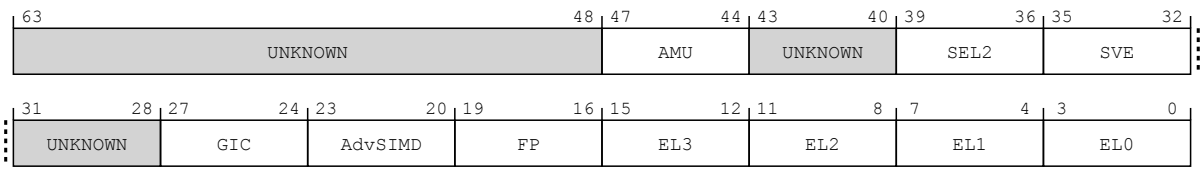

Bits [63:48]

Reserved, UNKNOWN.

AMU, bits [47:44]

Indicates support for Activity Monitors Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AMU	Meaning
0b0000	Activity Monitors Extension is not implemented.
0b0001	FEAT_AMUv1 is implemented.
0b0010	FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

FEAT_AMUv1 implements the functionality identified by the value 0b0001.

FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0000.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [43:40]

Reserved, UNKNOWN.

SEL2, bits [39:36]

Secure EL2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SEL2	Meaning
0b0000	Secure EL2 is not implemented.
0b0001	Secure EL2 is implemented.

FEAT_SEL2 implements the functionality identified by the value 0b0001

All other values are reserved.

Access to this field is RO.

SVE, bits [35:32]

Scalable Vector Extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SVE	Meaning
0b0000	SVE is not implemented.
0b0001	SVE is implemented.

FEAT_SVE implements the functionality identified by the value 0b0001.

All other values are reserved.

Access to this field is RO.

Bits [31:28]

Reserved, UNKNOWN.

GIC, bits [27:24]

System register GIC interface support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface system registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

In an implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.GIC.

Access to this field is RO.

AdvSIMD, bits [23:20]

Advanced SIMD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AdvSIMD	Meaning
0b0000	Advanced SIMD is implemented, including support for the following SISD and SIMD operations: • Integer byte, halfword, word and doubleword element operations. • Single-precision and double-precision floating-point arithmetic. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- 0b0000 in an implementation with Advanced SIMD support, that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with Advanced SIMD support, that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without Advanced SIMD support.

In an implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.AdvSIMD.

Access to this field is RO.

FP, bits [19:16]

Floating-point.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FP	Meaning
0b0000	Floating-point is implemented, and includes support for: • Single-precision and double-precision floating-point types. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FP	Meaning
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- 0b0000 in an implementation with floating-point support, that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with floating-point support, that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without floating-point support.

In an implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.FP.

Access to this field is RO.

EL3, bits [15:12]

AArch64 EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented or cannot be executed in AArch64 state.
0b0001	EL3 can be executed in AArch64 state only.
0b0010	EL3 can be executed in both Execution states.

When the value of EDAA32PFR.EL3 is nonzero, this field must be 0b0000.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL3.

Access to this field is RO.

EL2, bits [11:8]

AArch64 EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented or cannot be executed in AArch64 state.
0b0001	EL2 can be executed in AArch64 state only.
0b0010	EL2 can be executed in both Execution states.

When the value of EDAA32PFR.EL2 is nonzero, this field must be 0b0000.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL2.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL1, bits [7:4]

AArch64 EL1 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL1	Meaning
0b0000	EL1 cannot be executed in AArch64 state. EL1 can be executed in AArch32 state only.
0b0001	EL1 can be executed in AArch64 state only.
0b0010	EL1 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL1.

Access to this field is RO.

EL0, bits [3:0]

AArch64 EL0 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL0	Meaning
0b0000	EL0 cannot be executed in AArch64 state. EL0 can be executed in AArch32 state only.
0b0001	EL0 can be executed in AArch64 state only.
0b0010	EL0 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL0.

Access to this field is RO.

Accessing EDPFR

EDPFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD20	EDPFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.37 EDPIDR0, External Debug Peripheral Identification Register 0

The EDPIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR0 is in the Core power domain. Otherwise, EDPIDR0 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - EDPIDR1.PART_1 contains part number bits [11:8].
  - EDPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - EDPIDR1.PART_1 contains part number bits [15:12].
  - EDPIDR0.PART_0 contains part number bits [11:4].
  - EDPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, EDPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR0

EDPIDR0 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0xFE0`	EDPIDR0

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.38 EDPIDR1, External Debug Peripheral Identification Register 1

The EDPIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR1 is in the Core power domain. Otherwise, EDPIDR1 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- EDPIDR1.DES_0: JEP106 identification code bits[3:0].
- EDPIDR2.DES_1: JEP106 identification code bits[6:4].
- EDPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - EDPIDR1.PART_1 contains part number bits [11:8].
  - EDPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - EDPIDR1.PART_1 contains part number bits [15:12].
  - EDPIDR0.PART_0 contains part number bits [11:4].
  - EDPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, EDPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR1

EDPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE4`	EDPIDR1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.39 EDPIDR2, External Debug Peripheral Identification Register 2

The EDPIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR2 is in the Core power domain. Otherwise, EDPIDR2 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR2 is a 32-bit register.

Field descriptions

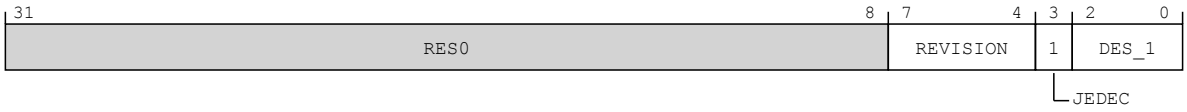

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- EDPIDR2.REVISION indicates the 4-bit revision number.
- EDPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- EDPIDR2.REVISION indicates the 4-bit major revision number.
- EDPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- EDPIDR2.REVISION contains part number bits [3:0].
- EDPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- EDPIDR1.DES_0: JEP106 identification code bits[3:0].
- EDPIDR2.DES_1: JEP106 identification code bits[6:4].
- EDPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing EDPIDR2

EDPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE8`	EDPIDR2

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.40 EDPIDR3, External Debug Peripheral Identification Register 3

The EDPIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR3 is in the Core power domain. Otherwise, EDPIDR3 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- EDPIDR2.REVISION indicates the 4-bit revision number.
- EDPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- EDPIDR2.REVISION indicates the 4-bit major revision number.
- EDPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- EDPIDR2.REVISION contains part number bits [3:0].
- EDPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

EDPIDR3.CMOD might also indicate component modifications.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

EDPIDR3.REVAND might also indicate component modifications.

Access to this field is RO.

Accessing EDPIDR3

EDPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFEC`	EDPIDR3

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.41 EDPIDR4, External Debug Peripheral Identification Register 4

The EDPIDR4 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR4 is in the Core power domain. Otherwise, EDPIDR4 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR4 is a 32-bit register.

Field descriptions

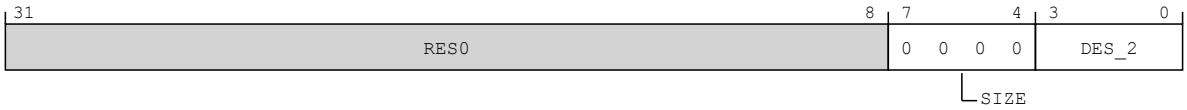

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- EDPIDR1.DES_0: JEP106 identification code bits[3:0].
- EDPIDR2.DES_1: JEP106 identification code bits[6:4].
- EDPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing EDPIDR4

EDPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFD0`	EDPIDR4

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.42 EDPRCR, External Debug Power/Reset Control Register

The EDPRCR characteristics are:

Purpose

Controls the PE functionality related to powerup, reset, and powerdown.

Configuration

When FEAT_DoPD is implemented, EDPRCR is in the Core power domain. Otherwise, EDPRCR contains fields that are in the Core power domain and fields that are in the Debug power domain

External register EDPRCR bit [0] is architecturally mapped to AArch64 System register DBGPRCR_EL1[0].

External register EDPRCR bit [0] is architecturally mapped to AArch32 System register DBGPRCR[0].

Attributes

EDPRCR is a 32-bit register.

Field descriptions

When FEAT_DoPD is implemented:

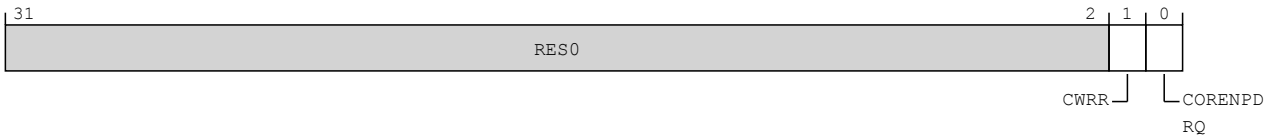

Bits [31:2]

Reserved, RES0.

CWRR, bit [1]

When FEAT_RME is implemented:

Access to this field is RAZ/WI.

Otherwise:

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

CWRR	Meaning
0b0	No action.
0b1	Request Warm reset.

This field is in the Core power domain

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation that includes the recommended external debug interface, this bit drives the **DBGRSTREQ** signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - OSLockStatus()
  - SoftwareLockStatus(addrdesc)
- Access to this field is RAZ/WI if all of the following are true:
  - !ExternalSecureInvasiveDebugEnabled()
  - FEAT_Secure is implemented
- Access to this field is RAZ/WI if all of the following are true:
  - !ExternalInvasiveDebugEnabled()
  - FEAT_Secure is not implemented
- Otherwise, access to this field is WO/RAZ.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
`0b0`	If the system responds to a powerdown request, it powers down Core power domain.
`0b1`	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.

Accessing this field has the following behavior:

- When OSLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

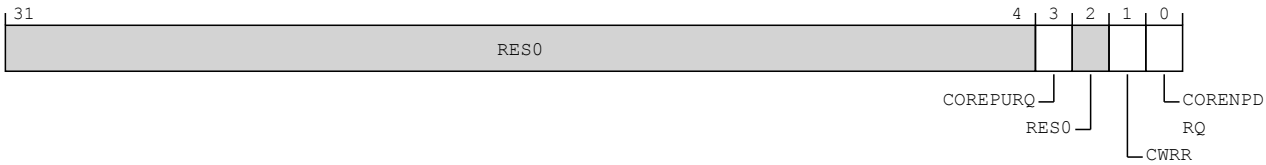

Bits [31:4]

Reserved, RES0.

COREPURQ, bit [3]

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain, and that the power controller emulates powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

COREPURQ	Meaning
`0b0`	Do not request power up of the Core power domain.
`0b1`	Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGPWRUPREQ** signal.

This field is in the Debug power domain and can be read and written when the Core power domain is powered off.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RW.

Bit [2]

Reserved, RES0.

CWRR, bit [1]

When FEAT_RME is implemented:

Access to this field is RAZ/WI.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

CWRR	Meaning
`0b0`	No action.
`0b1`	Request Warm reset.

This field is in the Core power domain

In an implementation that includes the recommended external debug interface, this bit drives the **DBGRSTREQ** signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
  - OSLockStatus()
  - SoftwareLockStatus(addrdesc)
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_Secure is implemented
  - !ExternalSecureInvasiveDebugEnabled()
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_Secure is not implemented
  - !ExternalInvasiveDebugEnabled()
- Otherwise, access to this field is WO/RAZ.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
`0b0`	If the system responds to a powerdown request, it powers down Core power domain.
`0b1`	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
  - OSLockStatus()
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RW.

Accessing EDPRCR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

EDPRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x310`	EDPRCR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.43 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

Purpose

Holds information about the reset and powerdown state of the PE.

Configuration

When FEAT_DoPD is implemented, EDPRSR is in the Core power domain. Otherwise, EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain

If FEAT_DoPD is implemented, then all fields in this register are in the Core power domain.

Attributes

EDPRSR is a 32-bit register.

Field descriptions

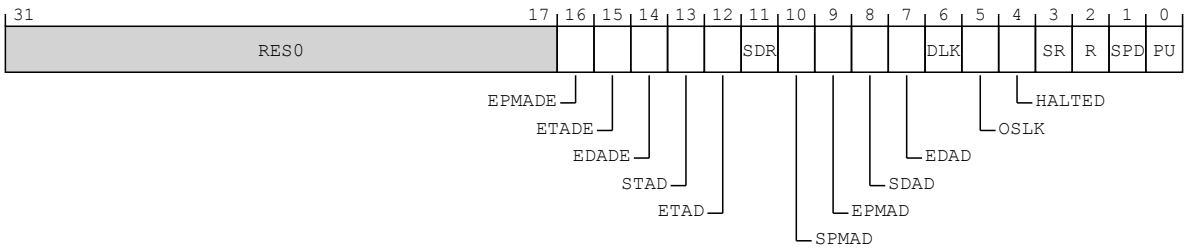

Bits [31:17]

Reserved, RES0.

EPMADE, bit [16]

When FEAT_RME is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable Extended status. Together with EDPRSR.EPMAD, reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.{EPMAD, EPMADE} controls.

For a description of the values derived by evaluating EDPRSR.EPMAD and EDPRSR.EPMADE together, see EDPRSR.EPMAD.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == ‘1’
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

ETADE, bit [15]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented, FEAT_TRC_EXT is implemented, and FEAT_TRBE is implemented:

External Trace Access Disable Extended status. Together with EDPRSR.ETAD, reports whether access to trace unit registers by an external debugger is prohibited by the MDCR_EL3.{ETAD, ETADE} controls.

For a description of the values derived by evaluating EDPRSR.ETAD and EDPRSR.ETADE together, see EDPRSR.ETAD.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

EDADE, bit [14]

When FEAT_RME is implemented:

External Debug Access Disable Extended status. Together with EDPRSR.EDAD, reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.{EDAD, EDADE} controls.

For a description of the values derived by evaluating EDPRSR.EDAD and EDPRSR.EDADE together, see EDPRSR.EDAD.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

STAD, bit [13]

When FEAT_TRC_EXT is implemented and FEAT_TRBE is implemented:

Sticky ETAD error. Set to 1 when a Non-secure external debug interface access to an external trace register returns an error because `AllowExternalTraceAccess()` == FALSE for the access.

STAD	Meaning
`0b0`	Since EDPRSR was last read, no external accesses to the trace unit registers have failed because `AllowExternalTraceAccess()` was FALSE for the access.
`0b1`	Since EDPRSR was last read, at least one external access to the trace unit registers has failed because `AllowExternalTraceAccess()` was FALSE for the access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `IsCorePowered()` == TRUE, the Core power domain is powered up, then, following a read of EDPRSR, then this bit clears to 0.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Otherwise, access to this field is RC/WI.

Otherwise:

Reserved, RES0.

ETAD, bit [12]

When FEAT_RME is implemented, FEAT_TRC_EXT is implemented, and FEAT_TRBE is implemented:

External Trace Access Disable status. Together with EDPRSR.ETADE, reports whether access to trace unit registers by an external debugger is prohibited by the MDCR_EL3.{ETAD, ETADE} controls.

ETADE	ETAD	Meaning
0b0	0b0	No accesses from an external debugger to trace unit registers are prohibited.
0b0	0b1	Realm and Non-secure accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected.
0b1	0b0	Secure and Non-secure accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == ‘1’
- Otherwise, access to this field is RO.

When FEAT_TRC_EXT is implemented and FEAT_TRBE is implemented:

External Trace Access Disable status. Reports whether Non-secure access to trace unit registers by an external debugger is prohibited by the MDCR_EL3.ETAD control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ETAD	Meaning
0b0	External Non-secure trace unit accesses not affected. `AllowExternalTraceAccess()` == TRUE for a Non-secure access.
0b1	External Non-secure trace unit accesses are prohibited. `AllowExternalTraceAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

SDR, bit [11]

Sticky Debug Restart. Set to 1 when the PE exits Debug state.

SDR	Meaning
0b0	The PE has not restarted since EDPRSR was last read.
0b1	The PE has restarted since EDPRSR was last read.

Note

If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to determine whether the PE has left Debug state.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RC/WI.

SPMAD, bit [10]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error. Set to 1 if an external debug interface access to a Performance Monitors register returns an error because `AllowExternalPMUAccess()` == FALSE.

SPMAD	Meaning
`0b0`	No Non-secure external debug interface accesses to the external Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.
`0b1`	At least one Non-secure external debug interface access to the external Performance Monitors register has failed and returned an error because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RC/WI.

When FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error.

SPMAD	Meaning
`0b0`	No external debug interface accesses to the Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.
`0b1`	At least one external debug interface access to the Performance Monitors registers has failed and returned an error because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented, and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When OSLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RC/WI.

Otherwise:

Reserved, RES0.

EPMAD, bit [9]

When FEAT_RME is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Together with EDPRSR.EPMADE, reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.{EPMAD, EPMADE} controls.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMADE	EPMAD	Meaning
0b0	0b0	No accesses from an external debugger to affected Performance Monitor registers are prohibited.
0b0	0b1	Realm and Non-secure accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected.
0b1	0b0	Secure and Non-secure accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == ‘1’
- Otherwise, access to this field is RO.

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether Non-secure access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
0b0	External Non-secure access to Performance Monitor registers not affected. `AllowExternalPMUAccess()` == TRUE for a Non-secure access.
0b1	External Non-secure access to affected Performance Monitor registers is prohibited. `AllowExternalPMUAccess()` == FALSE for a Non-secure access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
`0b0`	External access to Performance Monitor registers not affected. `AllowExternalPMUAccess()` == TRUE.
`0b1`	External access to affected Performance Monitor registers is prohibited. `AllowExternalPMUAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When OSLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3 is implemented:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

SDAD, bit [8]

When FEAT_Debugv8p4 is implemented:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess()` == FALSE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SDAD	Meaning
0b0	No Non-secure external debug interface accesses to the debug registers have failed because AllowExternalDebugAccess() == FALSE for the access since EDPRSR was last read.
0b1	At least one Non-secure external debug interface access to the debug registers has failed and returned an error because AllowExternalDebugAccess() == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because AllowExternalDebugAccess() == FALSE.

SDAD	Meaning
0b0	No external debug interface accesses to the debug registers have failed because AllowExternalDebugAccess() == FALSE since EDPRSR was last read.
0b1	At least one external debug interface access to the debug registers has failed and returned an error because AllowExternalDebugAccess() == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
  - OSLockStatus()
  - external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess(addrdesc) == FALSE
- Otherwise, access to this field is RO.

EDAD, bit [7]

When FEAT_RME is implemented:

External Debug Access Disable status. Together with EDPRSR.EDADE, reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.{EDAD, EDADE} controls.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDADE	EDAD	Meaning
0b0	0b0	No accesses from an external debugger to affected debug registers are prohibited.
0b0	0b1	Realm and Non-secure accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected.
0b1	0b0	Secure and Non-secure accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

When FEAT_Debugv8p4 is implemented:

External Debug Access Disable status. Reports whether Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
0b0	External Non-secure access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE for a Non-secure access.
0b1	External Non-secure access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_Debugv8p2 is implemented:

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
`0b0`	External access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE.
`0b1`	External access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
  - OSLockStatus()
  - external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess(addrdesc) == FALSE
- Otherwise, access to this field is RO.

Otherwise:

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and optionally OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
`0b0`	External access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE.
`0b1`	External access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

DLK, bit [6]

When FEAT_Debugv8p4 is implemented:

This field is RES0.

When FEAT_Debugv8p2 is implemented and FEAT_DoubleLock is implemented:

Double Lock. From Armv8.2, use of this field is deprecated.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RAZ/WI.

When FEAT_DoubleLock is implemented:

Double Lock.

This field returns the result of the pseudocode function `DoubleLockStatus()`.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether:

- EDPRSR.PU reads as 1, EDPRSR.DLK reads as 1, and EDPRSR.SPD is UNKNOWN.
- EDPRSR.PU reads as 0, EDPRSR.DLK is UNKNOWN, and EDPRSR.SPD reads as 0.

This field is in the Core power domain.

DLK	Meaning
0b0	`DoubleLockStatus()` returns FALSE.
0b1	`DoubleLockStatus()` returns TRUE and the Core power domain is powered up.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

OSLK, bit [5]

OS Lock status bit.

A read of this bit returns the value of OSLSR_EL1.OSLK.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— !IsCorePowered()
- EDPRSR.R == '1'

- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

HALTED, bit [4]

Halted status bit.

HALTED	Meaning
`0b0`	PE is in Non-debug state.
`0b1`	PE is in Debug state.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Otherwise, access to this field is RO.

SR, bit [3]

Sticky core Reset status bit.

SR	Meaning
`0b0`	The non-debug logic of the PE is not in reset state and has not been reset since the last time EDPRSR was read.
`0b1`	The non-debug logic of the PE is in reset state or has been reset since the last time EDPRSR was read.

If EDPRSR.PU reads as 1 and EDPRSR.R reads as 0, which means that the Core power domain is in a powerup state and that the non-debug logic of the PE is not in reset state, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RC/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R, bit [2]

PE Reset status bit.

R	Meaning
`0b0`	The non-debug logic of the PE is not in reset state.
`0b1`	The non-debug logic of the PE is in reset state.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see 'EDPRSR.{DLK, R} and reset state'.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RO.

SPD, bit [1]

Sticky core Powerdown status bit.

If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, then:

- If FEAT_Debugv8p2 is implemented, this bit reads as 0.
- If FEAT_Debugv8p2 is not implemented, this bit might read as 0 or 1.

For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

SPD	Meaning
`0b0`	If EDPRSR.PU is 0, it is not known whether the state of the debug registers in the Core power domain is lost. If EDPRSR.PU is 1, the state of the debug registers in the Core power domain has not been lost.
`0b1`	The state of the debug registers in the Core power domain has been lost.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

When FEAT_DoPD is not implemented and the Core power domain is in either retention or powerdown state, the value of EDPRSR.SPD is IMPLEMENTATION DEFINED. For more information, see 'EDPRSR.SPD when the Core domain is in either retention or powerdown state'.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'1'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Access to this field is UNKNOWN/WI if all of the following are true:
  - IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RC/WI.

PU, bit [0]

When FEAT_DoPD is implemented:

Core powerup status bit.

Access to this field is RAO/WI.

When FEAT_Debugv8p2 is implemented:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
`0b0`	Either the Core power domain is in a low-power or powerdown state, or FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, meaning the debug registers in the Core power domain cannot be accessed.
`0b1`	The Core power domain is in a powerup state, and either FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, meaning the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see ‘EDPRSR.{DLK, R} and reset state’

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see ‘EDPRSR.{DLK, SPD, PU} and the Core power domain’

Access to this field is RO.

Otherwise:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
`0b0`	Core power domain is in a low-power or powerdown state where the debug registers in the Core power domain cannot be accessed.
`0b1`	Core power domain is in a powerup state where the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see ‘EDPRSR.{DLK, R} and reset state’

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

When the Core power domain is powered-up and `DoubleLockStatus()` == TRUE, then the value of EDPRSR.PU is IMPLEMENTATION DEFINED. See the description of the DLK bit for more information.

If FEAT_DoubleLock is implemented, the Core power domain is powered up, and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether this bit reads as 0 or 1.

Access to this field is RO.

Accessing EDPRSR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, then:
  - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
  - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If FEAT_DoPD is not implemented and the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all UNKNOWN, and are either reset or restored on being powered up.
- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

The clearing of bits is an indirect write to EDPRSR.

EDPRSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x314`	EDPRSR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.44 EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

Purpose

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

Configuration

EDRCR is in the Core power domain

Attributes

EDRCR is a 32-bit register.

Field descriptions

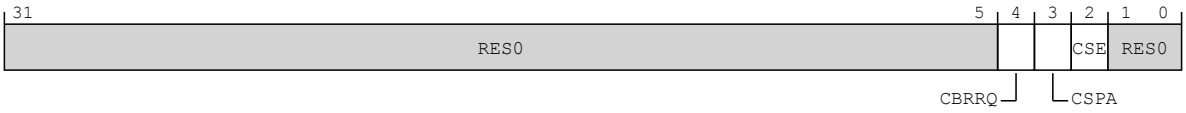

Bits [31:5]

Reserved, RES0.

CBRRQ, bit [4]

Allow imprecise entry to Debug state. The actions on writing to this bit are:

CBRRQ	Meaning
0b0	No action.
0b1	Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

CSPA, bit [3]

Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:

CSPA	Meaning
0b0	No action.
0b1	Clear the EDSCR.PipeAdv bit to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSE, bit [2]

Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:

CSE	Meaning
`0b0`	No action.
`0b1`	Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.

Bits [1:0]

Reserved, RES0.

Accessing EDRCR

EDRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x090`	EDRCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.45 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

EDSCR is in the Core power domain

When FEAT_AA64 is implemented, External register EDSCR bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to AArch64 System register MDSCR_EL1[31:29, 27:26, 23:21, 19, 14, 6].

External register EDSCR bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to AArch32 System register DBGDSCRext[31:29, 27:26, 23:21, 19, 14, 6].

When FEAT_AA64 is implemented, External register EDSCR bits [30:29] are architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

External register EDSCR bits [30:29] are architecturally mapped to AArch32 System register DBGDSCRint[30:29].

Attributes

EDSCR is a 32-bit register.

Field descriptions

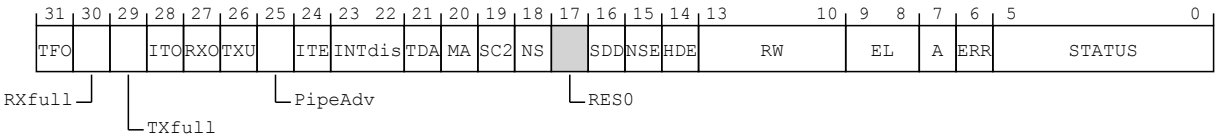

TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter Override. Overrides the Trace Filter controls allowing the external debugger to trace any visible Exception level.

TFO	Meaning
0b0	Trace Filter controls are not affected.
0b1	Trace Filter controls in TRFCR_EL1 and TRFCR_EL2 are ignored. Trace Filter controls TRFCR and HTRFCR are ignored.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

This bit is ignored by the PE when any of the following is true:

- `ExternalSecureNoninvasiveDebugEnabled()` is FALSE and the Effective value of MDCR_EL3.STE is 1.
- FEAT_RME is implemented, `ExternalRealmNoninvasiveDebugEnabled()` is FALSE, and the Effective value of MDCR_EL3.RLTE is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

DTRRX full.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

TXfull, bit [29]

DTRTX full.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

ITO, bit [28]

ITR overrun. Set to 0 on entry to Debug state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

RXO, bit [27]

DTRRX overrun.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

TXU, bit [26]

DTRTX underrun.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PipeAdv, bit [25]

Pipeline Advance. Indicates that software execution is progressing.

PipeAdv	Meaning
0b0	No progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.
0b1	Progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.

The architecture does not define precisely when this field is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing. For example, a PE might set this field to 1 each time the PE retires one or more instructions, or at periodic intervals during the progression of an instruction.

When FEAT_MOPS is implemented, `CPY`, `CPYF`, `SET`, and `SETG` are examples of instructions that periodically make forward progress.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

ITE, bit [24]

ITR empty.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

INTdis, bits [23:22]

When FEAT_RME is implemented:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
0b00	This bit has no effect on the masking of interrupts and SError exceptions.
0b01	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked. If ExternalRootInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Root state are masked. If ExternalRealmInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Realm state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MDSCR_EL1.
- DBGDSCRext.

The Effective value of this field is `0b00` when `ExternalInvasiveDebugEnabled()` is FALSE.

When FEAT_RME is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

When FEAT_Debugv8p4 is implemented:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
`0b00`	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
`0b01`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.
- DBGDSCRext.

The Effective value of this field is `0b00` when `ExternalInvasiveDebugEnabled()` is FALSE.

When FEAT_Debugv8p4 is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

Otherwise:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
`0b00`	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
`0b01`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure EL1 are masked.
`0b10`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure EL1 are masked.
`0b11`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.
- DBGDSCRext.

The Effective value of this field is `0b00` when `ExternalInvasiveDebugEnabled()` is FALSE.

Support for the values `0b01` and `0b10` is IMPLEMENTATION DEFINED. If these values are not supported, they are reserved. If programmed with a reserved value, the PE behaves as if EDSCR.INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

TDA, bit [21]

Traps accesses to the following debug registers:

- AArch64: DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1.
- AArch32: DBGBCR<n>, DBGBVR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>.

TDA	Meaning
`0b0`	Accesses to debug registers do not generate a Software Access Debug event.
`0b1`	Accesses to debug registers generate a Software Access Debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

MA, bit [20]

Memory access mode. Controls the use of Memory access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

MA	Meaning
`0b0`	Normal access mode.
`0b1`	Memory access mode.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

SC2, bit [19]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PCSRv8 is implemented, EL2 is implemented, FEAT_Debugv8p1 is implemented, and FEAT_PCSRv8p2 is not implemented:

Sample CONTEXTIDR_EL2. Controls whether the PC Sample-based Profiling Extension samples CONTEXTIDR_EL2 or VTTBR_EL2.VMID.

SC2	Meaning
0b0	Sample VTTBR_EL2.VMID.
0b1	Sample CONTEXTIDR_EL2.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NS, bit [18]

When FEAT_RME is implemented:

Non-secure status. Together with the NSE field, gives the current Security state:

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Non-secure status. In Debug state, gives the current Security state:

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, access to this field is RO.

Bit [17]

Reserved, RES0.

SDD, bit [16]

When FEAT_RME is implemented:

EL3 debug disabled.

On entry to Debug state:

- If entering from EL3, SDD is set to 0.
- Otherwise, SDD is set to the inverse of `ExternalRootInvasiveDebugEnabled()`.

In Debug state, the value of SDD does not change, even if `ExternalRootInvasiveDebugEnabled()` changes.

In Non-debug state, SDD returns the inverse of `ExternalRootInvasiveDebugEnabled()`, regardless of the current Security state of the PE.

Access to this field is RO.

When FEAT_Secure is implemented:

Secure debug disabled.

On entry to Debug state:

- If entering in Secure state, then SDD is set to 0.
- If entering in Non-secure state, then SDD is set to the inverse of `ExternalSecureInvasiveDebugEnabled()`.

In Debug state, the value of the SDD bit does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

In Non-debug state, SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`, regardless of the current Security state of the PE. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, then a context synchronization event is required to guarantee their effect.

Access to this field is RO.

Otherwise:

Reserved, RES1.

NSE, bit [15]

When FEAT_RME is implemented:

Together with the NS field, this field gives the current Security state.

For a description of the values derived by evaluating NS and NSE together, see EDSCR.NS.

In Non-debug state, this bit is UNKNOWN.

Access to this field is RO.

Otherwise:

Reserved, RES0.

HDE, bit [14]

Halting debug enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HDE	Meaning
0b0	Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
0b1	Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

RW, bits [13:10]

Exception level Execution state status. In Debug state, each bit gives the current Execution state of each Exception level.

RW	Meaning	Applies when
0b1111	Any of the following: • The PE is in Non-debug state. • The PE is at EL0 using AArch64. • The PE is not at EL0, and EL1 is using AArch64. If implemented and enabled in the current Security state, EL2 and EL3 are using AArch64.	
0b1110	The PE is in Debug state at EL0. EL0 is using AArch32. EL1 is using AArch64. If implemented and enabled in the current Security state, EL2 and EL3 are using AArch64.	FEAT_AA32 is implemented
0b110x	The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 is enabled in the current Security state and is using AArch64. If implemented, EL3 is using AArch64.	FEAT_AA32 is implemented and EL2 is implemented
0b10xx	The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 is not implemented, disabled in the current Security state, or using AArch32. EL3 is using AArch64.	FEAT_AA32 is implemented and EL3 is implemented
0b0xxx	The PE is in Debug state. All Exception levels are using AArch32.	FEAT_AA32 is implemented

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAO/WI.
- Otherwise, access to this field is RO.

EL, bits [9:8]

Exception level. In Debug state, gives the current Exception level of the PE.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A, bit [7]

SError exception pending. In Debug state, indicates whether an SError exception is pending:

- If EL2 is enabled in the current Security state, the PE is executing at EL0 or EL1, HCR_EL2.TGE is 0 and either HCR_EL2.AMO is 1 or FEAT_DoubleFault2 is implemented and the Effective value of HCRX_EL2.TMEA is 1, a virtual SError exception.
- Otherwise, if FEAT_E3DSE is implemented, the PE is executing at EL0, EL1, or EL2, and SCR_EL3.EnDSE is 1, a delegated SError exception.
- Otherwise, a physical SError exception.

A	Meaning
`0b0`	No SError exception pending.
`0b1`	SError exception pending.

A debugger can read EDSCR to check whether an SError exception is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

ERR, bit [6]

Cumulative error flag. This bit is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

STATUS, bits [5:0]

On entering Debug state, the PE sets this field to indicate the reason for halting.

STATUS	Meaning
`0b000001`	PE is restarting, exiting Debug state.
`0b000010`	PE is in Non-debug state.
`0b000111`	Breakpoint.
`0b010011`	External debug request.
`0b011011`	Halting step, normal.
`0b011111`	Halting step, exclusive.
`0b100011`	OS Unlock Catch.
`0b100111`	Reset Catch.
`0b101011`	Watchpoint.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STATUS	Meaning
`0b101111`	HLT instruction.
`0b110011`	Software access to debug register.
`0b110111`	Exception Catch.
`0b111011`	Halting step, no syndrome.

All other values are reserved.

The PE resets into Non-debug state. However, if the PE enters Debug state immediately after reset, then the reset value is overwritten with the reason for halting.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'000010'`.

Access to this field is RO.

Accessing EDSCR

EDSCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x088`	EDSCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.46 EDSCR2, External Debug Status and Control Register 2

The EDSCR2 characteristics are:

Purpose

Main control register 2 for the debug implementation.

Configuration

EDSCR2 is in the Core power domain

External register EDSCR2 bits [3, 1] are architecturally mapped to AArch64 System register MDSCR_EL1[35, 33].

This register is present only when FEAT_Debugv8p9 is implemented or FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to EDSCR2 are RES0.

Attributes

EDSCR2 is a 32-bit register.

Field descriptions

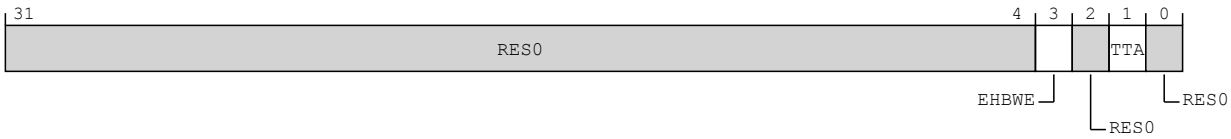

Bits [31:4]

Reserved, RES0.

EHBWE, bit [3]

When FEAT_Debugv8p9 is implemented:

Extended Halting Breakpoint and Watchpoint Enable. Enables use of additional breakpoints or watchpoints.

EHBWE	Meaning
0b0	Halting disabled for Breakpoint and Watchpoint debug events generated by each breakpoint <n> and Watchpoint <n>, where n is greater than or equal to 16.
0b1	Breakpoints and Watchpoint debug events are not affected by this mechanism.

When OSLSR_EL1.OSLK is 1, this field can be read and written through the MDSCR_EL1 System register.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [2]

Reserved, RES0.

TTA, bit [1]

When FEAT_TRBE_EXT is implemented or FEAT_ETEv1p3 is implemented:

Trap Trace Accesses.

Traps access to the following System registers:

AArch64: TRBBASER_EL1, TRBIDR_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBMPAM_EL1, TRBPTR_EL1, TRBSR_EL1, TRBTRG_EL1, TRCACATR<n>, TRCACVR<n>, TRCAUTHSTATUS, TRCAUXCTLR, TRCBBCTLR, TRCCCCTLR, TRCCIDCCTLR0, TRCCIDCCTLR1, TRCCIDCVR<n>, TRCCLAIMCLR, TRCCLAIMSET, TRCCNTCTLR<n>, TRCCNTRLDVR<n>, TRCCNTVR<n>, TRCCONFIGR, TRCDEVARCH, TRCDEVID, TRCEVENTCTL0R, TRCEVENTCTL1R, TRCEXTINSELR<n>, TRCIDR0, TRCIDR1, TRCIDR2, TRCIDR3, TRCIDR4, TRCIDR5, TRCIDR6, TRCIDR7, TRCIDR8, TRCIDR9, TRCIDR10, TRCIDR11, TRCIDR12, TRCIDR13, TRCIMSPEC0, TRCIMSPEC<n>, TRCITEEDCR, TRCOSLSR, TRCPRGCTLR, TRCQCTLR, TRCRSCTLR<n>, TRCRSR, TRCSEQEVR<n>, TRCSEQRSTEVR, TRCSEQSTR, TRCSSCCR<n>, TRCSSCSR<n>, TRCSSPCICR<n>, TRCSTALLCTLR, TRCSTATR, TRCSYNCPR, TRCTRACEIDR, TRCTSCTLR, TRCVICTLR, TRCVIIECTLR, TRCVIPCSSCTLR, TRCVISSCTLR, TRCVMIDCCTLR0, TRCVMIDCCTLR1, and TRCVMIDCVR<n>.

TTA	Meaning
`0b0`	Accesses to trace System registers do not generate a Software Access debug event.
`0b1`	Accesses to trace System registers generate a Software Access debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

When OSLSR_EL1.OSLK is 1, this field can be read and written through the MDSCR_EL1 System register.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [0]

Reserved, RES0.

Accessing EDSCR2

EDSCR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x028`	EDSCR2

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.47 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

Purpose

Contains sampled values captured on reading EDPCSR[31:0].

Configuration

EDVIDSR is in the Core power domain

If FEAT_Debugv8p1 is implemented, the format of this register differs depending on the value of EDSCR.SC2.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

When FEAT_PCSRv8 is implemented in the external debug registers space, if EL2 is not implemented and EL3 is not implemented, it is IMPLEMENTATION DEFINED whether EDVIDSR is implemented.

Note

FEAT_PCSRv8p2 implements the FEAT_PCSRv8 in the Performance Monitors registers space.

This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDVIDSR are RES0.

Attributes

EDVIDSR is a 32-bit register.

Field descriptions

When FEAT_Debugv8p1 is not implemented or EDSCR.SC2 == '0':

31	30	29	28	27 … 16	15 … 8	7 … 0
NS	E2	E3	HV	RES0	VMID[15:8]	VMID

This format applies in all Armv8.0 implementations.

NS, bit [31]

Non-secure state sample. Indicates the Security state associated with the most recent EDPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

NS	Meaning
0b0	Sample is from Secure state.
0b1	Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E2, bit [30]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL2 is implemented:

Exception level 2 status sample. Indicates whether the most recent EDPCSR sample was associated with EL2.

E2	Meaning
0b0	Sample is not from EL2.
0b1	Sample is from EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3, bit [29]

When EL3 is implemented and FEAT_AA64 is implemented:

Exception level 3 status sample. Indicates whether the most recent EDPCSR sample was associated with EL3 using AArch64.

E3	Meaning
0b0	Sample is not from EL3 using AArch64.
0b1	Sample is from EL3 using AArch64.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HV, bit [28]

EDPCSRhi (EDPCSR[63:32]) valid. Indicates whether bits [63:32] of the most recent EDPCSR sample might be nonzero:

HV	Meaning
0b0	Bits[63:32] of the most recent EDPCSR sample are zero.
0b1	Bits[63:32] of the most recent EDPCSR sample might be nonzero.

An EDVIDSR.HV value of 1 does not mean that the value of EDPCSRhi is nonzero. An EDVIDSR.HV value of 0 is a hint that EDPCSRhi (EDPCSR[63:32]) does not need to be read.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When FEAT_VMID16 is implemented and EL2 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

When EL2 is implemented:

VMID sample. The VMID associated with the most recent EDPCSRlo (EDPCSR[31:0]) sample. When the most recent EDPCSR sample was generated:

- This field is RES0 if any of the following apply:
  - The PE is executing in Secure state.
  - The PE is executing at EL2.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When EL2 is implemented, FEAT_Debugv8p1 is implemented, and EDSCR.SC2 == '1':

31 … 0
CONTEXTIDR_EL2

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent EDPCSR sample. When the most recent EDPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing EDVIDSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

EDVIDSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x0A8`	EDVIDSR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.48 EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

Purpose

Returns the virtual data address being accessed when a Watchpoint Debug Event was triggered.

Configuration

EDWAR is in the Core power domain

The value of this register is UNKNOWN if the PE is in Non-debug state, or if EDSCR.STATUS is not `0b101011`.

Attributes

EDWAR is a 64-bit register.

Field descriptions

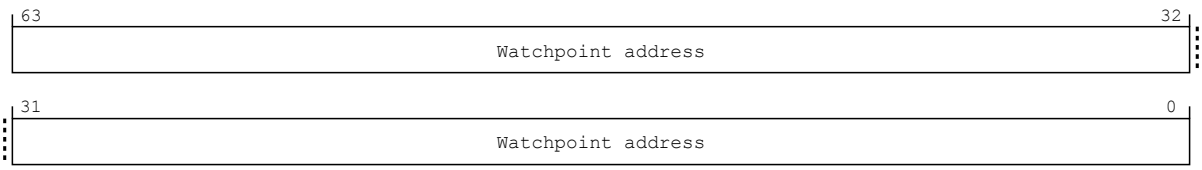

ADDR, bits [63:0]

Watchpoint address. The data virtual address being accessed when a Watchpoint Debug Event was triggered and caused entry to Debug state. This address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

The value of this register is UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a Watchpoint debug event.

The value of EDWAR[63:32] is UNKNOWN if Debug state was entered for a Watchpoint debug event taken from AArch32 state.

The EDWAR is subject to the same alignment rules as the reporting of a watchpointed address in the FAR. See ‘Exception syndrome information and preferred return address’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDWAR

EDWAR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x030`	EDWAR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.49 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

The power domain of MIDR_EL1 is IMPLEMENTATION DEFINED

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch64 System register MIDR_EL1[31:0].

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MIDR[31:0].

Attributes

MIDR_EL1 is a 32-bit register.

Field descriptions

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
Implementer	Variant	Architecture	PartNum	Revision

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
`0b0001`	Armv4.
`0b0010`	Armv4T.
`0b0011`	Armv5 (obsolete).
`0b0100`	Armv5T.
`0b0101`	Armv5TE.
`0b0110`	Armv5TEJ.
`0b0111`	Armv6.
`0b1111`	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR_EL1

MIDR_EL1 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0xD00`	MIDR_EL1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register are IMPLEMENTATION DEFINED.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.50 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

Configuration

OSLAR_EL1 is in the Core power domain

The OS Lock can also be locked or unlocked using DBGOSLAR.

If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether external debug accesses to OSLAR_EL1 are ignored and return an error when `AllowExternalDebugAccess()` returns FALSE for the access.

If FEAT_Debugv8p2 is implemented, external debug accesses to OSLAR_EL1 are ignored and return an error when `AllowExternalDebugAccess()` returns FALSE for the access.

External register OSLAR_EL1 bits [31:0] are architecturally mapped to AArch64 System register OSLAR_EL1[31:0].

Attributes

OSLAR_EL1 is a 32-bit register.

Field descriptions

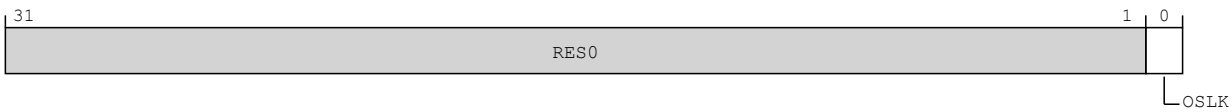

Bits [31:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use EDPRSR.OSLK to check the current status of the lock.

Accessing OSLAR_EL1

OSLAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x300`	OSLAR_EL1

Accessible as follows:

- When (DoubleLockStatus() || (!IsCorePowered())) || ((!AllowExternalDebugAccess(addrdesc)) && IsFeatureImplemented(FEAT_Debugv8p2)), accesses to this register return an ERROR.
- When AllowExternalDebugAccess(addrdesc) and SoftwareLockStatus(addrdesc), accesses to this register are WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When AllowExternalDebugAccess(addrdesc) and !SoftwareLockStatus(addrdesc), accesses to this register are WO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3 External trace registers

This section lists the external Trace unit registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.1 TRCACATR<n>, Trace Address Comparator Access Type Register <n>, n = 0 - 15

The TRCACATR<n> characteristics are:

Purpose

Defines the type of access for the corresponding TRCACVR<n> Register. This register configures the context type, Exception levels, alignment, masking that is applied by the Address Comparator, and how the Address Comparator behaves when it is one half of an Address Range Comparator.

Configuration

External register TRCACATR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCACATR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and (UInt(TRCIDR4.NUMACPAIRS) * 2) > n. Otherwise, direct accesses to TRCACATR<n> are RES0.

Attributes

TRCACATR<n> is a 64-bit register.

Field descriptions

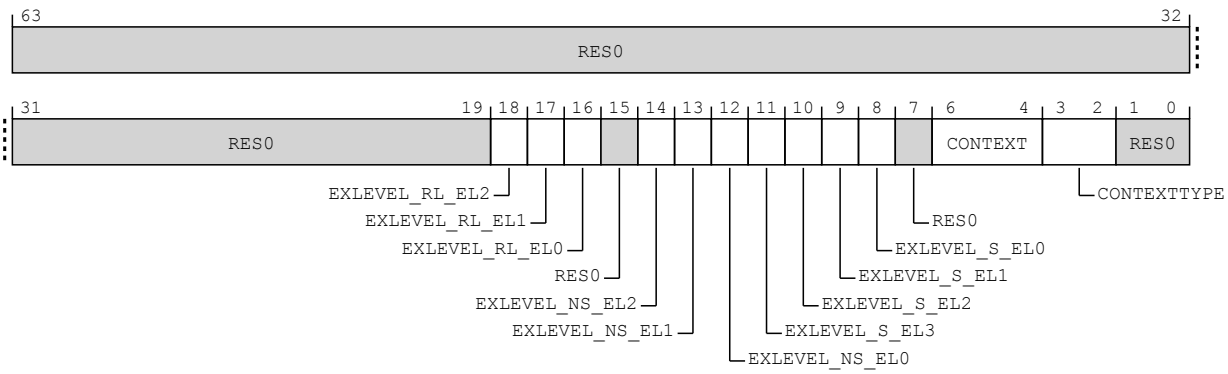

Bits [63:19]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [18]

When FEAT_RME is implemented:

Realm EL2 address comparison control. Controls whether a comparison can occur at EL2 in Realm state.

EXLEVEL_RL_EL2	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator performs comparisons in Realm EL2. When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator does not perform comparisons in Realm EL2.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator does not perform comparisons in Realm EL2. When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator performs comparisons in Realm EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL1, bit [17]

When FEAT_RME is implemented:

Realm EL1 address comparison control. Controls whether a comparison can occur at EL1 in Realm state.

EXLEVEL_RL_EL1	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator performs comparisons in Realm EL1. When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator does not perform comparisons in Realm EL1.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator does not perform comparisons in Realm EL1. When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator performs comparisons in Realm EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [16]

When FEAT_RME is implemented:

Realm EL0 address comparison control. Controls whether a comparison can occur at EL0 in Realm state.

EXLEVEL_RL_EL0	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator performs comparisons in Realm EL0. When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator does not perform comparisons in Realm EL0.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator does not perform comparisons in Realm EL0. When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator performs comparisons in Realm EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [15]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [14]

When Non-secure EL2 is implemented:

Non-secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Non-secure state.

EXLEVEL_NS_EL2	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL2.
0b1	The Address Comparator does not perform comparisons in Non-secure EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [13]

When Non-secure EL1 is implemented:

Non-secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Non-secure state.

EXLEVEL_NS_EL1	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL1.
0b1	The Address Comparator does not perform comparisons in Non-secure EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL0, bit [12]

When Non-secure EL0 is implemented:

Non-secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Non-secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL0	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL0.
0b1	The Address Comparator does not perform comparisons in Non-secure EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL3, bit [11]

When EL3 is implemented:

EL3 address comparison control. Controls whether a comparison can occur at EL3.

EXLEVEL_S_EL3	Meaning
0b0	The Address Comparator performs comparisons at EL3.
0b1	The Address Comparator does not perform comparisons at EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL2, bit [10]

When Secure EL2 is implemented:

Secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Secure state.

EXLEVEL_S_EL2	Meaning
0b0	The Address Comparator performs comparisons in Secure EL2.
0b1	The Address Comparator does not perform comparisons in Secure EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_S_EL1, bit [9]

When Secure EL1 is implemented:

Secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Secure state.

EXLEVEL_S_EL1	Meaning
`0b0`	The Address Comparator performs comparisons in Secure EL1.
`0b1`	The Address Comparator does not perform comparisons in Secure EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL0, bit [8]

When Secure EL0 is implemented:

Secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Secure state.

EXLEVEL_S_EL0	Meaning
`0b0`	The Address Comparator performs comparisons in Secure EL0.
`0b1`	The Address Comparator does not perform comparisons in Secure EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

CONTEXT, bits [6:4]

When TRCIDR4.NUMCIDC != '0000' or TRCIDR4.NUMVMIDC != '0000':

Selects a Context Identifier Comparator or Virtual Context Identifier Comparator:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXT	Meaning	Applies when
0b000	Comparator 0.	
0b001	Comparator 1.	UInt(TRCIDR4.NUMCIDC) > 1 or UInt(TRCIDR4.NUMVMIDC) > 1
0b010	Comparator 2.	UInt(TRCIDR4.NUMCIDC) > 2 or UInt(TRCIDR4.NUMVMIDC) > 2
0b011	Comparator 3.	UInt(TRCIDR4.NUMCIDC) > 3 or UInt(TRCIDR4.NUMVMIDC) > 3
0b100	Comparator 4.	UInt(TRCIDR4.NUMCIDC) > 4 or UInt(TRCIDR4.NUMVMIDC) > 4
0b101	Comparator 5.	UInt(TRCIDR4.NUMCIDC) > 5 or UInt(TRCIDR4.NUMVMIDC) > 5
0b110	Comparator 6.	UInt(TRCIDR4.NUMCIDC) > 6 or UInt(TRCIDR4.NUMVMIDC) > 6
0b111	Comparator 7.	UInt(TRCIDR4.NUMCIDC) > 7 or UInt(TRCIDR4.NUMVMIDC) > 7

The width of this field is dependent on the maximum number of Context Identifier Comparators or Virtual Context Identifier Comparators implemented. Unimplemented bits are RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CONTEXTTYPE, bits [3:2]

When TRCIDR4.NUMCIDC != '0000' or TRCIDR4.NUMVMIDC != '0000':

Controls whether the Address Comparator is dependent on a Context Identifier Comparator, a Virtual Context Identifier Comparator, or both comparisons.

CONTEXTTYPE	Meaning	Applies when
0b00	The Address Comparator is not dependent on the Context Identifier Comparators or Virtual Context Identifier Comparators.	
0b01	The Address Comparator is dependent on the Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Context Identifier Comparator and the address comparison match.	TRCIDR4.NUMCIDC != ‘0000’
0b10	The Address Comparator is dependent on the Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Virtual Context Identifier Comparator and the address comparison match.	TRCIDR4.NUMVMIDC != ‘0000’

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXTTYPE	Meaning	Applies when
`0b11`	The Address Comparator is dependent on the Context Identifier Comparator and Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if the Context Identifier Comparator, the Virtual Context Identifier Comparator, and address comparison all match.	TRCIDR4.NUMCIDC != ‘0000’ and TRCIDR4.NUMVMIDC != ‘0000’

If TRCIDR4.NUMCIDC == `0b0000`, then bit [2] is RES0.

If TRCIDR4.NUMVMIDC == `0b0000`, then bit [3] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [1:0]

Reserved, RES0.

Accessing TRCACATR<n>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<a>.GROUP == `0b0100` and TRCRSCTLR<a>.SAC[n] == 1.
- TRCRSCTLR<a>.GROUP == `0b0101` and TRCRSCTLR<a>.ARC[n/2] == 1.
- TRCVIIECTLR.EXCLUDE[n/2] == 1.
- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
- TRCVISSCTLR.STOP[n] == 1.
- TRCSSCCR<>.ARC[n/2] == 1.
- TRCSSCCR<>.SAC[n] == 1.
- TRCQCTLR.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCACATR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x480 + (8 * n)`	TRCACATR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.2 TRCACVR<n>, Trace Address Comparator Value Register <n>, n = 0 - 15

The TRCACVR<n> characteristics are:

Purpose

Contains the address value.

Configuration

External register TRCACVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCACVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and (UInt(TRCIDR4.NUMACPAIRS) * 2) > n. Otherwise, direct accesses to TRCACVR<n> are RES0.

Attributes

TRCACVR<n> is a 64-bit register.

Field descriptions

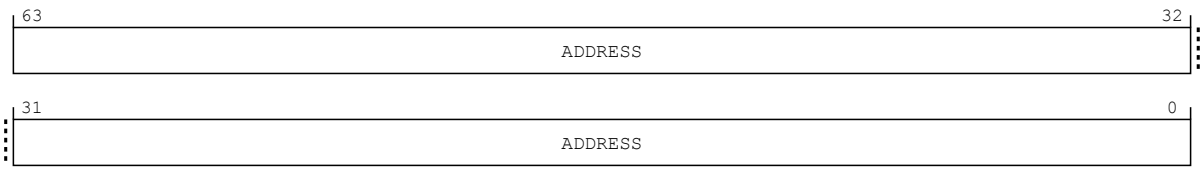

ADDRESS, bits [63:0]

Address Value.

The Address Comparators can support implementations that use multiple address widths. When the trace unit compares the ADDRESS field with an address that has a width less than this field, then the address must be zero-extended to the ADDRESS field width. The trace unit then compares all implemented bits. For example, in a system that supports both 32-bit and 64-bit addresses, when the PE is in AArch32 state the comparator must zero-extend the 32-bit address and compare against the full 64 bits that are stored in TRCACVR<n>.ADDRESS. This requires that the trace analyzer always programs all implemented bits of TRCACVR<n>.ADDRESS.

The result of writing a value other than all zeros or all ones to ADDRESS at bits[63:P] is an UNKNOWN value, where P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The result of writing a value of all zeros or all ones to ADDRESS at bits[63:P] is the written value, and a read of the register returns the written value.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCACVR<n>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<a>.GROUP == `0b0100` and TRCRSCTLR<a>.SAC[n] == 1.
- TRCRSCTLR<a>.GROUP == `0b0101` and TRCRSCTLR<a>.ARC[n/2] == 1.
- TRCVIIECTLR.EXCLUDE[n/2] == 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
- TRCVISSCTLR.STOP[n] == 1.
- TRCSSCCR<a>.ARC[n/2] == 1.
- TRCSSCCR<a>.SAC[n] == 1.
- TRCQCTLR.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCACVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x400 + (8 * n)`	TRCACVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.3 TRCAUTHSTATUS, Trace Authentication Status Register

The TRCAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCAUTHSTATUS bits [31:0] are architecturally mapped to AArch64 System register TRCAUTHSTATUS[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCAUTHSTATUS are RES0.

Attributes

TRCAUTHSTATUS is a 32-bit register.

Field descriptions

31 28	27 26	25 24	23 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
RES0	RTNID	RTID	RES0	RLNID	RLID	HNID	HID	SNID	SID	NSNID	NSID

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTNID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
0b00	Not implemented.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLNID.

RLID, bits [13:12]

Realm invasive debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLID	Meaning
0b00	Not implemented.

HNID, bits [11:10]

Hyp Non-invasive Debug. Indicates whether a separate enable control for EL2 non-invasive debug features is implemented and enabled.

HNID	Meaning
0b00	Separate Hyp non-invasive debug enable not implemented, or EL2 non-invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

HID, bits [9:8]

Hyp Invasive Debug. Indicates whether a separate enable control for EL2 invasive debug features is implemented and enabled.

HID	Meaning
0b00	Separate Hyp invasive debug enable not implemented, or EL2 invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

SNID, bits [7:6]

Secure Non-invasive Debug. Indicates whether Secure non-invasive debug features are implemented and enabled.

SNID	Meaning
0b00	Secure non-invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Secure state is implemented, this field reads as `0b10` or `0b11` depending whether Secure non-invasive debug is enabled.

When Secure state is not implemented, this field reads as `0b00`.

SID, bits [5:4]

Secure Invasive Debug. Indicates whether Secure invasive debug features are implemented and enabled.

SID	Meaning
`0b00`	Secure invasive debug features not implemented.
`0b10`	Implemented and disabled.
`0b11`	Implemented and enabled.

All other values are reserved.

This field reads as `0b00`.

NSNID, bits [3:2]

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug features are implemented and enabled.

NSNID	Meaning
`0b00`	Non-secure non-invasive debug features not implemented.
`0b10`	Implemented and disabled.
`0b11`	Implemented and enabled.

All other values are reserved.

When Non-secure state is implemented, this field reads as `0b11`.

When Non-secure state is not implemented, this field reads as `0b00`.

NSID, bits [1:0]

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug features are implemented and enabled.

NSID	Meaning
`0b00`	Non-secure invasive debug features not implemented.
`0b10`	Implemented and disabled.
`0b11`	Implemented and enabled.

All other values are reserved.

This field reads as `0b00`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCAUTHSTATUS

For implementations that support multiple access mechanisms, different access mechanisms can return different values for reads of TRCAUTHSTATUS if the authentication signals have changed and that change has not yet been synchronized by a Context synchronization event. This scenario can happen if, for example, the external debugger view is implemented separately from the system instruction view to allow for separate power domains, and so observes changes on the signals differently.

External debugger accesses to this register are unaffected by the OS Lock.

TRCAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFB8`	TRCAUTHSTATUS

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.4 TRCAUXCTLR, Trace Auxiliary Control Register

The TRCAUXCTLR characteristics are:

Purpose

The function of this register is IMPLEMENTATION DEFINED.

Configuration

External register TRCAUXCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCAUXCTLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCAUXCTLR are RES0.

Attributes

TRCAUXCTLR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to the expression `0x00000000`.

Accessing TRCAUXCTLR

If this register is nonzero then it might cause the behavior of a trace unit to contradict this architecture specification. See the documentation of the specific implementation for information about the IMPLEMENTATION DEFINED support for this register.

TRCAUXCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x018`	TRCAUXCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.5 TRCBBCTLR, Trace Branch Broadcast Control Register

The TRCBBCTLR characteristics are:

Purpose

Controls the regions in the memory map where branch broadcasting is active.

Configuration

External register TRCBBCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCBBCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, TRCIDR0.TRCBB == '1', and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCBBCTLR are RES0.

Attributes

TRCBBCTLR is a 32-bit register.

Field descriptions

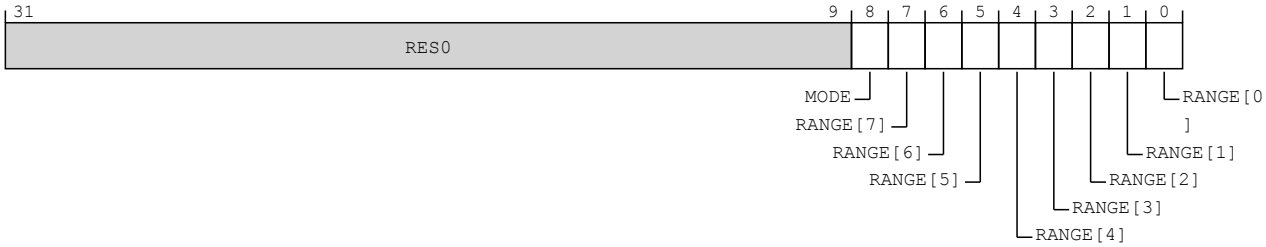

Bits [31:9]

Reserved, RES0.

MODE, bit [8]

Mode.

MODE	Meaning
0b0	Exclude Mode. Branch broadcasting is not active for instructions in the address ranges defined by TRCBBCTLR.RANGE. If TRCBBCTLR.RANGE == 0x00 then branch broadcasting is active for all instructions.
0b1	Include Mode. Branch broadcasting is active for instructions in the address ranges defined by TRCBBCTLR.RANGE. If TRCBBCTLR.RANGE == 0x00 then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE. That is, the trace unit might or might not consider any instructions to be in a branch broadcasting region.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RANGE[<m>] , bits [m], for m = 7 to 0

Selects whether Address Range Comparator <m> is used with branch broadcasting.

RANGE[<m>]	Meaning
`0b0`	The address range that Address Range Comparator <m> defines, is not selected.
`0b1`	The address range that Address Range Comparator <m> defines, is selected.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCBBCTLR

Must be programmed if TRCCONFIGR.BB == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCBBCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x03C`	TRCBBCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.6 TRCCCCTLR, Trace Cycle Count Control Register

The TRCCCCTLR characteristics are:

Purpose

Set the threshold value for cycle counting.

Configuration

External register TRCCCCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCCCCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR0.TRCCCI == '1'. Otherwise, direct accesses to TRCCCCTLR are RES0.

Attributes

TRCCCCTLR is a 32-bit register.

Field descriptions

31 … 12	11 … 0
RES0	THRESHOLD

Bits [31:12]

Reserved, RES0.

THRESHOLD, bits [11:0]

Sets the threshold value for instruction trace cycle counting.

The minimum threshold value that can be programmed into THRESHOLD is given in TRCIDR3.CCITMIN. If the THRESHOLD value is smaller than the value in TRCIDR3.CCITMIN then the behavior is CONSTRAINED UNPREDICTABLE. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

Writing a value of zero when TRCCONFIGR.CCI enables instruction trace cycle counting results in CONSTRAINED UNPREDICTABLE behavior. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCCCTLR

Must be programmed if TRCCONFIGR.CCI == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCCCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x038	TRCCCCTLR

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.7 TRCCIDCCTLR0, Trace Context Identifier Comparator Control Register 0

The TRCCIDCCTLR0 characteristics are:

Purpose

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 0 to 3.

Configuration

External register TRCCIDCCTLR0 bits [31:0] are architecturally mapped to AArch64 System register TRCCIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMCIDC) > `0x0`, and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR0 are RES0.

Attributes

TRCCIDCCTLR0 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP3[<m>]	COMP2[<m>]	COMP1[<m>]	COMP0[<m>]

COMP3[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 3:

TRCCIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR3. Each bit in this field corresponds to a byte in TRCCIDCVR3.

COMP3[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP2[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMCIDC) > 2:

TRCCIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR2. Each bit in this field corresponds to a byte in TRCCIDCVR2.

COMP2[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP1[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 1:

TRCCIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR1. Each bit in this field corresponds to a byte in TRCCIDCVR1.

COMP1[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP0[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 0:

TRCCIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR0. Each bit in this field corresponds to a byte in TRCCIDCVR0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP0[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCCIDCCTLR0

If software uses the TRCCIDCVR<n> registers, for n = 0 to 3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCCTLR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x680	TRCCIDCCTLR0

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.8 TRCCIDCCTLR1, Trace Context Identifier Comparator Control Register 1

The TRCCIDCCTLR1 characteristics are:

Purpose

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 4 to 7.

Configuration

External register TRCCIDCCTLR1 bits [31:0] are architecturally mapped to AArch64 System register TRCCIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMCIDC) > `0x4`, and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR1 are RES0.

Attributes

TRCCIDCCTLR1 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP7[<m>]	COMP6[<m>]	COMP5[<m>]	COMP4[<m>]

COMP7[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 7:

TRCCIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR7. Each bit in this field corresponds to a byte in TRCCIDCVR7.

COMP7[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP6[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMCIDC) > 6:

TRCCIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR6. Each bit in this field corresponds to a byte in TRCCIDCVR6.

COMP6[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP5[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 5:

TRCCIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR5. Each bit in this field corresponds to a byte in TRCCIDCVR5.

COMP5[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP4[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 4:

TRCCIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR4. Each bit in this field corresponds to a byte in TRCCIDCVR4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP4[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCCIDCCTLR1

If software uses the TRCCIDCVR<n> registers, for n = 4 to 7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCCTLR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x684	TRCCIDCCTLR1

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.9 TRCCIDCVR<n>, Trace Context Identifier Comparator Value Registers <n>, n = 0 - 7

The TRCCIDCVR<n> characteristics are:

Purpose

Contains a Context identifier value.

Configuration

External register TRCCIDCVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCCIDCVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMCIDC) > n. Otherwise, direct accesses to TRCCIDCVR<n> are RES0.

Attributes

TRCCIDCVR<n> is a 64-bit register.

Field descriptions

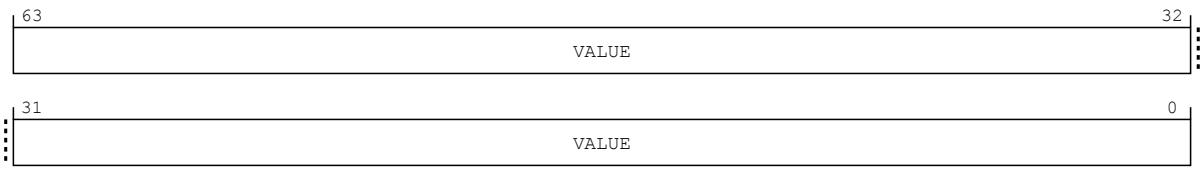

VALUE, bits [63:0]

Context identifier value. The width of this field is indicated by TRCIDR2.CIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Context identifier is zero until the PE updates the Context identifier.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCIDCVR<n>

Must be programmed if any of the following are true:

- TRCRSCTLR<a>.GROUP == `0b0110` and TRCRSCTLR<a>.CID[n] == 1.
- TRCACATR<a>.CONTEXTTYPE == `0b01` or `0b11` and TRCACATR<a>.CONTEXT == n.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x600 + (8 * n)`	TRCCIDCVR<n>

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.10 TRCCIDR0, Trace Component Identification Register 0

The TRCCIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR0 are RES0.

Attributes

TRCCIDR0 is a 32-bit register.

Field descriptions

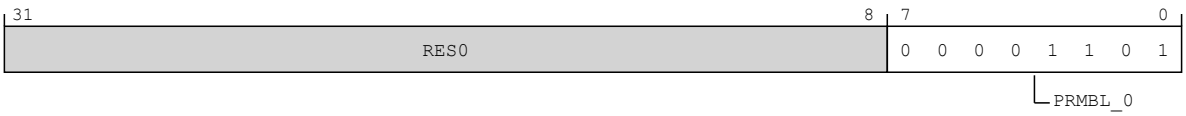

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Component identification preamble, segment 0.

Reads as `0x0D`

Access to this field is RO.

Accessing TRCCIDR0

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFF0`	TRCCIDR0

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.11 TRCCIDR1, Trace Component Identification Register 1

The TRCCIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR1 are RES0.

Attributes

TRCCIDR1 is a 32-bit register.

Field descriptions

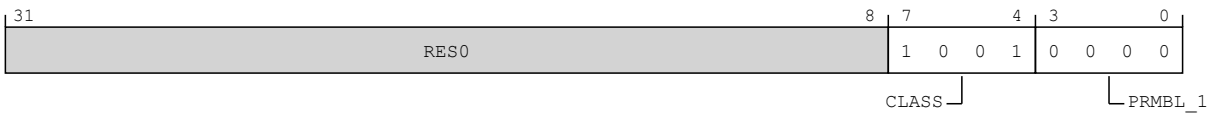

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight peripheral.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Component identification preamble, segment 1.

Reads as 0b0000

Access to this field is RO.

Accessing TRCCIDR1

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR1 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0xFF4`	TRCCIDR1

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.12 TRCCIDR2, Trace Component Identification Register 2

The TRCCIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR2 are RES0.

Attributes

TRCCIDR2 is a 32-bit register.

Field descriptions

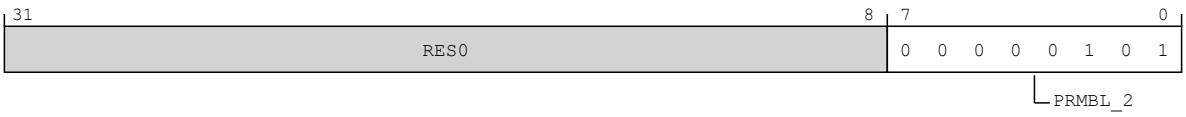

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Component identification preamble, segment 2.

Reads as `0x05`

Access to this field is RO.

Accessing TRCCIDR2

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFF8`	TRCCIDR2

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.13 TRCCIDR3, Trace Component Identification Register 3

The TRCCIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR3 are RES0.

Attributes

TRCCIDR3 is a 32-bit register.

Field descriptions

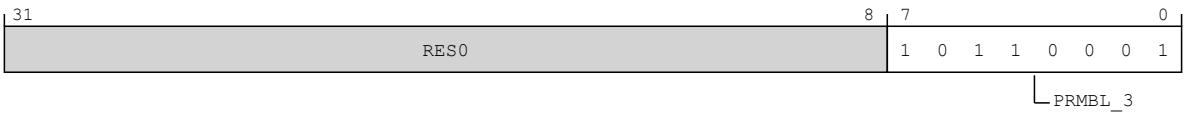

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Component identification preamble, segment 3.

Reads as `0xB1`

Access to this field is RO.

Accessing TRCCIDR3

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFFC`	TRCCIDR3

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.14 TRCCLAIMCLR, Trace Claim Tag Clear Register

The TRCCLAIMCLR characteristics are:

Purpose

In conjunction with TRCCLAIMSET, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMCLR[31:0].

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMSET[31:0].

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to External register TRCCLAIMSET[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCLAIMCLR are RES0.

Attributes

TRCCLAIMCLR is a 32-bit register.

Field descriptions

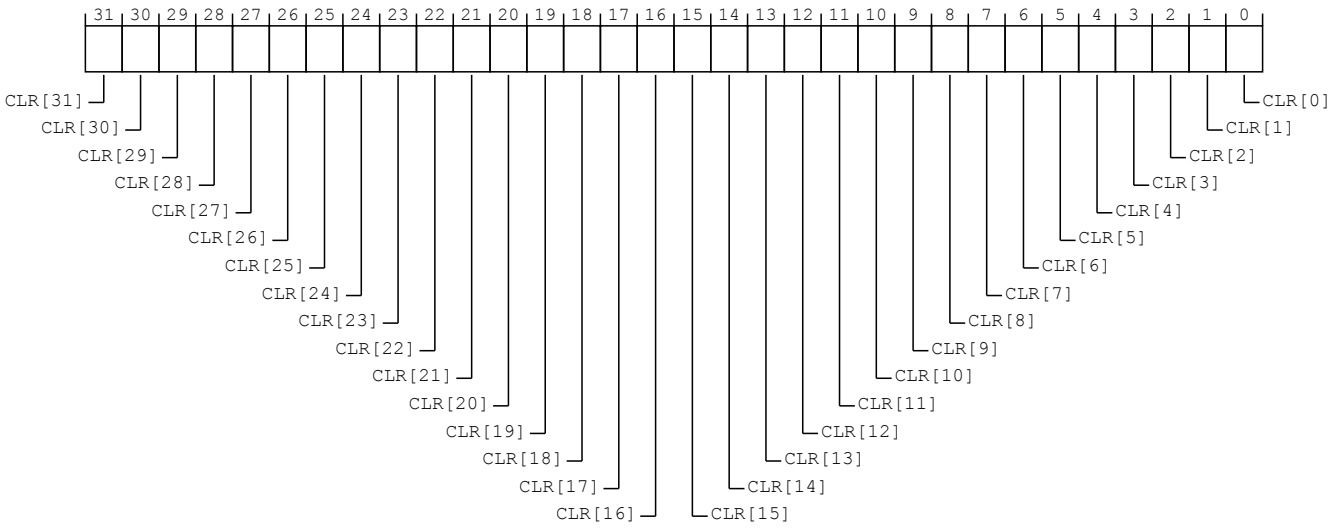

CLR[<m>] , bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLR[<m>]	Meaning
`0b0`	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
`0b1`	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in TRCCLAIMSET.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When m >= NUM_TRC_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing TRCCLAIMCLR

TRCCLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFA4`	TRCCLAIMCLR

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.15 TRCCLAIMSET, Trace Claim Tag Set Register

The TRCCLAIMSET characteristics are:

Purpose

In conjunction with TRCCLAIMCLR, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configuration

The number of claim tag bits implemented is IMPLEMENTATION DEFINED. Arm recommends that implementations support a minimum of four claim tag bits, that is, SET[3:0] reads as `0b1111`.

External register TRCCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMSET[31:0].

External register TRCCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMCLR[31:0].

External register TRCCLAIMSET bits [31:0] are architecturally mapped to External register TRCCLAIMCLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCLAIMSET are RES0.

Attributes

TRCCLAIMSET is a 32-bit register.

Field descriptions

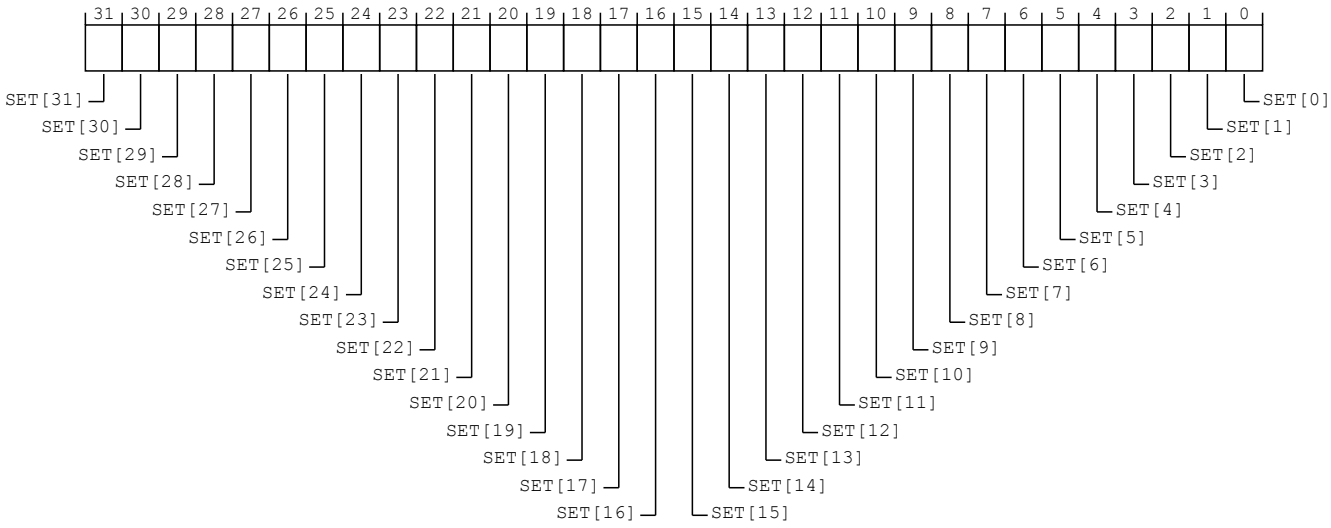

SET[<m>] , bits [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET[<m>]	Meaning
`0b0`	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
`0b1`	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

Accessing this field has the following behavior:

- When m >= NUM_TRC_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/W1S.

Accessing TRCCLAIMSET

TRCCLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFA0`	TRCCLAIMSET

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.16 TRCCNTCTLR<n>, Trace Counter Control Register <n>, n = 0 - 3

The TRCCNTCTLR<n> characteristics are:

Purpose

Controls the operation of Counter <n>.

Configuration

External register TRCCNTCTLR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTCTLR<n> are RES0.

Attributes

TRCCNTCTLR<n> is a 32-bit register.

Field descriptions

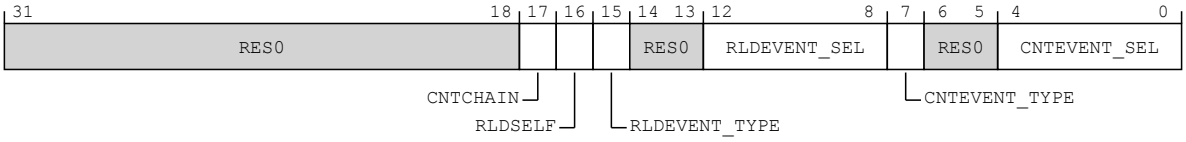

Bits [31:18]

Reserved, RES0.

CNTCHAIN, bit [17]

When (n MOD 2) != 0:

For TRCCNTCTLR3 and TRCCNTCTLR1, this field controls whether the Counter decrements when a reload event occurs for Counter <n-1>.

CNTCHAIN	Meaning
0b0	The Counter does not decrement when a reload event for Counter <n-1> occurs.
0b1	Counter <n> decrements when a reload event for Counter <n-1> occurs. This concatenates Counter <n> and Counter <n-1>, to provide a larger count value.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLDSELF, bit [16]

Controls whether a reload event occurs for the Counter, when the Counter reaches zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLDSELF	Meaning
0b0	Normal mode. The Counter is in Normal mode.
0b1	Self-reload mode. The Counter is in Self-reload mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RLDEVENT_TYPE, bit [15]

Selects an event, that when it occurs causes a reload event for Counter <n>

Chooses the type of Resource Selector.

RLDEVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCCNTCTLR<n>.RLDEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCCNTCTLR<n>.RLDEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR<n>.RLDEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

RLDEVENT_SEL, bits [12:8]

Selects an event, that when it occurs causes a reload event for Counter <n>

Defines the selected Resource Selector or pair of Resource Selectors. TRCCNTCTLR<n>.RLDEVENT.TYPE controls whether TRCCNTCTLR<n>.RLDEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

CNTEVENT_TYPE, bit [7]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Selects an event, that when it occurs causes Counter <n> to decrement.

Chooses the type of Resource Selector.

CNTEVENT_TYPE	Meaning
`0b0`	A single Resource Selector. TRCCNTCTLR<n>.CNTEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCCNTCTLR<n>.CNTEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR<n>.CNTEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

CNTEVENT_SEL, bits [4:0]

Selects an event, that when it occurs causes Counter <n> to decrement.

Defines the selected Resource Selector or pair of Resource Selectors. TRCCNTCTLR<n>.CNTEVENT.TYPE controls whether TRCCNTCTLR<n>.CNTEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTCTLR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCNTCTLR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x150 + (4 * n)`	TRCCNTCTLR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.17 TRCCNTRLDVR<n>, Trace Counter Reload Value Register <n>, n = 0 - 3

The TRCCNTRLDVR<n> characteristics are:

Purpose

This sets or returns the reload count value for Counter <n>.

Configuration

External register TRCCNTRLDVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTRLDVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTRLDVR<n> are RES0.

Attributes

TRCCNTRLDVR<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	VALUE

Bits [31:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the reload value for Counter <n>. When a reload event occurs for Counter <n> then the trace unit copies the VALUE<n> field into Counter <n>.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTRLDVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCNTRLDVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x140 + (4 * n)`	TRCCNTRLDVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.18 TRCCNTVR<n>, Trace Counter Value Register <n>, n = 0 - 3

The TRCCNTVR<n> characteristics are:

Purpose

This sets or returns the value of Counter <n>.

Configuration

External register TRCCNTVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTVR<n> are RES0.

Attributes

TRCCNTVR<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	VALUE

Bits [31:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the count value of Counter.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCCNTVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x160 + (4 * n)`	TRCCNTVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.19 TRCCONFIGR, Trace Configuration Register

The TRCCONFIGR characteristics are:

Purpose

Controls the tracing options.

Configuration

External register TRCCONFIGR bits [31:0] are architecturally mapped to AArch64 System register TRCCONFIGR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCONFIGR are RES0.

Attributes

TRCCONFIGR is a 32-bit register.

Field descriptions

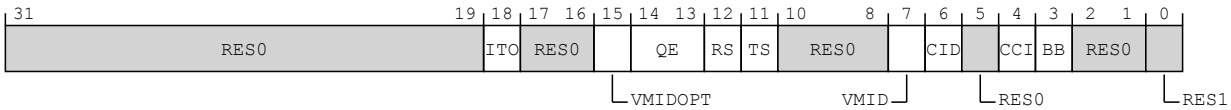

Bits [31:19]

Reserved, RES0.

ITO, bit [18]

When TRCIDR0.ITE == '1':

Instrumentation Trace Override.

ITO	Meaning
0b0	Instrumentation Trace Override disabled.
0b1	Instrumentation Trace Override enabled.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [17:16]

Reserved, RES0.

VMIDOPT, bit [15]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR2.VMIDOPT == '01':

Virtual context identifier selection control.

VMIDOPT	Meaning
0b0	VTTBR_EL2.VMID is used as the Virtual context identifier.
0b1	CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

When TRCIDR2.VMIDOPT == '00':

Reserved, RES0.

Virtual context identifier selection control.

VTTBR_EL2.VMID is used as the Virtual context identifier.

When TRCIDR2.VMIDOPT == '10':

Reserved, RES1.

Virtual context identifier selection control.

CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

Otherwise:

Reserved, RES0.

QE, bits [14:13]

When TRCIDR0.QSUPP == '01':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b01	Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.

All other values are reserved.

When TRCIDR0.QSUPP == '10':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b11	Q elements with instruction counts are enabled. Q elements without instruction counts are enabled.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR0.QSUPP == '11':

Q element generation control.

QE	Meaning
`0b00`	Q elements are disabled.
`0b01`	Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.
`0b11`	Q elements with instruction counts are enabled. Q elements without instruction counts are enabled.

All other values are reserved.

Otherwise:

Reserved, RES0.

RS, bit [12]

When TRCIDR0.RETSTACK == '1':

Return stack control.

RS	Meaning
`0b0`	Return stack is disabled.
`0b1`	Return stack is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TS, bit [11]

When TRCIDR0.TSSIZE != '00000':

Global timestamp tracing control.

TS	Meaning
`0b0`	Global timestamp tracing is disabled.
`0b1`	Global timestamp tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [10:8]

Reserved, RES0.

VMID, bit [7]

When TRCIDR2.VMIDSIZE != '00000':

Virtual context identifier tracing control.

VMID	Meaning
0b0	Virtual context identifier tracing is disabled.
0b1	Virtual context identifier tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CID, bit [6]

When TRCIDR2.CIDSIZE != '00000':

Context identifier tracing control.

CID	Meaning
0b0	Context identifier tracing is disabled.
0b1	Context identifier tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

CCI, bit [4]

When TRCIDR0.TRCCCI == '1':

Cycle counting instruction tracing control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCI	Meaning
0b0	Cycle counting instruction tracing is disabled.
0b1	Cycle counting instruction tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BB, bit [3]

When TRCIDR0.TRCBB == '1':

Branch broadcasting control.

BB	Meaning
0b0	Branch broadcasting is disabled.
0b1	Branch broadcasting is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [2:1]

Reserved, RES0.

Bit [0]

Reserved, RES1.

Accessing TRCCONFIGR

Must always be programmed.

TRCCONFIGR.QE must be set to 0b00 if TRCCONFIGR.BB is not 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCONFIGR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x010	TRCCONFIGR

Accessible as follows:

- When OSLockStatus(), or !IsTraceCorePowered(), or !AllowExternalTraceAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.20 TRCDEVAFF, Trace Device Affinity Register

The TRCDEVAFF characteristics are:

Purpose

For additional information, see the CoreSight Architecture Specification.

Reads the same value as the MPIDR_EL1 register for the PE that this trace unit has affinity with.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVAFF are RES0.

Attributes

TRCDEVAFF is a 64-bit register.

Field descriptions

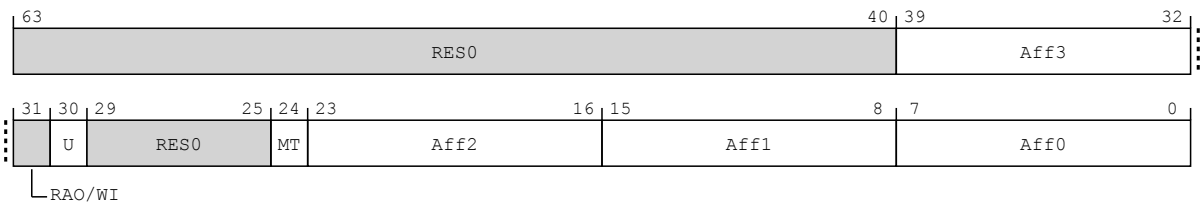

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCDEVAFF

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVAFF can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFA8`	TRCDEVAFF

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.21 TRCDEVARCH, Trace Device Architecture Register

The TRCDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCDEVARCH bits [31:0] are architecturally mapped to AArch64 System register TRCDEVARCH[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVARCH are RES0.

Attributes

TRCDEVARCH is a 32-bit register.

Field descriptions

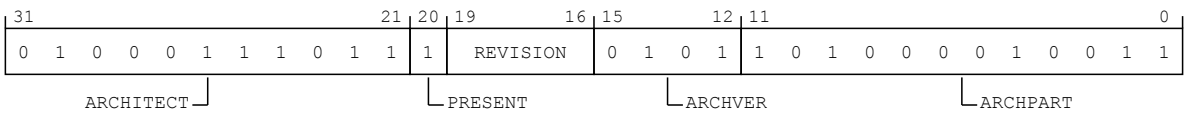

ARCHITECT, bits [31:21]

Defines the architect of the component. For Trace, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the TRCDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	ETEv1.0, FEAT_ETE.
`0b0001`	ETEv1.1, FEAT_ETEv1p1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION	Meaning
0b0010	ETEv1.2, FEAT_ETEv1p2.
0b0011	ETEv1.3, FEAT_ETEv1p3.

All other values are reserved.

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0101	ETEv1.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0x5.

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA13	Arm PE trace architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA13.

Access to this field is RO.

Accessing TRCDEVARCH

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFBC	TRCDEVARCH

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.22 TRCDEVID, Trace Device Configuration Register

The TRCDEVID characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCDEVID bits [31:0] are architecturally mapped to AArch64 System register TRCDEVID[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVID are RES0.

Attributes

TRCDEVID is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCDEVID

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFC8	TRCDEVID

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.23 TRCDEVID1, Trace Device Configuration Register 1

The TRCDEVID1 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVID1 are RES0.

Attributes

TRCDEVID1 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCDEVID1

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFC4	TRCDEVID1

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.24 TRCDEVID2, Trace Device Configuration Register 2

The TRCDEVID2 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVID2 are RES0.

Attributes

TRCDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCDEVID2

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFC0`	TRCDEVID2

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.25 TRCDEVTYPE, Trace Device Type Register

The TRCDEVTYPE characteristics are:

Purpose

Provides discovery information for the component. If the part number field is not recognized, a debugger can report the information that is provided by TRCDEVTYPE about the component instead.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVTYPE are RES0.

Attributes

TRCDEVTYPE is a 32-bit register.

Field descriptions

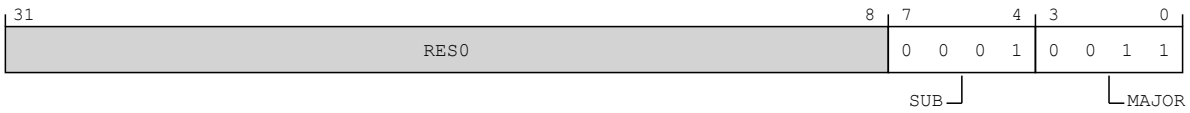

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Component sub-type.

SUB	Meaning
`0b0001`	When MAJOR == `0x3` (Trace source): Associated with a PE.

This field reads as `0x1`.

Access to this field is RO.

MAJOR, bits [3:0]

Component major type.

MAJOR	Meaning
`0b0011`	Trace source.

Other values are defined by the CoreSight Architecture.

This field reads as `0x3`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCDEVTYPE

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFCC`	TRCDEVTYPE

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.26 TRCEVENTCTL0R, Trace Event Control 0 Register

The TRCEVENTCTL0R characteristics are:

Purpose

Controls the generation of ETEEvents.

Configuration

External register TRCEVENTCTL0R bits [31:0] are architecturally mapped to AArch64 System register TRCEVENTCTL0R[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR4.NUMRSPAIR != ‘0000’. Otherwise, direct accesses to TRCEVENTCTL0R are RES0.

Attributes

TRCEVENTCTL0R is a 32-bit register.

Field descriptions

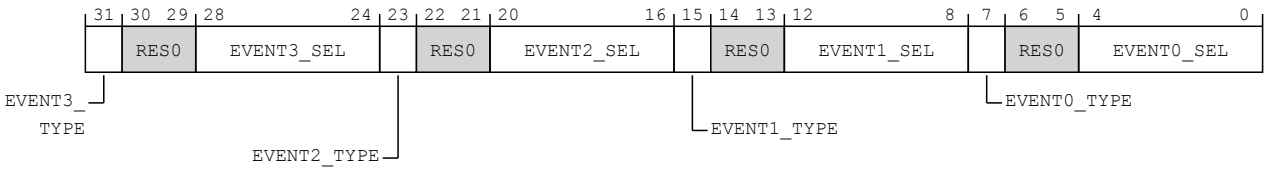

EVENT3_TYPE, bit [31]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 3:

Chooses the type of Resource Selector.

EVENT3_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT3.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT3.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [30:29]

Reserved, RES0.

EVENT3_SEL, bits [28:24]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 3:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[3] == 1, then Event element 3 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.TYPE controls whether TRCEVENTCTL0R.EVENT3.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT2_TYPE, bit [23]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 2:

Chooses the type of Resource Selector.

EVENT2_TYPE	Meaning
`0b0`	A single Resource Selector. TRCEVENTCTL0R.EVENT2.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT2.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [22:21]

Reserved, RES0.

EVENT2_SEL, bits [20:16]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 2:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[2] == 1, then Event element 2 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.TYPE controls whether TRCEVENTCTL0R.EVENT2.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT1_TYPE, bit [15]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 1:

Chooses the type of Resource Selector.

EVENT1_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT1.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT1.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT1.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [14:13]

Reserved, RES0.

EVENT1_SEL, bits [12:8]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 1:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[1] == 1, then Event element 1 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT1.TYPE controls whether TRCEVENTCTL0R.EVENT1.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT0_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT0_TYPE	Meaning
`0b0`	A single Resource Selector. TRCEVENTCTL0R.EVENT0.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT0.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT0.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

EVENT0_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[0] == 1, then Event element 0 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT0.TYPE controls whether TRCEVENTCTL0R.EVENT0.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Accessing TRCEVENTCTL0R

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEVENTCTL0R can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x020`	TRCEVENTCTL0R

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.27 TRCEVENTCTL1R, Trace Event Control 1 Register

The TRCEVENTCTL1R characteristics are:

Purpose

Controls the behavior of the ETEEvents that TRCEVENTCTL0R selects.

Configuration

External register TRCEVENTCTL1R bits [31:0] are architecturally mapped to AArch64 System register TRCEVENTCTL1R[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCEVENTCTL1R are RES0.

Attributes

TRCEVENTCTL1R is a 32-bit register.

Field descriptions

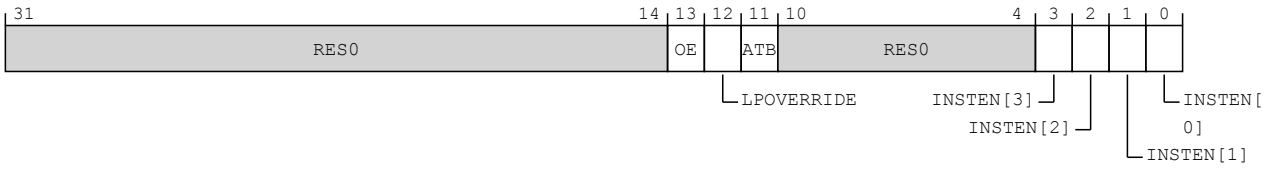

Bits [31:14]

Reserved, RES0.

OE, bit [13]

When TRCIDR5.OE == '1':

ETE Trace Output Enable control.

OE	Meaning
0b0	Trace output to any IMPLEMENTATION DEFINED trace output interface is disabled.
0b1	Trace output to any IMPLEMENTATION DEFINED trace output interface is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

LPOVERRIDE, bit [12]

When TRCIDR5.LPOVERRIDE == '1':

Low-power Override Mode select.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LPOVERRIDE	Meaning
0b0	Trace unit Low-power Override Mode is not enabled. That is, the trace unit is permitted to enter low-power state.
0b1	Trace unit Low-power Override Mode is enabled. That is, entry to a low-power state does not affect the trace unit resources or trace generation.

Otherwise:

Reserved, RES0.

ATB, bit [11]

When TRCIDR5.ATBTRIG == '1':

AMBA Trace Bus (ATB) trigger enable.

If a CoreSight ATB interface is implemented then when ETEEvent 0 occurs the trace unit sets:

- ATID == 0x7D.
- ATDATA to the value of TRCTRACEIDR.

If the width of ATDATA is greater than the width of TRCTRACEIDR.TRACEID then the trace unit zeros the upper ATDATA bits.

If ETEEvent 0 is programmed to occur based on program execution, such as an Address Comparator, the ATB trigger might not be inserted into the ATB stream at the same time as any trace generated by that program execution is output by the trace unit. Typically, the generated trace might be buffered in a trace unit which means that the ATB trigger would be output before the associated trace is output.

If ETEEvent 0 is asserted multiple times in close succession, the trace unit is required to generate an ATB trigger for the first assertion, but might ignore one or more of the subsequent assertions. Arm recommends that the window in which ETEEvent 0 is ignored is limited only by the time taken to output an ATB trigger.

ATB	Meaning
0b0	ATB trigger is disabled.
0b1	ATB trigger is enabled.

Otherwise:

Reserved, RES0.

Bits [10:4]

Reserved, RES0.

INSTEN[<m>] , bits [m], for m = 3 to 0

Event element control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INSTEN[<m>]	Meaning
0b0	The trace unit does not generate an Event element <m>.
0b1	The trace unit generates an Event element <m> when ETEEvent <m> occurs.

Accessing this field has the following behavior:

- When TRCIDR4.NUMRSPAIR == ‘0000’, access to this field is RES0.
- Access to this field is RES0 if all of the following are true:
  - TRCIDR4.NUMRSPAIR != ‘0000’
  - m > UInt(TRCIDR0.NUMEVENT)
- Otherwise, access to this field is RW.

Accessing TRCEVENTCTL1R

Must be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEVENTCTL1R can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x024	TRCEVENTCTL1R

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.28 TRCEXTINSELR<n>, Trace External Input Select Register <n>, n = 0 - 3

The TRCEXTINSELR<n> characteristics are:

Purpose

Use this to set, or read, which External Inputs are resources to the trace unit.

The name TRCEXTINSELR is an alias of TRCEXTINSELR0.

Configuration

External register TRCEXTINSELR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCEXTINSELR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMEXTINSEL) > n. Otherwise, direct accesses to TRCEXTINSELR<n> are RES0.

Attributes

TRCEXTINSELR<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

PMU event to select.

The event number as defined by the Arm ARM.

Software must program this field with a PMU event that is supported by the PE being programmed.

If evtCount is programmed to a PMU event that is reserved or not supported by the PE, the behavior depends on the PMU event type:

- For the range `0x0000` to `0x003F`, then the PMU event is not active, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED PMU events, it is UNPREDICTABLE what PMU event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

UNPREDICTABLE means the PMU event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include a PMU event from a set of common IMPLEMENTATION DEFINED PMU events, then no PMU event is counted and the value read back on evtCount is the value written.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCEXTINSELR<n>

Must be programmed if any of the following is true: TRCRSCTLR<a>.GROUP == `0b0000` and TRCRSCTLR<a>.EXTIN[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCEXTINSELR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x120 + (4 * n)`	TRCEXTINSELR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.29 TRCIDR0, Trace ID Register 0

The TRCIDR0 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR0 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR0[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR0 are RES0.

Attributes

TRCIDR0 is a 32-bit register.

Field descriptions

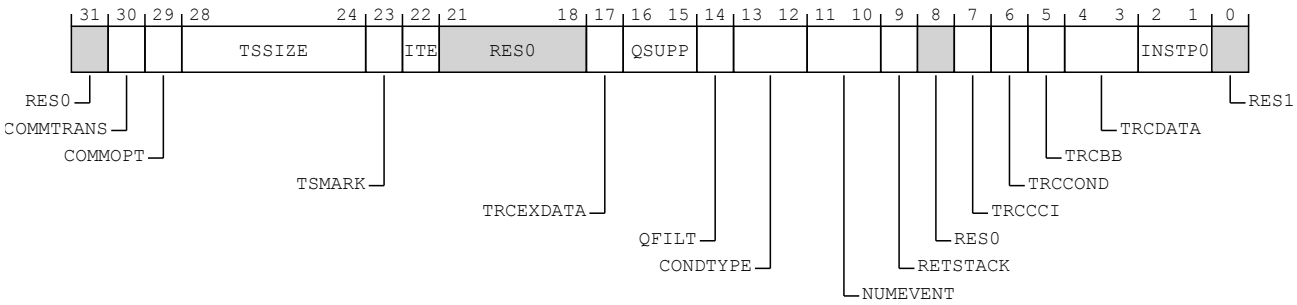

Bit [31]

Reserved, RES0.

COMMTRANS, bit [30]

Transaction Start element behavior.

The value of this field is an IMPLEMENTATION DEFINED choice of:

COMMTRANS	Meaning
0b0	Transaction Start elements are P0 elements.
0b1	Transaction Start elements are not P0 elements.

Access to this field is RO.

COMMOPT, bit [29]

Indicates the contents and encodings of Cycle count packets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMMOPT	Meaning
0b0	Commit mode 0.
0b1	Commit mode 1.

The Commit mode defines the contents and encodings of Cycle Count packets, in particular how Commit elements are indicated by these packets. See the descriptions of these packets for more details.

Accessing this field has the following behavior:

- Access to this field is RAO/WI if all of the following are true:
  - TRCIDR0.TRCCCI == '1'
  - UInt(TRCIDR8.MAXSPEC) == 0x0
- When TRCIDR0.TRCCCI == '0', access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

TSSIZE, bits [28:24]

Indicates that the trace unit implements Global timestamping and the size of the timestamp value.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TSSIZE	Meaning
0b00000	Global timestamping not implemented.
0b01000	Global timestamping implemented with a 64-bit timestamp value.

All other values are reserved.

This field reads as 0b01000.

Access to this field is RO.

TSMARK, bit [23]

Indicates whether Timestamp Marker elements are generated.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TSMARK	Meaning
0b0	Timestamp Marker elements are not generated.
0b1	Timestamp Marker elements are generated.

Access to this field is RO.

ITE, bit [22]

Indicates whether Instrumentation Trace is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITE	Meaning
0b0	Instrumentation Trace not implemented.
0b1	Instrumentation Trace implemented.

This field has the value 1 if FEAT_ITE is implemented.

Access to this field is RO.

Bits [21:18]

Reserved, RES0.

TRCEXDATA, bit [17]

When TRCIDR0.TRCDATA != '00':

Indicates if the trace unit implements tracing of data transfers for exceptions and exception returns. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCEXDATA	Meaning
0b0	Tracing of data transfers for exceptions and exception returns not implemented.
0b1	Tracing of data transfers for exceptions and exception returns implemented.

Access to this field is RO.

Otherwise:

Reserved, RES0.

QSUPP, bits [16:15]

Indicates that the trace unit implements Q element support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

QSUPP	Meaning
0b00	Q element support is not implemented.
0b01	Q element support is implemented, and only supports Q elements with instruction counts.
0b10	Q element support is implemented, and only supports Q elements without instruction counts.
0b11	Q element support is implemented, and supports: • Q elements with instruction counts. • Q elements without instruction counts.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

QFILT, bit [14]

Indicates if the trace unit implements Q element filtering.

The value of this field is an IMPLEMENTATION DEFINED choice of:

QFILT	Meaning
`0b0`	Q element filtering is not implemented.
`0b1`	Q element filtering is implemented.

If TRCIDR0.QSUPP == `0b00` then this field is 0.

Access to this field is RO.

CONDTYPE, bits [13:12]

When TRCIDR0.TRCCOND == '1':

Indicates how conditional instructions are traced. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CONDTYPE	Meaning
`0b00`	Conditional instructions are traced with an indication of whether they pass or fail their condition code check.
`0b01`	Conditional instructions are traced with an indication of the APSR condition flags.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NUMEVENT, bits [11:10]

When TRCIDR4.NUMRSPAIR == '0000':

Indicates the number of ETEEvents implemented.

NUMEVENT	Meaning
`0b00`	The trace unit supports 0 ETEEvents.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR4.NUMRSPAIR != '0000':

Indicates the number of ETEEvents implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMEVENT	Meaning
`0b00`	The trace unit supports 1 ETEEvent.
`0b01`	The trace unit supports 2 ETEEvents.
`0b10`	The trace unit supports 3 ETEEvents.
`0b11`	The trace unit supports 4 ETEEvents.

Access to this field is RO.

Otherwise:

Reserved, RES0.

RETSTACK, bit [9]

Indicates if the trace unit supports the return stack.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RETSTACK	Meaning
`0b0`	Return stack not implemented.
`0b1`	Return stack implemented.

Access to this field is RO.

Bit [8]

Reserved, RES0.

TRCCCI, bit [7]

Indicates if the trace unit implements cycle counting.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCCCI	Meaning
`0b0`	Cycle counting not implemented.
`0b1`	Cycle counting implemented.

This field reads as 1.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCCOND, bit [6]

Indicates if the trace unit implements conditional instruction tracing. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCCOND	Meaning
`0b0`	Conditional instruction tracing not implemented.
`0b1`	Conditional instruction tracing implemented.

This field reads as 0.

Access to this field is RO.

TRCBB, bit [5]

Indicates if the trace unit implements branch broadcasting.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCBB	Meaning
`0b0`	Branch broadcasting not implemented.
`0b1`	Branch broadcasting implemented.

This field reads as 1.

Access to this field is RO.

TRCDATA, bits [4:3]

Indicates if the trace unit implements data tracing. Data tracing is not implemented in ETE and this field is reserved for other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCDATA	Meaning
`0b00`	Data tracing not implemented.
`0b11`	Data tracing implemented.

All other values are reserved.

This field reads as `0b00`.

Access to this field is RO.

INSTP0, bits [2:1]

Indicates if load and store instructions are P0 instructions. Load and store instructions as P0 instructions is not implemented in ETE and this field is reserved for other trace architectures.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

INSTP0	Meaning
`0b00`	Load and store instructions are not P0 instructions.
`0b11`	Load and store instructions are P0 instructions.

All other values are reserved.

When FEAT_ETE is implemented, the only permitted value is `0b00`.

Access to this field is RO.

Bit [0]

Reserved, RES1.

Accessing TRCIDR0

TRCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1E0`	TRCIDR0

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.30 TRCIDR1, Trace ID Register 1

The TRCIDR1 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR1 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR1[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR1 are RES0.

Attributes

TRCIDR1 is a 32-bit register.

Field descriptions

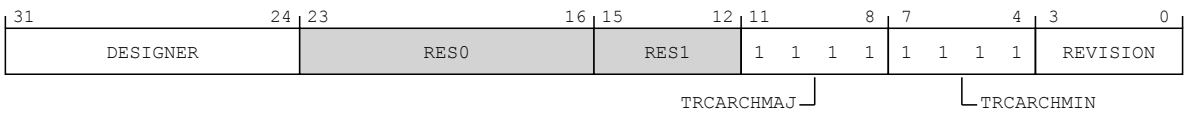

DESIGNER, bits [31:24]

Indicates which company designed the trace unit. The permitted values of this field are the same as MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RES1.

TRCARCHMAJ, bits [11:8]

Major architecture version.

TRCARCHMAJ	Meaning
`0b1111`	If both TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN == `0xF` then refer to TRCDEVARCH.

All other values are reserved.

This field reads as `0b1111`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCARCHMIN, bits [7:4]

Minor architecture version.

TRCARCHMIN	Meaning
`0b1111`	If both TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN == `0xF` then refer to TRCDEVARCH.

All other values are reserved.

This field reads as `0b1111`.

Access to this field is RO.

REVISION, bits [3:0]

Implementation revision.

Returns an IMPLEMENTATION DEFINED value that identifies the revision of the trace unit.

Arm deprecates any use of this field and recommends that implementations set this field to zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCIDR1

TRCIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1E4`	TRCIDR1

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.31 TRCIDR10, Trace ID Register 10

The TRCIDR10 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR10 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR10[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR10 are RES0.

Attributes

TRCIDR10 is a 32-bit register.

Field descriptions

31	0
NUMP1KEY	

NUMP1KEY, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR10

TRCIDR10 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x188`	TRCIDR10

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.32 TRCIDR11, Trace ID Register 11

The TRCIDR11 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR11 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR11[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR11 are RES0.

Attributes

TRCIDR11 is a 32-bit register.

Field descriptions

31	0
NUMP1SPC	

NUMP1SPC, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of special P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR11

TRCIDR11 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x18C	TRCIDR11

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.33 TRCIDR12, Trace ID Register 12

The TRCIDR12 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR12 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR12[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR12 are RES0.

Attributes

TRCIDR12 is a 32-bit register.

Field descriptions

31	0
NUMCONDKEY	

NUMCONDKEY, bits [31:0]

When TRCIDR0.TRCCOND == '1':

Indicates the number of conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR12

TRCIDR12 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x190	TRCIDR12

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.34 TRCIDR13, Trace ID Register 13

The TRCIDR13 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR13 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR13[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR13 are RES0.

Attributes

TRCIDR13 is a 32-bit register.

Field descriptions

31	0
NUMCONDSPC	

NUMCONDSPC, bits [31:0]

When TRCIDR0.TRCCOND == '1':

Indicates the number of special conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR13

TRCIDR13 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x194	TRCIDR13

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.35 TRCIDR2, Trace ID Register 2

The TRCIDR2 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR2 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR2[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR2 are RES0.

Attributes

TRCIDR2 is a 32-bit register.

Field descriptions

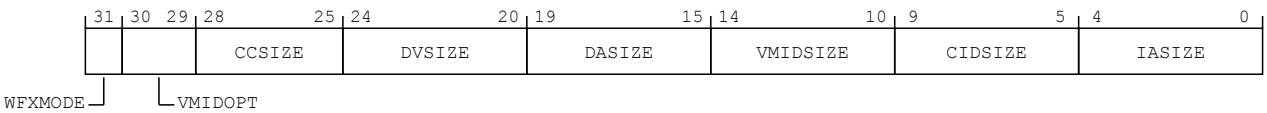

WFXMODE, bit [31]

Indicates whether `WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions:

The value of this field is an IMPLEMENTATION DEFINED choice of:

WFXMODE	Meaning
`0b0`	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are not classified as P0 instructions.
`0b1`	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions.

Access to this field is RO.

VMIDOPT, bits [30:29]

Indicates the options for Virtual context identifier selection.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDOPT	Meaning
`0b00`	Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES0.
`0b01`	Virtual context identifier selection supported. TRCCONFIGR.VMIDOPT is implemented.
`0b10`	Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES1.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If TRCIDR2.VMIDSIZE == `0b00000` then this field is `0b00`.

If TRCIDR2.VMIDSIZE != `0b00000` then this field is `0b10`.

Access to this field is RO.

CCSIZE, bits [28:25]

When TRCIDR0.TRCCCI == '1':

Indicates the size of the cycle counter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCSIZE	Meaning
`0b0000`	The cycle counter is 12 bits in length.
`0b0001`	The cycle counter is 13 bits in length.
`0b0010`	The cycle counter is 14 bits in length.
`0b0011`	The cycle counter is 15 bits in length.
`0b0100`	The cycle counter is 16 bits in length.
`0b0101`	The cycle counter is 17 bits in length.
`0b0110`	The cycle counter is 18 bits in length.
`0b0111`	The cycle counter is 19 bits in length.
`0b1000`	The cycle counter is 20 bits in length.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

DVSIZE, bits [24:20]

When TRCIDR0.TRCDATA != '00':

Indicates the data value size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DVSIZE	Meaning
`0b00000`	Data value tracing not implemented.
`0b00100`	Data value tracing has a maximum of 32-bit data values.
`0b01000`	Data value tracing has a maximum of 64-bit data values.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DASIZE, bits [19:15]

When TRCIDR0.TRCDATA != '00':

Indicates the data address size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DASIZE	Meaning
`0b00000`	Data address tracing not implemented.
`0b00100`	Data address tracing has a maximum of 32-bit data addresses.
`0b01000`	Data address tracing has a maximum of 64-bit data addresses.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

VMIDSIZE, bits [14:10]

Indicates the trace unit Virtual context identifier size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDSIZE	Meaning
`0b00000`	Virtual context identifier tracing is not supported.
`0b00001`	8-bit Virtual context identifier size.
`0b00010`	16-bit Virtual context identifier size.
`0b00100`	32-bit Virtual context identifier size.

All other values are reserved.

If the PE does not implement EL2 then this field is `0b00000`.

If the PE implements EL2 then this field is `0b00100`.

Access to this field is RO.

CIDSIZE, bits [9:5]

Indicates the Context identifier size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CIDSIZE	Meaning
`0b00000`	Context identifier tracing is not supported.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CIDSIZE	Meaning
0b00100	32-bit Context identifier size.

All other values are reserved.

This field reads as 0b00100.

Access to this field is RO.

IASIZE, bits [4:0]

Virtual instruction address size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IASIZE	Meaning
0b00100	Maximum of 32-bit instruction address size.
0b01000	Maximum of 64-bit instruction address size.

All other values are reserved.

This field reads as 0b01000.

Access to this field is RO.

Accessing TRCIDR2

TRCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x1E8	TRCIDR2

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.36 TRCIDR3, Trace ID Register 3

The TRCIDR3 characteristics are:

Purpose

Returns the base architecture of the trace unit.

Configuration

External register TRCIDR3 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR3[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR3 are RES0.

Attributes

TRCIDR3 is a 32-bit register.

Field descriptions

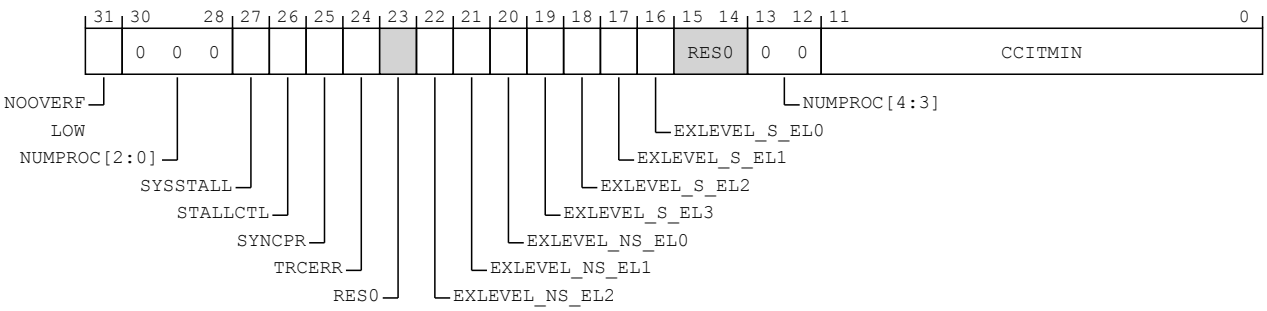

NOOVERFLOW, bit [31]

Indicates if overflow prevention is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NOOVERFLOW	Meaning
0b0	Overflow prevention is not implemented.
0b1	Overflow prevention is implemented.

If TRCIDR3.STALLCTL == 0 then this field is 0.

Access to this field is RO.

NUMPROC, bits [13:12, 30:28]

Indicates the number of PEs available for tracing.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMPROC	Meaning
0b00000	The trace unit can trace one PE.

This field reads as 0b00000.

Access to this field is RO.

SYSSTALL, bit [27]

Indicates if stalling of the PE is permitted.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYSSTALL	Meaning
0b0	Stalling of the PE is not permitted.
0b1	Stalling of the PE is permitted.

The value of this field might be dynamic and change based on system conditions.

If TRCIDR3.STALLCTL == 0 then this field is 0.

Access to this field is RO.

STALLCTL, bit [26]

Indicates if trace unit implements stalling of the PE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

STALLCTL	Meaning
0b0	Stalling of the PE is not implemented.
0b1	Stalling of the PE is implemented.

Access to this field is RO.

SYNCPR, bit [25]

Indicates if an implementation has a fixed synchronization period.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYNCPR	Meaning
0b0	TRCSYNCPR is read/write so software can change the synchronization period.
0b1	TRCSYNCPR is read-only so the synchronization period is fixed.

This field reads as 0.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCERR, bit [24]

Indicates forced tracing of System Error exceptions is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCERR	Meaning
`0b0`	Forced tracing of System Error exceptions is not implemented.
`0b1`	Forced tracing of System Error exceptions is implemented.

This field reads as 1.

Access to this field is RO.

Bit [23]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]

Indicates if Non-secure EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL2	Meaning
`0b0`	Non-secure EL2 is not implemented.
`0b1`	Non-secure EL2 is implemented.

Access to this field is RO.

EXLEVEL_NS_EL1, bit [21]

Indicates if Non-secure EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL1	Meaning
`0b0`	Non-secure EL1 is not implemented.
`0b1`	Non-secure EL1 is implemented.

Access to this field is RO.

EXLEVEL_NS_EL0, bit [20]

Indicates if Non-secure EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL0	Meaning
0b0	Non-secure EL0 is not implemented.
0b1	Non-secure EL0 is implemented.

Access to this field is RO.

EXLEVEL_S_EL3, bit [19]

Indicates if EL3 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL3	Meaning
0b0	EL3 is not implemented.
0b1	EL3 is implemented.

Access to this field is RO.

EXLEVEL_S_EL2, bit [18]

Indicates if Secure EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL2	Meaning
0b0	Secure EL2 is not implemented.
0b1	Secure EL2 is implemented.

Access to this field is RO.

EXLEVEL_S_EL1, bit [17]

Indicates if Secure EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL1	Meaning
0b0	Secure EL1 is not implemented.
0b1	Secure EL1 is implemented.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_S_EL0, bit [16]

Indicates if Secure EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL0	Meaning
`0b0`	Secure EL0 is not implemented.
`0b1`	Secure EL0 is implemented.

Access to this field is RO.

Bits [15:14]

Reserved, RES0.

CCITMIN, bits [11:0]

When TRCIDR0.TRCCCI == '0':

Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

Reads as `0x000`

Access to this field is RO.

When TRCIDR0.TRCCCI == '1':

Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCITMIN	Meaning
`0x001..0xFFF`	The minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

The minimum value of this field is `0x001`.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR3

TRCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1EC`	TRCIDR3

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.37 TRCIDR4, Trace ID Register 4

The TRCIDR4 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR4 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR4[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR4 are RES0.

Attributes

TRCIDR4 is a 32-bit register.

Field descriptions

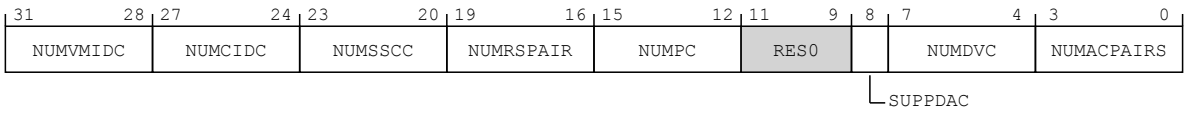

NUMVMIDC, bits [31:28]

Indicates the number of Virtual Context Identifier Comparators that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMVMIDC	Meaning
`0b0000..0b1000`	The number of Virtual Context Identifier Comparators in this implementation.

All other values are reserved.

If the PE does not implement EL2 then this field is `0b0000`.

Access to this field is RO.

NUMCIDC, bits [27:24]

Indicates the number of Context Identifier Comparators that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMCIDC	Meaning
`0b0000..0b1000`	The number of Context Identifier Comparators in this implementation.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMSSCC, bits [23:20]

Indicates the number of Single-shot Comparator Controls that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMSSCC	Meaning
`0b0000..0b1000`	The number of Single-shot Comparator Controls in this implementation.

All other values are reserved.

Access to this field is RO.

NUMRSPAIR, bits [19:16]

Indicates the number of resource selector pairs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMRSPAIR	Meaning
`0b0000`	This implementation has zero resource selector pairs.
`0b0001..0b1111`	The number of resource selector pairs in this implementation, minus one.

All other values are reserved.

Access to this field is RO.

NUMPC, bits [15:12]

Indicates the number of PE Comparator Inputs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMPC	Meaning
`0b0000..0b1000`	The number of PE Comparator Inputs in this implementation.

All other values are reserved.

Access to this field is RO.

Bits [11:9]

Reserved, RES0.

SUPPDAC, bit [8]

When TRCIDR4.NUMACPAIRS != '0000':

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates whether data address comparisons are implemented. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SUPPDAC	Meaning
`0b0`	Data address comparisons not implemented.
`0b1`	Data address comparisons implemented.

This field reads as 0.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NUMDVC, bits [7:4]

Indicates the number of data value comparators. Data value comparators are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMDVC	Meaning
`0b0000..0b1000`	The number of data value comparators in this implementation.

All other values are reserved.

This field reads as `0b0000`.

Access to this field is RO.

NUMACPAIRS, bits [3:0]

Indicates the number of Address Comparator pairs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMACPAIRS	Meaning
`0b0000..0b1000`	The number of Address Comparator pairs in this implementation.

All other values are reserved.

Access to this field is RO.

Accessing TRCIDR4

TRCIDR4 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0x1F0`	TRCIDR4

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.38 TRCIDR5, Trace ID Register 5

The TRCIDR5 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR5 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR5[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR5 are RES0.

Attributes

TRCIDR5 is a 32-bit register.

Field descriptions

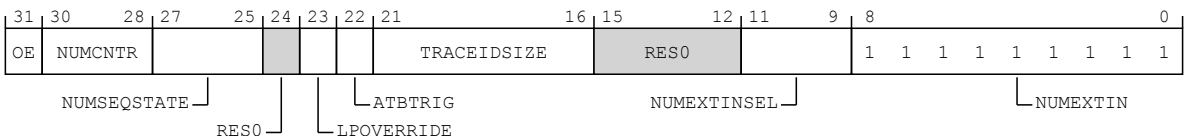

OE, bit [31]

Indicates support for the ETE Trace Output Enable.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OE	Meaning
0b0	ETE Trace Output Enable is not implemented.
0b1	ETE Trace Output Enable is implemented.

When FEAT_ETEv1p3 is implemented and when any IMPLEMENTATION DEFINED trace output interface is implemented, this field is 1.

Access to this field is RO.

NUMCNTR, bits [30:28]

Indicates the number of Counters that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMCNTR	Meaning
0b000..0b100	The number of Counters implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

If TRCIDR4.NUMRSPAIR == `0b0000` then this field is `0b000`.

Access to this field is RO.

NUMSEQSTATE, bits [27:25]

Indicates if the Sequencer is implemented and the number of Sequencer states that are implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMSEQSTATE	Meaning
`0b000`	The Sequencer is not implemented.
`0b100`	Four Sequencer states are implemented.

All other values are reserved.

If TRCIDR4.NUMRSPAIR == `0b0000` then this field is `0b000`.

Access to this field is RO.

Bit [24]

Reserved, RES0.

LPOVERRIDE, bit [23]

Indicates support for Low-power Override Mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LPOVERRIDE	Meaning
`0b0`	The trace unit does not support Low-power Override Mode.
`0b1`	The trace unit supports Low-power Override Mode.

Access to this field is RO.

ATBTRIG, bit [22]

Indicates if the implementation can support ATB triggers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ATBTRIG	Meaning
`0b0`	The implementation does not support ATB triggers.
`0b1`	The implementation supports ATB triggers.

If TRCIDR4.NUMRSPAIR == `0b0000` then this field is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

TRACEIDSIZE, bits [21:16]

Indicates the trace ID width.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRACEIDSIZE	Meaning
`0b000000`	The external trace interface is not implemented.
`0b000111`	The implementation supports a 7-bit trace ID.

All other values are reserved.

Note

AMBA ATB requires a 7-bit trace ID width.

Access to this field is RO.

Bits [15:12]

Reserved, RES0.

NUMEXTINSEL, bits [11:9]

Indicates how many External Input Selector resources are implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMEXTINSEL	Meaning
`0b000..0b100`	The number of External Input Selector resources implemented.

All other values are reserved.

Access to this field is RO.

NUMEXTIN, bits [8:0]

Indicates how many External Inputs are implemented.

NUMEXTIN	Meaning
`0b111111111`	Unified PMU event selection.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCIDR5

TRCIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1F4`	TRCIDR5

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.39 TRCIDR6, Trace ID Register 6

The TRCIDR6 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR6 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR6[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR6 are RES0.

Attributes

TRCIDR6 is a 32-bit register.

Field descriptions

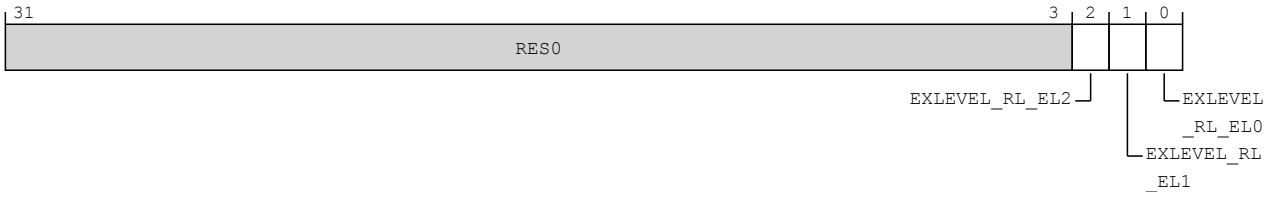

Bits [31:3]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [2]

Indicates if Realm EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_RL_EL2	Meaning
0b0	Realm EL2 is not implemented.
0b1	Realm EL2 is implemented.

Access to this field is RO.

EXLEVEL_RL_EL1, bit [1]

Indicates if Realm EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_RL_EL1	Meaning
0b0	Realm EL1 is not implemented.
0b1	Realm EL1 is implemented.

Access to this field is RO.

EXLEVEL_RL_EL0, bit [0]

Indicates if Realm EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_RL_EL0	Meaning
0b0	Realm EL0 is not implemented.
0b1	Realm EL0 is implemented.

Access to this field is RO.

Accessing TRCIDR6

TRCIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x1F8	TRCIDR6

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.40 TRCIDR7, Trace ID Register 7

The TRCIDR7 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR7 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR7[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR7 are RES0.

Attributes

TRCIDR7 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCIDR7

TRCIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x1FC	TRCIDR7

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.41 TRCIDR8, Trace ID Register 8

The TRCIDR8 characteristics are:

Purpose

Returns the maximum speculation depth of the instruction trace element stream.

Configuration

External register TRCIDR8 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR8[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR8 are RES0.

Attributes

TRCIDR8 is a 32-bit register.

Field descriptions

31	0
MAXSPEC	

MAXSPEC, bits [31:0]

Indicates the maximum speculation depth of the instruction trace element stream. This is the maximum number of P0 elements in the trace element stream that can be speculative at any time.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCIDR8

TRCIDR8 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x180`	TRCIDR8

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.42 TRCIDR9, Trace ID Register 9

The TRCIDR9 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR9 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR9[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR9 are RES0.

Attributes

TRCIDR9 is a 32-bit register.

Field descriptions

31	0
NUMP0KEY	

NUMP0KEY, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of P0 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR9

TRCIDR9 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x184`	TRCIDR9

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.43 TRCIMSPEC0, Trace IMP DEF Register 0

The TRCIMSPEC0 characteristics are:

Purpose

TRCIMSPEC0 shows the presence of any IMPLEMENTATION DEFINED features, and provides an interface to enable the features that are provided.

Configuration

External register TRCIMSPEC0 bits [31:0] are architecturally mapped to AArch64 System register TRCIMSPEC0[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIMSPEC0 are RES0.

Attributes

TRCIMSPEC0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

EN, bits [7:4]

When TRCIMSPEC0.SUPPORT != '0000':

Enable. Controls whether the IMPLEMENTATION DEFINED features are enabled.

EN	Meaning
0b0000	The IMPLEMENTATION DEFINED features are not enabled. The trace unit must behave as if the IMPLEMENTATION DEFINED features are not supported.
0b0001	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0010	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0011	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0100	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0101	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0110	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0111	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1000	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1001	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1010	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1011	The trace unit behavior is IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EN	Meaning
0b1100	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1101	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1110	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1111	The trace unit behavior is IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0000'.

Otherwise:

Reserved, RES0.

SUPPORT, bits [3:0]

Indicates whether the implementation supports IMPLEMENTATION DEFINED features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SUPPORT	Meaning
0b0000	No IMPLEMENTATION DEFINED features are supported.
0b0001	IMPLEMENTATION DEFINED features are supported.
0b0010	IMPLEMENTATION DEFINED features are supported.
0b0011	IMPLEMENTATION DEFINED features are supported.
0b0100	IMPLEMENTATION DEFINED features are supported.
0b0101	IMPLEMENTATION DEFINED features are supported.
0b0110	IMPLEMENTATION DEFINED features are supported.
0b0111	IMPLEMENTATION DEFINED features are supported.
0b1000	IMPLEMENTATION DEFINED features are supported.
0b1001	IMPLEMENTATION DEFINED features are supported.
0b1010	IMPLEMENTATION DEFINED features are supported.
0b1011	IMPLEMENTATION DEFINED features are supported.
0b1100	IMPLEMENTATION DEFINED features are supported.
0b1101	IMPLEMENTATION DEFINED features are supported.
0b1110	IMPLEMENTATION DEFINED features are supported.
0b1111	IMPLEMENTATION DEFINED features are supported.

Use of nonzero values requires written permission from Arm.

Access to this field is RO.

Accessing TRCIMSPEC0

TRCIMSPEC0 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0x1C0`	TRCIMSPEC0

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.44 TRCIMSPEC<n>, Trace IMP DEF Register <n>, n = 1 - 7

The TRCIMSPEC<n> characteristics are:

Purpose

These registers might return information that is specific to an implementation, or enable features specific to an implementation to be programmed. The product Technical Reference Manual describes these registers.

Configuration

External register TRCIMSPEC<n> bits [31:0] are architecturally mapped to AArch64 System register TRCIMSPEC<n>[31:0].

This register is present only when an implementation implements TRCIMSPEC<n>, FEAT_ETE is implemented, and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIMSPEC<n> are RES0.

Attributes

TRCIMSPEC<n> is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

Accessing TRCIMSPEC<n>

TRCIMSPEC<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1C0 + (4 * n)`	TRCIMSPEC<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.45 TRCITCTRL, Trace Integration Mode Control Register

The TRCITCTRL characteristics are:

Purpose

A component can use TRCITCTRL to dynamically switch between functional mode and integration mode. In integration mode, topology detection is enabled. After switching to integration mode and performing integration tests or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCITCTRL are RES0.

Attributes

TRCITCTRL is a 32-bit register.

Field descriptions

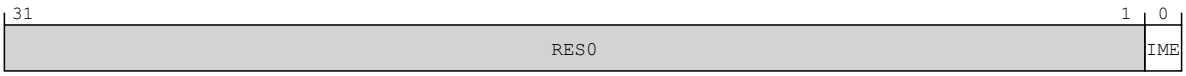

Bits [31:1]

Reserved, RES0.

IME, bit [0]

When topology detection or integration functionality is implemented:

Integration Mode Enable.

IME	Meaning
0b0	Component functional mode.
0b1	Component integration mode. Support for topology detection and integration testing is enabled.

Otherwise:

Reserved, RES0.

Accessing TRCITCTRL

External debugger accesses to this register are IMPLEMENTATION DEFINED when the trace unit is not in the Idle state.

TRCITCTRL can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0xF00`	TRCITCTRL

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.46 TRCITEEDCR, Instrumentation Trace Extension External Debug Control Register

The TRCITEEDCR characteristics are:

Purpose

Controls instrumentation trace filtering.

Configuration

External register TRCITEEDCR bits [31:0] are architecturally mapped to AArch64 System register TRCITEEDCR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and FEAT_ITE is implemented. Otherwise, direct accesses to TRCITEEDCR are RES0.

Attributes

TRCITEEDCR is a 32-bit register.

Field descriptions

31 … 7	6	5	4	3	2	1	0
RES0	RL	S	NS	E3	E2	E1	E0

Bits [31:7]

Reserved, RES0.

RL, bit [6]

When FEAT_RME is implemented:

Instrumentation Trace in Realm state.

RL	Meaning
0b0	Instrumentation trace prohibited in Realm state.
0b1	Instrumentation trace permitted in Realm state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When HaveSecureState():

Instrumentation Trace in Secure state.

S	Meaning
`0b0`	Instrumentation trace prohibited in Secure state.
`0b1`	Instrumentation trace permitted in Secure state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

When FEAT_RME is not implemented, this field is used in conjunction with TRCCONFIGR.ITO, TRCITEEDCR.E3, and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Secure state.

When FEAT_RME is implemented, this field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NS, bit [4]

When EL3 is implemented or NonSecureOnlyImplementation():

Instrumentation Trace in Non-secure state.

NS	Meaning
`0b0`	Instrumentation trace prohibited in Non-secure state.
`0b1`	Instrumentation trace permitted in Non-secure state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3, bit [3]

When EL3 is implemented:

Instrumentation Trace Enable at EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3	Meaning
0b0	Instrumentation trace prohibited at EL3.
0b1	Instrumentation trace permitted at EL3.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

When FEAT_RME is not implemented, TRCITEEDCR.E3 is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.S to control whether Instrumentation trace is permitted or prohibited at EL3.

When FEAT_RME is implemented, TRCITEEDCR.E3 is used in conjunction with TRCCONFIGR.ITO to control whether Instrumentation trace is permitted or prohibited at EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E<m> , bits [m], for m = 2 to 0

Instrumentation Trace Enable at EL<m>.

E<m>	Meaning
0b0	Instrumentation trace prohibited at EL<m>.
0b1	Instrumentation trace permitted at EL<m>.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This bit is used in conjunction with TRCCONFIGR.ITO, TRCITEEDCR.NS, TRCITEEDCR.S, and TRCITEEDCR.RL to control whether Instrumentation trace is permitted or prohibited at EL<m> in the specified Security states.

TRCITEEDCR.E<2> is RES0 if EL2 is not implemented in any Security states.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCITEEDCR

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCITEEDCR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x048	TRCITEEDCR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.47 TRCLAR, Trace Lock Access Register

The TRCLAR characteristics are:

Purpose

Used to lock and unlock the Software Lock.

Note

ETE does not implement the Software Lock.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and ETE Software Lock is implemented. Otherwise, direct accesses to TRCLAR are RES0.

Attributes

TRCLAR is a 32-bit register.

Field descriptions

31	0
KEY	

KEY, bits [31:0]

When ETE Software Lock is implemented:

Software Lock Key.

A value of `0xC5ACCE55` unlocks the Software Lock.

Any other value locks the Software Lock.

Otherwise:

Reserved, RES0.

Accessing TRCLAR

External debugger accesses to this register are unaffected by the OS Lock.

TRCLAR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFB0`	TRCLAR

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.48 TRCLSR, Trace Lock Status Register

The TRCLSR characteristics are:

Purpose

Indicates whether the Software Lock is implemented, and the current status of the Software Lock.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCLSR are RES0.

Attributes

TRCLSR is a 32-bit register.

Field descriptions

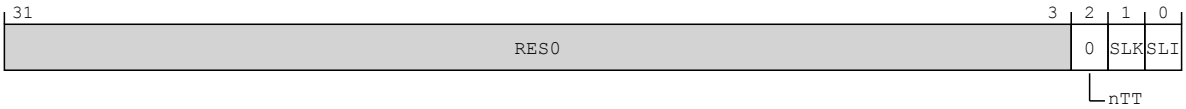

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Software lock size.

Reads as `0b0`

Access to this field is RO.

SLK, bit [1]

The current Software Lock status.

SLK	Meaning
`0b0`	Software Lock is unlocked.
`0b1`	Software Lock is locked. Writes to the other registers in this component, except for the TRCLAR, are ignored.

This field reads as 0.

SLI, bit [0]

Indicates whether the Software Lock is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SLI	Meaning
`0b0`	Software Lock is not implemented. Writes to the TRCLAR are ignored.
`0b1`	Software Lock is implemented.

This field reads as 0.

Access to this field is RO.

Accessing TRCLSR

External debugger accesses to this register are unaffected by the OS Lock.

TRCLSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFB4`	TRCLSR

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.49 TRCOSLSR, Trace OS Lock Status Register

The TRCOSLSR characteristics are:

Purpose

Returns the status of the Trace OS Lock.

Configuration

External register TRCOSLSR bits [31:0] are architecturally mapped to AArch64 System register TRCOSLSR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCOSLSR are RES0.

Attributes

TRCOSLSR is a 32-bit register.

Field descriptions

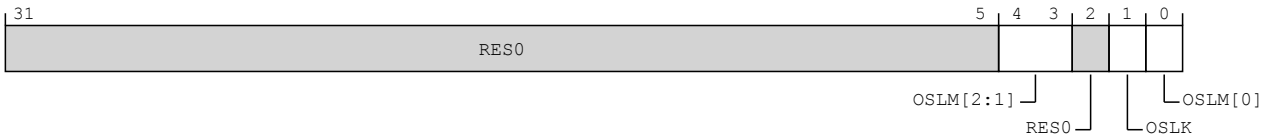

Bits [31:5]

Reserved, RES0.

OSLM, bits [4:3, 0]

OS Lock model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b000	Trace OS Lock is not implemented.
0b010	Trace OS Lock is implemented.
0b100	Trace OS Lock is not implemented, and the trace unit is controlled by the PE OS Lock.

All other values are reserved.

When FEAT_ETE is implemented, the values 0b000 and 0b010 are not permitted.

Access to this field is RO.

Bit [2]

Reserved, RES0.

OSLK, bit [1]

OS Lock status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLK	Meaning
0b0	The OS Lock is unlocked.
0b1	The OS Lock is locked.

When FEAT_ETE is implemented, this field indicates the state of the PE OS Lock.

When FEAT_ETMv4 is implemented, this field indicates the state of the Trace OS Lock.

Accessing TRCOSLSR

External debugger accesses to this register are unaffected by the OS Lock.

TRCOSLSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x304	TRCOSLSR

Accessible as follows:

- When !AllowExternalTraceAccess(addrdesc) or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.50 TRCPDCR, Trace PowerDown Control Register

The TRCPDCR characteristics are:

Purpose

Requests the system to provide power to the trace unit.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPDCR are RES0.

Attributes

TRCPDCR is a 32-bit register.

Field descriptions

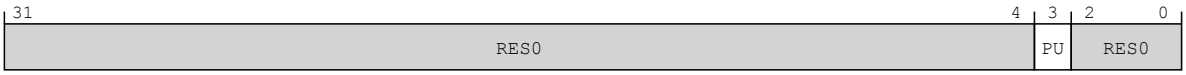

Bits [31:4]

Reserved, RES0.

PU, bit [3]

Power Up Request.

PU	Meaning
0b0	The system can remove power from the trace unit core power domain, or requests for power to the trace unit core power domain are implemented outside of the trace unit.
0b1	The system must provide power to the trace unit core power domain.

This field is RES0.

Bits [2:0]

Reserved, RES0.

Accessing TRCPDCR

External debugger accesses to this register are unaffected by the OS Lock.

TRCPDCR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x310	TRCPDCR

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.51 TRCPDSR, Trace PowerDown Status Register

The TRCPDSR characteristics are:

Purpose

Indicates the power status of the trace unit.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPDSR are RES0.

Attributes

TRCPDSR is a 32-bit register.

Field descriptions

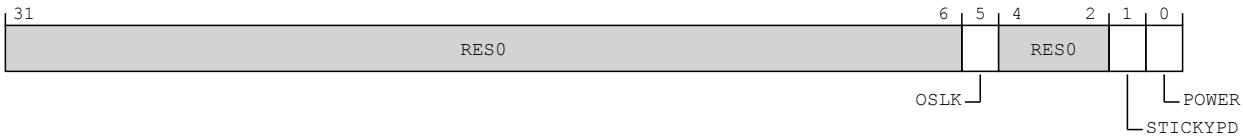

Bits [31:6]

Reserved, RES0.

OSLK, bit [5]

OS Lock Status.

OSLK	Meaning
0b0	The OS Lock is unlocked.
0b1	The OS Lock is locked.

Note

This field indicates the state of the PE OS Lock.

Bits [4:2]

Reserved, RES0.

STICKYPD, bit [1]

Sticky powerdown status. Indicates whether the trace register state is valid.

STICKYPD	Meaning
0b0	The state of TRCOSLSR and the trace registers are valid.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STICKYPD	Meaning
0b1	The state of TRCOSLSR and the trace registers might not be valid.

This field is set to 1 if the power to the trace unit core power domain is removed and the trace unit register state is not valid.

The STICKYPD field is read-sensitive. On a read of the TRCPDSR, this field is cleared to 0 after the register has been read.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RC/WI.

POWER, bit [0]

Power Status.

POWER	Meaning
0b0	The trace unit core power domain is not powered. All trace unit registers are not accessible and they all return an error response.
0b1	The trace unit core power domain is powered. Trace unit registers are accessible.

Access to this field is RAO/WI.

Accessing TRCPDSR

External debugger accesses to this register are unaffected by the OS Lock.

TRCPDSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x314	TRCPDSR

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.52 TRCPIDR0, Trace Peripheral Identification Register 0

The TRCPIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR0 are RES0.

Attributes

TRCPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - TRCPIDR1.PART_1 contains part number bits [11:8].
  - TRCPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - TRCPIDR1.PART_1 contains part number bits [15:12].
  - TRCPIDR0.PART_0 contains part number bits [11:4].
  - TRCPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, TRCPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCPIDR0

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFE0`	TRCPIDR0

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.53 TRCPIDR1, Trace Peripheral Identification Register 1

The TRCPIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR1 are RES0.

Attributes

TRCPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- TRCPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRCPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRCPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - TRCPIDR1.PART_1 contains part number bits [11:8].
  - TRCPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - TRCPIDR1.PART_1 contains part number bits [15:12].
  - TRCPIDR0.PART_0 contains part number bits [11:4].
  - TRCPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, TRCPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCPIDR1

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFE4`	TRCPIDR1

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.54 TRCPIDR2, Trace Peripheral Identification Register 2

The TRCPIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR2 are RES0.

Attributes

TRCPIDR2 is a 32-bit register.

Field descriptions

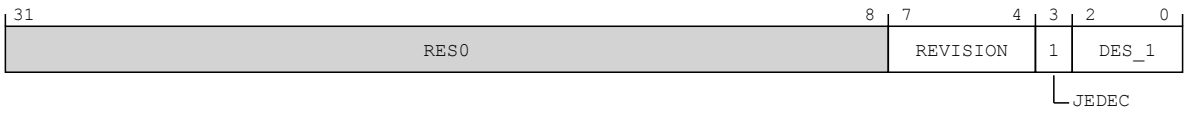

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- TRCPIDR2.REVISION indicates the 4-bit revision number.
- TRCPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- TRCPIDR2.REVISION indicates the 4-bit major revision number.
- TRCPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- TRCPIDR2.REVISION contains part number bits [3:0].
- TRCPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- TRCPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRCPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRCPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing TRCPIDR2

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFE8`	TRCPIDR2

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.55 TRCPIDR3, Trace Peripheral Identification Register 3

The TRCPIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR3 are RES0.

Attributes

TRCPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- TRCPIDR2.REVISION indicates the 4-bit revision number.
- TRCPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- TRCPIDR2.REVISION indicates the 4-bit major revision number.
- TRCPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- TRCPIDR2.REVISION contains part number bits [3:0].
- TRCPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

TRCPIDR3.CMOD might also indicate component modifications.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

TRCPIDR3.REVAND might also indicate component modifications.

Access to this field is RO.

Accessing TRCPIDR3

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFEC`	TRCPIDR3

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.56 TRCPIDR4, Trace Peripheral Identification Register 4

The TRCPIDR4 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR4 are RES0.

Attributes

TRCPIDR4 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	SIZE	DES_2

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIZE	Meaning
0b0000	One of the following is true: • The component uses a single 4KB block. • The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.
0b0001..0b1111	The component occupies $2^{\text{TRCPIDR4.SIZE}}$ 4KB blocks.

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

This field has the value 0b0000.

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRCPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRCPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing TRCPIDR4

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFD0`	TRCPIDR4

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.57 TRCPIDR5, Trace Peripheral Identification Register 5

The TRCPIDR5 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR5 are RES0.

Attributes

TRCPIDR5 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCPIDR5

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFD4`	TRCPIDR5

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.58 TRCPIDR6, Trace Peripheral Identification Register 6

The TRCPIDR6 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR6 are RES0.

Attributes

TRCPIDR6 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCPIDR6

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFD8	TRCPIDR6

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.59 TRCPIDR7, Trace Peripheral Identification Register 7

The TRCPIDR7 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR7 are RES0.

Attributes

TRCPIDR7 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCPIDR7

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFDC	TRCPIDR7

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.60 TRCPRGCTLR, Trace Programming Control Register

The TRCPRGCTLR characteristics are:

Purpose

Enables the trace unit.

Configuration

External register TRCPRGCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCPRGCTLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPRGCTLR are RES0.

Attributes

TRCPRGCTLR is a 32-bit register.

Field descriptions

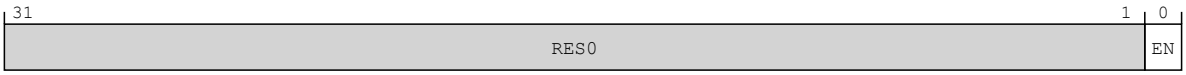

Bits [31:1]

Reserved, RES0.

EN, bit [0]

Trace unit enable.

EN	Meaning
0b0	The trace unit is disabled.
0b1	The trace unit is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Accessing TRCPRGCTLR

Must be programmed.

TRCPRGCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x004	TRCPRGCTLR

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.61 TRCQCTLR, Trace Q Element Control Register

The TRCQCTLR characteristics are:

Purpose

Controls when Q elements are enabled.

Configuration

External register TRCQCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCQCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR0.QFILT == '1'. Otherwise, direct accesses to TRCQCTLR are RES0.

Attributes

TRCQCTLR is a 32-bit register.

Field descriptions

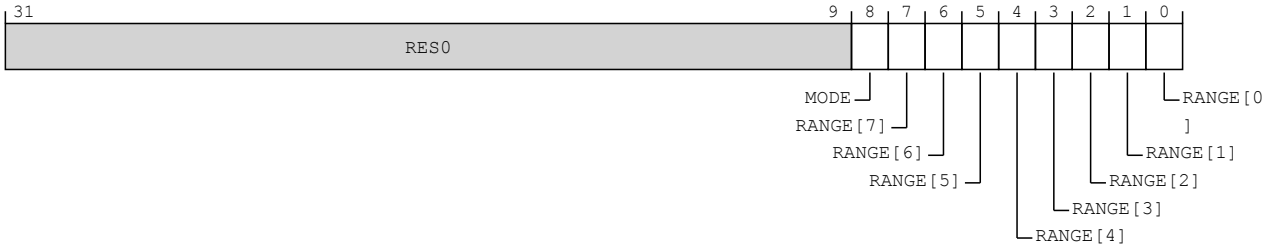

Bits [31:9]

Reserved, RES0.

MODE, bit [8]

Selects whether the Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit is permitted to generate Q elements or address ranges where the trace unit is not permitted to generate Q elements:

MODE	Meaning
0b0	Exclude mode. The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit must not generate Q elements. If no ranges are selected, Q elements are permitted across the entire memory map.
0b1	Include Mode. The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit can generate Q elements. If all the implemented bits in RANGE are set to 0 then Q elements are disabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RANGE[<m>] , bits [m], for m = 7 to 0

Specifies whether Address Range Comparator <m> controls Q elements.

RANGE[<m>]	Meaning
`0b0`	The address range that Address Range Comparator <m> defines is not selected.
`0b1`	The address range that Address Range Comparator <m> defines is selected.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCQCTLR

Must be programmed if TRCCONFIGR.QE != `0b00`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCQCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x044`	TRCQCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.62 TRCRSCTLR<n>, Trace Resource Selection Control Register <n>, n = 2 - 31

The TRCRSCTLR<n> characteristics are:

Purpose

Controls the selection of the resources in the trace unit.

Configuration

Resource selector 0 always returns FALSE.

Resource selector 1 always returns TRUE.

Resource selectors are implemented in pairs. Each odd numbered resource selector is part of a pair with the even numbered resource selector that is numbered as one less than it. For example, resource selectors 2 and 3 form a pair.

External register TRCRSCTLR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCRSCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and ((UInt(TRCIDR4.NUMRSPAIR) + 1) * 2) > n. Otherwise, direct accesses to TRCRSCTLR<n> are RES0.

Attributes

TRCRSCTLR<n> is a 32-bit register.

Field descriptions

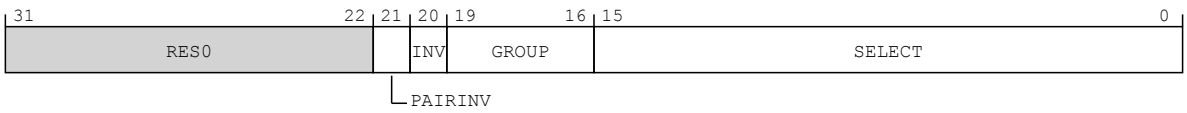

Bits [31:22]

Reserved, RES0.

PAIRINV, bit [21]

When (n MOD 2) == 0:

Controls whether the combined result from a resource selector pair is inverted.

PAIRINV	Meaning
0b0	Do not invert the combined output of the 2 resource selectors.
0b1	Invert the combined output of the 2 resource selectors.

If:

- A is the register TRCRSCTLR<n>.
- B is the register TRCRSCTLR<n+1>.

Then the combined output of the 2 resource selectors A and B depends on the value of (A.PAIRINV, A.INV, B.INV) as follows:

- 0b000 -> A and B.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `0b001` -> Reserved.
- `0b010` -> not(A) and B.
- `0b011` -> not(A) and not(B).
- `0b100` -> not(A) or not(B).
- `0b101` -> not(A) or B.
- `0b110` -> Reserved.
- `0b111` -> A or B.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

INV, bit [20]

Controls whether the resource, that TRCRSCTLR<n>.GROUP and TRCRSCTLR<n>.SELECT selects, is inverted.

INV	Meaning
`0b0`	Do not invert the output of this selector.
`0b1`	Invert the output of this selector.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

GROUP, bits [19:16]

Selects a group of resources.

GROUP	Meaning	SELECT
`0b0000`	External Input Selectors.	SELECT encoding for External Input Selectors
`0b0001`	PE Comparator Inputs.	SELECT encoding for PE Comparator Inputs
`0b0010`	Counters and Sequencer.	SELECT encoding for Counters and Sequencer
`0b0011`	Single-shot Comparator Controls.	SELECT encoding for Single-shot Comparator Controls
`0b0100`	Single Address Comparators.	SELECT encoding for Single Address Comparators
`0b0101`	Address Range Comparators.	SELECT encoding for Address Range Comparators
`0b0110`	Context Identifier Comparators.	SELECT encoding for Context Identifier Comparators
`0b0111`	Virtual Context Identifier Comparators.	SELECT encoding for Virtual Context Identifier Comparators

All other values are reserved.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT, bits [15:0]

Resource Specific Controls. Contains the controls specific to the resource group selected by GROUP, described in the following sections.

SELECT encoding for External Input Selectors

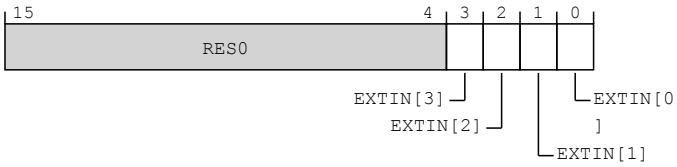

Bits [15:4]

Reserved, RES0.

EXTIN[<m>] , bits [m], for m = 3 to 0

Selects one or more External Inputs.

EXTIN[<m>]	Meaning
0b0	Ignore EXTIN <m>.
0b1	Select EXTIN <m>.

This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for PE Comparator Inputs

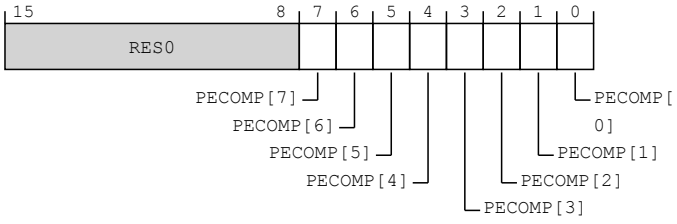

Bits [15:8]

Reserved, RES0.

PECOMP[<m>] , bits [m], for m = 7 to 0

Selects one or more PE Comparator Inputs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PECOMP[<m>]	Meaning
0b0	Ignore PE Comparator Input <m>.
0b1	Select PE Comparator Input <m>.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Counters and Sequencer

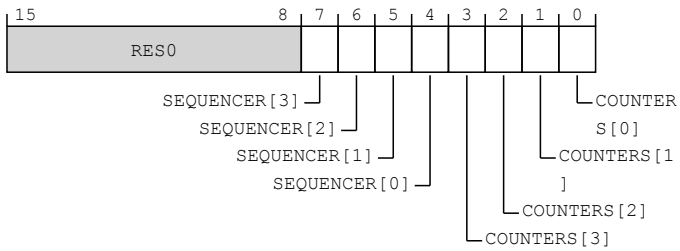

Bits [15:8]

Reserved, RES0.

SEQUENCER[<m>] , bits [m+4], for m = 3 to 0

Sequencer states.

SEQUENCER[<m>]	Meaning
0b0	Ignore Sequencer state <m>.
0b1	Select Sequencer state <m>.

This bit is RES0 if m >= TRCIDR5.NUMSEQSTATE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COUNTERS[<m>] , bits [m], for m = 3 to 0

Counters resources at zero.

COUNTERS[<m>]	Meaning
0b0	Ignore Counter <m>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COUNTERS[<m>]	Meaning
0b1	Select Counter <m> is zero.

This bit is RES0 if m >= TRCIDR5.NUMCNTR.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Single-shot Comparator Controls

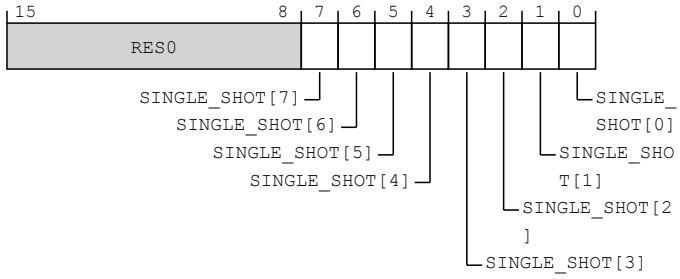

Bits [15:8]

Reserved, RES0.

SINGLE_SHOT[<m>] , bits [m], for m = 7 to 0

Selects one or more Single-shot Comparator Controls.

SINGLE_SHOT[<m>]	Meaning
0b0	Ignore Single-shot Comparator Control <m>.
0b1	Select Single-shot Comparator Control <m>.

This bit is RES0 if m >= TRCIDR4.NUMSSCC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Single Address Comparators

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

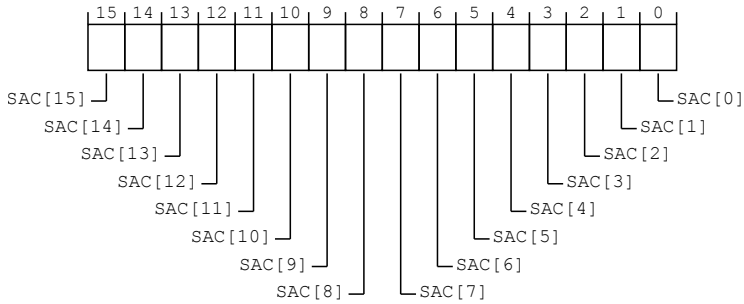

SAC[<m>] , bits [m], for m = 15 to 0

Selects one or more Single Address Comparators.

SAC[<m>]	Meaning
0b0	Ignore Single Address Comparator <m>.
0b1	Select Single Address Comparator <m>.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Address Range Comparators

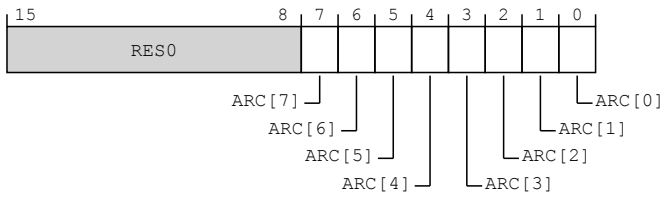

Bits [15:8]

Reserved, RES0.

ARC[<m>] , bits [m], for m = 7 to 0

Selects one or more Address Range Comparators.

ARC[<m>]	Meaning
0b0	Ignore Address Range Comparator <m>.
0b1	Select Address Range Comparator <m>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Context Identifier Comparators

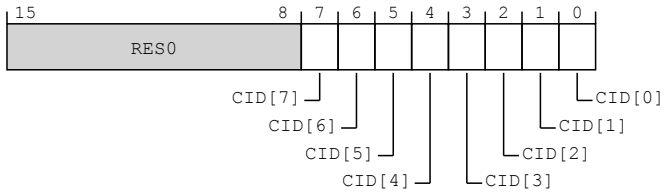

Bits [15:8]

Reserved, RES0.

CID[<m>] , bits [m], for m = 7 to 0

Selects one or more Context Identifier Comparators.

CID[<m>]	Meaning
0b0	Ignore Context Identifier Comparator <m>.
0b1	Select Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMCIDC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Virtual Context Identifier Comparators

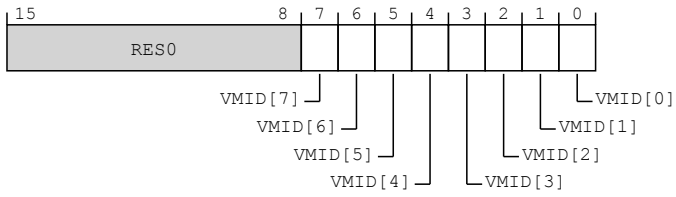

Bits [15:8]

Reserved, RES0.

VMID[<m>] , bits [m], for m = 7 to 0

Selects one or more Virtual Context Identifier Comparators.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID[<m>]	Meaning
0b0	Ignore Virtual Context Identifier Comparator <m>.
0b1	Select Virtual Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMVMIDC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCRSCTLR<n>

Must be programmed if any of the following are true:

- TRCCNTCTLR<a>.RLDEVENT_TYPE == 0 and TRCCNTCTLR<a>.RLDEVENT_SEL == n.
- TRCCNTCTLR<a>.RLDEVENT_TYPE == 1 and TRCCNTCTLR<a>.RLDEVENT_SEL == n/2.
- TRCCNTCTLR<a>.CNTEVENT_TYPE == 0 and TRCCNTCTLR<a>.CNTEVENT_SEL == n.
- TRCCNTCTLR<a>.CNTEVENT_TYPE == 1 and TRCCNTCTLR<a>.CNTEVENT_SEL == n/2.
- TRCEVENTCTL0R.EVENT0_TYPE == 0 and TRCEVENTCTL0R.EVENT0_SEL == n.
- TRCEVENTCTL0R.EVENT0_TYPE == 1 and TRCEVENTCTL0R.EVENT0_SEL == n/2.
- TRCEVENTCTL0R.EVENT1_TYPE == 0 and TRCEVENTCTL0R.EVENT1_SEL == n.
- TRCEVENTCTL0R.EVENT1_TYPE == 1 and TRCEVENTCTL0R.EVENT1_SEL == n/2.
- TRCEVENTCTL0R.EVENT2_TYPE == 0 and TRCEVENTCTL0R.EVENT2_SEL == n.
- TRCEVENTCTL0R.EVENT2_TYPE == 1 and TRCEVENTCTL0R.EVENT2_SEL == n/2.
- TRCEVENTCTL0R.EVENT3_TYPE == 0 and TRCEVENTCTL0R.EVENT3_SEL == n.
- TRCEVENTCTL0R.EVENT3_TYPE == 1 and TRCEVENTCTL0R.EVENT3_SEL == n/2.
- TRCSEQEVR<a>.B_TYPE == 0 and TRCSEQEVR<a>.B_SEL = n.
- TRCSEQEVR<a>.B_TYPE == 1 and TRCSEQEVR<a>.B_SEL = n/2.
- TRCSEQEVR<a>.F_TYPE == 0 and TRCSEQEVR<a>.F_SEL = n.
- TRCSEQEVR<a>.F_TYPE == 1 and TRCSEQEVR<a>.F_SEL = n/2.
- TRCSEQRSTEVR.RST_TYPE == 0 and TRCSEQRSTEVR.RST_SEL == n.
- TRCSEQRSTEVR.RST_TYPE == 1 and TRCSEQRSTEVR.RST_SEL == n/2.
- TRCTSCTLR.EVENT_TYPE == 0 and TRCTSCTLR.EVENT_SEL == n.
- TRCTSCTLR.EVENT_TYPE == 1 and TRCTSCTLR.EVENT_SEL == n/2.
- TRCVICTLR.EVENT_TYPE == 0 and TRCVICTLR.EVENT_SEL == n.
- TRCVICTLR.EVENT_TYPE == 1 and TRCVICTLR.EVENT_SEL == n/2.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCRSCTLR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x200 + (4 * n)	TRCRSCTLR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.63 TRCRSR, Trace Resources Status Register

The TRCRSR characteristics are:

Purpose

Use this to set, or read, the status of the resources.

Configuration

External register TRCRSR bits [31:0] are architecturally mapped to AArch64 System register TRCRSR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCRSR are RES0.

Attributes

TRCRSR is a 32-bit register.

Field descriptions

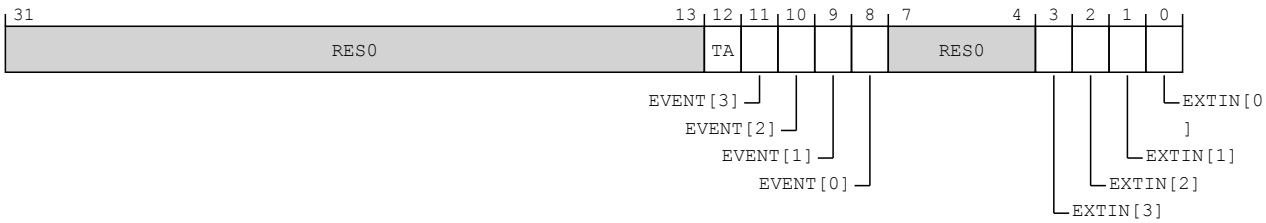

Bits [31:13]

Reserved, RES0.

TA, bit [12]

Tracing active.

TA	Meaning
0b0	Tracing is not active.
0b1	Tracing is active.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

EVENT[<m>] , bits [m+8], for m = 3 to 0

Untraced status of ETEEvents.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVENT[<m>]	Meaning
0b0	An ETEEvent <m> has not occurred.
0b1	An ETEEvent <m> has occurred while the resources were in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When TRCIDR4.NUMRSPAIR == '0000', access to this field is RES0.
- Access to this field is RES0 if all of the following are true:
  - TRCIDR4.NUMRSPAIR != '0000'
  - m > UInt(TRCIDR0.NUMEVENT)
- Otherwise, access to this field is RW.

Bits [7:4]

Reserved, RES0.

EXTIN[<m>] , bits [m], for m = 3 to 0

The sticky status of the External Input Selectors.

EXTIN[<m>]	Meaning
0b0	An event selected by External Input Selector <m> has not occurred.
0b1	At least one event selected by External Input Selector <m> has occurred while the resources were in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR5.NUMEXTINSEL), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCRSR

Must always be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCRSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x028	TRCRSR

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.64 TRCSEQEVR<n>, Trace Sequencer State Transition Control Register <n>, n = 0 - 2

The TRCSEQEVR<n> characteristics are:

Purpose

Moves the Sequencer state:

- Backwards, from state n+1 to state n when a programmed resource event occurs.
- Forwards, from state n to state n+1 when a programmed resource event occurs.

Configuration

External register TRCSEQEVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSEQEVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR5.NUMSEQSTATE != '000'. Otherwise, direct accesses to TRCSEQEVR<n> are RES0.

Attributes

TRCSEQEVR<n> is a 32-bit register.

Field descriptions

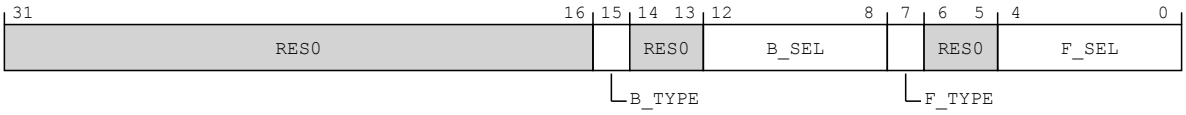

Bits [31:16]

Reserved, RES0.

B_TYPE, bit [15]

Backward field. Selects an event that causes the Sequencer to move from state n+1 to state n. For example, if TRCSEQEVR2.B_SEL == `0x14`, then when event `0x14` occurs, the Sequencer moves from state 3 to state 2.

Chooses the type of Resource Selector.

B_TYPE	Meaning
`0b0`	A single Resource Selector. TRCSEQEVR<n>.B.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCSEQEVR<n>.B.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR<n>.B.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B_SEL, bits [12:8]

Backward field. Selects an event that causes the Sequencer to move from state n+1 to state n. For example, if TRCSEQEVR2.B_SEL == `0x14`, then when event `0x14` occurs, the Sequencer moves from state 3 to state 2.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.B.TYPE controls whether TRCSEQEVR<n>.B.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

F_TYPE, bit [7]

Forward field. Selects an event that causes the Sequencer to move from state n to state n+1. For example, if TRCSEQEVR1.F_SEL == `0x12`, then when event `0x12` occurs, the Sequencer moves from state 1 to state 2.

Chooses the type of Resource Selector.

F_TYPE	Meaning
`0b0`	A single Resource Selector. TRCSEQEVR<n>.F.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCSEQEVR<n>.F.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR<n>.F.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

F_SEL, bits [4:0]

Forward field. Selects an event that causes the Sequencer to move from state n to state n+1. For example, if TRCSEQEVR1.F_SEL == `0x12`, then when event `0x12` occurs, the Sequencer moves from state 1 to state 2.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.F.TYPE controls whether TRCSEQEVR<n>.F.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQEVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.SEQUENCER != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSEQEVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x100 + (4 * n)`	TRCSEQEVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.65 TRCSEQRSTEVR, Trace Sequencer Reset Control Register

The TRCSEQRSTEVR characteristics are:

Purpose

Moves the Sequencer to state 0 when a programmed resource event occurs.

Configuration

External register TRCSEQRSTEVR bits [31:0] are architecturally mapped to AArch64 System register TRCSEQRSTEVR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR5.NUMSEQSTATE != '000'. Otherwise, direct accesses to TRCSEQRSTEVR are RES0.

Attributes

TRCSEQRSTEVR is a 32-bit register.

Field descriptions

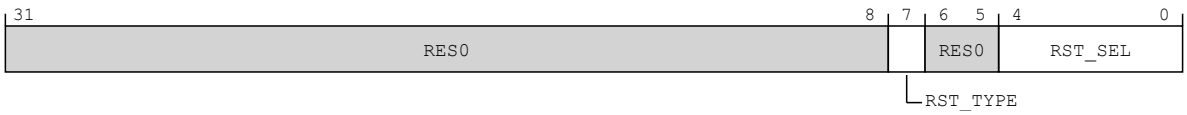

Bits [31:8]

Reserved, RES0.

RST_TYPE, bit [7]

Reset field. Selects an event that causes the Sequencer to move to state 0.

Chooses the type of Resource Selector.

RST_TYPE	Meaning
0b0	A single Resource Selector. TRCSEQRSTEVR.RST.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCSEQRSTEVR.RST.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQRSTEVR.RST.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

RST_SEL, bits [4:0]

Reset field. Selects an event that causes the Sequencer to move to state 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQRSTEVR.RST.TYPE controls whether TRCSEQRSTEVR.RST.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQRSTEVR

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.SEQUENCER != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSEQRSTEVR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x118`	TRCSEQRSTEVR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.66 TRCSEQSTR, Trace Sequencer State Register

The TRCSEQSTR characteristics are:

Purpose

Use this to set, or read, the Sequencer state.

Configuration

External register TRCSEQSTR bits [31:0] are architecturally mapped to AArch64 System register TRCSEQSTR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR5.NUMSEQSTATE != '000'. Otherwise, direct accesses to TRCSEQSTR are RES0.

Attributes

TRCSEQSTR is a 32-bit register.

Field descriptions

31 … 2	1 0
RES0	STATE

Bits [31:2]

Reserved, RES0.

STATE, bits [1:0]

Set or returns the state of the Sequencer.

STATE	Meaning
0b00	State 0.
0b01	State 1.
0b10	State 2.
0b11	State 3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQSTR

Must be programmed if TRCRSCTLR<a>.GROUP == 0b0010 and TRCRSCTLR<a>.SEQUENCER != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSEQSTR can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0x11C`	TRCSEQSTR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.67 TRCSSCCR<n>, Trace Single-shot Comparator Control Register <n>, n = 0 - 7

The TRCSSCCR<n> characteristics are:

Purpose

Controls the corresponding Single-shot Comparator Control resource.

Configuration

External register TRCSSCCR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSCCR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCCR<n> are RES0.

Attributes

TRCSSCCR<n> is a 32-bit register.

Field descriptions

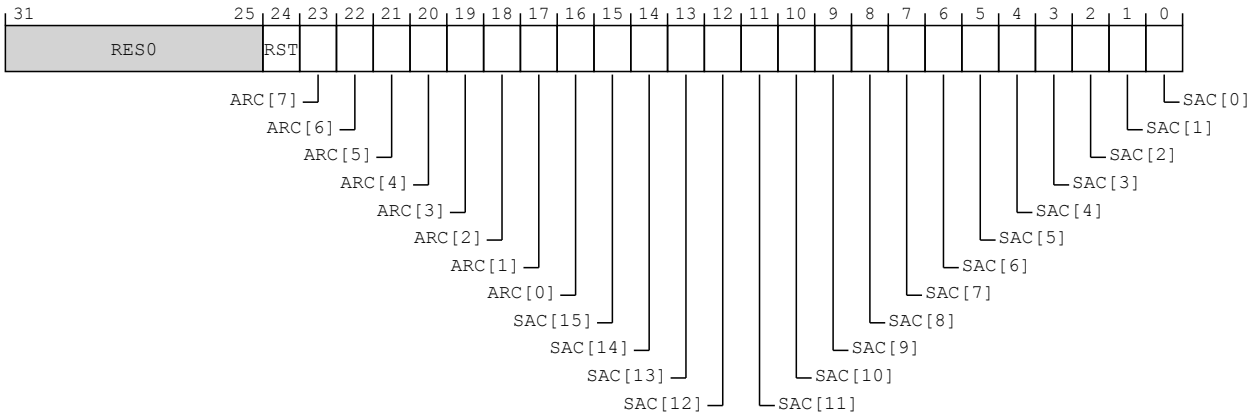

Bits [31:25]

Reserved, RES0.

RST, bit [24]

Selects the Single-shot Comparator Control mode.

RST	Meaning
0b0	The Single-shot Comparator Control is in single-shot mode.
0b1	The Single-shot Comparator Control is in multi-shot mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

ARC[<m>] , bits [m+16], for m = 7 to 0

Selects one or more Address Range Comparators for Single-shot control.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARC[<m>]	Meaning
0b0	The Address Range Comparator <m>, is not selected for Single-shot control.
0b1	The Address Range Comparator <m>, is selected for Single-shot control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

SAC[<m>] , bits [m], for m = 15 to 0

Selects one or more Single Address Comparators for Single-shot control.

SAC[<m>]	Meaning
0b0	The Single Address Comparator <m>, is not selected for Single-shot control.
0b1	The Single Address Comparator <m>, is selected for Single-shot control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCSSCCR<n>

Must be programmed if any TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSSCCR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x280 + (4 * n)	TRCSSCCR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.68 TRCSSCSR<n>, Trace Single-shot Comparator Control Status Register <n>, n = 0 - 7

The TRCSSCSR<n> characteristics are:

Purpose

Returns the status of the corresponding Single-shot Comparator Control.

Configuration

External register TRCSSCSR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSCSR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCSR<n> are RES0.

Attributes

TRCSSCSR<n> is a 32-bit register.

Field descriptions

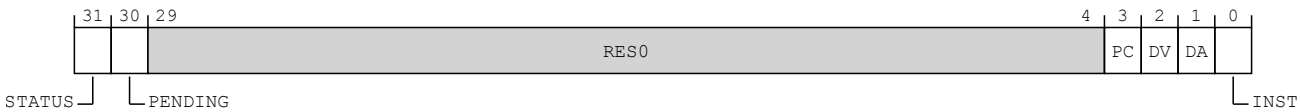

STATUS, bit [31]

Single-shot Comparator Control status. Indicates if any of the comparators selected by this Single-shot Comparator control have matched. The selected comparators are defined by TRCSSCCR<n>.ARC, TRCSSCCR<n>.SAC, and TRCSSPCICR<n>.PC.

STATUS	Meaning
0b0	No match has occurred. When the first match occurs, this field takes a value of 1. It remains at 1 until explicitly modified by a write to this register.
0b1	One or more matches has occurred. If TRCSSCCR<n>.RST == 0 then: • There is only one match and no more matches are possible. • Software must reset this field to 0 to re-enable the Single-shot Comparator Control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

PENDING, bit [30]

Single-shot pending status. The Single-shot Comparator Control fired while the resources were in the Paused state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PENDING	Meaning
0b0	No match has occurred.
0b1	One or more matches has occurred.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [29:4]

Reserved, RES0.

PC, bit [3]

PE Comparator Input support. Indicates if the Single-shot Comparator Control supports PE Comparator Inputs.

PC	Meaning
0b0	This Single-shot Comparator Control does not support PE Comparator Inputs. Selecting any PE Comparator Inputs using the associated TRCSSPCICR<n> results in CONSTRAINED UNPREDICTABLE behavior of the Single-shot Comparator Control resource. The Single-shot Comparator Control might match unexpectedly or might not match.
0b1	This Single-shot Comparator Control supports PE Comparator Inputs.

Access to this field is RO.

DV, bit [2]

Data value comparator support. Data value comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

DV	Meaning
0b0	This Single-shot Comparator Control does not support data value comparisons.
0b1	This Single-shot Comparator Control supports data value comparisons.

This field reads as 0.

Access to this field is RO.

DA, bit [1]

Data Address Comparator support. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

DA	Meaning
0b0	This Single-shot Comparator Control does not support data address comparisons.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	This Single-shot Comparator Control supports data address comparisons.

This field reads as 0.

Access to this field is RO.

INST, bit [0]

Instruction Address Comparator support. Indicates if the Single-shot Comparator Control supports instruction address comparisons.

INST	Meaning
0b0	This Single-shot Comparator Control does not support instruction address comparisons.
0b1	This Single-shot Comparator Control supports instruction address comparisons.

This field reads as 1.

Access to this field is RO.

Accessing TRCSSCSR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSSCSR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x2A0 + (4 * n)	TRCSSCSR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.69 TRCSSPCICR<n>, Trace Single-shot Processing Element Comparator Input Control Register <n>, n = 0 - 7

The TRCSSPCICR<n> characteristics are:

Purpose

Returns the status of the corresponding Single-shot Comparator Control.

Configuration

External register TRCSSPCICR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSPCICR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMSSCC) > n, UInt(TRCIDR4.NUMPC) > 0, and TRCSSCSR<n>.PC == '1'. Otherwise, direct accesses to TRCSSPCICR<n> are RES0.

Attributes

TRCSSPCICR<n> is a 32-bit register.

Field descriptions

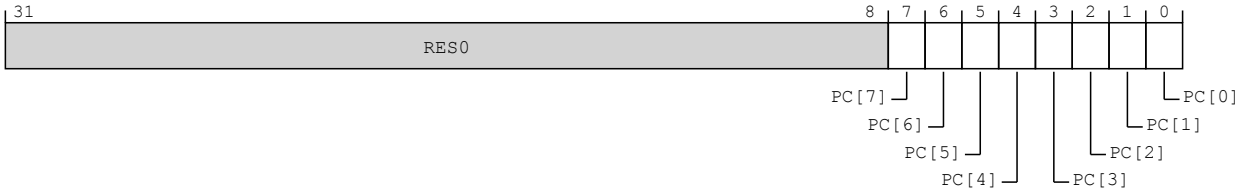

Bits [31:8]

Reserved, RES0.

PC[<m>] , bits [m], for m = 7 to 0

Selects one or more PE Comparator Inputs for Single-shot control.

PC[<m>]	Meaning
0b0	The single PE Comparator Input <m>, is not selected as for Single-shot control.
0b1	The single PE Comparator Input <m>, is selected as for Single-shot control.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSSPCICR<n>

Must be programmed if implemented and any TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSSPCICR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x2C0 + (4 * n)`	TRCSSPCICR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.70 TRCSTALLCTLR, Trace Stall Control Register

The TRCSTALLCTLR characteristics are:

Purpose

Enables trace unit functionality that prevents trace unit buffer overflows.

Configuration

External register TRCSTALLCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCSTALLCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR3.STALLCTL == '1'. Otherwise, direct accesses to TRCSTALLCTLR are RES0.

Attributes

TRCSTALLCTLR is a 32-bit register.

Field descriptions

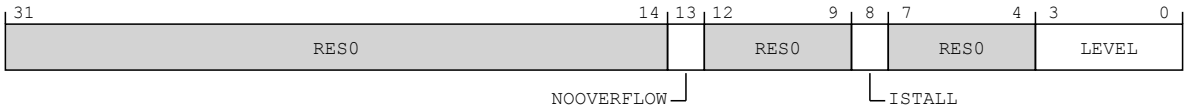

Bits [31:14]

Reserved, RES0.

NOOVERFLOW, bit [13]

When TRCIDR3.NOOVERFLOW == '1':

Trace overflow prevention.

NOOVERFLOW	Meaning
0b0	Trace unit buffer overflow prevention is disabled.
0b1	Trace unit buffer overflow prevention is enabled.

Note

Enabling this feature might cause a significant performance impact.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISTALL, bit [8]

Instruction stall control. Controls if a trace unit can stall the PE when the trace buffer space is less than LEVEL.

ISTALL	Meaning
`0b0`	The trace unit must not stall the PE.
`0b1`	The trace unit can stall the PE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

LEVEL, bits [3:0]

Threshold level field. The field can support 16 monotonic levels from `0b0000` to `0b1111`.

The value `0b0000` defines the Minimal invasion level. This setting has a greater risk of a trace unit buffer overflow.

The value `0b1111` defines the Maximum invasion level. This setting has a reduced risk of a trace unit buffer overflow.

Note

For some implementations, invasion might occur at the minimal invasion level.

One or more of the least significant bits of LEVEL are permitted to be RES0. Arm recommends that LEVEL[3:2] are fully implemented. Arm strongly recommends that LEVEL[3] is always implemented. If one or more bits are RES0 and are written with a nonzero value, the effective value of LEVEL is rounded down to the nearest power of 2 value which has the RES0 bits as zero. For example, if LEVEL[1:0] are RES0 and a value of `0b1110` is written to LEVEL, the effective value of LEVEL is `0b1100`.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSTALLCTLR

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSTALLCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x02C`	TRCSTALLCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.71 TRCSTATR, Trace Status Register

The TRCSTATR characteristics are:

Purpose

Returns the trace unit status.

Configuration

External register TRCSTATR bits [31:0] are architecturally mapped to AArch64 System register TRCSTATR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCSTATR are RES0.

Attributes

TRCSTATR is a 32-bit register.

Field descriptions

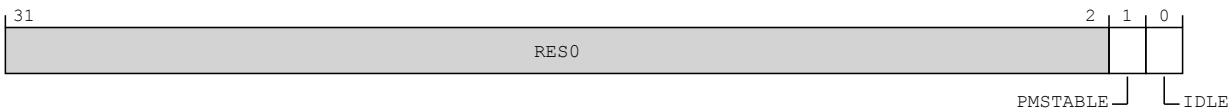

Bits [31:2]

Reserved, RES0.

PMSTABLE, bit [1]

Programmers' model stable.

PMSTABLE	Meaning
0b0	The programmers' model is not stable.
0b1	The programmers' model is stable.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - TRCPRGCTLR.EN == '1'
  - !OSLockStatus()
- Otherwise, access to this field is RO.

IDLE, bit [0]

Idle status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDLE	Meaning
`0b0`	The trace unit is not idle.
`0b1`	The trace unit is idle.

Accessing TRCSTATR

TRCSTATR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x00C`	TRCSTATR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.72 TRCSYNCPR, Trace Synchronization Period Register

The TRCSYNCPR characteristics are:

Purpose

Controls how often trace protocol synchronization requests occur.

Configuration

External register TRCSYNCPR bits [31:0] are architecturally mapped to AArch64 System register TRCSYNCPR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCSYNCPR are RES0.

Attributes

TRCSYNCPR is a 32-bit register.

Field descriptions

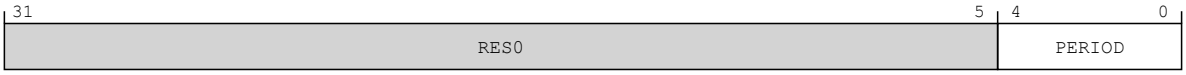

Bits [31:5]

Reserved, RES0.

PERIOD, bits [4:0]

Defines the number of bytes of trace between each periodic trace protocol synchronization request.

PERIOD	Meaning
0b00000	Trace protocol synchronization is disabled.
0b01000	Trace protocol synchronization request occurs after 2^8 bytes of trace.
0b01001	Trace protocol synchronization request occurs after 2^9 bytes of trace.
0b01010	Trace protocol synchronization request occurs after 2^{10} bytes of trace.
0b01011	Trace protocol synchronization request occurs after 2^{11} bytes of trace.
0b01100	Trace protocol synchronization request occurs after 2^{12} bytes of trace.
0b01101	Trace protocol synchronization request occurs after 2^{13} bytes of trace.
0b01110	Trace protocol synchronization request occurs after 2^{14} bytes of trace.
0b01111	Trace protocol synchronization request occurs after 2^{15} bytes of trace.
0b10000	Trace protocol synchronization request occurs after 2^{16} bytes of trace.
0b10001	Trace protocol synchronization request occurs after 2^{17} bytes of trace.
0b10010	Trace protocol synchronization request occurs after 2^{18} bytes of trace.
0b10011	Trace protocol synchronization request occurs after 2^{19} bytes of trace.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b10100	Trace protocol synchronization request occurs after 2^{20} bytes of trace.

Other values are reserved. If a reserved value is programmed into PERIOD, then the behavior of the synchronization period counter is CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:

- No trace protocol synchronization requests are generated by this counter.
- Trace protocol synchronization requests occur at the specified period.
- Trace protocol synchronization requests occur at some other UNKNOWN period which can vary.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSYNCPR

Must be programmed if TRCIDR3.SYNCPR == 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSYNCPR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x034	TRCSYNCPR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.73 TRCTRACEIDR, Trace ID Register

The TRCTRACEIDR characteristics are:

Purpose

Sets the trace ID for instruction trace.

Configuration

External register TRCTRACEIDR bits [31:0] are architecturally mapped to AArch64 System register TRCTRACEIDR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCTRACEIDR are RES0.

Attributes

TRCTRACEIDR is a 32-bit register.

Field descriptions

31 … 7	6 … 0
RES0	TRACEID

Bits [31:7]

Reserved, RES0.

TRACEID, bits [6:0]

Trace ID field. Sets the trace ID value for instruction trace. The width of the field is indicated by the value of TRCIDR5.TRACEIDSIZE. Unimplemented bits are RES0.

If an implementation supports AMBA ATB, then:

- The width of the field is 7 bits.
- Writing a reserved trace ID value does not affect behavior of the trace unit but it might cause UNPREDICTABLE behavior of the trace capture infrastructure.

See the AMBA ATB Protocol Specification for information about which ATID values are reserved.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCTRACEIDR

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCTRACEIDR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x040	TRCTRACEIDR

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !IsTraceCorePowered(), or !AllowExternalTraceAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.74 TRCTSCTLR, Trace Timestamp Control Register

The TRCTSCTLR characteristics are:

Purpose

Controls the insertion of global timestamps in the trace stream.

Configuration

External register TRCTSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCTSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR0.TSSIZE != '00000'. Otherwise, direct accesses to TRCTSCTLR are RES0.

Attributes

TRCTSCTLR is a 32-bit register.

Field descriptions

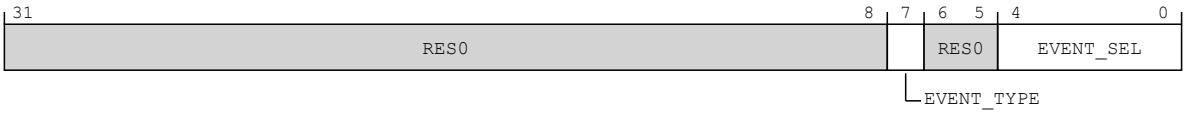

Bits [31:8]

Reserved, RES0.

EVENT_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCTSCTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCTSCTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCTSCTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVENT_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

Defines the selected Resource Selector or pair of Resource Selectors. TRCTSCTLR.EVENT.TYPE controls whether TRCTSCTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRCTSCTLR

Must be programmed if TRCCONFIGR.TS == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCTSCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x030`	TRCTSCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.75 TRCVICTLR, Trace ViewInst Main Control Register

The TRCVICTLR characteristics are:

Purpose

Controls instruction trace filtering.

Configuration

External register TRCVICTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVICTLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCVICTLR are RES0.

Attributes

TRCVICTLR is a 32-bit register.

Field descriptions

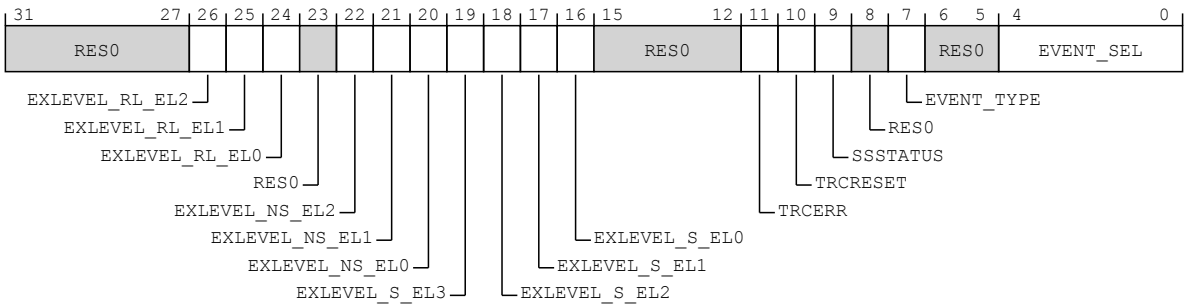

Bits [31:27]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [26]

When FEAT_RME is implemented:

Filter instruction trace for EL2 in Realm state.

EXLEVEL_RL_EL2	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit generates instruction trace for EL2 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit does not generate instruction trace for EL2 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit does not generate instruction trace for EL2 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit generates instruction trace for EL2 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL1, bit [25]

When FEAT_RME is implemented:

Filter instruction trace for EL1 in Realm state.

EXLEVEL_RL_EL1	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit generates instruction trace for EL1 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit does not generate instruction trace for EL1 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit does not generate instruction trace for EL1 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit generates instruction trace for EL1 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [24]

When FEAT_RME is implemented:

Filter instruction trace for EL0 in Realm state.

EXLEVEL_RL_EL0	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit generates instruction trace for EL0 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit does not generate instruction trace for EL0 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit does not generate instruction trace for EL0 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit generates instruction trace for EL0 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL2, bit [22]

When Non-secure EL2 is implemented:

Filter instruction trace for EL2 in Non-secure state.

EXLEVEL_NS_EL2	Meaning
`0b0`	The trace unit generates instruction trace for EL2 in Non-secure state.
`0b1`	The trace unit does not generate instruction trace for EL2 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [21]

When Non-secure EL1 is implemented:

Filter instruction trace for EL1 in Non-secure state.

EXLEVEL_NS_EL1	Meaning
`0b0`	The trace unit generates instruction trace for EL1 in Non-secure state.
`0b1`	The trace unit does not generate instruction trace for EL1 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL0, bit [20]

When Non-secure EL0 is implemented:

Filter instruction trace for EL0 in Non-secure state.

EXLEVEL_NS_EL0	Meaning
`0b0`	The trace unit generates instruction trace for EL0 in Non-secure state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	The trace unit does not generate instruction trace for EL0 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL3, bit [19]

When EL3 is implemented:

Filter instruction trace for EL3.

EXLEVEL_S_EL3	Meaning
0b0	The trace unit generates instruction trace for EL3.
0b1	The trace unit does not generate instruction trace for EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL2, bit [18]

When Secure EL2 is implemented:

Filter instruction trace for EL2 in Secure state.

EXLEVEL_S_EL2	Meaning
0b0	The trace unit generates instruction trace for EL2 in Secure state.
0b1	The trace unit does not generate instruction trace for EL2 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL1, bit [17]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Secure EL1 is implemented:

Filter instruction trace for EL1 in Secure state.

EXLEVEL_S_EL1	Meaning
0b0	The trace unit generates instruction trace for EL1 in Secure state.
0b1	The trace unit does not generate instruction trace for EL1 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL0, bit [16]

When Secure EL0 is implemented:

Filter instruction trace for EL0 in Secure state.

EXLEVEL_S_EL0	Meaning
0b0	The trace unit generates instruction trace for EL0 in Secure state.
0b1	The trace unit does not generate instruction trace for EL0 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [15:12]

Reserved, RES0.

TRCERR, bit [11]

When TRCIDR3.TRCERR == '1':

Controls the forced tracing of System Error exceptions.

TRCERR	Meaning
0b0	Forced tracing of System Error exceptions is disabled.
0b1	Forced tracing of System Error exceptions is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCERR	Meaning

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCRESET, bit [10]

Controls the forced tracing of PE Resets.

TRCRESET	Meaning
`0b0`	Forced tracing of PE Resets is disabled.
`0b1`	Forced tracing of PE Resets is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SSSTATUS, bit [9]

ViewInst start/stop function status.

SSSTATUS	Meaning
`0b0`	Stopped State. The ViewInst start/stop function is in the stopped state.
`0b1`	Started State. The ViewInst start/stop function is in the started state.

Before software enables the trace unit, it must write to this field to set the initial state of the ViewInst start/stop function. If the ViewInst start/stop function is not used then set this field to 1. Arm recommends that the value of this field is set before each trace session begins.

If the trace unit becomes disabled while a start point or stop point is still speculative, then the value of TRCVICTLR.SSSTATUS is UNKNOWN and might represent the result of a speculative start point or stop point.

If software which is running on the PE being traced disables the trace unit, either by clearing TRCPRGCTLR.EN or locking the OS Lock, Arm recommends that a DSB and an ISB instruction are executed before disabling the trace unit to prevent any start points or stop points being speculative at the point of disabling the trace unit. This procedure assumes that all start points or stop points occur before the barrier instructions are executed. The procedure does not guarantee that there are no speculative start points or stop points when disabling, although it helps minimize the probability.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES1 if all of the following are true:
  - TRCIDR4.NUMACPAIRS == ‘0000’

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCIDR4.NUMPC == ‘0000’
- Otherwise, access to this field is RW.

Bit [8]

Reserved, RES0.

EVENT_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCVICTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCVICTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCVICTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

Bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

EVENT_SEL, bits [4:0]

Defines the selected Resource Selector or pair of Resource Selectors. TRCVICTLR.EVENT.TYPE controls whether TRCVICTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

When TRCIDR4.NUMRSPAIR == '0000':

Reserved, bits [4:0]

This field is reserved:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits [4:1] are RES0.
- Bit [0] is RES1.

Otherwise:

Reserved, RES0.

Accessing TRCVICTLR

Must be programmed.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCVICTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x080`	TRCVICTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.76 TRCVIIECTLR, Trace ViewInst Include/Exclude Control Register

The TRCVIIECTLR characteristics are:

Purpose

Use this to select, or read, the Address Range Comparators for the ViewInst include/exclude function.

Configuration

External register TRCVIIECTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVIIECTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVIIECTLR are RES0.

Attributes

TRCVIIECTLR is a 32-bit register.

Field descriptions

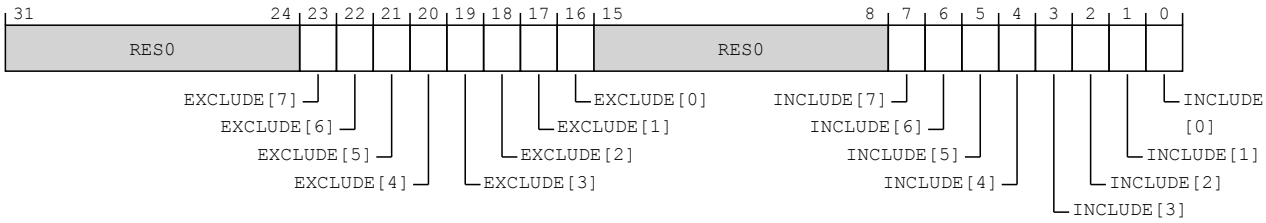

Bits [31:24]

Reserved, RES0.

EXCLUDE[<m>] , bits [m+16], for m = 7 to 0

Exclude Address Range Comparator <m>. Selects whether Address Range Comparator <m> is in use with the ViewInst exclude function.

EXCLUDE[<m>]	Meaning
0b0	The address range that Address Range Comparator <m> defines, is not selected for the ViewInst exclude function.
0b1	The address range that Address Range Comparator <m> defines, is selected for the ViewInst exclude function.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Bits [15:8]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INCLUDE[<m>] , bits [m], for m = 7 to 0

Include Address Range Comparator <m>.

Selects whether Address Range Comparator <m> is in use with the ViewInst include function.

Selecting no comparators for the ViewInst include function indicates that all instructions are included by default.

The ViewInst exclude function then indicates which ranges are excluded.

INCLUDE[<m>]	Meaning
`0b0`	The address range that Address Range Comparator <m> defines, is not selected for the ViewInst include function.
`0b1`	The address range that Address Range Comparator <m> defines, is selected for the ViewInst include function.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVIIECTLR

Must be programmed if TRCIDR4.NUMACPAIRS > `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVIIECTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x084`	TRCVIIECTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.77 TRCVIPCSSCTLR, Trace ViewInst Start/Stop PE Comparator Control Register

The TRCVIPCSSCTLR characteristics are:

Purpose

Use this to select, or read, which PE Comparator Inputs can control the ViewInst start/stop function.

Configuration

External register TRCVIPCSSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVIPCSSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMPC) > 0. Otherwise, direct accesses to TRCVIPCSSCTLR are RES0.

Attributes

TRCVIPCSSCTLR is a 32-bit register.

Field descriptions

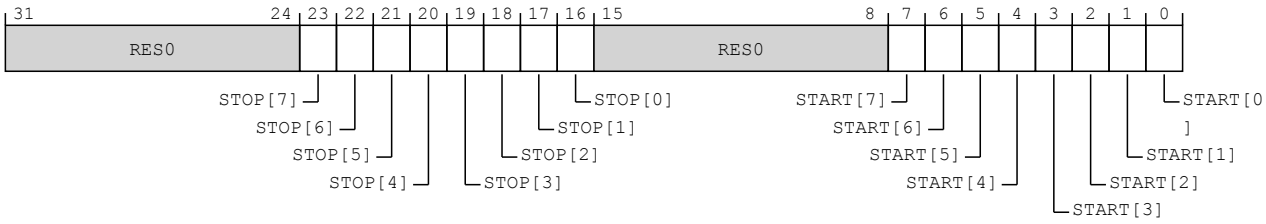

Bits [31:24]

Reserved, RES0.

STOP[<m>] , bits [m+16], for m = 7 to 0

Selects whether PE Comparator Input <m> is in use with the ViewInst start/stop function for the purpose of stopping trace.

STOP[<m>]	Meaning
0b0	The PE Comparator Input <m> is not selected as a stop resource.
0b1	The PE Comparator Input <m> is selected as a stop resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMPC), access to this field is RES0.
- Otherwise, access to this field is RW.

Bits [15:8]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

START[<m>] , bits [m], for m = 7 to 0

Selects whether PE Comparator Input <m> is in use with the ViewInst start/stop function for the purpose of starting trace.

START[<m>]	Meaning
`0b0`	The PE Comparator Input <m> is not selected as a start resource.
`0b1`	The PE Comparator Input <m> is selected as a start resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMPC), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVIPCSSCTLR

Must be programmed if TRCIDR4.NUMPC != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVIPCSSCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x08C`	TRCVIPCSSCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.78 TRCVISSCTLR, Trace ViewInst Start/Stop Control Register

The TRCVISSCTLR characteristics are:

Purpose

Use this to select, or read, the Single Address Comparators for the ViewInst start/stop function.

Configuration

External register TRCVISSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVISSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVISSCTLR are RES0.

Attributes

TRCVISSCTLR is a 32-bit register.

Field descriptions

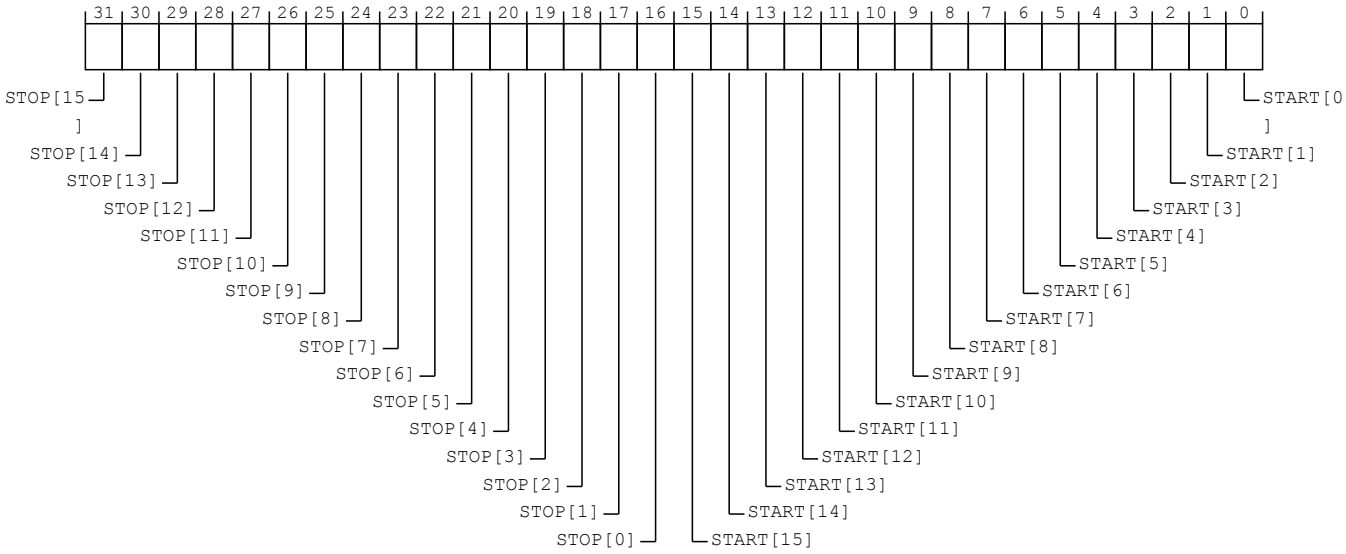

STOP[<m>] , bits [m+16], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function for the purpose of stopping trace.

STOP[<m>]	Meaning
0b0	The Single Address Comparator <m> is not selected as a stop resource.
0b1	The Single Address Comparator <m> is selected as a stop resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

START[<m>] , bits [m], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function for the purpose of starting trace.

START[<m>]	Meaning
`0b0`	The Single Address Comparator <m> is not selected as a start resource.
`0b1`	The Single Address Comparator <m> is selected as a start resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVISSCTLR

Must be programmed if TRCIDR4.NUMACPAIRS > `0b0000`.

For any 2 comparators selected for the ViewInst start/stop function, the comparator containing the lower address must be a lower numbered comparator.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVISSCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x088`	TRCVISSCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.79 TRCVMIDCCTLR0, Trace Virtual Context Identifier Comparator Control Register 0

The TRCVMIDCCTLR0 characteristics are:

Purpose

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=0-3.

Configuration

External register TRCVMIDCCTLR0 bits [31:0] are architecturally mapped to AArch64 System register TRCVMIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMVMIDC) > `0x0`, and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR0 are RES0.

Attributes

TRCVMIDCCTLR0 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP3[<m>]	COMP2[<m>]	COMP1[<m>]	COMP0[<m>]

COMP3[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 3:

TRCVMIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR3. Each bit in this field corresponds to a byte in TRCVMIDCVR3.

COMP3[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP2[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMVMIDC) > 2:

TRCVMIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR2. Each bit in this field corresponds to a byte in TRCVMIDCVR2.

COMP2[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP1[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 1:

TRCVMIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR1. Each bit in this field corresponds to a byte in TRCVMIDCVR1.

COMP1[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP0[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 0:

TRCVMIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR0. Each bit in this field corresponds to a byte in TRCVMIDCVR0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP0[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCVMIDCCTLR0

If software uses the TRCVMIDCVR<n> registers, where n=0-3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCCTLR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x688	TRCVMIDCCTLR0

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.80 TRCVMIDCCTLR1, Trace Virtual Context Identifier Comparator Control Register 1

The TRCVMIDCCTLR1 characteristics are:

Purpose

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=4-7.

Configuration

External register TRCVMIDCCTLR1 bits [31:0] are architecturally mapped to AArch64 System register TRCVMIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMVMIDC) > `0x4`, and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR1 are RES0.

Attributes

TRCVMIDCCTLR1 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP7[<m>]	COMP6[<m>]	COMP5[<m>]	COMP4[<m>]

COMP7[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 7:

TRCVMIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR7. Each bit in this field corresponds to a byte in TRCVMIDCVR7.

COMP7[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP6[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMVMIDC) > 6:

TRCVMIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR6. Each bit in this field corresponds to a byte in TRCVMIDCVR6.

COMP6[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP5[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 5:

TRCVMIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR5. Each bit in this field corresponds to a byte in TRCVMIDCVR5.

COMP5[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP4[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 4:

TRCVMIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR4. Each bit in this field corresponds to a byte in TRCVMIDCVR4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP4[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCVMIDCCTLR1

If software uses the TRCVMIDCVR<n> registers, where n=4-7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCCTLR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x68C	TRCVMIDCCTLR1

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.81 TRCVMIDCVR<n>, Trace Virtual Context Identifier Comparator Value Register <n>, n = 0 - 7

The TRCVMIDCVR<n> characteristics are:

Purpose

Contains the Virtual Context Identifier Comparator value.

Configuration

External register TRCVMIDCVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCVMIDCVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMVMIDC) > n. Otherwise, direct accesses to TRCVMIDCVR<n> are RES0.

Attributes

TRCVMIDCVR<n> is a 64-bit register.

Field descriptions

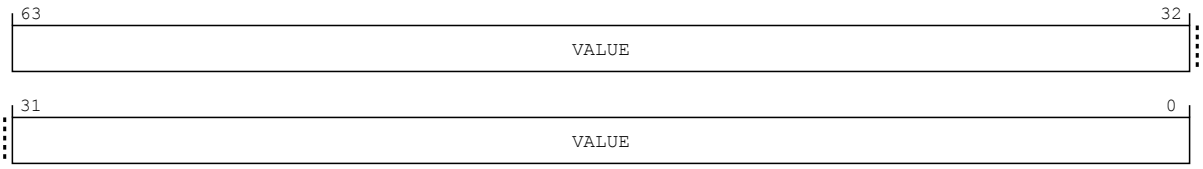

VALUE, bits [63:0]

Virtual context identifier value. The width of this field is indicated by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Virtual context identifier is zero until the PE updates the Virtual context identifier .

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCVMIDCVR<n>

Must be programmed if any of the following are true:

- TRCRSCTLR<a>.GROUP == `0b0111` and TRCRSCTLR<a>.VMID[n] == 1.
- TRCACATR<a>.CONTEXTTYPE == `0b10` or `0b11` and TRCACATR<a>.CONTEXT == n.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x640 + (8 * n)`	TRCVMIDCVR<n>

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4 External Trace Buffer registers

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.1 TRBAUTHSTATUS, Authentication Status Register

The TRBAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBAUTHSTATUS is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBAUTHSTATUS are RES0.

Attributes

TRBAUTHSTATUS is a 32-bit register.

Field descriptions

31:28	27:26	25:24	23:16	15:14	13:12	11:8	7:6	5:4	3:2	1:0
RES0	RTNID	RTID	RES0	RLNID	RLID	RES0	SNID	SID	NSNID	NSID

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

RTNID	Meaning
0b00	Not implemented.

RTID, bits [25:24]

Root invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTID.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLNID	Meaning
0b00	Not implemented.

RLID, bits [13:12]

Realm invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLID.

Bits [11:8]

Reserved, RES0.

SNID, bits [7:6]

Secure non-invasive debug.

SNID	Meaning
0b00	Not implemented.

SID, bits [5:4]

Secure invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

NSNID, bits [3:2]

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Not implemented.

NSID, bits [1:0]

Non-secure invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.NSID.

Accessing TRBAUTHSTATUS

TRBAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFB8	TRBAUTHSTATUS

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.2 TRBBASER_EL1, Trace Buffer Base Address Register

The TRBBASER_EL1 characteristics are:

Purpose

Defines the base address for the trace buffer.

Configuration

TRBBASER_EL1 is in the Core power domain

External register TRBBASER_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBBASER_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBBASER_EL1 are RES0.

Attributes

TRBBASER_EL1 is a 64-bit register.

Field descriptions

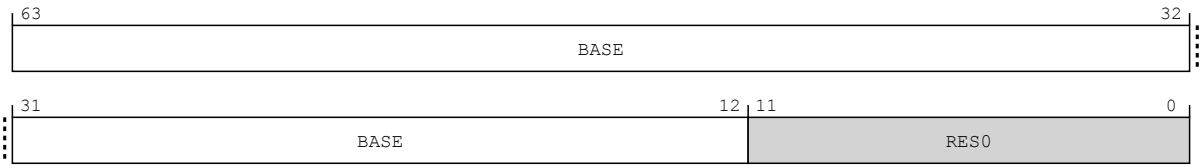

BASE, bits [63:12]

Trace Buffer Base pointer address. (TRBBASER_EL1.BASE « 12) is the address of the first byte in the trace buffer. Bits [11:0] of the Base pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBBASER_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

Accessing TRBBASER_EL1

The PE might ignore a write to TRBBASER_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBBASER_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0x000	TRBBASER_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.3 TRBCIDR0, Component Identification Register 0

The TRBCIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR0 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR0 are RES0.

Attributes

TRBCIDR0 is a 32-bit register.

Field descriptions

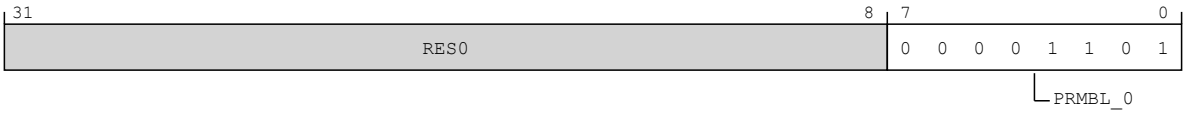

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Component identification preamble, segment 0.

Reads as `0x0D`

Access to this field is RO.

Accessing TRBCIDR0

TRBCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFF0`	TRBCIDR0

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.4 TRBCIDR1, Component Identification Register 1

The TRBCIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR1 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR1 are RES0.

Attributes

TRBCIDR1 is a 32-bit register.

Field descriptions

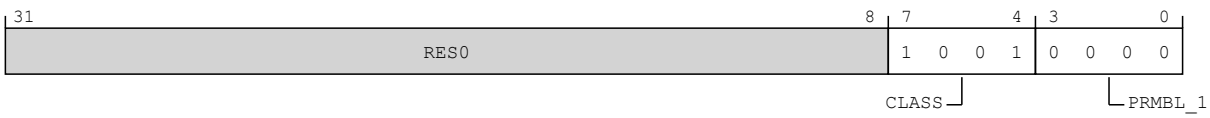

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight peripheral.

Other values are defined by the CoreSight Architecture.

Access to this field is RO.

PRMBL_1, bits [3:0]

Component identification preamble, segment 1.

Reads as 0b0000

Access to this field is RO.

Accessing TRBCIDR1

TRBCIDR1 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFF4`	TRBCIDR1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.5 TRBCIDR2, Component Identification Register 2

The TRBCIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR2 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR2 are RES0.

Attributes

TRBCIDR2 is a 32-bit register.

Field descriptions

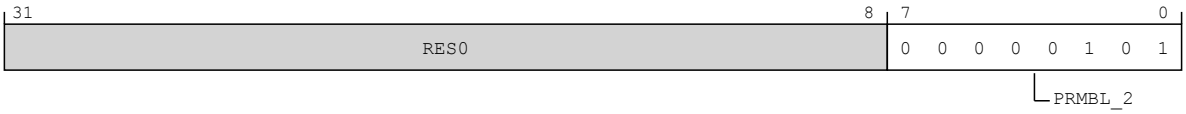

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Component identification preamble, segment 2.

Reads as `0x05`

Access to this field is RO.

Accessing TRBCIDR2

TRBCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFF8`	TRBCIDR2

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.6 TRBCIDR3, Component Identification Register 3

The TRBCIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR3 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR3 are RES0.

Attributes

TRBCIDR3 is a 32-bit register.

Field descriptions

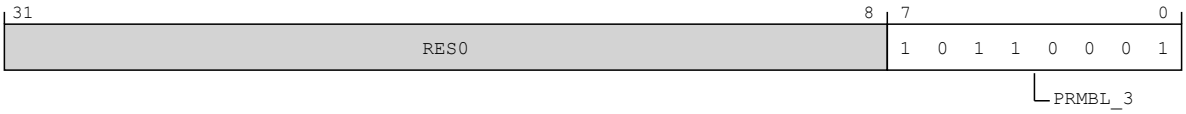

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Component identification preamble, segment 3.

Reads as `0xB1`

Access to this field is RO.

Accessing TRBCIDR3

TRBCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFFC`	TRBCIDR3

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.7 TRBCR, Trace Buffer Control Register

The TRBCR characteristics are:

Purpose

Provides trace buffer controls for an external debugger.

Configuration

TRBCR is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCR are RES0.

Attributes

TRBCR is a 64-bit register.

Field descriptions

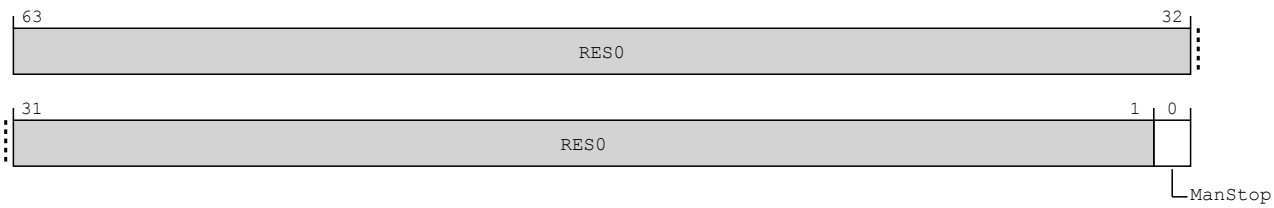

Bits [63:1]

Reserved, RES0.

ManStop, bit [0]

Flush and Stop collection. A write of 1 to this field causes a trace buffer flush, and on completion of the flush, Collection is stopped and the Trace Buffer Unit writes all trace data it has Accepted from the trace unit to memory, adding padding data if necessary.

Access to this field is WO/RAZ.

Accessing TRBCR

TRBCR can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0x038	TRBCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.8 TRBDEVAFF, Device Affinity Register

The TRBDEVAFF characteristics are:

Purpose

For additional information, see the CoreSight Architecture Specification.

Reads the same value as the MPIDR_EL1 register for the PE that this trace buffer has affinity with.

Depending on the IMPLEMENTATION DEFINED nature of the system, it might be possible that TRBDEVAFF is read before system firmware has configured the trace buffer and/or the PE or group of PEs that the trace buffer has affinity with. When this is the case, TRBDEVAFF reads as zero.

Configuration

TRBDEVAFF is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVAFF are RES0.

Attributes

TRBDEVAFF is a 64-bit register.

Field descriptions

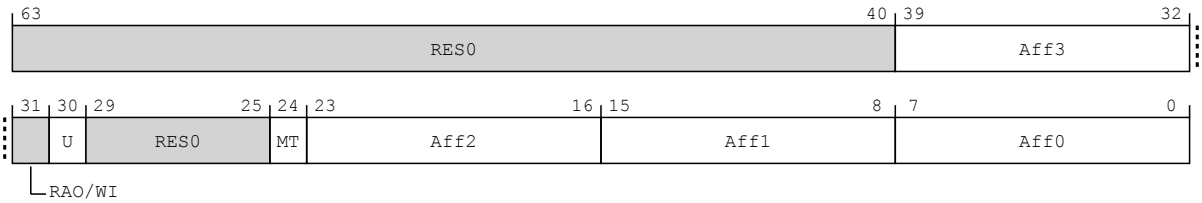

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRBDEVAFF

TRBDEVAFF can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFA8`	TRBDEVAFF

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.9 TRBDEVARCH, Trace Buffer Device Architecture Register

The TRBDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

Configuration

TRBDEVARCH is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVARCH are RES0.

Attributes

TRBDEVARCH is a 32-bit register.

Field descriptions

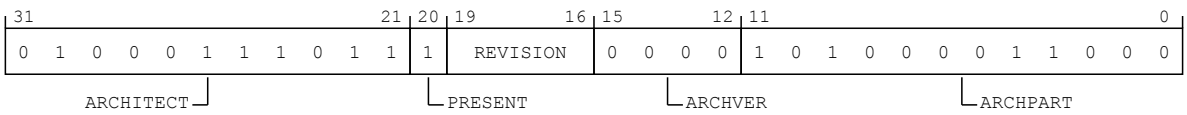

ARCHITECT, bits [31:21]

Defines the architect of the component. For Trace Buffer, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the TRBDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	First revision.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION	Meaning
0b0001	As 0b0000, and adds: • If EL2 and FEAT_FGT are implemented, a fine-grained trap on the TSB CSYNC instruction. • If EL2 is implemented, an EL2 control to override TRBLIMITR_EL1.nVM. • The TRBE Profiling exception extension, FEAT_TRBE_EXC.

All other values are reserved.

FEAT_TRBE implements the functionality identified by the value 0b0000.

FEAT_TRBEv1p1 implements the functionality identified by the value 0b0001.

From Armv9.6, the value 0b0000 is not permitted.

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0000	Trace Buffer Extension version 1.

All other values are reserved.

TRBDEVARCH.ARCHVER and TRBDEVARCH.ARCHPART are also defined as a single field, TRBDEVARCH.ARCHID, so that TRBDEVARCH.ARCHVER is TRBDEVARCH.ARCHID[15:12].

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA18	Armv9-A Trace Buffer Extension.

TRBDEVARCH.ARCHVER and TRBDEVARCH.ARCHPART are also defined as a single field, TRBDEVARCH.ARCHID, so that TRBDEVARCH.ARCHPART is TRBDEVARCH.ARCHID[11:0].

Access to this field is RO.

Accessing TRBDEVARCH

TRBDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFBC	TRBDEVARCH

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.10 TRBDEVID, Device Configuration Register

The TRBDEVID characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVID is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVID are RES0.

Attributes

TRBDEVID is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBDEVID

TRBDEVID can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFC8`	TRBDEVID

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.11 TRBDEVID1, Device Configuration Register 1

The TRBDEVID1 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVID1 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVID1 are RES0.

Attributes

TRBDEVID1 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 0
RES0	PMG_MAX	PARTID_MAX

Bits [31:24]

Reserved, RES0.

PMG_MAX, bits [23:16]

When FEAT_TRBE_MPAM is implemented:

Largest permitted PMG value. The TRBMPAM_EL1.PMG field must implement at least enough bits to represent TRBDEVID1.PMG_MAX.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

PARTID_MAX, bits [15:0]

When FEAT_TRBE_MPAM is implemented:

Largest permitted PARTID value. The TRBMPAM_EL1.PARTID field must implement at least enough bits to represent TRBDEVID1.PARTID_MAX.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBDEVID1

TRBDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFC4`	TRBDEVID1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.12 TRBDEVID2, Device Configuration Register 2

The TRBDEVID2 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVID2 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVID2 are RES0.

Attributes

TRBDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBDEVID2

TRBDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFC0`	TRBDEVID2

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.13 TRBDEVTYPE, Device Type Register

The TRBDEVTYPE characteristics are:

Purpose

Provides discovery information for the component. If the part number field is not recognized, a debugger can report the information that is provided by TRBDEVTYPE about the component instead.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVTYPE is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVTYPE are RES0.

Attributes

TRBDEVTYPE is a 32-bit register.

Field descriptions

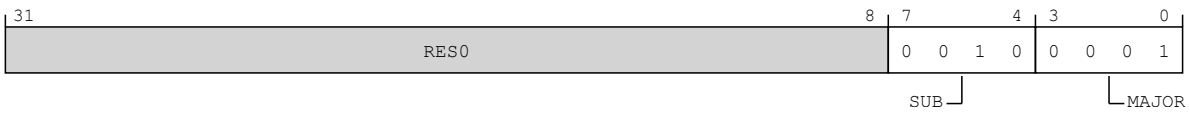

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Component sub-type.

SUB	Meaning
`0b0010`	When MAJOR == `0x1` (Trace sink), Trace buffer or router.

This field reads as `0x2`.

Access to this field is RO.

MAJOR, bits [3:0]

Component major type.

MAJOR	Meaning
`0b0001`	Trace sink.

This field reads as `0x1`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBDEVTYPE

TRBDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFCC`	TRBDEVTYPE

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.14 TRBIDR_EL1, Trace Buffer ID Register

The TRBIDR_EL1 characteristics are:

Purpose

Describes constraints on using the Trace Buffer Unit to an external debugger.

Configuration

TRBIDR_EL1 is in the Core power domain

External register TRBIDR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBIDR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBIDR_EL1 are RES0.

Attributes

TRBIDR_EL1 is a 64-bit register.

Field descriptions

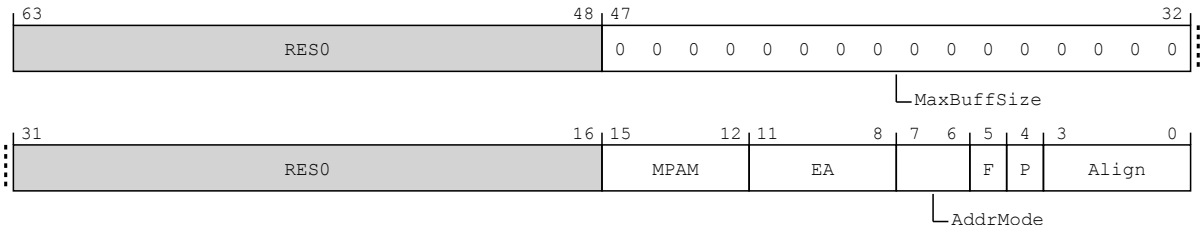

Bits [63:48]

Reserved, RES0.

MaxBuffSize, bits [47:32]

Maximum supported trace buffer size. Reserved for software use.

Reads as `0x0000`

The only permitted value is `0x0000`, indicating there is no limit to the maximum buffer size.

Access to this field is RO.

Bits [31:16]

Reserved, RES0.

MPAM, bits [15:12]

When FEAT_TRBE_EXT is implemented:

FEAT_MPAMv0p1 or FEAT_MPAMv1p0 extensions. Indicates Memory Partitioning and Monitoring (MPAM) support in the Trace Buffer Unit when using External mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM	Meaning
0b0000	FEAT_MPAMv0p1 and FEAT_MPAMv1p0 are not implemented by the PE.
0b0001	FEAT_MPAMv0p1 or FEAT_MPAMv1p0 is implemented by the Trace Buffer Unit, using Default physical PARTID and PMG values in External mode.
0b0010	Trace Buffer MPAM extensions implemented, using FEAT_MPAMv0p1 or FEAT_MPAMv1p0.

When FEAT_MPAMv0p1 and FEAT_MPAMv1p0 are not implemented by the PE, the only permitted value is 0b0000.

When either FEAT_MPAMv0p1 or FEAT_MPAMv1p0 is implemented by the PE, the value 0b0000 is not permitted.

FEAT_TRBE_MPAM implements the functionality identified by the value 0b0010.

Access to this field is RO.

Otherwise:

Reserved, RES0.

EA, bits [11:8]

External Abort handling. Describes how the PE manages External aborts on writes made by the Trace Buffer Unit to the trace buffer.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EA	Meaning
0b0000	Not described.
0b0001	The PE ignores External aborts on writes made by the Trace Buffer Unit.
0b0010	An External abort on a write made by the Trace Buffer Unit generates an asynchronous SError exception at the PE.

All other values are reserved.

From Armv9.3, the value 0b0000 is not permitted.

The behavior described by this field does not apply for External aborts on a translation table walk, translation table update, or GPT walk made by the Trace Buffer Unit that are reported as MMU faults using TRBSR_ELx. For more information, see ‘External aborts’.

Access to this field is RO.

AddrMode, bits [7:6]

This field reads as an UNKNOWN value.

F, bit [5]

Flag updates. Describes how address translations performed by the Trace Buffer Unit manage the Access flag and dirty state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F	Meaning
0b0	Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is always disabled for all translation stages.
0b1	Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is controlled in the same way as explicit memory accesses in the trace buffer owning translation regime.

Note

If hardware management of the Access flag is disabled for a stage of translation, an access to a Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.
If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor and might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

From Armv9.3, the value 0 is not permitted.

Access to this field is RO.

P, bit [4]

This field reads as an UNKNOWN value.

Align, bits [3:0]

Defines the minimum alignment constraint for writes to TRBPTR_EL1 and TRBTRG_EL1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Align	Meaning
0b0000	Byte.
0b0001	Halfword.
0b0010	Word.
0b0011	Doubleword.
0b0100	16 bytes.
0b0101	32 bytes.
0b0110	64 bytes.
0b0111	128 bytes.
0b1000	256 bytes.
0b1001	512 bytes.
0b1010	1KB.
0b1011	2KB.

All other values are reserved.

Access to this field is RO.

Accessing TRBIDR_EL1

TRBIDR_EL1 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0x030`	TRBIDR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.15 TRBITCTRL, Integration Mode Control Register

The TRBITCTRL characteristics are:

Purpose

A component can use TRBITCTRL to dynamically switch between functional mode and integration mode. In integration mode, topology detection is enabled. After switching to integration mode and performing integration tests or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBITCTRL is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBITCTRL are RES0.

Attributes

TRBITCTRL is a 32-bit register.

Field descriptions

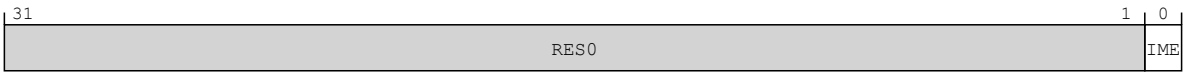

Bits [31:1]

Reserved, RES0.

IME, bit [0]

When topology detection or integration functionality is implemented:

Integration Mode Enable.

IME	Meaning
0b0	Component functional mode.
0b1	Component integration mode. Support for topology detection and integration testing is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRBITCTRL

The PE might ignore a write to TRBITCTRL if any of the following apply:

- TRBLIMITR_EL1.E == 1 and Trace Buffer Unit is using Self-hosted mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBLIMITR_EL1.XE == 1 and the Trace Buffer Unit is using External mode.

TRBITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xF00`	TRBITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.16 TRBLAR, Lock Access Register

The TRBLAR characteristics are:

Purpose

For components that implement the Software Lock, used to lock and unlock the Software Lock. This component does not implement the Software Lock.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBLAR is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBLAR are RES0.

Attributes

TRBLAR is a 32-bit register.

Field descriptions

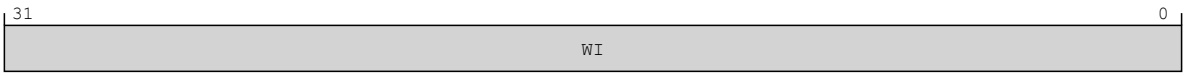

Bits [31:0]

Reserved, WI.

Software Lock Key. The Software Lock is not implemented.

This field ignores writes.

Accessing TRBLAR

TRBLAR can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFB0`	TRBLAR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.17 TRBLIMITR_EL1, Trace Buffer Limit Address Register

The TRBLIMITR_EL1 characteristics are:

Purpose

Defines the top address for the trace buffer, and controls the trace buffer modes and enable.

Configuration

TRBLIMITR_EL1 is in the Core power domain

External register TRBLIMITR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBLIMITR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBLIMITR_EL1 are RES0.

Attributes

TRBLIMITR_EL1 is a 64-bit register.

Field descriptions

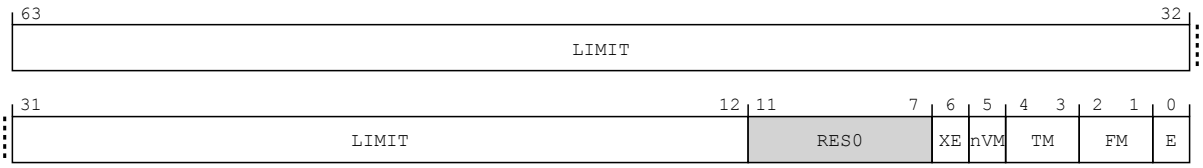

LIMIT, bits [63:12]

Trace buffer Limit pointer address. (TRBLIMITR_EL1.LIMIT « 12) is the address of the last byte in the trace buffer plus one. Bits [11:0] of the Limit pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBLIMITR_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

XE, bit [6]

Trace Buffer Unit External mode enable. Controls whether the Trace Buffer Unit is enabled when `SelfHostedTraceEnabled()` == FALSE.

XE	Meaning
`0b0`	Trace Buffer Unit is not enabled by this control.
`0b1`	If `SelfHostedTraceEnabled()` is FALSE, the Trace Buffer Unit is enabled.

If `SelfHostedTraceEnabled()` == TRUE, then TRBLIMITR_EL1.E controls whether the Trace Buffer Unit is enabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All output is discarded by the Trace Buffer Unit when the Trace Buffer Unit is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

nVM, bit [5]

Address mode.

nVM	Meaning
`0b0`	The trace buffer pointers are virtual addresses.
`0b1`	The trace buffer pointers are: • Physical address in the owning security state if the owning translation regime has no stage 2 translation. • Intermediate physical addresses in the owning security state if the owning translation regime has stage 2 translations.

If `SelfHostedTraceEnabled()` == FALSE, then the PE ignores the value of this field and the trace buffer pointers are always physical addresses.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When !SelfHostedTraceEnabled(), access to this field is RES1.
- Otherwise, access to this field is RW.

TM, bits [4:3]

Trigger mode.

TM	Meaning
`0b00`	Stop on trigger. Flush trace, then stop collection and set TRBSR_EL1.IRQ to 1 on Trigger Event.
`0b01`	IRQ on trigger. Continue collection and set TRBSR_EL1.IRQ to 1 on Trigger Event.
`0b11`	Ignore trigger. Continue collection and leave TRBSR_EL1.IRQ unchanged on Trigger Event.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

FM, bits [2:1]

Trace buffer mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FM	Meaning
0b00	Fill mode. Stop collection and set TRBSR_EL1.IRQ to 1 on current write pointer wrap.
0b01	Wrap mode. Continue collection and set TRBSR_EL1.IRQ to 1 on current write pointer wrap.
0b11	Circular Buffer mode. Continue collection and leave TRBSR_EL1.IRQ unchanged on current write pointer wrap.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Trace Buffer Unit enable. Controls whether the Trace Buffer Unit is enabled when `SelfHostedTraceEnabled()` == TRUE.

E	Meaning
0b0	Trace Buffer Unit is not enabled by this control.
0b1	If `SelfHostedTraceEnabled()` is TRUE, the Trace Buffer Unit is enabled.

If FEAT_TRBE_EXT is implemented and `SelfHostedTraceEnabled()` == FALSE, then TRBLIMITR_EL1.XE controls whether the Trace Buffer Unit is enabled.

If FEAT_TRBE_EXT is not implemented, then the Trace Buffer Unit is disabled when `SelfHostedTraceEnabled()` == FALSE.

All output is discarded by the Trace Buffer Unit when the Trace Buffer Unit is disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRBLIMITR_EL1

The PE might ignore a write to TRBLIMITR_EL1, other than a write that modifies TRBLIMITR_EL1.E or TRBLIMITR_EL1.XE as appropriate, if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBLIMITR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0x010	TRBLIMITR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.18 TRBLSR, Lock Status Register

The TRBLSR characteristics are:

Purpose

Indicates the Software Lock is not implemented.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBLSR is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBLSR are RES0.

Attributes

TRBLSR is a 32-bit register.

Field descriptions

31 … 3	2 … 1	0
RES0	RAZ	SLI

Bits [31:3]

Reserved, RES0.

Bits [2:1]

Reserved, RAZ.

Not thirty-two bit. Describes the size of the TRBLAR register.

This field reads-as-zero.

SLI, bit [0]

Indicates the Software Lock is not implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock is not implemented. Writes to the TRBLAR are ignored.
0b1	Software Lock is implemented.

Access to this field is RAZ/WI.

Accessing TRBLSR

TRBLSR can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFB4`	TRBLSR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.19 TRBMAR_EL1, Trace Buffer Memory Attribute Register

The TRBMAR_EL1 characteristics are:

Purpose

Controls Trace Buffer Unit accesses to memory.

Configuration

TRBMAR_EL1 is in the Core power domain

External register TRBMAR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBMAR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBMAR_EL1 are RES0.

Attributes

TRBMAR_EL1 is a 64-bit register.

Field descriptions

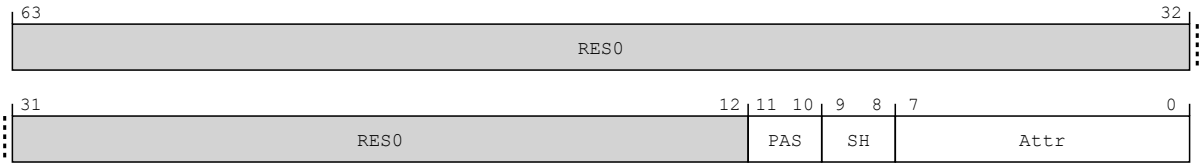

Bits [63:12]

Reserved, RES0.

PAS, bits [11:10]

Physical address specifier. Defines the PAS attribute for memory addressed by the buffer in External mode.

PAS	Meaning	Applies when
`0b00`	Secure.	FEAT_Secure is implemented
`0b01`	Non-secure.	
`0b10`	Root.	FEAT_RME is implemented
`0b11`	Realm.	FEAT_RME is implemented

All other values are reserved.

If the Trace Buffer Unit is using external mode and either TRBMAR_EL1.PAS is set to a reserved value, or the IMPLEMENTATION DEFINED authentication interface prohibits invasive debug of the Security state corresponding to the physical address space selected by TRBMAR_EL1.PAS, then when the Trace Buffer Unit receives trace data from the trace unit, it does not write the trace data to memory and generates a trace buffer management event. That is, if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- TRBMAR_EL1.PAS is `0b00`, and either `ExternalSecureInvasiveDebugEnabled()` == FALSE or Secure state is not implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBMAR_EL1.PAS is `0b10`, and either `ExternalRootInvasiveDebugEnabled()` == FALSE or FEAT_RME is not implemented.
- TRBMAR_EL1.PAS is `0b11`, and either `ExternalRealmInvasiveDebugEnabled()` == FALSE or FEAT_RME is not implemented.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SH, bits [9:8]

Trace buffer stage 1 shareability domain. Defines the shareability domain for Normal memory used by the trace buffer.

SH	Meaning
`0b00`	Non-shareable.
`0b10`	Outer Shareable.
`0b11`	Inner Shareable.

All other values are reserved.

If the trace buffer pointers specify virtual addresses, the memory attributes are defined by the translation tables and this field is ignored.

This field is ignored when TRBMAR_EL1.Attr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Attr, bits [7:0]

Trace buffer stage 1 memory type and attributes. Defines the memory type and, for Normal memory, the cacheability attributes, for memory addressed by the trace buffer.

The encoding of this field is the same as that of a MAIR_ELx.Attr<n> field, as follows:

Attr		Meaning
`0b0000`dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
`0b0000`dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
`0b0000`dd1x		UNPREDICTABLE.
0boooooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
`0b01000000`		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0b10100000		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`R` encodes the Inner Read-Allocate policy and `W` encodes the Inner Write-Allocate policy.

In `oooo` and `iiii`, `R` and `W` are encoded as follows:

'R' or 'W'	Meaning
`0b0`	No Allocate.
`0b1`	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

If the trace buffer pointers specify virtual addresses, the memory attributes are defined by the translation tables and this field is ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBMAR_EL1

The PE might ignore a write to TRBMAR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBMAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x028`	TRBMAR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.20 TRBMPAM_EL1, Trace Buffer MPAM Configuration Register

The TRBMPAM_EL1 characteristics are:

Purpose

Defines the PARTID, PMG, and MPAM_SP values used by the trace buffer unit in external mode.

Configuration

TRBMPAM_EL1 is in the Core power domain

External register TRBMPAM_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBMPAM_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented and FEAT_TRBE_MPAM is implemented. Otherwise, direct accesses to TRBMPAM_EL1 are RES0.

Attributes

TRBMPAM_EL1 is a 64-bit register.

Field descriptions

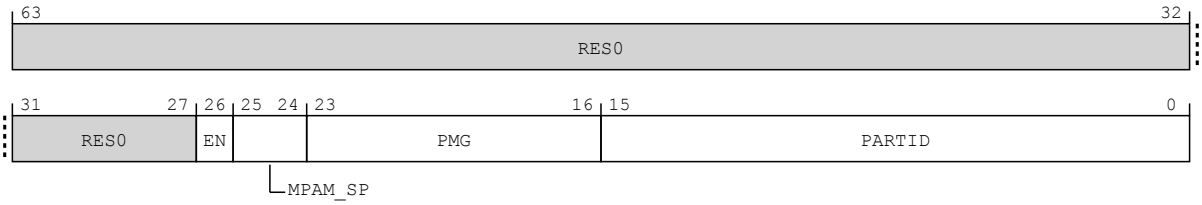

Bits [63:27]

Reserved, RES0.

EN, bit [26]

Enable. Enables use of non-Default MPAM values.

EN	Meaning
0b0	Use Default physical PARTID and Default physical PMG values, and the PARTID space corresponding to the trace physical address space defined by TRBMAR_EL1.PAS.
0b1	Use TRBMPAM_EL1.{PARTID, PMG, MPAM_SP}.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

MPAM_SP, bits [25:24]

Partition Identifier space. Selects the PARTID space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM_SP	Meaning	Applies when
`0b00`	PARTID is in the Secure PARTID space.	FEAT_Secure is implemented
`0b01`	PARTID is in the Non-secure PARTID space.	
`0b10`	PARTID is in the Root PARTID space.	FEAT_RME is implemented
`0b11`	PARTID is in the Realm PARTID space.	FEAT_RME is implemented

If the Trace Buffer Unit is using external mode and either TRBMPAM_EL1.MPAM_SP is set to reserved value, or the IMPLEMENTATION DEFINED authentication interface prohibits invasive debug of the Security state corresponding to the Partition Identifier space selected by TRBMPAM_EL1.MPAM_SP, then when the Trace Buffer Unit receives trace data from the trace unit, it does not write the trace data to memory and generates a trace buffer management event.

The interface prohibits invasive debug of the Security state if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure state is implemented, `ExternalSecureInvasiveDebugEnabled()` == FALSE and TRBMPAM_EL1.MPAM_SP is `0b00`.
- FEAT_RME is implemented, `ExternalRootInvasiveDebugEnabled()` == FALSE, and TRBMPAM_EL1.MPAM_SP is `0b10`.
- FEAT_RME is implemented, `ExternalRealmInvasiveDebugEnabled()` == FALSE, and TRBMPAM_EL1.MPAM_SP is `0b11`.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PMG, bits [23:16]

Performance Monitoring Group. Selects the PMG.

Only sufficient low-order bits are required to represent the TRBDEVID1.PMG_MAX. Higher-order bits are RES0.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PARTID, bits [15:0]

Partition Identifier. Selects the PARTID.

Only sufficient low-order bits are required to represent the TRBDEVID1.PARTID_MAX. Higher-order bits are RES0.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBMPAM_EL1

The PE might ignore a write to TRBMPAM_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBMPAM_EL1 can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0x040`	TRBMPAM_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.21 TRBPIDR0, Peripheral Identification Register 0

The TRBPIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR0 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR0 are RES0.

Attributes

TRBPIDR0 is a 32-bit register.

Field descriptions

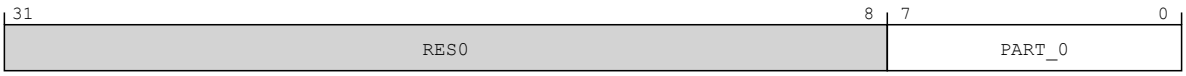

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - TRBPIDR1.PART_1 contains part number bits [11:8].
  - TRBPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - TRBPIDR1.PART_1 contains part number bits [15:12].
  - TRBPIDR0.PART_0 contains part number bits [11:4].
  - TRBPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, TRBPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRBPIDR0

TRBPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFE0`	TRBPIDR0

Accessible as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.22 TRBPIDR1, Peripheral Identification Register 1

The TRBPIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR1 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR1 are RES0.

Attributes

TRBPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- TRBPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRBPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRBPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - TRBPIDR1.PART_1 contains part number bits [11:8].
  - TRBPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - TRBPIDR1.PART_1 contains part number bits [15:12].
  - TRBPIDR0.PART_0 contains part number bits [11:4].
  - TRBPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, TRBPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRBPIDR1

TRBPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFE4`	TRBPIDR1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.23 TRBPIDR2, Peripheral Identification Register 2

The TRBPIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR2 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR2 are RES0.

Attributes

TRBPIDR2 is a 32-bit register.

Field descriptions

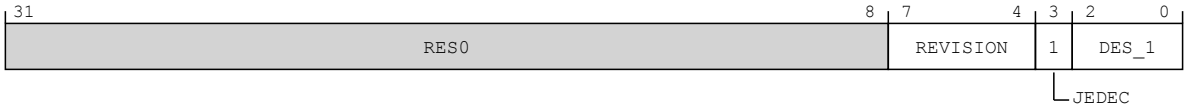

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- TRBPIDR2.REVISION indicates the 4-bit revision number.
- TRBPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- TRBPIDR2.REVISION indicates the 4-bit major revision number.
- TRBPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- TRBPIDR2.REVISION contains part number bits [3:0].
- TRBPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- TRBPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRBPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRBPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing TRBPIDR2

TRBPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFE8`	TRBPIDR2

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.24 TRBPIDR3, Peripheral Identification Register 3

The TRBPIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR3 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR3 are RES0.

Attributes

TRBPIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- TRBPIDR2.REVISION indicates the 4-bit revision number.
- TRBPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- TRBPIDR2.REVISION indicates the 4-bit major revision number.
- TRBPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- TRBPIDR2.REVISION contains part number bits [3:0].
- TRBPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

TRBPIDR3.CMOD might also indicate component modifications.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

TRBPIDR3.REVAND might also indicate component modifications.

Access to this field is RO.

Accessing TRBPIDR3

TRBPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFEC`	TRBPIDR3

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.25 TRBPIDR4, Peripheral Identification Register 4

The TRBPIDR4 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR4 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR4 are RES0.

Attributes

TRBPIDR4 is a 32-bit register.

Field descriptions

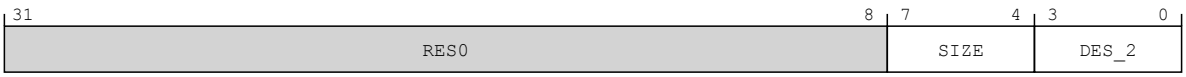

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIZE	Meaning
0b0000	One of the following is true: • The component uses a single 4KB block. • The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.
0b0001..0b1111	The component occupies $2^{\text{TRBPIDR4.SIZE}}$ 4KB blocks.

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRBPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRBPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing TRBPIDR4

TRBPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFD0`	TRBPIDR4

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.26 TRBPIDR5, Peripheral Identification Register 5

The TRBPIDR5 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR5 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR5 are RES0.

Attributes

TRBPIDR5 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBPIDR5

TRBPIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFD4	TRBPIDR5

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.27 TRBPIDR6, Peripheral Identification Register 6

The TRBPIDR6 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR6 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR6 are RES0.

Attributes

TRBPIDR6 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBPIDR6

TRBPIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFD8	TRBPIDR6

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.28 TRBPIDR7, Peripheral Identification Register 7

The TRBPIDR7 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR7 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR7 are RES0.

Attributes

TRBPIDR7 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBPIDR7

TRBPIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFDC	TRBPIDR7

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.29 TRBPTR_EL1, Trace Buffer Write Pointer Register

The TRBPTR_EL1 characteristics are:

Purpose

Defines the current write pointer for the trace buffer.

Configuration

TRBPTR_EL1 is in the Core power domain

External register TRBPTR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBPTR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPTR_EL1 are RES0.

Attributes

TRBPTR_EL1 is a 64-bit register.

Field descriptions

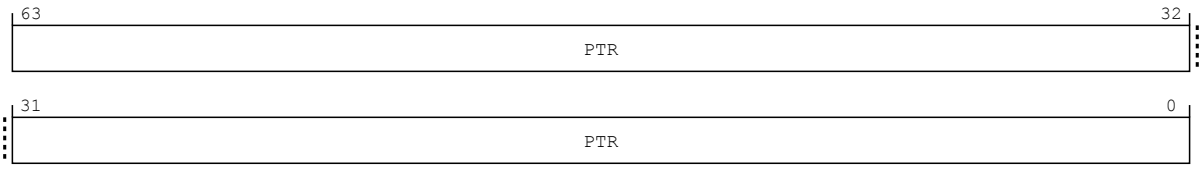

PTR, bits [63:0]

Trace Buffer current write pointer address.

Defines the virtual address of the next entry to be written to the trace buffer.

If TRBIDR_EL1.Align is not zero, then it is IMPLEMENTATION DEFINED whether bits [M-1:0] are RES0 or read/write, where M is an integer between 1 and TRBIDR_EL1.Align inclusive.

The architecture places restrictions on the values that software can write to the pointer. For more information see Restrictions on Programming the Trace Buffer Unit.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBPTR_EL1

The PE might ignore a write to TRBPTR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBPTR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0x008	TRBPTR_EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.30 TRBSR_EL1, Trace Buffer Status/syndrome Register

The TRBSR_EL1 characteristics are:

Purpose

Provides syndrome information to software for a trace buffer management event.

Configuration

TRBSR_EL1 is in the Core power domain

External register TRBSR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBSR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBSR_EL1 are RES0.

Attributes

TRBSR_EL1 is a 64-bit register.

Field descriptions

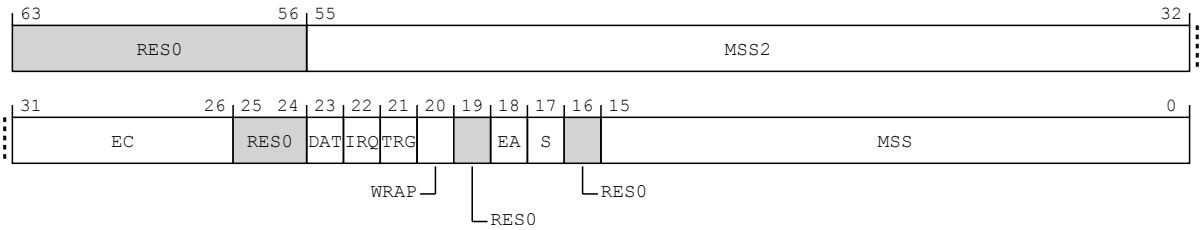

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other trace buffer management events

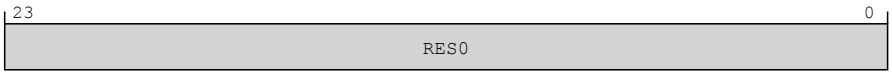

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

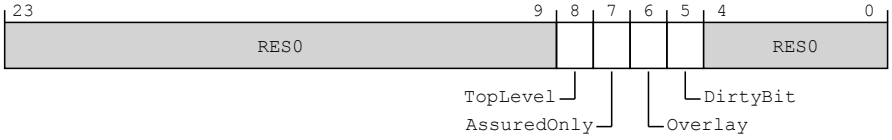

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, TRBSR_EL1.EC == '100101', and GetTRBSR_EL1_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetTRBSR_EL1_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetTRBSR_EL1_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to trace buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the trace buffer management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other trace buffer management event. All trace buffer management events other than those described by the other defined Event class codes.	MSS encoding for other trace buffer management events	MSS2 encoding for other trace buffer management events	
0b011110	Granule Protection Check fault on write to trace buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to trace buffer	MSS2 encoding for Granule Protection Check faults on write to trace buffer	FEAT_RME is implemented
0b011111	Trace buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	
0b100101	Stage 2 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [25:24]

Reserved, RES0.

DAT, bit [23]

When FEAT_TRBE_EXT is implemented:

Data. Indicates when the Trace Buffer Unit has trace data that has not yet been written to memory.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DAT	Meaning
0b0	Internal buffers are empty. All Trace operations Accepted by the Trace Buffer Unit will Complete in finite time.
0b1	Internal buffers are not empty.

When TRBSR_EL1.{DAT, S} is {0, 1}, meaning Collection is stopped and the Trace Buffer Unit internal buffers are empty, then all trace data has been written to memory. An additional Data Synchronization Barrier may be required to ensure that the writes are Complete. When TRBSR_EL1.DAT is 0 and Collection is not stopped, there may still be trace data held by the trace unit that the Trace Buffer Unit has not Accepted.

That is, TRBSR_EL1.DAT reads as 1 when the Trace Buffer Unit has Accepted trace data from the trace unit, but has not yet written it to memory.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IRQ, bit [22]

Maintenance status. Indicates that a trace buffer management event has been recorded.

IRQ	Meaning
0b0	No trace buffer management event for EL1 has been recorded.
0b1	A trace buffer management event for EL1 has been recorded.

When FEAT_TRBE_EXC is implemented, this field indicates a management event for EL1.

If FEAT_TRBE_EXC is implemented and the TRBE Profiling exception for EL1 is enabled, then when this field is 1, a TRBE Profiling exception for EL1 is pending

If FEAT_TRBE_EXC is not implemented or the TRBE Profiling exception for EL1 is disabled, then this field drives the **TRBIRQ** trace buffer interrupt request signal.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

TRG, bit [21]

Triggered.

TRG	Meaning
0b0	No Detected Trigger has been observed since this field was last cleared to zero.
0b1	A Detected Trigger has been observed since this field was last cleared to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WRAP, bit [20]

Wrapped.

WRAP	Meaning
0b0	The current write pointer has not wrapped since this field was last cleared to zero.
0b1	The current write pointer has wrapped since this field was last cleared to zero.

For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in the current write pointer, if the current write pointer is equal to the Limit pointer minus one, the current write pointer is wrapped by setting it to the Base pointer, and this field is set to 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

EA, bit [18]

When the PE sets this bit as the result of an External abort:

External Abort.

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Trace Buffer Unit.

It is IMPLEMENTATION DEFINED whether this field is set to 1 or is unchanged by the PE when a write by the Trace Buffer Unit generates an External abort on a translation table walk, translation table update, or GPT walk that is reported as an MMU fault. From Armv9.3, this field is not set to 1 by the PE for any other trace buffer management event.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Stopped.

S	Meaning
0b0	Collection has not been stopped.
0b1	Collection is stopped.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [16]

Reserved, RES0.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the trace buffer management event.

The syndrome contents for each trace buffer management event are described in the following sections.

MSS encoding for other trace buffer management events

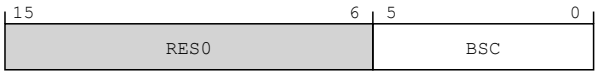

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Trace buffer status code

BSC	Meaning
0b000000	Collection not stopped, or access not allowed.
0b000001	Trace buffer filled. Collection stopped because the current write pointer wrapped to the base pointer and the trace buffer mode is Fill mode.
0b000010	Trigger Event. Collection stopped because of a Trigger Event. See TRBTRG_EL1 for more information.
0b000011	Manual Stop. Collection stopped because of a Manual Stop event. See TRBCR.ManStop for more information.
0b000100	Buffer size. The requested trace buffer size was too large.

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

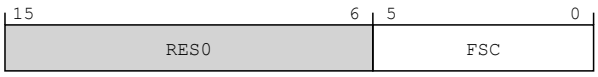

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAF is implemented

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to trace buffer

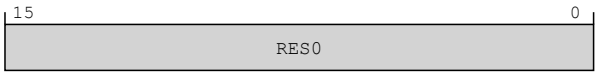

Bits [15:0]

Reserved, RES0.

MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

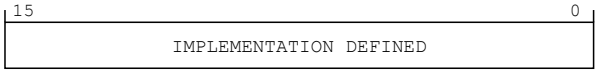

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBSR_EL1

The PE might ignore a write to TRBSR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBSR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x018`	TRBSR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.31 TRBTRG_EL1, Trace Buffer Trigger Counter Register

The TRBTRG_EL1 characteristics are:

Purpose

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

Configuration

TRBTRG_EL1 is in the Core power domain

External register TRBTRG_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBTRG_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBTRG_EL1 are RES0.

Attributes

TRBTRG_EL1 is a 64-bit register.

Field descriptions

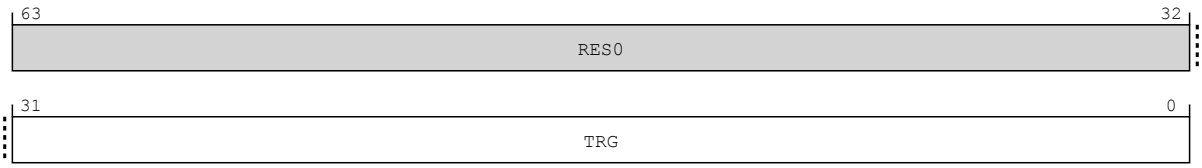

Bits [63:32]

Reserved, RES0.

TRG, bits [31:0]

Trigger count.

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

TRBTRG_EL1 decrements by 1 for every byte of trace written to the trace buffer when all of the following are true:

- TRBTRG_EL1 is nonzero.
- TRBSR_EL1.TRG is 1.

The architecture places restrictions on the values that software can write to the counter.

Note

As a result of the restrictions an implementation might treat some of TRG[M:0] as RES0, where M is defined by TRBIDR_EL1.Align.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBTRG_EL1

The PE might ignore a write to TRBTRG_EL1 if any of the following apply:

- TRBLIMITR_EL1.E is 1 and the Trace Buffer Unit is using Self-hosted mode.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBLIMITR_EL1.XE is 1 and the Trace Buffer Unit is using External mode.

TRBTRG_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x020`	TRBTRG_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

Purpose

Can be used to provide IMPLEMENTATION DEFINED controls for the CTI. For example, the register might be used to control multiplexors for additional IMPLEMENTATION DEFINED triggers. The IMPLEMENTATION DEFINED controls provided by this register might modify the architecturally defined behavior of the CTI.

Note

The architecturally-defined triggers must not be multiplexed.

Configuration

It is IMPLEMENTATION DEFINED whether ASICCTL is implemented in the Core power domain or in the Debug power domain

If it is implemented in the Core power domain, then it is IMPLEMENTATION DEFINED whether it is in the Cold reset domain or the Warm reset domain.

This register must reset to a value that supports the architecturally-defined behavior of the CTI. Changing the value of the register from its reset value causes IMPLEMENTATION DEFINED behavior that might differ from the architecturally-defined behavior of the CTI.

Other than the requirements listed in this register description, all aspects of the reset behavior of the ASICCTL are IMPLEMENTATION DEFINED.

Attributes

ASICCTL is a 32-bit register.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing ASICCTL

ASICCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x144`	ASICCTL

Accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess(addrdesc), and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

Purpose

Clears the application triggers.

Configuration

CTIAPPCLEAR is in the Debug power domain

External register CTIAPPCLEAR bits [31:0] are architecturally mapped to External register CTIAPPSET[31:0].

Attributes

CTIAPPCLEAR is a 32-bit register.

Field descriptions

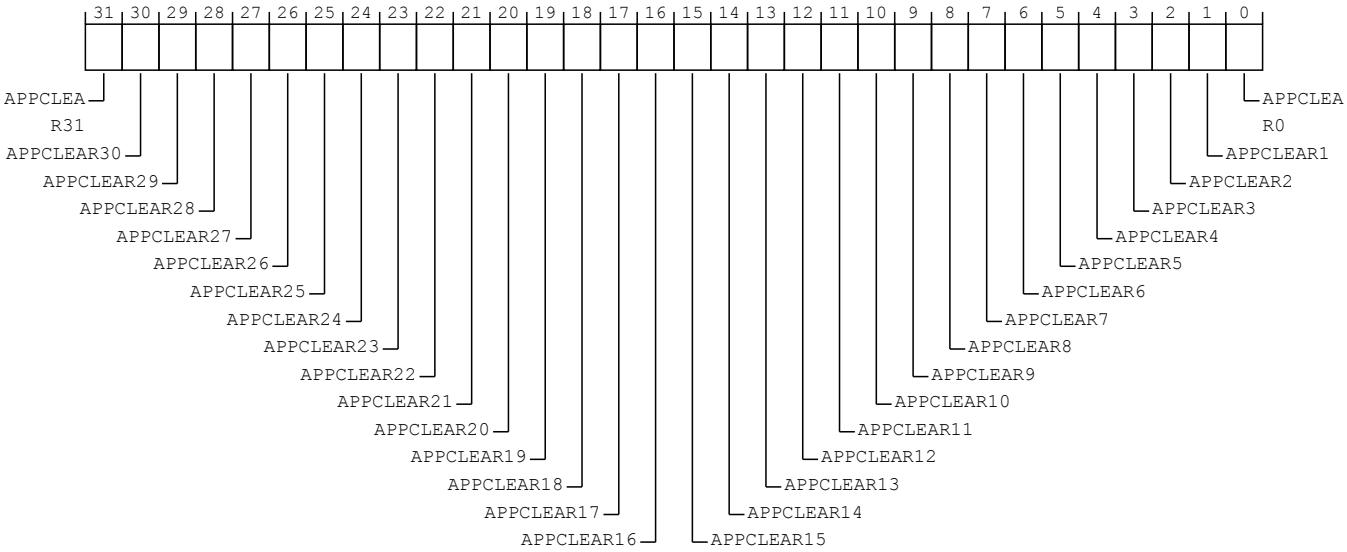

APPCLEAR<x> , bits [x], for x = 31 to 0

Application trigger <x> disable.

Writing to this bit has the following effect:

APPCLEAR<x>	Meaning
0b0	No effect.
0b1	Clear corresponding application trigger to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use CTIAPPPULSE.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RAZ/W1C.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAPPCLEAR

CTIAPPCLEAR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x018`	CTIAPPCLEAR

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

Purpose

Causes event pulses to be generated on ECT channels.

Configuration

CTIAPPPULSE is in the Debug power domain

Attributes

CTIAPPPULSE is a 32-bit register.

Field descriptions

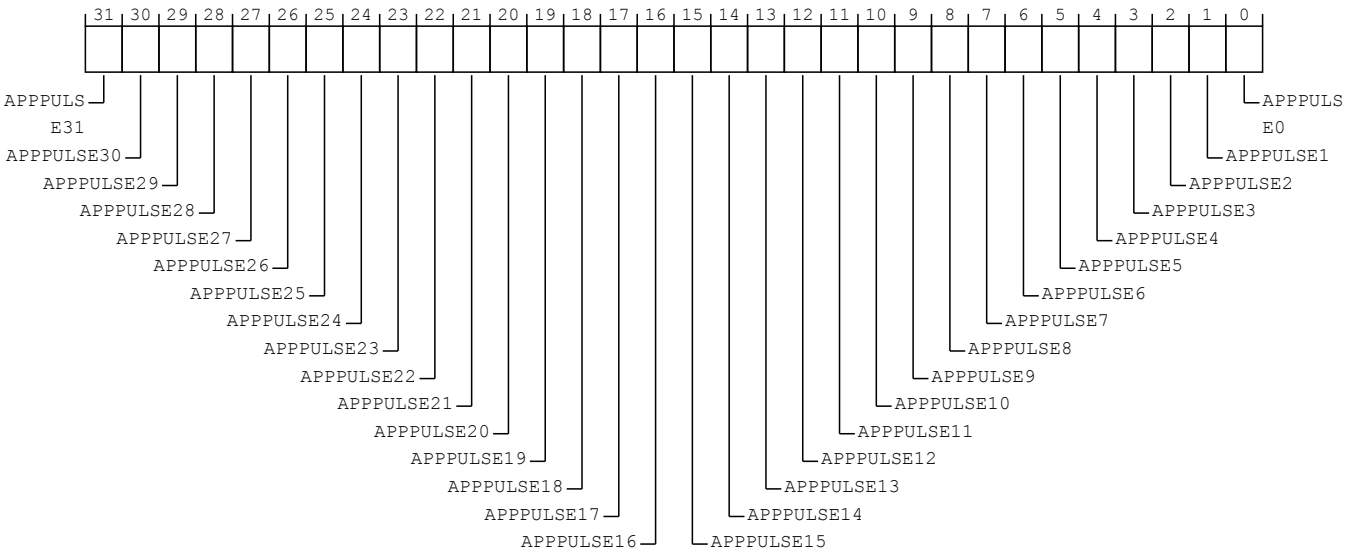

APPPULSE<x> , bits [x], for x = 31 to 0

Generate event pulse on ECT channel <x>.

APPPULSE<x>	Meaning
0b0	No effect.
0b1	Channel <x> event pulse generated.

Note

- The CTIAPPPULSE operation does not affect the state of the application trigger. If the channel is active, either because of an earlier event or from the application trigger, then the value written to CTIAPPPULSE might have no effect.
- Multiple pulse events that occur close together might be merged into a single pulse event.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIAPPPULSE

It is CONSTRAINED UNPREDICTABLE whether a write to CTIAPPPULSE generates an event on a channel if CTICONTROL.GLBEN is 0.

CTIAPPPULSE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x01C`	CTIAPPPULSE

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

Purpose

Sets the application triggers.

Configuration

CTIAPPSET is in the Debug power domain

External register CTIAPPSET bits [31:0] are architecturally mapped to External register CTIAPPCLEAR[31:0].

Attributes

CTIAPPSET is a 32-bit register.

Field descriptions

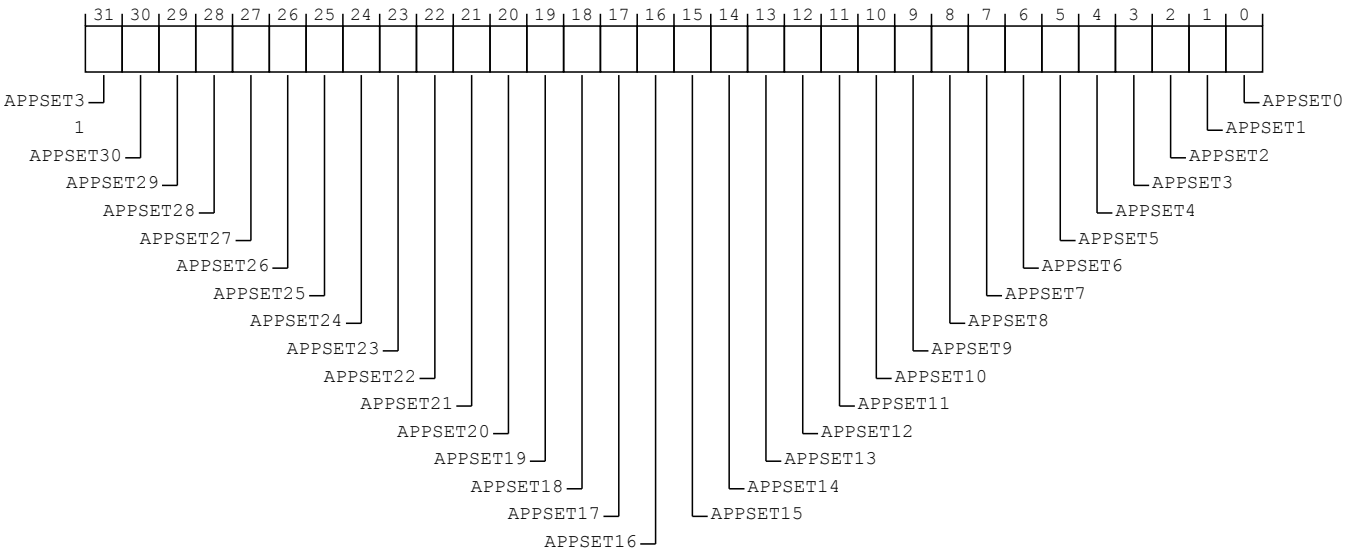

APPSET<x> , bits [x], for x = 31 to 0

Application trigger <x> enable.

APPSET<x>	Meaning
0b0	Reading this means the application trigger is inactive. Writing this has no effect.
0b1	Reading this means the application trigger is active. Writing this sets the corresponding application trigger to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the debugger must only use CTIAPPPULSE.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing CTIAPPSET

CTIAPPSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x014`	CTIAPPSET

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

Configuration

CTIAUTHSTATUS is in the Debug power domain

This register is OPTIONAL, and is required for CoreSight compliance.

Attributes

CTIAUTHSTATUS is a 32-bit register.

Field descriptions

31:28	27:24	23:16	15:12	11:8	7:4	3:2	1:0
RES0	RAZ	RES0	RAZ	RES0	RAZ	NSNID	NSID

Bits [31:28]

Reserved, RES0.

Bits [27:24]

Reserved, RAZ.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RAZ.

Bits [11:8]

Reserved, RES0.

Bits [7:4]

Reserved, RAZ.

NSNID, bits [3:2]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSNID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SNID.

NSID, bits [1:0]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAUTHSTATUS

CTIAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFB8`	CTIAUTHSTATUS

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

Purpose

Provides the raw status of the ECT channel inputs to the CTI.

Configuration

CTICHINSTATUS is in the Debug power domain

Attributes

CTICHINSTATUS is a 32-bit register.

Field descriptions

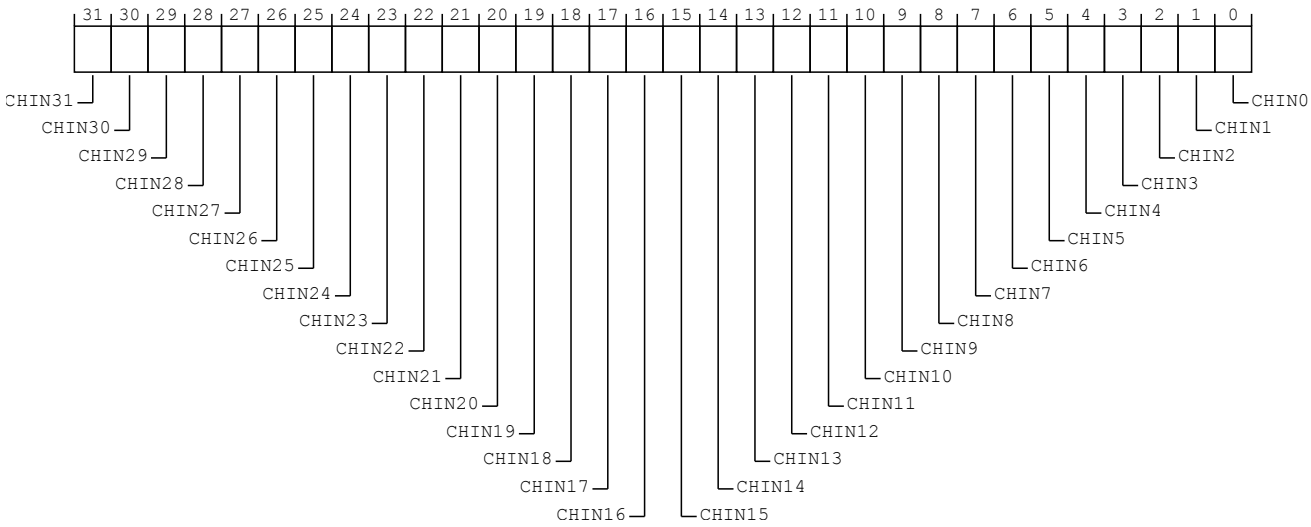

CHIN<n> , bits [n], for n = 31 to 0

Input channel <n> status.

CHIN<n>	Meaning
0b0	Input channel <n> is inactive.
0b1	Input channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an input channel can be observed as active.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICHINSTATUS

CTICHINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x138`	CTICHINSTATUS

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

Purpose

Provides the status of the ECT channel outputs from the CTI.

Configuration

CTICHOUTSTATUS is in the Debug power domain

Attributes

CTICHOUTSTATUS is a 32-bit register.

Field descriptions

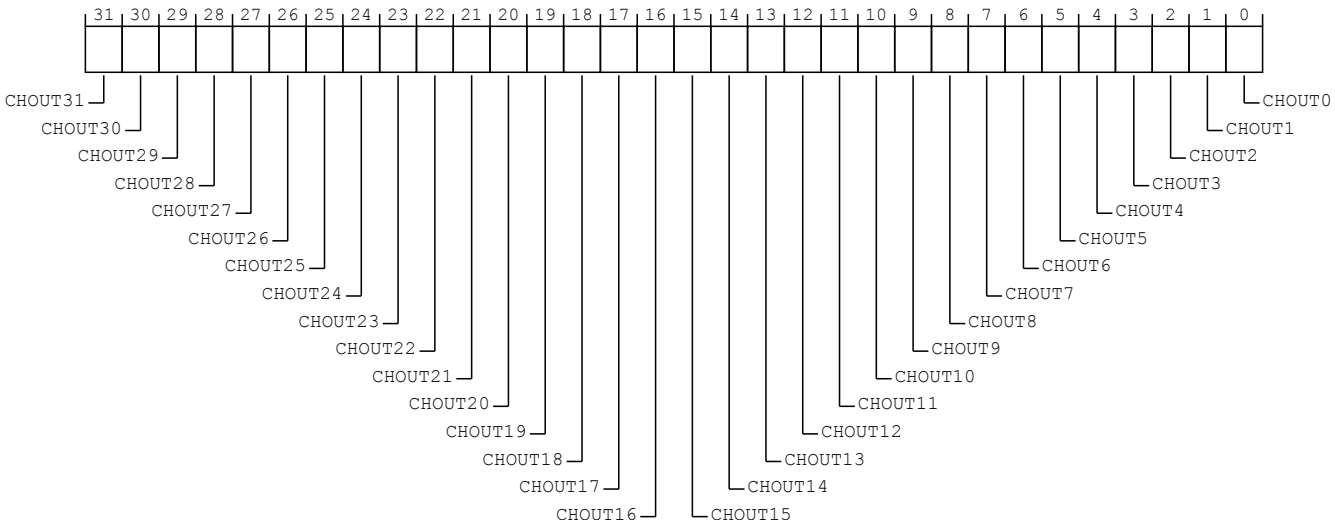

CHOUT<n> , bits [n], for n = 31 to 0

Output channel <n> status.

CHOUT<n>	Meaning
0b0	Output channel <n> is inactive.
0b1	Output channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an output channel can be observed as active.

Note

The value in CTICHOUTSTATUS is after gating by the channel gate. For more information, see CTIGATE.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICHOUTSTATUS

CTICHOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x13C`	CTICHOUTSTATUS

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.8 CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR0 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR0 is a 32-bit register.

Field descriptions

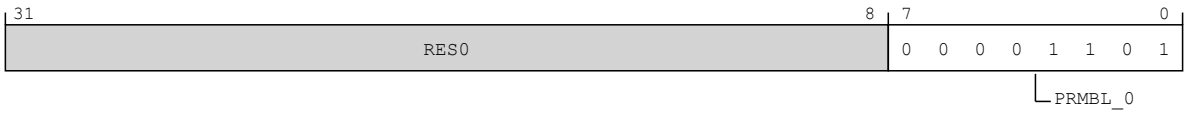

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing CTICIDR0

CTICIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF0`	CTICIDR0

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see 'About the Component Identification scheme'.

Configuration

CTICIDR1 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR1 is a 32-bit register.

Field descriptions

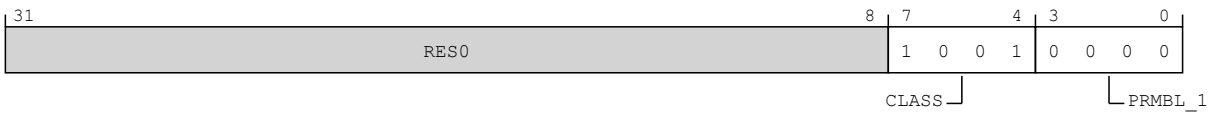

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
`0b1001`	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as `0x9`.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as `0b0000`

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICIDR1

CTICIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF4`	CTICIDR1

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR2 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR2 is a 32-bit register.

Field descriptions

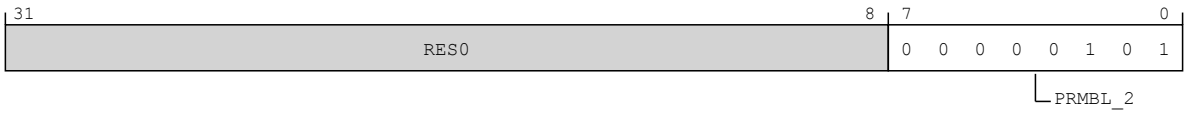

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing CTICIDR2

CTICIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF8`	CTICIDR2

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.11 CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR3 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR3 is a 32-bit register.

Field descriptions

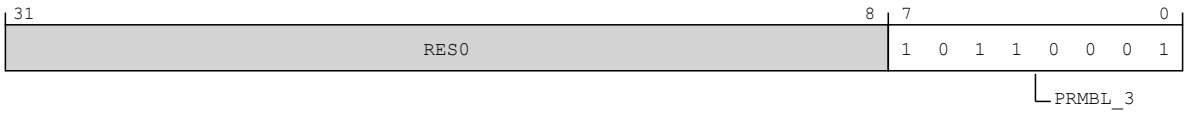

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing CTICIDR3

CTICIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFFC`	CTICIDR3

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM bits, and to clear CLAIM tag bits to 0.

Configuration

CTICLAIMCLR is in the Debug power domain

External register CTICLAIMCLR bits [31:0] are architecturally mapped to External register CTICLAIMSET[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMCLR is a 32-bit register.

Field descriptions

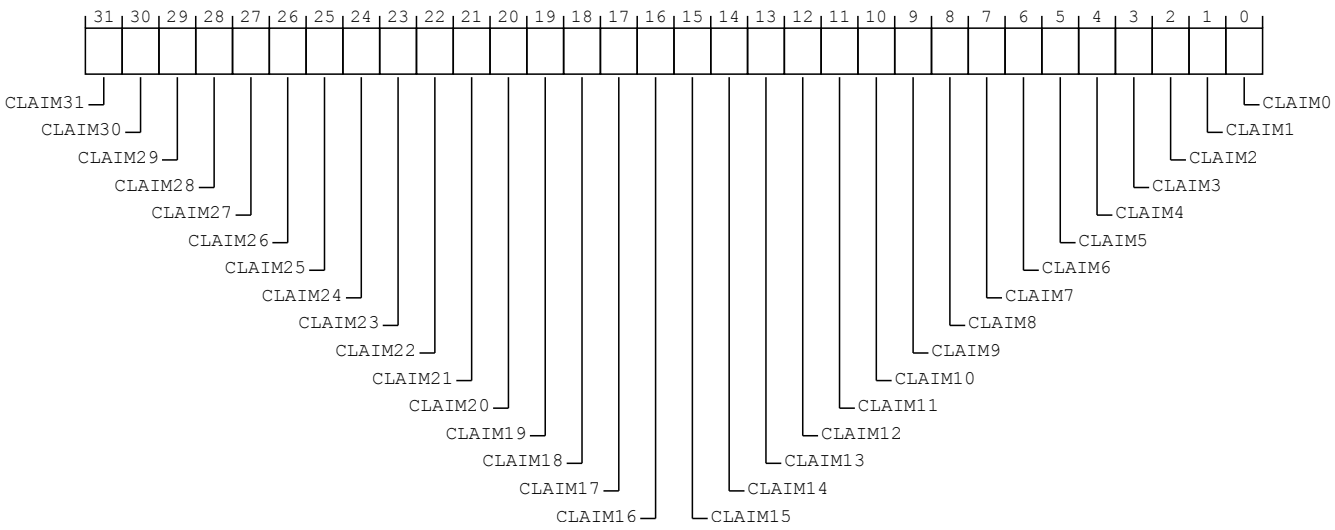

CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in CTICLAIMSET.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When m >= NUM_CTI_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing CTICLAIMCLR

CTICLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA4`	CTICLAIMCLR

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.13 CTICLAIMSET, CTI CLAIM Tag Set register

The CTICLAIMSET characteristics are:

Purpose

Used by software to set CLAIM bits to 1.

Configuration

CTICLAIMSET is in the Debug power domain

External register CTICLAIMSET bits [31:0] are architecturally mapped to External register CTICLAIMCLR[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMSET is a 32-bit register.

Field descriptions

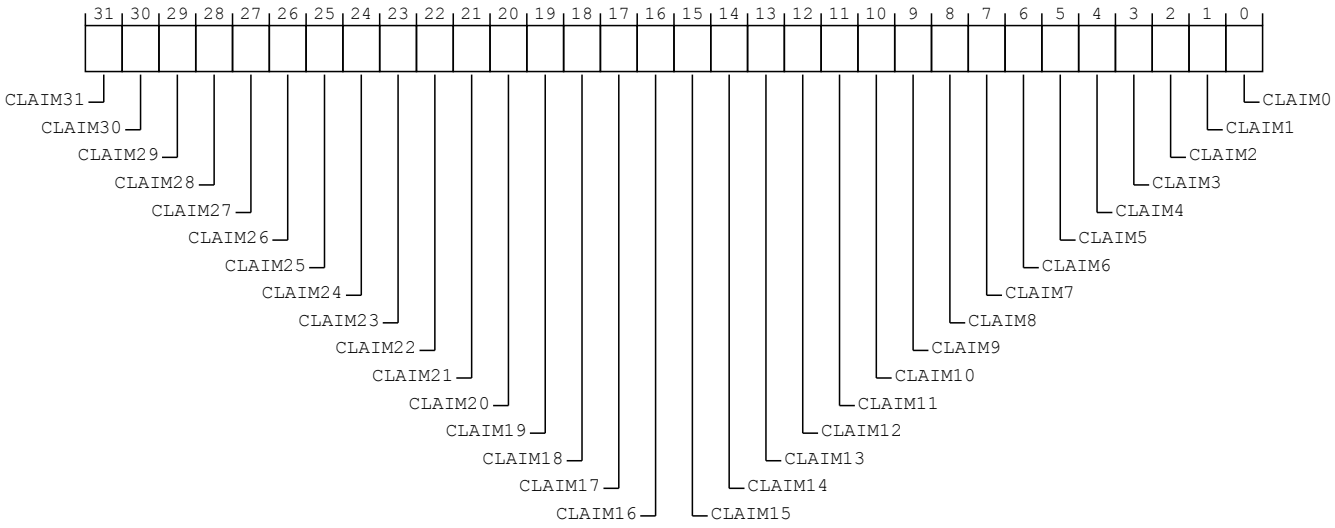

CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When m >= NUM_CTI_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/W1S.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICLAIMSET

CTICLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA0`	CTICLAIMSET

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

Purpose

Controls whether the CTI is enabled.

Configuration

CTICONTROL is in the Debug power domain

Attributes

CTICONTROL is a 32-bit register.

Field descriptions

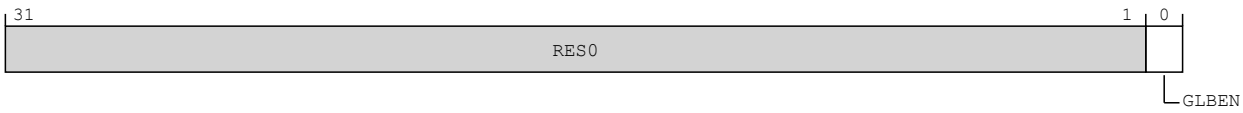

Bits [31:1]

Reserved, RES0.

GLBEN, bit [0]

Enables or disables the CTI mapping functions. Possible values of this field are:

GLBEN	Meaning
0b0	CTI mapping functions and application trigger disabled.
0b1	CTI mapping functions and application trigger enabled.

When GLBEN is 0, the input channel to output trigger, input trigger to output channel, and application trigger functions are disabled and do not signal new events on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it is CONSTRAINED UNPREDICTABLE which of the following occurs:

- The output trigger remains asserted after the mapping functions are disabled.
- The output trigger is deasserted after the mapping functions are disabled.

All output triggers are disabled by CTI reset.

If the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'0'`.

Accessing CTICONTROL

CTICONTROL can be accessed through the external debug interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	`0x000`	CTICONTROL

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF0 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF1 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF0 is a 32-bit register.

Field descriptions

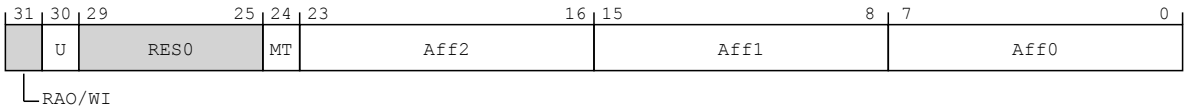

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF0

CTIDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA8`	CTIDEVAFF0

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF1 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF0 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF1 is a 32-bit register.

Field descriptions

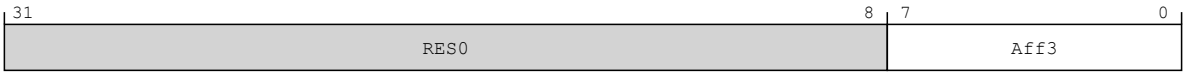

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of CTIDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF1

CTIDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFAC	CTIDEVAFF1

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the CTI component.

Configuration

CTIDEVARCH is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is not implemented, CTIDEVAFF0 and CTIDEVAFF1 are also not implemented.

Attributes

CTIDEVARCH is a 32-bit register.

Field descriptions

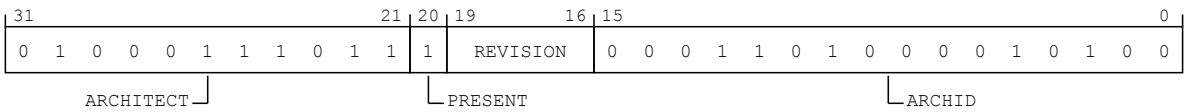

ARCHITECT, bits [31:21]

Defines the architect of the component. For CTI, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the CTIDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	First revision.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b0001`	As `0b0000`, and also adds support for CTIDEVCTL.

All other values are reserved.

When FEAT_DoPD is implemented, the value `0b0000` is not permitted.

Access to this field is RO.

ARCHID, bits [15:0]

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For CTI:

- Bits [15:12] are the architecture version, `0x1`.
- Bits [11:0] are the architecture part number, `0xA14`.

This corresponds to CTI architecture version CTIv2.

Reads as `0x1A14`

Access to this field is RO.

Accessing CTIDEVARCH

CTIDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFBC`	CTIDEVARCH

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.18 CTIDEVCTL, CTI Device Control register

The CTIDEVCTL characteristics are:

Purpose

Provides target-specific device controls

Configuration

CTIDEVCTL is in the Debug power domain

This register is present only when FEAT_DoPD is implemented. Otherwise, direct accesses to CTIDEVCTL are RES0.

Attributes

CTIDEVCTL is a 32-bit register.

Field descriptions

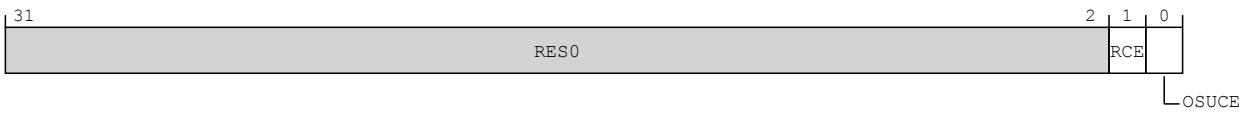

Bits [31:2]

Reserved, RES0.

RCE, bit [1]

Reset Catch Enable.

RCE	Meaning
0b0	Reset Catch debug event disabled.
0b1	Reset Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

OSUCE, bit [0]

OS Unlock Catch Enable

OSUCE	Meaning
0b0	OS Unlock Catch debug event disabled.
0b1	OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIDEVCTL

CTIDEVCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x150`	CTIDEVCTL

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.19 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

Purpose

Describes the CTI component to the debugger.

Configuration

CTIDEVID is in the Debug power domain

Attributes

CTIDEVID is a 32-bit register.

Field descriptions

31 26	25 24	23 22	21 16	15 14	13 8	7 5	4 0
RES0	INOUT	RES0	NUMCHAN	RES0	NUMTRIG	RES0	EXTMUXNUM

Bits [31:26]

Reserved, RES0.

INOUT, bits [25:24]

Input/output options. Indicates presence of the input gate. If the CTM is not implemented or CTIv2 is not implemented, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

INOUT	Meaning
0b00	CTIGATE does not mask propagation of input events from external channels.
0b01	CTIGATE masks propagation of input events from external channels.

All other values are reserved.

Access to this field is RO.

Bits [23:22]

Reserved, RES0.

NUMCHAN, bits [21:16]

Number of ECT channels implemented. For Armv8, valid values are:

- 0b000011 3 channels (0..2) implemented.
- 0b000100 4 channels (0..3) implemented.
- 0b000101 5 channels (0..4) implemented.
- 0b000110 6 channels (0..5) implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

and so on up to `0b100000`, 32 channels (0..31) implemented.

All other values are reserved.

Access to this field is RO.

Bits [15:14]

Reserved, RES0.

NUMTRIG, bits [13:8]

Upper bound for number of triggers. The indices of all implemented input and output triggers are less than this value.

This field has an IMPLEMENTATION DEFINED value.

There is no guarantee that all of the input and output triggers, including the highest numbered, are connected to any components, or that the implementation of input and output triggers is symmetrical.

Access to this field is RO.

Bits [7:5]

Reserved, RES0.

EXTMUXNUM, bits [4:0]

Number of multiplexors available on triggers. This value is used in conjunction with External Control register, ASICCTL.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVID

CTIDEVID can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFC8`	CTIDEVID

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.20 CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configuration

CTIDEVID1 is in the Debug power domain

Attributes

CTIDEVID1 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID1

CTIDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC4	CTIDEVID1

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.21 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configuration

CTIDEVID2 is in the Debug power domain

Attributes

CTIDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID2

CTIDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC0	CTIDEVID2

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.22 CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's cross-trigger interface.

Configuration

CTIDEVTYPE is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

CTIDEVTYPE is a 32-bit register.

Field descriptions

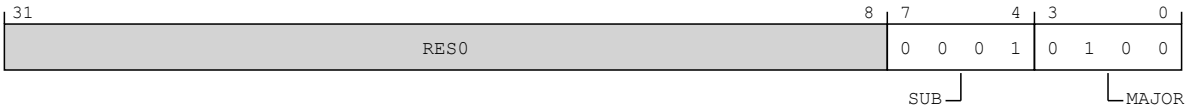

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type. Indicates this is a cross-trigger component.

Reads as `0b0100`

Access to this field is RO.

Accessing CTIDEVTYPE

CTIDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFCC`	CTIDEVTYPE

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.23 CTIGATE, CTI Channel Gate Enable register

The CTIGATE characteristics are:

Purpose

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

Configuration

CTIGATE is in the Debug power domain

Attributes

CTIGATE is a 32-bit register.

Field descriptions

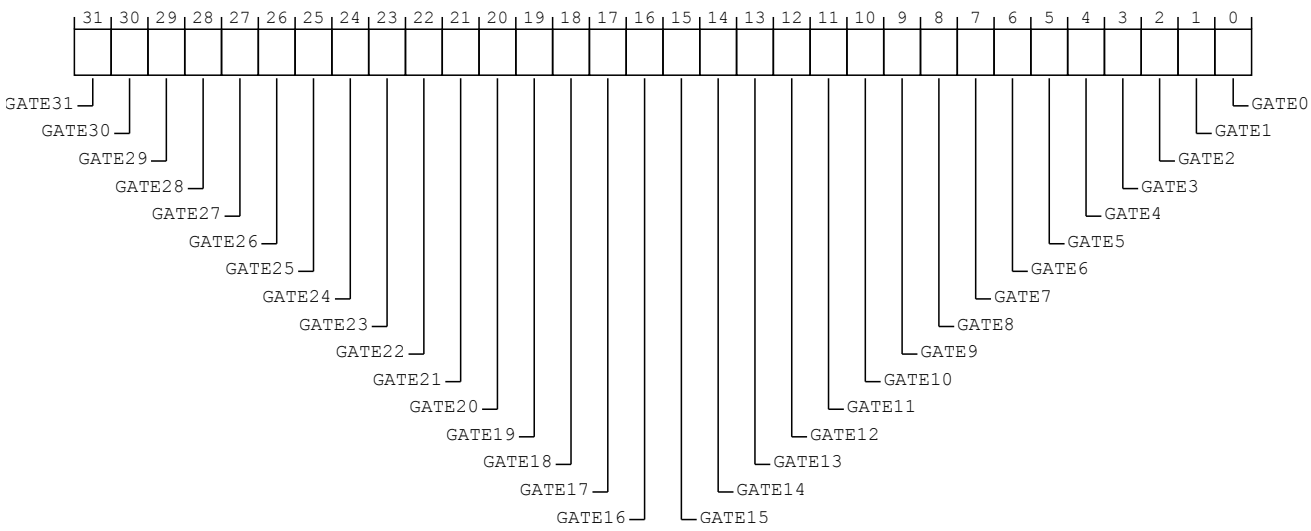

GATE<x> , bits [x], for x = 31 to 0

Channel <x> gate enable.

GATE<x>	Meaning
`0b0`	Disable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.
`0b1`	Enable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.

If GATE<x> is set to 0, no new events will be propagated to the ECT, and if the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIGATE

CTIGATE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x140`	CTIGATE

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.24 CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

Purpose

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

Configuration

CTIINEN<n> is in the Debug power domain

If input trigger n is not implemented or not connected, CTIINEN<n> is RES0.

Attributes

CTIINEN<n> is a 32-bit register.

Field descriptions

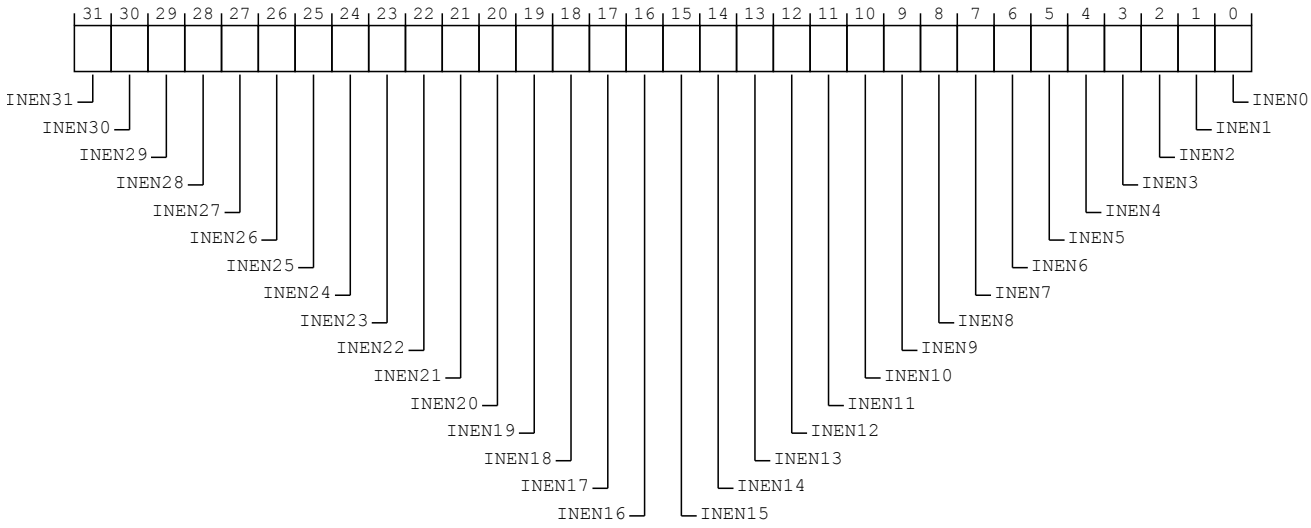

INEN<x> , bits [x], for x = 31 to 0

Input trigger <n> to output channel <x> enable.

INEN<x>	Meaning
0b0	Input trigger <n> will not generate an event on output channel <x>.
0b1	Input trigger <n> will generate an event on output channel <x>.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIINEN<n>

CTIINEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x020 + (4 * n)`	CTIINEN<n>

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.25 CTIINTACK, CTI Output Trigger Acknowledge register

The CTIINTACK characteristics are:

Purpose

Can be used to deactivate the output triggers.

Configuration

CTIINTACK is in the Debug power domain

Attributes

CTIINTACK is a 32-bit register.

Field descriptions

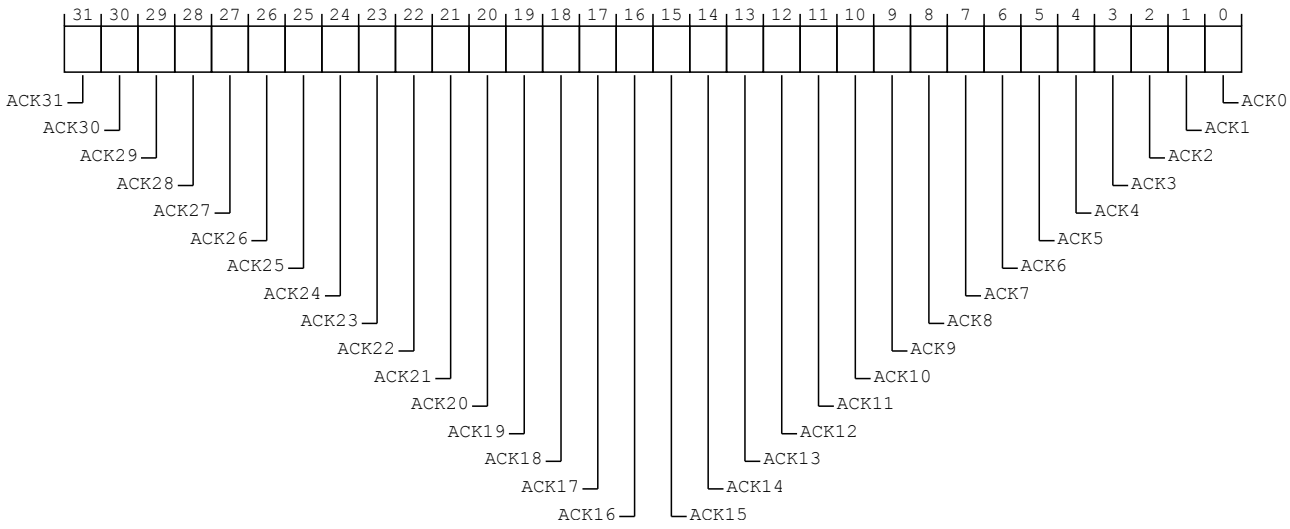

ACK<n> , bits [n], for n = 31 to 0

Acknowledge for output trigger <n>.

If any of the following is true, writes to ACK<n> are ignored:

- n >= CTIDEVID.NUMTRIG, the number of implemented triggers.
- Output trigger n is not active.
- The channel mapping function output, as controlled by CTIOUTEN<n>, is still active.

Otherwise, if any of the following are true, ACK<n> is RES0:

- Output trigger n is not implemented.
- Output trigger n is not connected.
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ACK<n>	Meaning
0b0	No effect
0b1	Deactivate the trigger.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIINTACK

A debugger must read CTITRIGOUTSTATUS to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

CTIINTACK can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x010	CTIINTACK

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.26 CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

Purpose

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

The power domain of CTIITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

CTIITCTRL is a 32-bit register.

Field descriptions

31 … 1	0
RES0	IME

Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to 0.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to 0.
  - On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIITCTRL

CTIITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xF00`	CTIITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.27 CTILAR, CTI Lock Access Register

The CTILAR characteristics are:

Purpose

Allows or disallows access to the CTI registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILAR is in the Debug power domain

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILAR is a 32-bit register.

Field descriptions

When CTI Software Lock is implemented:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Accessing CTILAR

CTILAR can be accessed through the memory-mapped interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	`0xFB0`	CTILAR

Accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.28 CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

Purpose

Indicates the current status of the Software Lock for CTI registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILSR is in the Debug power domain

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILSR is a 32-bit register.

Field descriptions

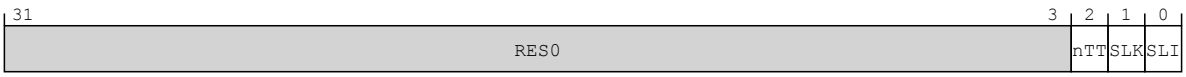

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When CTI Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.
0b1	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to '1'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
`0b0`	Software Lock not implemented or not memory-mapped access.
`0b1`	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing CTILSR

CTILSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
CTI	`0xFB4`	CTILSR

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.29 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

Purpose

Defines which input channels generate output trigger n.

Configuration

CTIOUTEN<n> is in the Debug power domain

If output trigger n is not implemented or not connected, CTIOUTEN<n> is RES0.

Attributes

CTIOUTEN<n> is a 32-bit register.

Field descriptions

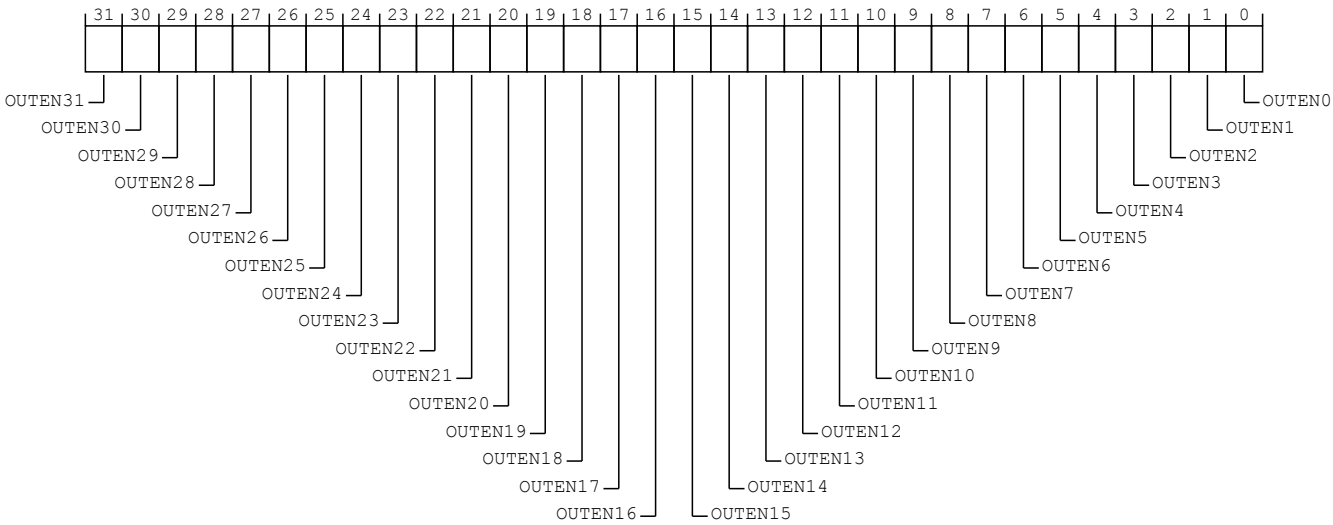

OUTEN<x> , bits [x], for x = 31 to 0

Input channel <x> to output trigger <n> enable.

OUTEN<x>	Meaning
0b0	An event on input channel <x> will not cause output trigger <n> to be asserted.
0b1	An event on input channel <x> will cause output trigger <n> to be asserted.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIOUTEN<n>

CTIOUTEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x0A0 + (4 * n)`	CTIOUTEN<n>

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.30 CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR0 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - CTIPIDR1.PART_1 contains part number bits [11:8].
  - CTIPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - CTIPIDR1.PART_1 contains part number bits [15:12].
  - CTIPIDR0.PART_0 contains part number bits [11:4].
  - CTIPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, CTIPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR0

CTIPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE0`	CTIPIDR0

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.31 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR1 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- CTIPIDR1.DES_0: JEP106 identification code bits[3:0].
- CTIPIDR2.DES_1: JEP106 identification code bits[6:4].
- CTIPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - CTIPIDR1.PART_1 contains part number bits [11:8].
  - CTIPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - CTIPIDR1.PART_1 contains part number bits [15:12].
  - CTIPIDR0.PART_0 contains part number bits [11:4].
  - CTIPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, CTIPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR1

CTIPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE4`	CTIPIDR1

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.32 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR2 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR2 is a 32-bit register.

Field descriptions

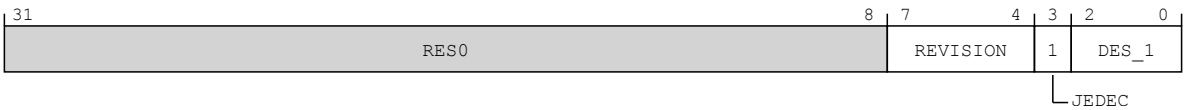

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- CTIPIDR2.REVISION indicates the 4-bit revision number.
- CTIPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- CTIPIDR2.REVISION indicates the 4-bit major revision number.
- CTIPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- CTIPIDR2.REVISION contains part number bits [3:0].
- CTIPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- CTIPIDR1.DES_0: JEP106 identification code bits[3:0].
- CTIPIDR2.DES_1: JEP106 identification code bits[6:4].
- CTIPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing CTIPIDR2

CTIPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE8`	CTIPIDR2

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.33 CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR3 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- CTIPIDR2.REVISION indicates the 4-bit revision number.
- CTIPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- CTIPIDR2.REVISION indicates the 4-bit major revision number.
- CTIPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- CTIPIDR2.REVISION contains part number bits [3:0].
- CTIPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

CTIPIDR3.CMOD might also indicate component modifications.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

CTIPIDR3.REVAND might also indicate component modifications.

Access to this field is RO.

Accessing CTIPIDR3

CTIPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFEC`	CTIPIDR3

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.34 CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR4 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- CTIPIDR1.DES_0: JEP106 identification code bits[3:0].
- CTIPIDR2.DES_1: JEP106 identification code bits[6:4].
- CTIPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing CTIPIDR4

CTIPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFD0`	CTIPIDR4

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.35 CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

Purpose

Provides the status of the trigger inputs.

Configuration

CTITRIGINSTATUS is in the Debug power domain

Attributes

CTITRIGINSTATUS is a 32-bit register.

Field descriptions

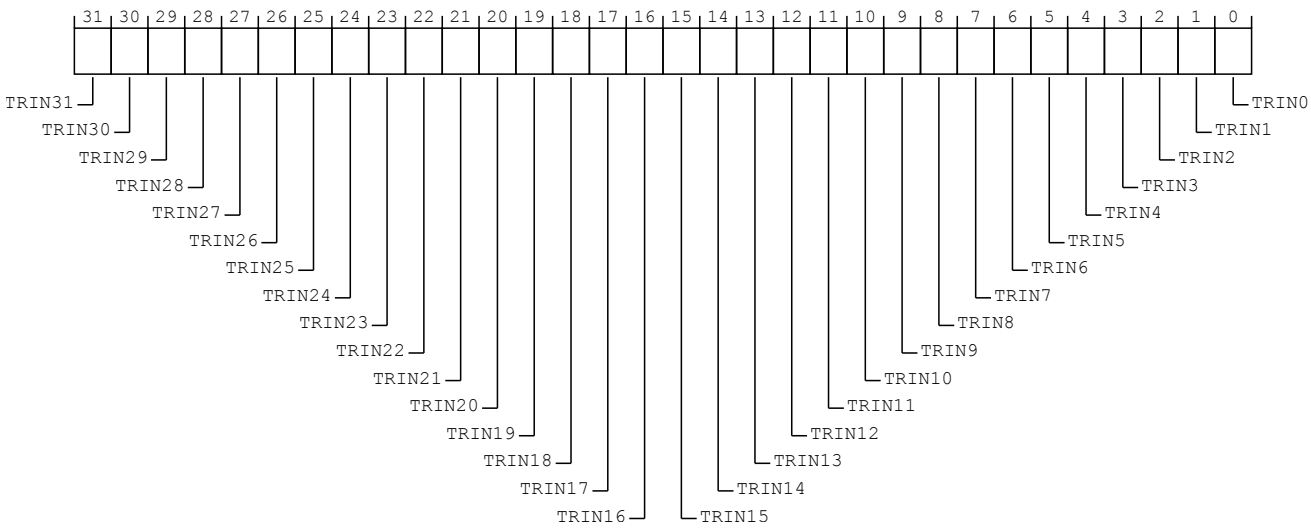

TRIN<n> , bits [n], for n = 31 to 0

Trigger input <n> status.

TRIN<n>	Meaning
0b0	Input trigger n is inactive.
0b1	Input trigger n is active.

Not implemented and not-connected input triggers are always inactive.

It is IMPLEMENTATION DEFINED whether an input trigger that does not support multicycle events can be observed as active.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTITRIGINSTATUS

CTITRIGINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x130`	CTITRIGINSTATUS

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.36 CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

Purpose

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicycle channel events.

Configuration

CTITRIGOUTSTATUS is in the Debug power domain

Attributes

CTITRIGOUTSTATUS is a 32-bit register.

Field descriptions

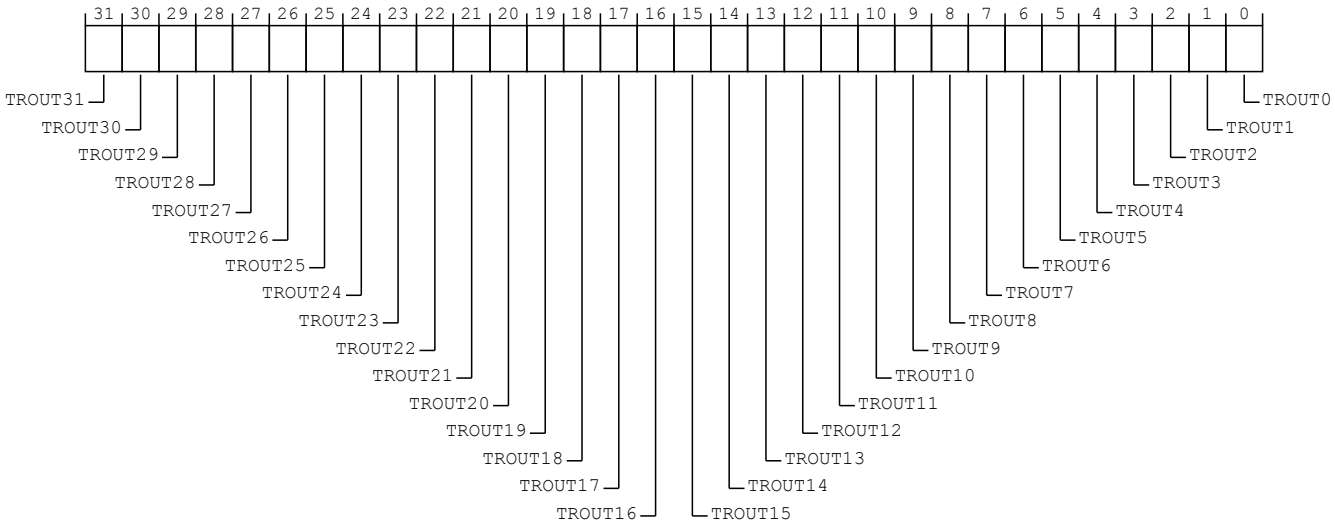

TROUT<n> , bits [n], for n = 31 to 0

Trigger output <n> status.

If n is less than CTIDEVID.NUMTRIG, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicycle channel events, then permitted values for TROUT<n> are:

TROUT<n>	Meaning
0b0	Output trigger n is inactive.
0b1	Output trigger n is active.

Otherwise when n is less than CTIDEVID.NUMTRIG, it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

Accessing this field has the following behavior:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - Any of the following are true:
    - Output trigger <n> is not implemented or not connected
    - All of the following are true:
      - Output trigger <n> is self-acknowledging
      - CTI does not implement multicycle channel events
  - an implementation implements TROUT<n>
- Otherwise, access to this field is RO.

Accessing CTITRIGOUTSTATUS

CTITRIGOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x134`	CTITRIGOUTSTATUS

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part I
Memory-mapped Components of the Arm Architecture

Chapter I1
Requirements for Memory-mapped Components

This chapter provides some additional information about memory-mapped components. It contains the following sections:

- Supported access sizes.
- Synchronization of memory-mapped registers.
- Access requirements for reserved and unallocated registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I1.1 Supported access sizes

The information in this section applies to all accesses to memory-mapped components of the Armv8 and later architectures, unless a register or component description explicitly states otherwise.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

An implementation of a memory-mapped component that is compatible with the Armv8 and later architectures must support the following:

- Word-aligned 32-bit accesses to access 32-bit registers.
- If the system includes any component with direct memory access to the memory-mapped component which needs to access the component but does not support making 64-bit accesses, word-aligned 32-bit accesses to either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations. This includes, but is not limited to:
  - A PE that implements a 32-bit ISA that does not include 64-bit memory operations. For example, Armv8-M T32.
  - A Debug Access Port that cannot make 64-bit accesses.
  - A PE that supports AArch32 at any Exception level.

  Arm deprecates support for 32-bit accesses to either half of 64-bit registers.

  Note

  Although AArch32 implementations might make 64-bit accesses for `LDP` and `STP` instructions, this is not architecturally required. To guarantee ordering of accesses, portable AArch32 software should use `LDR` and `STR`. For compatibility with such software, the memory-mapped component should treat the PE that supports AArch32 as not making 64-bit accesses using portable code.

  Note

  Some memory-mapped components of the Arm architecture require support for word-aligned 32-bit accesses to either half of a 64-bit memory mapped register even if all components with direct memory access to the memory-mapped component support making 64-bit accesses. These include:

  - The memory-mapped interface to the external debug and CTI registers that are described in External Debug Register Descriptions.
  - The memory-mapped interfaces to the Generic Timer registers that are described in System Level Implementation of the Generic Timer.
  - When FEAT_PMUv3_EXT32 is implemented, the memory-mapped interfaces to the Performance Monitors registers that are described in Recommended External Interface to the Performance Monitors.
  - When FEAT_AMU_EXT32 is implemented, the memory-mapped interfaces to the Activity Monitors registers that are described in Recommended External Interface to the Activity Monitors.

- Doubleword-aligned 64-bit accesses to access 64-bit registers that are mapped to a doubleword-aligned pair of adjacent 32-bit locations.

Unless otherwise specified, all registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte accesses.
- Halfword accesses.
- Unaligned word accesses. These accesses are not word single-copy atomic.
- Unaligned doubleword accesses. These accesses are not doubleword single-copy atomic.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair that forms a 64-bit register.
- Quadword accesses or higher accesses.
- Exclusive accesses.

For unsupported accesses, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether:

- The access generates an External abort or not.
- The defined side-effects of a read occur or not. A read returns an IMPLEMENTATION DEFINED value that might be UNKNOWN.
- A write is ignored or sets the accessed register or registers to UNKNOWN values.
- The access generates a fault handling interrupt or not.

Such accesses are often due to software faults. That is, software has accessed a location in a way that the component does not support. See also *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100) for recommended behavior on accesses that constitute software faults.

For memory-mapped accesses from a PE that complies with an Arm architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory that is accessed, determine the size of the access that is made by an instruction. Example I1-1 shows this.

Example I1-1 Access sizes for memory-mapped accesses

Two Load Doubleword instructions that are made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

The Arm architecture does not require the size of each element that is accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Unless otherwise specified by the component, any access to a memory-mapped component of the Arm architecture is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory type, and only single register load and store instructions, to create memory accesses that are supported by the peripheral. For more information, see Memory types and attributes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I1.2 Synchronization of memory-mapped registers

This section describes the synchronization requirements for the memory-mapped accesses to System registers.

This section refers to accesses to external system control registers as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers and Synchronization of changes to AArch32 System registers define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. Unless otherwise stated, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible as external system control registers.

Unless required to be observable to all observers in finite time, without explicit synchronization, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a Context synchronization event.
- The Context synchronization event was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by:

- Synchronization requirements for AArch64 System registers.
- Synchronization of changes to AArch32 System registers.

The external agent must be able to guarantee completion of a write. For example, the agent can:

- Mark the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- If the register is read/write and reads are not destructive, read back the value written.
- Use some guaranteed property of the connection between the PE and the external agent.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the Context synchronization event, and relying on the memory ordering rules provided by the memory model.

External reads and external write complete in the order in which they arrive at the PE. For accesses to different register locations, the external agent must create this order. The agent can:

- Mark the memory as Device-nGnRnE or Device-nGnRE.
- Use the appropriate memory barriers.
- Rely on some guaranteed property of the connection between the PE and the external agent.

However, the external agent cannot force the synchronization of completed writes.

In a simple sequential execution, an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For example, some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means the simultaneous writes must be merged.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I1.3 Access requirements for reserved and unallocated registers

This section describes the access requirements for reserved and unallocated memory-mapped components.

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
  - OPTIONAL features that are not implemented.
  - Breakpoints and watchpoints that are not implemented.
  - Performance Monitors counters that are not implemented.
  - CTI triggers that are not implemented.
  - Error records that are not implemented.

  These accesses are reserved.
- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses are normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I2
System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:

- About the Generic Timer specification.
- Memory-mapped counter module.
- Memory-mapped timer components.

Note

- Generic Timer memory-mapped register descriptions describes the System level Generic Timer registers. These registers are memory-mapped.
- Additional Information for Implementations of the Generic Timer gives additional information, that does not form part of the architectural definition of a system level implementation of the Generic Timer.
- The Generic Timer in AArch64 state gives a general description of the AArch64 state view of the Generic Timer, and describes the AArch64 System register interface to the Generic Timer.
- The Generic Timer in AArch32 state gives a general description of the AArch32 state view of the Generic Timer, and describes the AArch32 System register interface to the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.1 About the Generic Timer specification

The Generic Timer in AArch64 state describes the Arm Generic Timer and its implementation as seen from AArch64 state. The Generic Timer in AArch32 state describes the Arm Generic Timer and its implementation as seen from AArch32 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the Arm Generic Timer architecture also defines a memory-mapped component, that comprises:

- A memory-mapped counter module, that controls the generation of the Count value used by the Generic Timer. This memory-mapped counter module is required in any Arm Generic Timer implementation that requires software control of the Count value of the Generic Timer.
- Optional memory-mapped timer modules. These give a standardized way of providing timers for programmable system components other than PEs that implement the Arm architecture.

The full set of Generic Timer components summarizes these components as seen from AArch64 state, and The full set of Generic Timer components summarizes them as seen from AArch32 state. The system level components of the Generic Timer summarizes the system level components.

I2.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into *frames*. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame are IMPLEMENTATION DEFINED.

Note

The final 12 words of the first or only 4KB block of a register memory frame is an ID block.

Each frame must be in its own memory page, or memory protection region, and must be aligned to the size of the translation granule or protection granule.

Note

When a system level implementation of the Generic Timer is accessed by a PE:

- Using a VMSA, each frame is in its own memory page, aligned to the size of the translation granule.
- Using a PMSA, each frame is in its own memory protection region, aligned to the size of the memory protection granule.

The following sections give more information about the requirements for the system level Generic Timer component:

- Endianness and supported access sizes.
- Power and reset domains for the system level implementation of the Generic Timer.

I2.1.1.1 Endianness and supported access sizes

All memory-mapped peripherals defined in the Arm architecture must be little-endian. This means the system-level Generic Timer registers, and the register frames, are little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the memory-mapped Generic Timer registers implementations must:

- Comply with the requirements of Supported access sizes.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the Generic Timer support making 64-bit accesses.

I2.1.1.2 Power and reset domains for the system level implementation of the Generic Timer

The power and reset domains of the system level implementation of the Generic Timer are IMPLEMENTATION DEFINED as part of the system implementation. In register descriptions, they are called Timer resets to indicate they can be outside the PE power and reset domains defined by the remainder of this manual.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Arm architecture requires that the CNTCR.{FCREQ, EN} and CNTSR.FCACK fields reset to 0. These Timer reset values apply only on powerup of the power domain in which the registers are implemented or a reset of the reset domain in which they are implemented.

Every other register, or register field, of a system level implementation of the Generic Timer resets to a value that is architecturally UNKNOWN if it has a meaningful reset value. These Timer resets apply on powerup of the power domain in which the register is implemented, and on a reset of the reset domain in which it is implemented.

I2.1.2 The system level components of the Generic Timer

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

Memory-mapped counter module describes this component.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

The CNTCTLBase frame describes this frame.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

The CNTBaseN and CNTEL0BaseN frames describes these frames.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The CNTControlBase frame holds the registers for the memory-mapped counter, and provides:

- An RW control register, CNTCR, that provides:
  - An enable bit for the system counter.
  - An enable bit for Halt-on-debug. For more information, see Halt-on-debug.
  - A field that can be written to request a change to the Counter resolution. This mechanism means that, for example, if the Counter resolution is halved then the increment to the counter at each tick of the clock is doubled.
    For more information, see Control of counter resolution and increment.

  Writes to this register are rare.
- A RO status register, CNTSR, that provides:
  - A bit that indicates whether the system counter is halted because of an asserted Halt-on-debug signal.
  - A field that indicates whether a requested change to the Counter resolution has completed.
- Two contiguous 32-bit RW registers that hold the current system counter value, CNTCV. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

  The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.

  Writes to these registers are rare.
- A *Frequency modes table* of one or more 32-bit entries, where:
  - The first entry in the table defines the *base frequency* of the system counter. It is IMPLEMENTATION DEFINED whether the base frequency indicates the effective frequency of the counter, or the maximum Counter resolution.
    Arm recommends that the base frequency indicates the maximum Counter resolution.
  - Each subsequent entry in the table defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

  A 32-bit zero entry immediately follows the last table entry.

  This table can be RO or RW. For more information, see The Frequency modes table.

In an implementation with multiple physical address spaces, CNTControlBase is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, CNTControlBase is only accessible by Secure accesses. For example, in a system with FEAT_RME, CNTControlBase is only accessible by Root accesses.

In addition, the CNTReadBase frame includes a read-only copy of the system counter value, CNTCV, as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

Counter module control and status register summary describes CNTReadBase and CNTControlBase memory maps, and the registers in each frame.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.2.1 Control of counter resolution and increment

The system counter has a fixed *base frequency*, and must maintain the required counter accuracy, meaning Arm recommends that it does not gain or lose more than ten seconds in a 24-hour period, see The system counter. However, the counter can operate at a lower resolution than the base frequency, using a correspondingly larger increment. For example, it can increment by a value four times larger when running at a quarter of the base frequency. Any lower resolution operation, and any switching between resolutions, must not reduce the accuracy of the counter.

Control of the system Counter resolution and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- The Frequency modes table.
- Changing the system counter and increment.

I2.2.1.1 The Frequency modes table

The Frequency modes table starts at offset `0x20` in the CNTControlBase frame.

Table entries are 32-bits, and each entry specifies a selected frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system counter is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

Note

Where the Effective frequency and the base frequency are not the same, the increment is not solely defined by the ratio of the base frequency and the selected frequency. For example, a counter with an Effective frequency of 1GHz and a base frequency of 125MHz will increment by a value of 8 on each tick of the base frequency. If the Frequency modes table provides an additional selected frequency of 31.25MHz, then when this frequency is selected the counter increments by 32 on each tick of the selected frequency.

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the Frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the CNTControlBase memory map shows the table region as RO or RW.

Arm strongly recommends that the Frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED, up to this limit.

Note

- Arm considers it likely that implementations will require significantly fewer entries than the architectural limit.
- In the CNTControlBase frame, the offset range `0x0C0-0x0FC` can be used for IMPLEMENTATION DEFINED registers. If any registers are defined in this space, then the Frequency modes table cannot extend beyond offset `0x0B8`, with a zero word at offset `0x0BC`. This means that if any IMPLEMENTATION DEFINED registers are defined the maximum number of entries in the table is 40, including the zero-word end marker.

I2.2.1.2 Changing the system counter and increment

The value of the CNTCR.FCREQ field specifies which entry in the Frequency modes table specifies the selected frequency.

Changing the value of CNTCR.FCREQ requests a change to the selected frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

- When changing from a higher frequency to a lower frequency, the counter:
  1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. Switches to operating at the lower frequency.

- When changing from a lower frequency to a higher frequency, the counter:
  1. Waits until the end of the current lower-frequency cycle.
  2. Makes the counter increment required for operation at that lower frequency.
  3. Switches to operating at the higher frequency.

When the frequency has changed, CNTSR is updated to indicate the new selected frequency. Therefore, a system component that is waiting for a frequency change can poll CNTSR to detect the change.

I2.2.2 Halt-on-debug

The CNTCR register provides an enable bit for an OPTIONAL Halt-on-debug signal.

When the CNTCR.HDBG bit is set to 1, and the Halt-on-debug signal is implemented and asserted, the system counter is halted. Otherwise, the system counter ignores the state of this signal.

Where the system counter implements a Halt-on-debug signal and the system supports halting the system counter, Arm recommends that the Halt-on-debug signal can be controlled by a debugger using the Embedded Cross-Trigger (ECT) using a system-level cross-trigger interface that includes:

- A debug request output trigger event that asserts the Halt-on-debug signal.
- A restart request output trigger event that deasserts the Halt-on-debug signal.

For more information, see About the Embedded Cross-Trigger.

Note

Software must use the Halt-on-debug enable bit to ensure that the timers cannot be halted maliciously in an attempt to prohibit progress.

For more information about Halt-on-debug, contact Arm.

I2.2.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory *frames*:

- A control frame, with base address CNTControlBase.
- A status frame, with base address CNTReadBase.

Each of these register memory frames is in its own memory page or memory protection region, and the frame base address points to the start of this region. Each base address must be aligned to the size of the translation granule or protection granule.

Note

Each frame of a memory-mapped Generic Timer takes the name of its base address.

In each register memory frame, the memory at offset `0xFD0-0xFFF` is reserved for twelve 32-bit IMPLEMENTATION DEFINED ID registers, see the CounterID<n> register descriptions for more information.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the counter is little-endian.

In an implementation with multiple physical address spaces, CNTControlBase is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, CNTControlBase is only accessible by Secure accesses. For example, in a system with FEAT_RME, CNTControlBase is only accessible by Root accesses.

Table I2-1 shows the CNTControlBase control registers, in order of their offsets from the CNTControlBase base address, for an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, and also has

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

fewer than 39 CNTFID<n> registers. The Frequency modes table describes how this memory map differs if more CNTFID<n> registers are implemented.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-1 CNTControlBase memory map

Offset	Name	Type	Description
0x000	CNTCR	RW	Counter Control Register.
0x004	CNTSR	RO	Counter Status Register.
0x008	CNTCV[31:0]	RW	Counter Count Value register.
0x00C	CNTCV[63:32]	RW	
0x010	CNTSCR[a]	RW	Counter Scale Register.
0x014-0x018	-	RES0	Reserved.
0x01C	CNTID[a]	RO	Counter Identification Register.
0x020	CNTFID0	RO or RW	Frequency modes table, and end marker. For more information, see The Frequency modes table.
0x020+4*n*	CNTFID<n>	RO or RW	
0x024+4*n*	-	RO or RW, RAZ	
(0x028+4*n*)-0x0BC	-	RO, RES0	Reserved.
0x0C0-0x0FC	-	IMPLEMENTATION DEFINED	Reserved for IMPLEMENTATION DEFINED registers.
0x100-0xFCC	-	RO, RES0	Reserved.
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11.

a. Implemented only if FEAT_CNTSC is implemented.

Table I2-2 shows the CNTReadBase control registers, in order of their offsets from the CNTReadBase base address. Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-2 CNTReadBase memory map

Offset	Name	Type	Description
0x000	CNTCV[31:0]	RO	Counter Count Value register
0x004	CNTCV[63:32]	RO	
0x008-0xFCC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.3 Memory-mapped timer components

This part of the Arm Generic Timer specification defines an optional memory-mapped timer component. This can be implemented as part of any programmable system component that does not incorporate a System register mapped Arm Generic Timer, to provide that system component with the timer functionality of an Arm Generic Timer.

The memory map consists of up to eight timer *frames*. The base address of a frame is CNTBaseN, where *N* numbers from 0 up to a maximum permitted value of 7.

Each CNTBaseN timer frame:

- Provides its own set of timers and associated interrupts.
- Is implemented in its own memory page or memory protection region.
- Is implemented at a base address, identified as CNTBaseN, that is aligned to the size of the translation granule or memory protection region.

For each implemented CNTBaseN frame the system can optionally provide an unprivileged view of the frame, described as the EL0 view of the frame. The base address of this second view of the CNTBaseN frame is CNTEL0BaseN.

Note

In the naming of the registers associated with a CNTBaseN or CNTEL0BaseN frame, the value of *N* is represented as <n>, for example CNTACR<n>.

If a CNTEL0BaseN frame is implemented:

- Is implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.
- All registers visible in CNTBaseN, except for CNTVOFF and CNTEL0ACR, can be visible in CNTEL0BaseN.
  - Control fields in CNTEL0ACR determine whether each register is visible.
- The offsets of all visible registers are the same as their offsets in the CNTBaseN frame.

In addition to the implemented CNTBaseN and CNTEL0BaseN frames, the system must provide a single control frame at base address CNTCTLBase. CNTCTLBase must be implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.

The system defines the position of each frame in the memory map. This means the values of each of the CNTBaseN, CNTEL0BaseN, and CNTCTLBase base addresses is IMPLEMENTATION DEFINED.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the memory-mapped timers are little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:

- The CNTCTLBase frame.
- The CNTBaseN and CNTEL0BaseN frames.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level Generic Timer implementation that provides equivalent features to a System registers Generic Timer implementation in a PE that includes all of the Exception levels.

I2.3.1 The CNTCTLBase frame

The CNTCTLBase frame contains:

- An identification register for the features of the memory-mapped counter and timer implementation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access controls for each CNTBaseN frame.
- A virtual offset register for frames that implement a virtual timer.

Table I2-3 shows the CNTCTLBase registers, in order of their offsets from the CNTCTLBase base address.

Note

CNTFRQ and CNTVOFF registers are also implemented in a System register interface to the Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-3 CNTCTLBase memory map

Offset	Register	Type	Security[a]	Description
0x000	CNTFRQ[b]	RW	Restricted	Counter Frequency register.
0x004	CNTNSAR	RW	Restricted	Counter Non-Secure Access register.
0x008	CNTTIDR	RO	No restrictions	Counter Timer ID register.
0x00C- 0x03F	-	RES0	-	Reserved.
0x040+4*N*[c]	CNTACR<n>	RW	Programmable[d]	Counter Access Control register *N*.
0x060- 0x07F	-	RES0	-	Reserved.
0x080+8*N*[c]	CNTVOFF<n>[31:0][b]	RW[e]	Programmable[d]	Virtual Offset register *N*. If the CNTBaseN frame has virtual timer capability then CNTVOFF<n> is implemented as an RW register, otherwise its location is RAZ/WI.
0x084+8*N*[c]	CNTVOFF<n>[63:32][b]	RW[e]		
0x0C0-0x0FC	-	RES0	-	Reserved.
0x100-0x7FC	-	-	-	IMPLEMENTATION DEFINED.
0x800-0xFBC	-	RES0	-	Reserved.
0xFC0-0xFCF	-	-	-	IMPLEMENTATION DEFINED.
0xFD0- 0xFFC	CounterID<n>	RO	No restrictions	Counter ID registers 0-11.

a. Access security restrictions in an implementation with multiple Security states. See the individual register descriptions and CNTNSAR for details on the restrictions. In an implementation that supports only a single Security state all registers are accessible as shown in the *Type* column.

b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

c. Implemented for each value of *N* from 0 to 7 for which a CNTBaseN frame is implemented.

d. The CNTNSAR determines the Non-secure accessibility of the CNTACR<n> and the CNTVOFF<n> in the CNTCTLBase frame. For more information, see the register descriptions.

e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time, then it can implement CNTVOFF as RAZ/WI. Otherwise CNTVOFF is an RW register, and Arm strongly recommends that the system only permits access to CNTVOFF from EL2 or higher.

I2.3.2 The CNTBaseN and CNTEL0BaseN frames

Each CNTBaseN frame, or {CNTBaseN, CNTEL0BaseN} pair of frames, provides a memory-mapped counter and timer, see:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CNTBaseN frame.
- The CNTEL0BaseN frame.
- CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames.

I2.3.2.1 The CNTBase*N* frame

Table I2-4 shows the CNTBaseN registers, in order of their offsets from the CNTBaseN base address. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not, then memory at offsets `0x030-0x03C` is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in a System register interface to the timer component of a Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-4 CNTBaseN memory map

Offset	Register	Type	Description
`0x000`	CNTPCT[31:0][a]	RO	Physical Count register.
`0x004`	CNTPCT[63:32][a]	RO	
`0x008`	CNTVCT[31:0][a]	RO	Virtual Count register.
`0x00C`	CNTVCT[63:32][a]	RO	
`0x010`	CNTFRQ[a]	RO[c]	Counter Frequency register.
`0x014`	CNTEL0ACR	RW[b]	Counter EL0 Access Control Register, optional in the CNTBaseN memory map.
`0x018`	CNTVOFF[31:0][a]	RO[c]	Virtual Offset register. If CNTVOFF in the CNTCTLBase frame is an RW register, a read of this register returns the value of that register. Otherwise is RAZ.
`0x01C`	CNTVOFF[63:32][a]	RO[c]	
`0x020`	CNTP_CVAL[31:0][a]	RW	Physical Timer CompareValue register.
`0x024`	CNTP_CVAL[63:32][a]	RW	
`0x028`	CNTP_TVAL[a]	RW	Physical TimerValue register.
`0x02C`	CNTP_CTL[a]	RW	Physical Timer Control register.
`0x030`	CNTV_CVAL[31:0][a]	RW[b]	Virtual Timer CompareValue register, optional in the CNTBaseN memory map.
`0x034`	CNTV_CVAL[63:32][a]	RW[b]	
`0x038`	CNTV_TVAL[a]	RW[b]	Virtual TimerValue register, optional in the CNTBaseN memory map.
`0x03C`	CNTV_CTL[a]	RW[b]	Virtual Timer Control register, optional in the CNTBaseN memory map.
`0x040-0xFCF`	-	RES0	Reserved.
`0xFD0-0xFFC`	CounterID<n>	RO	Counter ID registers 0-11.

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

b. Address is reserved, RAZ/WI if register not implemented.

c. The CNTCTLBase frame includes an RW view of this register.

I2.3.2.2 The CNTEL0Base*N* frame

For any value of *N*, the layout of the registers in the CNTEL0Base*N* frame is identical to the CNTBaseN frame, except that, in the CNTEL0Base*N* frame:

- CNTVOFF is never visible, and the memory at `0x018-0x01C` is RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTEL0ACR is never visible, and the memory at `0x014` is RAZ/WI.
- If implemented in the CNTBaseN frame, CNTEL0ACR controls whether CNTPCT, CNTVCT, CNTFRQ, the Physical Timer, and the Virtual Timer registers are visible in the CNTEL0BaseN frame.

  If CNTEL0ACR is not implemented then these registers are not visible in the CNTEL0BaseN frame, and their addresses in that frame are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then CNTPCT, CNTVCT, CNTVOFF, CNTP_CVAL, and CNTV_CVAL must be accessible as atomic 64-bit values.

I2.3.2.3 CNTCTLBase status and control fields for the CNTBase*N* and CNTEL0Base*N* frames

In the CNTCTLBase frame:

CNTTIDR controls:

- Whether each CNTBaseN frame is implemented.
- If a CNTBaseN frame is implemented, whether:
  - That CNTBaseN frame has virtual timer capability.
  - A corresponding CNTEL0BaseN frame is implemented.

CNTNSAR controls:

Whether all the following are accessible by Non-secure accesses and, if FEAT_RME is implemented, Root accesses and Realm accesses:

- CNTACR<n>.
- CNTVOFF<n>.
- Each implemented CNTBaseN frame, and any corresponding CNTEL0BaseN frame.

The CNTACR<n> registers control:

For each implemented CNTBaseN frame, the accessibility of the following registers in that frame:

- CNTP_CTL, CNTP_CVAL, and CNTP_TVAL.
- CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.
- CNTVOFF.
- CNTFRQ.
- CNTPCT.
- CNTVCT.

For CNTACR<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

The CNTVOFF<n> registers provide:

For each implemented CNTBaseN frame that has virtual capability, the RW copy of the CNTVOFF register for that frame.

Note

In a CNTBaseN frame that has virtual timer capability the CNTVOFF register is RO.

For CNTVOFF<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I3
Recommended External Interface to the Performance Monitors

This chapter describes the recommended external interface to the Performance Monitors. It contains the following section:

- About the external interface to the Performance Monitors registers.

Note

Performance Monitors external register descriptions describes the external view of the Performance Monitors registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I3.1 About the external interface to the Performance Monitors registers

Arm recommends that:

- An implementation provides the OPTIONAL external debug interface to the Performance Monitors registers, FEAT_PMUv3_EXT.

  Note

  A debugger can use this interface to access counters in the Performance Monitors.

- The implementation includes the OPTIONAL support for memory-mapped access to the External debug interface.

  Note

  — Software running on any PE in a system can use this interface to access counters in the Performance Monitors.

  — Privileged software should use the MMU to control access to this interface.

- The external debug interface is implemented as defined in Recommended External Debug Interface.

When FEAT_PMUv3_EXT64, the 64-bit external PMU programmers' model extension is implemented, all Performance Monitors registers are 64-bit except the 32-bit CoreSight management registers.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- Differences in the external views of the Performance Monitors registers.
- Synchronization of changes to the memory-mapped views.
- Access permissions for external views of the Performance Monitors.

In this section, unless the context explicitly indicates otherwise, any reference to *a memory-mapped view* applies equally to a register view using:

- An access through an external debug interface.
- A memory-mapped access.

I3.1.1 Endianness and supported access sizes

When an implementation supports memory-mapped access to the external debug interface the interface is accessed as a little-endian memory-mapped peripheral. External Performance Monitors registers summary gives the memory map of these registers.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the external interface to the Performance Monitors registers implementations must:

- Comply with the requirements of Supported access sizes.
- When FEAT_PMUv3_EXT32 is implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the PMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity. When FEAT_PMUv3_EXT64 is implemented, permitted doubleword-aligned 64-bit accesses are single-copy atomic at doubleword granularity.

I3.1.2 Differences in the external views of the Performance Monitors registers

An external view of the Performance Monitors registers accesses the same registers as the System registers interface described in Performance Monitors Extension registers, except that:

- The PMSELR is accessible only in the System registers interface.
- The following registers are accessible only in external views:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - PMAUTHSTATUS.
  - PMCCR.
  - PMCFGR.
  - PMCIDR0.
  - PMCIDR1.
  - PMCIDR2.
  - PMCIDR3.
  - PMDEVARCH.
  - PMDEVTYPE.
  - PMEVFILT2R<n>.
  - PMIIDR.
  - PMLAR.
  - PMLSR.
  - PMPCSCTL.
  - PMPCSR.
  - PMPIDR0.
  - PMPIDR1.
  - PMPIDR2.
  - PMPIDR3.
  - PMPIDR4.

  Performance Monitors external register descriptions describes these registers.
- The following registers are accessible only in the external view when FEAT_PMUv3_EXT64 is implemented:
  - PMCCIDSR.
  - PMCNTEN.
  - PMDEVAFF.
  - PMINTEN.
  - PMOVS.
  - PMVCIDSR.

  Performance Monitors external register descriptions describes these registers.
- The following registers are accessible only in the external view when FEAT_PMUv3_EXT32 is implemented:
  - PMCID1SR.
  - PMCID2SR.
  - PMDEVAFF0.
  - PMDEVAFF1.
  - PMVIDSR.

  Performance Monitors external register descriptions describes these registers.
- The following controls do not affect the external views:
  - PMSELR.
  - PMUSERENR.
  - HDCR.{TPM, TPMCR, HPMN}.

  Instead, see the register descriptions in External System Control Register Descriptions.
- The PMSWINC_EL0 register is OPTIONAL in the external view when FEAT_PMUv3_EXT32 is implemented. It is not implemented in the external view when FEAT_PMUv3_EXT64 is implemented.

I3.1.3 Synchronization of changes to the memory-mapped views

I_{CJJNX} Synchronization complies with Synchronization of memory-mapped registers.

In particular, if a Performance Monitor is visible in both System register and an external view, and is accessed simultaneously through these two mechanisms, the behavior must be as if the accesses occurred atomically and in any order. For more information, see Synchronization of changes to the external debug registers.

I_{YWTNH} If FEAT_PMUv3_EXTPMN is implemented, then external accesses to PMCCR.OSLO are required to be observable to the indirect reads made in determining the response to subsequent external accesses without explicit synchronization.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I3.1.4 Access permissions for external views of the Performance Monitors

IDZGKJ For more information, see External debug interface register access permissions.

IYMBZZ The following tables show the access permissions for the Performance Monitors registers in a Debug implementation for Armv8 or later architectures:

- Table I3-1 when the 64-bit external PMU programmers' model extension, FEAT_PMUv3_EXT64, is implemented.
- Table I3-2 when FEAT_PMUv3_EXT32 is implemented.

These tables use the following terms:

DLK When FEAT_DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE, accesses to some registers produce an error. Applies to both interfaces.

EPMAD When `AllowExternalPMUAccess()` == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access.

EPMSSAD If FEAT_PMUv3_SS is implemented, when `AllowExternalPMSSAccess()` == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access.

Error Indicates that the access gives an error response.

Def This shows the default access permissions, if none of the conditions in this list prevent access to the register. See ISCSWJ.

Off The Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and EDPRSR.PU will read as zero.

Note

If debug power is off, then all external debug interface accesses return an error.

OSLK When the OS Lock is locked, OSLSR_EL1.OSLK is 1, accesses to some registers produces an error.

If FEAT_PMUv3_EXTPMN is implemented, the access is a Most secure access, and the Effective value of PMCCR.OSLO is 1, then the value of OSLSR_EL1.OSLK is treated as 0 for the access. This column shows the effect of this control on accesses using the external debug interface.

SLK This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the optional Software Lock is locked. See Register access permissions for memory-mapped accesses.

For all other accesses, this column is ignored.

Note

When FEAT_PMUv3_EXT64 is implemented, the Software Lock is not implemented.

- Indicates that the control has no effect on the behavior of the access:

- If no other control affects the behavior, the Default access behavior applies.
- However, another control might determine the behavior.

Table I3-1 Access permissions for the Performance Monitors registers when FEAT_PMUv3_EXT64 is implemented

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def
0x000+8xn	PMEVCNTR<n>_EL0[a]	Core	Error	Error	Error	Error	-	RW
0x0F8	PMCCNTR_EL0	Core	Error	Error	Error	Error	-	RW
0x100	PMICNTR_EL0	Core	Error	Error	Error	Error	-	RW
0x200	PMPCSR	Core	Error	Error	Error	-	-	RO
0x208	PMVCIDSR	Core	Error	Error	Error	-	-	RO
0x220	PMPCSR	Core	Error	Error	Error	-	-	RO
0x228	PMCCIDSR	Core	Error	Error	Error	-	-	RO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def
0x400+8xn	PMEVTYPER<n>_EL0[a]	Core	Error	Error	Error	Error	-	RW
0x4F8	PMCCFILTR_EL0	Core	Error	Error	Error	Error	-	RW
0x500	PMICFILTR_EL0	Core	Error	Error	Error	Error	-	RW
0x600+8xn[b]	PMEVCNTSVR<n>_EL1[a]	Core	Error	Error	-	-	Error	RO
0x6F8[b]	PMCCNTSVR_EL1	Core	Error	Error	-	-	Error	RO
0x700[b]	PMICNTSVR_EL1	Core	Error	Error	-	-	Error	RO
0x708-0x7FC[b]	Reserved	-						
0x800+8xn	PMEVFILT2R<n>	Core	Error	Error	Error	Error	-	RW
0xC00	PMCNTENSET_EL0	Core	Error	Error	Error	Error	-	RW
0xC10	PMCNTEN	Core	Error	Error	Error	Error	-	RW
0xC20	PMCNTENCLR_EL0	Core	Error	Error	Error	Error	-	RW
0xC40	PMINTENSET_EL1	Core	Error	Error	Error	Error	-	RW
0xC50	PMINTEN	Core	Error	Error	Error	Error	-	RW
0xC60	PMINTENCLR_EL1	Core	Error	Error	Error	Error	-	RW
0xC80	PMOVSCLR_EL0	Core	Error	Error	Error	Error	-	RW
0xC90	PMOVS	Core	Error	Error	Error	Error	-	RW
0xCA0	PMZR_EL0	Core	Error	Error	Error	Error	-	WO
0xCC0	PMOVSSET_EL0	Core	Error	Error	Error	Error	-	RW
0xCE0	PMCGCR0	Core	Error	Error	Error	Error	-	RO
0xD80-0xDFC	-	-	IMPLEMENTATION DEFINED registers[c]					
0xE00	PMCFGR	Core	Error	Error	Error	Error	-	RO
0xE08	PMIIDR	Core	Error	Error	Error	Error	-	RO
0xE10	PMCR_EL0	Core	Error	Error	Error	Error	-	RW
0xE30[b]	PMSSCR_EL1	Core	Error	Error	-	-	Error	RO
0xE38-0xE3C	-	-	IMPLEMENTATION DEFINED registers[c]					
0xE44	PMMIR	Core	Error	Error	Error	Error	-	RO
0xE50	PMPCSCTL	Core	Error	Error	Error	-	-	RW
0xE58 [d]	PMCCR	Core	Error	Error	-	-	-	RW
0xE80-0xEFC	Integration registers	-	IMPLEMENTATION DEFINED registers[c]					
0xF00-0xFFC	Management registers and CoreSight compliance							

a. Implemented event counters only. *n* is the counter number.
b. If FEAT_PMUv3_SS is implemented, then 0x600 - 0x7FC are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.
c. See IMPLEMENTATION DEFINED registers.
d. Only when the access to the register is a Most secure access. Otherwise, the default access permissions to this location are RAZ/WI.

Table I3-2 Access permissions for the Performance Monitors registers when FEAT_PMUv3_EXT32 is implemented

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0x000+8xn 0x004+8xn	PMEVCNTR<n>_EL0[31:0][a] PMEVCNTR<n>_EL0[63:32][a]	Core	Error	Error	Error	Error	-	RW	RO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0x0F8 0x0FC	PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0x100 0x104	PMICNTR_EL0[31:0] PMICNTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0x200 0x204	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	-	RO	RO
0x208	PMCID1SR	Core	Error	Error	Error	-	-	RO	RO
0x20C	PMVIDSR	Core	Error	Error	Error	-	-	RO	RO
0x220 0x224	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	-	RO	RO
0x228	PMCID1SR	Core	Error	Error	Error	-	-	RO	RO
0x22C	PMCID2SR	Core	Error	Error	Error	-	-	RO	RO
0x400+4xn	PMEVTYPER<n>_EL0[a][31:0]	Core	Error	Error	Error	Error	-	RW	RO
0x47C	PMCCFILTR_EL0[31:0]	Core	Error	Error	Error	Error	-	RW	RO
0x480	PMICFILTR_EL0[31:0]	Core	Error	Error	Error	Error	-	RW	RO
0x600+8xn[b] 0x604+8xn[b]	PMEVCNTSVR<n>_EL1[31:0] PMEVCNTSVR<n>_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0x6F8[b] 0x6FC[b]	PMCCNTSVR_EL1[31:0] PMCCNTSVR_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0x700[b] 0x704[b]	PMICNTSVR_EL1[31:0] PMICNTSVR_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0x708- 0x7FC[b]	Reserved	-							
0x800+4xn	PMEVFILT2R<n>	Core	Error	Error	Error	Error	-	RW	RO
0xA00+4xn[c]	PMEVTYPER<n>_EL0 [63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xA7C[c]	PMCCFILTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xA80[c]	PMICFILTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xA84- 0xAFC[c]	Reserved	-							
0xB00- 0xBFC	-	-	IMPLEMENTATION DEFINED registers[d]						
0xC00 0xC04	PMCNTENSET_EL0[31:0] PMCNTENSET_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC20 0xC24	PMCNTENCLR_EL0[31:0] PMCNTENCLR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC40 0xC44	PMINTENSET_EL1[31:0] PMINTENSET_EL1[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC60 0xC64	PMINTENCLR_EL1[31:0] PMINTENCLR_EL1[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC80 0xC84	PMOVSCLR_EL0[31:0] PMOVSCLR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xCA0	PMSWINC_EL0	Core	Error	Error	Error	Error	-	WO	WI
0xCA0 0xCA4	PMZR_EL0[31:0] PMZR_EL0[63:32]	Core	Error	Error	Error	Error	-	WO	WI

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0xCC0 0xCC4	PMOVSSET_EL0[31:0] PMOVSSET_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xCE0	PMCGCR0	Core	Error	Error	Error	Error	-	RO	RO
0xD80- 0xDFC	-	-	IMPLEMENTATION DEFINED registers[d]						
0xE00	PMCFGR	Core	Error	Error	Error	Error	-	RO	RO
0xE04	PMCR_EL0	Core	Error	Error	Error	Error	-	RW	RO
0xE08	PMIIDR	Core	Error	Error	Error	Error	-	RO	RO
0xE20 0xE24 0xE28 0xE2C	PMCEID0 PMCEID1 PMCEID2 PMCEID3	Core	Error	Error	Error	Error	-	RO	RO
0xE30[b] 0xE34[b]	PMSSCR_EL1[31:0] PMSSCR_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0xE38- 0xE3C		-	IMPLEMENTATION DEFINED registers[d]						
0xE40 0xE44	PMMIR[31:0] PMMIR[63:32]	Core	Error	Error	Error	Error	-	RO	RO
0xE50 0xE54	PMPCSCTL[31:0] PMPCSCTL[63:32]	Core	Error	Error	Error	-	-	RW	RW
0xE58[e]	PMCCR	Core	Error	Error	-	-	-	RW	RW
0xE80- 0xEFC	Integration registers	-	IMPLEMENTATION DEFINED registers[d]						
0xF00- 0xFFC	Management registers and CoreSight compliance								

a. Implemented event counters only. *n* is the counter number.

b. If FEAT_PMUv3_SS is implemented, then 0x600 - 0x7FC are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.

c. If any of FEAT_PMUv3_TH, FEAT_PMUv3p8, or FEAT_PMUv3_SME is implemented, then 0xA00 - 0xAFC are defined or reserved for top halves of the TYPER/FILTR registers. Otherwise, these locations are IMPLEMENTATION DEFINED.

d. See IMPLEMENTATION DEFINED registers.

e. Only when the access to the register is a Most secure access. Otherwise, the default access permissions for this location are RAZ/WI.

I_{LQFWT} An external access to a register is a Most secure access, if it is one of the following:

- Root access, when FEAT_RME is implemented.
- Secure access, when FEAT_RME is not implemented and Secure state is implemented.
- Non-secure access, when both FEAT_RME and Secure state are not implemented.

I_{MWDRP} The pseudocode functions `AllowExternalPMUAccess()` and `AllowExternalPMSSAccess()` always return True for a Most secure access.

I_{SCSWJ} Table I3-3 shows how the counter range affects the behavior of permitted accesses to the event counter registers.

Table I3-3 Result of PMEVCNTR<n>_EL0 event counter external access

Counter range	External access	
	Most secure access	Not Most secure access
First range	Succeeds	Succeeds

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Counter range	External access	
	Most secure access	Not Most secure access
Second range	Succeeds	Succeeds
Third range	Succeeds	No access
Counter not implemented	No access	No access

R_{DQYGL} Where Table I3-3 shows no access for an event counter *n* due to the access not being a Most secure access, all of the following apply for permitted accesses:

- The following registers are RAZ/WI:
  - PMEVCNTR<n>_EL0.
  - PMEVTYPER<n>_EL0.
  - PMEVFILT2R<n>.
  - If FEAT_PMUv3_SS is implemented, PMEVCNTSVR<n>_EL1.
- The following register fields are RAZ/WI:
  - PMOVSCLR_EL0[n].
  - PMOVSSET_EL0[n].
  - PMCNTENSET_EL0[n].
  - PMCNTENCLR_EL0[n].
  - PMINTENSET_EL1[n].
  - PMINTENCLR_EL1[n].
  - If FEAT_PMUv3_EXT64 is implemented:
    - PMOVS[n].
    - PMCNTEN[n].
    - PMINTEN[n].
  - If FEAT_PMUv3p9 is implemented, PMZR_EL0[n].
  - If PMSWINC_EL0 is implemented in the external interface, PMSWINC_EL0[n].
- A write of 1 to PMCR_EL0.P does not reset PMEVCNTR<n>_EL0.

I_{WYJGF} When FEAT_PMUv3_EXTPMN is implemented, for an external access that is not a Most secure access, all of the following apply:

- If FEAT_PMUv3_ICNTR is implemented, then PMCFGR.N and PMCGCR0.CG0NC reads as the Effective value of PMCCR.EPMN plus one.
- If FEAT_PMUv3_ICNTR is not implemented, then PMCFGR.N reads as the Effective value of PMCCR.EPMN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I4
Recommended External Interface to the Activity Monitors

This chapter describes the optional external interface to the Activity Monitors Extension registers. It contains the following section:

- About the external interface to the Activity Monitors registers

Note

Activity Monitors external register descriptions describes the external view of the Activity Monitors Extension registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I4.1 About the external interface to the Activity Monitors Extension registers

If an implementation supports the Activity Monitors Extension, it can optionally support an external memory-mapped interface to the Activity Monitors Extension, FEAT_AMU_EXT. If the implementation supports FEAT_AMU_EXT, it may further optionally support CoreSight device registers and ID registers.

When FEAT_AMU_EXT64 is implemented, all AMU registers are 64-bit except the 32-bit CoreSight management registers, when implemented.

The base address of the memory-mapped view is aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. The address offsets for the memory-mapped view are given in Table I5-159.

When FEAT_AMU_EXT32 is implemented, AMCNTENSET0, AMCNTENSET1, AMCNTENCLR0, and AMCNTENCLR1 are all read-only views of the 32-bit AMU count enable and disable control registers.

When FEAT_AMU_EXT64 is implemented, all of the following apply:

- AMCNTENSET0 and AMCNTENSET1 are accessed as a single 64-bit register, AMCNTENSET. AMCNTENSET[31:0] is architecturally mapped to AMCNTENSET0_EL0[31:0] and AMCNTENSET[63:32] is architecturally mapped to AMCNTENSET1_EL0[31:0].
- AMCNTENCLR0 and AMCNTENCLR1 are accessed as a single 64-bit register AMCNTENCLR. AMCNTENCLR[31:0] is architecturally mapped to AMCNTENCLR0_EL0[31:0], and AMCNTENCLR[63:32] is architecturally mapped to AMCNTENCLR1_EL0[31:0].
- AMCNTEN is provided as a 64-bit register, accessing the same state as the 64-bit AMU count enable and disable control registers, AMCNTENSET and AMCNTENCLR. This means that AMCNTEN, AMCNTENSET, and AMCNTENCLR are all read-only views of the same state.
- AMDEVAFF0 and AMDEVAFF1 are accessed as a single 64-bit register, AMDEVAFF.

I4.1.1 Endianness and supported access sizes

In the external interface to the Activity Monitors, all registers are little endian.

Permitted accesses to the external interface to the Activity Monitors:

- Comply with the requirements of Supported access sizes.
- When FEAT_AMU_EXT32 is implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register, even if all components with direct memory access to the AMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity. When FEAT_AMU_EXT64 is implemented, permitted doubleword-aligned 64-bit accesses are single-copy atomic at doubleword granularity.

I4.1.2 Differences in the external views of the Activity Monitors Extension registers

The external memory-mapped interface view of the Activity Monitors Extension registers accesses the same registers as the System registers interface to the registers, except that:

- The following are accessible only in the System registers interface:
  - AMUSERENR_EL0.
  - AMEVCNTVOFF0<n>_EL2.
  - AMEVCNTVOFF1<n>_EL2.
  - AMCG1IDR_EL0.
- If implemented, the following registers are accessible only in the memory-mapped view:
  - AMIIDR.
  - AMDEVAFF.
  - AMDEVAFF0.
  - AMDEVAFF1.
  - AMDEVARCH.
  - AMDEVTYPE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— AMPIDR0.
— AMPIDR1.
— AMPIDR2.
— AMPIDR3.
— AMPIDR4.
— AMCIDR0.
— AMCIDR1.
— AMCIDR2.
— AMCIDR3.

Activity Monitors external register descriptions describes these registers.

- If FEAT_AMUv1p1 virtualization of the activity monitors is enabled, the memory-mapped view of the registers presents the physical view of the counter without any offset. Virtualization of the Activity Monitors does not affect the memory-mapped view of the registers.

Note

The memory mapped view of the activity monitors is unaffected by AMCR_EL0.CG1RZ and AMCR.CG1RZ.

I4.1.3 Access during reset and power transitions

As described in Power and reset domains, the power and reset domains of the Activity Monitoring Unit are named the AMU domain and AMU reset, and when reset of the AMU power domain occurs, the Activity Monitoring Unit is reset and the counters are reset to zero.

If the AMU domain is an always-on power domain, while the PE is reset or powered down counter values may be preserved and might be accessible by memory-mapped access.

If the AMU domain is the Core power domain, while the PE is reset or powered down and when a memory-mapped access occurs, the access reads as zero and the bus access completes without an error.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I5
External System Control Register Descriptions

This chapter describes the external system control registers. It excludes the External debug registers that are described in External Debug Register Descriptions. It contains the following sections:

- About the external system control register descriptions.
- External Performance Monitors registers summary.
- Performance Monitors external register descriptions.
- External Activity Monitors Extension registers summary.
- Activity Monitors external register descriptions.
- Generic Timer memory-mapped registers overview.
- Generic Timer memory-mapped register descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.1 About the external system control register descriptions

This chapter describes the external system control registers other than the external debug registers. That is, it describes:

An external view of the Performance Monitors registers

Arm recommends that implementations provide access to the Performance Monitors registers through the OPTIONAL External debug interface, and provide the OPTIONAL memory-mapped interface to this interface:

- External Performance Monitors registers summary lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- Performance Monitors external register descriptions describes each of the memory-mapped registers.

Recommended External Interface to the Performance Monitors describes the recommended interface to these registers.

Note

The Performance Monitors Extension describes the Performance Monitors. The following sections describe the System register interfaces to the Performance Monitors:

- Performance Monitors registers, for accesses from an Exception level that is using AArch64.
- Performance Monitors registers, for accesses from an Exception level that is using AArch32.

An external view of the Activity Monitors Extension registers

An implementation which supports the Activity Monitors Extension may support an optional external memory-mapped interface to the Activity Monitors Extension registers.

- External Activity Monitors Extension registers summary lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- Activity Monitors external register descriptions describes each of the memory-mapped registers.

Recommended External Interface to the Activity Monitors describes the recommended interface to these registers.

Note

The Activity Monitors Extension describes the Activity Monitors. The following sections describe the System register interfaces to the Activity Monitors:

- Activity Monitors registers, for accesses from an Exception level that is using AArch64.
- Activity Monitors registers, for accesses from an Exception level that is using AArch32.

The registers for the system level Generic Timer component

Any implementation that includes the Generic Timer must include the memory-mapped system level component described in System Level Implementation of the Generic Timer. In this chapter:

- Generic Timer memory-mapped registers overview gives an overview of the registers, referring to System Level Implementation of the Generic Timer for more information.
- Generic Timer memory-mapped register descriptions describes each of the memory-mapped registers.

Note

The Generic Timer in AArch64 state describes the Generic Timer component that is accessible using the System registers. The following sections describe the System register interfaces to that component:

- Generic Timer registers, for accesses from an Exception level that is using AArch64.
- Generic Timer registers, for accesses from an Exception level that is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

External Debug Register Descriptions describes the external debug registers.
Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* (ARM IHI 0099)* describes the MPAM memory-mapped registers.
Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100) describes the RAS memory-mapped registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.2 External Performance Monitors registers summary

When an implementation provides access to the Performance Monitors registers through the External debug interface, that interface provides access to:

- Performance Monitors registers.
- A read-only configuration register, PMCFGR.
- The OPTIONAL CoreSight registers for the Performance Monitors, if they are implemented.

The locations of the registers are defined as offsets from a system-defined base address. Performance Monitors external register views defines this memory map.

I5.2.1 Performance Monitors external register views

The following tables show the external view of the Performance Monitors registers:

- Table I5-1 when FEAT_PMUv3_EXT64, the 64-bit external PMU programmers' model extension is implemented.
- Table I5-2 when FEAT_PMUv3_EXT32 is implemented.

All other entries are reserved.

Note

- Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any external view.
- The registers that relate to an implemented event counter, PMN*x*, are PMEVCNTR<n> and PMEVTYPER<n>.
- Tables in this section only list the Armv8 registers. For encoding information of the registers introduced by Armv9, see the individual register descriptions in this chapter.

Each entry in the Name column links to the register description in Performance Monitors external register descriptions.

Table I5-1 Performance Monitors external register views when FEAT_PMUv3_EXT64 is implemented

Name	Type	Description	Offset
PMEVCNTR<n>_EL0	RW	Performance Monitors Event Counter Register	`0x000+8xn`
PMCCNTR_EL0	RW	Performance Monitors Cycle Counter Register	`0x0F8`
PMICNTR_EL0	RW	Performance Monitors Instruction Counter Register	`0x100`
PMPCSR[a]	RW	Program Counter Sample Register	`0x200`
PMVCIDSR[a]	RW	CONTEXTIDR_EL1 and VMID Sample Register	`0x208`
PMPCSR[a]	RW	Program Counter Sample Register, alias	`0x220`
PMCCIDSR[a]	RW	CONTEXTIDR_ELx Sample Register	`0x228`
PMEVTYPER<n>_EL0	RW	Performance Monitors Event Type and Filter Register	`0x400+8xn`
PMCCFILTR_EL0	RW	Performance Monitors Cycle Counter Filter Register	`0x4F8`
PMICFILTR_EL0	RW	Performance Monitors Instruction Counter Filter Register	`0x500`
PMEVCNTSVR<n>_EL1	RO	Performance Monitors Event Count Saved Value Register<n>	`0x600+8xn`[b]
PMCCNTSVR_EL1	RO	Performance Monitors Cycle Count Saved Value Register	`0x6F8`[b]
PMICNTSVR_EL1	RO	Performance Monitors Instruction Count Saved Value Register	`0x700`[b]
-	-	Reserved	`0x708-0x7FC`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Type	Description	Offset
PMEVFILT2R<n>	RW	Performance Monitors Event Filter Register	0800+8xn
PMCNTENSET_EL0	RW	Performance Monitors Count Enable Set Register	0xC00
PMCNTEN	RW	Performance Monitors Count Enable Register	0xC10
PMCNTENCLR_EL0	RW	Performance Monitors Count Enable Clear Register	0xC20
PMINTENSET_EL1	RW	Performance Monitors Interrupt Enable Set Register	0xC40
PMINTEN	RW	Performance Monitors Interrupt Enable Register	0xC50
PMINTENCLR_EL1	RW	Performance Monitors Interrupt Enable Clear Register	0xC60
PMOVSCLR_EL0	RW	Performance Monitors Overflow Flag Status Clear Register	0xC80
PMOVS	RW	Performance Monitors Overflow Flag Status Register	0xC90
PMZR_EL0	WO	Performance Monitors Zero with Mask	0xCA0
PMOVSSET_EL0	RW	Performance Monitors Overflow Flag Status Set Register	0xCC0
PMCGCR0	RO	Counter Group Configuration Register 0	0xCE0
-	-	IMPLEMENTATION DEFINED	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	0xE00
PMIIDR	RO	Performance Monitors Implementation Identification Register	0xE08
PMCR_EL0	RW	Performance Monitors Control Register	0xE10
PMSSCR_EL1	RO	Performance Monitors Snapshot Status and Capture Register	0xE30[b]
-	-	IMPLEMENTATION DEFINED	0xE38-0xE3C
PMMIR	RO	Performance Monitors Machine Identification Register	0xE40
PMPCSCTL	RW	PC Sample-based Profiling Control Register	0xE50
-	-	IMPLEMENTATION DEFINED	0xE80-0xEFC
PMITCTRL[c]	RW	Integration Model Control registers	0xF00
PMDEVAFF[c]	RO	Device Affinity Register	0xFA8
PMLAR[c, d]	WO	Lock Access Register	0xFB0
PMLSR[c, d]	RO	Lock Status Register	0xFB4
PMAUTHSTATUS[c]	RO	Authentication Status Register	0XFB8
PMDEVARCH[c]	RO	Device Architecture Register	0xFBC
PMDEVID[a]	RO	Performance Monitors Device ID Register	0xFC8
PMDEVTYPE[c]	RO	Device Type Register	0xFCC
PMPIDR4[c] PMPIDR0[c] PMPIDR1[c] PMPIDR2[c] PMPIDR3[c]	RO	Peripheral ID registers	0xFD0 0xFE0 0xFE4 0xFE8 0xFEC
PMCIDR0[c] PMCIDR1[c] PMCIDR2[c] PMCIDR3[c]	RO	Component ID registers	0xFF0 0xFF4 0xFF8 0xFFC

a. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2 PMDEVID is required regardless of whether FEAT_PCSRv8p2 is implemented.

Before Armv8.2. the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see The PC Sample-based Profiling Extension.

b. If FEAT_PMUv3_SS is implemented, then 0x600 - 0x7FC are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.

c. CoreSight interface registers, see Management registers and CoreSight compliance.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

Table I5-2 Performance Monitors external register views when FEAT_PMUv3_EXT32 is implemented

Name	Type	Description	Offset
PMEVCNTR<n>_EL0[31:0] PMEVCNTR<n>_EL0[63:32]	RW	Performance Monitors Event Counter Register	0x000+8xn 0x004+8xn
PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	RW	Performance Monitors Cycle Counter Register[a]	0x0F8 0x0FC
PMICNTR_EL0[31:0] PMICNTR_EL0[63:32]	RW	Performance Monitors Instruction Counter Register	0x100 0x104
PMPCSR[31:0][b] PMPCSR[63:32][b]	RW	Program Counter Sample Register	0x200 0x204
PMCID1SR[b]	RW	CONTEXTIDR_EL1 Sample Register	0x208
PMVIDSR[b]	RW	VMID Sample Register	0x20C
PMPCSR[31:0][b] PMPCSR[63:32][b]	RW	Program Counter Sample Register, alias	0x220 0x224
PMCID1SR[b]	RW	CONTEXTIDR_EL1 Sample Register (alias)	0x228
PMCID2SR[b]	RW	CONTEXTIDR_EL2 Sample Register	0x22C
PMEVTYPER<n>_EL0 [31:0]	RW	Performance Monitors Event Type and Filter Register	0x400+4xn
PMCCFILTR_EL0 [31:0]	RW	Performance Monitors Cycle Counter Filter Register	0x47C
PMICFILTR_EL0[31:0]	RW	Performance Monitors Instruction Counter Filter Register	0x480
PMEVCNTSVR<n>_EL1[31:0] PMEVCNTSVR<n>_EL1[63:32]	RO	Performance Monitors Event Count Saved Value Register	0x600+8xn[c] 0x604+8xn[c]
PMCCNTSVR_EL1[31:0] PMCCNTSVR_EL1[63:32]	RO	Performance Monitors Cycle Count Saved Value Register	0x6F8[c] 0x6FC[c]
PMICNTSVR_EL1[31:0] PMICNTSVR_EL1[63:32]	RO	Performance Monitors Instruction Count Saved Value Register	0x700[c] 0x704[c]
-	-	Reserved	0x708- 0x7FC[c]
PMEVFILT2R<n>	RW	Performance Monitors Event Filter Registers	0x800+8xn
PMEVTYPER<n>_EL0 [63:32]	RW	Performance Monitors Event Type and Filter Register	0xA00+4xn[d]
PMCCFILTR_EL0 [63:32]	RW	Performance Monitors Cycle Counter Filter Register	0xA7C[d]
PMICFILTR_EL0[63:32]	RW	Performance Monitors Instruction Counter Filter Register	0xA80[d]
-	-	Reserved	0xA84- 0xAFC[d]
-	-	IMPLEMENTATION DEFINED	0xB00-0xBFC
PMCNTENSET_EL0[31:0][e] PMCNTENSET_EL0[63:32] [h]	RW	Performance Monitors Count Enable Set Register	0xC00 0xC04
PMCNTENCLR_EL0[31:0][e] PMCNTENCLR_EL0[63:32] [e]	RW	Performance Monitors Count Enable Clear Register	0xC20 0xC24
PMINTENSET_EL1[31:0][e] PMINTENSET_EL1[63:32] [e]	RW	Performance Monitors Interrupt Enable Set Register	0xC40 0xC44

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Type	Description	Offset
PMINTENCLR_EL1[31:0][e] PMINTENCLR_EL1[63:32] [e]	RW	Performance Monitors Interrupt Enable Clear Register	0xC60 0xC64
PMOVSCLR_EL0[31:0][e] PMOVSCLR_EL0[63:32][e]	RW	Performance Monitors Overflow Flag Status Clear Register	0xC80 0xC84
PMSWINC_EL0[f]	WO	Performance Monitors Software Increment Register	0xCA0
PMZR_EL0[31:0] PMZR_EL0[63:32]	WO	Performance Monitors Zero with Mask	0xCA0 0xCA4
PMOVSSET_EL0[31:0][e] PMOVSSET_EL0[63:32][e]	RW	Performance Monitors Overflow Flag Status Set Register	0xCC0 0xCC4
PMCGCR0	RO	Counter Group Configuration Register 0	0xCE0
-	-	IMPLEMENTATION DEFINED	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	0xE00
PMCR_EL0	RW	Performance Monitors Control Register	0xE04
PMIIDR	RO	Performance Monitors Implementation Identification Register	0xE08
PMCEID0 PMCEID1 PMCEID2 PMCEID3	RO	Performance Monitors Common Event Identification Registers	0xE20 0xE24 0xE28 0xE2C
PMSSCR_EL1[31:0] PMSSCR_EL1[63:32]	RO	Performance Monitors Snapshot Status and Capture Register, bits [31:0]	0xE30[c] 0xE34[c]
-	-	IMPLEMENTATION DEFINED	0xE38-0xE3C
PMMIR[31:0][e] PMMIR[63:32][e]	RO	Performance Monitors Machine Identification Register, bits [31:0]	0xE40 0xE44
PMPCSCTL[31:0][e] PMPCSCTL[63:32][e]	RW	PC Sample-based Profiling Control Register, bits [31:0]	0xE50 0xE54
-	-	IMPLEMENTATION DEFINED	0xE80-0xEFC
PMITCTRL[g]	RW	Integration Model Control registers	0xF00
PMDEVAFF0[g] PMDEVAFF1[g]	RO	Device Affinity registers	0xFA8 0xFAC
PMLAR[g , h]	WO	Lock Access Register	0xFB0
PMLSR[g , h]	RO	Lock Status Register	0xFB4
PMAUTHSTATUS[g]	RO	Authentication Status Register	0XFB8
PMDEVARCH[g]	RO	Device Architecture Register	0xFBC
PMDEVID[b]	RO	Performance Monitors Device ID Register	0xFC8
PMDEVTYPE[g]	RO	Device Type Register	0xFCC
PMPIDR4[g] PMPIDR0[g] PMPIDR1[g] PMPIDR2[g] PMPIDR3[g]	RO	Peripheral ID registers	0xFD0 0xFE0 0xFE4 0xFE8 0xFEC
PMCIDR0[g] PMCIDR1[g] PMCIDR2[g] PMCIDR3[g]	RO	Component ID registers	0xFF0 0xFF4 0xFF8 0xFFC

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMDEVID is required regardless of whether FEAT_PCSRv8p2 is implemented.

Before Armv8.2. the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see The PC Sample-based Profiling Extension.

c. If FEAT_PMUv3_SS is implemented, then `0x600` - `0x7FC` are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.

d. If any of FEAT_PMUv3_TH, FEAT_PMUv3p8, or FEAT_PMUv3_SME is implemented, then `0xA00` - `0xAFC` are defined or reserved for top halves of the TYPER/FILTR registers. Otherwise, these locations are IMPLEMENTATION DEFINED.

e. If FEAT_PMUv3p9 is implemented, then the register is 64-bits in the external debug interface to the PMU, even if FEAT_PMUv3_EXT64 is not implemented. 64-bit accesses to the registers must be supported, however, these are not required to be atomic at more than a 32-bit granularity. When `AllowExternalPMUAccess()` == FALSE, the behavior of 32-bit accesses to the top half of these registers is well-defined.

f. OPTIONAL if FEAT_PMUv3p9 is not implemented. Not permitted if FEAT_PMUv3p9 is implemented.

g. CoreSight interface registers, see Management registers and CoreSight compliance.

h. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3 Performance Monitors external register descriptions

This section describes the external view of the Performance Monitors registers. External Performance Monitors registers summary lists these registers in offset order.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is OPTIONAL, and is required for CoreSight compliance. Arm recommends that this register is implemented.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMAUTHSTATUS are RES0.

Attributes

PMAUTHSTATUS is a 32-bit register.

Field descriptions

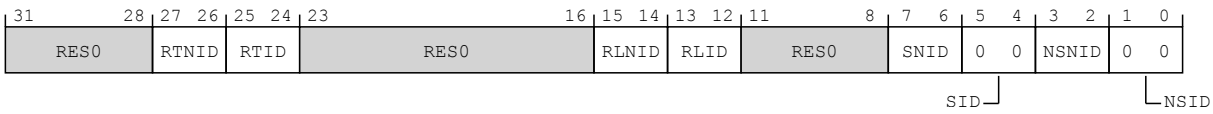

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTNID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
0b00	Not implemented.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLNID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLID, bits [13:12]

Realm invasive debug.

RLID	Meaning
`0b00`	Not implemented.

Bits [11:8]

Reserved, RES0.

SNID, bits [7:6]

Holds the same value as DBGAUTHSTATUS_EL1.SNID.

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
`0b00`	Not implemented.

All other values are reserved.

Access to this field is RO.

NSNID, bits [3:2]

Holds the same value as DBGAUTHSTATUS_EL1.NSNID.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
`0b00`	Not implemented.

All other values are reserved.

Access to this field is RO.

Accessing PMAUTHSTATUS

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFB8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configuration

PMCCFILTR_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, External register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_TH is implemented, or FEAT_PMUv3p8 is implemented, External register PMCCFILTR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCCFILTR_EL0[63:0].

External register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCCFILTR[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are RES0.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions

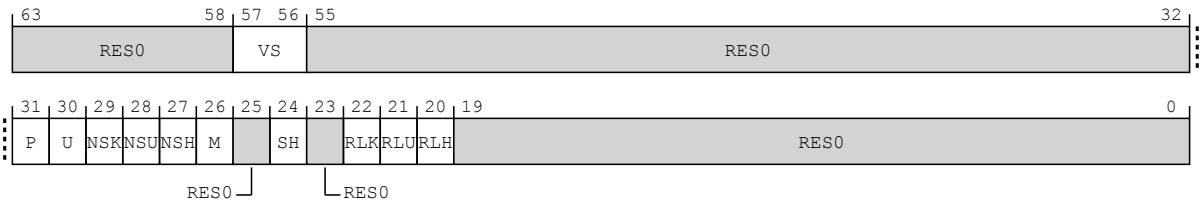

Bits [63:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting cycles in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of cycles.
0b01	The PE does not count cycles in Streaming SVE mode.
0b10	The PE does not count cycles in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting cycles in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR_EL0.NSK.

If FEAT_RME is implemented, then counting cycles in Realm EL1 is further controlled by PMCCFILTR_EL0.RLK.

If EL3 is implemented, then counting cycles at EL3 is further controlled by PMCCFILTR_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting cycles in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR_EL0.NSU.

If FEAT_RME is implemented, then counting cycles in Realm EL0 is further controlled by PMCCFILTR_EL0.RLU.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting cycles in Non-secure EL1. If PMCCFILTR_EL0.NSK is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL1.

NSK	Meaning
`0b0`	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in Non-secure EL1.
`0b1`	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in Non-secure EL1. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting cycles in Non-secure EL0. If PMCCFILTR_EL0.NSU is not equal to PMCCFILTR_EL0.U, then the PE does not count cycles in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL0.

NSU	Meaning
`0b0`	When PMCCFILTR_EL0.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.U == 1, the PE does not count cycles in Non-secure EL0.
`0b1`	When PMCCFILTR_EL0.U == 0, the PE does not count cycles in Non-secure EL0. When PMCCFILTR_EL0.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
`0b0`	The PE does not count cycles in EL2.
`0b1`	This mechanism has no effect on filtering of cycles.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting cycles in Secure EL2 is further controlled by PMCCFILTR_EL0.SH.

If FEAT_RME is implemented, then counting cycles in Realm EL2 is further controlled by PMCCFILTR_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented and FEAT_AA64 is implemented:

EL3 filtering. Controls counting cycles in EL3. If PMCCFILTR_EL0.M is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in EL3. Otherwise, this mechanism has no effect on filtering of cycles in EL3.

M	Meaning
`0b0`	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in EL3.
`0b1`	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in EL3. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting cycles in Secure EL2. If PMCCFILTR_EL0.SH is equal to PMCCFILTR_EL0.NSH, then the PE does not count cycles in Secure EL2. Otherwise, this mechanism has no effect on filtering of cycles in Secure EL2.

SH	Meaning
`0b0`	When PMCCFILTR_EL0.NSH == 0, the PE does not count cycles in Secure EL2. When PMCCFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of cycles.
`0b1`	When PMCCFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.NSH == 1, the PE does not count cycles in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting cycles in Realm EL1. If PMCCFILTR_EL0.RLK is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in Realm EL1. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLK	Meaning
0b0	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in Realm EL1.
0b1	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in Realm EL1. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting cycles in Realm EL0. If PMCCFILTR_EL0.RLU is not equal to PMCCFILTR_EL0.U, then the PE does not count cycles in Realm EL0. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL0.

RLU	Meaning
0b0	When PMCCFILTR_EL0.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.U == 1, the PE does not count cycles in Realm EL0.
0b1	When PMCCFILTR_EL0.U == 0, the PE does not count cycles in Realm EL0. When PMCCFILTR_EL0.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting cycles in Realm EL2. If PMCCFILTR_EL0.RLH is equal to PMCCFILTR_EL0.NSH, then the PE does not count cycles in Realm EL2. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLH	Meaning
0b0	When PMCCFILTR_EL0.NSH == 0, the PE does not count cycles in Realm EL2. When PMCCFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of cycles.
0b1	When PMCCFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.NSH == 1, the PE does not count cycles in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

If FEAT_PMUv3_EXT32 is implemented, and any of the following apply, then bits [63:32] of this register are accessible at offset 0xA7C:

- FEAT_PMUv3_TH is implemented.
- FEAT_PMUv3p8 is implemented.
- FEAT_PMUv3_SME is implemented.

Otherwise accesses at this offset are IMPLEMENTATION DEFINED.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0x47C from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset 0x4F8 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

When IsFeatureImplemented(FEAT_PMUv3_EXT32) && ((IsFeatureImplemented(FEAT_PMUv3_TH) || IsFeatureImplemented(FEAT_PMUv3p8)) || IsFeatureImplemented(FEAT_PMUv3_SME))

[63:32] Accessible at offset 0xA7C from PMU

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.3 PMCCIDSR, CONTEXTIDR_ELx Sample Register

The PMCCIDSR characteristics are:

Purpose

PC Sample-based Profiling Extension register that contains the sampled value of CONTEXTIDR_EL1 and CONTEXTIDR_EL2, captured on reading PMPCSR.

Configuration

External register PMCCIDSR bits [31:0] are architecturally mapped to External register PMVCIDSR[31:0].

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCCIDSR is a 64-bit register.

Field descriptions

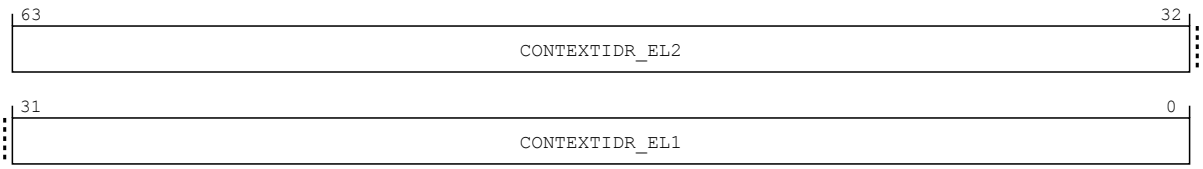

CONTEXTIDR_EL2, bits [63:32]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

Because the value written to this field is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether this field is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and this register samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCCIDSR

If FEAT_PCSRv8p2 and FEAT_PMUv3_EXT32 are implemented, then the same content is present in the same locations, and can be accessed using PMCID2SR[31:0] and PMCID1SR[31:0].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0x228` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.4 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. For more information, see 'Time as measured by the Performance Monitors cycle counter'.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configuration

PMCCNTR_EL0 is in the Core power domain.

External register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCCNTR_EL0[63:0].

External register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch32 System register PMCCNTR[63:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are RES0.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions

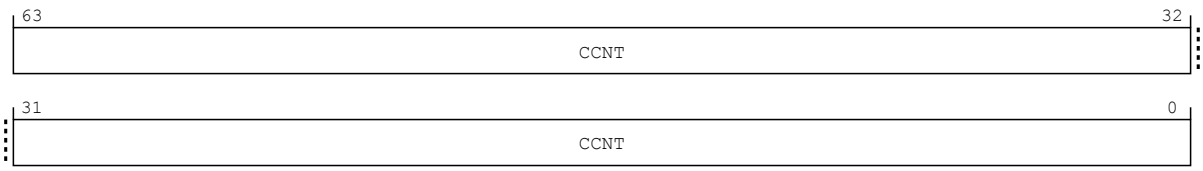

CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x0F8` from PMU

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x0F8` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented

[63:32] Accessible at offset `0x0FC` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.5 PMCCNTSVR_EL1, Performance Monitors Cycle Count Saved Value Register

The PMCCNTSVR_EL1 characteristics are:

Purpose

Captures the PMU Cycle counter, PMCCNTR_EL0.

Configuration

When FEAT_PMUv3_SS is implemented, External register PMCCNTSVR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMCCNTSVR_EL1[63:0].

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCCNTSVR_EL1 are RES0.

Attributes

PMCCNTSVR_EL1 is a 64-bit register.

Field descriptions

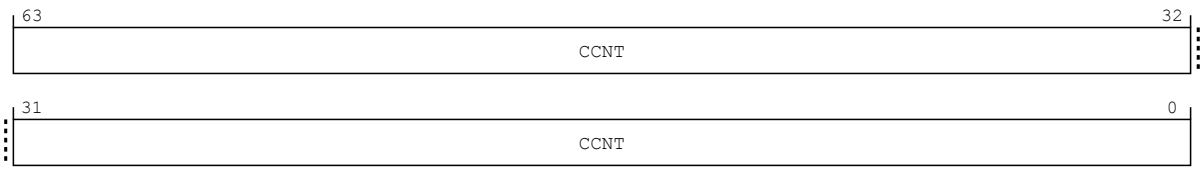

CCNT, bits [63:0]

Sampled Cycle Count. The value of PMCCNTR_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTSVR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented

Accessible at offset `0x6F8` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.6 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x0000` to `0x001F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID0_EL0.

Configuration

PMCEID0 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID0 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID0 bits [31:0] are architecturally mapped to AArch32 System register PMCEID0[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0 are RES0.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions

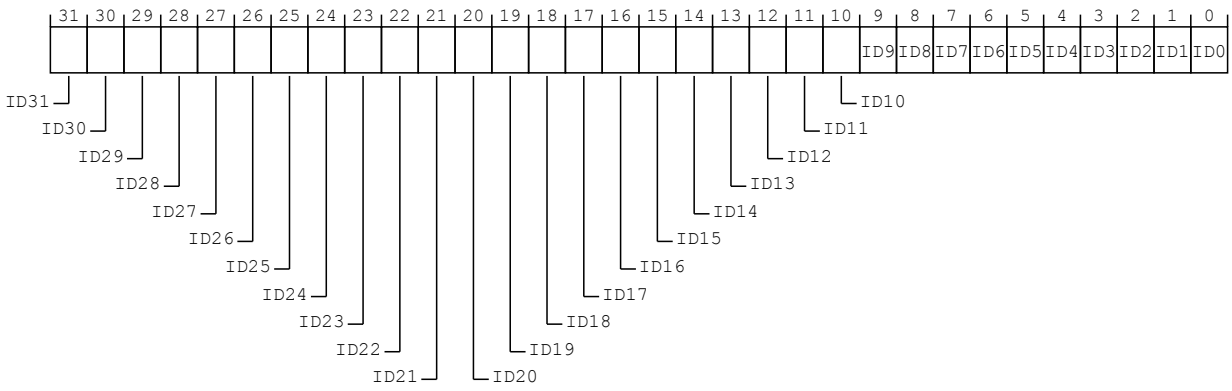

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE20` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.7 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x020` to `0x03F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID1_EL0.

Configuration

PMCEID1 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID1 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID1 bits [31:0] are architecturally mapped to AArch32 System register PMCEID1[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1 are RES0.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions

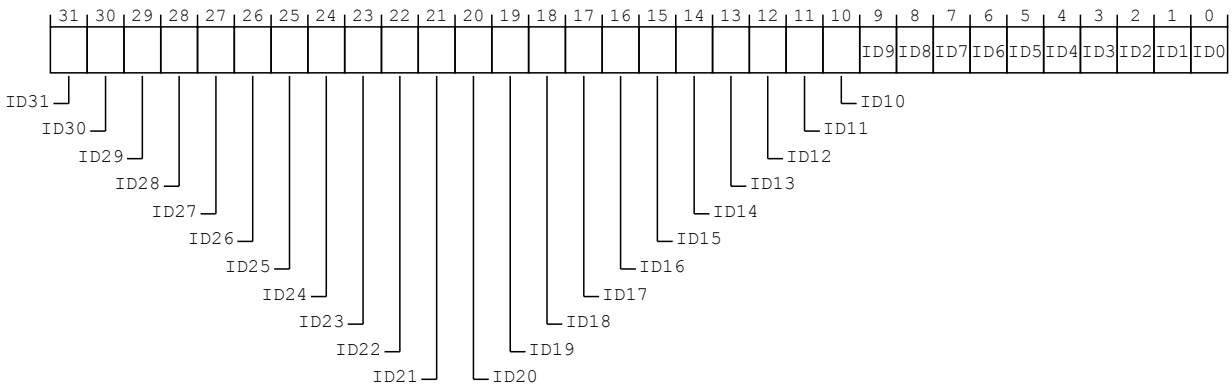

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (`0x0020` + n).

For each bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID1

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE24` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.8 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4000` to `0x401F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Use of this register is deprecated.

Configuration

PMCEID2 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID2 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[63:32].

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID2 bits [31:0] are architecturally mapped to AArch32 System register PMCEID2[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID2 are RES0.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

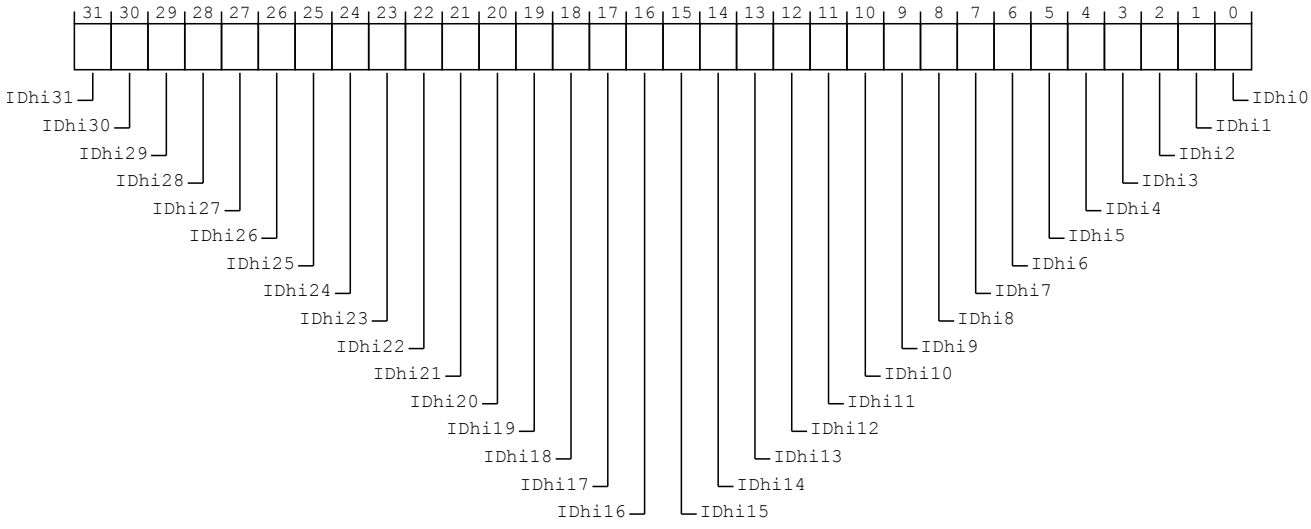

IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4000` + n).

For each bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID2

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented

Accessible at offset `0xE28` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.9 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4020` to `0x403F`.

For more information about the Common events and the use of the PMCEIDn registers, see 'The PMU event number space and common events'.

Use of this register is deprecated.

Configuration

PMCEID3 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID3 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[63:32].

When FEAT_PMUv3_EXT32 is implemented, External register PMCEID3 bits [31:0] are architecturally mapped to AArch32 System register PMCEID3[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID3 are RES0.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

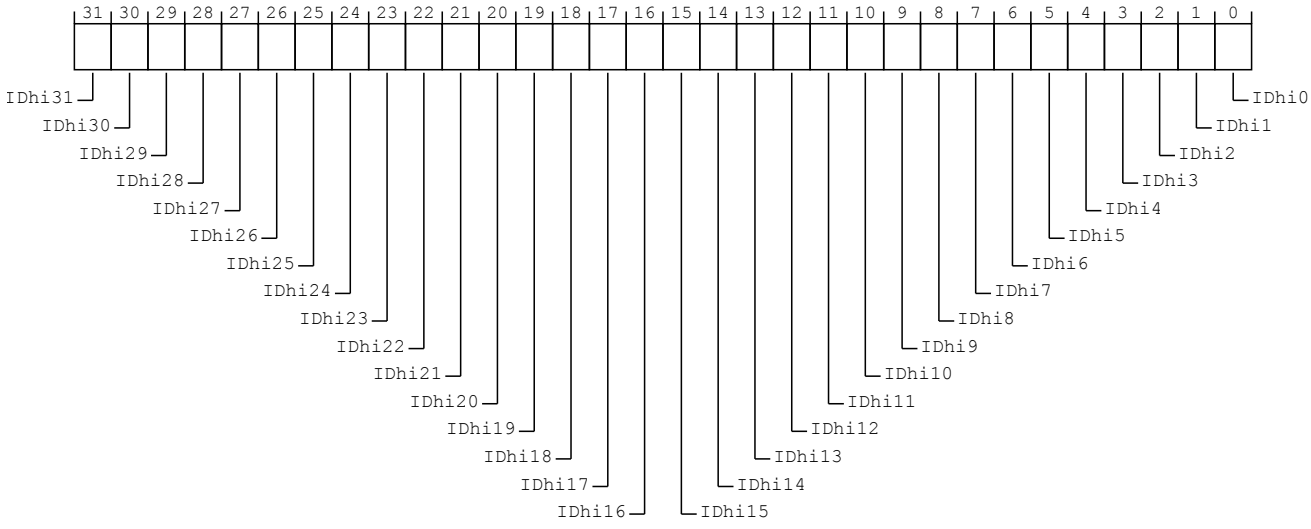

IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4020` + n).

For each bit:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID3

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented

Accessible at offset `0xE2C` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.10 PMCGCR0, Counter Group Configuration Register 0

The PMCGCR0 characteristics are:

Purpose

Encodes the number of counters accessible.

Configuration

PMCGCR0 is in the Core power domain.

This register is present only when FEAT_PMUv3_ICNTR is implemented, FEAT_PMUv3_EXT is implemented, and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCGCR0 are RES0.

Attributes

PMCGCR0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

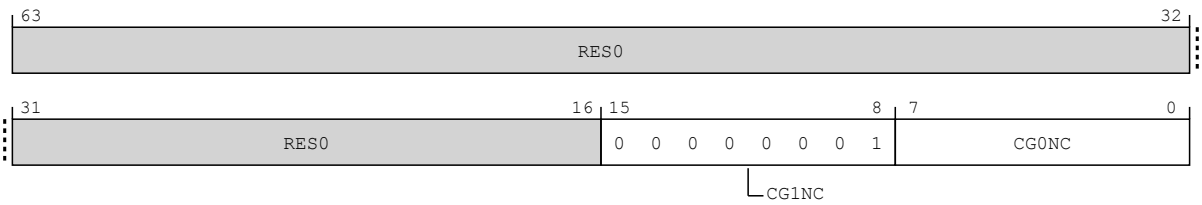

Bits [63:16]

Reserved, RES0.

CG1NC, bits [15:8]

Number of counters in group 1, which comprises the instruction counter PMICNTR_EL0.

CG1NC	Meaning
0x01	PMICNTR_EL0 implemented.

Other values are reserved.

Access to this field is RO.

CG0NC, bits [7:0]

Number of counters in group 0, which comprises the event counters PMEVCNTR<n>_EL0 and the cycle counter PMCCNTR_EL0.

This field has an IMPLEMENTATION DEFINED value.

When FEAT_PMUv3_EXTPMN is implemented and the external access to this register is not a Most secure access, this field reads as the Effective value of PMCCR.EPMN plus one.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise, this field reads as the number of event counters implemented plus one.

Access to this field is RO.

Otherwise:

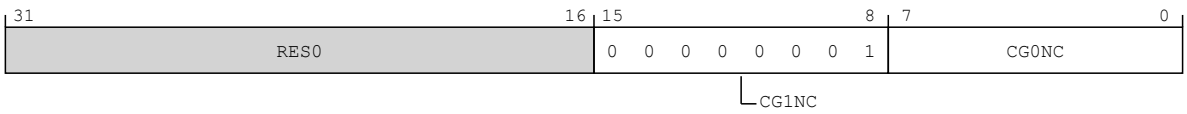

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Number of counters in group 1, which comprises the instruction counter PMICNTR_EL0.

CG1NC	Meaning
`0x01`	PMICNTR_EL0 implemented.

Other values are reserved.

Access to this field is RO.

CG0NC, bits [7:0]

Number of counters in group 0, which comprises the event counters PMEVCNTR<n>_EL0 and the cycle counter PMCCNTR_EL0.

This field has an IMPLEMENTATION DEFINED value.

When FEAT_PMUv3_EXTPMN is implemented and the external access to this register is not a Most secure access, this field reads as the Effective value of PMCCR.EPMN plus one.

Otherwise, this field reads as the number of event counters implemented plus one.

Access to this field is RO.

Accessing PMCGCR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3_ICNTR is implemented

[31:0] Accessible at offset `0xCE0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3_ICNTR is implemented

[63:0] Accessible at offset `0xCE0` from PMU

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.11 PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

Purpose

Contains PMU-specific configuration data.

Configuration

PMCFGR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCFGR are RES0.

Attributes

PMCFGR is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

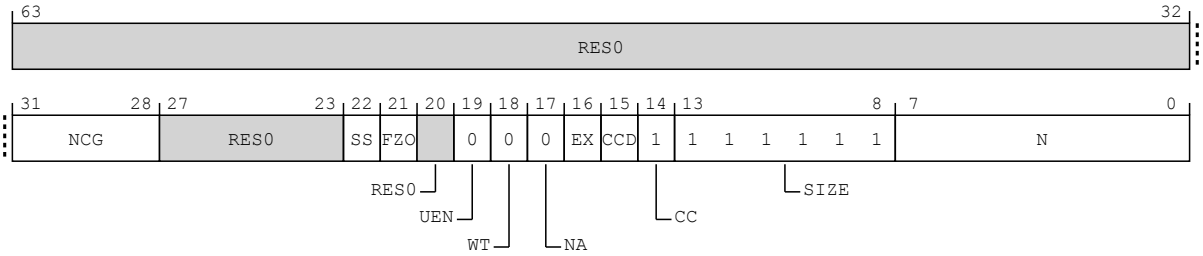

Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

FEAT_PMUv3_ICNTR implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [27:23]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS, bit [22]

Snapshot supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SS	Meaning
0b0	Snapshot mechanism not supported. The locations 0x600-0x7FC and 0xE30-0xE3C are IMPLEMENTATION DEFINED.
0b1	Snapshot mechanism supported. If FEAT_PMUv3_SS is implemented, then the following registers are implemented: • PMEVCNTSVR<n>_EL1. • PMCCNTSVR_EL1. • If FEAT_PMUv3_ICNTR is implemented, PMICNTSVR_EL1. • PMSSCR_EL1. Otherwise, locations 0x600-0x7FC and 0xE30-0xE3C contain IMPLEMENTATION DEFINED snapshot registers.

FEAT_PMUv3_SS implements the functionality identified by the value 1.

If FEAT_PMUv3_SS is not implemented, a PMU might include an IMPLEMENTATION DEFINED snapshot mechanism, including one using the IMPLEMENTATION DEFINED registers 0x600-0x7FC and 0xE30-0xE3C.

Access to this field is RO.

FZO, bit [21]

Freeze-on-overflow supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FZO	Meaning
0b0	Freeze-on-overflow mechanism is not supported. PMCR_EL0.FZO is RES0.
0b1	Freeze-on-overflow mechanism is supported. PMCR_EL0.FZO is RW.

FEAT_PMUv3p7 implements the functionality added by the value 1.

From Armv8.7, if FEAT_PMUv3 is implemented, the only permitted value is 1.

Access to this field is RO.

Bit [20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WT, bit [18]

This feature is not supported, so this bit is RAZ.

Reads as `0b0`

Access to this field is RO.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

Reads as `0b0`

Access to this field is RO.

EX, bit [16]

Export supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EX	Meaning
`0b0`	PMCR_EL0.X is RES0.
`0b1`	PMCR_EL0.X is read/write.

Access to this field is RO.

CCD, bit [15]

Cycle counter has prescale.

This is RES1 if AArch32 is supported, and RAZ otherwise.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCD	Meaning
`0b0`	PMCR_EL0.D is RES0.
`0b1`	PMCR_EL0.D is read/write.

Access to this field is RO.

CC, bit [14]

Dedicated cycle counter (counter 31) supported.

Reads as `0b1`

Access to this field is RO.

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8.0, the largest counter is 64-bits, so the value of this field is `0b111111`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8.0, the counters are a doubleword-aligned addresses.

Reads as `0b111111`

Access to this field is RO.

N, bits [7:0]

Number of counters accessible, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

N	Meaning
`0x00`	Only PMCCNTR_EL0 accessible.
`0x01..0x20`	Number of counters accessible, 2 to 33, minus one.

All other values are reserved.

If FEAT_PMUv3_ICNTR is implemented, then the value `0x00` is not permitted.

The count includes:

- The cycle counter, PMCCNTR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, the Instruction Counter, PMICNTR_EL0.

When FEAT_PMUv3_EXTPMN is implemented and the access to this register is not a Most secure access, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the Effective value of PMCCR.EPMN.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the Effective value of PMCCR.EPMN plus one.

Otherwise, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the number of event counters implemented.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the number of event counters implemented plus one.

For example, if PMCFGR.N == `0x07`, then:

- There are eight counters in total.
- If FEAT_PMUv3_ICNTR is not implemented, this comprises 7 event counters and the cycle counter.
- If FEAT_PMUv3_ICNTR is implemented, this comprises 6 event counters, the cycle counter, and the instruction counter.

The maximum number of event counters is 31.

Access to this field is RO.

Otherwise:

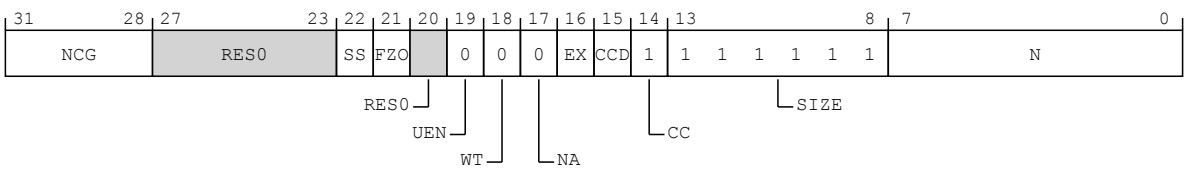

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

FEAT_PMUv3_ICNTR implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [27:23]

Reserved, RES0.

SS, bit [22]

Snapshot supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SS	Meaning
0b0	Snapshot mechanism not supported. The locations 0x600-0x7FC and 0xE30-0xE3C are IMPLEMENTATION DEFINED.
0b1	Snapshot mechanism supported. If FEAT_PMUv3_SS is implemented, then the following registers are implemented: • PMEVCNTSVR<n>_EL1. • PMCCNTSVR_EL1. • If FEAT_PMUv3_ICNTR is implemented, PMICNTSVR_EL1. • PMSSCR_EL1. Otherwise, locations 0x600-0x7FC and 0xE30-0xE3C contain IMPLEMENTATION DEFINED snapshot registers.

FEAT_PMUv3_SS implements the functionality identified by the value 1.

If FEAT_PMUv3_SS is not implemented, a PMU might include an IMPLEMENTATION DEFINED snapshot mechanism, including one using the IMPLEMENTATION DEFINED registers 0x600-0x7FC and 0xE30-0xE3C.

Access to this field is RO.

FZO, bit [21]

Freeze-on-overflow supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FZO	Meaning
0b0	Freeze-on-overflow mechanism is not supported. PMCR_EL0.FZO is RES0.
0b1	Freeze-on-overflow mechanism is supported. PMCR_EL0.FZO is RW.

FEAT_PMUv3p7 implements the functionality added by the value 1.

From Armv8.7, if FEAT_PMUv3 is implemented, the only permitted value is 1.

Access to this field is RO.

Bit [20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

WT, bit [18]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

EX, bit [16]

Export supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EX	Meaning
0b0	PMCR_EL0.X is RES0.
0b1	PMCR_EL0.X is read/write.

Access to this field is RO.

CCD, bit [15]

Cycle counter has prescale.

This is RES1 if AArch32 is supported, and RAZ otherwise.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCD	Meaning
0b0	PMCR_EL0.D is RES0.
0b1	PMCR_EL0.D is read/write.

Access to this field is RO.

CC, bit [14]

Dedicated cycle counter (counter 31) supported.

Reads as 0b1

Access to this field is RO.

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8.0, the largest counter is 64-bits, so the value of this field is 0b111111.

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8.0, the counters are a doubleword-aligned addresses.

Reads as 0b111111

Access to this field is RO.

N, bits [7:0]

Number of counters accessible, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

N	Meaning
0x00	Only PMCCNTR_EL0 accessible.
0x01..0x20	Number of counters accessible, 2 to 33, minus one.

All other values are reserved.

If FEAT_PMUv3_ICNTR is implemented, then the value 0x00 is not permitted.

The count includes:

- The cycle counter, PMCCNTR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, the Instruction Counter, PMICNTR_EL0.

When FEAT_PMUv3_EXTPMN is implemented and the access to this register is not a Most secure access, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the Effective value of PMCCR.EPMN.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the Effective value of PMCCR.EPMN plus one.

Otherwise, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the number of event counters implemented.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the number of event counters implemented plus one.

For example, if PMCFGR.N == 0x07, then:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- There are eight counters in total.
- If FEAT_PMUv3_ICNTR is not implemented, this comprises 7 event counters and the cycle counter.
- If FEAT_PMUv3_ICNTR is implemented, this comprises 6 event counters, the cycle counter, and the instruction counter.

The maximum number of event counters is 31.

Access to this field is RO.

Accessing PMCFGR

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0xE00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xE00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.12 PMCID1SR, CONTEXTIDR_EL1 Sample Register

The PMCID1SR characteristics are:

Purpose

PC Sample-based Profiling Extension register that contains the sampled value of CONTEXTIDR_EL1, captured on reading PMPCSR[31:0].

Configuration

PMCID1SR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCID1SR is a 32-bit register.

Field descriptions

31	0
CONTEXTIDR_EL1	

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and this register samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCID1SR

If FEAT_PCSRv8p2 and FEAT_PMUv3_EXT64 are implemented, then the same content is present in the same locations, and can be accessed using PMCCIDSR[31:0] or PMCVIDSR[31:0].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x208` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x228` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.13 PMCID2SR, CONTEXTIDR_EL2 Sample Register

The PMCID2SR characteristics are:

Purpose

PC Sample-based Profiling Extension register that contains the sampled value of CONTEXTIDR_EL2, captured on reading PMPCSR[31:0].

Configuration

PMCIDR2SR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCID2SR is a 32-bit register.

Field descriptions

31	0
CONTEXTIDR_EL2	

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

Because the value written to this field is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether this field is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCID2SR

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMCCIDSR[63:32].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x22C` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.14 PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR0. Otherwise, direct accesses to PMCIDR0 are RES0.

Attributes

PMCIDR0 is a 32-bit register.

Field descriptions

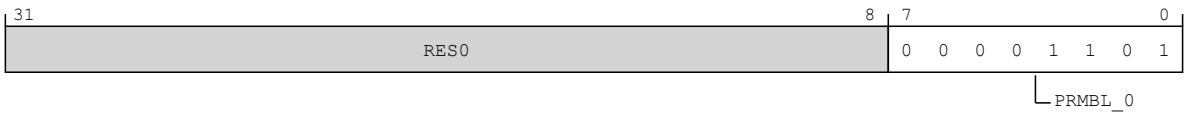

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing PMCIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.15 PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR1. Otherwise, direct accesses to PMCIDR1 are RES0.

Attributes

PMCIDR1 is a 32-bit register.

Field descriptions

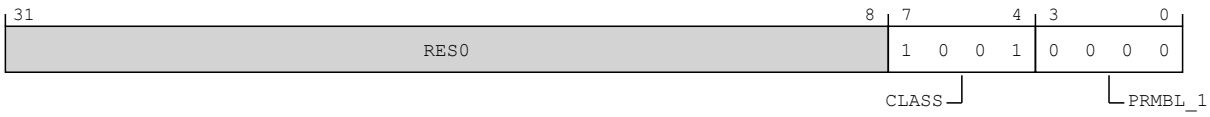

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMCIDR1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.16 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR2. Otherwise, direct accesses to PMCIDR2 are RES0.

Attributes

PMCIDR2 is a 32-bit register.

Field descriptions

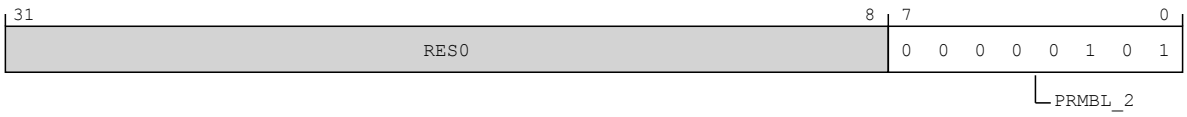

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing PMCIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.17 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR3. Otherwise, direct accesses to PMCIDR3 are RES0.

Attributes

PMCIDR3 is a 32-bit register.

Field descriptions

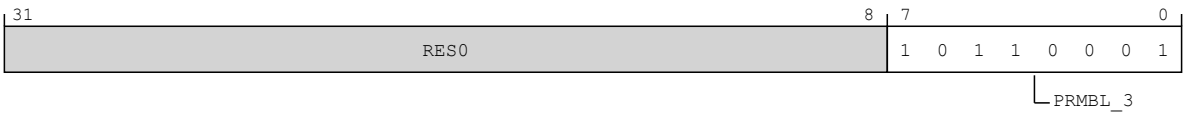

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing PMCIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFFC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.18 PMCNTEN, Performance Monitors Count Enable register

The PMCNTEN characteristics are:

Purpose

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>.

Configuration

External register PMCNTEN bits [63:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[63:0].

External register PMCNTEN bits [63:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[63:0].

External register PMCNTEN bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

External register PMCNTEN bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTEN are RES0.

Attributes

PMCNTEN is a 64-bit register.

Field descriptions

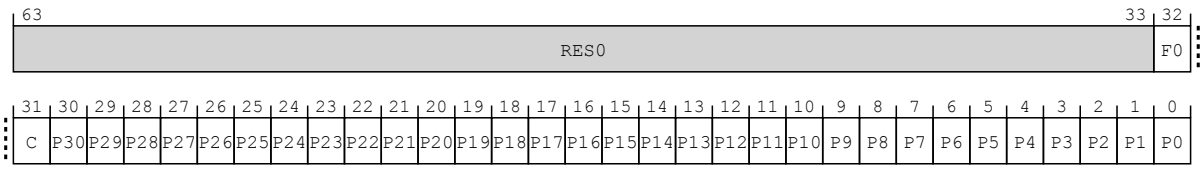

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 counter enable. Enables the instruction counter.

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable. Enables the cycle counter register. Possible values are:

C	Meaning
0b0	PMCCNTR_EL0 is disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

Event counter enable for PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 is disabled.
0b1	PMEVCNTR<m>_EL0 is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- Otherwise, access to this field is RW.

Accessing PMCNTEN

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xC10` from PMU

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.19 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which counters are enabled.

Configuration

PMCNTENCLR_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3p9 is not implemented, and FEAT_PMUv3_ICNTR is not implemented, External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3p9 is not implemented, and FEAT_PMUv3_ICNTR is not implemented, External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMCNTENCLR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[63:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMCNTENCLR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[63:0].

External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are RES0.

Attributes

PMCNTENCLR_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

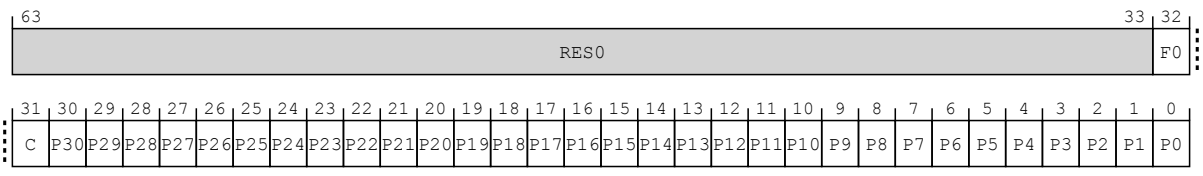

Bits [63:33]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 disable. On writes, allows software to disable PMICNTR_EL0. On reads, returns the PMICNTR_EL0 enable status.

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMCNTENCLR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset 0xC20 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset 0xC20 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.20 PMCNTENSET_EL0, Performance Monitors Count Enable Set Register

The PMCNTENSET_EL0 characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which counters are enabled.

Configuration

PMCNTENSET_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3p9 is not implemented, and FEAT_PMUv3_ICNTR is not implemented, External register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3p9 is not implemented, and FEAT_PMUv3_ICNTR is not implemented, External register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMCNTENSET_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[63:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMCNTENSET_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[63:0].

External register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

External register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are RES0.

Attributes

PMCNTENSET_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

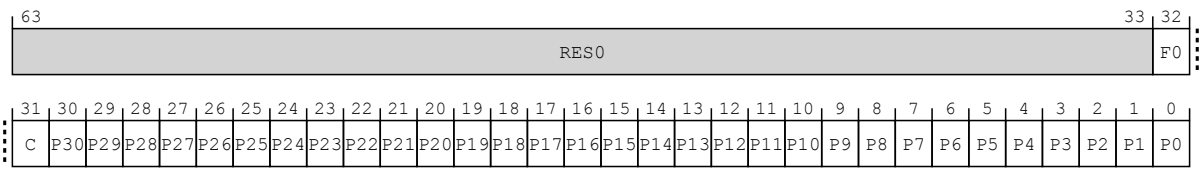

Bits [63:33]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 enable. On writes, allows software to enable PMICNTR_EL0. On reads, returns the PMICNTR_EL0 enable status.

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMCNTENSET_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset 0xC00 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset 0xC00 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.21 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Configures and controls the Performance Monitors counters.

Configuration

PMCR_EL0 is in the Core power domain.

This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

When FEAT_PMUv3_EXT64 is implemented, External register PMCR_EL0 bits [63:32] are architecturally mapped to AArch64 System register PMCR_EL0[63:32].

External register PMCR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCR_EL0[31:0].

External register PMCR_EL0 bits [10:0] are architecturally mapped to AArch32 System register PMCR[10:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR_EL0 are RES0.

Attributes

PMCR_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

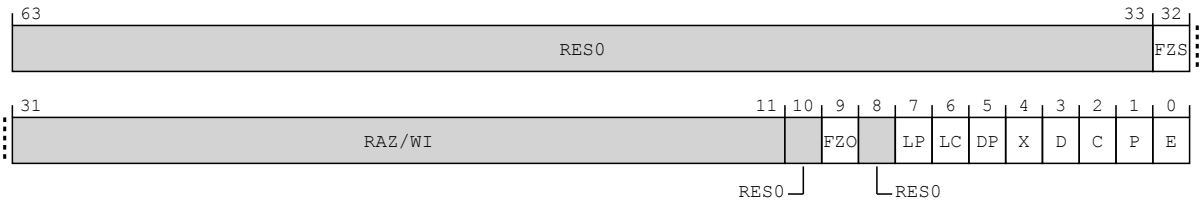

Bits [63:33]

Reserved, RES0.

FZS, bit [32]

When FEAT_SPEv1p2 is implemented:

Freeze-on-SPE event. Stop counters when PMBLIMITR_EL1.{PMFZ,E} is {1,1} and profiling is stopped.

FZS	Meaning
0b0	Do not freeze on a Statistical Profiling Buffer Management event.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Affected counters do not count following a Statistical Profiling Buffer Management event.

The pseudocode function `SPEProfilingStopped()` describes when profiling is stopped.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If FEAT_SPE_DPFZS is implemented and PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When FEAT_SPE_DPFZS is not implemented or PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA32 is implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

FZO	Meaning
0b0	Do not freeze on overflow.
0b1	Affected counters do not count when any of the following applies: • For any event counter PMEVCNTR<m>_EL0 in the first range, PMOVSCLR_EL0[m] is 1. • FEAT_PMUv3_ICNTR is implemented and PMOVSCLR_EL0.F0 is 1.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

LP, bit [7]

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.P[n].

LP	Meaning
0b0	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].
0b1	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

The counters affected by this field are the event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

When FEAT_AA32 is implemented:

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm deprecates use of PMCR_EL0.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

When ((HaveEL(EL3) || (IsFeatureImplemented(FEAT_PMUv3p1) && HaveEL(EL2))) || IsFeatureImplemented(FEAT_PMUv3p7)) || IsFeatureImplemented(FEAT_SPE_DPFZS):

Disable cycle counter when event counting is prohibited.

DP	Meaning
`0b0`	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD or HDCR.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2. • If FEAT_SPE_DPFZS is implemented and event counting is frozen by PMCR_EL0.FZS, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3. • If EL3 is implemented, MDCR_EL3.SPME or SDCR.SPME is 0, and one of FEAT_PMUv3p7 is not implemented, EL3 is using AArch32, or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited or frozen. For more information about event counter ranges, see MDCR_EL2.HPMN.

If FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented, meaning PMCR_EL0.FZS is implemented, and FEAT_SPE_DPFZS is not implemented, then cycle counting by PMCCNTR_EL0 is not affected by PMCR_EL0.FZS.

For more information, see ‘Prohibiting event and cycle counting’.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETE is implemented, this field does not affect the use of PMU events as an External Input by the trace unit.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When FEAT_AA32 is implemented:

Clock divider.

D	Meaning
0b0	When enabled, PMCCNTR_EL0 counts every clock cycle.
0b1	When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If the Effective value of PMCR_EL0.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
0b0	No action.
0b1	Reset all affected event counters PMEVCNTR<n>_EL0 to zero.

The event counters affected by this field are:

- If FEAT_PMUv3_EXTPMN is implemented and the access to this register is a Most secure access, all event counters.
- Otherwise, only event counters in the first and second ranges.

Writes to this field do not affect other event counters, the cycle counter PMCCNTR_EL0, or the instruction counter PMICNTR_EL0.

For more information about event counter ranges, see MDCR_EL2.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP or HDCR.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

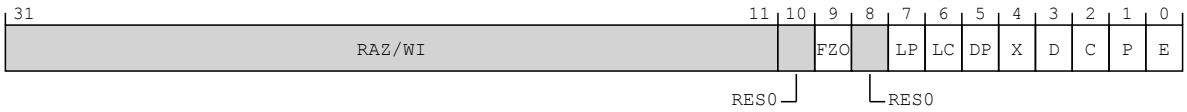

Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

FZO	Meaning
0b0	Do not freeze on overflow.
0b1	Affected counters do not count when any of the following applies: • For any event counter PMEVCNTR<m>_EL0 in the first range, PMOVSCLR_EL0[m] is 1. • FEAT_PMUv3_ICNTR is implemented and PMOVSCLR_EL0.F0 is 1.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [8]

Reserved, RES0.

LP, bit [7]

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.P[n].

LP	Meaning
0b0	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].
0b1	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

The counters affected by this field are the event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

When FEAT_AA32 is implemented:

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

Arm deprecates use of PMCR_EL0.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When ((HaveEL(EL3) || (IsFeatureImplemented(FEAT_PMUv3p1) && HaveEL(EL2))) || IsFeatureImplemented(FEAT_PMUv3p7)) || IsFeatureImplemented(FEAT_SPE_DPFZS):

Disable cycle counter when event counting is prohibited.

DP	Meaning
`0b0`	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD or HDCR.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2. • If FEAT_SPE_DPFZS is implemented and event counting is frozen by PMCR_EL0.FZS, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3. • If EL3 is implemented, MDCR_EL3.SPME or SDCR.SPME is 0, and one of FEAT_PMUv3p7 is not implemented, EL3 is using AArch32, or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited or frozen. For more information about event counter ranges, see MDCR_EL2.HPMN.

If FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented, meaning PMCR_EL0.FZS is implemented, and FEAT_SPE_DPFZS is not implemented, then cycle counting by PMCCNTR_EL0 is not affected by PMCR_EL0.FZS.

For more information, see ‘Prohibiting event and cycle counting’.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
`0b0`	Do not export events.
`0b1`	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETE is implemented, this field does not affect the use of PMU events as an External Input by the trace unit.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When FEAT_AA32 is implemented:

Clock divider.

D	Meaning
`0b0`	When enabled, PMCCNTR_EL0 counts every clock cycle.
`0b1`	When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If the Effective value of PMCR_EL0.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
`0b0`	No action.
`0b1`	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
`0b0`	No action.
`0b1`	Reset all affected event counters PMEVCNTR<n>_EL0 to zero.

The event counters affected by this field are:

- If FEAT_PMUv3_EXTPMN is implemented and the access to this register is a Most secure access, all event counters.
- Otherwise, only event counters in the first and second ranges.

Writes to this field do not affect other event counters, the cycle counter PMCCNTR_EL0, or the instruction counter PMICNTR_EL0.

For more information about event counter ranges, see MDCR_EL2.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP or HDCR.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
`0b0`	Affected counters are disabled and do not count.
`0b1`	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing PMCR_EL0

Accesses to this register use the following encodings:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE04` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xE10` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.22 PMDEVAFF0, Performance Monitors Device Affinity register 0

The PMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF0 is a 32-bit register.

Field descriptions

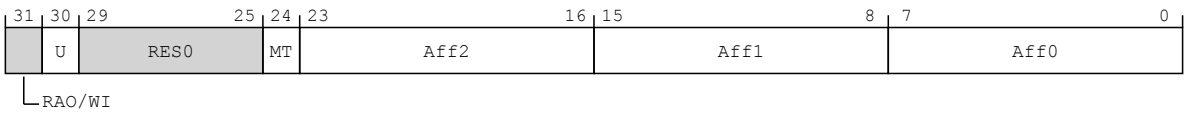

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF0

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMDEVAFF[31:0].

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xFA8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.23 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF1 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	Aff3

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of PMDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF1

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMDEVAFF[63:32].

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xFAC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.24 PMDEVAFF, Performance Monitors Device Affinity register

The PMDEVAFF characteristics are:

Purpose

Copy of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT64 is implemented.

Attributes

PMDEVAFF is a 64-bit register.

Field descriptions

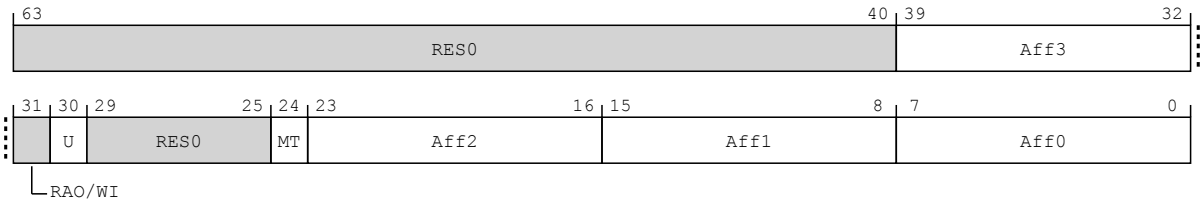

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF

If FEAT_PMUv3_EXT32 is implemented, then the same content is present in the same locations, and can be accessed using PMDEVAFF0[31:0] and PMDEVAFF1[31:0].

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xFA8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.25 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the Performance Monitor component.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMDEVARCH are RES0.

Attributes

PMDEVARCH is a 32-bit register.

Field descriptions

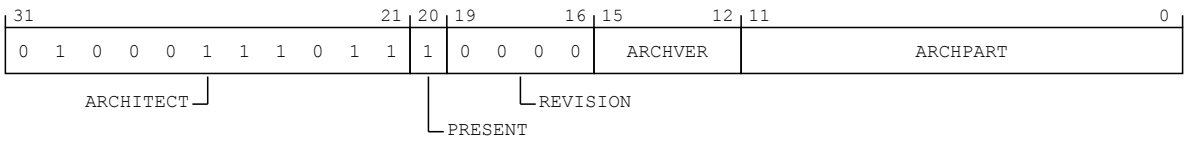

ARCHITECT, bits [31:21]

Defines the architect of the component. For Performance Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the PMDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For Performance Monitors, the revision defined by Armv8 is `0x0`.

All other values are reserved.

Reads as `0b0000`

Access to this field is RO.

ARCHVER, bits [15:12]

When UInt(PMU.PMDEVARCH.ARCHPART) == `0xA16` or UInt(PMU.PMDEVARCH.ARCHPART) == `0xA26`:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0010	Performance Monitors Extension version 3, PMUv3.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

Access to this field is RO.

Otherwise:

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0000	PC Sample-based Profiling version 2, FEAT_PCSRv8p2.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ARCHPART	Meaning
0xA10	PC Sample-based Profiling, including the 32-bit programmers' model extension.
0xA16	Armv8-A PE performance monitors, including the 32-bit programmers' model extension.
0xA20	PC Sample-based Profiling, including the 64-bit programmers' model extension.
0xA26	Armv8-A PE performance monitors, including the 64-bit programmers' model extension.

FEAT_PMUv3_EXT32 implements the functionality described by the values 0xA10 and 0xA16.

FEAT_PMUv3_EXT64 implements the functionality described by the values 0xA20 and 0xA26.

The values 0xA10 and 0xA20 indicate that FEAT_PCSRv8p2 is implemented but the Performance Monitors Extension is not implemented.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHPART is PMDEVARCH.ARCHID[11:0].

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMDEVARCH

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFBC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.26 PMDEVID, Performance Monitors Device ID register

The PMDEVID characteristics are:

Purpose

Provides information about features of the Performance Monitors implementation.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when (Armv8.2 or FEAT_PCSRv8p2 is implemented) and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMDEVID are RES0.

Attributes

PMDEVID is a 32-bit register.

Field descriptions

31 … 12	11 … 8	7 … 4	3 … 0
RES0	EXTPMN	PMSS	PCSample

Bits [31:12]

Reserved, RES0.

EXTPMN, bits [11:8]

Reserving PMU event counters for external agents.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXTPMN	Meaning
0b0000	PMUv3 Extension for reserving PMU event counters for external agents not implemented.
0b0001	PMUv3 Extension for reserving PMU event counters for external agents implemented.

All other values are reserved.

FEAT_PMUv3_EXTPMN implements the functionality identified by the value 0b0001.

Access to this field is RO.

PMSS, bits [7:4]

PMU Snapshot extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSS	Meaning
0b0000	PMU snapshot extension not implemented.
0b0001	PMU snapshot extension implemented.

All other values are reserved.

FEAT_PMUv3_SS implements the functionality identified by the value 0b0001.

Access to this field is RO.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using Performance Monitors registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the Performance Monitors register space.
0b0001	PC Sample-based Profiling Extension is implemented in the Performance Monitors register space.
0b0010	As 0b0001, and adds support for PMPCSCTL.

All other values are reserved.

FEAT_PCSRv8p2 implements the functionality identified by the value 0b0001.

FEAT_PCSRv8p9 implements the functionality identified by the value 0b0010.

If FEAT_PCSRv8p2 is not implemented, then the only permitted value is 0b0000.

From Armv8.2, when FEAT_PCSRv8p2 is implemented, the value 0b0000 is not permitted.

From Armv8.9, when FEAT_PCSRv8p9 is implemented, the value 0b0001 is not permitted.

Access to this field is RO.

Accessing PMDEVID

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented and (Armv8.2 or FEAT_PCSRv8p2 is implemented)

Accessible at offset 0xFC8 from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.27 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's performance monitor interface.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMDEVTYPE. Otherwise, direct accesses to PMDEVTYPE are RES0.

Attributes

PMDEVTYPE is a 32-bit register.

Field descriptions

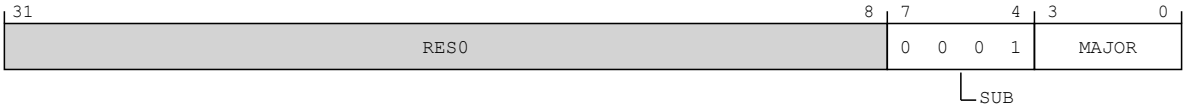

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MAJOR	Meaning
`0b0000`	Unspecified.
`0b0110`	Performance monitor component.

FEAT_PMUv3 implements the functionality described by the value `0b0110`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMDEVTYPE

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFCC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.28 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter <n>, which counts events, where <n> is 0 to 30.

Configuration

PMEVCNTR<n>_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p5 is implemented, External register PMEVCNTR<n>_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0[63:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p5 is not implemented, External register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0[31:0].

External register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVCNTR<n>[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are RES0.

Attributes

PMEVCNTR<n>_EL0 is a:

- 64-bit register when FEAT_PMUv3p5 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3p5 is implemented:

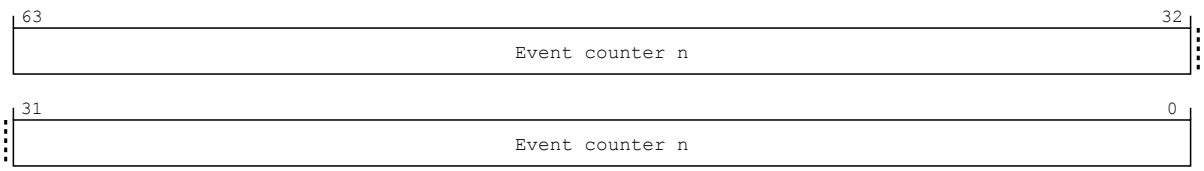

EVCNT, bits [63:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

If the highest implemented Exception level is using AArch32, the optional external interface to the performance monitors is implemented, and the PMCR.LP and HDCR.HLP bits are RAZ/WI, then locations in the external interface to the performance monitors that map to PMEVCNTR<n>_EL0[63:32] return UNKNOWN values on reads.

If the implementation does not support AArch64, bits [63:32] of the event counters are not required to be implemented.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

31	0
Event counter n	

EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>_EL0

If FEAT_PMUv3p5 is not implemented, when `IsCorePowered()`, `DoubleLockStatus()`, `OSLockStatus()` or `!AllowExternalPMUAccess()`, 32-bit accesses to `0x004`+8×n have a CONSTRAINED UNPREDICTABLE behavior.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p5 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p5 is not implemented

[31:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.29 PMEVCNTSVR<n>_EL1, Performance Monitors Event Count Saved Value Registers, n = 0 - 30

The PMEVCNTSVR<n>_EL1 characteristics are:

Purpose

Captures the PMU Event counter <n>, PMEVCNTR<n>_EL0.

Configuration

PMEVCNTSVR<n>_EL1 is in the Core power domain.

If event counter n is not implemented:

- When `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalPMUAccess(addrdesc)`, accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

When FEAT_PMUv3_SS is implemented, External register PMEVCNTSVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMEVCNTSVR<n>_EL1[63:0].

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMEVCNTSVR<n>_EL1 are RES0.

Attributes

PMEVCNTSVR<n>_EL1 is a 64-bit register.

Field descriptions

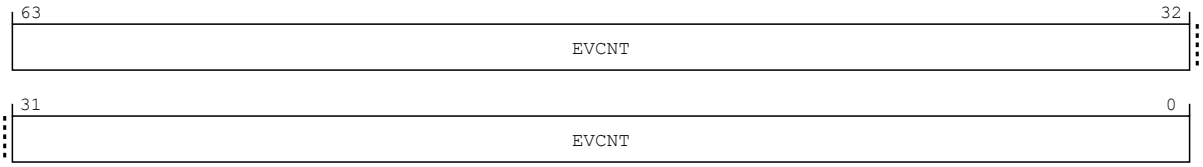

EVCNT, bits [63:0]

Sampled Event Count. The value of PMEVCNTR<n>_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTSVR<n>_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented

Accessible at offset `0x600 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.30 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented:

- When `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalPMUAccess()`, accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

External register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[31:0].

When FEAT_PMUv3_TH is implemented, or FEAT_PMUv3p8 is implemented, or FEAT_PMUv3_EXT64 is implemented, External register PMEVTYPER<n>_EL0 bits [63:32] are architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[63:32].

External register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVTYPER<n>[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are RES0.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions

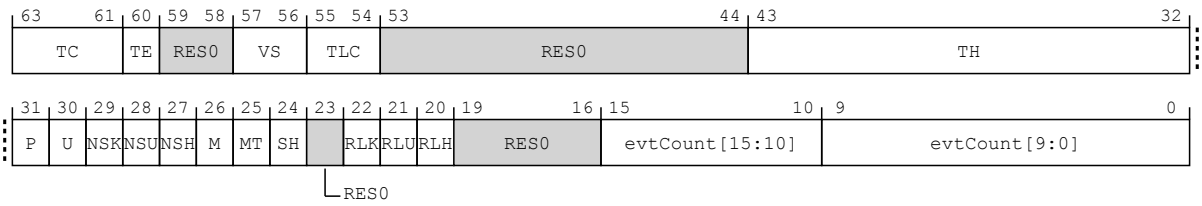

TC, bits [63:61]

When (IsFeatureImplemented(FEAT_PMUv3_TH) && ((!IsFeatureImplemented(FEAT_PMUv3_EDGE)) || (PMU.PMEVTYPER<n>_EL0.TE == '0'))) && (((!IsFeatureImplemented(FEAT_PMUv3_TH2)) || ((n MOD 2) == 0)) || (PMU.PMEVTYPER<n>_EL0.TLC IN {'0x'})):

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- For odd values of n, V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero. V[n-1] is not defined for even values of n.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TC	Meaning
0b000	Not-equal. The counter increments by V_B[n] on each processor cycle when V_B[n] is not equal to TH[n].
0b001	Not-equal, count. The counter increments by 1 on each processor cycle when V_B[n] is not equal to TH[n].
0b010	Equals. The counter increments by V_B[n] on each processor cycle when V_B[n] is equal to TH[n].
0b011	Equals, count. The counter increments by 1 on each processor cycle when V_B[n] is equal to TH[n].
0b100	Greater-than-or-equal. The counter increments by V_B[n] on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b101	Greater-than-or-equal, count. The counter increments by 1 on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b110	Less-than. The counter increments by V_B[n] on each processor cycle when V_B[n] is less than TH[n].
0b111	Less-than, count. The counter increments by 1 on each processor cycle when V_B[n] is less than TH[n].

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC[2:1] is true:

- If PMEVTYPER<n>_EL0.TC[0] is 0, then the counter increments by V_B[n].
- If PMEVTYPER<n>_EL0.TC[0] is 1, then the counter increments by 1.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC[2:1] is false:

- If FEAT_PMUv3_TH2 is implemented, n is odd, and PMEVTYPER<n>_EL0.TLC is `0b01`, then the counter increments by V[n-1].
- Otherwise, the counter does not increment.

If PMEVTYPER<n>_EL0.{TC, TLC, TH} are zero, then the threshold function is disabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3_TH2 is implemented, PMU.PMEVTYPER<n>_EL0.TE == '0', (n MOD 2) == 1, and PMU.PMEVTYPER<n>_EL0.TLC == '10':

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
0b000	Not-equal. The counter increments by V[n-1] on each processor cycle when V_B[n] is not equal to TH[n].
0b010	Equals. The counter increments by V[n-1] on each processor cycle when V_B[n] is equal to TH[n].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TC	Meaning
0b100	Greater-than-or-equal. The counter increments by V[n-1] on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b110	Less-than. The counter increments by V[n-1] on each processor cycle when V_B[n] is less than TH[n].

All other values are reserved.

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is true, the counter increments by V[n-1].

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is false, the counter does not increment.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3_EDGE is implemented and PMU.PMEVTYPER<n>_EL0.TE == '1':

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- For odd values of n, V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero. V[n-1] is not defined for even values of n.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
0b001	Equal to not-equal. The counter increments on each processor cycle when V_B[n] is not equal to TH[n] and V_B[n] was equal to TH[n] on the previous processor cycle.
0b010	Equal to/from not-equal. The counter increments on each processor cycle when either: • V_B[n] is not equal to TH[n] and V_B[n] was equal to TH[n] on the previous processor cycle. • V_B[n] is equal to TH[n] and V_B[n] was not equal to TH[n] on the previous processor cycle.
0b011	Not-equal to equal. The counter increments on each processor cycle when V_B[n] is equal to TH[n] and V_B[n] was not equal to TH[n] on the previous processor cycle.
0b101	Less-than to greater-than-or-equal. The counter increments on each processor cycle when V_B[n] is greater than or equal to TH[n] and V_B[n] was less than TH[n] on the previous processor cycle.
0b110	Less-than to/from greater-than-or-equal. The counter increments on each processor cycle when either: • V_B[n] is greater than or equal to TH[n] and V_B[n] was less than TH[n] on the previous processor cycle. • V_B[n] is less than TH[n] and V_B[n] was greater than or equal to TH[n] on the previous processor cycle.
0b111	Greater-than-or-equal to less-than. The counter increments on each processor cycle when V_B[n] is less than TH[n] and V_B[n] was greater than or equal to TH[n] on the previous processor cycle.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is true:

- If FEAT_PMUv3_TH2 is implemented, n is odd, and PMEVTYPER<n>_EL0.TLC is `0b10`, then the counter increments by V[n-1].
- Otherwise, the counter increments by 1.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is false, the counter does not increment.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TE, bit [60]

When FEAT_PMUv3_EDGE is implemented:

Threshold Edge. Enables the edge condition. When PMEVTYPER<n>_EL0.TE is 1, the event counter increments on cycles when the result of the threshold condition changes. See PMEVTYPER<n>_EL0.TC for more information.

TE	Meaning
`0b0`	Threshold edge condition disabled.
`0b1`	Threshold edge condition enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [59:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVE mode filtering. Controls counting events in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of events.
0b01	The PE does not count events in Streaming SVE mode.
0b10	The PE does not count events in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLC, bits [55:54]

When FEAT_PMUv3_TH2 is implemented and (n MOD 2) == 1:

Threshold Linking Control. Extends PMEVTYPER<n>_EL0.TC with additional controls for event linking. See PMEVTYPER<n>_EL0.TC.

TLC	Meaning
0b00	Threshold linking disabled.
0b01	Threshold linking enabled. If the threshold condition described by PMEVTYPER<n>_EL0.TC is false, the counter increments by V[n-1]. Otherwise, the counter increments as described by PMEVTYPER<n>_EL0.TC.
0b10	Threshold linking enabled. If the threshold condition described by PMEVTYPER<n>_EL0.TC is true, the counter increments by V[n-1]. Otherwise, the counter does not increment.

All other values are reserved.

See PMEVTYPER<n>_EL0.TC for more information

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [53:44]

Reserved, RES0.

TH, bits [43:32]

When FEAT_PMUv3_TH is implemented:

Threshold value. Provides the unsigned value for the threshold function defined by PMEVTYPER<n>_EL0.TC.

If PMEVTYPER<n>_EL0.{TC, TH} are both zero and either FEAT_PMUv3_TH2 is not implemented or PMEVTYPER<n>_EL0.TLC is also zero, then the threshold function is disabled.

If PMU.PMMIR_EL1.THWIDTH is less than 12, then bits PMEVTYPER<n>_EL0.TH[11:UInt(PMU.PMMIR_EL1.THWIDTH)] are RES0. This accounts for the behavior when writing a value greater-than-or-equal-to $2^{\text{UInt(PMU.PMMIR_EL1.THWIDTH)}}$.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to the expression `0x000`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting events in EL1.

P	Meaning
`0b0`	This mechanism has no effect on filtering of events.
`0b1`	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>_EL0.NSK.

If FEAT_RME is implemented, then counting events in Realm EL1 is further controlled by PMEVTYPER<n>_EL0.RLK.

If EL3 is implemented, then counting events at EL3 is further controlled by PMEVTYPER<n>_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting events in EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>_EL0.NSU.

If FEAT_RME is implemented, then counting events in Realm EL0 is further controlled by PMEVTYPER<n>_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting events in Non-secure EL1. If PMEVTYPER<n>_EL0.NSK is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL1.

NSK	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in Non-secure EL1.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in Non-secure EL1. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting events in Non-secure EL0. If PMEVTYPER<n>_EL0.NSU is not equal to PMEVTYPER<n>_EL0.U, then the PE does not count events in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSU	Meaning
0b0	When PMEVTYPER<n>_EL0.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.U == 1, the PE does not count events in Non-secure EL0.
0b1	When PMEVTYPER<n>_EL0.U == 0, the PE does not count events in Non-secure EL0. When PMEVTYPER<n>_EL0.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting events in Secure EL2 is further controlled by PMEVTYPER<n>_EL0.SH.

If FEAT_RME is implemented, then counting events in Realm EL2 is further controlled by PMEVTYPER<n>_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented and FEAT_AA64 is implemented:

EL3 filtering. Controls counting events in EL3. If PMEVTYPER<n>_EL0.M is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in EL3. Otherwise, this mechanism has no effect on filtering of events in EL3.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in EL3.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in EL3. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MT, bit [25]

When FEAT_MTPMU is implemented or an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See ‘Multithreaded implementations’ and ‘Cycle event counting in multithreaded implementations’.

From Armv8.6, the IMPLEMENTATION DEFINED multi-threaded PMU extension is not permitted, meaning if FEAT_MTPMU is not implemented, this field is RES0. See ID_AA64DFR0_EL1.MTPMU.

This field is ignored by the PE and treated as zero when FEAT_MTPMU is implemented and disabled.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting events in Secure EL2. If PMEVTYPER<n>_EL0.SH is equal to PMEVTYPER<n>_EL0.NSH, then the PE does not count events in Secure EL2. Otherwise, this mechanism has no effect on filtering of events in Secure EL2.

SH	Meaning
`0b0`	When PMEVTYPER<n>_EL0.NSH == 0, the PE does not count events in Secure EL2. When PMEVTYPER<n>_EL0.NSH == 1, this mechanism has no effect on filtering of events.
`0b1`	When PMEVTYPER<n>_EL0.NSH == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.NSH == 1, the PE does not count events in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting events in Realm EL1. If PMEVTYPER<n>_EL0.RLK is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in Realm EL1. Otherwise, this mechanism has no effect on filtering of events in Realm EL1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLK	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in Realm EL1.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in Realm EL1. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting events in Realm EL0. If PMEVTYPER<n>_EL0.RLU is not equal to PMEVTYPER<n>_EL0.U, then the PE does not count events in Realm EL0. Otherwise, this mechanism has no effect on filtering of events in Realm EL0.

RLU	Meaning
0b0	When PMEVTYPER<n>_EL0.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.U == 1, the PE does not count events in Realm EL0.
0b1	When PMEVTYPER<n>_EL0.U == 0, the PE does not count events in Realm EL0. When PMEVTYPER<n>_EL0.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting events in Realm EL2. If PMEVTYPER<n>_EL0.RLH is equal to PMEVTYPER<n>_EL0.NSH, then the PE does not count events in Realm EL2. Otherwise, this mechanism has no effect on filtering of events in Realm EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLH	Meaning
`0b0`	When PMEVTYPER<n>_EL0.NSH == 0, the PE does not count events in Realm EL2. When PMEVTYPER<n>_EL0.NSH == 1, this mechanism has no effect on filtering of events.
`0b1`	When PMEVTYPER<n>_EL0.NSH == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.NSH == 1, the PE does not count events in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When FEAT_PMUv3p1 is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMU.PMEVCNTR<n>_EL0.

The ranges of event numbers allocated to each type of event are shown in ‘Allocation of the PMU event number space’.

If FEAT_PMUv3p8 is implemented and PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.

Note

Arm recommends this behavior for all implementations of FEAT_PMUv3.

Otherwise, if PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_PMUv3p1 is implemented, for the range `0x4000` to `0x403F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>_EL0

If FEAT_PMUv3_EXT32 is implemented and any of the following apply, then bits [63:32] of this register are accessible at offset `0xA00` + (4*n):

- FEAT_PMUv3_TH is implemented.
- FEAT_PMUv3p8 is implemented.
- FEAT_PMUv3_SME is implemented.

Otherwise accesses at this offset are IMPLEMENTATION DEFINED.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x400 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x400 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When IsFeatureImplemented(FEAT_PMUv3_EXT32) && ((IsFeatureImplemented(FEAT_PMUv3_TH) || IsFeatureImplemented(FEAT_PMUv3p8)) || IsFeatureImplemented(FEAT_PMUv3_SME))

[63:32] Accessible at offset `0xA00 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.31 PMEVFILT2R<n>, Performance Monitors Event Filter Registers, n = 0 - 63

The PMEVFILT2R<n> characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED configuration controls for PMU counters.

Each PMEVFILT2R<n> register can provide additional configuration controls for a PMU counter, where:

- For values of n less than 31, if event counter n is implemented, then the controls are for PMU event counter <n>.
- For n equal to 31, the controls are for the cycle counter, PMCCNTR_EL0;
- For n equal to 32, if FEAT_PMUv3_ICNTR is implemented, the controls are for the instruction counter, PMICNTR_EL0;
- For all other values of n, PMEVFILT2R<n> is not implemented.

Although this mapping is recommended, it is not required and the function of each register is IMPLEMENTATION DEFINED.

Configuration

PMEVFILT2R<n> is in the Core power domain.

If PMEVFILT2R<n> is not implemented:

- When `IsCorePowered()` && !`DoubleLockStatus()` && !`OSLockStatus()` && `AllowExternalPMUAccess()`, accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_EXT is implemented, FEAT_PMUv3 is implemented, and an implementation implements PMEVFILT2R<n>. Otherwise, direct accesses to PMEVFILT2R<n> are RES0.

Attributes

PMEVFILT2R<n> is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

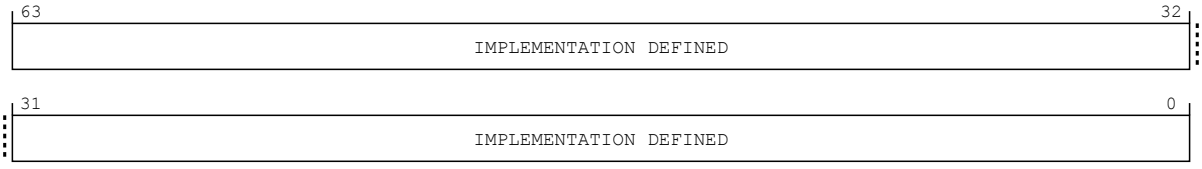

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing PMEVFILT2R<n>

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x800 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x800 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.32 PMICFILTR_EL0, Performance Monitors Instruction Counter Filter Register

The PMICFILTR_EL0 characteristics are:

Purpose

Configures the Instruction Counter.

Configuration

PMICFILTR_EL0 is in the Core power domain.

When FEAT_PMUv3_ICNTR is implemented, External register PMICFILTR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMICFILTR_EL0[63:0].

This register is present only when FEAT_PMUv3_ICNTR is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMICFILTR_EL0 are RES0.

Attributes

PMICFILTR_EL0 is a 64-bit register.

Field descriptions

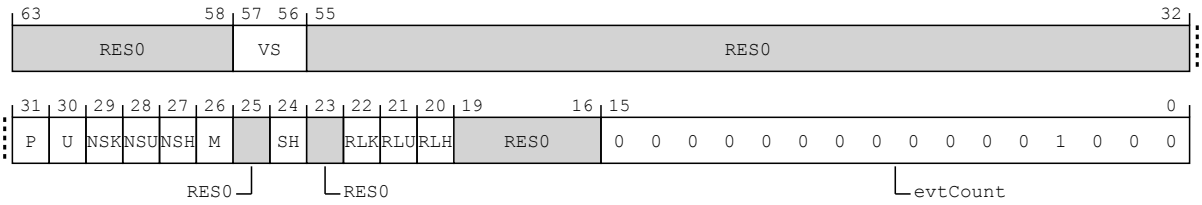

Bits [63:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting instructions in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of instructions.
0b01	The PE does not count instructions in Streaming SVE mode.
0b10	The PE does not count instructions in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting instructions in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of instructions.
0b1	The PE does not count instructions in EL1.

If Secure and Non-secure states are implemented, then counting instructions in Non-secure EL1 is further controlled by PMICFILTR_EL0.NSK.

If FEAT_RME is implemented, then counting instructions in Realm EL1 is further controlled by PMICFILTR_EL0.RLK.

If EL3 is implemented, then counting instructions at EL3 is further controlled by PMICFILTR_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting instructions in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of instructions.
0b1	The PE does not count instructions in EL0.

If Secure and Non-secure states are implemented, then counting instructions in Non-secure EL0 is further controlled by PMICFILTR_EL0.NSU.

If FEAT_RME is implemented, then counting instructions in Realm EL0 is further controlled by PMICFILTR_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting instructions in Non-secure EL1. If PMICFILTR_EL0.NSK is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of instructions in Non-secure EL1.

NSK	Meaning
`0b0`	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in Non-secure EL1.
`0b1`	When PMICFILTR_EL0.P == 0, the PE does not count instructions in Non-secure EL1. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting instructions in Non-secure EL0. If PMICFILTR_EL0.NSU is not equal to PMICFILTR_EL0.U, then the PE does not count instructions in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of instructions in Non-secure EL0.

NSU	Meaning
`0b0`	When PMICFILTR_EL0.U == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.U == 1, the PE does not count instructions in Non-secure EL0.
`0b1`	When PMICFILTR_EL0.U == 0, the PE does not count instructions in Non-secure EL0. When PMICFILTR_EL0.U == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting instructions in EL2.

NSH	Meaning
0b0	The PE does not count instructions in EL2.
0b1	This mechanism has no effect on filtering of instructions.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting instructions in Secure EL2 is further controlled by PMICFILTR_EL0.SH.

If FEAT_RME is implemented, then counting instructions in Realm EL2 is further controlled by PMICFILTR_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

EL3 filtering. Controls counting instructions in EL3. If PMICFILTR_EL0.M is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in EL3. Otherwise, this mechanism has no effect on filtering of instructions in EL3.

M	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in EL3.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in EL3. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting instructions in Secure EL2. If PMICFILTR_EL0.SH is equal to PMICFILTR_EL0.NSH, then the PE does not count instructions in Secure EL2. Otherwise, this mechanism has no effect on filtering of instructions in Secure EL2.

SH	Meaning
0b0	When PMICFILTR_EL0.NSH == 0, the PE does not count instructions in Secure EL2. When PMICFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of instructions.
0b1	When PMICFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.NSH == 1, the PE does not count instructions in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting instructions in Realm EL1. If PMICFILTR_EL0.RLK is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in Realm EL1. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL1.

RLK	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in Realm EL1.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in Realm EL1. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting instructions in Realm EL0. If PMICFILTR_EL0.RLU is not equal to PMICFILTR_EL0.U, then the PE does not count instructions in Realm EL0. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL0.

RLU	Meaning
0b0	When PMICFILTR_EL0.U == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.U == 1, the PE does not count instructions in Realm EL0.
0b1	When PMICFILTR_EL0.U == 0, the PE does not count instructions in Realm EL0. When PMICFILTR_EL0.U == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting instructions in Realm EL2. If PMICFILTR_EL0.RLH is equal to PMICFILTR_EL0.NSH, then the PE does not count instructions in Realm EL2. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL2.

RLH	Meaning
0b0	When PMICFILTR_EL0.NSH == 0, the PE does not count instructions in Realm EL2. When PMICFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of instructions.
0b1	When PMICFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.NSH == 1, the PE does not count instructions in Realm EL2.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count.

Reads as `0x0008`

Access to this field is RO.

Accessing PMICFILTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3_ICNTR is implemented

[31:0] Accessible at offset `0x480` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3_ICNTR is implemented

[63:0] Accessible at offset `0x500` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3_ICNTR is implemented

[63:32] Accessible at offset `0xA80` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.33 PMICNTR_EL0, Performance Monitors Instruction Counter Register

The PMICNTR_EL0 characteristics are:

Purpose

If event counting is not prohibited and the instruction counter is enabled, the counter increments for each architecturally-executed instruction, according to the configuration specified by PMICFILTR_EL0.

Configuration

PMICNTR_EL0 is in the Core power domain.

When FEAT_PMUv3_ICNTR is implemented, External register PMICNTR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMICNTR_EL0[63:0].

This register is present only when FEAT_PMUv3_ICNTR is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMICNTR_EL0 are RES0.

Attributes

PMICNTR_EL0 is a 64-bit register.

Field descriptions

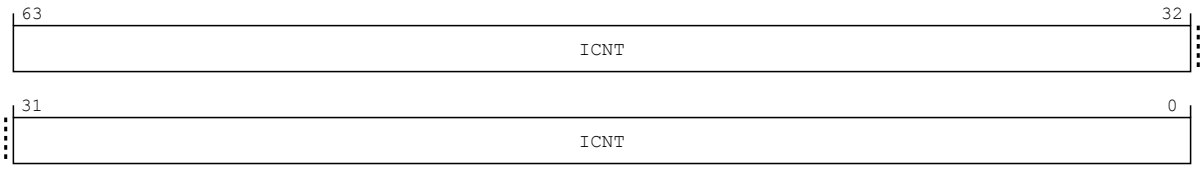

ICNT, bits [63:0]

Instruction Counter.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMICNTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_ICNTR is implemented

Accessible at offset `0x100` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.34 PMICNTSVR_EL1, Performance Monitors Instruction Count Saved Value Register

The PMICNTSVR_EL1 characteristics are:

Purpose

Captures the PMU Instruction counter, PMICNTR_EL0.

Configuration

When FEAT_PMUv3_ICNTR is implemented and FEAT_PMUv3_SS is implemented, External register PMICNTSVR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMICNTSVR_EL1[63:0].

This register is present only when FEAT_PMUv3_ICNTR is implemented, FEAT_PMUv3_SS is implemented, and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMICNTSVR_EL1 are RES0.

Attributes

PMICNTSVR_EL1 is a 64-bit register.

Field descriptions

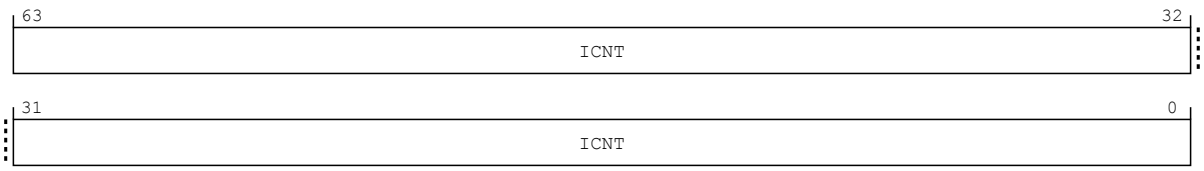

ICNT, bits [63:0]

Sampled Instruction Count. The value of PMICNTR_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMICNTSVR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented and FEAT_PMUv3_ICNTR is implemented

Accessible at offset `0x700` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.35 PMIIDR, Performance Monitors Implementation Identification Register

The PMIIDR characteristics are:

Purpose

Provides discovery information about the Performance Monitor component.

Configuration

This register is present only when (FEAT_PMUv3_EXT32 is implemented and an implementation implements PMIIDR) or FEAT_PMUv3_EXT64 is implemented. Otherwise, direct accesses to PMIIDR are RES0.

Attributes

PMIIDR is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

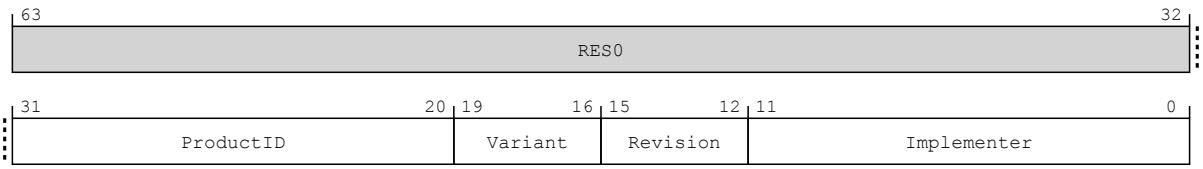

Bits [63:32]

Reserved, RES0.

ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the part number represented in the Peripheral ID registers PMPIDRn, if those registers are present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by PMIIDR.ProductID, or the major revision of the component.

When defining a major revision, PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the most significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, PMIIDR.Revision should be set to `0b0000`.

Arm recommends this field matches the PMPIDR2.REVISION field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Revision, bits [15:12]

Component minor revision.

PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the least significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If PMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

Arm recommends this field matches the PMPIDR3.REVAND field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for PMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If PMPIDR4 is implemented, PMPIDR4.DES_2 matches bits [11:8] of this field.

If PMPIDR2 is implemented, PMPIDR2.DES_1 matches bits [6:4] of this field.

If PMPIDR1 is implemented, PMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Otherwise:

31 … 20	19 … 16	15 … 12	11 … 0
ProductID	Variant	Revision	Implementer

ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the part number represented in the Peripheral ID registers PMPIDRn, if those registers are present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by PMIIDR.ProductID, or the major revision of the component.

When defining a major revision, PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the most significant part.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, PMIIDR.Revision should be set to `0b0000`.

Arm recommends this field matches the PMPIDR2.REVISION field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [15:12]

Component minor revision.

PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the least significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If PMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

Arm recommends this field matches the PMPIDR3.REVAND field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for PMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If PMPIDR4 is implemented, PMPIDR4.DES_2 matches bits [11:8] of this field.

If PMPIDR2 is implemented, PMPIDR2.DES_1 matches bits [6:4] of this field.

If PMPIDR1 is implemented, PMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Accessing PMIIDR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xE08` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.36 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register

The PMINTENCLR_EL1 characteristics are:

Purpose

Allows software to disable the generation of interrupt requests or, when FEAT_EBEP is implemented, PMU Profiling exceptions on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which overflow interrupt requests or PMU Profiling exceptions are enabled.

Configuration

PMINTENCLR_EL1 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMINTENCLR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[63:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMINTENCLR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[63:0].

External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are RES0.

Attributes

PMINTENCLR_EL1 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable status.

F0	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
`0b0`	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
`0b1`	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset `0xC60` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xC60` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.37 PMINTEN, Performance Monitors Interrupt Enable register

The PMINTEN characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0.

Configuration

External register PMINTEN bits [63:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[63:0].

External register PMINTEN bits [63:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[63:0].

External register PMINTEN bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

External register PMINTEN bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTEN are RES0.

Attributes

PMINTEN is a 64-bit register.

Field descriptions

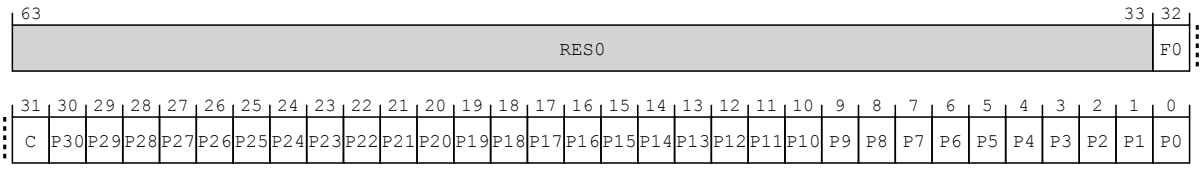

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request on unsigned overflow of PMICNTR_EL0 enable.

F0	Meaning
0b0	Interrupt request on unsigned overflow of PMICNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 unsigned overflow interrupt request enable bit. Possible values are:

C	Meaning
0b0	The cycle counter overflow interrupt request is disabled.
0b1	The cycle counter overflow interrupt request is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

Event counter unsigned overflow interrupt request enable bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	The PMEVCNTR<m>_EL0 event counter interrupt request is disabled.
0b1	The PMEVCNTR<m>_EL0 event counter interrupt request is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- Otherwise, access to this field is RW.

Accessing PMINTEN

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset 0xC50 from PMU

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.38 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register

The PMINTENSET_EL1 characteristics are:

Purpose

Allows software to enable the generation of interrupt requests or, when FEAT_EBEP is implemented, PMU Profiling exceptions on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which overflow interrupt requests or PMU Profiling exceptions are enabled.

Configuration

PMINTENSET_EL1 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMINTENSET_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[63:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMINTENSET_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[63:0].

External register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

External register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are RES0.

Attributes

PMINTENSET_EL1 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63–33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable status.

F0	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
`0b0`	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
`0b1`	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMINTENSET_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset `0xC40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xC40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.39 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

Purpose

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

This register is present only when FEAT_PMUv3_EXT is implemented, FEAT_PMUv3 is implemented, and an implementation implements PMITCTRL. Otherwise, direct accesses to PMITCTRL are RES0.

Attributes

PMITCTRL is a 32-bit register.

Field descriptions

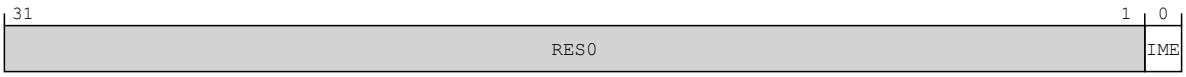

Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to 0.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to 0.
  - On a Warm reset, the value of this field is unchanged.

Accessing PMITCTRL

Accesses to this register use the following encodings:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xF00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.40 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

Purpose

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMLAR are RES0.

Attributes

PMLAR is a 32-bit register.

Field descriptions

When PMU Software Lock is implemented:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMLAR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFB0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.41 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

Purpose

Indicates the current status of the software lock for Performance Monitors registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMLSR are RES0.

Attributes

PMLSR is a 32-bit register.

Field descriptions

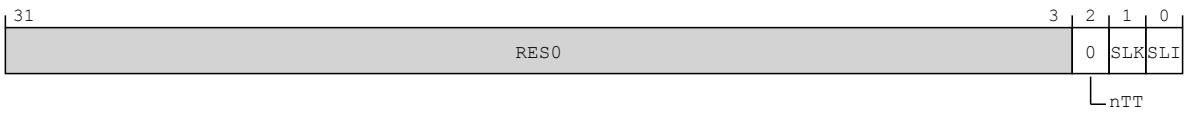

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required.

Reads as `0b0`

Access to this field is RO.

SLK, bit [1]

When PMU Software Lock is implemented and FEAT_DoPD is not implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
`0b0`	Lock clear. Writes are permitted to this component's registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b1`	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'1'`.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
`0b0`	Software Lock not implemented or not memory-mapped access.
`0b1`	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing PMLSR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFB4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.42 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation.

Configuration

PMMIR is in the Core power domain.

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented, External register PMMIR bits [31:0] are architecturally mapped to AArch32 System register PMMIR[31:0].

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented, External register PMMIR bits [31:0] are architecturally mapped to AArch64 System register PMMIR_EL1[31:0].

When (IsFeatureImplemented(FEAT_PMUv3_EXT) && IsFeatureImplemented(FEAT_PMUv3p4)) && (IsFeatureImplemented(FEAT_PMUv3_EXT64) || IsFeatureImplemented(FEAT_PMUv3p9)), External register PMMIR bits [63:0] are architecturally mapped to AArch64 System register PMMIR_EL1[63:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR are RES0.

Attributes

PMMIR is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented:

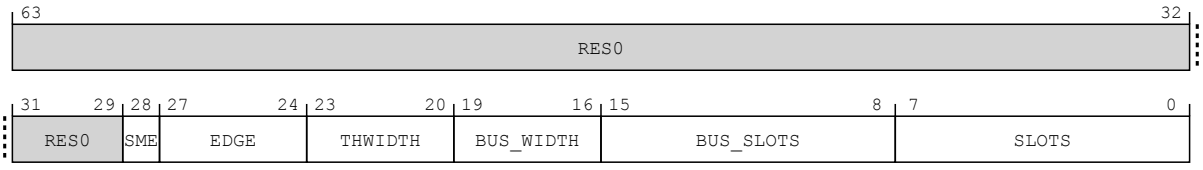

Bits [63:29]

Reserved, RES0.

SME, bit [28]

PMUv3 for SME. Adds support for the Streaming SVE mode filter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
0b0	Streaming SVE mode filter not implemented.
0b1	Adds support for the Streaming SVE mode filter.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_PMUv3_SME implements the functionality identified by the value 1.

Access to this field is RO.

EDGE, bits [27:24]

PMU event edge detection. With PMMIR_EL1.THWIDTH, indicates implementation of event counter thresholding features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDGE	Meaning
`0b0000`	FEAT_PMUv3_EDGE is not implemented.
`0b0001`	FEAT_PMUv3_EDGE is implemented.
`0b0010`	As `0b0001`, and adds support for threshold value linking between a pair of counters.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, the only permitted value is `0b0000`.

FEAT_PMUv3_EDGE implements the functionality identified by the value `0b0001`.

FEAT_PMUv3_TH2 implements the functionality identified by the value `0b0010`.

Access to this field is RO.

THWIDTH, bits [23:20]

PMEVTYPER<n>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<n>_EL0.TH field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

THWIDTH	Meaning
`0b0000`	FEAT_PMUv3_TH is not implemented.
`0b0001`	1 bit. PMEVTYPER<n>_EL0.TH[11:1] are RES0.
`0b0010`	2 bits. PMEVTYPER<n>_EL0.TH[11:2] are RES0.
`0b0011`	3 bits. PMEVTYPER<n>_EL0.TH[11:3] are RES0.
`0b0100`	4 bits. PMEVTYPER<n>_EL0.TH[11:4] are RES0.
`0b0101`	5 bits. PMEVTYPER<n>_EL0.TH[11:5] are RES0.
`0b0110`	6 bits. PMEVTYPER<n>_EL0.TH[11:6] are RES0.
`0b0111`	7 bits. PMEVTYPER<n>_EL0.TH[11:7] are RES0.
`0b1000`	8 bits. PMEVTYPER<n>_EL0.TH[11:8] are RES0.
`0b1001`	9 bits. PMEVTYPER<n>_EL0.TH[11:9] are RES0.
`0b1010`	10 bits. PMEVTYPER<n>_EL0.TH[11:10] are RES0.
`0b1011`	11 bits. PMEVTYPER<n>_EL0.TH[11] is RES0.
`0b1100`	12 bits.

All other values are reserved.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<n>_EL0.TH is $2^{(PMMIR.THWIDTH)}$ minus one.

Access to this field is RO.

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
`0b0000`	The information is not available.
`0b0011`	Four bytes.
`0b0100`	8 bytes.
`0b0101`	16 bytes.
`0b0110`	32 bytes.
`0b0111`	64 bytes.
`0b1000`	128 bytes.
`0b1001`	256 bytes.
`0b1010`	512 bytes.
`0b1011`	1024 bytes.
`0b1100`	2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

31–29	28	27–24	23–20	19–16	15–8	7–0
RES0	SME	EDGE	THWIDTH	BUS_WIDTH	BUS_SLOTS	SLOTS

Bits [31:29]

Reserved, RES0.

SME, bit [28]

PMUv3 for SME. Adds support for the Streaming SVE mode filter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
0b0	Streaming SVE mode filter not implemented.
0b1	Adds support for the Streaming SVE mode filter.

All other values are reserved.

FEAT_PMUv3_SME implements the functionality identified by the value 1.

Access to this field is RO.

EDGE, bits [27:24]

PMU event edge detection. With PMMIR_EL1.THWIDTH, indicates implementation of event counter thresholding features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDGE	Meaning
0b0000	FEAT_PMUv3_EDGE is not implemented.
0b0001	FEAT_PMUv3_EDGE is implemented.
0b0010	As 0b0001, and adds support for threshold value linking between a pair of counters.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, the only permitted value is 0b0000.

FEAT_PMUv3_EDGE implements the functionality identified by the value 0b0001.

FEAT_PMUv3_TH2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

THWIDTH, bits [23:20]

PMEVTYPER<n>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<n>_EL0.TH field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

THWIDTH	Meaning
0b0000	FEAT_PMUv3_TH is not implemented.
0b0001	1 bit. PMEVTYPER<n>_EL0.TH[11:1] are RES0.
0b0010	2 bits. PMEVTYPER<n>_EL0.TH[11:2] are RES0.
0b0011	3 bits. PMEVTYPER<n>_EL0.TH[11:3] are RES0.
0b0100	4 bits. PMEVTYPER<n>_EL0.TH[11:4] are RES0.
0b0101	5 bits. PMEVTYPER<n>_EL0.TH[11:5] are RES0.
0b0110	6 bits. PMEVTYPER<n>_EL0.TH[11:6] are RES0.
0b0111	7 bits. PMEVTYPER<n>_EL0.TH[11:7] are RES0.
0b1000	8 bits. PMEVTYPER<n>_EL0.TH[11:8] are RES0.
0b1001	9 bits. PMEVTYPER<n>_EL0.TH[11:9] are RES0.
0b1010	10 bits. PMEVTYPER<n>_EL0.TH[11:10] are RES0.
0b1011	11 bits. PMEVTYPER<n>_EL0.TH[11] is RES0.
0b1100	12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<n>_EL0.TH is $2^{(PMMIR.THWIDTH)}$ minus one.

Access to this field is RO.

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
0b0000	The information is not available.
0b0011	Four bytes.
0b0100	8 bytes.
0b0101	16 bytes.
0b0110	32 bytes.
0b0111	64 bytes.
0b1000	128 bytes.
0b1001	256 bytes.
0b1010	512 bytes.
0b1011	1024 bytes.
0b1100	2048 bytes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR

If the Core power domain is off or in a low-power state, access to this register returns an Error.

Accesses to this register use the following encodings:

When FEAT_PMUv3p4 is implemented and (FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented)

[63:0] Accessible at offset `0xE40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3p4 is implemented, FEAT_PMUv3_EXT32 is implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xE40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.43 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSCLR_EL0 characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSCLR_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMOVSCLR_EL0 bits [63:32] are architecturally mapped to AArch64 System register PMOVSSET_EL0[63:32].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMOVSCLR_EL0 bits [63:32] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[63:32].

External register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

External register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSSET[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are RES0.

Attributes

PMOVSCLR_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

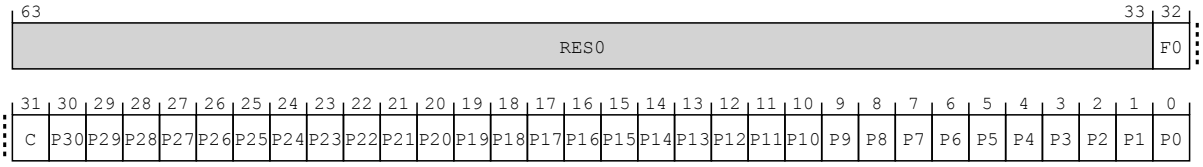

Bits [63:33]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Unsigned overflow flag for PMICNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMICNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMICNTR_EL0.

F0	Meaning
`0b0`	PMICNTR_EL0 has not overflowed.
`0b1`	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
`0b0`	PMCCNTR_EL0 has not overflowed.
`0b1`	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

P<m>	Meaning
`0b0`	PMEVCNTR<m>_EL0 has not overflowed.
`0b1`	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMOVSCLR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset `0xC80` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xC80` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.44 PMOVS, Performance Monitors Overflow Flag Status register

The PMOVS characteristics are:

Purpose

The unsigned overflow flags for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>.

Configuration

External register PMOVS bits [63:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[63:0].

External register PMOVS bits [63:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[63:0].

External register PMOVS bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

External register PMOVS bits [31:0] are architecturally mapped to AArch32 System register PMOVSSET[31:0].

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVS are RES0.

Attributes

PMOVS is a 64-bit register.

Field descriptions

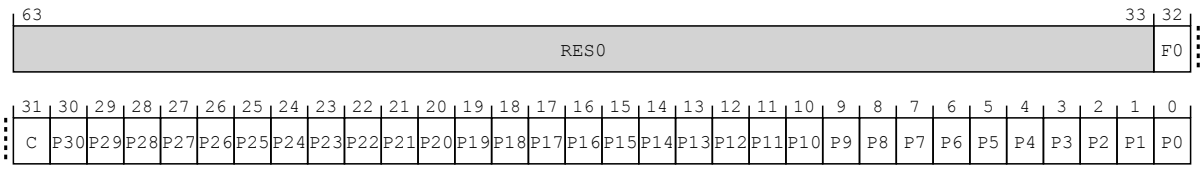

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 unsigned overflow flag.

F0	Meaning
0b0	PMICNTR_EL0 has not overflowed.
0b1	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C, bit [31]

Cycle counter unsigned overflow flag.

C	Meaning
0b0	The cycle counter has not overflowed since this bit was last cleared to 0.
0b1	The cycle counter has overflowed since this bit was last cleared to 0.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

Event counter unsigned overflow bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed since this bit was last cleared to 0.
0b1	PMEVCNTR<m>_EL0 has overflowed since this bit was last cleared to 0.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- Otherwise, access to this field is RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMOVS

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xC90` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.45 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register

The PMOVSSET_EL0 characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSSET_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMOVSSET_EL0 bits [63:32] are architecturally mapped to AArch64 System register PMOVSSET_EL0[63:32].

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented, External register PMOVSSET_EL0 bits [63:32] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[63:32].

External register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

External register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSSET[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are RES0.

Attributes

PMOVSSET_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63:33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_ICNTR is implemented:

Unsigned overflow flag for PMICNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMICNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMICNTR_EL0.

F0	Meaning
`0b0`	PMICNTR_EL0 has not overflowed.
`0b1`	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
`0b0`	PMCCNTR_EL0 has not overflowed.
`0b1`	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

P<m>	Meaning
`0b0`	PMEVCNTR<m>_EL0 has not overflowed.
`0b1`	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMOVSSET_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset `0xCC0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xCC0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.46 PMPCSCTL, PC Sample-based Profiling Control Register

The PMPCSCTL characteristics are:

Purpose

Controls the PC Sample-based Profiling feature.

Configuration

PMPCSCTL is in the Core power domain.

This register is present only when FEAT_PCSRv8p9 is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMPCSCTL are RES0.

Attributes

PMPCSCTL is a 64-bit register.

Field descriptions

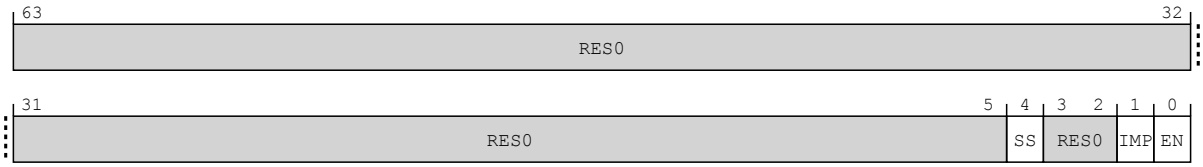

Bits [63:5]

Reserved, RES0.

SS, bit [4]

When FEAT_PMUv3_SS is implemented:

Sample on Snapshot.

Controls whether the following registers are sampled on a PMU snapshot Capture event:

- If FEAT_PMUv3_EXT32 is implemented: PMCID1SR, PMCID2SR, PMPCSR, and PMVIDSR.
- If FEAT_PMUv3_EXT64 is implemented: PMCCIDSR, PMPCSR, and PMVCIDSR.

SS	Meaning
0b0	Sample on Read.
0b1	Sample on Snapshot.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bits [3:2]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMP, bit [1]

Profiling enable implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IMP	Meaning
`0b0`	PMPCSCTL.EN reads-as-zero and ignores writes.
`0b1`	PMPCSCTL.EN is a read-write control bit.

Access to this field is RO.

EN, bit [0]

When PMU.PMPCSCTL.IMP == '1':

PC Sample-based Profiling Enable.

EN	Meaning
`0b0`	PC Sample-based Profiling is suspended.
`0b1`	PC Sample-based Profiling is active.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RAZ/WI.

Accessing PMPCSCTL

Accesses to this register use the following encodings:

When FEAT_PCSRv8p9 is implemented

Accessible at offset `0xE50` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.47 PMCCR, PMU Configuration Control Register

The PMCCR characteristics are:

Purpose

Contains PMU configuration controls.

Configuration

This register is present only when FEAT_PMUv3_EXTPMN is implemented. Otherwise, direct accesses to PMCCR are RES0.

Attributes

PMCCR is a 64-bit register.

Field descriptions

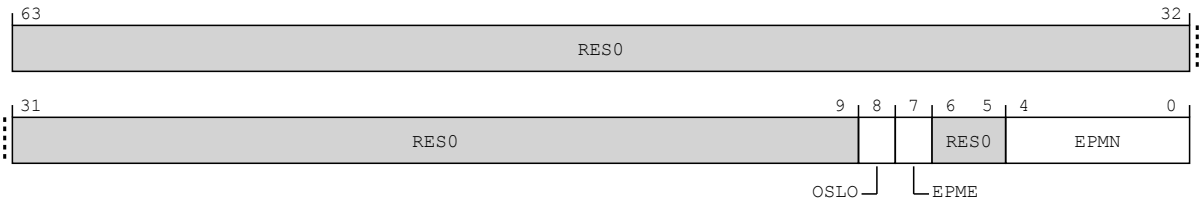

Bits [63:9]

Reserved, RES0.

OSLO, bit [8]

When FEAT_PMUv3_EXTPMN is implemented:

OS Lock Override.

OSLO	Meaning
0b0	No external access to any Performance Monitor register is affected by this control.
0b1	For the purpose of determining the access permissions of Performance Monitor registers, an external access that is a Most secure access ignores OSLSR_EL1.OSLK.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

EPME, bit [7]

When FEAT_PMUv3_EXTPMN is implemented:

External Enable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPME	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are the event counters in the third range.

Other event counters, PMCCNTR_EL0, and, if FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 are not affected by this field.

If the Effective value of PMCCR.EPMN is equal to NUM_PMU_COUNTERS, then this field has no effect.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

EPMN, bits [4:0]

When FEAT_PMUv3_EXTPMN is implemented:

Defines the number of event counters PMEVCNTR<n>_EL0 and, if FEAT_PMUv3_SS is implemented, snapshot registers PMEVCNTSVR<n>_EL1, that are reserved for external use.

PMCCR.EPMN divides the event counters into event counters that are accessible from self-hosted software, and which might be further divided into first and second ranges by MDCR_EL2.HPMN, and a third range that is inaccessible from self-hosted software.

If PMCCR.EPMN is not 0 and is less than the number of PMU event counters implemented by the PE, NUM_PMU_COUNTERS, then event counters [0..(PMCCR.EPMN-1)] are in the first and second ranges, and event counters [PMCCR.EPMN..(NUM_PMU_COUNTERS-1)] are in the third range.

If PMCCR.EPMN is equal to NUM_PMU_COUNTERS, or FEAT_PMUv3_EXTPMN is not implemented, then all of the following apply:

- All counters are in the first and second ranges.
- No counters are in the third range.

If PMCCR.EPMN is zero, then all of the following apply:

- No counters are in the first and second ranges.
- All counters are in the third range.

All the following apply for an event counter PMEVCNTR<n>_EL0 in the third range:

- The counter is accessible to a Most secure access of PMEVCNTR<n>_EL0. That is, an external access which is one of the following:
  - Root access, when FEAT_RME is implemented.
  - Secure access, when FEAT_RME is not implemented and Secure state is implemented.
  - Non-secure access, otherwise.
- The counter is not accessible to other external accesses, or as the System register PMEVCNTR<n>_EL0 at any Exception level.
- The counter overflow flag PMOVSSET_EL0[n] is set on unsigned overflow of PMEVCNTR<n>_EL0[63:0].
- PMCCR.EPME and PMCNTENSET_EL0 enable operation of the event counter.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_PMUv3_SS is implemented, then all of the following apply for an event counter snapshot register PMEVCNTSVR<n>_EL1 in the third range:

- The event counter snapshot register is accessible to a Most secure access of PMEVCNTSVR<n>_EL1.
- The event counter snapshot register is not accessible to other external accesses, or as the System register PMEVCNTSVR<n>_EL1 at any Exception level.

For information about counters in the first and second ranges, see the description of MDCR_EL2.HPMN.

Values greater than `NUM_PMU_COUNTERS` are reserved.

If this field is set to a reserved value, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by an external read of PMCCR.EPMN is UNKNOWN and less than or equal to 31.
- For the purpose of indirect reads of PMCCR.EPMN as a result of the following reads, the Effective value of PMCCR.EPMN is UNKNOWN and less than or equal to 31:
  - Direct reads of PMCR_EL0.N or PMCR.N.
  - Direct reads of MDCR_EL2.HPMN.
  - External reads of PMCFGR.N.
  - If FEAT_PMUv3_ICNTR is implemented, external reads of PMCGCR0.CG0NC.
- For all other purposes the Effective value of PMCCR.EPMN is UNKNOWN and less than or equal to `NUM_PMU_COUNTERS`.

If FEAT_PMUv3_EXTPMN is not implemented, then the Effective value of PMCCR.EPMN is `NUM_PMU_COUNTERS`.

The reset behavior of this field is:

- On a Cold reset, this field resets to the expression `NUM_PMU_COUNTERS`.

Otherwise:

Reserved, RES0.

Accessing PMCCR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXTPMN is implemented

Accessible at offset `0xE58` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.48 PMPCSR, Program Counter Sample Register

The PMPCSR characteristics are:

Purpose

PC Sample-based Profiling Extension register that holds a sampled instruction address value.

Configuration

PMPCSR is in the Core power domain.

If FEAT_PMUv3_EXT32 is implemented, support for 64-bit atomic reads is IMPLEMENTATION DEFINED. If 64-bit atomic reads are implemented, a 64-bit read of PMPCSR has the same side-effect as a 32-bit read of PMCSR[31:0] followed by a 32-bit read of PMPCSR[63:32], returning the combined value. For example, if the PE is in Debug state, then a 64-bit atomic read returns bits[31:0] == `0xFFFFFFFF` and bits[63:32] UNKNOWN.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PCSRv8p2 is implemented. Otherwise, direct accesses to PMPCSR are RES0.

Attributes

PMPCSR is a 64-bit register.

Field descriptions

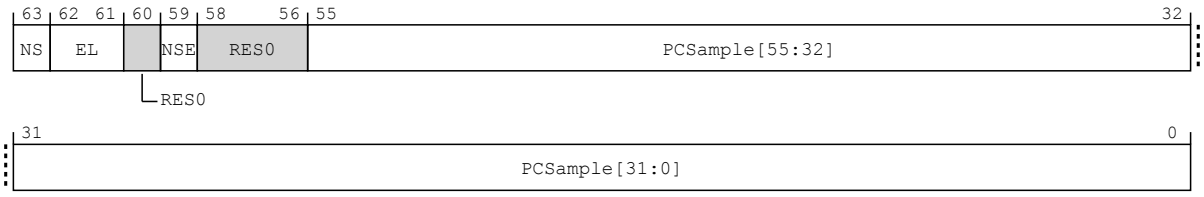

NS, bit [63]

When FEAT_RME is implemented:

Together with the NSE field, indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

NSE	NS	Meaning
`0b0`	`0b0`	When Secure state is implemented, Secure. Otherwise reserved.
`0b0`	`0b1`	Non-secure.
`0b1`	`0b0`	Root.
`0b1`	`0b1`	Realm.

Otherwise:

Non-secure state sample. Indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	Sample is from Secure state.
0b1	Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

EL	Meaning
0b00	Sample is from EL0.
0b01	Sample is from EL1.
0b10	Sample is from EL2.
0b11	Sample is from EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [60]

Reserved, RES0.

NSE, bit [59]

When FEAT_RME is implemented:

Together with the NS field, indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

For a description of the values derived by evaluating NS and NSE together, see PMPCSR.NS.

Otherwise:

Reserved, RES0.

Bits [58:56]

Reserved, RES0.

PCSample[55:32], bits [55:32]

Bits[55:32] of the sampled instruction address value. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PCSample[31:0], bits [31:0]

Bits[31:0] of the sampled instruction address value.

PMPCSR[31:0] reads as 0xFFFFFFFF when any of the following are true:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of PMPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

PMPCSR[31:0] reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of PMPCSR[31:0].

For the cases where a read of PMPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR to UNKNOWN values.

Otherwise, a read of PMPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

For a read of PMPCSR[31:0] from the memory-mapped interface, if PMLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMPCSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Note

A 32-bit access to PMPCSR[63:32] does not update the PC sample registers. Only a 64-bit access to PMPCSR[63:0] or a 32-bit access to PMPCSR[31:0] updates the PC sample registers. This includes the value a subsequent 32-bit read of PMPCSR[63:32] will return.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x200` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[31:0] Accessible at offset `0x200` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[63:32] Accessible at offset `0x204` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x220` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[31:0] Accessible at offset `0x220` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[63:32] Accessible at offset `0x224` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.49 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see 'About the Peripheral identification scheme'.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR0. Otherwise, direct accesses to PMPIDR0 are RES0.

Attributes

PMPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - PMPIDR1.PART_1 contains part number bits [11:8].
  - PMPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - PMPIDR1.PART_1 contains part number bits [15:12].
  - PMPIDR0.PART_0 contains part number bits [11:4].
  - PMPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, PMPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.50 PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see 'About the Peripheral identification scheme'.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR1. Otherwise, direct accesses to PMPIDR1 are RES0.

Attributes

PMPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- PMPIDR1.DES_0: JEP106 identification code bits[3:0].
- PMPIDR2.DES_1: JEP106 identification code bits[6:4].
- PMPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - PMPIDR1.PART_1 contains part number bits [11:8].
  - PMPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - PMPIDR1.PART_1 contains part number bits [15:12].
  - PMPIDR0.PART_0 contains part number bits [11:4].
  - PMPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, PMPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.51 PMPIDR2, Performance Monitors Peripheral Identification Register 2

The PMPIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR2. Otherwise, direct accesses to PMPIDR2 are RES0.

Attributes

PMPIDR2 is a 32-bit register.

Field descriptions

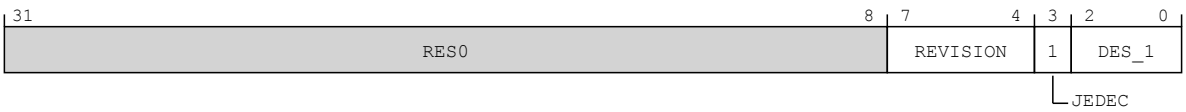

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- PMPIDR2.REVISION indicates the 4-bit revision number.
- PMPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- PMPIDR2.REVISION indicates the 4-bit major revision number.
- PMPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- PMPIDR2.REVISION contains part number bits [3:0].
- PMPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- PMPIDR1.DES_0: JEP106 identification code bits[3:0].
- PMPIDR2.DES_1: JEP106 identification code bits[6:4].
- PMPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing PMPIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.52 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR3. Otherwise, direct accesses to PMPIDR3 are RES0.

Attributes

PMPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- PMPIDR2.REVISION indicates the 4-bit revision number.
- PMPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- PMPIDR2.REVISION indicates the 4-bit major revision number.
- PMPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- PMPIDR2.REVISION contains part number bits [3:0].
- PMPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

PMPIDR3.CMOD might also indicate component modifications.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

PMPIDR3.REVAND might also indicate component modifications.

Access to this field is RO.

Accessing PMPIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFEC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.53 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see 'About the Peripheral identification scheme'.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR4. Otherwise, direct accesses to PMPIDR4 are RES0.

Attributes

PMPIDR4 is a 32-bit register.

Field descriptions

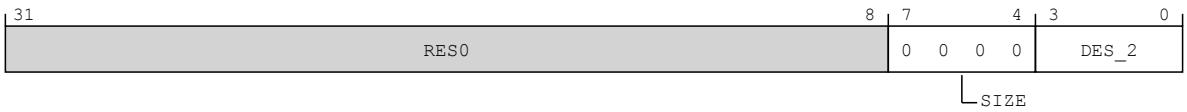

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- PMPIDR1.DES_0: JEP106 identification code bits[3:0].
- PMPIDR2.DES_1: JEP106 identification code bits[6:4].
- PMPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing PMPIDR4

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFD0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.54 PMSSCR_EL1, Performance Monitors Snapshot Status and Capture Register

The PMSSCR_EL1 characteristics are:

Purpose

Holds status information about the captured counters and provides a mechanism for software to initiate a sample.

Configuration

External register PMSSCR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMSSCR_EL1[63:0].

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMSSCR_EL1 are RES0.

Attributes

PMSSCR_EL1 is a 64-bit register.

Field descriptions

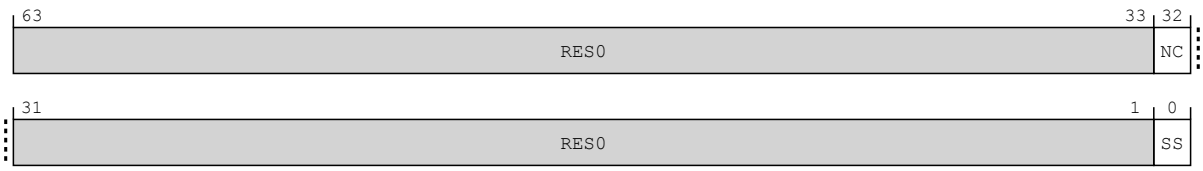

Bits [63:33]

Reserved, RES0.

NC, bit [32]

No Capture. Indicates whether the PMU counters have been captured.

NC	Meaning
0b0	PMU counters captured.
0b1	PMU counters not captured.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [31:1]

Reserved, RES0.

SS, bit [0]

Snapshot Capture and Status.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS	Meaning
`0b0`	On a read, the Capture event has completed.
`0b1`	On a read, the Capture event has not completed. On a write, request a Capture event.

A write of 0 to this field is ignored.

It is CONSTRAINED UNPREDICTABLE whether a Capture event has completed if this field is modified when the Capture event is ongoing.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- When PMU capture events are disabled, access to this field is RO.
- Otherwise, access to this field is RW.

Accessing PMSSCR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented

Accessible at offset `0xE30` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.55 PMSWINC_EL0, Performance Monitors Software Increment Register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see 'SW_INCR'.

Configuration

PMSWINC_EL0 is in the Core power domain.

If this register is implemented, use of it is deprecated.

If 1 is written to bit [n] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter n. This is consistent with not implementing the register in the external debug interface.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMSWINC_EL0[31:0].

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented, External register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMSWINC[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3p9 is not implemented, and an implementation implements PMSWINC_EL0.

Attributes

PMSWINC_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0
RES0																															

Bit [31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Event counter software increment bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
`0b0`	No action. The write to this bit is ignored.
`0b1`	It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter m.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMSWINC_EL0

If FEAT_PMUv3p9 or FEAT_PMUv3_EXT64 are implemented, this location is used for accesses to PMZR_EL0.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented

Accessible at offset `0xCA0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.56 PMVIDSR, VMID Sample Register

The PMVIDSR characteristics are:

Purpose

PC Sample-based Profiling Extension register that contains the sampled VMID value that is captured on reading PMPCSR[31:0].

Configuration

PMVIDSR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented, FEAT_PCSRv8p2 is implemented, and EL2 is implemented.

Attributes

PMVIDSR is a 32-bit register.

Field descriptions

31 … 16	15 … 8	7 … 0
RES0	VMID[15:8]	VMID

Bits [31:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When FEAT_VMID16 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - EL2 is disabled in the current Security state.
  - The PE is executing at EL2.
  - The PE is executing at EL0, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.

Because the value written to PMVIDSR is an indirect read of the VMID value, it is CONSTRAINED UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An instruction that writes to the VMID value.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMVIDSR

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMVCIDSR[63:32].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x20C` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.57 PMVCIDSR, CONTEXTIDR_EL1 and VMID Sample Register

The PMVCIDSR characteristics are:

Purpose

PC Sample-based Profiling Extension register that contains the sampled CONTEXTIDR_EL1 and VMID values that are captured on reading PMPCSR.

Configuration

External register PMVCIDSR bits [31:0] are architecturally mapped to External register PMCCIDSR[31:0].

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMVCIDSR is a 64-bit register.

Field descriptions

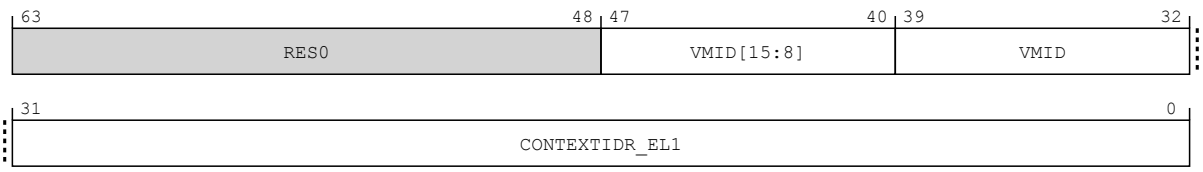

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [39:32]

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - EL2 is disabled in the current Security state.
  - The PE is executing at EL2.
  - The PE is executing at EL0, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Because the value written to PMVIDSR is an indirect read of the VMID value, it is CONSTRAINED UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:

- An instruction that writes to the VMID value.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and this register samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMVCIDSR

If FEAT_PCSRv8p2 and FEAT_PMUv3_EXT32 are implemented, then the same content is present in the same locations, and can be accessed using PMVIDSR[31:0] and PMCID1SR[31:0].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0x208` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.58 PMZR_EL0, Performance Monitors Zero with Mask

The PMZR_EL0 characteristics are:

Purpose

Zero the set of counters specified by the mask written to PMZR_EL0.

Configuration

PMZR_EL0 is in the Core power domain.

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is implemented, External register PMZR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMZR_EL0[63:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is implemented.

Attributes

PMZR_EL0 is a 64-bit register.

Field descriptions

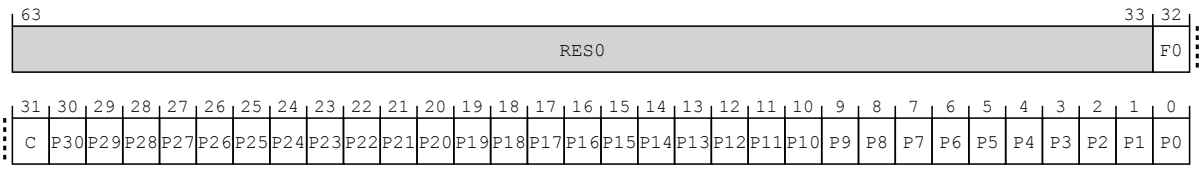

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Zero PMICNTR_EL0.

F0	Meaning
0b0	Write is ignored.
0b1	Set PMICNTR_EL0 to zero.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Otherwise:

Reserved, RES0.

C, bit [31]

Zero PMCCNTR_EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	Write is ignored.
0b1	Set PMCCNTR_EL0 to zero.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

P<m> , bits [m], for m = 30 to 0

Zero PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	Write is ignored.
0b1	Set PMEVCNTR<m>_EL0 to zero.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(addrdesc), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMZR_EL0

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is not implemented, then this location might be used for PMSWINC_EL0. If PMSWINC_EL0 is not implemented, then accesses to this location are RES0.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is implemented

Accessible at offset 0xCA0 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR().OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.4 External Activity Monitors Extension registers summary

The memory-mapped interface to the Activity Monitors Extension registers provides read-only access to:

- Read-only copies of the Activity Monitors Extension System registers, with the exception of AMUSERENR.
- An implementation identification register, AMIIDR.
- If they are implemented, the OPTIONAL Activity Monitors CoreSight and ID registers.

The locations of the registers are defined as offsets from a base address. The base address of the memory-mapped view must be aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. Activity Monitors external register views defines this memory map.

I5.4.1 Activity Monitors external register views

The following tables show the external view of the Activity Monitors registers. All implemented registers are RO. Offsets within the 4KB regions not defined in these tables are RAZ/WI.

- Table I5-159 shows the external view of the Activity Monitors registers when FEAT_AMU_EXT32 is implemented.
- Table I5-160 shows the external view of the Activity Monitors registers when FEAT_AMU_EXT64 is implemented.
- Table I5-161 shows the Peripheral and Component Identification Registers AMPIDR<n>, and AMCIDR<n>. The offsets of these registers are the same in both FEAT_AMU_EXT32 and FEAT_AMU_EXT64.

Each entry in the Name column links to the register description in Activity Monitors external register descriptions, and:

- If the System register? column of the table shows that the register is a System register, the memory-mapped interface provides a view of the System register described in:
  - — Activity Monitors registers, for the AArch64 System register.
  - — Activity Monitors registers, for the AArch32 System register.
- Otherwise, the register is accessible only using the external memory-mapped interface.

Table I5-159 Activity Monitors external register views when FEAT_AMU_EXT32 is implemented

Name	Description	System register?	Size	Offset
AMEVCNTR0<n>[31:0] AMEVCNTR0<n>[63:32]	Activity Monitor Event Counter registers 0	Yes	64	`0x000+8n` `0x004+8n`
AMEVCNTR1<n>[31:0] AMEVCNTR1<n>[63:32]	Activity Monitor Event Counter registers 1	Yes	64	`0x100+8n` `0x104+8n`
AMEVTYPER0<n>	Activity Monitor Event Type registers 0	Yes	32	`0x400+4n`
AMEVTYPER1<n>	Activity Monitor Event Type registers 1	Yes	32	`0x480+4n`
AMCNTENSET0	Activity Monitors Counter Enable Set register 0	Yes	32	`0xC00`
AMCNTENSET1	Activity Monitors Counter Enable Set register 1	Yes	32	`0xC04`
AMCNTENCLR0	Activity Monitors Counter Enable Clear register 0	Yes	32	`0xC20`
AMCNTENCLR1	Activity Monitors Counter Enable Clear register 1	Yes	32	`0xC24`
AMCGCR	Activity Monitors Counter Group Configuration Register	Yes	32	`0xCE0`
AMCFGR	Activity Monitors Configuration Register	Yes	32	`0xE00`
AMCR	Activity Monitors Control Register	Yes	32	`0xE04`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Description	System register?	Size	Offset
AMIIDR	Activity Monitors Implementation Identification Register	No	32	0xE08
AMDEVAFF0[a]	Device Affinity registers	No	32	0xFA8
AMDEVAFF1[a]		No	32	0xFAC
AMDEVARCH[a]	Device Architecture register	No	32	0xFBC
AMDEVTYPE[a]	Device Type register	No	32	0xFCC

a. CoreSight interface registers, see Management registers and CoreSight compliance.

Table I5-160 Activity Monitors external register views when FEAT_AMU_EXT64 is implemented

Name	Description	System register?	Size	Offset
AMEVCNTR0<n>[63:0]	Activity Monitor Event Counter registers 0	Yes	64	0x000+8n
AMEVCNTR1<n>[63:0]	Activity Monitor Event Counter registers 1	Yes	64	0x100+8n
AMEVTYPER0<n>	Activity Monitor Event Type registers 0	Yes	64	0x400+8n
AMEVTYPER1<n>	Activity Monitor Event Type registers 1	Yes	64	0x500+8n
AMCNTENSET	Activity Monitors Counter Enable Set register	Yes	64	0xC00
AMCNTEN	Activity Monitors Counter Enable register	Yes	64	0xC10
AMCNTENCLR	Activity Monitors Counter Enable Clear register	Yes	64	0xC20
AMCGCR	Activity Monitors Counter Group Configuration Register	Yes	64	0xCE0
AMCFGR	Activity Monitors Configuration Register	Yes	64	0xE00
AMIIDR	Activity Monitors Implementation Identification Register	No	64	0xE08
AMCR	Activity Monitors Control Register	Yes	64	0xE10
AMDEVAFF[a]	Device Affinity registers	No	64	0xFA8
AMDEVARCH[a]	Device Architecture register	No	32	0xFBC
AMDEVTYPE[a]	Device Type register	No	32	0xFCC

a. CoreSight interface registers, see Management registers and CoreSight compliance.

Table I5-161 Activity Monitors external register views of Peripheral and Component Identification Registers

Name	Description	System register?	Size	Offset
AMPIDR4[a]	Peripheral ID registers	No	32	0xFD0
AMPIDR0[a]		No	32	0xFE0
AMPIDR1[a]		No	32	0xFE4
AMPIDR2[a]		No	32	0xFE8
AMPIDR3[a]		No	32	0xFEC
AMCIDR0[a]	Component ID registers	No	32	0xFF0
AMCIDR1[a]		No	32	0xFF4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Description	System register?	Size	Offset
AMCIDR2[a]		No	32	`0xFF8`
AMCIDR3[a]		No	32	`0xFFC`

a. CoreSight interface registers, see Management registers and CoreSight compliance.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5 Activity Monitors external register descriptions

This section lists the external Activity Monitors registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

Configuration

It is IMPLEMENTATION DEFINED whether AMCFGR is implemented in the Core power domain or in the Debug power domain

When FEAT_AMU_EXT32 is implemented, External register AMCFGR bits [31:0] are architecturally mapped to AArch64 System register AMCFGR_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMCFGR bits [63:0] are architecturally mapped to AArch64 System register AMCFGR_EL0[63:0].

External register AMCFGR bits [31:0] are architecturally mapped to AArch32 System register AMCFGR[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCFGR are RES0.

Attributes

AMCFGR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

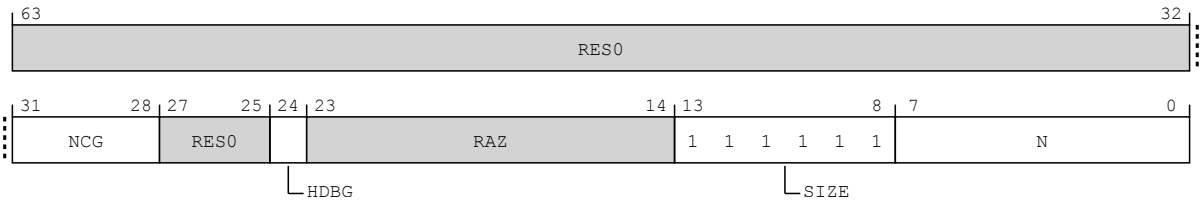

Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NCG	Meaning
`0b0001`	Two counter groups implemented.

All other values are reserved.

Access to this field is RO.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

This feature must be supported, and so this bit is 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HDBG	Meaning
`0b0`	AMCR.HDBG is RES0.
`0b1`	AMCR.HDBG is read/write.

Access to this field is RO.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of the activity monitor event counters, minus one.

The counters are 64-bit, so the value of this field is `0b111111`.

This field is used by software to determine the spacing of the counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as `0b111111`

Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters implemented in all groups. One less than the sum of AMCFGR.CG1NC and AMCFGR.CG0NC.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

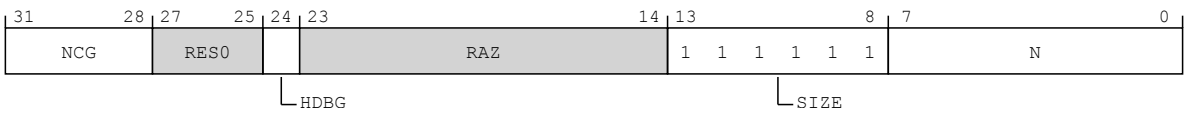

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
`0b0000`	One counter group implemented.
`0b0001`	Two counter groups implemented.

All other values are reserved.

Access to this field is RO.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

This feature must be supported, and so this bit is 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HDBG	Meaning
`0b0`	AMCR.HDBG is RES0.
`0b1`	AMCR.HDBG is read/write.

Access to this field is RO.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of the activity monitor event counters, minus one.

The counters are 64-bit, so the value of this field is `0b111111`.

This field is used by software to determine the spacing of the counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as `0b111111`

Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters implemented in all groups. One less than the sum of AMCFGR.CG1NC and AMCFGR.CG0NC.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMCFGR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xE00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xE00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configuration

It is IMPLEMENTATION DEFINED whether AMCGCR is implemented in the Core power domain or in the Debug power domain

When FEAT_AMU_EXT32 is implemented, External register AMCGCR bits [31:0] are architecturally mapped to AArch64 System register AMCGCR_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMCGCR bits [63:0] are architecturally mapped to AArch64 System register AMCGCR_EL0[63:0].

External register AMCGCR bits [31:0] are architecturally mapped to AArch32 System register AMCGCR[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCGCR are RES0.

Attributes

AMCGCR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

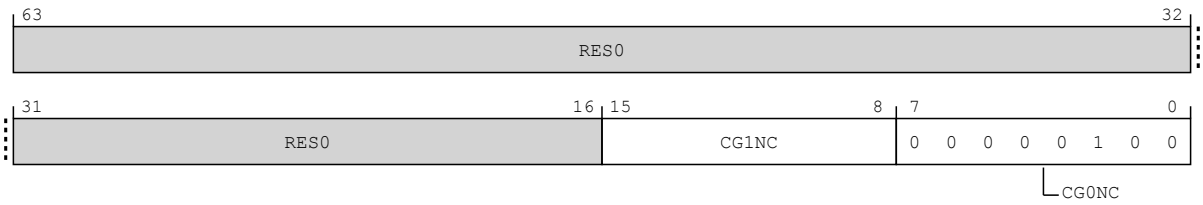

Bits [63:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. Highest index of the implemented counters in the auxiliary counter group, plus one. Zero indicates that there are no counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16.

If FEAT_AMUv1p1 is implemented, then AMCG1IDR_EL0 further defines which auxiliary counters are implemented, and the counters in the auxiliary counter group might be implemented non-contiguously.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reads as `0x04`

Access to this field is RO.

Otherwise:

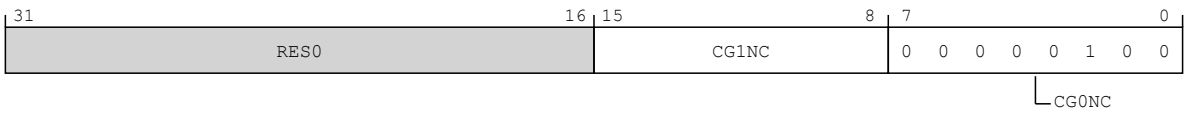

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. Highest index of the implemented counters in the auxiliary counter group, plus one. Zero indicates that there are no counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16.

If FEAT_AMUv1p1 is implemented, then AMCG1IDR_EL0 further defines which auxiliary counters are implemented, and the counters in the auxiliary counter group might be implemented non-contiguously.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as `0x04`

Access to this field is RO.

Accessing AMCGCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xCE0` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xCE0` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.3 AMCIDR0, Activity Monitors Component Identification Register 0

The AMCIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR0, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR0 are RES0.

Attributes

AMCIDR0 is a 32-bit register.

Field descriptions

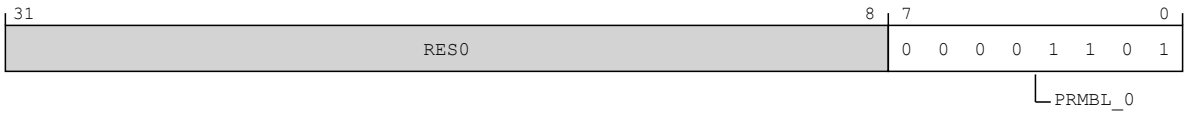

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing AMCIDR0

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFF0` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.4 AMCIDR1, Activity Monitors Component Identification Register 1

The AMCIDR1 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR1, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR1 are RES0.

Attributes

AMCIDR1 is a 32-bit register.

Field descriptions

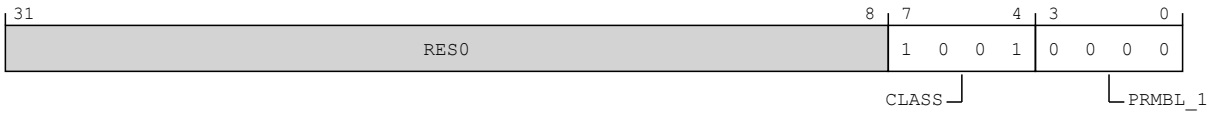

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMCIDR1

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFF4` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.5 AMCIDR2, Activity Monitors Component Identification Register 2

The AMCIDR2 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR2 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR2, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR2 are RES0.

Attributes

AMCIDR2 is a 32-bit register.

Field descriptions

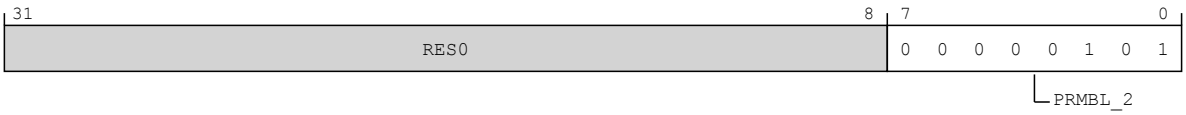

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing AMCIDR2

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFF8` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.6 AMCIDR3, Activity Monitors Component Identification Register 3

The AMCIDR3 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR3 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR3, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR3 are RES0.

Attributes

AMCIDR3 is a 32-bit register.

Field descriptions

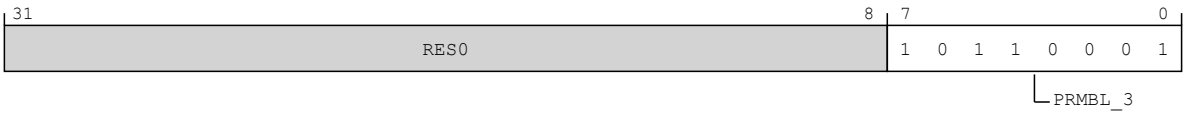

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing AMCIDR3

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFFC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.7 AMCNTEN, Activity Monitors Count Set and Clear Register

The AMCNTEN characteristics are:

Purpose

Control bits for the architected and auxiliary activity monitors event counters, AMEVCNTR0<n>. and AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTEN is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT64 is implemented. Otherwise, direct accesses to AMCNTEN are RES0.

Attributes

AMCNTEN is a 64-bit register.

Field descriptions

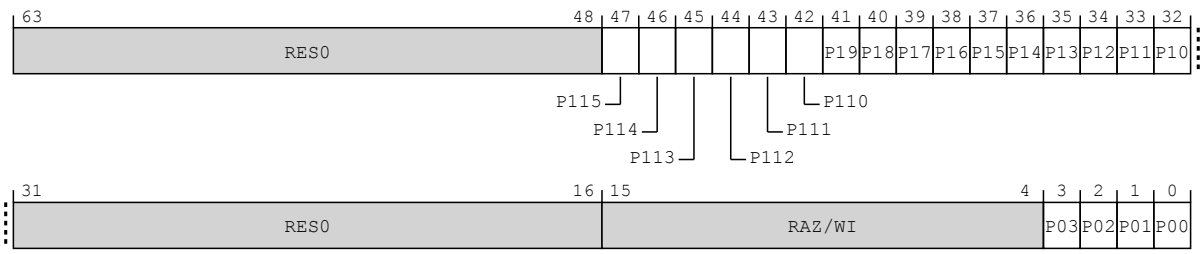

Bits [63:48]

Reserved, RES0.

P1<n> , bits [n+32], for n = 15 to 0

Activity monitor event counter control bit for AMEVCNTR1<n>.

Possible values of each bit are:

P1<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P0<n> , bits [n], for n = 3 to 0

Activity monitor event counter control bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P0<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTEN

If there are no auxiliary monitor event counters implemented, reads of AMCNTEN[63:32] are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xC10` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.8 AMCNTENCLR, Activity Monitors Count Enable Clear Register

The AMCNTENCLR characteristics are:

Purpose

Disable control bits for the architected and auxiliary activity monitors event counters, AMEVCNTR0<n> and AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENCLR is implemented in the Core power domain or in the Debug power domain

External register AMCNTENCLR bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMCNTENCLR bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT64 is implemented. Otherwise, direct accesses to AMCNTENCLR are RES0.

Attributes

AMCNTENCLR is a 64-bit register.

Field descriptions

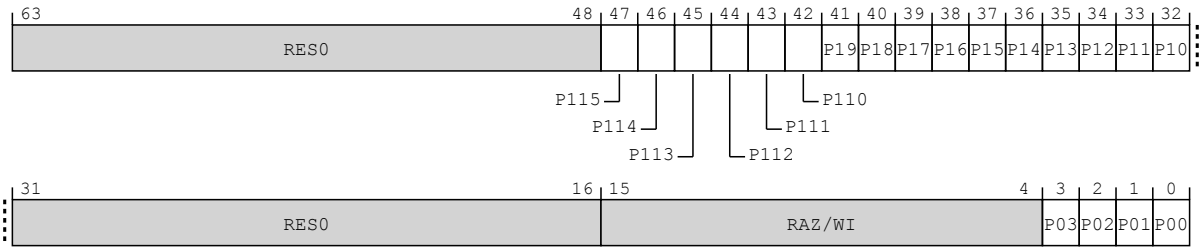

Bits [63:48]

Reserved, RES0.

P1<n> , bits [n+32], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P1<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression 0x00000000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P0<n> , bits [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P0<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR

If there are no auxiliary monitor event counters implemented, reads of AMCNTENCLR[63:32] are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xC20` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.9 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

Purpose

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENCLR0 is implemented in the Core power domain or in the Debug power domain

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET0[31:0].

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to External register AMCNTENSET0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT32 is implemented. Otherwise, direct accesses to AMCNTENCLR0 are RES0.

Attributes

AMCNTENCLR0 is a 32-bit register.

Field descriptions

31 … 16	15 … 4	3	2	1	0
RES0	RAZ/WI	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR0

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC20` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.10 AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENCLR1 is implemented in the Core power domain or in the Debug power domain

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCNTENCLR1 are RES0.

Attributes

AMCNTENCLR1 is a 32-bit register.

Field descriptions

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression 0x00000000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- When !IsG1ActivityMonitorImplemented(n), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR1

If there are no auxiliary monitor event counters implemented, reads of AMCNTENCLR1 are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC24` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.11 AMCNTENSET, Activity Monitors Count Enable Set Register

The AMCNTENSET characteristics are:

Purpose

Enable control bits for the architected and auxiliary activity monitors event counters, AMEVCNTR0<n> and AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENSET is implemented in the Core power domain or in the Debug power domain

External register AMCNTENSET bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMCNTENSET bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT64 is implemented. Otherwise, direct accesses to AMCNTENSET are RES0.

Attributes

AMCNTENSET is a 64-bit register.

Field descriptions

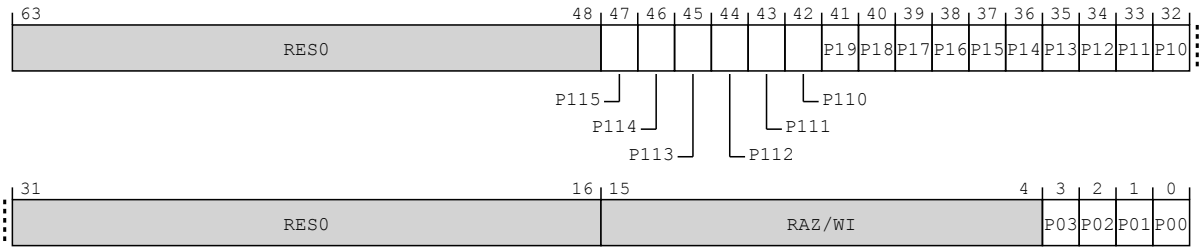

Bits [63:48]

Reserved, RES0.

P1<n> , bits [n+32], for n = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P1<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression 0x00000000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P0<n> , bits [n], for n = 3 to 0

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P0<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET

If there are no auxiliary monitor event counters implemented, reads of AMCNTENSET[63:32] are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xC00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.12 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENSET0 is implemented in the Core power domain or in the Debug power domain

External register AMCNTENSET0 bits [31:0] are architecturally mapped to External register AMCNTENCLR0[31:0].

External register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

External register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

External register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

External register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET0[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT32 is implemented. Otherwise, direct accesses to AMCNTENSET0 are RES0.

Attributes

AMCNTENSET0 is a 32-bit register.

Field descriptions

31 … 16	15 … 4	3	2	1	0
RES0	RAZ/WI	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x00000000`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET0

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.13 AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENSET1 is implemented in the Core power domain or in the Debug power domain

External register AMCNTENSET1 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

External register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

External register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

External register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

External register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT32 is implemented. Otherwise, direct accesses to AMCNTENSET1 are RES0.

Attributes

AMCNTENSET1 is a 32-bit register.

Field descriptions

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression 0x00000000.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- When !IsG1ActivityMonitorImplemented(n), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET1

If there are no auxiliary monitor event counters implemented, reads of AMCNTENSET1 are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC04` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.14 AMCR, Activity Monitors Control Register

The AMCR characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR is applicable to both the architected and the auxiliary counter groups.

Configuration

It is IMPLEMENTATION DEFINED whether AMCR is implemented in the Core power domain or in the Debug power domain

When FEAT_AMU_EXT32 is implemented, External register AMCR bits [31:0] are architecturally mapped to AArch64 System register AMCR_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMCR bits [63:0] are architecturally mapped to AArch64 System register AMCR_EL0[63:0].

External register AMCR bits [31:0] are architecturally mapped to AArch32 System register AMCR[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCR are RES0.

Attributes

AMCR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

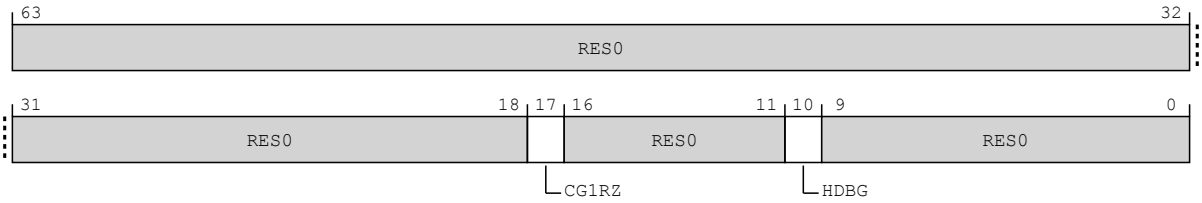

Bits [63:18]

Reserved, RES0.

CG1RZ, bit [17]

When FEAT_AMUv1p1 is implemented:

Counter Group 1 Read Zero.

CG1RZ	Meaning
0b0	System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.
0b1	If the current Exception level is the highest implemented Exception level, System register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Reads from the memory-mapped view of AMEVCNTR1<n> are unaffected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

HDBG	Meaning
0b0	Activity monitors do not halt counting when the PE is halted in Debug state.
0b1	Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Otherwise:

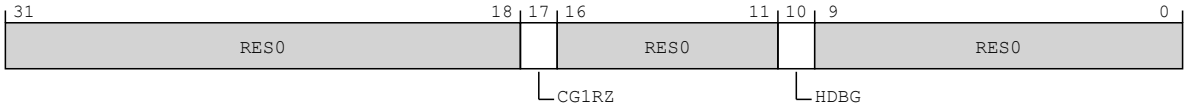

Bits [31:18]

Reserved, RES0.

CG1RZ, bit [17]

When FEAT_AMUv1p1 is implemented:

Counter Group 1 Read Zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CG1RZ	Meaning
0b0	System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.
0b1	If the current Exception level is the highest implemented Exception level, System register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

Note

Reads from the memory-mapped view of AMEVCNTR1<n> are unaffected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

HDBG	Meaning
0b0	Activity monitors do not halt counting when the PE is halted in Debug state.
0b1	Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing AMCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset 0xE04 from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xE10` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.15 AMDEVAFF, Activity Monitors Device Affinity Register

The AMDEVAFF characteristics are:

Purpose

Copy of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVAFF is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, FEAT_AMU_EXT64 is implemented, and an implementation implements AMDEVAFF1. Otherwise, direct accesses to AMDEVAFF are RES0.

Attributes

AMDEVAFF is a 64-bit register.

Field descriptions

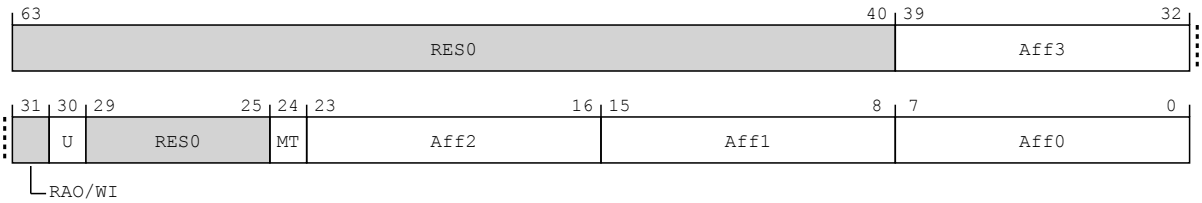

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMDEVAFF

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xFA8` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.16 AMDEVAFF0, Activity Monitors Device Affinity Register 0

The AMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVAFF0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, FEAT_AMU_EXT32 is implemented, and an implementation implements AMDEVAFF0. Otherwise, direct accesses to AMDEVAFF0 are RES0.

Attributes

AMDEVAFF0 is a 32-bit register.

Field descriptions

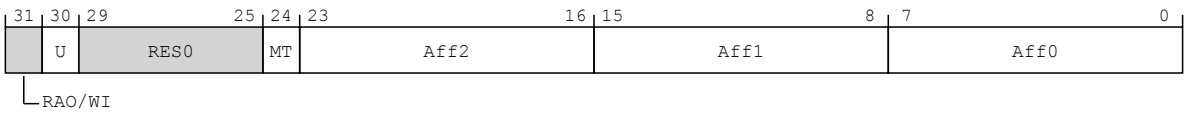

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMDEVAFF0

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset 0xFA8 from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.17 AMDEVAFF1, Activity Monitors Device Affinity Register 1

The AMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVAFF1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, FEAT_AMU_EXT32 is implemented, and an implementation implements AMDEVAFF1. Otherwise, direct accesses to AMDEVAFF1 are RES0.

Attributes

AMDEVAFF1 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	Aff3

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of AMDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMDEVAFF1

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFAC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.18 AMDEVARCH, Activity Monitors Device Architecture Register

The AMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the AMU component.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVARCH is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMDEVARCH, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMDEVARCH are RES0.

Attributes

AMDEVARCH is a 32-bit register.

Field descriptions

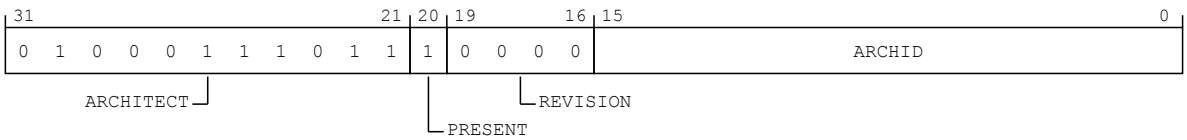

ARCHITECT, bits [31:21]

Defines the architect of the component. For Activity Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the AMDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

REVISION	Meaning
`0b0000`	Architecture revision is AMUv1.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARCHID, bits [15:0]

Defines this part to be an AMU component. For architectures defined by Arm this is further subdivided.

For AMU:

- Bits [15:12] are the architecture version, also identified as AMDEVARCH.ARCHVER.
- Bits [11:0] are the architecture part number, also identified as AMDEVARCH.ARCHPART.

AMDEVARCH.ARCHVER = `0x0`, which corresponds to AMU architecture version AMUv1.

If FEAT_AMU_EXT32 is implemented, AMDEVARCH is `0xA66`.

If FEAT_AMU_EXT64 is implemented, AMDEVARCH is `0xA67`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ARCHID	Meaning
`0x0A66`	AMUv1, with FEAT_AMU_EXT32 implemented.
`0x0A67`	AMUv1, with FEAT_AMU_EXT64 implemented.

Access to this field is RO.

Accessing AMDEVARCH

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xFBC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFBC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.19 AMDEVTYPE, Activity Monitors Device Type Register

The AMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's performance monitor interface.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVTYPE is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMDEVTYPE, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMDEVTYPE are RES0.

Attributes

AMDEVTYPE is a 32-bit register.

Field descriptions

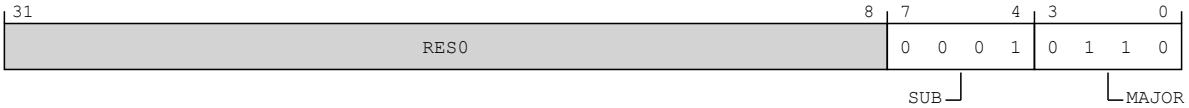

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype.

SUB	Meaning
0b0001	Component within a PE.

Access to this field is RO.

MAJOR, bits [3:0]

Major type.

MAJOR	Meaning
0b0110	Performance monitor component.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMDEVTYPE

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xFCC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFCC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.20 AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 3

The AMEVCNTR0<n> characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVCNTR0<n> is implemented in the Core power domain or in the Debug power domain

External register AMEVCNTR0<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR0<n>_EL0[63:0].

External register AMEVCNTR0<n> bits [31:0] are architecturally mapped to AArch32 System register AMEVCNTR0<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVCNTR0<n> are RES0.

Attributes

AMEVCNTR0<n> is a 64-bit register.

Field descriptions

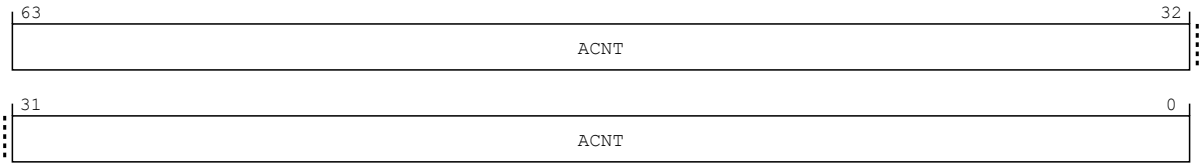

ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 3.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x0000000000000000`.

Accessing AMEVCNTR0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads of AMEVCNTR0<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.21 AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n> characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVCNTR1<n> is implemented in the Core power domain or in the Debug power domain

External register AMEVCNTR1<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR1<n>_EL0[63:0].

External register AMEVCNTR1<n> bits [31:0] are architecturally mapped to AArch32 System register AMEVCNTR1<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVCNTR1<n> are RES0.

Attributes

AMEVCNTR1<n> is a 64-bit register.

Field descriptions

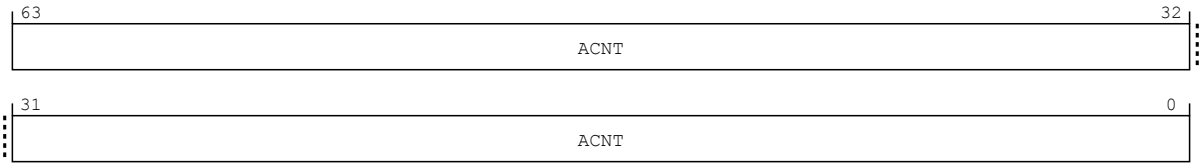

ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

The reset behavior of this field is:

- On a AMU reset, this field resets to the expression `0x0000000000000000`.

Accessing AMEVCNTR1<n>

If <n> is greater than or equal to AMCGCR.CG1NC or FEAT_AMUv1p1 is implemented and auxiliary counter <n> is not implemented, then reads of AMEVCNTR1<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

[63:0] Accessible at offset `0x100 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

[63:0] Accessible at offset `0x100 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.22 AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n> characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVTYPER0<n> is implemented in the Core power domain or in the Debug power domain

When FEAT_AMU_EXT32 is implemented, External register AMEVTYPER0<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMEVTYPER0<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[63:0].

External register AMEVTYPER0<n> bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER0<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are RES0.

Attributes

AMEVTYPER0<n> is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

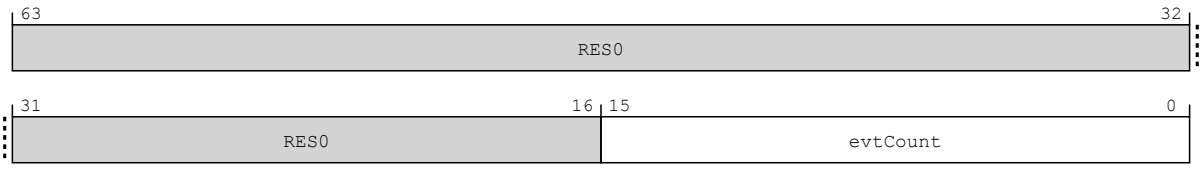

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

evtCount	Meaning	Applies when
0x0011	Processor frequency cycles.	n == 0
0x4004	Constant frequency cycles.	n == 1
0x0008	Instructions retired.	n == 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

evtCount	Meaning	Applies when
0x4005	Memory stall cycles.	n == 3

Access to this field is RO.

Otherwise:

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

evtCount	Meaning	Applies when
0x0011	Processor frequency cycles.	n == 0
0x4004	Constant frequency cycles.	n == 1
0x0008	Instructions retired.	n == 2
0x4005	Memory stall cycles.	n == 3

Access to this field is RO.

Accessing AMEVTYPER0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads of AMEVTYPER0<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0x400 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0x400 + (4 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.23 AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n> characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n> counts.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVTYPER1<n> is implemented in the Core power domain or in the Debug power domain

When FEAT_AMU_EXT32 is implemented, External register AMEVTYPER1<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER1<n>_EL0[31:0].

When FEAT_AMU_EXT64 is implemented, External register AMEVTYPER1<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVTYPER1<n>_EL0[63:0].

External register AMEVTYPER1<n> bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER1<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVTYPER1<n> are RES0.

Attributes

AMEVTYPER1<n> is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

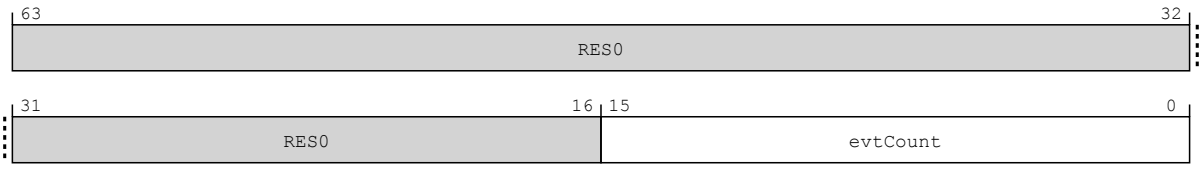

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMEVTYPER1<n>

If <n> is greater than or equal to AMCGCR.CG1NC or FEAT_AMUv1p1 is implemented and auxiliary counter <n> is not implemented, then reads of AMEVTYPER1<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0x480 + (4 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0x500 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.24 AMIIDR, Activity Monitors Implementation Identification Register

The AMIIDR characteristics are:

Purpose

Defines the implementer and revisions of the AMU.

Configuration

It is IMPLEMENTATION DEFINED whether AMIIDR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMIIDR are RES0.

Attributes

AMIIDR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

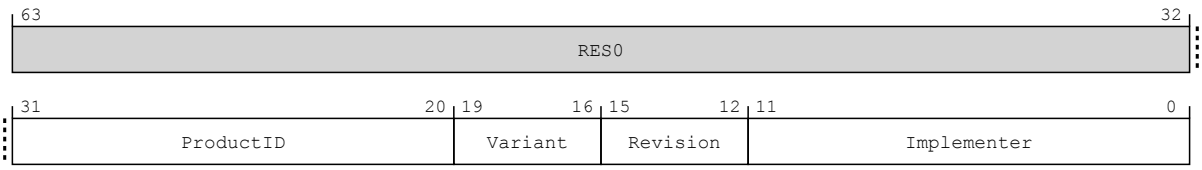

Bits [63:32]

Reserved, RES0.

ProductID, bits [31:20]

Part number of the AMU, bits [11:0]. The part number is selected by the designer of the component.

Matches the part number represented in the Peripheral ID registers AMPIDRn, if those registers are present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by AMIIDR.ProductID, or the major revision of the component. When defining a major revision, AMIIDR.Variant and AMIIDR.Revision together form the revision number of the component, with this field being the most significant part. When a component is changed, AMIIDR.Variant or AMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, AMIIDR.Revision should be set to `0b0000`.

This field has an IMPLEMENTATION DEFINED value.

Arm recommends this field matches the AMPIDR2.REVISION field, if present.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Revision, bits [15:12]

Component minor revision.

AMIIDR.Variant and AMIIDR.Revision together form the revision number of the component, with this field being the least significant part. When a component is changed, AMIIDR.Variant or AMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If AMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

This field has an IMPLEMENTATION DEFINED value.

Arm recommends this field matches the AMPIDR3.REVAND field, if present.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the AMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for AMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If AMPIDR4 is implemented, AMPIDR4.DES_2 matches bits [11:8] of this field.

If AMPIDR2 is implemented, AMPIDR2.DES_1 matches bits [6:4] of this field.

If AMPIDR1 is implemented, AMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Otherwise:

31 … 20	19 … 16	15 … 12	11 … 0
ProductID	Variant	Revision	Implementer

ProductID, bits [31:20]

Part number of the AMU, bits [11:0]. The part number is selected by the designer of the component.

Matches the part number represented in the Peripheral ID registers AMPIDRn, if those registers are present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by AMIIDR.ProductID, or the major revision of the component. When defining a major revision, AMIIDR.Variant and AMIIDR.Revision together form the revision number of the component, with this field being the most significant part. When a component is changed, AMIIDR.Variant or AMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, AMIIDR.Revision should be set to `0b0000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

Arm recommends this field matches the AMPIDR2.REVISION field, if present.

Access to this field is RO.

Revision, bits [15:12]

Component minor revision.

AMIIDR.Variant and AMIIDR.Revision together form the revision number of the component, with this field being the least significant part. When a component is changed, AMIIDR.Variant or AMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If AMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

This field has an IMPLEMENTATION DEFINED value.

Arm recommends this field matches the AMPIDR3.REVAND field, if present.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the AMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for AMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If AMPIDR4 is implemented, AMPIDR4.DES_2 matches bits [11:8] of this field.

If AMPIDR2 is implemented, AMPIDR2.DES_1 matches bits [6:4] of this field.

If AMPIDR1 is implemented, AMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Accessing AMIIDR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xE08` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xE08` from AMU

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.25 AMPIDR0, Activity Monitors Peripheral Identification Register 0

The AMPIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR0, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR0 are RES0.

Attributes

AMPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - AMPIDR1.PART_1 contains part number bits [11:8].
  - AMPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - AMPIDR1.PART_1 contains part number bits [15:12].
  - AMPIDR0.PART_0 contains part number bits [11:4].
  - AMPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, AMPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR0

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFE0` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.26 AMPIDR1, Activity Monitors Peripheral Identification Register 1

The AMPIDR1 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR1, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR1 are RES0.

Attributes

AMPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- AMPIDR1.DES_0: JEP106 identification code bits[3:0].
- AMPIDR2.DES_1: JEP106 identification code bits[6:4].
- AMPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
  - AMPIDR1.PART_1 contains part number bits [11:8].
  - AMPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
  - AMPIDR1.PART_1 contains part number bits [15:12].
  - AMPIDR0.PART_0 contains part number bits [11:4].
  - AMPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, AMPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR1

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFE4` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.27 AMPIDR2, Activity Monitors Peripheral Identification Register 2

The AMPIDR2 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR2 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR2, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR2 are RES0.

Attributes

AMPIDR2 is a 32-bit register.

Field descriptions

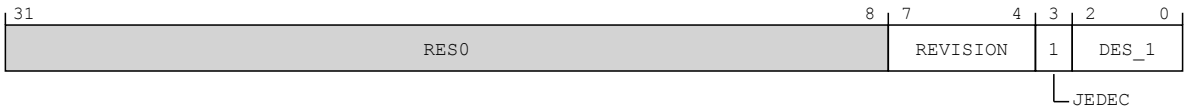

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- AMPIDR2.REVISION indicates the 4-bit revision number.
- AMPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- AMPIDR2.REVISION indicates the 4-bit major revision number.
- AMPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- AMPIDR2.REVISION contains part number bits [3:0].
- AMPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- AMPIDR1.DES_0: JEP106 identification code bits[3:0].
- AMPIDR2.DES_1: JEP106 identification code bits[6:4].
- AMPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing AMPIDR2

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFE8` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.28 AMPIDR3, Activity Monitors Peripheral Identification Register 3

The AMPIDR3 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR3 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR3, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR3 are RES0.

Attributes

AMPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- AMPIDR2.REVISION indicates the 4-bit revision number.
- AMPIDR3.REVAND indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- AMPIDR2.REVISION indicates the 4-bit major revision number.
- AMPIDR3.REVAND indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- AMPIDR2.REVISION contains part number bits [3:0].
- AMPIDR3.REVAND indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

AMPIDR3.CMOD might also indicate component modifications.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

AMPIDR3.REVAND might also indicate component modifications.

Access to this field is RO.

Accessing AMPIDR3

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFEC` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.29 AMPIDR4, Activity Monitors Peripheral Identification Register 4

The AMPIDR4 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR4 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR4, and FEAT_AMU_EXT. Otherwise, direct accesses to AMPIDR4 are RES0.

Attributes

AMPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- AMPIDR1.DES_0: JEP106 identification code bits[3:0].
- AMPIDR2.DES_1: JEP106 identification code bits[6:4].
- AMPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing AMPIDR4

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFD0` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR().RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR().RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR().RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR().NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.30 AMROOTCR, Activity Monitors Root Control Register

The AMROOTCR characteristics are:

Purpose

Control register for Root, Realm, Secure, and Non-secure access to External AMU registers.

Configuration

It is IMPLEMENTATION DEFINED whether AMROOTCR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMU_EXTACR is implemented and FEAT_RME is implemented. Otherwise, direct accesses to AMROOTCR are RES0.

Attributes

AMROOTCR is a 64-bit register.

Field descriptions

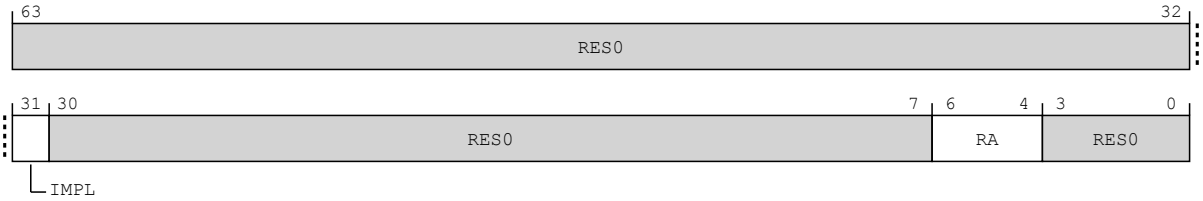

Bits [63:32]

Reserved, RES0.

IMPL, bit [31]

IMPL	Meaning
0b1	Indicates AMROOTCR is present.

Access to this field is RAO/WI.

Bits [30:7]

Reserved, RES0.

RA, bits [6:4]

Register Access to all External Activity Monitors registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RA	Meaning
`0b000`	Root register access is enabled. Access from other address spaces is disabled, meaning accesses to all External AMU registers are RAZ/WI.
`0b001`	Root and Realm register access is enabled. Access from other address spaces is disabled, meaning accesses to all External AMU registers are RAZ/WI.
`0b010`	Root and Secure register access is enabled. Access from other address spaces is disabled, meaning accesses to all External AMU registers are RAZ/WI.
`0b011`	Root, Secure, Non-secure and Realm register access is enabled.

Other values are reserved.

For the CoreSight management registers, `0xFA8` to `0xFFC`, it is IMPLEMENTATION DEFINED whether these registers are RO or RAZ/WI when register access is disabled by this field.

Bits [3:0]

Reserved, RES0.

Accessing AMROOTCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXTACR is implemented and FEAT_RME is implemented

Accessible at offset `0xE48` from AMU

- When an access is Root, accesses to this register are RW.
- Otherwise, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.31 AMSCR, Activity Monitors Secure Control Register

The AMSCR characteristics are:

Purpose

Control register for Secure, and Non-secure access to External AMU registers.

Configuration

It is IMPLEMENTATION DEFINED whether AMSCR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMU_EXTACR is implemented and FEAT_RME is not implemented. Otherwise, direct accesses to AMSCR are RES0.

Attributes

AMSCR is a 64-bit register.

Field descriptions

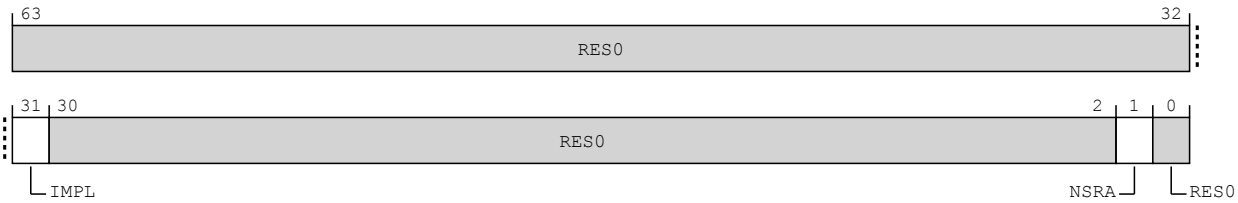

Bits [63:32]

Reserved, RES0.

IMPL, bit [31]

IMPL	Meaning
0b1	Indicates AMSCR is present.

Access to this field is RAO/WI.

Bits [30:2]

Reserved, RES0.

NSRA, bit [1]

Register Access to all External Activity Monitors registers.

NSRA	Meaning
0b0	Non-secure access is disabled, RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSRA	Meaning
`0b1`	Non-Secure access is enabled.

Bit [0]

Reserved, RES0.

Accessing AMSCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXTACR is implemented and FEAT_RME is not implemented

Accessible at offset `0xE40` from AMU

- When an access is Secure or an access is Root, accesses to this register are RW.
- Otherwise, accesses to this register are RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.6 Generic Timer memory-mapped registers overview

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBase*N* register frames, each with its own base address CNTBase*N*.
- For each CNTBase*N* register frame, if required, a CNTEL0Base*N* register frame, at base address CNTEL0Base*N*, that provides an EL0 view of the CNTBase*N* register frame.

For more information, see:

- Memory-mapped timer components.
- The CNTBaseN and CNTEL0BaseN frames. This section includes the memory map of the CNTBase*N* and CNTBase*N* register frames.
- The CNTCTLBase frame. This section includes the memory map of the CNTCTLBase register frame.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level implementation of the Generic Timer.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. Generic Timer memory-mapped registers overview gives an overview of these registers, and includes links to their memory maps.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

Purpose

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible by Non-secure accesses.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTACR<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implemented only if the value of CNTTIDR.Frame<n> is 1.

An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

Attributes

CNTACR<n> is a 32-bit register.

Field descriptions

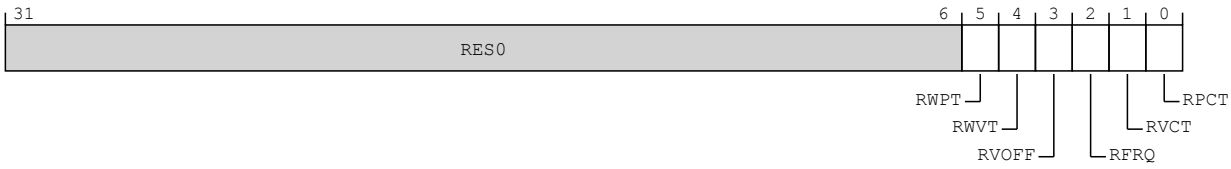

Bits [31:6]

Reserved, RES0.

RWPT, bit [5]

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>.

RWPT	Meaning
0b0	No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the EL1 Physical Timer registers in frame <n>.

The reset behavior of this field is:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RWVT, bit [4]

Read/write access to the Virtual Timer register CNTV_CVAL, CNTV_TVAL, and CNTV_CTL, in frame <n>.

RWVT	Meaning
0b0	No access to the Virtual Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the Virtual Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVOFF, bit [3]

Read-only access to CNTVOFF, in frame <n>.

RVOFF	Meaning
0b0	No access to CNTVOFF in frame <n>. The register is RES0.
0b1	Read-only access to CNTVOFF in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RFRQ, bit [2]

Read-only access to CNTFRQ, in frame <n>.

RFRQ	Meaning
0b0	No access to CNTFRQ in frame <n>. The register is RES0.
0b1	Read-only access to CNTFRQ in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVCT, bit [1]

Read-only access to CNTVCT, in frame <n>.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RVCT	Meaning
`0b0`	No access to CNTVCT in frame <n>. The register is RES0.
`0b1`	Read-only access to CNTVCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RPCT, bit [0]

Read-only access to CNTPCT, in frame <n>.

RPCT	Meaning
`0b0`	No access to CNTPCT in frame <n>. The register is RES0.
`0b1`	Read-only access to CNTPCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTACR<n>

In a system that supports the Realm Management Extension, CNTNSAR.NS<n> describes how these registers can be accessed by Root or Realm accesses.

In a system that recognizes two Security states:

- CNTACR<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTACR<n> is accessible by Non-secure accesses.

CNTACR<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x040 + (4 * n)`	CNTACR<n>

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

Purpose

Enables the counter, controls the counter resolution, and controls counter behavior during debug.

Configuration

It is IMPLEMENTATION DEFINED whether CNTCR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTCR is a 32-bit register.

Field descriptions

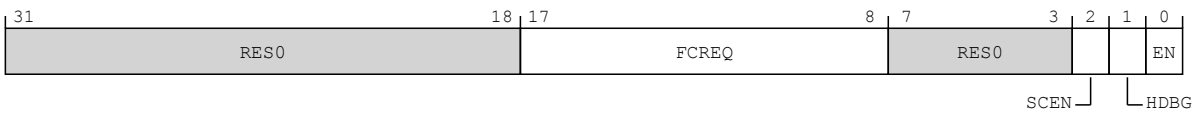

Bits [31:18]

Reserved, RES0.

FCREQ, bits [17:8]

Frequency change request. Indicates the number of the entry in the Frequency modes table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

The maximum number of entries in the Frequency modes table is IMPLEMENTATION DEFINED up to a maximum of 1004 entries, see 'The Frequency modes table'. An implementation is only required to implement an FCREQ field that can hold values from 0 to the highest supported Frequency modes table entry. Any unrequired most-significant bits of FCREQ can be implemented as RES0.

The reset behavior of this field is:

- On a Timer reset, this field resets to the expression `0x0`.

Bits [7:3]

Reserved, RES0.

SCEN, bit [2]

When FEAT_CNTSC is implemented:

Scale Enable.

SCEN	Meaning
`0b0`	Scaling is not enabled. The counter value is incremented by `0x1`.0000000 for each counter tick.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Scaling is enabled. The counter is incremented by CNTSCR.ScaleVal for each counter tick.

The SCEN bit can only be changed when the counter is disabled, when CNTCR.EN == 0.

If the value of CNTCR.SCEN changes when CNTCR.EN == 1, then:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

When the CNTCV register in the CNTControlBase frame of the memory mapped counter module is written to, the accumulated fraction information is reset to zero.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HDBG, bit [1]

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

HDBG	Meaning
0b0	System counter ignores Halt-on-debug.
0b1	Asserted Halt-on-debug signal halts system counter update.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enables the counter:

EN	Meaning
0b0	System counter disabled.
0b1	System counter enabled.

The reset behavior of this field is:

- On a Timer reset, this field resets to '0'.

Accessing CNTCR

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTCR can be accessed through the memory-mapped interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x000`	CNTCR

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

Purpose

Indicates the current count value.

Configuration

It is IMPLEMENTATION DEFINED whether CNTCV is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTCV is a 64-bit register.

Field descriptions

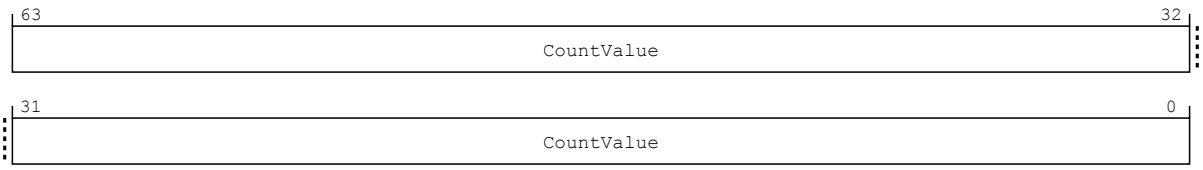

CountValue, bits [63:0]

Indicates the counter value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTCV

Frame	Accessibility
CNTControlBase	RW
CNTReadBase	RO

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the instance of the register in the CNTControlBase frame:

- In a system that supports the Realm Management Extension, this register is implemented only in the Root physical address space.
- In a system that supports Secure and Non-secure physical address spaces, this register is implemented only in the Secure physical address space.
- If the counter is enabled, the effect of writing to the register is UNKNOWN.

For the instance of the register in the CNTReadBase frame, this register is accessible in all physical address spaces.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTCV can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTControlBase	`0x008`	CNTCV	63:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTReadBase	`0x000`	CNTCV	63:0

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

Purpose

An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the CNTPCT, CNTVCT, CNTFRQ, EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTEL0ACR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implementation of this register is OPTIONAL.

Attributes

CNTEL0ACR is a 32-bit register.

Field descriptions

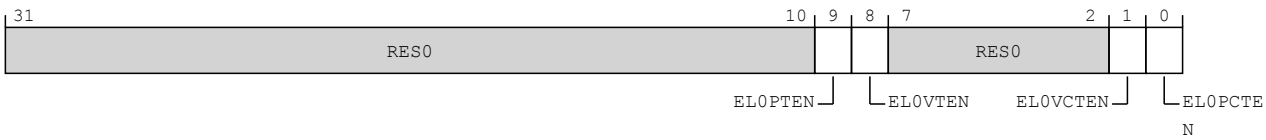

Bits [31:10]

Reserved, RES0.

EL0PTEN, bit [9]

Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the CNTP_CVAL, CNTP_TVAL, and CNTP_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

EL0PTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Second view read/write access control for the Virtual Timer registers. This bit controls whether the CNTV_CVAL, CNTV_TVAL, and CNTV_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0VTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bits [7:2]

Reserved, RES0.

EL0VCTEN, bit [1]

Second view read access control for CNTVCT and CNTFRQ.

EL0VCTEN	Meaning
0b0	CNTVCT is not visible in the second view. If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTVCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Second view read access control for CNTPCT and CNTFRQ.

EL0PCTEN	Meaning
0b0	CNTPCT is not visible in the second view. If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTPCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTEL0ACR

CNTEL0ACR can be implemented in any implemented CNTBaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

If CNTEL0ACR is not implemented in an implemented CNTBaseN frame:

- The register location in that frame is RAZ/WI.
- If the corresponding CNTEL0BaseN frame is implemented, the registers CNTFRQ, CNTP_CTL, CNTP_CVAL, CNTP_TVAL, CNTPCT, CNTV_CTL, CNTV_CVAL, CNTV_TVAL, and CNTVCT are not visible and accesses are RAZ/WI in that frame.

CNTEL0ACR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x014`	CNTEL0ACR

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

Purpose

Indicates the base frequency of the system counter.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID0 is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0. For more information, see 'The Frequency modes table'.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID0 is a 32-bit register.

Field descriptions

31	0
Frequency	

Frequency, bits [31:0]

The base frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID0

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTFID0 can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020`	CNTFID0

Accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.6 CNTFID<n>, Counter Frequency IDs, n > 0, n = 1 - 1003

The CNTFID<n> characteristics are:

Purpose

Indicates alternative system counter frequencies.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0, see ‘The Frequency modes table’.

The number of CNTFID<n> registers is IMPLEMENTATION DEFINED, and the only required CNTFID<n> register is CNTFID0.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED up to this limit. For an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, the maximum number of entries in the Frequency modes table is 40, including the zero-word end marker.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID<n> is a 32-bit register.

Field descriptions

31 … 0
Frequency

Frequency, bits [31:0]

A system counter frequency, in Hz. Must be an exact divisor of the base frequency. Arm strongly recommends that all frequency values in the Frequency modes table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID<n>

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes these registers, is implemented only in the Root physical address space.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes these registers, is implemented only in the Secure physical address space.

CNTFID<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020 + (4 * n)`	CNTFID<n>

Accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. The instance of the register in the CNTCTLBase frame must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFRQ is implemented in the Core power domain or in the Debug power domain

For more information see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions

31	0
Clock frequency	

ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ

CNTFRQ must be implemented as an RW register in the CNTCTLBase frame.

In a system that supports the Realm Management Extension, the instance of the register in the CNTCTLBase frame is accessible as follows:

- For Root accesses, it is IMPLEMENTATION DEFINED whether accesses to the register are permitted or behave as RES0.
- For Realm accesses, this register behaves as RES0.

In a system that recognizes two Security states, the instance of the register in the CNTCTLBase frame is only accessible by Secure accesses.

CNTFRQ can be implemented as a RO register in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For an implemented CNTBaseN frame:

- CNTFRQ is accessible in that frame, as a RO register, if the value of CNTACR<n>.RFRQ is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTFRQ is accessible as a RO register in that frame if both:
  - CNTFRQ is accessible in the corresponding CNTBaseN frame.
  - Either the value of CNTEL0ACR.EL0VCTEN is 1 or the value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

CNTFRQ can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x010`	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x010`	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x000`	CNTFRQ

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.8 CNTID, Counter Identification Register

The CNTID characteristics are:

Purpose

Indicates whether counter scaling is implemented.

Configuration

It is IMPLEMENTATION DEFINED whether CNTID is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTID are RES0.

Attributes

CNTID is a 32-bit register.

Field descriptions

31 … 4	3 … 0
RES0	CNTSC

Bits [31:4]

Reserved, RES0.

CNTSC, bits [3:0]

Indicates whether Counter Scaling is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CNTSC	Meaning
0b0000	Counter scaling is not implemented.
0b0001	Counter scaling is implemented.

All other values are reserved.

Access to this field is RO.

Accessing CNTID

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTID can be accessed through the memory-mapped interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x1C`	CNTID

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.9 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

Purpose

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

Configuration

It is IMPLEMENTATION DEFINED whether CNTNSAR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTNSAR is a 32-bit register.

Field descriptions

31	8	7	6	5	4	3	2	1	0
RES0		NS7	NS6	NS5	NS4	NS3	NS2	NS1	NS0

Bits [31:8]

Reserved, RES0.

NS<n> , bits [n], for n = 7 to 0

Non-secure access to frame n.

NS<n>	Meaning
0b0	Secure access only. Behaves as RES0 to Non-secure accesses. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether Root accesses to the specified registers are permitted or behave as RES0. For Realm accesses, the specified registers behave as RES0.
0b1	Secure and Non-secure accesses permitted. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether Root and Realm accesses to the specified registers are permitted. If not permitted, the specified registers behave as RES0 for Root and Realm accesses.

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF<n> are accessible to Non-secure accesses.

If frame CNTBase<n>:

- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.
- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTNSAR

In a system that supports the Realm Management Extension, this register is accessible as follows:

- For Root accesses, it is IMPLEMENTATION DEFINED whether accesses to the register are permitted or behave as RES0.
- For Realm accesses, this register behaves as RES0.

In a system that recognizes two Security states, this register is only accessible by Secure accesses.

CNTNSAR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x004`	CNTNSAR

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.10 CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTP_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTP_CTL is a 32-bit register.

Field descriptions

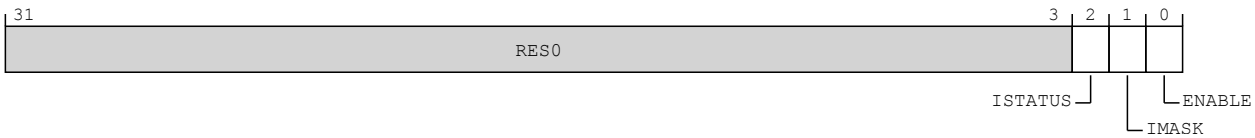

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CTL

CNTP_CTL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CTL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CTL is accessible in that frame if both:
  - CNTP_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTP_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x02C`	CNTP_CTL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x02C`	CNTP_CTL

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.11 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the EL1 physical timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTP_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions

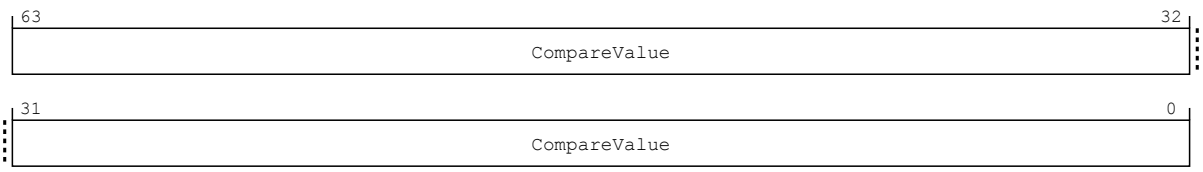

CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTP_CTL.IMASK is 0.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

CNTP_CVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTP_CVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CVAL is accessible in that frame if both:
  - CNTP_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

CNTP_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x020`	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x024`	CNTP_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x020`	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x024`	CNTP_CVAL	63:32

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.12 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTP_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTP_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CompareValue - CNTPCT).

On a write of this register, CompareValue is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL

CNTP_TVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_TVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_TVAL is accessible in that frame if both:
  - CNTP_TVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

CNTP_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x028`	CNTP_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x028`	CNTP_TVAL

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.13 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configuration

It is IMPLEMENTATION DEFINED whether CNTPCT is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTPCT is a 64-bit register.

Field descriptions

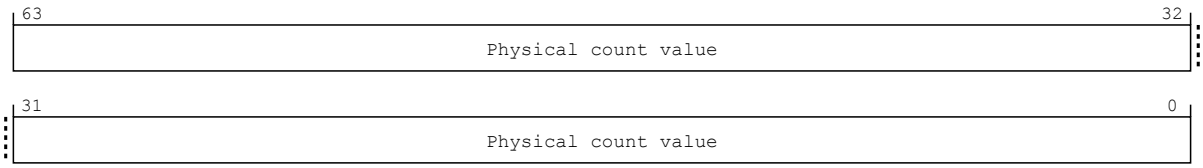

PhysicalCount, bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT

CNTPCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTPCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RPCT is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTPCT is accessible in that frame if both:
  - CNTPCT is accessible in the corresponding CNTBaseN frame.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - The value of CNTEL0ACR.EL0PCTEN is 1.
  - Otherwise, the CNTPCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

CNTPCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x000`	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x004`	CNTPCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x000`	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x004`	CNTPCT	63:32

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.14 CNTSCR, Counter Scale Register

The CNTSCR characteristics are:

Purpose

Enables the counter, controls the counter effective frequency setting, and controls counter behavior during debug.

Configuration

It is IMPLEMENTATION DEFINED whether CNTSCR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTSCR are RES0.

Attributes

CNTSCR is a 32-bit register.

Field descriptions

31	0
ScaleVal	

ScaleVal, bits [31:0]

Scale Value

When counter scaling is enabled, ScaleVal is the average amount added to the counter value for one period of the frequency of the Generic counter as described in the CNTFRQ register.

The actual rate of update of the counter value is determined by the counter resolution.

ScaleVal is expressed as an unsigned fixed point number with an 8-bit integer value and a 24-bit fractional value.

CNTSCR.ScaleVal can only be changed when CNTCR.EN == 0. If the value of this field is changed when CNTCR.EN == 1:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTSCR

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSCR can be accessed through the memory-mapped interface:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x10`	CNTSCR

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.15 CNTSR, Counter Status Register

The CNTSR characteristics are:

Purpose

Provides counter resolution status information.

Configuration

It is IMPLEMENTATION DEFINED whether CNTSR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTSR is a 32-bit register.

Field descriptions

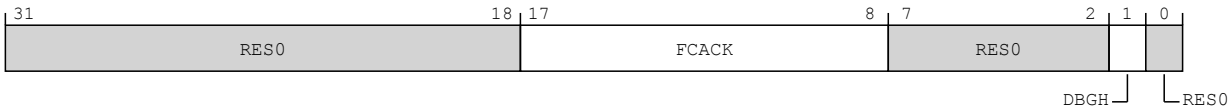

Bits [31:18]

Reserved, RES0.

FCACK, bits [17:8]

Frequency Change Acknowledge. Indicates the currently selected entry in the Frequency modes table, see 'The Frequency modes table'.

The reset behavior of this field is:

- On a Timer reset, this field resets to the expression `0x000000`.

Bits [7:2]

Reserved, RES0.

DBGH, bit [1]

Indicates whether the counter is halted because the Halt-on-debug signal is asserted:

DBGH	Meaning
`0b0`	Counter is not halted.
`0b1`	Counter is halted.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTSR

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x004`	CNTSR

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.16 CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

Purpose

Indicates the implemented timers in the physical address space, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

Configuration

It is IMPLEMENTATION DEFINED whether CNTTIDR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTTIDR is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
Frame7	Frame6	Frame5	Frame4	Frame3	Frame2	Frame1	Frame0

Frame<n> , bits [4n+3:4n], for n = 7 to 0

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2], the FEL0 subfield, indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

Bit[2]	Meaning
0b0	Frame<n> does not have a second view. The CNTEL0ACR register in the first view of the frame is RES0.
0b1	Frame<n> has a second view, CNTEL0Base<n>.

If bit[0] is 0, bit[2] is RES0.

Bit[1], the FVI subfield, indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers CNTV_CVAL, CNTV_TVAL, and CNTV_CTL.
- This CNTCTLBase frame implements the virtual timer offset register CNTVOFF<n>.

The possible values of bit[1] are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit[1]	Meaning
0b0	Frame<n> does not have virtual capability. The virtual time and offset registers are RES0.
0b1	Frame<n> has virtual capability. The virtual time and offset registers are implemented.

If bit[0] is 0, bit[1] is RES0.

Bit[0], the FI subfield, indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

Bit[0]	Meaning
0b0	Frame<n> is not implemented. All registers associated with the frame are RES0.
0b1	Frame<n> is implemented.

Accessing CNTTIDR

In a system that supports the Realm Management Extension, it is IMPLEMENTATION DEFINED whether Root and Realm accesses to this register are permitted. If not permitted, this register behaves as RES0 for Root and Realm accesses.

In a system that recognizes two Security states, this register is accessible by both Secure and Non-secure accesses.

CNTTIDR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x008	CNTTIDR

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.17 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTV_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions

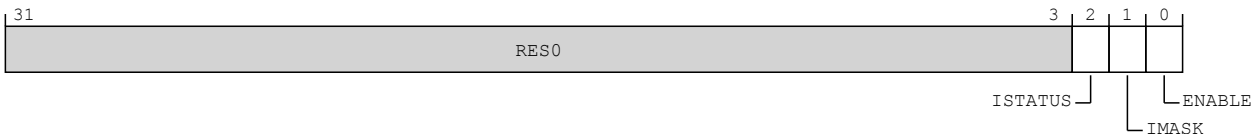

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CTL

CNTV_CTL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CTL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CTL is accessible in that frame if both:
  - CNTV_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTV_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x03C`	CNTV_CTL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x03C`	CNTV_CTL

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.18 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the virtual timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTV_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions

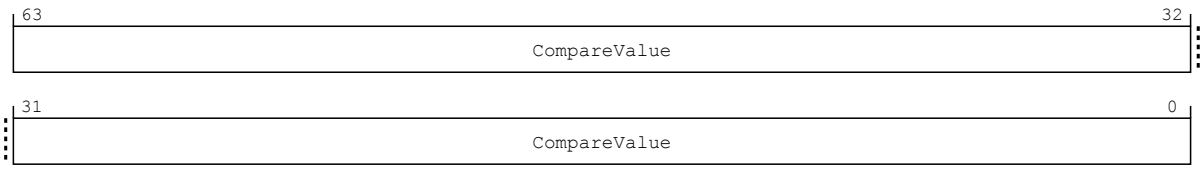

CompareValue, bits [63:0]

Holds the virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTV_CTL.IMASK is 0.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL

CNTV_CVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTV_CVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CVAL is accessible in that frame if both:
  - CNTV_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

CNTV_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x030`	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x034`	CNTV_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x030`	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x034`	CNTV_CVAL	63:32

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.19 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTV_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CompareValue - CNTVCT).

On a write of this register, CompareValue is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL

CNTV_TVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_TVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_TVAL is accessible in that frame if both:
  - CNTV_TVAL is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

CNTV_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x038`	CNTV_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x038`	CNTV_TVAL

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.20 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVCT is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTVCT is a 64-bit register.

Field descriptions

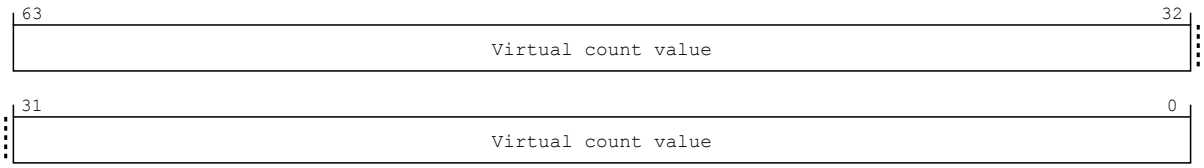

VirtualCount, bits [63:0]

Virtual count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT

CNTVCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTVCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVCT is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTVCT is accessible in that frame if both:
  - CNTVCT is accessible in the corresponding CNTBaseN frame.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The value of CNTEL0ACR.EL0VCTEN is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

CNTVCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x008`	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x00C`	CNTVCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x008`	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x00C`	CNTVCT	63:32

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.21 CNTVOFF, Counter-timer Virtual Offset

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset for a CNTBaseN frame that has virtual timer capability. This is the offset between real time and virtual time.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVOFF is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

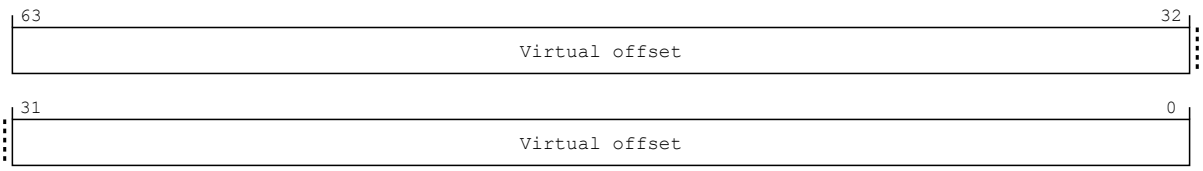

VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

CNTVOFF is implemented, as a RO register, in any implemented CNTBaseN frame that has virtual timer capability.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTVOFF is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVOFF is 1.
- Otherwise, the CNTVOFF address in that frame is RAZ/WI.

Note

CNTVOFF is never visible in any CNTEL0BaseN frame. This means that the CNTVOFF address in any implemented CNTEL0BaseN frame is RAZ/WI.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF register must be accessible as an atomic 64-bit value.

CNTVOFF can be accessed through the memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTBaseN	`0x018`	31:0

Accesses to this register are RO.

Component	Frame	Offset	Range
Timer	CNTBaseN	`0x01C`	63:32

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.22 CNTVOFF<n>, Counter-timer Virtual Offsets, n = 0 - 7

The CNTVOFF<n> characteristics are:

Purpose

Holds the 64-bit virtual offset for frame CNTBase<n>. This is the offset between real time and virtual time.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVOFF<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implementation of this register is OPTIONAL.

Attributes

CNTVOFF<n> is a 64-bit register.

Field descriptions

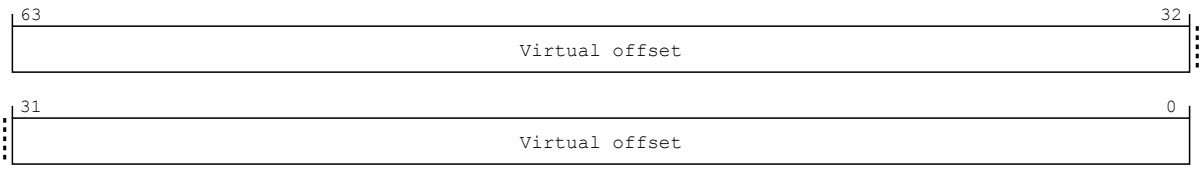

VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF<n>

In the CNTCTLBase frame a CNTVOFF<n> register must be implemented, as a RW register, for each CNTBaseN frame that has virtual timer capability. For more information, see 'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames'.

Note

The value of <n> in an instance of CNTVOFF<n> specifies the value of N for the associated CNTBaseN frame.

In a system that supports the Realm Management Extension, CNTNSAR.NS<n> describes how these registers can be accessed by Root or Realm accesses.

In a system that recognizes two Security states, for any CNTVOFF<n> register in the CNTCTLBase frame:

- CNTVOFF<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTVOFF<n> is accessible by Non-secure accesses.

The register location of any unimplemented CNTVOFF<n> register in the CNTCTLBase frame is RAZ/WI.

The CNTVOFF<n> register is accessible in the CNTBaseN frame using CNTVOFF.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF<n> registers must be accessible as atomic 64-bit values.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTVOFF<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x080 + (8 * n)`	31:0

Accesses to this register are RW.

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x084 + (8 * n)`	63:32

Accesses to this register are RW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.23 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

Purpose

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

Configuration

It is IMPLEMENTATION DEFINED whether CounterID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

These registers are implemented independently in each of the implemented Generic Timer memory-mapped frames.

If the implementation of the Counter ID registers requires an architecture version, the value for this version of the Arm Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of `0xF`.

Attributes

CounterID<n> is a 32-bit register.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing CounterID<n>

These registers must be implemented, as RO registers, in every implemented Generic Timer memory-mapped frame.

For the CNTCTLBase frame, in a system that supports the Realm Management Extension, it is IMPLEMENTATION DEFINED whether Root and Realm accesses to these registers are permitted. If not permitted, these registers behave as RES0 for Root and Realm accesses.

For the CNTCTLBase frame, in a system that recognizes two Security states these registers are accessible by both Secure and Non-secure accesses.

For the CNTControlBase frame, in a system that supports the Realm Management Extension, the frame is implemented only in the Root physical address space, meaning these registers are implemented only in the Root physical address space.

For the CNTControlBase frame, in a system that supports Secure and Non-secure physical address spaces, the frame is implemented only in the Secure physical address space, meaning these registers are implemented only in the Secure physical address space.

For the CNTReadBase frame, these registers are accessible in all physical address spaces.

For the CNTBaseN frames, ‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

CounterID<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTReadBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part J
Architectural Pseudocode

Chapter J1
A-profile Architecture Pseudocode

This chapter contains pseudocode that describes many features of the A-profile architecture. It contains the following sections:

- Standard library
- Pseudocode for AArch64 operation
- Pseudocode for AArch32 operation
- Shared pseudocode

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1 Standard library

The pseudocode in this manual is written in the Architecture Specification Language (ASL). The ASL makes use of a standard library.

Implementations of the standard library functions can be found within the herdtools7 github repository.

The standard library implements the following functions:

- `Abs(x: integer) => integer`
- `Abs(x: real) => real`
- `AlignDown{N}(x: bits(N), y: integer{1..N}) => bits(N)`
- `AlignDownP2(x: integer, p2: integer) => integer`
- `AlignDownP2{N}(x: bits(N), p2: integer{0..N}) => bits(N)`
- `AlignDownSize(x: integer, size: integer) => integer`
- `AlignDownSize{N}(x: bits(N), size: integer{1..2^N}) => bits(N)`
- `AlignUp{N}(x: bits(N), y: integer{1..N}) => bits(N)`
- `AlignUpP2(x: integer, p2: integer) => integer`
- `AlignUpP2{N}(x: bits(N), p2: integer{0..N}) => bits(N)`
- `AlignUpSize(x: integer, size: integer) => integer`
- `AlignUpSize{N}(x: bits(N), size: integer{1..2^N}) => bits(N)`
- `ASR_C{N}(x: bits(N), shift: integer) => (bits(N), bit)`
- `ASR{N}(x: bits(N), shift: integer) => bits(N)`
- `BitCount{N}(x: bits(N)) => integer{0..N}`
- `CeilLog2(a: integer) => integer`
- `CeilPow2(x : integer) => integer`
- `CountLeadingSignBits{N}(x: bits(N)) => integer{0..N-1}`
- `CountLeadingZeroBits{N}(x: bits(N)) => integer{0..N}`
- `Extend{N,M}(x: bits(M), unsigned: boolean) => bits(N)`
- `FloorLog2(a: integer) => integer`
- `FloorPow2(x : integer) => integer`
- `HighestSetBit{N}(x: bits(N)) => integer{-1..N-1}`
- `HighestSetBitNZ{N}(x: bits(N)) => integer{0..N-1}`
- `ILog2(value : real) => integer`
- `IsAlignedP2(x: integer, p2: integer) => boolean`
- `IsAlignedP2{N}(x: bits(N), p2: integer{0..N}) => boolean`
- `IsAlignedSize(x: integer, size: integer) => boolean`
- `IsAlignedSize{N}(x: bits(N), size: integer{1..2^N}) => boolean`
- `IsEven(a: integer) => boolean`
- `IsOdd(a: integer) => boolean`
- `IsOnes{N}(x: bits(N)) => boolean`
- `IsPow2(x : integer) => boolean`
- `IsZero{N}(x: bits(N)) => boolean`
- `Len{N}(x: bits(N)) => integer{N}`
- `LowestSetBit{N}(x: bits(N)) => integer{0..N}`
- `LowestSetBitNZ{N}(x: bits(N)) => integer{0..N-1}`
- `LSL_C{N}(x: bits(N), shift: integer) => (bits(N), bit)`
- `LSL{N}(x: bits(N), shift: integer) => bits(N)`
- `LSR_C{N}(x: bits(N), shift: integer) => (bits(N), bit)`
- `LSR{N}(x: bits(N), shift: integer) => bits(N)`
- `Max(a: integer, b: integer) => integer`
- `Max(a: real, b: real) => real`
- `Min(a: integer, b: integer) => integer`
- `Min(a: real, b: real) => real`
- `Ones{N}() => bits(N)`
- `Real(x: integer) => real`
- `Replicate{N,M}(x: bits(M)) => bits(N)`
- `ReplicateBit{N}(isZero: boolean) => bits(N)`
- `ROL_C{N}(x: bits(N), shift: integer) => (bits(N), bit)`
- `ROL{N}(x: bits(N), shift: integer) => bits(N)`
- `ROR_C{N}(x: bits(N), shift: integer) => (bits(N), bit)`
- `ROR{N}(x: bits(N), shift: integer) => bits(N)`
- `RoundDown(x: real) => integer`
- `RoundTowardsZero(x: real) => integer`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `RoundUp(x: real) => integer`
- `SignExtend{N,M}(x: bits(M)) => bits(N)`
- `SInt{N} (x: bits(N))`
- `SqrtRounded(value : real, fracbits : integer) => real`
- `UInt{N} (x: bits(N)) => integer{0..2^N-1}`
- `ZeroExtend{N,M}(x: bits(M)) => bits(N)`
- `Zeros{N}() => bits(N)`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2 Pseudocode for AArch64 operation

This section holds the pseudocode for execution in AArch64 state. Functions that are listed in this section are identified as `AArch64.FunctionName`. Some of these functions have an equivalent AArch32 function, `AArch32.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch64/debug/breakpoint`.

The top-level sections of the AArch64 pseudocode hierarchy are:

- aarch64/debug.
- aarch64/exceptions.
- aarch64/functions.
- aarch64/translation.

J1.2.1 aarch64/debug

This section includes the following pseudocode functions:

- brbe
- BRBCycleCountingEnabled
- BRBEBranch
- BRBEBranchOnISB
- BRBEDebugStateEntry
- BRBEDebugStateExit
- BRBEException
- BRBEExceptionReturn
- BRBEFreeze
- BRBEISB
- BRBEMispredictAllowed
- BRBETimeStamp
- BRB_IALL
- BRB_INJ
- BranchEncCycleCount
- BranchMispredict
- BranchRawCycleCount
- BranchRecordAllowed
- FilterBranchRecord
- FirstBranchAfterProhibited
- GetBRBENumRecords
- ShouldBRBEFreeze
- UpdateBranchRecordBuffer
- AArch64_BreakpointMatch
- AArch64_BreakpointValueMatch
- AArch64_ReservedBreakpointType
- AArch64_StateMatch
- DebugAddrTop
- EffectiveMDSELR_EL1_BANK
- IsAddressLinkingBreakpoint
- IsBreakpointEnabled
- NumAddressLinkingBreakpointsImplemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SelfHostedExtendedBPWPEnabled
- CheckForPMUException
- EffectivePMEE
- PMUExceptionEnabled
- PMUExceptionMasked
- PMUInterruptEnabled
- AArch64_GenerateDebugExceptions
- AArch64_GenerateDebugExceptionsFrom
- AArch64_TRCIT
- TraceInstrumentation
- AArch64_IncrementCycleCounter
- AArch64_IncrementEventCounter
- AArch64_PMUCycle
- TakeProfilingException
- CheckForSPEException
- CheckMDCR_EL3_NSPBTrap
- CollectContextIDR1
- CollectContextIDR2
- CollectPhysicalAddress
- CollectTimeStamp
- DefaultSPEEvent
- EffectiveMDCR_EL3_NSPB
- EffectivePMBLIMITR_EL1_nVM
- EffectivePMSCR_EL1_EE
- EffectivePMSCR_EL2_EE
- GetPMBSR_EL1_FSC
- GetPMBSR_EL2_FSC
- GetPMBSR_EL3_FSC
- OtherSPEManagementEvent
- PMBSR_EL
- ProfilingBufferEnabled
- ProfilingBufferOwner
- ProfilingSynchronizationBarrier
- ReportSPEEvent
- SPE
- SPEAddByteToRecord
- SPEAddPacketToRecord
- SPEBranch
- SPEBufferIsFull
- SPECollectRecord
- SPECompleteSample
- SPEConstructClass
- SPEConstructRecord
- SPECycle
- SPEEmptyRecord
- SPEEncodeETS
- SPEEncodeEVL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SPEEvent
- SPEFilterByDataSource
- SPEFilterByEvents
- SPEFilterByLatency
- SPEFilterByType
- SPEFreezeOnEvent
- SPEGetDataSource
- SPEGetDataSourcePayloadSize
- SPEGetEventsPayloadSize
- SPEGetRandomBoolean
- SPEGetRandomInterval
- SPEISB
- SPEInterruptEnabled
- SPELDSTType
- SPEMultiAccessSample
- SPEOpAttr
- SPEOpType
- SPEProfilingStopped
- SPEResetSampleCounter
- SPEResetSampleStorage
- SPESampleAddAddressPCVirtual
- SPESampleAddContext
- SPESampleAddTimeStamp
- SPESampleCollision
- SPESampleExtendedLoadStore
- SPESampleGCSSS2
- SPESampleGeneralPurposeLoadStore
- SPESampleLoadStore
- SPESampleMemCopy
- SPESampleMemSet
- SPESampleOnSharedResource
- SPESampleOpOther
- SPESampleOpSMEArrayOther
- SPESampleOpSVEOther
- SPESampleOpSVESMELoadStore
- SPESampleSIMDFPLoadStore
- SPESetDataPhysicalAddress
- SPESetDataVirtualAddress
- SPEStartCounter
- SPEStartSample
- SPEStopCounter
- SPEToCollectSample
- SPEWriteToBuffer
- StatisticalProfilingEnabled
- TimeStamp
- AArch64_TakeExceptionInDebugState
- AArch64_WatchpointByteMatch
- AArch64_WatchpointMatch
- IsWatchpointEnabled

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.1 brbe

```
// Contents of the Branch Record Buffer
//=====================================

var Records_SRC : array [[64]] of BRBSRC_EL1_Type;
var Records_TGT : array [[64]] of BRBTGT_EL1_Type;
var Records_INF : array [[64]] of BRBINF_EL1_Type;



// Getter functions for branch records
//=====================================
// Functions used by MRS instructions that access branch records

func BRBSRC_EL1(n : integer) => BRBSRC_EL1_Type
begin
    assert n IN {0..31};
    let record_index : integer = UInt(BRBFCR_EL1().BANK::n[4:0]);
    if record_index < GetBRBENumRecords() then
        return Records_SRC[[record_index]];
    else
        return Zeros{64};
    end;
end;

func BRBTGT_EL1(n : integer) => BRBTGT_EL1_Type
begin
    assert n IN {0..31};
    let record_index : integer = UInt(BRBFCR_EL1().BANK::n[4:0]);
    if record_index < GetBRBENumRecords() then
        return Records_TGT[[record_index]];
    else
        return Zeros{64};
    end;
end;

func BRBINF_EL1(n : integer) => BRBINF_EL1_Type
begin
    assert n IN {0..31};
    let record_index : integer = UInt(BRBFCR_EL1().BANK::n[4:0]);
    if record_index < GetBRBENumRecords() then
        return Records_INF[[record_index]];
    else
        return Zeros{64};
    end;
end;
```

J1.2.1.2 BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =========================
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

func BRBCycleCountingEnabled() => boolean
begin
    if HaveEL(EL2) && BRBCR_EL2().CC == '0' then return FALSE; end;
    if BRBCR_EL1().CC == '0' then return FALSE; end;
    return TRUE;
end;
```

J1.2.1.3 BRBEBranch

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BRBEBranch()
// ============
// Called to write branch record for the following branches when BRBE is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

func BRBEBranch(br_type : BranchType, cond : boolean, target_address : bits(64))
begin
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        var branch_type : bits(6);
        case br_type of
            when BranchType_DIR =>
                branch_type = if cond then '001000' else '000000';
            when BranchType_INDIR   => branch_type = '000001';
            when BranchType_DIRCALL => branch_type = '000010';
            when BranchType_INDCALL => branch_type = '000011';
            when BranchType_RET     => branch_type = '000101';
            otherwise               => unreachable;
        end;

        var ccu : bit;
        var cc : bits(14);
        (ccu, cc) = BranchEncCycleCount();
        let el : bits(2) = PSTATE.EL;
        let mispredict : bit = (if BRBEMispredictAllowed() && BranchMispredict() then '1'
                                else '0');

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 '11', PC64(), target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

J1.2.1.4 BRBEBranchOnISB

```
// BRBEBranchOnISB()
// =================
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

func BRBEBranchOnISB() => boolean
begin
    return ImpDefBool("ISB generates Branch records");
end;
```

J1.2.1.5 BRBEDebugStateEntry

```
// BRBEDebugStateEntry()
// =====================
// Called to write Debug state entry branch record when BRBE is active.

func BRBEDebugStateEntry(source_address : bits(64))
begin
    if BranchRecordAllowed(PSTATE.EL) then
        let branch_type : bits(6) = '100001';
        var ccu : bit;
        var cc : bits(14);
        (ccu, cc)            = BranchEncCycleCount();
        let el : bits(2)     = '00';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let mispredict : bit = '0';

        // Debug state is a prohibited region, therefore target_address=0
        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 '10', source_address, Zeros{64});

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

J1.2.1.6 BRBEDebugStateExit

```
// BRBEDebugStateExit()
// ====================
// Called to write Debug state exit branch record when BRBE is active.

func BRBEDebugStateExit(target_address : bits(64))
begin
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        let branch_type : bits(6) = '111001';
        let ccu : bit             = '1';
        let cc : bits(14)         = Zeros{14};
        let el : bits(2)          = PSTATE.EL;
        let mispredict : bit      = '0';

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 '01', Zeros{64}, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

J1.2.1.7 BRBEException

```
// BRBEException()
// ===============
// Called to write exception branch record when BRBE is active.

func BRBEException(erec : ExceptionRecord, source_valid : boolean,
                   source_address_in : bits(64),
                   target_address_in : bits(64), target_el : bits(2),
                   trappedsyscallinst : boolean)
begin
    var target_address : bits(64) = target_address_in;
    let except : Exception = erec.exceptype;
    let iss : bits(25)     = erec.syndrome.iss;
    case target_el of
        when EL3 =>
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3().E3BREC == MDCR_EL3().E3BREW) then
                return;
            end;
        when EL2 => if BRBCR_EL2().EXCEPTION == '0' then return; end;
        when EL1 => if BRBCR_EL1().EXCEPTION == '0' then return; end;
    end;

    let target_valid : boolean = BranchRecordAllowed(target_el);

    if source_valid || target_valid then
        var branch_type : bits(6);
        case except of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when Exception_Uncategorized         => branch_type = '100011'; // Trap
    when Exception_WFxTrap               => branch_type = '100011'; // Trap
    when Exception_CP15RTTrap            => branch_type = '100011'; // Trap
    when Exception_CP15RRTTrap           => branch_type = '100011'; // Trap
    when Exception_CP14RTTrap            => branch_type = '100011'; // Trap
    when Exception_CP14DTTrap            => branch_type = '100011'; // Trap
    when Exception_AdvSIMDFPAccessTrap   => branch_type = '100011'; // Trap
    when Exception_FPIDTrap              => branch_type = '100011'; // Trap
    when Exception_PACTrap               => branch_type = '100011'; // Trap
    when Exception_CP14RRTTrap           => branch_type = '100011'; // Trap
    when Exception_OtherTrap             => branch_type = '100011'; // Trap
    when Exception_BranchTarget          => branch_type = '101011'; // Inst Fault
    when Exception_IllegalState          => branch_type = '100011'; // Trap
    when Exception_SupervisorCall        =>
        if !trappedsyscallinst then      branch_type = '100010';    // Call
        else                             branch_type = '100011';    // Trap
        end;
    when Exception_HypervisorCall        => branch_type = '100010'; // Call
    when Exception_MonitorCall           =>
        if !trappedsyscallinst then      branch_type = '100010';    // Call
        else                             branch_type = '100011';    // Trap
        end;
    when Exception_SystemRegisterTrap    => branch_type = '100011'; // Trap
    when Exception_SystemRegister128Trap => branch_type = '100011'; // Trap
    when Exception_SVEAccessTrap         => branch_type = '100011'; // Trap
    when Exception_SMEAccessTrap         => branch_type = '100011'; // Trap
    when Exception_ERetTrap              => branch_type = '100011'; // Trap
    when Exception_PACFail               => branch_type = '101100'; // Data Fault
    when Exception_InstructionAbort      => branch_type = '101011'; // Inst Fault
    when Exception_PCAlignment           => branch_type = '101010'; // Alignment
    when Exception_DataAbort             => branch_type = '101100'; // Data Fault
    when Exception_NV2DataAbort          => branch_type = '101100'; // Data Fault
    when Exception_SPAlignment           => branch_type = '101010'; // Alignment
    when Exception_FPTrappedException    => branch_type = '100011'; // Trap
    when Exception_SError                => branch_type = '100100'; // System Error
    when Exception_Breakpoint            => branch_type = '100110'; // Inst debug
    when Exception_SoftwareStep          => branch_type = '100110'; // Inst debug
    when Exception_Watchpoint            => branch_type = '100111'; // Data debug
    when Exception_NV2Watchpoint         => branch_type = '100111'; // Data debug
    when Exception_SoftwareBreakpoint    => branch_type = '100110'; // Inst debug
    when Exception_IRQ                   => branch_type = '101110'; // IRQ
    when Exception_FIQ                   => branch_type = '101111'; // FIQ
    when Exception_MemCpyMemSet          => branch_type = '100011'; // Trap
    when Exception_GCSFail               =>
        if iss[23:20] == '0000' then        branch_type = '101100'; // Data Fault
        elsif iss[23:20] == '0001' then     branch_type = '101011'; // Inst Fault
        elsif iss[23:20] == '0010' then     branch_type = '100011'; // Trap
        else                                unreachable;
        end;
    when Exception_Profiling             => branch_type = '100110'; // Inst debug
    when Exception_GPC                   =>
        if iss[20] == '1' then              branch_type = '101011'; // Inst fault
        else                                branch_type = '101100'; // Data fault
        end;
    otherwise                            => unreachable;
end;

var ccu : bit;
var cc : bits(14);
(ccu, cc) = BranchEncCycleCount();
let el : bits(2) = if target_valid then target_el else '00';
let mispredict : bit = '0';
let sv : bit = if source_valid then '1' else '0';
let tv : bit = if target_valid then '1' else '0';
let source_address : bits(64) = if source_valid then source_address_in else Zeros{64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !target_valid then
            target_address = Zeros{64};
        else
            target_address = AArch64_BranchAddr(target_address, target_el);
        end;

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 sv::tv, source_address, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

J1.2.1.8 BRBEExceptionReturn

```
// BRBEExceptionReturn()
// =====================
// Called to write exception return branch record when BRBE is active.

func BRBEExceptionReturn(target_address_in : bits(64), source_el : bits(2),
                         source_valid : boolean, source_address_in : bits(64))
begin
    var target_address : bits(64) = target_address_in;
    case source_el of
        when EL3 =>
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3().E3BREC == MDCR_EL3().E3BREW) then
                return;
            end;
        when EL2 => if BRBCR_EL2().ERTN == '0' then return; end;
        when EL1 => if BRBCR_EL1().ERTN == '0' then return; end;
    end;

    let target_valid : boolean = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        let branch_type : bits(6) = '000111';
        var ccu : bit;
        var cc : bits(14);
        (ccu, cc) = BranchEncCycleCount();
        let el : bits(2)     = if target_valid then PSTATE.EL else '00';
        let mispredict : bit = (if source_valid && BRBEMispredictAllowed() &&
                                BranchMispredict() then '1' else '0');
        let sv : bit         = if source_valid then '1' else '0';
        let tv : bit         = if target_valid then '1' else '0';
        let source_address : bits(64) = if source_valid then source_address_in else Zeros{64};
        if !target_valid then
            target_address = Zeros{64};
        end;

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 sv::tv, source_address, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

J1.2.1.9 BRBEFreeze

```
// BRBEFreeze()
// ============
// Generates BRBE freeze event.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func BRBEFreeze()
begin
    BRBFCR_EL1().PAUSED = '1';
    BRBTS_EL1() = GetTimestamp(BRBETimeStamp());
end;
```

J1.2.1.10 BRBEISB

```
// BRBEISB()
// =========
// Handles ISB instruction for BRBE.

func BRBEISB()
begin
    let branch_conditional : boolean = FALSE;
    BRBEBranch(BranchType_DIR, branch_conditional, PC64() + 4);
end;
```

J1.2.1.11 BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =======================
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

func BRBEMispredictAllowed() => boolean
begin
    if HaveEL(EL2) && BRBCR_EL2().MPRED == '0' then return FALSE; end;
    if BRBCR_EL1().MPRED == '0' then return FALSE; end;
    return TRUE;
end;
```

J1.2.1.12 BRBETimeStamp

```
// BRBETimeStamp()
// ===============
// Returns captured timestamp.

func BRBETimeStamp() => TimeStamp
begin
    if HaveEL(EL2) then
        var TS_el2 : bits(2) = BRBCR_EL2().TS;
        if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits{2}(Unpredictable_EL2TIMESTAMP);
        end;
        case TS_el2 of
            when '00' =>
                // Falls out to check BRBCR_EL1.TS
                pass;
            when '01' =>
                return TimeStamp_Virtual;
            when '10' =>
                assert IsFeatureImplemented(FEAT_ECV); // Otherwise ConstrainUnpredictableBits
                                                       // removes this case
                return TimeStamp_OffsetPhysical;
            when '11' =>
                return TimeStamp_Physical;
        end;
    end;

    var TS_el1 : bits(2) = BRBCR_EL1().TS;
    if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits{2}(Unpredictable_EL1TIMESTAMP);
    end;
    case TS_el1 of
        when '01' =>
            return TimeStamp_Virtual;
        when '10' =>
            return TimeStamp_OffsetPhysical;
        when '11' =>
            return TimeStamp_Physical;
        otherwise =>
            unreachable;           // ConstrainUnpredictableBits removes this case
    end;
end;
```

J1.2.1.13 BRB_IALL

```
// BRB_IALL()
// ==========
// Called to perform invalidation of branch records

func BRB_IALL()
begin
    let nbrberecords : integer = GetBRBENumRecords();
    for i = 0 to nbrberecords - 1 do
        Records_SRC[[i]] = Zeros{64};
        Records_TGT[[i]] = Zeros{64};
        Records_INF[[i]] = Zeros{64};
    end;
end;
```

J1.2.1.14 BRB_INJ

```
// BRB_INJ()
// =========
// Called to perform manual injection of branch records.

func BRB_INJ()
begin
    UpdateBranchRecordBuffer(BRBINFINJ_EL1().CCU, BRBINFINJ_EL1().CC,
                             BRBINFINJ_EL1().TYPE, BRBINFINJ_EL1().EL,
                             BRBINFINJ_EL1().MPRED, BRBINFINJ_EL1().VALID, BRBSRCINJ_EL1().ADDRESS,
                             BRBTGTINJ_EL1().ADDRESS);
    BRBINFINJ_EL1() = ARBITRARY : bits(64);
    BRBSRCINJ_EL1() = ARBITRARY : bits(64);
    BRBTGTINJ_EL1() = ARBITRARY : bits(64);

    if ConstrainUnpredictableBool(Unpredictable_BRBFILTRATE) then
        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;
end;
```

J1.2.1.15 BranchEncCycleCount

```
// BranchEncCycleCount()
// =====================

// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
//  - bits[7:0] indicate the mantissa M.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
//  - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1'::M::Zeros{UInt(E)-1}))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.

func BranchEncCycleCount() => (bit, bits(14))
begin
    var cc : integer = BranchRawCycleCount();
    if !BRBCycleCountingEnabled() || FirstBranchAfterProhibited() then
        return ('1', Zeros{14});
    end;

    // The format of this field uses a mantissa and exponent to express the cycle count value.
    //  - bits[7:0] indicate the mantissa M.
    //  - bits[13:8] indicate the exponent E.
    // The cycle count is expressed using the following function:
    //   cycle_count = (if IsZero(E) then UInt(M) else UInt('1'::M::Zeros{UInt(E)-1}))
    // A value of all ones in both the mantissa and exponent indicates the cycle count value
    // exceeded the size of the cycle counter.

    var E : bits(6);
    var M : bits(8);

    if cc < 2^8 then
        E = Zeros{6};
        M = cc[7:0];
    elsif cc >= 2^20 then
        E = Ones{6};
        M = Ones{8};
    else
        E = 1[5:0];
        while cc >= 2^9 looplimit 11 do
            E = E + 1;
            cc = cc DIVRM 2;
        end;
        M = cc[7:0];
    end;

    return ('0', E::M);
end;
```

J1.2.1.16 BranchMispredict

```
// BranchMispredict()
// ==================
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.

impdef func BranchMispredict() => boolean
begin
    return FALSE;
end;
```

J1.2.1.17 BranchRawCycleCount

```
// BranchRawCycleCount()
// =====================
// If the cycle count is known, the return result is the cycle count since the last branch.

impdef func BranchRawCycleCount() => integer
begin
    return 0;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.18 BranchRecordAllowed

```
// BranchRecordAllowed()
// =====================
// Returns TRUE if branch recording is allowed, FALSE otherwise.

func BranchRecordAllowed(el : bits(2)) => boolean
begin
    if ELUsingAArch32(el) then
        return FALSE;
    end;

    if BRBFCR_EL1().PAUSED == '1' then
        return FALSE;
    end;

    if el == EL3 && IsFeatureImplemented(FEAT_BRBEv1p1) then
        return (MDCR_EL3().E3BREC != MDCR_EL3().E3BREW);
    end;

    if HaveEL(EL3) && (MDCR_EL3().SBRBE == '00' ||
        (CurrentSecurityState() == SS_Secure && MDCR_EL3().SBRBE == '01')) then
        return FALSE;
    end;

    case el of
        when EL3 =>  return FALSE;                // FEAT_BRBEv1p1 not implemented
        when EL2 =>  return BRBCR_EL2().E2BRE == '1';
        when EL1 =>  return BRBCR_EL1().E1BRE == '1';
        when EL0 =>
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                return BRBCR_EL2().E0HBRE == '1';
            else
                return BRBCR_EL1().E0BRE == '1';
            end;
    end;
end;
```

J1.2.1.19 FilterBranchRecord

```
// FilterBranchRecord()
// ====================
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

func FilterBranchRecord(br : BranchType, cond : boolean) => boolean
begin
    case br of
        when BranchType_DIRCALL =>
            return BRBFCR_EL1().DIRCALL != BRBFCR_EL1().EnI;
        when BranchType_INDCALL =>
            return BRBFCR_EL1().INDCALL != BRBFCR_EL1().EnI;
        when BranchType_RET =>
            return BRBFCR_EL1().RTN != BRBFCR_EL1().EnI;
        when BranchType_DIR =>
            if cond then
                return BRBFCR_EL1().CONDDIR != BRBFCR_EL1().EnI;
            else
                return BRBFCR_EL1().DIRECT != BRBFCR_EL1().EnI;
            end;
        when BranchType_INDIR =>
            return BRBFCR_EL1().INDIRECT != BRBFCR_EL1().EnI;
        otherwise =>  unreachable;
    end;
    return FALSE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.20 FirstBranchAfterProhibited

```
// FirstBranchAfterProhibited()
// ============================
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.

impdef func FirstBranchAfterProhibited() => boolean
begin
    return FALSE;
end;
```

J1.2.1.21 GetBRBENumRecords

```
// GetBRBENumRecords()
// ===================
// Returns the number of branch records implemented.

func GetBRBENumRecords() => integer
begin
    assert UInt(BRBIDR0_EL1().NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return ImpDefInt("Number of BRBE Branch records");
end;
```

J1.2.1.22 ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// ==================
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

func ShouldBRBEFreeze() => boolean
begin
    if !BranchRecordAllowed(PSTATE.EL) then
        return FALSE;
    end;

    var include_r1 : boolean;
    var include_r2 : boolean;
    let include_r3 : boolean = FALSE;

    if HaveEL(EL2) then
        include_r1 = (BRBCR_EL1().FZP == '1');
        include_r2 = (BRBCR_EL2().FZP == '1');
    else
        include_r1 = TRUE;
        include_r2 = TRUE;
    end;

    return CheckPMUOverflowCondition(PMUOverflowCondition_BRBEFreeze,
                                     include_r1, include_r2, include_r3);
end;
```

J1.2.1.23 UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// ==========================
// Add a new Branch record to the buffer.

func UpdateBranchRecordBuffer(ccu : bit, cc : bits(14), branch_type : bits(6), el : bits(2),
                              mispredict : bit, valid : bits(2),
                              source_address : bits(64), target_address : bits(64))
begin
    // Shift the Branch Records in the buffer
    let nbrberecords : integer = GetBRBENumRecords();
    for i = nbrberecords - 1 downto 1 do
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Records_SRC[[i]] = Records_SRC[[i - 1]];
        Records_TGT[[i]] = Records_TGT[[i - 1]];
        Records_INF[[i]] = Records_INF[[i - 1]];
    end;

    Records_INF[[0]].CCU   = ccu;
    Records_INF[[0]].CC    = cc;

    Records_INF[[0]].EL    = el;
    Records_INF[[0]].VALID = valid;
    Records_INF[[0]].MPRED = mispredict;
    Records_INF[[0]].TYPE  = branch_type;

    Records_SRC[[0]]       = source_address;
    Records_TGT[[0]]       = target_address;

    return;
end;
```

J1.2.1.24 AArch64_BreakpointMatch

```
// AArch64_BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.
// Returns BreakpointInfo structure that contains breakpoint type, a boolean to indicate the
// type of breakpoint, matched address and whether the breakpoint is active and matched
// successfully. For Address Mismatch breakpoints, the returned boolean is the inverted result.
func AArch64_BreakpointMatch(n : integer, vaddress : bits(64), accdesc : AccessDescriptor,
                             size : integer) => BreakpointInfo
begin
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();
    var brkptinfo : BreakpointInfo;

    let linking_enabled : boolean = (DBGBCR_EL1(n).BT IN {'0x11', '1xx1'} ||
                                     (IsFeatureImplemented(FEAT_ABLE) && DBGBCR_EL1(n).BT2 == '1'));

    // A breakpoint that has linking enabled does not generate debug events in isolation
    if linking_enabled then
        brkptinfo.bptype = BreakpointType_Inactive;
        brkptinfo.match = FALSE;
        return brkptinfo;
    end;

    let enabled : boolean    = IsBreakpointEnabled(n);
    let linked : boolean     = DBGBCR_EL1(n).BT == '0x01';
    let isbreakpnt : boolean = TRUE;
    let linked_to : boolean  = FALSE;
    let from_linking_enabled : boolean = FALSE;
    let lbnx : bits(2) = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1(n).LBNX else '00';
    let linked_n : integer{} = UInt(lbnx :: DBGBCR_EL1(n).LBN);
    let ssce : bit = if IsFeatureImplemented(FEAT_RME) then DBGBCR_EL1(n).SSCE else '0';
    let state_match = AArch64_StateMatch(DBGBCR_EL1(n).SSC, ssce, DBGBCR_EL1(n).HMC,
                                         DBGBCR_EL1(n).PMC, linked, linked_n, isbreakpnt,
                                         vaddress, accdesc);

    var (bp_type, value_match) = AArch64_BreakpointValueMatch(n, vaddress, linked_to, isbreakpnt,
                                                              from_linking_enabled);

    if HaveAArch32() && size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        let (-, match_i) = AArch64_BreakpointValueMatch(n, vaddress + 2, linked_to, isbreakpnt,
                                                        from_linking_enabled);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        end;
    end;

    if vaddress[1] == '1' && DBGBCR_EL1(n).BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1(n).BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1(n)+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        end;
    end;

    if !(state_match && enabled) then
        brkptinfo.bptype = BreakpointType_Inactive;
        brkptinfo.match = FALSE;
    else
        brkptinfo.bptype = bp_type;
        brkptinfo.match = value_match;
    end;
    return brkptinfo;
end;
```

J1.2.1.25 AArch64_BreakpointValueMatch

```
// AArch64_BreakpointValueMatch()
// ==============================
// Returns breakpoint type to indicate the type of breakpoint and a boolean to indicate
// whether the breakpoint matched successfully. For Address Mismatch breakpoints, the
// returned boolean is the inverted result. If the breakpoint type return value is Inactive,
// then the boolean result is FALSE.

func AArch64_BreakpointValueMatch(n_in : integer, vaddress : bits(64),
                                  linked_to : boolean, isbreakpnt : boolean,
                                  from_linking_enabled : boolean
                                 ) => (BreakpointType, boolean) recurselimit 2
begin

    // "n_in" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    // "isbreakpnt"  TRUE is this is a call from BreakpointMatch or from StateMatch for a
    // linked breakpoint or from BreakpointValueMatch for a linked breakpoint with linking enabled.
    // "from_linking_enabled" is TRUE if this is a call from BreakpointValueMatch for a linked
    // breakpoint with linking enabled.
    var n : integer = n_in;
    var c : Constraint;
    var dbgtype : bits(5);

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                               Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (BreakpointType_Inactive, FALSE); end;
    end;

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if !IsBreakpointEnabled(n) then return (BreakpointType_Inactive, FALSE); end;

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    if IsFeatureImplemented(FEAT_ABLE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    dbgtype = DBGBCR_EL1(n).[BT2,BT];
else
    dbgtype = '0' :: DBGBCR_EL1(n).BT;
end;

(c, dbgtype) = AArch64_ReservedBreakpointType(n, dbgtype);
if c == Constraint_DISABLED then return (BreakpointType_Inactive, FALSE); end;
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
let match_addr : boolean      = (dbgtype == 'x0x0x');
let mismatch : boolean        = (dbgtype == 'x010x');
let match_vmid : boolean      = (dbgtype == 'x10xx');
let match_cid : boolean       = (dbgtype == 'x001x');
let match_cid1 : boolean      = (dbgtype IN {'x101x', 'xx11x'});
let match_cid2 : boolean      = (dbgtype == 'x11xx');
let linking_enabled : boolean = (dbgtype IN {'xxx11', 'x1xx1', '1xxxx'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not
// programmed with linking enabled.
if linked_to && !linking_enabled then
    return (BreakpointType_Inactive, FALSE);
end;

// If called from BreakpointMatch return FALSE for breakpoint with linking enabled.
if !linked_to && linking_enabled then
    return (BreakpointType_Inactive, FALSE);
end;

let linked : boolean = (dbgtype == 'x0x01');
if from_linking_enabled then  // A breakpoint with linking enabled has called this function.
    assert linked_to && isbreakpnt;
    if linked then
        // A breakpoint with linking enabled is linked to a linked breakpoint. This is
        // architecturally UNPREDICTABLE, but treated as disabled in the pseudo code to
        // avoid potential recursion in BreakpointValueMatch().
        return (BreakpointType_Inactive, FALSE);
    end;
end;

// If a linked breakpoint is linked to an address matching breakpoint,
// the behavior is CONSTRAINED UNPREDICTABLE.
if linked_to && match_addr && isbreakpnt then
    if !ConstrainUnpredictableBool(Unpredictable_BPLINKEDADDRMATCH) then
        return (BreakpointType_Inactive, FALSE);
    end;
end;

// A breakpoint programmed for address mismatch does not match in AArch32 state.
if mismatch && UsingAArch32() then
    return (BreakpointType_Inactive, FALSE);
end;

var bvr_match : boolean  = FALSE;
var bxvr_match : boolean = FALSE;
var bp_type : BreakpointType;
var mask : integer{0..31};

if IsFeatureImplemented(FEAT_BWE) then
    mask = UInt(DBGBCR_EL1(n).MASK);

    // If the mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask IN {1, 2} then
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESBPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            case c of
                when Constraint_DISABLED => return (BreakpointType_Inactive, FALSE);  // Disabled
                when Constraint_NONE =>     mask = 0;                                 // No masking
                // Otherwise the value returned by ConstrainUnpredictableBits must
                // be a not-reserved value.
                otherwise => mask = unpred_mask as integer{3..31};
            end;
        end;

        if mask != 0 then
            // When DBGBCR_EL1(n).MASK is a valid nonzero value, the behavior is
            // CONSTRAINED UNPREDICTABLE if any of the following are true:
            //     - DBGBCR_EL1(n).[BT2,BT] is programmed for a Context matching breakpoint.
            //     - DBGBCR_EL1(n).BAS is not '1111' and AArch32 is supported at EL0.
            if ((match_cid || match_cid1 || match_cid2) ||
                (DBGBCR_EL1(n).BAS != '1111' && HaveAArch32())) then
                if !ConstrainUnpredictableBool(Unpredictable_BPMASK) then
                    return (BreakpointType_Inactive, FALSE);
                end;
            end;
        else
            // A stand-alone mismatch of a single address is not supported.
            if mismatch then
                return (BreakpointType_Inactive, FALSE);
            end;
        end;

    else
        mask = 0;
    end;

    // Do the comparison.
    if match_addr then
        var byte_select_match : boolean;
        let byte : integer = UInt(vaddress[1:0]);

        if HaveAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                       // "vaddress" is halfword aligned
            byte_select_match = (DBGBCR_EL1(n).BAS[byte] == '1');
        else
            assert byte == 0;                           // "vaddress" is word aligned
            byte_select_match = TRUE;                   // DBGBCR_EL1(n).BAS[byte] is RES1
        end;

        // When FEAT_LVA3 is not implemented, if the DBGBVR_EL1(n).RESS field bits are not a
        // sign extension of the MSB of DBGBVR_EL1(n).VA, it is UNPREDICTABLE whether they
        // appear to be included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
        // generates a Translation fault.
        let dbgtop : AddressSize = DebugAddrTop();
        let unpredictable_ress : boolean = (dbgtop < 55 && !IsOnes(DBGBVR_EL1(n)[63:dbgtop]) &&
                                            !IsZero(DBGBVR_EL1(n)[63:dbgtop]) &&
                                        ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
        let cmpmsb : integer{} = if unpredictable_ress then 63 else dbgtop;
        let cmplsb : integer{} = if mask > 2 then mask else 2;
        bvr_match = ((vaddress[cmpmsb:cmplsb] == DBGBVR_EL1(n)[cmpmsb:cmplsb]) &&
                     byte_select_match);
        if mask > 2 then
            // If masked bits of DBGBVR_EL1(n) are not zero, the behavior
            // is CONSTRAINED UNPREDICTABLE.
            let masktop : integer{} = mask - 1;
            if bvr_match && !IsZero(DBGBVR_EL1(n)[masktop:2]) then
                bvr_match = ConstrainUnpredictableBool(Unpredictable_BPMASKEDBITS);
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif match_cid then
    if IsInHost() then
        bvr_match = (CONTEXTIDR_EL2()[31:0] == DBGBVR_EL1(n)[31:0]);
    else
        bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1()[31:0] == DBGBVR_EL1(n)[31:0]);
    end;

elsif match_cid1 then
    bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() &&
                 CONTEXTIDR_EL1()[31:0] == DBGBVR_EL1(n)[31:0]);
end;

if match_vmid then
    let vmid : bits(NUM_VMIDBITS) = VMID();
    var bvr_vmid : bits(NUM_VMIDBITS);

    if !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2().VS == '0' then
        bvr_vmid = ZeroExtend{NUM_VMIDBITS}(DBGBVR_EL1(n)[39:32]);
    else
        bvr_vmid = DBGBVR_EL1(n)[47:32];
    end;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);

elsif match_cid2 then
    bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
                  DBGBVR_EL1(n)[63:32] == CONTEXTIDR_EL2()[31:0]);
end;

let bvr_match_valid : boolean  = (match_addr || match_cid || match_cid1);
let bxvr_match_valid : boolean = (match_vmid || match_cid2);

var value_match : boolean = ((!bxvr_match_valid || bxvr_match) &&
                             (!bvr_match_valid || bvr_match));

// A watchpoint might be linked to a linked address matching breakpoint with linking enabled,
// which is in turn linked to a context matching breakpoint.
if linked_to && linked then
    // If called from StateMatch and breakpoint is a linked breakpoint then it must be a
    // watchpoint that is linked to an address matching breakpoint which is linked to a
    // context matching breakpoint.
    assert !isbreakpnt && match_addr && IsFeatureImplemented(FEAT_ABLE);
    let lbnx : bits(2) = (if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1(n).LBNX
                                                                   else '00');
    let linked_linked_n : integer{} = UInt(lbnx :: DBGBCR_EL1(n).LBN);
    var linked_value_match : boolean;
    let linked_vaddress = ARBITRARY : bits(64);
    let linked_linked_to = TRUE;
    let linked_isbreakpnt = TRUE;
    let linked_from_linking_enabled = TRUE;
    (bp_type, linked_value_match) = AArch64_BreakpointValueMatch(linked_linked_n,
                                                                 linked_vaddress,
                                                                 linked_linked_to,
                                                                 linked_isbreakpnt,
                                                                 linked_from_linking_enabled);
    value_match = value_match && linked_value_match;
end;

if match_addr && !mismatch then
    bp_type = BreakpointType_AddrMatch;
elsif match_addr && mismatch then
    bp_type = BreakpointType_AddrMismatch;
elsif match_vmid || match_cid || match_cid1 || match_cid2 then
    bp_type = BreakpointType_CtxtMatch;
else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        unreachable;
    end;

    return (bp_type, value_match);
end;
```

J1.2.1.26 AArch64_ReservedBreakpointType

```
// AArch64_ReservedBreakpointType()
// ================================
// Checks if the given DBGBCR_EL1(n).[BT2,BT] values are reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

func AArch64_ReservedBreakpointType(n : integer, bt_in : bits(5)) => (Constraint, bits(5))
begin
    var bt : bits(5)            = bt_in;
    var reserved : boolean      = FALSE;
    let context_aware : boolean = IsContextAwareBreakpoint(n);

    if bt[4] == '0' then
        // Context matching
        if bt != '00x0x' && !context_aware then
            reserved = TRUE;
        end;

        // EL2 extension
        if bt == '01xxx' && !HaveEL(EL2) then
            reserved = TRUE;
        end;

        // Context matching
        if (bt IN {'0011x','011xx'} && !IsFeatureImplemented(FEAT_VHE) &&
              !IsFeatureImplemented(FEAT_Debugv8p2)) then
            reserved = TRUE;
        end;

        // Reserved
        if bt == '0010x' && !IsFeatureImplemented(FEAT_BWE) && !HaveAArch32EL(EL1) then
            reserved = TRUE;
        end;
    else
        // Reserved
        if bt != '10x0x' then
            reserved = TRUE;
        end;
    end;

    if reserved then
        var c : Constraint;
        (c, bt) = ConstrainUnpredictableBits{5}(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, ARBITRARY : bits(5));
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
        end;
    end;

    return (Constraint_NONE, bt);
end;
```

J1.2.1.27 AArch64_StateMatch

```
// AArch64_StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_StateMatch(ssc_in : bits(2), ssce_in : bit, hmc_in : bit,
                        pxc_in : bits(2), linked_in : boolean, linked_n_in : integer,
                        isbreakpnt : boolean, vaddress : bits(64),
                        accdesc : AccessDescriptor) => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0'; end;

    // "ssc_in","ssce_in","hmc_in","pxc_in" are the control fields from
    // the DBGBCR_EL1(n) or DBGWCR_EL1(n) register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1(n) or
    // DBGWCR_EL1(n) register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "vaddress" is the program counter for a linked watchpoint or the same value passed to
    // AArch64_CheckBreakpoint for a linked breakpoint.
    // "accdesc" describes the properties of the access being matched.
    var ssc : bits(2)      = ssc_in;
    var ssce : bit         = ssce_in;
    var hmc : bit          = hmc_in;
    var pxc : bits(2)      = pxc_in;
    let linked : boolean   = linked_in;
    let linked_n : integer = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    var c : Constraint;
    (c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE; end;
    // Otherwise the hmc,ssc,ssce,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    let EL3_match : boolean = HaveEL(EL3) && hmc == '1' && ssc[0] == '0';
    let EL2_match : boolean = HaveEL(EL2) && ((hmc == '1' && (ssc::pxc != '1000')) || ssc == '11');
    let EL1_match : boolean = pxc[0] == '1';
    let EL0_match : boolean = pxc[1] == '1';

    var priv_match : boolean;
    case accdesc.el of
        when EL3 =>  priv_match = EL3_match;
        when EL2 =>  priv_match = EL2_match;
        when EL1 =>  priv_match = EL1_match;
        when EL0 =>  priv_match = EL0_match;
    end;

    // Security state match
    var ss_match : boolean;
    case ssce::ssc of
        when '000' => ss_match = hmc == '1' || accdesc.ss != SS_Root;
        when '001' => ss_match = accdesc.ss == SS_NonSecure;
        when '010' => ss_match = (hmc == '1' && accdesc.ss == SS_Root) || accdesc.ss == SS_Secure;
        when '011' => ss_match = (hmc == '1' && accdesc.ss != SS_Root) || accdesc.ss == SS_Secure;
        when '101' => ss_match = accdesc.ss == SS_Realm;
    end;

    var linked_match : boolean = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit. If it is not context-aware
        // then it is CONSTRAINED UNPREDICTABLE whether this gives no match, gives a match without
        // linking, or linked_n is mapped to some UNKNOWN breakpoint that is context-aware.
        if (!IsContextAwareBreakpoint(linked_n) &&
            (isbreakpnt || !IsAddressLinkingBreakpoint(linked_n))) then
            return ConstrainUnpredictableBool(Unpredictable_BPNOTLINKING);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if linked then
        let linked_to : boolean = TRUE;
        var bp_type : BreakpointType;
        let from_linking_enabled : boolean = FALSE;
        (bp_type, linked_match) = AArch64_BreakpointValueMatch(linked_n, vaddress,
                                                               linked_to, isbreakpnt,
                                                               from_linking_enabled);
        if bp_type == BreakpointType_AddrMismatch then
            linked_match = !linked_match;
        end;
    end;

    return priv_match && ss_match && (!linked || linked_match);
end;
```

J1.2.1.28 DebugAddrTop

```
// DebugAddrTop()
// ==============
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

func DebugAddrTop() => AddressSize
begin
    if IsFeatureImplemented(FEAT_LVA3) then
        return 55;
    elsif IsFeatureImplemented(FEAT_LVA) then
        return 52;
    else
        return 48;
    end;
end;
```

J1.2.1.29 EffectiveMDSELR_EL1_BANK

```
// EffectiveMDSELR_EL1_BANK()
// ==========================
// Return the effective value of MDSELR_EL1.BANK.

func EffectiveMDSELR_EL1_BANK() => bits(2)
begin
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented,
    // then the field is RES0.
    let num_bp : integer = NumBreakpointsImplemented();
    let num_wp : integer = NumWatchpointsImplemented();
    if num_bp <= 16 && num_wp <= 16 then
        return '00';
    end;

    // At EL3, the Effective value of this field is zero if MDCR_EL3.EBWE is 0.
    // At EL2, the Effective value is zero if the Effective value of MDCR_EL2.EBWE is 0.
    // That is, if either MDCR_EL3.EBWE is 0 or MDCR_EL2.EBWE is 0.
    // At EL1, the Effective value is zero if the Effective value of MDSCR_EL2.EMBWE is 0.
    // That is, if any of MDCR_EL3.EBWE, MDCR_EL2.EBWE, or MDSCR_EL1.EMBWE is 0.
    if ((HaveEL(EL3) && MDCR_EL3().EBWE == '0') ||
          (PSTATE.EL != EL3 && EL2Enabled() && MDCR_EL2().EBWE == '0') ||
          (PSTATE.EL == EL1 && MDSCR_EL1().EMBWE == '0')) then
        return '00';
    end;

    var bank : bits(2) = MDSELR_EL1().BANK;

    // Values are reserved depending on the number of breakpoints or watchpoints
    // implemented.
    if ((bank == '11' && num_bp <= 48 && num_wp <= 48) ||
          (bank == '10' && num_bp <= 32 && num_wp <= 32)) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Reserved value
        (-, bank) = ConstrainUnpredictableBits{2}(Unpredictable_RESMDSELR);
        // The value returned by ConstrainUnpredictableBits must be a not-reserved value
    end;

    return bank;
end;
```

J1.2.1.30 IsAddressLinkingBreakpoint

```
// IsAddressLinkingBreakpoint()
// ============================
// Returns TRUE if DBGBCR_EL1(n) supports being configured as an Address matching breakpoint with
// linking enabled.

func IsAddressLinkingBreakpoint(n : integer) => boolean
begin
    return n < NumAddressLinkingBreakpointsImplemented();
end;
```

J1.2.1.31 IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGBCR_EL1(n).E is '1', and FALSE otherwise.

func IsBreakpointEnabled(n : integer) => boolean
begin
    if (n > 15 &&
          ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEnabled()) ||
          (HaltOnBreakpointOrWatchpoint() && EDSCR2().EHBWE == '0'))) then
        return FALSE;
    end;

    return DBGBCR_EL1(n).E == '1';
end;
```

J1.2.1.32 NumAddressLinkingBreakpointsImplemented

```
// NumAddressLinkingBreakpointsImplemented()
// =========================================
// Returns the number of breakpoints that support being configured as Address matching breakpoint
// with linking enabled, as described by ID_AA64DFR1_EL1().ABL_CMPs.

func NumAddressLinkingBreakpointsImplemented() => integer
begin
    if !IsFeatureImplemented(FEAT_ABLE) then
        return 0;
    else
        return ImpDefInt("Number of address-linking breakpoints");
    end;
end;
```

J1.2.1.33 SelfHostedExtendedBPWPEnabled

```
// SelfHostedExtendedBPWPEnabled()
// ===============================
// Returns TRUE if the extended breakpoints and watchpoints are enabled, and FALSE otherwise
// from a self-hosted debug perspective.

func SelfHostedExtendedBPWPEnabled() => boolean
begin
    if NumBreakpointsImplemented() <= 16 && NumWatchpointsImplemented() <= 16 then
        return FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if ((HaveEL(EL3) && MDCR_EL3().EBWE == '0') ||
          (EL2Enabled() && MDCR_EL2().EBWE == '0')) then
        return FALSE;
    end;

    return MDSCR_EL1().EMBWE == '1';
end;
```

J1.2.1.34 CheckForPMUException

```
// CheckForPMUException()
// ======================
// Take a PMU exception if enabled, permitted, and unmasked.

func CheckForPMUException()
begin
    var enabled : boolean;
    var target_el : bits(2);
    var pmu_exception : boolean;
    var synchronous : boolean;
    (enabled, target_el) = PMUExceptionEnabled();
    if !enabled || PMUExceptionMasked(target_el, PSTATE.EL, PSTATE.PM) then
        pmu_exception = FALSE;
    else
        let include_r1 : boolean = TRUE;
        let include_r2 : boolean = TRUE;
        let include_r3 : boolean = TRUE;
        pmu_exception = CheckPMUOverflowCondition(PMUOverflowCondition_PMUException,
                                                  include_r1, include_r2, include_r3);
        synchronous = FALSE;
    end;
    if pmu_exception then
        let fsc : bits(5) = '00000';
        TakeProfilingException(target_el, fsc, synchronous);
    end;
end;
```

J1.2.1.35 EffectivePMEE

```
// EffectivePMEE()
// ===============
// Return the Effective value of PMEE from MDCR_ELx or PMECR_EL1, handling reserved encodings:
// '10' (MDCR_ELx), '10' and '01' (PMECR_EL1).

func EffectivePMEE(el : bits(2), for_interrupt : boolean) => bits(2)
begin
    var val : bits(2);
    case el of
        when EL1  => val = PMECR_EL1().PMEE;
        when EL2  => val = MDCR_EL2().PMEE;
        when EL3  => val = MDCR_EL3().PMEE;
        otherwise => unreachable;
    end;

    if (val == '10' || (el == EL1 && val == '01')) then
        var c : Constraint;
        (c, val) = ConstrainUnpredictableBits{2}(Unpredictable_RESPMEE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            val = if for_interrupt then '11' else '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    return val;
end;
```

J1.2.1.36 PMUExceptionEnabled

```
// PMUExceptionEnabled()
// =====================
// The first return value is TRUE if the PMU exception is enabled, and FALSE otherwise.
// The second return value is the target Exception level for an enabled PMU exception.

func PMUExceptionEnabled() => (boolean, bits(2))
begin

    if !IsFeatureImplemented(FEAT_EBEP) then
        return (FALSE, ARBITRARY : bits(2));
    end;

    var enabled : boolean;
    var target : bits(2) = ARBITRARY : bits(2);
    let for_interrupt : boolean = FALSE;
    let pmee_el3 : bits(2) = EffectivePMEE(EL3, for_interrupt);
    let pmee_el2 : bits(2) = EffectivePMEE(EL2, for_interrupt);
    let pmee_el1 : bits(2) = EffectivePMEE(EL1, for_interrupt);

    if HaveEL(EL3) && pmee_el3 != '01' then
        enabled = pmee_el3 == '11';
        if enabled then target = EL3; end;

    elsif EL2Enabled() && pmee_el2 != '01' then
        enabled = pmee_el2 == '11';
        if enabled then target = EL2; end;

    else
        enabled = pmee_el1 == '11';
        if enabled then
            target = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        end;
    end;

    return (enabled, target);
end;
```

J1.2.1.37 PMUExceptionMasked

```
// PMUExceptionMasked()
// ====================
// Return TRUE if the PMU Exception is masked at the specified target Exception level
// relative to the specified source Exception level, and by the value of PSTATE.PM,
// and FALSE otherwise.

func PMUExceptionMasked(target_el : bits(2), from_el : bits(2), pm : bit) => boolean
begin
    assert IsFeatureImplemented(FEAT_EBEP);
    let for_interrupt : boolean = FALSE;

    if Halted() then
        return TRUE;
    elsif UInt(target_el) < UInt(from_el) then
        return TRUE;
    elsif (from_el == EL2 && target_el == EL2 &&
             EffectivePMEE(EL2, for_interrupt) != '11') then
        return TRUE;
    elsif target_el == from_el && (PMECR_EL1().KPME == '0' || pm == '1') then
        return TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    return FALSE;
end;
```

J1.2.1.38 PMUInterruptEnabled

```
// PMUInterruptEnabled()
// =====================
// Return TRUE if the PMU interrupt request (PMUIRQ) is enabled, FALSE otherwise.

func PMUInterruptEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_EBEP) then
        return TRUE;
    end;

    var enabled : boolean;
    let for_interrupt : boolean = TRUE;
    let pmee_el3 : bits(2) = EffectivePMEE(EL3, for_interrupt);
    let pmee_el2 : bits(2) = EffectivePMEE(EL2, for_interrupt);
    let pmee_el1 : bits(2) = EffectivePMEE(EL1, for_interrupt);

    if HaveEL(EL3) && pmee_el3 != '01' then
        enabled = pmee_el3 == '00';

    elsif EL2Enabled() && pmee_el2 != '01' then
        enabled = pmee_el2 == '00';

    else
        enabled = pmee_el1 == '00';
    end;

    return enabled;
end;
```

J1.2.1.39 AArch64_GenerateDebugExceptions

```
// AArch64_GenerateDebugExceptions()
// =================================

func AArch64_GenerateDebugExceptions() => boolean
begin
    let ss : SecurityState = CurrentSecurityState();
    return AArch64_GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
end;
```

J1.2.1.40 AArch64_GenerateDebugExceptionsFrom

```
// AArch64_GenerateDebugExceptionsFrom()
// =====================================

func AArch64_GenerateDebugExceptionsFrom(from_el : bits(2), from_state : SecurityState,
                                         mask : bit) => boolean
begin

    if OSLSR_EL1().OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    end;

    let route_to_el2 : boolean = (HaveEL(EL2) &&
                                  (from_state != SS_Secure || IsSecureEL2Enabled()) &&
                                  (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1'));
    let target : bits(2) = (if route_to_el2 then EL2 else EL1);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var enabled : boolean;
    if HaveEL(EL3) && from_state == SS_Secure then
        enabled = MDCR_EL3().SDD == '0';
        if from_el == EL0 && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3().SUIDEN == '1';
        end;
    else
        enabled = TRUE;
    end;

    if from_el == target then
        enabled = enabled && MDSCR_EL1().KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);
    end;

    return enabled;
end;
```

J1.2.1.41 AArch64_TRCIT

```
// AArch64_TRCIT()
// ===============
// Determines whether an Instrumentation trace packet should
// be generated and then generates an instrumentation trace packet
// containing the value of the register passed as an argument

func AArch64_TRCIT(Xt : bits(64))
begin
    let ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);
    end;
end;
```

J1.2.1.42 TraceInstrumentation

```
// TraceInstrumentation()
// ======================
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

impdef func TraceInstrumentation(Xt : bits(64))
begin
    return;
end;
```

J1.2.1.43 AArch64_IncrementCycleCounter

```
// AArch64_IncrementCycleCounter()
// ===============================
// Increment the cycle counter and possibly set overflow bits.

func AArch64_IncrementCycleCounter()
begin
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return; end;
    var d : bit = PMCR_EL0().D;   // Check divide-by-64
    var lc : bit = PMCR_EL0().LC;
    var lc_enabled : boolean;
    (lc_enabled, -) = PMUExceptionEnabled();
    lc = if lc_enabled then '1' else lc;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0'; end;
    if d == '1' && !HasElapsed64Cycles() then return; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let old_value : integer = UInt(PMCCNTR_EL0());
    let new_value : integer = old_value + 1;
    PMCCNTR_EL0() = new_value[63:0];

    let ovflw : integer{} = if HaveAArch32() && lc == '0' then 32 else 64;

    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET_EL0().C = '1';
    end;

    return;
end;
```

J1.2.1.44 AArch64_IncrementEventCounter

```
// AArch64_IncrementEventCounter()
// ==============================
// Increment the specified event counter 'idx' by the specified amount 'increment'.
// 'Vm' is the value event counter 'idx-1' is being incremented by if 'idx' is odd,
// zero otherwise.
// Returns the amount the counter was incremented by.

func AArch64_IncrementEventCounter(idx : integer, increment_in : integer, Vm : integer) => integer
begin
    var old_value : integer;
    var new_value : integer;

    old_value = UInt(PMEVCNTR_EL0(idx));
    let increment : integer = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    var lp : bit;
    if IsFeatureImplemented(FEAT_PMUv3p5) then
        PMEVCNTR_EL0(idx) = new_value[63:0];
        var pmuexception_enabled : boolean;
        (pmuexception_enabled, -) = PMUExceptionEnabled();
        if pmuexception_enabled then
            lp = '1';
        else
            case GetPMUCounterRange(idx) of
                when PMUCounterRange_R1 =>
                    lp = PMCR_EL0().LP;
                when PMUCounterRange_R2 =>
                    lp = MDCR_EL2().HLP;
                when PMUCounterRange_R3 =>
                    lp = '1';
                otherwise =>
                    unreachable;
            end;
        end;
    else
        lp = '0';
        PMEVCNTR_EL0(idx) = ZeroExtend{64}(new_value[31:0]);
    end;
    let ovflw : integer{} = if lp == '1' then 64 else 32;
    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET_EL0()[idx] = '1';
        // Check for the CHAIN event from an even counter
        if (idx[0] == '0' && idx + 1 < NUM_PMU_COUNTERS && lp == '0' &&
              (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
                 ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    return increment;
end;
```

J1.2.1.45 AArch64_PMUCycle

```
// AArch64_PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

func AArch64_PMUCycle()
begin
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    PMUEvent(PMU_EVENT_CPU_CYCLES);
    let counters : integer = NUM_PMU_COUNTERS;
    var Vm : integer = 0;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            if CountPMUEvents(idx) then
                let accumulated : integer = PMUEventAccumulator[[idx]];
                if (idx MOD 2) == 0 then Vm = 0; end;
                Vm = AArch64_IncrementEventCounter(idx, accumulated, Vm);
            end;
            PMUEventAccumulator[[idx]] = 0;
        end;
    end;
    AArch64_IncrementCycleCounter();
    CheckForPMUOverflow();
end;
```

J1.2.1.46 TakeProfilingException

```
// TakeProfilingException()
// ========================
// Takes a Profiling exception.

func TakeProfilingException(target_el : bits(2), fsc : bits(5), synchronous : boolean)
begin
    var except : ExceptionRecord = ExceptionSyndrome(Exception_Profiling);
    except.syndrome.iss[5:1] = fsc;
    if synchronous then
        except.syndrome.iss[0] = '1';
    end;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.1.47 CheckForSPEException

```
// CheckForSPEException()
// ======================
// Take an SPE Profiling exception if pending, permitted, and unmasked.

func CheckForSPEException()
begin
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if Halted() || Restarting() then
        return;
    end;

    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;
    var route_to_el1 : boolean = FALSE;

    if HaveEL(EL3) && MDCR_EL3().PMSEE == '1x' then
        let pending : boolean = PMBSR_EL3().S == '1';
        let masked : boolean = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;
    end;

    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = ProfilingBufferOwner();
    let in_owning_ss : boolean = IsCurrentSecurityState(owning_ss);

    if EffectivePMSCR_EL2_EE() == '1x' then
        let pending : boolean = PMBSR_EL2().S == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL == EL3 ||
                                (PSTATE.EL == EL2 && (PMSCR_EL2().EE != '11' ||
                                 PMSCR_EL2().KE == '0' || PSTATE.PM == '1'))
                                );
        route_to_el2 = pending && !masked;
    end;

    if EffectivePMSCR_EL1_EE() == '11' then
        let pending : boolean = PMBSR_EL1().S == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL IN {EL3, EL2} ||
                                (PSTATE.EL == EL1 && (PMSCR_EL1().KE == '0' || PSTATE.PM == '1'))
                                );
        if EffectiveTGE() == '1' then
            route_to_el2 = route_to_el2 || (pending && !masked);
        else
            route_to_el1 = pending && !masked;
        end;
    end;

    let fsc : bits(5) = '00001';        // SPE exception
    let synchronous : boolean = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    end;
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    end;
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
    end;
end;
```

J1.2.1.48 CheckMDCR_EL3_NSPBTrap

```
// CheckMDCR_EL3_NSPBTrap()
// ========================
// Check if the register access is trappable by MDCR_EL3.[NSPBE, NSPB]

func CheckMDCR_EL3_NSPBTrap() => boolean
begin
    var state_bits : bits(3);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var reserved : boolean;
    (state_bits, reserved) = EffectiveMDCR_EL3_NSPB();
    return ((reserved && ConstrainUnpredictableBool(Unpredictable_RESERVEDNSxB_Trap)) ||
             state_bits[0] == '0' || state_bits[1] != EffectiveSCR_EL3_NS() ||
             (IsFeatureImplemented(FEAT_RME) && state_bits[2] != EffectiveSCR_EL3_NSE()));
end;
```

J1.2.1.49 CollectContextIDR1

```
// CollectContextIDR1()
// ====================
func CollectContextIDR1() => boolean
begin
    if !StatisticalProfilingEnabled() then return FALSE; end;
    if PSTATE.EL == EL2 then return FALSE; end;
    if EL2Enabled() && HCR_EL2().TGE == '1' then return FALSE; end;
    return PMSCR_EL1().CX == '1';
end;
```

J1.2.1.50 CollectContextIDR2

```
// CollectContextIDR2()
// ====================

func CollectContextIDR2() => boolean
begin
    if !StatisticalProfilingEnabled() then return FALSE; end;
    if !EL2Enabled() then return FALSE; end;
    return PMSCR_EL2().CX == '1';
end;
```

J1.2.1.51 CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

func CollectPhysicalAddress() => boolean
begin
    if !StatisticalProfilingEnabled() then return FALSE; end;
    let (owning_ss, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2().PA == '1' && (owning_el == EL2 || PMSCR_EL1().PA == '1');
    else
        return PMSCR_EL1().PA == '1';
    end;
end;
```

J1.2.1.52 CollectTimeStamp

```
// CollectTimeStamp()
// ==================
func CollectTimeStamp() => TimeStamp
begin
    if !StatisticalProfilingEnabled() then return TimeStamp_None; end;
    let (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2().TS == '0' then return TimeStamp_None; end;
    else
        if PMSCR_EL1().TS == '0' then return TimeStamp_None; end;
    end;

    var PCT_el1 : bits(2);
    if !IsFeatureImplemented(FEAT_ECV) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PCT_el1 = '0'::PMSCR_EL1().PCT[0];                        // PCT[1] is RES0
    else
        PCT_el1 = PMSCR_EL1().PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits{2}(Unpredictable_PMSCR_PCT);
        end;
    end;
    if EL2Enabled() then
        var PCT_el2 : bits(2);
        if !IsFeatureImplemented(FEAT_ECV) then
            PCT_el2 = '0'::PMSCR_EL2().PCT[0];                    // PCT[1] is RES0
        else
            PCT_el2 = PMSCR_EL2().PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits{2}(Unpredictable_PMSCR_PCT);
            end;
        end;
        case PCT_el2 of
            when '00' =>
                return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
            when '01' =>
                if owning_el == EL2 then return TimeStamp_Physical; end;
            when '11' =>
                assert IsFeatureImplemented(FEAT_ECV);          // FEAT_ECV must be implemented
                if owning_el == EL1 && PCT_el1 == '00' then
                    return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                else
                    return TimeStamp_OffsetPhysical;
                end;
            otherwise =>
                unreachable;
        end;
    end;

    case PCT_el1 of
        when '00' => return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
        when '01' => return TimeStamp_Physical;
        when '11' =>
            assert IsFeatureImplemented(FEAT_ECV);              // FEAT_ECV must be implemented
            return TimeStamp_OffsetPhysical;
        otherwise => unreachable;
    end;
end;
```

J1.2.1.53 DefaultSPEEvent

```
// DefaultSPEEvent()
// =================
// Return the target ELx for an indirect write to PMBSR_ELx for an Other buffer management
// event or anything other than a buffer management event.

func DefaultSPEEvent() => bits(2)
begin
    return ReportSPEEvent(Zeros{6}, ARBITRARY : bits(6));
end;
```

J1.2.1.54 EffectiveMDCR_EL3_NSPB

```
// EffectiveMDCR_EL3_NSPB()
// ========================
// Return the Effective value of MDCR_EL3.[NSPBE, NSPB] field and whether it is a reserved value.

readonly func EffectiveMDCR_EL3_NSPB() => (bits(3), boolean)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    var state_bits : bits(3);
    var reserved : boolean = FALSE;

    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3().[NSPBE, NSPB];
        if state_bits == '10x' || (!IsFeatureImplemented(FEAT_Secure) && state_bits == '00x') then
            // Reserved value
            reserved = TRUE;
            (-, state_bits) = ConstrainUnpredictableBits{3}(Unpredictable_RESERVEDNSxB);
        end;
    else
        state_bits = '0' :: MDCR_EL3().NSPB;
    end;

    return (state_bits, reserved);
end;
```

J1.2.1.55 EffectivePMBLIMITR_EL1_nVM

```
// EffectivePMBLIMITR_EL1_nVM()
// ============================

func EffectivePMBLIMITR_EL1_nVM() => bit
begin
    if !IsFeatureImplemented(FEAT_SPE_nVM) then
        return '0';
    end;
    if HaveEL(EL2) then
        let (owning_ss, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();
        if ((owning_ss != SS_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
              PMSCR_EL2().EnVM == '0') then
            return '0';
        end;
    end;
    return PMBLIMITR_EL1().nVM;
end;
```

J1.2.1.56 EffectivePMSCR_EL1_EE

```
// EffectivePMSCR_EL1_EE()
// =======================
// Return the Effective value of PMSCR_EL1.EE for the purpose of controlling the
// SPE Profiling exception.

func EffectivePMSCR_EL1_EE() => bits(2)
begin
    if EffectivePMSCR_EL2_EE() == '00' then
        return '00';
    end;

    var ee : bits(2) = PMSCR_EL1().EE;
    if ee IN {'01', '10'} then            // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee[0] = ee[1];
        else
            var c : Constraint;
            (c, ee) = ConstrainUnpredictableBits{2}(Unpredictable_RESPMSEE);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                ee = '00';
            end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    return ee;
end;
```

J1.2.1.57 EffectivePMSCR_EL2_EE

```
// EffectivePMSCR_EL2_EE()
// =======================
// Return the Effective value of PMSCR_EL2.EE.

readonly func EffectivePMSCR_EL2_EE() => bits(2)
begin
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return '00';
    end;

    if HaveEL(EL3) && MDCR_EL3().PMSEE == '00' then
        return '00';
    end;

    let check_el2 : boolean = (HaveEL(EL2) &&
                               (EffectiveSCR_EL3_NS() == '1' || IsSecureEL2Enabled()));
    return if check_el2 then PMSCR_EL2().EE else '01';
end;
```

J1.2.1.58 GetPMBSR_EL1_FSC

```
// GetPMBSR_EL1_FSC()
// ==================
// Query the PMBSR_EL1.FSC field.

func GetPMBSR_EL1_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = PMBSR_EL1()[5:0];
    return FSC;
end;
```

J1.2.1.59 GetPMBSR_EL2_FSC

```
// GetPMBSR_EL2_FSC()
// ==================
// Query the PMBSR_EL2.FSC field.

func GetPMBSR_EL2_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = PMBSR_EL2()[5:0];
    return FSC;
end;
```

J1.2.1.60 GetPMBSR_EL3_FSC

```
// GetPMBSR_EL3_FSC()
// ==================
// Query the PMBSR_EL3.FSC field.

func GetPMBSR_EL3_FSC() => bits(6)
begin
    var FSC : bits(6);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    FSC = PMBSR_EL3()[5:0];
    return FSC;
end;
```

J1.2.1.61 OtherSPEManagementEvent

```
// OtherSPEManagementEvent()
// =========================
// Report an Other buffer management event, with the status code 'bsc'

func OtherSPEManagementEvent(bsc : bits(6))
begin
    let target_el : bits(2) = DefaultSPEEvent();
    if PMBSR_EL(target_el).S == '0' then
        PMBSR_EL(target_el).S   = '1';       // Assert interrupt or exception
        PMBSR_EL(target_el).EC  = '000000';  // Other buffer management event
        PMBSR_EL(target_el).MSS = ZeroExtend{16}(bsc);
        PMBSR_EL(target_el).MSS2 = Zeros{24};
    end;
end;
```

J1.2.1.62 PMBSR_EL

```
// PMBSR_EL - accessor
// ===================

accessor PMBSR_EL(el : bits(2)) <=> value : PMBSRType
begin
    getter
        var r : bits(64);
        case el of
            when EL1  => r = PMBSR_EL1();
            when EL2  => r = PMBSR_EL2();
            when EL3  => r = PMBSR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        let r : bits(64) = value;
        case el of
            when EL1  => PMBSR_EL1() = r;
            when EL2  => PMBSR_EL2() = r;
            when EL3  => PMBSR_EL3() = r;
            otherwise => unreachable;
        end;
        return;
    end;
end;
```

J1.2.1.63 ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

readonly func ProfilingBufferEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SPE) then
        return FALSE;
    end;

    let (-, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();

    return !ELUsingAArch32(owning_el) && PMBLIMITR_EL1().E == '1';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.64 ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

readonly func ProfilingBufferOwner() => (SecurityState, bits(2))
begin
    var owning_ss : SecurityState;

    var state_bits : bits(3);
    if HaveEL(EL3) then
        (state_bits, -) = EffectiveMDCR_EL3_NSPB();
    else
        state_bits = if SecureOnlyImplementation() then '001' else '011';
    end;

    case state_bits of
        when '00x' => owning_ss = SS_Secure;
        when '01x' => owning_ss = SS_NonSecure;
        when '11x' => owning_ss = SS_Realm;
    end;

    var owning_el : bits(2);
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2().E2PB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    end;

    return (owning_ss, owning_el);
end;
```

J1.2.1.65 ProfilingSynchronizationBarrier

```
// ProfilingSynchronizationBarrier()
// =================================
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.

impdef func ProfilingSynchronizationBarrier()
begin
    return;
end;
```

J1.2.1.66 ReportSPEEvent

```
// ReportSPEEvent()
// ================
// Return the target ELx for an indirect write to PMBSR_ELx.
// When the indirect write is due to a buffer management event:
// 'ec_bits' is the Event Class for the management event.
// 'fsc_bits' is the Fault Status Code when this is a fault, ignored otherwise.
// Otherwise, 'ec_bits' should be Zeros().

func ReportSPEEvent(ec_bits : bits(6), fsc_bits : bits(6)) => bits(2)
begin
    var target_el : bits(2);
    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;

    if IsFeatureImplemented(FEAT_SPE_EXC) then
        if ec_bits == '011111' then                     // implementation defined fault
            return ImpDefBits{2}("target_el for SPE ImpDef Fault");
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let s1fault : boolean = (ec_bits == '100100');   // Stage 1 fault
        let s2fault : boolean = (ec_bits == '100101');   // Stage 2 fault

        var gpcfault, gpfault : boolean;
        if IsFeatureImplemented(FEAT_RME) then
            // Granule Protection Check fault, other than GPF. That is, a GPT address size fault,
            // GPT walk fault, or synchronous External abort on GPT fetch.
            gpcfault = (ec_bits == '011110');
            // Other Granule Protection Fault, reported as Stage 1 or Stage 2 fault.
            gpfault  =  ((s1fault || s2fault) && fsc_bits IN {'10001x', '1001xx', '101000'});
        else
            gpcfault = FALSE;
            gpfault  = FALSE;
        end;
        let sync_ext_abort : boolean = ((s1fault || s2fault) &&
                                        fsc_bits IN {'010000', '01001x', '0101xx', '011011'});

        var owning_ss : SecurityState;
        var owning_el : bits(2);
        (owning_ss, owning_el) = ProfilingBufferOwner();

        if HaveEL(EL3) && MDCR_EL3().PMSEE == '1x' then
            route_to_el3 = (MDCR_EL3().PMSEE == '11' ||
                            gpcfault || (gpfault && SCR_EL3().GPF == '1') ||
                            (sync_ext_abort && EffectiveEA() == '1'));
        end;

        if EffectivePMSCR_EL2_EE() == '1x' then
            route_to_el2 = (PMSCR_EL2().EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                            gpcfault || (gpfault && HCR_EL2().GPF == '1') ||
                            (sync_ext_abort && EffectiveHCR_TEA() == '1'));
        end;
    end;

    if route_to_el3 then
        target_el = EL3;
    elsif route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;
    end;

    return target_el;
end;
```

J1.2.1.67 SPE

```
// SPE Implementation Constants
// ============================

constant SPEMaxAddrs : integer{}      = 32;
constant SPEMaxCounters : integer{}   = 32;
constant SPEMaxRecordSize : integer{} = 2048;
```

J1.2.1.68 SPEAddByteToRecord

```
// SPEAddByteToRecord()
// ====================
// Add one byte to a record and increase size property appropriately.

func SPEAddByteToRecord(b : bits(8))
begin
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[[SPERecordSize]] = b;
    SPERecordSize = SPERecordSize + 1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.1.69 SPEAddPacketToRecord

```
// SPEAddPacketToRecord()
// ======================
// Add passed header and payload data to the record.
// Payload must be a multiple of 8.

func SPEAddPacketToRecord{N}(header_hi : bits(2), header_lo : bits(4),
                             payload : bits(N))
begin
    assert N MOD 8 == 0;
    var sz : bits(2);
    case N of
        when 8  => sz = '00';
        when 16 => sz = '01';
        when 32 => sz = '10';
        when 64 => sz = '11';
        otherwise => unreachable;
    end;

    let header : bits(8) = header_hi::sz::header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1 do
        SPEAddByteToRecord(payload[i*:8]);
    end;
end;
```

J1.2.1.70 SPEBranch

```
// SPEBranch()
// ===========
// Called on every branch if SPE is present. Maintains previous branch target
// and branch related SPE functionality.

func SPEBranch{N}(target : bits(N), branch_type : BranchType, conditional : boolean,
                  taken_flag : boolean)
begin
    let is_isb : boolean = FALSE;
    SPEBranch{N}(target, branch_type, conditional, taken_flag, is_isb);
end;

func SPEBranch{N}(target : bits(N), branch_type : BranchType, conditional : boolean,
                  taken_flag : boolean, is_isb : boolean)
begin
    // If the PE implements branch prediction, data about (mis)prediction is collected
    // through the PMU events.

    var collect_prev_br : boolean;
    let collect_prev_br_eret : boolean = ImpDefBool("SPE prev br on eret");
    let collect_prev_br_exception : boolean = (ImpDefBool(
                                                "SPE prev br on exception"));
    let collect_prev_br_isb : boolean = ImpDefBool("SPE prev br on isb");
    case branch_type of
        when BranchType_EXCEPTION => collect_prev_br = collect_prev_br_exception;
        when BranchType_ERET      => collect_prev_br = collect_prev_br_eret;
        otherwise                 => collect_prev_br = !is_isb || collect_prev_br_isb;
    end;

    // Implements previous branch target functionality
    if (taken_flag && IsFeatureImplemented(FEAT_SPE_PBT) && StatisticalProfilingEnabled() &&
            collect_prev_br) then

        if SPESampleInFlight then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let previous_target : bits(64) = SPESamplePreviousBranchAddress;
            SPESampleAddress[[SPEAddrPosPrevBranchTarget]][63:0] = previous_target[63:0];
            let previous_branch_valid : boolean = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[[SPEAddrPosPrevBranchTarget]] = previous_branch_valid;
        end;
        // Save the target address for it to be added to a future record.
        SPESamplePreviousBranchAddress[55:0] = target[55:0];

        var ns : bit;
        var nse : bit;
        case CurrentSecurityState() of
            when SS_Secure =>
                ns = '0';
                nse = '0';
            when SS_NonSecure =>
                ns = '1';
                nse = '0';
            when SS_Realm =>
                ns  = '1';
                nse = '1';
            otherwise => unreachable;
        end;

        SPESamplePreviousBranchAddress[63] = ns;
        SPESamplePreviousBranchAddress[60] = nse;
        SPESamplePreviousBranchAddress[62:61] = PSTATE.EL;
        SPESamplePreviousBranchAddressValid   = TRUE;
    end;

    if !StatisticalProfilingEnabled() then
        if taken_flag then
            // Invalidate previous branch address, if profiling is disabled
            // or prohibited.
            SPESamplePreviousBranchAddressValid = FALSE;
        end;
        return;
    end;

    if SPESampleInFlight then
        SPESampleOpAttr.branch_is_direct = branch_type IN {BranchType_DIR, BranchType_DIRCALL};
        SPESampleOpAttr.branch_has_link  = branch_type IN {BranchType_DIRCALL, BranchType_INDCALL};
        SPESampleOpAttr.procedure_return = branch_type == BranchType_RET;
        SPESampleOpAttr.op_type          = SPEOpType_Branch;
        SPESampleOpAttr.is_conditional   = conditional;
        SPESampleOpAttr.cond_pass        = taken_flag;

        // Save the target address.
        if taken_flag then
            var ns : bit;
            var nse : bit;
            case CurrentSecurityState() of
                when SS_Secure =>
                    ns = '0';
                    nse = '0';
                when SS_NonSecure =>
                    ns = '1';
                    nse = '0';
                when SS_Realm =>
                    ns  = '1';
                    nse = '1';
                otherwise => unreachable;
            end;
            let el : bits(2) = PSTATE.EL;

            SPESampleAddress[[SPEAddrPosBranchTarget]][55:0]  = target[55:0];
            SPESampleAddress[[SPEAddrPosBranchTarget]][63:56] = ns::el::nse::Zeros{4};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            SPESampleAddressValid[[SPEAddrPosBranchTarget]]   = TRUE;
        end;
    end;
end;
```

J1.2.1.71 SPEBufferIsFull

```
// SPEBufferIsFull()
// =================
// Return true if another full size sample record would not fit in the
// profiling buffer.

func SPEBufferIsFull() => boolean
begin
    let write_pointer_limit : integer   = UInt(PMBLIMITR_EL1().LIMIT::Zeros{12});
    let current_write_pointer : integer = UInt(PMBPTR_EL1());
    let record_max_size : integer       = 1 << UInt(PMSIDR_EL1().MaxSize);
    return current_write_pointer > (write_pointer_limit - record_max_size);
end;
```

J1.2.1.72 SPECollectRecord

```
// SPECollectRecord()
// ==================
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

func SPECollectRecord(events : bits(64), total_latency : integer, optype : SPEOpType) => boolean
begin
    // Filtering by events
    var is_rejected_event : boolean;
    var is_rejected_nevent : boolean;
    (is_rejected_event, is_rejected_nevent) = SPEFilterByEvents(events);

    // Filtering by type
    let is_rejected_type : boolean = SPEFilterByType(SPESampleOpAttr);

    // Filtering by latency
    let is_rejected_latency : boolean = SPEFilterByLatency(total_latency);

    let is_rejected_data_source : boolean = SPEFilterByDataSource(optype);

    var return_value : boolean;
    return_value = !(is_rejected_nevent || is_rejected_event ||
                     is_rejected_type || is_rejected_latency ||
                     is_rejected_data_source);

    if return_value then
        PMUEvent(PMU_EVENT_SAMPLE_FILTRATE);
    end;

    return return_value;
end;
```

J1.2.1.73 SPECompleteSample

```
// SPECompleteSample()
// ===================
// Called to complete the sampling process.

func SPECompleteSample()
begin
    assert SPESampleInFlight;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PMUEvent(PMU_EVENT_SAMPLE_FEED);

    // Stop any pending counters
    for counter_index = 0 to (SPEMaxCounters - 1) do
        if SPESampleCounterPending[[counter_index]] then
            SPEStopCounter(counter_index);
        end;
    end;

    // Record any IMPLEMENTATION DEFINED events
    let impdef_events : bits(64) = ImpDefBits{}("SPE EVENTS");
    SPESampleEvents[63:48] = impdef_events[63:48];
    // The number of bits available for IMPLEMENTATION DEFINED events
    // is reduced by FEAT_SPEv1p4
    if !IsFeatureImplemented(FEAT_SPEv1p4) then
        SPESampleEvents[31:24] = impdef_events[31:24];
    end;
    SPESampleEvents[15:12] = impdef_events[15:12];
    // Bit 24 encodes whether the sample was collected in Streaming SVE mode.
    if IsFeatureImplemented(FEAT_SPE_SME) then
        SPESampleEvents[24] = PSTATE.SM;
    end;
    SPESampleEvents[6] = if (SPESampleOpAttr.is_conditional &&
                             !SPESampleOpAttr.cond_pass) then '1' else '0';

    var collect_record : boolean;
    collect_record = SPECollectRecord(SPESampleEvents,
                                      SPESampleCounter[[SPECounterPosTotalLatency]],
                                      SPESampleOpAttr.op_type);
    var discard : boolean = FALSE;
    if IsFeatureImplemented(FEAT_SPEv1p2) then
        discard = PMBLIMITR_EL1().FM == '10';
    end;
    if collect_record && !discard then
        SPEConstructRecord();
        if SPEBufferIsFull() then
            let bsc : bits(6) = '000001';     // Buffer full event
            OtherSPEManagementEvent(bsc);
            PMUEvent(PMU_EVENT_SAMPLE_BUFFER_FULL);
        end;
    end;

    SPESampleInFlight = FALSE;

    SPEResetSampleStorage();
end;

// Order of addresses in sample storage and architectural index values
constant SPEAddrPosPCVirtual : integer    = 0;
constant SPEAddrPosBranchTarget : integer = 1;
constant SPEAddrPosDataVirtual : integer  = 2;
constant SPEAddrPosDataPhysical : integer = 3;
constant SPEAddrPosPrevBranchTarget : integer = 4;
// Order of counters in sample storage and architectural index values
constant SPECounterPosTotalLatency : integer = 0;
constant SPECounterPosIssueLatency : integer = 1;
constant SPECounterPosTranslationLatency : integer = 2;
constant SPECounterPosAltIssueLatency : integer = 4;
// Sample in-flight flag
var SPESampleInFlight : boolean = FALSE;

// Globally declared variables which store data about the sample.

// Context storage
var SPESampleContextEL1 : bits(32);
var SPESampleContextEL1Valid : boolean;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var SPESampleContextEL2 : bits(32);
var SPESampleContextEL2Valid : boolean;

// Counter storage
var SPESampleCounter : array [[SPEMaxCounters]] of integer;
var SPESampleCounterValid : array [[SPEMaxCounters]] of boolean;
var SPESampleCounterPending : array [[SPEMaxCounters]] of boolean;

// Address storage
var SPESampleAddress : array [[SPEMaxAddrs]] of bits(64);
var SPESampleAddressValid : array [[SPEMaxAddrs]] of boolean;
var SPESamplePreviousBranchAddress : bits(64);
var SPESamplePreviousBranchAddressValid : boolean;

// Data source storage
var SPESampleDataSource : bits(16);
var SPESampleDataSourceValid : boolean;

// OpAttr storage
var SPESampleOpAttr : SPEOpAttr;

// Timestamp storage
var SPESampleTimestamp : bits(64);
var SPESampleTimestampValid : boolean;

// Event storage
var SPESampleEvents : bits(64);
```

J1.2.1.74 SPEConstructClass

```
// SPEConstructClass()
// ===================
// Constructs the encodings for the class and subclass fields.

func SPEConstructClass() => (bits(2), bits(8))
begin
    var op_class : bits(2);
    var op_subclass : bits(8);
    let cond : bit     = if SPESampleOpAttr.is_conditional then '1' else '0';
    let fp : bit       = if SPESampleOpAttr.is_floating_point then '1' else '0';
    let ldst : bit     = if SPESampleOpAttr.op_type == SPEOpType_Store then '1' else '0';
    let ar : bit       = if SPESampleOpAttr.is_acquire_release then '1' else '0';
    let excl : bit     = if SPESampleOpAttr.is_exclusive then '1' else '0';
    let at : bit       = if SPESampleOpAttr.at then '1' else '0';
    let indirect : bit = if SPESampleOpAttr.branch_is_direct then '0' else '1';
    let pred : bit = if SPESampleOpAttr.is_predicated then '1' else '0';
    let sg : bit   = if SPESampleOpAttr.is_gather_scatter then '1' else '0';
    let evl : bits(3) = SPESampleOpAttr.evl;
    let simd : bit = if SPESampleOpAttr.is_simd then '1' else '0';
    let ets : bits(4) = SPESampleOpAttr.ets;
    // Since this implementation of SPE samples instruction instead of micro-operations, a
    // Branch with link or Procedure return instruction will never be recorded as a "Load/store,
    // GCS" format Operation Type packet. Therefore the COMMON bit is hard-wired to '1'.
    let common : bit = '1';
    if SPESampleOpAttr.op_type == SPEOpType_Other then
        op_class = '00';
        op_subclass = Zeros{5}::simd::fp::cond;
    elsif SPESampleOpAttr.op_type == SPEOpType_OtherSVE then
        op_class = '00';
        op_subclass = '0'::evl::'1'::pred::fp::'0';
    elsif SPESampleOpAttr.op_type == SPEOpType_OtherSME then
        op_class = '00';
        op_subclass = '1'::ets[3:1]::'1'::ets[0]::fp::'0';
    elsif SPESampleOpAttr.op_type == SPEOpType_Branch then
        op_class = '10';
        var cr : bits(2) = '00';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        var gcs : bit = '0';
        if IsFeatureImplemented(FEAT_SPE_CRR) then
            if SPESampleOpAttr.branch_has_link then
                cr = '01';
            elsif SPESampleOpAttr.procedure_return then
                cr = '10';
            else
                cr = '11';
            end;
        end;
        if IsFeatureImplemented(FEAT_GCS) then
            if (SPESampleOpAttr.ldst_type == SPELDSTType_GCS &&
                 (SPESampleOpAttr.branch_has_link || SPESampleOpAttr.procedure_return)) then
                gcs = '1';
            end;
        end;
        op_subclass = Zeros{3}::cr::gcs::indirect::cond;
    elsif SPESampleOpAttr.op_type IN {SPEOpType_Load, SPEOpType_Store, SPEOpType_LoadAtomic} then
        op_class = '01';
        case SPESampleOpAttr.ldst_type of
            when SPELDSTType_NV2         => op_subclass = '0011000'::ldst;
            when SPELDSTType_Extended    => op_subclass = '000'::ar::excl::at::'1'::ldst;
            when SPELDSTType_General     => op_subclass = Zeros{7}::ldst;
            when SPELDSTType_SIMDFP      => op_subclass = '0000010'::ldst;
            when SPELDSTType_SVESME      => op_subclass = sg::evl::'1'::pred::'0'::ldst;
            when SPELDSTType_Unspecified => op_subclass = '0001000'::ldst;
            when SPELDSTType_Tags        => op_subclass = '0001010'::ldst;
            when SPELDSTType_MemCopy     => op_subclass = '0010000'::ldst;
            when SPELDSTType_MemSet      => op_subclass = '00100101';
            when SPELDSTType_GCS         => op_subclass = '01000'::common::'0'::ldst;
            when SPELDSTType_GCSSS2      => unreachable;
                // GCSSS2 is converted to GCS, should not appear here
            otherwise                    => unreachable;
        end;
    else
        unreachable;
    end;
    return (op_class, op_subclass);
end;
```

J1.2.1.75 SPEConstructRecord

```
// SPEConstructRecord()
// ====================
// Create new record and populate it with packets using sample storage data.
// This is an example implementation, packets may appear in
// any order as long as the record ends with an End or Timestamp packet.

func SPEConstructRecord()
begin
    // Empty the record.
    SPEEmptyRecord();

    var op_class : bits(2);
    var op_subclass : bits(8);
    (op_class, op_subclass) = SPEConstructClass();
    // Add operation details
    SPEAddPacketToRecord{8}('01', '10'::op_class, op_subclass);


    // Add contextEL1 if available
    if SPESampleContextEL1Valid then
        SPEAddPacketToRecord{32}('01', '0100', SPESampleContextEL1);
    end;

     // Add contextEL2 if available
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if SPESampleContextEL2Valid then
    SPEAddPacketToRecord{32}('01', '0101', SPESampleContextEL2);
end;

// Add valid counters
for counter_index = 0 to (SPEMaxCounters - 1) do
    if SPESampleCounterValid[[counter_index]] then
        if counter_index >= 8 then
            // Need extended format
            SPEAddByteToRecord('001000'::counter_index[4:3]);
        end;
        // Check for overflow
        let large_counters : boolean = ImpDefBool("SPE 16bit counters");
        if large_counters then
            if SPESampleCounter[[counter_index]] > 0xFFFF then
                SPESampleCounter[[counter_index]] = 0xFFFF;
            end;
        else
            if SPESampleCounter[[counter_index]] > 0xFFF then
                SPESampleCounter[[counter_index]] = 0xFFF;
            end;
        end;

        // Add byte0 for short format (byte1 for extended format)
        SPEAddPacketToRecord{16}('10', '1'::counter_index[2:0],
                                 SPESampleCounter[[counter_index]][15:0]);
    end;
end;

// Add valid addresses
if IsFeatureImplemented(FEAT_SPE_PBT) then
    // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
    // previous branch packet is present.
    let include_prev_br : boolean = (ImpDefBool(
                                        "SPE get prev br if not br"));
    if SPESampleOpAttr.op_type != SPEOpType_Branch && !include_prev_br then
        SPESampleAddressValid[[SPEAddrPosPrevBranchTarget]] = FALSE;
    end;
end;

// Data Virtual address should not be collected if this was an NV2 access and Statistical
// Profiling is disabled at EL2.
if !StatisticalProfilingEnabled(EL2) && SPESampleOpAttr.ldst_type == SPELDSTType_NV2 then
    SPESampleAddressValid[[SPEAddrPosDataVirtual]] = FALSE;
end;

for address_index = 0 to (SPEMaxAddrs - 1) do
    if SPESampleAddressValid[[address_index]] then
        if address_index >= 8 then
            // Need extended format
            SPEAddByteToRecord('001000'::address_index[4:3]);
        end;
        // Add byte0 for short format (byte1 for extended format)
        SPEAddPacketToRecord{64}('10', '0'::address_index[2:0],
                                 SPESampleAddress[[address_index]]);
    end;
end;

// Add Data Source
if SPESampleDataSourceValid then
    let ds_payload_size : integer{} = SPEGetDataSourcePayloadSize();
    SPEAddPacketToRecord{8 * ds_payload_size}('01', '0011',
                                              SPESampleDataSource[8*ds_payload_size-1:0]);
end;

// Add events
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Get size of payload in bytes.
    let payload_size : integer{} = SPEGetEventsPayloadSize();
    SPEAddPacketToRecord{8 * payload_size}('01', '0010', SPESampleEvents[8*payload_size-1:0]);

    // Add Timestamp to end the record if one is available.
    // Otherwise end with an End packet.
    if SPESampleTimestampValid then
        SPEAddPacketToRecord{64}('01', '0001', SPESampleTimestamp);
    else
        SPEAddByteToRecord('00000001');
    end;

    // Add padding
    while SPERecordSize MOD (1<<UInt(PMBIDR_EL1().Align)) != 0 looplimit 2048 do
        SPEAddByteToRecord(Zeros{8});
    end;
    SPEWriteToBuffer();
    CTI_SignalEvent(CrossTriggerIn_SPESample);
end;
```

J1.2.1.76 SPECycle

```
// SPECycle()
// ==========
// Function called at the end of every cycle. Responsible for asserting interrupts
// and advancing counters.

func SPECycle()
begin
    if !IsFeatureImplemented(FEAT_SPE) then
        return;
    end;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1) do
            if SPESampleCounterPending[[i]] then
                SPESampleCounter[[i]] = SPESampleCounter[[i]] + 1;
            end;
        end;
    end;

    // Assert PMBIRQ if appropriate.
    if SPEInterruptEnabled() && PMBSR_EL1().S == '1' then
        SetInterruptRequestLevel(InterruptID_PMBIRQ, HIGH);
    else
        SetInterruptRequestLevel(InterruptID_PMBIRQ, LOW);
    end;
end;
```

J1.2.1.77 SPEEmptyRecord

```
// SPEEmptyRecord()
// ================
// Reset record data.

func SPEEmptyRecord()
begin
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1) do
        SPERecordData[[i]] = Zeros{8};
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.78 SPEEncodeETS

```
// SPEEncodeETS()
// ==============
// Encodes an integer tile size length into the ets field for the SPE operation type packet.

func SPEEncodeETS(size : integer) => bits(4)
begin
    var ets : bits(4);
    if size <= 128 then
        ets = '0000';
    elsif size <= 256 then
        ets = '0001';
    elsif size <= 512 then
        ets = '0010';
    elsif size <= 1024 then
        ets = '0011';
    elsif size <= 2048 then
        ets = '0100';
    elsif size <= 4096 then
        ets = '0101';
    elsif size <= 8192 then
        ets = '0110';
    elsif size <= 16384 then
        ets = '0111';
    elsif size <= 32768 then
        ets = '1000';
    elsif size <= 65536 then
        ets = '1001';
    elsif size <= 131072 then
        ets = '1010';
    elsif size <= 262144 then
        ets = '1011';
    else
        ets = '1111';
    end;
    return ets;
end;
```

J1.2.1.79 SPEEncodeEVL

```
// SPEEncodeEVL()
// ==============
// Encodes an integer vector length into the evl field for the SPE operation type packet.

func SPEEncodeEVL(vl : integer) => bits(3)
begin
    var evl : bits(3);
    if vl <= 32 then
        evl = '000';
    elsif vl <= 64 then
        evl = '001';
    elsif vl <= 128 then
        evl = '010';
    elsif vl <= 256 then
        evl = '011';
    elsif vl <= 512 then
        evl = '100';
    elsif vl <= 1024 then
        evl = '101';
    elsif vl <= 2048 then
        evl = '110';
    else
        if IsFeatureImplemented(FEAT_SPE_SME) then
            evl = '111';
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            unreachable;
        end;
    end;
    return evl;
end;
```

J1.2.1.80 SPEEvent

```
// SPEEvent()
// ==========
// Called when a PMU event is generated by the sampled structure.
// Sets appropriate bit in SPESampleStorage.events.

func SPEEvent(pmuevent : bits(16))
begin
    if !SPESampleInFlight then
        return;
    end;
    case pmuevent of
        when PMU_EVENT_DSNP_HIT_RD =>
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents[23] = '1';                                  // Snoop hit
            end;
        when {PMU_EVENT_L1D_LFB_HIT_RD, PMU_EVENT_L2D_LFB_HIT_RD,
              PMU_EVENT_L3D_LFB_HIT_RD, PMU_EVENT_LL_LFB_HIT_RD} =>
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents[22] = '1';                                  // Recent fetch
            end;
        when {PMU_EVENT_L1D_CACHE_HITM_RD, PMU_EVENT_L2D_CACHE_HITM_RD,
              PMU_EVENT_L3D_CACHE_HITM_RD, PMU_EVENT_LL_CACHE_HITM_RD} =>
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents[21] = '1';                                  // Modified
            end;
        when PMU_EVENT_L2D_CACHE_LMISS_RD =>
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents[20] = '1';                                  // L2 miss
            end;
        when PMU_EVENT_L2D_CACHE_RD =>
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents[19] = '1';                                  // L2 access
            end;
        when PMU_EVENT_SVE_PRED_EMPTY_SPEC =>
            if IsFeatureImplemented(FEAT_SPEv1p1) then
                SPESampleEvents[18] = '1';                                  // Empty predicate
            end;
        when PMU_EVENT_SVE_PRED_NOT_FULL_SPEC =>
            if IsFeatureImplemented(FEAT_SPEv1p1) then
                SPESampleEvents[17] = '1';                                  // Partial predicate
            end;
        when PMU_EVENT_LDST_ALIGN_LAT =>
            if IsFeatureImplemented(FEAT_SPEv1p1) then
                SPESampleEvents[11] = '1';                                  // Misaligned
            end;
        when PMU_EVENT_REMOTE_ACCESS =>         SPESampleEvents[10]  = '1';    // Remote access
        when PMU_EVENT_LL_CACHE_MISS =>         SPESampleEvents[9]   = '1';    // LLC miss
        when PMU_EVENT_LL_CACHE =>              SPESampleEvents[8]   = '1';    // LLC access
        when PMU_EVENT_BR_MIS_PRED =>           SPESampleEvents[7]   = '1';    // Mispredicted
        when PMU_EVENT_BR_MIS_PRED_RETIRED =>   SPESampleEvents[7]   = '1';    // Not taken
        when PMU_EVENT_DTLB_WALK =>             SPESampleEvents[5]   = '1';    // TLB walk
        when PMU_EVENT_L1D_TLB =>               SPESampleEvents[4]   = '1';    // TLB access
        when PMU_EVENT_L1D_CACHE_REFILL =>
            if !IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents[3] = '1';                                   // L1 refill
            end;
        when PMU_EVENT_L1D_CACHE_LMISS_RD =>
            if IsFeatureImplemented(FEAT_SPEv1p4) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                SPESampleEvents[3] = '1';                                   // L1 miss
            end;
        when PMU_EVENT_L1D_CACHE =>             SPESampleEvents[2]   = '1';    // L1 access
        when PMU_EVENT_INST_RETIRED =>          SPESampleEvents[1]   = '1';    // Retire
        when PMU_EVENT_EXC_TAKEN =>             SPESampleEvents[0]   = '1';    // Exception
        otherwise => return;
    end;
    return;
end;
```

J1.2.1.81 SPEFilterByDataSource

```
// SPEFilterByDataSource()
// =======================
// Carry out filtering by data source.

func SPEFilterByDataSource(optype : SPEOpType) => boolean
begin
    // Filtering by Data Source
    var is_rejected_data_source : boolean = FALSE;
    if (IsFeatureImplemented(FEAT_SPE_FDS) && SPESampleDataSourceValid &&
         (optype IN {SPEOpType_Load, SPEOpType_LoadAtomic})) then
        let data_source : bits(16) = SPESampleDataSource;
        let index : integer        = UInt(data_source[5:0]);
        let is_ds : boolean        = PMSDSFR_EL1()[index] == '1';
        if is_ds then PMUEvent(PMU_EVENT_SAMPLE_FEED_DS); end;
        if PMSFCR_EL1().FDS == '1' then
            // Filtering by Data Source is enabled
            is_rejected_data_source = !is_ds;
        end;
    end;
    return is_rejected_data_source;
end;
```

J1.2.1.82 SPEFilterByEvents

```
// SPEFilterByEvents()
// ===================
// Carries out filtering by the event bits.

func SPEFilterByEvents(events : bits(64)) => (boolean, boolean)
begin
    // "mask" defines which Events packet bits are checked by the filter
    var mask : bits(64) = Zeros{};
    let impdef_mask : bits(64) = ImpDefBits{}("SPE mask");

    mask[63:48] = impdef_mask[63:48];
    if IsFeatureImplemented(FEAT_SPE_SME) && IsFeatureImplemented(FEAT_SPEv1p4) then
        mask[25:24] = '11';                 // Streaming mode, SMCU
    end;
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask[23:19] = '11111';              // Snoop hit, recent fetch, modified,
                                            // L2 miss, L2 access
    else
        mask[31:24] = impdef_mask[31:24];
    end;
    if IsFeatureImplemented(FEAT_SPEv1p1) && IsFeatureImplemented(FEAT_SVE) then
        mask[18:17] = '11';                 // Predicates
    end;
    mask[15:12] = impdef_mask[15:12];
    if IsFeatureImplemented(FEAT_SPEv1p1) then
        mask[11] = '1';                     // Data alignment
    end;
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask[10:8] = '111';                 // Remote access, LLC access, LLC miss
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        mask[10:8] = impdef_mask[10:8];
    end;
    mask[7] = '1';                          // Mispredicted
    if IsFeatureImplemented(FEAT_SPE_FnE) then
        mask[6] = '1';                      // Not taken
    end;
    mask[5,3,1] = '111';                    // TLB walk, L1 miss, retired
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask[4,2] = '11';                   // TLB access, L1 access
    else
        mask[4,2] = impdef_mask[4,2];
    end;

    let e : bits(64) = events AND mask;

    // Filtering by event
    let evfr : bits(64) = PMSEVFR_EL1() AND mask;
    var is_rejected_event : boolean = FALSE;
    let is_evt : boolean = IsZero(NOT(e) AND evfr);
    if PMSFCR_EL1().FE == '1' then
        // Filtering by event is enabled
        if !IsZero(evfr) then
            // Not a CONSTRAINED UNPREDICTABLE case
            is_rejected_event = !is_evt;
        else
            is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        end;
    end;

    // Filtering by inverse event
    var is_rejected_nevent : boolean = FALSE;
    var is_nevt : boolean;
    if IsFeatureImplemented(FEAT_SPE_FnE) then
        let nevfr : bits(64) = PMSNEVFR_EL1() AND mask;
        is_nevt = IsZero(e AND nevfr);
        if PMSFCR_EL1().FnE == '1' then
            // Inverse filtering by event is enabled
            if !IsZero(nevfr) then
                // Not a CONSTRAINED UNPREDICTABLE case
                is_rejected_nevent = !is_nevt;
            else
                is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
            end;
        end;
    else
        is_nevt = TRUE; // not implemented
    end;
    if is_evt && is_nevt then
        PMUEvent(PMU_EVENT_SAMPLE_FEED_EVENT);
    end;
    if (IsFeatureImplemented(FEAT_SPE_FnE) && PMSFCR_EL1().[FnE,FE]== '11' &&
          !IsZero(PMSEVFR_EL1() AND PMSNEVFR_EL1() AND mask)) then
        // CONSTRAINED UNPREDICTABLE case due to combination of filter and inverse filter
        is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    end;

    return (is_rejected_event, is_rejected_nevent);
end;
```

J1.2.1.83 SPEFilterByLatency

```
// SPEFilterByLatency()
// ====================
// Carries out filtering by latency.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func SPEFilterByLatency(total_latency : integer) => boolean
begin

    let is_lat : boolean = (total_latency >= UInt(PMSLATFR_EL1().MINLAT));
    if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT); end;

    var is_rejected_latency : boolean = FALSE;
    if PMSFCR_EL1().FL == '1' then
        // Filtering by latency is enabled
        if !IsZero(PMSLATFR_EL1().MINLAT) then
            // Not a CONSTRAINED UNPREDICTABLE case
            is_rejected_latency = is_rejected_latency || !is_lat;
        else
            is_rejected_latency = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        end;
    end;

    return is_rejected_latency;
end;
```

J1.2.1.84 SPEFilterByType

```
// SPEFilterByType()
// =================
// Returns TRUE if the operation is to be discarded because of its OpAttrs, and FALSE otherwise.

func SPEFilterByType(opattr : SPEOpAttr) => boolean
begin
    // Bit positions in the PMSFCR_EL1().[TYPE, TYPEm] fields
    let B : integer    = 0;
    let LD : integer   = 1;
    let ST : integer   = 2;
    let FP : integer   = 3;
    let SIMD : integer = 4;

    var flags : bits(5) = Zeros{};
    var ctrl : bits(5)  = Zeros{};
    // With GCS, Branch Link and Procedure Return instructions write and read the GCS.
    // BL and RET instructions with GCS enabled call SPESampleLoadStore() before
    // branch/packet construction, setting ldst_type to indicate a GCS access.
    let is_gcs_ldst : boolean = (opattr.op_type == SPEOpType_Branch &&
                                 opattr.ldst_type == SPELDSTType_GCS);
    let is_load : boolean     = (opattr.op_type IN {SPEOpType_Load, SPEOpType_LoadAtomic} ||
                                    (is_gcs_ldst && opattr.procedure_return));
    let is_store : boolean    = (opattr.op_type IN {SPEOpType_Store, SPEOpType_LoadAtomic} ||
                                    (is_gcs_ldst && opattr.branch_has_link));
    flags[B]    = (if opattr.op_type == SPEOpType_Branch then '1' else '0');
    flags[LD]   = (if is_load then '1' else '0');
    flags[ST]   = (if is_store then '1' else '0');
    flags[FP]   = (if opattr.is_floating_point then '1' else '0');
    flags[SIMD] = (if opattr.is_simd then '1' else '0');
    ctrl[2:0]   = PMSFCR_EL1().TYPE[2:0];
    var mask : bits(5) = Zeros{};
    if IsFeatureImplemented(FEAT_SPE_EFT) then
        ctrl[4:3] = PMSFCR_EL1().TYPE[4:3];
        mask[4:0] = PMSFCR_EL1().TYPEm;
    end;

    let ctrl_or : bits(5)  = (ctrl AND (NOT mask));
    let ctrl_and : bits(5) = (ctrl AND mask);

    let is_op : boolean = ((IsZero(ctrl_or) || !IsZero(flags AND ctrl_or)) &&
                               ((flags AND mask) == ctrl_and));

    if flags[B]    == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if flags[LD]   == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD); end;
    if flags[ST]   == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST); end;
    if flags[FP]   == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_FP); end;
    if flags[SIMD] == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_SIMD); end;

    var is_rejected_type : boolean = FALSE;
    if PMSFCR_EL1().FT == '1' then
        // Filtering by type is enabled
        if IsFeatureImplemented(FEAT_SPE_EFT) || !IsZero(ctrl) then
            is_rejected_type = !is_op;
        else
            is_rejected_type = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        end;
    end;
    if is_op && (!IsZero(ctrl) || !IsZero(mask)) then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP); end;

    return is_rejected_type;
end;
```

J1.2.1.85 SPEFreezeOnEvent

```
// SPEFreezeOnEvent()
// ==================
// Returns TRUE if PMU event counter idx should be frozen due to an SPE event, and FALSE otherwise.

func SPEFreezeOnEvent(idx : integer) => boolean
begin
    let counters : integer = NUM_PMU_COUNTERS;
    assert (idx >= 0 &&
            (idx < counters || idx == CYCLE_COUNTER_ID ||
             (idx == INSTRUCTION_COUNTER_ID && IsFeatureImplemented(FEAT_PMUv3_ICNTR))));

    if !IsFeatureImplemented(FEAT_SPEv1p2) || !IsFeatureImplemented(FEAT_PMUv3p7) then
        return FALSE;
    end;
    if idx == CYCLE_COUNTER_ID && !IsFeatureImplemented(FEAT_SPE_DPFZS) then
        // FZS does not affect the cycle counter when FEAT_SPE_DPFZS is not implemented
        return FALSE;
    end;

    let freeze_on_event : boolean = PMBLIMITR_EL1().[E,PMFZ] == '11';
    let stopped : boolean = SPEProfilingStopped();

    case GetPMUCounterRange(idx) of
        when PMUCounterRange_R1 =>
            return freeze_on_event && stopped && PMCR_EL0().FZS == '1';
        when PMUCounterRange_R2 =>
            return freeze_on_event && stopped && MDCR_EL2().HPMFZS == '1';
        otherwise =>
            return FALSE;
    end;
end;
```

J1.2.1.86 SPEGetDataSource

```
// SPEGetDataSource()
// ==================
// Returns a tuple indicating the data source for an access passed to SPESampleLoadStore, and
// whether the data source is valid.

impdef func SPEGetDataSource(is_load : boolean,
                             accdesc : AccessDescriptor,
                             addrdesc : AddressDescriptor) => (boolean, bits(16))
begin
    return (TRUE, Zeros{16});
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.1.87 SPEGetDataSourcePayloadSize

```
// SPEGetDataSourcePayloadSize()
// ============================
// Returns the size of the Data Source payload in bytes.

func SPEGetDataSourcePayloadSize() => integer{1..2}
begin
    return ImpDefInt("SPE Data Source packet payload size") as integer{1..2};
end;
```

J1.2.1.88 SPEGetEventsPayloadSize

```
// SPEGetEventsPayloadSize()
// =========================
// Returns the size in bytes of the Events packet payload as an integer.

func SPEGetEventsPayloadSize() => integer{1..4}
begin
    return ImpDefInt("SPE Events packet payload size") as integer{1..4};
end;
```

J1.2.1.89 SPEGetRandomBoolean

```
// SPEGetRandomBoolean()
// =====================
// Returns a random or pseudo-random boolean value.

impdef func SPEGetRandomBoolean() => boolean
begin
    return SPEGetRandomInterval()[0] == '1';
end;
```

J1.2.1.90 SPEGetRandomInterval

```
// SPEGetRandomInterval()
// ======================
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

impdef func SPEGetRandomInterval() => bits(8)
begin
    return _PC[0+:8];
end;
```

J1.2.1.91 SPEISB

```
// SPEISB()
// ========
// Called by ISB instruction, correctly calls SPEBranch to save previous branches.

func SPEISB()
begin
    let address : bits(64)           = PC64() + 4;
    let branch_type : BranchType     = BranchType_DIR;
    let branch_conditional : boolean = FALSE;
    let taken : boolean              = FALSE;
    let is_isb : boolean             = TRUE;

    SPEBranch{64}(address, branch_type, branch_conditional, taken, is_isb);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.92 SPEInterruptEnabled

```
// SPEInterruptEnabled()
// =====================
// Return TRUE if the SPE interrupt request (PMBIRQ) is enabled, FALSE otherwise.

func SPEInterruptEnabled() => boolean
begin
    return EffectivePMSCR_EL1_EE() == '00';
end;
```

J1.2.1.93 SPELDSTType

```
// SPELDSTType
// ===========
// Type of a load or store operation.

type SPELDSTType of enumeration {
    SPELDSTType_NV2,
    SPELDSTType_Extended,
    SPELDSTType_General,
    SPELDSTType_SIMDFP,
    SPELDSTType_SVESME,
    SPELDSTType_Tags,
    SPELDSTType_MemCopy,
    SPELDSTType_MemSet,
    SPELDSTType_GCS,
    SPELDSTType_GCSSS2,
    SPELDSTType_Unspecified
};
```

J1.2.1.94 SPEMultiAccessSample

```
// SPEMultiAccessSample()
// ======================
// Called by instructions which make at least one store and one load access, where the configuration
// of the operation type filter affects which access is sampled.

func SPEMultiAccessSample() => boolean
begin
    // If loads or stores are filtered out, the other should be recorded.
    // If neither or both are filtered out, pick one in an unbiased way.

    // Bit positions in the PMSFCR_EL1().[TYPE, TYPEm] fields.
    let LD : integer = 1;
    let ST : integer = 2;

    // Are loads allowed by filter?
    let loads_pass_filter : boolean  = PMSFCR_EL1().FT == '1' && PMSFCR_EL1().TYPE[LD] == '1';
    // Are stores allowed by filter?
    let stores_pass_filter : boolean = PMSFCR_EL1().FT == '1' && PMSFCR_EL1().TYPE[ST] == '1';

    var record_load : boolean;
    if loads_pass_filter && !stores_pass_filter then
        // Only loads pass filter
        record_load = TRUE;
    elsif !loads_pass_filter && stores_pass_filter then
        // Only stores pass filter
        record_load = FALSE;
    else
        // Both loads and stores pass the filter or neither pass the filter.
        // Pick between the load or the store access (pseudo-)randomly.
        record_load = SPEGetRandomBoolean();
    end;
    return record_load;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.95 SPEOpAttr

```
// SPEOpAttr
// =========
// Attributes of sampled operation filtered by SPECollectRecord().

type SPEOpAttr of record {
    op_type : SPEOpType,
    ldst_type : SPELDSTType,
    branch_is_direct : boolean,
    branch_has_link : boolean,
    procedure_return : boolean,
    is_conditional : boolean,
    is_floating_point : boolean,
    is_simd : boolean,
    cond_pass : boolean,
    at : boolean,
    is_acquire_release : boolean,
    is_predicated : boolean,
    evl : bits(3),
    is_gather_scatter : boolean,
    is_exclusive : boolean,
    ets : bits(4),
    addr_valid : boolean
};
```

J1.2.1.96 SPEOpType

```
// SPEOpType
// =========
// Types of operation filtered by SPECollectRecord().

type SPEOpType of enumeration {
    SPEOpType_Load,        // Any memory-read operation other than atomics, compare-and-swap,
                           // and swap
    SPEOpType_Store,       // Any memory-write operation, including atomics without return
    SPEOpType_LoadAtomic,  // Atomics with return, compare-and-swap and swap
    SPEOpType_Branch,      // Software write to the PC
    SPEOpType_OtherSVE,    // Other SVE operation
    SPEOpType_OtherSME,    // Other SME operation
    SPEOpType_Other,       // Any other class of operation
    SPEOpType_Invalid
};
```

J1.2.1.97 SPEProfilingStopped

```
// SPEProfilingStopped()
// =====================

readonly func SPEProfilingStopped() => boolean
begin
    var stopped : boolean = (PMBSR_EL1().S == '1');
    if IsFeatureImplemented(FEAT_SPE_EXC) then
        if HaveEL(EL3) && MDCR_EL3().PMSEE == '1x' then
            stopped = stopped || (PMBSR_EL3().S == '1');
        end;
        if EffectivePMSCR_EL2_EE() == '1x' then
            stopped = stopped || (PMBSR_EL2().S == '1');
        end;
    end;
    return stopped;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.98 SPEResetSampleCounter

```
// SPEResetSampleCounter()
// =======================
// Reset PMSICR_EL1.Counter

func SPEResetSampleCounter()
begin
    PMSICR_EL1().COUNT[31:8] = PMSIRR_EL1().INTERVAL;
    if PMSIRR_EL1().RND == '1' && !IsFeatureImplemented(FEAT_SPE_ERnd) then
        PMSICR_EL1().COUNT[7:0] = SPEGetRandomInterval();
    else
        PMSICR_EL1().COUNT[7:0] = Zeros{8};
    end;
end;
```

J1.2.1.99 SPEResetSampleStorage

```
// SPEResetSampleStorage()
// =======================
// Reset all variables inside sample storage.

func SPEResetSampleStorage()
begin
    // Context values
    SPESampleContextEL1      = Zeros{32};
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2      = Zeros{32};
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1) do
        SPESampleCounter[[i]] = 0;
        SPESampleCounterValid[[i]] = FALSE;
        SPESampleCounterPending[[i]] = FALSE;
    end;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1) do
        SPESampleAddressValid[[i]] = FALSE;
        SPESampleAddress[[i]] = Zeros{64};
    end;

    // Data source values
    SPESampleDataSource      = Zeros{16};
    SPESampleDataSourceValid = FALSE;

    // Timestamp values
    SPESampleTimestamp      = Zeros{64};
    SPESampleTimestampValid = FALSE;

    // Event values
    SPESampleEvents = Zeros{64};

    // Operation attributes
    SPESampleOpAttr.op_type = SPEOpType_Invalid;
    SPESampleOpAttr.ldst_type          = SPELDSTType_Unspecified;
    SPESampleOpAttr.branch_is_direct   = FALSE;
    SPESampleOpAttr.branch_has_link    = FALSE;
    SPESampleOpAttr.procedure_return   = FALSE;
    SPESampleOpAttr.is_conditional     = FALSE;
    SPESampleOpAttr.is_floating_point  = FALSE;
    SPESampleOpAttr.is_simd            = FALSE;
    SPESampleOpAttr.cond_pass          = FALSE;
    SPESampleOpAttr.at                 = FALSE;
    SPESampleOpAttr.is_acquire_release = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SPESampleOpAttr.is_exclusive       = FALSE;
    SPESampleOpAttr.is_predicated      = FALSE;
    SPESampleOpAttr.evl                = '000';
    SPESampleOpAttr.is_gather_scatter  = FALSE;
    SPESampleOpAttr.addr_valid         = FALSE;
end;

var SPERecordData : array [[SPEMaxRecordSize]] of bits(8);
var SPERecordSize : integer;
```

J1.2.1.100 SPESampleAddAddressPCVirtual

```
// SPESampleAddAddressPCVirtual()
// ==============================
// Save the current PC address to sample storage.

func SPESampleAddAddressPCVirtual()
begin
    var ns : bit;
    var nse : bit;
    case CurrentSecurityState() of
        when SS_Secure =>
            ns = '0';
            nse = '0';
        when SS_NonSecure =>
            ns = '1';
            nse = '0';
        when SS_Realm =>
            ns = '1';
            nse = '1';
        otherwise => unreachable;
    end;
    let el : bits(2) = PSTATE.EL;

    let pc : bits(64) = ThisInstrAddr{}();
    SPESampleAddress[[SPEAddrPosPCVirtual]][55:0] = pc[55:0];
    SPESampleAddress[[SPEAddrPosPCVirtual]][63:56] = ns::el::nse::Zeros{4};
    SPESampleAddressValid[[SPEAddrPosPCVirtual]]   = TRUE;
end;
```

J1.2.1.101 SPESampleAddContext

```
// SPESampleAddContext()
// =====================
// Save contexts to sample storage if appropriate.

func SPESampleAddContext()
begin
    if CollectContextIDR1() then
        SPESampleContextEL1      = CONTEXTIDR_EL1()[31:0];
        SPESampleContextEL1Valid = TRUE;
    end;
    if CollectContextIDR2() then
        SPESampleContextEL2      = CONTEXTIDR_EL2()[31:0];
        SPESampleContextEL2Valid = TRUE;
    end;
end;
```

J1.2.1.102 SPESampleAddTimeStamp

```
// SPESampleAddTimeStamp()
// =======================
// Save the appropriate type of timestamp to sample storage.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func SPESampleAddTimeStamp()
begin
    let timestamp : TimeStamp = CollectTimeStamp();
    case timestamp of
        when TimeStamp_None =>
            SPESampleTimestampValid = FALSE;
        otherwise =>
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimestamp(timestamp);
    end;
end;
```

J1.2.1.103 SPESampleCollision

```
// SPESampleCollision()
// ====================
// Called when there is an SPE sample collision.

func SPESampleCollision()
begin
    // Sample collision with the previous sample
    PMUEvent(PMU_EVENT_SAMPLE_COLLISION);
    let target_el : bits(2)  = DefaultSPEEvent();
    PMBSR_EL(target_el).COLL = '1';
end;
```

J1.2.1.104 SPESampleExtendedLoadStore

```
// SPESampleExtendedLoadStore()
// ============================
// Sets the subclass of the operation type packet for
// extended load/store operations.

func SPESampleExtendedLoadStore(ar : boolean, excl : boolean, at : boolean, is_load : boolean)
begin
    SPESampleOpAttr.is_acquire_release = ar;
    SPESampleOpAttr.is_exclusive       = excl;
    SPESampleOpAttr.ldst_type          = SPELDSTType_Extended;
    SPESampleOpAttr.at                 = at;
    if is_load then
        if at then
            SPESampleOpAttr.op_type = SPEOpType_LoadAtomic;
        else
            SPESampleOpAttr.op_type = SPEOpType_Load;
        end;
    else
        SPESampleOpAttr.op_type = SPEOpType_Store;
    end;
end;
```

J1.2.1.105 SPESampleGCSSS2

```
// SPESampleGCSSS2()
// =================
// Sets the subclass of the operation type packet for GCSSS2 load/store operations.

func SPESampleGCSSS2()
begin
    // GCSSS2 does a read and a write.
    let record_load : boolean = SPEMultiAccessSample();
    SPESampleOpAttr.op_type   = if record_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_GCSSS2;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.106 SPESampleGeneralPurposeLoadStore

```
// SPESampleGeneralPurposeLoadStore()
// ==================================
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

func SPESampleGeneralPurposeLoadStore(is_load : boolean)
begin
    SPESampleOpAttr.ldst_type = SPELDSTType_General;
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
end;
```

J1.2.1.107 SPESampleLoadStore

```
// SPESampleLoadStore()
// ====================
// Called if a sample is in flight when writing or reading memory,
// indicating that the operation being sampled is in the Load, Store or atomic category.

func SPESampleLoadStore(is_load : boolean, accdesc : AccessDescriptor,
                        addrdesc : AddressDescriptor)
begin
    // Check if this access type is suitable to be sampled.
    // This implementation of SPE always samples the first access made by a suitable instruction.
    // FEAT_MOPS instructions are an exception, where the first load or first store access may be
    // selected based on the configuration of the sample filters.
    if accdesc.acctype IN {AccessType_SPE,
                           AccessType_IFETCH,
                           AccessType_TRBE,
                           AccessType_DC,
                           AccessType_TTW,
                           AccessType_AT} then
        return;
    end;

    var sample_access : boolean = FALSE;
    // For FEAT_MOPS and FEAT_GCS GCSSS2 instructions which perform both loads and stores, the
    // filter configuration will influence which part of the access is chosen to be sampled.
    if (SPESampleOpAttr.ldst_type IN
          {SPELDSTType_MemCopy, SPELDSTType_MemSet, SPELDSTType_GCSSS2}) then
        // SPEMultiAccessSample() will have been called before this function,
        // and chooses whether to sample a load or a store.
        var sample_load : boolean;
        sample_load = SPESampleOpAttr.op_type IN {SPEOpType_Load, SPEOpType_LoadAtomic};

        // If no valid data has been collected, and this operation is acceptable for sampling.
        if !SPESampleOpAttr.addr_valid && (is_load == sample_load) then
            sample_access = TRUE;
        end;
    else
        if !SPESampleOpAttr.addr_valid then
            sample_access = TRUE;
        end;
    end;

    if sample_access then
        // Data access virtual address
        SPESetDataVirtualAddress(addrdesc.vaddress);

        // Data access physical address
        if CollectPhysicalAddress() then
            SPESetDataPhysicalAddress(addrdesc, accdesc);
        end;

        SPESampleOpAttr.addr_valid = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if SPESampleOpAttr.op_type == SPEOpType_Invalid then
        // Set as unspecified load/store by default, instructions will overwrite this if it does not
        // apply to them.
        SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;

        if accdesc.acctype == AccessType_NV2 then
            // NV2 register load/store
            SPESampleOpAttr.ldst_type = SPELDSTType_NV2;
        end;
    end;

    // Set SPELDSTType to GCS for all GCS instruction, overwriting type GCSSS2.
    // After selection of which operation of a GCSSS2 instruction to sample, GCSSS2 is treated the
    // same as other GCS instructions.
    if accdesc.acctype == AccessType_GCS then
        SPESampleOpAttr.ldst_type = SPELDSTType_GCS;
        // If the GCS access is from a BL or RET, this will get overwritten to SPEOpType_Branch.
        SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
    end;
    (SPESampleDataSourceValid, SPESampleDataSource) = SPEGetDataSource(is_load, accdesc, addrdesc);
end;
```

J1.2.1.108 SPESampleMemCopy

```
// SPESampleMemCopy()
// ==================
// Sets the subclass of the operation type packet for Memory Copy load/store
// operations.

func SPESampleMemCopy()
begin
    // MemCopy does a read and a write.
    let record_load : boolean = SPEMultiAccessSample();
    SPESampleOpAttr.op_type   = if record_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_MemCopy;
end;
```

J1.2.1.109 SPESampleMemSet

```
// SPESampleMemSet()
// =================
// Callback used by memory set instructions to pass data back to the SPU.

func SPESampleMemSet()
begin
    SPESampleOpAttr.op_type   = SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_MemSet;
end;
```

J1.2.1.110 SPESampleOnSharedResource

```
// SPESampleOnSharedResource()
// ===========================
// Called when the sampled instruction is executed on an SMCU or other shared resource, sets bit 25
// of the events packet to 0b1.
func SPESampleOnSharedResource()
begin
    SPESampleEvents[25] = '1';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.1.111 SPESampleOpOther

```
// SPESampleOpOther()
// ==================
// Add other operation to sample storage.

func SPESampleOpOther(conditional : boolean, cond_pass : boolean,
                      is_fp : boolean, is_simd : boolean)
begin
    SPESampleOpAttr.is_simd = is_simd;
    SPESampleOpOther(conditional, cond_pass, is_fp);
end;

func SPESampleOpOther(conditional : boolean, cond_pass : boolean, is_fp : boolean)
begin
    SPESampleOpAttr.cond_pass = cond_pass;
    SPESampleOpAttr.is_floating_point = is_fp;
    SPESampleOpOther(conditional);
end;

func SPESampleOpOther(conditional : boolean)
begin
    SPESampleOpAttr.is_conditional = conditional;
    SPESampleOpAttr.op_type = SPEOpType_Other;
end;
```

J1.2.1.112 SPESampleOpSMEArrayOther

```
// SPESampleOpSMEArrayOther()
// ==========================
// Sets the subclass of the operation type packet to Other, SME array.

func SPESampleOpSMEArrayOther(floating_point : boolean, size : integer)
begin
    // If the sampled effective vector or tile size is not a power of two, or is less than 128 bits,
    // the value is rounded up before it is encoded in the ets field.

    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.ets               = SPEEncodeETS(size);
    SPESampleOpAttr.op_type           = SPEOpType_OtherSME;
    SPESampleOpAttr.is_simd           = TRUE;
end;
```

J1.2.1.113 SPESampleOpSVEOther

```
// SPESampleOpSVEOther()
// =====================
// Callback used by SVE, Other operations to pass data back to the SPU.

func SPESampleOpSVEOther(vl : integer, predicated : boolean, floating_point : boolean)
begin
    SPESampleOpAttr.is_predicated     = predicated;
    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.evl               = SPEEncodeEVL(vl);
    SPESampleOpAttr.op_type           = SPEOpType_OtherSVE;
end;
```

J1.2.1.114 SPESampleOpSVESMELoadStore

```
// SPESampleOpSVESMELoadStore()
// ============================
// Callback used by SVE or SME loads and stores to pass data to SPE.

func SPESampleOpSVESMELoadStore(is_gather_scatter : boolean, vl : integer, predicated : boolean,
                                is_load : boolean)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    SPESampleOpAttr.is_gather_scatter = is_gather_scatter;
    SPESampleOpAttr.is_predicated     = predicated;
    SPESampleOpAttr.evl               = SPEEncodeEVL(vl);
    assert SPESampleOpAttr.evl != '111';
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_SVESME;
end;
```

J1.2.1.115 SPESampleSIMDFPLoadStore

```
// SPESampleSIMDFPLoadStore()
// ==========================
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

func SPESampleSIMDFPLoadStore(is_load : boolean, scalar : boolean)
begin
    SPESampleOpAttr.ldst_type = SPELDSTType_SIMDFP;
    SPESampleOpAttr.op_type   = if is_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.is_simd   = !scalar;
    // Scalar operations in SIMD&FP are treated as floating point.
    SPESampleOpAttr.is_floating_point = scalar;
end;
```

J1.2.1.116 SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// ==========================
// Called from SampleLoadStore() to save data physical packet.

func SPESetDataPhysicalAddress(addrdesc : AddressDescriptor, accdesc : AccessDescriptor)
begin
    var ns : bit;
    var nse : bit;
    var ch : bit;
    var lat : bits(4);
    case addrdesc.paddress.paspace of
        when PAS_Secure =>
            ns = '0';
            nse = '0';
        when PAS_NonSecure =>
            ns = '1';
            nse = '0';
        when PAS_Realm =>
            ns  = '1';
            nse = '1';
        otherwise => unreachable;
    end;

    if IsFeatureImplemented(FEAT_MTE2) then
        if accdesc.tagchecked then
            ch = '1';
            lat = AArch64_LogicalAddressTag(addrdesc.vaddress);
        else
            ch = '0';
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED choice
            // between 0b0000 and the Logical Address Tag
            var zero_unchecked : boolean;
            zero_unchecked = ImpDefBool("SPE PAT for tag unchecked access zero");
            if !zero_unchecked then
                lat = AArch64_LogicalAddressTag(addrdesc.vaddress);
            else
                lat = Zeros{4};
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else
        ch = '0';
        lat = '0000';
    end;

    let paddr : bits(NUM_PABITS) = addrdesc.paddress.address;
    SPESampleAddress[[SPEAddrPosDataPhysical]][55:0] = ZeroExtend{56}(paddr);
    SPESampleAddress[[SPEAddrPosDataPhysical]][63:56] = ns::ch::'0'::nse::lat;
    SPESampleAddressValid[[SPEAddrPosDataPhysical]] = TRUE;
end;
```

J1.2.1.117 SPESetDataVirtualAddress

```
// SPESetDataVirtualAddress()
// ==========================
// Called from SampleLoadStore() to save data virtual packet.
// Also used by exclusive load/stores to save virtual addresses if exclusive monitor is lost
// before a read/write is completed.

func SPESetDataVirtualAddress(vaddress : bits(64))
begin
    var tbi : bit;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
    var non_tbi_is_zeros : boolean;
    non_tbi_is_zeros = ImpDefBool("SPE non-tbi tag is zero");
    if tbi == '1' || !non_tbi_is_zeros then
        SPESampleAddress[[SPEAddrPosDataVirtual]][63:0]  = vaddress[63:0];
    else
        SPESampleAddress[[SPEAddrPosDataVirtual]][55:0]  = vaddress[55:0];
        SPESampleAddress[[SPEAddrPosDataVirtual]][63:56] = Zeros{8};
    end;
    SPESampleAddressValid[[SPEAddrPosDataVirtual]]       = TRUE;
end;
```

J1.2.1.118 SPEStartCounter

```
// SPEStartCounter()
// =================
// Enables incrementing of the counter at the passed index when SPECycle is called.

func SPEStartCounter(counter_index : integer)
begin
    assert counter_index < SPEMaxCounters;
    SPESampleCounterPending[[counter_index]] = TRUE;
end;
```

J1.2.1.119 SPEStartSample

```
// SPEStartSample()
// ================
// Called to start the sampling process. Returns TRUE if a new sample is to be collected, and
// FALSE otherwise.

func SPEStartSample() => boolean
begin
    if !StatisticalProfilingEnabled() then
        return FALSE;
    end;
    PMUEvent(PMU_EVENT_SAMPLE_POP);
    if !SPEToCollectSample() then
        return FALSE;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if SPESampleInFlight then
        SPESampleCollision();
        return FALSE;
    end;

    SPESampleInFlight = TRUE;

    SPEStartCounter(SPECounterPosTotalLatency);
    SPEStartCounter(SPECounterPosIssueLatency);
    SPESampleAddAddressPCVirtual();

    // Many operations are type other and not conditional, can save footprint
    // and overhead by having this as the default and not calling SPESampleOpOther
    // if conditional == FALSE
    SPESampleOpOther(FALSE);

    SPESampleAddContext();

    // Timestamp may be collected at any point in the sampling operation.
    // Collecting prior to execution is one possible choice.
    // This choice is IMPLEMENTATION DEFINED.
    SPESampleAddTimeStamp();

    return TRUE;
end;
```

J1.2.1.120 SPEStopCounter

```
// SPEStopCounter()
// ================
// Disables incrementing of the counter at the passed index when SPECycle is called.

func SPEStopCounter(counter_index : integer)
begin
    SPESampleCounterValid[[counter_index]] = TRUE;
    SPESampleCounterPending[[counter_index]] = FALSE;
end;
```

J1.2.1.121 SPEToCollectSample

```
// SPEToCollectSample()
// ====================
// Returns TRUE if the instruction which is about the executed should be sampled, and FALSE
// otherwise.

func SPEToCollectSample() => boolean
begin
    if IsZero(PMSICR_EL1().COUNT) then
        SPEResetSampleCounter();
    else
        PMSICR_EL1().COUNT = PMSICR_EL1().COUNT - 1;
        if IsZero(PMSICR_EL1().COUNT) then
            if PMSIRR_EL1().RND == '1' && IsFeatureImplemented(FEAT_SPE_ERnd) then
                PMSICR_EL1().ECOUNT = SPEGetRandomInterval();
                return IsZero(PMSICR_EL1().ECOUNT);
            else
                return TRUE;
            end;
        end;
    end;
    if (PMSIRR_EL1().RND == '1' && IsFeatureImplemented(FEAT_SPE_ERnd) &&
          !IsZero(PMSICR_EL1().ECOUNT)) then
        PMSICR_EL1().ECOUNT = PMSICR_EL1().ECOUNT - 1;
        if IsZero(PMSICR_EL1().ECOUNT) then
            return TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
    return FALSE;
end;
```

J1.2.1.122 SPEWriteToBuffer

```
// SPEWriteToBuffer()
// ==================
// Write the active record to the Profiling Buffer.

func SPEWriteToBuffer()
begin
    assert ProfilingBufferEnabled() && !SPEProfilingStopped();

    // Check alignment
    let align : integer{} = UInt(PMBIDR_EL1().Align);
    let aligned : boolean = IsAlignedP2(PMBPTR_EL1().PTR, align);

    let ttw_abort_as_fault : boolean = (ImpDefBool(
                                        "Report SPE ExtAbort on TTW as fault"));

    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = ProfilingBufferOwner();
    let accdesc : AccessDescriptor = CreateAccDescSPE(owning_ss, owning_el);

    let start_vaddr : bits(64) = PMBPTR_EL1();
    for i = 0 to SPERecordSize - 1 do
        // If a previous write did not cause an issue
        if !SPEProfilingStopped() then
            let address : bits(64) = PMBPTR_EL1();
            var memstatus : PhysMemRetStatus;
            var addrdesc : AddressDescriptor;
            (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned,
                                                  SPERecordData[[i]]);

            let fault : FaultRecord = addrdesc.fault;
            let ttw_abort : boolean = fault.statuscode IN {Fault_SyncExternalOnWalk,
                                                           Fault_SyncParityOnWalk};

            if IsFault(fault.statuscode) && (!ttw_abort || ttw_abort_as_fault) then
                DebugWriteFault(address, fault);
            elsif IsFault(memstatus) || (ttw_abort && !ttw_abort_as_fault) then
                DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);
            end;

            // Move pointer if no Buffer Management Event has been caused.
            if !SPEProfilingStopped() then
                PMBPTR_EL1() = PMBPTR_EL1() + 1;
            end;
        end;
    end;

    return;
end;
```

J1.2.1.123 StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =============================
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

func StatisticalProfilingEnabled() => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return StatisticalProfilingEnabled(PSTATE.EL);
end;

// StatisticalProfilingEnabled()
// ============================
// Return TRUE if Statistical Profiling is Enabled in the specified EL, FALSE otherwise.

func StatisticalProfilingEnabled(el : bits(2)) => boolean
begin
    if (!IsFeatureImplemented(FEAT_SPE) || UsingAArch32() ||
          !ProfilingBufferEnabled() || SPEProfilingStopped() || Halted()) then
        return FALSE;
    end;

    let tge_set : boolean = EL2Enabled() && HCR_EL2().TGE == '1';
    let (owning_ss, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();
    if (UInt(owning_el) <  UInt(el) || (tge_set && owning_el == EL1) ||
          owning_ss != SecurityStateAtEL(el))  then
        return FALSE;
    end;
    var spe_bit : bit;
    case el of
        when EL3 =>  unreachable;
        when EL2 =>  spe_bit = PMSCR_EL2().E2SPE;
        when EL1 =>  spe_bit = PMSCR_EL1().E1SPE;
        when EL0 =>  spe_bit = (if tge_set then PMSCR_EL2().E0HSPE else PMSCR_EL1().E0SPE);
    end;

    return spe_bit == '1';
end;
```

J1.2.1.124 TimeStamp

```
// TimeStamp
// =========

type TimeStamp of enumeration {
    TimeStamp_None,              // No timestamp
    TimeStamp_CoreSight,         // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,          // Physical counter value with no offset
    TimeStamp_OffsetPhysical,    // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual  };        // Physical counter value minus CNTVOFF_EL2
```

J1.2.1.125 AArch64_TakeExceptionInDebugState

```
// AArch64_TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception level using AArch64.

noreturn func AArch64_TakeExceptionInDebugState(target_el : bits(2), exception_in : ExceptionRecord)
begin
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR().SDD == '0';
    var except : ExceptionRecord = exception_in;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_EL(target_el).EIS == '1' then
        // Synchronize the context, including Instruction Fetch Barrier effect
        SynchronizeContext();
    end;

    // If coming from AArch32 state, the top parts of the X() registers might be set to zero
    let from_32 : boolean = UsingAArch32();
    if from_32 then AArch64_MaybeZeroRegisterUppers(); end;
    if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        ResetSVEState();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        MaybeZeroSVEUppers(target_el);
    end;

    AArch64_ReportException(except, target_el);

    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0';  // Effective value of GCSCR_ELx.EXLOCKEN is 0 in Debug state
    end;

    PSTATE.EL  = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP  = '1';

    SPSR_ELx() = ARBITRARY : bits(64);
    ELR_ELx() = ARBITRARY : bits(64);
    // PSTATE.[SS,D,A,I,F] are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.[SS,D,A,I,F] = ARBITRARY : bits(5);
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = ARBITRARY : bit; end;
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
          (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
          SCTLR_ELx().SPAN == '0') then
        PSTATE.PAN = '1';
    end;
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    end;
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00'; end;
    DLR_EL0() = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    EDSCR().ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.2.1.126 AArch64_WatchpointByteMatch

```
// AArch64_WatchpointByteMatch()
// ============================

func AArch64_WatchpointByteMatch(n : integer, vaddress : bits(64)) => boolean
begin
    let dbgtop : AddressSize        = DebugAddrTop();
    let cmpbottom : integer{}       = if DBGWVR_EL1(n)[2] == '1' then 2 else 3;// Word or doubleword
    var bottom : integer{}          = cmpbottom;
    let select : integer{}          = UInt(vaddress[cmpbottom-1:0]);
    var byte_select_match : boolean = (DBGWCR_EL1(n).BAS[select] != '0');
    var mask : integer{}            = UInt(DBGWCR_EL1(n).MASK);

    // If DBGWCR_EL1(n).MASK is a nonzero value and DBGWCR_EL1(n).BAS is not set to '11111111', or
    // DBGWCR_EL1(n).BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1(n).BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        let LSB : bits(8) = (DBGWCR_EL1(n).BAS AND NOT(DBGWCR_EL1(n).BAS - 1));
        let MSB : bits(8) = (DBGWCR_EL1(n).BAS + LSB);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !IsZero(MSB AND (MSB - 1)) then          // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom            = 3;                  // For the whole doubleword
        end;
    end;

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        var c : Constraint;
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED => return FALSE; // Disabled
            when Constraint_NONE     => mask = 0;     // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
            otherwise                => mask = unpred_mask as integer{3..31};
        end;
    end;

    // When FEAT_LVA3 is not implemented, if the DBGWVR_EL1(n).RESS field bits are not a
    // sign extension of the MSB of DBGWVR_EL1(n).VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    let unpredictable_ress : boolean = (dbgtop < 55 && !IsOnes(DBGWVR_EL1(n)[63:dbgtop]) &&
                                        !IsZero(DBGWVR_EL1(n)[63:dbgtop]) &&
                                        ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
    let cmpmsb : integer{}    = if unpredictable_ress then 63 else dbgtop;
    let cmplsb : integer{}    = if mask > bottom then mask else bottom;
    let bottombit : integer{} = bottom;
    var WVR_match : boolean   = (vaddress[cmpmsb:cmplsb] == DBGWVR_EL1(n)[cmpmsb:cmplsb]);
    if mask > bottom then
        // If masked bits of DBGWVR_EL1(n) are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1(n)[cmplsb-1:bottombit]) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
        end;
    end;

    return (WVR_match && byte_select_match);
end;
```

J1.2.1.127 AArch64_WatchpointMatch

```
// AArch64_WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

func AArch64_WatchpointMatch(n : integer, vaddress : bits(64), size : integer,
                             accdesc : AccessDescriptor) => WatchpointInfo
begin
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    let enabled : boolean     = IsWatchpointEnabled(n);
    let linked : boolean      = DBGWCR_EL1(n).WT == '1';
    let isbreakpnt : boolean  = FALSE;
    let lbnx : bits(2) = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGWCR_EL1(n).LBNX else '00';
    let linked_n : integer{}  = UInt(lbnx :: DBGWCR_EL1(n).LBN);
    let ssce : bit            = if IsFeatureImplemented(FEAT_RME) then DBGWCR_EL1(n).SSCE else '0';
    let mismatch : boolean    = IsFeatureImplemented(FEAT_BWE2) && DBGWCR_EL1(n).WT2 == '1';
    let state_match : boolean = AArch64_StateMatch(DBGWCR_EL1(n).SSC, ssce, DBGWCR_EL1(n).HMC,
                                                   DBGWCR_EL1(n).PAC,
                                                   linked, linked_n, isbreakpnt, PC64(), accdesc);

    var watchptinfo : WatchpointInfo;
    var ls_match : boolean;
    case DBGWCR_EL1(n).LSC[1:0] of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '00' => ls_match = FALSE;
        when '01' => ls_match = accdesc.read;
        when '10' => ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' => ls_match = TRUE;
    end;

    var value_match : boolean = FALSE;
    watchptinfo.vaddress = vaddress;
    for byte = 0 to size - 1 do
        if (!value_match &&
              !AddressInNaturallyAlignedBlock(watchptinfo.vaddress, vaddress + byte)) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = vaddress + byte;
        end;
        value_match = value_match || AArch64_WatchpointByteMatch(n, vaddress + byte);
    end;

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match;
    if !(state_match && ls_match && enabled) then
        watchptinfo.wptype = WatchpointType_Inactive;
        watchptinfo.value_match = FALSE;
    elsif mismatch then
        watchptinfo.wptype = WatchpointType_AddrMismatch;
    else
        watchptinfo.wptype = WatchpointType_AddrMatch;
    end;
    return watchptinfo;
end;
```

J1.2.1.128 IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGWCR_EL1(n).E is '1', and FALSE otherwise.

func IsWatchpointEnabled(n : integer) => boolean
begin
    if (n > 15 &&
          ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEnabled()) ||
           (HaltOnBreakpointOrWatchpoint() && EDSCR2().EHBWE == '0'))) then
        return FALSE;
    end;
    return DBGWCR_EL1(n).E == '1';
end;
```

J1.2.2 aarch64/exceptions

This section includes the following pseudocode functions:

- AArch64_Abort
- AArch64_AbortSyndrome
- AArch64_CheckPCAlignment
- AArch64_DataAbort
- AArch64_EffectiveTCF
- AArch64_InstructionAbort
- AArch64_PCAlignmentFault
- AArch64_RaiseTagCheckFault
- AArch64_ReportTagCheckFault

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64_RouteToSErrorOffset
- AArch64_SPAlignmentFault
- BranchTargetException
- TCFType
- TakeGPCException
- AArch64_TakeDelegatedSErrorException
- AArch64_TakePhysicalFIQException
- AArch64_TakePhysicalIRQException
- AArch64_TakePhysicalSErrorException
- AArch64_TakeVirtualFIQException
- AArch64_TakeVirtualIRQException
- AArch64_TakeVirtualSErrorException
- AArch64_BreakpointException
- AArch64_SoftwareBreakpoint
- AArch64_SoftwareStepException
- AArch64_VectorCatchException
- AArch64_WatchpointException
- AArch64_ExceptionClass
- AArch64_ReportException
- AArch64_ResetControlRegisters
- AArch64_TakeReset
- GetESR_ELx_DFSC
- GetESR_ELx_ExType
- GetESR_ELx_IFSC
- AArch64_FPTrappedException
- AArch64_CallHypervisor
- AArch64_CallSecureMonitor
- AArch64_CallSupervisor
- AArch64_TakeException
- AArch64_AArch32SystemAccessTrap
- AArch64_AArch32SystemAccessTrapSyndrome
- AArch64_AdvSIMDFPAccessTrap
- AArch64_CheckCP15InstrCoarseTraps
- AArch64_CheckFPAdvSIMDEnabled
- AArch64_CheckFPAdvSIMDTrap
- AArch64_CheckFPEnabled
- AArch64_CheckForERetTrap
- AArch64_CheckForSMCUndefOrTrap
- AArch64_CheckForSVCTrap
- AArch64_CheckForWFxTrap
- AArch64_CheckIllegalState
- AArch64_MonitorModeTrap
- AArch64_OtherInstrTrap
- AArch64_SystemAccessTrap
- AArch64_SystemAccessTrapSyndrome
- AArch64_Undefined
- AArch64_WFxTrap

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckLDST64BEnabled
- CheckST64BV0Enabled
- CheckST64BVEnabled
- WFETrapDelay
- WaitForEventUntilDelay

J1.2.2.1 AArch64_Abort

```
// AArch64_Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

func AArch64_Abort(fault : FaultRecord)
begin
    if IsDebugException(fault) then
        if fault.accessdesc.acctype == AccessType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64_VectorCatchException(fault);
            else
                AArch64_BreakpointException(fault);
            end;
        else
            AArch64_WatchpointException(fault);
        end;
    elsif fault.gpcf.gpf != GPCF_None && ReportAsGPCException(fault) then
        TakeGPCException(fault);
    elsif fault.statuscode == Fault_TagCheck then
        AArch64_RaiseTagCheckFault(fault);
    elsif IsExternalAbort(fault) && !IsExternalSyncAbort(fault) then
        PendSErrorInterrupt(fault);
    elsif fault.accessdesc.acctype == AccessType_IFETCH then
        AArch64_InstructionAbort(fault);
    else
        AArch64_DataAbort(fault);
    end;
end;
```

J1.2.2.2 AArch64_AbortSyndrome

```
// AArch64_AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.
func AArch64_AbortSyndrome(exceptype : Exception, fault : FaultRecord,
                           target_el : bits(2)) => ExceptionRecord
begin
    var except : ExceptionRecord = ExceptionSyndrome(exceptype);

    except.syndrome = AArch64_FaultSyndrome(exceptype, fault, target_el);
    if exceptype IN {Exception_NV2Watchpoint, Exception_Watchpoint} then
        except.vaddress = fault.watchptinfo.vaddress;
    elsif fault.statuscode == Fault_TagCheck then
        if IsFeatureImplemented(FEAT_MTE_TAGGED_FAR) then
            except.vaddress = ZeroExtend{64}(fault.vaddress);
        else
            except.vaddress = ARBITRARY : bits(4) :: fault.vaddress[59:0];
        end;
    else
        except.vaddress = ZeroExtend{64}(fault.vaddress);
    end;

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;
    end;

    if except.syndrome.iss[14] == '1' then
        except.pavalid  = TRUE;
        except.paddress = fault.paddress;
    else
        except.pavalid  = FALSE;
        except.paddress = ARBITRARY : FullAddress;
    end;

    return except;
end;
```

J1.2.2.3 AArch64_CheckPCAlignment

```
// AArch64_CheckPCAlignment()
// ==========================

func AArch64_CheckPCAlignment()
begin
    let pc : bits(64) = ThisInstrAddr{}();

    if pc[1:0] != '00' then
        AArch64_PCAlignmentFault();
    end;
end;
```

J1.2.2.4 AArch64_DataAbort

```
// AArch64_DataAbort()
// ===================

func AArch64_DataAbort(fault : FaultRecord)
begin
    var target_el : bits(2);
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        let route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                            (HCR_EL2().TGE == '1' ||
                            (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
                             HCR_EL2().GPF == '1') ||
                             (IsFeatureImplemented(FEAT_NV2) &&
                              fault.accessdesc.acctype == AccessType_NV2) ||
                             IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;
        end;
    end;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let route_to_serr : boolean = (IsExternalAbort(fault) &&
                                   AArch64_RouteToSErrorOffset(target_el));
    let vect_offset : integer = if route_to_serr then 0x180 else 0x0;

    var except : ExceptionRecord;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        except = AArch64_AbortSyndrome(Exception_NV2DataAbort, fault, target_el);
    else
        except = AArch64_AbortSyndrome(Exception_DataAbort, fault, target_el);
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.5 AArch64_EffectiveTCF

```
// AArch64_EffectiveTCF()
// ======================
// Indicate if a Tag Check Fault should cause a synchronous exception,
// be asynchronously accumulated, or have no effect on the PE.

func AArch64_EffectiveTCF(el : bits(2), read : boolean) => TCFType
begin
    var tcf : bits(2);

    let regime : Regime = TranslationRegime(el);

    case regime of
        when Regime_EL3 =>  tcf = SCTLR_EL3().TCF;
        when Regime_EL2 =>  tcf = SCTLR_EL2().TCF;
        when Regime_EL20 => tcf = if el == EL0 then SCTLR_EL2().TCF0 else SCTLR_EL2().TCF;
        when Regime_EL10 => tcf = if el == EL0 then SCTLR_EL1().TCF0 else SCTLR_EL1().TCF;
        otherwise =>        unreachable;
    end;

    if tcf == '11' then        // Reserved value
        if !IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
            (-,tcf) = ConstrainUnpredictableBits{2}(Unpredictable_RESTCF);
        end;
    end;

    case tcf of
        when '00' =>   // Tag Check Faults have no effect on the PE
            return TCFType_Ignore;
        when '01' =>   // Tag Check Faults cause a synchronous exception
            return TCFType_Sync;
        when '10' =>
            if IsFeatureImplemented(FEAT_MTE_ASYNC) then
                // If asynchronous faults are implemented,
                // Tag Check Faults are asynchronously accumulated
                return TCFType_Async;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType_Ignore;
            end;
        when '11' =>
            if IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
                // Tag Check Faults cause a synchronous exception on reads or on
                // a read/write access, and are asynchronously accumulated on writes
                if read then
                    return TCFType_Sync;
                else
                    return TCFType_Async;
                end;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType_Ignore;
            end;
        otherwise =>
            unreachable;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.6 AArch64_InstructionAbort

```
// AArch64_InstructionAbort()
// ==========================

func AArch64_InstructionAbort(fault : FaultRecord)
begin
    // External aborts on instruction fetch must be taken synchronously
    if IsFeatureImplemented(FEAT_DoubleFault) then
        assert fault.statuscode != Fault_AsyncExternal;
    end;

    var target_el : bits(2);
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        let route_to_el2 : boolean = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                                      (HCR_EL2().TGE == '1' ||
                                      (IsFeatureImplemented(FEAT_RME) &&
                                       fault.gpcf.gpf != GPCF_None &&
                                       HCR_EL2().GPF == '1') ||
                                       IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;
        end;
    end;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var vect_offset : integer;

    if IsExternalAbort(fault) && AArch64_RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    end;

    let except : ExceptionRecord = AArch64_AbortSyndrome(Exception_InstructionAbort, fault,
                                                         target_el);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.7 AArch64_PCAlignmentFault

```
// AArch64_PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

func AArch64_PCAlignmentFault()
begin

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_PCAlignment);
    except.vaddress = ThisInstrAddr{64}();
    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.8 AArch64_RaiseTagCheckFault

```
// AArch64_RaiseTagCheckFault()
// ============================
// Raise a Tag Check Fault exception.

func AArch64_RaiseTagCheckFault(fault : FaultRecord)
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;

    let except : ExceptionRecord = AArch64_AbortSyndrome(Exception_DataAbort, fault, target_el);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.9 AArch64_ReportTagCheckFault

```
// AArch64_ReportTagCheckFault()
// ============================
// Records a Tag Check Fault exception into the appropriate TFSR_ELx.

func AArch64_ReportTagCheckFault(el : bits(2), ttbr : bit)
begin
    case el of
        when EL3 => assert ttbr == '0'; TFSR_EL3().TF0   = '1';
        when EL2 => if ttbr == '0' then TFSR_EL2().TF0   = '1'; else TFSR_EL2().TF1   = '1'; end;
        when EL1 => if ttbr == '0' then TFSR_EL1().TF0   = '1'; else TFSR_EL1().TF1   = '1'; end;
        when EL0 => if ttbr == '0' then TFSRE0_EL1().TF0 = '1'; else TFSRE0_EL1().TF1 = '1'; end;
    end;
end;
```

J1.2.2.10 AArch64_RouteToSErrorOffset

```
// AArch64_RouteToSErrorOffset()
// ============================
// Returns TRUE if synchronous External abort exceptions are taken to the
// appropriate SError vector offset, and FALSE otherwise.

func AArch64_RouteToSErrorOffset(target_el : bits(2)) => boolean
begin
    if !IsFeatureImplemented(FEAT_DoubleFault) then return FALSE; end;

    var ease_bit : bit;
    case target_el of
        when EL3 =>
            ease_bit = SCR_EL3().EASE;
        when EL2 =>
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL2Enabled() then
                ease_bit = SCTLR2_EL2().EASE;
            else
                ease_bit = '0';
            end;
        when EL1 =>
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL1Enabled() then
                ease_bit = SCTLR2_EL1().EASE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            else
                ease_bit = '0';
            end;
    end;
    return (ease_bit == '1');
end;
```

J1.2.2.11 AArch64_SPAlignmentFault

```
// AArch64_SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

func AArch64_SPAlignmentFault()
begin

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_SPAlignment);

    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.12 BranchTargetException

```
// BranchTargetException()
// =======================
// Raise branch target exception.

func AArch64_BranchTargetException(vaddress : bits(52))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_BranchTarget);
    except.syndrome.iss[1:0]   = PSTATE.BTYPE;
    except.syndrome.iss[24:2]  = Zeros{23};         // RES0

    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.13 TCFType

```
// TCFType
// =======

type TCFType of enumeration { TCFType_Sync, TCFType_Async, TCFType_Ignore };
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.14 TakeGPCException

```
// TakeGPCException()
// ==================
// Report Granule Protection Exception faults

func TakeGPCException(fault : FaultRecord)
begin
    assert IsFeatureImplemented(FEAT_RME);
    assert IsFeatureImplemented(FEAT_LSE);
    assert IsFeatureImplemented(FEAT_HAFDBS);
    assert IsFeatureImplemented(FEAT_DoubleFault);

    var except : ExceptionRecord;

    except.exceptype = Exception_GPC;
    except.vaddress  = ZeroExtend{64}(fault.vaddress);
    except.paddress  = fault.paddress;
    except.pavalid   = TRUE;

    if IPAValid(fault) then
        except.ipavalid  = TRUE;
        except.NS        = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid  = FALSE;
    end;

    except.syndrome.iss2[11] = if fault.hdbssf then '1' else '0';   // HDBSSF
    if fault.accessdesc.acctype == AccessType_GCS then
        except.syndrome.iss2[8] = '1'; //GCS
    end;

    // Populate the fields grouped in ISS
    except.syndrome.iss[24:22] = Zeros{3}; // RES0
    except.syndrome.iss[21]    = if fault.gpcfs2walk then '1' else '0';  // S2PTW
    if fault.accessdesc.acctype == AccessType_IFETCH then
        except.syndrome.iss[20] = '1';     // InD
    else
        except.syndrome.iss[20] = '0';     // InD
    end;
    except.syndrome.iss[19:14] = EncodeGPCSC(fault.gpcf); // GPCSC
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
        except.syndrome.iss[13] = '1';     // VNCR
    else
        except.syndrome.iss[13] = '0';     // VNCR
    end;
    except.syndrome.iss[12:11]  = '00';     // RES0
    except.syndrome.iss[10:9]   = '00';     // RES0

    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        except.syndrome.iss[8] = '1'; // CM
    else
        except.syndrome.iss[8] = '0'; // CM
    end;

    except.syndrome.iss[7] = if fault.s2fs1walk then '1' else '0'; // S1PTW

    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        except.syndrome.iss[6] = '1';                              // WnR
    elsif fault.statuscode IN {Fault_UnsupportedAtomicHWUpdate, Fault_Exclusive} then
        except.syndrome.iss[6] = ARBITRARY : bit;                  // WnR
    elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
        except.syndrome.iss[6] = ARBITRARY : bit;                  // WnR
    else
        except.syndrome.iss[6] = if fault.write then '1' else '0'; // WnR
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    except.syndrome.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let target_el : bits(2) = EL3;

    var vect_offset : integer;
    if IsExternalAbort(fault) && AArch64_RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    end;

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.15 AArch64_TakeDelegatedSErrorException

```
// AArch64_TakeDelegatedSErrorException()
// ======================================

func AArch64_TakeDelegatedSErrorException()
begin
    assert IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && SCR_EL3().[EnDSE,DSE] == '11';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset = 0x180;
    var except : ExceptionRecord = ExceptionSyndrome(Exception_SError);

    var target_el : bits(2);
    var dsei_masked : boolean;
    (dsei_masked, target_el)  = AArch64_DelegatedSErrorTarget();
    assert !dsei_masked;
    except.syndrome.iss[24]   = VSESR_EL3().IDS;
    except.syndrome.iss[23:0] = VSESR_EL3().ISS;
    ClearPendingDelegatedSError();

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.16 AArch64_TakePhysicalFIQException

```
// AArch64_TakePhysicalFIQException()
// ==================================

func AArch64_TakePhysicalFIQException()
begin

    let route_to_el3 : boolean = EffectiveSCR_EL3_FIQ() == '1';
    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR_EL2().TGE == '1' || HCR_EL2().FMO == '1'));
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x100;
    let except : ExceptionRecord = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.17 AArch64_TakePhysicalIRQException

```
// AArch64_TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

func AArch64_TakePhysicalIRQException()
begin

    let route_to_el3 : boolean = EffectiveSCR_EL3_IRQ() == '1';
    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR_EL2().TGE == '1' || HCR_EL2().IMO == '1'));
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x80;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.18 AArch64_TakePhysicalSErrorException

```
// AArch64_TakePhysicalSErrorException()
// =====================================

func AArch64_TakePhysicalSErrorException(implicit_esb : boolean)
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer    = 0x180;
    let fault : FaultRecord      = GetPendingPhysicalSError();
    var except : ExceptionRecord = ExceptionSyndrome(Exception_SError);
    let is_esb : boolean         = FALSE;
    let syndrome : bits(25)      = AArch64_PhysicalSErrorSyndrome(is_esb, implicit_esb);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    end;

    if except.syndrome.iss[14] == '1' then
        except.pavalid  = TRUE;
        except.paddress = fault.paddress;
    else
        except.pavalid  = FALSE;
        except.paddress = ARBITRARY : FullAddress;
    end;
    except.syndrome.iss = syndrome;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.19 AArch64_TakeVirtualFIQException

```
// AArch64_TakeVirtualFIQException()
// ================================

func AArch64_TakeVirtualFIQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual IRQ enabled if TGE==0 and FMO==1
    assert HCR_EL2().TGE == '0' && HCR_EL2().FMO == '1';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x100;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_FIQ);

    AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.20 AArch64_TakeVirtualIRQException

```
// AArch64_TakeVirtualIRQException()
// ================================

func AArch64_TakeVirtualIRQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual IRQ enabled if TGE==0 and IMO==1
    assert HCR_EL2().TGE == '0' && HCR_EL2().IMO == '1';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x80;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_IRQ);

    AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.21 AArch64_TakeVirtualSErrorException

```
// AArch64_TakeVirtualSErrorException()
// ===================================

func AArch64_TakeVirtualSErrorException()
begin

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual SError enabled if TGE==0 and AMO==1 or TMEA==1
    assert (HCR_EL2().TGE == '0' &&
            (HCR_EL2().AMO == '1' || (IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer                 = 0x180;
    var except : ExceptionRecord              = ExceptionSyndrome(Exception_SError);

    if IsFeatureImplemented(FEAT_RAS) then
        except.syndrome.iss[24]    = VSESR_EL2().IDS;
        except.syndrome.iss[23:0] = VSESR_EL2().ISS;
    else
        let syndrome : bits(25)       = ImpDefBits{}("Virtual SError syndrome");
        let impdef_syndrome : boolean = syndrome[24] == '1';
        if impdef_syndrome then except.syndrome.iss = syndrome; end;
    end;

    ClearPendingVirtualSError();
    AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.2.22 AArch64_BreakpointException

```
// AArch64_BreakpointException()
// ============================

func AArch64_BreakpointException(fault : FaultRecord)
begin
    assert PSTATE.EL != EL3;

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var target_el : bits(2);
    let vect_offset : integer = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    let vaddress : bits(64)      = ARBITRARY : bits(64);
    let except : ExceptionRecord = AArch64_AbortSyndrome(Exception_Breakpoint, fault, target_el);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.23 AArch64_SoftwareBreakpoint

```
// AArch64_SoftwareBreakpoint()
// ============================

func AArch64_SoftwareBreakpoint(immediate : bits(16))
begin

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} &&
                                  EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    except.syndrome.iss[15:0]    = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.24 AArch64_SoftwareStepException

```
// AArch64_SoftwareStepException()
// ===============================

func AArch64_SoftwareStepException()
begin
    assert PSTATE.EL != EL3;

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var except : ExceptionRecord = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        except.syndrome.iss[24] = '0';
    else
        except.syndrome.iss[24] = '1';
        except.syndrome.iss[6] = if SoftwareStep_SteppedEX() then '1' else '0';
    end;
    except.syndrome.iss[5:0] = '100010';                // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.25 AArch64_VectorCatchException

```
// AArch64_VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

func AArch64_VectorCatchException(fault : FaultRecord)
begin
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1');

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let vaddress : bits(64)      = ARBITRARY : bits(64);
    let except : ExceptionRecord = AArch64_AbortSyndrome(Exception_VectorCatch, fault, EL2);

    AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.26 AArch64_WatchpointException

```
// AArch64_WatchpointException()
// =============================

func AArch64_WatchpointException(fault : FaultRecord)
begin
    assert PSTATE.EL != EL3;

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var target_el : bits(2);
    let vect_offset : integer = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    var except : ExceptionRecord;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
        except = AArch64_AbortSyndrome(Exception_NV2Watchpoint, fault, target_el);
    else
        except = AArch64_AbortSyndrome(Exception_Watchpoint, fault, target_el);
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.27 AArch64_ExceptionClass

```
// AArch64_ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

func AArch64_ExceptionClass(exceptype : Exception, target_el : bits(2)) => (integer,bit)
begin

    var il_is_valid : boolean = TRUE;
    let from_32 : boolean = UsingAArch32();
    var ec : integer;
    case exceptype of
        when Exception_Uncategorized         =>  ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap               =>  ec = 0x01;
        when Exception_CP15RTTrap            =>  ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap           =>  ec = 0x04; assert from_32;
        when Exception_CP14RTTrap            =>  ec = 0x05; assert from_32;
        when Exception_CP14DTTrap            =>  ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap   =>  ec = 0x07;
        when Exception_FPIDTrap              =>  ec = 0x08;
        when Exception_PACTrap               =>  ec = 0x09;
        when Exception_OtherTrap             =>  ec = 0x0A;
        when Exception_GPC                   =>  ec = 0x1E;
        when Exception_CP14RRTTrap           =>  ec = 0x0C; assert from_32;
        when Exception_BranchTarget          =>  ec = 0x0D;
        when Exception_IllegalState          =>  ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        =>  ec = if from_32 then 0x11 else 0x15;
        when Exception_HypervisorCall        =>  ec = if from_32 then 0x12 else 0x16;
        when Exception_MonitorCall           =>  ec = if from_32 then 0x13 else 0x17;
        when Exception_SystemRegisterTrap    =>  ec = 0x18; assert !from_32;
        when Exception_SystemRegister128Trap =>  ec = 0x14; assert !from_32;
        when Exception_SVEAccessTrap         =>  ec = 0x19; assert !from_32;
        when Exception_ERetTrap              =>  ec = 0x1A; assert !from_32;
        when Exception_PACFail               =>  ec = 0x1C; assert !from_32;
        when Exception_SMEAccessTrap         =>  ec = 0x1D; assert !from_32;
        when Exception_InstructionAbort      =>
            ec = if target_el == PSTATE.EL then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception_PCAlignment           =>  ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort             =>  ec = if target_el == PSTATE.EL then 0x25 else 0x24;
        when Exception_NV2DataAbort          =>  ec = 0x25;
        when Exception_SPAlignment           =>  ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_MemCpyMemSet          =>  ec = 0x27;
        when Exception_GCSFail               =>  ec = 0x2D; assert !from_32;
        when Exception_FPTrappedException    =>  ec = if from_32 then 0x28 else 0x2C;
        when Exception_SError                =>  ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint =>
            ec = if target_el == PSTATE.EL then 0x31 else 0x30;
            il_is_valid = FALSE;
        when Exception_SoftwareStep =>
            ec = if target_el == PSTATE.EL then 0x33 else 0x32;
            il_is_valid = FALSE;
        when Exception_Watchpoint =>
            ec = if target_el == PSTATE.EL then 0x35 else 0x34;
            il_is_valid = FALSE;
        when Exception_NV2Watchpoint         =>  ec = 0x35; il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint    =>  ec = if from_32 then 0x38 else 0x3C;
        when Exception_VectorCatch           =>  ec = 0x3A; il_is_valid = FALSE; assert from_32;
        when Exception_Profiling             =>  ec = 0x3D;
        otherwise                            =>  unreachable;
    end;

    var il : bit;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        il = '1';
    end;
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    return (ec,il);
end;
```

J1.2.2.28 AArch64_ReportException

```
// AArch64_ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

func AArch64_ReportException(except : ExceptionRecord, target_el : bits(2))
begin

    let exceptype : Exception = except.exceptype;

    var (ec,il) = AArch64_ExceptionClass(exceptype, target_el);
    let iss : bits(25)  = except.syndrome.iss;
    let iss2 : bits(24) = except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss[24] == '0' then
        il = '1';
    end;

    ESR_EL(target_el) = (Zeros{8}  ::   // [63:56]
                         iss2      ::   // [55:32]
                         ec[5:0]   ::   // [31:26]
                         il        ::   // [25]
                         iss);          // [24:0]

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_GPC,
        Exception_Watchpoint
    } then
        FAR_EL(target_el) = except.vaddress;
    else
        FAR_EL(target_el) = ARBITRARY : bits(64);
    end;

    if except.ipavalid then
        HPFAR_EL2()[NUM_PABITS-9:4] = except.ipaddress[NUM_PABITS-1:12];
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2().NS = except.NS;
        else
            HPFAR_EL2().NS = '0';
        end;
    elsif target_el == EL2 then
        HPFAR_EL2()[NUM_PABITS-9:4] = ARBITRARY : bits(NUM_PABITS-12);
    end;

    if except.pavalid then
        var faultaddr : bits(64) = ZeroExtend{}(except.paddress.address);
        if IsFeatureImplemented(FEAT_RME) then
            case except.paddress.paspace of
                when PAS_Secure =>     faultaddr[63:62] = '00';
                when PAS_NonSecure =>  faultaddr[63:62] = '10';
                when PAS_Root =>       faultaddr[63:62] = '01';
                when PAS_Realm =>      faultaddr[63:62] = '11';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
            if exceptype == Exception_GPC then
                faultaddr[11:0] = Zeros{12};
            end;
        else
            faultaddr[63] = if except.paddress.paspace == PAS_NonSecure then '1' else '0';
        end;
        PFAR_EL(target_el) = faultaddr;
    elsif (IsFeatureImplemented(FEAT_PFAR) ||
             (IsFeatureImplemented(FEAT_RME) && target_el == EL3)) then
        PFAR_EL(target_el) = ARBITRARY : bits(64);
    end;
    return;
end;
```

J1.2.2.29 AArch64_ResetControlRegisters

```
// AArch64_ResetControlRegisters()
// ==============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

impdef func AArch64_ResetControlRegisters(cold_reset : boolean)
begin
    return;
end;
```

J1.2.2.30 AArch64_TakeReset

```
// AArch64_TakeReset()
// ===================
// Reset into AArch64 state

func AArch64_TakeReset(cold_reset : boolean)
begin
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;
    end;

    // Reset System registers and other system components
    AArch64_ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';                    // Select stack pointer
    PSTATE.[D,A,I,F]  = '1111';         // All asynchronous exceptions masked
    PSTATE.SS = '0';                    // Clear software step bit
    PSTATE.DIT = '0';                   // PSTATE.DIT is reset to 0 when resetting into AArch64
    if IsFeatureImplemented(FEAT_PAuth_LR) then
        PSTATE.PACM = '0';              // PAC modifier
    end;
    if IsFeatureImplemented(FEAT_SME) then
        PSTATE.[SM,ZA] = '00';          // Disable Streaming SVE mode & ZA storage
        ResetSMEState('0');
    end;
    if IsFeatureImplemented(FEAT_SSBS) then
        PSTATE.SSBS = ImpDefBit("PSTATE.SSBS bit at reset");
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0';            // PSTATE.EXLOCK is reset to 0 when resetting into AArch64
    end;
    if IsFeatureImplemented(FEAT_UINJ) then
        PSTATE.UINJ = '0';              // PSTATE.UINJ is reset to 0 when resetting into AArch64
    end;
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64_ResetGeneralRegisters();
    if IsFeatureImplemented(FEAT_SME) || IsFeatureImplemented(FEAT_SVE) then
        ResetSVERegisters();
    else
        AArch64_ResetSIMDFPRegisters();
    end;
    AArch64_ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    var rv : bits(64);                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3();
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2();
    else
        rv = RVBAR_EL1();
    end;

    // The reset vector must be correctly aligned
    let pamax : AddressSize = AArch64_PAMax();
    assert IsZero(rv[63:pamax]) && IsZero(rv[1:0]);

    let branch_conditional : boolean = FALSE;
    EDPRSR().R = '0';                 // Leaving Reset State.
    BranchTo{64}(rv, BranchType_RESET, branch_conditional);
end;
```

J1.2.2.31 GetESR_ELx_DFSC

```
// GetESR_ELx_DFSC()
// =================
// Query the ESR_ELx to get the DFSC field.

func GetESR_ELx_DFSC(el : bits(2)) =>  bits(6)
begin
    let ec : bits(6) = ESR_EL(el).EC;
    assert ec IN {'10010x', '101111', '11010x'};

    return ESR_EL(el)[5:0];
end;
```

J1.2.2.32 GetESR_ELx_ExType

```
// GetESR_ELx_ExType()
// ===================
// Query the ESR_ELx to get the ExType field.

func GetESR_ELx_ExType(el : bits(2)) =>  bits(4)
begin
    let ec : bits(6) = ESR_EL(el).EC;
    assert ec == '101101';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return ESR_EL(el)[23:20];
end;
```

J1.2.2.33 GetESR_ELx_IFSC

```
// GetESR_ELx_IFSC()
// =================
// Query the ESR_ELx to get the IFSC field.

func GetESR_ELx_IFSC(el : bits(2)) =>  bits(6)
begin
    let ec : bits(6) = ESR_EL(el).EC;
    assert ec IN {'1100xx', '10000x'};

    return ESR_EL(el)[5:0];
end;
```

J1.2.2.34 AArch64_FPTrappedException

```
// AArch64_FPTrappedException()
// ============================

func AArch64_FPTrappedException(is_ase : boolean, accumulated_exceptions : bits(8))
begin
    var except : ExceptionRecord = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if ImpDefBool("vector instructions set TFV to 1") then
            except.syndrome.iss[23] = '1';                          // TFV
        else
            except.syndrome.iss[23] = '0';                          // TFV
        end;
    else
        except.syndrome.iss[23]     = '1';                          // TFV
    end;
    except.syndrome.iss[10:8] = ARBITRARY : bits(3);                // VECITR
    if except.syndrome.iss[23] == '1' then
        except.syndrome.iss[7,4:0] = accumulated_exceptions[7,4:0]; // IDF, IXF, UFF, OFF, DZF, IOF
    else
        except.syndrome.iss[7,4:0] = ARBITRARY : bits(6);
    end;

    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.35 AArch64_CallHypervisor

```
// AArch64_CallHypervisor()
// ========================
// Performs a HVC call

func AArch64_CallHypervisor(immediate : bits(16))
begin
    assert HaveEL(EL2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if UsingAArch32() then AArch32_ITAdvance(); end;
    SSAdvance();
    let preferred_exception_return : bits(64) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss[15:0] = immediate;

    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if PSTATE.EL == EL3 then
        AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.36 AArch64_CallSecureMonitor

```
// AArch64_CallSecureMonitor()
// ==========================

func AArch64_CallSecureMonitor(immediate : bits(16))
begin
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32_ITAdvance(); end;
    HSAdvance();
    SSAdvance();
    let preferred_exception_return : bits(64) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome.iss[15:0] = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.37 AArch64_CallSupervisor

```
// AArch64_CallSupervisor()
// ========================
// Calls the Supervisor

func AArch64_CallSupervisor(immediate : bits(16))
begin
    if UsingAArch32() then AArch32_ITAdvance(); end;
    SSAdvance();
    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';

    let preferred_exception_return : bits(64) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome.iss[15:0] = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.38 AArch64_TakeException

```
// AArch64_TakeException()
// =======================
// Take an exception to an Exception level using AArch64.

noreturn func AArch64_TakeException(target_el : bits(2), exception_in : ExceptionRecord,
                                    preferred_exception_return : bits(64), vect_offset_in : integer)
begin
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    if Halted() then
        AArch64_TakeExceptionInDebugState(target_el, exception_in);
    end;
    var except : ExceptionRecord = exception_in;
    var sync_errors : boolean;
    var iesb_req : boolean;
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_EL(target_el).IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3().[EA,NMEA] == '11' && target_el == EL3);
        end;
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
            if except.exceptype != Exception_SError then
                iesb_req = FALSE;
                sync_errors = FALSE;
                TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
            end;
        end;
    else
        sync_errors = FALSE;
    end;

    var brbe_source_allowed : boolean = FALSE;
    var brbe_source_address : bits(64) = Zeros{};
    if IsFeatureImplemented(FEAT_BRBE) then
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = preferred_exception_return;
    end;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_EL(target_el).EIS == '1' then
        // Synchronize the context, including Instruction Fetch Barrier effect
        SynchronizeContext();
    elsif !(except.exceptype == Exception_SoftwareBreakpoint ||
             (except.exceptype IN {Exception_SupervisorCall,
                                   Exception_HypervisorCall,
                                   Exception_MonitorCall} &&
              !except.trappedsyscallinst)) then
        InstructionFetchBarrier();
    end;

    // If coming from AArch32 state, the top parts of the X() registers might be set to zero
    let from_32 : boolean = UsingAArch32();
    if from_32 then AArch64_MaybeZeroRegisterUppers(); end;
    if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);
    end;

    var vect_offset : integer = vect_offset_in;
    if UInt(target_el) > UInt(PSTATE.EL) then
        var lower_32 : boolean;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            lower_32 = ELUsingAArch32(EL1);
        end;
    elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    end;
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;
end;

assert vect_offset < 2048;
let vect_base : bits(64) = VBAR_EL(target_el)[63:11]::Zeros{11};
let target_vector : bits(64) = vect_base + vect_offset;

var spsr : bits(64) = GetPSRFromPSTATE{}(AArch64_NonDebugState);

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if EffectiveHCR_EL2_NVx() IN {'x01', '111'} then
        spsr[3:2] = '10';
    end;
end;

if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
    var zero_btype : boolean;
    // SPSR_ELx[].BTYPE is only guaranteed valid for these exception types
    if except.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                            Exception_SoftwareStep, Exception_PCAlignment,
                            Exception_InstructionAbort, Exception_Breakpoint,
                            Exception_VectorCatch, Exception_SoftwareBreakpoint,
                            Exception_IllegalState, Exception_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    end;
    if zero_btype then spsr[11:10] = '00'; end;
end;

if (IsFeatureImplemented(FEAT_NV2) &&
      except.exceptype == Exception_NV2DataAbort && target_el == EL3) then
    // External aborts are configured to be taken to EL3
    except.exceptype = Exception_DataAbort;
end;
if ! except.exceptype IN {Exception_IRQ, Exception_FIQ} then
    AArch64_ReportException(except, target_el);
end;

if IsFeatureImplemented(FEAT_BRBE) then
    BRBEException(except, brbe_source_allowed, brbe_source_address,
                  target_vector, target_el, except.trappedsyscallinst);
end;

if IsFeatureImplemented(FEAT_GCS) then
    if PSTATE.EL == target_el then
        if GetCurrentEXLOCKEN() then
            PSTATE.EXLOCK = '1';
        else
            PSTATE.EXLOCK = '0';
        end;
    else
        PSTATE.EXLOCK = '0';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR_ELx() = spsr;
    ELR_ELx() = preferred_exception_return;
    PSTATE.SS = '0';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR_ELx().DSSBS; end;
    if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = '1'; end;
    if IsFeatureImplemented(FEAT_NMI) then
        PSTATE.ALLINT = NOT SCTLR_ELx().SPINTMASK;
    end;
    PSTATE.[D,A,I,F] = '1111';
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if IsFeatureImplemented(FEAT_PAN) && SCTLR_ELx().SPAN == '0' then
        if PSTATE.EL == EL1 then
            PSTATE.PAN = '1';
        elsif PSTATE.EL == EL2 && ELIsInHost(EL0) then
            PSTATE.PAN = '1';
        elsif (PSTATE.EL == EL2 && ELIsInHost(EL2) &&
                ImpDefBool("Set PAN as 1 when E2H,TGE is '10'")) then
            PSTATE.PAN = '1';
        end;
    end;
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    end;
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00'; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{64}(target_vector, BranchType_EXCEPTION, branch_conditional);
    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    end;

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.2.2.39 AArch64_AArch32SystemAccessTrap

```
// AArch64_AArch32SystemAccessTrap()
// ================================
// Trapped AArch32 System register access.

func AArch64_AArch32SystemAccessTrap(target_el : bits(2), ec : integer)
begin
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = AArch64_AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.40 AArch64_AArch32SystemAccessTrapSyndrome

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64_AArch32SystemAccessTrapSyndrome()
// =========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

func AArch64_AArch32SystemAccessTrapSyndrome(instr : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;

    case ec of
        when 0x0 =>    except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3 =>    except = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4 =>    except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5 =>    except = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6 =>    except = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7 =>    except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8 =>    except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC =>    except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise =>   unreachable;
    end;

    var iss : bits(20) = Zeros{};

    if except.exceptype == Exception_Uncategorized then
        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                               Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if except.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss[19:17] = instr[7:5];           // opc2
            iss[16:14] = instr[23:21];         // opc1
            iss[13:10] = instr[19:16];         // CRn
            iss[4:1]   = instr[3:0];           // CRm
        else
            iss[19:17] = '000';
            iss[16:14] = '111';
            iss[13:10] = instr[19:16];         // reg
            iss[4:1]   = '0000';
        end;

        if instr[20] == '1' && instr[15:12] == '1111' then    // MRC, Rt==15
            iss[9:5] = '11111';
        elsif instr[20] == '0' && instr[15:12] == '1111' then // MCR, Rt==15
            iss[9:5] = ARBITRARY : bits(5);
        else
            iss[9:5] = LookUpRIndex(UInt(instr[15:12]), PSTATE.M)[4:0];
        end;
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                               Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss[19:16] = instr[7:4];          // opc1
        if instr[19:16] == '1111' then    // Rt2==15
            iss[14:10] = ARBITRARY : bits(5);
        else
            iss[14:10] = LookUpRIndex(UInt(instr[19:16]), PSTATE.M)[4:0];
        end;

        if instr[15:12] == '1111' then    // Rt==15
            iss[9:5] = ARBITRARY : bits(5);
        else
            iss[9:5] = LookUpRIndex(UInt(instr[15:12]), PSTATE.M)[4:0];
        end;
        iss[4:1]   = instr[3:0];         // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        iss[19:12] = instr[7:0];         // imm8
        iss[4]     = instr[23];          // U
        iss[2:1]   = instr[24,21];       // P,W
        if instr[19:16] == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss[9:5] = ARBITRARY : bits(5);
            iss[3]   = '1';
        end;
    end;
    iss[0] = instr[20];                  // Direction

    except.syndrome.iss[24:20] = ConditionSyndrome();
    except.syndrome.iss[19:0]  = iss;

    return except;
end;
```

J1.2.2.41 AArch64_AdvSIMDFPAccessTrap

```
// AArch64_AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR.

func AArch64_AdvSIMDFPAccessTrap(target_el : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var except : ExceptionRecord;
    let vect_offset : integer = 0x0;

    let route_to_el2 : boolean = (target_el == EL1 && EL2Enabled() && HCR_EL2().TGE == '1');

    if route_to_el2 then
        except = ExceptionSyndrome(Exception_Uncategorized);
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        except.syndrome.iss[24:20] = ConditionSyndrome();
        AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;

    return;
end;
```

J1.2.2.42 AArch64_CheckCP15InstrCoarseTraps

```
// AArch64_CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.

func AArch64_CheckCP15InstrCoarseTraps(CRn : integer, nreg : integer, CRm : integer)
begin
    let trapped_encoding : boolean = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                                      (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                                      (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
    if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && !IsInHost() &&
           !ELUsingAArch32(EL1) && SCTLR_EL1().TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64_AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64_AArch32SystemAccessTrap(EL1, 0x3);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && IsInHost() &&
              SCTLR_EL2().TIDCP == '1' && trapped_encoding) then
            AArch64_AArch32SystemAccessTrap(EL2, 0x3);
        end;

        let major : integer = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2().CRn/HSTR().CRm
        // and MRC and MCR disabled by HCR_EL2.TIDCP.
        if ((!IsInHost() && ! major IN {4,14} && HSTR_EL2()[major] == '1') ||
                (HCR_EL2().TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    ImpDefBool("UNDEF unallocated CP15 access at EL0")) then
                Undefined();
            end;
            AArch64_AArch32SystemAccessTrap(EL2, 0x3);
        end;
    end;
end;
```

J1.2.2.43 AArch64_CheckFPAdvSIMDEnabled

```
// AArch64_CheckFPAdvSIMDEnabled()
// ==============================

func AArch64_CheckFPAdvSIMDEnabled()
begin
    AArch64_CheckFPEnabled();
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
    end;
end;
```

J1.2.2.44 AArch64_CheckFPAdvSIMDTrap

```
// AArch64_CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

func AArch64_CheckFPAdvSIMDTrap()
begin
    if HaveEL(EL3) && CPTR_EL3().TFP == '1' && EL3SDDUndefPriority() then
        Undefined();
    end;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if ELIsInHost(EL2) then
            var disabled : boolean;
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        else
            if CPTR_EL2().TFP == '1' then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        end;
    end;

    if HaveEL(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AdvSIMDFPAccessTrap(EL3);
            end;
        end;
    end;
end;
```

J1.2.2.45 AArch64_CheckFPEnabled

```
// AArch64_CheckFPEnabled()
// ========================
// Check against CPACR

func AArch64_CheckFPEnabled()
begin
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        var disabled : boolean;
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch64_AdvSIMDFPAccessTrap(EL1); end;
    end;

    AArch64_CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
end;
```

J1.2.2.46 AArch64_CheckForERetTrap

```
// AArch64_CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

func AArch64_CheckForERetTrap(eret_with_pac : boolean, pac_uses_key_a : boolean)
begin
    if PSTATE.EL == EL1 && EL2Enabled() then
        var route_to_el2 : boolean = FALSE;

        // Check for a fine-grained trap by the hypervisor.
        if (IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
              HFGITR_EL2().ERET == '1') then
            route_to_el2 = TRUE;
        // Check for a trap by the Effective value of the HCR_EL2.NV bit
        elsif EffectiveHCR_EL2_NVx()[0] == '1' then
            route_to_el2 = TRUE;
        end;

        if route_to_el2 then
            let preferred_exception_return : bits(64) = ThisInstrAddr{}();
            let vect_offset : integer = 0x0;
            var except : ExceptionRecord = ExceptionSyndrome(Exception_ERetTrap);

            if !eret_with_pac then                          // ERET
                except.syndrome.iss[1] = '0';
                except.syndrome.iss[0] = '0';               // RES0
            else
                except.syndrome.iss[1] = '1';
                except.syndrome.iss[0] = if pac_uses_key_a then '0' else '1';
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
        end;
    end;

    return;
end;
```

J1.2.2.47 AArch64_CheckForSMCUndefOrTrap

```
// AArch64_CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

func AArch64_CheckForSMCUndefOrTrap(imm : bits(16))
begin
    if PSTATE.EL == EL0 then Undefined(); end;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2().TSC == '1') &&
        HaveEL(EL3) && SCR_EL3().SMD == '1') then
        Undefined();
    end;
    var route_to_el2 : boolean = FALSE;
    if !HaveEL(EL3) then
        if (PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2().TSC == '1' &&
              (EffectiveHCR_EL2_NVx() == 'xx1' ||
              (ImpDefBool("Trap SMC execution at EL1 to EL2")))) then
            route_to_el2 = TRUE;
        else
            Undefined();
        end;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2().TSC == '1';
    end;
    if route_to_el2 then
        let preferred_exception_return : bits(64) = ThisInstrAddr{}();
        let vect_offset : integer = 0x0;
        var except : ExceptionRecord = ExceptionSyndrome(Exception_MonitorCall);
        except.syndrome.iss[15:0] = imm;
        except.trappedsyscallinst = TRUE;
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.48 AArch64_CheckForSVCTrap

```
// AArch64_CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

func AArch64_CheckForSVCTrap(immediate : bits(16))
begin
    if IsFeatureImplemented(FEAT_FGT) then
        var route_to_el2 : boolean = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                            EL2Enabled() && HFGITR_EL2().SVC_EL0 == '1' &&
                            (!IsInHost() && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1')));

        elsif PSTATE.EL == EL1 then
            route_to_el2 = (EL2Enabled() && HFGITR_EL2().SVC_EL1 == '1' &&
                            (!HaveEL(EL3) || SCR_EL3().FGTEn == '1'));
        end;

        if route_to_el2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            except.syndrome.iss[15:0] = immediate;
            except.trappedsyscallinst = TRUE;
            let preferred_exception_return : bits(64) = ThisInstrAddr{}();
            let vect_offset : integer = 0x0;

            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
        end;
    end;
end;
```

J1.2.2.49 AArch64_CheckForWFxTrap

```
// AArch64_CheckForWFxTrap()
// =========================
// Checks for a trap on a WFE, WFET, WFI or WFIT instruction.

func AArch64_CheckForWFxTrap(wfxtype : WFxType) => (boolean, bits(2))
begin
    let is_wfe : boolean = wfxtype IN {WFxType_WFE, WFxType_WFET};
    var target_el : bits(2);
    var trap : boolean = FALSE;

    if HaveEL(EL3) && EL3SDDUndefPriority() && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
        trap      = (if is_wfe then SCR_EL3().TWE else SCR_EL3().TWI) == '1';
        target_el = EL3;
    end;

    if !trap && PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        trap      = (if is_wfe then SCTLR_ELx().nTWE else SCTLR_ELx().nTWI) == '0';
        target_el = EL1;
    end;

    if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        trap      = (if is_wfe then HCR_EL2().TWE else HCR_EL2().TWI) == '1';
        target_el = EL2;
    end;

    if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        trap      = (if is_wfe then SCR_EL3().TWE else SCR_EL3().TWI) == '1';
        target_el = EL3;
    end;
    return (trap, target_el);
end;
```

J1.2.2.50 AArch64_CheckIllegalState

```
// AArch64_CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

func AArch64_CheckIllegalState()
begin
    if PSTATE.IL == '1' then
        let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';

        let preferred_exception_return : bits(64) = ThisInstrAddr{}();
        let vect_offset : integer = 0x0;

        let except : ExceptionRecord = ExceptionSyndrome(Exception_IllegalState);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
        else
            AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
        end;
    end;
end;
```

J1.2.2.51 AArch64_MonitorModeTrap

```
// AArch64_MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

func AArch64_MonitorModeTrap()
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer                 = 0x0;

    let except : ExceptionRecord              = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
    AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.52 AArch64_OtherInstrTrap

```
// AArch64_OtherInstrTrap()
// ========================
// Trapped access to the following instructions:
// - LD64B
// - ST64B
// - ST64BV
// - ST64BV0
// - PSB
// - TSB

func AArch64_OtherInstrTrap(target_el : bits(2), iss : bits(25))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_OtherTrap);
    except.syndrome.iss = iss;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.53 AArch64_SystemAccessTrap

```
// AArch64_SystemAccessTrap()
// ==========================
// Trapped access to AArch64 System register or system instruction.

func AArch64_SystemAccessTrap(target_el : bits(2), ec : integer)
begin
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let except : ExceptionRecord = AArch64_SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.2.54 AArch64_SystemAccessTrapSyndrome

```
// AArch64_SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

func AArch64_SystemAccessTrapSyndrome(instr_in : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;
    var instr : bits(32) = instr_in;
    case ec of
        when 0x0 =>                      // Trapped access due to unknown reason.
            except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7 =>                      // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            except.syndrome.iss[24:20] = ConditionSyndrome();
        when 0x14 =>                     // Trapped access to 128-bit System register or
                                         // 128-bit System instruction.
            except = ExceptionSyndrome(Exception_SystemRegister128Trap);
            instr = ThisInstr();
            except.syndrome.iss[21:20] = instr[20:19]; // Op0
            except.syndrome.iss[19:17] = instr[7:5];   // Op2
            except.syndrome.iss[16:14] = instr[18:16]; // Op1
            except.syndrome.iss[13:10] = instr[15:12]; // CRn
            except.syndrome.iss[9:6]   = instr[4:1];   // Rt
            except.syndrome.iss[4:1]   = instr[11:8];  // CRm
            except.syndrome.iss[0]     = instr[21];    // Direction
        when 0x18 =>                     // Trapped access to System register or system instruction.
            except = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome.iss[21:20] = instr[20:19]; // Op0
            except.syndrome.iss[19:17] = instr[7:5];   // Op2
            except.syndrome.iss[16:14] = instr[18:16]; // Op1
            except.syndrome.iss[13:10] = instr[15:12]; // CRn
            except.syndrome.iss[9:5]   = instr[4:0];   // Rt
            except.syndrome.iss[4:1]   = instr[11:8];  // CRm
            except.syndrome.iss[0]     = instr[21];    // Direction
        when 0x19 =>                         // Trapped access to SVE System register
            except = ExceptionSyndrome(Exception_SVEAccessTrap);
        when 0x1D =>                         // Trapped access to SME System register
            except = ExceptionSyndrome(Exception_SMEAccessTrap);
        otherwise =>
            unreachable;
    end;

    return except;
end;
```

J1.2.2.55 AArch64_Undefined

```
// AArch64_Undefined()
// ===================

noreturn func AArch64_Undefined()
begin

    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_Uncategorized);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.56 AArch64_WFxTrap

```
// AArch64_WFxTrap()
// =================
// Generate an exception for a trapped WFE, WFI, WFET or WFIT instruction.

func AArch64_WFxTrap(wfxtype : WFxType, target_el : bits(2))
begin
    assert UInt(target_el) > UInt(PSTATE.EL);
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    var except : ExceptionRecord = ExceptionSyndrome(Exception_WFxTrap);
    except.syndrome.iss[24:20] = ConditionSyndrome();
    case wfxtype of
        when WFxType_WFI =>
            except.syndrome.iss[1:0] = '00';
        when WFxType_WFE =>
            except.syndrome.iss[1:0] = '01';
        when WFxType_WFIT =>
            except.syndrome.iss[1:0] = '10';
            except.syndrome.iss[2]   = '1';   // Register field is valid
            except.syndrome.iss[9:5] = ThisInstr()[4:0];
        when WFxType_WFET =>
            except.syndrome.iss[1:0] = '11';
            except.syndrome.iss[2]   = '1';   // Register field is valid
            except.syndrome.iss[9:5] = ThisInstr()[4:0];
    end;

    if target_el == EL1 && EL2Enabled() && HCR_EL2().TGE == '1' then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.2.57 CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====================
// Checks for trap on ST64B and LD64B instructions

func CheckLDST64BEnabled()
begin
    var trap : boolean = FALSE;
    let iss : bits(25) = 0x2[24:0];
    var target_el : bits(2);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1().EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2().EnALS == '0';
            target_el = EL2;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        target_el = EL1;
    end;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2().EnALS == '0';
        target_el = EL2;
    end;

    if trap then AArch64_OtherInstrTrap(target_el, iss); end;
end;
```

J1.2.2.58 CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// ====================
// Checks for trap on ST64BV0 instruction

func CheckST64BV0Enabled()
begin
    var trap : boolean = FALSE;
    let iss : bits(25) = 0x1[24:0];
    var target_el : bits(2);

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3().EnAS0 == '0' && EL3SDDUndefPriority()) then
        Undefined();
    end;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1().EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2().EnAS0 == '0';
            target_el = EL2;
        end;
    end;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2().EnAS0 == '0';
        target_el = EL2;
    end;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3().EnAS0 == '0';
        target_el = EL3;
    end;

    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        else
            AArch64_OtherInstrTrap(target_el, iss);
        end;
    end;
end;
```

J1.2.2.59 CheckST64BVEnabled

```
// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func CheckST64BVEnabled()
begin
    var trap : boolean = FALSE;
    let iss : bits(25) = 0x0[24:0];
    var target_el : bits(2);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1().EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2().EnASR == '0';
            target_el = EL2;
        end;
    end;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2().EnASR == '0';
        target_el = EL2;
    end;

    if trap then AArch64_OtherInstrTrap(target_el, iss); end;
end;
```

J1.2.2.60 WFETrapDelay

```
// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

func WFETrapDelay(target_el : bits(2)) => (boolean, integer)
begin
    var delay_enabled : boolean;
    var delay : integer;
    case target_el of
        when EL1 =>
            if !IsInHost() then
                delay_enabled = SCTLR_EL1().TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL1().TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2().TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL2().TWEDEL) + 8);
            end;
        when EL2 =>
            assert EL2Enabled();
            delay_enabled = HCR_EL2().TWEDEn == '1';
            delay         = 1 << (UInt(HCR_EL2().TWEDEL) + 8);
        when EL3 =>
            delay_enabled = SCR_EL3().TWEDEn == '1';
            delay         = 1 << (UInt(SCR_EL3().TWEDEL) + 8);
    end;
    return (delay_enabled, delay);
end;
```

J1.2.2.61 WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// ========================
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires, including potentially
// some IMPLEMENTATION SPECIFIC additional delay, FALSE otherwise.

impdef func WaitForEventUntilDelay(delay_enabled : boolean, delay : integer) => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    return FALSE;
end;
```

J1.2.3 aarch64/functions

This section includes the following pseudocode functions:

- AArch64_FaultSyndrome
- AArch64_InstructionSyndromeValid
- EncodeGPCSC
- LS64InstructionSyndrome
- WatchpointFARNotPrecise
- AArch64_APAS
- AArch64_LocationSupportsAPAS
- APASRecord
- APAS_OP
- AArch64_AT
- AArch64_EncodePAR
- AArch64_PARFaultStatus
- AArch64_isPARFormatD128
- GetPAR_EL1_D128
- GetPAR_EL1_F
- MemBarrierOp
- BFXPreferred
- AltDecodeBitMasks
- DecodeBitMasks
- AArch64_DataMemZero
- AArch64_WriteTagMem
- CompareOp
- CountOp
- EffectiveCPTA
- EffectiveCPTM
- PointerAddCheck
- PointerAddCheckAtEL
- PointerCheckAtEL
- PointerMultiplyAddCheck
- IsD128Enabled
- AArch64_CanTrapDC
- AArch64_DC
- AArch64_MemZero
- AArch64_TreatDCAsNOP
- MemZero
- AArch64_ExceptionReturn
- AArch64_ExclusiveMonitorsPass
- AArch64_IsExclusiveVA
- AArch64_MarkExclusiveVA
- AArch64_SetExclusiveMonitors

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- DecodeRegExtend
- ExtendReg
- ExtendType
- FPConvOp
- FPMaxMinOp
- CheckFPMREnabled
- FPClampScale
- FPScale
- FPUnaryOp
- FPRSqrtStepFused
- FPRecipStepFused
- AddGCSExRecord
- AddGCSRecord
- CheckGCSExRecord
- CheckGCSSTREnabled
- EXLOCKException
- GCSDataCheckException
- GCSEnabled
- GCSInstruction
- GCSPCREnabled
- GCSPCRSelected
- GCSPOPCX
- GCSPOPM
- GCSPOPX
- GCSPUSHM
- GCSPUSHX
- GCSReturnValueCheckEnabled
- GCSSS1
- GCSSS2
- GCSSTRTrapException
- GCSSynchronizationBarrier
- GetCurrentEXLOCKEN
- GetCurrentGCSPointer
- LoadCheckGCSRecord
- SetCurrentGCSPointer
- hacdbs
- IsHACDBSIRQAsserted
- ProcessHACDBSEntry
- AArch64_CanTrapIC
- AArch64_EffectiveOpScope
- AArch64_IC
- AArch64_TreatICAsNOP
- ImmediateOp
- LogicalOp
- mec
- AArch64_S1AMECFault
- AArch64_S1DisabledOutputMECID

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64_S1OutputMECID
- AArch64_S1TTWalkMECID
- AArch64_S2OutputMECID
- AArch64_S2TTWalkMECID
- AArch64_AccessIsTagChecked
- AArch64_AddressWithAllocationTag
- AArch64_AllocationTagCheck
- AArch64_AllocationTagFromAddress
- AArch64_CanonicalTagCheck
- AArch64_CheckTag
- AArch64_IsUnprivAccessPriv
- AArch64_LogicalAddressTag
- AArch64_MemSingle
- AArch64_MemSingleRead
- AArch64_MemSingleWrite
- AArch64_MemTag
- AArch64_MemTagRead
- AArch64_MemTagWrite
- AArch64_UnalignedAccessFaults
- AddressSupportsLS64
- CASCompare
- CheckSPAlignment
- IsConventionalMemory
- Mem
- MemAtomic
- MemAtomicFP
- MemAtomicInt
- MemAtomicRCW
- MemLoad64B
- MemSingleGranule
- MemStore64B
- MemStore64BWithRet
- MemStore64BWithRetStatus
- NVMem
- PhysMemTagRead
- PhysMemTagWrite
- StoreOnlyTagCheckingEnabled
- ArchMaxMOPSBlockSize
- ArchMaxMOPSCPYSize
- ArchMaxMOPSSETGSize
- CPYFOptionA
- CPYOptionA
- CPYParams
- CPYPostSizeChoice
- CPYPreSizeChoice
- CPYSizeChoice
- CheckCPYConstrainedUnpredictable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckMOPSEnabled
- CheckMemCpyParams
- CheckMemSetParams
- CheckSETConstrainedUnpredictable
- IsMemCpyForward
- MOPSBlockSize
- MOPSStage
- MaxBlockSizeCopiedBytes
- MaxMOPSBlockSize
- MemCpyBytes
- MemCpyParametersIllformedE
- MemCpyParametersIllformedM
- MemCpyStageSize
- MemCpyZeroSizeCheck
- MemSetBytes
- MemSetParametersIllformedE
- MemSetParametersIllformedM
- MemSetStageSize
- MemSetTags
- MemSetZeroSizeCheck
- MemStageCpyZeroSizeCheck
- MemStageSetZeroSizeCheck
- MismatchedCpySetTargetEL
- MismatchedMemCpyException
- MismatchedMemSetException
- SETGOptionA
- SETOptionA
- SETParams
- SETPostSizeChoice
- SETPreSizeChoice
- SETSizeChoice
- UpdateCpyRegisters
- UpdateSetRegisters
- MoveWideOp
- MoveWidePreferred
- AddPAC
- AddPAC2
- InsertPAC
- AddPACDA
- AddPACDB
- AddPACGA
- AddPACIA
- AddPACIA2
- AddPACIB
- AddPACIB2
- AArch64_PACFailException
- Auth

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Auth2
- Authenticate
- AuthDA
- AuthDB
- AuthIA
- AuthIA2
- AuthIB
- AuthIB2
- AArch64_PACEffectiveTxSZ
- CalculateBottomPACBit
- ComputePAC
- ComputePAC2
- ComputePAC2IMPDEF
- ComputePAC2QARMA
- ComputePACIMPDEF
- ComputePACQARMA
- PACCellInvShuffle
- PACCellShuffle
- PACInvSub
- PACMult
- PACSub
- PacSub1
- RC
- TweakCellInvRot
- TweakCellRot
- TweakInvShuffle
- TweakShuffle
- IsAPDAKeyEnabled
- IsAPDBKeyEnabled
- IsAPIAKeyEnabled
- IsAPIBKeyEnabled
- IsPACMEnabled
- IsTrivialPACMImplementation
- PtrHasUpperAndLowerAddRanges
- Strip
- TrapPACUse
- AArch64_RestrictPrediction
- Prefetch
- PSTATEField
- AArch64_DelegatedSErrorTarget
- AArch64_ESBOperation
- AArch64_EncodeAsyncErrorSyndrome
- AArch64_EncodeSyncErrorSyndrome
- AArch64_PhysicalSErrorSyndrome
- AArch64_dESBOperation
- AArch64_vESBOperation
- EffectiveSCR_EL3_EnDSE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FirstRecordOfNode
- IsCommonFaultInjectionImplemented
- IsCountableErrorsRecorded
- IsErrorAddressIncluded
- IsErrorRecordImplemented
- IsFirstRecordOfNode
- IsPFARValid
- IsSPMUCounterImplemented
- rcw
- ProtectionEnabled
- RCWCheck
- FPReduce
- IntReduce
- ReduceOp
- AArch64_MaybeZeroRegisterUppers
- AArch64_ResetGeneralRegisters
- AArch64_ResetSIMDFPRegisters
- AArch64_ResetSpecialRegisters
- AArch64_ResetSystemRegisters
- PC64
- SP
- X
- DecodeShift
- ShiftReg
- ShiftType
- CounterToPredicate
- EncodePredCount
- Lookup
- PredCountTest
- System
- ZAhslice
- ZAslice
- ZAtile
- ZAvector
- ZAvslice
- ZT0
- AArch32_IsFPEnabled
- AArch64_IsFPEnabled
- ActivePredicateElement
- AllElementsActive
- AnyActiveElement
- BitDeposit
- BitExtract
- BitGroup
- CheckNonStreamingSVEEnabled
- CheckOriginalSVEEnabled
- CheckSMEAccess

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckSMEAndZAEnabled
- CheckSMEEnabled
- CheckSMEZT0Enabled
- CheckSVEEnabled
- CheckStreamingSVEAndZAEnabled
- CheckStreamingSVEEnabled
- CmpOp
- CurrentNSVL
- CurrentSVL
- CurrentVL
- DecodePredCount
- ElemFFR
- FFR
- FPCompareNE
- FPCompareUN
- FPConvertSVE
- FPExpA
- FPExpCoefficient
- FPLogB
- FPMinNormal
- FPOne
- FPPointFive
- FPReducePredicated
- FPTrigMAdd
- FPTrigMAddCoefficient
- FPTrigSMul
- FPTrigSSel
- FirstActive
- HaveSVE2FP8DOT2
- HaveSVE2FP8DOT4
- HaveSVE2FP8FMA
- ImplementedSMEVectorLength
- ImplementedSVEVectorLength
- InStreamingMode
- IntReducePredicated
- IsFPEnabled
- IsFullA64Enabled
- IsOriginalSVEEnabled
- IsSMEEnabled
- IsSVEEnabled
- LastActive
- LastActiveElement
- MaxImplementedAnyVL
- MaxImplementedSVL
- MaxImplementedVL
- MaybeZeroSVEUppers
- MemNF

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MemSingleNF
- NoneActive
- P
- PredLen
- PredTest
- PredicateElement
- ResetSMEState
- ResetSVERegisters
- ResetSVEState
- SMEAccessTrap
- SMEExceptionType
- SVCR
- SVEAccessTrap
- SVEMoveMaskPreferred
- SetPSTATE_SM
- SetPSTATE_ZA
- SupportedPowerTwoSVL
- System
- VecLen
- Z
- SystemHintOp
- SysLOp
- SystemLOp
- SysOp
- SystemOp
- SysOp128
- SystemOp128
- ELR_EL
- ELR_ELx
- ESR_EL
- ESR_ELx
- FAR_EL
- FAR_ELx
- PFAR_EL
- PFAR_ELx
- SCTLR_EL
- SCTLR_ELx
- VBAR_EL
- VBAR_ELx
- AArch64_CheckDAIFAccess
- AArch64_ChooseNonExcludedTag
- AArch64_ChooseNonExludedTagOrZero
- AArch64_ChooseTagOrZero
- AArch64_ExecutingERETInstr
- AArch64_ImpDefSysInstr
- AArch64_ImpDefSysInstr128
- AArch64_ImpDefSysInstrWithResult

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64_ImpDefSysRegRead
- AArch64_ImpDefSysRegRead128
- AArch64_ImpDefSysRegWrite
- AArch64_ImpDefSysRegWrite128
- AArch64_InterruptPending
- AArch64_NextRandomTagBit
- AArch64_RandomTag
- AArch64_SysInstr
- AArch64_SysInstrWithResult
- AArch64_SysRegRead
- AArch64_SysRegWrite
- BTypeCompatible
- BTypeCompatible_BTI
- BTypeCompatible_PAC
- BTypeCompatible_PACIXSP
- BTypeNext
- ChooseRandomNonExcludedTag
- EffectiveBADDR
- EffectiveICC_SRE_EL2_Enable
- EffectiveICC_SRE_EL3_Enable
- InGuardedPage
- IsHCRXEL2Enabled
- IsMTEEnabled
- IsPMTESelected
- IsSCTLR2EL1Enabled
- IsSCTLR2EL2Enabled
- IsSystemRegisterMaskingEnabled
- IsTCR2EL1Enabled
- IsTCR2EL2Enabled
- PACInstType
- SetBTypeCompatible
- SetBTypeNext
- SetInGuardedPage
- AArch64_SysInstr128
- AArch64_SysRegRead128
- AArch64_SysRegWrite128
- AArch64_EffectiveBroadcast
- AArch64_TLBIP_IPAS2
- AArch64_TLBIP_RIPAS2
- AArch64_TLBIP_RVA
- AArch64_TLBIP_RVAA
- AArch64_TLBIP_VA
- AArch64_TLBIP_VAA
- AArch64_TLBI_ALL
- AArch64_TLBI_ASID
- AArch64_TLBI_IPAS2
- AArch64_TLBI_PAALL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64_TLBI_RIPAS2
- AArch64_TLBI_RPA
- AArch64_TLBI_RVA
- AArch64_TLBI_RVAA
- AArch64_TLBI_VA
- AArch64_TLBI_VAA
- AArch64_TLBI_VMALL
- AArch64_TLBI_VMALLS12
- AArch64_TLBI_VMALLWS2
- DecodeTLBITG
- GPTTLBIMatch
- HasLargeAddress
- ResTLBIRTTL
- ResTLBITTL
- TGBits
- TLBIMatch
- TLBIPRange
- TLBIRange
- VBitOp

J1.2.3.1 AArch64_FaultSyndrome

```
// AArch64_FaultSyndrome()
// =======================
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to an Exception level using AArch64.

func AArch64_FaultSyndrome(exceptype : Exception, fault : FaultRecord,
                           target_el : bits(2)) => IssType
begin
    assert fault.statuscode != Fault_None;

    var isstype : IssType;
    isstype.iss  = Zeros{25};
    isstype.iss2 = Zeros{24};

    let d_side : boolean = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort,
                                         Exception_Watchpoint, Exception_NV2Watchpoint};
    if IsFeatureImplemented(FEAT_RAS) && fault.statuscode == Fault_SyncExternal then
        let errstate : ErrorState = PEErrorState(fault);
        isstype.iss[12:11] = AArch64_EncodeSyncErrorSyndrome(errstate);  // SET
    end;

    if d_side then
        if fault.accessdesc.acctype == AccessType_GCS then
            isstype.iss2[8] = '1';
        end;
        if exceptype IN {Exception_Watchpoint, Exception_NV2Watchpoint} then
            isstype.iss[23:0] = WatchpointRelatedSyndrome(fault);
        end;

        if AArch64_InstructionSyndromeValid(fault) then
            if fault.accessdesc.ls64 then
                (isstype.iss2, isstype.iss[24:14]) = LS64InstructionSyndrome();
            else
                isstype.iss[24:14] = LSInstructionSyndrome();
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
            isstype.iss[13] = '1';   // Fault is generated by use of VNCR_EL2
        end;

        if (IsFeatureImplemented(FEAT_LS64) &&
              fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            isstype.iss[12:11] = GetLoadStoreType();
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[8] = '1';
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[6] = '1';
        elsif fault.statuscode IN {Fault_UnsupportedAtomicHWUpdate, Fault_Exclusive} then
            isstype.iss[6] = ARBITRARY : bit;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss[6] = ARBITRARY : bit;
        else
            isstype.iss[6] = if fault.write then '1' else '0';
        end;
        isstype.iss2[9] = if fault.tagaccess then '1' else '0';
        if fault.statuscode == Fault_Permission then
            isstype.iss2[5] = if fault.dirtybit then '1' else '0';
            isstype.iss2[6] = if fault.overlay then '1' else '0';
            if isstype.iss[24] == '0' then
                isstype.iss[21] = if fault.toplevel then '1' else '0';
            end;
            isstype.iss2[7] = if fault.assuredonly then '1' else '0';
            isstype.iss2[10] = if fault.s1tagnotdata then '1' else '0';
        end;

    else
        if (fault.accessdesc.acctype == AccessType_IFETCH &&
              fault.statuscode == Fault_Permission) then
            isstype.iss2[5] = if fault.dirtybit then '1' else '0';
            isstype.iss[21] = if fault.toplevel then '1' else '0';
            isstype.iss2[7] = if fault.assuredonly then '1' else '0';
            isstype.iss2[6] = if fault.overlay then '1' else '0';
        end;
    end;
    if fault.hdbssf then
        isstype.iss2[11] = '1';
    end;

    if (IsFeatureImplemented(FEAT_PFAR) && IsExternalSyncAbort(fault) &&
          !(EL2Enabled() && (HCR_EL2().VM == '1' || HCR_EL2().DC == '1') && target_el == EL1)) then
        isstype.iss[14] = if IsPFARValid(fault) then '1' else '0';
    end;

    if IsExternalAbort(fault) then isstype.iss[9] = fault.extflag; end;
    isstype.iss[7] = if fault.s2fs1walk then '1' else '0';
    isstype.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype;
end;
```

J1.2.3.2 AArch64_InstructionSyndromeValid

```
// AArch64_InstructionSyndromeValid()
// ==================================
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

func AArch64_InstructionSyndromeValid(fault : FaultRecord) => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !fault.accessdesc.lowestaddress then
        return FALSE;
    end;

    if fault.accessdesc.ls64 then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission};
    end;

    if fault.accessdesc.acctype == AccessType_GCS then
        return FALSE;
    end;

    if IsSecondStage(fault) && !fault.s2fs1walk then
        return (!IsExternalSyncAbort(fault) ||
                (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                  ImpDefBool("ISV on second stage translation table walk")));
    end;

    return FALSE;
end;
```

J1.2.3.3 EncodeGPCSC

```
// EncodeGPCSC()
// =============
// Function that gives the GPCSC code for types of GPT Fault

func EncodeGPCSC(gpcf : GPCFRecord) => bits(6)
begin
    assert gpcf.level IN {0,1};

    case gpcf.gpf of
        when GPCF_AddressSize => return '00000'::gpcf.level[0];
        when GPCF_Walk =>        return '00010'::gpcf.level[0];
        when GPCF_Fail =>        return '00110'::gpcf.level[0];
        when GPCF_EABT =>        return '01010'::gpcf.level[0];
    end;
end;
```

J1.2.3.4 LS64InstructionSyndrome

```
// LS64InstructionSyndrome()
// =========================
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.

impdef func LS64InstructionSyndrome() => (bits(24), bits(11))
begin
    return (Zeros{24}, LSInstructionSyndrome());
end;
```

J1.2.3.5 WatchpointFARNotPrecise

```
// WatchpointFARNotPrecise()
// =========================
// Returns TRUE If the lowest watchpointed address that is higher than or equal to the address
// recorded in EDWAR might not have been accessed by the instruction, other than the CONSTRAINED
// UNPREDICTABLE condition of watchpoint matching a range of addresses with lowest address 16 bytes
// rounded down and upper address rounded up to nearest 16 byte multiple,
// FALSE otherwise.

impdef func WatchpointFARNotPrecise(fault : FaultRecord) => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_SVE) || IsFeatureImplemented(FEAT_SME) then
        return fault.watchptinfo.maybe_false_match;
    end;
    return FALSE;
end;
```

J1.2.3.6 AArch64_APAS

```
// AArch64_APAS()
// ==============
// Decode Xt and perform an APAS operation for the decoded record.

func AArch64_APAS(Xt : bits(64))
begin
    var apas : APASRecord;
    let nse2 : bit = '0';
    apas.paspace = DecodePASpace(nse2, Xt[62], Xt[63]);

    apas.pa = Xt[NUM_PABITS-1:6] :: '000000';
    apas.target_attributes = Xt[2:0];

    if AArch64_LocationSupportsAPAS(apas) then
        APAS_OP(apas);
    end;
end;
```

J1.2.3.7 AArch64_LocationSupportsAPAS

```
// AArch64_LocationSupportsAPAS()
// ==============================
// Returns TRUE if the given memory location supports the APAS instruction.

impdef func AArch64_LocationSupportsAPAS(apas : APASRecord) => boolean
begin
    return TRUE;
end;
```

J1.2.3.8 APASRecord

```
// APASRecord
// ==========
// Details related to an APAS operation.

type APASRecord of record {
    pa : bits(NUM_PABITS),
    paspace : PASpace,
    target_attributes : bits(3)
};
```

J1.2.3.9 APAS_OP

```
// APAS_OP()
// =========
// Sets the PA Space of the address in the APASRecord to the target PA space. If the location
// does not support the APAS instruction or cannot be associated with the indicated PASpace,
// then the instruction has no effect on the location and does not generate an External abort.

impdef func APAS_OP(apas : APASRecord)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.10 AArch64_AT

```
// AArch64_AT()
// ============
// Perform address translation as per AT instructions.

func AArch64_AT(address : bits(64), stage : TranslationStage, el_in : bits(2), ataccess : ATAccess)
begin
    var el : bits(2) = el_in;
    // For stage 1 translation, when HCR_EL2().[E2H, TGE] is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if ELIsInHost(EL0) && el == EL1 && stage == TranslationStage_1 then
        el = EL2;
    end;

    let ss : SecurityState = SecurityStateAtEL(el);

    let accdesc : AccessDescriptor = CreateAccDescAT(ss, el, ataccess);
    let aligned : boolean = TRUE;

    var fault : FaultRecord = NoFault(accdesc, address);
    let regime : Regime = TranslationRegime(el);

    var addrdesc : AddressDescriptor;
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime_EL2 || TTBCR().EAE == '1' then
            (fault, addrdesc) = AArch32_S1TranslateLD(fault, regime, address[31:0], aligned,
                                                      accdesc);
        else
            (fault, addrdesc, -) = AArch32_S1TranslateSD(fault, regime, address[31:0], aligned,
                                                         accdesc);
        end;
    else
        let size : integer = 1;
        (fault, addrdesc) = AArch64_S1Translate(fault, regime, address, size, aligned, accdesc);
    end;

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        let s1aarch64 : boolean = TRUE;
        if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend{64}(address);
            (fault, addrdesc) = AArch32_S2Translate(fault, addrdesc, aligned, accdesc);
        elsif regime == Regime_EL10 && EL2Enabled() then
            (fault, addrdesc) = AArch64_S2Translate(fault, addrdesc, s1aarch64, aligned, accdesc);
        end;
    end;

    let is_ATS1Ex : boolean = stage != TranslationStage_12;
    if fault.statuscode != Fault_None then
        addrdesc = CreateFaultyAddressDescriptor(fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        // * A GPC Exception (FEAT_RME)
        // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
        if (IsExternalAbort(fault) ||
              (PSTATE.EL == EL1 && fault.s2fs1walk) ||
               (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None && (
                ReportAsGPCException(fault) ||
                (EL2Enabled() && HCR_EL2().GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} &&
                 is_ATS1Ex)
                ))) then
            if IsFeatureImplemented(FEAT_D128) then
                PAR_EL1()       = ARBITRARY : bits(128);
            else
                PAR_EL1()[63:0] = ARBITRARY : bits(64);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64_Abort(addrdesc.fault);
        end;
    end;

    AArch64_EncodePAR(regime, el, is_ATS1Ex, addrdesc);
    return;
end;
```

J1.2.3.11 AArch64_EncodePAR

```
// AArch64_EncodePAR()
// ===================
// Encode PAR register with result of translation.

func AArch64_EncodePAR(regime : Regime, el : bits(2), is_ATS1Ex : boolean,
                       addrdesc : AddressDescriptor)
begin
    let paspace : PASpace = addrdesc.paddress.paspace;
    if IsFeatureImplemented(FEAT_D128) then
        PAR_EL1() = Zeros{128};
        if AArch64_isPARFormatD128(regime, is_ATS1Ex) then
            PAR_EL1().D128 = '1';
        else
            PAR_EL1().D128 = '0';
        end;
    else
        PAR_EL1()[63:0] = Zeros{64};
    end;

    if !IsFault(addrdesc) then
        PAR_EL1().F = '0';
        if IsFeatureImplemented(FEAT_RME) then
            if regime == Regime_EL3 then
                case paspace of
                    when PAS_Secure =>     PAR_EL1().[NSE,NS] = '00';
                    when PAS_NonSecure =>  PAR_EL1().[NSE,NS] = '01';
                    when PAS_Root =>       PAR_EL1().[NSE,NS] = '10';
                    when PAS_Realm =>      PAR_EL1().[NSE,NS] = '11';
                end;

            elsif SecurityStateForRegime(regime) == SS_Secure  then
                PAR_EL1().NSE = ARBITRARY : bit;
                PAR_EL1().NS = if paspace == PAS_Secure then '0' else '1';

            elsif SecurityStateForRegime(regime) == SS_Realm then
                if regime == Regime_EL10 && is_ATS1Ex then
                    PAR_EL1().NSE = ARBITRARY : bit;
                    PAR_EL1().NS  = ARBITRARY : bit;
                else
                    PAR_EL1().NSE = ARBITRARY : bit;
                    PAR_EL1().NS  = if paspace == PAS_Realm then '0' else '1';
                end;

            else
                PAR_EL1().NSE = ARBITRARY : bit;
                PAR_EL1().NS  = ARBITRARY : bit;
            end;
        else
            PAR_EL1()[11] = '1'; // RES1
            if SecurityStateForRegime(regime) == SS_Secure then
                PAR_EL1().NS = if paspace == PAS_Secure then '0' else '1';
            else
                PAR_EL1().NS = ARBITRARY : bit;
            end;
        end;
        PAR_EL1().SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_D128) && PAR_EL1().D128 == '1' then
            PAR_EL1()[119:76] = addrdesc.paddress.address[55:12];
        else
            PAR_EL1()[55:12] = addrdesc.paddress.address[55:12];
        end;
        PAR_EL1().ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1()[10] = ImpDefBit("Non-Faulting PAR");
    else
        PAR_EL1().F   = '1';
        PAR_EL1().DirtyBit    = if addrdesc.fault.dirtybit then '1' else '0';
        PAR_EL1().Overlay     = if addrdesc.fault.overlay then '1' else '0';
        PAR_EL1().TopLevel    = if addrdesc.fault.toplevel then '1' else '0';
        PAR_EL1().AssuredOnly = if addrdesc.fault.assuredonly then '1' else '0';
        PAR_EL1().FST = AArch64_PARFaultStatus(addrdesc.fault);
        PAR_EL1().PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR_EL1().S   = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1()[11] = '1'; // RES1
        PAR_EL1()[63:48] = ImpDefBits{16}("Faulting PAR");
    end;
    return;
end;
```

J1.2.3.12 AArch64_PARFaultStatus

```
// AArch64_PARFaultStatus()
// ========================
// Fault status field decoding of 64-bit PAR.

func AArch64_PARFaultStatus(fault : FaultRecord) => bits(6)
begin
    var fst : bits(6);

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst[1:0] = if fault.level == 1 then '01' else '10';
        fst[5:2] = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    end;
    return fst;
end;
```

J1.2.3.13 AArch64_isPARFormatD128

```
// AArch64_isPARFormatD128()
// =========================
// Check if last stage of translation uses VMSAv9-128.
// Last stage of translation is stage 2 if enabled, else it is stage 1.

func AArch64_isPARFormatD128(regime : Regime, is_ATS1Ex : boolean) => boolean
begin
    var isPARFormatD128 : boolean;
    // Regime_EL2 does not support VMSAv9-128
    if regime == Regime_EL2 || !IsFeatureImplemented(FEAT_D128) then
        isPARFormatD128 = FALSE;
    else
        isPARFormatD128 = FALSE;
        case regime of
            when Regime_EL3 =>
                isPARFormatD128 = TCR_EL3().D128 == '1';
            when Regime_EL20 =>
                isPARFormatD128 = IsTCR2EL2Enabled() && TCR2_EL2().D128 == '1';
            when Regime_EL10 =>
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2().[VM,DC] == '00' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    isPARFormatD128 = IsTCR2EL1Enabled() && TCR2_EL1().D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2().D128 == '1';
                end;
        end;
    end;

    return isPARFormatD128;
end;
```

J1.2.3.14 GetPAR_EL1_D128

```
// GetPAR_EL1_D128()
// =================
// Query the PAR_EL1.D128 field

func GetPAR_EL1_D128() => bit
begin
    return if IsFeatureImplemented(FEAT_D128) then PAR_EL1().D128 else '0';
end;
```

J1.2.3.15 GetPAR_EL1_F

```
// GetPAR_EL1_F()
// ==============
// Query the PAR_EL1.F field.

func GetPAR_EL1_F() => bit
begin
    var F : bit;

    F = PAR_EL1().F;
    return F;
end;
```

J1.2.3.16 MemBarrierOp

```
// MemBarrierOp
// ============
// Memory barrier instruction types.

type MemBarrierOp of enumeration {
                                MemBarrierOp_DSB,     // Data Synchronization Barrier
                                MemBarrierOp_DMB,     // Data Memory Barrier
                                MemBarrierOp_ISB,     // Instruction Synchronization Barrier
                                MemBarrierOp_SSBB,    // Speculative Synchronization Barrier to VA
                                MemBarrierOp_PSSBB,   // Speculative Synchronization Barrier to PA
                                MemBarrierOp_SB       // Speculation Barrier
                            };
```

J1.2.3.17 BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

func BFXPreferred(sf : bit, uns : bit, imms : bits(6), immr : bits(6)) => boolean
begin

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return FALSE;
    end;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf::'11111' then
        return FALSE;
    end;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        end;
        // must not match 64-bit SXT[BHW]
        if sf::uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;
        end;
    end;

    // must be UBFX/SBFX alias
    return TRUE;
end;
```

J1.2.3.18 AltDecodeBitMasks

```
// AltDecodeBitMasks()
// ===================
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

func AltDecodeBitMasks{M}(immN : bit, imms : bits(6), immr : bits(6),
                          immediate : boolean) => (bits(M), bits(M))
begin
    var tmask, wmask : bits(64);
    var tmask_and, wmask_and : bits(6);
    var tmask_or, wmask_or : bits(6);
    var levels : bits(6);

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    if immN::NOT(imms) == '000000x' then Undefined(); end;
    let len : integer{} = HighestSetBitNZ(immN::NOT(imms));
    assert 2 <= (2^len) && (2^len) <= M;

    // Determine s, r and s - r parameters
    levels = ZeroExtend{6}(Ones{len});

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then Undefined(); end;

    let s : integer{} = UInt(imms AND levels);
    let r : integer{} = UInt(immr AND levels);
    let diff : integer{} = s - r;    // 6-bit subtract with borrow

    // Compute "top mask"
    tmask_and = diff[5:0] OR NOT(levels);
    tmask_or  = diff[5:0] AND levels;

    tmask = Ones{64};
    tmask = ((tmask
              AND Replicate{64, 2}(Replicate{1}(tmask_and[0]) :: Ones{1}))
               OR  Replicate{64, 2}(Zeros{1} :: Replicate{1}(tmask_or[0])));
    // optimization of first step:
    // tmask = Replicate{64}(tmask_and[0] :: '1');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    tmask = ((tmask
              AND Replicate{64, 4}(Replicate{2}(tmask_and[1]) :: Ones{2}))
               OR  Replicate{64, 4}(Zeros{2} :: Replicate{2}(tmask_or[1])));
    tmask = ((tmask
              AND Replicate{64, 8}(Replicate{4}(tmask_and[2]) :: Ones{4}))
               OR  Replicate{64, 8}(Zeros{4} :: Replicate{4}(tmask_or[2])));
    tmask = ((tmask
              AND Replicate{64, 16}(Replicate{8}(tmask_and[3]) :: Ones{8}))
               OR  Replicate{64, 16}(Zeros{8} :: Replicate{8}(tmask_or[3])));
    tmask = ((tmask
              AND Replicate{64, 32}(Replicate{16}(tmask_and[4]) :: Ones{16}))
               OR  Replicate{64, 32}(Zeros{16} :: Replicate{16}(tmask_or[4])));
    tmask = ((tmask
              AND Replicate{64, 64}(Replicate{32}(tmask_and[5]) :: Ones{32}))
               OR  Replicate{64, 64}(Zeros{32} :: Replicate{32}(tmask_or[5])));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros{64};
    wmask = ((wmask
              AND Replicate{64, 2}(Ones{1} :: Replicate{1}(wmask_and[0])))
               OR  Replicate{64, 2}(Replicate{1}(wmask_or[0]) :: Zeros{1}));
    // wmask = Replicate{64}(wmask_or[0] :: '0');
    wmask = ((wmask
              AND Replicate{64, 4}(Ones{2} :: Replicate{2}(wmask_and[1])))
               OR  Replicate{64, 4}(Replicate{2}(wmask_or[1]) :: Zeros{2}));
    wmask = ((wmask
              AND Replicate{64, 8}(Ones{4} :: Replicate{4}(wmask_and[2])))
               OR  Replicate{64, 8}(Replicate{4}(wmask_or[2]) :: Zeros{4}));
    wmask = ((wmask
              AND Replicate{64, 16}(Ones{8} :: Replicate{8}(wmask_and[3])))
               OR  Replicate{64, 16}(Replicate{8}(wmask_or[3]) :: Zeros{8}));
    wmask = ((wmask
              AND Replicate{64, 32}(Ones{16} :: Replicate{16}(wmask_and[4])))
               OR  Replicate{64, 32}(Replicate{16}(wmask_or[4]) :: Zeros{16}));
    wmask = ((wmask
              AND Replicate{64, 64}(Ones{32} :: Replicate{32}(wmask_and[5])))
               OR  Replicate{64, 64}(Replicate{32}(wmask_or[5]) :: Zeros{32}));

    if diff[6] != '0' then // borrow from s - r
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;
    end;

    return (wmask[M-1:0], tmask[M-1:0]);
end;
```

J1.2.3.19 DecodeBitMasks

```
// DecodeBitMasks()
// ================
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

func DecodeBitMasks{M}(immN : bit, imms : bits(6), immr : bits(6),
                       immediate : boolean) => (bits(M), bits(M))
begin
    var levels : bits(6);

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    if immN::NOT(imms) == '000000x' then Undefined(); end;
    let len : integer{} = HighestSetBitNZ(immN::NOT(imms));
    assert 2 <= (2^len) && (2^len) <= M;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Determine s, r and s - r parameters
    levels = ZeroExtend{6}(Ones{len});

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then Undefined(); end;

    let s : integer{} = UInt(imms AND levels);
    let r : integer{} = UInt(immr AND levels);
    let diff : integer{} = s - r;    // 6-bit subtract with borrow

    let esize : integer{} = 1 << len;
    let d : integer{}     = UInt(diff[len-1:0]);
    let welem : bits(esize) = ZeroExtend{}(Ones{s + 1});
    let telem : bits(esize) = ZeroExtend{}(Ones{d + 1});
    let wmask : bits(M) = Replicate{}(ROR(welem, r));
    let tmask : bits(M) = Replicate{}(telem);
    return (wmask, tmask);
end;
```

J1.2.3.20 AArch64_DataMemZero

```
// AArch64_DataMemZero()
// =====================
// Write Zero to data memory.

func AArch64_DataMemZero(regval : bits(64), vaddress : bits(64),
                         accdesc_in : AccessDescriptor, size : integer)
begin
    var accdesc : AccessDescriptor = accdesc_in;

    // If the instruction targets tags as a payload, confer with system register configuration
    // which may override this.
    if accdesc.tagaccess then
        accdesc.tagaccess = IsMTEEnabled(accdesc.el);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(vaddress, accdesc);
    end;

    let aligned : boolean = TRUE;
    var memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(vaddress,
                                                                   accdesc, aligned, size);

    if IsFault(memaddrdesc) then
        if !IsDebugException(memaddrdesc.fault) then
            memaddrdesc.fault.vaddress = regval;
        end;
        AArch64_Abort(memaddrdesc.fault);
    end;

    for i = 0 to size-1 do
        if accdesc.tagchecked then
            let ltag : bits(4)      = AArch64_LogicalAddressTag(vaddress);
            let bytes : integer = 1;
            let readcheck : boolean = FALSE;
            var fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck,
                                                       bytes, ltag);
            if fault.statuscode != Fault_None then
                if ImpDefBool(
                      "DC_ZVA tag fault reported with lowest faulting address") then
                    AArch64_Abort(fault);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                else
                    fault.vaddress = regval;
                    AArch64_Abort(fault);
                end;
            end;
        end;
        let memstatus : PhysMemRetStatus = PhysMemWrite{8}(memaddrdesc, accdesc, Zeros{8});
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    end;
    return;
end;
```

J1.2.3.21 AArch64_WriteTagMem

```
// AArch64_WriteTagMem()
// =====================
// Write to tag memory.

func AArch64_WriteTagMem(regval : bits(64), cachetype : CacheType,
                         accdesc_in : AccessDescriptor, size : integer)
begin
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    var accdesc : AccessDescriptor = accdesc_in;
    let vaddress : bits(64) = AlignDownSize{64}(regval, size as integer{16..2048});
    let count : integer     = size >> LOG2_TAG_GRANULE;
    let tag : bits(4) = AArch64_AllocationTagFromAddress(vaddress);
    let aligned : boolean   = IsAlignedSize{64}(vaddress, TAG_GRANULE);
    assert aligned;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    var (memtagtype, memaddrdesc) = AArch64_TranslateTagAddress(vaddress, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64_Abort(memaddrdesc.fault);
        else
            memaddrdesc.fault.vaddress = regval;
            AArch64_Abort(memaddrdesc.fault);
        end;
    end;

    if !accdesc.tagaccess || memtagtype != MemTag_AllocationTagged then
        return;
    end;

    var memstatus : PhysMemRetStatus;

    for i = 0 to count-1 do
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG_GRANULE;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + TAG_GRANULE;
    end;

    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.22 CompareOp

```
// CompareOp
// =========
// Vector compare instruction types.

type CompareOp of enumeration {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                               CompareOp_LE, CompareOp_LT};
```

J1.2.3.23 CountOp

```
// CountOp
// =======
// Bit counting instruction types.

type CountOp of enumeration {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

J1.2.3.24 EffectiveCPTA

```
// EffectiveCPTA()
// ===============
// Returns the CPTA bit applied to Checked Pointer Arithmetic for Addition in the given EL.

func EffectiveCPTA(el : bits(2)) => bit
begin
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';
    end;

    if Halted() then
        return '0';
    end;

    var cpta : bits(1);
    let regime : Regime = TranslationRegime(el);

    case regime of
        when Regime_EL3 =>
            cpta = SCTLR2_EL3().CPTA;
        when Regime_EL2 =>
            if IsSCTLR2EL2Enabled() then
                cpta = SCTLR2_EL2().CPTA;
            else
                cpta = '0';
            end;
        when Regime_EL20 =>
            if IsSCTLR2EL2Enabled() then
                cpta = if el == EL0 then SCTLR2_EL2().CPTA0 else SCTLR2_EL2().CPTA;
            else
                cpta = '0';
            end;
        when Regime_EL10 =>
            if IsSCTLR2EL1Enabled() then
                cpta = if el == EL0 then SCTLR2_EL1().CPTA0 else SCTLR2_EL1().CPTA;
            else
                cpta = '0';
            end;
        otherwise =>
            unreachable;
    end;

    return cpta;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.25 EffectiveCPTM

```
// EffectiveCPTM()
// ===============
// Returns the CPTM bit applied to Checked Pointer Arithmetic for Multiplication in the given EL.

func EffectiveCPTM(el : bits(2)) => bit
begin
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';
    end;

    if EffectiveCPTA(el) == '0' then
        return '0';
    end;

    if Halted() then
        return '0';
    end;

    var cptm : bits(1);
    let regime : Regime = TranslationRegime(el);

    case regime of
        when Regime_EL3 =>
            cptm = SCTLR2_EL3().CPTM;
        when Regime_EL2 =>
            if IsSCTLR2EL2Enabled() then
                cptm = SCTLR2_EL2().CPTM;
            else
                cptm = '0';
            end;
        when Regime_EL20 =>
            if IsSCTLR2EL2Enabled() then
                cptm = if el == EL0 then SCTLR2_EL2().CPTM0 else SCTLR2_EL2().CPTM;
            else
                cptm = '0';
            end;
        when Regime_EL10 =>
            if IsSCTLR2EL1Enabled() then
                cptm = if el == EL0 then SCTLR2_EL1().CPTM0 else SCTLR2_EL1().CPTM;
            else
                cptm = '0';
            end;
        otherwise =>
            unreachable;
    end;

    return cptm;
end;
```

J1.2.3.26 PointerAddCheck

```
// PointerAddCheck()
// =================
// Apply Checked Pointer Arithmetic for addition.

func PointerAddCheck(result : bits(64), base : bits(64)) => bits(64)
begin
    return PointerCheckAtEL(PSTATE.EL, result, base, FALSE);
end;
```

J1.2.3.27 PointerAddCheckAtEL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// PointerAddCheckAtEL()
// =====================
// Apply Checked Pointer Arithmetic for addition at the specified EL.

func PointerAddCheckAtEL(el : bits(2), result : bits(64), base : bits(64)) => bits(64)
begin
    return PointerCheckAtEL(el, result, base, FALSE);
end;
```

J1.2.3.28 PointerCheckAtEL

```
// PointerCheckAtEL()
// ==================
// Apply Checked Pointer Arithmetic at the specified EL.

func PointerCheckAtEL(el : bits(2), result : bits(64), base : bits(64),
                      cptm_detected : boolean) => bits(64)
begin
    var rv : bits(64)  = result;

    let previous_detection : boolean = (base[55] != base[54]);
    let cpta_detected : boolean = (result[63:56] != base[63:56] || previous_detection);

    if ((cpta_detected && EffectiveCPTA(el) == '1') ||
          (cptm_detected && EffectiveCPTM(el) == '1')) then
        rv[63:55] = base[63:55];
        rv[54] = NOT(rv[55]);
    end;

    return rv;
end;
```

J1.2.3.29 PointerMultiplyAddCheck

```
// PointerMultiplyAddCheck()
// =========================
// Apply Checked Pointer Arithmetic for multiplication.

func PointerMultiplyAddCheck(result : bits(64), base : bits(64),
                             cptm_detected : boolean) => bits(64)
begin
    return PointerCheckAtEL(PSTATE.EL, result, base, cptm_detected);
end;
```

J1.2.3.30 IsD128Enabled

```
// IsD128Enabled()
// ===============
// Returns true if 128-bit page descriptor is enabled

func IsD128Enabled(el : bits(2)) => boolean
begin
    var d128enabled : boolean;
    if IsFeatureImplemented(FEAT_D128) then
        case el of
            when EL0 =>
                if !ELIsInHost(EL0) then
                    d128enabled = IsTCR2EL1Enabled() && TCR2_EL1().D128 == '1';
                else
                    d128enabled = IsTCR2EL2Enabled() && TCR2_EL2().D128 == '1';
                end;
            when EL1 =>
                d128enabled = IsTCR2EL1Enabled() && TCR2_EL1().D128 == '1';
            when EL2 =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                d128enabled = IsTCR2EL2Enabled() && IsInHost() && TCR2_EL2().D128 == '1';
            when EL3 =>
                d128enabled = TCR_EL3().D128 == '1';
        end;
    else
        d128enabled = FALSE;
    end;

    return d128enabled;
end;
```

J1.2.3.31 AArch64_CanTrapDC

```
// AArch64_CanTrapDC()
// ===================
// Determines whether the execution of the DC instruction can be trapped.

func AArch64_CanTrapDC(cachetype : CacheType, cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    return (!AArch64_TreatDCAsNOP(cachetype, cacheop, opscope) || ImpDefBool(
            "When DC is treated as NOP, data cache maintenance operations are trapped"));
end;
```

J1.2.3.32 AArch64_DC

```
// AArch64_DC()
// ============
// Perform Data Cache Operation.

func AArch64_DC(regval : bits(64), cachetype : CacheType,
                cacheop : CacheOp, opscope_in : CacheOpScope)
begin
    var opscope : CacheOpScope = opscope_in;
    var cache : CacheRecord;

    cache.acctype   = AccessType_DC;
    cache.cachetype = cachetype;
    cache.cacheop   = cacheop;
    cache.opscope   = opscope;

    if opscope == CacheOpScope_SetWay then
        let ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        cache.(setnum, waynum, level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
          (HCR_EL2().SWIO == '1' || HCR_EL2().[DC,VM] != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;
        end;

        CACHE_OP(cache);
        return;
    end;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID();
        else
            cache.is_vmid_valid = FALSE;
        end;
    else
        cache.is_vmid_valid = FALSE;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid          = ASID();
else
    cache.is_asid_valid = FALSE;
end;

if (opscope == CacheOpScope_PoPS &&
      ImpDefBool("Memory system does not support PoPS")) then
    opscope = CacheOpScope_PoC;
end;
if (opscope == CacheOpScope_PoDP &&
      ImpDefBool("Memory system does not support PoDP")) then
    opscope = CacheOpScope_PoP;
end;
if (opscope == CacheOpScope_PoP &&
      ImpDefBool("Memory system does not support PoP")) then
    opscope = CacheOpScope_PoC;
end;
var vaddress : bits(64) = regval;

var size : integer = 0;        // by default no watchpoint address
if cacheop == CacheOp_Invalidate then
    size = DataCacheWatchpointSize();
    vaddress = AlignDownSize{64}(regval, size as AddressSize);
end;

if opscope == CacheOpScope_PoE then
    cache.shareability     = Shareability_OSH;
    cache.paddress.address = regval[NUM_PABITS-1:0];
    let nse2 : bit = if IsFeatureImplemented(FEAT_RME_GDI) then regval[61] else '0';
    cache.paddress.paspace = DecodePASpace(nse2, regval[62], regval[63]);
    cache.cpas             = CPASAtPAS(cache.paddress.paspace);

    // If a Reserved encoding is selected, the instruction is permitted to be treated as a NOP.
    if cache.paddress.paspace != PAS_Realm then
        ExecuteAsNOP();
    end;

    if ImpDefBool("Apply granule protection check on DC to PoE") then
        var memaddrdesc : AddressDescriptor;
        let accdesc : AccessDescriptor = CreateAccDescDC(cache);
        memaddrdesc.paddress     = cache.paddress;
        memaddrdesc.fault.gpcf   = GranuleProtectionCheck(memaddrdesc, accdesc);

        if memaddrdesc.fault.gpcf.gpf != GPCF_None then
            memaddrdesc.fault.statuscode = Fault_GPCFOnOutput;
            memaddrdesc.fault.paddress   = memaddrdesc.paddress;
            memaddrdesc.fault.vaddress   = ARBITRARY : bits(64);
            AArch64_Abort(memaddrdesc.fault);
        end;
    end;
elsif opscope == CacheOpScope_PoPA then
    cache.shareability = Shareability_OSH;
    cache.paddress.address = regval[NUM_PABITS-1:0];
    let nse2 : bit = if IsFeatureImplemented(FEAT_RME_GDI) then regval[61] else '0';
    cache.paddress.paspace = DecodePASpace(nse2, regval[62], regval[63]);

    // If {NSE2, NSE, NS} is reserved, then no cache entries are required
    // to be cleaned or invalidated.
    if cache.paddress.paspace IN {PAS_NA6, PAS_NA7} then
        ExecuteAsNOP();
    end;
    cache.cpas = CPASAtPAS(cache.paddress.paspace);
else
    cache.vaddress  = vaddress;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let aligned : boolean = TRUE;
        let accdesc : AccessDescriptor = CreateAccDescDC(cache);
        var memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(vaddress, accdesc,
                                                                       aligned, size);
        if IsFault(memaddrdesc) then
            memaddrdesc.fault.vaddress = regval;
            AArch64_Abort(memaddrdesc.fault);
        end;

        cache.paddress   = memaddrdesc.paddress;
        cache.cpas       = CPASAtPAS(memaddrdesc.paddress.paspace);

        if (opscope IN {
                        CacheOpScope_PoDP,
                        CacheOpScope_PoP,
                        CacheOpScope_PoC,
                        CacheOpScope_PoU
                    }) then
            cache.shareability = memaddrdesc.memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;
        end;
    end;

    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            HCR_EL2().[DC,VM] != '00') then
        cache.cacheop = CacheOp_CleanInvalidate;
    end;

    // If Secure state is not implemented, but RME is, the instruction acts as a NOP
    if cache.cpas == CPAS_Secure && !HaveSecureState() then
        return;
    end;

    CACHE_OP(cache);
    return;
end;
```

J1.2.3.33 AArch64_MemZero

```
// AArch64_MemZero()
// =================

func AArch64_MemZero(regval : bits(64), cachetype : CacheType)
begin
    let size : integer{} = 4*(2^(UInt(DCZID_EL0().BS)));
    assert size <= MAX_ZERO_BLOCK_SIZE;
    if IsFeatureImplemented(FEAT_MTE2) then
        assert size >= TAG_GRANULE;
    end;
    let vaddress : bits(64) = AlignDownSize{}(regval, size);

    let accdesc : AccessDescriptor = CreateAccDescDCZero(cachetype);

    if cachetype != CacheType_Data then
        AArch64_WriteTagMem(regval, cachetype, accdesc, size);
    end;

    if cachetype IN {CacheType_Data, CacheType_Data_Tag} then
        AArch64_DataMemZero(regval, vaddress, accdesc, size);
    end;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.34 AArch64_TreatDCAsNOP

```
// AArch64_TreatDCAsNOP()
// ======================
// Determines whether the execution of the DC instruction is treated as a NOP.

func AArch64_TreatDCAsNOP(cachetype : CacheType, cacheop : CacheOp,
                          opscope : CacheOpScope) => boolean
begin
    // DC to PoU: IMPLEMENTATION DEFINED - treated as NOP if LoUU and LoUIS are 0
    if opscope == CacheOpScope_PoU && CLIDR_EL1().LoUU == '000' && CLIDR_EL1().LoUIS == '000' then
        return ImpDefBool("DC to PoU is treated as a NOP");
    end;

    // DC to PoC: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0
    if opscope == CacheOpScope_PoC && CLIDR_EL1().LoC == '000' then
        return ImpDefBool("DC to PoC is treated as a NOP");
    end;

    // DC to PoP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP unsupported
    if (opscope == CacheOpScope_PoP && CLIDR_EL1().LoC == '000' &&
          !ImpDefBool("Memory system supports PoP")) then
        return ImpDefBool("DC to PoP is treated as a NOP");
    end;

    // DC to PoDP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP/PoDP unsupported
    if (opscope == CacheOpScope_PoDP && CLIDR_EL1().LoC == '000' &&
          !ImpDefBool("Memory system supports PoP") &&
          !ImpDefBool("Memory system supports PoDP")) then
        return ImpDefBool("DC to PoDP is treated as a NOP");
    end;

    return FALSE;
end;
```

J1.2.3.35 MemZero

```
// MemZero block size
// ==================

constant MAX_ZERO_BLOCK_SIZE : integer = 2048;
```

J1.2.3.36 AArch64_ExceptionReturn

```
// AArch64_ExceptionReturn()
// =========================

func AArch64_ExceptionReturn(new_pc_in : bits(64), spsr : bits(64))
begin
    var new_pc : bits(64) = new_pc_in;

    if IsFeatureImplemented(FEAT_IESB) then
        var sync_errors : boolean = SCTLR_ELx().IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3().[EA,NMEA] == '11' && PSTATE.EL == EL3);
        end;
        if sync_errors then
            SynchronizeErrors();
            let iesb_req : boolean = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
        end;
    end;

    var brbe_source_allowed : boolean = FALSE;
    var brbe_source_address : bits(64) = Zeros{};
    if IsFeatureImplemented(FEAT_BRBE) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = PC64();
    end;


    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_ELx().EOS == '1' then
        SynchronizeContext();
    end;

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    let source_el : bits(2) = PSTATE.EL;
    let illegal_psr_state : boolean = IllegalExceptionReturn{64}(spsr);
    SetPSTATEFromPSR{64}(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr[4] == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc[63:32] = ARBITRARY : bits(32);
        new_pc[1:0] = ARBITRARY : bits(2);
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc[0] = '0';                 // T32
        else
            new_pc[1:0] = '00';              // A32
        end;
    else                                     // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64_BranchAddr(new_pc, PSTATE.EL);
    end;

    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEExceptionReturn(new_pc, source_el,
                            brbe_source_allowed, brbe_source_address);
    end;


    if UsingAArch32() then
        if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then ResetSVEState(); end;

        // 32 most significant bits are ignored.
        let branch_conditional : boolean = FALSE;
        BranchTo{32}(new_pc[31:0], BranchType_ERET, branch_conditional);
    else
        BranchToAddr{64}(new_pc, BranchType_ERET);
    end;

    CheckExceptionCatch(FALSE);              // Check for debug event on exception return
end;
```

J1.2.3.37 AArch64_ExclusiveMonitorsPass

```
// AArch64_ExclusiveMonitorsPass()
// ==============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give a memory abort.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_ExclusiveMonitorsPass(address : bits(64), size : integer{1, 2, 4, 8, 16, 32},
                                   accdesc : AccessDescriptor) => boolean
begin
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, size) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
    end;

    if !AArch64_IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;
    end;

    let memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    var passed : boolean = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    return passed;
end;
```

J1.2.3.38 AArch64_IsExclusiveVA

```
// AArch64_IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.


impdef func AArch64_IsExclusiveVA(address : bits(64),
                                  processorid : integer,
                                  size : integer) => boolean
begin
    return TRUE;
end;
```

J1.2.3.39 AArch64_MarkExclusiveVA

```
// AArch64_MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

func AArch64_MarkExclusiveVA(address : bits(64), processorid : integer, size : integer)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.40 AArch64_SetExclusiveMonitors

```
// AArch64_SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

func AArch64_SetExclusiveMonitors(address : bits(64), size : integer{1, 2, 4, 8, 16, 32},
                                  accdesc : AccessDescriptor)
begin
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, size) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
    end;

    let memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    end;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64_MarkExclusiveVA(address, ProcessorID(), size);
end;
```

J1.2.3.41 DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

func DecodeRegExtend(op : bits(3)) => ExtendType
begin
    case op of
        when '000' => return ExtendType_UXTB;
        when '001' => return ExtendType_UXTH;
        when '010' => return ExtendType_UXTW;
        when '011' => return ExtendType_UXTX;
        when '100' => return ExtendType_SXTB;
        when '101' => return ExtendType_SXTH;
        when '110' => return ExtendType_SXTW;
        when '111' => return ExtendType_SXTX;
    end;
end;
```

J1.2.3.42 ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

func ExtendReg{N}(reg : integer, exttype : ExtendType, shift : integer{0..4}) => bits(N)
begin
    let val : bits(N) = X{}(reg);
    var unsigned : boolean;
    var len : ESize;

    case exttype of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when ExtendType_SXTB => unsigned = FALSE; len = 8;
        when ExtendType_SXTH => unsigned = FALSE; len = 16;
        when ExtendType_SXTW => unsigned = FALSE; len = 32;
        when ExtendType_SXTX => unsigned = FALSE; len = 64;
        when ExtendType_UXTB => unsigned = TRUE;  len = 8;
        when ExtendType_UXTH => unsigned = TRUE;  len = 16;
        when ExtendType_UXTW => unsigned = TRUE;  len = 32;
        when ExtendType_UXTX => unsigned = TRUE;  len = 64;
    end;

    // Sign or zero extend bottom LEN bits of register and shift left by SHIFT
    let nbits : integer{} = Min(len, N) as integer{8..N};
    let extval : bits(N) = Extend{}(val[nbits-1:0], unsigned);
    return LSL(extval, shift);
end;
```

J1.2.3.43 ExtendType

```
// ExtendType
// ==========
// AArch64 register extend and shift.

type ExtendType of enumeration {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                                ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

J1.2.3.44 FPConvOp

```
// FPConvOp
// ========
// Floating-point convert/move instruction types.

type FPConvOp of enumeration {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                              FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
                              FPConvOp_CVT_FtoI_JS
};
```

J1.2.3.45 FPMaxMinOp

```
// FPMaxMinOp
// ==========
// Floating-point min/max instruction types.

type FPMaxMinOp of enumeration {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                                FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

J1.2.3.46 CheckFPMREnabled

```
// CheckFPMREnabled()
// ==================
// Check for Undefined instruction exception on indirect FPMR accesses.

func CheckFPMREnabled()
begin
    assert IsFeatureImplemented(FEAT_FPMR);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            if SCTLR_EL1().EnFPM == '0' then Undefined(); end;
        else
            if SCTLR_EL2().EnFPM == '0' then Undefined(); end;
        end;
    end;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !IsHCRXEL2Enabled() || HCRX_EL2().EnFPM == '0' then Undefined(); end;
    end;

    if PSTATE.EL != EL3 && HaveEL(EL3) then
        if SCR_EL3().EnFPM == '0' then Undefined(); end;
    end;
end;
```

J1.2.3.47 FPClampScale

```
// FPClampScale()
// ==============

func FPClampScale{N}(op : bits(N), scale_in : integer) => integer
begin
    assert N IN {16,32,64};
    var E : integer;
    var exp : integer;
    case N of
        when 16 =>
            E = 5;
            exp = UInt(op[14:10]);
        when 32 =>
            E = 8;
            exp = UInt(op[30:23]);
        when 64 =>
            E = 11;
            exp = UInt(op[62:52]);
        otherwise => unreachable;
    end;

    let F : integer = N - (E + 1);
    let emax : integer = (1 << E) - 1;
    let min_scale : integer = -(F + 1);
    let max_scale : integer = emax + (F + 1);

    let scale : integer = Max(min_scale - exp, Min(scale_in, max_scale - exp));
    return scale;
end;
```

J1.2.3.48 FPScale

```
// FPScale()
// =========

func FPScale{N}(op : bits(N), scale : integer, fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N);
    let (fptype,sign,value) : (FPType, bit, real) = FPUnpack{N}(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN{N}(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero{N}(sign);
    elsif fptype == FPType_Infinity then
        result = FPInfinity{N}(sign);
    else
        let clamped_scale : integer = FPClampScale{N}(op, scale);
        result = FPRound{N}(value * (2.0^clamped_scale), fpcr);
        FPProcessDenorm(fptype, N, fpcr);
    end;
    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.49 FPUnaryOp

```
// FPUnaryOp
// =========
// Floating-point unary instruction types.

type FPUnaryOp of enumeration {FPUnaryOp_ABS, FPUnaryOp_MOV,
                               FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

J1.2.3.50 FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

func FPRSqrtStepFused{N}(op1_in : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16, 32, 64};
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(N);
    var op1 : bits(N) = op1_in;
    var done : boolean;
    op1 = FPNeg{N}(op1, fpcr);
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    let fpexc : boolean = !altfp;                    // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end;         // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; end;            // Use RNE rounding mode

    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    let rounding : FPRounding = FPRoundingMode(fpcr);

    if !done then
        let inf1 : boolean = (type1 == FPType_Infinity);
        let inf2 : boolean = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive{N}('0');
        elsif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        else
            // Fully fused multiply-add and halve
            let result_value : real = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                let sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;
    end;

    return result;
end;
```

J1.2.3.51 FPRecipStepFused

```
// FPRecipStepFused()
// ==================

func FPRecipStepFused{N}(op1_in : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16, 32, 64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var fpcr : FPCR_Type = fpcr_in;
    var op1 : bits(N)    = op1_in;
    var result : bits(N);
    var done : boolean;
    op1 = FPNeg{N}(op1, fpcr);

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    let fpexc : boolean = !altfp;                    // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end;         // Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00'; end;         // Use RNE rounding mode

    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    let rounding : FPRounding = FPRoundingMode(fpcr);

    if !done then
        let inf1 : boolean  = (type1 == FPType_Infinity);
        let inf2 : boolean  = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo{N}('0');
        elsif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        else
            // Fully fused multiply-add
            let result_value : real = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                let sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;
    end;

    return result;
end;
```

J1.2.3.52 AddGCSExRecord

```
// AddGCSExRecord()
// ================
// Generates and then writes an exception record to the
// current Guarded Control Stack.

func AddGCSExRecord(elr : bits(64), spsr : bits(64), lr : bits(64))
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem{64}(ptr-8, accdesc) = lr;
    Mem{64}(ptr-16, accdesc) = spsr;
    Mem{64}(ptr-24, accdesc) = elr;
    Mem{64}(ptr-32, accdesc) = Zeros{60}::'1001';

    // Decrement the pointer value
    ptr = ptr - 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SetCurrentGCSPointer(ptr);
    return;
end;
```

J1.2.3.53 AddGCSRecord

```
// AddGCSRecord()
// ==============
// Generates and then writes a record to the current Guarded
// control stack.

func AddGCSRecord(vaddress : bits(64))
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem{64}(ptr-8, accdesc) = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPointer(ptr);
    return;
end;
```

J1.2.3.54 CheckGCSExRecord

```
// CheckGCSExRecord()
// ==================
// Validates the provided values against the top entry of the
// current Guarded Control Stack.

func CheckGCSExRecord(elr : bits(64), spsr : bits(64), lr : bits(64), gcsinst_type : GCSInstruction)
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_LOAD, privileged);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    let recorded_first_dword : bits(64) = Mem{64}(ptr, accdesc);
    if recorded_first_dword != Zeros{60}::'1001' then
        GCSDataCheckException(gcsinst_type);
    end;

    // Check the ELR matches the recorded value
    let recorded_elr : bits(64) = Mem{64}(ptr+8, accdesc);
    if recorded_elr != elr then
        GCSDataCheckException(gcsinst_type);
    end;

    // Check the SPSR matches the recorded value
    let recorded_spsr : bits(64) = Mem{64}(ptr+16, accdesc);
    if recorded_spsr != spsr then
        GCSDataCheckException(gcsinst_type);
    end;

    // Check the LR matches the recorded value
    let recorded_lr : bits(64) = Mem{64}(ptr+24, accdesc);
    if recorded_lr != lr then
        GCSDataCheckException(gcsinst_type);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
end;
```

J1.2.3.55 CheckGCSSTREnabled

```
// CheckGCSSTREnabled()
// ====================
// Trap GCSSTR or GCSSTTR instruction if trapping is enabled.

func CheckGCSSTREnabled()
begin
    case PSTATE.EL of
        when EL0 =>
            if GCSCRE0_EL1().STREn == '0' then
                if EL2Enabled() && HCR_EL2().TGE == '1' then
                    GCSSTRTrapException(EL2);
                else
                    GCSSTRTrapException(EL1);
                end;
            end;
        when EL1 =>
            if GCSCR_EL1().STREn == '0' then
                GCSSTRTrapException(EL1);
            elsif (EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
                    HFGITR_EL2().nGCSSTR_EL1 == '0') then
                GCSSTRTrapException(EL2);
            end;
        when EL2 =>
            if GCSCR_EL2().STREn == '0' then
                GCSSTRTrapException(EL2);
            end;
        when EL3 =>
            if GCSCR_EL3().STREn == '0' then
                GCSSTRTrapException(EL3);
            end;
    end;
    return;
end;
```

J1.2.3.56 EXLOCKException

```
// EXLOCKException()
// =================
// Handle an EXLOCK exception condition.

func EXLOCKException()
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_GCSFail);
    except.syndrome.iss[24] = '0';
    except.syndrome.iss[23:20] = '0001';
    except.syndrome.iss[19:0] = Zeros{20};
    AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.57 GCSDataCheckException

```
// GCSDataCheckException()
// =======================
// Handle a GCS data check fault condition.

func GCSDataCheckException(gcsinst_type : GCSInstruction)
begin
    var target_el : bits(2);
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    var rn_unknown : boolean  = FALSE;
    var is_ret : boolean      = FALSE;
    var is_reta : boolean     = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if (EL2Enabled() && HCR_EL2().TGE == '1') then EL2 else EL1;
    else
        target_el = PSTATE.EL;
    end;
    var except : ExceptionRecord = ExceptionSyndrome(Exception_GCSFail);
    case gcsinst_type of
        when GCSInstType_PRET =>
            except.syndrome.iss[4:0] = '00000';
            is_ret = TRUE;
        when GCSInstType_POPM =>
            except.syndrome.iss[4:0] = '00001';
        when GCSInstType_PRETAA =>
            except.syndrome.iss[4:0] = '00010';
            is_reta = TRUE;
        when GCSInstType_PRETAB =>
            except.syndrome.iss[4:0] = '00011';
            is_reta = TRUE;
        when GCSInstType_SS1 =>
            except.syndrome.iss[4:0] = '00100';
        when GCSInstType_SS2 =>
            except.syndrome.iss[4:0] = '00101';
            rn_unknown = TRUE;
        when GCSInstType_POPCX =>
            rn_unknown = TRUE;
            except.syndrome.iss[4:0] = '01000';
        when GCSInstType_POPX =>
            except.syndrome.iss[4:0] = '01001';
    end;
    if rn_unknown == TRUE then
        except.syndrome.iss[9:5] = ARBITRARY : bits(5);
    elsif is_ret == TRUE then
        except.syndrome.iss[9:5] = ThisInstr()[9:5];
    elsif is_reta == TRUE then
        except.syndrome.iss[9:5] = '11110';
    else
        except.syndrome.iss[9:5] = ThisInstr()[4:0];
    end;
    except.syndrome.iss[24:10] = Zeros{15};
    except.vaddress = ARBITRARY : bits(64);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.3.58 GCSEnabled

```
// GCSEnabled()
// ============
// Returns TRUE if the Guarded Control Stack is enabled at
// the provided Exception level.

func GCSEnabled(el : bits(2)) => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    if UsingAArch32() then
        return FALSE;
    end;

    if HaveEL(EL3) && el != EL3 && SCR_EL3().GCSEn == '0' then
        return FALSE;
    end;

    if (el IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2().GCSEn == '0')) then
        return FALSE;
    end;

    return GCSPCRSelected(el);
end;
```

J1.2.3.59 GCSInstruction

```
// GCSInstruction
// ==============

type GCSInstruction of enumeration {
    GCSInstType_PRET,    // Procedure return without Pointer authentication
    GCSInstType_POPM,    // GCSPOPM instruction
    GCSInstType_PRETAA,  // Procedure return with Pointer authentication that used key A
    GCSInstType_PRETAB,  // Procedure return with Pointer authentication that used key B
    GCSInstType_SS1,     // GCSSS1 instruction
    GCSInstType_SS2,     // GCSSS2 instruction
    GCSInstType_POPCX,   // GCSPOPCX instruction
    GCSInstType_POPX     // GCSPOPX instruction
};
```

J1.2.3.60 GCSPCREnabled

```
// GCSPCREnabled()
// ===============
// Returns TRUE if the Guarded Control Stack is PCR enabled
// at the provided Exception level.

func GCSPCREnabled(el : bits(2)) => boolean
begin
    return GCSPCRSelected(el) && GCSEnabled(el);
end;
```

J1.2.3.61 GCSPCRSelected

```
// GCSPCRSelected()
// ================
// Returns TRUE if the Guarded Control Stack is PCR selected
// at the provided Exception level.

func GCSPCRSelected(el : bits(2)) => boolean
begin
    case el of
        when EL0 => return GCSCRE0_EL1().PCRSEL == '1';
        when EL1 => return GCSCR_EL1().PCRSEL == '1';
        when EL2 => return GCSCR_EL2().PCRSEL == '1';
        when EL3 => return GCSCR_EL3().PCRSEL == '1';
    end;
    unreachable;
    return TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.62 GCSPOPCX

```
// GCSPOPCX()
// ==========
// Called to pop and compare a GCS exception return record.

func GCSPOPCX()
begin
    let spsr : bits(64) = SPSR_ELx();
    CheckGCSExRecord(ELR_ELx(), spsr, X{64}(30), GCSInstType_POPCX);
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
end;
```

J1.2.3.63 GCSPOPM

```
// GCSPOPM()
// =========
// Called to pop a GCS procedure return record.

func GCSPOPM() => bits(64)
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_LOAD, privileged);

    ptr = GetCurrentGCSPointer();
    let gcs_entry : bits(64) = Mem{64}(ptr, accdesc);

    if gcs_entry[1:0] != '00' then
        GCSDataCheckException(GCSInstType_POPM);
    end;

    ptr = ptr + 8;
    SetCurrentGCSPointer(ptr);
    return gcs_entry;
end;
```

J1.2.3.64 GCSPOPX

```
// GCSPOPX()
// =========
// Called to pop a GCS exception return record.

func GCSPOPX()
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_LOAD, privileged);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    let recorded_first_dword : bits(64) = Mem{64}(ptr, accdesc);
    if recorded_first_dword != Zeros{60}::'1001' then
        GCSDataCheckException(GCSInstType_POPX);
    end;

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    let recorded_elr : bits(64) = Mem{64}(ptr+8, accdesc);
    let recorded_spsr : bits(64) = Mem{64}(ptr+16, accdesc);
    let recorded_lr : bits(64) = Mem{64}(ptr+24, accdesc);

    // Increment the pointer value
    ptr = ptr + 32;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SetCurrentGCSPointer(ptr);
    return;
end;
```

J1.2.3.65 GCSPUSHM

```
// GCSPUSHM()
// ==========
// Called to push a GCS procedure return record.

func GCSPUSHM(value : bits(64))
begin
    AddGCSRecord(value);
    return;
end;
```

J1.2.3.66 GCSPUSHX

```
// GCSPUSHX()
// ==========
// Called to push a GCS exception return record.

func GCSPUSHX()
begin
    let spsr : bits(64) = SPSR_ELx();
    AddGCSExRecord(ELR_ELx(), spsr, X{64}(30));
    PSTATE.EXLOCK = '0';
    return;
end;
```

J1.2.3.67 GCSReturnValueCheckEnabled

```
// GCSReturnValueCheckEnabled()
// ============================
// Returns TRUE if the Guarded Control Stack has return value
// checking enabled at the current Exception level.

func GCSReturnValueCheckEnabled(el : bits(2)) => boolean
begin
    if UsingAArch32() then
        return FALSE;
    end;
    case el of
        when EL0 => return GCSCRE0_EL1().RVCHKEN == '1';
        when EL1 => return GCSCR_EL1().RVCHKEN == '1';
        when EL2 => return GCSCR_EL2().RVCHKEN == '1';
        when EL3 => return GCSCR_EL3().RVCHKEN == '1';
    end;
end;
```

J1.2.3.68 GCSSS1

```
// GCSSS1()
// ========
// Operational pseudocode for GCSSS1 instruction.

func GCSSS1(incoming_pointer : bits(64))
begin
    var outgoing_pointer, cmpoperand, operand, data : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCSSS1(privileged);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid GCS cap record is expected
    cmpoperand = incoming_pointer[63:12]::'000000000001';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // In-progress GCS cap record should be stored if the comparison is successful
    operand = outgoing_pointer[63:3]::'101';

    data = MemAtomic{64}(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer[63:3]::'000');
    else
        GCSDataCheckException(GCSInstType_SS1);
    end;
    return;
end;
```

J1.2.3.69 GCSSS2

```
// GCSSS2()
// ========
// Operational pseudocode for GCSSS2 instruction.

func GCSSS2() => bits(64)
begin
    var outgoing_pointer, incoming_pointer, outgoing_value : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc_ld : AccessDescriptor = CreateAccDescGCS(MemOp_LOAD, privileged);
    let accdesc_st : AccessDescriptor = CreateAccDescGCS(MemOp_STORE, privileged);
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem{64}(incoming_pointer, accdesc_ld);

    if outgoing_value[2:0] == '101' then  //in_progress token
        outgoing_pointer[63:3] = (outgoing_value[63:3]) - 1;
        outgoing_pointer[2:0] = '000';
        outgoing_value = outgoing_pointer[63:12]::'000000000001';
        Mem{64}(outgoing_pointer, accdesc_st) = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType_SS2);
    end;
    return outgoing_pointer;
end;
```

J1.2.3.70 GCSSTRTrapException

```
// GCSSTRTrapException()
// =====================
// Handle a trap on GCSSTR instruction condition.

func GCSSTRTrapException(target_el : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_GCSFail);
    except.syndrome.iss[24]      = '0';
    except.syndrome.iss[23:20]   = '0010';
    except.syndrome.iss[19:15]   = '00000';
    except.syndrome.iss[14:10]   = ThisInstr()[9:5];
    except.syndrome.iss[9:5]     = ThisInstr()[4:0];
    except.syndrome.iss[4:0]     = '00000';
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.71 GCSSynchronizationBarrier

```
// GCSSynchronizationBarrier()
// ===========================
// Barrier instruction that synchronizes GCS accesses
// in relation to other load and store accesses

impdef func GCSSynchronizationBarrier()
begin
    return;
end;
```

J1.2.3.72 GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// ====================

func GetCurrentEXLOCKEN() => boolean
begin
    if Halted() || Restarting() then
        return FALSE;
    end;

    case PSTATE.EL of
        when EL0 =>
            unreachable;
        when EL1 =>
            return GCSCR_EL1().EXLOCKEN == '1';
        when EL2 =>
            return GCSCR_EL2().EXLOCKEN == '1';
        when EL3 =>
            return GCSCR_EL3().EXLOCKEN == '1';
    end;
end;
```

J1.2.3.73 GetCurrentGCSPointer

```
// GetCurrentGCSPointer()
// ======================
// Returns the value of the current GCS pointer register.

func GetCurrentGCSPointer() => bits(64)
begin
    var ptr : bits(64);

    case PSTATE.EL of
        when EL0 =>
            ptr = GCSPR_EL0().PTR::'000';
        when EL1 =>
            ptr = GCSPR_EL1().PTR::'000';
        when EL2 =>
            ptr = GCSPR_EL2().PTR::'000';
        when EL3 =>
            ptr = GCSPR_EL3().PTR::'000';
    end;
    return ptr;
end;
```

J1.2.3.74 LoadCheckGCSRecord

```
// LoadCheckGCSRecord()
// ====================
// Validates the provided address against the top entry of the
// current Guarded Control Stack.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func LoadCheckGCSRecord(vaddress : bits(64), gcsinst_type : GCSInstruction) => bits(64)
begin
    var ptr : bits(64);
    var recorded_va : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp_LOAD, privileged);

    ptr = GetCurrentGCSPointer();
    recorded_va = Mem{64}(ptr, accdesc);
    if GCSReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress) then
        GCSDataCheckException(gcsinst_type);
    end;

    return recorded_va;
end;
```

J1.2.3.75 SetCurrentGCSPointer

```
// SetCurrentGCSPointer()
// ======================
// Writes a value to the current GCS pointer register.

func SetCurrentGCSPointer(ptr : bits(64))
begin
    case PSTATE.EL of
        when EL0 =>
            GCSPR_EL0().PTR = ptr[63:3];
        when EL1 =>
            GCSPR_EL1().PTR = ptr[63:3];
        when EL2 =>
            GCSPR_EL2().PTR = ptr[63:3];
        when EL3 =>
            GCSPR_EL3().PTR = ptr[63:3];
    end;
    return;
end;
```

J1.2.3.76 hacdbs

```
constant HACDBS_ERR_REASON_STRUCTF : bits(2) = '01';
constant HACDBS_ERR_REASON_IPAF : bits(2)    = '10';
constant HACDBS_ERR_REASON_IPHACF : bits(2)  = '11';
```

J1.2.3.77 IsHACDBSIRQAsserted

```
// IsHACDBSIRQAsserted()
// =====================
// Returns TRUE if HACDBSIRQ is asserted, and FALSE otherwise.

impdef func IsHACDBSIRQAsserted() => boolean
begin
    return FALSE;
end;
```

J1.2.3.78 ProcessHACDBSEntry

```
// ProcessHACDBSEntry()
// ====================
// Process a single entry entry from the HACDBS.

func ProcessHACDBSEntry()
begin
    if !IsFeatureImplemented(FEAT_HACDBS) then return; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if (HaveEL(EL3) && SCR_EL3().HACDBSEn == '0') || HACDBSBR_EL2().EN == '0' then
    SetInterruptRequestLevel(InterruptID_HACDBSIRQ, LOW);
    return;
end;

if HCR_EL2().VM == '0' then return; end;

if (UInt(HACDBSCONS_EL2().INDEX) >= (2 ^ (UInt(HACDBSBR_EL2().SZ) + 12)) DIVRM 8 ||
      HACDBSCONS_EL2().ERR_REASON != '00') then
    SetInterruptRequestLevel(InterruptID_HACDBSIRQ, HIGH);
    return;
elsif IsHACDBSIRQAsserted() then
    SetInterruptRequestLevel(InterruptID_HACDBSIRQ, LOW);
end;

let hacdbs_size : integer{} = UInt(HACDBSBR_EL2().SZ);
var baddr : bits(NUM_PABITS) = HACDBSBR_EL2().BADDR[NUM_PABITS-13 : 0] :: Zeros{12};
baddr[11 + hacdbs_size : 12] = Zeros{hacdbs_size};

var accdesc : AccessDescriptor = CreateAccDescHACDBS();

var addrdesc : AddressDescriptor;
addrdesc.paddress.address = baddr + (8 * UInt(HACDBSCONS_EL2().INDEX));
let nse2 : bit = '0';     // NSE2 has the Effective value of 0 within a PE.
addrdesc.paddress.paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(), EffectiveSCR_EL3_NS());

// Accesses to the HACDBS use the same memory attributes as used for stage 2 translation walks.
addrdesc.memattrs = WalkMemAttrs(VTCR_EL2().SH0, VTCR_EL2().IRGN0, VTCR_EL2().ORGN0);
let emec : bit = (if IsFeatureImplemented(FEAT_MEC) &&
                     IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');
addrdesc.mecid = AArch64_S2TTWalkMECID(emec, accdesc.ss);

var fault : FaultRecord = NoFault(accdesc);

if IsFeatureImplemented(FEAT_RME) then
    fault.gpcf = GranuleProtectionCheck(addrdesc, accdesc);

    if fault.gpcf.gpf != GPCF_None then
        HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
        return;
    end;
end;

var memstatus : PhysMemRetStatus;
var hacdbs_entry : bits(64);
(memstatus, hacdbs_entry) = PhysMemRead{64}(addrdesc, accdesc);

if IsFault(memstatus) then
    HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
    return;
end;

if BigEndian(accdesc.acctype) then
    hacdbs_entry = BigEndianReverse{64}(hacdbs_entry);
end;

// If the Valid field is clear, do not perform any cleaning operation
// and increment HACDBSCONS_EL2.INDEX.
if hacdbs_entry[0] == '0' then
    HACDBSCONS_EL2().INDEX = HACDBSCONS_EL2().INDEX + 1;
    return;
end;

accdesc = CreateAccDescTTEUpdate(accdesc);
var ipa : AddressDescriptor;
ipa.paddress.address = hacdbs_entry[NUM_PABITS-1:12] :: Zeros{12};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let nsipa : bit       = hacdbs_entry[11];
let paspace : PASpace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(), EffectiveSCR_EL3_NS());
ipa.paddress.paspace = (if accdesc.ss == SS_Secure && nsipa == '1' then PAS_NonSecure
                                                                  else paspace);

let s1aarch64 : boolean = TRUE;
let walkparams : S2TTWParams = AArch64_GetS2TTWParams(accdesc.ss, ipa.paddress.paspace,
                                                      s1aarch64);

var descpaddr : AddressDescriptor;
var walkstate : TTWState;
var descriptor : bits(128);
if walkparams.d128 == '1' then
    (fault, descpaddr, walkstate, descriptor) = AArch64_S2Walk{128}(fault, ipa, walkparams,
                                                                    accdesc);
else
    (fault, descpaddr, walkstate, descriptor[63:0]) = AArch64_S2Walk{64}(fault, ipa, walkparams,
                                                                         accdesc);
end;

// If the Access flag on the Block or Page descriptor is set to 0, this does not generate
// an Access flag fault and the PE can still perform the cleaning operation on that descriptor.
if fault.statuscode == Fault_AccessFlag then
    fault.statuscode = Fault_None;
elsif fault.statuscode != Fault_None then
    HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_IPAF;
    return;
end;

let hacdbs_level : integer = SInt(hacdbs_entry[3:1]);
if walkstate.level != hacdbs_level || walkstate.contiguous == '1' then
    HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_IPHACF;
    return;
end;

// For the purpose of cleaning HACDBS entries, it is not required that HW update of dirty bit
// is enabled for a descriptor to be qualified as writable-clean or writable-dirty.

// Check if the descriptor is neither writable-clean nor writable-dirty.
if walkparams.s2pie == '1' then
    let perms : S2AccessControls = AArch64_S2ComputePermissions(walkstate.permissions,
                                                                walkparams, accdesc);
    if perms.w == '0' && perms.w_mmu == '0' then
        HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_IPHACF;
        return;
    end;

// If DBM is 0, the descriptor is not writable-clean or writable-dirty.
elsif descriptor[51] == '0' then
    HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_IPHACF;
    return;
end;

// If the descriptor is writable-clean, do not perform any cleaning
// operation and increment HACDBSCONS_EL2.INDEX.
if descriptor[7] == '0' then
    HACDBSCONS_EL2().INDEX = HACDBSCONS_EL2().INDEX + 1;
    return;
end;

var new_descriptor : bits(128) = descriptor;
new_descriptor[7] = '0';

let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
if walkparams.d128 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (fault, -) = AArch64_MemSwapTableDesc{128}(fault, descriptor, new_descriptor, walkparams.ee,
                                                   descaccess, descpaddr);
    else
        (fault, -) = AArch64_MemSwapTableDesc{64}(fault, descriptor[63:0], new_descriptor[63:0],
                                                  walkparams.ee, descaccess, descpaddr);
    end;

    if fault.statuscode != Fault_None then
        HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_IPAF;
    else
        HACDBSCONS_EL2().INDEX = HACDBSCONS_EL2().INDEX + 1;
    end;

    return;
end;
```

J1.2.3.79 AArch64_CanTrapIC

```
// AArch64_CanTrapIC()
// ===================
// Determines whether the execution of the IC instruction can be trapped.

func AArch64_CanTrapIC(cachetype : CacheType, cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    return (!AArch64_TreatICAsNOP(cachetype, cacheop, opscope) || ImpDefBool(
            "When IC is treated as NOP, data cache maintenance operations are trapped"));
end;
```

J1.2.3.80 AArch64_EffectiveOpScope

```
// AArch64_EffectiveOpScope()
// ==========================

func AArch64_EffectiveOpScope(opscope_in : CacheOpScope) => CacheOpScope
begin
    var opscope : CacheOpScope = opscope_in;

    if PSTATE.EL != EL1 then return opscope; end;

    if opscope == CacheOpScope_ALLU && EL2Enabled() && HCR_EL2().FB == '1' then
        opscope = CacheOpScope_ALLUIS;
    end;

    return opscope;
end;
```

J1.2.3.81 AArch64_IC

```
// AArch64_IC()
// ============
// Perform Instruction Cache Operation.

func AArch64_IC(opscope : CacheOpScope)
begin
    let regval : bits(64) = ARBITRARY : bits(64);
    AArch64_IC(regval, opscope);
end;

// AArch64_IC()
// ============
// Perform Instruction Cache Operation.

func AArch64_IC(regval : bits(64), opscope_in : CacheOpScope)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var cache : CacheRecord;
var opscope : CacheOpScope = opscope_in;

cache.acctype   = AccessType_IC;
cache.cachetype = CacheType_Instruction;
cache.cacheop   = CacheOp_Invalidate;
cache.opscope   = opscope;

if opscope IN {CacheOpScope_ALLU,
               CacheOpScope_ALLUIS} then
    let ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
    cache.cpas = CPASAtSecurityState(ss);
    opscope = AArch64_EffectiveOpScope(opscope);

    case opscope of
        when CacheOpScope_ALLU =>
            cache.shareability = Shareability_NSH;
        when CacheOpScope_ALLUIS =>
            cache.shareability = Shareability_ISH;
        otherwise =>
            unreachable;
    end;

    cache.regval = regval;
    CACHE_OP(cache);
else
    assert opscope == CacheOpScope_PoU;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID();
        else
            cache.is_vmid_valid = FALSE;
        end;
    else
        cache.is_vmid_valid = FALSE;
    end;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID();
    else
        cache.is_asid_valid = FALSE;
    end;

    let vaddress : bits(64) = regval;
    cache.vaddress = regval;

    let accdesc : AccessDescriptor = CreateAccDescIC(cache);
    let aligned : boolean = TRUE;
    let size : integer = 0;
    var memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(vaddress, accdesc,
                                                                   aligned, size);

    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = regval;
        AArch64_Abort(memaddrdesc.fault);
    end;

    cache.cpas         = CPASAtPAS(memaddrdesc.paddress.paspace);
    cache.paddress     = memaddrdesc.paddress;
    cache.shareability = memaddrdesc.memattrs.shareability;
    if memaddrdesc.memattrs.shareability == Shareability_OSH then
        cache.shareability = Shareability_ISH;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        CACHE_OP(cache);
    end;
    return;
end;
```

J1.2.3.82 AArch64_TreatICAsNOP

```
// AArch64_TreatICAsNOP()
// ======================
// Determines whether the execution of the IC instruction is treated as a NOP.

func AArch64_TreatICAsNOP(cachetype : CacheType, cacheop : CacheOp,
                          opscope : CacheOpScope) => boolean
begin
    if CTR_EL0().DIC == '1' then
        return ImpDefBool("IC is treated as NOP");
    end;

    return FALSE;
end;
```

J1.2.3.83 ImmediateOp

```
// ImmediateOp
// ===========
// Vector logical immediate instruction types.

type ImmediateOp of enumeration {ImmediateOp_MOVI, ImmediateOp_MVNI,
                                 ImmediateOp_ORR, ImmediateOp_BIC};
```

J1.2.3.84 LogicalOp

```
// LogicalOp
// =========
// Logical instruction types.

type LogicalOp of enumeration {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

J1.2.3.85 mec

```
constant DEFAULT_MECID : bits(16) = Zeros{};
```

J1.2.3.86 AArch64_S1AMECFault

```
// AArch64_S1AMECFault()
// =====================
// Returns TRUE if a Translation fault should occur for Realm EL2 and Realm EL2&0
// stage 1 translated addresses to Realm PA space.

func AArch64_S1AMECFault{N}(walkparams : S1TTWParams, paspace : PASpace, regime : Regime,
                            descriptor : bits(N)) => boolean
begin
    assert N IN {64,128};
    let descriptor_amec : bit = (if walkparams.d128 == '1' then descriptor[108]
                                 else descriptor[63]);

    return (walkparams.[emec,amec] == '10' &&
            regime IN {Regime_EL2, Regime_EL20} &&
            paspace == PAS_Realm &&
            descriptor_amec == '1');
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.87 AArch64_S1DisabledOutputMECID

```
// AArch64_S1DisabledOutputMECID()
// ===============================
// Returns the output MECID when stage 1 address translation is disabled.

func AArch64_S1DisabledOutputMECID(walkparams : S1TTWParams, regime : Regime,
                                   paspace : PASpace) => bits(16)
begin
    if walkparams.emec == '0' then
        return DEFAULT_MECID;
    end;

    if ! regime IN {Regime_EL2, Regime_EL20, Regime_EL10} then
        return DEFAULT_MECID;
    end;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;
    end;

    if regime == Regime_EL10 then
        return VMECID_P_EL2().MECID;
    else
        return MECID_P0_EL2().MECID;
    end;
end;
```

J1.2.3.88 AArch64_S1OutputMECID

```
// AArch64_S1OutputMECID()
// =======================
// Returns the output MECID when stage 1 address translation is enabled.

func AArch64_S1OutputMECID{N}(walkparams : S1TTWParams, regime : Regime, varange : VARange,
                              paspace : PASpace, descriptor : bits(N)) => bits(16)
begin
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;
    end;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;
    end;

    let descriptor_amec : bit = (if walkparams.d128 == '1' then descriptor[108]
                                 else descriptor[63]);
    case regime of
        when Regime_EL3 =>
            return MECID_RL_A_EL3().MECID;
        when Regime_EL2 =>
            if descriptor_amec == '0' then
                return MECID_P0_EL2().MECID;
            else
                return MECID_A0_EL2().MECID;
            end;
        when Regime_EL20 =>
            if varange == VARange_LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2().MECID;
                else
                    return MECID_A0_EL2().MECID;
                end;
            else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                if descriptor_amec == '0' then
                    return MECID_P1_EL2().MECID;
                else
                    return MECID_A1_EL2().MECID;
                end;
            end;
        // Stage 2 translation might later override the MECID according to AMEC configuration.
        when Regime_EL10 =>
            return VMECID_P_EL2().MECID;
    end;
end;
```

J1.2.3.89 AArch64_S1TTWalkMECID

```
// AArch64_S1TTWalkMECID()
// =======================
// Returns the associated MECID for the stage 1 translation table walk of the given
// translation regime and Security state.

func AArch64_S1TTWalkMECID(emec : bit, regime : Regime, ss : SecurityState) => bits(16)
begin
    if emec == '0' then
        return DEFAULT_MECID;
    end;

    if ss != SS_Realm then
        return DEFAULT_MECID;
    end;

    case regime of
        when Regime_EL2 =>
            return MECID_P0_EL2().MECID;
        when Regime_EL20 =>
            if TCR_EL2().A1 == '0' then
                return MECID_P1_EL2().MECID;
            else
                return MECID_P0_EL2().MECID;
            end;
        // Stage 2 translation for a stage 1 walk might later override the
        // MECID according to AMEC configuration.
        when Regime_EL10 =>
            return VMECID_P_EL2().MECID;
        otherwise =>
            unreachable;
    end;
end;
```

J1.2.3.90 AArch64_S2OutputMECID

```
// AArch64_S2OutputMECID()
// =======================
// Returns the output MECID for stage 2 address translation.

func AArch64_S2OutputMECID{N}(walkparams : S2TTWParams, paspace : PASpace,
                              descriptor : bits(N)) => bits(16)
begin
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;
    end;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let descriptor_amec : bit = (if walkparams.d128 == '1' then descriptor[108]
                                 else descriptor[63]);
    if descriptor_amec == '0' then
        return VMECID_P_EL2().MECID;
    else
        return VMECID_A_EL2().MECID;
    end;
end;
```

J1.2.3.91 AArch64_S2TTWalkMECID

```
// AArch64_S2TTWalkMECID()
// =======================
// Returns the associated MECID for the stage 2 translation table walk of the
// given Security state.

func AArch64_S2TTWalkMECID(emec : bit, ss : SecurityState) => bits(16)
begin
    if emec == '0' then
        return DEFAULT_MECID;
    end;

    if ss != SS_Realm then
        return DEFAULT_MECID;
    end;

    //Stage 2 translation might later override the MECID according to AMEC configuration
    return VMECID_P_EL2().MECID;
end;
```

J1.2.3.92 AArch64_AccessIsTagChecked

```
// AArch64_AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

func AArch64_AccessIsTagChecked(vaddr : bits(64), accdesc : AccessDescriptor) => boolean
begin
    assert accdesc.tagchecked;

    if UsingAArch32() then
        return FALSE;
    end;

    if !IsMTEEnabled(accdesc.el) then
        return FALSE;
    end;

    if PSTATE.TCO == '1' then
        return FALSE;
    end;

    if (Halted() && EDSCR().MA == '1' &&
          ConstrainUnpredictableBool(Unpredictable_NODTRTAGCHK)) then
        return FALSE;
    end;

    if (IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && !accdesc.write &&
          StoreOnlyTagCheckingEnabled(accdesc.el)) then
        return FALSE;
    end;

    let is_instr : boolean = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
          EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;
    end;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
          (vaddr[59:55] == '00000' || vaddr[59:55] == '11111')) then
        return FALSE;
    end;

    return TRUE;
end;
```

J1.2.3.93 AArch64_AddressWithAllocationTag

```
// AArch64_AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.

func AArch64_AddressWithAllocationTag(address : bits(64), allocation_tag : bits(4)) => bits(64)
begin
    return address[63:60]::allocation_tag::address[55:0];
end;
```

J1.2.3.94 AArch64_AllocationTagCheck

```
// AArch64_AllocationTagCheck()
// ============================
// Performs an Allocation Tag Check operation for a memory access and
// returns whether the check passed.

func AArch64_AllocationTagCheck(memaddrdesc : AddressDescriptor, accdesc : AccessDescriptor,
                                ltag : bits(4)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, memaddrdesc.vaddress);
    if memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        var memstatus : PhysMemRetStatus;
        var readtag : bits(4);

        // Physical tagging needs no further translation, use the data PA to read the tag
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            let iswrite : boolean = FALSE;
            return ExternalFault(memstatus, iswrite, memaddrdesc, 1, accdesc);
        end;

        if ltag != readtag then
            fault.statuscode = Fault_TagCheck;
        end;
    end;

    return fault;
end;
```

J1.2.3.95 AArch64_AllocationTagFromAddress

```
// AArch64_AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

func AArch64_AllocationTagFromAddress(tagged_address : bits(64)) => bits(4)
begin
    return tagged_address[59:56];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.96 AArch64_CanonicalTagCheck

```
// AArch64_CanonicalTagCheck()
// ===========================
// Performs a Canonical Tag Check operation for a memory access and
// returns whether the check passed.

func AArch64_CanonicalTagCheck(memaddrdesc : AddressDescriptor, ltag : bits(4)) => boolean
begin
    let expected_tag : bits(4) = if memaddrdesc.vaddress[55] == '0' then '0000' else '1111';
    return ltag == expected_tag;
end;
```

J1.2.3.97 AArch64_CheckTag

```
// AArch64_CheckTag()
// ==================
// Performs a Tag Check operation for a memory access and returns a FaultRecord indicating if
// the check passed. If Tag Check Faults are asynchronously accumulated, a Tag Check Fault
// exception is recorded in TFSR_ELx.

func AArch64_CheckTag(memaddrdesc_in : AddressDescriptor, accdesc : AccessDescriptor,
                      readcheck : boolean, size : integer, ltag : bits(4)) => FaultRecord
begin
    var memaddrdesc : AddressDescriptor = memaddrdesc_in;
    // NoFault() will set fault.write to FALSE to accesses that perform both a read and a write.
    var fault : FaultRecord            = NoFault(accdesc, memaddrdesc.vaddress);
    let granules : integer             = Max(size DIVRM TAG_GRANULE, 1);
    let forcesync : boolean            = accdesc.nonfault || (accdesc.firstfault && !accdesc.first);
    let tcf : TCFType                  = AArch64_EffectiveTCF(accdesc.el, readcheck);

    for i = 0 to granules - 1 do
        case memaddrdesc.memattrs.tags of
            when MemTag_AllocationTagged =>
                fault = AArch64_AllocationTagCheck(memaddrdesc, accdesc, ltag);
                if fault.statuscode == Fault_TagCheck then
                    if tcf == TCFType_Sync || forcesync then
                        fault.statuscode = Fault_TagCheck;
                        return fault;
                    elsif tcf == TCFType_Async then
                        AArch64_ReportTagCheckFault(accdesc.el, memaddrdesc.vaddress[55]);
                        fault.statuscode = Fault_None;
                        return fault;
                    else
                        // Tag Check Faults have no effect on the PE.
                        fault.statuscode = Fault_None;
                    end;
                elsif fault.statuscode != Fault_None then
                    return fault;
                end;
            when MemTag_CanonicallyTagged =>
                if !AArch64_CanonicalTagCheck(memaddrdesc, ltag) then
                    if tcf == TCFType_Sync || forcesync then
                        fault.statuscode = Fault_TagCheck;
                        return fault;
                    elsif tcf == TCFType_Async then
                        AArch64_ReportTagCheckFault(accdesc.el, memaddrdesc.vaddress[55]);
                        return fault;
                    else
                        // Tag Check Faults have no effect on the PE.
                        pass;
                    end;
                end;
            when MemTag_Untagged =>
                pass;
            otherwise =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                unreachable;
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG_GRANULE;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + TAG_GRANULE;
    end;

    return fault;
end;
```

J1.2.3.98 AArch64_IsUnprivAccessPriv

```
// AArch64_IsUnprivAccessPriv()
// ============================
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

func AArch64_IsUnprivAccessPriv() => boolean
begin
    var privileged : boolean;

    case PSTATE.EL of
        when EL0 =>
            privileged = FALSE;
        when EL1 => privileged = EffectiveHCR_EL2_NVx()[1:0] == '11';
        when EL2 => privileged = !ELIsInHost(EL0);
        when EL3 =>
            privileged = TRUE;
    end;

    if IsFeatureImplemented(FEAT_UAO) && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;
    end;

    return privileged;
end;
```

J1.2.3.99 AArch64_LogicalAddressTag

```
// AArch64_LogicalAddressTag()
// ===========================
// Extract the Logical Address Tag from an address

func AArch64_LogicalAddressTag(vaddr : bits(64)) => bits(4)
begin
    return vaddr[59:56];
end;
```

J1.2.3.100 AArch64_MemSingle

```
// AArch64_MemSingle - accessor
// ============================

accessor AArch64_MemSingle{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                                     accdesc : AccessDescriptor,
                                                                     aligned : boolean
                                                                    ) <=> value : bits(size)
begin
    // Perform an atomic, little-endian read of 'size' bits.
    getter
        let bytes : integer{} = size DIV 8;
        var value : bits(size);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (value, memaddrdesc, memstatus) = AArch64_MemSingleRead{size}(address, accdesc, aligned);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch64_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, bytes, accdesc);
        end;

        return value;
    end;

    // Perform an atomic, little-endian write of 'size' bits.
    setter
        let bytes : integer{} = size DIV 8;
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (memaddrdesc, memstatus) = AArch64_MemSingleWrite{size}(address, accdesc, aligned, value);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch64_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;
end;
```

J1.2.3.101 AArch64_MemSingleRead

```
// AArch64_MemSingleRead()
// =======================
// Perform an atomic, little-endian read of 'size' bits.

func AArch64_MemSingleRead{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                                     accdesc_in : AccessDescriptor,
                                                                     aligned : boolean
                                                                    ) => (bits(size),
                                                                          AddressDescriptor,
                                                                          PhysMemRetStatus)
begin
    var value : bits(size) = ARBITRARY : bits(size);
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    let bytes : integer{} = size DIV 8;
    memstatus.statuscode = Fault_None;
    var accdesc : AccessDescriptor = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        let quantity : integer = MemSingleGranule();
        assert ((IsFeatureImplemented(FEAT_LS64WB) &&
                 bytes == 32 && accdesc.acctype == AccessType_ASIMD) ||
                 AllInAlignedQuantity{64}(address, bytes, quantity));
    else
        assert IsAlignedSize(address, bytes);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

var memaddrdesc : AddressDescriptor;
memaddrdesc = AArch64_TranslateAddress(address, accdesc, aligned, bytes);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (value, memaddrdesc, memstatus);
end;

// Memory array access
if accdesc.tagchecked then
    let ltag : bits(4) = AArch64_LogicalAddressTag(address);
    let readcheck : boolean = TRUE;
    let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
    if fault.statuscode != Fault_None then
        memaddrdesc.fault = fault;
        return (value, memaddrdesc, memstatus);
    end;
end;

if accdesc.acctype != AccessType_IFETCH && SPESampleInFlight then
    let is_load : boolean = TRUE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);
end;
var atomic : boolean;
if IsWBShareable(memaddrdesc.memattrs) then
    atomic = TRUE;
elsif accdesc.exclusive then
    atomic = TRUE;
elsif (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
        bytes == 8 && IsAlignedSize(address, 8)) then
    // An SVE predicated load of a 128-bit element that is 64-bit aligned
    // is treated as a pair of 64-bit single-copy atomic accesses.
    // This is one of the 64-bit single-copy atomic access.
    atomic = TRUE;
elsif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within MemSingleGranule() byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = ImpDefBool("FEAT_LSE2: access is atomic");
end;

if atomic then
    (memstatus, value) = PhysMemRead{size}(memaddrdesc, accdesc);

elsif accdesc.acctype == AccessType_ASIMD && bytes == 32 && accdesc.ispair then
    // A 32 byte LDP (SIMD&FP) that does not target Normal Inner Write-Back, Outer
    // Write-Back cacheable, Shareable memory is treated as four 8 byte atomic accesses.
    // As this access was not split in Mem(), it must be aligned to 32 bytes.
    assert IsAlignedSize(address, 32);
    accdesc.ispair = FALSE;
    for i = 0 to 3 do
        if !IsFault(memstatus) then     // Do not continue past a fault
            (memstatus, value[i*64+:64]) = PhysMemRead{64}(memaddrdesc, accdesc);
        end;

        if !IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 8;
        end;
    end;

elsif aligned && accdesc.ispair then
    let half : integer{} = (size DIV 2) as integer{32, 64, 128};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (memstatus, value[0*:half]) = PhysMemRead{half}(memaddrdesc, accdesc);

        if !IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + (half DIV 8);
            memaddrdesc.vaddress         = memaddrdesc.vaddress + (half DIV 8);
            (memstatus, value[1*:half]) = PhysMemRead{half}(memaddrdesc, accdesc);
        end;

    else
        for i = 0 to bytes-1 do
            if !IsFault(memstatus) then     // Do not continue past a fault
                (memstatus, value[i*:8]) = PhysMemRead{8}(memaddrdesc, accdesc);
            end;

            if !IsFault(memstatus) then
                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
                memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;
            end;

        end;
    end;

    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);
    end;

    if accdesc.acctype == AccessType_IFETCH then
        memaddrdesc.fault = AArch64_CheckDebug(address, accdesc, bytes);
    end;

    return (value, memaddrdesc, memstatus);
end;
```

J1.2.3.102 AArch64_MemSingleWrite

```
// AArch64_MemSingleWrite()
// ========================
// Perform an atomic, little-endian write of 'size' bits.

func AArch64_MemSingleWrite{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                                      accdesc_in : AccessDescriptor,
                                                                      aligned : boolean,
                                                                      value : bits(size)
                                                                  ) => (AddressDescriptor, PhysMemRetStatus)
begin
    var accdesc : AccessDescriptor = accdesc_in;
    let bytes : integer{} = size DIV 8;
    if IsFeatureImplemented(FEAT_LSE2) then
        let quantity : integer = MemSingleGranule();
        assert ((IsFeatureImplemented(FEAT_LS64WB) &&
                 bytes == 32 && accdesc.acctype == AccessType_ASIMD) ||
                 AllInAlignedQuantity{64}(address, bytes, quantity));
    else
        assert IsAlignedSize(address, bytes);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    memaddrdesc = AArch64_TranslateAddress(address, accdesc, aligned, bytes);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (memaddrdesc, memstatus);
end;

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
end;

if accdesc.tagchecked then
    let ltag : bits(4) = AArch64_LogicalAddressTag(address);
    let readcheck : boolean = FALSE;
    let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
    if fault.statuscode != Fault_None then
        memaddrdesc.fault = fault;
        return (memaddrdesc, memstatus);
    end;
end;

if SPESampleInFlight then
    let is_load : boolean = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);
end;

var atomic : boolean;
if IsWBShareable(memaddrdesc.memattrs) then
    atomic = TRUE;
elsif accdesc.exclusive then
    atomic = TRUE;
elsif (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
        bytes == 8 && IsAlignedSize(address, 8)) then
    // An SVE predicated load of a 128-bit element that is 64-bit aligned
    // is treated as a pair of 64-bit single-copy atomic accesses.
    // This is one of the 64-bit single-copy atomic access.
    atomic = TRUE;
elsif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within MemSingleGranule() byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = ImpDefBool("FEAT_LSE2: access is atomic");
end;
if atomic then
    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);
    end;

elsif accdesc.acctype == AccessType_ASIMD && bytes == 32 && accdesc.ispair then
    // A 32 byte STP (SIMD&FP) that does not target Normal Inner Write-Back, Outer
    // Write-Back cacheable, Shareable memory is treated as four 8 byte atomic accesses.
    // As this access was not split in Mem(), it must be aligned to 32 bytes.
    assert IsAlignedSize(address, 32);
    accdesc.ispair = FALSE;
    for i = 0 to 3 do
        memstatus = PhysMemWrite{64}(memaddrdesc, accdesc, value[64*i+:64]);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        end;
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + 8;
    end;

elsif aligned && accdesc.ispair then
    let half : integer{} = (size DIV 2) as integer{32, 64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memstatus = PhysMemWrite{half}(memaddrdesc, accdesc, value[0*:half]);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + (half DIV 8);
        memaddrdesc.vaddress         = memaddrdesc.vaddress + (half DIV 8);
        memstatus = PhysMemWrite{half}(memaddrdesc, accdesc, value[1*:half]);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        end;

    else
        for i = 0 to bytes-1 do
            memstatus = PhysMemWrite{8}(memaddrdesc, accdesc, value[i*:8]);
            if IsFault(memstatus) then
                return (memaddrdesc, memstatus);
            end;
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;
        end;
    end;

    return (memaddrdesc, memstatus);
end;
```

J1.2.3.103 AArch64_MemTag

```
// AArch64_MemTag - accessor
// ========================

accessor AArch64_MemTag(address : bits(64), accdesc : AccessDescriptor) <=> value : bits(4)
begin
    // Load an Allocation Tag from memory.
    getter
        var value : bits(4);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;
        (value, memaddrdesc, memstatus) = AArch64_MemTagRead(address, accdesc);

        if IsFault(memaddrdesc) then
            AArch64_Abort(memaddrdesc.fault);
        end;

        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        end;

        return value;
    end;

    // Store an Allocation Tag to memory.
    setter
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;
        (memaddrdesc, memstatus) = AArch64_MemTagWrite(address, accdesc, value);

        if IsFault(memaddrdesc) then
            AArch64_Abort(memaddrdesc.fault);
        end;

        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.104 AArch64_MemTagRead

```
// AArch64_MemTagRead()
// ====================
// Load an Allocation Tag from memory.

func AArch64_MemTagRead(address : bits(64),
                        accdesc_in : AccessDescriptor
                        ) => (bits(4), AddressDescriptor, PhysMemRetStatus)
begin
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    var accdesc : AccessDescriptor      = accdesc_in;
    var tag : bits(4)                   = ARBITRARY : bits(4);
    var memaddrdesc : AddressDescriptor = ARBITRARY : AddressDescriptor;
    var memtagtype : MemTagType         = ARBITRARY : MemTagType;
    var memstatus : PhysMemRetStatus    = ARBITRARY : PhysMemRetStatus;

    let aligned : boolean = TRUE;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64_TranslateTagAddress(address, accdesc, aligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (tag, memaddrdesc, memstatus);
    end;

    if accdesc.tagaccess && memtagtype == MemTag_AllocationTagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        return (tag, memaddrdesc, memstatus);
    elsif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
             accdesc.tagaccess && memtagtype == MemTag_CanonicallyTagged) then
        tag = if address[55] == '0' then '0000' else '1111';
        return (tag, memaddrdesc, memstatus);
    else
        // Otherwise read the tag as zero
        tag = '0000';
        return (tag, memaddrdesc, memstatus);
    end;
end;
```

J1.2.3.105 AArch64_MemTagWrite

```
// AArch64_MemTagWrite()
// =====================
// Store an Allocation Tag to memory.

func AArch64_MemTagWrite(address : bits(64),
                         accdesc_in : AccessDescriptor,
                         value : bits(4)) => (AddressDescriptor, PhysMemRetStatus)
begin
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    var accdesc : AccessDescriptor      = accdesc_in;
    var memaddrdesc : AddressDescriptor = ARBITRARY : AddressDescriptor;
    var memtagtype : MemTagType         = ARBITRARY : MemTagType;
    var memstatus : PhysMemRetStatus    = ARBITRARY : PhysMemRetStatus;

    let aligned : boolean = IsAlignedSize(address, TAG_GRANULE);

    // Stores of allocation tags must be aligned
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !aligned then
        memaddrdesc.fault = AlignmentFault(accdesc, address);
        return (memaddrdesc, memstatus);
    end;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64_TranslateTagAddress(address, accdesc, aligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);
    end;

    if accdesc.tagaccess && memtagtype == MemTag_AllocationTagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    end;
    return (memaddrdesc, memstatus);
end;
```

J1.2.3.106 AArch64_UnalignedAccessFaults

```
// AArch64_UnalignedAccessFaults()
// ==============================
// Determine whether the unaligned access generates an Alignment fault

func AArch64_UnalignedAccessFaults(accdesc : AccessDescriptor,
                                   address : bits(64),
                                   size : integer) => boolean
begin
    if AlignmentEnforced() then
        return TRUE;
    elsif accdesc.acctype == AccessType_GCS then
        return TRUE;
    elsif accdesc.rcw then
        return TRUE;
    elsif accdesc.ls64 then
        return TRUE;
    elsif (accdesc.exclusive || accdesc.atomicop)  then
        let quantity : integer = MemSingleGranule();
        return (!IsFeatureImplemented(FEAT_LSE2) ||
                  !AllInAlignedQuantity{64}(address, size, quantity));
    elsif (accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        // If nAA is 0, the address accessed by each register in the pair
        // must lie within a single 16-byte aligned block
        if accdesc.ispair then
            return  (SCTLR_ELx().nAA == '0' &&
                       (!AllInAlignedQuantity{64}(address, size DIV 2, 16) ||
                        !AllInAlignedQuantity{64}(address + size DIV 2, size DIV 2, 16)));
        else
            return  (SCTLR_ELx().nAA == '0' &&
                       !AllInAlignedQuantity{64}(address, size, 16));
        end;
    else
        return FALSE;
    end;
end;
```

J1.2.3.107 AddressSupportsLS64

```
// AddressSupportsLS64()
// =====================
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
impdef func AddressSupportsLS64(paddress : bits(NUM_PABITS)) => boolean
begin
    return ImpDefBool("Memory location supports ST64B and LD64B");
end;
```

J1.2.3.108 CASCompare

```
// CASCompare()
// ============
// Performs a comparison for CAS

func CASCompare{N}(oldvalue : bits(N),
                   comparevalue : bits(N),
                   newvalue : bits(N)) => (bits(N), boolean, bits(N))
begin
    var memresult : bits(N);
    var cmpfail : boolean;
    var regresult : bits(N) = oldvalue;

    if oldvalue == comparevalue then
        cmpfail = FALSE;
        memresult = newvalue;
        if !ConstrainUnpredictableBool(Unpredictable_CASRETURNOLDVALUE) then
            regresult = comparevalue;
        end;
    else
        cmpfail = TRUE;
        memresult = oldvalue;
    end;

    return (memresult, cmpfail, regresult);
end;
```

J1.2.3.109 CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

func CheckSPAlignment()
begin
    let sp : bits(64) = SP{}();
    var stack_align_check : boolean;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR_ELx().SA0 != '0');
    else
        stack_align_check = (SCTLR_ELx().SA != '0');
    end;

    if stack_align_check && sp != AlignDownSize(sp, 16) then
        AArch64_SPAlignmentFault();
    end;

    return;
end;
```

J1.2.3.110 IsConventionalMemory

```
// IsConventionalMemory()
// ======================
// Returns TRUE if the memory location is in Conventional memory, and FALSE otherwise.

impdef func IsConventionalMemory(addrdesc : AddressDescriptor) => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return TRUE;
end;
```

J1.2.3.111 Mem

```
// Mem - accessor
// ==============

accessor Mem{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                      accdesc_in : AccessDescriptor
                                                  ) <=>  value_in : bits(size)
begin
    // Perform a read of 'size' bits. The access byte order is reversed for a big-endian access.
    // Instruction fetches would call AArch64_MemSingle directly.
    getter
        var accdesc : AccessDescriptor = accdesc_in;
        let bytes : integer{} = size DIV 8;
        var value : bits(size);

        // Check alignment on size of element accessed, not overall access size
        var aligned : boolean;
        if accdesc.ispair && !accdesc.exclusive then
            let half : integer{} = (size DIV 2) as integer{32, 64, 128};
            aligned = IsAlignedSize(address, half DIV 8);
        else
            aligned = IsAlignedSize(address, bytes);
        end;

        let quantity : integer = MemSingleGranule();

        if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
            let fault : FaultRecord = AlignmentFault(accdesc, address);
            AArch64_Abort(fault);
        end;
        if accdesc.acctype == AccessType_ASIMD && bytes == 16 && IsAlignedSize(address, 8) then
            // If 128-bit SIMD&FP ordered access are treated as a pair of
            // 64-bit single-copy atomic accesses, then these single copy atomic
            // access can be observed in any order.
            let half : integer{} = (size DIV 2) as integer{64};
            let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
            value[0*:half] = AArch64_MemSingle{half}(address, accdesc, aligned);
            value[1*:half] = AArch64_MemSingle{half}(highaddress, accdesc, aligned);
        elsif (IsFeatureImplemented(FEAT_LS64WB) && accdesc.acctype == AccessType_ASIMD &&
                bytes == 32 && accdesc.ispair && IsAlignedSize(address, 32)) then
            value = AArch64_MemSingle{size}(address, accdesc, aligned);
        elsif accdesc.acctype == AccessType_ASIMD && bytes == 32 && IsAlignedSize(address, 8) then
            // If a 32 byte LDP (SIMD&FP) access is not aligned to 32 bytes but aligned to
            // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
            accdesc.ispair = FALSE;
            aligned        = TRUE;
            for i = 0 to 3 do
                let blockaddress : bits(64) = AddressIncrement(address, i*8, accdesc);
                value[64*i+:64] = AArch64_MemSingle{64}(blockaddress, accdesc, aligned);
            end;
        elsif (IsFeatureImplemented(FEAT_LSE2) &&
            AllInAlignedQuantity{64}(address, bytes, quantity)) then
            value = AArch64_MemSingle{size}(address, accdesc, aligned);
        elsif ((aligned && accdesc.ispair) ||
                  (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
                     bytes == 16 && IsAlignedSize(address, 8))) then
            // Either: an aligned pair access, OR
            // an SVE predicated load of a 128-bit element that is 64-bit aligned,
            // which is treated as two 64-bit single-copy atomic accesses.
            accdesc.ispair = FALSE;
            let half : integer{} = (size DIV 2) as integer{32, 64, 128};
            let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
                value[1*:half] = AArch64_MemSingle{half}(highaddress, accdesc, aligned);
                value[0*:half] = AArch64_MemSingle{half}(address, accdesc, aligned);
            else
                value[0*:half] = AArch64_MemSingle{half}(address, accdesc, aligned);
                value[1*:half] = AArch64_MemSingle{half}(highaddress, accdesc, aligned);
            end;
        elsif aligned then
            value = AArch64_MemSingle{size}(address, accdesc, aligned);
        else
            assert bytes > 1;
            if (IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair &&
                    accdesc.highestaddressfirst) then
                let half : integer{} = (size DIV 2) as integer{32, 64, 128};
                var lowhalf, highhalf : bits(half);

                for i = 0 to (half DIV 8)-1 do
                    let byteaddress : bits(64) = AddressIncrement(address,
                                                                  (half DIV 8) + i, accdesc);
                    // Individual byte access can be observed in any order
                    highhalf[i*:8] = AArch64_MemSingle{8}(byteaddress, accdesc, aligned);
                end;
                for i = 0 to (half DIV 8)-1 do
                    let byteaddress : bits(64) = AddressIncrement(address, i, accdesc);
                    // Individual byte access can be observed in any order
                    lowhalf[i*:8] = AArch64_MemSingle{8}(byteaddress, accdesc, aligned);
                end;

                value = highhalf::lowhalf;

            else
                value[7:0] = AArch64_MemSingle{8}(address, accdesc, aligned);
                accdesc.lowestaddress = FALSE;

                // For subsequent bytes, if they cross to a new translation page which assigns
                // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
                // will generate an Alignment Fault.
                let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
                assert c IN {Constraint_FAULT, Constraint_NONE};
                if c == Constraint_NONE then aligned = TRUE; end;

                for i = 1 to bytes-1 do
                    let byteaddress : bits(64) = AddressIncrement(address, i, accdesc);
                    value[i*:8] = AArch64_MemSingle{8}(byteaddress, accdesc, aligned);
                end;
            end;
        end;

        if BigEndian(accdesc.acctype) then
            value = BigEndianReverse{size}(value);
        end;

        return value;
    end;

    // Perform a write of 'size' bits. The byte order is reversed for a big-endian access.
    setter
        var value : bits(size) = value_in;
        let bytes : integer{} = size DIV 8;
        var accdesc : AccessDescriptor = accdesc_in;

        // Check alignment on size of element accessed, not overall access size
        var aligned : boolean;
        if accdesc.ispair && !accdesc.exclusive then
            let half : integer{} = (size DIV 2) as integer{32, 64, 128};
            aligned = IsAlignedSize(address, half DIV 8);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    aligned = IsAlignedSize(address, bytes);
end;
let quantity : integer = MemSingleGranule();

if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
    let fault : FaultRecord = AlignmentFault(accdesc, address);
    AArch64_Abort(fault);
end;

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse{size}(value);
end;
if accdesc.acctype == AccessType_ASIMD && bytes == 16 && IsAlignedSize(address, 8) then
    let half : integer{} = (size DIV 2) as integer{64};
    // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
    AArch64_MemSingle{half}(address, accdesc, aligned) = value[0+:half];
    AArch64_MemSingle{half}(highaddress, accdesc, aligned) = value[half+:half];
elsif (IsFeatureImplemented(FEAT_LS64WB) && accdesc.acctype == AccessType_ASIMD &&
        bytes == 32 && accdesc.ispair && IsAlignedSize(address, 32)) then
    AArch64_MemSingle{size}(address, accdesc, aligned) = value;
elsif accdesc.acctype == AccessType_ASIMD && bytes == 32 && IsAlignedSize(address, 8) then
    // If a 32 byte STP (SIMD&FP) access is not aligned to 32 bytes but aligned to
    // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
    accdesc.ispair = FALSE;
    aligned        = TRUE;
    for i = 0 to 3 do
        let blockaddress : bits(64) = AddressIncrement(address, i*8, accdesc);
        AArch64_MemSingle{64}(blockaddress, accdesc, aligned) = value[64*i+:64];
    end;
elsif (IsFeatureImplemented(FEAT_LSE2) &&
    AllInAlignedQuantity{64}(address, bytes, quantity)) then
    AArch64_MemSingle{size}(address, accdesc, aligned) = value;
elsif ((aligned && accdesc.ispair) ||
          (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
             bytes == 16 && IsAlignedSize(address, 8))) then
    // Either: an aligned pair access, OR
    // an SVE predicated load of a 128-bit element that is 64-bit aligned,
    // which is treated as two 64-bit single-copy atomic accesses.
    let half : integer{} = (size DIV 2) as integer{32, 64, 128};
    accdesc.ispair = FALSE;
    let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
    if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
        AArch64_MemSingle{half}(highaddress, accdesc, aligned) = value[half+:half];
        AArch64_MemSingle{half}(address, accdesc, aligned) = value[0+:half];
    else
        AArch64_MemSingle{half}(address, accdesc, aligned) = value[0+:half];
        AArch64_MemSingle{half}(highaddress, accdesc, aligned) = value[half+:half];
    end;
elsif aligned then
    AArch64_MemSingle{size}(address, accdesc, aligned) = value;
else
    assert bytes > 1;
    if (IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair &&
            accdesc.highestaddressfirst) then
        let half : integer{} = (size DIV 2) as integer{32, 64};
        var lowhalf, highhalf : bits(half);
        (highhalf, lowhalf) = (value[half+:half], value[0+:half]);
        for i = 0 to (half DIV 8)-1 do
            let byteaddress : bits(64) = AddressIncrement(address,
                                                          (half DIV 8) + i, accdesc);
            // Individual byte access can be observed in any order
            AArch64_MemSingle{8}(byteaddress, accdesc, aligned) = highhalf[i*:8];
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                for i = 0 to (half DIV 8)-1 do
                    let byteaddress : bits(64) = AddressIncrement(address,
                                                                  (half DIV 8) + i, accdesc);
                    // Individual byte access can be observed in any order,
                    // but implies observability of highhalf
                    AArch64_MemSingle{8}(byteaddress, accdesc, aligned) = lowhalf[i*:8];
                end;
            else
                AArch64_MemSingle{8}(address, accdesc, aligned) = value[7:0];
                accdesc.lowestaddress = FALSE;

                // For subsequent bytes, if they cross to a new translation page which assigns
                // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
                // will generate an Alignment Fault.

                let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
                assert c IN {Constraint_FAULT, Constraint_NONE};
                if c == Constraint_NONE then aligned = TRUE; end;

                for i = 1 to bytes-1 do
                    let byteaddress : bits(64) = AddressIncrement(address, i, accdesc);
                    AArch64_MemSingle{8}(byteaddress, accdesc, aligned) = value[i*:8];
                end;
            end;
        end;
    end;
end;
```

J1.2.3.112 MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

func MemAtomic{size : integer{8, 16, 32, 64, 128}}(address : bits(64),
                                                   cmpoperand : bits(size),
                                                   operand : bits(size),
                                                   accdesc_in : AccessDescriptor
                                                  ) => bits(size)
begin
    assert accdesc_in.atomicop;

    let bytes : integer{} = (size DIV 8) as integer{1, 2, 4, 8, 16};

    var newvalue : bits(size);
    var oldvalue : bits(size);
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
    end;

    // MMU or MPU lookup
    let memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc,
                                                                   aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64_Abort(memaddrdesc.fault);
end;

if (!IsWBShareable(memaddrdesc.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable_Atomic_NOP)) then
    return ARBITRARY : bits(size);
end;

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
end;

// For Store-only Tag checking, the tag check is performed on the store.
if (accdesc.tagchecked &&
      (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
       !StoreOnlyTagCheckingEnabled(accdesc.el))) then
    let ltag : bits(4) = AArch64_LogicalAddressTag(address);
    let readcheck : boolean = TRUE;
    let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
    if fault.statuscode != Fault_None then
        AArch64_Abort(fault);
    end;
end;

// All observers in the shareability domain observe the following load and store atomically.
var memstatus : PhysMemRetStatus;
(memstatus, oldvalue) = PhysMemRead{size}(memaddrdesc, accdesc);
// Depending on the memory type of the physical address, the access might generate
// either a synchronous External abort or an SError exception
// among other CONSTRAINED UNPREDICTABLE choices.

if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
end;
if BigEndian(accdesc.acctype) then
    oldvalue = BigEndianReverse{size}(oldvalue);
end;

var cmpfail : boolean = FALSE;
var retvalue : bits(size) = oldvalue;
if accdesc.acctype == AccessType_FP then
    newvalue = MemAtomicFP{size}(accdesc.modop, oldvalue, operand);
else
    (newvalue, cmpfail, retvalue) = MemAtomicInt{size}(accdesc.modop, oldvalue,
                                                       operand, cmpoperand);
end;

let requirewrite : boolean = (!cmpfail ||
                              ConstrainUnpredictableBool(Unpredictable_WRITEFAILEDCAS));

if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckingEnabled(accdesc.el) then
    // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the
    // Tag check is performed.
    if accdesc.tagchecked && !requirewrite then
        accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable_STRONLYTAGCHECKEDCAS);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        var fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck,
                                                   bytes, ltag);
        if fault.statuscode != Fault_None then
            // For a synchronous Tag Check Fault due to FEAT_MTE_STORE_ONLY, set WnR.
            fault.write = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_Abort(fault);
            end;
        end;
    end;

    if requirewrite then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse{size}(newvalue);
        end;
        memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    // Load operations return the old (pre-operation) value.
    // Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
    // this might be the value from the compare operand or from memory.
    return retvalue;
end;
```

J1.2.3.113 MemAtomicFP

```
// MemAtomicFP()
// =============
// Performs FP Atomic operation

func MemAtomicFP{N}(modop : MemAtomicOp, op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    var fpcr : FPCR_Type = FPCR();
    let altfp : boolean  = FALSE;
    let fpexc : boolean  = FALSE;
    fpcr.[AH,DN]   = '01';
    fpcr.FZ        = fpcr.FZ OR fpcr.FIZ; // Treat FPCR.FIZ as equivalent to FPCR.FZ
    var result : bits(N);

    case modop of
        when MemAtomicOp_FPADD =>    result = FPAdd{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp_FPMAX =>    result = FPMax{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_FPMIN =>    result = FPMin{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_FPMAXNM =>  result = FPMaxNum{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp_FPMINNM =>  result = FPMinNum{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp_BFADD =>    result = BFAdd{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp_BFMAX =>    result = BFMax{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_BFMIN =>    result = BFMin{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_BFMAXNM =>  result = BFMaxNum{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp_BFMINNM =>  result = BFMinNum{N}(op1, op2, fpcr, fpexc);
    end;
    return result;
end;
```

J1.2.3.114 MemAtomicInt

```
// MemAtomicInt()
// ==============
// Performs Integer Atomic operation

func MemAtomicInt{N}(modop : MemAtomicOp, op1 : bits(N), op2 : bits(N), cmpop : bits(N)
                     ) => (bits(N), boolean, bits(N))
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var result : bits(N);
    var cmpfail : boolean = FALSE;
    var retvalue : bits(N) = op1;

    case modop of
        when MemAtomicOp_ADD =>      result = op1 + op2;
        when MemAtomicOp_BIC =>      result = op1 AND NOT(op2);
        when MemAtomicOp_EOR =>      result = op1 XOR op2;
        when MemAtomicOp_ORR =>      result = op1 OR op2;
        when MemAtomicOp_SMAX =>     result = Max(SInt(op1), SInt(op2))[N-1:0];
        when MemAtomicOp_SMIN =>     result = Min(SInt(op1), SInt(op2))[N-1:0];
        when MemAtomicOp_UMAX =>     result = Max(UInt(op1), UInt(op2))[N-1:0];
        when MemAtomicOp_UMIN =>     result = Min(UInt(op1), UInt(op2))[N-1:0];
        when MemAtomicOp_SWP =>      result = op2;
        when MemAtomicOp_CAS =>      (result, cmpfail, retvalue) = CASCompare{N}(op1, cmpop, op2);
        when MemAtomicOp_GCSSS1 =>   (result, cmpfail, retvalue) = CASCompare{N}(op1, cmpop, op2);
    end;
    return (result, cmpfail, retvalue);
end;
```

J1.2.3.115 MemAtomicRCW

```
// MemAtomicRCW()
// ==============
// Perform a single-copy-atomic access with Read-Check-Write operation

func MemAtomicRCW{size : integer{64, 128}}(address : bits(64), cmpoperand : bits(size),
                                            operand : bits(size), accdesc_in : AccessDescriptor
                                  ) => (bits(4), bits(size))
begin
    assert accdesc_in.atomicop;
    assert accdesc_in.rcw;

    let bytes : integer{} = (size DIV 8) as integer{8, 16};

    var nzcv : bits(4);
    var oldvalue : bits(size);
    var newvalue : bits(size);
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
    end;

    // MMU or MPU lookup
    let memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc,
                                                                   aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    if (!IsWBShareable(memaddrdesc.memattrs) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_NOP)) then
        return (ARBITRARY : bits(4), ARBITRARY : bits(size));
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Effect on exclusives
if memaddrdesc.mematrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
end;

// For Store-only Tag checking, the tag check is performed on the store.
if (accdesc.tagchecked &&
      (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
       !StoreOnlyTagCheckingEnabled(accdesc.el))) then
    let ltag : bits(4) = AArch64_LogicalAddressTag(address);
    let readcheck : boolean = TRUE;
    let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
    if fault.statuscode != Fault_None then
        AArch64_Abort(fault);
    end;
end;

// All observers in the shareability domain observe the following load and store atomically.
var memstatus : PhysMemRetStatus;
(memstatus, oldvalue) = PhysMemRead{size}(memaddrdesc, accdesc);
// Depending on the memory type of the physical address, the access might generate
// either a synchronous External abort or an SError exception
// among other CONSTRAINED UNPREDICTABLE choices.

if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
end;
if BigEndian(accdesc.acctype) then
    oldvalue = BigEndianReverse{size}(oldvalue);
end;

var cmpfail : boolean = FALSE;
var retvalue : bits(size) = oldvalue;
case accdesc.modop of
    when MemAtomicOp_BIC => newvalue = oldvalue AND NOT(operand);
    when MemAtomicOp_ORR => newvalue = oldvalue OR operand;
    when MemAtomicOp_SWP => newvalue = operand;
    when MemAtomicOp_CAS =>
        (newvalue, cmpfail, retvalue) = CASCompare{size}(oldvalue, cmpoperand, operand);
end;

if cmpfail then
    nzcv = '1010'; // N = 1 indicates compare failure
else
    nzcv = RCWCheck{size}(retvalue, newvalue, accdesc.rcws);
end;

// If RCWCheck() passes, it returns nzcv == '0010'
let requirewrite : boolean = (nzcv == '0010' ||
                              ConstrainUnpredictableBool(Unpredictable_WRITEFAILEDRCWCAS));

if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckingEnabled(accdesc.el) then
    // If the RCW(S) check fails, or if the compare on a RCW(S)CAS fails,
    // then it is CONSTRAINED UNPREDICTABLE whether the Tag check is performed.
    if accdesc.tagchecked && !requirewrite then
        accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable_STRONLYTAGCHECKEDRCWSCAS);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        var fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck,
                                                   bytes, ltag);
        if fault.statuscode != Fault_None then
            // For a synchronous Tag Check Fault due to FEAT_MTE_STORE_ONLY, set WnR.
            fault.write = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch64_Abort(fault);
            end;
        end;
    end;

    if requirewrite then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse{size}(newvalue);
        end;

        memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, newvalue);

        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    // Load operations return the old (pre-operation) value.
    // Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
    // this might be the value from the compare operand or from memory.
    return (nzcv, retvalue);
end;
```

J1.2.3.116 MemLoad64B

```
// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

func MemLoad64B(address : bits(64), accdesc_in : AccessDescriptor) => bits(512)
begin
    let size : integer{} = 512;
    let bytes : integer{} = size DIV 8;
    var data : bits(size);
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc,
                                                                   aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    // Effect on exclusives
    if memaddrdesc.mematrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if accdesc.tagchecked then
        let ltag : bits(4)      = AArch64_LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault_None then
            AArch64_Abort(fault);
        end;
    end;

    var byte_atomic : boolean = FALSE;
    if ((memaddrdesc.memattrs.memtype == MemType_Device ||
          (memaddrdesc.memattrs.inner.attrs == MemAttr_NC &&
             memaddrdesc.memattrs.outer.attrs == MemAttr_NC)) &&
          !AddressSupportsLS64(memaddrdesc.paddress.address)) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64_Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    elsif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if ImpDefBool("LD64B faults to iWBoWB non-Conventional memory") then
            // Generate a Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64_Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    var memstatus : PhysMemRetStatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to bytes-1 do
            (memstatus, data[i*:8]) = PhysMemRead{8}(memaddrdesc, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            end;
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;
        end;
    else
        (memstatus, data) = PhysMemRead{size}(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;

    return data;
end;
```

J1.2.3.117 MemSingleGranule

```
// MemSingleGranule()
// ==================
// When FEAT_LSE2 is implemented, for some memory accesses if all bytes
// of the accesses are within 16-byte quantity aligned to 16-bytes and
// satisfy additional requirements - then the access is guaranteed to
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// be single copy atomic.
// However, when the accesses do not all lie within such a boundary, it
// is CONSTRAINED UNPREDICTABLE if the access is single copy atomic.
// In the pseudocode, this CONSTRAINED UNPREDICTABLE aspect is modeled via
// MemSingleGranule() which is IMPLEMENTATION DEFINED and, is at least 16 bytes
// and at most 4096 bytes.
// This is a limitation of the pseudocode.

func MemSingleGranule() => integer
begin
    let size : integer = ImpDefInt("Aligned quantity for atomic access");
    // access is assumed to be within 4096 byte aligned quantity to
    // avoid multiple translations for a single copy atomic access.
    assert (size >= 16) && (size <= 4096);
    return size;
end;
```

J1.2.3.118 MemStore64B

```
// MemStore64B()
// =============
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

func MemStore64B(address : bits(64), value : bits(512), accdesc_in : AccessDescriptor)
begin
    let size : integer{} = 512;
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc,
                                                                   aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4)      = AArch64_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault_None then
            AArch64_Abort(fault);
        end;
    end;

    var byte_atomic : boolean = FALSE;
    if ((memaddrdesc.memattrs.memtype == MemType_Device ||
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (memaddrdesc.memattrs.inner.attrs == MemAttr_NC &&
               memaddrdesc.memattrs.outer.attrs == MemAttr_NC)) &&
            !AddressSupportsLS64(memaddrdesc.paddress.address)) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64_Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    elsif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if ImpDefBool("ST64B faults to iWBoWB non-Conventional memory") then
            // Generate a Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64_Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    var memstatus : PhysMemRetStatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to bytes-1 do
            memstatus = PhysMemWrite{8}(memaddrdesc, accdesc, value[i*:8]);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            end;
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;
        end;
    else
        memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;

    return;
end;
```

J1.2.3.119 MemStore64BWithRet

```
// MemStore64BWithRet()
// ====================
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

func MemStore64BWithRet(address : bits(64), value : bits(512),
                        accdesc_in : AccessDescriptor) => bits(64)
begin
    let size : integer{} = 512;
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    if !aligned && AArch64_UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64_Abort(fault);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    let memaddrdesc : AddressDescriptor = AArch64_TranslateAddress(address, accdesc,
                                                                   aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64_Abort(memaddrdesc.fault);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4)      = AArch64_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        let fault : FaultRecord = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault_None then
            AArch64_Abort(fault);
        end;
    end;

    var memstatus : PhysMemRetStatus;
    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);

    // If an access generated by ST64BV or ST64BV0 fails solely because the memory location
    // does not support 64-byte access, then memstatus does not indicate a fault,
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;

    if SPESampleInFlight then
        let is_load : boolean = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    return memstatus.store64bstatus;
end;
```

J1.2.3.120 MemStore64BWithRetStatus

```
// MemStore64BWithRetStatus()
// ==========================
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.

impdef func MemStore64BWithRetStatus() => bits(64)
begin
    return Zeros{64};
end;
```

J1.2.3.121 NVMem

```
// NVMem - accessor
// ================
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
accessor NVMem(offset : integer) <=> value : bits(64)
begin
    // This function is the load memory access for the transformed System register read access
    // when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
    // The address for the load memory access is calculated using
    // the formula SignExtend{64}(VNCR_EL2.BADDR :: Offset[11:0]) where,
    //  * VNCR_EL2.BADDR holds the base address of the memory location, and
    //  * Offset is the unique offset value defined architecturally for each System register that
    //    supports transformation of register access to memory access.
    getter
        assert offset > 0 && offset MOD 8 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64)     = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64)   = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_LOAD);
        return Mem{64}(address, accdesc);
    end;

    // This function is the store memory access for the transformed System register write access
    // when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
    // The address for the store memory access is calculated using
    // the formula SignExtend{64}(VNCR_EL2.BADDR :: Offset[11:0]) where,
    //  * VNCR_EL2.BADDR holds the base address of the memory location, and
    //  * Offset is the unique offset value defined architecturally for each System register that
    //    supports transformation of register access to memory access.
    setter
        assert offset > 0 && offset MOD 8 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64)     = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64)   = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_STORE);
        Mem{64}(address, accdesc) = value;
        return;
    end;
end;

accessor NVMem128(offset : integer) <=> value : bits(128)
begin
    getter
        assert offset > 0 && offset MOD 16 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64)     = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64)   = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_LOAD);
        return Mem{128}(address, accdesc);
    end;

    setter
        assert offset > 0 && offset MOD 16 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64)     = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64)   = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_STORE);
        Mem{128}(address, accdesc) = value;
    end;
end;
```

J1.2.3.122 PhysMemTagRead

```
// PhysMemTagRead()
// ================
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

impdef func PhysMemTagRead(desc : AddressDescriptor,
                           accdesc : AccessDescriptor) => (PhysMemRetStatus, bits(4))
begin
    return (ARBITRARY : PhysMemRetStatus, Zeros{4});
end;
```

J1.2.3.123 PhysMemTagWrite

```
// PhysMemTagWrite()
// =================
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

impdef func PhysMemTagWrite(desc : AddressDescriptor,
                            accdesc : AccessDescriptor,
                            value : bits(4)) => PhysMemRetStatus
begin
    return ARBITRARY : PhysMemRetStatus;
end;
```

J1.2.3.124 StoreOnlyTagCheckingEnabled

```
// StoreOnlyTagCheckingEnabled()
// =============================
// Returns TRUE if loads executed at the given Exception level are Tag unchecked.

func StoreOnlyTagCheckingEnabled(el : bits(2)) => boolean
begin
    assert IsFeatureImplemented(FEAT_MTE_STORE_ONLY);
    var tcso : bit;

    case el of
        when EL0 =>
            if !ELIsInHost(el) then
                tcso = SCTLR_EL1().TCSO0;
            else
                tcso = SCTLR_EL2().TCSO0;
            end;
        when EL1 =>
            tcso = SCTLR_EL1().TCSO;
        when EL2 =>
            tcso = SCTLR_EL2().TCSO;
        otherwise =>
            tcso = SCTLR_EL3().TCSO;
    end;

    return tcso == '1';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.125 ArchMaxMOPSBlockSize

```
// ArchMaxMOPSBlockSize
// ====================
// Maximum number of bytes CPY/SET instructions can use

constant ArchMaxMOPSBlockSize : integer{} = 0x7FFF_FFFF_FFFF_FFFF;
```

J1.2.3.126 ArchMaxMOPSCPYSize

```
// ArchMaxMOPSCPYSize
// ==================
// Maximum number of bytes CPY instructions can use

constant ArchMaxMOPSCPYSize : integer{} = 0x007F_FFFF_FFFF_FFFF;
```

J1.2.3.127 ArchMaxMOPSSETGSize

```
// ArchMaxMOPSSETGSize
// ===================
// Maximum number of bytes SETG instructions can use

constant ArchMaxMOPSSETGSize : integer{} = 0x7FFF_FFFF_FFFF_FFF0;
```

J1.2.3.128 CPYFOptionA

```
// CPYFOptionA()
// =============
// Returns TRUE if the implementation uses Option A for the
// CPYF* instructions, and FALSE otherwise.

func CPYFOptionA() => boolean
begin
    return ImpDefBool("CPYF* instructions use Option A");
end;
```

J1.2.3.129 CPYOptionA

```
// CPYOptionA()
// ============
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

func CPYOptionA() => boolean
begin
    return ImpDefBool("CPY* instructions use Option A");
end;
```

J1.2.3.130 CPYParams

```
// CPYParams
// =========

type CPYParams of record {
    stage : MOPSStage,
    implements_option_a : boolean,
    forward : boolean,
    cpysize : integer,
    stagecpysize : integer,
    toaddress : bits(64),
    fromaddress : bits(64),
    nzcv : bits(4),
    n : integer,
    d : integer,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    s : integer
};
```

J1.2.3.131 CPYPostSizeChoice

```
// CPYPostSizeChoice()
// ===================
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.

impdef func CPYPostSizeChoice(memcpy : CPYParams) => integer
begin
    return ARBITRARY : integer;
end;
```

J1.2.3.132 CPYPreSizeChoice

```
// CPYPreSizeChoice()
// ==================
// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.

impdef func CPYPreSizeChoice(memcpy : CPYParams) => integer
begin
    return ARBITRARY : integer;
end;
```

J1.2.3.133 CPYSizeChoice

```
// CPYSizeChoice()
// ===============
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.

impdef func CPYSizeChoice(memcpy : CPYParams) => MOPSBlockSize
begin
    return ARBITRARY : MOPSBlockSize;
end;
```

J1.2.3.134 CheckCPYConstrainedUnpredictable

```
// CheckCPYConstrainedUnpredictable()
// ==================================
// Check for CONSTRAINED UNPREDICTABLE behavior in the CPY* and CPYF* instructions.

func CheckCPYConstrainedUnpredictable(n : integer, d : integer, s : integer)
begin
    if (s == n || s == d || n == d) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF =>      Undefined();
            when Constraint_NOP =>        ExecuteAsNOP();
        end;
    end;

    if (d == 31 || s == 31 || n == 31) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_MOPS_R31);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF =>      Undefined();
            when Constraint_NOP =>        ExecuteAsNOP();
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.135 CheckMOPSEnabled

```
// CheckMOPSEnabled()
// ==================
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

func CheckMOPSEnabled()
begin
    if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2().MSCEn == '0')) then
        Undefined();
    end;

    if PSTATE.EL == EL0 && !IsInHost() && SCTLR_EL1().MSCEn == '0' then
        Undefined();
    end;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2().MSCEn == '0' then
        Undefined();
    end;
end;
```

J1.2.3.136 CheckMemCpyParams

```
// CheckMemCpyParams()
// ===================
// Check if the parameters to a CPY* or CPYF* instruction are consistent with the
// PE state and well-formed.

func CheckMemCpyParams(memcpy : CPYParams, options : bits(4))
begin
    let from_epilogue : boolean = memcpy.stage == MOPSStage_Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memcpy.stagecpysize != 0 || MemStageCpyZeroSizeCheck()) &&
          (memcpy.cpysize != 0 || MemCpyZeroSizeCheck())) then
        let using_option_a : boolean = memcpy.nzcv[1] == '0';
        if memcpy.implements_option_a != using_option_a then
            let formatoption : bits(2) = '1':: (if memcpy.implements_option_a then '1' else '0');
            MismatchedMemCpyException(memcpy, options, formatoption);
        end;
    end;

    // Check if the parameters to this instruction are valid.
    if memcpy.stage == MOPSStage_Main then
        if MemCpyParametersIllformedM(memcpy) then
            let formatoption : bits(2) = '0':: (if memcpy.implements_option_a then '1' else '0');
            MismatchedMemCpyException(memcpy, options, formatoption);
        end;
    else
        let postsize : integer = CPYPostSizeChoice(memcpy);
        if memcpy.cpysize != postsize || MemCpyParametersIllformedE(memcpy) then
            let formatoption : bits(2) = '0':: (if memcpy.implements_option_a then '1' else '0');
            MismatchedMemCpyException(memcpy, options, formatoption);
        end;
    end;

    return;
end;
```

J1.2.3.137 CheckMemSetParams

```
// CheckMemSetParams()
// ===================
// Check if the parameters to a SET* or SETG* instruction are consistent with the
// PE state and well-formed.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func CheckMemSetParams(memset : SETParams, options : bits(2))
begin
    let from_epilogue : boolean = memset.stage == MOPSStage_Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          (memset.setsize != 0 || MemSetZeroSizeCheck())) then
        let using_option_a : boolean = memset.nzcv[1] == '0';
        if memset.implements_option_a != using_option_a then
            let formatoption : bits(2) = '1':: (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
        end;
    end;

    // Check if the parameters to this instruction are valid.
    if memset.stage == MOPSStage_Main then
        if MemSetParametersIllformedM(memset) then
            let formatoption : bits(2) = '0':: (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
        end;
    else
        let postsize : integer = SETPostSizeChoice(memset);
        if memset.setsize != postsize || MemSetParametersIllformedE(memset) then
            let formatoption : bits(2) = '0':: (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
        end;
    end;

    return;
end;
```

J1.2.3.138 CheckSETConstrainedUnpredictable

```
// CheckSETConstrainedUnpredictable()
// ==================================
// Check for CONSTRAINED UNPREDICTABLE behavior in the SET* and SETG* instructions.

func CheckSETConstrainedUnpredictable(memset : SETParams)
begin
    if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF => Undefined();
            when Constraint_NOP   => ExecuteAsNOP();
        end;
    end;

    if (memset.d == 31 || memset.n == 31) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_MOPS_R31);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF => Undefined();
            when Constraint_NOP   => ExecuteAsNOP();
        end;
    end;
end;
```

J1.2.3.139 IsMemCpyForward

```
// IsMemCpyForward()
// =================
// Returns TRUE if in a memcpy of size cpysize bytes from the source address fromaddress
// to destination address toaddress is done in the forward direction on this implementation.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func IsMemCpyForward(memcpy : CPYParams) => boolean
begin
    var forward : boolean;

    // Check for overlapping cases
    if ((UInt(memcpy.fromaddress[55:0]) > UInt(memcpy.toaddress[55:0])) &&
          (UInt(memcpy.fromaddress[55:0]) < UInt(ZeroExtend{64}(memcpy.toaddress[55:0]) +
             memcpy.cpysize))) then
        forward = TRUE;

    elsif ((UInt(memcpy.fromaddress[55:0]) < UInt(memcpy.toaddress[55:0])) &&
             (UInt(ZeroExtend{64}(memcpy.fromaddress[55:0]) + memcpy.cpysize) >
                UInt(memcpy.toaddress[55:0]))) then
        forward = FALSE;

    // Non-overlapping case
    else
        forward = ImpDefBool("CPY in the forward direction");
    end;

    return forward;
end;
```

J1.2.3.140 MOPSBlockSize

```
// MOPSBlockSize
// ================

type MOPSBlockSize of integer{0..MaxMOPSBlockSize};
```

J1.2.3.141 MOPSStage

```
// MOPSStage
// =========

type MOPSStage of enumeration { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };
```

J1.2.3.142 MaxBlockSizeCopiedBytes

```
// MaxBlockSizeCopiedBytes()
// =========================
// Returns the maximum number of bytes that can used in a single block of the copy.

func MaxBlockSizeCopiedBytes() => integer
begin
    return ImpDefInt("Maximum bytes used in a single block of a copy");
end;
```

J1.2.3.143 MaxMOPSBlockSize

```
// MaxMOPSBlockSize
// ================
// Maximum number of bytes CPY/SET instructions can use

config MaxMOPSBlockSize : integer{1..ArchMaxMOPSBlockSize} = 64;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.144 MemCpyBytes

```
// MemCpyBytes()
// =============
// Copies 'bytes' bytes of memory from fromaddress to toaddress.
// The integer return parameter indicates the number of bytes copied. The boolean return parameter
// indicates if a Fault or Abort occurred on the write. The AddressDescriptor and PhysMemRetStatus
// parameters contain Fault or Abort information for the caller to handle.

func MemCpyBytes(toaddress : bits(64),
                 fromaddress : bits(64),
                 forward : boolean,
                 bytes : MOPSBlockSize,
                 raccdesc : AccessDescriptor,
                 waccdesc : AccessDescriptor
                ) => (integer, boolean, AddressDescriptor, PhysMemRetStatus)
begin
    var rmemaddrdesc : AddressDescriptor;                 // AddressDescriptor for reads
    var rmemstatus : PhysMemRetStatus;                   // PhysMemRetStatus  for writes
    rmemaddrdesc.fault    = NoFault();
    rmemstatus.statuscode = Fault_None;

    var wmemaddrdesc : AddressDescriptor;                 // AddressDescriptor for writes
    var wmemstatus : PhysMemRetStatus;                   // PhysMemRetStatus  for writes
    wmemaddrdesc.fault    = NoFault();
    wmemstatus.statuscode = Fault_None;

    var value : bits(8*bytes);
    let aligned : boolean = TRUE;

    if forward then
        var read : integer  = 0;                          // Bytes read
        var write : integer = 0;                          // Bytes written

        // Read until all bytes are read or until a fault is encountered.
        while (read < bytes && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus))
                                        looplimit MaxMOPSBlockSize do
            (value[8 * read +:8], rmemaddrdesc, rmemstatus) = AArch64_MemSingleRead{8}(
                                                                    fromaddress + read,
                                                                    raccdesc, aligned);
            read = read + 1;
        end;

        // Ensure no UNKNOWN data is written.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            read = read - 1;
        end;

        // Write all bytes that were read or until a fault is encountered.
        while write < read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus)
                                    looplimit MaxMOPSBlockSize  do
            (wmemaddrdesc, wmemstatus) = AArch64_MemSingleWrite{8}(toaddress + write,
                                                        waccdesc, aligned,
                                                        value[8 * write +:8]);
            write = write + 1;
        end;

        // Check all bytes were written.
        if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
            let fault_on_write : boolean = TRUE;
            return (write - 1, fault_on_write, wmemaddrdesc, wmemstatus);
        end;

        // Check all bytes were read.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            let fault_on_write : boolean = FALSE;
            return (read, fault_on_write, rmemaddrdesc, rmemstatus);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;

    else
        var read : integer  = bytes;                        // Bytes to read
        var write : integer = bytes;                        // Bytes to write

        // Read until all bytes are read or until a fault is encountered.
        while read > 0 && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus)
                                          looplimit MaxMOPSBlockSize do
            read = read - 1;
            (value[8 * read +:8], rmemaddrdesc, rmemstatus) = AArch64_MemSingleRead{8}(
                                                                        fromaddress + read,
                                                                        raccdesc, aligned);
        end;

        // Ensure no UNKNOWN data is written.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            read = read + 1;
        end;

        // Write all bytes that were read or until a fault is encountered.
        while write > read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus)
                                             looplimit MaxMOPSBlockSize do
            write = write - 1;
            (wmemaddrdesc, wmemstatus) = AArch64_MemSingleWrite{8}(toaddress + write,
                                                            waccdesc, aligned,
                                                            value[8 * write +:8]);
        end;

        // Check all bytes were written.
        if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
            let fault_on_write : boolean = TRUE;
            return (bytes - (write + 1), fault_on_write, wmemaddrdesc, wmemstatus);
        end;

        // Check all bytes were read.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            let fault_on_write : boolean = FALSE;
            return (bytes - read, fault_on_write, rmemaddrdesc, rmemstatus);
        end;
    end;

    // Return any AddressDescriptor and PhysMemRetStatus.
    return (bytes, FALSE, wmemaddrdesc, wmemstatus);
end;
```

J1.2.3.145 MemCpyParametersIllformedE

```
// MemCpyParametersIllformedE()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

impdef func MemCpyParametersIllformedE(memcpy : CPYParams) => boolean
begin
    return FALSE;
end;
```

J1.2.3.146 MemCpyParametersIllformedM

```
// MemCpyParametersIllformedM()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// source and size of the copy.

impdef func MemCpyParametersIllformedM(memcy : CPYParams) => boolean
begin
    return FALSE;
end;
```

J1.2.3.147 MemCpyStageSize

```
// MemCpyStageSize()
// =================
// Returns the number of bytes copied by the given stage of a CPY* or CPYF* instruction.

func MemCpyStageSize(memcy : CPYParams) => integer
begin
    var stagecpysize : integer;

    if memcy.stage == MOPSStage_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(memcy);
        assert stagecpysize == 0 || (stagecpysize < 0) == (memcy.cpysize < 0);

        if memcy.cpysize > 0 then
            assert stagecpysize <= memcy.cpysize;
        else
            assert stagecpysize >= memcy.cpysize;
        end;

    else
        let postsize : integer = CPYPostSizeChoice(memcy);
        assert postsize == 0 || (postsize < 0) == (memcy.cpysize < 0);

        if memcy.stage == MOPSStage_Main then
            stagecpysize = memcy.cpysize - postsize;
        else
            stagecpysize = postsize;
        end;
    end;

    return stagecpysize;
end;
```

J1.2.3.148 MemCpyZeroSizeCheck

```
// MemCpyZeroSizeCheck()
// =====================
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

func MemCpyZeroSizeCheck() => boolean
begin
    return ImpDefBool("Implementation option is checked with a cpysize of 0");
end;
```

J1.2.3.149 MemSetBytes

```
// MemSetBytes()
// =============
// Writes a byte of data to the given address 'bytes' times.
// The integer return parameter indicates the number of bytes set. The AddressDescriptor and
// PhysMemRetStatus parameters contain Fault or Abort information for the caller to handle.

func MemSetBytes(toaddress : bits(64), data : bits(8),
                 bytes : MOPSBlockSize,
                 accdesc : AccessDescriptor) => (integer, AddressDescriptor, PhysMemRetStatus)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    memaddrdesc.fault    = NoFault();
    memstatus.statuscode = Fault_None;

    let aligned : boolean = TRUE;
    var write : integer   = 0;                            // Bytes written

    // Write until all bytes are written or a fault is encountered.
    while write < bytes && !IsFault(memaddrdesc) && !IsFault(memstatus)
                                                  looplimit MaxMOPSBlockSize do
        (memaddrdesc, memstatus) = AArch64_MemSingleWrite{8}(toaddress + write, accdesc,
                                                                aligned, data);
        write = write + 1;
    end;

    // Check all bytes were written.
    if IsFault(memaddrdesc) || IsFault(memstatus) then
        return (write - 1, memaddrdesc, memstatus);
    end;

    return (bytes, memaddrdesc, memstatus);
end;
```

J1.2.3.150 MemSetParametersIllformedE

```
// MemSetParametersIllformedE()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.

impdef func MemSetParametersIllformedE(memset : SETParams) => boolean
begin
    return FALSE;
end;
```

J1.2.3.151 MemSetParametersIllformedM

```
// MemSetParametersIllformedM()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.

impdef func MemSetParametersIllformedM(memset : SETParams) => boolean
begin
    return FALSE;
end;
```

J1.2.3.152 MemSetStageSize

```
// MemSetStageSize()
// =================
// Returns the number of bytes set by the given stage of a SET* or SETG* instruction.

func MemSetStageSize(memset : SETParams) => integer
begin
    var stagesetsize : integer;

    if memset.stage == MOPSStage_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagesetsize = SETPreSizeChoice(memset);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert stagesetsize == 0 || (stagesetsize < 0) == (memset.setsize < 0);

        if memset.is_setg then assert stagesetsize[3:0] == '0000'; end;

        if memset.setsize > 0 then
            assert stagesetsize <= memset.setsize;
        else
            assert stagesetsize >= memset.setsize;
        end;

    else
        let postsize : integer = SETPostSizeChoice(memset);
        assert postsize == 0 || (postsize < 0) == (memset.setsize < 0);
        if memset.is_setg then assert postsize[3:0] == '0000'; end;

        if memset.stage == MOPSStage_Main then
            stagesetsize = memset.setsize - postsize;
        else
            stagesetsize = postsize;
        end;
    end;

    return stagesetsize;
end;
```

J1.2.3.153 MemSetTags

```
// MemSetTags()
// ============
// Write Allocation Tags for each Tag Granule in 'size'.
// The integer return parameter indicates the number of Tag Granules written. The
// AddressDescriptor and PhysMemRetStatus parameters contain Fault or Abort information for
// the caller to handle.

func MemSetTags(toaddress : bits(64), tag : bits(4),
                size : integer, accdesc : AccessDescriptor
            ) => (integer, AddressDescriptor, PhysMemRetStatus)
begin
    assert IsAlignedSize{64}(toaddress, TAG_GRANULE) && size MOD TAG_GRANULE == 0;

    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    memaddrdesc.fault     = NoFault();
    memstatus.statuscode  = Fault_None;
    var tagstep : integer = size DIV TAG_GRANULE;

    // Write until all tags are written or a fault is encountered.
    while tagstep > 0 && !IsFault(memaddrdesc) && !IsFault(memstatus)
                                            looplimit ArchMaxMOPSSETGSize DIV TAG_GRANULE do
        let tagaddr : bits(64) = toaddress + (tagstep - 1) * TAG_GRANULE;
        (memaddrdesc, memstatus) = AArch64_MemTagWrite(tagaddr, accdesc, tag);
        tagstep = tagstep - 1;
    end;

    if IsFault(memaddrdesc) || IsFault(memstatus) then
        tagstep = tagstep + 1;
    end;

    let tags_written : integer = (size DIV TAG_GRANULE) - tagstep;
    return (tags_written, memaddrdesc, memstatus);
end;
```

J1.2.3.154 MemSetZeroSizeCheck

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// MemSetZeroSizeCheck()
// =====================
// Returns TRUE if the implementation option is checked on a set of size zero remaining.

func MemSetZeroSizeCheck() => boolean
begin
    return ImpDefBool("Implementation option is checked with a setsize of 0");
end;
```

J1.2.3.155 MemStageCpyZeroSizeCheck

```
// MemStageCpyZeroSizeCheck()
// ==========================
// Returns TRUE if the implementation option is checked on a stage copy of size zero remaining.

func MemStageCpyZeroSizeCheck() => boolean
begin
    return (ImpDefBool(
            "Implementation option is checked with a stage cpysize of 0"));
end;
```

J1.2.3.156 MemStageSetZeroSizeCheck

```
// MemStageSetZeroSizeCheck()
// ==========================
// Returns TRUE if the implementation option is checked on a stage set of size zero remaining.

func MemStageSetZeroSizeCheck() => boolean
begin
    return (ImpDefBool(
            "Implementation option is checked with a stage setsize of 0"));
end;
```

J1.2.3.157 MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// ==========================
// Return the target exception level for an Exception_MemCpyMemSet.

func MismatchedCpySetTargetEL() => bits(2)
begin
    var target_el : bits(2);

    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2().MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;
    end;

    return target_el;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.158 MismatchedMemCpyException

```
// MismatchedMemCpyException()
// ===========================
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

func MismatchedMemCpyException(memcpy : CPYParams, options : bits(4), formatoption : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer                 = 0x0;
    let target_el : bits(2)                   = MismatchedCpySetTargetEL();

    var except : ExceptionRecord = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome.iss[24]    = '0';
    except.syndrome.iss[23]    = '0';
    except.syndrome.iss[22:19] = options;
    except.syndrome.iss[18]    = if memcpy.stage == MOPSStage_Epilogue then '1' else '0';
    except.syndrome.iss[17:16] = formatoption;
    // exception.syndrome[15] is RES0.
    except.syndrome.iss[14:10] = memcpy.d[4:0];
    except.syndrome.iss[9:5]   = memcpy.s[4:0];
    except.syndrome.iss[4:0]   = memcpy.n[4:0];

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.3.159 MismatchedMemSetException

```
// MismatchedMemSetException()
// ===========================
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

func MismatchedMemSetException(memset : SETParams, options : bits(2), formatoption : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    let target_el : bits(2) = MismatchedCpySetTargetEL();

    var except : ExceptionRecord = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome.iss[24]    = '1';
    except.syndrome.iss[23]    = if memset.is_setg then '1' else '0';
    // exception.syndrome[22:21] is RES0.
    except.syndrome.iss[20:19] = options;
    except.syndrome.iss[18]    = if memset.stage == MOPSStage_Epilogue then '1' else '0';
    except.syndrome.iss[17:16] = formatoption;
    // exception.syndrome[15] is RES0.
    except.syndrome.iss[14:10] = memset.d[4:0];
    except.syndrome.iss[9:5]   = memset.s[4:0];
    except.syndrome.iss[4:0]   = memset.n[4:0];

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.3.160 SETGOptionA

```
// SETGOptionA()
// =============
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

func SETGOptionA() => boolean
begin
    return ImpDefBool("SETG* instructions use Option A");
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.161 SETOptionA

```
// SETOptionA()
// ============
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

func SETOptionA() => boolean
begin
    return ImpDefBool("SET* instructions use Option A");
end;
```

J1.2.3.162 SETParams

```
// SETParams
// =========

type SETParams of record {
    stage : MOPSStage,
    implements_option_a : boolean,
    is_setg : boolean,
    setsize : integer,
    stagesetsize : integer,
    toaddress : bits(64),
    nzcv : bits(4),
    n : integer,
    d : integer,
    s : integer
};
```

J1.2.3.163 SETPostSizeChoice

```
// SETPostSizeChoice()
// ===================
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.

impdef func SETPostSizeChoice(memset : SETParams) => integer
begin
    return ARBITRARY : integer;
end;
```

J1.2.3.164 SETPreSizeChoice

```
// SETPreSizeChoice()
// ==================
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.

impdef func SETPreSizeChoice(memset : SETParams) => integer
begin
    return ARBITRARY : integer;
end;
```

J1.2.3.165 SETSizeChoice

```
// SETSizeChoice()
// ===============
// Returns the size of the block this performed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of alignsize.

impdef func SETSizeChoice(memset : SETParams, alignsize : integer) => MOPSBlockSize
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return ARBITRARY : MOPSBlockSize;
end;
```

J1.2.3.166 UpdateCpyRegisters

```
// UpdateCpyRegisters()
// ====================
// Performs updates to the X[n], X[d], and X[s] registers, as appropriate, for the CPY* and CPYF*
// instructions. When fault is TRUE, the values correspond to the first element not copied,
// such that a return to the instruction will enable a resumption of the copy.

func UpdateCpyRegisters(memcpy : CPYParams, fault : boolean, copied : integer)
begin
    if fault then
        if memcpy.stage == MOPSStage_Prologue then
            // Undo any formatting of the input parameters performed in the prologue.
            if memcpy.implements_option_a then
                if memcpy.forward then
                    // cpysize is negative.
                    let cpysize : integer  = memcpy.cpysize + copied;
                    X{64}(memcpy.n)  = (0 - cpysize)[63:0];
                    X{64}(memcpy.d)  = memcpy.toaddress   + cpysize;
                    X{64}(memcpy.s)  = memcpy.fromaddress + cpysize;

                else
                    X{64}(memcpy.n) = (memcpy.cpysize - copied)[63:0];
                end;

            else
                if memcpy.forward then
                    X{64}(memcpy.n) = (memcpy.cpysize - copied)[63:0];
                    X{64}(memcpy.d) = memcpy.toaddress   + copied;
                    X{64}(memcpy.s) = memcpy.fromaddress + copied;

                else
                    X{64}(memcpy.n)  = (memcpy.cpysize - copied)[63:0];
                end;
            end;

        else
            if memcpy.implements_option_a then
                if memcpy.forward then
                    X{64}(memcpy.n) = (memcpy.cpysize + copied)[63:0];
                else
                    X{64}(memcpy.n) = (memcpy.cpysize - copied)[63:0];
                end;

            else
                X{64}(memcpy.n) = (memcpy.cpysize - copied)[63:0];

                if memcpy.forward then
                    X{64}(memcpy.d) = memcpy.toaddress   + copied;
                    X{64}(memcpy.s) = memcpy.fromaddress + copied;
                else
                    X{64}(memcpy.d) = memcpy.toaddress   - copied;
                    X{64}(memcpy.s) = memcpy.fromaddress - copied;
                end;
            end;
        end;
    else
        X{64}(memcpy.n) = memcpy.cpysize[63:0];
        if memcpy.stage == MOPSStage_Prologue || !memcpy.implements_option_a then
            X{64}(memcpy.d) = memcpy.toaddress;
            X{64}(memcpy.s) = memcpy.fromaddress;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
end;
```

J1.2.3.167 UpdateSetRegisters

```
// UpdateSetRegisters()
// ====================
// Performs updates to the X[n] and X[d] registers, as appropriate, for the SET* and SETG*
// instructions. When fault is TRUE, the values correspond to the first element not set, such
// that a return to the instruction will enable a resumption of the memory set.

func UpdateSetRegisters(memset : SETParams, fault : boolean, memory_set : integer)
begin
    if fault then
        // Undo any formatting of the input parameters performed in the prologue.
        if memset.stage == MOPSStage_Prologue then
            if memset.implements_option_a then
                // setsize is negative.
                let setsize : integer = memset.setsize   + memory_set;
                X{64}(memset.n) = (0 - setsize)[63:0];
                X{64}(memset.d) = memset.toaddress + setsize;
            else
                X{64}(memset.n) = (memset.setsize - memory_set)[63:0];
                X{64}(memset.d) = memset.toaddress + memory_set;
            end;

        else
            if memset.implements_option_a then
                X{64}(memset.n) = (memset.setsize + memory_set)[63:0];
            else
                X{64}(memset.n) = (memset.setsize - memory_set)[63:0];
                X{64}(memset.d) = memset.toaddress + memory_set;
            end;
        end;
    else
        X{64}(memset.n) = memset.setsize[63:0];
        if memset.stage == MOPSStage_Prologue || !memset.implements_option_a then
            X{64}(memset.d) = memset.toaddress;
        end;
    end;

    return;
end;
```

J1.2.3.168 MoveWideOp

```
// MoveWideOp
// ==========
// Move wide 16-bit immediate instruction types.

type MoveWideOp of enumeration {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

J1.2.3.169 MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

func MoveWidePreferred(sf : bit, immN : bit, imms : bits(6), immr : bits(6)) => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let s : integer = UInt(imms);
    let r : integer = UInt(immr);
    let width : integer = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && (immN::imms) != '1xxxxxx' then
        return FALSE;
    end;
    if sf == '0' && (immN::imms) != '00xxxxx' then
        return FALSE;
    end;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);
    end;

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));
    end;

    return FALSE;
end;
```

J1.2.3.170 AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

func AddPAC(ptr : bits(64), modifier : bits(64), K : bits(128), data : boolean) => bits(64)
begin
    let use_modifier2 : boolean = FALSE;
    return InsertPAC(ptr, modifier, Zeros{64}, use_modifier2, K, data);
end;
```

J1.2.3.171 AddPAC2

```
// AddPAC2()
// =========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

func AddPAC2(ptr : bits(64), modifier1 : bits(64), modifier2 : bits(64),
             K : bits(128), data : boolean) => bits(64)
begin
    let use_modifier2 : boolean = TRUE;
    return InsertPAC(ptr, modifier1, modifier2, use_modifier2, K, data);
end;
```

J1.2.3.172 InsertPAC

```
// InsertPAC()
// ===========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

func InsertPAC(ptr : bits(64), modifier : bits(64), modifier2 : bits(64), use_modifier2 : boolean,
               K : bits(128), data : boolean) => bits(64)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var PAC : bits(64);
var result : bits(64);
var ext_ptr : bits(64);
var extfield : bits(64);
var selbit : bit;
let tbi : boolean = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
let mtx : boolean = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                     EffectiveMTX(ptr, !data, PSTATE.EL) == '1');
let top_bit : integer{} = if tbi then 55 else 63;
let EL3_using_lva3 : boolean = (IsFeatureImplemented(FEAT_LVA3) &&
                                TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                AArch64_IASize(TCR_EL3().T0SZ) > 52);
let is_VA_56bit : boolean = (TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                             AArch64_IASize(TCR_EL3().T0SZ) == 56);

// If tagged pointers are in use for a regime with two TTBRs, use bit[55] of
// the pointer to select between upper and lower ranges, and preserve this.
// This handles the awkward case where there is apparently no correct choice between
// the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
// and 0xxxxxxx1 with TBI1=0 and TBI0=1:
if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
        // EL1 translation regime registers
        if data then
            if TCR_EL1().TBI1 == '1' || TCR_EL1().TBI0 == '1' then
                selbit = ptr[55];
            else
                selbit = ptr[63];
            end;
        else
            if ((TCR_EL1().TBI1 == '1' && TCR_EL1().TBID1 == '0') ||
                (TCR_EL1().TBI0 == '1' && TCR_EL1().TBID0 == '0')) then
                selbit = ptr[55];
            else
                selbit = ptr[63];
            end;
        end;
    else
        // EL2 translation regime registers
        if data then
            if TCR_EL2().TBI1 == '1' || TCR_EL2().TBI0 == '1' then
                selbit = ptr[55];
            else
                selbit = ptr[63];
            end;
        else
            if ((TCR_EL2().TBI1 == '1' && TCR_EL2().TBID1 == '0') ||
                (TCR_EL2().TBI0 == '1' && TCR_EL2().TBID0 == '0')) then
                selbit = ptr[55];
            else
                selbit = ptr[63];
            end;
        end;
    end;
else
    selbit = if tbi then ptr[55] else ptr[63];
end;

if IsFeatureImplemented(FEAT_PAuth2) && IsFeatureImplemented(FEAT_CONSTPACFIELD) then
    selbit = ptr[55];
end;
let bottom_PAC_bit : AddressSize = CalculateBottomPACBit(selbit);

if EL3_using_lva3 then
    extfield = Replicate{64}('0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    extfield = Replicate{64}(selbit);
end;

// Compute the pointer authentication code for a ptr with good extension bits
if tbi then
    if bottom_PAC_bit <= 55 then
        ext_ptr = (ptr[63:56] ::
                extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
    else
        ext_ptr = ptr[63:56] :: ptr[55:0];
    end;
elsif mtx then
    if bottom_PAC_bit <= 55 then
        ext_ptr = (extfield[63:60] :: ptr[59:56] ::
                    extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
    else
        ext_ptr = extfield[63:60] :: ptr[59:56] :: ptr[55:0];
    end;
else
    ext_ptr =  extfield[63:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0];
end;

if use_modifier2 then
    assert IsFeatureImplemented(FEAT_PAuth_LR);
    PAC = ComputePAC2(ext_ptr, modifier, modifier2, K[127:64], K[63:0]);
else
    PAC = ComputePAC(ext_ptr, modifier, K[127:64], K[63:0]);
end;

if !IsFeatureImplemented(FEAT_PAuth2) then
    // If FEAT_PAuth2 is not implemented, the PAC is corrupted if the pointer does not have
    // a canonical VA.
    assert bottom_PAC_bit <= 52;
    if !IsZero(ptr[top_bit:bottom_PAC_bit]) && !IsOnes(ptr[top_bit:bottom_PAC_bit]) then
        PAC[top_bit-1] = NOT(PAC[top_bit-1]);
    end;
end;

// Preserve the determination between upper and lower address at bit[55] and insert PAC into
// bits that are not used for the address or the tag(s).
if !IsFeatureImplemented(FEAT_PAuth2) then
    assert (bottom_PAC_bit <= 52);
    if tbi then
        result = ptr[63:56]::selbit::PAC[54:bottom_PAC_bit]::ptr[bottom_PAC_bit-1:0];
    else
        result = PAC[63:56]::selbit::PAC[54:bottom_PAC_bit]::ptr[bottom_PAC_bit-1:0];
    end;
else
    var bit55 : bit;
    if EL3_using_lva3 then
        // Bit 55 is an address bit (when VA size is 56-bits) or
        // used to store PAC (when VA size is less than 56-bits)
        if is_VA_56bit then
            bit55 = ptr[55];
        else
            bit55 = ptr[55] XOR PAC[55];
        end;
    else
        bit55 = selbit;
    end;
    if tbi then
        if bottom_PAC_bit < 55 then
            result = (ptr[63:56]                                :: bit55 ::
                      (ptr[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit]) ::
                      ptr[bottom_PAC_bit-1:0]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            else
                result = (ptr[63:56] :: bit55 :: ptr[54:0]);
            end;
        elsif mtx then
            if bottom_PAC_bit < 55 then
                result = ((ptr[63:60] XOR PAC[63:60]) :: ptr[59:56] :: bit55 ::
                          (ptr[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit]) ::
                          ptr[bottom_PAC_bit-1:0]);
            else
                result = ((ptr[63:60] XOR PAC[63:60]) :: ptr[59:56] :: bit55 ::
                           ptr[54:0]);
            end;
        else
            if bottom_PAC_bit < 55 then
                result = ((ptr[63:56] XOR PAC[63:56])              :: bit55 ::
                          (ptr[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit]) ::
                          ptr[bottom_PAC_bit-1:0]);
            else
                result = ((ptr[63:56] XOR PAC[63:56])              :: bit55 ::
                           ptr[54:0]);
            end;
        end;
    end;
    return result;
end;
```

J1.2.3.173 AddPACDA

```
// AddPACDA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

func AddPACDA(x : bits(64), y : bits(64)) => bits(64)
begin
    let APDAKey_EL1 : bits(128) = APDAKeyHi_EL1()[63:0] :: APDAKeyLo_EL1()[63:0];
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
    end;
end;
```

J1.2.3.174 AddPACDB

```
// AddPACDB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

func AddPACDB(x : bits(64), y : bits(64)) => bits(64)
begin
    let APDBKey_EL1 : bits(128) = APDBKeyHi_EL1()[63:0] :: APDBKeyLo_EL1()[63:0];
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.175 AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

func AddPACGA(x : bits(64), y : bits(64)) => bits(64)
begin
    var TrapEL2 : boolean;
    let APGAKey_EL1 : bits(128) = APGAKeyHi_EL1()[63:0] :: APGAKeyLo_EL1()[63:0];

    var TrapEL3 : boolean;
    case PSTATE.EL of
        when EL0 =>
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return ComputePAC(x, y, APGAKey_EL1[127:64], APGAKey_EL1[63:0])[63:32]::Zeros{32};
    end;
end;
```

J1.2.3.176 AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

func AddPACIA(x : bits(64), y : bits(64)) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0]::APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.177 AddPACIA2

```
// AddPACIA2()
// ===========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, z, and
// the APIAKey_EL1.

func AddPACIA2(x : bits(64), y : bits(64), z : bits(64)) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0]::APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIAKey_EL1, FALSE);
    end;
end;
```

J1.2.3.178 AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

func AddPACIB(x : bits(64), y : bits(64)) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
    end;
end;
```

J1.2.3.179 AddPACIB2

```
// AddPACIB2()
// ===========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, z, and
// the APIBKey_EL1.

func AddPACIB2(x : bits(64), y : bits(64), z : bits(64)) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIBKey_EL1, FALSE);
    end;
end;
```

J1.2.3.180 AArch64_PACFailException

```
// AArch64_PACFailException()
// ==========================
// Generates a PAC Fail Exception

func AArch64_PACFailException(syndrome : bits(2))
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_PACFail);
    except.syndrome.iss[1:0]     = syndrome;
    except.syndrome.iss[24:2]    = Zeros{23};                // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.3.181 Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

func Auth(ptr : bits(64), modifier : bits(64), K : bits(128), data : boolean, key_number : bit,
          is_combined : boolean) => bits(64)
begin
    let use_modifier2 : boolean = FALSE;
    return Authenticate(ptr, modifier, Zeros{64}, use_modifier2,
                        K, data, key_number, is_combined);
end;
```

J1.2.3.182 Auth2

```
// Auth2()
// =======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

func Auth2(ptr : bits(64), modifier1 : bits(64), modifier2 : bits(64), K : bits(128),
           data : boolean, key_number : bit, is_combined : boolean) => bits(64)
begin
    let use_modifier2 : boolean = TRUE;
    return Authenticate(ptr, modifier1, modifier2, use_modifier2, K, data, key_number, is_combined);
end;
```

J1.2.3.183 Authenticate

```
// Authenticate()
// ==============
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

func Authenticate(ptr : bits(64), modifier : bits(64), modifier2 : bits(64),
                  use_modifier2 : boolean, K : bits(128), data : boolean, key_number : bit,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    is_combined : boolean) => bits(64)
begin
    var PAC : bits(64);
    var result : bits(64);
    var original_ptr : bits(64);
    var error_code : bits(2);
    var extfield : bits(64);

    // Reconstruct the extension field used of adding the PAC to the pointer
    let tbi : boolean = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    let mtx : boolean = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                         EffectiveMTX(ptr, !data, PSTATE.EL) == '1');
    let bottom_PAC_bit : AddressSize = CalculateBottomPACBit(ptr[55]);
    let EL3_using_lva3 : boolean = (IsFeatureImplemented(FEAT_LVA3) &&
                                    TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                    AArch64_IASize(TCR_EL3().T0SZ) > 52);
    let is_VA_56bit : boolean = (TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                 AArch64_IASize(TCR_EL3().T0SZ) == 56);
    if EL3_using_lva3 then
        extfield = Replicate{64}('0');
    else
        extfield = Replicate{64}(ptr[55]);
    end;

    if tbi then
        if bottom_PAC_bit <= 55 then
            original_ptr = (ptr[63:56] ::
                            extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
        else
            original_ptr = ptr[63:56] :: ptr[55:0];
        end;
    elsif mtx then
        if bottom_PAC_bit <= 55 then
            original_ptr = (extfield[63:60] :: ptr[59:56] ::
                            extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
        else
            original_ptr = extfield[63:60] :: ptr[59:56] :: ptr[55:0];
        end;
    else
        original_ptr =  extfield[63:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0];
    end;

    if use_modifier2 then
        assert IsFeatureImplemented(FEAT_PAuth_LR);
        PAC = ComputePAC2(original_ptr, modifier, modifier2, K[127:64], K[63:0]);
    else
        PAC = ComputePAC(original_ptr, modifier, K[127:64], K[63:0]);
    end;
    // Check pointer authentication code
    if tbi then
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC[54:bottom_PAC_bit] == ptr[54:bottom_PAC_bit] then
                result = original_ptr;
            else
                error_code = key_number::NOT(key_number);
                result = original_ptr[63:55]::error_code::original_ptr[52:0];
            end;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result[55] = result[55] XOR PAC[55];
            end;
            if (bottom_PAC_bit < 55) then
                result[54:bottom_PAC_bit] = result[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit];
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                  (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if (EL3_using_lva3 && !is_VA_56bit && !IsZero(result[55:bottom_PAC_bit])) then
                    error_code = (if data then '1' else '0')::key_number;
                    AArch64_PACFailException(error_code);
                elsif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
                         result[54:bottom_PAC_bit] !=
                         Replicate{55-bottom_PAC_bit}(result[55])) then
                    error_code = (if data then '1' else '0')::key_number;
                    AArch64_PACFailException(error_code);
                end;
            end;
        end;
    elsif mtx then
        assert IsFeatureImplemented(FEAT_PAuth2);
        result = ptr;
        if EL3_using_lva3 && !is_VA_56bit then
            result[55] = result[55] XOR PAC[55];
        end;
        if (bottom_PAC_bit < 55) then
            result[54:bottom_PAC_bit] = result[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit];
        end;
        result[63:60] = result[63:60] XOR PAC[63:60];
        if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
              (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
            if (EL3_using_lva3 && !is_VA_56bit &&
                  (!IsZero(result[55:bottom_PAC_bit]) || !IsZero(result[63:60]))) then
                error_code = (if data then '1' else '0')::key_number;
                AArch64_PACFailException(error_code);
            elsif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
                     (((result[54:bottom_PAC_bit] !=
                        Replicate{55-bottom_PAC_bit}(result[55]))) ||
                      (result[63:60] != Replicate{4}(result[55])))) then
                error_code = (if data then '1' else '0')::key_number;
                AArch64_PACFailException(error_code);
            end;
        end;
    else
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC[54:bottom_PAC_bit] == ptr[54:bottom_PAC_bit] && PAC[63:56] == ptr[63:56] then
                result = original_ptr;
            else
                error_code = key_number::NOT(key_number);
                result     = original_ptr[63]::error_code::original_ptr[60:0];
            end;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result[55] = result[55] XOR PAC[55];
            end;
            if bottom_PAC_bit < 55 then
                result[54:bottom_PAC_bit] = result[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit];
            end;
            result[63:56] = result[63:56] XOR PAC[63:56];
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                  (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if (EL3_using_lva3 && !IsZero(result[63:bottom_PAC_bit])) then
                    error_code = (if data then '1' else '0')::key_number;
                    AArch64_PACFailException(error_code);
                elsif (!EL3_using_lva3 &&
                         result[63:bottom_PAC_bit] !=
                         Replicate{64-bottom_PAC_bit}(result[55])) then
                    error_code = (if data then '1' else '0')::key_number;
                    AArch64_PACFailException(error_code);
                end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
        end;
    end;
    return result;
end;
```

J1.2.3.184 AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

func AuthDA(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APDAKey_EL1 : bits(128) = APDAKeyHi_EL1()[63:0] :: APDAKeyLo_EL1()[63:0];
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
    end;
end;
```

J1.2.3.185 AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

func AuthDB(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APDBKey_EL1 : bits(128) = APDBKeyHi_EL1()[63:0] :: APDBKeyLo_EL1()[63:0];
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
    end;
end;
```

J1.2.3.186 AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

func AuthIA(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0] :: APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.187 AuthIA2

```
// AuthIA2()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA2().

func AuthIA2(x : bits(64), y : bits(64), z : bits(64), is_combined : boolean) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0] :: APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth2(x, y, z, APIAKey_EL1, FALSE, '0', is_combined);
    end;
end;
```

J1.2.3.188 AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

func AuthIB(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
    end;
end;
```

J1.2.3.189 AuthIB2

```
// AuthIB2()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB2().

func AuthIB2(x : bits(64), y : bits(64), z : bits(64), is_combined : boolean) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth2(x, y, z, APIBKey_EL1, FALSE, '1', is_combined);
    end;
end;
```

J1.2.3.190 AArch64_PACEffectiveTxSZ

```
// AArch64_PACEffectiveTxSZ()
// ==========================
// Compute the effective value for TxSZ used to determine the placement of the PAC field

func AArch64_PACEffectiveTxSZ(regime : Regime, walkparams : S1TTWParams) => bits(6)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let s1maxtxsz : integer = AArch64_MaxTxSZ(walkparams.tgx);
    let s1mintxsz : integer = AArch64_S1MinTxSZ(regime, walkparams);

    if AArch64_S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable_RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < s1mintxsz then
                return s1mintxsz[5:0];
            end;
            if UInt(walkparams.txsz) > s1maxtxsz then
                return s1maxtxsz[5:0];
            end;
        end;
    elsif UInt(walkparams.txsz) < s1mintxsz then
        return s1mintxsz[5:0];
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        return s1maxtxsz[5:0];
    end;

    return walkparams.txsz;
end;
```

J1.2.3.191 CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

func CalculateBottomPACBit(top_bit : bit) => AddressSize
begin
    var regime : Regime;
    var walkparams : S1TTWParams;
    var bottom_PAC_bit : AddressSize;

    regime = TranslationRegime(PSTATE.EL);
    let ss : SecurityState = CurrentSecurityState();
    walkparams             = AArch64_GetS1TTWParams(regime, PSTATE.EL, ss, Replicate{64}(top_bit));
    bottom_PAC_bit         = ((64 - UInt(AArch64_PACEffectiveTxSZ(regime, walkparams)))
                                as AddressSize);

    return bottom_PAC_bit;
end;
```

J1.2.3.192 ComputePAC

```
// ComputePAC()
// ============

func ComputePAC(data : bits(64), modifier : bits(64), key0 : bits(64), key1 : bits(64)) => bits(64)
begin
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        let isqarma3 : boolean = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        let isqarma3 : boolean = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    end;
    unreachable;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.193 ComputePAC2

```
// ComputePAC2()
// =============

func ComputePAC2(data : bits(64), modifier1 : bits(64), modifier2 : bits(64),
                 key0 : bits(64), key1 : bits(64)) => bits(64)
begin
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePAC2IMPDEF(data, modifier1, modifier2, key0, key1);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        let isqarma3 : boolean = TRUE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        let isqarma3 : boolean = FALSE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    end;
    unreachable;
end;
```

J1.2.3.194 ComputePAC2IMPDEF

```
// ComputePAC2IMPDEF()
// ===================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

impdef func ComputePAC2IMPDEF(data : bits(64),
                              modifier1 : bits(64),
                              modifier2 : bits(64),
                              key0 : bits(64),
                              key1 : bits(64)) => bits(64)
begin
    Unimplemented();
end;
```

J1.2.3.195 ComputePAC2QARMA

```
// ComputePAC2QARMA()
// ==================

func ComputePAC2QARMA(data : bits(64),
                      modifier1 : bits(64),
                      modifier2 : bits(64),
                      key0 : bits(64),
                      key1 : bits(64),
                      isqarma3 : boolean) => bits(64)
begin
    let concat_modifiers : bits(64) = modifier2[36:5]::modifier1[35:4];
    return ComputePACQARMA(data, concat_modifiers, key0, key1, isqarma3);
end;
```

J1.2.3.196 ComputePACIMPDEF

```
// ComputePACIMPDEF()
// ==================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

impdef func ComputePACIMPDEF(data : bits(64), modifier : bits(64),
                             key0 : bits(64), key1 : bits(64)) => bits(64)
begin
    Unimplemented();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.197 ComputePACQARMA

```
// ComputePACQARMA()
// =================
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

func ComputePACQARMA(data : bits(64),
                     modifier : bits(64),
                     key0 : bits(64),
                     key1 : bits(64),
                     isqarma3 : boolean) => bits(64)
begin
    var workingval : bits(64);
    var runningmod : bits(64);
    var roundkey : bits(64);
    var modk0 : bits(64);
    let Alpha : bits(64) = 0xC0AC29B7C97C50DD[63:0];

    var iterations : integer;
    RC[[0]] = 0x0000000000000000[63:0];
    RC[[1]] = 0x13198A2E03707344[63:0];
    RC[[2]] = 0xA4093822299F31D0[63:0];

    if isqarma3 then
        iterations = 2;
    else // QARMA5
        iterations = 4;
        RC[[3]] = 0x082EFA98EC4E6C89[63:0];
        RC[[4]] = 0x452821E638D01377[63:0];
    end;

    modk0 = key0[0]::key0[63:2]::(key0[63] XOR key0[1]);
    runningmod = modifier;
    workingval = data XOR key0;

    for i = 0 to iterations do
        roundkey   = key1 XOR runningmod;
        workingval = workingval XOR roundkey;
        workingval = workingval XOR RC[[i]];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        end;
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACSub(workingval);
        end;
        runningmod = TweakShuffle(runningmod[63:0]);
    end;
    roundkey   = modk0 XOR runningmod;
    workingval = workingval XOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    end;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 XOR workingval;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval XOR key0;
    workingval = workingval XOR runningmod;
    for i = 0 to iterations do
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACInvSub(workingval);
        end;
        if i < iterations then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        end;
        runningmod = TweakInvShuffle(runningmod[63:0]);
        roundkey = key1 XOR runningmod;
        workingval = workingval XOR RC[[iterations-i]];
        workingval = workingval XOR roundkey;
        workingval = workingval XOR Alpha;
    end;
    workingval = workingval XOR modk0;

    return workingval;
end;
```

J1.2.3.198 PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

func PACCellInvShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = indata[15:12];
    outdata[7:4]   = indata[27:24];
    outdata[11:8]  = indata[51:48];
    outdata[15:12] = indata[39:36];
    outdata[19:16] = indata[59:56];
    outdata[23:20] = indata[47:44];
    outdata[27:24] = indata[7:4];
    outdata[31:28] = indata[19:16];
    outdata[35:32] = indata[35:32];
    outdata[39:36] = indata[55:52];
    outdata[43:40] = indata[31:28];
    outdata[47:44] = indata[11:8];
    outdata[51:48] = indata[23:20];
    outdata[55:52] = indata[3:0];
    outdata[59:56] = indata[43:40];
    outdata[63:60] = indata[63:60];
    return outdata;
end;
```

J1.2.3.199 PACCellShuffle

```
// PACCellShuffle()
// ================

func PACCellShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = indata[55:52];
    outdata[7:4]   = indata[27:24];
    outdata[11:8]  = indata[47:44];
    outdata[15:12] = indata[3:0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    outdata[19:16] = indata[31:28];
    outdata[23:20] = indata[51:48];
    outdata[27:24] = indata[7:4];
    outdata[31:28] = indata[43:40];
    outdata[35:32] = indata[35:32];
    outdata[39:36] = indata[15:12];
    outdata[43:40] = indata[59:56];
    outdata[47:44] = indata[23:20];
    outdata[51:48] = indata[11:8];
    outdata[55:52] = indata[39:36];
    outdata[59:56] = indata[19:16];
    outdata[63:60] = indata[63:60];
    return outdata;
end;
```

J1.2.3.200 PACInvSub

```
// PACInvSub()
// ===========

func PACInvSub(Tinput : bits(64)) => bits(64)
begin
    // This is a 4-bit substitution from the PRINCE-family cipher
    var Toutput : bits(64);
    for i = 0 to 15 do
        case Tinput[i*:4] of
            when '0000' =>  Toutput[i*:4] = '0101';
            when '0001' =>  Toutput[i*:4] = '1110';
            when '0010' =>  Toutput[i*:4] = '1101';
            when '0011' =>  Toutput[i*:4] = '1000';
            when '0100' =>  Toutput[i*:4] = '1010';
            when '0101' =>  Toutput[i*:4] = '1011';
            when '0110' =>  Toutput[i*:4] = '0001';
            when '0111' =>  Toutput[i*:4] = '1001';
            when '1000' =>  Toutput[i*:4] = '0010';
            when '1001' =>  Toutput[i*:4] = '0110';
            when '1010' =>  Toutput[i*:4] = '1111';
            when '1011' =>  Toutput[i*:4] = '0000';
            when '1100' =>  Toutput[i*:4] = '0100';
            when '1101' =>  Toutput[i*:4] = '1100';
            when '1110' =>  Toutput[i*:4] = '0111';
            when '1111' =>  Toutput[i*:4] = '0011';
        end;
    end;
    return Toutput;
end;
```

J1.2.3.201 PACMult

```
// PACMult()
// =========

func PACMult(Sinput : bits(64)) => bits(64)
begin
    var t0 : bits(4);
    var t1 : bits(4);
    var t2 : bits(4);
    var t3 : bits(4);
    var Soutput : bits(64);

    for i = 0 to 3 do
        t0[3:0] = ROL(Sinput[((i+8))*:4], 1) XOR ROL(Sinput[((i+4))*:4], 2);
        t0[3:0] = t0[3:0] XOR ROL(Sinput[i*:4], 1);
        t1[3:0] = ROL(Sinput[((i+12))*:4], 1) XOR ROL(Sinput[((i+4))*:4], 1);
        t1[3:0] = t1[3:0] XOR ROL(Sinput[i*:4], 2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        t2[3:0] = ROL(Sinput[((i+12))*:4], 2) XOR ROL(Sinput[((i+8))*:4], 1);
        t2[3:0] = t2[3:0] XOR ROL(Sinput[i*:4], 1);
        t3[3:0] = ROL(Sinput[((i+12))*:4], 1) XOR ROL(Sinput[((i+8))*:4], 2);
        t3[3:0] = t3[3:0] XOR ROL(Sinput[((i+4))*:4], 1);
        Soutput[i*:4] = t3[3:0];
        Soutput[((i+4))*:4] = t2[3:0];
        Soutput[((i+8))*:4] = t1[3:0];
        Soutput[((i+12))*:4] = t0[3:0];
    end;
    return Soutput;
end;
```

J1.2.3.202 PACSub

```
// PACSub()
// ========

func PACSub(Tinput : bits(64)) => bits(64)
begin
    // This is a 4-bit substitution from the PRINCE-family cipher
    var Toutput : bits(64);
    for i = 0 to 15 do
        case Tinput[i*:4] of
            when '0000' =>  Toutput[i*:4] = '1011';
            when '0001' =>  Toutput[i*:4] = '0110';
            when '0010' =>  Toutput[i*:4] = '1000';
            when '0011' =>  Toutput[i*:4] = '1111';
            when '0100' =>  Toutput[i*:4] = '1100';
            when '0101' =>  Toutput[i*:4] = '0000';
            when '0110' =>  Toutput[i*:4] = '1001';
            when '0111' =>  Toutput[i*:4] = '1110';
            when '1000' =>  Toutput[i*:4] = '0011';
            when '1001' =>  Toutput[i*:4] = '0111';
            when '1010' =>  Toutput[i*:4] = '0100';
            when '1011' =>  Toutput[i*:4] = '0101';
            when '1100' =>  Toutput[i*:4] = '1101';
            when '1101' =>  Toutput[i*:4] = '0010';
            when '1110' =>  Toutput[i*:4] = '0001';
            when '1111' =>  Toutput[i*:4] = '1010';
        end;
    end;
    return Toutput;
end;
```

J1.2.3.203 PacSub1

```
// PacSub1()
// =========

func PACSub1(Tinput : bits(64)) => bits(64)
begin
    // This is a 4-bit substitution from Qarma sigma1
    var Toutput : bits(64);
    for i = 0 to 15 do
        case Tinput[i*:4] of
            when '0000' => Toutput[i*:4] = '1010';
            when '0001' => Toutput[i*:4] = '1101';
            when '0010' => Toutput[i*:4] = '1110';
            when '0011' => Toutput[i*:4] = '0110';
            when '0100' => Toutput[i*:4] = '1111';
            when '0101' => Toutput[i*:4] = '0111';
            when '0110' => Toutput[i*:4] = '0011';
            when '0111' => Toutput[i*:4] = '0101';
            when '1000' => Toutput[i*:4] = '1001';
            when '1001' => Toutput[i*:4] = '1000';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '1010' => Toutput[i*:4] = '0000';
            when '1011' => Toutput[i*:4] = '1100';
            when '1100' => Toutput[i*:4] = '1011';
            when '1101' => Toutput[i*:4] = '0001';
            when '1110' => Toutput[i*:4] = '0010';
            when '1111' => Toutput[i*:4] = '0100';
        end;
    end;
    return Toutput;
end;
```

J1.2.3.204 RC

```
// RC[]
// ====

var RC : array [[5]] of bits(64);
```

J1.2.3.205 TweakCellInvRot

```
// TweakCellInvRot()
// =================

func TweakCellInvRot(incell : bits(4)) => bits(4)
begin
    var outcell : bits(4);
    outcell[3] = incell[2];
    outcell[2] = incell[1];
    outcell[1] = incell[0];
    outcell[0] = incell[0] XOR incell[3];
    return outcell;
end;
```

J1.2.3.206 TweakCellRot

```
// TweakCellRot()
// ==============

func TweakCellRot(incell : bits(4)) => bits(4)
begin
    var outcell : bits(4);
    outcell[3] = incell[0] XOR incell[1];
    outcell[2] = incell[3];
    outcell[1] = incell[2];
    outcell[0] = incell[1];
    return outcell;
end;
```

J1.2.3.207 TweakInvShuffle

```
// TweakInvShuffle()
// =================

func TweakInvShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = TweakCellInvRot(indata[51:48]);
    outdata[7:4]   = indata[55:52];
    outdata[11:8]  = indata[23:20];
    outdata[15:12] = indata[27:24];
    outdata[19:16] = indata[3:0];
    outdata[23:20] = indata[7:4];
    outdata[27:24] = TweakCellInvRot(indata[11:8]);
    outdata[31:28] = indata[15:12];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    outdata[35:32] = TweakCellInvRot(indata[31:28]);
    outdata[39:36] = TweakCellInvRot(indata[63:60]);
    outdata[43:40] = TweakCellInvRot(indata[59:56]);
    outdata[47:44] = TweakCellInvRot(indata[19:16]);
    outdata[51:48] = indata[35:32];
    outdata[55:52] = indata[39:36];
    outdata[59:56] = indata[43:40];
    outdata[63:60] = TweakCellInvRot(indata[47:44]);
    return outdata;
end;
```

J1.2.3.208 TweakShuffle

```
// TweakShuffle()
// ==============

func TweakShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = indata[19:16];
    outdata[7:4]   = indata[23:20];
    outdata[11:8]  = TweakCellRot(indata[27:24]);
    outdata[15:12] = indata[31:28];
    outdata[19:16] = TweakCellRot(indata[47:44]);
    outdata[23:20] = indata[11:8];
    outdata[27:24] = indata[15:12];
    outdata[31:28] = TweakCellRot(indata[35:32]);
    outdata[35:32] = indata[51:48];
    outdata[39:36] = indata[55:52];
    outdata[43:40] = indata[59:56];
    outdata[47:44] = TweakCellRot(indata[63:60]);
    outdata[51:48] = TweakCellRot(indata[3:0]);
    outdata[55:52] = indata[7:4];
    outdata[59:56] = TweakCellRot(indata[43:40]);
    outdata[63:60] = TweakCellRot(indata[39:36]);
    return outdata;
end;
```

J1.2.3.209 IsAPDAKeyEnabled

```
// IsAPDAKeyEnabled()
// ==================
// Returns TRUE if authentication using the APDAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPDAKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1().EnDA else SCTLR_EL2().EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable  = SCTLR_EL1().EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable  = SCTLR_EL2().EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL3 =>
            Enable  = SCTLR_EL3().EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

J1.2.3.210 IsAPDBKeyEnabled

```
// IsAPDBKeyEnabled()
// ==================
// Returns TRUE if authentication using the APDBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPDBKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1().EnDB else SCTLR_EL2().EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable  = SCTLR_EL1().EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable  = SCTLR_EL2().EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable  = SCTLR_EL3().EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

J1.2.3.211 IsAPIAKeyEnabled

```
// IsAPIAKeyEnabled()
// ==================
// Returns TRUE if authentication using the APIAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPIAKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1().EnIA else SCTLR_EL2().EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable  = SCTLR_EL1().EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable  = SCTLR_EL2().EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable  = SCTLR_EL3().EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

J1.2.3.212 IsAPIBKeyEnabled

```
// IsAPIBKeyEnabled()
// ==================
// Returns TRUE if authentication using the APIBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPIBKeyEnabled() => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1().EnIB else SCTLR_EL2().EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable  = SCTLR_EL1().EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable  = SCTLR_EL2().EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable  = SCTLR_EL3().EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

J1.2.3.213 IsPACMEnabled

```
// IsPACMEnabled()
// ===============
// Returns TRUE if the effects of the PACM instruction are enabled, otherwise FALSE.

func IsPACMEnabled() => boolean
begin
    assert IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_PAuth_LR);

    if IsTrivialPACMImplementation() then
        return FALSE;
    end;

    var enabled : boolean;

    // EL2 could force the behavior at EL1 and EL0 to NOP.
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        enabled = IsHCRXEL2Enabled() && HCRX_EL2().PACMEn == '1';
    else
        enabled = TRUE;
    end;

    // Otherwise, the SCTLR2_ELx bit determines the behavior.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if enabled then
        var enpacm_bit : bit;
        case PSTATE.EL of
            when EL3 =>
                enpacm_bit = SCTLR2_EL3().EnPACM;
            when EL2 =>
                enpacm_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2().EnPACM else '0';
            when EL1 =>
                enpacm_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1().EnPACM else '0';
            when EL0 =>
                if IsInHost() then
                    enpacm_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2().EnPACM0 else '0';
                else
                    enpacm_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1().EnPACM0 else '0';
                end;
        end;
        enabled = enpacm_bit == '1';
    end;

    return enabled;
end;
```

J1.2.3.214 IsTrivialPACMImplementation

```
// IsTrivialPACMImplementation()
// ============================
// Returns TRUE if the PE has a trivial implementation of PACM.

func IsTrivialPACMImplementation() => boolean
begin
    return (IsFeatureImplemented(FEAT_PACIMP) &&
              ImpDefBool("Trivial PSTATE.PACM implementation"));
end;
```

J1.2.3.215 PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

func PtrHasUpperAndLowerAddRanges() => boolean
begin
    let regime : Regime = TranslationRegime(PSTATE.EL);

    return HasUnprivileged(regime);
end;
```

J1.2.3.216 Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

func Strip(A : bits(64), data : boolean) => bits(64)
begin
    var original_ptr : bits(64);
    var extfield : bits(64);
    let tbi : boolean = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    let mtx : boolean = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                         EffectiveMTX(A, !data, PSTATE.EL) == '1');
    let bottom_PAC_bit : AddressSize = CalculateBottomPACBit(A[55]);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let EL3_using_lva3 : boolean = (IsFeatureImplemented(FEAT_LVA3) &&
                                    TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                    AArch64_IASize(TCR_EL3().T0SZ) > 52);
    if EL3_using_lva3 then
        extfield = Replicate{64}('0');
    else
        extfield = Replicate{64}(A[55]);
    end;

    if tbi then
        if (bottom_PAC_bit <= 55) then
            original_ptr = (A[63:56] ::
                            extfield[55:bottom_PAC_bit] :: A[bottom_PAC_bit-1:0]);
        else
            original_ptr = A[63:56] :: A[55:0];
        end;
    elsif mtx then
        if (bottom_PAC_bit <= 55) then
            original_ptr = (extfield[63:60] :: A[59:56] ::
                            extfield[55:bottom_PAC_bit] :: A[bottom_PAC_bit-1:0]);
        else
            original_ptr = extfield[63:60] :: A[59:56] :: A[55:0];
        end;
    else
        original_ptr =  extfield[63:bottom_PAC_bit] :: A[bottom_PAC_bit-1:0];
    end;

    return original_ptr;
end;
```

J1.2.3.217 TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

noreturn func TrapPACUse(target_el : bits(2))
begin
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var except : ExceptionRecord;
    let vect_offset : integer = 0;
    except = ExceptionSyndrome(Exception_PACTrap);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

J1.2.3.218 AArch64_RestrictPrediction

```
// AArch64_RestrictPrediction()
// ============================
// Clear all predictions in the context.

func AArch64_RestrictPrediction(val : bits(64), restriction : RestrictType)
begin

    var c : ExecutionCntxt;
    let target_el : bits(2)    = val[25:24];

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP(); end;

    let ns : bit  = val[26];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let nse : bit = val[27];
    let ss : SecurityState = TargetSecurityState(ns, nse);

    // If the combination of Security state and Exception level is not implemented,
    // the instruction is treated as a NOP.
    if ss == SS_Root && target_el != EL3 then ExecuteAsNOP(); end;
    if !IsFeatureImplemented(FEAT_RME) && target_el == EL3 && ss != SS_Secure then
        ExecuteAsNOP();
    end;

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID();

        elsif (target_el == EL0 && !ELIsInHost(target_el)) || target_el == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val[48] == '1';
            c.vmid          = val[47:32];       // Only valid if  val[48] == '0';

        else
            c.is_vmid_valid = FALSE;
        end;
    else
        c.is_vmid_valid = FALSE;
    end;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID();

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val[16] == '1';
        c.asid          = val[15:0];            // Only valid if  val[16] == '0';

    else
        c.is_asid_valid = FALSE;
    end;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
end;
```

J1.2.3.219 Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

func Prefetch(address : bits(64), prfop : bits(5), supports_ir : boolean)
begin
    var hint : PrefetchHint;
    var target : integer;
    var stream : boolean;

    case prfop of
        when '00xxx'                  => hint = Prefetch_READ;      // PLD: prefetch for load
        when '01xxx'                  => hint = Prefetch_EXEC;      // PLI: preload instructions
        when '10xxx'                  => hint = Prefetch_WRITE;     // PST: prepare for store
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '11000' where supports_ir => hint = Prefetch_IR;        // IR: intent to read
        otherwise                      => return;                    // unallocated hint
    end;

    // If <type> is IR, then <target> and <policy> are not specified and Rt[2:0] are '000'.
    if hint != Prefetch_IR then
        target = UInt(prfop[2:1]);                 // target cache level
        stream = (prfop[0] != '0');                // streaming (non-temporal)
    else
        target = ARBITRARY: integer;
        stream = ARBITRARY: boolean;
    end;
    Hint_Prefetch(address, hint, target, stream);
    return;
end;
```

J1.2.3.220 PSTATEField

```
// PSTATEField
// ===========
// MSR (immediate) instruction destinations.

type PSTATEField of enumeration {
                        PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                        PSTATEField_PAN, // Armv8.1
                        PSTATEField_UAO, // Armv8.2
                        PSTATEField_DIT, // Armv8.4
                        PSTATEField_SSBS,
                        PSTATEField_TCO, // Armv8.5
                        PSTATEField_SVCRSM,
                        PSTATEField_SVCRZA,
                        PSTATEField_SVCRSMZA,
                        PSTATEField_ALLINT,
                        PSTATEField_PM,
                        PSTATEField_SP
                        };
```

J1.2.3.221 AArch64_DelegatedSErrorTarget

```
// AArch64_DelegatedSErrorTarget()
// ===============================
// Returns whether a delegated SError exception pended by SCR_EL3.VSE is masked,
// and the target Exception level of the delegated SError exception.

func AArch64_DelegatedSErrorTarget() => (boolean, bits(2))
begin
    assert IsFeatureImplemented(FEAT_E3DSE);
    if Halted() || PSTATE.EL == EL3 then
        return (TRUE, ARBITRARY : bits(2));
    end;

    let effective_amo : bit = EffectiveHCR_AMO();
    let effective_tge : bit = EffectiveTGE();

    // The exception is masked by software.
    var masked : boolean;
    case PSTATE.EL of
        when EL2 =>
            masked = ((effective_tge == '0' && effective_amo == '0') || PSTATE.A == '1');
        when EL1, EL0 =>
            masked = (effective_amo == '0' && PSTATE.A == '1');
        otherwise =>
            unreachable;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // The exception might be disabled debug in the Security state indicated by
    // SCR_EL3().[NS, NSE] by external debug.
    let intdis : boolean = ExternalDebugInterruptsDisabled(EL1);

    var target_el : bits(2) = ARBITRARY : bits(2);
    if EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
        target_el = EL2;
        masked = FALSE;

    elsif (EffectiveHCRX_EL2_TMEA() == '1' && !intdis &&
             ((PSTATE.EL == EL1 && PSTATE.A == '1') ||
              (PSTATE.EL == EL0 && masked && !IsInHost()))) then
        target_el = EL2;
        masked = FALSE;

    elsif PSTATE.EL == EL2 || IsInHost() then
        if !masked then target_el = EL2; end;

    else
        assert (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()));
        if !masked then target_el = EL1; end;
    end;

    // External debug might disable the delegated exception for the target Exception level.
    if ExternalDebugInterruptsDisabled(target_el) then
        masked = TRUE;
        target_el = ARBITRARY : bits(2);
    end;

    return (masked, target_el);
end;
```

J1.2.3.222 AArch64_ESBOperation

```
// AArch64_ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

func AArch64_ESBOperation()
begin
    var target_el : bits(2);
    var masked : boolean;

    (masked, target_el) = PhysicalSErrorTarget();

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interprocessing case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            var syndrome : bits(32) = Zeros{};
            syndrome[31]      = '1';    // A
            syndrome[15:0]    = AArch32_PhysicalSErrorSyndrome();
            DISR()            = syndrome;
        else
            let implicit_esb : boolean = FALSE;
            let is_esb : boolean       = TRUE;
            var syndrome : bits(64)    = Zeros{};

            syndrome[31]   = '1';       // A
            syndrome[24:0] = AArch64_PhysicalSErrorSyndrome(is_esb, implicit_esb);
            DISR_EL1()     = syndrome;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ClearPendingPhysicalSError();   // Set ISR_EL1.A to 0
    end;

    return;
end;
```

J1.2.3.223 AArch64_EncodeAsyncErrorSyndrome

```
// AArch64_EncodeAsyncErrorSyndrome()
// ==================================
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch64 state.

func AArch64_EncodeAsyncErrorSyndrome(errorstate : ErrorState) => bits(3)
begin
    case errorstate of
        when ErrorState_UC  => return '000';
        when ErrorState_UEU => return '001';
        when ErrorState_UEO => return '010';
        when ErrorState_UER => return '011';
        when ErrorState_CE  => return '110';
        otherwise => unreachable;
    end;
end;
```

J1.2.3.224 AArch64_EncodeSyncErrorSyndrome

```
// AArch64_EncodeSyncErrorSyndrome()
// =================================
// Return the encoding for specified ErrorState for a synchronous Abort
// exception taken to AArch64 state.

func AArch64_EncodeSyncErrorSyndrome(errorstate : ErrorState) => bits(2)
begin
    case errorstate of
        when ErrorState_UC  => return '10';
        when ErrorState_UEU => return '10';    // UEU is reported as UC
        when ErrorState_UEO => return '11';
        when ErrorState_UER => return '00';
        otherwise => unreachable;
    end;
end;
```

J1.2.3.225 AArch64_PhysicalSErrorSyndrome

```
// AArch64_PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

func AArch64_PhysicalSErrorSyndrome(is_esb : boolean, implicit_esb : boolean) => bits(25)
begin
    var syndrome : bits(25) = Zeros{};
    var target_el : bits(2);
    (-, target_el) = PhysicalSErrorTarget();

    if ReportErrorAsUncategorized() then
        syndrome = Zeros{25};
    elsif ReportErrorAsIMPDEF() then
        syndrome[24]   = '1';                                          // IDS
        syndrome[23:0] = ImpDefBits{24}("IMPDEF ErrorState");
    else
        let fault : FaultRecord = GetPendingPhysicalSError();
        let errorstate : ErrorState = PEErrorState(fault);
        syndrome[24]    = '0';                                         // IDS
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if (!is_esb && IsFeatureImplemented(FEAT_PFAR) &&
              !(EL2Enabled() &&
              (HCR_EL2().VM == '1' || HCR_EL2().DC == '1') && target_el == EL1)) then
            syndrome[14] = if IsPFARValid(fault) then '1' else '0';
        end;
        syndrome[13]    = if implicit_esb then '1' else '0';              // IESB
        syndrome[12:10] = AArch64_EncodeAsyncErrorSyndrome(errorstate);   // AET
        syndrome[9]     = fault.extflag;                                  // EA
        syndrome[5:0]   = '010001';                                       // DFSC
    end;

    return syndrome;
end;
```

J1.2.3.226 AArch64_dESBOperation

```
// AArch64_dESBOperation()
// =======================
// Perform the AArch64 ESB operation for a pending delegated SError exception.

func AArch64_dESBOperation()
begin
    assert (IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) && PSTATE.EL != EL3);
    // When FEAT_E3DSE is implemented, SCR_EL3.DSE might inject a delegated SError exception.
    var dsei_pending, dsei_masked : boolean;
    dsei_pending = EffectiveSCR_EL3_EnDSE() == '1' && SCR_EL3().DSE == '1';
    (dsei_masked, -) = AArch64_DelegatedSErrorTarget();
    if dsei_pending && dsei_masked then
        var target : bits(64) = Zeros{};
        target[31] = '1';                  // A
        target[24:0] = VSESR_EL3()[24:0];
        VDISR_EL3() = target;
        ClearPendingDelegatedSError();
    end;
    return;
end;
```

J1.2.3.227 AArch64_vESBOperation

```
// AArch64_vESBOperation()
// =======================
// Perform the AArch64 ESB operation for an unmasked pending virtual SError exception.
// If FEAT_E3DSE is implemented and there is no unmasked virtual SError exception
// pending, then AArch64_dESBOperation() is called to perform the AArch64 ESB operation
// for a pending delegated SError exception.

func AArch64_vESBOperation()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2);

    // If physical SError exceptions are routed to EL2, and TGE is not set, then a virtual
    // SError exception might be pending.
    let vsei_pending : boolean = (IsVirtualSErrorPending() && EffectiveTGE() == '0' &&
                                  (EffectiveHCR_AMO() == '1' || EffectiveHCRX_EL2_TMEA() == '1'));
    let vsei_masked : boolean = PSTATE.A == '1' || Halted() || ExternalDebugInterruptsDisabled(EL1);

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        // This function might be called for the interprocessing case, and INTdis is masking
        // the virtual SError exception.
        if ELUsingAArch32(EL1) then
            var target : bits(32) = Zeros{};
            target[31]    = '1';             // A
            target[15:14] = VDFSR()[15:14];  // AET
            target[12]    = VDFSR()[12];     // ExT
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            target[9]     = TTBCR().EAE;     // LPAE
            if TTBCR().EAE == '1' then       // Long-descriptor format
                target[5:0]    = '010001';   // STATUS
            else                             // Short-descriptor format
                target[10,3:0] = '10110';    // FS
            end;
            VDISR() = target;
        else
            var target : bits(64) = Zeros{};
            target[31]   = '1';              // A
            target[24:0] = VSESR_EL2()[24:0];
            VDISR_EL2() = target;
        end;
        ClearPendingVirtualSError();
    elsif IsFeatureImplemented(FEAT_E3DSE) then
        AArch64_dESBOperation();
    end;

    return;
end;
```

J1.2.3.228 EffectiveSCR_EL3_EnDSE

```
// EffectiveSCR_EL3_EnDSE()
// ========================
// Return the Effective value of SCR_EL3().EnDSE.

func EffectiveSCR_EL3_EnDSE() => bit
begin
    if !IsFeatureImplemented(FEAT_E3DSE) then
        return '0';
    end;
    if SCR_EL3().EA == '0' && EffectiveSCR_EL3_TMEA() == '0' then
        return '0';
    end;
    return SCR_EL3().EnDSE;
end;
```

J1.2.3.229 FirstRecordOfNode

```
// FirstRecordOfNode()
// ===================
// Return the first record in the node that contains the record n.

func FirstRecordOfNode(n : integer) => integer
begin
    for q = n downto 0 do
        if IsFirstRecordOfNode(q) then return q; end;
    end;
    unreachable;
end;
```

J1.2.3.230 IsCommonFaultInjectionImplemented

```
// IsCommonFaultInjectionImplemented()
// ===================================
// Check if the Common Fault Injection Model Extension is implemented by the node that owns this
// error record.

impdef func IsCommonFaultInjectionImplemented(n : integer) => boolean
begin
    return TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.231 IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// ===========================
// Check whether Error record n records countable errors.

impdef func IsCountableErrorsRecorded(n : integer) => boolean
begin
    return TRUE;
end;
```

J1.2.3.232 IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// ========================
// Check whether Error record n includes an address associated with an error.

impdef func IsErrorAddressIncluded(n : integer) => boolean
begin
    return TRUE;
end;
```

J1.2.3.233 IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// ==========================
// Is the error record n implemented

impdef func IsErrorRecordImplemented(n : integer) => boolean
begin
    return TRUE;
end;
```

J1.2.3.234 IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====================
// Check if the record q is the first error record in its node.

impdef func IsFirstRecordOfNode(q : integer) => boolean
begin
    return TRUE;
end;
```

J1.2.3.235 IsPFARValid

```
// IsPFARValid()
// =============
// Determine if faulting PA can be reported.

impdef func IsPFARValid(fault : FaultRecord) => boolean
begin
    return !IsAsyncAbort(fault);
end;
```

J1.2.3.236 IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// ==========================
// Does the System PMU s implement the counter n.

impdef func IsSPMUCounterImplemented(s : integer, n : integer) => boolean
begin
    return TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.237 rcw

```
constant RCW64_PROTECTED_BIT : integer = 52;
constant RCW128_PROTECTED_BIT : integer = 114;
```

J1.2.3.238 ProtectionEnabled

```
// ProtectionEnabled()
// ===================
// Returns TRUE if the ProtectedBit is
// enabled in the current Exception level.

func ProtectionEnabled(el : bits(2)) => boolean
begin
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));
    if (!IsD128Enabled(el)) then
        case regime of
            when EL1 =>
                return IsTCR2EL1Enabled() && TCR2_EL1().PnCH == '1';
            when EL2 =>
                return IsTCR2EL2Enabled() && TCR2_EL2().PnCH == '1';
            when EL3 =>
                return TCR_EL3().PnCH == '1';
        end;
    else
        return TRUE;
    end;
    return FALSE;
end;
```

J1.2.3.239 RCWCheck

```
// RCWCheck()
// ==========
// Returns nzcv based on : if the new value for RCW/RCWS instructions satisfy RCW and/or RCWS checks
// Z is set to 1 if RCW checks fail
// C is set to 0 if RCWS checks fail

func RCWCheck{N}(old : bits(N), new : bits(N), soft : boolean) => bits(4)
begin
    assert N IN {64,128};
    let protectedbit : integer = if N == 128 then RCW128_PROTECTED_BIT else RCW64_PROTECTED_BIT;
    var rcw_fail : boolean        = FALSE;
    var rcws_fail : boolean       = FALSE;
    var rcw_state_fail : boolean  = FALSE;
    var rcws_state_fail : boolean = FALSE;
    var rcw_mask_fail : boolean   = FALSE;
    var rcws_mask_fail : boolean  = FALSE;

    //Effective RCWMask calculation
    var rcwmask : bits(N) = RCWMASK_EL1()[N-1:0];
    if N == 64 then
        rcwmask[49:18] = Replicate{32}(rcwmask[17]);
        rcwmask[0] = '0';
    else
        rcwmask[55:17] = Replicate{39}(rcwmask[16]);
        rcwmask[126:125,120:119,107:101,90:56,1:0] = Zeros{48};
    end;

    //Effective RCWSMask calculation
    var rcwsoftmask : bits(N) = RCWSMASK_EL1()[N-1:0];
    if N == 64 then
        rcwsoftmask[49:18] = Replicate{32}(rcwsoftmask[17]);
        rcwsoftmask[0] = '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if(ProtectionEnabled(PSTATE.EL)) then
            rcwsoftmask[52] = '0';
        end;
    else
        rcwsoftmask[55:17] = Replicate{39}(rcwsoftmask[16]);
        rcwsoftmask[126:125,120:119,107:101,90:56,1:0] = Zeros{48};
        rcwsoftmask[114] = '0';
    end;

    //RCW Checks
    //State Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old[protectedbit] == '1' then
            rcw_state_fail = new[protectedbit,0] != old[protectedbit,0];
        elsif old[protectedbit] == '0' then
            rcw_state_fail = new[protectedbit] != old[protectedbit];
        end;
    end;

    //Mask Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old[protectedbit,0] == '11' then
            rcw_mask_fail = !IsZero((new XOR old) AND NOT(rcwmask));
        end;
    end;

    //RCWS Checks
    if soft then
        //State Check
        if old[0] == '1' then
            rcws_state_fail = new[0] != old[0];
        elsif (!ProtectionEnabled(PSTATE.EL) ||
              (ProtectionEnabled(PSTATE.EL) && old[protectedbit] == '0')) then
            rcws_state_fail = new[0] != old[0] ;
        end;
        //Mask Check
        if old[0] == '1' then
            rcws_mask_fail = !IsZero((new XOR old) AND NOT(rcwsoftmask));
        end;
    end;

    rcw_fail = rcw_state_fail  || rcw_mask_fail ;
    rcws_fail = rcws_state_fail || rcws_mask_fail;

    let n : bit = '0';
    let z : bit = if rcw_fail then '1' else '0';
    let c : bit = if rcws_fail then '0' else '1';
    let v : bit = '0';
    return n::z::c::v;
end;
```

J1.2.3.240 FPReduce

```
// FPReduce()
// ==========
// Perform the floating-point operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

func FPReduce{esize, N}(op : ReduceOp, input : bits(N),
                        fpcr : FPCR_Type) => bits(esize) recurselimit 8
begin
    var hi : bits(esize);
    var lo : bits(esize);
    var result : bits(esize);
    let half : integer{} = N DIV 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if N == esize then
        return input[esize-1:0];
    end;

    hi = FPReduce{esize, (N - N DIV 2)}(op, input[N-1:half], fpcr);
    lo = FPReduce{esize, N DIV 2}(op, input[half-1:0], fpcr);
    case op of
        when ReduceOp_FMINNUM =>
            result = FPMinNum{esize}(lo, hi, fpcr);
        when ReduceOp_FMAXNUM =>
            result = FPMaxNum{esize}(lo, hi, fpcr);
        when ReduceOp_FMIN =>
            result = FPMin{esize}(lo, hi, fpcr);
        when ReduceOp_FMAX =>
            result = FPMax{esize}(lo, hi, fpcr);
        when ReduceOp_FADD =>
            result = FPAdd{esize}(lo, hi, fpcr);
    end;

    return result;
end;
```

J1.2.3.241 IntReduce

```
// IntReduce()
// ===========
// Perform the integer operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

func IntReduce{esize, N}(op : ReduceOp, input : bits(N)) => bits(esize) recurselimit 8
begin
    var hi : bits(esize);
    var lo : bits(esize);
    var result : bits(esize);
    let half : integer{} = N DIV 2;
    if N == esize then
        return input[esize-1:0];
    end;

    hi = IntReduce{esize, (N - N DIV 2)}(op, input[N-1:half]);
    lo = IntReduce{esize, N DIV 2}(op, input[half-1:0]);
    case op of
        when ReduceOp_ADD =>
            result = lo + hi;
    end;

    return result;
end;
```

J1.2.3.242 ReduceOp

```
// ReduceOp
// ========
// Vector reduce instruction types.

type ReduceOp of enumeration {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                              ReduceOp_FMIN, ReduceOp_FMAX,
                              ReduceOp_FADD, ReduceOp_ADD};
```

J1.2.3.243 AArch64_MaybeZeroRegisterUppers

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64_MaybeZeroRegisterUppers()
// ================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

func AArch64_MaybeZeroRegisterUppers()
begin
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    var first : integer;
    var last : integer;
    var include_R15 : boolean;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;
    end;

    for n = first to last do
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[[n]][63:32] = Zeros{32};
        end;
    end;

    return;
end;
```

J1.2.3.244 AArch64_ResetGeneralRegisters

```
// AArch64_ResetGeneralRegisters()
// ==============================

func AArch64_ResetGeneralRegisters()
begin

    for i = 0 to 30 do
        X{64}(i) = ARBITRARY : bits(64);
    end;
end;
```

J1.2.3.245 AArch64_ResetSIMDFPRegisters

```
// AArch64_ResetSIMDFPRegisters()
// =============================

func AArch64_ResetSIMDFPRegisters()
begin

    for i = 0 to 31 do
        V{128}(i) = ARBITRARY : bits(128);
    end;
end;
```

J1.2.3.246 AArch64_ResetSpecialRegisters

```
// AArch64_ResetSpecialRegisters()
// ==============================

func AArch64_ResetSpecialRegisters()
begin

    // AArch64 special registers
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SP_EL0()       = ARBITRARY : bits(64);
    SP_EL1()       = ARBITRARY : bits(64);

    SPSR_EL1()     = ARBITRARY : bits(64);
    ELR_EL1()      = ARBITRARY : bits(64);
    if HaveEL(EL2) then
        SP_EL2()   = ARBITRARY : bits(64);
        SPSR_EL2() = ARBITRARY : bits(64);
        ELR_EL2()  = ARBITRARY : bits(64);
    end;


    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq()[31:0] = ARBITRARY : bits(32);
        SPSR_irq()[31:0] = ARBITRARY : bits(32);
        SPSR_abt()[31:0] = ARBITRARY : bits(32);
        SPSR_und()[31:0] = ARBITRARY : bits(32);
    end;

    // External debug special registers
    DLR_EL0()   = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    return;
end;
```

J1.2.3.247 AArch64_ResetSystemRegisters

```
// AArch64_ResetSystemRegisters()
// ==============================

impdef func AArch64_ResetSystemRegisters(cold_reset : boolean)
begin
    return;
end;
```

J1.2.3.248 PC64

```
// PC64 - getter
// =============
// Read 64-bit program counter.

func PC64() => bits(64)
begin
    return _PC;
end;
```

J1.2.3.249 SP

```
// SP - accessor
// =============

accessor SP{width}() <=> value : bits(width)
begin
    // Write a 32-bit or 64-bit value to the current stack pointer.
    setter
        assert width IN {64, 32};
        if PSTATE.SP == '0' then
            SP_EL0() = ZeroExtend{64}(value);
        else
            case PSTATE.EL of
                when EL0 =>  SP_EL0() = ZeroExtend{64}(value);
                when EL1 =>  SP_EL1() = ZeroExtend{64}(value);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                when EL2 =>  SP_EL2() = ZeroExtend{64}(value);
                when EL3 =>  SP_EL3() = ZeroExtend{64}(value);
            end;
        end;
    end;

    // Read the least-significant 32 or 64 bits from the current stack pointer.
    getter
        assert width IN {64, 32};
        if PSTATE.SP == '0' then
            return SP_EL0()[width-1:0];
        else
            case PSTATE.EL of
                when EL0 =>  return SP_EL0()[width-1:0];
                when EL1 =>  return SP_EL1()[width-1:0];
                when EL2 =>  return SP_EL2()[width-1:0];
                when EL3 =>  return SP_EL3()[width-1:0];
            end;
        end;
    end;
end;
```

J1.2.3.250 X

```
// X - accessor
// ============

accessor X{width}(n : integer) <=> value : bits(width)
begin
    // Write a 32-bit or 64-bit value to a general-purpose register.
    setter
        assert n >= 0 && n <= 31;
        assert width IN {32,64};
        if n != 31 then
            _R[[n]] = ZeroExtend{64}(value);
        end;
        return;
    end;

    // Read the least-significant 8, 16, 32, or 64 bits from a general-purpose register.
    getter
        assert n >= 0 && n <= 31;
        let rw : integer{} = width as integer{8, 16, 32, 64};
        if n != 31 then
            return _R[[n]][rw-1:0];
        else
            return Zeros{rw};
        end;
    end;
end;

// X - accessor
// ============

accessor X{width}(lr : integer, hr : integer) <=> value : bits(width)
begin
    // Write a 128-bit value to two general-purpose registers.
    setter
        assert width == 128;
        let half : integer{} = width DIV 2;
        X{half}(lr)          = value[0+:half];
        X{half}(hr)          = value[half+:half];
        return;
    end;

    // Read 128 bits from two separate general-purpose registers.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    getter
        assert width == 128;
        let half : integer{} = width DIV 2;
        return X{half}(hr) :: X{half}(lr);
    end;
end;
```

J1.2.3.251 DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

func DecodeShift(op : bits(2)) => ShiftType
begin
    case op of
        when '00' =>  return ShiftType_LSL;
        when '01' =>  return ShiftType_LSR;
        when '10' =>  return ShiftType_ASR;
        when '11' =>  return ShiftType_ROR;
    end;
end;
```

J1.2.3.252 ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

func ShiftReg{N}(reg : integer, shiftype : ShiftType, amount : integer) => bits(N)
begin
    var result : bits(N) = X{}(reg);
    case shiftype of
        when ShiftType_LSL => result = LSL(result, amount);
        when ShiftType_LSR => result = LSR(result, amount);
        when ShiftType_ASR => result = ASR(result, amount);
        when ShiftType_ROR => result = ROR(result, amount);
    end;
    return result;
end;
```

J1.2.3.253 ShiftType

```
// ShiftType
// =========
// AArch64 register shifts.

type ShiftType of enumeration {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

J1.2.3.254 CounterToPredicate

```
// CounterToPredicate()
// ====================

func CounterToPredicate{width}(pred : bits(16)) => bits(width)
begin
    var count : integer;
    var esize : ESize;
    var elements : integer;
    let VL : VecLen  = CurrentVL();
    let PL : PredLen = VL DIV 8;
    let maxbit : integer{} = FloorLog2(PL * 4) as integer{0..14};
    var result : bits(PL*4);
    let invert : boolean = pred[15] == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert width == PL || width == PL*2 || width == PL*3 || width == PL*4;

    case pred[3:0] of
        when '0000' =>
            return Zeros{width};
        when 'xxx1' =>
            count = UInt(pred[maxbit:1]);
            esize = 8;
        when 'xx10' =>
            count = UInt(pred[maxbit:2]);
            esize = 16;
        when 'x100' =>
            count = UInt(pred[maxbit:3]);
            esize = 32;
        when '1000' =>
            count = UInt(pred[maxbit:4]);
            esize = 64;
    end;

    elements = (VL * 4) DIV esize;
    result = Zeros{PL*4};
    let psize : integer{} = esize DIV 8;
    for e = 0 to elements-1 do
        var pbit : bit = if e < count then '1' else '0';
        if invert then
            pbit = NOT(pbit);
        end;
        result[e*:psize] = ZeroExtend{psize}(pbit);
    end;

    return result[width-1:0];
end;
```

J1.2.3.255 EncodePredCount

```
// EncodePredCount()
// =================

func EncodePredCount{width}(esize : ESize, elements : integer,
                            count_in : integer, invert_in : boolean) => bits(width)
begin
    var count : integer  = count_in;
    var invert : boolean = invert_in;
    let PL : PredLen = CurrentVL() DIV 8;
    assert width == PL;
    assert count >=0 && count <= elements;
    var pred : bits(16);

    if count == 0 then
        return Zeros{width};
    end;

    if invert then
        count = elements - count;
    elsif count == elements then
        count = 0;
        invert = TRUE;
    end;

    let inv : bit = (if invert then '1' else '0');
    case esize of
        when 8 =>  pred = inv :: count[13:0] ::    '1';
        when 16 => pred = inv :: count[12:0] ::   '10';
        when 32 => pred = inv :: count[11:0] ::  '100';
        when 64 => pred = inv :: count[10:0] :: '1000';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    return ZeroExtend{width}(pred);
end;
```

J1.2.3.256 Lookup

```
// Lookup Table
// ============

var _ZT0 : bits(ZT0_LEN);
```

J1.2.3.257 PredCountTest

```
// PredCountTest()
// ===============

func PredCountTest(elements : integer, count : integer, invert : boolean) => bits(4)
begin
    var n, z, c, v : bit;
    z = (if count == 0 then '1' else '0');              // none active
    if !invert then
        n = (if count != 0 then '1' else '0');          // first active
        c = (if count == elements then '0' else '1');   // NOT last active
    else
        n = (if count == elements then '1' else '0');   // first active
        c = (if count != 0 then '0' else '1');          // NOT last active
    end;
    v = '0';

    return n::z::c::v;
end;
```

J1.2.3.258 System

```
// System Registers
// ================

var _ZA : array [[256]] of bits(MAX_VL);
```

J1.2.3.259 ZAhslice

```
// ZAhslice - accessor
// ===================

accessor ZAhslice{width}(tile : integer, esize : ESize, slice : integer) <=> value : bits(width)
begin
    getter
        let tiles : integer = esize DIV 8;
        assert tile >= 0 && tile < tiles;
        let slices : integer = CurrentSVL() DIV esize;
        assert slice >= 0 && slice < slices;

        return ZAvector{width}(tile + slice * tiles);
    end;
    setter
        let tiles : integer = esize DIV 8;
        assert tile >= 0 && tile < tiles;
        let slices : integer = CurrentSVL() DIV esize;
        assert slice >= 0 && slice < slices;

        ZAvector{width}(tile + slice * tiles) = value;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.260 ZAslice

```
// ZAslice - accessor
// ==================

accessor ZAslice{width}(tile : integer, esize : ESize,
                        vertical : boolean, slice : integer) <=> value : bits(width)
begin
    getter
        var result : bits(width);

        if vertical then
            result = ZAvslice{width}(tile, esize, slice);
        else
            result = ZAhslice{width}(tile, esize, slice);
        end;

        return result;
    end;
    setter
        if vertical then
            ZAvslice{width}(tile, esize, slice) = value;
        else
            ZAhslice{width}(tile, esize, slice) = value;
        end;
    end;
end;
```

J1.2.3.261 ZAtile

```
// ZAtile - accessor
// =================

accessor ZAtile{width}(tile : integer, esize : ESize) <=> value : bits(width)
begin
    getter
        let SVL : VecLen = CurrentSVL();
        let slices : integer = SVL DIV esize;
        assert width == SVL * slices;
        var result : bits(width);

        for slice = 0 to slices-1 do
            result[slice*:SVL] = ZAhslice{SVL}(tile, esize, slice);
        end;

        return result;
    end;
    setter
        let SVL : VecLen = CurrentSVL();
        let slices : integer = SVL DIV esize;
        assert width == SVL * slices;

        for slice = 0 to slices-1 do
            ZAhslice{SVL}(tile, esize, slice) = value[slice*:SVL];
        end;
    end;
end;
```

J1.2.3.262 ZAvector

```
// ZAvector - accessor
// ===================

accessor ZAvector{width}(index : integer) <=> value : bits(width)
begin
    getter
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert width == CurrentSVL();
        assert index >= 0 && index < (width DIV 8);

        return _ZA[[index]][width-1:0];
    end;

    setter
        assert width == CurrentSVL();
        assert index >= 0 && index < (width DIV 8);

        if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
            _ZA[[index]] = ZeroExtend{MAX_VL}(value);
        else
            _ZA[[index]][width-1:0] = value;
        end;
    end;
end;
```

J1.2.3.263 ZAvslice

```
// ZAvslice - accessor
// ===================

accessor ZAvslice{width}(tile : integer, esize : ESize, slice : integer) <=> value : bits(width)
begin
    getter
        let slices : integer = CurrentSVL() DIV esize;
        var result : bits(width);

        for s = 0 to slices-1 do
            let hslice : bits(width) = ZAhslice{}(tile, esize, s);
            result[s*:esize] = hslice[slice*:esize];
        end;

        return result;
    end;

    setter
        let slices : integer = CurrentSVL() DIV esize;

        for s = 0 to slices-1 do
            var hslice : bits(width) = ZAhslice{}(tile, esize, s);
            hslice[slice*:esize] = value[s*:esize];
            ZAhslice{width}(tile, esize, s) = hslice;
        end;
    end;
end;
```

J1.2.3.264 ZT0

```
// ZT0 - accessor
// ==============

accessor ZT0{width}() <=> value : bits(width)
begin
    getter
        assert width == ZT0_LEN;
        return _ZT0[width-1:0];
    end;
    setter
        assert width == ZT0_LEN;
        _ZT0[width-1:0] = value;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.265 AArch32_IsFPEnabled

```
// AArch32_IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

func AArch32_IsFPEnabled(el : bits(2)) => boolean
begin
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64_IsFPEnabled(el);
    end;

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
        // Check if access disabled in NSACR
        if NSACR().cp10 == '0' then return FALSE; end;
    end;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        var disabled : boolean;
        case CPACR().cp10 of
            when '00' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '10' => disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64_IsFPEnabled(EL2);
        end;
        if HCPTR().TCP10 == '1' then return FALSE; end;
    end;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3().TFP == '1' then return FALSE; end;
    end;

    return TRUE;
end;
```

J1.2.3.266 AArch64_IsFPEnabled

```
// AArch64_IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

func AArch64_IsFPEnabled(el : bits(2)) => boolean
begin
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        var disabled : boolean;
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            var disabled : boolean;
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = el == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then return FALSE; end;
        else
            if CPTR_EL2().TFP == '1' then return FALSE; end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().TFP == '1' then return FALSE; end;
    end;

    return TRUE;
end;
```

J1.2.3.267 ActivePredicateElement

```
// ActivePredicateElement()
// ========================
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

func ActivePredicateElement{N}(pred : bits(N), e : integer, esize : integer) => boolean
begin
    assert esize IN {8, 16, 32, 64, 128};
    let n : integer = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred[n] == '1';
end;
```

J1.2.3.268 AllElementsActive

```
// AllElementsActive()
// ===================
// Return TRUE if all the elements are active in the mask. Otherwise,
// return FALSE.

func AllElementsActive{N}(mask : bits(N), esize : integer) => boolean
begin
    let elements : integer = N DIV (esize DIV 8);
    var active : integer = 0;
    for e = 0 to elements-1 do
        if ActivePredicateElement{N}(mask, e, esize) then active = active + 1; end;
    end;
    return active == elements;
end;
```

J1.2.3.269 AnyActiveElement

```
// AnyActiveElement()
// ==================
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

func AnyActiveElement{N}(mask : bits(N), esize : integer) => boolean
begin
    return LastActiveElement{N}(mask, esize) >= 0;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.270 BitDeposit

```
// BitDeposit()
// ============
// Deposit the least significant bits from DATA into result positions
// selected by nonzero bits in MASK, setting other result bits to zero.

func BitDeposit{N}(data : bits(N), mask : bits(N)) => bits(N)
begin
    var res : bits(N) = Zeros{};
    var db : integer = 0;
    for rb = 0 to N-1 do
        if mask[rb] == '1' then
            res[rb] = data[db];
            db      = db + 1;
        end;
    end;
    return res;
end;
```

J1.2.3.271 BitExtract

```
// BitExtract()
// ============
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, setting other result bits to zero.

func BitExtract{N}(data : bits(N), mask : bits(N)) => bits(N)
begin
    var res : bits(N) = Zeros{};
    var rb : integer = 0;
    for db = 0 to N-1 do
        if mask[db] == '1' then
            res[rb] = data[db];
            rb      = rb + 1;
        end;
    end;
    return res;
end;
```

J1.2.3.272 BitGroup

```
// BitGroup()
// ==========
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

func BitGroup{N}(data : bits(N), mask : bits(N)) => bits(N)
begin
    var res : bits(N);
    var rb : integer = 0;

    // compress masked bits to right
    for db = 0 to N-1 do
        if mask[db] == '1' then
            res[rb] = data[db];
            rb      = rb + 1;
        end;
    end;
    // compress unmasked bits to left
    for db = 0 to N-1 do
        if mask[db] == '0' then
            res[rb] = data[db];
            rb      = rb + 1;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    return res;
end;
```

J1.2.3.273 CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// ============================
// Checks for traps on SVE instructions that are not legal when executed in Streaming mode.

func CheckNonStreamingSVEEnabled()
begin
    CheckSVEEnabled();

    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
    end;
end;
```

J1.2.3.274 CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =========================
// Checks for traps on SVE instructions and instructions that access SVE System
// registers.

func CheckOriginalSVEEnabled()
begin
    assert IsFeatureImplemented(FEAT_SVE);
    var disabled : boolean;

    if (HaveEL(EL3) && (CPTR_EL3().EZ == '0' || CPTR_EL3().TFP == '1') &&
          EL3SDDUndefPriority()) then
        Undefined();
    end;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1().ZEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then SVEAccessTrap(EL1); end;

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch64_AdvSIMDFPAccessTrap(EL1); end;
    end;

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SVE at EL2
            case CPTR_EL2().ZEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then SVEAccessTrap(EL2); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            // Check SIMD&FP at EL2
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        else
            if CPTR_EL2().TZ == '1'  then SVEAccessTrap(EL2); end;
            if CPTR_EL2().TFP == '1' then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().EZ == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SVEAccessTrap(EL3);
        end;

        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then Undefined(); end;
            AArch64_AdvSIMDFPAccessTrap(EL3);
        end;
    end;
end;
```

J1.2.3.275 CheckSMEAccess

```
// CheckSMEAccess()
// ================
// Check that access to SME System registers is enabled.

func CheckSMEAccess()
begin
    var disabled : boolean;

    if HaveEL(EL3) && CPTR_EL3().ESM == '0' && EL3SDDUndefPriority() then
        Undefined();
    end;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1().SMEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1); end;
    end;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SME at EL2
            case CPTR_EL2().SMEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2); end;
        else
            if CPTR_EL2().TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2); end;
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().ESM == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
        end;
    end;
end;
```

J1.2.3.276 CheckSMEAndZAEnabled

```
// CheckSMEAndZAEnabled()
// ======================

func CheckSMEAndZAEnabled()
begin
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
    end;
end;
```

J1.2.3.277 CheckSMEEnabled

```
// CheckSMEEnabled()
// =================

func CheckSMEEnabled()
begin
    var disabled : boolean;

    if HaveEL(EL3) && CPTR_EL3().[ESM,TFP] != '10' && EL3SDDUndefPriority() then
        Undefined();
    end;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1().SMEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1); end;

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch64_AdvSIMDFPAccessTrap(EL1); end;
    end;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SME at EL2
            case CPTR_EL2().SMEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            // Check SIMD&FP at EL2
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        else
            if CPTR_EL2().TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2); end;
            if CPTR_EL2().TFP == '1' then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().ESM == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
        end;

        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then Undefined(); end;
            AArch64_AdvSIMDFPAccessTrap(EL3);
        end;
    end;
end;
```

J1.2.3.278 CheckSMEZT0Enabled

```
// CheckSMEZT0Enabled()
// ====================
// Checks for ZT0 enabled.

func CheckSMEZT0Enabled()
begin
    if HaveEL(EL3) && SMCR_EL3().EZT0 == '0' && EL3SDDUndefPriority() then
        Undefined();
    end;

    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
    end;

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        if SMCR_EL1().EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL1);
        end;
    end;

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2().EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL2);
        end;
    end;

    // Check if all accesses to ZT0 are disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3().EZT0 == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL3);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.3.279 CheckSVEEnabled

```
// CheckSVEEnabled()
// =================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

func CheckSVEEnabled()
begin
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        CheckSMEEnabled();
    elsif IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE) then
        CheckStreamingSVEEnabled();
    else
        CheckOriginalSVEEnabled();
    end;
end;
```

J1.2.3.280 CheckStreamingSVEAndZAEnabled

```
// CheckStreamingSVEAndZAEnabled()
// ===============================

func CheckStreamingSVEAndZAEnabled()
begin
    CheckStreamingSVEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
    end;
end;
```

J1.2.3.281 CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// ==========================

func CheckStreamingSVEEnabled()
begin
    CheckSMEEnabled();

    if PSTATE.SM == '0' then
        SMEAccessTrap(SMEExceptionType_NotStreaming, PSTATE.EL);
    end;
end;
```

J1.2.3.282 CmpOp

```
// CmpOp
// =====

type CmpOp of enumeration { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.283 CurrentNSVL

```
// CurrentNSVL
// ===========
// Non-Streaming VL

readonly func CurrentNSVL() => VecLen
begin
    var vl : integer;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1().LEN);
    end;

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2().LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2().LEN));
    end;

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3().LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3().LEN));
    end;

    return ImplementedSVEVectorLength((vl + 1) * 128);
end;
```

J1.2.3.284 CurrentSVL

```
// CurrentSVL
// ==========
// Streaming SVL

readonly func CurrentSVL() => VecLen
begin
    var vl : integer;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR_EL1().LEN);
    end;

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR_EL2().LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2().LEN));
    end;

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3().LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3().LEN));
    end;

    return ImplementedSMEVectorLength((vl + 1) * 128);
end;
```

J1.2.3.285 CurrentVL

```
// CurrentVL
// =========

readonly func CurrentVL() => VecLen
begin
    return (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then CurrentSVL()
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                    else CurrentNSVL());
end;
```

J1.2.3.286 DecodePredCount

```
// DecodePredCount()
// =================

func DecodePredCount(bitpattern : bits(5), esize : integer) => integer
begin
    let elements : integer = CurrentVL() DIV esize;
    var numElem : integer;
    case bitpattern of
        when '00000' => numElem = FloorPow2(elements);
        when '00001' => numElem = if elements >= 1 then 1 else 0;
        when '00010' => numElem = if elements >= 2 then 2 else 0;
        when '00011' => numElem = if elements >= 3 then 3 else 0;
        when '00100' => numElem = if elements >= 4 then 4 else 0;
        when '00101' => numElem = if elements >= 5 then 5 else 0;
        when '00110' => numElem = if elements >= 6 then 6 else 0;
        when '00111' => numElem = if elements >= 7 then 7 else 0;
        when '01000' => numElem = if elements >= 8 then 8 else 0;
        when '01001' => numElem = if elements >= 16 then 16 else 0;
        when '01010' => numElem = if elements >= 32 then 32 else 0;
        when '01011' => numElem = if elements >= 64 then 64 else 0;
        when '01100' => numElem = if elements >= 128 then 128 else 0;
        when '01101' => numElem = if elements >= 256 then 256 else 0;
        when '11101' => numElem = elements - (elements MOD 4);
        when '11110' => numElem = elements - (elements MOD 3);
        when '11111' => numElem = elements;
        otherwise    => numElem = 0;
    end;
    return numElem;
end;
```

J1.2.3.287 ElemFFR

```
// ElemFFR - accessor
// ==================

accessor ElemFFR(e : integer, esize : ESize) <=> value : bit
begin
    getter
        return PredicateElement{MAX_PL}(_FFR, e, esize);
    end;
    setter
        let psize : integer{} = esize DIV 8;
        _FFR[e*:psize] = ZeroExtend{psize}(value);
    end;
end;
```

J1.2.3.288 FFR

```
// FFR - accessor
// ==============

accessor FFR{width}() <=> value : bits(width)
begin
    getter
        assert width == CurrentVL() DIV 8;
        return _FFR[width-1:0];
    end;

    setter
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert width == CurrentVL() DIV 8;
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend{MAX_PL}(value);
        else
            _FFR[width-1:0] = value;
        end;
    end;
end;
```

J1.2.3.289 FPCompareNE

```
// FPCompareNE()
// =============

func FPCompareNE{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    var result : boolean;
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    let op1_nan : boolean = type1 IN {FPType_SNaN, FPType_QNaN};
    let op2_nan : boolean = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);
    end;
    return result;
end;
```

J1.2.3.290 FPCompareUN

```
// FPCompareUN()
// =============

func FPCompareUN{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    end;

    let result : boolean = (type1 IN {FPType_SNaN, FPType_QNaN} ||
                            type2 IN {FPType_SNaN, FPType_QNaN});
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);
    end;

    return result;
end;
```

J1.2.3.291 FPConvertSVE

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPConvertSVE()
// ==============

func FPConvertSVE{M, N}(op : bits(N), fpcr_in : FPCR_Type, rounding : FPRounding) => bits(M)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert{M, N}(op, fpcr, rounding);
end;

// FPConvertSVE()
// ==============

func FPConvertSVE{M, N}(op : bits(N), fpcr_in : FPCR_Type) => bits(M)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert{M, N}(op, fpcr, FPRoundingMode(fpcr));
end;
```

J1.2.3.292 FPExpA

```
// FPExpA()
// ========

func FPExpA{N}(op : bits(N)) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N);
    var coeff : bits(N);
    let idx : integer = if N == 16 then UInt(op[4:0]) else UInt(op[5:0]);
    coeff = FPExpCoefficient{N}(idx);
    if N == 16 then
        result[15:0] = '0'::op[9:5]::coeff[9:0];
    elsif N == 32 then
        result[31:0] = '0'::op[13:6]::coeff[22:0];
    else // N == 64
        result[63:0] = '0'::op[16:6]::coeff[51:0];
    end;

    return result;
end;
```

J1.2.3.293 FPExpCoefficient

```
// FPExpCoefficient()
// ==================

func FPExpCoefficient{N}(index : integer) => bits(N)
begin
    assert N IN {16,32,64};
    var result : integer;

    if N == 16 then
        case index of
            when  0 => result = 0x000;
            when  1 => result = 0x016;
            when  2 => result = 0x02d;
            when  3 => result = 0x045;
            when  4 => result = 0x05d;
            when  5 => result = 0x075;
            when  6 => result = 0x08e;
            when  7 => result = 0x0a8;
            when  8 => result = 0x0c2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when  9 => result = 0x0dc;
            when 10 => result = 0x0f8;
            when 11 => result = 0x114;
            when 12 => result = 0x130;
            when 13 => result = 0x14d;
            when 14 => result = 0x16b;
            when 15 => result = 0x189;
            when 16 => result = 0x1a8;
            when 17 => result = 0x1c8;
            when 18 => result = 0x1e8;
            when 19 => result = 0x209;
            when 20 => result = 0x22b;
            when 21 => result = 0x24e;
            when 22 => result = 0x271;
            when 23 => result = 0x295;
            when 24 => result = 0x2ba;
            when 25 => result = 0x2e0;
            when 26 => result = 0x306;
            when 27 => result = 0x32e;
            when 28 => result = 0x356;
            when 29 => result = 0x37f;
            when 30 => result = 0x3a9;
            when 31 => result = 0x3d4;
        end;

    elsif N == 32 then
        case index of
            when  0 => result = 0x000000;
            when  1 => result = 0x0164d2;
            when  2 => result = 0x02cd87;
            when  3 => result = 0x043a29;
            when  4 => result = 0x05aac3;
            when  5 => result = 0x071f62;
            when  6 => result = 0x08980f;
            when  7 => result = 0x0a14d5;
            when  8 => result = 0x0b95c2;
            when  9 => result = 0x0d1adf;
            when 10 => result = 0x0ea43a;
            when 11 => result = 0x1031dc;
            when 12 => result = 0x11c3d3;
            when 13 => result = 0x135a2b;
            when 14 => result = 0x14f4f0;
            when 15 => result = 0x16942d;
            when 16 => result = 0x1837f0;
            when 17 => result = 0x19e046;
            when 18 => result = 0x1b8d3a;
            when 19 => result = 0x1d3eda;
            when 20 => result = 0x1ef532;
            when 21 => result = 0x20b051;
            when 22 => result = 0x227043;
            when 23 => result = 0x243516;
            when 24 => result = 0x25fed7;
            when 25 => result = 0x27cd94;
            when 26 => result = 0x29a15b;
            when 27 => result = 0x2b7a3a;
            when 28 => result = 0x2d583f;
            when 29 => result = 0x2f3b79;
            when 30 => result = 0x3123f6;
            when 31 => result = 0x3311c4;
            when 32 => result = 0x3504f3;
            when 33 => result = 0x36fd92;
            when 34 => result = 0x38fbaf;
            when 35 => result = 0x3aff5b;
            when 36 => result = 0x3d08a4;
            when 37 => result = 0x3f179a;
            when 38 => result = 0x412c4d;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 39 => result = 0x4346cd;
            when 40 => result = 0x45672a;
            when 41 => result = 0x478d75;
            when 42 => result = 0x49b9be;
            when 43 => result = 0x4bec15;
            when 44 => result = 0x4e248c;
            when 45 => result = 0x506334;
            when 46 => result = 0x52a81e;
            when 47 => result = 0x54f35b;
            when 48 => result = 0x5744fd;
            when 49 => result = 0x599d16;
            when 50 => result = 0x5bfbb8;
            when 51 => result = 0x5e60f5;
            when 52 => result = 0x60ccdf;
            when 53 => result = 0x633f89;
            when 54 => result = 0x65b907;
            when 55 => result = 0x68396a;
            when 56 => result = 0x6ac0c7;
            when 57 => result = 0x6d4f30;
            when 58 => result = 0x6fe4ba;
            when 59 => result = 0x728177;
            when 60 => result = 0x75257d;
            when 61 => result = 0x77d0df;
            when 62 => result = 0x7a83b3;
            when 63 => result = 0x7d3e0c;
        end;

    else // N == 64
        case index of
            when  0 => result = 0x0000000000000;
            when  1 => result = 0x02C9A3E778061;
            when  2 => result = 0x059B0D3158574;
            when  3 => result = 0x0874518759BC8;
            when  4 => result = 0x0B5586CF9890F;
            when  5 => result = 0x0E3EC32D3D1A2;
            when  6 => result = 0x11301D0125B51;
            when  7 => result = 0x1429AAEA92DE0;
            when  8 => result = 0x172B83C7D517B;
            when  9 => result = 0x1A35BEB6FCB75;
            when 10 => result = 0x1D4873168B9AA;
            when 11 => result = 0x2063B88628CD6;
            when 12 => result = 0x2387A6E756238;
            when 13 => result = 0x26B4565E27CDD;
            when 14 => result = 0x29E9DF51FDEE1;
            when 15 => result = 0x2D285A6E4030B;
            when 16 => result = 0x306FE0A31B715;
            when 17 => result = 0x33C08B26416FF;
            when 18 => result = 0x371A7373AA9CB;
            when 19 => result = 0x3A7DB34E59FF7;
            when 20 => result = 0x3DEA64C123422;
            when 21 => result = 0x4160A21F72E2A;
            when 22 => result = 0x44E086061892D;
            when 23 => result = 0x486A2B5C13CD0;
            when 24 => result = 0x4BFDAD5362A27;
            when 25 => result = 0x4F9B2769D2CA7;
            when 26 => result = 0x5342B569D4F82;
            when 27 => result = 0x56F4736B527DA;
            when 28 => result = 0x5AB07DD485429;
            when 29 => result = 0x5E76F15AD2148;
            when 30 => result = 0x6247EB03A5585;
            when 31 => result = 0x6623882552225;
            when 32 => result = 0x6A09E667F3BCD;
            when 33 => result = 0x6DFB23C651A2F;
            when 34 => result = 0x71F75E8EC5F74;
            when 35 => result = 0x75FEB564267C9;
            when 36 => result = 0x7A11473EB0187;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 37 => result = 0x7E2F336CF4E62;
            when 38 => result = 0x82589994CCE13;
            when 39 => result = 0x868D99B4492ED;
            when 40 => result = 0x8ACE5422AA0DB;
            when 41 => result = 0x8F1AE99157736;
            when 42 => result = 0x93737B0CDC5E5;
            when 43 => result = 0x97D829FDE4E50;
            when 44 => result = 0x9C49182A3F090;
            when 45 => result = 0xA0C667B5DE565;
            when 46 => result = 0xA5503B23E255D;
            when 47 => result = 0xA9E6B5579FDBF;
            when 48 => result = 0xAE89F995AD3AD;
            when 49 => result = 0xB33A2B84F15FB;
            when 50 => result = 0xB7F76F2FB5E47;
            when 51 => result = 0xBCC1E904BC1D2;
            when 52 => result = 0xC199BDD85529C;
            when 53 => result = 0xC67F12E57D14B;
            when 54 => result = 0xCB720DCEF9069;
            when 55 => result = 0xD072D4A07897C;
            when 56 => result = 0xD5818DCFBA487;
            when 57 => result = 0xDA9E603DB3285;
            when 58 => result = 0xDFC97337B9B5F;
            when 59 => result = 0xE502EE78B3FF6;
            when 60 => result = 0xEA4AFA2A490DA;
            when 61 => result = 0xEFA1BEE615A27;
            when 62 => result = 0xF50765B6E4540;
            when 63 => result = 0xFA7C1819E90D8;
        end;
    end;

    return result[N-1:0];
end;
```

J1.2.3.294 FPLogB

```
// FPLogB()
// ========

func FPLogB{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var result : integer;
    var (fptype,sign,value) : (FPType, bit, real) = FPUnpack{N}(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN || fptype == FPType_Zero then
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = -(2^(N-1));            // MinInt, 100..00
    elsif fptype == FPType_Infinity then
        result = 2^(N-1) - 1;           // MaxInt, 011..11
    else
        // FPUnpack has already scaled a subnormal input
        value = Abs(value);
        (value, result) = NormalizeReal(value);

        FPProcessDenorm(fptype, N, fpcr);
    end;
    return result[N-1:0];
end;
```

J1.2.3.295 FPMinNormal

```
// FPMinNormal()
// =============

func FPMinNormal{N}(sign : bit) => bits(N)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = Zeros{E-1}::'1';
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;
```

J1.2.3.296 FPOne

```
// FPOne()
// =======

func FPOne{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = '0'::Ones{E-1};
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;
```

J1.2.3.297 FPPointFive

```
// FPPointFive()
// =============

func FPPointFive{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = '0'::Ones{E-2}::'0';
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;
```

J1.2.3.298 FPReducePredicated

```
// FPReducePredicated()
// ====================
func FPReducePredicated{esize, N, M}(op : ReduceOp, input : bits(N), mask : bits(M),
                                     identity : bits(esize), fpcr : FPCR_Type) => bits(esize)
begin
    assert(N == M * 8);
    assert IsPow2(N);
    var operand : bits(N);
    let elements : integer = N DIV esize;

    for e = 0 to elements-1 do
        if e * esize < N && ActivePredicateElement{M}(mask, e, esize) then
            operand[e*:esize] = input[e*:esize];
        else
            operand[e*:esize] = identity;
        end;
    end;

    return FPReduce{esize, N}(op, operand, fpcr);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.299 FPTrigMAdd

```
// FPTrigMAdd()
// ============

func FPTrigMAdd{N}(x_in : integer, op1 : bits(N), op2_in : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var coeff : bits(N);
    var op2 : bits(N) = op2_in;
    var x : integer = x_in;
    assert x >= 0;
    assert x < 8;

    if op2[N-1] == '1' then
        x = x + 8;
    end;

    coeff = FPTrigMAddCoefficient{N}(x);
    // Safer to use EffectiveFPCR() in case the input fpcr argument
    // is modified as opposed to actual value of FPCR

    op2                     = FPAbs{N}(op2, EffectiveFPCR());
    let result : bits(N)    = FPMulAdd{}(coeff, op1, op2, fpcr);
    return result;
end;
```

J1.2.3.300 FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =======================

func FPTrigMAddCoefficient{N}(index : integer) => bits(N)
begin
    assert N IN {16,32,64};
    var result : integer;

    if N == 16 then
        case index of
            when  0 => result = 0x3c00;
            when  1 => result = 0xb155;
            when  2 => result = 0x2030;
            when  3 => result = 0x0000;
            when  4 => result = 0x0000;
            when  5 => result = 0x0000;
            when  6 => result = 0x0000;
            when  7 => result = 0x0000;
            when  8 => result = 0x3c00;
            when  9 => result = 0xb800;
            when 10 => result = 0x293a;
            when 11 => result = 0x0000;
            when 12 => result = 0x0000;
            when 13 => result = 0x0000;
            when 14 => result = 0x0000;
            when 15 => result = 0x0000;
        end;
    elsif N == 32 then
        case index of
            when  0 => result = 0x3f800000;
            when  1 => result = 0xbe2aaaab;
            when  2 => result = 0x3c088886;
            when  3 => result = 0xb95008b9;
            when  4 => result = 0x36369d6d;
            when  5 => result = 0x00000000;
            when  6 => result = 0x00000000;
            when  7 => result = 0x00000000;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when  8 => result = 0x3f800000;
            when  9 => result = 0xbf000000;
            when 10 => result = 0x3d2aaaa6;
            when 11 => result = 0xbab60705;
            when 12 => result = 0x37cd37cc;
            when 13 => result = 0x00000000;
            when 14 => result = 0x00000000;
            when 15 => result = 0x00000000;
        end;
    else // N == 64
        case index of
            when  0 => result = 0x3ff0000000000000;
            when  1 => result = 0xbfc5555555555543;
            when  2 => result = 0x3f8111111110f30c;
            when  3 => result = 0xbf2a01a019b92fc6;
            when  4 => result = 0x3ec71de351f3d22b;
            when  5 => result = 0xbe5ae5e2b60f7b91;
            when  6 => result = 0x3de5d8408868552f;
            when  7 => result = 0x0000000000000000;
            when  8 => result = 0x3ff0000000000000;
            when  9 => result = 0xbfe0000000000000;
            when 10 => result = 0x3fa5555555555536;
            when 11 => result = 0xbf56c16c16c13a0b;
            when 12 => result = 0x3efa01a019b1e8d8;
            when 13 => result = 0xbe927e4f7282f468;
            when 14 => result = 0x3e21ee96d2641b13;
            when 15 => result = 0xbda8f76380fbb401;
        end;
    end;

    return result[N-1:0];
end;
```

J1.2.3.301 FPTrigSMul

```
// FPTrigSMul()
// ============

func FPTrigSMul{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N) = FPMul{}(op1, op1, fpcr);
    let fpexc = FALSE;
    let (fptype, sign, value) : (FPType, bit, real) = FPUnpack{N}(result, fpcr, fpexc);

    if ! fptype IN {FPType_QNaN, FPType_SNaN} then
        result[N-1] = op2[0];
    end;

    return result;
end;
```

J1.2.3.302 FPTrigSSel

```
// FPTrigSSel()
// ============

func FPTrigSSel{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N);

    if op2[0] == '1' then
        result = FPOne{N}(op2[1]);
    elsif op2[1] == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result = FPNeg{N}(op1, EffectiveFPCR());
    else
        result = op1;
    end;

    return result;
end;
```

J1.2.3.303 FirstActive

```
// FirstActive()
// =============

func FirstActive{N}(mask : bits(N), x : bits(N), esize : integer) => bit
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = 0 to elements-1 do
        if ActivePredicateElement{N}(mask, e, esize) then
            return PredicateElement{N}(x, e, esize);
        end;
    end;
    return '0';
end;
```

J1.2.3.304 HaveSVE2FP8DOT2

```
// HaveSVE2FP8DOT2()
// =================
// Returns TRUE if SVE2 FP8 dot product to half-precision instructions
// are implemented, FALSE otherwise.

func HaveSVE2FP8DOT2() => boolean
begin
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT2)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT2));
end;
```

J1.2.3.305 HaveSVE2FP8DOT4

```
// HaveSVE2FP8DOT4()
// =================
// Returns TRUE if SVE2 FP8 dot product to single-precision instructions
// are implemented, FALSE otherwise.

func HaveSVE2FP8DOT4() => boolean
begin
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT4)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT4));
end;
```

J1.2.3.306 HaveSVE2FP8FMA

```
// HaveSVE2FP8FMA()
// ================
// Returns TRUE if SVE2 FP8 multiply-accumulate to half-precision and single-precision
// instructions are implemented, FALSE otherwise.

func HaveSVE2FP8FMA() => boolean
begin
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8FMA)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8FMA));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.307 ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// ============================
// Reduce SVE/SME vector length to a supported value (power of two)

readonly func ImplementedSMEVectorLength(nbits_in : integer) => VecLen
begin
    let maxbits : VecLen = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    var nbits : integer = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && AlignDownSize(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 && !SupportedPowerTwoSVL(nbits) looplimit 7 do
        nbits = nbits - 128;
    end;

    // Return the smallest supported power-of-two VL
    while nbits < maxbits && !SupportedPowerTwoSVL(nbits) looplimit 7 do
        nbits = nbits * 2;
    end;

    return nbits as VecLen;
end;
```

J1.2.3.308 ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (power of two)

readonly func ImplementedSVEVectorLength(nbits_in : integer) => VecLen
begin
    let maxbits : integer = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    var nbits : integer = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && AlignDownSize(nbits, 128) == nbits;

    while !IsPow2(nbits) looplimit 7 do
        nbits = nbits - 128;
    end;
    return nbits as VecLen;
end;
```

J1.2.3.309 InStreamingMode

```
// InStreamingMode()
// =================

func InStreamingMode() => boolean
begin
    return IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1';
end;
```

J1.2.3.310 IntReducePredicated

```
// IntReducePredicated()
// =====================
func IntReducePredicated{esize, N, M}(op : ReduceOp, input : bits(N),
                                      mask : bits(M), identity : bits(esize)) => bits(esize)
begin
    assert(N == M * 8);
    assert IsPow2(N);
    var operand : bits(N);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let elements : integer = N DIV esize;

    for e = 0 to elements-1 do
        if e * esize < N && ActivePredicateElement{M}(mask, e, esize) then
            operand[e*:esize] = input[e*:esize];
        else
            operand[e*:esize] = identity;
        end;
    end;

    return IntReduce{esize, N}(op, operand);
end;
```

J1.2.3.311 IsFPEnabled

```
// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

func IsFPEnabled(el : bits(2)) => boolean
begin
    if ELUsingAArch32(el) then
        return AArch32_IsFPEnabled(el);
    else
        return AArch64_IsFPEnabled(el);
    end;
end;
```

J1.2.3.312 IsFullA64Enabled

```
// IsFullA64Enabled()
// ==================
// Returns TRUE if full A64 is enabled in Streaming mode and FALSE otherwise.

func IsFullA64Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SME_FA64) then return FALSE; end;

    // Check if full A64 disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full A64 at EL0/EL1
        if SMCR_EL1().FA64 == '0' then return FALSE; end;
    end;

    // Check if full A64 disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2().FA64 == '0' then return FALSE; end;
    end;

    // Check if full A64 disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3().FA64 == '0' then return FALSE; end;
    end;

    return TRUE;
end;
```

J1.2.3.313 IsOriginalSVEEnabled

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// IsOriginalSVEEnabled()
// ======================
// Returns TRUE if access to SVE functionality is enabled at the target
// exception level and FALSE otherwise.

func IsOriginalSVEEnabled(el : bits(2)) => boolean
begin
    var disabled : boolean;
    if ELUsingAArch32(el) then
        return FALSE;
    end;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1().ZEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            case CPTR_EL2().ZEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = el == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then return FALSE; end;
        else
            if CPTR_EL2().TZ == '1' then return FALSE; end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().EZ == '0' then return FALSE; end;
    end;

    return TRUE;
end;
```

J1.2.3.314 IsSMEEnabled

```
// IsSMEEnabled()
// ==============
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.

func IsSMEEnabled(el : bits(2)) => boolean
begin
    var disabled : boolean;
    if ELUsingAArch32(el) then
        return FALSE;
    end;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1().SMEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = el == EL0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            case CPTR_EL2().SMEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = el == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then return FALSE; end;
        else
            if CPTR_EL2().TSM == '1' then return FALSE; end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().ESM == '0' then return FALSE; end;
    end;

    return TRUE;
end;
```

J1.2.3.315 IsSVEEnabled

```
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE registers is enabled at the target exception
// level and FALSE otherwise.

func IsSVEEnabled(el : bits(2)) => boolean
begin
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elsif IsFeatureImplemented(FEAT_SVE) then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;
    end;
end;
```

J1.2.3.316 LastActive

```
// LastActive()
// ============

func LastActive{N}(mask : bits(N), x : bits(N), esize : integer) => bit
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = elements-1 downto 0 do
        if ActivePredicateElement{N}(mask, e, esize) then
            return PredicateElement{N}(x, e, esize);
        end;
    end;
    return '0';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.317 LastActiveElement

```
// LastActiveElement()
// ===================

func LastActiveElement{N}(mask : bits(N), esize : integer) => integer
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = elements-1 downto 0 do
        if ActivePredicateElement{N}(mask, e, esize) then return e; end;
    end;
    return -1;
end;
```

J1.2.3.318 MaxImplementedAnyVL

```
// MaxImplementedAnyVL()
// =====================

readonly func MaxImplementedAnyVL() => integer
begin
    if IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_SVE) then
        return Max(MaxImplementedVL(), MaxImplementedSVL());
    end;
    if IsFeatureImplemented(FEAT_SME) then
        return MaxImplementedSVL();
    end;
    return MaxImplementedVL();
end;
```

J1.2.3.319 MaxImplementedSVL

```
// MaxImplementedSVL()
// ===================

readonly func MaxImplementedSVL() => VecLen
begin
    return ImpDefInt("Max implemented SVL") as VecLen;
end;
```

J1.2.3.320 MaxImplementedVL

```
// MaxImplementedVL()
// ==================

readonly func MaxImplementedVL() => integer
begin
    return ImpDefInt("Max implemented VL");
end;
```

J1.2.3.321 MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// ====================

func MaybeZeroSVEUppers(target_el : bits(2))
begin
    var lower_enabled : boolean;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;
    end;

    if target_el == EL3 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if EL2Enabled() then
            lower_enabled = IsFPEnabled(EL2);
        else
            lower_enabled = IsFPEnabled(EL1);
        end;
    elsif target_el == EL2 then
        assert EL2Enabled() && !ELUsingAArch32(EL2);
        if HCR_EL2().TGE == '0' then
            lower_enabled = IsFPEnabled(EL1);
        else
            lower_enabled = IsFPEnabled(EL0);
        end;
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEnabled(EL0);
    end;

    if lower_enabled then
        let VL : integer{} = if IsSVEEnabled(PSTATE.EL) then CurrentVL() else 128;
        let PL : integer{} = VL DIV 8;
        for n = 0 to 31 do
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[[n]] = ZeroExtend{MAX_VL}(_Z[[n]][VL-1:0]);
            end;
        end;
        for n = 0 to 15 do
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _P[[n]] = ZeroExtend{MAX_PL}(_P[[n]][PL-1:0]);
            end;
        end;
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend{MAX_PL}(_FFR[PL-1:0]);
        end;
        if IsFeatureImplemented(FEAT_SME) && PSTATE.ZA == '1' then
            let SVL : integer{}          = CurrentSVL();
            let accessiblevecs : integer = SVL DIV 8;
            let allvecs : integer        = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1 do
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[[n]] = ZeroExtend{MAX_VL}(_ZA[[n]][SVL-1:0]);
                end;
            end;
            for n = accessiblevecs to allvecs - 1 do
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[[n]] = Zeros{MAX_VL};
                end;
            end;
        end;
    end;
end;
```

J1.2.3.322 MemNF

```
// MemNF
// =====

func MemNF{size : integer{8, 16, 32, 64, 128}}(address : bits(64),
                                                accdesc : AccessDescriptor
                                               ) => (bits(size), boolean)
begin
    let bytes : integer{} = size DIV 8;
    var value : bits(size);
    var bad : boolean;

    var aligned : boolean = IsAlignedSize{64}(address, bytes);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !aligned && AlignmentEnforced() then
        return (ARBITRARY : bits(size), TRUE);
    end;

    let atomic : boolean = aligned || bytes == 1;

    if !atomic then
        (value[7:0], bad) = MemSingleNF{8}(address, accdesc, aligned);

        if bad then
            return (ARBITRARY : bits(size), TRUE);
        end;

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.
        if !aligned then
            let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE; end;
        end;

        for i = 1 to bytes-1 do
            (value[i*:8], bad) = MemSingleNF{8}(address+i, accdesc, aligned);

            if bad then
                return (ARBITRARY : bits(size), TRUE);
            end;
        end;
    else
        (value, bad) = MemSingleNF{size}(address, accdesc, aligned);
        if bad then
            return (ARBITRARY : bits(size), TRUE);
        end;
    end;

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse{size}(value);
    end;

    return (value, FALSE);
end;
```

J1.2.3.323 MemSingleNF

```
// MemSingleNF
// ===========

func MemSingleNF{size : integer{8, 16, 32, 64, 128}}(address : bits(64),
                                                     accdesc_in : AccessDescriptor,
                                                     aligned : boolean) => (bits(size), boolean)
begin
    assert accdesc_in.acctype == AccessType_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.first);

    let bytes : integer{} = size DIV 8;
    var value : bits(size);
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var accdesc : AccessDescriptor = accdesc_in;
    var fault : FaultRecord = NoFault(accdesc, address);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
        return (ARBITRARY : bits(size), TRUE);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64_AccessIsTagChecked(address, accdesc);
    end;

    // MMU or MPU
    memaddrdesc = AArch64_TranslateAddress(address, accdesc, aligned, bytes);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType_Device then
        return (ARBITRARY : bits(size), TRUE);
    end;

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (ARBITRARY : bits(size), TRUE);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64_LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        fault = AArch64_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault_None then
            return (ARBITRARY : bits(size), TRUE);
        end;
    end;

    (memstatus, value) = PhysMemRead{size}(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        let iswrite : boolean = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, bytes, accdesc) then
            return (ARBITRARY : bits(size), TRUE);
        end;
        fault.merrorstate = memstatus.merrorstate;
        fault.extflag     = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);
    end;

    return (value, FALSE);
end;
```

J1.2.3.324 NoneActive

```
// NoneActive()
// ============

func NoneActive{N}(mask : bits(N), x : bits(N), esize : integer) => bit
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = 0 to elements-1 do
        if ActivePredicateElement{N}(mask, e, esize) && ActivePredicateElement{N}(x, e, esize) then
            return '0';
        end;
    end;
    return '1';
end;
```

J1.2.3.325 P

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// P - accessor
// ============

accessor P{width}(n : integer) <=> value : bits(width)
begin
    getter
        assert n >= 0 && n <= 31;
        assert width == CurrentVL() DIV 8;
        return _P[[n]][width-1:0];
    end;

    setter
        assert n >= 0 && n <= 31;
        assert width == CurrentVL() DIV 8;
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _P[[n]] = ZeroExtend{MAX_PL}(value);
        else
            _P[[n]][width-1:0] = value;
        end;
    end;
end;
```

J1.2.3.326 PredLen

```
// PredLen
// =======

type PredLen of integer{16, 32, 64, 128, 256};
```

J1.2.3.327 PredTest

```
// PredTest()
// ==========

func PredTest{N}(mask : bits(N), result : bits(N), esize : integer) => bits(4)
begin
    let n : bit = FirstActive{N}(mask, result, esize);
    let z : bit = NoneActive{N}(mask, result, esize);
    let c : bit = NOT LastActive{N}(mask, result, esize);
    let v : bit = '0';
    return n::z::c::v;
end;
```

J1.2.3.328 PredicateElement

```
// PredicateElement()
// ==================
// Returns the predicate bit

func PredicateElement{N}(pred : bits(N), e : integer, esize : integer) => bit
begin
    assert esize IN {8, 16, 32, 64, 128};
    let n : integer = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred[n];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.329 ResetSMEState

```
// ResetSMEState()
// ===============

func ResetSMEState(newenable : bit)
begin
    let vectors : integer = MAX_VL DIV 8;
    if newenable == '1' then
        for n = 0 to vectors - 1 do
            _ZA[[n]] = Zeros{MAX_VL};
        end;
        if IsFeatureImplemented(FEAT_SME2) then
            _ZT0 = Zeros{ZT0_LEN};
        end;
    else
        for n = 0 to vectors - 1 do
            _ZA[[n]] = ARBITRARY : bits(MAX_VL);
        end;
        if IsFeatureImplemented(FEAT_SME2) then
            _ZT0 = ARBITRARY : bits(ZT0_LEN);
        end;
    end;
end;
```

J1.2.3.330 ResetSVERegisters

```
// ResetSVERegisters()
// ===================

func ResetSVERegisters()
begin
    for n = 0 to 31 do
        _Z[[n]] = ARBITRARY : bits(MAX_VL);
    end;
    for n = 0 to 15 do
        _P[[n]] = ARBITRARY : bits(MAX_PL);
    end;
    _FFR = ARBITRARY : bits(MAX_PL);
end;
```

J1.2.3.331 ResetSVEState

```
// ResetSVEState()
// ===============

func ResetSVEState()
begin
    for n = 0 to 31 do
        _Z[[n]] = Zeros{MAX_VL};
    end;
    for n = 0 to 15 do
        _P[[n]] = Zeros{MAX_PL};
    end;
    _FFR = Zeros{MAX_PL};
    FPSR() = ZeroExtend{64}(0x0800009f[31:0]);
    FPMR() = Zeros{64};
end;
```

J1.2.3.332 SMEAccessTrap

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// SMEAccessTrap()
// ===============
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

func SMEAccessTrap(etype : SMEExceptionType, target_el_in : bits(2))
begin
    var target_el : bits(2) = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    end;
    let route_to_el2 : boolean = (PSTATE.EL == EL0 && target_el == EL1 &&
                                  EL2Enabled() && HCR_EL2().TGE == '1');

    var except : ExceptionRecord              = ExceptionSyndrome(Exception_SMEAccessTrap);
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer                 = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap =>
            except.syndrome.iss[2:0] = '000';
        when SMEExceptionType_Streaming =>
            except.syndrome.iss[2:0] = '001';
        when SMEExceptionType_NotStreaming =>
            except.syndrome.iss[2:0] = '010';
        when SMEExceptionType_InactiveZA =>
            except.syndrome.iss[2:0] = '011';
        when SMEExceptionType_InaccessibleZT0 =>
            except.syndrome.iss[2:0] = '100';
    end;

    if route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.3.333 SMEExceptionType

```
// SMEExceptionType
// ================
type SMEExceptionType of enumeration {
    SMEExceptionType_AccessTrap,        // SME functionality trapped or disabled
    SMEExceptionType_Streaming,         // Illegal instruction in Streaming SVE mode
    SMEExceptionType_NotStreaming,      // Illegal instruction not in Streaming SVE mode
    SMEExceptionType_InactiveZA,        // Illegal instruction when ZA is inactive
    SMEExceptionType_InaccessibleZT0,   // Access to ZT0 is disabled
};
```

J1.2.3.334 SVCR

```
// SVCR
// ====

accessor SVCR() <=> value : SVCR_Type
begin
    // Returns PSTATE.[ZA, SM]
    getter
        let value : SVCR_Type = Zeros{62} :: PSTATE.ZA :: PSTATE.SM;
        return value;
    end;

    // Sets PSTATE.[ZA, SM]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    setter
        SetPSTATE_SM(value[0]);
        SetPSTATE_ZA(value[1]);
        return;
    end;
end;
```

J1.2.3.335 SVEAccessTrap

```
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

func SVEAccessTrap(target_el : bits(2))
begin
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    let route_to_el2 : boolean = target_el == EL1 && EL2Enabled() && HCR_EL2().TGE == '1';

    let except : ExceptionRecord              = ExceptionSyndrome(Exception_SVEAccessTrap);
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer{}               = 0x0;

    if route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;
end;
```

J1.2.3.336 SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

func SVEMoveMaskPreferred(imm13 : bits(13)) => boolean
begin
    var imm : bits(64);
    (imm, -) = DecodeBitMasks{64}(imm13[12], imm13[5:0], imm13[11:6], TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm[7:0]) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm[63:7]) || IsOnes(imm[63:7]) then
            return FALSE;
        end;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm[63:32] == imm[31:0] && (IsZero(imm[31:7]) || IsOnes(imm[31:7])) then
            return FALSE;
        end;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if (imm[63:32] == imm[31:0] && imm[31:16] == imm[15:0] &&
              (IsZero(imm[15:7]) || IsOnes(imm[15:7]))) then
            return FALSE;
        end;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm[63:32] == imm[31:0] && imm[31:16] == imm[15:0] && (imm[15:8] == imm[7:0]) then
            return FALSE;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm[63:15]) || IsOnes(imm[63:15]) then
            return FALSE;
        end;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm[63:32] == imm[31:0] && (IsZero(imm[31:7]) || IsOnes(imm[31:7])) then
            return FALSE;
        end;

        // Check for 'xy00xy00xy00xy00'
        if imm[63:32] == imm[31:0] && imm[31:16] == imm[15:0] then
            return FALSE;
        end;
    end;

    return TRUE;
end;
```

J1.2.3.337 SetPSTATE_SM

```
// SetPSTATE_SM()
// ==============

func SetPSTATE_SM(value : bit)
begin
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
    end;
end;
```

J1.2.3.338 SetPSTATE_ZA

```
// SetPSTATE_ZA()
// ==============

func SetPSTATE_ZA(value : bit)
begin
    if PSTATE.ZA != value then
        ResetSMEState(value);
        PSTATE.ZA = value;
    end;
end;
```

J1.2.3.339 SupportedPowerTwoSVL

```
// SupportedPowerTwoSVL()
// ======================
// Return an IMPLEMENTATION DEFINED specific value
// returns TRUE if SVL is supported and is a power of two, FALSE otherwise

readonly impdef func SupportedPowerTwoSVL(nbits : integer) => boolean
begin
    return IsPow2(nbits) && (nbits <= MAX_VL);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.340 System

```
// System Registers
// ================

constant MAX_VL : VecLen = 2048;
constant MAX_PL : PredLen = 256;
constant ZT0_LEN : integer{} = 512;
var _Z : array [[32]] of bits(MAX_VL);
var _P : array [[16]] of bits(MAX_PL);
var _FFR : bits(MAX_PL);
```

J1.2.3.341 VecLen

```
// VecLen
// ======

type VecLen of integer{128, 256, 512, 1024, 2048};
```

J1.2.3.342 Z

```
// Z - accessor
// ============

accessor Z{width}(n : integer) <=> value : bits(width)
begin
    getter
        assert n >= 0 && n <= 31;
        assert width == CurrentVL();
        return _Z[[n]][width-1:0];
    end;
    setter
        assert n >= 0 && n <= 31;
        assert width == CurrentVL();
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _Z[[n]] = ZeroExtend{MAX_VL}(value);
        else
            _Z[[n]][width-1:0] = value;
        end;
    end;
end;
```

J1.2.3.343 SystemHintOp

```
// SystemHintOp
// ============
// System Hint instruction types.

type SystemHintOp of enumeration {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CLRBHB,
    SystemHintOp_GCSB,
    SystemHintOp_CHKFEAT,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SystemHintOp_STSHH,
    SystemHintOp_CSDB
};
```

J1.2.3.344 SysLOp

```
// SysLOp()
// ========

func SysLOp(op1 : bits(3), CRn : bits(4), CRm : bits(4), op2 : bits(3)) => SystemLOp
begin
    return Sysl_SYSL;
end;
```

J1.2.3.345 SystemLOp

```
// SystemLOp
// =========
// System instruction with result types.

type SystemLOp of enumeration {
    Sysl_SYSL
};
```

J1.2.3.346 SysOp

```
// SysOp()
// =======

func SysOp(op1 : bits(3), CRn : bits(4), CRm : bits(4), op2 : bits(3)) => SystemOp
begin
    case op1::CRn::CRm::op2 of
        when '000 0111 1000 000' => return Sys_AT;    // S1E1R
        when '000 0111 1000 001' => return Sys_AT;    // S1E1W
        when '000 0111 1000 010' => return Sys_AT;    // S1E0R
        when '000 0111 1000 011' => return Sys_AT;    // S1E0W
        when '000 0111 1001 000' => return Sys_AT;    // S1E1RP
        when '000 0111 1001 001' => return Sys_AT;    // S1E1WP
        when '000 0111 1001 010' => return Sys_AT;    // S1E1A
        when '100 0111 1000 000' => return Sys_AT;    // S1E2R
        when '100 0111 1000 001' => return Sys_AT;    // S1E2W
        when '100 0111 1001 010' => return Sys_AT;    // S1E2A
        when '100 0111 1000 100' => return Sys_AT;    // S12E1R
        when '100 0111 1000 101' => return Sys_AT;    // S12E1W
        when '100 0111 1000 110' => return Sys_AT;    // S12E0R
        when '100 0111 1000 111' => return Sys_AT;    // S12E0W
        when '110 0111 1000 000' => return Sys_AT;    // S1E3R
        when '110 0111 1000 001' => return Sys_AT;    // S1E3W
        when '110 0111 1001 010' => return Sys_AT;    // S1E3A
        when '001 0111 0010 100' => return Sys_BRB;   // IALL
        when '001 0111 0010 101' => return Sys_BRB;   // INJ
        when '000 0111 0110 001' => return Sys_DC;    // IVAC
        when '000 0111 0110 010' => return Sys_DC;    // ISW
        when '000 0111 0110 011' => return Sys_DC;    // IGVAC
        when '000 0111 0110 100' => return Sys_DC;    // IGSW
        when '000 0111 0110 101' => return Sys_DC;    // IGDVAC
        when '000 0111 0110 110' => return Sys_DC;    // IGDSW
        when '000 0111 1010 010' => return Sys_DC;    // CSW
        when '000 0111 1010 100' => return Sys_DC;    // CGSW
        when '000 0111 1010 110' => return Sys_DC;    // CGDSW
        when '000 0111 1110 010' => return Sys_DC;    // CISW
        when '000 0111 1110 100' => return Sys_DC;    // CIGSW
        when '000 0111 1110 110' => return Sys_DC;    // CIGDSW
        when '011 0111 0100 001' => return Sys_DC;    // ZVA
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '011 0111 0100 011' => return Sys_DC;    // GVA
when '011 0111 0100 100' => return Sys_DC;    // GZVA
when '011 0111 1010 001' => return Sys_DC;    // CVAC
when '011 0111 1010 011' => return Sys_DC;    // CGVAC
when '011 0111 1010 101' => return Sys_DC;    // CGDVAC
when '011 0111 1011 001' => return Sys_DC;    // CVAU
when '011 0111 1100 001' => return Sys_DC;    // CVAP
when '011 0111 1100 011' => return Sys_DC;    // CGVAP
when '011 0111 1100 101' => return Sys_DC;    // CGDVAP
when '011 0111 1101 001' => return Sys_DC;    // CVADP
when '011 0111 1101 011' => return Sys_DC;    // CGVADP
when '011 0111 1101 101' => return Sys_DC;    // CGDVADP
when '011 0111 1110 001' => return Sys_DC;    // CIVAC
when '011 0111 1110 011' => return Sys_DC;    // CIGVAC
when '011 0111 1110 101' => return Sys_DC;    // CIGDVAC
when '100 0111 1110 000' => return Sys_DC;    // CIPAE
when '100 0111 1110 111' => return Sys_DC;    // CIGDPAE
when '110 0111 1110 001' => return Sys_DC;    // CIPAPA
when '110 0111 1110 101' => return Sys_DC;    // CIGDPAPA
when '000 0111 1111 001' => return Sys_DC;    // CIVAPS
when '000 0111 1111 101' => return Sys_DC;    // CIGDVAPS
when '000 0111 0001 000' => return Sys_IC;    // IALLUIS
when '000 0111 0101 000' => return Sys_IC;    // IALLU
when '011 0111 0101 001' => return Sys_IC;    // IVAU
when '000 1000 0001 000' => return Sys_TLBI;  // VMALLE1OS
when '000 1000 0001 001' => return Sys_TLBI;  // VAE1OS
when '000 1000 0001 010' => return Sys_TLBI;  // ASIDE1OS
when '000 1000 0001 011' => return Sys_TLBI;  // VAAE1OS
when '000 1000 0001 101' => return Sys_TLBI;  // VALE1OS
when '000 1000 0001 111' => return Sys_TLBI;  // VAALE1OS
when '000 1000 0010 001' => return Sys_TLBI;  // RVAE1IS
when '000 1000 0010 011' => return Sys_TLBI;  // RVAAE1IS
when '000 1000 0010 101' => return Sys_TLBI;  // RVALE1IS
when '000 1000 0010 111' => return Sys_TLBI;  // RVAALE1IS
when '000 1000 0011 000' => return Sys_TLBI;  // VMALLE1IS
when '000 1000 0011 001' => return Sys_TLBI;  // VAE1IS
when '000 1000 0011 010' => return Sys_TLBI;  // ASIDE1IS
when '000 1000 0011 011' => return Sys_TLBI;  // VAAE1IS
when '000 1000 0011 101' => return Sys_TLBI;  // VALE1IS
when '000 1000 0011 111' => return Sys_TLBI;  // VAALE1IS
when '000 1000 0101 001' => return Sys_TLBI;  // RVAE1OS
when '000 1000 0101 011' => return Sys_TLBI;  // RVAAE1OS
when '000 1000 0101 101' => return Sys_TLBI;  // RVALE1OS
when '000 1000 0101 111' => return Sys_TLBI;  // RVAALE1OS
when '000 1000 0110 001' => return Sys_TLBI;  // RVAE1
when '000 1000 0110 011' => return Sys_TLBI;  // RVAAE1
when '000 1000 0110 101' => return Sys_TLBI;  // RVALE1
when '000 1000 0110 111' => return Sys_TLBI;  // RVAALE1
when '000 1000 0111 000' => return Sys_TLBI;  // VMALLE1
when '000 1000 0111 001' => return Sys_TLBI;  // VAE1
when '000 1000 0111 010' => return Sys_TLBI;  // ASIDE1
when '000 1000 0111 011' => return Sys_TLBI;  // VAAE1
when '000 1000 0111 101' => return Sys_TLBI;  // VALE1
when '000 1000 0111 111' => return Sys_TLBI;  // VAALE1
when '000 1001 0001 000' => return Sys_TLBI;  // VMALLE1OSNXS
when '000 1001 0001 001' => return Sys_TLBI;  // VAE1OSNXS
when '000 1001 0001 010' => return Sys_TLBI;  // ASIDE1OSNXS
when '000 1001 0001 011' => return Sys_TLBI;  // VAAE1OSNXS
when '000 1001 0001 101' => return Sys_TLBI;  // VALE1OSNXS
when '000 1001 0001 111' => return Sys_TLBI;  // VAALE1OSNXS
when '000 1001 0010 001' => return Sys_TLBI;  // RVAE1ISNXS
when '000 1001 0010 011' => return Sys_TLBI;  // RVAAE1ISNXS
when '000 1001 0010 101' => return Sys_TLBI;  // RVALE1ISNXS
when '000 1001 0010 111' => return Sys_TLBI;  // RVAALE1ISNXS
when '000 1001 0011 000' => return Sys_TLBI;  // VMALLE1ISNXS
when '000 1001 0011 001' => return Sys_TLBI;  // VAE1ISNXS
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '000 1001 0011 010' => return Sys_TLBI;  // ASIDE1ISNXS
when '000 1001 0011 011' => return Sys_TLBI;  // VAAE1ISNXS
when '000 1001 0011 101' => return Sys_TLBI;  // VALE1ISNXS
when '000 1001 0011 111' => return Sys_TLBI;  // VAALE1ISNXS
when '000 1001 0101 001' => return Sys_TLBI;  // RVAE1OSNXS
when '000 1001 0101 011' => return Sys_TLBI;  // RVAAE1OSNXS
when '000 1001 0101 101' => return Sys_TLBI;  // RVALE1OSNXS
when '000 1001 0101 111' => return Sys_TLBI;  // RVAALE1OSNXS
when '000 1001 0110 001' => return Sys_TLBI;  // RVAE1NXS
when '000 1001 0110 011' => return Sys_TLBI;  // RVAAE1NXS
when '000 1001 0110 101' => return Sys_TLBI;  // RVALE1NXS
when '000 1001 0110 111' => return Sys_TLBI;  // RVAALE1NXS
when '000 1001 0111 000' => return Sys_TLBI;  // VMALLE1NXS
when '000 1001 0111 001' => return Sys_TLBI;  // VAE1NXS
when '000 1001 0111 010' => return Sys_TLBI;  // ASIDE1NXS
when '000 1001 0111 011' => return Sys_TLBI;  // VAAE1NXS
when '000 1001 0111 101' => return Sys_TLBI;  // VALE1NXS
when '000 1001 0111 111' => return Sys_TLBI;  // VAALE1NXS
when '100 1000 0000 001' => return Sys_TLBI;  // IPAS2E1IS
when '100 1000 0000 010' => return Sys_TLBI;  // RIPAS2E1IS
when '100 1000 0000 101' => return Sys_TLBI;  // IPAS2LE1IS
when '100 1000 0000 110' => return Sys_TLBI;  // RIPAS2LE1IS
when '100 1000 0001 000' => return Sys_TLBI;  // ALLE2OS
when '100 1000 0001 001' => return Sys_TLBI;  // VAE2OS
when '100 1000 0001 100' => return Sys_TLBI;  // ALLE1OS
when '100 1000 0001 101' => return Sys_TLBI;  // VALE2OS
when '100 1000 0001 110' => return Sys_TLBI;  // VMALLS12E1OS
when '100 1000 0010 001' => return Sys_TLBI;  // RVAE2IS
when '100 1000 0010 101' => return Sys_TLBI;  // RVALE2IS
when '100 1000 0011 000' => return Sys_TLBI;  // ALLE2IS
when '100 1000 0011 001' => return Sys_TLBI;  // VAE2IS
when '100 1000 0011 100' => return Sys_TLBI;  // ALLE1IS
when '100 1000 0011 101' => return Sys_TLBI;  // VALE2IS
when '100 1000 0011 110' => return Sys_TLBI;  // VMALLS12E1IS
when '100 1000 0100 000' => return Sys_TLBI;  // IPAS2E1OS
when '100 1000 0100 001' => return Sys_TLBI;  // IPAS2E1
when '100 1000 0100 010' => return Sys_TLBI;  // RIPAS2E1
when '100 1000 0100 011' => return Sys_TLBI;  // RIPAS2E1OS
when '100 1000 0100 100' => return Sys_TLBI;  // IPAS2LE1OS
when '100 1000 0100 101' => return Sys_TLBI;  // IPAS2LE1
when '100 1000 0100 110' => return Sys_TLBI;  // RIPAS2LE1
when '100 1000 0100 111' => return Sys_TLBI;  // RIPAS2LE1OS
when '100 1000 0101 001' => return Sys_TLBI;  // RVAE2OS
when '100 1000 0101 101' => return Sys_TLBI;  // RVALE2OS
when '100 1000 0110 001' => return Sys_TLBI;  // RVAE2
when '100 1000 0110 101' => return Sys_TLBI;  // RVALE2
when '100 1000 0111 000' => return Sys_TLBI;  // ALLE2
when '100 1000 0111 001' => return Sys_TLBI;  // VAE2
when '100 1000 0111 100' => return Sys_TLBI;  // ALLE1
when '100 1000 0111 101' => return Sys_TLBI;  // VALE2
when '100 1000 0111 110' => return Sys_TLBI;  // VMALLS12E1
when '100 1001 0000 001' => return Sys_TLBI;  // IPAS2E1ISNXS
when '100 1001 0000 010' => return Sys_TLBI;  // RIPAS2E1ISNXS
when '100 1001 0000 101' => return Sys_TLBI;  // IPAS2LE1ISNXS
when '100 1001 0000 110' => return Sys_TLBI;  // RIPAS2LE1ISNXS
when '100 1001 0001 000' => return Sys_TLBI;  // ALLE2OSNXS
when '100 1001 0001 001' => return Sys_TLBI;  // VAE2OSNXS
when '100 1001 0001 100' => return Sys_TLBI;  // ALLE1OSNXS
when '100 1001 0001 101' => return Sys_TLBI;  // VALE2OSNXS
when '100 1001 0001 110' => return Sys_TLBI;  // VMALLS12E1OSNXS
when '100 1001 0010 001' => return Sys_TLBI;  // RVAE2ISNXS
when '100 1001 0010 101' => return Sys_TLBI;  // RVALE2ISNXS
when '100 1001 0011 000' => return Sys_TLBI;  // ALLE2ISNXS
when '100 1001 0011 001' => return Sys_TLBI;  // VAE2ISNXS
when '100 1001 0011 100' => return Sys_TLBI;  // ALLE1ISNXS
when '100 1001 0011 101' => return Sys_TLBI;  // VALE2ISNXS
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '100 1001 0011 110' => return Sys_TLBI;  // VMALLS12E1ISNXS
        when '100 1001 0100 000' => return Sys_TLBI;  // IPAS2E1OSNXS
        when '100 1001 0100 001' => return Sys_TLBI;  // IPAS2E1NXS
        when '100 1001 0100 010' => return Sys_TLBI;  // RIPAS2E1NXS
        when '100 1001 0100 011' => return Sys_TLBI;  // RIPAS2E1OSNXS
        when '100 1001 0100 100' => return Sys_TLBI;  // IPAS2LE1OSNXS
        when '100 1001 0100 101' => return Sys_TLBI;  // IPAS2LE1NXS
        when '100 1001 0100 110' => return Sys_TLBI;  // RIPAS2LE1NXS
        when '100 1001 0100 111' => return Sys_TLBI;  // RIPAS2LE1OSNXS
        when '100 1001 0101 001' => return Sys_TLBI;  // RVAE2OSNXS
        when '100 1001 0101 101' => return Sys_TLBI;  // RVALE2OSNXS
        when '100 1001 0110 001' => return Sys_TLBI;  // RVAE2NXS
        when '100 1001 0110 101' => return Sys_TLBI;  // RVALE2NXS
        when '100 1001 0111 000' => return Sys_TLBI;  // ALLE2NXS
        when '100 1001 0111 001' => return Sys_TLBI;  // VAE2NXS
        when '100 1001 0111 100' => return Sys_TLBI;  // ALLE1NXS
        when '100 1001 0111 101' => return Sys_TLBI;  // VALE2NXS
        when '100 1001 0111 110' => return Sys_TLBI;  // VMALLS12E1NXS
        when '110 1000 0001 000' => return Sys_TLBI;  // ALLE3OS
        when '110 1000 0001 001' => return Sys_TLBI;  // VAE3OS
        when '110 1000 0001 100' => return Sys_TLBI;  // PAALLOS
        when '110 1000 0001 101' => return Sys_TLBI;  // VALE3OS
        when '110 1000 0010 001' => return Sys_TLBI;  // RVAE3IS
        when '110 1000 0010 101' => return Sys_TLBI;  // RVALE3IS
        when '110 1000 0011 000' => return Sys_TLBI;  // ALLE3IS
        when '110 1000 0011 001' => return Sys_TLBI;  // VAE3IS
        when '110 1000 0011 101' => return Sys_TLBI;  // VALE3IS
        when '110 1000 0100 011' => return Sys_TLBI;  // RPAOS
        when '110 1000 0100 111' => return Sys_TLBI;  // RPALOS
        when '110 1000 0101 001' => return Sys_TLBI;  // RVAE3OS
        when '110 1000 0101 101' => return Sys_TLBI;  // RVALE3OS
        when '110 1000 0110 001' => return Sys_TLBI;  // RVAE3
        when '110 1000 0110 101' => return Sys_TLBI;  // RVALE3
        when '110 1000 0111 000' => return Sys_TLBI;  // ALLE3
        when '110 1000 0111 001' => return Sys_TLBI;  // VAE3
        when '110 1000 0111 100' => return Sys_TLBI;  // PAALL
        when '110 1000 0111 101' => return Sys_TLBI;  // VALE3
        when '110 1001 0001 000' => return Sys_TLBI;  // ALLE3OSNXS
        when '110 1001 0001 001' => return Sys_TLBI;  // VAE3OSNXS
        when '110 1001 0001 101' => return Sys_TLBI;  // VALE3OSNXS
        when '110 1001 0010 001' => return Sys_TLBI;  // RVAE3ISNXS
        when '110 1001 0010 101' => return Sys_TLBI;  // RVALE3ISNXS
        when '110 1001 0011 000' => return Sys_TLBI;  // ALLE3ISNXS
        when '110 1001 0011 001' => return Sys_TLBI;  // VAE3ISNXS
        when '110 1001 0011 101' => return Sys_TLBI;  // VALE3ISNXS
        when '110 1001 0101 001' => return Sys_TLBI;  // RVAE3OSNXS
        when '110 1001 0101 101' => return Sys_TLBI;  // RVALE3OSNXS
        when '110 1001 0110 001' => return Sys_TLBI;  // RVAE3NXS
        when '110 1001 0110 101' => return Sys_TLBI;  // RVALE3NXS
        when '110 1001 0111 000' => return Sys_TLBI;  // ALLE3NXS
        when '110 1001 0111 001' => return Sys_TLBI;  // VAE3NXS
        when '110 1001 0111 101' => return Sys_TLBI;  // VALE3NXS
        otherwise =>                return Sys_SYS;
    end;
end;
```

J1.2.3.347 SystemOp

```
// SystemOp
// ========
// System instruction types.

type SystemOp of enumeration {
    Sys_AT,
    Sys_BRB,
    Sys_DC,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Sys_IC,
    Sys_TLBI,
    Sys_SYS
};
```

J1.2.3.348 SysOp128

```
// SysOp128()
// ==========

func SysOp128(op1 : bits(3), CRn : bits(4), CRm : bits(4), op2 : bits(3)) => SystemOp128
begin
    case op1::CRn::CRm::op2 of
        when '000 1000 0001 001' => return Sys_TLBIP;  // VAE1OS
        when '000 1000 0001 011' => return Sys_TLBIP;  // VAAE1OS
        when '000 1000 0001 101' => return Sys_TLBIP;  // VALE1OS
        when '000 1000 0001 111' => return Sys_TLBIP;  // VAALE1OS
        when '000 1000 0011 001' => return Sys_TLBIP;  // VAE1IS
        when '000 1000 0011 011' => return Sys_TLBIP;  // VAAE1IS
        when '000 1000 0011 101' => return Sys_TLBIP;  // VALE1IS
        when '000 1000 0011 111' => return Sys_TLBIP;  // VAALE1IS
        when '000 1000 0111 001' => return Sys_TLBIP;  // VAE1
        when '000 1000 0111 011' => return Sys_TLBIP;  // VAAE1
        when '000 1000 0111 101' => return Sys_TLBIP;  // VALE1
        when '000 1000 0111 111' => return Sys_TLBIP;  // VAALE1
        when '000 1001 0001 001' => return Sys_TLBIP;  // VAE1OSNXS
        when '000 1001 0001 011' => return Sys_TLBIP;  // VAAE1OSNXS
        when '000 1001 0001 101' => return Sys_TLBIP;  // VALE1OSNXS
        when '000 1001 0001 111' => return Sys_TLBIP;  // VAALE1OSNXS
        when '000 1001 0011 001' => return Sys_TLBIP;  // VAE1ISNXS
        when '000 1001 0011 011' => return Sys_TLBIP;  // VAAE1ISNXS
        when '000 1001 0011 101' => return Sys_TLBIP;  // VALE1ISNXS
        when '000 1001 0011 111' => return Sys_TLBIP;  // VAALE1ISNXS
        when '000 1001 0111 001' => return Sys_TLBIP;  // VAE1NXS
        when '000 1001 0111 011' => return Sys_TLBIP;  // VAAE1NXS
        when '000 1001 0111 101' => return Sys_TLBIP;  // VALE1NXS
        when '000 1001 0111 111' => return Sys_TLBIP;  // VAALE1NXS
        when '100 1000 0001 001' => return Sys_TLBIP;  // VAE2OS
        when '100 1000 0001 101' => return Sys_TLBIP;  // VALE2OS
        when '100 1000 0011 001' => return Sys_TLBIP;  // VAE2IS
        when '100 1000 0011 101' => return Sys_TLBIP;  // VALE2IS
        when '100 1000 0111 001' => return Sys_TLBIP;  // VAE2
        when '100 1000 0111 101' => return Sys_TLBIP;  // VALE2
        when '100 1001 0001 001' => return Sys_TLBIP;  // VAE2OSNXS
        when '100 1001 0001 101' => return Sys_TLBIP;  // VALE2OSNXS
        when '100 1001 0011 001' => return Sys_TLBIP;  // VAE2ISNXS
        when '100 1001 0011 101' => return Sys_TLBIP;  // VALE2ISNXS
        when '100 1001 0111 001' => return Sys_TLBIP;  // VAE2NXS
        when '100 1001 0111 101' => return Sys_TLBIP;  // VALE2NXS
        when '110 1000 0001 001' => return Sys_TLBIP;  // VAE3OS
        when '110 1000 0001 101' => return Sys_TLBIP;  // VALE3OS
        when '110 1000 0011 001' => return Sys_TLBIP;  // VAE3IS
        when '110 1000 0011 101' => return Sys_TLBIP;  // VALE3IS
        when '110 1000 0111 001' => return Sys_TLBIP;  // VAE3
        when '110 1000 0111 101' => return Sys_TLBIP;  // VALE3
        when '110 1001 0001 001' => return Sys_TLBIP;  // VAE3OSNXS
        when '110 1001 0001 101' => return Sys_TLBIP;  // VALE3OSNXS
        when '110 1001 0011 001' => return Sys_TLBIP;  // VAE3ISNXS
        when '110 1001 0011 101' => return Sys_TLBIP;  // VALE3ISNXS
        when '110 1001 0111 001' => return Sys_TLBIP;  // VAE3NXS
        when '110 1001 0111 101' => return Sys_TLBIP;  // VALE3NXS
        when '100 1000 0000 001' => return Sys_TLBIP;  // IPAS2E1IS
        when '100 1000 0000 101' => return Sys_TLBIP;  // IPAS2LE1IS
        when '100 1000 0100 000' => return Sys_TLBIP;  // IPAS2E1OS
        when '100 1000 0100 001' => return Sys_TLBIP;  // IPAS2E1
        when '100 1000 0100 100' => return Sys_TLBIP;  // IPAS2LE1OS
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '100 1000 0100 101' => return Sys_TLBIP;  // IPAS2LE1
when '100 1001 0000 001' => return Sys_TLBIP;  // IPAS2E1ISNXS
when '100 1001 0000 101' => return Sys_TLBIP;  // IPAS2LE1ISNXS
when '100 1001 0100 000' => return Sys_TLBIP;  // IPAS2E1OSNXS
when '100 1001 0100 001' => return Sys_TLBIP;  // IPAS2E1NXS
when '100 1001 0100 100' => return Sys_TLBIP;  // IPAS2LE1OSNXS
when '100 1001 0100 101' => return Sys_TLBIP;  // IPAS2LE1NXS
when '000 1000 0010 001' => return Sys_TLBIP;  // RVAE1IS
when '000 1000 0010 011' => return Sys_TLBIP;  // RVAAE1IS
when '000 1000 0010 101' => return Sys_TLBIP;  // RVALE1IS
when '000 1000 0010 111' => return Sys_TLBIP;  // RVAALE1IS
when '000 1000 0101 001' => return Sys_TLBIP;  // RVAE1OS
when '000 1000 0101 011' => return Sys_TLBIP;  // RVAAE1OS
when '000 1000 0101 101' => return Sys_TLBIP;  // RVALE1OS
when '000 1000 0101 111' => return Sys_TLBIP;  // RVAALE1OS
when '000 1000 0110 001' => return Sys_TLBIP;  // RVAE1
when '000 1000 0110 011' => return Sys_TLBIP;  // RVAAE1
when '000 1000 0110 101' => return Sys_TLBIP;  // RVALE1
when '000 1000 0110 111' => return Sys_TLBIP;  // RVAALE1
when '000 1001 0010 001' => return Sys_TLBIP;  // RVAE1ISNXS
when '000 1001 0010 011' => return Sys_TLBIP;  // RVAAE1ISNXS
when '000 1001 0010 101' => return Sys_TLBIP;  // RVALE1ISNXS
when '000 1001 0010 111' => return Sys_TLBIP;  // RVAALE1ISNXS
when '000 1001 0101 001' => return Sys_TLBIP;  // RVAE1OSNXS
when '000 1001 0101 011' => return Sys_TLBIP;  // RVAAE1OSNXS
when '000 1001 0101 101' => return Sys_TLBIP;  // RVALE1OSNXS
when '000 1001 0101 111' => return Sys_TLBIP;  // RVAALE1OSNXS
when '000 1001 0110 001' => return Sys_TLBIP;  // RVAE1NXS
when '000 1001 0110 011' => return Sys_TLBIP;  // RVAAE1NXS
when '000 1001 0110 101' => return Sys_TLBIP;  // RVALE1NXS
when '000 1001 0110 111' => return Sys_TLBIP;  // RVAALE1NXS
when '100 1000 0010 001' => return Sys_TLBIP;  // RVAE2IS
when '100 1000 0010 101' => return Sys_TLBIP;  // RVALE2IS
when '100 1000 0101 001' => return Sys_TLBIP;  // RVAE2OS
when '100 1000 0101 101' => return Sys_TLBIP;  // RVALE2OS
when '100 1000 0110 001' => return Sys_TLBIP;  // RVAE2
when '100 1000 0110 101' => return Sys_TLBIP;  // RVALE2
when '100 1001 0010 001' => return Sys_TLBIP;  // RVAE2ISNXS
when '100 1001 0010 101' => return Sys_TLBIP;  // RVALE2ISNXS
when '100 1001 0101 001' => return Sys_TLBIP;  // RVAE2OSNXS
when '100 1001 0101 101' => return Sys_TLBIP;  // RVALE2OSNXS
when '100 1001 0110 001' => return Sys_TLBIP;  // RVAE2NXS
when '100 1001 0110 101' => return Sys_TLBIP;  // RVALE2NXS
when '110 1000 0010 001' => return Sys_TLBIP;  // RVAE3IS
when '110 1000 0010 101' => return Sys_TLBIP;  // RVALE3IS
when '110 1000 0101 001' => return Sys_TLBIP;  // RVAE3OS
when '110 1000 0101 101' => return Sys_TLBIP;  // RVALE3OS
when '110 1000 0110 001' => return Sys_TLBIP;  // RVAE3
when '110 1000 0110 101' => return Sys_TLBIP;  // RVALE3
when '110 1001 0010 001' => return Sys_TLBIP;  // RVAE3ISNXS
when '110 1001 0010 101' => return Sys_TLBIP;  // RVALE3ISNXS
when '110 1001 0101 001' => return Sys_TLBIP;  // RVAE3OSNXS
when '110 1001 0101 101' => return Sys_TLBIP;  // RVALE3OSNXS
when '110 1001 0110 001' => return Sys_TLBIP;  // RVAE3NXS
when '110 1001 0110 101' => return Sys_TLBIP;  // RVALE3NXS
when '100 1000 0000 010' => return Sys_TLBIP;  // RIPAS2E1IS
when '100 1000 0000 110' => return Sys_TLBIP;  // RIPAS2LE1IS
when '100 1000 0100 010' => return Sys_TLBIP;  // RIPAS2E1
when '100 1000 0100 011' => return Sys_TLBIP;  // RIPAS2E1OS
when '100 1000 0100 110' => return Sys_TLBIP;  // RIPAS2LE1
when '100 1000 0100 111' => return Sys_TLBIP;  // RIPAS2LE1OS
when '100 1001 0000 010' => return Sys_TLBIP;  // RIPAS2E1ISNXS
when '100 1001 0000 110' => return Sys_TLBIP;  // RIPAS2LE1ISNXS
when '100 1001 0100 010' => return Sys_TLBIP;  // RIPAS2E1NXS
when '100 1001 0100 011' => return Sys_TLBIP;  // RIPAS2E1OSNXS
when '100 1001 0100 110' => return Sys_TLBIP;  // RIPAS2LE1NXS
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '100 1001 0100 111' => return Sys_TLBIP;  // RIPAS2LE1OSNXS
        otherwise                => return Sys_SYSP;
    end;
end;
```

J1.2.3.349 SystemOp128

```
// SystemOp128()
// =============
// System instruction types.

type SystemOp128 of enumeration {Sys_TLBIP, Sys_SYSP};
```

J1.2.3.350 ELR_EL

```
// ELR_EL - accessor
// =================

accessor ELR_EL(el : bits(2)) <=> value : bits(64)
begin
    getter
        var r : bits(64);
        case el of
            when EL1 =>  r = ELR_EL1();
            when EL2 =>  r = ELR_EL2();
            when EL3 =>  r = ELR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        let r : bits(64) = value;
        case el of
            when EL1 =>  ELR_EL1() = r;
            when EL2 =>  ELR_EL2() = r;
            when EL3 =>  ELR_EL3() = r;
            otherwise => unreachable;
        end;
    end;
end;
```

J1.2.3.351 ELR_ELx

```
// ELR_ELx - accessor
// ==================

accessor ELR_ELx() <=> value : bits(64)
begin
    getter
        assert PSTATE.EL != EL0;
        return ELR_EL(PSTATE.EL);
    end;
    setter
        assert PSTATE.EL != EL0;
        ELR_EL(PSTATE.EL) = value;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.352 ESR_EL

```
// ESR_EL - accessor
// =================

accessor ESR_EL(regime : bits(2)) <=> value : ESRType
begin
    getter
        var r : bits(64);
        case regime of
            when EL1 =>  r = ESR_EL1();
            when EL2 =>  r = ESR_EL2();
            when EL3 =>  r = ESR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        let r : bits(64) = value;
        case regime of
            when EL1 =>  ESR_EL1() = r;
            when EL2 =>  ESR_EL2() = r;
            when EL3 =>  ESR_EL3() = r;
            otherwise => unreachable;
        end;
    end;
end;
```

J1.2.3.353 ESR_ELx

```
// ESR_ELx - accessor
// ==================

accessor ESR_ELx() <=> value : ESRType
begin
    getter
        return ESR_EL(S1TranslationRegime());
    end;

    setter
        ESR_EL(S1TranslationRegime()) = value;
    end;
end;
```

J1.2.3.354 FAR_EL

```
// FAR_EL - accessor
// =================

accessor FAR_EL(regime : bits(2)) <=> value : bits(64)
begin
    getter
        var r : bits(64);
        case regime of
            when EL1 =>  r = FAR_EL1();
            when EL2 =>  r = FAR_EL2();
            when EL3 =>  r = FAR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        let r : bits(64) = value;
        case regime of
            when EL1 =>  FAR_EL1() = r;
            when EL2 =>  FAR_EL2() = r;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when EL3 =>  FAR_EL3() = r;
            otherwise => unreachable;
        end;
    end;
end;
```

J1.2.3.355 FAR_ELx

```
// FAR_ELx - accessor
// ==================

accessor FAR_ELx() <=> value : bits(64)
begin
    getter
        return FAR_EL(S1TranslationRegime());
    end;

    setter
        FAR_EL(S1TranslationRegime()) = value;
    end;
end;
```

J1.2.3.356 PFAR_EL

```
// PFAR_EL - accessor
// ==================

accessor PFAR_EL(regime : bits(2)) <=> value : bits(64)
begin
    getter
        assert (IsFeatureImplemented(FEAT_PFAR) ||
                 (regime == EL3 && IsFeatureImplemented(FEAT_RME)));
        var r : bits(64);
        case regime of
            when EL1 => r = PFAR_EL1();
            when EL2 => r = PFAR_EL2();
            when EL3 => r = MFAR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        let r : bits(64) = value;
        assert (IsFeatureImplemented(FEAT_PFAR) ||
                   (IsFeatureImplemented(FEAT_RME) && regime == EL3));
        case regime of
            when EL1 => PFAR_EL1() = r;
            when EL2 => PFAR_EL2() = r;
            when EL3 => MFAR_EL3() = r;
            otherwise => unreachable;
        end;
    end;
end;
```

J1.2.3.357 PFAR_ELx

```
// PFAR_ELx - accessor
// ===================

accessor PFAR_ELx() <=> value : bits(64)
begin
    getter
        return PFAR_EL(S1TranslationRegime());
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    setter
        PFAR_EL(S1TranslationRegime()) = value;
    end;
end;
```

J1.2.3.358 SCTLR_EL

```
// SCTLR_EL - accessor
// ===================

accessor SCTLR_EL(regime : bits(2)) <=> value : SCTLRType
begin
    getter
        var r : bits(64);
        case regime of
            when EL1 =>  r = SCTLR_EL1();
            when EL2 =>  r = SCTLR_EL2();
            when EL3 =>  r = SCTLR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        let r : bits(64) = value;
        case regime of
            when EL1 => SCTLR_EL1() = r;
            when EL2 => SCTLR_EL2() = r;
            when EL3 => SCTLR_EL3() = r;
            otherwise => unreachable;
        end;
    end;
end;
```

J1.2.3.359 SCTLR_ELx

```
// SCTLR_ELx - accessor
// ====================

accessor SCTLR_ELx() <=> value : SCTLRType
begin
    getter
        return SCTLR_EL(S1TranslationRegime());
    end;
    setter
        SCTLR_EL(S1TranslationRegime()) = value;
    end;
end;
```

J1.2.3.360 VBAR_EL

```
// VBAR_EL - accessor
// ==================

accessor VBAR_EL(regime : bits(2)) <=> value : bits(64)
begin
    getter
        var r : bits(64);
        case regime of
            when EL1 =>  r = VBAR_EL1();
            when EL2 =>  r = VBAR_EL2();
            when EL3 =>  r = VBAR_EL3();
            otherwise => unreachable;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return r;
    end;
    setter
        pass;
    end;
end;
```

J1.2.3.361 VBAR_ELx

```
// VBAR_ELx - accessor
// ===================

accessor VBAR_ELx() <=> value : bits(64)
begin
    getter
        return VBAR_EL(S1TranslationRegime());
    end;
    setter
        VBAR_EL(S1TranslationRegime())[63:0] = value;
    end;
end;
```

J1.2.3.362 AArch64_CheckDAIFAccess

```
// AArch64_CheckDAIFAccess()
// =========================
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

func AArch64_CheckDAIFAccess(field : PSTATEField)
begin
    if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
        if IsInHost() || SCTLR_EL1().UMA == '0' then
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                AArch64_SystemAccessTrap(EL2, 0x18);
            else
                AArch64_SystemAccessTrap(EL1, 0x18);
            end;
        end;
    end;
end;
```

J1.2.3.363 AArch64_ChooseNonExcludedTag

```
// AArch64_ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

func AArch64_ChooseNonExcludedTag(tag_in : bits(4), offset_in : bits(4),
                                  exclude : bits(16)) => bits(4)
begin
    var tag : bits(4) = tag_in;
    var offset : bits(4) = offset_in;

    if IsOnes(exclude) then
        return '0000';
    end;

    if offset == '0000' then
        while exclude[UInt(tag)] == '1' looplimit 16 do
            tag = tag + '0001';
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    while offset != '0000' looplimit 16 do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude[UInt(tag)] == '1' looplimit 16 do
            tag = tag + '0001';
        end;
    end;

    return tag;
end;
```

J1.2.3.364 AArch64_ChooseNonExludedTagOrZero

```
// AArch64_ChooseNonExludedTagOrZero()
// ===================================
// Return a tag derived from the start and the offset values, excluding any
// tags in the given mask, or zero if Allocation Tag access is not enabled.

func AArch64_ChooseNonExcludedTagOrZero(tag : bits(4), offset : bits(4),
                                        exclude : bits(16)) => bits(4)
begin
    if IsMTEEnabled(PSTATE.EL) then
        return AArch64_ChooseNonExcludedTag(tag, offset, exclude);
    else
        return '0000';
    end;
end;
```

J1.2.3.365 AArch64_ChooseTagOrZero

```
// AArch64_ChooseTagOrZero()
// =========================
// Return a tag, excluding any tags in the given mask, , or zero if Allocation
// Tag access is not enabled.

func AArch64_ChooseTagOrZero(exclude : bits(16)) => bits(4)
begin
    if IsMTEEnabled(PSTATE.EL) then
        if GCR_EL1().RRND == '1' then
            if IsOnes(exclude) then
                return '0000';
            else
                return ChooseRandomNonExcludedTag(exclude);
            end;
        else
            let start_tag : bits(4) = RGSR_EL1().TAG;
            let offset : bits(4)    = AArch64_RandomTag();

            RGSR_EL1().TAG = AArch64_ChooseNonExcludedTag(start_tag, offset, exclude);

            return RGSR_EL1().TAG;
        end;
    else
        return '0000';
    end;
end;
```

J1.2.3.366 AArch64_ExecutingERETInstr

```
// AArch64_ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

func AArch64_ExecutingERETInstr() => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    let instr : bits(32) = ThisInstr();
    return instr[31:12] == '11010110100111110000';
end;
```

J1.2.3.367 AArch64_ImpDefSysInstr

```
// AArch64_ImpDefSysInstr()
// ========================
// Execute an implementation-defined system instruction with write (source operand).

impdef func AArch64_ImpDefSysInstr(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                   crm : bits(4), op2 : bits(3), t : integer)
begin
    Undefined();
end;
```

J1.2.3.368 AArch64_ImpDefSysInstr128

```
// AArch64_ImpDefSysInstr128()
// ===========================
// Execute an implementation-defined system instruction with write (128-bit source operand).

impdef func AArch64_ImpDefSysInstr128(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                      crm : bits(4), op2 : bits(3),
                                      t : integer, t2 : integer)
begin
    Undefined();
end;
```

J1.2.3.369 AArch64_ImpDefSysInstrWithResult

```
// AArch64_ImpDefSysInstrWithResult()
// ==================================
// Execute an implementation-defined system instruction with read (result operand).

impdef func AArch64_ImpDefSysInstrWithResult(op0 : bits(2),
                                             op1 : bits(3),
                                             crn : bits(4),
                                             crm : bits(4),
                                             op2 : bits(3),
                                             t : integer)
begin
    Undefined();
end;
```

J1.2.3.370 AArch64_ImpDefSysRegRead

```
// AArch64_ImpDefSysRegRead()
// ==========================
// Read from an implementation-defined System register and write the contents of the register
// to X[t].

impdef func AArch64_ImpDefSysRegRead(op0 : bits(2), op1 : bits(3),
                                     crn : bits(4), crm : bits(4),
                                     op2 : bits(3), t : integer)
begin
    Undefined();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.371 AArch64_ImpDefSysRegRead128

```
// AArch64_ImpDefSysRegRead128()
// ============================
// Read from an 128-bit implementation-defined System register
// and write the contents of the register to X[t], X[t+1].

impdef func AArch64_ImpDefSysRegRead128(op0 : bits(2),
                                         op1 : bits(3), crn : bits(4),
                                         crm : bits(4), op2 : bits(3),
                                         t : integer, t2 : integer)
begin
    Undefined();
end;
```

J1.2.3.372 AArch64_ImpDefSysRegWrite

```
// AArch64_ImpDefSysRegWrite()
// ==========================
// Write to an implementation-defined System register.

impdef func AArch64_ImpDefSysRegWrite(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                      crm : bits(4), op2 : bits(3), t : integer)
begin
    Undefined();
end;
```

J1.2.3.373 AArch64_ImpDefSysRegWrite128

```
// AArch64_ImpDefSysRegWrite128()
// ==============================
// Write the contents of X[t], X[t+1] to an 128-bit implementation-defined System register.

impdef func AArch64_ImpDefSysRegWrite128(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                          crm : bits(4), op2 : bits(3),
                                          t : integer, t2 : integer)
begin
    Undefined();
end;
```

J1.2.3.374 AArch64_InterruptPending

```
// AArch64_InterruptPending()
// ==========================
// Returns TRUE if there are any pending physical, virtual, or delegated
// interrupts, and FALSE otherwise.

func AArch64_InterruptPending() => boolean
begin
    let (irq_pending, -) = IRQPending();
    let (fiq_pending, -) = FIQPending();
    let pending_physical_interrupt : boolean = (irq_pending || fiq_pending ||
                                                IsPhysicalSErrorPending());

    var pending_virtual_interrupt : boolean = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2().TGE == '0' then
        let virq_pending : boolean = (HCR_EL2().IMO == '1' && (VirtualIRQPending() ||
                                      HCR_EL2().VI == '1'));
        let vfiq_pending : boolean = (HCR_EL2().FMO == '1' && (VirtualFIQPending() ||
                                      HCR_EL2().VF == '1'));
        let vsei_pending : boolean = ((HCR_EL2().AMO == '1' ||
                                     (IsFeatureImplemented(FEAT_DoubleFault2) &&
                                      IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) &&
                                     (IsVirtualSErrorPending() || HCR_EL2().VSE == '1'));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;
    end;

    let pending_delegated_interrupt : boolean = (PSTATE.EL != EL3 &&
                                                 EffectiveSCR_EL3_EnDSE() == '1' &&
                                                 SCR_EL3().DSE == '1');

    return pending_physical_interrupt || pending_virtual_interrupt || pending_delegated_interrupt;
end;
```

J1.2.3.375 AArch64_NextRandomTagBit

```
// AArch64_NextRandomTagBit()
// ==========================
// Generate a random bit suitable for generating a random Allocation Tag.

func AArch64_NextRandomTagBit() => bit
begin
    assert GCR_EL1().RRND == '0';
    let lfsr : bits(16) = RGSR_EL1().SEED[15:0];
    let top : bit = lfsr[5] XOR lfsr[3] XOR lfsr[2] XOR lfsr[0];
    RGSR_EL1().SEED[15:0] = top::lfsr[15:1];
    return top;
end;
```

J1.2.3.376 AArch64_RandomTag

```
// AArch64_RandomTag()
// ===================
// Generate a random Allocation Tag.

func AArch64_RandomTag() => bits(4)
begin
    var tag : bits(4);
    for i = 0 to 3 do
        tag[i] = AArch64_NextRandomTagBit();
    end;
    return tag;
end;
```

J1.2.3.377 AArch64_SysInstr

```
// AArch64_SysInstr()
// ==================
// Execute a system instruction with write (source operand).

impdef func AArch64_SysInstr(op0 : bits(2), op1 : bits(3),
                             crn : bits(4), crm : bits(4), op2 : bits(3), t : integer)
begin
    Unimplemented();
end;
```

J1.2.3.378 AArch64_SysInstrWithResult

```
// AArch64_SysInstrWithResult()
// ============================
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].

impdef func AArch64_SysInstrWithResult(op0 : bits(2), op1 : bits(3),
                                       crn : bits(4), crm : bits(4),
                                       op2 : bits(3), t : integer)
begin
    Unimplemented();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.379 AArch64_SysRegRead

```
// AArch64_SysRegRead()
// ====================
// Read from a System register and write the contents of the register to X[t].

impdef func AArch64_SysRegRead(op0 : bits(2), op1 : bits(3), crn : bits(4),
                               crm : bits(4), op2 : bits(3), t : integer)
begin
    Unimplemented();
end;
```

J1.2.3.380 AArch64_SysRegWrite

```
// AArch64_SysRegWrite()
// =====================
// Write to a System register.

impdef func AArch64_SysRegWrite(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                crm : bits(4), op2 : bits(3), t : integer)
begin
    Unimplemented();
end;
```

J1.2.3.381 BTypeCompatible

```
// BTypeCompatible
// ===============
// Records the branch target compatibility.
// Returns TRUE if the branch target is compatible with PSTATE.BTYPE, else FALSE.

var BTypeCompatible : boolean;
```

J1.2.3.382 BTypeCompatible_BTI

```
// BTypeCompatible_BTI
// ===================
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

func BTypeCompatible_BTI(hintcode : bits(2)) => boolean
begin
    case hintcode of
        when '00' =>
            return PSTATE.BTYPE == '00';
        when '01' =>
            return PSTATE.BTYPE != '11';
        when '10' =>
            return PSTATE.BTYPE != '10';
        when '11' =>
            return TRUE;
    end;
end;
```

J1.2.3.383 BTypeCompatible_PAC

```
// BTypeCompatible_PAC()
// =====================
// Returns TRUE if PACIxSP or PACIxSPPC instructions are implicitly compatible with PSTATE.BTYPE,
// FALSE otherwise.

func BTypeCompatible_PAC(pacinst : PACInstType) => boolean
begin
    if PSTATE.BTYPE != '11' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return TRUE;
    else
        let index : integer = if PSTATE.EL == EL0 then 35 else 36;
        return SCTLR_ELx()[index] == '0';
    end;
end;
```

J1.2.3.384 BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =========================
// Returns TRUE if PACIASP or PACIBSP instructions are implicitly compatible with PSTATE.BTYPE,
// FALSE otherwise.

func BTypeCompatible_PACIXSP() => boolean
begin
    let pacinst : PACInstType = PACIxSP;
    return BTypeCompatible_PAC(pacinst);
end;
```

J1.2.3.385 BTypeNext

```
// BTypeNext
// =========
// Updated every cycle with a value that depends upon the instruction being executed. Assigned to
// PSTATE.BTYPE at the end of each cycle and then cleared to zero. Allows SPSR save/restore of
// BTYPE for the current instruction being executed.

var BTypeNext : bits(2);
```

J1.2.3.386 ChooseRandomNonExcludedTag

```
// ChooseRandomNonExcludedTag()
// ============================
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation should select a tag from a uniform distribution and
// avoid common pitfalls such as modulo bias.
//
// This function can read RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a previous nonzero
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSR_EL1.SEED[15:0] is set to zero.

impdef func ChooseRandomNonExcludedTag(exclude_in : bits(16)) => bits(4)
begin
    return ARBITRARY : bits(4);
end;
```

J1.2.3.387 EffectiveBADDR

```
// EffectiveBADDR()
// ================
// Check if the given VA, held in a register BADDR field, is RESS and apply
// CONSTRAINED UNPREDICTABLE behavior if it is not.

func EffectiveBADDR(baddr_in : bits(64), directread : boolean) => bits(64)
begin
    var baddr : bits(64) = baddr_in;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Determine top bit position based on feature support.
    let n : integer{} = (if IsFeatureImplemented(FEAT_LVA3) then 57
                         else (if IsFeatureImplemented(FEAT_LVA) then 53
                         else 49));

    // Check if the upper bits of the base address form a valid sign extension.
    if baddr[63:n] != Replicate{64-n}(baddr[n-1]) then
        let c = ConstrainUnpredictable(Unpredictable_BADDR_RESS);
        assert c IN {Constraint_RESS, Constraint_ALLRESS, Constraint_FAULT};
        if c == Constraint_ALLRESS || (!directread && c == Constraint_RESS) then
            baddr[63:n] = Replicate{64-n}(baddr[n-1]);
        end;
        // If c == Constraint_FAULT, baddr will cause an Address Size fault when used.
    end;

    return baddr;
end;
```

J1.2.3.388 EffectiveICC_SRE_EL2_Enable

```
// EffectiveICC_SRE_EL2_Enable()
// ============================
// Returns the Effective value of ICC_SRE_EL2.Enable

func EffectiveICC_SRE_EL2_Enable() => bit
begin
    if (ImpDefBool("Treat ICC_SRE_EL2.SRE as RAO/WI") &&
          ImpDefBool("Treat ICC_SRE_EL2.Enable as RAO/WI")) then
        return '1';
    end;
    if ICC_SRE_EL2().SRE == '0' then
        return '1';
    end;

    return ICC_SRE_EL2().Enable;
end;
```

J1.2.3.389 EffectiveICC_SRE_EL3_Enable

```
// EffectiveICC_SRE_EL3_Enable()
// ============================
// Returns the Effective value of ICC_SRE_EL3.Enable

func EffectiveICC_SRE_EL3_Enable() => bit
begin

    return ICC_SRE_EL3().Enable;
end;
```

J1.2.3.390 InGuardedPage

```
// InGuardedPage
// =============
// Records whether the currently fetched instruction was retrieved from a guarded page, will be
// TRUE if the GP bit in the page or block descriptor for the current instruction fetch was equal
// to one.

var InGuardedPage : boolean;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.391 IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// ==================
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

readonly func IsHCRXEL2Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_HCX) then return FALSE; end;
    if ELUsingAArch32(EL2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().HXEn == '0' then
        return FALSE;
    end;

    return EL2Enabled();
end;
```

J1.2.3.392 IsMTEEnabled

```
// IsMTEEnabled()
// ==============
// Returns TRUE if the currently selected MTE mechanism is enabled, and FALSE otherwise.

func IsMTEEnabled(el : bits(2)) => boolean
begin
    let regime : Regime = TranslationRegime(el);
    var selected : boolean = FALSE;

    if IsPMTESelected(el) then
        if HaveEL(EL3) && SCR_EL3().ATA == '0' && el IN {EL0, EL1, EL2} then
            return FALSE;
        end;

        if EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().ATA == '0' && el IN {EL0, EL1} then
            return FALSE;
        end;

        selected = TRUE;
    end;

    if selected then
        case regime of
            when Regime_EL3 =>
                return SCTLR_EL3().ATA == '1';
            when Regime_EL2 =>
                return SCTLR_EL2().ATA == '1';
            when Regime_EL20 =>
                return if el == EL0 then SCTLR_EL2().ATA0 == '1' else SCTLR_EL2().ATA == '1';
            when Regime_EL10 =>
                return if el == EL0 then SCTLR_EL1().ATA0 == '1' else SCTLR_EL1().ATA == '1';
            otherwise => unreachable;
        end;
    end;

    return FALSE;
end;
```

J1.2.3.393 IsPMTESelected

```
// IsPMTESelected()
// ================
// Returns TRUE if Physical MTE is selected for the translation regime
// associated with the EL, and FALSE otherwise

func IsPMTESelected(el : bits(2)) => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    if !IsFeatureImplemented(FEAT_MTE2) then
        return FALSE;
    end;

    return TRUE;
end;
```

J1.2.3.394 IsSCTLR2EL1Enabled

```
// IsSCTLR2EL1Enabled()
// ====================
// Returns TRUE if access to SCTLR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL1 returns 0 when access is not enabled.

func IsSCTLR2EL1Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

J1.2.3.395 IsSCTLR2EL2Enabled

```
// IsSCTLR2EL2Enabled()
// ====================
// Returns TRUE if access to SCTLR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL2 returns 0 when access is not enabled.

func IsSCTLR2EL2Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        return FALSE;
    end;

    return EL2Enabled();
end;
```

J1.2.3.396 IsSystemRegisterMaskingEnabled

```
// IsSystemRegisterMaskingEnabled()
// ================================
// Returns TRUE if access to the MASK registers introduced by FEAT_SRMASK is enabled and the
// values in the register are not treated as 0.

func IsSystemRegisterMaskingEnabled(el : bits(2)) => boolean
begin
    assert IsFeatureImplemented(FEAT_SRMASK) && el IN {EL1, EL2};

    if HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        return FALSE;
    end;

    return el == EL2 || !EL2Enabled() || (IsHCRXEL2Enabled() && HCRX_EL2().SRMASKEn == '1');
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.397 IsTCR2EL1Enabled

```
// IsTCR2EL1Enabled()
// ==================
// Returns TRUE if access to TCR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL1 returns 0 when access is not enabled.

func IsTCR2EL1Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0')) then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

J1.2.3.398 IsTCR2EL2Enabled

```
// IsTCR2EL2Enabled()
// ==================
// Returns TRUE if access to TCR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL2 returns 0 when access is not enabled.

func IsTCR2EL2Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        return FALSE;
    end;

    return EL2Enabled();
end;
```

J1.2.3.399 PACInstType

```
// PACInstType
// ===========
type PACInstType of enumeration {
    PACIxSP,
    PACIxSPPC,
};
```

J1.2.3.400 SetBTypeCompatible

```
// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI

func SetBTypeCompatible(x : boolean)
begin
    BTypeCompatible = x;
end;
```

J1.2.3.401 SetBTypeNext

```
// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

func SetBTypeNext(x : bits(2))
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    BTypeNext = x;
end;
```

J1.2.3.402 SetInGuardedPage

```
// SetInGuardedPage()
// ==================
// Global state updated to denote if memory access is from a guarded page.

func SetInGuardedPage(guardedpage : boolean)
begin
    InGuardedPage = guardedpage;
end;
```

J1.2.3.403 AArch64_SysInstr128

```
// AArch64_SysInstr128()
// =====================
// Execute a system instruction with write (2 64-bit source operands).

impdef func AArch64_SysInstr128(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                crm : bits(4), op2 : bits(3), t : integer, t2 : integer)
begin
    Unimplemented();
end;
```

J1.2.3.404 AArch64_SysRegRead128

```
// AArch64_SysRegRead128()
// =======================
// Read from a 128-bit System register and write the contents of the register to X[t] and X[t2].

impdef func AArch64_SysRegRead128(op0 : bits(2), op1 : bits(3),
                                  crn : bits(4), crm : bits(4),
                                  op2 : bits(3), t : integer, t2 : integer)
begin
    Unimplemented();
end;
```

J1.2.3.405 AArch64_SysRegWrite128

```
// AArch64_SysRegWrite128()
// ========================
// Read the contents of X[t] and X[t2] and write the contents to a 128-bit System register.

impdef func AArch64_SysRegWrite128(op0 : bits(2), op1 : bits(3),
                                   crn : bits(4), crm : bits(4),
                                   op2 : bits(3), t : integer, t2 : integer)
begin
    Unimplemented();
end;
```

J1.2.3.406 AArch64_EffectiveBroadcast

```
// AArch64_EffectiveBroadcast()
// ============================

func AArch64_EffectiveBroadcast(broadcast_in : Broadcast) => Broadcast
begin
    var broadcast : Broadcast = broadcast_in;

    if PSTATE.EL != EL1 then return broadcast; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if broadcast == Broadcast_NSH && EL2Enabled() && HCR_EL2().FB == '1' then
        broadcast = Broadcast_ForcedISH;
    end;

    return broadcast;
end;
```

J1.2.3.407 AArch64_TLBIP_IPAS2

```
// AArch64_TLBIP_IPAS2()
// =====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

func AArch64_TLBIP_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                         broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                         Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt[47:44];
    r.address      = ZeroExtend{64}(Xt[107:64] :: Zeros{12});
    r.d64          = r.ttl == '00xx';
    r.d128         = TRUE;

    case security of
        when SS_NonSecure =>
            r.ipaspace = PAS_NonSecure;
        when SS_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm =>
            r.ipaspace = PAS_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.408 AArch64_TLBIP_RIPAS2

```
// AArch64_TLBIP_RIPAS2()
// ======================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_TLBIP_RIPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                          broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                          Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = r.ttl[1:0] == '00';
    r.d128         = TRUE;

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return; end;

    case security of
        when SS_NonSecure =>
            r.ipaspace = PAS_NonSecure;
        when SS_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm =>
            r.ipaspace = PAS_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.409 AArch64_TLBIP_RVA

```
// AArch64_TLBIP_RVA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_RVA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                       Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    elsif regime == Regime_EL2 && ELIsInHost(EL2) then
        regime = Regime_EL20;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt[63:48];
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = r.ttl[1:0] == '00';
    r.d128         = TRUE;

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.2.3.410 AArch64_TLBIP_RVAA

```
// AArch64_TLBIP_RVAA()
// ====================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_RVAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                        Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast    = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = r.ttl[1:0] == '00';
    r.d128         = TRUE;

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.411 AArch64_TLBIP_VA

```
// AArch64_TLBIP_VA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_VA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                      Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if regime == Regime_EL10 && ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    elsif regime == Regime_EL2 && ELIsInHost(EL2) then
        regime = Regime_EL20;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt[63:48];
    r.ttl          = Xt[47:44];
    r.address      = ZeroExtend{64}(Xt[107:64] :: Zeros{12});
    r.d64          = r.ttl == '00xx';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.d128         = TRUE;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.412 AArch64_TLBIP_VAA

```
// AArch64_TLBIP_VAA()
// ===================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_VAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                       Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;
    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt[47:44];
    r.address      = ZeroExtend{64}(Xt[107:64] :: Zeros{12});
    r.d64          = r.ttl == '00xx';
    r.d128         = TRUE;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.413 AArch64_TLBI_ALL

```
// AArch64_TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_TLBI_ALL(security : SecurityState, regime : Regime,
                      broadcast_in : Broadcast, attr : TLBIMemAttr, Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.2.3.414 AArch64_TLBI_ASID

```
// AArch64_TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_ASID(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, attr_in : TLBIMemAttr,
                       Xt :  bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast    = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime    = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Xt[63:48];

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.415 AArch64_TLBI_IPAS2

```
// AArch64_TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

func AArch64_TLBI_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                        Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt[47:44];
    r.address      = ZeroExtend{64}(Xt[39:0] :: Zeros{12});
    r.d64          = TRUE;
    r.d128         = r.ttl == '00xx';

    case security of
        when SS_NonSecure =>
            r.ipaspace = PAS_NonSecure;
        when SS_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm =>
            r.ipaspace = PAS_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.416 AArch64_TLBI_PAALL

```
// AArch64_TLBI_PAALL()
// ====================
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to all TLB entries containing GPT information.

func AArch64_TLBI_PAALL(broadcast : Broadcast)
begin
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    var r : TLBIRecord;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_PAALL;
    r.level = TLBILevel_Any;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.attr  = TLBI_AllAttr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.2.3.417 AArch64_TLBI_RIPAS2

```
// AArch64_TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

func AArch64_TLBI_RIPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                         broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                         Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = TRUE;
    r.d128         = r.ttl[1:0] == '00';

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return; end;

    case security of
        when SS_NonSecure =>
            r.ipaspace = PAS_NonSecure;
        when SS_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm =>
            r.ipaspace = PAS_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.2.3.418 AArch64_TLBI_RPA

```
// AArch64_TLBI_RPA()
// ==================
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Shareabilty domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries containing GPT information
//                      from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                      from the last level of the GPT walk

func AArch64_TLBI_RPA(level : TLBILevel, Xt : bits(64), broadcast : Broadcast)
begin
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    var r : TLBIRecord;
    var range_bits : AddressSize;
    var p : AddressSize;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_RPA;
    r.level = level;
    r.attr  = TLBI_AllAttr;

    // SIZE field
    case Xt[47:44] of
        when '0000' => range_bits = 12; // 4KB
        when '0001' => range_bits = 14; // 16KB
        when '0010' => range_bits = 16; // 64KB
        when '0011' => range_bits = 21; // 2MB
        when '0100' => range_bits = 25; // 32MB
        when '0101' => range_bits = 29; // 512MB
        when '0110' => range_bits = 30; // 1GB
        when '0111' => range_bits = 34; // 16GB
        when '1000' => range_bits = 36; // 64GB
        when '1001' => range_bits = 39; // 512GB
        otherwise =>   return;  // Reserved encoding, no TLB entries are required to be invalidated
    end;

    // If SIZE selects a range smaller than PGS, then PGS is used instead
    case DecodePGS(GPCCR_EL3().PGS) of
        when PGS_4KB =>  p = 12;
        when PGS_16KB => p = 14;
        when PGS_64KB => p = 16;
        otherwise => return;    // Reserved encoding, no TLB entries are required to be invalidated
    end;

    if range_bits < p then
        range_bits = p;
    end;

    var BaseADDR : bits(NUM_PABITS) = Zeros{};
    case GPCCR_EL3().PGS of
        when '00' => BaseADDR[51:12] = Xt[39:0];   // 4KB
        when '10' => BaseADDR[51:14] = Xt[39:2];   // 16KB
        when '01' => BaseADDR[51:16] = Xt[39:4];   // 64KB
    end;

    if IsFeatureImplemented(FEAT_D128) && AArch64_PAMax() == 56 then
        BaseADDR[55:52] = Xt[43:40]; // Extension to Address
    end;

    // The calculation here automatically aligns BaseADDR to the size of
    // the region specified in SIZE. However, the architecture does not
    // require this alignment and if BaseADDR is not aligned to the region
    // specified by SIZE then no entries are required to be invalidated.
    let range_pbits : integer{} = range_bits;
    let start_addr : bits(NUM_PABITS) = BaseADDR AND NOT ZeroExtend{NUM_PABITS}(Ones{range_pbits});
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let end_addr : bits(NUM_PABITS)   = start_addr + ZeroExtend{NUM_PABITS}(Ones{range_pbits});

    // PASpace is not considered in TLBI by PA operations
    r.address     = ZeroExtend{64}(start_addr);
    r.end_address = ZeroExtend{64}(end_addr);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.2.3.419 AArch64_TLBI_RVA

```
// AArch64_TLBI_RVA()
// ==================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_RVA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                      Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if regime == Regime_EL10 && ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    elsif regime == Regime_EL2 && ELIsInHost(EL2) then
        regime = Regime_EL20;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt[63:48];
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = TRUE;
    r.d128         = r.ttl[1:0] == '00';

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.420 AArch64_TLBI_RVAA

```
// AArch64_TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_RVAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                       Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = TRUE;
    r.d128         = r.ttl[1:0] == '00';

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.2.3.421 AArch64_TLBI_VA

```
// AArch64_TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_TLBI_VA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                     broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                     Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;
    if regime == Regime_EL10 && ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    elsif regime == Regime_EL2 && ELIsInHost(EL2) then
        regime = Regime_EL20;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt[63:48];
    r.ttl          = Xt[47:44];
    r.address      = ZeroExtend{64}(Xt[43:0] :: Zeros{12});
    r.d64          = TRUE;
    r.d128         = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.422 AArch64_TLBI_VAA

```
// AArch64_TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_VAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                      Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt[47:44];
    r.address      = ZeroExtend{64}(Xt[43:0] :: Zeros{12});
    r.d64          = TRUE;
    r.d128         = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.423 AArch64_TLBI_VMALL

```
// AArch64_TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

func AArch64_TLBI_VMALL(security : SecurityState, regime_in : Regime,
                        vmid_in : bits(NUM_VMIDBITS), broadcast_in : Broadcast,
                        attr_in : TLBIMemAttr, Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast   = AArch64_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI_AllAttr && ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var regime : Regime   = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime_EL20;
        vmid   = VMID_NONE;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.attr         = attr;

    TLBI(r);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.424 AArch64_TLBI_VMALLS12

```
// AArch64_TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

func AArch64_TLBI_VMALLS12(security : SecurityState, regime : Regime,
                           vmid : bits(NUM_VMIDBITS), broadcast_in : Broadcast,
                           attr : TLBIMemAttr, Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = AArch64_EffectiveBroadcast(broadcast_in);

    var r : TLBIRecord;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.425 AArch64_TLBI_VMALLWS2

```
// AArch64_TLBI_VMALLWS2()
// =======================
// Remove stage 2 dirty state from entries for the indicated translation regime
// with the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID.

func AArch64_TLBI_VMALLWS2(security : SecurityState, regime : Regime, vmid : bits(16),
                           broadcast_in : Broadcast, attr : TLBIMemAttr,
                           Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    assert regime == Regime_EL10;

    if security == SS_Secure && HaveEL(EL3) && SCR_EL3().EEL2 == '0' then
        return;
    end;

    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op           = TLBIOp_VMALLWS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

J1.2.3.426 DecodeTLBITG

```
// DecodeTLBITG()
// ==============
// Decode translation granule size in TLBI range instructions

func DecodeTLBITG(tg : bits(2)) => TGx
begin
    case tg of
        when '01' =>   return TGx_4KB;
        when '10' =>   return TGx_16KB;
        when '11' =>   return TGx_64KB;
    end;
end;
```

J1.2.3.427 GPTTLBIMatch

```
// GPTTLBIMatch()
// ==============
// Determine whether the GPT TLB entry lies within the scope of invalidation

func GPTTLBIMatch(tlbi : TLBIRecord, gpt_entry : GPTEntry) => boolean
begin
    assert tlbi.op IN {TLBIOp_RPA, TLBIOp_PAALL};

    var match : boolean;
    let entry_size_mask : bits(64)     = ZeroExtend{}(Ones{gpt_entry.size});
    let entry_end_address : bits(64)   = (ZeroExtend{64}(gpt_entry.pa[NUM_PABITS-1:0] OR
                                              entry_size_mask[NUM_PABITS-1:0]));
    let entry_start_address : bits(64) = (ZeroExtend{64}(gpt_entry.pa[NUM_PABITS-1:0] AND NOT
                                              entry_size_mask[NUM_PABITS-1:0]));

    case tlbi.op of
        when TLBIOp_RPA =>
            match = ((UInt(tlbi.address[NUM_PABITS-1:0]) <=
                        UInt(entry_end_address[NUM_PABITS-1:0])) &&
                     (UInt(tlbi.end_address[NUM_PABITS-1:0]) >
                        UInt(entry_start_address[NUM_PABITS-1:0])) &&
                     (tlbi.level == TLBILevel_Any || !gpt_entry.istable));
        when TLBIOp_PAALL =>
            match = TRUE;
    end;

    return match;
end;
```

J1.2.3.428 HasLargeAddress

```
// HasLargeAddress()
// =================
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

func HasLargeAddress(regime : Regime) => boolean
begin
    if !IsFeatureImplemented(FEAT_LPA2) then
        return FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    case regime of
        when Regime_EL3 =>
            return TCR_EL3().DS == '1';
        when Regime_EL2 =>
            return TCR_EL2().DS == '1';
        when Regime_EL20 =>
            return TCR_EL2().DS == '1';
        when Regime_EL10 =>
            return TCR_EL1().DS == '1';
        otherwise =>
            unreachable;
    end;
end;
```

J1.2.3.429 ResTLBIRTTL

```
// ResTLBIRTTL()
// =============
// Determine whether the TTL field in TLBI instructions that do apply
// to a range of addresses contains a reserved value

func ResTLBIRTTL(tg : bits(2), ttl : bits(2)) => boolean
begin
    case ttl of
        when '00' => return TRUE;
        when '01' => return DecodeTLBITG(tg) == TGx_16KB && !IsFeatureImplemented(FEAT_LPA2);
        otherwise => return FALSE;
    end;
end;
```

J1.2.3.430 ResTLBITTL

```
// ResTLBITTL()
// ============
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

func ResTLBITTL(ttl : bits(4)) => boolean
begin
    case ttl of
        when '00xx' => return TRUE;
        when '0100' => return !IsFeatureImplemented(FEAT_LPA2);
        when '1000' => return TRUE;
        when '1001' => return !IsFeatureImplemented(FEAT_LPA2);
        when '1100' => return TRUE;
        otherwise =>   return FALSE;
    end;
end;
```

J1.2.3.431 TGBits

```
// TGBits()
// ========
// Return the number of least-significant address bits within a single Translation Granule.

func TGBits(tg : bits(2)) => AddressSize
begin
    case tg of
        when '01' => return 12; // 4KB
        when '10' => return 14; // 16KB
        when '11' => return 16; // 64KB
        otherwise =>
            unreachable;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

J1.2.3.432 TLBIMatch

```
// TLBIMatch()
// ===========
// Determine whether the TLB entry lies within the scope of invalidation

func TLBIMatch(tlbi : TLBIRecord, tlb_entry : TLBRecord) => boolean
begin
    var match : boolean;
    let entry_block_mask : bits(64)    = ZeroExtend{}(Ones{tlb_entry.blocksize});
    var entry_end_address : bits(64)   = tlb_entry.context.ia OR entry_block_mask;
    var entry_start_address : bits(64) = tlb_entry.context.ia AND NOT entry_block_mask;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL =>
            match = (tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
        when TLBIOp_DASID, TLBIOp_IASID =>
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                        tlbi.asid  == tlb_entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA =>
            var regime_match : boolean;
            var context_match : boolean;
            var address_match : boolean;
            var level_match : boolean;
            regime_match = (tlb_entry.context.includes_s1 &&
                            tlbi.security == tlb_entry.context.ss &&
                            tlbi.regime   == tlb_entry.context.regime);
            context_match = (tlbi.use_vmid == tlb_entry.context.use_vmid &&
                             (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                             (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                                tlb_entry.context.nG == '0'));
            let addr_lsb : integer{} = tlb_entry.blocksize;
            address_match = tlbi.address[55:addr_lsb] == tlb_entry.context.ia[55:addr_lsb];
            level_match = (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable);
            match = regime_match && context_match  && address_match  && level_match;
        when TLBIOp_ALL =>
            let relax_regime : boolean = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                            tlb_entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == tlb_entry.context.ss &&
                     (tlbi.regime  == tlb_entry.context.regime || relax_regime));
        when TLBIOp_ASID =>
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                        tlbi.asid  == tlb_entry.context.asid));
        when TLBIOp_IPAS2, TLBIPOp_IPAS2 =>
            let addr_lsb : integer{} = tlb_entry.blocksize;
            match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     tlbi.ipaspace == tlb_entry.context.ipaspace &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                (tlbi.address[NUM_PABITS-1:addr_lsb] ==
                   tlb_entry.context.ia[NUM_PABITS-1:addr_lsb]) &&
                (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                    DecodeTLBITG(tlbi.ttl[3:2]) == tlb_entry.context.tg &&
                    UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level)
                ) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
                (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
when TLBIOp_VAA, TLBIPOp_VAA =>
    let addr_lsb : integer{} = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                tlbi.use_vmid == tlb_entry.context.use_vmid &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                tlbi.address[55:addr_lsb] == tlb_entry.context.ia[55:addr_lsb] &&
                (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                    DecodeTLBITG(tlbi.ttl[3:2]) == tlb_entry.context.tg &&
                    UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level)
                ) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
                (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
when TLBIOp_VA, TLBIPOp_VA =>
    let addr_lsb : integer{} = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                tlbi.use_vmid == tlb_entry.context.use_vmid &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                   tlb_entry.context.nG == '0') &&
                tlbi.address[55:addr_lsb] == tlb_entry.context.ia[55:addr_lsb] &&
                (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                    DecodeTLBITG(tlbi.ttl[3:2]) == tlb_entry.context.tg &&
                    UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level)
                ) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
                (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
when TLBIOp_VMALL =>
    match = (tlb_entry.context.includes_s1 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                tlbi.use_vmid == tlb_entry.context.use_vmid &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
when TLBIOp_VMALLS12 =>
    match = (tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
when TLBIOp_RIPAS2, TLBIPOp_RIPAS2 =>
    match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                tlbi.ipaspace == tlb_entry.context.ipaspace &&
                (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl[1:0]) ||
                   UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    (UInt(tlbi.address[NUM_PABITS-1:0]) <=
                       UInt(entry_end_address[NUM_PABITS-1:0])) &&
                    (UInt(tlbi.end_address[NUM_PABITS-1:0]) >
                       UInt(entry_start_address[NUM_PABITS-1:0])));
        when TLBIOp_RVAA, TLBIPOp_RVAA =>
            match = (tlb_entry.context.includes_s1 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl[1:0]) ||
                       UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level) &&
                    ((tlbi.d128  && tlb_entry.context.isd128) ||
                     (tlbi.d64  && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlbi.d128)) &&
                    UInt(tlbi.address[55:0]) <= UInt(entry_end_address[55:0]) &&
                    UInt(tlbi.end_address[55:0]) > UInt(entry_start_address[55:0]));
        when TLBIOp_RVA, TLBIPOp_RVA =>
            match = (tlb_entry.context.includes_s1 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                       tlb_entry.context.nG == '0') &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl[1:0]) ||
                       UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level) &&
                    ((tlbi.d128  && tlb_entry.context.isd128) ||
                     (tlbi.d64  && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlbi.d128)) &&
                    UInt(tlbi.address[55:0]) <= UInt(entry_end_address[55:0]) &&
                    UInt(tlbi.end_address[55:0]) > UInt(entry_start_address[55:0]));
        when TLBIOp_RPA =>
            let addr = tlb_entry.walkstate.baseaddress.address;
            entry_end_address[NUM_PABITS-1:0] = (addr[NUM_PABITS-1:0] OR
                                                 entry_block_mask[NUM_PABITS-1:0]);
            entry_start_address[NUM_PABITS-1:0] = (addr[NUM_PABITS-1:0] AND
                                                   NOT entry_block_mask[NUM_PABITS-1:0]);
            match = (tlb_entry.context.includes_gpt &&
                    (UInt(tlbi.address[NUM_PABITS-1:0]) <=
                       UInt(entry_end_address[NUM_PABITS-1:0])) &&
                    (UInt(tlbi.end_address[NUM_PABITS-1:0]) >
                       UInt(entry_start_address[NUM_PABITS-1:0])));
        when TLBIOp_PAALL =>
            match = tlb_entry.context.includes_gpt;
    end;

    if tlbi.attr == TLBI_ExcludeXS && tlb_entry.context.xs == '1' then
        match = FALSE;
    end;

    return match;
end;
```

J1.2.3.433 TLBIPRange

```
// TLBIPRange()
// ============
// Extract the input address range information from encoded Xt.

func TLBIPRange(regime : Regime, Xt : bits(128)) => (boolean, bits(2), bits(64), bits(64))
begin
    let valid : boolean = TRUE;
    var start_address : bits(64) = Zeros{};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var end_address : bits(64)   = Zeros{};

    let tg : bits(2)    = Xt[47:46];
    let scale : integer = UInt(Xt[45:44]);
    let num : integer   = UInt(Xt[43:39]);

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);
    end;

    let tg_bits : AddressSize = TGBits(tg);
    // The more-significant bits of the start_address is not updated,
    // as they are not used when performing address matching in TLB
    start_address[55:tg_bits] = Xt[107:64+(tg_bits-12)];

    let range : integer = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range[63:0];

    if end_address[55] != start_address[55] then
        // overflow, saturate it
        end_address = Replicate{9}(start_address[55]) :: Ones{55};
    end;

    return (valid, tg, start_address, end_address);
end;
```

J1.2.3.434 TLBIRange

```
// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.

func TLBIRange(regime : Regime, Xt : bits(64)) => (boolean, bits(2), bits(64), bits(64))
begin
    let valid : boolean = TRUE;
    var start_address : bits(64) = Zeros{};
    var end_address : bits(64)   = Zeros{};

    let tg : bits(2)    = Xt[47:46];
    let scale : integer = UInt(Xt[45:44]);
    let num : integer   = UInt(Xt[43:39]);
    var tg_bits : integer;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);
    end;

    case tg of
        when '01' => // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address[52:16] = Xt[36:0];
                start_address[63:53] = Replicate{11}(Xt[36]);
            else
                start_address[48:12] = Xt[36:0];
                start_address[63:49] = Replicate{15}(Xt[36]);
            end;
        when '10' => // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address[52:16] = Xt[36:0];
                start_address[63:53] = Replicate{11}(Xt[36]);
            else
                start_address[50:14] = Xt[36:0];
                start_address[63:51] = Replicate{13}(Xt[36]);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '11' => // 64KB
            tg_bits = 16;
            start_address[52:16] = Xt[36:0];
            start_address[63:53] = Replicate{11}(Xt[36]);
        otherwise =>
            unreachable;
    end;

    let range : integer = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range[63:0];

    if IsFeatureImplemented(FEAT_LVA3) && end_address[56] != start_address[56] then
        // overflow, saturate it
        end_address = Replicate{8}(start_address[56]) :: Ones{56};
    elsif end_address[52] != start_address[52] then
        // overflow, saturate it
        end_address = Replicate{12}(start_address[52]) :: Ones{52};
    end;

    return (valid, tg, start_address, end_address);
end;
```

J1.2.3.435 VBitOp

```
// VBitOp
// ======
// Vector bit select instruction types.

type VBitOp of enumeration {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

J1.2.4 aarch64/translation

This section includes the following pseudocode functions:

- AArch64_MAIRAttr
- AArch64_CheckBreakpoint
- AArch64_CheckDebug
- AArch64_CheckWatchpoint
- AppendToHDBSS
- CanAppendToHDBSS
- CreateHDBSSEntry
- AArch64_IASize
- AArch64_NextTableBase
- AArch64_PhysicalAddressSize
- AArch64_S1LeafBase
- AArch64_S1SLTTEntryAddress
- AArch64_S1StartLevel
- AArch64_S1TTBaseAddress
- AArch64_S1TTEntryAddress
- AArch64_S2LeafBase
- AArch64_S2SLTTEntryAddress
- AArch64_S2StartLevel
- AArch64_S2TTBaseAddress
- AArch64_S2TTEntryAddress
- AArch64_AddrTop

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64_ContiguousBitFaults
- AArch64_IPAIsOutOfRange
- AArch64_OAOutOfRange
- AArch64_PermissionOverlaysApplied
- AArch64_S1CheckPermissions
- AArch64_S1ComputePermissions
- AArch64_S1DirectBasePermissions
- AArch64_S1HasAlignmentFaultDueToMemType
- AArch64_S1IndirectBasePermissions
- AArch64_S1OAOutOfRange
- AArch64_S1OverlayPermissions
- AArch64_S1TxSZFaults
- AArch64_S2CheckPermissions
- AArch64_S2ComputePermissions
- AArch64_S2DirectBasePermissions
- AArch64_S2HasAlignmentFaultDueToMemType
- AArch64_S2InconsistentSL
- AArch64_S2IndirectBasePermissions
- AArch64_S2InvalidSL
- AArch64_S2OAOutOfRange
- AArch64_S2OverlayPermissions
- AArch64_S2TxSZFaults
- AArch64_VAIsOutOfRange
- AArch64_S2ApplyFWBMemAttrs
- AArch64_GetS1TLBContext
- AArch64_GetS2TLBContext
- AArch64_TLBContextEL10
- AArch64_TLBContextEL2
- AArch64_TLBContextEL20
- AArch64_TLBContextEL3
- AArch64_FullTranslate
- AArch64_MemSwapTableDesc
- AArch64_S1DisabledOutput
- AArch64_S1Translate
- AArch64_S1TreatAsNormalNC
- AArch64_S2Translate
- AArch64_S2TreatAsNormalNC
- AArch64_SetAccessFlag
- AArch64_SetDirtyState
- AArch64_SettingAccessFlagPermitted
- AArch64_SettingDirtyStatePermitted
- AArch64_TranslateAddress
- AArch64_TranslateTagAddress
- AArch64_BlockDescSupported
- AArch64_ContiguousBit
- AArch64_DecodeDescriptorType
- AArch64_S1ApplyOutputPerms

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64_S1ApplyTablePerms
- AArch64_S2ApplyOutputPerms
- AArch64_nTFaults
- AArch64_S1InitialTTWState
- AArch64_S1NextWalkStateLeaf
- AArch64_S1NextWalkStateTable
- AArch64_S1Walk
- AArch64_S2InitialTTWState
- AArch64_S2NextWalkStateLeaf
- AArch64_S2NextWalkStateTable
- AArch64_S2Walk
- AArch64_SS2InitialTTWState
- AArch64_SS2OutputPASpace
- AArch64_GetS1TTWParams
- AArch64_GetS2TTWParams
- AArch64_GetVARange
- AArch64_HaveS1TG
- AArch64_HaveS2TG
- AArch64_MaxTxSZ
- AArch64_NSS2TTWParams
- AArch64_PAMax
- AArch64_RLS2TTWParams
- AArch64_S1DCacheEnabled
- AArch64_S1DecodeTG0
- AArch64_S1DecodeTG1
- AArch64_S1E0POEnabled
- AArch64_S1EPD
- AArch64_S1Enabled
- AArch64_S1ICacheEnabled
- AArch64_S1MinTxSZ
- AArch64_S1POEnabled
- AArch64_S1POR
- AArch64_S1TTBR
- AArch64_S1TTWParamsEL10
- AArch64_S1TTWParamsEL2
- AArch64_S1TTWParamsEL20
- AArch64_S1TTWParamsEL3
- AArch64_S2DecodeTG0
- AArch64_S2MinTxSZ
- AArch64_SS2TTWParams
- S2DCacheEnabled

J1.2.4.1 AArch64_MAIRAttr

```
// AArch64_MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

func AArch64_MAIRAttr(index : integer, mair2 : MAIRType, mair : MAIRType) => bits(8)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert (index < 8 || (IsFeatureImplemented(FEAT_AIE) && (index < 16)));
    if (index > 7) then
        return mair2[(index-8)*:8]; // Read from LSB at MAIR2
    else
        return mair[index*:8];
    end;
end;
```

J1.2.4.2 AArch64_CheckBreakpoint

```
// AArch64_CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

func AArch64_CheckBreakpoint(fault_in : FaultRecord, vaddress : bits(64),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    var fault : FaultRecord = fault_in;
    var match : boolean     = FALSE;
    var addr_match_bp : boolean     = FALSE; // Default assumption that all address match
                                             // breakpoints are inactive or disabled.
    var addr_mismatch_bp : boolean  = FALSE; // Default assumption that all address mismatch
                                             // breakpoints are inactive or disabled.
    var addr_match : boolean        = FALSE;
    var addr_mismatch : boolean     = TRUE;  // Default assumption that the given virtual address
                                             // is outside the range of all address mismatch
                                             // breakpoints.
    var ctxt_match : boolean        = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1 do
        let brkptinfo : BreakpointInfo = AArch64_BreakpointMatch(i, vaddress, accdesc, size);
        if brkptinfo.bptype == BreakpointType_AddrMatch then
            addr_match_bp = TRUE;
            addr_match = addr_match || brkptinfo.match;
        elsif brkptinfo.bptype == BreakpointType_AddrMismatch then
            addr_mismatch_bp = TRUE;
            addr_mismatch = addr_mismatch && !brkptinfo.match;
        elsif brkptinfo.bptype == BreakpointType_CtxtMatch then
            ctxt_match = ctxt_match || brkptinfo.match;
        end;
    end;
    if addr_match_bp && addr_mismatch_bp then
        match = addr_match && addr_mismatch;
    else
        match = (addr_match_bp && addr_match) || (addr_mismatch_bp && addr_mismatch);
    end;

    match = match || ctxt_match;

    if match then
        fault.statuscode = Fault_Debug;
        fault.vaddress   = vaddress;
        if HaltOnBreakpointOrWatchpoint() then
            let reason : bits(6) = DebugHalt_Breakpoint;
            Halt(reason);
        end;
    end;

    return fault;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.3 AArch64_CheckDebug

```
// AArch64_CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

func AArch64_CheckDebug(vaddress : bits(64), accdesc : AccessDescriptor,
                        size : integer) => FaultRecord
begin

    var fault : FaultRecord = NoFault(accdesc, vaddress);
    var generate_exception : boolean;

    let d_side : boolean = IsWatchpointableAccess(accdesc);
    let i_side : boolean = (accdesc.acctype == AccessType_IFETCH);
    if accdesc.acctype == AccessType_NV2 then
        let mask : bit = '0';
        let ss : SecurityState = CurrentSecurityState();
        generate_exception = (AArch64_GenerateDebugExceptionsFrom(EL2, ss, mask) &&
                              MDSCR_EL1().MDE == '1');
    else
        generate_exception = AArch64_GenerateDebugExceptions() && MDSCR_EL1().MDE == '1';
    end;
    let halt : boolean = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64_CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch64_CheckBreakpoint(fault, vaddress, accdesc, size);
        end;
    end;

    return fault;
end;
```

J1.2.4.4 AArch64_CheckWatchpoint

```
// AArch64_CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

func AArch64_CheckWatchpoint(fault_in : FaultRecord, vaddress_in : bits(64),
                             accdesc : AccessDescriptor, size_in : integer) => FaultRecord
begin
    assert !ELUsingAArch32(S1TranslationRegime());
    var fault : FaultRecord          = fault_in;
    var fault_match : FaultRecord    = fault_in;
    var fault_mismatch : FaultRecord = fault_in;
    var vaddress : bits(64)          = vaddress_in;
    var size : integer               = size_in;
    var rounded_match : boolean      = FALSE;
    let original_vaddress : bits(64) = vaddress;
    let original_size : integer = size;
    var addr_match_wp : boolean      = FALSE;// Default assumption that all address
                                             //  match watchpoints are inactive or disabled.
    var addr_mismatch_wp : boolean   = FALSE; // Default assumption that all address mismatch
                                              // watchpoints are inactive or disabled.
    var addr_match : boolean         = FALSE;
    var addr_mismatch : boolean      = TRUE;// Default assumption that the given virtual address is
                                            // outside the range of all address mismatch watchpoints

    // For memory accesses of below type
    // - Contiguous SVE access
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // - SME access
    // - SIMD&FP access when the PE is in Streaming SVE mode
    // each call to this function is such that:
    // - the lowest accessed address is rounded down to the nearest multiple of 16 bytes
    // - the highest accessed address is rounded up to the nearest multiple of 16 bytes
    // Since the WPF field is set if the implementation does rounding, regardless of true or
    // false match, it would be acceptable to return TRUE for either/both of the first and last
    // access.
    if IsRelaxedWatchpointAccess(accdesc) then
        var upper_vaddress : integer = UInt(original_vaddress) + original_size;
        if ConstrainUnpredictableBool(Unpredictable_16BYTEROUNDEDDOWNACCESS) then
            vaddress = AlignDownSize{64}(vaddress, 16);
            rounded_match = TRUE;
        end;
        if ConstrainUnpredictableBool(Unpredictable_16BYTEROUNDEDUPACCESS) then
            upper_vaddress = AlignUpSize(upper_vaddress,16);
            rounded_match = TRUE;
        end;
        size = upper_vaddress - UInt(vaddress);
    end;

    for i = 0 to NumWatchpointsImplemented() - 1 do
        let watchptinfo : WatchpointInfo = AArch64_WatchpointMatch(i, vaddress, size, accdesc);
        if watchptinfo.wptype == WatchpointType_AddrMatch then
            addr_match_wp = TRUE;
            addr_match = addr_match || watchptinfo.value_match;
            if watchptinfo.value_match then
                fault_match.statuscode = Fault_Debug;
                if DBGWCR_EL1(i).LSC[0] == '1' && accdesc.read then
                    fault_match.write = FALSE;
                elsif DBGWCR_EL1(i).LSC[1] == '1' && accdesc.write then
                    fault_match.write = TRUE;
                end;
                fault_match.watchptinfo = watchptinfo;
            end;
        elsif watchptinfo.wptype == WatchpointType_AddrMismatch then
            addr_mismatch_wp = TRUE;
            addr_mismatch = addr_mismatch && !watchptinfo.value_match;
            if !watchptinfo.value_match then
                fault_mismatch.statuscode = Fault_Debug;
                if DBGWCR_EL1(i).LSC[0] == '1' && accdesc.read then
                    fault_mismatch.write = FALSE;
                elsif DBGWCR_EL1(i).LSC[1] == '1' && accdesc.write then
                    fault_mismatch.write = TRUE;
                end;
                fault_mismatch.watchptinfo = watchptinfo;
            end;
        end;
    end;
    if ((addr_match_wp && addr_mismatch_wp && addr_match && addr_mismatch) ||
        (addr_match_wp && !addr_mismatch_wp && addr_match)) then
        fault = fault_match;
    elsif !addr_match_wp && addr_mismatch_wp && addr_mismatch then
        fault = fault_mismatch;
    end;
    fault.vaddress = vaddress;
    fault.watchptinfo.maybe_false_match = rounded_match;
    if (fault.statuscode == Fault_Debug && HaltOnBreakpointOrWatchpoint() &&
            !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first)) then
        let reason : bits(6) = DebugHalt_Watchpoint;
        EDWAR() = fault.vaddress;
        let is_async : boolean = FALSE;
        Halt(reason, is_async, fault);
    end;
    return fault;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.5 AppendToHDBSS

```
// AppendToHDBSS()
// ===============
// Appends an entry to the HDBSS when the dirty state of a stage 2 descriptor is updated
// from writable-clean to writable-dirty by hardware.

func AppendToHDBSS(fault_in : FaultRecord, ipa_in : FullAddress, accdesc : AccessDescriptor,
                   walkparams : S2TTWParams, level : integer) => FaultRecord
begin
    assert CanAppendToHDBSS();

    var fault : FaultRecord  = fault_in;
    var ipa : FullAddress    = ipa_in;
    let hdbss_size : integer{} = UInt(HDBSSBR_EL2().SZ) as integer{0..9};

    var hdbss_addrdesc : AddressDescriptor;

    var baddr : bits(NUM_PABITS) = HDBSSBR_EL2().BADDR[NUM_PABITS-13 : 0] :: Zeros{12};
    baddr[11 + hdbss_size : 12]  = Zeros{hdbss_size};

    hdbss_addrdesc.paddress.address = baddr + (8 * UInt(HDBSSPROD_EL2().INDEX));
    let nse2 : bit = '0';     // NSE2 has the Effective value of 0 within a PE.
    hdbss_addrdesc.paddress.paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(),
                                                    EffectiveSCR_EL3_NS());

    // Accesses to the HDBSS use the same memory attributes as used for stage 2 translation walks.
    hdbss_addrdesc.memattrs         = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    let hdbss_access : AccessDescriptor = CreateAccDescHDBSS(accdesc);
    hdbss_addrdesc.mecid            = AArch64_S2TTWalkMECID(walkparams.emec, accdesc.ss);

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(hdbss_addrdesc, hdbss_access);

        if fault.gpcf.gpf != GPCF_None then
            if (ImpDefBool(
                  "GPC fault on HDBSSS write reported in HDBSSPROD_EL2")) then
                HDBSSPROD_EL2().FSC = '101000';
            else
                fault.statuscode = Fault_GPCFOnWalk;
                fault.paddress   = hdbss_addrdesc.paddress;
                fault.level      = level;
                fault.gpcfs2walk = TRUE;
                fault.hdbssf     = TRUE;
            end;

            return fault;
        end;
    end;

    // The reported IPA must be aligned to the size of the translation.
    let lsb : AddressSize = TranslationSize(walkparams.d128, walkparams.tgx, level);
    ipa.address           = ipa.address[NUM_PABITS-1:lsb] :: Zeros{lsb};
    var hdbss_entry : bits(64) = CreateHDBSSEntry(ipa, hdbss_access.ss, level);

    if walkparams.ee == '1' then
        hdbss_entry = BigEndianReverse{64}(hdbss_entry);
    end;

    let memstatus : PhysMemRetStatus = PhysMemWrite{64}(hdbss_addrdesc, hdbss_access,
                                                        hdbss_entry);

    if IsFault(memstatus) then
        if (ImpDefBool(
              "External Abort on HDBSS write reported in HDBSSPROD_EL2")) then
            HDBSSPROD_EL2().FSC = '010000';
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let iswrite : boolean = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, hdbss_addrdesc,
                                           hdbss_access, 8, fault);
            fault.level  = level;
            fault.hdbssf = TRUE;
        end;
    else
        HDBSSPROD_EL2().INDEX = HDBSSPROD_EL2().INDEX + 1;
    end;

    return fault;
end;
```

J1.2.4.6 CanAppendToHDBSS

```
// CanAppendToHDBSS()
// ==================
// Return TRUE if HDBSS can be appended.

readonly func CanAppendToHDBSS() => boolean
begin
    if !IsFeatureImplemented(FEAT_HDBSS) then
        return FALSE;
    end;
    assert EL2Enabled();
    // The PE cannot append entries to the HDBSS if HDBSSPROD_EL2.FSC is
    // any other value than 0b000000, or HDBSS buffer is full.

    if ((UInt(HDBSSPROD_EL2().INDEX) >= ((2 ^ (UInt(HDBSSBR_EL2().SZ) + 12)) DIV 8)) ||
         (HDBSSPROD_EL2().FSC != '000000')) then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

J1.2.4.7 CreateHDBSSEntry

```
// CreateHDBSSEntry()
// ==================
// Returns a HDBSS entry.

func CreateHDBSSEntry(ipa : FullAddress, ss : SecurityState, level : integer) => bits(64)
begin
    let ns_ipa : bit = if ss == SS_Secure && ipa.paspace == PAS_NonSecure then '1' else '0';
    return ZeroExtend{64}(ipa.address[NUM_PABITS-1:12] ::
                          ns_ipa :: Zeros{7} :: level[2:0] :: '1');
end;
```

J1.2.4.8 AArch64_IASize

```
// AArch64_IASize()
// ================
// Retrieve the number of bits containing the input address

func AArch64_IASize(txsz : bits(6)) => AddressSize
begin
    return (64 - UInt(txsz)) as AddressSize;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.9 AArch64_NextTableBase

```
// AArch64_NextTableBase()
// =======================
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

func AArch64_NextTableBase{N}(descriptor : bits(N), d128 : bit, skl : bits(2),
                              ds : bit, tgx : TGx) => bits(NUM_PABITS)
begin
    var tablebase : bits(NUM_PABITS) = Zeros{};
    let granulebits : AddressSize = TGxGranuleBits(tgx);
    var tablesize : integer;

    if d128 == '1' then
        let descsizelog2 : integer = 4;
        let stride : integer = granulebits - descsizelog2;
        tablesize = stride*(1 + UInt(skl)) + descsizelog2;
    else
        tablesize = granulebits;
    end;

    case tgx of
        when TGx_4KB =>  tablebase[47:12] = descriptor[47:12];
        when TGx_16KB => tablebase[47:14] = descriptor[47:14];
        when TGx_64KB => tablebase[47:16] = descriptor[47:16];
    end;

    tablebase = AlignDownP2{NUM_PABITS}(tablebase, tablesize as integer{0..NUM_PABITS});

    if d128 == '1' then
        tablebase[55:48] = descriptor[55:48];
    elsif tgx == TGx_64KB && (AArch64_PAMax() >= 52 ||
            ImpDefBool("descriptor[15:12] for 64KB granule are OA[51:48]")) then
        tablebase[51:48] = descriptor[15:12];
    elsif ds == '1' then
        tablebase[51:48] = descriptor[9:8]::descriptor[49:48];
    end;

    return tablebase;
end;
```

J1.2.4.10 AArch64_PhysicalAddressSize

```
// AArch64_PhysicalAddressSize()
// =============================
// Retrieve the number of bits bounding the physical address

func AArch64_PhysicalAddressSize(d128 : bit, ds : bit, encoded_ps : bits(3),
                                 tgx : TGx) => AddressSize
begin
    var ps : integer;
    var max_ps : integer;

    case encoded_ps of
        when '000' =>  ps = 32;
        when '001' =>  ps = 36;
        when '010' =>  ps = 40;
        when '011' =>  ps = 42;
        when '100' =>  ps = 44;
        when '101' =>  ps = 48;
        when '110' =>  ps = 52;
        when '111' =>  ps = 56;
    end;

    if d128 == '1' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        max_ps = AArch64_PAMax();
    elsif IsFeatureImplemented(FEAT_LPA) && (tgx == TGx_64KB || ds == '1') then
        max_ps = Min(52, AArch64_PAMax());
    else
        max_ps = Min(48, AArch64_PAMax());
    end;

    return Min(ps, max_ps) as AddressSize;
end;
```

J1.2.4.11 AArch64_S1LeafBase

```
// AArch64_S1LeafBase()
// ====================
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

func AArch64_S1LeafBase{N}(descriptor : bits(N), walkparams : S1TTWParams,
                           level : integer) => bits(NUM_PABITS)
begin
    var leafbase : bits(NUM_PABITS) = Zeros{};

    let granulebits : AddressSize  = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{}   = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer = granulebits - descsizelog2;
    let leafsize : integer = granulebits + stride * (FINAL_LEVEL - level);

    leafbase[47:0] = AlignDownP2{48}(descriptor[47:0], leafsize as integer{0..48});

    if walkparams.d128 == '1' then
        leafbase[55:48] = descriptor[55:48];
    elsif walkparams.tgx == TGx_64KB && (AArch64_PAMax() >= 52 ||
            ImpDefBool("descriptor[15:12] for 64KB granule are OA[51:48]")) then
        leafbase[51:48] = descriptor[15:12];
    elsif walkparams.ds == '1' then
        leafbase[51:48] = descriptor[9:8,49:48];
    end;

    return leafbase;
end;
```

J1.2.4.12 AArch64_S1SLTTEntryAddress

```
// AArch64_S1SLTTEntryAddress()
// ============================
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

func AArch64_S1SLTTEntryAddress(level : integer, walkparams : S1TTWParams,
                                ia : bits(64), tablebase : FullAddress) => FullAddress
begin
    // Input Address size
    let iasize : AddressSize       = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize  = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{}       = granulebits - descsizelog2;
    let levels : integer         = FINAL_LEVEL - level;

    var index : bits(NUM_PABITS);
    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let msb : AddressSize = (iasize - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ia[msb:lsb]::Zeros{descsizelog2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

J1.2.4.13 AArch64_S1StartLevel

```
// AArch64_S1StartLevel()
// ======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

func AArch64_S1StartLevel(walkparams : S1TTWParams) => integer
begin
    // Input Address size
    let iasize : AddressSize       = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize  = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer       = granulebits - descsizelog2;
    var s1startlevel : integer = FINAL_LEVEL - (((iasize-1) - granulebits) DIVRM stride);
    if walkparams.d128 == '1' then
        s1startlevel = s1startlevel + UInt(walkparams.skl);
    end;
    return s1startlevel;
end;
```

J1.2.4.14 AArch64_S1TTBaseAddress

```
// AArch64_S1TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 1

func AArch64_S1TTBaseAddress{N}(walkparams : S1TTWParams,
                                regime : Regime, ttbr : bits(N)) => bits(NUM_PABITS)
begin
    var tablebase : bits(NUM_PABITS) = Zeros{};

    // Input Address size
    let iasize : AddressSize      = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{}    = granulebits - descsizelog2;
    let startlevel : integer  = AArch64_S1StartLevel(walkparams);
    let levels : integer      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    var tsize : integer = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if regime == Regime_EL3 then
            tablebase[55:5] = ttbr[55:5];
        else
            tablebase[55:5] = ttbr[87:80]::ttbr[47:5];
        end;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
            (IsFeatureImplemented(FEAT_LPA) ||
             ImpDefBool("BADDR expresses 52 bits for 64KB granule"))) then
        tsize = Max(tsize, 6);
        tablebase[51:6] = ttbr[5:2]::ttbr[47:6];
    else
        tablebase[47:1] = ttbr[47:1];
    end;
    tablebase = AlignDownP2{NUM_PABITS}(tablebase,tsize as integer{0..NUM_PABITS});
    return tablebase;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.4.15 AArch64_S1TTEntryAddress

```
// AArch64_S1TTEntryAddress()
// ==========================
// Compute translation table descriptor address within the table pointed to by
// the table base

func AArch64_S1TTEntryAddress{N}(level : integer, walkparams : S1TTWParams, skl : bits(2),
                                 ia : bits(64), tablebase : FullAddress,
                                 descriptor : bits(N)) => FullAddress
begin
    // Input Address size
    let iasize : AddressSize      = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{}      = granulebits - descsizelog2;
    let levels : integer        = FINAL_LEVEL - level;

    var index : bits(NUM_PABITS);

    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let nstride : integer = if walkparams.d128 == '1' then UInt(skl) + 1 else 1;
    let msb : AddressSize = ((lsb + (stride * nstride)) - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ia[msb:lsb]::Zeros{descsizelog2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

J1.2.4.16 AArch64_S2LeafBase

```
// AArch64_S2LeafBase()
// ====================
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

func AArch64_S2LeafBase{N}(descriptor : bits(N), walkparams : S2TTWParams,
                           level : integer) => bits(NUM_PABITS)
begin
    var leafbase : bits(NUM_PABITS) = Zeros{};

    let granulebits : AddressSize  = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer   = granulebits - descsizelog2;
    let leafsize : integer = granulebits + stride * (FINAL_LEVEL - level);

    leafbase[47:0] = AlignDownP2{48}(descriptor[47:0], leafsize as integer{0..48});

    if walkparams.d128 == '1' then
        leafbase[55:48] = descriptor[55:48];
    elsif walkparams.tgx == TGx_64KB && (AArch64_PAMax() >= 52 ||
            (ImpDefBool(
               "descriptor[15:12] for 64KB granule are OA[51:48]"))) then
        leafbase[51:48] = descriptor[15:12];
    elsif walkparams.ds == '1' then
        leafbase[51:48] = descriptor[9:8]::descriptor[49:48];
    end;

    return leafbase;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.17 AArch64_S2SLTTEntryAddress

```
// AArch64_S2SLTTEntryAddress()
// ============================
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

func AArch64_S2SLTTEntryAddress(walkparams : S2TTWParams, ipa : bits(NUM_PABITS),
                                tablebase : FullAddress) => FullAddress
begin
    let startlevel   = AArch64_S2StartLevel(walkparams);
    let iasize       = AArch64_IASize(walkparams.txsz);
    let granulebits  = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{}       = granulebits - descsizelog2;
    let levels : integer         = FINAL_LEVEL - startlevel;

    var index : bits(NUM_PABITS);
    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let msb : AddressSize = (iasize - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ipa[msb:lsb]::Zeros{descsizelog2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

J1.2.4.18 AArch64_S2StartLevel

```
// AArch64_S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

func AArch64_S2StartLevel(walkparams : S2TTWParams) => integer
begin
    if walkparams.d128 == '1' then
        let iasize       : AddressSize = AArch64_IASize(walkparams.txsz);
        let granulebits  : AddressSize = TGxGranuleBits(walkparams.tgx);
        let descsizelog2 : integer{} = 4;
        let stride : integer = granulebits - descsizelog2;
        var s2startlevel : integer = FINAL_LEVEL - (((iasize-1) - granulebits) DIVRM stride);
        s2startlevel = s2startlevel + UInt(walkparams.skl);

        return s2startlevel;
    end;

    case walkparams.tgx of
        when TGx_4KB =>
            case walkparams.sl2::walkparams.sl0 of
                when '000' => return 2;
                when '001' => return 1;
                when '010' => return 0;
                when '011' => return 3;
                when '100' => return -1;
            end;
        when TGx_16KB =>
            case walkparams.sl0 of
                when '00' => return 3;
                when '01' => return 2;
                when '10' => return 1;
                when '11' => return 0;
            end;
        when TGx_64KB =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            case walkparams.sl0 of
                when '00' => return 3;
                when '01' => return 2;
                when '10' => return 1;
            end;
    end;
end;
```

J1.2.4.19 AArch64_S2TTBaseAddress

```
// AArch64_S2TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 2

func AArch64_S2TTBaseAddress{N}(walkparams : S2TTWParams, paspace : PASpace,
                                ttbr : bits(N)) => bits(NUM_PABITS)
begin
    var tablebase : bits(NUM_PABITS) = Zeros{};

    // Input Address size
    let iasize : AddressSize      = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{}  = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{}        = granulebits - descsizelog2;
    let startlevel : integer      = AArch64_S2StartLevel(walkparams);
    let levels : integer          = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    var tsize : integer = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if paspace == PAS_Secure then
            tablebase[55:5] = ttbr[55:5];
        else
            tablebase[55:5] = ttbr[87:80]::ttbr[47:5];
        end;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
            (IsFeatureImplemented(FEAT_LPA) ||
             ImpDefBool("BADDR expresses 52 bits for 64KB granule"))) then
        tsize = Max(tsize, 6);
        tablebase[51:6] = ttbr[5:2]::ttbr[47:6];
    else
        tablebase[47:1] = ttbr[47:1];
    end;
    tablebase = AlignDownP2{NUM_PABITS}(tablebase, tsize as integer{0..NUM_PABITS});
    return tablebase;
end;
```

J1.2.4.20 AArch64_S2TTEntryAddress

```
// AArch64_S2TTEntryAddress()
// ==========================
// Compute translation table descriptor address within the table pointed to by
// the table base

func AArch64_S2TTEntryAddress(level : integer, walkparams : S2TTWParams, skl :  bits(2),
                              ipa :  bits(NUM_PABITS), tablebase : FullAddress) => FullAddress
begin
    let ipasize : AddressSize     = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{}      = granulebits - descsizelog2;
    let levels : integer        = FINAL_LEVEL - level;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var index : bits(NUM_PABITS);

    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let nstride : integer = if walkparams.d128 == '1' then UInt(skl) + 1 else 1;
    let msb : AddressSize = ((lsb + (stride * nstride)) - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ipa[msb:lsb]::Zeros{descsizelog2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

J1.2.4.21 AArch64_AddrTop

```
// AArch64_AddrTop()
// =================
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

func AArch64_AddrTop(tbid : bit, acctype : AccessType, tbi : bit) => AddressSize
begin
    if tbid == '1' && acctype == AccessType_IFETCH then
        return 63;
    end;

    if tbi == '1' then
        return 55;
    else
        return 63;
    end;
end;
```

J1.2.4.22 AArch64_ContiguousBitFaults

```
// AArch64_ContiguousBitFaults()
// =============================
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

func AArch64_ContiguousBitFaults(d128 : bit, txsz : bits(6), tgx : TGx, level : integer) => boolean
begin
    // Input Address size
    let iasize : AddressSize = AArch64_IASize(txsz);
    // Translation size
    let tsize : integer = TranslationSize(d128, tgx, level) + ContiguousSize(d128, tgx, level);

    return (tsize > iasize &&
            ImpDefBool("Translation fault on misprogrammed contiguous bit"));
end;
```

J1.2.4.23 AArch64_IPAIsOutOfRange

```
// AArch64_IPAIsOutOfRange()
// =========================
// Check bits not resolved by translation are ZERO

func AArch64_IPAIsOutOfRange(ipa : bits(NUM_PABITS), walkparams : S2TTWParams) => boolean
begin
    //Input Address size
    let iasize : integer{} = AArch64_IASize(walkparams.txsz);

    if iasize < NUM_PABITS then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return !IsZero(ipa[NUM_PABITS-1:iasize]);
    else
        return FALSE;
    end;
end;
```

J1.2.4.24 AArch64_OAOutOfRange

```
// AArch64_OAOutOfRange()
// ======================
// Returns whether output address is expressed in the configured size number of bits

func AArch64_OAOutOfRange(address : bits(NUM_PABITS), d128 : bit,
                          ds : bit, ps : bits(3), tgx : TGx) => boolean
begin
    // Output Address size
    let oasize : integer{} = AArch64_PhysicalAddressSize(d128, ds, ps, tgx);

    if oasize < NUM_PABITS then
        return !IsZero(address[NUM_PABITS-1:oasize]);
    else
        return FALSE;
    end;
end;
```

J1.2.4.25 AArch64_PermissionOverlaysApplied

```
// AArch64_PermissionOverlaysApplied()
// ===================================
// Returns TRUE if Permission overlays are applied for the given access type.

func AArch64_PermissionsOverlaysApplied(acctype : AccessType) => boolean
begin
    case acctype of
        when AccessType_SPE        => return FALSE;
        when AccessType_TRBE       => return FALSE;
        otherwise                  => return TRUE;
    end;
end;
```

J1.2.4.26 AArch64_S1CheckPermissions

```
// AArch64_S1CheckPermissions()
// ============================
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

func AArch64_S1CheckPermissions(fault_in : FaultRecord, va : bits(64), size : integer,
                                regime : Regime, walkstate : TTWState, walkparams : S1TTWParams,
                                accdesc : AccessDescriptor) => FaultRecord
begin
    var fault : FaultRecord = fault_in;
    let permissions : Permissions = walkstate.permissions;
    var s1perms : S1AccessControls = AArch64_S1ComputePermissions(regime, walkstate,
                                                                  walkparams, accdesc);

    if accdesc.acctype == AccessType_IFETCH then
        // Flag the access is from a guarded page
        SetInGuardedPage(walkstate.guardedpage == '1' && s1perms.x == '1');

        if s1perms.overlay && s1perms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (walkstate.memattrs.memtype == MemType_Device &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;
        elsif s1perms.x == '0' then
            fault.statuscode = Fault_Permission;
        end;
    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.hd != '1' && walkparams.pie == '1' &&
                     permissions.ndirty == '1') then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
            end;
        // DC from privileged context which clean cannot generate a Permission fault
        elsif accdesc.el == EL0 then
            if s1perms.overlay && s1perms.or == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif (walkparams.cmow == '1' &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s1perms.overlay && s1perms.ow == '0') then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif s1perms.r == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s1perms.w == '0') then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' && walkparams.hd != '1' && walkparams.pie == '1' &&
                     permissions.ndirty == '1' && accdesc.cacheop == CacheOp_CleanInvalidate) then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
            end;
        end;
    elsif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == EL0 then
            if (s1perms.overlay && s1perms.or == '0' &&
                  ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif walkparams.cmow == '1' && s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif (s1perms.r == '0' &&
                  ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
                fault.statuscode = Fault_Permission;
            elsif walkparams.cmow == '1' && s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' && walkparams.hd != '1' && walkparams.pie == '1' &&
                     permissions.ndirty == '1') then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
            end;
        end;
    elsif IsFeatureImplemented(FEAT_GCS) && accdesc.acctype == AccessType_GCS then
        if s1perms.gcs == '0' then
            fault.statuscode = Fault_Permission;
        elsif accdesc.write && walkparams.hd != '1' && permissions.ndirty == '1' then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            fault.write      = TRUE;
        end;
    elsif accdesc.read && s1perms.overlay && s1perms.or == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.overlay && s1perms.ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = TRUE;
    elsif accdesc.read && s1perms.r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;
    elsif (accdesc.write && accdesc.tagaccess &&
             walkstate.memattrs.tags == MemTag_CanonicallyTagged) then
        fault.statuscode   = Fault_Permission;
        fault.write        = TRUE;
        fault.s1tagnotdata = TRUE;
    elsif (accdesc.write && walkparams.hd != '1' && walkparams.pie == '1' &&
             permissions.ndirty == '1') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;
        fault.write      = TRUE;
    end;

    return fault;
end;
```

J1.2.4.27 AArch64_S1ComputePermissions

```
// AArch64_S1ComputePermissions()
// ==============================
// Computes the overall stage 1 permissions

func AArch64_S1ComputePermissions(regime : Regime, walkstate : TTWState,
                                  walkparams : S1TTWParams,
                                  accdesc : AccessDescriptor) => S1AccessControls
begin
    let permissions : Permissions = walkstate.permissions;
    var s1perms : S1AccessControls;

    if walkparams.pie == '1' then
        s1perms = AArch64_S1IndirectBasePermissions(regime, walkstate, walkparams, accdesc);
    else
        s1perms = AArch64_S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);
    end;

    if accdesc.el == EL0 && !AArch64_S1E0POEnabled(regime, walkparams.nv1) then
        s1perms.overlay = FALSE;
    elsif accdesc.el != EL0 && !AArch64_S1POEnabled(regime) then
        s1perms.overlay = FALSE;
    end;

    if s1perms.overlay then
        let s1overlay_perms : S1AccessControls = AArch64_S1OverlayPermissions(regime,
                                                                              walkstate,
                                                                              accdesc);
        s1perms.or = s1overlay_perms.or;
        s1perms.ow = s1overlay_perms.ow;
        s1perms.ox = s1overlay_perms.ox;
    end;

    if s1perms.overlay then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // If WXN and the overlay X permission is present, the overlay W permission is removed.
        s1perms.ow = s1perms.ow AND NOT(s1perms.wxn AND s1perms.ox);
    else
        // If WXN and the W and X permissions are present, the X permission is removed.
        // The W and X permissions are factored into the WXN computation.
        s1perms.x  = s1perms.x AND NOT(s1perms.wxn);
    end;

    return s1perms;
end;
```

J1.2.4.28 AArch64_S1DirectBasePermissions

```
// AArch64_S1DirectBasePermissions()
// ================================
// Computes the stage 1 direct base permissions

func AArch64_S1DirectBasePermissions(regime : Regime, walkstate : TTWState,
                                     walkparams : S1TTWParams,
                                     accdesc : AccessDescriptor) => S1AccessControls
begin
    var r,  w,  x,  wxn  : bit;
    var pr, pw, px, pwxn : bit;
    var ur, uw, ux, uwxn : bit;
    var permissions : Permissions = walkstate.permissions;
    var s1perms : S1AccessControls;
    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if permissions.dbm == '1' && walkparams.hd == '1' then
        permissions.ap[2] = '0';
    end;

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case permissions.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only
        end;

        // Apply hierarchical permissions
        case permissions.ap_table of
            when '00' => (pr,pw,ur,uw) = (pr, pw, ur, uw);  // No effect
            when '01' => (pr,pw,ur,uw) = (pr, pw,'0','0');  // Privileged access
            when '10' => (pr,pw,ur,uw) = (pr,'0', ur,'0');  // Read-only
            when '11' => (pr,pw,ur,uw) = (pr,'0','0','0');  // Read-only, privileged access
        end;

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);
        pwxn = walkparams.wxn AND pw AND px;
        uwxn = walkparams.wxn AND uw AND ux;

        if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime == Regime_EL10 &&
              walkparams.nv1 == '1')) then
            if (ImpDefBool("SCR_EL3.SIF affects EPAN") && accdesc.ss == SS_Secure &&
                  walkstate.baseaddress.paspace == PAS_NonSecure && walkparams.sif == '1') then
                ux = '0';
            end;

            if (ImpDefBool("Realm EL2&0 regime affects EPAN") && accdesc.ss == SS_Realm &&
                  regime == Regime_EL20 && walkstate.baseaddress.paspace != PAS_Realm) then
                ux = '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;

            let pan : bit = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);
        end;

    else
        // Apply leaf permissions
        case permissions.ap[2] of
            when '0' => (pr,pw) = ('1','1'); // No effect
            when '1' => (pr,pw) = ('1','0'); // Read-only
        end;

        // Apply hierarchical permissions
        case permissions.ap_table[1] of
            when '0' => (pr,pw) = (pr, pw);  // No effect
            when '1' => (pr,pw) = (pr,'0');  // Read-only
        end;

        px   = NOT(permissions.xn OR permissions.xn_table);
        pwxn = walkparams.wxn AND pw AND px;
    end;

    (r,w,x,wxn) = if accdesc.el == EL0 then (ur,uw,ux,uwxn) else (pr,pw,px,pwxn);

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
    end;
    // Prevent execution from non-Root space by Root
    if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        x = '0';
    end;
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
            walkstate.baseaddress.paspace != PAS_Realm) then
        x = '0';
    end;


    s1perms.r   = r;
    s1perms.w   = w;
    s1perms.x   = x;
    s1perms.gcs = '0';
    s1perms.wxn = wxn;
    s1perms.overlay = TRUE;

    return s1perms;
end;
```

J1.2.4.29 AArch64_S1HasAlignmentFaultDueToMemType

```
// AArch64_S1HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 1 output fails alignment requirement on data accesses due to memory type

func AArch64_S1HasAlignmentFaultDueToMemType(regime : Regime, accdesc : AccessDescriptor,
                                             aligned : boolean, ntlsmd : bit,
                                             memattrs : MemoryAttributes) => boolean
begin
    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && AArch64_S1DCacheEnabled(regime)) &&
              ConstrainUnpredictableBool(Unpredictable_LSE2_ALIGNMENT_FAULT)) then
            return TRUE;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif ((accdesc.acctype == AccessType_DCZero && accdesc.cachetype == CacheType_Tag) ||
              accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE) == Constraint_FAULT;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.device != DeviceType_GRE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

J1.2.4.30 AArch64_S1IndirectBasePermissions

```
// AArch64_S1IndirectBasePermissions()
// ===================================
// Computes the stage 1 indirect base permissions

func AArch64_S1IndirectBasePermissions(regime : Regime, walkstate : TTWState,
                                       walkparams : S1TTWParams,
                                       accdesc : AccessDescriptor) => S1AccessControls
begin

    var r,  w,  x,  gcs,  wxn,   overlay : bit;
    var pr, pw, px, pgcs, pwxn, p_overlay : bit;
    var ur, uw, ux, ugcs, uwxn, u_overlay : bit;
    let permissions : Permissions = walkstate.permissions;
    var s1perms : S1AccessControls;

    // Apply privileged indirect permissions
    case permissions.ppi of
        when '0000' => (pr,pw,px,pgcs) = ('0','0','0','0'); // No access
        when '0001' => (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '0010' => (pr,pw,px,pgcs) = ('0','0','1','0'); // Privileged execute
        when '0011' => (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '0100' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '0101' => (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '0110' => (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '0111' => (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1000' => (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '1001' => (pr,pw,px,pgcs) = ('1','0','0','1'); // Privileged read and gcs
        when '1010' => (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '1011' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1100' => (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '1101' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1110' => (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1111' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
    end;

    p_overlay = NOT(permissions.ppi[3]);
    pwxn = if permissions.ppi == '0110' then '1' else '0';
    var reserved_upi : boolean = FALSE;

    if HasUnprivileged(regime) then
        // Apply unprivileged indirect permissions
        case permissions.upi of
            when '0000' => (ur,uw,ux,ugcs) = ('0','0','0','0'); // No access
            when '0001' => (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '0010' => (ur,uw,ux,ugcs) = ('0','0','1','0'); // Unprivileged execute
            when '0011' => (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '0100' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0');         // Reserved
                reserved_upi    = TRUE;
            when '0101' => (ur,uw,ux,ugcs)=('1','1','0','0');// Unprivileged read and write
            when '0110' => (ur,uw,ux,ugcs)=('1','1','1','0');// Unprivileged read, write and execute
            when '0111' => (ur,uw,ux,ugcs)=('1','1','1','0');// Unprivileged read, write and execute
            when '1000' => (ur,uw,ux,ugcs)=('1','0','0','0');// Unprivileged read
            when '1001' => (ur,uw,ux,ugcs)=('1','0','0','1');// Unprivileged read and gcs
            when '1010' => (ur,uw,ux,ugcs)=('1','0','1','0');// Unprivileged read and execute
            when '1011' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0');          // Reserved
                reserved_upi    = TRUE;
            when '1100' => (ur,uw,ux,ugcs) =('1','1','0','0');// Unprivileged read and write
            when '1101' =>
                (ur,uw,ux,ugcs) =('0','0','0','0');           // Reserved
                reserved_upi    = TRUE;
            when '1110' => (ur,uw,ux,ugcs) =('1','1','1','0');// Unprivileged read, write
                                                              // and execute
            when '1111' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0');          // Reserved
                reserved_upi    = TRUE;

        end;

        u_overlay = NOT(permissions.upi[3]);
        uwxn = if permissions.upi == '0110' then '1' else '0';

        // If the decoded permissions has either px or pgcs along with either uw or ugcs,
        // then all effective Stage 1 Base Permissions are set to 0
        if ((px == '1' || pgcs == '1') && (uw == '1' || ugcs == '1')) then
            (pr,pw,px,pgcs) = ('0','0','0','0');
            (ur,uw,ux,ugcs) = ('0','0','0','0');
        end;
    end;

    // Prevent execution from Non-secure space by PE in secure state
    if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        pgcs = '0';
        if HasUnprivileged(regime) then ugcs = '0'; end;

        if walkparams.sif == '1' then
            px = '0';
            if HasUnprivileged(regime) then ux = '0'; end;
        end;
    end;

    // Prevent execution from non-Root space by Root
    if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        (px, pgcs)   = ('0','0');
        if HasUnprivileged(regime) then
            (ux, ugcs)   = ('0','0');
        end;
    end;

    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
          walkstate.baseaddress.paspace != PAS_Realm) then
        (px, pgcs)   = ('0','0');
        if HasUnprivileged(regime) then
            (ux, ugcs)   = ('0','0');
        end;
    end;

    if (HasUnprivileged(regime) && IsFeatureImplemented(FEAT_PAN) && accdesc.pan &&
          !(regime == Regime_EL10 && walkparams.nv1 == '1') &&
          PSTATE.PAN == '1' && !IsZero(permissions.upi)) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if reserved_upi then
            if ImpDefBool("PAN applies when S1UnprivBasePerm is reserved") then
                (pr,pw) = ('0','0');
            end;
        elsif (!IsZero(ur::uw::ux::ugcs) ||
                 ImpDefBool("PAN applies when S1UnprivBasePerms are removed")) then
            (pr,pw) = ('0','0');
        end;
    end;

    if accdesc.el == EL0 then
        (r,w,x,gcs,wxn,overlay) = (ur,uw,ux,ugcs,uwxn,u_overlay);
    else
        (r,w,x,gcs,wxn,overlay) = (pr,pw,px,pgcs,pwxn,p_overlay);
    end;

    s1perms.r       = r;
    s1perms.w       = w;
    s1perms.x       = x;
    s1perms.gcs     = gcs;
    s1perms.wxn     = wxn;
    s1perms.overlay = overlay == '1';

    return s1perms;
end;
```

J1.2.4.31 AArch64_S1OAOutOfRange

```
// AArch64_S1OAOutOfRange()
// ========================
// Returns whether stage 1 output address is expressed in the configured size number of bits

func AArch64_S1OAOutOfRange(address : bits(NUM_PABITS), walkparams : S1TTWParams) => boolean
begin
    return AArch64_OAOutOfRange(address, walkparams.d128, walkparams.ds, walkparams.ps,
                                walkparams.tgx);
end;
```

J1.2.4.32 AArch64_S1OverlayPermissions

```
// AArch64_S1OverlayPermissions()
// ==============================
// Computes the stage 1 overlay permissions

func AArch64_S1OverlayPermissions(regime : Regime, walkstate : TTWState,
                                  accdesc : AccessDescriptor) => S1AccessControls
begin

    var r,  w,  x : bit;
    var pr, pw, px : bit;
    var ur, uw, ux : bit;
    let permissions : Permissions = walkstate.permissions;
    var s1overlay_perms : S1AccessControls;

    let por : S1PORType = AArch64_S1POR(regime);
    let bit_index : integer{} = 4 * UInt(permissions.po_index);

    let ppo : bits(4) = por[bit_index+3:bit_index];

    // Apply privileged overlay permissions
    case ppo of
        when '0000' => (pr,pw,px) = ('0','0','0'); // No access
        when '0001' => (pr,pw,px) = ('1','0','0'); // Privileged read
        when '0010' => (pr,pw,px) = ('0','0','1'); // Privileged execute
        when '0011' => (pr,pw,px) = ('1','0','1'); // Privileged read and execute
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '0100' => (pr,pw,px) = ('0','1','0'); // Privileged write
        when '0101' => (pr,pw,px) = ('1','1','0'); // Privileged read and write
        when '0110' => (pr,pw,px) = ('0','1','1'); // Privileged write and execute
        when '0111' => (pr,pw,px) = ('1','1','1'); // Privileged read, write and execute
        when '1xxx' => (pr,pw,px) = ('0','0','0'); // Reserved
    end;

    if HasUnprivileged(regime) then
        var upo : bits(4) = '0000';
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
               ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            upo = POR_EL0()[bit_index+3:bit_index];
        end;

        // Apply unprivileged overlay permissions
        case upo of
            when '0000' => (ur,uw,ux) = ('0','0','0'); // No access
            when '0001' => (ur,uw,ux) = ('1','0','0'); // Unprivileged read
            when '0010' => (ur,uw,ux) = ('0','0','1'); // Unprivileged execute
            when '0011' => (ur,uw,ux) = ('1','0','1'); // Unprivileged read and execute
            when '0100' => (ur,uw,ux) = ('0','1','0'); // Unprivileged write
            when '0101' => (ur,uw,ux) = ('1','1','0'); // Unprivileged read and write
            when '0110' => (ur,uw,ux) = ('0','1','1'); // Unprivileged write and execute
            when '0111' => (ur,uw,ux) = ('1','1','1'); // Unprivileged read, write and execute
            when '1xxx' => (ur,uw,ux) = ('0','0','0'); // Reserved
        end;
    end;

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    s1overlay_perms.or = r;
    s1overlay_perms.ow = w;
    s1overlay_perms.ox = x;

    return s1overlay_perms;
end;
```

J1.2.4.33 AArch64_S1TxSZFaults

```
// AArch64_S1TxSZFaults()
// ======================
// Detect whether configuration of stage 1 TxSZ field generates a fault

func AArch64_S1TxSZFaults(regime : Regime, walkparams : S1TTWParams) => boolean
begin
    let mintxsz : integer = AArch64_S1MinTxSZ(regime, walkparams);
    let maxtxsz : integer = AArch64_MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (IsFeatureImplemented(FEAT_LVA) ||
                ImpDefBool("Fault on TxSZ value below minimum"));
    end;
    if UInt(walkparams.txsz) > maxtxsz then
        return ImpDefBool("Fault on TxSZ value above maximum");
    end;

    return FALSE;
end;
```

J1.2.4.34 AArch64_S2CheckPermissions

```
// AArch64_S2CheckPermissions()
// ============================
// Verifies memory access with available permissions.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_S2CheckPermissions(fault_in : FaultRecord, walkstate : TTWState,
                                walkparams : S2TTWParams, ipa : AddressDescriptor,
                                accdesc : AccessDescriptor) => (FaultRecord, boolean)
begin
    let memtype : MemType = walkstate.memattrs.memtype;
    let permissions : Permissions = walkstate.permissions;
    var fault : FaultRecord = fault_in;
    let s2perms : S2AccessControls = AArch64_S2ComputePermissions(permissions, walkparams,
                                                                  accdesc);

    var r,  w : bit;
    var or, ow : bit;

    if accdesc.acctype == AccessType_TTW then
        r = s2perms.r_mmu;
        w = s2perms.w_mmu;
        or = s2perms.or_mmu;
        ow = s2perms.ow_mmu;
    elsif accdesc.rcw then
        r = s2perms.r_rcw;
        w = s2perms.w_rcw;
        or = s2perms.or_rcw;
        ow = s2perms.ow_rcw;
    else
        r = s2perms.r;
        w = s2perms.w;
        or = s2perms.or;
        ow = s2perms.ow;
    end;

    if accdesc.acctype == AccessType_TTW then
        if (accdesc.toplevel && accdesc.varange == VARange_LOWER &&
              ((walkparams.tl0 == '1' && s2perms.toplevel0 == '0') ||
              (walkparams.tl1 == '1' && s2perms.[toplevel1,toplevel0] == '10'))) then
            fault.statuscode = Fault_Permission;
            fault.toplevel   = TRUE;
        elsif (accdesc.toplevel && accdesc.varange == VARange_UPPER &&
                ((walkparams.tl1 == '1' && s2perms.toplevel1 == '0') ||
                (walkparams.tl0 == '1' && s2perms.[toplevel1,toplevel0] == '01'))) then
            fault.statuscode = Fault_Permission;
            fault.toplevel   = TRUE;
        // Stage 2 Permission fault due to AssuredOnly check
        elsif (walkstate.s2assuredonly == '1' && !ipa.s1assured) then
                fault.statuscode  = Fault_Permission;
                fault.assuredonly = TRUE;

        elsif s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif accdesc.write && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;

        elsif walkparams.ptw == '1' && memtype == MemType_Device then
            fault.statuscode = Fault_Permission;
        // Prevent translation table walks in Non-secure space by Realm state
        elsif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            fault.statuscode = Fault_Permission;
        elsif r == '0' then
            fault.statuscode = Fault_Permission;
        elsif accdesc.write && w == '0' then
            fault.statuscode = Fault_Permission;
            fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS() && permissions.dbm == '1';
        elsif (accdesc.write && walkparams.s2pie == '1' && permissions.s2dirty == '0' &&
                 (walkparams.hd != '1' ||
                 (accdesc.acctype != AccessType_AT &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                walkparams.hdbss == '1' && !CanAppendToHDBSS()))) then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
            fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS();

        end;

    // Stage 2 Permission fault due to AssuredOnly check
    elsif ((walkstate.s2assuredonly == '1' && !ipa.s1assured) ||
            (walkstate.s2assuredonly != '1' && IsFeatureImplemented(FEAT_GCS) &&
             VTCR_EL2().GCSH == '1' && accdesc.acctype == AccessType_GCS && accdesc.el != EL0)) then
        fault.statuscode  = Fault_Permission;
        fault.assuredonly = TRUE;

    elsif accdesc.acctype == AccessType_IFETCH then
        if s2perms.overlay && s2perms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (memtype == MemType_Device &&
              ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;

        // Prevent execution from Non-secure space by Realm state
        elsif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            fault.statuscode = Fault_Permission;
        elsif s2perms.x == '0' then
            fault.statuscode = Fault_Permission;
        end;

    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if !ELUsingAArch32(EL1) && s2perms.overlay && ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif !ELUsingAArch32(EL1) && w == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.hd != '1' && walkparams.s2pie == '1' &&
                      permissions.s2dirty == '0') then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
            end;
        elsif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (walkparams.cmow == '1' && accdesc.cacheop == CacheOp_CleanInvalidate &&
                s2perms.overlay && ow == '0') then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif !ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' then
            fault.statuscode = Fault_Permission;
        elsif (walkparams.cmow == '1' && accdesc.cacheop == CacheOp_CleanInvalidate &&
                w == '0') then
            fault.statuscode = Fault_Permission;
        elsif (walkparams.cmow == '1' && accdesc.cacheop == CacheOp_CleanInvalidate &&
                 walkparams.hd != '1' && walkparams.s2pie == '1' && permissions.s2dirty == '0') then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
        end;

    elsif accdesc.acctype == AccessType_IC then
        if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' &&
                ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif walkparams.cmow == '1' && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault_Permission;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            fault.overlay    = TRUE;
        elsif (!ELUsingAArch32(EL1) && accdesc.el == EL0 &&  r == '0' &&
              ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
            fault.statuscode = Fault_Permission;
        elsif walkparams.cmow == '1' && w == '0' then
            fault.statuscode = Fault_Permission;
        elsif (walkparams.cmow == '1' && walkparams.hd != '1' && walkparams.s2pie == '1' &&
                 permissions.s2dirty == '0') then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
        end;

    elsif accdesc.read && s2perms.overlay && or == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = FALSE;
    elsif accdesc.write && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = TRUE;
    elsif accdesc.read && r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
    elsif accdesc.write && w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;
        fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS() && permissions.dbm == '1';
    elsif (IsFeatureImplemented(FEAT_MTE_PERM) &&
             ((accdesc.tagchecked &&
               AArch64_EffectiveTCF(accdesc.el, accdesc.read) != TCFType_Ignore) ||
              accdesc.tagaccess) &&
             ipa.memattrs.tags == MemTag_AllocationTagged &&
             permissions.s2tag_na == '1' && S2DCacheEnabled()) then
        fault.statuscode = Fault_Permission;
        fault.tagaccess  = TRUE;
        fault.write      = accdesc.tagaccess && accdesc.write;
    elsif (accdesc.write &&
             (walkparams.hd != '1' || (walkparams.hdbss == '1' && !CanAppendToHDBSS())) &&
              walkparams.s2pie == '1' && permissions.s2dirty == '0') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;
        fault.write      = TRUE;
        fault.hdbssf     = walkparams.hdbss == '1' && !CanAppendToHDBSS();
    end;
    // MRO* allows only RCW and MMU writes
    var mro : boolean;
    if s2perms.overlay then
        mro = (s2perms.[w,w_rcw,w_mmu] AND s2perms.[ow,ow_rcw,ow_mmu]) == '011';
    else
        mro = s2perms.[w,w_rcw,w_mmu] == '011';
    end;

    return (fault, mro);
end;
```

J1.2.4.35 AArch64_S2ComputePermissions

```
// AArch64_S2ComputePermissions()
// ==============================
// Compute the overall stage 2 permissions.

func AArch64_S2ComputePermissions(permissions : Permissions, walkparams : S2TTWParams,
                                  accdesc : AccessDescriptor) => S2AccessControls
begin

    var s2perms : S2AccessControls;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if walkparams.s2pie == '1' then
        s2perms = AArch64_S2IndirectBasePermissions(permissions, accdesc);
        s2perms.overlay = IsFeatureImplemented(FEAT_S2POE) && VTCR_EL2().S2POE == '1';
        if s2perms.overlay then
            let s2overlay_perms : S2AccessControls = AArch64_S2OverlayPermissions(permissions,
                                                                                   accdesc);
            s2perms.or          = s2overlay_perms.or;
            s2perms.ow          = s2overlay_perms.ow;
            s2perms.ox          = s2overlay_perms.ox;
            s2perms.or_rcw      = s2overlay_perms.or_rcw;
            s2perms.ow_rcw      = s2overlay_perms.ow_rcw;
            s2perms.or_mmu      = s2overlay_perms.or_mmu;
            s2perms.ow_mmu      = s2overlay_perms.ow_mmu;

            // Toplevel is applicable only when the effective S2 permissions is MRO
            if ((s2perms.[w,w_rcw,w_mmu] AND s2perms.[ow,ow_rcw,ow_mmu]) == '011') then
                s2perms.toplevel0 = s2perms.toplevel0 OR s2overlay_perms.toplevel0;
                s2perms.toplevel1 = s2perms.toplevel1 OR s2overlay_perms.toplevel1;

            else
                s2perms.toplevel0 = '0';
                s2perms.toplevel1 = '0';
            end;
        end;
    else
        s2perms = AArch64_S2DirectBasePermissions(permissions, accdesc, walkparams);
    end;

    return s2perms;
end;
```

J1.2.4.36 AArch64_S2DirectBasePermissions

```
// AArch64_S2DirectBasePermissions()
// =================================
// Computes the stage 2 direct base permissions.

func AArch64_S2DirectBasePermissions(permissions : Permissions,
                                     accdesc : AccessDescriptor,
                                     walkparams : S2TTWParams) => S2AccessControls
begin
    var s2perms : S2AccessControls;
    var w : bit;
    let r : bit = permissions.s2ap[0];
    if permissions.s2ap[1] == '1' then
        w = '1';
    // Descriptors marked with DBM set have the effective value of S2AP[1] set.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    elsif permissions.dbm == '1' && walkparams.hd == '1' then
        // An update occurs here, conditional to being able to append to HDBSS
        if walkparams.hdbss == '1' then
            w = if CanAppendToHDBSS() then '1' else '0';
        else
            w = '1';
        end;
    else
        w = '0';
    end;

    var px, ux : bit;
    case (permissions.s2xn::permissions.s2xnx) of
        when '00' => (px,ux) = ('1','1');
        when '01' => (px,ux) = ('0','1');
        when '10' => (px,ux) = ('0','0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '11' => (px,ux) = ('1','0');
    end;

    let x : bit = if accdesc.el == EL0 then ux else px;
    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r;
    s2perms.w_rcw = w;
    s2perms.r_mmu = r;
    s2perms.w_mmu = w;
    s2perms.toplevel0 = '0';
    s2perms.toplevel1 = '0';
    s2perms.overlay = FALSE;

    return s2perms;
end;
```

J1.2.4.37 AArch64_S2HasAlignmentFaultDueToMemType

```
// AArch64_S2HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

func AArch64_S2HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                             memattrs : MemoryAttributes) => boolean
begin

    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && S2DCacheEnabled()) &&
                ConstrainUnpredictableBool(Unpredictable_LSE2_ALIGNMENT_FAULT)) then
            return TRUE;
        end;
    end;

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif ((accdesc.acctype == AccessType_DCZero && accdesc.cachetype == CacheType_Tag) ||
             accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE) == Constraint_FAULT;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

J1.2.4.38 AArch64_S2InconsistentSL

```
// AArch64_S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

func AArch64_S2InconsistentSL(walkparams : S2TTWParams) => boolean
begin
    let startlevel : integer      = AArch64_S2StartLevel(walkparams);
    let levels : integer          = FINAL_LEVEL - startlevel;
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsizelog2 : integer{}  = 3;
    let stride : integer{}        = granulebits - descsizelog2;

    // Input address size must at least be large enough to be resolved from the start level
    let sl_min_iasize : integer = (
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
    let sl_max_iasize : integer = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    let iasize : AddressSize    = AArch64_IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
end;
```

J1.2.4.39 AArch64_S2IndirectBasePermissions

```
// AArch64_S2IndirectBasePermissions()
// ===================================
// Computes the stage 2 indirect base permissions.

func AArch64_S2IndirectBasePermissions(permissions : Permissions,
                                       accdesc : AccessDescriptor) => S2AccessControls
begin
    var r, w : bit;
    var r_rcw, w_rcw : bit;
    var r_mmu, w_mmu : bit;
    var px, ux : bit;
    var toplevel0, toplevel1 : bit;
    var s2perms : S2AccessControls;

    let s2pi : bits(4) = permissions.s2pi;
    case s2pi of
        when '0000' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // No Access
        when '0001' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO
        when '0011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL1
        when '0100' => (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0');  // Write Only
        when '0101' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL0
        when '0111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL01
        when '1000' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0');  // Read Only
        when '1001' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0');  // Read, Unpriv Execute
        when '1010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0');  // Read, Priv Execute
        when '1011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0');  // Read, All Execute
        when '1100' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1');  // RW
        when '1101' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1');  // RW, Unpriv Execute
        when '1110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1');  // RW, Priv Execute
        when '1111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1');  // RW, All Execute
    end;

    let x : bit = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2pi of
        when '0110' => (toplevel0,toplevel1) = ('1','0');
        when '0011' => (toplevel0,toplevel1) = ('0','1');
        when '0111' => (toplevel0,toplevel1) = ('1','1');
        otherwise =>   (toplevel0,toplevel1) = ('0','0');
    end;

    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r_rcw;
    s2perms.r_mmu = r_mmu;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    s2perms.w_rcw = w_rcw;
    s2perms.w_mmu = w_mmu;
    s2perms.toplevel0 = toplevel0;
    s2perms.toplevel1 = toplevel1;

    return s2perms;
end;
```

J1.2.4.40 AArch64_S2InvalidSL

```
// AArch64_S2InvalidSL()
// =====================
// Detect invalid configuration of SL field

func AArch64_S2InvalidSL(walkparams : S2TTWParams) => boolean
begin
    case walkparams.tgx of
        when TGx_4KB =>
            case walkparams.sl2::walkparams.sl0 of
                when '1x1' => return TRUE;
                when '11x' => return TRUE;
                when '100' => return AArch64_PAMax() < 52;
                when '010' => return AArch64_PAMax() < 44;
                when '011' => return !IsFeatureImplemented(FEAT_TTST);
                otherwise  =>  return FALSE;
            end;
        when TGx_16KB =>
            case walkparams.sl0 of
                when '11' => return walkparams.ds == '0' || AArch64_PAMax() < 52;
                when '10' => return AArch64_PAMax() < 42;
                otherwise =>  return FALSE;
            end;
        when TGx_64KB =>
            case walkparams.sl0 of
                when '11' =>  return TRUE;
                when '10' =>  return AArch64_PAMax() < 44;
                otherwise =>  return FALSE;
            end;
    end;
end;
```

J1.2.4.41 AArch64_S2OAOutOfRange

```
// AArch64_S2OAOutOfRange()
// ========================
// Returns whether stage 2 output address is expressed in the configured size number of bits

func AArch64_S2OAOutOfRange(address : bits(NUM_PABITS), walkparams : S2TTWParams) => boolean
begin
    return AArch64_OAOutOfRange(address, walkparams.d128, walkparams.ds, walkparams.ps,
                                walkparams.tgx);
end;
```

J1.2.4.42 AArch64_S2OverlayPermissions

```
// AArch64_S2OverlayPermissions()
// ==============================
// Computes the stage 2 overlay permissions.

func AArch64_S2OverlayPermissions(permissions : Permissions,
                                  accdesc : AccessDescriptor) => S2AccessControls
begin
    var r, w : bit;
    var r_rcw, w_rcw : bit;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var r_mmu, w_mmu : bit;
    var px, ux : bit;
    var toplevel0, toplevel1 : bit;
    var s2overlay_perms : S2AccessControls;

    let index : integer{} = 4 * UInt(permissions.s2po_index);
    var s2po : bits(4) = '0000';
    if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
          ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
        s2po = S2POR_EL1()[index+3:index];
    end;
    case s2po of
        when '0000' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // No Access
        when '0001' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO
        when '0011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL1
        when '0100' => (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0');  // Write Only
        when '0101' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL0
        when '0111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL01
        when '1000' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0');  // Read Only
        when '1001' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0');  // Read, Unpriv Execute
        when '1010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0');  // Read, Priv Execute
        when '1011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0');  // Read, All Execute
        when '1100' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1');  // RW
        when '1101' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1');  // RW, Unpriv Execute
        when '1110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1');  // RW, Priv Execute
        when '1111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1');  // RW, All Execute
    end;

    let x : bit = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2po of
        when '0110' => (toplevel0,toplevel1) = ('1','0');
        when '0011' => (toplevel0,toplevel1) = ('0','1');
        when '0111' => (toplevel0,toplevel1) = ('1','1');
        otherwise =>   (toplevel0,toplevel1) = ('0','0');
    end;

    s2overlay_perms.or = r;
    s2overlay_perms.ow = w;
    s2overlay_perms.ox = x;
    s2overlay_perms.or_rcw = r_rcw;
    s2overlay_perms.ow_rcw = w_rcw;
    s2overlay_perms.or_mmu = r_mmu;
    s2overlay_perms.ow_mmu = w_mmu;
    s2overlay_perms.toplevel0 = toplevel0;
    s2overlay_perms.toplevel1 = toplevel1;

    return s2overlay_perms;
end;
```

J1.2.4.43 AArch64_S2TxSZFaults

```
// AArch64_S2TxSZFaults()
// ======================
// Detect whether configuration of stage 2 TxSZ field generates a fault

func AArch64_S2TxSZFaults(walkparams : S2TTWParams, s1aarch64 : boolean) => boolean
begin
    let mintxsz : integer = AArch64_S2MinTxSZ(walkparams, s1aarch64);
    let maxtxsz : integer = AArch64_MaxTxSZ(walkparams.tgx);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if UInt(walkparams.txsz) < mintxsz then
        return (IsFeatureImplemented(FEAT_LPA) ||
                ImpDefBool("Fault on TxSZ value below minimum"));
    end;
    if UInt(walkparams.txsz) > maxtxsz then
        return ImpDefBool("Fault on TxSZ value above maximum");
    end;

    return FALSE;
end;
```

J1.2.4.44 AArch64_VAIsOutOfRange

```
// AArch64_VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value

func AArch64_VAIsOutOfRange(va_in : bits(64), acctype : AccessType,
                            regime : Regime, walkparams : S1TTWParams) => boolean
begin
    var va : bits(64) = va_in;

    let addrtop : AddressSize = AArch64_AddrTop(walkparams.tbid, acctype, walkparams.tbi);

    // If the VA has a Logical Address Tag then the bits holding the Logical Address Tag are
    // ignored when checking if the address is out of range.
    if walkparams.mtx == '1' && acctype != AccessType_IFETCH then
        va[59:56] = if AArch64_GetVARange(va) == VARange_UPPER then '1111' else '0000';
    end;

    // Input Address size
    let iasize : integer{} = AArch64_IASize(walkparams.txsz);

    // The min value of TxSZ can be 8, with LVA3 implemented.
    // If TxSZ is set to 8 iasize becomes 64 - 8 = 56
    // If tbi is also set, addrtop becomes 55
    // Then the return statements check va[56:55]
    // The check here is to guard against this corner case.
    if addrtop < iasize then
        return FALSE;
    end;

    if HasUnprivileged(regime) then
        if AArch64_GetVARange(va) == VARange_LOWER then
            return !IsZero(va[addrtop:iasize]);
        else
            return !IsOnes(va[addrtop:iasize]);
        end;
    else
        return !IsZero(va[addrtop:iasize]);
    end;
end;
```

J1.2.4.45 AArch64_S2ApplyFWBMemAttrs

```
// AArch64_S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

func AArch64_S2ApplyFWBMemAttrs{N}(s1_memattrs : MemoryAttributes, walkparams : S2TTWParams,
                                   descriptor : bits(N)) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;
    let s2_attr : bits(4) = descriptor[5:2];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let s2_sh : bits(2)   = if walkparams.ds == '1' then walkparams.sh else descriptor[9:8];
let s2_fnxs : bit     = descriptor[11];

if s2_attr[2] == '0' then          // S2 Device, S1 any
    let s2_device : DeviceType = DecodeDevice(s2_attr[1:0]);
    memattrs.memtype = MemType_Device;
    if s1_memattrs.memtype == MemType_Device then
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
    else
        memattrs.device = s2_device;
    end;

    memattrs.xs = s1_memattrs.xs;

elsif s2_attr[1:0] == '11' then    // S2 attr = S1 attr
    memattrs = s1_memattrs;

elsif s2_attr[1:0] == '10' then    // Force writeback
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.outer.attrs = MemAttr_WB;

    if (s1_memattrs.memtype == MemType_Normal &&
            s1_memattrs.inner.attrs != MemAttr_NC) then
        memattrs.inner.hints     = s1_memattrs.inner.hints;
        memattrs.inner.transient = s1_memattrs.inner.transient;
    else
        memattrs.inner.hints     = MemHint_RWA;
        memattrs.inner.transient = FALSE;
    end;

    if (s1_memattrs.memtype == MemType_Normal &&
            s1_memattrs.outer.attrs != MemAttr_NC) then
        memattrs.outer.hints     = s1_memattrs.outer.hints;
        memattrs.outer.transient = s1_memattrs.outer.transient;
    else
        memattrs.outer.hints     = MemHint_RWA;
        memattrs.outer.transient = FALSE;
    end;

    memattrs.xs = '0';

else                               // Non-cacheable unless S1 is device
    if s1_memattrs.memtype == MemType_Device then
        memattrs = s1_memattrs;
    else
        var cacheability_attr : MemAttrHints;
        cacheability_attr.attrs = MemAttr_NC;

        memattrs.memtype = MemType_Normal;
        memattrs.inner   = cacheability_attr;
        memattrs.outer   = cacheability_attr;

        memattrs.xs = s1_memattrs.xs;
    end;
end;

let s2_shareability : Shareability = DecodeShareability(s2_sh);
memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
memattrs.tags         = S2MemTagType(memattrs, s1_memattrs.tags);
memattrs.notagaccess  = (s2_attr[3:1] == '111' && memattrs.tags == MemTag_AllocationTagged);

if s2_fnxs == '1' then
    memattrs.xs = '0';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
end;
```

J1.2.4.46 AArch64_GetS1TLBContext

```
// AArch64_GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

func AArch64_GetS1TLBContext(regime : Regime, ss : SecurityState,
                             va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    case regime of
        when Regime_EL3 =>  tlbcontext = AArch64_TLBContextEL3(ss, va, tg);
        when Regime_EL2 =>  tlbcontext = AArch64_TLBContextEL2(ss, va, tg);
        when Regime_EL20 => tlbcontext = AArch64_TLBContextEL20(ss, va, tg);
        when Regime_EL10 => tlbcontext = AArch64_TLBContextEL10(ss, va, tg);
        otherwise =>
            unreachable;

    end;

    tlbcontext.includes_s1  = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2  = FALSE;
    tlbcontext.use_vmid     = UseVMID(regime);
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
end;
```

J1.2.4.47 AArch64_GetS2TLBContext

```
// AArch64_GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

func AArch64_GetS2TLBContext(ss : SecurityState, ipa : FullAddress, tg : TGx) => TLBContext
begin
    assert EL2Enabled();

    var tlbcontext : TLBContext;

    tlbcontext.ss       = ss;
    tlbcontext.regime   = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid     = VMID();
    tlbcontext.tg       = tg;
    tlbcontext.ia       = ZeroExtend{64}(ipa.address);
    if IsFeatureImplemented(FEAT_TTCNP) then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2().CnP else VTTBR_EL2().CnP;
    else
        tlbcontext.cnp = '0';
    end;


    tlbcontext.includes_s1  = FALSE;
    tlbcontext.includes_s2  = TRUE;
    tlbcontext.use_vmid     = TRUE;
    // This may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.4.48 AArch64_TLBContextEL10

```
// AArch64_TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

func AArch64_TLBContextEL10(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID();

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() && TCR2_EL1().A2 == '1' then
        let varange : VARange = AArch64_GetVARange(va);
        tlbcontext.asid = if varange == VARange_LOWER then TTBR0_EL1().ASID else TTBR1_EL1().ASID;
    else
        tlbcontext.asid = if TCR_EL1().A1 == '0' then TTBR0_EL1().ASID else TTBR1_EL1().ASID;
    end;

    if TCR_EL1().AS == '0' then
        tlbcontext.asid[15:8] = Zeros{8};
    end;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64_GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1().CnP;
        else
            tlbcontext.cnp = TTBR1_EL1().CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```

J1.2.4.49 AArch64_TLBContextEL2

```
// AArch64_TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

func AArch64_TLBContextEL2(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_EL2().CnP else '0';

    return tlbcontext;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.50 AArch64_TLBContextEL20

```
// AArch64_TLBContextEL20()
// ========================
// Gather translation context for accesses under EL20 regime to match against TLB entries

func AArch64_TLBContextEL20(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL20;

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() && TCR2_EL2().A2 == '1' then
        let varange : VARange = AArch64_GetVARange(va);
        tlbcontext.asid = if varange == VARange_LOWER then TTBR0_EL2().ASID else TTBR1_EL2().ASID;
    else
        tlbcontext.asid = if TCR_EL2().A1 == '0' then TTBR0_EL2().ASID else TTBR1_EL2().ASID;
    end;

    if TCR_EL2().AS == '0' then
        tlbcontext.asid[15:8] = Zeros{8};
    end;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64_GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL2().CnP;
        else
            tlbcontext.cnp = TTBR1_EL2().CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```

J1.2.4.51 AArch64_TLBContextEL3

```
// AArch64_TLBContextEL3()
// =======================
// Gather translation context for accesses under EL3 regime to match against TLB entries

func AArch64_TLBContextEL3(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL3;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_EL3().CnP else '0';

    return tlbcontext;
end;
```

J1.2.4.52 AArch64_FullTranslate

```
// AArch64_FullTranslate()
// =======================
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_FullTranslate(va : bits(64), size : integer, accdesc : AccessDescriptor,
                           aligned : boolean) => AddressDescriptor
begin
    let regime : Regime = TranslationRegime(accdesc.el);
    var fault : FaultRecord = NoFault(accdesc, va);

    var ipa : AddressDescriptor;
    (fault, ipa) = AArch64_S1Translate(fault, regime, va, size, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(fault);
    end;

    if accdesc.ss == SS_Realm then
        assert EL2Enabled();
    end;
    if regime == Regime_EL10 && EL2Enabled() then
        let s1aarch64 : boolean = TRUE;
        var pa : AddressDescriptor;
        (fault, pa) = AArch64_S2Translate(fault, ipa, s1aarch64, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(fault);
        else
            return pa;
        end;
    else
        return ipa;
    end;
end;
```

J1.2.4.53 AArch64_MemSwapTableDesc

```
// AArch64_MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

func AArch64_MemSwapTableDesc{N : integer{64, 128}}(fault_in : FaultRecord, prev_desc : bits(N),
                                                    new_desc : bits(N), ee : bit,
                                                    descaccess : AccessDescriptor,
                                                    descpaddr : AddressDescriptor
                                                 ) => (FaultRecord, bits(N))
begin
    assert descaccess.acctype == AccessType_TTW;
    var fault : FaultRecord = fault_in;
    var iswrite : boolean;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(descpaddr, descaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = descpaddr.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    var mem_desc : bits(N);
    var memstatus : PhysMemRetStatus;
    (memstatus, mem_desc) = PhysMemRead{N}(descpaddr, descaccess);

    if ee == '1' then
        mem_desc = BigEndianReverse{N}(mem_desc);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    if mem_desc == prev_desc  then
        let ordered_new_desc : bits(N) = (if ee == '1' then BigEndianReverse{N}(new_desc)
                                                        else new_desc);
        memstatus = PhysMemWrite{N}(descpaddr, descaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8,
                                           fault);

            if IsFault(fault.statuscode) then
                return (fault, ARBITRARY : bits(N));
            end;
        end;

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;
    end;

    return (fault, mem_desc);
end;
```

J1.2.4.54 AArch64_S1DisabledOutput

```
// AArch64_S1DisabledOutput()
// ==========================
// Map the VA to IPA/PA and assign default memory attributes

func AArch64_S1DisabledOutput(fault_in : FaultRecord, regime : Regime,
                              va_in :  bits(64), accdesc : AccessDescriptor,
                              aligned : boolean) => (FaultRecord, AddressDescriptor)
begin
    var va : bits(64) = va_in;
    let walkparams : S1TTWParams = AArch64_GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    var fault : FaultRecord = fault_in;

    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    // Output Address
    var oa : FullAddress;
    oa.address = va[NUM_PABITS-1:0];
    case accdesc.ss of
        when SS_Secure =>    oa.paspace = PAS_Secure;
        when SS_NonSecure => oa.paspace = PAS_NonSecure;
        when SS_Root =>      oa.paspace = PAS_Root;
        when SS_Realm =>     oa.paspace = PAS_Realm;
    end;

    var memattrs : MemoryAttributes;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
        var default_cacheability : MemAttrHints;
        default_cacheability.attrs     = MemAttr_WB;
        default_cacheability.hints     = MemHint_RWA;
        default_cacheability.transient = FALSE;

        memattrs.memtype      = MemType_Normal;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        mematrs.outer        = default_cacheability;
        mematrs.inner        = default_cacheability;
        mematrs.shareability = Shareability_NSH;
        if walkparams.dct == '1' then
            mematrs.tags     = MemTag_AllocationTagged;
        elsif IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && walkparams.mtx == '1' then
            mematrs.tags     = MemTag_CanonicallyTagged;
        else
            mematrs.tags     = MemTag_Untagged;
        end;
        mematrs.xs           = '0';
    elsif accdesc.acctype == AccessType_IFETCH then
        var i_cache_attr : MemAttrHints;
        if AArch64_S1ICacheEnabled(regime) then
            i_cache_attr.attrs     = MemAttr_WT;
            i_cache_attr.hints     = MemHint_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs     = MemAttr_NC;
        end;

        mematrs.memtype      = MemType_Normal;
        mematrs.outer        = i_cache_attr;
        mematrs.inner        = i_cache_attr;
        mematrs.shareability = Shareability_OSH;
        mematrs.tags         = MemTag_Untagged;
        mematrs.xs           = '1';
    elsif accdesc.acctype == AccessType_SPE && EffectivePMBLIMITR_EL1_nVM() == '1' then
        mematrs = S1DecodeMemAttrs(PMBMAR_EL1().Attr, PMBMAR_EL1().SH, TRUE,
                                   walkparams, accdesc.acctype);
    elsif accdesc.acctype == AccessType_TRBE && EffectiveTRBLIMITR_EL1_nVM() == '1' then
        mematrs = S1DecodeMemAttrs(TRBMAR_EL1().Attr, TRBMAR_EL1().SH, TRUE,
                                   walkparams, accdesc.acctype);
    else
        mematrs.memtype      = MemType_Device;
        mematrs.device       = DeviceType_nGnRnE;
        mematrs.shareability = Shareability_OSH;
        if IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && walkparams.mtx == '1' then
            mematrs.tags = MemTag_CanonicallyTagged;
        else
            mematrs.tags = MemTag_Untagged;
        end;
        mematrs.xs           = '1';
    end;
    mematrs.notagaccess = FALSE;

    if walkparams.mtx == '1' && walkparams.tbi == '0' && accdesc.acctype != AccessType_IFETCH then
        // For the purpose of the checks in this function, the MTE tag bits are ignored.
        va[59:56] = if HasUnprivileged(regime) then Replicate{4}(va[55]) else '0000';
    end;

    fault.level = 0;
    let addrtop : AddressSize = AArch64_AddrTop(walkparams.tbid, accdesc.acctype,
                                                walkparams.tbi);
    let pamax : AddressSize = AArch64_PAMax();

    if !IsZero(va[addrtop:pamax]) then
        fault.statuscode = Fault_AddressSize;
    elsif AArch64_S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmd,
                                                  mematrs) then
        fault.statuscode = Fault_Alignment;
    elsif ((accdesc.exclusive || accdesc.atomicop) &&
            !(regime == Regime_EL10 && EL2Enabled() && HCR_EL2().DC == '1') &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor);
    else
        var ipa : AddressDescriptor = CreateAddressDescriptor(va_in, oa, memattrs, accdesc);
        ipa.mecid = AArch64_S1DisabledOutputMECID(walkparams, regime, ipa.paddress.paspace);
        return (fault, ipa);
    end;
end;
```

J1.2.4.55 AArch64_S1Translate

```
// AArch64_S1Translate()
// =====================
// Translate VA to IPA/PA depending on the regime

func AArch64_S1Translate(fault_in : FaultRecord, regime : Regime,
                         va : bits(64), size : integer, aligned : boolean,
                         accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
                         recurselimit Unbounded_DescriptorUpdate
begin
    var fault : FaultRecord = fault_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch64_S1Enabled(regime, accdesc.acctype) then
        return AArch64_S1DisabledOutput(fault, regime, va, accdesc, aligned);
    end;

    var walkparams : S1TTWParams = AArch64_GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    let s1mintxsz : integer = AArch64_S1MinTxSZ(regime, walkparams);
    let s1maxtxsz : integer = AArch64_MaxTxSZ(walkparams.tgx);
    if AArch64_S1TxSZFaults(regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    elsif UInt(walkparams.txsz) < s1mintxsz then
        walkparams.txsz = s1mintxsz[5:0];
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        walkparams.txsz = s1maxtxsz[5:0];
    end;

    if AArch64_VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if accdesc.el == EL0 && walkparams.e0pd == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if (IsFeatureImplemented(FEAT_SVE) && accdesc.el == EL0 && walkparams.nfd == '1' &&
          ((accdesc.nonfault && accdesc.contiguous) ||
           (accdesc.firstfault && !accdesc.first && !accdesc.contiguous))) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var descipaddr : AddressDescriptor;
    var walkstate : TTWState;
    var descriptor : bits(128);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if walkparams.d128 == '1' then
    (fault, descipaddr, walkstate, descriptor) = AArch64_S1Walk{128}(fault, walkparams, va,
                                                                     regime, accdesc);
else
    (fault, descipaddr, walkstate, descriptor[63:0]) = AArch64_S1Walk{64}(fault, walkparams, va,
                                                                          regime, accdesc);
    descriptor[127:64] = Zeros{64};
end;
if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor);
end;
if AArch64_S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmd,
                                           walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
end;

let fault_perm : FaultRecord = AArch64_S1CheckPermissions(fault, va, size, regime,
                                                          walkstate, walkparams, accdesc);

var mem_desc : bits(128);
var new_desc : bits(128) = descriptor;

if AArch64_SetAccessFlag(walkparams.ha, accdesc, fault, fault_perm) then
    // Set descriptor AF bit
    new_desc[10] = '1';
end;

// If HW update of dirty bit is enabled, the walk state permissions
// will already reflect a configuration permitting writes.
// The update of the descriptor occurs only if the descriptor bits in
// memory do not reflect that and the access instigates a write.

if AArch64_SetDirtyState(walkparams.hd, (walkparams.pie OR descriptor[51]),
                         accdesc, fault, fault_perm) then
    // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes
    new_desc[7] = '0';
end;

if fault.statuscode == Fault_None && fault_perm.statuscode != Fault_None then
    fault = fault_perm;
end;

// Either the access flag was clear or AP[2]/nDirty was set
if new_desc != descriptor then
    if (!IsWBShareable(descipaddr.memattrs) &&
          ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
        fault.statuscode = Fault_UnsupportedAtomicHWUpdate;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var descpaddr : AddressDescriptor;
    let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);

    if regime == Regime_EL10 && EL2Enabled() then
        var s2fault : FaultRecord;
        let s1aarch64 : boolean = TRUE;
        let s2aligned : boolean = TRUE;
        (s2fault, descpaddr) = AArch64_S2Translate(fault, descipaddr, s1aarch64, s2aligned,
                                                   descaccess);
        if s2fault.statuscode != Fault_None then
            return (s2fault, ARBITRARY : AddressDescriptor);
        end;

    else
        descpaddr = descipaddr;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if walkparams.d128 == '1' then
            (fault, mem_desc) = AArch64_MemSwapTableDesc{128}(fault, descriptor,
                                                              new_desc, walkparams.ee,
                                                              descaccess, descpaddr);
        else
            (fault, mem_desc[63:0]) = AArch64_MemSwapTableDesc{64}(fault, descriptor[63:0],
                                                                   new_desc[63:0], walkparams.ee,
                                                                   descaccess, descpaddr);
            mem_desc[127:64] = Zeros{64};
        end;

        if fault.statuscode != Fault_None then
            if (accdesc.acctype == AccessType_AT &&
                    !(ImpDefBool("AT reports the HW update fault"))) then
                // Mask the fault
                fault.statuscode = Fault_None;
            else
                return (fault, ARBITRARY : AddressDescriptor);
            end;
        elsif new_desc != descriptor && mem_desc != new_desc then
            // HW update of Dirty state or AF was not successful due to the descriptor being updated
            // not matching the descriptor used for translation. Due to this, the walk is restarted.
            return AArch64_S1Translate(fault_in, regime, va, size, aligned, accdesc);
        end;
    end;

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    // Output Address
    let oa : FullAddress = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
    var memattrs : MemoryAttributes;
    if AArch64_S1TreatAsNormalNC(walkstate, regime, accdesc) then
       // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;

        // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE on the Tagged attribute
        // when the memory region is Allocation Tagged.
        if (IsFeatureImplemented(FEAT_MTE2) &&
              walkstate.memattrs.tags == MemTag_AllocationTagged &&
              ConstrainUnpredictableBool(Unpredictable_S1CTAGGED)) then
            memattrs.tags = MemTag_AllocationTagged;
        // SCTLR_ELx.C has no effect on whether the memory region is treated as Canonically Tagged.
        elsif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
                 walkstate.memattrs.tags == MemTag_CanonicallyTagged) then
            memattrs.tags = MemTag_CanonicallyTagged;
        end;
    else
        memattrs = walkstate.memattrs;
    end;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2().VM == '1' &&
            !(ImpDefBool("Apply effective shareability at stage 1"))) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);
    end;

    var ipa : AddressDescriptor = CreateAddressDescriptor(va, oa, memattrs, accdesc);
    ipa.s1assured = walkstate.s1assured;
    let varange : VARange = AArch64_GetVARange(va);
    ipa.mecid = AArch64_S1OutputMECID{128}(walkparams, regime, varange, ipa.paddress.paspace,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                        descriptor);

    if (accdesc.atomicop && !IsWBShareable(memattrs) &&
          ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
        return (fault, ipa);
    end;

    if accdesc.ls64 && memattrs.memtype == MemType_Normal then
        if IsFeatureImplemented(FEAT_LS64WB) && !accdesc.withstatus then
            if (!IsWBShareable(memattrs) &&
                  !(memattrs.inner.attrs == MemAttr_NC &&
                    memattrs.outer.attrs == MemAttr_NC) &&
                  (ImpDefBool(
                     "LD64B or ST64B faults to cacheable non-iWBoWB memory"))) then
                fault.statuscode = Fault_Exclusive;
                return (fault, ipa);
            end;
        elsif !(memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC) then
            fault.statuscode = Fault_Exclusive;
            return (fault, ipa);
        end;
    end;

    return (fault, ipa);
end;
```

J1.2.4.56 AArch64_S1TreatAsNormalNC

```
// AArch64_S1TreatAsNormalNC()
// ===========================
// Returns TRUE if stage 1 memory attributes should be treated as Normal Non-Cacheable

func AArch64_S1TreatAsNormalNC(walkstate : TTWState, regime : Regime,
                               accdesc : AccessDescriptor) => boolean
begin
    return ((accdesc.acctype == AccessType_IFETCH &&
              (walkstate.memattrs.memtype == MemType_Device || !AArch64_S1ICacheEnabled(regime))) ||
            (accdesc.acctype != AccessType_IFETCH &&
              !AArch64_S1DCacheEnabled(regime) && walkstate.memattrs.memtype == MemType_Normal));
end;
```

J1.2.4.57 AArch64_S2Translate

```
// AArch64_S2Translate()
// =====================
// Translate stage 1 IPA to PA and combine memory attributes

func AArch64_S2Translate(fault_in : FaultRecord, ipa : AddressDescriptor,
                         s1aarch64 : boolean, aligned : boolean,
                         accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
                         recurselimit Unbounded_DescriptorUpdate
begin
    var walkparams : S2TTWParams = AArch64_GetS2TTWParams(accdesc.ss, ipa.paddress.paspace,
                                                          s1aarch64);
    var fault : FaultRecord = fault_in;
    var s2fs1mro : boolean;
    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault_None; // Ignore any faults from stage 1
    fault.dirtybit     = FALSE;
    fault.overlay      = FALSE;
    fault.tagaccess    = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.secondstage  = TRUE;
    fault.s2fs1walk    = accdesc.acctype == AccessType_TTW;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
fault.ipaddress    = ipa.paddress;

if walkparams.vm != '1' then
    // Stage 2 translation is disabled
    return (fault, ipa);
end;

let s2mintxsz : integer = AArch64_S2MinTxSZ(walkparams, s1aarch64);
let s2maxtxsz : integer = AArch64_MaxTxSZ(walkparams.tgx);
if AArch64_S2TxSZFaults(walkparams, s1aarch64) then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, ARBITRARY : AddressDescriptor);
elsif UInt(walkparams.txsz) < s2mintxsz then
    walkparams.txsz = s2mintxsz[5:0];
elsif UInt(walkparams.txsz) > s2maxtxsz then
    walkparams.txsz = s2maxtxsz[5:0];
end;

if (walkparams.d128 == '0' &&
    (AArch64_S2InvalidSL(walkparams) || AArch64_S2InconsistentSL(walkparams))) then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, ARBITRARY : AddressDescriptor);
end;

if AArch64_IPAIsOutOfRange(ipa.paddress.address, walkparams) then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, ARBITRARY : AddressDescriptor);
end;

var descpaddr : AddressDescriptor;
var walkstate : TTWState;
var descriptor : bits(128);
if walkparams.d128 == '1' then
    (fault, descpaddr, walkstate, descriptor) = AArch64_S2Walk{128}(fault, ipa, walkparams,
                                                                    accdesc);
else
    (fault, descpaddr, walkstate, descriptor[63:0]) = AArch64_S2Walk{64}(fault, ipa, walkparams,
                                                                         accdesc);
    descriptor[127:64] = Zeros{64};
end;
if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor);
end;

if AArch64_S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
end;

var fault_perm : FaultRecord;
(fault_perm, s2fs1mro) = AArch64_S2CheckPermissions(fault, walkstate, walkparams, ipa, accdesc);

var mem_desc : bits(128);
var new_desc : bits(128) = descriptor;

if AArch64_SetAccessFlag(walkparams.ha, accdesc, fault, fault_perm) then
    // Set descriptor AF bit
    new_desc[10] = '1';
end;

// If HW update of dirty bit is enabled, the walk state permissions
// will already reflect a configuration permitting writes.
// The update of the descriptor occurs only if the descriptor bits in
// memory do not reflect that and the access instigates a write.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if AArch64_SetDirtyState(walkparams.hd, (walkparams.s2pie OR descriptor[51]),
                         accdesc, fault, fault_perm) then
    // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
    new_desc[7] = '1';
end;

if fault.statuscode == Fault_None && fault_perm.statuscode != Fault_None then
    fault = fault_perm;
end;

// Either the access flag was clear or S2AP[1]/Dirty was clear
if new_desc != descriptor then
    if (!IsWBShareable(descpaddr.memattrs) &&
          ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
        fault.statuscode = Fault_UnsupportedAtomicHWUpdate;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if walkparams.hdbss == '1' && descriptor[7] == '0' && new_desc[7] == '1' then
        fault = AppendToHDBSS(fault, ipa.paddress, accdesc, walkparams, walkstate.level);
    end;

    // If an error, other than a synchronous External abort, occurred on the HDBSS update,
    // stage 2 hardware update of dirty state is not permitted.
    if (!(walkparams.hdbss == '1' && descriptor[7] != new_desc[7]) ||
          (HDBSSPROD_EL2().FSC != '101000' &&
          (!fault.hdbssf || IsExternalAbort(fault.statuscode)))) then
        let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
        if walkparams.d128 == '1' then
            (fault, mem_desc) = AArch64_MemSwapTableDesc{128}(fault, descriptor, new_desc,
                                                              walkparams.ee, descaccess,
                                                              descpaddr);
        else
            (fault, mem_desc[63:0]) = AArch64_MemSwapTableDesc{64}(fault, descriptor[63:0],
                                                                   new_desc[63:0], walkparams.ee,
                                                                   descaccess, descpaddr);
            mem_desc[127:64] = Zeros{64};
        end;
    end;

    if fault.statuscode != Fault_None then
        if (accdesc.acctype == AccessType_AT &&
                !(ImpDefBool("AT reports the HW update fault"))) then
            // Mask the fault
            fault.statuscode = Fault_None;
        else
            return (fault, ARBITRARY : AddressDescriptor);
        end;
    elsif new_desc != descriptor && mem_desc != new_desc then
        // HW update of Dirty state or AF was not successful due to the descriptor being updated
        // not matching the descriptor used for translation. Due to this, the walk is restarted.
        return AArch64_S2Translate(fault_in, ipa, s1aarch64, aligned, accdesc);
    end;
end;

if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor);
end;

let ipa_64 : bits(64) = ZeroExtend{}(ipa.paddress.address);
// Output Address
let oa : FullAddress = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
var s2_memattrs : MemoryAttributes;

if AArch64_S2TreatAsNormalNC(walkstate, walkparams, accdesc) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
        if walkstate.memattrs.tags == MemTag_CanonicallyTagged then
            s2_memattrs.tags = MemTag_CanonicallyTagged;
        end;
    else
        s2_memattrs = walkstate.memattrs;
    end;

    let s2aarch64 : boolean = TRUE;
    var memattrs : MemoryAttributes;
    if walkparams.fwb == '0' then
        memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    else
        memattrs = s2_memattrs;
    end;

    var pa : AddressDescriptor = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);
    pa.s2fs1mro = s2fs1mro;
    pa.mecid = AArch64_S2OutputMECID{128}(walkparams, pa.paddress.paspace, descriptor);

    if (accdesc.acctype == AccessType_TTW && accdesc.atomicop && !IsWBShareable(pa.memattrs) &&
          ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
        fault.statuscode = Fault_UnsupportedAtomicHWUpdate;
        return (fault, pa);
    end;

    if (accdesc.acctype != AccessType_TTW && accdesc.atomicop && !IsWBShareable(pa.memattrs) &&
          ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
        return (fault, pa);
    end;

    if accdesc.ls64 && s2_memattrs.memtype == MemType_Normal then
        if IsFeatureImplemented(FEAT_LS64WB) && !accdesc.withstatus then
            if (!IsWBShareable(s2_memattrs) &&
                  !(s2_memattrs.inner.attrs == MemAttr_NC &&
                    s2_memattrs.outer.attrs == MemAttr_NC) &&
                  (ImpDefBool(
                     "LD64B or ST64B faults to cacheable non-iWBoWB memory"))) then
                fault.statuscode = Fault_Exclusive;
                return (fault, pa);
            end;
        elsif !(s2_memattrs.inner.attrs == MemAttr_NC && s2_memattrs.outer.attrs == MemAttr_NC) then
            fault.statuscode = Fault_Exclusive;
            return (fault, pa);
        end;
    end;

    return (fault, pa);
end;
```

J1.2.4.58 AArch64_S2TreatAsNormalNC

```
// AArch64_S2TreatAsNormalNC()
// ==========================
// Returns TRUE if stage 2 memory attributes should be treated as Normal Non-cacheable

func AArch64_S2TreatAsNormalNC(walkstate : TTWState, walkparams : S2TTWParams,
                               accdesc : AccessDescriptor) => boolean
begin
    return ((accdesc.acctype == AccessType_TTW &&
                walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
            (accdesc.acctype == AccessType_IFETCH &&
                (walkstate.memattrs.memtype == MemType_Device || HCR_EL2().ID == '1')) ||
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (accdesc.acctype != AccessType_IFETCH &&
               walkstate.memattrs.memtype == MemType_Normal && !S2DCacheEnabled()));
end;
```

J1.2.4.59 AArch64_SetAccessFlag

```
// AArch64_SetAccessFlag()
// =======================
// Determine whether the access flag could be set by HW given the fault status

func AArch64_SetAccessFlag(ha : bit, accdesc : AccessDescriptor, fault : FaultRecord,
                           fault_perm : FaultRecord) => boolean
begin
    if ha == '0' || !AArch64_SettingAccessFlagPermitted(fault, fault_perm) then
        return FALSE;
    elsif accdesc.acctype == AccessType_AT then
        return ImpDefBool("AT updates AF");
    elsif accdesc.acctype IN {AccessType_DC, AccessType_IC} then
        return ImpDefBool("Generate access flag fault on IC/DC operations");
    else
        // Set descriptor AF bit
        return TRUE;
    end;
end;
```

J1.2.4.60 AArch64_SetDirtyState

```
// AArch64_SetDirtyState()
// =======================
// Determine whether dirty state is required to be updated by HW given the fault status

func AArch64_SetDirtyState(hd : bits(1), dbm : bits(1),
                           accdesc : AccessDescriptor,
                           fault : FaultRecord,
                           fault_perm : FaultRecord) => boolean
begin
    if hd == '0' then
        return FALSE;
    elsif !AArch64_SettingDirtyStatePermitted(fault, fault_perm) then
        return FALSE;
    elsif accdesc.acctype IN {AccessType_AT, AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif !accdesc.write then
        return FALSE;
    else
        return dbm == '1';
    end;
end;
```

J1.2.4.61 AArch64_SettingAccessFlagPermitted

```
// AArch64_SettingAccessFlagPermitted()
// ====================================
// Determine whether the access flag could be set by HW given the fault status

func AArch64_SettingAccessFlagPermitted(fault : FaultRecord, fault_perm : FaultRecord) => boolean
begin
    if fault.statuscode == Fault_None && fault_perm.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode == Fault_Alignment || fault_perm.statuscode == Fault_Permission then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.62 AArch64_SettingDirtyStatePermitted

```
// AArch64_SettingDirtyStatePermitted()
// ====================================
// Determine whether the dirty state could be set by HW given the fault status

func AArch64_SettingDirtyStatePermitted(fault : FaultRecord, fault_perm : FaultRecord) => boolean
begin
    if fault_perm.statuscode != Fault_None then
        return FALSE;
    elsif fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode == Fault_Alignment then
        return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);
    else
        return FALSE;
    end;
end;
```

J1.2.4.63 AArch64_TranslateAddress

```
// AArch64_TranslateAddress()
// ==========================
// Main entry point for translating an address

func AArch64_TranslateAddress(va : bits(64), accdesc : AccessDescriptor,
                              aligned : boolean, size : integer) => AddressDescriptor
begin
    if (SPESampleInFlight &&
        !(accdesc.acctype IN {AccessType_IFETCH, AccessType_SPE, AccessType_TRBE})) then
        SPEStartCounter(SPECounterPosTranslationLatency);
    end;

    var result : AddressDescriptor = AArch64_FullTranslate(va, size, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch64_CheckDebug(va, accdesc, size);
    end;

    if (IsFeatureImplemented(FEAT_RME) && !IsFault(result) &&
          IsGranuleProtectionCheckedAccess(accdesc)) then
        result.fault.gpcf = GranuleProtectionCheck(result, accdesc);

        if result.fault.gpcf.gpf != GPCF_None then
            result.fault.statuscode = Fault_GPCFOnOutput;
            result.fault.paddress   = result.paddress;
            result.fault.vaddress   = result.vaddress;
        end;
    end;
    if SPESampleInFlight &&
        !(accdesc.acctype IN {AccessType_IFETCH, AccessType_TRBE, AccessType_SPE}) then
        SPEStopCounter(SPECounterPosTranslationLatency);
    end;

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend{64}(va);

    return result;
end;
```

J1.2.4.64 AArch64_TranslateTagAddress

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64_TranslateTagAddress()
// ============================
// Translate an address for accessing an Allocation Tag.

func AArch64_TranslateTagAddress(va : bits(64),
                                 accdesc_in : AccessDescriptor,
                                 aligned : boolean,
                                 size : integer) => (MemTagType, AddressDescriptor)
begin
    var accdesc : AccessDescriptor = accdesc_in;
    let taddrdesc : AddressDescriptor = AArch64_TranslateAddress(va, accdesc, aligned, size);

    return (taddrdesc.memattrs.tags, taddrdesc);
end;
```

J1.2.4.65 AArch64_BlockDescSupported

```
// AArch64_BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

func AArch64_BlockDescSupported(d128 : bit, ds : bit, tgx : TGx, level : integer) => boolean
begin
    case tgx of
        when TGx_4KB => return ((level == 0 && (ds == '1' || d128 == '1')) ||
                                level == 1 ||
                                level == 2);
        when TGx_16KB => return ((level == 1 && (ds == '1' || d128 == '1')) ||
                                 level == 2);
        when TGx_64KB => return ((level == 1 && (d128 == '1' || AArch64_PAMax() >= 52)) ||
                                 level == 2);
    end;
    return FALSE;
end;
```

J1.2.4.66 AArch64_ContiguousBit

```
// AArch64_ContiguousBit()
// =======================
// Get the value of the contiguous bit

func AArch64_ContiguousBit{N}(tgx : TGx, d128 : bit, level : integer, descriptor : bits(N)) => bit
begin
    if ImpDefBool("Treat Contiguous bit as 0") then
        return '0';
    end;

    if d128 == '1' then
        if (tgx == TGx_64KB && level == 1) || (tgx == TGx_4KB && level == 0) then
            return '0'; // RES0
        else
            return descriptor[111];
        end;
    end;
    // When using TGx 64KB then the Contiguous bit is RES0 for Block
    // descriptors at level 1. For VMSAv8-64, level 1 Block descriptors
    // can exist only if FEAT_LPA is implemented.

    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0
    end;

    // When the Effective value of TCR_ELx.DS is '1',
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // the Contiguous bit is RES0 for all the following:
    //      * For TGx 4KB, Block descriptors at level 0
    //      * For TGx 16KB, Block descriptors at level 1
    // For VMSAv8-64, the above Block descriptors at the specified
    // levels can exist only if the Effective value of TCR_ELx.DS is 1.

    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0
    end;

    if tgx == TGx_4KB  && level == 0 then
        return '0'; // RES0
    end;

    return descriptor[52];
end;
```

J1.2.4.67 AArch64_DecodeDescriptorType

```
// AArch64_DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

func AArch64_DecodeDescriptorType{N}(descriptor : bits(N), d128 : bit, ds : bit,
                                     tgx : TGx, level : integer) => DescriptorType
begin
    if descriptor[0] == '0' then
        return DescriptorType_Invalid;
    elsif d128 == '1' then
        let skl : bits(2) = descriptor[110:109];
        if tgx IN {TGx_16KB, TGx_64KB} && UInt(skl) == 3 then
            return DescriptorType_Invalid;
        end;

        let effective_level : integer = level + UInt(skl);
        if effective_level > FINAL_LEVEL then
            return DescriptorType_Invalid;
        elsif effective_level == FINAL_LEVEL then
            return DescriptorType_Leaf;
        else
            return DescriptorType_Table;
        end;
    else
        if descriptor[1] == '1' then
            if level == FINAL_LEVEL then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Table;
            end;
        elsif descriptor[1] == '0' then
            if AArch64_BlockDescSupported(d128, ds, tgx, level) then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Invalid;
            end;
        end;
        unreachable;
    end;
end;
```

J1.2.4.68 AArch64_S1ApplyOutputPerms

```
// AArch64_S1ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 1 page/block descriptors
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_S1ApplyOutputPerms{N}(permissions_in : Permissions, descriptor : bits(N),
                                   regime : Regime, walkparams : S1TTWParams) => Permissions
begin
    var permissions : Permissions = permissions_in;

    var pi_index : bits (4);
    if walkparams.pie == '1' then
        if walkparams.d128 == '1' then
            pi_index = descriptor[118:115];
        else
            pi_index = descriptor[54:53]::descriptor[51]::descriptor[6];
        end;
        permissions.ppi    = walkparams.pir[UInt(pi_index)*:4];
        permissions.upi    = walkparams.pire0[UInt(pi_index)*:4];
        permissions.ndirty = descriptor[7];
    else
        if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
            permissions.ap[2:1] = descriptor[7]::'0';
            permissions.pxn     = descriptor[54];
        elsif HasUnprivileged(regime) then
            permissions.ap[2:1] = descriptor[7:6];
            permissions.uxn     = descriptor[54];
            permissions.pxn     = descriptor[53];
        else
            permissions.ap[2:1] = descriptor[7]::'1';
            permissions.xn      = descriptor[54];
        end;
        permissions.dbm = descriptor[51];
    end;
    if IsFeatureImplemented(FEAT_S1POE) then
        if walkparams.d128 == '1' then
            permissions.po_index[3:0] = descriptor[124:121];
        else
            permissions.po_index[3:0] = '0'::descriptor[62:60];
        end;
    end;

    return permissions;
end;
```

J1.2.4.69 AArch64_S1ApplyTablePerms

```
// AArch64_S1ApplyTablePerms()
// ===========================
// Apply hierarchical permissions encoded in stage 1 table descriptors

func AArch64_S1ApplyTablePerms(permissions_in : Permissions, descriptor : bits(64),
                               regime : Regime, walkparams : S1TTWParams) => Permissions
begin
    var permissions : Permissions = permissions_in;
    var ap_table : bits(2);
    var pxn_table : bit;
    var uxn_table : bit;
    var xn_table : bit;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table  = descriptor[62]::'0';
        pxn_table = descriptor[60];
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elsif HasUnprivileged(regime) then
        ap_table  = descriptor[62:61];
        uxn_table = descriptor[60];
        pxn_table = descriptor[59];
        permissions.ap_table  = permissions.ap_table  OR ap_table;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor[62]::'0';
        xn_table = descriptor[60];
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;
    end;

    return permissions;
end;
```

J1.2.4.70 AArch64_S2ApplyOutputPerms

```
// AArch64_S2ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 2 page/block descriptors

func AArch64_S2ApplyOutputPerms{N}(descriptor : bits(N), walkparams : S2TTWParams) => Permissions
begin
    var permissions : Permissions;
    var s2pi_index : bits(4);
    if walkparams.s2pie == '1' then
        if walkparams.d128 == '1' then
            s2pi_index = descriptor[118:115];
        else
            s2pi_index = descriptor[54:53,51,6];
        end;
        permissions.s2pi = walkparams.s2pir[UInt(s2pi_index)*:4];
        permissions.s2dirty = descriptor[7];
    else
        permissions.s2ap = descriptor[7:6];
        if walkparams.d128 == '1' then
            permissions.s2xn = descriptor[118];
        else
            permissions.s2xn = descriptor[54];
        end;

        if IsFeatureImplemented(FEAT_XNX) then
            if walkparams.d128 == '1' then
                permissions.s2xnx = descriptor[117];
            else
                permissions.s2xnx = descriptor[53];
            end;
        else
            permissions.s2xnx = '0';
        end;

        permissions.dbm = descriptor[51];
    end;
    if IsFeatureImplemented(FEAT_S2POE) then
        if walkparams.d128 == '1' then
            permissions.s2po_index = descriptor[124:121];
        else
            permissions.s2po_index = descriptor[62:59];
        end;
    end;
    return permissions;
end;
```

J1.2.4.71 AArch64_nTFaults

```
// AArch64_nTFaults()
// ==================
// Identify whether the nT bit in a block or table descriptor is effectively set
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// causing a translation fault

func AArch64_nTFaults{N}(d128 : bit, descriptor : bits(N)) => boolean
begin
    if !IsFeatureImplemented(FEAT_BBML1) then
        return FALSE;
    end;
    let nT : bit = if d128 == '1' then descriptor[6] else descriptor[16];

    return nT == '1' && ImpDefBool("nT bit causes Translation Fault");
end;
```

J1.2.4.72 AArch64_S1InitialTTWState

```
// AArch64_S1InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 1

func AArch64_S1InitialTTWState(walkparams : S1TTWParams, va : bits(64), regime : Regime,
                               ss : SecurityState) => TTWState
begin
    var walkstate : TTWState;
    var tablebase : FullAddress;
    var permissions : Permissions;
    var ttbr : bits(128);

    ttbr              = AArch64_S1TTBR(regime, va);
    case ss of
        when SS_Secure =>    tablebase.paspace = PAS_Secure;
        when SS_NonSecure => tablebase.paspace = PAS_NonSecure;
        when SS_Root =>      tablebase.paspace = PAS_Root;
        when SS_Realm =>     tablebase.paspace = PAS_Realm;
    end;

    tablebase.address = AArch64_S1TTBaseAddress{128}(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table  = '0';
    end;

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64_S1StartLevel(walkparams);
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;
    if regime == Regime_EL10 && EL2Enabled() && HCR_EL2().VM == '1' then
        if ((AArch64_GetVARange(va) == VARange_LOWER && VTCR_EL2().TL0 == '1') ||
            (AArch64_GetVARange(va) == VARange_UPPER && VTCR_EL2().TL1 == '1')) then
            walkstate.s1assured = TRUE;
        else
            walkstate.s1assured = FALSE;
        end;
    else
        walkstate.s1assured = FALSE;
    end;
    walkstate.disch = walkparams.disch;

    return walkstate;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.2.4.73 AArch64_S1NextWalkStateLeaf

```
// AArch64_S1NextWalkStateLeaf()
// ============================
// Decode stage 1 page or block descriptor as output to this stage of translation

func AArch64_S1NextWalkStateLeaf{N}(currentstate : TTWState, s2fs1mro : boolean, regime : Regime,
                                    accdesc : AccessDescriptor, walkparams : S1TTWParams,
                                    descriptor : bits(N)) => TTWState
begin
    var nextstate : TTWState;
    var baseaddress : FullAddress;
    baseaddress.address = AArch64_S1LeafBase{N}(descriptor, walkparams, currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        let ns : bit = if walkparams.d128 == '1' then descriptor[127] else descriptor[5];
        baseaddress.paspace = if ns == '0' then PAS_Secure else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Root then
        // Determine PA space of the block from NSE and NS bits
        let ns : bit = if walkparams.d128 == '1' then descriptor[127] else descriptor[5];
        let nse : bit = descriptor[11];
        let nse2 : bit = '0';     // NSE2 has the Effective value of 0 within a PE.
        baseaddress.paspace = DecodePASpace(nse2, nse, ns);

        // If Secure state is not implemented, but RME is,
        // force Secure space accesses to Non-secure space
        if baseaddress.paspace == PAS_Secure && !HaveSecureState() then
            baseaddress.paspace = PAS_NonSecure;
        end;

    elsif (currentstate.baseaddress.paspace == PAS_Realm &&
            regime IN {Regime_EL2, Regime_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        let ns : bit = if walkparams.d128 == '1' then descriptor[127] else descriptor[5];
        baseaddress.paspace = if ns == '0' then PAS_Realm else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Realm then
        // Realm EL1&0 regime does not have a stage 1 NS bit
        baseaddress.paspace = PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;
    end;

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    var attrindx : bits(4);
    if walkparams.aie == '1' then
        if walkparams.d128 == '1' then
            attrindx = descriptor[5:2];
        else
            attrindx = descriptor[59,4:2];
        end;
    else
        attrindx = '0'::descriptor[4:2];
    end;

    var sh : bits(2);
    if walkparams.d128 == '1' then
        sh = descriptor[9:8];
    elsif walkparams.ds == '1' then
        sh = walkparams.sh;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        sh = descriptor[9:8];
    end;
    let attr : bits(8) = AArch64_MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.mair);
    let s1aarch64 : boolean = TRUE;

    nextstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64, walkparams, accdesc.acctype);
    nextstate.permissions = AArch64_S1ApplyOutputPerms{N}(currentstate.permissions,
                                                          descriptor, regime, walkparams);
    var protectedbit : bit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor[114];
    else
        protectedbit = if walkparams.pnch == '1' then descriptor[52] else '0';
    end;
    if (currentstate.s1assured && s2fs1mro && protectedbit == '1') then
        nextstate.s1assured = TRUE;
    else
        nextstate.s1assured = FALSE;
    end;

    if walkparams.pnch == '1' || currentstate.disch == '1' then
        nextstate.contiguous = '0';
    else
        nextstate.contiguous = AArch64_ContiguousBit{N}(walkparams.tgx, walkparams.d128,
                                                        currentstate.level, descriptor);
    end;
    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif accdesc.ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure then
        // In Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
        nextstate.nG = '1';
    elsif walkparams.fng == '1' then
        // Translations are treated as non-global regardless of the value of the nG bit.
        nextstate.nG = '1';
    elsif (regime == Regime_EL10 && EL2Enabled() && HCR_EL2().VM == '1' &&
            (walkparams.d128 == '1' || walkparams.pnch == '1') &&
            !nextstate.s1assured && walkparams.fngna == '1') then
        // Translations are treated as non-global regardless of the value of the nG bit.
        nextstate.nG = '1';
    else
        nextstate.nG = descriptor[11];
    end;

    if walkparams.d128 == '1' then
        nextstate.guardedpage = descriptor[113];
    else
        nextstate.guardedpage = descriptor[50];
    end;

    return nextstate;
end;
```

J1.2.4.74 AArch64_S1NextWalkStateTable

```
// AArch64_S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

func AArch64_S1NextWalkStateTable{N}(currentstate : TTWState, s2fs1mro : boolean, regime : Regime,
                                     walkparams : S1TTWParams, descriptor : bits(N)) => TTWState
begin
    var nextstate : TTWState;
    var tablebase : FullAddress;
    let skl : bits(2) = if walkparams.d128 == '1' then descriptor[110:109] else '00';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    tablebase.address = AArch64_NextTableBase{N}(descriptor, walkparams.d128, skl, walkparams.ds,
                                                 walkparams.tgx);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        var nstable : bit;
        nstable = if walkparams.d128 == '1' then descriptor[127] else descriptor[63];
        tablebase.paspace = if nstable == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;
    end;

    nextstate.istable     = TRUE;
    nextstate.nG          = currentstate.nG;
    if walkparams.d128 == '1' then
        nextstate.level   = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level   = currentstate.level + 1;
    end;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;
    if walkparams.hpd == '0' && walkparams.pie == '0' then
        nextstate.permissions = AArch64_S1ApplyTablePerms(currentstate.permissions,
                                                          descriptor[63:0], regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;
    end;
    var protectedbit : bit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor[114];
    else
        protectedbit = if walkparams.pnch == '1' then descriptor[52] else '0';
    end;
    if (currentstate.s1assured && s2fs1mro && protectedbit == '1') then
        nextstate.s1assured = TRUE;
    else
        nextstate.s1assured = FALSE;
    end;
    nextstate.disch = if walkparams.d128 == '1' then descriptor[112] else '0';

    return nextstate;
end;
```

J1.2.4.75 AArch64_S1Walk

```
// AArch64_S1Walk()
// ================
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

func AArch64_S1Walk{N : integer{64, 128}}(fault_in : FaultRecord,
                                          walkparams : S1TTWParams,
                                          va : bits(64), regime : Regime,
                                          accdesc : AccessDescriptor
                                         ) => (FaultRecord, AddressDescriptor, TTWState, bits(N))
begin
    var fault : FaultRecord = fault_in;
    var aligned : boolean;

    if HasUnprivileged(regime) && AArch64_S1EPD(regime, va) == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var walkstate : TTWState = AArch64_S1InitialTTWState(walkparams, va, regime, accdesc.ss);
let startlevel : integer = walkstate.level;

if startlevel > 3 then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
end;

var descriptor : bits(N);
var walkaddress : AddressDescriptor;
var skl : bits(2) = '00';
walkaddress.vaddress = va;
walkaddress.mecid = AArch64_S1TTWalkMECID(walkparams.emec, regime, accdesc.ss);

if !AArch64_S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2().VM == '1' &&
        !(ImpDefBool("Apply effective shareability at stage 1"))) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
end;

var s2fs1mro : boolean = FALSE;

var desctype : DescriptorType;
var descaddress : FullAddress = AArch64_S1SLTTEntryAddress(walkstate.level, walkparams, va,
                                                           walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64_S1OAOutOfRange(descaddress.address, walkparams) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
end;

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    walkaddress.s1assured = walkstate.s1assured;

    let toplevel : boolean = walkstate.level == startlevel;
    let varange : VARange  = AArch64_GetVARange(va);
    let walkaccess : AccessDescriptor = CreateAccDescS1TTW(toplevel, varange, accdesc);
    var s2fault : FaultRecord;
    var s2walkaddress : AddressDescriptor;
    if regime == Regime_EL10 && EL2Enabled() then
        let s1aarch64 : boolean = TRUE;
        aligned   = TRUE;
        (s2fault, s2walkaddress) = AArch64_S2Translate(fault, walkaddress, s1aarch64, aligned,
                                                       walkaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                    ARBITRARY : bits(N));
        end;

        s2fs1mro = s2walkaddress.s2fs1mro;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    (fault, descriptor) = FetchDescriptor{N}(walkparams.ee, s2walkaddress, walkaccess,
                                             fault);
else
    (fault, descriptor) = FetchDescriptor{N}(walkparams.ee, walkaddress, walkaccess,
                                              fault);
end;

if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
            ARBITRARY : bits(N));
end;

var new_descriptor : bits(N);
repeat
    new_descriptor = descriptor;
    desctype = AArch64_DecodeDescriptorType{N}(descriptor, walkparams.d128, walkparams.ds,
                                               walkparams.tgx, walkstate.level);
    case desctype of
        when DescriptorType_Table =>
            walkstate = AArch64_S1NextWalkStateTable{N}(walkstate, s2fs1mro,
                                                        regime, walkparams, descriptor);
            skl = if walkparams.d128 == '1' then descriptor[110:109] else '00';
            descaddress = AArch64_S1TTEntryAddress{N}(walkstate.level, walkparams, skl, va,
                                                      walkstate.baseaddress, descriptor);

            // Detect Address Size Fault by Descriptor Address
            if AArch64_S1OAOutOfRange(descaddress.address, walkparams) then
                fault.statuscode = Fault_AddressSize;
                return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                        ARBITRARY : bits(N));
            end;

            if walkparams.haft == '1' then
                new_descriptor[10] = '1';
            end;
            if (walkparams.d128 == '1' && skl != '00' &&
                  AArch64_nTFaults{N}(walkparams.d128, descriptor)) then
                fault.statuscode = Fault_Translation;
                return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                        ARBITRARY : bits(N));
            end;
        when DescriptorType_Leaf =>
            walkstate = AArch64_S1NextWalkStateLeaf{N}(walkstate, s2fs1mro, regime, accdesc,
                                                       walkparams, descriptor);
        when DescriptorType_Invalid =>
            fault.statuscode = Fault_Translation;
            return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                    ARBITRARY : bits(N));
        otherwise =>
            unreachable;
    end;

    if new_descriptor != descriptor then

        if (!IsWBShareable(walkaddress.memattrs) &&
              ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
            fault.statuscode = Fault_UnsupportedAtomicHWUpdate;
            return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                    ARBITRARY : bits(N));
        end;

        var descpaddr : AddressDescriptor;
        let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
        if regime == Regime_EL10 && EL2Enabled() then
            let s1aarch64 : boolean = TRUE;
            aligned   = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    (s2fault, descpaddr) = AArch64_S2Translate(fault, walkaddress,
                                                               s1aarch64, aligned,
                                                               descaccess);

                    if s2fault.statuscode != Fault_None then
                        return (s2fault, ARBITRARY : AddressDescriptor,
                                ARBITRARY : TTWState, ARBITRARY : bits(N));
                    end;
                else
                    descpaddr = walkaddress;
                end;

                (fault, descriptor) = AArch64_MemSwapTableDesc{N}(fault, descriptor, new_descriptor,
                                                                  walkparams.ee, descaccess,
                                                                  descpaddr);
                if fault.statuscode != Fault_None then
                    return (fault, ARBITRARY : AddressDescriptor,
                            ARBITRARY : TTWState, ARBITRARY : bits(N));
                end;
            end;
        until new_descriptor == descriptor looplimit Unbounded_DescriptorUpdate;
    until desctype == DescriptorType_Leaf looplimit MAX_WALK_LEVELS;

    let oa : FullAddress = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
    if (walkstate.contiguous == '1' &&
        AArch64_ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                    walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif walkstate.level < FINAL_LEVEL && AArch64_nTFaults{N}(walkparams.d128, descriptor) then
        fault.statuscode = Fault_Translation;
    elsif AArch64_S1AMECFault{N}(walkparams, walkstate.baseaddress.paspace, regime, descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64_S1OAOutOfRange(oa.address, walkparams) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor[10] == '0' && walkparams.ha == '0' &&
            (!accdesc.acctype IN {AccessType_DC, AccessType_IC} ||
             ImpDefBool("Generate access flag fault on IC/DC operations"))) then
        fault.statuscode = Fault_AccessFlag;
    end;

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    return (fault, walkaddress, walkstate, descriptor);
end;
```

J1.2.4.76 AArch64_S2InitialTTWState

```
// AArch64_S2InitialTTWState()
// ==========================
// Set properties of first access to translation tables in stage 2

func AArch64_S2InitialTTWState(ss : SecurityState, walkparams : S2TTWParams) => TTWState
begin
    var walkstate : TTWState;
    var tablebase : FullAddress;
    var ttbr : bits(128);

    ttbr = ZeroExtend{128}(VTTBR_EL2());
    case ss of
        when SS_NonSecure => tablebase.paspace = PAS_NonSecure;
        when SS_Realm     => tablebase.paspace = PAS_Realm;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    tablebase.address = AArch64_S2TTBaseAddress{128}(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64_S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
end;
```

J1.2.4.77 AArch64_S2NextWalkStateLeaf

```
// AArch64_S2NextWalkStateLeaf()
// ============================
// Decode stage 2 page or block descriptor as output to this stage of translation

func AArch64_S2NextWalkStateLeaf{N}(currentstate : TTWState, ss : SecurityState,
                                    walkparams : S2TTWParams, ipa : AddressDescriptor,
                                    descriptor : bits(N)) => TTWState
begin
    var nextstate : TTWState;
    var baseaddress : FullAddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64_SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    elsif ss == SS_Realm then
        var ns : bit;
        ns = if walkparams.d128 == '1' then descriptor[127] else descriptor[55];
        baseaddress.paspace = if ns == '1' then PAS_NonSecure else PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;
    end;
    baseaddress.address   = AArch64_S2LeafBase{N}(descriptor, walkparams, currentstate.level);

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64_S2ApplyOutputPerms{N}(descriptor, walkparams);

    let s2_attr : bits(4) = descriptor[5:2];
    let s2_sh : bits(2)   = if walkparams.ds == '1' then walkparams.sh else descriptor[9:8];
    let s2_fnxs : bit     = descriptor[11];
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64_S2ApplyFWBMemAttrs{N}(ipa.memattrs, walkparams, descriptor);
        if s2_attr[3:1] == '111' then
            nextstate.permissions.s2tag_na = '1';
        else
            nextstate.permissions.s2tag_na = '0';
        end;
    else
        let s2aarch64 : boolean = TRUE;
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
        // FnXS is used later to mask the XS value from stage 1
        nextstate.memattrs.xs = NOT s2_fnxs;
        if s2_attr == '0100' then
            nextstate.permissions.s2tag_na = '1';
        else
            nextstate.permissions.s2tag_na = '0';
        end;
    end;
    nextstate.contiguous = AArch64_ContiguousBit{N}(walkparams.tgx, walkparams.d128,
                                                    currentstate.level, descriptor);
    if walkparams.d128 == '1' then
        nextstate.s2assuredonly = descriptor[114];
    else
        nextstate.s2assuredonly = if walkparams.assuredonly == '1' then descriptor[58] else '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    return nextstate;
end;
```

J1.2.4.78 AArch64_S2NextWalkStateTable

```
// AArch64_S2NextWalkStateTable()
// ==============================
// Decode stage 2 table descriptor to transition to the next level

func AArch64_S2NextWalkStateTable{N}(currentstate : TTWState, walkparams : S2TTWParams,
                                     descriptor : bits(N)) => TTWState
begin
    var nextstate : TTWState;
    var tablebase : FullAddress;
    let skl : bits(2) = if walkparams.d128 == '1' then descriptor[110:109] else '00';
    tablebase.address = AArch64_NextTableBase{N}(descriptor, walkparams.d128, skl, walkparams.ds,
                                                 walkparams.tgx);

    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    if walkparams.d128 == '1' then
        nextstate.level   = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level   = currentstate.level + 1;
    end;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
end;
```

J1.2.4.79 AArch64_S2Walk

```
// AArch64_S2Walk()
// ================
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

func AArch64_S2Walk{N : integer{64, 128}}(fault_in : FaultRecord,
                                          ipa : AddressDescriptor,
                                          walkparams : S2TTWParams,
                                          accdesc : AccessDescriptor) => (FaultRecord,
                                                                          AddressDescriptor,
                                                                          TTWState,
                                                                          bits(N))
begin

    var fault : FaultRecord = fault_in;
    let ipa_64 : bits(64) = ZeroExtend{}(ipa.paddress.address);

    var walkstate : TTWState;
    if accdesc.ss == SS_Secure then
        walkstate = AArch64_SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64_S2InitialTTWState(accdesc.ss, walkparams);
    end;
    let startlevel : integer = walkstate.level;

    if startlevel > 3 then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

var descriptor : bits(N);
let walkaccess : AccessDescriptor = CreateAccDescS2TTW(accdesc);
var walkaddress : AddressDescriptor;
var skl : bits(2) = '00';

walkaddress.vaddress = ipa.vaddress;
walkaddress.mecid = AArch64_S2TTWalkMECID(walkparams.emec, accdesc.ss);

if !S2DCacheEnabled() then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

var desctype : DescriptorType;

// Initial lookup might index into concatenated tables
var descaddress : FullAddress = AArch64_S2SLTTEntryAddress(walkparams, ipa.paddress.address,
                                                           walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64_S2OAOutOfRange(descaddress.address, walkparams) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
end;

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor{N}(walkparams.ee, walkaddress, walkaccess, fault);

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor,
                ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    var new_descriptor : bits(N);
    repeat
        new_descriptor = descriptor;
        desctype = AArch64_DecodeDescriptorType{N}(descriptor, walkparams.d128, walkparams.ds,
                                                   walkparams.tgx, walkstate.level);
        case desctype of
            when DescriptorType_Table =>
                walkstate = AArch64_S2NextWalkStateTable{N}(walkstate, walkparams, descriptor);
                skl = if walkparams.d128 == '1' then descriptor[110:109] else '00';
                descaddress = AArch64_S2TTEntryAddress(walkstate.level, walkparams, skl,
                                                       ipa.paddress.address,
                                                       walkstate.baseaddress);

                // Detect Address Size Fault by table descriptor
                if AArch64_S2OAOutOfRange(descaddress.address, walkparams) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                            ARBITRARY : bits(N));
                end;

                if walkparams.haft == '1' then
                    new_descriptor[10] = '1';
                end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    if (walkparams.d128 == '1' && skl != '00' &&
                          AArch64_nTFaults{N}(walkparams.d128, descriptor)) then
                        fault.statuscode = Fault_Translation;
                        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                                ARBITRARY : bits(N));
                    end;

                when DescriptorType_Leaf =>
                    walkstate = AArch64_S2NextWalkStateLeaf{N}(walkstate, accdesc.ss, walkparams,
                                                               ipa, descriptor);
                when DescriptorType_Invalid =>
                    fault.statuscode = Fault_Translation;
                    return (fault, ARBITRARY : AddressDescriptor,
                            ARBITRARY : TTWState, ARBITRARY : bits(N));

                otherwise =>
                    unreachable;
            end;

            if new_descriptor != descriptor then

                if (!IsWBShareable(walkaddress.memattrs) &&
                      ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
                    fault.statuscode = Fault_UnsupportedAtomicHWUpdate;
                    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                            ARBITRARY : bits(N));
                end;

                let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
                (fault, descriptor) = AArch64_MemSwapTableDesc{N}(fault, descriptor, new_descriptor,
                                                                  walkparams.ee, descaccess,
                                                                  walkaddress);
                if fault.statuscode != Fault_None then
                    return (fault, ARBITRARY : AddressDescriptor,
                            ARBITRARY : TTWState, ARBITRARY : bits(N));
                end;
            end;
        until new_descriptor == descriptor looplimit Unbounded_DescriptorUpdate;
    until desctype == DescriptorType_Leaf looplimit MAX_WALK_LEVELS;

    let oa : FullAddress = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    if (walkstate.contiguous == '1' &&
        AArch64_ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                    walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif walkstate.level < FINAL_LEVEL && AArch64_nTFaults{N}(walkparams.d128, descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64_S2OAOutOfRange(oa.address, walkparams) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor[10] == '0' && walkparams.ha == '0' &&
            (!accdesc.acctype IN {AccessType_DC, AccessType_IC} ||
             ImpDefBool("Generate access flag fault on IC/DC operations"))) then
        fault.statuscode = Fault_AccessFlag;
    end;

    return (fault, walkaddress, walkstate, descriptor);
end;
```

J1.2.4.80 AArch64_SS2InitialTTWState

```
// AArch64_SS2InitialTTWState()
// ============================
// Set properties of first access to translation tables in Secure stage 2
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_SS2InitialTTWState(walkparams : S2TTWParams, ipaspace : PASpace) => TTWState
begin
    var walkstate : TTWState;
    var tablebase : FullAddress;
    var ttbr : bits(128);

    if ipaspace == PAS_Secure then
        ttbr = ZeroExtend{128}(VSTTBR_EL2());
    else
        ttbr = ZeroExtend{128}(VTTBR_EL2());
    end;

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
        end;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
        end;
    end;
    tablebase.address = AArch64_S2TTBaseAddress{128}(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64_S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
end;
```

J1.2.4.81 AArch64_SS2OutputPASpace

```
// AArch64_SS2OutputPASpace()
// ==========================
// Assign PA Space to output of Secure stage 2 translation

func AArch64_SS2OutputPASpace(walkparams : S2TTWParams, ipaspace : PASpace) => PASpace
begin
    if ipaspace == PAS_Secure then
        if walkparams.[sw,sa] == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
        end;
    else
        if walkparams.[sw,sa,nsw,nsa] == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
        end;
    end;
end;
```

J1.2.4.82 AArch64_GetS1TTWParams

```
// AArch64_GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch64_GetS1TTWParams(regime : Regime, el : bits(2),
                            ss : SecurityState, va : bits(64)) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    let varange : VARange = AArch64_GetVARange(va);

    case regime of
        when Regime_EL3 =>  walkparams = AArch64_S1TTWParamsEL3();
        when Regime_EL2 =>  walkparams = AArch64_S1TTWParamsEL2(ss);
        when Regime_EL20 => walkparams = AArch64_S1TTWParamsEL20(el, ss, varange);
        when Regime_EL10 => walkparams = AArch64_S1TTWParamsEL10(el, varange);
    end;

    return walkparams;
end;
```

J1.2.4.83 AArch64_GetS2TTWParams

```
// AArch64_GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

func AArch64_GetS2TTWParams(ss : SecurityState, ipaspace : PASpace,
                            s1aarch64 : boolean) => S2TTWParams
begin
    var walkparams : S2TTWParams;

    if ss == SS_NonSecure then
        walkparams = AArch64_NSS2TTWParams(s1aarch64);
    elsif IsFeatureImplemented(FEAT_SEL2) && ss == SS_Secure then
        walkparams = AArch64_SS2TTWParams(ipaspace, s1aarch64);
    elsif ss == SS_Realm then
        walkparams = AArch64_RLS2TTWParams(s1aarch64);
    else
        unreachable;
    end;

    return walkparams;
end;
```

J1.2.4.84 AArch64_GetVARange

```
// AArch64_GetVARange()
// ====================
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

func AArch64_GetVARange(va : bits(64)) => VARange
begin
    if va[55] == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
    end;
end;
```

J1.2.4.85 AArch64_HaveS1TG

```
// AArch64_HaveS1TG()
// ==================
// Determine whether the given translation granule is supported for stage 1

func AArch64_HaveS1TG(tgx : TGx) => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    case tgx of
        when TGx_4KB =>  return IsFeatureImplemented(FEAT_TGran4K);
        when TGx_16KB => return IsFeatureImplemented(FEAT_TGran16K);
        when TGx_64KB => return IsFeatureImplemented(FEAT_TGran64K);
    end;
end;
```

J1.2.4.86 AArch64_HaveS2TG

```
// AArch64_HaveS2TG()
// ==================
// Determine whether the given translation granule is supported for stage 2

func AArch64_HaveS2TG(tgx : TGx) => boolean
begin
    assert HaveEL(EL2);

    if IsFeatureImplemented(FEAT_GTG) then
        case tgx of
            when TGx_4KB =>  return IsFeatureImplemented(FEAT_S2TGran4K);
            when TGx_16KB => return IsFeatureImplemented(FEAT_S2TGran16K);
            when TGx_64KB => return IsFeatureImplemented(FEAT_S2TGran64K);
        end;
    else
        return AArch64_HaveS1TG(tgx);
    end;
end;
```

J1.2.4.87 AArch64_MaxTxSZ

```
// AArch64_MaxTxSZ()
// =================
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

func AArch64_MaxTxSZ(tgx : TGx) => integer
begin
    if IsFeatureImplemented(FEAT_TTST) then
        case tgx of
            when TGx_4KB =>   return 48;
            when TGx_16KB =>  return 48;
            when TGx_64KB =>  return 47;
        end;
    end;

    return 39;
end;
```

J1.2.4.88 AArch64_NSS2TTWParams

```
// AArch64_NSS2TTWParams()
// =======================
// Gather walk parameters specific for Non-secure stage 2 translation

func AArch64_NSS2TTWParams(s1aarch64 : boolean) => S2TTWParams
begin
    var walkparams : S2TTWParams;

    walkparams.vm   = HCR_EL2().VM OR HCR_EL2().DC;
    walkparams.tgx  = AArch64_S2DecodeTG0(VTCR_EL2().TG0);
    walkparams.txsz = VTCR_EL2().T0SZ;
    walkparams.ps   = VTCR_EL2().PS;
    walkparams.irgn = VTCR_EL2().IRGN0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkparams.orgn = VTCR_EL2().ORGN0;
    walkparams.sh   = VTCR_EL2().SH0;
    walkparams.ee   = SCTLR_EL2().EE;
    walkparams.d128 = if IsFeatureImplemented(FEAT_D128)  then VTCR_EL2().D128 else '0';
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2().SKL;
    else
        walkparams.sl0 = VTCR_EL2().SL0;
    end;

    walkparams.ptw = if HCR_EL2().TGE == '0'                then HCR_EL2().PTW else '0';
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB)  then HCR_EL2().FWB else '0';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then VTCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = VTCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2().DS;
    else
        walkparams.ds = '0';
    end;
    if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.sl2 = VTCR_EL2().SL2 AND VTCR_EL2().DS;
    else
        walkparams.sl2 = '0';
    end;
    walkparams.cmow = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2Enabled() then HCRX_EL2().CMOW
                       else '0');
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2().S2PIE else '0';
    end;
    if IsFeatureImplemented(FEAT_S2PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
              ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.s2pir = S2PIR_EL2() as S2PIRType;
        else
            walkparams.s2pir = Zeros{64} as S2PIRType;
        end;
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2().AssuredOnly;
    else
        walkparams.assuredonly = '0';
    end;
    walkparams.tl0   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL0 else '0';
    walkparams.tl1   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL1 else '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;
    if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
          (!HaveEL(EL3) || SCR_EL3().HDBSSEn == '1')) then
        walkparams.hdbss = VTCR_EL2().HDBSS;
    else
        walkparams.hdbss = '0';
    end;

    return walkparams;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.89 AArch64_PAMax

```
// AArch64_PAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this PE

func AArch64_PAMax() => AddressSize
begin
    return ImpDefInt("Maximum Physical Address Size") as AddressSize;
end;
```

J1.2.4.90 AArch64_RLS2TTWParams

```
// AArch64_RLS2TTWParams()
// =======================
// Gather walk parameters specific for Realm stage 2 translation

func AArch64_RLS2TTWParams(s1aarch64 : boolean) => S2TTWParams
begin
    // Realm stage 2 walk parameters are similar to Non-secure
    var walkparams : S2TTWParams = AArch64_NSS2TTWParams(s1aarch64);
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                       IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');
    return walkparams;
end;
```

J1.2.4.91 AArch64_S1DCacheEnabled

```
// AArch64_S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

func AArch64_S1DCacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL3 =>  return SCTLR_EL3().C == '1';
        when Regime_EL2 =>  return SCTLR_EL2().C == '1';
        when Regime_EL20 => return SCTLR_EL2().C == '1';
        when Regime_EL10 => return SCTLR_EL1().C == '1';
    end;
end;
```

J1.2.4.92 AArch64_S1DecodeTG0

```
// AArch64_S1DecodeTG0()
// =====================
// Decode stage 1 granule size configuration bits TG0

func AArch64_S1DecodeTG0(tg0_in : bits(2)) => TGx
begin
    var tg0 : bits(2) = tg0_in;
    var tgx : TGx;

    if tg0 == '11' then
        tg0 = ImpDefBits{2}("TG0 encoded granule size");
    end;

    case tg0 of
        when '00' =>   tgx = TGx_4KB;
        when '01' =>   tgx = TGx_64KB;
        when '10' =>   tgx = TGx_16KB;
    end;

    if !AArch64_HaveS1TG(tgx) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        case ImpDefBits{2}("TG0 encoded granule size") of
            when '00' =>   tgx = TGx_4KB;
            when '01' =>   tgx = TGx_64KB;
            when '10' =>   tgx = TGx_16KB;
        end;
    end;

    return tgx;
end;
```

J1.2.4.93 AArch64_S1DecodeTG1

```
// AArch64_S1DecodeTG1()
// =====================
// Decode stage 1 granule size configuration bits TG1

func AArch64_S1DecodeTG1(tg1_in : bits(2)) => TGx
begin
    var tg1 : bits(2) = tg1_in;
    var tgx : TGx;

    if tg1 == '00' then
        tg1 = ImpDefBits{2}("TG1 encoded granule size");
    end;

    case tg1 of
        when '10' =>   tgx = TGx_4KB;
        when '11' =>   tgx = TGx_64KB;
        when '01' =>   tgx = TGx_16KB;
    end;

    if !AArch64_HaveS1TG(tgx) then
        case ImpDefBits{2}("TG1 encoded granule size") of
            when '10' =>   tgx = TGx_4KB;
            when '11' =>   tgx = TGx_64KB;
            when '01' =>   tgx = TGx_16KB;
        end;
    end;

    return tgx;
end;
```

J1.2.4.94 AArch64_S1E0POEnabled

```
// AArch64_S1E0POEnabled()
// =======================
// Determine whether stage 1 unprivileged permission overlay is enabled

func AArch64_S1E0POEnabled(regime : Regime, nv1 : bit) => boolean
begin
    assert HasUnprivileged(regime);

    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;
    end;

    case regime of
        when Regime_EL20 => return IsTCR2EL2Enabled() && TCR2_EL2().E0POE == '1';
        when Regime_EL10 => return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1().E0POE == '1';
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.95 AArch64_S1EPD

```
// AArch64_S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range

func AArch64_S1EPD(regime : Regime, va : bits(64)) => bit
begin
    assert HasUnprivileged(regime);
    let varange : VARange = AArch64_GetVARange(va);

    case regime of
        when Regime_EL20 => return (if varange == VARange_LOWER then TCR_EL2().EPD0
                                                                else TCR_EL2().EPD1);
        when Regime_EL10 => return (if varange == VARange_LOWER then TCR_EL1().EPD0
                                                                else TCR_EL1().EPD1);
    end;
end;
```

J1.2.4.96 AArch64_S1Enabled

```
// AArch64_S1Enabled()
// ===================
// Determine if stage 1 is enabled for the access type for this translation regime

func AArch64_S1Enabled(regime : Regime, acctype : AccessType) => boolean
begin
    if acctype == AccessType_TRBE && EffectiveTRBLIMITR_EL1_nVM()  == '1' then
        return FALSE;
    end;
    if acctype == AccessType_SPE && EffectivePMBLIMITR_EL1_nVM() == '1' then
        return FALSE;
    end;
    case regime of
        when Regime_EL3 =>  return SCTLR_EL3().M == '1';
        when Regime_EL2 =>  return SCTLR_EL2().M == '1';
        when Regime_EL20 => return SCTLR_EL2().M == '1';
        when Regime_EL10 => return ((!EL2Enabled() || HCR_EL2().[DC,TGE] == '00') &&
                                    SCTLR_EL1().M == '1');
    end;
end;
```

J1.2.4.97 AArch64_S1ICacheEnabled

```
// AArch64_S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

func AArch64_S1ICacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL3 =>  return SCTLR_EL3().I == '1';
        when Regime_EL2 =>  return SCTLR_EL2().I == '1';
        when Regime_EL20 => return SCTLR_EL2().I == '1';
        when Regime_EL10 => return SCTLR_EL1().I == '1';
    end;
end;
```

J1.2.4.98 AArch64_S1MinTxSZ

```
// AArch64_S1MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

func AArch64_S1MinTxSZ(regime : Regime, walkparams : S1TTWParams) => integer
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    if IsFeatureImplemented(FEAT_LVA3) then
        if walkparams.d128 == '1' then
            if HasUnprivileged(regime) then
                return 9;
            else
                return 8;
            end;
        elsif walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            return 12;
        else
            return 16;
        end;
    end;
    if IsFeatureImplemented(FEAT_LVA) then
        if walkparams.tgx == TGx_64KB || walkparams.ds == '1' || walkparams.d128 == '1' then
            return 12;
        else
            return 16;
        end;
    end;

    return 16;
end;
```

J1.2.4.99 AArch64_S1POEnabled

```
// AArch64_S1POEnabled()
// =====================
// Determine whether stage 1 privileged permission overlay is enabled

func AArch64_S1POEnabled(regime : Regime) => boolean
begin
    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;
    end;

    case regime of
        when Regime_EL3 =>  return TCR_EL3().POE == '1';
        when Regime_EL2 =>  return IsTCR2EL2Enabled() && TCR2_EL2().POE == '1';
        when Regime_EL20 => return IsTCR2EL2Enabled() && TCR2_EL2().POE == '1';
        when Regime_EL10 => return IsTCR2EL1Enabled() && TCR2_EL1().POE == '1';
    end;
end;
```

J1.2.4.100 AArch64_S1POR

```
// AArch64_S1POR()
// ===============
// Identify stage 1 permissions overlay register for the acting translation regime

func AArch64_S1POR(regime : Regime) => S1PORType
begin
    if (HaveEL(EL3) && SCR_EL3().PIEn == '0' && regime != Regime_EL3 &&
          ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
        return Zeros{64} as S1PORType;
    end;

    case regime of
        when Regime_EL3 =>  return POR_EL3() as S1PORType;
        when Regime_EL2 =>  return POR_EL2() as S1PORType;
        when Regime_EL20 => return POR_EL2() as S1PORType;
        when Regime_EL10 => return POR_EL1() as S1PORType;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.101 AArch64_S1TTBR

```
// AArch64_S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

func AArch64_S1TTBR(regime : Regime, va : bits(64)) => bits(128)
begin
    let varange : VARange = AArch64_GetVARange(va);

    case regime of
        when Regime_EL3 =>  return ZeroExtend{128}(TTBR0_EL3());
        when Regime_EL2 =>  return ZeroExtend{128}(TTBR0_EL2());
        when Regime_EL20 =>
            if varange == VARange_LOWER then
                return ZeroExtend{128}(TTBR0_EL2());
            else
                return ZeroExtend{128}(TTBR1_EL2());
            end;
        when Regime_EL10 =>
            if varange == VARange_LOWER then
                return ZeroExtend{128}(TTBR0_EL1());
            else
                return ZeroExtend{128}(TTBR1_EL1());
            end;
    end;
end;
```

J1.2.4.102 AArch64_S1TTWParamsEL10

```
// AArch64_S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

func AArch64_S1TTWParamsEL10(el : bits(2), varange : VARange) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL1Enabled() then
        walkparams.d128 = TCR2_EL1().D128;
    else
        walkparams.d128 = '0';
    end;
    let nvs : bits(3) = EffectiveHCR_EL2_NVx();
    walkparams.nv1 = nvs[1];

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL1() as MAIRType;
    end;
    walkparams.aie  = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL1Enabled() then TCR2_EL1().AIE
                       else '0');
    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
                          IsTCR2EL1Enabled() then TCR2_EL1().PIE else '0');
    end;
    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
               ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.pir = PIR_EL1() as S1PIRType;
            if walkparams.nv1 == '1' then
                walkparams.pire0 = Zeros{64} as S1PIRType;
            else
                walkparams.pire0 = PIRE0_EL1() as S1PIRType;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
        else
            walkparams.pir   = Zeros{64} as S1PIRType;
            walkparams.pire0 = Zeros{64} as S1PIRType;
        end;
    end;

    if varange == VARange_LOWER then
        walkparams.tgx   = AArch64_S1DecodeTG0(TCR_EL1().TG0);
        walkparams.txsz  = TCR_EL1().T0SZ;
        walkparams.irgn  = TCR_EL1().IRGN0;
        walkparams.orgn  = TCR_EL1().ORGN0;
        walkparams.sh    = TCR_EL1().SH0;
        walkparams.tbi   = TCR_EL1().TBI0;

        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL1().NFD0  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1().TBID0 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL1().E0PD0 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL1().HPD0  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if (AArch64_S1POEnabled(Regime_EL10) ||
                  AArch64_S1E0POEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
            end;
        end;
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL1().MTX0;
        else
            walkparams.mtx  = '0';
        end;
        walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL1().SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1().DisCH0 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() then
            walkparams.fng = TCR2_EL1().FNG0;
        else
            walkparams.fng = '0';
        end;
        if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
            walkparams.fngna = TCR2_EL1().FNGNA0;
        else
            walkparams.fngna = '0';
        end;
    else
        walkparams.tgx   = AArch64_S1DecodeTG1(TCR_EL1().TG1);
        walkparams.txsz  = TCR_EL1().T1SZ;
        walkparams.irgn  = TCR_EL1().IRGN1;
        walkparams.orgn  = TCR_EL1().ORGN1;
        walkparams.sh    = TCR_EL1().SH1;
        walkparams.tbi   = TCR_EL1().TBI1;

        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL1().NFD1  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1().TBID1 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL1().E0PD1 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL1().HPD1  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if (AArch64_S1POEnabled(Regime_EL10) ||
                  AArch64_S1E0POEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
            end;
        end;
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL1().MTX1;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        walkparams.mtx  = '0';
    end;
    walkparams.skl   = if walkparams.d128 == '1' then TTBR1_EL1().SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1().DisCH1 else '0';
    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() then
        walkparams.fng = TCR2_EL1().FNG1;
    else
        walkparams.fng = '0';
    end;
    if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
        walkparams.fngna = TCR2_EL1().FNGNA1;
    else
        walkparams.fngna = '0';
    end;
end;

walkparams.mair = MAIR_EL1() as MAIRType;
walkparams.wxn  = SCTLR_EL1().WXN;
walkparams.ps   = TCR_EL1().IPS;
walkparams.ee   = SCTLR_EL1().EE;
if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
    walkparams.sif = SCR_EL3().SIF;
else
    walkparams.sif = '0';
end;

if EL2Enabled() then
    walkparams.dc  = HCR_EL2().DC;
    walkparams.dct = if IsFeatureImplemented(FEAT_MTE2) then HCR_EL2().DCT else '0';
end;

if IsFeatureImplemented(FEAT_LSMAOC) then
    walkparams.ntlsmd = SCTLR_EL1().nTLSMD;
else
    walkparams.ntlsmd = '1';
end;

walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW)   then SCTLR_EL1().CMOW else '0';
walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL1().HA else '0';
if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
    walkparams.hd = TCR_EL1().HD;
else
    walkparams.hd = '0';
end;
if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
      walkparams.d128 == '0') then
    walkparams.ds = TCR_EL1().DS;
else
    walkparams.ds = '0';
end;
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL1().EPAN else '1';
else
    walkparams.epan = '0';
end;
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL1Enabled() then
    walkparams.pnch = TCR2_EL1().PnCH;
else
    walkparams.pnch = '0';
end;
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL1Enabled() then
    walkparams.haft = TCR2_EL1().HAFT;
else
    walkparams.haft = '0';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                       IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');

    return walkparams;
end;
```

J1.2.4.103 AArch64_S1TTWParamsEL2

```
// AArch64_S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime

func AArch64_S1TTWParamsEL2(ss : SecurityState) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    walkparams.tgx  = AArch64_S1DecodeTG0(TCR_EL2().TG0);
    walkparams.txsz = TCR_EL2().T0SZ;
    walkparams.ps   = TCR_EL2().PS;
    walkparams.irgn = TCR_EL2().IRGN0;
    walkparams.orgn = TCR_EL2().ORGN0;
    walkparams.sh   = TCR_EL2().SH0;
    walkparams.tbi  = TCR_EL2().TBI;
    walkparams.mair = MAIR_EL2() as MAIRType;

    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) && IsTCR2EL2Enabled() then TCR2_EL2().PIE
                      else '0');

    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
                ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.pir = PIR_EL2() as S1PIRType;
        else
            walkparams.pir = Zeros{64} as S1PIRType;
        end;
    end;
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2() as MAIRType;
    end;
    walkparams.aie  = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled() then TCR2_EL2().AIE
                       else '0');
    walkparams.wxn  = SCTLR_EL2().WXN;
    walkparams.ee   = SCTLR_EL2().EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
        walkparams.sif = SCR_EL3().SIF;
    else
        walkparams.sif = '0';
    end;

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth)  then TCR_EL2().TBID else '0';
    walkparams.hpd  = if IsFeatureImplemented(FEAT_HPDS)   then TCR_EL2().HPD  else '0';
    if walkparams.hpd == '0' then
        if walkparams.aie == '1' then walkparams.hpd = '1'; end;
        if walkparams.pie == '1' then walkparams.hpd = '1'; end;
        if AArch64_S1POEnabled(Regime_EL2) then walkparams.hpd = '1'; end;
    end;
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = TCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = TCR_EL2().DS;
    else
        walkparams.ds = '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
          IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
        walkparams.mtx  = TCR_EL2().MTX;
    else
        walkparams.mtx  = '0';
    end;
    walkparams.pnch = (if IsFeatureImplemented(FEAT_THE) && IsTCR2EL2Enabled() then TCR2_EL2().PnCH
                       else '0');
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                       IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');
    if IsFeatureImplemented(FEAT_MEC) && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = TCR2_EL2().AMEC0;
    else
        walkparams.amec = '0';
    end;

    return walkparams;
end;
```

J1.2.4.104 AArch64_S1TTWParamsEL20

```
// AArch64_S1TTWParamsEL20()
// =========================
// Gather stage 1 translation table walk parameters for EL2&0 regime

func AArch64_S1TTWParamsEL20(el : bits(2), ss : SecurityState, varange : VARange) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL2Enabled() then
        walkparams.d128 = TCR2_EL2().D128;
    else
        walkparams.d128 = '0';
    end;

    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
                          IsTCR2EL2Enabled() then TCR2_EL2().PIE else '0');
    end;
    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
                ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.pir   = PIR_EL2() as S1PIRType;
            walkparams.pire0 = PIRE0_EL2() as S1PIRType;
        else
            walkparams.pir   = Zeros{64} as S1PIRType;
            walkparams.pire0 = Zeros{64} as S1PIRType;
        end;
    end;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2() as MAIRType;
    end;
    walkparams.aie  = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled() then TCR2_EL2().AIE
                       else '0');
    if varange == VARange_LOWER then
        walkparams.tgx   = AArch64_S1DecodeTG0(TCR_EL2().TG0);
        walkparams.txsz  = TCR_EL2().T0SZ;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        walkparams.irgn  = TCR_EL2().IRGN0;
        walkparams.orgn  = TCR_EL2().ORGN0;
        walkparams.sh    = TCR_EL2().SH0;
        walkparams.tbi   = TCR_EL2().TBI0;

        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL2().NFD0  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2().TBID0 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL2().E0PD0 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL2().HPD0  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if AArch64_S1POEnabled(Regime_EL20) || AArch64_S1E0POEnabled(Regime_EL20, '0') then
                walkparams.hpd = '1';
            end;
        end;
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL2().MTX0;
        else
            walkparams.mtx  = '0';
        end;
        walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL2().SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2().DisCH0 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() then
            walkparams.fng = TCR2_EL2().FNG0;
        else
            walkparams.fng = '0';
        end;
    else
        walkparams.tgx   = AArch64_S1DecodeTG1(TCR_EL2().TG1);
        walkparams.txsz  = TCR_EL2().T1SZ;
        walkparams.irgn  = TCR_EL2().IRGN1;
        walkparams.orgn  = TCR_EL2().ORGN1;
        walkparams.sh    = TCR_EL2().SH1;
        walkparams.tbi   = TCR_EL2().TBI1;

        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL2().NFD1  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2().TBID1 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL2().E0PD1 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL2().HPD1  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if AArch64_S1POEnabled(Regime_EL20) || AArch64_S1E0POEnabled(Regime_EL20, '0') then
                walkparams.hpd = '1';
            end;
        end;
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL2().MTX1;
        else
            walkparams.mtx  = '0';
        end;
        walkparams.skl   = if walkparams.d128 == '1' then TTBR1_EL2().SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2().DisCH1 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() then
            walkparams.fng = TCR2_EL2().FNG1;
        else
            walkparams.fng = '0';
        end;
    end;

    walkparams.mair = MAIR_EL2() as MAIRType;
    walkparams.wxn  = SCTLR_EL2().WXN;
    walkparams.ps   = TCR_EL2().IPS;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkparams.ee   = SCTLR_EL2().EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
        walkparams.sif = SCR_EL3().SIF;
    else
        walkparams.sif = '0';
    end;

    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = SCTLR_EL2().nTLSMD;
    else
        walkparams.ntlsmd = '1';
    end;

    walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW)   then SCTLR_EL2().CMOW else '0';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = TCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;
    if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
          walkparams.d128 == '0') then
        walkparams.ds = TCR_EL2().DS;
    else
        walkparams.ds = '0';
    end;
    if IsFeatureImplemented(FEAT_PAN3) then
        walkparams.epan = if walkparams.pie == '0' then SCTLR_EL2().EPAN else '1';
    else
        walkparams.epan = '0';
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL2Enabled() then
        walkparams.pnch = TCR2_EL2().PnCH;
    else
        walkparams.pnch = '0';
    end;
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) && IsSCTLR2EL2Enabled()
                        then SCTLR2_EL2().EMEC else '0');
    if IsFeatureImplemented(FEAT_MEC) && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = if varange == VARange_LOWER then TCR2_EL2().AMEC0 else TCR2_EL2().AMEC1;
    else
        walkparams.amec = '0';
    end;

    return walkparams;
end;
```

J1.2.4.105 AArch64_S1TTWParamsEL3

```
// AArch64_S1TTWParamsEL3()
// ========================
// Gather stage 1 translation table walk parameters for EL3 regime

func AArch64_S1TTWParamsEL3() => S1TTWParams
begin
    var walkparams : S1TTWParams;

    walkparams.tgx  = AArch64_S1DecodeTG0(TCR_EL3().TG0);
    walkparams.txsz = TCR_EL3().T0SZ;
    walkparams.ps   = TCR_EL3().PS;
    walkparams.irgn = TCR_EL3().IRGN0;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkparams.orgn = TCR_EL3().ORGN0;
    walkparams.sh   = TCR_EL3().SH0;
    walkparams.tbi  = TCR_EL3().TBI;
    walkparams.mair = MAIR_EL3() as MAIRType;
    walkparams.d128  = if IsFeatureImplemented(FEAT_D128)  then TCR_EL3().D128 else '0';
    walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL3().SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR_EL3().DisCH0 else '0';

    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if IsFeatureImplemented(FEAT_S1PIE) then TCR_EL3().PIE else '0';
    end;
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL3() as S1PIRType;
    end;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL3() as MAIRType;
    end;
    walkparams.aie  = if IsFeatureImplemented(FEAT_AIE) then TCR_EL3().AIE else '0';
    walkparams.wxn  = SCTLR_EL3().WXN;
    walkparams.ee   = SCTLR_EL3().EE;
    walkparams.sif = (if !IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2)
                      then SCR_EL3().SIF else '0');

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth)  then TCR_EL3().TBID else '0';
    walkparams.hpd  = if IsFeatureImplemented(FEAT_HPDS)   then TCR_EL3().HPD else '0';
    if walkparams.hpd == '0' then
        if walkparams.aie == '1' then walkparams.hpd = '1'; end;
        if walkparams.pie == '1' then walkparams.hpd = '1'; end;
        if AArch64_S1POEnabled(Regime_EL3) then walkparams.hpd = '1'; end;
    end;
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL3().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = TCR_EL3().HD;
    else
        walkparams.hd = '0';
    end;
    if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
          walkparams.d128 == '0') then
        walkparams.ds = TCR_EL3().DS;
    else
        walkparams.ds = '0';
    end;
    if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
          IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
        walkparams.mtx  = TCR_EL3().MTX;
    else
        walkparams.mtx  = '0';
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' then
        walkparams.pnch = TCR_EL3().PnCH;
    else
        walkparams.pnch = '0';
    end;
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3().HAFT;
    else
        walkparams.haft = '0';
    end;
    walkparams.emec = if IsFeatureImplemented(FEAT_MEC) then SCTLR2_EL3().EMEC else '0';

    return walkparams;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.106 AArch64_S2DecodeTG0

```
// AArch64_S2DecodeTG0()
// ======================
// Decode stage 2 granule size configuration bits TG0

func AArch64_S2DecodeTG0(tg0_in : bits(2)) => TGx
begin
    var tg0 : bits(2) = tg0_in;
    var tgx : TGx;

    if tg0 == '11' then
        tg0 = ImpDefBits{2}("TG0 encoded granule size");
    end;

    case tg0 of
        when '00' =>   tgx = TGx_4KB;
        when '01' =>   tgx = TGx_64KB;
        when '10' =>   tgx = TGx_16KB;
    end;

    if !AArch64_HaveS2TG(tgx) then
        case ImpDefBits{2}("TG0 encoded granule size") of
            when '00' =>   tgx = TGx_4KB;
            when '01' =>   tgx = TGx_64KB;
            when '10' =>   tgx = TGx_16KB;
        end;
    end;

    return tgx;
end;
```

J1.2.4.107 AArch64_S2MinTxSZ

```
// AArch64_S2MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

func AArch64_S2MinTxSZ(walkparams : S2TTWParams, s1aarch64 : boolean) => integer
begin
    var ips : integer;

    if AArch64_PAMax() == 56 then
        if walkparams.d128 == '1' then
            ips = 56;
        elsif walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            ips = 52;
        else
            ips = 48;
        end;
    elsif AArch64_PAMax() == 52 then
        if walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            ips = 52;
        else
            ips = 48;
        end;
    else
        ips = AArch64_PAMax();
    end;

    var min_txsz : integer = 64 - ips;
    if !s1aarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return min_txsz;
end;
```

J1.2.4.108 AArch64_SS2TTWParams

```
// AArch64_SS2TTWParams()
// ======================
// Gather walk parameters specific for secure stage 2 translation

func AArch64_SS2TTWParams(ipaspace : PASpace, s1aarch64 : boolean) => S2TTWParams
begin
    var walkparams : S2TTWParams;

    walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2().D128 else '0';
    if ipaspace == PAS_Secure then
        walkparams.tgx  = AArch64_S2DecodeTG0(VSTCR_EL2().TG0);
        walkparams.txsz = VSTCR_EL2().T0SZ;
        if walkparams.d128 == '1' then
            walkparams.skl = VSTTBR_EL2().SKL;
        else
            walkparams.sl0 = VSTCR_EL2().SL0;
        end;
        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VSTCR_EL2().SL2 AND VTCR_EL2().DS;
        else
            walkparams.sl2 = '0';
        end;
    elsif ipaspace == PAS_NonSecure then
        walkparams.tgx  = AArch64_S2DecodeTG0(VTCR_EL2().TG0);
        walkparams.txsz = VTCR_EL2().T0SZ;
        if walkparams.d128 == '1' then
            walkparams.skl = VTTBR_EL2().SKL;
        else
            walkparams.sl0 = VTCR_EL2().SL0;
        end;
        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2().SL2 AND VTCR_EL2().DS;
        else
            walkparams.sl2 = '0';
        end;
    else
        unreachable;
    end;

    walkparams.sw   = VSTCR_EL2().SW;
    walkparams.nsw  = VTCR_EL2().NSW;
    walkparams.sa   = VSTCR_EL2().SA;
    walkparams.nsa  = VTCR_EL2().NSA;
    walkparams.vm   = HCR_EL2().VM OR HCR_EL2().DC;
    walkparams.ps   = VTCR_EL2().PS;
    walkparams.irgn = VTCR_EL2().IRGN0;
    walkparams.orgn = VTCR_EL2().ORGN0;
    walkparams.sh   = VTCR_EL2().SH0;
    walkparams.ee   = SCTLR_EL2().EE;

    walkparams.ptw = if HCR_EL2().TGE == '0'                then HCR_EL2().PTW else '0';
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB)  then HCR_EL2().FWB else '0';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then VTCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = VTCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2().DS;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        walkparams.ds = '0';
    end;
    walkparams.cmow = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2Enabled() then HCRX_EL2().CMOW
                       else '0');
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2().S2PIE else '0';
    end;
    if IsFeatureImplemented(FEAT_S2PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
               ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.s2pir = S2PIR_EL2() as S2PIRType;
        else
            walkparams.s2pir = Zeros{64} as S2PIRType;
        end;
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2().AssuredOnly;
    else
        walkparams.assuredonly = '0';
    end;
    walkparams.tl0   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL0 else '0';
    walkparams.tl1   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL1 else '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;
    walkparams.emec = '0';
    if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
          (!HaveEL(EL3) || SCR_EL3().HDBSSEn == '1')) then
        walkparams.hdbss = VTCR_EL2().HDBSS;
    else
        walkparams.hdbss = '0';
    end;

    return walkparams;
end;
```

J1.2.4.109 S2DCacheEnabled

```
// S2DCacheEnabled()
// =================
// Returns TRUE if Stage 2 Data access cacheability is enabled

func S2DCacheEnabled() => boolean
begin
    return HCR_EL2().CD == '0';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions that are listed in this section are identified as `AArch32.FunctionName`. Some of these functions have an equivalent AArch64 function, `AArch64.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch32/debug/breakpoint`.

Note

Many AArch32 pseudocode functions have not been updated to show the constraints on the Armv7 UNPREDICTABLE behaviors that are described in Architectural Constraints on UNPREDICTABLE Behaviors. Where AArch32 pseudocode shows something to be UNPREDICTABLE, check Architectural Constraints on UNPREDICTABLE Behaviors for possible constraints on the permitted behavior.

The top-level sections of the AArch32 pseudocode hierarchy are:

- aarch32/debug.
- aarch32/exceptions.
- aarch32/functions.
- aarch32/translation.

J1.3.1 aarch32/debug

This section includes the following pseudocode functions:

- AArch32_VCRMatch
- AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled
- AArch32_BreakpointMatch
- AArch32_BreakpointValueMatch
- AArch32_ReservedBreakpointType
- AArch32_StateMatch
- AArch32_GenerateDebugExceptions
- AArch32_GenerateDebugExceptionsFrom
- AArch32_IncrementCycleCounter
- AArch32_IncrementEventCounter
- AArch32_PMUCycle
- AArch32_EnterHypModeInDebugState
- AArch32_EnterModeInDebugState
- AArch32_EnterMonitorModeInDebugState
- AArch32_WatchpointByteMatch
- AArch32_WatchpointMatch

J1.3.1.1 AArch32_VCRMatch

```
// AArch32_VCRMatch()
// ==================

func AArch32_VCRMatch(vaddress : bits(32)) => boolean
begin

    var match : boolean;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        var match_word : bits(32) = Zeros{};

        let ss : SecurityState = CurrentSecurityState();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if vaddress[31:5] == ExcVectorBase()[31:5] then
            if HaveEL(EL3) && ss == SS_NonSecure then
                match_word[UInt(vaddress[4:2]) + 24] = '1';     // Non-secure vectors
            else
                match_word[UInt(vaddress[4:2]) + 0] = '1';      // Secure vectors (or no EL3)
            end;
        end;
        if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress[31:5] == MVBAR()[31:5] &&
            ss == SS_Secure) then
            match_word[UInt(vaddress[4:2]) + 8] = '1';          // Monitor vectors
        end;
        // Mask out bits not corresponding to vectors.
        var mask : bits(32);
        if !HaveEL(EL3) then
            mask = '00000000'::'00000000'::'00000000'::'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110'::'00000000'::'00000000'::'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110'::'00000000'::'11011100'::'11011110';
        end;

        match_word = match_word AND DBGVCR() AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word[28:27,12:11,4:3]) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
        end;

        if !IsZero(vaddress[1:0]) && match then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
        end;
    else
        match = FALSE;
    end;

    return match;
end;
```

J1.3.1.2 AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

func AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled() => boolean
begin
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE; end;
    return DBGEN == HIGH && SPIDEN == HIGH;
end;
```

J1.3.1.3 AArch32_BreakpointMatch

```
// AArch32_BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

func AArch32_BreakpointMatch(n : integer, vaddress : bits(32), accdesc : AccessDescriptor,
                             size : integer) => BreakpointInfo
begin
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var brkptinfo : BreakpointInfo;
    let enabled : boolean    = DBGBCR(n).E == '1';
    let isbreakpnt : boolean = TRUE;
    let linked : boolean     = DBGBCR(n).BT == '0x01';
    let linked_to : boolean  = FALSE;
    let linked_n : integer{} = UInt(DBGBCR(n).LBN);

    let state_match : boolean = AArch32_StateMatch(DBGBCR(n).SSC, DBGBCR(n).HMC, DBGBCR(n).PMC,
                                                   linked, linked_n, isbreakpnt,  accdesc);
    var (value_match, value_mismatch) = AArch32_BreakpointValueMatch(n,vaddress, linked_to);

    if size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether this breakpoint generates a Breakpoint debug
        // event.
        let (match_i, mismatch_i) = AArch32_BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        end;

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
        end;
    end;

    if vaddress[1] == '1' && DBGBCR(n).BAS == '1111' then
        // The above notwithstanding, if DBGBCR(n).BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR(n)+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        end;

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
        end;
    end;

    brkptinfo.match    = value_match && state_match && enabled;
    brkptinfo.mismatch = value_mismatch && state_match && enabled;

    return brkptinfo;
end;
```

J1.3.1.4 AArch32_BreakpointValueMatch

```
// AArch32_BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

func AArch32_BreakpointValueMatch(n_in : integer, vaddress : bits(32),
                                  linked_to : boolean) => (boolean, boolean)
begin

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    var n : integer = n_in;
    var c : Constraint;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplemented() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                           Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE, FALSE); end;
end;

// If this breakpoint is not enabled, it cannot generate a match.
// (This could also happen on a call from StateMatch for linking).
if DBGBCR(n).E == '0' then return (FALSE, FALSE); end;

var dbgtype : bits(4) = DBGBCR(n).BT;

(c, dbgtype) = AArch32_ReservedBreakpointType(n, dbgtype);
if c == Constraint_DISABLED then return (FALSE, FALSE); end;
// Otherwise the dbgtype value returned by AArch32_ReservedBreakpointType is valid.

// Determine what to compare against.
let match_addr : boolean      = (dbgtype == '0x0x');
let mismatch : boolean        = (dbgtype == '010x');
let match_vmid : boolean      = (dbgtype == '10xx');
let match_cid1 : boolean      = (dbgtype == 'xx1x');
let match_cid2 : boolean      = (dbgtype == '11xx');
let linking_enabled : boolean = (dbgtype == 'xxx1');

// If called from StateMatch, is CONSTRAINED UNPREDICTABLE if the
// breakpoint is not programmed with linking enabled.
if linked_to && !linking_enabled then
    if !ConstrainUnpredictableBool(Unpredictable_BPLINKINGDISABLED) then
        return (FALSE, FALSE);
    end;
end;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linking_enabled && !match_addr then
    return (FALSE, FALSE);
end;

var bvr_match : boolean  = FALSE;
var bxvr_match : boolean = FALSE;

// Do the comparison.
if match_addr then
    let byte : integer = UInt(vaddress[1:0]);
    assert byte IN {0,2};                     // "vaddress" is halfword aligned

    let byte_select_match : boolean = (DBGBCR(n).BAS[byte] == '1');
    bvr_match = (vaddress[31:2] == DBGBVR(n)[31:2]) && byte_select_match;

elsif match_cid1 then
    bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR() == DBGBVR(n)[31:0]);
end;

if match_vmid then
    let vmid : bits(NUM_VMIDBITS) = VMID();
    var bvr_vmid : bits(NUM_VMIDBITS);

    if ELUsingAArch32(EL2) then
        bvr_vmid = ZeroExtend{NUM_VMIDBITS}(DBGBXVR(n)[7:0]);
    elsif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2().VS == '0' then
        bvr_vmid = ZeroExtend{NUM_VMIDBITS}(DBGBXVR(n)[7:0]);
    else
        bvr_vmid = DBGBXVR(n)[15:0];
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32(EL2) &&
                      DBGBXVR(n)[31:0] == CONTEXTIDR_EL2()[31:0]);
    end;

    let bvr_match_valid : boolean  = (match_addr || match_cid1);
    let bxvr_match_valid : boolean = (match_vmid || match_cid2);

    let match : boolean = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return (match && !mismatch, !match && mismatch);
end;
```

J1.3.1.5 AArch32_ReservedBreakpointType

```
// AArch32_ReservedBreakpointType()
// ================================
// Checks if the given DBGBCR(n).BT value is reserved and will generate Constrained Unpredictable
// behavior, otherwise returns Constraint_NONE.

func AArch32_ReservedBreakpointType(n : integer, bt_in : bits(4)) => (Constraint, bits(4))
begin
    var bt : bits(4)       = bt_in;
    var reserved : boolean = FALSE;
    let context_aware : boolean = IsContextAwareBreakpoint(n);

    // Address mismatch
    if bt == '010x' && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;
    end;

    // Context matching
    if bt != '0x0x' && !context_aware then
        reserved = TRUE;
    end;

    // EL2 extension
    if bt == '1xxx' && !HaveEL(EL2) then
        reserved = TRUE;
    end;

    // Context matching
    if (bt IN {'011x','11xx'} && !IsFeatureImplemented(FEAT_VHE) &&
          !IsFeatureImplemented(FEAT_Debugv8p2)) then
        reserved = TRUE;
    end;

    if reserved then
        var c : Constraint;
        (c, bt) = ConstrainUnpredictableBits{4}(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, ARBITRARY : bits(4));
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
        end;
    end;

    return (Constraint_NONE, bt);
end;
```

J1.3.1.6 AArch32_StateMatch

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32_StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

func AArch32_StateMatch(ssc_in : bits(2),  hmc_in : bit, pxc_in : bits(2), linked_in : boolean,
                        linked_n_in : integer, isbreakpnt : boolean,
                        accdesc : AccessDescriptor) => boolean
begin

    // "ssc_in","hmc_in","pxc_in" are the control fields from the DBGBCR(n) or DBGWCR(n) register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR(n) or DBGWCR(n) register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "accdesc" describes the properties of the access being matched.
    var hmc : bit          = hmc_in;
    var ssc : bits(2)      = ssc_in;
    var pxc : bits(2)      = pxc_in;
    var linked : boolean   = linked_in;
    var linked_n : integer = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    var c : Constraint;
    // SSCE value discarded as there is no SSCE bit in AArch32_
    (c, ssc, -, hmc, pxc) = CheckValidStateMatch(ssc, '0', hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE; end;
    // Otherwise the hmc,ssc,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    let pl2_match : boolean = HaveEL(EL2) && ((hmc == '1' && (ssc::pxc != '1000')) || ssc == '11');
    let pl1_match : boolean = pxc[0] == '1';
    let pl0_match : boolean = pxc[1] == '1';
    let ssu_match : boolean = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

    var priv_match : boolean;
    if ssu_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case accdesc.el of
            when EL3 => priv_match = pl1_match;           // EL3 and EL1 are both PL1
            when EL2 => priv_match = pl2_match;
            when EL1 => priv_match = pl1_match;
            when EL0 => priv_match = pl0_match;
        end;
    end;

    // Security state match
    var ss_match : boolean;
    case ssc of
        when '00' => ss_match = TRUE;                                    // Both
        when '01' => ss_match = accdesc.ss == SS_NonSecure;              // Non-secure only
        when '10' => ss_match = accdesc.ss == SS_Secure;                 // Secure only
        when '11' => ss_match = (hmc == '1' || accdesc.ss == SS_Secure); // HMC=1 -> Both,
                                                                         // HMC=0 -> Secure only
    end;

    var linked_match : boolean = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit.
        // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE whether
        // this gives no match, gives a match without linking, or linked_n is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        if !IsContextAwareBreakpoint(linked_n) then
            let (first_ctx_cmp, last_ctx_cmp) : (integer, integer) = ContextAwareBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                        Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

            case c of
                when Constraint_DISABLED =>  return FALSE;      // Disabled
                when Constraint_NONE =>      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
            end;
        end;

        let vaddress : bits(32) = ARBITRARY : bits(32);
        let linked_to : boolean = TRUE;
        (linked_match,-) = AArch32_BreakpointValueMatch(linked_n, vaddress, linked_to);
    end;

    return priv_match && ss_match && (!linked || linked_match);
end;
```

J1.3.1.7 AArch32_GenerateDebugExceptions

```
// AArch32_GenerateDebugExceptions()
// =================================

func AArch32_GenerateDebugExceptions() => boolean
begin
    let ss : SecurityState = CurrentSecurityState();
    return AArch32_GenerateDebugExceptionsFrom(PSTATE.EL, ss);
end;
```

J1.3.1.8 AArch32_GenerateDebugExceptionsFrom

```
// AArch32_GenerateDebugExceptionsFrom()
// =====================================

func AArch32_GenerateDebugExceptionsFrom(from_el : bits(2), from_state : SecurityState) => boolean
begin
    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        let mask : bit = '0';    // No PSTATE.D in AArch32 state
        return AArch64_GenerateDebugExceptionsFrom(from_el, from_state, mask);
    end;
    if DBGOSLSR().OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    end;

    var enabled : boolean;
    if HaveEL(EL3) && from_state == SS_Secure then
        assert from_el != EL2;    // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3().SDD == '0';
        else
            let spd : bits(2) = if ELUsingAArch32(EL3) then SDCR().SPD else MDCR_EL3().SPD32;
            if spd[1] == '1' then
                enabled = spd[0] == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled();
            end;
        end;
        if from_el == EL0 then enabled = enabled || SDER().SUIDEN == '1'; end;
    else
        enabled = from_el != EL2;
    end;

    return enabled;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.3.1.9 AArch32_IncrementCycleCounter

```
// AArch32_IncrementCycleCounter()
// ==============================
// Increment the cycle counter and possibly set overflow bits.

func AArch32_IncrementCycleCounter()
begin
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return; end;
    var d : bit = PMCR().D;   // Check divide-by-64
    var lc : bit = PMCR().LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0'; end;
    if d == '1' && !HasElapsed64Cycles() then return; end;

    let old_value : integer = UInt(PMCCNTR());
    let new_value : integer = old_value + 1;
    PMCCNTR() = new_value[63:0];

    let ovflw : integer{} = if lc == '1' then 64 else 32;

    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET().C = '1';
    end;

    return;
end;
```

J1.3.1.10 AArch32_IncrementEventCounter

```
// AArch32_IncrementEventCounter()
// ==============================
// Increment the specified event counter 'idx' by the specified amount 'increment'.
// 'Vm' is the value event counter 'idx-1' is being incremented by if 'idx' is odd,
// zero otherwise.
// Returns the amount the counter was incremented by.

func AArch32_IncrementEventCounter(idx : integer, increment_in : integer, Vm : integer) => integer
begin
    if HaveAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        return AArch64_IncrementEventCounter(idx, increment_in, Vm);
    end;

    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    var old_value : integer;
    var new_value : integer;

    old_value = UInt(PMEVCNTR(idx));
    let increment : integer = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    PMEVCNTR(idx) = new_value[31:0];
    let ovflw : integer{} = 32;
    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET()[idx] = '1';
        // Check for the CHAIN event from an even counter
        if (idx[0] == '0' && idx + 1 < NUM_PMU_COUNTERS &&
              (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
                ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
        end;
    end;

    return increment;
end;
```

J1.3.1.11 AArch32_PMUCycle

```
// AArch32_PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

func AArch32_PMUCycle()
begin
    if HaveAArch64() then
        AArch64_PMUCycle();
        return;
    end;

    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    PMUEvent(PMU_EVENT_CPU_CYCLES);
    let counters : integer = NUM_PMU_COUNTERS;
    var Vm : integer = 0;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            if CountPMUEvents(idx) then
                let accumulated : integer = PMUEventAccumulator[[idx]];
                if (idx MOD 2) == 0 then Vm = 0; end;
                Vm = AArch32_IncrementEventCounter(idx, accumulated, Vm);
            end;
            PMUEventAccumulator[[idx]] = 0;
        end;
    end;
    AArch32_IncrementCycleCounter();
    CheckForPMUOverflow();
end;
```

J1.3.1.12 AArch32_EnterHypModeInDebugState

```
// AArch32_EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

noreturn func AArch32_EnterHypModeInDebugState(except : ExceptionRecord)
begin
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32_ReportHypEntry(except);
    AArch32_WriteMode(M32_Hyp);
    SPSR_curr() = ARBITRARY : bits(32);
    ELR_hyp()   = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.[SS,A,I,F] = ARBITRARY : bits(4);
    DLR()   = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2() = ARBITRARY : bits(32);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    PSTATE.E  = HSCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    EDSCR().ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.3.1.13 AArch32_EnterModeInDebugState

```
// AArch32_EnterModeInDebugState()
// ==============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

noreturn func AArch32_EnterModeInDebugState(target_mode : bits(5))
begin
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    AArch32_WriteMode(target_mode);
    SPSR_curr() = ARBITRARY : bits(32);
    R(14)       = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.[SS,A,I,F] = ARBITRARY : bits(4);
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2() = ARBITRARY : bits(32);
    end;
    PSTATE.E  = SCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLR().SPAN == '0' then PSTATE.PAN = '1'; end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    EDSCR().ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.3.1.14 AArch32_EnterMonitorModeInDebugState

```
// AArch32_EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

func AArch32_EnterMonitorModeInDebugState()
begin
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    let from_secure : boolean = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    AArch32_WriteMode(M32_Monitor);
    SPSR_curr() = ARBITRARY : bits(32);
    R(14) = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.[SS,A,I,F] = ARBITRARY : bits(4);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.E = SCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR().SPAN == '0' then
            PSTATE.PAN = '1';
        end;
    end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2() = ARBITRARY : bits(32);
    end;
    EDSCR().ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.3.1.15 AArch32_WatchpointByteMatch

```
// AArch32_WatchpointByteMatch()
// ============================

func AArch32_WatchpointByteMatch(n : integer, vaddress : bits(32)) => boolean
begin
    let dbgtop : integer{} = 31;
    let cmpbottom : integer{} = if DBGWVR(n)[2] == '1' then 2 else 3; // Word or doubleword
    var bottom = cmpbottom;
    let select : integer{} = UInt(vaddress[cmpbottom-1:0]);
    var byte_select_match : boolean = (DBGWCR(n).BAS[select] != '0');
    var mask : integer{} = UInt(DBGWCR(n).MASK);

    // If DBGWCR(n).MASK is a nonzero value and DBGWCR(n).BAS is not set to '11111111', or
    // DBGWCR(n).BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR(n).BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        let LSB : bits(8) = (DBGWCR(n).BAS AND NOT(DBGWCR(n).BAS - 1));
        let MSB : bits(8) = (DBGWCR(n).BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then          // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                             // For the whole doubleword
        end;
    end;

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        var c : Constraint;
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED => return FALSE; // Disabled
            when Constraint_NONE     => mask = 0;     // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
            otherwise                => mask = unpred_mask as integer{3..31};
        end;
    end;

    let cmpmsb : integer{}    = dbgtop;
    let cmplsb : integer{}    = if mask > bottom then mask else bottom;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let bottombit : integer{} = bottom;
    var WVR_match : boolean   = (vaddress[cmpmsb:cmplsb] == DBGWVR(n)[cmpmsb:cmplsb]);
    if mask > bottom then
        // If masked bits of DBGWVR(n) are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR(n)[cmplsb-1:bottombit]) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
        end;
    end;

    return (WVR_match && byte_select_match);
end;
```

J1.3.1.16 AArch32_WatchpointMatch

```
// AArch32_WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

func AArch32_WatchpointMatch(n : integer, vaddress : bits(32), size : integer,
                             accdesc : AccessDescriptor) => WatchpointInfo
begin
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    let enabled : boolean     = DBGWCR(n).E == '1';
    let linked : boolean      = DBGWCR(n).WT == '1';
    let isbreakpnt : boolean  = FALSE;
    let linked_n : integer{}  = UInt(DBGWCR_EL1(n).LBN);
    let state_match : boolean = AArch32_StateMatch(DBGWCR(n).SSC, DBGWCR(n).HMC, DBGWCR(n).PAC,
                                                   linked, linked_n, isbreakpnt, accdesc);

    var watchptinfo : WatchpointInfo;
    var ls_match : boolean;
    case DBGWCR(n).LSC[1:0] of
        when '00' => ls_match = FALSE;
        when '01' => ls_match = accdesc.read;
        when '10' => ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' => ls_match = TRUE;
    end;

    var value_match : boolean = FALSE;
    watchptinfo.vaddress = ZeroExtend{64}(vaddress);
    for byte = 0 to size - 1 do
        if (!value_match &&
              !AddressInNaturallyAlignedBlock(watchptinfo.vaddress,
                                              ZeroExtend{64}(vaddress + byte))) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = ZeroExtend{64}(vaddress + byte);
        end;
        value_match = value_match || AArch32_WatchpointByteMatch(n, vaddress + byte);
    end;

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match && state_match && ls_match && enabled;
    return watchptinfo;
end;
```

J1.3.2 aarch32/exceptions

This section includes the following pseudocode functions:

- AArch32_Abort

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32_AbortSyndrome
- AArch32_CheckPCAlignment
- AArch32_CommonFaultStatus
- AArch32_ReportDataAbort
- AArch32_ReportPrefetchAbort
- AArch32_TakeDataAbortException
- AArch32_TakePrefetchAbortException
- AArch32_TakePhysicalFIQException
- AArch32_TakePhysicalIRQException
- AArch32_TakePhysicalSErrorException
- AArch32_TakeVirtualFIQException
- AArch32_TakeVirtualIRQException
- AArch32_TakeVirtualSErrorException
- AArch32_SoftwareBreakpoint
- DebugException
- AArch32_CheckAdvSIMDOrFPRegisterTraps
- AArch32_ExceptionClass
- AArch32_GeneralExceptionsToAArch64
- AArch32_ReportHypEntry
- AArch32_ResetControlRegisters
- AArch32_TakeReset
- ExcVectorBase
- AArch32_FPTrappedException
- AArch32_CallHypervisor
- AArch32_CallSupervisor
- AArch32_TakeHVCException
- AArch32_TakeSMCException
- AArch32_TakeSVCException
- AArch32_EnterHypMode
- AArch32_EnterMode
- AArch32_EnterMonitorMode
- AArch32_CheckAdvSIMDOrFPEnabled
- AArch32_CheckFPAdvSIMDTrap
- AArch32_CheckForSMCUndefOrTrap
- AArch32_CheckForSVCTrap
- AArch32_CheckForWFxTrap
- AArch32_CheckITEnabled
- AArch32_CheckIllegalState
- AArch32_CheckSETENDEnabled
- AArch32_SystemAccessTrap
- AArch32_SystemAccessTrapSyndrome
- AArch32_TakeHypTrapException
- AArch32_TakeMonitorTrapException
- AArch32_TakeUndefInstrException
- AArch32_Undefined

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.1 AArch32_Abort

```
// AArch32_Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

func AArch32_Abort(fault : FaultRecord)
begin
    // Check if routed to AArch64 state
    var route_to_aarch64 : boolean = ((IsExternalAbort(fault) &&
                                      !ELUsingAArch32(SyncExternalAbortTarget(fault))) ||
                                      (PSTATE.EL == EL0 && !ELUsingAArch32(EL1)));

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2().TGE == '1' || IsSecondStage(fault) ||
                            (IsDebugException(fault) && MDCR_EL2().TDE == '1'));
    end;

    if route_to_aarch64 then
        AArch64_Abort(fault);
    end;
    if fault.accessdesc.acctype == AccessType_IFETCH then
        AArch32_TakePrefetchAbortException(fault);
    else
        AArch32_TakeDataAbortException(fault);
    end;
end;
```

J1.3.2.2 AArch32_AbortSyndrome

```
// AArch32_AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions
// taken to Hyp mode
// from an AArch32 translation regime.
func AArch32_AbortSyndrome(exceptype : Exception, fault : FaultRecord,
                           target_el : bits(2)) => ExceptionRecord
begin
    var except : ExceptionRecord = ExceptionSyndrome(exceptype);

    except.syndrome.iss = AArch32_FaultSyndrome(exceptype, fault);
    if exceptype == Exception_Watchpoint then
        except.vaddress = fault.watchptinfo.vaddress;
    else
        except.vaddress = ZeroExtend{64}(fault.vaddress);
    end;

    if IPAValid(fault) then
        except.ipavalid  = TRUE;
        except.NS        = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = ZeroExtend{NUM_PABITS}(fault.ipaddress.address);
    else
        except.ipavalid = FALSE;
    end;

    return except;
end;
```

J1.3.2.3 AArch32_CheckPCAlignment

```
// AArch32_CheckPCAlignment()
// ==========================

func AArch32_CheckPCAlignment()
begin
    let pc : bits(32) = ThisInstrAddr{}();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if (CurrentInstrSet() == InstrSet_A32 && pc[1] == '1') || pc[0] == '1' then
        if AArch32_GeneralExceptionsToAArch64() then AArch64_PCAlignmentFault(); end;
        let accdesc : AccessDescriptor = CreateAccDescIFetch();
        var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(pc));
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32_Abort(fault);
    end;
end;
```

J1.3.2.4 AArch32_CommonFaultStatus

```
// AArch32_CommonFaultStatus()
// ===========================
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.

func AArch32_CommonFaultStatus(fault : FaultRecord, long_format : boolean) => bits(32)
begin
    var target : bits(32) = Zeros{};
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        let errstate : ErrorState = PEErrorState(fault);
        target[15:14] = AArch32_EncodeAsyncErrorSyndrome(errstate);   // AET
    end;
    if IsExternalAbort(fault) then target[12] = fault.extflag; end;   // ExT
    target[9] = if long_format then '1' else '0';                     // LPAE
    if long_format then                                               // Long-descriptor format
        target[5:0]    = EncodeLDFSC(fault.statuscode, fault.level);  // STATUS
    else                                                              // Short-descriptor format
        target[10,3:0] = EncodeSDFSC(fault.statuscode, fault.level);  // FS
    end;
    return target;
end;
```

J1.3.2.5 AArch32_ReportDataAbort

```
// AArch32_ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

func AArch32_ReportDataAbort(route_to_monitor : boolean, fault : FaultRecord)
begin
    var long_format : boolean;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = ((TTBCR_S().EAE == '1') ||
                       (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR().EAE == '1') ||
                        (fault.secondstage && ImpDefBool(
                                              "Report abort using Long-descriptor format")))));
    else
        long_format = TTBCR().EAE == '1';
    end;
    var syndrome : bits(32) = AArch32_CommonFaultStatus(fault, long_format);

    // bits of syndrome that are not common to I and D side
    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        syndrome[13] = '1';                              // CM
        syndrome[11] = '1';                              // WnR
    else
        syndrome[11] = if fault.write then '1' else '0'; // WnR
    end;

    if !long_format then
        syndrome[7:4] = fault.domain;                    // Domain
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if fault.accessdesc.acctype == AccessType_IC then
        var i_syndrome : bits(32);
        if (!long_format &&
            ImpDefBool("Report I-cache maintenance fault in IFSR")) then
            i_syndrome = syndrome;
            syndrome[10,3:0] = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = ARBITRARY : bits(32);
        end;
        if route_to_monitor then
            IFSR_S() = i_syndrome;
        else
            IFSR() = i_syndrome;
        end;
    end;

    if route_to_monitor then
        DFSR_S() = syndrome;
        DFAR_S() = fault.vaddress[31:0];
    else
        DFSR()   = syndrome;
        DFAR()   = fault.vaddress[31:0];
    end;
end;
```

J1.3.2.6 AArch32_ReportPrefetchAbort

```
// AArch32_ReportPrefetchAbort()
// ============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

func AArch32_ReportPrefetchAbort(route_to_monitor : boolean, fault : FaultRecord)
begin
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    var long_format : boolean = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = TTBCR_S().EAE == '1' || PSTATE.EL == EL2 || TTBCR().EAE == '1';
    else
        long_format = TTBCR().EAE == '1';
    end;

    let fsr : bits(32) = AArch32_CommonFaultStatus(fault, long_format);
    if route_to_monitor then
        IFSR_S() = fsr;
        IFAR_S() = fault.vaddress[31:0];
    else
        IFSR() = fsr;
        IFAR() = fault.vaddress[31:0];
    end;
    return;
end;
```

J1.3.2.7 AArch32_TakeDataAbortException

```
// AArch32_TakeDataAbortException()
// ================================

func AArch32_TakeDataAbortException(fault : FaultRecord)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin

    var sea_target : bits(2);
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = ARBITRARY : bits(2);
    end;

    let route_to_monitor : boolean = IsExternalAbort(fault) && sea_target == EL3;
    let route_to_hyp : boolean     = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                                      (HCR().TGE == '1' ||
                                      (IsExternalAbort(fault) && sea_target == EL2) ||
                                      (IsDebugException(fault) && HDCR().TDE == '1') ||
                                       IsSecondStage(fault)));

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let lr_offset : integer = 8;

    if IsDebugException(fault) then DBGDSCRext().MOE = fault.debugmoe; end;
    if route_to_monitor then
        AArch32_ReportDataAbort(route_to_monitor, fault);
        AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = AArch32_AbortSyndrome(Exception_DataAbort, fault, EL2);
        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32_ReportDataAbort(route_to_monitor, fault);
        AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

J1.3.2.8 AArch32_TakePrefetchAbortException

```
// AArch32_TakePrefetchAbortException()
// ====================================

func AArch32_TakePrefetchAbortException(fault : FaultRecord)
begin

    var sea_target : bits(2);
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = ARBITRARY : bits(2);
    end;

    let route_to_monitor : boolean = IsExternalAbort(fault) && sea_target == EL3;
    let route_to_hyp : boolean = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                                  (HCR().TGE == '1' ||
                                  (IsExternalAbort(fault) && sea_target == EL2) ||
                                  (IsDebugException(fault) && HDCR().TDE == '1') ||
                                   IsSecondStage(fault)));

    var except : ExceptionRecord;
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0C;
    let lr_offset : integer = 4;

    if IsDebugException(fault) then DBGDSCRext().MOE = fault.debugmoe; end;
    if route_to_monitor then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_ReportPrefetchAbort(route_to_monitor, fault);
        AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then             // PC Alignment fault
            except = ExceptionSyndrome(Exception_PCAlignment);
            except.vaddress = ThisInstrAddr{64}();
        else
            except = AArch32_AbortSyndrome(Exception_InstructionAbort, fault, EL2);
        end;
        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32_ReportPrefetchAbort(route_to_monitor, fault);
        AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

J1.3.2.9 AArch32_TakePhysicalFIQException

```
// AArch32_TakePhysicalFIQException()
// ==================================

func AArch32_TakePhysicalFIQException()
begin

    // Check if routed to AArch64 state
    var route_to_aarch64 : boolean = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2().TGE == '1' || (HCR_EL2().FMO == '1' && !IsInHost());
    end;

    if !route_to_aarch64 then
        route_to_aarch64 = EffectiveSCR_EL3_FIQ() == '1';
    end;

    if route_to_aarch64 then AArch64_TakePhysicalFIQException(); end;
    let route_to_monitor : boolean = HaveEL(EL3) && SCR().FIQ == '1';
    let route_to_hyp : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR().TGE == '1' || HCR().FMO == '1'));
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x1C;
    let lr_offset : integer   = 4;
    if route_to_monitor then
        AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = ExceptionSyndrome(Exception_FIQ);
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

J1.3.2.10 AArch32_TakePhysicalIRQException

```
// AArch32_TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

func AArch32_TakePhysicalIRQException()
begin

    // Check if routed to AArch64 state
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var route_to_aarch64 : boolean = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2().TGE == '1' || (HCR_EL2().IMO == '1' && !IsInHost());
    end;
    if !route_to_aarch64 then
        route_to_aarch64 = EffectiveSCR_EL3_IRQ() == '1';
    end;

    if route_to_aarch64 then AArch64_TakePhysicalIRQException(); end;

    let route_to_monitor : boolean = HaveEL(EL3) && SCR().IRQ == '1';
    let route_to_hyp : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                  (HCR().TGE == '1' || HCR().IMO == '1'));
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x18;
    let lr_offset : integer   = 4;
    if route_to_monitor then
        AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = ExceptionSyndrome(Exception_IRQ);
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

J1.3.2.11 AArch32_TakePhysicalSErrorException

```
// AArch32_TakePhysicalSErrorException()
// =====================================

func AArch32_TakePhysicalSErrorException(implicit_esb : boolean)
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    // Check if routed to AArch64 state
    if !ELUsingAArch32(target_el) then
        AArch64_TakePhysicalSErrorException(implicit_esb);
    end;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let fault : FaultRecord   = GetPendingPhysicalSError();
    let lr_offset : integer   = 8;

    let except : ExceptionRecord   = AArch32_AbortSyndrome(Exception_DataAbort, fault, target_el);
    let route_to_monitor : boolean = (target_el == EL3);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    end;

    case target_el of
        when EL3 =>
            AArch32_ReportDataAbort(route_to_monitor, fault);
            AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2 =>
            if PSTATE.EL == EL2 then
                AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL1 =>
            AArch32_ReportDataAbort(route_to_monitor, fault);
            AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        otherwise =>
            unreachable;
    end;
end;
```

J1.3.2.12 AArch32_TakeVirtualFIQException

```
// AArch32_TakeVirtualFIQException()
// ================================

func AArch32_TakeVirtualFIQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR().TGE == '0' && HCR().FMO == '1';
    else
        assert HCR_EL2().TGE == '0' && HCR_EL2().FMO == '1';
    end;
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64_TakeVirtualFIQException(); end;

    let preferred_exception_return : bits(32)= ThisInstrAddr{}();
    let vect_offset : integer = 0x1C;
    let lr_offset : integer   = 4;

    AArch32_EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
end;
```

J1.3.2.13 AArch32_TakeVirtualIRQException

```
// AArch32_TakeVirtualIRQException()
// ================================

func AArch32_TakeVirtualIRQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR().TGE == '0' && HCR().IMO == '1';
    else
        assert HCR_EL2().TGE == '0' && HCR_EL2().IMO == '1';
    end;

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64_TakeVirtualIRQException(); end;

    let preferred_exception_return :  bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x18;
    let lr_offset : integer   = 4;

    AArch32_EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
end;
```

J1.3.2.14 AArch32_TakeVirtualSErrorException

```
// AArch32_TakeVirtualSErrorException()
// ===================================

func AArch32_TakeVirtualSErrorException()
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR().TGE == '0' && HCR().AMO == '1';
    else
        assert HCR_EL2().TGE == '0' && HCR_EL2().AMO == '1';
    end;
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64_TakeVirtualSErrorException(); end;
    let route_to_monitor : boolean = FALSE;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let lr_offset : integer   = 8;

    let vaddress : bits(32)   = ARBITRARY : bits(32);
    let parity : boolean      = FALSE;
    let fault : Fault         = Fault_AsyncExternal;
    let level : integer       = ARBITRARY : integer;
    var fsr : bits(32)        = Zeros{};
    if IsFeatureImplemented(FEAT_RAS) then
        if ELUsingAArch32(EL2) then
            fsr[15:14] =  VDFSR().AET;
            fsr[12] =  VDFSR().ExT;
        else
            fsr[15:14] =  VSESR_EL2().AET;
            fsr[12] =  VSESR_EL2().ExT;
        end;
    else
        fsr[12] = ImpDefBit("Virtual External abort type");
    end;
    if TTBCR().EAE == '1' then // Long-descriptor format
        fsr[9] = '1';
        fsr[5:0] = EncodeLDFSC(fault, level);
    else                       // Short-descriptor format
        fsr[9] = '0';
        fsr[10,3:0] = EncodeSDFSC(fault, level);
    end;
    DFSR() = fsr;
    DFAR() = ARBITRARY : bits(32);
    ClearPendingVirtualSError();
    AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
end;
```

J1.3.2.15 AArch32_SoftwareBreakpoint

```
// AArch32_SoftwareBreakpoint()
// ============================

func AArch32_SoftwareBreakpoint(immediate : bits(16))
begin

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64_SoftwareBreakpoint(immediate);
    end;

    let accdesc : AccessDescriptor = CreateAccDescIFetch();
    var fault : FaultRecord = NoFault(accdesc, ARBITRARY : bits(64));

    fault.statuscode = Fault_Debug;
    fault.debugmoe   = DebugException_BKPT;

    AArch32_Abort(fault);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.16 DebugException

```
// DebugException
// ==============
// Reason codes for debug exceptions, taken to AArch32

constant DebugException_Breakpoint : bits(4)  = '0001';
constant DebugException_BKPT : bits(4)        = '0011';
constant DebugException_VectorCatch : bits(4) = '0101';
constant DebugException_Watchpoint : bits(4)  = '1010';
```

J1.3.2.17 AArch32_CheckAdvSIMDOrFPRegisterTraps

```
// AArch32_CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

func AArch32_CheckAdvSIMDOrFPRegisterTraps(reg : bits(4))
begin
    if PSTATE.EL == EL1 && EL2Enabled() then
        let tid0 : bit = if ELUsingAArch32(EL2) then HCR().TID0 else HCR_EL2().TID0;
        let tid3 : bit = if ELUsingAArch32(EL2) then HCR().TID3 else HCR_EL2().TID3;
        if ((tid0 == '1' && reg == '0000') ||                          // FPSID
              (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then     // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32_SystemAccessTrap(M32_Hyp, 0x8);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x8);
            end;
        end;
    end;
end;
```

J1.3.2.18 AArch32_ExceptionClass

```
// AArch32_ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

func AArch32_ExceptionClass(exceptype : Exception) => (integer,bit)
begin
    var il_is_valid : boolean = TRUE;
    var ec : integer;
    case exceptype of
        when Exception_Uncategorized =>        ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap =>              ec = 0x01;
        when Exception_CP15RTTrap =>           ec = 0x03;
        when Exception_CP15RRTTrap =>          ec = 0x04;
        when Exception_CP14RTTrap =>           ec = 0x05;
        when Exception_CP14DTTrap =>           ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap =>  ec = 0x07;
        when Exception_FPIDTrap =>             ec = 0x08;
        when Exception_CP14RRTTrap =>          ec = 0x0C;
        when Exception_IllegalState =>         ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall =>       ec = 0x11;
        when Exception_HypervisorCall =>       ec = 0x12;
        when Exception_MonitorCall =>          ec = 0x13;
        when Exception_InstructionAbort =>
            ec = if PSTATE.EL == EL2 then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception_PCAlignment =>          ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort =>            ec = if PSTATE.EL == EL2 then 0x25 else 0x24;
        when Exception_FPTrappedException =>   ec = 0x28;
        when Exception_Profiling =>            ec = 0x3D;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        otherwise =>                          unreachable;
    end;

    var il : bit;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    end;

    return (ec,il);
end;
```

J1.3.2.19 AArch32_GeneralExceptionsToAArch64

```
// AArch32_GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

func AArch32_GeneralExceptionsToAArch64() => boolean
begin
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2().TGE == '1'));
end;
```

J1.3.2.20 AArch32_ReportHypEntry

```
// AArch32_ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

func AArch32_ReportHypEntry(except : ExceptionRecord)
begin

    let exceptype : Exception = except.exceptype;

    var (ec,il) : (integer, bit) = AArch32_ExceptionClass(exceptype);
    let iss  : bits(25)= except.syndrome.iss;
    let iss2 : bits(24)= except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss[24] == '0' then
        il = '1';
    end;

    HSR() = ec[5:0]::il::iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR() = except.vaddress[31:0];
        HDFAR() = ARBITRARY : bits(32);
    elsif exceptype == Exception_DataAbort then
        HIFAR() = ARBITRARY : bits(32);
        HDFAR() = except.vaddress[31:0];
    end;

    if except.ipavalid then
        HPFAR()[31:4] = except.ipaddress[39:12];
    else
        HPFAR()[31:4] = ARBITRARY : bits(28);
    end;

    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.21 AArch32_ResetControlRegisters

```
// AArch32_ResetControlRegisters()
// ==============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

impdef func AArch32_ResetControlRegisters(cold_reset : boolean)
begin
    return;
end;
```

J1.3.2.22 AArch32_TakeReset

```
// AArch32_TakeReset()
// ===================
// Reset into AArch32 state

func AArch32_TakeReset(cold_reset : boolean)
begin
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32_WriteMode(M32_Svc);
        SCR().NS = '0';                                 // Secure state
    elsif HaveEL(EL2) then
        AArch32_WriteMode(M32_Hyp);
    else
        AArch32_WriteMode(M32_Svc);
    end;

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32_ResetControlRegisters(cold_reset);
    FPEXC().EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.[A,I,F] = '111';             // All asynchronous exceptions masked
    PSTATE.IT = '00000000';             // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR().TE;           // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR().EE;           // Endianness: EE=0: little-endian, EE=1: big-endian.
    else
        PSTATE.T = SCTLR().TE;            // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR().EE;            // Endianness: EE=0: little-endian, EE=1: big-endian.
    end;
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32_ResetGeneralRegisters();
    if IsFeatureImplemented(FEAT_SME) || IsFeatureImplemented(FEAT_SVE) then
        ResetSVERegisters();
    else
        AArch32_ResetSIMDFPRegisters();
    end;
    AArch32_ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);


    var rv : bits(32);                      // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if MVBAR()[0] == '1' then           // Reset vector in MVBAR
            rv = MVBAR()[31:1]::'0';
        else
            rv = ImpDefBits{32}("reset vector address");
        end;
    else
        rv = RVBAR()[31:1]::'0';
    end;

    // The reset vector must be correctly aligned
    assert rv[0] == '0' && (PSTATE.T == '1' || rv[1] == '0');

    let branch_conditional : boolean = FALSE;
    EDPRSR().R = '0';                 // Leaving Reset State.
    BranchTo{32}(rv, BranchType_RESET, branch_conditional);
end;
```

J1.3.2.23 ExcVectorBase

```
// ExcVectorBase()
// ===============

func ExcVectorBase() => bits(32)
begin
    if SCTLR().V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones{16}::Zeros{16};
    else
        return VBAR()[31:5]::Zeros{5};
    end;
end;
```

J1.3.2.24 AArch32_FPTrappedException

```
// AArch32_FPTrappedException()
// ============================

func AArch32_FPTrappedException(accumulated_exceptions : bits(8))
begin
    if AArch32_GeneralExceptionsToAArch64() then
        let is_ase : boolean = FALSE;
        let element : integer = 0;
        AArch64_FPTrappedException(is_ase, accumulated_exceptions);
    end;
    FPEXC().DEX    = '1';
    FPEXC().TFV    = '1';
    FPEXC()[7,4:0] = accumulated_exceptions[7,4:0];                  // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC()[10:8]  = '111';                                          // VECITR is RES1

    AArch32_TakeUndefInstrException();
end;
```

J1.3.2.25 AArch32_CallHypervisor

```
// AArch32_CallHypervisor()
// ========================
// Performs a HVC call

func AArch32_CallHypervisor(immediate : bits(16))
begin
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64_CallHypervisor(immediate);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch32_TakeHVCException(immediate);
    end;
end;
```

J1.3.2.26 AArch32_CallSupervisor

```
// AArch32_CallSupervisor()
// ========================
// Calls the Supervisor

func AArch32_CallSupervisor(immediate_in : bits(16))
begin
    var immediate : bits(16) = immediate_in;
    if CurrentCond() != '1110' then
        immediate = ARBITRARY : bits(16);
    end;
    if AArch32_GeneralExceptionsToAArch64() then
        AArch64_CallSupervisor(immediate);
    else
        AArch32_TakeSVCException(immediate);
    end;
end;
```

J1.3.2.27 AArch32_TakeHVCException

```
// AArch32_TakeHVCException()
// ==========================

func AArch32_TakeHVCException(immediate : bits(16))
begin
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32_ITAdvance();
    SSAdvance();
    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x08;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss[15:0] = immediate;

    if PSTATE.EL == EL2 then
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
    end;
end;
```

J1.3.2.28 AArch32_TakeSMCException

```
// AArch32_TakeSMCException()
// ==========================

func AArch32_TakeSMCException()
begin
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32_ITAdvance();
    HSAdvance();
    SSAdvance();
    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer = 0x08;
    let lr_offset : integer = 0;
    AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.29 AArch32_TakeSVCException

```
// AArch32_TakeSVCException()
// ==========================

func AArch32_TakeSVCException(immediate : bits(16))
begin

    AArch32_ITAdvance();
    SSAdvance();
    let route_to_hyp : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';

    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x08;
    let lr_offset : integer{} = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
        except.syndrome.iss[15:0] = immediate;
        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32_EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

J1.3.2.30 AArch32_EnterHypMode

```
// AArch32_EnterHypMode()
// ======================
// Take an exception to Hyp mode.

noreturn func AArch32_EnterHypMode(except : ExceptionRecord, preferred_exception_return : bits(32),
                                   vect_offset : integer)
begin
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    if Halted() then
        AArch32_EnterHypModeInDebugState(except);
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32_NonDebugState);
    if ! except.exceptype IN {Exception_IRQ, Exception_FIQ} then
        AArch32_ReportHypEntry(except);
    end;
    AArch32_WriteMode(M32_Hyp);
    SPSR_curr() = spsr;
    ELR_hyp() = preferred_exception_return;
    PSTATE.T  = HSCTLR().TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) then
        PSTATE.A = '1';
        PSTATE.I = '1';
        PSTATE.F = '1';
    else
        if ELUsingAArch32(EL3) then
            if SCR().EA  == '0' then PSTATE.A = '1'; end;
            if SCR().IRQ == '0' then PSTATE.I = '1'; end;
            if SCR().FIQ == '0' then PSTATE.F = '1'; end;
        else
            if SCR_EL3().EA  == '0' then PSTATE.A = '1'; end;
            if EffectiveSCR_EL3_IRQ() == '0' then PSTATE.I = '1'; end;
            if EffectiveSCR_EL3_FIQ() == '0' then PSTATE.F = '1'; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
    PSTATE.E  = HSCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = HSCTLR().DSSBS; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(HVBAR()[31:5]::vect_offset[4:0], BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.3.2.31 AArch32_EnterMode

```
// AArch32_EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

noreturn func AArch32_EnterMode(target_mode : bits(5),
                                preferred_exception_return : bits(32),
                                lr_offset : integer,
                                vect_offset : integer)
begin
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if Halted() then
        AArch32_EnterModeInDebugState(target_mode);
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    AArch32_WriteMode(target_mode);
    SPSR_curr() = spsr;
    R(14) = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR().TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.[A,I,F] = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.[A,I] = '11';
    else
        PSTATE.I = '1';
    end;
    PSTATE.E = SCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLR().SPAN == '0' then PSTATE.PAN = '1'; end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR().DSSBS; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(ExcVectorBase()[31:5]::vect_offset[4:0], BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.3.2.32 AArch32_EnterMonitorMode

```
// AArch32_EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

func AArch32_EnterMonitorMode(preferred_exception_return : bits(32), lr_offset : integer,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                vect_offset : integer)
begin
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    let from_secure : boolean = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32_EnterMonitorModeInDebugState();
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    AArch32_WriteMode(M32_Monitor);
    SPSR_curr() = spsr;
    R(14) = preferred_exception_return + lr_offset;
    PSTATE.T       = SCTLR().TE;                        // PSTATE.J is RES0
    PSTATE.SS      = '0';
    PSTATE.[A,I,F] = '111';
    PSTATE.E       = SCTLR().EE;
    PSTATE.IL      = '0';
    PSTATE.IT      = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR().SPAN == '0' then
            PSTATE.PAN = '1';
        end;
    end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR().DSSBS; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(MVBAR()[31:5]::vect_offset[4:0], BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

J1.3.2.33 AArch32_CheckAdvSIMDOrFPEnabled

```
// AArch32_CheckAdvSIMDOrFPEnabled()
// ================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

func AArch32_CheckAdvSIMDOrFPEnabled(fpexc_check_in : boolean, advsimd : boolean)
begin
    var fpexc_check : boolean = fpexc_check_in;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        // When executing at EL0 using AArch32, if EL1 is using AArch64 then the Effective value of
        // FPEXC.EN is 1. This includes when EL2 is using AArch64 and enabled in the current
        // Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1.
        AArch64_CheckFPAdvSIMDEnabled();
    else
        var cpacr_asedis = CPACR().ASEDIS;
        var cpacr_cp10 = CPACR().cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
            // Check if access disabled in NSACR
            if NSACR().NSASEDIS == '1' then cpacr_asedis = '1'; end;
            if NSACR().cp10 == '0' then cpacr_cp10 = '00'; end;
        end;

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then AArch32_Undefined(); end;

            // Check if access disabled in CPACR
            var disabled : boolean;
            case cpacr_cp10 of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                when '00' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0;
                when '10' => disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch32_Undefined(); end;
        end;

        // If required, check FPEXC enabled bit.
        if (fpexc_check && PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) &&
              HCR_EL2().TGE == '1') then
            // When executing at EL0 using AArch32, if EL2 is using AArch64 and enabled in the
            // current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 0
            // then it is IMPLEMENTATION DEFINED whether the Effective value of FPEXC.EN is 1
            // or the value of FPEXC32_EL2.EN.
            fpexc_check = ImpDefBool("Use FPEXC32_EL2.EN value when {TGE,RW} == {1,0}");
        end;

        if fpexc_check && FPEXC().EN == '0' then
            AArch32_Undefined();
        end;

        AArch32_CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
    end;
end;
```

J1.3.2.34 AArch32_CheckFPAdvSIMDTrap

```
// AArch32_CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

func AArch32_CheckFPAdvSIMDTrap(advsimd : boolean)
begin
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64_CheckFPAdvSIMDTrap();
    else
        if (HaveEL(EL3) && !ELUsingAArch32(EL3) &&
            CPTR_EL3().TFP == '1' && EL3SDDUndefPriority()) then
            Undefined();
        end;

        let ss : SecurityState = CurrentSecurityState();
        if HaveEL(EL2) && ss != SS_Secure then
            var hcptr_tase = HCPTR().TASE;
            var hcptr_cp10 = HCPTR().TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) then
                // Check if access disabled in NSACR
                if NSACR().NSASEDIS == '1' then hcptr_tase = '1'; end;
                if NSACR().cp10 == '0' then hcptr_cp10 = '1'; end;
            end;

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                var except : ExceptionRecord = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                except.syndrome.iss[24:20] = ConditionSyndrome();

                if advsimd then
                    except.syndrome.iss[5] = '1';
                else
                    except.syndrome.iss[5]   = '0';
                    except.syndrome.iss[3:0] = '1010';         // coproc field, always 0xA
                end;

                if PSTATE.EL == EL2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                AArch32_TakeUndefInstrException(except);
            else
                AArch32_TakeHypTrapException(except);
            end;
        end;
    end;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AdvSIMDFPAccessTrap(EL3);
            end;
        end;
    end;
  end;
end;
```

J1.3.2.35 AArch32_CheckForSMCUndefOrTrap

```
// AArch32_CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

func AArch32_CheckForSMCUndefOrTrap()
begin
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        Undefined();
    end;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64_CheckForSMCUndefOrTrap(Zeros{16});
    else
        let route_to_hyp : boolean = EL2Enabled() && PSTATE.EL == EL1 && HCR().TSC == '1';
        if route_to_hyp then
            let except : ExceptionRecord = ExceptionSyndrome(Exception_MonitorCall);
            AArch32_TakeHypTrapException(except);
        end;
    end;
end;
```

J1.3.2.36 AArch32_CheckForSVCTrap

```
// AArch32_CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

func AArch32_CheckForSVCTrap(immediate : bits(16))
begin
    if IsFeatureImplemented(FEAT_FGT) then
        let route_to_el2 : boolean = (PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && EL2Enabled() &&
                                      HFGITR_EL2().SVC_EL0 == '1' &&
                                      (!IsInHost() && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1')));

        if route_to_el2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome.iss[15:0] = immediate;
            except.trappedsyscallinst = TRUE;
            let preferred_exception_return : bits(64) = ThisInstrAddr{}();
            let vect_offset : integer = 0x0;

            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

J1.3.2.37 AArch32_CheckForWFxTrap

```
// AArch32_CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

func AArch32_CheckForWFxTrap(target_el : bits(2), wfxtype : WFxType)
begin
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        var trap : boolean;
        let is_wfe : boolean = wfxtype == WFxType_WFE;
        case target_el of
            when EL1 =>
                trap = (if is_wfe then SCTLR_ELx().nTWE else SCTLR_ELx().nTWI) == '0';
            when EL2 =>
                trap = (if is_wfe then HCR_EL2().TWE else HCR_EL2().TWI) == '1';
            when EL3 =>
                trap = (if is_wfe then SCR_EL3().TWE else SCR_EL3().TWI) == '1';
        end;

        if trap then
            if target_el == EL3 && EL3SDDUndef() then
                Undefined();
            else
                AArch64_WFxTrap(wfxtype, target_el);
            end;
        end;
        return;
    end;

    let is_wfe : boolean = wfxtype == WFxType_WFE;
    var trap : boolean;
    case target_el of
        when EL1 =>
            trap = (if is_wfe then SCTLR().nTWE else SCTLR().nTWI) == '0';
        when EL2 =>
            trap = (if is_wfe then HCR().TWE else HCR().TWI) == '1';
        when EL3 =>
            trap = (if is_wfe then SCR().TWE else SCR().TWI) == '1';
    end;

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2().TGE == '1' then
            AArch64_WFxTrap(wfxtype, target_el);
        end;

        if target_el == EL3 && !EL3SDDUndef() then
            AArch32_TakeMonitorTrapException();
        elsif target_el == EL2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_WFxTrap);
            except.syndrome.iss[24:20]   = ConditionSyndrome();
            except.syndrome.iss[0]       = if wfxtype == WFxType_WFI then '0' else '1';
            AArch32_TakeHypTrapException(except);
        else
            AArch32_TakeUndefInstrException();
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.38 AArch32_CheckITEnabled

```
// AArch32_CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

func AArch32_CheckITEnabled(mask : bits(4))
begin
    var it_disabled : bit;

    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR().ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR().ITD else SCTLR_ELx().ITD);
    end;
    if it_disabled == '1' then
        if mask != '1000' then Undefined(); end;

        let accdesc : AccessDescriptor = CreateAccDescIFetch();
        let aligned = TRUE;
        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        let next_instr = AArch32_MemSingle{16}(NextInstrAddr{32}(), accdesc, aligned);

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            Undefined();
        end;
    end;

    return;
end;
```

J1.3.2.39 AArch32_CheckIllegalState

```
// AArch32_CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

func AArch32_CheckIllegalState()
begin
    if AArch32_GeneralExceptionsToAArch64() then
        AArch64_CheckIllegalState();
    elsif PSTATE.IL == '1' then
        let route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';

        let preferred_exception_return : bits(32) = ThisInstrAddr{}();
        let vect_offset : integer = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            let except : ExceptionRecord = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
            end;
        else
            AArch32_TakeUndefInstrException();
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.40 AArch32_CheckSETENDEnabled

```
// AArch32_CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

func AArch32_CheckSETENDEnabled()
begin
    var setend_disabled : bit;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR().SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR().SED else SCTLR_ELx().SED);
    end;
    if setend_disabled == '1' then
        Undefined();
    end;

    return;
end;
```

J1.3.2.41 AArch32_SystemAccessTrap

```
// AArch32_SystemAccessTrap()
// ==========================
// Trapped AArch32 System register access.

func AArch32_SystemAccessTrap(mode : bits(5), ec : integer)
begin
    let (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        let except : ExceptionRecord = AArch32_SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32_TakeHypTrapException(except);
    else
        AArch32_TakeUndefInstrException();
    end;
end;
```

J1.3.2.42 AArch32_SystemAccessTrapSyndrome

```
// AArch32_SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

func AArch32_SystemAccessTrapSyndrome(instr : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;

    case ec of
        when 0x0  =>   except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3  =>   except = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4  =>   except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5  =>   except = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6  =>   except = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7  =>   except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8  =>   except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC  =>   except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise =>   unreachable;
    end;

    var iss : bits(20) = Zeros{};

    if except.exceptype == Exception_Uncategorized then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                               Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss[13:10] = instr[19:16];       // CRn, Reg in case of VMRS
        iss[8:5]   = instr[15:12];       // Rt
        iss[9]     = '0';                // RES0

        if except.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss[19:17] = instr[7:5];     // opc2
            iss[16:14] = instr[23:21];   // opc1
            iss[4:1]   = instr[3:0];     // CRm
        else //VMRS Access
            iss[19:17] = '000';          // opc2 - Hardcoded for VMRS
            iss[16:14] = '111';          // opc1 - Hardcoded for VMRS
            iss[4:1]   = '0000';         // CRm  - Hardcoded for VMRS
        end;
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                               Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss[19:16] = instr[7:4];         // opc1
        iss[13:10] = instr[19:16];       // Rt2
        iss[8:5]   = instr[15:12];       // Rt
        iss[4:1]   = instr[3:0];         // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss[19:12] = instr[7:0];         // imm8
        iss[4]     = instr[23];          // U
        iss[2:1]   = instr[24,21];       // P, W
        if instr[19:16] == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss[8:5] = ARBITRARY : bits(4);
            iss[3]   = '1';
        end;
    end;
    iss[0] = instr[20];                  // Direction

    except.syndrome.iss[24:20] = ConditionSyndrome();
    except.syndrome.iss[19:0]  = iss;

    return except;
end;
```

J1.3.2.43 AArch32_TakeHypTrapException

```
// AArch32_TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

func AArch32_TakeHypTrapException(ec : integer)
begin
    let except : ExceptionRecord = AArch32_SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32_TakeHypTrapException(except);
end;

// AArch32_TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

func AArch32_TakeHypTrapException(except : ExceptionRecord)
begin
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x14;

    AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.3.2.44 AArch32_TakeMonitorTrapException

```
// AArch32_TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

func AArch32_TakeMonitorTrapException()
begin
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    let preferred_exception_return :  bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x04;
    let lr_offset : integer = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
end;
```

J1.3.2.45 AArch32_TakeUndefInstrException

```
// AArch32_TakeUndefInstrException()
// =================================

noreturn func AArch32_TakeUndefInstrException(except : ExceptionRecord)
begin

    let route_to_hyp : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x04;
    let lr_offset : integer = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32_EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
    end;
end;

// AArch32_TakeUndefInstrException()
// =================================

noreturn func AArch32_TakeUndefInstrException()
begin
    let except : ExceptionRecord = ExceptionSyndrome(Exception_Uncategorized);
    AArch32_TakeUndefInstrException(except);
end;
```

J1.3.2.46 AArch32_Undefined

```
// AArch32_Undefined()
// ===================

noreturn func AArch32_Undefined()
begin

    if AArch32_GeneralExceptionsToAArch64() then AArch64_Undefined(); end;
    AArch32_TakeUndefInstrException();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3 aarch32/functions

This section includes the following pseudocode functions:

- AArch32_DomainValid
- AArch32_FaultSyndrome
- AArch32_InstructionSyndromeValid
- EncodeSDFSC
- A32ExpandImm
- A32ExpandImm_C
- DecodeImmShift
- DecodeRegShift
- RRX
- RRX_C
- SRType
- Shift
- Shift_C
- T32ExpandImm
- T32ExpandImm_C
- VBitOps
- VCGEType
- VCGTtype
- VFPNegMul
- AArch32_CheckCP15InstrCoarseTraps
- AArch32_ExclusiveMonitorsPass
- AArch32_IsExclusiveVA
- AArch32_MarkExclusiveVA
- AArch32_SetExclusiveMonitors
- CheckAdvSIMDEnabled
- CheckAdvSIMDOrVFPEnabled
- CheckCryptoEnabled32
- CheckVFPEnabled
- FPHalvedSub
- FPRSqrtStep
- FPRecipStep
- StandardFPCR
- AArch32_MemSingle
- AArch32_MemSingleRead
- AArch32_MemSingleWrite
- AArch32_UnalignedAccessFaults
- Hint_PreloadData
- Hint_PreloadDataForWrite
- Hint_PreloadInstr
- MemA
- MemO
- MemS
- MemU
- MemU_unpriv

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Mem_with_type
- AArch32_ESBOperation
- AArch32_EncodeAsyncErrorSyndrome
- AArch32_PhysicalSErrorSyndrome
- AArch32_vESBOperation
- AArch32_ResetGeneralRegisters
- AArch32_ResetSIMDFPRegisters
- AArch32_ResetSpecialRegisters
- AArch32_ResetSystemRegisters
- ALUExceptionReturn
- ALUWritePC
- BXWritePC
- BranchWritePC
- CBWritePC
- D
- Din
- H
- LR
- LoadWritePC
- LookUpRIndex
- Monitor
- PC32
- PCStoreValue
- Q
- Qin
- R
- RBankSelect
- ReadAnyAllocatedRegister
- ReadAnyAllocatedSPSR
- Rmode
- S
- WriteAnyAllocatedRegister
- WriteAnyAllocatedSPSR
- _Dclone
- AArch32_ExceptionReturn
- AArch32_ExecutingCP10or11Instr
- AArch32_ITAdvance
- AArch32_InterruptPending
- AArch32_SysRegRead
- AArch32_SysRegRead64
- AArch32_SysRegReadCanWriteAPSR
- AArch32_SysRegWrite
- AArch32_SysRegWrite64
- AArch32_SysRegWriteM
- AArch32_WriteMode
- AArch32_WriteModeByInstr
- BadMode

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BankedRegisterAccessValid
- CPSRWriteByInstr
- ConditionPassed
- CurrentCond
- InITBlock
- LastInITBlock
- SPSRWriteByInstr
- SPSRaccessValid
- SelectInstrSet
- AArch32_DTLBI_ALL
- AArch32_DTLBI_ASID
- AArch32_DTLBI_VA
- AArch32_EffectiveBroadcast
- AArch32_ITLBI_ALL
- AArch32_ITLBI_ASID
- AArch32_ITLBI_VA
- AArch32_TLBI_ALL
- AArch32_TLBI_ASID
- AArch32_TLBI_IPAS2
- AArch32_TLBI_VA
- AArch32_TLBI_VAA
- AArch32_TLBI_VMALL
- AArch32_TLBI_VMALLS12
- Sat
- AArch32_RestrictPrediction

J1.3.3.1 AArch32_DomainValid

```
// AArch32_DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

func AArch32_DomainValid(statuscode : Fault, level : integer) => boolean
begin
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain =>
            return TRUE;
        when Fault_Translation,Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk =>
            return level == 2;
        otherwise =>
            return FALSE;
    end;
end;
```

J1.3.3.2 AArch32_FaultSyndrome

```
// AArch32_FaultSyndrome()
// =======================
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to AArch32 Hyp mode.

func AArch32_FaultSyndrome(exceptype : Exception, fault : FaultRecord) => bits(25)
begin
    assert fault.statuscode != Fault_None;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var isstype : IssType;
    isstype.iss  = Zeros{25};

    let d_side : boolean = exceptype == Exception_DataAbort;
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        let errstate : ErrorState = PEErrorState(fault);
        isstype.iss[11:10] = AArch32_EncodeAsyncErrorSyndrome(errstate); // AET
    end;

    if d_side then

        if AArch32_InstructionSyndromeValid(fault) then
            isstype.iss[24:14] = LSInstructionSyndrome();
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[8] = '1';
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[6] = '1';
        elsif fault.statuscode == Fault_Exclusive then
            isstype.iss[6] = ARBITRARY : bit;
        else
            isstype.iss[6] = if fault.write then '1' else '0';
        end;

    end;
    if IsExternalAbort(fault) then isstype.iss[9] = fault.extflag; end;
    isstype.iss[7] = if fault.s2fs1walk then '1' else '0';
    isstype.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype.iss;
end;
```

J1.3.3.3 AArch32_InstructionSyndromeValid

```
// AArch32_InstructionSyndromeValid()
// ==================================
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

func AArch32_InstructionSyndromeValid(fault : FaultRecord) => boolean
begin
    if IsSecondStage(fault) && !fault.s2fs1walk then
        return (!IsExternalSyncAbort(fault) ||
                (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                   ImpDefBool("ISV on second stage translation table walk")));
    end;

    return FALSE;
end;
```

J1.3.3.4 EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

func EncodeSDFSC(statuscode : Fault, level : integer) => bits(5)
begin
    var result : bits(5);
    case statuscode of
        when Fault_AccessFlag =>
            assert level IN {1,2};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment =>
            result = '00001';
        when Fault_Permission =>
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain =>
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation =>
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal =>
            result = '01000';
        when Fault_SyncExternalOnWalk =>
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity =>
            result = '11001';
        when Fault_SyncParityOnWalk =>
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity =>
            result = '11000';
        when Fault_AsyncExternal =>
            result = '10110';
        when Fault_Debug =>
            result = '00010';
        when Fault_TLBConflict =>
            result = '10000';
        when Fault_Lockdown =>
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive =>
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint =>
            result = '00100';
        otherwise =>
            unreachable;
    end;

    return result;
end;
```

J1.3.3.5 A32ExpandImm

```
// A32ExpandImm()
// ==============

func A32ExpandImm(imm12 : bits(12)) => bits(32)
begin
    // PSTATE.C argument to following function call does not affect the imm32 result.
    let (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
end;
```

J1.3.3.6 A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

func A32ExpandImm_C(imm12 : bits(12), carry_in : bit) => (bits(32), bit)
begin
    let unrotated_value : bits(32) = ZeroExtend{}(imm12[7:0]);
    let (imm32, carry_out) : (bits(32), bit) = Shift_C{32}(unrotated_value, SRType_ROR,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                    2*UInt(imm12[11:8]), carry_in);

    return (imm32, carry_out);
end;
```

J1.3.3.7 DecodeImmShift

```
// DecodeImmShift()
// ================

func DecodeImmShift(srtype : bits(2), imm5 : bits(5)) => (SRType, integer{0..32})
begin
    var shift_t : SRType;
    var shift_n : integer{0..32};
    case srtype of
        when '00' =>
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01' =>
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10' =>
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11' =>
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);
            end;
    end;

    return (shift_t, shift_n);
end;
```

J1.3.3.8 DecodeRegShift

```
// DecodeRegShift()
// ================

func DecodeRegShift(srtype : bits(2)) => SRType
begin
    var shift_t : SRType;
    case srtype of
        when '00' =>  shift_t = SRType_LSL;
        when '01' =>  shift_t = SRType_LSR;
        when '10' =>  shift_t = SRType_ASR;
        when '11' =>  shift_t = SRType_ROR;
    end;
    return shift_t;
end;
```

J1.3.3.9 RRX

```
// RRX()
// =====

func RRX{N}(x : bits(N), carry_in : bit) => bits(N)
begin
    let (result, -) = RRX_C{N}(x, carry_in);
    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.10 RRX_C

```
// RRX_C()
// =======

func RRX_C{N}(x : bits(N), carry_in : bit) => (bits(N), bit)
begin
    let result : bits(N) = carry_in :: x[N-1:1];
    let carry_out : bit = x[0];
    return (result, carry_out);
end;
```

J1.3.3.11 SRType

```
// SRType
// ======

type SRType of enumeration {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

J1.3.3.12 Shift

```
// Shift()
// =======

func Shift{N}(value : bits(N), srtype : SRType, amount : integer, carry_in : bit) => bits(N)
begin
    let (result, -) = Shift_C{N}(value, srtype, amount, carry_in);
    return result;
end;
```

J1.3.3.13 Shift_C

```
// Shift_C()
// =========

func Shift_C{N}(value : bits(N), srtype : SRType, amount : integer,
                carry_in : bit) => (bits(N), bit)
begin
    assert !(srtype == SRType_RRX && amount != 1);

    var result : bits(N);
    var carry_out : bit;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL =>
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR =>
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR =>
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR =>
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX =>
                (result, carry_out) = RRX_C{N}(value, carry_in);
        end;
    end;

    return (result, carry_out);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.14 T32ExpandImm

```
// T32ExpandImm()
// ==============

func T32ExpandImm(imm12 : bits(12)) => bits(32)
begin
    // PSTATE.C argument to following function call does not affect the imm32 result.
    let (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
end;
```

J1.3.3.15 T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

func T32ExpandImm_C(imm12 : bits(12), carry_in : bit) => (bits(32), bit)
begin
    var imm32 : bits(32);
    var carry_out : bit;
    if imm12[11:10] == '00' then
        case imm12[9:8] of
            when '00' =>
                imm32 = ZeroExtend{32}(imm12[7:0]);
            when '01' =>
                imm32 = '00000000' :: imm12[7:0] :: '00000000' :: imm12[7:0];
            when '10' =>
                imm32 = imm12[7:0] :: '00000000' :: imm12[7:0] :: '00000000';
            when '11' =>
                imm32 = imm12[7:0] :: imm12[7:0] :: imm12[7:0] :: imm12[7:0];
        end;
        carry_out = carry_in;
    else
        let unrotated_value : bits(32) = ZeroExtend{}('1'::imm12[6:0]);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12[11:7]));
    end;

    return (imm32, carry_out);
end;
```

J1.3.3.16 VBitOps

```
// VBitOps
// =======

type VBitOps of enumeration {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

J1.3.3.17 VCGEType

```
// VCGEType
// ========

type VCGEType of enumeration {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

J1.3.3.18 VCGTtype

```
// VCGTtype
// ========

type VCGTtype of enumeration {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.19 VFPNegMul

```
// VFPNegMul
// =========

type VFPNegMul of enumeration {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

J1.3.3.20 AArch32_CheckCP15InstrCoarseTraps

```
// AArch32_CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

func AArch32_CheckCP15InstrCoarseTraps(CRn : integer, nreg : integer, CRm : integer)
begin
    if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
            (EL2Enabled() && !ELUsingAArch32(EL2))) then
        AArch64_CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
    end;

    let trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                            (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                            (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        let major : integer = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR().CRn/HSTR().CRm
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((! major IN {4,14} && HSTR()[major] == '1') ||
              (HCR().TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                  ImpDefBool("UNDEF unallocated CP15 access at EL0")) then
                Undefined();
            end;
            if ELUsingAArch32(EL2) then
                AArch32_SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x3);
            end;
        end;
    end;
end;
```

J1.3.3.21 AArch32_ExclusiveMonitorsPass

```
// AArch32_ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give a memory abort.


func AArch32_ExclusiveMonitorsPass(address : bits(32),
                                   size : integer{1, 2, 4, 8, 16, 32}) => boolean
begin
    let acqrel : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                                tagchecked, privileged);
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    if !AArch32_IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;
    end;

    let memaddrdesc : AddressDescriptor = AArch32_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32_Abort(memaddrdesc.fault);
    end;

    var passed : boolean = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    return passed;
end;
```

J1.3.3.22 AArch32_IsExclusiveVA

```
// AArch32_IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

impdef func AArch32_IsExclusiveVA(address : bits(32),
                                  processorid : integer,
                                  size : integer) => boolean
begin
    return TRUE;
end;
```

J1.3.3.23 AArch32_MarkExclusiveVA

```
// AArch32_MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

func AArch32_MarkExclusiveVA(address : bits(32), processorid : integer, size : integer)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.24 AArch32_SetExclusiveMonitors

```
// AArch32_SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

func AArch32_SetExclusiveMonitors(address : bits(32), size : integer{1, 2, 4, 8, 16, 32})
begin
    let acqrel : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel,
                                                         tagchecked, privileged);
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let memaddrdesc : AddressDescriptor = AArch32_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    end;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32_MarkExclusiveVA(address, ProcessorID(), size);
end;
```

J1.3.3.25 CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

func CheckAdvSIMDEnabled()
begin
    let fpexc_check : boolean = TRUE;
    let advsimd : boolean = TRUE;

    AArch32_CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31 do
        _Dclone[[i]] = D(i);
    end;

    return;
end;
```

J1.3.3.26 CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

func CheckAdvSIMDOrVFPEnabled(include_fpexc_check : boolean, advsimd : boolean)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch32_CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
end;
```

J1.3.3.27 CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

func CheckCryptoEnabled32()
begin
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
end;
```

J1.3.3.28 CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

func CheckVFPEnabled(include_fpexc_check : boolean)
begin
    let advsimd : boolean = FALSE;
    AArch32_CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
end;
```

J1.3.3.29 FPHalvedSub

```
// FPHalvedSub()
// =============

func FPHalvedSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    let rounding : FPRounding = FPRoundingMode(fpcr);
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean  = (type1 == FPType_Infinity);
        let inf2 : boolean  = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN{N}(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity{N}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity{N}('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero{N}(sign1);
        else
            let result_value : real = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr);
            end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
    return result;
end;
```

J1.3.3.30 FPRSqrtStep

```
// FPRSqrtStep()
// =============

func FPRSqrtStep{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    assert N IN {16,32};
    let fpcr : FPCR_Type = StandardFPCR();
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean  = (type1 == FPType_Infinity);
        let inf2 : boolean  = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);
        var product : bits(N);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero{N}('0');
        else
            product = FPMul{N}(op1, op2, fpcr);
        end;
        let three : bits(N) = FPThree{}('0');
        result = FPHalvedSub{N}(three, product, fpcr);
    end;
    return result;
end;
```

J1.3.3.31 FPRecipStep

```
// FPRecipStep()
// =============

func FPRecipStep{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    assert N IN {16,32};
    let fpcr : FPCR_Type = StandardFPCR();
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean  = (type1 == FPType_Infinity);
        let inf2 : boolean  = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);
        var product : bits(N);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero{N}('0');
        else
            product = FPMul{N}(op1, op2, fpcr);
        end;
        let two : bits(N) = FPTwo{}('0');
        result = FPSub{N}(two, product, fpcr);
    end;
    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.32 StandardFPCR

```
// StandardFPCR()
// ==============

func StandardFPCR() => FPCR_Type
begin
    let value : bits(32) = ('00000' :: FPSCR().AHP :: '110000' ::
                            FPSCR().FZ16 :: '0000000000000000000');
    return ZeroExtend{64}(value);
end;
```

J1.3.3.33 AArch32_MemSingle

```
// AArch32_MemSingle
// =================

accessor AArch32_MemSingle{size : integer{8, 16, 32, 64}}(address : bits(32),
                                                          accdesc : AccessDescriptor,
                                                          aligned : boolean) <=> value : bits(size)
begin
    // Perform an atomic, little-endian read of 'size' bits.
    getter
        let bytes : integer{} = size DIV 8;
        var value : bits(size);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (value, memaddrdesc, memstatus) = AArch32_MemSingleRead{size}(address, accdesc, aligned);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch32_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, bytes, accdesc);
        end;

        return value;
    end;

    // Perform an atomic, little-endian write of 'size' bits.
    setter
        let bytes : integer{} = size DIV 8;
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (memaddrdesc, memstatus) = AArch32_MemSingleWrite{size}(address, accdesc, aligned, value);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch32_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;

        return;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.34 AArch32_MemSingleRead

```
// AArch32_MemSingleRead()
// =======================
// Perform an atomic, little-endian read of 'size' bits.

func AArch32_MemSingleRead{size : integer{8, 16, 32, 64}}(address : bits(32),
                                                          accdesc_in : AccessDescriptor,
                                                          aligned : boolean
                                                          ) => (bits(size), AddressDescriptor,
                                                                PhysMemRetStatus)
begin
    let bytes : integer{}            = size DIV 8;
    var value : bits(size)           = ARBITRARY : bits(size);
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    var accdesc : AccessDescriptor = accdesc_in;
    assert IsAlignedSize(address, bytes);

    var memaddrdesc : AddressDescriptor;
    memaddrdesc = AArch32_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);
    end;

    // Memory array access
    (memstatus, value) = PhysMemRead{size}(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);
    end;

    if accdesc.acctype == AccessType_IFETCH then
        if ELUsingAArch32(S1TranslationRegime()) then
            memaddrdesc.fault = AArch32_CheckDebug(address, accdesc, bytes);
        else
            memaddrdesc.fault = AArch64_CheckDebug(ZeroExtend{64}(address), accdesc, bytes);
        end;
    end;

    return (value, memaddrdesc, memstatus);
end;
```

J1.3.3.35 AArch32_MemSingleWrite

```
// AArch32_MemSingleWrite()
// ========================
// Perform an atomic, little-endian write of 'size' bits.

func AArch32_MemSingleWrite{size : integer{8, 16, 32, 64}}(address : bits(32),
                                                           accdesc_in : AccessDescriptor,
                                                           aligned : boolean,
                                                           value : bits(size)
                                                          ) => (AddressDescriptor, PhysMemRetStatus)
begin
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    assert IsAlignedSize(address, bytes);

    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    memaddrdesc = AArch32_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;

    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);
    end;

    return (memaddrdesc, memstatus);
end;
```

J1.3.3.36 AArch32_UnalignedAccessFaults

```
// AArch32_UnalignedAccessFaults()
// ==============================
// Determine whether the unaligned access generates an Alignment fault

func AArch32_UnalignedAccessFaults(accdesc : AccessDescriptor) => boolean
begin
    return (AlignmentEnforced() ||
            accdesc.a32lsmd     ||
            accdesc.exclusive   ||
            accdesc.acqsc       ||
            accdesc.relsc);
end;
```

J1.3.3.37 Hint_PreloadData

```
// Hint_PreloadData()
// ==================
// Perform a preload data hint.

impdef func Hint_PreloadData(address : bits(32))
begin
    return;
end;
```

J1.3.3.38 Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// ==========================
// Perform a preload data hint with a probability that the use will be for a write.

impdef func Hint_PreloadDataForWrite(address : bits(32))
begin
    return;
end;
```

J1.3.3.39 Hint_PreloadInstr

```
// Hint_PreloadInstr()
// ===================
// Perform a preload instruction hint.

impdef func Hint_PreloadInstr(address : bits(32))
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.40 MemA

```
// MemA - accessor
// ===============

accessor MemA{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    getter
        let acqrel : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                             privileged);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let acqrel : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel, tagchecked,
                                                             privileged);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;
```

J1.3.3.41 MemO

```
// MemO
// ====

accessor MemO{size : integer{8, 16, 32, 64} }(address : bits(32)) <=> value : bits(size)
begin
    getter
        let acquire : boolean = TRUE;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let acquire : boolean = FALSE;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;
```

J1.3.3.42 MemS

```
// MemS - accessor
// ===============

accessor MemS{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    // Memory accessor for streaming load multiple instructions
    getter
        let accdesc : AccessDescriptor = CreateAccDescA32LSMD(MemOp_LOAD);
        return Mem_with_type{size}(address, accdesc);
    end;

    // Memory accessor for streaming store multiple instructions
    setter
        let accdesc : AccessDescriptor = CreateAccDescA32LSMD(MemOp_STORE);
        Mem_with_type{size}(address, accdesc) = value;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.3.3.43 MemU

```
// MemU - accessor
// ===============

accessor MemU{size : integer{8, 16, 32, 64} }(address : bits(32) ) <=> value: bits(size)
begin
    getter
        let nontemporal : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                          tagchecked);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let nontemporal : boolean = FALSE;
        let privileged : boolean  = PSTATE.EL != EL0;
        let tagchecked : boolean  = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                          tagchecked);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;
```

J1.3.3.44 MemU_unpriv

```
// MemU_unpriv
// ===========

accessor MemU_unpriv{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    getter
        let nontemporal : boolean = FALSE;
        let privileged : boolean  = FALSE;
        let tagchecked : boolean  = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                          tagchecked);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let nontemporal : boolean = FALSE;
        let privileged : boolean  = FALSE;
        let tagchecked : boolean  = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                          tagchecked);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;
```

J1.3.3.45 Mem_with_type

```
// Mem_with_type
// =============

accessor Mem_with_type{size : integer{8, 16, 32, 64}}(address : bits(32),
                                                      accdesc_in : AccessDescriptor
                                                     ) <=> value_in : bits(size)
begin
    // Perform a read of 'size' bits. The access byte order is reversed for a big-endian access.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Instruction fetches would call AArch32_MemSingle directly.
getter
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    var value : bits(size);

    // Check alignment on size of element accessed, not overall access size
    var aligned : boolean;
    if accdesc.ispair then
        let half : integer{} = (size DIV 2) as integer{32};
        aligned = IsAlignedSize(address, half DIV 8);
    else
        aligned = IsAlignedSize(address, bytes);
    end;

    if !aligned && AArch32_UnalignedAccessFaults(accdesc) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;
    if aligned then
        value = AArch32_MemSingle{size}(address, accdesc, aligned);
    else
        assert bytes > 1;
        value[7:0] = AArch32_MemSingle{8}(address, accdesc, aligned);

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.
        let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE; end;

        for i = 1 to bytes-1 do
            value[i*:8] = AArch32_MemSingle{8}(address+i, accdesc, aligned);
        end;
    end;

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse{size}(value);
    end;

    return value;
end;

// Perform a write of 'size' bits. The byte order is reversed for a big-endian access.
setter
    let bytes : integer{}          = size DIV 8;
    var value :  bits(size)        = value_in;
    var accdesc : AccessDescriptor = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    var aligned : boolean;
    if accdesc.ispair then
        let half : integer{} = (size DIV 2) as integer{32};
        aligned = IsAlignedSize(address, half DIV 8);
    else
        aligned = IsAlignedSize(address, bytes);
    end;

    if !aligned && AArch32_UnalignedAccessFaults(accdesc) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse{size}(value);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
        if aligned then
            AArch32_MemSingle{size}(address, accdesc, aligned) = value;
        else
            assert bytes > 1;
            AArch32_MemSingle{8}(address, accdesc, aligned) = value[7:0];

            // For subsequent bytes, if they cross to a new translation page which assigns
            // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
            // will generate an Alignment Fault.

            let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE; end;

            for i = 1 to bytes-1 do
                AArch32_MemSingle{8}(address+i, accdesc, aligned) = value[i*:8];
            end;
        end;
        return;
    end;
end;
```

J1.3.3.46 AArch32_ESBOperation

```
// AArch32_ESBOperation()
// ======================
// Perform the AArch32 ESB operation for ESB executed in AArch32 state.

func AArch32_ESBOperation()
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalSErrorTarget();

    // Check if routed to AArch64 state
    if !masked && !ELUsingAArch32(target_el) then
        AArch64_ESBOperation();
        return;
    end;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        var syndrome : bits(32) = Zeros{};
        syndrome[31] = '1';  // A
        syndrome[15:0] = AArch32_PhysicalSErrorSyndrome();
        DISR() = syndrome;
        ClearPendingPhysicalSError();
    end;

    return;
end;
```

J1.3.3.47 AArch32_EncodeAsyncErrorSyndrome

```
// AArch32_EncodeAsyncErrorSyndrome()
// ==================================
// Return the encoding for specified  ErrorState for an SError exception taken
// to AArch32 state.

func AArch32_EncodeAsyncErrorSyndrome(errorstate : ErrorState) => bits(2)
begin
    case errorstate of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when ErrorState_UC =>   return '00';
        when ErrorState_UEU =>  return '01';
        when ErrorState_UEO =>  return '10';
        when ErrorState_UER =>  return '11';
        otherwise => unreachable;
    end;
end;
```

J1.3.3.48 AArch32_PhysicalSErrorSyndrome

```
// AArch32_PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

func AArch32_PhysicalSErrorSyndrome() => bits(16)
begin
    var syndrome : bits(32) = Zeros{};
    let fault : FaultRecord = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        let errstate : ErrorState = PEErrorState(fault);
        syndrome[11:10] = AArch32_EncodeAsyncErrorSyndrome(errstate);   // AET
        syndrome[9]     = fault.extflag;                                // EA
        syndrome[5:0]   = '010001';                                     // DFSC
    else
        let long_format : boolean = TTBCR().EAE == '1';
        syndrome = AArch32_CommonFaultStatus(fault, long_format);
    end;
    return syndrome[15:0];
end;
```

J1.3.3.49 AArch32_vESBOperation

```
// AArch32_vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state.
// If FEAT_E3DSE is implemented and there is no unmasked virtual SError exception
// pending, then AArch64_dESBOperation() is called to perform the AArch64 ESB operation
// for a pending delegated SError exception.

func AArch32_vESBOperation()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64_vESBOperation();
        return;
    end;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a virtual
    // SError interrupt might be pending.
    let vsei_pending : boolean = IsVirtualSErrorPending() && HCR().TGE == '0' && HCR().AMO == '1';
    let vsei_masked : boolean  = (PSTATE.A == '1' || Halted() ||
                                  ExternalDebugInterruptsDisabled(EL1));

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        var syndrome : bits(32) = Zeros{};
        syndrome[31]    = '1';           // A
        syndrome[15:14] = VDFSR()[15:14];  // AET
        syndrome[12]    = VDFSR()[12];     // ExT
        syndrome[9]     = TTBCR().EAE;     // LPAE
        if TTBCR().EAE == '1' then         // Long-descriptor format
            syndrome[5:0]    = '010001'; // STATUS
        else                               // Short-descriptor format
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            syndrome[10,3:0] = '10110';  // FS
        end;
        VDISR() = syndrome;
        ClearPendingVirtualSError();
    elsif IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) then
        AArch64_dESBOperation();
    end;

    return;
end;
```

J1.3.3.50 AArch32_ResetGeneralRegisters

```
// AArch32_ResetGeneralRegisters()
// ==============================

func AArch32_ResetGeneralRegisters()
begin

    for i = 0 to 7 do
        R(i) = ARBITRARY : bits(32);
    end;
    for i = 8 to 12 do
        Rmode(i, M32_User) = ARBITRARY : bits(32);
        Rmode(i, M32_FIQ)  = ARBITRARY : bits(32);
    end;
    if HaveEL(EL2) then Rmode(13, M32_Hyp) = ARBITRARY : bits(32); end;   // No R14_hyp
    for i = 13 to 14 do
        Rmode(i, M32_User)  = ARBITRARY : bits(32);
        Rmode(i, M32_FIQ)   = ARBITRARY : bits(32);
        Rmode(i, M32_IRQ)   = ARBITRARY : bits(32);
        Rmode(i, M32_Svc)   = ARBITRARY : bits(32);
        Rmode(i, M32_Abort) = ARBITRARY : bits(32);
        Rmode(i, M32_Undef) = ARBITRARY : bits(32);
        if HaveEL(EL3) then Rmode(i, M32_Monitor) = ARBITRARY : bits(32); end;
    end;

    return;
end;
```

J1.3.3.51 AArch32_ResetSIMDFPRegisters

```
// AArch32_ResetSIMDFPRegisters()
// ==============================

func AArch32_ResetSIMDFPRegisters()
begin

    for i = 0 to 15 do
        Q(i) = ARBITRARY : bits(128);
    end;

    return;
end;
```

J1.3.3.52 AArch32_ResetSpecialRegisters

```
// AArch32_ResetSpecialRegisters()
// ==============================

func AArch32_ResetSpecialRegisters()
begin

    // AArch32 special registers
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SPSR_fiq()[31:0] = ARBITRARY : bits(32);
    SPSR_irq()[31:0] = ARBITRARY : bits(32);
    SPSR_svc()[31:0] = ARBITRARY : bits(32);
    SPSR_abt()[31:0] = ARBITRARY : bits(32);
    SPSR_und()[31:0] = ARBITRARY : bits(32);
    if HaveEL(EL2) then
        SPSR_hyp() = ARBITRARY : bits(32);
        ELR_hyp()  = ARBITRARY : bits(32);
    end;
    if HaveEL(EL3) then
        SPSR_mon() = ARBITRARY : bits(32);
    end;

    // External debug special registers
    DLR()   = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);

    return;
end;
```

J1.3.3.53 AArch32_ResetSystemRegisters

```
// AArch32_ResetSystemRegisters()
// ==============================

impdef func AArch32_ResetSystemRegisters(cold_reset : boolean)
begin
    return;
end;
```

J1.3.3.54 ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

func ALUExceptionReturn(address : bits(32))
begin
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.M IN {M32_User,M32_System} then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
        end;
    else
        AArch32_ExceptionReturn(address, SPSR_curr());
    end;
end;
```

J1.3.3.55 ALUWritePC

```
// ALUWritePC()
// ============

func ALUWritePC(address : bits(32))
begin
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

J1.3.3.56 BXWritePC

```
// BXWritePC()
// ===========

func BXWritePC(address_in : bits(32), branch_type : BranchType)
begin
    var address : bits(32) = address_in;
    if address[0] == '1' then
        SelectInstrSet(InstrSet_T32);
        address[0] = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the PE takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address[1] == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address[1] = '0';
        end;
    end;
    let branch_conditional : boolean = CurrentCond() != '111x';
    BranchTo{32}(address, branch_type, branch_conditional);
end;
```

J1.3.3.57 BranchWritePC

```
// BranchWritePC()
// ===============

func BranchWritePC(address_in : bits(32), branch_type : BranchType)
begin
    var address : bits(32) = address_in;
    if CurrentInstrSet() == InstrSet_A32 then
        address[1:0] = '00';
    else
        address[0] = '0';
    end;
    let branch_conditional : boolean = CurrentCond() != '111x';
    BranchTo{32}(address, branch_type, branch_conditional);
end;
```

J1.3.3.58 CBWritePC

```
// CBWritePC()
// ===========
// Takes a branch from a CBNZ/CBZ instruction.

func CBWritePC(address_in : bits(32))
begin
    var address : bits(32) = address_in;
    assert CurrentInstrSet() == InstrSet_T32;
    address[0] = '0';
    let branch_conditional : boolean = TRUE;
    BranchTo{32}(address, BranchType_DIR, branch_conditional);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.59 D

```
// D - accessor
// ============

accessor D(n : integer) <=> value : bits(64)
begin
    getter
        assert n >= 0 && n <= 31;
        let vreg : bits(128) = V{}(n DIVRM 2);
        return vreg[(n MOD 2)*:64];
    end;
    setter
        assert n >= 0 && n <= 31;
        var vreg : bits(128) = V{}(n DIVRM 2);
        vreg[(n MOD 2)*:64] = value;
        V{128}(n DIVRM 2) = vreg;
    end;
end;
```

J1.3.3.60 Din

```
// Din
// ===
// Return the initial value of D input

func Din(n : integer) => bits(64)
begin
    assert n >= 0 && n <= 31;
    return _Dclone[[n]];
end;
```

J1.3.3.61 H

```
// H - accessor
// ============

accessor H(n : integer) <=> value : bits(16)
begin
    getter
        assert n >= 0 && n <= 31;
        return S(n)[15:0];
    end;
    setter
        S(n) = ZeroExtend{32}(value);
    end;
end;
```

J1.3.3.62 LR

```
// LR - accessor
// =============

accessor LR() <=> value : bits(32)
begin
    getter
        return R(14);
    end;
    setter
        R(14) = value;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.63 LoadWritePC

```
// LoadWritePC()
// =============

func LoadWritePC(address : bits(32))
begin
    BXWritePC(address, BranchType_INDIR);
end;
```

J1.3.3.64 LookUpRIndex

```
// LookUpRIndex()
// ==============

func LookUpRIndex(n : integer, mode : bits(5)) => integer
begin
    assert n >= 0 && n <= 14;

    var result : integer;
    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8    =>  result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9    =>  result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10   =>  result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11   =>  result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12   =>  result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13   =>  result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14   =>  result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise =>  result = n;
    end;

    return result;
end;
```

J1.3.3.65 Monitor

```
// Monitor mode registers
// ======================
// The Monitor mode registers do not map to X registers, so must be defined separately

var SP_mon : bits(32);

var LR_mon : bits(32);
```

J1.3.3.66 PC32

```
// PC32
// ====
// Read 32-bit program counter

func PC32() => bits(32)
begin
    return R(15);               // This includes the offset from AArch32 state
end;
```

J1.3.3.67 PCStoreValue

```
// PCStoreValue()
// ==============

func PCStoreValue() => bits(32)
begin
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC32();
end;
```

J1.3.3.68 Q

```
// Q - accessor
// ============

accessor Q(n : integer) <=> value : bits(128)
begin
    getter
        assert n >= 0 && n <= 15;
        return V{128}(n);
    end;
    setter
        assert n >= 0 && n <= 15;
        V{128}(n) = value;
    end;
end;
```

J1.3.3.69 Qin

```
// Qin
// ===
// Return the initial value of Q input

func Qin(n : integer) => bits(128)
begin
    assert n >= 0 && n <= 15;
    return Din(2*n+1)::Din(2*n);
end;
```

J1.3.3.70 R

```
// R - accessor
// ============

accessor R(n : integer) <=> value : bits(32)
begin
    getter
        if n == 15 then
            let offset : integer{} = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
            return _PC[31:0] + offset;
        else
            return Rmode(n, PSTATE.M);
        end;
    end;
    setter
        Rmode(n, PSTATE.M) = value;
        return;
    end;
end;

// R - accessor
// ============

accessor R(lr : integer, hr : integer) <=> value : bits(64)
begin
    getter
        return R(hr) :: R(lr);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    setter
        R(lr) = value[0+:32];
        R(hr) = value[32+:32];
    end;
end;
```

J1.3.3.71 RBankSelect

```
// RBankSelect()
// =============

func RBankSelect(mode : bits(5), usr : integer, fiq : integer, irq : integer,
                 svc : integer, abt : integer, und : integer, hyp : integer) => integer
begin

    var result : integer;
    case mode of
        when M32_User   => result = usr;  // User mode
        when M32_FIQ    => result = fiq;  // FIQ mode
        when M32_IRQ    => result = irq;  // IRQ mode
        when M32_Svc    => result = svc;  // Supervisor mode
        when M32_Abort  => result = abt;  // Abort mode
        when M32_Hyp    => result = hyp;  // Hyp mode
        when M32_Undef  => result = und;  // Undefined mode
        when M32_System => result = usr;  // System mode uses User mode registers
        otherwise       => unreachable;   // Monitor mode
    end;

    return result;
end;
```

J1.3.3.72 ReadAnyAllocatedRegister

```
// ReadAnyAllocatedRegister()
// ==========================

impdef func ReadAnyAllocatedRegister() => bits(32)
begin
    return Zeros{32};
end;
```

J1.3.3.73 ReadAnyAllocatedSPSR

```
// ReadAnyAllocatedSPSR()
// ======================

impdef func ReadAnyAllocatedSPSR() => bits(32)
begin
    return Zeros{32};
end;
```

J1.3.3.74 Rmode

```
// Rmode - accessor
// ================

accessor Rmode(n : integer, mode : bits(5)) <=> value : bits(32)
begin
    getter
        assert n >= 0 && n <= 14;

        // Check for attempted use of Monitor mode in Non-secure state.
        if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor; end;
        assert !BadMode(mode);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if mode == M32_Monitor then
            if n == 13 then return SP_mon;
            elsif n == 14 then return LR_mon;
            else return _R[[n]][31:0];
            end;
        else
            return _R[[LookUpRIndex(n, mode)]][31:0];
        end;
    end;

    setter
        assert n >= 0 && n <= 14;

        // Check for attempted use of Monitor mode in Non-secure state.
        if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor; end;
        assert !BadMode(mode);

        if mode == M32_Monitor then
            if n == 13 then SP_mon = value;
            elsif n == 14 then LR_mon = value;
            else _R[[n]][31:0] = value;
            end;
        else
            // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
            // register are unchanged or set to zero. This is also tested for on
            // exception entry, as this applies to all AArch32 registers.
            if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
                _R[[LookUpRIndex(n, mode)]] = ZeroExtend{64}(value);
            else
                _R[[LookUpRIndex(n, mode)]][31:0] = value;
            end;
        end;
    end;
end;
```

J1.3.3.75 S

```
// S - accessor
// ============

accessor S(n : integer) <=> value : bits(32)
begin
    getter
        assert n >= 0 && n <= 31;
        let vreg : bits(128) = V{}(n DIVRM 4);
        return vreg[(n MOD 4)*:32];
    end;

    setter
        assert n >= 0 && n <= 31;
        var vreg : bits(128) = V{}(n DIVRM 4);
        vreg[(n MOD 4)*:32] = value;
        V{128}(n DIVRM 4) = vreg;
    end;
end;
```

J1.3.3.76 WriteAnyAllocatedRegister

```
// WriteAnyAllocatedRegister()
// ===========================

impdef func WriteAnyAllocatedRegister(value : bits(32))
begin
    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.3.3.77 WriteAnyAllocatedSPSR

```
// WriteAnyAllocatedSPSR()
// =======================

impdef func WriteAnyAllocatedSPSR(value : bits(32))
begin
    return;
end;
```

J1.3.3.78 _Dclone

```
// _Dclone[]
// =========
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use as input to
// instruction pseudocode, to avoid read-after-write for Advanced SIMD and VFP operations.

var _Dclone : array [[32]] of bits(64);
```

J1.3.3.79 AArch32_ExceptionReturn

```
// AArch32_ExceptionReturn()
// =========================

func AArch32_ExceptionReturn(new_pc_in : bits(32), spsr : bits(32))
begin
    var new_pc : bits(32) = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR{32}(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc[1:0] = ARBITRARY : bits(2);
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc[0] = '0';                // T32
        else
            new_pc[1:0] = '00';             // A32
        end;
    end;

    let branch_conditional : boolean = CurrentCond() != '111x';
    BranchTo{32}(new_pc, BranchType_ERET, branch_conditional);
    CheckExceptionCatch(FALSE);             // Check for debug event on exception return
end;
```

J1.3.3.80 AArch32_ExecutingCP10or11Instr

```
// AArch32_ExecutingCP10or11Instr()
// ================================

func AArch32_ExecutingCP10or11Instr() => boolean
begin
    let instr : bits(32) =  ThisInstr();
    let instr_set : InstrSet = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if instr_set == InstrSet_A32 then
        return ((instr[27:24] == '1110' || instr[27:25] == '110') && instr[11:8] == '101x');
    else // InstrSet_T32
        return (instr[31:28] == '111x' && (instr[27:24] == '1110' || instr[27:25] == '110') &&
                instr[11:8] == '101x');
    end;
end;
```

J1.3.3.81 AArch32_ITAdvance

```
// AArch32_ITAdvance()
// ===================

func AArch32_ITAdvance()
begin
    if PSTATE.IT[2:0] == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT[4:0] = LSL(PSTATE.IT[4:0], 1);
    end;
    return;
end;
```

J1.3.3.82 AArch32_InterruptPending

```
// AArch32_InterruptPending()
// ==========================
// Returns TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise.

func AArch32_InterruptPending() => boolean
begin
    if !ELUsingAArch32(EL2) then
        return AArch64_InterruptPending();
    end;

    let (irq_pending, -) = IRQPending();
    let (fiq_pending, -) = FIQPending();
    let pending_physical_interrupt : boolean = (irq_pending || fiq_pending ||
                                                IsPhysicalSErrorPending());

    var pending_virtual_interrupt : boolean = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR().TGE == '0' then
        let virq_pending : boolean = (HCR().IMO == '1' && (VirtualIRQPending() || HCR().VI == '1'));
        let vfiq_pending : boolean = (HCR().FMO == '1' && (VirtualFIQPending() || HCR().VF == '1'));
        let vsei_pending : boolean = (HCR().AMO == '1' &&
                                      (IsVirtualSErrorPending() || HCR().VA == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;
    end;

    return pending_physical_interrupt || pending_virtual_interrupt;
end;
```

J1.3.3.83 AArch32_SysRegRead

```
// AArch32_SysRegRead()
// ====================
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].

impdef func AArch32_SysRegRead(cp_num : integer, instr : bits(32), t : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[21+:3];
    let CRn  = instr[16+:4];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let CRm  = instr[0+:4];
    let opc2 = instr[5+:3];
end;
```

J1.3.3.84 AArch32_SysRegRead64

```
// AArch32_SysRegRead64()
// ======================
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].

impdef func AArch32_SysRegRead64(cp_num : integer, instr : bits(32), t : integer, t2 : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[4+:4];
    let CRm  = instr[0+:4];
end;
```

J1.3.3.85 AArch32_SysRegReadCanWriteAPSR

```
// AArch32_SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

func AArch32_SysRegReadCanWriteAPSR(cp_num : integer, instr : bits(32)) => boolean
begin
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr[11:8]);

    let opc1 : integer{} = UInt(instr[23:21]);
    let opc2 : integer{} = UInt(instr[7:5]);
    let CRn : integer{}  = UInt(instr[19:16]);
    let CRm : integer{}  = UInt(instr[3:0]);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;
    end;

    return FALSE;
end;
```

J1.3.3.86 AArch32_SysRegWrite

```
// AArch32_SysRegWrite()
// =====================
// Read the contents of R[t] and write to a 32-bit AArch32 System register.

impdef func AArch32_SysRegWrite(cp_num : integer, instr : bits(32), t : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[21+:3];
    let CRn  = instr[16+:4];
    let CRm  = instr[ 0+:4];
    let opc2 = instr[ 5+:3];
end;
```

J1.3.3.87 AArch32_SysRegWrite64

```
// AArch32_SysRegWrite64()
// =======================
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.

impdef func AArch32_SysRegWrite64(cp_num : integer,
                                  instr : bits(32),
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    t : integer,
                                    t2 : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[4+:4];
    let CRm  = instr[0+:4];
end;
```

J1.3.3.88 AArch32_SysRegWriteM

```
// AArch32_SysRegWriteM()
// ======================
// Read a value from a virtual address and write it to an AArch32 System register.

impdef func AArch32_SysRegWriteM(cp_num : integer, instr : bits(32), address : bits(32))
begin
    let CRd : bits(4) = instr[15:12];
end;
```

J1.3.3.89 AArch32_WriteMode

```
// AArch32_WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

func AArch32_WriteMode(mode : bits(5))
begin
    let (valid,el) : (boolean, bits(2)) = ELFromM32(mode);
    assert valid;
    PSTATE.M   = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
end;
```

J1.3.3.90 AArch32_WriteModeByInstr

```
// AArch32_WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

func AArch32_WriteModeByInstr(mode : bits(5))
begin
    var (valid,el) : (boolean, bits(2)) = ELFromM32(mode);

    // 'valid' is set to FALSE if' mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;
    end;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;
    end;
    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if (PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR().NS == '1'
            && HCR().TGE == '1') then
        valid = FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32_WriteMode(mode);
    end;
end;
```

J1.3.3.91 BadMode

```
// BadMode()
// =========

readonly func BadMode(mode : bits(5)) => boolean
begin
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    var valid : boolean;
    case mode of
        when M32_Monitor =>
            valid = HaveAArch32EL(EL3);
        when M32_Hyp =>
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System =>
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32
            valid = HaveAArch32EL(EL1);
        when M32_User =>
            valid = HaveAArch32EL(EL0);
        otherwise =>
            valid = FALSE;           // Passed an illegal mode value
    end;
    return !valid;
end;
```

J1.3.3.92 BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

func BankedRegisterAccessValid(SYSm : bits(5), mode : bits(5)) => (boolean, Constraint)
begin
    var valid : boolean = TRUE;
    var c : Constraint = Constraint_NONE;
    var banked_unpred : boolean = FALSE;
    case SYSm of
        when '000xx', '00100' =>                          // R8_usr to R12_usr
            banked_unpred = mode != M32_FIQ;
        when '00101' =>                                   // SP_usr
            banked_unpred = mode == M32_System;
        when '00110' =>                                   // LR_usr
            banked_unpred = mode IN {M32_Hyp,M32_System};
        when '010xx', '0110x', '01110' =>                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            banked_unpred = mode == M32_FIQ;
        when '1000x' =>                                   // LR_irq, SP_irq
            banked_unpred = mode == M32_IRQ;
        when '1001x' =>                                   // LR_svc, SP_svc
            banked_unpred = mode == M32_Svc;
        when '1010x' =>                                   // LR_abt, SP_abt
            banked_unpred = mode == M32_Abort;
        when '1011x' =>                                   // LR_und, SP_und
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            banked_unpred = mode == M32_Undef;
        when '1110x' =>                                     // LR_mon, SP_mon
            banked_unpred = (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure ||
                             mode == M32_Monitor);
        when '11110' =>                                     // ELR_hyp, only from Monitor or Hyp mode
            banked_unpred = !HaveEL(EL2) || ! mode IN {M32_Monitor,M32_Hyp};
        when '11111' =>                                     // SP_hyp, only from Monitor mode
            banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
        otherwise =>
            valid = FALSE;
            c     = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
            assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_ANYREG};
    end;

    if banked_unpred then
        valid = FALSE;
        c     = ConstrainUnpredictable(Unpredictable_BankedRegister);
        assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_UNKNOWN};
    end;

    return (valid, c);
end;
```

J1.3.3.93 CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.[N,Z,C,V,Q,GR,E,A,I,F,M] from a CPSR value written by an MSR instruction.

func CPSRWriteByInstr(value : bits(32), bytemask : bits(4))
begin
    let privileged : boolean = PSTATE.EL != EL0;        // PSTATE.[A,I,F,M] are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask[3] == '1' then
        PSTATE.[N,Z,C,V,Q] = value[31:27];
        // Bits [26:24] are ignored
    end;

    if bytemask[2] == '1' then
        if IsFeatureImplemented(FEAT_SSBS) then
            PSTATE.SSBS = value[23];
        end;
        if privileged then
            PSTATE.PAN = value[22];
        end;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = value[21];
        end;
        // Bit [20] is RES0
        PSTATE.GE = value[19:16];
    end;

    if bytemask[1] == '1' then
        // Bits [15:10] are RES0
        PSTATE.E = value[9];                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value[8];
        end;
    end;

    if bytemask[0] == '1' then
        if privileged then
            PSTATE.[I,F] = value[7:6];
            // Bit [5] is RES0
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch32_WriteModeByInstr(value[4:0]);
        end;
    end;
    return;
end;
```

J1.3.3.94 ConditionPassed

```
// ConditionPassed()
// =================

func ConditionPassed() => boolean
begin
    return ConditionHolds(CurrentCond());
end;
```

J1.3.3.95 CurrentCond

```
// CurrentCond()
// =============
// Returns the condition code that applies to the currently executing instruction

impdef func CurrentCond() => bits(4)
begin
    return Zeros{4};
end;
```

J1.3.3.96 InITBlock

```
// InITBlock()
// ===========

func InITBlock() => boolean
begin
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT[3:0] != '0000';
    else
        return FALSE;
    end;
end;
```

J1.3.3.97 LastInITBlock

```
// LastInITBlock()
// ===============

func LastInITBlock() => boolean
begin
    return (PSTATE.IT[3:0] == '1000');
end;
```

J1.3.3.98 SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

func SPSRWriteByInstr(value : bits(32), bytemask : bits(4))
begin
    var new_spsr : bits(32) = SPSR_curr();

    if bytemask[3] == '1' then
        new_spsr[31:24] = value[31:24];  // N, Z, C, V, Q flags, IT[1:0],J bits
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if bytemask[2] == '1' then
        new_spsr[23:16] = value[23:16];  // IL bit, GE[3:0] flags
    end;

    if bytemask[1] == '1' then
        new_spsr[15:8] = value[15:8];    // IT[7:2] bits, E bit, A interrupt mask
    end;

    if bytemask[0] == '1' then
        new_spsr[7:0] = value[7:0];      // I, F interrupt masks, T bit, Mode bits
    end;

    SPSR_curr() = new_spsr;              // UNPREDICTABLE if User or System mode

    return;
end;
```

J1.3.3.99 SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

func SPSRaccessValid(SYSm : bits(5), mode : bits(5)) => (boolean, Constraint)
begin
    var valid : boolean = TRUE;
    var c : Constraint = Constraint_NONE;
    var banked_unpred : boolean = FALSE;
    case SYSm of
        when '01110' =>                                                  // SPSR_fiq
            banked_unpred = mode == M32_FIQ;
        when '10000' =>                                                  // SPSR_irq
            banked_unpred = mode == M32_IRQ;
        when '10010' =>                                                  // SPSR_svc
            banked_unpred = mode == M32_Svc;
        when '10100' =>                                                  // SPSR_abt
            banked_unpred = mode == M32_Abort;
        when '10110' =>                                                  // SPSR_und
            banked_unpred = mode == M32_Undef;
        when '11100' =>                                                  // SPSR_mon
            banked_unpred = (!HaveEL(EL3) || mode == M32_Monitor ||
                             CurrentSecurityState() != SS_Secure);
        when '11110' =>                                                  // SPSR_hyp
            banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
        otherwise =>
            valid = FALSE;
            c = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
            assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_ANYREG};
    end;

    if banked_unpred then
        valid = FALSE;
        c     = ConstrainUnpredictable(Unpredictable_BankedRegister);
        assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_UNKNOWN};
    end;

    return (valid, c);
end;
```

J1.3.3.100 SelectInstrSet

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// SelectInstrSet()
// ================

func SelectInstrSet(iset : InstrSet)
begin
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
end;
```

J1.3.3.101 AArch32_DTLBI_ALL

```
// AArch32_DTLBI_ALL()
// ===================
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_DTLBI_ALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, attr_in : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.102 AArch32_DTLBI_ASID

```
// AArch32_DTLBI_ASID()
// ====================
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_DTLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_DASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros{8} :: Rt[7:0];

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.103 AArch32_DTLBI_VA

```
// AArch32_DTLBI_VA()
// ==================
// Invalidate by VA all stage 1 data TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_DTLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, level : TLBILevel,  attr_in : TLBIMemAttr,
                      Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros{8} :: Rt[7:0];
    r.address      = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.104 AArch32_EffectiveBroadcast

```
// AArch32_EffectiveBroadcast()
// ============================

func AArch32_EffectiveBroadcast(broadcast_in : Broadcast) => Broadcast
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var broadcast : Broadcast = broadcast_in;

    if PSTATE.EL != EL1 then return broadcast; end;

    if (broadcast == Broadcast_NSH && ((ELUsingAArch32(EL2) && HCR().FB == '1') ||
          (!ELUsingAArch32(EL2) && HCR_EL2().FB == '1'))) then
        broadcast = Broadcast_ForcedISH;
    end;

    return broadcast;
end;
```

J1.3.3.105 AArch32_ITLBI_ALL

```
// AArch32_ITLBI_ALL()
// ===================
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_ITLBI_ALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, attr_in : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_IALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.106 AArch32_ITLBI_ASID

```
// AArch32_ITLBI_ASID()
// ====================
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID
// (where regime supports) and ASID in the parameter Rt in the indicated translation
// regime with the indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_ITLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var r : TLBIRecord;
    r.op           = TLBIOp_IASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros{8} :: Rt[7:0];

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.107 AArch32_ITLBI_VA

```
// AArch32_ITLBI_VA()
// ==================
// Invalidate by VA all stage 1 instruction TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_ITLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, level : TLBILevel,  attr_in : TLBIMemAttr,
                      Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros{8} :: Rt[7:0];
    r.address      = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.108 AArch32_TLBI_ALL

```
// AArch32_TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_TLBI_ALL(security : SecurityState, regime : Regime,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        broadcast : Broadcast, attr : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2};

    var r : TLBIRecord;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.109 AArch32_TLBI_ASID

```
// AArch32_TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros{8} :: Rt[7:0];

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.110 AArch32_TLBI_IPAS2

```
// AArch32_TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.

func AArch32_TLBI_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast : Broadcast, level : TLBILevel, attr : TLBIMemAttr, Rt : bits(32))
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS_NonSecure;

    var r : TLBIRecord;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros{24} :: Rt[27:0] :: Zeros{12};
    r.ipaspace     = PAS_NonSecure;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.111 AArch32_TLBI_VA

```
// AArch32_TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                     broadcast_in : Broadcast, level : TLBILevel,  attr_in : TLBIMemAttr,
                     Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros{8} :: Rt[7:0];
    r.address      = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.112 AArch32_TLBI_VAA

```
// AArch32_TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_VAA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
                      Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.113 AArch32_TLBI_VMALL

```
// AArch32_TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

func AArch32_TLBI_VMALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, attr_in : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.3.3.114 AArch32_TLBI_VMALLS12

```
// AArch32_TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

func AArch32_TLBI_VMALLS12(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                           broadcast : Broadcast, attr : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2};

    var r : TLBIRecord;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

J1.3.3.115 Sat

```
// Sat()
// =====

func Sat{N}(i : integer, unsigned : boolean) => bits(N)
begin
    let result : bits(N) = if unsigned then UnsignedSat{N}(i) else SignedSat{N}(i);
    return result;
end;
```

J1.3.3.116 AArch32_RestrictPrediction

```
// AArch32_RestrictPrediction()
// ============================
// Clear all predictions in the context.

func AArch32_RestrictPrediction(val : bits(32), restriction : RestrictType)
begin

    var c : ExecutionCntxt;
    let target_el : bits(2) = val[25:24];

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP(); end;

    let ns : bit  = val[26];
    let nse : bit = ARBITRARY : bit;
    let ss : SecurityState = TargetSecurityState(ns, nse);

    c.security  = ss;
    c.target_el = target_el;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID();

        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val[27] == '1';
            c.vmid          = ZeroExtend{NUM_VMIDBITS}(val[23:16]);// Only valid if  val[27] == '0'
        else
            c.is_vmid_valid = FALSE;
        end;
    else
        c.is_vmid_valid = FALSE;
    end;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID();

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val[8] == '1';
        c.asid          = ZeroExtend{NUM_ASIDBITS}(val[7:0]);      // Only valid if  val[8] == '0'

    else
        c.is_asid_valid = FALSE;
    end;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
end;
```

J1.3.4 aarch32/translation

This section includes the following pseudocode functions:

- AArch32_DefaultTEXDecode
- AArch32_MAIRAttr
- AArch32_RemappedTEXDecode
- AArch32_CheckBreakpoint
- AArch32_CheckDebug
- AArch32_CheckVectorCatch
- AArch32_CheckWatchpoint
- AArch32_IPAIsOutOfRange
- AArch32_S1HasAlignmentFaultDueToMemType
- AArch32_S1LDHasPermissionsFault
- AArch32_S1SDHasPermissionsFault
- AArch32_S2HasAlignmentFaultDueToMemType
- AArch32_S2HasPermissionsFault
- AArch32_S2InconsistentSL
- AArch32_VAIsOutOfRange
- AArch32_GetS1TLBContext
- AArch32_GetS2TLBContext
- AArch32_TLBContextEL10

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32_TLBContextEL2
- AArch32_TLBContextEL30
- AArch32_EL2Enabled
- AArch32_FullTranslate
- AArch32_OutputDomain
- AArch32_S1DisabledOutput
- AArch32_S1Enabled
- AArch32_S1TranslateLD
- AArch32_S1TranslateSD
- AArch32_S2Translate
- AArch32_SDStageOA
- AArch32_TranslateAddress
- SDFSize
- AArch32_DecodeDescriptorTypeLD
- AArch32_DecodeDescriptorTypeSD
- AArch32_S1IASize
- AArch32_S1WalkLD
- AArch32_S1WalkSD
- AArch32_S2IASize
- AArch32_S2StartLevel
- AArch32_S2Walk
- RemapRegsHaveResetValues
- AArch32_GetS1TTWParams
- AArch32_GetS2TTWParams
- AArch32_GetVARange
- AArch32_S1DCacheEnabled
- AArch32_S1ICacheEnabled
- AArch32_S1TTWParamsEL10
- AArch32_S1TTWParamsEL2
- AArch32_S1TTWParamsEL30

J1.3.4.1 AArch32_DefaultTEXDecode

```
// AArch32_DefaultTEXDecode()
// ==========================
// Apply short-descriptor format memory region attributes, without TEX remap

func AArch32_DefaultTEXDecode(TEX_in : bits(3), c_in : bit, b_in : bit, s : bit) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;
    var TEX : bits(3) = TEX_in;
    var c : bit = c_in;
    var b : bit = b_in;

    // Reserved values map to allocated values
    if (TEX == '001' && c::b == '01') || (TEX == '010' && c::b != '00') || TEX == '011' then
        var texcb : bits(5);
        (-, texcb) = ConstrainUnpredictableBits{5}(Unpredictable_RESTEXCB);
        TEX = texcb[4:2];  c = texcb[1];  b = texcb[0];
    end;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX::c::b of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '00000' =>
            // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '00001', '01000' =>
            // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '00010' =>
            // Write-through Read allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WT;
            memattrs.inner.hints  = MemHint_RA;
            memattrs.outer.attrs  = MemAttr_WT;
            memattrs.outer.hints  = MemHint_RA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00011' =>
            // Write-back Read allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WB;
            memattrs.inner.hints  = MemHint_RA;
            memattrs.outer.attrs  = MemAttr_WB;
            memattrs.outer.hints  = MemHint_RA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00100' =>
            // Non-cacheable
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_NC;
            memattrs.outer.attrs  = MemAttr_NC;
            memattrs.shareability = Shareability_OSH;
        when '00110' =>
            memattrs = ImpDefMemoryAttributes("IMPLEMENTATION_DEFINED");
        when '00111' =>
            // Write-back Read and Write allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WB;
            memattrs.inner.hints  = MemHint_RWA;
            memattrs.outer.attrs  = MemAttr_WB;
            memattrs.outer.hints  = MemHint_RWA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '1xxxx' =>
            // Cacheable, TEX[1:0] = Outer attrs, {c,b} = Inner attrs
            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(c::b);
            memattrs.outer   = DecodeSDFAttr(TEX[1:0]);

            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
            end;
        otherwise =>
            // Reserved, handled above
            unreachable;
    end;

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags            = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memattrs.xs = '1';
    end;

    return memattrs;
end;
```

J1.3.4.2 AArch32_MAIRAttr

```
// AArch32_MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

func AArch32_MAIRAttr(index : integer, mair : bits(64)) => bits(8)
begin
    assert (index < 8);
    return mair[index*:8];
end;
```

J1.3.4.3 AArch32_RemappedTEXDecode

```
// AArch32_RemappedTEXDecode()
// ===========================
// Apply short-descriptor format memory region attributes, with TEX remap

func AArch32_RemappedTEXDecode(regime : Regime, TEX : bits(3),
                               c : bit, b : bit, s : bit) => MemoryAttributes
begin

    var memattrs : MemoryAttributes;
    var prrr : PRRR_Type;
    var nmrr : NMRR_Type;

    let region : integer{} = UInt(TEX[0]::c::b);   // TEX[2:1] are ignored in this mapping scheme
    if region == 6 then
        return ImpDefMemoryAttributes("IMPLEMENTATION_DEFINED");
    end;

    if regime == Regime_EL30 then
        prrr = PRRR_S();
        nmrr = NMRR_S();
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        prrr = PRRR_NS();
        nmrr = NMRR_NS();
    else
        prrr = PRRR();
        nmrr = NMRR();
    end;

    let base : integer{} = 2 * region;
    var attrfield : bits(2) = prrr[region*:2];

    if attrfield == '11' then      // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits{2}(Unpredictable_RESPRRR);
    end;

    case attrfield of
        when '00' =>                  // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '01' =>                  // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '10' =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let NSn : bit  = if s == '0' then prrr.NS0 else prrr.NS1;
            let NOSm : bit = prrr[region+24] AND NSn;
            let IRn : bits(2)  = nmrr[base+1:base];
            let ORn : bits(2)  = nmrr[base+17:base+16];

            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(IRn);
            memattrs.outer   = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                let sh : bits(2) = NSn::NOSm;
                memattrs.shareability = DecodeShareability(sh);
            end;
        when '11' =>
            unreachable;
    end;

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags            = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';
    end;

    return memattrs;
end;
```

J1.3.4.4 AArch32_CheckBreakpoint

```
// AArch32_CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

func AArch32_CheckBreakpoint(fault_in : FaultRecord, vaddress : bits(32),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;
    var mismatch : boolean = FALSE;
    var brkptinfo : BreakpointInfo;

    for i = 0 to NumBreakpointsImplemented() - 1 do
        brkptinfo = AArch32_BreakpointMatch(i, vaddress, accdesc, size);
        match = match || brkptinfo.match;
        mismatch = mismatch || brkptinfo.mismatch;
    end;

    if match && HaltOnBreakpointOrWatchpoint() then
        let reason : bits(6) = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_Breakpoint;
        fault.vaddress   = ZeroExtend{64}(vaddress);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return fault;
end;
```

J1.3.4.5 AArch32_CheckDebug

```
// AArch32_CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

func AArch32_CheckDebug(vaddress : bits(32), accdesc : AccessDescriptor,
                        size : integer) => FaultRecord
begin

    var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(vaddress));

    let d_side : boolean = IsWatchpointableAccess(accdesc);
    let i_side : boolean = (accdesc.acctype == AccessType_IFETCH);
    let generate_exception : boolean = (AArch32_GenerateDebugExceptions() &&
                                        DBGDSCRext().MDBGen == '1');
    let halt : boolean = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    let vector_catch_first : boolean = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32_CheckVectorCatch(fault, vaddress, size);
    end;

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32_CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch32_CheckBreakpoint(fault, vaddress, accdesc, size);
        end;
    end;

    if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
        return AArch32_CheckVectorCatch(fault, vaddress, size);
    end;

    return fault;
end;
```

J1.3.4.6 AArch32_CheckVectorCatch

```
// AArch32_CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

func AArch32_CheckVectorCatch(fault_in : FaultRecord, vaddress : bits(32),
                              size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());

    var fault : FaultRecord = fault_in;
    var match : boolean = AArch32_VCRMatch(vaddress);
    if size == 4 && !match && AArch32_VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    end;

    if match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_VectorCatch;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return fault;
end;
```

J1.3.4.7 AArch32_CheckWatchpoint

```
// AArch32_CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

func AArch32_CheckWatchpoint(fault_in : FaultRecord, vaddress : bits(32),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());
    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;

    if !IsWatchpointableAccess(accdesc) then
        return fault;
    end;

    for i = 0 to NumWatchpointsImplemented() - 1 do
        let watchptinfo : WatchpointInfo = AArch32_WatchpointMatch(i, vaddress, size, accdesc);
        match = match || watchptinfo.value_match;
    end;

    if match && HaltOnBreakpointOrWatchpoint() then
        let reason : bits(6) = DebugHalt_Watchpoint;
        EDWAR()  = ZeroExtend{64}(vaddress);
        Halt(reason);
    elsif match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_Watchpoint;
        fault.vaddress   = ZeroExtend{64}(vaddress);
    end;

    return fault;
end;
```

J1.3.4.8 AArch32_IPAIsOutOfRange

```
// AArch32_IPAIsOutOfRange()
// =========================
// Check intermediate physical address bits not resolved by translation are ZERO

func AArch32_IPAIsOutOfRange(walkparams : S2TTWParams, ipa : bits(40)) => boolean
begin
    // Input Address size
    let iasize : integer{} = AArch32_S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa[39:iasize]);
end;
```

J1.3.4.9 AArch32_S1HasAlignmentFaultDueToMemType

```
// AArch32_S1HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 1 output fails alignment requirement on data accesses to memory type

func AArch32_S1HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                             ntlsmd : bit, memattrs : MemoryAttributes) => boolean
begin
    if memattrs.memtype != MemType_Device then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return FALSE;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.device != DeviceType_GRE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

J1.3.4.10 AArch32_S1LDHasPermissionsFault

```
// AArch32_S1LDHasPermissionsFault()
// ================================
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

func AArch32_S1LDHasPermissionsFault(regime : Regime, walkparams : S1TTWParams, perms : Permissions,
                                     memtype : MemType, paspace : PASpace,
                                     accdesc : AccessDescriptor) => boolean
begin
    var r,  w,  x : bit;
    var pr, pw : bit;
    var ur, uw : bit;
    var xn : bit;
    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        end;

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' => (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' => (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' => (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' => (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access
        end;

        xn   = perms.xn OR perms.xn_table;
        let pxn : bit  = perms.pxn OR perms.pxn_table;

        let ux : bit = ur AND NOT(xn OR (uw AND walkparams.wxn));
        let px : bit = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
            let pan : bit = PSTATE.PAN AND (ur OR uw);
            pr  = pr AND NOT(pan);
            pw  = pw AND NOT(pan);
        end;

        (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

        // Prevent execution from Non-secure space by PE in Secure state if SIF is set
        if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
            x = x AND NOT(walkparams.sif);
        end;
    else
        // Apply leaf permissions
        case perms.ap[2] of
            when '0' => (r,w) = ('1','1'); // No effect
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '1' => (r,w) = ('1','0'); // Read-only
        end;

        // Apply hierarchical permissions
        case perms.ap_table[1] of
            when '0' => (r,w) = ( r, w );  // No effect
            when '1' => (r,w) = ( r, '0'); // Read-only
        end;

        xn = perms.xn OR perms.xn_table;
        x  = NOT(xn OR (w AND walkparams.wxn));
    end;

    if accdesc.acctype == AccessType_IFETCH then
        let constraint : Constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
        end;
    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
    end;
end;
```

J1.3.4.11 AArch32_S1SDHasPermissionsFault

```
// AArch32_S1SDHasPermissionsFault()
// ================================
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

func AArch32_S1SDHasPermissionsFault(regime : Regime, perms_in : Permissions, memtype : MemType,
                                     paspace : PASpace, accdesc : AccessDescriptor) => boolean
begin
    var perms : Permissions = perms_in;
    var pr, pw : bit;
    var ur, uw : bit;
    var sctlr : SCTLR_Type;
    if regime == Regime_EL30 then
        sctlr = SCTLR_S();
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        sctlr = SCTLR_NS();
    else
        sctlr = SCTLR();
    end;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = ImpDefBits{3}("Reserved short descriptor AP encoding");
        end;

        case perms.ap of
            when '000' => (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' => (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' => (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
            when '011' => (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            //   '100' is reserved
            when '101' => (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '110' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
            when '111' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    else // Simplified access permissions model
        case perms.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        end;
    end;

    let ux : bit = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    let px : bit = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
        let pan : bit = PSTATE.PAN AND (ur OR uw);
        pr  = pr AND NOT(pan);
        pw  = pw AND NOT(pan);
    end;

    var (r,w,x) : (bit, bit, bit) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR().SIF else SCR_EL3().SIF);
    end;

    if accdesc.acctype == AccessType_IFETCH then
        if (memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            return TRUE;
        else
            return x == '0';
        end;
    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
    end;
end;
```

J1.3.4.12 AArch32_S2HasAlignmentFaultDueToMemType

```
// AArch32_S2HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

func AArch32_S2HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                             memattrs : MemoryAttributes) => boolean
begin

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

J1.3.4.13 AArch32_S2HasPermissionsFault

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32_S2HasPermissionsFault()
// ==============================
// Returns whether stage 2 access violates permissions of target memory

func AArch32_S2HasPermissionsFault(walkparams : S2TTWParams, perms : Permissions, memtype : MemType,
                                   accdesc : AccessDescriptor) => boolean
begin
    let r : bit = perms.s2ap[0];
    let w : bit = perms.s2ap[1];
    var x : bit;
    var ux : bit;
    var px : bit;
    if IsFeatureImplemented(FEAT_XNX) then
        case perms.s2xn::perms.s2xnx of
            when '00' =>  (px, ux) = ( r, r );
            when '01' =>  (px, ux) = ('0', r );
            when '10' =>  (px, ux) = ('0', '0');
            when '11' =>  (px, ux) = ( r, '0');
        end;

        x = if accdesc.el == EL0 then ux else px;
    else
        x = r AND NOT(perms.s2xn);
    end;

    if accdesc.acctype == AccessType_TTW then
        return (walkparams.ptw == '1' && memtype == MemType_Device) || r == '0';

    elsif accdesc.acctype == AccessType_IFETCH then
        let constraint : Constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        return (constraint == Constraint_FAULT && memtype == MemType_Device) || x == '0';

    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;

    elsif accdesc.write then
        return w == '0';

    else
        return r == '0';
    end;
end;
```

J1.3.4.14 AArch32_S2InconsistentSL

```
// AArch32_S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

func AArch32_S2InconsistentSL(walkparams : S2TTWParams) => boolean
begin
    let startlevel : integer  = AArch32_S2StartLevel(walkparams.sl0);
    let levels : integer      = FINAL_LEVEL - startlevel;
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let stride : integer{}      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    let sl_min_iasize : integer = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
    let sl_max_iasize : integer = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let iasize : AddressSize        = AArch32_S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
end;
```

J1.3.4.15 AArch32_VAIsOutOfRange

```
// AArch32_VAIsOutOfRange()
// ========================
// Check virtual address bits not resolved by translation are identical
// and of accepted value

func AArch32_VAIsOutOfRange(regime : Regime, walkparams : S1TTWParams, va : bits(32)) => boolean
begin
    if regime == Regime_EL2 then
        // Input Address size
        let iasize : integer{} = AArch32_S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va[31:iasize]);
    elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        let lo_iasize : integer{} = AArch32_S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        let up_iasize : integer{} = AArch32_S1IASize(walkparams.t1sz);
        return !IsZero(va[31:lo_iasize]) && !IsOnes(va[31:up_iasize]);
    else
        return FALSE;
    end;
end;
```

J1.3.4.16 AArch32_GetS1TLBContext

```
// AArch32_GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

func AArch32_GetS1TLBContext(regime : Regime, ss : SecurityState, va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;

    case regime of
        when Regime_EL2  => tlbcontext = AArch32_TLBContextEL2(va);
        when Regime_EL10 => tlbcontext = AArch32_TLBContextEL10(ss, va);
        when Regime_EL30 => tlbcontext = AArch32_TLBContextEL30(va);
    end;

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
end;
```

J1.3.4.17 AArch32_GetS2TLBContext

```
// AArch32_GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

func AArch32_GetS2TLBContext(ipa : FullAddress) => TLBContext
begin
    assert ipa.paspace == PAS_NonSecure;

    var tlbcontext : TLBContext;

    tlbcontext.ss          = SS_NonSecure;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend{NUM_VMIDBITS}(VTTBR().VMID);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend{64}(ipa.address);
    tlbcontext.cnp         = if IsFeatureImplemented(FEAT_TTCNP) then VTTBR().CnP else '0';

    return tlbcontext;
end;
```

J1.3.4.18 AArch32_TLBContextEL10

```
// AArch32_TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

func AArch32_TLBContextEL10(ss : SecurityState, va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;
    var ttbcr : TTBCR_Type;
    var ttbr0 : TTBR0_Type;
    var ttbr1 : TTBR1_Type;
    var contextidr : CONTEXTIDR_Type;

    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        ttbcr      = TTBCR_NS();
        ttbr0      = TTBR0_NS();
        ttbr1      = TTBR1_NS();
        contextidr = CONTEXTIDR_NS();
    else
        ttbcr      = TTBCR();
        ttbr0      = TTBR0();
        ttbr1      = TTBR1();
        contextidr = CONTEXTIDR();
    end;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;

    if AArch32_EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend{NUM_VMIDBITS}(VTTBR().VMID);
    end;

    if ttbcr.EAE == '1' then
        if ttbcr.A1 == '0' then
            tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(ttbr0.ASID);
        else
            tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(ttbr1.ASID);
        end;
    else
        tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(contextidr.ASID);
    end;

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend{64}(va);

    if IsFeatureImplemented(FEAT_TTCNP) && ttbcr.EAE == '1' then
        if AArch32_GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```

J1.3.4.19 AArch32_TLBContextEL2

```
// AArch32_TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

func AArch32_TLBContextEL2(va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss     = SS_NonSecure;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.ia     = ZeroExtend{64}(va);
    tlbcontext.tg     = TGx_4KB;
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then HTTBR().CnP else '0';

    return tlbcontext;
end;
```

J1.3.4.20 AArch32_TLBContextEL30

```
// AArch32_TLBContextEL30()
// ========================
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

func AArch32_TLBContextEL30(va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss     = SS_Secure;
    tlbcontext.regime = Regime_EL30;
    if TTBCR_S().EAE == '1' then
        tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(if TTBCR_S().A1 == '0' then TTBR0_S().ASID
                                                                            else TTBR1_S().ASID);
    else
        tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(CONTEXTIDR_S().ASID);
    end;

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend{64}(va);

    if IsFeatureImplemented(FEAT_TTCNP) && TTBCR_S().EAE == '1' then
        if AArch32_GetVARange(va, TTBCR_S().T0SZ, TTBCR_S().T1SZ) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_S().CnP;
        else
            tlbcontext.cnp = TTBR1_S().CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.21 AArch32_EL2Enabled

```
// AArch32_EL2Enabled()
// ====================
// Returns whether EL2 is enabled for the given Security State

func AArch32_EL2Enabled(ss : SecurityState) => boolean
begin
    if ss == SS_Secure then
        if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2)) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3().EEL2 == '1';
        else
            return ImpDefBool("Secure-only implementation");
        end;
    else
        return HaveEL(EL2);
    end;
end;
```

J1.3.4.22 AArch32_FullTranslate

```
// AArch32_FullTranslate()
// =======================
// Perform address translation as specified by VMSA-A32

func AArch32_FullTranslate(va : bits(32), accdesc : AccessDescriptor,
                           aligned : boolean) => AddressDescriptor
begin

    // Prepare fault fields in case a fault is detected
    var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(va));
    let regime : Regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    var ipa : AddressDescriptor;
    if regime == Regime_EL2 || TTBCR().EAE == '1' then
        (fault, ipa)    = AArch32_S1TranslateLD(fault, regime, va, aligned, accdesc);
    else
        (fault, ipa, -) = AArch32_S1TranslateSD(fault, regime, va, aligned, accdesc);
    end;

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(fault);
    end;

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend{64}(va);
        var pa : AddressDescriptor;
        (fault, pa) = AArch32_S2Translate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(fault);
        else
            return pa;
        end;
    else
        return ipa;
    end;
end;
```

J1.3.4.23 AArch32_OutputDomain

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32_OutputDomain()
// ======================
// Determine the domain the translated output address

func AArch32_OutputDomain(regime : Regime, domain : bits(4)) => bits(2)
begin
    var Dn : bits(2);
    if regime == Regime_EL30 then
        Dn = DACR_S()[UInt(domain)*:2];
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        Dn = DACR_NS()[UInt(domain)*:2];
    else
        Dn = DACR()[UInt(domain)*:2];
    end;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits{2}(Unpredictable_RESDACR);
    end;

    return Dn;
end;
```

J1.3.4.24 AArch32_S1DisabledOutput

```
// AArch32_S1DisabledOutput()
// ==========================
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

func AArch32_S1DisabledOutput(fault_in : FaultRecord, regime : Regime,
                              va : bits(32), aligned : boolean,
                              accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin

    var fault : FaultRecord = fault_in;
    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    var memattrs : MemoryAttributes;
    var default_cacheable : bit;
    if regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) then
        if ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
            default_cacheable = HCR().DC;
        else
            default_cacheable = HCR_EL2().DC;
        end;
    else
        default_cacheable = '0';
    end;

    if default_cacheable == '1' then
        // Use default cacheable settings
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs  = MemAttr_WB;
        memattrs.inner.hints  = MemHint_RWA;
        memattrs.outer.attrs  = MemAttr_WB;
        memattrs.outer.hints  = MemHint_RWA;
        memattrs.shareability = Shareability_NSH;
        if (EL2Enabled() && !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) &&
              IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().DCT == '1') then
            memattrs.tags     = MemTag_AllocationTagged;
        else
            memattrs.tags     = MemTag_Untagged;
        end;
        memattrs.xs           = '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif accdesc.acctype == AccessType_IFETCH then
        memattrs.memtype      = MemType_Normal;
        memattrs.shareability = Shareability_OSH;
        memattrs.tags         = MemTag_Untagged;
        if AArch32_S1ICacheEnabled(regime) then
            memattrs.inner.attrs = MemAttr_WT;
            memattrs.inner.hints = MemHint_RA;
            memattrs.outer.attrs = MemAttr_WT;
            memattrs.outer.hints = MemHint_RA;
        else
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.outer.attrs = MemAttr_NC;
        end;
        memattrs.xs           = '1';
    else
        // Treat memory region as Device
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
        memattrs.tags         = MemTag_Untagged;
        memattrs.xs           = '1';
    end;

    var ntlsmd : bit;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        case regime of
            when Regime_EL30 => ntlsmd = SCTLR_S().nTLSMD;
            when Regime_EL2  => ntlsmd = HSCTLR().nTLSMD;
            when Regime_EL10 =>
                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    ntlsmd = SCTLR_NS().nTLSMD;
                else
                    ntlsmd = SCTLR().nTLSMD;
                end;
        end;
    else
        ntlsmd = '1';
    end;

    if AArch32_S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, memattrs) then
        fault.statuscode  = Fault_Alignment;
        return (fault, ARBITRARY : AddressDescriptor);
    elsif (accdesc.exclusive &&
            !(regime == Regime_EL10 && EL2Enabled() &&
               ((!ELUsingAArch32(EL2) && HCR_EL2().DC == '1') ||
                  (ELUsingAArch32(EL2) && HCR().DC == '1'))) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
    end;

    var oa : FullAddress;
    oa.address = ZeroExtend{NUM_PABITS}(va);
    oa.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                          oa,
                                                          memattrs,
                                                          accdesc);

    return (fault, ipa);
end;
```

J1.3.4.25 AArch32_S1Enabled

```
// AArch32_S1Enabled()
// ===================
// Returns whether stage 1 translation is enabled for the active translation regime
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch32_S1Enabled(regime : Regime, ss : SecurityState) => boolean
begin
    if regime == Regime_EL2 then
        return HSCTLR().M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S().M == '1';
    elsif !AArch32_EL2Enabled(ss) then
        return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS().M else SCTLR().M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            return HCR().[TGE,DC] == '00' && SCTLR_NS().M == '1';
        else
            return HCR().[TGE,DC] == '00' && SCTLR().M == '1';
        end;
    else
        return EL2Enabled() && HCR_EL2().[TGE,DC] == '00' && SCTLR().M == '1';
    end;
end;
```

J1.3.4.26 AArch32_S1TranslateLD

```
// AArch32_S1TranslateLD()
// =======================
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

func AArch32_S1TranslateLD(fault_in : FaultRecord, regime : Regime,
                           va : bits(32), aligned : boolean,
                           accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin
    var fault : FaultRecord = fault_in;

    if !AArch32_S1Enabled(regime, accdesc.ss) then
        return AArch32_S1DisabledOutput(fault, regime, va, aligned, accdesc);
    end;

    let walkparams : S1TTWParams = AArch32_GetS1TTWParams(regime, va);

    if AArch32_VAIsOutOfRange(regime, walkparams, va) then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var walkstate : TTWState;
    (fault, walkstate) = AArch32_S1WalkLD(fault, regime, walkparams, accdesc, va);

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    if AArch32_S1HasAlignmentFaultDueToMemType(accdesc, aligned, walkparams.ntlsmd,
                                               walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif AArch32_S1LDHasPermissionsFault(regime, walkparams,
                                          walkstate.permissions,
                                          walkstate.memattrs.memtype,
                                          walkstate.baseaddress.paspace,
                                          accdesc) then
        fault.statuscode = Fault_Permission;
    end;

    if fault.statuscode != Fault_None then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var memattrs : MemoryAttributes;
    if ((accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || !AArch32_S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && !AArch32_S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;
    end;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR().VM
                                                            else HCR_EL2().VM) == '1' &&
        !(ImpDefBool("Apply effective shareability at stage 1"))) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);
    end;

    // Output Address
    let oa : FullAddress = StageOA(ZeroExtend{64}(va), walkparams.d128, walkparams.tgx, walkstate);
    let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                          oa, memattrs, accdesc);

    return (fault, ipa);
end;
```

J1.3.4.27 AArch32_S1TranslateSD

```
// AArch32_S1TranslateSD()
// =======================
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

func AArch32_S1TranslateSD(fault_in : FaultRecord, regime : Regime,
                           va : bits(32), aligned : boolean,
                           accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor, SDFType)
begin

    var fault : FaultRecord = fault_in;

    if !AArch32_S1Enabled(regime, accdesc.ss) then
        var ipa : AddressDescriptor;
        (fault, ipa) = AArch32_S1DisabledOutput(fault, regime, va, aligned, accdesc);
        return (fault, ipa, ARBITRARY : SDFType);
    end;

    var walkstate : TTWState;
    (fault, walkstate) = AArch32_S1WalkSD(fault, regime, accdesc, va);

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : SDFType);
    end;

    let domain : bits(2) = AArch32_OutputDomain(regime, walkstate.domain);
    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    var ntlsmd : bit;
    if IsFeatureImplemented(FEAT_LSMAOC) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        case regime of
            when Regime_EL30 => ntlsmd = SCTLR_S().nTLSMD;
            when Regime_EL10 =>
                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    ntlsmd = SCTLR_NS().nTLSMD;
                else
                    ntlsmd = SCTLR().nTLSMD;
                end;
        end;
    else
        ntlsmd = '1';
    end;

    if AArch32_S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif (! accdesc.acctype IN {AccessType_IC, AccessType_DC} &&
           domain == Domain_NoAccess) then
        fault.statuscode = Fault_Domain;
    elsif domain == Domain_Client then
        if AArch32_S1SDHasPermissionsFault(regime, walkstate.permissions,
                                           walkstate.memattrs.memtype,
                                           walkstate.baseaddress.paspace,
                                           accdesc) then
            fault.statuscode = Fault_Permission;
        end;
    end;

    if fault.statuscode != Fault_None then
        fault.domain = walkstate.domain;
        return (fault, ARBITRARY : AddressDescriptor, walkstate.sdftype);
    end;

    var memattrs : MemoryAttributes;
    if ((accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || !AArch32_S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && !AArch32_S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;
    end;
    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR().VM
                                                            else HCR_EL2().VM) == '1' &&
        !(ImpDefBool("Apply effective shareability at stage 1"))) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);
    end;
    // Output Address
    let oa : FullAddress = AArch32_SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
    let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                          oa, memattrs, accdesc);

    return (fault, ipa, walkstate.sdftype);
end;
```

J1.3.4.28 AArch32_S2Translate

```
// AArch32_S2Translate()
// =====================
// Perform a stage 2 translation mapping an IPA to a PA
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AArch32_S2Translate(fault_in : FaultRecord, ipa : AddressDescriptor,
                         aligned : boolean, accdesc : AccessDescriptor
                        ) => (FaultRecord, AddressDescriptor)
begin

    var fault : FaultRecord = fault_in;
    assert IsZero(ipa.paddress.address[NUM_PABITS-1:40]);
    if !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
        let s1aarch64 : boolean = FALSE;
        return AArch64_S2Translate(fault, ipa, s1aarch64, aligned, accdesc);
    end;
    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None;
    fault.secondstage = TRUE;
    fault.s2fs1walk   = accdesc.acctype == AccessType_TTW;
    fault.ipaddress   = ipa.paddress;

    let walkparams : S2TTWParams = AArch32_GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);
    end;

    if AArch32_IPAIsOutOfRange(walkparams, ipa.paddress.address[39:0]) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var walkstate : TTWState;
    (fault, walkstate) = AArch32_S2Walk(fault, walkparams, accdesc, ipa);

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if AArch32_S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif AArch32_S2HasPermissionsFault(walkparams,
                                        walkstate.permissions,
                                        walkstate.memattrs.memtype,
                                        accdesc) then
        fault.statuscode = Fault_Permission;
    end;
    var s2_memattrs : MemoryAttributes;
    if ((accdesc.acctype == AccessType_TTW &&
             walkstate.memattrs.memtype == MemType_Device) ||
        (accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || HCR2().ID == '1')) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && HCR2().CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;
    end;

    let s2aarch64 : boolean = FALSE;
    let memattrs : MemoryAttributes = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    let ipa_64 : bits(64) = ZeroExtend{}(ipa.paddress.address[39:0]);
    // Output Address
    let oa : FullAddress = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    let pa : AddressDescriptor = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (fault, pa);
end;
```

J1.3.4.29 AArch32_SDStageOA

```
// AArch32_SDStageOA()
// ===================
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

func AArch32_SDStageOA(baseaddress : FullAddress, va : bits(32), sdftype : SDFType) => FullAddress
begin
    let tsize : integer{} = SDFSize(sdftype);

    // Output Address
    var oa : FullAddress;
    oa.address = baseaddress.address[NUM_PABITS-1:tsize] :: va[tsize-1:0];
    oa.paspace = baseaddress.paspace;
    return oa;
end;
```

J1.3.4.30 AArch32_TranslateAddress

```
// AArch32_TranslateAddress()
// ==========================
// Main entry point for translating an address

func AArch32_TranslateAddress(va : bits(32), accdesc : AccessDescriptor,
                              aligned : boolean, size : integer) => AddressDescriptor
begin

    let regime : Regime = TranslationRegime(PSTATE.EL);
    if !RegimeUsingAArch32(regime) then
        return AArch64_TranslateAddress(ZeroExtend{64}(va), accdesc, aligned, size);
    end;

    var result : AddressDescriptor = AArch32_FullTranslate(va, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch32_CheckDebug(va, accdesc, size);
    end;

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend{64}(va);

    return result;
end;
```

J1.3.4.31 SDFSize

```
// SDFSize()
// =========
// Returns the short-descriptor format translation granule size

func SDFSize(sdftype : SDFType) => AddressSize
begin
    case sdftype of
        when SDFType_SmallPage =>    return 12;
        when SDFType_LargePage =>    return 16;
        when SDFType_Section =>      return 20;
        when SDFType_Supersection => return 24;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        otherwise => unreachable;
    end;
end;
```

J1.3.4.32 AArch32_DecodeDescriptorTypeLD

```
// AArch32_DecodeDescriptorTypeLD()
// ================================
// Determine whether the long-descriptor is a page, block or table

func AArch32_DecodeDescriptorTypeLD(descriptor : bits(64), level : integer) => DescriptorType
begin
    if descriptor[1:0] == '11' && level == FINAL_LEVEL then
        return DescriptorType_Leaf;
    elsif descriptor[1:0] == '11' then
        return DescriptorType_Table;
    elsif descriptor[1:0] == '01' && level != FINAL_LEVEL then
        return DescriptorType_Leaf;
    else
        return DescriptorType_Invalid;
    end;
end;
```

J1.3.4.33 AArch32_DecodeDescriptorTypeSD

```
// AArch32_DecodeDescriptorTypeSD()
// ================================
// Determine the type of the short-descriptor

func AArch32_DecodeDescriptorTypeSD(descriptor : bits(32), level : integer) => SDFType
begin
    if level == 1 && descriptor[1:0] == '01' then
        return SDFType_Table;
    elsif level == 1 && descriptor[18,1] == '01' then
        return SDFType_Section;
    elsif level == 1 && descriptor[18,1] == '11' then
        return SDFType_Supersection;
    elsif level == 2 && descriptor[1:0] == '01' then
        return SDFType_LargePage;
    elsif level == 2 && descriptor[1:0] == '1x' then
        return SDFType_SmallPage;
    else
        return SDFType_Invalid;
    end;
end;
```

J1.3.4.34 AArch32_S1IASize

```
// AArch32_S1IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 1 translation

func AArch32_S1IASize(txsz : bits(3)) => AddressSize
begin
    return 32 - UInt(txsz);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.35 AArch32_S1WalkLD

```
// AArch32_S1WalkLD()
// ==================
// Traverse stage 1 translation tables in long format to obtain the final descriptor

func AArch32_S1WalkLD(fault_in : FaultRecord, regime : Regime,
                      walkparams : S1TTWParams, accdesc : AccessDescriptor,
                      va : bits(32)) => (FaultRecord, TTWState)
begin
    var fault : FaultRecord = fault_in;
    var txsz : bits(3);
    var ttbr : bits(64);
    var epd : bit;
    var varange : VARange;
    if regime == Regime_EL2 then
        ttbr = HTTBR();
        txsz = walkparams.t0sz;
        varange = VARange_LOWER;
    else
        varange = AArch32_GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        var ttbr0 : bits(64);
        var ttbr1 : bits(64);
        var ttbcr : TTBCR_Type;
        if regime == Regime_EL30 then
            ttbcr = TTBCR_S();
            ttbr0 = TTBR0_S();
            ttbr1 = TTBR1_S();
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
            ttbcr = TTBCR_NS();
            ttbr0 = TTBR0_NS();
            ttbr1 = TTBR1_NS();
        else
            ttbcr = TTBCR();
            ttbr0 = TTBR0();
            ttbr1 = TTBR1();
        end;

        assert ttbcr.EAE == '1';
        if varange == VARange_LOWER then
            txsz = walkparams.t0sz;
            ttbr = ttbr0;
            epd  = ttbcr.EPD0;
        else
            txsz = walkparams.t1sz;
            ttbr = ttbr1;
            epd  = ttbcr.EPD1;
        end;
    end;

    if regime != Regime_EL2 && epd == '1' then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, ARBITRARY : TTWState);
    end;

    // Input Address size
    let iasize : AddressSize      = AArch32_S1IASize(txsz);
    let granulebits : integer{} = TGxGranuleBits(walkparams.tgx) as integer{12, 14, 16};
    let stride : integer{}      = granulebits - 3;
    let startlevel : integer  = FINAL_LEVEL - (((iasize-1) - granulebits) DIVRM stride);
    let levels : integer      = FINAL_LEVEL - startlevel;

    if !IsZero(ttbr[47:40]) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, ARBITRARY : TTWState);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

var baseaddress : FullAddress;
let baselsb = ((iasize - (levels*stride + granulebits)) + 3) as AddressSize;
baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend{NUM_PABITS}(ttbr[39:baselsb]::Zeros{baselsb});

var walkstate : TTWState;
walkstate.baseaddress = baseaddress;
walkstate.level       = startlevel;
walkstate.istable     = TRUE;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global
walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table  = '00';
walkstate.permissions.xn_table  = '0';
walkstate.permissions.pxn_table = '0';

var descriptor : bits(64);
var walkaddress : AddressDescriptor;

walkaddress.vaddress = ZeroExtend{64}(va);

if !AArch32_S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR().VM
                                                        else HCR_EL2().VM) == '1' &&
    !(ImpDefBool("Apply effective shareability at stage 1"))) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
end;

var desctype : DescriptorType;
var msb_residual : integer = iasize - 1;
repeat
    fault.level = walkstate.level;
    let indexlsb = ((FINAL_LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
    let indexmsb = msb_residual as AddressSize;
    let index : bits(40) = ZeroExtend{}(va[indexmsb:indexlsb]::'000');

    walkaddress.paddress.address = (walkstate.baseaddress.address OR
                                        ZeroExtend{NUM_PABITS}(index));
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    let toplevel : boolean = walkstate.level == startlevel;
    let walkaccess : AccessDescriptor = CreateAccDescS1TTW(toplevel, varange, accdesc);
    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) then
        let s2aligned : boolean = TRUE;
        let (s2fault, s2walkaddress) = AArch32_S2Translate(fault, walkaddress, s2aligned,
                                                           walkaccess);
        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault_None then
            return (s2fault, ARBITRARY : TTWState);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;

            (fault, descriptor) = FetchDescriptor{64}(walkparams.ee, s2walkaddress, walkaccess,
                                                      fault);
        else
            (fault, descriptor) = FetchDescriptor{64}(walkparams.ee, walkaddress, walkaccess,
                                                      fault);
        end;

        if fault.statuscode != Fault_None then
            return (fault, ARBITRARY : TTWState);
        end;

        desctype = AArch32_DecodeDescriptorTypeLD(descriptor, walkstate.level);

        case desctype of
            when DescriptorType_Table =>
                if !IsZero(descriptor[47:40]) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, ARBITRARY : TTWState);
                end;

                walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[39:12]::
                                                                        Zeros{12});
                if walkstate.baseaddress.paspace == PAS_Secure && descriptor[63] == '1' then
                    walkstate.baseaddress.paspace = PAS_NonSecure;
                end;
                if walkparams.hpd == '0' then
                    walkstate.permissions.xn_table  = (walkstate.permissions.xn_table  OR
                                                       descriptor[60]);
                    walkstate.permissions.ap_table  = (walkstate.permissions.ap_table  OR
                                                       descriptor[62:61]);
                    walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                                       descriptor[59]);
                end;
                walkstate.level = walkstate.level + 1;
                msb_residual = indexlsb - 1;

            when DescriptorType_Invalid =>
                fault.statuscode = Fault_Translation;
                return (fault, ARBITRARY : TTWState);

            when DescriptorType_Leaf =>
                walkstate.istable = FALSE;
        end;

    until desctype == DescriptorType_Leaf looplimit 3;

    // Check the output address is inside the supported range
    if !IsZero(descriptor[47:40]) then
        fault.statuscode = Fault_AddressSize;
        return (fault, ARBITRARY : TTWState);
    end;

    // Check the access flag
    if descriptor[10] == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, ARBITRARY : TTWState);
    end;

    walkstate.permissions.xn  = descriptor[54];
    walkstate.permissions.pxn = descriptor[53];
    walkstate.permissions.ap  = descriptor[7:6]::'1';
    walkstate.contiguous      = descriptor[52];
    if regime == Regime_EL2 then
        // All EL2 regime accesses are treated as Global
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        walkstate.nG = '0';
    elsif accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        // When a PE is using the Long-descriptor translation table format,
        // and is in Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk.
        walkstate.nG = '1';
    else
        walkstate.nG = descriptor[11];
    end;

    let indexlsb = ((FINAL_LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
    walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[39:indexlsb]::
                                                            Zeros{indexlsb});
    if walkstate.baseaddress.paspace == PAS_Secure && descriptor[5] == '1' then
        walkstate.baseaddress.paspace = PAS_NonSecure;
    end;

    let memattr : bits(3) = descriptor[4:2];
    let sh : bits(2)      = descriptor[9:8];
    let attr : bits(8)    = AArch32_MAIRAttr(UInt(memattr), walkparams.mair);
    let s1aarch64 : boolean = FALSE;
    walkstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64);

    return (fault, walkstate);
end;
```

J1.3.4.36 AArch32_S1WalkSD

```
// AArch32_S1WalkSD()
// ==================
// Traverse stage 1 translation tables in short format to obtain the final descriptor

func AArch32_S1WalkSD(fault_in : FaultRecord, regime : Regime,
                      accdesc : AccessDescriptor, va : bits(32)) => (FaultRecord, TTWState)
begin
    var fault : FaultRecord = fault_in;
    var sctlr : SCTLR_Type;
    var ttbcr : TTBCR_Type;
    var ttbr0 : TTBR0_Type;
    var ttbr1 : TTBR1_Type;
    // Determine correct translation control registers to use.
    if regime == Regime_EL30 then
        sctlr = SCTLR_S();
        ttbcr = TTBCR_S();
        ttbr0 = TTBR0_S();
        ttbr1 = TTBR1_S();
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        sctlr = SCTLR_NS();
        ttbcr = TTBCR_NS();
        ttbr0 = TTBR0_NS();
        ttbr1 = TTBR1_NS();
    else
        sctlr = SCTLR();
        ttbcr = TTBCR();
        ttbr0 = TTBR0();
        ttbr1 = TTBR1();
    end;

    assert ttbcr.EAE == '0';
    let ee : bit  = sctlr.EE;
    let afe : bit = sctlr.AFE;
    let tre : bit = sctlr.TRE;
    let ttbcr_n : integer{} = UInt(ttbcr.N);
    let varange : VARange = (if ttbcr_n == 0 || IsZero(va[31:(32-ttbcr_n)]) then VARange_LOWER
                             else VARange_UPPER);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let n : integer{} = if varange == VARange_LOWER then ttbcr_n else 0;
var ttb : bits(32);
var pd : bits(1);
var irgn : bits(2);
var rgn : bits(2);
var s : bits(1);
var nos : bits(1);
if varange == VARange_LOWER then
    ttb  = ttbr0.TTB0::Zeros{7};
    pd   = ttbcr.PD0;
    irgn = ttbr0.IRGN;
    rgn  = ttbr0.RGN;
    s    = ttbr0.S;
    nos  = ttbr0.NOS;
else
    ttb  = ttbr1.TTB1::Zeros{7};
    pd   = ttbcr.PD1;
    irgn = ttbr1.IRGN;
    rgn  = ttbr1.RGN;
    s    = ttbr1.S;
    nos  = ttbr1.NOS;
end;

// Check if Translation table walk disabled for translations with this Base register.
if pd == '1' then
    fault.level      = 1;
    fault.statuscode = Fault_Translation;
    return (fault, ARBITRARY : TTWState);
end;

var baseaddress : FullAddress;
baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend{NUM_PABITS}(ttb[31:14-n]::Zeros{14-n});

let startlevel : integer = 1;
var walkstate : TTWState;
walkstate.baseaddress = baseaddress;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global. Translations in Short-Descriptor Format
// always support global & non-global translations.
walkstate.nG          = '1';
walkstate.memattrs    = WalkMemAttrs(s::nos, irgn, rgn);
walkstate.level       = startlevel;
walkstate.istable     = TRUE;

var domain : bits(4);
var descriptor : bits(32);
var walkaddress : AddressDescriptor;

walkaddress.vaddress = ZeroExtend{64}(va);

if !AArch32_S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR().VM
                                                        else HCR_EL2().VM) == '1' &&
    !(ImpDefBool("Apply effective shareability at stage 1"))) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
end;

var nG : bit;
var ns : bit;
var pxn : bit;
var ap : bits(3);
var tex : bits(3);
var c : bit;
var b : bit;
var xn : bit;
repeat
    fault.level = walkstate.level;

    var index : bits(32);
    if walkstate.level == 1 then
        index = ZeroExtend{32}(va[31-n:20]::'00');
    else
        index = ZeroExtend{32}(va[19:12]::'00');
    end;

    walkaddress.paddress.address = (walkstate.baseaddress.address OR
                                        ZeroExtend{NUM_PABITS}(index));
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    let toplevel : boolean = walkstate.level == startlevel;
    let walkaccess : AccessDescriptor = CreateAccDescS1TTW(toplevel, varange, accdesc);
    if regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) then
        let s2aligned : boolean = TRUE;
        let (s2fault, s2walkaddress) = AArch32_S2Translate(fault, walkaddress, s2aligned,
                                                           walkaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, ARBITRARY : TTWState);
        end;

        (fault, descriptor) = FetchDescriptor{32}(ee, s2walkaddress, walkaccess, fault);
    else
        (fault, descriptor) = FetchDescriptor{32}(ee, walkaddress, walkaccess, fault);
    end;

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : TTWState);
    end;

    walkstate.sdftype = AArch32_DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDFType_Invalid =>
            fault.domain     = domain;
            fault.statuscode = Fault_Translation;
            return (fault, ARBITRARY : TTWState);

        when SDFType_Table =>
            domain = descriptor[8:5];
            ns     = descriptor[3];
            pxn    = descriptor[2];

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[31:10]::
                                                                   Zeros{10});
            walkstate.level = 2;

        when SDFType_SmallPage =>
            nG  = descriptor[11];
            s   = descriptor[10];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ap  = descriptor[9,5:4];
            tex = descriptor[8:6];
            c   = descriptor[3];
            b   = descriptor[2];
            xn  = descriptor[0];

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[31:12]::
                                                                    Zeros{12});
            walkstate.istable = FALSE;

        when SDFType_LargePage =>
            xn  = descriptor[15];
            tex = descriptor[14:12];
            nG  = descriptor[11];
            s   = descriptor[10];
            ap  = descriptor[9,5:4];
            c   = descriptor[3];
            b   = descriptor[2];

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[31:16]::
                                                                    Zeros{16});
            walkstate.istable = FALSE;

        when SDFType_Section =>
            ns     = descriptor[19];
            nG     = descriptor[17];
            s      = descriptor[16];
            ap     = descriptor[15,11:10];
            tex    = descriptor[14:12];
            domain = descriptor[8:5];
            xn     = descriptor[4];
            c      = descriptor[3];
            b      = descriptor[2];
            pxn    = descriptor[0];

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[31:20]::
                                                                    Zeros{20});
            walkstate.istable = FALSE;

        when SDFType_Supersection =>
            ns     = descriptor[19];
            nG     = descriptor[17];
            s      = descriptor[16];
            ap     = descriptor[15,11:10];
            tex    = descriptor[14:12];
            xn     = descriptor[4];
            c      = descriptor[3];
            b      = descriptor[2];
            pxn    = descriptor[0];
            domain = '0000';

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[8:5,23:20,31:24]::
                                                                    Zeros{24});

            walkstate.istable = FALSE;
    end;

until walkstate.sdftype != SDFType_Table looplimit 2;

if afe == '1' && ap[0] == '0' then
    fault.domain     = domain;
    fault.statuscode = Fault_AccessFlag;
    return (fault, ARBITRARY : TTWState);
end;

// Decode the TEX, C, B and S bits to produce target memory attributes
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if tre == '1' then
        walkstate.memattrs = AArch32_RemappedTEXDecode(regime, tex, c, b, s);
    elsif RemapRegsHaveResetValues() then
        walkstate.memattrs = AArch32_DefaultTEXDecode(tex, c, b, s);
    else
        walkstate.memattrs = ImpDefMemoryAttributes("IMPLEMENTATION_DEFINED");
    end;

    walkstate.permissions.ap  = ap;
    walkstate.permissions.xn  = xn;
    walkstate.permissions.pxn = pxn;
    walkstate.domain = domain;
    walkstate.nG     = nG;

    if accdesc.ss == SS_Secure && ns == '0' then
        walkstate.baseaddress.paspace = PAS_Secure;
    else
        walkstate.baseaddress.paspace = PAS_NonSecure;
    end;


    return (fault, walkstate);
end;
```

J1.3.4.37 AArch32_S2IASize

```
// AArch32_S2IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 2 translation

func AArch32_S2IASize(t0sz : bits(4)) => AddressSize
begin
    return 32 - SInt(t0sz);
end;
```

J1.3.4.38 AArch32_S2StartLevel

```
// AArch32_S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

func AArch32_S2StartLevel(sl0 : bits(2)) => integer
begin
    return 2 - UInt(sl0);
end;
```

J1.3.4.39 AArch32_S2Walk

```
// AArch32_S2Walk()
// ================
// Traverse stage 2 translation tables in long format to obtain the final descriptor

func AArch32_S2Walk(fault_in : FaultRecord, walkparams : S2TTWParams,
                    accdesc : AccessDescriptor, ipa : AddressDescriptor) => (FaultRecord, TTWState)
begin
    var fault : FaultRecord = fault_in;

    if walkparams.sl0 == '1x' || AArch32_S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, ARBITRARY : TTWState);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Input Address size
let iasize : AddressSize      = AArch32_S2IASize(walkparams.t0sz);
let startlevel : integer  = AArch32_S2StartLevel(walkparams.sl0);
let levels : integer      = FINAL_LEVEL - startlevel;
let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
let stride : integer{}      = granulebits - 3;

if !IsZero(VTTBR()[47:40]) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, ARBITRARY : TTWState);
end;

var baseaddress : FullAddress;
let baselsb = ((iasize - (levels*stride + granulebits)) + 3) as AddressSize;
baseaddress.paspace = PAS_NonSecure;
baseaddress.address = ZeroExtend{NUM_PABITS}(VTTBR()[39:baselsb]::Zeros{baselsb});

var walkstate : TTWState;
walkstate.baseaddress = baseaddress;
walkstate.level       = startlevel;
walkstate.istable     = TRUE;
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                     walkparams.orgn);

var descriptor : bits(64);
let walkaccess : AccessDescriptor = CreateAccDescS2TTW(accdesc);
var walkaddress : AddressDescriptor;

walkaddress.vaddress = ipa.vaddress;

if HCR2().CD == '1' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

var msb_residual : integer = iasize - 1;
var desctype : DescriptorType;
repeat
    fault.level = walkstate.level;

    let indexlsb = ((FINAL_LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
    let indexmsb = msb_residual as AddressSize;
    let index : bits(40) = ZeroExtend{}(ipa.paddress.address[indexmsb:indexlsb]::'000');

    walkaddress.paddress.address = (walkstate.baseaddress.address OR
                                    ZeroExtend{NUM_PABITS}(index));
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    (fault, descriptor) = FetchDescriptor{64}(walkparams.ee, walkaddress, walkaccess, fault);

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : TTWState);
    end;

    desctype = AArch32_DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case desctype of
        when DescriptorType_Table =>
            if !IsZero(descriptor[47:40]) then
                fault.statuscode = Fault_AddressSize;
                return (fault, ARBITRARY : TTWState);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                end;

                walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[39:12]::
                                                                        Zeros{12});
                walkstate.level = walkstate.level + 1;
                msb_residual = indexlsb - 1;

            when DescriptorType_Invalid =>
                fault.statuscode = Fault_Translation;
                return (fault, ARBITRARY : TTWState);

            when DescriptorType_Leaf =>
                walkstate.istable = FALSE;
        end;

    until desctype == DescriptorType_Leaf looplimit 3;

    // Check the output address is inside the supported range
    if !IsZero(descriptor[47:40]) then
        fault.statuscode = Fault_AddressSize;
        return (fault, ARBITRARY : TTWState);
    end;

    // Check the access flag
    if descriptor[10] == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, ARBITRARY : TTWState);
    end;

    // Unpack the descriptor into address and upper and lower block attributes
    let indexlsb = ((FINAL_LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
    walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[39:indexlsb]::
                                                            Zeros{indexlsb});

    walkstate.permissions.s2ap = descriptor[7:6];
    walkstate.permissions.s2xn = descriptor[54];
    if IsFeatureImplemented(FEAT_XNX) then
        walkstate.permissions.s2xnx = descriptor[53];
    else
        walkstate.permissions.s2xnx = '0';
    end;

    let memattr : bits(4) = descriptor[5:2];
    let sh : bits(2)      = descriptor[9:8];
    let s2aarch64 : boolean = FALSE;
    walkstate.memattrs   = S2DecodeMemAttrs(memattr, sh, s2aarch64);
    walkstate.contiguous = descriptor[52];


    return (fault, walkstate);
end;
```

J1.3.4.40 RemapRegsHaveResetValues

```
// RemapRegsHaveResetValues()
// ==========================

impdef func RemapRegsHaveResetValues() => boolean
begin
    return TRUE;
end;
```

J1.3.4.41 AArch32_GetS1TTWParams

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32_GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

func AArch32_GetS1TTWParams(regime : Regime, va : bits(32)) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    case regime of
        when Regime_EL2  => walkparams = AArch32_S1TTWParamsEL2();
        when Regime_EL10 => walkparams = AArch32_S1TTWParamsEL10(va);
        when Regime_EL30 => walkparams = AArch32_S1TTWParamsEL30(va);
    end;

    return walkparams;
end;
```

J1.3.4.42 AArch32_GetS2TTWParams

```
// AArch32_GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

func AArch32_GetS2TTWParams() => S2TTWParams
begin
    var walkparams : S2TTWParams;

    walkparams.tgx  = TGx_4KB;
    walkparams.s    = VTCR().S;
    walkparams.t0sz = VTCR().T0SZ;
    walkparams.sl0  = VTCR().SL0;
    walkparams.irgn = VTCR().IRGN0;
    walkparams.orgn = VTCR().ORGN0;
    walkparams.sh   = VTCR().SH0;
    walkparams.ee   = HSCTLR().EE;
    walkparams.ptw  = HCR().PTW;
    walkparams.vm   = HCR().VM OR HCR().DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz[3] then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits{4}(Unpredictable_RESVTCRS);
    end;

    return walkparams;
end;
```

J1.3.4.43 AArch32_GetVARange

```
// AArch32_GetVARange()
// ====================
// Select the translation base address for stage 1 long-descriptor walks

func AArch32_GetVARange(va : bits(32), t0sz : bits(3), t1sz : bits(3)) => VARange
begin
    // Lower range Input Address size
    let lo_iasize : AddressSize = AArch32_S1IASize(t0sz);
    // Upper range Input Address size
    let up_iasize : AddressSize = AArch32_S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va[31:lo_iasize]) then VARange_LOWER else VARange_UPPER;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif t0sz == '000' then
        return if IsOnes(va[31:up_iasize]) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va[31:lo_iasize]) then
        return VARange_LOWER;
    elsif IsOnes(va[31:up_iasize]) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return ARBITRARY : VARange;
    end;
end;
```

J1.3.4.44 AArch32_S1DCacheEnabled

```
// AArch32_S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

func AArch32_S1DCacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL30 => return SCTLR_S().C == '1';
        when Regime_EL2  => return HSCTLR().C == '1';
        when Regime_EL10 =>
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS().C else SCTLR().C) == '1';
    end;
end;
```

J1.3.4.45 AArch32_S1ICacheEnabled

```
// AArch32_S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

func AArch32_S1ICacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL30 => return SCTLR_S().I == '1';
        when Regime_EL2  => return HSCTLR().I == '1';
        when Regime_EL10 =>
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS().I else SCTLR().I) == '1';
    end;
end;
```

J1.3.4.46 AArch32_S1TTWParamsEL10

```
// AArch32_S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

func AArch32_S1TTWParamsEL10(va : bits(32)) => S1TTWParams
begin
    var mair : bits(64);
    var sif : bit;
    var ttbcr : TTBCR_Type;
    var ttbcr2 : TTBCR2_Type;
    var sctlr : SCTLR_Type;

    if ELUsingAArch32(EL3) then
        ttbcr  = TTBCR_NS();
        ttbcr2 = TTBCR2_NS();
        sctlr  = SCTLR_NS();
        mair   = MAIR1_NS()::MAIR0_NS();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        sif    = SCR().SIF;
    else
        ttbcr  = TTBCR();
        ttbcr2 = TTBCR2();
        sctlr  = SCTLR();
        mair   = MAIR1()::MAIR0();
        sif    = if HaveEL(EL3) then SCR_EL3().SIF else '0';
    end;

    assert ttbcr.EAE == '1';
    var walkparams : S1TTWParams;

    walkparams.t0sz   = ttbcr.T0SZ;
    walkparams.t1sz   = ttbcr.T1SZ;
    walkparams.ee     = sctlr.EE;
    walkparams.wxn    = sctlr.WXN;
    walkparams.uwxn   = sctlr.UWXN;
    walkparams.ntlsmd = if IsFeatureImplemented(FEAT_LSMAOC) then sctlr.nTLSMD else '1';
    walkparams.mair   = mair;
    walkparams.sif    = sif;

    let varange : VARange = AArch32_GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGN0;
        walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD0
                           else '0');
    else
        walkparams.sh   = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGN1;
        walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD1
                           else '0');
    end;

    return walkparams;
end;
```

J1.3.4.47 AArch32_S1TTWParamsEL2

```
// AArch32_S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime

func AArch32_S1TTWParamsEL2() => S1TTWParams
begin
    var walkparams : S1TTWParams;

    walkparams.tgx  = TGx_4KB;
    walkparams.t0sz = HTCR().T0SZ;
    walkparams.irgn = HTCR().IRGN0;
    walkparams.orgn = HTCR().ORGN0;
    walkparams.sh   = HTCR().SH0;
    walkparams.hpd  = if IsFeatureImplemented(FEAT_AA32HPD) then HTCR().HPD else '0';
    walkparams.ee   = HSCTLR().EE;
    walkparams.wxn  = HSCTLR().WXN;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = HSCTLR().nTLSMD;
    else
        walkparams.ntlsmd = '1';
    end;

    walkparams.mair = HMAIR1()::HMAIR0();

    return walkparams;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.3.4.48 AArch32_S1TTWParamsEL30

```
// AArch32_S1TTWParamsEL30()
// =========================
// Gather stage 1 translation table walk parameters for EL3&0 regime

func AArch32_S1TTWParamsEL30(va : bits(32)) => S1TTWParams
begin
    assert TTBCR_S().EAE == '1';
    var walkparams : S1TTWParams;

    walkparams.t0sz   = TTBCR_S().T0SZ;
    walkparams.t1sz   = TTBCR_S().T1SZ;
    walkparams.ee     = SCTLR_S().EE;
    walkparams.wxn    = SCTLR_S().WXN;
    walkparams.uwxn   = SCTLR_S().UWXN;
    walkparams.ntlsmd = if IsFeatureImplemented(FEAT_LSMAOC) then SCTLR_S().nTLSMD else '1';
    walkparams.mair   = MAIR1_S()::MAIR0_S();
    walkparams.sif    = SCR().SIF;

    let varange : VARange = AArch32_GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = TTBCR_S().SH0;
        walkparams.irgn = TTBCR_S().IRGN0;
        walkparams.orgn = TTBCR_S().ORGN0;
        walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD)
                           then TTBCR_S().T2E AND TTBCR2_S().HPD0
                           else '0');
    else
        walkparams.sh   = TTBCR_S().SH1;
        walkparams.irgn = TTBCR_S().IRGN1;
        walkparams.orgn = TTBCR_S().ORGN1;
        walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD)
                           then TTBCR_S().T2E AND TTBCR2_S().HPD1
                           else '0');
    end;

    return walkparams;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a `FunctionName`, without an `AArch64.` or `AArch32.` prefix. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

- shared/debug.
- shared/exceptions.
- shared/functions.
- shared/trace.
- shared/translation.

J1.4.1 shared/debug

This section includes the following pseudocode functions:

- ClearStickyErrors
- DebugTarget
- DebugTargetFrom
- DoubleLockStatus
- OSLockStatus
- Component
- GetAccessComponent
- SoftwareLockStatus
- IsG1ActivityMonitorImplemented
- IsG1ActivityMonitorOffsetImplemented
- AllowExternalDebugAccess
- AllowExternalPMSSAccess
- AllowExternalPMUAccess
- AllowExternalTraceAccess
- AllowRASMemoryMappedAccess
- Debug
- ExternalInvasiveDebugEnabled
- ExternalNoninvasiveDebugAllowed
- ExternalNoninvasiveDebugEnabled
- ExternalRealmInvasiveDebugEnabled
- ExternalRealmNoninvasiveDebugEnabled
- ExternalRootInvasiveDebugEnabled
- ExternalRootNoninvasiveDebugEnabled
- ExternalSecureInvasiveDebugEnabled
- ExternalSecureNoninvasiveDebugEnabled
- InvasiveDebugPermittedPAS
- IsAccessNonSecure
- IsAccessRealm
- IsAccessRoot
- IsAccessSecure
- IsCorePowered

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- IsMostSecureAccess
- IsPASValid
- BreakpointInfo
- BreakpointType
- CheckValidStateMatch
- ContextAwareBreakpointRange
- IsContextAwareBreakpoint
- NumBreakpointsImplemented
- NumContextAwareBreakpointsImplemented
- NumWatchpointsImplemented
- CTI_ProcessEvent
- CTI_SetEventLevel
- CTI_SignalEvent
- CrossTrigger
- CheckForDCCInterrupts
- DTR
- Read_DBGDTRRX_EL0
- Read_DBGDTRTX_EL0
- Read_DBGDTR_EL0
- Write_DBGDTRRX_EL0
- Write_DBGDTRTX_EL0
- Write_DBGDTR_EL0
- Write_EDITR
- DCPSInstruction
- DRPSInstruction
- DebugHalt
- DebugRestorePSR
- DisableITRAndResumeInstructionPrefetch
- ExecuteA64
- ExecuteT32
- ExitDebugState
- Halt
- HaltOnBreakpointOrWatchpoint
- Halted
- HaltingAllowed
- Restarting
- StopInstructionPrefetchAndEnableITR
- UpdateDbgAuthStatus
- UpdateEDHSR
- UpdateEDSCRFields
- haltingevents
- CheckEDBGRQ
- CheckExceptionCatch
- CheckExternalDebugRequestEvents
- CheckHaltingStep
- CheckOSUnlockCatch
- CheckPMUHalt

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckPendingExceptionCatch
- CheckPendingOSUnlockCatch
- CheckPendingResetCatch
- CheckResetCatch
- CheckSoftwareAccessToDebugRegisters
- CheckTRBEHalt
- ExternalDebugRequest
- HSAdvance
- HaltingStep_DidNotStep
- HaltingStep_SteppedEX
- ExternalDebugInterruptsDisabled
- pmu
- CheckForPMUOverflow
- CheckPMUOverflowCondition
- ClearEventCounters
- CountPMUEvents
- EffectiveEPMN
- EffectiveHPMN
- GetNumEventCountersAccessible
- GetNumEventCountersSelfHosted
- GetPMUAccessMask
- GetPMUCounterRange
- GetPMUReadMask
- GetPMUWriteMask
- HasElapsed64Cycles
- IncrementInstructionCounter
- IsRange3Counter
- PMUCaptureEvent
- PMUCaptureEventAllowed
- PMUCaptureEventEnabled
- PMUCountValue
- PMUCounterRange
- PMUEvent
- PMUOverflowCondition
- PMUSwIncrement
- ReservedPMUThreshold
- SMEPMUEventPredicate
- SVEPMUEventPredicate
- ShouldPMUFreeze
- ZeroCycleCounter
- ZeroPMUCounters
- CreatePCSample
- PCSRSuspended
- PCSample
- Read_EDPCSRlo
- Read_PMPCSR
- SetPCSRActive

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SetPCSRUnknown
- SetPCSample
- CheckSoftwareStep
- DebugExceptionReturnSS
- SSAdvance
- SoftwareStepOpEnabled
- SoftwareStep_DidNotStep
- SoftwareStep_SteppedEX
- DataCacheWatchpointSize
- WatchpointInfo
- WatchpointType

J1.4.1.1 ClearStickyErrors

```
// ClearStickyErrors()
// ===================

func ClearStickyErrors()
begin
    EDSCR().TXU = '0';            // Clear TX underrun flag
    EDSCR().RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR().ITO = '0';        // Clear ITR overrun flag
    end;

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted() && EDSCR().ITE == '0' &&
          ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO)) then
        return;
    end;
    EDSCR().ERR = '0';            // Clear cumulative error flag

    return;
end;
```

J1.4.1.2 DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

func DebugTarget() => bits(2)
begin
    let ss : SecurityState = CurrentSecurityState();
    return DebugTargetFrom(ss);
end;
```

J1.4.1.3 DebugTargetFrom

```
// DebugTargetFrom()
// =================

func DebugTargetFrom(from_state : SecurityState) => bits(2)
begin
    var route_to_el2 : boolean;
    if HaveEL(EL2) && (from_state != SS_Secure ||
        (IsFeatureImplemented(FEAT_SEL2) && (!HaveEL(EL3) || SCR_EL3().EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            route_to_el2 = (HDCR().TDE == '1' || HCR().TGE == '1');
        else
            route_to_el2 = (MDCR_EL2().TDE == '1' || HCR_EL2().TGE == '1');
        end;
    else
        route_to_el2 = FALSE;
    end;

    var target : bits(2);
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;
    end;

    return target;
end;
```

J1.4.1.4 DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

func DoubleLockStatus() => boolean
begin
    if !IsFeatureImplemented(FEAT_DoubleLock) then
        return FALSE;
    end;
    if ELUsingAArch32(EL1) then
        return DBGOSDLR().DLK == '1' && DBGPRCR().CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1().DLK == '1' && DBGPRCR_EL1().CORENPDRQ == '0' && !Halted();
    end;
end;
```

J1.4.1.5 OSLockStatus

```
// OSLockStatus()
// ==============
// Returns the state of the OS Lock.

readonly func OSLockStatus() => boolean
begin
    return (if ELUsingAArch32(EL1) then DBGOSLSR().OSLK else OSLSR_EL1().OSLK) == '1';
end;
```

J1.4.1.6 Component

```
// Component
// =========
// Component Types.

type Component of enumeration {
        Component_AMU,
        Component_ETE,
        Component_TRBE,
        Component_RAS,
        Component_GIC,
        Component_PMU,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Component_Debug,
        Component_CTI
};
```

J1.4.1.7 GetAccessComponent

```
// GetAccessComponent()
// ====================
// Determines the component by decoding the physical address in the address descriptor.

impdef func GetAccessComponent(addrdesc : AddressDescriptor) => Component
begin
    return Component_Debug;
end;
```

J1.4.1.8 SoftwareLockStatus

```
// SoftwareLockStatus()
// ====================
// Returns the state of the Software Lock.

func SoftwareLockStatus(addrdesc : AddressDescriptor) => boolean
begin
    let component : Component = GetAccessComponent(addrdesc);
    if !HaveSoftwareLock(component) then
        return FALSE;
    end;
    case component of
        when Component_ETE   => return TRCLSR().SLK == '1';
        when Component_Debug => return EDLSR().SLK == '1';
        when Component_PMU   => return PMLSR().SLK == '1';
        when Component_CTI   => return CTILSR().SLK == '1';
        otherwise            => return FALSE;
    end;
end;
```

J1.4.1.9 IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// ================================
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

impdef func IsG1ActivityMonitorImplemented(i : integer) => boolean
begin
    case i of
        when 0 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 0"));
        when 1 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 1"));
        when 2 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 2"));
        when 3 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 3"));
        when 4 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 4"));
        when 5 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 5"));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when 6 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 6"));
        when 7 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 7"));
        when 8 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 8"));
        when 9 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 9"));
        when 10 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 10"));
        when 11 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 11"));
        when 12 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 12"));
        when 13 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 13"));
        when 14 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 14"));
        when 15 =>
            return (ImpDefBool(
                    "G1 activity monitor is implemented for counter 15"));
        otherwise =>
            return FALSE;
    end;
end;
```

J1.4.1.10 IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// ======================================
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

impdef func IsG1ActivityMonitorOffsetImplemented(i : integer) => boolean
begin
    case i of
        when 0 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 0"));
        when 1 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 1"));
        when 2 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 2"));
        when 3 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 3"));
        when 4 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 4"));
        when 5 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 5"));
        when 6 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 6"));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when 7 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 7"));
        when 8 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 8"));
        when 9 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 9"));
        when 10 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 10"));
        when 11 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 11"));
        when 12 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 12"));
        when 13 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 13"));
        when 14 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 14"));
        when 15 =>
            return (ImpDefBool(
                    "G1 activity monitor offset is implemented for counter 15"));
        otherwise =>
            return FALSE;
    end;
end;
```

J1.4.1.11 AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External DBGBVR<n>_EL1,
// DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1 registers, and, from Armv8.2, the
// OSLAR_EL1 register is allowed for the access. Returns FALSE otherwise.

func AllowExternalDebugAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3().[EDADE,EDAD] of
            when '00' => return TRUE;
            when '01' => return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
            when '10' => return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
            when '11' => return addrdesc.paddress.paspace == PAS_Root;
        end;
    end;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS_Secure then return TRUE; end;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE; end;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE; end;
    end;

    if HaveEL(EL3) then
        let EDAD_bit : bit = if ELUsingAArch32(EL3) then SDCR().EDAD else MDCR_EL3().EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.12 AllowExternalPMSSAccess

```
// AllowExternalPMSSAccess()
// =========================
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed for the given Security state, FALSE otherwise.

func AllowExternalPMSSAccess(addrdesc : AddressDescriptor) => boolean
begin
    assert IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64();
    // FEAT_Debugv8p4 is always implemented when FEAT_PMUv3_SS is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    // The access may also be subject to the OS Double Lock, power-down, etc.
    var epmssad : bits(2) = if HaveEL(EL3) then MDCR_EL3().EPMSSAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && epmssad IN {'01','10'} then
        (-, epmssad) = ConstrainUnpredictableBits{2}(Unpredictable_RESEPMSSAD);
        // The value returned by ConstrainUnpredictableBits() must be a
        // non-reserved value
        assert epmssad IN {'00','11'};
    end;

    case epmssad of
        when '00' =>
            if IsFeatureImplemented(FEAT_RME) then
                return addrdesc.paddress.paspace == PAS_Root;
            else
                return addrdesc.paddress.paspace == PAS_Secure;
            end;
        when '01' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
        when '10' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
        when '11' =>
            return TRUE;
    end;
end;
```

J1.4.1.13 AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

func AllowExternalPMUAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3().[EPMADE,EPMAD] of
            when '00' => return TRUE;
            when '01' => return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
            when '10' => return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
            when '11' => return addrdesc.paddress.paspace == PAS_Root;
        end;
    end;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS_Secure then return TRUE; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE; end;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE; end;
    end;

    if HaveEL(EL3) then
        let EPMAD_bit : bit = if ELUsingAArch32(EL3) then SDCR().EPMAD else MDCR_EL3().EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
    end;
end;
```

J1.4.1.14 AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// ==========================
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given PAS, FALSE otherwise.

func AllowExternalTraceAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS lock, power-down, etc.
    if !IsFeatureImplemented(FEAT_TRBE) then return TRUE; end;
    assert IsFeatureImplemented(FEAT_Debugv8p4);
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3().[ETADE,ETAD] of
            when '00' => return TRUE;
            when '01' => return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
            when '10' => return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
            when '11' => return addrdesc.paddress.paspace == PAS_Root;
        end;
    end;

    if addrdesc.paddress.paspace == PAS_Secure then return TRUE; end;
    if HaveEL(EL3) then
        // External Trace access is not supported for EL3 using AArch32
        assert !ELUsingAArch32(EL3);
        return MDCR_EL3().ETAD == '0';
    else
        return NonSecureOnlyImplementation();
    end;
end;
```

J1.4.1.15 AllowRASMemoryMappedAccess

```
// AllowRASMemoryMappedAccess()
// ============================
// Returns TRUE if read and/or write access to RAS memory-mapped registers is
// permitted for the given access.

func AllowRASMemoryMappedAccess(addrdesc : AddressDescriptor, iswrite : boolean) => boolean
begin
    // An access from the most secure implemented PAS is always permitted.
    if IsMostSecureAccess(addrdesc) then
        return TRUE;
    end;

    if IsFeatureImplemented(FEAT_RASSA_ACR) && IsFeatureImplemented(FEAT_RASSAv2) then
        var ra : bits(2);
        let implements_rl_and_s : boolean = (IsFeatureImplemented(FEAT_RME) &&
                                             ImpDefBool(
              "the error record group allows configuration of Secure and Realm register accesses"));
        if IsAccessRealm(addrdesc) then
            ra = if implements_rl_and_s then ERRACR().RLRA else ERRACR().NSRA;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsAccessSecure(addrdesc) then
            ra = if implements_rl_and_s then ERRACR().SRA else ERRACR().NSRA;
        elsif IsAccessNonSecure(addrdesc) then
            ra = ERRACR().NSRA;
        else
            unreachable;
        end;
        // '10' is reserved
        if ra == '10' then (-, ra) = ConstrainUnpredictableBits{2}(Unpredictable_RES_RA); end;
        // Interpret configured permissions to a boolean for the requested access
        case ra of
            when '00' => return FALSE;          // RAZ/WI => no read or write
            when '01' => return !iswrite;       // RO => reads allowed, writes not
            when '11' => return TRUE;           // RW => both allowed
            otherwise => unreachable;
        end;
    end;
    return ImpDefBool("error records are accessible to all PAS");
end;
```

J1.4.1.16 Debug

```
// Debug authentication signals
// ============================

var DBGEN : Signal;
var NIDEN : Signal;
var SPIDEN : Signal;
var SPNIDEN : Signal;
var RLPIDEN : Signal;
var RTPIDEN : Signal;
```

J1.4.1.17 ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

readonly func ExternalInvasiveDebugEnabled() => boolean
begin
    return DBGEN == HIGH;
end;
```

J1.4.1.18 ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

func ExternalNoninvasiveDebugAllowed() => boolean
begin
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);
end;

// ExternalNoninvasiveDebugAllowed()
// =================================

func ExternalNoninvasiveDebugAllowed(el : bits(2)) => boolean
begin
    if !ExternalNoninvasiveDebugEnabled() then return FALSE; end;
    let ss : SecurityState = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ss == SS_Secure && SDER().SUNIDEN == '1') then
        return TRUE;
    end;

    case ss of
        when SS_NonSecure => return TRUE;
        when SS_Secure =>    return ExternalSecureNoninvasiveDebugEnabled();
        when SS_Realm =>     return ExternalRealmNoninvasiveDebugEnabled();
        when SS_Root =>      return ExternalRootNoninvasiveDebugEnabled();
    end;
end;
```

J1.4.1.19 ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// ================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

readonly func ExternalNoninvasiveDebugEnabled() => boolean
begin
    return (IsFeatureImplemented(FEAT_Debugv8p4) || ExternalInvasiveDebugEnabled() ||
            NIDEN == HIGH);
end;
```

J1.4.1.20 ExternalRealmInvasiveDebugEnabled

```
// ExternalRealmInvasiveDebugEnabled()
// ===================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

readonly func ExternalRealmInvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == HIGH;
end;
```

J1.4.1.21 ExternalRealmNoninvasiveDebugEnabled

```
// ExternalRealmNoninvasiveDebugEnabled()
// ======================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

readonly func ExternalRealmNoninvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return ExternalRealmInvasiveDebugEnabled();
end;
```

J1.4.1.22 ExternalRootInvasiveDebugEnabled

```
// ExternalRootInvasiveDebugEnabled()
// ==================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND RTPIDEN AND SPIDEN) signal when FEAT_SEL2 is implemented
// and the (DBGEN AND RLPIDEN AND RTPIDEN) signal when FEAT_SEL2 is not implemented.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func ExternalRootInvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return (ExternalInvasiveDebugEnabled() &&
            (!IsFeatureImplemented(FEAT_SEL2) || ExternalSecureInvasiveDebugEnabled()) &&
            ExternalRealmInvasiveDebugEnabled() &&
            RTPIDEN == HIGH);
end;
```

J1.4.1.23 ExternalRootNoninvasiveDebugEnabled

```
// ExternalRootNoninvasiveDebugEnabled()
// =====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

func ExternalRootNoninvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return ExternalRootInvasiveDebugEnabled();
end;
```

J1.4.1.24 ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

readonly func ExternalSecureInvasiveDebugEnabled() => boolean
begin
    if !HaveSecureState() then
        return IsFeatureImplemented(FEAT_RME) && RTPIDEN == HIGH;
    end;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
end;
```

J1.4.1.25 ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

readonly func ExternalSecureNoninvasiveDebugEnabled() => boolean
begin
    if !HaveSecureState() then return FALSE; end;
    if !IsFeatureImplemented(FEAT_Debugv8p4) then
        return (ExternalNoninvasiveDebugEnabled() &&
                (SPIDEN == HIGH || SPNIDEN == HIGH));
    else
        return ExternalSecureInvasiveDebugEnabled();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.26 InvasiveDebugPermittedPAS

```
// InvasiveDebugPermittedPAS()
// ===========================
// Returns TRUE if the invasive debug of the configured PASpace is permitted by
// the authentication interface, and FALSE otherwise.

func InvasiveDebugPermittedPAS(pas : PASpace) => boolean
begin
    case pas of
        when PAS_Secure =>    return ExternalSecureInvasiveDebugEnabled();
        when PAS_NonSecure => return ExternalInvasiveDebugEnabled();
        when PAS_Root =>      return ExternalRootInvasiveDebugEnabled();
        when PAS_Realm =>     return ExternalRealmInvasiveDebugEnabled();
        otherwise =>          return FALSE;
    end;
end;
```

J1.4.1.27 IsAccessNonSecure

```
// IsAccessNonSecure()
// ===================
// Returns TRUE when an access is Non-Secure

func IsAccessNonSecure(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.paddress.paspace == PAS_NonSecure;
end;
```

J1.4.1.28 IsAccessRealm

```
// IsAccessRealm()
// ===============
// Returns TRUE when an access is Realm

func IsAccessRealm(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.paddress.paspace == PAS_Realm;
end;
```

J1.4.1.29 IsAccessRoot

```
// IsAccessRoot()
// ==============
// Returns TRUE when an access is Root

func IsAccessRoot(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.paddress.paspace == PAS_Root;
end;
```

J1.4.1.30 IsAccessSecure

```
// IsAccessSecure()
// ================
// Returns TRUE when an access is Secure

func IsAccessSecure(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.paddress.paspace == PAS_Secure;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.31 IsCorePowered

```
// IsCorePowered()
// ===============
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

impdef func IsCorePowered() => boolean
begin
    return TRUE;
end;
```

J1.4.1.32 IsMostSecureAccess

```
// IsMostSecureAccess()
// ====================
// Returns TRUE if the security state of an access is the most secure PAS.

func IsMostSecureAccess(addrdesc : AddressDescriptor) => boolean
begin
    if IsFeatureImplemented(FEAT_RME) then
        return addrdesc.paddress.paspace == PAS_Root;
    elsif HaveEL(EL3) || SecureOnlyImplementation() then
        return addrdesc.paddress.paspace == PAS_Secure;
    else
        assert addrdesc.paddress.paspace == PAS_NonSecure;
        return TRUE;
    end;
end;
```

J1.4.1.33 IsPASValid

```
// IsPASValid()
// ============
// Returns TRUE if the given value of 'pas' is not reserved, and FALSE otherwise.

func IsPASValid(pas : bits(2)) => boolean
begin
    case pas of
        when '00' => return IsFeatureImplemented(FEAT_Secure);
        when '01' => return TRUE;
        when '10' => return IsFeatureImplemented(FEAT_RME);
        when '11' => return IsFeatureImplemented(FEAT_RME);
    end;
end;
```

J1.4.1.34 BreakpointInfo

```
// BreakpointInfo
// ==============
// Breakpoint related fields.

type BreakpointInfo of record {
    bptype : BreakpointType, // Type of breakpoint matched
    match : boolean,         // breakpoint match
    mismatch : boolean       // breakpoint mismatch
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.35 BreakpointType

```
// BreakpointType
// ==============

type BreakpointType of enumeration {
    BreakpointType_Inactive,     // Breakpoint inactive or disabled
    BreakpointType_AddrMatch,    // Address Match breakpoint
    BreakpointType_AddrMismatch, // Address Mismatch breakpoint
    BreakpointType_CtxtMatch  }; // Context matching breakpoint
```

J1.4.1.36 CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

func CheckValidStateMatch(ssc_in : bits(2), ssce_in : bit,
                          hmc_in : bit, pxc_in : bits(2),
                          isbreakpnt : boolean) => (Constraint, bits(2), bit, bit, bits(2))
begin
    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0'; end;
    var reserved : boolean = FALSE;
    var ssc : bits(2) = ssc_in;
    var ssce : bit    = ssce_in;
    var hmc : bit     = hmc_in;
    var pxc : bits(2) = pxc_in;

    // Values that are not allocated in any architecture version
    case hmc::ssce::ssc::pxc of
        when '0 0 11 10' => reserved = TRUE;
        when '0 0 1x xx' => reserved = !HaveSecureState();
        when '1 0 00 x0' => reserved = TRUE;
        when '1 0 01 10' => reserved = TRUE;
        when '1 0 1x 10' => reserved = TRUE;
        when 'x 1 xx xx' => reserved = ssc != '01' || (hmc::pxc) IN {'000','110'};
        otherwise =>        reserved = FALSE;
    end;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && hmc::pxc == '000' && ssc != '11' then
        reserved = TRUE;
    end;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;
    end;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc::ssc::pxc != '10100' then
        reserved = TRUE;
    end;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc::ssc::pxc == '11000' then
        reserved = TRUE;
    end;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc::ssc::pxc == '11100' then
        reserved = TRUE;
    end;

    // Secure EL2 is not implemented
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !IsFeatureImplemented(FEAT_SEL2) && (hmc::ssc::pxc)  IN {'01100','10100','x11x1'} then
        reserved = TRUE;
    end;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        var c : Constraint;
        var unpred_state_bits : bits(6);
        (c, unpred_state_bits) = ConstrainUnpredictableBits{6}(Unpredictable_RESBPWPCTRL);
        hmc = unpred_state_bits[5];
        ssc = unpred_state_bits[4:3];
        ssce = unpred_state_bits[2];
        pxc = unpred_state_bits[1:0];
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, ARBITRARY : bits(2), ARBITRARY : bit, ARBITRARY : bit, ARBITRARY : bits(2));
        end;
    end;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, ssce, hmc, pxc);
end;
```

J1.4.1.37 ContextAwareBreakpointRange

```
// ContextAwareBreakpointRange()
// ============================
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

func ContextAwareBreakpointRange() => (integer, integer)
begin
    let b : integer = NumBreakpointsImplemented();
    let c : integer = NumContextAwareBreakpointsImplemented();

    if b <= 16 then
        return (b - c, b - 1);
    elsif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);
    end;
end;
```

J1.4.1.38 IsContextAwareBreakpoint

```
// IsContextAwareBreakpoint()
// ==========================
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

func IsContextAwareBreakpoint(n : integer) => boolean
begin
    let (lower, upper) : (integer, integer) = ContextAwareBreakpointRange();
    return n >= lower && n <= upper;
end;
```

J1.4.1.39 NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// ===========================
// Returns the number of breakpoints implemented.

readonly func NumBreakpointsImplemented() => integer
begin
    return ImpDefInt("Number of breakpoints");
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.40 NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =======================================
// Returns the number of context-aware breakpoints implemented.

readonly func NumContextAwareBreakpointsImplemented() => integer
begin
    return ImpDefInt("Number of context-aware breakpoints");
end;
```

J1.4.1.41 NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// ===========================
// Returns the number of watchpoints implemented.

readonly func NumWatchpointsImplemented() => integer
begin
    return ImpDefInt("Number of watchpoints");
end;
```

J1.4.1.42 CTI_ProcessEvent

```
// CTI_ProcessEvent()
// ==================
// Process a discrete event on a Cross Trigger output event trigger.

impdef func CTI_ProcessEvent(id : CrossTriggerOut)
begin
    return;
end;
```

J1.4.1.43 CTI_SetEventLevel

```
// CTI_SetEventLevel()
// ===================
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

impdef func CTI_SetEventLevel(id : CrossTriggerIn, level : Signal)
begin
    return;
end;
```

J1.4.1.44 CTI_SignalEvent

```
// CTI_SignalEvent()
// =================
// Signal a discrete event on a Cross Trigger input event trigger.

impdef func CTI_SignalEvent(id : CrossTriggerIn)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.45 CrossTrigger

```
// CrossTrigger
// ============

type CrossTriggerOut of enumeration {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                                     CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                                     CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                                     CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

type CrossTriggerIn of enumeration {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                                    CrossTriggerIn_SPESample,     CrossTriggerIn_RSVD3,
                                    CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                                    CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3,
                                    CrossTriggerIn_TRBEStop,      CrossTriggerIn_TRBEMgmt,
                                    CrossTriggerIn_TRBEWrap};
```

J1.4.1.46 CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

func CheckForDCCInterrupts()
begin
    let commrx : boolean = (EDSCR().RXfull == '1');
    let commtx : boolean = (EDSCR().TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    var commirq : boolean;
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT().RX == '1') ||
                   (commtx && DBGDCCINT().TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1().RX == '1') ||
                   (commtx && MDCCINT_EL1().TX == '1'));
    end;
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
end;
```

J1.4.1.47 DTR

```
// DTR
// ===

var DTRRX : bits(32);
var DTRTX : bits(32);
```

J1.4.1.48 Read_DBGDTRRX_EL0

```
// Read_DBGDTRRX_EL0()
// ===================
// Called on reads of debug register 0x080.

func Read_DBGDTRRX_EL0(memory_mapped : boolean) => bits(32)
begin
    return DTRRX;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.49 Read_DBGDTRTX_EL0

```
// Read_DBGDTRTX_EL0()
// ===================
// Called on reads of debug register 0x08C.

func Read_DBGDTRTX_EL0(memory_mapped : boolean) => bits(32)
begin
    let underrun : boolean = (EDSCR().TXfull == '0' ||
                              (Halted() && EDSCR().MA == '1' && EDSCR().ITE == '0'));
    let value : bits(32)   = if underrun then ARBITRARY : bits(32) else DTRTX;

    if EDSCR().ERR == '1' then return value; end;       // Error flag set: no side-effects

    if underrun then
        EDSCR().TXU = '1';  EDSCR().ERR = '1';          // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN
    end;

    EDSCR().TXfull = '0';
    if Halted() && EDSCR().MA == '1' then
        EDSCR().ITE = '0';                              // See comments in Write_EDITR()

        if !UsingAArch32() then
            ExecuteA64(0xB8404401[31:0]);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850[15:0] /*hw1*/, 0x1B04[15:0] /*hw2*/);      // T32 "LDR R1,[R0],#4"
        end;
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR().ERR == '1' then
            EDSCR().TXfull = ARBITRARY : bit;
            DBGDTRTX_EL0() = ARBITRARY : bits(64);
        else

            if !UsingAArch32() then
                ExecuteA64(0xD5130501[31:0]);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00[15:0] /*hw1*/, 0x1E15[15:0] /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            end;
            // "MSR DBGDTRTX_EL0,X1" calls Write_DBGDTR_EL0() which sets TXfull.
            assert EDSCR().TXfull == '1';
        end;
        if !UsingAArch32() then
            X{64}(1) = ARBITRARY : bits(64);
        else
            R(1) = ARBITRARY : bits(32);
        end;
        EDSCR().ITE = '1';                                // See comments in Write_EDITR()
    end;

    return value;
end;
```

J1.4.1.50 Read_DBGDTR_EL0

```
// Read_DBGDTR_EL0()
// =================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

func Read_DBGDTR_EL0{N}() => bits(N)
begin
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    var result : bits(N);
    if EDSCR().RXfull == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result = ARBITRARY : bits(N);
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result[63:32] = DTRTX; end;
        result[31:0] = DTRRX;
    end;
    EDSCR().RXfull = '0';
    return result;
end;
```

J1.4.1.51 Write_DBGDTRRX_EL0

```
// Write_DBGDTRRX_EL0()
// ====================
// Called on writes to debug register 0x080.

func Write_DBGDTRRX_EL0(memory_mapped : boolean, value : bits(32))
begin
    if EDSCR().ERR == '1' then return; end;                // Error flag set: ignore write

    if EDSCR().RXfull == '1' || (Halted() && EDSCR().MA == '1' && EDSCR().ITE == '0') then
        EDSCR().RXO = '1';  EDSCR().ERR = '1';              // Overrun condition: ignore write
        return;
    end;

    EDSCR().RXfull = '1';
    DTRRX          = value;

    if Halted() && EDSCR().MA == '1' then
        EDSCR().ITE = '0';                                 // See comments in Write_EDITR()
        if !UsingAArch32() then
            ExecuteA64(0xD5330501[31:0]);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401[31:0]);               // A64 "STR W1,[X0],#4"
            X{64}(1) = ARBITRARY : bits(64);
        else
            ExecuteT32(0xEE10[15:0] /*hw1*/, 0x1E15[15:0] /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840[15:0] /*hw1*/, 0x1B04[15:0] /*hw2*/);  // T32 "STR R1,[R0],#4"
            R(1) = ARBITRARY : bits(32);
        end;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR().ERR == '1' then
            EDSCR().RXfull = ARBITRARY : bit;
            DBGDTRRX_EL0() = ARBITRARY : bits(64);
        else
            // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears RXfull.
            assert EDSCR().RXfull == '0';
        end;

        EDSCR().ITE = '1';                                 // See comments in Write_EDITR()
    end;
    return;
end;
```

J1.4.1.52 Write_DBGDTRTX_EL0

```
// Write_DBGDTRTX_EL0()
// ====================
// Called on writes to debug register 0x08C.

func Write_DBGDTRTX_EL0(memory_mapped : boolean, value : bits(32))
begin
    DTRTX = value;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.53 Write_DBGDTR_EL0

```
// Write_DBGDTR_EL0()
// ==================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

func Write_DBGDTR_EL0{N}(value_in : bits(N))
begin
    var value : bits(N) = value_in;
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR().TXfull == '1' then
        value = ARBITRARY : bits(N);
    end;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value[63:32]; end;
    DTRTX = value[31:0];        // 32-bit or 64-bit write
    EDSCR().TXfull = '1';
    return;
end;
```

J1.4.1.54 Write_EDITR

```
// Write_EDITR()
// =============
// Called on writes to debug register 0x084.

func Write_EDITR(memory_mapped : boolean, value : bits(32))
begin
    if EDSCR().ERR == '1' then return; end;         // Error flag set: ignore write

    if !Halted() then return; end;                  // Non-debug state: ignore write

    if EDSCR().ITE == '0' || EDSCR().MA == '1' then
        EDSCR().ITO = '1';  EDSCR().ERR = '1';      // Overrun condition: block write
        return;
    end;

    // ITE indicates whether the PE is ready to accept another instruction; the PE
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR().ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value[15:0]/*hw1*/, value[31:16] /*hw2*/);
    end;

    EDSCR().ITE = '1';

    return;
end;
```

J1.4.1.55 DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

func DCPSInstruction(target_el : bits(2))
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SynchronizeContext();

var handle_el : bits(2);
case target_el of
    when EL1 =>
        if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
            handle_el = PSTATE.EL;
        elsif EL2Enabled() && HCR_EL2().TGE == '1' then
            Undefined();
        else
            handle_el = EL1;
        end;
    when EL2 =>
        if !HaveEL(EL2) then
            Undefined();
        elsif PSTATE.EL == EL3 && !UsingAArch32() then
            handle_el = EL3;
        elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            Undefined();
        else
            handle_el = EL2;
        end;
    when EL3 =>
        if EDSCR().SDD == '1' || !HaveEL(EL3) then
            Undefined();
        else
            handle_el = EL3;
        end;
    otherwise =>
        unreachable;
end;

let from_secure : boolean = CurrentSecurityState() == SS_Secure;
if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1 =>
            AArch32_WriteMode(M32_Svc);
            if IsFeatureImplemented(FEAT_PAN) && SCTLR().SPAN == '0' then
                PSTATE.PAN = '1';
            end;
        when EL2 =>  AArch32_WriteMode(M32_Hyp);
        when EL3 =>
            AArch32_WriteMode(M32_Monitor);
            if IsFeatureImplemented(FEAT_PAN) then
                if !from_secure then
                    PSTATE.PAN = '0';
                elsif SCTLR().SPAN == '0' then
                    PSTATE.PAN = '1';
                end;
            end;
    end;
    if handle_el == EL2 then
        ELR_hyp() = ARBITRARY : bits(32);  HSR() = ARBITRARY : bits(32);
    else
        LR() = ARBITRARY : bits(32);
    end;
    SPSR_curr() = ARBITRARY : bits(32);
    if handle_el == EL2 then
        PSTATE.E = HSCTLR().EE;
    else
        PSTATE.E = SCTLR().EE;
    end;
    DLR() = ARBITRARY : bits(32);  DSPSR() = ARBITRARY : bits(32);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else                                        // Targeting AArch64
        let from_32 : boolean = UsingAArch32();
        if from_32 then AArch64_MaybeZeroRegisterUppers(); end;
        if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
            ResetSVEState();
        else
            MaybeZeroSVEUppers(target_el);
        end;
        PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
        if IsFeatureImplemented(FEAT_PAN) && ((handle_el == EL1 && SCTLR_EL1().SPAN == '0') ||
                                              (handle_el == EL2 && ELIsInHost(EL0) &&
                                               SCTLR_EL2().SPAN == '0')) then
            PSTATE.PAN = '1';
        end;
        ELR_ELx() = ARBITRARY : bits(64);
        SPSR_ELx() = ARBITRARY : bits(64);
        ESR_ELx() = ARBITRARY : bits(64);
        DLR_EL0() = ARBITRARY : bits(64);  DSPSR_EL0() = ARBITRARY : bits(64);
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
        if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = '0'; end;
    end;
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    UpdateEDSCRFields();                        // Update EDSCR PE state flags
    return;
end;
```

J1.4.1.56 DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

func DRPSInstruction()
begin
    SynchronizeContext();

    DebugRestorePSR();

    return;
end;
```

J1.4.1.57 DebugHalt

```
// DebugHalt
// =========
// Reason codes for entry to Debug state

constant DebugHalt_Breakpoint : bits(6)      = '000111';
constant DebugHalt_EDBGRQ : bits(6)          = '010011';
constant DebugHalt_Step_Normal : bits(6)     = '011011';
constant DebugHalt_Step_Exclusive : bits(6)  = '011111';
constant DebugHalt_OSUnlockCatch : bits(6)   = '100011';
constant DebugHalt_ResetCatch : bits(6)      = '100111';
constant DebugHalt_Watchpoint : bits(6)      = '101011';
constant DebugHalt_HaltInstruction : bits(6) = '101111';
constant DebugHalt_SoftwareAccess : bits(6)  = '110011';
constant DebugHalt_ExceptionCatch : bits(6)  = '110111';
constant DebugHalt_Step_NoSyndrome : bits(6) = '111011';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.58 DebugRestorePSR

```
// DebugRestorePSR()
// =================

func DebugRestorePSR()
begin
    // PSTATE.[N,Z,C,V,Q,GE,SS,D,A,I,F] are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        let spsr : bits(32) = SPSR_curr();
        SetPSTATEFromPSR{32}(spsr);
        PSTATE.[N,Z,C,V,Q,GE,SS,A,I,F] = ARBITRARY : bits(13);
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
        DLR() = ARBITRARY : bits(32);  DSPSR() = ARBITRARY : bits(32);
    else
        let spsr : bits(64) = SPSR_ELx();
        SetPSTATEFromPSR{64}(spsr);
        PSTATE.[N,Z,C,V,SS,D,A,I,F] = ARBITRARY : bits(9);
        DLR_EL0() = ARBITRARY : bits(64);  DSPSR_EL0() = ARBITRARY : bits(64);
    end;
    UpdateEDSCRFields();                                // Update EDSCR PE state flags
end;
```

J1.4.1.59 DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// ========================================

impdef func DisableITRAndResumeInstructionPrefetch()
begin
    return;
end;
```

J1.4.1.60 ExecuteA64

```
// ExecuteA64()
// ============
// Execute an A64 instruction in Debug state.

impdef func ExecuteA64(instr : bits(32))
begin
    Unimplemented();
end;
```

J1.4.1.61 ExecuteT32

```
// ExecuteT32()
// ============
// Execute a T32 instruction in Debug state.

impdef func ExecuteT32(hw1 : bits(16), hw2 : bits(16))
begin
    Unimplemented();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.62 ExitDebugState

```
// ExitDebugState()
// ================

func ExitDebugState()
begin
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR().STATUS = '000001';                          // Signal restarting
    // Clear any pending Halting debug events
    if IsFeatureImplemented(FEAT_Debugv8p8) then
        EDESR()[3:0] = '0000';
    else
        EDESR()[2:0] = '000';
    end;

    var new_pc : bits(64);
    var spsr : bits(64);

    if UsingAArch32() then
        new_pc = ZeroExtend{64}(DLR());
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            spsr = DSPSR2() :: DSPSR();
        else
            spsr = ZeroExtend{64}(DSPSR());
        end;
    else
        new_pc = DLR_EL0();
        spsr = DSPSR_EL0();
    end;

    let illegal_psr_state : boolean = IllegalExceptionReturn{64}(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR{64}(spsr); // Can update privileged bits, even at EL0

    let branch_conditional : boolean = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc[0] = '0'; end;
        // AArch32 branch
        BranchTo{32}(new_pc[31:0], BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr[4] == '1' then
            new_pc[63:32] = ARBITRARY : bits(32);
            new_pc[1:0] = ARBITRARY : bits(2);
        end;
        if IsFeatureImplemented(FEAT_BRBE) then
            BRBEDebugStateExit(AArch64_BranchAddr(new_pc,PSTATE.EL));
        end;
        // A type of branch that is never predicted
        BranchTo{64}(new_pc, BranchType_DBGEXIT, branch_conditional);
    end;

     // Atomically signal restarted
    EDSCR().STATUS = '000010';
    EDPRSR().SDR = '1';
    // End of atomically signal
    EDPRSR().HALTED = '0';
    UpdateEDSCRFields();                                // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.63 Halt

```
// Halt()
// ======

func Halt(reason : bits(6))
begin
    let is_async : boolean = FALSE;
    let fault : FaultRecord = NoFault();
    Halt(reason, is_async, fault);
end;

// Halt()
// ======

func Halt(reason : bits(6), is_async : boolean, fault : FaultRecord)
begin
    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    let preferred_restart_address : bits(64) = ThisInstrAddr{}();
    var spsr : bits(64) = GetPSRFromPSTATE{64}(DebugState);

    if (IsFeatureImplemented(FEAT_BTI) && !is_async &&
          ! reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
                       DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint,
                       DebugHalt_HaltInstruction} &&
          ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
        spsr[11:10] = '00';
    end;

    if UsingAArch32() then
        DLR() = preferred_restart_address[31:0];
        DSPSR() = spsr[31:0];
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DSPSR2() = spsr[63:32];
        end;
    else
        DLR_EL0() = preferred_restart_address;
        DSPSR_EL0() = spsr;
    end;
    EDSCR().ITE = '1';
    EDSCR().ITO = '0';
    if IsFeatureImplemented(FEAT_RME) then
        if PSTATE.EL == EL3 then
            EDSCR().SDD = '0';
        else
            EDSCR().SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
        end;
    elsif CurrentSecurityState() == SS_Secure then
        EDSCR().SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR().SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        EDSCR().SDD = '1';                        // Otherwise EDSCR.SDD is RES1
    end;
    EDSCR().MA = '0';

    // In Debug state:
    // * PSTATE.[SS,SSBS,D,A,I,F] are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.[N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT] are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.[IT,T] are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.BTYPE is ignored and behave-as-if 0.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // * PSTATE.TCO is set 1.
    // * PSTATE.PACM is ignored and behave-as-if 0.
    // * PSTATE.[UAO,PAN] are observable and not changed on entry into Debug state.
    // * PSTATE.UINJ is set to 0.

    if UsingAArch32() then
        PSTATE.[IT,SS,SSBS,A,I,F,T] = ARBITRARY : bits(14);
    else
        PSTATE.[SS,SSBS,D,A,I,F]    = ARBITRARY : bits(6);
    end;

    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00'; end;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEDebugStateEntry(preferred_restart_address);
    end;
    StopInstructionPrefetchAndEnableITR();
    // atomic write
    EDSCR().STATUS = reason;
    EDPRSR().HALTED = '1';
    // end of atomic write
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.
    if IsFeatureImplemented(FEAT_EDHSR) then
        UpdateEDHSR(reason, fault);             // Update EDHSR fields.
    end;
    if !is_async then EndOfInstruction(EndOfInstructionReason_Exception); end;
    return;
end;
```

J1.4.1.64 HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

func HaltOnBreakpointOrWatchpoint() => boolean
begin
    return HaltingAllowed() && EDSCR().HDE == '1' && OSLSR_EL1().OSLK == '0';
end;
```

J1.4.1.65 Halted

```
// Halted()
// ========

readonly func Halted() => boolean
begin
    return ! EDSCR().STATUS IN {'000001', '000010'};                    // Halted
end;
```

J1.4.1.66 HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

func HaltingAllowed() => boolean
begin
    if Halted() || DoubleLockStatus() then
        return FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    let ss : SecurityState = CurrentSecurityState();
    case ss of
        when SS_NonSecure => return ExternalInvasiveDebugEnabled();
        when SS_Secure =>    return ExternalSecureInvasiveDebugEnabled();
        when SS_Root =>      return ExternalRootInvasiveDebugEnabled();
        when SS_Realm =>     return ExternalRealmInvasiveDebugEnabled();
    end;
end;
```

J1.4.1.67 Restarting

```
// Restarting()
// ============

readonly func Restarting() => boolean
begin
    return EDSCR().STATUS == '000001';                                  // Restarting
end;
```

J1.4.1.68 StopInstructionPrefetchAndEnableITR

```
// StopInstructionPrefetchAndEnableITR()
// =====================================

impdef func StopInstructionPrefetchAndEnableITR()
begin
    return;
end;
```

J1.4.1.69 UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====================
// Provides information about the state of the
// IMPLEMENTATION DEFINED authentication interface for debug.

func UpdateDbgAuthStatus()
begin
    var nsid, nsnid : bits(2);
    var sid, snid : bits(2);
    var rlid, rtid : bits(2);
    if SecureOnlyImplementation() then
        nsid = '00';
    elsif ExternalInvasiveDebugEnabled() then
        nsid = '11';          // Non-secure Invasive debug implemented and enabled.
    else
        nsid = '10';          // Non-secure Invasive debug implemented and disabled.
    end;

    if SecureOnlyImplementation() then
        nsnid = '00';
    elsif ExternalNoninvasiveDebugEnabled() then
        nsnid = '11';         // Non-secure Non-Invasive debug implemented and enabled.
    else
        nsnid = '10';         // Non-secure Non-Invasive debug implemented and disabled.
    end;

    if !HaveSecureState() then
        sid = '00';
    elsif ExternalSecureInvasiveDebugEnabled() then
        sid = '11';           // Secure Invasive debug implemented and enabled.
    else
        sid = '10';           // Secure Invasive debug implemented and disabled.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    if !HaveSecureState() then
        snid = '00';
    elsif ExternalSecureNoninvasiveDebugEnabled() then
        snid = '11';          // Secure Non-Invasive debug implemented and enabled.
    else
        snid = '10';          // Secure Non-Invasive debug implemented and disabled.
    end;

    if !IsFeatureImplemented(FEAT_RME) then
        rlid = '00';
    elsif ExternalRealmInvasiveDebugEnabled() then
        rlid = '11';          // Realm Invasive debug implemented and enabled.
    else
        rlid = '10';          // Realm Invasive debug implemented and disabled.
    end;

    if !IsFeatureImplemented(FEAT_RME) then
        rtid = '00';
    elsif ExternalRootInvasiveDebugEnabled() then
        rtid = '11';          // Root Invasive debug implemented and enabled.
    else
        rtid = '10';          // Root Invasive debug implemented and disabled.
    end;

    DBGAUTHSTATUS_EL1().NSID  = nsid;
    DBGAUTHSTATUS_EL1().NSNID = nsnid;
    DBGAUTHSTATUS_EL1().SID   = sid;
    DBGAUTHSTATUS_EL1().SNID  = snid;
    DBGAUTHSTATUS_EL1().RLID  = rlid;
    DBGAUTHSTATUS_EL1().RLNID = rlid;    // Field has the same value as DBGAUTHSTATUS_EL1.RLID.
    DBGAUTHSTATUS_EL1().RTID  = rtid;
    DBGAUTHSTATUS_EL1().RTNID = rtid;    // Field has the same value as DBGAUTHSTATUS_EL1.RTID.
    return;
end;
```

J1.4.1.70 UpdateEDHSR

```
// UpdateEDHSR()
// =============
// Update EDHSR watchpoint related fields.

func UpdateEDHSR(reason : bits(6), fault : FaultRecord)
begin
    var syndrome : bits(64) = Zeros{};
    if reason == DebugHalt_Watchpoint then
        if IsFeatureImplemented(FEAT_GCS) && fault.accessdesc.acctype == AccessType_GCS then
            syndrome[40] = '1';                        // GCS
        end;
        syndrome[23:0] = WatchpointRelatedSyndrome(fault);
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            if fault.write then syndrome[6] = '1'; end; // WnR
            if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
                syndrome[8] = '1';                     // CM
            end;
            if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
                syndrome[13] = '1';                    // VNCR
            end;
        end;
    else
        syndrome = ARBITRARY : bits(64);
    end;

    EDHSR() = syndrome;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.71 UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

func UpdateEDSCRFields()
begin
    if !Halted() then
        EDSCR().EL = '00';
        if IsFeatureImplemented(FEAT_RME) then
           // SDD bit.
            EDSCR().SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
            EDSCR().[NSE,NS] = ARBITRARY : bits(2);
        else
            // SDD bit.
            EDSCR().SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
            EDSCR().NS = ARBITRARY : bit;
        end;

        EDSCR().RW = '1111';
    else
        EDSCR().EL = PSTATE.EL;
        // SError Pending.
        if EL2Enabled() && HCR_EL2().[AMO,TGE] == '10' && PSTATE.EL IN {EL0,EL1} then
            EDSCR().A = if IsVirtualSErrorPending() then '1' else '0';
        elsif (EffectiveSCR_EL3_EnDSE() == '1' && PSTATE.EL IN {EL0,EL1,EL2}) then
            EDSCR().A = if IsDelegatedSErrorPending() then '1' else '0';
        else
            EDSCR().A = if IsPhysicalSErrorPending() then '1' else '0';
        end;

        let ss : SecurityState = CurrentSecurityState();
        if IsFeatureImplemented(FEAT_RME) then
            case ss of
                when SS_Secure =>    EDSCR().[NSE,NS] = '00';
                when SS_NonSecure => EDSCR().[NSE,NS] = '01';
                when SS_Root =>      EDSCR().[NSE,NS] = '10';
                when SS_Realm =>     EDSCR().[NSE,NS] = '11';
            end;
        else
            EDSCR().NS = if ss == SS_Secure then '0' else '1';
        end;

        var RW : bits(4);
        RW[1] = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW[0] = RW[1];
        else
            RW[0] = if UsingAArch32() then '0' else '1';
        end;
        if !EL2Enabled() then
            RW[2] = RW[1];
        else
            RW[2] = if ELUsingAArch32(EL2) then '0' else '1';
        end;
        if !HaveEL(EL3) then
            RW[3] = RW[2];
        else
            RW[3] = if ELUsingAArch32(EL3) then '0' else '1';
        end;

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW[3] == '0' then RW[2:0] = ARBITRARY : bits(3);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif RW[2] == '0' then RW[1:0] = ARBITRARY : bits(2);
        elsif RW[1] == '0' then RW[0] = ARBITRARY : bit; end;
        EDSCR().RW = RW;
    end;
    return;
end;
```

J1.4.1.72 haltingevents

```
var EDBGRQ : Signal;  // Input Signal in external debug interface
```

J1.4.1.73 CheckEDBGRQ

```
// CheckEDBGRQ()
// =============
// Checks IMPLEMENTATION DEFINED EDBGRQ Input Signal of external debug interface.
// This is an example of an IMPLEMENTATION DEFINED source of External Debug Request debug events.

func CheckEDBGRQ()
begin
    if EDBGRQ == HIGH then
        ExternalDebugRequest();
    end;
end;
```

J1.4.1.74 CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

func CheckExceptionCatch(exception_entry : boolean)
begin
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    let ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
    var base : integer;

    case ss of
        when SS_Secure    => base = 0;
        when SS_NonSecure => base = 4;
        when SS_Realm     => base = 16;
        when SS_Root      => base = 0;
    end;
    if HaltingAllowed() then
        var halt : boolean;
        if IsFeatureImplemented(FEAT_Debugv8p2) then
            let exception_exit : boolean = !exception_entry;
            let increment : integer{} = if ss == SS_Realm then 4 else 8;
            let ctrl : bits(2) = (EDECCR()[UInt(PSTATE.EL) + base + increment]::
                                  EDECCR()[UInt(PSTATE.EL) + base]);
            case ctrl of
                when '00' =>  halt = FALSE;
                when '01' =>  halt = TRUE;
                when '10' =>  halt = (exception_exit == TRUE);
                when '11' =>  halt = (exception_entry == TRUE);
            end;
        else
            halt = (EDECCR()[UInt(PSTATE.EL) + base] == '1');
        end;

        if halt then
            if IsFeatureImplemented(FEAT_Debugv8p8) && exception_entry then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                EDESR().EC = '1';
            else
                Halt(DebugHalt_ExceptionCatch);
            end;
        end;
    end;
end;
```

J1.4.1.75 CheckExternalDebugRequestEvents

```
// CheckExternalDebugRequestEvents()
// =================================
// Checks for all External Debug Request debug events.

func CheckExternalDebugRequestEvents()
begin
    CheckTRBEHalt();
    CheckPMUHalt();
    CheckEDBGRQ();
    return;
end;
```

J1.4.1.76 CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

func CheckHaltingStep(is_async : boolean)
begin
    let step_enabled : boolean = EDECR().SS == '1' && HaltingAllowed();
    let active_pending : boolean = step_enabled && EDESR().SS == '1';
    if active_pending then
        if HaltingStep_DidNotStep() then
            let fault : FaultRecord = NoFault();
            Halt(DebugHalt_Step_NoSyndrome, is_async, fault);
        elsif HaltingStep_SteppedEX() then
            let fault : FaultRecord = NoFault();
            Halt(DebugHalt_Step_Exclusive, is_async, fault);
        else
            let fault : FaultRecord = NoFault();
            Halt(DebugHalt_Step_Normal, is_async, fault);
        end;
    end;
    if step_enabled then ShouldAdvanceHS = TRUE; end;
    return;
end;
```

J1.4.1.77 CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

func CheckOSUnlockCatch()
begin
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL().OSUCE == '1') ||
          (!IsFeatureImplemented(FEAT_DoPD) && EDECR().OSUCE == '1')) then
        if !Halted() then EDESR().OSUC = '1'; end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.78 CheckPMUHalt

```
// CheckPMUHalt()
// ==============

func CheckPMUHalt()
begin
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_PMUv3p9) then
        return;
    end;

    // The request remains set until the condition is cleared.
    // For example, an interrupt handler or cross-triggered event handler clears
    // the overflow status flag by writing to PMOVSCLR_EL0

    let include_r1 : boolean = TRUE;
    let include_r2 : boolean = TRUE;
    let include_r3 : boolean = TRUE;

    let pmuhalt : boolean = CheckPMUOverflowCondition(PMUOverflowCondition_EDBGRQ,
                                                      include_r1, include_r2, include_r3);
    if pmuhalt && EDECR().PME == '1' then
        ExternalDebugRequest();
    end;
end;
```

J1.4.1.79 CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// ============================
// Check whether EDESR.EC has been set by an Exception Catch debug event.

func CheckPendingExceptionCatch(is_async : boolean)
begin
    if IsFeatureImplemented(FEAT_Debugv8p8) && HaltingAllowed() && EDESR().EC == '1' then
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt_ExceptionCatch, is_async, fault);
    end;
end;
```

J1.4.1.80 CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

func CheckPendingOSUnlockCatch()
begin
    if HaltingAllowed() && EDESR().OSUC == '1' then
        let is_async : boolean = TRUE;
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt_OSUnlockCatch, is_async, fault);
    end;
end;
```

J1.4.1.81 CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

func CheckPendingResetCatch()
begin
    if HaltingAllowed() && EDESR().RC == '1' then
        let is_async : boolean = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt_ResetCatch, is_async, fault);
    end;
end;
```

J1.4.1.82 CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

func CheckResetCatch()
begin
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL().RCE == '1') ||
          (!IsFeatureImplemented(FEAT_DoPD) && EDECR().RCE == '1')) then
        EDESR().RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch); end;
    end;
end;
```

J1.4.1.83 CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

func CheckSoftwareAccessToDebugRegisters()
begin
    let os_lock : bit = (if ELUsingAArch32(EL1) then DBGOSLSR().OSLK else OSLSR_EL1().OSLK);
    if HaltingAllowed() && EDSCR().TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
    end;
end;
```

J1.4.1.84 CheckTRBEHalt

```
// CheckTRBEHalt()
// ===============

func CheckTRBEHalt()
begin
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_TRBE_EXT) then
        return;
    end;

    if TraceBufferEnabled() && TRBSR_EL1().IRQ == '1' && EDECR().TRBE == '1' then
        ExternalDebugRequest();
    end;
end;
```

J1.4.1.85 ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

func ExternalDebugRequest()
begin
    if HaltingAllowed() then
        let is_async : boolean = TRUE;
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt_EDBGRQ, is_async, fault);
    // Otherwise the CTI continues to assert the debug request until it is taken.
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.86 HSAdvance

```
// HSAdvance()
// ===========
// Advance the Halting Step State Machine

func HSAdvance()
begin
    if !ShouldAdvanceHS then return; end;
    let step_enabled : boolean = EDECR().SS == '1' && HaltingAllowed();
    let active_not_pending : boolean = step_enabled && EDESR().SS == '0';
    if active_not_pending then EDESR().SS = '1'; end;  // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
end;
```

J1.4.1.87 HaltingStep_DidNotStep

```
// HaltingStep_DidNotStep()
// ========================
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

impdef func HaltingStep_DidNotStep() => boolean
begin
    return FALSE;
end;
```

J1.4.1.88 HaltingStep_SteppedEX

```
// HaltingStep_SteppedEX()
// =======================
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

impdef func HaltingStep_SteppedEX() => boolean
begin
    return FALSE;
end;
```

J1.4.1.89 ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'.

func ExternalDebugInterruptsDisabled(target : bits(2)) => boolean
begin
    var int_dis : boolean;
    let ss : SecurityState = SecurityStateAtEL(target);
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if EDSCR().INTdis[0] == '1' then
            case ss of
                when SS_NonSecure => int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure =>    int_dis = ExternalSecureInvasiveDebugEnabled();
                when SS_Realm =>     int_dis = ExternalRealmInvasiveDebugEnabled();
                when SS_Root =>      int_dis = ExternalRootInvasiveDebugEnabled();
            end;
        else
            int_dis = FALSE;
        end;
    else
        case target of
            when EL3 =>
                int_dis = (EDSCR().INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when EL2 =>
                int_dis = (EDSCR().INTdis == '1x' && ExternalInvasiveDebugEnabled());
            when EL1 =>
                if ss == SS_Secure then
                    int_dis = (EDSCR().INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR().INTdis != '00' && ExternalInvasiveDebugEnabled());
                end;
        end;
    end;
    return int_dis;
end;
```

J1.4.1.90 pmu

```
var PMUEventAccumulator : array [[31]] of integer;   // Accumulates PMU events for a cycle
var PMULastThresholdValue : array [[31]] of boolean; // A record of the threshold result for each
                                                     // event counter on the previous cycle

// Constant used in PMU functions to represent actions on the cycle counter.
constant CYCLE_COUNTER_ID : integer = 31;
// Constant used in PMU functions to represent actions on the instruction counter.
constant INSTRUCTION_COUNTER_ID : integer = 32;
```

J1.4.1.91 CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====================
// Called before each instruction is executed.
// If a PMU event counter has overflowed, this function might do any of:
//  - Signal a Performance Monitors overflow interrupt request.
//  - Signal a CTI Performance Monitors overflow event.
//  - Generate an External Debug Request debug event.
//  - Generate a BRBE freeze event.

func CheckForPMUOverflow()
begin
    let include_r1 : boolean = TRUE;
    let include_r2 : boolean = TRUE;
    let include_r3 : boolean = TRUE;

    let enabled : boolean = PMUInterruptEnabled();
    let pmuirq : boolean = CheckPMUOverflowCondition(PMUOverflowCondition_IRQ,
                                                     include_r1, include_r2, include_r3);

    SetInterruptRequestLevel(InterruptID_PMUIRQ,
                             if enabled && pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow,
                      if pmuirq then HIGH else LOW);

    if ShouldBRBEFreeze() then
        BRBEFreeze();
    end;

    return;
end;
```

J1.4.1.92 CheckPMUOverflowCondition

```
// CheckPMUOverflowCondition()
// ===========================
// Checks for PMU overflow under certain parameter conditions described by 'reason'.
// If 'include_r1' is TRUE, then check counters in the range [0..(HPMN-1)], CCNTR
//     and ICNTR, unless excluded by 'reason'.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// If 'include_r2' is TRUE, then check counters in the range [HPMN..(EPMN-1)].
// If 'include_r3' is TRUE, then check counters in the range [EPMN..(N-1)].

func CheckPMUOverflowCondition(reason : PMUOverflowCondition,
                               include_r1 : boolean, include_r2 : boolean, include_r3 : boolean
                              ) => boolean
begin
    // 'reason' is decoded into a further set of parameters:
    // If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
    // If 'check_inten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
    // If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
    var check_e : boolean;
    var check_inten : boolean;
    var exclude_cyc : boolean;
    case reason of
        when PMUOverflowCondition_PMUException =>
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
        when PMUOverflowCondition_BRBEFreeze =>
            check_e      = FALSE;
            check_inten  = FALSE;
            exclude_cyc  = TRUE;
        when PMUOverflowCondition_Freeze =>
            check_e      = FALSE;
            check_inten  = FALSE;
            exclude_cyc  = TRUE;
        when PMUOverflowCondition_IRQ, PMUOverflowCondition_EDBGRQ =>
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
        otherwise =>
            unreachable;
    end;

    var ovsf : bits(64);

    if HaveAArch64() then
        ovsf = PMOVSSET_EL0();
        ovsf[63:33] = Zeros{31};
        if !IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
            ovsf[INSTRUCTION_COUNTER_ID] = '0';
        end;
    else
        ovsf = ZeroExtend{64}(PMOVSSET());
    end;

    let counters : integer{} = NUM_PMU_COUNTERS;
    // Remove unimplemented counters - these fields are RES0
    if counters < 31 then
        ovsf[30:counters] = Zeros{31-counters};
    end;

    for idx = 0 to counters - 1 do
        var global_en : bit;
        case GetPMUCounterRange(idx) of
            when PMUCounterRange_R1 =>
                global_en = if HaveAArch64() then PMCR_EL0().E else PMCR().E;
                if !include_r1 then
                    ovsf[idx] = '0';
                end;
            when PMUCounterRange_R2 =>
                global_en = if HaveAArch64() then MDCR_EL2().HPME else HDCR().HPME;
                if !include_r2 then
                    ovsf[idx] = '0';
                end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when PMUCounterRange_R3 =>
                global_en = PMCCR().EPME;
                if !include_r3 then
                    ovsf[idx] = '0';
                end;
            otherwise =>
                unreachable;
        end;
        if check_e then
            ovsf[idx] = ovsf[idx] AND global_en;
        end;
    end;

    // Cycle counter
    if exclude_cyc || !include_r1 then
        ovsf[CYCLE_COUNTER_ID] = '0';
    end;

    if check_e then
        ovsf[CYCLE_COUNTER_ID] = ovsf[CYCLE_COUNTER_ID] AND PMCR_EL0().E;
    end;

    // Instruction counter
    if HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        if !include_r1 then
            ovsf[INSTRUCTION_COUNTER_ID] = '0';
        end;
        if check_e then
            ovsf[INSTRUCTION_COUNTER_ID] = ovsf[INSTRUCTION_COUNTER_ID] AND PMCR_EL0().E;
        end;
    end;

    if check_inten then
        let inten : bits(64) = (if HaveAArch64() then PMINTENCLR_EL1()
                                else ZeroExtend{64}(PMINTENCLR()));
        ovsf = ovsf AND inten;
    end;

    return !IsZero(ovsf);
end;
```

J1.4.1.93 ClearEventCounters

```
// ClearEventCounters()
// ====================
// Zero all the event counters.
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.P or PMCR.P.

func ClearEventCounters()
begin
    // Although ZeroPMUCounters implements the functionality for PMUACR_EL1
    // that is part of FEAT_PMUv3p9, it should be noted that writes to
    // PMCR_EL0 are not allowed at EL0 when PMUSERENR_EL0.UEN is 1, meaning
    // it is not relevant in this case.
    ZeroPMUCounters(Zeros{33} :: Ones{31});
end;
```

J1.4.1.94 CountPMUEvents

```
// CountPMUEvents()
// ================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (31).
// For the instruction counter, idx == INSTRUCTION_COUNTER_ID (32).
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func CountPMUEvents(idx : integer) => boolean
begin
    let counters : integer = NUM_PMU_COUNTERS;
    assert (idx == CYCLE_COUNTER_ID || idx < counters ||
            (idx == INSTRUCTION_COUNTER_ID && IsFeatureImplemented(FEAT_PMUv3_ICNTR)));

    var debug : boolean;
    var enabled : boolean;
    var prohibited : boolean;
    var filtered : boolean;
    var frozen : boolean;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters
    let counter_range : PMUCounterRange = GetPMUCounterRange(idx);
    var ss : SecurityState = CurrentSecurityState();

    // Main enable controls
    var global_en : bit;
    var counter_en : bit;
    case counter_range of
        when PMUCounterRange_R1 =>
            global_en = if HaveAArch64() then PMCR_EL0().E else PMCR().E;
        when PMUCounterRange_R2 =>
            global_en = if HaveAArch64() then MDCR_EL2().HPME else HDCR().HPME;
        when PMUCounterRange_R3 =>
            assert IsFeatureImplemented(FEAT_PMUv3_EXTPMN);
            global_en = PMCCR().EPME;
        otherwise =>
            unreachable;
    end;

    case idx of
        when INSTRUCTION_COUNTER_ID =>
            assert HaveAArch64();
            counter_en = PMCNTENSET_EL0().F0;
        when CYCLE_COUNTER_ID =>
            counter_en = if HaveAArch64() then PMCNTENSET_EL0().C else PMCNTENSET().C;
        otherwise =>
            counter_en = if HaveAArch64() then PMCNTENSET_EL0()[idx] else PMCNTENSET()[idx];
    end;

    enabled = global_en == '1' && counter_en == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state or at EL3 is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMUv3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Either not executing at EL0 using AArch32, or one of the following is true:
    //     - EL3 is using AArch32 and SDER.SUNIDEN == 0
    //     - EL3 is using AArch64, EL1 is using AArch32, and SDER32_EL3.SUNIDEN == 0
    // * PMNx is not reserved for use by the external interface
    if (HaveEL(EL3) && (ss == SS_Secure || PSTATE.EL == EL3) &&
          counter_range != PMUCounterRange_R3) then
        if !ELUsingAArch32(EL3) then
            prohibited = (MDCR_EL3().SPME == '0' &&
                          (!IsFeatureImplemented(FEAT_PMUv3p7) || MDCR_EL3().MPMX == '0'));
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        prohibited = SDCR().SPME == '0';
    end;

    if prohibited && PSTATE.EL == EL0 then
        if ELUsingAArch32(EL3) then
            prohibited = SDER().SUNIDEN == '0';
        elsif ELUsingAArch32(EL1) then
            prohibited = SDER32_EL3().SUNIDEN == '0';
        end;
    end;
end;

// Event counting at EL3 is prohibited if all of:
// * FEAT_PMUv3p7 is implemented
// * EL3 is using AArch64
// * One of the following is true:
//   - MDCR_EL3.SPME == 0
//   - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
// * PMNx is not reserved for use by the external interface
if (!prohibited && IsFeatureImplemented(FEAT_PMUv3p7) && PSTATE.EL == EL3 &&
      HaveAArch64() && counter_range != PMUCounterRange_R3) then
    prohibited = (MDCR_EL3().MPMX == '1' &&
                  (MDCR_EL3().SPME == '0' || counter_range == PMUCounterRange_R1));
end;

// Event counting at EL2 is prohibited if all of:
// * FEAT_PMUv3p1 is implemented
// * PMNx is not reserved for EL2 or the external interface
// * EL2 is using AArch64 and MDCR_EL2.HPMD == 1, or EL2 is using AArch32 and HDCR.HPMD == 1
if (!prohibited && PSTATE.EL == EL2 && IsFeatureImplemented(FEAT_PMUv3p1) &&
      counter_range == PMUCounterRange_R1) then
    let hpmd : bit = if HaveAArch64() then MDCR_EL2().HPMD else HDCR().HPMD;
    prohibited = hpmd == '1';
end;

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !IsFeatureImplemented(FEAT_Debugv8p2) then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();
end;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if IsFeatureImplemented(FEAT_PMUv3p7) then
    var fz : bit;
    case counter_range of
        when PMUCounterRange_R1 =>
            fz = if HaveAArch64() then PMCR_EL0().FZO else PMCR().FZO;
        when PMUCounterRange_R2 =>
            fz = if HaveAArch64() then MDCR_EL2().HPMFZO else HDCR().HPMFZO;
        when PMUCounterRange_R3 =>
            fz = '0';
        otherwise =>
            unreachable;
    end;
    frozen = (fz == '1') && ShouldPMUFreeze(counter_range);
    frozen = frozen || SPEFreezeOnEvent(idx);
else
    frozen = FALSE;
end;

// PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
// or frozen
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    let dp : bit = if HaveAArch64() then PMCR_EL0().DP else PMCR().DP;
    enabled = enabled && dp == '0';
    // Otherwise whether event counting is prohibited or frozen does not affect the cycle
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // counter
    prohibited = FALSE;
    frozen = FALSE;
end;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p5) && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && (ss == SS_Secure || PSTATE.EL == EL3) then
        let sccd = if HaveAArch64() then MDCR_EL3().SCCD else SDCR().SCCD;
        if sccd == '1' then
            prohibited = TRUE;
        end;
    end;

    if PSTATE.EL == EL2 then
        let hccd = if HaveAArch64() then MDCR_EL2().HCCD else HDCR().HCCD;
        if hccd == '1' then
            prohibited = TRUE;
        end;
    end;
end;

// If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p7) && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && HaveAArch64() && MDCR_EL3().MCCD == '1' then
        prohibited = TRUE;
    end;
end;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
var filter : bits(32);
case idx of
    when INSTRUCTION_COUNTER_ID =>
        filter = PMICFILTR_EL0()[31:0];
    when CYCLE_COUNTER_ID =>
        filter = if HaveAArch64() then PMCCFILTR_EL0()[31:0] else PMCCFILTR();
    otherwise =>
        filter = if HaveAArch64() then PMEVTYPER_EL0(idx)[31:0] else PMEVTYPER(idx);
end;

let p : bit   = filter[31];
let u : bit   = filter[30];
let nsk : bit = if HaveEL(EL3) then filter[29] else '0';
let nsu : bit = if HaveEL(EL3) then filter[28] else '0';
let nsh : bit = if HaveEL(EL2) then filter[27] else '0';
let m : bit   = if HaveEL(EL3) && HaveAArch64() then filter[26] else '0';
let sh : bit  = if HaveEL(EL3) && IsFeatureImplemented(FEAT_SEL2) then filter[24] else '0';
let rlk : bit = if IsFeatureImplemented(FEAT_RME) then filter[22] else '0';
let rlu : bit = if IsFeatureImplemented(FEAT_RME) then filter[21] else '0';
let rlh : bit = if IsFeatureImplemented(FEAT_RME) then filter[20] else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0 =>
        case ss of
            when SS_NonSecure => filtered = u != nsu;
            when SS_Secure =>    filtered = u == '1';
            when SS_Realm =>     filtered = u != rlu;
        end;
    when EL1 =>
        case ss of
            when SS_NonSecure => filtered = p != nsk;
            when SS_Secure =>    filtered = p == '1';
            when SS_Realm =>     filtered = p != rlk;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
        when EL2 =>
            case ss of
                when SS_NonSecure => filtered = nsh == '0';
                when SS_Secure =>    filtered = nsh == sh;
                when SS_Realm =>     filtered = nsh == rlh;
            end;
        when EL3 =>
            if HaveAArch64() then
                filtered = m != p;
            else
                filtered = p == '1';
            end;
    end;

    if IsFeatureImplemented(FEAT_PMUv3_SME) then
        let is_streaming_mode : boolean = PSTATE.SM == '1';
        var vs : bits(2);
        case idx of
            when INSTRUCTION_COUNTER_ID =>
                vs = PMICFILTR_EL0().VS;
            when CYCLE_COUNTER_ID =>
                vs = PMCCFILTR_EL0().VS;
            otherwise =>
                vs = PMEVTYPER_EL0(idx).VS;
        end;

        var streaming_mode_filtered : boolean;
        if vs == '11' then
            streaming_mode_filtered = ConstrainUnpredictableBool(Unpredictable_RES_PMU_VS);
        else
            streaming_mode_filtered =  ((is_streaming_mode && vs[0] == '1') ||
                                        (!is_streaming_mode && vs[1] == '1'));
        end;

        filtered = filtered || streaming_mode_filtered;
    end;

    return !debug && enabled && !prohibited && !filtered && !frozen;
end;
```

J1.4.1.95 EffectiveEPMN

```
// EffectiveEPMN()
// ===============
// Returns the Effective value of PMCCR.EPMN.

func EffectiveEPMN() => bits(5)
begin
    let counters : integer = NUM_PMU_COUNTERS;
    var epmn_bits : bits(5);

    if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
        epmn_bits = PMCCR().EPMN;
        if UInt(epmn_bits) > counters then
            (-, epmn_bits) = ConstrainUnpredictableBits{5}(Unpredictable_RES_EPMN);
        end;
    else
        epmn_bits = counters[4:0];
    end;

    return epmn_bits;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.96 EffectiveHPMN

```
// EffectiveHPMN()
// ===============
// Returns the Effective value of MDCR_EL2.HPMN or HDCR.HPMN.

func EffectiveHPMN() => bits(5)
begin
    let counters : integer = UInt(EffectiveEPMN());
    var hpmn_bits : bits(5);

    if HaveEL(EL2) then     // Software can reserve some event counters for EL2
        hpmn_bits = if HaveAArch64() then MDCR_EL2().HPMN else HDCR().HPMN;

        // When FEAT_PMUv3_EXTPMN is implemented, out of range values are capped.
        if UInt(hpmn_bits) > counters && IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
            hpmn_bits = counters[4:0];
        end;

        if (UInt(hpmn_bits) > counters ||
              (!IsFeatureImplemented(FEAT_HPMN0) && IsZero(hpmn_bits))) then
            (-, hpmn_bits) = ConstrainUnpredictableBits{5}(Unpredictable_RES_HPMN);
        end;
    else
        hpmn_bits = counters[4:0];
    end;

    return hpmn_bits;
end;
```

J1.4.1.97 GetNumEventCountersAccessible

```
// GetNumEventCountersAccessible()
// ===============================
// Return the number of event counters that can be accessed at the current Exception level.

func GetNumEventCountersAccessible() => integer{0..NUM_PMU_COUNTERS}
begin
    var n : integer;

    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(EffectiveHPMN());
    else
        n = UInt(EffectiveEPMN());
    end;

    return n as integer{0..NUM_PMU_COUNTERS};
end;
```

J1.4.1.98 GetNumEventCountersSelfHosted

```
// GetNumEventCountersSelfHosted()
// ===============================
// Return the number of event counters that can be accessed by the Self-hosted software.

func GetNumEventCountersSelfHosted() => integer
begin
    if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
        return UInt(EffectiveEPMN());
    else
        return NUM_PMU_COUNTERS;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.99 GetPMUAccessMask

```
// GetPMUAccessMask()
// ==================
// Return a mask of the PMU counters accessible at the current Exception level

func GetPMUAccessMask() => bits(64)
begin
    var mask : bits(64) = Zeros{};

    // PMICNTR_EL0 is only accessible at EL0 using AArch64 when PMUSERENR_EL0.UEN is 1.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && !UsingAArch32() then
        assert IsFeatureImplemented(FEAT_PMUv3p9);
        if PSTATE.EL != EL0 || PMUSERENR_EL0().UEN == '1' then
            mask[INSTRUCTION_COUNTER_ID] = '1';
        end;
    end;

    // PMCCNTR_EL0 is always implemented and accessible
    mask[CYCLE_COUNTER_ID] = '1';

    // PMEVCNTR_EL0(n)
    let counters : integer{} = GetNumEventCountersAccessible();
    if counters > 0 then
        mask[counters-1:0] = Ones{counters};
    end;

    // Check EL0 ignore access conditions
    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
          PSTATE.EL == EL0 && PMUSERENR_EL0().UEN == '1') then
        mask = mask AND PMUACR_EL1();  // User access control
    end;

    return mask;
end;
```

J1.4.1.100 GetPMUCounterRange

```
// GetPMUCounterRange()
// ====================
// Returns the range that a counter is currently in.

func GetPMUCounterRange(n : integer) => PMUCounterRange
begin

    let counters : integer = NUM_PMU_COUNTERS;
    let epmn : integer = UInt(EffectiveEPMN());
    let hpmn : integer = UInt(EffectiveHPMN());

    if n < hpmn then
        return PMUCounterRange_R1;
    elsif n < epmn then
        return PMUCounterRange_R2;
    elsif n < counters then
        assert IsFeatureImplemented(FEAT_PMUv3_EXTPMN);
        return PMUCounterRange_R3;
    elsif n == CYCLE_COUNTER_ID then
        return PMUCounterRange_R1;
    elsif n == INSTRUCTION_COUNTER_ID then
        assert IsFeatureImplemented(FEAT_PMUv3_ICNTR);
        return PMUCounterRange_R1;
    else
        unreachable;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.101 GetPMUReadMask

```
// GetPMUReadMask()
// ================
// Return a mask of the PMU counters that can be read at the current
// Exception level.
// This mask masks reads from PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCLR_EL0.

func GetPMUReadMask() => bits(64)
begin
    var mask : bits(64) = GetPMUAccessMask();

    // Additional PMICNTR_EL0 accessibility checks. PMICNTR_EL0 controls read-as-zero
    // if a read of PMICFILTR_EL0 would be trapped to a higher Exception level.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && mask[INSTRUCTION_COUNTER_ID] == '1' then
        // Check for trap to EL3.
        if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3().EnPM2 == '0' then
            mask[INSTRUCTION_COUNTER_ID] = '0';
        end;

        // Check for trap to EL2.
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !ELIsInHost(EL0) then
            // If FEAT_PMUv3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            if ((HaveEL(EL3) && SCR_EL3().FGTEn2 == '0') ||
                  HDFGRTR2_EL2().nPMICFILTR_EL0 == '0') then
                mask[INSTRUCTION_COUNTER_ID] = '0';
            end;
        end;
    end;

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
end;
```

J1.4.1.102 GetPMUWriteMask

```
// GetPMUWriteMask()
// =================
// Return a mask of the PMU counters writable at the current Exception level.
// This mask masks writes to PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, PMOVSCLR_EL0, and PMZR_EL0.
// 'write_counter' is TRUE for a write to PMZR_EL0, when the counter is being
// updated, and FALSE for other cases when the controls are being updated.

func GetPMUWriteMask(write_counter : boolean) => bits(64)
begin
    var mask : bits(64) = GetPMUAccessMask();

    // Check EL0 ignore write conditions
    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
          PSTATE.EL == EL0 && PMUSERENR_EL0().UEN == '1') then
        if (IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
              PMUSERENR_EL0().IR == '1') then          // PMICNTR_EL0 read-only
            mask[INSTRUCTION_COUNTER_ID] = '0';
        end;
        if PMUSERENR_EL0().CR == '1' then              // PMCCNTR_EL0 read-only
            mask[CYCLE_COUNTER_ID] = '0';
        end;
        if PMUSERENR_EL0().ER == '1' then              // PMEVCNTR[n]_EL0 read-only
            mask[30:0] = Zeros{31};
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Additional PMICNTR_EL0 accessibility checks. PMICNTR_EL0 controls ignore writes
    // if a write of PMICFILTR_EL0 would be trapped to a higher Exception level.
    // Indirect writes to PMICNTR_EL0 (through PMZR_EL0) are ignored if a write of
    // PMICNTR_EL0 would be trapped to a higher Exception level.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && mask[INSTRUCTION_COUNTER_ID] == '1' then
        // Check for trap to EL3.
        if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3().EnPM2 == '0' then
            mask[INSTRUCTION_COUNTER_ID] = '0';
        end;

        // Check for trap to EL2.
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !ELIsInHost(EL0) then
            // If FEAT_PMUv3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            let fgt_bit : bit = (if write_counter then HDFGWTR2_EL2().nPMICNTR_EL0
                                 else HDFGWTR2_EL2().nPMICFILTR_EL0);
            if (HaveEL(EL3) && SCR_EL3().FGTEn2 == '0') || fgt_bit == '0' then
                mask[INSTRUCTION_COUNTER_ID] = '0';
            end;
        end;
    end;

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
end;
```

J1.4.1.103 HasElapsed64Cycles

```
// HasElapsed64Cycles()
// ====================
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

impdef func HasElapsed64Cycles() => boolean
begin
    return TRUE;
end;
```

J1.4.1.104 IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =============================
// Increment the instruction counter and possibly set overflow bits.

func IncrementInstructionCounter(increment : integer)
begin
    if CountPMUEvents(INSTRUCTION_COUNTER_ID) then
        let old_value : integer = UInt(PMICNTR_EL0());
        let new_value : integer = old_value + increment;
        PMICNTR_EL0()           = new_value[63:0];

        // The effective value of PMCR_EL0.LP is '1' for the instruction counter
        if old_value[64] != new_value[64] then
            PMOVSSET_EL0().F0 = '1';
        end;
    end;
    return;
end;
```

J1.4.1.105 IsRange3Counter

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// IsRange3Counter()
// =================
// Returns TRUE if the counter is in the third range.

func IsRange3Counter(n : integer) => boolean
begin
    return PMUCounterRange_R3 == GetPMUCounterRange(n);
end;
```

J1.4.1.106 PMUCaptureEvent

```
// PMUCaptureEvent()
// =================
// If permitted and enabled, generate a PMU snapshot Capture event.

func PMUCaptureEvent()
begin
    assert HaveEL(EL3) && IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64();
    let debug_state : boolean = Halted();

    if !PMUCaptureEventAllowed() then
        // Indicate a Capture event completed, unsuccessfully
        PMSSCR_EL1().[NC,SS] = '10';
        return;
    end;
    let counters : integer = NUM_PMU_COUNTERS;
    for idx = 0 to counters - 1 do
        PMEVCNTSVR_EL1(idx) = PMEVCNTR_EL0(idx) as PMEVCNTSVR_EL1_Type;
    end;
    PMCCNTSVR_EL1() = PMCCNTR_EL0() as PMCCNTSVR_EL1_Type;

    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        PMICNTSVR_EL1() = PMICNTR_EL0() as PMICNTSVR_EL1_Type;
    end;

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSCTL().SS == '1' then
        if pc_sample.valid && !debug_state then
            SetPCSRActive();
            SetPCSample();
        else
            PMPCSR()[31:0] = Ones{32};
        end;
    end;

    if (IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed(PSTATE.EL) &&
          BRBCR_EL1().FZPSS == '1' && (!HaveEL(EL2) || BRBCR_EL2().FZPSS == '1')) then
        BRBEFreeze();
    end;

    // Indicate a successful Capture event
    PMSSCR_EL1().[NC,SS] = '00';
    if !debug_state || ConstrainUnpredictableBool(Unpredictable_PMUSNAPSHOTEVENT) then
        PMUEvent(PMU_EVENT_PMU_SNAPSHOT);
    end;

    return;
end;
```

J1.4.1.107 PMUCaptureEventAllowed

```
// PMUCaptureEventAllowed()
// ========================
// Returns TRUE if PMU Capture events are allowed, and FALSE otherwise.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func PMUCaptureEventAllowed() => boolean
begin
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    end;

    if !PMUCaptureEventEnabled() || OSLockStatus() then
        return FALSE;
    elsif HaveEL(EL3) && MDCR_EL3().PMSSE != '01' then
        return MDCR_EL3().PMSSE == '11';
    elsif HaveEL(EL2) && MDCR_EL2().PMSSE != '01' then
        return MDCR_EL2().PMSSE == '11';
    else
        var pmsse_el1 : bits(2) = PMECR_EL1().SSE;
        if pmsse_el1 == '01' then            // Reserved value
            var c : Constraint;
            (c, pmsse_el1) = ConstrainUnpredictableBits{2}(Unpredictable_RESPMSSE);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00'; end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
        return pmsse_el1 == '11';
    end;
end;
```

J1.4.1.108 PMUCaptureEventEnabled

```
// PMUCaptureEventEnabled()
// ========================
// Returns TRUE if PMU Capture events are enabled, and FALSE otherwise.

func PMUCaptureEventEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    end;
    if HaveEL(EL3) && MDCR_EL3().PMSSE != '01' then
        return MDCR_EL3().PMSSE == '1x';
    elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
        return FALSE;
    elsif HaveEL(EL2) && MDCR_EL2().PMSSE != '01' then
        return MDCR_EL2().PMSSE == '1x';
    elsif ELUsingAArch32(EL1) then
        return FALSE;
    else
        var pmsse_el1 : bits(2) = PMECR_EL1().SSE;
        if pmsse_el1 == '01' then            // Reserved value
            var c : Constraint;
            (c, pmsse_el1) = ConstrainUnpredictableBits{2}(Unpredictable_RESPMSSE);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00'; end;
        end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 == '1x';
    end;
end;
```

J1.4.1.109 PMUCountValue

```
// PMUCountValue()
// ===============
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// 'Vb' is the base value of the event that event counter 'n' is configured to count.
// 'Vm' is the value to increment event counter 'n-1' by if 'n' is odd, zero otherwise.

func PMUCountValue(n : integer, Vb : integer, Vm : integer) => integer
begin
    assert (n MOD 2) == 1 || Vm == 0;
    assert n < NUM_PMU_COUNTERS;

    if !IsFeatureImplemented(FEAT_PMUv3_TH) || !HaveAArch64() then
        return Vb;
    end;

    let TH : integer = UInt(PMEVTYPER_EL0(n).TH);

    // Control register fields
    var tc : bits(3) = PMEVTYPER_EL0(n).TC;
    var te : bit = '0';
    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
        te = PMEVTYPER_EL0(n).TE;
    end;
    var tlc : bits(2) = '00';
    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        tlc = PMEVTYPER_EL0(n).TLC;
    end;

    // Check for reserved cases
    var c : Constraint;
    (c, tc, te, tlc) = ReservedPMUThreshold(n, tc, te, tlc);
    if c == Constraint_DISABLED then
        return Vb;
    end;
    // Otherwise the values returned by ReservedPMUThreshold must be defined values

    // Check if disabled. Note that this function will return the value of Vb when
    // the control register fields are all zero, even without this check.
    if tc == '000' && TH == 0 && te == '0' && tlc == '00' then
        return Vb;
    end;

    // Threshold condition
    var Ct : boolean;
    case tc[2:1] of
        when '00' => Ct = (Vb != TH);        // Disabled or not-equal
        when '01' => Ct = (Vb == TH);        // Equals
        when '10' => Ct = (Vb >= TH);        // Greater-than-or-equal
        when '11' => Ct = (Vb <  TH);        // Less-than
    end;

    var Vn : integer;
    if te == '1' then
        // Edge condition
        let Cp : boolean = PMULastThresholdValue[[n]];
        var Ce : boolean;
        var Ve : integer;
        case tc[1:0] of
            when '10' =>  Ce = (Cp != Ct);   // Both edges
            when 'x1' =>  Ce = (!Cp && Ct);  // Single edge
            otherwise =>  unreachable;     // Covered by ReservedPMUThreshold
        end;
        case tlc of
            when '00' =>  Ve = (if Ce then 1 else 0);
            when '10' =>  Ve = (if Ce then Vm else 0);
            otherwise =>  unreachable;     // Covered by ReservedPMUThreshold
        end;
        Vn = Ve;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Threshold condition
        var Vt : integer;
        case tc[0]::tlc of
            when '0 00' =>  Vt = (if Ct then Vb else 0);
            when '0 01' =>  Vt = (if Ct then Vb else Vm);
            when '0 10' =>  Vt = (if Ct then Vm else 0);
            when '1 00' =>  Vt = (if Ct then 1 else 0);
            when '1 01' =>  Vt = (if Ct then 1 else Vm);
            otherwise =>    unreachable;   // Covered by ReservedPMUThreshold
        end;
        Vn = Vt;
    end;

    PMULastThresholdValue[[n]] = Ct;

    return Vn;
end;
```

J1.4.1.110 PMUCounterRange

```
// PMUCounterRange
// ===============
// Enumerates the ranges to which an event counter belongs to.

type PMUCounterRange of enumeration {
    PMUCounterRange_R1,
    PMUCounterRange_R2,
    PMUCounterRange_R3
    };
```

J1.4.1.111 PMUEvent

```
// PMUEvent()
// ==========
// Generate a PMU event. By default, increment by 1.
func PMUEvent(pmuevent : bits(16))
begin
    PMUEvent(pmuevent, 1);
end;

// PMUEvent()
// ==========
// Accumulate a PMU Event.

func PMUEvent(pmuevent : bits(16), increment : integer)
begin
    let counters : integer = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            PMUEvent(pmuevent, increment, idx);
        end;
    end;

    if (HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
          pmuevent == PMU_EVENT_INST_RETIRED) then
        IncrementInstructionCounter(increment);
    end;

end;

// PMUEvent()
// ==========
// Accumulate a PMU Event for a specific event counter.

func PMUEvent(pmuevent : bits(16), increment : integer, idx : integer)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    if UsingAArch32() then
        if PMEVTYPER(idx).evtCount == pmuevent then
            PMUEventAccumulator[[idx]] = PMUEventAccumulator[[idx]] + increment;
        end;
    else
        if PMEVTYPER_EL0(idx).evtCount == pmuevent then
            PMUEventAccumulator[[idx]] = PMUEventAccumulator[[idx]] + increment;
        end;
    end;
end;
```

J1.4.1.112 PMUOverflowCondition

```
// PMUOverflowCondition()
// ======================
// Enumerates the reasons for which the PMU overflow condition is evaluated.

type PMUOverflowCondition of enumeration {
    PMUOverflowCondition_PMUException,
    PMUOverflowCondition_BRBEFreeze,
    PMUOverflowCondition_Freeze,
    PMUOverflowCondition_IRQ,
    PMUOverflowCondition_EDBGRQ
};
```

J1.4.1.113 PMUSwIncrement

```
// PMUSwIncrement()
// ================
// Generate PMU Events on a write to PMSWINC

func PMUSwIncrement(sw_incr_in : bits(64))
begin

    var sw_incr : bits(64)   = sw_incr_in;
    var mask : bits(31)      = Zeros{};
    let counters : integer{} = GetNumEventCountersAccessible();
    if counters > 0 then
        mask[counters-1:0] = Ones{counters};
    end;

    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
          PSTATE.EL == EL0 && PMUSERENR_EL0().[UEN,SW] == '10') then
        mask = mask AND PMUACR_EL1()[30:0];
    end;

    sw_incr = sw_incr AND ZeroExtend{64}(mask);
    for idx = 0 to 30 do
        if sw_incr[idx] == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
        end;
    end;

    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.114 ReservedPMUThreshold

```
// ReservedPMUThreshold()
// ======================
// Checks if the given PMEVTYPER_EL1().[TH,TE,TLC] values are reserved and will
// generate Constrained Unpredictable behavior, otherwise return Constraint_NONE.

func ReservedPMUThreshold(n : integer, tc_in : bits(3),
                          te_in : bit, tlc_in : bits(2)) => (Constraint, bits(3), bit, bits(2))
begin
    var tc : bits(3) = tc_in;
    var te : bit = te_in;
    var tlc : bits(2) = tlc_in;

    var reserved : boolean = FALSE;

    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
        if te == '1' && tc[1:0] == '00' then      // Edge condition
            reserved = TRUE;
        end;
    else
        te = '0';                                 // Control is RES0
    end;

    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        if tlc == '11' then                       // Reserved value
            reserved = TRUE;
        end;
        if te == '1' then                         // Edge condition
            if tlc == '01' then
                reserved = TRUE;
            end;
        else                                      // Threshold condition
            if tc[0] == '1' && tlc == '10' then
                reserved = TRUE;
            end;
        end;
    else
        tlc = '00';                               // Controls are RES0
    end;

    var c : Constraint = Constraint_NONE;
    if reserved then
        var unpred_reserved_bits : bits(6);
        (c, unpred_reserved_bits) = ConstrainUnpredictableBits{6}(Unpredictable_RESTC);
        tc  = unpred_reserved_bits[5:3];
        te  = unpred_reserved_bits[2];
        tlc = unpred_reserved_bits[1:0];
    end;

    return (c, tc, te, tlc);
end;
```

J1.4.1.115 SMEPMUEventPredicate

```
// SMEPMUEventPredicate()
// ======================
// Call the relevant PMU predication events based on the SME instruction properties.

func SMEPMUEventPredicate{N}(mask1 : bits(N), mask2 : bits(N), esize : integer)
begin
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    PMUEvent(PMU_EVENT_SME_PRED2_SPEC);
    if AllElementsActive{N}(mask1, esize) && AllElementsActive{N}(mask2, esize) then
        PMUEvent(PMU_EVENT_SME_PRED2_FULL_SPEC);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PMUEvent(PMU_EVENT_SME_PRED2_NOT_FULL_SPEC);
        if !AnyActiveElement{N}(mask1, esize) && !AnyActiveElement{N}(mask2, esize) then
            PMUEvent(PMU_EVENT_SME_PRED2_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SME_PRED2_PARTIAL_SPEC);
        end;
    end;
end;
```

J1.4.1.116 SVEPMUEventPredicate

```
// SVEPMUEventPredicate()
// ======================
// Call the relevant PMU predication events based on the SVE instruction properties.

func SVEPMUEventPredicate{N}(mask : bits(N), esize : integer)
begin
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    if AllElementsActive{N}(mask, esize) then
        PMUEvent(PMU_EVENT_SVE_PRED_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SVE_PRED_NOT_FULL_SPEC);
        if !AnyActiveElement{N}(mask, esize) then
            PMUEvent(PMU_EVENT_SVE_PRED_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SVE_PRED_PARTIAL_SPEC);
        end;
    end;
end;
```

J1.4.1.117 ShouldPMUFreeze

```
// ShouldPMUFreeze()
// =================

func ShouldPMUFreeze(r : PMUCounterRange) => boolean
begin
    let include_r1 : boolean = (r == PMUCounterRange_R1);
    let include_r2 : boolean = (r == PMUCounterRange_R2);
    let include_r3 : boolean = FALSE;

    if r == PMUCounterRange_R3 then
        return FALSE;
    end;

    let overflow : boolean = CheckPMUOverflowCondition(PMUOverflowCondition_Freeze,
                                                        include_r1, include_r2, include_r3);
    return overflow;
end;
```

J1.4.1.118 ZeroCycleCounter

```
// ZeroCycleCounter()
// ==================
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.C or PMCR.C.

func ZeroCycleCounter()
begin
    var mask : bits(64) = Zeros{};
    mask[CYCLE_COUNTER_ID] = '1';
    ZeroPMUCounters(mask);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.1.119 ZeroPMUCounters

```
// ZeroPMUCounters()
// =================
// Zero set of counters specified by the mask in 'val'.
// For a System register write to PMZR_EL0, 'val' is the value passed in X<t>.

func ZeroPMUCounters(val : bits(64))
begin
    let masked_val : bits(64) = val AND GetPMUWriteMask(TRUE);

    for idx = 0 to 63 do
        if masked_val[idx] == '1' && !IsRange3Counter(idx) then
            case idx of
                when INSTRUCTION_COUNTER_ID =>
                    PMICNTR_EL0() = Zeros{64};
                when CYCLE_COUNTER_ID =>
                    if !HaveAArch64() then
                        PMCCNTR() = Zeros{64};
                    else
                        PMCCNTR_EL0() = Zeros{64};
                    end;
                otherwise =>
                    if !HaveAArch64() then
                        PMEVCNTR(idx) = Zeros{32};
                    elsif IsFeatureImplemented(FEAT_PMUv3p5) then
                        PMEVCNTR_EL0(idx) = Zeros{64};
                    else
                        PMEVCNTR_EL0(idx)[31:0] = Zeros{32};
                    end;
            end;
        end;
    end;

    return;
end;
```

J1.4.1.120 CreatePCSample

```
// CreatePCSample()
// ================

func CreatePCSample()
begin
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
    if PCSRSuspended() then return; end;
    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr{64}();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR() else CONTEXTIDR_EL1()[31:0];
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        pc_sample.vmid = VMID();
        if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
              !ELUsingAArch32(EL2)) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2()[31:0];
        else
            pc_sample.contextidr_el2 = ARBITRARY : bits(32);
        end;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
end;
```

J1.4.1.121 PCSRSuspended

```
// PCSRSuspended()
// ===============
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

func PCSRSuspended() => boolean
begin
    if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSCTL().SS == '1' then
        return FALSE;
    end;
    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSCTL().IMP == '1' then
        return PMPCSCTL().EN == '0';
    end;
    return ImpDefBool("PCSR is suspended");
end;
```

J1.4.1.122 PCSample

```
// PCSample
// ========

type PCSample of record {
    valid : boolean,
    pc : bits(64),
    el : bits(2),
    rw : bit,
    ss : SecurityState,
    has_el2 : boolean,
    contextidr : bits(32),
    contextidr_el2 : bits(32),
    el0h : boolean,
    vmid : bits(NUM_VMIDBITS)
};


var pc_sample : PCSample;
```

J1.4.1.123 Read_EDPCSRlo

```
// Read_EDPCSRlo()
// ===============

func Read_EDPCSRlo(memory_mapped : boolean) => bits(32)
begin
    // The Software lock is OPTIONAL.
    let update : boolean = !memory_mapped || EDLSR().SLK == '0';// Software locked: no side-effects
    var sample : bits(32);
    if pc_sample.valid then
        sample = pc_sample.pc[31:0];
        if update then
            if IsFeatureImplemented(FEAT_VHE) && EDSCR().SC2 == '1' then
                EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros{24} else pc_sample.pc[55:32]);
                EDPCSRhi.EL = pc_sample.el;
                EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros{32} else pc_sample.pc[63:32]);
            end;
            EDCIDSR() = pc_sample.contextidr;
            if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
                  EDSCR().SC2 == '1') then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                EDVIDSR() = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                             else ARBITRARY : bits(32));
            else
                EDVIDSR().VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                                  then pc_sample.vmid else Zeros{NUM_VMIDBITS});
                EDVIDSR().NS = (if pc_sample.ss == SS_Secure then '0' else '1');
                EDVIDSR().E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR().E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                // The conditions for setting HV are not specified if PCSRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR().HV = (if !IsZero(EDPCSRhi) then '1'
                                else ImpDefBit("0 or 1"));
            end;
        end;
    else
        sample = Ones{32};
        if update then
            EDPCSRhi  = ARBITRARY : bits(32);
            EDCIDSR() = ARBITRARY : bits(32);
            EDVIDSR() = ARBITRARY : bits(32);
        end;
    end;

    return sample;
end;
```

J1.4.1.124 Read_PMPCSR

```
// Read_PMPCSR()
// =============

func Read_PMPCSR(memory_mapped : boolean) => bits(64)
begin
    // The Software lock is OPTIONAL.
    var update : boolean = !memory_mapped || PMLSR().SLK == '0'; // Software locked: no side-effects

    if IsFeatureImplemented(FEAT_PCSRv8p9) && update then
        if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSCTL().SS == '1' then
            update = FALSE;
        elsif PMPCSCTL().[IMP,EN] == '10' || (PMPCSCTL().IMP == '0' && PCSRSuspended()) then
            pc_sample.valid = FALSE;
            SetPCSRActive();
        end;
    end;

    if pc_sample.valid then
        if update then SetPCSample(); end;
        return PMPCSR();
    else
        if update then SetPCSRUnknown(); end;
        return (ARBITRARY : bits(32) :: Ones{32});
    end;
end;
```

J1.4.1.125 SetPCSRActive

```
// SetPCSRActive()
// ===============
// Sets PC Sample-based Profiling to active state.

func SetPCSRActive()
begin
    if PMPCSCTL().IMP == '1' then
        PMPCSCTL().EN = '1';
    // If PMPCSCTL.IMP reads as `0b0`, then PMPCSCTL.EN is RES0, and it is
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // IMPLEMENTATION DEFINED whether PSCR is suspended or active at reset.
    end;
end;
```

J1.4.1.126 SetPCSRUnknown

```
// SetPCSRUnknown()
// ================
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

func SetPCSRUnknown()
begin
    PMPCSR()[31:0]  = Ones{32};
    PMPCSR()[55:32] = ARBITRARY : bits(24);
    PMPCSR().EL     = ARBITRARY : bits(2);
    PMPCSR().NS     = ARBITRARY : bit;

    if IsFeatureImplemented(FEAT_PMUv3_EXT64) then
        PMCCIDSR()      = ARBITRARY : bits(64);
        PMVCIDSR().VMID = ARBITRARY : bits(NUM_VMIDBITS);
    end;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR()      = ARBITRARY : bits(32);
        PMCID2SR()      = ARBITRARY : bits(32);
        PMVIDSR().VMID  = ARBITRARY : bits(NUM_VMIDBITS);
    end;

    return;
end;
```

J1.4.1.127 SetPCSample

```
// SetPCSample()
// =============
// Sets the PC sample registers to the appropriate sample values.

func SetPCSample()
begin
    PMPCSR()[31:0] = pc_sample.pc[31:0];
    PMPCSR()[55:32] = (if pc_sample.rw == '0' then Zeros{24} else pc_sample.pc[55:32]);
    PMPCSR().EL = pc_sample.el;
    if IsFeatureImplemented(FEAT_RME) then
        case pc_sample.ss of
            when SS_Secure =>
                PMPCSR().NSE = '0'; PMPCSR().NS = '0';
            when SS_NonSecure =>
                PMPCSR().NSE = '0'; PMPCSR().NS = '1';
            when SS_Root =>
                PMPCSR().NSE = '1'; PMPCSR().NS = '0';
            when SS_Realm =>
                PMPCSR().NSE = '1'; PMPCSR().NS = '1';
        end;
    else
        PMPCSR().NS = (if pc_sample.ss == SS_Secure then '0' else '1');
    end;
    let contextidr_el2 : bits(32) = (if pc_sample.has_el2 then
                                     pc_sample.contextidr_el2 else ARBITRARY : bits(32));
    var vmid : bits(NUM_VMIDBITS) = ARBITRARY : bits(NUM_VMIDBITS);
    if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h then
        vmid = pc_sample.vmid;
    end;
    if IsFeatureImplemented(FEAT_PMUv3_EXT64) then
        PMCCIDSR() = contextidr_el2::pc_sample.contextidr;
        PMVCIDSR().VMID = vmid;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR() = pc_sample.contextidr;
        PMCID2SR() = contextidr_el2;
        PMVIDSR().VMID = vmid;
    end;

    return;
end;
```

J1.4.1.128 CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

func CheckSoftwareStep()
begin

    // Other self-hosted debug functions will call AArch32_GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64_GenerateDebugExceptions().
    let step_enabled : boolean = (!ELUsingAArch32(DebugTarget()) &&
                                  AArch64_GenerateDebugExceptions() &&
                                  MDSCR_EL1().SS == '1');
    let active_pending : boolean = step_enabled && PSTATE.SS == '0';   // active-pending
    if active_pending then
        AArch64_SoftwareStepException();
    end;
    ShouldAdvanceSS = TRUE;
    return;
end;
```

J1.4.1.129 DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

func DebugExceptionReturnSS{N}(spsr : bits(N)) => bit
begin
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    var enabled_at_source : boolean;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32_GenerateDebugExceptions();
    else
        enabled_at_source = AArch64_GenerateDebugExceptions();
    end;
    var valid : boolean;
    var dest_el : bits(2);
    if IllegalExceptionReturn{N}(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR{N}(spsr);  assert valid;
    end;

    let dest_ss : SecurityState = SecurityStateAtEL(dest_el);
    var enabled_at_dest : boolean;
    let dest_using_32 : boolean = (if dest_el == EL0 then spsr[4] == '1'
                                                     else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32_GenerateDebugExceptionsFrom(dest_el, dest_ss);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        let mask : bit = spsr[9];
        enabled_at_dest = AArch64_GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);
    end;

    let ELd : bits(2) = DebugTargetFrom(dest_ss);
    var SS_bit : bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1().SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr[21];
    else
        SS_bit = '0';
    end;
    return SS_bit;
end;
```

J1.4.1.130 SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

func SSAdvance()
begin

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // PE only needs to consider advancing the state machine from the active-not-pending
    // state.
    if !ShouldAdvanceSS then return; end;
    let target : bits(2) = DebugTarget();
    let step_enabled : boolean = !ELUsingAArch32(target) && MDSCR_EL1().SS == '1';
    let active_not_pending : boolean = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0'; end;
    ShouldAdvanceSS = FALSE;
    return;
end;
```

J1.4.1.131 SoftwareStepOpEnabled

```
// SoftwareStepOpEnabled()
// =======================
// Returns a boolean indicating if execution from MDSTEPOP_EL1 is enabled.

func SoftwareStepOpEnabled() => boolean
begin

    if !IsFeatureImplemented(FEAT_STEP2) || UsingAArch32() then
        return FALSE;
    end;

    let step_enabled = AArch64_GenerateDebugExceptions() && MDSCR_EL1().SS == '1';
    let active_not_pending = step_enabled && PSTATE.SS == '1';
    let stepop = (MDSCR_EL1().EnSTEPOP == '1' &&
                  (!HaveEL(EL3) || MDCR_EL3().EnSTEPOP == '1') &&
                  (!EL2Enabled() || MDCR_EL2().EnSTEPOP == '1'));
    return active_not_pending && stepop;
end;
```

J1.4.1.132 SoftwareStep_DidNotStep

```
// SoftwareStep_DidNotStep()
// =========================
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception.  Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.

func SoftwareStep_DidNotStep() => boolean
begin
    return FALSE;
end;
```

J1.4.1.133 SoftwareStep_SteppedEX

```
// SoftwareStep_SteppedEX()
// ========================
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test.  Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.

func SoftwareStep_SteppedEX() => boolean
begin
    return FALSE;
end;
```

J1.4.1.134 DataCacheWatchpointSize

```
// DataCacheWatchpointSize()
// =========================
// Return the IMPLEMENTATION DEFINED data cache watchpoint size
func DataCacheWatchpointSize() => integer
begin
    let size : integer = ImpDefInt("Data Cache Invalidate Watchpoint Size");
    assert IsPow2(size) && size >= 2^(UInt(CTR_EL0().DminLine) + 2) && size <= 2048;
    return size;
end;
```

J1.4.1.135 WatchpointInfo

```
// WatchpointInfo
// ==============
// Watchpoint related fields.

type WatchpointInfo of record {
    wptype : WatchpointType,     // Type of watchpoint matched
    maybe_false_match : boolean, // Watchpoint matches rounded range
    watchpt_num : integer,       // Matching watchpoint number
    value_match : boolean,       // Watchpoint match
    vaddress : bits(64)          // Matching Virtual Address
};
```

J1.4.1.136 WatchpointType

```
// WatchpointType
// ==============

type WatchpointType of enumeration {
    WatchpointType_Inactive,      // Watchpoint inactive or disabled
    WatchpointType_AddrMatch,     // Address Match watchpoint
    WatchpointType_AddrMismatch   // Address Mismatch watchpoint
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.2 shared/exceptions

This section includes the following pseudocode functions:

- EffectiveHCRX_EL2_TMEA
- EffectiveHCR_AMO
- EffectiveHCR_TEA
- EffectiveNMEA
- EffectiveSCR_EL3_TMEA
- PhysicalSErrorTarget
- SyncExternalAbortTarget
- ConditionSyndrome
- Exception
- ExceptionRecord
- ExceptionSyndrome
- Undefined

J1.4.2.1 EffectiveHCRX_EL2_TMEA

```
// EffectiveHCRX_EL2_TMEA()
// ========================
// Return the Effective value of HCRX_EL2.TMEA.

func EffectiveHCRX_EL2_TMEA() => bit
begin
    if (IsFeatureImplemented(FEAT_DoubleFault2) && EL2Enabled() &&
          !ELUsingAArch32(EL2) && IsHCRXEL2Enabled()) then
        return HCRX_EL2().TMEA;
    else
        return '0';
    end;
end;
```

J1.4.2.2 EffectiveHCR_AMO

```
// EffectiveHCR_AMO()
// ==================
// Return the Effective value of HCR_EL2.AMO.

func EffectiveHCR_AMO() => bit
begin
    if EffectiveTGE() == '1' then
        return (if ELUsingAArch32(EL2) || EffectiveHCR_EL2_E2H() == '0' then '1' else '0');
    elsif EL2Enabled() then
        return (if ELUsingAArch32(EL2) then HCR().AMO else HCR_EL2().AMO);
    else
        return '0';
    end;
end;
```

J1.4.2.3 EffectiveHCR_TEA

```
// EffectiveHCR_TEA()
// ==================
// Return the Effective value of HCR_EL2.TEA.

func EffectiveHCR_TEA() => bit
begin
    if EL2Enabled() && IsFeatureImplemented(FEAT_RAS) then
        return (if ELUsingAArch32(EL2) then HCR2().TEA else HCR_EL2().TEA);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return '0';
    end;
end;
```

J1.4.2.4 EffectiveNMEA

```
// EffectiveNMEA()
// ===============
// Return the Effective value of SCR_EL3.NMEA or SCTLR2_ELx.NMEA.

func EffectiveNMEA() => bit
begin
    if IsFeatureImplemented(FEAT_DoubleFault2) then
        if PSTATE.EL == EL3 && !UsingAArch32() then
            return SCR_EL3().NMEA;
        elsif (PSTATE.EL == EL2 || IsInHost()) && !ELUsingAArch32(EL2) then
            return (if IsSCTLR2EL2Enabled() then SCTLR2_EL2().NMEA else '0');
        elsif !ELUsingAArch32(EL1) then
            return (if IsSCTLR2EL1Enabled() then SCTLR2_EL1().NMEA else '0');
        else
            return '0';
        end;
    elsif IsFeatureImplemented(FEAT_DoubleFault) && PSTATE.EL == EL3 && !UsingAArch32() then
        return SCR_EL3().NMEA AND EffectiveEA();
    else
        return '0';
    end;
end;
```

J1.4.2.5 EffectiveSCR_EL3_TMEA

```
// EffectiveSCR_EL3_TMEA()
// =======================
// Return the Effective value of SCR_EL3.TMEA.

func EffectiveSCR_EL3_TMEA() => bit
begin
    if (IsFeatureImplemented(FEAT_DoubleFault2) && HaveEL(EL3) &&
          !ELUsingAArch32(EL3)) then
        return SCR_EL3().TMEA;
    else
        return '0';
    end;
end;
```

J1.4.2.6 PhysicalSErrorTarget

```
// PhysicalSErrorTarget()
// ======================
// Returns a tuple of whether SError exception can be taken and, if so, the
// target Exception level.
// If EL3 is implemented and using AArch32, then a target Exception level of
// EL1 means Abort mode, and EL3 means Monitor mode, including in Secure
// state when Abort mode is part of EL3.

func PhysicalSErrorTarget() => (boolean, bits(2))
begin
    if Halted() then
        return (TRUE, ARBITRARY : bits(2));
    end;

    let effective_ea : bit = EffectiveEA();
    let effective_amo : bit = EffectiveHCR_AMO();
    let effective_tge : bit = EffectiveTGE();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let effective_nmea : bit = EffectiveNMEA();

// When EL3 is implemented and using AArch32, the SCR.AW bit can allow PSTATE.A
// to mask SError exceptions in Non-secure state when SCR.EA is 1 and the Effective
// value of HCR.AMO is 0.
var effective_aw : bit;
if (ELUsingAArch32(EL3) && effective_ea == '1' &&
      CurrentSecurityState() == SS_NonSecure && effective_amo == '0') then
    effective_aw = SCR().AW;
else
    effective_aw = '0';
end;

// The exception is masked by software.
var masked : boolean;
case PSTATE.EL of
    when EL3 =>
        masked = (!UsingAArch32() && effective_ea == '0') || PSTATE.A == '1';
    when EL2 =>
        masked = ((effective_ea == '0' || effective_aw == '1') &&
                  ((!UsingAArch32() && effective_tge == '0' && effective_amo == '0') ||
                   PSTATE.A == '1'));
    when EL1, EL0 =>
        masked = ((effective_ea == '0' || effective_aw == '1') &&
                  effective_amo == '0' && PSTATE.A == '1');
end;

// When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the mask might be overridden.
masked = (masked && effective_nmea == '0');

// External debug might disable the exception in the current Security state.
// This is not relevant at EL3.
let intdis : boolean = PSTATE.EL != EL3 && ExternalDebugInterruptsDisabled(EL1);

var target_el : bits(2) = ARBITRARY : bits(2);
if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
    if !masked then target_el = EL3; end;

elsif EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
    target_el = EL2;
    masked = FALSE;

elsif (EffectiveHCRX_EL2_TMEA() == '1' && !intdis &&
         ((PSTATE.EL == EL1 && PSTATE.A == '1') ||
          (PSTATE.EL == EL0 && masked && !IsInHost()))) then
    target_el = EL2;
    masked = FALSE;

elsif (EffectiveSCR_EL3_TMEA() == '1' &&
         ((PSTATE.EL IN {EL2, EL1} && PSTATE.A == '1') ||
         (PSTATE.EL IN {EL2, EL0} && masked) || intdis)) then
    target_el = EL3;
    masked = FALSE;

elsif PSTATE.EL == EL2 || IsInHost() then
    if !masked then target_el = EL2; end;

else
    assert (PSTATE.EL == EL1 ||
              (PSTATE.EL == EL3 && ELUsingAArch32(EL3)) ||
              (PSTATE.EL == EL0 && !IsInHost()));
    if !masked then target_el = EL1; end;
end;

// External debug might disable the exception for the target Exception level.
if !masked && ExternalDebugInterruptsDisabled(target_el) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        masked = TRUE;
        target_el = ARBITRARY : bits(2);
    end;

    return (masked, target_el);
end;
```

J1.4.2.7 SyncExternalAbortTarget

```
// SyncExternalAbortTarget()
// =========================
// Returns the target Exception level for a Synchronous External Data or
// Instruction or Prefetch Abort.
// If EL3 is implemented and using AArch32, then a target Exception level of
// EL1 means Abort mode, and EL3 means Monitor mode, including in Secure
// state when Abort mode is part of EL3.

func SyncExternalAbortTarget(fault : FaultRecord) => bits(2)
begin
    let effective_ea : bit = EffectiveEA();
    let effective_tea : bit = EffectiveHCR_TEA();
    let effective_tge : bit = EffectiveTGE();

    var target_el : bits(2);
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        target_el = EL3;

    elsif (EL2Enabled() && PSTATE.EL IN {EL1, EL0} &&
           (effective_tea == '1' || IsSecondStage(fault) ||
            fault.accessdesc.acctype == AccessType_NV2 ||
            (PSTATE.EL == EL0 && effective_tge == '1'))) then
        target_el = EL2;

    elsif EffectiveHCRX_EL2_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL == EL1 then
        target_el = EL2;

    elsif EffectiveSCR_EL3_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL IN {EL1, EL2} then
        target_el = EL3;

    else
        assert PSTATE.EL != EL3 || ELUsingAArch32(EL3);
        target_el = (if PSTATE.EL == EL2 then EL2 else EL1);
    end;

    return target_el;
end;
```

J1.4.2.8 ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

func ConditionSyndrome() => bits(5)
begin

    var syndrome : bits(5);

    if UsingAArch32() then
        let cond = CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome[4] = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome[3:0] = '1110';
            else
                syndrome[3:0] = cond;
            end;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if ImpDefBool("Condition valid for trapped T32") then
                syndrome[4] = '1';
                syndrome[3:0] = cond;
            else
                syndrome[4] = '0';
                syndrome[3:0] = ARBITRARY : bits(4);
            end;
        end;
    else
        syndrome[4] = '1';
        syndrome[3:0] = '1110';
    end;
    return syndrome;
end;
```

J1.4.2.9 Exception

```
// Exception
// =========
// Classes of exception.

type Exception of enumeration {
        Exception_Uncategorized,        // Uncategorized or unknown reason
        Exception_WFxTrap,              // Trapped WFI or WFE instruction
        Exception_CP15RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b111
        Exception_CP15RRTTrap,          // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
        Exception_CP14RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b1110
        Exception_CP14DTTrap,           // Trapped AArch32 LDC or STC access, coproc=0b1110
        Exception_CP14RRTTrap,          // Trapped AArch32 MRRC access, coproc=0b1110
        Exception_AdvSIMDFPAccessTrap,  // HCPTR-trapped access to SIMD or FP
        Exception_FPIDTrap,             // Trapped access to SIMD or FP ID register
        Exception_OtherTrap,            // Trapped access to other instructions
        // Trapped BXJ instruction not supported in Armv8
        Exception_PACTrap,               // Trapped invalid PAC use
        Exception_IllegalState,          // Illegal Execution state
        Exception_SupervisorCall,        // Supervisor Call
        Exception_HypervisorCall,        // Hypervisor Call
        Exception_MonitorCall,           // Monitor Call or Trapped SMC instruction
        Exception_SystemRegisterTrap,    // Trapped MRS or MSR System register access
        Exception_ERetTrap,              // Trapped invalid ERET use
        Exception_InstructionAbort,      // Instruction Abort or Prefetch Abort
        Exception_PCAlignment,           // PC alignment fault
        Exception_DataAbort,             // Data Abort
        Exception_NV2DataAbort,          // Data abort at EL1 reported as being from EL2
        Exception_PACFail,               // PAC Authentication failure
        Exception_SPAlignment,           // SP alignment fault
        Exception_FPTrappedException,    // IEEE trapped FP exception
        Exception_SError,                // SError interrupt
        Exception_Breakpoint,            // (Hardware) Breakpoint
        Exception_SoftwareStep,          // Software Step
        Exception_Watchpoint,            // Watchpoint
        Exception_NV2Watchpoint,         // Watchpoint at EL1 reported as being from EL2
        Exception_SoftwareBreakpoint,    // Software Breakpoint Instruction
        Exception_VectorCatch,           // AArch32 Vector Catch
        Exception_IRQ,                   // IRQ interrupt
        Exception_SVEAccessTrap,         // HCPTR trapped access to SVE
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Exception_SMEAccessTrap,         // HCPTR trapped access to SME
        Exception_GPC,                   // Granule protection check
        Exception_BranchTarget,          // Branch Target Identification
        Exception_MemCpyMemSet,          // Exception from a CPY* or SET* instruction
        Exception_GCSFail,               // GCS Exceptions
        Exception_Profiling,             // Profiling exception
        Exception_SystemRegister128Trap, // Trapped MRRS or MSRR System register or SYSP access
        Exception_FIQ};                  // FIQ interrupt
```

J1.4.2.10 ExceptionRecord

```
// ExceptionRecord
// ===============

type ExceptionRecord of record {
    exceptype : Exception,           // Exception class
    syndrome : IssType,              // Syndrome record
    paddress : FullAddress,          // Physical fault address
    vaddress : bits(64),             // Virtual fault address
    ipavalid : boolean,              // Validity of Intermediate Physical fault address
    pavalid : boolean,               // Validity of Physical fault address
    NS : bit,                        // Intermediate Physical fault address space
    ipaddress : bits(NUM_PABITS),            // Intermediate Physical fault address
    trappedsyscallinst : boolean};   // Trapped SVC or SMC instruction
```

J1.4.2.11 ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

func ExceptionSyndrome(exceptype : Exception) => ExceptionRecord
begin

    var r : ExceptionRecord;

    r.exceptype      = exceptype;

    // Initialize all other fields
    r.syndrome.iss   = Zeros{25};
    r.syndrome.iss2  = Zeros{24};
    r.vaddress       = Zeros{64};
    r.ipavalid       = FALSE;
    r.pavalid        = FALSE;
    r.NS             = '0';
    r.ipaddress      = Zeros{NUM_PABITS};
    r.paddress.paspace = ARBITRARY : PASpace;
    r.paddress.address = ARBITRARY : bits(NUM_PABITS);
    r.trappedsyscallinst = FALSE;
    return r;
end;
```

J1.4.2.12 Undefined

```
// Undefined()
// ===========

noreturn func Undefined()
begin
    if UsingAArch32() then
        AArch32_Undefined();
    else
        AArch64_Undefined();
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3 shared/functions

This section includes the following pseudocode functions:

- EncodeLDFSC
- IPAValid
- IsAsyncAbort
- IsDebugException
- IsExternalAbort
- IsExternalAbortOnWalk
- IsExternalSyncAbort
- IsFault
- IsSErrorInterrupt
- IsSecondStage
- LSInstructionSyndrome
- ReportAsGPCException
- CACHE_OP
- CPASAtPAS
- CPASAtSecurityState
- CacheRecord
- DecodeSW
- GetCacheInfo
- IsZeroBit
- NormalizeReal
- RShr
- Reverse
- Signal
- counters
- AArch32_CheckTimerConditions
- AArch64_CheckTimerConditions
- CNTHCTL_EL2_VHE
- GenericCounterTick
- IsTimerConditionMet
- SetEventRegister
- TestEventCNTP
- TestEventCNTV
- VirtualCounterTimer
- BitReverse
- Poly32Mod2
- AESInvMixColumns
- AESInvShiftRows
- AESInvSubBytes
- AESMixColumns
- AESShiftRows
- AESSubBytes
- FFmul02
- FFmul03
- FFmul09

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FFmul0B
- FFmul0D
- FFmul0E
- SHA256hash
- SHA256hashSIGMA0
- SHA256hashSIGMA1
- SHAchoose
- SHAmajority
- SHAparity
- Sbox
- DecodeType
- EndOfDecode
- ClearExclusiveByAddress
- ClearExclusiveLocal
- ExclusiveMonitorsStatus
- IsExclusiveGlobal
- IsExclusiveLocal
- MarkExclusiveGlobal
- MarkExclusiveLocal
- ProcessorID
- HaveSoftwareLock
- HaveTraceExt
- InsertIESBBeforeException
- ActionRequired
- ClearPendingDelegatedSError
- ClearPendingPhysicalSError
- ClearPendingVirtualSError
- ErrorIsContained
- ErrorIsSynchronized
- ExtAbortToAArch64
- ExternalAbort
- ExternalFault
- FaultIsCorrected
- GetPendingPhysicalSError
- HandleExternalAbort
- HandleExternalReadAbort
- HandleExternalTTWAbort
- HandleExternalWriteAbort
- IsDelegatedSErrorPending
- IsExternalAbortTakenSynchronously
- IsPhysicalSErrorPending
- IsSErrorEdgeTriggered
- IsSynchronizablePhysicalSErrorPending
- IsVirtualSErrorPending
- PEErrorState
- PendSErrorInterrupt
- ReportErrorAsIMPDEF

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ReportErrorAsUC
- ReportErrorAsUER
- ReportErrorAsUEU
- ReportErrorAsUncategorized
- StateIsRecoverable
- BFAdd
- BFAdd_ZA
- BFClampScale
- BFDotAdd
- BFInfinity
- BFMatMulAddH
- BFMax
- BFMaxNum
- BFMin
- BFMinNum
- BFMul
- BFMulAdd
- BFMulAddH
- BFMulAddH_ZA
- BFMulAdd_ZA
- BFMulH
- BFNeg
- BFRound
- BFScale
- BFSub
- BFSub_ZA
- BFUnpack
- BFZero
- FPAdd_BF16
- FPConvertBF
- FPRoundBF
- FixedToFP
- BFConvertFP8
- FP8Bits
- FP8ConvertBF
- FP8ConvertFP
- FP8DecodeType
- FP8DefaultNaN
- FP8DotAddFP
- FP8Infinity
- FP8MatMulAddFP
- FP8MaxNormal
- FP8MulAddFP
- FP8Round
- FP8Type
- FP8Unpack
- FP8Zero

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FPConvertFP8
- FPAbs
- FPAbsMax
- FPAbsMin
- FPAdd
- FPAdd_ZA
- FPBits
- FPBitsType
- FPFracBits
- FPCompare
- FPCompareEQ
- FPCompareGE
- FPCompareGT
- FPConvert
- FPConvertNaN
- FPDecodeRM
- FPDecodeRounding
- FPDefaultNaN
- FPDiv
- FPDot
- FPDotAdd
- FPDotAdd_ZA
- FPExc
- FPInfinity
- FPMatMulAdd
- FPMatMulAddH
- FPMax
- FPMaxNormal
- FPMaxNum
- IsMerging
- FPMin
- FPMinNum
- FPMul
- FPMulAdd
- FPMulAdd_ZA
- FPMulAddH
- FPMulAddH_ZA
- FPProcessNaNs3H
- FPMulX
- FPNeg
- FPOnePointFive
- FPProcessDenorm
- FPProcessDenorms
- FPProcessDenorms3
- FPProcessDenorms4
- FPProcessException
- FPProcessNaN

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FPProcessNaNs
- FPProcessNaNs3
- FPProcessNaNs4
- FPRecipEstimate
- RecipEstimate
- FPRecpX
- FPRound
- FPRoundBase
- FPRoundCV
- FPRound_FP8
- FPRounding
- FPRoundingMode
- FPRoundInt
- FPRoundIntN
- FPRSqrtEstimate
- RecipSqrtEstimate
- FPSqrt
- FPSub
- FPSub_ZA
- FPThree
- FPToFixed
- FPToFixedJS
- FPTwo
- FPType
- FPUnpack
- FPUnpackBase
- FPUnpackCV
- FPZero
- VFPExpandImm
- AddWithCarry
- InterruptID
- SetInterruptRequestLevel
- AArch64_BranchAddr
- AccessDescriptor
- AccessType
- AddrTop
- AddressSize
- AlignmentEnforced
- AllInAlignedQuantity
- Allocation
- BigEndian
- BigEndianReverse
- CacheOp
- CacheOpScope
- CachePASpace
- CacheType
- Cacheability

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CreateAccDescA32LSMD
- CreateAccDescASIMD
- CreateAccDescASIMDAcqRel
- CreateAccDescAT
- CreateAccDescAcqRel
- CreateAccDescAtomicOp
- CreateAccDescDC
- CreateAccDescDCZero
- CreateAccDescExLDST
- CreateAccDescFPAtomicOp
- CreateAccDescGCS
- CreateAccDescGCSSS1
- CreateAccDescGPR
- CreateAccDescGPTW
- CreateAccDescHACDBS
- CreateAccDescHDBSS
- CreateAccDescIC
- CreateAccDescIFetch
- CreateAccDescLDAcqPC
- CreateAccDescLDGSTG
- CreateAccDescLOR
- CreateAccDescLS64
- CreateAccDescMOPS
- CreateAccDescNV2
- CreateAccDescRCW
- CreateAccDescS1TTW
- CreateAccDescS2TTW
- CreateAccDescSME
- CreateAccDescSPE
- CreateAccDescSTGMOPS
- CreateAccDescSVE
- CreateAccDescSVEFF
- CreateAccDescSVENF
- CreateAccDescTRBE
- CreateAccDescTTEUpdate
- DataMemoryBarrier
- DataSynchronizationBarrier
- DeviceType
- EffectiveMTX
- EffectiveTBI
- EffectiveTCMA
- ErrorState
- Fault
- FaultRecord
- FullAddress
- GPCF
- GPCFRecord

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Hint_Prefetch
- Hint_RangePrefetch
- IsContiguousSVEAccess
- IsRelaxedWatchpointAccess
- IsSIMDFPAccess
- IsSMEAccess
- IsWatchpointableAccess
- MBMaintenanceScope
- MBReqTypes
- MemAtomicOp
- MemAttrHints
- MemOp
- MemType
- Memory
- MemoryAttributes
- NewAccDesc
- PASpace
- Permissions
- PhysMemRead
- PhysMemRetStatus
- PhysMemWrite
- PrefetchHint
- S1AccessControls
- S2AccessControls
- Shareability
- SpeculativeStoreBypassBarrierToPA
- SpeculativeStoreBypassBarrierToVA
- Tag
- VARange
- AltPARTIDSpace
- AltPIDRealm
- AltPIDSecure
- DefaultMPAMInfo
- GenMPAM
- GenMPAMAtEL
- GenMPAMCurEL
- GenNewMPAMData
- GenPARTID
- GenPMG
- GetMPAM_PARTID
- GetMPAM_PMG
- MAP_vPARTID
- MPAM
- MPAMIsEnabled
- MPAMIsVirtual
- PARTIDSpaceFromSS
- UsePrimarySpaceEL10

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- UsePrimarySpaceEL2
- mapvpmw
- ASID
- ExecutionCntxt
- RESTRICT_PREDICTIONS
- RestrictType
- TargetSecurityState
- BranchTo
- BranchToAddr
- BranchType
- Branchtypetaken
- ESize
- EffectiveFPCR
- Hint_Branch
- NextInstrAddr
- ResetExternalDebugRegisters
- ThisInstrAddr
- UnimplementedIDRegister
- V
- Vpart
- _PC
- _R
- SPSR_ELx
- SPSR_curr
- AArch64_ChkFeat
- AddressAdd
- AddressInNaturallyAlignedBlock
- AddressIncrement
- BranchTargetCheck
- ClearEventRegister
- ConditionHolds
- ConsumptionOfSpeculativeDataBarrier
- CurrentInstrSet
- CurrentSecurityState
- DSBAlias
- EL0
- EL2Enabled
- EL3SDDUndef
- EL3SDDUndefPriority
- ELFromM32
- ELFromSPSR
- ELIsInHost
- ELStateUsingAArch32
- ELStateUsingAArch32K
- ELUsingAArch32
- ELUsingAArch32K
- EffectiveEA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EffectiveHCR_EL2_E2H
- EffectiveHCR_EL2_NVx
- EffectiveSCR_EL3_FIQ
- EffectiveSCR_EL3_IRQ
- EffectiveSCR_EL3_NS
- EffectiveSCR_EL3_NSE
- EffectiveSCR_EL3_RW
- EffectiveTGE
- EndOfInstruction
- EndOfInstructionException
- EndOfInstructionReason
- EnterLowPowerState
- EventRegister
- ExceptionalOccurrenceTargetState
- ExecuteAsNOP
- FIQPending
- GetAccumulatedFPExceptions
- GetLoadStoreType
- GetPSRFromPSTATE
- HaveAArch32
- HaveAArch32EL
- HaveAArch64
- HaveEL
- HaveELUsingSecurityState
- HaveSecureState
- HighestEL
- Hint_CLRBHB
- Hint_DGH
- Hint_StoreShared
- Hint_WFE
- Hint_WFET
- Hint_WFI
- Hint_WFIT
- Hint_Yield
- IRQPending
- IllegalExceptionReturn
- InstrSet
- InstructionFetchBarrier
- InstructionSynchronizationBarrier
- IsASEInstruction
- IsCurrentSecurityState
- IsEventRegisterSet
- IsHighestEL
- IsInHost
- IsSecureBelowEL3
- IsSecureEL2Enabled
- LocalTimeoutEvent

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Mode
- NonSecureOnlyImplementation
- PLOfEL
- PSTATE
- PhysicalCountInt
- PrivilegeLevel
- RestoredITBits
- SecureOnlyImplementation
- SecurityState
- SecurityStateAtEL
- SendEvent
- SendEventLocal
- SetAccumulatedFPExceptions
- SetPSTATEFromPSR
- ShouldAdvanceHS
- ShouldAdvanceIT
- ShouldAdvanceSS
- SmallestTranslationGranule
- SpeculationBarrier
- SynchronizeContext
- SynchronizeErrors
- TakeUnmaskedPhysicalSErrorInterrupts
- TakeUnmaskedSErrorInterrupts
- ThisInstr
- ThisInstrLength
- UndefinedInjectionCheck
- UsingAArch32
- ValidSecurityStateAtEL
- VirtualFIQPending
- VirtualIRQPending
- WFxType
- WaitForEvent
- WaitForInterrupt
- WatchpointRelatedSyndrome
- tlbi
- Broadcast
- BroadcastTLBI
- ExcludeXS
- TLBI
- TLBILevel
- TLBIMemAttr
- TLBIOp
- TLBIRecord
- VMID
- Maximum
- Unbounded
- ConstrainUnpredictable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ConstrainUnpredictableBits
- ConstrainUnpredictableBool
- ConstrainUnpredictableInteger
- ConstrainUnpredictableProcedure
- Constraint
- Unpredictable
- AdvSIMDExpandImm
- MatMulAdd
- PolynomialMult
- SatQ
- ShiftSat
- SignedSat
- SignedSatQ
- UnsignedRSqrtEstimate
- UnsignedRecipEstimate
- UnsignedSat
- UnsignedSatQ

J1.4.3.1 EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

func EncodeLDFSC(statuscode : Fault, level : integer) => bits(6)
begin
    var result : bits(6);

    // 128-bit descriptors will start from level -2 for 4KB to resolve bits IA[55:51]
    if level == -2 then
        assert IsFeatureImplemented(FEAT_D128);
        case statuscode of
            when Fault_AddressSize =>          result = '101100';
            when Fault_Translation =>          result = '101010';
            when Fault_SyncExternalOnWalk =>   result = '010010';
            when Fault_SyncParityOnWalk =>
                result = '011010';
                assert !IsFeatureImplemented(FEAT_RAS);
            when Fault_GPCFOnWalk =>           result = '100010';
            otherwise =>                       unreachable;
        end;
        return result;
    end;

    if level == -1 then
        assert IsFeatureImplemented(FEAT_LPA2);
        case statuscode of
            when Fault_AddressSize =>          result = '101001';
            when Fault_Translation =>          result = '101011';
            when Fault_SyncExternalOnWalk =>   result = '010011';
            when Fault_SyncParityOnWalk =>
                result = '011011';
                assert !IsFeatureImplemented(FEAT_RAS);
            when Fault_GPCFOnWalk =>           result = '100011';
            otherwise =>                       unreachable;
        end;

        return result;
    end;
    case statuscode of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Fault_AddressSize =>         result = '0000'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_AccessFlag =>          result = '0010'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_Permission =>          result = '0011'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_Translation   =>       result = '0001'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_AsyncExternal =>       result = '010001'; assert UsingAArch32();
        when Fault_SyncExternal  =>       result = '010000';
        when Fault_SyncExternalOnWalk =>  result = '0101'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_SyncParity    =>       result = '011000';
        when Fault_SyncParityOnWalk =>    result = '0111'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_AsyncParity   =>       result = '011001';
        when Fault_TagCheck =>            result = '010001'; assert IsFeatureImplemented(FEAT_MTE2);
        when Fault_Alignment =>           result = '100001';
        when Fault_Debug =>               result = '100010';
        when Fault_GPCFOnWalk =>          result = '1001'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_GPCFOnOutput =>        result = '101000';
        when Fault_TLBConflict =>         result = '110000';
        when Fault_UnsupportedAtomicHWUpdate =>
                                          result = '110001';
        when Fault_Lockdown =>            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive =>           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise =>                      unreachable;
    end;

    return result;
end;
```

J1.4.3.2 IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

func IPAValid(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;

    if fault.gpcf.gpf != GPCF_None then
        return fault.secondstage;
    elsif fault.s2fs1walk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
    end;
end;
```

J1.4.3.3 IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

func IsAsyncAbort(statuscode : Fault) => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});
end;

// IsAsyncAbort()
// ==============

func IsAsyncAbort(fault : FaultRecord) => boolean
begin
    return IsAsyncAbort(fault.statuscode);
end;
```

J1.4.3.4 IsDebugException

```
// IsDebugException()
// ==================

func IsDebugException(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
end;
```

J1.4.3.5 IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

readonly func IsExternalAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;

    return (statuscode IN {
              Fault_SyncExternal,
              Fault_SyncParity,
              Fault_SyncExternalOnWalk,
              Fault_SyncParityOnWalk,
              Fault_AsyncExternal,
              Fault_AsyncParity
    });
end;

// IsExternalAbort()
// =================

readonly func IsExternalAbort(fault : FaultRecord) => boolean
begin
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
end;
```

J1.4.3.6 IsExternalAbortOnWalk

```
// IsExternalAbortOnWalk()
// =======================

func IsExternalAbortOnWalk(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;
    return fault.statuscode IN {Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk};
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.7 IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

readonly func IsExternalSyncAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;
    if IsFeatureImplemented(FEAT_RAS) then
        assert ! statuscode IN {Fault_SyncParity, Fault_SyncParityOnWalk};
    end;

    return (statuscode IN {
              Fault_SyncExternal,
              Fault_SyncParity,
              Fault_SyncExternalOnWalk,
              Fault_SyncParityOnWalk
    });
end;

// IsExternalSyncAbort()
// =====================

readonly func IsExternalSyncAbort(fault : FaultRecord) => boolean
begin
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
end;
```

J1.4.3.8 IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

func IsFault(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.fault.statuscode != Fault_None;
end;

// IsFault()
// =========
// Return TRUE if a fault is associated with a memory access.

func IsFault(fault : Fault) => boolean
begin
    return fault != Fault_None;
end;

// IsFault()
// =========
// Return TRUE if a fault is associated with status returned by memory.

func IsFault(retstatus : PhysMemRetStatus) => boolean
begin
    return retstatus.statuscode != Fault_None;
end;
```

J1.4.3.9 IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func IsSErrorInterrupt(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});
end;

// IsSErrorInterrupt()
// ===================

func IsSErrorInterrupt(fault : FaultRecord) => boolean
begin
    return IsSErrorInterrupt(fault.statuscode);
end;


// Add a specific type of return value for FaultSyndrome
type IssType of record {
    iss : bits(25),
    iss2 : bits(24)
};
```

J1.4.3.10 IsSecondStage

```
// IsSecondStage()
// ===============

func IsSecondStage(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
end;
```

J1.4.3.11 LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =======================
// Returns the extended syndrome information for a second stage fault.
//  [10]  - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
//  [9:8] - Access size.
//  [7]   - Sign extended (for loads).
//  [6:2] - Transfer register.
//  [1]   - Transfer register is 64-bit.
//  [0]   - Instruction has acquire/release semantics.

impdef func LSInstructionSyndrome() => bits(11)
begin
    return ARBITRARY : bits(11);
end;
```

J1.4.3.12 ReportAsGPCException

```
// ReportAsGPCException()
// ======================
// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

func ReportAsGPCException(fault : FaultRecord) => boolean
begin
    assert IsFeatureImplemented(FEAT_RME);
    assert fault.statuscode IN {Fault_GPCFOnWalk, Fault_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF_None;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if Halted() && EDSCR().SDD == '1' then
        return FALSE;
    end;

    case fault.gpcf.gpf of
        when GPCF_Walk =>        return TRUE;
        when GPCF_AddressSize => return TRUE;
        when GPCF_EABT =>        return TRUE;
        when GPCF_Fail =>        return SCR_EL3().GPF == '1' && PSTATE.EL != EL3;
    end;
end;
```

J1.4.3.13 CACHE_OP

```
// CACHE_OP()
// ==========
// Performs Cache maintenance operations as per CacheRecord.

impdef func CACHE_OP(cache : CacheRecord)
begin
    return;
end;
```

J1.4.3.14 CPASAtPAS

```
// CPASAtPAS()
// ===========
// Get cache PA space for given PA space.

func CPASAtPAS(pas : PASpace) => CachePASpace
begin
    case pas of
        when PAS_NonSecure =>
            return CPAS_NonSecure;
        when PAS_Secure =>
            return CPAS_Secure;
        when PAS_Root =>
            return CPAS_Root;
        when PAS_Realm =>
            return CPAS_Realm;
        when PAS_SystemAgent =>
            return CPAS_SystemAgent;
        when PAS_NonSecureProtected =>
            return CPAS_NonSecureProtected;
        when PAS_NA6 =>
            return CPAS_NA6;
        when PAS_NA7 =>
            return CPAS_NA7;
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.15 CPASAtSecurityState

```
// CPASAtSecurityState()
// =====================
// Get cache PA space for given security state.

func CPASAtSecurityState(ss : SecurityState) => CachePASpace
begin
    case ss of
        when SS_NonSecure =>
            return CPAS_NonSecure;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when SS_Secure =>
            return CPAS_SecureNonSecure;
        when SS_Root =>
            return CPAS_Any;
        when SS_Realm =>
            return CPAS_RealmNonSecure;
    end;
end;
```

J1.4.3.16 CacheRecord

```
// CacheRecord
// ===========
// Details related to a cache operation.

type CacheRecord of record {
    acctype       : AccessType,        // Access type
    cacheop       : CacheOp,           // Cache operation
    opscope       : CacheOpScope,      // Cache operation type
    cachetype     : CacheType,         // Cache type
    regval        : bits(64),
    paddress      : FullAddress,
    vaddress      : bits(64),          // For VA operations
    setnum        : integer,           // For SW operations
    waynum        : integer,           // For SW operations
    level         : integer,           // For SW operations
    shareability  : Shareability,
    is_vmid_valid : boolean,           // is vmid valid for current context
    vmid          : bits(NUM_VMIDBITS),
    is_asid_valid : boolean,           // is asid valid for current context
    asid          : bits(NUM_ASIDBITS),
    security      : SecurityState,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    cpas : CachePASpace
};
```

J1.4.3.17 DecodeSW

```
// DecodeSW()
// ==========
// Decode input value into setnum, waynum and level for SW instructions.

func DecodeSW(regval : bits(64), cachetype : CacheType) => (integer, integer, integer)
begin
    let level : integer = UInt(regval[3:1]);
    let (numsets, associativity, linesize) = GetCacheInfo(level, cachetype);
    // For the given level and cachetype, get the number of sets, associativity and
    // cache line size in terms of actual bytes.

    let max_waybits : integer{} = if IsFeatureImplemented(FEAT_CCIDX) then 21 else 10;
    let max_setbits : integer{} = if IsFeatureImplemented(FEAT_CCIDX) then 24 else 15;

    let waybits : integer{} = CeilLog2(associativity) as integer{0..max_waybits};
    let setbits : integer{} = CeilLog2(numsets) as integer{0..max_setbits};
    let linebits : integer{} = FloorLog2(linesize) as integer{4..11};

    let waynum : integer = if associativity == 1 then 0 else UInt(regval[31:32-waybits]);
    let setnum : integer = if numsets == 1 then 0 else UInt(regval[linebits +: setbits]);

    return (setnum, waynum, level);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.18 GetCacheInfo

```
// GetCacheInfo()
// ==============
// Returns numsets, associativity & linesize in terms of actual bytes.

impdef func GetCacheInfo(level : integer, cachetype : CacheType) => (integer, integer, integer)
begin
    let numsets : integer = ImpDefInt("Numsets for DC SW instructions");
    let associativity : integer = ImpDefInt("Associativity for DC SW instructions");
    let linesize : integer = ImpDefInt("Linesize for DC SW instructions");
    return (numsets, associativity, linesize);
end;
```

J1.4.3.19 IsZeroBit

```
// IsZeroBit()
// ===========

func IsZeroBit{N}(x : bits(N)) => bit
begin
    return if IsZero(x) then '1' else '0';
end;
```

J1.4.3.20 NormalizeReal

```
// NormalizeReal
// =============
// Normalizes x to the form 1.xxx... x 2^y and returns (mantissa, exponent)

func NormalizeReal(x : real) => (real, integer)
begin
    let exponent : integer = ILog2(x);
    let mantissa : real = x / (2.0 ^ exponent);
    return (mantissa, exponent);
end;
```

J1.4.3.21 RShr

```
// RShr()
// ======
// Shift integer value right with rounding

func RShr(value : integer, shift : integer, round : boolean) => integer
begin
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
    end;
end;
```

J1.4.3.22 Reverse

```
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

func Reverse{N}(word : bits(N), M : integer) => bits(N)
begin
    assert N MOD M == 0;
    var result : bits(N);
    let swsize : integer{} = M as integer{1..N};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let sw : integer = N DIV swsize;
    for s = 0 to sw-1 do
        result[((sw - 1) - s)*:swsize] = word[s*:swsize];
    end;
    return result;
end;
```

J1.4.3.23 Signal

```
// Signal
// ======
// Available signal types

type Signal of enumeration {LOW, HIGH};
```

J1.4.3.24 counters

```
var PhysicalCount : bits(88);

var IsLocalTimeoutEventPending : boolean;
var LocalTimeoutVal : bits(64);           // Value to compare against the Virtual Counter Timer
                                          // to generate the local timeout event.
```

J1.4.3.25 AArch32_CheckTimerConditions

```
// AArch32_CheckTimerConditions()
// ==============================
// Checking timer conditions for all A32 timer registers

func AArch32_CheckTimerConditions()
begin
    var status : boolean;
    var offset : bits(64);
    offset = Zeros{64};
    assert !HaveAArch64();
    if HaveEL(EL3) then
        if CNTP_CTL_S().ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S(),
                                         CNTP_CTL_S().IMASK, InterruptID_CNTPS);
            CNTP_CTL_S().ISTATUS = if status then '1' else '0';
        end;

        if CNTP_CTL_NS().ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS(),
                                         CNTP_CTL_NS().IMASK, InterruptID_CNTP);
            CNTP_CTL_NS().ISTATUS = if status then '1' else '0';
        end;
    else
        if CNTP_CTL().ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL(),
                                         CNTP_CTL().IMASK, InterruptID_CNTP);
            CNTP_CTL().ISTATUS = if status then '1' else '0';
        end;
    end;
    if HaveEL(EL2) && CNTHP_CTL().ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL(),
                                     CNTHP_CTL().IMASK, InterruptID_CNTHP);
        CNTHP_CTL().ISTATUS = if status then '1' else '0';
    end;
    if CNTV_CTL_EL0().ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2(), CNTV_CVAL_EL0(),
                                     CNTV_CTL_EL0().IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0().ISTATUS = if status then '1' else '0';
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
end;
```

J1.4.3.26 AArch64_CheckTimerConditions

```
// AArch64_CheckTimerConditions()
// ==============================
// Checking timer conditions for all A64 timer registers

func AArch64_CheckTimerConditions()
begin
    var status : boolean;
    var offset : bits(64);
    var imask : bit;
    let ss : SecurityState = CurrentSecurityState();
    if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && !ELIsInHost(EL0) &&
          CNTHCTL_EL2().ECV == '1' && SCR_EL3().ECVEn == '1') then
        offset = CNTPOFF_EL2();
    else
        offset = Zeros{64};
    end;
    if CNTP_CTL_EL0().ENABLE == '1' then
        imask = CNTP_CTL_EL0().IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS_Root, SS_Realm} &&
              CNTHCTL_EL2().CNTPMASK == '1') then
            imask = '1';
        end;
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0(),
                                     imask, InterruptID_CNTP);
        CNTP_CTL_EL0().ISTATUS = if status then '1' else '0';
    end;
    if ((HaveEL(EL3) || (HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
       CNTHP_CTL_EL2().ENABLE == '1') then
        status = IsTimerConditionMet(Zeros{64}, CNTHP_CVAL_EL2(),
                                     CNTHP_CTL_EL2().IMASK, InterruptID_CNTHP);
        CNTHP_CTL_EL2().ISTATUS = if status then '1' else '0';
    end;
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && CNTHPS_CTL_EL2().ENABLE == '1' then
        status = IsTimerConditionMet(Zeros{64}, CNTHPS_CVAL_EL2(),
                                     CNTHPS_CTL_EL2().IMASK, InterruptID_CNTHPS);
        CNTHPS_CTL_EL2().ISTATUS = if status then '1' else '0';
    end;

    if CNTPS_CTL_EL1().ENABLE == '1' then
        status = IsTimerConditionMet(Zeros{64}, CNTPS_CVAL_EL1(),
                                     CNTPS_CTL_EL1().IMASK, InterruptID_CNTPS);
        CNTPS_CTL_EL1().ISTATUS = if status then '1' else '0';
    end;

    if CNTV_CTL_EL0().ENABLE == '1' then
        imask = CNTV_CTL_EL0().IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS_Root, SS_Realm} &&
              CNTHCTL_EL2().CNTVMASK == '1') then
            imask = '1';
        end;
        status = IsTimerConditionMet(CNTVOFF_EL2(), CNTV_CVAL_EL0(),
                                     imask, InterruptID_CNTV);
        CNTV_CTL_EL0().ISTATUS = if status then '1' else '0';
    end;

    if ((IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || !IsFeatureImplemented(FEAT_SEL2))) &&
          CNTHV_CTL_EL2().ENABLE == '1') then
        status = IsTimerConditionMet(Zeros{64}, CNTHV_CVAL_EL2(),
                                     CNTHV_CTL_EL2().IMASK, InterruptID_CNTHV);
        CNTHV_CTL_EL2().ISTATUS = if status then '1' else '0';
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ((IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE)) &&
          CNTHVS_CTL_EL2().ENABLE == '1') then
        status = IsTimerConditionMet(Zeros{64}, CNTHVS_CVAL_EL2(),
                                     CNTHVS_CTL_EL2().IMASK, InterruptID_CNTHVS);
        CNTHVS_CTL_EL2().ISTATUS = if status then '1' else '0';
    end;
    return;
end;
```

J1.4.3.27 CNTHCTL_EL2_VHE

```
// CNTHCTL_EL2_VHE()
// =================
// In the case where EL2 accesses the CNTKCTL_EL1 register, and the access
// is redirected to CNTHCTL_EL2 as a result of HCR_EL2.E2H being 1,
// then the bits of CNTHCTL_EL2 that are RES0 in CNTKCTL_EL1 are
// treated as being UNKNOWN. This function applies the UNKNOWN behavior.

func CNTHCTL_EL2_VHE(original_value : bits(64)) => bits(64)
begin
    assert PSTATE.EL == EL2;
    assert HCR_EL2().E2H == '1';

    var return_value : bits(64) = original_value;
    if !IsFeatureImplemented(FEAT_NV2p1) then
        return_value[19:18] = ARBITRARY : bits(2);
        return_value[16:10] = ARBITRARY : bits(7);
    end;
    return return_value;
end;
```

J1.4.3.28 GenericCounterTick

```
// GenericCounterTick()
// ====================
// Increments PhysicalCount value for every clock tick.

func GenericCounterTick()
begin
    var prev_physical_count : bits(64);
    if CNTCR().EN == '0' then
        if !HaveAArch64() then
            AArch32_CheckTimerConditions();
        else
            AArch64_CheckTimerConditions();
        end;
        return;
    end;
    prev_physical_count = PhysicalCountInt();
    if IsFeatureImplemented(FEAT_CNTSC) && CNTCR().SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend{88}(CNTSCR());
    else
        PhysicalCount[87:24] = PhysicalCount[87:24] + 1;
    end;
    if !HaveAArch64() then
        AArch32_CheckTimerConditions();
    else
        AArch64_CheckTimerConditions();
    end;
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.29 IsTimerConditionMet

```
// IsTimerConditionMet()
// =====================

func IsTimerConditionMet(offset : bits(64), compare_value : bits(64),
                         imask : bits(1), intid : InterruptID) => boolean
begin
    var condition_met : boolean;
    var level : Signal;
    condition_met = (UInt(PhysicalCountInt() - offset) - UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
end;
```

J1.4.3.30 SetEventRegister

```
// SetEventRegister()
// ==================
// Sets the Event Register of this PE

func SetEventRegister()
begin
    EventRegister = '1';
    return;
end;
```

J1.4.3.31 TestEventCNTP

```
// TestEventCNTP()
// ===============
// Generate Event stream from the physical counter

func TestEventCNTP(prev_physical_count : bits(64), current_physical_count : bits(64))
begin
    var offset : bits(64);
    var samplebit, previousbit : bit;
    var n : integer;
    if CNTHCTL_EL2().EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2().EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EVNTIS == '1' then
            n = n + 8;
        end;
        if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && !ELIsInHost(EL0) &&
             CNTHCTL_EL2().ECV == '1' && SCR_EL3().ECVEn == '1') then
            offset = CNTPOFF_EL2();
        else
            offset = Zeros{64};
        end;
        samplebit   = (current_physical_count - offset)[n];
        previousbit = (prev_physical_count - offset)[n];
        if CNTHCTL_EL2().EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister(); end;
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister(); end;
        end;
    end;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.32 TestEventCNTV

```
// TestEventCNTV()
// ===============
// Generate Event stream from the virtual counter

func TestEventCNTV(prev_physical_count : bits(64), current_physical_count : bits(64))
begin
    var offset : bits(64);
    var samplebit, previousbit : bit;
    var n : integer;
    if (EffectiveHCR_EL2_E2H()::EffectiveTGE() != '11' &&
          CNTKCTL_EL1().EVNTEN == '1') then
        n = UInt(CNTKCTL_EL1().EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTKCTL_EL1().EVNTIS == '1' then
            n = n + 8;
        end;
        offset      = if HaveEL(EL2) then CNTVOFF_EL2() else Zeros{64};

        samplebit   = (current_physical_count - offset)[n];
        previousbit = (prev_physical_count - offset)[n];
        if CNTKCTL_EL1().EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister(); end;
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister(); end;
        end;
    end;
    return;
end;
```

J1.4.3.33 VirtualCounterTimer

```
// VirtualCounterTimer()
// =====================
// Returns the Counter-Timer Virtual Count value, the value is as read by CurrentEL to CNTVCT_EL0.

func VirtualCounterTimer() => bits(64)
begin
    var cntvct : bits(64);

    if PSTATE.EL != EL3 then
        if HaveEL(EL2) && !ELIsInHost(PSTATE.EL) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2();
        else
            cntvct = PhysicalCountInt();
        end;
    else
        if HaveEL(EL2) && !ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF();
        else
            cntvct = PhysicalCountInt();
        end;
    end;

    return cntvct;
end;
```

J1.4.3.34 BitReverse

```
// BitReverse()
// ============

func BitReverse{N}(data : bits(N)) => bits(N)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var result : bits(N);
    for i = 0 to N-1 do
        result[(N-i)-1] = data[i];
    end;
    return result;
end;
```

J1.4.3.35 Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

func Poly32Mod2{N}(data_in : bits(N), poly : bits(32)) => bits(32)
begin
    assert N > 32;
    var data : bits(N) = data_in;
    for i = N-1 downto 32 do
        if data[i] == '1' then
            data[i-1:0] = data[i-1:0] XOR (poly::Zeros{i-32});
        end;
    end;
    return data[31:0];
end;
```

J1.4.3.36 AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

func AESInvMixColumns(op : bits (128)) => bits(128)
begin
    let in0 : bits(4*8) = op[ 96+:8] :: op[ 64+:8] :: op[ 32+:8] :: op[  0+:8];
    let in1 : bits(4*8) = op[104+:8] :: op[ 72+:8] :: op[ 40+:8] :: op[  8+:8];
    let in2 : bits(4*8) = op[112+:8] :: op[ 80+:8] :: op[ 48+:8] :: op[ 16+:8];
    let in3 : bits(4*8) = op[120+:8] :: op[ 88+:8] :: op[ 56+:8] :: op[ 24+:8];

    var out0 : bits(4*8);
    var out1 : bits(4*8);
    var out2 : bits(4*8);
    var out3 : bits(4*8);

    for c = 0 to 3 do
        out0[c*8+:8] =  (FFmul0E(in0[c*8+:8]) XOR FFmul0B(in1[c*8+:8]) XOR FFmul0D(in2[c*8+:8]) XOR
                         FFmul09(in3[c*8+:8]));
        out1[c*8+:8] =  (FFmul09(in0[c*8+:8]) XOR FFmul0E(in1[c*8+:8]) XOR FFmul0B(in2[c*8+:8]) XOR
                         FFmul0D(in3[c*8+:8]));
        out2[c*8+:8] =  (FFmul0D(in0[c*8+:8]) XOR FFmul09(in1[c*8+:8]) XOR FFmul0E(in2[c*8+:8]) XOR
                         FFmul0B(in3[c*8+:8]));
        out3[c*8+:8] =  (FFmul0B(in0[c*8+:8]) XOR FFmul0D(in1[c*8+:8]) XOR FFmul09(in2[c*8+:8]) XOR
                         FFmul0E(in3[c*8+:8]));
    end;

    return (
        out3[3*8+:8] :: out2[3*8+:8] :: out1[3*8+:8] :: out0[3*8+:8] ::
        out3[2*8+:8] :: out2[2*8+:8] :: out1[2*8+:8] :: out0[2*8+:8] ::
        out3[1*8+:8] :: out2[1*8+:8] :: out1[1*8+:8] :: out0[1*8+:8] ::
        out3[0*8+:8] :: out2[0*8+:8] :: out1[0*8+:8] :: out0[0*8+:8]
    );
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.37 AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

func AESInvShiftRows(op : bits(128)) => bits(128)
begin
    return (
        op[ 31: 24] :: op[ 55: 48] :: op[ 79: 72] :: op[103: 96] ::
        op[127:120] :: op[ 23: 16] :: op[ 47: 40] :: op[ 71: 64] ::
        op[ 95: 88] :: op[119:112] :: op[ 15:  8] :: op[ 39: 32] ::
        op[ 63: 56] :: op[ 87: 80] :: op[111:104] :: op[  7:  0]
    );
end;
```

J1.4.3.38 AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

func AESInvSubBytes(op : bits(128)) => bits(128)
begin
    // Inverse S-box values
    let GF2_inv : bits(16*16*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17[127:0] ::
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0[127:0] ::
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160[127:0] ::
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f[127:0] ::
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc[127:0] ::
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147[127:0] ::
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96[127:0] ::
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a[127:0] ::
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0[127:0] ::
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890[127:0] ::
        /*5*/ 0x849d8da75746155edab9edfd5048706c[127:0] ::
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872[127:0] ::
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08[127:0] ::
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54[127:0] ::
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c[127:0] ::
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952[127:0]
    );
    var out : bits(128);
    for i = 0 to 15 do
        out[i*8+:8] = GF2_inv[UInt(op[i*8+:8])*8+:8];
    end;
    return out;
end;
```

J1.4.3.39 AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

func AESMixColumns(op : bits (128)) => bits(128)
begin
    let in0 : bits(4*8) = op[ 96+:8] :: op[ 64+:8] :: op[ 32+:8] :: op[  0+:8];
    let in1 : bits(4*8) = op[104+:8] :: op[ 72+:8] :: op[ 40+:8] :: op[  8+:8];
    let in2 : bits(4*8) = op[112+:8] :: op[ 80+:8] :: op[ 48+:8] :: op[ 16+:8];
    let in3 : bits(4*8) = op[120+:8] :: op[ 88+:8] :: op[ 56+:8] :: op[ 24+:8];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var out0 : bits(4*8);
    var out1 : bits(4*8);
    var out2 : bits(4*8);
    var out3 : bits(4*8);

    for c = 0 to 3 do
        out0[c*8+:8] = (FFmul02(in0[c*8+:8]) XOR FFmul03(in1[c*8+:8]) XOR
                                 in2[c*8+:8]  XOR         in3[c*8+:8]);
        out1[c*8+:8] = (FFmul02(in1[c*8+:8])  XOR FFmul03(in2[c*8+:8]) XOR
                                in3[c*8+:8]   XOR         in0[c*8+:8]);
        out2[c*8+:8] = (FFmul02(in2[c*8+:8])  XOR FFmul03(in3[c*8+:8]) XOR
                                in0[c*8+:8]   XOR         in1[c*8+:8]);
        out3[c*8+:8] = (FFmul02(in3[c*8+:8])  XOR FFmul03(in0[c*8+:8]) XOR
                                in1[c*8+:8]   XOR         in2[c*8+:8]);
    end;

    return (
        out3[3*8+:8] :: out2[3*8+:8] :: out1[3*8+:8] :: out0[3*8+:8] ::
        out3[2*8+:8] :: out2[2*8+:8] :: out1[2*8+:8] :: out0[2*8+:8] ::
        out3[1*8+:8] :: out2[1*8+:8] :: out1[1*8+:8] :: out0[1*8+:8] ::
        out3[0*8+:8] :: out2[0*8+:8] :: out1[0*8+:8] :: out0[0*8+:8]
    );
end;
```

J1.4.3.40 AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

func AESShiftRows(op : bits(128)) => bits(128)
begin
    return (
        op[ 95: 88] :: op[ 55: 48] :: op[ 15:  8] :: op[103: 96] ::
        op[ 63: 56] :: op[ 23: 16] :: op[111:104] :: op[ 71: 64] ::
        op[ 31: 24] :: op[119:112] :: op[ 79: 72] :: op[ 39: 32] ::
        op[127:120] :: op[ 87: 80] :: op[ 47: 40] :: op[  7:  0]
    );
end;
```

J1.4.3.41 AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

func AESSubBytes(op : bits(128)) => bits(128)
begin
    // S-box values
    let GF2 : bits(16*16*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c[127:0] ::
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1[127:0] ::
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70[127:0] ::
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba[127:0] ::
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7[127:0] ::
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0[127:0] ::
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160[127:0] ::
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd[127:0] ::
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351[127:0] ::
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0[127:0] ::
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153[127:0] ::
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309[127:0] ::
        /*3*/ 0x75b227ebe28012079a059618c323c704[127:0] ::
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7[127:0] ::
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca[127:0] ::
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63[127:0]
    );
    var out : bits(128);
    for i = 0 to 15 do
        out[i*8+:8] = GF2[UInt(op[i*8+:8])*8+:8];
    end;
    return out;
end;
```

J1.4.3.42 FFmul02

```
// FFmul02()
// =========

func FFmul02(b : bits(8)) => bits(8)
begin
    let FFmul_02 : bits(256*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB[127:0] ::
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB[127:0] ::
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB[127:0] ::
        /*C*/ 0x858781838D8F898B959791939D9F999B[127:0] ::
        /*B*/ 0x656761636D6F696B757771737D7F797B[127:0] ::
        /*A*/ 0x454741434D4F494B555751535D5F595B[127:0] ::
        /*9*/ 0x252721232D2F292B353731333D3F393B[127:0] ::
        /*8*/ 0x050701030D0F090B151711131D1F191B[127:0] ::
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0[127:0] ::
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0[127:0] ::
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0[127:0] ::
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280[127:0] ::
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260[127:0] ::
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240[127:0] ::
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220[127:0] ::
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200[127:0]
    );
    return FFmul_02[UInt(b)*8+:8];
end;
```

J1.4.3.43 FFmul03

```
// FFmul03()
// =========

func FFmul03(b : bits(8)) => bits(8)
begin
    let FFmul_03 : bits(256*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B[127:0] ::
        /*E*/ 0x2A292C2F26252023323134373E3D383B[127:0] ::
        /*D*/ 0x7A797C7F76757073626164676E6D686B[127:0] ::
        /*C*/ 0x4A494C4F46454043525154575E5D585B[127:0] ::
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB[127:0] ::
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB[127:0] ::
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB[127:0] ::
        /*8*/ 0x8A898C8F86858083929194979E9D989B[127:0] ::
        /*7*/ 0x818287848D8E8B88999A9F9C95969390[127:0] ::
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0[127:0] ::
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0[127:0] ::
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0[127:0] ::
        /*3*/ 0x414247444D4E4B48595A5F5C55565350[127:0] ::
        /*2*/ 0x717277747D7E7B78696A6F6C65666360[127:0] ::
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        /*1*/ 0x212227242D2E2B28393A3F3C35363330[127:0] ::
        /*0*/ 0x111217141D1E1B18090A0F0C05060300[127:0]
    );
    return FFmul_03[UInt(b)*8+:8];
end;
```

J1.4.3.44 FFmul09

```
// FFmul09()
// =========

func FFmul09(b : bits(8)) => bits(8)
begin
    let FFmul_09 : bits(256*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831[127:0] ::
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1[127:0] ::
        /*D*/ 0x7D746F6659504B42353C272E1118030A[127:0] ::
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A[127:0] ::
        /*B*/ 0x3039222B141D060F78716A635C554E47[127:0] ::
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7[127:0] ::
        /*9*/ 0x0B0219102F263D34434A5158676E757C[127:0] ::
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC[127:0] ::
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD[127:0] ::
        /*6*/ 0x3A3328211E170C05727B6069565F444D[127:0] ::
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6[127:0] ::
        /*4*/ 0x0108131A252C373E49405B526D647F76[127:0] ::
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB[127:0] ::
        /*2*/ 0x4C455E5768617A73040D161F2029323B[127:0] ::
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990[127:0] ::
        /*0*/ 0x777E656C535A41483F362D241B120900[127:0]
    );
    return FFmul_09[UInt(b)*8+:8];
end;
```

J1.4.3.45 FFmul0B

```
// FFmul0B()
// =========

func FFmul0B(b : bits(8)) => bits(8)
begin
    let FFmul_0B : bits(256*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA[127:0] ::
        /*E*/ 0x1318050E3F3429224B405D56676C717A[127:0] ::
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1[127:0] ::
        /*C*/ 0x68637E75444F5259303B262D1C170A01[127:0] ::
        /*B*/ 0x555E434879726F640D061B10212A373C[127:0] ::
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C[127:0] ::
        /*9*/ 0x2E2538330209141F767D606B5A514C47[127:0] ::
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7[127:0] ::
        /*7*/ 0x545F424978736E650C071A11202B363D[127:0] ::
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D[127:0] ::
        /*5*/ 0x2F2439320308151E777C616A5B504D46[127:0] ::
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6[127:0] ::
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB[127:0] ::
        /*2*/ 0x1219040F3E3528234A415C57666D707B[127:0] ::
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0[127:0] ::
        /*0*/ 0x69627F74454E5358313A272C1D160B00[127:0]
    );
    return FFmul_0B[UInt(b)*8+:8];
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.46 FFmul0D

```
// FFmul0D()
// =========

func FFmul0D(b : bits(8)) => bits(8)
begin
    let FFmul_0D : bits(256*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC[127:0] ::
        /*E*/ 0x474A5D50737E69642F2235381B16010C[127:0] ::
        /*D*/ 0x2C21363B1815020F44495E53707D6A67[127:0] ::
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7[127:0] ::
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1[127:0] ::
        /*A*/ 0x2A27303D1E130409424F5855767B6C61[127:0] ::
        /*9*/ 0x414C5B5675786F622924333E1D10070A[127:0] ::
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA[127:0] ::
        /*7*/ 0x4D40575A7974636E25283F32111C0B06[127:0] ::
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6[127:0] ::
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD[127:0] ::
        /*4*/ 0x262B3C31121F08054E4354597A77606D[127:0] ::
        /*3*/ 0x202D3A3714190E034845525F7C71666B[127:0] ::
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB[127:0] ::
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0[127:0] ::
        /*0*/ 0x4B46515C7F726568232E3934171A0D00[127:0]
    );
    return FFmul_0D[UInt(b)*8+:8];
end;
```

J1.4.3.47 FFmul0E

```
// FFmul0E()
// =========

func FFmul0E(b : bits(8)) => bits(8)
begin
    let FFmul_0E : bits(256*8) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7[127:0] ::
        /*E*/ 0x6D63717F555B49471D13010F252B3937[127:0] ::
        /*D*/ 0x56584A446E60727C26283A341E10020C[127:0] ::
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC[127:0] ::
        /*B*/ 0x202E3C321816040A505E4C426866747A[127:0] ::
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A[127:0] ::
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1[127:0] ::
        /*8*/ 0x1B150709232D3F316B657779535D4F41[127:0] ::
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896[127:0] ::
        /*6*/ 0x2C22303E141A08065C52404E646A7876[127:0] ::
        /*5*/ 0x17190B052F21333D67697B755F51434D[127:0] ::
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD[127:0] ::
        /*3*/ 0x616F7D735957454B111F0D032927353B[127:0] ::
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB[127:0] ::
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0[127:0] ::
        /*0*/ 0x5A544648626C7E702A243638121C0E00[127:0]
    );
    return FFmul_0E[UInt(b)*8+:8];
end;
```

J1.4.3.48 SHA256hash

```
// SHA256hash()
// ============

func SHA256hash(x_in : bits (128), y_in : bits(128), w : bits(128), part1 : boolean) => bits(128)
begin
    var chs, maj, t : bits(32);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var x : bits(128) = x_in;
    var y : bits(128) = y_in;

    for e = 0 to 3 do
        chs = SHAchoose(y[31:0], y[63:32], y[95:64]);
        maj = SHAmajority{32}(x[31:0], x[63:32], x[95:64]);
        t   = y[127:96] + SHA256hashSIGMA1(y[31:0]) + chs + w[e*:32];
        x[127:96] = t + x[127:96];
        y[127:96] = t + SHA256hashSIGMA0(x[31:0]) + maj;
        let yx : bits(256) = ROL(y :: x, 32);
        (y, x) = (yx[128+:128], yx[0+:128]);
    end;
    return (if part1 then x else y);
end;
```

J1.4.3.49 SHA256hashSIGMA0

```
// SHA256hashSIGMA0()
// ==================

func SHA256hashSIGMA0(x : bits(32)) => bits(32)
begin
    return ROR(x, 2) XOR ROR(x, 13) XOR ROR(x, 22);
end;
```

J1.4.3.50 SHA256hashSIGMA1

```
// SHA256hashSIGMA1()
// ==================

func SHA256hashSIGMA1(x : bits(32)) => bits(32)
begin
    return ROR(x, 6) XOR ROR(x, 11) XOR ROR(x, 25);
end;
```

J1.4.3.51 SHAchoose

```
// SHAchoose()
// ===========
func SHAchoose(x : bits(32), y : bits(32), z : bits(32)) => bits(32)
begin
    return (((y XOR z) AND x) XOR z);
end;
```

J1.4.3.52 SHAmajority

```
// SHAmajority()
// =============

func SHAmajority{N}(x : bits(N), y : bits(N), z : bits(N)) => bits(N)
begin
    assert N IN {32, 64};
    return ((x AND y) OR ((x OR y) AND z));
end;
```

J1.4.3.53 SHAparity

```
// SHAparity()
// ===========

func SHAparity(x : bits(32), y : bits(32), z : bits(32)) => bits(32)
begin
    return (x XOR y XOR z);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.54 Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

func Sbox(sboxin : bits(8)) => bits(8)
begin
    var sboxout : bits(8);
    let sboxstring : bits(2048) = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05[127:0] ::
        /*E*/ 0x2b679a762abe04c3aa44132649860699[127:0] ::
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62[127:0] ::
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6[127:0] ::
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8[127:0] ::
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35[127:0] ::
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887[127:0] ::
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e[127:0] ::
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1[127:0] ::
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3[127:0] ::
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f[127:0] ::
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51[127:0] ::
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8[127:0] ::
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0[127:0] ::
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684[127:0] ::
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948[127:0]
    );
    let sboxindex : integer = 255 - UInt(sboxin);
    sboxout = sboxstring[sboxindex*:8];
    return sboxout;
end;
```

J1.4.3.55 DecodeType

```
// DecodeType
// ==========

type DecodeType of enumeration {
    Decode_UNDEF,
    Decode_NOP,
    Decode_OK
};
```

J1.4.3.56 EndOfDecode

```
// EndOfDecode()
// =============
// This function is invoked to end the Decode phase and performs Branch target Checks
// before taking any UNDEFINED exceptions, NOPs, or continuing to execute.

func EndOfDecode(reason : DecodeType)
begin
    if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
        BranchTargetCheck();
    end;
    case reason of
        when Decode_NOP   => ExecuteAsNOP();
        when Decode_UNDEF => Undefined();
        when Decode_OK    => pass;          // Continue to execute.
    end;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.57 ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =========================
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

impdef func ClearExclusiveByAddress(paddress : FullAddress,
                                    processorid : integer,
                                    size : integer)
begin
    return;
end;
```

J1.4.3.58 ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====================
// Clear the local Exclusives monitor for the specified processorid.

impdef func ClearExclusiveLocal(processorid : integer)
begin
    return;
end;
```

J1.4.3.59 ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =========================
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

impdef func ExclusiveMonitorsStatus() => bit
begin
    return '0';
end;
```

J1.4.3.60 IsExclusiveGlobal

```
// IsExclusiveGlobal()
// ===================
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.


impdef func IsExclusiveGlobal(paddress : FullAddress,
                              processorid : integer,
                              size : integer) => boolean
begin
    return TRUE;
end;
```

J1.4.3.61 IsExclusiveLocal

```
// IsExclusiveLocal()
// ==================
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

impdef func IsExclusiveLocal(paddress : FullAddress, processorid : integer,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                size : integer) => boolean
begin
    return FALSE;
end;
```

J1.4.3.62 MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====================
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

impdef func MarkExclusiveGlobal(paddress : FullAddress,
                                processorid : integer,
                                size : integer)
begin
    return;
end;
```

J1.4.3.63 MarkExclusiveLocal

```
// MarkExclusiveLocal()
// ====================
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

impdef func MarkExclusiveLocal(paddress : FullAddress, processorid : integer, size : integer)
begin
    return;
end;
```

J1.4.3.64 ProcessorID

```
// ProcessorID()
// =============
// Return the ID of the currently executing PE.

impdef func ProcessorID() => integer
begin
    if HaveAArch64() then
        return UInt(MPIDR_EL1().Aff3 :: MPIDR_EL1().Aff2 :: MPIDR_EL1().Aff1 :: MPIDR_EL1().Aff0);
    else
        return UInt(MPIDR().Aff2 :: MPIDR().Aff1 :: MPIDR().Aff0);
    end;
end;
```

J1.4.3.65 HaveSoftwareLock

```
// HaveSoftwareLock()
// ==================
// Returns TRUE if Software Lock is implemented.

func HaveSoftwareLock(component : Component) => boolean
begin
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        return FALSE;
    end;
    if IsFeatureImplemented(FEAT_DoPD) && component != Component_CTI then
        return FALSE;
    end;
    case component of
        when Component_ETE =>
            return ImpDefBool("ETE has Software Lock");
        when Component_Debug =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return ImpDefBool("Debug has Software Lock");
        when Component_PMU =>
            return ImpDefBool("PMU has Software Lock");
        when Component_CTI =>
            return ImpDefBool("CTI has Software Lock");
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.66 HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

readonly func HaveTraceExt() => boolean
begin
    return IsFeatureImplemented(FEAT_ETE) || IsFeatureImplemented(FEAT_ETMv4);
end;
```

J1.4.3.67 InsertIESBBeforeException

```
// InsertIESBBeforeException()
// ===========================
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

func InsertIESBBeforeException(el : bits(2)) => boolean
begin
    return (IsFeatureImplemented(FEAT_IESB) && ImpDefBool(
            "Has Implicit Error Synchronization Barrier before Exception"));
end;
```

J1.4.3.68 ActionRequired

```
// ActionRequired()
// ================
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

impdef func ActionRequired() => boolean
begin
    return TRUE;
end;
```

J1.4.3.69 ClearPendingDelegatedSError

```
// ClearPendingDelegatedSError()
// =============================
// Clear a pending delegated SError interrupt.

func ClearPendingDelegatedSError()
begin
    assert IsFeatureImplemented(FEAT_E3DSE);
    SCR_EL3().DSE = '0';
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.70 ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// ============================
// Clear a pending physical SError interrupt.

impdef func ClearPendingPhysicalSError()
begin
    return;
end;
```

J1.4.3.71 ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// ===========================
// Clear a pending virtual SError interrupt.

func ClearPendingVirtualSError()
begin
    if ELUsingAArch32(EL2) then
        HCR().VA      = '0';
    else
        HCR_EL2().VSE = '0';
    end;
end;
```

J1.4.3.72 ErrorIsContained

```
// ErrorIsContained()
// ==================
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

impdef func ErrorIsContained() => boolean
begin
    return TRUE;
end;
```

J1.4.3.73 ErrorIsSynchronized

```
// ErrorIsSynchronized()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is synchronized by any synchronization event
// FALSE otherwise.

impdef func ErrorIsSynchronized() => boolean
begin
    return TRUE;
end;
```

J1.4.3.74 ExtAbortToAArch64

```
// ExtAbortToAArch64()
// ===================
// Returns TRUE if synchronous exception is being taken to an Exception level using AArch64_

func ExtAbortToAArch64(fault : FaultRecord) => boolean
begin
    assert IsExternalSyncAbort(fault.statuscode);

    return !ELUsingAArch32(SyncExternalAbortTarget(fault));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.75 ExternalAbort

```
// ExternalAbort()
// ===============

func ExternalAbort(fault : FaultRecord) recurselimit 1
begin
    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32_Abort(fault);
        else
            AArch64_Abort(fault);
        end;

    else
        PendSErrorInterrupt(fault);
    end;
end;
```

J1.4.3.76 ExternalFault

```
// ExternalFault()
// ===============
// Return a fault recording indicating a fault for a Synchronous/Asynchronous External abort.

func ExternalFault(memretstatus : PhysMemRetStatus, iswrite : boolean,
                   memaddrdesc : AddressDescriptor, size : integer,
                   accdesc : AccessDescriptor) => FaultRecord
begin
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
           (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault_SyncParity,
                                                                           Fault_AsyncParity}));

    var fault : FaultRecord = NoFault(accdesc, memaddrdesc.vaddress);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
          ((IsFeatureImplemented(FEAT_RASv2) && ExtAbortToAArch64(fault) &&
            PEErrorState(fault) IN {ErrorState_UC, ErrorState_UEU}) ||
           !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                              size, accdesc))) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;
        end;
    end;

    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memretstatus.merrorstate;
    end;
    fault.paddress = memaddrdesc.paddress;
    return fault;
end;
```

J1.4.3.77 FaultIsCorrected

```
// FaultIsCorrected()
// ==================
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

readonly impdef func FaultIsCorrected() => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    return FALSE;
end;
```

J1.4.3.78 GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// ==========================
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

impdef func GetPendingPhysicalSError() => FaultRecord
begin
    return ARBITRARY : FaultRecord;
end;
```

J1.4.3.79 HandleExternalAbort

```
// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.

func HandleExternalAbort(memretstatus : PhysMemRetStatus, iswrite : boolean,
                         memaddrdesc : AddressDescriptor, size : integer,
                         accdesc : AccessDescriptor)
begin
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
            (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault_SyncParity,
                                                                            Fault_AsyncParity}));

    let fault : FaultRecord = ExternalFault(memretstatus, iswrite, memaddrdesc, size, accdesc);
    ExternalAbort(fault);
end;
```

J1.4.3.80 HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

func HandleExternalReadAbort(memstatus : PhysMemRetStatus, memaddrdesc : AddressDescriptor,
                             size : integer, accdesc : AccessDescriptor)
begin
    let iswrite : boolean = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
end;
```

J1.4.3.81 HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// ========================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

func HandleExternalTTWAbort(memretstatus : PhysMemRetStatus, iswrite : boolean,
                            memaddrdesc : AddressDescriptor,
                            accdesc : AccessDescriptor, size : integer,
                            input_fault : FaultRecord) => FaultRecord
begin
    var output_fault : FaultRecord = input_fault;
    output_fault.extflag    = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ((IsFeatureImplemented(FEAT_RASv2) && ExtAbortToAArch64(output_fault) &&
              PEErrorState(output_fault) IN {ErrorState_UC, ErrorState_UEU}) ||
             !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                                size, accdesc))) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;
        end;
    end;

    // If a synchronous fault is on a translation table walk, then update the fault type.
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
        end;
    end;
    if IsFeatureImplemented(FEAT_RAS) then
        output_fault.merrorstate = memretstatus.merrorstate;
    end;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    end;
    output_fault.paddress = memaddrdesc.paddress;
    return output_fault;
end;
```

J1.4.3.82 HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// ==========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

func HandleExternalWriteAbort(memstatus : PhysMemRetStatus, memaddrdesc : AddressDescriptor,
                              size : integer, accdesc : AccessDescriptor)
begin
    let iswrite : boolean = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
end;
```

J1.4.3.83 IsDelegatedSErrorPending

```
// IsDelegatedSErrorPending()
// ==========================
// Return TRUE if a delegated SError interrupt is pending.

func IsDelegatedSErrorPending() => boolean
begin
    return SCR_EL3().DSE == '1';
end;
```

J1.4.3.84 IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// ===================================
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.


impdef func IsExternalAbortTakenSynchronously(memstatus : PhysMemRetStatus, iswrite : boolean,
                                              desc : AddressDescriptor, size : integer,
                                              accdesc : AccessDescriptor) => boolean
begin
    return FALSE;
end;
```

J1.4.3.85 IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =========================
// Returns TRUE if a physical SError interrupt is pending.

impdef func IsPhysicalSErrorPending() => boolean
begin
    return FALSE;
end;
```

J1.4.3.86 IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

func IsSErrorEdgeTriggered() => boolean
begin
    if IsFeatureImplemented(FEAT_DoubleFault) then
        return TRUE;
    else
        return ImpDefBool("Edge-triggered SError");
    end;
end;
```

J1.4.3.87 IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =======================================
// Returns TRUE if a synchronizable physical SError interrupt is pending.

impdef func IsSynchronizablePhysicalSErrorPending() => boolean
begin
    return FALSE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.88 IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// ========================
// Return TRUE if a virtual SError interrupt is pending.

func IsVirtualSErrorPending() => boolean
begin
    if ELUsingAArch32(EL2) then
        return HCR().VA == '1';
    else
        return HCR_EL2().VSE == '1';
    end;
end;
```

J1.4.3.89 PEErrorState

```
// PEErrorState()
// ==============
// Returns the error state of the PE on taking an error exception:
// The PE error state reported to software through the exception syndrome also
// depends on how the exception is taken, and so might differ from the value
// returned from this function.

func PEErrorState(fault : FaultRecord) => ErrorState
begin
    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
         ReportErrorAsUC()) then
        return ErrorState_UC;
    end;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;
    end;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;
    end;

    return ErrorState_UEO;
end;
```

J1.4.3.90 PendSErrorInterrupt

```
// PendSErrorInterrupt()
// =====================
// Pend the SError Interrupt.

impdef func PendSErrorInterrupt(fault : FaultRecord)
begin
    return;
end;
```

J1.4.3.91 ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

impdef func ReportErrorAsIMPDEF() => boolean
begin
    return FALSE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.92 ReportErrorAsUC

```
// ReportErrorAsUC()
// =================
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

impdef func ReportErrorAsUC() => boolean
begin
    return FALSE;
end;
```

J1.4.3.93 ReportErrorAsUER

```
// ReportErrorAsUER()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

impdef func ReportErrorAsUER() => boolean
begin
    return FALSE;
end;
```

J1.4.3.94 ReportErrorAsUEU

```
// ReportErrorAsUEU()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Unrecoverable, FALSE otherwise.

impdef func ReportErrorAsUEU() => boolean
begin
    return FALSE;
end;
```

J1.4.3.95 ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()
// ============================
// Return an implementation specific value:
// returns TRUE if Error is uncategorized, FALSE otherwise.

impdef func ReportErrorAsUncategorized() => boolean
begin
    return FALSE;
end;
```

J1.4.3.96 StateIsRecoverable

```
// StateIsRecoverable()
// ====================
// Return an implementation specific value:
// returns TRUE is PE State is unrecoverable else FALSE.

impdef func StateIsRecoverable() => boolean
begin
    return TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.97 BFAdd

```
// BFAdd()
// =======
// Non-widening BFloat16 addition used by SVE2 instructions.

func BFAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFAdd{N}(op1, op2, fpcr, fpexc);
end;

// BFAdd()
// =======
// Non-widening BFloat16 addition following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 + op2.
// The 'fpcr' argument supplies the FPCR control bits.

func BFAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(2*N);

    let op1_s : bits(2*N) = op1 :: Zeros{N};
    let op2_s : bits(2*N) = op2 :: Zeros{N};
    let (type1,sign1,value1) = FPUnpack{2*N}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{2*N}(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs{2*N}(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        let inf1  = (type1 == FPType_Infinity);
        let inf2  = (type2 == FPType_Infinity);
        let zero1 = (type1 == FPType_Zero);
        let zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN{2*N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity{2*N}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity{2*N}('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero{2*N}(sign1);
        else
            let result_value : real = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{2*N}(result_sign);
            else
                result = FPRoundBF{2*N}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr); end;
    end;


    return result[2*N-1:N];
end;
```

J1.4.3.98 BFAdd_ZA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BFAdd_ZA()
// ==========
// Non-widening BFloat16 addition used by SME2 ZA-targeting instructions.

func BFAdd_ZA(op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = FALSE;
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFAdd{16}(op1, op2, fpcr, fpexc);
end;
```

J1.4.3.99 BFClampScale

```
// BFClampScale()
// ==============

func BFClampScale(op : bits(16), scale_in : integer) => integer
begin
    let E : integer = 8;
    let F : integer = 7;
    let exp : integer = UInt(op[14:7]);
    let emax : integer = (1 << E) - 1;
    let min_scale : integer = -(F + 1);
    let max_scale : integer = emax + (F + 1);

    let scale : integer = Max(min_scale - exp, Min(scale_in, max_scale - exp));
    return scale;
end;
```

J1.4.3.100 BFDotAdd

```
// BFDotAdd()
// ==========
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

func BFDotAdd(addend : bits(32), op1_a : bits(16), op1_b : bits(16),
              op2_a : bits(16), op2_b : bits(16), fpcr_in : FPCR_Type) => bits(32)
begin
    var fpcr : FPCR_Type = fpcr_in;
    var prod : bits(32);

    var result : bits(32);
    if !IsFeatureImplemented(FEAT_EBF16) || fpcr.EBF == '0' then   // Standard BFloat16 behaviors
        prod   = FPAdd_BF16(BFMulH(op1_a, op2_a, fpcr), BFMulH(op1_b, op2_b, fpcr), fpcr);
        result = FPAdd_BF16(addend, prod, fpcr);
    else                                                           // Extended BFloat16 behaviors
        let isbfloat16 : boolean = TRUE;
        let fpexc : boolean = FALSE;    // Do not generate floating-point exceptions
        fpcr.DN = '1';                  // Generate default NaN values
        prod    = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
        result  = FPAdd{32}(addend, prod, fpcr, fpexc);
    end;

    return result;
end;
```

J1.4.3.101 BFInfinity

```
// BFInfinity()
// ============

func BFInfinity{N}(sign : bit) => bits(N)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    assert N == 16;
    let E : integer{} = 8;
    let F : integer{} = N - (E + 1);
    return sign :: Ones{E} :: Zeros{F};
end;
```

J1.4.3.102 BFMatMulAddH

```
// BFMatMulAddH()
// ==============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

func BFMatMulAddH(addend : bits(128), op1 : bits(128),
                  op2 : bits(128), fpcr : FPCR_Type) => bits(128)
begin

    var result : bits(128);
    var sum : bits(32);

    for i = 0 to 1 do
        for j = 0 to 1 do
            sum = addend[(2*i + j)*:32];
            for k = 0 to 1 do
                let elt1_a : bits(16) = op1[(4*i + 2*k + 0)*:16];
                let elt1_b : bits(16) = op1[(4*i + 2*k + 1)*:16];
                let elt2_a : bits(16) = op2[(4*j + 2*k + 0)*:16];
                let elt2_b : bits(16) = op2[(4*j + 2*k + 1)*:16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, fpcr);
            end;
            result[(2*i + j)*:32] = sum;
        end;
    end;

    return result;
end;
```

J1.4.3.103 BFMax

```
// BFMax()
// =======
// BFloat16 maximum.

func BFMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return BFMax{N}(op1, op2, fpcr, altfp, fpexc);
end;

// BFMax()
// =======
// BFloat16 maximum.

func BFMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMax{N}(op1, op2, fpcr, altfp, fpexc);
end;

// BFMax()
// =======
// BFloat16 maximum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Compare op1 and op2 and return the larger value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func BFMax{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
              altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    var fpcr : FPCR_Type = fpcr_in;
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(2*N);

    let op1_s : bits(2*N) = op1 :: Zeros{N};
    let op2_s : bits(2*N) = op2 :: Zeros{N};
    let (type1,sign1,value1) = FPUnpack{2*N}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{2*N}(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero{N}(sign2);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FPType_Zero then BFZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{2*N}(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        var fptype : FPType;
        var sign : bit;
        var value : real;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FPType_Infinity then
            result = FPInfinity{2*N}(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;              // Use most positive sign
            result = FPZero{2*N}(sign);
        else
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            end;
            result = FPRoundBF{2*N}(value, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr); end;
    end;

    return result[2*N-1:N];
end;
```

J1.4.3.104 BFMaxNum

```
// BFMaxNum()
// ==========

func BFMaxNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMaxNum{N}(op1, op2, fpcr, fpexc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BFMaxNum()
// ==========
// BFloat16 maximum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMaxNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    let isbfloat16 : boolean = TRUE;
    var op1 : bits(N)   = op1_in;
    var op2 : bits(N)   = op2_in;
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    var result : bits(N);

    let (type1,-,-) = FPUnpackBase{N}(op1, fpcr, fpexc, isbfloat16);
    let (type2,-,-) = FPUnpackBase{N}(op2, fpcr, fpexc, isbfloat16);

    let type1_nan : boolean = type1 IN {FPType_QNaN, FPType_SNaN};
    let type2_nan : boolean = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity{N}('1');
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity{N}('1');
        end;
    end;

    let altfmaxfmin : boolean = FALSE;    // Do not use alternate NaN handling
    result = BFMax{N}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

J1.4.3.105 BFMin

```
// BFMin()
// =======
// BFloat16 minimum.

func BFMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return BFMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// BFMin()
// =======
// BFloat16 minimum.

func BFMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// BFMin()
// =======
// BFloat16 minimum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// if the function should use alternative floating-point behavior.

func BFMin{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
              altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    var fpcr : FPCR_Type = fpcr_in;
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(2*N);

    let op1_s : bits(2*N) = op1 :: Zeros{N};
    let op2_s : bits(2*N) = op2 :: Zeros{N};
    let (type1,sign1,value1) = FPUnpack{2*N}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{2*N}(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero{N}(sign2);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FPType_Zero then BFZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{2*N}(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        var fptype : FPType;
        var sign : bit;
        var value : real;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FPType_Infinity then
            result = FPInfinity{2*N}(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero{2*N}(sign);
        else
            if altfp then                       // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            end;
            result = FPRoundBF{2*N}(value, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr); end;
    end;

    return result[2*N-1:N];
end;
```

J1.4.3.106 BFMinNum

```
// BFMinNum()
// ==========

func BFMinNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMinNum{N}(op1, op2, fpcr, fpexc);
end;

// BFMinNum()
// ==========
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BFloat16 minimum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMinNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    let isbfloat16 : boolean = TRUE;
    var op1 : bits(N)   = op1_in;
    var op2 : bits(N)   = op2_in;
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    var result : bits(N);

    let (type1,-,-) = FPUnpackBase{N}(op1, fpcr, fpexc, isbfloat16);
    let (type2,-,-) = FPUnpackBase{N}(op2, fpcr, fpexc, isbfloat16);

    let type1_nan : boolean = type1 IN {FPType_QNaN, FPType_SNaN};
    let type2_nan : boolean = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity{N}('0');
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity{N}('0');
        end;
    end;

    let altfmaxfmin : boolean = FALSE;    // Do not use alternate NaN handling
    result = BFMin{N}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

J1.4.3.107 BFMul

```
// BFMul()
// =======
// Non-widening BFloat16 multiply used by SVE2 instructions.

func BFMul(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);
end;

// BFMul()
// =======
// Non-widening BFloat16 multiply following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 * op2.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMul(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type, fpexc : boolean) => bits(16)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(32);

    let op1_s : bits(32) = op1 :: Zeros{16};
    let op2_s : bits(32) = op2 :: Zeros{16};
    let (type1,sign1,value1) = FPUnpack{32}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{32}(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs{32}(type1, type2, op1_s, op2_s, fpcr, fpexc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !done then
        let inf1  = (type1 == FPType_Infinity);
        let inf2  = (type2 == FPType_Infinity);
        let zero1 = (type1 == FPType_Zero);
        let zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        elsif inf1 || inf2 then
            result = FPInfinity{32}(sign1 XOR sign2);
        elsif zero1 || zero2 then
            result = FPZero{32}(sign1 XOR sign2);
        else
            result = FPRoundBF{32}(value1*value2, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, 32, fpcr); end;
    end;

    return result[31:16];
end;
```

J1.4.3.108 BFMulAdd

```
// BFMulAdd()
// ==========
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

func BFMulAdd(addend : bits(16), op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
end;

// BFMulAdd()
// ==========
// Non-widening BFloat16 fused multiply-add following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates addend + op1*op2 with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMulAdd(addend : bits(16), op1 : bits(16), op2 : bits(16),
              fpcr : FPCR_Type, fpexc : boolean) => bits(16)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(32);

    let addend_s : bits(32)  = addend :: Zeros{16};
    let op1_s : bits(32)     = op1 :: Zeros{16};
    let op2_s : bits(32)     = op2 :: Zeros{16};
    let (typeA,signA,valueA) = FPUnpack{32}(addend_s, fpcr, fpexc);
    let (type1,sign1,value1) = FPUnpack{32}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{32}(op2_s, fpcr, fpexc);

    let inf1  = (type1 == FPType_Infinity);
    let inf2  = (type2 == FPType_Infinity);
    let zero1 = (type1 == FPType_Zero);
    let zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3{32}(typeA, type1, type2, addend_s, op1_s, op2_s, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        end;
    end;

    if !done then
        let infA  = (typeA == FPType_Infinity);
        let zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        let signP = sign1 XOR sign2;
        let infP  = inf1 || inf2;
        let zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        let invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity{32}('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity{32}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero{32}(signA);

        // Otherwise calculate numerical result and round it.
        else
            let result_value : real = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRoundBF{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, 32, fpcr);
        end;
    end;

    return result[31:16];
end;
```

J1.4.3.109 BFMulAddH

```
// BFMulAddH()
// ===========
// Used by BFMLALB, BFMLALT, BFMLSLB and BFMLSLT instructions.

func BFMulAddH(addend : bits(32), op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(32)
begin
    let value1 : bits(32) = op1 :: Zeros{16};
    let value2 : bits(32) = op2 :: Zeros{16};
    var fpcr : FPCR_Type  = fpcr_in;
    let altfp : boolean   = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    // When using alternative floating-point behavior, do not generate floating-point exceptions
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let fpexc : boolean   = !altfp;
    if altfp then fpcr.[FIZ,FZ] = '11'; end;                        // Flush denormal input and
                                                                    // output to zero
    if altfp then fpcr.RMode    = '00'; end;                        // Use RNE rounding mode
    return FPMulAdd{32}(addend, value1, value2, fpcr, fpexc);
end;
```

J1.4.3.110 BFMulAddH_ZA

```
// BFMulAddH_ZA()
// ==============
// Used by SME2 ZA-targeting BFMLAL and BFMLSL instructions.

func BFMulAddH_ZA(addend : bits(32), op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    let value1 : bits(32) = op1 :: Zeros{16};
    let value2 : bits(32) = op2 :: Zeros{16};
    return FPMulAdd_ZA{32}(addend, value1, value2, fpcr);
end;
```

J1.4.3.111 BFMulAdd_ZA

```
// BFMulAdd_ZA()
// =============
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting instructions.

func BFMulAdd_ZA(addend : bits(16), op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(16)
begin
    let fpexc : boolean  = FALSE;
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
end;
```

J1.4.3.112 BFMulH

```
// BFMulH()
// ========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

func BFMulH(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    var result : bits(32);

    let (type1,sign1,value1) = BFUnpack{16}(op1);
    let (type2,sign2,value2) = BFUnpack{16}(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN{32}(fpcr);
    else
        let inf1  = (type1 == FPType_Infinity);
        let inf2  = (type2 == FPType_Infinity);
        let zero1 = (type1 == FPType_Zero);
        let zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{32}(fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity{32}(sign1 XOR sign2);
        elsif zero1 || zero2 then
            result = FPZero{32}(sign1 XOR sign2);
        else
            result = BFRound(value1*value2);
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return result;
end;
```

J1.4.3.113 BFNeg

```
// BFNeg()
// =======

func BFNeg(op : bits(16)) => bits(16)
begin
    let honor_altfp : boolean = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_altfp);
end;

// BFNeg()
// =======

func BFNeg(op : bits(16), honor_altfp : boolean) => bits(16)
begin
    if honor_altfp && !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if FPCR().AH == '1' then
            let fpexc : boolean = FALSE;
            let isbfloat16 : boolean = TRUE;
            let (fptype, -, -) = FPUnpackBase{16}(op, FPCR(), fpexc, isbfloat16);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When FPCR().AH=1, sign of NaN has no consequence
            end;
        end;
    end;
    return NOT(op[15]) :: op[14:0];
end;
```

J1.4.3.114 BFRound

```
// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

func BFRound(op : real) => bits(32)
begin
    assert op != 0.0;
    var result : bits(32);

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    let minimum_exp : integer = -126;  let E : integer{} = 8;  let F : integer{} = 23;

    // Split value into sign, unrounded mantissa and exponent.
    var sign : bit;
    var exponent : integer;
    var mantissa : real;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    end;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero{32}(sign);
    end;

    // Start creating the exponent value for the result. Start by biasing the actual exponent
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    let biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    var int_mant : integer = RoundDown(mantissa * 2.0^F);
    let error : real = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 && int_mant[0] == '0' then
        int_mant = int_mant + 1;
    end;

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity{32}(sign);      // Overflows generate appropriately-signed Infinity
    else
        result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0];
    end;

    return result;
end;
```

J1.4.3.115 BFScale

```
// BFScale()
// =========
// Scales BFloat16 operand by 2.0 to the power of the signed integer value.

func BFScale(op : bits(16), scale : integer, fpcr : FPCR_Type) => bits(16)
begin
    var result : bits(32);

    let op_s : bits(32) = op :: Zeros{16};
    let (fptype,sign,value) = FPUnpack{32}(op_s, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN{32}(fptype, op_s, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero{32}(sign);
    elsif fptype == FPType_Infinity then
        result = FPInfinity{32}(sign);
    else
        let rounding : FPRounding = FPRoundingMode(fpcr);
        let fpexc : boolean = TRUE;
        let clamped_scale : integer = BFClampScale(op, scale);
        result = FPRoundBF{32}(value * (2.0^clamped_scale), fpcr, rounding, fpexc);
        if fpexc then FPProcessDenorm(fptype, 32, fpcr); end;
    end;

    return result[31:16];
end;
```

J1.4.3.116 BFSub

```
// BFSub()
// =======
// Non-widening BFloat16 subtraction used by SVE2 instructions.

func BFSub(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = TRUE;
    return BFSub(op1, op2, fpcr, fpexc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BFSub()
// =======
// Non-widening BFloat16 subtraction following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 - op2.
// The 'fpcr' argument supplies the FPCR control bits.

func BFSub(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type, fpexc : boolean) => bits(16)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(32);

    let op1_s : bits(32) = op1 :: Zeros{16};
    let op2_s : bits(32) = op2 :: Zeros{16};
    let (type1,sign1,value1) = FPUnpack{32}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{32}(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs{32}(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        let inf1  = (type1 == FPType_Infinity);
        let inf2  = (type2 == FPType_Infinity);
        let zero1 = (type1 == FPType_Zero);
        let zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity{32}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity{32}('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero{32}(sign1);
        else
            let result_value : real = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRoundBF{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if fpexc then FPProcessDenorms(type1, type2, 32, fpcr); end;
    end;

    return result[31:16];
end;
```

J1.4.3.117 BFSub_ZA

```
// BFSub_ZA()
// ==========
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions.

func BFSub_ZA(op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = FALSE;
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.118 BFUnpack

```
// BFUnpack()
// ==========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

func BFUnpack{N}(fpval : bits(N)) => (FPType, bit, real)
begin
    assert N IN {16,32};

    var sign : bit;
    var exp : bits(8);
    var frac : bits(23);
    if N == 16 then
        sign   = fpval[15];
        exp    = fpval[14:7];
        frac   = fpval[6:0] :: Zeros{16};
    else  // N == 32
        sign   = fpval[31];
        exp    = fpval[30:23];
        frac   = fpval[22:0];
    end;

    var fptype : FPType;
    var value : real;
    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else    // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
        end;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
    end;

    if sign == '1' then value = -value; end;

    return (fptype, sign, value);
end;
```

J1.4.3.119 BFZero

```
// BFZero()
// ========

func BFZero{N}(sign : bit) => bits(N)
begin
    assert N == 16;
    let E : integer{} = 8;
    let F : integer{} = N - (E + 1);
    return sign :: Zeros{E} :: Zeros{F};
end;
```

J1.4.3.120 FPAdd_BF16

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPAdd_BF16()
// ============
// Single-precision add following BFloat16 computation behaviors.

func FPAdd_BF16(op1 : bits(32), op2 : bits(32), fpcr : FPCR_Type) => bits(32)
begin
    var result : bits(32);

    let (type1,sign1,value1) = BFUnpack{32}(op1);
    let (type2,sign2,value2) = BFUnpack{32}(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN{32}(fpcr);
    else
        let inf1  = (type1 == FPType_Infinity);
        let inf2  = (type2 == FPType_Infinity);
        let zero1 = (type1 == FPType_Zero);
        let zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN{32}(fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity{32}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity{32}('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero{32}(sign1);
        else
            let result_value : real = value1 + value2;
            if result_value == 0.0 then
                result = FPZero{32}('0');    // Positive sign when Round to Odd
            else
                result = BFRound(result_value);
            end;
        end;
    end;

    return result;
end;
```

J1.4.3.121 FPConvertBF

```
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value using the
// Round to Nearest Even rounding mode when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

func FPConvertBF(op : bits(32), fpcr_in : FPCR_Type, rounding_in : FPRounding) => bits(16)
begin
    let halfsize : integer{}  = 16;
    var fpcr : FPCR_Type      = fpcr_in;
    var rounding : FPRounding = rounding_in;
    var result : bits(32);                              // BF16 value in top 16 bits
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;                      // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end;           // Flush denormal input and output to zero
    if altfp then rounding = FPRounding_TIEEVEN; end;  // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    let (fptype,sign,value) = FPUnpack{32}(op, fpcr, fpexc);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN{32}(fpcr);
        else
            result = FPConvertNaN{32, 32}(op);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
        if fptype == FPType_SNaN then
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        end;
    elsif fptype == FPType_Infinity then
        result = FPInfinity{32}(sign);
    elsif fptype == FPType_Zero then
        result = FPZero{32}(sign);
    else
        result = FPRoundBF{32}(value, fpcr, rounding, fpexc);
    end;

    // Returns correctly rounded BF16 value from top 16 bits
    return result[(2*halfsize)-1:halfsize];
end;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

func FPConvertBF(op : bits(32), fpcr : FPCR_Type) => bits(16)
begin
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
end;
```

J1.4.3.122 FPRoundBF

```
// FPRoundBF()
// ===========
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

func FPRoundBF{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding, fpexc : boolean) => bits(N)
begin
    assert N == 32;
    let isbfloat16 : boolean = TRUE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;
```

J1.4.3.123 FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

func FixedToFP{N, M}(op : bits(M), fbits : integer, unsigned : boolean, fpcr : FPCR_Type,
                     rounding : FPRounding) => bits(N)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    var result : bits(N);
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    let int_operand : integer{} = if unsigned then UInt(op) else SInt(op);

    // Scale by fractional bits and generate a real value
    let real_operand : real = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero{N}('0');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        result = FPRound{N}(real_operand, fpcr, rounding);
    end;

    return result;
end;
```

J1.4.3.124 BFConvertFP8

```
// BFConvertFP8()
// ==============
// Converts a BFloat16 OP to FP8 value.

func BFConvertFP8(op_in : bits(16), fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(8)
begin
    let op : bits(32) = op_in :: Zeros{16};
    return FPConvertFP8{8, 32}(op, fpcr, fpmr);
end;
```

J1.4.3.125 FP8Bits

```
// FP8Bits()
// =========
// Returns the minimum exponent, numbers of exponent and fraction bits.

func FP8Bits(fp8type : FP8Type) => FPBitsType
begin
    var minimum_exp : integer;
    var F : integer{2..3};
    if fp8type == FP8Type_OFP8_E4M3 then
        minimum_exp = -6;  F = 3;
    else  // fp8type == FP8Type_OFP8_E5M2
        minimum_exp = -14;  F = 2;
    end;

    return (F, minimum_exp);
end;
```

J1.4.3.126 FP8ConvertBF

```
// FP8ConvertBF()
// ==============
// Converts an FP8 operand to BFloat16 value.

func FP8ConvertBF(op : bits(8), issrc2 : boolean, fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(16)
begin
    let isbfloat16 : boolean = TRUE;
    let result : bits(32) = FP8ConvertFP{}(op, issrc2, fpcr, fpmr, isbfloat16);
    return result[16+:16];
end;
```

J1.4.3.127 FP8ConvertFP

```
// FP8ConvertFP()
// ==============
// Converts an FP8 operand to half-precision value.

func FP8ConvertFP(op : bits(8), issrc2 : boolean, fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(16)
begin
    let isbfloat16 : boolean = FALSE;
    return FP8ConvertFP{16}(op, issrc2, fpcr, fpmr, isbfloat16);
end;

// FP8ConvertFP()
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ==============
// Converts an FP8 operand to half-precision or BFloat16 value.
// The downscaling factor in FPMR.LSCALE or FPMR.LSCALE2 is applied to
// the value before rounding.

func FP8ConvertFP{M}(op : bits(8), issrc2 : boolean, fpcr_in : FPCR_Type, fpmr : FPMR_Type,
                     isbfloat16 : boolean) => bits(M)
begin
    assert M IN {16,32};
    var result : bits(M);

    let fpexc : boolean = TRUE;
    var fpcr : FPCR_Type = fpcr_in;
    // Do not flush denormal inputs and outputs to zero.
    // Do not support alternative half-precision format.
    fpcr.[FIZ,FZ,FZ16,AHP] = '0000';
    let rounding = FPRounding_TIEEVEN;
    let fp8type : FP8Type = (if issrc2 then FP8DecodeType(fpmr.F8S2)
                             else FP8DecodeType(fpmr.F8S1));

    let (fptype,sign,value) = FP8Unpack{8}(op, fp8type);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPDefaultNaN{M}(fpcr);
        if fptype == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    elsif fptype == FPType_Infinity then
        result = FPInfinity{M}(sign);
    elsif fptype == FPType_Zero then
        result = FPZero{M}(sign);
    else
        var dscale : integer;
        if issrc2 then
            dscale = (if M == 16 then UInt(fpmr.LSCALE2[3:0])
                      else UInt(fpmr.LSCALE2[:6]));
        else
            dscale = (if M == 16 then UInt(fpmr.LSCALE[3:0])
                       else UInt(fpmr.LSCALE[:6]));
        end;
        let result_value : real = value * (2.0^-dscale);
        result = FPRoundBase{M}(result_value, fpcr, rounding, isbfloat16, fpexc);
    end;

    return result;
end;
```

J1.4.3.128 FP8DecodeType

```
// FP8DecodeType()
// ===============
// Decode the FP8 format encoded in F8S1, F8S2 or F8D field in FPMR

func FP8DecodeType(f8format : bits(3)) => FP8Type
begin
    case f8format of
        when '000' => return FP8Type_OFP8_E5M2;
        when '001' => return FP8Type_OFP8_E4M3;
        otherwise =>  return FP8Type_UNSUPPORTED;
    end;
end;
```

J1.4.3.129 FP8DefaultNaN

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FP8DefaultNaN()
// ===============

func FP8DefaultNaN{N}(fp8type : FP8Type, fpcr : FPCR_Type) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    let sign : bit = if IsFeatureImplemented(FEAT_AFP) then fpcr.AH else '0';
    let E : integer{} = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);
    var exp : bits(E);
    var frac : bits(F);

    case fp8type of
        when FP8Type_OFP8_E4M3 =>
            exp  = Ones{E};
            frac = Ones{F};
        when FP8Type_OFP8_E5M2 =>
            exp  = Ones{E};
            frac = '1'::Zeros{F-1};
    end;

    return sign :: exp :: frac;
end;
```

J1.4.3.130 FP8DotAddFP

```
// FP8DotAddFP()
// =============

func FP8DotAddFP{M, N}(addend : bits(M), op1 : bits(N), op2 : bits(N),
                       fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(M)
begin
    let E : integer{} = (N DIV 8) as integer{1, 2, 4, 8};
    return FP8DotAddFP{M, N}(addend, op1, op2, E, fpcr, fpmr);
end;

// FP8DotAddFP()
// ==============
// Calculates result of "E"-way 8-bit floating-point dot-product with scaling
// and addition to half-precision or single-precision value without
// intermediate rounding.
// c = round(c + 2^-S*(a1*b1+..+aE*bE))
// The 8-bit floating-point format for op1 is determined by FPMR.F8S1
// and the one for op2 by FPMR.F8S2. The scaling factor in FPMR.LSCALE
// is applied to the sum-of-products before adding to the addend and rounding.

func FP8DotAddFP{M, N}(addend : bits(M), op1 : bits(N), op2 : bits(N), E : integer{1, 2, 4, 8},
                       fpcr_in : FPCR_Type, fpmr : FPMR_Type) => bits(M)
begin
    assert M IN {16,32};
    assert N IN {2*M, M, M DIV 2, M DIV 4};
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(M);

    fpcr.[FIZ,FZ,FZ16] = '000';        // Do not flush denormal inputs and outputs to zero
    fpcr.DN = '1';
    let rounding = FPRounding_TIEEVEN;

    let fp8type1 : FP8Type = FP8DecodeType(fpmr.F8S1);
    let fp8type2 : FP8Type = FP8DecodeType(fpmr.F8S2);

    var type1 : array[[E]] of FPType;
    var type2 : array[[E]] of FPType;
    var sign1 : array[[E]] of bit;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
var sign2 : array[[E]] of bit;
var value1 : array[[E]] of real;
var value2 : array[[E]] of real;
var inf1 : array[[E]] of boolean;
var inf2 : array[[E]] of boolean;
var zero1 : array[[E]] of boolean;
var zero2 : array[[E]] of boolean;

let fpexc : boolean = FALSE;
let (typeA,signA,valueA) = FPUnpack{M}(addend, fpcr, fpexc);
let infA = (typeA == FPType_Infinity);   let zeroA = (typeA == FPType_Zero);
var any_nan : boolean = typeA IN {FPType_SNaN, FPType_QNaN};
for i = 0 to E-1 do
    (type1[[i]], sign1[[i]], value1[[i]]) = FP8Unpack{N DIV E}(op1[i*:(N DIV E)], fp8type1);
    (type2[[i]], sign2[[i]], value2[[i]]) = FP8Unpack{N DIV E}(op2[i*:(N DIV E)], fp8type2);
    inf1[[i]] = (type1[[i]] == FPType_Infinity); zero1[[i]] = (type1[[i]] == FPType_Zero);
    inf2[[i]] = (type2[[i]] == FPType_Infinity); zero2[[i]] = (type2[[i]] == FPType_Zero);
    any_nan = (any_nan || type1[[i]] IN {FPType_SNaN, FPType_QNaN} ||
               type2[[i]] IN {FPType_SNaN, FPType_QNaN});
end;

if any_nan then
    result = FPDefaultNaN{M}(fpcr);
else
    // Determine sign and type products will have if it does not cause an Invalid
    // Operation.
    var signP : array [[E]] of bit;
    var infP : array [[E]] of boolean;
    var zeroP : array [[E]] of boolean;
    for i = 0 to E-1 do
        signP[[i]] = sign1[[i]] XOR sign2[[i]];
        infP[[i]]  = inf1[[i]] || inf2[[i]];
        zeroP[[i]] = zero1[[i]] || zero2[[i]];
    end;

    // Detect non-numeric results of dot product and accumulate
    var posInfR : boolean = (infA && signA == '0');
    var negInfR : boolean = (infA && signA == '1');
    var zeroR : boolean = zeroA;
    var invalidop : boolean = FALSE;
    for i = 0 to E-1 do
        // Result is infinity if any input is infinity
        posInfR  = posInfR || (infP[[i]] && signP[[i]] == '0');
        negInfR  = negInfR || (infP[[i]] && signP[[i]] == '1');
        // Result is zero if the addend and the products are zeroes of the same sign
        zeroR    = zeroR && zeroP[[i]] && (signA == signP[[i]]);
        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (invalidop || (inf1[[i]] && zero2[[i]]) || (zero1[[i]] && inf2[[i]]) ||
                     (infA && infP[[i]] && (signA != signP[[i]])));
        for j = i+1 to E-1 do
            invalidop = invalidop || (infP[[i]] && infP[[j]] && (signP[[i]] != signP[[j]]));
        end;
    end;

    if invalidop then
        result = FPDefaultNaN{M}(fpcr);

    // Other cases involving infinities produce an infinity of the same sign.
    elsif posInfR then
        result = FPInfinity{M}('0');
    elsif negInfR then
        result = FPInfinity{M}('1');

    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif zeroR then
            result = FPZero{M}(signA);

        // Otherwise calculate numerical value and round it.
        else
            // Apply scaling to sum-of-product
            let dscale : integer = if M == 32 then UInt(fpmr.LSCALE) else UInt(fpmr.LSCALE[3:0]);

            var dp_value : real = value1[[0]] * value2[[0]];
            for i = 1 to E-1 do
                dp_value = dp_value + value1[[i]] * value2[[i]];
            end;

            let result_value : real = valueA + dp_value * (2.0^-dscale);
            if result_value == 0.0 then  // Sign of exact zero result is '0' for RNE rounding mode
                result = FPZero{M}('0');
            else
                let satoflo : boolean = (fpmr.OSM == '1');
                result = FPRound_FP8{M}(result_value, fpcr, rounding, satoflo);
            end;
        end;
    end;

    return result;
end;
```

J1.4.3.131 FP8Infinity

```
// FP8Infinity()
// =============

func FP8Infinity{N}(fp8type : FP8Type, sign : bit) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    let E : integer{} = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);
    var exp : bits(E);
    var frac : bits(F);

    case fp8type of
        when FP8Type_OFP8_E4M3 =>
            exp  = Ones{E};
            frac = Ones{F};
        when FP8Type_OFP8_E5M2 =>
            exp  = Ones{E};
            frac = Zeros{F};
    end;

    return sign :: exp :: frac;
end;
```

J1.4.3.132 FP8MatMulAddFP

```
// FP8MatMulAddFP()
// ================
// 8-bit floating-point matrix multiply with scaling and add to half-precision
// or single-precision matrix.
// result[2, 2] = addend[2, 2] + (op1[2, E] * op2[E, 2])

func FP8MatMulAddFP{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), E : integer{4, 8},
                       fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(N)
begin
    assert N IN {64, 128};
    assert N == E*16;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let M : integer{} = N DIV 4;
    var result : bits(N);

    for i = 0 to 1 do
        for j = 0 to 1 do
            let elt1 : bits(2*M) = op1[i*:(2*M)];
            let elt2 : bits(2*M) = op2[j*:(2*M)];
            let sum : bits(M) = addend[(2*i + j)*:M];
            result[(2*i + j)*:M] = FP8DotAddFP{M, N DIV 2}(sum, elt1, elt2, E, fpcr, fpmr);
        end;
    end;

    return result;
end;
```

J1.4.3.133 FP8MaxNormal

```
// FP8MaxNormal()
// =============

func FP8MaxNormal{N}(fp8type : FP8Type, sign : bit) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    let E : integer{} = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);
    var exp : bits(E);
    var frac : bits(F);

    case fp8type of
        when FP8Type_OFP8_E4M3 =>
            exp  = Ones{E};
            frac = Ones{F-1}::'0';
        when FP8Type_OFP8_E5M2 =>
            exp  = Ones{E-1}::'0';
            frac = Ones{F};
    end;

    return sign :: exp :: frac;
end;
```

J1.4.3.134 FP8MulAddFP

```
// FP8MulAddFP()
// =============

func FP8MulAddFP{M}(addend : bits(M), op1 : bits(8), op2 : bits(8), fpcr : FPCR_Type,
                    fpmr : FPMR_Type) => bits(M)
begin
    assert M IN {16,32};
    let E : integer{} = 1;
    return FP8DotAddFP{M, 8}(addend, op1, op2, E, fpcr, fpmr);
end;
```

J1.4.3.135 FP8Round

```
// FP8Round()
// ==========
// Used by FP8 downconvert instructions which observe FPMR.OSC
// to convert a real number OP into an FP8 value.

func FP8Round{N}(op : real, fp8type : FP8Type, fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(N)
begin
    assert N == 8;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
assert op != 0.0;
var result : bits(N);

// Format parameters - minimum exponent, numbers of exponent and fraction bits.
let (F, minimum_exp) = FP8Bits(fp8type);
let E : integer{} = (N - F) - 1;

// Split value into sign, unrounded mantissa and exponent.
var sign : bit;
var exponent : integer;
var mantissa : real;
if op < 0.0 then
    sign = '1';  mantissa = -op;
else
    sign = '0';  mantissa = op;
end;

(mantissa, exponent) = NormalizeReal(mantissa);
// When TRUE, detection of underflow occurs after rounding.
let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';

var biased_exp_unconstrained : integer = (exponent - minimum_exp) + 1;
var int_mant_unconstrained : integer= RoundDown(mantissa * 2.0^F);
let error_unconstrained : real = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
var biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
// < 2.0^F if biased_exp == 0, >= 2.0^F if not
var int_mant : integer = RoundDown(mantissa * 2.0^F);
var error : real = mantissa * 2.0^F - Real(int_mant);

let trapped_UF : boolean = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());

var round_up_unconstrained : boolean;
var round_up : boolean;

if altfp then
    // Round to Nearest Even
    round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained[0] == '1'));
    round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained   = int_mant_unconstrained DIV 2;
        end;
    end;

    // Follow alternate floating-point behavior of underflow after rounding
    if (biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 &&
        (error != 0.0 || trapped_UF)) then
        FPProcessException(FPExc_Underflow, fpcr);
    end;
else // altfp == FALSE
    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
    if biased_exp == 0 && (error != 0.0 || trapped_UF) then
        FPProcessException(FPExc_Underflow, fpcr);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Round to Nearest Even
        round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
    end;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        end;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;
        end;
    end;

    // Deal with overflow and generate result.
    var overflow : boolean;
    case fp8type of
        when FP8Type_OFP8_E4M3 =>
            overflow = biased_exp >= 2^E || (biased_exp == 2^E - 1 && int_mant == 2^(F+1) - 1);
        when FP8Type_OFP8_E5M2 =>
            overflow = biased_exp >= 2^E - 1;
    end;

    if overflow then
        result = (if fpmr.OSC == '0' then FP8Infinity{N}(fp8type, sign)
                  else FP8MaxNormal{N}(fp8type, sign));
        // Flag Overflow exception regardless of FPMR.OSC
        FPProcessException(FPExc_Overflow, fpcr);
        error = 1.0;  // Ensure that an Inexact exception occurs
    else
        result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0];
    end;

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
    end;

    return result;
end;
```

J1.4.3.136 FP8Type

```
// FP8Type
// =======

type FP8Type of enumeration {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3, FP8Type_UNSUPPORTED};
```

J1.4.3.137 FP8Unpack

```
// FP8Unpack()
// ===========
// Unpacks an FP8 value into its type, sign bit and real number that
// it represents.

func FP8Unpack{N}(fpval : bits(N), fp8type : FP8Type) => (FPType, bit, real)
begin
    assert N == 8;
    let E : integer{} = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);

    let sign : bit = fpval[N-1];
    let exp : bits(E) = fpval[(E+F)-1:F];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let frac : bits(F) = fpval[F-1:0];

    var value : real;
    var fptype : FPType;

    if fp8type == FP8Type_OFP8_E4M3 then
        if IsZero(exp) then
            if IsZero(frac) then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-6 * (Real(UInt(frac)) * 2.0^-3);
            end;
        elsif IsOnes(exp) && IsOnes(frac) then
            fptype = FPType_SNaN;
            value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp)-7) * (1.0 + Real(UInt(frac)) * 2.0^-3);
        end;

    elsif fp8type == FP8Type_OFP8_E5M2 then
        if IsZero(exp) then
            if IsZero(frac) then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac)) * 2.0^-2);
            end;
        elsif IsOnes(exp) then
            if IsZero(frac) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac[1] == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
            end;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp)-15) * (1.0 + Real(UInt(frac)) * 2.0^-2);
        end;

    else // fp8type == FP8Type_UNSUPPORTED
        fptype = FPType_SNaN;
        value = 0.0;
    end;

    if sign == '1' then value = -value; end;

    return (fptype, sign, value);
end;
```

J1.4.3.138 FP8Zero

```
// FP8Zero()
// =========

func FP8Zero{N}(fp8type : FP8Type, sign : bit) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    let E : integer{} = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);
    return sign :: Zeros{E} :: Zeros{F};
end;
```

J1.4.3.139 FPConvertFP8

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPConvertFP8()
// ==============
// Converts a half-precision or single-precision OP to FP8 value.
// The scaling factor in FPMR.NSCALE is applied to the value before rounding.

func FPConvertFP8{M, N}(op : bits(N), fpcr_in : FPCR_Type, fpmr : FPMR_Type) => bits(M)
begin
    assert N IN {16,32} && M == 8;
    var result : bits(M);

    let fpexc : boolean = TRUE;
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.[FIZ,FZ,FZ16] = '000';    // Do not flush denormal inputs and outputs to zero
    let fp8type : FP8Type = FP8DecodeType(fpmr.F8D);

    let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpexc);

    if fp8type == FP8Type_UNSUPPORTED then
        result = Ones{M};
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FP8DefaultNaN{M}(fp8type, fpcr);     // Always generate Default NaN as result
        if fptype == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    elsif fptype == FPType_Infinity then
        result = (if fpmr.OSC == '0' then FP8Infinity{M}(fp8type, sign)
                  else FP8MaxNormal{M}(fp8type, sign));
    elsif fptype == FPType_Zero then
        result = FP8Zero{M}(fp8type, sign);
    else
        let scale : integer = if N == 16 then SInt(fpmr.NSCALE[4:0]) else SInt(fpmr.NSCALE);
        let result_value : real = value * (2.0^scale);
        result = FP8Round{M}(result_value, fp8type, fpcr, fpmr);
    end;

    return result;
end;
```

J1.4.3.140 FPAbs

```
// FPAbs()
// =======

func FPAbs{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if fpcr.AH == '1' then
            let (fptype, -, -) = FPUnpack{N}(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;       // When fpcr.AH=1, sign of NaN has no consequence
            end;
        end;
    end;
    return '0' :: op[N-2:0];
end;
```

J1.4.3.141 FPAbsMax

```
// FPAbsMax()
// ==========
// Compare absolute value of two operands and return the larger absolute
// value without rounding.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func FPAbsMax{N}(op1_in : bits(N), op2_in : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type  = fpcr_in;
    fpcr.[AH,FIZ,FZ,FZ16] = '0000';

    let op1 = '0'::op1_in[N-2:0];
    let op2 = '0'::op2_in[N-2:0];
    let (type1,-,value1) = FPUnpack{N}(op1, fpcr);
    let (type2,-,value2) = FPUnpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1_in, op2_in, fpcr);

    if !done then
        // This condition covers all results other than NaNs,
        // including Zero & Infinity
        result = if value1 > value2 then op1 else op2;
    end;

    return result;
end;
```

J1.4.3.142 FPAbsMin

```
// FPAbsMin()
// ==========
// Compare absolute value of two operands and return the smaller absolute
// value without rounding.

func FPAbsMin{N}(op1_in : bits(N), op2_in : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.[AH,FIZ,FZ,FZ16] = '0000';

    let op1 = '0'::op1_in[N-2:0];
    let op2 = '0'::op2_in[N-2:0];
    let (type1,-,value1) = FPUnpack{N}(op1, fpcr);
    let (type2,-,value2) = FPUnpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1_in, op2_in, fpcr);

    if !done then
        // This condition covers all results other than NaNs,
        // including Zero & Infinity
        result = if value1 < value2 then op1 else op2;
    end;

    return result;
end;
```

J1.4.3.143 FPAdd

```
// FPAdd()
// =======

func FPAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;       // Generate floating-point exceptions
    return FPAdd{N}(op1, op2, fpcr, fpexc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

// FPAdd()
// =======

func FPAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin

    assert N IN {16,32,64};
    let rounding : FPRounding = FPRoundingMode(fpcr);

    let (type1,sign1,value1) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{N}(op2, fpcr, fpexc);

    var (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        let inf1 : boolean  = (type1 == FPType_Infinity);
        let inf2 : boolean  = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity{N}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity{N}('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero{N}(sign1);
        else
            let result_value : real = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
    end;
    return result;
end;
```

J1.4.3.144 FPAdd_ZA

```
// FPAdd_ZA()
// ==========
// Calculates op1+op2 for SME2 ZA-targeting instructions.

func FPAdd_ZA{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPAdd{N}(op1, op2, fpcr, fpexc);
end;
```

J1.4.3.145 FPBits

```
// FPBits()
// ========
// Returns the minimum exponent, numbers of exponent and fraction bits.

func FPBits(N : integer, isbfloat16 : boolean) => FPBitsType
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    var F : FPFracBits;
    var minimum_exp : integer;
    if N == 16 then
        minimum_exp = -14;   F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126;  F = 7;
    elsif N == 32 then
        minimum_exp = -126;  F = 23;
    else  // N == 64
        minimum_exp = -1022; F = 52;
    end;

    return (F, minimum_exp);
end;
```

J1.4.3.146 FPBitsType

```
// FPBitsType
// ==========

type FPBitsType of (FPFracBits, integer);
```

J1.4.3.147 FPFracBits

```
// FPFracBits
// ==========

type FPFracBits of integer{2, 3, 7, 10, 23, 52};
```

J1.4.3.148 FPCompare

```
// FPCompare()
// ===========

func FPCompare{N}(op1 : bits(N), op2 : bits(N), signal_nans : boolean, fpcr : FPCR_Type) => bits(4)
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    var result : bits(4);
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = '0011';
        if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';
        end;

        FPProcessDenorms(type1, type2, N, fpcr);
    end;
    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.149 FPCompareEQ

```
// FPCompareEQ()
// =============

func FPCompareEQ{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    var result : boolean;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    end;
    return result;
end;
```

J1.4.3.150 FPCompareGE

```
// FPCompareGE()
// =============

func FPCompareGE{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    var result : boolean;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    end;
    return result;
end;
```

J1.4.3.151 FPCompareGT

```
// FPCompareGT()
// =============

func FPCompareGT{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    var result : boolean;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    end;
    return result;
end;
```

J1.4.3.152 FPConvert

```
// FPConvert()
// ===========

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

func FPConvert{M, N}(op : bits(N), fpcr : FPCR_Type, rounding : FPRounding) => bits(M)
begin

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    var result : bits(M);

    // Unpack floating-point operand optionally with flush-to-zero.
    let (fptype,sign,value) : (FPType, bit, real) = FPUnpackCV{N}(op, fpcr);

    let alt_hp : boolean = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero{M}(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN{M}(fpcr);
        else
            result = FPConvertNaN{M, N}(op);
        end;
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
        end;
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign::Ones{M-1};
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity{M}(sign);
        end;
    elsif fptype == FPType_Zero then
        result = FPZero{M}(sign);
    else
        result = FPRoundCV{M}(value, fpcr, rounding);
        FPProcessDenorm(fptype, N, fpcr);

    end;
    return result;
end;

// FPConvert()
// ===========

func FPConvert{M, N}(op : bits(N), fpcr : FPCR_Type) => bits(M)
begin
    return FPConvert{M, N}(op, fpcr, FPRoundingMode(fpcr));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.153 FPConvertNaN

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

func FPConvertNaN{M, N}(op : bits(N)) => bits(M)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    var result : bits(M);
    var frac : bits(51);

    let sign : bit = op[N-1];

    // Unpack payload from input NaN
    case N of
        when 64 => frac = op[50:0];
        when 32 => frac = op[21:0]::Zeros{29};
        when 16 => frac = op[8:0]::Zeros{42};
    end;

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 => result = sign::Ones{M-52}::frac;
        when 32 => result = sign::Ones{M-23}::frac[50:29];
        when 16 => result = sign::Ones{M-10}::frac[50:42];
    end;

    return result;
end;
```

J1.4.3.154 FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

func FPDecodeRM(rm : bits(2)) => FPRounding
begin
    var result : FPRounding;
    case rm of
        when '00' => result = FPRounding_TIEAWAY; // A
        when '01' => result = FPRounding_TIEEVEN; // N
        when '10' => result = FPRounding_POSINF;  // P
        when '11' => result = FPRounding_NEGINF;  // M
    end;

    return result;
end;
```

J1.4.3.155 FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

func FPDecodeRounding(rmode : bits(2)) => FPRounding
begin
    case rmode of
        when '00' => return FPRounding_TIEEVEN; // N
        when '01' => return FPRounding_POSINF;  // P
        when '10' => return FPRounding_NEGINF;  // M
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '11' => return FPRounding_ZERO;    // Z
    end;
end;
```

J1.4.3.156 FPDefaultNaN

```
// FPDefaultNaN()
// ==============

func FPDefaultNaN{N}(fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let sign : bit = if IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() then fpcr.AH else '0';

    let exp : bits(E)  = Ones{};
    let frac : bits(F) = '1'::Zeros{F-1};
    return sign :: exp :: frac;
end;
```

J1.4.3.157 FPDiv

```
// FPDiv()
// =======

func FPDiv{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);

    if !done then
        let inf1 : boolean  = type1 == FPType_Infinity;
        let inf2 : boolean  = type2 == FPType_Infinity;
        let zero1 : boolean = type1 == FPType_Zero;
        let zero2 : boolean = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN{N}(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity{N}(sign1 XOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr); end;
        elsif zero1 || inf2 then
            result = FPZero{N}(sign1 XOR sign2);
        else
            result = FPRound{N}(value1/value2, fpcr);
        end;

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);
        end;
    end;
    return result;
end;
```

J1.4.3.158 FPDot

```
// FPDot()
// =======
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

func FPDot(op1_a : bits(16), op1_b : bits(16), op2_a : bits(16),
           op2_b : bits(16), fpcr : FPCR_Type, isbfloat16 : boolean) => bits(32)
begin
    let fpexc : boolean = TRUE;       // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
end;

func FPDot(op1_a : bits(16), op1_b : bits(16), op2_a : bits(16),
           op2_b : bits(16), fpcr_in : FPCR_Type, isbfloat16 : boolean, fpexc : boolean) => bits(32)
begin
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(32);
    var done : boolean;
    fpcr.AHP = '0';           // Ignore alternative half-precision option
    let rounding : FPRounding = FPRoundingMode(fpcr);

    let (type1_a,sign1_a,value1_a) = FPUnpackBase{16}(op1_a, fpcr, fpexc, isbfloat16);
    let (type1_b,sign1_b,value1_b) = FPUnpackBase{16}(op1_b, fpcr, fpexc, isbfloat16);
    let (type2_a,sign2_a,value2_a) = FPUnpackBase{16}(op2_a, fpcr, fpexc, isbfloat16);
    let (type2_b,sign2_b,value2_b) = FPUnpackBase{16}(op2_b, fpcr, fpexc, isbfloat16);

    let inf1_a = (type1_a == FPType_Infinity); let zero1_a = (type1_a == FPType_Zero);
    let inf1_b = (type1_b == FPType_Infinity); let zero1_b = (type1_b == FPType_Zero);
    let inf2_a = (type2_a == FPType_Infinity); let zero2_a = (type2_a == FPType_Zero);
    let inf2_b = (type2_b == FPType_Infinity); let zero2_b = (type2_b == FPType_Zero);

    (done,result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                   op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

    if !done then
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        let signPa = sign1_a XOR sign2_a;
        let signPb = sign1_b XOR sign2_b;
        let infPa  = inf1_a || inf2_a;
        let infPb  = inf1_b || inf2_b;
        let zeroPa = zero1_a || zero2_a;
        let zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        let invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
                         (inf1_b && zero2_b) || (zero1_b && inf2_b) ||
                         (infPa && infPb && signPa != signPb));

        if invalidop then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;

       // Other cases involving infinities produce an infinity of the same sign.
        elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity{32}('0');
        elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity{32}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroPa && zeroPb && signPa == signPb then
            result = FPZero{32}(signPa);

        // Otherwise calculate fused sum of products and round it.
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            let result_value = (value1_a * value2_a) + (value1_b * value2_b);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRound{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;
    end;

    return result;
end;
```

J1.4.3.159 FPDotAdd

```
// FPDotAdd()
// ==========
// Half-precision 2-way dot-product and add to single-precision.

func FPDotAdd(addend : bits(32), op1_a : bits(16), op1_b : bits(16),
              op2_a : bits(16), op2_b : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    var prod : bits(32);
    let isbfloat16 : boolean = FALSE;
    let fpexc : boolean      = TRUE;     // Generate floating-point exceptions
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    let result : bits(32) = FPAdd{}(addend, prod, fpcr, fpexc);

    return result;
end;
```

J1.4.3.160 FPDotAdd_ZA

```
// FPDotAdd_ZA()
// =============
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

func FPDotAdd_ZA(addend : bits(32), op1_a : bits(16), op1_b : bits(16),
                 op2_a : bits(16), op2_b : bits(16), fpcr_in : FPCR_Type) => bits(32)
begin
    var fpcr : FPCR_Type = fpcr_in;
    var prod : bits(32);
    let isbfloat16 : boolean = FALSE;
    let fpexc : boolean      = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                    // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    let result : bits(32) = FPAdd{}(addend, prod, fpcr, fpexc);

    return result;
end;
```

J1.4.3.161 FPExc

```
// FPExc
// =====

type FPExc of enumeration {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                           FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.162 FPInfinity

```
// FPInfinity()
// ============

func FPInfinity{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = Ones{};
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;
```

J1.4.3.163 FPMatMulAdd

```
// FPMatMulAdd()
// =============
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

func FPMatMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N),
                    esize : ESize, fpcr : FPCR_Type) => bits(N)
begin
    assert N == esize * 2 * 2;
    var result : bits(N);
    var prod0, prod1, sum : bits(esize);

    for i = 0 to 1 do
        for j = 0 to 1 do
            sum   = addend[(2*i + j)*:esize];
            prod0 = FPMul{esize}(op1[(2*i + 0)*:esize],
                         op2[(2*j + 0)*:esize], fpcr);
            prod1 = FPMul{esize}(op1[(2*i + 1)*:esize],
                         op2[(2*j + 1)*:esize], fpcr);
            sum   = FPAdd{esize}(sum, FPAdd{esize}(prod0, prod1, fpcr), fpcr);
            result[(2*i + j)*:esize] = sum;
        end;
    end;

    return result;
end;
```

J1.4.3.164 FPMatMulAddH

```
// FPMatMulAddH()
// ==============
// Half-precision matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

func FPMatMulAddH{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N == 128;
    let M : integer{} = 32;
    var result : bits(N);

    let isbfloat16 : boolean = FALSE;
    for i = 0 to 1 do
        for j = 0 to 1 do
            var sum : bits(M) = addend[(2*i + j)*:M];
            var prod : array[[2]] of bits(M);
            for k = 0 to 1 do
                let elt1_a : bits(M DIV 2) = op1[(4*i + 2*k + 0)*:(M DIV 2)];
                let elt1_b : bits(M DIV 2) = op1[(4*i + 2*k + 1)*:(M DIV 2)];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                let elt2_a : bits(M DIV 2) = op2[(4*j + 2*k + 0)*:(M DIV 2)];
                let elt2_b : bits(M DIV 2) = op2[(4*j + 2*k + 1)*:(M DIV 2)];
                prod[[k]] = FPDot(elt1_a, elt1_b, elt2_a, elt2_b, fpcr, isbfloat16);
            end;
            sum = FPAdd{M}(sum, FPAdd{M}(prod[[0]], prod[[1]], fpcr), fpcr);
            result[(2*i + j)*:M] = sum;
        end;
    end;

    return result;
end;
```

J1.4.3.165 FPMax

```
// FPMax()
// =======

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return FPMax{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMax()
// =======

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMax{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMax()
// =======
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
              altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero{N}(sign2);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FPType_Zero then FPZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        var fptype : FPType;
        var sign : bit;
        var value : real;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FPType_Infinity then
            result = FPInfinity{N}(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;         // Use most positive sign
            result = FPZero{N}(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            let rounding : FPRounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            end;

            result = FPRound{N}(value, fpcr, rounding, fpexc);
        end;
        if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
    end;

    return result;
end;
```

J1.4.3.166 FPMaxNormal

```
// FPMaxNormal()
// =============

func FPMaxNormal{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = Ones{E-1}::'0';
    let frac : bits(F) = Ones{};
    return sign :: exp :: frac;
end;
```

J1.4.3.167 FPMaxNum

```
// FPMaxNum()
// ==========

func FPMaxNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMaxNum{N}(op1, op2, fpcr, fpexc);
end;

// FPMaxNum()
// ==========

func FPMaxNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let (type1,-,-) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,-,-) = FPUnpack{N}(op2, fpcr, fpexc);

    let type1_nan : boolean = type1 IN {FPType_QNaN, FPType_SNaN};
    let type2_nan : boolean = type2 IN {FPType_QNaN, FPType_SNaN};
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity{N}('1');
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity{N}('1');
        end;
    end;

    let altfmaxfmin : boolean = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    let result : bits(N) = FPMax{}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

J1.4.3.168 IsMerging

```
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

func IsMerging(fpcr : FPCR_Type) => boolean
begin
    let nep : bit = (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
                        !IsFullA64Enabled() then '0' else fpcr.NEP);
    return IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && nep == '1';
end;
```

J1.4.3.169 FPMin

```
// FPMin()
// =======

func FPMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return FPMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMin()
// =======

func FPMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMin()
// =======
// Compare two inputs and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func FPMin{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
              altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero{N}(sign2);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FPType_Zero then FPZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        var fptype : FPType;
        var sign : bit;
        var value : real;
        var rounding : FPRounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FPType_Infinity then
            result = FPInfinity{N}(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero{N}(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            end;

            result = FPRound{N}(value, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
    end;
    return result;
end;
```

J1.4.3.170 FPMinNum

```
// FPMinNum()
// ==========

func FPMinNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMinNum{N}(op1, op2, fpcr, fpexc);
end;

// FPMinNum()
// ==========

func FPMinNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let (type1,-,-) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,-,-) = FPUnpack{N}(op2, fpcr, fpexc);

    let type1_nan : boolean = type1 IN {FPType_QNaN, FPType_SNaN};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let type2_nan : boolean = type2 IN {FPType_QNaN, FPType_SNaN};
    let altfp : boolean     = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
                                                                 fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity{N}('0');
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity{N}('0');
        end;
    end;

    let altfmaxfmin : boolean = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    let result : bits(N) = FPMin{}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

J1.4.3.171 FPMul

```
// FPMul()
// =======

func FPMul{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean  = (type1 == FPType_Infinity);
        let inf2 : boolean  = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{N}(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        elsif zero1 || zero2 then
            result = FPZero{N}(sign1 XOR sign2);
        else
            result = FPRound{N}(value1*value2, fpcr);
        end;

        FPProcessDenorms(type1, type2, N, fpcr);

    end;
    return result;
end;
```

J1.4.3.172 FPMulAdd

```
// FPMulAdd()
// ==========

func FPMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;       // Generate floating-point exceptions
    return FPMulAdd{N}(addend, op1, op2, fpcr, fpexc);
end;

// FPMulAdd()
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ==========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

func FPMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N),
                 fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};

    let (typeA,signA,valueA) : (FPType, bit, real) = FPUnpack{N}(addend, fpcr, fpexc);
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
    let rounding : FPRounding = FPRoundingMode(fpcr);
    let inf1 : boolean = (type1 == FPType_Infinity); let zero1 : boolean = (type1 == FPType_Zero);
    let inf2 : boolean = (type2 == FPType_Infinity); let zero2 : boolean = (type2 == FPType_Zero);

    var (done,result) = FPProcessNaNs3{N}(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        end;
    end;

    if !done then
        let infA : boolean = (typeA == FPType_Infinity);
        let zeroA : boolean = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        let signP : bit = sign1 XOR sign2;
        let infP : boolean  = inf1 || inf2;
        let zeroP : boolean = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        let invalidop : boolean = ((inf1 && zero2) || (zero1 && inf2) ||
                                   (infA && infP && signA != signP));

        if invalidop then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity{N}('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity{N}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero{N}(signA);

        // Otherwise calculate numerical result and round it.
        else
            let result_value : real = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, N, fpcr);
        end;
    end;
    return result;
end;
```

J1.4.3.173 FPMulAdd_ZA

```
// FPMulAdd_ZA()
// =============
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

func FPMulAdd_ZA{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean  = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                // Generate default NaN values
    return FPMulAdd{N}(addend, op1, op2, fpcr, fpexc);
end;
```

J1.4.3.174 FPMulAddH

```
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

func FPMulAddH(addend : bits(32), op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    let fpexc : boolean = TRUE;       // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
end;

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

func FPMulAddH(addend : bits(32), op1 : bits(16), op2 : bits(16),
               fpcr : FPCR_Type, fpexc : boolean) => bits(32)
begin

    let rounding : FPRounding = FPRoundingMode(fpcr);
    let (typeA,signA,valueA)  = FPUnpack{32}(addend, fpcr, fpexc);
    let (type1,sign1,value1)  = FPUnpack{16}(op1, fpcr, fpexc);
    let (type2,sign2,value2)  = FPUnpack{16}(op2, fpcr, fpexc);
    let inf1 = (type1 == FPType_Infinity); let zero1 = (type1 == FPType_Zero);
    let inf2 = (type2 == FPType_Infinity); let zero2 = (type2 == FPType_Zero);

    var (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        end;
    end;

    if !done then
        let infA = (typeA == FPType_Infinity); let zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        let signP = sign1 XOR sign2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let infP  = inf1 || inf2;
        let zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        let invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity{32}('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity{32}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero{32}(signA);

        // Otherwise calculate numerical result and round it.
        else
            let result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRound{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if !invalidop && fpexc then
            FPProcessDenorm(typeA, 32, fpcr);
        end;
    end;

    return result;
end;
```

J1.4.3.175 FPMulAddH_ZA

```
// FPMulAddH_ZA()
// ==============
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

func FPMulAddH_ZA(addend : bits(32), op1 : bits(16), op2 : bits(16),
                  fpcr_in : FPCR_Type) => bits(32)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';               // Generate default NaN values
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
end;
```

J1.4.3.176 FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

func FPProcessNaNs3H(type1 : FPType, type2 : FPType, type3 : FPType,
                     op1 : bits(32), op2 : bits(16), op3 : bits(16),
                     fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(32))
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var result : bits(32);
    var type_nan : FPType;
    // When TRUE, use alternative NaN propagation rules.
    let altfp : boolean   = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let op1_nan : boolean = type1 IN {FPType_SNaN, FPType_QNaN};
    let op2_nan : boolean = type2 IN {FPType_SNaN, FPType_QNaN};
    let op3_nan : boolean = type3 IN {FPType_SNaN, FPType_QNaN};
    if altfp then
        if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;
        end;
    end;

    var done : boolean;
    if altfp && op1_nan && op2_nan && op3_nan then          // [n] register NaN selected
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type_nan, op2, fpcr, fpexc));
    elsif altfp && op2_nan && (op1_nan || op3_nan) then     // [n] register NaN selected
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type_nan, op2, fpcr, fpexc));
    elsif altfp && op3_nan && op1_nan then                  // [m] register NaN selected
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type_nan, op3, fpcr, fpexc));
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN{32}(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type3, op3, fpcr, fpexc));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN{32}(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type3, op3, fpcr, fpexc));
    else
        done = FALSE; result = Zeros{32}; // 'Don't care' result
    end;
    return (done, result);
end;
```

J1.4.3.177 FPMulX

```
// FPMulX()
// ========

func FPMulX{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N);
    var done : boolean;
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FPType_Infinity);
        let inf2 : boolean = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo{N}(sign1 XOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        elsif zero1 || zero2 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = FPZero{N}(sign1 XOR sign2);
        else
            result = FPRound{N}(value1*value2, fpcr);
        end;

        FPProcessDenorms(type1, type2, N, fpcr);

    end;
    return result;
end;
```

J1.4.3.178 FPNeg

```
// FPNeg()
// =======

func FPNeg{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if fpcr.AH == '1' then
            let (fptype, -, -) = FPUnpack{N}(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When fpcr.AH=1, sign of NaN has no consequence
            end;
        end;
    end;
    return NOT(op[N-1]) :: op[N-2:0];
end;
```

J1.4.3.179 FPOnePointFive

```
// FPOnePointFive()
// ================

func FPOnePointFive{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = '0'::Ones{E-1};
    let frac : bits(F) = '1'::Zeros{F-1};
    let result : bits((F + E) + 1) = sign :: exp :: frac;

    return result;
end;
```

J1.4.3.180 FPProcessDenorm

```
// FPProcessDenorm()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPProcessDenorm(fptype : FPType, N : integer, fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.181 FPProcessDenorms

```
// FPProcessDenorms()
// ==================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPProcessDenorms(type1 : FPType, type2 : FPType, N : integer, fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
    end;
end;
```

J1.4.3.182 FPProcessDenorms3

```
// FPProcessDenorms3()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPProcessDenorms3(type1 : FPType, type2 : FPType, type3 : FPType, N : integer,
                       fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
    end;
end;
```

J1.4.3.183 FPProcessDenorms4

```
// FPProcessDenorms4()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPProcessDenorms4(type1 : FPType, type2 : FPType, type3 : FPType, type4 : FPType,
                       N : integer, fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
    end;
end;
```

J1.4.3.184 FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

func FPProcessException(except : FPExc, fpcr : FPCR_Type)
begin
    var cumul : integer;
    // Determine the cumulative exception bit number
    case except of
        when FPExc_InvalidOp =>     cumul = 0;
        when FPExc_DivideByZero =>  cumul = 1;
        when FPExc_Overflow =>      cumul = 2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when FPExc_Underflow =>     cumul = 3;
        when FPExc_Inexact =>       cumul = 4;
        when FPExc_InputDenorm =>   cumul = 7;
    end;
    let enable : integer = cumul + 8;
    if (fpcr[enable] == '1' && (!IsFeatureImplemented(FEAT_SME) || PSTATE.SM == '0' ||
          IsFullA64Enabled())) then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        var accumulated_exceptions : bits(8) = GetAccumulatedFPExceptions();
        accumulated_exceptions[cumul] = '1';
        if ImpDefBool("Support trapping of floating-point exceptions") then
            if UsingAArch32() then
                AArch32_FPTrappedException(accumulated_exceptions);
            else
                let is_ase : boolean = IsASEInstruction();
                AArch64_FPTrappedException(is_ase, accumulated_exceptions);
            end;
        else
            // The exceptions generated by this instruction are accumulated by the PE and
            // FPTrappedException is called later during its execution, before the next
            // instruction is executed. This field is cleared at the start of each FP instruction.
            SetAccumulatedFPExceptions(accumulated_exceptions);
        end;
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR()[cumul] = '1';
    else
        // Set the cumulative exception bit
        FPSR()[cumul] = '1';
    end;
    return;
end;
```

J1.4.3.185 FPProcessNaN

```
// FPProcessNaN()
// ==============

func FPProcessNaN{N}(fptype : FPType, op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;   // Generate floating-point exceptions
    return FPProcessNaN{N}(fptype, op, fpcr, fpexc);
end;

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

func FPProcessNaN{N}(fptype : FPType, op : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    var topfrac : integer;

    case N of
        when 16 => topfrac =  9;
        when 32 => topfrac = 22;
        when 64 => topfrac = 51;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var result : bits(N) = op;
    if fptype == FPType_SNaN then
        result[topfrac] = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
    end;
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN{N}(fpcr);
    end;
    return result;
end;
```

J1.4.3.186 FPProcessNaNs

```
// FPProcessNaNs()
// ===============

func FPProcessNaNs{N}(type1 : FPType, type2 : FPType, op1 : bits(N),
                      op2 : bits(N), fpcr : FPCR_Type) => (boolean, bits(N))
begin
    let fpexc : boolean = TRUE;     // Generate floating-point exceptions
    return FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
end;

// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behavior for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

func FPProcessNaNs{N}(type1 : FPType, type2 : FPType, op1 : bits(N), op2 : bits(N),
                      fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(N))
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    let altfp : boolean    = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let op1_nan : boolean  = type1 IN {FPType_SNaN, FPType_QNaN};
    let op2_nan : boolean  = type2 IN {FPType_SNaN, FPType_QNaN};
    let any_snan : boolean = type1 == FPType_SNaN || type2 == FPType_SNaN;
    let type_nan : FPType = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // [n] register NaN selected
        done = TRUE;  result = FPProcessNaN{N}(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN{N}(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN{N}(type2, op2, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros{N};  // 'Don't care' result
    end;
    return (done, result);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.187 FPProcessNaNs3

```
// FPProcessNaNs3()
// ================

func FPProcessNaNs3{N}(type1 : FPType, type2 : FPType, type3 : FPType,
                       op1 : bits(N), op2 : bits(N), op3 : bits(N),
                       fpcr : FPCR_Type) => (boolean, bits(N))
begin
    let fpexc : boolean = TRUE;   // Generate floating-point exceptions
    return FPProcessNaNs3{N}(type1, type2, type3, op1, op2, op3, fpcr, fpexc);
end;

// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

func FPProcessNaNs3{N}(type1 : FPType, type2 : FPType, type3 : FPType,
                       op1 : bits(N), op2 : bits(N), op3 : bits(N),
                       fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(N))
begin
    assert N IN {16,32,64};
    var result : bits(N);
    let op1_nan : boolean = type1 IN {FPType_SNaN, FPType_QNaN};
    let op2_nan : boolean = type2 IN {FPType_SNaN, FPType_QNaN};
    let op3_nan : boolean = type3 IN {FPType_SNaN, FPType_QNaN};

    let altfp : boolean   = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    var type_nan : FPType;
    if altfp then
        if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;
        end;
    end;

    var done : boolean;
    if altfp && op1_nan && op2_nan && op3_nan then
        // [n] register NaN selected
        done = TRUE;  result = FPProcessNaN{N}(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // [n] register NaN selected
        done = TRUE;  result = FPProcessNaN{N}(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // [m] register NaN selected
        done = TRUE;  result = FPProcessNaN{N}(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN{N}(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN{N}(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN{N}(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN{N}(type3, op3, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros{N};  // 'Don't care' result
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    return (done, result);
end;
```

J1.4.3.188 FPProcessNaNs4

```
// FPProcessNaNs4()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

func FPProcessNaNs4(type1 : FPType, type2 : FPType, type3 : FPType, type4 : FPType,
                    op1 : bits(16), op2 : bits(16), op3 : bits(16),
                    op4 : bits(16), fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(32))
begin

    var result : bits(32);
    var done : boolean;
    // The FPCR.AH control does not affect these checks
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type1, op1, fpcr, fpexc));
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type3, op3, fpcr, fpexc));
    elsif type4 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type4, op4, fpcr, fpexc));
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type1, op1, fpcr, fpexc));
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type3, op3, fpcr, fpexc));
    elsif type4 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type4, op4, fpcr, fpexc));
    else
        done = FALSE;  result = Zeros{32};  // 'Don't care' result
    end;

    return (done, result);
end;
```

J1.4.3.189 FPRecipEstimate

```
// FPRecipEstimate()
// =================

func FPRecipEstimate{N}(operand : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(N);
    var overflow_to_inf : boolean;
    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;
    if altfp then fpcr.[FIZ,FZ] = '11'; end;
    if altfp then fpcr.RMode    = '00'; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let (fptype,sign,value) : (FPType, bit, real) = FPUnpack{N}(operand, fpcr, fpexc);

let rounding : FPRounding = FPRoundingMode(fpcr);
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    result = FPProcessNaN{N}(fptype, operand, fpcr, fpexc);
elsif fptype == FPType_Infinity then
    result = FPZero{N}(sign);
elsif fptype == FPType_Zero then
    result = FPInfinity{N}(sign);
    if fpexc then FPProcessException(FPExc_DivideByZero, fpcr); end;
elsif (
        (N == 16 && Abs(value) < 2.0^-16) ||
        (N == 32 && Abs(value) < 2.0^-128) ||
        (N == 64 && Abs(value) < 2.0^-1024)
      ) then
    case rounding of
        when FPRounding_TIEEVEN =>
            overflow_to_inf = TRUE;
        when FPRounding_POSINF =>
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF =>
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO =>
            overflow_to_inf = FALSE;
    end;
    result = if overflow_to_inf then FPInfinity{N}(sign) else FPMaxNormal{N}(sign);
    if fpexc then
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    end;
elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
      && (
           (N == 16 && Abs(value) >= 2.0^14) ||
           (N == 32 && Abs(value) >= 2.0^126) ||
           (N == 64 && Abs(value) >= 2.0^1022)
         ) then
    // Result flushed to zero of correct sign
    result = FPZero{N}(sign);

    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
        FPSCR().UFC = '1';
    else
        if fpexc then FPSR().UFC = '1'; end;
    end;
else
    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
    // calculate result exponent. Scaled value has copied sign bit,
    // exponent = 1022 = double-precision biased version of -1,
    // fraction = original fraction
    var fraction : bits(52);
    var exp : integer;
    case N of
        when 16 =>
            fraction = operand[9:0] :: Zeros{42};
            exp = UInt(operand[14:10]);
        when 32 =>
            fraction = operand[22:0] :: Zeros{29};
            exp = UInt(operand[30:23]);
        when 64 =>
            fraction = operand[51:0];
            exp = UInt(operand[62:52]);
    end;

    if exp == 0 then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if fraction[51] == '0' then
                exp = -1;
                fraction = fraction[49:0]::'00';
            else
                fraction = fraction[50:0]::'0';
            end;
        end;

        var scaled : integer;
        let increasedprecision : boolean = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

        if !increasedprecision then
            scaled = UInt('1'::fraction[51:44]);
        else
            scaled = UInt('1'::fraction[51:41]);
        end;

        var result_exp : integer;
        case N of
            when 16 => result_exp = 29 - exp;   // In range 29-30 = -1 to 29+1 = 30
            when 32 => result_exp = 253 - exp;  // In range 253-254 = -1 to 253+1 = 254
            when 64 => result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046
        end;

        // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
        // fixed-point number in range [0.5 .. 1.0].
        let estimate : integer = RecipEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.
        if !increasedprecision then
            fraction = estimate[7:0] :: Zeros{44};
        else
            fraction = estimate[11:0] :: Zeros{40};
        end;

        if result_exp == 0 then
            fraction = '1' :: fraction[51:1];
        elsif result_exp == -1 then
            fraction = '01' :: fraction[51:2];
            result_exp = 0;
        end;

        case N of
            when 16 => result = sign :: result_exp[N-12:0] :: fraction[51:42];
            when 32 => result = sign :: result_exp[N-25:0] :: fraction[51:29];
            when 64 => result = sign :: result_exp[N-54:0] :: fraction[51:0];
        end;
    end;

    return result;
end;
```

J1.4.3.190 RecipEstimate

```
// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func RecipEstimate(a_in : integer, increasedprecision : boolean) => integer
begin
    var a : integer = a_in;
    var r : integer;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1;                        // Round to nearest
        let b : integer = (2 ^ 19) DIVRM a;
        r = (b+1) DIVRM 2;                // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1;                        // Round to nearest
        let b : integer = (2 ^ 26) DIVRM a;
        r = (b+1) DIVRM 2;                // Round to nearest
        assert 4096 <= r && r < 8192;
    end;
    return r;
end;
```

J1.4.3.191 FPRecpX

```
// FPRecpX()
// =========

func FPRecpX{N}(op : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var fpcr : FPCR_Type = fpcr_in;
    let isbfloat16 : boolean = FALSE;
    let (F, -) = FPBits(N, isbfloat16);
    let E = (N - F) - 1;
    var result : bits(N);
    var exp : bits(E);
    var max_exp : bits(E);
    let frac : bits(F) = Zeros{};

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    let fpexc : boolean = !altfp;            // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end; // Flush denormal input and output to zero
    let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpexc);
    exp = op[F+:E];
    max_exp = Ones{E} - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN{N}(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then                 // Zero and denormals
            result = ZeroExtend{N}(sign::max_exp::frac);
        else                                // Infinities and normals
            result = ZeroExtend{N}(sign::NOT(exp)::frac);
        end;
    end;

    return result;
end;
```

J1.4.3.192 FPRound

```
// FPRound()
// =========
// Generic conversion from precise, unbounded real data type to IEEE format.

func FPRound{N}(op : real, fpcr : FPCR_Type) => bits(N)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    return FPRound{N}(op, fpcr, FPRoundingMode(fpcr));
end;

// FPRound()
// =========
// For directed FP conversion, includes an explicit 'rounding' argument.

func FPRound{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding) => bits(N)
begin
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    return FPRound{N}(op, fpcr_in, rounding, fpexc);
end;

// FPRound()
// =========
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

func FPRound{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding, fpexc : boolean) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let isbfloat16 : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;
```

J1.4.3.193 FPRoundBase

```
// FPRoundBase()
// =============
// For BFloat16, includes an explicit 'isbfloat16' argument.

func FPRoundBase{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding,
                    isbfloat16 : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;

// FPRoundBase()
// =============
// For FP8 multiply-accumulate, dot product, and outer product instructions, includes
// an explicit saturation overflow argument.

func FPRoundBase{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding, isbfloat16 : boolean,
                    fpexc : boolean) => bits(N)
begin
    let satoflo : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc, satoflo);
end;

// FPRoundBase()
// =============
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate. The 'satoflo' argument
// controls whether overflow generates Infinity or MaxNorm for 8-bit floating-point
// data processing instructions.

func FPRoundBase{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding, isbfloat16 : boolean,
                    fpexc : boolean, satoflo : boolean) => bits(N)
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
assert N IN {16,32,64};
assert op != 0.0;
assert rounding != FPRounding_TIEAWAY;
var result : bits(N);

// Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
let (F, minimum_exp) = FPBits(N, isbfloat16);
let zeros = if N == 32 && isbfloat16 then 16 else 0;
let E = N - (F + 1 + zeros);
// Split value into sign, unrounded mantissa and exponent.
var sign : bit;
var exponent : integer;
var mantissa : real;
if op < 0.0 then
    sign = '1';  mantissa = -op;
else
    sign = '0';  mantissa = op;
end;
(mantissa, exponent) = NormalizeReal(mantissa);

// When TRUE, detection of underflow occurs after rounding and the test for a
// denormalized number for single and double precision values occurs after rounding.
let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';

// Deal with flush-to-zero before rounding if FPCR.AH != '1'.
if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
      exponent < minimum_exp) then
    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
        FPSCR().UFC = '1';
    else
        if fpexc then FPSR().UFC = '1'; end;
    end;
    return FPZero{N}(sign);
end;

var biased_exp_unconstrained : integer = (exponent - minimum_exp) + 1;
var int_mant_unconstrained : integer = RoundDown(mantissa * 2.0^F);
let error_unconstrained : real = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
var biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
// < 2.0^F if biased_exp == 0, >= 2.0^F if not
var int_mant : integer = RoundDown(mantissa * 2.0^F);
var error : real = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
let trapped_UF : boolean = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
    if fpexc then FPProcessException(FPExc_Underflow, fpcr); end;
end;

// Round result according to rounding mode.
var round_up_unconstrained : boolean;
var round_up : boolean;
var overflow_to_inf : boolean;
if altfp then

    case rounding of
        when FPRounding_TIEEVEN =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            round_up_unconstrained = (error_unconstrained > 0.5 ||
               (error_unconstrained == 0.5 && int_mant_unconstrained[0] == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
            overflow_to_inf = !satoflo;
        when FPRounding_POSINF =>
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF =>
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD =>
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;
    end;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained   = int_mant_unconstrained DIV 2;
        end;
    end;

    // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
    if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
        // the result of unconstrained rounding is less than the minimum normalized number
        if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then   // Flush-to-zero
            if fpexc then
                FPSR().UFC = '1';
                FPProcessException(FPExc_Inexact, fpcr);
            end;
            return FPZero{N}(sign);
        elsif error != 0.0 || trapped_UF then
            if fpexc then FPProcessException(FPExc_Underflow, fpcr); end;
        end;
    end;
else    // altfp == FALSE
    case rounding of
        when FPRounding_TIEEVEN =>
            round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
            overflow_to_inf = !satoflo;
        when FPRounding_POSINF =>
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF =>
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD =>
            round_up = FALSE;
            overflow_to_inf = FALSE;
    end;
end;

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then      // Rounded up from denormalized to normalized
        biased_exp = 1;
    end;
    if int_mant == 2^(F+1) then  // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD && int_mant[0] == '0' then
        int_mant = int_mant + 1;
    end;

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity{N}(sign) else FPMaxNormal{N}(sign);
            if fpexc then FPProcessException(FPExc_Overflow, fpcr); end;
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0] :: Zeros{N-(E+F+1)};
        end;
    else                                    // Alternative half precision
        if biased_exp >= 2^E then
            result = sign :: Ones{N-1};
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0] :: Zeros{N-(E+F+1)};
        end;
    end;

    // Deal with Inexact exception.
    if error != 0.0 then
        if fpexc then FPProcessException(FPExc_Inexact, fpcr); end;
    end;

    return result;
end;
```

J1.4.3.194 FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

func FPRoundCV{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.FZ16 = '0';
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    let isbfloat16 : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;
```

J1.4.3.195 FPRound_FP8

```
// FPRound_FP8()
// =============
// Used by FP8 multiply-accumulate, dot product, and outer product instructions
// which observe FPMR.OSM.

func FPRound_FP8{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding,
                    satoflo : boolean) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let fpexc : boolean = FALSE;
    let isbfloat16 : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc, satoflo);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.196 FPRounding

```
// FPRounding
// ==========
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

type FPRounding of enumeration {FPRounding_TIEEVEN, FPRounding_POSINF,
                                FPRounding_NEGINF,  FPRounding_ZERO,
                                FPRounding_TIEAWAY, FPRounding_ODD};
```

J1.4.3.197 FPRoundingMode

```
// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

func FPRoundingMode(fpcr : FPCR_Type) => FPRounding
begin
    return FPDecodeRounding(fpcr.RMode);
end;
```

J1.4.3.198 FPRoundInt

```
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

func FPRoundInt{N}(op : bits(N), fpcr : FPCR_Type, rounding : FPRounding,
                   exact : boolean) => bits(N)
begin
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpexc);

    var result : bits(N);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN{N}(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity{N}(sign);
    elsif fptype == FPType_Zero then
        result = FPZero{N}(sign);
    else
        // Extract integer component.
        var int_result : integer = RoundDown(value);
        let error : real = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        var round_up : boolean;
        case rounding of
            when FPRounding_TIEEVEN =>
                round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
            when FPRounding_POSINF =>
                round_up = (error != 0.0);
            when FPRounding_NEGINF =>
                round_up = FALSE;
            when FPRounding_ZERO =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY =>
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
        end;

        if round_up then int_result = int_result + 1; end;

        // Convert integer value into an equivalent real value.
        let real_result : real = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero{N}(sign);
        else
            result = FPRound{N}(real_result, fpcr, FPRounding_ZERO);
        end;

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);
        end;
    end;

    return result;
end;
```

J1.4.3.199 FPRoundIntN

```
// FPRoundIntN()
// =============

func FPRoundIntN{N}(op : bits(N), fpcr : FPCR_Type,
                    rounding : FPRounding, intsize : integer) => bits(N)
begin
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    var exp : integer;
    var result : bits(N);
    var round_up : boolean;
    let E : integer{} = (if N == 32 then 8 else 11);
    let F : integer{} = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    let altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpexc);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp    = 126 + intsize;
            result = '1'::exp[(E-1):0]::Zeros{F};
        else
            exp    = 1022+intsize;
            result = '1'::exp[(E-1):0]::Zeros{F};
        end;
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero{N}(sign);
    else
        // Extract integer component.
        var int_result = RoundDown(value);
        var error      = value - Real(int_result);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding_TIEEVEN =>
                round_up = error > 0.5 || (error == 0.5 && int_result[0] == '1');
            when FPRounding_POSINF =>
                round_up = error != 0.0;
            when FPRounding_NEGINF =>
                round_up = FALSE;
            when FPRounding_ZERO =>
                round_up = error != 0.0 && int_result < 0;
            when FPRounding_TIEAWAY =>
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);
        end;

        if round_up then int_result = int_result + 1; end;
        let overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp    = 126 + intsize;
                result = '1'::exp[(E-1):0]::Zeros{F};
            else
                exp    = 1022 + intsize;
                result = '1'::exp[(E-1):0]::Zeros{F};
            end;
            FPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            let real_result : real = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero{N}(sign);
            else
                result = FPRound{N}(real_result, fpcr, FPRounding_ZERO);
            end;
        end;

        // Generate inexact exceptions.
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);
        end;
    end;

    return result;
end;
```

J1.4.3.200 FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

func FPRSqrtEstimate{N}(operand : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var fpcr : FPCR_Type = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;
    if altfp then fpcr.[FIZ,FZ] = '11'; end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
let (fptype,sign,value) : (FPType, bit, real) = FPUnpack{N}(operand, fpcr, fpexc);

var result : bits(N);
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    result = FPProcessNaN{N}(fptype, operand, fpcr, fpexc);
elsif fptype == FPType_Zero then
    result = FPInfinity{N}(sign);
    if fpexc then FPProcessException(FPExc_DivideByZero, fpcr); end;
elsif sign == '1' then
    result = FPDefaultNaN{N}(fpcr);
    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
elsif fptype == FPType_Infinity then
    result = FPZero{N}('0');
else
    // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
    // evenness or oddness of the exponent unchanged, and calculate result exponent.
    // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
    // biased version of -1 or -2, fraction = original fraction extended with zeros.

    var fraction : bits(52);
    var exp : integer;
    case N of
        when 16 =>
            fraction = operand[9:0] :: Zeros{42};
            exp      = UInt(operand[14:10]);
        when 32 =>
            fraction = operand[22:0] :: Zeros{29};
            exp      = UInt(operand[30:23]);
        when 64 =>
            fraction = operand[51:0];
            exp      = UInt(operand[62:52]);
    end;

    if exp == 0 then
        while fraction[51] == '0' looplimit 51 do
            fraction = fraction[50:0] :: '0';
            exp      = exp - 1;
        end;
        fraction = fraction[50:0] :: '0';
    end;

    var scaled : integer;
    let increasedprecision : boolean = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

    if !increasedprecision then
        if exp[0] == '0' then
            scaled = UInt('1'::fraction[51:44]);
        else
            scaled = UInt('01'::fraction[51:45]);
        end;
    else
        if exp[0] == '0' then
            scaled = UInt('1'::fraction[51:41]);
        else
            scaled = UInt('01'::fraction[51:42]);
        end;
    end;

    var result_exp : integer;
    case N of
        when 16 => result_exp = (  44 - exp) DIVRM 2;
        when 32 => result_exp = ( 380 - exp) DIVRM 2;
        when 64 => result_exp = (3068 - exp) DIVRM 2;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        let estimate : integer = RecipSqrtEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 => result = '0' :: result_exp[N-12:0] :: estimate[7:0]::Zeros{2};
            when 32 =>
                if !increasedprecision then
                    result = '0' :: result_exp[N-25:0] :: estimate[7:0]::Zeros{15};
                else
                    result = '0' :: result_exp[N-25:0] :: estimate[11:0]::Zeros{11};
                end;
            when 64 => result = '0' :: result_exp[N-54:0] :: estimate[7:0]::Zeros{44};
        end;
    end;

    return result;
end;
```

J1.4.3.201 RecipSqrtEstimate

```
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

func RecipSqrtEstimate(a_in : integer, increasedprecision : boolean) => integer
begin
    var a : integer = a_in;
    var r : integer;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then                     // a in [128, 255], represents a value in [0.25, 0.5)
            a = a*2+1;                      // promote to 9-bit range in units of 1/512
        else                                // a in [256, 511], represents a value in [0.5, 1.0)
            a = (a >> 1) << 1;              // Discard bottom bit
            a = (a+1)*2;                    // round up to nearest 1/256 and convert
        end;
        var b : integer = 512;
        while a*(b+1)*(b+1) < 2^28 looplimit 510 do
            b = b+1;
        end;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIVRM 2;                  // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        if a < 2048 then                    // a in [1024, 2047], represents a value in [0.25, 0.5)
            a = a*2 + 1;                    // promote to 13-bit range in units of 1/8192
        else                                // a in [2048, 4095], represents a value in [0.5, 1.0)
            a = (a >> 1) << 1;              // Discard bottom bit
            a = (a+1)*2;                    // round up to nearest 1/4096 and convert
        end;
        var b : integer = 8192;
        while a*(b+1)*(b+1) < 2^39 looplimit (2^39 - (2049 * 8193^2)) do
            b = b+1;
        end;
        r = (b+1) DIVRM 2;                  // Round to nearest
        assert 4096 <= r && r < 8192;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    return r;
end;
```

J1.4.3.202 FPSqrt

```
// FPSqrt()
// ========

func FPSqrt{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var (fptype,sign,value) : (FPType, bit, real) = FPUnpack{N}(op, fpcr);

    var result : bits(N);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN{N}(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero{N}(sign);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity{N}(sign);
    elsif sign == '1' then
        result = FPDefaultNaN{N}(fpcr);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        var prec : integer;
        if N == 16 then
            prec = 13; // 10 fraction bits + 3
        elsif N == 32 then
            prec = 26; // 23 fraction bits + 3
        else // N == 64
            prec = 55; // 52 fraction bits + 3
        end;
        value = SqrtRounded(value, prec);
        result = FPRound{N}(value, fpcr);
        FPProcessDenorm(fptype, N, fpcr);

    end;
    return result;
end;
```

J1.4.3.203 FPSub

```
// FPSub()
// =======

func FPSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;       // Generate floating-point exceptions
    return FPSub{N}(op1, op2, fpcr, fpexc);
end;

// FPSub()
// =======

func FPSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin

    assert N IN {16,32,64};
    let rounding : FPRounding = FPRoundingMode(fpcr);

    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);

    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !done then
        let inf1 : boolean = (type1 == FPType_Infinity);
        let inf2 : boolean = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity{N}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity{N}('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero{N}(sign1);
        else
            let result_value : real = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
    end;
    return result;
end;
```

J1.4.3.204 FPSub_ZA

```
// FPSub_ZA()
// ==========
// Calculates op1-op2 for SME2 ZA-targeting instructions.

func FPSub_ZA{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPSub{N}(op1, op2, fpcr, fpexc);
end;
```

J1.4.3.205 FPThree

```
// FPThree()
// =========

func FPThree{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = '1'::Zeros{E-1};
    let frac : bits(F) = '1'::Zeros{F-1};
    let result : bits((F + E) + 1) = sign :: exp :: frac;

    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.206 FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

func FPToFixed{M, N}(op : bits(N), fbits : integer, unsigned : boolean, fpcr : FPCR_Type,
                     rounding : FPRounding) => bits(M)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    let altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    var (fptype,sign,value) : (FPType, bit, real) = FPUnpack{N}(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    end;

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    var int_result : integer = RoundDown(value);
    let error : real         = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    var round_up : boolean;
    case rounding of
        when FPRounding_TIEEVEN =>
            round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
        when FPRounding_POSINF =>
            round_up = (error != 0.0);
        when FPRounding_NEGINF =>
            round_up = FALSE;
        when FPRounding_ZERO =>
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY =>
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
    end;

    if round_up then int_result = int_result + 1; end;

    // Generate saturated result and exceptions.
    let (result, overflow) : (bits(M), boolean) = SatQ{M}(int_result, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
    end;

    return result;
end;
```

J1.4.3.207 FPToFixedJS

```
// FPToFixedJS()
// =============
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

func FPToFixedJS(op : bits(64), fpcr : FPCR_Type) => (bits(32), bit)
begin

    // If FALSE, never generate Input Denormal floating-point exceptions.
    let fpexc_idenorm : boolean = !(IsFeatureImplemented(FEAT_AFP) &&
                                    !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    let (fptype,sign,value) = FPUnpack{64}(op, fpcr, fpexc_idenorm);

    var z : bit = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    end;

    var int_result = RoundDown(value);
    let error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    let round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1; end;

    var result : integer;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));
    end;

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op[51:0]) then
        z = '0';
    end;

    if fptype == FPType_Infinity then result = 0; end;

    return (result[31:0], z);
end;
```

J1.4.3.208 FPTwo

```
// FPTwo()
// =======

func FPTwo{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = '1'::Zeros{E-1};
    let frac : bits(F) = Zeros{};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let result : bits((F + E) + 1) = sign :: exp :: frac;
    return result;
end;
```

J1.4.3.209 FPType

```
// FPType
// ======

type FPType of enumeration {FPType_Zero,
                            FPType_Denormal,
                            FPType_Nonzero,
                            FPType_Infinity,
                            FPType_QNaN,
                            FPType_SNaN};
```

J1.4.3.210 FPUnpack

```
// FPUnpack()
// ==========

func FPUnpack{N}(fpval : bits(N), fpcr_in : FPCR_Type) => (FPType, bit, real)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let fpexc : boolean = TRUE;   // Generate floating-point exceptions
    let (fp_type, sign, value) : (FPType, bit, real) = FPUnpackBase{N}(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
end;

// FPUnpack()
// ==========
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

func FPUnpack{N}(fpval : bits(N), fpcr_in : FPCR_Type, fpexc : boolean) => (FPType, bit, real)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let (fp_type, sign, value) : (FPType, bit, real) = FPUnpackBase{N}(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
end;
```

J1.4.3.211 FPUnpackBase

```
// FPUnpackBase()
// ==============

func FPUnpackBase{N}(fpval : bits(N), fpcr : FPCR_Type, fpexc : boolean) => (FPType, bit, real)
begin
    let isbfloat16 : boolean = FALSE;
    let (fp_type, sign, value) = FPUnpackBase{N}(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);
end;

// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
//
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls the
// generation of floating-point exceptions and 'isbfloat16' determines whether
// N=16 signifies BFloat16 or half-precision type. Status information is updated
// directly in the FPSR where appropriate.

func FPUnpackBase{N}(fpval : bits(N), fpcr_in : FPCR_Type, fpexc : boolean,
                     isbfloat16 : boolean) => (FPType, bit, real)
begin
    assert N IN {16,32,64};

    let fpcr : FPCR_Type = fpcr_in;

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
    let fiz : boolean   = altfp && fpcr.FIZ == '1';
    let fz : boolean    = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    var value : real;
    var sign : bit;
    var fptype : FPType;

    if N == 16 && !isbfloat16 then
        sign   = fpval[15];
        let exp16 : bits(5)   = fpval[14:10];
        let frac16 : bits(10) = fpval[9:0];
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
            end;
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac16[9] == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
            end;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);
        end;

    elsif N == 32 || isbfloat16 then
        var exp32 : bits(8);
        var frac32 : bits(23);
        if isbfloat16 then
            sign   = fpval[15];
            exp32  = fpval[14:7];
            frac32 = fpval[6:0] :: Zeros{16};
        else
            sign   = fpval[31];
            exp32  = fpval[30:23];
            frac32 = fpval[22:0];
        end;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr); end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                end;
            else
                fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
            end;
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac32[22] == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
            end;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
        end;

    else // N == 64
        sign   = fpval[63];
        let exp64 : bits(11)  = fpval[62:52];
        let frac64 : bits(52) = fpval[51:0];

        if IsZero(exp64) then
            if IsZero(frac64) then
                // Produce zero if value is zero.
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr); end;
                end;
            else
                fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
            end;
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac64[51] == '1' then FPType_QNaN else FPType_SNaN;
                value  = 0.0;
            end;
        else
            fptype = FPType_Nonzero;
            value  = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
        end;
    end;

    if sign == '1' then value = -value; end;

    return (fptype, sign, value);
end;
```

J1.4.3.212 FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

func FPUnpackCV{N}(fpval : bits(N), fpcr_in : FPCR_Type) => (FPType, bit, real)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.FZ16 = '0';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let fpexc : boolean = TRUE;   // Generate floating-point exceptions
    let (fp_type, sign, value) : (FPType, bit, real) = FPUnpackBase{N}(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
end;
```

J1.4.3.213 FPZero

```
// FPZero()
// ========

func FPZero{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{}  = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{}  = N - (E + 1);
    let exp : bits(E)  = Zeros{};
    let frac : bits(F) = Zeros{};
    let result : bits((F + E) + 1) = sign :: exp :: frac;
    return result;
end;
```

J1.4.3.214 VFPExpandImm

```
// VFPExpandImm()
// ==============

func VFPExpandImm{N}(imm8 : bits(8)) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = (N - E) - 1;
    let sign : bit = imm8[7];
    let exp : bits(E)  = NOT(imm8[6])::Replicate{E - 3}(imm8[6])::imm8[5:4];
    let frac : bits(F) = imm8[3:0]::Zeros{F-4};
    let result : bits((F + E) + 1) = sign :: exp :: frac;

    return result;
end;
```

J1.4.3.215 AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

func AddWithCarry{N}(x : bits(N), y : bits(N), carry_in : bit) => (bits(N), bits(4))
begin
    let unsigned_sum : integer = UInt(x) + UInt(y) + UInt(carry_in);
    let signed_sum : integer = SInt(x) + SInt(y) + UInt(carry_in);
    let result : bits(N) = unsigned_sum[N-1:0]; // same value as signed_sum[N-1:0]
    let n : bit = result[N-1];
    let z : bit = if IsZero(result) then '1' else '0';
    let c : bit = if UInt(result) == unsigned_sum then '0' else '1';
    let v : bit = if SInt(result) == signed_sum then '0' else '1';
    return (result, n::z::c::v);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.216 InterruptID

```
// InterruptID
// ===========

type InterruptID of enumeration {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
    InterruptID_PMBIRQ,
    InterruptID_HACDBSIRQ,

    InterruptID_TRBIRQ,
};
```

J1.4.3.217 SetInterruptRequestLevel

```
// SetInterruptRequestLevel()
// ==========================
// Set a level-sensitive interrupt to the specified level.

impdef func SetInterruptRequestLevel(id : InterruptID, level : Signal)
begin
    return;
end;
```

J1.4.3.218 AArch64_BranchAddr

```
// AArch64_BranchAddr()
// ====================
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

func AArch64_BranchAddr(vaddress : bits(64), el : bits(2)) => bits(64)
begin
    assert !UsingAArch32();
    let msbit : integer{} = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress[msbit] == '1' then
        return SignExtend{64}(vaddress[msbit:0]);
    else
        return ZeroExtend{64}(vaddress[msbit:0]);
    end;
end;
```

J1.4.3.219 AccessDescriptor

```
// AccessDescriptor
// ================
// Memory access or translation invocation details that steer architectural behavior


type AccessDescriptor of record {
    acctype : AccessType,
    el : bits(2),                  // Acting EL for the access
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    ss : SecurityState,            // Acting Security State for the access
    acqsc : boolean,               // Acquire with Sequential Consistency
    acqpc : boolean,               // FEAT_LRCPC: Acquire with Processor Consistency
    relsc : boolean,               // Release with Sequential Consistency
    limitedordered : boolean,      // FEAT_LOR: Acquire/Release with limited ordering
    exclusive : boolean,           // Access has Exclusive semantics
    atomicop : boolean,            // FEAT_LSE: Atomic read-modify-write access
    modop : MemAtomicOp,           // FEAT_LSE: The modification operation in the 'atomicop' access
    nontemporal : boolean,         // Hints the access is non-temporal
    read : boolean,                // Read from memory or only require read permissions
    write : boolean,               // Write to memory or only require write permissions
    cacheop : CacheOp,             // DC/IC: Cache operation
    opscope : CacheOpScope,        // DC/IC: Scope of cache operation
    cachetype : CacheType,         // DC/IC: Type of target cache
    pan : boolean,                 // FEAT_PAN: The access is subject to PSTATE.PAN
    nonfault : boolean,            // SVE: Non-faulting load
    firstfault : boolean,          // SVE: First-fault load
    first : boolean,               // SVE: First-fault load for the first active element
    contiguous : boolean,          // SVE: Contiguous load/store not gather load/scatter store
    predicated : boolean,          // SVE: Predicated load/store
    streamingsve : boolean,        // SME: Access made by PE while in streaming SVE mode
    ls64 : boolean,                // FEAT_LS64: Accesses by accelerator support loads/stores
    withstatus : boolean,          // FEAT_LS64: Store with status result
    mops : boolean,                // FEAT_MOPS: Memory operation (CPY/SET) accesses
    rcw : boolean,                 // FEAT_THE: Read-Check-Write access
    rcws : boolean,                // FEAT_THE: Read-Check-Write Software access
    toplevel : boolean,            // FEAT_THE: Translation table walk access for TTB address
    varange : VARange,             // FEAT_THE: The corresponding TTBR supplying the TTB
    a32lsmd : boolean,             // A32 Load/Store Multiple Data access
    tagchecked : boolean,          // FEAT_MTE2: Access is tag checked
    tagaccess : boolean,           // FEAT_MTE: Access targets the tag bits
    stzgm : boolean,               // FEAT_MTE: Accesses that store Allocation tags to Device
                                   //           memory are CONSTRAINED UNPREDICTABLE
    Rt : integer,                  // Register named Rt in the instruction
    Rt2 : integer,                 // Register named Rt2 in the instruction
    Rs : integer,                  // Register named Rs in the instruction
    Rs2 : integer,                 // Register named Rs2 in the instruction
    ispair : boolean,              // Access represents a Load/Store pair access
    highestaddressfirst : boolean, // FEAT_LRCPC3: Highest address is accessed first
    lowestaddress : boolean,       // Is the current access the lowest address accessed by
                                   // this instruction
    mpam : MPAMinfo                // FEAT_MPAM: MPAM information
};
```

J1.4.3.220 AccessType

```
// AccessType
// ==========

type AccessType of enumeration {
    AccessType_IFETCH,     // Instruction FETCH
    AccessType_GPR,        // Software load/store to a General Purpose Register
    AccessType_FP,         // Software load/store to an FP register
    AccessType_ASIMD,      // Software ASIMD extension load/store instructions
    AccessType_SVE,        // Software SVE load/store instructions
    AccessType_SME,        // Software SME load/store instructions
    AccessType_IC,         // Sysop IC
    AccessType_DC,         // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,     // Sysop DC {Z,G,GZ}VA
    AccessType_AT,         // Sysop AT
    AccessType_NV2,        // NV2 memory redirected access
    AccessType_SPE,        // Statistical Profiling buffer access
    AccessType_GCS,        // Guarded Control Stack access
    AccessType_TRBE,       // Trace Buffer access
    AccessType_GPTW,       // Granule Protection Table Walk
    AccessType_HACDBS,     // Access to the HACDBS structure
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AccessType_HDBSS,       // Access to entries in HDBSS
    AccessType_TTW          // Translation Table Walk
};
```

J1.4.3.221 AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

func AddrTop(address : bits(64), IsInstr : boolean, el : bits(2)) => AddressSize
begin
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
        end;
    end;
end;
```

J1.4.3.222 AddressSize

```
// AddressSize
// ============

type AddressSize of integer{3..64};
```

J1.4.3.223 AlignmentEnforced

```
// AlignmentEnforced()
// ===================
// For the active translation regime, determine if alignment is required by all accesses

func AlignmentEnforced() => boolean
begin
    let regime : Regime = TranslationRegime(PSTATE.EL);
    var A : bit;
    case regime of
        when Regime_EL30 => A = SCTLR().A;
        when Regime_EL3 =>  A = SCTLR_EL3().A;
        when Regime_EL2 =>  A = if ELUsingAArch32(EL2) then HSCTLR().A else SCTLR_EL2().A;
        when Regime_EL20 => A = SCTLR_EL2().A;
        when Regime_EL10 => A = if ELUsingAArch32(EL1) then SCTLR().A  else SCTLR_EL1().A;
        otherwise => unreachable;
    end;
    return A == '1';
end;
```

J1.4.3.224 AllInAlignedQuantity

```
// AllInAlignedQuantity()
// ======================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

readonly func AllInAlignedQuantity{N : integer{32, 64}}(address : bits(N),
                                                        size : integer,
                                                        alignment : integer) => boolean
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    return (AlignDownSize(address+(size-1), alignment as AddressSize) ==
            AlignDownSize(address, alignment as AddressSize));
end;
```

J1.4.3.225 Allocation

```
// Allocation hints
// ================

constant MemHint_No : bits(2)  = '00';    // No Read-Allocate, No Write-Allocate
constant MemHint_WA : bits(2)  = '01';    // No Read-Allocate, Write-Allocate
constant MemHint_RA : bits(2)  = '10';    // Read-Allocate, No Write-Allocate
constant MemHint_RWA : bits(2) = '11';    // Read-Allocate, Write-Allocate
```

J1.4.3.226 BigEndian

```
// BigEndian()
// ===========

func BigEndian(acctype : AccessType) => boolean
begin
    var bigend : boolean;
    if IsFeatureImplemented(FEAT_NV2) && acctype == AccessType_NV2 then
        return SCTLR_EL2().EE == '1';
    end;

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_ELx().E0E != '0');
    else
        bigend = (SCTLR_ELx().EE != '0');
    end;
    return bigend;
end;
```

J1.4.3.227 BigEndianReverse

```
// BigEndianReverse()
// ==================

func BigEndianReverse{width}(value : bits(width)) => bits(width)
begin
    assert width IN {8, 16, 32, 64, 128, 256};

    if width == 8 then return value; end;
    return Reverse{width}(value, 8);
end;
```

J1.4.3.228 CacheOp

```
// CacheOp
// =======

type CacheOp of enumeration {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.229 CacheOpScope

```
// CacheOpScope
// ============

type CacheOpScope of enumeration {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoE,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_PoPA,
    CacheOpScope_PoPS,
    CacheOpScope_OuterCache,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

J1.4.3.230 CachePASpace

```
// CachePASpace
// ============

type CachePASpace of enumeration {
    CPAS_NonSecure,
    CPAS_Any,                   // Applicable only for DC *SW / IC IALLU* in Root state:
                                // match entries from any PA Space
    CPAS_RealmNonSecure,        // Applicable only for DC *SW / IC IALLU* in Realm state:
                                // match entries from Realm or Non-Secure PAS
    CPAS_Realm,
    CPAS_Root,
    CPAS_SystemAgent,           // Applicable only for DC by PA:
                                // match entries from the System Agent PAS
    CPAS_NonSecureProtected,    // Applicable only for DC by PA:
                                // match entries from the Non-Secure Protected PAS
    CPAS_NA6,                   // Reserved
    CPAS_NA7,                   // Reserved
    CPAS_SecureNonSecure,       // Applicable only for DC *SW / IC IALLU* in Secure state:
                                // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

J1.4.3.231 CacheType

```
// CacheType
// =========

type CacheType of enumeration {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};
```

J1.4.3.232 Cacheability

```
// Cacheability attributes
// =======================

constant MemAttr_NC : bits(2) = '00';     // Non-cacheable
constant MemAttr_WT : bits(2) = '10';     // Write-through
constant MemAttr_WB : bits(2) = '11';     // Write-back
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.233 CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// ======================
// Access descriptor for A32 loads/store multiple general purpose registers

func CreateAccDescA32LSMD(memop : MemOp) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.a32lsmd         = TRUE;

    return accdesc;
end;
```

J1.4.3.234 CreateAccDescASIMD

```
// CreateAccDescASIMD()
// ====================
// Access descriptor for ASIMD&FP loads/stores

func CreateAccDescASIMD(memop : MemOp, nontemporal : boolean, tagchecked : boolean,
                        privileged : boolean) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    return CreateAccDescASIMD(memop, nontemporal, tagchecked, privileged, ispair);
end;

// CreateAccDescASIMD()
// ====================

func CreateAccDescASIMD(memop : MemOp, nontemporal : boolean, tagchecked : boolean,
                        privileged : boolean, ispair : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_ASIMD);


    accdesc.nontemporal     = nontemporal;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.ispair          = ispair;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;
    return accdesc;
end;
```

J1.4.3.235 CreateAccDescASIMDAcqRel

```
// CreateAccDescASIMDAcqRel()
// ==========================
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

func CreateAccDescASIMDAcqRel(memop : MemOp, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_ASIMD);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.acqpc           = memop == MemOp_LOAD;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;

    return accdesc;
end;
```

J1.4.3.236 CreateAccDescAT

```
// CreateAccDescAT()
// =================
// Access descriptor for address translation operations

func CreateAccDescAT(ss : SecurityState, el : bits(2), ataccess : ATAccess) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_AT);

    accdesc.el              = el;
    accdesc.ss              = ss;
    if ImpDefBool("MPAM uses the EL targeted by the AT instruction") then
        accdesc.mpam = GenMPAMAtEL(AccessType_AT, el);
    end;
    case ataccess of
        when ATAccess_Read =>
            accdesc.(read, write, pan) = (TRUE, FALSE, FALSE);
        when ATAccess_ReadPAN =>
            accdesc.(read, write, pan) = (TRUE, FALSE, TRUE);
        when ATAccess_Write =>
            accdesc.(read, write, pan) = (FALSE, TRUE, FALSE);
        when ATAccess_WritePAN =>
            accdesc.(read, write, pan) = (FALSE, TRUE, TRUE);
        when ATAccess_Any =>
            accdesc.(read, write, pan) = (FALSE, FALSE, FALSE);
    end;

    return accdesc;
end;
```

J1.4.3.237 CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====================
// Access descriptor for general purpose register loads/stores with ordering semantics

func CreateAccDescAcqRel(memop : MemOp, tagchecked : boolean, acqsc : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescAcqRel(memop, tagchecked, acqsc, Rt);
end;

func CreateAccDescAcqRel(memop : MemOp, tagchecked : boolean,
                         acqsc : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescAcqRel(memop, tagchecked, ispair, acqsc, Rt, Rt2);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;

func CreateAccDescAcqRel(memop : MemOp, tagchecked : boolean, ispair : boolean,
                         acqsc : boolean, Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acqsc;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
end;
```

J1.4.3.238 CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =======================
// Access descriptor for atomic read-modify-write memory accesses

func CreateAccDescAtomicOp(modop : MemAtomicOp, acquire : boolean, release : boolean,
                           tagchecked : boolean, privileged : boolean, Rt : integer,
                           Rs : integer) => AccessDescriptor
begin
    let Rt2 : integer = -1;
    let Rs2 : integer = -1;

    return CreateAccDescAtomicOp(modop, acquire, release, tagchecked, privileged, Rt, Rt2, Rs, Rs2);
end;

func CreateAccDescAtomicOp(modop : MemAtomicOp, acquire : boolean, release : boolean,
                           tagchecked : boolean, privileged : boolean, Rt : integer,
                           Rt2 : integer, Rs : integer, Rs2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.relsc           = release;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rs              = Rs;
    accdesc.Rs2             = Rs2;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
end;
```

J1.4.3.239 CreateAccDescDC

```
// CreateAccDescDC()
// =================
// Access descriptor for data cache operations
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func CreateAccDescDC(cache : CacheRecord) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_DC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
end;
```

J1.4.3.240 CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====================
// Access descriptor for data cache zero operations

func CreateAccDescDCZero(cachetype : CacheType) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_DCZero);

    accdesc.write            = TRUE;
    accdesc.pan              = TRUE;
    accdesc.tagchecked       = cachetype == CacheType_Data;
    accdesc.tagaccess        = cachetype IN {CacheType_Tag,
                                             CacheType_Data_Tag};
    accdesc.cachetype        = cachetype;

    return accdesc;
end;
```

J1.4.3.241 CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====================
// Access descriptor for general purpose register loads/stores with exclusive semantics

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, tagchecked : boolean,
                         privileged : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescExLDST(memop, acqrel, tagchecked, privileged, Rt);
end;

// CreateAccDescExLDST()
// =====================

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, tagchecked : boolean,
                         privileged : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescExLDST(memop, acqrel, tagchecked, privileged, ispair, Rt, Rt2);
end;

// CreateAccDescExLDST()
// =====================

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, tagchecked : boolean,
                         privileged : boolean, ispair : boolean,
                         Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc            = acqrel && memop == MemOp_LOAD;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.relsc           = acqrel && memop == MemOp_STORE;
    accdesc.exclusive       = TRUE;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.ispair          = ispair;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
end;
```

J1.4.3.242 CreateAccDescFPAtomicOp

```
// CreateAccDescFPAtomicOp()
// =========================
// Access descriptor for FP atomic read-modify-write memory accesses

func CreateAccDescFPAtomicOp(modop : MemAtomicOp, acquire : boolean, release : boolean,
                             tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_FP);

    accdesc.acqsc           = acquire;
    accdesc.relsc           = release;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;

    return accdesc;
end;
```

J1.4.3.243 CreateAccDescGCS

```
// CreateAccDescGCS()
// ==================
// Access descriptor for memory accesses to the Guarded Control Stack

func CreateAccDescGCS(memop : MemOp, privileged : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GCS);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;

    return accdesc;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.244 CreateAccDescGCSSS1

```
// CreateAccDescGCSSS1()
// =====================
// Access descriptor for memory accesses to the Guarded Control Stack that switch stacks

func CreateAccDescGCSSS1(privileged : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GCS);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_GCSSS1;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;

    return accdesc;
end;
```

J1.4.3.245 CreateAccDescGPR

```
// CreateAccDescGPR()
// ==================
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
                      tagchecked : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, Rt);
end;

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
                      tagchecked : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, ispair, Rt, Rt2);
end;

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
                      tagchecked : boolean, ispair : boolean,
                      Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;
    accdesc.ispair          = ispair;

    return accdesc;
end;
```

J1.4.3.246 CreateAccDescGPTW

```
// CreateAccDescGPTW()
// ===================
// Access descriptor for Granule Protection Table walks
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func CreateAccDescGPTW(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;

    return accdesc;
end;
```

J1.4.3.247 CreateAccDescHACDBS

```
// CreateAccDescHACDBS()
// =====================
// Access descriptor for memory accesses to the HACDBS structure.

func CreateAccDescHACDBS() => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_HACDBS);

    accdesc.read            = TRUE;

    return accdesc;
end;
```

J1.4.3.248 CreateAccDescHDBSS

```
// CreateAccDescHDBSS()
// ====================
// Access descriptor for appending entries to the HDBSS

func CreateAccDescHDBSS(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_HDBSS, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.write           = TRUE;

    return accdesc;
end;
```

J1.4.3.249 CreateAccDescIC

```
// CreateAccDescIC()
// =================
// Access descriptor for instruction cache operations

func CreateAccDescIC(cache : CacheRecord) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_IC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.250 CreateAccDescIFetch

```
// CreateAccDescIFetch()
// =====================
// Access descriptor for instruction fetches

func CreateAccDescIFetch() => AccessDescriptor
begin
    let accdesc : AccessDescriptor = NewAccDesc(AccessType_IFETCH);

    return accdesc;
end;
```

J1.4.3.251 CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// ======================
// Access descriptor for general purpose register loads with local ordering semantics

func CreateAccDescLDAcqPC(tagchecked : boolean, acquire : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescLDAcqPC(tagchecked, ispair, acquire, Rt, Rt2);
end;

func CreateAccDescLDAcqPC(tagchecked : boolean, ispair : boolean, acquire : boolean,
                          Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqpc           = acquire;
    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
end;
```

J1.4.3.252 CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====================
// Access descriptor for tag memory loads/stores

func CreateAccDescLDGSTG(memop : MemOp, stzgm : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescLDGSTG(memop, stzgm, ispair, Rt, Rt2);
end;

// CreateAccDescLDGSTG()
// =====================

func CreateAccDescLDGSTG(memop : MemOp, stzgm : boolean, ispair : boolean, Rt : integer,
                         Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.tagaccess       = TRUE;
    accdesc.stzgm           = stzgm;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
end;
```

J1.4.3.253 CreateAccDescLOR

```
// CreateAccDescLOR()
// ==================
// Access descriptor for general purpose register loads/stores with limited ordering semantics

func CreateAccDescLOR(memop : MemOp, tagchecked : boolean,
                      acqsc : boolean, Rt : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acqsc;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.limitedordered  = TRUE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;

    return accdesc;
end;
```

J1.4.3.254 CreateAccDescLS64

```
// CreateAccDescLS64()
// ===================
// Access descriptor for accelerator-supporting memory accesses

func CreateAccDescLS64(memop : MemOp, withstatus : boolean,
                       tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.ls64            = TRUE;
    accdesc.withstatus      = withstatus;
    accdesc.tagchecked      = tagchecked;
    return accdesc;
end;
```

J1.4.3.255 CreateAccDescMOPS

```
// CreateAccDescMOPS()
// ===================
// Access descriptor for data memory copy and set instructions

func CreateAccDescMOPS(memop : MemOp, privileged : boolean,
                       nontemporal : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagchecked      = TRUE;

    return accdesc;
end;
```

J1.4.3.256 CreateAccDescNV2

```
// CreateAccDescNV2()
// ==================
// Access descriptor nested virtualization memory indirection loads/stores

func CreateAccDescNV2(memop : MemOp) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_NV2);

    accdesc.el              = EL2;
    accdesc.ss              = SecurityStateAtEL(EL2);
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;

    return accdesc;
end;
```

J1.4.3.257 CreateAccDescRCW

```
// CreateAccDescRCW()
// ==================
// Access descriptor for atomic read-check-write memory accesses

func CreateAccDescRCW(modop : MemAtomicOp, soft : boolean, acquire : boolean,
                      release : boolean, tagchecked : boolean,
                      Rt : integer, Rs : integer) => AccessDescriptor
begin
    let Rt2 : integer = -1;
    let Rs2 : integer = -1;

    return CreateAccDescRCW(modop, soft, acquire, release, tagchecked, Rt, Rt2, Rs, Rs2);
end;

func CreateAccDescRCW(modop : MemAtomicOp, soft : boolean, acquire : boolean,
                      release : boolean, tagchecked : boolean, Rt : integer, Rt2 : integer,
                      Rs : integer, Rs2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.relsc           = release;
    accdesc.rcw             = TRUE;
    accdesc.rcws            = soft;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;
    accdesc.Rs              = Rs;
    accdesc.Rs2             = Rs2;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return accdesc;
end;
```

J1.4.3.258 CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// ====================
// Access descriptor for stage 1 translation table walks

func CreateAccDescS1TTW(toplevel : boolean,
                        varange : VARange,
                        accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_TTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;
    accdesc.toplevel        = toplevel;
    accdesc.varange         = varange;

    return accdesc;
end;
```

J1.4.3.259 CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// ====================
// Access descriptor for stage 2 translation table walks

func CreateAccDescS2TTW(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_TTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;

    return accdesc;
end;
```

J1.4.3.260 CreateAccDescSME

```
// CreateAccDescSME()
// ==================
// Access descriptor for SME loads/stores

func CreateAccDescSME(memop : MemOp, nontemporal : boolean, contiguous : boolean,
                      tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SME);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = TRUE;
    if ImpDefBool("No tag checking of SME LDR & STR instructions") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return accdesc;
end;
```

J1.4.3.261 CreateAccDescSPE

```
// CreateAccDescSPE()
// ==================
// Access descriptor for memory accesses by Statistical Profiling unit

func CreateAccDescSPE(owning_ss : SecurityState, owning_el : bits(2)) => AccessDescriptor
begin
    let mpaminfo : MPAMinfo = GenMPAMAtEL(AccessType_SPE, owning_el);
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SPE, mpaminfo);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;

    return accdesc;
end;
```

J1.4.3.262 CreateAccDescSTGMOPS

```
// CreateAccDescSTGMOPS()
// ======================
// Access descriptor for tag memory set instructions

func CreateAccDescSTGMOPS(privileged : boolean, nontemporal : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagaccess       = TRUE;

    return accdesc;
end;
```

J1.4.3.263 CreateAccDescSVE

```
// CreateAccDescSVE()
// ==================
// Access descriptor for general SVE loads/stores

func CreateAccDescSVE(memop : MemOp, nontemporal : boolean, contiguous : boolean,
                      tagchecked : boolean) => AccessDescriptor
begin
    let predicated : boolean = FALSE;
    return CreateAccDescSVE(memop, nontemporal, contiguous, predicated, tagchecked);
end;

func CreateAccDescSVE(memop : MemOp, nontemporal : boolean, contiguous : boolean,
                      predicated : boolean, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SVE);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.contiguous      = contiguous;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;
    accdesc.predicated      = predicated;

    return accdesc;
end;
```

J1.4.3.264 CreateAccDescSVEFF

```
// CreateAccDescSVEFF()
// ====================
// Access descriptor for first-fault SVE loads

func CreateAccDescSVEFF(contiguous : boolean, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SVE);

    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.firstfault      = TRUE;
    accdesc.first           = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;
    accdesc.predicated      = TRUE;

    return accdesc;
end;
```

J1.4.3.265 CreateAccDescSVENF

```
// CreateAccDescSVENF()
// ====================
// Access descriptor for non-fault SVE loads

func CreateAccDescSVENF(contiguous : boolean, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SVE);

    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.nonfault        = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    end;
    accdesc.predicated      = TRUE;

    return accdesc;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.266 CreateAccDescTRBE

```
// CreateAccDescTRBE()
// ===================
// Access descriptor for memory accesses by Trace Buffer Unit

func CreateAccDescTRBE(owning_ss : SecurityState, owning_el : bits(2)) => AccessDescriptor
begin
    var mpam : MPAMinfo;
    if SelfHostedTraceEnabled() then
        mpam = GenMPAMAtEL(AccessType_TRBE, owning_el);
    else
        mpam = GenMPAMCurEL(AccessType_TRBE);
    end;
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_TRBE, mpam);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;

    return accdesc;
end;
```

J1.4.3.267 CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// ========================
// Access descriptor for translation table entry HW update

func CreateAccDescTTEUpdate(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_TTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_CAS;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;

    return accdesc;
end;
```

J1.4.3.268 DataMemoryBarrier

```
// DataMemoryBarrier()
// ===================

impdef func DataMemoryBarrier(types : MBReqTypes)
begin
    return;
end;
```

J1.4.3.269 DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// ============================

impdef func DataSynchronizationBarrier(scope : MBMaintenanceScope,
                                       types : MBReqTypes,
                                       nXS : boolean)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.270 DeviceType

```
// DeviceType
// ==========
// Extended memory types for Device memory.

type DeviceType of enumeration {DeviceType_GRE, DeviceType_nGRE,
                                DeviceType_nGnRE, DeviceType_nGnRnE};
```

J1.4.3.271 EffectiveMTX

```
// EffectiveMTX()
// ==============
// Returns the effective MTX in the AArch64 stage 1 translation regime for "el".

func EffectiveMTX(address : bits(64), is_instr : boolean, el : bits(2)) => bit
begin
    var mtx : bit;
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if is_instr then
        mtx = '0';
    else
        case regime of
            when EL1 =>
                mtx = if address[55] == '1' then TCR_EL1().MTX1 else TCR_EL1().MTX0;
            when EL2 =>
                if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                    mtx = if address[55] == '1' then TCR_EL2().MTX1 else TCR_EL2().MTX0;
                else
                    mtx = TCR_EL2().MTX;
                end;
            when EL3 =>
                mtx = TCR_EL3().MTX;
        end;
    end;

    return mtx;
end;
```

J1.4.3.272 EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

func EffectiveTBI(address : bits(64), IsInstr : boolean, el : bits(2)) => bit
begin
    var tbi : bit;
    var tbid : bit;
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1 =>
            tbi = if address[55] == '1' then TCR_EL1().TBI1 else TCR_EL1().TBI0;
            if IsFeatureImplemented(FEAT_PAuth) then
                tbid = if address[55] == '1' then TCR_EL1().TBID1 else TCR_EL1().TBID0;
            end;
        when EL2 =>
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tbi = if address[55] == '1' then TCR_EL2().TBI1 else TCR_EL2().TBI0;
                if IsFeatureImplemented(FEAT_PAuth) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    tbid = if address[55] == '1' then TCR_EL2().TBID1 else TCR_EL2().TBID0;
                end;
            else
                tbi = TCR_EL2().TBI;
                if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL2().TBID; end;
            end;
        when EL3 =>
            tbi = TCR_EL3().TBI;
            if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL3().TBID; end;
    end;

    return (if (tbi == '1' && (!IsFeatureImplemented(FEAT_PAuth) || tbid == '0' ||
            !IsInstr)) then '1' else '0');
end;
```

J1.4.3.273 EffectiveTCMA

```
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

func EffectiveTCMA(address : bits(64), el : bits(2)) => bit
begin
    var tcma : bit;
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1 =>
            tcma = if address[55] == '1' then TCR_EL1().TCMA1 else TCR_EL1().TCMA0;
        when EL2 =>
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tcma = if address[55] == '1' then TCR_EL2().TCMA1 else TCR_EL2().TCMA0;
            else
                tcma = TCR_EL2().TCMA;
            end;
        when EL3 =>
            tcma = TCR_EL3().TCMA;
    end;

    return tcma;
end;
```

J1.4.3.274 ErrorState

```
// ErrorState
// ==========
// The allowed error states that can be returned by memory and used by the PE.

type ErrorState of enumeration {
                        ErrorState_UC,          // Uncontainable
                        ErrorState_UEU,         // Unrecoverable state
                        ErrorState_UEO,         // Restartable state
                        ErrorState_UER,         // Recoverable state
                        ErrorState_CE};         // Corrected
```

J1.4.3.275 Fault

```
// Fault
// =====
// Fault types.

type Fault of enumeration {
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    Fault_None,
                    Fault_AccessFlag,
                    Fault_Alignment,
                    Fault_Background,
                    Fault_Domain,
                    Fault_Permission,
                    Fault_Translation,
                    Fault_AddressSize,
                    Fault_SyncExternal,
                    Fault_SyncExternalOnWalk,
                    Fault_SyncParity,
                    Fault_SyncParityOnWalk,
                    Fault_GPCFOnWalk,
                    Fault_GPCFOnOutput,
                    Fault_AsyncParity,
                    Fault_AsyncExternal,
                    Fault_TagCheck,
                    Fault_Debug,
                    Fault_TLBConflict,
                    Fault_BranchTarget,
                    Fault_UnsupportedAtomicHWUpdate,
                    Fault_Lockdown,
                    Fault_Exclusive,
                    Fault_ICacheMaint};
```

J1.4.3.276 FaultRecord

```
// FaultRecord
// ===========
// Fields that relate only to Faults.

type FaultRecord of record {
    statuscode : Fault,             // Fault Status
    accessdesc : AccessDescriptor,  // Details of the faulting access
    vaddress   : bits(64),          // Faulting virtual address
    ipaddress : FullAddress,        // Intermediate physical address
    paddress : FullAddress,         // Physical address
    gpcf : GPCFRecord,              // Granule Protection Check Fault record
    gpcfs2walk : boolean,           // GPC for a stage 2 translation table walk
    s2fs1walk : boolean,            // Is on a Stage 1 translation table walk
    write : boolean,                // TRUE for a write, FALSE for a read
    s1tagnotdata : boolean,         // TRUE for a fault due to tag not accessible at stage 1.
    tagaccess : boolean,            // TRUE for a fault due to NoTagAccess permission.
    level : integer,                // For translation, access flag and Permission faults
    extflag : bit,                  // IMPLEMENTATION DEFINED syndrome for External aborts
    secondstage : boolean,          // Is a Stage 2 abort
    assuredonly : boolean,          // Stage 2 Permission fault due to AssuredOnly attribute
    toplevel    : boolean,          // Stage 2 Permission fault due to TopLevel
    overlay : boolean,              // Fault due to overlay permissions
    dirtybit : boolean,             // Fault due to dirty state
    domain : bits(4),               // Domain number, AArch32 only
    merrorstate : ErrorState,       // Incoming error state from memory
    hdbssf : boolean,               // Fault caused by HDBSS
    watchptinfo : WatchpointInfo,   // Watchpoint related fields
    debugmoe : bits(4)              // Debug method of entry, from AArch32 only
};
```

J1.4.3.277 FullAddress

```
// FullAddress
// ===========
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has NUM_PABITS bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// but never exceeds NUM_PABITS bits.

type FullAddress of record {
    paspace : PASpace,
    address : bits(NUM_PABITS)
};
```

J1.4.3.278 GPCF

```
// GPCF
// ====
// Possible Granule Protection Check Fault reasons

type GPCF of enumeration {
    GPCF_None,        // No fault
    GPCF_AddressSize, // GPT address size fault
    GPCF_Walk,        // GPT walk fault
    GPCF_EABT,        // Synchronous External abort on GPT fetch
    GPCF_Fail         // Granule protection fault
};
```

J1.4.3.279 GPCFRecord

```
// GPCFRecord
// ==========
// Full details of a Granule Protection Check Fault

type GPCFRecord of record {
    gpf : GPCF,
    level : integer
};
```

J1.4.3.280 Hint_Prefetch

```
// Hint_Prefetch()
// ===============
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

impdef func Hint_Prefetch(address : bits(64),
                          hint : PrefetchHint,
                          target : integer,
                          stream : boolean)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.281 Hint_RangePrefetch

```
// Hint_RangePrefetch()
// ====================
// Signals the memory system that data memory accesses from a specified range
// of addresses are likely to occur in the near future. The memory system can
// respond by taking actions that are expected to speed up the memory accesses
// when they do occur, such as preloading the locations within the specified
// address ranges into one or more caches.

impdef func Hint_RangePrefetch(address : bits(64), length : integer, stride : integer,
                               count : integer, reuse : integer, operation : bits(6))
begin
    return;
end;
```

J1.4.3.282 IsContiguousSVEAccess

```
// IsContiguousSVEAccess()
// =======================
// Return TRUE if memory access is contiguous load/stores in an SVE mode.

func IsContiguousSVEAccess(accdesc : AccessDescriptor) => boolean
begin
    return accdesc.acctype == AccessType_SVE && accdesc.contiguous;
end;
```

J1.4.3.283 IsRelaxedWatchpointAccess

```
// IsRelaxedWatchpointAccess()
// ===========================
// Return TRUE if memory access is one of -
// - SIMD&FP load/store instruction when the PE is in Streaming SVE mode
// - SVE contiguous vector load/store instruction.
// - SME load/store instruction

func IsRelaxedWatchpointAccess(accdesc : AccessDescriptor) => boolean
begin
    return (IsContiguousSVEAccess(accdesc) ||
            IsSMEAccess(accdesc) ||
            (IsSIMDFPAccess(accdesc) && InStreamingMode()));
end;
```

J1.4.3.284 IsSIMDFPAccess

```
// IsSIMDFPAccess()
// ================
// Return TRUE if access is SIMD&FP.

func IsSIMDFPAccess(accdesc : AccessDescriptor) => boolean
begin
    return accdesc.acctype == AccessType_ASIMD;
end;
```

J1.4.3.285 IsSMEAccess

```
// IsSMEAccess()
// =============
// Return TRUE if access is of SME load/stores.

func IsSMEAccess(accdesc : AccessDescriptor) => boolean
begin
    return accdesc.acctype == AccessType_SME;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.286 IsWatchpointableAccess

```
// IsWatchpointableAccess()
// ========================
// Return TRUE if access should be checked for watchpoints.

func IsWatchpointableAccess(accdesc : AccessDescriptor) => boolean
begin
    return (!(accdesc.acctype IN {AccessType_IFETCH,
                                  AccessType_TTW,
                                  AccessType_DC,
                                  AccessType_IC,
                                  AccessType_SPE,
                                  AccessType_TRBE,
                                  AccessType_AT}) ||
            (accdesc.acctype == AccessType_DC &&
             accdesc.cacheop == CacheOp_Invalidate &&
             (!UsingAArch32() || (ImpDefBool("DCIMVAC generates watchpoint")))));
end;
```

J1.4.3.287 MBMaintenanceScope

```
// MBMaintenanceScope
// ==================
// Memory barrier scope.

type MBMaintenanceScope of enumeration {MBMaintenanceScope_Nonshareable,
                                        MBMaintenanceScope_InnerShareable,
                                        MBMaintenanceScope_OuterShareable,
                                        MBMaintenanceScope_None};
```

J1.4.3.288 MBReqTypes

```
// MBReqTypes
// ==========
// Memory barrier read/write.

type MBReqTypes of enumeration {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

J1.4.3.289 MemAtomicOp

```
// MemAtomicOp
// ===========
// Atomic data processing instruction types.

type MemAtomicOp of enumeration {
    MemAtomicOp_GCSSS1,
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS,
    MemAtomicOp_FPADD,
    MemAtomicOp_FPMAX,
    MemAtomicOp_FPMIN,
    MemAtomicOp_FPMAXNM,
    MemAtomicOp_FPMINNM,
    MemAtomicOp_BFADD,
    MemAtomicOp_BFMAX,
    MemAtomicOp_BFMIN,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    MemAtomicOp_BFMAXNM,
    MemAtomicOp_BFMINNM
};
```

J1.4.3.290 MemAttrHints

```
// MemAttrHints
// ============
// Attributes and hints for Normal memory.

type MemAttrHints of record {
    attrs : bits(2),  // See MemAttr_*, Cacheability attributes
    hints : bits(2),  // See MemHint_*, Allocation hints
    transient : boolean
};
```

J1.4.3.291 MemOp

```
// MemOp
// =====
// Memory access instruction types.

type MemOp of enumeration {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

J1.4.3.292 MemType

```
// MemType
// =======
// Basic memory types.

type MemType of enumeration {MemType_Normal, MemType_Device};
```

J1.4.3.293 Memory

```
// Memory Tag type
// ===============

type MemTagType of enumeration {
    MemTag_AllocationTagged,
    MemTag_CanonicallyTagged,
    MemTag_Untagged
};
```

J1.4.3.294 MemoryAttributes

```
// MemoryAttributes
// ================
// Memory attributes descriptor

type MemoryAttributes of record {
    memtype : MemType,
    device : DeviceType,          // For Device memory types
    inner : MemAttrHints,         // Inner hints and attributes
    outer : MemAttrHints,         // Outer hints and attributes
    shareability : Shareability, // Shareability attribute
    tags : MemTagType,            // MTE tag type for this memory.
    notagaccess : boolean,        // Allocation Tag access permission
    xs : bit                      // XS attribute
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.295 NewAccDesc

```
// NewAccDesc()
// ============
// Create a new AccessDescriptor with initialized fields

func NewAccDesc(acctype : AccessType) => AccessDescriptor
begin
    let mpaminfo : MPAMinfo = GenMPAMCurEL(acctype);
    return NewAccDesc(acctype, mpaminfo);
end;

func NewAccDesc(acctype : AccessType, mpam : MPAMinfo) => AccessDescriptor
begin
    var accdesc : AccessDescriptor;

    accdesc.acctype             = acctype;
    accdesc.el                  = PSTATE.EL;
    accdesc.ss                  = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc               = FALSE;
    accdesc.acqpc               = FALSE;
    accdesc.relsc               = FALSE;
    accdesc.limitedordered      = FALSE;
    accdesc.exclusive           = FALSE;
    accdesc.rcw                 = FALSE;
    accdesc.rcws                = FALSE;
    accdesc.atomicop            = FALSE;
    accdesc.nontemporal         = FALSE;
    accdesc.read                = FALSE;
    accdesc.write               = FALSE;
    accdesc.pan                 = FALSE;
    accdesc.nonfault            = FALSE;
    accdesc.firstfault          = FALSE;
    accdesc.first               = FALSE;
    accdesc.contiguous          = FALSE;
    accdesc.predicated          = FALSE;
    accdesc.streamingsve        = FALSE;
    accdesc.ls64                = FALSE;
    accdesc.withstatus          = FALSE;
    accdesc.mops                = FALSE;
    accdesc.a32lsmd             = FALSE;
    accdesc.tagchecked          = FALSE;
    accdesc.tagaccess           = FALSE;
    accdesc.stzgm               = FALSE;
    accdesc.mpam                = mpam;
    accdesc.Rs                  = -1;
    accdesc.Rs2                 = -1;
    accdesc.Rt                  = -1;
    accdesc.Rt2                 = -1;
    accdesc.ispair              = FALSE;
    accdesc.highestaddressfirst = FALSE;
    accdesc.lowestaddress       = TRUE;

    return accdesc;
end;
```

J1.4.3.296 PASpace

```
// PASpace
// =======
// Physical address spaces

type PASpace of enumeration {
    PAS_Root,
    PAS_SystemAgent,
    PAS_NonSecureProtected,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PAS_NA6,                // Reserved
    PAS_NA7,                // Reserved
    PAS_Realm,
    PAS_Secure,
    PAS_NonSecure
};
```

J1.4.3.297 Permissions

```
// Permissions
// ===========
// Access Control bits in translation table descriptors

type Permissions of record {
    ap_table  : bits(2),  // Stage 1 hierarchical access permissions
    xn_table  : bit,      // Stage 1 hierarchical execute-never for single EL regimes
    pxn_table : bit,      // Stage 1 hierarchical privileged execute-never
    uxn_table : bit,      // Stage 1 hierarchical unprivileged execute-never
    ap  : bits(3),        // Stage 1 access permissions
    xn  : bit,            // Stage 1 execute-never for single EL regimes
    uxn : bit,            // Stage 1 unprivileged execute-never
    pxn : bit,            // Stage 1 privileged execute-never
    ppi : bits(4),        // Stage 1 privileged indirect permissions
    upi : bits(4),        // Stage 1 unprivileged indirect permissions
    ndirty : bit,         // Stage 1 dirty state for indirect permissions scheme
    s2pi    : bits(4),    // Stage 2 indirect permissions
    s2dirty : bit,        // Stage 2 dirty state
    po_index : bits(4),   // Stage 1 overlay permissions index
    s2po_index : bits(4), // Stage 2 overlay permissions index
    s2ap : bits(2),       // Stage 2 access permissions
    s2tag_na : bit,       // Stage 2 tag access
    s2xnx : bit,          // Stage 2 extended execute-never
    dbm : bit,            // Dirty bit management
    s2xn : bit            // Stage 2 execute-never
};
```

J1.4.3.298 PhysMemRead

```
// PhysMemRead()
// =============
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

impdef func PhysMemRead{size : integer{8, 16, 32, 64, 128, 256, 512}}(
                                                    desc : AddressDescriptor,
                                                    accdesc : AccessDescriptor
                                              ) => (PhysMemRetStatus, bits(size))
begin
    return (ARBITRARY : PhysMemRetStatus, ARBITRARY : bits(size));
end;
```

J1.4.3.299 PhysMemRetStatus

```
// PhysMemRetStatus
// ================
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus of record {
    statuscode : Fault,       // Fault Status
    extflag : bit,            // IMPLEMENTATION DEFINED syndrome for External aborts
    merrorstate : ErrorState, // Optional error state returned on a physical memory access
    store64bstatus : bits(64) // Status of 64B store
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.300 PhysMemWrite

```
// PhysMemWrite()
// ==============
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

impdef func PhysMemWrite{size : integer{8, 16, 32, 64, 128, 256, 512}}(desc : AddressDescriptor,
                                                                        accdesc : AccessDescriptor,
                                                                value : bits(size)) => PhysMemRetStatus
begin
    return ARBITRARY : PhysMemRetStatus;
end;
```

J1.4.3.301 PrefetchHint

```
// PrefetchHint
// ============
// Prefetch hint types.

type PrefetchHint of enumeration {
    Prefetch_READ,
    Prefetch_WRITE,
    Prefetch_IR,
    Prefetch_EXEC
};
```

J1.4.3.302 S1AccessControls

```
// S1AccessControls
// ================
// Effective access controls defined by stage 1 translation

type S1AccessControls of record {
    r : bit,                // Stage 1 base read permission
    w : bit,                // Stage 1 base write permission
    x : bit,                // Stage 1 base execute permission
    gcs : bit,              // Stage 1 GCS permission
    overlay : boolean,      // Stage 1 FEAT_S1POE overlay applies
    or : bit,               // Stage 1 FEAT_S1POE overlay read permission
    ow : bit,               // Stage 1 FEAT_S1POE overlay write permission
    ox : bit,               // Stage 1 FEAT_S1POE overlay execute permission
    wxn : bit               // Stage 1 write permission implies execute-never

};
```

J1.4.3.303 S2AccessControls

```
// S2AccessControls
// ================
// Effective access controls defined by stage 2 translation

type S2AccessControls of record {
    r : bit,                // Stage 2 read permission.
    w : bit,                // Stage 2 write permission.
    x : bit,                // Stage 2 execute permission.
    r_rcw : bit,            // Stage 2 Read perms for RCW instruction.
    w_rcw : bit,            // Stage 2 Write perms for RCW instruction.
    r_mmu : bit,            // Stage 2 Read perms for TTW data.
    w_mmu : bit,            // Stage 2 Write perms for TTW data.
    toplevel0 : bit,        // IPA as top level table for TTBR0_EL1.
    toplevel1 : bit,        // IPA as top level table for TTBR1_EL1.
    overlay : boolean,      // Overlay enable
    or : bit,               // Stage 2 overlay read permission.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    ow : bit,                // Stage 2 overlay write permission.
    ox : bit,                // Stage 2 overlay execute permission.
    or_rcw : bit,            // Stage 2 overlay Read perms for RCW instruction.
    ow_rcw : bit,            // Stage 2 overlay Write perms for RCW instruction.
    or_mmu : bit,            // Stage 2 overlay Read perms for TTW data.
    ow_mmu : bit,            // Stage 2 overlay Write perms for TTW data.
};
```

J1.4.3.304 Shareability

```
// Shareability
// ============

type Shareability of enumeration {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

J1.4.3.305 SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// ===================================

impdef func SpeculativeStoreBypassBarrierToPA()
begin
    // Since there is no speculation in ASL model, this is a NOP.
    return;
end;
```

J1.4.3.306 SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// ===================================

impdef func SpeculativeStoreBypassBarrierToVA()
begin
    // Since there is no speculation in ASL model, this is a NOP.
    return;
end;
```

J1.4.3.307 Tag

```
// Tag Granule size
// ================

constant LOG2_TAG_GRANULE : integer{} = 4;
constant TAG_GRANULE : integer{} = 1 << LOG2_TAG_GRANULE;
```

J1.4.3.308 VARange

```
// VARange
// =======
// Virtual address ranges

type VARange of enumeration {
    VARange_LOWER,
    VARange_UPPER
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.309 AltPARTIDSpace

```
// AltPARTIDSpace()
// ================
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

func AltPARTIDSpace(el : bits(2), security : SecurityState,
                    primaryPIDSpace : PARTIDSpaceType) => PARTIDSpaceType
begin
    case security of
        when SS_NonSecure =>
            assert el != EL3;
            return primaryPIDSpace;
        when SS_Secure =>
            assert el != EL3;
            if primaryPIDSpace == PIDSpace_NonSecure then
                return primaryPIDSpace;
            end;
            return AltPIDSecure(el, primaryPIDSpace);
        when SS_Root =>
            assert el == EL3;
            if MPAM3_EL3().ALTSP_EL3 == '1' then
                if MPAM3_EL3().RT_ALTSP_NS == '1' then
                    return PIDSpace_NonSecure;
                else
                    return PIDSpace_Secure;
                end;
            else
                return primaryPIDSpace;
            end;
        when SS_Realm =>
            assert el != EL3;
            return AltPIDRealm(el, primaryPIDSpace);
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.310 AltPIDRealm

```
// AltPIDRealm()
// =============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDSpace.

func AltPIDRealm(el : bits(2), primaryPIDSpace : PARTIDSpaceType) => PARTIDSpaceType
begin
    var PIDSpace : PARTIDSpaceType = primaryPIDSpace;
    case el of
        when EL0 =>
            if ELIsInHost(EL0) then
                if !UsePrimarySpaceEL2() then
                    PIDSpace = PIDSpace_NonSecure;
                end;
            elsif !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace_NonSecure;
            end;
        when EL1 =>
            if !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace_NonSecure;
            end;
        when EL2 =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace_NonSecure;
            end;
        otherwise =>
            unreachable;
    end;
    return PIDSpace;
end;
```

J1.4.3.311 AltPIDSecure

```
// AltPIDSecure()
// ==============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDSpace.

func AltPIDSecure(el : bits(2), primaryPIDSpace : PARTIDSpaceType) => PARTIDSpaceType
begin
    var PIDSpace : PARTIDSpaceType = primaryPIDSpace;
    case el of
        when EL0 =>
            if EL2Enabled() then
                if ELIsInHost(EL0) then
                    if !UsePrimarySpaceEL2() then
                        PIDSpace = PIDSpace_NonSecure;
                    end;
                elsif !UsePrimarySpaceEL10() then
                    PIDSpace = PIDSpace_NonSecure;
                end;
            elsif MPAM3_EL3().ALTSP_HEN == '0' && MPAM3_EL3().ALTSP_HFC == '1' then
                PIDSpace = PIDSpace_NonSecure;
            end;
        when EL1 =>
            if EL2Enabled() then
                if !UsePrimarySpaceEL10() then
                    PIDSpace = PIDSpace_NonSecure;
                end;
            elsif MPAM3_EL3().ALTSP_HEN == '0' && MPAM3_EL3().ALTSP_HFC == '1' then
                PIDSpace = PIDSpace_NonSecure;
            end;
        when EL2 =>
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace_NonSecure;
            end;
        otherwise =>
            unreachable;
    end;
    return PIDSpace;
end;
```

J1.4.3.312 DefaultMPAMInfo

```
// DefaultMPAMInfo()
// =================
// Returns default MPAM info.  The partidspace argument sets
// the PARTID space of the default MPAM information returned.

func DefaultMPAMInfo(partidspace : PARTIDSpaceType) => MPAMinfo
begin
    var defaultinfo : MPAMinfo;
    defaultinfo.mpam_sp = partidspace;
    defaultinfo.partid  = DEFAULT_PARTID;
    defaultinfo.pmg     = DEFAULT_PMG;
    return defaultinfo;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.4.3.313 GenMPAM

```
// GenMPAM()
// =========
// Returns MPAMinfo for Exception level el.
// If mpamdata.sm is TRUE returns MPAM information using MPAMSM_EL1.
// If mpamdata.trb is TRUE returns MPAM information using TRBMPAM_EL1.
// Otherwise returns MPAM information using MPAMn_ELx.

func GenMPAM(el_in : bits(2), mpamdata : MPAMdata, pspace : PARTIDSpaceType) => MPAMinfo
begin
    var returninfo : MPAMinfo;
    var partidel : PARTIDType;
    var perr : boolean;
    var el : bits(2) = el_in;

    // Check if the guest OS application is locked by the EL2 hypervisor to
    // only use the EL1 virtual machine's PARTIDs.
    if (el == EL0 && EL2Enabled() && MPAMHCR_EL2().GSTAPP_PLK == '1' &&
          HCR_EL2().TGE == '0') then
        el = EL1;
    end;

    (partidel, perr)      = GenPARTID(el, mpamdata, pspace);
    let groupel : PMGType = GenPMG(el, mpamdata, perr, pspace);

    returninfo.mpam_sp = pspace;
    returninfo.partid  = partidel;
    returninfo.pmg     = groupel;
    return returninfo;
end;
```

J1.4.3.314 GenMPAMAtEL

```
// GenMPAMAtEL()
// =============
// Returns MPAMinfo for the specified EL.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32_  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

func GenMPAMAtEL(acctype : AccessType, el : bits(2)) => MPAMinfo
begin
    var mpamEL : bits(2);
    var pspace : PARTIDSpaceType;
    let security : SecurityState = SecurityStateAtEL(el);
    var mpamdata : MPAMdata      = GenNewMPAMData();
    pspace = PARTIDSpaceFromSS(security);
    if pspace == PIDSpace_NonSecure && !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    end;
    mpamEL = if acctype == AccessType_NV2 then EL2 else el;
    case acctype of
        when AccessType_IFETCH, AccessType_IC =>
            mpamdata.in_d = TRUE;
        when AccessType_SME =>
            mpamdata.sm = (ImpDefBool("Shared SMCU") ||
                           ImpDefBool("MPAMSM_EL1 label precedence"));
        when AccessType_FP, AccessType_ASIMD, AccessType_SVE =>
            mpamdata.sm = (IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
                           (ImpDefBool("Shared SMCU") ||
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                            ImpDefBool("MPAMSM_EL1 label precedence")));
        when AccessType_TRBE =>
            if !SelfHostedTraceEnabled() then
                if !IsFeatureImplemented(FEAT_TRBE_MPAM) || TRBMPAM_EL1().EN == '0' then
                    return DefaultMPAMInfo(pspace);
                else
                    mpamdata.trb = TRUE;
                end;
            end;
            if mpamdata.trb then
                var ss : SecurityState;
                case TRBMPAM_EL1().MPAM_SP of
                    when '00' => ss = SS_Secure;
                    when '01' => ss = SS_NonSecure;
                    when '10' => ss = SS_Root;
                    when '11' => ss = SS_Realm;
                end;
                pspace = PARTIDSpaceFromSS(ss);
            end;
        otherwise =>
            // Other access types are DATA accesses
            mpamdata.in_d = FALSE;
    end;
    if IsFeatureImplemented(FEAT_RME) && MPAMIDR_EL1().HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDSpace(mpamEL, security, pspace);
    end;
    if IsFeatureImplemented(FEAT_MPAMv0p1) && MPAMIDR_EL1().HAS_FORCE_NS == '1' then
        if MPAM3_EL3().FORCE_NS == '1' && security == SS_Secure then
            pspace = PIDSpace_NonSecure;
        end;
    end;
    if ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p1)) &&
          MPAMIDR_EL1().HAS_SDEFLT == '1') then
        if MPAM3_EL3().SDEFLT == '1' && security == SS_Secure then
            return DefaultMPAMInfo(pspace);
        end;
    end;
    if !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    else
        return GenMPAM(mpamEL, mpamdata, pspace);
    end;
end;
```

J1.4.3.315 GenMPAMCurEL

```
// GenMPAMCurEL()
// ==============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32_  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

func GenMPAMCurEL(acctype : AccessType) => MPAMinfo
begin
    return GenMPAMAtEL(acctype, PSTATE.EL);
end;
```

J1.4.3.316 GenNewMPAMData

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// GenNewMPAMData()
// ================

func GenNewMPAMData() => MPAMdata
begin
    var mpamdata : MPAMdata;
    mpamdata.in_d   = FALSE;
    mpamdata.sm  = FALSE;
    mpamdata.trb = FALSE;
    return mpamdata;
end;
```

J1.4.3.317 GenPARTID

```
// GenPARTID()
// ===========
// Returns physical PARTID and error boolean for Exception level el.
// If mpamdata.sm is TRUE then PARTID is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PARTID is from TRBMPAM_EL1.
// Otherwise, the PARTID is from MPAMn_ELx.

func GenPARTID(el : bits(2), mpamdata : MPAMdata, pspace : PARTIDSpaceType) => (PARTIDType, boolean)
begin
    let partidel : PARTIDType = GetMPAM_PARTID(el, mpamdata);
    let partid_max : PARTIDType = (if mpamdata.trb then TRBDEVID1().PARTID_MAX
                                   else MPAMIDR_EL1().PARTID_MAX);
    if UInt(partidel) > UInt(partid_max) then
        return (DEFAULT_PARTID, TRUE);
    end;
    if MPAMIsVirtual(el, mpamdata) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
    end;
end;
```

J1.4.3.318 GenPMG

```
// GenPMG()
// ========
// Returns PMG for Exception level el.
// if mpamdata.sm is TRUE then PMG is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PMG is from TRBMPAM_EL1.
// Otherwise, PMG is from MPAMn_ELx.
// If PARTID generation (GenPARTID) encountered an error, GenPMG() should be
// called with partid_err as TRUE.

func GenPMG(el : bits(2), mpamdata : MPAMdata, partid_err : boolean,
            pspace : PARTIDSpaceType) => PMGType
begin
    if partid_err && ConstrainUnpredictableBool(Unpredictable_USE_DEFAULT_PMG) then
        return DEFAULT_PMG;
    end;
    let groupel : PMGType = GetMPAM_PMG(el, mpamdata);
    let pmg_max : integer = (if mpamdata.trb then UInt(TRBDEVID1().PMG_MAX)
                             else UInt(MPAMIDR_EL1().PMG_MAX));
    if UInt(groupel) <= pmg_max then
        return groupel;
    end;
    return DEFAULT_PMG;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.319 GetMPAM_PARTID

```
// GetMPAM_PARTID()
// ================
// Returns a PARTID
// If mpamdata.sm is TRUE then PARTID is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PARTID is from TRBMPAM_EL1.
// Otherwise, the PARTID is from MPAMn_ELx.

func GetMPAM_PARTID(MPAMn : bits(2), mpamdata : MPAMdata) => PARTIDType
begin
    if mpamdata.sm then
        return MPAMSM_EL1().PARTID_D;
    end;

    if mpamdata.trb then
        return TRBMPAM_EL1().PARTID;
    end;

    if mpamdata.in_d then
        case MPAMn of
            when '11' => return MPAM3_EL3().PARTID_I;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PARTID_I else DEFAULT_PARTID);
            when '01' => return MPAM1_EL1().PARTID_I;
            when '00' => return MPAM0_EL1().PARTID_I;
        end;
    else
        case MPAMn of
            when '11' => return MPAM3_EL3().PARTID_D;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PARTID_D else DEFAULT_PARTID);
            when '01' => return MPAM1_EL1().PARTID_D;
            when '00' => return MPAM0_EL1().PARTID_D;
        end;
    end;
end;
```

J1.4.3.320 GetMPAM_PMG

```
// GetMPAM_PMG()
// =============
// Returns a PMG.
// if mpamdata.sm is TRUE then PMG is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PMG is from TRBMPAM_EL1.
// Otherwise, PMG is from MPAMn_ELx.

func GetMPAM_PMG(MPAMn : bits(2), mpamdata : MPAMdata) => PMGType
begin
    if mpamdata.sm then
        return MPAMSM_EL1().PMG_D;
    end;

    if mpamdata.trb then
        return TRBMPAM_EL1().PMG;
    end;

    if mpamdata.in_d then
        case MPAMn of
            when '11' => return MPAM3_EL3().PMG_I;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PMG_I else DEFAULT_PMG);
            when '01' => return MPAM1_EL1().PMG_I;
            when '00' => return MPAM0_EL1().PMG_I;
        end;
    else
        case MPAMn of
            when '11' => return MPAM3_EL3().PMG_D;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PMG_D else DEFAULT_PMG);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '01' => return MPAM1_EL1().PMG_D;
            when '00' => return MPAM0_EL1().PMG_D;
        end;
    end;
end;
```

J1.4.3.321 MAP_vPARTID

```
// MAP_vPARTID()
// =============
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

func MAP_vPARTID(vpartid : PARTIDType) => (PARTIDType, boolean)
begin
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    var ret : PARTIDType;
    var err : boolean;
    var virt : integer    = UInt(vpartid);
    let vpmrmax : integer = UInt(MPAMIDR_EL1().VPMR_MAX);

    // vpartid_max is largest vpartid supported
    let vpartid_max : integer = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);
    end;

    // Check for valid mapping entry.
    if MPAMVPMV_EL2()[virt] == '1' then
        // vpartid has a valid mapping, access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Check for default virtual PARTID valid
    elsif MPAMVPMV_EL2()[0] == '1' then
        // Use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2()[0 +: 16];
        err = FALSE;

    // Neither is valid, use default physical PARTID.
    else
        ret = DEFAULT_PARTID;
        err = TRUE;
    end;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    let partid_max : integer = UInt(MPAMIDR_EL1().PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DEFAULT_PARTID;
        err = TRUE;
    end;
    return (ret, err);
end;
```

J1.4.3.322 MPAM

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// MPAM Types
// ==========

type PARTIDType of bits(16);
type PMGType of bits(8);

type PARTIDSpaceType of enumeration {
    PIDSpace_Secure,
    PIDSpace_Root,
    PIDSpace_Realm,
    PIDSpace_NonSecure
};

type MPAMinfo of record {
     mpam_sp : PARTIDSpaceType,
     partid : PARTIDType,
     pmg : PMGType
};
constant DEFAULT_PARTID : PARTIDType = 0[15:0];
constant DEFAULT_PMG : PMGType = 0[7:0];

type MPAMdata of record {
    in_d : boolean,       // TRUE for instruction accesses
    sm : boolean,         // TRUE for SME, SVE, SIMD&FP access, and SVE prefetch
                          // instructions, when the PE is in Streaming mode
    trb : boolean         // TRUE for TRBE accesses using External mode when TRBMPAM_EL1.EN is 0b1
};
```

J1.4.3.323 MPAMIsEnabled

```
// MPAMIsEnabled()
// ===============
// Returns TRUE if MPAM is enabled.

func MPAMIsEnabled() => boolean
begin
    let el : bits(2) = HighestEL();
    case el of
        when EL3 => return MPAM3_EL3().MPAMEN == '1';
        when EL2 => return MPAM2_EL2().MPAMEN == '1';
        when EL1 => return MPAM1_EL1().MPAMEN == '1';
    end;
end;
```

J1.4.3.324 MPAMIsVirtual

```
// MPAMIsVirtual()
// ===============
// Returns TRUE if MPAM is configured to be virtual at the given el.

func MPAMIsVirtual(el : bits(2), mpamdata : MPAMdata) => boolean
begin
    if mpamdata.trb then
        return FALSE;
    end;

    return (MPAMIDR_EL1().HAS_HCR == '1' && EL2Enabled() &&
            ((el == EL0 && MPAMHCR_EL2().EL0_VPMEN == '1' && !ELIsInHost(EL0)) ||
             (el == EL1 && MPAMHCR_EL2().EL1_VPMEN == '1')));
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.325 PARTIDSpaceFromSS

```
// PARTIDSpaceFromSS()
// ===================
// Returns the primary PARTID space from the Security State.

func PARTIDSpaceFromSS(security : SecurityState) => PARTIDSpaceType
begin
    case security of
        when SS_NonSecure =>
            return PIDSpace_NonSecure;
        when SS_Root =>
            return PIDSpace_Root;
        when SS_Realm =>
            return PIDSpace_Realm;
        when SS_Secure =>
            return PIDSpace_Secure;
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.326 UsePrimarySpaceEL10

```
// UsePrimarySpaceEL10()
// =====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

func UsePrimarySpaceEL10() => boolean
begin
    if MPAM3_EL3().ALTSP_HEN == '0' then
        return MPAM3_EL3().ALTSP_HFC == '0';
    end;
    return !MPAMIsEnabled() || !EL2Enabled() || MPAM2_EL2().ALTSP_HFC == '0';
end;
```

J1.4.3.327 UsePrimarySpaceEL2

```
// UsePrimarySpaceEL2()
// ====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

func UsePrimarySpaceEL2() => boolean
begin
    if MPAM3_EL3().ALTSP_HEN == '0' then
        return MPAM3_EL3().ALTSP_HFC == '0';
    end;
    return !MPAMIsEnabled() || MPAM2_EL2().ALTSP_EL2 == '0';
end;
```

J1.4.3.328 mapvpmw

```
// mapvpmw()
// =========
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

func mapvpmw(vpartid : integer) => PARTIDType
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var vpmw : bits(64);
    let wd : integer = vpartid DIVRM 4;
    case wd of
        when 0    => vpmw = MPAMVPM0_EL2();
        when 1    => vpmw = MPAMVPM1_EL2();
        when 2    => vpmw = MPAMVPM2_EL2();
        when 3    => vpmw = MPAMVPM3_EL2();
        when 4    => vpmw = MPAMVPM4_EL2();
        when 5    => vpmw = MPAMVPM5_EL2();
        when 6    => vpmw = MPAMVPM6_EL2();
        when 7    => vpmw = MPAMVPM7_EL2();
        otherwise => vpmw = Zeros{64};
    end;
    // vpme_lsb selects LSB of field within register
    let vpme_lsb : integer = (vpartid MOD 4) * 16;
    return vpmw[vpme_lsb +: 16];
end;
```

J1.4.3.329 ASID

```
// ASID
// ====
// Effective ASID.

func ASID() => bits(NUM_ASIDBITS)
begin
    if ELIsInHost(EL0) then
        if TCR_EL2().A1 == '1' then
            return TTBR1_EL2().ASID;
        else
            return TTBR0_EL2().ASID;
        end;
    end;
    if !ELUsingAArch32(EL1) then
        if TCR_EL1().A1 == '1' then
            return TTBR1_EL1().ASID;
        else
            return TTBR0_EL1().ASID;
        end;
    else
        if TTBCR().EAE == '0' then
            return ZeroExtend{NUM_ASIDBITS}(CONTEXTIDR().ASID);
        else
            if TTBCR().A1 == '1' then
                return ZeroExtend{NUM_ASIDBITS}(TTBR1().ASID);
            else
                return ZeroExtend{NUM_ASIDBITS}(TTBR0().ASID);
            end;
        end;
    end;
end;
```

J1.4.3.330 ExecutionCntxt

```
// ExecutionCntxt
// ===============
// Context information for prediction restriction operation.

type ExecutionCntxt of record {
    is_vmid_valid : boolean,        // is vmid valid for current context
    all_vmid      : boolean,        // should the operation be applied for all vmids
    vmid          : bits(NUM_VMIDBITS), // if all_vmid = FALSE, vmid to which operation is applied
    is_asid_valid : boolean,        // is asid valid for current context
    all_asid      : boolean,        // should the operation be applied for all asids
    asid          : bits(NUM_ASIDBITS), // if all_asid = FALSE, ASID to which operation is applied
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    target_el     : bits(2),        // target EL at which operation is performed
    security      : SecurityState,
    restriction   : RestrictType    // type of restriction operation
};
```

J1.4.3.331 RESTRICT_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// ======================
// Clear all speculated values.

impdef func RESTRICT_PREDICTIONS(c : ExecutionCntxt)
begin
    return;
end;
```

J1.4.3.332 RestrictType

```
// RestrictType
// ============
// Type of restriction on speculation.

type RestrictType of enumeration {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch,
    RestrictType_Other              // Any other trained speculation mechanisms than those above
};
```

J1.4.3.333 TargetSecurityState

```
// TargetSecurityState()
// =====================
// Decode the target security state for the prediction context.

func TargetSecurityState(NS : bit, NSE : bit) => SecurityState
begin
    let curr_ss : SecurityState =  SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' => return SS_Secure;
            when '1' => return SS_NonSecure;
        end;
    elsif IsFeatureImplemented(FEAT_RME) then
        if curr_ss == SS_Root then
            case NSE::NS of
                when '00' => return SS_Secure;
                when '01' => return SS_NonSecure;
                when '11' => return SS_Realm;
                when '10' => return SS_Root;
            end;
        elsif curr_ss == SS_Realm then
            return SS_Realm;
        end;
    end;
    unreachable;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.334 BranchTo

```
// BranchTo()
// ==========
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

func BranchTo{N}(target : bits(N), branch_type : BranchType, branch_conditional : boolean)
begin
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend{64}(target);
    else
        assert N == 64 && !UsingAArch32();
        let target_vaddress : bits(64) = AArch64_BranchAddr(target[63:0], PSTATE.EL);
        if (IsFeatureImplemented(FEAT_BRBE) &&
              branch_type IN {BranchType_DIR, BranchType_INDIR,
                              BranchType_DIRCALL, BranchType_INDCALL,
                              BranchType_RET}) then
            BRBEBranch(branch_type, branch_conditional, target_vaddress);
        end;
        let branch_taken : boolean = TRUE;

        if IsFeatureImplemented(FEAT_SPE) then
            SPEBranch{N}(target, branch_type, branch_conditional, branch_taken);
        end;

        _PC = target_vaddress;
    end;
    return;
end;
```

J1.4.3.335 BranchToAddr

```
// BranchToAddr()
// ==============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

func BranchToAddr{N}(target : bits(N), branch_type : BranchType)
begin
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend{64}(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target[63:0];
    end;
    return;
end;
```

J1.4.3.336 BranchType

```
// BranchType
// ==========
// Information associated with a change in control flow.

type BranchType of enumeration {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    BranchType_DIR,          // Direct branch
    BranchType_INDIR,        // Indirect branch
    BranchType_EXCEPTION,    // Exception entry
    BranchType_RESET,        // Reset
    BranchType_UNKNOWN};     // Other
```

J1.4.3.337 Branchtypetaken

```
// Branchtypetaken
// ===============
var Branchtypetaken : BranchType;
```

J1.4.3.338 ESize

```
// ESize
// =====

type ESize of integer{8, 16, 32, 64, 128};
```

J1.4.3.339 EffectiveFPCR

```
// EffectiveFPCR()
// ===============
// Returns the effective FPCR value

func EffectiveFPCR() => FPCR_Type
begin
    if UsingAArch32() then
        var fpcr : FPCR_Type = ZeroExtend{64}(FPSCR());
        fpcr[7:0]   = '00000000';
        fpcr[31:27] = '00000';
        return fpcr;
    end;
    return FPCR();
end;
```

J1.4.3.340 Hint_Branch

```
// Hint_Branch()
// =============
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

impdef func Hint_Branch(hint : BranchType)
begin
    Branchtypetaken = hint;
    return;
end;
```

J1.4.3.341 NextInstrAddr

```
// NextInstrAddr()
// ===============
// Return address of the sequentially next instruction.

impdef func NextInstrAddr{N}() => bits(N)
begin
    return Zeros{N};
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.342 ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// ============================
// Reset the External Debug registers in the Core power domain.

impdef func ResetExternalDebugRegisters(cold_reset : boolean)
begin
    return;
end;
```

J1.4.3.343 ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

func ThisInstrAddr{N}() => bits(N)
begin
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC[N-1:0];
end;
```

J1.4.3.344 UnimplementedIDRegister

```
// UnimplementedIDRegister()
// =========================
// Trap access to unimplemented encodings in the feature ID register space.

func UnimplementedIDRegister()
begin
    if IsFeatureImplemented(FEAT_IDST) then
        var target_el : bits(2) = PSTATE.EL;
        if PSTATE.EL == EL0 then
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        end;
        AArch64_SystemAccessTrap(target_el, 0x18);
    end;
    Undefined();
end;
```

J1.4.3.345 V

```
// V - accessor
// ============

accessor V{width : ESize}(n : integer) <=> value : bits(width)
begin
    // Read from SIMD&FP register with implicit slice of 8, 16
    // 32, 64 or 128 bits.
    getter
        assert n >= 0 && n <= 31;
        return _Z[[n]][width-1:0];
    end;

    // Write to SIMD&FP register with implicit extension from
    // 8, 16, 32, 64 or 128 bits.
    setter
        assert n >= 0 && n <= 31;
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _Z[[n]] = ZeroExtend{MAX_VL}(value);
        else
            let vlen : VecLen = if IsSVEEnabled(PSTATE.EL) then CurrentVL() else 128;
            _Z[[n]][vlen-1:0] = ZeroExtend{vlen}(value);
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

J1.4.3.346 Vpart

```
// Vpart - accessor
// ================

accessor Vpart{width : ESize}(n : integer, part : integer) <=> value : bits(width)
begin
    // Reads a 128-bit SIMD&FP register in up to two parts:
    //  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
    //  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
    //  value held in the register.
    getter
        assert n >= 0 && n <= 31;
        assert part IN {0, 1};
        if part == 0 then
            assert width < 128;
            return V{width}(n);
        else
            assert width IN {32,64};
            let vreg : bits(128) = V{}(n);
            return vreg[(width * 2)-1:width];
        end;
    end;

    setter
    // Writes a 128-bit SIMD&FP register in up to two parts:
    //  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
    //  part 1 inserts a 64-bit value into the top half of the register.
        assert n >= 0 && n <= 31;
        assert part IN {0, 1};
        if part == 0 then
            assert width < 128;
            V{width}(n) = value;
        else
            assert width == 64;
            let vreg : bits(64) = V{}(n);
            V{128}(n) = value[63:0] :: vreg;
        end;
    end;
end;
```

J1.4.3.347 _PC

```
// _PC - the program counter
// =========================

var _PC : bits(64);
```

J1.4.3.348 _R

```
// _R - the general-purpose register file array
// ============================================


var _R : array [[31]] of bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.349 SPSR_ELx

```
// SPSR_ELx - accessor
// ===================

accessor SPSR_ELx() <=> value :  bits(64)
begin
    getter
        var result : bits(64);
        case PSTATE.EL of
            when EL1  => result = SPSR_EL1()[63:0];
            when EL2  => result = SPSR_EL2()[63:0];
            when EL3  => result = SPSR_EL3()[63:0];
            otherwise => unreachable;
        end;
        return result;
    end;

    setter
        case PSTATE.EL of
            when EL1   => SPSR_EL1()[63:0] = value[63:0];
            when EL2   => SPSR_EL2()[63:0] = value[63:0];
            when EL3   => SPSR_EL3()[63:0] = value[63:0];
            otherwise  => unreachable;
        end;
    end;
end;
```

J1.4.3.350 SPSR_curr

```
// SPSR_curr - accessor
// ====================

accessor SPSR_curr() <=> value : bits(32)
begin
    getter
        var result : bits(32);
        case PSTATE.M of
            when M32_FIQ     => result = SPSR_fiq()[31:0];
            when M32_IRQ     => result = SPSR_irq()[31:0];
            when M32_Svc     => result = SPSR_svc()[31:0];
            when M32_Monitor => result = SPSR_mon()[31:0];
            when M32_Abort   => result = SPSR_abt()[31:0];
            when M32_Hyp     => result = SPSR_hyp()[31:0];
            when M32_Undef   => result = SPSR_und()[31:0];
            otherwise        => unreachable;
        end;
        return result;
    end;

    setter
        case PSTATE.M of
            when M32_FIQ     => SPSR_fiq()[31:0] = value[31:0];
            when M32_IRQ     => SPSR_irq()[31:0] = value[31:0];
            when M32_Svc     => SPSR_svc()[31:0] = value[31:0];
            when M32_Monitor => SPSR_mon()[31:0] = value[31:0];
            when M32_Abort   => SPSR_abt()[31:0] = value[31:0];
            when M32_Hyp     => SPSR_hyp()[31:0] = value[31:0];
            when M32_Undef   => SPSR_und()[31:0] = value[31:0];
            otherwise        => unreachable;
        end;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.351 AArch64_ChkFeat

```
// AArch64_ChkFeat()
// =================
// Indicates the status of some features

func AArch64_ChkFeat(feat_select : bits(64)) => bits(64)
begin
    var feat_en : bits(64) = Zeros{};
    feat_en[0] = if IsFeatureImplemented(FEAT_GCS) && GCSEnabled(PSTATE.EL) then '1' else '0';
    return feat_select AND NOT(feat_en);
end;
```

J1.4.3.352 AddressAdd

```
// AddressAdd()
// ============
// Add an address with an offset and return the result.
// If FEAT_CPA2 is implemented, the pointer arithmetic is checked.

func AddressAdd(base : bits(64), offset : integer, accdesc : AccessDescriptor) => bits(64)
begin
    return AddressAdd(base, offset[63:0], accdesc);
end;

func AddressAdd(base : bits(64), offset : bits(64), accdesc : AccessDescriptor) => bits(64)
begin
    var result : bits(64) = base + offset;
    result = PointerAddCheckAtEL(accdesc.el, result, base);
    return result;
end;
```

J1.4.3.353 AddressInNaturallyAlignedBlock

```
// AddressInNaturallyAlignedBlock()
// ================================
// Returns TRUE if the given addresses are in the same naturally aligned block, and FALSE otherwise.
// An address is in a naturally aligned block if it meets any of the below conditions:
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set to 0b0, or EDHSR is not
//   implemented or EDHSR.FnP is being set to 0b0 (as appropriate).
// * Is no larger than the smallest implemented translation granule if ESR_ELx.FnP, or EDHSR.FnP
//   (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or set of contiguous memory
//   accesses that triggered the watchpoint.

impdef func AddressInNaturallyAlignedBlock(address1 : bits(64), address2 : bits(64)) => boolean
begin
    // Returns TRUE if address in within a naturally aligned block of memory as the watchpointed
    // location, else FALSE.
    return FALSE;
end;
```

J1.4.3.354 AddressIncrement

```
// AddressIncrement()
// ==================
// Increment an address and return the result.
// If FEAT_CPA2 is implemented, the pointer arithmetic may be checked.

func AddressIncrement(base : bits(64), increment : integer, accdesc : AccessDescriptor) => bits(64)
begin
    return AddressIncrement(base, increment[63:0], accdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func AddressIncrement(base : bits(64), increment : bits(64), accdesc : AccessDescriptor) => bits(64)
begin
    var result : bits(64) = base + increment;
    // Checking the Pointer Arithmetic on an increment is equivalent to checking the
    // bytes in a sequential access crossing the 0xXXFF_FFFF_FFFF_FFFF boundary.
    if ConstrainUnpredictableBool(Unpredictable_CPACHECK) then
        result = PointerAddCheckAtEL(accdesc.el, result, base);
    end;
    return result;
end;
```

J1.4.3.355 BranchTargetCheck

```
// BranchTargetCheck()
// ===================
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

func BranchTargetCheck()
begin
    assert IsFeatureImplemented(FEAT_BTI) && !UsingAArch32();

    // The branch target check considers the following state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if Halted() then
        return;
    elsif IsZero(PSTATE.BTYPE) then
        return;
    elsif InGuardedPage && !BTypeCompatible then
        let pc : bits(64) = ThisInstrAddr{}();
        AArch64_BranchTargetException(pc[51:0]);
    end;
end;
```

J1.4.3.356 ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE.

func ClearEventRegister()
begin
    EventRegister = '0';
    return;
end;
```

J1.4.3.357 ConditionHolds

```
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

func ConditionHolds(cond : bits(4)) => boolean
begin
    // Evaluate base condition.
    var result : boolean;
    case cond[3:1] of
        when '000' => result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' => result = (PSTATE.C == '1');                          // CS or CC
        when '010' => result = (PSTATE.N == '1');                          // MI or PL
        when '011' => result = (PSTATE.V == '1');                          // VS or VC
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '100' => result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' => result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' => result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' => result = TRUE;                                       // AL
    end;

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond[0] == '1' && cond != '1111' then
        result = !result;
    end;

    return result;
end;
```

J1.4.3.358 ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====================================

impdef func ConsumptionOfSpeculativeDataBarrier()
begin
    return;
end;
```

J1.4.3.359 CurrentInstrSet

```
// CurrentInstrSet()
// =================

readonly func CurrentInstrSet() => InstrSet
begin
    var result : InstrSet;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    end;
    return result;
end;
```

J1.4.3.360 CurrentSecurityState

```
// CurrentSecurityState()
// ======================
// Returns the effective security state at the exception level based off current settings.

readonly func CurrentSecurityState() => SecurityState
begin
    return SecurityStateAtEL(PSTATE.EL);
end;
```

J1.4.3.361 DSBAlias

```
// DSBAlias
// ========
// Aliases of DSB.

type DSBAlias of enumeration {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.362 EL0

```
// EL0-3
// =====
// PSTATE.EL Exception level bits.

constant EL3 : bits(2) = '11';
constant EL2 : bits(2) = '10';
constant EL1 : bits(2) = '01';
constant EL0 : bits(2) = '00';
```

J1.4.3.363 EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

readonly func EL2Enabled() => boolean
begin
    var scr_ns : bit;
    if HaveEL(EL3) then
        if ELUsingAArch32(EL3) then
            scr_ns = SCR().NS;
        else
            scr_ns = SCR_EL3().NS;
        end;
    end;
    return HaveEL(EL2) && (!HaveEL(EL3) || scr_ns == '1' || IsSecureEL2Enabled());
end;
```

J1.4.3.364 EL3SDDUndef

```
// EL3SDDUndef()
// =============
// Returns TRUE if in Debug state and EDSCR.SDD is set.

func EL3SDDUndef() => boolean
begin
    if Halted() && EDSCR().SDD == '1' then
        assert (PSTATE.EL != EL3  &&
                  (IsFeatureImplemented(FEAT_RME) || CurrentSecurityState() != SS_Secure));
        return TRUE;
    else
        return FALSE;
    end;
end;
```

J1.4.3.365 EL3SDDUndefPriority

```
// EL3SDDUndefPriority()
// =====================
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has priority over other traps.
// The IMPLEMENTATION DEFINED priority may be different for each case.

func EL3SDDUndefPriority() => boolean
begin
    return EL3SDDUndef() && ImpDefBool("EL3 trap priority when SDD == '1'");
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.366 ELFromM32

```
// ELFromM32()
// ===========

func ELFromM32(mode : bits(5)) => (boolean,bits(2))
begin
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode[4:0]' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    var el : bits(2);
    // Check for modes that are not valid for this implementation
    var valid : boolean = !BadMode(mode);
    let effective_nse_ns : bits(2) = EffectiveSCR_EL3_NSE() :: EffectiveSCR_EL3_NS();

    case mode of
        when M32_Monitor =>
            el = EL3;
        when M32_Hyp =>
            el = EL2;
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System =>
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR().NS == '0' then EL3 else EL1);
        when M32_User =>
            el = EL0;
        otherwise =>
            valid = FALSE;           // Passed an illegal mode value
    end;

    if valid && el == EL2 && HaveEL(EL3) then
        if ELUsingAArch32(EL3) && SCR().NS == '0' then
            valid = FALSE; // EL2 only valid in Non-secure state in AArch32
        elsif !ELUsingAArch32(EL3) && SCR_EL3().NS == '0' then
            valid = FALSE; // EL2 only valid in Non-secure state in AArch32
        end;
    elsif valid && IsFeatureImplemented(FEAT_RME) && effective_nse_ns == '10' then
        valid = FALSE;               // Illegal Exception Return from EL3 if SCR_EL3.[NSE,NS]
                                     // selects a reserved encoding
    end;

    if !valid then el = ARBITRARY : bits(2); end;
    return (valid, el);
end;
```

J1.4.3.367 ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr[4:0]' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

func ELFromSPSR{N}(spsr : bits(N)) => (boolean,bits(2))
begin
    var el : bits(2);
    var valid : boolean;
    if spsr[4] == '0' then      // AArch64 state
        el = spsr[3:2];
        let effective_nse_ns : bits(2) = EffectiveSCR_EL3_NSE() :: EffectiveSCR_EL3_NS();
        if !HaveAArch64() then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            valid = FALSE;      // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE;      // Exception level not implemented
        elsif spsr[1] == '1' then
            valid = FALSE;      // M[1] must be 0
        elsif el == EL0 && spsr[0] == '1' then
            valid = FALSE;      // for EL0, M[0] must be 0
        elsif IsFeatureImplemented(FEAT_RME) && el != EL3 && effective_nse_ns == '10' then
            valid = FALSE;      // Only EL3 valid in Root state
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && EffectiveSCR_EL3_NS() == '0' then
            valid = FALSE;      // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
        else
            valid = TRUE;
        end;
    elsif HaveAArch32() then    // AArch32 state
        (valid, el) = ELFromM32(spsr[4:0]);
    else
        valid = FALSE;
    end;

    if !valid then el = ARBITRARY : bits(2); end;
    return (valid,el);
end;
```

J1.4.3.368 ELIsInHost

```
// ELIsInHost()
// ============

readonly func ELIsInHost(el : bits(2)) => boolean
begin
    if !IsFeatureImplemented(FEAT_VHE) || ELUsingAArch32(EL2) then
        return FALSE;
    end;
    case el of
        when EL3 =>
            return FALSE;
        when EL2 =>
            return EL2Enabled() && EffectiveHCR_EL2_E2H() == '1';
        when EL1 =>
            return FALSE;
        when EL0 =>
            return EL2Enabled() && EffectiveHCR_EL2_E2H()::HCR_EL2().TGE == '11';
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.369 ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

readonly func ELStateUsingAArch32(el : bits(2), secure : boolean) => boolean
begin
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    let (known, aarch32) : (boolean, boolean) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.370 ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================
// Returns (known, aarch32):
//   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
//             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
//   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.

readonly func ELStateUsingAArch32K(el : bits(2), secure : boolean) => (boolean, boolean)
begin
    assert HaveEL(el);

    if !HaveAArch32EL(el) then
        return (TRUE, FALSE);   // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE);   // Secure EL2 is using AArch64
    elsif !HaveAArch64() then
        return (TRUE, TRUE);    // Highest Exception level, therefore all levels are using AArch32
    end;

    // Remainder of function deals with the interprocessing cases when highest
    // Exception level is using AArch64.

    if el == EL3 then
        return (TRUE, FALSE);
    end;

    if (HaveEL(EL3) && EffectiveSCR_EL3_RW() == '0' &&
          (!secure || !IsFeatureImplemented(FEAT_SEL2) || SCR_EL3().EEL2 == '0')) then
        // AArch32 below EL3.
        return (TRUE, TRUE);
    end;

    if el == EL2 then
        return (TRUE, FALSE);
    end;

    if (HaveEL(EL2) && !ELIsInHost(EL0) && HCR_EL2().RW == '0' &&
          (!secure || (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3().EEL2 == '1'))) then
        // AArch32 below EL2.
        return (TRUE, TRUE);
    end;

    if el == EL1 then
        return (TRUE, FALSE);
    end;

    // The execution state of EL0 is only known from PSTATE.nRW when executing at EL0.
    if PSTATE.EL == EL0 then
        return (TRUE, PSTATE.nRW == '1');
    else
        return (FALSE, ARBITRARY : boolean);
    end;
end;
```

J1.4.3.371 ELUsingAArch32

```
// ELUsingAArch32()
// ================

readonly func ELUsingAArch32(el : bits(2)) => boolean
                                    recurselimit 1
begin
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.372 ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

readonly func ELUsingAArch32K(el : bits(2)) => (boolean,boolean)
begin
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
end;
```

J1.4.3.373 EffectiveEA

```
// EffectiveEA()
// =============
// Returns effective SCR_EL3.EA value

readonly func EffectiveEA() => bit
begin
    if !HaveEL(EL3) || Halted() then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3().EA else SCR().EA;
    end;
end;
```

J1.4.3.374 EffectiveHCR_EL2_E2H

```
// EffectiveHCR_EL2_E2H()
// ======================
// Return the Effective HCR_EL2.E2H value.

readonly func EffectiveHCR_EL2_E2H() => bit
begin
    if !IsFeatureImplemented(FEAT_VHE) then
        return '0';
    end;

    if !IsFeatureImplemented(FEAT_E2H0) then
        return '1';
    end;

    return HCR_EL2().E2H;
end;
```

J1.4.3.375 EffectiveHCR_EL2_NVx

```
// EffectiveHCR_EL2_NVx()
// ======================
// Return the Effective value of HCR_EL2.[NV2,NV1,NV].

func EffectiveHCR_EL2_NVx() => bits(3)
begin
    if !EL2Enabled() || !IsFeatureImplemented(FEAT_NV) then
        return '000';
    end;

    var nv1 : bit = HCR_EL2().NV1;
    if (!IsFeatureImplemented(FEAT_E2H0) &&
          ImpDefBool("HCR_EL2.NV1 is implemented as RAZ")) then
        nv1 = '0';
    end;

    if HCR_EL2().NV == '0' then
        if nv1 == '1' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                when Constraint_NVNV1_00 => return '000';
                when Constraint_NVNV1_01 => return '010';
                when Constraint_NVNV1_11 => return '011';
            end;
        else
            return '000';
        end;
    end;

    if !IsFeatureImplemented(FEAT_NV2) then
        return '0' :: nv1 :: '1';
    end;

    var nv2 : bit = HCR_EL2().NV2;
    if (nv2 == '0' && ImpDefBool(
          "Programming HCR_EL2.<NV,NV2> to '10' behaves as '11'")) then
        nv2 = '1';
    end;

    return nv2 :: nv1 :: '1';
end;
```

J1.4.3.376 EffectiveSCR_EL3_FIQ

```
// EffectiveSCR_EL3_FIQ()
// ======================
// Returns effective SCR_EL3().FIQ value

readonly func EffectiveSCR_EL3_FIQ() => bit
begin
    if !HaveEL(EL3) || ELUsingAArch32(EL3) then
        return '0';
    end;
    return SCR_EL3().FIQ;
end;
```

J1.4.3.377 EffectiveSCR_EL3_IRQ

```
// EffectiveSCR_EL3_IRQ()
// ======================
// Returns effective SCR_EL3().IRQ value

readonly func EffectiveSCR_EL3_IRQ() => bit
begin
    if !HaveEL(EL3) || ELUsingAArch32(EL3) then
        return '0';
    end;
    return SCR_EL3().IRQ;
end;
```

J1.4.3.378 EffectiveSCR_EL3_NS

```
// EffectiveSCR_EL3_NS()
// =====================
// Return Effective SCR_EL3.NS value.

readonly func EffectiveSCR_EL3_NS() => bit
begin
    if !HaveSecureState() then
        return '1';
    elsif !HaveEL(EL3) then
        return '0';
    elsif ELUsingAArch32(EL3) then
        return SCR().NS;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        return SCR_EL3().NS;
    end;
end;
```

J1.4.3.379 EffectiveSCR_EL3_NSE

```
// EffectiveSCR_EL3_NSE()
// ======================
// Return Effective SCR_EL3.NSE value.

readonly func EffectiveSCR_EL3_NSE() => bit
begin
    return if !IsFeatureImplemented(FEAT_RME) then '0' else SCR_EL3().NSE;
end;
```

J1.4.3.380 EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====================
// Returns effective SCR_EL3.RW value

readonly func EffectiveSCR_EL3_RW() => bit
begin
    if !HaveAArch64() then
        return '0';
    end;
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    end;
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && EffectiveSCR_EL3_NS() == '1' then
            return '1';
        end;
        if (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3().EEL2 == '1' &&
              EffectiveSCR_EL3_NS() == '0') then
            return '1';
        end;
    end;
    return SCR_EL3().RW;
end;
```

J1.4.3.381 EffectiveTGE

```
// EffectiveTGE()
// ==============
// Returns effective TGE value

func EffectiveTGE() => bit
begin
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR().TGE else HCR_EL2().TGE;
    else
        return '0';        // Effective value of TGE is zero
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.382 EndOfInstruction

```
// EndOfInstruction()
// ==================
// Terminate processing of the current instruction.

noreturn impdef func EndOfInstruction(EndReason : EndOfInstructionReason)
begin
    throw EndOfInstructionException{reason=EndReason};
end;
```

J1.4.3.383 EndOfInstructionException

```
// EndOfInstructionException
// =========================
// End of Instruction exception type.

type EndOfInstructionException of exception {
    reason : EndOfInstructionReason
};
```

J1.4.3.384 EndOfInstructionReason

```
// EndOfInstructionReason
// ======================

type EndOfInstructionReason of enumeration {
    EndOfInstructionReason_Retired,
    EndOfInstructionReason_Exception
};
```

J1.4.3.385 EnterLowPowerState

```
// EnterLowPowerState()
// ====================
// PE enters a low-power state.


impdef func EnterLowPowerState()
begin
    return;
end;
```

J1.4.3.386 EventRegister

```
// EventRegister
// =============
// Event Register for this PE.

var EventRegister : bits(1);
```

J1.4.3.387 ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// ================================
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

type ExceptionalOccurrenceTargetState of enumeration {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.388 ExecuteAsNOP

```
// ExecuteAsNOP()
// ==============

func ExecuteAsNOP()
begin
    EndOfInstruction(EndOfInstructionReason_Retired);
end;
```

J1.4.3.389 FIQPending

```
// FIQPending()
// ============
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.

impdef func FIQPending() => (boolean, boolean)
begin
    return (FALSE, FALSE);
end;
```

J1.4.3.390 GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// ============================
// Returns FP exceptions accumulated by the PE.

impdef func GetAccumulatedFPExceptions() => bits(8)
begin
    return ARBITRARY : bits(8);
end;
```

J1.4.3.391 GetLoadStoreType

```
// GetLoadStoreType()
// ==================
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.

impdef func GetLoadStoreType() => bits(2)
begin
    return ARBITRARY : bits(2);
end;
```

J1.4.3.392 GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

func GetPSRFromPSTATE{N}(targetELState : ExceptionalOccurrenceTargetState) => bits(N)
begin
    if UsingAArch32() && targetELState == AArch32_NonDebugState then
        assert N == 32;
    else
        assert N == 64;
    end;

    var spsr : bits(N) = Zeros{};
    if IsFeatureImplemented(FEAT_UINJ) && targetELState == DebugState then
        spsr[36] = PSTATE.UINJ;
    end;
    spsr[31:28] = PSTATE.[N,Z,C,V];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_PAN) then spsr[22] = PSTATE.PAN; end;
    spsr[20]     = PSTATE.IL;
    if PSTATE.nRW == '1' then                          // AArch32 state
        spsr[27]     = PSTATE.Q;
        spsr[26:25]  = PSTATE.IT[1:0];
        if IsFeatureImplemented(FEAT_SSBS) then spsr[23] = PSTATE.SSBS; end;
        if IsFeatureImplemented(FEAT_DIT) then
            if targetELState == AArch32_NonDebugState then
                spsr[21] = PSTATE.DIT;
            else                                       // AArch64_NonDebugState or DebugState
                spsr[24] = PSTATE.DIT;
            end;
        end;
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr[21] = PSTATE.SS;
        end;
        spsr[19:16]  = PSTATE.GE;
        spsr[15:10]  = PSTATE.IT[7:2];
        spsr[9]      = PSTATE.E;
        spsr[8:6]    = PSTATE.[A,I,F];                 // No PSTATE.D in AArch32 state
        spsr[5]      = PSTATE.T;
        assert PSTATE.M[4] == PSTATE.nRW;              // bit [4] is the discriminator
        spsr[4:0]    = PSTATE.M;
    else                                               // AArch64 state
        if IsFeatureImplemented(FEAT_PAuth_LR) then spsr[35] = PSTATE.PACM; end;
        if IsFeatureImplemented(FEAT_GCS) then spsr[34] = PSTATE.EXLOCK; end;
        if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
              IsFeatureImplemented(FEAT_TRBE_EXC)) then
            spsr[32] = PSTATE.PM;
        end;
        if IsFeatureImplemented(FEAT_MTE) then spsr[25] = PSTATE.TCO; end;
        if IsFeatureImplemented(FEAT_DIT) then spsr[24] = PSTATE.DIT; end;
        if IsFeatureImplemented(FEAT_UAO) then spsr[23] = PSTATE.UAO; end;
        spsr[21]    = PSTATE.SS;
        if IsFeatureImplemented(FEAT_NMI) then spsr[13] = PSTATE.ALLINT; end;
        if IsFeatureImplemented(FEAT_SSBS) then spsr[12] = PSTATE.SSBS; end;
        if IsFeatureImplemented(FEAT_BTI) then spsr[11:10] = PSTATE.BTYPE; end;
        spsr[9:6]    = PSTATE.[D,A,I,F];
        spsr[4]      = PSTATE.nRW;
        spsr[3:2]    = PSTATE.EL;
        spsr[0]      = PSTATE.SP;
    end;
    return spsr;
end;
```

J1.4.3.393 HaveAArch32

```
// HaveAArch32()
// =============
// Return TRUE if AArch32 state is supported at at least EL0.

readonly func HaveAArch32() => boolean
begin
    return IsFeatureImplemented(FEAT_AA32EL0);
end;
```

J1.4.3.394 HaveAArch32EL

```
// HaveAArch32EL()
// ===============
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

readonly func HaveAArch32EL(el : bits(2)) => boolean
begin
    case el of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL0 => return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 => return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 => return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 => return IsFeatureImplemented(FEAT_AA32EL3);
    end;
end;
```

J1.4.3.395 HaveAArch64

```
// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

readonly func HaveAArch64() => boolean
begin
    return (IsFeatureImplemented(FEAT_AA64EL0)
            || IsFeatureImplemented(FEAT_AA64EL1)
            || IsFeatureImplemented(FEAT_AA64EL2)
            || IsFeatureImplemented(FEAT_AA64EL3)
            );
end;
```

J1.4.3.396 HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

readonly func HaveEL(el : bits(2)) => boolean
begin
    case el of
        when EL1,EL0 =>
            return TRUE;                        // EL1 and EL0 must exist
        when EL2 =>
            return IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 =>
            return IsFeatureImplemented(FEAT_AA64EL3) || IsFeatureImplemented(FEAT_AA32EL3);
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.397 HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

readonly func HaveELUsingSecurityState(el : bits(2), secure : boolean) => boolean
begin

    case el of
        when EL3 =>
            assert secure;
            return HaveEL(EL3);
        when EL2 =>
            if secure then
                return HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2);
            else
                return HaveEL(EL2);
            end;
        otherwise =>
            return (HaveEL(EL3) ||
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    (secure == ImpDefBool("Secure-only implementation")));
    end;
end;
```

J1.4.3.398 HaveSecureState

```
// HaveSecureState()
// =================
// Return TRUE if Secure State is supported.

readonly func HaveSecureState() => boolean
begin
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    end;
    if IsFeatureImplemented(FEAT_RME) && !IsFeatureImplemented(FEAT_SEL2) then
        return FALSE;
    end;
    return TRUE;
end;
```

J1.4.3.399 HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

readonly func HighestEL() => bits(2)
begin
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
    end;
end;
```

J1.4.3.400 Hint_CLRBHB

```
// Hint_CLRBHB()
// =============
// Provides a hint to clear the branch history for the current context.

impdef func Hint_CLRBHB()
begin
    return;
end;
```

J1.4.3.401 Hint_DGH

```
// Hint_DGH()
// ==========
// Provides a hint to close any gathering occurring within the implementation.

impdef func Hint_DGH()
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.402 Hint_StoreShared

```
// Hint_StoreShared()
// ==================
// Provides a hint that if the next instruction is an explicit write it is being waited on by
// observers and as such the data should propagate to them with minimum latency.
// A stream value of FALSE indicates KEEP whilst a value of TRUE indicates STRM.

impdef func Hint_StoreShared(stream : boolean)
begin
    return;
end;
```

J1.4.3.403 Hint_WFE

```
// Hint_WFE()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

func Hint_WFE()
begin
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;
    end;

    var trap : boolean;
    var target_el : bits(2);
    (trap, target_el) = AArch64_CheckForWFxTrap(WFxType_WFE);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            var delay_enabled : boolean;
            var delay : integer;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if WaitForEventUntilDelay(delay_enabled, delay) then
                // Event arrived before delay
                return;
            end;
        end;

        // Proceed with trapping
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        else
            AArch64_WFxTrap(WFxType_WFE, target_el);
        end;
    else
        WaitForEvent();
    end;
end;
```

J1.4.3.404 Hint_WFET

```
// Hint_WFET()
// ===========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET,  a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

func Hint_WFET(localtimeout : integer)
begin
    if IsEventRegisterSet() then
        ClearEventRegister();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return;
    end;

    if IsFeatureImplemented(FEAT_WFxT) && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction(EndOfInstructionReason_Retired);
    end;

    var trap : boolean;
    var target_el : bits(2);
    (trap, target_el) = AArch64_CheckForWFxTrap(WFxType_WFET);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            var delay_enabled : boolean;
            var delay : integer;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if WaitForEventUntilDelay(delay_enabled, delay) then
                // Event arrived before the delay expired
                return;
            end;
        end;

        // Proceed with trapping
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        else
            AArch64_WFxTrap(WFxType_WFET, target_el);
        end;
    else
        WaitForEvent(localtimeout);
    end;
end;
```

J1.4.3.405 Hint_WFI

```
// Hint_WFI()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

func Hint_WFI()
begin
    if AArch64_InterruptPending() then
        // No further operation if an interrupt is pending.
        EndOfInstruction(EndOfInstructionReason_Retired);
    end;

    var trap : boolean;
    var target_el : bits(2);
    (trap, target_el) = AArch64_CheckForWFxTrap(WFxType_WFI);
    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        end;
        AArch64_WFxTrap(WFxType_WFI, target_el);
    else
        WaitForInterrupt();
    end;
end;
```

J1.4.3.406 Hint_WFIT

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Hint_WFIT()
// ===========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

func Hint_WFIT(localtimeout : integer)
begin
    if (AArch64_InterruptPending() || (IsFeatureImplemented(FEAT_WFxT) &&
          LocalTimeoutEvent(localtimeout))) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction(EndOfInstructionReason_Retired);
    end;

    var trap : boolean;
    var target_el : bits(2);
    (trap, target_el) = AArch64_CheckForWFxTrap(WFxType_WFIT);
    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        end;
        AArch64_WFxTrap(WFxType_WFIT, target_el);
    else
        WaitForInterrupt(localtimeout);
    end;
end;
```

J1.4.3.407 Hint_Yield

```
// Hint_Yield()
// ============
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

impdef func Hint_Yield()
begin
    return;
end;
```

J1.4.3.408 IRQPending

```
// IRQPending()
// ============
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.

impdef func IRQPending() => (boolean, boolean)
begin
    return (FALSE, FALSE);
end;
```

J1.4.3.409 IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

func IllegalExceptionReturn{N}(spsr : bits(N)) => boolean
begin
    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state, when SecureEL2 is not enabled.
    //   * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //   * To AArch64 state with SPSR.M[1]==1.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    //   * To AArch32 state with an illegal value of SPSR.M.
    let (valid, target) : (boolean, bits(2)) = ELFromSPSR{N}(spsr);
    if !valid then return TRUE; end;

    // Check for return to higher Exception level.
    if UInt(target) > UInt(PSTATE.EL) then return TRUE; end;

    let spsr_mode_is_aarch32 : boolean = (spsr[4] == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above).
    let (known, target_el_is_aarch32) : (boolean, boolean) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE; end;

    // Check for illegal return from AArch32 to AArch64.
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE; end;
    // Check for illegal return to EL1 when HCR_EL2.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if EL2Enabled() && target == EL1 && HCR_EL2().TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE; end;
    end;

    if (IsFeatureImplemented(FEAT_GCS) && PSTATE.EXLOCK == '0' &&
          PSTATE.EL == target && GetCurrentEXLOCKEN()) then
        return TRUE;
    end;

    return FALSE;
end;
```

J1.4.3.410 InstrSet

```
// InstrSet
// ========

type InstrSet of enumeration {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

J1.4.3.411 InstructionFetchBarrier

```
// InstructionFetchBarrier()
// =========================

impdef func InstructionFetchBarrier()
begin
    return;
end;
```

J1.4.3.412 InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// ===================================

impdef func InstructionSynchronizationBarrier()
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.413 IsASEInstruction

```
// IsASEInstruction()
// ==================
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

impdef func IsASEInstruction() => boolean
begin
    return FALSE;
end;
```

J1.4.3.414 IsCurrentSecurityState

```
// IsCurrentSecurityState()
// ========================
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

func IsCurrentSecurityState(ss : SecurityState) => boolean
begin
    return CurrentSecurityState() == ss;
end;
```

J1.4.3.415 IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

func IsEventRegisterSet() => boolean
begin
    return EventRegister == '1';
end;
```

J1.4.3.416 IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

readonly func IsHighestEL(el : bits(2)) => boolean
begin
    return HighestEL() == el;
end;
```

J1.4.3.417 IsInHost

```
// IsInHost()
// ==========

readonly func IsInHost() => boolean
begin
    return ELIsInHost(PSTATE.EL);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.418 IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

readonly func IsSecureBelowEL3() => boolean
begin
    if HaveEL(EL3) then
        return if !HaveAArch64() then SCR().NS == '0' else SCR_EL3().NS == '0';
    elsif HaveEL(EL2) && (!IsFeatureImplemented(FEAT_SEL2) || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if PE is Secure or FALSE if Non-secure.
        return ImpDefBool("Secure-only implementation");
    end;
end;
```

J1.4.3.419 IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

readonly func IsSecureEL2Enabled() => boolean
begin
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3().EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
            end;
        else
            return SecureOnlyImplementation();
        end;
    else
        return FALSE;
    end;
end;
```

J1.4.3.420 LocalTimeoutEvent

```
// LocalTimeoutEvent()
// ===================
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.

func LocalTimeoutEvent(localtimeout : integer) => boolean
begin
    assert localtimeout >= 0;

    let cntvct : bits(64) = VirtualCounterTimer();
    if UInt(cntvct) >= localtimeout then
        return TRUE;
    end;

    IsLocalTimeoutEventPending = TRUE;
    LocalTimeoutVal = localtimeout[63:0];   // Store value to compare against
                                            // Virtual Counter Timer at subsequent clock ticks
    return FALSE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.421 Mode

```
// Mode bits
// =========
// AArch32 PSTATE.M mode bits.

constant M32_User : bits(5)    = '10000';
constant M32_FIQ : bits(5)     = '10001';
constant M32_IRQ : bits(5)     = '10010';
constant M32_Svc : bits(5)     = '10011';
constant M32_Monitor : bits(5) = '10110';
constant M32_Abort : bits(5)   = '10111';
constant M32_Hyp : bits(5)     = '11010';
constant M32_Undef : bits(5)   = '11011';
constant M32_System : bits(5)  = '11111';
```

J1.4.3.422 NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =============================
// Returns TRUE if the security state is always Non-secure for this implementation.

func NonSecureOnlyImplementation() => boolean
begin
    return ImpDefBool("Non-secure only implementation");
end;
```

J1.4.3.423 PLOfEL

```
// PLOfEL()
// ========

func PLOfEL(el : bits(2)) => PrivilegeLevel
begin
    case el of
        when EL3 =>  return if !HaveAArch64() then PL1 else PL3;
        when EL2 =>  return PL2;
        when EL1 =>  return PL1;
        when EL0 =>  return PL0;
    end;
end;
```

J1.4.3.424 PSTATE

```
// PSTATE
// ======
// PE state bits.
// There is no significance to the field order.

var PSTATE : collection {
    N : bits (1),        // Negative condition flag
    Z : bits (1),        // Zero condition flag
    C : bits (1),        // Carry condition flag
    V : bits (1),        // Overflow condition flag
    D : bits (1),        // Debug mask bit                    [AArch64 only]
    A : bits (1),        // SError interrupt mask bit
    I : bits (1),        // IRQ mask bit
    F : bits (1),        // FIQ mask bit
    EXLOCK : bits (1),   // Lock exception return state
    PAN : bits (1),      // Privileged Access Never Bit       [v8.1]
    UAO : bits (1),      // User Access Override              [v8.2]
    DIT : bits (1),      // Data Independent Timing           [v8.4]
    TCO : bits (1),      // Tag Check Override                [v8.5, AArch64 only]
    PM : bits (1),       // PMU exception Mask
    BTYPE : bits (2),    // Branch Type                       [v8.5]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PACM : bits (1),     // PAC instruction modifier
    ZA : bits (1),       // Accumulation array enabled         [SME]
    SM : bits (1),       // Streaming SVE mode enabled         [SME]
    ALLINT : bits (1),   // Interrupt mask bit
    UINJ : bits (1),     // Undefined Exception Injection
    SS : bits (1),       // Software step bit
    IL : bits (1),       // Illegal Execution state bit
    EL : bits (2),       // Exception level
    nRW : bits (1),      // Execution state: 0=AArch64, 1=AArch32
    SP : bits (1),       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    Q : bits (1),        // Cumulative saturation flag         [AArch32 only]
    GE : bits (4),       // Greater than or Equal flags        [AArch32 only]
    SSBS : bits (1),     // Speculative Store Bypass Safe
    IT : bits (8),       // If-then bits, RES0 in CPSR         [AArch32 only]
    J : bits (1),        // J bit, RES0                        [AArch32 only, RES0 in SPSR and CPSR]
    T : bits (1),        // T32 bit, RES0 in CPSR              [AArch32 only]
    E : bits (1),        // Endianness bit                     [AArch32 only]
    M : bits (5)         // Mode field                         [AArch32 only]
};
```

J1.4.3.425 PhysicalCountInt

```
// PhysicalCountInt()
// ==================
// Returns the integral part of physical count value of the System counter.

func PhysicalCountInt() => bits(64)
begin
    return PhysicalCount[87:24];
end;
```

J1.4.3.426 PrivilegeLevel

```
// PrivilegeLevel
// ==============
// Privilege Level abstraction.

type PrivilegeLevel of enumeration {PL3, PL2, PL1, PL0};
```

J1.4.3.427 RestoredITBits

```
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

func RestoredITBits{N}(spsr : bits(N)) => bits(8)
begin
    let it : bits(8) = spsr[15:10,26:25];

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;
        end;
    end;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it[7:4]) && IsZero(it[3:0]) then
        return '00000000';
    end;

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let itd : bit = if PSTATE.EL == EL2 then HSCTLR().ITD else SCTLR().ITD;
    if (spsr[5] == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it[2:0])) then
        return '00000000';
    else
        return it;
    end;
end;
```

J1.4.3.428 SecureOnlyImplementation

```
// SecureOnlyImplementation()
// ==========================
// Returns TRUE if the security state is always Secure for this implementation.

readonly func SecureOnlyImplementation() => boolean
begin
    return ImpDefBool("Secure-only implementation");
end;
```

J1.4.3.429 SecurityState

```
// SecurityState
// =============
// The Security state of an execution context

type SecurityState of enumeration {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```

J1.4.3.430 SecurityStateAtEL

```
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

readonly func SecurityStateAtEL(EL : bits(2)) => SecurityState
begin
    if IsFeatureImplemented(FEAT_RME) then
        if EL == EL3 then return SS_Root; end;
        let effective_nse_ns : bits(2) = EffectiveSCR_EL3_NSE() :: EffectiveSCR_EL3_NS();
        case effective_nse_ns of
            when '00' =>
                if IsFeatureImplemented(FEAT_SEL2) then
                    return SS_Secure;
                else
                    unreachable;
                end;
            when '01' =>
                return SS_NonSecure;
            when '11' =>
                return SS_Realm;
            otherwise =>
                unreachable;
        end;
    end;

    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    elsif EL == EL3 then
        return SS_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if EffectiveSCR_EL3_NS() == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR().NS == '1' then SS_NonSecure else SS_Secure;
        end;
    end;
end;
```

J1.4.3.431 SendEvent

```
// SendEvent()
// ===========
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

impdef func SendEvent()
begin
    SendEventLocal();
end;
```

J1.4.3.432 SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

func SendEventLocal()
begin
    EventRegister = '1';
    return;
end;
```

J1.4.3.433 SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// ============================
// Stores FP Exceptions accumulated by the PE.

impdef func SetAccumulatedFPExceptions(accumulated_exceptions : bits(8))
begin
    return;
end;
```

J1.4.3.434 SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================

func SetPSTATEFromPSR{N}(spsr : bits(N))
begin
    let illegal_psr_state : boolean = IllegalExceptionReturn{N}(spsr);
    SetPSTATEFromPSR{N}(spsr, illegal_psr_state);
end;

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
func SetPSTATEFromPSR{N}(spsr_in : bits(N), illegal_psr_state : boolean)
begin
    var spsr : bits(N) = spsr_in;
    let from_aarch64 : boolean = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS{N}(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = ARBITRARY : bits(2); end;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr[20];
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = spsr[36]; end;
        if spsr[4] == '1' then                    // AArch32 state
            AArch32_WriteMode(spsr[4:0]);         // Sets PSTATE.EL correctly
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr[23]; end;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            (-, PSTATE.EL) = ELFromSPSR{N}(spsr);
            PSTATE.SP  = spsr[0];
            if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = spsr[11:10]; end;
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr[12]; end;
            if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = spsr[23]; end;
            if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = spsr[24]; end;
            if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = spsr[25]; end;
            if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = spsr[34]; end;
            if IsFeatureImplemented(FEAT_PAuth_LR) then
                PSTATE.PACM = if IsPACMEnabled() then spsr[35] else '0';
            end;
        end;
    end;

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr[5] = '0'; end;
    end;

    // State that is reinstated regardless of illegal exception return
    PSTATE.[N,Z,C,V] = spsr[31:28];
    if IsFeatureImplemented(FEAT_PAN) then PSTATE.PAN = spsr[22]; end;
    if PSTATE.nRW == '1' then                     // AArch32 state
        PSTATE.Q         = spsr[27];
        PSTATE.IT        = RestoredITBits{N}(spsr);
        ShouldAdvanceIT  = FALSE;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr[24] else spsr[21]);
        end;
        PSTATE.GE        = spsr[19:16];
        PSTATE.E         = spsr[9];
        PSTATE.[A,I,F]   = spsr[8:6];             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr[5];               // PSTATE.J is RES0
    else                                          // AArch64 state
        if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
              IsFeatureImplemented(FEAT_TRBE_EXC)) then
            PSTATE.PM    = spsr[32];
        end;
        if IsFeatureImplemented(FEAT_NMI) then PSTATE.ALLINT  = spsr[13]; end;
        PSTATE.[D,A,I,F] = spsr[9:6];             // No PSTATE.[Q,IT,GE,E,T] in AArch64 state
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    return;
end;
```

J1.4.3.435 ShouldAdvanceHS

```
// ShouldAdvanceHS
// ===============
// Cleared if we should not advance the EDESR.SS after the current instruction.

var ShouldAdvanceHS : boolean;
```

J1.4.3.436 ShouldAdvanceIT

```
// ShouldAdvanceIT
// ===============
// Cleared if we should not advance the PSTATE.IT after the current instruction.

var ShouldAdvanceIT : boolean;
```

J1.4.3.437 ShouldAdvanceSS

```
// ShouldAdvanceSS
// ===============
// Cleared if PSTATE.SS is written by the current instruction.

var ShouldAdvanceSS : boolean;
```

J1.4.3.438 SmallestTranslationGranule

```
// SmallestTranslationGranule()
// ============================
// Smallest implemented translation granule.

func SmallestTranslationGranule() => integer
begin
    if IsFeatureImplemented(FEAT_TGran4K) then return 12; end;
    if IsFeatureImplemented(FEAT_TGran16K) then return 14; end;
    if IsFeatureImplemented(FEAT_TGran64K)  then return 16; end;
    unreachable;
end;
```

J1.4.3.439 SpeculationBarrier

```
// SpeculationBarrier()
// ====================

impdef func SpeculationBarrier()
begin
    // Since there is no speculation in asl model, this is a NOP.
    return;
end;
```

J1.4.3.440 SynchronizeContext

```
// SynchronizeContext()
// ====================
// Context Synchronization event, includes Instruction Fetch Barrier effect

impdef func SynchronizeContext()
begin
    InstructionFetchBarrier();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.441 SynchronizeErrors

```
// SynchronizeErrors()
// ===================
// Implements the error synchronization event.

impdef func SynchronizeErrors()
begin
    return;
end;
```

J1.4.3.442 TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// ======================================
// Take any pending unmasked physical SError interrupt.

impdef func TakeUnmaskedPhysicalSErrorInterrupts(iesb_req : boolean)
begin
    return;
end;
```

J1.4.3.443 TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// ==============================
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

impdef func TakeUnmaskedSErrorInterrupts()
begin
    return;
end;
```

J1.4.3.444 ThisInstr

```
// ThisInstr()
// ===========
// Return the bitstring encoding of the currently executing instruction.

impdef func ThisInstr() => bits(32)
begin
    Unimplemented();
end;
```

J1.4.3.445 ThisInstrLength

```
// ThisInstrLength()
// =================
// Return the length of the current instruction.

impdef func ThisInstrLength() => integer
begin
    Unimplemented();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.446 UndefinedInjectionCheck

```
// UndefinedInjectionCheck()
// =========================
// Check PSTATE.UINJ to determine if execution of the current
// instruction should cause an Undefined exception.

func UndefinedInjectionCheck()
begin
    if IsFeatureImplemented(FEAT_UINJ) && PSTATE.UINJ == '1' then
        Undefined();
    end;
end;
```

J1.4.3.447 UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64_

readonly func UsingAArch32() => boolean
begin
    let aarch32 : boolean = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32; end;
    if !HaveAArch64() then assert aarch32; end;
    return aarch32;
end;
```

J1.4.3.448 ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// ========================
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

func ValidSecurityStateAtEL(el : bits(2)) => boolean
begin
    if !HaveEL(el) then
        return FALSE;
    end;

    if el == EL3 then
        return TRUE;
    end;

    if IsFeatureImplemented(FEAT_RME) then
        let effective_nse_ns : bits(2) = EffectiveSCR_EL3_NSE() :: EffectiveSCR_EL3_NS();
        if effective_nse_ns == '10' then
            return FALSE;
        end;
    end;

    if el == EL2 then
        return EL2Enabled();
    end;

    return TRUE;
end;
```

J1.4.3.449 VirtualFIQPending

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// VirtualFIQPending()
// ===================
// Returns TRUE if there is any pending virtual FIQ.

impdef func VirtualFIQPending() => boolean
begin
    return FALSE;
end;
```

J1.4.3.450 VirtualIRQPending

```
// VirtualIRQPending()
// ===================
// Returns TRUE if there is any pending virtual IRQ.

impdef func VirtualIRQPending() => boolean
begin
    return FALSE;
end;
```

J1.4.3.451 WFxType

```
// WFxType
// =======
// WFx instruction types.

type WFxType of enumeration {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WFIT};
```

J1.4.3.452 WaitForEvent

```
// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

func WaitForEvent()
begin
    if Halted() then return; end;
    if !IsEventRegisterSet() then
        EnterLowPowerState();
    end;
    return;
end;

// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

func WaitForEvent(localtimeout : integer)
begin
    if Halted() then return; end;
    if !(IsEventRegisterSet() || LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
end;
```

J1.4.3.453 WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.

func WaitForInterrupt()
begin
    if Halted() then return; end;
    EnterLowPowerState();
    return;
end;

// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

func WaitForInterrupt(localtimeout : integer)
begin
    if Halted() then return; end;
    if !LocalTimeoutEvent(localtimeout) then
        EnterLowPowerState();
    end;
    return;
end;
```

J1.4.3.454 WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// ===========================
// Update common Watchpoint related fields.

func WatchpointRelatedSyndrome(fault : FaultRecord) => bits(24)
begin
    var syndrome : bits(24) = Zeros{};

    if fault.watchptinfo.maybe_false_match then
        syndrome[16] = '1';                                         // WPF
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        syndrome[16] = ImpDefBit("WPF value on TRUE Watchpoint match");
    end;

    if IsRelaxedWatchpointAccess(fault.accessdesc) then
        if HaltOnBreakpointOrWatchpoint() then
            if ImpDefBool("EDWAR is not valid on watchpoint debug event") then
                syndrome[10] = '1';                                 // FnV
            end;
        else
            if ImpDefBool("FAR is not valid on watchpoint exception") then
                syndrome[10] = '1';                                 // FnV
            end;
        end;
    else
        if WatchpointFARNotPrecise(fault) then
            syndrome[15] = '1';                                     // FnP
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;

    // Watchpoint number is valid if FEAT_Debugv8p9 is implemented or
    // if Feat_Debugv8p2 is implemented and below set of conditions are satisfied:
    // - Either FnV = 1 or FnP = 1.
    // - If the address recorded in FAR is not within a naturally-aligned block of memory.
    // Otherwise, it is IMPLEMENTATION DEFINED if watchpoint number is valid.
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        syndrome[17]    = '1';                                      // WPTV
        syndrome[23:18] = fault.watchptinfo.watchpt_num[5:0];       // WPT
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        if syndrome[15] == '1' || syndrome[10] == '1' then          // Either of FnP or FnV is 1
            syndrome[17] = '1';                                     // WPTV
        elsif !AddressInNaturallyAlignedBlock(fault.vaddress, fault.watchptinfo.vaddress) then
            syndrome[17] = '1';                                     // WPTV
        elsif ImpDefBool("WPTV field is valid") then
            syndrome[17] = '1';
        end;
        if syndrome[17] == '1' then
            syndrome[23:18] = fault.watchptinfo.watchpt_num[5:0];   // WPT
        else
            syndrome[23:18] = ARBITRARY : bits(6);
        end;
    end;

    return syndrome;
end;
```

J1.4.3.455 tlbi

```
constant VMID_NONE : bits(NUM_VMIDBITS) = Zeros{};
constant ASID_NONE : bits(NUM_ASIDBITS) = Zeros{};
```

J1.4.3.456 Broadcast

```
// Broadcast
// =========

type Broadcast of enumeration {
    Broadcast_ISH,
    Broadcast_ForcedISH,
    Broadcast_OSH,
    Broadcast_NSH
};
```

J1.4.3.457 BroadcastTLBI

```
// BroadcastTLBI()
// ===============
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated broadcast
// domain.

impdef func BroadcastTLBI(broadcast : Broadcast, r : TLBIRecord)
begin
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.458 ExcludeXS

```
// ExcludeXS()
// ===========
// Returns TRUE if a TLBI maintenance instruction without the nXS qualifier executed at EL1
// behaves in the same way as the corresponding TLBI maintenance instruction with the nXS qualifier.

func ExcludeXS() => boolean
begin
    return (IsFeatureImplemented(FEAT_XS) && PSTATE.EL == EL1 &&
            IsHCRXEL2Enabled() && HCRX_EL2().FnXS == '1');
end;
```

J1.4.3.459 TLBI

```
// TLBI()
// ======
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

impdef func TLBI(r : TLBIRecord)
begin
    return;
end;
```

J1.4.3.460 TLBILevel

```
// TLBILevel
// =========

type TLBILevel of enumeration {
    TLBILevel_Any,        // this applies to TLB entries at all levels
    TLBILevel_Last        // this applies to TLB entries at last level only
};
```

J1.4.3.461 TLBIMemAttr

```
// TLBIMemAttr
// ===========
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

type TLBIMemAttr of enumeration {
    TLBI_AllAttr,         // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS        // Only TLB entries with XS=0 within the scope of the invalidation
};
```

J1.4.3.462 TLBIOp

```
// TLBIOp
// ======

type TLBIOp of enumeration {
    TLBIOp_DALL,          // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,          // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIPOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    TLBIPOp_VAA,
    TLBIPOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIPOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIPOp_RVAA,
    TLBIPOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
    TLBIOp_VMALLWS2
};
```

J1.4.3.463 TLBIRecord

```
// TLBIRecord
// ==========
// Details related to a TLBI operation.

type TLBIRecord of record {
    op : TLBIOp,
    from_aarch64 : boolean, // originated as an AArch64 operation
    security : SecurityState,
    regime : Regime,
    use_vmid : boolean,
    vmid : bits(NUM_VMIDBITS),
    asid : bits(NUM_ASIDBITS),
    level : TLBILevel,
    attr : TLBIMemAttr,
    ipaspace : PASpace,     // For operations that take IPA as input address
    address : bits(64),     // input address, for range operations, start address
    end_address : bits(64), // for range operations, end; address
    d64 : boolean,          // For operations that evict VMSAv8-64 based TLB entries
    d128 : boolean,         // For operations that evict VMSAv9-128 based TLB entries
    ttl : bits(4),          // translation table walk level holding the leaf entry
                            // for the address being invalidated
                            // For Non-Range Invalidations:
                            //   When the ttl is
                            //     '00xx'    : this applies to all TLB entries
                            //     Otherwise : TLBIP instructions invalidates D128 TLB
                            //                 entries only
                            //                 TLBI instructions invalidates D64 TLB
                            //                 entries only
                            // For Range Invalidations:
                            //   When the ttl is
                            //     '00'      : this applies to all TLB entries
                            //     Otherwise : TLBIP instructions invalidates D128 TLB
                            //                 entries only
                            //                 TLBI instructions invalidates D64 TLB
                            //                 entries only
    tg : bits(2)            // for range operations, translation granule
};
```

J1.4.3.464 VMID

```
// VMID
// ====
// Effective VMID.

func VMID() => bits(NUM_VMIDBITS)
begin
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if IsFeatureImplemented(FEAT_VMID16) && VTCR_EL2().VS == '1' then
                return VTTBR_EL2().VMID;
            else
                return ZeroExtend{NUM_VMIDBITS}(VTTBR_EL2().VMID[7:0]);
            end;
        else
            return ZeroExtend{NUM_VMIDBITS}(VTTBR().VMID);
        end;
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        return Zeros{NUM_VMIDBITS};
    else
        return VMID_NONE;
    end;
end;
```

J1.4.3.465 Maximum

```
// Maximum walk levels
// ===================
let MAX_WALK_LEVELS : integer{} = (if IsFeatureImplemented(FEAT_D128) then 6
                                   else (if IsFeatureImplemented(FEAT_LPA2) then 5
                                   else 4));
```

J1.4.3.466 Unbounded

```
// Unbounded Variables
// ===================
// Sets the upper limit for a specific loop or function.

constant Unbounded : integer{} = 2^128;
config Unbounded_DiscardTransactionalWrites : integer{0..Unbounded} = Unbounded;
config Unbounded_DescriptorUpdate : integer{0..Unbounded} = Unbounded;
config Unbounded_semihost : integer{0..Unbounded} = Unbounded;
config Unbounded_ParseTrace : integer{0..Unbounded} = Unbounded;
config Unbounded_ParseExtensionPacket : integer{0..Unbounded} = Unbounded;
config Unbounded_ULEB128 : integer{0..Unbounded} = Unbounded;
config Unbounded_ULEB128n : integer{0..Unbounded} = Unbounded;
config Unbounded_FloorPow2 : integer{0..Unbounded} = Unbounded;
config Unbounded_NormalizeReal : integer{0..Unbounded} = Unbounded;
```

J1.4.3.467 ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

readonly impdef func ConstrainUnpredictable(which : Unpredictable) => Constraint
begin
    case which of
        when Unpredictable_VMSR =>
            return Constraint_UNDEF;
        when Unpredictable_WBOVERLAPLD =>
            return Constraint_WBSUPPRESS; // return loaded value
        when Unpredictable_WBOVERLAPST =>
            return Constraint_NONE;     // store pre-writeback value
        when Unpredictable_LDPOVERLAP =>
            return Constraint_UNDEF;    // instruction is UNDEFINED
        when Unpredictable_BASEOVERLAP =>
            return Constraint_UNKNOWN;  // use UNKNOWN address
        when Unpredictable_DATAOVERLAP =>
            return Constraint_UNKNOWN;  // store UNKNOWN value
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when Unpredictable_DEVPAGE2 =>
    return Constraint_FAULT;    // take an alignment fault
when Unpredictable_RESTCF =>
    return Constraint_UNKNOWN;  // Do not take a fault
when Unpredictable_DEVICETAGSTORE =>
    return Constraint_NONE;     // Do not take a fault
when Unpredictable_INSTRDEVICE =>
    return Constraint_NONE;     // Do not take a fault
when Unpredictable_RESCPACR =>
    return Constraint_TRUE;     // Map to UNKNOWN value
when Unpredictable_RESMAIR =>
    return Constraint_UNKNOWN;  // Map to UNKNOWN value
when Unpredictable_S1CTAGGED =>
    return Constraint_FALSE;    // SCTLR_ELx.C == '0' marks address as untagged
when Unpredictable_S2RESMEMATTR =>
    return Constraint_NC;       // Map to Noncacheable value
when Unpredictable_RESTEXCB =>
    return Constraint_UNKNOWN;  // Map to UNKNOWN value
when Unpredictable_RESDACR =>
    return Constraint_UNKNOWN;  // Map to UNKNOWN value
when Unpredictable_RESPRRR =>
    return Constraint_UNKNOWN;  // Map to UNKNOWN value
when Unpredictable_RESVTCRS =>
    return Constraint_UNKNOWN;  // Map to UNKNOWN value
when Unpredictable_RESTnSZ =>
    return Constraint_FORCE;    // Map to the limit value
when Unpredictable_OORTnSZ =>
    return Constraint_FORCE;    // Map to the limit value
when Unpredictable_LARGEIPA =>
    return Constraint_FORCE;    // Restrict the IA size to the PAMax value
when Unpredictable_ESRCONDPASS =>
    return Constraint_FALSE;    // Report as "AL"
when Unpredictable_ILZEROIT =>
    return Constraint_FALSE;    // Do not zero PSTATE.IT
when Unpredictable_ILZEROT =>
    return Constraint_FALSE;    // Do not zero PSTATE.T
when Unpredictable_BPVECTORCATCHPRI =>
    return Constraint_TRUE;     // Debug Vector Catch: match on 2nd halfword
when Unpredictable_VCMATCHHALF =>
    return Constraint_FALSE;    // No match
when Unpredictable_VCMATCHDAPA =>
    return Constraint_FALSE;    // No match on Data Abort or Prefetch abort
when Unpredictable_WPMASKANDBAS =>
    return Constraint_FALSE;    // Watchpoint disabled
when Unpredictable_WPBASCONTIGUOUS =>
    return Constraint_FALSE;    // Watchpoint disabled
when Unpredictable_RESWPMASK =>
    return Constraint_DISABLED; // Watchpoint disabled
when Unpredictable_WPMASKEDBITS =>
    return Constraint_FALSE;    // Watchpoint disabled
when Unpredictable_RESBPWPCTRL =>
    return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL =>
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESBPTYPE =>
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP =>
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESMDSELR =>
    return Constraint_UNKNOWN;  // Map to UNKNOWN value
when Unpredictable_BPMATCHHALF =>
    return Constraint_FALSE;    // No match
when Unpredictable_BPMISMATCHHALF =>
    return Constraint_FALSE;    // No match
when Unpredictable_BPLINKINGDISABLED =>
    return Constraint_FALSE;    // Breakpoint does not match
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when Unpredictable_RESTARTALIGNPC =>
    return Constraint_FALSE;    // Do not force alignment
when Unpredictable_RESTARTZEROUPPERPC =>
    return Constraint_TRUE;     // Force zero extension
when Unpredictable_ZEROUPPER =>
    return Constraint_TRUE;     // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC =>
    return Constraint_TRUE;     // zero top half of PC
when Unpredictable_A32FORCEALIGNPC =>
    return Constraint_FALSE;    // Do not force alignment
when Unpredictable_SMD =>
    return Constraint_UNDEF;    // disabled SMC is Unallocated
when Unpredictable_NONFAULT =>
    return Constraint_FALSE;    // Speculation enabled
when Unpredictable_SVEZEROUPPER =>
    return Constraint_TRUE;     // zero top bits of Z registers
when Unpredictable_SVELDNFDATA =>
    return Constraint_TRUE;     // Load mem data in NF loads
when Unpredictable_SVELDNFZERO =>
    return Constraint_TRUE;     // Write zeros in NF loads
when Unpredictable_CHECKSPNONEACTIVE =>
    return Constraint_TRUE;     // Check SP alignment
when Unpredictable_SMEZEROUPPER =>
    return Constraint_TRUE;     // zero top bits of ZA registers
when Unpredictable_NVNV1 =>
    return Constraint_NVNV1_00; // Map unpredictable configuration of HCR_EL2[NV,NV1]
                                // to NV = 0 and NV1 = 0
when Unpredictable_BADDR_RESS =>
    return Constraint_RESS;     // Values behave as RESS
when Unpredictable_Shareability =>
    return Constraint_OSH;      // Map reserved encoding of shareability to Outer Shareable
when Unpredictable_AFUPDATE =>  // AF update for alignment or Permission fault
    return Constraint_TRUE;
when Unpredictable_DBUPDATE =>  // DB State update for alignment fault
    return Constraint_TRUE;
when Unpredictable_Unsupported_Atomic_HW_Update =>
    return Constraint_FALSE;    // Do not raise Unsupported Atomic HW Update fault
                                // when encountering unsupported memory attributes
when Unpredictable_IESBinDebug =>  // Use SCTLR_ELx[].IESB in Debug state
    return Constraint_TRUE;
when Unpredictable_BADPMSFCR =>    // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
    return Constraint_TRUE;
when Unpredictable_RES_PMU_VS =>   // Bad setting for SVE Streaming mode filter
    return Constraint_FALSE;
when Unpredictable_COUNT_CHAIN =>  // Chained PMU counters idx, idx+1 are not in same range
    return Constraint_FALSE;
when Unpredictable_EL1TIMESTAMP => // Bad settings for TRFCR_EL1
    return Constraint_EL1TIMESTAMP;
when Unpredictable_EL2TIMESTAMP => // Bad settings for TRFCR_EL2
    return Constraint_EL2TIMESTAMP;
when Unpredictable_ZEROBTYPE =>
    return Constraint_TRUE;           // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable_CLEARERRITEZERO => // Clearing sticky errors when instruction in flight
    return Constraint_FALSE;
when Unpredictable_ALUEXCEPTIONRETURN =>
    return Constraint_UNDEF;
when Unpredictable_DBGxVR_RESS =>
    return Constraint_FALSE;
when Unpredictable_PMSCR_PCT =>
    return Constraint_PMSCR_PCT_VIRT;
when Unpredictable_WFxTDEBUG =>
    return Constraint_FALSE;    // WFxT in Debug state does not execute as a NOP
// Accesses are not single-copy atomic above the byte level
when Unpredictable_LS64UNSUPPORTED =>
    return Constraint_LIMITED_ATOMICITY;
// Unaligned exclusives, atomics, acquire/release
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// to a region that is not to Normal inner write-back
// outer write-back shareable generate an Alignment fault.
when Unpredictable_LSE2_ALIGNMENT_FAULT =>
    return Constraint_FALSE;
when Unpredictable_LSE128OVERLAP =>
    return Constraint_UNDEF;    // instruction is UNDEFINED
when Unpredictable_IGNORETRAPINDEBUG =>
    return Constraint_FALSE;    // Trap to register access in debug state is enabled
when Unpredictable_PMUEVENTCOUNTER =>
    return Constraint_UNDEF;    // Accesses to the register are UNDEFINED
when Unpredictable_RES_HPMN =>
    return Constraint_UNKNOWN;  // The counter is reserved for EL2 use
when Unpredictable_RES_EPMN =>
    return Constraint_UNKNOWN;  // The counter is reserved for external use
when Unpredictable_BRBFILTRATE =>
    return Constraint_FALSE;    // BRB_FILTRATE event not generated on BRB injection
when Unpredictable_PMUSNAPSHOTEVENT => // PMU_SNAPSHOT event not generated in Debug state
    return Constraint_FALSE;
when Unpredictable_RESEPMSSAD =>
    // External access to PMUv3 Snapshot extension allowed in Secure state
    // when FEAT_RME is not implemented or Root state otherwise
    return Constraint_UNKNOWN;
when Unpredictable_RESPMSSE =>
    return Constraint_DISABLED;   // PMU capture events disabled
when Unpredictable_RESPMEE =>
    return Constraint_DISABLED;   // PMU Profiling exception disabled, PMUIRQ enabled
when Unpredictable_RESPMSEE =>
    return Constraint_DISABLED;   // SPE Profiling exception disabled
when Unpredictable_RESTRFEE =>
    return Constraint_DISABLED;   // TRBE Profiling exception disabled
when Unpredictable_RESTC =>
    return Constraint_DISABLED;   // Threshold features disabled
when Unpredictable_MOPSOVERLAP => // Instruction is UNDEFINED
    return Constraint_UNDEF;
when Unpredictable_MOPS_R31 =>    // Instruction is UNDEFINED
    return Constraint_UNDEF;
when Unpredictable_CASRETURNOLDVALUE =>
    return Constraint_TRUE;
when Unpredictable_WRITEFAILEDCAS =>
    return Constraint_FALSE;
when Unpredictable_WRITEFAILEDRCWCAS =>
    return Constraint_FALSE;
when Unpredictable_STRONLYTAGCHECKEDCAS =>
    return Constraint_TRUE;     // CAS with compare fail does a Tag Check
when Unpredictable_STRONLYTAGCHECKEDRCWSCAS =>
    return Constraint_TRUE;     // RCW(S)CAS with RCW(S) fail/compare fail does a Tag Check
when Unpredictable_RESERVEDNSxB =>
    return Constraint_MAPTOALLOCATED;
when Unpredictable_RESERVEDNSxB_Trap =>
    return Constraint_TRUE;
when Unpredictable_RES_ETBAD =>
    // ExternalTracebufferaccess disabled for res values
    return Constraint_DISABLED;
when Unpredictable_RESBPMASK =>
    return Constraint_DISABLED; // Mask set to 0
when Unpredictable_BPMASK =>
    return Constraint_FALSE;    // Breakpoint match will fail
when Unpredictable_BPMASKEDBITS =>
    return Constraint_FALSE;    // Breakpoint match will fail
when Unpredictable_BPLINKEDADDRMATCH =>
    return Constraint_FALSE;    // Breakpoint match will fail
when Unpredictable_16BYTEROUNDEDUPACCESS =>
    // Watchpoint match of 16 byte rounded range in case of SVE access
    return Constraint_TRUE;
when Unpredictable_16BYTEROUNDEDDOWNACCESS =>
    // Watchpoint match of 16 byte rounded range in case of SVE access
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return Constraint_TRUE;
        when Unpredictable_NODTRTAGCHK =>
            // The load or store operation does not perform the Tag check in Debug state
            return Constraint_TRUE;
        when Unpredictable_CPACHECK =>     // Do not perform Checked Pointer Arithmetic
            return Constraint_FALSE;
        when Unpredictable_Atomic_SYNC_ABORT =>
            return Constraint_FALSE;
        when Unpredictable_Atomic_SERROR =>
            return Constraint_FALSE;
        when Unpredictable_Atomic_NOP =>
            return Constraint_FALSE;
        when Unpredictable_Atomic_MMU_IMPDEF_FAULT =>
            return Constraint_FALSE;
        when Unpredictable_USE_DEFAULT_PMG =>
            return Constraint_TRUE;
        when Unpredictable_BankedRegister =>
            return Constraint_NOP;
        when Unpredictable_UnimplementedRegister =>
            return Constraint_NOP;
        when Unpredictable_RES_RA =>
            return Constraint_TRUE;
        when Unpredictable_BPNOTLINKING =>
            return Constraint_FALSE;
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.3.468 ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

readonly impdef func ConstrainUnpredictableBits{width}(which : Unpredictable
                                                        ) => (Constraint, bits(width))
begin
    let c : Constraint = ConstrainUnpredictable(which);

    case c of
        when Constraint_UNKNOWN =>
            return (c, Zeros{width});     // See notes; this is an example implementation only
        when Constraint_PMSCR_PCT_VIRT =>
            return (c, Zeros{width});
        when Constraint_NC where which == Unpredictable_S2RESMEMATTR =>
            assert width == 4;
            return (c, ZeroExtend{width}('0101'));
        when Constraint_WT where which == Unpredictable_S2RESMEMATTR =>
            assert width == 4;
            return (c, ZeroExtend{width}('1010'));
        when Constraint_WB where which == Unpredictable_S2RESMEMATTR =>
            assert width == 4;
            return (c, ZeroExtend{width}('1111'));
        when Constraint_EL1TIMESTAMP =>
            return (c, ZeroExtend{width}('01'));
        when Constraint_EL2TIMESTAMP =>
            return (c, ZeroExtend{width}('01'));
        when Constraint_MAPTOALLOCATED =>
            return (c, ZeroExtend{width}('010'));  // return non-secure state_bits
        otherwise =>
            return (c, ARBITRARY : bits(width));   // bits result not used
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
end;
```

J1.4.3.469 ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

impdef func ConstrainUnpredictableBool(which : Unpredictable) => boolean
begin
    let c : Constraint = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
end;
```

J1.4.3.470 ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

impdef func ConstrainUnpredictableInteger(low : integer,
                                          high : integer,
                                          which : Unpredictable) => (Constraint,integer)
begin
    let c : Constraint = ConstrainUnpredictable(which);
    if c == Constraint_UNKNOWN then
        return (c, low);                // See notes; this is an example implementation only
    else
        return (c, ARBITRARY : integer); // integer result not used
    end;
end;
```

J1.4.3.471 ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =================================
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

impdef func ConstrainUnpredictableProcedure(which : Unpredictable)
begin
    case which of
        when Unpredictable_PMUEVENTCOUNTER =>
            Undefined();
        when Unpredictable_MRC_APSR_TARGET =>
            PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
        otherwise =>
            ImpDef("IMPLEMENTATION_DEFINED");
    end;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.472 Constraint

```
// Constraint
// ==========
// List of Constrained Unpredictable behaviors.

type Constraint of enumeration {
                          // General
                          Constraint_NONE,              // Instruction executes with
                                                        // no change or side-effect
                                                        // to its described behavior
                          Constraint_UNKNOWN,           // Destination register
                                                        // has UNKNOWN value
                          Constraint_UNDEF,             // Instruction is UNDEFINED
                          Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                          Constraint_NOP,               // Instruction executes as NOP
                          Constraint_TRUE,
                          Constraint_FALSE,
                          Constraint_DISABLED,
                          Constraint_UNCOND,            // Instruction executes unconditionally
                          Constraint_COND,              // Instruction executes conditionally
                          Constraint_ADDITIONAL_DECODE, // Instruction executes
                                                        // with additional decode
                          // Load-store
                          Constraint_WBSUPPRESS,
                          Constraint_FAULT,
                          Constraint_LIMITED_ATOMICITY, // Accesses are not
                                                        // single-copy atomic
                                                        // above the byte level
                          Constraint_NVNV1_00,
                          Constraint_NVNV1_01,
                          Constraint_NVNV1_11,
                          Constraint_EL1TIMESTAMP,      // Constrain to Virtual Timestamp
                          Constraint_EL2TIMESTAMP,      // Constrain to Virtual Timestamp
                          Constraint_OSH,               // Constrain to Outer Shareable
                          Constraint_ISH,               // Constrain to Inner Shareable
                          Constraint_NSH,               // Constrain to Nonshareable

                          Constraint_NC,                // Constrain to Noncacheable
                          Constraint_WT,                // Constrain to Writethrough
                          Constraint_WB,                // Constrain to Writeback
                          Constraint_RESS,              // Bits behave as RESS for all purposes
                                                        // other than reading back the register
                          Constraint_ALLRESS,           // Bits behave as RESS for all purposes

                          // IPA too large
                          Constraint_FORCE, Constraint_FORCENOSLCHECK,
                          // An unallocated System register value maps onto an allocated value
                          Constraint_MAPTOALLOCATED,
                          // PMSCR_PCT reserved values select Virtual timestamp
                          Constraint_PMSCR_PCT_VIRT,
                          Constraint_ANYREG,            // Any allocated register is chosen
};
```

J1.4.3.473 Unpredictable

```
// Unpredictable
// =============
// List of Constrained Unpredictable situations.

type Unpredictable of enumeration {
                          // VMSR on MVFR
                          Unpredictable_VMSR,
                          // Writeback/transfer register overlap (load)
                          Unpredictable_WBOVERLAPLD,
                          // Writeback/transfer register overlap (store)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Unpredictable_WBOVERLAPST,
// Load Pair transfer register overlap
Unpredictable_LDPOVERLAP,
// Store-exclusive base/status register overlap
Unpredictable_BASEOVERLAP,
// Store-exclusive data/status register overlap
Unpredictable_DATAOVERLAP,
// Load-store alignment checks
Unpredictable_DEVPAGE2,
// Instruction fetch from Device memory
Unpredictable_INSTRDEVICE,
// Reserved CPACR value
Unpredictable_RESCPACR,
// Reserved MAIR value
Unpredictable_RESMAIR,
// Effect of SCTLR_ELx.C on Tagged attribute
Unpredictable_S1CTAGGED,
// Reserved Stage 2 MemAttr value
Unpredictable_S2RESMEMATTR,
// Reserved TEX::C::B value
Unpredictable_RESTEXCB,
// Reserved PRRR value
Unpredictable_RESPRRR,
// Reserved DACR field
Unpredictable_RESDACR,
// Reserved VTCR.S value
Unpredictable_RESVTCRS,
// Reserved TCR.TnSZ value
Unpredictable_RESTnSZ,
// Reserved SCTLR_ELx.TCF value
Unpredictable_RESTCF,
// Tag stored to Device memory
Unpredictable_DEVICETAGSTORE,
// Out-of-range TCR.TnSZ value
Unpredictable_OORTnSZ,

// IPA size exceeds PA size
Unpredictable_LARGEIPA,
// Syndrome for a known-passing conditional A32 instruction
Unpredictable_ESRCONDPASS,
// Illegal State exception: zero PSTATE.IT
Unpredictable_ILZEROIT,
// Illegal State exception: zero PSTATE.T
Unpredictable_ILZEROT,
// Debug: prioritization of Vector Catch
Unpredictable_BPVECTORCATCHPRI,
// Debug Vector Catch: match on 2nd halfword
Unpredictable_VCMATCHHALF,
// Debug Vector Catch: match on Data Abort
// or Prefetch abort
Unpredictable_VCMATCHDAPA,
// Debug watchpoints: nonzero MASK and non-ones BAS
Unpredictable_WPMASKANDBAS,
// Debug watchpoints: non-contiguous BAS
Unpredictable_WPBASCONTIGUOUS,
// Debug watchpoints: reserved MASK
Unpredictable_RESWPMASK,
// Debug watchpoints: nonzero MASKed bits of address
Unpredictable_WPMASKEDBITS,
// Debug breakpoints and watchpoints: reserved control bits
Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints and watchpoints: reserved MDSELR_EL1.BANK
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Unpredictable_RESMDSELR,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug breakpoints: a breakpoint is linked to that is not
// programmed with linking enabled
Unpredictable_BPLINKINGDISABLED,
// Debug breakpoints: reserved MASK
Unpredictable_RESBPMASK,
// Debug breakpoints: MASK is set for a Context matching
// breakpoint or when DBGBCR_EL1(n).BAS != '1111'
Unpredictable_BPMASK,
// Debug breakpoints: nonzero MASKed bits of address
Unpredictable_BPMASKEDBITS,
// Debug breakpoints: A linked breakpoint is
// linked to an address matching breakpoint
Unpredictable_BPLINKEDADDRMATCH,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to
// AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking
// branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
// Zero top bits of ZA registers in EL change
Unpredictable_SMEZEROUPPER,
// Watchpoint match of last rounded up memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDUPACCESS,
// Watchpoint match of first rounded down memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDDOWNACCESS,
// HCR_EL2.[NV,NV1] == '01'
Unpredictable_NVNV1,
// Upper bits of a BADDR are not RESS
Unpredictable_BADDR_RESS,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Dirty Bit State Update by HW
Unpredictable_DBUPDATE,
// Unsupported memory attributes for HW update
Unpredictable_Unsupported_Atomic_HW_Update,
// Consider SCTLR_ELx().IESB in Debug state
Unpredictable_IESBinDebug,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// The HCR_EL2.TERR / HCR2.TERR and SCR_EL3.TERR / SCR.TERR
// trap control can be treated as RES0.
Unpredictable_MINIMALRASv2,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
 // Reserved MDCR_EL3.[NSTBE,NSTB] or MDCR_EL3.[NSPBE,NSPB] value
 Unpredictable_RESERVEDNSxB,
 Unpredictable_RESERVEDNSxB_Trap,
// WFET or WFIT instruction in Debug state
Unpredictable_WFxTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Unaligned exclusives, atomics, acquire/release
// to a region that is not to Normal inner write-back
// outer write-back shareable generate an Alignment fault.
Unpredictable_LSE2_ALIGNMENT_FAULT,
// 128-bit Atomic or 128-bit RCW{S} transfer register overlap
Unpredictable_LSE128OVERLAP,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in
// A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGBVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behavior
Unpredictable_PMSCR_PCT,
// MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
// FEAT_HPMN0 is not implemented and HPMN is 0.
Unpredictable_RES_HPMN,
// Chained PMU counters idx and idx+1 are not in same range
Unpredictable_COUNT_CHAIN,
// PMCCR.EPMN is larger than PMCR.N
Unpredictable_RES_EPMN,
// Generate BRB_FILTRATE event on BRB injection
Unpredictable_BRBFILTRATE,
// Generate PMU_SNAPSHOT event in Debug state
Unpredictable_PMUSNAPSHOTEVENT,
// Reserved MDCR_EL3.EPMSSAD value
Unpredictable_RESEPMSSAD,
// Reserved PMECR_EL1.SSE value
Unpredictable_RESPMSSE,
// Enable for PMU Profiling exception and PMUIRQ
Unpredictable_RESPMEE,
// Enables for SPE Profiling exceptions and PMSIRQ
Unpredictable_RESPMSEE,
// Enables for TRBE Profiling exceptions and PMSIRQ
Unpredictable_RESTRFEE,
// Operands for CPY*/SET* instructions overlap
Unpredictable_MOPSOVERLAP,
// Operands for CPY*/SET* instructions use 0b11111
// as a register specifier
Unpredictable_MOPS_R31,
// Chooses which value to return in a non failed Atomic Compare and Swap
Unpredictable_CASRETURNOLDVALUE,
// Enables write of the newvalue in a failed Atomic Compare and Swap
Unpredictable_WRITEFAILEDCAS,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                // Enables write of the newvalue in a failed Atomic Compare and Swap
                                // or failed RCW{S}
                                Unpredictable_WRITEFAILEDRCWCAS,
                                // Store-only Tag checking on a failed Atomic Compare and Swap
                                Unpredictable_STRONLYTAGCHECKEDCAS,
                                // Store-only Tag checking on a failed RCW(S) checks
                                // or RCW(S) Atomic Compare and Swap
                                Unpredictable_STRONLYTAGCHECKEDRCWSCAS,
                                // Reserved MDCR_EL3.ETBAD value
                                Unpredictable_RES_ETBAD,
                                // Invalid Streaming Mode filter bits
                                Unpredictable_RES_PMU_VS,
                                // Apply Checked Pointer Arithmetic on a sequential access to bytes
                                // that cross the 0xXXFF_FFFF_FFFF_FFFF boundary.
                                Unpredictable_CPACHECK,
                                // Reserved PMEVTYPER[n]_EL0.[TC,TE,TC2] values
                                Unpredictable_RESTC,
                                // When FEAT_MTE is implemented, if Memory access mode is enabled
                                // and PSTATE.TCO is 0,  Reads and writes to the external debug
                                // interface DTR registers are CONSTRAINED UNPREDICTABLE for tagcheck
                                Unpredictable_NODTRTAGCHK,
                                // Use the default PMG when the default PARTID is generated
                                // due to MPAM error
                                Unpredictable_USE_DEFAULT_PMG,
                                // If the atomic instructions are not atomic in regard to other
                                // agents that access memory, then the instruction can have one or
                                // more of the following effects
                                Unpredictable_Atomic_SYNC_ABORT,
                                Unpredictable_Atomic_SERROR,
                                Unpredictable_Atomic_NOP,
                                Unpredictable_Atomic_MMU_IMPDEF_FAULT,
                                // Accessing DBGDSCRint via MRC in debug state
                                Unpredictable_MRC_APSR_TARGET,
                                // Accessing Banked register not accessible from the PE mode
                                Unpredictable_BankedRegister,
                                // Accessing unimplemented Banked register
                                Unpredictable_UnimplementedRegister,
                                // Reserved ERRACR.{RLRA,SRA,NSRA} values
                                Unpredictable_RES_RA,
                                // Linked target does not support linking
                                Unpredictable_BPNOTLINKING
};
```

J1.4.3.474 AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

func AdvSIMDExpandImm(op : bit, cmode : bits(4), imm8 : bits(8)) => bits(64)
begin
    var imm64 : bits(64);
    case cmode[3:1] of
        when '000' =>
            imm64 = Replicate{64}(Zeros{24}::imm8);
        when '001' =>
            imm64 = Replicate{64}(Zeros{16}::imm8::Zeros{8});
        when '010' =>
            imm64 = Replicate{64}(Zeros{8}::imm8::Zeros{16});
        when '011' =>
            imm64 = Replicate{64}(imm8::Zeros{24});
        when '100' =>
            imm64 = Replicate{64}(Zeros{8}::imm8);
        when '101' =>
            imm64 = Replicate{64}(imm8::Zeros{8});
        when '110' =>
            if cmode[0] == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                imm64 = Replicate{64}(Zeros{16}::imm8::Ones{8});
            else
                imm64 = Replicate{64}(Zeros{8}::imm8::Ones{16});
            end;
        when '111' =>
            if cmode[0] == '0' && op == '0' then
                imm64 = Replicate{64}(imm8);
            end;
            if cmode[0] == '0' && op == '1' then
                let imm8a : bits(8) = Replicate{}(imm8[7]);
                let imm8b : bits(8) = Replicate{}(imm8[6]);
                let imm8c : bits(8) = Replicate{}(imm8[5]);
                let imm8d : bits(8) = Replicate{}(imm8[4]);
                let imm8e : bits(8) = Replicate{}(imm8[3]);
                let imm8f : bits(8) = Replicate{}(imm8[2]);
                let imm8g : bits(8) = Replicate{}(imm8[1]);
                let imm8h : bits(8) = Replicate{}(imm8[0]);
                imm64 = imm8a::imm8b::imm8c::imm8d::imm8e::imm8f::imm8g::imm8h;
            end;
            if cmode[0] == '1' && op == '0' then
                let imm32 : bits(32) = (imm8[7]::NOT(imm8[6])::Replicate{5}(imm8[6])::
                                        imm8[5:0]::Zeros{19});
                imm64 = Replicate{64}(imm32);
            end;
            if cmode[0] == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding(); end;
                imm64 = imm8[7]::NOT(imm8[6])::Replicate{8}(imm8[6])::imm8[5:0]::Zeros{48};
            end;
    end;

    return imm64;
end;
```

J1.4.3.475 MatMulAdd

```
// MatMulAdd()
// ===========
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

func MatMulAdd(addend : bits(128), op1 : bits(128), op2 : bits(128), op1_unsigned : boolean,
             op2_unsigned : boolean) => bits(128)
begin

    var result : bits(128);
    var sum : bits(32);
    var prod : integer;

    for i = 0 to 1 do
        for j = 0 to 1 do
            sum = addend[(2*i + j)*:32];
            for k = 0 to 7 do
                let opelt1 : bits(8) = op1[(8*i + k)*:8];
                let opelt2 : bits(8) = op2[(8*j + k)*:8];
                let element1 : integer = if op1_unsigned then UInt(opelt1) else SInt(opelt1);
                let element2 : integer = if op2_unsigned then UInt(opelt2) else SInt(opelt2);
                prod = element1 * element2;
                sum  = sum + prod;
            end;
            result[(2*i + j)*:32] = sum;
        end;
    end;

    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.476 PolynomialMult

```
// PolynomialMult()
// ================

func PolynomialMult{M, N}(op1 : bits(M), op2 : bits(N)) => bits(M+N)
begin
    var result : bits(N + M) = Zeros{M+N};
    let extended_op2 : bits(N + M) = ZeroExtend{M+N}(op2);
    for i=0 to M-1 do
        if op1[i] == '1' then
            result = result XOR LSL(extended_op2, i);
        end;
    end;
    return result;
end;
```

J1.4.3.477 SatQ

```
// SatQ()
// ======

func SatQ{N}(i : integer, unsigned : boolean) => (bits(N), boolean)
begin
    let (result, sat) : (bits(N), boolean) = (if unsigned then UnsignedSatQ{N}(i)
                                                          else SignedSatQ{N}(i));
    return (result, sat);
end;
```

J1.4.3.478 ShiftSat

```
// ShiftSat()
// ==========

func ShiftSat(shift : integer, esize : integer) => integer
begin
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1); end;
    return shift;
end;
```

J1.4.3.479 SignedSat

```
// SignedSat()
// ===========

func SignedSat{N}(i : integer) => bits(N)
begin
    let (result, -) = SignedSatQ{N}(i);
    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.480 SignedSatQ

```
// SignedSatQ()
// ============

func SignedSatQ{N}(i : integer) => (bits(N), boolean)
begin
    var result : integer;
    var saturated : boolean;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    end;
    return (result[N-1:0], saturated);
end;
```

J1.4.3.481 UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

func UnsignedRSqrtEstimate{N}(operand : bits(N)) => bits(N)
begin
    assert N == 32;
    var result : bits(N);
    if operand[N-1:N-2] == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones{N};
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        let increasedprecision : boolean = FALSE;
        let estimate : integer = RecipSqrtEstimate(UInt(operand[31:23]), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate[8:0] :: Zeros{N-9};
    end;

    return result;
end;
```

J1.4.3.482 UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

func UnsignedRecipEstimate{N}(operand : bits(N)) => bits(N)
begin
    assert N == 32;
    var result : bits(N);
    if operand[N-1] == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones{N};
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        let increasedprecision : boolean = FALSE;
        let estimate : integer = RecipEstimate(UInt(operand[31:23]), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate[8:0] :: Zeros{N-9};
    end;

    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.3.483 UnsignedSat

```
// UnsignedSat()
// =============

func UnsignedSat{N}(i : integer) => bits(N)
begin
    let (result, -) = UnsignedSatQ{N}(i);
    return result;
end;
```

J1.4.3.484 UnsignedSatQ

```
// UnsignedSatQ()
// ==============

func UnsignedSatQ{N}(i : integer) => (bits(N), boolean)
begin
    var result : integer;
    var saturated : boolean;
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    end;
    return (result[N-1:0], saturated);
end;
```

J1.4.4 shared/trace

This section includes the following pseudocode functions:

- DebugMemWrite
- DebugWriteExternalAbort
- DebugWriteFault
- GetTimestamp
- PhysicalOffsetIsValid
- TRBCRManStopWrite
- BranchNotTaken
- AllowExternalTraceBufferAccess
- CheckForTRBEException
- CheckMDCR_EL3_NSTBTrap
- CollectTrace
- DefaultTRBEEvent
- DetectedTraceTrigger
- EffectiveMDCR_EL3_NSTB
- EffectiveTRBLIMITR_EL1_nVM
- EffectiveTRFCR_EL1_EE
- EffectiveTRFCR_EL2_EE
- ElementStreamSize
- GetTRBSR_EL1_FSC
- GetTRBSR_EL2_FSC
- GetTRBSR_EL3_FSC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- HaveImpDefTraceOutput
- ImpDefTraceOutput
- OtherTRBEManagementEvent
- ReportTRBEEvent
- ReportTRBEManagementEvent
- TRBEInternalBufferFull
- TRBEInterruptEnabled
- TRBE_TRBIDR_P_Read
- TRBSR_EL
- TraceBufferEnabled
- TraceBufferOwner
- TraceBufferRunning
- TraceUnitFlushOnTriggerComplete
- TryAssertTRBIRQ
- TraceInstrumentationAllowed
- TraceUnitFlush
- EffectiveE0HTRE
- EffectiveE0TRE
- EffectiveE1TRE
- EffectiveE2TRE
- SelfHostedTraceEnabled
- TraceAllowed
- TraceContextIDR2
- TraceSynchronizationBarrier
- TraceTimeStamp
- IsTraceCorePowered

J1.4.4.1 DebugMemWrite

```
// DebugMemWrite()
// ===============
// Write data to memory one byte at a time. Starting at the passed virtual address.
// Used by SPE and TRBE.

func DebugMemWrite(vaddress : bits(64), accdesc : AccessDescriptor,
                   aligned : boolean, data : bits(8)) => (PhysMemRetStatus, AddressDescriptor)
begin

    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;

    // Translate virtual address
    var addrdesc : AddressDescriptor;
    let size : integer = 1;
    addrdesc = AArch64_TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(addrdesc) then
        return (memstatus, addrdesc);
    end;

    memstatus = PhysMemWrite{8}(addrdesc, accdesc, data);

    return (memstatus, addrdesc);
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.4.2 DebugWriteExternalAbort

```
// DebugWriteExternalAbort()
// =========================
// Populate the syndrome register for an External abort caused by a call of DebugMemWrite().

func DebugWriteExternalAbort(memstatus : PhysMemRetStatus, addrdesc : AddressDescriptor,
                             start_vaddr : bits(64))
begin

    let iswrite : boolean = TRUE;

    var handle_as_SError : boolean = FALSE;
    case addrdesc.fault.accessdesc.acctype of
        when AccessType_SPE =>
            handle_as_SError = ImpDefBool("Report SPE ExtAbort as SError");
        when AccessType_TRBE =>
            handle_as_SError = ImpDefBool("Report TRBE ExtAbort as SError");
        otherwise =>
            unreachable;
    end;

    let ttw_abort : boolean = addrdesc.fault.statuscode IN {Fault_SyncExternalOnWalk,
                                                            Fault_SyncParityOnWalk};
    let statuscode : Fault = (if ttw_abort then addrdesc.fault.statuscode
                              else memstatus.statuscode);

    if statuscode IN {Fault_AsyncExternal, Fault_AsyncParity} || handle_as_SError then
        // Report the abort as an SError
        var fault : FaultRecord = NoFault();
        let parity : boolean = statuscode IN {Fault_SyncParity, Fault_AsyncParity,
                                              Fault_SyncParityOnWalk};
        fault.statuscode = if parity then Fault_AsyncParity else Fault_AsyncExternal;
        if IsFeatureImplemented(FEAT_RAS) then
            fault.merrorstate = memstatus.merrorstate;
        end;
        let extflag : bit = if ttw_abort then addrdesc.fault.extflag else memstatus.extflag;
        fault.extflag = extflag;
        fault.accessdesc.acctype = addrdesc.fault.accessdesc.acctype;
        PendSErrorInterrupt(fault);
        return;
    end;

    // Generate a buffer management event, modifying the existing syndrome.
    var handle_async : boolean = FALSE;
    var syndrome : bits(64);
    case addrdesc.fault.accessdesc.acctype of
        when AccessType_SPE =>
            handle_async = ImpDefBool("Report SPE ExtAbort asynchronously");
            assert !IsFeatureImplemented(FEAT_SPE_EXC);
            syndrome = PMBSR_EL1();
        when AccessType_TRBE =>
            handle_async = ImpDefBool("Report TRBE ExtAbort asynchronously");
            assert !IsFeatureImplemented(FEAT_TRBE_EXC);
            syndrome = TRBSR_EL1();
        otherwise =>
            unreachable;
    end;

    var ec : bits(6);
    if (IsFeatureImplemented(FEAT_RME) && addrdesc.fault.gpcf.gpf != GPCF_None &&
          addrdesc.fault.gpcf.gpf != GPCF_Fail) then
        ec = '011110';
    else
        ec = if addrdesc.fault.secondstage then '100101' else '100100';
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    let mss2 : bits(24) = Zeros{};
    var mss : bits(16) = Zeros{};
    if handle_async then                // FSC bits
        mss[5:0] = '010001';
    else
        mss[5:0] = EncodeLDFSC(statuscode, addrdesc.fault.level);
    end;

    // The following values are always updated in the syndrome register.
    if (addrdesc.fault.accessdesc.acctype == AccessType_SPE &&
          (handle_async || start_vaddr != addrdesc.vaddress)) then
        syndrome[19] = '1';             // DL bit (SPE only)
    end;

    syndrome[18] = '1';                 // EA bit

    // The following values are not modified if a previous buffer management event
    // has not been handled. Note that in this simple sequential model, this test
    // will never fail.
    if syndrome[17] == '0' then         // Check previous 'S' bit.
        syndrome[55:32] = mss2;         // MSS2 bits
        syndrome[31:26] = ec;           // EC bits
        if addrdesc.fault.accessdesc.acctype == AccessType_TRBE then
            syndrome[22] = '1';         // IRQ bit (TRBE only)
        end;
        syndrome[17] = '1';             // S bit
        syndrome[15:0] = mss;           // MSS bits
    end;

    case addrdesc.fault.accessdesc.acctype of
        when AccessType_SPE =>
            PMBSR_EL1() = syndrome;
        when AccessType_TRBE =>
            TRBSR_EL1() = syndrome;
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.4.3 DebugWriteFault

```
// DebugWriteFault()
// =================
// Populate the syndrome register for a fault caused by a call of DebugMemWrite().

func DebugWriteFault(vaddress : bits(64), fault : FaultRecord)
begin
    var ec : bits(6);
    if (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
          fault.gpcf.gpf != GPCF_Fail) then
        ec = '011110';
    else
        ec = if fault.secondstage then '100101' else '100100';
    end;

    var mss2 : bits(24) = Zeros{};
    if fault.statuscode == Fault_Permission then
        mss2[8] = if fault.toplevel then '1' else '0';      // TopLevel bit
        mss2[7] = if fault.assuredonly then '1' else '0';   // AssuredOnly bit
        mss2[6] = if fault.overlay then '1' else '0';       // Overlay bit
        mss2[5] = if fault.dirtybit then '1' else '0';      // DirtyBit
    end;


    var mss : bits(16) = Zeros{};
    if !(IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            fault.gpcf.gpf != GPCF_Fail) then
        mss[5:0] = EncodeLDFSC(fault.statuscode, fault.level);  // FSC bits
    end;

    // Generate a buffer management event, modifying the existing syndrome.
    var target_el : bits(2);
    var syndrome : bits(64);
    case fault.accessdesc.acctype of
        when AccessType_SPE =>
            target_el = ReportSPEEvent(ec, mss[5:0]);
            syndrome = PMBSR_EL(target_el);
        when AccessType_TRBE =>
            target_el = ReportTRBEEvent(ec, mss[5:0]);
            syndrome = TRBSR_EL(target_el);
        otherwise =>
            unreachable;
    end;

    // The following values are not modified if a previous buffer management event
    // has not been handled. Note that in this simple sequential model, this test
    // will never fail.
    if syndrome[17] == '0' then         // Check previous 'S' bit.
        syndrome[55:32] = mss2;         // MSS2 bits
        syndrome[31:26] = ec;           // EC bits
        if fault.accessdesc.acctype == AccessType_TRBE then
            syndrome[22] = '1';         // IRQ bit (TRBE only)
        end;
        syndrome[17] = '1';             // S bit
        syndrome[15:0] = mss;           // MSS bits
    end;

    // For SPE, PMBPTR_EL1 points to the address that generated the fault, and writing
    // to memory never started. Therefore, there isno data loss and DL is unchanged.

    case fault.accessdesc.acctype of
        when AccessType_SPE =>
            PMBSR_EL(target_el) = syndrome;

        when AccessType_TRBE =>
            TRBSR_EL(target_el) = syndrome;

        otherwise =>
            unreachable;
    end;

    return;
end;
```

J1.4.4.4 GetTimestamp

```
// GetTimestamp()
// ==============
// Returns the Timestamp depending on the type

func GetTimestamp(timeStampType : TimeStamp) => bits(64)
begin
    case timeStampType of
        when TimeStamp_Physical =>
            return PhysicalCountInt();
        when TimeStamp_Virtual =>
            return PhysicalCountInt() - CNTVOFF_EL2();
        when TimeStamp_OffsetPhysical =>
            let physoff : bits(64) = if PhysicalOffsetIsValid() then CNTPOFF_EL2() else Zeros{64};
            return PhysicalCountInt() - physoff;
        when TimeStamp_None =>
            return Zeros{64};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when TimeStamp_CoreSight =>
            return ImpDefBits{64}("CoreSight timestamp");
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.4.5 PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =======================
// Returns whether the Physical offset for the timestamp is valid

func PhysicalOffsetIsValid() => boolean
begin
    if !HaveAArch64() then
        return FALSE;
    elsif !HaveEL(EL2) || !IsFeatureImplemented(FEAT_ECV_POFF) then
        return FALSE;
    elsif HaveEL(EL3) && EffectiveSCR_EL3_NS() == '1' && EffectiveSCR_EL3_RW() == '0' then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3().ECVEn == '0' then
        return FALSE;
    elsif CNTHCTL_EL2().ECV == '0' then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

J1.4.4.6 TRBCRManStopWrite

```
// TRBCRManStopWrite()
// ===================
// Called on a write of 1 to TRBCR.ManStop.

func TRBCRManStopWrite()
begin
    TraceUnitFlush();
    OtherTRBEManagementEvent('000011');
    TryAssertTRBIRQ();
end;
```

J1.4.4.7 BranchNotTaken

```
// BranchNotTaken()
// ================
// Called when a branch is not taken.

func BranchNotTaken(branchtype : BranchType, branch_conditional : boolean)
begin
    let branchtaken : boolean = FALSE;
    if IsFeatureImplemented(FEAT_SPE) then
        SPEBranch{64}(ARBITRARY : bits(64), branchtype, branch_conditional, branchtaken);
    end;
    return;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.4.8 AllowExternalTraceBufferAccess

```
// AllowExternalTraceBufferAccess()
// ================================
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed for the given Security state, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

func AllowExternalTraceBufferAccess(addrdesc : AddressDescriptor) => boolean
begin
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    // FEAT_Debugv8p4 is always implemented when FEAT_TRBE_EXT is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    var etbad : bits(2) = if HaveEL(EL3) then MDCR_EL3().ETBAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01','10'} then
        (-, etbad) = ConstrainUnpredictableBits{2}(Unpredictable_RES_ETBAD);
        // The value returned by ConstrainUnpredictableBits must be a
        // non-reserved value
        assert etbad IN {'00', '11'};
    end;

    case etbad of
        when '00' =>
            if IsFeatureImplemented(FEAT_RME) then
                return addrdesc.paddress.paspace == PAS_Root;
            else
                return addrdesc.paddress.paspace == PAS_Secure;
            end;
        when '01' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
        when '10' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
        when '11' =>
            return TRUE;
    end;
end;
```

J1.4.4.9 CheckForTRBEException

```
// CheckForTRBEException()
// =======================
// Take a TRBE Profiling exception if pending, permitted, and unmasked.

func CheckForTRBEException()
begin
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return;
    end;

    if Halted() || Restarting() then
        return;
    end;

    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;
    var route_to_el1 : boolean = FALSE;

    if HaveEL(EL3) && MDCR_EL3().TRBEE == '1x' then
        let pending : boolean = TRBSR_EL3().IRQ == '1';
        let masked  : boolean = PSTATE.EL       == EL3;
        route_to_el3 = pending && !masked;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;

    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = TraceBufferOwner();
    let in_owning_ss : boolean = IsCurrentSecurityState(owning_ss);

    if EffectiveTRFCR_EL2_EE() IN {'1x'} then
        let pending : boolean = TRBSR_EL2().IRQ == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL == EL3 ||
                                    (PSTATE.EL == EL2 && (TRFCR_EL2().EE != '11' ||
                                                          TRFCR_EL2().KE == '0' ||
                                                          PSTATE.PM == '1')));
        route_to_el2 = pending && !masked;
    end;

    if EffectiveTRFCR_EL1_EE() == '11' then
        let pending : boolean = TRBSR_EL1().IRQ == '1';
        let masked  : boolean = (!in_owning_ss || PSTATE.EL IN {EL3, EL2} ||
                                    (PSTATE.EL == EL1 && (TRFCR_EL1().KE == '0' ||
                                                          PSTATE.PM == '1')));
        if EffectiveTGE() == '1' then
            route_to_el2 = route_to_el2 || (pending && !masked);
        else
            route_to_el1 = pending && !masked;
        end;
    end;

    let fsc : bits(5) = '00010';        // TRBE exception
    let synchronous : boolean = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    end;
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    end;
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
    end;
end;
```

J1.4.4.10 CheckMDCR_EL3_NSTBTrap

```
// CheckMDCR_EL3_NSTBTrap()
// ========================
// Check if the register access is trappable by MDCR_EL3.[NSTBE, NSTB]

func CheckMDCR_EL3_NSTBTrap() => boolean
begin
    var state_bits : bits(3);
    var reserved   : boolean;
    (state_bits, reserved) = EffectiveMDCR_EL3_NSTB();
    return ((reserved && ConstrainUnpredictableBool(Unpredictable_RESERVEDNSxB_Trap)) ||
             state_bits[0] == '0' || state_bits[1] != EffectiveSCR_EL3_NS() ||
             (IsFeatureImplemented(FEAT_RME) && state_bits[2] != EffectiveSCR_EL3_NSE()));
end;
```

J1.4.4.11 CollectTrace

```
// CollectTrace()
// ==============
// Called for each byte generated by the trace unit.
// Returns TRUE if the Trace Buffer Unit accepts or discards the trace
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// data, and FALSE if the Trace Buffer Unit rejects the trace data.

func CollectTrace(datum : bits(8)) => boolean recurselimit ElementStreamSize
begin
    if !TraceBufferEnabled() then  // Trace buffer disabled
        // 'datum' is discarded
        if HaveImpDefTraceOutput() then
            return ImpDefTraceOutput(datum);
        else
            return TRUE;           // Discard the trace byte
        end;
    end;

    // If the TRBE cannot accept the trace data, it must return FALSE
    if TRBEInternalBufferFull() then
        return FALSE;
    end;

    if TraceBufferRunning() then   // Accept the data
        let address            : bits(64) = TRBPTR_EL1();
        var ttw_abort          : boolean = FALSE;
        let ttw_abort_as_fault : boolean = (ImpDefBool(
                                            "Report TRBE ExtAbort on TTW as fault"));
        var addrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        if !SelfHostedTraceEnabled() then
            // The Trace Buffer Unit is using External mode.
            if IsFeatureImplemented(FEAT_RME) && !ExternalRootInvasiveDebugEnabled() then
                if IsZero(GPCCR_EL3().[TBGPCD, GPC]) then
                    return FALSE;
                end;
            end;

            let pas : bits(2) = TRBMAR_EL1().PAS;
            let paspace : PASpace = DecodePASpace('0', pas[1], pas[0]);
            var valid_config : boolean = IsPASValid(pas) && InvasiveDebugPermittedPAS(paspace);
            if IsFeatureImplemented(FEAT_TRBE_MPAM) && TRBMPAM_EL1().EN == '1' then
                let mpam_sp : bits(2) = TRBMPAM_EL1().MPAM_SP;
                let mpam_pa : PASpace = DecodePASpace('0', mpam_sp[1], mpam_sp[0]);
                valid_config = (valid_config && IsPASValid(mpam_sp) &&
                                InvasiveDebugPermittedPAS(mpam_pa));
            end;

            if !valid_config then
                OtherTRBEManagementEvent('000000');
                TryAssertTRBIRQ();
                return TRUE;
            end;

            let el : bits(2) = ARBITRARY : bits(2);
            let ss : SecurityState = ARBITRARY : SecurityState;
            let accdesc : AccessDescriptor = CreateAccDescTRBE(ss, el);

            var pa : FullAddress;
            pa.address = address[NUM_PABITS-1:0];
            pa.paspace = paspace;
            let memattrs : MemoryAttributes = S1DecodeMemAttrs(TRBMAR_EL1().Attr,
                                                               TRBMAR_EL1().SH, TRUE);
            addrdesc = CreateAddressDescriptor(pa, memattrs, accdesc);
            addrdesc.mecid = DEFAULT_MECID;
            if IsFeatureImplemented(FEAT_RME) && !ExternalRootInvasiveDebugEnabled() then
                let gpcf : GPCFRecord = GranuleProtectionCheck(addrdesc, accdesc);
                if gpcf.gpf == GPCF_None then
                    memstatus = PhysMemWrite{8}(addrdesc, accdesc, datum);
                else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                addrdesc.fault.gpcf       = gpcf;
                addrdesc.fault.statuscode = Fault_GPCFOnOutput;
            end;
        else
            memstatus = PhysMemWrite{8}(addrdesc, accdesc, datum);
        end;
    else
        // The Trace Buffer Unit is using Self-hosted mode.
        var owning_ss : SecurityState;
        var owning_el : bits(2);
        (owning_ss, owning_el) = TraceBufferOwner();
        let accdesc : AccessDescriptor = CreateAccDescTRBE(owning_ss, owning_el);

        let aligned : boolean = TRUE;
        (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned, datum);

        ttw_abort = addrdesc.fault.statuscode IN {Fault_SyncExternalOnWalk,
                                                  Fault_SyncParityOnWalk};
    end;

    if IsFault(addrdesc.fault.statuscode) && (!ttw_abort || ttw_abort_as_fault) then
        DebugWriteFault(address, addrdesc.fault);
        TryAssertTRBIRQ();
        return TRUE;
    elsif IsFault(memstatus) || (ttw_abort && !ttw_abort_as_fault) then
        DebugWriteExternalAbort(memstatus, addrdesc, address);
        TryAssertTRBIRQ();
        return TRUE;
    end;

    // Check for Trigger Event
    let target_el : bits(2) = DefaultTRBEEvent();
    let triggered : boolean = TRBSR_EL(target_el).TRG == '1';
    if triggered && !IsZero(TRBTRG_EL1().TRG) then
        TRBTRG_EL1().TRG = (TRBTRG_EL1().TRG - 1)[31:0];
        if IsZero(TRBTRG_EL1().TRG) && TRBLIMITR_EL1().TM != '11' then
            TraceUnitFlush();
            TraceUnitFlushOnTriggerComplete();
        end;
    end;

    // Increment the pointer
    var next_address : bits(64) = TRBPTR_EL1() + 1;
    if next_address[63:12] == TRBLIMITR_EL1().LIMIT then
        next_address = TRBBASER_EL1().BASE::Zeros{12};
        TRBSR_EL(target_el).WRAP = '1';
        CTI_SignalEvent(CrossTriggerIn_TRBEWrap);
        if TRBLIMITR_EL1().FM == '00' then     // Fill mode
            let bsc : bits(6) = '000001'; // Buffer full event
            OtherTRBEManagementEvent(bsc);
        elsif TRBLIMITR_EL1().FM != '11' then  // Not Circular Buffer mode
            if TRBSR_EL(target_el).IRQ == '0' then
                TRBSR_EL(target_el).IRQ = '1';   // Assert interrupt or exception
                CTI_SignalEvent(CrossTriggerIn_TRBEMgmt);
            end;
        end;
    end;
    TRBPTR_EL1() = next_address[63:0];

    TryAssertTRBIRQ();
  end;
  return TRUE;
end;
```

J1.4.4.12 DefaultTRBEEvent

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// DefaultTRBEEvent()
// ==================
// Return the target ELx for an indirect write to TRBSR_ELx for an Other buffer management
// event or anything other than a buffer management event.

func DefaultTRBEEvent() => bits(2)
begin
    return ReportTRBEEvent(Zeros{6}, ARBITRARY : bits(6));
end;
```

J1.4.4.13 DetectedTraceTrigger

```
// DetectedTraceTrigger()
// ======================
// Called when the trace unit detects a trace trigger

func DetectedTraceTrigger()
begin
    if TraceBufferRunning() then
        let target_el : bits(2) = DefaultTRBEEvent();
        if TRBSR_EL(target_el).TRG == '0' then
            TRBSR_EL(target_el).TRG = '1';
            if IsZero(TRBTRG_EL1().TRG) && TRBLIMITR_EL1().TM != '11' then
                TraceUnitFlush();
                TraceUnitFlushOnTriggerComplete();
            end;
        end;
    end;
end;
```

J1.4.4.14 EffectiveMDCR_EL3_NSTB

```
// EffectiveMDCR_EL3_NSTB()
// ========================
// Return the Effective value of MDCR_EL3().[NSTBE, NSTB] field and whether it is a reserved value.

func EffectiveMDCR_EL3_NSTB() => (bits(3), boolean)
begin
    var state_bits : bits(3);
    var reserved   : boolean = FALSE;

    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3().[NSTBE, NSTB];
        if state_bits == '10x' || (!IsFeatureImplemented(FEAT_Secure) && state_bits == '00x') then
            // Reserved value
            reserved = TRUE;
            (-, state_bits) = ConstrainUnpredictableBits{3}(Unpredictable_RESERVEDNSxB);
        end;
    else
        state_bits = '0' :: MDCR_EL3().NSTB;
    end;

    return (state_bits, reserved);
end;
```

J1.4.4.15 EffectiveTRBLIMITR_EL1_nVM

```
// EffectiveTRBLIMITR_EL1_nVM()
// ============================

func EffectiveTRBLIMITR_EL1_nVM() => bit
begin
    if !SelfHostedTraceEnabled() then
        // If SelfHostedTraceEnabled() is FALSE, then this function is only called when
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // FEAT_TRBE_EXT is implemented.
        assert IsFeatureImplemented(FEAT_TRBE_EXT);
        return '1';
    end;
    if IsFeatureImplemented(FEAT_TRBEv1p1) && HaveEL(EL2) then
        let (owning_ss, owning_el) = TraceBufferOwner();
        if ((owning_ss != SS_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
              TRFCR_EL2().DnVM == '1') then
            return '0';
        end;
    end;
    return TRBLIMITR_EL1().nVM;
end;
```

J1.4.4.16 EffectiveTRFCR_EL1_EE

```
// EffectiveTRFCR_EL1_EE()
// =======================
// Return the Effective value of TRFCR_EL1.EE for the purpose of controlling the
// TRBE Profiling exception.

func EffectiveTRFCR_EL1_EE() => bits(2)
begin
    if EffectiveTRFCR_EL2_EE() == '00' then
        return '00';
    end;

    var ee : bits(2) = TRFCR_EL1().EE;
    if ee IN {'01', '10'} then            // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee[0] = ee[1];
        else
            var c : Constraint;
            (c, ee) = ConstrainUnpredictableBits{2}(Unpredictable_RESTRFEE);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                ee = '00';
            end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
    end;

    return ee;
end;
```

J1.4.4.17 EffectiveTRFCR_EL2_EE

```
// EffectiveTRFCR_EL2_EE()
// =======================
// Return the Effective value of TRFCR_EL2.EE.

func EffectiveTRFCR_EL2_EE() => bits(2)
begin
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return '00';
    end;

    if HaveEL(EL3) && MDCR_EL3().TRBEE == '00' then
        return '00';
    end;

    let check_el2 : boolean = HaveEL(EL2) && (EffectiveSCR_EL3_NS() == '1' ||
                                              IsSecureEL2Enabled());
    return if check_el2 then TRFCR_EL2().EE else '01';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
end;
```

J1.4.4.18 ElementStreamSize

```
// ElementStreamSize
// =================

config ElementStreamSize : integer{0..4096} = 4096;
```

J1.4.4.19 GetTRBSR_EL1_FSC

```
// GetTRBSR_EL1_FSC()
// ==================
// Query the TRBSR_EL1.FSC field.

func GetTRBSR_EL1_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = TRBSR_EL1()[5:0];
    return FSC;
end;
```

J1.4.4.20 GetTRBSR_EL2_FSC

```
// GetTRBSR_EL2_FSC()
// ==================
// Query the TRBSR_EL2.FSC field.

func GetTRBSR_EL2_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = TRBSR_EL2()[5:0];
    return FSC;
end;
```

J1.4.4.21 GetTRBSR_EL3_FSC

```
// GetTRBSR_EL3_FSC()
// ==================
// Query the TRBSR_EL3.FSC field.

func GetTRBSR_EL3_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = TRBSR_EL3()[5:0];
    return FSC;
end;
```

J1.4.4.22 HaveImpDefTraceOutput

```
// HaveImpDefTraceOutput()
// =======================

func HaveImpDefTraceOutput() => boolean
begin
    return ImpDefBool("Has Enabled External Trace Port");
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.4.23 ImpDefTraceOutput

```
// ImpDefTraceOutput()
// ===================

func ImpDefTraceOutput(datum : bits(8)) => boolean
begin
    // Send 'datum' to an IMPLEMENTATION DEFINED trace output port
    // return TRUE if the byte is sent
    return FALSE;
end;
```

J1.4.4.24 OtherTRBEManagementEvent

```
// OtherTRBEManagementEvent()
// ==========================
// Report an Other buffer management event, with the status code 'bsc'

func OtherTRBEManagementEvent(bsc : bits(6))
begin
    ReportTRBEManagementEvent('000000', bsc);
end;
```

J1.4.4.25 ReportTRBEEvent

```
// ReportTRBEEvent()
// =================
// Return the target ELx for an indirect write to TRBSR_ELx.
// When the indirect write is due to a buffer management event:
// 'ec_bits' is the Event Class for the management event.
// 'fsc_bits' is the Fault Status Code when this is a fault, ignored otherwise.
// Otherwise, 'ec_bits' should be Zeros().

func ReportTRBEEvent(ec_bits : bits(6), fsc_bits : bits(6)) => bits(2)
begin
    var target_el : bits(2);
    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;

    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        if ec_bits == '011111' then                     // implementation defined fault
            return ImpDefBits{2}("target_el for TRBE ImpDef Fault");
        end;

        let s1fault : boolean = (ec_bits == '100100');   // Stage 1 fault
        let s2fault : boolean = (ec_bits == '100101');   // Stage 2 fault

        var gpcfault, gpfault : boolean;
        if IsFeatureImplemented(FEAT_RME) then
            // Granule Protection Check fault, other than GPF. That is, a GPT address size fault,
            // GPT walk fault, or synchronous External abort on GPT fetch.
            gpcfault = (ec_bits == '011110');
            // Other Granule Protection Fault, reported as Stage 1 or Stage 2 fault.
            gpfault  =  ((s1fault || s2fault) && fsc_bits IN {'10001x', '1001xx', '101000'});
        else
            gpcfault = FALSE;
            gpfault  = FALSE;
        end;
        let sync_ext_abort : boolean = ((s1fault || s2fault) &&
                                        fsc_bits IN {'010000', '01001x', '0101xx', '011011'});

        var owning_ss : SecurityState;
        var owning_el : bits(2);
        (owning_ss, owning_el) = TraceBufferOwner();

        if HaveEL(EL3) && MDCR_EL3().TRBEE == '1x' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            route_to_el3 = (MDCR_EL3().TRBEE == '11' ||
                            gpcfault || (gpfault && SCR_EL3().GPF == '1') ||
                            (sync_ext_abort && EffectiveEA() == '1'));
        end;

        if EffectiveTRFCR_EL2_EE() == '1x' then
            route_to_el2 = (TRFCR_EL2().EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                            gpcfault || (gpfault && HCR_EL2().GPF == '1') ||
                            (sync_ext_abort && EffectiveHCR_TEA() == '1'));
        end;
    end;

    if route_to_el3 then
        target_el = EL3;
    elsif route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;
    end;

    return target_el;
end;
```

J1.4.4.26 ReportTRBEManagementEvent

```
// ReportTRBEManagementEvent()
// ==========================
// Report a buffer management event with the event class 'ec' and status code 'bsc'

func ReportTRBEManagementEvent(ec : bits(6), bsc : bits(6))
begin
    let target_el : bits(2) = DefaultTRBEEvent();
    if TRBSR_EL(target_el).S == '0' then
        TRBSR_EL(target_el).S    = '1';     // Stop collection
        if TRBSR_EL(target_el).IRQ == '0' then
            TRBSR_EL(target_el).IRQ  = '1';     // Assert interrupt or exception
            CTI_SignalEvent(CrossTriggerIn_TRBEMgmt);
        end;
        TRBSR_EL(target_el).EC   = ec;
        TRBSR_EL(target_el).MSS  = ZeroExtend{16}(bsc);
        TRBSR_EL(target_el).MSS2 = Zeros{24};
        CTI_SignalEvent(CrossTriggerIn_TRBEStop);
    end;
end;
```

J1.4.4.27 TRBEInternalBufferFull

```
// TRBEInternalBufferFull()
// ========================

func TRBEInternalBufferFull() => boolean
begin
    // In the simple sequential model, the internal buffer never fills
    return FALSE;
end;
```

J1.4.4.28 TRBEInterruptEnabled

```
// TRBEInterruptEnabled()
// ======================
// Return TRUE if the TRBE interrupt request (TRBIRQ) is enabled, FALSE otherwise.

func TRBEInterruptEnabled() => boolean
begin
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return EffectiveTRFCR_EL1_EE() == '00';
end;
```

J1.4.4.29 TRBE_TRBIDR_P_Read

```
// TRBE_TRBIDR_P_Read()
// ====================
// Called when TRBIDR_EL1 is read, returns the value of TRBIDR_EL1.P

func TRBE_TRBIDR_P_Read() => bit
begin
    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = TraceBufferOwner();

    // Reads as one if the Trace Buffer is owned by a higher Exception
    // Level or another Security state.
    if (UInt(owning_el) > UInt(PSTATE.EL) ||
          (PSTATE.EL != EL3 && owning_ss != CurrentSecurityState())) then
        return '1';
    else
        return '0';
    end;
end;
```

J1.4.4.30 TRBSR_EL

```
// TRBSR_EL - accessor
// ===================

accessor TRBSR_EL(el : bits(2)) <=> value : TRBSRType
begin
    getter
        var r : bits(64);
        case el of
            when EL1 =>   r = TRBSR_EL1();
            when EL2 =>   r = TRBSR_EL2();
            when EL3 =>   r = TRBSR_EL3();
            otherwise =>  unreachable;
        end;
        return r;
    end;

    setter
        let r : bits(64) = value;
        case el of
            when EL1 =>   TRBSR_EL1() = r;
            when EL2 =>   TRBSR_EL2() = r;
            when EL3 =>   TRBSR_EL3() = r;
            otherwise =>  unreachable;
        end;
        return;
    end;
end;
```

J1.4.4.31 TraceBufferEnabled

```
// TraceBufferEnabled()
// ====================

func TraceBufferEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_TRBE) then
        return FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif SelfHostedTraceEnabled() then
        if TRBLIMITR_EL1().E == '0' then
            return FALSE;
        end;
        var el : bits(2);
        (-, el) = TraceBufferOwner();
        return !ELUsingAArch32(el);
    elsif IsFeatureImplemented(FEAT_TRBE_EXT) then
        return TRBLIMITR_EL1().XE == '1';
    else
        return FALSE;
    end;
end;
```

J1.4.4.32 TraceBufferOwner

```
// TraceBufferOwner()
// ==================
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

func TraceBufferOwner() => (SecurityState, bits(2))
begin
    assert IsFeatureImplemented(FEAT_TRBE);

    var owning_ss : SecurityState;
    var state_bits :  bits(3);
    if HaveEL(EL3) then
        (state_bits, -) = EffectiveMDCR_EL3_NSTB();
    else
        state_bits = if SecureOnlyImplementation() then '001' else '011';
    end;

    case state_bits of
        when '00x' => owning_ss = SS_Secure;
        when '01x' => owning_ss = SS_NonSecure;
        when '11x' => owning_ss = SS_Realm;
    end;
    var owning_el : bits(2);
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2().E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    end;
    return (owning_ss, owning_el);
end;
```

J1.4.4.33 TraceBufferRunning

```
// TraceBufferRunning()
// ====================

func TraceBufferRunning() => boolean
begin
    if !TraceBufferEnabled() then
        return FALSE;
    end;

    var stopped : boolean = TRBSR_EL1().S == '1';
    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        if HaveEL(EL3) && MDCR_EL3().TRBEE == '1x' then
            stopped = stopped || (TRBSR_EL3().S == '1');
        end;
        if EffectiveTRFCR_EL2_EE() == '1x' then
            stopped = stopped || (TRBSR_EL2().S == '1');
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        end;
    end;
    return !stopped;
end;
```

J1.4.4.34 TraceUnitFlushOnTriggerComplete

```
// TraceUnitFlushOnTriggerComplete()
// =================================
// Called when a trace unit flush completes following a call to
// TraceUnitFlush() due to a trace trigger.

func TraceUnitFlushOnTriggerComplete()
begin
    if TRBLIMITR_EL1().TM == '00' then     // Stop on trigger
        let bsc : bits(6) = '000010';      // Trigger event
        OtherTRBEManagementEvent(bsc);
    elsif TRBLIMITR_EL1().TM != '11' then  // Not Ignore trigger
        let target_el : bits(2) = DefaultTRBEEvent();
        TRBSR_EL(target_el).IRQ = '1';     // Assert interrupt or exception
    end;
end;
```

J1.4.4.35 TryAssertTRBIRQ

```
// TryAssertTRBIRQ()
// =================
// Assert TRBIRQ pin when appropriate.

func TryAssertTRBIRQ()
begin
    if TRBEInterruptEnabled() && TRBSR_EL1().IRQ == '1' then
        SetInterruptRequestLevel(InterruptID_TRBIRQ, HIGH);
    else
        SetInterruptRequestLevel(InterruptID_TRBIRQ, LOW);
    end;
    return;
end;
```

J1.4.4.36 TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// =============================
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

func TraceInstrumentationAllowed(ss : SecurityState, el : bits(2)) => boolean
begin
    if !IsFeatureImplemented(FEAT_ITE) then return FALSE; end;
    if ELUsingAArch32(el) then return FALSE; end;

    if TraceAllowed(el) then
        var ite_bit : bit;
        case el of
            when EL3 => ite_bit = '0';
            when EL2 => ite_bit = TRCITECR_EL2().E2E;
            when EL1 => ite_bit = TRCITECR_EL1().E1E;
            when EL0 =>
                if EffectiveTGE() == '1' then
                    ite_bit = TRCITECR_EL2().E0HE;
                else
                    ite_bit = TRCITECR_EL1().E0E;
                end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if SelfHostedTraceEnabled() then
            return ite_bit == '1';
        else
            var el_bit : bit;
            var ss_bit : bit;
            case el of
                when EL0 => el_bit = TRCITEEDCR().E0;
                when EL1 => el_bit = TRCITEEDCR().E1;
                when EL2 => el_bit = TRCITEEDCR().E2;
                when EL3 => el_bit = TRCITEEDCR().E3;
            end;
            case ss of
                when SS_Realm =>     ss_bit = TRCITEEDCR().RL;
                when SS_Secure =>    ss_bit = TRCITEEDCR().S;
                when SS_NonSecure => ss_bit = TRCITEEDCR().NS;
                otherwise => ss_bit = '1';
            end;

            let ed_allowed : boolean = ss_bit == '1' && el_bit == '1';

            if TRCCONFIGR().ITO == '1' then
                return ed_allowed;
            else
                return ed_allowed && ite_bit == '1';
            end;
        end;
    else
        return FALSE;
    end;
end;
```

J1.4.4.37 TraceUnitFlush

```
// TraceUnitFlush()
// ================
// Called when a trace unit flush is requested, to output previous recorded trace.

impdef func TraceUnitFlush()
begin
    return;
end;
```

J1.4.4.38 EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =================
// Returns effective E0HTRE value

func EffectiveE0HTRE() => bit
begin
    return if ELUsingAArch32(EL2) then HTRFCR().E0HTRE else TRFCR_EL2().E0HTRE;
end;
```

J1.4.4.39 EffectiveE0TRE

```
// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

func EffectiveE0TRE() => bit
begin
    return if ELUsingAArch32(EL1) then TRFCR().E0TRE else TRFCR_EL1().E0TRE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.4.40 EffectiveE1TRE

```
// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

func EffectiveE1TRE() => bit
begin
    return if UsingAArch32() then TRFCR().E1TRE else TRFCR_EL1().E1TRE;
end;
```

J1.4.4.41 EffectiveE2TRE

```
// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

func EffectiveE2TRE() => bit
begin
    return if UsingAArch32() then HTRFCR().E2TRE else TRFCR_EL2().E2TRE;
end;
```

J1.4.4.42 SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

readonly func SelfHostedTraceEnabled() => boolean
begin
    var secure_trace_enable : bit = '0';
    if !(HaveTraceExt() && IsFeatureImplemented(FEAT_TRF)) then return FALSE; end;
    if EDSCR().TFO == '0' then return TRUE; end;
    if IsFeatureImplemented(FEAT_RME) then
        secure_trace_enable = if IsFeatureImplemented(FEAT_SEL2) then MDCR_EL3().STE else '0';
        return ((secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled()) ||
                (MDCR_EL3().RLTE == '1' && !ExternalRealmNoninvasiveDebugEnabled()));
    end;
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR().STE else MDCR_EL3().STE;
    else
        secure_trace_enable = if SecureOnlyImplementation() then '1' else '0';
    end;

    if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
        return TRUE;
    end;

    return FALSE;
end;
```

J1.4.4.43 TraceAllowed

```
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

func TraceAllowed(el : bits(2)) => boolean
begin
    if !HaveTraceExt() then
        return FALSE;
    end;
    // If in Debug state then tracing is not allowed
    if Halted() && !Restarting() then
        return FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    end;
    if SelfHostedTraceEnabled() then
        var trace_allowed : boolean;
        let ss : SecurityState = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never allowed.
        case ss of
            when SS_NonSecure =>
                trace_allowed = TRUE;
            when SS_Secure =>
                var trace_bit : bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR().STE else MDCR_EL3().STE;
                else
                    trace_bit = '1';
                end;
                trace_allowed = trace_bit == '1';
            when SS_Realm =>
                trace_allowed = MDCR_EL3().RLTE == '1';
            when SS_Root =>
                trace_allowed = FALSE;
        end;

        // Tracing is prohibited if the trace buffer owning security state is not the
        // current Security state or the owning Exception level is a lower Exception level.
        if IsFeatureImplemented(FEAT_TRBE) && TraceBufferEnabled() then
            let (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;
            end;
        end;

        var TRE_bit : bit;
        case el of
            when EL3 =>  TRE_bit = if !HaveAArch64() then TRFCR().E1TRE else '0';
            when EL2 =>  TRE_bit = EffectiveE2TRE();
            when EL1 =>  TRE_bit = EffectiveE1TRE();
            when EL0 =>
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveE0HTRE();
                else
                    TRE_bit = EffectiveE0TRE();
                end;
        end;

        return trace_allowed && TRE_bit == '1';
    else
        return ExternalNoninvasiveDebugAllowed(el);
    end;
end;
```

J1.4.4.44 TraceContextIDR2

```
// TraceContextIDR2()
// ==================

func TraceContextIDR2() => boolean
begin
    if !TraceAllowed(PSTATE.EL)|| !HaveEL(EL2) then return FALSE; end;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2().CX == '1');
end;
```

J1.4.4.45 TraceSynchronizationBarrier

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// TraceSynchronizationBarrier()
// ============================
// Barrier instruction that preserves the relative order of accesses to System
// registers due to trace operations and other accesses to the same registers.
// When FEAT_TRBE is implemented, a TraceSynchronizationBarrier also acts as a memory
// barrier operation to flush any trace data generated by the trace unit, such that
// a subsequent Data Synchronization Barrier does not complete until the trace data
// has been written to memory.

func TraceSynchronizationBarrier()
begin
    if IsFeatureImplemented(FEAT_TRBE) && !TraceAllowed(PSTATE.EL) then
        TraceUnitFlush();
    end;
    return;
end;
```

J1.4.4.46 TraceTimeStamp

```
// TraceTimeStamp()
// ================

func TraceTimeStamp() => TimeStamp
begin
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            var TS_el2 : bits(2) = TRFCR_EL2().TS;
            if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits{2}(Unpredictable_EL2TIMESTAMP);
            end;

            case TS_el2 of
                when '00' =>
                    // Falls out to check TRFCR_EL1.TS
                    pass;
                when '01' =>
                    return TimeStamp_Virtual;
                when '10' =>
                    // Otherwise ConstrainUnpredictableBits removes this case
                    assert IsFeatureImplemented(FEAT_ECV);
                    return TimeStamp_OffsetPhysical;
                when '11' =>
                    return TimeStamp_Physical;
            end;
        end;

        var TS_el1 : bits(2) = TRFCR_EL1().TS;
        if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits{2}(Unpredictable_EL1TIMESTAMP);
        end;

        case TS_el1 of
            when '01' =>
                return TimeStamp_Virtual;
            when '10' =>
                assert IsFeatureImplemented(FEAT_ECV);
                return TimeStamp_OffsetPhysical;
            when '11' =>
                return TimeStamp_Physical;
            otherwise =>
                unreachable;         // ConstrainUnpredictableBits removes this case
        end;
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return TimeStamp_CoreSight;
    end;
end;
```

J1.4.4.47 IsTraceCorePowered

```
// IsTraceCorePowered()
// ====================
// Returns TRUE if the trace unit Core power domain is powered up

impdef func IsTraceCorePowered() => boolean
begin
    return TRUE;
end;
```

J1.4.5 shared/translation

This section includes the following pseudocode functions:

- at
- EncodePARAttrs
- PAREncodeShareability
- ReportedPARAttrs
- ReportedPARShareability
- DecodeDevice
- DecodeLDFAttr
- DecodeSDFAttr
- DecodeShareability
- EffectiveShareability
- IsTaggableMemAttr
- IsWBShareable
- NormalNCMemAttr
- S1ConstrainUnpredictableRESMAIR
- S1DecodeMemAttrs
- S2CombineS1AttrHints
- S2CombineS1Device
- S2CombineS1MemAttrs
- S2CombineS1Shareability
- S2DecodeCacheability
- S2DecodeMemAttrs
- S2MemTagType
- S2ResMemAttr
- WalkMemAttrs
- AlignmentFault
- ExclusiveFault
- NoFault
- AbovePPS
- DecodeGPTBlock
- DecodeGPTContiguous
- DecodeGPTGranules
- DecodeGPTTable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- DecodePGS
- DecodePGSRange
- DecodePPS
- GPCBW_EL3BWSTRIDEValid
- GPCFault
- GPCNoFault
- GPCRegistersConsistent
- GPICheck
- GPIIndex
- GPIValid
- GPT
- GPTBlockDescriptorValid
- GPTContigDescriptorValid
- GPTEntry
- GPTGranulesDescriptorValid
- GPTL0Size
- GPTLevel0EntryValid
- GPTLevel0Index
- GPTLevel1EntryValid
- GPTLevel1Index
- GPTTable
- GPTTableDescriptorValid
- GPTWalk
- GranuleProtectionCheck
- IsGranuleProtectionCheckedAccess
- PAWithinGPCBypassWindow
- PGS
- Table
- S1TranslationRegime
- AddressDescriptor
- ContiguousSize
- CreateAddressDescriptor
- CreateFaultyAddressDescriptor
- DecodePASpace
- DescriptorType
- Domains
- FetchDescriptor
- HasUnprivileged
- NUM_PABITS
- Regime
- RegimeUsingAArch32
- S1TTWParams
- S2TTWParams
- SDFType
- SecurityStateForRegime
- StageOA
- TGx

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TGxGranuleBits
- TLBContext
- TLBRecord
- TTWState
- TranslationRegime
- TranslationSize
- UseASID
- UseVMID

J1.4.5.1 at

```
type TranslationStage of enumeration {
    TranslationStage_1,
    TranslationStage_12
};

type ATAccess of enumeration {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_Any,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

J1.4.5.2 EncodePARAttrs

```
// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

func EncodePARAttrs(memattrs : MemoryAttributes) => bits(8)
begin
    var result : bits(8);

    if IsFeatureImplemented(FEAT_MTE) && memattrs.tags == MemTag_AllocationTagged then
        if IsFeatureImplemented(FEAT_MTE_PERM) && memattrs.notagaccess then
            result[7:0] = '11100000';
        else
            result[7:0] = '11110000';
        end;
        return result;
    end;

    if memattrs.memtype == MemType_Device then
        result[7:4] = '0000';
        case memattrs.device of
            when DeviceType_nGnRnE => result[3:0] = '0000';
            when DeviceType_nGnRE  => result[3:0] = '0100';
            when DeviceType_nGRE   => result[3:0] = '1000';
            when DeviceType_GRE    => result[3:0] = '1100';
            otherwise              => unreachable;
        end;
        result[0] = NOT memattrs.xs;
    else
        if memattrs.xs == '0' then
            if (memattrs.outer.attrs == MemAttr_WT && memattrs.inner.attrs == MemAttr_WT &&
                  !memattrs.outer.transient && memattrs.outer.hints == MemHint_RA) then
                return '10100000';
            elsif memattrs.outer.attrs == MemAttr_NC && memattrs.inner.attrs == MemAttr_NC then
                return '01000000';
            end;
        end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if memattrs.outer.attrs == MemAttr_WT then
            result[7:6] = if memattrs.outer.transient then '00' else '10';
            result[5:4] = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr_WB then
            result[7:6] = if memattrs.outer.transient then '01' else '11';
            result[5:4] = memattrs.outer.hints;
        else // MemAttr_NC
            result[7:4] = '0100';
        end;

        if memattrs.inner.attrs == MemAttr_WT then
            result[3:2] = if memattrs.inner.transient then '00' else '10';
            result[1:0] = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr_WB then
            result[3:2] = if memattrs.inner.transient then '01' else '11';
            result[1:0] = memattrs.inner.hints;
        else // MemAttr_NC
            result[3:0] = '0100';
        end;
    end;

    return result;
end;
```

J1.4.5.3 PAREncodeShareability

```
// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

func PAREncodeShareability(memattrs : MemoryAttributes) => bits(2)
begin
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';
    end;

    case memattrs.shareability of
        when Shareability_NSH => return '00';
        when Shareability_ISH => return '11';
        when Shareability_OSH => return '10';
    end;
end;
```

J1.4.5.4 ReportedPARAttrs

```
// ReportedPARAttrs()
// ==================
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

impdef func ReportedPARAttrs(parattrs : bits(8)) => bits(8)
begin
    return parattrs;
end;
```

J1.4.5.5 ReportedPARShareability

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ReportedPARShareability()
// =========================
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

impdef func ReportedPARShareability(sh : bits(2)) => bits(2)
begin
    return sh;
end;
```

J1.4.5.6 DecodeDevice

```
// DecodeDevice()
// ==============
// Decode output Device type

func DecodeDevice(device : bits(2)) => DeviceType
begin
    case device of
        when '00' => return DeviceType_nGnRnE;
        when '01' => return DeviceType_nGnRE;
        when '10' => return DeviceType_nGRE;
        when '11' => return DeviceType_GRE;
    end;
end;
```

J1.4.5.7 DecodeLDFAttr

```
// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

func DecodeLDFAttr(attr : bits(4)) => MemAttrHints
begin
    var ldfattr : MemAttrHints;

    if    attr == 'x0xx' then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldfattr.attrs = MemAttr_WB; // Write-back
    else                      unreachable;
    end;

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr[1:0] of
            when '00' => ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' => ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' => ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' => ldfattr.hints = MemHint_RWA; // Read/Write allocate
        end;
    end;

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr[3] == '0';
    end;

    return ldfattr;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.8 DecodeSDFAttr

```
// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

func DecodeSDFAttr(rgn : bits(2)) => MemAttrHints
begin
    var sdfattr : MemAttrHints;

    case rgn of
        when '00' =>                    // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01' =>                    // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10' =>                    // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11' =>                    // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;
    end;

    sdfattr.transient = FALSE;

    return sdfattr;
end;
```

J1.4.5.9 DecodeShareability

```
// DecodeShareability()
// ====================
// Decode shareability of target memory region

func DecodeShareability(sh : bits(2)) => Shareability
begin
    case sh of
        when '10' => return Shareability_OSH;
        when '11' => return Shareability_ISH;
        when '00' => return Shareability_NSH;
        otherwise =>
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH => return Shareability_OSH;
                when Constraint_ISH => return Shareability_ISH;
                when Constraint_NSH => return Shareability_NSH;
            end;
    end;
end;
```

J1.4.5.10 EffectiveShareability

```
// EffectiveShareability()
// =======================
// Force Outer Shareability on Device and Normal iNCoNC memory

func EffectiveShareability(memattrs : MemoryAttributes) => Shareability
begin
    if (memattrs.memtype == MemType_Device ||
          (memattrs.inner.attrs == MemAttr_NC &&
           memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.11 IsTaggableMemAttr

```
// IsTaggableMemAttr()
// ===================
// Determine whether the current memory attributes support MTE

func IsTaggableMemAttr(attrs : MemoryAttributes) => boolean
begin
    return ((attrs.memtype == MemType_Normal)  &&
            (attrs.inner.attrs == MemAttr_WB)  &&
            (attrs.inner.hints == MemHint_RWA) &&
            (!attrs.inner.transient)           &&
            (attrs.outer.attrs == MemAttr_WB)  &&
            (attrs.outer.hints == MemHint_RWA) &&
            (!attrs.outer.transient));
end;
```

J1.4.5.12 IsWBShareable

```
// IsWBShareable()
// ===============
// Determines whether the given memory attributes are iWBoWB Shareable

func IsWBShareable(memattrs : MemoryAttributes) => boolean
begin

    return (memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.outer.attrs == MemAttr_WB &&
            memattrs.shareability IN {Shareability_ISH, Shareability_OSH});
end;
```

J1.4.5.13 NormalNCMemAttr

```
// NormalNCMemAttr()
// =================
// Normal Non-cacheable memory attributes

func NormalNCMemAttr() => MemoryAttributes
begin
    var non_cacheable : MemAttrHints;
    non_cacheable.attrs = MemAttr_NC;

    var nc_memattrs : MemoryAttributes;
    nc_memattrs.memtype      = MemType_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;
    nc_memattrs.tags         = MemTag_Untagged;
    nc_memattrs.notagaccess  = FALSE;

    return nc_memattrs;
end;
```

J1.4.5.14 S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =================================
// Determine whether a reserved value occupies MAIR_ELx.AttrN

func S1ConstrainUnpredictableRESMAIR(attr : bits(8), s1aarch64 : boolean) => boolean
begin
    case attr of
        when '0000xx01' => return !(s1aarch64 && IsFeatureImplemented(FEAT_XS));
        when '0000xxxx' => return attr[1:0] != '00';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '01000000' => return !(s1aarch64 && IsFeatureImplemented(FEAT_XS));
        when '10100000' => return !(s1aarch64 && IsFeatureImplemented(FEAT_XS));
        when '11110000' => return !(s1aarch64 && IsFeatureImplemented(FEAT_MTE2));
        when 'xxxx0000' => return TRUE;
        otherwise       => return FALSE;
    end;
end;
```

J1.4.5.15 S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

func S1DecodeMemAttrs(attr_in : bits(8), sh : bits(2), s1aarch64 : boolean) => MemoryAttributes
begin
    var attr : bits(8) = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits{8}(Unpredictable_RESMAIR);
    end;

    var memattrs : MemoryAttributes;
    case attr of
        when '0000xxxx' => // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device  = DecodeDevice(attr[3:2]);
            memattrs.xs      = if s1aarch64 then NOT attr[0] else '1';
        when '01000000' =>
            assert s1aarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs = MemAttr_NC;
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.xs          = '0';
        when '10100000' =>
            assert s1aarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs     = MemAttr_WT;
            memattrs.outer.hints     = MemHint_RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WT;
            memattrs.inner.hints     = MemHint_RA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        when '11110000' => // Tagged memory
            assert s1aarch64 && IsFeatureImplemented(FEAT_MTE2);
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs     = MemAttr_WB;
            memattrs.outer.hints     = MemHint_RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WB;
            memattrs.inner.hints     = MemHint_RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        otherwise =>
            memattrs.memtype = MemType_Normal;
            memattrs.outer   = DecodeLDFAttr(attr[7:4]);
            memattrs.inner   = DecodeLDFAttr(attr[3:0]);

            if (memattrs.inner.attrs == MemAttr_WB &&
                    memattrs.outer.attrs == MemAttr_WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';
            end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    memattrs.shareability = DecodeShareability(sh);
    memattrs.tags = MemTag_Untagged;
    memattrs.notagaccess = FALSE;

    return memattrs;
end;

// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

func S1DecodeMemAttrs(attr_in : bits(8), sh : bits(2), s1aarch64 : boolean,
                      walkparams : S1TTWParams, acctype : AccessType) => MemoryAttributes
begin
    var memattrs : MemoryAttributes = S1DecodeMemAttrs(attr_in, sh, s1aarch64);
    if s1aarch64 && attr_in == '11110000' then
        memattrs.tags = MemTag_AllocationTagged;
    elsif IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && s1aarch64 && walkparams.mtx == '1' then
        memattrs.tags = MemTag_CanonicallyTagged;
    end;

    return memattrs;
end;
```

J1.4.5.16 S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// ======================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

func S2CombineS1AttrHints(s1_attrhints : MemAttrHints, s2_attrhints : MemAttrHints) => MemAttrHints
begin
    var attrhints : MemAttrHints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;
    end;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints     = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;
    end;

    return attrhints;
end;
```

J1.4.5.17 S2CombineS1Device

```
// S2CombineS1Device()
// ===================
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

func S2CombineS1Device(s1_device : DeviceType, s2_device : DeviceType) => DeviceType
begin
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
    end;
end;
```

J1.4.5.18 S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====================
// Combine stage 2 with stage 1 memory attributes

func S2CombineS1MemAttrs(s1_memattrs : MemoryAttributes, s2_memattrs : MemoryAttributes,
                         s2aarch64 : boolean) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device  = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType_Device then    // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType_Device then    // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else                                                // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner   = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer   = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);
    end;

    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);

    if !IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = FALSE;
    else
        memattrs.notagaccess = (s2_memattrs.notagaccess &&
                                s1_memattrs.tags == MemTag_AllocationTagged);
    end;
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);

    if (memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.outer.attrs == MemAttr_WB) then
        memattrs.xs = '0';
    elsif s2aarch64 then
        memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
    else
        memattrs.xs = s1_memattrs.xs;
    end;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
end;
```

J1.4.5.19 S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

func S2CombineS1Shareability(s1_shareability : Shareability,
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                s2_shareability : Shareability) => Shareability
begin

    if (s1_shareability == Shareability_OSH ||
            s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
    end;
end;
```

J1.4.5.20 S2DecodeCacheability

```
// S2DecodeCacheability()
// ======================
// Determine the stage 2 cacheability for Normal memory

func S2DecodeCacheability(attr : bits(2)) => MemAttrHints
begin
    var s2attr : MemAttrHints;

    case attr of
        when '01' => s2attr.attrs = MemAttr_NC;  // Non-cacheable
        when '10' => s2attr.attrs = MemAttr_WT;  // Write-through
        when '11' => s2attr.attrs = MemAttr_WB;  // Write-back
        otherwise => unreachable;
    end;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints     = ARBITRARY : bits(2);
    s2attr.transient = ARBITRARY : boolean;

    return s2attr;
end;
```

J1.4.5.21 S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// ==================
// Decode stage 2 memory attributes when FWB is 0

func S2DecodeMemAttrs(attr_in : bits(4), sh : bits(2), s2aarch64 : boolean) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;

    var attr : bits(4);
    if S2ResMemAttr(s2aarch64, attr_in) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits{4}(Unpredictable_S2RESMEMATTR);
    else
        attr = attr_in;
    end;

    case attr of
        when '00xx' => // Device memory
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DecodeDevice(attr[1:0]);
        when '0100' => // Normal, Inner+Outer WB cacheable NoTagAccess memory
            assert s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM);
            memattrs.memtype      = MemType_Normal;
            memattrs.outer        = S2DecodeCacheability('11'); // Write-back
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memattrs.inner        = S2DecodeCacheability('11'); // Write-back
        otherwise =>   // Normal memory
            memattrs.memtype      = MemType_Normal;
            memattrs.outer        = S2DecodeCacheability(attr[3:2]);
            memattrs.inner        = S2DecodeCacheability(attr[1:0]);
    end;

    memattrs.shareability = DecodeShareability(sh);

    if s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = attr == '0100';
    else
        memattrs.notagaccess = FALSE;
    end;

    return memattrs;
end;
```

J1.4.5.22 S2MemTagType

```
// S2MemTagType()
// ==============
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

func S2MemTagType(s2_memattrs : MemoryAttributes, s1_tagtype : MemTagType) => MemTagType
begin

    if !IsFeatureImplemented(FEAT_MTE2) then
        return MemTag_Untagged;
    end;

    if s1_tagtype == MemTag_AllocationTagged && IsTaggableMemAttr(s2_memattrs) then
        return MemTag_AllocationTagged;
    end;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag_AllocationTagged then
        return s1_tagtype;
    end;

    return MemTag_Untagged;
end;
```

J1.4.5.23 S2ResMemAttr

```
// S2ResMemAttr()
// ==============
// Determine whether a reserved value occupies stage 2 MemAttr field when FWB is 0

func S2ResMemAttr(s2aarch64 : boolean, attr : bits(4)) => boolean
begin
    case attr of
        when '0100' => return !(s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM));
        when '1x00' => return TRUE;
        otherwise   => return FALSE;
    end;
end;
```

J1.4.5.24 WalkMemAttrs

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// WalkMemAttrs()
// ==============
// Retrieve memory attributes of translation table walk

func WalkMemAttrs(sh : bits(2), irgn : bits(2), orgn : bits(2)) => MemoryAttributes
begin
    var walkmemattrs : MemoryAttributes;

    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
    walkmemattrs.outer        = DecodeSDFAttr(orgn);
    walkmemattrs.tags         = MemTag_Untagged;
    if (walkmemattrs.inner.attrs == MemAttr_WB &&
            walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';
    end;
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;
end;
```

J1.4.5.25 AlignmentFault

```
// AlignmentFault()
// ================
// Return a fault record indicating an Alignment fault not due to memory type has occurred
// for a specific access

func AlignmentFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    fault.statuscode        = Fault_Alignment;

    return fault;
end;
```

J1.4.5.26 ExclusiveFault

```
// ExclusiveFault()
// ================
// Return a fault record indicating a fault for an unsupported Exclusive access

func ExclusiveFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    fault.statuscode        = Fault_Exclusive;

    return fault;
end;
```

J1.4.5.27 NoFault

```
// NoFault()
// =========
// Return a clear fault record indicating no faults have occurred

func NoFault() => FaultRecord
begin
    var fault : FaultRecord;

    fault.vaddress = ARBITRARY : bits(64);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    fault.statuscode  = Fault_None;
    fault.accessdesc  = ARBITRARY : AccessDescriptor;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.toplevel    = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess    = FALSE;
    fault.gpcfs2walk   = FALSE;
    fault.gpcf         = GPCNoFault();
    fault.hdbssf       = FALSE;

    return fault;
end;

// NoFault()
// =========
// Return a clear fault record indicating no faults have occurred for a specific access

func NoFault(accdesc : AccessDescriptor) => FaultRecord
begin
    var fault : FaultRecord;

    fault.statuscode  = Fault_None;
    fault.accessdesc  = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.toplevel    = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess   = FALSE;
    fault.write       = !accdesc.read && accdesc.write;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();
    fault.hdbssf      = FALSE;

    return fault;
end;

// NoFault()
// =========

func NoFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault();
    fault.accessdesc  = accdesc;
    fault.write       = !accdesc.read && accdesc.write;
    fault.vaddress    = vaddress;

    return fault;
end;
```

J1.4.5.28 AbovePPS

```
// AbovePPS()
// ==========
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

func AbovePPS(address : bits(NUM_PABITS)) => boolean
begin
    let pps : integer{} = DecodePPS();
    if pps >= NUM_PABITS then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return FALSE;
    end;

    return !IsZero(address[NUM_PABITS-1:pps]);
end;
```

J1.4.5.29 DecodeGPTBlock

```
// DecodeGPTBlock()
// ================
// Decode a GPT Block descriptor.

func DecodeGPTBlock(pgs : PGSe, gpt_entry : bits(64)) => GPTEntry
begin
    assert gpt_entry[3:0] == GPT_Block;
    var result : GPTEntry;
    result.gpi   = gpt_entry[7:4];
    result.level = 0;
    result.istable = FALSE;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS_4KB =>  result.size = GPTRange_4KB;
        when PGS_16KB => result.size = GPTRange_16KB;
        when PGS_64KB => result.size = GPTRange_64KB;
        otherwise => unreachable;
    end;
    result.contig_size = GPTL0Size();

    return result;
end;
```

J1.4.5.30 DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====================
// Decode a GPT Contiguous descriptor.

func DecodeGPTContiguous(pgs : PGSe, gpt_entry : bits(64)) => GPTEntry
begin
    assert gpt_entry[3:0] == GPT_Contig;
    var result : GPTEntry;
    result.gpi = gpt_entry[7:4];

    case pgs of
        when PGS_4KB =>  result.size = GPTRange_4KB;
        when PGS_16KB => result.size = GPTRange_16KB;
        when PGS_64KB => result.size = GPTRange_64KB;
        otherwise     => unreachable;
    end;

    case gpt_entry[9:8] of
        when '01' => result.contig_size = GPTRange_2MB;
        when '10' => result.contig_size = GPTRange_32MB;
        when '11' => result.contig_size = GPTRange_512MB;
        otherwise => unreachable;
    end;

    result.level = 1;
    result.istable = FALSE;

    return result;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.31 DecodeGPTGranules

```
// DecodeGPTGranules()
// ===================
// Decode a GPT Granules descriptor.

func DecodeGPTGranules(pgs : PGSe, index : integer, gpt_entry : bits(64)) => GPTEntry
begin
    var result : GPTEntry;
    result.gpi = gpt_entry[index*4 +:4];

    case pgs of
        when PGS_4KB  => result.size = GPTRange_4KB;
        when PGS_16KB => result.size = GPTRange_16KB;
        when PGS_64KB => result.size = GPTRange_64KB;
        otherwise     => unreachable;
    end;

    result.contig_size = result.size; // No contiguity
    result.level = 1;
    result.istable = FALSE;

    return result;
end;
```

J1.4.5.32 DecodeGPTTable

```
// DecodeGPTTable()
// ================
// Decode a GPT Table descriptor.

func DecodeGPTTable(pgs : PGSe, gpt_entry : bits(64)) => GPTTable
begin
    assert gpt_entry[3:0] == GPT_Table;
    var result : GPTTable;

    // Descriptor bits [L0GPTSZ-PGS-2:12] are RES0
    var s : integer{30, 34, 36, 39};
    case GPTL0Size() of
        when GPTRange_1GB   => s = 30;
        when GPTRange_16GB  => s = 34;
        when GPTRange_64GB  => s = 36;
        when GPTRange_512GB => s = 39;
        otherwise           => unreachable;
    end;

    var p : integer{12, 14, 16};
    case pgs of
        when PGS_4KB  => p = 12;
        when PGS_16KB => p = 14;
        when PGS_64KB => p = 16;
        otherwise     => unreachable;
    end;

    let align_bits :  integer{} = (s - p) - 1;
    result.address              = AlignDownP2(gpt_entry[NUM_PABITS-1:0],align_bits);

    return result;
end;
```

J1.4.5.33 DecodePGS

```
// DecodePGS()
// ===========

func DecodePGS(pgs : bits(2)) => PGSe
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
begin
    case pgs of
        when '00' => return PGS_4KB;
        when '10' => return PGS_16KB;
        when '01' => return PGS_64KB;
        otherwise => unreachable;
    end;
end;
```

J1.4.5.34 DecodePGSRange

```
// DecodePGSRange()
// ================

func DecodePGSRange(pgs : PGSe) => AddressSize
begin
    case pgs of
        when PGS_4KB =>  return GPTRange_4KB;
        when PGS_16KB => return GPTRange_16KB;
        when PGS_64KB => return GPTRange_64KB;
        otherwise => unreachable;
    end;
end;
```

J1.4.5.35 DecodePPS

```
// DecodePPS()
// ===========
// Size of region protected by the GPT, in bits.

func DecodePPS() => AddressSize
begin
    case GPCCR_EL3().[PPS3, PPS] of
        when '0000' => return 32;
        when '0001' => return 36;
        when '0010' => return 40;
        when '0011' => return 42;
        when '0100' => return 44;
        when '0101' => return 48;
        when '0110' => return 52;
        when '0111' => assert IsFeatureImplemented(FEAT_RME_GPC3); return 56;
        when '1000' => assert IsFeatureImplemented(FEAT_RME_GPC3); return 46;
        when '1001' => assert IsFeatureImplemented(FEAT_RME_GPC3); return 47;
        otherwise => unreachable;
    end;
end;
```

J1.4.5.36 GPCBW_EL3BWSTRIDEValid

```
// GPCBW_EL3BWSTRIDEValid()
// ========================
// Returns whether the current GPCBW_EL3.BWSTRIDE value is valid

func GPCBW_EL3BWSTRIDEValid() => boolean
begin
    assert IsFeatureImplemented(FEAT_RME_GPC3);
    return GPCBW_EL3().BWSTRIDE IN {
        '00000',
        '00010',
        '00100',
        '00110',
        '00111',
        '01000',
        '01001',
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        '01010',
        '10000'
    };
end;
```

J1.4.5.37 GPCFault

```
// GPCFault()
// ==========
// Constructs and returns a GPCF

func GPCFault(gpf : GPCF, level : integer) => GPCFRecord
begin
    var fault : GPCFRecord;
    fault.gpf   = gpf;
    fault.level = level;
    return fault;
end;
```

J1.4.5.38 GPCNoFault

```
// GPCNoFault()
// ============
// Returns the default properties of a GPCF that does not represent a fault

func GPCNoFault() => GPCFRecord
begin
    var result : GPCFRecord;
    result.gpf = GPCF_None;
    return result;
end;
```

J1.4.5.39 GPCRegistersConsistent

```
// GPCRegistersConsistent()
// ========================
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

func GPCRegistersConsistent() => boolean
begin
    // Check for Reserved register values
    if IsFeatureImplemented(FEAT_RME_GPC3) then
        if  ! GPCCR_EL3().[PPS3, PPS] IN {'0xxx', '100x'} then
            return FALSE;
        end;

        if GPCCR_EL3().GPCBW == '1' then
            if ! GPCBW_EL3().BWSIZE IN {'00x', '1x0', '010'} then
                return FALSE;
            end;

            if !GPCBW_EL3BWSTRIDEValid() then
                return FALSE;
            end;

            let bwstride : integer = 1 << (40 + UInt(GPCBW_EL3().BWSTRIDE));
            let bwaddr : integer = UInt(GPCBW_EL3().BWADDR) << 30;
            if bwstride <= bwaddr then
                return FALSE;
            end;

            if !IsAlignedSize{26}(GPCBW_EL3().BWADDR, 1 << UInt(GPCBW_EL3().BWSIZE)) then
                return FALSE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            end;
        end;
    else
        if GPCCR_EL3().PPS == '111' then
            return FALSE;
        end;
    end;

    if DecodePPS() > AArch64_PAMax() then
        return FALSE;
    end;
    if GPCCR_EL3().PGS == '11' then
        return FALSE;
    end;
    if GPCCR_EL3().SH == '01' then
        return FALSE;
    end;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3().[ORGN, IRGN] == '0000' && GPCCR_EL3().SH != '10' then
        return FALSE;
    end;

    return TRUE;
end;
```

J1.4.5.40 GPICheck

```
// GPICheck()
// ==========
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access
// ss: Security state of the access

func GPICheck(paspace : PASpace, gpi : bits(4), ss : SecurityState) => boolean
begin
    case gpi of
        when GPT_NoAccess =>
            return FALSE;
        when GPT_Secure =>
            assert IsFeatureImplemented(FEAT_SEL2);
            return paspace == PAS_Secure;
        when GPT_NonSecure =>
            return paspace == PAS_NonSecure;
        when GPT_Root =>
            return paspace == PAS_Root;
        when GPT_Realm =>
            return paspace == PAS_Realm;
        when GPT_NonSecureOnly =>
            assert IsFeatureImplemented(FEAT_RME_GPC2);
            return paspace == PAS_NonSecure && (ss IN {SS_Root, SS_NonSecure});
        when GPT_SystemAgent =>
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return paspace == PAS_SystemAgent;
        when GPT_NonSecureProtected =>
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return paspace == PAS_NonSecureProtected;
        when GPT_NA6 =>
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return FALSE;
        when GPT_NA7 =>
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return FALSE;
        when GPT_Any =>
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return TRUE;
        otherwise =>
            unreachable;
    end;
end;
```

J1.4.5.41 GPIIndex

```
// GPIIndex()
// ==========

func GPIIndex(pa : bits(NUM_PABITS)) => integer
begin
    case DecodePGS(GPCCR_EL3().PGS) of
        when PGS_4KB =>  return UInt(pa[15:12]);
        when PGS_16KB => return UInt(pa[17:14]);
        when PGS_64KB => return UInt(pa[19:16]);
        otherwise => unreachable;
    end;
end;
```

J1.4.5.42 GPIValid

```
// GPIValid()
// ==========
// Returns whether a given value is a valid encoding for a GPI value

func GPIValid(gpi : bits(4)) => boolean
begin
    case gpi of
        when GPT_NoAccess =>
            return TRUE;
        when GPT_NonSecureProtected =>
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().NSP == '1';
        when GPT_SystemAgent =>
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().SA  == '1';
        when GPT_NA6 =>
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().NA6 == '1';
        when GPT_NA7 =>
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().NA7 == '1';
        when GPT_Secure =>
            return IsFeatureImplemented(FEAT_SEL2);
        when GPT_NonSecure =>
            return TRUE;
        when GPT_Realm =>
            return TRUE;
        when GPT_Root =>
            return TRUE;
        when GPT_NonSecureOnly =>
            return IsFeatureImplemented(FEAT_RME_GPC2) && GPCCR_EL3().NSO == '1';
        when GPT_Any =>
            return TRUE;
        otherwise =>
            return FALSE;
    end;
end;
```

J1.4.5.43 GPT

```
// GPT dimensions
// ==============

constant GPTRange_4KB : AddressSize   = 12;
constant GPTRange_16KB : AddressSize  = 14;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant GPTRange_64KB : AddressSize  = 16;
constant GPTRange_2MB : AddressSize   = 21;
constant GPTRange_32MB : AddressSize  = 25;
constant GPTRange_512MB : AddressSize = 29;
constant GPTRange_1GB : AddressSize   = 30;
constant GPTRange_16GB : AddressSize  = 34;
constant GPTRange_64GB : AddressSize  = 36;
constant GPTRange_512GB : AddressSize = 39;
```

J1.4.5.44 GPTBlockDescriptorValid

```
// GPTBlockDescriptorValid()
// =========================
// Returns TRUE if the given GPT Block descriptor is valid, and FALSE otherwise.

func GPTBlockDescriptorValid(level_0_entry : bits(64)) => boolean
begin
    assert level_0_entry[3:0] == GPT_Block;
    return IsZero(level_0_entry[63:8]) && GPIValid(level_0_entry[7:4]);
end;
```

J1.4.5.45 GPTContigDescriptorValid

```
// GPTContigDescriptorValid()
// ==========================
// Returns TRUE if the given GPT Contiguous descriptor is valid, and FALSE otherwise.

func GPTContigDescriptorValid(level_1_entry : bits(64)) => boolean
begin
    assert level_1_entry[3:0] == GPT_Contig;
    return (IsZero(level_1_entry[63:10]) &&
            !IsZero(level_1_entry[9:8])  &&
            GPIValid(level_1_entry[7:4]));
end;
```

J1.4.5.46 GPTEntry

```
// GPTEntry
// ========

type GPTEntry of record {
    gpi         : bits(4),         // GPI value for this region
    size        : AddressSize,     // Region size
    contig_size : AddressSize,     // Contiguous region size
    istable     : boolean,         // Flag a table entry
    level       : integer,         // Level of GPT lookup
    pa          : bits(NUM_PABITS) // PA uniquely identifying the GPT entry
};
```

J1.4.5.47 GPTGranulesDescriptorValid

```
// GPTGranulesDescriptorValid()
// ============================
// Returns TRUE if the given GPT Granules descriptor is valid, and FALSE otherwise.

func GPTGranulesDescriptorValid(level_1_entry : bits(64)) => boolean
begin
    for i = 0 to 15 do
        if !GPIValid(level_1_entry[i*4 +:4]) then
            return FALSE;
        end;
    end;
    return TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.48 GPTL0Size

```
// GPTL0Size()
// ===========
// Returns number of bits covered by a level 0 GPT entry

func GPTL0Size() => AddressSize
begin
    case GPCCR_EL3().L0GPTSZ of
        when '0000' => return GPTRange_1GB;
        when '0100' => return GPTRange_16GB;
        when '0110' => return GPTRange_64GB;
        when '1001' => return GPTRange_512GB;
        otherwise   => unreachable;
    end;
    return 30;
end;
```

J1.4.5.49 GPTLevel0EntryValid

```
// GPTLevel0EntryValid()
// =====================
// Returns TRUE if the given level 0 gpt descriptor is valid, and FALSE otherwise.

func GPTLevel0EntryValid(gpt_entry : bits(64)) => boolean
begin
    case gpt_entry[3:0] of
        when GPT_Block => return GPTBlockDescriptorValid(gpt_entry);
        when GPT_Table => return GPTTableDescriptorValid(gpt_entry);
        otherwise =>      return FALSE;
    end;
end;
```

J1.4.5.50 GPTLevel0Index

```
// GPTLevel0Index()
// ================
// Compute the level 0 index based on input PA.

func GPTLevel0Index(pa : bits(NUM_PABITS)) => integer
begin
    // Input address and index bounds
    let pps : integer{} = DecodePPS();
    let l0sz : integer{} = GPTL0Size();
    if pps <= l0sz then
        return 0;
    end;

    return UInt(pa[pps-1:l0sz]);
end;
```

J1.4.5.51 GPTLevel1EntryValid

```
// GPTLevel1EntryValid()
// =====================
// Returns TRUE if the given level 1 gpt descriptor is valid, and FALSE otherwise.

func GPTLevel1EntryValid(gpt_entry : bits(64)) => boolean
begin
    case gpt_entry[3:0] of
        when GPT_Contig => return GPTContigDescriptorValid(gpt_entry);
        otherwise =>       return GPTGranulesDescriptorValid(gpt_entry);
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.52 GPTLevel1Index

```
// GPTLevel1Index()
// ================
// Compute the level 1 index based on input PA.

func GPTLevel1Index(pa : bits(NUM_PABITS)) => integer
begin
    // Input address and index bounds
    let l0sz : integer{} = GPTL0Size();
    case DecodePGS(GPCCR_EL3().PGS) of
        when PGS_4KB =>  return UInt(pa[l0sz-1:16]);
        when PGS_16KB => return UInt(pa[l0sz-1:18]);
        when PGS_64KB => return UInt(pa[l0sz-1:20]);
        otherwise => unreachable;
    end;
end;
```

J1.4.5.53 GPTTable

```
// GPTTable
// ========

type GPTTable of record {
    address : bits(NUM_PABITS)   // Base address of next table
};
```

J1.4.5.54 GPTTableDescriptorValid

```
// GPTTableDescriptorValid()
// =========================
// Returns TRUE if the given GPT Table descriptor is valid, and FALSE otherwise.

func GPTTableDescriptorValid(level_0_entry : bits(64)) => boolean
begin
    assert level_0_entry[3:0] == GPT_Table;
    let l0sz : integer{} = GPTL0Size();
    let pgs : PGSe       = DecodePGS(GPCCR_EL3().PGS);
    let p : integer{}    = DecodePGSRange(pgs);
    let top : integer{}  = if DecodePPS() == 56 then 56 else 52;
    return IsZero(level_0_entry[63:top,11:4]) && IsZero(level_0_entry[(l0sz-p)-2:12]);
end;
```

J1.4.5.55 GPTWalk

```
// GPTWalk()
// =========
// Get the GPT entry for a given physical address, pa

func GPTWalk(pa : bits(NUM_PABITS), accdesc : AccessDescriptor) => (GPCFRecord, GPTEntry)
begin
    // GPT base address
    var base : bits(NUM_PABITS);
    let pgs : PGSe = DecodePGS(GPCCR_EL3().PGS);

    // The level 0 GPT base address is aligned to the greater of:
    // * the size of the level 0 GPT, determined by GPCCR_EL3().[PPS, L0GPTSZ].
    // * 4KB
    base = ZeroExtend{NUM_PABITS}(GPTBR_EL3().BADDR::Zeros{12});
    let pps : AddressSize = DecodePPS();
    let l0sz : integer{} = GPTL0Size();
    let alignment : integer{} = Max((pps - l0sz) + 3, 12) as AddressSize;
    base[alignment-1:0] = Zeros{alignment};

    let gptaccdesc : AccessDescriptor = CreateAccDescGPTW(accdesc);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Access attributes and address for GPT fetches
var gptaddrdesc : AddressDescriptor;
gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3().SH, GPCCR_EL3().IRGN, GPCCR_EL3().ORGN);
gptaddrdesc.fault    = NoFault(gptaccdesc);

gptaddrdesc.paddress.paspace = PAS_Root;
gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

// Fetch L0GPT entry
var level_0_entry : bits(64);
var memstatus : PhysMemRetStatus;
(memstatus, level_0_entry) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
if IsFault(memstatus) then
    return (GPCFault(GPCF_EABT, 0), ARBITRARY : GPTEntry);
end;

if !GPTLevel0EntryValid(level_0_entry) then
    return (GPCFault(GPCF_Walk, 0), ARBITRARY : GPTEntry);
end;

var result : GPTEntry;
var table : GPTTable;
case level_0_entry[3:0] of
    when GPT_Block =>
        // Decode the GPI value and return that
        result = DecodeGPTBlock(pgs, level_0_entry);
        result.pa = pa;
        return (GPCNoFault(), result);
    when GPT_Table =>
        // Decode the table entry and continue walking
        table = DecodeGPTTable(pgs, level_0_entry);
        // The address must be within the range covered by the GPT
        if AbovePPS(table.address) then
            return (GPCFault(GPCF_AddressSize, 0), ARBITRARY : GPTEntry);
        end;
    otherwise =>
        // An invalid encoding would be caught by GPTLevel0EntryValid()
        unreachable;
end;

// Must be a GPT Table entry
assert level_0_entry[3:0] == GPT_Table;

// Address of level 1 GPT entry
let offset : integer = GPTLevel1Index(pa) * 8;

var level_1_entry : bits(64);

if IsFeatureImplemented(FEAT_RME_GDI) then
    // When FEAT_RME_GDI is implemented, the descriptor validation checks are performed
    // on a pair of descriptors within a naturally aligned 16-byte region of memory.
    gptaddrdesc.paddress.address = AlignDownSize{56}(table.address + offset, 16);
    var level_1_entry_lower : bits(64);
    (memstatus, level_1_entry_lower) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), ARBITRARY : GPTEntry);
    end;

    gptaddrdesc.paddress.address = gptaddrdesc.paddress.address + 8;
    var level_1_entry_upper : bits(64);
    (memstatus, level_1_entry_upper) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), ARBITRARY : GPTEntry);
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // An individual GPT descriptor is valid when both descriptors within the pair are valid.
        if (!GPTLevel1EntryValid(level_1_entry_upper) ||
              !GPTLevel1EntryValid(level_1_entry_lower)) then
            return (GPCFault(GPCF_Walk, 1), ARBITRARY : GPTEntry);
        end;

        if offset[3] == '1' then
            level_1_entry = level_1_entry_upper;
        else
            level_1_entry = level_1_entry_lower;
        end;
    else
        gptaddrdesc.paddress.address = table.address + offset;
        // Fetch L1GPT entry
        (memstatus, level_1_entry) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
        if IsFault(memstatus) then
            return (GPCFault(GPCF_EABT, 1), ARBITRARY : GPTEntry);
        end;

        if !GPTLevel1EntryValid(level_1_entry) then
            return (GPCFault(GPCF_Walk, 1), ARBITRARY : GPTEntry);
        end;
    end;

    case level_1_entry[3:0] of
        when GPT_Contig =>
            result = DecodeGPTContiguous(pgs, level_1_entry);
        otherwise =>
            let gpi_index : integer = GPIIndex(pa);
            result = DecodeGPTGranules(pgs, gpi_index, level_1_entry);
    end;

    result.pa = pa;
    return (GPCNoFault(), result);
end;
```

J1.4.5.56 GranuleProtectionCheck

```
// GranuleProtectionCheck()
// ========================
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

func GranuleProtectionCheck(addrdesc : AddressDescriptor, accdesc : AccessDescriptor) => GPCFRecord
begin

    assert IsFeatureImplemented(FEAT_RME);
    // The address to be checked
    let address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3().GPC == '0' then
        return GPCNoFault();
    end;

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF_Walk, 0);
    end;

    if IsFeatureImplemented(FEAT_RME_GPC2) then
        var access_disabled : boolean;

        case address.paspace of
            when PAS_Secure =>    access_disabled = GPCCR_EL3().SPAD  == '1';
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when PAS_NonSecure => access_disabled = GPCCR_EL3().NSPAD == '1';
            when PAS_Realm =>     access_disabled = GPCCR_EL3().RLPAD == '1';
            when PAS_Root =>      access_disabled = FALSE;
            otherwise =>          unreachable;
        end;

        if access_disabled then
            return GPCFault(GPCF_Fail, 0);
        end;
    end;

    // Input address size check
    if AbovePPS(address.address) then
        if (address.paspace == PAS_NonSecure ||
              (IsFeatureImplemented(FEAT_RME_GPC2) && GPCCR_EL3().APPSAA == '1')) then
            return GPCNoFault();
        else
            return GPCFault(GPCF_Fail, 0);
        end;
    end;

    if (IsFeatureImplemented(FEAT_RME_GPC3) && GPCCR_EL3().GPCBW == '1' &&
          PAWithinGPCBypassWindow(address.address)) then
        return GPCNoFault();
    end;

    // GPT base address size check
    let gpt_base : bits(NUM_PABITS) = ZeroExtend{}(GPTBR_EL3().BADDR::Zeros{12});
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF_AddressSize, 0);
    end;

    // GPT lookup
    var (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF_None then
        return gpcf;
    end;

    // Check input physical address space against GPI
    let permitted : boolean = GPICheck(address.paspace, gpt_entry.gpi, accdesc.ss);

    if !permitted then
        gpcf = GPCFault(GPCF_Fail, gpt_entry.level);
        return gpcf;
    end;

    // Check passed

    return GPCNoFault();
end;
```

J1.4.5.57 IsGranuleProtectionCheckedAccess

```
// IsGranuleProtectionCheckedAccess()
// ==================================
// Check if the access should be subject to Granule protection check returns
// true if it is, false otherwise

func IsGranuleProtectionCheckedAccess(accdesc : AccessDescriptor) => boolean
begin
    if accdesc.acctype == AccessType_DC then
        return ImpDefBool("GPC Fault on DC operations");
    end;

    return TRUE;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.58 PAWithinGPCBypassWindow

```
// PAWithinGPCBypassWindow()
// =========================
// Check if the supplied address is within a GPC Bypass window.

func PAWithinGPCBypassWindow(pa_in : bits(56)) => boolean
begin
    // Only check the top 26 bits as the minimum window size is 1GB
    let pa : bits(26) = pa_in[55:30];

    let gpcbwl : integer{} = UInt(GPCBW_EL3().BWSIZE);
    let gpcbwu : integer{} = 9 + UInt(GPCBW_EL3().BWSTRIDE);

    return pa[gpcbwu:gpcbwl] == GPCBW_EL3().BWADDR[gpcbwu:gpcbwl];
end;
```

J1.4.5.59 PGS

```
// PGS
// ===
// Physical granule size

type PGSe of enumeration {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```

J1.4.5.60 Table

```
// Table format information
// ========================
// Granule Protection Table constants

constant GPT_NoAccess : bits(4)      = '0000';
constant GPT_Table : bits(4)         = '0011';
constant GPT_Block : bits(4)         = '0001';
constant GPT_Contig : bits(4)        = '0001';
constant GPT_SystemAgent : bits(4)   = '0100';
constant GPT_NonSecureProtected : bits(4) = '0101';
constant GPT_NA6 : bits(4)           = '0110';
constant GPT_NA7 : bits(4)           = '0111';
constant GPT_Secure : bits(4)        = '1000';
constant GPT_NonSecure : bits(4)     = '1001';
constant GPT_Root : bits(4)          = '1010';
constant GPT_Realm : bits(4)         = '1011';
constant GPT_NonSecureOnly : bits(4) = '1101';
constant GPT_Any : bits(4)           = '1111';
```

J1.4.5.61 S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

readonly func S1TranslationRegime(el : bits(2)) => bits(2)
begin
    if el != EL0 then
        return el;

    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR().NS == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return EL3;
    elsif IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
        return EL2;
    else
        return EL1;
    end;
end;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR_ELx, etc.) implicitly
// return the correct value.

readonly func S1TranslationRegime() => bits(2)
begin
    return S1TranslationRegime(PSTATE.EL);
end;
```

J1.4.5.62 AddressDescriptor

```
// AddressDescriptor
// =================
// Descriptor used to access the underlying memory array.

type AddressDescriptor of record {
    fault : FaultRecord,      // fault.statuscode indicates whether the address is valid
    memattrs : MemoryAttributes,
    paddress : FullAddress,
    s1assured : boolean,      // Stage 1 Assured Translation Property
    s2fs1mro : boolean,       // Stage 2 MRO permission for Stage 1
    mecid : bits(16),         // FEAT_MEC: Memory Encryption Context ID
    vaddress : bits(64)
};

constant FINAL_LEVEL : integer = 3;
```

J1.4.5.63 ContiguousSize

```
// ContiguousSize()
// ================
// Return the number of entries log 2 marking a contiguous output range

func ContiguousSize(d128 : bit, tgx : TGx, level : integer) => integer
begin
    if d128 == '1' then
        case tgx of
            when TGx_4KB =>
                assert level IN {1, 2, 3};
                return if level == 1 then 2 else 4;
            when TGx_16KB =>
                assert level IN {1, 2, 3};
                if level == 1 then
                    return 2;
                elsif level == 2 then
                    return 4;
                else
                    return 6;
                end;
            when TGx_64KB =>
                assert level IN {2, 3};
                return if level == 2 then 6 else 4;
        end;
    else
        case tgx of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when TGx_4KB =>
                assert level IN {1, 2, 3};
                return 4;
            when TGx_16KB =>
                assert level IN {2, 3};
                return if level == 2 then 5 else 7;
            when TGx_64KB =>
                assert level IN {2, 3};
                return 5;
        end;
    end;
end;
```

J1.4.5.64 CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

func CreateAddressDescriptor(pa : FullAddress, memattrs : MemoryAttributes,
                             accdesc : AccessDescriptor) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.paddress = pa;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault(accdesc);

    return addrdesc;
end;

// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

func CreateAddressDescriptor(va : bits(64), pa : FullAddress,
                             memattrs : MemoryAttributes,
                             accdesc : AccessDescriptor) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.paddress  = pa;
    addrdesc.vaddress  = va;
    addrdesc.memattrs  = memattrs;
    addrdesc.fault     = NoFault(accdesc);
    addrdesc.s1assured = FALSE;

    return addrdesc;
end;
```

J1.4.5.65 CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// ===============================
// Set internal members for address descriptor type with values indicating error

func CreateFaultyAddressDescriptor(fault : FaultRecord) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.vaddress = fault.vaddress;
    addrdesc.fault    = fault;

    return addrdesc;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.66 DecodePASpace

```
// DecodePASpace()
// ===============
// Decode the target PA Space

func DecodePASpace(nse2 : bit, nse : bit, ns : bit) => PASpace
begin
    case nse2::nse::ns of
        when '000' =>   return PAS_Secure;
        when '001' =>   return PAS_NonSecure;
        when '010' =>   return PAS_Root;
        when '011' =>   return PAS_Realm;
        when '100' =>   return PAS_SystemAgent;
        when '101' =>   return PAS_NonSecureProtected;
        when '110' =>   return PAS_NA6;
        when '111' =>   return PAS_NA7;
        otherwise =>    unreachable;
    end;
end;
```

J1.4.5.67 DescriptorType

```
// DescriptorType
// ==============
// Translation table descriptor formats

type DescriptorType of enumeration {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

J1.4.5.68 Domains

```
// Domains
// =======
// Short-descriptor format Domains

constant Domain_NoAccess : bits(2) = '00';
constant Domain_Client   : bits(2) = '01';
constant Domain_Manager  : bits(2) = '11';
```

J1.4.5.69 FetchDescriptor

```
// FetchDescriptor()
// =================
// Fetch a translation table descriptor

func FetchDescriptor{N : integer{32, 64, 128}}(ee : bit, walkaddress : AddressDescriptor,
                                               walkaccess : AccessDescriptor,
                                               fault_in : FaultRecord) => (FaultRecord, bits(N))
begin
    var descriptor : bits(N);
    var fault : FaultRecord = fault_in;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, ARBITRARY : bits(N));
        end;
    end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    var memstatus : PhysMemRetStatus;
    (memstatus, descriptor) = PhysMemRead{N}(walkaddress, walkaccess);
    if IsFault(memstatus) then
        let iswrite : boolean = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                       walkaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    if ee == '1' then
        descriptor = BigEndianReverse{N}(descriptor);
    end;

    return (fault, descriptor);
end;
```

J1.4.5.70 HasUnprivileged

```
// HasUnprivileged()
// =================
// Returns whether a translation regime serves EL0 as well as a higher EL

readonly func HasUnprivileged(regime : Regime) => boolean
begin
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });
end;
```

J1.4.5.71 NUM_PABITS

```
// NUM_PABITS
// ==========

constant NUM_PABITS : integer{} = 56;
```

J1.4.5.72 Regime

```
// Regime
// ======
// Translation regimes

type Regime of enumeration {
    Regime_EL3,             // EL3
    Regime_EL30,            // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,             // EL2
    Regime_EL20,            // EL2&0
    Regime_EL10             // EL1&0
};
```

J1.4.5.73 RegimeUsingAArch32

```
// RegimeUsingAArch32()
// ====================
// Determine if the EL controlling the regime executes in AArch32 state

func RegimeUsingAArch32(regime : Regime) => boolean
begin
    case regime of
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Regime_EL10 => return ELUsingAArch32(EL1);
        when Regime_EL30 => return TRUE;
        when Regime_EL20 => return FALSE;
        when Regime_EL2  => return ELUsingAArch32(EL2);
        when Regime_EL3  => return FALSE;
    end;
end;
```

J1.4.5.74 S1TTWParams

```
// S1TTWParams
// ===========
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor format)

type S1TTWParams of record {
// A64-VMSA exclusive parameters
    ha : bit,         // TCR_ELx.HA
    hd : bit,         // TCR_ELx.HD
    tbi : bit,        // TCR_ELx.TBI{x}
    tbid : bit,       // TCR_ELx.TBID{x}
    nfd : bit,        // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    e0pd : bit,       // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    d128 : bit,       // TCR_ELx.D128
    aie : bit,        // (TCR2_ELx/TCR_EL3).AIE
    mair2 : MAIRType, // MAIR2_ELx
    ds : bit,         // TCR_ELx.DS
    ps : bits(3),     // TCR_ELx.{I}PS
    txsz : bits(6),   // TCR_ELx.TxSZ
    epan : bit,       // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    dct : bit,        // HCR_EL2.DCT
    nv1 : bit,        // HCR_EL2.NV1
    cmow : bit,       // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
    pnch : bit,       // TCR{2}_ELx.PnCH
    disch : bit,      // TCR{2}_ELx.DisCH
    haft : bit,       // TCR{2}_ELx.HAFT
    mtx : bit,        // TCR_ELx.MTX{y}
    skl : bits(2),    // TTBRn_ELx.SKL
    pie : bit,        // TCR2_ELx.PIE or TCR_EL3.PIE
    pir : S1PIRType,  // PIR_ELx
    pire0 : S1PIRType,// PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H == '1'
    emec : bit,       // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    amec : bit,       // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EL2.E2H == '1'
    fng : bit,        // TCR2_EL1.FNGx or TCR2_EL2.FNGx when HCR_EL2.E2H == '1'
    fngna : bit,      // TCR2_EL1.FNGx

// A32-VMSA exclusive parameters
    t0sz : bits(3),   // TTBCR.T0SZ
    t1sz : bits(3),   // TTBCR.T1SZ
    uwxn : bit,       // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    tgx : TGx,        // TCR_ELx.TGx      / Always TGx_4KB
    irgn : bits(2),   // TCR_ELx.IRGNx    / TTBCR.IRGNx or HTCR.IRGN0
    orgn : bits(2),   // TCR_ELx.ORGNx    / TTBCR.ORGNx or HTCR.ORGN0
    sh : bits(2),     // TCR_ELx.SHx      / TTBCR.SHx or HTCR.SH0
    hpd : bit,        // TCR_ELx.HPD{x}   / TTBCR2.HPDx or HTCR.HPD
    ee : bit,         // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    wxn : bit,        // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    ntlsmd : bit,     // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    dc : bit,         // HCR_EL2.DC       / HCR.DC
    sif : bit,        // SCR_EL3.SIF      / SCR.SIF
    mair : MAIRType   // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.75 S2TTWParams

```
// S2TTWParams
// ===========
// Register fields corresponding to stage 2 translation.

type S2TTWParams of record {
// A64-VMSA exclusive parameters
    ha : bit,           // VTCR_EL2.HA
    hd : bit,           // VTCR_EL2.HD
    sl2 : bit,          // V{S}TCR_EL2.SL2
    ds : bit,           // VTCR_EL2.DS
    d128 : bit,         // VTCR_ELx.D128
    sw : bit,           // VSTCR_EL2.SW
    nsw : bit,          // VTCR_EL2.NSW
    sa : bit,           // VSTCR_EL2.SA
    nsa : bit,          // VTCR_EL2.NSA
    ps : bits(3),       // VTCR_EL2.PS
    txsz : bits(6),     // V{S}TCR_EL2.T0SZ
    fwb : bit,          // HCR_EL2.FWB
    cmow : bit,         // HCRX_EL2.CMOW
    skl : bits(2),      // VTTBR_EL2.SKL
    s2pie : bit,        // VTCR_EL2.S2PIE
    s2pir : S2PIRType,  // S2PIR_EL2
    tl0 : bit,          // VTCR_EL2.TL0
    tl1 : bit,          // VTCR_EL2.TL1
    assuredonly : bit,  // VTCR_EL2.AssuredOnly
    haft : bit,         // VTCR_EL2.HAFT
    emec : bit,         // SCTLR2_EL2.EMEC
    hdbss : bit,        // VTCR_EL2.HDBSS

// A32-VMSA exclusive parameters
    s : bit,            // VTCR.S
    t0sz : bits(4),     // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    tgx : TGx,          // V{S}TCR_EL2.TG0  / Always TGx_4KB
    sl0 : bits(2),      // V{S}TCR_EL2.SL0  / VTCR.SL0
    irgn : bits(2),     // VTCR_EL2.IRGN0   / VTCR.IRGN0
    orgn : bits(2),     // VTCR_EL2.ORGN0   / VTCR.ORGN0
    sh : bits(2),       // VTCR_EL2.SH0     / VTCR.SH0
    ee : bit,           // SCTLR_EL2.EE     / HSCTLR.EE
    ptw : bit,          // HCR_EL2.PTW      / HCR.PTW
    vm : bit            // HCR_EL2.VM       / HCR.VM
};
```

J1.4.5.76 SDFType

```
// SDFType
// =======
// Short-descriptor format type

type SDFType of enumeration {
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersection,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};
```

J1.4.5.77 SecurityStateForRegime

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// SecurityStateForRegime()
// ========================
// Return the Security State of the given translation regime

func SecurityStateForRegime(regime : Regime) => SecurityState
begin
    case regime of
        when Regime_EL3 =>     return SecurityStateAtEL(EL3);
        when Regime_EL30 =>    return SS_Secure; // A32 EL3 is always Secure
        when Regime_EL2 =>     return SecurityStateAtEL(EL2);
        when Regime_EL20 =>    return SecurityStateAtEL(EL2);
        when Regime_EL10 =>    return SecurityStateAtEL(EL1);
    end;
end;
```

J1.4.5.78 StageOA

```
// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

func StageOA(ia : bits(64), d128 : bit, tgx : TGx, walkstate : TTWState) => FullAddress

begin
    // Output Address
    var oa : FullAddress;
    let tsize : integer = TranslationSize(d128, tgx, walkstate.level);
    let csize : integer = (if walkstate.contiguous == '1' then
                           ContiguousSize(d128, tgx, walkstate.level)
                           else 0);
    let ia_msb : AddressSize = (tsize + csize) as AddressSize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address[NUM_PABITS-1:ia_msb]::ia[ia_msb-1:0];

    return oa;
end;
```

J1.4.5.79 TGx

```
// TGx
// ===
// Translation granules sizes

type TGx of enumeration {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

J1.4.5.80 TGxGranuleBits

```
// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

func TGxGranuleBits(tgx : TGx) => AddressSize
begin
    case tgx of
        when TGx_4KB =>  return 12;
        when TGx_16KB => return 14;
        when TGx_64KB => return 16;
    end;
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.81 TLBContext

```
// TLBContext
// ==========
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext of record {
    ss : SecurityState,
    regime : Regime,
    vmid : bits(NUM_VMIDBITS),
    asid : bits(NUM_ASIDBITS),
    nG : bit,
    ipaspace : PASpace,    // Used in stage 2 lookups & invalidations only
    includes_s1 : boolean,
    includes_s2 : boolean,
    use_vmid : boolean,
    includes_gpt : boolean,
    ia : bits(64),         // Input Address
    tg : TGx,
    cnp : bit,
    level : integer,       // Assist TLBI level hints (FEAT_TTL)
    isd128 : boolean,
    xs : bit               // XS attribute (FEAT_XS)
};
```

J1.4.5.82 TLBRecord

```
// TLBRecord
// =========
// Translation output as a TLB payload

type TLBRecord of record {
    context : TLBContext,
    walkstate : TTWState,
    blocksize : AddressSize,  // Number of bits directly mapped from IA to OA
    contigsize : integer,     // Number of entries log 2 marking a contiguous output range
    s1descriptor : bits(128), // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    s2descriptor : bits(128)  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
};
```

J1.4.5.83 TTWState

```
// TTWState
// ========
// Translation table walk state

type TTWState of record {
    istable : boolean,
    level : integer,
    baseaddress : FullAddress,
    contiguous : bit,
    s1assured : boolean,  // Stage 1 Assured Translation Property
    s2assuredonly : bit,  // Stage 2 AssuredOnly attribute
    disch : bit,          // Stage 1 Disable Contiguous Hint
    nG : bit,
    guardedpage : bit,
    sdftype : SDFType,    // AArch32 Short-descriptor format walk only
    domain : bits(4),     // AArch32 Short-descriptor format walk only
    memattrs : MemoryAttributes,
    permissions : Permissions
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.4.5.84 TranslationRegime

```
// TranslationRegime()
// ===================
// Select the translation regime given the target EL and PE state

func TranslationRegime(el : bits(2)) => Regime
begin
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    elsif el == EL1 then
        return Regime_EL10;
    elsif el == EL0 then
        if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3) then
            return Regime_EL30;
        elsif ELIsInHost(EL0) then
            return Regime_EL20;
        else
            return Regime_EL10;
        end;
    else
        unreachable;
    end;
end;
```

J1.4.5.85 TranslationSize

```
// TranslationSize()
// =================
// Compute the number of bits directly mapped from the input address
// to the output address

func TranslationSize(d128 : bit, tgx : TGx, level : integer) => AddressSize
begin
    let granulebits : AddressSize = TGxGranuleBits(tgx);
    let descsizelog2 : integer{} = if d128 == '1' then 4 else 3;
    let blockbits : integer   = (FINAL_LEVEL - level) * (granulebits - descsizelog2);

    return (granulebits + blockbits) as AddressSize;
end;
```

J1.4.5.86 UseASID

```
// UseASID()
// =========
// Determine whether the translation context for the access requires ASID or is a global entry

func UseASID(accesscontext : TLBContext) => boolean
begin
    return HasUnprivileged(accesscontext.regime);
end;
```

J1.4.5.87 UseVMID

```
// UseVMID()
// =========
// Determine whether the translation context for the access requires VMID to match a TLB entry

func UseVMID(regime : Regime) => boolean
begin
    return regime == Regime_EL10 && EL2Enabled();
end;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part K
Appendixes

Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the Armv8 and later architectures. It contains the following sections:

- AArch32 CONSTRAINED UNPREDICTABLE behaviors.
- AArch64 CONSTRAINED UNPREDICTABLE behaviors.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

From the introduction of Armv8, the architecture defines architecturally required constraints on many behaviors that are UNPREDICTABLE in Armv7. The following sections define those constraints:

- Overview of the constraints on Armv7 UNPREDICTABLE behaviors.
- Using R13 by instruction.
- Using R15 by instruction.
- Branching into an IT block.
- Branching to an unaligned PC.
- Loads and Stores to unaligned locations.
- CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT.
- Unallocated System register access instructions.
- SBZ or SBO fields T32 and A32 in instructions.
- UNPREDICTABLE cases in immediate constants in T32 data-processing instructions.
- UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization
- Translation Table Base Address alignment.
- Handling of System register control fields for Advanced SIMD and floating-point operation.
- Mapping of non-idempotent memory locations using the Normal memory type.
- The Performance Monitors Extension.
- The Activity Monitors Extension.
- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED.
- Out of range VA.
- Instruction fetches from Device memory.
- Multi-access instructions that load the PC from Device memory.
- Programming CSSELR.Level for a cache level that is not implemented.
- Crossing a page boundary with different memory types or Shareability attributes.
- Crossing a 4KB boundary with a Device access.
- UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs.
- CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.
- Out of range values of the Set/Way/Index fields in cache maintenance instructions.
- CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set.
- CONSTRAINED UNPREDICTABLE behavior, T32 and A32 Advanced SIMD and floating-point instructions.
- CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR.
- CONSTRAINED UNPREDICTABLE behavior of EL2 features.
- Reserved values in System and memory-mapped registers and translation table entries.
- CONSTRAINED UNPREDICTABLE behavior in Debug state.

K1.1.1 Overview of the constraints on Armv7 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch32 state, where previous versions of the architecture define behavior as UNPREDICTABLE, from the introduction of Armv8-A the architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.2 Using R13 by instruction

In prior versions of the architecture, the use of R13 by instruction as a named register specifier was described as UNPREDICTABLE in the pseudocode. From the introduction of the Armv8-A architecture, the use of R13 as a named register specifier is not UNPREDICTABLE, unless this is specifically stated, and R13 can be used in the regular form. Bits[1:0] of R13 are not treated as SBZP in the Armv7 architecture or RES0 in the Armv8 and later architectures, but can hold any values programmed into them.

K1.1.3 Using R15 by instruction

All uses of R15 by instruction as a named register specifier for a source register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this Manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Read the Program Counter with the standard offset that applies for the current instruction set.
- Read the Program Counter with the standard offset that applies for the current instruction set with alignment to a word boundary.
- Read 0. This is Arm preferred behavior.
- Read or return an UNKNOWN value for the source register specified as R15.

All uses of R15 as a named register specifier for a destination register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Ignore the write.
- Branch to an UNKNOWN location in either A32 or T32 state.

Instructions that are CONSTRAINED UNPREDICTABLE when the base register is R15 and the instruction specifies a writeback of the base register, are treated as having R15 as both a source register and a destination register.

For instructions that have two destination registers, for example LDRD, MRRC, and many of the multiply instructions, if Rt, Rt2, RdLo, or RdHi is R15, then the other destination register of the pair is UNKNOWN if the CONSTRAINED UNPREDICTABLE behavior for the write to R15 is either to ignore the write or to branch to an UNKNOWN location.

For instructions that affect any or all of PSTATE.{N, Z, C, V}, PSTATE.Q, and PSTATE.GE when the register specifier is not R15, any flags affected by an instruction that is CONSTRAINED UNPREDICTABLE when the register specifier is R15 become UNKNOWN.

In addition, for MRC instructions that use R15 as the destination register descriptor, and therefore target APSR_nzcv where these are described as being CONSTRAINED UNPREDICTABLE, PSTATE.{N, Z, C, V} becomes UNKNOWN.

K1.1.4 Branching into an IT block

Branching into an IT block leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address determined by the branch, but each instruction in the IT block is:

- Executed as if it were not in an IT block. This means that it is executed unconditionally.
- Executed as if it had passed its Condition code check within an IT block.
- Executed as a NOP. That is, it behaves as if it had failed the Condition code check.

K1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word-aligned and is defined to be CONSTRAINED UNPREDICTABLE, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The unaligned address generates a Prefetch Abort on the first instruction using the unaligned PC value.

  If that instruction is executed at EL0 and either of the following applies, the exception is taken to EL2:

  — EL2 is using AArch32 and the value of HCR.TGE is 1.

  — EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

  If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1.

  If the instruction is executed at an Exception level that is higher than EL0 the exception is taken to the Exception level at which the instruction was executed.

  In all cases, the exception is generated only if the first instruction using the unaligned PC value is architecturally executed.

If the exception that results from a branch to an unaligned PC value:

- Is taken to an Exception level that is using AArch64, it is reported as a PC alignment fault exception, see ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.
- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception, see Prefetch Abort exception reporting a PC alignment fault exception.

Note

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of IFAR, HIFAR, or FAR_ELx is 0 for all these cases.

K1.1.6 Loads and Stores to unaligned locations

Some unaligned loads and stores in the Armv8-A and later architectures are described as CONSTRAINED UNPREDICTABLE to do one of the following:

- Take an alignment fault.
- Perform the specified load or store to the unaligned memory location.

K1.1.7 CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT

A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception results.
- The instructions are executed as if they had passed the Condition code check.
- The instructions execute as NOPs. This means that they behave as if they had failed the Condition code check.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the PSTATE.IT state machine must be treated in one of the following ways:

- PSTATE.IT is cleared to 0.
- PSTATE.IT advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This does not apply to an instruction that is the last instruction in an IT block.

The instructions addressed by the updated PC must do one of the following:

- Execute as if they had passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they had failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or, if the instructions are part of another IT block, in accordance with the behavior described in Branching into an IT block.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

For exception returns or Debug state exits that cause PSTATE.IT to be set to a reserved value in T32 state or that return to A32 state with a nonzero value in PSTATE.IT, the PSTATE.IT bits are forced to `0b00000000`. The reserved values are:

```
PSTATE.IT[7:4] != '0000' && PSTATE.IT[3:0] == '0000'
PSTATE.IT[2:0] != '000' && (SCTLR().ITD == '1' || SCTLR_ELx().ITD == '1')
```

Exception returns or Debug state exits that set PSTATE.IT to a non-reserved value in T32 state can occur when the flow of execution returns to a point:

- Outside an IT block, but with the PSTATE.IT bits set to a value other than `0b00000000`.
- Inside an IT block, but with a different value of the PSTATE.IT bits than if the IT block had been executed without an exception return or Debug state exit.

In this case, the instructions at the target of the exception return or Debug state exit must do one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or as if the instruction were part of another IT block, in accordance with the behavior in Branching into an IT block.

The remaining iterations of the PSTATE.IT state machine must behave in one of the following ways:

- The PSTATE.IT state machine advances as if it were in an IT block.
- The PSTATE.IT bits are ignored.
- The PSTATE.IT bits are forced to `0b00000000`.

K1.1.8 Unallocated System register access instructions

From the introduction of the Armv8-A architecture, accesses to unallocated System register encodings are UNDEFINED.

This includes:

- Reads using encodings that are defined as WO.
- Writes using encodings that are defined as RO.
- `MCR` or `MRC` accesses to using a set of {`coproc`, `CRn`, `opc1`, `CRm`, `opc2`} values that the Armv7 architecture defined as UNPREDICTABLE.
- `MCRR` and `MRRC` instructions with unallocated values of `opc1` or `CRm` that are described as UNPREDICTABLE are UNDEFINED in the Armv8-A and later architectures.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.9 SBZ or SBO fields T32 and A32 in instructions

Many of the T32 and A32 instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO. If the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- LDM, LDMIA, LDMFD.
- LDMDB, LDMEA.
- LDR (literal).
- LDRB (literal).
- LDRD (immediate).
- LDRD (register).
- LDRD (literal).
- LDRH (literal).
- LDRSB (literal).
- LDRSH (literal).
- POP.
- PUSH.
- SDIV.
- STM, STMIA, STMEA.
- STMDB, STMFD.
- UDIV.

K1.1.10 UNPREDICTABLE cases in immediate constants in T32 data-processing instructions

The description of immediate constants in T32 data processing Modified immediate constants in T32 instructions include constant values that were UNPREDICTABLE in Armv7. Instruction encodings describes 32-bit T32 instructions as {`hw1`, `hw2`}, where `hw1` is the left-hand halfword in the 32-bit encoding diagram for the instruction. The UNPREDICTABLE cases are those where both:

- `hw2`[7:0] == `0b0000000`.
- `hw1`[10] == `0` and either:
  - `hw2`[14:12] == `0b001`.
  - `hw2`[14:12] == `0b010`.
  - `hw2`[14:12] == `0b011`.

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce the value `0b0000000`.

K1.1.11 UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions

The description of immediate constants in Modified immediate constants in T32 and A32 Advanced SIMD instructions include constant values that were UNPREDICTABLE in Armv7. The UNPREDICTABLE cases are those where:

- The bits that the encoding diagram shows as `abcd` are all 0.
  In the A32 encoding these are bits[24, 18:6, 3:0]. In the T32 encoding they are bits {`hw1`[12, 2:0], `hw2`[3:0]}.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The bits that the encoding diagram shows as `cmode[3:1]` are one of {`0b001`, `0b010`, `0b011`, `0b101`, `0b110`}. In the A32 encoding these are bits[11:9]. In the T32 encoding they are bits `hw2`[11:9].

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce an immediate constant value of zero.

K1.1.12 CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values

The Arm architecture allows copies of System register control or data values to be cached in a cache or TLB. This can lead to CONSTRAINED UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the Effective value of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that Effective value must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.
- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits within a TLB might generate an exception, reporting the exception using the TLB Conflict fault code, see TLB conflict aborts.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

K1.1.13 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to CONSTRAINED UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

K1.1.14 Unallocated values with register fields of CP15 registers and Translation Table entries

Unless stated elsewhere, all unallocated or reserved values of fields with allocated values within CP15 registers and Translation Table entries be have in one in one of the following ways:

- The encoding maps onto any of the allocated values but otherwise does not cause UNPREDICTABLE behavior.
- The encodings cause effects that could be achieved by a combination of more than one of the allocated encodings.
- The encodings cause the field to have no functional effect.

K1.1.15 Translation Table Base Address alignment

A misaligned Translation Table Base Address can occur if:

- The VMSAv8-32 Short-descriptor translation table format is enabled and TTBR0[13-N:7], which is defined to be RES0, contains a nonzero value.
- The VMSAv8-32 Long-descriptor translation table format is enabled, and TTBR0[x-1:3], TTBR1[x-1:3], HTTBR[x-1:3], or VTTBR[x-1:3], which are defined to be RES0, contains a nonzero value.

In the event of a misaligned Translation Table Base Address, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all bits were zero:
  - The value that is read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using that register might be corrupted in those bits that are nonzero.

K1.1.16 Handling of System register control fields for Advanced SIMD and floating-point operation

For historical reasons described in Background to the System register interface, each of the CPACR, HCPTR, and NSACR has a pair of control fields that were defined to have identical functionality for controlling Advanced SIMD and floating-point operation. These fields are:

- CPACR.{cp10, cp11}.
- HCPTR.{TCP10, TCP11}.
- NSACR.{cp10, cp11}.

The architecture requires that both fields in one of these pairs are programmed to the same value. If this is not done, then the CONSTRAINED UNPREDICTABLE behavior is that behavior is the same as if the cp11 or TCP11 control field was equal to the cp10 or TCP10 field. This is in all respects except for the value read back by a direct read of the register.

After a register write that writes different values to the two fields of a pair, a direct read of the register might return an UNKNOWN value for the cp11 or TCP11 field.

Note

This means that, when different values are written to the {cp10, cp11} fields in a single register, the architecture permits but does not require that a read of that register returns the value written to the cp11 field.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.16.1 CONSTRAINED UNPREDICTABLE CPACR and NSACR settings

If CPACR.cp<n> contains the encoding `'10'`, then one of the following behaviors must occur:

- The encoding maps onto any of the allocated values, but otherwise does not cause UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

Note

In Armv7, CPACR had a D32DIS bit, and NSACR had an NSD32DIS bit. There is no CPACR.D32DIS or NSACR.NSD32DIS in Armv8-A and later architectures, and the corresponding bits in the two registers are RES0.

K1.1.17 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in the following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.1.18 The Performance Monitors Extension

I_{WKXBB} The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch32 state:

- CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR.
- CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>.
- CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN.

K1.1.18.1 CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR

R_{XSVNC} When FEAT_FGT is not implemented and if PMSELR.SEL is greater than the number of event counters accessible at the current Exception level, then accesses to PMXEVTYPER or PMXEVCNTR can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when any of the following is true:

- If PMSELR.SEL is not equal to 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in EL2 or EL3.
- If FEAT_SEL2 is disabled or is not implemented, PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to HDCR.HPMN, and the PE is executing in EL1 or EL0.

In these cases, the PE does one of the following:

- Accesses to PMXEVTYPER or PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVTYPER or PMXEVCNTR execute as NOPs.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of accessible event counters but greater than the number of accessible counters at this Exception level, access to PMXEVTYPER or PMXEVCNTR from EL1 or permitted access from EL0 is trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If EL2 is implemented and enabled in the current Security state, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of accessible event counters.
Accesses from EL0 to PMXEVCNTR are permitted when:

- EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

Accesses from EL0 to PMXEVTYPER are permitted when:

- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

I_{HNSSP} When FEAT_FGT is not implemented and if PMSELR.SEL is equal to 31, the PE does one of the following:

- Accesses to PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVCNTR execute as NOPs.
- Accesses to PMXEVCNTR behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, for an access to PMXEVCNTR from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

K1.1.18.2 CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>

I_{GRBYY} If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n> and PMEVTYPER<n> are not CONSTRAINED UNPREDICTABLE.

I_{GRBHK} When FEAT_FGT is not implemented and if <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of PMEVCNTR<n> and PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the PE does one of the following:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n> or PMEVTYPER<n> from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.
  Accesses from EL0 are permitted to PMEVCNTR<n> when:
  — EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
  — EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.
  Accesses from EL0 are permitted to PMEVTYPER<n> when:
  — EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  — EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

Note

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of accessible event counters.

K1.1.18.3 CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN

R_{ZLGPG} If EL2 is implemented, NUM_PMU_COUNTERS is nonzero, and any of the following is true:

- FEAT_HPMN0 is not implemented and HDCR.HPMN is 0.
- FEAT_PMUv3_EXTPMN is not implemented and HDCR.HPMN is set to a value greater than NUM_PMU_COUNTERS.

Then the following CONSTRAINED UNPREDICTABLE behaviors apply:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN nonzero value less than or equal to NUM_PMU_COUNTERS.
  - All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

K1.1.18.4 CONSTRAINED UNPREDICTABLE behavior caused by PMCCR.EPMN

R_{SMXTG} If FEAT_PMUv3_EXTPMN is implemented and PMCCR.EPMN is greater than NUM_PMU_COUNTERS, then all of the following apply:

- External reads of PMCCR.EPMN returns an UNKNOWN value that is less than or equal to 31.
- For the purposes of indirect reads of PMCCR.EPMN as a result of the following reads, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to 31:
  - Direct reads of PMCR.N.
  - Direct reads of HDCR.HPMN.
  - External reads of PMCFGR.N.
  - If FEAT_PMUv3_ICNTR is implemented, external reads of PMCGCR0.CG0NC.
- For all other purposes, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to NUM_PMU_COUNTERS.

K1.1.19 The Activity Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Activity Monitors registers in AArch32 state:

- CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>.
- CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>.
- CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1.

K1.1.19.1 CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>

If <n> is greater than the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> and AMEVTYPER0<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

K1.1.19.2 CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>

If <n> is greater than the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n> and AMEVTYPER1<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.19.3 CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of AMCNTENCLR1 and AMCNTENSET1 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

The number of auxiliary activity monitor event counters that are implemented is zero exactly when AMCFGR.NCG is `0b0000`.

K1.1.20 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, this generates an exception:

- If this exception is taken to an Exception level that is using AArch64, then ESR_ELx is UNKNOWN.
- If this exception is taken to EL2 and EL2 is using AArch32, then the HSR is unknown.

Note

The value written to ESR or HSR must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but resulted from a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.1.21 Out of range VA

If the PE executes an instruction for which the instruction address, size, and alignment mean it contains the bytes `0xFFFF_FFFF` and `0x0000_0000`, then the bytes that wrap around and appear to be from `0x0000_0000` onwards come from an UNKNOWN address.

If the PE executes a load or store instruction for which the calculated address, total access size, and alignment mean it accesses bytes `0xFFFF_FFFF` and `0x0000_0000`, then the bytes that wrap around and appear to be from `0x0000_0000` onwards come from an UNKNOWN address.

K1.1.22 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location when address translation is enabled.

If a branch causes the Program Counter to point to a location in memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

K1.1.23 Multi-access instructions that load the PC from Device memory

Multi-access instructions that load the PC from Device memory when address translation is enabled are UNPREDICTABLE in AArch32 state. From the introduction of the Armv8-A architecture in AArch32 state an implementation must perform one of the following behaviors:

- It loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- It generates a Permission fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.24 Programming CSSELR.Level for a cache level that is not implemented

If CSSELR.Level is programmed to a cache level that is not implemented, then a read of CSSELR returns an UNKNOWN value in CSSELR.Level.

If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR an implementation must perform one of the following behaviors:

- The CCSIDR read is treated as a NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

When FEAT_CCIDX is implemented, CCSIDR2 is implemented. If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR2 an implementation must perform one of the following behaviors:

- The CCSIDR2 read is treated as a NOP.
- The CCSIDR2 read is UNDEFINED.
- The CCSIDR2 read returns an UNKNOWN value.

K1.1.25 Crossing a page boundary with different memory types or Shareability attributes

If a single load or store instruction generates multiple memory accesses, such that the total set of accesses crosses a page boundary to a memory location that has a different memory type, Normal or Device, or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an alignment fault caused by the memory type.
  For the Non-secure PL1&0 translation regime:
  - If the stage 1 translation causes the mismatch, the resulting exception is taken to PL1.
  - If the stage 2 translation causes the mismatch, the resulting exception is taken to PL2.
  - If both stages of translation cause the mismatch, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

K1.1.26 Crossing a 4KB boundary with a Device access

A memory access from a load or store instruction to Device memory that crosses a 4KB boundary results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses, except that there is no guarantee of ordering between memory accesses.
- The instruction generates an Alignment fault caused by the memory type.
  For the Non-secure PL1&0 translation regime:
  - If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to PL1.
  - If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to PL2.
  - If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

Note

The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.27 UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. These cases were UNPREDICTABLE in Armv7. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in Load-Exclusive and Store-Exclusive instruction usage restrictions for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

Note

Additional CONSTRAINED UNPREDICTABLE cases can apply to Load-Exclusive and Store-Exclusive instructions, see CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set.

K1.1.28 CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings

The T32 and A32 instruction sets include encodings that result in CONSTRAINED UNPREDICTABLE behavior when they are decoded.

K1.1.28.1 CONSTRAINED UNPREDICTABLE behavior of CRC32 instruction encodings

In the T32 and A32 instruction sets, there are encodings of the CRC32 and CRC32C instructions that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in the following places in the T32 and A32 instruction sets:

- Cyclic Redundancy Check for the A32 instruction set, with `sz = 11`.
- Data-processing (two source registers) for the T32 instruction set, with `op1 = 10x` and `op2 = 11`.

The CONSTRAINED UNPREDICTABLE behavior for these encodings is described in CRC32 and CRC32C.

K1.1.28.2 CONSTRAINED UNPREDICTABLE behavior of other A32 instruction encodings

In the A32 instruction set, there are encodings that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in:

- Miscellaneous.
- Memory hints and barriers.
- Barriers.

The CONSTRAINED UNPREDICTABLE behavior is that an implementation must treat the encodings in one of the following ways:

- The instruction encoding is UNDEFINED.
- The instruction encoding executes as a NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.29 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Note

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the A64 cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW.

K1.1.30 CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different T32 and A32 System instructions.

Note

If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

K1.1.30.1 SRS (T32)

For a description of this instruction and the encoding, see SRS, SRSDA, SRSDB, SRSIA, SRSIB.

K1.1.30.1.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

K1.1.30.2 SRS (A32)

For a description of this instruction and the encoding, see SRS, SRSDA, SRSDB, SRSIA, SRSIB.

K1.1.30.2.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

K1.1.30.3 SUBS PC, LR and related instructions (T32)

For a description of this instruction and the encoding, see the exception return form of SUB, SUBS (immediate).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.30.3.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

For encoding T5:

- If `hw1`[3:0] are not `0b1110`, and the instruction is executed when not in Hyp mode, System mode, or User mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a NOP.
  - The instruction is treated as if `hw1`[3:0] are `0b1110`.
  - The Program Counter is set using the value in the register specified by `hw1`[3:0].

K1.1.30.4 SUBS PC. LR and related instructions (A32)

For a description of this instruction and the encoding, see the exception return forms of MOV, MOVS (register) and SUB, SUBS (immediate).

K1.1.30.4.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31 CONSTRAINED UNPREDICTABLE behavior, T32 and A32 Advanced SIMD and floating-point instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different T32 and A32 Advanced SIMD and floating-point instructions listed in Alphabetical list of Advanced SIMD and floating-point instructions.

Note

- The pseudocode used in this section to describe cases that can result in CONSTRAINED UNPREDICTABLE behavior does not necessarily match the encoding-specific pseudocode for a specific instruction.
- If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

K1.1.31.1 VCVT (between floating-point and fixed-point)

For a description of this instruction and the encoding, see VCVT (between floating-point and fixed-point, floating-point).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.2 VLD1 (multiple single elements)

For a description of this instruction and the encoding, see VLD1 (multiple single elements).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.3 VLD1 (single element to all lanes)

For a description of this instruction and the encoding, see VLD1 (single element to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.4 VLD2 (multiple 2-element structures)

For a description of this instruction and the encoding, see VLD2 (multiple 2-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.5 VLD2 (single 2-element structure to one lane)

For a description of this instruction and the encoding, see VLD2 (single 2-element structure to one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.6 VLD2 (single 2-element structure to all lanes)

For a description of this instruction and the encoding, see VLD2 (single 2-element structure to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31.7 VLD3 (multiple 3-element structures)

For a description of this instruction and the encoding, see VLD3 (multiple 3-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.8 VLD3 (single 3-element structure to one lane)

For a description of this instruction and the encoding, see VLD3 (single 3-element structure to one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.9 VLD3 (single 3-element structure to all lanes)

For a description of this instruction and the encoding, see VLD3 (single 3-element structure to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.10 VLD4 (multiple 4-element structures)

For a description of this instruction and the encoding, see VLD4 (multiple 4-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.11 VLD4 (single 4-element structure to one lane)

For a description of this instruction and the encoding, see VLD4 (single 4-element structure to one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.12 VLD4 (single 4-element structure to all lanes)

For a description of this instruction and the encoding, see VLD4 (single 4-element structure to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.13 VLDM

For a description of this instruction and the encoding, see VLDM, VLDMDB, VLDMIA.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31.14 VMOV (between two general-purpose registers and two single-precision registers)

For a description of this instruction and the encoding, see VMOV (between two general-purpose registers and two single-precision registers).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.15 VMOV (between two general-purpose registers and a doubleword floating-point register)

For a description of this instruction and the encoding, see VMOV (between two general-purpose registers and a doubleword floating-point register).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.16 VST1 (multiple single elements)

For a description of this instruction and the encoding, see VST1 (multiple single elements).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.17 VST2 (multiple 2-element structures)

For a description of this instruction and the encoding, see VST2 (multiple 2-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.18 VST2 (single 2-element structure from one lane)

For a description of this instruction and the encoding, see VST2 (single 2-element structure from one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.19 VST3 (multiple 3-element structures)

For a description of this instruction and the encoding, see VST3 (multiple 3-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.20 VST3 (single 3-element structure from one lane)

For a description of this instruction and the encoding, see VST3 (single 3-element structure from one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31.21 VST4 (multiple 4-element structures)

For a description of this instruction and the encoding, see VST4 (multiple 4-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.22 VST4 (single 4-element structure from one lane)

For a description of this instruction and the encoding, see VST4 (single 4-element structure from one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.23 VSTM

For a description of this instruction and the encoding, see VSTM, VSTMDB, VSTMIA.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.32 CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR

The following subsections describe the CONSTRAINED UNPREDICTABLE behavior associated with programming the VTCR:

- Misprogramming VTCR.S.
- Misprogramming VTCR.{SL0, T0SZ}.

K1.1.32.1 Misprogramming VTCR.S

VTCR.S must be programmed to the value of T0SZ[3], or the effect is CONSTRAINED UNPREDICTABLE. From the introduction of the Armv8-A architecture, if VTCR.S is not programmed correctly, then the VTCR.T0SZ value is treated as an UNKNOWN value.

Note

The CONSTRAINED UNPREDICTABLE behavior described in Misprogramming VTCR.{SL0, T0SZ} means the UNKNOWN VTCR.T0SZ value might generate a Translation fault.

K1.1.32.2 Misprogramming VTCR.{SL0, T0SZ}

If the stage 2 input address size, as programmed in VTCR.T0SZ, is out of range with respect to the starting level, as programmed in the VTCR.SL0 field, or the VTCR.SL0 field is programmed to a reserved value, then at the time of a translation walk that uses the stage 2 translation, a stage 2 level 1 Translation Fault is generated.

K1.1.33 CONSTRAINED UNPREDICTABLE behavior of EL2 features

The following sections, and the instruction descriptions, describe CONSTRAINED UNPREDICTABLE behavior that can occur in an implementation that includes EL2 where EL2 can use AArch32:

- ERET in User mode or System mode.
- Illegal entry into, or exit from, Hyp mode.
- Exception return to Hyp mode.
- Stage 1 default memory type.
- Trapping of general exceptions to Hyp mode.
- MSR (banked register) and MRS (banked register).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.33.1 ERET in User mode or System mode

If ERET is executed in User mode or System mode, it behaves as described in SUBS PC, LR and related instructions (T32).

K1.1.33.2 Illegal entry into, or exit from, Hyp mode

Illegal entry to Hyp mode or exit from Hyp mode might be due to, for example:

- Modifying PSTATE.M when in Hyp mode.
- Secure state use of Hyp mode.

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in SRS (T32) and SRS (A32).

K1.1.33.3 Exception return to Hyp mode

Exception returns to Hyp mode when SCR.NS == 0 or from a Non-secure PL1 mode invokes the illegal exception return.

K1.1.33.4 Stage 1 default memory type

If HCR.DC == 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with:

- SCTLR.M == 0, regardless of the actual value of SCTLR.M, other than for the value returned by an explicit read of SCTLR.M.
- HCR.VM == 1, regardless of the actual value of HCR.VM, other than for an explicit read of this bit.

K1.1.33.5 Trapping of general exceptions to Hyp mode

Attempting to perform an exception return to a Non-secure PL1 mode when HCR.TGE == 1 invokes an illegal exception return.

Attempting to change from Monitor mode to a Non-secure PL1 mode when HCR.TGE == 1 by executing a `CPS` or `MSR` instruction generates an Illegal Execution state exception, by not changing the mode, and setting PSTATE.IL to 1.

When EL3 is using AArch32, attempting to change from a Secure PL1 mode to a Non-secure PL1 mode when HCR.TGE is 1, by changing SCR.NS from 0 to 1, results in no change of SCR.NS.

Because taking an exception into Non-secure PL1 modes leads to a CONSTRAINED UNPREDICTABLE situation, the following additional properties apply when HCR.TGE == 1:

- All exceptions that would be routed to EL1 are routed to EL2.
- Non-secure SCTLR.M is treated as being 0, regardless of its actual value, other than for an explicit read of this bit.
- HCR.FMO, HCR.IMO, and HCR.AMO are treated as being 1, regardless of their actual value, other than for an explicit read of these bits.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.

K1.1.33.6 MSR (banked register) and MRS (banked register)

If the target register specified by the {`R`, `SYSm`} fields of the instruction encoding is not accessible from the PE mode in which the instruction was executed (see Usage restrictions on the banked register transfer instructions), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For `MRS` (banked register) instructions, the destination general-purpose register becomes UNKNOWN.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For MSR (banked register) instructions, if the register specified could be accessed from the current mode by other mechanisms, then this register is UNKNOWN. Otherwise, the instruction is a NOP.

If the instruction was executed specifying an unallocated {R, SYSm} field value or an unimplemented register (see Encoding the register argument in the banked register transfer instructions), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- An allocated MRS (banked register) or MSR (banked register) instruction is executed.

K1.1.34 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch64 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

K1.1.35 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors

It contains the following sections:

- Overview of the constraints on AArch64 UNPREDICTABLE behaviors.
- SBZ or SBO fields in A64 instructions.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization.
- Translation table base address alignment.
- The Performance Monitors Extension.
- The Activity Monitors Extension.
- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED.
- Out of range virtual address.
- Mapping of non-idempotent memory locations using the Normal memory type.
- Instruction fetches from Device memory.
- Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented.
- Crossing a page boundary with different memory types or Shareability attributes.
- Crossing a peripheral boundary with a Device access.
- CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs.
- CONSTRAINED UNPREDICTABLE behavior for A64 instructions.
- Out of range values of the Set/Way/Index fields in cache maintenance instructions.
- Reserved values in System and memory-mapped registers and translation table entries.
- CONSTRAINED UNPREDICTABLE behavior in Debug state.

K1.2.1 Overview of the constraints on AArch64 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch64 state, the Armv8-A and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases being handled in any way other than those listed under the heading CONSTRAINED UNPREDICTABLE.

K1.2.2 SBZ or SBO fields in A64 instructions

Some A64 instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO, as described in Fixed values in AArch64 instruction and System register descriptions. Except for specific cases identified in CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs, if the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.3 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The Arm architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the Effective value of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that Effective value must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.
- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits in a TLB might generate an exception, reporting the exception using the TLB conflict fault code, see TLB conflict abort.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

K1.2.4 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.5 Translation table base address alignment

In the translation table base registers TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, VTTBR_EL2, and TTBR0_EL3, register bits[48:*x*] hold the translation table base address, where *x* depends on the translation table granule size and the size of the addressed translation table, as described in Translation granules. Register bits[(*x*-1):0], unless redefined for another purpose, correspond to bits[(*x*-1):0] of the translation table base address and therefore are RES0.

Note

- When FEAT_LPA is implemented and the 64KB granule size is used, register bits[5:2] are redefined to hold bits[51:48] of the translation table base address.
- When FEAT_TTCNP is implemented register bit[0] is redefined as the CnP bit.

For these registers, if one or more RES0 bits in register bits [(x-1):0] does not have a value of 0, this can result in a misaligned translation table base address. In this case, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all the bits had a value of 0:
  - The value read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using those registers might be corrupted in those bits that are nonzero.

For more information, see the appropriate TTBR.BADDR field description.

K1.2.6 The Performance Monitors Extension

I_{GTRGP} The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch64 state:

- CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0.
- CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0.
- CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN.

K1.2.6.1 CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0

R_{TVKJV} When FEAT_FGT is not implemented, if the PE is using AArch64, and if PMSELR_EL0.SEL is greater than the number of event counters accessible at the current Exception level, then accesses to PMXEVTYPER_EL0 and PMXEVCNTR_EL0 can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when any of the following is true:

- If PMSELR_EL0.SEL is not equal to 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in EL2 or EL3.
- If FEAT_SEL2 is disabled or is not implemented, PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to MDCR_EL2.HPMN, and the PE is executing in EL0 or EL1.

In these cases, the PE does one of the following:

- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 are UNDEFINED.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behave as RAZ/WI.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 execute as NOPs.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of accessible event counters but greater than or equal to the number of accessible counters at this Exception level, access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from EL1 or a permitted access from EL0 is trapped to EL2.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If EL2 is implemented and enabled in the current Security state, MDCR_EL2.HPMN identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of accessible event counters. Accesses from EL0 to PMXEVCNTR_EL0 are permitted when:

- EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

Accesses from EL0 to PMXEVTYPER_EL0 are permitted when:

- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

R_{CJXGK} When FEAT_FGT is not implemented, if the PE is using AArch64, and if PMSELR_EL0.SEL is equal to 31, the PE does one of the following:

- Accesses to PMXEVCNTR_EL0 are UNDEFINED.
- Accesses to PMXEVCNTR_EL0 behave as RAZ/WI.
- Accesses to PMXEVCNTR_EL0 execute as NOPs.
- Accesses to PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an unknown value that is less than the number of counters accessible at the current Exception level and Security state.

K1.2.6.2 CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0

I_{MBTNR} If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are not CONSTRAINED UNPREDICTABLE.

R_{QRHGP} When FEAT_FGT is not implemented, if the PE is using AArch64, and if <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the PE does one of the following:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n>_EL0 or PMEVTYPER<n>_EL0 from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.
  Accesses from EL0 to PMEVCNTR<n>_EL0 are permitted when:
  — EL1 is using AArch32 and the value of PMUSERENR.{ER, EN} are both 1.
  — EL1 is using AArch64 and the value of PMUSERENR_EL0.{ER, EN} are both 1.
  Accesses from EL0 to PMEVTYPER<n>_EL0 are permitted when:
  — EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  — EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

K1.2.6.3 CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN

R_{LGBTC} If EL2 is implemented, NUM_PMU_COUNTERS is nonzero, and any of the following is true:

- FEAT_HPMN0 is not implemented and MDCR_EL2.HPMN is 0.
- FEAT_PMUv3_EXTPMN is not implemented and MDCR_EL2.HPMN is greater than NUM_PMU_COUNTERS.

Then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  — An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if MDCR_EL2.HPMN is an UNKNOWN nonzero value less than or equal to NUM_PMU_COUNTERS.
  — All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.6.4 CONSTRAINED UNPREDICTABLE behavior caused by PMCCR.EPMN

RKJLJH If FEAT_PMUv3_EXTPMN is implemented and PMCCR.EPMN is greater-than NUM_PMU_COUNTERS, then all of the following apply:

- External reads of PMCCR.EPMN returns an UNKNOWN value that is less than or equal to 31.
- For the purposes of indirect reads of PMCCR.EPMN as a result of the following reads, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to 31:
  - Direct reads of PMCR_EL0.N or PMCR.N.
  - Direct reads of MDCR_EL2.HPMN.
  - External reads of PMCFGR.N.
  - If FEAT_PMUv3_ICNTR is implemented, external reads of PMCGCR0.CG0NC.
- For all other purposes, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to NUM_PMU_COUNTERS.

K1.2.7 The Activity Monitors Extension

If <n> is greater than the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n>_EL0 and AMEVTYPER0<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

If <n> is greater than the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n>_EL0 and AMEVTYPER1<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of AMCNTENCLR1_EL0 and AMCNTENSET0_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

The number of auxiliary activity monitor event counters that are implemented is zero exactly when AMCFGR_EL0.NCG is `0b0000`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.8 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, ESR_ELx is UNKNOWN.

Note

The value written to ESR_ELx must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but was the result of a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.2.9 Out of range virtual address

If the PE executes a load or store instruction with tagged addressing disabled in the current translation regime, and where the calculated virtual address, total access size, and alignment mean that it accesses the bytes at `0xFFFF_FFFF_FFFF_FFFF` and `0x0000_0000_0000_0000`, then the bytes that appear to be from `0x0000_0000_0000_0000` onwards are accessed at an UNKNOWN address. It is permitted for the UNKNOWN address used to be a fixed value such that it always generates an MMU fault.

If the PE executes a load or store instruction with tagged addressing enabled in the current translation regime, and where the calculated address, total access size, and alignment mean that it accesses the bytes at `0xFFFF_FFFF_FFFF_FFFF` and `0x0000_0000_0000_0000`, then the bytes that appear to be from `0x0000_0000_0000_0000` onwards are accessed at an UNKNOWN address and the tags associated with address also become UNKNOWN. It is permitted for the UNKNOWN address used to be a fixed value such that it always generates an MMU fault.

Note

Because of Program Counter alignment constraints, it is impossible for a PE to fetch an A64 instruction that includes both the byte at virtual address `0xFFFF_FFFF_FFFF_FFFF` and the byte at virtual address `0x0000_0000_0000_0000`.

K1.2.10 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.2.11 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location at times when address translation is enabled.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- Take a Permission fault.

K1.2.12 Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented

If the CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 returns an UNKNOWN value in CSSELR_EL1.{Level, InD, TnD}.

If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR_EL1 an implementation must perform one of the following behaviors:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CCSIDR_EL1 read is treated as a NOP.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

When FEAT_CCIDX is implemented, CCSIDR2_EL1 is implemented. If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR2_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR2_EL1 read is treated as a NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

K1.2.13 Crossing a page boundary with different memory types or Shareability attributes

If a single load or store instruction generates multiple memory accesses, such that the total set of accesses crosses a page boundary to a memory location that has a different memory type, Normal or Device, or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type.
  For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
  - If the stage 2 translation generated the mismatch, the resulting exception is taken to EL2.
  - If both stages of translation generate the mismatch, the exception can be taken to either EL1 or EL2.
- The instruction executes as a NOP.

Memory accesses from a Memory Copy and Memory Set CPY* or Memory Copy and Memory Set SET* instruction that cross a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type. For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
  - If the stage 2 translation generated the mismatch, the resulting exception is taken to EL2.
  - If both stages of translation generate the mismatch, the exception can be taken to either EL1 or EL2.

K1.2.14 Crossing a peripheral boundary with a Device access

Performing memory accesses from one load or store instruction to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The instruction generates an alignment fault caused by the memory type.
  For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to EL1.
  - If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to EL2.
  - If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either EL1 or EL2.
- The instruction executes as a NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The boundary referred to is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

K1.2.15 CONSTRAINED UNPREDICTABLE behaviors with SVE memory accesses

The CONSTRAINED UNPREDICTABLE behaviors that are associated with memory accesses due to loads and stores also apply to SVE vector load and store instructions. The CONSTRAINED UNPREDICTABLE behaviors are described in Crossing a page boundary with different memory types or Shareability attributes and Crossing a peripheral boundary with a Device access.

The CONSTRAINED UNPREDICTABLE behaviors apply to the following SVE memory accesses:

- When an SVE unpredicated contiguous load or store instruction accesses an address range that crosses a boundary between memory types, Shareability attributes, or peripherals.
- When an SVE predicated contiguous load or store instruction performs memory accesses that are associated with Active elements on both sides of a boundary between different memory types, Shareability attributes, or peripherals.
- When an SVE predicated non-contiguous load or store instruction performs a memory access that is associated with an Active element that crosses a boundary between memory types, Shareability attributes, or peripherals.

Memory addresses that are associated with *Inactive elements* cannot trigger CONSTRAINED UNPREDICTABLE behaviors.

If SVE vector loads and stores trigger a CONSTRAINED UNPREDICTABLE behavior that then generates an alignment fault, the fault is handled in the same as any other synchronous memory fault caused by an SVE load or store instruction.

K1.2.16 CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in Load-Exclusive and Store-Exclusive instruction usage restrictions for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

K1.2.17 CONSTRAINED UNPREDICTABLE behavior for A64 instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A64 instructions listed in A64 Base Instruction Descriptions and A64 Advanced SIMD and Floating-point Instruction Descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.1 128-bit Atomic Memory Operations

For a description of these instructions and encodings, see :

- LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL
- LDSETP, LDSETPA, LDSETPAL, LDSETPL
- RCWCLRP, RCWCLRPA, RCWCLRPAL, RCWCLRPL
- RCWSETP, RCWSETPA, RCWSETPAL, RCWSETPL
- RCWSSWPP, RCWSSWPPA, RCWSSWPPAL, RCWSSWPPL
- RCWSCLRP, RCWSCLRPA, RCWSCLRPAL, RCWSCLRPL
- RCWSSETP, RCWSSETPA, RCWSSETPAL, RCWSSETPL
- RCWSWPP, RCWSWPPA, RCWSWPPAL, RCWSWPPL
- SWPP, SWPPA, SWPPAL, SWPPL

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.2 Memory Copy and Memory Set CPY*

For a list of these instructions, see Memory Copy and Memory Set instructions.

K1.2.17.2.1 CONSTRAINED UNPREDICTABLE behavior

If `s == n || s == d || n == d || d == 31 || s == 31 || n == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.3 LD64B

For a description of this instruction and encoding, see LD64B.

K1.2.17.3.1 CONSTRAINED UNPREDICTABLE behavior

If a memory location does not support this instruction, then one of the following behaviors must occur:

- A stage 1 Data Abort, reported using the DFSC code of 110101, is generated.
- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

K1.2.17.4 LDAXP

For a description of this instruction and the encoding, see LDAXP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.4.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.5 LDNP

For a description of this instruction and the encoding, see LDNP.

K1.2.17.5.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.6 LDNP (SIMD&FP)

For a description of this instruction and the encoding, see LDNP (SIMD&FP).

K1.2.17.6.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.7 LDP and LDIAPP

For a description of these instructions and the encoding, see LDP and LDIAPP.

K1.2.17.7.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.8 LDP (SIMD&FP)

For a description of this instruction and the encoding, see LDP (SIMD&FP).

K1.2.17.8.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.9 LDPSW

For a description of this instruction and the encoding, see LDPSW.

K1.2.17.9.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.10 LDR (immediate)

For a description of this instruction and the encoding, see LDR (immediate).

K1.2.17.10.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.11 LDRB (immediate)

For a description of this instruction and the encoding, see LDRB (immediate).

K1.2.17.11.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.12 LDRH (immediate)

For a description of this instruction and the encoding, see LDRH (immediate).

K1.2.17.12.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.13 LDRSB (immediate)

For a description of this instruction and the encoding, see LDRSB (immediate).

K1.2.17.13.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.14 LDRSH (immediate)

For a description of this instruction and the encoding, see LDRSH (immediate).

K1.2.17.14.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.15 LDRSW (immediate)

For a description of this instruction and the encoding, see LDRSW (immediate).

K1.2.17.15.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.16 LDXP

For a description of this instruction and the encoding, see LDXP.

K1.2.17.16.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.17 Memory Copy and Memory Set SET*

For a list of these instructions, see Memory Copy and Memory Set instructions.

K1.2.17.17.1 CONSTRAINED UNPREDICTABLE behavior

If `s == n || s == d || n == d || d == 31 || n == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.18 ST64B

For a description of this instruction and encoding, see ST64B.

K1.2.17.18.1 CONSTRAINED UNPREDICTABLE behavior

If a memory location does not support this instruction, then one of the following behaviors must occur:

- A stage 1 Data Abort, reported using the DFSC code of 110101, is generated.
- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

K1.2.17.19 STP and STILP

For a description of these instructions and the encoding, see STP and STILP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.19.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.20 STLXP

For a description of this instruction and the encoding, see STLXP.

K1.2.17.20.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.21 STLXR

For a description of this instruction and the encoding, see STLXR.

K1.2.17.21.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.22 STLXRB

For a description of this instruction and the encoding, see STLXRB.

K1.2.17.22.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.23 STLXRH

For a description of this instruction and the encoding, see STLXRH.

K1.2.17.23.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.24 STR (immediate)

For a description of this instruction and the encoding, see STR (immediate).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.24.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.25 STRB (immediate)

For a description of this instruction and the encoding, see STRB (immediate).

K1.2.17.25.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.26 STRH (immediate)

For a description of this instruction and the encoding, see STRH (immediate).

K1.2.17.26.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.27 STXP

For a description of this instruction and the encoding, see STXP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.27.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.28 STXR

For a description of this instruction and the encoding, see STXR.

K1.2.17.28.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.29 STXRB

For a description of this instruction and the encoding, see STXRB.

K1.2.17.29.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.30 STXRH

For a description of this instruction and the encoding, see STXRH.

K1.2.17.30.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.18 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Note

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the AArch32 cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.19 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The unallocated value maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch32 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

K1.2.20 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K2
Recommendations for Reporting Memory Attributes on an Interconnect

This appendix describes the Arm recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

- Arm recommendations for reporting memory attributes on an interconnect.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K2.1 Arm recommendations for reporting memory attributes on an interconnect

The Arm architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and Cacheability attributes are presented on an interface to an interconnect fabric such as AMBA® AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that Arm strongly recommends, but these recommendations do not form part of the Arm architecture.

K2.1.1 Effect of microarchitectural choices on memory attributes

Implementations of the Arm architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-Through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-Through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-Through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, Arm strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early Write Acknowledgment attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

K2.1.1.1 Effect when memory accesses are forced to be Non-cacheable

Arm also strongly recommends that the effects of forcing accesses to Normal memory to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses for AArch64 and in Enabling and disabling the caching of memory accesses in AArch32 state for AArch32, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K3
ETE Recommended Configurations

This appendix describes the ETE features that Arm recommends are implemented. It contains the following section:

- Configurations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K3.1 Configurations

IYCKVP For optional features not described here, it is IMPLEMENTATION DEFINED whether the feature is implemented. For features which have an IMPLEMENTATION DEFINED size or number, and are not described here, the size or number of that feature is IMPLEMENTATION DEFINED.

Parameter	Description	Configuration
ATBTRIG	ATB Trigger Support	Yes, if ATB is implemented
NUMACPAIRS	Address Comparator pairs	4
NUMCIDC	Context Identifier Comparators	>= 1
NUMVMIDC	Virtual Context Identifier Comparators	>= 1, if EL2 is implemented
NUMCNTR	Number of Counters	2
NUMEVENT	Number of ETEEvents	4
NUMEXTINSEL	Number of External Input Selectors	4
NUMRSPAIR	Number of Resource selection pairs	>= 8
NUMSEQSTATE	Number of Sequencer states	4
NUMSSCC	Number of Single-shot Comparator Controls	>= 1
RETSTACK	Return stack	Yes
STALLCTL	PE stalling capability	Yes
TRACEIDSIZE	Trace ID size	7-bits, if ATB is implemented
CCITMIN	Cycle count minimum threshold	4
CCSIZE	Cycle counter size	>= 12
WFXMODE	`WFI`, `WFIT`, `WFE`, and `WFET` instruction classification	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K4
Stages of Execution

This appendix shows the relationship between the stages of execution. It contains the following sections:

- Stages of execution.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K4.1 Stages of execution

IKHCRN Figure K4-1 shows the stages of execution in a PE.

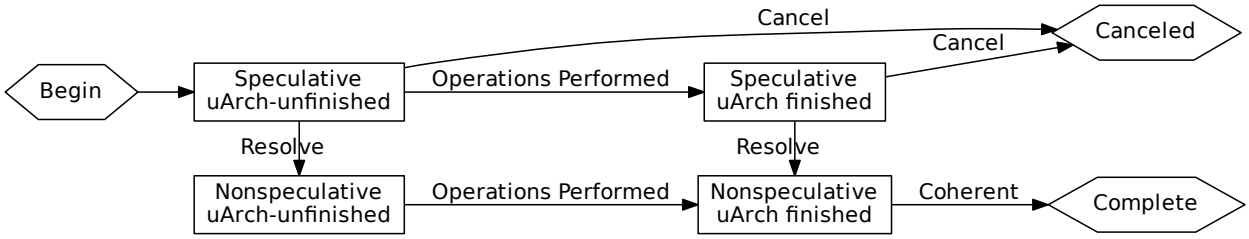

Figure K4-1 Stages of execution

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K5
Recommended External Debug Interface

This appendix describes the recommended external debug interface. It contains the following sections:

- About the recommended external debug interface.
- PMUEVENT bus.
- Recommended authentication interface.
- Management registers and CoreSight compliance.

Note

This recommended external debug interface specification is not part of the Arm architecture specification. Implementers and users of the architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

- As reference material for users of Arm products that implement this interface.
- As an example of how an external debug interface might be implemented.

The inclusion of this appendix is no indication of whether any Arm products might, or might not, implement this external debug interface. For details of the implemented external debug interface, you must always see the appropriate product documentation.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.1 About the recommended external debug interface

See the *Note* on the first page of this appendix for information about the architectural status of this recommended debug interface.

This specification provides a recommended external debug interface for Armv8 and later architectures to define a standard set of connections for validation environments. Generally, the connection between components, such as between the PE and Trace extension, is not described here, although the table does include the signals for the CTI connection. Table K5-1 shows the signals in the recommended interface.

Table K5-1 Recommended debug interface signals

Name	Direction	Description	Notes
DBGEN	In	External debug enable	-
SPIDEN	In	Secure privileged external debug enable	-
		Secure privileged self-hosted debug enable	Only in Secure AArch32 modes when enabled by MDCR_EL3.SPD32
RLPIDEN	In	Realm privileged external debug enable	Only if FEAT_RME is implemented.
RTPIDEN	In	Root privileged external debug enable	
NIDEN	In	External profiling and trace enable	If FEAT_Debugv8p4 is implemented, this signal is not implemented.
SPNIDEN	In	Secure external profiling and trace enable	If FEAT_Debugv8p4 is implemented, this signal is not implemented.
EDBGRQ	In	External halt request	IMPLEMENTATION DEFINED mechanism to halt the PE. See EDBGRQ and DBGACK.
DBGACK	Out	Debug Acknowledge	Indicate to the system that a PE is in Debug state. See EDBGRQ and DBGACK.
COMMIRQ	Out	DCC interrupt	Interface to an interrupt controller. See Interrupt-driven use of the DCC and the pseudocode for function `CheckForDCCInterrupts()`.
PMUIRQ	Out	Performance Monitor overflow	Interface to an interrupt controller. See Behavior on overflow.
COMMRX	Out	DTRRX is full	Provided for legacy connection to an interrupt controller only. See Interrupt-driven use of the DCC and the pseudocode for function `CheckForDCCInterrupts()`.
COMMTX	Out	DTRTX is empty	
PMUEVENT[n:0]	Out	Performance Monitors event bus	See PMUEVENT bus.
DBGNOPWRDWN	Out	Emulate low-power state request	Interface to a power controller. See Emulating low-power states.
DBGPWRUPREQ	Out	Core powerup request	Interface to a power controller. See Powerup request mechanism.
DBGRSTREQ	Out	Warm reset request	Interface to a power controller. See EDPRCR.CWRR.
DBGBUSCANCELREQ	Out	Allow asynchronous entry to Debug state	Extension to the bus interface. See EDRCR.CBRRQ.
DBGPWRDUP	In	Core powerup status	Interface to a power controller. See EDPRSR.PU.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Direction	Description	Notes
DBGROMADDR[n:12]	In	MDRAR_EL1.ROMADDR	*n* depends on the size of the physical address space. Arm recommends these signals are tied **LOW**.
DBGROMADDRV	In	MDRAR_EL1.Valid	Arm recommends these signals are tied **LOW**.
PRESETDBG	In	External Debug reset	-
CPUPORESET	In	Cold reset	-
CORERESET	In	Warm reset	-
PSELDBG	In	Debug APB interface[a]	For details, see *AMBA APB3*. Arm recommends a single port for all integrated debug components. `PADDRDBG31` distinguishes memory-mapped and Debug Access Port accesses: **0** Memory-mapped access **1** Debug Access Port access If FEAT_Debugv8p4 is implemented, **PPROTDBG[1]** distinguishes between Secure and Non-secure accesses. For details, see Non-secure, Secure, Realm, and Root views of the debug registers.
PENABLEDBG	In		
PWRITEDBG	In		
PRDATADBG[31:0]	Out		
PWDATADBG[31:0]	In		
PADDRDBG[n:2][b]	In		
PREADYDBG	Out		
PSLVERRDBG	Out		
PCLKDBG	In		
PCLKENDBG	In		
PPROTDBG[1]	In		
CTICHIN	In	CoreSight channel interface	For details, see the *Arm® CoreSight™ Architecture Specification*. The ACK signals are not required if the channel interface is synchronous.
CTICHOUTACK	In		
CTICHOUT	Out		
CTICHINACK	Out		
CTIIRQ	Out	CTI interrupt, see Description and allocation of CTI triggers	Implements a handshake for an edge-sensitive interrupt.
CTIIRQACK	In		
ATDATA[n×8-1:0]	Out	AMBA 4 ATB interface[c]	For details, see the *AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*. Only available if the OPTIONAL Trace extension is implemented.
ATBYTES[n-1:0]	Out		
ATID[6:0]	Out		
ATREADY	In		
ATVALID	Out	AMBA 4 ATB interface[c]	For details, see the *AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*. Only available if the OPTIONAL Trace extension is implemented.
AFREADY	Out		
AFVALID	Out		
SYNCREQ	In		
ATCLK	In		
ATCLKEN	In		
ATRESET	In		

a. This is the port where the PE completes debug APB transactions. Arm recommends a single port for all integrated debug components.

b. The value of *n* depends on the size of the address space occupied by the Debug port.

c. This is the port where the PE outputs trace.

Figure K5-1 shows the recommended debug interface.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

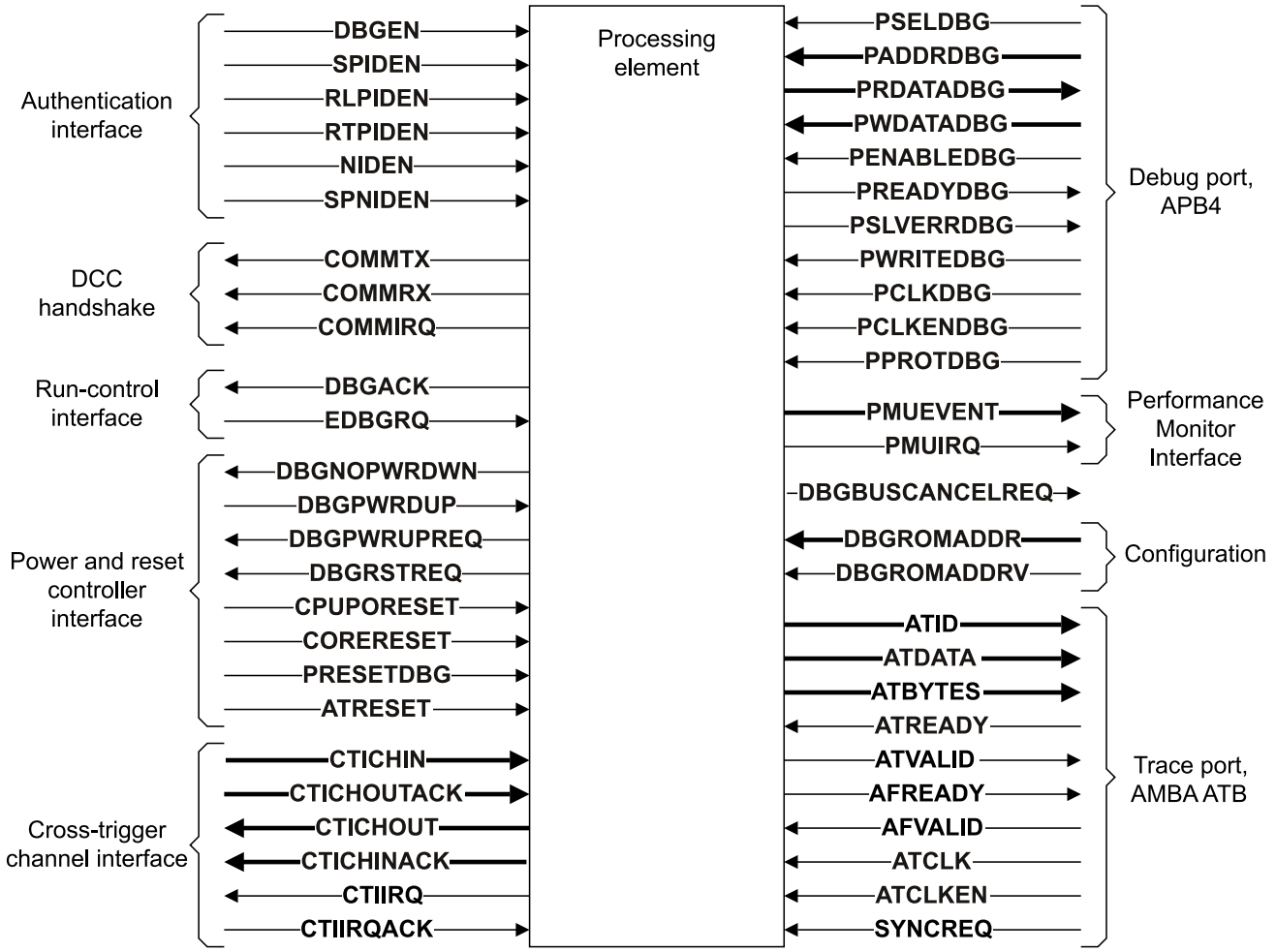

Figure K5-1 Recommended external debug interface, including the APB4 Completer port

K5.1.1 EDBGRQ and DBGACK

EDBGRQ is an IMPLEMENTATION DEFINED means of generating the External Debug Request debug event described in External Debug Request debug event.

The PE asserts **DBGACK** when the PE is in Debug state. The PE might also include variants of this signal:

DBGTRIGGER

Asserted by the PE when it commits to entering Debug state.

DBGCPUDONE

Asserted by the PE when it has completed all Non-debug state memory accesses and Debug state entry is complete. **DBGCPUDONE** indicates that memory accesses issued by the PE result from operations originating from debugger commands.

In previous architecture versions, these signals provide an interface between the PE and cross-trigger logic. In Armv8, the architectural Cross-Trigger Interface provides this functionality for external debuggers.

K5.1.2 Non-secure, Secure, Realm, and Root views of the debug registers

If FEAT_Debugv8p4 is implemented, the external debug interface has views of debug registers for each physical address space, Non-secure, Secure and, if FEAT_RME is implemented, Realm and Root. The DAP must ensure that accesses are

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

made only when permitted. The Arm Debug Interface describes a standard APB-AP programmers' model for APB4, with the **PPROTDBG[1]** and, if FEAT_RME is implemented, **PNSEDBG** signals used to determine the physical address space that is accessed. This model is recommended for new designs that do not implement FEAT_RME, and is required for new designs that implement FEAT_RME.

If FEAT_RME is not implemented, FEAT_Debugv8p4 is implemented, and an APB-AP implements an APB3 Requester port, which does not support Secure and Non-secure views, Arm recommends that the following is implemented:

- If **SPIDEN** is HIGH and **DBGEN** is HIGH, all external debug accesses are treated as Secure.
- If either **SPIDEN** is LOW or **DBGEN** is LOW, all external debug accesses are treated as Non-secure.

If the PE APB Completer port is APB4, this might be implemented by, for example, fixing **PPROTDBG[1]** to the inverse of (**SPIDEN** & **DBGEN**) when bridging from APB3 to APB4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.2 PMUEVENT bus

The PMUEVENT bus exports Performance Monitor events from the PE to an on-chip agent. Arm recommends that it has the following characteristics:

- The bus is synchronous.
- The width of the bus is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED which events are exported on the bus.
- Each exported event occupies a contiguous sub-field of the bus. Arm recommends that the sub-fields of the bus are occupied in the same order as the event numbers.
- If the event can only occur once per cycle, it occupies a single bit. If the event can occur more than once per cycle, it is IMPLEMENTATION DEFINED how the event is encoded. The encoding depends on constraints such as the designated use of the event bus and the number of pins available. For example, the event can be encoded:
  - As a count, using a plain binary number. This is the most useful encoding when exporting to an external counter. It is not a useful encoding for exporting to a Trace extension external input.
  - As a count, using thermometer encoding. This is the most useful encoding when exporting to a Trace extension.
  - Using a single bit encoding to indicate whether the event count is zero or nonzero. This is useful for exporting to an activity monitor where the number of pins is constrained.

If an ETMv4 Trace extension is implemented, the PMUEVENT bus is normally connected to the Trace extension using the external inputs. TRCEXTINSELR multiplexes a wide PMUEVENT bus to a narrow set of inputs. An external PMUEVENT bus might also be provided. For more information, contact Arm.

When FEAT_ETE is implemented, PMU events are connected to the trace unit, however the PMUEVENT bus is not used to connect the PMU events to the trace unit. See External inputs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.3 Recommended authentication interface

A PE implements the debug authentication interface defined in Required debug authentication.

The details of the debug authentication interface are IMPLEMENTATION DEFINED. Arm recommends the use of the CoreSight authentication interface, which includes the following signals for external debug authentication:

- **DBGEN**.
- **SPIDEN**.

If FEAT_Debugv8p4 is not implemented, then Arm also recommends using the following signals:

- **NIDEN**.
- **SPNIDEN**.

If FEAT_RME is implemented, then Arm recommends the following additional signals for external debug authentication:

- **RLPIDEN**.
- **RTPIDEN**.

Arm recommends an interface in which **DBGEN** and **SPIDEN** are also used for self-hosted Secure debug authentication if either:

- EL3 is using AArch32 and SDCR.SPD == `0b00`.
- Secure EL1 is using AArch32 and MDCR_EL3.SPD32 == `0b00`.

If FEAT_Secure is not implemented, then all of the following apply:

- **SPIDEN** and **SPNIDEN** are not implemented.
- If FEAT_RME is implemented, then the PE behaves as if **SPIDEN** has the same value as **RTPIDEN**.
- If FEAT_RME is not implemented, then the PE behaves as if **SPIDEN** is tied LOW.
- The PE behaves as if **SPNIDEN** is tied LOW.

If FEAT_RME is not implemented, then **RLPIDEN** and **RTPIDEN** are not implemented, and the PE behaves as if these signals were tied LOW.

If EL3 is not implemented and FEAT_Secure is implemented, then it is often the case that **SPIDEN** is connected to **DBGEN** and **SPNIDEN** is connected to **NIDEN**, but this is not required.

The recommended interface is defined as if all four signals are implemented.

In order to include External debug state in Realm attestation when FEAT_RME is implemented, the authentication signals that control External debug in the Realm and Root Security states must be sampled before execution starts in the corresponding Security state and not change value until a system reset event:

- For the **RTPIDEN** authentication signal, Arm expects that this behavior will be guaranteed by system construction.
- For the **RLPIDEN** authentication signal, this behavior can be guaranteed by system construction or by Root firmware.

How the authentication signals are driven is IMPLEMENTATION DEFINED. For example, the signals might be hard-wired, connected to fuses, or to an authentication module. The architecture permits PEs within a cluster to have independent authentication interfaces, but this is not required. Any Trace extension has the same authentication interface as the PE it is connected to.

If FEAT_Debugv8p4 and CoreSight ETR are both implemented, then the ETR has an independent **DBGEN** signal that must be tied HIGH to enable self-hosted use of trace.

Table K5-2 shows the debug authentication pseudocode functions and the recommended implementations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K5-2 Recommended implementation of debug enable pseudocode functions

Pseudocode function	Description	Implementation
`ExternalNoninvasiveDebugEnabled()`[a]	Non-secure non-invasive debug enabled	(**DBGEN** OR **NIDEN**[c])
`ExternalInvasiveDebugEnabled()`	Non-secure invasive debug enabled	**DBGEN**
`AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled()`	Secure invasive self-hosted debug enabled in AArch32 state (legacy)	(**DBGEN** AND **SPIDEN**)
`ExternalSecureNoninvasiveDebugEnabled()`[b]	Secure non-invasive debug enabled	(**DBGEN** OR **NIDEN**[c]) AND (**SPIDEN**[c] OR **SPNIDEN**[b,c])
`ExternalSecureInvasiveDebugEnabled()`[c]	Secure invasive debug enabled	(**DBGEN** AND **SPIDEN**[c])
`ExternalRealmInvasiveDebugEnabled()`[d]	Realm invasive debug enabled	(**DBGEN** AND **RLPIDEN**[d])
`ExternalRootInvasiveDebugEnabled()`[d]	Root invasive debug enabled	(**DBGEN** AND **RLPIDEN**[d] AND **SPIDEN**[c] AND **RTPIDEN**[d])

a. If FEAT_Debugv8p4 is implemented, then the **NIDEN** signal is not implemented and `ExternalNoninvasiveDebugEnabled()` == TRUE.

b. If FEAT_Debugv8p4 is implemented, then the **SPNIDEN** signal is not implemented and `ExternalSecureNoninvasiveDebugEnabled()` == `ExternalSecureInvasiveDebugEnabled()`.

c. If FEAT_Secure is not implemented, then the **SPIDEN** and **SPNIDEN** signals are not implemented.

d. If FEAT_RME is not implemented, then the **RLPIDEN** and **RTPIDEN** signals are not implemented.

Table K5-3 shows the Security states that can be externally debugged with different values for the debug authentication signals, based on the recommended mapping:

Table K5-3 Permitted external debug of Security states

DBGEN	SPIDEN[a]	RLPIDEN[b]	RTPIDEN[b]	Non-secure[c]	Secure[d]	Realm[e]	Root[e]
LOW	x	x	x	No	No	No	No
HIGH	LOW	LOW	x	Yes	No	No	No
HIGH	LOW	HIGH	x	Yes	No	Yes	No
HIGH	HIGH	LOW	x	Yes	Yes	No	No
HIGH	HIGH	HIGH	LOW	Yes	Yes	Yes	No
HIGH	HIGH	HIGH	HIGH	Yes	Yes	Yes	Yes

a. If FEAT_Secure is not implemented and FEAT_RME is implemented, then the PE behaves as if **SPIDEN** has the same value as **RTPIDEN**. If FEAT_Secure is not implemented and FEAT_RME is not implemented, then the PE behaves as if **SPIDEN** is tied LOW.

b. If FEAT_RME is not implemented, then the PE behaves as if **RLPIDEN** and **RTPIDEN** are tied LOW.

c. When EL3 is implemented or FEAT_Secure is not implemented.

d. When FEAT_Secure is implemented.

e. When FEAT_RME is implemented.

The `Debug_authentication` pseudocode in shared/debug defines the authentication signals **DBGEN**, **SPIDEN**, **RLPIDEN**, **RTPIDEN**, **NIDEN**, and **SPNIDEN**.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.4 Management registers and CoreSight compliance

The CoreSight architecture requires the implementation of a set of management registers that occupy the memory map from `0xF00` upwards in each of the debug components.

CoreSight compliance and complete implementation of the management registers is OPTIONAL, but Arm recommends that the registers are implemented.

The CoreSight architecture specification recommends that any integration test registers are implemented starting from `0xEFC` downwards. Each of the debug components has an IMPLEMENTATION DEFINED region from `0xE80` to `0xEFC` for this purpose.

K5.4.1 CoreSight interface register map

Table K5-4 shows the external management register maps for the following registers:

ED These are the external debug register.

CTI These are the Cross-trigger interface registers.

PMU These are the Performance Monitors registers.

Table K5-4 CoreSight interface register map

Offset	Mnemonic			Name
	ED	CTI	PMU	
0xF00	EDITCTRL	CTIITCTRL	PMITCTRL	Integration Model Control registers
0xF04-0xF9C	-	-	-	Reserved, RES0
0xFA0	DBGCLAIMSET_EL1[a]	CTICLAIMSET[b]	-	CLAIM Tag Set registers
0xFA4	DBGCLAIMCLR_EL1[a]	CTICLAIMCLR[b]	-	CLAIM Tag Clear registers
0xFA8	EDDEVAFF0[a]	CTIDEVAFF0[c]	PMDEVAFF0	Device Affinity registers
0xFAC	EDDEVAFF1[a]	CTIDEVAFF1[c]	PMDEVAFF1	
0xFB0	EDLAR[d]	CTILAR[d]	PMLAR[d]	Lock Access register
0xFB4	EDLSR[d]	CTILSR[d]	PMLSR[d]	Lock Status register
0XFB8	DBGAUTHSTATUS_EL1[a]	CTIAUTHSTATUS	PMAUTHSTATUS	Authentication Status register
0xFBC	EDDEVARCH	CTIDEVARCH	PMDEVARCH	Device Architecture register
0xFC0	EDDEVID2[a]	CTIDEVID2[a]	-	Device ID register
0xFC4	EDDEVID1[a]	CTIDEVID1[a]	-	
0xFC8	EDDEVID[a]	CTIDEVID[a]	PMDEVID[a,e]	
0xFCC	EDDEVTYPE	CTIDEVTYPE	PMDEVTYPE	Device Type register
0xFD0	EDPIDR4	CTIPIDR4	PMPIDR4	Peripheral ID registers
0xFD4-0xFDC	-	-	-	Reserved, RES0
0xFE0	EDPIDR0	CTIPIDR0	PMPIDR0	Peripheral ID registers
0xFE4	EDPIDR1	CTIPIDR1	PMPIDR1	
0xFE8	EDPIDR2	CTIPIDR2	PMPIDR2	
0xFEC	EDPIDR3	CTIPIDR3	PMPIDR3	
0xFF0	EDCIDR0	CTICIDR0	PMCIDR0	Component ID registers
0xFF4	EDCIDR1	CTICIDR1	PMCIDR1	

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Mnemonic			Name
	ED	CTI	PMU	
0xFF8	EDCIDR2	CTICIDR2	PMCIDR2	
0xFFC	EDCIDR3	CTICIDR3	PMCIDR3	

a. This register must always be implemented, regardless of whether the component is CoreSight compliant.

b. If implemented, the number of CLAIM bits is IMPLEMENTATION DEFINED and can be discovered by reading CLAIMSET.

c. If the CTI implements CTIv1, this register is not implemented. See the register description for details.

d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

e. PMDEVID is implemented only from Armv8.2 or if FEAT_PCSRv8p2 is implemented, otherwise its offset is RES0.

K5.4.2 Management register access permissions

Access to the OPTIONAL Integration Control register (ITCTRL) is IMPLEMENTATION DEFINED.

Table K5-5, Table K5-6, Table K5-7, Table K5-8, and Table K5-9 show the response to accesses by the external debug interface to the CoreSight management registers.

Note

Access to the CoreSight management registers is not affected by the values of EDAD and EPMAD.

Table K5-5, Table K5-6, Table K5-7, Table K5-8, and Table K5-9 include reserved management registers, because the CoreSight architecture requires that these registers are always RES0. The descriptions in Reserved and unallocated registers do not apply to reserved management registers if the implementation is CoreSight compliant.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses. See Register access permissions for memory-mapped accesses.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

The terms in Table K5-5, Table K5-6, Table K5-7, Table K5-8, and Table K5-9 are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If FEAT_DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table K5-5 and Table K5-8.

If FEAT_DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table K5-6 and Table K5-9.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

The conditions are:

Off EDPRSR.PU == 0. The Core power domain is completely off, or in low-power state. In these cases, the Core power domain registers cannot be accessed.

Note

When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DLK If the OS Double Lock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

OSLK OSLSR.OSLK == 1. The OS Lock is locked.

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

SLK This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See Register access permissions for memory-mapped accesses. If FEAT_DoPD is implemented, the Software Lock is not locked, or not implemented, this column is ignored.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO This means that the register or field is read-only.

RW This means that the register or field is read/write. Individual fields within the register might be RO. See the relevant register description for details.

RC This means that the bit clears to 0 after a read.

(SE) This means that accesses to this register have indirect write side effects. A side effect occurs when a direct read or a direct write of a register creates an indirect write to the same register or to another register.

WO This means that the register or field is write-only.

WI This means that the register or field ignores writes.

IMP DEF This means that the access permissions are IMPLEMENTATION DEFINED.

Table K5-5 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is implemented

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
`0xF00`	EDITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF
`0xF04-0xF8C`	Reserved		-	-	-	RES0
`0xFA0`	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW (SE)
`0xFA4`	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW (SE)
`0xFA8`	EDDEVAFF0	Core	Error	-	-	RO
`0xFAC`	EDDEVAFF1	Core	Error	-	-	RO
`0xFB0`	EDLAR	Core	Error	-	-	WO (SE)
`0xFB4`	EDLSR	Core	Error	-	-	RO
`0xFB8`	DBGAUTHSTATUS_EL1	Core	Error	-	-	RO
`0xFBC`	EDDEVARCH	Core	Error	-	-	RO
`0xFC0`	EDDEVID2	Core	Error	-	-	RO
`0xFC4`	EDDEVID1	Core	Error	-	-	RO
`0xFC8`	EDDEVID	Core	Error	-	-	RO

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xFCC	EDDEVTYPE	Core	Error	-	-	RO
0xFD0	EDPIDR4	Core	Error	-	-	RO
0xFD4-0xFDC	Reserved		-	-	-	RES0
0xFE0-0xFEC	EDPIDR0	Core	Error	-	-	RO
0xFE4	EDPIDR1	Core	Error	-	-	RO
0xFE8	EDPIDR2	Core	Error	-	-	RO
0xFEC	EDPIDR3	Core	Error	-	-	RO
0xFF0	EDCIDR0	Core	Error	-	-	RO
0xFF4	EDCIDR1	Core	Error	-	-	RO
0xFF8	EDCIDR2	Core	Error	-	-	RO
0xFFC	EDCIDR3	Core	Error	-	-	RO

Table K5-6 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	EDITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xF8C	Reserved	Debug	-	-	-	RES0	-
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW (SE)	RO
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW (SE)	RO
0xFA8	EDDEVAFF0	Debug	-	-	-	RO	-
0xFAC	EDDEVAFF1	Debug	-	-	-	RO	-
0xFB0	EDLAR	Debug	-	-	-	WO (SE)	-
0xFB4	EDLSR	Debug	-	-	-	RO	-
0xFB8	DBGAUTHSTATUS_EL1	Debug	-	-	-	RO	-
0xFBC	EDDEVARCH	Debug	-	-	-	RO	-
0xFC0	EDDEVID2	Debug	-	-	-	RO	-
0xFC4	EDDEVID1	Debug	-	-	-	RO	-
0xFC8	EDDEVID	Debug	-	-	-	RO	-
0xFCC	EDDEVTYPE	Debug	-	-	-	RO	-
0xFD0	EDPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0-0xFEC	EDPIDR0	Debug	-	-	-	RO	-
0xFE4	EDPIDR1	Debug	-	-	-	RO	-
0xFE8	EDPIDR2	Debug	-	-	-	RO	-
0xFEC	EDPIDR3	Debug	-	-	-	RO	-
0xFF0	EDCIDR0	Debug	-	-	-	RO	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFF4	EDCIDR1	Debug	-	-	-	RO	-
0xFF8	EDCIDR2	Debug	-	-	-	RO	-
0xFFC	EDCIDR3	Debug	-	-	-	RO	-

Table K5-7 External debug interface access permissions, CoreSight registers (CTI)

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	CTIITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xF8C	Reserved	Debug	-	-	-	RES0	-
0xFA0	CTICLAIMSET	Debug	-	-	-	RW (SE)	RO
0xFA4	CTICLAIMCLR	Debug	-	-	-	RW (SE)	RO
0xFA8	CTIDEVAFF0	Debug	-	-	-	RO	-
0xFAC	CTIDEVAFF1	Debug	-	-	-	RO	-
0xFB0	CTILAR	Debug	-	-	-	WO (SE)	-
0xFB4	CTILSR	Debug	-	-	-	RO	-
0xFB8	CTIAUTHSTATUS	Debug	-	-	-	RO	-
0xFBC	CTIDEVARCH	Debug	-	-	-	RO	-
0xFC0	CTIDEVID2	Debug	-	-	-	RO	-
0xFC4	CTIDEVID1	Debug	-	-	-	RO	-
0xFC8	CTIDEVID	Debug	-	-	-	RO	-
0xFCC	CTIDEVTYPE	Debug	-	-	-	RO	-
0xFD0	CTIPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0	CTIPIDR0	Debug	-	-	-	RO	-
0xFE4	CTIPIDR1	Debug	-	-	-	RO	-
0xFE8	CTIPIDR2	Debug	-	-	-	RO	-
0xFEC	CTIPIDR3	Debug	-	-	-	RO	-
0xFF0	CTICIDR0	Debug	-	-	-	RO	-
0xFF4	CTICIDR1	Debug	-	-	-	RO	-
0xFF8	CTICIDR2	Debug	-	-	-	RO	-
0xFFC	CTICIDR3	Debug	-	-	-	RO	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K5-8 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is implemented

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xF00	PMITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF
0xF04-0xFA4	Reserved		-	-	-	RES0
0xFA8	PMDEVAFF0	Core	Error	-	-	RO
0xFAC	PMDEVAFF1	Core	Error	-	-	RO
0xFB0	PMLAR	Core	Error	-	-	WO (SE)
0xFB4	PMLSR	Core	Error	-	-	RO
0xFB8	PMAUTHSTATUS	Core	Error	-	-	RO
0xFBC	PMDEVARCH	Core	Error	-	-	RO
0xFC0-0xFC4	Reserved		-	-	-	RES0
0xFC8	PMDEVID[a]	Core	Error	-	-	RO
0xFCC	PMDEVTYPE	Core	Error	-	-	RO
0xFD0	PMPIDR4	Core	Error	-	-	RO
0xFD4-0xFDC	Reserved		-	-	-	RES0
0xFE0	PMPIDR0	Core	Error	-	-	RO
0xFE4	PMPIDR1	Core	Error	-	-	RO
0xFE8	PMPIDR2	Core	Error	-	-	RO
0xFEC	PMPIDR3	Core	Error	-	-	RO
0xFF0	PMCIDR0	Core	Error	-	-	RO
0xFF4	PMCIDR1	Core	Error	-	-	RO
0xFF8	PMCIDR2	Core	Error	-	-	RO
0xFFC	PMCIDR3	Core	Error	-	-	RO

a. Implemented from Armv8.2, or if FEAT_PCSRv8p2 is implemented. Otherwise this location is RES0.

Table K5-9 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	PMITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xFA4	Reserved	Debug	-	-	-	RES0	-
0xFA8	PMDEVAFF0	Debug	-	-	-	RO	-
0xFAC	PMDEVAFF1	Debug	-	-	-	RO	-
0xFB0	PMLAR	Debug	-	-	-	WO (SE)	-
0xFB4	PMLSR	Debug	-	-	-	RO	-
0xFB8	PMAUTHSTATUS	Debug	-	-	-	RO	-
0xFBC	PMDEVARCH	Debug	-	-	-	RO	-
0xFC0-0xFC4	Reserved	Debug	-	-	-	RES0	-

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFC8	PMDEVID[a]	Debug	-	-	-	RO	-
0xFCC	PMDEVTYPE	Debug	-	-	-	RO	-
0xFD0	PMPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0	PMPIDR0	Debug	-	-	-	RO	-
0xFE4	PMPIDR1	Debug	-	-	-	RO	-
0xFE8	PMPIDR2	Debug	-	-	-	RO	-
0xFEC	PMPIDR3	Debug	-	-	-	RO	-
0xFF0	PMCIDR0	Debug	-	-	-	RO	-
0xFF4	PMCIDR1	Debug	-	-	-	RO	-
0xFF8	PMCIDR2	Debug	-	-	-	RO	-
0xFFC	PMCIDR3	Debug	-	-	-	RO	-

a. Implemented from Armv8.2, or if FEAT_PCSRv8p2 is implemented. Otherwise this location is RES0.

K5.4.3 Management register resets

Table K5-10 shows the management register resets. This table does not include:

- Read-only identification registers that have a fixed value from reset. These registers include those with the DEVAFFn, DEVARCH, DEVID{n}, DEVTYPE, PIDRn, and CIDRn mnemonics.
- Registers that have the AUTHSTATUS mnemonic. This is a read-only status register that reflects the status outside of the reset domain of the register.
- Registers that have the LAR mnemonic. These are write-only registers that only have an effect on writes.

All other fields in the management registers are reset to an IMPLEMENTATION DEFINED value which can be UNKNOWN. The registers are in the reset domain specified in the table.

Table K5-10 shows a summary of the management register resets.

Table K5-10 Management register resets

Register	Reset domain	Field	Value	Description
CTIITCTRL EDITCTRL PMITCTRL	IMPLEMENTATION DEFINED	IME	0	Integration mode enable
DBGCLAIMCLR_EL1	Cold reset	CLAIM	0x00	CLAIM tags
CTICLAIMCLR	External debug	CLAIM	0x0000_0000	
CTILSR[a] EDLSR[a] PMLSR[a]	If FEAT_DoPD is implemented, reset by Cold reset, otherwise External debug.	SLK	1	Software Lock

a. Only if the OPTIONAL Software Lock is implemented

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.4.4 About the Peripheral identification scheme

The Peripheral Identification scheme provides the standard information required by all components that conform to the *Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2*, that implements the CoreSight identification scheme. They identify a peripheral in a particular namespace. For more information, see the *Arm® CoreSight™ Architecture Specification*.

Table K5-11 lists the Peripheral ID Registers that make up the Peripheral Identification scheme for each component.

Table K5-11 Peripheral Identification Registers

		Reference				
Register offset	**Description**	**Debug**	**CTI**	**PMU**	**Trace**	**Trace Buffer**
`0xFD0`	Peripheral ID4, PIDR4	EDPIDR4	CTIPIDR4	PMPIDR4	TRCPIDR4	TRBPIDR4
`0xFD4`	Reserved for Peripheral ID5	-	-	-	-	-
`0xFD8`	Reserved for Peripheral ID6	-	-	-	-	-
`0xFDC`	Reserved for Peripheral ID7	-	-	-	-	-
`0xFE0`	Peripheral ID0, PIDR0	EDPIDR0	CTIPIDR0	PMPIDR0	TRCPIDR0	TRBPIDR0
`0xFE4`	Peripheral ID1, PIDR1	EDPIDR1	CTIPIDR1	PMPIDR1	TRCPIDR1	TRBPIDR1
`0xFE8`	Peripheral ID2, PIDR2	EDPIDR2	CTIPIDR2	PMPIDR2	TRCPIDR2	TRBPIDR2
`0xFEC`	Peripheral ID3, PIDR3	EDPIDR3	CTIPIDR3	PMPIDR3	TRCPIDR3	TRBPIDR3

Table K5-12 shows the fields in the Peripheral Identification scheme.

Table K5-12 Fields in the Peripheral Identification Registers

Name	Description	Registers
DES_x	Identifies the designer of the component using the JEDEC JEP106 scheme. This value consists of: • A continuation code, also described as the bank number. • An identification code, identifying the designer in the bank. For implementations designed by Arm, the continuation code is `0x4`, indicating bank 5, and the identification code is `0x3B`.	PIDR1, PIDR2, PIDR4
PART_x	Part number for the component. Each organization designing to the Arm Debug architecture specification keeps its own part number list.	PIDR0, PIDR1
REVISION	Revision number for the component.	PIDR2
REVAND	Additional revision information for the component.	PIDR3
CMOD	Indicates an endorsed modification to the component. If the system designer cannot modify the implementation supplied by the implementation designer, then this field is RES0.	PIDR3
SIZE	Log2 of the number of 4KB blocks occupied by the component . Use of this field is deprecated, and Arm recommends this field always has the value zero.	PIDR4

A component is identified uniquely by the combination of the following fields:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- JEP106 continuation code.
- JEP106 identification code.
- Part number.
- REVISION.
- REVAND.

For components with a *Component class* of `0x9`, Debug component, indicated by the Component Identification Registers, multiple components can have the same Part number, provided each component has a different CoreSight *Device type* or *Device architecture*. However, Arm strongly recommends that each device has a unique Part number. For more information:

- About the Component Identification Registers, see About the Component Identification scheme.
- About the CoreSight Device type, see EDDEVTYPE, CTIDEVTYPE, PMDEVTYPE, TRCDEVTYPE, or TRBDEVTYPE.
- About the CoreSight Device architecture, see EDDEVARCH, CTIDEVARCH, PMDEVARCH, TRCDEVARCH, or TRBDEVARCH.
- About CoreSight components and their identification, see the *Arm® Debug Interface Architecture Specification*.

K5.4.4.1 Allocating revisions and part numbers

Within the Peripheral Identification registers, the allocation of major and minor revisions, part numbers, and customer-modified fields is IMPLEMENTATION DEFINED.

The size of the part number is specific to the designer of the component, but is typically 12-bits wide or 16-bits wide. How component revision information is presented is specific to the designer of the component, and might also be specific to each individual component with a different part number. Arm recommends one of the following styles is used:

- For a component with a 12-bit part number:
  - PIDR1.PART_1 contains part number bits [11:8].
  - PIDR0.PART_0 contains part number bits [7:0].
  - Where the component uses single 4-bit revision number:
    - PIDR2.REVISION indicates the 4-bit revision number.
    - PIDR3.REVAND indicates component modifications.
  - Where the component uses separate major and minor revision numbers:
    - PIDR2.REVISION indicates the 4-bit major revision number.
    - PIDR3.REVAND indicates the 4-bit minor revision number.
- For a component with a 16-bit part number:
  - PIDR1.PART_1 contains part number bits [15:12].
  - PIDR0.PART_0 contains part number bits [11:4].
  - PIDR2.REVISION contains part number bits [3:0].
  - PIDR3.REVAND indicates the 4-bit revision number.

Arm recommends that the revision information is updated for each update to the component, regardless of whether this is a major or minor update. The revision information increases monotonically with each revision of the component. When the REVAND field is updated on a new revision, this field is incremented monotonically. When the REVISION field is updated on a new revision, REVAND is typically set to zero on the first release of this new revision.

K5.4.5 About the Component Identification scheme

The Component Identification Registers identify the processor as an Arm Debug Interface v5 component. For more information, see the *Arm® Debug Interface Architecture Specification* and the *Arm® CoreSight™ Architecture Specification*.

The Component Identification Registers occupy the last four words of the 4KB block of debug registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K5-13 Component Identification Registers

Register offset	Description	Debug	CTI	PMU	Trace	Trace Buffer
`0xFF0`	Component ID0, CIDR0	EDCIDR0	CTICIDR0	PMCIDR0	TRCCIDR0	TRBCIDR0
`0xFF4`	Component ID1, CIDR1	EDCIDR1	CTICIDR1	PMCIDR1	TRCCIDR1	TRBCIDR1
`0xFF8`	Component ID2, CIDR2	EDCIDR2	CTICIDR2	PMCIDR2	TRCCIDR2	TRBCIDR2
`0xFFC`	Component ID3, CIDR3	EDCIDR3	CTICIDR3	PMCIDR3	TRCCIDR3	TRBCIDR3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K6
Additional Information for Implementations of the Generic Timer

This appendix gives additional information about implementations of the Generic Timer. It contains the following sections:

- Providing a complete set of features in a system level implementation.
- Gray count scheme for timer distribution scheme.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K6.1 Providing a complete set of features in a system level implementation

As an example system design, using memory-mapped Generic Timer components as described in System Level Implementation of the Generic Timer, the feature set of a System registers counter and timer, in an implementation that includes Non-secure EL2 and EL3, can be implemented using the following set of timer frames:

- A CNTCTLBase control frame.
- The following CNTBase *N* timer frames:

Frame 0 Accessible by Non-secure accesses, with second view and virtual capability. This provides the Non-secure EL1&0 timers.

Frame 1 Accessible by Non-secure accesses, with no second view and no virtual capability. This provides the Non-secure EL2 timers.

Frame 2 Accessible only by Secure accesses, with a second view but no virtual capability. This provides the Secure PL1&0 timers, meaning:

- Compared to a PE where EL3 is using AArch64, it provides the Secure EL1&0 timer.
- Compared to a PE where EL3 is using AArch32, it provides the only Secure state timer.

Frame 3 Accessible only by Secure accesses, with no second view and no virtual capability. This provides the EL3 timers.

Note

This frame is not required for a memory-mapped timer that provides only the feature set of a PE for which EL3 is using AArch32.

In this implementation, the full set of implemented frames, and accessibility as memory pages in the different translation regimes, is as follows:

CNTCTLBase

The control frame. This frame is accessible in both the Secure and Non-secure memory maps, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase0 The first view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible only at EL1.

CNTEL0Base0

The second view of CNTBase0, meaning it is the EL0 view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, the architecture permits this frame to be accessible at EL1, or at EL1 and EL0, but does not require either of these options.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.

CNTBase1 The first and only view of the Non-secure EL2 timers. This frame is accessible only in Non-secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the EL3 translation regime, this frame is accessible.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase2 The first view of the Secure EL1&0, or PL1&0 timers.

Note

In AArch64 state, these timers are always called the Secure EL1&0 timers. In AArch32 state they are usually called the Secure PL1&0 timers because, in AArch32 Secure state, whether some of the PE modes map to EL1 or to EL3 depends on whether EL3 is using AArch64 or AArch32. See Security state, Exception levels, and AArch32 execution privilege.

This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTEL0Base2

The second view of CNTBase2, meaning it is the EL0 view of the Secure EL1&0, or PL1&0, timers.

Note

See the Note in the description of the CNTBase2 frame for more information about the naming of these timers.

This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.
  - In the EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible at PL1 (EL3) and EL0.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTBase3 The first and only view of the EL3 timers. This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is not accessible.
  - In the EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, this frame is not accessible.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K6.2 Gray count scheme for timer distribution scheme

The distribution of the Counter value using a Gray code provides a relatively simple mechanism to avoid any danger of the count being sampled with an intermediate value even if the clocking is asynchronous. It has a further advantage that the distribution is relatively low power, since only one bit changes on the main distribution wires for each clock tick.

A suitable Gray-coding scheme can be achieved with the following logic:

Conversion from count to gray

```
let gray : bits(N) = count XOR ('0'::count[N-1]);
```

Conversion from gray to count

```
count[N] = gray[N];
for i = N-1 downto 0 do
    count[i] = gray[i] XOR count[i+1];
end;
```

This is for an N+1 bit counter, where `count` is a conventional binary count value, and `gray` is the corresponding Gray count value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K7
Legacy Instruction Syntax for AArch32 Instruction Sets

This appendix describes the legacy instruction syntax in the Arm instruction sets, and their *Unified Assembler Language* (UAL) equivalents. It contains the following section:

- Legacy Instruction Syntax.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K7.1 Legacy Instruction Syntax

Early versions of the Arm Architecture defined an assembly language for A32 (ARM) instructions, and a separate assembly language for T32 (Thumb) instructions. UAL is based on the A32 assembly language, with some changes to the instruction syntax. The appendix describes those changes. The pre-UAL mnemonics are compatible with UAL, and might be supported by an assembler.

The original T32 assembly language is not compatible with UAL, and is not described in the manual.

K7.1.1 Pre-UAL instruction syntax for the A32 base instructions

Table K7-1 lists the syntax for the A32 base instructions that have changed after UAL was introduced.

Table K7-1 Pre-UAL instruction syntax for the A32 base instructions

Pre-UAL syntax	UAL equivalent	See
ADC<c>S	ADCS<c>	ADC, ADCS (immediate), ADC, ADCS (register), ADC, ADCS (register-shifted register)
ADD<c>S	ADDS<c>	ADD, ADDS (immediate), ADD, ADDS (register), ADD, ADDS (register-shifted register), ADD, ADDS (SP plus immediate), ADD, ADDS (SP plus register)
AND<c>S	ANDS<c>	AND, ANDS (immediate), AND, ANDS (register), AND, ANDS (register-shifted register)
BIC<c>S	BICS<c>	BIC, BICS (immediate), BIC, BICS (register), BIC, BICS (register-shifted register)
EOR<c>S	EORS<c>	EOR, EORS (immediate), EOR, EORS (register), EOR, EORS (register-shifted register)
LDC<c>L	LDCL<c>	LDC (immediate), LDC (literal)
LDM<c>IA, LDM<c>FD	LDM<c>	LDM, LDMIA, LDMFD
LDM<c>DA, LDM<c>FA	LDMDA<c>	LDMDA, LDMFA
LDM<c>DB, LDM<c>EA	LDMDB<c>	LDMDB, LDMEA
LDM<c>IB, LDM<c>ED	LDMIB<c>	LDMIB, LDMED
LDR<c>B	LDRB<c>	LDRB (immediate), LDRB (literal), LDRB (register)
LDR<c>BT	LDRBT<c>	LDRBT
LDR<c>D	LDRD<c>	LDRD (immediate), LDRD (literal), LDRD (register)
LDR<c>H	LDRH<c>	LDRH (immediate), LDRH (literal), LDRH (register)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
LDR<c>SB	LDRSB<c>	LDRSB (immediate). LDRSB (literal), LDRSB (register)
LDR<c>SH	LDRSH<c>	LDRSH (immediate), LDRSH (literal), LDRSH (register)
LDR<c>T	LDRT<c>	LDRT
MLA<c>S	MLAS<c>	MLA, MLAS
LSLS <Rd>, <Rn>, #0	MOVS <Rd>, <Rm>	MOV, MOVS (immediate), MOV, MOVS (register)
MOV<c>S	MOVS<c>	
MUL<c>S	MULS<c>	MUL, MULS
MVN<c>S	MVNS<c>	MVN, MVNS (immediate), MVN, MVNS (register), MVN, MVNS (register-shifted register)
ORR<c>S	ORRS<C>	ORR, ORRS (immediate), ORR, ORRS (register), ORR, ORRS (register-shifted register)
QADDSUBX	QASX	QASX
QSUBADDX	QSAX	QSAX
RSB<c>S	RSBS<c>	RSB, RSBS (immediate), RSB, RSBS (register), RSB, RSBS (register-shifted register)
RSC<c>S	RSCS<c>	RSC, RSCS (immediate), RSC, RSCS (register), RSC, RSCS (register-shifted register)
SADDSUBX	SASX	SASX
SBC<c>S	SBCS<c>	SBC, SBCS (immediate), SBC, SBCS (register), SBC, SBCS (register-shifted register)
SHADDSUBX	SHASX	SHASX
SHSUBADDX	SHSAX	SHSAX
SMI<c>	SMC<c>	SMC
SMLAL<c>S	SMLALS<c>	SMLAL, SMLALS
SMULL<c>S	SMULLS<c>	SMULL, SMULLS
SSUBADDX<c>	SSAX<c>	SSAX
STC<c>L	STCL<c>	STC
STM<c>EA, STM<c>IA	STM<c>	STM, STMIA, STMEA
STM<c>DA, STM<c>ED	STMDA<c>	STMDA, STMED
STM<c>DB, STM<c>FD	STMDB<c>	STMDB, STMFD
STM<c>IB, STM<c>FA	STMIB<c>	STMIB, STMFA
STR<c>B	STRB<c>	STRB (immediate), STRB (register)
STR<c>BT	STRBT<c>	STRBT
STR<c>D	STRD<c>	STRD (immediate), STRD (register)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
STR<c>H	STRH<c>	STRH (immediate), STRH (register)
STR<c>T	STRT<c>	STRT
SUB<c>S	SUBS<c>	SUB, SUBS (immediate), SUB, SUBS (register), SUB, SUBS (register-shifted register), SUB, SUBS (SP minus immediate), SUB, SUBS (SP minus register)
SWI	SVC	SVC
UADDSUBX	UASX	UASX
UHADDSUBX	UHASX	UHASX
UHSUBADDX	UHSAX	UHSAX
UMLAL<c>S	UMLALS<c>	UMLAL, UMLALS
UMULL<c>S	UMULLS<c>	UMULL, UMULLS
UQADDSUBX	UQASX	UQASX
UQSUBADDX	UQSAX	UQSAX
USUBADDX	USAX	USAX
UEXT8	UXTB	UXTB
UEXT16	UXTH	UXTH

K7.1.2 Pre-UAL instruction syntax for the A32 floating-point instructions

Table K7-2 lists the syntax for A32 floating-point instructions that have changed after UAL was introduced.

Table K7-2 Pre-UAL instruction syntax for A32 floating-point instructions

Pre-UAL syntax	UAL equivalent	See
FABSD	VABS.F64	VABS
FABSS	VABS.F32	
FADDD	VADD.F64	VADD (floating-point)
FADDS	VADD.F32	
FCMPEZD	VCMPE.F64	VCMPE
FCMPEZS	VCMPE.F32	
FCMPZD	VCMP.F64	VCMP,
FCMPZS	VCMP.F32	
FCONSTD <Dd>, #<imm8>	VMOV.F64 <Dd>, #<fpimm>	VMOV (immediate) For more information, see FCONST.
FCONSTS <Sd>, #<imm8>	VMOV.F32 <Sd>, #<fpimm>	
FCPYD	VMOV.F64	VMOV (register)
FCPYS	VMOV.F32	
FCVTDS	VCVT.F64.F32	VCVT (between double-precision and single-precision)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
FCVTSD	VCVT.F32.F64	
FDIVD	VDIV.F64	VDIV
FDIVS	VDIV.F32	
FLDD	VLDR.F64	VLDR (immediate) VLDR (literal)
FLDMD , FLDMIAD	VLDM.F64	VLDM, VLDMDB, VLDMIA
FLDMS	VLDM.F32	
FLDS	VLDR.F32	VLDR (immediate) VLDR (literal)
FMACD	VMLA.F64	VMLA (floating-point)
FMACS	VMLA.F32	
FMDHR <Dd>, <Rt>	VMOV <Dd[1]>, <Rt>	VMOV (general-purpose register to scalar)
FMDLR <Dd>, <Rt>	VMOV <Dd[0]>, <Rt>	
FMDRR	VMOV	VMOV (between two general-purpose registers and a doubleword floating-point register)
FMRDH <Rt>, <Dd>	VMOV <Rt>, <Dd[1]>	VMOV (scalar to general-purpose register)
FMRDL <Rt>, <Dd>	VMOV <Rt>, <Dd[0]>	
FMRRD	VMOV	VMOV (between two general-purpose registers and a doubleword floating-point register)
FMRRS	VMOV	VMOV (between two general-purpose registers and two single-precision registers)
FMRS	VMOV	VMOV (between general-purpose register and single-precision)
FMRX	VMRS	VMRS
FMSCD	VNMLS.F64	VNMLS
FMSCS	VNMLS.F32	
FMSR	VMOV	VMOV (between general-purpose register and single-precision)
FMSRR	VMOV	VMOV (between two general-purpose registers and two single-precision registers)
FMSTAT	VMRS APSR_nzcv, FPSCR	VMRS
FMULD	VMUL.F64	VMUL (floating-point)
FMULS	VMUL.F32	
FMXR	VMSR	VMSR
FNEGD	VNEG.F64	VNEG
FNEGS	VNEG.F32	
FNMACD	VMLS.F64	VNMLS
FNMACS	VMLS.F32	
FNMSCD	VNMLA.F64	VNMLA
FNMSCS	VNMLA.F32	
FNMULD	VNMUL.F64	VNMUL
FNMULS	VNMUL.F32	
FSHTOD	VCVT.F64.S16	VCVT (between floating-point and fixed-point, floating-point)
FSHTOS	VCVT.F32.S16	
FSITOD	VCVT.F64.S32	VCVT (between floating-point and integer, Advanced SIMD), VCVTR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
FSITOS	VCVT.F32.S32	
FSLTOD	VCVT.F64.S32	VCVT (between floating-point and fixed-point, floating-point)
FSLTOS	VCVT.F32.S32	
FSQRTD	VSQRT.F64	VSQRT
FSQRTS	VSQRT.F32	
FSTD	VSTR	VSTR
FSTMD, FSTMIAS	VSTM.F64	VSTM, VSTMDB, VSTMIA
FSTMS	VSTM.F32	
FSTS	VSTR	VSTR
FSUBD	VSUB.F64	VSUB (floating-point)
FSUBS	VSUB.F32	
FTOSHD	VCVT.S16.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOSHS	VCVT.S16.F23	
FTOSID	VCVT.S32.F64	VCVT (between floating-point and integer, Advanced SIMD)
FTOSIS	VCVT.S32.F32	
FTOSIZD	VCVTR.S32.F64	VCVTR
FTOSIZS	VCVTR.S32.F32	
FTOSLD	VCVT.S32.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOSLS	VCVT.S32.F32	
FTOUHD	VCVT.U16.F64	
FTOUHS	VCVT.U16.F32	
FTOUID	VCVT.U32.F64	VCVT (between floating-point and integer, Advanced SIMD)
FTOUIS	VCVT.U32.F32	
FTOUIZD	VCVTR.U32.F64	VCVTR
FTOUIZS	VCVTR.U32.F32	
FTOULD	VCVT.U32.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOULS	VCVT.U32.F32	
FUHTOD,	VCVT.F64.U16	
FUHTOS	VCVT.F64.U16	
FUITOD	VCVT.F64.U32	VCVT (between floating-point and integer, Advanced SIMD)
FUITOS	VCVT.F32.U32	
FULTOD	VCVT.F64.U32	VCVT (between floating-point and fixed-point, floating-point)
FULTOS	VCVT.F32.U32	

K7.1.3 FCONST

The syntax of `FCONST` is

```
FCONST<dest>{<c>} <Fd>, #<imm8>
```

where:

`<dest>` Specifies the destination data type. It must be one of:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`S` Single-precision floating-point.

`D` Double-precision floating-point.

`<c>` This is an optional field. It specifies the condition under which the instruction is executed. See Conditional execution for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (`AL`).

`<Fd>` Specifies the destination register. It must be one of:

`<Dd>` 64-bit name of the SIMD&FP destination register.

`<Sd>` 32-bit name of the SMID&FP destination register.

`<imm8>` Specifies the immediate value used to generate the floating-point constant.

`FCONSTD{<c>} <Dd>, #<imm8>` maps to `VMOV.F64 <Dd>, #<fpimm>`

`FCONSTS{<c>} <Sd>, #<imm8>` maps to `VMOV.F32 <Sd>, #<fpimm>`

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K8
Address Translation Examples

This appendix gives examples of address translations using the translation regimes described in The AArch64 Virtual Memory System Architecture and The AArch32 Virtual Memory System Architecture. It contains the following sections:

- AArch64 Address translation examples.
- AArch32 Address translation examples.

Note

This chapter gives examples of translation table lookups for the Armv8 address translation stages. It does not define any part of the address translation mechanism. If any information in this appendix appears to contradict the information in The AArch64 Virtual Memory System Architecture or The AArch32 Virtual Memory System Architecture then the information in The AArch64 Virtual Memory System Architecture or The AArch32 Virtual Memory System Architecture must be taken as the definition of the required behavior.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1 AArch64 Address translation examples

The VMSAv8 address translation stages that are controlled by an Exception level that is using AArch64 are described in Relationship between translation regimes and implemented Exception levels. Address translation describes the VMSAv8-64 address translation scheme. This section gives examples of the use of that scheme, for common translation requirements.

System registers relevant to MMU operation specifies the relevant registers, including the TCR_ELx and TTBR_ELx, or TTBR_ELx, for each stage of address translation.

For any stage of translation, a TCR_ELx.T*n*SZ field indicates the supported input address size. For a stage of address translation controlled from an Exception level using AArch64, the supported input address size is $2^{(64-TnSZ)}$.

This section describes:

- Performing the initial lookup, for an address for which the initial lookup is either:
  - At the highest lookup level used for the appropriate translation granule size.
  - Because of the concatenation of translation tables at the initial lookup level, one level down from the highest level used for the translation granule size.

  These descriptions take account of the following cases:
  - The IA size is smaller than the largest size for the translation level, see Input address size configuration.
  - For a stage 2 translation, translation tables are concatenated, to move the initial lookup level down by one level, see Concatenated translation tables.

  For examples of performing the initial lookup, see Examples of performing the initial lookup.
- The full translation flow for resolving a page of memory. These examples describe resolving the largest IA size supported by the initial lookup level. For these examples, see Full translation flows for VMSAv8-64 address translation.

K8.1.1 Examples of performing the initial lookup

The address ranges used for the initial translation table lookup depend on the translation granule, as described in:

- Performing the initial lookup using the 4KB translation granule.
- Performing the initial lookup using the 16KB granule.
- Performing the initial lookup using the 64KB translation granule.

K8.1.1.1 Performing the initial lookup using the 4KB translation granule

This subsection describes examples of the initial lookup when using the 4KB translation granule that VMSAv8-64 Stage 2 address translation using the 4KB translation granule describes as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in Translation process.

Note

For stage 2 translations, the same principles apply to an initial lookup that VMSAv8-64 Stage 2 address translation using the 4KB translation granule describes as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 0, 4KB translation granule.
- Initial lookup at level 1, 4KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see VMSAv8-64 Stage 2 address translation using the 4KB translation granule.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1.1.1.1 Initial lookup at level 0, 4KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As described in VMSAv8-64 translation using the 4KB granule, a stage 1 or stage 2 initial lookup at level 0 is required when $39 \leq n \leq 47$. For these lookups:

- TTBR_ELx[47:(n-35)] specify the translation table base address.
- Bits[n:39] of the input address are bits[(n-36):3] of the descriptor offset in the translation table.

Note

This means that, when the input address width is less than 48 bits:

- The size of the translation table is reduced.
- More low-order bits of the TTBR_ELx are required to specify the translation table base address.
- Fewer input address bit are used to specify the descriptor offset in the translation table.

For example, if the input address width is 46 bits:

- The translation table size is 1KB.
- TTBR_ELx bits[47:10] specify the translation table base address.
- Input address bits[45:39] specify bits[9:3] of the descriptor offset.

Figure K8-1 shows this lookup:

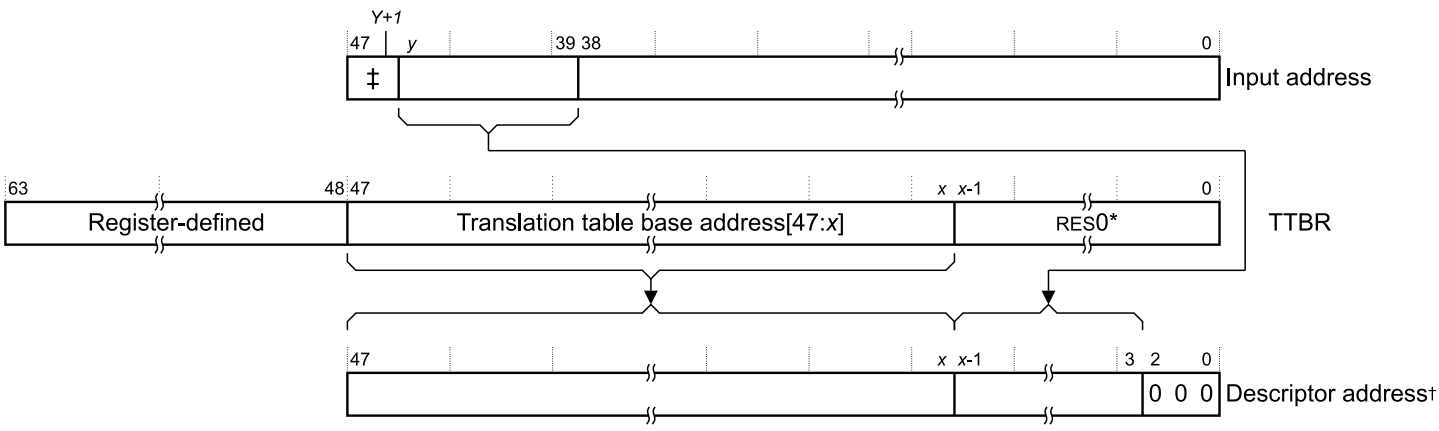

Supported input address range is IA[y:0], $4 \leq x \leq 12$, $y = x + 35$. When y is 47 the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-1 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 0

K8.1.1.1.2 Initial lookup at level 1, 4KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As described in VMSAv8-64 translation using the 4KB granule, this applies to IA[n:0], where $30 \leq n \leq 38$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-26)] specify the translation table base address.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset in the translation table.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure K8-2 shows this lookup:

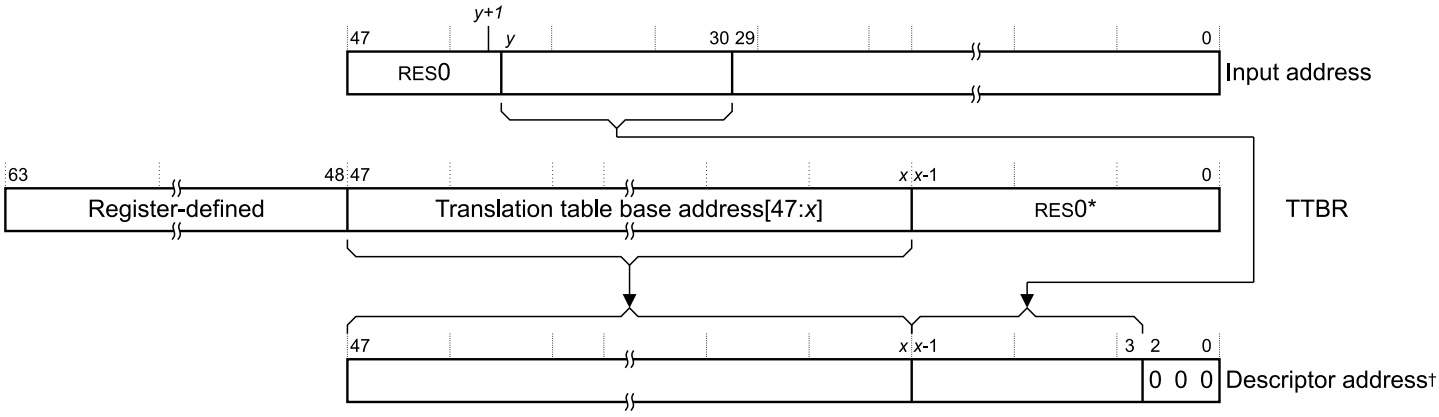

Supported input address range is IA[y:0], $4 \le x \le 12$, $y = x + 26$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-2 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As described in VMSAv8-64 Stage 2 address translation using the 4KB translation granule, this applies to IA[n:0], where $39 \le n \le 42$. For these lookups:

- There are $2^{(n-38)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(n-38)} \times$ 4KB. This means TTBR_ELx[(n -27):12] must be zero.
- TTBR_ELx[47:(n-26)] specify the base address of the block of concatenated translation tables.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K8-3 shows this lookup:

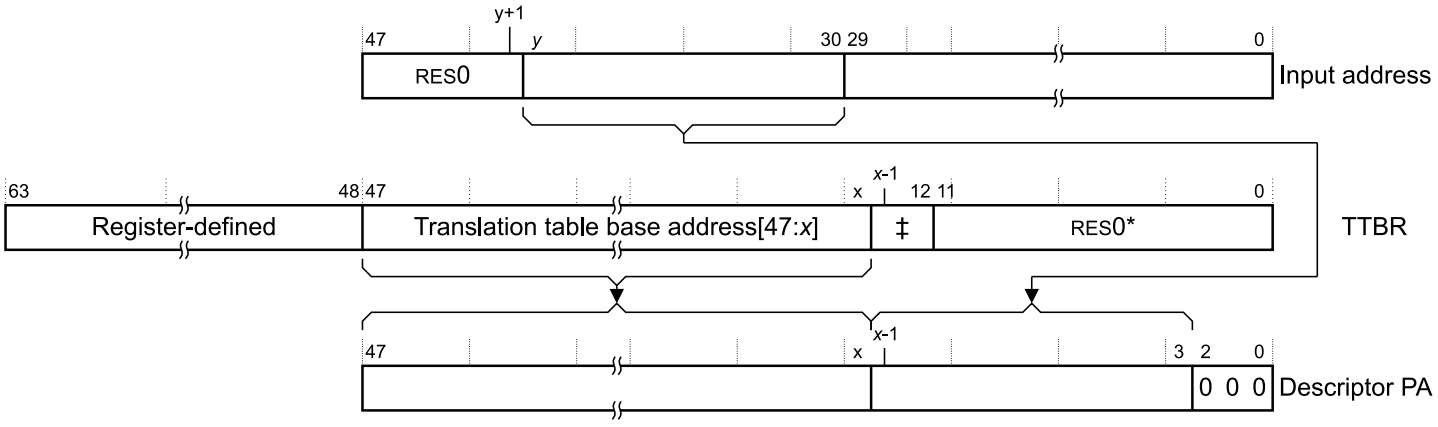

Supported input address range is IPA[y:0], $4 \le x \le 16$, $y = x + 26$. The field marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-3 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, with concatenation

K8.1.1.2 Performing the initial lookup using the 16KB granule

This subsection describes examples of the initial lookup when using the 16KB translation granule that VMSAv8-64 Stage 2 address translation using the 16KB translation granule describes as starting at level 0 or at level 1. It includes

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in Translation process.

Note

For stage 2 translations, the same principles apply to an initial lookup that VMSAv8-64 Stage 2 address translation using the 16KB translation granule describes as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 0, 16KB translation granule.
- Initial lookup at level 1, 16KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see VMSAv8-64 Stage 2 address translation using the 16KB translation granule.

K8.1.1.2.1 Initial lookup at level 0, 16KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As described in VMSAv8-64 translation using the 16KB granule, the only case where an address translation using the 16KB granule starts at level 0 is a stage 1 translation of a 48-bit input address, IA[47:0]. For this lookup:

- The required translation table has only two entries, meaning its size is 16 bytes, and it must be aligned to 16 bytes.
- TTBR_ELx[47:4] specify the translation table base address.
- Bit[47] of the input address is bit[3] of the descriptor offset in the translation table.

Figure K8-4 shows this lookup:

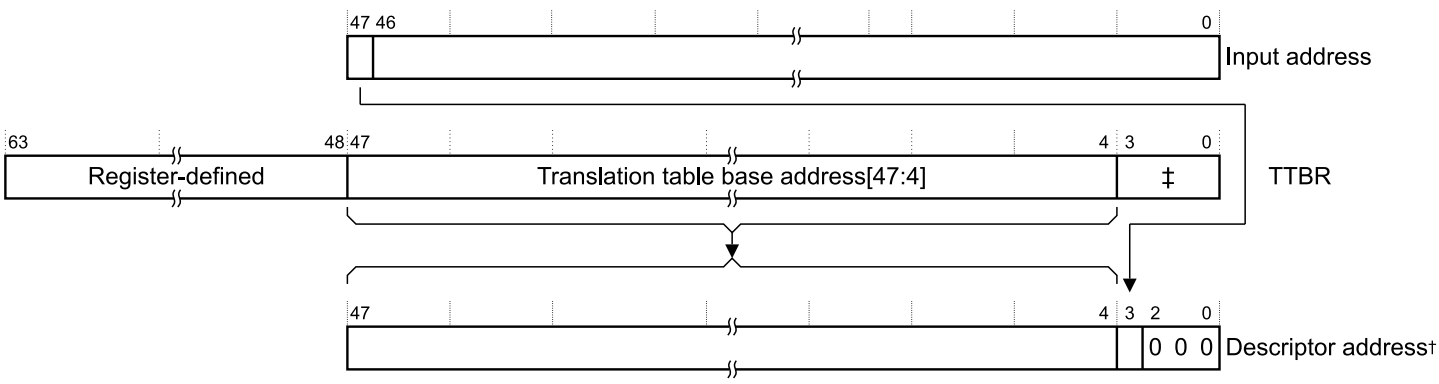

Supported input address range is IA[47:0]. The field marked ‡ is RES0*.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-4 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 0

K8.1.1.2.2 Initial lookup at level 1, 16KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As described in VMSAv8-64 translation using the 16KB granule, this applies to IA[n:0], where $36 \leq n \leq 46$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-32)] specify the translation table base address.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits[n:36] of the input address are bits[(n-33):3] of the descriptor offset in the translation table.

Figure K8-5 shows this lookup:

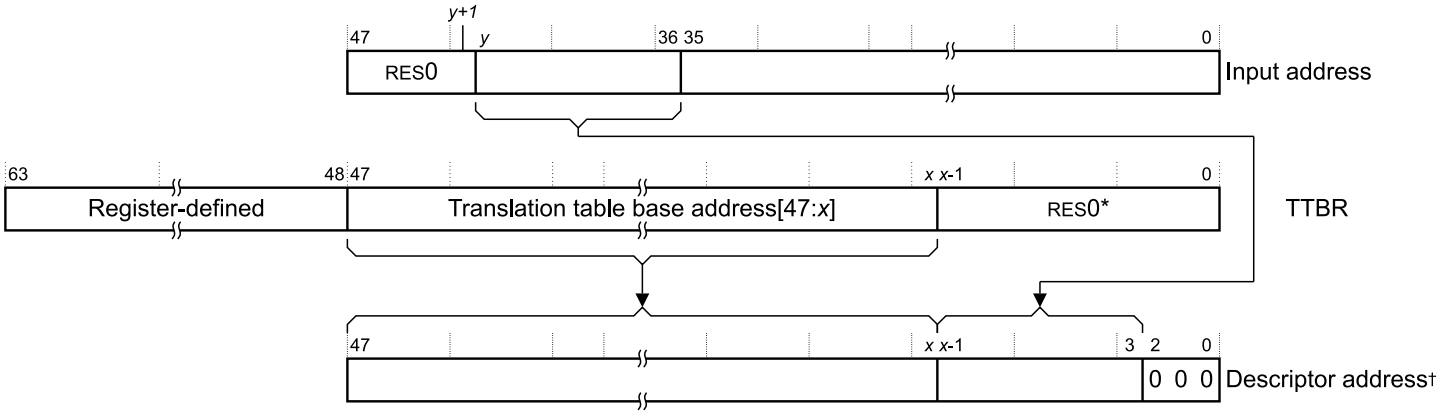

Supported input address range is IA[y:0], $4 \le x \le 14$, $y = x + 32$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-5 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As described in VMSAv8-64 Stage 2 address translation using the 16KB translation granule, the only case where an address translation using the 16KB granule starts at level 1 because of concatenation of translation tables is a stage 2 translation of a 48-bit input address, IA[47:0]. For this lookup:

- There are two concatenated translation tables at this level.
- These concatenated translation tables must be aligned to 2 × 16KB. This means TTBR_ELx[14] must be zero.
- TTBR_ELx[47:15] specify the base address of the block of two concatenated translation tables.
- Bits[47:36] of the input address are bits[14:3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K8-6 shows this lookup:

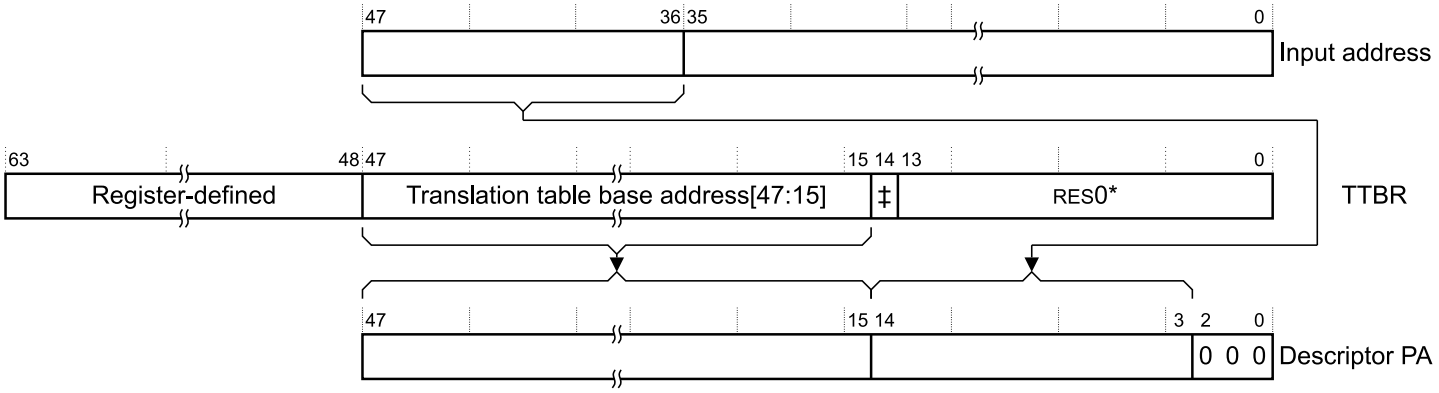

Supported input address range is IPA[47:0]. The bit marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-6 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, with concatenation

K8.1.1.3 Performing the initial lookup using the 64KB translation granule

This subsection describes examples of the initial lookup when using the 64KB translation granule that VMSAv8-64 Stage 2 address translation using the 64KB translation granule describes as starting at level 1 or at level 2. It includes

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 2. This means that it gives specific examples of the mechanisms described in Translation process.

Note

For stage 2 translations, the same principles apply to an initial lookup that VMSAv8-64 Stage 2 address translation using the 64KB translation granule describes as starting at level 2. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 3.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 1, 64KB translation granule.
- Initial lookup at level 2, 64KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see VMSAv8-64 Stage 2 address translation using the 64KB translation granule.

K8.1.1.3.1 Initial lookup at level 1, 64KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As described in VMSAv8-64 translation using the 64KB granule, a stage 1 or stage 2 initial lookup at level 1 is required when $42 \leq n \leq 47$. For these lookups:

- The size of the translation table is $2^{(n-39)}$ bytes. This means the size of the translation table, at this level, is always less than the granule size. The address of this translation table must align to the size of the table.
- Bits[n:42] of the input address are bits[(n - 39):3] of the descriptor offset in the translation table.
- Bits[47:(n-38)] of the TTBR_ELx specify the translation table base address.

Figure K8-7 shows this lookup:

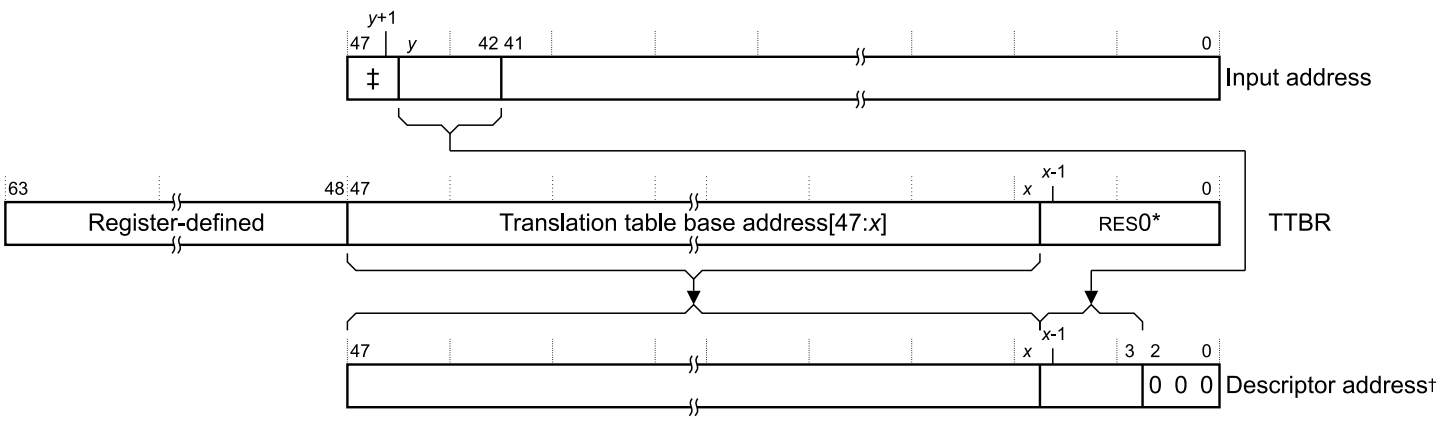

Supported input address range is IA[y:0], $4 \leq x$ $y = x + 38$. When y is 47 the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-7 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 1

K8.1.1.3.2 Initial lookup at level 2, 64KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

For a stage 1 or stage 2 initial lookup at level 2, without the use of concatenated translation tables

As described in VMSAv8-64 translation using the 64KB granule, this applies to IA[n:0], where $29 \leq n \leq 41$. For these lookups:

- There is a single translation table at this level.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TTBR_ELx[47:(n-25)] of the specify the translation table base address.
- Bits[n:29] of the input address are bits[(n-26):3] of the descriptor offset in the translation table.

Figure K8-8 shows this lookup:

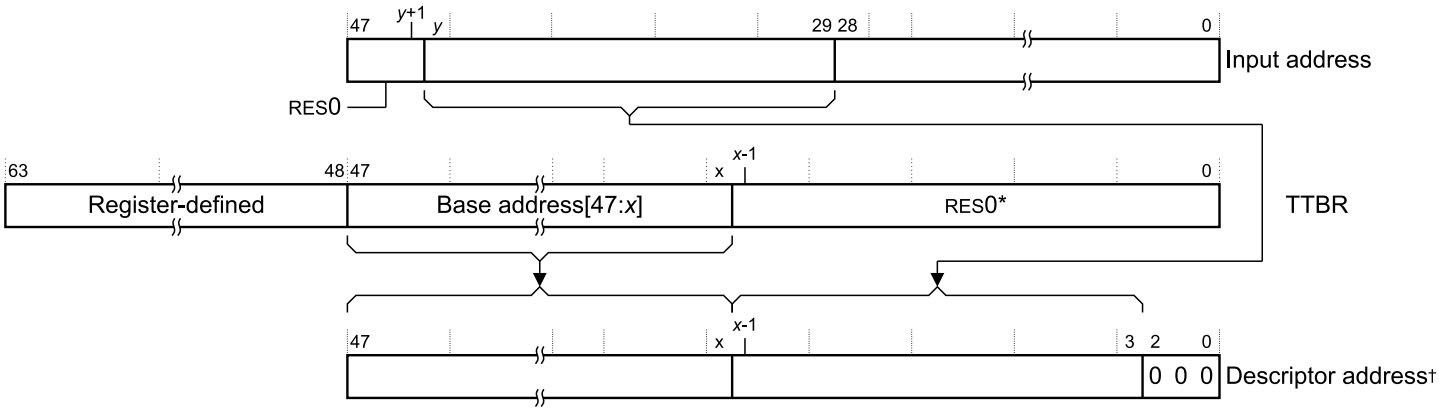

Supported input address range is IA[y:0]. $4 \le x \le 16$, $y = x + 25$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-8 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, without concatenation

For a stage 2 initial lookup at level 2, with concatenated translation tables

As described in VMSAv8-64 Stage 2 address translation using the 64KB translation granule, this applies to IA[n:0], where $42 \le n \le 45$. For these lookups:

- There are $2^{(m-41)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(m-41)} \times$ 64KB. This means TTBR_ELx[(n-26):16] must be zero.
- TTBR_ELx[47:(n-25)] specify the base address of the block of translation tables.
- Bits[n:42] of the input address are bits[(n-26):16] of the descriptor offset from the base address of the block of translation tables.

Figure K8-9 shows this lookup:

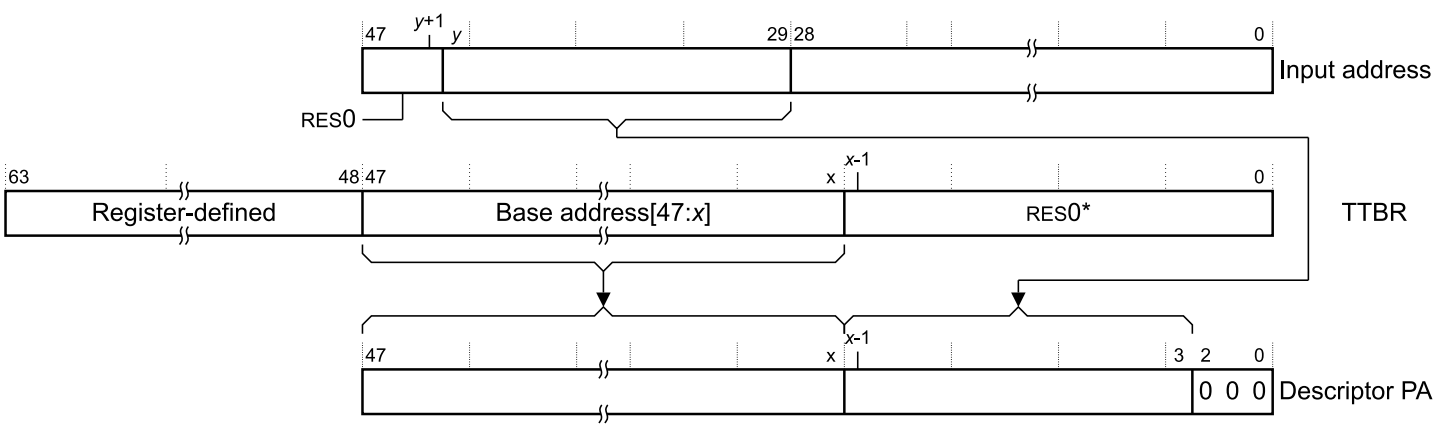

Supported input address range is IPA[y:0], $4 \le x \le 20$, $y = x + 25$.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-9 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, with concatenation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1.2 Full translation flows for VMSAv8-64 address translation

In a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR_ELx. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

This section describes example full translation flows, from the initial lookup to the address of a memory page. The example flows:

- Resolve the maximum-sized IA range supported by the initial lookup level.
- Do not have any concatenation of translation tables.
- Cover only the 4KB and the 64KB translation granules.

Examples of performing the initial lookup described how either reducing the IA range or concatenating translation tables affects the initial lookup.

Note

Reducing the IA range or concatenating translation tables affects only the initial lookup.

The following sections describe full VMSAv8-64 translation flows, down to an entry for a memory page:

- The address and properties fields shown in the translation flows.
- Full translation flow using the 4KB granule and starting at level 0.
- Full translation flow using the 4KB granule and starting at level 1.
- Full translation flow using the 64KB granule and starting at level 1.
- Full translation flow using the 64KB granule and starting at level 2.

K8.1.2.1 The address and properties fields shown in the translation flows

For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, an EL1&0 stage 2 translation is performed to translate the IPA to the required PA.

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see Translation table descriptor formats.

K8.1.2.2 Full translation flow using the 4KB granule and starting at level 0

Figure K8-10 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

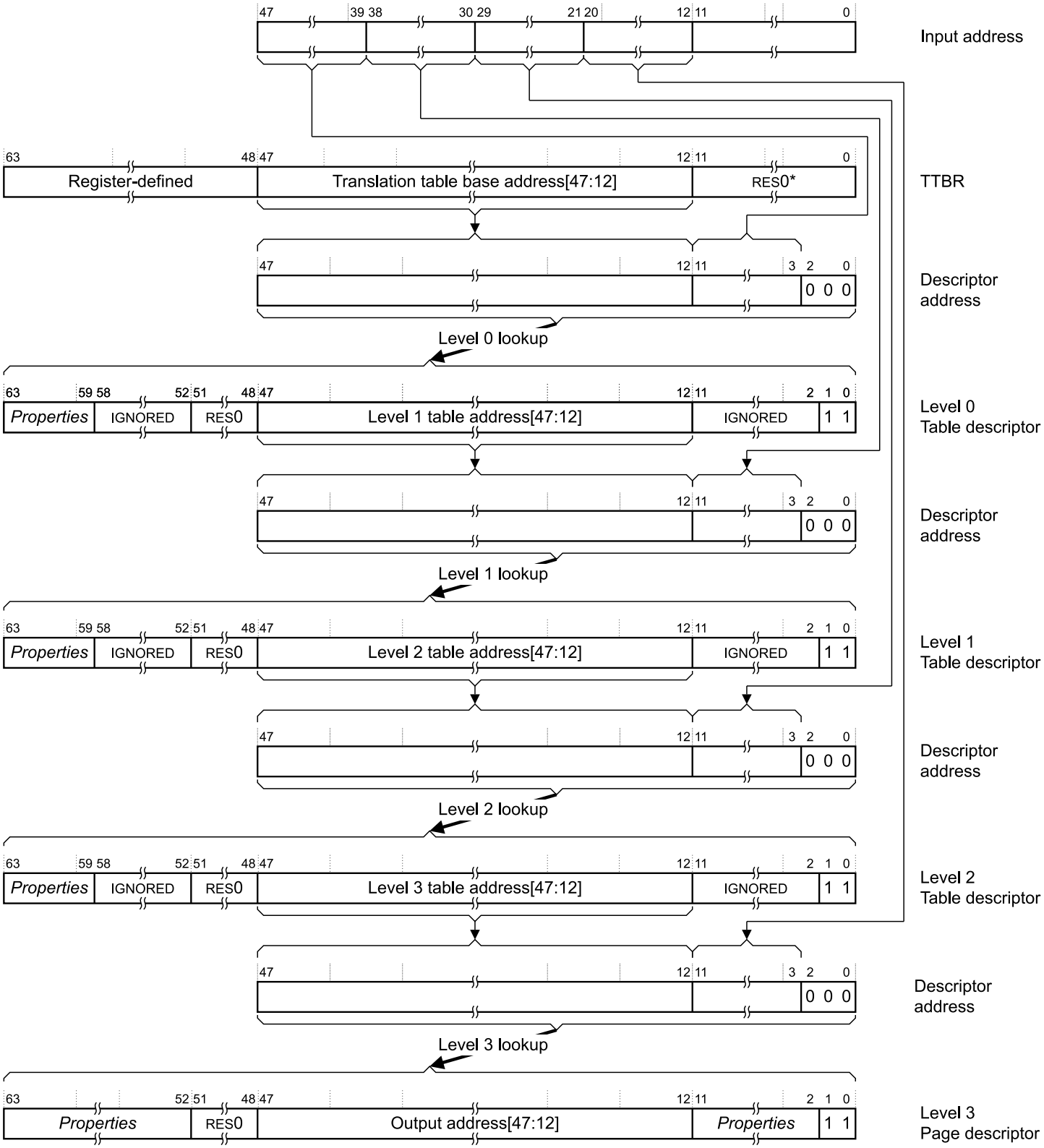

Figure K8-10 Complete stage 1 translation of a 48-bit address using the 4KB translation granule

If the level 1 lookup or level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-10 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

descriptors are SBZ.

K8.1.2.3 Full translation flow using the 4KB granule and starting at level 1

Figure K8-11 shows the complete translation flow for a stage 1 translation table walk for a 39-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

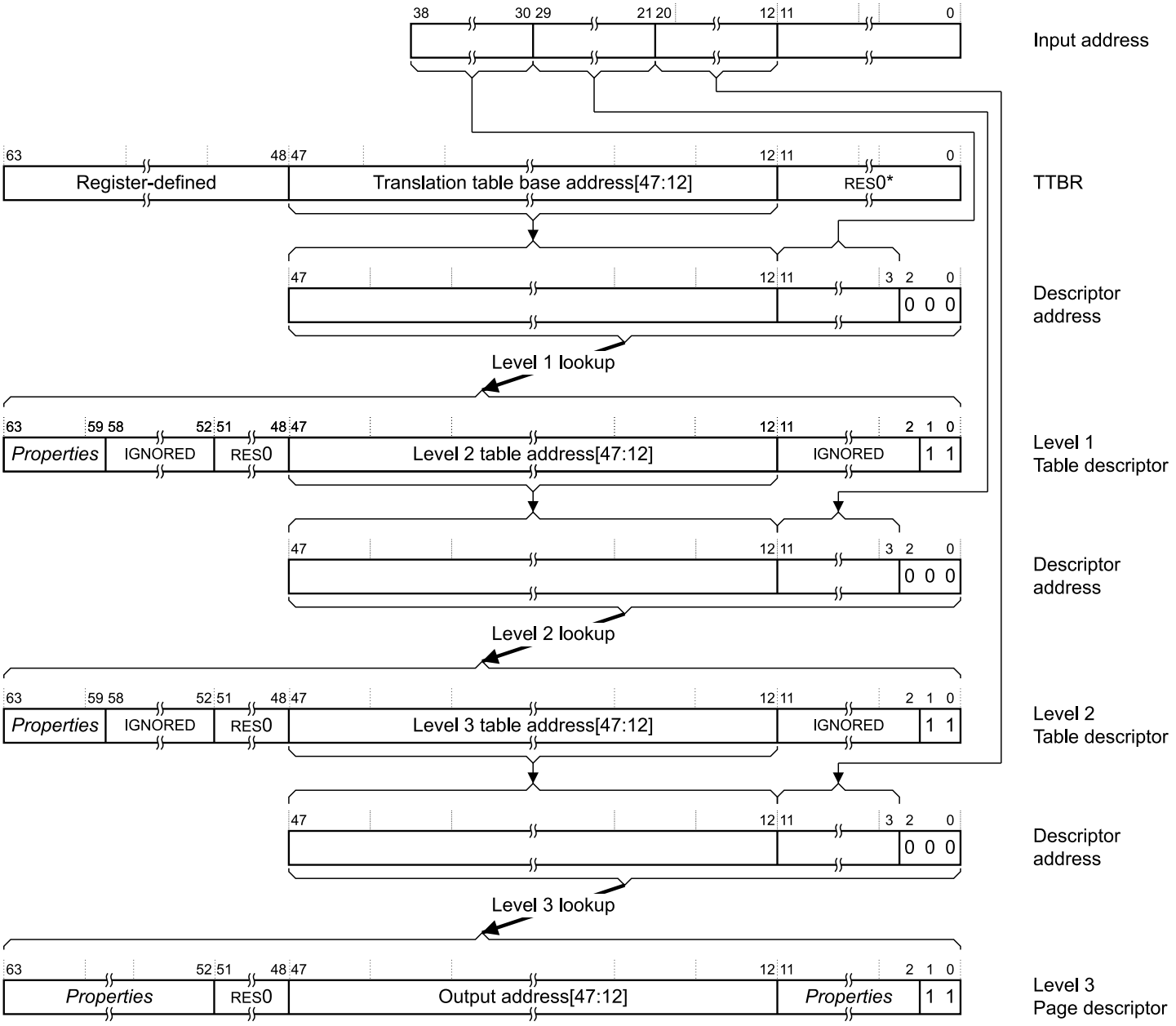

For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-11 Complete stage 1 translation of a 39-bit address using the 4KB translation granule

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-11 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K8-10, shows how the translation for the 42-bit address start the same lookup process one stage later.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1.2.4 Full translation flow using the 64KB granule and starting at level 1

Figure K8-12 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

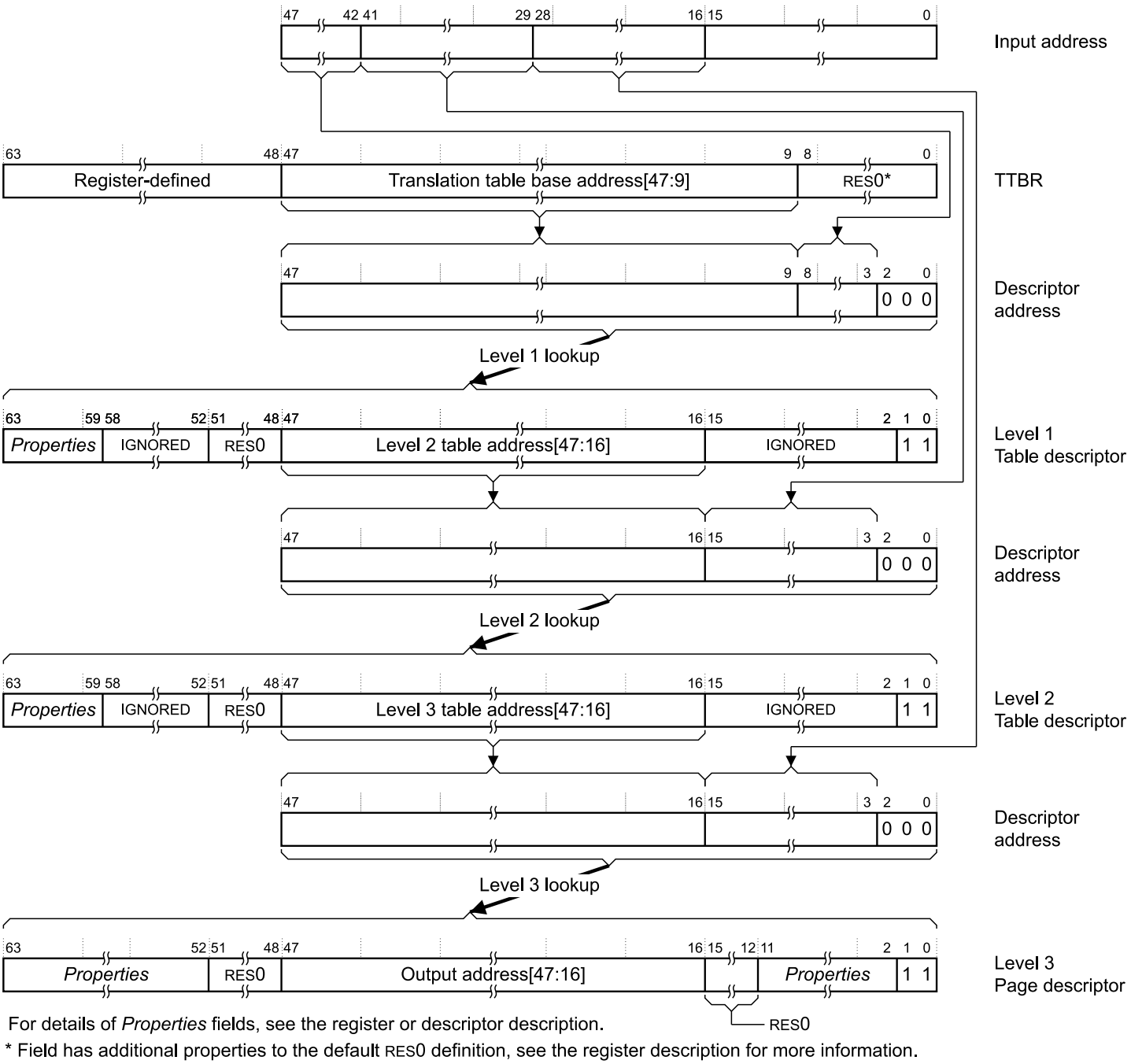

Figure K8-12 Complete stage 1 translation of a 48-bit address using the 64KB translation granule

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-12 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

The level 1 lookup resolves only 6 bits of the input address. As described in Performing the initial lookup using the 64KB translation granule, this means:

- The translation table size for this level is only 512 bytes.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The required translation table alignment for this level is 512 bytes.
- The Base address field in the TTBR_ELx is extended, at the low-order end, to be bits[47:9].

K8.1.2.5 Full translation flow using the 64KB granule and starting at level 2

Figure K8-13 shows the complete translation flow for a stage 1 translation table walk for a 42-bit input address. This lookup must start with a level 2 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

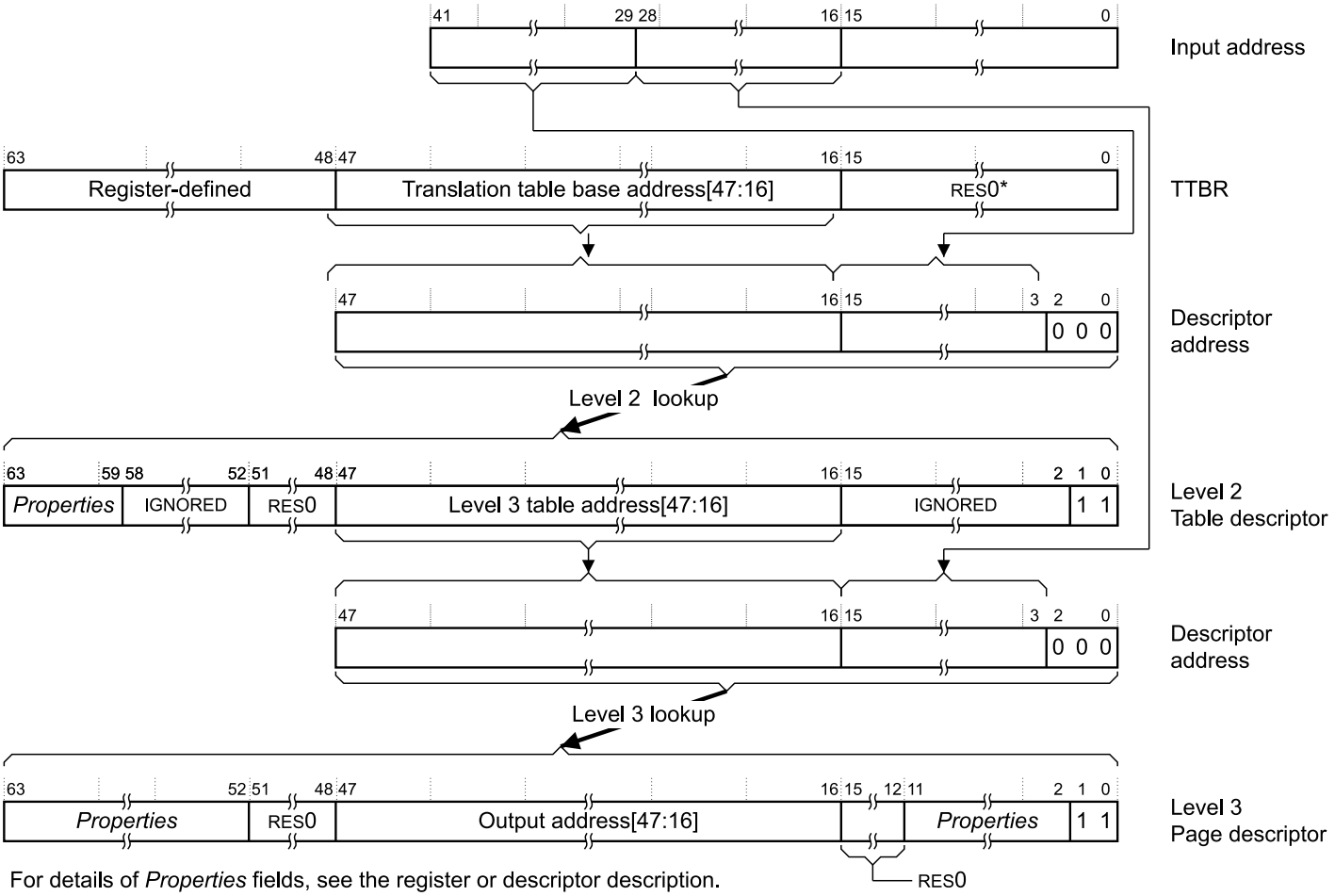

For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-13 Complete stage 1 translation of a 42-bit address using the 64KB translation granule

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-13 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K8-12, shows:

- The translation for the 42-bit address starts the same lookup process one stage later.
- Because the initial lookup resolves 13 bits of address:
  - The translation table size for this level is 64KB.
  - The required translation table alignment for this level is 64KB.
  - The Base address field in the TTBR_ELx is bits[47:16].

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.2 AArch32 Address translation examples

The following sections give address translation examples for the VMSAv8-32 address translation formats:

- Address translation examples using the VMSAv8-32 Short descriptor translation table format.
- Address translation examples using the VMSAv8-32 Long descriptor translation table format.

K8.2.1 Address translation examples using the VMSAv8-32 Short descriptor translation table format

VMSAv8-32 Short-descriptor Translation Table format descriptors describes the memory section and page option for a single VMSAv8-32 address translation. The following sections show the full translation flow for each of these options:

- Translation flow for a Supersection.
- Translation flow for a Section.
- Translation flow for a Large page.
- Translation flow for a Small page.

The address and Properties fields shown in the translation flows summarizes the information returned by the lookup.

K8.2.1.1 Translation flow for a Supersection

Figure K8-14 shows the complete translation flow for a Supersection. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

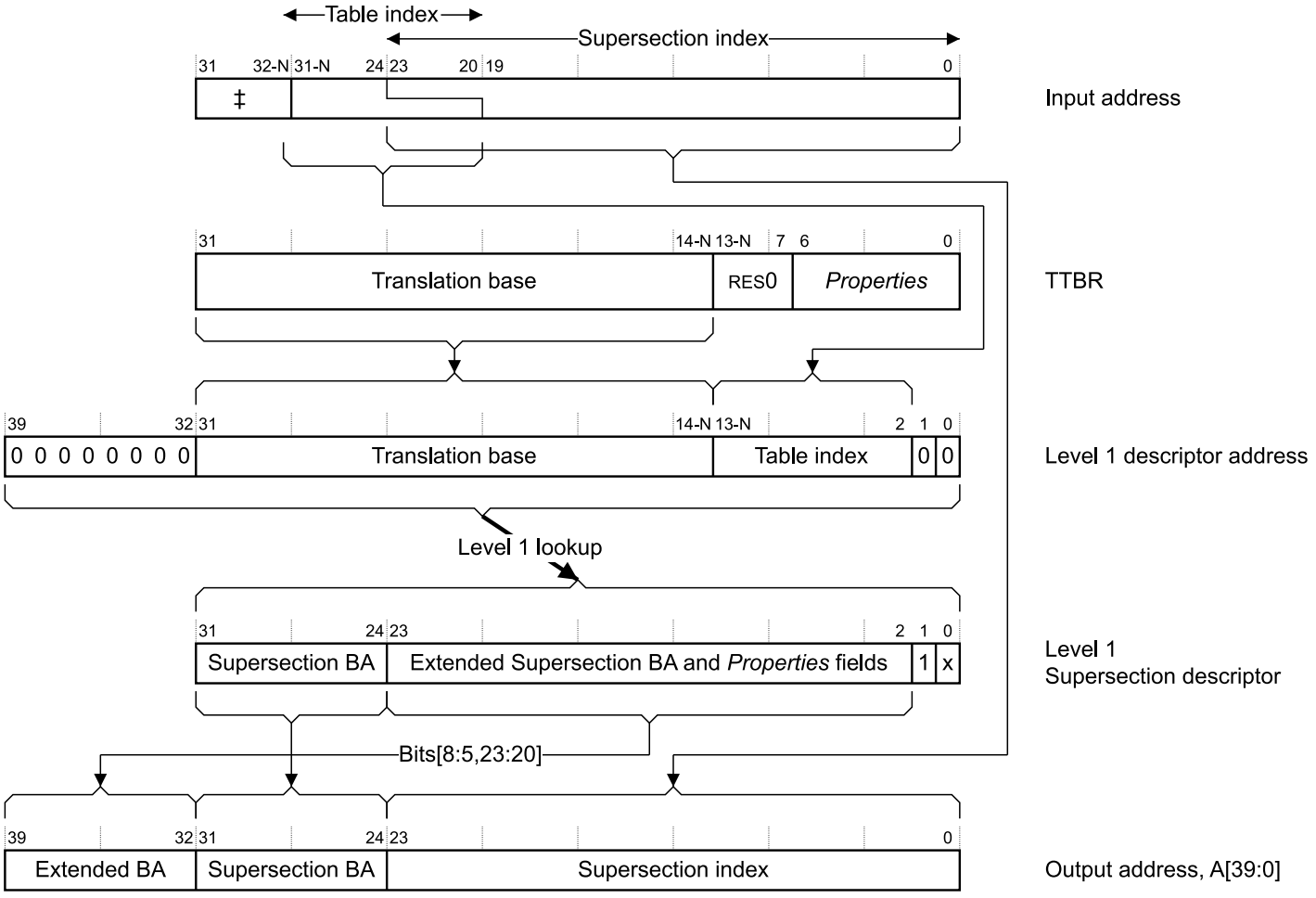

‡ This field is absent if N is 0.
BA = Base address.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of *Properties* fields, see the register or descriptor description.

Figure K8-14 VMSAv8-32 Short-descriptor Supersection address translation

Note

Figure K8-14 shows how, when the input address, the VA, addresses a Supersection, the top four bits of the *Supersection index* bits of the address overlap the bottom four bits of the *Table index* bits. For more information, see Additional requirements for Short-descriptor format translation tables.

K8.2.1.2 Translation flow for a Section

Figure K8-15 shows the complete translation flow for a Section. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

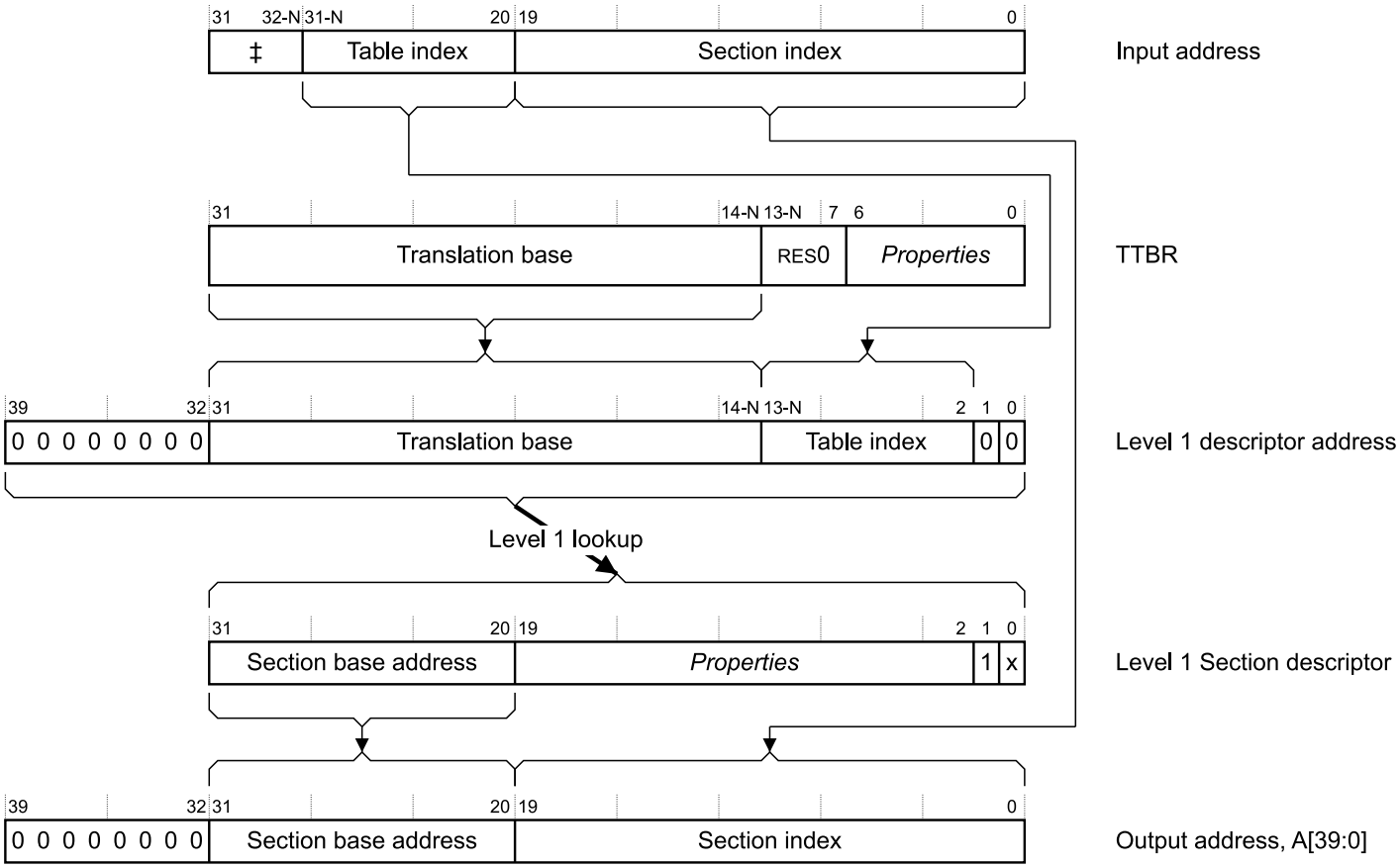

‡ This field is absent if N is 0.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of *Properties* fields, see the register or descriptor description.

Figure K8-15 VMSAv8-32 Short-descriptor Section address translation

K8.2.1.3 Translation flow for a Large page

Figure K8-16 shows the complete translation flow for a Large page. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

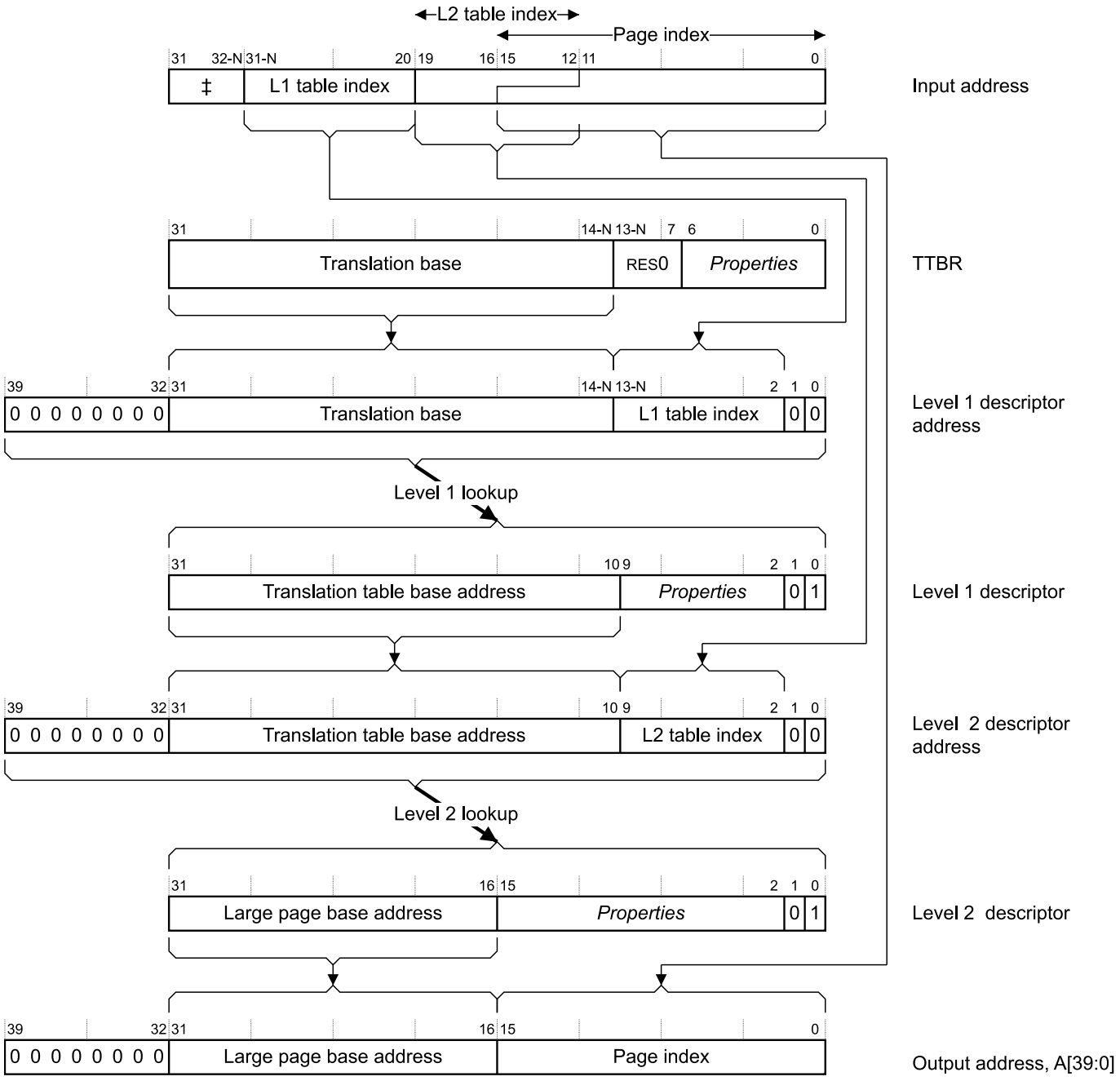

‡ This field is absent if N is 0.

L1 = Level 1, L2 = Level 2.

For a translation based on TTBR0, N is the value of TTBCR.N.

For a translation based on TTBR1, N is 0.

For details of *Properties* fields, see the register or descriptor description.

Figure K8-16 VMSAv8-32 Short-descriptor Large page address translation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Figure K8-16 shows how, when the input address, the VA, addresses a Large page, the top four bits of the *page index* bits of the address overlap the bottom four bits of the *level 1 table index* bits. For more information, see Additional requirements for Short-descriptor format translation tables.

K8.2.1.4 Translation flow for a Small page

Figure K8-17 shows the complete translation flow for a Small page. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

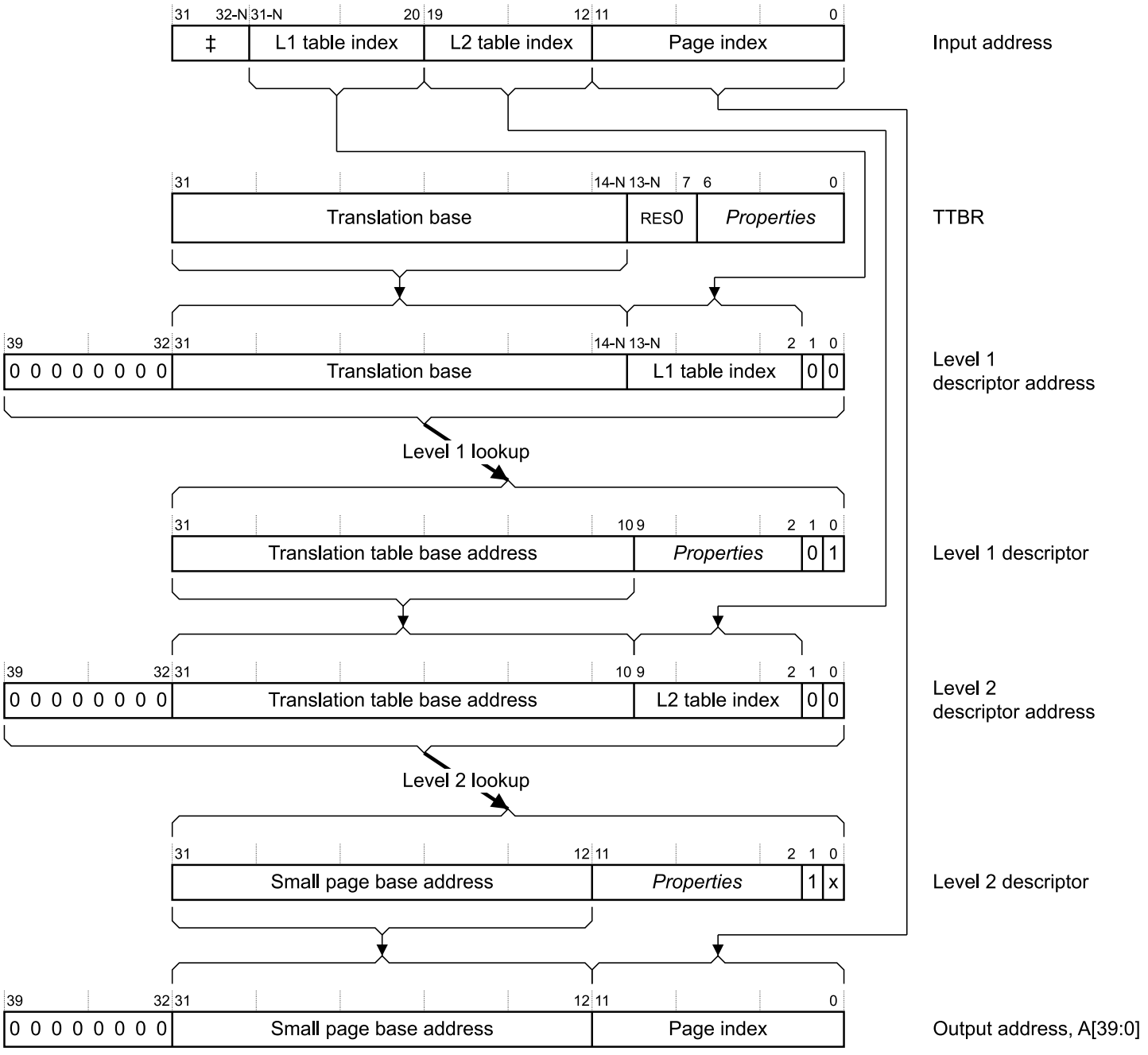

‡ This field is absent if N is 0.
L1 = Level 1, L2 = Level 2.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of *Properties* fields, see the register or descriptor description.

Figure K8-17 VMSAv8-32 Short-descriptor Small page address translation

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.2.1.5 The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation tables:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the Section, Supersection, Large page, or Small page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

Otherwise, the address is the PA of the descriptor, Section, Supersection, Large page, or Small page.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see Information returned by a translation table lookup, and the description of the register or translation table descriptor.

For translations using the Short-descriptor translation table format, VMSAv8-32 Short-descriptor Translation Table format descriptors describes the descriptors formats.

K8.2.2 Address translation examples using the VMSAv8-32 Long descriptor translation table format

As described in Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format, in a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

The following sections give examples of full VMSAv8-32 Long-descriptor format address translation flows, down to an entry for a 4KB page:

- Full translation flow, starting at level 1 lookup.
- Full translation flow, starting at level 2 lookup.

The address and Properties fields shown in the translation flows summarizes the information returned by the lookup.

K8.2.2.1 Full translation flow, starting at level 1 lookup

Figure K8-18 shows the complete translation flow for a VMSAv8-32 Long-descriptor stage 1 translation table walk that starts with a level 1 lookup. For more information about the fields shown in the figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

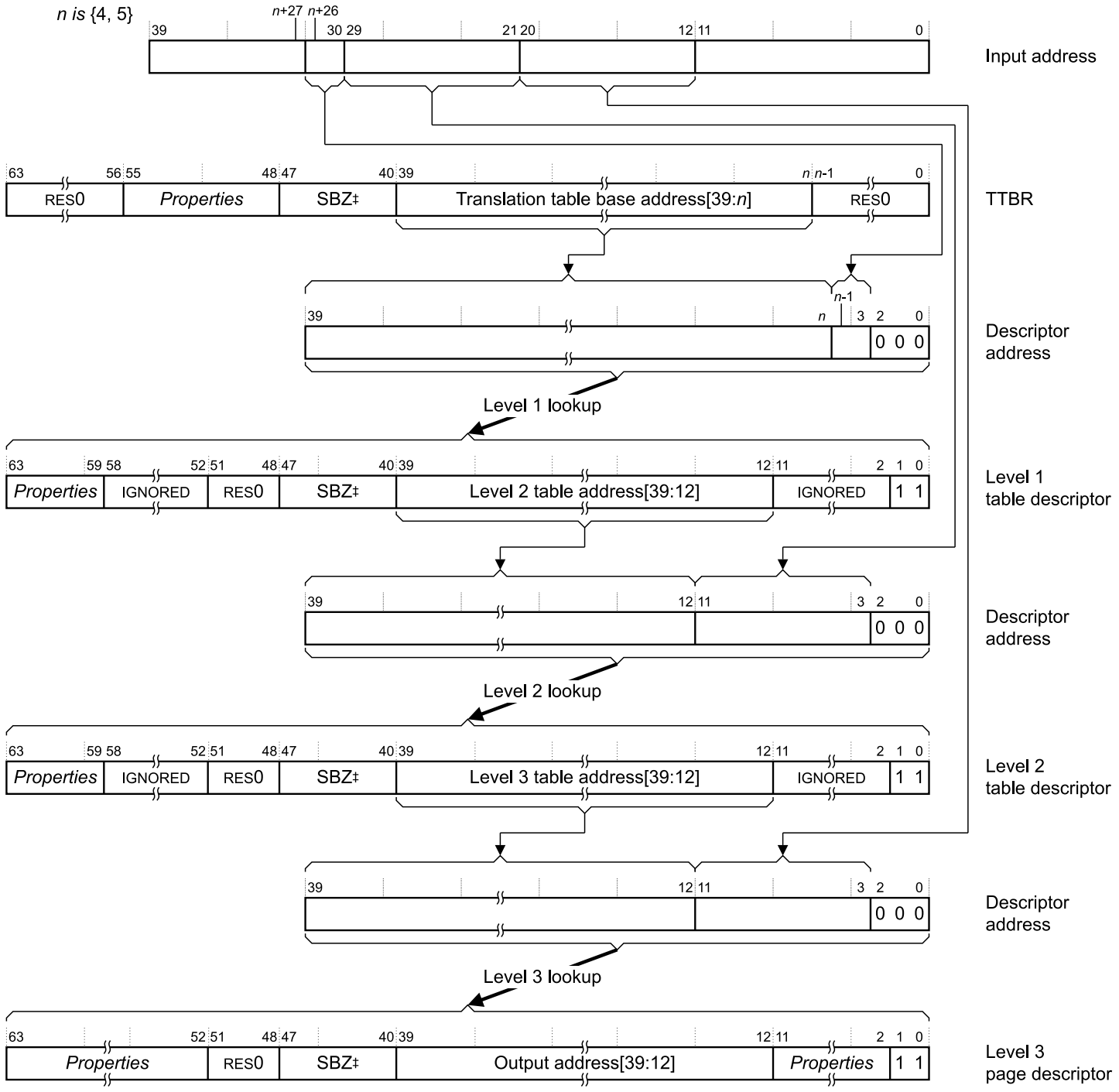

For details of *Properties* fields, see the register or descriptor description.

‡ See the lookup description for more information about bits[40:47] of the TTBR and descriptors

Figure K8-18 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 1

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see Address size fault.

A stage 2 translation that starts at a level 1 lookup differs from the translation shown in Figure K8-18 only as follows:

- The possible values of *n* are 4-13, to support an input address of between 31 and 40 bits.
- A descriptor and output addresses are always PAs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.2.2.2 Full translation flow, starting at level 2 lookup

Figure K8-19 shows the complete translation flow for a stage 1 VMSAv8-32 Long-descriptor translation table walk that starts at a level 2 lookup. For more information about the fields shown in the figure, see The address and Properties fields shown in the translation flows.

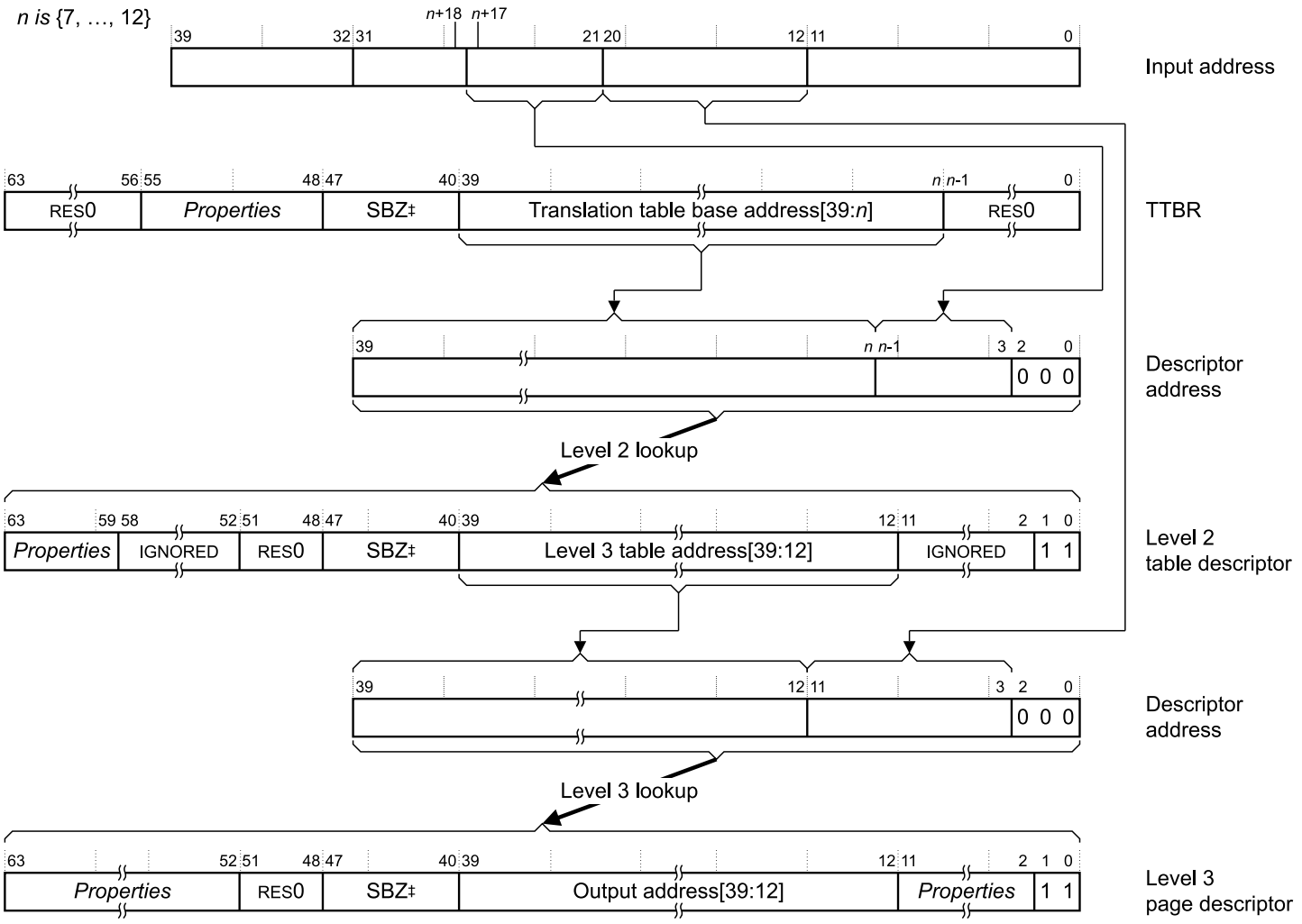

Figure K8-19 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 2

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see Address size fault.

A stage 2 translation that starts at a level 2 lookup differs from the translation shown in Figure K8-19 only as follows:

- The possible values of *n* are 7-16, to support an input address of up to 34 bits.
- The descriptor and output addresses are always PAs.

K8.2.2.3 The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see Information returned by a translation table lookup, and the description of the register or translation table descriptor.

For translations using the Long-descriptor translation table format, VMSAv8-32 Long-descriptor Translation Table format descriptors describes the descriptors formats.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K9
Recommended Upload and Download Processes for External Debug

This appendix contains the following section:

- Using Memory access mode in AArch64 state.

Note

This description is not part of the Arm architecture specification. It is supplementary, for the convenience of developers and users who might find this information useful.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K9.1 Using Memory access mode in AArch64 state

Figure K9-1 and Figure K9-2 show the processes for using Memory access mode to implement a download (external host to target) and an upload (target to external host).

To transfer *n* words of data:

- The download sequence needs *n*+6 accesses by the external debug interface.
- The upload sequence needs *n*+8 accesses by the external debug interface.

In both cases, in the innermost loop the debugger can make an external access to a DTR without polling EDSCR after each write as underrun and overrun detection prevent failure. Normally external accesses from the debugger are outpaced by the memory accesses of the PE, making underruns and overruns unlikely. If this is not the case, the EDSCR.ERR flag is set to 1. This is checked once at the end of the sequence, although a debugger can check it more often, for example once for each page. If the EDSCR.ERR flag is set to 1 because of overrun or underrun, the debugger can restart. The address to restart from is frozen in X0. EDSCR.ERR might also be set because of a Data Abort.

If underruns and overruns are common, the debugger can pace itself accordingly.

Note

- The base address must be a multiple of 4.
- The order of the writes that set up the address does not matter in Debug state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

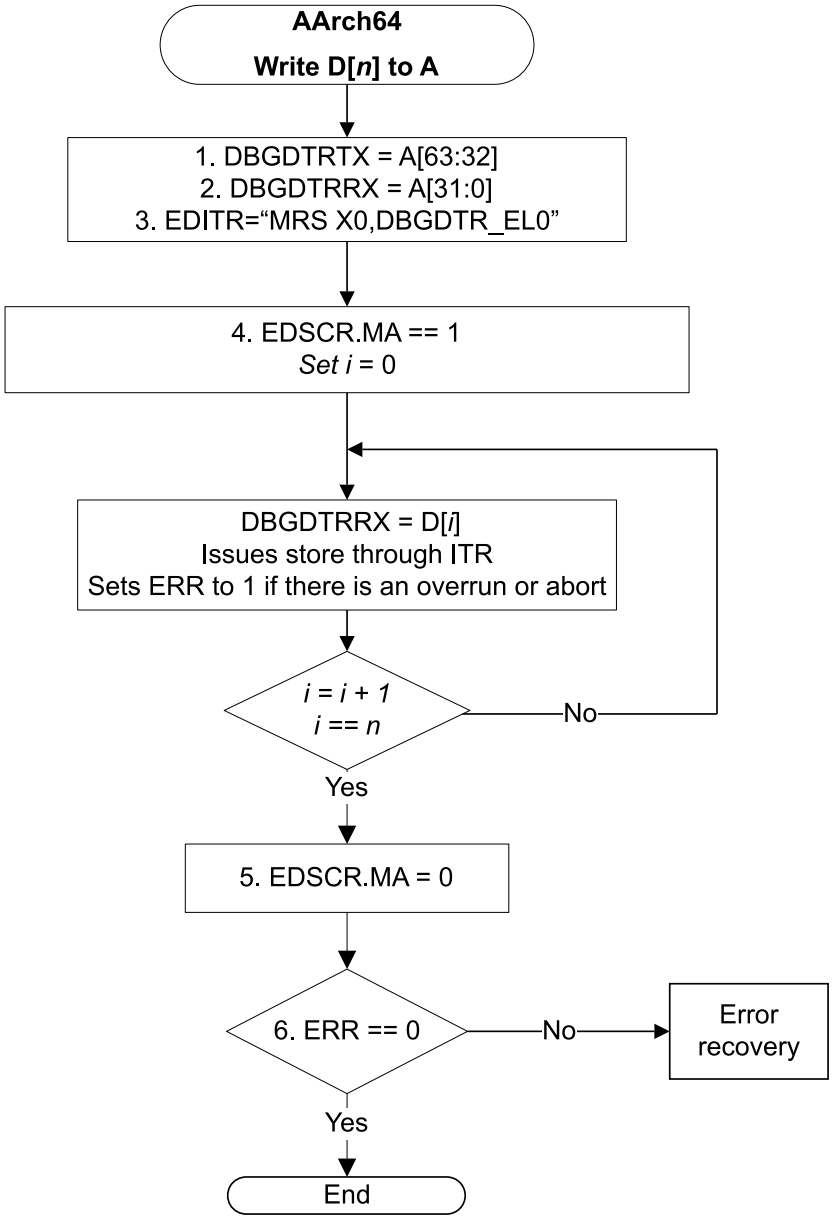

Figure K9-1 Fast code download in AArch64 state (external host to target)

In Figure K9-1, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR. The PE executes this instruction.
   d. Set EDSCR.MA to 1.
2. Loop *n* times. From the external debug interface:
   a. Write to DBGDTRRX_EL0. The PE reads the word from DTRRX and stores it to memory. It increments X0 by 4.
3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read EDSCR to check for overruns or Data Aborts during download.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

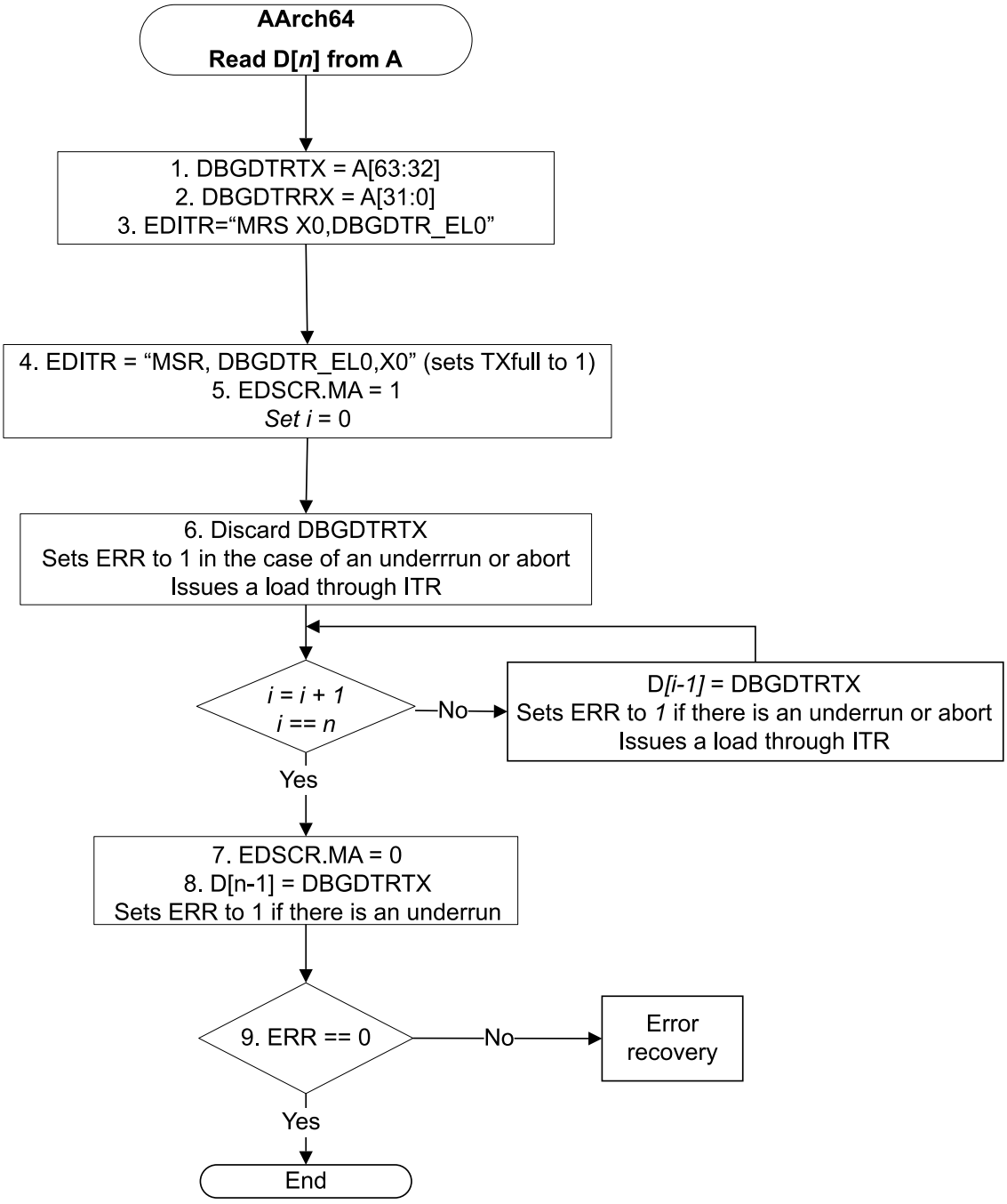

Figure K9-2 Fast data upload in AArch64 state (target to external host)

In Figure K9-2, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR.
   d. Write `MSR DBGDTR_EL0, X0` to EDITR. This dummy operation ensures EDSCR.TXfull == 1.
   e. Set EDSCR.MA to 1.
   f. Read DBGDTRTX_EL0 and discard the value. The PE returns the previous DTR value, loads the first word, and writes it to DTR. It increments X0 by 4.
2. Loop *n*-1 times. From the external debug interface:
   a. Read DBGDTRTX_EL0. The PE returns the previous DTRTX value, loads a new word, and writes it to DTRTX. It increments X0 by 4.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read DBGDTRTX_EL0 for the nth value.
   c. Read EDSCR to check for underruns, overruns or Data Aborts during upload.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K10
Software Usage Examples

This appendix gives Software usage examples, for cases where these are likely to contribute significantly to an understanding of the Arm architecture.

It contains the following sections:

- Use of the Advanced SIMD complex number instructions
- Use of the Armv8.2 extensions to the Cryptographic Extension
- SVE and SVE2 Programming Examples
- Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) hint instructions
- Use of the Embedded Trace and Trace Buffer Extensions
- Use of the GCS Extension
- Use of the RAS Extension
- Example OS Save and Restore sequences
- RME example software flows
- MEC example software flows

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.1 Use of the Advanced SIMD complex number instructions

FEAT_FCMA provides instructions to aid floating-point calculations of complex numbers. This section illustrates the use of these instructions for complex arithmetic. It is not part of the Arm architecture definition.

This section uses the AArch64 instructions `FCADD` and `FCMLA` - usage of the AArch32 instructions `VCADD` and `VCMLA` is similar.

When using the instructions implemented by FEAT_FCMA, a complex number is represented in a SIMD&FP register as a pair of adjacent elements, each holding a floating-point number, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number.

K10.1.1 Complex addition

Simple complex addition on a vector of complex numbers is already provided by the vector form of the `FADD` instruction.

The functionality that `FCADD` adds is to rotate each complex number in the second vector by 90 degrees or 270 degrees counterclockwise (considering the complex numbers on an Argand diagram) before performing the addition. Mathematically, this is equivalent to multiplying the second complex number by i or -i before addition.

This means, given a complex number z stored in a pair of elements in one vector, and a complex number w stored in the corresponding element pair in another vector:

- `FADD` calculates z + w.
- `FCADD` calculates z ± iw.

K10.1.2 Complex multiplication

The `FCMLA` instruction does not provide functionality for complex multiplication directly. However, a pair of `FCMLA` instructions can provide this function.

The `FCMLA` instruction operates on corresponding pairs of complex numbers stored in SIMD&FP vector registers, and adds the result to the corresponding complex number in the destination SIMD&FP vector register. This calculation is as follows:

1. The second complex number is rotated by 0, 90, 180 or 270 degrees counterclockwise.
2. That complex number is multiplied by either the real or imaginary part of the first complex number:
   - When the rotation is 0 or 180 degrees, the real part is used.
   - When the rotation is 90 or 270 degrees, the imaginary part is used.
3. The resulting complex number is added to the corresponding complex number in the destination register.

Mathematically, considering the complex numbers on an Argand diagram:

- Rotation by 180 degrees is equivalent to negation.
- Rotation by 90 degrees is equivalent to multiplying by i.
- Rotation by 270 degrees is equivalent to multiplying by -i.

This means that, for a first complex number z, where z = a+bi, and a second complex number w, if initially the corresponding complex number in the destination register is zero:

- When the rotation is 0 degrees the result of the multiply-add is aw.
- When the rotation is 180 degrees, the result is -aw.
- When the rotation is 90 degrees, the result is biw.
- When the rotation is 270 degrees, the result is -biw.

This means that, if the destination register is zeroed and an `FCMLA` instruction is executed with a rotation parameter of 0, and then the same instruction is executed with a rotation parameter of 90:

- The first execution returns aw in the destination register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The second execution accumulates biw to this, meaning the result is aw+biw.
- This result is the product of (a+bi)w, which is the product zw.

So, this pair of instructions can be used to implement complex multiplication.

After zeroing V0, the syntax of a pair of instructions to perform this complex number multiplication might be:

```
FCMLA V0.4S, V1.4S, V2.4S, #0
FCMLA V0.4S, V1.4S, V2.4S, #90
```

Other simple pairs of `FCMLA` instructions perform useful calculations. For example, considering a first complex number z and second complex number w, defined as before, and a destination register that has been zeroed before the first `FCMLA` instruction is executed:

1. The following pair of instructions calculates the complex conjugate of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #0
   FCMLA V0.4S, V1.4S, V2.4S, #270
   ```

2. The following pair of instructions calculates the negation of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #180
   FCMLA V0.4S, V1.4S, V2.4S, #270
   ```

3. The following pair of instructions calculates the negation of the complex conjugate of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #180
   FCMLA V0.4S, V1.4S, V2.4S, #90
   ```

Note

For these examples, the following caveats must be considered:

- `FCMLA` performs a fused multiply-add, meaning there is no intermediate rounding. This lack of intermediate rounding can give unexpected results in some cases. Arm expects that these instructions are only used in situations where the effect of the rounding of these results is not material to the calculation.
- When using the `FCMLA` instructions, the behavior of (∞+∞i) multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex ∞.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.2 Use of the Armv8.2 extensions to the Cryptographic Extension

K10.2.1 Use of the SHA512 instructions

These instructions are implemented when FEAT_SHA512 is implemented.

The following code sequence shows the use of the SHA512 instructions to calculate a SHA512 hash iteration of 80 rounds. This code is not fully optimized.

```
// X0 contains the pointer to the bottom of the (padded) 16*64 bytes of message to be
// hashed, with space above the that message to hold a further 64 * 64 bytes of working
// data
// X1 contains the pointer to the 0th element of 80 64-bit constants (in ascending addresses) defined in
    ↪the SHA2 specification
// X2 contains a loop variable
// V4,V5,V6, V7 hold VS0 to VS3 respectively
// V8 holds running hash V1
// V9 holds running hash V0
    MOV        X2, #0
loop1:
  LD1        {V0.2D}, [X0]                 // Data
  LD1        {V1.2D}, [X1]                 // K values
  ADD        X1, X1, #16
  ADD        X0, X0, #16
  ADD        X2, X2, #16
  ADD        V2.2D, V0.2D, V1.2D
  EXT        V2.16B, V2.16B, V2.16B, #8
  EXT        V8.16B, V6.16B, V7.16B, #8
  EXT        V9.16B, V5.16B, V6.16B, #8
  ADD        V7.2D, V7.2D, V2.2D
  SHA512H    Q7, Q8, V9.2D
  ADD        V10.2D, V5.2D, V7.2D
  SHA512H2   Q7, Q5, V4.2D
  MOV        V5.16B, V4.16B
  MOV        V4.16B, V7.16B
  MOV        V7.16B, V6.16B
  MOV        V6.16B, V10.16B
  CMP        X2, #128
  BLT        loop1

  // work out pointers to previous words in the data
  SUB       X3, X0, #128
  SUB       X4, X0, #112
  SUB       X5, X0, #16
  SUB       X6, X0, #56
loop2:
  LD1      {V11.2D}, [X3]
  LD1      {V12.2D}, [X4]
  LD1      {V13.2D}, [X5]
  LD1      {V14.2D}, [X6]
  SHA512SU0 V11.2D, V12.2D
  SHA512SU1 V11.2D, V13.2D, V14.2D
  ST1      {V11.2D}, [X0]
  LD1      {V1.2D}, [X1]                     // K values
  ADD      X0, X0, #16
  ADD      X1, X1, #16
  ADD      X3, X3, #16
  ADD      X4, X4, #16
  ADD      X5, X5, #16
  ADD      X6, X6, #16
  ADD      X2, X2, #16
  ADD      V2.2D, V11.2D, V1.2D
  EXT      V2.16B, V2.16B, V2.16B, #8
  EXT      V8.16B, V6.16B, V7.16B, #8
  EXT      V9.16B, V5.16B, V6.16B, #8
  ADD      V7.2D, V7.2D, V2.2D
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SHA512H  Q7, Q8, V9.2D
ADD      V10.2D, V5.2D, V7.2D
SHA512H2 Q7, Q5, V4.2D
MOV      V5.16B, V4.16B
MOV      V4.16B, V7.16B
MOV      V7.16B, V6.16B
MOV      V6.16B, V10.16B
CMP      X2, #320
BLT      loop2
```

K10.2.2 Use of the SHA3 instructions

These instructions are implemented when FEAT_SHA3 is implemented.

The following code sequence shows the use of the SHA3 instructions to obtain the combined theta, phi, rho and chi operations of a SHA3 iteration. Arm expects the iota operation to be performed using a lookup table.

This code is not fully optimized for multiple iterations.

```
// Input State:
    //      x=0   x=1   x=2   x=3   x=4
    // y=0  v12   v13   v14   v10   v11
    // y=1   v7    v8    v9    v5    v6
    // y=2   v2    v3    v4    v0    v1
    // y=3  v22   v23   v24   v20   v21
    // y=4  v17  v18    v19   v15   v16

    //- Theta Calculations -//
    eor3 v25.16B, v12.16B, v7.16B,  v2.16B
    eor3 v25.16B, v25.16B, v22.16B, v17.16B
    eor3 v26.16B, v13.16B, v8.16B,  v3.16B
    eor3 v26.16B, v26.16B, v23.16B, v18.16B
    eor3 v27.16B, v14.16B, v9.16B,  v4.16B
    eor3 v27.16B, v27.16B, v24.16B, v19.16B
    eor3 v28.16B, v10.16B, v5.16B,  v0.16B
    eor3 v28.16B, v28.16B, v20.16B, v15.16B
    eor3 v29.16B, v11.16B, v6.16B,  v1.16B
    eor3 v29.16B, v29.16B, v21.16B, v16.16B

    rax1 v30.2D, v29.2D, v26.2D
    rax1 v31.2D, v27.2D, v29.2D
    rax1 v29.2D, v25.2D, v27.2D
    rax1 v27.2D, v28.2D, v25.2D
    rax1 v25.2D, v26.2D, v28.2D

    //- Phi\rho Stage -//
    eor v12.8B, v12.8B, v30.8B
    xar v26.2D, v21.2D, v27.2D, #56
    xar v21.2D, v15.2D, v31.2D, #8
    xar v15.2D, v22.2D, v30.2D, #23
    xar v22.2D, v11.2D, v27.2D, #37
    xar v11.2D, v16.2D, v27.2D, #50
    xar v16.2D, v18.2D, v29.2D, #62
    xar v18.2D, v5.2D,  v31.2D, #9
    xar v5.2D,  v23.2D, v29.2D, #19
    xar v23.2D, v7.2D,  v30.2D, #28
    xar v7.2D,  v10.2D, v31.2D, #36
    xar v10.2D, v20.2D, v31.2D, #43
    xar v20.2D, v24.2D, v25.2D, #49
    xar v24.2D, v3.2D,  v29.2D, #54
    xar v3.2D,  v9.2D,  v25.2D, #58
    xar v9.2D,  v2.2D,  v30.2D, #61
    xar v2.2D,  v13.2D, v29.2D, #63
    xar v13.2D, v8.2D,  v29.2D, #20
    xar v8.2D,  v6.2D,  v27.2D, #44
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
xar v6.2D,  v19.2D, v25.2D, #3
xar v19.2D, v1.2D,  v27.2D, #25
xar v1.2D,  v17.2D, v30.2D, #46
xar v17.2D, v14.2D, v25.2D, #2
xar v14.2D, v4.2D,  v25.2D, #21
xar v4.2D,  v0.2D,  v31.2D, #39

// XAR Output:
//
// v12  v2 v17  v7 v22
// v23 v13  v3 v18  v8
//  v9 v24 v14  v4 v19
// v15  v5 v20 v10 v26
//  v1 v16  v6 v21 v11
//
// temp: v0, v25, v27, v28, v29, v30, v31

// Phi Output:
//
// v12 v13 v14 v10 v11
//  v7  v8  v9  v5  v6
//  v2  v3  v4 v26  v1
// v22 v23 v24 v20 v21
// v17 v18 v19 v15 v16

//- Chi transformations -//
bcax v31.16B, v26.16B, v2.16B,  v1.16B
bcax v27.16B, v1.16B,  v3.16B,  v2.16B
bcax v28.16B, v2.16B,  v4.16B,  v3.16B
bcax v29.16B, v3.16B,  v26.16B, v4.16B
bcax v30.16B, v4.16B,  v1.16B,  v26.16B
bcax v0.16B,  v5.16B,  v7.16B,  v6.16B
bcax v1.16B,  v6.16B,  v8.16B,  v7.16B
bcax v2.16B,  v7.16B,  v9.16B,  v8.16B
bcax v3.16B,  v8.16B,  v5.16B,  v9.16B
bcax v4.16B,  v9.16B,  v6.16B,  v5.16B
bcax v5.16B,  v10.16B, v12.16B, v11.16B
bcax v6.16B,  v11.16B, v13.16B, v12.16B
bcax v7.16B,  v12.16B, v14.16B, v13.16B
bcax v8.16B,  v13.16B, v10.16B, v14.16B
bcax v9.16B,  v14.16B, v11.16B, v10.16B
bcax v10.16B, v15.16B, v17.16B, v16.16B
bcax v11.16B, v16.16B, v18.16B, v17.16B
bcax v12.16B, v17.16B, v19.16B, v18.16B
bcax v13.16B, v18.16B, v15.16B, v19.16B
bcax v14.16B, v19.16B, v16.16B, v15.16B
bcax v15.16B, v20.16B, v22.16B, v21.16B
bcax v16.16B, v21.16B, v23.16B, v22.16B
bcax v17.16B, v22.16B, v24.16B, v23.16B
bcax v18.16B, v23.16B, v20.16B, v24.16B
bcax v19.16B, v24.16B, v21.16B, v20.16B

// Output State from Chi:
//
//      x=0   x=1   x=2   x=3   x=4
// y=0   v7    v8    v9    v5    v6
// y=1   v2    v3    v4    v0    v1
// y=2  v28   v29   v30   v31   v27
// y=3  v17   v18   v19   v15   v16
// y=4  v12   v13   v14   v10   v11
```

K10.2.3 Use of the SM3 instructions

These instructions are implemented when FEAT_SM3 is implemented.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following code sequence shows the use of the SM3 instructions to generate a SM3 hash.

```
.macro MessageExpand VA, VB, VC, VD, VOUT
 EXT \VOUT().16B, \VB().16B, \VC().16B, #12
 SM3PARTW1 \VOUT().4S, \VA().4S, \VD().4S
 EXT V17.16B, \VA().16B, \VB().16B, #12
 EXT V18.16B, \VC().16B, \VD().16B, #8
 SM3PARTW2 \VOUT().4S, V18.4S, V17.4S
 .endm

.macro HashPt1 VA, VB, Number SM3SS1 V23.4S, V20.4S, V22.4S, V19.4S
 EOR V21.16B, \VA().16B, \VB().16B
 SM3TT1a V20.4S, V23.4S, V21.S[\Number]
 SM3TT2a V19.4S, V23.4S, \VA().S[\Number]
 SHL V24.4S, V22.4S, #1
 SRI V24.4S, V22.4S, #31
 MOV V22.16B, V24.16B
 .endm

.macro HashPt2 VA, VB, Number SM3SS1 V23.4S, V20.4S, V25.4S, V19.4S
 EOR V21.16B, \VA().16B, \VB().16B
 SM3TT1b V20.4S, V23.4S, V21.S[\Number]
 SM3TT2b V19.4S, V23.4S, \VA().S[\Number]
 SHL V26.4S, V25.4S, #1
 SRI V26.4S, V25.4S, #31
 MOV V25.16B, V26.16B
 .endm


// V0-V3 holds the initial message
// V19 holds EFGH which is the lower half of the input hash
// V20 holds ABCD which is the upper half of the input hash
// V21 = current VPrime
// V22 holds T in bits[127:96] = 0x79cc4519
// V25 holds second value of T in bits[127:96] = 0x9d8a7a87<31:0>;

MessageExpand V0, V1, V2, V3, V4
MessageExpand V1, V2, V3, V4, V5
MessageExpand V2, V3, V4, V5, V6
MessageExpand V3, V4, V5, V6, V7
MessageExpand V4, V5, V6, V7, V8
MessageExpand V5, V6, V7, V8, V9
MessageExpand V6, V7, V8, V9, V10
MessageExpand V7, V8, V9, V10, V11
MessageExpand V8, V9, V10, V11, V12
MessageExpand V9, V10, V11, V12, V13
MessageExpand V10, V11, V12, V13, V14
MessageExpand V11, V12, V13, V14, V15
MessageExpand V12, V13, V14, V15, V16


MOV V29.16B, V19.16B
MOV V30.16B, V20.16B

HashPt1 V0,V1, 0
HashPt1 V0,V1, 1
HashPt1 V0,V1, 2
HashPt1 V0,V1, 3
HashPt1 V1,V2, 0
HashPt1 V1,V2, 1
HashPt1 V1,V2, 2
HashPt1 V1,V2, 3
HashPt1 V2,V3, 0
HashPt1 V2,V3, 1
HashPt1 V2,V3, 2
HashPt1 V2,V3, 3
HashPt1 V3,V4, 0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
HashPt1 V3,V4, 1
HashPt1 V3,V4, 2
HashPt1 V3,V4, 3

HashPt2 V4,V5, 0
HashPt2 V4,V5, 1
HashPt2 V4,V5, 2
HashPt2 V4,V5, 3
HashPt2 V5,V6, 0
HashPt2 V5,V6, 1
HashPt2 V5,V6, 2
HashPt2 V5,V6, 3
HashPt2 V6,V7, 0
HashPt2 V6,V7, 1
HashPt2 V6,V7, 2
HashPt2 V6,V7, 3
HashPt2 V7,V8, 0
HashPt2 V7,V8, 1
HashPt2 V7,V8, 2
HashPt2 V7,V8, 3
HashPt2 V8,V9, 0
HashPt2 V8,V9, 1
HashPt2 V8,V9, 2
HashPt2 V8,V9, 3
HashPt2 V9,V10, 0
HashPt2 V9,V10, 1
HashPt2 V9,V10, 2
HashPt2 V9,V10, 3
HashPt2 V10,V11, 0
HashPt2 V10,V11, 1
HashPt2 V10,V11, 2
HashPt2 V10,V11, 3
HashPt2 V11,V12, 0
HashPt2 V11,V12, 1
HashPt2 V11,V12, 2
HashPt2 V11,V12, 3
HashPt2 V12,V13, 0
HashPt2 V12,V13, 1
HashPt2 V12,V13, 2
HashPt2 V12,V13, 3
HashPt2 V13,V14, 0
HashPt2 V13,V14, 1
HashPt2 V13,V14, 2
HashPt2 V13,V14, 3
HashPt2 V14,V15, 0
HashPt2 V14,V15, 1
HashPt2 V14,V15, 2
HashPt2 V14,V15, 3
HashPt2 V15,V16, 0
HashPt2 V15,V16, 1
HashPt2 V15,V16, 2
HashPt2 V15,V16, 3


EOR V19.16B, V29.16B, V19.16B
EOR V20.16B, V30.16B, V20.16B

// V19 holds EFGH which is the lower half of the output hash
// V20 holds ABCD which is the upper half of the output hash
```

K10.2.4 Use of the SM4 instructions

These instructions are implemented when FEAT_SM4 is implemented.

The following code sequences show the use of the SM4 instructions to perform SM4 encryption and decryption:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encryption

```
// Encryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>;
// V8 contains the Key
// V2 contains the data to be encrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8afb6bd8c939aa170777e85;
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f969d6c737a8150575e65;
// V20 contains: 0x141b2229f8ff060ddce3eaf1c0c7ced5;
// V21 contains: 0x848b9299686f767d4c535a6130373e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a0a7aeb5;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

SM4E V2.4S, V8.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V15.4S

// need to reverse the order of the words at the end of the operation
REV64 v2.4S, v2.4S
EXT V2.16B, V2.16B, V2.16B, #8
```

Decryption

```
// Decryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>;
// V8 contains the Key
// V2 contains the data to be decrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8afb6bd8c939aa170777e85;
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f969d6c737a8150575e65;
// V20 contains: 0x141b2229f8ff060ddce3eaf1c0c7ced5;
// V21 contains: 0x848b9299686f767d4c535a6130373e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a0a7aeb5;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

// need to reverse the order of the keys to do a decryption:

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

REV64 V8.4S, V8.4S
EXT V8.16B, V8.16B, V8.16B, #8
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
REV64 V9.4S, V9.4S
EXT V9.16B, V9.16B, V9.16B, #8
REV64 V10.4S, V10.4S
EXT V10.16B, V10.16B, V10.16B, #8
REV64 V11.4S, V11.4S
EXT V11.16B, V11.16B, V11.16B, #8
REV64 V12.4S, V12.4S
EXT V12.16B, V12.16B, V12.16B, #8
REV64 V13.4S, V13.4S
EXT V13.16B, V13.16B, V13.16B, #8
REV64 V14.4S, V14.4S
EXT V14.16B, V14.16B, V14.16B, #8
REV64 V15.4S, V15.4S
EXT V15.16B, V15.16B, V15.16B, #8

SM4E V2.4S, V15.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V8.4S

// final reversal of the order of the words in the result:
REV64 V2.4S, V2.4S
EXT V2.16B, V2.16B, V2.16B, #8
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.3 SVE and SVE2 Programming Examples

The DAI0548 SVE/SVE2 Programming Examples document illustrates Arm Scalable Vector Extension and provides programming guidance and the selection of code examples.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.4 Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) hint instructions

This section provides example uses of the `STSHH` and `PRFM IR` (immediate) hint instructions that FEAT_PCDPHINT provides. It is not part of the Arm architecture definition.

K10.4.1 Example ticket lock use of the STSHH hint instruction

FEAT_PCDPHINT provides a `STSHH` hint instruction, that PE software can use to hint to the memory system that if the next instruction the PE executes generates an explicit write effect, it will be to a location being observed, and it might be beneficial if the memory system takes some action to propagate the updated value to the observers with minimal latency.

One example use of `STSHH` is in ticket lock code, where a thread currently holding the lock can use `STSHH` before it releases the lock, as shown in the following example. In the example:

- There are two counters in the system:
  - Ticket assignment. A counter holding the number of the next ticket in the queue.
  - Current ticket number. Holds the number currently being served.
- In the piece of code:
  - In the Consuming portion, the thread has already acquired a ticket from the ticket assignment counter and is waiting for its number to match the current ticket number. The current ticket number increments by one each time a thread holding the lock releases the lock.
  - In the Producing portion, the thread has already acquired the lock and finished the task it needed to do while it held the lock, and is now releasing the lock, by:
    1. Incrementing the current ticket number.
    2. Sending a `STSHH` hint that it is about to perform an explicit write effect, and that it might be beneficial if the memory system takes some action to propagate the updated current ticket number to the other observers as soon as possible.
    3. Storing the updated current ticket number back to the memory system. This releases the lock, allowing the next thread in ticket order to acquire the lock. The observing threads waiting for the current ticket number to match their assigned ticket number might be polling the current ticket counter, or might have executed a `PRFM IR` (immediate) targeting the current ticket counter.

Example K10-1

```
; Consuming portion

SPIN:   LDAR W0, <lock>
        CMP <assigned ticket_value>, W0
        B.NE SPIN
; ...

; Producing portion
        LDR W0, <lock>
        ADD W0, W0, #0x1
        STSHH KEEP
        STLR W0, <lock>
```

In the example, `STSHH` uses the KEEP policy because if no other observer acquires the lock afterwards, and our thread needs to reenter the task it was doing while it held the lock, it can quickly do so.

K10.4.2 Example data sharing use of the STSHH and PRFM IR (immediate) instructions

The following is a common pattern for data sharing between two threads:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-2

```
P0:             P1:
STR <payload>   SEVL
STLR <flag>     LOOP:
                WFE
                LDAXR <flag>
                CMP <flag>, <expected value>    ; Test to see if flag has been updated.
                B.NE LOOP                       ; Branch back to WFE and wait if flag not updated.
                LDR <payload>
                CLREX
```

With `STSHH` and `PRFM IR` (immediate), the following can be used instead:

Example K10-3

```
P0:             P1:
STSHH STRM      SEVL
STR <payload>   LOOP:
STSHH STRM      WFE
STLR <flag>     LDAXR <flag>
                CMP <flag>, <expected value>    ; test to see if flag has been updated.
                B.EQ READY                      ; branch over the PRFM if the flag has been set.
                PRFM IR <payload>
                B.NE LOOP                       ; branch back to WFE and wait if flag not updated.
                READY:
                LDR <payload>
                CLREX
```

In Example K10-3, `STSHH` uses the STRM policy because the producing thread does not read from the locations, and there must be a consumption by another thread before the producing thread updates the locations again.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5 Use of the Embedded Trace and Trace Buffer Extensions

This appendix describes a simple trace decompressor. It contains the following sections:

- Trace analyzer.
- ETE common pseudocode
- ETE programming.
- Trace examples.
- Differences between ETM and ETE.
- Context switching.
- Controlling generation of trace buffer management events.

K10.5.1 Trace analyzer

K10.5.1.1 Using Trace Info elements to start trace analysis

IWDKLF After the trace analyzer has located an Alignment Synchronization packet and synchronized with the trace byte stream, it must search for the following elements to begin to analyze the trace byte stream:

1. A *Trace Info element*.
2. A *Context element* and a *Target Address element*.

IFCGKX The trace unit might not generate a *Context element* and *Target Address element* immediately after it generates a *Trace Info element*.

IBWQGD If a *Cancel element* cancels a *Trace Info element* then the trace analyzer can still use the information from the discarded *Trace Info element*, but if the *Context element* and *Target Address element* are also discarded, then the trace analyzer must wait for the trace unit to generate a new *Context element* and *Target Address element*.

K10.5.1.2 Encountering Trace Info elements after trace analysis has started

IHTFNG The trace unit might generate *Trace Info elements* periodically, as a result of trace protocol synchronization requests. This is useful if trace is stored in a circular buffer, because it provides multiple points where trace analysis can start.

After a trace analyzer observes the first *Trace Info element*, it can ignore subsequent *Trace Info elements* in the same trace session because the static trace programming cannot change and the speculation depth is updated by other element types during the trace session.

K10.5.1.3 Decompression information

IRGSRY To decompress a trace byte stream, the trace analyzer requires a number of values which differ between implementations. All the information required by the decompressor to analyze the trace byte stream is provided in the TRCIDR0 to TRCIDR13 registers.

Table K10-1 lists the static variables required by the decompressing stages and the fields that provide this information.

Table K10-1 Static trace unit information

Variable	ID Field	Stage
Commit Mode	TRCIDR0.COMMOPT	1
Virtual context identifier size	TRCIDR2.VMIDSIZE	1
Maximum speculation depth	TRCIDR8.MAXSPEC	1
WFx Instructions	TRCIDR2.WFXMODE	3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2 Stage 1, parsing the byte stream

I_{ZLYTH} The first stage of analyzing the trace is to convert from the bits of the trace byte stream to the elements that are encoded in that trace byte stream.

The ETE architecture enables a trace unit to use techniques that can reduce the trace bandwidth and trace storage requirements. Some of these techniques require the trace analyzer to retain some information between packets so that it can successfully analyze future packets.

K10.5.2.1 Retained state

R_{ZPQMY} The trace analyzer maintains an independent copy of the address history buffer of the last three *Target Address elements*.

I_{HDKHN} The address history buffer in the trace analyzer is required to reconstruct the *Target Address elements* from the trace byte stream.

R_{JBGCR} The trace analyzer must maintain the current speculation depth of the parsed trace byte stream.

R_{HDJJV} The trace analyzer must have the maximum speculation depth supported by the trace unit.

R_{RZNPD} The trace analyzer maintains a copy of the last *Timestamp element* value decompressed.

I_{NPCCX} The last *Timestamp element* value in the trace analyzer is required to reconstruct the full timestamp value for a Timestamp Packet.

R_{ZPNYS} The trace analyzer must maintain a copy of the context:

- Context identifier.
- Virtual context identifier.
- Execution state.
- Exception level
- Security state

R_{MWFSN} The trace analyzer must maintain a copy of the cycle count threshold.

K10.5.2.1.1 InstructionParserState()

```
// InstructionParserState
// ======================
// State of the instruction parser.

type InstructionParserState is (
    bits(64) timestamp,          // The most recently broadcast timestamp value.
    // The Address History Buffer.
    array [0..2] of AddressHistoryBufferEntry address_history_buffer,

    // Context parameters.
    bits(32) context_id,         // Most recently broadcast Context ID.
    bits(32) vmid,               // Most recently broadcast VMID.
    bits(2) exception_level,     // Most recently broadcast Exception level.
    SecurityLevel security,      // Most recently broadcast Security state.
    boolean sixty_four_bit,      // Most recently broadcast AArch state
                                 // (32 or 64?).

    // Speculation
    integer current_spec_depth,  // The current speculation depth.
    boolean T,                   // The current transactional state.

    // Trace Session static
    integer cc_threshold,        // Cycle count threshold value.

    // Static state
    integer max_spec_depth,      // The maximum speculation depth.
    boolean commit_mode,         // Commit mode.
    boolean comm_trans           // How transactions traced.
)
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.2 Parsing

IPXZJK The first stage of the decompressor is to convert from the trace byte stream to trace element stream. The trace byte stream can start at the an Alignment Sync packet boundary.

K10.5.2.2.1 Parse_Trace()

```
// Parse_Trace()
// =============
// Parses a trace bytestream generated by an ETE trace unit.

Parse_Trace(bits(S) stream)
    repeat
        header = ReadAndConsume(8, stream);
        case header of
            when '00000000'  Parse_ExtensionPacket(header, stream);
            when '00000001'  Parse_TraceInfoPacket(header, stream);
            when '0000001x'  Parse_TimestampPacket(header, stream);
            when '00000100'  TraceOnPacket();
            when '00000110'  Parse_ExceptionPacket(header, stream);
            when '00001010'  TransactionStartPacket();
            when '00001011'  TransactionCommitPacket();
            when '0000110x'  Parse_CycleCountPackets(header, stream);
            when '0000111x'  Parse_CycleCountPackets(header, stream);
            when '0001xxxx'  Parse_CycleCountPackets(header, stream);
            when '00101101'  Parse_CommitPacket(header, stream);
            when '0010111x'  Parse_CancelPackets(header, stream);
            when '001100xx'  Parse_MispredictPacket(header, stream);
            when '001101xx'  Parse_CancelPackets(header, stream);
            when '00111xxx'  Parse_CancelPackets(header, stream);
            when '01110000'  // Ignore packet
            when '0111xxxx'  Parse_EventTracingPacket(header, stream);
            when '1000000x'  Parse_ContextPacket(header, stream);
            when '1000001x'  Parse_AddressWithContextPacket(header, stream);
            when '10000101'  Parse_AddressWithContextPacket(header, stream);
            when '10000110'  Parse_AddressWithContextPacket(header, stream);
            when '1001000x'  Parse_TargetAddressPacket(header, stream);
            when '10010010'  Parse_TargetAddressPacket(header, stream);
            when '100101xx'  Parse_TargetAddressPacket(header, stream);
            when '1001100x'  Parse_TargetAddressPacket(header, stream);
            when '1001101x'  Parse_TargetAddressPacket(header, stream);
            when '10011101'  Parse_TargetAddressPacket(header, stream);
            when '10011110'  Parse_TargetAddressPacket(header, stream);
            when '1010xxxx'  Parse_QPacket(header, stream);
            when '101100xx'  Parse_SourceAddressPacket(header,stream);
            when '1011010x'  Parse_SourceAddressPacket(header,stream);
            when '1011011x'  Parse_SourceAddressPacket(header,stream);
            when '1011100x'  Parse_SourceAddressPacket(header,stream);
            when '11xxxxxx'  Parse_AtomPackets(header, stream);
            otherwise        ReservedEncoding();
    until EndOfStream(stream);

    return;
```

K10.5.2.3 Alignment Sync packet

IVJHHN The Alignment Sync packet is a unique sequence of bits that identifies the boundary of another packet. The unique sequence is a header byte, `0b00000000`, followed by a minimum of ten payload bytes of `0b00000000` and one final payload byte of `0b10000000`.

K10.5.2.3.1 Parse_ExtensionPacket()

```
// Parse_ExtensionPacket()
// =======================
// Parses Alignment Synchronization, Discard and Overflow packets.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Parse_ExtensionPacket(bits(8) header, bits(S) stream)
    extension = ReadAndConsume(8, stream);
    case extension of
        when '00000000'                 // A-sync
            while extension == '00000000' do
                extension = ReadAndConsume(8, stream);
            if extension != '10000000' then
                ReservedEncoding();

        when '00000011'                 // Discard
            DiscardPacket();

        when '00000101'                 // Overflow
            OverflowPacket();

        otherwise
            ReservedEncoding();

    return;
```

K10.5.2.4 Discard

R_{RVFVY} The current speculation depth must be reset to 0.

K10.5.2.4.1 DiscardPacket()

```
// DiscardPacket()
// ===============
// Processes a Discard packet.

DiscardPacket()
    Emit(DiscardElement());
    if DSTATE.IA.T then
       Emit(TransactionFailureElement());

    DSTATE.IA.current_spec_depth = 0;
    DSTATE.IA.T = FALSE;

    return;
```

K10.5.2.5 Overflow

K10.5.2.5.1 OverflowPacket()

```
// OverflowPacket()
// ================
// Processes an Overflow packet.

OverflowPacket()
    Emit(DiscardElement());
    Emit(OverflowElement());

    if DSTATE.IA.T then
        Emit(TransactionFailureElement());

    DSTATE.IA.T = FALSE;
    DSTATE.IA.current_spec_depth = 0;

    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.6 Trace Info

I_{JMXMP} A Trace Info packet indicates where the compression algorithms used by the trace unit have been set to a known architectural state. As the architectural state of the compression algorithms is known a trace analyzer can start decompression of the trace byte stream at this point.

If the trace unit exposes some trace speculation to the trace analyzer then the trace info packet indicates the trace speculation depth at this point in the trace element stream.

If the trace analyzer starts analysis where the trace speculation depth is nonzero then the analyzer should ignore speculation depth of *Commit elements*.

R_{PKZKZ} A Trace Info Packet sets all entries of the address history buffer to have an address of `0x0` and to sub_isa of IS0.

R_{LPYLR} The current_spec_depth is set to the speculation depth indicated in the trace info element.

K10.5.2.6.1 Parse_TraceInfoPacket()

```
// Parse_TraceInfoPacket()
// =======================
// Parses a Trace Info packet.

Parse_TraceInfoPacket(bits(8) header, bits(S) stream)
    bits(8) INFO = Zeros();
    bits(96) SPEC = Zeros();
    bits(96) CYCT = Zeros();

    bits(8) PLCTL = ReadAndConsume(8, stream);

    //  Extract the INFO section if present
    if PLCTL<0> == '1' then
        INFO = ReadAndConsume(8, stream);

    //  Extract the SPEC section if present
    if PLCTL<2> == '1' then
        SPEC = ULEB128(stream);

    //  Extract the CYCT section if present
    if PLCTL<3> == '1' then
        CYCT = ULEB128(stream);

    TraceInfoPacket(PLCTL, INFO, SPEC, CYCT);

    return;
```

K10.5.2.6.2 TraceInfoPacket()

```
// TraceInfoPacket()
// =================
// Processes a Trace Info packet.

TraceInfoPacket(bits(8) PLCTL,
                bits(8) INFO,
                bits(SN) SPEC,
                bits(CN) CYCT)
    DSTATE.IA.timestamp = Zeros();
    DSTATE.IA.context_id = Zeros();
    DSTATE.IA.vmid = Zeros();
    DSTATE.IA.exception_level = EL0;
    DSTATE.IA.security = SecurityLevel_SECURE;
    DSTATE.IA.sixty_four_bit = FALSE;
    AddressHistoryBuffer.Reset();

    cc_threshold = if INFO<0> == '1' then UInt(CYCT) else 0;
    DSTATE.IA.current_spec_depth = UInt(SPEC);

    Emit(TraceInfoElement(INFO<0> == '1',     // cc_enabled
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                cc_threshold,
                                DSTATE.IA.current_spec_depth,
                                INFO<6> == '1'));

    return;
```

K10.5.2.7 Trace On

IGHTQJ

The Trace On packet indicates that there has been a discontinuity in the instruction trace element stream. It is output whenever a gap occurs, after the gap occurs. This means that a Trace On packet is output:

- When trace generation becomes operative, after the first A-Sync and Trace Info packets but before any packet types that indicate any *P0 elements*.
- After a trace unit buffer overflow. Again, the Trace On packet is output after the A-Sync and Trace Info packets but before any packet types that indicate any *P0 elements*.
- After gaps caused by filtering. For example, if filtering is applied to the generation of the trace element stream, so that the trace unit only generates trace for a particular program code sequence, the trace unit might spend much of its time in an inactive state, only generating trace periodically. In this case, a Trace On packet is output after each discontinuity in the trace element stream. The Trace On packet must be output before any packet types that indicate any *P0 elements*.

K10.5.2.7.1 TraceOnPacket()

```
// TraceOnPacket()
// ===============
// Processes a Trace On packet.

TraceOnPacket()
    Emit(TraceOnElement());
    return;
```

K10.5.2.8 Speculation

RPSHDJ

The *Commit element* must modify the current speculation depth.

K10.5.2.8.1 Parse_CommitPacket()

```
// Parse_CommitPacket()
// ====================
// Parses a Commit packet.

Parse_CommitPacket(bits(8) header, bits(S) stream)
    bits(32) COMMIT = ULEB128(stream);
    CommitPacket(COMMIT);
    return;
```

K10.5.2.8.2 CommitPacket()

```
// CommitPacket()
// ==============
// Processes a Commit packet.

CommitPacket(bits(N) COMMIT)
    Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
    return;
```

RNQHYR

The *Cancel element* must modify the current speculation depth.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.8.3 Parse_CancelPackets()

```
// Parse_CancelFormatPackets()
// ===========================
// Parses all the various Cancel packets.

Parse_CancelPackets(bits(8) header, bits(S) stream)
    case header of
        when '0010111x'         // Cancel Format 1
            M = header<0>;
            bits(32) CANCEL = ULEB128(stream);
            CancelFormat1Packet(M, CANCEL);

        when '001101xx'         // Cancel Format 2
            A2 = header<1:0>;
            CancelFormat2Packet(A2);

        when '00111xxx'         // Cancel Format 3
            A = header<0>;
            CC = header<2:1>;
            CancelFormat3Packet(CC, A);

    return;
```

K10.5.2.8.4 CancelFormat1Packet()

```
// CancelFormat1Packet()
// =====================
// Processes a Cancel packet, format 1.

CancelFormat1Packet(bit M, bits(N) CANCEL)
    count = UInt(CANCEL);
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    if M == '1' then
        Emit(MispredictElement());

    return;
```

K10.5.2.8.5 CancelFormat2Packet()

```
// CancelFormat2Packet()
// =====================
// Processes a Cancel packet, format 2.

CancelFormat2Packet(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_E);
        when '10'
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_N);

    count = 1;
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    Emit(MispredictElement());

    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.8.6 CancelFormat3Packet()

```
// CancelFormat3Packet()
// =====================
// Processes a Cancel packet, format 3.

CancelFormat3Packet(bits(2) CC, bit A)
    if A == '1' then
        HandleAtom(Atom_E);
    count = UInt(CC) + 2;
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    Emit(MispredictElement());

    return;
```

K10.5.2.9 Mispredict

K10.5.2.9.1 Parse_MispredictPacket()

```
// Parse_MispredictPacket()
// ========================
// Parses a Mispredict packet.

Parse_MispredictPacket(bits(8) header, bits(S) stream)
    A = header<1:0>;
    MispredictPacket(A);
    return;
```

K10.5.2.9.2 MispredictPacket()

```
// MispredictPacket()
// ==================
// Processes a Mispredict packet.

MispredictPacket(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_E);
        when '10'
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_N);
        otherwise

    Emit(MispredictElement());
    return;
```

K10.5.2.10 Atom packets

R_{RVVFW} Each *Atom element* decoded from an Atom packet must increment the current speculation depth of this stage of the decompressor.

K10.5.2.10.1 Parse_AtomPacket()

```
// Parse_AtomPackets()
// ===================
// Parses all the various Atom packets.

Parse_AtomPackets(bits(8) header, bits(S) stream)
    case header of
        when '1111011x'                 // Atom Format 1
            bit A = header<0>;
            AtomFormat1Packet(A);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '110110xx'                 // Atom Format 2
            bits(2) A = header<1:0>;
            AtomFormat2Packet(A);

        when '11111xxx'                 // Atom Format 3
            bits(3) A = header<2:0>;
            AtomFormat3Packet(A);

        when '110111xx'                 // Atom Format 4
            bits(2) A = header<1:0>;
            AtomFormat4Packet(A);

        when '11110101'                 // Atom Format 5.1
            AtomFormat5_1Packet();

        when '11010101', '11010110', '11010111' // Atom Format 5.2
            bits(2) A = header<1:0>;
            AtomFormat5_2Packet(A);

        when '11xxxxxx'                 // Atom Format 6
            bit A = header<5>;
            bits(5) COUNT = header<4:0>;
            AtomFormat6Packet(A, COUNT);

    return;
```

K10.5.2.10.2 AtomFormat1Packet()

```
// AtomFormat1Packet()
// ===================
// Processes an Atom packet, format 1.

AtomFormat1Packet(bit A)
    if A == '1' then
        HandleAtom(Atom_E);
    else
        HandleAtom(Atom_N);

    return;
```

K10.5.2.10.3 AtomFormat2Packet()

```
// AtomFormat2Packet()
// ===================
// Processes an Atom packet, format 2.

AtomFormat2Packet(bits(2) A)
    for I = 0 to 1
        if A<I> == '1' then
            HandleAtom(Atom_E);
        else
            HandleAtom(Atom_N);

    return;
```

K10.5.2.10.4 AtomFormat3Packet()

```
// AtomFormat3Packet()
// ===================
// Processes an Atom packet, format 3.

AtomFormat3Packet(bits(3) A)
    for I = 0 to 2
        if A<I> == '1' then
            HandleAtom(Atom_E);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            HandleAtom(Atom_N);

    return;
```

K10.5.2.10.5 AtomFormat4Packet()

```
// AtomFormat4Packet()
// ===================
// Processes an Atom packet, format 4.

AtomFormat4Packet(bits(2) A)
    case A of
        when '00'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '01'
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
        when '10'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);

    return;
```

K10.5.2.10.6 AtomFormat5_1Packet()

```
// AtomFormat5_1Packet()
// =====================
// Processes an Atom packet, format 5.1.

AtomFormat5_1Packet()
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);

    return;
```

K10.5.2.10.7 AtomFormat5_2Packet()

```
// AtomFormat5_2Packet()
// =====================
// Processes an Atom packet, format 5.2.

AtomFormat5_2Packet(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '10'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
        when '11'
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);

    return;
```

K10.5.2.10.8 AtomFormat6Packet()

```
// AtomFormat6Packet()
// ===================
// Processes an Atom packet, format 6.

AtomFormat6Packet(bit A, bits(5) COUNT)
    for I = 0 to UInt(COUNT) + 2
        HandleAtom(Atom_E);
    if A == '1' then
        HandleAtom(Atom_N);
    else
        HandleAtom(Atom_E);

    return;
```

K10.5.2.11 Q packets

RRZFZW The *Q element* must increment the current speculation depth at this stage of the decompressor.

K10.5.2.11.1 Parse_QPacket()

```
// Parse_QPacket()
// ===============
// Parses a Q packet.

Parse_QPacket(bits(8) header, bits(S) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    bits(32) count;

    TYPE = header<3:0>;
    case TYPE of
        when '0000', '0001', '0010'
            ExactMatchBytes(header, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '0101'
            ShortAddressBytes(IS0, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '0110'
            ShortAddressBytes(IS1, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '1010', '1011'
            LongAddressBytes(header, stream);
            count = ULEB128(stream);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            QPacket(TYPE, entry, count);

        when '1100'
            count = ULEB128(stream);
            QPacket(TYPE, UNKNOWN_ADDRESS, count);

        when '1111'
            QPacket(TYPE, UNKNOWN_ADDRESS, UNKNOWN_COUNT);

        otherwise
            ReservedEncoding();

    return;
```

K10.5.2.11.2 QPacket()

```
// QPacket()
// =========
// Processes a Q packet.

QPacket(bits(4) TYPE, AddressHistoryBufferEntry A, bits(CN) COUNT)
    Emit(QElement(UInt(COUNT)));
    UpdateSpecDepth(1);

    // The decoding of the Address field is done by the AddressPacket function,
    // but this did not Emit the address element.
    if (TYPE != '11xx' && TYPE != '00xx') then
        Emit(TargetAddressElement(A));

    return;
```

K10.5.2.12 Source address packets

K10.5.2.12.1 Parse_SourceAddressPacket()

```
// Parse_SourceAddressPacket()
// ===========================
// Parses a Source Address packet.

Parse_SourceAddressPacket(bits(8) header, bits(S) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);

    case header of
        when '10111000'
            data1 = ReadAndConsume(8, stream);
            data2 = ReadAndConsume(8, stream);
            data3 = ReadAndConsume(48, stream);
            entry.sub_isa = IS0;
            entry.address<63:0> = data3:data2<6:0>:data1<6:0>:'00';
            AddressHistoryBuffer.Add(entry);
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10111001'
            data11 = ReadAndConsume(8, stream);
            data21 = ReadAndConsume(56, stream);
            a = entry;
            entry.sub_isa = IS1;
            entry.address<63:0> = data21:data11<6:0>:'0';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10110110'
            data12 = ReadAndConsume(8, stream);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            data22 = ReadAndConsume(8, stream);
            data32 = ReadAndConsume(16, stream);
            a = entry;
            entry.sub_isa = IS0;
            entry.address<31:0> = data32:data22<6:0>:data12<6:0>:'00';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10110111'
            data13 = ReadAndConsume(8, stream);
            data23 = ReadAndConsume(24, stream);
            a = entry;
            entry.sub_isa = IS1;
            entry.address<31:0> = data23:data13<6:0>:'0';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10110100'
            data14 = ReadAndConsume(8, stream);
            a = entry;
            entry.sub_isa = IS0;
            if data14<7> == '1' then
                data24 = ReadAndConsume(8, stream);
                entry.address<16:0> = data24<7:0>:data14<6:0>:'00';
            else
                entry.address<8:0> = data14<6:0>:'00';

            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10011101'
            data15 = ReadAndConsume(8, stream);
            a = entry;
            entry.sub_isa = IS0;
            if data15<7> == '1' then
                data25 = ReadAndConsume(8, stream);
                entry.address<15:0> = data25<7:0>:data15<6:0>:'0';
            else
                entry.address<7:0> = data15<6:0>:'0';

            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '101100xx'             // Exact match
            q = UInt(header<1:0>);
            entry = AddressHistoryBuffer.Get(q);
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

    return;
```

K10.5.2.13 Exceptions

R_{SRPFR} The *Exception element* must increment the current speculation depth at this stage of the decompressor.

K10.5.2.13.1 Parse_ExceptionPacket()

```
// Parse_ExceptionPacket()
// =======================
// Parses an exception packet.
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Parse_ExceptionPacket(bits(8) header, bits(S) stream)
    payload = ReadAndConsume(8, stream);
    bits(2) E;
    E<0> = payload<0>;
    E<1> = payload<6>;
    bits(5) TYPE = payload<5:1>;
    bits(8) AH;

    if E == '01' || E == '10' then
        // Treat the ADDRESS bytes as an Address packet, including
        // updating the address registers.
        AH = ReadAndConsume(8, stream);

        case AH of
            when '1001000x', '10010010'                // Exact Match.
                ExactMatchBytes(AH, stream);

            when '10010101'                            // Short Address IS0.
                ShortAddressBytes(IS0, stream);

            when '10010110'                            // Short Address IS1.
                ShortAddressBytes(IS1, stream);

            when '1001101x', '10011101', '10011110'    // Long Address.
                LongAddressBytes(AH, stream);

            when '1000001x', '10000101', '10000110'    // Long Address with
                LongAddressBytes(AH, stream);          // Context.
                ContextBytes(stream);

            when '01110000'                            // Unknown address
                UnknownAddressHistoryBuffer();

        ExceptionPacket(E, TYPE, AH);
    else
        ReservedEncoding();

    return;
```

K10.5.2.13.2 ExceptionPacket()

```
// ExceptionPacket()
// =================
// Processes an Exception packet.

ExceptionPacket(bits(2) E, bits(5) TYPE, bits(8) AH)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    case E of
        when '01'
            if TYPE == '11000' then
                Emit(TransactionFailureElement());
            else
                Emit(ExceptionElement(UInt(TYPE), entry.address));
        when '10'
            // The new context and address must now be Emitted.
            if AH<7:4> == '1000' then   // Long Address with Context
                Emit(ContextElement(DSTATE.IA.context_id,
                                    DSTATE.IA.vmid,
                                    DSTATE.IA.exception_level,
                                    DSTATE.IA.security,
                                    DSTATE.IA.sixty_four_bit));
            Emit(TargetAddressElement(entry));

            if TYPE == '00000' && DSTATE.IA.T then
                Emit(TransactionFailureElement());
                Emit(ExceptionElement(UInt(TYPE), entry.address));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            elsif TYPE == '11000' then
                Emit(TransactionFailureElement());
            else
                Emit(ExceptionElement(UInt(TYPE), entry.address));

        otherwise
            ReservedEncoding();

    UpdateSpecDepth(1);

    return;
```

K10.5.2.14 Address and context

K10.5.2.14.1 Parse_TargetAddressPacket()

```
// Parse_TargetAddressPacket()
// ===========================
// Parses a Target Address packet.

Parse_TargetAddressPacket(bits(8) header, bits(S) stream)
    case header of
        when '1001101x', '10011101', '10011110' // Long Address
            LongAddressBytes(header, stream);

        when '10010101'                         // Short Address IS0
            ShortAddressBytes(IS0, stream);

        when '10010110'                         // Short Address IS1
            ShortAddressBytes(IS1, stream);

        when '1001000x', '10010010'             // Exact match
            ExactMatchBytes(header, stream);

    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    Emit(TargetAddressElement(entry));
    return;
```

K10.5.2.14.2 Parse_LongAddressBytes()

```
// LongAddressBytes()
// ==================
// Reads and parses the long address form used in some packets.

LongAddressBytes(bits(8) header, bits(S) stream)
    case header<2:0> of
        when '010'      // 32-bit IS0
            data32_is0 = ReadAndConsume(16, stream);
            A8_2 = data32_is0<6:0>;  SBZ(data32_is0<7>);
            A15_9 = data32_is0<14:8>; SBZ(data32_is0<15>);
            bits(16) A31_16;
            A31_16 = POD(16, stream);
            LongAddressPacket(header, A31_16:A15_9:A8_2);

        when '011'      // 32-bit IS1
            data32_is1 = ReadAndConsume(8, stream);
            A7_1 = data32_is1<6:0>;  SBZ(data32_is1<7>);
            A31_8 = POD(24, stream);
            LongAddressPacket(header, A31_8:A7_1);

        when '101'      // 64-bit IS0
            data64_is0 = ReadAndConsume(16, stream);
            A8_2 = data64_is0<6:0>;
            SBZ(data64_is0<7>);
            A15_9 = data64_is0<14:8>;
            SBZ(data64_is0<15>);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            A63_16 = POD(48, stream);
            LongAddressPacket(header, A63_16:A15_9:A8_2);

        when '110'      // 64-bit IS1
            data64_is1 = ReadAndConsume(8, stream);
            A7_1 = data64_is1<6:0>;  SBZ(data64_is1<7>);
            A63_8 = POD(56, stream);
            LongAddressPacket(header, A63_8:A7_1);

    return;
```

K10.5.2.14.3 LongAddressPacket()

```
// LongAddressPacket()
// ===================
// Parses the long form address used in some packets.

LongAddressPacket(bits(8) header, bits(AN) A)

    a = AddressHistoryBuffer.Get(0);

    // Called from a variety of packet types, so only look at bits <2:0> of
    // the header.
    case header<2:0> of
        when '010'              // 32-bit address, IS0
            assert(AN == 30);
            a.sub_isa = IS0;
            // address<63:32> unchanged
            a.address<31:2> = A<29:0>;
            a.address<1:0> = '00';

        when '011'              // 32-bit address, IS1
            assert(AN == 31);
            a.sub_isa = IS1;
            // address<63:32> unchanged
            a.address<31:1> = A<30:0>;
            a.address<0> = '0';

        when '101'              // 64-bit address, IS0
            assert(AN == 62);
            a.sub_isa = IS0;
            a.address<63:2> = A<61:0>;
            a.address<1:0> = '00';

        when '110'              // 64-bit address, IS1
            assert(AN == 63);
            a.sub_isa = IS1;
            a.address<63:1> = A<62:0>;
            a.address<0> = '0';

    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;
```

K10.5.2.14.4 Parse_ShortAddressBytes()

```
// ShortAddressBytes()
// ===================
// Reads and parses the short form address used in some packets.

ShortAddressBytes(SubISA sub_isa, bits(S) stream)
    bits(15) data;

    if sub_isa == IS0 then
        data = BitReplacement(stream,
                              AddressHistoryBuffer.Get(0).address<16:2>);
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        data = BitReplacement(stream,
                              AddressHistoryBuffer.Get(0).address<15:1>);

    ShortAddressPacket(sub_isa, data);
    return;
```

K10.5.2.14.5 ShortAddressPacket()

```
// ShortAddressPacket()
// ====================
// Parses the short form address used in some packets.

ShortAddressPacket(SubISA sub_isa, bits(AN) A)
    a = AddressHistoryBuffer.Get(0);

    assert (AN == 7 || AN == 15);

    case sub_isa of
        when IS0                                // IS0
            a.sub_isa = IS0;
            // address<63:AN+2> unchanged
            a.address<AN+1:0> = A:'00';

        when IS1                                // IS1
            a.sub_isa = IS1;
            // address<63:AN+1> unchanged
            a.address<AN:0> = A:'0';

    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;
```

K10.5.2.14.6 Parse_ExactMatchBytes()

```
// ExactMatchBytes()
// =================
// Reads and parses an exact address match used in some packets.

ExactMatchBytes(bits(8) header, bits(S) stream)
    QE = header<1:0>;
    ExactMatchPacket(QE);
    return;
```

K10.5.2.14.7 ExactMatchPacket()

```
// ExactMatchPacket()
// ==================
// Parses an exact address match.

ExactMatchPacket(bits(2) QE)
    q = UInt(QE);
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(q);
    AddressHistoryBuffer.Add(entry);
    return;
```

R_{TKBWR} The *Context element* value is created by combining the value encoded in a Context Packet and the last *Context element* value.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.14.8 Parse_AddressWithContextPacket()

```
// Parse_AddressWithContextPacket()
// ================================
// Parses a Target Address with Context packet.

Parse_AddressWithContextPacket(bits(8) header, bits(S) stream)
    LongAddressBytes(header, stream);
    ContextBytes(stream);
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);

    Emit(ContextElement(DSTATE.IA.context_id,
                        DSTATE.IA.vmid,
                        DSTATE.IA.exception_level,
                        DSTATE.IA.security,
                        DSTATE.IA.sixty_four_bit));
    Emit(TargetAddressElement(entry));
    return;
```

K10.5.2.14.9 Parse_ContextBytes()

```
// ContextBytes()
// ==============
// Generates a Context packet from the stream.

ContextBytes(bits(S) stream)
    payload = ReadAndConsume(8, stream);
    EL = payload<1:0>;
    SBZ(payload<3:2>);
    SF = payload<4>;
    NS = payload<5>;
    V = payload<6>;
    C = payload<7>;
    case C:V of
        when '00'
            ContextPacket('1', C, V, NS, SF, EL,
                          DSTATE.IA.vmid,
                          DSTATE.IA.context_id);
        when '01'
            bits(32) VMID;
            VMID = POD(32, stream);
            ContextPacket('1', C, V, NS, SF, EL,
                          VMID,
                          DSTATE.IA.context_id);
        when '10'
            context_id = POD(32, stream);
            ContextPacket('1', C, V, NS, SF, EL,
                          DSTATE.IA.vmid,
                          context_id);
        when '11'
            bits(32) VMID;
            VMID = POD(32, stream);
            context_id = POD(32, stream);
            ContextPacket('1', C, V, NS, SF, EL,
                          VMID,
                          context_id);

    return;

// Parse_ContextPacket()
// =====================
// Parses a Context packet.

Parse_ContextPacket(bits(8) header, bits(S) stream)
    P = header<0>;
    if P == '0' then
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ContextPacket(P, '0', '0', '0', '0', '00',
                      DSTATE.IA.vmid,
                      DSTATE.IA.context_id);
    else
        ContextBytes(stream);

    Emit(ContextElement(DSTATE.IA.context_id,
                        DSTATE.IA.vmid,
                        DSTATE.IA.exception_level,
                        DSTATE.IA.security,
                        DSTATE.IA.sixty_four_bit));
    return;
```

K10.5.2.14.10 ContextPacket()

```
// ContextPacket()
// ===============
// Processes a Context packet.

ContextPacket(bit P,
              bit C,
              bit V,
              bit NS,
              bit SF,
              bits(2) EL,
              bits(32) VMID,
              bits(32) context_id)
    if P == '1' then
        if C == '1' then
            DSTATE.IA.context_id = context_id;
        if V == '1' then
            DSTATE.IA.vmid = VMID;
        DSTATE.IA.exception_level<1:0> = EL<1:0>;
        if NS == '1' then
            DSTATE.IA.security = SecurityLevel_NONSECURE;
        else
            DSTATE.IA.security = SecurityLevel_SECURE;
        DSTATE.IA.sixty_four_bit = (SF == '1');

    return;
```

K10.5.2.15 Timestamps

R_{KJNMB} The *Timestamp element* value is created by combining the value encoded in a Timestamp Packet and the last *Timestamp element* value.

K10.5.2.15.1 Parse_TimestampPacket()

```
// Parse_TimestampPacket()
// =======================
// Parses a Timestamp packet.

Parse_TimestampPacket(bits(8) header, bits(S) stream)
    bit N;
    bits(64) TS = BitReplacement(stream, DSTATE.IA.timestamp);
    bits(20) COUNT = Zeros();
    N = header<0>;
    if N == '1' then
        COUNT = ULEB128(stream);
        TimestampPacket(N, TS, COUNT);
    else
        TimestampPacket(N, TS, COUNT);

    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.15.2 Parse_CycleCountPackets()

```
// Parse_CycleCountPackets()
// =========================
// Parses all the various Cycle Count packets.

Parse_CycleCountPackets(bits(8) header, bits(S) stream)
    case header of
        when '0000111x'                 // Cycle Count format 1
            bit U = header<0>;
            bits(32) COMMIT = Zeros();
            bits(20) COUNT = Zeros();
            if TRCIDR0.COMMOPT == '0' then
                COMMIT = ULEB128(stream);
            if U == '0' then
                COUNT = ULEB128(stream);
            CycleCountFormat1Packet(U, COMMIT, COUNT);

        when '0000110x'                 // Cycle Count format 2
            bits(8) payload = POD(8, stream);
            bits(4) BBBB = payload<3:0>;
            if TRCIDR0.COMMOPT == '0' then
                bit F = header<0>;
                bits(4) AAAA = payload<7:4>;
                CycleCountFormat2Packet(F, AAAA, BBBB);
            else
                CycleCountFormat2Packet('1', '1111', BBBB);

        when '0001xxxx'                 // Cycle Count format 3
            bits(2) BB = header<1:0>;
            if TRCIDR0.COMMOPT == '0' then
                bits(2) AA = header<3:2>;
                CycleCountFormat3Packet(AA, BB);
            else
                CycleCountFormat3Packet('00', BB);

    return;
```

K10.5.2.15.3 TimestampPacket()

```
// TimestampPacket()
// =================
// Processes a Timestamp packet.

TimestampPacket(bit N, bits(64) TS, bits(CN) COUNT)
    DSTATE.IA.timestamp = TS;
    if N == '1' then
        Emit(TimestampElement(UInt(DSTATE.IA.timestamp), UInt(COUNT)));
    else
        Emit(TimestampElement(UInt(DSTATE.IA.timestamp), integer UNKNOWN));

    return;
```

K10.5.2.15.4 CycleCountFormat1Packet()

```
// CycleCountFormat1Packet()
// =========================
// Processes a Cycle Count packet, format 1.

CycleCountFormat1Packet(bit U, bits(N) COMMIT, bits(20) COUNT)
    if UInt(COMMIT) > 0 then
        Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
    if U == '1' then
        Emit(CycleCountElement(integer UNKNOWN));
    else
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Emit(CycleCountElement(UInt(COUNT) + DSTATE.IA.cc_threshold));

    return;
```

K10.5.2.15.5 CycleCountFormat2Packet()

```
// CycleCountFormat2Packet()
// =========================
// Processes a Cycle Count packet, format 2.

CycleCountFormat2Packet(bit F, bits(4) AAAA, bits(4) BBBB)
    if F == '1' then
        commit_count = DSTATE.IA.max_spec_depth + UInt(AAAA) - 15;
    else
        commit_count = UInt(AAAA) + 1;

    if commit_count > 0 then
        Emit(CommitElement(commit_count));
        UpdateSpecDepth(-commit_count);
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BBBB)));

    return;
```

K10.5.2.15.6 CycleCountFormat3Packet()

```
// CycleCountFormat3Packet()
// =========================
// Processes a Cycle Count packet, format 3.

CycleCountFormat3Packet(bits(2) AA, bits(2) BB)
    if !DSTATE.IA.commit_mode then
        Emit(CommitElement(UInt(AA) + 1));
        UpdateSpecDepth(-(UInt(AA) + 1));
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BB)));

    return;
```

K10.5.2.16 Event tracing

K10.5.2.16.1 Parse_EventTracingPacket()

```
// Parse_EventTracingPacket()
// ==========================
// Parses an Event packet.

Parse_EventTracingPacket(bits(8) header, bits(S) stream)
    EventTracingPacket(header<3:0>);
    return;
```

K10.5.2.16.2 EventTracingPacket()

```
// EventTracingPacket()
// ====================
// Processes an Event packet.

EventTracingPacket(bits(4) EVENT)
    for I = 0 to 3
        if EVENT<I> == '1' then
            Emit(EventElement(I));

    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.17 Functions

K10.5.2.17.1 ReadAndConsume()

```
// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.

bits(N) ReadAndConsume(integer N, bits(S) stream)
    if DSTATE.stream_ptr == S then
        print ("Reached end of trace\n");
        assert(FALSE);

    bits(N) data = stream<DSTATE.stream_ptr + N - 1: DSTATE.stream_ptr>;
    DSTATE.stream_ptr = DSTATE.stream_ptr + N;
    return data;
```

K10.5.2.17.2 HandleAtom()

```
// HandleAtom()
// =============
// Emits an atom, and updates the speculation depth.

HandleAtom(Atom t)
    Emit(AtomElement(t));
    UpdateSpecDepth(1);
    return;
```

K10.5.2.17.3 UpdateSpecDepth()

```
// UpdateSpecDepth()
// ===================
// Update the speculation depth by a number of elements.

UpdateSpecDepth(integer count)
    DSTATE.IA.current_spec_depth = DSTATE.IA.current_spec_depth + count;
    if DSTATE.IA.current_spec_depth > DSTATE.IA.max_spec_depth then
        commit_number = DSTATE.IA.current_spec_depth - DSTATE.IA.max_spec_depth;
        Emit(CommitElement(commit_number));
        DSTATE.IA.current_spec_depth = DSTATE.IA.max_spec_depth;

    return;
```

K10.5.2.17.4 UpdateAddressHistoryBuffer()

```
// UpdateAddressHistoryBuffer()
// ============================
// Adds the given address and sub_isa to the AHB.

UpdateAddressHistoryBuffer(bits(64) address, SubISA sub_isa)
    AddressHistoryBuffer.Add(address, sub_isa);
    return;
```

K10.5.2.17.5 UnknownAddressHistoryBuffer()

```
// UnknownAddressHistoryBuffer()
// =============================
// Adds an unknown address and sub_isa to the AHB.

UnknownAddressHistoryBuffer()
    AddressHistoryBuffer.Add(Zeros(), IS0);
    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.3 Stage 2, speculation resolution

I_{KRXLC} The resolution stage operates on the Elements in turn. The Elements are buffered until their resolution is determined.

K10.5.3.1 Emit

K10.5.3.1.1 Emit()

```
// Emit()
// ======
Emit(Element e)
    e.committed = FALSE;
    case e.kind of
        when ELEM_TRACE_INFO     ProcessTraceInfo(e);

        // Speculation Support
        when ELEM_CANCEL             ProcessCancel(e);
        when ELEM_COMMIT             ProcessCommit(e);
        when ELEM_DISCARD            ProcessDiscard(e);

        // Transactional Support
        when ELEM_TRANS_START        ProcessTransactionStart(e);
        when ELEM_TRANS_COMMIT       ProcessTransactionCommit(e);
        when ELEM_TRANS_FAILURE      ProcessTransactionFailure(e);

        // Others
        otherwise           Stack(e);
```

K10.5.3.2 Trace Info element

I_{RJDHD} The *Trace Info element* can be used as a point to start decompression of the trace element stream. When the *Trace Info element* is generated there might still be some speculative *P0 elements*. The number of speculative *P0 elements* is indicated by the current speculation depth member of the *Trace Info element*.

R_{JKLZY} If the analysis of the trace starts with a *Trace Info element* with a nonzero current speculation depth the decompressor must ignore the *Commit element* or *Cancel elements* for these *P0 elements* as they will not have been observed by the decompressor.

K10.5.3.2.1 ProcessTraceInfo()

```
// ProcessTraceInfo()
// ==================
// Processes a Trace Info element, resetting the analyzer to a known state.

ProcessTraceInfo(Element e)
    if ResolutionQueue.Uninitialized() then
        ResolutionQueue.Initialize(e.payload.current_spec_depth);

        if e.payload.in_transaction then
            TransactionQueue.StartTransaction();

    Stack(e);
    return;
```

K10.5.3.3 Commit element

R_{WGSCJ} The *Commit element* marks a number of *P0 elements* as resolved, and if at the head of the queue pass these elements onto the next stage of the decompressor.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.3.3.1 ProcessCommit()

```
// ProcessCommit()
// ===============
// Processes a Commit element, committing the given number of speculative
// elements.

ProcessCommit(Element e)
    integer I = 0;

    repeat
        if !ResolutionQueue.Aligned() then
            I = I + 1;
            ResolutionQueue.Align();
        else
            case ResolutionQueue.Front().kind of

                when ELEM_EXCEPTION, ELEM_ATOM, ELEM_Q, ELEM_SOURCE_ADDRESS
                    if !ResolutionQueue.Front().committed then
                        I = I + 1;

                    ProcessTransaction(ResolutionQueue.Front());
                    AnalyzeElement(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();


                when ELEM_TRANS_START
                    if (TRCIDR0.COMMTRANS == '0' &&
                        !ResolutionQueue.Front().committed) then
                        I = I + 1;
                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();

                otherwise
                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();

    until I == e.payload.count;

    return;
```

K10.5.3.4 Cancel element

INYYFS For example, if a *Cancel element* indicates that the three most recent *P0 elements* are canceled, then the trace analyzer must discard:

- The *Cancel element*.
- All elements back to, and including, the third most recent *P0 element*.
- Any *Trace On elements* encountered in that section of the element stream.

RSPBCG When discarding *P0 elements* that have been canceled, a trace analyzer must also discard many other element types that occur in the element stream between the *Cancel element* and the oldest *P0 element* that the *Cancel element* cancels. Table K10-2 shows which elements must be discarded.

Table K10-2 Cancel Element Operation

Element	Behavior on cancellation
Atom	Remove
Commit	Illegal
Context	Remove

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Element	Behavior on cancellation
Cycle Count	Process
Discard	Illegal
Exception	Remove
Event	Keep
Mispredict	Remove
Overflow	Illegal
Target Address	Remove
Source Address	Remove
Timestamp	Process
Trace Info	Keep
Trace On	Remove

R_{CBYHC} When a *Cancel element* occurs, a trace analyzer must not discard *Cycle Count elements*.

R_{GZWHY} When a *Cancel element* occurs, a trace analyzer must not discard *ETEEvents*.

R_{ZLVXG} When a *Cancel element* occurs, a trace analyzer must discard *Mispredict elements*.

K10.5.3.4.1 ProcessCancel()

```
// ProcessCancel()
// ===============
// Processes a Cancel element, canceling the given number of speculative
// elements.

ProcessCancel(Element e)
    integer I = 0;

    repeat
        if !ResolutionQueue.Aligned() then
            I = I + 1;
            ResolutionQueue.Align();
        else
            case ResolutionQueue.Back().kind of

                when ELEM_ATOM, ELEM_EXCEPTION, ELEM_Q, ELEM_SOURCE_ADDRESS
                    if !ResolutionQueue.Back().committed then
                        I = I + 1;
                when ELEM_TRANS_START
                    if (TRCIDR0.COMMTRANS == '0' &&
                        !ResolutionQueue.Back().committed) then
                        I = I + 1;
                        TransactionQueue.EndTransaction();
                when ELEM_CYCLE_COUNT, ELEM_EVENT, ELEM_TRACE_INFO
                    AnalyzeElement(ResolutionQueue.Back());

                when ELEM_TIMESTAMP
                    AnalyzeElement(ResolutionQueue.Back());

            ResolutionQueue.PopBack();
    until I == e.payload.count;

    return;
```

K10.5.3.5 Discard element

I_{LYPKS} A *Discard element* indicates that tracing has become inactive while uncommitted *P0 elements* remain.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RYHRPG The trace analyzer must cancel all speculative *P0 elements*.

K10.5.3.5.1 ProcessDiscard()

```
// ProcessDiscard()
// ================
// Processes a Discard element, discarding all speculative elements.

ProcessDiscard(Element e)

    while !ResolutionQueue.Aligned() do
            ResolutionQueue.Align();

    while ResolutionQueue.Length() > 0 do
        case ResolutionQueue.Back().kind of

            when ELEM_EVENT, ELEM_TRACE_INFO, ELEM_TIMESTAMP
                AnalyzeElement(ResolutionQueue.Back());
                ResolutionQueue.PopBack();

            otherwise
                ResolutionQueue.PopBack();

    TransactionQueue.EndTransaction();
    return;
```

K10.5.3.6 Stack

IJTJSW The Elements processed by this stage of the decompressor must be stored temporarily until the speculation has been resolved.

K10.5.3.6.1 Stack()

```
// Stack()
// =======
// Pushes an element onto the resolution queue.

Stack(Element e)
    ResolutionQueue.Push(e);
    return;
```

K10.5.4 Stage 3, analysis

K10.5.4.1 Analyze element

K10.5.4.1.1 AnalyzeElement()

```
// AnalyzeElement()
// ================
// Analyzes any element.

AnalyzeElement(Element e)
    case e.kind of
        when ELEM_TARGET_ADDRESS AnalyzeTargetAddress(e);
        when ELEM_CONTEXT        AnalyzeContext(e);
        when ELEM_MISPREDICT     AnalyzeMispredict(e);
        when ELEM_TRACE_ON       AnalyzeTraceOn(e);
        when ELEM_ATOM           AnalyzeAtom(e);
        when ELEM_EXCEPTION      AnalyzeException(e);
        when ELEM_Q              AnalyzeQ(e);
        when ELEM_CANCEL         ERROR("cancel element reached analysis stage");
        when ELEM_COMMIT         ERROR("commit element reached analysis stage");
        when ELEM_DISCARD        AnalyzeDiscard(e);
        when ELEM_OVERFLOW       AnalyzeOverflow(e);
        when ELEM_EVENT          AnalyzeEvent(e);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when ELEM_TRACE_INFO     AnalyzeTraceInfo(e);
        when ELEM_TIMESTAMP      AnalyzeTimestamp(e);
        when ELEM_CYCLE_COUNT    AnalyzeCycleCount(e);
        when ELEM_SOURCE_ADDRESS AnalyzeSourceAddress(e);
        otherwise
            ERROR("Unrecognised element kind in analysis stage");

    return;
```

K10.5.4.2 Retained state

R_{GWFMZ} The trace analyzer must maintain a copy of the context:

- Context identifier.
- Virtual context identifier.
- Execution state.
- Exception level.
- Security state.

K10.5.4.2.1 ReconstructState()

```
// ReconstructState
// ================
// Temporary storage of reconstructor state, can change after resolution.

type ReconstructState is (
    bits(64) address,           // Current address
    bits(32) context_id,        // Current context ID
    bits(32) vmid,              // Current VMID
    bits(2) exception_level,    // Current Exception level
    SecurityLevel security,     // Current Security state
    boolean sixty_four_bit,     // Are we in AArch64?
    SubISA sub_isa              // Current sub_isa
)
```

K10.5.4.3 Operation of the return stack

I_{SZYZF} The trace analyzer maintains an independent copy of the return stack which is used to determine when *Target Address elements* have been removed and then infer the target of indirect *P0 instructions*.

I_{LGKYD} The trace analyzer return stack only operates after a certain point in the tracing flow, that is after the trace analyzer has decoded the trace packets and after all the elements that indicate speculative execution, except for *Mispredict elements*, have been removed from the trace element stream.

R_{HYMQH} Whenever a Branch with Link instruction is initially traced with an E *Atom element*, the link return address and sub_isa are pushed onto the trace analyzer return stack. This means that the trace unit return stack grows with each new entry, until its maximum depth is reached and the oldest entries start being discarded.

R_{MGHSB} Whenever an indirect *P0 instruction* is traced with a final E *Atom element*, and no *Target Address element* is traced before the next *P0 element*, the top entry of the trace analyzer return stack is removed and the value of that entry is the target of the indirect *P0 instruction*.

R_{XSVZC} A trace analyzer is not required to be aware of the depth of the trace unit return stack, and implements a return stack depth of 15 entries.

I_{GBVND} A trace analyzer return stack push always occurs whenever a Branch with Link instruction is traced with an E *Atom element*, even if the status of the E *Atom element* later changes to be an N *Atom element* as a result of a subsequent *Mispredict element*.

For example, the following sequence might occur:

1. The PE speculatively executes a Branch with Link instruction that the trace unit traces with an E *Atom element*. The trace unit pushes the target address of the Branch with Link instruction onto the trace unit return stack.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. The trace analyzer receives the E *Atom element* and pushes the target address of the Branch with Link instruction onto the trace analyzer return stack.
3. The PE then cancels the speculative execution. The trace unit generates a *Mispredict element*.
4. The trace analyzer receives the *Mispredict element* and changes the status of the E *Atom element* so that it becomes an N *Atom element*. The trace analyzer then knows which direction the program flow has taken, and also knows that the target address stored at the top of the trace analyzer return stack is mispredicted.

Note: Whenever the trace unit generates a *Mispredict element* to correct a Branch with Link instruction to an N *Atom element*, the mispredicted address remains in the return stack because there is no reason to remove it. There are no adverse consequences of leaving such a mispredicted address in the stack.

R_{GMTBD} If more than one *Mispredict element* is output corresponding to a particular *Atom element*, the status of the *Atom element* alternates between E and N until it settles in its final E or N state. If the final state of the *Atom element* is E, then when the PE executes an indirect *P0 instruction* and the trace unit compares the target address with the top entry in its return stack, an address match might occur. An address match can only occur if the final status of the *Atom element* is E.

I_{MPGDD} The trace analyzer never needs to discard the entries in its copy of the return stack. If the trace unit discards the entries in its return stack then the entries in the trace analyzer return stack remain. As more entries are pushed on to the return stack, the old entries are discarded when they are pushed off the end of the stack.

I_{WJLDR} The trace analyzer does not need to prevent the return stack from being modified while in a branch broadcasting region. The fact that the trace unit discards the entries in its return stack when entering the branch broadcasting region ensures that the return stack in the trace unit and the return stack in the trace analyzer remain synchronized.

K10.5.4.3.1 UpdateReturnStack()

```
// UpdateReturnStack()
// ===================

// Push the given instruction to the return stack if necessary.

UpdateReturnStack(DecodedInst inst)
    if inst.is_link then
        nxt_state = DSTATE.current_analyzer_state;
        nxt_state.address = nxt_state.address + inst.size;
        if !nxt_state.sixty_four_bit then
            nxt_state.address<63:32> = Zeros();

        ReturnStack.Push(nxt_state.address, nxt_state.sub_isa);

    return;
```

K10.5.4.4 Atom element

R_{SKGMZ} An *Atom element* implies that one or more instructions have been traced, up to and including the next *P0 instruction*.

I_{VHHGW} A trace analyzer must analyze each instruction in the program image from the current address until it observes a *P0 instruction*. This indicates that the PE has executed each instruction between the current address and the *P0 instruction*.

K10.5.4.4.1 AnalyzeAtom()

```
// AnalyzeAtom()
// =============
// Analyzes an atom element.

AnalyzeAtom(Element e)
    if e.payload.atom_type == Atom_E then
        DSTATE.most_recent_branch_was_taken = TRUE;
    else
        DSTATE.most_recent_branch_was_taken = FALSE;

    CheckForReturnStackMatch();

    if DSTATE.sync_state == ADDRESS_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if DSTATE.sync_state != FULL_SYNC_STATE then
        // If we are unsure of context or address then we cannot meaningfully
        // analyze the atom.
        return;

    boolean cur_inst_is_branch = FALSE;

    // Continue logging instructions until we hit a P0 instruction.
    while !cur_inst_is_branch do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();

        case decoded_inst.branchtype of
            when InstType_BRANCH_DIR, InstType_BRANCH_INDIR
                ProcessBranchInstruction(decoded_inst,
                                         DSTATE.most_recent_branch_was_taken);
                cur_inst_is_branch = TRUE;
                UpdateReturnStack(decoded_inst);
            when InstType_WFX, InstType_ISB
                ProcessBranchInstruction(decoded_inst,
                                         DSTATE.most_recent_branch_was_taken);
                cur_inst_is_branch = TRUE;
            when InstType_OTHER
                ReconstructState nxt_state = DSTATE.current_analyzer_state;
                nxt_state.address = nxt_state.address + decoded_inst.size;
                if !nxt_state.sixty_four_bit then
                    // mask off the left-most bits
                    nxt_state.address<63:32> = Replicate('0', 32);
                DSTATE.next_analyzer_state = nxt_state;

        OutputInstruction(decoded_inst);
        DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;

    if DSTATE.return_stack_clear_pending then
        ReturnStack.Reset();
    return;
```

K10.5.4.5 Context element

K10.5.4.5.1 AnalyzeContext()

```
// AnalyzeContext()
// ================
// Analyzes a context element.

AnalyzeContext(Element e)
    DSTATE.current_analyzer_state.context_id = e.payload.context_id;
    DSTATE.current_analyzer_state.vmid = e.payload.vmid;
    DSTATE.current_analyzer_state.security = e.payload.security;
    DSTATE.current_analyzer_state.exception_level = e.payload.exception_level;
    DSTATE.current_analyzer_state.sixty_four_bit = e.payload.sixty_four_bit;

    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = CONTEXT_STATE;
        when ADDRESS_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise

    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4.6 Exception element

RPJFBL For an *Exception element*, a trace analyzer must analyze each instruction from the current address, up to but not including the exception return address that the element provides. The PE has executed each instruction in that address range. The number of instructions that are executed can be zero.

Note: Trace analysis tools must be aware, that if PE execution is at the top of memory space, the address that the *Exception element* provides might be lower than the target address of the most recent *P0 element*.

K10.5.4.6.1 AnalyzeException()

```
// AnalyzeException()
// ==================
// Analyzes an exception element.

AnalyzeException(Element e)
    CheckForReturnStackMatch();

    if DSTATE.sync_state == CONTEXT_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        return;

    integer PER = UInt(e.payload.address);
    if (ExceptionWithUnknownAddress(e)) then
        continue_forward = FALSE;
    elsif UInt(DSTATE.current_analyzer_state.address) < PER then
        continue_forward = TRUE;
    else
        continue_forward = FALSE;

    // Continue logging instructions until we reach the specified address.
    while continue_forward do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();

        if decoded_inst.branchtype == InstType_OTHER then
            ReconstructState nxt_state = DSTATE.current_analyzer_state;
            nxt_state.address = nxt_state.address + decoded_inst.size;
            DSTATE.next_analyzer_state = nxt_state;
            if !DSTATE.next_analyzer_state.sixty_four_bit then
                // mask off the left-most bits
                DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
        else
            ProcessBranchInstruction(decoded_inst, FALSE);
            UpdateReturnStack(decoded_inst);

        OutputInstruction(decoded_inst);

        bits(64) next_addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
        DSTATE.current_analyzer_state.address = next_addr;
        if !DSTATE.current_analyzer_state.sixty_four_bit then
            // mask off the left-most bits
            DSTATE.current_analyzer_state.address<63:32> = Replicate('0', 32);

        if UInt(DSTATE.current_analyzer_state.address) >= PER then
           continue_forward = FALSE;

    if DSTATE.return_stack_clear_pending then
        ReturnStack.Reset();

    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4.7 Source address element

R_{VDRPP} A *Source Address element* indicates that one or more instructions have been traced, up to and including the instruction at the address associated with the element.

R_{YJHMV} A *Source Address element* indicates that the instruction at the address associated with the element was taken.

I_{GWZMF} A trace analyzer must analyze each instruction in the program image from the current address until it analyzes the instruction at the address associated with the *Source Address element*. This indicates that the PE has executed each instruction between the current address and that instruction, and each *P0 instruction* except the final instruction was not taken.

K10.5.4.7.1 AnalyzeSourceAddress()

```
// AnalyzeSourceAddress()
// ======================
// Analyzes a source address element.

AnalyzeSourceAddress(Element e)
    CheckForReturnStackMatch();

    if DSTATE.sync_state == ADDRESS_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        // If we are unsure of context or address then we cannot meaningfully
        // analyze the source address.
        return;

    DSTATE.most_recent_branch_was_taken = FALSE;
    integer address = UInt(e.payload.address);

    // Continue logging instructions until we hit the specified address.
    while (UInt(DSTATE.current_analyzer_state.address) <= address) do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();

        if decoded_inst.branchtype == InstType_OTHER then
            ReconstructState nxt_state = DSTATE.current_analyzer_state;
            nxt_state.address = nxt_state.address + decoded_inst.size;
            DSTATE.next_analyzer_state = nxt_state;
            if !DSTATE.next_analyzer_state.sixty_four_bit then
                DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
        else
            if DSTATE.current_analyzer_state.address == e.payload.address then
                 DSTATE.most_recent_branch_was_taken = TRUE;
            ProcessBranchInstruction(decoded_inst,
                                     DSTATE.most_recent_branch_was_taken);

            cur_inst_is_branch = TRUE;
            UpdateReturnStack(decoded_inst);

        OutputInstruction(decoded_inst);

        DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;

    if DSTATE.return_stack_clear_pending then
        ReturnStack.Reset();
    return;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4.8 Target Address element

K10.5.4.8.1 AnalyzeTargetAddress()

```
// AnalyzeTargetAddress()
// ======================
// Analyzes a target address element.

AnalyzeTargetAddress(Element e)
    DSTATE.current_analyzer_state.address = e.payload.address;
    DSTATE.current_analyzer_state.sub_isa = e.payload.sub_isa;

    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = ADDRESS_STATE;
        when CONTEXT_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise

    return;
```

K10.5.4.9 Trace Info element

K10.5.4.9.1 AnalyzeTraceInfo()

```
// AnalyzeTraceInfo()
// ==================
// Analyzes a trace info element.

AnalyzeTraceInfo(Element e)
    CheckForReturnStackMatch();
    return_stack_clear_pending = TRUE;
    return;
```

K10.5.4.10 Trace On element

K10.5.4.10.1 AnalyzeTraceOn()

```
// AnalyzeTraceOn()
// ================
// Analyzes a trace on element.

AnalyzeTraceOn(Element e)
    return_stack_clear_pending = TRUE;
    DSTATE.sync_state = NOT_SYNC_STATE;
    return;
```

K10.5.4.11 Mispredict element

RZRVMZ When a *Mispredict element* corresponds to an *Atom element* for a direct *P0 instruction*, before the trace analyzer can calculate the target of the direct *P0 instruction*, it must apply any applicable *Mispredict elements* so that it can determine whether it is an E *Atom element* or an N *Atom element*.

K10.5.4.11.1 AnalyzeMispredict()

```
// AnalyzeMispredict()
// ===================
// Analyzes a mispredict element.

AnalyzeMispredict(Element e)
    DSTATE.most_recent_branch_was_taken = !DSTATE.most_recent_branch_was_taken;

    ReconstructState nxt_state;
    nxt_state = UpdateBranchState(DSTATE.most_recent_branch_decoded_inst,
                                  DSTATE.most_recent_branch_state,
                                  DSTATE.most_recent_branch_was_taken);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    DSTATE.current_analyzer_state = nxt_state;

    return;
```

K10.5.4.12 ETEEvent element

K10.5.4.12.1 AnalyzeEvent()

```
// AnalyzeEvent()
// ==============
// Analyzes an event element.

AnalyzeEvent(Element e);
```

K10.5.4.13 Discard element

I_{JYPQC} When a trace analyzer encounters a *Discard element* it must be aware that if the last committed *P0 element* is a conditional *P0 instruction*, the E or N status of that *Atom element* might not be correct. This is because the trace unit might be unable to generate any *Mispredict elements* that the conditional *P0 instruction* might require.

I_{HPQFK} If the last *P0 instruction* is an indirect *P0 instruction* then the target address indicated in the trace stream might be incorrect. This is because the trace unit might be unable to generate any *Target Address elements* that the indirect *P0 instruction* might require.

K10.5.4.13.1 AnalyzeDiscard()

```
// AnalyzeDiscard()
// ================
// Analyzes a discard element.

AnalyzeDiscard(Element e)
    DSTATE.sync_state = NOT_SYNC_STATE;
    return;
```

K10.5.4.14 Overflow element

K10.5.4.14.1 AnalyzeOverflow()

```
// AnalyzeOverflow()
// =================
// Analyzes an overflow element.

AnalyzeOverflow(Element e)
    DSTATE.sync_state = NOT_SYNC_STATE;
    return;
```

K10.5.4.15 Q element

I_{TXGPZ} When a trace analyzer encounters a *Q element* which has a count of M executed instructions, it must proceed through the program image, analyzing each instruction until it has analyzed M instructions. If it encounters a conditional *P0 instruction*, the status of the condition code check for that instruction is UNKNOWN. The status of these *P0 instructions* is not explicitly given in the trace element stream but it might be possible to infer the status of a given *P0 instruction* that is based on other trace that is generated. After the trace analyzer has analyzed M instructions, the following *Target Address element* indicates where PE execution continues.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4.15.1 AnalyzeQ()

```
// AnalyzeQ()
// ==========
// Analyzes a Q element.

AnalyzeQ(Element e)
    CheckForReturnStackMatch();

    q_with_count = e.payload.count > 0;
    if q_with_count then
        further_analysis_possible = TRUE;
    else
        further_analysis_possible = FALSE;
        DSTATE.sync_state = CONTEXT_STATE;
        // If we have no count then just wait to resync, it is not safe to guess

    i = 0;
    while further_analysis_possible do
        if ProgramImage.DecodeAvailable() then
            decoded_inst = ProgramImage.DecodeNextInst();
            addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
            DSTATE.current_analyzer_state.address = addr;
        else
            DSTATE.sync_state = CONTEXT_STATE;
            return;

        i = i + 1;

        further_analysis_possible = (i < e.payload.count);
        OutputInstruction(decoded_inst);

    return;
```

K10.5.4.16 Timestamp element

K10.5.4.16.1 AnalyzeTimestamp()

```
// AnalyzeTimestamp()
// ==================
// Analyzes a timestamp element.

AnalyzeTimestamp(Element e);
```

K10.5.4.17 Cycle Count element

I_{HTQDP} To produce a total cycle count, a trace analyzer can cumulatively add the values from all *Cycle Count elements*.

R_{TVWLZ} A trace analyzer must not use the cycle count values in *Timestamp elements* to produce a total cycle count. Such cycle count values are not a *Cycle Count element*.

K10.5.4.17.1 AnalyzeCycleCount()

```
// AnalyzeCycleCount()
// ===================
// Analyzes a cycle count element.

AnalyzeCycleCount(Element e);
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4.18 Functions

K10.5.4.18.1 OutputInstruction()

```
// OutputInstruction()
// ===================
// Log that an instruction has been executed.

OutputInstruction(DecodedInst inst)
    DSTATE.inst_out_count = DSTATE.inst_out_count + 1;
    return;
```

K10.5.4.18.2 CheckReturnStackMatch()

```
// CheckForReturnStackMatch()
// ==========================
// Check if there is a return stack match, and log the result.

CheckForReturnStackMatch()
    if DSTATE.sync_state == CONTEXT_STATE then
        if !ReturnStack.IsEmpty() then
            ReturnStackEntry top = ReturnStack.Pop();
            DSTATE.current_analyzer_state.address = top.address;
            DSTATE.current_analyzer_state.sub_isa = top.sub_isa;
            DSTATE.sync_state = FULL_SYNC_STATE;

    return;
```

K10.5.4.18.3 UpdateBranchState()

```
// UpdateBranchState()
// ===================
// Returns an updated state based on what was executed.

ReconstructState UpdateBranchState(DecodedInst inst,
                                   ReconstructState in_state,
                                   boolean branch_was_taken)

    out_state = DSTATE.current_analyzer_state;
    out_state.address = in_state.address;
    out_state.sixty_four_bit = in_state.sixty_four_bit;
    out_state.sub_isa = in_state.sub_isa;

    if branch_was_taken then
        if inst.branchtype == InstType_BRANCH_INDIR then
            DSTATE.sync_state = CONTEXT_STATE;
        else
            if inst.branchtype == InstType_BRANCH_DIR then
                out_state.address = out_state.address + inst.addressoffset;
            else
                out_state.address = out_state.address + inst.size;
            if !in_state.sixty_four_bit then
                out_state.address<63:32> = Zeros();
            out_state.sub_isa = inst.next_sub_isa;
    else
        out_state.address = out_state.address + inst.size;
        if !out_state.sixty_four_bit then
            out_state.address<63:32> = Zeros();

    return out_state;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4.18.4 ProcessBranchInstruction()

```
// ProcessBranchInstruction()
// ==========================
// Store current state before a branch instruction, as it could change if there
// is a misprediction.

ProcessBranchInstruction(DecodedInst inst, boolean branch_was_taken)
        DSTATE.most_recent_branch_state = DSTATE.current_analyzer_state;
        DSTATE.most_recent_branch_decoded_inst = inst;
        DSTATE.most_recent_branch_was_taken = branch_was_taken;
        DSTATE.next_analyzer_state = UpdateBranchState(inst,
                                                DSTATE.current_analyzer_state,
                                                branch_was_taken);

        return;
```

K10.5.4.18.5 DecodedInst()

```
// DecodedInst
// ===========
// Data extracted from an instruction.

type DecodedInst is (
    bits(32) instruction,    // The instruction itself
    InstType branchtype,     // Type of P0 instruction
    boolean is_link,         // Is it a linking branch?
    integer size,            // Size (32 or 16)
    SubISA next_sub_isa,     // sub_isa of the following instruction to be
                             // executed
    bits(64) addressoffset
)
```

K10.5.5 ETE common pseudocode

K10.5.5.1 Element ASL

K10.5.5.1.1 Atom enumeration

```
// Atom
// ====
// Atom enum. Atoms are either E (taken) or N (not taken).
enumeration Atom {
    Atom_E,
    Atom_N
};
```

K10.5.5.1.2 AtomElement()

```
// AtomElement()
// =============
// Generates an Atom element based on the given atom.
Element AtomElement(Atom t)
    Element a;

    a.kind = ELEM_ATOM;
    a.payload.atom_type = t;

    return a;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.1.3 QElement()

```
// QElement()
// ==========
// Generates a Q element based on the number of elements
// to resolve.
Element QElement(integer count)
    Element a;

    a.kind = ELEM_Q;
    a.payload.count = count;

    return a;
```

K10.5.5.1.4 CancelElement()

```
// CancelElement()
// ===============
// Generates a Cancel element based on a given number
// of elements to cancel.
Element CancelElement(integer count)
    Element a;

    a.kind = ELEM_CANCEL;
    a.payload.count = count;

    return a;
```

K10.5.5.1.5 CommitElement()

```
// CommitElement()
// ===============
// Generates a commit element based on the
// number of elements to commit.
Element CommitElement(integer count)
    Element a;

    a.kind = ELEM_COMMIT;
    a.payload.count = count;

    return a;
```

K10.5.5.1.6 ContextElement()

```
// ContextElement()
// ================
// Generates a context element based on context ID, VMID,
// Exception level, Security state and AArch32/64 state.
Element ContextElement(bits(32) context_id,
                       bits(32) vmid,
                       bits(2) exception_level,
                       SecurityLevel secure,
                       boolean sixty_four_bit)
    Element a;

    a.kind = ELEM_CONTEXT;
    a.payload.context_id = context_id;
    a.payload.vmid = vmid;
    a.payload.exception_level = exception_level;
    a.payload.security = secure;
    a.payload.sixty_four_bit = sixty_four_bit;

    return a;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.1.7 CycleCountElement()

```
// CycleCountElement()
// ===================
// Generates a Cycle Count element based on a number of cycles.
Element CycleCountElement(integer count)
    Element a;

    a.kind = ELEM_CYCLE_COUNT;
    a.payload.count = count;

    return a;
```

K10.5.5.1.8 DiscardElement()

```
// DiscardElement()
// ================
// Generates a Discard element.
Element DiscardElement()
    Element a;

    a.kind = ELEM_DISCARD;

    return a;
```

K10.5.5.1.9 ExceptionElement()

```
// ExceptionElement()
// ==================
// Generates an Exception element based on the address to branch
// to and the type of exception.
Element ExceptionElement(integer exception_type, bits(64) address)
    Element a;

    a.kind = ELEM_EXCEPTION;
    a.payload.exception_type = exception_type;
    a.payload.address = address;

    return a;
```

K10.5.5.1.10 EventElement()

```
// EventElement()
// ==============
// Generates an Event element based on the number of the event that fired.
Element EventElement(integer idx)
    Element a;

    a.kind = ELEM_EVENT;
    a.payload.event_id = idx;

    return a;
```

K10.5.5.1.11 MispredictElement()

```
// MispredictElement()
// ===================
// Generates a Mispredict element.
Element MispredictElement()
    Element a;

    a.kind = ELEM_MISPREDICT;

    return a;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.1.12 OverflowElement()

```
// OverflowElement()
// =================
// Generates an Overflow element.
Element OverflowElement()
    Element a;

    a.kind = ELEM_OVERFLOW;

    return a;
```

K10.5.5.1.13 TimestampElement()

```
// TimestampElement()
// ==================
// Generates a Timestamp element based on a timestamp value
// and a cycle count value.
Element TimestampElement(integer timestamp, integer cycles)
    Element a;

    a.kind = ELEM_TIMESTAMP;
    a.payload.timestamp = timestamp;
    a.payload.cycle_count = cycles;

    return a;
```

K10.5.5.1.14 TraceInfoElement()

```
// TraceInfoElement()
// ==================
// Generates a Trace Info element based on cycle counting parameters,
// speculation depth, and transaction status.
Element TraceInfoElement(boolean cc_enabled,
                         integer cc_threshold,
                         integer current_spec_depth,
                         boolean in_transaction)
    Element a;

    a.kind = ELEM_TRACE_INFO;
    a.payload.cc_enabled = cc_enabled;
    a.payload.cc_threshold = cc_threshold;
    a.payload.current_spec_depth = current_spec_depth;
    a.payload.in_transaction = in_transaction;

    return a;
```

K10.5.5.1.15 TraceOnElement()

```
// TraceOnElement()
// ================
// Generates a Trace On element.
Element TraceOnElement()
    Element a;

    a.kind = ELEM_TRACE_ON;

    return a;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.1.16 TargetAddressElement()

```
// TargetAddressElement()
// ======================
// Generates a Target Address element based on a given
// address and sub_isa.
Element TargetAddressElement(AddressHistoryBufferEntry reg)
    Element a;
    a.kind = ELEM_TARGET_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;

    return a;
```

K10.5.5.1.17 SourceAddressElement()

```
// SourceAddressElement()
// ======================
// Generates a Source Address element based on an instruction's address
// and sub_isa.
Element SourceAddressElement(AddressHistoryBufferEntry reg)
    Element a;

    a.kind = ELEM_SOURCE_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;

    return a;
```

K10.5.5.1.18 TransactionStartElement()

```
// TransactionStartElement()
// =========================
Element TransactionStartElement()
    Element a;

    a.kind = ELEM_TRANS_START;
    return a;
```

K10.5.5.1.19 TransactionCommitElement()

```
// TransactionCommitElement()
// ==========================
Element TransactionCommitElement()
    Element a;

    a.kind = ELEM_TRANS_COMMIT;
    return a;
```

K10.5.5.1.20 TransactionFailureElement()

```
// TransactionFailureElement()
// ===========================
Element TransactionFailureElement()
    Element a;

    a.kind = ELEM_TRANS_FAILURE;
    return a;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.2 Decompressor enumerations

K10.5.5.2.1 SubISA enumeration

```
// SubISA
// ======
// Represents sub instruction set.
// IS0 = AArch32 or AArch64, IS1 = AArch32 Thumb
enumeration SubISA {
    IS0,
    IS1
};
```

K10.5.5.2.2 SynchronizationState enumeration

```
// States to represent synchronization of the reconstructor, state
// transitions as follows:
// ____________________________________________
// | Init State| Input              | Final State  |
// |___________|____________________|______________|
// | NOT_SYNC  | context element    | CONTEXT      |
// | NOT_SYNC  | address element    | ADDRESS      |
// | ADDRESS   | context element    | FULL_SYNC    |
// | ADDRESS   | overflow element   | NOT_SYNC     |
// | ADDRESS   | discard element    | NOT_SYNC     |
// | ADDRESS   | trace on element   | NOT_SYNC     |
// | ADDRESS   | atom element       | NOT_SYNC     |
// | ADDRESS   | exception element  | NOT_SYNC     |
// | CONTEXT   | address element    | FULL_SYNC    |
// | CONTEXT   | overflow element   | NOT_SYNC     |
// | CONTEXT   | discard element    | NOT_SYNC     |
// | CONTEXT   | trace on element   | NOT_SYNC     |
// | FULL_SYNC | indirect branch    | CONTEXT      |
// | FULL_SYNC | discard element    | NOT_SYNC     |
// | FULL_SYNC | overflow element   | NOT_SYNC     |
// | FULL_SYNC | trace on element   | NOT_SYNC     |
// |___________|____________________|______________|

enumeration SynchronizationState {
    NOT_SYNC_STATE, // Not syncing, need sync
    CONTEXT_STATE, // Have context, need address
    ADDRESS_STATE, // Have address, need context
    FULL_SYNC_STATE // Fully synced
};
```

K10.5.5.2.3 InstType enumeration

```
// InstType
// ========
// Instruction type. Cannot use BranchType as this does not cover other P0
// non-branching instructions (WFE/WFI, ISB).
// WFX counts as 'Other' if it is not a P0 element (see TRCIDR2.WFXMODE).

enumeration InstType {
    InstType_BRANCH_DIR, // Direct branch
    InstType_BRANCH_INDIR, // Indirect branch
    InstType_WFX, // WFI/WFE instruction
    InstType_ISB, // Instruction barrier
    InstType_OTHER // Non-P0 instructions
};
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.3 Decompressor functions

K10.5.5.3.1 EndOfStream()

```
// EndOfStream()
// =============
// Returns TRUE iff all the data in the stream have been consumed.
boolean EndOfStream(bits(S) stream);
```

K10.5.5.3.2 ReservedEncoding()

```
// ReservedEncoding()
// ==================
// The trace byte stream is not compliant to the protocol. The trace analyzer
// has to stop.
ReservedEncoding();
```

K10.5.5.3.3 ReadAndConsume()

```
// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.
bits(N) ReadAndConsume(integer N, bits(S) stream);
```

K10.5.5.3.4 LogDecompressor()

```
// Instrumentation functions
// =========================

LogDecompressor(Decomp_Level lvl, string details);
LogElem(Decomp_Level lvl, Element e, string details);
integer GetNextDebugId();
ERROR(string msg);
LogReturnStack();
PrintElement(Element e);
string ExcepTypeToStr(integer type_val);
```

K10.5.5.3.5 SBZ()

```
// SBZ()
// =====
// Raise an error if the field B is not zero.
SBZ(bits(N) B);
```

K10.5.5.3.6 ResolutionQueue

```
// ResolutionQueue.Initialize()
// ============================
// If decompression starts at a Trace Info element that has a nonzero
// speculation depth, the trace analyzer must wait until the speculation
// of these unseen P0 elements has been resolved.
//
// Set the number of unseen P0 elements that are outstanding that need to be
// resolved.
ResolutionQueue.Initialize(integer i);

// ResolutionQueue.Uninitialized()
// ===============================
// Returns TRUE if the resolution queue is uninitialized.
boolean ResolutionQueue.Uninitialized();

// ResolutionQueue.Aligned()
// =========================
// Returns TRUE if all the unseen P0 elements have been resolved.
boolean ResolutionQueue.Aligned();
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ResolutionQueue.Align()
// =======================
// Mark the oldest oldest unseen P0 element as resolved.
ResolutionQueue.Align();

// ResolutionQueue.Length()
// ========================
// Returns the number of elements in the queue.
integer ResolutionQueue.Length();

// ResolutionQueue.PopBack()
// =========================
// Discards the element at the back (youngest) of the queue.
ResolutionQueue.PopBack();

// ResolutionQueue.Back()
// ======================
// Returns the element at the back (youngest) of the queue.
Element ResolutionQueue.Back();

// ResolutionQueue.PopFront()
// ==========================
// Removes the element at the front (oldest) from the queue.
ResolutionQueue.PopFront();

// ResolutionQueue.Front()
// =======================
// Returns the element at the front (oldest) of the queue.
Element ResolutionQueue.Front();

// ResolutionQueue.CommitFront()
// =============================
// Commits the element at the front of the queue.
ResolutionQueue.CommitFront()

// ResolutionQueue.Push()
// ======================
// Add element e to the back of the queue.
ResolutionQueue.Push(Element e);
```

K10.5.5.3.7 TransactionQueue

```
// TransactionQueue.Length()
// =========================
// Return the number of entries in the transaction queue.
integer TransactionQueue.Length();
```

```
// TransactionQueue.FrontPop()
// ===========================
// Remove the first entry in the transaction queue.
TransactionQueue.FrontPop();

// TransactionQueue.Front()
// ========================
// Return the element at the front of the transaction queue.
Element TransactionQueue.Front();

// TransactionQueue.Push()
// =======================
// Add an element to the back of the transaction queue.
TransactionQueue.Push(Element e);

// TransactionQueue.InTransaction()
// ================================
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Are we currently in a transaction?
boolean TransactionQueue.InTransaction();

// TransactionQueue.StartTransaction()
// ===================================
// Enter a transaction.
TransactionQueue.StartTransaction();

// TransactionQueue.EndTransaction()
// =================================
// Leave a transaction.
TransactionQueue.EndTransaction();
```

K10.5.5.3.8 ReturnStack

```
// ReturnStack.Reset()
// ===================
// Resets the return stack.
ReturnStack.Reset();

// ReturnStack.Push(bits(64) addr, SubISA sub_isa)
// ===============================================
// Pushes onto the return stack.
ReturnStack.Push(bits(64) addr, SubISA sub_isa);

// ReturnStack.Pop()
// =================
// Pops the top of the return stack.
ReturnStackEntry ReturnStack.Pop();

// ReturnStack.IsEmpty()
// =====================
// Returns TRUE iff the return stack is empty.
boolean ReturnStack.IsEmpty();
```

K10.5.5.3.9 AddressHistoryBufferEntry()

```
// AddressHistoryBufferEntry
// =========================
// An entry in the address history buffer.
type AddressHistoryBufferEntry is (
    bits(64) address,
    SubISA sub_isa
)

AddressHistoryBufferEntry UNKNOWN_ADDRESS;
```

K10.5.5.3.10 AddressHistoryBuffer()

```
// AddressHistoryBuffer.Reset()
// ============================
// Resets the address history buffer.
AddressHistoryBuffer.Reset()
    for i = 0 to 2
        DSTATE.IA.address_history_buffer[i].address = Zeros();
        DSTATE.IA.address_history_buffer[i].sub_isa = IS0;

    return;

// AddressHistoryBuffer.Add()
// ==========================
// Adds an address to the address history buffer.
AddressHistoryBuffer.Add(AddressHistoryBufferEntry entry)
    DSTATE.IA.address_history_buffer[2] = DSTATE.IA.address_history_buffer[1];
    DSTATE.IA.address_history_buffer[1] = DSTATE.IA.address_history_buffer[0];
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    DSTATE.IA.address_history_buffer[0] = entry;
    return;

AddressHistoryBuffer.Add(bits(64) address, SubISA sub_isa)
    AddressHistoryBufferEntry entry;
    entry.address = address;
    entry.sub_isa = sub_isa;

    AddressHistoryBuffer.Add(entry);
    return;

// AddressHistoryBuffer.Get()
// ==========================
// Returns the given entry from the address history buffer.
AddressHistoryBufferEntry AddressHistoryBuffer.Get(integer n)
    assert n < 3;
    return DSTATE.IA.address_history_buffer[n];
```

K10.5.5.3.11 ProgramImage

```
// ProgramImage.DecodeNextInst()
// ============================
// Returns the decoded next instruction in the program image.
DecodedInst ProgramImage.DecodeNextInst();

// ProgramImageDecodeAvilable()
// ============================
// Returns TRUE iff we are currently inside the program image.
boolean ProgramImage.DecodeAvailable();
```

K10.5.5.3.12 ExceptionWithUnknownAddress()

```
// ExceptionWithUnknownAddress()
// =============================
// Does this exception type have an unknown
// prefered exception return address.

boolean ExceptionWithUnknownAddress(Element e)
    case e.payload.exception_type<4:0> of
        when '00000', '11001'
            return TRUE;
        when '11000'
            ERROR("Transation Failure Element");
        otherwise
            return FALSE;
```

K10.5.5.4 Data encodings

K10.5.5.4.1 POD()

```
// POD()
// =====
// Return data from stream in Plain Old Data Little Endian format.
bits(N) POD(integer N, bits(S) stream)
    return ReadAndConsume(N, stream);
```

K10.5.5.4.2 ULEB128()

```
// ULEB128()
// =========
// Gets N bits of continuable data from the stream.
bits(N) ULEB128(bits(S) stream)
    return BitReplacement(stream, Zeros(N));
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.4.3 BitReplacement()

```
// BitReplacement()
// ================
// Gets N bits of continuable, bit replacement data from the stream.
bits(N) BitReplacement(bits(S) stream, bits(N) original)
    R = original;

    I = 0;
    bits(8) BYTE;
    repeat
        BYTE = ReadAndConsume(8, stream);
        R<I+6:I> = BYTE<6:0>;
        I = I + 7;
    until BYTE<7> == '0' || I >= N - 8;

    if BYTE<7> == '1' then
        BYTE = ReadAndConsume(8, stream);
        R<I+7:I> = BYTE;
        end = N MOD 7;
        if end == 0 then end = 7;
        if I + 8 > N then SBZ(BYTE<7:end>);

    return R;
```

K10.5.6 ETE programming

K10.5.6.1 Example code sequences

SPWXHJ The enabling sequence should be from the trace sink, such as the trace buffer, to the trace unit. This is to ensure the trace sink is ready to capture trace before the trace unit generates any trace.

SWPVBX The disabling sequence should be from the trace unit to the trace sink. This is to ensure that any buffered trace reaches the trace sink while the trace sink is still enabled.

K10.5.6.1.1 Enabling the trace unit

SXCRYM The following example describes the code sequence to enable the trace unit.

```
    ;; Program the trace unit registers, except TRCPRGCTLR
    ISB                 ;; Synchronize the System Register updates.
    MOV x0, #0x1
    MSR TRCPRGCTLR, x0  ;; Enable the ETE.
    ;; Wait for TRCSTATR.IDLE==0
poll_idle
    ISB                 ;; Synchronize the write to TRCPRGCTLR
    MRS x1, TRCSTATR
    TBNZ x1, #1, poll_idle
```

K10.5.6.1.2 Disabling the trace unit

SVRJNB The following example describes the code sequence to disable the trace unit.

```
    STP x0, x1, [sp, #-16]!

    MRS x0, TRFCR_EL1    ;; Save the current programming of TRFCR_EL1.
    MOV x1, #0x3
    BIC x1, x0, x1
    MSR TRFCR_EL1, x1    ;; Clear the values of TRFCR_EL1.ExTRE.
                         ;; to put the PE in to a Trace Prohibited region
    ISB                  ;; Synchronize the entry to the Trace Prohibited region
    TSB CSYNC            ;; Ensure that all trace has reached the
                         ;; trace buffer and address translations have
                         ;; taken place.
    MOV x1, #0x0
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    MSR TRCPRGCTLR, x1   ;; Disable the trace unit
    ;; Wait for TRCSTATR.IDLE==1 and TRCSTATR.PMSTABLE==1
poll_idle
    ISB
    MRS x1, TRCSTATR
    AND x1, x1, #3
    CMP x1, #3
    B.NE poll_idle

    MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.

    LDP x0, x1, [sp], #16
```

K10.5.6.1.3 Example save restore routine

SXJRPQ The following example describes the code sequence for saving the trace unit state over a power down.

```
    STP x0, x1, [sp, #-16]!
    ;; Enter a Trace Prohibited region
    MRS x0, TRFCR_EL1    ;; Save the current programming of TRFCR_EL1.
    MOV x1, #0x3
    BIC x1, x0, x1
    MSR TRFCR_EL1, x1    ;; Clear the values of TRFCR_EL1.ExTRE.
    ISB                  ;; Synchronizes the entry to the Trace Prohibited region
    TSB CSYNC            ;; Ensure the trace unit is synchronized
    MOV x1, #1
    MSR OSLAR_EL1, x1    ;; Lock the OS lock
    ;; Wait for TRCSTATR.PMSTABLE==1
poll_pmstable
    ISB
    MRS x1, TRCSTATR
    TST x1, #2
    B.EQ poll_pmstable

    MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.

<save the trace unit registers, including TRCPRGCTLR>

    ;; Wait for TRCSTATR.IDLE==1
poll_idle
    ISB
    MRS x1, TRCSTATR
    TST x1, #1
    B.EQ poll_idle

    LDP x0, x1, [sp], #16
```

SDNGPX The following example describes the code sequence for restoring the trace unit state when power is restored.

```
<restore the trace unit registers, including TRCPRGCTLR>

    STP x0, x1, [sp, #-16]!
    MOV x0, #0
    MSR OSLAR_EL1, x0    ;; Clear the OS lock
    LDP x0, x1, [sp], #16
    ISB
```

IDHQGM When programming the trace unit, it is important to be aware that the loops that poll TRCSTATR in Figure D4-2 might never complete. Arm recommends that such scenarios are avoided except in rare conditions. However, some system conditions might prevent a trace unit from either leaving the idle state or becoming idle. In particular, a trace unit might never become idle if the trace unit is unable to output all trace due to a system condition.

STTPKR If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.6.2 Minimal programming

IPDMBT

Table K10-3 gives the values for programming the trace unit to enable tracing of a single process at Non-secure EL0. When FEAT_RME is implemented, this will enable tracing for both Non-secure EL0 and Realm EL0.

Table K10-3 Minimal programming values

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • The return stack. • Global timestamping. • Context identifier tracing. • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace
TRCTRACEIDR	Nonzero	Set a value for the trace ID
TRCTSCTLR	`0x00000000`	Disable the timestamp event The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL1 in Realm state. • EL2 in Realm state. • EL2-EL0 in Secure state. • EL3.
TRCVIIECTLR	`0x00000000`	No address range filtering for logic started
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst

SSXJLN

Disabling tracing of Secure state might not be strictly necessary as secure tracing might be disabled by MDCR_EL3.STE, but Arm recommends not enabling trace for un-required Exception levels, to limit the amount of trace.

SQCWTJ

Disabling tracing of EL1 and EL2 of Non-secure state might not be strictly necessary as non-secure tracing might be disabled by TRFCR_EL2.E2TRE and TRFCR_EL1.E1TRE, but Arm recommends not enabling trace for un-required Exception levels to limit the amount of trace.

K10.5.6.3 Filtering models

IVDBGM

Different trace applications require different usage models of a trace unit. For example, one trace application might only require basic program flow trace, whereas another might require tracing of a specific program function.

The ETE architecture provides for each of these usage models. A trace unit can be implemented with a particular set of implementation options, so that a trade-off between functionality and cost can be achieved in meeting the requirements of a trace application. Discovery of the particular set of implementation is achieved by reading TRCIDR0 to TRCIDR13.

In a trace unit that includes all implementation options, the simplest way to use the trace unit is to turn on tracing of all aspects of PE operation and let the trace analyzer pick out the required information. However, full trace comes at a high

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cost in terms of port bandwidth and trace storage. These costs have an impact on the design of a system, so that a higher pin count and larger buffers might be required.

A trace unit provides on-chip filtering, that facilitates a reduction of the trace bandwidth and therefore provides for a lower system cost. By suspending and enabling trace during a trace that is run to suit the particular requirements of the trace run, the best use of both port bandwidth and trace storage can be made.

The ETE architecture provides the following basic filtering models:

Continuous tracing

This is where no filtering is applied. The following modes can be used:

- Continuous instruction tracing only, where only the instruction trace stream is output.

Instruction-based filtering

This is where instruction tracing, and data tracing if it is implemented and enabled, is active only for certain code sequences, such as for a particular process or function.

For all the possible filtering modes, the trace unit can be programmed before a trace run to enable various options, including:

- Context identifier tracing, if implemented, to indicate to a trace analyzer the Context identifier value.
- Virtual context identifier tracing, if implemented, to distinguish between different virtual machines.
- Cycle counting, to enable a trace analyzer to analyze program performance.
- Global timestamping, if implemented, to enable correlation of the two trace streams with other trace sources in the system.
- Branch broadcasting, if implemented, to force all taken *P0 instruction* targets to be traced with an explicit target address.

A trace unit is programmed for continuous instruction tracing when no filtering is applied to the instruction trace stream.

When a trace unit is programmed for continuous instruction tracing, ViewInst is always active during a trace run. See Filtering trace generation.

K10.5.6.4 Filtering used the exclude function

I_{GFWSJ} The following table describes the programming for excluding a single address range. When FEAT_RME is implemented, this applies to both Non-secure EL0 and Realm EL0.

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • The return stack. • Global timestamping. • Context identifier. • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing.
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented.
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID.
TRCTSCTLR	`0x00000000`	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Value	Description
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started. Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL1 in Realm state. • EL2 in Realm state. • EL2-EL0 in Secure state. • EL3.
TRCVIIECTLR	`0x00010000`	Use ARC0 for the exclude logic.
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst.
TRCACATR0	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR0	*Start Address*	
TRCACATR1	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR1	*End Address*	

K10.5.6.5 Filtering used the include function

IPPMSF

The following table describes the programming for including a single address range. When FEAT_RME is implemented, this applies to both Non-secure EL0 and Realm EL0.

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • The return stack. • Global timestamping. • Context identifier. • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing.
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented.
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID.
TRCTSCTLR	`0x00000000`	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Value	Description
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started. Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL1 in Realm state. • EL2 in Realm state. • EL2-EL0 in Secure state. • EL3.
TRCVIIECTLR	`0x00000001`	Use ARC0 for the include logic.
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst.
TRCACATR0	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR0	*Start Address*	
TRCACATR1	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR1	*End Address*	

K10.5.6.6 OS save and restore routines

SBMBXL When the PE is context switching of trace unit the following registers need to save and restored. Not all these registers are necessarily implemented for all implementations. Please refer to the register description page for information on if the register is implemented.

- TRCPRGCTLR
- TRCCONFIGR
- TRCAUXCTLR
- TRCEVENTCTL0R
- TRCEVENTCTL1R
- TRCRSR
- TRCSTALLCTLR
- TRCTSCTLR
- TRCSYNCPR
- TRCCCCTLR
- TRCBBCTLR
- TRCTRACEIDR
- TRCQCTLR
- TRCVICTLR
- TRCVIIECTLR
- TRCVISSCTLR
- TRCVIPCSSCTLR
- TRCSEQEVR<n>
- TRCSEQRSTEVR
- TRCSEQSTR
- TRCEXTINSELR<n>
- TRCCNTRLDVR<n>
- TRCCNTCTLR<n>
- TRCCNTVR<n>
- TRCIMSPEC<n>
- TRCRSCTLR<n>
- TRCSSCCR<n>
- TRCSSCSR<n>

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCSSPCICR<n>
- TRCACVR<n>[31:0]
- TRCACVR<n>[63:32]
- TRCACATR<n>[31:0]
- TRCACATR<n>[63:32]
- TRCCIDCVR<n>[31:0]
- TRCCIDCVR<n>[63:32]
- TRCVMIDCVR<n>[31:0]
- TRCVMIDCVR<n>[63:32]
- TRCCIDCCTLR0
- TRCCIDCCTLR1
- TRCVMIDCCTLR0
- TRCVMIDCCTLR1

SHYDYT If the trace unit has not been programmed since the last context switch then there is no requirement to save and restore the registers.

K10.5.7 Trace examples

K10.5.7.1 Basic Examples

IXZCWN The following example shows basic program trace.

Table K10-6 Example of program trace

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info(. . .) trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000`	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BEQ` instruction.
`0x2010`	`STR`		
	`IRQ`	exception (IRQ,`0x2014`)	An IRQ occurs. The *Exception element* indicates the `STR` instruction was executed.

IHKPNY The following example shows basic program trace when filtering is applied. In this example, the trace unit is programmed to exclude all instructions in the address range `0x2000` to `0x200F` inclusive, and the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K10-7 Example of program trace with filtering

	Execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	Y	trace_info(. . .) trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`	N		
`0x2004`	`LDR`	N		
`0x2008`	`CMP`	N		
`0x200C`	`BEQ -> 0x3000`	N		
`0x2010`	`STR`	Y	trace_on() address(`0x2010`)	
	`IRQ`		exception (IRQ,`0x2014`)	An IRQ occurs. The *Exception element* indicates the `STR` instruction was executed.

IBXYHM

The example in Table K10-8 shows basic program trace and demonstrates the canceling of some speculative execution because of an exception. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-8 Example of basic program trace when an exception occurs, example one

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - *Context element*. - *Target Address element* The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BEQ` instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	Cancel back to and including `0x2000`	cancel(1)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the three instructions that it implied. Note Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
	IRQ	exception(IRQ,`0x2000`)	The trace unit generates an *Exception element* with the address `0x2000`, which indicates no instructions have executed since the target of the branch at `0x1000`.
	commit all execution	commit(2)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* that was generated for the IRQ exception.

IXJLJN The example in Table K10-9 shows basic program trace, and shows the trace generated when a synchronous Data Abort occurs. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-9 Example of basic program trace when an exception occurs, example two

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - *Context element*. - *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	LDR aborts Cancel back to and including `0x2004`	cancel(1) exception_(data fault, `0x2004`)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the four instructions that it implied. Note Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction. The data fault exception occurred at `0x2004`. The *Exception element* indicates the `MOV` instruction at `0x2000` was executed. In summary: 1. The `MOV` instruction was first implied by the N *Atom element* at `0x200C`. However, the trace analyzer canceled this because of the *Cancel element*. 2. The `MOV` instruction is now implied by the *Exception element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
	commit all execution	commit(2)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* that was generated for the IRQ exception.

I_{DLXSS}

The example in Table K10-10 extends the example shown in Table K10-9, and shows how exceptions are traced when two exceptions occur without any execution between them. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-10 Example of basic program trace when two consecutive exceptions occur

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - *Context element*. - *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as P0 elements, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	LDR aborts Cancel back to and including `0x2004`	cancel(1) exception_(data fault, `0x2004`)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the four instructions that it implied. Note Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction. The data fault exception occurred at `0x2004`. The *Exception element* indicates the `MOV` instruction at `0x2000` was executed. In summary: 1. The `MOV` instruction was first implied by the N *Atom element* at `0x200C`. However, the trace analyzer canceled this because of the *Cancel element*. 2. The `MOV` instruction is now implied by the *Exception element*.
	IRQ	address(`0x4000`) exception(IRQ,`0x4000`)	This *Exception element* contains the address of the exception vector of the DataFault exception. This implies that no instructions have executed since the DataFault exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
	commit all execution	commit(3)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* generated for the Data fault exception and the *Exception element* that was generated for the IRQ exception.

K10.5.7.2 Examples of changes in context

I_{RWRWJ} When the PE executes an instruction that changes the execution context, the exact time at which the new element is traced depends on the PE operation after the write. An example of an instruction that changes the execution context is an instruction that writes a value to the CONTEXTIDR_EL1. See Context element for more information about the rules controlling the generation of Context elements. This section provides examples of PE trace that contain changes of execution context to illustrate these rules. This section is split into the following:

- Exception in software executed after context synchronization.
- Exception immediately after ISB.
- Exception immediately before ISB.

I_{GCRKW} Table K10-11 shows a write to the CONTEXTIDR_EL1 register, followed by an `ISB` to synchronize that write, followed by an exception that changes the context again.

Table K10-11 Program trace containing a context changing operation

	Execution	Context	Trace elements	Notes
`0x1000`	`B-> 0x2000`	`0xAA`	trace_info() trace_on() context(`0xAA`) address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MSR CONTEXTIDR`	`0xAA`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context `0xAA` or `0xBB` but they are always traced as occurring from context `0xAA`.
`0x2004`	`ADD`	`0xAA` or `0xBB`		
`0x2008`	`ISB`	`0xAA` or `0xBB`	atom(E)	The trace unit generates an E *Atom element*, because the `ISB` is a Context synchronization event. All execution is traced as executing in context `0xAA`.
`0x200C`	`SUB`	`0xBB`	context(`0xBB`)	A *Context element* is traced to indicate the new context.
	`IRQ`	`0xBB`	exception(IRQ,`0x2010`)	An IRQ exception occurs. The trace unit generates an *Exception element*.
`0x3000`	`B -> 0x4000`	`0xCC`	context(`0xCC`) address(`0x3000`) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by a *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

I_{LSZCM} Table K10-12 shows the same execution as Table K10-11 but the exception occurs one instruction earlier. This means that no execution takes place between the ISB and the exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K10-12 Program trace containing a context changing operation (exception immediately after ISB)

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
0x2008	ISB	0xAA or 0xBB	atom(E)	The trace unit generates an E *Atom element*, because the ISB is a Context synchronization event. All execution is traced as executing in context 0xAA.
	IRQ	0xBB	context(0xBB) exception(IRQ,0x200C)	A *Context element* is traced to indicate the new context. An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by a *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

IPDLNN

Table K10-13 is the same as Table K10-12 but the exception occurs one instruction earlier. This means that the exception occurs before the ISB instruction that was present in previous examples.

Table K10-13 Program trace containing a context changing operation (exception immediately before ISB)

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
	IRQ	0xAA or 0xBB	exception(IRQ,0x2008)	An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by a *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.7.3 An example of the use of the trace unit return stack

IHRHBK This section contains two examples of tracing the same piece of program code. However:

- In the first example the trace unit return stack is disabled.
- In the second example trace unit the return stack is enabled.

The examples demonstrate that use of the trace unit return stack can help to reduce the amount of trace generated. Table K10-14 the first example, and Table K10-15 shows the second example. In these examples, the trace unit is programmed for basic program flow, where only branch instructions are traced as *P0 instructions*, and is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-14 Basic program trace when Branch with Link instructions are executed and the return stack is disabled

	Execution	Trace elements	Notes
`0x1000`	`BL -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BLEQ -> 0x3000` (not taken)		This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BLEQ` instruction.
`0x2010`	`STR`		
`0x2014`	`BX LR`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` and the `BX` instructions.
`0x1004`	`MOV`	address(`0x1004`)	This instruction is not traced as a *P0 element*, therefore no trace element is generated. However, the last instruction executed was a taken indirect branch instruction, so the trace unit generates an *Target Address element* to indicate the target of that branch.
`0x1008`	`B -> 0x4000`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `MOV` instruction at `0x1004` and the `B` instruction.
	commit all execution	commit(4)	This commits all four of the following: • The E *Atom element* generated for the branch at `0x1000`. • The N *Atom element* generated for the branch at `0x200C`. • The E *Atom element* generated for the branch at `0x2014`. • The E *Atom element* generated for the branch at `0x1008`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K10-15 Basic program trace when Branch with Link instructions are executed and the return stack is enabled

	Execution	Trace elements	Notes
`0x1000`	`BL -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*. In addition, because the return stack is enabled, the Branch with Link instruction causes the address `0x1004` to be pushed onto the trace unit return stack.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BLEQ -> 0x3000` (not taken)		This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BLEQ` instruction. Nothing is pushed onto the trace unit return stack because the branch is not taken.
`0x2010`	`STR`		
`0x2014`	`BX LR`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` and the `BX` instructions.
`0x1004`	`MOV`		This instruction is not traced as a *P0 element*, therefore no trace element is generated. The address of this instruction matches the top entry on the trace unit return stack. Therefore, the trace analyzer knows to restart program execution here and an *Target Address element* is not required. The top entry on the return stack, address `0x1004`, is popped from the return stack.
`0x1008`	`B -> 0x4000`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `MOV` instruction at `0x1004` and the `B` instruction.
	commit all execution	commit(4)	This commits all four of the following: • The E *Atom element* generated for the branch at `0x1000`. • The N *Atom element* generated for the branch at `0x200C`. • The E *Atom element* generated for the branch at `0x2014`. • The E *Atom element* generated for the branch at `0x1008`.

K10.5.8 Differences between ETM and ETE

IJNLFZ

ETE has a considerable overlap with the ETMv4 architecture, with the intent that broadly unified software stack can program a trace unit and interpret the trace stream from either an ETMv4 trace unit or an ETE trace unit.

This section describes the primary functional differences between ETMv4 and ETE.

- Removal of data trace documentation, since this is not permitted in A-profile.
- Removal of conditional non-branch documentation, since this is not permitted in A-profile.
- TRCDEVARCH.PRESENT == 1 is mandatory.
- TRCDEVARCH.ARCHVER and TRCDEVARCH.REVISION take new values.
- TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN take new values.
- TRCIDR9 is fixed at zero.
- Context identifier tracing is mandatory, defined in TRCIDR2.CIDSIZE.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Virtual context identifier tracing is mandatory when the PE implements EL2, defined in TRCIDR2.VMIDSIZE.
- The Virtual context identifier is always based on CONTEXTIDR_EL2, with support for tracing VTTBR_EL2.VMID removed.
- 64-bit timestamp is the only supported timestamp size.
- Timestamping is mandatory in ETE.
- TRCIDR2.IASIZE is only permitted to indicate a 64-bit instruction address size.
- External Inputs are unified with the PMU event space, with new TRCEXTINSELR<n> registers introduced.
- TRCIDR5.NUMEXTIN indicates the unified External Input model.
- Added TRCRSR.EXTIN for reading and setting the External Input Selectors state.
- Added TRCRSR.EVENT for reading and setting the *ETEEvent* state.
- Added TRCRSR.TA for reading and setting whether tracing was active.
- Changed requirements for the tracing of Exceptions to be dependent on the new TRCRSR.TA field.
- Removal of memory-mapped accesses. This was deprecated in ETMv4.4 for Armv8-A.
- Removal of trace unit sharing.
- Added a requirement that trace must be output within finite time.
- Added a requirement that the trace unit resources are paused when entering a Trace Prohibited region.
- Added a bit to TRCSSCSR<n> to indicate that the Single-shot Comparator Control fired while the resources are paused.
- Added requirements for dependencies on the `TSB CSYNC` instruction.
- Execution of `TSB CSYNC` instruction requests a timestamp element.
- The Unified Power Domain Model from ETMv4.5 for Armv8-A is mandatory in ETE.
- Changes to the enable and disable code sequences.
- Tightened the requirements for obeying the order of start point and stop points for the ViewInst start/stop function, and tightened the rules for programming the start/stop function.
- Addition of *Source Address elements*.
- Added rules to require no trace to be generated in Trace Prohibited regions, under some circumstances.
- Added constraints for the effect of system instructions causing the trace unit to become enabled or disabled.
- Additional constraints for the forced tracing of exceptions around Trace Prohibited regions, to ensure trace is not generated in Trace Prohibited regions.
- Removed the flexibility around tracing of an *Exceptional occurrence* immediately after a Trace Prohibited region or when trace generation becomes operative. Such *Exceptional occurrences* are not traced.
- Added a requirement that the resource operations must complete before a TSB instruction completes.
- Defined the behavior of the visibility of reads and writes to trace unit registers from system instructions, external debugger and by the trace unit.
- Changed branch broadcasting to be required in all implementations, see TRCIDR0.TRCBB.
- TRCSYNCPR is read/write in all implementations, see TRCIDR3.SYNCPR.
- Forced tracing of System Error exceptions is required in all implementations, see TRCIDR3.TRCERR.
- Changed cycle counting to be required in all implementations, see TRCIDR0.TRCCCI.
- Removed the trace unit OS Lock mechanism, and changed to require the PE OS Lock to affect the trace unit.
- Removed the Exception Return element and Exception Return packet.
- Constrained TRCCLAIMCLR and TRCCLAIMSET to not require explicit synchronization.
- Added more constraints to the operation of the Single-shot Comparator Controls when the trace unit becomes disabled, or when entering a Trace Prohibited region.
- Added more constraints to the operation of the ViewInst start/stop function when the trace unit becomes disabled, or when entering a Trace Prohibited region.
- Constrained the behavior of cycle counting after a trace unit buffer overflow, to require the cycle count to be traced as UNKNOWN on the first *Cycle Count element* after an overflow.
- Export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.
- *Event elements* are permitted to be generated before the first *Trace Info element*.

K10.5.9 Context switching

SVKHHY When switching out a process being traced, to save the current trace context and ensure all trace operations are written to the correct context:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. Prohibit program-flow trace using TRFCR_ELx. In many cases this is done before the process is traced. For example if all of the following are true:
   - TRFCR_EL1.E0TRE is 1 to allow tracing of a process executed at EL0.
   - TRFCR_EL1.E1TRE is 0 to prohibit tracing of the Operating System performing the context switch executed at EL1.
2. Execute a Context synchronization event to guarantee no new program-flow trace is generated. In the common case, this Context synchronization event is an exception taken to an Exception level where tracing is prohibited.
   - If the trace unit is an ETE and the ETE is enabled, this also pauses the ETE Resources.
3. Execute a `TSB CSYNC` instruction to ensure the program-flow trace is flushed.
4. If necessary, disable the trace unit.
   - For an ETE this is necessary if context is being switched. Software must set TRCPRGCTLR.EN to zero. This is necessary as:
     - The ETE must be disabled if saving the ETE state, as the ETE System registers can only be read when the ETE is disabled.
     - ETE trace compression logic is stateful, and disabling the ETE resets this compression state.
5. Disable the Trace Buffer Unit. Set TRBLIMITR_EL1.E to zero.
   - This must be done before changing the VMSA System registers to prevent the Trace Buffer Unit from speculatively accessing translation table entries.
6. Execute a `DSB` operation.
   - This is required if software will be reading the trace buffer contents, to ensure the writes to memory are Complete.
7. Execute a further Context synchronization event.
   - This is required to synchronize the effects of any System register writes since the previous Context synchronization event.
   - This is also required if software will be reading the Trace Buffer Unit or trace unit System registers as part of the context switch, to capture indirect writes to those registers by trace operations synchronized by the `TSB CSYNC`.
   - For a subsequent direct read to capture the indirect write to TRBSR_EL1 resulting from an External abort on a completed write, this Context synchronization event must follow the `DSB` above.
8. Save and/or change the context. For example, save the MDCR_EL3, MDCR_EL2 (if applicable), Trace Buffer Unit, trace unit, and TRFCR_ELx System registers, and update the VMSA System registers for the new process.

S_{FMBCL}

In other uses cases, tracing is not prohibited when software wants to save the trace context. For this case, if using an ETE, the sequence is slightly different:

1. Disable the ETE. Set TRCPRGCTLR.EN to zero.
2. Execute a Context synchronization event to guarantee no new program-flow trace is generated.
3. Execute a `TSB CSYNC` instruction to ensure the program-flow trace is flushed.
4. Execute a `DSB` and/or Context synchronization event as required by the previous example.
5. Save and/or change the context.

For an ETE this sequence does not guarantee that all instructions before disabling the ETE are traced. The ETE might discard trace for preceding instructions when it is disabled.

S_{PKLXF}

To restore the state of the Trace Buffer Unit and trace unit for switching in a process being traced, while tracing is prohibited:

1. Restore the context. For example:
   - Restore MDCR_EL3, MDCR_EL2 (if applicable), and the Trace Buffer Unit System registers, other than TRBLIMITR_EL1.
   - Restore the trace unit System registers, other than enabling the trace unit.
   - Ensure the TRFCR_ELx System registers are correct for the process being traced.
   - Update VMSA System registers for the process being returned to.
2. Execute a Context synchronization event to guarantee the trace unit and Trace Buffer Unit will observe the new values of the System registers written by the previous step.
3. Enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1. This must be done after setting up the correct VMSA System registers for the trace buffer, as the Trace Buffer Unit might now speculatively prefetch and cache address translations. See R_{BSMLW} and S_{YKCND}.
4. If necessary, enable the trace unit. If using an ETE, software must set TRCPRGCTLR.EN to one.
5. Execute a Context synchronization event to guarantee tracing is allowed. In the common case, this is an `ERET` instruction that returns to a different Exception level where tracing is allowed.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This must be done after saving the state from the previous process, if applicable.

SYKCND Because the Trace Buffer Unit can prefetch and cache address translations when the Trace Buffer Unit is enabled:

- Software must not enable the Trace Buffer Unit before programming the System registers for the owning translation regime. In particular, during a context switch operation:
  - If switching from a context using the Trace Buffer Unit, the Trace Buffer Unit must be disabled before modifying the System registers for the owning translation regime being switched from.
  - If switching to a context using the Trace Buffer Unit, the Trace Buffer Unit must not be enabled until after modifying the System registers for the owning translation regime being switched to.
- The Trace Buffer Unit must not be enabled when the PE is not executing in the owning Security state or when executing at EL3 and SCR_EL3.NS does not indicate the owning Security state.
- In normal conditions, enabling the Trace Buffer Unit early before returning to the context being traced might be advantageous if the implementation does prefetch address translations.

K10.5.10 Controlling generation of trace buffer management events

SNFQZJ The TRBE does not include a direct capability to program the Trace Buffer Unit to generate a maintenance interrupt when the trace buffer reaches a programmed level below the Limit pointer, and continue collecting trace until either the interrupt is serviced or (possibly) the trace buffer fills (whichever comes first). This allows an almost lossless collection of trace.

However, the Trace Buffer Unit can be programmed to give similar behavior in one of the following ways:

1. Using Wrap mode. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, but set the trace buffer mode to Wrap mode the current write pointer to point part way through the trace buffer, such that the remaining space in the trace buffer is the watermark level. When the amount of trace collected reaches the watermark level, the current write pointer is wrapped and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:
   - The trace buffer management event is generated and the trace unit receives the TRB_WRAP event at the watermark level.
   - The oldest trace in the trace buffer will be lost if more trace is generated than fits in the trace buffer, because it is overwritten by newer trace. Note that some loss of trace is inevitable if more trace is generated than fits in the trace buffer.
   - The trace history does not start at the start of the trace buffer, and must be aligned by software.
2. Use a *Trigger Event*. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, and set the trace buffer mode to Fill mode and the current write pointer to the start of the trace buffer. Set the trigger mode to *IRQ on trigger*, the *Trigger Counter* to the watermark level, and TRBSR_EL1.TRG to 1. When the amount of trace collected reaches the watermark level, a *Trigger Event* occurs and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:
   - The trace buffer management event is generated and the trace unit receives the TRB_TRIG event at the watermark level.
   - The newest trace in the trace buffer will be discarded if more trace is generated than fits in the trace buffer. To overwrite the oldest trace instead, set the trace buffer mode to Circular Buffer mode.
   - This method cannot be used if also searching for a *Detected Trigger* event from the trace unit.
   - The current write pointer does not have to be set to the start of the trace buffer. If the trace buffer already contains data that software does not want to be overwritten, the current write pointer can be set to point to after this data. In this case using Circular Buffer mode or *Stop on trigger* can also be used to control when *collection is stopped* and what data is overwritten.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.6 Use of the GCS Extension

This appendix describes the example codes for the Guarded Control Stack. It contains the following sections:

- Recording the call stacks from the current PE.
- Recording the call stacks from a different PE.
- Overwriting a GCS record from a higher Exception level.
- Thread migration between PEs.
- Switching EL0 Guarded Control Stacks from EL1.
- Synchronization of GCS accesses.

K10.6.1 Recording the call stacks from the current PE

SNNHCN

Example K10-4 shows an code sequence whereby code running on a PE can inspect the contents of the current GCS, and the synchronization that is required as part of that process.

Instructions 1, 7 and 8 are part of the program being recorded. All other instructions are part of Call stack recording software.

Example K10-4

```
S.No.
 Instruction
1       BL                  ; Produces a store to an address 0x100.
2       MRS x0,GCSPR_EL0
3       GCSB DSYNC          ; The Memory effect from instruction 1 is Ordered-before
                            ; the Memory effect from instruction 4 as both access the same Location.
4       LDR x1,[x0]         ; Load from the address 0x100.
5       STR x1,[x2]         ; Store the Call stack elsewhere.
6       GCSB DSYNC          ; The Memory effect from instruction 4 is Ordered-before
                            ; the Memory effect from instruction 7 as both access the same Location.
7       RET                 ; Produces a load from the address 0x100.
8       BL                  ; Produces a store to the address 0x100.
```

K10.6.2 Recording the call stacks from a different PE

SMLWCC

Example K10-5 shows an code sequence of code running on PE1 where the call stack is being recorded by the GCS, and PE2 is consuming the call stack from the GCS.

Instructions 1, 9 and 10 from PE1 are part of the program being recorded. All other instructions are part of Call stack recording software.

x1 is a pointer to a shared variable whose initial value is 0.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-5

PE1:

```
S.No.
 Instruction
1       BL              ; Produces a store to an address 0x100
2       GCSB DSYNC      ; Instructions which observe the Memory effect from instruction 4
                        ; also observe the Memory effect from instruction 1
3       MOV x0,#1
4       STLR x0,[x1]    ; Set flag to 1
  loop1:
5       LDAR x0,[x1]
6       SUB x0,x0,#0x1
7       CBZ loop1
8       GCSB DSYNC      ; Instructions with Memory effects that are Ordered-before
                        ; Memory effects from instruction 5 are also Ordered-before
                        ; Memory effects from instruction 10.
9       RET             ; Produces a load from the address 0x100
10      BL              ; Produces a store to the address 0x100
```

PE2:

```
S.No.
 Instruction
  loop2:
1       LDAR x0,[x1]
2       CBZ loop2
3       LDR             ; Load from the address 0x100
4       STR             ; Store the Call stack elsewhere
5       MOV x0,#2
6       STLR x0,[x1]    ; Set flag to 2
```

K10.6.3 Overwriting a GCS record from a higher Exception level

SKPMKV

Example K10-6 shows an code sequence where a EL0 GCS record is modified by code running at a higher Exception level.

Instructions 1, 2, 11 and 12 are executed at EL0. Execution of instruction 3 is attempted from EL0 and a synchronous exception is generated. All other instructions are executed at EL1.

Example K10-6

```
S.No.
 Instruction or Exception
1       BL              ; Produces a store to an address 0x100
2       <program legitimately increments the LR by 4>
3       RET             ; Causes an exception as LR did not match the GCS record
4       MRS x0,GCSPR_EL0
5       GCSB DSYNC      ; The Memory effect of instructions 1, 3 are Ordered-before
                        ; the Memory effect of instructions 6, 8
6       LDR x1,[x0]     ; Load from the address 0x100
7       ADD x1,x1,#0x4  ; Increment the loaded value by 4
8       STR x1,[x0]     ; Store the incremented value at the address 0x100
9       GCSB DSYNC      ; The Memory effect of instructions 6, 8 are Ordered-before
                        ; the Memory effect of instructions 11, 12
10      ERET
11      RET             ; Observes the store from instruction 8
12      BL              ; Produces a store to the address 0x100
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.6.4 Thread migration between PEs

SQDRRJ

Example K10-7 shows an code sequence whereby Stack A is migrated from PE1 to PE2, entirely in EL0 code.

x1 is a pointer to a shared variable whose initial value is 0, x3 is a pointer to a shared variable where the address of Stack A is stored, and x2 is a pointer to the head of Stack B which is migrated in to PE1.

Example K10-7

PE1:

```
S.No.
 Instruction
1       BL              ; Produces a store to stack A
                        ; GCSPR_EL0 is 0x1000 after instruction 1
2       MOV  x2,#0x8000 ; Stack B starts at 0x8000
3       GCSSS1 x2       ; x2 is input and points to stack B
4       GCSSS2 x2       ; x2 is output and contains 0xFF8 after this instruction
                        ; Instruction 4 includes a GCSB effect
                        ; so that stack A contents are
                        ; visible to PE2
5       STR  x2,[x3]    ; Store stack A address
6       MOV  x0,#0x1
7       STLR x0,[x1]    ; set flag to 1
8       BL              ; Produces a store to stack B
```

PE2:

```
S.No.
 Instruction
1       BL              ; Produces a store to stack C
   loop:
2       LDAR x0,[x1]
3       CBZ loop
4       LDR  x2,[x3]    ; Load stack A address, x2 will be 0xFF8 after this
                        ; instruction
5       GCSSS1 x2       ; x2 is input and points to stack A
6       GCSSS2 x2       ; x2 is output and points to stack C
                        ; GCSPR_EL0 is 0x1000 after instruction 6
                        ; Instruction 6 includes a GCSB effect
                        ; so that stack A contents are
                        ; visible to PE2
7       RET             ; Load from stack A
                        ; Observes the store from instruction 1 on PE1
```

K10.6.5 Switching EL0 Guarded Control Stacks from EL1

SVSNDV

Example K10-8 and Example K10-9 show code sequences for EL1 software switching the Guarded Control Stack of EL0 tasks. The EL1 software switches out Task 1 prior to confirming that Task 2 is ready for switching.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-8 Switching out an EL0 stack prior to synchronizing incoming stack

```
S.No.
 Instruction
       <Task 1 is executed at EL0 and an exception is taken to EL1>
1      ADR  x1, PCB1           ; Load the address of a memory structure that
                               ; Stores the context of Task 1
2      MRS  x0, GCSCRE0_EL1
3      STR  x0,[x1],#0x8       ; Save GCSCRE0_EL1 of Task 1
4      MRS  x0, GCSPR_EL0
5      STR  x0,[x1],#0x8       ; Save GCSPR_EL0 of Task 1
6      GCSB DSYNC              ; Synchronize GCS data of Task 1
7      MOV  x0,#0x1
8      STLR x0,[x1]            ; Save a flag to indicate that Task 1 is switched out
9      ADR  x2, PCB2           ; Load the address of a memory structure that
                               ; Stores the context of Task 2
10 loop: LDAR x0,[x2,#0x10]
11     CBZ  x0, loop           ; Wait till Task 2 is switched out by some other PE
12     GCSB DSYNC              ; Synchronize GCS data of Task 2
13     LDR  x0,[x2],#0x8
14     MSR  GCSCRE0_EL1,x0     ; Set GCSCRE0_EL1 of Task 2
15     LDR  x0,[x2],#0x8
16     MSR  GCSPR_EL0,x0       ; Set GCSPR_EL0 of Task 2
17     ISB
       <An ERET will be executed to resume EL0 execution of Task 2>
```

Example K10-9 Switching out an EL0 stack after synchronizing incoming stack

```
S.No.
 Instruction
       <Task 1 is executed at EL0 and an exception is taken to EL1>
1      ADR  x2, PCB2           ; Load the address of a memory structure that
                               ; Stores the context of Task 2
2 loop: LDAR x0,[x2,#0x10]
3      CBZ  x0, loop           ; Wait till Task 2 is switched out by some other PE
4      GCSB DSYNC              ; Synchronize GCS data of Task 1 and Task 2
5      ADR  x1, PCB1           ; Load the address of a memory structure that
                               ; Stores the context of Task 1
6      MRS  x0, GCSCRE0_EL1
7      STR  x0,[x1],#0x8       ; Save GCSCRE0_EL1 of Task 1
8      MRS  x0, GCSPR_EL0
9      STR  x0,[x1],#0x8       ; Save GCSPR_EL0 of Task 1
10     MOV  x0,#0x1
11     STLR x0,[x1]            ; Save a flag to indicate that Task 1 is switched out
12     LDR  x0,[x2],#0x8
13     MSR  GCSCRE0_EL1,x0     ; Set GCSCRE0_EL1 of Task 2
14     LDR  x0,[x2],#0x8
15     MSR  GCSPR_EL0,x0       ; Set GCSPR_EL0 of Task 2
16     ISB
       <An ERET will be executed to resume EL0 execution of Task 2>
```

K10.6.6 Synchronization of GCS accesses

I_{BSXKN} In Example K10-10, the store access from the `STR` instruction is observed by the GCS load access from the `RET` instruction, by virtue of the `GCSB` instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-10

```
STR             ; Store to an address 0x100
GCSB DSYNC      ; Produce a GCSB effect
RET             ; Load from the address 0x100
```

I_{JVVRW} In Example K10-11, the GCS store access from the `BL` instruction is observed by the GCS load access from the `RET` instruction without requiring any explicit synchronization.

Example K10-11

```
BL     ; Store to an address 0x100
ADD
RET    ; Load from the address 0x100
```

I_{ZFDQL} In Example K10-12, the atomic GCS access from the `GCSSS1` instruction is observed by the GCS load access from the `GCSSS2` instruction without requiring any explicit synchronization.

Example K10-12

```
GCSSS1       ; Store to an address 0x100
GCSSS2       ; Load from the address 0x100
```

I_{HKFDQ} In Example K10-13, the GCS store access from the `BL` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instructions and the ordering ensured by the `STLR` and `LDAR` instructions.

Example K10-13

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC      ; Produce a GCSB effect
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSB DSYNC      ; Produce a GCSB effect
    RET             ; Load from the address 0x100
```

I_{WFPWV} In Example K10-14, the GCS store access from the `BL` instruction on PE1 is observed by the GCS accesses from the `GCSSS1` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-14

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC      ; Produce a GCSB effect
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSSS1          ; Load from the address 0x100
```

I_{DBSTS}

In Example K10-15, the GCS store access from the `BL` instruction is observed by the load access from the `LDR` instruction, by virtue of the `GCSB` instruction.

Example K10-15

```
BL              ; Store to an address 0x100
GCSB DSYNC      ; Produce a GCSB effect
MOV x3,#0x100
LDR x2,[x3]     ; Load from the address 0x100
```

I_{PRKMB}

In Example K10-16, the GCS store access from the `BL` instruction on PE1 is observed by the load access from the LDR instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Example K10-16

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC      ; Produce a GCSB effect
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    MOV x3,#0x100
    LDR x2,[x3]     ; Load from the address 0x100
```

I_{RCSNZ}

In Example K10-17, the GCS store access from the `GCSSS1` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-17

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    GCSSS1          ; Store to an address 0x100
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSB DSYNC      ; Produce a GCSB effect
    RET             ; Load from the address 0x100
```

I_{FDQNJ} In Example K10-18, the store access from the `STR` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Example K10-18

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    STR             ; Store to an address 0x100
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSB DSYNC      ; Produce a GCSB effect
    RET             ; Load from the address 0x100
```

I_{WLGDG} In Example K10-19, the GCS store access from the `BL` instruction on PE1 is observed by the load access from the `LDR` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the `DSB` instruction ensuring completion of the store.

Example K10-19

```
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC
    DSB
    <raise an IRQ to PE2>
PE2:
    <IRQ exception occurred>
    LDR  x1,[x0]    ; Read from the address 0x100
```

I_{KWRFP} In Example K10-20, the GCS store access from the `BL` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instructions and the `DSB` ensuring completion of the store.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-20

```
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC
    DSB
    <raise an IRQ to PE2>
PE2:
    <IRQ exception occurred>
    GCSB DSYNC
    RET             ; Read from the address 0x100
```

I_{KXZTV} In Example K10-21, the GCS store access from the `BL` instruction on PE1 is observed by the load access from the `LDR` instruction on PE2, on the assumption that the `TLBI` operation is observed by PE1 after the `BL`, and that the `TLBI` operation is completed by the `DSB` instruction on PE2.

Example K10-21

```
PE1:
    BL              ; Store to an address 0x100
    <TLBI operation has been received from PE2>
PE2:
    TLBI            ; Invalidate the address 0x100
    DSB
    LDR  x1,[x0]    ; Read from the address 0x100
```

I_{CHFPV} In Example K10-22, the `DC IVAC` instruction in the following sequence does not invalidate the data produced by the `BL` instruction:

Example K10-22

```
PE1:
    MRS x0,GCSPR_ELx
    SUB x0,x0,#-8
    DC IVAC, x0     ; Data cache invalidation for the address 0x100
    BL              ; Store to an address 0x100
```

K10.6.6.1 Interaction with non-coherent observers

I_{WCSFM} The GCS store access from the `BL` instruction on PE1 is observed by the load access from the `LDR` instruction on, by virtue of the ordering induced by the `DC` and `DMB` instructions on PE1 and the `LDAR` on E1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-23

```
PE1:
    BL              ; Store to an address 0x100
    MRS x0,GCSPR_ELx
    DC CVAC, x0     ; Data cache clean for the address 0x100
    DMB
    STR x1,[x4]     ; Send a flag to E1
E1:
loop:
    LDAR x2,[x4]    ; Wait for the flag from PE1
    CBZ  x2,loop
    LDR  x3,[x0]    ; Read from the address 0x100
```

IYMVPM

In Example K10-24, the `STR` instruction from a non-coherent observer E1 in the following sequence is observed by the `RET` instruction on PE1.

Example K10-24

```
E1:
    STR  x1,[x0]    ; Write to an address 0x100
    DC CVAC, x0
    DMB
    STR x2,[x4]     ; Send a flag to PE1
PE1:
loop:
    LDAR x3,[x4]    ; Wait for the flag from E1
    CBZ  x3,loop
    DMB             ; Ensure LDAR and DC IVAC are ordered
    DC IVAC, x0     ; Data cache invalidation for the address 0x100
    RET             ; Load from the address 0x100
```

IJPQYQ

In Example K10-25, `the Memory Write effect from the STR` instruction from a non-coherent observer E1 in the following sequence is observed by the GCS Memory Read effect from the `RET` instruction on PE2. PE1 and PE2 are in a same Inner Shareable domain, and the address `0x100` is marked with the Inner Shareable attribute.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-25

```
E1:
    STR  x1,[x0]    ; Write to an address 0x100
    DC CVAC, x0
    DMB
    MOV  x2,#1
    STR x2,[x4]     ; Send a flag to PE1
PE1:
loop:
    LDAR x3,[x4]    ; Wait for the flag from E1
    CBZ  x3,loop
    DC IVAC, x0
    DMB             ; Ensure DC IVAC and STR are ordered
    MOV  x2,#2
    STR x2,[x4]     ; Send a flag to PE2
PE2:
loop:
    LDAR x3,[x4]    ; Wait for the flag from PE1
    SUB  x3,x3,#2
    CBNZ x3,loop
    GCSB DSYNC      ; Synchronize GCS accesses
    RET             ; Load from the address 0x100
```

I_{BWBVR}

In Example K10-26, the GCS Memory Write effect from the `BL` instruction on PE1 is observed by the `Memory Read effect from the LDR` instruction from a non-coherent observer E1 in the following sequence. PE1 and PE2 are in a same Inner Shareable domain, and the address `0x100` is marked with the Inner Shareable attribute.

Example K10-26

```
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC
    MRS x0,GCSPR_ELx
    MOV  x2,#1
    STLR x1,[x4]    ; Send a flag to E1
PE2:
loop:
    LDAR x2,[x4]    ; Wait for the flag from PE1
    CBZ  x2,loop
    DMB
    DC CVAC, x0     ; Clean for the address 0x100
    DMB
    MOV  x2,#2
    STR x2,[x4]     ; Send a flag to PE2
E1:
loop:
    LDAR x3,[x4]    ; Wait for the flag from PE1
    SUB  x3,x3,#2
    CBNZ  x3,loop
    LDR  x3,[x0]    ; Read from the address 0x100
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.7 Use of the RAS Extension

K10.7.1 Using DC {G}ZVA and DC CI{GD}VAPS to remove poison

ITFLPJ The following Software usage statements detail possible mechanisms used to remove poison from a Location. For more information, see Clearing Deferred errors from poisoned Locations.

SSLHGP An error exception is generated when the PE consumes a poisoned Location. This might be a synchronous External Abort exception or an SError exception, depending on the implementation.

If software is able to recover from the exception, it will place the translation granule into quarantine by logging it as having a latent fault, and will not use it again.

If software needs to remove a translation granule from quarantine, it must remove the poison from the poisoned Location. This might be done directly by using architecture-defined mechanisms defined below, or accelerated by an IMPLEMENTATION DEFINED peripheral abstracted by a firmware call or other software interface.

SFCDTN Software can remove poison by using the following steps:

1. Discovering from firmware tables whether the `DC ZVA` or, if FEAT_MTE2 is implemented, `DC GZVA` instruction is poison-atomic for the region of physical memory containing the poisoned Location.
2. If `DC {G}ZVA` is poison-atomic for the region, using one or more `DC {G}ZVA` instructions to remove the poison.

Otherwise, it is IMPLEMENTATION DEFINED whether an IMPLEMENTATION DEFINED mechanism exists to remove poison from a Location, including cached copies of the Location.

It might be acceptable to reuse the memory at this point, or wait some time for the Location to naturally be written back to memory, or be pushed to memory by an IMPLEMENTATION DEFINED mechanism.

The Location might remain in other caches, meaning that even if a `DC {G}ZVA` instruction has removed the poisoned copy, other copies might mask a persistent fault in main memory for the Location, until the Location is cleaned and invalidated from all caches and read again from main memory.

STWYNP If FEAT_PoPS is implemented, software can use a sequence of `DC CI{GD}VAPS` and `IC IVAU` instructions to ensure that all copies of a poisoned Location are removed after attempting to remove poison by using a poison-atomic `DC {G}ZVA` instruction or an IMPLEMENTATION DEFINED mechanism.

IYTKLZ The `DC {G}ZVA` instructions write zeros to the Location. Any value can be written to the poison granule to remove poison if the write is poison-atomic.

Software might want to vary the value, for example to test the poison granule for stuck-at-one or stuck-at-zero faults.

However, Arm recommends that software uses `DC {G}ZVA` instructions first, if they are poison-atomic, and then overwrites the zeros with any other test pattern as necessary.

SRDSYX After performing a clean and invalidate, software can verify that poison has been removed with a Read of the Location. The clean and invalidate of the Location is required to be Ordered-before the Memory Read Effect.

If the Read does not read back the value written to the Location, this might be due to one of the following:

- The Location remains poisoned.
- The error is persistent.
- If FEAT_MEC is implemented, different keys were used for the write and subsequent read.
- The Location was accessed with mismatched attributes.

If the error condition persists after attempting to clear the fault again, then the translation granule should be returned to quarantine.

Software may choose to implement different levels of quarantine, to avoid repeatedly attempting to remove poison from a Location with a known persistent error. That is, treat the translation granule as permanently corrupted and off-line until the next system reboot. Software might want to analyze these Locations further to diagnose the fault.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.8 Example OS Save and Restore sequences

This appendix provides possible OS Save and Restore sequences for a Debug implementation using the V8A or later architectures. It contains the following sections:

- Save Debug registers.
- Restore Debug registers.

K10.8.1 Save Debug registers

This section shows how to save the registers that are used by an external debugger.

```
; On entry, X0 points to a block to save the debug registers in.
; Returns the pointer beyond the block and corrupts X1-X3

SaveDebugRegisters
    ; (1) Set OS Lock.
    MOV     X2,#1                       ; Set the OS Lock. In AArch64 state, the OS Lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization event

    ; (2) Walk through the registers, saving them
    MRS     X1,OSDTRRX_EL1              ; Read DTRRX
    MRS     X2,OSDTRTX_EL1              ; Read DTRTX
    STP     W1,W2,[X0],#8               ; Save { DTRRX, DTRTX }
    MRS     X1,OSECCR_EL1               ; Read ECCR
    MRS     X2,MDSCR_EL1                ; Read DSCR
    STP     W1,W2,[X0],#8               ; Save { ECCR, DSCR }
    [ AARCH32_SUPPORTED
    MRS     X1,DBGVCR32_EL2             ; Read DBGVCR
    MRS     X2,DBGCLAIMCLR_EL1          ; Read CLAIM - note, have to read via CLAIMCLR
    STP     W1,W2,[X0],#8               ; Save { VCR, CLAIM }
    ]

    ;; Macros for saving off a "register pair"
    ;; $WB      is W for watchpoint, B for breakpoint
    ;; $num     is the pair's number
    ;; X0 contains a pointer for the value words
    ;; X1 contains a pointer for the control words
    ;; W2 contains the max index
    MACRO
    SaveRP  $WB,$num, $exit
    MRS     X3,DBG$WB.VR$num._EL1       ; Read DBGxVRn
    STR     X3,[X0],#8                  ; Save { xVRn }
    MRS     X3,DBG$WB.CR$num._EL1       ; Read DBGxCRn
    STR     W3,[X0],#4                  ; Save { xCRn }.
    [ $num > 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
    MACRO
    SaveBRP $num                        ; Save a Breakpoint Register Pair
    SaveRP  B,$num,SaveDebugRegisters_Watchpoints
    MEND
    SaveBRP 0
    SaveBRP 1
    SaveBRP 2
    ;; and so on to ...
    SaveBRP 15
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SaveDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    SaveWRP $num                        ; Save a Watchpoint Register Pair
    SaveRP  W,$num,SaveDebugRegisters_Exit
    MEND
    SaveWRP 0
    SaveWRP 1
    SaveWRP 2
    ;; and so on to ...
    SaveWRP 15

SaveDebugRegisters_Exit
    ; (5) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function. The OS double-lock (OSDLR_EL1.DLK) is
    ; locked later, just before the final entry to WFI state.
    RET
```

K10.8.2 Restore Debug registers

This section shows how to restore the registers that are used by an external debugger.

```
; On entry, X0 points to a block of saved debug registers.
; Returns the pointer beyond the block and corrupts R1-R3,R12.

RestoreDebugRegisters
    ; (1) Lock OS Lock. The lock will already be set, but this write is included to ensure it
    ; is locked.
    MOV     X2,#1                       ; Lock the OS Lock. In AArch64 state, the OS Lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization event

    MSR MDSCR_EL1, XZR                  ; Initialize MDSCR_EL1

    ; (2) Walk through the registers, restoring them
    LDP     W1,W2,[X0],#8               ; Read { DTRRX,DTRTX }
    MSR     OSDTRRX_EL1,X1              ; Restore DTRRX
    MSR     OSDTRTX_EL1,X2              ; Restore DTRTX
    LDP     W1,W3,[X0],#8               ; Read { DSCR, ECCR }
    MSR     OSECCR_EL1,X2               ; Restore ECCR
    [ AARCH32_SUPPORTED
    LDP     W1,W2,[X0],#8               ; Read { VCR, CLAIM }
    MSR     DBGVCR32_EL2,X1             ; Restore DBGVCR
    MSR     DBGCLAIMSET_EL1,X2          ; Restore CLAIM - note, writes CLAIMSET
    ]

    ;; Macro for restoring a "register pair"
    MACRO
    RestoreRP $WB,$num,$exit
    LDR     X3,[X0],#8                  ; Read { xVRn }
    MSR     DBG$WB.VR$num._EL1,X3       ; Restore DBGxVRn
    LDR     W3,[X0],#4                  ; Read { xCRn }
    MSR     DBG$WB.CR$num._EL1,X3       ; Restore DBGxCRn
    [ $num >= 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    MACRO
    RestoreBRP $num                     ; Restore a Breakpoint Register Pair
    RestoreRP  B,$num,RestoreDebugRegisters_Watchpoints
    MEND
    RestoreBRP 0
    RestoreBRP 1
    RestoreBRP 2
    ;; and so on until ...
    RestoreBRP 15

RestoreDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    RestoreWRP $num                     ; Restore a Watchpoint Register Pair
    RestoreRP  W,$num,RestoreDebugRegisters_Exit
    MEND
    RestoreWRP 0
    RestoreWRP 1
    RestoreWRP 2
    ;; and so on until ...
    RestoreWRP 15

RestoreDebugRegisters_Exit
    MSR MDSCR_EL1, X3                   ; Restore DSCR

    ; (5) Clear the OS Lock.
    ISB
    MOV     X2,#0                       ; Clear the OS Lock. In AArch64 state, the OS Lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.

    ; (6) A final ISB guarantees the restored register values are visible to subsequent
    ; instructions.
    ISB

    ; (7) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function.
    RET
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.9 RME example software flows

This appendix describes example software flows for RME.

It contains the following sections:

- Granule Transition Flow
- Procedures for changing the size of a GPT contiguous region

K10.9.1 Granule Transition Flow

The properties of the *Granule Transition Flow* (GTF) are:

1. The GTF changes the *Physical Address Space* (PAS) association of a physical granule from a previous physical address space to a new physical address space.
2. The GTF completes when the association of the physical granule with the new physical address space is observable.
3. When the GTF completes, the following outcomes are guaranteed:
   - Writes to the previous physical address space do not become observable.
   - If the previous physical address space is Realm or Secure, then no accesses, including Speculative read accesses, to the previous physical address space can observe unscrubbed values from before the GTF.
   - If the previous physical address space is Realm or Secure then no instructions, including execution under Speculation, can observe unscrubbed values from before the GTF.
   - If the previous physical address space is Realm or Secure then no accesses to the granule in the new physical address space observe unscrubbed values.
4. GTF outcomes are guaranteed by EL3 without relying on cooperative behavior of SW that has access to the previous physical address space (for example, software running at EL2) other than performing scrubbing operations where appropriate.

K10.9.1.1 Delegate

This sequence transitions the physical granule at address `addr` from Non-secure to Secure or Realm physical address space.

On implementations with FEAT_MTE2, Root firmware must issue `DC CIGDPAPA` instead of `DC CIPAPA`, to additionally clean and invalidate Allocation Tags associated with the affected locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-27

```
Delegate(phys_addr* addr, PAS target_pas) {

    // In order to maintain mutual distrust between Realm and Secure
    // states, remove any data speculatively fetched into the target
    // physical address space.
    for (i = 0; i<granule_size; i+=cache_line_size)
        DC_CIPAPA((addr+i)), target_p);

    DSB(OSH);

    write_gpt(addr, target_pas)
    DSB(OSHST);

    TLBI_RPALOS(addr, granule_size);
    DSB(OSH)

    for (i = 0; i<granule_size; i+=cache_line_size)
        DC_CIPAPA((addr+i), PAS_Ns);

    DSB(OSH);

}
```

K10.9.1.2 Undelegate

This sequence transitions the physical granule at address `addr` from Secure or Realm physical address space to Non-secure.

The sequence assumes that the EL2 software for Secure or Realm state has already scrubbed the appropriate locations.

On implementations with FEAT_MTE2, Root firmware must issue `DC CIGDPAPA` instead of `DC CIPAPA`, to additionally clean and invalidate Allocation Tags associated with the affected locations.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-28

```
Undelegate(phys_addr* addr, PAS current_pas)

    // In order to maintain mutual distrust between Realm and Secure
    // states, remove access now, in order to guarantee that writes
    // to the currently-accessible physical address space will not
    // later become observable.
    write_gpt(addr, No_access);
    DSB(OSHST;)
    TLBI_RPALOS(addr, granule_size);
    DSB (OSH);

    // Ensure that the scrubbed data has made it past the PoPA
    for (i = 0; i<granule_size; i+=cache_line_size)
        DC_CIPAPA((addr+i), current_pas);

    DSB(OSH);

    // Ensure that GPC completes, non-coherent caches are properly flushed
    TLBI_RPALOS(addr, granule_size);
    DSB(OSH);


    // Remove any data loaded speculatively in NS space from before the scrubbing
    for (i = 0; i<granule_size; i+=cache_line_size)
        DC_CIPAPA((addr+i), PAS_NS);

    DSB(OSH);

    write_gpt (addr, PAS_NS);
    DSB(OSHST);

    // Ensure that all agents observe the new NS configuration
    TLBI_RPALOS(addr, granule_size);
    DSB(OSH);
```

K10.9.2 Procedures for changing the size of a GPT contiguous region

The following example shows a procedure to increase contiguity from 4KB to 2MB, assuming PGS is 4KB. This example procedure does not consider mutual exclusion.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-29

```
// Parameters:
// base = base address of desired contig region
// expected_gpi = value of all GPIs in the region

    assert IS_ALIGNED(base,2MB);
    assert IS_VALID_GPI (expected_gpi)

    // Required GPTE is 16 GPI values, all the same
    uint64_t required_gpte;
    required_gpte = expected_gpi;
    required_gpte |= required_gpte << 4;
    required_gpte |= required_gpte << 8;
    required_gpte |= required_gpte << 16;
    required_gpte |= required_gpte << 32;

    for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
        actual_gpte = gpt_entry(gpte_addr);
        // All entries must be consistent before the change
        if (actual_gpte !=required_gpte)
            return false;
    }

    uint64_t new_gpte = 0x1; // Contiguous descriptor
    new_gpte |= expected_gpi<<4; // GPI field
    new_gpte |= 0b01<<8; // Contig field

    for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
        set_gpt_entry(gpte_addr, new_gpte);
    }

    // No TLB maintenance required
    return true;
```

The following example shows a procedure to decrease contig from 2MB to 4KB, assuming PGS is 4KB. This example procedure does not consider mutual exclusion.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-30

```
// Parameters
// base = = base address of desired contig region
// expected_gpi = value of all GPIs in the region

    assert IS_ALIGNED(base, 2MB);
    assert IS_VALID_GPI(expected_gpi);

    // Required GPTE value
    uint64_t required_gpte = 0x1; // Contiguous descriptor
    required_gpte |= expected_gpi<<4; // GPI field
    required_gpte |= 0b01<<8; // Contig field

    for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
        // All entries must be consistent before the change
        if (actual_gpte !=required_gpte)
            return false;
    }

    // New GPTE is 16 GPI values, all the same
    uint64_t new_gpte;
    new_gpte = expected_gpi;
    new_gpte |= new_gpte << 4;
    new_gpte |= new_gpte << 8;
    new_gpte |= new_gpte << 16;
    new_gpte |= new_gpte << 32;

    for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
        set_gpt_entry(gpte_addr, new_gpte);
    }

    // It is assumed that the entries are being cracked so that they can
    // be changed, for whatever reason. In which case it required to
    // perform TLB maintenance now.
    DSB();
    TLBI_RPALOS(base, 2MB);
    DSB();
    return true;
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.10 MEC example software flows

This appendix describes example software flows for MEC.

It contains the following sections:

- Mismatched MECID avoidance
- Granule MECID association flows
- Primary and alternate MECID use models

K10.10.1 Mismatched MECID avoidance

Software must ensure that mismatched *Memory Encryption Context ID* (MECID) memory accesses do not occur, because they corrupt memory within the Realm PA space.

Software achieves this by ensuring that:

- All active translations to a physical granule use the same associated MECID value:
  - — If multiple translations exist to a physical granule with different MECID values, then mismatched MECIDs can occur due to architected or speculative accesses.
- Granules are not accessible through an active translation when updating the associated MECID value in a system register.
- All cache lines are cleaned and invalidated to the PoE, before any change of associated MECID or MECID state.

K10.10.2 Granule MECID association flows

The properties required for the granule MECID association flows are:

1. Granules are present in the Realm PA space by following the Granule Transition Flow.
   - Granules removed from the Realm PA space by the Granule Transition Flow do not require DC CIPAE operations.
2. Granules that are not allocated to a Realm or used by Realm management software have the following properties:
   - The granule is not mapped by any translation table.
   - The granule's cache lines are not present in any cache.
   - The granule may contain confidential data if the previously associated MECID is still valid.
3. Realm management software will track the following:
   - MECID association with every granule, for example by tracking the Realm owner of each granule.
   - MECID validity and life cycle.

K10.10.2.1 Remove a granule from Realm

The following illustrates how to remove a granule from Realm.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-31

```
RemoveGranule(phys_addr* addr){

    ASSERT(IsAligned(addr, TG_size));
    // Remove stage 2 mapping for granule
    // including all TLBIs and DSBs
    unmap_granule_EL10_S2(addr)
    // Remove any R_EL2 stage 1 mappings if present
    unmap_granule_EL20_S1(addr)

    // Clean and invalidate granule to PoE
    for (i = 0; i<granule_size; i+=cache_line_size)
    DC_CIPAE((addr+i), REALM);

    DSB(OSH);
}
```

Note

Unmapping can happen in bulk before the CMOs.

K10.10.2.2 Add a granule from Realm

The following example illustrates how to add a granule from Realm.

Example K10-32

```
AddGranule(phys_addr* pa, ipa_addr+ ipa, old_MECID_state) {

    if( old_MECID_state == live ) {
        temp_va = map_granule_EL2_S1(pa, AMEC = 1 )

        // Scrub pa granule of correlate-able contents
        for (i = 0; i<granule_size; i+=cache_line_size)
            DC_ZVA(temp_va+i); // or writing random data instead of ZVA is→
                acceptable

        unmap_granule_EL2_S1(pa)

        // Assume a constant MECID value when operating on a Realm VM
        // where MECID_A_EL2 == VMECID_P_EL2
        // no need for additional CMOs after the scrub operation
    }
    // Add stage 2 mapping for pa granule
    map_granule_EL10_S2(ipa,pa)
}
```

Note

Scrubbing can happen in bulk before granule mapping.

K10.10.2.3 Enable of SCTLR2_EL2.EMEC

SCTLR2_EL2.EMEC can be enabled from either:

- EL3 software, since EL2 is out-of-context.
- EL2 software if MECID_P0_EL2 == 0 and MECID_P1_EL2 == 0, to match the default MECID of zero.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.10.2.4 Update AMEC value in a descriptor

Changing the AMEC bit is a destructive operation to the memory contents of that granule.

The `RemoveGranule()` flow should be applied to the granule before remapping with a new AMEC value, and in the general case, software is required to follow a break before make procedure when changing the AMEC value.

K10.10.2.5 Update Primary MECID register

Primary MECID registers can be safely updated in the following situations:

- By a higher exception level.
- For Realm EL2 the MECID_P0_EL2 value of zero can be chosen, in order to align with the default MECID of zero, before enabling SCTLR2_EL2.EMEC.

K10.10.2.6 Update Alternate MECID register

Alternate MECID registers can be safely updated when there are no valid translations associated with the MECID and the TLBs with AMEC = 1 have been invalidated.

TCR2_EL2.AMEC0 and TCR2_EL2.AMEC1 are provided to allow Realm EL2 software to update MECID_A0_EL2 or MECID_A1_EL2 respectively, by preventing any subsequent TTD.AMEC == 1 translations.

K10.10.2.7 MECID reuse and reset of MECID tweak value

Resetting the MECID tweak value is a IMPLEMENTATION DEFINED function provided by EL3 firmware. The MECID tweak will be assigned a new value, unique within the current boot session.

Resetting a MECID tweak is destructive to all memory contents associated with the previous tweak value.

To guarantee data destruction and prepare all granules previously associated with a MECID for reuse, the granules should be either:

- Transitioned to the Non-secure PA space using the EL3 Granule Transition Flow.
- Processed using the `RemoveGranule()` flow.

K10.10.3 Primary and alternate MECID use models

The following are examples of Primary and alternate MECID use models.

K10.10.3.1 Realm EL1&0 translation regimes

The EL1&0 translation regime includes two translation tables. This translation regime is subject to stage 2 translation.

Primary MECID

The VMECID_P_EL2 is the primary MECID, for the stage 2 Realm EL1&0 translation regime. VMECID_P_EL2 is used implicitly by the EL1&0 Realm VM to protect the Realm VM state and provide consistent security guarantees. The Realm state includes:

- Private code and data pages.
- EL1&0 stage 1 translation tables.
- EL1&0 stage 2 translation tables.

Alternate MECID

The VMECID_A_EL2 is the alternate MECID, for the stage 2 Realm EL1&0 translation regime. VMECID_A_EL2 is designed to support the secondary use cases of:

- Shared data between two or more Realm VMs.
- Shared code between two or more Realm VMs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A shared data model between Realm VMs has significant challenges with mutual authentication, that exceed the current scope of the CCA Security Model.

MEC provides only one alternate context per EL1 Realm. This greatly limits the creation of 1:1 encrypted shared memory channels, between multiple Realms.

K10.10.3.2 Realm EL2&0 translation regimes

When HCR_EL2.E2H == 1 the Realm EL2&0 translation regime has two sets of translation tables, referenced by TTBR0_EL2 and TTBR1_EL2, each with their own primary and alternate MECIDs.

A common usage model is for supervisory software to use TTBR1_EL2 translation tables for its own code and data, and dynamically use the TTBR0_EL2 translation tables for the software it is currently supervising. The following explanations of EL2 MECIDs assumes this model.

Primary 0 MECID

The EL2&0 primary 0 MECID, when set to be equal VMECID_P_EL2, can enable the Realm management software to access the Realm VM state and stage 2 translation tables.

And optionally, Realm management software may protect additional Realm state with this MECID:

- Meta-data.
- vCPU contexts.

This MECID is configured in MECID_P0_EL2 for the TTBR0_EL2 translation table.

The primary Realm VM MECID value will be identical between VMECID_P_EL2 and MECID_P0_EL2.

Alternate 0 MECID

The alternative 0 MECID can be used by the Realm management software to implement use cases such as code or data sharing.

Realm management software having simultaneous access to primary 0 MECID and alternate 0 MECID of a Realm VM, can allow for efficient implementation of data sharing schemes that require data copying between Realm contexts.

This MECID is configured in MECID_A0_EL2 for the TTBR0_EL2 translation tables. For Realm management software to access memory shared with a Realm VM, the alternate Realm VM MECID value, VMECID_A_EL2, would be identical to that in MECID_A0_EL2.

Primary 1 MECID

The primary 1 MECID can be used by the Realm management software for its own code, private data and the TTBR1_EL2 and TTBR0_EL2 Realm EL2 translation tables.

This MECID is configured in MECID_P1_EL2 for the TTBR1_EL2 translation tables.

Alternate 1 MECID

The alternate 1 MECID can be used by the Realm management software for private data it wishes to isolate from the primary 1 MECID context, if the Realm management software security model requires such isolation. This MECID is configured in MECID_A1_EL2 for the TTBR1_EL2 translation tables.

K10.10.3.3 EL3 translation regime

All accesses to the Secure, Non-Secure, and Root PA space from EL3 use the default MECID zero.

Memory accesses to the Realm PAS, when translation table descriptor {NSE,NS} = {1,1}, are associated with the MECID configured in MECID_RL_A_EL3.

MECID_RL_A_EL3 is most likely to contain the MECID associated with the Realm management software.

EL3 Monitor can use MECID_RL_A_EL3 when accessing the Realm EL2 memory space to:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Populate Realm PA space memory during Realm EL2 boot.
- Write attestation reports for Realm management software.

If non-delegated attestation signing is in use, the Monitor could directly write into the requesting Realm's memory, provided it is supplied with the granule PA and Realm MECID

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K11
Barrier Litmus Tests

This appendix gives examples of the use of the barrier instructions provided by the Arm architecture. It contains the following sections:

- Introduction.
- Load-Acquire, Store-Release and barriers.
- Load-Acquire Exclusive, Store-Release Exclusive and barriers.
- Using a mailbox to send an interrupt.
- Cache and TLB maintenance instructions and barriers.

Note

This information is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.1 Introduction

The exact rules for the insertion of barriers into code sequences is a complicated subject, and this appendix describes many of the corner cases and behaviors that are possible in an implementation of the Arm architecture.

This appendix is to help programmers, hardware design engineers, and validation engineers understand the need for the different kinds of barriers.

K11.1.1 Overview of memory consistency

Early generations of microprocessors were relatively simple processing engines that executed each instruction in program order. In such processors, the effective behavior was that each instruction was executed in its entirety before a subsequent instruction started to be executed. This behavior is sometimes referred to as the *Sequential Execution Model* (SEM), and in this Manual it is described as Simple sequential execution of the program.

In later processor generations, the needs to increase processor performance, both in terms of the frequency of operation and the number of instructions executed each cycle, mean that such a simple form of execution is abandoned. Many techniques, such as pipelining, write buffering, caching, speculation, and out-of-order execution, are introduced to provide improved performance.

For general-purpose PEs, such as Arm, these microarchitectural innovations are largely hidden from the programmer by a number of microarchitectural techniques. These techniques ensure that, within an individual PE, the behavior of the PE largely remains the same as the SEM. There are some exceptions to this where explicit synchronization is required. In the Arm architecture, these are limited to cases such as:

- Synchronization of changes to the instruction stream.
- Synchronization of changes to System registers.

In both these cases, the `ISB` instruction provides the necessary synchronization.

While the effect of ordering is largely hidden from the programmer within a single PE, the microarchitectural innovations have a profound impact on the ordering of memory accesses. Write buffering, speculation, and cache coherency protocols, in particular, can all mean that the order in which memory accesses occur, as seen by an external observer, differs significantly from the order of accesses that would appear in the SEM. This is usually invisible in a uniprocessor environment, but the effect becomes much more significant when multiple PEs are trying to communicate with memory. In reality, these effects are often only significant at particular synchronization boundaries between the different threads of execution.

The problems that arise from memory ordering considerations are sometimes described as the problem of *memory consistency*. Processor architectures have adopted one or more *memory consistency models*, or *memory models*, that describe the permitted limits of the memory reordering that can be performed by an implementation of the architecture. The comparison and categorization of these has generated significant research and comment in academic circles, and Arm recommends the *Memory Consistency Models for Shared Memory-Multiprocessors* paper as an excellent detailed treatment of this subject.

This appendix does not reproduce such a work, but instead concentrates on some cases that demonstrate the features of the weakly-ordered memory model of the Arm architecture from Armv6. In particular, the examples show how the use of the `DMB` and `DSB` memory barrier instructions can provide the necessary safeguards to limit memory ordering effects at the required synchronization points.

K11.1.2 Barrier operation definitions

The following reference, or provide, definitions of terms used in this appendix:

DMB See Data Memory Barrier.

DSB See Data Synchronization Barrier.

ISB See Instruction Synchronization Barrier.

Observer, Completion See Ordering requirements defined by the formal concurrency model.

See Completion and endpoint ordering.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Program order

The order of instructions as they appear in an assembly language program. This appendix does not attempt to describe or define the legal transformations from a program written in a higher-level programming language, such as C or C++, into the machine language that can then be disassembled to give an equivalent assembly language program. Such transformations are a function of the semantics of the higher-level language and the capabilities and options on the compiler.

K11.1.3 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences.

`AArch32`

The construct `WAIT([Rx]==1)` describes the following sequence:

```
loop
    LDR R12, [Rx]
    CMP R12, #1
    BNE loop
```

Also, the construct `WAIT_ACQ([Rx]==1)` describes the following sequence:

```
loop
    LDA R12, [Rx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP R12, #1
    BNE loop
```

R12 is chosen as an arbitrary temporary register that is not in use. It is named to permit the generation of a false dependency to ensure ordering.

`AArch64`

The construct `WAIT([Xx]==1)` describes the following sequence:

```
loop
    LDR W12, [Xx]
    CMP W12, #1
    B.NE loop
```

Also, the construct `WAIT_ACQ([Xx]==1)` and describes the following sequence:

```
loop
    LDAR W12, [Xx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP W12, #1
    B.NE loop
```

For each example, a code sequence is preceded by an identifier of the observer running it:

- P0, P1. . .P*x* refer to caching coherent PEs that implement the Armv9 or Armv8 architecture and are in the same shareability domain.
- E0, E1. . .E*x* refer to non-caching observers that do not participate in the coherency protocol, but execute Armv9 or Armv8 instructions and have a weakly ordered memory model. This does not preclude these observers being different objects, such as DMA engines or other system Requesters.

These observers are unsynchronized other than as required by the documented code sequence.

Note

Throughout this appendix, *Armv9 or Armv8 instruction* and *instruction* refer to instructions from the A64, A32, or T32 instruction set, provided by Armv9 or Armv8 implementations.

Results are expressed in terms of `<agent>:<register>`, such as `P0:R5`. The results can be described as:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Permissible This does not imply that the results expressed are required or are the only possible results. In most cases they are results that would not be possible under a sequentially consistent running of the code sequences on the agents involved. In general terms, this means that these results might be unexpected to anyone unfamiliar with memory consistency issues.

Not permissible Results that the architecture expressly forbids.

Required Results that the architecture expressly requires.

The examples omit the required shareability domain arguments of `DMB` and `DSB` instructions. The arguments are assumed to be selected appropriately for the shareability domains of the observers.

In AArch32 state, where the barrier function in the litmus test can be achieved by a `DMB ST`, that is a barrier to stores only, this is shown by the use of `DMB [ST]`. This indicates that the `ST` qualifier can be omitted without affecting the result of the test. In some implementations `DMB ST` is faster than `DMB`.

For AArch64 code, the shareability domain of the `DMB` or `DSB` must be included. This is shown in this manual using the notation `DMB <domain>` and `DSB <domain>` respectively.

Except where otherwise stated, other conventions are:

- All memory initializes to 0.
- R0 and W0 contain the value 1.
- R1 - R4 and W1 - W4 contain arbitrary independent addresses that initialize to the same value on all PEs. The addresses held in these registers are shareable and:
  - The addresses held in R1 and R2 are in Write-Back Cacheable Normal memory.
  - The address held in R3 is in Write-Through Cacheable Normal memory.
  - The address held in R4 is in Non-cacheable Normal memory.
- R5 - R8 and W5 - W8 contain:
  - When used with an `STR` instruction, `0x55`, `0x66`, `0x77`, and `0x88` respectively.
  - When used with an `LDR` instruction, the value 0.
- R11 and W11 contain a new instruction or new translation table entry, as appropriate, and R10 contains the virtual address and the ASID, for use in this change of translation table entry.
- Memory locations are Normal memory locations unless otherwise stated.

The examples use mnemonics for the cache maintenance and TLB maintenance instructions. The following tables describe the mnemonics:

- Cache maintenance System instructions.
- TLB maintenance System instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.2 Load-Acquire, Store-Release and barriers

The Load-Acquire and Store-Release instructions are described in Load-Acquire, Load-AcquirePC, and Store-Release.

K11.2.1 Message passing

The following sections describe:

- Resolving weakly-ordered message passing by using Acquire and Release.
- Resolving message passing by the use of Store-Release and address dependency.

K11.2.1.1 Resolving weakly-ordered message passing by using Acquire and Release

Weakly-ordered message passing can be solved by the use of Load-Acquire and Store-Release instructions when accessing the communications flag:

`AArch32`

P1

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

`AArch64`

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

This ensures the observed order of both the reads and the writes allows transfer of data such that the result P2:R5==`0x55` is guaranteed.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

`AArch32`

P3

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

`AArch64`

P3

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

K11.2.1.2 Resolving message passing by the use of Store-Release and address dependency

Where the ordering of Normal memory accesses is not resolved by the use of barriers or dependencies, then different observers might observe the accesses in a different order. This can be resolved by the use of Store-Release for the store of the valid flag by P1, even when the observers are using an address dependency:

`AArch32`

P1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

`AArch64`

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([X2]==1)
AND W12, W12, WZR     ; W12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```

This ensures the observed order of the writes allows transfer of data such that P2:R5 and P3:R5 contain the same value of `0x55`.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

`AArch32`

P3

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

`AArch64`

P3

```
WAIT([X2]==1)
AND W12, W12, WZR     ; R12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```

K11.2.2 Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object. A Store-Release can be used to ensure the order of the update of the base address:

`AArch32`

P1

```
STR R5, [R1, #offset] ; sets new data in a field
STL R1, [R2]          ; updates base address
```

P2

```
LDR R1, [R2]          ; reads base address
CMP R1, #0            ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset] ; uses base address to read field
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`AArch64`

P1

```
STR W5, [X1, #offset] ; sets new data in a field
STLR X1, [X2]         ; updates base address
```

P2

```
LDR X1, [X2]          ; reads base address
CMP X1, #0            ; check if it is valid
B.EQ null_trap
LDR W5, [X1, #offset] ; uses base address to read field
```

It is required that P2:R5==`0x55` if the null_trap is not taken. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not need a barrier to ensure this behavior.

The read of the base address in P2 could be a Load-Acquire, but it is not necessary in this case.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers

The Armv8 architecture adds the acquire and release semantics to Load-Exclusive and Store-Exclusive instructions, which allows them to gain ordering acquire and/or release semantics.

The Load-Exclusive instruction can be specified to have acquire semantics, and the Store-Exclusive instruction can be specified to have release semantics. These can be arbitrarily combined to allow the atomic update created by a successful Load-Exclusive and Store-Exclusive pair to have any of:

- No Ordering semantics (using `LDREX` and `STREX`).
- Acquire only semantics (using `LDAEX` and `STREX`).
- Release only semantics (using `LDREX` and `STLEX`).
- Sequentially consistent semantics (using `LDAEX` and `STLEX`).

In addition, the Armv8 and later specifications require that the clearing of a global monitor will generate an event for the PE associated with the global monitor, which can simplify the use of `WFE`, by removing the need for a `DSB` barrier and `SEV` instruction.

K11.3.1 Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

Note

The inclusion of AArch32 `PLDW` instructions or AArch64 `PRFM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

For a critical region, the requirement on taking a lock is usually for acquire semantics, while the clearing of a lock requires release semantics:

`AArch32`

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store suceeded
    BNE Loop              ; retry if not

    ; loads and stores in the critical region can now be performed
```

`AArch64`

Px

```
    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

The acquire associated with the load is sufficient to ensure the required ordering in a lock situation. The Store-Exclusive will fail (and so be retried) if there is a store to the location being monitored between the Load-Exclusive and the Store-Exclusive.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.3.2 Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, the lock release needs release semantics:

`AArch32`

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock with release semantics
```

`AArch64`

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock with release semantics
```

K11.3.3 Ticket locks

When a lock is free, in order to avoid a rush to get the lock by many PEs, the use of ticket locks is common in more advanced systems. When the use is requested, the ticket locks determine the order of the users of the critical sections, in order to avoid starvation that can occur with a simple contention-based spin-lock.

A ticket lock allocates each thread a ticket number when it first requests the lock, and then compares that number with the current number for the lock. If they are the same, then the critical section can be entered. Otherwise the thread waits until the current number is equal to the ticket number for that thread.

The reading of the current number of the lock needs acquire semantics for the lock to be acquired.

Note

- The code in this section is little-endian code, as it views the combined current and next values as a single combined quantity. The addresses of the current and next ticket values must be adjusted for a big-endian system.
- The inclusion of AArch32 `PLDW` instructions or AArch64 `PRFM PST*` instructions in these examples is not a functional requirement, but improves performance on many implementations. The performance benefit of adding these instructions varies between different implementations of the architecture.

This is shown in the implementation below:

`AArch32`

Px

```
    ; R1 holds two 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R3, R5, #0x10000  ; increment the next number
    STREX R6, R3, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop1             ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    MOV R6, R5
    BEQ block_start
Loop2
    LDAH R6, [R1]         ; read current value
    CMP R6, R5, LSR #16   ; compare it with our allocated ticket
    BNE Loop2             ; retry (spin) if it is not the same
block_start
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64

Px

```
; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W3, W5, #0x10000  ; increment the next number
    STXR W6, W3, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, #0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
Loop2
    LDARH W6, [X1]        ; read current value
    CMP W6, W5, LSR #16   ; compare it with the our allocated ticket
    B.NE Loop2            ; retry (spin) if it isn't the same
block_start
```

Releasing the ticket lock simply involves incrementing the current ticket number, which is assumed in this example to be in R6, and doing a Store-Release:

AArch32

```
    ADD R6, R6, #1
    STLH R6, [R1]
```

AArch64

```
    ADD W6, W6, #1
    STLRH W6, [X1]
```

K11.3.4 Use of Wait For Event (WFE) and Send Event (SEV) with locks

The Armv8 and later architectures can use the Wait For Event mechanism to minimize the energy cost of polling variables by putting the PE into a low-power state, suspending execution, until an asynchronous exception or an explicit event is seen by that PE. In Armv8 and later architectures, the event can be generated as a result of clearing the global monitor, so removing the need for a `DSB` barrier or an explicit send event message. This can be used with simple locks or with ticket locks.

Note

The inclusion of AArch32 `PLDW` instructions or AArch64 `PRFM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

K11.3.4.1 Simple lock

The following is an example of lock acquire code using `WFE`:

AArch32

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    WFENE                 ; sleep if the lock is held
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store succeeded
    BNE Loop              ; retry if not
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64

Px

```
    SEVL                  ; invalidates the WFE on the first loop iteration
    PRFM PSTL1KEEP, [X1]  ; allocate into cache in unique state
Loop
    WFE
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

And the following is an example of lock release code:

AArch32

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock
```

AArch64

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock
```

K11.3.4.2 Ticket lock

In the Ticket lock case, the Load-Exclusive instruction can be used to move the monitor into the exclusive state for the express purpose of creating an event when the monitor changes state:

AArch32

Px

```
    ; R1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R3, R5, #0x10000  ; increment the next number
    STREX R6, R3, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop              ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    MOV R6, R5
    BEQ block_start
    SEVL
Loop2
    WFE                   ; wait if there has not been a change to the count since last
                          ; read
    LDAEXH R6, [R1]       ; check the current count
    CMP R6, R5, LSR #16   ; check if it is equal
    BNE Loop2
block_start
```

AArch64

Px

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W3, W5, #0x10000  ; increment the next number
    STXR W6, W3, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, 0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
    SEVL
Loop2
    WFE
    LDAXRH W6, [X1]       ; read current value
    CMP W6, W5, LSR #16   ; compare it with our allocated ticket
    B.NE Loop2            ; retry (spin) if it is not the same
block_start
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.4 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a `DSB` instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

`AArch32`

P1

```
STR R5, [R1]          ; message stored to shared memory location
DSB ST
STR R0, [R4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR R5, [R1]
```

`AArch64`

P1

```
STR W5, [X1]          ; message stored to shared memory location
DSB ST
STR W0, [X4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR W5, [X1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- Data cache maintenance instructions.
- Instruction cache maintenance instructions.
- TLB maintenance instructions and barriers.

K11.5.1 Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- Message passing to non-caching observers.
- Multiprocessing message passing to non-caching observers.
- Invalidating DMA buffers, non-functional example.
- Invalidating DMA buffers, functional example with single PE.
- Invalidating DMA buffers, functional example with multiple coherent PEs.

K11.5.1.1 Message passing to non-caching observers

The Armv8 and later architectures require the use of `DMB` instructions to ensure the ordering of data cache maintenance instructions and their effects. The Load-Acquire and Store-Release instructions have no effect on cache maintenance instruction. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

`AArch32`

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
DCCMVAC R1            ; cleans cache to point of coherency
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag is set
STR R0, [R4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for the flag (with order)
LDR R5, [R1]          ; reads the data
```

`AArch64`

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
DC CVAC, X1           ; cleans cache to point of coherency
DMB ISH               ; ensures effects of the clean will be observed before the
                      ; flag is set
STR W0, [X4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for the flag (with order)
LDR W5, [X1]          ; reads the data
```

In this example, it is required that E1:R5==`0x55`.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5.1.2 Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions combined with properties of barriers, means that the message passing principle for non-caching observers is:

`AArch32`

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
STL R0, [R2]          ; sends a flag for P2 (ordered by the store release)
```

P2

```
WAIT ([R2] == 1)      ; waits for P1 flag
DMB                   ; ensures cache clean is observed after P1 flag is observed
DCCMVAC R1            ; cleans cache to point of coherency - will clean P1 cache
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR R0, [R4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for P2 flag (ordered)
LDR R5, [R1]          ; reads data
```

`AArch64`

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
STLR W0, [X2]         ; sends a flag for P2 (ordered)
```

P2

```
WAIT ([X2] == 1)      ; waits sfor P1 flag
DMB SY                ; ensure cache clean is observed after P1 flag is observed
DC CVAC, X1           ; cleans cache to point of coherency, will clean P1 cache
DMB SY                ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR W0, [X4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for P2 flag
LDR W5, [X1]          ; reads data
```

In this example, it is required that E1:R5==`0x55`. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

Note

The cache maintenance instructions are not ordered by the Load-Acquire and Store-Release instructions.

K11.5.1.3 Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions with a Normal Cacheable attribute can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

`AArch32`

P1

```
DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB                   ; ensures cache invalidation is observed before the next store
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        ; is observed
STR R0, [R3]            ; sends flag to external agent
WAIT_ACQ ([R4]==1)      ; waits for a different flag from an external agent
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
STL R0, [R4]            ; sends a flag
```

AArch64

P1

```
DC IVAC, X1             ; ensure cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB SY                  ; ensures cache invalidation is observed before the next store
                        ; is observed
STR W0, [X3]            ; sends flag to external agent
WAIT_ACQ ([X4]==1)      ; waits for a different flag from an external agent
LDR W5, [X1]
```

E1

```
WAIT ([X3] == 1)        ; waits for flag
STR W5, [X1]            ; stores new data
STLR W0, [X4]           ; sends a flag
```

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

K11.5.1.4 Invalidating DMA buffers, functional example with single PE

AArch32

P1

```
DCIMVAC R1              ; ensures cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB                     ; ensures cache invalidation is observed before the next store
                        ; is observed
STR R0, [R3]            ; sends flag to external agent
WAIT ([R4]==1)          ; waits for a different flag from an external agent
DMB                     ; ensures that cache invalidate is observed after the flag
                        ; from external agent is observed
DCIMVAC R1              ; ensures cache discards stale copies before use
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
STL R0, [R4]            ; sends a flag
```

AArch64

P1

```
DC IVAC, X1             ; ensures cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB SY                  ; ensures cache invalidation is observed before the next store
                        ; is observed
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR W0, [X3]          ; sends flag to external agent
WAIT ([X4]==1)        ; waits for a different flag from an external agent
DMB SY                ; ensures that cache invalidate is observed after the flag
                      ; from external agent is observed
DC IVAC, X1           ; ensures cache discards stale copies before use
LDR W5, [X1]
```

E1

```
WAIT ([X3] == 1)      ; waits for flag
STR W5, [X1]          ; stores new data
STLR W0, [X4]         ; sends a flag
```

In this example, the result P1:R5 == `0x55` is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

K11.5.1.5 Invalidating DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of `DMB` instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

`AArch32`

P0

```
(Use data from [R1], potentially using [R1] as scratch space)
STL R0, [R2]          ; signals release of [R1]
WAIT_ACQ ([R2] == 0)  ; waits for new value from DMA
LDR R5, [R1]
```

P1

```
WAIT ([R2] == 1)      ; waits for release of [R1] by P0
DCIMVAC R1            ; ensures caches are not dirty, an invalidate is sufficient
DMB
STR R0, [R3]          ; requests new data for [R1]
WAIT ([R4] == 1)      ; waits for new data
DMB
DCIMVAC R1            ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2]          ; signals availability of new [R1]
```

E1

```
WAIT ([R3] == 1)      ; waits for new data request
STR R5, [R1]          ; sends new [R1]
DMB [ST]
STR R0, [R4]          ; indicates that new data is available to P1
```

`AArch64`

P0

```
(Use data from [X1], potentially using [X1] as scratch space)
STLR W0, [X2]         ; signals release of [X1]
WAIT_ACQ ([X2] == 0)  ; waits for new value from DMA
LDR W5, [X1]
```

P1

```
WAIT ([X2] == 1)      ; waits for release of [R1] by P0
DC IVAC, X1           ; ensures caches are not dirty, an invalidate is sufficient
DMB SY
STR W0, [X3]          ; requests new data for [R1]
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
WAIT ([X4] == 1)      ; waits for new data
DMB SY
DCIMVAC X1            ; ensures caches discard stale copies before use
DMB SY
STR WZR, [X2]         ; signals availability of new [R1]
```

E1

```
WAIT ([X3] == 1)      ; waits for new data request
STR W5, [X1]          ; sends new [R1]
STR W0, [X4]          ; indicates new data is available to P1
```

In this example, the result P0:R5 == `0x55` is required. The `DMB` issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The `DMB` issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].

K11.5.2 Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- Ensuring the visibility of updates to instructions for a uniprocessor.
- Ensuring the visibility of updates to instructions for a multiprocessor.

K11.5.2.1 Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

`AArch32`

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB
ICIMVAU R1            ; ensures instruction cache/branch predictor discards stale data
BPIMVA R1
DSB                   ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BX R1
```

In AArch64 state, the branch predictor maintenance is not required.

`AArch64`

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BR X1
```

Note

Where the changes to the instructions span multiple cache lines, then the data cache and instruction cache maintenance instructions can be duplicated to cover each of the lines to be cleaned and to be invalidated.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5.2.2 Ensuring the visibility of updates to instructions for a multiprocessor

The Armv8 and later architectures require a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

`AArch32`

P1

```
STR R11, [R1]          ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1             ; clean to PoU makes the new instruction visible to the instruction cache
DSB                    ; ensures completion of the clean on all PEs
ICIMVAU R1             ; ensures instruction cache discards stale data
BPIMVA R               ; ensures branch predictor discards stale data
DSB                    ; ensures completion of the instruction cache and branch predictor
                       ; invalidation on all PEs
STR R0, [R2]           ; sets flag to signal completion
ISB                    ; synchronizes context on this PE
BX R1                  ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)       ; waits for flag signalling completion
ISB                    ; synchronizes context on this PE
BX R1                  ; branches to new code
```

`AArch64`

P1

```
STR X11, [X1]          ; X11 contains a new instruction to be stored in program memory
DC CVAU, X1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB ISH                ; ensures completion of the clean on all PEs
IC IVAU, X1            ; ensures instruction cache/branch predictor discards stale data
DSB ISH                ; ensures completion of the instruction cache/branch predictor
                       ; invalidation on all PEs
STR W0, [X2]           ; sets flag to signal completion
ISB                    ; synchronizes context on this PE
BR R1                  ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)       ; waits for flag signalling completion
ISB                    ; synchronizes context on this PE
BR X1                  ; branches to new code
```

K11.5.2.2.1 Nonfunctional approach

The following sequence does not have the same effect, because a `DSB` is not required to complete the instruction cache maintenance instructions that other PEs issue:

`AArch32`

P1

```
STR R11, [R1]          ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1             ; clean to PoU makes the new instruction visible to the instruction cache
DSB                    ; ensures completion of the clean on all PEs
ICIMVAU R1             ; ensures instruction cache discards stale data
BPIMVA R1              : ensures branch predictor discards stale data
DMB                    ; ensures ordering of the store after the invalidation
                       ; DOES NOT guarantee completion of instruction cache/branch
                       ; predictor on other PEs
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR R0, [R2]          ; sets flag to signal completion
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
DSB                   ; this DSB does not guarantee completion of P1
                      ; ICIMVAU/BPIMVA
ISB
BX R1
```

`AArch64`

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DMB ISH               ; ensures ordering of the store after the invalidation
                      ; DOES NOT guarantee completion of instruction cache/branch
                      ; predictor on other PEs
STR W0, [X2]          ; sets flag to signal completion
DSB ISH               ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
DSB ISH               ; this DSB does not guarantee completion of P1
                      ; ICIMVAU/BPIMVA
ISB
BR X1
```

In this example, P2. . .P*x* might not see the updated region of code at R1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5.3 TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- Ensuring the visibility of updates to translation tables for a uniprocessor.
- Ensuring the visibility of updates to translation tables for a multiprocessor.
- Paging memory in and out.
- Using break-before-make when updating translation table entries.

K11.5.3.1 Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

The Armv8 and later architectures require that translation table walks look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a `DSB` and an `ISB`. Therefore, the code for updating a translation table entry is:

`AArch32`

P1

```
STR R11, [R1]          ; updates the translation table entry
DSB                    ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                    ; ensures completion of the BP and TLB invalidation
ISB                    ; synchronises context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

`AArch64`

P1

```
STR X11, [X1]          ; updates the translation table entry
DSB ISH                ; ensures visibility of the update to translation table walks
TLBI VAE1, X10         ; assumes we are in the EL1
DSB ISH                ; ensures completion of the TLB invalidation
ISB                    ; synchronise context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.

K11.5.3.2 Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. The Armv8 and later architectures require a PE that executes a TLB maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a `DSB` that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

`AArch32`

P1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

AArch64

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1IS,  X10
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a `DSB` by the observer that performed the TLB maintenance instruction. The execution of a `DSB` by a different observer does not have this effect, even if the `DSB` is known to be executed after the TLB maintenance instruction is observed by that different observer.

K11.5.3.3 Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the `DSB` that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

AArch32

P1

```
STR R11, [R1]         ; updates the translation table for the region being paged out
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10         ; invalidates the old entry
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB                   ; ensures completion of the memory transfer (this could be part of
                      ; LoadMemoryFromBackingStore)
ICIALLUIS             ; also invalidates the branch predictor
DSB                   ; ensures completion of the instruction cache
                      ; and branch predictor invalidation
STR R9, [R1]          ; creates a new translation table entry with a new mapping
DSB                   ; ensures visibility of the new translation table mapping
ISB                   ; ensures synchronisation of this instruction stream
```

AArch64

P1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR X11, [X1]          ; updates the translation table for the region being paged out
DSB ISH                ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X10       ; invalidates the old entry
DSB ISH                ; ensures completion of the invalidation on all PEs
ISB                    ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ISH                ; ensures completion of the memory transfer (this could be part of
                       ; LoadMemoryFromBackingStore)
IC IALLUIS             ; also invalidates the branch predictor
DSB ISH                ; ensures completion of the instruction cache
                       ; and branch predictor invalidation
STR X9, [X1]           ; creates a new translation table entry with a new mapping
DSB ISH                ; ensures visibility of the new translation table mapping
ISB                    ; ensures synchronisation of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of `SaveMemoryPageToBackingStore` and `LoadMemoryFromBackingStore`. `LoadMemoryFromBackingStore` is required to ensure that the memory updates that it makes are visible to instruction fetches.

In this example, the use of ICIALLUIS in AArch32 state and IC IALLUIS in AArch64 state to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU in AArch32 state and IC IVAU operations in AArch64 state. This invalidation must be done when the mapping used for the ICIMVAU and IC IVAU operations is valid but not executable.

K11.5.3.4 Using break-before-make when updating translation table entries

The Arm Architecture requires that reads to the same location are observed in order, and since application level software relies on this behavior, the operating system needs to maintain this illusion when it is changing a virtual to physical address mapping for a location, as is the case with copy on write or other memory management techniques. This illusion can be maintained provided that the software uses a break-before-make sequence when updating translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- Changing the memory type.
- Changing the cacheability attributes
- Changing the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable.

The architecture requires use of a break-before-make sequence in these situations, see Using break-before-make when updating translation table entries for more information. However, if software did not use a break-before-make approach, an implementation might give a result that would occur if the two reads to the same virtual address did not occur in program order. An example of such an occurrence would be an implementation of copy-on-write, where one PE is performing two reads to the same virtual address at the same time as a second PE, running code associated with the operating system, is copying the data from one physical location that is mapped to by that virtual address, where the page was mapped as read-only, to a different physical location which will be mapped as read/write.

If the operating system changed the address mapping without going through an invalid entry, then it would be possible for a third PE to perform a write to the location that would be seen by the first load by the first PE, and not seen by the second load by the same PE.

The required break-before-make code sequence in this case is:

`AArch32`

P1

```
; R1, R2 contain an invalid translation table entry (that is, one with bit[0] == 0)
; R3 contains the address of the translation table entry
; R4 contains the Virtual Address and ASID of the VA being remapped
; R5, R6 contain the new valid translation table entry
STRD R1, R2, [R3]          ; stores invalid entry
DSB ISH                    ; ensures visibility of the update to translation table walks
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIMVAIS R4              ; invalidates the old entry
DSB ISH                   ; ensures completion of the invalidation on all PEs
ICIALLUIS                 ; also invalidates the branch predictor
STRD R5, R6, [R3]         ; store new mapping
DSB ISH                   ; ensures visibility of the update to translation table walks
ISB                       ; ensures synchronisation of this instruction stream
```

Note

This example shows an update to an entry in a translation table that is using the long-descriptor format.

AArch64

P1

```
; X1 contains an invalid translation table entry (that is, one with bit[0] == 0)
; X2 contains the address of the translation table entry
; X3 contains the Virtual Address and ASID of the VA being remapped
; X4 contains the new valid translation table entry
STR X1, [X2]              ; stores invalid entry
DSB ISH                   ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X3           ; invalidates the old entry
DSB ISH                   ; ensures completion of the invalidation on all PEs
IC IALLUIS                ; also invalidates the branch predictor
STR X4, [X2]              ; store new mapping
DSB ISH                   ; ensures visibility of the update to translation table walks
ISB                       ; ensures synchronisation of this instruction stream
```

If this sequence is correctly followed, then the architecture guarantees that the loads to a virtual address being remapped will be seen in the correct order.

The instruction cache maintenance is only required if the mapping from input address to output address has been changed as part of the change of the translation table entries, and the memory being moved is executable.

In this example, the use of `ICIALLUIS` in AArch32 state and `IC IALLUIS` in AArch64 state to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using `ICIMVAU` in AArch32 state, and `IC IVAU` in AArch64 state. This invalidation must be done when the mapping used for the `ICIMVAU` and `IC IVAU` operations is valid but not executable.

K11.5.4 Ordering of Memory-mapped device control with payloads

With a Memory-mapped peripheral, such as a DMA, which can also access memory for its own use, it is common to have control or status registers which are Memory-mapped. These registers need to be accessed in an ordered manner with respect to the data that the Memory-mapped peripheral is handling.

Two simple examples of this are:

- When a processing element is writing a buffer of data, and then writing to a control register in the DMA peripheral to start that peripheral to access the buffer of data.
- When a DMA peripheral has written to a buffer of data in memory, and the processing element is reading a status register to determine that the DMA transfer has completed, and then is reading the data.

For the case of the processing element writing a buffer of data, before starting the DMA peripheral, the ordering requirements between the stores to the data buffer and the stores to the Memory-mapped a to the DMA peripheral can be met by the insertion of a `DSB <domain>` instruction between these sets of accesses as this ensures the global observation of the stores before the DMA is started. this is shown by the following code:

AArch32

P1

```
STR R5, [R2]          ; data written to the data buffer
DSB
STR R0, [R4]          ; R4 contains the address of the DMA control register
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`AArch64`

P1

```
STR W5, [X2]          ; data written to the data buffer
DSB <domain>
STR W0, [X4]          ; X4 contains the address of the DMA control register
```

For the case of DMA peripheral writing the data buffer and then setting a status register when those stores are complete (and so globally observed) and then having this status register polled by the processing element before the processing element reads the data buffer, the processing element must insert a `DSB <domain>` between the load that reads the status register, and the read of the buffer. A DMB, or load-acquire, is not sufficient as this problem is not solely concerned with observation order, since the polling read is actually a read of a status register at a Completer, not the polling a data value that has been written by an observer.

For this case, the code is therefore:

`AArch32`

P1

```
WAIT ([R4] == 1)      ; R4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB
LDR R5, [R2]          ; reads data from the data buffer
```

`AArch64`

P1

```
WAIT ([X4] == 1)      ; X4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB <domain>
LDR W5, [X2]          ; reads data from the data buffer
```


Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K12
Random Number Generation

This appendix provides further information on the generation of random numbers using FEAT_RNG. It contains the following section:

- Properties of the generated random number.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K12.1 Properties of the generated random number

When FEAT_RNG is implemented, reads of the RNDR and RNDRRS registers return 64-bit random numbers. The RNDR and RNDRRS implementation should conform to approved standards that are appropriate for the market requirements.

For example, the NIST SP 800-90 series of documents:

- SP 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators.
- SP 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation.
- SP 800-90C Recommendation for Random Bit Generator Constructions.

Note

- Since an entropy source can only generate random bits at a limited rate, the random number bits are commonly collected in an “entropy pool” until needed. An implementation should ensure that lower privileged software cannot impact the performance of higher privileged software by entirely draining this “entropy pool”. The refill time cost of the “entropy pool” should be paid for by the persistent caller.
- When FEAT_RNG_TRAP is implemented, reads of the RNDR and RNDRRS registers may be trapped to EL3. For more information about this trapping behavior, see control fields ID_AA64PFR1_EL1.RNDR_trap and SCR_EL3.TRNDR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K13
Arm Pseudocode Definition

The *Architecture Specification Language* (ASL) is defined in *Arm® Architecture Specification Language Reference* (ARM DDI 0626).

This appendix provides further information about the pseudocode used in this manual. It contains the following sections:

- About the Arm pseudocode.
- Pseudocode for instruction descriptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. Pseudocode for instruction descriptions gives general information about this instruction pseudocode, including its limitations.

K13.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in General limitations of Arm pseudocode and Limitations of the instruction pseudocode.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. Instruction encoding diagrams and instruction pseudocode gives more information about the pseudocode provided for each instruction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K14
Registers Index

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:

- Introduction and register disambiguation.
- Alphabetical index of AArch64 registers and System instructions.
- Functional index of AArch64 registers and System instructions.
- Alphabetical index of AArch32 registers and System instructions.
- Functional index of AArch32 registers and System instructions.
- Alphabetical index of memory-mapped registers.
- Functional index of memory-mapped registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.1 Introduction and register disambiguation

In some sections of this manual, registers are referred to by a *general name*, where the description applies to more than one context. Generally, this is one of the following:

- The description applies to both AArch32 state and AArch64 state, and therefore the register names could apply to either AArch32 System registers or AArch64 System registers.
- The description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception level suffix, _EL0, _EL1, _EL2, or _EL3.

The following sections disambiguate the general register names:

- Register name disambiguation by Execution state.
- Register name disambiguation by Exception level.

K14.1.1 Register name disambiguation by Execution state

Table K14-1 disambiguates the general names of the registers by Execution state.

Table K14-1 Disambiguation of general names of registers by Execution state

General name	Short description	AArch64 register	AArch32 register
CONTEXTIDR	Context ID	CONTEXTIDR_EL1	CONTEXTIDR
DBGAUTHSTATUS	Debug Authentication Status	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS
DBGBCR	Debug Breakpoint Control Registers	DBGBCR<n>_EL1	DBGBCR<n>
DBGBVR	Debug Breakpoint Value Registers	DBGBVR<n>_EL1	DBGBVR<n> DBGBXVR<n>
DBGCLAIMCLR	Debug CLAIM Tag Clear register	DBGCLAIMCLR_EL1	DBGCLAIMCLR
DBGCLAIMSET	Debug CLAIM Tag Set register	DBGCLAIMSET_EL1	DBGCLAIMSET
DBGDTRRX	Debug Data Transfer Register, Receive	DBGDTRRX_EL0	DBGDTRRXint
DBGDTRTX	Debug Data Transfer Register, Transmit	DBGDTRTX_EL0	DBGDTRTXint
DBGPRCR	Debug Power Control Register	DBGPRCR_EL1	DBGPRCR
DBGVCR	Debug Vector Catch Register	DBGVCR32_EL2	DBGVCR
DBGWCR	Debug Watchpoint Control Registers	DBGWCR<n>_EL1	DBGWCR<n>
DBGWVR	Debug Watchpoint Value Registers	DBGWVR<n>_EL1	DBGWVR<n>
DCCINT	Debug Comms Channel Interrupt Enable Register	MDCCINT_EL1	DBGDCCINT
DCCSR	Debug Comms Channel Status Register	MDCCSR_EL0	DBGDSCRint
DLR	Debug Link Register	DLR_EL0[31:0]	DLR
DSCR	Debug System Control Register	MDSCR_EL1	DBGDSCRext
DSPSR	Debug Saved PE State Register	DSPSR_EL0	DSPSR, DSPSR2
FAR	Fault Address Register	FAR_EL1 FAR_EL2 FAR_EL3 HPFAR_EL2	DFAR, IFAR HDFAR, HIFAR FAR_EL3 HPFAR
HCR	Hypervisor Configuration Register	HCR_EL2	HCR HCR2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General name	Short description	AArch64 register	AArch32 register
HDCR	Hyp or EL2 Debug Control Register	MDCR_EL2	HDCR
HSCTLR	Hypervisor System Control Register	SCTLR_EL2	HSCTLR
HTTBR	EL2 Translation Table Base Register	TTBR0_EL2	HTTBR
ISR	Interrupt Status Register	ISR_EL1	ISR
MPIDR	Multiprocessor Affinity Register	MPIDR_EL1	MPIDR
OSDLR	OS Double-Lock Register	OSDLR_EL1	DBGOSDLR
OSDTRRX	OS Lock Data Transfer Register, Receive	OSDTRRX_EL1	DBGDTRRXext
OSDTRTX	OS Lock Data Transfer Register, Transmit	OSDTRTX_EL1	DBGDTRTXext
OSECCR	OS Lock Exception Catch Control Register	OSECCR_EL1	DBGOSECCR
OSLAR	OS Lock Access Register	OSLAR_EL1	DBGOSLAR
OSLSR	OS Lock Status Register	OSLSR_EL1	DBGOSLSR
PMMIR	Performance Monitors Machine Identification Register	PMMIR_EL1	PMMIR
SCR	Secure Configuration Register	SCR_EL3	SCR
SCTLR	System Control Register	SCTLR_EL1 SCTLR_EL2 SCTLR_EL3	SCTLR (NS) HSCTLR SCTLR (S)
SDCR	Secure or EL3 Debug Configuration Register	MDCR_EL3	SDCR
SDER	Secure Debug Enable Register	SDER32_EL3	SDER
SPSR	Saved Program Status Register	SPSR_EL1 SPSR_EL2 SPSR_EL3	SPSR (general description) SPSR_abt SPSR_fiq SPSR_hyp SPSR_irq SPSR_mon SPSR_svc SPSR_und
TCR	Translation Control Register	TCR_EL1 TCR_EL2 TCR_EL3 VTCR_EL2	TTBCR(NS) HTCR TTBCR(S) VTCR
TTBR	Translation Table Base Register	TTBR0_EL1 TTBR0_EL2 TTBR0_EL3 TTBR1_EL1 VTTBR_EL2	TTBR0 TTBR1 HTTBR VTTBR
VBAR	Vector Base Address Register	VBAR_EL1 VBAR_EL2 VBAR_EL3	VBAR HVBAR MVBAR
VTCR	PL1&0 stage 2 Translation Control Register	VTCR_EL2	VTCR
VTTBR	PL1&0 stage 2 Translation Table Base Register	VTTBR_EL2	VTTBR

Table K14-2 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-2 Disambiguation of general names of the Performance Monitors System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
PMCCFILTR	Cycle Count Filter Register	PMCCFILTR_EL0	PMCCFILTR
PMCCNTR	Cycle Count Register	PMCCNTR_EL0	PMCCNTR
PMCEID0	Performance Monitors Common Event Identification Register 0	PMCEID0_EL0	PMCEID0
PMCEID1	Performance Monitors Common Event Identification Register 1	PMCEID1_EL0	PMCEID1
PMCEID2	Performance Monitors Common Event Identification Register 2	PMCEID0_EL0	PMCEID2
PMCEID3	Performance Monitors Common Event Identification Register 3	PMCEID1_EL0	PMCEID3
PMCNTENCLR	Performance Monitors Count Enable Clear register	PMCNTENCLR_EL0	PMCNTENCLR
PMCNTENSET	Performance Monitors Count Enable Set register	PMCNTENSET_EL0	PMCNTENSET
PMCR	Performance Monitors Control Register	PMCR_EL0	PMCR
PMEVCNTR<n>	Performance Monitors Event Count Registers, n = 0-30	PMEVCNTR<n>_EL0	PMEVCNTR<n>
PMEVTYPER<n>	Performance Monitors Event Type Registers, n = 0-30	PMEVTYPER<n>_EL0	PMEVTYPER<n>
PMINTENCLR	Performance Monitors Interrupt Enable Clear register	PMINTENCLR_EL1	PMINTENCLR
PMINTENSET	Performance Monitors Interrupt Enable Set register	PMINTENSET_EL1	PMINTENSET
PMMIR	Performance Monitors Machine Identification Register	PMMIR_EL1	PMMIR
PMOVSCLR	Performance Monitors Overflow Flag Status Register	PMOVSCLR_EL0	PMOVSR
PMOVSSET	Performance Monitors Overflow Flag Status Set register	PMOVSSET_EL0	PMOVSSET
PMSELR	Performance Monitors Event Counter Selection Register	PMSELR_EL0	PMSELR
PMSWINC	Performance Monitors Software Increment register	PMSWINC_EL0	PMSWINC
PMUSERENR	Performance Monitors User Enable Register	PMUSERENR_EL0	PMUSERENR
PMXEVCNTR	Performance Monitors Selected Event Count Register	PMXEVCNTR_EL0	PMXEVCNTR
PMXEVTYPER	Performance Monitors Selected Event Type Register	PMXEVTYPER_EL0	PMXEVTYPER

Table K14-3 disambiguates the general names of the System registers that provide access to the Activity Monitors by Execution state.

Table K14-3 Disambiguation of general names of the Activity Monitors System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
AMCFGR	Activity Monitors Configuration Register	AMCFGR_EL0	AMCFGR
AMCGCR	Activity Monitors Counter Group Configuration Register	AMCGCR_EL0	AMCGCR
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0	AMCNTENCLR0_EL0	AMCNTENCLR0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1	AMCNTENCLR1_EL0	AMCNTENCLR1
AMCNTENSET0	Activity Monitors Count Enable Set Register 0	AMCNTENSET0_EL0	AMCNTENSET0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1	AMCNTENSET1_EL0	AMCNTENSET1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General name	Short description	AArch64 register	AArch32 register
AMCR	Activity Monitors Control Register	AMCR_EL0	AMCR
AMEVCNTR0<n>	Activity Monitors Event Counter Registers 0, n = 0-15	AMEVCNTR0<n>_EL0	AMEVCNTR0<n>
AMEVCNTR1<n>	Activity Monitors Event Counter Registers 1, n = 0-15	AMEVCNTR1<n>_EL0	AMEVCNTR1<n>
AMEVTYPER0<n>	Activity Monitors Event Type Registers 0, n = 0-15	AMEVTYPER0<n>_EL0	AMEVTYPER0<n>
AMEVTYPER1<n>	Activity Monitors Event Type Registers 1, n = 0-15	AMEVTYPER1<n>_EL0	AMEVTYPER1<n>
AMUSERENR	Activity Monitors User Enable Register	AMUSERENR_EL0	AMUSERENR

Table K14-4 disambiguates the general names of the System registers that provide access to the Generic Timer System by Execution state.

Table K14-4 Disambiguation of general names of the Generic Timer System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
CNTFRQ	Counter-timer Frequency register	CNTFRQ_EL0	CNTFRQ
CNTHCTL	Counter-timer Hypervisor Control register	CNTHCTL_EL2	CNTHCTL
CNTHP_CTL	Counter-timer Hypervisor Physical Timer Control register	CNTHP_CTL_EL2	CNTHP_CTL
CNTHP_CVAL	Counter-timer Hypervisor Physical Timer CompareValue register	CNTHP_CVAL_EL2	CNTHP_CVAL
CNTHP_TVAL	Counter-timer Hypervisor Physical Timer TimerValue register	CNTHP_TVAL_EL2	CNTHP_TVAL
CNTKCTL	Counter-timer Kernel Control register	CNTKCTL_EL1	CNTKCTL
CNTP_CTL	Counter-timer Physical Timer Control register	CNTP_CTL_EL0	CNTP_CTL
CNTP_CVAL	Counter-timer Physical Timer CompareValue register	CNTP_CVAL_EL0	CNTP_CVAL
CNTP_TVAL	Counter-timer Physical Timer TimerValue register	CNTP_TVAL_EL0	CNTP_TVAL
CNTPCT	Counter-timer Physical Count register	CNTPCT_EL0	CNTPCT
CNTPS_CTL	Counter-timer Physical Secure Timer Control register	CNTPS_CTL_EL1	-
CNTPS_CVAL	Counter-timer Physical Secure Timer CompareValue register	CNTPS_CVAL_EL1	-
CNTPS_TVAL	Counter-timer Physical Secure Timer TimerValue register	CNTPS_TVAL_EL1	-
CNTV_CTL	Counter-timer Virtual Timer Control register	CNTV_CTL_EL0	CNTV_CTL
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register	CNTV_CVAL_EL0	CNTV_CVAL
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register	CNTV_TVAL_EL0	CNTV_TVAL
CNTVCT	Counter-timer Virtual Count register	CNTVCT_EL0	CNTVCT
CNTVOFF	Counter-timer Virtual Offset register	CNTVOFF_EL2	CNTVOFF

Table K14-5 shows the mappings between the writable AArch64 System registers and the AArch32 System registers.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-5 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
ACTLR_EL1[31:0]	ACTLR[a]
ACTLR_EL1[63:32]	ACTLR2[a] if implemented
AFSR0_EL1[31:0]	ADFSR[a]
AFSR1_EL1[31:0]	AIFSR[a]
AMAIR_EL1[31:0]	AMAIR0[a]
AMAIR_EL1[63:32]	AMAIR1[a]
CONTEXTIDR_EL1[31:0]	CONTEXTIDR[a]
CPACR_EL1[31:0]	CPACR
CSSELR_EL1[31:0]	CSSELR[a]
DACR32_EL2[31:0]	DACR[a]
FAR_EL1[31:0]	DFAR[a]
ESR_EL1[31:0]	DFSR[a]
HACR_EL2[31:0]	HACR
ACTLR_EL1[31:0]	HACTLR
ACTLR_EL2[63:32]	HACTLR2 if implemented
AFSR0_EL1[31:0]	HADFSR
AFSR1_EL1[31:0]	HAIFSR
AMAIR_EL2[31:0]	HAMAIR0
AMAIR_EL2[63:32]	HAMAIR1
CPTR_EL2[31:0]	HCPTR
HCR_EL2[31:0]	HCR
HCR_EL2[63:32]	HCR2
MDCR_EL2[31:0]	HDCR
FAR_EL2[31:0]	HDFAR
FAR_EL2[63:32]	HIFAR
MAIR_EL2[31:0]	HMAIR0
MAIR_EL2[63:32]	HAMAIR1
HPFAR_EL2[31:0]	HPFAR
SCTLR_EL2[31:0]	HSCTLR
ESR_EL2[31:0]	HSR
HSTR_EL2[31:0]	HSTR
TCR_EL3[31:0]	HTCR
TPIDR_EL2[31:0]	HTPIDR
TTBR0_EL2[47:1]	HTTBR
VBAR_EL2[31:0]	HVBAR
FAR_EL1[63:32]	IFAR[a]
IFSR32_EL2[31:0]	IFSR[a]
MAIR_EL1[63:32]	NMRR or MAIR1 [a]

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64 register	AArch32 register
PAR_EL1[63:0]	PAR[a]
MAIR_EL1[31:0]	PRRR or MAIR0 [a]
RMR_EL1[31:0]	RMR (at EL1)
RMR_EL2[31:0]	HRMR
RMR_EL3[31:0]	RMR (at EL3)
SCTLR_EL1[31:0]	SCTLR[a]
SDER32_EL3[31:0]	SDER
TPIDR_EL1[31:0]	TPIDRPRW[a]
TPIDRRO_EL0[31:0]	TPIDRURO[a]
TPIDR_EL0[31:0]	TPIDRURW[a]
TCR_EL1[31:0]	TTBCR[a]
TCR_EL1[63:32]	TTBCR2[a] if implemented
TTBR0_EL1[63:0]	TTBR0[a]
TTBR1_EL1[63:0]	TTBR1[a]
VBAR_EL1[31:0]	VBAR[a]
VMPIDR_EL2[31:0]	VMPIDR
VPIDR_EL2[31:0]	VPIDR
VTCR_EL2[31:0]	VTCR
VTTBR_EL2[63:0]	VTTBR
Timer registers	
CNTFRQ_EL0	CNTFRQ
CNTHCTL_EL2	CNTHCTL
CNTHP_CTL_EL2	CNTP_CTL
CNTHP_CVAL_EL2	CNTHP_CVAL
CNTHP_TVAL_EL2	CNTHVS_TVAL
CNTHPS_CTL_EL2	CNTHPS_CTL
CNTHPS_CVAL_EL2	CNTHPS_CVAL
CNTHPS_TVAL_EL2	CNTHPS_TVAL
CNTKCTL_EL1	CNTKCTL
CNTP_CTL_EL0	CNTP_CTL[a]
CNTP_CVAL_EL0	CNTP_CVAL[a]
CNTP_TVAL_EL0	CNTP_TVAL[a]
CNTPCT_EL0[63:0]	CNTPCT
CNTV_CTL_EL0[31:0]	CNTV_CTL
CNTV_CVAL_EL0[63:0]	CNTV_CVAL
CNTV_TVAL_EL0[31:0]	CNTV_TVAL
CNTHV_CTL_EL2[63:0]	CNTHV_CTL
CNTHV_CVAL_EL2[63:0]	CNTHV_CVAL
CNTHV_TVAL_EL2[63:0]	CNTHV_TVAL
CNTHVS_CTL_EL2[31:0]	CNTHVS_CTL

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64 register	AArch32 register
CNTHVS_CVAL_EL2[63:0]	CNTHVS_CVAL
CNTHVS_TVAL_EL2[63:0]	CNTHVS_TVAL
CNTVCT_EL0[63:0]	CNTVCT
CNTVOFF_EL2[63:0]	CNTVOFF
Debug System registers	
DBGAUTHSTATUS_EL1[31:0]	DBGAUTHSTATUS
DBGBCR<n>_EL1[31:0]	DBGBCR<n>
DBGBVR<n>_EL1[31:0]	DBGBVR<n>
DBGBVR<n>_EL1[63:32]	DBGBXVR<n>
DBGCLAIMCLR_EL1[31:0]	DBGCLAIMCLR
DBGCLAIMSET_EL1[31:0]	DBGCLAIMSET
DBGDTR_EL0[63:32]	DBGDTRRXint
DBGDTR_EL0[31:0]	DBGDTRTXint
DBGDTRRX_EL0[31:0]	DBGDTRRXint
DBGDTRTX_EL0[31:0]	DBGDTRTXint
DBGPRCR_EL1[31:0]	DBGPRCR
DBGVCR32_EL2[31:0]	DBGVCR
DBGWCR<n>_EL1[31:0]	DBGWCR<n>
DBGWVR<n>_EL1[31:0]	DBGWVR<n>
ID_DFR0_EL1[31:0]	ID_DFR0
MDCCSR_EL0 [b][30:29]	DBGDSCRint[b]
MDCR_EL2[31:0]	HDCR
MDRAR_EL1[63:0]	DBGDRAR
MDSCR_EL1 [b][31:0]	DBGDSCRext[b]
OSDLR_EL1[31:0]	DBGOSDLR
OSDTRRX_EL1 [b][31:0]	DBGDTRRXext[b]
OSDTRTX_EL1 [b][31:0]	DBGDTRTXext[b]
OSECCR_EL1[31:0]	DBGOSECCR
OSLAR_EL1[31:0]	DBGOSLAR
OSLSR_EL1[31:0]	DBGOSLSR
SDER32_EL3[31:0]	SDER
Performance Monitors System registers	
PMCCNTR_EL0[31:0]	PMCCNTR (MRC/MCR)
PMCEID0_EL0[31:0]	PMCEID0
PMCEID0_EL0[63:32]	PMCEID2
PMCEID1_EL0[31:0]	PMCEID1
PMCEID1_EL0[63:32]	PMCEID3
PMCNTENCLR_EL0[31:0]	PMCNTENCLR
PMCNTENSET_EL0[31:0]	PMCNTENSET
PMCR_EL0[31:0]	PMCR

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64 register	AArch32 register
PMEVCNTR<n>_EL0[31:0]	PMEVCNTR<n>
PMEVTYPER<n>_EL0[31:0]	PMEVTYPER<n>
PMINTENCLR_EL1[31:0]	PMINTENCLR
PMINTENSET_EL1[31:0]	PMINTENSET
PMSELR_EL0[31:0]	PMSELR
PMSWINC_EL0[31:0]	PMSWINC
PMUSERENR_EL0[31:0]	PMUSERENR
PMXEVCNTR_EL0[31:0]	PMXEVCNTR
PMXEVTYPER_EL0[31:0]	PMXEVTYPER
Activity Monitors System registers	
AMCNTENCLR0_EL0[31:0]	AMCNTENCLR0
AMCNTENCLR1_EL0[31:0]	AMEVCNTR1<n>
AMCNTENSET0_EL0[31:0]	AMCNTENSET0
AMCNTENSET1_EL0[31:0]	AMCNTENSET1
AMCR_EL0[31:0]	AMCR
AMEVCNTR0<n>_EL0[63:0]	AMEVCNTR0<n>
AMEVCNTR1<n>_EL0[63:0]	AMEVCNTR1<n>
AMEVCNTR1<n>_EL0[31:0]	AMEVTYPER1<n>
RAS System registers	
DISR_EL1[31:0]	DISR
ERRIDR_EL1[31:0]	ERRIDR
ERRSELR_EL1[31:0]	ERRSELR
ERXADDR_EL1[31:0]	ERXADDR
ERXADDR_EL1[63:32]	ERXADDR2
ERXCTLR_EL1[31:0]	ERXCTLR
ERXCTLR_EL1[63:32]	ERXCTLR2
ERXFR_EL1[31:0]	ERXFR
ERXFR_EL1[63:32]	ERXFR2
ERXMISC0_EL1[31:0]	ERXMISC0
ERXMISC0_EL1[63:32]	ERXMISC1
ERXMISC1_EL1[31:0]	ERXMISC2
ERXMISC1_EL1[63:32]	ERXMISC3
ERXMISC2_EL1[31:0]	ERXMISC4
ERXMISC2_EL1[63:32]	ERXMISC5
ERXMISC3_EL1[31:0]	ERXMISC6
ERXMISC3_EL1[63:32]	ERXMISC7
ERXSTATUS_EL1[31:0]	ERXSTATUS
VDISR_EL2[31:0]	VDISR
VSESR_EL2[31:0]	VDFSR

a. AArch32 registers that are banked if EL3 is using AArch32.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

b. These registers have overlapping register content. One or more bits of one register appear in the other register.

There are a small number of AArch32 System registers that are not mapped to any AArch64 System registers. The AArch64 registers listed in Table K14-6 can be used to access these from a higher Exception level that is using AArch64. The registers shown in the table are UNDEFINED if EL1 cannot use AArch32.

Table K14-6 AArch64 registers for accessing registers that are only used in AArch32 state

AArch32 register	Register for access from AArch64 state	Short description
DACR	DACR32_EL2	Domain Access Control Register
DBGVCR	DBGVCR32_EL2	Debug Vector Catch Register
FPEXC	FPEXC32_EL2	Floating-Point Exception Control Register
IFSR	IFSR32_EL2	Instruction Fault Status Register
SDER	SDER32_EL3	AArch32 Secure Debug Enable Register

Table K14-7 shows the AArch64 System registers that allow access from AArch64 state to the AArch32 ID registers. These AArch64 registers are UNKNOWN if no Exception level can use AArch32.

Table K14-7 AArch64 registers that access the AArch32 ID registers

AArch32 register	Register for access from AArch64 state	Short description
ID_AFR0	ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0	ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1	ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0	ID_ISAR0_EL1	EL1, AArch32 Instruction Set Attribute Register 0
ID_ISAR1	ID_ISAR1_EL1	EL1, AArch32 Instruction Set Attribute Register 1
ID_ISAR2	ID_ISAR2_EL1	EL1, AArch32 Instruction Set Attribute Register 2
ID_ISAR3	ID_ISAR3_EL1	EL1, AArch32 Instruction Set Attribute Register 3
ID_ISAR4	ID_ISAR4_EL1	EL1, AArch32 Instruction Set Attribute Register 4
ID_ISAR5	ID_ISAR5_EL1	EL1, AArch32 Instruction Set Attribute Register 5
ID_ISAR6	ID_ISAR6_EL1	EL1, AArch32 Instruction Set Attribute Register 6
ID_MMFR0	ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1	ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2	ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3	ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4	ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5	ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0	ID_PFR0_EL1	AArch32 PE Feature Register 0
ID_PFR1	ID_PFR1_EL1	AArch32 PE Feature Register 1
ID_PFR2	ID_PFR2_EL1	AArch32 PE Feature Register 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.1.2 Register name disambiguation by Exception level

Table K14-8 disambiguates the general names of the AArch64 System registers by Exception level.

Table K14-8 Disambiguation of AArch64 System registers by Exception level

General form	EL0	EL1	EL2	EL3
AFSR0_ELx	-	AFSR0_EL1	AFSR0_EL2	AFSR0_EL3
AFSR1_ELx	-	AFSR1_EL1	AFSR1_EL2	AFSR1_EL3
AMAIR2_ELx	-	AMAIR2_EL1	AMAIR2_EL2	AMAIR2_EL3
AMAIR_ELx	-	AMAIR_EL1	AMAIR_EL2	AMAIR_EL3
CONTEXTIDR_ELx	-	CONTEXTIDR_EL1	CONTEXTIDR_EL2	-
CPTR_ELx	-	-	CPTR_EL2	CPTR_EL3
ELR_ELx	-	ELR_EL1	ELR_EL2	ELR_EL3
ESR_ELx	-	ESR_EL1	ESR_EL2	ESR_EL3
FAR_ELx	-	FAR_EL1	FAR_EL2	FAR_EL3
GCSCR_ELx	-	GCSCR_EL1	GCSCR_EL2	GCSCR_EL3
GCSPR_ELx	GCSPR_EL0	GCSPR_EL1	GCSPR_EL2	GCSPR_EL3
MAIR2_ELx	-	MAIR2_EL1	MAIR2_EL2	MAIR2_EL3
MAIR_ELx	-	MAIR_EL1	MAIR_EL2	MAIR_EL3
MPAMn_ELx	-	MPAM0_EL1, MPAM1_EL1	MPAM2_EL2	MPAM3_EL3
MPAMVPMn_ELx	-	-	MPAMVPM0_EL2, MPAMVPM1_EL2, MPAMVPM2_EL2, MPAMVPM3_EL2, MPAMVPM4_EL2, MPAMVPM5_EL2, MPAMVPM6_EL2, MPAMVPM7_EL2	-
MPAMBWn_ELx	-	MPAMBW0_EL1, MPAMBW1_EL1	MPAMBW2_EL2	MPAMBW3_EL3
PFAR_ELx	-	PFAR_EL1	PFAR_EL2	-
PIR_ELx	-	PIR_EL1	PIR_EL2	PIR_EL3
PIRE0_ELx	-	PIRE0_EL1	PIRE0_EL2	-
PMBSR_ELx	-	PMBSR_EL1	PMBSR_EL2	PMBSR_EL3
POR_ELx	POR_EL0	POR_EL1	POR_EL2	POR_EL3
RMR_ELx	-	RMR_EL1	RMR_EL2	RMR_EL3
RVBAR_ELx	-	RVBAR_EL1	RVBAR_EL2	RVBAR_EL3
SCTLR2_ELx	-	SCTLR2_EL1	SCTLR2_EL2	SCTLR2_EL3
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
SCXTNUM_ELx	SCXTNUM_EL0	SCXTNUM_EL1	SCXTNUM_EL2	SCXTNUM_EL3
SMCR_ELx	-	SMCR_EL1	SMCR_EL2	SMCR_EL3
SP_ELx	SP_EL0	SP_EL1	SP_EL2	SP_EL3
SPSR_ELx	-	SPSR_EL1	SPSR_EL2	SPSR_EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General form	EL0	EL1	EL2	EL3
TCR2_ELx	-	TCR2_EL1	TCR2_EL2	-
TCR_ELx	-	TCR_EL1	TCR_EL2	TCR_EL3
TFSR_ELx	TFSRE0_EL1	TFSR_EL1	TFSR_EL2	TFSR_EL3
TRBSR_ELx	-	TRBSR_EL1	TRBSR_EL2	TRBSR_EL3
TRFCR_ELx	-	TRFCR_EL1	TRFCR_EL2	-
TTBR_ELx	-	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2, TTBR1_EL2, VTTBR_EL2, VSTTBR_EL2	TTBR0_EL3
TTBRn_ELx	-	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2, TTBR1_EL2	TTBR0_EL3
TTBR0_ELx	-	TTBR0_EL1	TTBR0_EL2	TTBR0_EL3
TTBR1_ELx	-	TTBR1_EL1	-	-
VBAR_ELx	-	VBAR_EL1	VBAR_EL2	VBAR_EL3
ZCR_ELx	-	ZCR_EL1	ZCR_EL2	ZCR_EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.2 Alphabetical index of AArch64 registers and System instructions

This section is an index of AArch64 registers and System instructions in alphabetical order.

Table K14-9 Alphabetical index of AArch64 Registers

Register	Description, see
ACCDATA_EL1	Accelerator Data
ACTLR_EL1	Auxiliary Control Register (EL1)
ACTLR_EL2	Auxiliary Control Register (EL2)
ACTLR_EL3	Auxiliary Control Register (EL3)
ACTLRMASK_EL1	Auxiliary Control Masking Register (EL1)
ACTLRMASK_EL2	Auxiliary Control Masking Register (EL2)
AFSR0_EL1	Auxiliary Fault Status Register 0 (EL1)
AFSR0_EL2	Auxiliary Fault Status Register 0 (EL2)
AFSR0_EL3	Auxiliary Fault Status Register 0 (EL3)
AFSR1_EL1	Auxiliary Fault Status Register 1 (EL1)
AFSR1_EL2	Auxiliary Fault Status Register 1 (EL2)
AFSR1_EL3	Auxiliary Fault Status Register 1 (EL3)
AIDR_EL1	Auxiliary ID Register
ALLINT	All Interrupt Mask Bit
AMAIR2_EL1	Extended Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR2_EL2	Extended Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR2_EL3	Extended Auxiliary Memory Attribute Indirection Register (EL3)
AMAIR_EL1	Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR_EL2	Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR_EL3	Auxiliary Memory Attribute Indirection Register (EL3)
AMCFGR_EL0	Activity Monitors Configuration Register
AMCG1IDR_EL0	Activity Monitors Counter Group 1 Identification Register
AMCGCR_EL0	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0_EL0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1_EL0	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0_EL0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1_EL0	Activity Monitors Count Enable Set Register 1
AMCR_EL0	Activity Monitors Control Register
AMUSERENR_EL0	Activity Monitors User Enable Register
APAS	Associate PA space
APDAKeyHi_EL1	Pointer Authentication Key A for Data (bits[127:64])
APDAKeyLo_EL1	Pointer Authentication Key A for Data (bits[63:0])
APDBKeyHi_EL1	Pointer Authentication Key B for Data (bits[127:64])
APDBKeyLo_EL1	Pointer Authentication Key B for Data (bits[63:0])

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
APGAKeyHi_EL1	Pointer Authentication Key A for Code (bits[127:64])
APGAKeyLo_EL1	Pointer Authentication Key A for Code (bits[63:0])
APIAKeyHi_EL1	Pointer Authentication Key A for Instruction (bits[127:64])
APIAKeyLo_EL1	Pointer Authentication Key A for Instruction (bits[63:0])
APIBKeyHi_EL1	Pointer Authentication Key B for Instruction (bits[127:64])
APIBKeyLo_EL1	Pointer Authentication Key B for Instruction (bits[63:0])
AT S12E0R	Address Translate Stages 1 and 2 EL0 Read
AT S12E0W	Address Translate Stages 1 and 2 EL0 Write
AT S12E1R	Address Translate Stages 1 and 2 EL1 Read
AT S12E1W	Address Translate Stages 1 and 2 EL1 Write
AT S1E0R	Address Translate Stage 1 EL0 Read
AT S1E0W	Address Translate Stage 1 EL0 Write
AT S1E1A	Address Translate Stage 1 EL1 Without Permission checks
AT S1E1R	Address Translate Stage 1 EL1 Read
AT S1E1RP	Address Translate Stage 1 EL1 Read PAN
AT S1E1W	Address Translate Stage 1 EL1 Write
AT S1E1WP	Address Translate Stage 1 EL1 Write PAN
AT S1E2A	Address Translate Stage 1 EL2 Without Permission checks
AT S1E2R	Address Translate Stage 1 EL2 Read
AT S1E2W	Address Translate Stage 1 EL2 Write
AT S1E3A	Address Translate Stage 1 EL3 Without Permission checks
AT S1E3R	Address Translate Stage 1 EL3 Read
AT S1E3W	Address Translate Stage 1 EL3 Write
BRB IALL	Invalidate the Branch Record Buffer
BRB INJ	Branch Record Injection into the Branch Record Buffer
BRBCR_EL1	Branch Record Buffer Control Register (EL1)
BRBCR_EL2	Branch Record Buffer Control Register (EL2)
BRBFCR_EL1	Branch Record Buffer Function Control Register
BRBIDR0_EL1	Branch Record Buffer ID0 Register
BRBINFINJ_EL1	Branch Record Buffer Information Injection Register
BRBSRCINJ_EL1	Branch Record Buffer Source Address Injection Register
BRBTGTINJ_EL1	Branch Record Buffer Target Address Injection Register
BRBTS_EL1	Branch Record Buffer Timestamp Register
CCSIDR2_EL1	Current Cache Size ID Register 2
CCSIDR_EL1	Current Cache Size ID Register
CFP RCTX	Control Flow Prediction Restriction by Context
CLIDR_EL1	Cache Level ID Register
CNTFRQ_EL0	Counter-timer Frequency Register
CNTHCTL_EL2	Counter-timer Hypervisor Control Register
CNTHP_CTL_EL2	Counter-timer Hypervisor Physical Timer Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTHP_CVAL_EL2	Counter-timer Physical Timer CompareValue Register (EL2)
CNTHP_TVAL_EL2	Counter-timer Physical Timer TimerValue Register (EL2)
CNTHPS_CTL_EL2	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL_EL2	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL_EL2	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL_EL2	Counter-timer Virtual Timer Control Register (EL2)
CNTHV_CVAL_EL2	Counter-timer Virtual Timer CompareValue Register (EL2)
CNTHV_TVAL_EL2	Counter-timer Virtual Timer TimerValue Register (EL2)
CNTHVS_CTL_EL2	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL_EL2	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL_EL2	Counter-timer Secure Virtual Timer TimerValue Register (EL2)
CNTKCTL_EL1	Counter-timer Kernel Control Register
CNTP_CTL_EL0	Counter-timer Physical Timer Control Register
CNTP_CVAL_EL0	Counter-timer Physical Timer CompareValue Register
CNTP_TVAL_EL0	Counter-timer Physical Timer TimerValue Register
CNTPCT_EL0	Counter-timer Physical Count Register
CNTPCTSS_EL0	Counter-timer Self-Synchronized Physical Count Register
CNTPOFF_EL2	Counter-timer Physical Offset Register
CNTPS_CTL_EL1	Counter-timer Physical Secure Timer Control Register
CNTPS_CVAL_EL1	Counter-timer Physical Secure Timer CompareValue Register
CNTPS_TVAL_EL1	Counter-timer Physical Secure Timer TimerValue Register
CNTV_CTL_EL0	Counter-timer Virtual Timer Control Register
CNTV_CVAL_EL0	Counter-timer Virtual Timer CompareValue Register
CNTV_TVAL_EL0	Counter-timer Virtual Timer TimerValue Register
CNTVCT_EL0	Counter-timer Virtual Count Register
CNTVCTSS_EL0	Counter-timer Self-Synchronized Virtual Count Register
CNTVOFF_EL2	Counter-timer Virtual Offset Register
CONTEXTIDR_EL1	Context ID Register (EL1)
CONTEXTIDR_EL2	Context ID Register (EL2)
COSP RCTX	Clear Other Speculative Prediction Restriction by Context
CPACR_EL1	Architectural Feature Access Control Register
CPACRMASK_EL1	Architectural Feature Access Control Masking Register
CPP RCTX	Cache Prefetch Prediction Restriction by Context
CPTR_EL2	Architectural Feature Trap Register (EL2)
CPTR_EL3	Architectural Feature Trap Register (EL3)
CPTRMASK_EL2	Architectural Feature Trap Masking Register
CSSELR_EL1	Cache Size Selection Register
CTR_EL0	Cache Type Register
CurrentEL	Current Exception Level
DACR32_EL2	Domain Access Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DAIF	Interrupt Mask Bits
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTR_EL0	Debug Data Transfer Register, half-duplex
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
DBGPRCR_EL1	Debug Power Control Register
DBGVCR32_EL2	Debug Vector Catch Register
DC CGDSW	Clean of Data and Allocation Tags by Set-Way
DC CGDVAC	Clean of Data and Allocation Tags by VA to PoC
DC CGDVADP	Clean of Data and Allocation Tags by VA to PoDP
DC CGDVAOC	Clean of Data and Allocation Tags by VA to Outer Cache level
DC CGDVAP	Clean of Data and Allocation Tags by VA to PoP
DC CGSW	Clean of Allocation Tags by Set-Way
DC CGVAC	Clean of Allocation Tags by VA to PoC
DC CGVADP	Clean of Allocation Tags by VA to PoDP
DC CGVAP	Clean of Allocation Tags by VA to PoP
DC CIGDPAE	Clean and invalidate of data and allocation tags by PA to PoE
DC CIGDPAPA	Clean and Invalidate of Data and Allocation Tags by PA to PoPA
DC CIGDSW	Clean and Invalidate of Data and Allocation Tags by Set-Way
DC CIGDVAC	Clean and Invalidate of Data and Allocation Tags by VA to PoC
DC CIGDVAOC	Clean and Invalidate of Data and Allocation Tags by VA to Outer Cache level
DC CIGDVAPS	Clean and Invalidate of Data and Allocation Tags by VA to PoPS
DC CIGSW	Clean and Invalidate of Allocation Tags by Set-Way
DC CIGVAC	Clean and Invalidate of Allocation Tags by VA to PoC
DC CIPAE	Data or unified Cache line Clean and Invalidate by PA to PoE
DC CIPAPA	Data or unified Cache line Clean and Invalidate by PA to PoPA
DC CISW	Data or unified Cache line Clean and Invalidate by Set-Way
DC CIVAC	Data or unified Cache line Clean and Invalidate by VA to PoC
DC CIVAOC	Data or unified Cache line Clean and Invalidate by VA to Outer Cache level
DC CIVAPS	Clean and Invalidate of Data by VA to PoPS
DC CSW	Data or unified Cache line Clean by Set-Way
DC CVAC	Data or unified Cache line Clean by VA to PoC
DC CVADP	Data or unified Cache line Clean by VA to PoDP
DC CVAOC	Data or unified Cache line Clean by VA to Outer Cache level
DC CVAP	Data or unified Cache line Clean by VA to PoP
DC CVAU	Data or unified Cache line Clean by VA to PoU
DC GVA	Data Cache set Allocation Tag by VA

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DC GZVA	Data Cache set Allocation Tags and Zero by VA
DC IGDSW	Invalidate of Data and Allocation Tags by Set-Way
DC IGDVAC	Invalidate of Data and Allocation Tags by VA to PoC
DC IGSW	Invalidate of Allocation Tags by Set-Way
DC IGVAC	Invalidate of Allocation Tags by VA to PoC
DC ISW	Data or unified Cache line Invalidate by Set-Way
DC IVAC	Data or unified Cache line Invalidate by VA to PoC
DC ZVA	Data Cache Zero by VA
DCZID_EL0	Data Cache Zero ID Register
DISR_EL1	Deferred Interrupt Status Register
DIT	Data Independent Timing
DLR_EL0	Debug Link Register
DSPSR_EL0	Debug Saved Program Status Register
DVP RCTX	Data Value Prediction Restriction by Context
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ELR_EL3	Exception Link Register (EL3)
ERRIDR_EL1	Error Record ID Register
ERRSELR_EL1	Error Record Select Register
ERXADDR_EL1	Selected Error Record Address Register
ERXCTLR_EL1	Selected Error Record Control Register
ERXFR_EL1	Selected Error Record Feature Register
ERXGSR_EL1	Selected Error Record Group Status Register
ERXMISC0_EL1	Selected Error Record Miscellaneous Register 0
ERXMISC1_EL1	Selected Error Record Miscellaneous Register 1
ERXMISC2_EL1	Selected Error Record Miscellaneous Register 2
ERXMISC3_EL1	Selected Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	Selected Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	Selected Pseudo-fault Generation Control Register
ERXPFGF_EL1	Selected Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	Selected Error Record Primary Status Register
ESR_EL1	Exception Syndrome Register (EL1)
ESR_EL2	Exception Syndrome Register (EL2)
ESR_EL3	Exception Syndrome Register (EL3)
FAR_EL1	Fault Address Register (EL1)
FAR_EL2	Fault Address Register (EL2)
FAR_EL3	Fault Address Register (EL3)
FGWTE3_EL3	Fine-Grained Write Traps EL3
FPCR	Floating-point Control Register
FPEXC32_EL2	Floating-Point Exception Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
FPMR	Floating-point Mode Register
FPSR	Floating-point Status Register
GCR_EL1	Tag Control Register.
GCSCR_EL1	Guarded Control Stack Control Register (EL1)
GCSCR_EL2	Guarded Control Stack Control Register (EL2)
GCSCR_EL3	Guarded Control Stack Control Register (EL3)
GCSCRE0_EL1	Guarded Control Stack Control Register (EL0)
GCSPOPCX	Guarded Control Stack Pop and Compare exception return record
GCSPOPM	Guarded Control Stack Pop
GCSPOPX	Guarded Control Stack Pop exception return record
GCSPR_EL0	Guarded Control Stack Pointer Register (EL0)
GCSPR_EL1	Guarded Control Stack Pointer Register (EL1)
GCSPR_EL2	Guarded Control Stack Pointer Register (EL2)
GCSPR_EL3	Guarded Control Stack Pointer Register (EL3)
GCSPUSHM	Guarded Control Stack Push
GCSPUSHX	Guarded Control Stack Push exception return record
GCSSS1	Guarded Control Stack Switch Stack 1
GCSSS2	Guarded Control Stack Switch Stack 2
GMID_EL1	Multiple tag transfer ID Register
GPCBW_EL3	Granule Protection Check Bypass Window Register (EL3)
GPCCR_EL3	Granule Protection Check Control Register (EL3)
GPTBR_EL3	Granule Protection Table Base Register
HACDBSBR_EL2	Hardware Accelerator for Cleaning Dirty State Base Register
HACDBSCONS_EL2	Hardware Accelerator for Cleaning Dirty State Consumer Register
HACR_EL2	Hypervisor Auxiliary Control Register
HAFGRTR_EL2	Hypervisor Activity Monitors Fine-Grained Read Trap Register
HCR_EL2	Hypervisor Configuration Register
HCRX_EL2	Extended Hypervisor Configuration Register
HDBSSBR_EL2	Hardware Dirty State Tracking Structure Base Register
HDBSSPROD_EL2	Hardware Dirty State Tracking Structure Producer Register
HDFGRTR2_EL2	Hypervisor Debug Fine-Grained Read Trap Register 2
HDFGRTR_EL2	Hypervisor Debug Fine-Grained Read Trap Register
HDFGWTR2_EL2	Hypervisor Debug Fine-Grained Write Trap Register 2
HDFGWTR_EL2	Hypervisor Debug Fine-Grained Write Trap Register
HFGITR2_EL2	Hypervisor Fine-Grained Instruction Trap Register 2
HFGITR_EL2	Hypervisor Fine-Grained Instruction Trap Register
HFGRTR2_EL2	Hypervisor Fine-Grained Read Trap Register 2
HFGRTR_EL2	Hypervisor Fine-Grained Read Trap Register
HFGWTR2_EL2	Hypervisor Fine-Grained Write Trap Register 2
HFGWTR_EL2	Hypervisor Fine-Grained Write Trap Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
HPFAR_EL2	Hypervisor IPA Fault Address Register
HSTR_EL2	Hypervisor System Trap Register
IC IALLU	Instruction Cache Invalidate All to PoU
IC IALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
IC IVAU	Instruction Cache line Invalidate by VA to PoU
ICC_ASGI1R_EL1	Interrupt Controller Alias Software Generated Interrupt Group 1 Register
ICC_BPR0_EL1	Interrupt Controller Binary Point Register 0
ICC_BPR1_EL1	Interrupt Controller Binary Point Register 1
ICC_CTLR_EL1	Interrupt Controller Control Register (EL1)
ICC_CTLR_EL3	Interrupt Controller Control Register (EL3)
ICC_DIR_EL1	Interrupt Controller Deactivate Interrupt Register
ICC_EOIR0_EL1	Interrupt Controller End Of Interrupt Register 0
ICC_EOIR1_EL1	Interrupt Controller End Of Interrupt Register 1
ICC_HPPIR0_EL1	Interrupt Controller Highest Priority Pending Interrupt Register 0
ICC_HPPIR1_EL1	Interrupt Controller Highest Priority Pending Interrupt Register 1
ICC_IAR0_EL1	Interrupt Controller Interrupt Acknowledge Register 0
ICC_IAR1_EL1	Interrupt Controller Interrupt Acknowledge Register 1
ICC_IGRPEN0_EL1	Interrupt Controller Interrupt Group 0 Enable Register
ICC_IGRPEN1_EL1	Interrupt Controller Interrupt Group 1 Enable Register
ICC_IGRPEN1_EL3	Interrupt Controller Interrupt Group 1 Enable Register (EL3)
ICC_NMIAR1_EL1	Interrupt Controller Non-maskable Interrupt Acknowledge Register 1
ICC_PMR_EL1	Interrupt Controller Interrupt Priority Mask Register
ICC_RPR_EL1	Interrupt Controller Running Priority Register
ICC_SGI0R_EL1	Interrupt Controller Software Generated Interrupt Group 0 Register
ICC_SGI1R_EL1	Interrupt Controller Software Generated Interrupt Group 1 Register
ICC_SRE_EL1	Interrupt Controller System Register Enable Register (EL1)
ICC_SRE_EL2	Interrupt Controller System Register Enable Register (EL2)
ICC_SRE_EL3	Interrupt Controller System Register Enable Register (EL3)
ICH_EISR_EL2	Interrupt Controller End of Interrupt Status Register
ICH_ELRSR_EL2	Interrupt Controller Empty List Register Status Register
ICH_HCR_EL2	Interrupt Controller Hyp Control Register
ICH_MISR_EL2	Interrupt Controller Maintenance Interrupt State Register
ICH_VMCR_EL2	Interrupt Controller Virtual Machine Control Register
ICH_VTR_EL2	Interrupt Controller VGIC Type Register
ICV_BPR0_EL1	Interrupt Controller Virtual Binary Point Register 0
ICV_BPR1_EL1	Interrupt Controller Virtual Binary Point Register 1
ICV_CTLR_EL1	Interrupt Controller Virtual Control Register
ICV_DIR_EL1	Interrupt Controller Deactivate Virtual Interrupt Register
ICV_EOIR0_EL1	Interrupt Controller Virtual End Of Interrupt Register 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ICV_EOIR1_EL1	Interrupt Controller Virtual End Of Interrupt Register 1
ICV_HPPIR0_EL1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0
ICV_HPPIR1_EL1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1
ICV_IAR0_EL1	Interrupt Controller Virtual Interrupt Acknowledge Register 0
ICV_IAR1_EL1	Interrupt Controller Virtual Interrupt Acknowledge Register 1
ICV_IGRPEN0_EL1	Interrupt Controller Virtual Interrupt Group 0 Enable Register
ICV_IGRPEN1_EL1	Interrupt Controller Virtual Interrupt Group 1 Enable Register
ICV_NMIAR1_EL1	Interrupt Controller Virtual Non-maskable Interrupt Acknowledge Register 1
ICV_PMR_EL1	Interrupt Controller Virtual Interrupt Priority Mask Register
ICV_RPR_EL1	Interrupt Controller Virtual Running Priority Register
ID_AA64AFR0_EL1	AArch64 Auxiliary Feature Register 0
ID_AA64AFR1_EL1	AArch64 Auxiliary Feature Register 1
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0
ID_AA64DFR1_EL1	AArch64 Debug Feature Register 1
ID_AA64DFR2_EL1	AArch64 Debug Feature Register 2
ID_AA64FPFR0_EL1	AArch64 Floating-point Feature Register 0
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2
ID_AA64ISAR3_EL1	AArch64 Instruction Set Attribute Register 3
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64MMFR3_EL1	AArch64 Memory Model Feature Register 3
ID_AA64MMFR4_EL1	AArch64 Memory Model Feature Register 4
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
ID_AA64PFR2_EL1	AArch64 Processor Feature Register 2
ID_AA64SMFR0_EL1	SME Feature ID Register 0
ID_AA64ZFR0_EL1	SVE Feature ID Register 0
ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0_EL1	AArch32 Instruction Set Attribute Register 0
ID_ISAR1_EL1	AArch32 Instruction Set Attribute Register 1
ID_ISAR2_EL1	AArch32 Instruction Set Attribute Register 2
ID_ISAR3_EL1	AArch32 Instruction Set Attribute Register 3
ID_ISAR4_EL1	AArch32 Instruction Set Attribute Register 4

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_ISAR5_EL1	AArch32 Instruction Set Attribute Register 5
ID_ISAR6_EL1	AArch32 Instruction Set Attribute Register 6
ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0_EL1	AArch32 Processor Feature Register 0
ID_PFR1_EL1	AArch32 Processor Feature Register 1
ID_PFR2_EL1	AArch32 Processor Feature Register 2
IFSR32_EL2	Instruction Fault Status Register (EL2)
ISR_EL1	Interrupt Status Register
LORC_EL1	LORegion Control (EL1)
LOREA_EL1	LORegion End Address (EL1)
LORID_EL1	LORegionID (EL1)
LORN_EL1	LORegion Number (EL1)
LORSA_EL1	LORegion Start Address (EL1)
MAIR2_EL1	Extended Memory Attribute Indirection Register (EL1)
MAIR2_EL2	Extended Memory Attribute Indirection Register (EL2)
MAIR2_EL3	Extended Memory Attribute Indirection Register (EL3)
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)
MAIR_EL3	Memory Attribute Indirection Register (EL3)
MDCCINT_EL1	Monitor DCC Interrupt Enable Register
MDCCSR_EL0	Monitor DCC Status Register
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDCR_EL3	Monitor Debug Configuration Register (EL3)
MDRAR_EL1	Monitor Debug ROM Address Register
MDSCR_EL1	Monitor Debug System Control Register
MDSELR_EL1	Breakpoint and Watchpoint Selection Register
MDSTEPOP_EL1	Monitor Debug Step Opcode Register
MECID_A0_EL2	Alternate MECID for EL2 and EL2&0 translation regimes
MECID_A1_EL2	Alternate MECID for EL2&0 translation regimes.
MECID_P0_EL2	Primary MECID for EL2 and EL2&0 translation regimes
MECID_P1_EL2	Primary MECID for EL2&0 translation regimes
MECID_RL_A_EL3	Realm PA space Alternate MECID for EL3 stage 1 translation regime
MECIDR_EL2	MEC Identification Register
MFAR_EL3	Physical Fault Address Register (EL3)
MIDR_EL1	Main ID Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MPAM0_EL1	MPAM0 Register (EL1)
MPAM1_EL1	MPAM1 Register (EL1)
MPAM2_EL2	MPAM2 Register (EL2)
MPAM3_EL3	MPAM3 Register (EL3)
MPAMBW0_EL1	MPAM PE-side Maximum Bandwidth Control Register (EL0)
MPAMBW1_EL1	MPAM PE-side Maximum Bandwidth Control Register (EL1)
MPAMBW2_EL2	MPAM PE-side Maximum Bandwidth Control Register (EL2)
MPAMBW3_EL3	MPAM PE-side Maximum Bandwidth Control Register (EL3)
MPAMBWCAP_EL2	MPAM PE-side Maximum Bandwidth Limit Virtualization Register
MPAMBWIDR_EL1	MPAM PE-side Bandwidth Controls ID Register
MPAMBWSM_EL1	MPAM Streaming Mode Bandwidth Control Register (EL1)
MPAMHCR_EL2	MPAM Hypervisor Control Register (EL2)
MPAMIDR_EL1	MPAM ID Register (EL1)
MPAMSM_EL1	MPAM Streaming Mode Register
MPAMVPM0_EL2	MPAM Virtual PARTID Mapping Register 0
MPAMVPM1_EL2	MPAM Virtual PARTID Mapping Register 1
MPAMVPM2_EL2	MPAM Virtual PARTID Mapping Register 2
MPAMVPM3_EL2	MPAM Virtual PARTID Mapping Register 3
MPAMVPM4_EL2	MPAM Virtual PARTID Mapping Register 4
MPAMVPM5_EL2	MPAM Virtual PARTID Mapping Register 5
MPAMVPM6_EL2	MPAM Virtual PARTID Mapping Register 6
MPAMVPM7_EL2	MPAM Virtual PARTID Mapping Register 7
MPAMVPMV_EL2	MPAM Virtual Partition Mapping Valid Register
MPIDR_EL1	Multiprocessor Affinity Register
MVFR0_EL1	AArch32 Media and VFP Feature Register 0
MVFR1_EL1	AArch32 Media and VFP Feature Register 1
MVFR2_EL1	AArch32 Media and VFP Feature Register 2
NZCV	Condition Flags
OSDLR_EL1	OS Double Lock Register
OSDTRRX_EL1	OS Lock Data Transfer Register, Receive
OSDTRTX_EL1	OS Lock Data Transfer Register, Transmit
OSECCR_EL1	OS Lock Exception Catch Control Register
OSLAR_EL1	OS Lock Access Register
OSLSR_EL1	OS Lock Status Register
PAN	Privileged Access Never
PAR_EL1	Physical Address Register
PFAR_EL1	Physical Fault Address Register (EL1)
PFAR_EL2	Physical Fault Address Register (EL2)
PIR_EL1	Permission Indirection Register 1 (EL1)
PIR_EL2	Permission Indirection Register 2 (EL2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PIR_EL3	Permission Indirection Register 3 (EL3)
PIRE0_EL1	Permission Indirection Register 0 (EL1)
PIRE0_EL2	Permission Indirection Register 0 (EL2)
PM	Profiling Exception Mask
PMBIDR_EL1	Profiling Buffer ID Register
PMBLIMITR_EL1	Profiling Buffer Limit Address Register
PMBMAR_EL1	Profiling Buffer Memory Attribute Register
PMBPTR_EL1	Profiling Buffer Write Pointer Register
PMBSR_EL1	Profiling Buffer Status-syndrome Register (EL1)
PMBSR_EL2	Profiling Buffer Syndrome Register (EL2)
PMBSR_EL3	Profiling Buffer Syndrome Register (EL3)
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Count Register
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0_EL0	Performance Monitors Common Event Identification Register 0
PMCEID1_EL0	Performance Monitors Common Event Identification Register 1
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMECR_EL1	Performance Monitors Extended Control Register (EL1)
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMICNTSVR_EL1	Performance Monitors Instruction Count Saved Value Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMMIR_EL1	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear Register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMSCR_EL1	Statistical Profiling Control Register (EL1)
PMSCR_EL2	Statistical Profiling Control Register (EL2)
PMSDSFR_EL1	Sampling Data Source Filter Register
PMSELR_EL0	Performance Monitors Event Counter Selection Register
PMSEVFR_EL1	Sampling Event Filter Register
PMSFCR_EL1	Sampling Filter Control Register
PMSICR_EL1	Sampling Interval Counter Register
PMSIDR_EL1	Sampling Profiling ID Register
PMSIRR_EL1	Sampling Interval Reload Register
PMSLATFR_EL1	Sampling Latency Filter Register
PMSNEVFR_EL1	Sampling Inverted Event Filter Register
PMSSCR_EL1	Performance Monitors Snapshot Status and Capture Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMSWINC_EL0	Performance Monitors Software Increment Register
PMUACR_EL1	Performance Monitors User Access Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
PMXEVCNTR_EL0	Performance Monitors Selected Event Count Register
PMXEVTYPER_EL0	Performance Monitors Selected Event Type Register
PMZR_EL0	Performance Monitors Zero with Mask
POR_EL0	Permission Overlay Register 0 (EL0)
POR_EL1	Permission Overlay Register 1 (EL1)
POR_EL2	Permission Overlay Register 2 (EL2)
POR_EL3	Permission Overlay Register 3 (EL3)
RCWMASK_EL1	Read Check Write Instruction Mask (EL1)
RCWSMASK_EL1	Software Read Check Write Instruction Mask (EL1)
REVIDR_EL1	Revision ID Register
RGSR_EL1	Random Allocation Tag Seed Register.
RMR_EL1	Reset Management Register (EL1)
RMR_EL2	Reset Management Register (EL2)
RMR_EL3	Reset Management Register (EL3)
RNDR	Random Number
RNDRRS	Random Number Full Entropy
RVBAR_EL1	Reset Vector Base Address Register (if EL2 and EL3 not implemented)
RVBAR_EL2	Reset Vector Base Address Register (if EL3 not implemented)
RVBAR_EL3	Reset Vector Base Address Register (if EL3 implemented)
S1_<op1><*Cn*><Cm>_<op2>	IMPLEMENTATION DEFINED System instructions
S2PIR_EL2	Stage 2 Permission Indirection Register (EL2)
S2POR_EL1	Stage 2 Permission Overlay Register (EL1)
S3_<op1><*Cn*><Cm>_<op2>	IMPLEMENTATION DEFINED Registers
SCR_EL3	Secure Configuration Register
SCTLR2_EL1	System Control Register (EL1)
SCTLR2_EL2	System Control Register (EL2)
SCTLR2_EL3	System Control Register (EL3)
SCTLR2MASK_EL1	Extended System Control Masking Register (EL1)
SCTLR2MASK_EL2	Extended System Control Masking Register (EL2)
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)
SCTLR_EL3	System Control Register (EL3)
SCTLRMASK_EL1	System Control Masking Register (EL1)
SCTLRMASK_EL2	System Control Masking Register (EL2)
SCXTNUM_EL0	EL0 Read-Write Software Context Number
SCXTNUM_EL1	EL1 Read-Write Software Context Number
SCXTNUM_EL2	EL2 Read-Write Software Context Number

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SCXTNUM_EL3	EL3 Read-Write Software Context Number
SDER32_EL2	AArch32 Secure Debug Enable Register
SDER32_EL3	AArch32 Secure Debug Enable Register
SMCR_EL1	SME Control Register (EL1)
SMCR_EL2	SME Control Register (EL2)
SMCR_EL3	SME Control Register (EL3)
SMIDR_EL1	Streaming Mode Identification Register
SMPRI_EL1	Streaming Mode Priority Register
SMPRIMAP_EL2	Streaming Mode Priority Mapping Register
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL1)
SP_EL2	Stack Pointer (EL2)
SP_EL3	Stack Pointer (EL3)
SPMACCESSR_EL1	System Performance Monitors Access Register (EL1)
SPMACCESSR_EL2	System Performance Monitors Access Register (EL2)
SPMACCESSR_EL3	System Performance Monitors Access Register (EL3)
SPMCFGR_EL1	System Performance Monitors Configuration Register
SPMCNTENCLR_EL0	System Performance Monitors Count Enable Clear Register
SPMCNTENSET_EL0	System Performance Monitors Count Enable Set Register
SPMCR_EL0	System Performance Monitor Control Register
SPMDEVAFF_EL1	System Performance Monitors Device Affinity Register
SPMDEVARCH_EL1	System Performance Monitors Device Architecture Register
SPMIIDR_EL1	System PMU Implementation Identification Register
SPMINTENCLR_EL1	System Performance Monitors Interrupt Enable Clear Register
SPMINTENSET_EL1	System Performance Monitors Interrupt Enable Set Register
SPMOVSCLR_EL0	System Performance Monitors Overflow Flag Status Clear Register
SPMOVSSET_EL0	System Performance Monitors Overflow Flag Status Set Register
SPMROOTCR_EL3	System Performance Monitors Root and Realm Control Register
SPMSCR_EL1	System Performance Monitors Secure Control Register
SPMSELR_EL0	System Performance Monitors Select Register
SPMZR_EL0	System Performance Monitors Zero with Mask
SPSel	Stack Pointer Select
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_EL3	Saved Program Status Register (EL3)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_und	Saved Program Status Register (Undefined mode)
SSBS	Speculative Store Bypass Safe

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SVCR	Streaming Vector Control Register
TCO	Tag Check Override
TCR2_EL1	Extended Translation Control Register (EL1)
TCR2_EL2	Extended Translation Control Register (EL2)
TCR2MASK_EL1	Extended Translation Control Masking Register (EL1)
TCR2MASK_EL2	Extended Translation Control Masking Register (EL2)
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TCR_EL3	Translation Control Register (EL3)
TCRMASK_EL1	Translation Control Masking Register (EL1)
TCRMASK_EL2	Translation Control Masking Register (EL2)
TFSR_EL1	Tag Fault Status Register (EL1)
TFSR_EL2	Tag Fault Status Register (EL2)
TFSR_EL3	Tag Fault Status Register (EL3)
TFSRE0_EL1	Tag Fault Status Register (EL0).
TLBI ALLE1, TLBI ALLE1NXS	TLB Invalidate All, EL1
TLBI ALLE1IS, TLBI ALLE1ISNXS	TLB Invalidate All, EL1, Inner Shareable
TLBI ALLE1OS, TLBI ALLE1OSNXS	TLB Invalidate All, EL1, Outer Shareable
TLBI ALLE2, TLBI ALLE2NXS	TLB Invalidate All, EL2
TLBI ALLE2IS, TLBI ALLE2ISNXS	TLB Invalidate All, EL2, Inner Shareable
TLBI ALLE2OS, TLBI ALLE2OSNXS	TLB Invalidate All, EL2, Outer Shareable
TLBI ALLE3, TLBI ALLE3NXS	TLB Invalidate All, EL3
TLBI ALLE3IS, TLBI ALLE3ISNXS	TLB Invalidate All, EL3, Inner Shareable
TLBI ALLE3OS, TLBI ALLE3OSNXS	TLB Invalidate All, EL3, Outer Shareable
TLBI ASIDE1, TLBI ASIDE1NXS	TLB Invalidate by ASID, EL1
TLBI ASIDE1IS, TLBI ASIDE1ISNXS	TLB Invalidate by ASID, EL1, Inner Shareable
TLBI ASIDE1OS, TLBI ASIDE1OSNXS	TLB Invalidate by ASID, EL1, Outer Shareable
TLBI IPAS2E1, TLBI IPAS2E1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI IPAS2LE1, TLBI IPAS2LE1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI PAALL	TLB Invalidate GPT Information by PA, All Entries, Local
TLBI PAALLOS	TLB Invalidate GPT Information by PA, All Entries, Outer Shareable
TLBI RIPAS2E1, TLBI RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RPALOS	TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable
TLBI RPAOS	TLB Range Invalidate GPT Information by PA, Outer Shareable
TLBI RVAAE1, TLBI RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBI RVAAE1IS, TLBI RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI RVAAE1OS, TLBI RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI RVAALE1, TLBI RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBI RVAALE1IS, TLBI RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI RVAALE1OS, TLBI RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI RVAE1, TLBI RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBI RVAE1IS, TLBI RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBI RVAE1OS, TLBI RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBI RVAE2, TLBI RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBI RVAE2IS, TLBI RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBI RVAE2OS, TLBI RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBI RVAE3, TLBI RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBI RVAE3IS, TLBI RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBI RVAE3OS, TLBI RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBI RVALE1, TLBI RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBI RVALE1IS, TLBI RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBI RVALE1OS, TLBI RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBI RVALE2, TLBI RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBI RVALE2IS, TLBI RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBI RVALE2OS, TLBI RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBI RVALE3, TLBI RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBI RVALE3IS, TLBI RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBI RVALE3OS, TLBI RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VAAE1, TLBI VAAE1NXS	TLB Invalidate by VA, All ASID, EL1
TLBI VAAE1IS, TLBI VAAE1ISNXS	TLB Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI VAAE1OS, TLBI VAAE1OSNXS	TLB Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI VAALE1, TLBI VAALE1NXS	TLB Invalidate by VA, All ASID, Last level, EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBI VAALE1IS, TLBI VAALE1ISNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI VAALE1OS, TLBI VAALE1OSNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI VAE1, TLBI VAE1NXS	TLB Invalidate by VA, EL1
TLBI VAE1IS, TLBI VAE1ISNXS	TLB Invalidate by VA, EL1, Inner Shareable
TLBI VAE1OS, TLBI VAE1OSNXS	TLB Invalidate by VA, EL1, Outer Shareable
TLBI VAE2, TLBI VAE2NXS	TLB Invalidate by VA, EL2
TLBI VAE2IS, TLBI VAE2ISNXS	TLB Invalidate by VA, EL2, Inner Shareable
TLBI VAE2OS, TLBI VAE2OSNXS	TLB Invalidate by VA, EL2, Outer Shareable
TLBI VAE3, TLBI VAE3NXS	TLB Invalidate by VA, EL3
TLBI VAE3IS, TLBI VAE3ISNXS	TLB Invalidate by VA, EL3, Inner Shareable
TLBI VAE3OS, TLBI VAE3OSNXS	TLB Invalidate by VA, EL3, Outer Shareable
TLBI VALE1, TLBI VALE1NXS	TLB Invalidate by VA, Last level, EL1
TLBI VALE1IS, TLBI VALE1ISNXS	TLB Invalidate by VA, Last level, EL1, Inner Shareable
TLBI VALE1OS, TLBI VALE1OSNXS	TLB Invalidate by VA, Last level, EL1, Outer Shareable
TLBI VALE2, TLBI VALE2NXS	TLB Invalidate by VA, Last level, EL2
TLBI VALE2IS, TLBI VALE2ISNXS	TLB Invalidate by VA, Last level, EL2, Inner Shareable
TLBI VALE2OS, TLBI VALE2OSNXS	TLB Invalidate by VA, Last level, EL2, Outer Shareable
TLBI VALE3, TLBI VALE3NXS	TLB Invalidate by VA, Last level, EL3
TLBI VALE3IS, TLBI VALE3ISNXS	TLB Invalidate by VA, Last level, EL3, Inner Shareable
TLBI VALE3OS, TLBI VALE3OSNXS	TLB Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VMALLE1, TLBI VMALLE1NXS	TLB Invalidate by VMID, All at stage 1, EL1
TLBI VMALLE1IS, TLBI VMALLE1ISNXS	TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable
TLBI VMALLE1OS, TLBI VMALLE1OSNXS	TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable
TLBI VMALLS12E1, TLBI VMALLS12E1NXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1
TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable
TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable
TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0
TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0, Inner Shareable
TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS	TLB Invalidate stage 2 write permission by VMID, EL1&0, Outer Shareable
TLBIP IPAS2E1, TLBIP IPAS2E1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1
TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RVAAE1, TLBIP RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBIP RVAALE1, TLBIP RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP RVAE1, TLBIP RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBIP RVAE1IS, TLBIP RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBIP RVAE1OS, TLBIP RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBIP RVAE2, TLBIP RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBIP RVAE2IS, TLBIP RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBIP RVAE2OS, TLBIP RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBIP RVAE3, TLBIP RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBIP RVAE3IS, TLBIP RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBIP RVAE3OS, TLBIP RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBIP RVALE1, TLBIP RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBIP RVALE1IS, TLBIP RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBIP RVALE1OS, TLBIP RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBIP RVALE2, TLBIP RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBIP RVALE2IS, TLBIP RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBIP RVALE2OS, TLBIP RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBIP RVALE3, TLBIP RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBIP RVALE3IS, TLBIP RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBIP RVALE3OS, TLBIP RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBIP VAAE1, TLBIP VAAE1NXS	TLB Invalidate Pair by VA, All ASID, EL1
TLBIP VAAE1IS, TLBIP VAAE1ISNXS	TLB Invalidate Pair by VA, All ASID, EL1, Inner Shareable
TLBIP VAAE1OS, TLBIP VAAE1OSNXS	TLB Invalidate Pair by VA, All ASID, EL1, Outer Shareable
TLBIP VAALE1, TLBIP VAALE1NXS	TLB Invalidate Pair by VA, All ASID, Last level, EL1
TLBIP VAALE1IS, TLBIP VAALE1ISNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Inner Shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBIP VAALE1OS, TLBIP VAALE1OSNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP VAE1, TLBIP VAE1NXS	TLB Invalidate Pair by VA, EL1
TLBIP VAE1IS, TLBIP VAE1ISNXS	TLB Invalidate Pair by VA, EL1, Inner Shareable
TLBIP VAE1OS, TLBIP VAE1OSNXS	TLB Invalidate Pair by VA, EL1, Outer Shareable
TLBIP VAE2, TLBIP VAE2NXS	TLB Invalidate Pair by VA, EL2
TLBIP VAE2IS, TLBIP VAE2ISNXS	TLB Invalidate Pair by VA, EL2, Inner Shareable
TLBIP VAE2OS, TLBIP VAE2OSNXS	TLB Invalidate Pair by VA, EL2, Outer Shareable
TLBIP VAE3, TLBIP VAE3NXS	TLB Invalidate Pair by VA, EL3
TLBIP VAE3IS, TLBIP VAE3ISNXS	TLB Invalidate Pair by VA, EL3, Inner Shareable
TLBIP VAE3OS, TLBIP VAE3OSNXS	TLB Invalidate Pair by VA, EL3, Outer Shareable
TLBIP VALE1, TLBIP VALE1NXS	TLB Invalidate Pair by VA, Last level, EL1
TLBIP VALE1IS, TLBIP VALE1ISNXS	TLB Invalidate Pair by VA, Last level, EL1, Inner Shareable
TLBIP VALE1OS, TLBIP VALE1OSNXS	TLB Invalidate Pair by VA, Last level, EL1, Outer Shareable
TLBIP VALE2, TLBIP VALE2NXS	TLB Invalidate Pair by VA, Last level, EL2
TLBIP VALE2IS, TLBIP VALE2ISNXS	TLB Invalidate Pair by VA, Last level, EL2, Inner Shareable
TLBIP VALE2OS, TLBIP VALE2OSNXS	TLB Invalidate Pair by VA, Last level, EL2, Outer Shareable
TLBIP VALE3, TLBIP VALE3NXS	TLB Invalidate Pair by VA, Last level, EL3
TLBIP VALE3IS, TLBIP VALE3ISNXS	TLB Invalidate Pair by VA, Last level, EL3, Inner Shareable
TLBIP VALE3OS, TLBIP VALE3OSNXS	TLB Invalidate Pair by VA, Last level, EL3, Outer Shareable
TPIDR2_EL0	EL0 Read-Write Software Thread ID Register 2
TPIDR_EL0	EL0 Read-Write Software Thread ID Register
TPIDR_EL1	EL1 Software Thread ID Register
TPIDR_EL2	EL2 Software Thread ID Register
TPIDR_EL3	EL3 Software Thread ID Register
TPIDRRO_EL0	EL0 Read-Only Software Thread ID Register
TRBBASER_EL1	Trace Buffer Base Address Register
TRBIDR_EL1	Trace Buffer ID Register
TRBLIMITR_EL1	Trace Buffer Limit Address Register
TRBMAR_EL1	Trace Buffer Memory Attribute Register
TRBMPAM_EL1	Trace Buffer MPAM Configuration Register
TRBPTR_EL1	Trace Buffer Write Pointer Register
TRBSR_EL1	Trace Buffer Status-syndrome Register (EL1)
TRBSR_EL2	Trace Buffer Syndrome Register (EL2)
TRBSR_EL3	Trace Buffer Syndrome Register (EL3)
TRBTRG_EL1	Trace Buffer Trigger Counter Register
TRCAUTHSTATUS	Trace Authentication Status Register
TRCAUXCTLR	Trace Auxiliary Control Register
TRCBBCTLR	Trace Branch Broadcast Control Register
TRCCCCTLR	Trace Cycle Count Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCCIDCCTLR0	Trace Context Identifier Comparator Control Register 0
TRCCIDCCTLR1	Trace Context Identifier Comparator Control Register 1
TRCCLAIMCLR	Trace Claim Tag Clear Register
TRCCLAIMSET	Trace Claim Tag Set Register
TRCCONFIGR	Trace Configuration Register
TRCDEVARCH	Trace Device Architecture Register
TRCDEVID	Trace Device Configuration Register
TRCEVENTCTL0R	Trace Event Control 0 Register
TRCEVENTCTL1R	Trace Event Control 1 Register
TRCIDR0	Trace ID Register 0
TRCIDR1	Trace ID Register 1
TRCIDR10	Trace ID Register 10
TRCIDR11	Trace ID Register 11
TRCIDR12	Trace ID Register 12
TRCIDR13	Trace ID Register 13
TRCIDR2	Trace ID Register 2
TRCIDR3	Trace ID Register 3
TRCIDR4	Trace ID Register 4
TRCIDR5	Trace ID Register 5
TRCIDR6	Trace ID Register 6
TRCIDR7	Trace ID Register 7
TRCIDR8	Trace ID Register 8
TRCIDR9	Trace ID Register 9
TRCIMSPEC0	Trace IMP DEF Register 0
TRCIT	Trace Instrumentation
TRCITECR_EL1	Instrumentation Trace Control Register (EL1)
TRCITECR_EL2	Instrumentation Trace Control Register (EL2)
TRCITEEDCR	Instrumentation Trace Extension External Debug Control Register
TRCOSLSR	Trace OS Lock Status Register
TRCPRGCTLR	Trace Programming Control Register
TRCQCTLR	Trace Q Element Control Register
TRCRSR	Trace Resources Status Register
TRCSEQRSTEVR	Trace Sequencer Reset Control Register
TRCSEQSTR	Trace Sequencer State Register
TRCSTALLCTLR	Trace Stall Control Register
TRCSTATR	Trace Status Register
TRCSYNCPR	Trace Synchronization Period Register
TRCTRACEIDR	Trace ID Register
TRCTSCTLR	Trace Timestamp Control Register
TRCVICTLR	Trace ViewInst Main Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCVIIECTLR	Trace ViewInst Include-Exclude Control Register
TRCVIPCSSCTLR	Trace ViewInst Start-Stop PE Comparator Control Register
TRCVISSCTLR	Trace ViewInst Start-Stop Control Register
TRCVMIDCCTLR0	Trace Virtual Context Identifier Comparator Control Register 0
TRCVMIDCCTLR1	Trace Virtual Context Identifier Comparator Control Register 1
TRFCR_EL1	Trace Filter Control Register (EL1)
TRFCR_EL2	Trace Filter Control Register (EL2)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
TTBR0_EL2	Translation Table Base Register 0 (EL2)
TTBR0_EL3	Translation Table Base Register 0 (EL3)
TTBR1_EL1	Translation Table Base Register 1 (EL1)
TTBR1_EL2	Translation Table Base Register 1 (EL2)
UAO	User Access Override
VBAR_EL1	Vector Base Address Register (EL1)
VBAR_EL2	Vector Base Address Register (EL2)
VBAR_EL3	Vector Base Address Register (EL3)
VDISR_EL2	Virtual Deferred Interrupt Status Register (EL2)
VDISR_EL3	Virtual Deferred Interrupt Status Register (EL3)
VMECID_A_EL2	Alternate MECID for EL1&0 stage 2 translation regime
VMECID_P_EL2	Primary MECID for EL1&0 stage 2 translation regime
VMPIDR_EL2	Virtualization Multiprocessor ID Register
VNCR_EL2	Virtual Nested Control Register
VPIDR_EL2	Virtualization Processor ID Register
VSESR_EL2	Virtual SError Exception Syndrome Register (EL2)
VSESR_EL3	Virtual SError Exception Syndrome Register (EL3)
VSTCR_EL2	Virtualization Secure Translation Control Register
VSTTBR_EL2	Virtualization Secure Translation Table Base Register
VTCR_EL2	Virtualization Translation Control Register
VTTBR_EL2	Virtualization Translation Table Base Register
ZCR_EL1	SVE Control Register (EL1)
ZCR_EL2	SVE Control Register (EL2)
ZCR_EL3	SVE Control Register (EL3)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.3 Functional index of AArch64 registers and System instructions

This section is an index of the AArch64 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors System registers.
- Activity monitors System registers.
- Debug System registers.
- Trace System registers.
- Branch Record Buffer registers.
- RAS registers.
- Root Security state registers.
- Memory Partitioning and Monitoring registers.
- Generic timer registers.
- Cache maintenance System instructions.
- Address translation System instructions.
- TLB maintenance System instructions.
- Prediction restriction System instructions.
- Base System registers.

K14.3.1 Special-purpose registers

This section is an index to the registers in the Special-purpose registers functional group.

Table K14-10 Special-purpose registers

Register	Description, see
DLR_EL0	Debug Link Register
DSPSR_EL0	Debug Saved Program Status Register
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ELR_EL3	Exception Link Register (EL3)
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL1)
SP_EL2	Stack Pointer (EL2)
SP_EL3	Stack Pointer (EL3)
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_EL3	Saved Program Status Register (EL3)
SPSR_fiq	Saved Program Status Register (FIQ mode)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_und	Saved Program Status Register (Undefined mode)

K14.3.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Table K14-11 VMSA-specific registers

Register	Description, see
AMAIR2_EL1	Extended Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR2_EL2	Extended Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR2_EL3	Extended Auxiliary Memory Attribute Indirection Register (EL3)
AMAIR_EL1	Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR_EL2	Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR_EL3	Auxiliary Memory Attribute Indirection Register (EL3)
CONTEXTIDR_EL1	Context ID Register (EL1)
CONTEXTIDR_EL2	Context ID Register (EL2)
DACR32_EL2	Domain Access Control Register
GPCBW_EL3	Granule Protection Check Bypass Window Register (EL3)
GPCCR_EL3	Granule Protection Check Control Register (EL3)
GPTBR_EL3	Granule Protection Table Base Register
HACDBSBR_EL2	Hardware Accelerator for Cleaning Dirty State Base Register
HACDBSCONS_EL2	Hardware Accelerator for Cleaning Dirty State Consumer Register
HDBSSBR_EL2	Hardware Dirty State Tracking Structure Base Register
HDBSSPROD_EL2	Hardware Dirty State Tracking Structure Producer Register
LORC_EL1	LORegion Control (EL1)
LOREA_EL1	LORegion End Address (EL1)
LORID_EL1	LORegionID (EL1)
LORN_EL1	LORegion Number (EL1)
LORSA_EL1	LORegion Start Address (EL1)
MAIR2_EL1	Extended Memory Attribute Indirection Register (EL1)
MAIR2_EL2	Extended Memory Attribute Indirection Register (EL2)
MAIR2_EL3	Extended Memory Attribute Indirection Register (EL3)
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MAIR_EL3	Memory Attribute Indirection Register (EL3)
PIR_EL1	Permission Indirection Register 1 (EL1)
PIR_EL2	Permission Indirection Register 2 (EL2)
PIR_EL3	Permission Indirection Register 3 (EL3)
PIRE0_EL1	Permission Indirection Register 0 (EL1)
PIRE0_EL2	Permission Indirection Register 0 (EL2)
POR_EL0	Permission Overlay Register 0 (EL0)
POR_EL1	Permission Overlay Register 1 (EL1)
POR_EL2	Permission Overlay Register 2 (EL2)
POR_EL3	Permission Overlay Register 3 (EL3)
RCWMASK_EL1	Read Check Write Instruction Mask (EL1)
RCWSMASK_EL1	Software Read Check Write Instruction Mask (EL1)
S2PIR_EL2	Stage 2 Permission Indirection Register (EL2)
S2POR_EL1	Stage 2 Permission Overlay Register (EL1)
TCR2_EL1	Extended Translation Control Register (EL1)
TCR2_EL2	Extended Translation Control Register (EL2)
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TCR_EL3	Translation Control Register (EL3)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
TTBR0_EL2	Translation Table Base Register 0 (EL2)
TTBR0_EL3	Translation Table Base Register 0 (EL3)
TTBR1_EL1	Translation Table Base Register 1 (EL1)
TTBR1_EL2	Translation Table Base Register 1 (EL2)
VTCR_EL2	Virtualization Translation Control Register
VTTBR_EL2	Virtualization Translation Table Base Register

K14.3.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-12 ID registers

Register	Description, see
CCSIDR2_EL1	Current Cache Size ID Register 2
CCSIDR_EL1	Current Cache Size ID Register
CLIDR_EL1	Cache Level ID Register
CSSELR_EL1	Cache Size Selection Register
CTR_EL0	Cache Type Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DCZID_EL0	Data Cache Zero ID Register
GMID_EL1	Multiple tag transfer ID Register
ID_AA64AFR0_EL1	AArch64 Auxiliary Feature Register 0
ID_AA64AFR1_EL1	AArch64 Auxiliary Feature Register 1
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0
ID_AA64DFR1_EL1	AArch64 Debug Feature Register 1
ID_AA64DFR2_EL1	AArch64 Debug Feature Register 2
ID_AA64FPFR0_EL1	AArch64 Floating-point Feature Register 0
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2
ID_AA64ISAR3_EL1	AArch64 Instruction Set Attribute Register 3
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64MMFR3_EL1	AArch64 Memory Model Feature Register 3
ID_AA64MMFR4_EL1	AArch64 Memory Model Feature Register 4
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
ID_AA64PFR2_EL1	AArch64 Processor Feature Register 2
ID_AA64SMFR0_EL1	SME Feature ID Register 0
ID_AA64ZFR0_EL1	SVE Feature ID Register 0
ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0_EL1	AArch32 Instruction Set Attribute Register 0
ID_ISAR1_EL1	AArch32 Instruction Set Attribute Register 1
ID_ISAR2_EL1	AArch32 Instruction Set Attribute Register 2
ID_ISAR3_EL1	AArch32 Instruction Set Attribute Register 3
ID_ISAR4_EL1	AArch32 Instruction Set Attribute Register 4
ID_ISAR5_EL1	AArch32 Instruction Set Attribute Register 5
ID_ISAR6_EL1	AArch32 Instruction Set Attribute Register 6
ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0_EL1	AArch32 Processor Feature Register 0
ID_PFR1_EL1	AArch32 Processor Feature Register 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_PFR2_EL1	AArch32 Processor Feature Register 2
MIDR_EL1	Main ID Register
MPAMIDR_EL1	MPAM ID Register (EL1)
MPIDR_EL1	Multiprocessor Affinity Register
MVFR0_EL1	AArch32 Media and VFP Feature Register 0
MVFR1_EL1	AArch32 Media and VFP Feature Register 1
MVFR2_EL1	AArch32 Media and VFP Feature Register 2
REVIDR_EL1	Revision ID Register
SMIDR_EL1	Streaming Mode Identification Register
VMPIDR_EL2	Virtualization Multiprocessor ID Register
VPIDR_EL2	Virtualization Processor ID Register

K14.3.4 Performance monitors System registers

This section is an index to the System registers in the Performance Monitors registers functional group.

Table K14-13 Performance monitors registers

Register	Description, see
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Count Register
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0_EL0	Performance Monitors Common Event Identification Register 0
PMCEID1_EL0	Performance Monitors Common Event Identification Register 1
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMECR_EL1	Performance Monitors Extended Control Register (EL1)
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMMIR_EL1	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear Register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMSELR_EL0	Performance Monitors Event Counter Selection Register
PMSWINC_EL0	Performance Monitors Software Increment Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMUACR_EL1	Performance Monitors User Access Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
PMXEVCNTR_EL0	Performance Monitors Selected Event Count Register
PMXEVTYPER_EL0	Performance Monitors Selected Event Type Register
PMZR_EL0	Performance Monitors Zero with Mask

K14.3.5 Activity monitors System registers

This section is an index to the System registers in the Activity Monitors registers functional group.

Table K14-14 Activity monitors registers

Register	Description, see
AMCFGR_EL0	Activity Monitors Configuration Register
AMCG1IDR_EL0	Activity Monitors Counter Group 1 Identification Register
AMCGCR_EL0	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0_EL0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1_EL0	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0_EL0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1_EL0	Activity Monitors Count Enable Set Register 1
AMCR_EL0	Activity Monitors Control Register
AMUSERENR_EL0	Activity Monitors User Enable Register

K14.3.6 Debug System registers

This section is an index to the System registers in the Debug registers functional group.

Table K14-15 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTR_EL0	Debug Data Transfer Register, half-duplex
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
DBGPRCR_EL1	Debug Power Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DBGVCR32_EL2	Debug Vector Catch Register
MDCCINT_EL1	Monitor DCC Interrupt Enable Register
MDCCSR_EL0	Monitor DCC Status Register
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDRAR_EL1	Monitor Debug ROM Address Register
MDSCR_EL1	Monitor Debug System Control Register
OSDLR_EL1	OS Double Lock Register
OSDTRRX_EL1	OS Lock Data Transfer Register, Receive
OSDTRTX_EL1	OS Lock Data Transfer Register, Transmit
OSECCR_EL1	OS Lock Exception Catch Control Register
OSLAR_EL1	OS Lock Access Register
OSLSR_EL1	OS Lock Status Register
TRFCR_EL1	Trace Filter Control Register (EL1)
TRFCR_EL2	Trace Filter Control Register (EL2)

K14.3.7 Trace System registers

This section is an index to the System registers in the Trace unit registers functional group.

Table K14-16 Trace registers

Register	Description, see
TRCAUTHSTATUS	Trace Authentication Status Register
TRCAUXCTLR	Trace Auxiliary Control Register
TRCBBCTLR	Trace Branch Broadcast Control Register
TRCCCCTLR	Trace Cycle Count Control Register
TRCCIDCCTLR0	Trace Context Identifier Comparator Control Register 0
TRCCIDCCTLR1	Trace Context Identifier Comparator Control Register 1
TRCCLAIMCLR	Trace Claim Tag Clear Register
TRCCLAIMSET	Trace Claim Tag Set Register
TRCCONFIGR	Trace Configuration Register
TRCDEVARCH	Trace Device Architecture Register
TRCDEVID	Trace Device Configuration Register
TRCEVENTCTL0R	Trace Event Control 0 Register
TRCEVENTCTL1R	Trace Event Control 1 Register
TRCIDR0	Trace ID Register 0
TRCIDR1	Trace ID Register 1
TRCIDR10	Trace ID Register 10

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCIDR11	Trace ID Register 11
TRCIDR12	Trace ID Register 12
TRCIDR13	Trace ID Register 13
TRCIDR2	Trace ID Register 2
TRCIDR3	Trace ID Register 3
TRCIDR4	Trace ID Register 4
TRCIDR5	Trace ID Register 5
TRCIDR6	Trace ID Register 6
TRCIDR7	Trace ID Register 7
TRCIDR8	Trace ID Register 8
TRCIDR9	Trace ID Register 9
TRCIMSPEC0	Trace IMP DEF Register 0
TRCITECR_EL1	Instrumentation Trace Control Register (EL1)
TRCITECR_EL2	Instrumentation Trace Control Register (EL2)
TRCITEEDCR	Instrumentation Trace Extension External Debug Control Register
TRCOSLSR	Trace OS Lock Status Register
TRCPRGCTLR	Trace Programming Control Register
TRCQCTLR	Trace Q Element Control Register
TRCRSR	Trace Resources Status Register
TRCSEQRSTEVR	Trace Sequencer Reset Control Register
TRCSEQSTR	Trace Sequencer State Register
TRCSTALLCTLR	Trace Stall Control Register
TRCSTATR	Trace Status Register
TRCSYNCPR	Trace Synchronization Period Register
TRCTRACEIDR	Trace ID Register
TRCTSCTLR	Trace Timestamp Control Register
TRCVICTLR	Trace ViewInst Main Control Register
TRCVIIECTLR	Trace ViewInst Include-Exclude Control Register
TRCVIPCSSCTLR	Trace ViewInst Start-Stop PE Comparator Control Register
TRCVISSCTLR	Trace ViewInst Start-Stop Control Register
TRCVMIDCCTLR0	Trace Virtual Context Identifier Comparator Control Register 0
TRCVMIDCCTLR1	Trace Virtual Context Identifier Comparator Control Register 1

K14.3.8 Branch Record Buffer registers

This section is an index to the registers in the Branch Record Buffer Extension registers functional group.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-17 Branch Record Buffer registers

Register	Description, see
BRBCR_EL1	Branch Record Buffer Control Register (EL1)
BRBCR_EL2	Branch Record Buffer Control Register (EL2)
BRBFCR_EL1	Branch Record Buffer Function Control Register
BRBIDR0_EL1	Branch Record Buffer ID0 Register
BRBINFINJ_EL1	Branch Record Buffer Information Injection Register
BRBSRCINJ_EL1	Branch Record Buffer Source Address Injection Register
BRBTGTINJ_EL1	Branch Record Buffer Target Address Injection Register
BRBTS_EL1	Branch Record Buffer Timestamp Register

K14.3.9 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-18 RAS registers

Register	Description, see
DISR_EL1	Deferred Interrupt Status Register
ERRIDR_EL1	Error Record ID Register
ERRSELR_EL1	Error Record Select Register
ERXADDR_EL1	Selected Error Record Address Register
ERXCTLR_EL1	Selected Error Record Control Register
ERXFR_EL1	Selected Error Record Feature Register
ERXGSR_EL1	Selected Error Record Group Status Register
ERXMISC0_EL1	Selected Error Record Miscellaneous Register 0
ERXMISC1_EL1	Selected Error Record Miscellaneous Register 1
ERXMISC2_EL1	Selected Error Record Miscellaneous Register 2
ERXMISC3_EL1	Selected Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	Selected Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	Selected Pseudo-fault Generation Control Register
ERXPFGF_EL1	Selected Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	Selected Error Record Primary Status Register
MFAR_EL3	Physical Fault Address Register (EL3)
VDISR_EL2	Virtual Deferred Interrupt Status Register (EL2)
VDISR_EL3	Virtual Deferred Interrupt Status Register (EL3)
VSESR_EL2	Virtual SError Exception Syndrome Register (EL2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
VSESR_EL3	Virtual SError Exception Syndrome Register (EL3)

K14.3.10 Root Security state registers

This section is an index to the registers in the Root Security state registers functional group.

Table K14-19 Root Security state registers

Register	Description, see
APAS	Associate PA space
GPCBW_EL3	Granule Protection Check Bypass Window Register (EL3)
GPCCR_EL3	Granule Protection Check Control Register (EL3)
GPTBR_EL3	Granule Protection Table Base Register

K14.3.11 Memory Partitioning and Monitoring registers

This section is an index to the registers in the Memory Partitioning and Monitoring registers functional group.

Table K14-20 Memory Partitioning and Monitoring registers

Register	Description, see
MPAM0_EL1	MPAM0 Register (EL1)
MPAM1_EL1	MPAM1 Register (EL1)
MPAM2_EL2	MPAM2 Register (EL2)
MPAM3_EL3	MPAM3 Register (EL3)
MPAMBW0_EL1	MPAM PE-side Maximum Bandwidth Control Register (EL0)
MPAMBW1_EL1	MPAM PE-side Maximum Bandwidth Control Register (EL1)
MPAMBW2_EL2	MPAM PE-side Maximum Bandwidth Control Register (EL2)
MPAMBW3_EL3	MPAM PE-side Maximum Bandwidth Control Register (EL3)
MPAMBWCAP_EL2	MPAM PE-side Maximum Bandwidth Limit Virtualization Register
MPAMBWIDR_EL1	MPAM PE-side Bandwidth Controls ID Register
MPAMBWSM_EL1	MPAM Streaming Mode Bandwidth Control Register (EL1)
MPAMHCR_EL2	MPAM Hypervisor Control Register (EL2)
MPAMSM_EL1	MPAM Streaming Mode Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MPAMVPM0_EL2	MPAM Virtual PARTID Mapping Register 0
MPAMVPM1_EL2	MPAM Virtual PARTID Mapping Register 1
MPAMVPM2_EL2	MPAM Virtual PARTID Mapping Register 2
MPAMVPM3_EL2	MPAM Virtual PARTID Mapping Register 3
MPAMVPM4_EL2	MPAM Virtual PARTID Mapping Register 4
MPAMVPM5_EL2	MPAM Virtual PARTID Mapping Register 5
MPAMVPM6_EL2	MPAM Virtual PARTID Mapping Register 6
MPAMVPM7_EL2	MPAM Virtual PARTID Mapping Register 7
MPAMVPMV_EL2	MPAM Virtual Partition Mapping Valid Register

K14.3.12 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K14-21 Generic timer registers

Register	Description, see
CNTFRQ_EL0	Counter-timer Frequency Register
CNTHCTL_EL2	Counter-timer Hypervisor Control Register
CNTHP_CTL_EL2	Counter-timer Hypervisor Physical Timer Control Register
CNTHP_CVAL_EL2	Counter-timer Physical Timer CompareValue Register (EL2)
CNTHP_TVAL_EL2	Counter-timer Physical Timer TimerValue Register (EL2)
CNTHPS_CTL_EL2	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL_EL2	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL_EL2	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL_EL2	Counter-timer Virtual Timer Control Register (EL2)
CNTHV_CVAL_EL2	Counter-timer Virtual Timer CompareValue Register (EL2)
CNTHV_TVAL_EL2	Counter-timer Virtual Timer TimerValue Register (EL2)
CNTHVS_CTL_EL2	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL_EL2	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL_EL2	Counter-timer Secure Virtual Timer TimerValue Register (EL2)
CNTKCTL_EL1	Counter-timer Kernel Control Register
CNTP_CTL_EL0	Counter-timer Physical Timer Control Register
CNTP_CVAL_EL0	Counter-timer Physical Timer CompareValue Register
CNTP_TVAL_EL0	Counter-timer Physical Timer TimerValue Register
CNTPCT_EL0	Counter-timer Physical Count Register
CNTPCTSS_EL0	Counter-timer Self-Synchronized Physical Count Register
CNTPOFF_EL2	Counter-timer Physical Offset Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTPS_CTL_EL1	Counter-timer Physical Secure Timer Control Register
CNTPS_CVAL_EL1	Counter-timer Physical Secure Timer CompareValue Register
CNTPS_TVAL_EL1	Counter-timer Physical Secure Timer TimerValue Register
CNTV_CTL_EL0	Counter-timer Virtual Timer Control Register
CNTV_CVAL_EL0	Counter-timer Virtual Timer CompareValue Register
CNTV_TVAL_EL0	Counter-timer Virtual Timer TimerValue Register
CNTVCT_EL0	Counter-timer Virtual Count Register
CNTVCTSS_EL0	Counter-timer Self-Synchronized Virtual Count Register
CNTVOFF_EL2	Counter-timer Virtual Offset Register

K14.3.13 Cache maintenance System instructions

This section is an index to the System instructions in the Cache maintenance instructions functional group.

Table K14-22 Cache maintenance system instructions

System instruction	Description, see
DC CGDSW	Clean of Data and Allocation Tags by Set-Way
DC CGDVAC	Clean of Data and Allocation Tags by VA to PoC
DC CGDVADP	Clean of Data and Allocation Tags by VA to PoDP
DC CGDVAOC	Clean of Data and Allocation Tags by VA to Outer Cache level
DC CGDVAP	Clean of Data and Allocation Tags by VA to PoP
DC CGSW	Clean of Allocation Tags by Set-Way
DC CGVAC	Clean of Allocation Tags by VA to PoC
DC CGVADP	Clean of Allocation Tags by VA to PoDP
DC CGVAP	Clean of Allocation Tags by VA to PoP
DC CIGDPAE	Clean and invalidate of data and allocation tags by PA to PoE
DC CIGDPAPA	Clean and Invalidate of Data and Allocation Tags by PA to PoPA
DC CIGDSW	Clean and Invalidate of Data and Allocation Tags by Set-Way
DC CIGDVAC	Clean and Invalidate of Data and Allocation Tags by VA to PoC
DC CIGDVAOC	Clean and Invalidate of Data and Allocation Tags by VA to Outer Cache level
DC CIGDVAPS	Clean and Invalidate of Data and Allocation Tags by VA to PoPS
DC CIGSW	Clean and Invalidate of Allocation Tags by Set-Way
DC CIGVAC	Clean and Invalidate of Allocation Tags by VA to PoC
DC CIPAE	Data or unified Cache line Clean and Invalidate by PA to PoE
DC CIPAPA	Data or unified Cache line Clean and Invalidate by PA to PoPA
DC CISW	Data or unified Cache line Clean and Invalidate by Set-Way
DC CIVAC	Data or unified Cache line Clean and Invalidate by VA to PoC

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
DC CIVAOC	Data or unified Cache line Clean and Invalidate by VA to Outer Cache level
DC CIVAPS	Clean and Invalidate of Data by VA to PoPS
DC CSW	Data or unified Cache line Clean by Set-Way
DC CVAC	Data or unified Cache line Clean by VA to PoC
DC CVADP	Data or unified Cache line Clean by VA to PoDP
DC CVAOC	Data or unified Cache line Clean by VA to Outer Cache level
DC CVAP	Data or unified Cache line Clean by VA to PoP
DC CVAU	Data or unified Cache line Clean by VA to PoU
DC GVA	Data Cache set Allocation Tag by VA
DC GZVA	Data Cache set Allocation Tags and Zero by VA
DC IGDSW	Invalidate of Data and Allocation Tags by Set-Way
DC IGDVAC	Invalidate of Data and Allocation Tags by VA to PoC
DC IGSW	Invalidate of Allocation Tags by Set-Way
DC IGVAC	Invalidate of Allocation Tags by VA to PoC
DC ISW	Data or unified Cache line Invalidate by Set-Way
DC IVAC	Data or unified Cache line Invalidate by VA to PoC
DC ZVA	Data Cache Zero by VA
IC IALLU	Instruction Cache Invalidate All to PoU
IC IALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
IC IVAU	Instruction Cache line Invalidate by VA to PoU

K14.3.14 Address translation System instructions

This section is an index to the System instructions in the Address translation instructions functional group.

Table K14-23 Address translation system instructions

System instruction	Description, see
AT S12E0R	Address Translate Stages 1 and 2 EL0 Read
AT S12E0W	Address Translate Stages 1 and 2 EL0 Write
AT S12E1R	Address Translate Stages 1 and 2 EL1 Read
AT S12E1W	Address Translate Stages 1 and 2 EL1 Write
AT S1E0R	Address Translate Stage 1 EL0 Read
AT S1E0W	Address Translate Stage 1 EL0 Write
AT S1E1A	Address Translate Stage 1 EL1 Without Permission checks
AT S1E1R	Address Translate Stage 1 EL1 Read
AT S1E1RP	Address Translate Stage 1 EL1 Read PAN
AT S1E1W	Address Translate Stage 1 EL1 Write

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
AT S1E1WP	Address Translate Stage 1 EL1 Write PAN
AT S1E2A	Address Translate Stage 1 EL2 Without Permission checks
AT S1E2R	Address Translate Stage 1 EL2 Read
AT S1E2W	Address Translate Stage 1 EL2 Write
AT S1E3A	Address Translate Stage 1 EL3 Without Permission checks
AT S1E3R	Address Translate Stage 1 EL3 Read
AT S1E3W	Address Translate Stage 1 EL3 Write

K14.3.15 TLB maintenance System instructions

This section is an index to the System instructions in the TLB maintenance instructions functional group.

Table K14-24 TLB maintenance system instructions

System instruction	Description, see
TLBI ALLE1, TLBI ALLE1NXS	TLB Invalidate All, EL1
TLBI ALLE1IS, TLBI ALLE1ISNXS	TLB Invalidate All, EL1, Inner Shareable
TLBI ALLE1OS, TLBI ALLE1OSNXS	TLB Invalidate All, EL1, Outer Shareable
TLBI ALLE2, TLBI ALLE2NXS	TLB Invalidate All, EL2
TLBI ALLE2IS, TLBI ALLE2ISNXS	TLB Invalidate All, EL2, Inner Shareable
TLBI ALLE2OS, TLBI ALLE2OSNXS	TLB Invalidate All, EL2, Outer Shareable
TLBI ALLE3, TLBI ALLE3NXS	TLB Invalidate All, EL3
TLBI ALLE3IS, TLBI ALLE3ISNXS	TLB Invalidate All, EL3, Inner Shareable
TLBI ALLE3OS, TLBI ALLE3OSNXS	TLB Invalidate All, EL3, Outer Shareable
TLBI ASIDE1, TLBI ASIDE1NXS	TLB Invalidate by ASID, EL1
TLBI ASIDE1IS, TLBI ASIDE1ISNXS	TLB Invalidate by ASID, EL1, Inner Shareable
TLBI ASIDE1OS, TLBI ASIDE1OSNXS	TLB Invalidate by ASID, EL1, Outer Shareable
TLBI IPAS2E1, TLBI IPAS2E1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI IPAS2LE1, TLBI IPAS2LE1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI PAALL	TLB Invalidate GPT Information by PA, All Entries, Local

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI PAALLOS	TLB Invalidate GPT Information by PA, All Entries, Outer Shareable
TLBI RIPAS2E1, TLBI RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RPALOS	TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable
TLBI RPAOS	TLB Range Invalidate GPT Information by PA, Outer Shareable
TLBI RVAAE1, TLBI RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBI RVAAE1IS, TLBI RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI RVAAE1OS, TLBI RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI RVAALE1, TLBI RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBI RVAALE1IS, TLBI RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI RVAALE1OS, TLBI RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI RVAE1, TLBI RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBI RVAE1IS, TLBI RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBI RVAE1OS, TLBI RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBI RVAE2, TLBI RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBI RVAE2IS, TLBI RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBI RVAE2OS, TLBI RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBI RVAE3, TLBI RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBI RVAE3IS, TLBI RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBI RVAE3OS, TLBI RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBI RVALE1, TLBI RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBI RVALE1IS, TLBI RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBI RVALE1OS, TLBI RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBI RVALE2, TLBI RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBI RVALE2IS, TLBI RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI RVALE2OS, TLBI RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBI RVALE3, TLBI RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBI RVALE3IS, TLBI RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBI RVALE3OS, TLBI RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VAAE1, TLBI VAAE1NXS	TLB Invalidate by VA, All ASID, EL1
TLBI VAAE1IS, TLBI VAAE1ISNXS	TLB Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI VAAE1OS, TLBI VAAE1OSNXS	TLB Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI VAALE1, TLBI VAALE1NXS	TLB Invalidate by VA, All ASID, Last level, EL1
TLBI VAALE1IS, TLBI VAALE1ISNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI VAALE1OS, TLBI VAALE1OSNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI VAE1, TLBI VAE1NXS	TLB Invalidate by VA, EL1
TLBI VAE1IS, TLBI VAE1ISNXS	TLB Invalidate by VA, EL1, Inner Shareable
TLBI VAE1OS, TLBI VAE1OSNXS	TLB Invalidate by VA, EL1, Outer Shareable
TLBI VAE2, TLBI VAE2NXS	TLB Invalidate by VA, EL2
TLBI VAE2IS, TLBI VAE2ISNXS	TLB Invalidate by VA, EL2, Inner Shareable
TLBI VAE2OS, TLBI VAE2OSNXS	TLB Invalidate by VA, EL2, Outer Shareable
TLBI VAE3, TLBI VAE3NXS	TLB Invalidate by VA, EL3
TLBI VAE3IS, TLBI VAE3ISNXS	TLB Invalidate by VA, EL3, Inner Shareable
TLBI VAE3OS, TLBI VAE3OSNXS	TLB Invalidate by VA, EL3, Outer Shareable
TLBI VALE1, TLBI VALE1NXS	TLB Invalidate by VA, Last level, EL1
TLBI VALE1IS, TLBI VALE1ISNXS	TLB Invalidate by VA, Last level, EL1, Inner Shareable
TLBI VALE1OS, TLBI VALE1OSNXS	TLB Invalidate by VA, Last level, EL1, Outer Shareable
TLBI VALE2, TLBI VALE2NXS	TLB Invalidate by VA, Last level, EL2
TLBI VALE2IS, TLBI VALE2ISNXS	TLB Invalidate by VA, Last level, EL2, Inner Shareable
TLBI VALE2OS, TLBI VALE2OSNXS	TLB Invalidate by VA, Last level, EL2, Outer Shareable
TLBI VALE3, TLBI VALE3NXS	TLB Invalidate by VA, Last level, EL3
TLBI VALE3IS, TLBI VALE3ISNXS	TLB Invalidate by VA, Last level, EL3, Inner Shareable
TLBI VALE3OS, TLBI VALE3OSNXS	TLB Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VMALLE1, TLBI VMALLE1NXS	TLB Invalidate by VMID, All at stage 1, EL1
TLBI VMALLE1IS, TLBI VMALLE1ISNXS	TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable
TLBI VMALLE1OS, TLBI VMALLE1OSNXS	TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable
TLBI VMALLS12E1, TLBI VMALLS12E1NXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1
TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable
TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0
TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0, Inner Shareable
TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS	TLB Invalidate stage 2 write permission by VMID, EL1&0, Outer Shareable
TLBIP IPAS2E1, TLBIP IPAS2E1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1
TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RVAAE1, TLBIP RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBIP RVAALE1, TLBIP RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP RVAE1, TLBIP RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBIP RVAE1IS, TLBIP RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBIP RVAE1OS, TLBIP RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBIP RVAE2, TLBIP RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBIP RVAE2IS, TLBIP RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBIP RVAE2OS, TLBIP RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBIP RVAE3, TLBIP RVAE3NXS	TLB Range Invalidate by VA, EL3

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBIP RVAE3IS, TLBIP RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBIP RVAE3OS, TLBIP RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBIP RVALE1, TLBIP RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBIP RVALE1IS, TLBIP RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBIP RVALE1OS, TLBIP RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBIP RVALE2, TLBIP RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBIP RVALE2IS, TLBIP RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBIP RVALE2OS, TLBIP RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBIP RVALE3, TLBIP RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBIP RVALE3IS, TLBIP RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBIP RVALE3OS, TLBIP RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBIP VAAE1, TLBIP VAAE1NXS	TLB Invalidate Pair by VA, All ASID, EL1
TLBIP VAAE1IS, TLBIP VAAE1ISNXS	TLB Invalidate Pair by VA, All ASID, EL1, Inner Shareable
TLBIP VAAE1OS, TLBIP VAAE1OSNXS	TLB Invalidate Pair by VA, All ASID, EL1, Outer Shareable
TLBIP VAALE1, TLBIP VAALE1NXS	TLB Invalidate Pair by VA, All ASID, Last level, EL1
TLBIP VAALE1IS, TLBIP VAALE1ISNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Inner Shareable
TLBIP VAALE1OS, TLBIP VAALE1OSNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP VAE1, TLBIP VAE1NXS	TLB Invalidate Pair by VA, EL1
TLBIP VAE1IS, TLBIP VAE1ISNXS	TLB Invalidate Pair by VA, EL1, Inner Shareable
TLBIP VAE1OS, TLBIP VAE1OSNXS	TLB Invalidate Pair by VA, EL1, Outer Shareable
TLBIP VAE2, TLBIP VAE2NXS	TLB Invalidate Pair by VA, EL2
TLBIP VAE2IS, TLBIP VAE2ISNXS	TLB Invalidate Pair by VA, EL2, Inner Shareable
TLBIP VAE2OS, TLBIP VAE2OSNXS	TLB Invalidate Pair by VA, EL2, Outer Shareable
TLBIP VAE3, TLBIP VAE3NXS	TLB Invalidate Pair by VA, EL3
TLBIP VAE3IS, TLBIP VAE3ISNXS	TLB Invalidate Pair by VA, EL3, Inner Shareable
TLBIP VAE3OS, TLBIP VAE3OSNXS	TLB Invalidate Pair by VA, EL3, Outer Shareable
TLBIP VALE1, TLBIP VALE1NXS	TLB Invalidate Pair by VA, Last level, EL1
TLBIP VALE1IS, TLBIP VALE1ISNXS	TLB Invalidate Pair by VA, Last level, EL1, Inner Shareable
TLBIP VALE1OS, TLBIP VALE1OSNXS	TLB Invalidate Pair by VA, Last level, EL1, Outer Shareable
TLBIP VALE2, TLBIP VALE2NXS	TLB Invalidate Pair by VA, Last level, EL2
TLBIP VALE2IS, TLBIP VALE2ISNXS	TLB Invalidate Pair by VA, Last level, EL2, Inner Shareable

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBIP VALE2OS, TLBIP VALE2OSNXS	TLB Invalidate Pair by VA, Last level, EL2, Outer Shareable
TLBIP VALE3, TLBIP VALE3NXS	TLB Invalidate Pair by VA, Last level, EL3
TLBIP VALE3IS, TLBIP VALE3ISNXS	TLB Invalidate Pair by VA, Last level, EL3, Inner Shareable
TLBIP VALE3OS, TLBIP VALE3OSNXS	TLB Invalidate Pair by VA, Last level, EL3, Outer Shareable

K14.3.16 Prediction restriction System instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table K14-25 Cache maintenance system instructions

System instruction	Description, see
CFP RCTX	Control Flow Prediction Restriction by Context
COSP RCTX	Clear Other Speculative Prediction Restriction by Context
CPP RCTX	Cache Prefetch Prediction Restriction by Context
DVP RCTX	Data Value Prediction Restriction by Context

K14.3.17 Base System registers

This section is an index to the System registers in the Base System registers functional group.

Table K14-26 Base system registers

Register	Description, see
ACTLRMASK_EL1	Auxiliary Control Masking Register (EL1)
ACTLRMASK_EL2	Auxiliary Control Masking Register (EL2)
CPACR_EL1	Architectural Feature Access Control Register
CPACRMASK_EL1	Architectural Feature Access Control Masking Register
CPTRMASK_EL2	Architectural Feature Trap Masking Register
HFGITR2_EL2	Hypervisor Fine-Grained Instruction Trap Register 2
SCTLR2_EL1	System Control Register (EL1)
SCTLR2_EL2	System Control Register (EL2)
SCTLR2_EL3	System Control Register (EL3)
SCTLR2MASK_EL1	Extended System Control Masking Register (EL1)
SCTLR2MASK_EL2	Extended System Control Masking Register (EL2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)
SCTLR_EL3	System Control Register (EL3)
SCTLRMASK_EL1	System Control Masking Register (EL1)
SCTLRMASK_EL2	System Control Masking Register (EL2)
SMCR_EL1	SME Control Register (EL1)
SMCR_EL2	SME Control Register (EL2)
SMCR_EL3	SME Control Register (EL3)
SMPRI_EL1	Streaming Mode Priority Register
SMPRIMAP_EL2	Streaming Mode Priority Mapping Register
TCR2MASK_EL1	Extended Translation Control Masking Register (EL1)
TCR2MASK_EL2	Extended Translation Control Masking Register (EL2)
TCRMASK_EL1	Translation Control Masking Register (EL1)
TCRMASK_EL2	Translation Control Masking Register (EL2)
ZCR_EL1	SVE Control Register (EL1)
ZCR_EL2	SVE Control Register (EL2)
ZCR_EL3	SVE Control Register (EL3)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.4 Alphabetical index of AArch32 registers and System instructions

This section is an index of AArch32 registers and System instructions in alphabetical order.

Table K14-27 Alphabetical index of AArch32 Registers

Register	Description, see
ACTLR	Auxiliary Control Register
ACTLR2	Auxiliary Control Register 2
ADFSR	Auxiliary Data Fault Status Register
AIDR	Auxiliary ID Register
AIFSR	Auxiliary Instruction Fault Status Register
AMAIR0	Auxiliary Memory Attribute Indirection Register 0
AMAIR1	Auxiliary Memory Attribute Indirection Register 1
AMCFGR	Activity Monitors Configuration Register
AMCGCR	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1
AMCR	Activity Monitors Control Register
AMUSERENR	Activity Monitors User Enable Register
APSR	Application Program Status Register
ATS12NSOPR	Address Translate Stages 1 and 2 Non-secure Only PL1 Read
ATS12NSOPW	Address Translate Stages 1 and 2 Non-secure Only PL1 Write
ATS12NSOUR	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read
ATS12NSOUW	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write
ATS1CPR	Address Translate Stage 1 Current state PL1 Read
ATS1CPRP	Address Translate Stage 1 Current state PL1 Read PAN
ATS1CPW	Address Translate Stage 1 Current state PL1 Write
ATS1CPWP	Address Translate Stage 1 Current state PL1 Write PAN
ATS1CUR	Address Translate Stage 1 Current state Unprivileged Read
ATS1CUW	Address Translate Stage 1 Current state Unprivileged Write
ATS1HR	Address Translate Stage 1 Hyp mode Read
ATS1HW	Address Translate Stage 1 Hyp mode Write
BPIALL	Branch Predictor Invalidate All
BPIALLIS	Branch Predictor Invalidate All, Inner Shareable
BPIMVA	Branch Predictor Invalidate by VA
CCSIDR	Current Cache Size ID Register
CCSIDR2	Current Cache Size ID Register 2
CFPRCTX	Control Flow Prediction Restriction by Context

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CLIDR	Cache Level ID Register
CNTFRQ	Counter-timer Frequency register
CNTHCTL	Counter-timer Hyp Control register
CNTHP_CTL	Counter-timer Hyp Physical Timer Control register
CNTHP_CVAL	Counter-timer Hyp Physical CompareValue register
CNTHP_TVAL	Counter-timer Hyp Physical Timer TimerValue register
CNTHPS_CTL	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL	Counter-timer Virtual Timer Control register (EL2)
CNTHV_CVAL	Counter-timer Virtual Timer CompareValue register (EL2)
CNTHV_TVAL	Counter-timer Virtual Timer TimerValue register (EL2)
CNTHVS_CTL	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL	Counter-timer Secure Virtual Timer TimerValue Register (EL2)
CNTKCTL	Counter-timer Kernel Control register
CNTP_CTL	Counter-timer Physical Timer Control register
CNTP_CVAL	Counter-timer Physical Timer CompareValue register
CNTP_TVAL	Counter-timer Physical Timer TimerValue register
CNTPCT	Counter-timer Physical Count register
CNTPCTSS	Counter-timer Self-Synchronized Physical Count register
CNTV_CTL	Counter-timer Virtual Timer Control register
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register
CNTVCT	Counter-timer Virtual Count register
CNTVCTSS	Counter-timer Self-Synchronized Virtual Count register
CNTVOFF	Counter-timer Virtual Offset register
CONTEXTIDR	Context ID Register
COSPRCTX	Clear Other Speculative Prediction Restriction by Context
CP15DMB	Data Memory Barrier System instruction
CP15DSB	Data Synchronization Barrier System instruction
CP15ISB	Instruction Synchronization Barrier System instruction
CPACR	Architectural Feature Access Control Register
CPPRCTX	Cache Prefetch Prediction Restriction by Context
CPSR	Current Program Status Register
CSSELR	Cache Size Selection Register
CTR	Cache Type Register
DACR	Domain Access Control Register
DBGAUTHSTATUS	Debug Authentication Status register
DBGCLAIMCLR	Debug CLAIM Tag Clear register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DBGCLAIMSET	Debug CLAIM Tag Set register
DBGDCCINT	DCC Interrupt Enable Register
DBGDEVID	Debug Device ID register 0
DBGDEVID1	Debug Device ID register 1
DBGDEVID2	Debug Device ID register 2
DBGDIDR	Debug ID Register
DBGDRAR	Debug ROM Address Register
DBGDSAR	Debug Self Address Register
DBGDSCRext	Debug Status and Control Register, External View
DBGDSCRint	Debug Status and Control Register, Internal View
DBGDTRRXext	Debug OS Lock Data Transfer Register, Receive, External View
DBGDTRRXint	Debug Data Transfer Register, Receive
DBGDTRTXext	Debug OS Lock Data Transfer Register, Transmit
DBGDTRTXint	Debug Data Transfer Register, Transmit
DBGOSDLR	Debug OS Double Lock Register
DBGOSECCR	Debug OS Lock Exception Catch Control Register
DBGOSLAR	Debug OS Lock Access Register
DBGOSLSR	Debug OS Lock Status Register
DBGPRCR	Debug Power Control Register
DBGVCR	Debug Vector Catch Register
DBGWFAR	Debug Watchpoint Fault Address Register
DCCIMVAC	Data Cache line Clean and Invalidate by VA to PoC
DCCISW	Data Cache line Clean and Invalidate by Set-Way
DCCMVAC	Data Cache line Clean by VA to PoC
DCCMVAU	Data Cache line Clean by VA to PoU
DCCSW	Data Cache line Clean by Set-Way
DCIMVAC	Data Cache line Invalidate by VA to PoC
DCISW	Data Cache line Invalidate by Set-Way
DFAR	Data Fault Address Register
DFSR	Data Fault Status Register
DISR	Deferred Interrupt Status Register
DLR	Debug Link Register
DSPSR	Debug Saved Program Status Register
DSPSR2	Debug Saved Process State Register 2
DTLBIALL	Data TLB Invalidate All
DTLBIASID	Data TLB Invalidate by ASID match
DTLBIMVA	Data TLB Invalidate by VA
DVPRCTX	Data Value Prediction Restriction by Context
ELR_hyp	Exception Link Register (Hyp mode)
ERRIDR	Error Record ID Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ERRSELR	Error Record Select Register
ERXADDR	Selected Error Record Address Register
ERXADDR2	Selected Error Record Address Register 2
ERXCTLR	Selected Error Record Control Register
ERXCTLR2	Selected Error Record Control Register 2
ERXFR	Selected Error Record Feature Register
ERXFR2	Selected Error Record Feature Register 2
ERXMISC0	Selected Error Record Miscellaneous Register 0
ERXMISC1	Selected Error Record Miscellaneous Register 1
ERXMISC2	Selected Error Record Miscellaneous Register 2
ERXMISC3	Selected Error Record Miscellaneous Register 3
ERXMISC4	Selected Error Record Miscellaneous Register 4
ERXMISC5	Selected Error Record Miscellaneous Register 5
ERXMISC6	Selected Error Record Miscellaneous Register 6
ERXMISC7	Selected Error Record Miscellaneous Register 7
ERXSTATUS	Selected Error Record Primary Status Register
FCSEIDR	FCSE Process ID register
FPEXC	Floating-Point Exception Control register
FPSCR	Floating-Point Status and Control Register
FPSID	Floating-Point System ID register
HACR	Hyp Auxiliary Configuration Register
HACTLR	Hyp Auxiliary Control Register
HACTLR2	Hyp Auxiliary Control Register 2
HADFSR	Hyp Auxiliary Data Fault Status Register
HAIFSR	Hyp Auxiliary Instruction Fault Status Register
HAMAIR0	Hyp Auxiliary Memory Attribute Indirection Register 0
HAMAIR1	Hyp Auxiliary Memory Attribute Indirection Register 1
HCPTR	Hyp Architectural Feature Trap Register
HCR	Hyp Configuration Register
HCR2	Hyp Configuration Register 2
HDCR	Hyp Debug Control Register
HDFAR	Hyp Data Fault Address Register
HIFAR	Hyp Instruction Fault Address Register
HMAIR0	Hyp Memory Attribute Indirection Register 0
HMAIR1	Hyp Memory Attribute Indirection Register 1
HPFAR	Hyp IPA Fault Address Register
HRMR	Hyp Reset Management Register
HSCTLR	Hyp System Control Register
HSR	Hyp Syndrome Register
HSTR	Hyp System Trap Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
HTCR	Hyp Translation Control Register
HTPIDR	Hyp Software Thread ID Register
HTRFCR	Hyp Trace Filter Control Register
HTTBR	Hyp Translation Table Base Register
HVBAR	Hyp Vector Base Address Register
ICC_ASGI1R	Interrupt Controller Alias Software Generated Interrupt Group 1 Register
ICC_BPR0	Interrupt Controller Binary Point Register 0
ICC_BPR1	Interrupt Controller Binary Point Register 1
ICC_CTLR	Interrupt Controller Control Register
ICC_DIR	Interrupt Controller Deactivate Interrupt Register
ICC_EOIR0	Interrupt Controller End Of Interrupt Register 0
ICC_EOIR1	Interrupt Controller End Of Interrupt Register 1
ICC_HPPIR0	Interrupt Controller Highest Priority Pending Interrupt Register 0
ICC_HPPIR1	Interrupt Controller Highest Priority Pending Interrupt Register 1
ICC_HSRE	Interrupt Controller Hyp System Register Enable register
ICC_IAR0	Interrupt Controller Interrupt Acknowledge Register 0
ICC_IAR1	Interrupt Controller Interrupt Acknowledge Register 1
ICC_IGRPEN0	Interrupt Controller Interrupt Group 0 Enable register
ICC_IGRPEN1	Interrupt Controller Interrupt Group 1 Enable register
ICC_MCTLR	Interrupt Controller Monitor Control Register
ICC_MGRPEN1	Interrupt Controller Monitor Interrupt Group 1 Enable register
ICC_MSRE	Interrupt Controller Monitor System Register Enable register
ICC_PMR	Interrupt Controller Interrupt Priority Mask Register
ICC_RPR	Interrupt Controller Running Priority Register
ICC_SGI0R	Interrupt Controller Software Generated Interrupt Group 0 Register
ICC_SGI1R	Interrupt Controller Software Generated Interrupt Group 1 Register
ICC_SRE	Interrupt Controller System Register Enable register
ICH_EISR	Interrupt Controller End of Interrupt Status Register
ICH_ELRSR	Interrupt Controller Empty List Register Status Register
ICH_HCR	Interrupt Controller Hyp Control Register
ICH_MISR	Interrupt Controller Maintenance Interrupt State Register
ICH_VMCR	Interrupt Controller Virtual Machine Control Register
ICH_VTR	Interrupt Controller VGIC Type Register
ICIALLU	Instruction Cache Invalidate All to PoU
ICIALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
ICIMVAU	Instruction Cache line Invalidate by VA to PoU
ICV_BPR0	Interrupt Controller Virtual Binary Point Register 0
ICV_BPR1	Interrupt Controller Virtual Binary Point Register 1
ICV_CTLR	Interrupt Controller Virtual Control Register
ICV_DIR	Interrupt Controller Deactivate Virtual Interrupt Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ICV_EOIR0	Interrupt Controller Virtual End Of Interrupt Register 0
ICV_EOIR1	Interrupt Controller Virtual End Of Interrupt Register 1
ICV_HPPIR0	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0
ICV_HPPIR1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1
ICV_IAR0	Interrupt Controller Virtual Interrupt Acknowledge Register 0
ICV_IAR1	Interrupt Controller Virtual Interrupt Acknowledge Register 1
ICV_IGRPEN0	Interrupt Controller Virtual Interrupt Group 0 Enable register
ICV_IGRPEN1	Interrupt Controller Virtual Interrupt Group 1 Enable register
ICV_PMR	Interrupt Controller Virtual Interrupt Priority Mask Register
ICV_RPR	Interrupt Controller Virtual Running Priority Register
ID_AFR0	Auxiliary Feature Register 0
ID_DFR0	Debug Feature Register 0
ID_DFR1	Debug Feature Register 1
ID_ISAR0	Instruction Set Attribute Register 0
ID_ISAR1	Instruction Set Attribute Register 1
ID_ISAR2	Instruction Set Attribute Register 2
ID_ISAR3	Instruction Set Attribute Register 3
ID_ISAR4	Instruction Set Attribute Register 4
ID_ISAR5	Instruction Set Attribute Register 5
ID_ISAR6	Instruction Set Attribute Register 6
ID_MMFR0	Memory Model Feature Register 0
ID_MMFR1	Memory Model Feature Register 1
ID_MMFR2	Memory Model Feature Register 2
ID_MMFR3	Memory Model Feature Register 3
ID_MMFR4	Memory Model Feature Register 4
ID_MMFR5	Memory Model Feature Register 5
ID_PFR0	Processor Feature Register 0
ID_PFR1	Processor Feature Register 1
ID_PFR2	Processor Feature Register 2
IFAR	Instruction Fault Address Register
IFSR	Instruction Fault Status Register
ISR	Interrupt Status Register
ITLBIALL	Instruction TLB Invalidate All
ITLBIASID	Instruction TLB Invalidate by ASID match
ITLBIMVA	Instruction TLB Invalidate by VA
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register
MAIR0	Memory Attribute Indirection Register 0
MAIR1	Memory Attribute Indirection Register 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MIDR	Main ID Register
MPIDR	Multiprocessor Affinity Register
MVBAR	Monitor Vector Base Address Register
MVFR0	Media and VFP Feature Register 0
MVFR1	Media and VFP Feature Register 1
MVFR2	Media and VFP Feature Register 2
NMRR	Normal Memory Remap Register
NSACR	Non-Secure Access Control Register
PAR	Physical Address Register
PMCCFILTR	Performance Monitors Cycle Count Filter Register
PMCCNTR	Performance Monitors Cycle Count Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2
PMCEID3	Performance Monitors Common Event Identification register 3
PMCNTENCLR	Performance Monitors Count Enable Clear register
PMCNTENSET	Performance Monitors Count Enable Set register
PMCR	Performance Monitors Control Register
PMINTENCLR	Performance Monitors Interrupt Enable Clear register
PMINTENSET	Performance Monitors Interrupt Enable Set register
PMMIR	Performance Monitors Machine Identification Register
PMOVSR	Performance Monitors Overflow Flag Status Register
PMOVSSET	Performance Monitors Overflow Flag Status Set register
PMSELR	Performance Monitors Event Counter Selection Register
PMSWINC	Performance Monitors Software Increment register
PMUSERENR	Performance Monitors User Enable Register
PMXEVCNTR	Performance Monitors Selected Event Count Register
PMXEVTYPER	Performance Monitors Selected Event Type Register
PRRR	Primary Region Remap Register
REVIDR	Revision ID Register
RMR	Reset Management Register
RVBAR	Reset Vector Base Address Register
SCR	Secure Configuration Register
SCTLR	System Control Register
SDCR	Secure Debug Control Register
SDER	Secure Debug Enable Register
SPSR	Saved Program Status Register
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_hyp	Saved Program Status Register (Hyp mode)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_mon	Saved Program Status Register (Monitor mode)
SPSR_svc	Saved Program Status Register (Supervisor mode)
SPSR_und	Saved Program Status Register (Undefined mode)
TCMTR	TCM Type Register
TLBIALL	TLB Invalidate All
TLBIALLH	TLB Invalidate All, Hyp mode
TLBIALLHIS	TLB Invalidate All, Hyp mode, Inner Shareable
TLBIALLIS	TLB Invalidate All, Inner Shareable
TLBIALLNSNH	TLB Invalidate All, Non-Secure Non-Hyp
TLBIALLNSNHIS	TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable
TLBIASID	TLB Invalidate by ASID match
TLBIASIDIS	TLB Invalidate by ASID match, Inner Shareable
TLBIIPAS2	TLB Invalidate by Intermediate Physical Address, Stage 2
TLBIIPAS2IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable
TLBIIPAS2L	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level
TLBIIPAS2LIS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable
TLBIMVA	TLB Invalidate by VA
TLBIMVAA	TLB Invalidate by VA, All ASID
TLBIMVAAIS	TLB Invalidate by VA, All ASID, Inner Shareable
TLBIMVAAL	TLB Invalidate by VA, All ASID, Last level
TLBIMVAALIS	TLB Invalidate by VA, All ASID, Last level, Inner Shareable
TLBIMVAH	TLB Invalidate by VA, Hyp mode
TLBIMVAHIS	TLB Invalidate by VA, Hyp mode, Inner Shareable
TLBIMVAIS	TLB Invalidate by VA, Inner Shareable
TLBIMVAL	TLB Invalidate by VA, Last level
TLBIMVALH	TLB Invalidate by VA, Last level, Hyp mode
TLBIMVALHIS	TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable
TLBIMVALIS	TLB Invalidate by VA, Last level, Inner Shareable
TLBTR	TLB Type Register
TPIDRPRW	PL1 Software Thread ID Register
TPIDRURO	PL0 Read-Only Software Thread ID Register
TPIDRURW	PL0 Read-Write Software Thread ID Register
TRFCR	Trace Filter Control Register
TTBCR	Translation Table Base Control Register
TTBCR2	Translation Table Base Control Register 2
TTBR0	Translation Table Base Register 0
TTBR1	Translation Table Base Register 1
VBAR	Vector Base Address Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
VDFSR	Virtual SError Exception Syndrome Register
VDISR	Virtual Deferred Interrupt Status Register
VMPIDR	Virtualization Multiprocessor ID Register
VPIDR	Virtualization Processor ID Register
VTCR	Virtualization Translation Control Register
VTTBR	Virtualization Translation Table Base Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.5 Functional index of AArch32 registers and System instructions

This section is an index of the AArch32 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors System registers.
- Activity Monitors System registers.
- Debug System registers.
- RAS registers.
- Generic timer registers.
- Cache maintenance System instructions.
- Address translation System instructions.
- TLB maintenance System instructions.
- Legacy feature registers and System instructions.
- Base System registers.

K14.5.1 Special-purpose registers

This section is an index to the registers in the Processor state registers functional group.

Table K14-28 Special-purpose registers

Register	Description, see
DLR	Debug Link Register
DSPSR	Debug Saved Program Status Register
ELR_hyp	Exception Link Register (Hyp mode)
SPSR	Saved Program Status Register
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_hyp	Saved Program Status Register (Hyp mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_mon	Saved Program Status Register (Monitor mode)
SPSR_svc	Saved Program Status Register (Supervisor mode)
SPSR_und	Saved Program Status Register (Undefined mode)

K14.5.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-29 VMSA-specific registers

Register	Description, see
AMAIR0	Auxiliary Memory Attribute Indirection Register 0
AMAIR1	Auxiliary Memory Attribute Indirection Register 1
CONTEXTIDR	Context ID Register
DACR	Domain Access Control Register
HAMAIR0	Hyp Auxiliary Memory Attribute Indirection Register 0
HAMAIR1	Hyp Auxiliary Memory Attribute Indirection Register 1
HMAIR0	Hyp Memory Attribute Indirection Register 0
HMAIR1	Hyp Memory Attribute Indirection Register 1
HTCR	Hyp Translation Control Register
HTTBR	Hyp Translation Table Base Register
MAIR0	Memory Attribute Indirection Register 0
MAIR1	Memory Attribute Indirection Register 1
NMRR	Normal Memory Remap Register
PRRR	Primary Region Remap Register
TTBCR	Translation Table Base Control Register
TTBCR2	Translation Table Base Control Register 2
TTBR0	Translation Table Base Register 0
TTBR1	Translation Table Base Register 1
VTCR	Virtualization Translation Control Register
VTTBR	Virtualization Translation Table Base Register

K14.5.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-30 ID registers

Register	Description, see
CCSIDR	Current Cache Size ID Register
CCSIDR2	Current Cache Size ID Register 2
CLIDR	Cache Level ID Register
CSSELR	Cache Size Selection Register
CTR	Cache Type Register
FPSID	Floating-Point System ID register
ID_AFR0	Auxiliary Feature Register 0
ID_DFR0	Debug Feature Register 0
ID_DFR1	Debug Feature Register 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_ISAR0	Instruction Set Attribute Register 0
ID_ISAR1	Instruction Set Attribute Register 1
ID_ISAR2	Instruction Set Attribute Register 2
ID_ISAR3	Instruction Set Attribute Register 3
ID_ISAR4	Instruction Set Attribute Register 4
ID_ISAR5	Instruction Set Attribute Register 5
ID_ISAR6	Instruction Set Attribute Register 6
ID_MMFR0	Memory Model Feature Register 0
ID_MMFR1	Memory Model Feature Register 1
ID_MMFR2	Memory Model Feature Register 2
ID_MMFR3	Memory Model Feature Register 3
ID_MMFR4	Memory Model Feature Register 4
ID_MMFR5	Memory Model Feature Register 5
ID_PFR0	Processor Feature Register 0
ID_PFR1	Processor Feature Register 1
ID_PFR2	Processor Feature Register 2
MIDR	Main ID Register
MPIDR	Multiprocessor Affinity Register
MVFR0	Media and VFP Feature Register 0
MVFR1	Media and VFP Feature Register 1
MVFR2	Media and VFP Feature Register 2
REVIDR	Revision ID Register
TCMTR	TCM Type Register
TLBTR	TLB Type Register
VMPIDR	Virtualization Multiprocessor ID Register
VPIDR	Virtualization Processor ID Register

K14.5.4 Performance monitors System registers

This section is an index to the System registers in the Performance Monitors registers functional group.

Table K14-31 Performance monitors registers

Register	Description, see
PMCCFILTR	Performance Monitors Cycle Count Filter Register
PMCCNTR	Performance Monitors Cycle Count Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCEID3	Performance Monitors Common Event Identification register 3
PMCNTENCLR	Performance Monitors Count Enable Clear register
PMCNTENSET	Performance Monitors Count Enable Set register
PMCR	Performance Monitors Control Register
PMINTENCLR	Performance Monitors Interrupt Enable Clear register
PMINTENSET	Performance Monitors Interrupt Enable Set register
PMMIR	Performance Monitors Machine Identification Register
PMOVSR	Performance Monitors Overflow Flag Status Register
PMOVSSET	Performance Monitors Overflow Flag Status Set register
PMSELR	Performance Monitors Event Counter Selection Register
PMSWINC	Performance Monitors Software Increment register
PMUSERENR	Performance Monitors User Enable Register
PMXEVCNTR	Performance Monitors Selected Event Count Register
PMXEVTYPER	Performance Monitors Selected Event Type Register

K14.5.5 Activity Monitors System registers

This section is an index to the System registers in the Activity Monitors registers functional group.

Table K14-32 Activity monitors registers

Register	Description, see
AMCFGR	Activity Monitors Configuration Register
AMCGCR	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1
AMCR	Activity Monitors Control Register
AMUSERENR	Activity Monitors User Enable Register

K14.5.6 Debug System registers

This section is an index to the System registers in the Debug registers functional group.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-33 Debug registers

Register	Description, see
DBGAUTHSTATUS	Debug Authentication Status register
DBGCLAIMCLR	Debug CLAIM Tag Clear register
DBGCLAIMSET	Debug CLAIM Tag Set register
DBGDCCINT	DCC Interrupt Enable Register
DBGDEVID	Debug Device ID register 0
DBGDEVID1	Debug Device ID register 1
DBGDEVID2	Debug Device ID register 2
DBGDIDR	Debug ID Register
DBGDRAR	Debug ROM Address Register
DBGDSAR	Debug Self Address Register
DBGDSCRext	Debug Status and Control Register, External View
DBGDSCRint	Debug Status and Control Register, Internal View
DBGDTRRXext	Debug OS Lock Data Transfer Register, Receive, External View
DBGDTRRXint	Debug Data Transfer Register, Receive
DBGDTRTXext	Debug OS Lock Data Transfer Register, Transmit
DBGDTRTXint	Debug Data Transfer Register, Transmit
DBGOSDLR	Debug OS Double Lock Register
DBGOSECCR	Debug OS Lock Exception Catch Control Register
DBGOSLAR	Debug OS Lock Access Register
DBGOSLSR	Debug OS Lock Status Register
DBGPRCR	Debug Power Control Register
DBGVCR	Debug Vector Catch Register
DBGWFAR	Debug Watchpoint Fault Address Register
HDCR	Hyp Debug Control Register
HTRFCR	Hyp Trace Filter Control Register
TRFCR	Trace Filter Control Register

K14.5.7 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-34 RAS registers

Register	Description, see
DISR	Deferred Interrupt Status Register
ERRIDR	Error Record ID Register
ERRSELR	Error Record Select Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ERXADDR	Selected Error Record Address Register
ERXADDR2	Selected Error Record Address Register 2
ERXCTLR	Selected Error Record Control Register
ERXCTLR2	Selected Error Record Control Register 2
ERXFR	Selected Error Record Feature Register
ERXFR2	Selected Error Record Feature Register 2
ERXMISC0	Selected Error Record Miscellaneous Register 0
ERXMISC1	Selected Error Record Miscellaneous Register 1
ERXMISC2	Selected Error Record Miscellaneous Register 2
ERXMISC3	Selected Error Record Miscellaneous Register 3
ERXMISC4	Selected Error Record Miscellaneous Register 4
ERXMISC5	Selected Error Record Miscellaneous Register 5
ERXMISC6	Selected Error Record Miscellaneous Register 6
ERXMISC7	Selected Error Record Miscellaneous Register 7
ERXSTATUS	Selected Error Record Primary Status Register
VDFSR	Virtual SError Exception Syndrome Register
VDISR	Virtual Deferred Interrupt Status Register

K14.5.8 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K14-35 Generic timer registers

Register	Description, see
CNTFRQ	Counter-timer Frequency register
CNTHCTL	Counter-timer Hyp Control register
CNTHP_CTL	Counter-timer Hyp Physical Timer Control register
CNTHP_CVAL	Counter-timer Hyp Physical CompareValue register
CNTHP_TVAL	Counter-timer Hyp Physical Timer TimerValue register
CNTHPS_CTL	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL	Counter-timer Virtual Timer Control register (EL2)
CNTHV_CVAL	Counter-timer Virtual Timer CompareValue register (EL2)
CNTHV_TVAL	Counter-timer Virtual Timer TimerValue register (EL2)
CNTHVS_CTL	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL	Counter-timer Secure Virtual Timer TimerValue Register (EL2)

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTKCTL	Counter-timer Kernel Control register
CNTP_CTL	Counter-timer Physical Timer Control register
CNTP_CVAL	Counter-timer Physical Timer CompareValue register
CNTP_TVAL	Counter-timer Physical Timer TimerValue register
CNTPCT	Counter-timer Physical Count register
CNTPCTSS	Counter-timer Self-Synchronized Physical Count register
CNTV_CTL	Counter-timer Virtual Timer Control register
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register
CNTVCT	Counter-timer Virtual Count register
CNTVCTSS	Counter-timer Self-Synchronized Virtual Count register
CNTVOFF	Counter-timer Virtual Offset register

K14.5.9 Cache maintenance System instructions

This section is an index to the System instructions in the Cache maintenance instructions functional group.

Table K14-36 Cache maintenance system instructions

System instruction	Description, see
BPIALL	Branch Predictor Invalidate All
BPIALLIS	Branch Predictor Invalidate All, Inner Shareable
BPIMVA	Branch Predictor Invalidate by VA
DCCIMVAC	Data Cache line Clean and Invalidate by VA to PoC
DCCISW	Data Cache line Clean and Invalidate by Set-Way
DCCMVAC	Data Cache line Clean by VA to PoC
DCCMVAU	Data Cache line Clean by VA to PoU
DCCSW	Data Cache line Clean by Set-Way
DCIMVAC	Data Cache line Invalidate by VA to PoC
DCISW	Data Cache line Invalidate by Set-Way
ICIALLU	Instruction Cache Invalidate All to PoU
ICIALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
ICIMVAU	Instruction Cache line Invalidate by VA to PoU

K14.5.10 Address translation System instructions

This section is an index to the System instructions in the Address translation instructions functional group.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-37 Address translation system instructions

System instruction	Description, see
ATS12NSOPR	Address Translate Stages 1 and 2 Non-secure Only PL1 Read
ATS12NSOPW	Address Translate Stages 1 and 2 Non-secure Only PL1 Write
ATS12NSOUR	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read
ATS12NSOUW	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write
ATS1CPR	Address Translate Stage 1 Current state PL1 Read
ATS1CPRP	Address Translate Stage 1 Current state PL1 Read PAN
ATS1CPW	Address Translate Stage 1 Current state PL1 Write
ATS1CPWP	Address Translate Stage 1 Current state PL1 Write PAN
ATS1CUR	Address Translate Stage 1 Current state Unprivileged Read
ATS1CUW	Address Translate Stage 1 Current state Unprivileged Write
ATS1HR	Address Translate Stage 1 Hyp mode Read
ATS1HW	Address Translate Stage 1 Hyp mode Write

K14.5.11 TLB maintenance System instructions

This section is an index to the System instructions in the TLB maintenance instructions functional group.

Table K14-38 TLB maintenance system instructions

System instruction	Description, see
COSPRCTX	Clear Other Speculative Prediction Restriction by Context
DTLBIALL	Data TLB Invalidate All
DTLBIASID	Data TLB Invalidate by ASID match
DTLBIMVA	Data TLB Invalidate by VA
ITLBIALL	Instruction TLB Invalidate All
ITLBIASID	Instruction TLB Invalidate by ASID match
ITLBIMVA	Instruction TLB Invalidate by VA
TLBIALL	TLB Invalidate All
TLBIALLH	TLB Invalidate All, Hyp mode
TLBIALLHIS	TLB Invalidate All, Hyp mode, Inner Shareable
TLBIALLIS	TLB Invalidate All, Inner Shareable
TLBIALLNSNH	TLB Invalidate All, Non-Secure Non-Hyp
TLBIALLNSNHIS	TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable
TLBIASID	TLB Invalidate by ASID match
TLBIASIDIS	TLB Invalidate by ASID match, Inner Shareable
TLBIIPAS2	TLB Invalidate by Intermediate Physical Address, Stage 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBIIPAS2IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable
TLBIIPAS2L	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level
TLBIIPAS2LIS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable
TLBIMVA	TLB Invalidate by VA
TLBIMVAA	TLB Invalidate by VA, All ASID
TLBIMVAAIS	TLB Invalidate by VA, All ASID, Inner Shareable
TLBIMVAAL	TLB Invalidate by VA, All ASID, Last level
TLBIMVAALIS	TLB Invalidate by VA, All ASID, Last level, Inner Shareable
TLBIMVAH	TLB Invalidate by VA, Hyp mode
TLBIMVAHIS	TLB Invalidate by VA, Hyp mode, Inner Shareable
TLBIMVAIS	TLB Invalidate by VA, Inner Shareable
TLBIMVAL	TLB Invalidate by VA, Last level
TLBIMVALH	TLB Invalidate by VA, Last level, Hyp mode
TLBIMVALHIS	TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable
TLBIMVALIS	TLB Invalidate by VA, Last level, Inner Shareable

K14.5.12 Prediction restriction instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table K14-39 Cache maintenance system instructions

System instruction	Description, see
CFPRCTX	Control Flow Prediction Restriction by Context
CPPRCTX	Cache Prefetch Prediction Restriction by Context
DVPRCTX	Data Value Prediction Restriction by Context

K14.5.13 Legacy feature registers and system instructions

This section is an index to the registers in the Legacy feature registers functional group.

Table K14-40 Legacy feature registers and system instructions

Register	Description, see
CP15DMB	Data Memory Barrier System instruction
CP15DSB	Data Synchronization Barrier System instruction

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CP15ISB	Instruction Synchronization Barrier System instruction
FCSEIDR	FCSE Process ID register
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register

K14.5.14 Base System registers

This section is an index to the System registers in the Base System registers functional group.

Table K14-41 Base system registers

Register	Description, see
CPACR	Architectural Feature Access Control Register
FCSEIDR	FCSE Process ID register
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register
SCTLR	System Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.6 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

Table K14-42 Alphabetical index of Memory-Mapped Registers

Register	Description, see
AMCFGR	Activity Monitors Configuration Register
AMCGCR	Activity Monitors Counter Group Configuration Register
AMCIDR0	Activity Monitors Component Identification Register 0
AMCIDR1	Activity Monitors Component Identification Register 1
AMCIDR2	Activity Monitors Component Identification Register 2
AMCIDR3	Activity Monitors Component Identification Register 3
AMCNTEN	Activity Monitors Count Set and Clear Register
AMCNTENCLR	Activity Monitors Count Enable Clear Register
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1
AMCNTENSET	Activity Monitors Count Enable Set Register
AMCNTENSET0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1
AMCR	Activity Monitors Control Register
AMDEVAFF	Activity Monitors Device Affinity Register
AMDEVAFF0	Activity Monitors Device Affinity Register 0
AMDEVAFF1	Activity Monitors Device Affinity Register 1
AMDEVARCH	Activity Monitors Device Architecture Register
AMDEVTYPE	Activity Monitors Device Type Register
AMIIDR	Activity Monitors Implementation Identification Register
AMPIDR0	Activity Monitors Peripheral Identification Register 0
AMPIDR1	Activity Monitors Peripheral Identification Register 1
AMPIDR2	Activity Monitors Peripheral Identification Register 2
AMPIDR3	Activity Monitors Peripheral Identification Register 3
AMPIDR4	Activity Monitors Peripheral Identification Register 4
AMROOTCR	Activity Monitors Root Control Register
AMSCR	Activity Monitors Secure Control Register
ASICCTL	CTI External Multiplexer Control register
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency
CNTID	Counter Identification Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer CompareValue
CNTP_TVAL	Counter-timer Physical Timer TimerValue
CNTPCT	Counter-timer Physical Count
CNTSCR	Counter Scale Register
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer CompareValue
CNTV_TVAL	Counter-timer Virtual Timer TimerValue
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register
CTIDEVCTL	CTI Device Control register
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDCIDSR	External Debug Context ID Sample Register
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDDFR	External Debug Feature Register
EDDFR1	External Debug Feature Register 1
EDDFR2	External Debug Feature Register 2
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDHSR	External Debug Halting Syndrome Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPCSR	External Debug Program Counter Sample Register
EDPFR	External Debug Processor Feature Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDSCR2	External Debug Status and Control Register 2
EDVIDSR	External Debug Virtual Context Sample Register
EDWAR	External Debug Watchpoint Address Register
ERRACR	Access Configuration Register
ERRCIDR0	Component Identification Register 0
ERRCIDR1	Component Identification Register 1
ERRCIDR2	Component Identification Register 2
ERRCIDR3	Component Identification Register 3
ERRCRICR0	Critical Error Interrupt Configuration Register 0
ERRCRICR1	Critical Error Interrupt Configuration Register 1
ERRCRICR2	Critical Error Interrupt Configuration Register 2
ERRDEVAFF	Device Affinity Register
ERRDEVARCH	Device Architecture Register
ERRDEVID	Device Configuration Register
ERRERICR0	Error Recovery Interrupt Configuration Register 0
ERRERICR1	Error Recovery Interrupt Configuration Register 1
ERRERICR2	Error Recovery Interrupt Configuration Register 2
ERRFHICR0	Fault Handling Interrupt Configuration Register 0
ERRFHICR1	Fault Handling Interrupt Configuration Register 1
ERRFHICR2	Fault Handling Interrupt Configuration Register 2
ERRIIDR	Implementation Identification Register
ERRIRQSR	Error Interrupt Status Register
ERRPIDR0	Peripheral Identification Register 0
ERRPIDR1	Peripheral Identification Register 1
ERRPIDR2	Peripheral Identification Register 2
ERRPIDR3	Peripheral Identification Register 3
ERRPIDR4	Peripheral Identification Register 4
GICC_ABPR	CPU Interface Aliased Binary Point Register
GICC_AEOIR	CPU Interface Aliased End Of Interrupt Register
GICC_AHPPIR	CPU Interface Aliased Highest Priority Pending Interrupt Register
GICC_AIAR	CPU Interface Aliased Interrupt Acknowledge Register
GICC_BPR	CPU Interface Binary Point Register
GICC_CTLR	CPU Interface Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GICC_DIR	CPU Interface Deactivate Interrupt Register
GICC_EOIR	CPU Interface End Of Interrupt Register
GICC_HPPIR	CPU Interface Highest Priority Pending Interrupt Register
GICC_IAR	CPU Interface Interrupt Acknowledge Register
GICC_IIDR	CPU Interface Identification Register
GICC_PMR	CPU Interface Priority Mask Register
GICC_RPR	CPU Interface Running Priority Register
GICC_STATUSR	CPU Interface Status Register
GICD_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICD_CLRSPI_SR	Clear Secure SPI Pending Register
GICD_CTLR	Distributor Control Register
GICD_IIDR	Distributor Implementer Identification Register
GICD_SETSPI_NSR	Set Non-secure SPI Pending Register
GICD_SETSPI_SR	Set Secure SPI Pending Register
GICD_SGIR	Software Generated Interrupt Register
GICD_STATUSR	Error Reporting Status Register
GICD_TYPER	Interrupt Controller Type Register
GICD_TYPER2	Interrupt Controller Type Register 2
GICH_EISR	End Interrupt Status Register
GICH_ELRSR	Empty List Register Status Register
GICH_HCR	Hypervisor Control Register
GICH_MISR	Maintenance Interrupt Status Register
GICH_VMCR	Virtual Machine Control Register
GICH_VTR	Virtual Type Register
GICM_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICM_CLRSPI_SR	Clear Secure SPI Pending Register
GICM_IIDR	Distributor Implementer Identification Register
GICM_SETSPI_NSR	Set Non-secure SPI Pending Register
GICM_SETSPI_SR	Set Secure SPI Pending Register
GICM_TYPER	Distributor MSI Type Register
GICR_CLRLPIR	Clear LPI Pending Register
GICR_CTLR	Redistributor Control Register
GICR_ICACTIVER0	Interrupt Clear-Active Register 0
GICR_ICENABLER0	Interrupt Clear-Enable Register 0
GICR_ICFGR0	Interrupt Configuration Register 0
GICR_ICFGR1	Interrupt Configuration Register 1
GICR_ICPENDR0	Interrupt Clear-Pending Register 0
GICR_IGROUPR0	Interrupt Group Register 0
GICR_IGRPMODR0	Interrupt Group Modifier Register 0
GICR_IIDR	Redistributor Implementer Identification Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GICR_INMIR0	Non-maskable Interrupt Register 0
GICR_INVALLR	Redistributor Invalidate All Register
GICR_INVLPIR	Redistributor Invalidate LPI Register
GICR_ISACTIVER0	Interrupt Set-Active Register 0
GICR_ISENABLER0	Interrupt Set-Enable Register 0
GICR_ISPENDR0	Interrupt Set-Pending Register 0
GICR_MPAMIDR	Report maximum PARTID and PMG Register
GICR_NSACR	Non-secure Access Control Register
GICR_PARTIDR	Set PARTID and PMG Register
GICR_PENDBASER	Redistributor LPI Pending Table Base Address Register
GICR_PROPBASER	Redistributor Properties Base Address Register
GICR_SETLPIR	Set LPI Pending Register
GICR_STATUSR	Error Reporting Status Register
GICR_SYNCR	Redistributor Synchronize Register
GICR_TYPER	Redistributor Type Register
GICR_VPENDBASER	Virtual Redistributor LPI Pending Table Base Address Register
GICR_VPROPBASER	Virtual Redistributor Properties Base Address Register
GICR_VSGIPENDR	Redistributor virtual SGI pending state register
GICR_VSGIR	Redistributor virtual SGI pending state request register
GICR_WAKER	Redistributor Wake Register
GICV_ABPR	Virtual Machine Aliased Binary Point Register
GICV_AEOIR	Virtual Machine Aliased End Of Interrupt Register
GICV_AHPPIR	Virtual Machine Aliased Highest Priority Pending Interrupt Register
GICV_AIAR	Virtual Machine Aliased Interrupt Acknowledge Register
GICV_BPR	Virtual Machine Binary Point Register
GICV_CTLR	Virtual Machine Control Register
GICV_DIR	Virtual Machine Deactivate Interrupt Register
GICV_EOIR	Virtual Machine End Of Interrupt Register
GICV_HPPIR	Virtual Machine Highest Priority Pending Interrupt Register
GICV_IAR	Virtual Machine Interrupt Acknowledge Register
GICV_IIDR	Virtual Machine CPU Interface Identification Register
GICV_PMR	Virtual Machine Priority Mask Register
GICV_RPR	Virtual Machine Running Priority Register
GICV_STATUSR	Virtual Machine Error Reporting Status Register
GITS_CBASER	ITS Command Queue Descriptor
GITS_CREADR	ITS Read Register
GITS_CTLR	ITS Control Register
GITS_CWRITER	ITS Write Register
GITS_IIDR	ITS Identification Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GITS_MPAMIDR	Report maximum PARTID and PMG Register
GITS_MPIDR	Report ITS's affinity.
GITS_PARTIDR	Set PARTID and PMG Register
GITS_SGIR	ITS SGI Register
GITS_STATUSR	ITS Error Reporting Status Register
GITS_TRANSLATER	ITS Translation Register
GITS_TYPER	ITS Type Register
GITS_UMSIR	ITS Unmapped MSI register
MIDR_EL1	Main ID Register
MPAMCFG_CASSOC	MPAM Cache Maximum Associativity Partition Configuration Register
MPAMCFG_CMAX	MPAM Cache Maximum Capacity Partition Configuration Register
MPAMCFG_CMIN	MPAM Cache Minimum Capacity Partition Configuration Register
MPAMCFG_DIS	MPAM Partition Configuration Disable Register
MPAMCFG_EN	MPAM Partition Configuration Enable Register
MPAMCFG_EN_FLAGS	MPAM Partition Configuration Enable Flags Register
MPAMCFG_IN_TL	MPAM Ingress PARTID Translation Configuration Register
MPAMCFG_IN_TL_BASE	MPAM Ingress PARTID Translation Base Configuration Register
MPAMCFG_IN_TL_MASK	MPAM Ingress PARTID Translation Mask Configuration Register
MPAMCFG_INTPARTID	MPAM Internal PARTID Narrowing Configuration Register
MPAMCFG_MBW_MAX	MPAM Memory Bandwidth Maximum Partition Configuration Register
MPAMCFG_MBW_MIN	MPAM Memory Bandwidth Minimum Partition Configuration Register
MPAMCFG_MBW_PROP	MPAM Memory Bandwidth Proportional Stride Partition Configuration Register
MPAMCFG_MBW_WINWD	MPAM Memory Bandwidth Partitioning Window Width Configuration Register
MPAMCFG_OUT_TL	MPAM Egress PARTID Translation Configuration Register
MPAMCFG_OUT_TL_BASE	MPAM Egress PARTID Translation Base Configuration Register
MPAMCFG_OUT_TL_MASK	MPAM Egress PARTID Translation Mask Configuration Register
MPAMCFG_PART_SEL	MPAM Partition Configuration Selection Register
MPAMCFG_PRI	MPAM Priority Partition Configuration Register
MPAMF_AIDR	MPAM Architecture Identification Register
MPAMF_CCAP_IDR	MPAM Features Cache Capacity Partitioning ID register
MPAMF_CPOR_IDR	MPAM Features Cache Portion Partitioning ID register
MPAMF_CSAMON_IDR	MPAM Features Cache Storage Allocation Monitoring ID register
MPAMF_CSUMON_IDR	MPAM Features Cache Storage Usage Monitoring ID register
MPAMF_ECR	MPAM Error Control Register
MPAMF_ERR_MSI_ADDR_H	MPAM Error MSI High-part Address Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MPAMF_ERR_MSI_ADDR_L	MPAM Error MSI Low-part Address Register
MPAMF_ERR_MSI_ATTR	MPAM Error MSI Write Attributes Register
MPAMF_ERR_MSI_DATA	MPAM Error MSI Data Register
MPAMF_ERR_MSI_MPAM	MPAM Error MSI Write MPAM Information Register
MPAMF_ESR	MPAM Error Status Register
MPAMF_IDR	MPAM Features Identification Register
MPAMF_IIDR	MPAM Implementation Identification Register
MPAMF_IMPL_IDR	MPAM Implementation-Specific Partitioning Feature Identification Register
MPAMF_IN_TL_IDR	MPAM Ingress PARTID Translation ID Register
MPAMF_MBW_IDR	MPAM Memory Bandwidth Partitioning Identification Register
MPAMF_MBWUMON_IDR	MPAM Features Memory Bandwidth Usage Monitoring ID register
MPAMF_MSMON_IDR	MPAM Resource Monitoring Identification Register
MPAMF_OUT_TL_IDR	MPAM Egress PARTID Translation ID Register
MPAMF_PARTID_NRW_IDR	MPAM PARTID Narrowing ID register
MPAMF_PRI_IDR	MPAM Priority Partitioning Identification Register
MPAMF_SIDR	MPAM Features Secure Identification Register
MSMON_CAPT_EVNT	MPAM Capture Event Generation Register
MSMON_CFG_CSA_CTL	MPAM Memory System Monitor Configure Cache Storage Allocation Monitor Control Register
MSMON_CFG_CSA_FLT	MPAM Memory System Monitor Configure Cache Storage Allocation Monitor Filter Register
MSMON_CFG_CSU_CTL	MPAM Memory System Monitor Configure Cache Storage Usage Monitor Control Register
MSMON_CFG_CSU_FLT	MPAM Memory System Monitor Configure Cache Storage Usage Monitor Filter Register
MSMON_CFG_MBWU_CTL	MPAM Memory System Monitor Configure Memory Bandwidth Usage Monitor Control Register
MSMON_CFG_MBWU_FLT	MPAM Memory System Monitor Configure Memory Bandwidth Usage Monitor Filter Register
MSMON_CFG_MON_SEL	MPAM Monitor Instance Selection Register
MSMON_CSA	MPAM Cache Storage Allocation Monitor Register
MSMON_CSA_CAPTURE	MPAM Cache Storage Allocation Monitor Capture Register
MSMON_CSA_L	MPAM Long Cache Storage Allocation Monitor Register
MSMON_CSA_L_CAPTURE	MPAM Long Cache Storage Allocation Monitor Capture Register
MSMON_CSA_OFSR	MPAM CSA Monitor Overflow Status Register
MSMON_CSU	MPAM Cache Storage Usage Monitor Register
MSMON_CSU_CAPTURE	MPAM Cache Storage Usage Monitor Capture Register
MSMON_CSU_OFSR	MPAM CSU Monitor Overflow Status Register
MSMON_MBWU	MPAM Memory Bandwidth Usage Monitor Register
MSMON_MBWU_CAPTURE	MPAM Memory Bandwidth Usage Monitor Capture Register
MSMON_MBWU_L	MPAM Long Memory Bandwidth Usage Monitor Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MSMON_MBWU_L_CAPTURE	MPAM Long Memory Bandwidth Usage Monitor Capture Register
MSMON_MBWU_OFSR	MPAM MBWU Monitor Overflow Status Register
MSMON_OFLOW_MSI_ADDR_H	MPAM Monitor Overflow MSI Write High-part Address Register
MSMON_OFLOW_MSI_ADDR_L	MPAM Monitor Overflow MSI Low-part Address Register
MSMON_OFLOW_MSI_ATTR	MPAM Monitor Overflow MSI Write Attributes Register
MSMON_OFLOW_MSI_DATA	MPAM Monitor Overflow MSI Write Data Register
MSMON_OFLOW_MSI_MPAM	MPAM Monitor Overflow MSI Write MPAM Information Register
MSMON_OFLOW_SR	MPAM Monitor Overflow Status Register
OSLAR_EL1	OS Lock Access Register
TRBAUTHSTATUS	Authentication Status Register
TRBBASER_EL1	Trace Buffer Base Address Register
TRBCIDR0	Component Identification Register 0
TRBCIDR1	Component Identification Register 1
TRBCIDR2	Component Identification Register 2
TRBCIDR3	Component Identification Register 3
TRBCR	Trace Buffer Control Register
TRBDEVAFF	Device Affinity Register
TRBDEVARCH	Trace Buffer Device Architecture Register
TRBDEVID	Device Configuration Register
TRBDEVID1	Device Configuration Register 1
TRBDEVID2	Device Configuration Register 2
TRBDEVTYPE	Device Type Register
TRBIDR_EL1	Trace Buffer ID Register
TRBITCTRL	Integration Mode Control Register
TRBLAR	Lock Access Register
TRBLIMITR_EL1	Trace Buffer Limit Address Register
TRBLSR	Lock Status Register
TRBMAR_EL1	Trace Buffer Memory Attribute Register
TRBMPAM_EL1	Trace Buffer MPAM Configuration Register
TRBPIDR0	Peripheral Identification Register 0
TRBPIDR1	Peripheral Identification Register 1
TRBPIDR2	Peripheral Identification Register 2
TRBPIDR3	Peripheral Identification Register 3
TRBPIDR4	Peripheral Identification Register 4
TRBPIDR5	Peripheral Identification Register 5
TRBPIDR6	Peripheral Identification Register 6
TRBPIDR7	Peripheral Identification Register 7
TRBPTR_EL1	Trace Buffer Write Pointer Register
TRBSR_EL1	Trace Buffer Status-syndrome Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRBTRG_EL1	Trace Buffer Trigger Counter Register
TRCAUTHSTATUS	Trace Authentication Status Register
TRCAUXCTLR	Trace Auxiliary Control Register
TRCBBCTLR	Trace Branch Broadcast Control Register
TRCCCCTLR	Trace Cycle Count Control Register
TRCCIDCCTLR0	Trace Context Identifier Comparator Control Register 0
TRCCIDCCTLR1	Trace Context Identifier Comparator Control Register 1
TRCCIDR0	Trace Component Identification Register 0
TRCCIDR1	Trace Component Identification Register 1
TRCCIDR2	Trace Component Identification Register 2
TRCCIDR3	Trace Component Identification Register 3
TRCCLAIMCLR	Trace Claim Tag Clear Register
TRCCLAIMSET	Trace Claim Tag Set Register
TRCCONFIGR	Trace Configuration Register
TRCDEVAFF	Trace Device Affinity Register
TRCDEVARCH	Trace Device Architecture Register
TRCDEVID	Trace Device Configuration Register
TRCDEVID1	Trace Device Configuration Register 1
TRCDEVID2	Trace Device Configuration Register 2
TRCDEVTYPE	Trace Device Type Register
TRCEVENTCTL0R	Trace Event Control 0 Register
TRCEVENTCTL1R	Trace Event Control 1 Register
TRCIDR0	Trace ID Register 0
TRCIDR1	Trace ID Register 1
TRCIDR10	Trace ID Register 10
TRCIDR11	Trace ID Register 11
TRCIDR12	Trace ID Register 12
TRCIDR13	Trace ID Register 13
TRCIDR2	Trace ID Register 2
TRCIDR3	Trace ID Register 3
TRCIDR4	Trace ID Register 4
TRCIDR5	Trace ID Register 5
TRCIDR6	Trace ID Register 6
TRCIDR7	Trace ID Register 7
TRCIDR8	Trace ID Register 8
TRCIDR9	Trace ID Register 9
TRCIMSPEC0	Trace IMP DEF Register 0
TRCITCTRL	Trace Integration Mode Control Register
TRCITEEDCR	Instrumentation Trace Extension External Debug Control Register
TRCLAR	Trace Lock Access Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCLSR	Trace Lock Status Register
TRCOSLSR	Trace OS Lock Status Register
TRCPDCR	Trace PowerDown Control Register
TRCPDSR	Trace PowerDown Status Register
TRCPIDR0	Trace Peripheral Identification Register 0
TRCPIDR1	Trace Peripheral Identification Register 1
TRCPIDR2	Trace Peripheral Identification Register 2
TRCPIDR3	Trace Peripheral Identification Register 3
TRCPIDR4	Trace Peripheral Identification Register 4
TRCPIDR5	Trace Peripheral Identification Register 5
TRCPIDR6	Trace Peripheral Identification Register 6
TRCPIDR7	Trace Peripheral Identification Register 7
TRCPRGCTLR	Trace Programming Control Register
TRCQCTLR	Trace Q Element Control Register
TRCRSR	Trace Resources Status Register
TRCSEQRSTEVR	Trace Sequencer Reset Control Register
TRCSEQSTR	Trace Sequencer State Register
TRCSTALLCTLR	Trace Stall Control Register
TRCSTATR	Trace Status Register
TRCSYNCPR	Trace Synchronization Period Register
TRCTRACEIDR	Trace ID Register
TRCTSCTLR	Trace Timestamp Control Register
TRCVICTLR	Trace ViewInst Main Control Register
TRCVIIECTLR	Trace ViewInst Include-Exclude Control Register
TRCVIPCSSCTLR	Trace ViewInst Start-Stop PE Comparator Control Register
TRCVISSCTLR	Trace ViewInst Start-Stop Control Register
TRCVMIDCCTLR0	Trace Virtual Context Identifier Comparator Control Register 0
TRCVMIDCCTLR1	Trace Virtual Context Identifier Comparator Control Register 1
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCIDSR	CONTEXTIDR_ELx Sample Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2
PMCEID3	Performance Monitors Common Event Identification register 3
PMCGCR0	Counter Group Configuration Register 0
PMCFGR	Performance Monitors Configuration Register
PMCID1SR	CONTEXTIDR_EL1 Sample Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCID2SR	CONTEXTIDR_EL2 Sample Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTEN	Performance Monitors Count Enable register
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVAFF	Performance Monitors Device Affinity register
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVID	Performance Monitors Device ID register
PMDEVTYPE	Performance Monitors Device Type register
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMICNTSVR_EL1	Performance Monitors Instruction Count Saved Value Register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTEN	Performance Monitors Interrupt Enable register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMMIR	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVS	Performance Monitors Overflow Flag Status register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPCSCTL	PC Sample-based Profiling Control Register
PMCCR	PMU Configuration Control Register
PMPCSR	Program Counter Sample Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMSSCR_EL1	Performance Monitors Snapshot Status and Capture Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMVIDSR	VMID Sample Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMVCIDSR	CONTEXTIDR_EL1 and VMID Sample Register
PMZR_EL0	Performance Monitors Zero with Mask

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.7 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group. Each of the following sections lists the registers for a functional group:

- ID registers.
- Memory-mapped Performance monitors registers.
- Generic timer registers.
- Debug registers.
- Cross-trigger interface registers.

K14.7.1 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-43 ID registers

Register	Description, see
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDDFR	External Debug Feature Register
EDDFR1	External Debug Feature Register 1
EDDFR2	External Debug Feature Register 2
EDPFR	External Debug Processor Feature Register
MIDR_EL1	Main ID Register

K14.7.2 Memory-mapped Performance monitors registers

This section is an index to the memory-mapped registers in the Performance Monitors registers functional group.

Table K14-44 Performance monitors registers

Register	Description, see
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCIDSR	CONTEXTIDR_ELx Sample Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCEID3	Performance Monitors Common Event Identification register 3
PMCGCR0	Counter Group Configuration Register 0
PMCFGR	Performance Monitors Configuration Register
PMCID1SR	CONTEXTIDR_EL1 Sample Register
PMCID2SR	CONTEXTIDR_EL2 Sample Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTEN	Performance Monitors Count Enable register
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVAFF	Performance Monitors Device Affinity register
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVID	Performance Monitors Device ID register
PMDEVTYPE	Performance Monitors Device Type register
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMICNTSVR_EL1	Performance Monitors Instruction Count Saved Value Register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTEN	Performance Monitors Interrupt Enable register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMMIR	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVS	Performance Monitors Overflow Flag Status register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPCSCTL	PC Sample-based Profiling Control Register
PMCCR	PMU Configuration Control Register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMPCSR	Program Counter Sample Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMSSCR_EL1	Performance Monitors Snapshot Status and Capture Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMVIDSR	VMID Sample Register
PMVCIDSR	CONTEXTIDR_EL1 and VMID Sample Register
PMZR_EL0	Performance Monitors Zero with Mask

K14.7.3 Generic timer registers

This section is an index to the registers in the Generic timer registers functional group.

Table K14-45 Generic timer registers

Register	Description, see
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency
CNTID	Counter Identification Register
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer CompareValue
CNTP_TVAL	Counter-timer Physical Timer TimerValue
CNTPCT	Counter-timer Physical Count
CNTSCR	Counter Scale Register
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer CompareValue
CNTV_TVAL	Counter-timer Virtual Timer TimerValue
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.7.4 Memory-mapped Debug registers

This section is an index to the memory-mapped registers in the Debug registers functional group.

Table K14-46 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDCIDSR	External Debug Context ID Sample Register
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDHSR	External Debug Halting Syndrome Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPCSR	External Debug Program Counter Sample Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDSCR2	External Debug Status and Control Register 2
EDVIDSR	External Debug Virtual Context Sample Register
EDWAR	External Debug Watchpoint Address Register
OSLAR_EL1	OS Lock Access Register

K14.7.5 Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

Table K14-47 Cross-trigger interface registers

Register	Description, see
ASICCTL	CTI External Multiplexer Control register
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register
CTIDEVCTL	CTI Device Control register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

A32 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in A32 state. A32 instructions must be word-aligned.

A32 instructions were previously called ARM instructions.

See also A32 state, A64 instruction, T32 instruction.

A32 state The AArch32 Instruction set state in which the PE executes A32 instructions.

A32 state was previously called ARM state.

See also T32 instruction, T32 state.

A64 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned. See also A32 instruction, T32 instruction.

AArch32 The 32-bit Execution state. In AArch32 state, addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

See also AArch64, A32 instruction, T32 instruction.

AArch64 The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

See also AArch32, A64 instruction.

Abort An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

Active element An Active element is an SVE vector element or predicate element that is a source register element or destination register element used by an SVE instruction. When the corresponding element in the instruction's Governing predicate is TRUE, the element is Active. If an instruction is unpredicated, all of the vector elements or predicate elements are implicitly Active.

Addressing mode Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Aligned A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

ALTSP Alternative PARTID space for MPAM.

AMBA Advanced Microcontroller Bus Architecture. The AMBA family of protocol specifications is the Arm open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.

Analysis of the trace element stream Refers to the process of:

- Tracing elements that carry information that a trace analyzer requires to enable it to analyze the trace successfully.
- Tracing elements that either directly indicate program execution, or carry information about program execution.

A trace element stream might also contain trace elements that contain timing information. This term is distinct from analysis of program execution.

Architecturally executed An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired, it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

See also Condition code check, Simple sequential execution.

Architecturally mapped Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

Architecturally UNKNOWN An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also IMPLEMENTATION DEFINED.

Architecturally Allowed Execution An *Architecturally Allowed Execution* must satisfy the External visibility requirement.

ARM core registers Some older documentation uses *ARM core registers* to refer to the following set of registers for execution in AArch32 state:

- The 13 general-purpose registers, R0-R12, that software can use for processing.
- SP, the *stack pointer*, that can also be referred to as R13.
- LR, the *link register*, that can also be referred to as R14.
- PC, the *Program Counter*, that can also be referred to as R15.

See also General-purpose registers.

ARM instruction See A32 instruction.

Associativity See Cache associativity.

Asynchronous accumulation Faults that are accumulated in a status register, where the update to the register is asynchronous to the instruction that causes the fault.

Atomicity Describes either single-copy atomicity or multi-copy atomicity. Atomicity in the Arm architecture defines these forms of atomicity for the Arm architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also Multi-copy atomicity and Single-copy atomicity.

Might also refer to poison-atomicity. See Poison atomicity.

Availability Readiness for correct service.

Banked register A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

Baseboard Management Controller A PE dedicated to system control and monitoring.

Base register A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register writeback Describes writing back a modified value to the base register used in an address calculation.

Behaves as if Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Big-endian memory Means that, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Endianness, Little-endian memory.

Blocking Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

Branch History Buffer (BHB) The Branch History Buffer is an internal structure that stores branch instruction information including the history of previously executed branches.

Branch prediction Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

See also Prefetching.

Breakpoint A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

Built-in self test (BIST) A mechanism that permits a machine to test itself.

Burst A group of transfers that form a single transaction.

BWA BandWidth Allocation.

BWPBM BandWidth Portion Bit Map.

Byte An 8-bit data item.

CE See Corrected Error.

Cache associativity The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

Cache level The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see Terms used in describing the cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cache line The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

Cache lockdown Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.

Cache miss A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.

Cache way A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set.

Coherence order See Coherent.

Coherent Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

Commit window The Commit window defines the range of *P0 elements* that are committed by a *Commit element*. The oldest *P0 element* in the Commit window is the first *P0 element* committed when a *Commit element* occurs. By default, the Commit window starts on the oldest uncommitted *P0 element* and moves forward to the next uncommitted *P0 element*, with each *P0 element* committed by a *Commit element*. The Commit Window Move element moves the start of the Commit window by a number of *P0 elements*, to allow a *Commit element* to commit *P0 elements* that are younger than the oldest uncommitted *P0 elements*, leaving these older *P0 elements* uncommitted.

Completer An agent in a computing system that responds to and completes a memory transaction that was initiated by a Requester.

See also Requester.

Constructive instruction encoding A constructive instruction encoding is an instruction encoding where the destination register is encoded independently of the source registers.

Condition code check The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of PSTATE.IT determines whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Condition flags The N, Z, C, and V bits of PSTATE, an SPSR, or FPSCR. See the register descriptions for more information.

See also Condition code check, PSTATE.

Conditional execution When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`.

See also Condition code check.

CONSTRAINED UNPREDICTABLE Where an instruction can result in UNPREDICTABLE behavior, the Armv8 and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Execution at Non-secure EL1 or EL0 of an instruction that is CONSTRAINED UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.

Containable error See Contained error.

Contained error An error that is not uncontained or uncontainable.

Context switch The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

Context Synchronization event One of:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Exception entry, if any of the following is true:
  - FEAT_ExS is not implemented.
  - FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
  - The exception is taken to AArch32.
- Return from an exception, if any of the following is true:
  - FEAT_ExS is not implemented.
  - FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
  - The exception is returning to AArch32.
  - Exit from Debug state.
- Executing a `DCPS` instruction.
- Executing a `DRPS` instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- The following invalidations, if completed before the Context synchronization event will affect all instructions that appear in program order after the Context synchronization event:
  - TLBs.
  - Instruction caches.
  - In AArch32 state branch predictors.
- In AArch32 state, all Non-cacheable writes that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- Changes to the Debug external authentication interfaces that are made before the Context synchronization event affect any instruction that appears in program order after the instruction causing the Context synchronization event.
- The effect of the completion of any of the instructions added by FEAT_SPECRES is synchronized to the current execution context.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Restrictions on the effects of speculation (as described in Restrictions on the effects of speculation) are observed.
- Ensuring that the `TSB` instruction is executed in the necessary order with respect to other instructions.
- Profiling operations for all instructions that are executed in program order are synchronized by execution of a `PSB` instruction before the Context synchronization event.
- See also:
  - Restrictions on the effects of speculation.
  - Trace Synchronization Barrier.
  - Synchronization and debug exceptions.
  - Synchronization in self-hosted trace.
  - Synchronization of register updates.
  - Speculation in the trace element stream.
  - Trace Synchronization event.
  - Self-hosted trace extension synchronization rules.
  - Execution, data prediction and prefetching restriction System instructions.
  - Synchronization and Statistical Profiling.
  - Synchronization and Halting debug events.
  - Synchronization of memory-mapped registers.
  - CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization.

Note

- The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively, except that the performance monitor counters are permitted to reveal such speculation.
- *Context synchronization events* were previously described as *context synchronization operations*.

Conventional memory Memory locations from which generic OSs and application runtimes expect to create allocations for general software use.

Core See Processing element (PE).

CoreSight Address Translation Unit A form of System MMU for trace streams.

Corrected error An error that is detected by hardware and that hardware has corrected.

Counter resolution See Counter resolution.

CPBM Cache-Portion Bit Map.

CSU Cache-Storage Usage.

Data independent timing (DIT) The time that it takes to execute a piece of code where the time is not a function of the data being operated on. For more information, see About PSTATE.DIT and About the DIT bit.

Debugger In most of this manual, *debugger* refers to any agent that is performing debug. However, some chapters or parts of this manual require a more rigorous definition, and define debugger locally. See:

- Definition of a debugger in the context of self-hosted debug.
- Definition of a debugger in the context of self-hosted debug.
- Definition and constraints of a debugger in the context of external debug.

Deferred error An error that has not been silently propagated but does not require immediate action at the producer. The error might have passed from the producer to a consumer.

Deprecated Something that is present in the Arm architecture for backward compatibility.

Features that are deprecated but are not OPTIONAL are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm strongly recommends software to avoid using deprecated features. Where a feature is deprecated for performance reasons, Arm strongly recommends software to avoid using that feature because of the likelihood of the significant negative impact on performance.

See also OPTIONAL.

Destructive instruction encoding A destructive instruction encoding is an instruction encoding where one of the source registers is also used as the destination register.

Detected error An error that has been detected and signaled to a consumer.

Detected Uncorrected error A detected error that has not been be corrected and causes failure.

Digital signal processing (DSP) Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct branch Branch instructions whose target is relative to the PC of the branch and the offset is determined solely from the opcode of the executed instruction.

See also Indirect branch.

Direct Memory Access (DMA) An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

Direct read A direct read of a System register is a read performed by a System register access instruction in which the contents of the System Register addressed by {`op1`, `Crm`, `Crn`, `op2`}

For more information, see Direct read.

See also Direct write, Indirect read, Indirect write.

Direct write A direct write of a System register is a write performed by a System register access instruction.

For more information, see Direct write.

See also Direct read, Indirect read, Indirect write.

DMA See Direct Memory Access (DMA).

DNM See Do-Not-Modify (DNM).

Domain In the Arm architecture, *domain* is used in the following contexts.

Shareability domain Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Memory regions domain

When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the *Domain Access Control Register* (DACR). Arm deprecates any use of memory regions domains.

Do-Not-Modify (DNM) Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

Double-precision value Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned Means that the address is divisible by 8.

Downstream The direction from Requesters to Completers.

DSB Data Synchronization Barrier.

DSP See Digital signal processing (DSP).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DUE See Detected Uncorrected error.

DUE FIT RATE The FIT rate for failures from a DUE.

ECC See Error Correction Code (ECC).

EDAC See Error Correction Code (ECC).

EDC See Error Detection Code (EDC).

Effective frequency See Effective frequency.

Effective value A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

- In some cases, the description of a control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the *Effective value* of register control field *b*. For example, when the value of HCR.DC is 1, the *Effective value* of HCR.VM is 1, regardless of its actual value.
  In other cases, in some contexts a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the *Effective value* of register control field *b*.

  Note

  Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an *Effective value* in that earlier version of the architecture.

- Otherwise, the *Effective value* of a register control field is the value of that field.

Effective Non-streaming SVE vector length The Non-streaming SVE vector length in bits at the current Exception level, is an implementation-supported power of two up to the Maximum implemented Non-streaming SVE vector length, further constrained by ZCR_ELx at the current and higher Exception levels.

See also Configurable SVE vector lengths.

Effective Streaming SVE vector length The Streaming SVE vector length in bits at the current Exception level is an implementation-supported power of two from 128 up to the Maximum implemented Streaming SVE vector length, further constrained by SMCR_ELx at the current and higher Exception levels.

See also Configurable SVE vector lengths.

Effective SVE vector length The vector length in bits that applies to the execution of SVE instructions at the current Exception level is the Effective Streaming SVE vector length when the PE is in Streaming SVE mode, otherwise it is the Effective Non-streaming SVE vector length.

See also Configurable SVE vector lengths.

Element number For a given element size of N bits, elements within a vector or predicate register are numbered with element[0] always representing bits[(N-1):0], element[1] always representing bits[(2N-1):N], and so on.

Endianness An aspect of the system memory mapping.

See also Big-endian memory and Little-endian memory.

Error Correction Code (ECC) A code capable of detecting and correcting a number of errors.

Error Detection Code (EDC) A code capable of detecting, but not correcting, errors.

Error propagation Passing an error from a producer to a consumer.

Error record Data recorded about an error, usually by hardware.

ETEEvent A feature of the trace unit that is used to generate *Event elements* and drive External Outputs. Each ETEEvent can be programmed to be sensitive to resource events.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Event trace The trace uses *Event elements* that indicate certain events have occurred in the program that the PE is executing. The program events to be indicated are selected before a trace session.

Exceptional occurrence Events indicated by an *Exception element* by the ETE architecture, including the following:

- PE architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

Exception Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Exception vector A fixed address that contains the address of the first instruction of the corresponding exception handler.

Executing processing element (PEe) The name given to an example PE that executes a specified instruction.

See also Processing element (PE).

Execution stream The stream of instructions that would have been executed by sequential execution of the program.

External abort An abort that is generated by the external memory system.

Fault An event that causes the program to deviate from the intended behavior, for example an invalid translation.

Field Replaceable Unit (FRU) A component or unit in a system that can be replaced without return to base.

First active element The First active element of an SVE vector or predicate register is the lowest numbered element that is an Active element.

First-fault load SVE provides a First-fault option for some SVE vector load instructions. This option causes memory access faults to be suppressed if they do not occur as a result of the First active element of the vector. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.

Flat address mapping Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

FRU See Field Replaceable Unit (FRU).

Gather-load Gather-load is a mechanism that allows the elements of a vector to be read from non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

General-purpose registers The registers that the base instructions use for processing:

- In AArch64 state the general-purpose registers are:
  - W0-W30 when accessed as 32-bit registers.
  - X0-X30 when accessed as 64-bit registers.
- In AArch32 state the general-purpose registers are R0-R14, that can also be described as R0-R12, SP, LR.

  Note

  Older documentation defines the AArch32 general-purpose registers as R0-R12, and the Arm core registers as R0-R12, SP, LR, and PC.

See also High registers, Low registers.

Generic Interrupt Controller (GIC) Arm system architecture interrupt controller for IRQ and FIQ interrupt exceptions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Governing predicate The predicate register that is used to determine the Active elements of a predicated instruction is known as the Governing predicate for that instruction.

GIC See Generic Interrupt Controller (GIC).

GPC See Granule Protection Check.

Granule Protection Check A mechanism for checking the protection information of a particular physical address and physical address space, For more information, see The Granule Protection Check Mechanism.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned Means that the address is divisible by 2.

High registers In AArch32 state, the general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

Note

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

See also General-purpose registers, Low registers.

High vectors An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

ICN InterConnect Network.

IGNORED Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware.

In body text, the term IGNORED is shown in SMALL CAPITALS.

Illegal In the context of SME, describes an implemented instruction whose attempted execution by a PE when `PSTATE.SM` and `PSTATE.ZA` are not in the required state causes an SME illegal instruction exception to be taken, unless its execution at the current Exception level is prevented by a higher priority configurable trap or enable.

See Streaming SVE mode.

Immediate and offset fields Are unsigned unless otherwise stated.

Immediate value A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many instructions can be used with an immediate argument.

IMP An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

IMPLEMENTATION SPECIFIC Means that the behavior is not architecturally defined, and might not be documented by an individual implementation. Used when there are several implementation options available and the option chosen does not affect software compatibility.

In body text, the term IMPLEMENTATION SPECIFIC is shown in SMALL CAPITALS.

Imprecise exception An exception that is not precise.

Inactive element An Inactive element is an SVE vector element or predicate element that is an unused source register element or destination register element for the associated instruction. When the corresponding element of an instruction's Governing predicate is FALSE, it is an Inactive element.

Index register A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indirect branch Branch instructions that are not direct branches.

See also Direct branch.

Indirect read An Indirect System Register Read is a System Register Read which is not a Direct System Register Read Effect. For example, when an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register.

For more information, including additional examples of indirect reads, see Indirect read.

See also Direct read, Direct write, Indirect write.

Indirect write An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a Direct write to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent.

For more information, see Indirect write.

See also Direct read, Direct write, Indirect read.

Inline literals These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

Intermediate physical address (IPA) An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical address (PA), Virtual address (VA).

Interrupt In a PE context, an asynchronous exception. There are three interrupt exceptions: IRQ, FIQ, and SError. IRQ and FIQ are always precise. In a system architecture context, an asynchronous event sent to a PE or GIC for processing as an interrupt exception.

Interworking A method of working that permits branches between software using the T32 and A32 instruction sets.

IPA See Intermediate physical address (IPA).

kvm Kernel-based Virtual Machine, an open-source software package that implements a type-2 hypervisor within Linux.

Latent error An error that is present in a system but not yet detected.

Last active element The Last active element of an SVE vector or predicate register is the highest numbered element that is an Active element.

Legal Describes an implemented instruction that can be executed by a PE when `PSTATE.SM` and `PSTATE.ZA` are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

See Streaming SVE mode.

Level See Cache level.

Level of Coherence (LoC) The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of coherency (PoC).

Level of Unification, Inner Shareable (LoUIS) The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable Shareability domain. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of unification (PoU).

Level of Unification, uniprocessor (LoUU) For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see Terms used in describing the cache maintenance instructions.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also Cache level, Point of unification (PoU).

Line See Cache line.

Little-endian memory Means that, for example:

- A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory, Endianness.

Load/store architecture An architecture where data-processing operations only operate on register contents, not directly on memory contents.

LoC See Level of Coherence (LoC).

LoUIS See Level of Unification, Inner Shareable (LoUIS).

LoUU See Level of Unification, uniprocessor (LoUU).

Lockdown See Cache lockdown.

Low registers In AArch32 state, general-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

See also General-purpose registers, High registers.

LPI Locality-specific Peripheral Interrupt.

Maximum implemented Non-streaming SVE vector length The maximum Non-streaming SVE vector length in bits supported by the implementation.

See Maximum implemented SVE vector lengths

Maximum implemented Streaming SVE vector length The maximum Streaming SVE vector length in bits supported by the implementation.

See Maximum implemented SVE vector lengths

Maximum implemented SVE vector length The maximum vector length in bits that applies to the execution of SVE instructions at the current Exception level is the Maximum implemented Streaming SVE vector length when the PE is in Streaming SVE mode; otherwise it is the Maximum implemented Non-streaming SVE vector length.

See Maximum implemented SVE vector lengths.

MEC See Memory Encryption Contexts.

MBWU Memory BandWidth Usage.

Memory barrier See Memory barriers.

Memory coherency The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

Memory element An item of data in memory that is transferred to or from a vector element by an SVE load or store instruction. Each memory element has an access size and a type. The memory element access size is specified by each load and store instruction independently of the vector element size.

Memory Encryption Contexts Unique cryptographic boundaries provided to the Realm physical address space for assignment to Realm virtual machines, with policy controlled by Realm EL2. For more information, see Memory Encryption Contexts.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Memory Management Unit (MMU) Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

Memory Protection Unit (MPU) A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

Merging predication When a predicated SVE instruction specifies merging predication, the Inactive elements of the destination register remain unchanged.

Memory system component MSC. A function, unit, or design block in a memory system that can have partitionable resources. MSCs consist of all units that handle load or store requests issued by any MPAM Requester. These can include cache memories, interconnects, memory management units, memory channel controllers, queues, buffers, and rate adapters. An MSC can contain one or more resources that each can have zero or more resource partitioning controls. For example, a PE can contain several caches, each of which might have zero or more resource partitioning controls.

Memory system resource A resource that affects the performance of software's use of the memory system and is either local to an MSC (such as cache-memory capacity) or non-local (such as memory bandwidth, which is present over an entire path, from Requester to Completer, that can pass through multiple MSCs).

MMR Memory-mapped Register.

MPAM Memory system resource Partitioning and Monitoring.

MPAM information The MPAM information bundle, comprising PARTID, PMG, and MPAM_NS.

MPAM resource partition See Resource partition.

Microarchitecturally-finished An operation that has finished all of its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.

Microarchitecturally-unfinished An operation that has not completed all of its operational pseudocode.

Miss See Cache miss.

MMU See Memory Management Unit (MMU).

MPU See Memory Protection Unit (MPU).

Multi-copy atomicity The form of atomicity described in Requirements for multi-copy atomicity.

See also Atomicity, Single-copy atomicity.

NaN Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *Quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the *IEEE Standard for Floating-point Arithmetic*.

See also Quiet NaN, Signaling NaN.

Natural eviction A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.

Non-fault load SVE provides a Non-fault option for some SVE vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.

NSVL See Effective Non-streaming SVE vector length.

Observer An observer is one of:

- The mechanism inside a PE that generates Implicit Translation Table Descriptor (TTD) Memory Effects.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The mechanism inside a PE that generates Implicit Instruction Memory Read Effects.
- The mechanism inside a PE that generates Explicit Memory Effects, Tag Memory Effects, Barrier Effects, Instruction Fetch Barrier Effects, TLBI Effects, DC Effects, IC Effects, Fault Effects, or GCS Effects.
- The mechanism inside a PE that generates Memory Effects for SPE.
- The mechanism inside a PE that generates Memory Effects for TRBE.
- The mechanism inside a PE that generates Memory Effects for HDBSS.
- The mechanism inside a PE that generates Memory Effects for HACDBS.
- Any mechanism outside the PE that can generate Memory Effects, for example, a peripheral device, an SMMU, or a GIC.

Obsolete Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

Offset addressing Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

OPTIONAL When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a features to be included in a new implementation only if there is a known backward-compatibility reason for the inclusion of the feature.
  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.
- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

Note

Do not confuse these Arm-specific uses of OPTIONAL with other uses of *optional*, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior determined by an implementation choice, for example the optional byte order reversal in an Armv7-R implementation, where the SCTLR.IE bit indicates the implemented option.

See also Deprecated.

Other-multi-copy atomic This is a Memory Write effect from an observer that, if observed by a different observer, is then observed by all other observers that access the location coherently.

PA See Physical address (PA).

Packed access A memory access that is performed as a result of an SVE load or store instruction for which the vector element size and the memory element size are the same.

PCIe See Peripheral Component Interconnect Express (PCI Express or PCIe).

PE See Processing element (PE).

Peripheral Component Interconnect Express (PCI Express or PCIe) A high-speed serial computer expansion bus standard maintained and developed by the PCI Special Interest Group.

PEe The executing processing element. See Executing processing element.

PFA See Predictive Failure Analysis (PFA).

Physical address (PA) An address that identifies a location in the physical memory map.

See also Intermediate physical address (IPA), Virtual address (VA).

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Point of coherency (PoC) See Terms used in describing the cache maintenance instructions for the AArch64 state and Terminology for Clean, Invalidate, and Clean and Invalidate instructions for the AArch32 state.

Point of Deep Persistence (PoDP) See Terms used in describing the cache maintenance instructions.

Point of encryption (PoE) See Terms used in describing the cache maintenance instructions.

Point of persistence (PoP) See Terms used in describing the cache maintenance instructions.

Point of physical aliasing (PoPA) See Terms used in describing the cache maintenance instructions.

Point of physical storage (PoPS) See Terms used in describing the cache maintenance instructions.

Point of unification (PoU) See Terms used in describing the cache maintenance instructions for the AArch64 state and Terminology for Clean, Invalidate, and Clean and Invalidate instructions for the AArch32 state.

Poison atomicity The form of atomicity described in Clearing Deferred errors from poisoned Locations.

See also Atomicity.

Poison granule A region of naturally-aligned memory affected by poison.

See Error propagation for information on poison.

Post-indexed addressing Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Precise exception An exception where the exception handler receives the state of the PE and the state of the memory system consistent with the PE having executed all of the instructions up to, but not including, the point in the instruction stream where the exception was taken. The state of the PE and the state of the memory do not include instructions that occurred after this point.

Predicate A one-dimensional array of predicate elements of the same size.

Predicate element Individual subdivisions of a predicate register that can be 1, 2, 4, or 8 bits in size. The predicate element size is specified independently by each instruction and is always one-eighth the size of the corresponding vector element. The lowest-numbered bit of each predicate element holds the Boolean value of that element, where 1 represents TRUE and 0 represents FALSE.

Predicated instruction An SVE instruction that has a Governing predicate operand, which determines the Active and Inactive elements for that instruction.

Predicate register An SVE predicate register, P0-P15, having a length that is a power of two, in the range 16 bits to 256 bits, inclusive.

Predictive Failure Analysis (PFA) Mechanisms to analyze errors and predict future failures.

Prefetching Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Note

The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

See also Simple sequential execution.

Prefixed instruction The instruction that immediately follows a MOVPRFX instruction in program order.

Pre-indexed addressing Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Processing element (PE) The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Protection granule A quantum of memory for which an EDC or ECC provides detection or correction. For example, a 72/64 SECDED ECC scheme has a 64-bit protection granule.

Protection region A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit See Memory Protection Unit (MPU).

Pseudo-instruction UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>, <Rm>, #<n>`.

PSTATE An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

See also Condition flags.

Quadword A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

Quadword-aligned Means that the address is divisible by 16.

Quiet NaN A NaN that propagates unchanged through most floating-point operations.

See also NaN, Signaling NaN.

RAO See Read-As-One (RAO).

RAZ See Read-As-Zero (RAZ).

RAO/SBOP In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes.

From the introduction of the Armv8 architecture, RES1 replaces this description.

See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).

RAO/WI Read-As-One, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ/SBZP In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

From the introduction of the Armv8 architecture, RES0 replaces this description.

See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

RAZ/WI Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-As-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-allocate cache A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

Read-As-One (RAO) Hardware must implement the field as reading as all 1s.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also RAO/SBOP, RAO/WI, RES1.

Read-As-Zero (RAZ) Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBZP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also RAZ/SBZP, RAZ/WI, RES0.

Read-to-Clear (RC) A read of the field clears the field to 0 once the read is complete.

Read, modify, write In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

Recoverable error A contained error that must be corrected to allow the correct operation of the system or smaller parts of the system to continue.

Reliability Continuity of correct service.

Requester An agent in a computing system that is capable of initiating memory transactions.

See also Completer.

RES0 A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES0 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is Architecturally mapped to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

The definition of a RES0 field in an architecturally-defined data structure that is held in memory is:

- A read of a RES0 bit returns the value last written.
- The value of a RES0 bit has no effect on the operation of the PE or any other hardware implementing the architecture such as the GIC or the SMMU, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.
- For an implicit memory write effect to the field by a PE or for a write to the field by any other hardware implementing the architecture such as the GIC or the SMMU, each bit defined as RES0 is set to 0.
- For a write to the data structure by a PE, or any other hardware implementing the architecture such as the GIC or the SMMU, if the write is specified to only update specific fields, the

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

hardware preserves the value of other RES0 bits in the same data structure. This applies, for example, to Hardware updates to the translation tables.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES0H, RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES0H

A reserved bit or field with SBZP. This behavior uses the hardwired to 0 subset of the RES0 definition.

See also Read-As-Zero (RAZ), RES0, RES1, Should-Be-Zero-or-Preserved (SBZP).

RES1

A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES1 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is Architecturally mapped to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

The definition of a RES1 field in an architecturally-defined data structure that is held in memory is:

- A read of a RES1 bit returns the value last written.
- The value of a RES1 bit has no effect on the operation of the PE or any other hardware implementing the architecture such as the GIC or the SMMU, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.
- For an implicit memory write effect to the field by a PE or for a write to the field by any other hardware implementing the architecture such as the GIC or the SMMU, each bit defined as RES1 is set to 1.
- For a write to the data structure by a PE, or any other hardware implementing the architecture such as the GIC or the SMMU, if the write is specified to only update specific fields, the hardware preserves the value of other RES1 bits in the same data structure. This applies, for example, to Hardware updates to the translation tables.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, RES0H, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions described as reserved are:

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— In an RW or WO register, RES0.
— In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES0H, RES1, UNDEFINED, UNK, UNPREDICTABLE.

Resource partition The collection of MPAM resource control settings associated with a software environment and identified by the combination of a physical PARTID space and a partition number.

RESS Reserved, Sign extended. A register value is extended by copying the sign bit into all of the reserved bits to the left of the most significant bit of the field. The values of these bits are identical to the most significant bit of the value being extended.

Within the architecture, a register bit or field can be treated:

- As RESS in few defined architectural contexts.
- In a different defined behavior in other architectural contexts.

This means the definition of RESS for fields in read/write registers is:

If a field [Q:P] is RESS in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired as a sign-extension of bit [P-1]. In this case:
   - Reads of the bit always return the value of bit [P-1].
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to the same value as bit [P-1].
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RESS bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a field [Q:P] is RESS only in some contexts

For a bit in a read/write register, when the bit is described as RESS:

- An indirect write to the register sets the bit to the same value as bit [P-1].
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- While the use of the register is such that the bit is described as RESS, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RESS description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RESS indicates that the bit reads as the value of bit [P-1], but software must treat the bit as UNKNOWN.
- For a write-only bit, RESS indicates that software must treat the bit as SBO or SBZ as necessary.

A bit that is RESS in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as [P-1].
- Must use a SBOP or SBZP policy to write to the bit.

This RESS description can apply to a single bit, or to a field for which each bit of the field must be treated as RESS.

Rewind point A rewind point is a point in the program flow to which execution can return if all subsequent execution is found to have been incorrectly speculatively executed.

RISC Reduced Instruction Set Computer.

RIS Resource instance selection. The value in MPAMCFG_PART_SEL.RIS selects the resource instance that is configured through MPAMCFG_* registers and described by the MPAMF ID registers.

RME Realm Management Extension.

Rounding error The value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding mode Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*, see Floating-point standards, and terminology.

Saturated arithmetic Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.

SBO See Should-Be-One (SBO).

SBOP See Should-Be-One-or-Preserved (SBOP).

SBZ See Should-Be-Zero (SBZ).

SBZP See Should-Be-Zero-or-Preserved (SBZP).

Scalable vector register An SVE scalable vector register, Z0-Z31, having a length that is a power of two, in the range 128 bits to 2048 bits, inclusive.

Scalar base register A scalar base register refers to an AArch64 general-purpose register, X0-X30, or the current stack pointer, SP.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Scalar index register A scalar index register refers to an AArch64 general-purpose register, X0-X30, or for certain instructions, XZR.

Scalable Matrix Extension Defines architectural state capable of holding two-dimensional matrix tiles, and a Streaming SVE mode which supports execution of SVE2 instructions with a vector length that matches the tile width, along with instructions that accumulate the outer product of two vectors into a tile, as well as load, store, and move instructions that transfer a vector to or from a tile row or column.

See The Scalable Matrix Extension.

Scalable Matrix Extension version 2 Extends the Scalable Matrix Extension by adding data-processing instructions with multi-vector operands and amulti-vector predication mechanism, a lookup table feature, a binary outer product instruction, and a range prefetch hint.

See SME2 Multi-vector operands, Vector predication, and SME2 ZT0 register.

Scatter-store Scatter-store is a mechanism that allows the elements of a vector to be written to non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

SDC See Silent Data Corruption (SDC).

SDC FIT rate The FIT rate for failures because of SDC.

SDEC Single device error correction EDAC. This can detect and correct multiple clustered errors in a protection granule, such as the types of errors that might be seen if a protection granule is striped across multiple devices and multiple errors come from a single device.

SECDED Single error correct, double error detect EDAC. This can detect a single or double bit error and correct a single bit error in a protection granule.

SED Single error detect EDC. This can detect a single bit error in a protection granule.

Security hole A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Self-modifying code Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see Caches and memory hierarchy.

SError exception An asynchronous exception in the Arm architecture.

Serviceability The ability to undergo modifications and repairs.

Service failure mode A mode entered to reduce the severity of an error.

Set See Cache sets.

Should-Be-One (SBO) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One-or-Preserved (SBOP) From the introduction of the Armv8 architecture, the description Should-Be-One-or-Preserved (SBOP) is superseded by RES1.

Hardware must ignore writes to the field.

When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero (SBZ) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero-or-Preserved (SBZP) From the introduction of the Armv8 architecture, the description Should-Be-Zero-or-Preserved (SBZP) is superseded by RES0.

Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Signaling NaN An Invalid Operation floating-point exception occurs whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

See also NaN, Quiet NaN.

Signed immediate and offset fields Are encoded in two's complement notation unless otherwise stated.

Silent Data Corruption (SDC) An error that is not detected by hardware or software.

SIMD Single-Instruction, Multiple-Data.

The SIMD instructions in AArch32 state are:

- The instructions summarized in Parallel addition and subtraction instructions.
- The Advanced SIMD instructions summarized in Advanced SIMD and floating-point instructions, when operating on vectors.

Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Single-copy atomicity The form of atomicity described in Properties of single-copy atomic accesses.

See also Atomicity, Multi-copy atomicity.

Single-precision value A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

SMCU See Streaming Mode Compute Unit.

SME See Scalable Matrix Extension.

SME2 See Scalable Matrix Extension version 2.

SMMU System Memory Management Unit.

SPE Statistical Profiling Extension.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Spatial locality The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

Special-purpose register One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization:

- Special purpose registers specifies the AArch64 Special-purpose registers.
- AArch32 general-purpose registers, the PC, and the Special-purpose registers lists the AArch32 Special-purpose registers.

Speculative Speculative operations are:

- Operations that are generated by instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Operations that are generated by conditional instructions for which the conditions for the instruction have not been architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after the point at which a precise asynchronous exception will be taken.
- Operations generated by instructions for which one or more of the arguments come from a register that has not been architecturally resolved.
- Operations generated by the hardware that are not directly generated by any instructions appearing in the Execution stream.
- Memory effects (M2) generated by load or store instructions (LS2) appearing in program order after load or store instructions (LS1) that generate memory effects (M1) where all of the following apply:
  - M1 is locally-ordered-before M2.
  - LS1 has not been executed before LS2.
- Read accesses generated for a translation table walk for which the granule protection check for the address being accessed has not been architecturally resolved.

See also Execution stream.

SPI Shared Peripheral Interrupt.

Streaming execution Execution of instructions by a PE when that PE is in Streaming SVE mode.

See Streaming SVE mode.

Streaming Mode Compute Unit Where more than one PE shares resources for Streaming execution of SVE and SME instructions, those shared resources are called a Streaming Mode Compute Unit (SMCU).

See Streaming execution priority.

Streaming SVE mode An execution mode that supports a substantial subset of the SVE2 instruction set and architectural state with a vector length that matches the width of SME tiles, which may be different from the vector length when the PE is not in Streaming SVE.

See Streaming SVE mode.

Streaming SVE register state The registers *Z0-Z31*, *P0-P15*, and *FFR* that are accessed by SVE and SME instructions when the PE is in Streaming SVE mode.

See Streaming SVE mode.

Superpriority An attribute that is used to denote virtual and physical IRQ and FIQ interrupts as non-maskable. IRQ and FIQ interrupts with Superpriority can be taken under certain conditions where usually they would be masked by the PSTATE.{I, F} bits.

SVL See Effective Streaming SVE vector length.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System Control Processor A PE dedicated to system control and monitoring.

T32 instruction One or two halfwords that specify an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in T32 state. T32 instructions must be halfword-aligned.

T32 instructions were previously called Thumb instructions.

See also A32 instruction, A64 instruction, T32 state.

T32 state The AArch32 Instruction set state in which the PE executes T32 instructions.

T32 state was previously called Thumb state.

See also A32 state, T32 instruction.

Temporal locality The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction See T32 instruction.

TLB See Translation Lookaside Buffer (TLB).

TLB lockdown A way to prevent specific translation table entries from being evicted from a TLB. This ensures that accesses to the associated regions of virtual memory never cause a translation table walk.

Translation context The context information used to uniquely identify a cached copy of a translation. See TLB maintenance.

Translation Lookaside Buffer (TLB) A microarchitectural structure containing the results of translation table walks. They help to reduce the average cost of a memory access.

Translation table A table held in memory that defines the properties of regions of virtual memory.

Translation table walk The process of doing a full translation table lookup. It is performed automatically by hardware.

Trap enable bits In VFPv2, VFPv3U, and VFPv4U, determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE.

Uncontainable error See Uncontained error.

Uncontained error An error that has been, or might have been, silently propagated.

Uncorrected error An error caused by a fault which was not removed.

UNDEFINED Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the UNDEFINED encoding was executed at EL0. This applies to:

- Any encoding that is not allocated to any instruction.
- Any encoding that is defined as never accessible at the current Exception level.
- Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

If the generated exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive. See the description of the Exception model for more information

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

See also Undefined Instruction exception.

Undetected error See Latent error.

Unified cache Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

UNK An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNK/SBOP Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

UNK/SBZP Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value can vary from moment to moment, and instruction to instruction, unless it has previously been assigned, other than at reset, to one of the following registers:

- Any of the general-purpose registers.
- Any of the Advanced SIMD and floating-point registers.
- Any of the Scalable Vector Extension registers.
- Any of the PSTATE N, Z, C, or V flags.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

Unpacked access A memory access that is performed as a result of a load or store instruction for which the vector element size is larger than the memory element size.

Unpredicated instruction An SVE instruction that does not have a Governing predicate operand and implicitly treats all other vector and predicate elements as Active.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UNPREDICTABLE Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

Unrecoverable error A contained error that is not recoverable. Continued correct operation is generally not possible. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part. Systems might use high-level recovery techniques to work around an unrecoverable yet contained error in a component so that the system recovers from the error.

Upstream The direction from Completers to Requesters.

VA See Virtual address (VA).

VFP In Armv7, an extension to the Arm architecture, that provides single-precision and double-precision floating-point arithmetic.

Vector A one-dimensional array of vector elements of the same size and data type.

Vector element Individual subdivisions of a vector register that can be 8, 16, 32, 64 or 128 bits in size. The vector element size and data type is specified independently by each instruction.

Virtual address (VA) An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also Intermediate physical address (IPA), Physical address (PA).

VL See Effective SVE vector length.

VM Virtual Machine.

VMM Virtual Machine Monitor. An alias for “hypervisor”.

Virtual PARTID A PARTID in MPAM that can be used by a virtual machine (VM). Virtual PARTIDs are mapped into physical PARTIDs using the virtual partition mapping entries in the MPAMVPM0 - MPAMVPM7 registers.

Watchpoint A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Way See Cache way.

WI Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

See also RAO/WI, RAZ/WI, RES0, RES1.

Word A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned Means that the address is divisible by 4.

Write-allocate cache A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back cache A cache in which when a cache hit occurs on a write access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in the next level of cache or memory. Any such data is written back to the next level of cache or memory when the cache line is cleaned. Another common term for a write-back cache is a *copy-back cache*.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Write-One-to-Clear (W1C) Writes of 0 to the bit are ignored. A write of 1 clears the bit to 0.

Write-One-to-Set (W1S) Writes of 0 to the bit are ignored. A write of 1 sets the bit to 1.

Write-Through cache A cache in which when a cache hit occurs on a write access, the data is written both to the cache and to the next level of cache or memory.

Write buffer A block of high-speed memory that optimizes stores to main memory.

ZA array A two-dimensional array of $[SVL_B \times SVL_B]$ bytes contained within the ZA storage.

See ZA storage.

ZA array vector A one-dimensional vector that is SVL bits in size within the ZA array.

See ZA array vector access.

ZA storage Architectural state added by SME that is capable of holding two-dimensional matrix tiles.

See ZA storage.

ZA tile A square, two-dimensional sub-array of elements within the ZA array.

See ZA tile access.

ZA tile slice A one-dimensional set of horizontally or vertically contiguous elements within a ZA tile.

See ZA tile access.

Zeroing predication When a predicated instruction specifies zeroing predication, the Inactive elements of the destination register are set to zero.

ZT0 register The 512-bit architectural register added by SME2 that consists of up to sixteen 32-bit table entries, each containing an 8-bit, 16-bit, or 32-bit element.

See SME2 ZT0 register.

Copyright © 2013-2026 Arm Limited or its affiliates. All rights reserved.
Non-Confidential